Microsoft

深入解析 SQL Server 2008

[美] Kalen Delaney　Paul S. Randal　Kimberly L. Tripp　Conor Cunningham　Adam Machanic　Ben Nevarez 著

陈宝国　李光杰　薛赛男　等　译

人民邮电出版社
POSTS & TELECOM PRESS

深入解析 SQL Server 2008

深层次展示核心引擎的功能及其工作原理

借助著名内部专家小组的指导，深入探究核心SQL Server引擎，并把这些知识运用在实际工作中。无论您是数据库开发人员、架构师，还是管理员，都能获得利用关键架构变更所需的深层知识，并挖掘产品的全部潜力。

深度揭示SQL Server的内部工作原理：

- 当SQL Server编译、扩展、压缩和移动数据库时，内部在进行什么操作
- 如何使用事件跟踪——从触发器到扩展事件引擎
- 为什么合适的索引能大大减少查询执行时间
- 如何用新的存储能力超越正常的行数限制
- 查询优化器是如何运行的
- 为有问题的查询计划排除故障的多种技术
- 何时强制SQL Server重用或创建新的缓存查询计划
- 运行DBCC时，SQL Server在内部检查什么
- 处理多个并发用户时，如何在5个隔离级别和2个并发模型中做出选择

关于作者

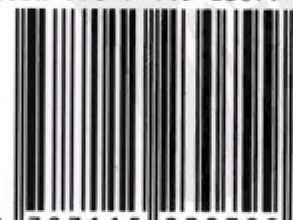
Kalen Delaney 自1993年起就是微软SQL Server的MVP，她为全世界的客户提供高级SQL Server培训。她是《SQL Server Magazine》的特约编辑和专栏作家，也是几本读者推崇的书籍的作者，包括《Inside Microsoft SQL Server 2005：The Storage Engine》和《Inside Microsoft SQL Server 2005: Query Tuning and Optimization》。

Paul S. Randal 微软MVP、培训师和TechNet Magazine的特约编辑。Kimberly L. Tripp是微软MVP、培训师和《SQL Server Magazine》的特约编辑。

Conor Cunningham 微软SQL Server Core Engine团队的首席架构师。

Adam Machanic MCITP、微软MVP、讲师，几本SQL Server书籍的合著者。

Ben Nevarez 从6.5版本开始使用SQL Server，是高级数据库管理员。



ISBN 978-7-115-23079-9
定价：99.00 元

封面设计：胡萍丽

分类建议：计算机 / 数据库
人民邮电出版社网址：www.ptpress.com.cn

深入解析 SQL Server 2008

Kalen Delaney　Paul S. Randal
[美] Kimberly L. Tripp　Conor Cunningham 著
Adam Machanic　Ben Nevarez

陈宝国　李光杰　薛赛男　等　译

人 民 邮 电 出 版 社
北 京

图书在版编目（C I P）数据

深入解析SQL Server 2008 /（美）德莱尼
(Delaney, K.）等著；陈宝国等译. — 北京：人民邮电
出版社，2010.7
ISBN 978-7-115-23079-9

Ⅰ. ①深… Ⅱ. ①德… ②陈… Ⅲ. ①关系数据库—
数据库管理系统，SQL Server 2008 Ⅳ. ①TP311.138

中国版本图书馆CIP数据核字(2010)第096143号

版权声明

Copyright © 2009 by Microsoft Corporation.
All rights reserved.
Original English language edition© Microsoft SQL Server 2008 Internals by Kalen Delaney,Paul S.Randal,Kimberly L.Tripp, Conor Cummingham, Adam Machanic, and Ben Nevarez.
Published by arrangement with the original publisher, Microsoft Corporation, Redmond, Washington, U.S.A.
本书原版由微软出版社出版。
本书简体中文版由微软出版社授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。
此版本权限在中华人民共和国境内（不包括中国香港、澳门特别行政区及中国台湾地区）销售发行。
版权所有，侵权必究。

深入解析 SQL Server 2008

◆ 著　[美] Kalen Delaney　Paul S.Randal
Kimberly L.Tripp　Conor Cunningham
Adam Machanic　Ben Nevarez
译　陈宝国　李光杰　薛赛男　等
责任编辑　刘　浩
◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街 14 号
邮编　100061　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京艺辉印刷有限公司印刷
◆ 开本：800×1000　1/16
印张：35.5
字数：967 千字　2010 年 7 月第 1 版
印数：1－3 000 册　2010 年 7 月北京第 1 次印刷
著作权合同登记号　图字：01-2009-5550 号
ISBN 978-7-115-23079-9

定价：99.00 元

读者服务热线：(010)67132705　印装质量热线：(010)67129223
反盗版热线：(010)67171154

内 容 提 要

本书全面探讨了 SQL Server 2008 的内部工作原理。全书共分为 11 章，首先在第 1 章中详细介绍了 SQL Server 2008 的架构和配置，然后在接下来的 10 个章节中深入探讨了 SQL Server 2008 数据存储和查询处理等各个方面的内部机制，包括：数据库和数据库文件、表、索引、跟踪、日志记录和恢复、特殊存储、查询优化、计划缓存和重新编译、事务和并发、DBCC 等。本书还有一个网站，上面有本书额外的第 12 章“查询执行”、本书的所有代码及其他工具和脚本。

本书由知识丰富的资深专家和数位具有多年产品使用经验的讲师联手打造，是一本关于 SQL Server 工作原理的权威参考指南，不管您是数据库开发人员、架构师，还是数据库管理员，都可以从阅读本书中获益。

致Dan：我永远的……

—Kalen

序

创造像 Microsoft SQL Server 这样产品的开发人员通常都是某一技术方面（如访问方法或查询执行）的专家。他们一直在与这个产品打交道，深知自己扮演的角色，以至于陷入了“知识的泥潭”：他们虽然非常了解自己特定领域的细节，却很难通过描述他们的工作就能帮助客户有效地使用这款产品。

另一方面，根据产品创作图书的技术作家们会由外到内地运用产品。这些作家中的大部分人对他们所写产品了解得非常广，但是又有点浅，仅停留在表面。他们写出的书籍有价值，这些书通常插入许多截图，能帮助新用户或中级用户快速学习如何用该产品进行工作。

在知识内部和知识面之间有个空档，就是许多产品开发人员创造的好功能并没有被用户（尤其是中级和高级用户）充分发挥其全部潜力，这就是本书的切入点。正如那些早期的“揭秘 SQL Server”系列，本书也是 SQL Server 如何工作的参考书。Kalen Delaney 已经和 SQL Server 产品团队一起工作了 10 多年，和开发人员花了无数个小时来深入知识内部，再用一种令人难以置信的清晰形式阐述结果，使中级和高级用户可以充分使用 SQL Server 的功能。在本书中，Kalen，还有 4 位 SQL Server 专家共享了深入的内部知识。Conor Cunningham 和 Paul Randal 具有多年的 SQL Server 产品开发经验，他们每个人都是资深技术专家，同时又是天才的演讲员。Kimberly Tripp 和 Adam Machanic 一起探讨事情的内部工作原理，然后将这些结果分享给其他人。在有关 SQL Server 的活动中，Kimberly 和 Adam 都是座无虚席的演讲员。这个团队了解并整合 SQL Server 2008 的关键架构更改的详细状况，从而造就了这本新的综合性的 SQL Server 内部参考书。

您可以用立见分晓的试验来确定一本技术图书是否应当归为“不容置疑的参考书”这个类别。测试相对容易，但是对大家来说很难执行。非常简单，这个测试就是看有多少正在开发产品的开发人员的书架上有这本书并参考它。我敢保证 Kalen 创作的《Inside Microsoft SQL Server》的每个版本都能通过这项测试，本书也会通过这项测试。

Dave Campbell
Microsoft SQL Server
狂热的技术追随者

前　言

您现在拿的这本书是“Inside SQL Server”系列的后续，该系列包括《Inside SQL Server 6.5》、《Inside SQL Server 7》、《Inside SQL Server 2000》和《Inside SQL Server 2005》（共4卷）。Inside系列变得太分散，“揭秘（Inside）”这个词也被其他作者甚至出版商扭曲了。我需要一个更具指示性的标题来说明本书的真正内容。

本书讲述微软公司的旗舰关系数据库产品——SQL Server的工作原理。此外，我还会解释如何利用有关工作原理方面的知识来获得更好的产品性能，不过这只是顺便得到的，并不是目标。市场上有其他几十本书讲述SQL Server的优化和最佳实践，本书则帮您理解为什么某些优化实践是那样工作的，也帮您在作为开发人员、数据架构师或DBA继续使用SQL Server的过程中，确定自己的最佳实践。

本书的读者

本书是为想深入理解SQL Server内在工作原理的所有人撰写的。它的重点是核心SQL Server引擎，特别是查询处理器和存储引擎。我希望大家有一些使用SQL Server引擎和T-SQL的经验。您无需是这两方面的专家，但是如果渴望成为专家并愿意了解提交查询执行以后SQL Server实际上做了些什么，本书会有所帮助。

本系列不讨论客户端编程界面、异类查询、商业智能或复制。实际上，大部分高可用性的功能都没有讲，但是在我们讨论数据库属性设置时，从较高层次讨论了一些功能，如镜像。我不会深入到一些内部操作的细节（如安全性）中，因为这是很大的话题，应当单独用整本书来讲。

我希望您看到的是满上的那半杯，而不是空着的那半杯，也就是能从本书所讲内容受益，而对于那些没有包括的主题，希望您能在其他资源中找到相关信息。

本书的内容

本书提供SQL Server处理查询和管理数据过程中的详细信息。首先在第1章中概述SQL Server关系数据库系统架构，然后在后面10章中继续研究查询处理和数据存储的多个方面。各章安排如下。

- 第1章　SQL Server 2008架构和配置
- 第2章　更改跟踪、跟踪和扩展事件
- 第3章　数据库和数据库文件
- 第4章　日志记录和恢复
- 第5章　表
- 第6章　索引：内部和管理
- 第7章　特殊存储
- 第8章　查询优化器

- 第 9 章　计划缓存和重新编译
- 第 10 章　事务和并发性
- 第 11 章　DBCC 揭秘

第 12 章可从随附内容（在下一节中讲述）获得，它讨论阅读查询计划的详细信息。这一章叫“查询执行”，是我的前一本书《Inside SQL Server 2005: Query Tuning and Optimization》的一部分。因为本章的 99%对 SQL Server 2008 仍然适用，所以我就把它“原封不动”地包括进来作为附加参考。

随附内容

本书有一个随附网站，上面有本书使用的所有代码，按章排列。随附内容还包括一章我以前写的书，以及我的书《SQL Server 2000》中的一章（“History of SQL Server”）。该站点还提供了额外的脚本和工具，增强您对 SQL Server 内部的体验和理解。您可以从以下地址访问随附站点上的内容：http://www.SQLServerInternals.com/companion。

本书的支持

我们会尽全力保证本书和随附网站的内容精确。如果有更正或更改，都会添加到微软公司的知识库文章中。

微软出版社在以下站点为本书提供支持：

http://www.microsoft.com/learning/support/books

问题和意见

如果您对本书有什么意见、问题或想法，或者有通过访问以上网站都不能解决的问题，请通过电子邮件发送给微软出版社：

mspinput@microsoft.com

或者通过邮件发到：

Microsoft Press

Attn: Microsoft SQL Server 2008 Internals Editor

One Microsoft Way

Redmond, WA 98052-6399

注意上述地址不提供微软软件的产品支持。

致　　谢

一直以来，这样的著作都不是靠个人的力量能够完成的，这本书更是如此。和其他几位 SQL Server 专家一起编写本书是我的荣幸，我一个人是无法完成本书的。我感谢 Adam Machanic、Paul Randal、Conor Cunningham 和 Kimberly Tripp 协助我完成此书。除了我那些聪慧的合著者之外，本书的出炉也得益于许多其他人的帮助和鼓励。

首先要感谢的就是您——亲爱的读者。感谢所有阅读本书的人，谢谢那些花时间写信告诉我对于本书的想法及想了解 SQL Server 其他一方面的人。我希望我能详细回答每个问题。就算我不能一一回复，你们的投入我也会铭记于心。我要特别感谢我以前一本书《Inside SQL Server 2005: The Storage Engine》的读者 Ben Nevarez。我了解到 Ben Nevarez 是一位非常机敏的读者，他发现了许多不易察觉的错误和细微的不一致之处，他通过我的网站礼貌而简洁地报告给我。几十封电子邮件以后，我开始期待 Ben 的邮件，并且非常高兴终于有机会见到他。Ben 现在是我最有价值的技术评论员，我深深地感激他极为仔细地阅读每一章。

和过去一样，微软公司的 SQL Server 团队从来都是非常了不起的。虽然 Lubor Kollar 和 Sunil Agarwal 没有直接参与本书的大部分研究，但是他们给予了我精神上的支持，无论什么时候见到他们，总是会听到他们鼓励的话语。

Boris Baryshnikov、Kevin Farlee、Marcel van der Holst、Perter Byrne、Sangeetha Shekar、Robin Dhamankar、Artem Oks、Srini Achaarya 和 Ryan Stonecipher 与我见面并不厌其烦地回复我的电子邮件。Jerome Halmans、Joanna Omel、Nikunj Koolar、Tres London、Mike Purtell、Lin Chan 和 Dipti Sangani 在回复我的电子邮件时，也提供了有价值的技术见解和信息。我希望他们都知道我是多么珍惜得到的每份信息。

我也深深地感谢 SQL Server 产品支持团队的 Bob Ward、Bob Dorr 和 Keith Elmoer，不只是感谢他们回答我偶尔问的问题，还感谢他们通过白皮书、会议讲稿和知识库文章提供这么多关于 SQL Server 的信息。我非常感谢 Alan Brewer 和 Gail Erikson 的杰出工作，是他们和他们的用户培训小组把 *SQL Server 联机丛书*中的 SQL Server 文档整理在一起。

当然，我要多次感谢 Buck Woody。首先感谢他在用户培训小组中所做的工作，然后感谢他是 SQL Server 开发团队的成员，在我有无法回答的问题时，他总是会帮助我。他的讲稿和博客日志总是寓教于乐，他的慷慨和不倦的好精神真是一种激励。

我还要谢谢 Leona Lowry 和 Cheryl Walter，他们为我在和大多数 SQL Server 团队在同一栋楼内找到了办公地点。非常感谢他们对我的接待。

我衷心感谢所有的 SQL Server MVP，特别感谢 Erland Sommarskog。Erland 编写了第 5 章排序规则的内容，只是因为他认为有这个必要。还值得我特别提到的是 Tibor Karaszi 和 Roy Harvey，他们给了我无限的支持和鼓励。在本书写作过程中给我灵感的其他 MVP 有 Tony Rogerson、John Paul Cook、Steve Kass、Paul Nielsen、Hugo Kornelis、Tom Moreau 和 Linchi Shea。作为 SQL Server MVP 团队的一员，我感到非常荣幸。

我深深地感谢“SQL Server Internals”课程中的学生，不但感谢他们对 SQL Server 产品的热

情、我教给他们和与他们分享的东西，也感谢他们和我分享的一切。我学到的很多东西都是受好奇学生的问题启发得到的。我的一些学生，如 Cindy Gross 和 Lara Rubbelke 成了我的朋友（而且还成了微软公司的员工），不断激发我的创作灵感。

最重要的是我的家庭，他们一直为我的工作提供坚实的基础。我和 Dan 已经结婚 24 年了，他一直是我生命的指路灯。我的女儿 Melissa 和 3 个儿子 Brendan、Rickey 和 Connor 现在都长大了，他们都是真诚、活泼、富有爱心的人。有他们在我的生命中，我感觉很幸福。

Kalen Delaney

Paul Randal

多年来我一直渴望写一本 DBCC CHECKDB 的完全说明——至少是把我脑子里的东西都倒出来为别的事情腾出空间！当 Kalen 让我写本书的“一致性检查”部分时，我不禁欢呼雀跃，我为此诚挚地感谢 Kalen。我想特别感谢微软公司中的两位以及在那里和我一起工作过的许多人（在很多情况下，我仍将和他们一起工作）。第一位是 Ryan Stonecipher，他原是 SQL 产品支持的升级工程师，我在 2003 年末把他请过来和我一起做 DBCC，两个月后我成为团队经理时，他突然就完成了一个 10 万行的 DBCC 代码。我怎么也找不到像他这么有能力的人来接管我珍贵的 DBCC 了……第二位是 Bob Ward，他领导 SQL 产品支持团队，我刚进微软公司时和他就成为了好朋友。这么多年来，我们肯定合作过几百个损坏案例，但还没遇到过比他更有能力解决客户问题和改进 Microsoft SQL Server 的人。我还要感谢 Steve Lindell，他参与了 SQL Server 2000 原始联机一致性检查代码的编写，他花了很多时间耐心解释 SQL Server 在 1999 年是怎样工作的。最后，我要感谢我妻子 Kimberly，她、Katelyn 和 Kiera 是我生命中除 SQL Server 以外的挚爱。

Kimberly Tripp

首先我要感谢好友 Kalen 邀请我参加本书的写作，在各种场合一起工作之后（甚至 1996 年一起开过公司），最终将我们的想法和内容都放在一本深奥的技术书籍里真是太好了。讲到性能优化，索引是最关键的，没有比创建*正确的*索引更好的系统改进方法了，但是知道什么是正确的需要多个组件，有些组件只有在实际体验、测试和操作后才会知道。为此，我要感谢很多人（读者、学生、参会者和客户）——那些问问题的、给我演示有趣情况的、呆到很晚“调试”和/或解决问题的人。正是“想知道为什么是这样工作的”这个强烈愿望让我对这个产品一直很有兴趣，也总是让我想投入得更多，从而了解事实真相。为此，我感谢 SQL 团队的全体成员——那些我认识且共事多年的人，他们一直给我灵感，提出聪明和富有见解的意见。我要特别谢谢 SQL 团队中的人，他们耐心、快速、彻底地回答我一些关于事实真相、原理方面的问题，他们是 Conor Cunningham、Cesar Galindo-Legaria 和我早期在 SQL Server 时共事的 Dave Campbell、Nigel Ellis 和 Rande Blackman。需要特别提到 Gert E. R. Drapers，他这么多年来和我一起花了很多时间讨论、辩论和解决问题。还要感谢 Paul，我最好的朋友和丈夫，他之前也是 SQL 信息的专家。

Conor Cunningham

我要感谢 Bob Beauchemin 和 Milind Joshi 为审阅本书中我写的“查询优化器”中的技术所做的努力，我还要感谢 Kimberly Tripp 和 Paul Randal 在我写本章时给予我的鼓励和支持。最后，我要谢谢回答我许多技术问题的所有 SQL Server Query Processor 小组成员。

Adam Machanic

首先我要感谢 Kalen Delaney 带领我们使本书从概念变成现实。Kalen 做得非常好，让我们把精力集中在任务上，她还帮忙找出那些隐藏的信息片段，让本书变成一本好书。一些 Microsoft SQL Server 小组成员抽出时间来审阅我写的内容：Extended Events 小组的 Jerome Halmans 和 Fabricio Voznika，以及 Change Tracking 小组的 Mark Scurrell。我想感谢你们每个人，谢谢你们回答我的问题、改进我写的那章的质量。最后，我要谢谢 Kate 和 Aura——我的妻子和女儿，在截稿之前，我总是没有时间陪她们，一直呆在办公室，而她们都非常理解和支持我。

关于作者

Kalen Delaney

Kalen Delaney 使用 Microsoft SQL Server 长达 21 年，她为全世界的客户提供高级 SQL Server 培训。自 1992 年起，她一直是 SQL Server MVP（最有价值的专家），也几乎是从那时起开始撰写关于 SQL Server 的文章。Kalen 在几十个技术会议上发表过演讲，包括在美国举办的每届 PASS Community Summit（该组织于 1999 年成立）。Kalen 是 SQL Tuners（www.sqltuners.net）的合作伙伴和培训经理，这是一家总部设在美国西北部的 SQL Server 调优和托管服务公司。

Kalen 是《SQL Server Magazine》的特约编辑和专栏作家，也是微软出版社的 SQL Server 图书的作者或合著者，包括《Inside Microsoft SQL Server 7》、《Inside Microsoft SQL Server 2000》、《Inside Microsoft SQL Server 2005: The Storage Engine》和《Inside Microsoft SQL Server 2005: Query Tuning and Optimization》。Kalen 的博客是 www.sqlblog.com，她的个人网站是 www.SQLServerInternals.com。

Paul S. Randal

Paul 是 SQLskills.com 的总经理，该公司由他和他妻子 Kimberly L. Tripp 一起经营。他也是 SQL Server MVP，他是《TechNet Magazine》的特约编辑之一。Paul 于 1999 年加入微软公司，在此之前他在 DEC 工作了 5 年，从事 OpenVMS 文件系统方面的工作。他为 SQL Server 2000 写了各种 DBCC 命令，转到 SQL Server 团队做管理之前，他重新写了 SQL Server 2005 的所有 DBCC CHECKDB。在 SQL Server 2008 的开发期间，他负责整个存储引擎。

Paul 定期教授一些主题课程，例如数据库维护、高可用性、灾难恢复和 SQL Server 揭秘。他是 Tech·Ed 的顶级演讲人，也是 SQL Server Connections 会议的合作主席。去年 Paul 写了大量 SQL Server 2008 材料，包括白皮书和《TechNet Magazine》上的文章。Paul 的热门博客是 www.SQLskills.com/blogs/paul，可以通过 Paul@SQLskills.com 与他联系。

Kimberly L. Tripp

Kimberly 是 SQLskills.com 的董事长兼创办人，该公司是她 1995 年离开微软公司后创建的，她在该公司身兼多职，包括 SQL Server 团队的技术作家和 Microsoft University 的主题专家/培训师。她是 SQL Server MVP、微软区域总监和《SQL Server Magazine》的特约编辑。从 1990 年开始，Kimberly 就一直关注 SQL Server 可用性的各个方面，重点是性能调整和优化。

Kimberly 定期教授一些主题课程，例如数据库设计、性能优化、数据库维护和 SQL Server 揭秘。她是 Tech·Ed 会议和 PASS 社区峰会的顶级演讲人，和 Paul Randal 共同担任 SQL Server Connections 会议的合作主席。Kimberly 用过从 SQL Server 1.0 版本开始的所有版本。她撰写过无

数资料，包括在线内容和网络直播、白皮书和最近写的面向 DBA 的 Microsoft SQL 2008 JumpStart 培训课程。Kimberly 的热门博客是 www.SQLskills.com/blogs/kimberly，可以通过 Kimberly@SQLskills.com 与她联系。

Conor Cunningham

Conor Cunningham 是 SQL Server Core Engine Team 的首席架构师，他有十多年微软数据库引擎构建经验。他的专长是查询处理和查询优化，他也设计和/或实现了一些 SQL Server 中的查询处理功能。Conor 在查询优化领域有一些专利，他也写了很多关于查询处理的学术文章。Conor 在 "Conor vs. SQL" 上的博客地址是 http://blogs.msdn.com/conor_cunningham_msft/default.aspx。

Adam Machanic

Adam Machanic 是居住在波士顿的独立数据库咨询师、作家和讲师。他参与过几十个高可用性 OLTP 和大规模数据仓库应用程序的 SQL Server 实施，他也为几个数据密集型应用程序优化过数据访问层性能。Adam 为许多网站和杂志撰文，包括 SQLBlog、Simple Talk、Search SQL Server、《SQL Server Professional》、《CoDe》和《Visual System Journal》。他也与别人合著过几本 SQL Server 书，包括《Expert SQL Server 2005 Development》（Apress，2007）和《Inside SQL Server 2005: Query Tuning and Optimization》（Microsoft Press，2007）。Adam 定期在用户组、社区活动和会议上做演讲，内容包括各种与 SQL Server 和.NET 有关的主题。他是 SQL Server MVP、Microsoft Certified IT Professional（MCITP）和 INETA North American Speakers Bureau 的会员。

技术审稿人：Benjamin Nevarez

Ben Nevarez 在关系数据库方面具有 15 年经验，他从 SQL Server 6.5 开始使用 SQL Server。他具有计算机工程硕士学位，曾在几个技术会议上发表过演讲，包括 PASS 社区峰会。Ben 现在是 American International Group（AIG）的一名高级数据库管理员。工作之余，Ben 与妻子 Rocio 和 3 个儿子 David、Benjamin 和 Diego 在一起共享天伦之乐。

目　录

第 1 章　SQL Server 2008 架构和配置 ……… 1

1.1　SQL Server 版本 ……… 1
1.2　SQL Server 元数据 ……… 2
1.2.1　兼容性视图 ……… 2
1.2.2　目录视图 ……… 3
1.2.3　其他元数据 ……… 4
1.3　SQL Server 引擎组件 ……… 6
1.3.1　观察引擎行为 ……… 7
1.3.2　协议 ……… 8
1.3.3　关系引擎 ……… 9
1.3.4　存储引擎 ……… 10
1.4　SQLOS ……… 13
1.5　计划程序 ……… 14
1.5.1　SQL Server 工作线程 ……… 15
1.5.2　将计划程序绑定到 CPU 中 ……… 17
1.5.3　专用管理员连接（DAC） ……… 20
1.6　内存 ……… 21
1.6.1　缓冲池与数据缓存 ……… 21
1.6.2　访问内存中的数据页 ……… 21
1.6.3　管理数据缓存中的页面 ……… 22
1.6.4　可用缓冲区列表和惰性编写器 ……… 22
1.6.5　检查点 ……… 23
1.6.6　管理其他缓存中的内存 ……… 24
1.6.7　调节内存大小 ……… 25
1.6.8　调节缓冲池大小 ……… 26
1.7　服务器资源调控器 ……… 30
1.7.1　资源调控器概述 ……… 30
1.7.2　资源调控器控制 ……… 37
1.7.3　资源调控器元数据 ……… 38
1.8　SQL Server 2008 配置 ……… 39
1.8.1　使用 SQL Server 配置管理器 ……… 39
1.8.2　配置网络协议 ……… 39
1.8.3　默认的网络配置 ……… 39
1.8.4　管理服务 ……… 40
1.9　SQL Server 系统配置 ……… 41
1.9.1　操作系统配置 ……… 41
1.9.2　跟踪标记 ……… 43
1.10　服务器配置设置 ……… 43
1.11　小结 ……… 52

第 2 章　更改跟踪、跟踪和扩展事件 ……… 53

2.1　基础知识：触发器和事件通知 ……… 53
2.1.1　运行时触发器行为 ……… 53
2.2　更改跟踪 ……… 54
2.2.1　更改跟踪配置 ……… 54
2.2.2　更改跟踪的运行时行为 ……… 57
2.3　跟踪和事件探查 ……… 60
2.3.1　SQL 跟踪架构和术语 ……… 61
2.3.2　安全性和权限 ……… 62
2.3.3　Profiler 入门 ……… 63
2.3.4　服务器端跟踪和收集 ……… 70
2.4　扩展事件 ……… 78
2.4.1　XE 体系结构的组件 ……… 79
2.4.2　事件会话 ……… 86
2.4.3　扩展事件 DDL 和查询 ……… 88
2.5　小结 ……… 91

第 3 章　数据库和数据库文件 ……… 92

3.1　系统数据库 ……… 92
3.1.1　master ……… 93
3.1.2　model ……… 93
3.1.3　tempdb ……… 93
3.1.4　资源数据库 ……… 93
3.1.5　msdb ……… 94
3.2　样例数据库 ……… 94
3.2.1　AdventureWorks ……… 94
3.2.2　pubs ……… 95
3.2.3　Northwind ……… 95

3.3 数据库文件 ········ 95
3.4 创建数据库 ········ 97
3.4.1 CREATE DATABASE 例子 ········ 99
3.5 扩展或收缩数据库 ········ 99
3.5.1 自动文件扩展 ········ 100
3.5.2 手动文件扩展 ········ 100
3.5.3 快速文件初始化 ········ 100
3.5.4 自动收缩性 ········ 100
3.5.5 手动收缩 ········ 101
3.6 使用数据库文件组 ········ 102
3.6.1 默认文件组 ········ 102
3.6.2 FILEGROUP CREATION 例子 ········ 103
3.6.3 文件流文件组 ········ 104
3.7 修改数据库 ········ 105
3.7.1 ALTER DATABASE 例子 ········ 106
3.8 数据库剖析 ········ 107
3.8.1 空间分配 ········ 107
3.9 设置数据库选项 ········ 110
3.9.1 状态选项 ········ 112
3.9.2 游标选项 ········ 114
3.9.3 自动选项 ········ 115
3.9.4 SQL 选项 ········ 115
3.9.5 数据库恢复选项 ········ 116
3.9.6 其他数据库选项 ········ 117
3.10 数据库快照 ········ 117
3.10.1 创建数据库快照 ········ 118
3.10.2 数据库快照使用的空间 ········ 119
3.10.3 管理快照 ········ 120
3.11 tempdb 数据库 ········ 121
3.11.1 tempdb 中的对象 ········ 121
3.11.2 tempdb 中的优化 ········ 122
3.11.3 最佳实践 ········ 123
3.11.4 tempdb 空间监视 ········ 124
3.12 数据库安全性 ········ 124
3.12.1 数据库访问 ········ 125
3.12.2 管理数据库安全性 ········ 126
3.12.3 数据与架构 ········ 127
3.12.4 主体与架构 ········ 127
3.12.5 默认架构 ········ 128
3.13 移动或复制数据库 ········ 128
3.13.1 分离和重新附加数据库 ········ 128
3.13.2 备份和还原数据库 ········ 130
3.13.3 移动系统数据库 ········ 130
3.13.4 移动 master 数据库 ········ 131
3.14 兼容性级别 ········ 131
3.15 小结 ········ 132

第 4 章 日志记录和恢复 ········ 133

4.1 事务日志基础 ········ 133
4.1.1 恢复阶段 ········ 134
4.1.2 读日志 ········ 137
4.2 更改日志大小 ········ 137
4.2.1 虚拟日志文件 ········ 137
4.2.2 观察虚拟日志文件 ········ 138
4.2.3 自动截断虚拟日志文件 ········ 141
4.2.4 维护可恢复日志 ········ 142
4.2.5 自动压缩日志 ········ 144
4.2.6 日志文件大小 ········ 145
4.3 备份和还原数据库 ········ 145
4.3.1 备份类型 ········ 145
4.3.2 恢复模型 ········ 146
4.3.3 选择备份类型 ········ 149
4.3.4 还原数据库 ········ 150
4.4 小结 ········ 154

第 5 章 表 ········ 155

5.1 创建表 ········ 155
5.1.1 命名表和列 ········ 156
5.1.2 保留关键字 ········ 157
5.1.3 分隔标识符 ········ 157
5.1.4 命名约定 ········ 158
5.1.5 数据类型 ········ 158
5.1.6 关于 NULL ········ 178
5.2 用户定义数据类型 ········ 180
5.3 IDENTITY 属性 ········ 181
5.4 内部存储 ········ 184
5.4.1 sys.indexes 目录视图 ········ 185

5.4.2 数据存储元数据 ······ 186
5.4.3 数据页 ······ 189
5.4.4 检查数据页 ······ 190
5.4.5 数据行的结构 ······ 193
5.4.6 查找一个物理页面 ······ 195
5.4.7 固定长度行的存储 ······ 197
5.4.8 可变长度行的存储 ······ 199
5.4.9 日期和时间数据的存储 ······ 204
5.4.10 sql_variant 数据的存储 ······ 206
5.5 约束 ······ 209
5.5.1 约束名称和目录视图信息 ······ 210
5.5.2 视图和多行数据修改中出现的约束故障 ······ 211
5.6 修改表 ······ 212
5.6.1 更改数据类型 ······ 212
5.6.2 添加一个新列 ······ 213
5.6.3 添加、删除、禁用或启用约束 ······ 213
5.6.4 删除列 ······ 214
5.6.5 启用或禁用一个触发器 ······ 215
5.6.6 修改表的内部 ······ 215
5.7 堆修改内部 ······ 217
5.7.1 分配结构 ······ 217
5.7.2 插入行 ······ 218
5.7.3 删除行 ······ 219
5.7.4 更新行 ······ 221
5.8 小结 ······ 224

第 6 章 索引：内部和管理 ······ 225

6.1 概述 ······ 225
6.1.1 SQL Server 索引 B 树 ······ 226
6.2 分析索引的工具 ······ 228
6.2.1 使用 dm_db_index_physical_stats 动态管理视图 ······ 228
6.2.2 使用 DBCC ID ······ 231
6.3 理解索引结构 ······ 233
6.3.1 聚集键的依赖关系 ······ 234
6.3.2 非聚集索引 ······ 236
6.3.3 约束和索引 ······ 236
6.4 索引创建选项 ······ 237
6.4.1 IGNORE_DUP_KEY ······ 238
6.4.2 STATISTICS_NORECOMPUTE ······ 238
6.4.3 MAXDOP ······ 238
6.4.4 索引放置 ······ 238
6.4.5 约束和索引 ······ 239
6.5 物理索引结构 ······ 239
6.5.1 索引行格式 ······ 239
6.5.2 聚集索引结构 ······ 240
6.5.3 聚集索引的非叶级 ······ 241
6.5.4 分析聚集索引结构 ······ 241
6.5.5 非聚集索引结构 ······ 246
6.6 特殊索引结构 ······ 255
6.6.1 计算列上的索引和索引视图 ······ 255
6.6.2 全文索引 ······ 262
6.6.3 空间索引 ······ 262
6.6.4 XML 索引 ······ 262
6.7 数据修改的内部 ······ 263
6.7.1 插入行 ······ 263
6.7.2 拆分页 ······ 264
6.7.3 删除行 ······ 267
6.7.4 更新行 ······ 272
6.7.5 表级数据修改与索引级数据修改 ······ 275
6.7.6 日志记录 ······ 276
6.7.7 锁定 ······ 276
6.7.8 碎片 ······ 276
6.8 管理索引结构 ······ 277
6.8.1 删除索引 ······ 277
6.8.2 ALTER INDEX ······ 278
6.8.3 检测碎片 ······ 279
6.8.4 删除碎片 ······ 280
6.8.5 重建索引 ······ 282
6.9 小结 ······ 284

第 7 章 特殊存储 ······ 285

7.1 大型对象存储 ······ 285

7.1.1 长度受限的大型对象数据（行溢出数据）……285
7.1.2 不限长度大型对象数据……289
7.1.3 最大长度数据的存储……294
7.2 文件流数据……295
7.2.1 为 SQL Server 启用文件流数据……295
7.2.2 创建一个启用文件流的数据库……296
7.2.3 创建一张表存储文件流数据……297
7.2.4 操纵文件流数据……298
7.2.5 文件流数据的元数据……302
7.2.6 文件流数据性能方面的考虑……304
7.3 稀疏列……305
7.3.1 稀疏列的管理……305
7.3.2 列集和稀疏列操作……307
7.3.3 物理存储……309
7.3.4 元数据……311
7.3.5 利用稀疏列节省存储空间……312
7.4 数据压缩……315
7.4.1 Vardecimal……315
7.4.2 行压缩……315
7.4.3 页压缩……322
7.5 表和索引分区……329
7.5.1 分区函数和分区方案……330
7.5.2 分区的元数据……331
7.5.3 分区的滑动窗口优势……334
7.6 小结……336

第 8 章 查询优化器……337

8.1 概述……337
8.1.1 树格式……337
8.2 什么是优化……338
8.3 查询优化器如何研究查询计划……339
8.3.1 规则……339
8.3.2 属性……339
8.3.3 替代项的存储——“备注”……341
8.3.4 运算符……341
8.4 优化器架构……346
8.4.1 优化之前……347
8.4.2 简化……347
8.4.3 琐碎计划/自动参数化……347
8.4.4 限制……348
8.4.5 备注——有效地探索多项计划……349
8.5 统计信息、基数估计和开销……350
8.5.1 统计信息设计……351
8.5.2 密度/频度信息……353
8.5.3 筛选的统计信息……355
8.5.4 字符串统计信息……356
8.5.5 基数估计细节……356
8.5.6 限制……359
8.5.7 成本计算……360
8.6 索引选择……361
8.6.1 筛选索引……363
8.6.2 索引视图……365
8.7 分区表……369
8.7.1 分区对齐索引视图……372
8.8 数据仓库……372
8.9 更新……372
8.9.1 Halloween 保护……375
8.9.2 拆分/排序/折叠……375
8.9.3 合并……377
8.9.4 大范围更新计划……379
8.9.5 稀疏列更新……381
8.9.6 分区更新……381
8.9.7 锁定……384
8.10 分布式查询……385
8.11 扩展的索引……387
8.11.1 全文索引……387
8.11.2 XML 索引……387
8.11.3 空间索引……388
8.12 计划提示……389
8.12.1 调试计划问题……389
8.12.2 {HASH|ORDER}GROUP……391
8.12.3 {MERGE|HASH|CONCAT} UNION……391
8.12.4 FORCE ORDER, {LOOP |

MERGE | HASH } JOIN……391
8.12.5 INDEX=<indexname> | <indexid>……392
8.12.6 FORCESEEK……392
8.12.7 FAST <number_rows>……393
8.12.8 MAXDOP <N>……393
8.12.9 OPTIMIZE FOR……393
8.12.10 PARAMETRIZATION {SIMPLE|FORCED}……395
8.12.11 NOEXPAND……395
8.12.12 USE PLAN……395
8.13 小结……397

第 9 章 计划缓存和重新编译……398

9.1 计划缓存……398
9.1.1 计划缓存元数据……398
9.1.2 清除计划缓存……399
9.2 缓存机制……399
9.2.1 即席查询缓存……400
9.2.2 即席工作负荷优化……402
9.2.3 简单参数化……404
9.2.4 已准备查询……408
9.2.5 已编译对象……410
9.2.6 重新编译的原因……412
9.3 计划缓存内部……420
9.3.1 缓存存储……420
9.3.2 编译计划……421
9.3.3 执行上下文……422
9.3.4 计划缓存元数据……422
9.3.5 句柄……422
9.3.6 sys.dm_exec_sql_text……423
9.3.7 sys.dm_exec_query_plan……424
9.3.8 sys.dm_exec_text_query_plan……424
9.3.9 sys.dm_exec_cached_plans……425
9.3.10 sys.dm_exec_cached_plan_dependent_objects……425
9.3.11 sys.dm_exec_requests……425
9.3.12 sys.dm_exex_query_stats……426
9.3.13 缓存大小管理……426
9.3.14 缓存项的成本……429
9.4 计划缓存中的对象：概况……429
9.5 缓存中的多个计划……431
9.6 何时使用存储过程和其他缓存机制……432
9.7 计划缓存问题故障排除……432
9.7.1 等待统计信息表明存在计划缓存问题……432
9.7.2 其他缓存问题……434
9.7.3 处理编译和重新编译问题……434
9.7.4 计划指南和优化提示……435
9.8 小结……444

第 10 章 事务和并发性……445

10.1 并发模型……445
10.1.1 悲观并发……445
10.1.2 乐观并发……445
10.2 事务处理……446
10.2.1 ACID 属性……446
10.2.2 事务依赖性……447
10.2.3 隔离级别……448
10.3 锁定……451
10.3.1 锁定基础……451
10.3.2 旋转锁……452
10.3.3 用户数据的锁类型……452
10.3.4 锁模式……452
10.3.5 锁粒度……455
10.3.6 锁的持续时间……460
10.3.7 锁的所有权……460
10.3.8 查看锁……461
10.3.9 锁定示例……463
10.4 锁兼容性……468
10.5 锁定内部架构……469
10.5.1 锁分区……470
10.5.2 锁块……471
10.5.3 锁所有者块……472
10.5.4 syslockinfo 表……472
10.6 行级别锁与页级别锁……475

10.6.1 锁升级 ····· 475
10.6.2 死锁 ····· 477
10.7 行版本控制 ····· 480
10.7.1 行版本控制概述 ····· 480
10.7.2 行版本控制细节 ····· 481
10.7.3 基于快照的隔离级别 ····· 481
10.7.4 选择并发模型 ····· 496
10.8 控制锁定 ····· 497
10.8.1 锁提示 ····· 497
10.8.2 设置锁超时 ····· 499
10.9 小结 ····· 500

第 11 章 DBCC 揭秘 ····· 501

11.1 获得数据库的一致性视图 ····· 502
11.1.1 获得一致性视图 ····· 502
11.2 有效地处理数据库 ····· 504
11.2.1 事实生成 ····· 505
11.2.2 使用查询处理器 ····· 506
11.2.3 批处理 ····· 508
11.2.4 读取要处理的页 ····· 509
11.2.5 并行性 ····· 509
11.3 早期的系统目录一致性检查 ····· 511
11.4 分配一致性检查 ····· 512
11.4.1 收集分配事实 ····· 512
11.4.2 检查分配事实 ····· 513
11.5 按表进行逻辑一致性检查 ····· 514
11.5.1 元数据一致性检查 ····· 515
11.5.2 页审核 ····· 516
11.5.3 数据和索引页处理 ····· 518
11.5.4 列处理 ····· 519
11.5.5 文本页处理 ····· 522
11.5.6 跨页一致性检查 ····· 523
11.6 跨表一致性检查 ····· 532
11.6.1 Service Broker 一致性检查 ····· 532
11.6.2 跨目录一致性检查 ····· 533
11.6.3 索引视图一致性检查 ····· 533
11.6.4 XML 索引一致性检查 ····· 534
11.6.5 空间索引一致性检查 ····· 534
11.7 DBCC CHECKDB 输出 ····· 535
11.7.1 标准输出 ····· 535
11.7.2 SQL Server 错误日志输出 ····· 537
11.7.3 应用程序事件日志输出 ····· 538
11.7.4 进度报告输出 ····· 538
11.8 DBCC CHECKDB 选项 ····· 539
11.8.1 NOINDEX ····· 540
11.8.2 修复选项 ····· 540
11.8.3 ALL_ERRORMSGS ····· 540
11.8.4 EXTENDED_LOGICAL_CHECKS ····· 541
11.8.5 NO_INFOMSGS ····· 541
11.8.6 TABLOCK ····· 541
11.8.7 ESTIMATEONLY ····· 541
11.8.8 PHYSICAL_ONLY ····· 542
11.8.9 DATA_PURITY ····· 542
11.9 数据库修复 ····· 542
11.9.1 修复机制 ····· 543
11.9.2 紧急模式修复 ····· 544
11.9.3 哪些数据可以由修复删除 ····· 545
11.10 除 DBCC CHECKDB 之外的一致性检查命令 ····· 545
11.10.1 DBCC CHECKALLOC ····· 546
11.10.2 DBCC CHECKTABLE ····· 547
11.10.3 DBCC CHECKFILEGROUP ····· 547
11.10.4 DBCC CHECKCATALOG ····· 547
11.10.5 DBCC CHECKIDENT ····· 548
11.10.6 DBCC CHECKCONSTRAINTS ····· 548
11.11 小结 ····· 548

第 1 章
SQL Server 2008 架构和配置

Kalen Delaney

SQL Server 是微软公司最著名的数据库管理系统，而 SQL Server 2008 是功能最强大、最完善的版本。除了核心数据库引擎之外，它还允许您存储和检索大量关系数据，使用世界一流的查询优化器。优化器能以最快的方式处理查询并访问数据。许多其他组件还提高了数据的可用性，并使数据和应用程序变得更有效和更易于扩展。可以想像，单独一本书不可能深入介绍所有这些特性。本书将介绍核心数据库引擎的主要特性。

本书将深入探究 SQL Server 数据库引擎特定功能的细节。在第 1 章中，您将高屋建瓴地了解数据库引擎的组件及其协调工作机制，目的是帮助您理解后续章节中介绍的主题如何与数据库引擎的整体操作相符合。

不过，我们将在本章深入介绍 SQL Server 数据库引擎一个大的方面：SQL 操作系统（SQLOS），特别是与内存管理和计划有关的组件，这些内容在后续章节中不再介绍。另外，我们还将介绍元数据，通过使用 SQL Server 元数据观察引擎行为和数据组织。

1.1 SQL Server 版本

SQL Server 的版本来自不同的版次，可以将它看成产品功能的子集，每个版本都拥有自身特定的价格和许可证需求。虽然我们在本书中不讨论价格和许可证，但某些版本信息非常重要，因为每种版本包含的功能不同。*SQL Server 联机丛书*详细描述了每种版本支持的可用性和功能清单。可以使用以下查询方式验证您正在使用的 SQL Server 版本：

```
SELECT SERVERPROPERTY('Edition');
```

另外，您还可以检查名为 *EngineEdition* 的服务器属性，方式如下：

```
SELECT SERVERPROPERTY('EngineEdition');
```

EngineEdition 属性将返回值 2、3 或 4（不可能返回 1），该值确定了可以使用哪些功能。3 表明您的 SQL Server 版本要么是企业（Enterprise）版本和企业评估（Enterprise Evaluation 版本，要么是开发人员（Developer）版本，这 3 种版本的特性和功能几乎完全相同。如果 *EngineEdition* 的值为 2，表明您的 SQL Server 版本要么是标准（Standard）版本，要么是工作组（Workgroup）版本，可以使用的功能更少。本书讨论的功能和行为是标准版本和工作组版本可以使用的某些功能。在企业版本（以及开发人员版本和企业评估版本）中而非标准版本中存在的功能通常与可伸缩性和高可用性功能有关，但也包括其他一些企业版本中特有的功能。我们在讨论这些企业版本特有的功能时将提醒您。有关每种版本所包含内容的完整信息，请参阅 *SQL Server 联机丛书*中的主题“SQL Server 2008 版本支持的功能”（*EngineEdition* 的值

为4，表明您的SQL Server版本是Express版本，它包括SQL Server Express、SQL Server Express with Advanced Services 和 Windows Embedded SQL。我们不再具体讨论这些版本）。另外，还有一个名为*EditionID*的*SERVERPROPERTY*属性，该属性允许您在特定版本间区分每个不同的*EngineEdition*值（即它允许您在企业版本、企业评估版本和开发人员版本间进行区分）。

1.2 SQL Server 元数据

SQL Server维护着一组表，这些表用于存储所有对象、数据类型、约束条件、配置选项的相关信息，以及SQL Server可用的资源。在SQL Server 2008中，这些表称为*系统基表*。某些系统基表仅存在于主数据中，并包含系统范围的信息。其他系统基表存在于所有数据库（包括*master*数据库）中，并包含属于特定数据库的对象和资源。从SQL Server 2005开始，在主数据库或任何其他数据库中，系统基表在默认情况下并不是始终可见的。在SQL Server Management Studio下的“对象资源管理器”中展开*tables*节点时，将无法查看系统基表。除非是系统管理员，否则当执行*sp_help*系统程序时，仍然无法查看系统基表。如果作为系统管理员登录，并从目录视图（简要讨论）中选择*sys.objects*对象时，可以查看所有系统表的名称。例如，以下查询将在SQL Server 2008实例中返回58个输出行：

```
USE master;
SELECT name FROM sys.objects
WHERE type_desc = 'SYSTEM_TABLE';
```

不过，即使作为系统管理员，如果试图从先前查询所返回的某个表名称中选择数据，将获取208错误信息，提示该对象名称不合法。使用专用管理员连接（DAC）权限进行连接是唯一可以在系统基表中查看数据的方法，在本章后面“计划程序”一节我们将介绍该方法。记住，系统基表仅在数据库引擎内部使用，而不是一般的功能。系统基表易于变化，因此无法保证其兼容性。在SQL Server 2008中，有3种系统元数据对象。一种是动态管理对象，我们将在本章后面讨论SQL Server计划和内存管理部分介绍该对象。这些动态管理对象实际上与物理表并不对应，它们包含从内部结构收集的一些信息，并允许您查看SQL Server实例的当前状态。其他两种系统对象用于查看系统基表顶部的结构。

1.2.1 兼容性视图

在SQL Server 2005之前的版本中，虽然允许您在系统表中查看数据，但并不鼓励您这样做。不过，很多用户使用系统表开发自己的故障诊断、报告工具和技术，提供的结果集无法使用系统过程。您可能认为，既然系统基表不可访问，那么使用SQL Server 2005或2008时，必须通过DAC来使用自己的开发工具。不过，您也许会失望，因为SQL Server 2000系统表的许多名称和内容都发生了改变，因此即使使用DAC，凡是名称和内容发生变化的代码都将完全不可用。DAC仅适用于提供紧急访问，不提供其他用途。为了不让您失望，SQL Server 2005和2008提供了一组兼容性视图，允许您继续访问SQL Server 2000系统表的子集。虽然这些视图在隐藏的资源数据库中创建，但您可以从任何数据库中获取它们。

某些兼容性视图的名称您也许非常熟悉，如*sysobjects*、*sysindexes*和*sysdatabases*。其他诸如*sysmembers*和*sysmessages*也许不太熟悉。考虑到兼容性因素，SQL Server 2008中的视图名称与SQL Server 2000对应部分具有相同的名称和相同的列名称。这意味着使用SQL Server 2000系统表的任何代码都不会出现中断。不过，当您从这些视图中进行选择时，不能保证您得到的结果与SQL Server 2000中得到的对应表完全相同。另外，兼容性视图不包含与SQL Server 2005或SQL Server 2008新功能相关的任何元数

据，如分区或资源调控器（Resource Governor）。您只需要考虑兼容性视图的向后兼容性，对于向前兼容性，您应该考虑使用其他元数据机制（如下一节将要讨论的目录视图）。在将来的 SQL Server 版本中将会删除所有的兼容性视图。

更多信息：

在 *SQL Server 联机丛书*中可以查看这些视图中的完整名称列表和列。

SQL Server 2005 和 SQL Server 2008 还为 SQL Server 2000 伪表提供了兼容性视图。伪表是一种表，但它不是基于磁盘数据储存的，而是出于内部结构需求而构建的，可以像表一样对它进行查询。SQL Server 2005 使用动态管理对象替换了这些伪表。注意，SQL Server 2000 伪表和 SQL Server 2005、SQL Server 2008 的动态管理对象之间并非总是一一对应的。例如，使用 SQL Server 2008 检索 *sysprocesses* 中所有可用信息时，您必须访问 3 个动态管理对象：*sys.dm_exec_connections*、*sys.dm_exec_sessions* 和 *sys.dm_exec_requests*。

1.2.2 目录视图

SQL Server 2005 引入了一组目录视图作为保留系统元数据的通用接口。所有目录视图（包括动态管理对象和兼容性视图）都在 *sys* 模式中，访问对象时，必须引用模式名称。某些模式的名称易于记忆，因为它们与 SQL Server 2000 系统表名称类似。例如，*sys* 模式中有一个名为 *objects* 的目录视图，引用该视图可以执行以下语句：

```
SELECT * FROM sys.objects;
```

类似地，还有名为 *sys.indexes* 和 *sys.databases* 的目录视图，但这些目录视图显示的列完全不同于兼容性视图中的列。因为这些查询类型的输出太宽，无法进行复制。我建议您亲自运行这两种查询，并观察它们的区别：

```
SELECT * FROM sys.databases;
SELECT * FROM sysdatabases;
```

Sysdatabases 兼容性视图位于 *sys* 架构中，因此可以将它引用为 *sys.sysdatabases*。另外，还可以使用 *dbo.sysdatabases* 引用它。同样，考虑到兼容性因素，不需要使用架构名称，因为名称是供目录视图使用的（也就是说，不能简单地从名为 *databases* 的视图中选择对象，必须使用架构 *sys* 作为前缀）。

对比前面两个查询的输出，您也许会发现，*sys.databases* 目录视图中包含更多的列。在 *sys.databases* 中，每个数据库属性都有自己的列，而不显示需要解码的位图 *status* 字段。在 SQL Server 2000 中，需要运行系统过程 *sp_helpdb* 对数据库选项进行解码，但 *sp_helpdb* 只是一个过程，很难对结果进行筛选。作为视图，可以查询并筛选 *sys.databases*。例如，如果需要了解哪些数据库处于 *simple* 恢复模式，可以运行以下语句：

```
SELECT name FROM sys.databases
WHERE recovery_model_desc = 'SIMPLE';
```

目录视图是在继承模型上构建的，因此许多对象的通用属性集不必在内部进行重复定义。例如，*sys.objects* 包含各种对象类型通用的所有属性列，视图 *sys.tables* 和 *sys.views* 含有与 *sys.objects* 完全相同

的列，还含有与特定对象类型相关的一些附加列。如果从 *sys.objects* 中选择对象，您将获取 12 列；如果从 *sys.tables* 中选择对象，您将获取排列顺序完全相同的 12 列和 15 个附加列，这些附加列不适用于所有对象类型，但对表起作用。另外，与派生视图（如 *sys.tables*）相比较，虽然基视图 *sys.objects* 包含列的子集，但它还包含行的超集。例如，*sys.objects* 视图展示了过程的元数据和除表之外的元数据，但 *sys.tables* 视图仅展示表的行。因此，可以将基视图和派生视图的关系总结为：基视图包含列的子集和行的超集，而派生视图包含列的超集和行的子集。

与在 SQL Server 2000 中一样，某些元数据仅显示在 *master* 数据库中，用于跟踪系统范围内的数据，如数据库和登录数据。而其他元数据可以显示在所有数据库中，如对象和权限数据。*SQL Server 联机丛书*中的主题“将系统视图映射到系统表”将其对象分成两类列表：仅在 *master* 数据库中显示的对象和在所有数据库中显示的对象。注意，通过目录视图无法使用仅在 *msdb* 数据库中显示的元数据，但仍然可以在系统表和 *dbo* 架构中使用它。这些元数据包括备份和还原、复制、数据库维护计划、Integration Services、日志传送和 SQL Server 代理。

作为视图，这些元数据对象建立在基础 Transact-SQL（T-SQL）定义上。查看这些视图定义最直接的方法是使用 *object_defi nition* 函数（也可以通过 *sp_helptext* 或从目录视图 *sys.system_sql_modules* 中进行选择来查看这些系统视图定义）。因此要查看 *sys.tables* 定义，可以执行以下语句：

```
SELECT object_definition (object_id('sys.tables'));
```

执行上面的 SELECT 语句，您将发现，sys.tables 定义引用了几个完全未记录的系统对象。另一方面，某些系统对象定义仅引用有记录的对象。例如，兼容性视图定义 *syscacheobjects* 仅引用 3 个记录完整的动态管理对象（1 个视图：sys.dm_exec_cached_plans；2 个函数：sys.dm_exec_sql_text 和 sys.dm_exec_plan_attributes）。

名称以“sys.dm_”开头的元数据都可以看做是动态管理对象，如刚才提到的 *sys.dm_exec_cached_plans*。我们将在下一节讨论 SQL Server 数据库引擎的行为时介绍这些元数据。

1.2.3 其他元数据

虽然目录视图是访问 SQL Server 2008 目录的推荐接口，但同样可以使用其他工具。

1. 信息架构视图

SQL Server 7.0 中引入的信息架构视图是 SQL Server 元数据的原始系统表无关视图。SQL Server 2008 中包含的信息视图架构遵从 SQL-92 标准，并且所有这些视图都在名为 *INFORMATION_SCHEMA* 的架构中。通过目录视图获取的某些信息也可以通过信息架构视图获取。需要编写访问元数据的可移植应用程序时，应该考虑使用这些对象。不过，信息架构视图仅显示那些与 SQL-92 标准兼容的对象。这意味着对于某些特定的功能，不存在信息架构视图，如标准中没有进行定义的索引（索引是实现细节）。如果代码不需要进行严格移植，或者您需要非标准功能的元数据（如索引、文件组、CRL 和 SQL Server Service Broker），建议使用 Microsoft 提供的目录视图。文档中的大部分示例，以及本书和其他参考书中的大部分示例都建立在目录视图接口上。

2. 系统函数

大多数 SQL Server 系统函数都是属性函数，SQL Server 7.0 中引入了这些函数，并在 SQL Server 2000

中增强了这些函数。SQL Server 2005 和 SQL Server 2008 又进一步增强了这些函数。属性函数为大量 SQL Server 对象、SQL Server 数据库及 SQL Server 实例本身提供了单独的值。与表不同的是，属性函数返回的值是标量，因此可以作为 *SELECT* 语句的返回值和表中列的填充值。以下是 SQL Server 2008 中可以使用的属性函数清单。

- *SERVERPROPERTY*
- *COLUMNPROPERTY*
- *DATABASEPROPERTY*
- *DATABASEPROPERTYEX*
- *INDEXPROPERTY*
- *INDEXKEY_PROPERTY*
- *OBJECTPROPERTY*
- *OBJECTPROPERTYEX*
- *SQL_VARIANT_PROPERTY*
- *FILEPROPERTY*
- *FILEGROUPPROPERTY*
- *TYPEPROPERTY*
- *CONNECTIONPROPERTY*
- *ASSEMBLYPROPERTY*

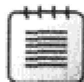
查找各种函数的可能属性值的唯一方法是查看 *SQL Server 联机丛书*。

使用目录视图也可以查看由属性函数返回的某些信息。例如，*DATABASEPROPERTYEX* 函数具有名为 *Recovery* 的属性，该属性将返回数据库的恢复模型。要查看单个数据库的恢复模型，可以使用如下所示的属性函数：

```
SELECT DATABASEPROPERTYEX('msdb', 'Recovery');
```

查看所有数据库的恢复模型，可以使用 *sys.databases* 视图：

```
SELECT name, recovery_model, recovery_model_desc
FROM sys.databases;
```



注意：

名称以_*desc* 结尾的列是所谓的友好名称列，它们总是与其他更紧凑而隐蔽的列成对出现。在本例中，*recovery_model* 列是 *tinyint* 型，其值为 1、2 或 3。因为不同的用户有不同的需求，因此成对出现的两列都可以在视图中获取。例如，在 Microsoft 内部，构建内部接口的团队需要绑定更紧凑的列，而运行临时查询的 DBA 也许更喜欢友好名称。

除了属性函数之外，系统函数还包括目录视图访问的快捷方式。例如，查找 *AdventureWorks2008* 数据库的数据库 ID，可以查询 *sys.databases* 目录视图，或者使用 *DB_ID()*函数。以下两个 *SELECT* 语句将返回相同的结果：

```
SELECT database_id
FROM sys.databases
```

```
WHERE name = 'AdventureWorks2008';

SELECT DB_ID('AdventureWorks2008');
```

3. 系统存储过程

除了系统表自身之外，系统存储过程是原始的元数据访问工具。在SQL Server第一个发布版本中引入的大多数系统存储过程仍然可用。不过相对于过程来说，目录视图是一个很大的进步：可以像对待表一样对这些视图进行查询，因此可以控制元数据的数量。对系统存储过程而言，必须接受它返回的数据。某些过程也允许使用参数，但这类过程数量有限。因此，对 *sp_helpdb* 过程来说，可以传递参数只查看一个数据库的信息，或者不传递参数来查看所有数据库的信息。不过，如果只查看登录 *sue* 拥有的数据库，或者只查看低兼容性级别的数据库，则无法使用系统提供的存储过程。使用目录视图，这些查询将非常简单：

```
SELECT name FROM sys.databases
WHERE suser_sname(owner_sid) ='sue';

SELECT name FROM sys.databases
WHERE compatibility_level < 90;
```

4. 元数据小结

图1-1展示了SQL Server 2008中可以使用的多个元数据层，最低层是系统基表（真正的目录）。任何访问系统基表信息的接口都受元数据的安全策略控制。对SQL Server 2008来说，这意味着用户将无法查看任何不需要查看或未特别授权的元数据（也有一些例外情况，但非常少）。“其他元数据”是指系统表中未包含的系统信息，如动态管理对象提供的内部信息。注意，系统元数据首选的接口是目录视图和系统函数。虽然并不是所有的兼容性视图、*INFORMATION_SCHEMA* 视图和系统过程都真正按照目录视图进行定义，但从概念上说，将它们看成是目录视图接口顶部的其他层是非常有用的。

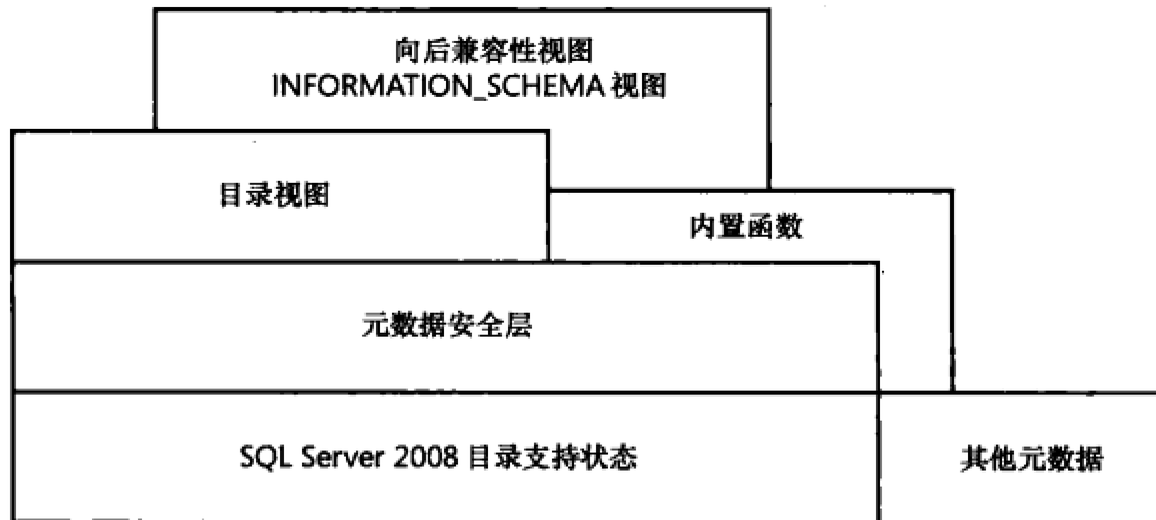


图1-1　SQL Server 2008 中的元数据层

1.3　SQL Server 引擎组件

图1-2展示了SQL Server的一般体系结构，它有4个主要组件。图中展示了其中的3个组件及它们

的子组件：关系引擎（也称为*查询处理器*）、存储引擎和 SQLOS（第 4 个组件是协议层，图中没有展示）。来自任何客户端的应用程序，只要是提交给 SQL Server 执行，都必须与这 4 个组件交互（为简捷起见，我进行了一些微小的省略和简化改动，并在子组件中忽略了某些“帮助程序”模块）。

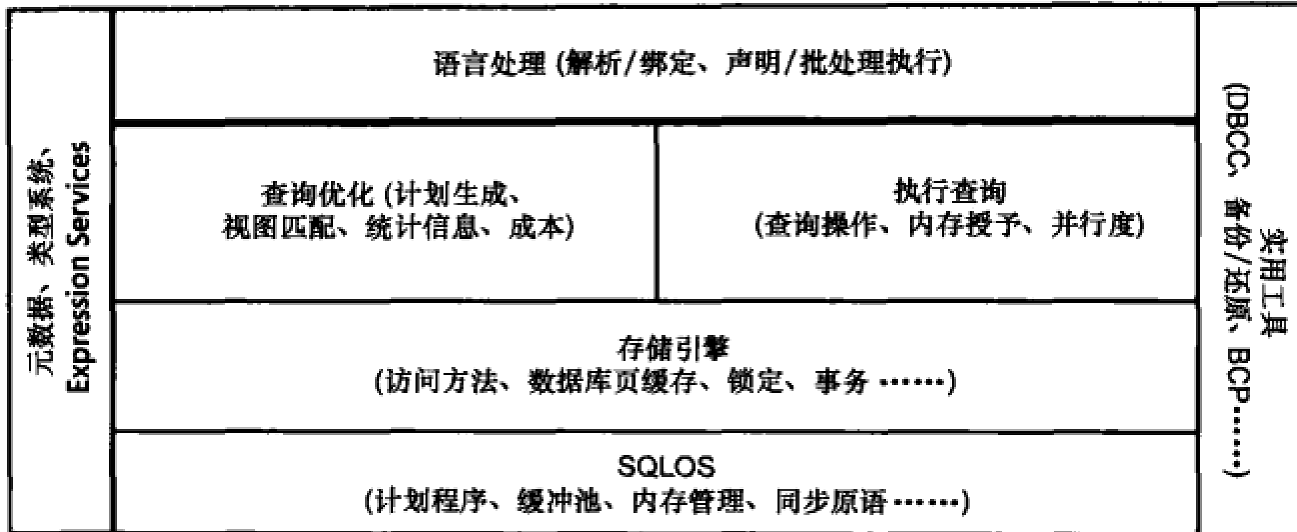


图 1-2 SQL Server 数据库引擎的主要组件

协议层用于接收请求，并把请求转换成关系引擎可以处理的形式。另外，协议层还获取所有查询的最终结果、状态消息或错误消息，并在将它们返回给客户端之前转换成客户端可以识别的形式。关系引擎层用于接收 T-SQL 批处理任务，并确定如何处理它们。对 T-SQL 查询和编程结构而言，关系引擎层用于解析、编译和优化查询，并监督批处理任务的执行过程。执行批处理任务时，如果需要数据，获取数据的请求将传送给储存引擎。存储引擎用于管理所有的数据访问，包括通过基于事务的命令和批量操作（如备份、批量插入）访问，以及特定的 DBCC 命令访问。通常认为 SQLOS 层处理的活动是操作系统的职责，如线程管理（计划）、同步原语、死锁检测、内存管理及缓冲池。

1.3.1 观察引擎行为

SQL Server 2008 包含很多系统对象，允许开发人员和数据库管理员观察许多 SQL Server 内部结构。SQL Server 2005 中引入的这些元数据对象名为*动态管理对象*。这些对象既有视图又有函数，但大部分是视图（通常大家将动态管理对象称为动态管理视图[DMV]，说明大多数对象都是视图）。这些元数据对象就像驻留在 *sys* 架构（每个 SQL Server 2008 数据库中都存在 sys 架构）中一样，可以对它们进行访问，但它们并不是存储在磁盘上的真实表，它们类似于 SQL Server 2000 中用于观察活动进程（*sysprocesses*）或计划缓存（*syscacheobjects*）内容时使用的伪表。不过，SQL Server 2000 中的伪表不提供任何详细的资源使用率追踪信息，因此不能直接用于检测资源问题或状态变化。通过某些 DMV 对象，可以追踪历史资源的详细信息，虽然不是所有 DMV 对象都有记录，但使用 SQL *SELECT* 语句可以直接对 100 多个 DMV 对象进行查询和连接。DMV 将展示发生变化的服务器状态信息，该信息也许涉及多个会话、多个事务和多个用户请求。这些对象可用于故障诊断、内存和进程调整，以及监视服务器中所有的会话。另外，它们还能通过管理数据仓库（它是 SQL Server 2008 的新功能）的性能报告提供更多有效的数据（注意，本章前面“SQL Server 元数据”一节提到的兼容性视图仍然可以使用 *sysprocesses* 和 *syscacheobjects*）。

DMV 不是基于数据库文件中存储的真实表，而是基于内部数据库结构，其中的某些内容我们会在本章进行讨论。本书将在不同的章节对 DMV 进行更详尽的介绍，其中的一个或多个对象内容可以解释讨论的主题。根据 DMV 对象展示的信息职能范围，它们被分成若干个目录。DMV 对象都在 *sys* 架构中，名

称以 *dm_* 开头，随后的代码指示对象处理的服务器范围。以下是我们即将讨论的主要目录。

dm_exec_*

包含与用户代码执行和相关连接直接或间接有关的信息。例如，SQL Server 上每个通过身份验证的会话，*sys.dm_exec_sessions* 将返回 1 行信息。该对象包含的许多信息与 *sysprocesses* 包含的信息相同，但它含有更多有关每个会话的操作环境信息。

dm_os_*

包含低级系统信息，如内存、锁定和计划。例如，*sys.dm_os_schedulers* 是一个 DMV，它将返回每个计划的 1 行信息。它主要用于监视计划程序的情况或标识失控的任务。

dm_tran_*

包含当前事务的细节信息。例如，*sys.dm_tran_locks* 返回当前活动的锁资源信息。每一行代表向锁管理组件发送当前的活动请求，申请已经被授权或正在等待授权的锁。

dm_io_*

跟踪网络和磁盘上的 I/O 活动。例如，函数 *sys.dm_io_virtual_fi le_stats* 将返回数据和日志文件的 I/O 统计信息。

dm_db_*

包含数据库和数据库对象（如索引）的细节信息。例如，*sys.dm_db_index_physical_stats* 函数将返回指定表或视图的数据和索引的大小及碎片信息。

另外，SQL Server 2008 还为很多功能组件提供了动态管理对象，包括用于监视全文检索目录、变更数据捕捉（CDC）信息、Service Broker、复制和 CLR 的对象。

现在，我们将介绍 SQL Server 数据库引擎的主要组件。

1.3.2　协议

应用程序与数据库引擎进行通信时，协议层公开的应用程序编程接口（API）使用 Microsoft 定义的*表格格式数据流*（TDS）*包*格式化通信信息。服务器和客户端计算机上的 SQL Server 网络接口（SNI）协议层在标准通信协议（如 TCP/IP 或 Named Pipes）内部封装 TDS 包。在通信服务器端，网络库是数据库引擎的组成部分。在客户端，网络库是 SQL 本地客户端的组成部分。客户端的配置和 SQL Server 实例确定使用哪些协议。

可以对 SQL Server 进行配置，使它同时支持来自不同客户端的多个协议。每个客户端使用单个协议连到 SQL Server。如果客户端程序不知道 SQL Server 正在侦听哪个协议，可以配置客户端按顺序尝试多个协议。可以使用以下协议。

Shared Memory。它是最易于使用的协议，没有可配置设置。使用 Shared Memory 协议的客户端只能连到在同一计算机上运行的 SQL Server 实例，因此该协议对大多数数据库活动没有作用。如果怀疑其他协议配置不正确，可以使用该协议进行故障诊断。使用 MDAC 2.8 或更早版本的客户端无法使用 Shared Memory 协议，如果试图进行连接，客户端将切换到 Named Pipes 协议。

Named Pipes。开发该协议旨在为局域网（LAN）提供服务。一个进程使用内存的一部分向其他进程

传递信息，因此一个进程的输出是另一个进程的输入。第二个进程可以是本地的（和第一个进程在同一台计算机上），也可以是远程的（在网络计算机上）。

TCP/IP。TCP/IP 协议是 Internet 上使用最广泛的协议。TCP/IP 可以在采用不同硬件架构和操作系统的计算机互联网之间进行通信。它包含一些用于路由网络流量的标准，并提供高级安全功能。在 SQL Server 上启用 TCP/IP 协议需要大量的配置工作，但大多数网络计算机都已经正确配置了该协议。

Virtual Interface Adapter（VIA）。该协议与 VIA 硬件一起工作。这是一种专用的协议，可以从硬件供应商获取详细的配置信息。

表格格式数据流端点

SQL Server 2008 还允许您创建 TDS 端点，以便 SQL Server 侦听附加的 TCP 端口。在安装过程中，SQL Server 自动为所支持的 4 个协议中的每一个创建一个端点，如果启用该协议，所有用户都可以访问它。对禁用的协议而言，端点仍然存在，但无法使用。附加端点是为 DAC 创建的，只有 *sysadmin* 固定服务器角色可以使用 DAC（稍后我们将详细讨论 DAC）。

1.3.3 关系引擎

如前文所述，关系引擎也称为查询处理器。它包含的 SQL Server 组件用于确定查询需要完成的任务，以及如何实现才是最佳方案。在图 1-2 中可以看出，关系引擎包含两个最主要的部分：查询优化（Query Optimization）和查询执行（Query Execution）。查询优化器（Query Optimizer）也许是查询处理器中最复杂的组件，甚至是整个 SQL Server 产品中最复杂的组件，它用于确定批处理中查询的最佳执行方案。在第 8 章中，我们将详细讨论查询优化器，本节仅简要介绍查询优化器和其他查询处理器组件。

另外，当关系引擎从存储引擎中请求数据和处理返回结果时，它还管理查询的执行过程。关系引擎和存储引擎间的通信一般通过 OLE DB 行集（row set）实现（行集是 OLE DB 中的结果集术语）。存储引擎包含需要真正访问和修改磁盘数据的组件。

1. 命令解析器

命令解析器用于处理发往 SQL Server 的 T-SQL 语言事件。它检查语法的正确性，并将 T-SQL 命令转换成可以进行操作的内部格式。这种内部格式称为*查询树*。如果解析器无法识别语法，将立即引起语法错误，并标识出现错误的位置。不过，非语法错误消息不能明确显示引起错误的源行。因为只有命令解析器可以访问语句源代码，真正执行命令时，源代码格式中的语句将不再有效。

2. 查询优化器

查询优化器从命令解析器中获取查询树，并为执行查询准备查询树。无法优化的语句，如控制流和数据定义语言（DDL）命令，将被编译成内部形式。可优化的语句将被做上标记，然后传递到查询优化器中。查询优化器主要涉及数据操作语言（DML）语句 *SELECT*、*INSERT*、*UPDATE* 和 *DELETE*，这些语句可以按多种方式进行处理，查询优化器将从许多可能的方式中确定一种最佳方式。查询优化器将编译整个命令批处理，优化可优化的查询，并检查安全性。该查询优化和编译将产生一个执行计划。

产生该计划的第一步是规范每个查询，该过程可以将单个查询分解成多个细化的查询。查询优化器规范查询后，将对查询进行优化，这意味着它将为执行查询确定计划。查询优化是基于成本的；查询优化器基于内部指标选择消耗最少的计划，内部指标包括评估内存需求、CPU 利用率和需要的 I/O 数目。

查询优化器将根据请求语句类型，检查各种相关表中的数据量，查看每个表中可用的索引，然后查看查询中引用的每个索引或列的数据值采样。数据值采样名为*分发统计信息*（第8章将详细讨论统计信息）。根据可用的信息，查询优化器分析查询中各种可用的访问方法和处理策略，并选择成本效益最高的计划。

查询优化器还使用修剪启发式算法，以确保优化查询的时间不会比简单选择并执行计划花费的时间长。查询优化器不需要执行详尽优化。某些产品考虑每种可能的计划，并选择成本效益最高的计划。详尽优化的优点是，无论用户使用什么语法，查询语法的选择在理论上不会引起性能区别。但使用复杂查询时，评估每种可能的计划花费的时间比接受并执行一个好计划（即使不是最好的计划）花费的时间要长得多。

完成规范化和最优化以后，这些过程产生的规范化树将被编译成执行计划。执行计划实际上是一种数据结构。执行计划中的每个命令精确指定了将涉及哪些表、将使用哪些索引（如果存在多个索引）、将检查哪些安全性，以及哪些条件（如等于某个特定值）在选择中的值必须为TRUE。这种执行计划表面上看似乎很简单，但其实内部过程相当复杂。除了实际的命令，执行计划还包括所有必须执行的步骤，以确保对约束条件进行检查。调用触发器的步骤与验证约束条件略有不同。如果操作行为中已经包含了触发器，还需要追加包含触发器的过程调用。如果触发器是*instead-of*触发器，触发器计划的调用将替换实际的数据修改命令。对*after*触发器而言，触发器计划在修改语句激活触发器之后和提交修改之前执行。与约束条件验证步骤不同的是，触发器的特定步骤没有编译到执行计划中。向含有多个约束条件的表插入一行的简单请求可以导致执行计划访问多个其他表，或者计算表达式的操作。另外，触发器的存在还可以导致更多步骤的执行。执行真正的*INSERT*语句的步骤也许只是整个执行计划的一小部分，但整个执行计划需要确保执行所有与添加一行相关的操作和约束。

3. 查询执行器

查询执行器运行查询优化器产生的执行计划，在执行计划中充当所有命令的调度程序。批处理完成之前，该模块将逐步跟踪执行计划的每个命令。大多数命令需要与存储引擎进行交互来修改或检索数据及管理事务和锁。更多有关查询执行和执行计划的信息，可以访问随附网站：*http://www.SQLServerInternals.com/companion*。

1.3.4 存储引擎

SQL Server存储引擎包括涉及访问和管理数据中数据的所有组件。在SQL Server 2008中，存储引擎主要包括3部分：访问方法、锁定和事务服务及实用工具命令。

1. 访问方法

SQL Server查找数据时，需要调用访问方法代码。访问方法代码建立并请求数据页和索引页扫描，并准备OLE DB行集来返回关系引擎。类似地，在插入数据时，访问方法代码可以从客户端检索OLE DB行集。访问方法代码含有对应的组件来执行打开表、检索合格数据和更新数据操作。访问方法代码实际上并不检索页面，它向缓冲管理器发送请求，最后缓冲管理器在其缓存中提供页面，或者将页面从磁盘读到缓存中。扫描启动时，预测先行（look-ahead）机制将对页面上的行或索引条目进行鉴定。满足特定标准的行检索名为*合格检索*。不仅*SELECT*语句中可以使用访问方法代码，有些*UPDATE*和*DELETE*语句（如含有WHERE从句的*UPDATE*语句），以及需要修改索引条目的任何数据修改操作也可以使用访问方法代码。下面列出了一些访问方法类型。

行和索引操作。您可以将行和索引操作看成是访问方法代码组件，因为它们执行真正的访问方法。每个组件分别负责操作和维护各自在磁盘上的数据结构（分别指数据行或 B 树索引）。它们理解并操作数据和索引页上的信息。

行操作代码用于检索、修改并执行单个行上的操作。它在一行中进行操作，如“检索第 2 列”或“将该值写入第 3 列”。在访问方法代码、锁及事务管理组件（稍后讨论）的协同工作下，需要的行被找到并被锁定（作为事务的一个执行部分）。在内存中格式化或修改行以后，行操作代码将插入或删除一行。如果数据是大型对象（Large Object，LOB）数据类型（如 *text*、*image* 或 *ntext*），或者行太大无法在单个页面上显示而需要存储为溢出数据时，行操作代码需要处理特殊的操作。我们将在第 5 章、第 6 章和第 7 章介绍各种类型的存储结构。

索引操作代码维护并支持搜索 B 树，B 树用于构造 SQL Server 索引。索引是一种树结构，含有根页面、中级页面和低级页面（如果树非常小，也许没有中级页面）。B 树把含有相似索引键的记录进行分组，因此通过搜索关键值可以快速访问数据。B 树的核心功能是能够平衡索引树（B 代表 *balanced*）。索引树的分支根据需要相互连在一起或拆开，因此对于任何给定记录的搜索，将始终遍历相同数目的级别，因而需要相同数目的页面访问。

页面分配操作。分配操作代码管理每个数据库页面的集合，并跟踪一些信息，如数据库中哪些页面已经使用了、为什么使用这些页面，以及每个页面上还有多少可用空间。每个数据库都是 8KB 的磁盘页面集合，这些磁盘页面分布在一个或多个物理文件中（在第 3 章中，您将看到更多有关物理数据库组织的详细内容）。

SQL Server 使用 13 种磁盘页面。我们将在本书中讨论的磁盘页面包括：数据页面、两种 LOB 页面、行溢出页面、索引页面、页面可用空间（PFS）页面、全局分配映射和共享全局分配映射（GAM 和 SGAM）页面、索引分配映射（IAM）页面、批处理更改映射（BCM）页面，以及差异更改映射（DCM）页面。

所有用户数据都存储在数据或 LOB 页面上，所有索引行都存储在索引页面上。PFS 页面用于跟踪数据库中哪些页面可用于存储新数据信息。分配页面（GAM、SGAM 和 IAM）用于跟踪其他页面，它们不包含数据库行，并且仅在内部使用。BCM 和 DCM 页面用于使备份和还原变得更加有效。我们将在第 3 章和第 4 章解释这些页面类型。

版本操作。版本存储是 SQL Server 2005 产品中添加的另一种数据访问类型。行版本允许 SQL Server 维护旧版本的更改行。SQL Server 中的行版本技术支持快照隔离和 SQL Server 2008 的其他功能，其中包括联机索引结构和触发器。版本操作代码负责维护行版本。

第 3、5、6 和 7 章将全面介绍访问方法代码使用的内部结构细节：数据库、表和索引。

2. 事务服务

SQL Server 的核心功能是能够确保事务是*原子*的——也就是说，全或无。另外，事务必须是持久的，这意味着如果事务已经提交了，那么无论在何种情况下，都可以通过 SQL Server 进行恢复——哪怕整个系统在确认提交后 1 毫秒出现故障。事实上，事务必须满足 4 个属性：*原子性*、*一致性*、*隔离性*、*持久性*，这称为 *ACID 属性*。我们将在第 10 章介绍事务管理和并发性问题时再讨论这 4 个属性。

在 SQL Server 中，如果任务正在处理中，并且在提交事务之前出现了系统故障，那么所有工作必须回滚到事务开始前的状态。预写日志记录（Write-ahead logging）用于使处理中的任务回滚，或者使未应用到数据页中的提交任务执行前滚成为可能。在事务确认提交之前，预写日志记录能够确保在磁盘的事务日志中捕获每个事务记录的变化，并且在把真正发生变化的数据页写入磁盘之前，始终先在磁盘中写

入日志记录。写入事务日志始终是同步的，即 SQL Server 必须等它们完成。但写入数据页可以是异步的，因为所有效果都可以根据需要在日志中进行重构。事务管理组件用于协调日志、恢复和缓冲管理。我们将在稍后讨论这些主题，在此仅简要介绍事务本身。

事务管理组件描述形成某个操作而必须组合在一起的语句边界。它用于处理相同 SQL Server 实例中的跨数据库事务，并能处理嵌套事务序列（不过，嵌套事务仅在第一级事务环境中执行；提交嵌套事务时，不能出现特别的操作。低级嵌套事务中指定的回滚操作将取消全部事务）。对其他 SQL Server 实例（或对任何其他资源管理器）的分布式事务而言，事务管理组件使用操作系统远程过程调用来处理 Microsoft 分布式事务协调器（Microsoft Distributed Transaction Coordinator，MS DTC）服务。事务管理组件标记了*保存点*，在事务中通过保存点指示可以在此处部分回滚或取消任务。

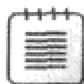
根据实际的隔离级别，当锁可以释放时，事务管理组件还可以配合相关锁代码进行工作。它还可以配合版本代码确定何时不再需要旧版本，并且可以把它从版本存储中删除。事务运行的隔离级别确定了应用程序受其他程序影响的敏感度，因而也确定了事务应对这些变化所必须保持锁定或维护版本数据的时间。

SQL Server 2008 支持两个并发模型来保证事务的 ACID 属性：乐观并发和悲观并发。乐观并发通过锁定数据来确保正确性和并发性，使数据不发生改变；SQL Server 2005 之前的 SQL Server 版本都以独占方式使用这种并发模型，SQL Server 2005 和 SQL Server 2008 在默认情况下也以独占方式使用这种模型。SQL Server 2005 引入了乐观并发，它在名为 *version store* 的 *tempdb* 区域通过保留旧的行版本和已提交的值来提供一致性数据。在悲观并发中，读程序不会阻塞写程序，写程序也不会阻塞读程序，但写程序仍然会阻塞写程序。实际操作中必须考虑这些非阻塞读取和写入行为的开销。为了支持乐观并发，SQL Server 需要花费更多的时间来管理版本存储。另外，管理员必须密切关注 *tempdb* 数据库，并计划为它提供特殊维护。

SQL Server 2008 提供了 5 种隔离级别语法。其中仅 3 种支持悲观并发：未提交读、可重复读和序列化。快照隔离级别支持乐观并发。默认的隔离级别——已提交读——能够同时支持乐观并发和悲观并发，具体情况取决于数据库设置。

事务的行为取决于采用的隔离级别和并发模型。全面理解隔离级别还需要理解锁定的含义，因为它们是密切相关的。下面我们将概要介绍锁定，在第 10 章将会介绍更多有关隔离、事务和并发性管理的详细信息。

锁定操作。锁定是多用户数据库系统（如 SQL Server）中一个非常重要的函数，即便在快照隔离级别中使用乐观并发进行操作，该函数也非常重要。SQL Server 允许您同时管理多个用户，并确保事务能查看选定隔离级别的属性。虽然在快照隔离中读程序不会阻塞写程序，写程序也不会阻塞读程序，但写程序仍然需要锁定，写程序可以阻塞其他写程序。如果两个写程序试图同时更改相同的数据，就必须解决冲突问题。锁定代码将获取和释放各种锁，如读取共享锁、写入独占锁、在更高粒度发出潜在“计划”执行某些操作的意向锁，以及供空间分配使用的扩展锁。它管理不同类型锁的兼容性，解决死锁问题，以及根据需要增强锁的功能。锁定代码用于控制表、页面、行和系统数据锁定。



注意：

并发、锁定和行版本是 SQL Server 非常重要的方面。许多开发人员对此非常感兴趣，因为它能潜在影响应用程序的性能。第 10 章将专门介绍该主题，在此我们不做深入讨论。

3. 其他操作

另外，存储引擎中还包含用于控制实用工具的组件，如大容量加载、DBCC 命令、全文索引填充与管理，以及备份和还原操作。第 11 章将详细讨论 DBCC。日志管理器确保日志记录的写入方式能保证事务的持久性和可恢复性，我们将在第 4 章详细讨论事务日志及其备份和还原操作。

1.4 SQLOS

SQLOS 是一个单独的应用层，它位于 SQL Server 数据库引擎的最低层；SQL Server 和 SQL Reporting Services 都在顶层运行。SQL Server 的早期版本在存储引擎和实际操作系统之间使用瘦接口层，通过该接口层，SQL Server 可以调用操作系统来执行内存分配、计划资源、线程和工作管理，以及同步对象。不过，SQL Server 中需要访问这些接口的服务可以位于引擎的任何部分。SQL Server 需要管理内存、计划程序、同步对象等，因此任务变得越来越复杂。于是，Microsoft 公司设计了单独的应用层来管理所有特定于 SQL Server 的操作系统资源，而不是让引擎的每个部分支持增加的功能。

SQLOS 的两个主要函数是计划和内存管理，我们将在稍后部分详细介绍它们。SQLOS 包括的其他函数如下。

- **同步化**。同步化对象包括 spinlocks、mutexes 和系统资源上的特殊读/写锁。
- **内存 broker**。内存 borker 用于在 SQL Server 中的各种组之间分发内存分配，但不执行任何分配，分配由内存管理器处理。
- **SQL Server 异常处理**。异常处理涉及处理用户错误和系统产生的错误。
- **死锁检测**。死锁检测机制不仅包括锁定，还包括检测任何占用资源的任务，这些任务相互阻塞。我们将在第 10 章讨论与锁定有关的死锁现象（目前最常见）。
- **扩展事件**。跟踪扩展事件类似于 SQL 跟踪（SQL Trace）功能，但它更有效。因为与 SQL 跟踪相比，跟踪过程在更低级别运行。另外，因为扩展事件层的级别非常低，因此可以跟踪更多事件类型。SQL Server 2008 资源调控器使用扩展事件管理资源使用率。我们将在第 2 章介绍扩展事件（在未来版本中，所有跟踪都将由扩展事件在该层次进行处理）。
- **异步 IO**。异步和同步的区别在于，系统的哪个部分实际用于等待不可用资源。SQL Server 请求同步 I/O 时，如果资源不可用，Windows 内核将把线程放入等待队列中，直到资源可用为止；对异步 I/O 而言，SQL Server 首先请求 Windows 初始化 I/O，接着 Windows 启动 I/O 操作，但不阻止运行线程，然后 SQL Server 把服务器线程放在 I/O 请求队列中，直到它从 Windows 获取资源可用的信号为止。

NUMA 架构

SQL Server 2008 采用均衡负载的非一致性内存访问（NUMA）机制，默认情况下，计划和内存管理都可以使用 NUMA 硬件。使用 NUMA 时，可以使用某些特殊的配置。在介绍计划程序和内存之前，我们首先介绍有关 NUMA 架构的一般背景知识。

NUMA 的主要优势是可伸缩性，当您使用对称多处理（SMP）架构时，可伸缩性有明确的限制。使用 SMP 时，所有内存访问都发往相同的共享内存总线。当 CPU 数量相对较少时，SMP 体系机构能正常工作；但使用多个 CPU 竞争访问共享内存总线时，就会出现问题。硬件发展的趋势是拥有多个系统总线。每个处理

器组拥有自己的内存，并且可能拥有自己的 I/O 通道。不过，每个 CPU 可以按照统一的访问方式访问与其他组有关的内存，我们将在本章稍后部分介绍该内容。每个组名为 *NUMA 节点*，并且节点通过高速互联方式相互连接。NUMA 节点中的 CPU 数目取决于硬件生产商。和访问与其他 NUMA 节点有关的内存比较，访问本地内存速度更快一些。这是使用*非一致性内存访问*命名的原因。图 1-3 展示了含有 4 个 CPU 的 NUMA 节点。

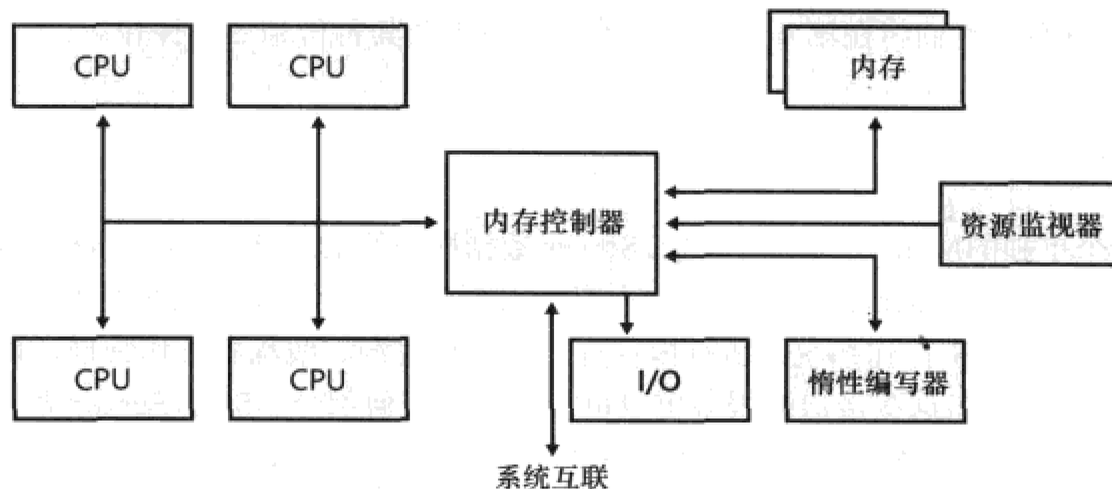


图 1-3　含有 4 个 CPU 的 NUMA 节点

SQL Server 2008 允许您将一个或多个物理 NUMA 节点细分为更小的 NUMA 节点，该过程名为*软件 NUMA* 或*软 NUMA*。使用多个 CPU，但没有硬件 NUMA 时，通常使用*软件 NUMA*，因为软件 NUMA 仅适用于细分 CPU，所以无法细分内存。与硬件 NUMA 相比，您可以用软件 NUMA 将硬件 NUMA 节点细分为更小的 CPU 组。另外，还可以将软件 NUMA 节点进行配置，以侦听它们自身的端口。

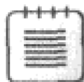
只有 SQL Server 计划程序和 SNI 是均衡负载的软件 NUMA。内存节点是基于硬件 NUMA 而创建的，因此不受软件 NUMA 的影响。

TCP/IP、VIA、Named Pipes 和共享内存可以利用 NUMA 的轮循计划机制，但只有 TCP 和 VIA 可以绑定到特定的 NUMA 节点集中。参阅 *SQL Server 联机丛书*可以了解如何使用 SQL Server 配置管理器设置 TCP/IP 地址和移植到单个或多个节点。

1.5　计划程序

在 SQL Server 7.0 之前的版本中，计划完全依赖于底层的 Microsoft Windows 操作系统。虽然这意味着 SQL Server 可以利用 Windows 工程师的辛苦劳动来增强可伸缩性、提高处理器的使用率，但也存在一定的限制。Windows 计划程序无法了解关系数据库的需求，因此它将 SQL Server 工作线程等同于操作系统上运行的其他进程。但是，当计划程序满足某些特别需求时，高性能系统（如 SQL Server）能发挥最佳功能。SQL Server 7.0 及后续版本旨在处理自身的计划以获取多种优点，具体包括：

- 专用计划程序可以像支持使用线程一样，轻松使用纤程支持 SQL Server 任务；
- 可以尽可能避免上下文切换和切换到内核模式。



注意：

在 SQL Server 7.0 和 SQL Server 2000 中，计划程序称为*用户架构计划程序*（UMS），说明它主要运行在用户模式中，而不是内核模式中。SQL Server 2005 和 SQL Server 2008 称计划程序为 SOS 计划程序，并进一步改进了 UMS。

SOS 计划程序与 Windows 计划程序的主要区别在于：SQL Server 计划程序是作为协作式计划程序而非抢先式计划程序运行的。这意味着它依靠工作线程、线程或纤程来自动让出系统资源，因此一个进程或线程不会独占控制系统。SQL Server 产品工作组必须确保它的代码有效地运行，并且在合适的地方让出计划程序；与使用 Windows 计划程序相比，这样做可以获取更高的控制权和可伸缩性。

虽然计划程序不是抢先式，但 SQL Server 计划程序仍然支持量程的概念。计划程序可以请求定期将 SQL Server 任务放入等待队列中，而不是通过操作系统强制放弃 SQL Server 任务。如果 SQL Server 任务含有过多的内部定义量程，并且在可停止操作队列中，它们将自动让出 CPU。

1.5.1 SQL Server 工作线程

您可以将 SQL Server 计划程序想像成 SQL Server 工作线程使用的逻辑 CPU。它可以绑定到逻辑计划程序的线程或纤程中。如果设置了关联掩码配置（Affinity Mask Configuration）选项，每个计划程序都将关联到某个特定的 CPU（我们将在本章后面介绍有关配置）。因此，每个工作线程也将与单个 CPU 相关联。根据最大工作线程数和计划程序数为每个计划程序分配工作线程的限制数，每个计划程序根据实际需要负责创建或销毁工作线程。无法将工作线程从一个计划程序向另一个计划程序移动，但可以销毁和创建它，因此工作线程看上去就像在计划程序之间移动一样。

当计划程序接收到请求，并且没有闲置工作线程时，将创建工作线程。如果某个工作线程至少已经闲置了 15 分钟，或者 SQL Server 存在内存压力，将销毁工作线程。在 32 位系统上，每个工作线程至少需要 0.5MB 内存；在 64 位系统上，每个工作线程至少需要 2MB 内存。因此，在内存不足的系统上销毁多个工作线程并释放内存可以显著增强性能。SQL Server 可以高效地处理工作线程池，即使在含有数百或上千用户的大型系统上，SQL Server 的实际工作线程数也许比配置的最大工作线程数要少得多，您也许对此感到很奇怪。在本节的稍后部分，我们将介绍一些动态管理对象，通过动态管理对象，您可以知道实际使用的工作线程数、计划程序和任务信息（具体内容将在下一节讨论）。

1. SQL Server 计划程序

在 SQL Server 2008 中，当 SQL Server 启动时，每个实际 CPU（无论是超线程还是物理线程）都将创建计划程序。即使配置了关联掩码选项也将创建计划程序，因此 SQL Server 不会使用所有可用的物理 CPU。在 SQL Server 2008 中，根据关联掩码的配置，每个计划程序要么设置为 ONLINE 状态，要么设置为 OFFLINE 状态。默认情况下，所有计划程序都设置为 OFFLINE 状态。更改关联掩码的值可以将一个或多个计划程序更改成 OFFLINE 状态,并且不需要重新启动 SQL Server。注意，由于配置的更改而导致计划程序从 ONLINE 向 OFFLINE 切换时，已经分配给该计划程序的任何工作必须优先完成，并且系统不再向该计划程序分配新的工作。

2. SQL Server 任务

SQL Server 工作线程的工作单元是*请求*或*任务*，可以将它看成是从客户端发向服务器的单个批处理任务。SQL Server 接收某个请求以后，把请求绑定到工作线程中，工作线程处理该请求之后才处理其他请求。即使由于某种原因该请求被阻塞了（如等待完成某个锁定或 I/O），该工作机制仍然有效。在解决阻塞状态和完成请求之前，特定的工作线程不会处理任何新的请求。注意，会话 ID（SPID）与任务不相同。SPID 是一个连接或通道，可以通过它发送请求，但特定的 SPID 上并不是始终存在活动的请求。

在 SQL Server 2008 中，SPID 不是绑定到特定的计划程序中。每个 SPID 都有优先计划程序（最近处

理了来自该 SPID 的某个请求的计划程序)。SPID 优先分配给负载量最低的计划程序(可以查看 DMV *sys.dm_os_schedulers* 中的 *load_factor* 列,了解每个计划程序的负载量)。不过,当后续请求来自相同的 SPID 时,如果其他计划程序的负载系数小于该计划程序整个负载系数的平均数的某个百分比,那么新任务将分配给具有最低负载系数的计划程序。计划程序必须在相同的 NUMA 节点上处理一个 SPID 的所有任务。当某个查询作为多 CPU 间的并行查询被执行时,优化器可以决定使用更多 NUMA 上可用的 CPU 来处理该查询,因此可以使用其他 CPU(和其他计划程序)。

3. 线程与纤程

如前文所述,设计 UMS 的目的在于使 UMS 和工作线程在线程或纤程上一起运行。与线程相比较,Windows 纤程的相关开销更少,并且可以在单线程上运行多纤程。将轻型池(Lightweight Pooling)选项设置为 1,可以使 SQL Server 在纤程架构下运行。虽然使用更少的开销和"轻型"机制是个很好的办法,但您需要对纤程的使用进行仔细评估。

SQL Server 在纤程架构下运行时,某些 SQL Server 组件不工作,或者不能正常工作。这些组件包括 SQLMail 和 SQLXML。在纤程模式下,还有一些组件(如异类和 CLR 查询)根本无法支持,因为它们需要 Windows 提供某些特定的线程工具。虽然 SQL Server 可以根据需要切换到线程架构来处理请求,但与专门使用线程相比,开销也许更大一些。实际上在某些特殊场合,由于 SQL Server 在线程上下文之间进行切换,或者在用户模式和内核模式之间进行切换需要花费太长时间,导致 SQL Server 在可伸缩性方面受到限制时才使用纤程架构。在大多数环境中,与借助其他方法获取的优势相比较,纤程产生的性能优势非常小。如果确定在某种情形下可以通过纤程获取优势,在对产品服务器选项进行设置之前请务必进行彻底测试。另外,为了进行确认,您也许还需要与 Microsoft 客户支持服务部门(*http://support.microsoft.com/ph/2855*)取得联系。

4. NUMA 与计划程序

通过 NUMA 配置,每个节点都含有机器的处理器子集和相同数目的计划程序。如果配置的机器用于硬件 NUMA,将在每个节点上预先设置处理器的数目。但对于自己配置的软件 NUMA 而言,您可以决定在每个节点上分配多少个处理器。不过,计划程序的数目仍然和处理器的数目相同。首次创建 SPID 时,将按照轮循机制将 SPID 分配到节点上。如前文所述,如果 SPID 移到其他计划程序中,它仍然保留在相同的节点上。单个处理器或 SMP 机器将被看成是含有单个 NUMA 节点的机器。就像在 SMP 机器上一样,计划程序和含有 NUMA 的 CPU 之间不存在硬映射,因此单个节点上的任何计划程序都可以在该节点上的任何一个 CPU 上运行。不过,如果已经设置了关联掩码配置选项,每个节点上的每个计划程序将固定在特定的 CPU 上运行。

每个 NUMA 节点都有自己的惰性编写器(我们将在本章的"内存"一节讨论惰性编写器)和 I/O 完成端口(IOCP)。其中,IOCP 是一种网络侦听器。另外,每个节点还含有自己的资源监视器,并使用隐藏计划程序管理资源监视器。可以在 *sys.dm_os_schedulers* 中查看隐藏计划程序。每个资源监视器都拥有自己的 SPID,通过查询 *sys.dm_exec_requests* 和 *sys.dm_os_workers* DMV 可以查看 SPID,如下所示:

```
SELECT session_id,
    CONVERT (varchar(10), t1.status) AS status,
    CONVERT (varchar(20), t1.command) AS command,
    CONVERT (varchar(15), t2.state) AS worker_state
FROM sys.dm_exec_requests AS t1 JOIN sys.dm_os_workers AS t2
```

```
ON t2.task_address = t1.task_address
WHERE command = 'RESOURCE MONITOR';
```

每个节点都拥有自己的计划程序监视器，可以在任何 SPID 上和抢先式模式上运行。计划程序监视器是一种线程，用于定期唤醒并检查每个计划程序，以便查看从上次计划程序监视器唤醒以后，计划程序是否让出了系统资源（除非计划程序闲置）。如果非闲置线程没有让出系统资源，计划程序监视器将产生错误（17883）。当应用程序（而不是 SQL Server）独占 CPU 时，将出现 17883 错误。计划程序监视器仅知道 CPU 没有让出系统资源，但它不能确定是什么任务正在使用该资源。计划程序监视器还负责向计划程序发送消息，帮助它们平衡工作负荷。

5. 动态关联

在 SQL Server 2008 中(除 SQL Server Express 之外的所有版本中)，可以动态控制处理器的关联。SQL Server 启动时，服务器将启动所有计划任务，因此每个 CPU 上都将分配一个计划程序。如果已经设置了关联掩码，某些计划程序将标识为脱机状态，因此不再向它们分配任务。

当关联掩码更改成包含附加 CPU 时，新的 CPU 将变成联机状态，计划程序监视器将发现工作负荷的不均衡，并开始选取工作线程使其向新的 CPU 移动。更改关联掩码使 CPU 成为脱机状态时，该 CPU 对应的计划程序将继续运行活动的工作线程，但计划程序本身将移到其他某个仍然处于联机状态的 CPU 中。不再向该计划程序分配新的工作线程时，它将处于脱机状态，当所有活动的线程完成任务时，该计划程序将停止运行。

1.5.2 将计划程序绑定到 CPU 中

注意在通常情况下，计划程序并不是严格按照一对一的关系被绑定到 CPU 中的，即使在计划程序的数目和 CPU 数目相同的情况下也是如此。只有设置了关联掩码时，计划程序才被绑定到 CPU 中。即使指定关联掩码使用所有的 CPU（关联掩码的默认设置），计划程序也不会绑定到 CPU 中。例如，默认关联掩码配置值是 0，这意味着使用所有的 CPU，而不是将计划程序硬绑定到 CPU 中。其实在某些情况下，当机器上的负载不大时，Windows 可以在一个 CPU 上运行两个计划程序。

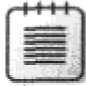
对含有 8 个处理器的机器而言，关联掩码的值为 3（位字符串为 00000011）意味着只使用 CPU 0 和 1，并且两个计划程序都分别被绑定到两个 CPU 中。如果将关联掩码的值设为 255（位字符串为 11111111），意味着使用所有的 CPU，即为默认配置。不过，如果设置了关联掩码，8 个 CPU 将分别绑定 8 个计划程序。

在某些情况下，也许需要限制可用的 CPU 数，但不希望将某个特定的计划程序绑定到单个 CPU 中。例如，您正在使用多 CPU 机器用于处理服务器整合，假设机器有 64 个处理器，有 8 个 SQL Server 实例在 64 个处理器上运行，并且每个实例需要使用 8 个处理器，每个实例的关联掩码都不相同，指定了 64 个处理器的不同子集，因此可能使用的关联掩码值有 255（0xFF）、65280（0xFF00）、16711680（0xFF0000）和 4278190080（0xFF000000）。因为设置了关联掩码，每个实例都将计划程序硬绑定到 CPU 中。如果需要限制 CPU 的数目，但不限制某个特定的计划程序在指定的 CPU 上运行，可以使用跟踪标志 8002 启动 SQL Server。通过使用跟踪标志，您可以将 CPU 映射到某个实例上。但在本实例中，没有将计划程序绑定到 CPU 中。

观察计划程序的内部

SQL Server 2008 含有一些动态管理对象，它们提供与计划程序、工作线程和任务相关的信息。这些

对象主要供 Microsoft 客户支持服务部门使用，但您可以通过它们更深入地了解 SQL Server 监视器的相关信息。



注意：

所有这些对象（及大多数其他动态管理对象）都需要授予*查看服务器状态*权限。默认情况下，只有 SQL Server 管理员拥有该权限，但可以向其他用户授予该权限。对其中的每个对象，我们将列出某些有用的列，并为每个列提供选自 *SQL Server 2008 联机丛书*的描述。对于列的完整列表，大多数仅对支持人员有用，具体信息可以参考 *SQL Server 2008 联机丛书*。参考联机丛书时，您将发现其中的某些列被列为“仅供内部使用”。

这些动态管理对象如下面所示。

- ***sys.dm_os_schedulers***。在 SQL Server 中，该视图将返回每个计划程序的一行。在 SQL Server 中，每个计划程序都被映射到单个处理器中。可以使用该视图来监视计划程序的情况或标识失控任务。比较重要的列如下。
- ***parent_node_id***。计划程序所属的节点的 ID，也称为*父节点*。它代表 NUMA 节点。
- ***scheduler_id***。计划程序的 ID。用于运行定期查询的所有计划程序都有小于 255 的 ID 号。那些 ID 大于或等于 255 的计划程序（例如，专用管理员连接计划程序）则供 SQL Server 内部使用。
- ***cpu_id***。与计划程序关联的 CPU 的 ID。如果将 SQL Server 配置为关联运行，则该值是假定将要运行该计划程序的 CPU 的 ID。如果不指定关联掩码，则 *cpu_id* 的值为 255。
- ***is_online***。如果将 SQL Server 配置为仅使用服务器上某些可用的处理器，这说明某些计划程序被映射到不在关联掩码中的处理器上。在这种情况下，该列将返回值 0。该值表示计划程序不会用于处理查询或批处理。
- ***current_tasks_count***。与该计划程序关联的当前任务数，包括以下任务（完成任务时，该计数将减少）。
  - 等待获取资源然后处理的任务。
  - 当前正在运行或可运行且正在等待执行的任务。
- **runnable_tasks_count**。计划程序上等待运行的任务数。
- ***current_workers_count***。与该计划程序关联的工作线程数，该计数包括未分配任何任务的工作线程。
- **active_workers_count**。已经分配任务的工作线程数。
- **work_queue_count**。等待工作线程的任务数。如果 current_workers_count 大于 active_workers_count，则工作队列数应该为 0，并且工作队列不应该增加。
- ***pending_disk_io_count***。挂起的 I/O 数。每个计划程序都有一个挂起 I/O 列表。每次有上下文切换时，都将检查该列表，以确定它们是否已经完成。插入请求时，计数将增加。完成请求时，计数将减少。该数字不指示 I/O 状态。
- ***load_factor***。内部值，用于指示在该计划程序上感觉到的负载。该值用于确定应该将新任务放在该计划程序上，还是应该放在另一个计划程序上。计划程序表现出负载不平衡时，该值在调试过程中非常有用。在 SQL Server 2008 中，路由决策是基于计划程序的负载来确定的。另外，SQL Server 2008 还使用节点和计划程序的负载系数来协助确定获取资源的最佳位置。向队列中

添加任务时，负载系数将增加。任务完成时，负载系数将减少。使用负载系数，可以帮助 SQLOS 更好地平衡工作负荷。

- *sys.dm_os_workers*。对于系统中的每个工作线程，该视图将返回一行。包括如下一些有用的列。
- *is_preemptive*。值为 1 表示正以抢先计划方式运行工作线程。任何运行外部代码的工作线程都在抢先计划下运行。
- *is_fiber*。值为 1 表示正用轻型池运行该工作线程。
- *sys.dm_os_threads*。该视图将返回所有在 SQL Server 进程下运行的 SQLOS 线程列表。包括如下一些有用的列。
- *started_by_sqlserver*。指示线程的发起方。1 表示 SQL Server 启动该线程；0 表示其他组件启动该线程，例如，SQL Server 中的扩展过程启动该线程。
- *creation_time*。线程的创建时间。
- *stack_bytes_used*。线程上正在使用的字节数。
- *affinity*。即将在该线程上运行的 CPU 掩码。这取决于在 *sp_configure*“关联掩码”中设置的值。
- *locale*。线程的缓存区域 LCID。
- *sys.dm_os_tasks*。该视图将为 SQL Server 实例中的活动任务返回一行。包括如下一些有用的列。
- *task_state*。任务的状态。值可以是下列选项之一。
  - PENDING：正在等待工作线程。
  - RUNNABLE：可运行，但正在接收量程。
  - RUNNING：当前正在计划程序中运行。
  - SUSPENDED：拥有工作线程，但正在等待事件。
  - DONE：完成。
  - SPINLOOP：等待信号时，正在处理自旋锁（spinlock）。
- *context_switches_count*。该任务完成时的计划程序上下文切换数。
- *pending_io_count*。该任务执行的物理 I/O 数。
- *pending_io_byte_count*。该任务执行的总 I/O 字节数。
- *pending_io_byte_average*。该任务执行的平均 I/O 字节数。
- *scheduler_id*。父计划程序的 ID。这是该任务的计划程序信息的句柄。
- *session_id*。与任务关联的会话 ID。
- *sys.dm_os_waiting_tasks*。该视图将返回正在等待某些资源的任务的等待队列的相关信息。包括如下一些有用的列。
- *session_id*。与任务关联的会话 ID。
- *exec_context_i*。与任务关联的执行上下文的 ID。
- *wait_duration_ms*。等待类型的总等待时间，单位是 ms。该时间包括 *signal_wait_time*。
- *wait_type*。等待类型的名称。
- *resource_address*。任务等待的资源地址。
- *blocking_task_addres*。当前持有该资源的任务。
- *blocking_session_id*。阻塞任务的会话的 ID。
- *blocking_exec_context_id*。正在阻塞任务的执行上下文的 ID。
- *resource_description*。对正在占用的资源的描述。

1.5.3　专用管理员连接（DAC）

在某些极端情况下（如完全缺乏可用的资源），SQL Server 可能会进入非正常状态，在该状态下，任何连接都无法连到 SQL Server 实例中。在 SQL Server 2005 之前的版本中，这种情况意味着管理员无法终止任何棘手的连接，甚至无法诊断问题的原因。SQL Server 2005 引入了名为 *DAC* 的特殊连接，即使无法进行其他访问，也可以访问 DAC。

通过 DAC 访问需要进行特别请求。可以使用命令行工具 SQLCMD 连到 DAC，并指定*-A*（或*/A*）标记。推荐使用该方法，因为与图形用户界面（GUI）相比，它占用的资源较少。通过 Management Studio 可以使用 DAC 指定需要的连接，具体方法是：在连接对话框中将 *ADMIN:*放在 SQL Server 名称之前。

例如，在我的机器上连到默认的 SQL Server 实例 TENAR 中，我们输入 **ADMIN:TENAR**。在相同的机器上还需要连到名为 SQL2008 的实例，我们输入 **ADMIN:TENAR\SQL2008**。

DAC 是一种特殊用途的连接，在 SQL Server 中用于诊断问题，并且可能解决问题。这并不是说它可以作为普通用户连接。如果已经存在一个活动的 DAC 连接，任何试图使用 DAC 进行连接的操作都将产生错误。客户端返回的消息仅说明该连接被拒绝，并不明确说明错误，因为已经存在一个活动的 DAC。不过，错误消息将写入错误日志中，指示获取第二个 DAC 连接的尝试操作失败。可以通过以下查询检查是否正在使用 DAC。如果存在一个活动的 DAC，查询将返回 DAC 的 SPID；否则，将返回空行。

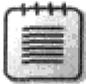
```
SELECT s.session_id
FROM sys.tcp_endpoints as e JOIN sys.dm_exec_sessions as s
   ON e.endpoint_id = s.endpoint_id
WHERE e.name='Dedicated Admin Connection';
```

使用 DAC 时需要注意以下几点。

- 默认情况下，DAC 只能在本地使用。不过，管理员也可以配置 SQL Server，通过名为 *Remote Admin Connections* 的配置选项进行远程连接。
- 通过 DAC 的用户登录连接必须是 *sysadmin* 服务器角色成员。
- DAC 只对少量的 SQL 可执行语句进行限制（例如，不能使用 DAC 运行 *BACKUP* 或 *RESTORE* 语句）。不过，推荐您不要使用任何需要消耗大量资源的查询，因为这些查询会使 DAC 需要解决的问题进一步恶化。创建 DAC 连接的主要目的是用于故障排除和诊断。通常，您可以使用 DAC 来运行查询，而不是使用动态管理对象，其中的某些内容我们已经介绍过，更多详细内容我们将在本书稍后部分讨论。
- 通过为 DAC 分配特殊的线程，允许它在单独的计划程序上执行诊断函数或查询。不能终止该线程，如果需要，您只能终止 DAC 会话。DAC 计划程序总是将 *scheduler_id* 的值设置为 255，因此该线程具有最高优先级。DAC 没有惰性编写器，但 DAC 却有自己的 IOCP、工作线程和闲置线程。

使用 DAC 未必一定能完成预期任务。假设有一个闲置连接正持有一个锁。如果该连接没有活动的任务，则没有线程与之关联，仅存在一个连接 ID。更进一步，假设大量其他进程正试图访问锁定的资源，这些线程将被阻塞。但那些连接仍然保留一个未完成的任务，因此它们并不释放线程。如果诸如 255 之类的进程（默认的工作线程号）试图获取相同的锁定，也许会用尽所有可用的工作线程，因此其他连接将无法连到 SQL Server 中。因为 DAC 拥有自己的计划程序，因此您可以启动这些计划程序，预期的解决方案将终止那些持有锁的连接，但无法进一步释放锁。如果试图使用 DAC 来终止持有锁的进程，操作将

失败。因为 SQL Server 需要工作线程来终止该连接，但已经没有了可用的工作线程。唯一的解决方案是终止几个仍然关联工作线程的阻塞进程。



注意:

为了保护资源，SQL Server 2008 Express 版本不支持 DAC 连接，除非启动时使用了跟踪标志 7806。

不能保证 DAC 一定可用，但由于它预留内存和使用专用计划程序，并且作为单独的节点执行，因此当使用其他方式无法连接时，也许可以考虑使用 DAC 连接。

1.6 内存

内存管理是一个庞大的主题，涉及内存的所有细节也许需要用一整本书来讲述。本节的介绍目的主要有两个：首先，提供 SQL Server 如何使用内存资源的有关信息，使您可以确定自己系统上的内存是否管理得当；其次，描述可以控制的内存管理内容，使您能掌握何时执行相应的内存控制。

默认情况下，SQL Server 2008 几乎完全采用动态方式来管理内存资源。分配内存时，SQL Server 必须不断地与系统进行通信，这也是 SQLOS 引擎层非常重要的原因之一。

1.6.1 缓冲池与数据缓存

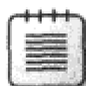
SQL Server 中的主要内存组件是缓冲池。没有被其他组件使用的所有内存都保留在缓冲池中，缓冲池用于从磁盘上的数据库文件中读取页面数据缓存。缓冲管理器用于管理磁盘 I/O 函数，负责向数据缓存中引入数据和索引页，以便其他用户共享数据。缓冲是内存中的页面，其大小与数据页或索引页相同。可以把缓冲看成是一个页面框架，该页面框架可以从数据库中获取页面。大多数其他内存组件使用的来自缓冲池的缓冲区都进入其他类型的内存缓存中，其中最大的是过程和查询计划缓存，通常称为*计划缓存*。

有时候，SQL Server 必须请求大于 8KB 的连续内存块，而缓冲池只能提供 8KB 的页面，因此必须从缓冲池外部分配内存。使用大型内存块通常遵循最小化原则，因此可以直接请求操作系统获取小部分 SQL Server 内存使用量。

1.6.2 访问内存中的数据页

访问数据缓存中的页面时，速度非常快。当拥有上千兆数据时，即使使用实际的内存，扫描整个数据缓存获取某个页面的效率也非常低。为了快速访问，数据缓存中的页面将采用哈希存储方法。*哈希*是一种技术，它通过哈希函数在哈希存储桶（hash bucket）集合之间均匀地映射关键字。*哈希表*是内存中的一种结构，它包含一组指向缓存页的指针（实现为链接列表）。如果所有指向哈希页面的指针无法满足单个哈希页，可以使用*链接列表*链指向其他哈希页。

给定一个 *dbid-fileno-pageno* 标识符（包括数据库 ID、文件号和页号），哈希函数把该关键字转换成哈希存储桶；哈希存储桶实质上是特定页需要的索引。通过使用哈希技术，即使存在庞大的内存，SQL Server 通过几秒钟的读操作就可以查找缓存中的特定数据页。类似地，SQL Server 仅需要较少的内存读取就可以确定目标页不在缓存中，因此必须从磁盘读取目标页。



注意：

通过哈希存储桶链（链接列表）查找某个数据页也许需要访问多个缓冲区。哈希函数尽量在可用的哈希存储桶上均匀分布 *dbid-fi leno-pageno* 值。哈希存储桶的数目由 SQL Server 设置，具体数目取决于缓冲池的总容量。

1.6.3 管理数据缓存中的页面

只有数据页或索引页在内存中时，您才可以使用它们。因此，数据缓存中的缓冲区必须可用来读取页面。使一批缓冲区可以即时使用是一种重要的性能优化。如果缓冲区不能随时可用，系统也许需要搜索大量的内存页来查找缓冲区，使该缓冲区释放空间而成为工作区。

在 SQL Server 2008 中，采用单一机制既负责向磁盘写入更改过的页面，又负责将那些在一段时间内未被引用的页面标记为空闲页。SQL Server 维护空闲页的地址链接列表，需要缓冲页的任何工作线程将使用该链接列表的第一页。

数据缓存中的每个缓冲区都有一个表头，该表头包含该页最后两次被引用的相关信息和某些状态信息，包括该页是否是脏页（即读入磁盘后该页被更改过）。引用信息用于实现数据缓存页的页面替换策略，它使用一种名为 *LRU-K* 的算法。该算法是由 Elizabeth O'Neil、Patrick O'Neil 和 Gerhard Weikum（在 1993 年 5 月的 ACM SIGMOD Conference 会议录中）共同引入的。与最近最少使用（LRU）替换策略相比，该算法有很大的改进，LRU 算法不考虑最近使用的页如何使用。与涉及引用次数的最不常使用（LFU）策略相比，LRU-K 也有较大改善，因为它需要更少的引擎调整和更少的记账开销。LRU-K 算法跟踪页面最近 *K* 次的引用记录，并可以区分不同类型的页面（如索引页和数据页）和不同的频率级别。实际上，该算法可以模拟在特意调整大小的不同缓冲池中分配页面的效果。SQL Server 2008 使用的 *K* 值为 2，因此它将跟踪每个缓冲页最近两次的访问记录。

定期扫描数据缓存的操作将贯穿始终。因为缓冲区缓存都在内存中，这些扫描操作速度非常快，并且不需要进行 I/O 操作。在扫描过程中，每个缓冲区的相关值取决于缓冲区的使用记录。当该值变得足够低时，系统将检查脏页指示符。如果该页是脏页，则执行写入操作，以便向磁盘写入修改的数据。SQL Server 实例使用预写日志，因此将修改数据的日志页首先写入磁盘时，脏数据页的写入操作将被阻塞（我们将在第 4 章进一步深入讨论日志的相关内容）。修改的页在磁盘中刷新以后，或者如果该页不是脏页，则该页将被释放。通过从哈希表中删除缓冲区的相关信息，缓冲区页和数据页之间的关联也将被删除，该缓冲区将列入可用表中。

使用 LRU-K 算法时，有价值的缓冲区持有页将保留在活动缓冲池中；相反，如果缓冲区持有页的引用频率不够高，这些缓冲区将逐渐返回到可用缓冲区列表中。SQL Server 实例将基于缓冲区缓存的大小内部确定可用缓冲区列表的大小，而无法配置可用缓冲区列表的大小。

1.6.4 可用缓冲区列表和惰性编写器

单独的工作线程在计划一个异步读操作之后并在完成该读操作之前，将主要执行扫描缓冲池、写入脏页和填充可用缓冲区列表的工作。工作线程将从 SQL Server 数据库引擎中心的数据结构中获取包含 64 个缓冲区的缓冲池的段地址。读操作被初始化以后，工作线程将检查可用列表是否太小（注意，该进程本身的读操作将消耗 1 个或更多个列表页）。如果可用列表太小，工作线程将搜索缓冲区以释放空间，无

论在该组的 64 个缓冲区中实际释放了多少空间，工作线程都将检查所有 64 个缓冲区。如果在扫描部分存在脏缓冲区而必须执行写操作，那么还需要计划写操作。

每个 SQL Server 实例还为每个 NUMA 节点准备了一个名为*惰性编写器*的线程（每个实例至少含有一个），该线程将扫描与 NUMA 节点有关的缓冲区缓存。惰性编写器线程在特定的时间段处于休眠状态；唤醒时它将检查可用缓冲区列表的大小。如果列表的大小低于某个阈值（该阈值具体取决于缓冲池的总大小），惰性编写器线程将扫描缓冲池来重新填充可用列表。向可用列表添加缓冲区时，如果这些缓冲区是脏缓冲区，还需将它们写入磁盘。

SQL Server 动态使用内存时，必须不断侦听可用内存的数量。每个节点的惰性编写器将定期查询系统，以确定可用的物理内存数量。惰性编写器用于扩大或缩小数据缓存，使操作系统的可用物理内存保持在 5MB（偏移量在 200KB 左右）大小，以防止分页。如果操作系统的可用内存小于 5MB，惰性编写器将释放内存给操作系统，而不是将自己添加到可用列表中。如果可用的物理内存大于 5MB，惰性编写器通过将自身添加到可用列表中来向缓冲池重新申请内存。只有当惰性编写器重新填充可用列表时，它才向缓冲池重新申请内存；处于休眠状态的服务器不会增加自身的缓冲池。

如果 SQL Server 检测到过多的分页，它也向操作系统释放内存。通过使用 SQL Server 的某种追踪机制来监视服务器内存更改事件（在服务器事件目录中），可以通知 SQL Server 何时增加或减少自身的总内存量。每当 SQL Server 中的内存增加或减少 1MB，或达到最大服务器内存的 5%时，系统都将生成事件。可以通过查看名为 *Event Sub Class* 的数据元素值来了解变化情况。*Event Sub Class* 的值为 1 表明内存增加了，值为 2 表明内存减少了。我们将在第 2 章详细讨论追踪的相关内容。

1.6.5 检查点

检查点进程也定期扫描缓冲区缓存，并为特定的数据库向磁盘写入脏数据页。检查点进程与惰性编写器（或管理页面的工作线程）的区别在于，检查点进程不会在可用列表上添加缓冲区。检查点进程的唯一目的是，确保把某一时刻之前写入的页面都写入磁盘中，以便使内存中的脏页数始终保持最小。出现故障时，它还能确保 SQL Server 所需的数据库恢复时间保持最短。在某些情况下，如果工作线程或惰性编写器在两个检查点之间已经将大部分脏页写入了磁盘，检查点也许还会发现少量需要写入磁盘的脏页。

出现检查点时，SQL Server 把检查点记录到事务日志中，事务日志将列出所有活动的事务。这样，恢复进程可以构建包含所有潜在脏页列表的表。检查点可以定期自动出现，也可以手动请求。

出现以下情况时会触发检查点。

- 数据库所有者（或备份操作员）明确发出 *CHECKPOINT* 命令在该数据库中执行某个检查点。在 SQL Server 2008 中，使用 *CHECKPOINT* 命令可以同时运行多个检查点(在不同的数据库中)。
- 日志正在变满（超过容量的 70%）及数据库处于自动截断模式（我们将在第 4 章为您详细介绍自动截断架构）。触发检查点可以截断事务日志并释放空间。不过，如果没有空间可释放（也许是由于长时间运行事务），则不会出现检查点。
- 预计需要较长的恢复时间。如果恢复时间比“恢复间隔”配置选项的预期时间要长，将触发检查点。SQL Server 2008 使用简单的度量标准来预测恢复时间，因为与运行原始操作相比，它可以在较短时间内进行恢复和重做。因此，如果出现检查点的次数与恢复间隔频率相同，恢复操作将在间隔内完成。如果恢复间隔的时间设置为 1，意味着数据库只要在处理事务，检查点每分钟将出现一次。为了自动激活检查点，必须完成的最少的工作量一般是每分钟 10MB 日志。

这样，SQL Server 不会在闲置的数据库上浪费时间来使用检查点。默认的恢复间隔是 0，它表示 SQL Server 将选择合适的值；对当前版本而言，该值是 1 分钟。

- 请求正常关闭 SQL Server，并且不使用 NOWAIT 选项。然后，检查点操作将在实例上的每个数据库中运行。当显示关闭 SQL Server 时，则为正常关闭（除非使用 *SHUTDOWN WITH NOWAIT* 命令完成该操作）。当通过服务控制管理器或使用操作系统提示符的网络停止命令来终止 SQL Server 服务时，也为正常关闭。

此外，也可以使用 *sp_configure* 恢复间隔选项来改变检查点的频率，以平衡恢复时间与对运行时性能的影响。在实际出现检查点时，如果对跟踪感兴趣，可以使用 SQL Server 扩展事件的 *sqlserver.checkpoint_begin* 和 *sqlserver.checkpoint_end* 来监视检查点的活动情况（有关扩展事件的详细信息请参考本书第 2 章）。

检查点进程通过缓冲池时按无序方式扫描页面。当它找到某个脏页时，将进一步查看相邻的页面（磁盘上）是否也存在一些脏页，以便它进行大块写入操作。例如，当检查点进程发现缓冲区 14 是脏页时，将对缓冲区 14、200、260 和 1000 执行写入操作（虽然这些页面在缓冲池中距离很远，但它们也许具有相邻的磁盘位置。这样，可以对缓冲池中非相邻的页面执行名为 *gather-write* 的单个写入操作）。进程将继续扫描缓冲池，直到它到达页面 1000。在某些情况下，已经写入的页面也可能会再次成为脏页，因此也许需要再次将它写入磁盘。

有时候，检查点也许会启动大量 I/O 操作，使 I/O 子系统不断获取写入操作，这将严重影响读取性能。另一方面，在一段时期也可能存在相对较低的 I/O 活动。SQL Server 2008 含有命令行选项，通过该选项，您可以对检查点 I/O 操作进行限制。您可以使用 SQL Server 配置管理器，并向 SQL Server 服务的启动参数列表中添加 -*k* 参数，其后再跟上小数。指定的值表示检查点进程每秒可以写入的兆字节数。通过使用 -*k* 选项，可以分散检查点的 I/O 开销，并更好地控制效果。请注意，默认情况下检查点进程将确保 SQL Server 可以在指定的恢复间隔内恢复数据库。如果启用 -*k* 选项，默认行为将发生改变，如果指定的参数值很低，将导致较长的恢复时间。由于存在检查点进程，备份初始化也将延迟，因此完成备份的时间也许会稍微长一些。在产品系统上启用该选项之前，应该确保有足够的硬件来支持 SQL Server 提交的 I/O 请求，并在系统上对应用程序进行全面测试。

1.6.6 管理其他缓存中的内存

数据缓存不使用的缓冲池内存可用于其他缓存类型，主要可用于计划缓存。页面替换策略和可用页面的搜寻机制与数据缓存都有较大区别。

除数据缓存之外，SQL Server 2008 的所有缓存都使用共同的缓存框架。该缓存框架包括存储集合和资源监视器。储存类型有 3 种：缓存存储、用户存储（该存储类型实际上与用户无关）和对象存储。缓存储存的主要示例是计划缓存，用户存储的主要示例是元数据缓存。缓存存储和用户存储使用相同的 LRU 机制和相同的成本计算算法来确定保留哪些页面和释放哪些页面。另一方面，对象存储只是一种内存池阻塞，不需要 LRU 机制或成本计算算法。SNI 是对象存储的一个使用示例，它使用对象存储来共用网络缓冲区。在本节的剩下部分，我们讨论的存储指缓冲存储和用户存储。

存储使用的 LRU 机制是时钟算法的直接变体。假设时钟指针在存储区扫描，查找每个条目；当它接触每个条目时，将减少开销。当条目的开销达到 0 时，可以将该条目从缓存中删除。每次重新使用某个条目时，都需要重新设置开销。

存储中的内存管理需要同时考虑全局和局部内存管理策略。全局策略考虑系统上的总内存并在所有缓存之间启用时钟算法。局部策略涉及查看一个存储区或单独的缓存，并确保它没有使用不相称的

内存量。

为了满足全局和局部策略，SQL Server 存储实现两种指针：外部指针和内部指针。每个存储都有两个时钟指针，通过查找 DMV *sys.dm_os_memory_cache_clock_hands* 来查看这些指针。对于每个缓存存储或用户存储，该视图都包含一个内部时钟和一个外部时钟。外部时钟指针实现全局策略，而内部时钟指针实现局部策略。资源监视器观察到内存压力时，负责移动外部指针。内存压力有各种类型，检测和故障诊断内存问题的具体内容超出了本书的范围。但是，如果查看 DMV *sys.dm_os_memory_cache_clock_hands*，特别是查看 *removed_last_round_count* 列，可以看到一个与其他值相比特别大的值。如果该值在急剧增加，那么这是出现内存压力的显著标志。本书的随附网站包含一个名为“在 SQL Server 2008 中故障诊断性能问题”的详细白皮书，它包含许多有关跟踪和处理内存问题的详细信息。

当单个缓存需要调整时，内部时钟将移动。与其他缓存相比，SQL Server 试图使每个缓存保持适当的容量。内部时钟指针的移动仅用于响应活动。如果运行访问缓存任务的工作线程观测到缓存中含有大量条目，或观测到缓存的大小大于某一内存百分率，则该缓存的内部时钟指针就开始为自身释放内存。

Memory Broker

因为 SQL Server 中的大量组件都需要内存，所以为了确保每个组件都有效地使用内存，SQL Server 使用 Memory Broker，它的职责是分析 SQL Server 中与内存消耗有关的行为，并改善动态内存分配。Memory Broker 是一种在缓冲池、查询执行器、查询优化器和各种缓存之间动态分配内存的集中式机制，它还尝试为各种工作负荷改编分配算法。可以将 Memory Broker 看成是一种带有反馈循环的控制机制。它使用组件监视内存的需求和消耗，并使用自身收集的信息来计算所有组件之间的最佳内存分配。它将收集的信息向其他组件广播，然后其他组件使用该信息来调整自身的内存使用情况。可以通过查询 Memory Broker 信号缓冲区来监视 Memory Broker 的行为，如下所示：

```
SELECT * FROM sys.dm_os_ring_buffers
WHERE ring_buffer_type =
'RING_BUFFER_MEMORY_BROKER';
```

只有 Memory Broker 需要指定组件的行为发生变化时，才对 Memory Broker 的信号缓冲区进行更新，即增加、缩小或保持不变（如果它之前已经增加或收缩了）。

1.6.7 调节内存大小

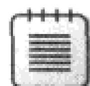
我们讨论 SQL Server 内存时，实际上并不只是指缓冲池。SQL Server 内存实际上由 3 部分组成，而缓冲池通常最大，使用频率也最高。缓冲池是一个 8KB 的缓冲区集合，因此任何大于 8KB 的内存块都需要进行单独管理。

名为 *sys.dm_os_memory_clerks* 的 DMV 含有名为 *multi_pages_kb* 的列，它显示缓冲区之外的内存组件使用的空间大小：

```
SELECT type, sum(multi_pages_kb)
FROM sys.dm_os_memory_clerks
WHERE multi_pages_kb != 0
GROUP BY type;
```

如果 SQL Server 实例配置为使用地址窗口化扩展（AWE）内存，可以将它看成是 1/3 内存区域。AWE 是一种 API，通过使用它，32 位应用程序可以访问超过 32 位地址限制的物理内存。虽然可以将 AWE 看

成是缓冲池的一部分，但必须对它进行单独跟踪，因为只有数据缓存页面才可以使用 AWE 内存，其他内存组件（如计划缓存）都不能使用 AWE 内存。



注意：

如果启用了 AWE，那么获取有关 SQL Server 实际内存消耗信息的唯一方式是使用指定的 SQL Server 计数器或服务器内部的 DMV，您无法从操作系统级的性能计数器中获取该信息。

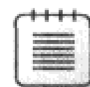
1.6.8 调节缓冲池大小

SQL Server 启动时，它将计算 SQL Server 进程的虚拟地址空间（VAS）的大小。每个在 Windows 上运行的进程都拥有自己的 VAS。可供进程使用的所有虚拟地址集合构成了 VAS 的大小。VAS 的大小取决于体系结构和操作系统。VAS 仅是所有可能地址的集合，它也许比机器上的物理内存大得多。

32 位机器只能对 4GB 的内存进行直接寻址，默认情况下，Windows 将保留顶部的 2GB 地址空间供自身使用，只保留 2GB 空间作为 VAS 的最大尺寸供其他应用程序使用（如 SQL Server)。可以在系统的 Boot.ini 文件中启动*/3GB* 标志来增加该地址空间，从而允许应用程序使用的 VAS 达到 3GB。如果系统的 RAM 大于 3GB，启用 AWE 是 32 位机器可以访问该 RAM 的唯一方式。在 SQL Server 2008 中使用 AWE 的一个优势是，可以将通过 AWE 机制分配的内存页看成是锁定页，不会被置换出来。

在 64 位平台上，提供启用配置选项的 AWE，但忽略其设置。不过，虽然 Windows 策略选项“锁定内存页”在默认方式下是禁用的，但可以启用它。该策略用于确定哪些账户可以利用 Windows 功能，从而使数据保留在物理内存中，阻止系统在磁盘上的虚拟内存中进行数据分页。推荐在 64 位系统上启用该策略。

在 32 位操作系统上使用 AWE 时，必须启用锁定内存页选项。如果不需要使用 AWE，则不推荐您在内存选项中启用该选项。虽然 SQL Server 在未启用 AWE 时将忽略该选项，但系统上的其他进程可能会受影响。



注意：

64 位机器上的内存管理更加简单。对 SQL Server 而言，它可以使用更多的 VAS；对管理员而言，他不必担心特定的操作系统标准，甚至不需要考虑是否启用 AWE。除非使用非常小的数据库，不需要使用上千兆字节的 RAM，否则，都应该考虑运行 64 位版本的 SQL Server 2008。

表 1-1 展示了各种 SQL Server 2008 版本可能的内存配置信息。

表 1-1　SQL Server 2008 内存配置

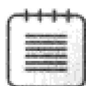
配　置	VAS	最大物理内存	AWE/锁定内存页支持
使用*/3GB* 启动参数的 32 位操作系统上的本机 32 位	2GB 3GB	64GB 16GB	AWE AWE
X64 操作系统上的 32 位（Windows on Windows）	4GB	64GB	AWE
X64 操作系统上的本机 64 位	8TB	1TB	锁定页
IA64 操作系统上的本机 64 位	7TB	1TB	锁定页

除了 VAS 大小，SQL Server 还计算一个名为 *Target Memory* 的值，该值是 SQL Server 预期能分配的 8KB 页面数。如果已经设置了“最大服务器内存配置”选项，则目标内存是两个值中的较小值。目标内存将定期被重新计算，特别是它从 Windows 获取内存通知时。正常加载的服务器上目标页数目的减少也许是对外部物理内存压力的一种响应。可以通过性能监视器查看目标页的数目（在 *SQL Server：Memory Manager* 对象下查看目标服务器页面的计数器）。另外，名为 *sys.dm_os_sys_info* 的 DMV 也包含一行通用的 SQL Server 配置信息，包括以下列。

- ***physical_memory_in_bytes***。可用的物理内存量。
- ***virtual_memory_in_bytes***。用户模式中进程可用的虚拟内存量。可以通过该值确定 SQL Server 是否使用 3-GB 开关启动。
- ***bpool_commited***。缓冲区总数和具有相关内存的页面，不包含虚拟内存。
- ***bpool_commit_targ***。缓冲池中缓冲区的最佳数目。
- ***bpool_visible***。缓冲池中的 8KB 缓冲区数目，可以在进程虚拟地址空间中直接获取它们。不使用 AWE 时，如果缓冲池已经获取自身的内存目标（*bpool_committed = bpool_commit_target*），那么 *bpool_visible* 的值将等于 *bpool_committed*。如果在 32 位版本的 SQL Server 上使用 AWE，那么 *bpool_visible* 代表 AWE 映射窗口的大小，该窗口用于访问由缓冲池分配的物理内存。映射窗口的大小受进程地址空间的约束，因此可见的数量比提交的数量要小，并且可以通过内部组件进一步减小。如果 *bpool_visible* 的值太小，也许会收到内存溢出的错误信息。

虽然保留了 VAS，但只有 SQL Server 实例正在处理的当前工作负荷所需的内存达到目标数量时，才提交目标数量的物理内存。为了支持工作负荷，实例需要继续获取的物理内存量取决于用户连接和被处理的请求。SQL Server 实例可以继续提交物理内存，直到它达到目标或操作系统指示不存在可用内存为止。当操作系统通知 SQL Server 缺少可用的内存时，如果 SQL Server 实例拥有的内存大于最小服务器内存的配置值，它将释放内存。注意，SQL Server 最初提交的内存不等于最小服务器内存。它只提交自身需要和内存可以提供的内存量。只有缓冲池大小大于最小服务器内存值时，最小服务器内存值才起作用，然后，SQL Server 不允许内存低于该设置。

当其他应用程序在运行 SQL Server 实例的计算机上启动时，它们需要消耗内存，SQL Server 也许需要调整自身的目标内存。通常，只有在这种情况下目标内存才小于提交内存，并且该状态将一直保持到可以释放内存为止。可能的话，SQL Server 实例将调整自身的内存开销。如果其他应用程序已经停止，更多内存变为可用状态，SQL Server 实例将增加自身的目标内存值，并根据需要允许增加内存分配。只有存在压力时，SQL Server 才调整自身目标并释放物理内存。因此，当系统变为静止状态时，暂时繁忙的服务器才可以提交大量不需要释放的内存。



注意：

在相同的机器上不存在特殊的多 SQL Server 实例处理方法，也无法在所有实例之间均衡内存。它们都需要竞争相同的物理内存，因此需要确保没有实例缺少物理内存。应该在多实例机器上的所有 SQL Server 实例中使用最小和最大服务器内存选项。

1. 观察内存的内部机制

SQL Server 2008 包含若干个动态管理对象，以提供有关内存和各种缓存的信息。例如，动态管理对

象包含有关计划程序的信息，这些信息主要供客户支持服务部门使用，用于查看 SQL Server 的工作情况，但您也可以通过它们了解 SQL Server 的工作情况。为了选择这些对象，必须具备查看服务器状态（View Server State）的权限。在此，我们再次列出每个对象的一些非常有用的列，这些描述大多来自 *SQL Server 联机丛书*。

- ***sys.dm_os_memory_clerks*。** 该视图针对 SQL Server 实例中正在活动的每个内存 clerk 返回一行。可以将每个 clerk 看成是账户单位。每个早期描述的存储就是一个 clerk，但某些 clerk 不是存储（如 CLR clerk 和全文本搜索 clerk）。以下查询将返回所有类型的 clerk 清单：

```
SELECT DISTINCT type FROM sys.dm_os_memory_clerks;
```

包括如下一些有用的列。

- ***single_pages_kb*。** 分配的单页内存量，单位为 KB。这是使用内存节点的单页分配器分配的内存量。该单页分配器直接从缓冲池窃取页。
- ***multi_pages_kb*。** 分配的多页内存量，单位为 KB。这是使用内存节点的多页分配器分配的内存量。该内存在缓冲池外分配，并利用了内存节点的虚拟分配器的优点。
- ***virtual_memory_reserved_kb*。** 内存 clerk 保留的虚拟内存量。这是由使用该 clerk 的组件直接保留的内存量。在大多数情况下，只有缓冲池才能通过使用其内存 clerk 来直接保留 VAS。
- ***virtual_memory_committed_kb*。** 内存 clerk 提交的内存量。提交的内存量应该始终小于保留的内存量。
- ***awe_allocated_kb*。** 内存 clerk 使用 AWE 分配的内存量。在 SQL Server 中，只有缓冲池 clerk（MEMORYCLERK_SQLBUFFERPOOL）使用该机制，且仅在启用 AWE 时使用。
- ***sys.dm_os_memory_cache_counters*。** 该视图返回每个用户储存和缓存存储的运行状况快照。它提供有关已分配的缓存条目、缓存条目的使用情况及内存源的运行时信息。包括如下一些有用的列。
- ***single_pages_kb*。** 分配的单页内存量，单位为 KB。这是通过单页分配器分配的内存量。它指从该缓存的缓冲池中直接获取的 8KB 页。
- ***multi_pages_kb*。** 分配的多页内存量，单位为 KB。这是通过多页分配器分配的内存量。该内存在缓冲池外分配，并利用了内存节点的虚拟分配器的优点。
- ***multi_pages_in_use_kb*。** 正在使用的单页内存量，单位为 KB。
- ***single_pages_in_use_kb*。** 正在使用的多页内存量，单位为 KB。
- ***entries_count*。** 缓存中的条目数。
- ***entries_in_use_count*。** 缓存中正在使用的条目数。
- ***sys.dm_os_memory_cache_hash_tables*。** 该视图针对 SQL Server 实例中的每个活动缓存返回一行。可以在 *cache_address* 列上将该视图连到 *sys.dm_os_memory_cache_counters* 中。包括如下一些有用的列。
- ***buckets_count*。** 哈希表中的存储桶数。
- ***buckets_in_use_count*。** 当前使用的存储桶数。
- ***buckets_min_length*。** 存储桶中的最小缓存条目数。
- ***buckets_max_length*。** 存储桶中的最大缓存条目数。
- ***buckets_avg_length*。** 每个存储桶中的平均缓存条目数。如果该数变得非常大，可能说明哈希算法不理想。

- ***buckets_avg_scan_hit_length***。在找到搜索项之前，存储桶中已检查条目的平均数。如上所述，非常大的数可能说明小于最佳缓存。可以考虑运行 *DBCC FREESYSTEMCACHE* 来删除缓存储存中的未用条目。有关该命令的详细信息可以参考 *SQL Server 联机丛书*。
- ***sys.dm_os_memory_cache_clock_hands***。如前文所述，可以使用 *cache_address* 列将 DMV 连到其他缓存 DMV 中。包括如下一些有用的列。
- ***clock_hand***。时钟指针的类型，要么是外部指针，要么是内部指针。记住，每个存储都有两个时钟指针。
- ***clock_status***。时钟指针的状态：挂起或运行。对应的策略生效时时钟指针开始运行。
- ***rounds_count***。时钟指针已经运行的轮数。所有的外部时钟指针在该列具有相同（或相似）的值。
- ***removed_all_rounds_count***。时钟指针在所有轮中删除的条目数。

2. NUMA 和内存

如前文所述，采用 NUMA 的一个主要原因是为了有效处理大量内存。随着时钟速度和处理器数量的增加，减少内存延迟变得日益困难，因此需要使用这种附加的处理功能。大型 L3 缓存可以部分缓解内存延迟问题，但这仅是一种有限的解决方案。NUMA 是可扩展的选择方案。SQL Server 2008 可以充分利用基于NUMA 的计算机优势，而无需对应用程序做任何改变。注意，NUMA 内存节点完全取决于硬件 NUMA 配置。如前文所述，如果定义了自己的软件 NUMA，不会影响 NUMA 内存节点的数目。因此，如果使用带有 8 个 CPU 的 SMP 计算机，并且需要创建 4 个软件 NUMA 节点，每个节点有两个 CPU，那么必须用一个 MEMORY 节点为所有 4 个 NUMA 节点提供服务。软件 NUMA 不提供内存与 CPU 的关联。不过，每个 NUMA（硬件 NUMA 或软件 NUMA）节点都含有网络 I/O 线程和惰性编写器线程。

在含有大量CPU而无硬件NUMA的计算机上使用软件NUMA的主要原因是减少I/O和惰性编写器瓶颈。例如，在含有 8 个 CPU 而无硬件 NUMA 的计算机上，拥有一个 I/O 线程和一个惰性编写器线程，这就存在瓶颈。配置 4 个软件 NUMA 节点，提供 4 个 I/O 线程和 4 个惰性编写器线程，这样能显著提高系统性能。

如果拥有多个 NUMA 内存节点，SQL Server 将把总目标内存在所有节点之间进行平均分配。因此，如果有 10GB 的物理内存和 4 个 NUMA 节点，并且 SQL Server 限度目标内存的值为 10GB，那么所有节点将平均分配目标内存，使用的 2.5GB 内存就像是它们自身的内存一样。事实上，如果其中 1 个节点的内存值小于其他节点的内存值，该节点就会使用其他节点的内存，使自身的内存值达到 2.5GB，该内存称为*外来内存*（foreign memory）。外来内存被看成是本地内存，因此如果 SQL Server 已经重新调整其目标内存，并且每个节点都需要释放一些内存，那么所有节点都不会首先释放外来页。另外，如果已经在可用的 NUMA 节点的子集上配置使用了 SQL Server，则不会对这些节点上的内存自动限制目标内存。必须设置最大服务器内存值来限制内存量。

一般来说，NUMA 节点大部分都能相互独立运行，但也有例外情况。例如，如果在节点 *N1* 上运行的工作线程需要访问已经在节点 *N2* 内存中的数据库页，那么它需要通过访问 *N2* 的内存来完成任务，*N2* 的内存称为*非本地内存*。注意，非本地内存与外来内存不一样。

3. 预读

SQL Server 支持一种称为*预读*的机制，无论是数据需求还是索引页需求都可以进行预读，并且在实际需要页面之前将它们读入缓冲池。这种性能优化可以使我们更有效地处理大数据量数据。预读机制完全是由系统内部管理的，不需进行任何配置和调整。

有两种类型的预读：一种用于堆上的表扫描，一种用于索引范围。

对表扫描而言，为了按磁盘顺序读取表，需要了解该表的分配结构。对于 1 次高达 32 区数（32×8 页/区数×8 192 字节/页=2MB）的预读，效果也许非常显著。1 次 4 区数（32 页）的读取是单个 256KB 的散点读取（scatter read）。如果表分布在文件组的多个文件上，SQL Server 将试图在文件间平均分配预读活动。

对索引范围而言，扫描使用 1 级索引结构（该级立即位于叶级之上）来确定预读哪些页。索引扫描启动时，索引的首个后代将调用预读机制，以最小化执行的读取次数。例如，扫描 *WHERE state = 'WA'*，预读将搜索 *key ='WA'*的索引，这样可以从 1 级节点中辨别出必须检查多少页来满足该扫描。如果预期的页数较小，初始预读将请求所有页；如果页面是非连续区域，它们将执行散点读取。如果该范围包含大量页面，将执行初始预读，此后，每次扫描将读入另外 16 页，读入另外 16 页时需要查询索引。该过程会产生几个有用的效果。

- 每当索引是连续区域时，小范围的预读在数据页级可以通过单个读取进行处理。
- 可以使用扫描范围（例如，*state = 'WA'*）来阻止预读不使用的页，因为在索引中可以利用该信息。
- 只有按照数据页级的页链接进行预读，预读的速度才不会减缓（可以在聚集索引和非聚集索引上实现预读）。

通过以上介绍可以看到，SQL Server 中的内存管理是一个非常庞大的主题，我介绍的这些内容只能使您简要了解 SQL Server 如何使用内存。这些信息应该能够帮助您初步解释通过 DMV 和故障诊断获取的大量信息。随附网站的白皮书提供了更多故障诊断的意见和方案。

1.7　服务器资源调控器

为了使系统良好地运行，拥有充足的内存和可用的计划程序资源是最重要的。虽然 SQL Server 和 SQLOS 含有许多内置算法来均衡分配这些资源，但您通常比 SQL Server 数据库引擎更了解自己的资源需求。

1.7.1　资源调控器概述

SQL Server 2008 企业版本为您提供了一个向进程组分配计划程序和内存资源的接口，分配是建立在您对进程组进行需求分析的基础上。该接口称为*资源调控器*，它具有以下用途。

- 允许监视每个工作负荷的资源使用情况，其中工作负荷可以定义为一组请求。
- 能够启用工作负荷的优先级。
- 通过某种方式指定工作负荷之间的资源边界，以便在可能存在资源竞争的地方预测执行这些工作负荷。
- 阻止或降低失控查询的可能性。

资源调控器的功能建立在工作负荷和资源池的基础上，资源池由 DBA 构建而成。只需使用一些基本的 DDL 命令，您就可以定义一组工作组，创建分类器函数来确定哪些用户会话是哪些组的成员，并创建资源池，使每个工作负荷组都拥有最小和最大内存量设置，以及它们可以使用的 CPU 资源百分比。

图 1-4 阐述了应用于每个会话、工作负荷组和资源池的分类器函数之间的示例关系。本节介绍了更多有关组和池的详细内容，但您可以在图中看到，新会话根据分类器函数的分类结果而放在不同的工作负荷组中。另外还需要注意的是，组和池之间具有多对一关系。许多工作负荷组可以分配到相同的池中，但每个工作负荷组仅属于一个池。

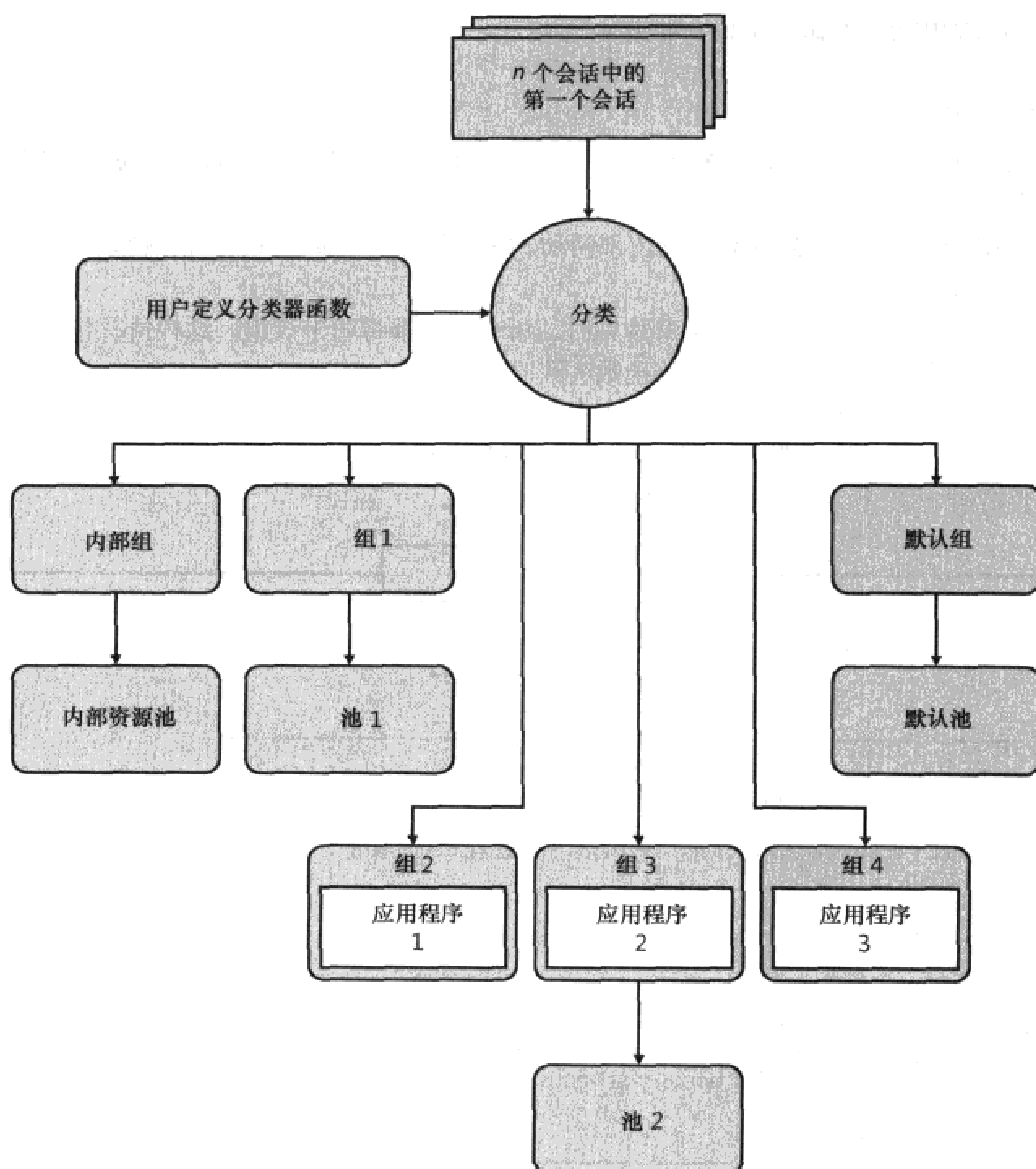


图 1-4 资源调控器组件

1. 启用资源调控器

使用 DDL 语句 *ALTER RESOURCE GOVERNOR* 启用资源调控器。使用该语句指定某个分类器函数向工作负荷分配会话，可以启用或禁用资源调控器，或者重新设置保存在资源调控器上的统计信息。

2. 分类器函数

定义分类器函数并启动资源调控器以后，每个新会话都将使用该函数来确定会话分配的工作负荷组名称。除非会话被显式分配到某个不同的组中，否则该会话将一直保留在相同的组中，直到会话终止为止。在任何指定的时间内，只能存在一个分类器函数的最大值，如果没有定义分类器函数，所有的会话都将被分配到默认组中。分类器函数通常是基于连接属性的，并基于系统函数（如 *SUSER_NAME()*、*SUSER_SNAME()*、*IS_SRVROLEMEMBER()* 和 *IS_MEMBER()*）和属性函数（如 *LOGINPROPERTY* 和

CONNECTIONPROPERTY）确定工作负荷组。

3. 工作负荷组

工作负荷组是按照 DBA 定义的名称，以便多个连接共享相同的资源。SQL Server 实例中有两种预定义工作负荷组。

- **内部组**。该组用于 SQL Server 的内部活动。用户不能向内部组添加会话或影响其资源使用情况，但是可以监视内部组。
- **默认组**。当无法使用其他分类器规则时，所有会话都将归于该组。这种情况包括分类器函数产生一个不存在的组，或者分类器函数失败时。

可以将大量会话分配到相同的工作负荷组中，并且每个会话可以启动多个连续的任务（或批处理）。每个批处理可以由多个语句组成，并且其中的某些语句（如存储过程调用）还可以进一步被分解。图 1-5 阐述了工作负荷组、会话、批处理和语句之间的这种关系。

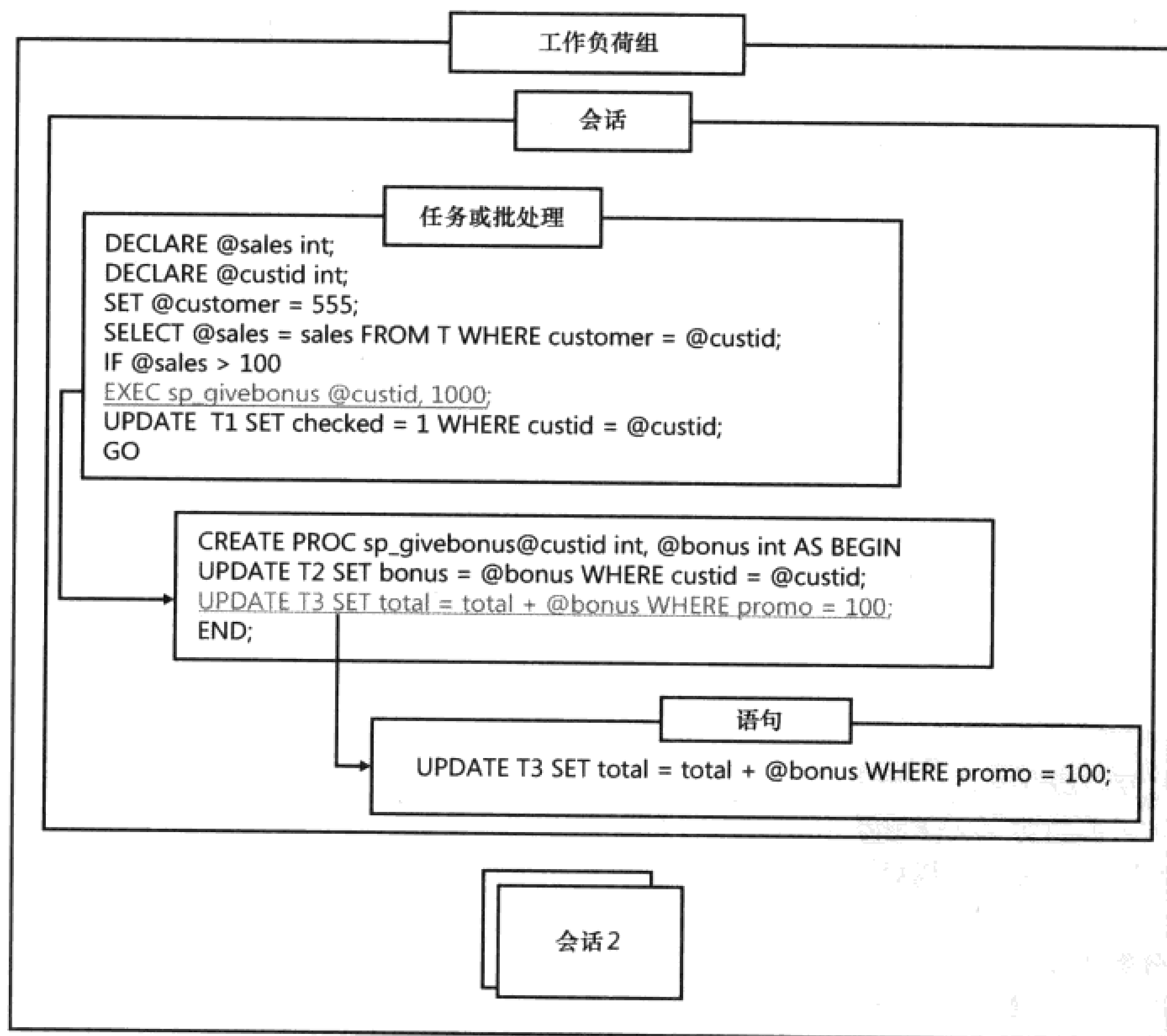


图 1-5　工作负荷组、会话、批处理和语句

创建工作负荷组时，需要为它取名，然后为 6 个特定的组属性提供值。任何不指定的属性将具有默认值。除了组属性之外，还需要将组分配到某个资源池中；如果没有指定资源池，将启用默认组。可以

指定的 6 个属性如下。

（1）IMPORTANCE。每个工作负荷组在它们的资源池中的重要性可以为 *low*、*medium*、*high*。默认值是 medium。该值用于确定 CPU 可用带宽与预设置比例中组的比（在未来版本或服务包中，该比率可能会改变）。当前的权值 low=1、medium=3、high=9。这表明计划程序试图从高优先级工作负荷组执行可运行会话的次数是从中等优先级工作负荷组执行会话次数的 3 倍，是从低优先级工作负荷组执行会话次数的 9 倍。DBA 的任务是确保组中没有过多的高优先级会话，或者不向过多的组中分配高优先级。如果从高优先级工作负荷组中获取会话的数目是低优先级对应数目的 9 倍，最终将导致所有的会话在计划程序上获取同样的时间。

（2）REQUEST_MAX_MEMORY_GRANT_PERCENT。该值用于指定来自该组的单个任务可以从资源池获取的最大内存量。该值是由池的 MAX_MEMORY_PERCENT 值指定的、相对于池大小的百分比，而不是正在使用的实际内存量。该数量仅指用于查询执行的内存，不用于数据缓存或缓存的计划，数据缓存或缓存的计划可以由许多请求共享。REQUEST_MAX_MEMORY_GRANT_PERCENT 的默认值是 25%，表示单个请求可以使用 1/4 的池内存。

（3）REQUEST_MAX_CPU_TIME_SEC。该值是工作负荷组中任意一个请求可以使用的最大 CPU 时间量，单位为秒。默认设置是 0，表明 CPU 时间无限制。

（4）REQUEST_MEMORY_GRANT_TIMEOUT_SEC。该值是查询等待资源变为可用的最大时间，单位为秒。如果资源没有变为可用，它将产生超时错误（在某些情况下，该查询也许不会失败，但在运行时，资源也许会急剧减少）。默认值为 0，表明服务器将基于查询开销计算超时。

（5）MAX_DOP。该值用于指定某个并行查询的最大并行度（DOP），并且该值优先于最大并行配置选项和任何查询提示。实际的运行时 DOP 还受计划程序数和并行线程可用性的限制。MAX_DOP 设置仅存在最大限制，意味着允许服务器使用比指定并行度更少的进程来运行查询。默认设置为 0，表明服务器使用全局设置。使用 MAX_DOP 值时，需要了解以下详细信息。

> 只要 MAXDOP 不超过工作负荷组的 MAX_DOP 值，系统就优先使用作为查询提示的 MAXDOP。
>
> 作为查询提示的 MAXDOP 总是覆盖“最大并行度”配置选项。
>
> 如果查询在编译时被标记为串行执行，工作负荷组和配置设置都无法将该查询更改为并行执行。
>
> 一旦确定并行度以后，只有出现内存压力时，才能降低该值。在授权内存队列中等待的任务将无法查看工作负荷组的重新配置过程。

（6）GROUP_MAX_REQUESTS。该值是工作负荷组中允许同时执行的最大请求数。默认值为 0，表明不对请求进行限制。

使用 *ALTER WORKLOAD GROUP* 可对工作负荷组的任何属性进行修改。

4. 资源池

资源池是服务器物理资源的子集。每个资源池都有两部分。一部分不能与其他资源池重叠，用于设置资源的最小值。资源池的另一部分与其他池共享，用于定义最大可能的资源消耗。通过对每个资源的以下选项进行指定来设置资源池。

- CPU 的 MIN 或 MAX。

■ 内存百分比的 MIN 或 MAX。

MIN 代表 CPU 或内存的最小保证资源可用性，MAX 代表 CPU 或内存的最大资源池的大小。

资源池的共享部分用于指示当资源可用时，从何处获取可用的资源。不过，当资源正在使用时，它们将进入指定的资源池，成为不可共享资源。当某个指定的资源池中没有请求时，使用该策略可能会提高资源的利用率，并且向该资源池配置的资源也可以被释放，供其他资源池使用。

以下是每个资源池可以指定的 4 个值的详细信息。

(1) MIN_CPU_PERCENT。存在 CPU 竞争时，该值是资源池中能提供给所有请求的平均 CPU 带宽。SQL Server 试图在单个请求之间尽可能平均分配 CPU 带宽，并且适度考虑每个工作负荷组的 *IMPORTANCE* 属性。默认值为 0，表明没有最小值。

(2) MAX_CPU_PERCENT。存在 CPU 竞争时，该值是资源池中所有请求接收的最大 CPU 带宽。默认值为 100，表明没有最大值。如果没有 CPU 资源的竞争，一个资源池可以使用 100%的 CPU 带宽。

(3) MIN_MEMORY_PERCENT。该值用于指定为该资源池保留的、不能与其他资源池共享的最小内存量。如果该资源池中没有请求，但它设置了最小内存值，则其他资源池中的请求无法使用该内存，该内存将被浪费。在资源池内部，请求之间的内存分配基于先来先服务原则。另外，某个请求的内存还可能受工作负荷组属性（如 REQUEST_MAX_MEMORY_GRANT_PERCENT）的影响。默认值为 0，表明没有保留的最小内存。

(4) MAX_MEMORY_PERCENT。该值用于指定资源池中所有请求可以使用的总服务器内存的百分比。该值可以达到 100%，但实际数量将减少，因为其他资源池已经通过指定的 MIN_MEMORY_ PERCENT 值保留了一定的内存。MAX_MEMORY_PERCENT 始终大于或等于 MIN_MEMORY_PERC- ENT。单个请求的内存量将受工作负荷组策略的影响，如 REQUEST_MAX_MEMORY_GRANT_PER- CENT。默认设置为 100，表明一个资源池可以使用所有的服务器内存。即使服务器内存不足时，也不能超过该设置值。

以下是某些资源池配置出现的极端情况。

■ 所有资源池定义的最小值相加达到服务器资源的 100%。这等价于将服务器资源分成非重叠的块，而不考虑在任何指定资源池内部使用的资源。

■ 所有资源池都没有最小值。所有资源池将竞争可用的资源，它们最终获取的资源大小将取决于每个资源池中的资源消耗情况。

资源调控器为每个 SQL Server 实例预定义了两种资源池。

内部池。该资源池表示由 SQL Server 自身消耗的资源。该资源池始终只包含内部工作负荷组，在任何情况下都不允许更改它。不限制内部资源池使用的资源；不能改变内部资源池的使用情况，或者向它添加工作负荷组。不过，可以监视内部组使用的资源。

默认池。默认资源池起初只包含默认工作负荷组。不能删除默认资源池，但可以更改默认资源池，并且可以向它添加其他工作负荷组。注意，不能在默认资源池中删除默认组。

5. 资源池大小

表 1-2 出自 *SQL Server 联机丛书*，阐述了几个资源池中 MIN 和 MAX 之间的关系，以及如何有效地计算 MAX 的值。该表展示了内部池、默认池和两个用户定义池的设置。以下公式用于计算有效 MAX 百分比和共享百分比。

■ Min(X,Y)表示 *X* 和 *Y* 中的较小值。

■ Sum(X)表示所有池的 *X* 值之和。

- 总计共享百分比=100–sum(MIN%)。
- 有效 MAX 百分比=min(X,Y)。
- 共享百分比=有效 MAX 百分比–MIN 百分比。

表 1-2　　工作负荷组的 MIN 和 MAX

池名称	MIN 百分比设置	MAX 百分比设置	计算的有效 MAX 百分比	计算的有效共享百分比	注　释
内部	0	100	100	0	有效 MAX 百分比和共享百分比不适用于内部池
默认	0	100	30	30	有效 MAX 值的计算如下：min(100,100–(20+50))=30。计算的共享百分比为有效 MAX–MIN=30
池 1	20	100	50	30	有效 MAX 值计算如下：min(100,100–50)=50。计算的共享百分比为有效 MAX–MIN=30
池 2	50	70	70	20	有效 MAX 值计算如下：min(70,100–20)=70。计算的共享百分比为有效 MAX–MIN=20

表 1-3 也是出自 *SQL Server 联机丛书*，展示了创建新资源池时，以上表中的值如何进行调整。该新资源池为池 3，MIN 百分比设置为 5。

表 1-3　　资源池的 MIN 和 MAX 值

池名称	MIN 百分比设置	MAX 百分比设置	计算的有效 MAX 百分比	计算的有效共享百分比	注　释
内部	0	100	100	0	有效 MAX 百分比和共享百分比不适用于内部池
默认	0	100	25	25	有效 MAX 值计算如下：min(100,100–(20+50+5))=25。计算的共享百分比为有效 MAX–MIN=25
池 1	20	100	45	25	有效 MAX 值计算如下：min(100,100–55)=45。计算的共享百分比为有效 MAX–MIN=25
池 2	50	70	70	20	有效 MAX 值计算如下：min(70,100–25)=70。计算的共享百分比为有效 MAX–MIN=20
池 3	5	100	30	25	有效 MAX 值计算如下：min(70,100–70)=30。计算的共享百分比为有效 MAX–MIN=25

6. 示例

本部分包含一些资源调控器 DDL 命令的语法示例，以便更深入介绍所有这些概念是如何协同工作的。对所有可能的 DDL 命令选项，这部分的讨论是不完整的，因此需要参考 *SQL Server 联机丛书*。

```
--- Create a resource pool for production processing
--- and set limits.
USE master;
GO
CREATE RESOURCE POOL pProductionProcessing
WITH
(
     MAX_CPU_PERCENT = 100,
     MIN_CPU_PERCENT = 50
);
```

```
GO
--- Create a workload group for production processing
--- and configure the relative importance.
CREATE WORKLOAD GROUP gProductionProcessing
WITH
(
     IMPORTANCE = MEDIUM
)
--- Assign the workload group to the production processing
--- resource pool.
USING pProductionProcessing;
GO
--- Create a resource pool for off-hours processing
--- and set limits.
CREATE RESOURCE POOL pOffHoursProcessing
WITH
(
     MAX_CPU_PERCENT = 50,
     MIN_CPU_PERCENT = 0
);
GO
--- Create a workload group for off-hours processing
--- and configure the relative importance.
CREATE WORKLOAD GROUP gOffHoursProcessing
WITH
(
     IMPORTANCE = LOW
)
--- Assign the workload group to the off-hours processing
--- resource pool.
USING pOffHoursProcessing;
GO
--- Any changes to workload groups or resource pools require that the
--- resource governor be reconfigured.
ALTER RESOURCE GOVERNOR RECONFIGURE;
GO
USE master;
GO
CREATE TABLE tblClassifierTimeTable (
     strGroupName     sysname          not null,
     tStartTime       time             not null,
     tEndTime         time             not null
);
GO
--- Add time values that the classifier will use to
--- determine the workload group for a session.
INSERT into tblClassifierTimeTable
     VALUES('gProductionProcessing', '6:35 AM', '6:15 PM');
GO
--- Create the classifier function
CREATE FUNCTION fnTimeClassifier()
RETURNS sysname
WITH SCHEMABINDING
AS
BEGIN
     DECLARE @strGroup sysname
```

```
    DECLARE @loginTime time
    SET @loginTime = CONVERT(time,GETDATE())
    SELECT TOP 1 @strGroup = strGroupName
    FROM dbo.tblClassifierTimeTable
    WHERE tStartTime <= @loginTime and tEndTime >= @loginTime
    IF(@strGroup is not null)
    BEGIN
        RETURN @strGroup
    END
--- Use the default workload group if there is no match
--- on the lookup.
    RETURN N'gOffHoursProcessing'
END;
GO
--- Reconfigure the Resource Governor to use the new function
ALTER RESOURCE GOVERNOR with (CLASSIFIER_FUNCTION = dbo.fnTimeClassifier);
ALTER RESOURCE GOVERNOR RECONFIGURE;
GO
```

1.7.2 资源调控器控制

资源的实际限制是由资源池设置控制的。在 SQL Server 2008 中，可以控制内存和 CPU 资源，但无法控制 I/O。在未来的版本中，也许能控制更多的资源。内存和 CPU 资源之间的限制方式有重要的区别。

可以将池的内存规范看成是硬限制，没有池会使用大于其最大值的内存设置。另外，SQL Server 始终为每个池保留最小内存，因此，如果没有为某个池分配工作负荷组中的会话，其他会话无法使用它的最小内存保留。

然而，CPU 限制是软限制，其他会话可以使用未使用的计划程序带宽。另外，最大值也不能始终大于限制值。例如，如果存在两个池，一个池的最大值是 25%，另一个池的最大值是 50%，如果第一个池已经使用了其 25%的计划程序，那么来自另一个池中的组的会话可以使用所有剩下的 CPU 资源。作为软限制，它们使 CPU 使用率并不像内存使用率那样容易预测。如前文所述，每个会话都被分配到计划程序中，不需要考虑该会话在哪个工作负荷组中。假设在最小情况下，在双 CPU 实例上仅运行两个会话，每个会话很可能被分配到不同的计划程序中，并且两个会话也许在两个不同资源池的两个不同工作负荷组中。

假设 CPU1 上的会话来自第一个资源池的某个工作负荷组，该资源池的最大 CPU 设置为 80%; CPU2 上的第二个会话来自第二个资源池的某个组，该资源池的最小 CPU 设置为 20%。因为仅存在两个会话，所以每个会话使用计划程序的 100%内存，或者每个会话在实例上使用 50%的总 CPU 资源。然后，如果来自 20%的资源池的某个工作负荷组为 CPU1 分配了另一个任务，情况将发生改变。使用 20%资源池的任务对 CPU1 拥有 20%的资源，但它仍然对 CPU2 拥有 100%的资源；而使用 80%资源池的任务仍然对 CPU1 拥有 80%的资源。这说明，来自 20%资源池中运行的任务对总 CPU 拥有 60%的资源，来自 80%资源池中的一个任务仅对总 CPU 拥有 40%的资源。当然，随着越来越多的资源被分配到计划程序中，这种反常现象会逐渐消失，但是，由于存在跨多个 CPU 管理计划程序资源的工作方式，因此缺少显式的控制方法。

为了进行测试和故障诊断，有时候需要能轻松关闭所有的资源调控函数。使用 *ALTER RESOURCE GOVERNOR DISABLE* 命令禁用资源调控器，然后再使用 *ALTER RESOURCE GOVERNOR RECONFIGURE* 命令重新启动资源调控器。如果需要确保资源调控器始终保持禁用状态，可以在该场合使用跟踪标志 8040

来启动 SQL Server 实例。使用跟踪标志时，资源调控器将始终保持关闭状态，禁止任何操作。如果在单用户架构使用 -*m* 和 -*f* 标志来启动 SQL Server 实例，将产生相同的结果。如果禁用了资源调控器，可以发现以下行为。

- 仅存在*内部*工作负荷组和资源池。
- 资源调控器配置元数据没有被载入内存。
- 不再自动执行分类器函数。
- 可以观察和操作资源调控器元数据。

1.7.3 资源调控器元数据

使用资源调控器时，需要查看3种特定的目录视图。

- ***sys.resource_governor_confi guration*。**该视图返回存储的资源调控器状态。
- ***sys.resource_governor_resource_pools*。**该视图返回存储的资源池配置。每个视图行确定单个池的配置。
- ***sys.resource_governor_workload_groups*。**该视图返回存储的工作负荷组配置。

另外，还有3个专用于资源调控器的 DMV。

- ***sys.dm_resource_governor_workload_groups*。**该视图返回工作负荷组的统计信息和当前工作负荷组的内存配置。
- ***sys.dm_resource_governor_resource_pools*。**该视图返回有关当前资源池状态、当前资源池配置和资源池统计信息的信息。
- ***sys.dm_resource_governor_confi guration*。**该视图返回一行，该行包含资源调控器当前内存配置状态。

最后，还有6个包含有关资源调控器信息的 DMV。

- ***sys.dm_exec_query_memory_grants*。**该视图返回有关已经获取内存授权或仍然需要内存授权的查询的信息。该视图不显示那些不必等待内存授权的查询。资源调控器增加了以下列：*group_id*、*pool_id*、*is_small* 和 *ideal_memory_kb*。
- ***sys.dm_exec_query_resource_semaphores*。**该视图返回有关当前查询的资源信号量状态信息。它提供常见的执行查询的内存状态信息，并允许您确定系统是否可以访问足够的内存。资源调控器增加了 *pool_id* 列。
- ***sys.dm_exec_sessions*。**该视图为 SQL Server 上已通过验证的每个会话返回一行。资源调控器增加了 *group_id* 列。
- ***sys.dm_exec_requests*。**该视图返回 SQL Server 中正在执行的每个请求信息。资源调控器增加了 *group_id* 列。。
- ***sys.dm_exec_cached_plans*。**该视图为每个查询计划返回一行，SQL Server 为了更快地查询执行而缓存该查询计划。资源调控器增加了 *pool_id* 列。
- ***sys.dm_os_memory_brokers*。**该视图返回 SQL Server 内部分配的信息，它需要使用 SQL Server 内存管理器。资源调控器增加了以下列：*pool_id*、*allocations_db_per_sec*、*predicated_allocations_kb* 和 *overall_limit_kb*。

乍一看，您也许觉得构建资源调控器是画蛇添足，但由于它能为工作负荷组和资源池指定属性，因此可以为您提供最大限度的控制性和灵活性。您可以将工作负荷组看成是提供给开发人员使用的控制工

具，将资源池看成是限制开发人员行为的管理员工具。

1.8 SQL Server 2008 配置

在本章的第二部分，我们将讨论用于控制 SQL Server 2008 如何运行的选项。控制数据库引擎行为的一种主要方法是调整配置选项设置，但也可以通过其他方式配置数据库引擎行为。首先，我们将介绍使用 SQL Server 配置管理器来控制网络协议和与 SQL Server 相关的服务，然后介绍能够影响 SQL Server 行为的其他设置，最后介绍 SQL Server 中用于控制服务器范围内设置的某些特定配置选项。

1.8.1 使用 SQL Server 配置管理器

配置管理器是一种工具，用于管理与 SQL Server 相关的服务、配置 SQL Server 使用的网络协议、管理从客户端计算机向 SQL Server 连接的网络连接配置。它经过安装成为 SQL Server 的一部分。在 Management Studio 中用右键单击已注册的服务器，可以获取配置管理器，或者将它添加到任何其他 Microsoft 管理控制台（MMC）显示中。

1.8.2 配置网络协议

为了使客户端与服务器进行连接和通信，必须在客户端和服务器上同时启用特定的协议。SQL Server 可以立刻在所有启用的协议上侦听请求。客户端和服务器上应该已经安装了底层操作系统协议（如 TCP/IP）。Windows 安装过程中通常已经安装了网络协议，网络协议属于 SQL Server 安装的组成部分。只有客户端和服务器上同时安装了相应的网络协议，SQL Server 网络库才能工作。

在客户端计算机上，必须安装和配置 SQL Native Client，以便启用服务器上使用的网络协议；这通常是在客户端工具连接安装过程中完成的。SQL Native Client 是一个可供 OLE DB 和 ODBC 使用的独立的数据访问 API。如果 SQL Native Client 可用，连到 SQL Server 上的某个特定客户端可以配置任何网络协议。可以使用 SQL Server 配置管理器来启用单个协议或多个协议，并指定协议的使用顺序。如果启用了共享内存协议设置，那么将首先尝试使用该协议。但是，如前文所述，使用共享内存协议时，只有客户端和服务器位于相同的机器上时，通信才可用。

以下查询将返回当前连接使用的协议，假设使用 DMV *sys.dm_exec_connections*：

```
SELECT net_transport
FROM sys.dm_exec_connections
WHERE session_id = @@SPID;
```

1.8.3 默认的网络配置

其他计算机可以使用网络协议与 SQL Server 2008 进行通信，但在安装过程中，SQL Server 并没有全部启用这些协议。为了从特定的客户端进行连接，也许需要启用特定的协议。在所有安装中，共享内存协议都是默认安装的，但它只能用于在相同的计算机上从客户端应用程序连到数据库引擎，因此用处有限。

SQL Server 2008 的 TCP/IP 连接在开发人员、评估和 SQL Express 版本的新安装中都是禁用的。与 MDAC 2.8 连接的 OLE DB 应用程序无法使用“.”、“(local)”或“(<blank>)”作为服务器名称来连到本地服务的默认实例中。为此，需要提供服务器名称或在服务器上启用 TCP/IP 协议。与本地命名实例的连接

不受网络协议的影响，使用 SQL Native Client 的连接也不受网络协议的影响。之前已经安装过 SQL Server 的安装也许不受网络协议的影响。表 1-4 描述了默认的网络配置设置。

表 1-4 SQL Server 2008 默认的网络配置设置

SQL Server 版本	安装类型	共享内存	TCP/IP	命名管道	VIA
企业版	全新安装	启用	启用	禁用（仅能本地使用）	禁用
企业版（群集）	全新安装	启用	启用	启用	禁用
开发人员版	全新安装	启用	禁用	禁用（仅能本地使用）	禁用
标准版	全新安装	启用	启用	禁用（仅能本地使用）	禁用
工作组版	全新安装	启用	启用	禁用（仅能本地使用）	禁用
评估版	全新安装	启用	禁用	禁用（仅能本地使用）	禁用
Web 版	全新安装	启用	启用	禁用（仅能本地使用）	禁用
SQL Server Express 版	全新安装	启用	禁用	禁用（仅能本地使用）	禁用
所有版本	升级或并行安装	启用	保留以前安装中的设置	保留以前安装中的设置	禁用

1.8.4 管理服务

可以使用配置管理器启动、暂停、继续或停止与 SQL Server 有关的服务。可用的服务取决于已经安装的 SQL Server 特定组件，但应该始终具备 SQL Server 自身的服务和 SQL Server 代理服务。其他服务还包括 SQL Server 全文搜索服务和 SQL Server 整合服务（SSIS）。另外，还可以使用配置管理器来查看当前的服务属性，如该服务是否设置为自动启动。对更改服务属性而言，配置管理器是首选的工具，而不是使用 Windows 服务管理工具。当使用 SQL Server 工具（如配置管理器）更改 SQL Server 或 SQL Server 代理服务使用的账户时，SQL Server 工具将自动实现附加的配置，如在 Windows 注册表中设置许可证以便新账户能读取 SQL Server 设置。使用配置管理器更改密码可以即时生效，而无需重新启动服务。

SQL Server Browser

另外一个值得特别关注的相关服务是 SQL Server Browser 服务。如果在某个机器上运行 SQL Server 命名实例，该服务尤其重要。SQL Server Browser 侦听访问 SQL Server 资源的请求，并提供其运行的计算机上安装的各种 SQL Server 实例信息。

在 SQL Server 2000 之前的版本中，在一台机器上一次只能有一个 SQL Server 安装，此时还没有“实例”的概念。SQL Server 将始终侦听端口 1433 上传来的请求，但任何端口每次只能被一个连接使用。SQL Server 2000 引入了多个 SQL Server 实例支持，便开始使用名为 *SQL Server Resolution Protocol*（SSRP）的新协议来侦听 UDP 端口 1434。该侦听器能使用安装的 SQL Server 实例名称来应答客户端，并提供实例使用的端口号或命名管道。SQL Server 2005 使用 SQL Server Browser 服务替换了 SSRP，SQL Server 2008 仍然使用 SQL Server Browser。

如果某个计算机上没有运行 SQL Server Browser 服务，将无法从该机器连到 SQL Server，除非提供正确的端口号。不过，如果 SQL Server Browser 服务没有运行，以下连接将无法工作。

■ 不提供端口号或管道的命名实例连接。

- 如果连接不是使用 TCP/IP 端口，而是使用 DAC 连到命名实例或默认实例的连接。
- Management Studio、企业管理器或查询分析器中的枚举服务器。

在通过网络连接而被访问的 SQL Server 机器上，建议您将 Browser 服务设置为自动启动。

1.9 SQL Server 系统配置

可以通过几种方式和各种接口来配置 SQL Server 运行的机器及数据库引擎本身。首先，我们将介绍一些操作系统级的设置，它们可以影响 SQL Server 的行为。随后介绍一些 SQL Server 选项，它们也可以影响 SQL Server 的行为，但大家并不专门将它们看成是配置选项。最后，我们将介绍用于控制 SQL Server 2008 行为的配置选项，主要使用名为 *sp_configure* 的过程接口来设置这些配置选项。

1.9.1 操作系统配置

为了使 SQL Server 运行良好，它必须在经过调整的操作系统和正确配置的机器上运行。虽然讨论操作系统、硬件配置与调整的内容已经超出了本书的范围，但有些非常简单的问题却能对 SQL Server 的性能产生重要影响，我们将对此进行描述。

1. 任务管理

在本章的第一部分您已经看到，操作系统在系统中计划执行所有的线程。每个进程的每个线程都有一个优先级，Windows 总执行最高优先级的可用线程。默认方式下，操作系统将使活动的应用程序具有较高的优先级，但该优先级设置也许不适用于在后台运行的应用程序，如 SQL Server 2008。为了改变这种情况，SQL Server 安装程序修改了优先级设置，以消除对前台应用程序的偏袒。

定期对优先级设置进行仔细检查是预防别人修改该设置的好办法。可以打开“性能选项”对话框的“高级”选项卡。

如果正在使用的是 Windows XP 或 Windows Server 2003 操作系统，单击“开始”菜单，用右键单击“我的电脑”，选择“属性”，“系统属性”对话框将被打开。在“高级”选项卡上的“性能”区域中单击“设置”按钮，打开“性能选项”对话框，再次选择“高级”选项卡。

如果正在使用的是 Windows Server 2008 操作系统，单击“开始”菜单，用右键单击“计算机”，并选择“属性”，这时将打开“系统”信息屏幕。从左侧的列表中选择“高级系统设置”，打开“系统属性”对话框。如同在 Windows XP 和 Windows Server 2003 中一样，在“高级”选项卡上的“性能”区域中单击“设置”按钮，打开“性能选项”对话框，再次选择“高级”选项卡，如图 1-6 所示。

第一个选项设置用于指定如何分配处理器资源，可以调整程序或后台服务的最佳性能。选择“后台服务”使所有程序（包括后台和前台程序）接收同样多的处理器资源。如果计划从本地客户端连到 SQL Server 2008（即客户端和服务器在相同的计算机上运行），可以使用该选项改善处理时间。

2. 系统分页文件的位置

如果需要，应该将操作系统分页文件放在不同的驱动器上，而不只是放在 SQL Server 使用的文件上。如果系统将被分页，这尤为重要。不过，更好的方法是增加内存或更改 SQL Server 内存配置，以有效消除分页。通常，设计 SQL Server 的目的是最小化分页，因此，如果内存配置值适合系统上的物理内存量，出现的分页文件活动量将非常小，文件的位置也无关紧要。

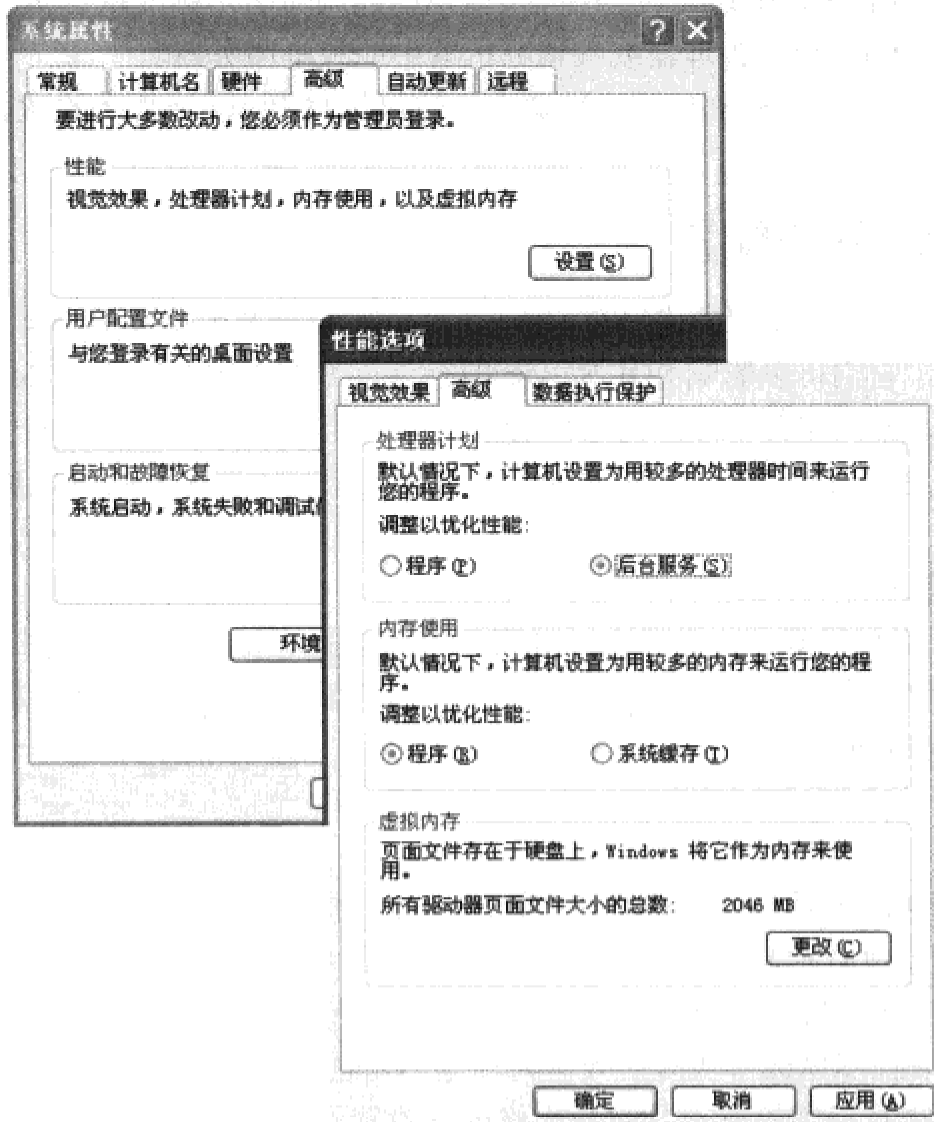


图 1-6 后台服务的优先级配置

3. 非必需的服务

您应该禁用不需要的服务。在 Windows Server 2003 中，用右键单击“我的电脑”，选择“管理”。在“计算机管理”工具中展开“服务和应用程序”节点，单击“服务”。在右边的窗格中，将看到操作系统上所有可用的服务清单。通过用右键单击服务的名称并选择“属性”来更改服务的启动属性。不必要的服务将增加系统的开销，并占用 SQL Server 本来可以使用的资源。不要将不必要的服务标识为自动启动。要避免使用运行 SQL Server 的服务器作为域控制器、组文件或打印服务器、Web 服务器或动态主机配置协议（DHCP）。另外，还需要考虑禁用警报器、剪贴板、计算机浏览器、Messenger、网络动态数据交换（DDE）和任务计划程序服务，这些服务都是默认启用的，但 SQL Server 不需要这些服务。

4. 连接

应该仅选择实际需要的网络协议。如本章先前所述，可以使用 SQL Server 配置管理器来禁用不需要的协议。

5. 防火墙设置

不合适的防火墙设置是导致网络间 SQL Server 连接失败的另一种系统配置问题。防火墙系统有助于

阻止计算机资源的未授权访问，通常是必要的。但通过防火墙访问 SQL Server 的实例，需要在运行 SQL Server 的计算机上配置防火墙来允许访问。可以使用多种防火墙系统，需要针对自己的系统查看相关文献，了解配置防火墙的详细信息。通常，防火墙配置需要执行以下步骤。

（1）配置 SQL 实例，以使用特定的 TCP/IP 端口。SQL Server 默认使用端口 1433，但可以对此进行更改。命名实例默认使用动态端口，也可以使用 SQL Server 配置管理器进行更改。

（2）配置防火墙，允许授权用户或计算机访问特定的端口。

（3）作为配置 SQL Server 侦听某个特定端口并打开该端口的备选方案，如果需要连到命名实例上，您可以列出 SQL Server 可执行文件（Sqlservr.exe）和 SQL Browser 可执行文件（Sqlbrowser.exe），阻塞程序除外。需要继续使用动态端口时，可以采用该方案。

1.9.2 跟踪标记

*SQL Server 联机丛书*仅列出了它完全支持的一些跟踪标记。可以将跟踪标记看成是特殊的开关，您可以打开或关闭该开关来改变 SQL Server 行为。实际上，跟踪标记有很多，但大多数跟踪标记都是为 SQL Server 开发小组进行内部测试产品而创建的，而不是供 Microsoft 外部人员使用的。

可以分别使用 *DBCC TRACEON* 或 *DBCC TRACEOFF* 命令将跟踪标记设置为打开或关闭；如果使用 Sqlservr.exe 启动 SQL Server，还可以在命令行将跟踪标记指定为打开或关闭。另外，每次启动 SQL Server 时，也可以使用 SQL Server 配置管理器来启用一个或多个跟踪标记（可以参阅 *SQL Server 联机丛书*了解实现细节）。使用 *DBCC TRACEON* 启用跟踪标记仅对单个连接合法（除非指定附加参数–1）。在指定附加参数的情况下，跟踪标记（甚至是在运行 *DBCC TRACEON* 之前打开的跟踪标记）对所有连接都是合法的。SQL Server 服务启动过程中启用的跟踪标记对所有会话都有效。

少数跟踪标记与本书讨论的主题关系非常密切，因此，我们将在描述相关主题时讨论特殊的跟踪标记。例如，我们在涉及资源调控器时已经提到了跟踪标记 8040。

警告：

因为跟踪标记改变了 SQL Server 的行为，所以如果使用不当将引发故障。仅尝试使用跟踪标记查看操作结果也会产生危害，尤其不要在产品系统上进行尝试。要想有效地使用跟踪标记，需要全面理解 SQL Server 的默认行为（以准确了解将会改变哪些行为），并进行全面测试，确定系统能真正从使用跟踪标记中受益。

1.10 服务器配置设置

如果选择让 SQL Server 自动配置系统，SQL Server 将为您动态调整大多数重要的配置选项。除非有合理的理由来更改配置选项，否则最好接受默认配置值。槽糕地配置系统能破坏系统性能。例如，未正确配置内存设置的系统会中断应用程序。

在特定情况下对设置进行适当的调整，而不只是让 SQL Server 动态调整设置，也许会稍微改进系统性能，若将调整放在应用程序和数据库设计、索引、查询调优和其他活动上也许效果更好，这些内容我们将在后面讨论。系统设置从合理的配置变为理想的配置，性能改进也许只有 5%，但糟糕地配置系统将严重损害应用程序的性能。

使用目录视图 *sys.configurations* 可查询 SQL Server 2008 中 68 个服务器配置选项。

只有当具有明确的理由时才更改配置选项，并且需要密切监视每个变化产生的效果，以便确定该更改是提高还是降低了性能。注意，每次只能更改和监视一个变化。可以通过各种方式更改服务器范围内选项，使用 *sp_configure* 系统存储过程可以设置其中所有的选项。不过，在 68 个选项当中，只有 16 个选项是高级选项。在默认方式下，无法使用 *sp_configure* 来管理高级选项。您首先需要将 Show Advanced Options 选项更改成 1，如下所示：

```
EXEC sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO
```

为了查看哪些选项是高级选项，再次查询 *sys.configurations* 视图，并查看名为 *is_advanced* 的列，它允许您查看哪些选项是高级选项：

```
SELECT * FROM sys.configurations
WHERE is_advanced = 1;
GO
```

另外，也可以在 Management Studio 的对象资源管理器窗口中的 Server Proerties 对话框中对大量配置选项进行设置，但无法使用单个对话框查看或更改所有的配置选项。大多数可以从 Server Properties 对话框中更改的选项都是通过一个属性页来控制的。通过用右键单击 Management Studio 中的 SQL Server 实例名称可以获取该属性页。在图 1-7 中可以查看属性页清单。

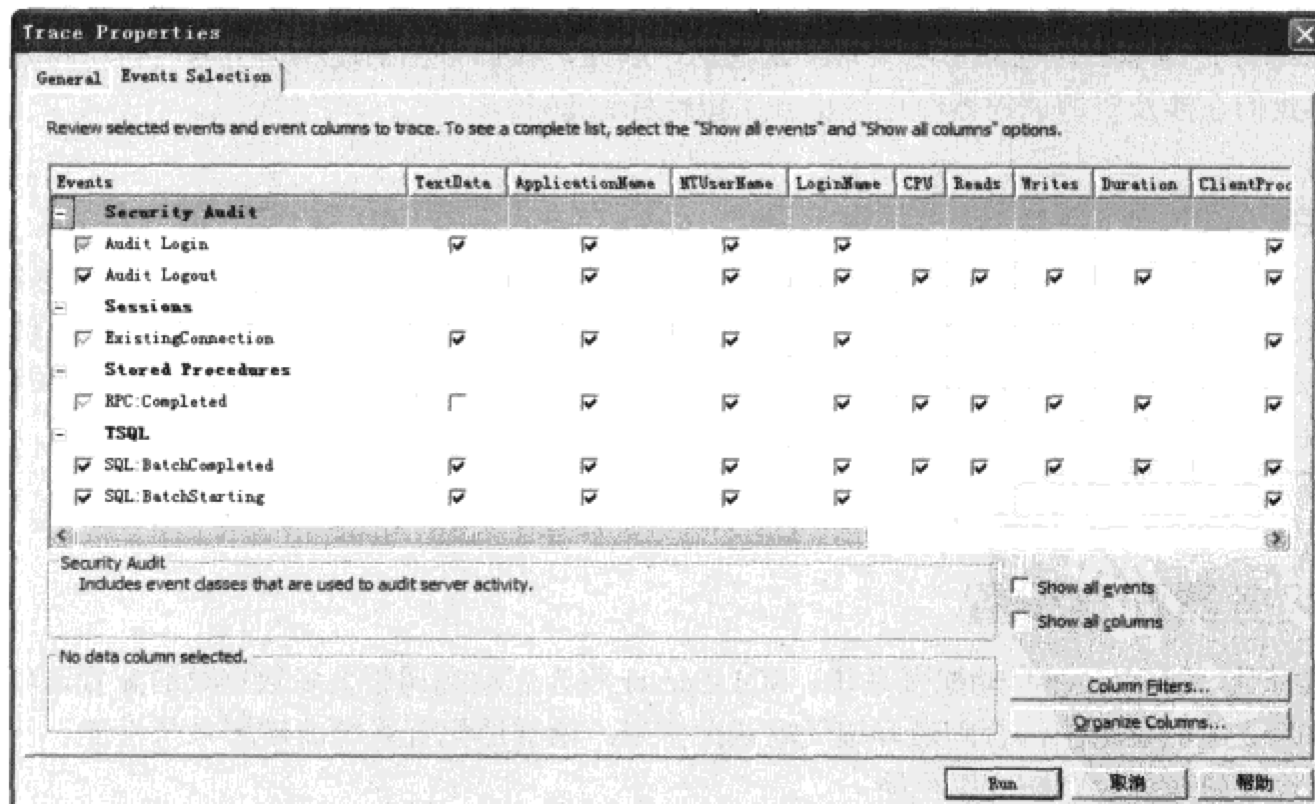


图 1-7 Magement Studio 中的服务器属性页清单

如果使用 *sp_configure* 存储过程，只有 *RECONFIGURE* 命令运行时，更改才能生效。在某些情况下，如果更改的选项值超出了推荐范围，可能必须指定 *RECONFIGURE WITH OVERRIDE*。动态更改的重新

配置将立刻生效，但其他更改只有等到服务器重新启动时才能生效。运行 *RECONFIGURE* 以后，通过 *sp_configure* 显示的某个选项的 *run_value* 与 *config_value* 不相同，如果该值与 *sys.configurations* 中的 value_in_use 不同，则必须重新启动 SQL Server 服务，使新值生效。可以使用 *sys.configurations* 视图判断哪些选项是动态的：

```
SELECT * FROM sys.configurations
WHERE is_dynamic = 1;
GO
```

在此我们不准备介绍所有的配置选项，只介绍其中最有用的一些配置选项，或者是与 SQL Server 性能有关的配置选项。在大多数情况下，我只讨论一些您不应该更改的选项。其中某些选项是与性能有关的资源设置，因为它们将消耗内存（如锁）。如果将它们设置得过高，它们将夺取系统内存并降低性能。我们将根据功能对配置设置进行分组。注意，SQL Server 几乎自动设置了所有选项，不需要查看选项，应用程序就能正常工作。

1. 内存选项

在前面部分中，您了解了 SQL Server 如何使用内存，包括它如何为不同的使用情况分配内存，以及它何时从磁盘读取数据、何时将数据写入磁盘。不过，我们没有讨论如何进行控制，以及 SQL Server 为了实现这些目标实际使用了多少内存。

最小服务器内存和最大服务器内存选项。默认情况下，SQL Server 会调整即将使用的总内存资源量。不过，您也可以使用“最小服务器内存”和“最大服务器内存”配置选项来进行手动控制。最小服务器内存的默认设置是 0MB，最大服务器内存的默认设置是 2147483647MB。如果使用 *sp_configure* 存储过程把这两个选项更改为相同的值，基本上就完全控制了服务器内存，并通知 SQL Server 使用固定的内存大小。最大的 2147483647MB 实际上是基础系统表的整型字段中可以存储的最大值，它与系统的实际资源无关。最小服务器内存选项不强制 SQL Server 在启动时获取最小内存量。内存需求是根据数据库工作负荷进行分配的。但是，一旦达到了最小服务器内存的阈值，即使 SQL Server 实际使用的内存量小于阈值，它也不会释放内存。为了确保每个实例分配的内存至少等于最小服务器内存，我们推荐在 SQL Server 启动后立刻执行数据库服务器加载。在正常的服务器活动过程中，每个实例可用的内存是变化的，但它永远不会小于每个实例的最小服务器内存值。

设置工作集大小。“设置工作集大小”选项是以前版本中的控制选项，后续版本的 Microsoft SQL Server 将删除该选项。尽管试图使用该值时，不会接收错误消息，但 SQL Server 2008 将忽略该设置。

启用 AWE。该选项用于启用 AWE API 以支持 32 位系统上的大容量内存。启用 AWE 以后，SQL Server 2008 可以使用和企业版、开发人员版及标准版同样多的内存。当 SQL Server 在 Windows Server 2003 或 Windows Server 2008 上运行时，在启动时它仅保留少量的 AWE 映射内存。当需要附加的 AWE 映射内存时，操作系统将向 SQL Server 动态分配内存。类似地，如果需要较少的资源，SQL Server 可以将 AWE 映射内存返还给操作系统，供其他进程或应用程序使用。

在 Windows Server 2003 或 Windows Server 2008 中使用 AWE 时，都需要锁定内存中的页，以防将它们写入分页文件。如果需要附加物理内存，Windows 必须置换出其他应用程序，因此这些应用程序的性能也许会受到影响。所以启用 AWE 时，还需要设置最大服务器内存。

如果在同一台计算机上运行多个 SQL Server 实例，并且每个实例都使用 AWE 映射内存，那么应该确保实例按预期执行。每个实例都应该具有最小服务器内存设置。因为无法将 AWE 映射内存置换到分页

文件中，所以所有实例的最小服务器内存值总和应该小于计算机上的总物理内存。

如果构建 SQL Server 用于故障转移群集，并配置使用 AWE 内存，则必须确保所有实例的最大服务器内存设置总和小于群集中任何服务器上可用的最少物理内存。如果故障转移节点的物理内存小于原始节点，那么 SQL Server 实例可能无法启动。

用户连接。如果“用户连接”配置设置保留其默认值 0，那么 SQL Server 2008 将动态调整同时连到该服务器的连接数。即使将该值设置为不同的数，在用户未真正连接之前，SQL Server 也不会为每个用户连接分配全额的内存需求量。SQL Server 启动时，它将为“用户连接”分配一组指针和与分配值大小相同的条目。如果必须使用该选项，选项值不宜过高，因为无论是否使用连接，每个连接的开销大约都为 28KB；不过，也不宜过低，因为如果超出了用户连接的最大数，将收到错误消息，并且在其他连接未完成之前，无法进行连接（在这种情况下，DAC 连接仍然可以使用该连接）。注意，用户连接值和用户数是不相同的，通过一个应用程序，可以打开多个 SQL Server 连接。理想状况下，应该让 SQL Server 动态调整“用户连接”选项的值。

重要提示：

“锁”配置选项是早期版本的设置，后续版本的 Microsoft SQL Server 将删除该选项。尽管试图使用该值时，不会收到错误消息，但 SQL Server 2008 将忽略该设置。

2. 计划选项

如前文所述，SQL Server 2008 通过 SQLOS 采用特殊的算法来计划用户进程，SQLOS 管理每个逻辑处理器上的一个计划程序，并确保在给定的时间内只有一个进程在计划程序上运行。SQLOS 管理工作线程的用户连接分配，使每个 CPU 上的用户数尽可能保持平衡。以下 5 个选项影响计划程序的行为：“轻型池”、“关联掩码”、“Affinity64 掩码”、“优先级提升”和“最大工作线程数”。

关联掩码和 Affinity64 掩码。从操作系统的角度看，Windows 可以在不同的处理之间有效地移动进程线程，但该活动会降低 SQL Server 的性能，因为每个处理器缓存的数据都将重新加载。通过设置关联掩码选项，可以允许 SQL Server 向特定的线程分配处理器，在负载过重时，通过消除处理器重新加载并减少处理器之间的线程迁移和上下文切换来改善性能。将关联掩码设置为非 0 值不仅可以控制绑定到处理器的计划程序数，还允许您来限制哪些处理器用于执行 SQL Server 请求。

关联掩码的值是一个 4 字节整数，每个位控制一个处理器。如果将代表某个处理器的位设置为 1，该处理器将被映射到特定的计划程序中。4 字节的关联掩码最多可以支持 32 个处理器。例如，在 8 路组合（eight-way box）上配置 SQL Server 使用 0 到 5 之间的处理器，可以将关联掩码设置为 63，它等价于位字符串 0011111。在 16 路组合（16-way box）上启用 8 到 11 之间的处理器，可以将关联掩码设为 3840 或 0000111100000000。例如，如果您需要在支持多实例的机器上使用该功能，可以将每个实例的关联设置为使用机器上不同的处理器集合。

为了支持多于 32 个 CPU 的情况，可以为最初的 32 个 CPU 配置 4 字节关联掩码，并使用 4 字节 Affinity64 掩码配置剩余的 CPU。注意，支持 33 到 64 个处理器的服务器关联仅适用于 64 位操作系统。

您可以为所有可用的 CPU 配置启用关联掩码。对 8 路组合机器而言，将关联掩码设置为 255 表示启用所有 CPU。这与将关联掩码值设置为 0 不完全相同。设置为*非零*值时，计划程序将被绑定到特定的 CPU

中；而对 *0* 值而言，则不是这样。

轻型池。默认情况下，SQL Server 在线程模式中操作轻型池。这表明，处理 SQL Server 工作请求的是线程。如先前所述，SQL Server 还可以让连接在纤程架构中运行。与线程相比，纤程开销更小。轻型池选项的值可以是 0 也可以是 1，1 表示 SQL Server 应该在纤程模式中运行。使用纤程也许会产生微小的性能优势，特别是当您使用 8 个或更多 CPU 并且几乎所有可用 CPU 都在满负荷运行时。

如果运行环境中的 CPU 资源在满负荷工作，并且系统监视器显示存在大量的上下文交换，那么将轻型池设为 1 也许会产生一些性能优势。

优先级提升。如果启用了"优先级提升"，SQL Server 将在更高计划优先级下运行。在 Windows 2000 和 Windows Server 2003 中执行优先级提升以后，服务器中*每个*进程的优先级都被设置为 13。大多数进程都在正常优先级下运行，正常优先级是 7。由此产生的直接结果是，如果服务器正在运行非常耗资源的工作负荷，并且接近了 CPU 的最大极限，这些正常优先级的进程将无法获取资源。

优先级提升的默认设置是 0，这表明无论是否在单处理器机器上执行优先级提升，SQL Server 都将在正常优先级下运行。也许很少站点或应用程序需要设置该选项来提升优先级，但是，如果机器完全专用于运行 SQL Server，可能需要启用该选项（将它设置为 1）。优先级提升可以在重负载的专用系统上提供潜在的性能优势。和大部分配置选项一样，应该谨慎使用优先级提升选项。如果将优先级提升得过高，可能会影响核心操作系统和网络操作，导致关闭 SQL Server 或在服务器上运行其他操作系统任务时出现问题。

最大工作线程数。通过保留工作线程（线程或纤程）池，SQL Server 使用操作系统的线程服务。工作线程用于从队列获取请求。最大工作线程数试图在 SQLOS 之间平均分配工作线程，因此，每个计划程序可以使用的线程数是根据 CPU 数目进行划分的最大工作线程数设置。对 100 个或更少的用户而言，工作线程数和活动用户（不只是已连接的可用用户）数通常是相同的。对于更多的用户而言，工作线程通常比活动用户更少才有意义。虽然某些用户请求必须等待工作线程变为可用状态，但由于减少了上下文切换，因此总吞吐量将增加。

最大工作线程的默认值为 0，表示 SQL Server 将根据处理器的数目和机器的架构来配置工作线程数。例如，对运行 SQL Server 的 4 路组合 32 位机器而言，默认值是 256 个工作线程。这并不是说 SQL Server 在启动时创建 256 个工作线程，它表示，如果某个链接正在等待服务，并且没有可用的工作线程，那么当总工作线程数小于 256 时将创建新的工作线程。如果将最大工作线程数配置为 256，同时执行命令的最大数假设是 125，那么实际工作线程数将不会超过 125。实际工作线程数甚至比 125 还要小，因为 SQL Server 将撤销和裁剪不再被使用的工作线程。如果系统同时处理的连接数小于等于 100，可能不需要使用该设置。在这种情况下，工作线程池将不超过 100 个。

表 1-5 列出了指定机器架构的默认工作线程数和处理器数（注意，Microsoft 推荐把 1024 作为 32 位操作系统的最大工作线程数）。

表 1-5　最大工作线程数的默认设置

CPU 数	32 位计算机	64 位计算机
小于等于 4 个处理器	256	512
8 个处理器	288	576
16 个处理器	352	704
32 个处理器	480	960

使用默认的设置，处理 4000 或更多个用户连接的系统也能正常运行。数以千计的用户同时连接时，实际的工作线程池通常远低于 SQL Server 设置的最大工作线程数。因为从数据库的角度看，即使用户在客户端上处理大量工作，大部分连接仍然是可用的。

3. 磁盘 I/O 选项

没有选项可用于控制 SQL Server 的磁盘读取行为。以前 SQL Server 版本中用于控制预读的所有调整选项现在完全采取内部处理的方式。其中一个选项可用于控制磁盘写行为和控制检查点进程写入磁盘的频率。

恢复间隔。“恢复间隔”选项可以自动配置。SQL Server 安装时将该选项设置为 0，表明进行自动配置。在 SQL Server 2008 中，该设置意味着恢复时间应该小于 1 分钟。该选项允许数据库管理员通过指定每个数据库恢复所需的最长时间来控制检查点频率。SQL Server 将估计有多少数据修改可以在此恢复时间间隔期间前滚，然后检查每个数据库日志（如果恢复间隔时间设置为 0，则每分钟执行一次），并为每个满足条件的数据库发布一个检查点。对于事务日志相对较小的数据库，SQL Server 将在日志达到总数的 70%且小于所估计的数量时发布检查点。

“恢复间隔”选项不会影响撤销长时间运行事务所需的时间。例如，如果在服务器禁用之前花两个小时来执行更新，那么实际的恢复时间必然比恢复间隔值长得多。

每个数据库中出现检查点的频率取决于数据修改量，而不是基于时间尺度来确定。因此，主要用于读操作的数据库不会有大量的检查点。为了避免出现过多的检查点，SQL Server 试图确保设置的恢复间隔值是连续检查点之间的最小值。

如前文所述，在检查点操作过程中通常不会出现磁盘写操作。检查点可以确保将那些未通过其他机制写入磁盘的所有脏页及时写入磁盘。因此，应该将恢复间隔值设置为 0（自动配置）。

关联 I/O 掩码和 Affinity64 I/O 掩码。这两个选项用于控制 I/O 操作的处理器关联，并且这两个选项在控制工作线程的处理关联时采用的工作方式大致相同。在这些位掩码中将某个处理器设置为 1 位，意味着对应处理器只能用于 I/O 操作。您也许永远都不需要使用该选项。不过，如果您确定需要使用它，可能只是用于测试，那么必须将它与关联掩码或 Affinity64 掩码选项一起使用，并确保位设置不重叠。因此，应该使用以下 3 种组合设置之一：关联 I/O 掩码和 CPU 关联掩码均为 0；关联 I/O 掩码选项为 1，关联掩码为 0；关联 I/O 掩码为 0，关联掩码为 1。

备份压缩默认值。备份压缩是 SQL Server 2008 中的新功能，为了向后兼容，备份压缩的默认值为 0，表示不压缩备份。虽然只有企业版实例可以创建压缩备份，但任何 SQL Server 2008 版本都可以还原压缩备份。启用备份压缩时，服务器上的压缩操作是在写入操作之前执行的，因此它可以大大减小备份体积和向外部设备写入备份的 I/O 需求。空间压缩量取决于多种因素，包括下面几个因素。

- **备份的数据类型**。例如，字符数据的压缩量大于其他数据类型。
- **数据是否加密**。加密数据的压缩明显小于同等条件下未加密的数据。如果使用透明数据加密方法对全部数据进行加密，压缩备份也许不能显著减少它们的大小。

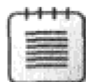
执行备份以后，可以使用以下语句在 *msdb* 数据库中检查 *backupset* 表，以确定压缩比率：

```
SELECT backup_size/compressed_backup_size FROM msdb..backupset;
```

虽然压缩备份可以大大减少所需的 I/O 资源，但它在执行压缩时也会显著增加 CPU 使用率。这种额外的负载将影响其他并行执行的操作。为了使影响降到最低，可以考虑使用资源调控器，为执行备份的

会话创建工作负荷组，并将该组分配到资源池中，同时限制资源池的最大 CPU 使用率。

配置值是备份压缩实例范围内的默认值，但通过指定 WITH COMPRESSION 或 WITH NO_COMPRESSION 执行特殊备份操作可以覆盖默认值。任何备份类型都可以使用压缩：完整备份、日志备份、差异或部分备份（文件或文件组备份）。



注意：

压缩备份的算法与数据库压缩算法差别很大。备份压缩使用类似于 zip 的压缩算法，它仅在数据中寻找架构。本书将在第 7 章讨论数据压缩。

文件流访问级别。文件流通过在文件系统上存储 BLOB 数据文件将数据库引擎与 NTFS 文件集成在一起，使用 T-SQL 或 Win32 文件系统接口访问该数据，以提供对该数据的流访问。文件流使用 Windows 系统缓存来缓存文件数据，有助于减少文件流数据对 SQL Server 性能可能产生的任何影响。文件流没有使用 SQL Server 缓冲池，因此不会减少查询进程的可用内存。

在设置该配置选项指示文件流数据的访问级别之前，必须使用 SQL Server 配置管理器从外部启动 *FILESTREAM*（如果在 SQL Server 安装时没有启动 *FILESTREAM*）。要使用 SQL Server 配置管理器，可用右键单击 SQL Server 服务的名称并选择“属性”。对话框的 *FILESTREAM* 选项具有单独的选项卡。必须选中顶部的复选框来启用针对 T-SQL 访问的 *FILESTREAM* 功能，然后如果需要，还可以选择为文件 I/O 流启用 *FILESTREAM*。

为 SQL Server 实例启用 *FILESTREAM* 以后，可以设置配置值。可以使用以下值之一。

- ***0 Disable FILESTREAM***。支持该实例。
- ***1 Enables FILESTREAM***。支持 T-SQL 访问。
- ***2 Enables FILESTREAM***。支持 T-SQL 和 Win32 流访问。

存储文件流数据的数据库必须包含特殊的文件流文件组。我们将在第 3 章讨论文件组。更多有关文件流存储的详细信息将在第 7 章介绍。

4. 查询处理选项

SQL Server 使用几个选项来控制可用于处理查询的资源。与所有其他调整选项一样，除非经过全面的测试来表明更改有利于改善性能，否则最好使用默认值。

每次查询占用的最小内存。如果查询需要附加内存资源，那么查询获取的页数则部分取决于每次查询占用的最小内存选项。当明确要求使用 ORDER BY 语句时，该选项与排序操作有关。另外，该选项也适用于合并连接操作、哈希连接及哈希分组操作所需的内部内存。在执行以上这些操作之前，该配置选项允许您指定应该为这些操作留出的最小内存量。排序、合并和哈希操作通过动态方式接收内存，因此很少需要调整该值。实际在大型机器上，排序和哈希查询需要的内存通常远远大于“每次查询占用的最小内存”设置，因此没必要对此进行严格限制。但是，如果需要实现大量哈希或排序操作，并且拥有很少用户或有大量可用内存，那么通过调整该值也许能改进性能。在小型机器上，将该值设置得过高会导致虚拟内存分页，这将降低服务器性能。

查询等待。该选项控制附加内存不可用时查询等待该附加内存所需的时间。如果值设置为-1，表示查询等待时间是估计执行查询时间的 25 倍，但该设置至少等待 25 秒。值为 0 或更大表示查询等待的秒数。如果等待超时了，SQL Server 将生成错误 8645。

```
Server: Msg 8645, Level 17, State 1, Line 1
A time out occurred while waiting for memory resources to execute the query. Re-run the
query.
```

尽管内存是动态分配的，但如果机器上的内存资源耗尽，SQL Server 仍然会出现内存不足的问题。如果查询超时并产生错误 8645，可以尝试增加分页文件的大小，或者添加更多的物理内存，也可以尝试创建更有效的索引来优化查询，以便消除哈希或合并操作。注意，该选项仅影响那些必须等待内存的查询（哈希和合并操作需要内存），而不影响那些由其他原因而必须等待的查询。

阻塞的进程阈值。该选项在用户任务被阻塞的时间超过配置的秒数时允许管理员请求通知。阻塞进程阈值设为 0 时，不提供通知。阈值最大可设为 86 400 秒。当死锁监视器检测到某个任务的被等待时间超过了配置值时，将会生成一个内部事件。可以在两种方式中选择一种来通知该事件。使用 SQL 跟踪来创建跟踪并捕获阻塞进程报告的事件类型，该阻塞进程报告可以在 SQL Server Profiler 中的“事件选择”屏幕上的“错误和警告”目录中找到。只要资源在可检测死锁资源上保持阻塞状态，每次死锁监视器检测死锁时就会提交该事件。在描述阻塞资源和被等待资源的“文本数据”跟踪列中可以捕获可扩展标记语言（XML）字符串。有关死锁检测的详细信息将在第 10 章介绍。

另外，也可以使用事件通知向 Service Broker 服务发送事件信息。事件通知可以为定义跟踪提供编程备用信息，它们可用于响应 SQL 跟踪捕获的许多相同事件。事件通知是异步执行的，用于在 SQL Server 2008 实例内部仅使用很少的内存资源开销来执行操作以响应事件。由于事件通知是异步执行的，因此这些操作不会消耗即时事务定义的任何资源。

索引创建内存。“每次查询占用的最小内存”选项仅适用于查询执行过程中使用的排序和哈希，它不适用于索引创建过程中出现的排序。另一个选项“索引创建内存”允许您为索引分配特定的内存量。索引创建内存的单位为千字节。

查询调控器开销限制。使用“查询调控器开销限制”选项来指定查询可以运行最大秒数。如果为该选项指定一个非零非负值，SQL Server 将禁止执行估计开销超过该值的查询。如果将该选项指定为 0（默认值），则关闭查询调控器，并允许所有查询在没有任何时间限制的条件下运行。

最大并行度和并行开销阈值。SQL Server 2008 允许在两个或多个处理器上同时运行某些复杂的查询。这些查询必须适合于分步执行，下面是一个示例：

```
SELECT AVG(charge_amt), category
FROM charge
GROUP BY category
```

如果费用表有 1 000 000 行，并且有 10 个不同的 *category* 值，那么 SQL Server 可以将行进行分组，并且仅让组的子集在每个处理器上处理。例如，对于包含 4 个 CPU 的机器来说，可以将类别 1 至 3 平均分配到第 1 个处理器上，将类别 4 至 6 平均分配到第 2 个处理器上，将类别 7 至 8 平均分配到第 3 个处理器上，将类别 9 至 10 平均分配到第 4 个处理器上。每个处理器可以为自身的组提供平均值，把单独的平均值汇集起来形成最终的结果。

在最优化过程中，查询优化器在考虑并行计划之前总是查找开销尽可能便宜的串行计划。如果串行计划开销小于并行开销阈值选项，那么将不产生并行计划。并行开销阈值是以秒为单位的查询开销；默认值是 5。如果最便宜的串行计划开销大于该配置阈值，查询优化器将根据处理器的数目和运行时实际可用的内存量来产生并行计划。该并行计划开销与串行计划开销进行比较，最终将选择较便宜的开销计划，

另一个计划被放弃。

并行查询执行计划可以使用多个线程；非并行查询使用串行执行计划，串行执行计划仅使用一个单个线程。并行查询使用的实际线程数是在查询计划执行初始化时确定的，即为 DOP。并行度的确定基于多种因素，包括关联掩码设置、最大并行度设置，以及查询启动执行时可用的线程数。

通过查询 DMV *sys.dm_os_tasks* 观察 SQL Server 何时并行执行查询。在多个 CPU 上运行的查询为每个线程分配 1 行，如下所示：

```
SELECT
    task_address,
    task_state,
    context_switches_count,
    pending_io_count,
    pending_io_byte_count,
    pending_io_byte_average,
    scheduler_id,
    session_id,
    exec_context_id,
    request_id,
    worker_address,
    host_address
FROM sys.dm_os_tasks
ORDER BY session_id, request_id;
```

使用最大并行度和并行开销阈值时请务必注意，它们仅在服务器范围内产生影响。

还有其他配置选项我们在这里不进行讨论，因为其中大部分涉及 SQL Server 的内容超出了本书的范围。这些选项包括配置远程查询、复制、SQL 代理、C2 审核和全文搜索。在编程 SQL Server 对象中存在一个禁用 CLR 的 Boolean 选项，默认值为关闭(0)。“允许更新”选项仍然存在，但它在 SQL Server 2008 中无效。还有一些配置选项涉及编程问题，详细信息可以参考《Inside SQL Server 2008: TSQL Programming》。这些选项包括处理递归和嵌套触发器、游标，以及数据库之间的访问对象。

5. 默认跟踪

最后一个不适合归为任何其他类别的选项是*默认跟踪已启用*。我们提它是因为它的默认值为 1，这意味着只要 SQL Server 启动，它将运行服务器范围内的跟踪，在预定位置捕获预定信息集。无法更改该默认跟踪的任何属性；唯一能做的就是关闭它。

默认跟踪输出文件存储在安装 SQL Server 的相同类别中（在\Log 子类别下）。如果在默认位置安装了 SQL Server，默认实例的捕获跟踪信息将保存在 C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSSERVER\MSSQL\LOG\Log.trc 中。每次停止和启动 SQL Server，或者文件达到最大文件大小 20MB 时，将创建新文件并使用连续数字后缀作为文件名。因此，第二个跟踪文件的名称为 Log_01.trc，随后是 Log_02.trc，依此类推。如果删除或重命名所有跟踪日志文件，那么下一个跟踪文件又重新从 log.trc 开始命名。SQL Server 为每个实例保存的跟踪文件不超过 5 个，因此创建第 6 个文件时，将删除最先创建的文件。

通过使用 SQL Server Profiler，可以打开由默认跟踪机制创建的跟踪文件，对其他跟踪文件可以执行同样操作。或者使用系统函数 *fn_trace_gettable* 将它们复制到表中，在跟踪文件运行的同时查看当前的跟踪内容。与任何写入文件的服务器范围内的跟踪一样，写操作以 128KB 块为单位。因此，在低使用率的

SQL Server实例上，很长时间内就像没有写入任何内容一样。对于任何物理文件写入操作，都需要128KB数据。另外，当SQL Server服务停止时，该跟踪积累的任何事件都将全部写入该跟踪文件中。

默认跟踪与黑箱跟踪不同，黑箱跟踪完全捕获每个批处理操作，因此很快就会获取庞大的信息。SQL Server 2008中的默认跟踪仅捕获少量可能会导致稳定问题或使SQL Server性能降低的事件集。这些事件包括：数据库文件大小更改操作；错误和警告条件；全文爬网操作；对象*CREATE*、*ALTER*和*DROP*操作；权限或对象所有权更改；内存更改事件。

您不仅无法更改有关已保存文件或其存储位置的任何信息，也无法添加或删除事件、与事件一起捕获的数据及可能适用于事件的筛选器。如果需要产生略微不同于默认跟踪的效果，可以禁用预定跟踪，并使用任意选择的事件、数据和筛选器来创建自己的跟踪。当然，然后还必须确保该跟踪能自动启动，这是完全可以实现的，但我们建议您使用默认跟踪（除非需要其他跟踪），因为这样可以知道至少捕获了有关在SQL Server上运行的活动的信息。

1.11　小结

本章介绍了SQL Server引擎的一般工作机制，包括构成引擎的关键组件和功能区域。另外，还介绍了SQL Server和操作系统之间的交互。在本章中，我根据需要对某些内容进行了简化，但这些信息应该可以使您了解SQL Server中主要组件的角色和职责及组件之间的相互关系。

本章还涉及了用于更改SQL Server行为的主要工具。使用配置选项是更改SQL Server行为的主要方式，因此介绍了可以对SQL Server行为（特别是其性能）产生显著影响的选项。为了真正了解在何时更改SQL Server最合适，您有必要了解SQL Server如何和为什么这样工作。我希望本章已经为您奠定了坚实的基础，使您能根据实际情况确定配置更改。

第 2 章 更改跟踪、跟踪和扩展事件

Adam Machanic

Microsoft SQL Server 引擎在处理用户请求时，可以产生各种操作：询问数据结构，读取或写入文件，分配、释放或访问内存，读取或修改数据，产生错误等。将这些操作分组以后，可以将它们称为 SQL Server 中可能发生的*运行时事件*集合。

从用户（使用 SQL Server 的 DBA 或数据库开发人员）的角度看，发生特定的事件在支持调试、审核和常规服务器维护任务的环境中可能非常有用。例如，跟踪何时产生特定的错误、何时更新特定的列，以及各种存储过程需要消耗多少 CPU 时间等事件对用户可能非常有用。

为了支持这些用户方案，SQL Server 引擎设计了各种基础结构来支持事件消耗。这些结构包括相对简单的系统（如触发器，它们调用用户代码来响应数据修改或其他事件）和复杂而灵活的扩展事件引擎，其中扩展事件引擎是 SQL Server 2008 中的新增功能。

本章将介绍 SQL Server DBA 或数据库开发人员可能遇到的、每个常见事件系统的关键部分：触发器、事件通知、更改跟踪和扩展事件。这些服务都有类似的基本目标——发生情况时进行响应或报告，但每个服务的工作机制略有不同。

2.1 基础知识：触发器和事件通知

虽然本章的主要内容涉及更大和更复杂的事件结构，但通过研究触发器和事件通知更易于了解 SQL Server 如何在内部处理事件。因此，它们是很好的入门工具。

SQL Server 中存在各种类型的触发器。在诸如插入和更新操作上可以激活数据操作语言（Data Manipulation Language，DML）触发器；在服务器级别或数据库级别的操作上可以激活数据定义语言（Data Definition Language，DDL）触发器，如创建登录或删除表操作。可以激活 DML 触发器但不触发事件，或者在事件已经完成之后但未提交之前激活 DML 触发器。同样，需要在事件完成之后和提交之前激活配置的 DDL 触发器。事件通知其实是特殊的 DDL 触发器，它们向 SQL Service Broker 队列发送消息，而不是调用用户程序。事件通知与 DDL 触发器最大的不同在于，它不需要事务，因此它支持大量非事务事件。例如，一名用户从 SQL Server 实例断开连接，标准 DDL 触发器不支持这类非事务事件。

2.1.1 运行时触发器行为

由于 DML 触发器和 DDL 触发器在触发器中具有不同的操作模式和所需的数据性质，因此它们的运行时行为略有不同。DDL 触发器与元数据操作有关，因此与 DML 相比，它需要更少的数据。

DML 触发器在 DML 编译过程中进行解析。解析后，通过内部函数检查每一个相关表，查找存在的触发器。如果找到触发器，包含触发器的表又需要编译和检查触发器，并且该过程将继续递归执行。触

发器在实际的DML操作过程中被激活；插入和删除虚拟表中的行可使用版本存储基础设施填充到*tempdb*数据库中。

DDL触发器和事件通知遵循类似的工作方式，它们与DML触发器的工作机制略有不同。在这两种情况下，只有已经应用了触发器绑定的DDL更改以后，才能通过检查来解析触发器自身。发生DDL操作后，DDL触发器和事件通知才被激活，这是一个POST操作步骤，与DML触发器的操作过程不同。DDL触发器和事件通知的唯一主要区别在于：DDL触发器运行用户定义代码，而事件通知向Service Broker队列发送消息。

2.2　更改跟踪

设计更改跟踪功能是为了帮助消除大量自定义同步架构需求，在应用程序的生存期内，开发人员通常必须从头创建这些自定义同步架构。这种类型的系统示例有，当应用程序从数据库获取数据再存入本地缓存中，而且偶尔需要询问数据库是否已经被更新时，使用更改跟踪可以使本地存储中的数据保持最新。这些系统大部分是使用触发器或时间戳来实现的，它们通常伴随着性能问题或微小的逻辑缺陷。例如，如果在插入时间而不是提交时间填充时间戳列时，使用时间戳的架构通常就会分解。如果大型插入和许多小型插入事件同时发生，使用时间戳就会出现问题。因为大型插入的提交比后来启动的小型插入要迟，因此会破坏时间戳的升序性质。触发器可以对这类特殊问题进行修补，但它们也会引发自身的问题，即它们延长了事务提交所需的时间，因此会带来阻塞问题。

与自定义系统不同，更改跟踪已经完全集成到了SQL Server关系引擎中，并且从开始设计起就充分考虑了性能和可伸缩性。设计更改跟踪系统的目的是，跟踪数据库中一个或多个表的数据更改，并允许用户轻松确定发生更改的顺序，从而成为一种支持多表同步的方式。作为事务发生更改的一部分，更改是被同步跟踪的，这意味着更改行列表始终是最新的，并与表中的实际数据保持一致。

更改跟踪是基于某个基线的正向工作（working forward）构思。数据用户首先请求跟踪表中所有行的当前状态，获取版本号和每行的信息。在出现下一个同步请求之前，查询基线版本号（实际上是系统当前知道的最大版本号）并进行记录。请求完成以后，基线版本号将返回跟踪系统中，系统将确定从第一个请求发出以后哪些行被修改过。这样用户只需要关注自身的增量，通常不需要重新获取未变化的行。除了发送一系列变化的行之外，系统还标识基线以后的变化性质——新行、对现有行的更新或删除行。请求更新变成新基线时，系统将返回最大行版本。

SQL Server 2008包含两个可用于支持同步化的相似技术：更改跟踪和变更数据捕获（变更数据捕获的详细内容超出了本书的范围，因为它本身不是引擎功能，它使用外部日志读取器完成自己的工作）。我们有必要稍微花一些时间来讨论使用更改跟踪的场合和时间。设计更改跟踪的目的在于支持脱机应用程序，偶尔也可用于支持联机应用程序，以及其他一些数据更新时不需要实时通知的应用程序。更改跟踪系统仅返回基线之后请求的当前行版本（不保留增量行的状态），因此理想的更改跟踪应用程序不需要知道指定行的所有历史信息。与变更数据捕获相比（它记录每个行的完整修改历史），更改跟踪是轻量级的，较少应用于审核和数据仓库提取、转换和加载（ETL）场景中。

2.2.1　更改跟踪配置

虽然设计更改跟踪是为了跟踪基于表的变化，但实际配置涉及两个级别：表属于的数据库和表自身。只有所在的数据库中启用更改跟踪功能以后，才能启用应用于更改跟踪的表。

1. 数据库级别配置

SQL Server 2008 扩展了 *ALTER DATABASE* 命令以支持启用和禁用更改跟踪，以及配置选项（用于定义参与表的历史更改信息是否被清除和多久清除一次）。为了启用数据库的更改跟踪的默认选项，使用以下 *ALTER DATABASE* 语法：

```
ALTER DATABASE AdventureWorks2008
SET CHANGE_TRACKING = ON;
```

运行该语句启用元数据配置更改，启用表级别配置后，元数据允许发生两个相关的更改：首先，目标数据库中的隐藏系统表将被填充，需要对事务发生进行资格鉴定（参考下一节）；其次，清除任务将开始清除在内部表或相关表中发现的旧行。

2. 提交表

隐藏表（也称为提交表）在数据库中维护每个事务的一行，该数据库至少修改参与更改跟踪的表中的一行。在事务提交时，每个合格事务被分配一个唯一的称为提交序号（CSN）的升序标识符。然后，CSN（与事务标识符、日志序列信息、开始时间和其他数据一起）将被插入提交表中。提交表对更改跟踪过程非常重要。通过维护一系列提交事务，当用户请求更新时，提交表用于帮助确定需要同步哪些更改。

虽然提交表是内部表，但用户（除管理员之外）无法直接访问它，通过专用管理员连接（DAC）仍然可以从 *sys.all_columns* 目录视图查看提交表的列和索引。以下查询将返回提交表的 6 行，如表 2-1 所示。

```
SELECT *
FROM sys.all_columns
WHERE object_id = OBJECT_ID('sys.syscommittab');
```

表 2-1　*sys.syscommittab* 系统表中的列

列　名	类　型	说　明
commit_ts	*BIGINT*	事务的升序 CSN
xdes_id	*BIGINT*	事务的内部标识符
commit_lbn	*BIGINT*	事务的日志块编号
commit_csn	*BIGINT*	事务的实例范围内序号
commit_time	*DATETIME*	提交事务的时间
dbfragid	*INT*	保留供将来使用

sys.syscommitab 表有两个索引（通过 *sys.indexes* 目录视图可以查看）：*commit_ts* 和 *xdes_id* 列上唯一的聚集索引和 *xdes_id* 列（包含 *dbfragid* 列）上的非聚集索引。这两个列都不能为空，因此聚集索引的每行数据大小是 44 字节，非聚集索引的每行数据大小是 20 字节。

注意，该表记录了有关事务的信息，但没有记录表中行的修改信息。有关表中行的修改信息存储在单独的系统表中，该系统表是在用户表上启用更改跟踪时创建的。由于一个事务可以跨越多个不同的表和每个表中的多个列，因此在大型事务过程中，在单个中心表中储存特定于事务的数据可以节省很大的空间。

通过新 *sys.dm_tran_commit_table* DMV 可以查看 *sys.syscommitab* 表中所有的列（*dbfragid* 列除外）。*SQL Server 联机丛书*将该视图描述为“支持性用途”，但对于学习更改跟踪行为的内部机制、观察清除任务的操作（将在下节讨论），该视图也非常有用。

3. 内部清除任务

启用更改跟踪和提交表后，相关隐藏表中包含许多行，它们将在数据库中占用大量空间。使用者（即同步数据库和应用程序）可能不需要超过某个时间点来更改记录，因此保留这些记录将浪费空间。为了消除这种开销，更改跟踪包含相应的功能来启动内部任务，定期删除历史记录。

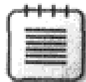
使用前面列出的语法启动更改跟踪时，将使用默认设置“Remove History Older Than Two Days”。对 *ALTER DATABASE* 语法使用选项参数时，在启用更改跟踪时可以指定该选项：

```
ALTER DATABASE AdventureWorks2008
SET CHANGE_TRACKING = ON
(AUTO_CLEANUP=ON, CHANGE_RETENTION=1 hours);
```

可以使用 AUTO_CLEANUP 选项完全禁用内部清除任务，使用 CHANGE_RETENTION 选项指定删除历史记录的时间间隔，可以使用分钟、小时或天来定义时间间隔。

如果启用了内部清除任务，它将每隔 30 分钟运行一次，并用当前时间减去保留间隔得出需要删除的事务，然后进入提交表的接口查找比该时期更旧的事务 ID 列表，最后从提交表和其他隐藏变更跟踪表中清除这些事务。

可以在 *sys.change_tracking_databases* 目录视图中查询每个数据库的当前清除和保留设置。



注意：

设置清除保留间隔时，尽量使用长时间间隔，以确保数据使用者不会由于断开的更改序列而终止。如果这成了问题的瓶颈，应用程序可以使用 *CHANGE_TRACKING_MIN_VALID_VERSION* 函数查找数据库中存储的当前最小版本号。如果最小版本号高于应用程序的当前基线，那么应用程序必须重新同步所有数据并使用新基线。

4. 表级别配置

在数据库级别启用更改跟踪后，必须配置参与的特定表。默认情况下，由于在数据库级别启动更改跟踪功能，因此在更改跟踪中没有登记表。

为了便于在表级别启用更改跟踪，以下代码修改了 *ALTER TABLE* 命令。若需要打开该功能，可以使用新的 ENABLE CHANGE_TRACKING 选项，如下所示：

```
ALTER TABLE HumanResources.Employee
ENABLE CHANGE_TRACKING;
```

如果已经在数据库级别启用了更改跟踪，运行以上语句将引发两个变化：首先，在数据库中创建一个新的内部表来跟踪目标表中行的变化；其次，在目标表中添加一个隐藏列，通过事务 ID 跟踪特定行的变化。另外，还可以启用名为“列跟踪”的可选功能，该内容将在本章后面的“列跟踪”中介绍。

5. 内部更改表

通过在表级别启用的更改跟踪创建的内部表称为 *sys.change_tracking_[object id]*，其中*[object id]*是目标表的数据库对象 ID。通过查询 *sys.all_objects* 目录视图，并基于关注的表对象 ID 筛选 *parent_object_id* 列，或者使用 209 *internal_type* 查看表的 *sys.internal_tables* 视图来查看该表。

该内部表包含 5 个静态列和至少 1 个附加列（取决于目标表主键中使用的列数），如表 2-2 所示。

表 2-2　　内部更改跟踪表中的列

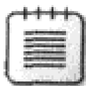
列　名	类　型	说　明
sys_change_xdes_id	*BIGINT NOT NULL*	修改该行事务的事务 ID
sys_change_xdes_id_seq	*BIGINT NOT NULL (IDENTITY)*	在事务中操作的序列标识符
sys_change_operation	*NCHAR(1) NULL*	影响该行的操作类型：插入、更新或删除
sys_change_columns	*VARBINARY(4100) NULL*	修改的列的列表（用于更新，仅当启用跟踪时可用）
sys_change_context	*VARBINARY(128) NULL*	在 DML 操作过程中使用 WITH CHANGE_ TRACKING_ CONTEXT 选项提供的特定于应用程序的上下文信息
k_[name]_[ord]	*[type] NOT NULL*	目标表的主键。*[name]*是主键列的名称；*[ord]*是键中序号的位置；*[type]*是列的数据类型

计算内部表每行的开销比计算提交表更复杂一些，因为几个因素都可以影响总体行大小。固定开销包括 18 字节的事务 ID、CSN 和操作类型，以及来自目标表的主键大小。如果是更新操作，并且启用了列跟踪（该内容将在本章的“列跟踪”中介绍），*sys_change_columns* 列的每行可能最多需要消耗 4 100 个附加字节。另外，使用新的 WITH CHANGE_TRACKING_CONTEXT DML 选项（参考本章的“查询处理和 DML 操作”）获取上下文信息（如应用程序的名称或用户更改信息），最多将在每行额外增加 128 字节。

内部表在事务 ID 上包含唯一的聚集索引和事务序列标识符，不包含非聚集索引。

6. 更改跟踪的隐藏列

为某个表启用更改跟踪时，除了创建内部表之外，还将向该表添加一个隐藏的 8 字节列，用于记录每行最后修改的事务 ID。在任何关系引擎元数据（即目录视图等）中都无法查看该列，但可以看到它在诸如*$sys_change_xdes_id* 的查询计划中的引用情况。另外，在更新更改跟踪后，您可能注意到表的数据大小相应地增加了。如果某个表禁用了更改跟踪，该列将和内部表一起被删除。



注意：

通过 DAC 进行连接和显式引用列名可以查看隐藏列的值，但它永远不会在 *SELECT* *查询结果中显示。

2.2.2 更改跟踪的运行时行为

当更改跟踪在运行时与查询处理器连接时，目前所涉及的各种隐藏和内部对象都有特定的作用。除了启用 *CHANGETABLE* 函数（该函数允许数据使用者查找哪些行已经更改了并且需要被同步）之外，也

可以启用表的更改跟踪，借助表修改每个后续 DML 操作的行为。

1. 查询处理和 DML 操作

为某个指定的表启用更改跟踪后，该表所有涉及行修改的现有查询计划都将被标记为重新编译。该表中涉及行修改的新计划包括内部更改表的某个插入操作，如图 2-1 所示。因为内部表代表所有的操作(插入、更新和删除)，所以通过插入新行操作添加到每个新查询计划中的子树几乎是相同的。

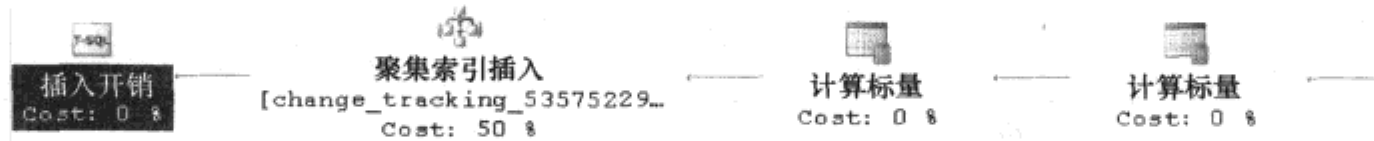
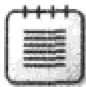


图 2-1　涉及向内部更改表执行查询操作的查询计划子树

由于存在更改跟踪的 *WITH CHANGE_TRACKING_CONTEXT* 函数，所以除了向内部表插入之外，查询处理器开始处理新的 DML 选项。*WITH CHANGE_TRACKING_CONTEXT* 函数允许在内部表的 *sys_change_context* 列存储多达 128 字节的二进制数据和其他相关更改信息。开发人员可以使用 *sys_change_context* 列保留有关哪个应用程序或用户给出指定更改的信息，并使用更改跟踪系统作为元数据存储库存储有关行的更改情况。

该选项的语法类似于公用表表达式，并应用在 DML 查询的开头，如下所示：

```
DECLARE @context VARBINARY(128) =
    CONVERT(VARBINARY(128), SUSER_SNAME());

WITH CHANGE_TRACKING_CONTEXT(@context)
UPDATE AdventureWorks2008.HumanResources.Employee
SET
    JobTitle = 'Production Engineer'
WHERE
    JobTitle = 'Design Engineer';
```



注意：

对于未启用更改跟踪的表而言，该语法完全合法。不过在这些情况下，查询处理器将忽略对 *CHANGE_TRACKING_CONTEXT* 函数的调用。

不仅内部表的插入操作在事务结束时同步发生，提交表的插入操作也在提交时发生。插入的行包含在内部表和目标表的隐藏列中使用的相同事务 ID。此时，还将为该事务分配一个 CSN，因此可以将 CSN 看成是版本号，该版本号应用于事务修改的所有行。

2. 列跟踪

使用包含大量列的表或者表包含一个或多个非常宽的列时，不从未更新的列中重新获取数据可以优化同步进程。为了支持这种优化，更改跟踪具有称为列跟踪的功能，仅当内部表出现更新操作时，该功能将记录列的更新情况。

列表保留在内部表的 *sys_change_columns* 列中。每个列存储为一个整数，并且每个列表最多可以存储 1 024 列。如果在某个事务中修改的列数超过 1 024，则不存储列表，应用程序必须重新获取整行数据。

为了启用列跟踪功能，*ALTER TABLE* 语句使用名为 TRACK_COLUMNS_UPDATED 的开关，如下所示：

```
ALTER TABLE HumanResources.Employee
ENABLE CHANGE_TRACKING
WITH (TRACK_COLUMNS_UPDATED = ON);
```

启用列功能后，通过 *CHANGETABLE(CHANGES)*输出函数返回更改的列表，下一节将说明该输出函数。为了显示某个特定的列，使用 *CHANGE_TRACKING_IS_ COLUMN_IN_MASK* 函数可以估计位图。

> **警告：**
>
> 为活动表启用列跟踪必须非常谨慎。虽然该功能由于同步化减少了发送的字节而有助于优化同步进程，但它也增加了对于目标表的每个更新而必须写入的字节数。如果列不是足够大，无法平衡位图的附加字节需求时，启用列跟踪可能会导致整体性能的下降。

3. *CHANGETABLE* 函数

CHANGETABLE 函数是用户用于平衡更改跟踪系统的主要 API。该函数具有双重作用：返回目标表中所有行的基线版本，以及返回一组仅包含更新版本和相关变更的信息。该函数在各种内部和隐藏结构的帮助下完成这些任务，这些内部和隐藏结构是在为数据库中的指定表或表集合启用更改跟踪时创建并填充的。

CHANGETABLE 是系统表值函数，但与其他表值函数不同的是，它的结果形式随着输入参数的不同在运行时发生改变。在 VERSION 模式（用于获取表中每行的基线值）下，该函数仅返回每行的主键、版本号和上下文信息。在 CHANGES 模式（用于获取更新行的列表）下，该函数还返回影响更改和列表的相关操作。

设计 *CHANGETABLE* 的 VERSION 模式的目的在于帮助调用方获取基线，因此在该模式下调用 *CHANGETABLE* 函数需要连接目标表，如下所示：

```
SELECT
    c.SYS_CHANGE_VERSION,
    c.SYS_CHANGE_CONTEXT,
    e.*
FROM AdventureWorks2008.HumanResources.Employee e
CROSS APPLY CHANGETABLE
(
    VERSION AdventureWorks2008.HumanResources.Employee,
     (BusinessEntityId),
     (e.BusinessEntityId)
) c;
```

以上是该功能的快速演练示例。在 VERSION 模式下，函数的第一个参数是目标表，第二个参数是目标表上主键列的逗号分隔列表，第三个参数是逗号分隔列表，它们和查询使用的目标表的有关主键列顺序相同。这些列在内部需要按这种顺序进行关联，以支持必要的连接来获取每行的基线版本。

执行该查询时，查询处理器将扫描目标表，访问每行并获取每列及隐藏列的值（修改该行的上一个事务 ID）。使用该事务 ID 作为连接到提交表的键，以选取相关的 CSN 并填充 *sys_change_version* 列。为了填充 *sys_change_context* 列，也可以使用该事务 ID 和主键来连接内部跟踪表。

获取基线后，数据使用者负责调用 *CHANGE_ TRACKING_CURRENT_VERSION* 函数，该函数用于返回数据库中当前存储的最大更改跟踪版本号。该版本号将成为应用程序将来用于同步化请求的基线版本号，它在 CHANGES 模式中传给 *CHANGETABLE* 函数，并获取表中行的后续版本，如下所示：

```
DECLARE @last_version BIGINT = 8;

SELECT
    c.*
FROM CHANGETABLE
(
    CHANGES AdventureWorks2008.HumanResources.Employee,
    @last_version
) c;
```

该查询将返回版本号为 8 之后的所有更改行列表，以及哪些操作修改了每一行的信息。注意，输出仅反映出运行该查询后的最新行版本。例如，如果现有的行版本是 8，随后该版本被更新了 3 次，最后被删除，则该查询将仅显示行的最后一个更改即删除。该查询在其输出中包含了更改的主键，因此可以连到目标表中获取每行已更改的最新版本。在该语句中使用 OUTER JOIN 必须谨慎，因为如果某行被删除后，它就不存在了，无法再进行连接。如下所示：

```
DECLARE @last_version BIGINT = 8;

SELECT
    c.SYS_CHANGE_VERSION,
    c.SYS_CHANGE_OPERATION,
    c.SYS_CHANGE_CONTEXT,
    e.*
FROM CHANGETABLE
(
    CHANGES AdventureWorks2008.HumanResources.Employee,
    @last_version
) c
LEFT OUTER JOIN AdventureWorks2008.HumanResources.Employee e ON
    e.BusinessEntityID = c.BusinessEntityID;
```

在 CHANGES 模式下运行 *CHANGETABLE* 函数时，使用的各种内部结构与 VERSION 示例中使用的结构略有不同。该过程的第一步是查询提交表，以获取 CSN 大于传递到该函数中的 CSN 的所有相关事务的 ID。随后，该事务 ID 列表将用于查询内部跟踪，以获取与这些事务更改相关的主键。为了查找每个主键的最新行，必须对该阶段产生的行进行聚合，聚合的标准是根据内部表的主键和事务序列标识符。

因为行可能一直在发生变化，包括应用程序正在请求更改列表时，所以使用更改跟踪时必须注意保持一致性。如果应用程序在检索更改的键列表，然后请求行的值，可以利用 SNAPSHOT 隔离；如果使用 JOIN 检索该值，可以利用 READ COMMITTED SNAPSHOT 隔离。这两种方式都可确保产生一致的结果。本书将在第 10 章讨论 SNAPSHOT 隔离和 READ COMMITTED SNAPSHOT 隔离。

2.3 跟踪和事件探查

通过查看 SQL Server 的工作情况，可以进行查询优化、优化和一般故障排除，甚至可以在无法识别故障原因的情况下对问题进行修复。SQL Server 提供了这样一款功能强大的 SQL 跟踪工具，使您能在微

小的粒度级别实时或准实时查看数据库引擎的工作情况。

跟踪工具集为您提供了 180 个可以监视、筛选和操作的事件，以便查看各种情况。这些事件可以是用户登录的总体概述信息，也可以是诸如特定会话 id（SPID）完成的锁活动的细粒度信息。通过使用 SQL Server Profiler、一系列服务器内部的存储过程及.NET 类，就可以获取各种信息数据，并在出现问题时灵活启用自定义解决方案。

2.3.1 SQL 跟踪架构和术语

SQL 跟踪是一种 SQL Server 数据库引擎技术，客户端 Profiler 工具其实是服务器端功能的包装器，理解这一点非常重要。启动跟踪时，当数据库引擎中发生各种操作时，我们将监视特定的事件。例如，用户登录到服务器或执行查询都是激活事件的操作。这些事件是通过检测数据库引擎代码来激活的；换言之，特定的代码已经添加到这些和其他执行路径中，当触及执行路径时，将激活不同的事件。

每个事件都有相关联的“列”集合，它们是一些属性，包含事件激活时收集的各种数据。以查询为例，我们可以收集查询的各种数据，包括查询的启动时间、查询的执行时间和查询使用的 CPU 时间。最后，每个跟踪可以指定筛选器，筛选器基于标准集限制返回的结果。例如，可以指定仅返回执行时间超过 50 毫秒的事件。

因为可以从 180 个事件和 66 个列中选择数据，所以收集的数据点量非常大。并不是每个事件都可以使用每个列，但允许使用的全部组合将超过 4 000。考虑到存储所有这些数据的内存利用率和创建数据所需的处理器时间，SQL Server 在生成如此多信息的同时如何有效地运行呢？您可能对此很感兴趣。答案是，没有收到请求之前，SQL Server 实际上并不收集任何数据。相反，该模式只在需要时才有选择地启用集合。

1. 内部跟踪组件

SQL 跟踪架构的中心组件是*跟踪控制器*，它是一个共享资源，管理任何使用者创建的所有跟踪。数据库引擎中存在各种*事件创建器*，例如，可以在查询处理器、锁管理器和缓存管理器中找到这些创建器。每个事件创建器负责生成属于特定服务器活动目录的事件，但默认情况下这些创建器是禁用的，因此不生成数据。当某个用户为特定事件请求启动跟踪时，该跟踪控制器中的全局位图将更新，通知事件创建器至少有一个跟踪正在侦听，并开始激活该事件。与该位图一起管理的是辅助列表，该列表列出了哪些跟踪正在监视哪些事件。

激活事件后，事件的数据将路由到一个全局事件接收器中，全局事件接收器将该事件的数据进行排队，然后分配到每个正在侦听的跟踪中。跟踪控制器根据自身的内部跟踪列表和跟踪事件向每个侦听跟踪路由数据。除了跟踪控制器自身的列表之外，每个单独的跟踪还跟踪它正在监控哪些事件、实际正在使用哪些列及使用什么筛选器比较合适等信息。跟踪控制器向每个跟踪返回的事件数据都经过了筛选，在数据路由到 *I/O 提供程序*之前，数据列进行了必要的裁剪。

2. 跟踪 I/O 提供程序

跟踪 I/O 提供程序实际是指如何向最终目的地发送数据。跟踪数据可用的输出格式要么是数据库服务器上（或网络共享）的文件，要么是客户端的行集。这两种提供程序都需要使用内部缓冲，以便确保如果没有及时使用数据（即写入磁盘或从行集中读取），数据将排队等待。但是，这与队列超出管理规模时，提供程序如何处理这种情况存在很大差别。

设计*文件提供程序*的目的在于保证不会丢失事件数据。若 I/O 速度减缓或出现停顿，为了确保该文件提供程序在磁盘写入的速度不够快的情况下仍能正常工作，数据会开始进行内部缓冲器填充。缓冲区填满数据后，向该跟踪发送事件数据的线程将等待缓冲区释放空间。为了避免线程等待跟踪缓冲区，请务必使用足够快的磁盘系统执行跟踪。为了监视这些等待，可以观察 SQLTRACE_LOCK 和 IO_COMPLETION 等待类型。

相反，设计*行集提供程序*的目的不是为了保证丢弃所有数据。如果没有及时使用数据，并且内部缓冲区已满，数据最多等待 20 秒钟，然后它将丢弃事件以释放缓冲区并继续处理随后的事件。如果事件已经被删除，SQL Server Profiler 客户端工具将发送特殊的错误消息。但是，通过在 SQL Server 中监视 TRACEWRITE 等待类型，也可以发现自己是否可能收到错误消息，随着线程等待缓冲区释放空间，收到错误消息的可能性将逐渐增加。

只要服务器上至少存在一个活动的跟踪，就会启动后台跟踪管理线程。该后台线程负责刷新文件提供程序的缓冲区（每 4 秒钟完成一次），另外还负责关闭基于行集的过期跟踪（如果跟踪删除事件已经超过了 10 分钟，将出现过期跟踪）。通过不定期刷新文件提供程序缓冲区，而不是每次收集了一个事件就向磁盘写入数据，SQL Server 可以利用大数据块写入操作降低跟踪的开销，特别是在活动性极高的服务器上，效果尤为明显。

不熟悉 SQL Server 的 DBA 经常询问的一个问题是，为什么没有可以直接向表写入跟踪数据的提供程序呢？出现这种限制的原因在于这类活动所需的开销量。因为表不支持大数据块写入，SQL Server 必须逐行写入事件数据。事件开销引起的性能开销需要删除很多事件；如果强制使用无损保证，就会出现大量阻塞。任何一种情况都不能接受，因此 SQL Server 不提供表的直接写入功能。但是，在本章的后面我们将看到，在跟踪过程中或者跟踪之后向表装载数据非常简单，因此，这种限制无关紧要。

2.3.2 安全性和权限

跟踪不仅可以展示大量有关服务器状态的信息，还可以展示用户从数据库引擎发送或返回的数据。从监视单个查询到监视批处理查询，甚至是监视查询计划级别，跟踪功能在表现其强大功能的同时也令人担忧：即使是展示储存过程输入参数也可以为攻击者提供大量有关数据库的数据信息。

为了防止用户从 SQL 跟踪展示的信息中获取不应该查看的数据，SQL Server 2005 之前的版本仅允许管理员用户（*sysadmin* 固定服务器角色成员）启动跟踪。这种限制机制对许多开发小组来说缺乏弹性，因此现在已经有所松动。

1. ALTER TRACE 权限

从 SQL Server 2005 开始，出现了名为 ALTER TRACE 的新权限。该权限是服务器级别的权限（授权给登录方），除了提供生成用户定义事件权限之外，它还允许授权方启动、停止或修改跟踪。

提示：

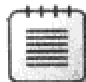
注意，ALTER TRACE 权限是在服务器级别授予的，因此访问也在服务器级别；如果用户可以启动某个跟踪，他就可以检索事件数据，而无需考虑在哪个数据库中生成事件。SQL Server 中包含该权限对开发人员而言是一个正确的导向，开发人员可能需要在产品系统上运行跟踪以调试应用程序问题，因此该权限有利于开发人员处理这些问题。但在授予该权限的时候需要慎重，即使它与完全 *sysadmin* 权限有很大不同，但它仍然是一个潜在安全问题的线程。

要向某个登录授权 ALTER TRACE 权限，可以按以下示例使用 *GRANT* 语句（在该示例中，授予权限的服务器方称为“Jane”）：

```
GRANT ALTER TRACE TO Jane;
```

2. 保护敏感事件数据

除了使用锁定策略限制特定的用户使用 SQL 跟踪之外，跟踪引擎自身也有一些内置的安全功能来防止非法用户（包括具有使用跟踪权限的用户）查看私有信息。如果事件包含与密码有关的存储过程或语句的调用，SQL 跟踪将自动忽略该数据。例如，SQL 跟踪将取消包含 WITH PASSWORD 选项的 *CREATE LOGIN* 调用。



注意：

在 SQL Server 2005 之前的版本中，如果查询文本中的任何地方包含字符串 *sp_password*，SQL 跟踪将自动取消该查询。SQL Server 2005 和 SQL Server 2008 删除了这种功能，因此您不能依靠它来保护私有信息。

加密模块是 SQL 跟踪的另一个安全功能。SQL 跟踪不返回在加密存储过程、用户定义函数或视图中生成的语句文本或查询计划。另外，加密模块还有助于保护特别敏感的数据，甚至对那些具有查看跟踪权限的用户同样有效。

2.3.3 Profiler 入门

SQL Server 2008 捆绑发行了 Profiler，Profiler 是一个功能强大的用户界面工具，用于创建、操作和管理跟踪。该工具主要用于跟踪活动，它可以帮助您轻松启动并运行跟踪，它可能是用于快速诊断数据库问题的最重要 SQL Server 组件了。另外，Profiler 还向工具集添加了一些 SQL 跟踪自身无法获取的新功能。作为跟踪功能的补充，本节将讨论这些新功能。

1. Profiler 基础

在 SQL Server 2008“开始”文件夹（通过单击“开始”，依次选择“所有程序”、“SQL Server 2008”、“Performance Toos”、“SQL Server Profiler”可以获取该工具）的 Performance Tools 子文件夹中可以获取 Profiler 工具。启动该工具后，您将看到一个空白屏幕。依次单击“File”、“New Trace”，然后连到某个 SQL Server 实例。此时，将显示带有两个选项卡的“Trace Properties”对话框，这两个选项卡为“General”和“Events Selection”。

使用者通过“General”选项卡（如图 2-2 所示）可以控制如何处理跟踪。默认设置是使用行集提供程序，在 SQL Server Profiler 窗口实时显示事件。也可以通过选项将事件保存在文件（在服务器上或客户端）或表中。但是，对于繁忙的服务器，我们一般建议您不使用这些选项。

要求 Profiler 在服务器端文件（通过选择 Server Processes Trace Data 选项）保存事件时，Profiler 实际上同时启动了两个相同的跟踪，一个跟踪使用行集提供程序，另一个跟踪使用文件提供程序。使用两个跟踪意味着需要双倍的开销，因此这种方法不可取。本章后面的 “服务器端跟踪和收集”一节将讨论该内容，介绍如何使用文件提供程序建立跟踪。通过文件提供程序，可以将事件有效地保存到服务器端文件中。将事件保存到客户端文件不需要使用文件提供程序，而是通过行集提供程序先将数据路由到

Profiler 工具中，然后从 Profiler 工具中转移到文件中进行保存。这样做比使用 Profiler 工具写入服务器端文件更有效，但由于使用了行集提供程序，因此对网络带宽有较高要求，并且无法获取文件提供程序提供的无损保证功能。

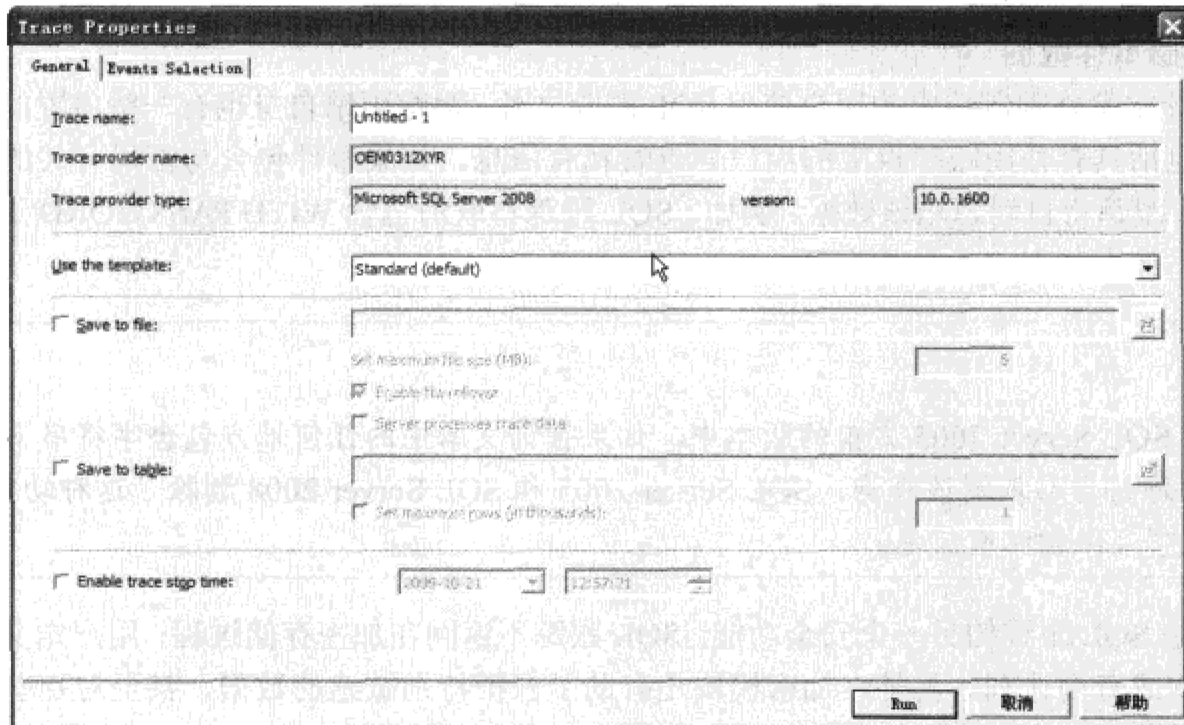


图 2-2　为跟踪选择 I/O 提供程序

注意：

看到 Save To Table 选项，您可能觉得奇怪，为什么我们在本章前面讲在 SQL 跟踪中不能直接将事件写入表呢？实际上是因为 SQL 跟踪没有表输出提供程序。相反，使用该选项时，Profiler 工具将使用行集提供程序，并将数据路由回表中。如果保存的表和跟踪的事件位于同一台服务器上，将产生大量的服务器开销和带宽使用率。因此，如果必须使用该选项，建议将数据保存到另一台服务器的某个表中。另外，Profiler 在您完成跟踪后，还将为您提供一个保存数据的选项，在大多数情况下这是一个可扩展选项。

如图 2-3 所示，在 Profiler 中，您将花费大部分时间在 Events Selection 选项卡中配置跟踪信息。通过该选项卡可以选择需要跟踪的事件和相关的数据列。如图 2-3 所示，当跟踪启动时（*ExistingConnection* 事件）、当出现登录或注销时（*Audit Login* 和 *Audit Logout* 事件）、当远程过程调用完成时（*RPC:Completed* 事件），以及当 T-SQL 批处理启动或完成时（*SQL:BatchStarting* 和 *SQL:BatchCompleted*），默认选项将收集所有连接信息。默认方式下将隐藏事件和可用的数据列的完整列表。选中 Show all events 和 Show all columns 复选框，使 UI 显示所有可用的选择项。

这些默认选择项是一个良好的起点，可用做大量一般需求跟踪的基础。DBA 使用 SQL 跟踪回答的最简单问题都是以查询开销和/或持续时间为基础的。什么是最长查询，或者什么查询使用的资源最多？默认选择项都可以回答这些问题。但在活动服务器上，必须收集大量的数据，这不仅意味着需要做更多的工作才能回答这些问题，还意味着服务器需要完成更多的工作才能收集和分配这么多数据。

为了缩小范围，并确保跟踪不引起性能问题，SQL 跟踪基于各种标准提供了筛选事件功能。在 SQL

Profiler 的 Events Selection 选项卡中，通过 Column Filters 按钮可以使用筛选功能。单击该按钮，显示类似于如图 2-4 所示的 Edit Filter 对话框。在本例中，我们只需要查看持续时间大于或等于 200 毫秒的事件。当然，这只是一个任意数；对于特定的应用程序，只有增强对跟踪需求的了解，才能逐步发现筛选器条件的最佳选择，使该数不断增大，直到从跟踪中获取大部分预期事件为止（此处指那些持续时间长的事件）。通过这种方式，可以方便、快捷地隔离最慢的查询。

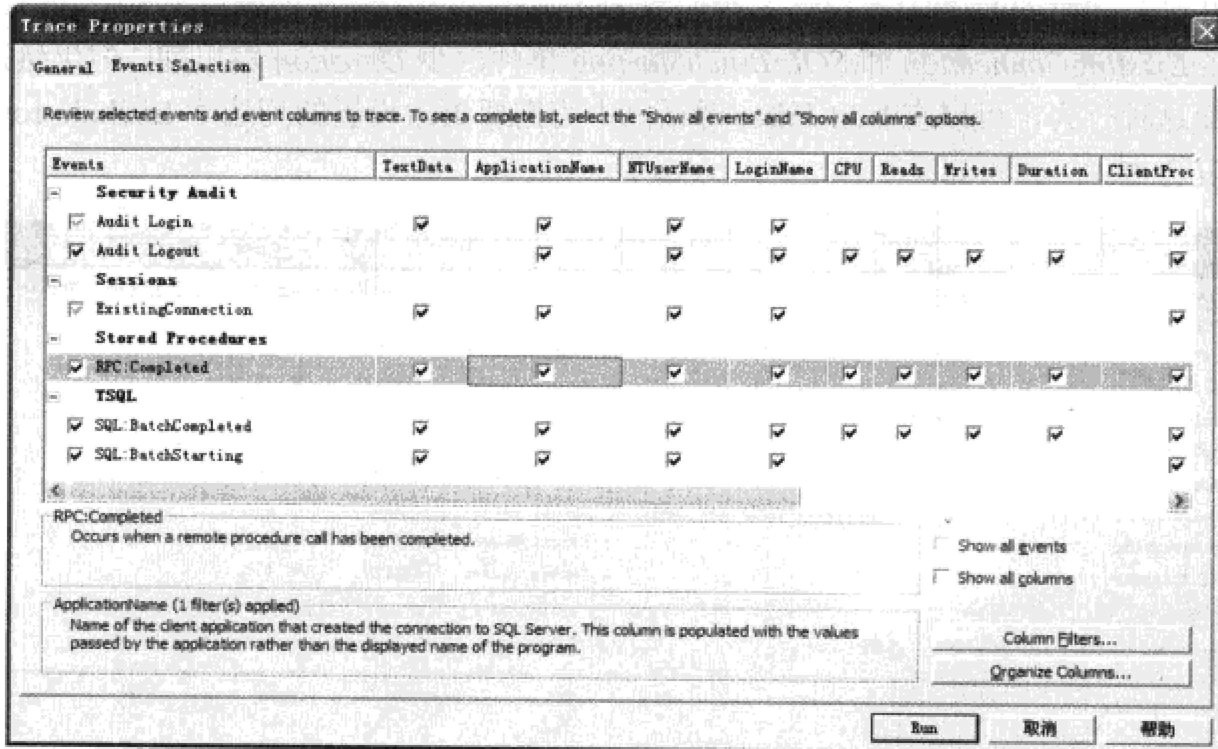


图 2-3　为事件选择事件/列组合

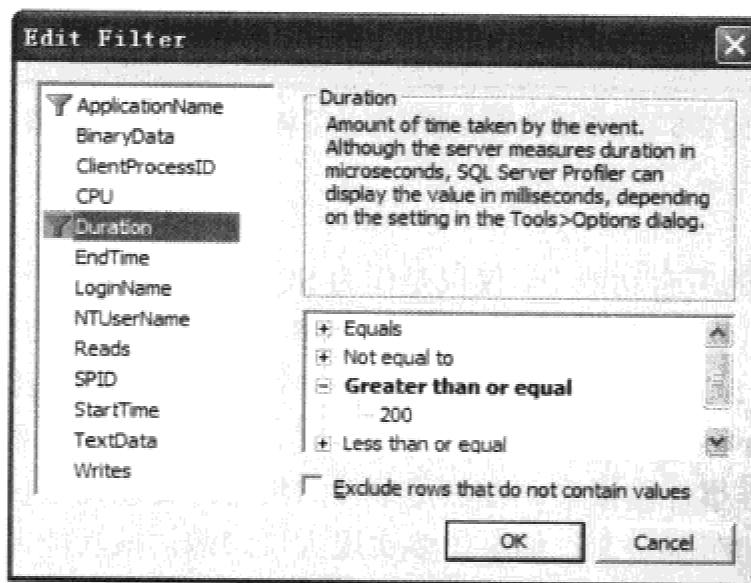


图 2-4　为查询时间大于 200 毫秒的事件定义筛选器

提示：

使用 SQL Profiler 作为筛选器获取的数据列列表和在外部 Events Selection 用户界面获取的列表完全相同。

选择事件和定义筛选器后，可以启动跟踪。在 Trace Properties 对话框中，单击 Run 按钮。Profiler 使用行集提供程序，因此运行跟踪后数据将开始迅速回流。如果您发现数据显示的速度太快而无法读取，可以考虑在 SQL Profiler 工具栏上使用 Auto Scroll Window 按钮来禁用自动滚动。

有关筛选器需要注意的是，在默认方式下，如果跟踪为某个不产生数据的列定义了筛选器，则该事件无法被筛选。例如，*SQL:BatchStarting* 事件不产生持续时间数据（认为该批处理开始的时间和提交给服务器的时间几乎是同时的）。图 2-5 展示了我们将 *Duration* 列上的筛选器值设为大于 200 毫秒的跟踪结果。注意，虽然 *ExistingConnection* 和 *SQL:BatchStarting* 事件缺少 *Duration* 输出列，但仍然返回了这两个事件。为了改变这种行为，可以在 Edit Filter 对话框中为需要更改设置的列选中 Exclude rows that do not contain values 复选框。

Untitled - 1 (OE■0312XYR)

EventClass	TextData	ApplicationName	NTUserName	LoginName	CPU	Reads	Writes	Duration
Trace Start								
ExistingConnection	-- network protocol: LPC set quote...	Report Server	SYSTEM	NT AUT...				
ExistingConnection	-- network protocol: LPC set quote...	Report Server	SYSTEM	NT AUT...				
ExistingConnection	-- network protocol: LPC set quote...	Report Server	SYSTEM	NT AUT...				
ExistingConnection	-- network protocol: LPC set quote...	SQLAgent - G...	SYSTEM	NT AUT...				
Audit Logout		Report Server	SYSTEM	NT AUT...	0	0	0	1000
Audit Login	-- network protocol: LPC set quote...	Report Server	SYSTEM	NT AUT...				
SQL:BatchStarting	...	Report Server	SYSTEM	NT AUT...				
SQL:BatchStarting	...	Report Server	SYSTEM	NT AUT...				
Trace Stop								

Trace is stopped.　Ln 10, Col 1　Rows: 10

图 2-5　跟踪筛选器在默认方式下将空值看成是合法值

2. 保存和重播跟踪

Profiler 在本章已经涉及的所有功能中，仅仅是作为 SQL 跟踪功能的一个包装器。在本章后面的“服务器端跟踪和收集”小节中，我们将介绍 Profiler 的工作机制。不过，我们首先研究 Profiler 提供的一些功能，这些功能使 Profiler 在 SQL 跟踪功能中不再只是一个简单的 UI 包装器。

前面在讨论 Trace Properties 窗口的 General 选项卡时，忽略了默认事件的实际设置原理：默认事件包含在与产品一起发布的标准事件模板中。*模板*是事件的集合，可以在模板中保存列选择项、筛选器和其他设置，以便创建可重用的跟踪定义。如果需要实现大量跟踪，该功能尤其有用；每次需要选项时都重新配置它们，将浪费大量时间。

除了可以保存自己的模板之外，Profiler 还附带了 9 个预定义模板。除了我们已经介绍的标准模板之外，最重要的模板之一是 TSQL_Replay 模板，如图 2-6 所示。该模板为 15 个不同的事件选择了各种列，Profiler 需要其中的每个列都能回放（重播）某个收集的跟踪。通过使用该模板启动跟踪，然后在收集完成后保存跟踪数据，可以完成特定的任务。例如，当某些存储过程按正确的顺序被调用时，可以将某个跟踪作为测试工具，再现可能出现的特定问题。

为阐述该功能，我们使用 TSQL_Replay 模板启动一个新跟踪，并从每个连接（共两个连接）中发送两

个批处理，如图 2-7 所示。第一个 SPID（在本例中为 53）选择 1，第二个 SPID（54）选择 2。回到 SPID 53，它选择了 3，再回到 SPID 54，它选择了 4。在该图中最有用的是第二列 *EventSequence*，可以将该列看成是表的 *IDENTITY* 属性，其值是全局递增的，当跟踪控制器记录事件时，它将创建一个唯一顺序，服务器中的事件将按该顺序发生。这就避免了按 StartTime/EndTime（该列也在跟踪中，但图 2-7 中没有显示）排序时可能出现的问题，因为不存在等同值（*EventSequence* 对每个跟踪是唯一的）。该数是一个 64 位整数，每次服务器重启时该数都将重新设置，因此可以跟踪的事件永远不可能超过 *EventSequence* 的范围。

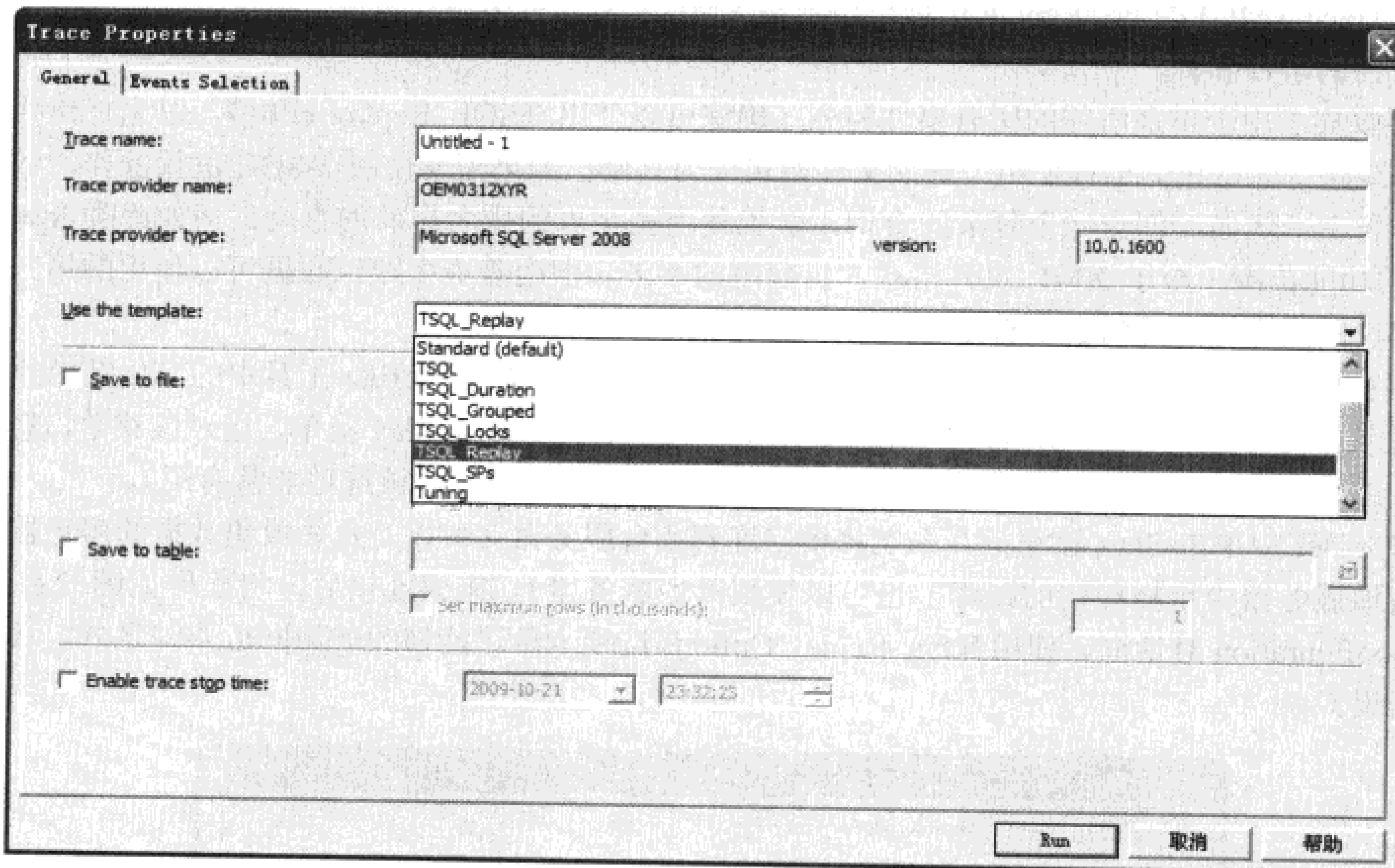


图 2-6　选择 TSQL_Replay 模板

Untitled - 1 (OE■0312XYR)

EventClass	EventSequence	TextData	ApplicationName	SPID
Trace Start				
ExistingConnection	6769	-- network protocol: LPC set quote...	Report Server	52
ExistingConnection	6770	-- network protocol: LPC set quote...	Report Server	53
ExistingConnection	6771	-- network protocol: LPC set quote...	SQLAgent - G...	56
RPC:Starting	6774		Report Server	53
Audit Logout	6775		Report Server	53
RPC:Completed	6776		Report Server	53
Audit Login	6777	-- network protocol: LPC set quote...	Report Server	53
RPC:Starting	6778		Report Server	53
RPC:Completed	6779		Report Server	53
RPC:Starting	6780		Report Server	53

Trace is stopped.　Ln 58, Col 1　Rows: 58

图 2-7　发送交叉批处理的两个 SPID

一旦跟踪数据被收集，在开始重播之前必须先将它保存，然后再重新打开。Profiler 提供以下选项用

于保存跟踪数据，可以从 File 菜单获取这些选项。

- Trace File 选项用于将数据保存到某个使用专用二进制格式进行格式化的文件中。通常这是保存数据的最快方式，同时它也是占用磁盘空间最小的数据保存方式。
- Trace Table 选项用于在新表或数据库先前已创建的表中保存数据。如果需要操作数据或使用 T-SQL 反馈数据，该选项非常有用。
- Trace XML File 选项用于在格式化为 XML 的文本文件中保存数据。
- Trace XML File For Replay 选项也用于在 XML 文本文件中保存数据，但仅保存那些重播功能所需的事件和列。

只要收集了用于重播所需的所有事件和列（确保可以使用 TSQL_Replay 模板），以上任何格式都可用于重播跟踪。通常我们推荐使用二进制文件格式作为基础；如果需要使用 T-SQL 进行操作，数据将保存到某个表中。例如，创建一个复杂的查询，查找使用特定表的排名靠前的查询；诸如此类查询超出了 Profiler 的功能范围。至于 XML 文件格式，目前用的不多。但随着更多跟踪数据可以使用的第三方工具的涌现，我们也许可以看到更多的使用案例。

数据被保存到文件或表中后，可以将原始跟踪窗口关闭，并通过 Profiler 工具中的 File 菜单重新打开该文件或表。按这种方式重新打开跟踪后，Profiler 工具栏上将显示 Replay 菜单，通过该菜单，您可以开始重播跟踪、停止重播或设置断点（当需要测试大型跟踪的某个部分时该选项非常有用）。

在 Profiler 中单击 Start 按钮后，系统将询问连到哪台服务器（可以是从中收集数据的服务器；如果需要在其他服务器中重播相同的跟踪，也可以使用其他服务器）。完成连接后，将出现如图 2-8 所示的 Replay Configuration 对话框。使用 Basic Replay Options 选项卡除了能修改如何回放跟踪之外，还可以保存跟踪结果。

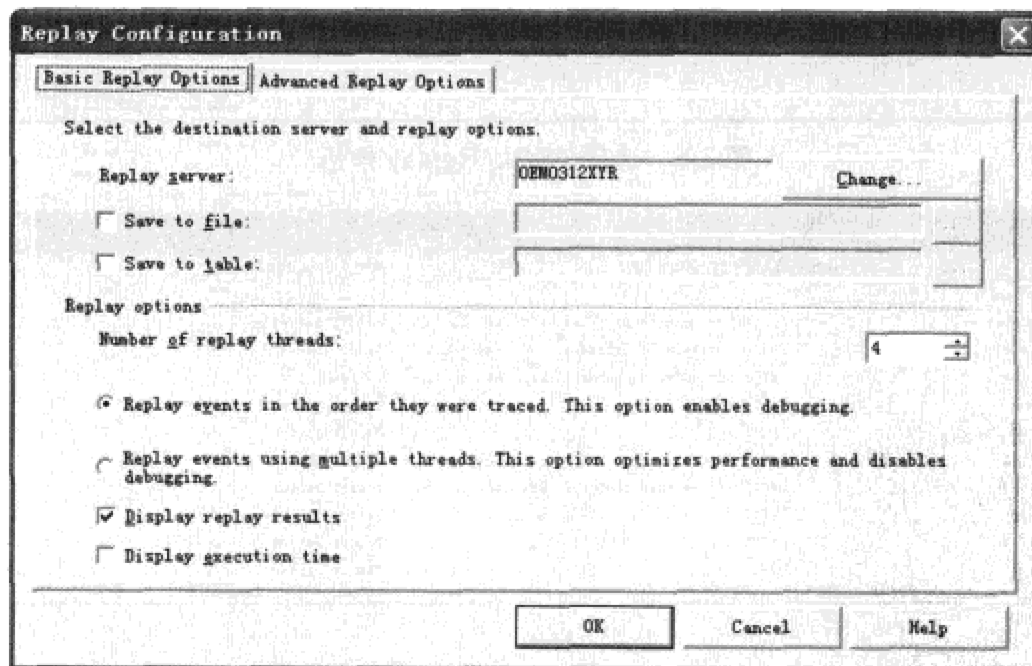


图 2-8　重播配置对话框

在重播过程中，重播的服务器将跟踪用于产生重播跟踪的相同事件。Save to file 和 Save to table 复选框用于在客户端保存数据，没有保存回放结果的服务器端选项。

Replay Configuration 对话框中的 Replay Options 窗格在措辞方面让人觉得有些迷惑。无论您选择哪一个选项，跟踪都在多线程上重播，对应于您在 Number of replay threads 数值框中选择的最大线程数。不

过，选择 Replay events in the order they were traced 选项可以确保所有事件完全按照它们发生的顺序进行回放，这是基于 *EventSequence* 列实现的。在此仍然使用多线程模拟多个 SPID。选择 Replay events using multiple threads 选项，同时允许 Profiler 按每个 SPID 开始执行事件的顺序重新安排次序，以提供重播性能。但是，在给定的 SPID 中，事件的顺序仍然与 *EventSequence* 保持一致。

为了解释这两个选项的区别，我们在如图 2-7 所示的界面中将该跟踪重播两次，每次使用不同的重播选项。图 2-9 展示了使用 Replay in order 选项的结果，图 2-10 展示了使用 Multiple threads 选项的结果。图 2-9 中的结果说明，启动和完成批处理的顺序与最初跟踪批处理的顺序完全相同。而在图 2-10 中，两个参与的 SPID 将所有的事件进行了分组，而不是相互交错的顺序。

图 2-9 使用 Replay In Order 选项进行重播

图 2-10 使用 Multiple Threads 选项进行重播

如果需要重播大量跟踪，而每个 SPID 没有与其他 SPID 关联时，Multiple Threads 选项将非常有用。例如，在测试服务器上，它用于模拟从产品系统中捕获工作负荷。在另一方面，如果需要复制跟踪发生过程中的特定条件，Replay In Order 选项将非常有用。例如，在调试死锁或由多线程访问相同数据的特定

交互产生阻塞条件时，可以使用该选项。

Profiler 是一个功能全面的工具，它广泛支持跟踪和处理跟踪数据的各种操作。但是，如果需要对收集的数据实施高级查询，或者在特别活跃的产品系统上运行跟踪，Profiler 将无法满足需求。再次重申，Profiler 实质上只是数据库引擎功能上的一个包装器，我们不是将它用于跟踪所有阶段，而是在某些情况下，直接利用该工具增加灵活性。以下部分将介绍 Profiler 如何与数据库引擎一起启动、停止和管理跟踪，以及如何利用相同的工具来满足自身的需求。

2.3.4 服务器端跟踪和收集

除了良好的用户界面之外，Profiler 只不过是展示 SQL 跟踪真实功能的部分系统存储过程上的轻型包装器。在本部分，我们将介绍使用哪些存储过程，以及如何作为脚本工具（而不是跟踪界面）来使用 SQL Server Profiler。

以下系统存储过程用于定义和管理跟踪。

- *sp_trace_create*。用于定义跟踪和指定输出文件的位置及其他一些随后即将讲述的操作。该存储过程将返回创建跟踪的句柄（整数形式的跟踪 ID）。
- *sp_trace_setevent*。用于向基于跟踪 ID 的跟踪添加事件/列组合和删除这些组合，如果需要，还可以从已经定义的跟踪中删除组合。
- *sp_trace_setfilter*。用于定义基于跟踪列的事件筛选器。
- *sp_trace_setstatus*。用于调用以打开跟踪、停止跟踪和删除跟踪。在跟踪的有效期内可以多次启动和停止跟踪。

1. 脚本服务器端跟踪

在此我们不是直接研究每个存储过程的语法规范（*SQL Server 联机丛书*已经详细介绍了这些规范），观察它们的操作可能更有用。首先，打开 SQL Server Profiler，使用默认模板启动一个新跟踪，然后清除 *SQL:BatchCompleted* 以外的所有事件，如图 2-11 所示。

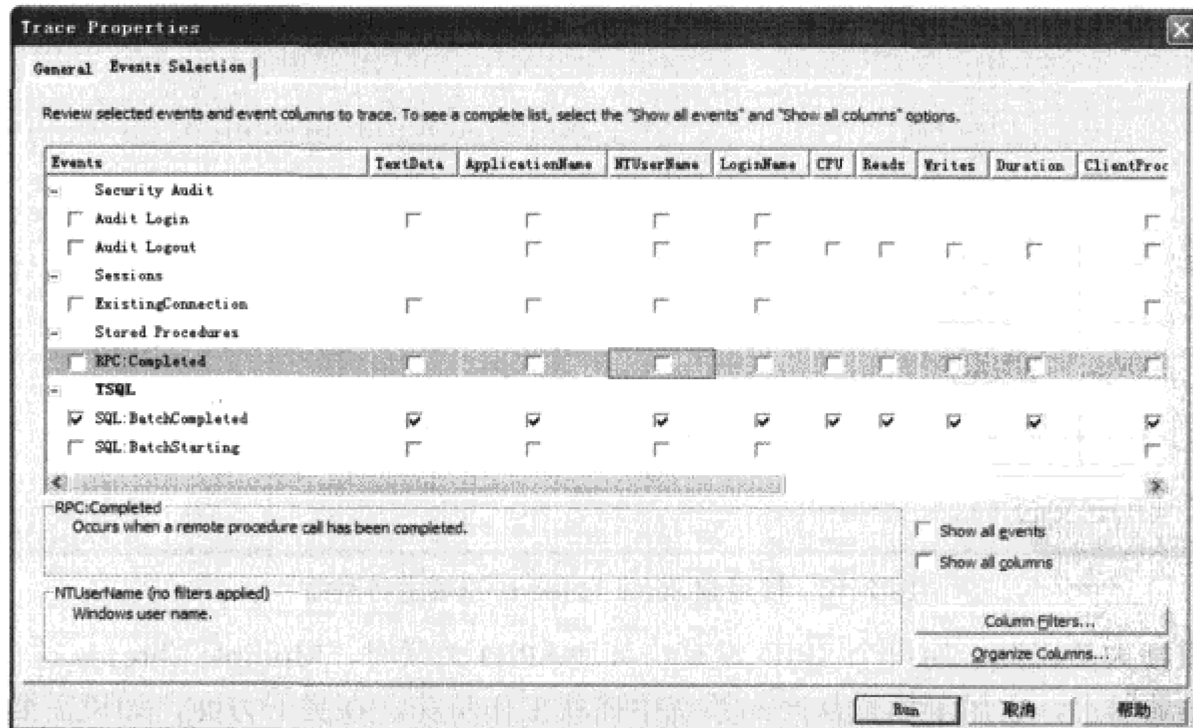


图 2-11 仅选择跟踪 SQL:BatchCompleted 事件

接下来，删除 *ApplicationName* 列（设置为不选取 SQL Server Profiler 事件）上的默认筛选器，并在 *Duration* 列上添加筛选器，筛选条件是大于或等于 10 毫秒，如图 2-12 所示。

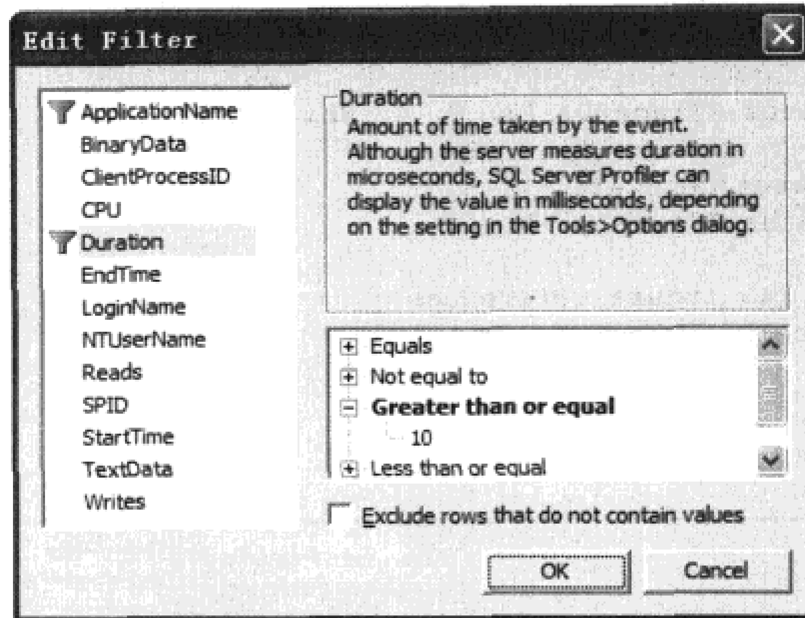


图 2-12 将 *Duration* 列上的筛选器设置为大于等于 10 毫秒

完成以上操作后，单击 Run 按钮启动跟踪，然后立即单击 Stop 按钮。由于 SQL Profiler 用户界面需要工作流，因此在使用脚本之前必须实际启动某个跟踪。在 File 菜单中依次选择 Export、Script Trace Definition。对于 SQL Server 2005 和 SQL Server 2008 版本，将产生以下类似脚本（为了简洁和可读性，此处对脚本进行了适当编辑）：

```
declare @rc int
declare @TraceID int
declare @maxfilesize bigint
set @maxfilesize = 5

exec @rc = sp_trace_create
    @TraceID output,
    0,
    N'InsertFileNameHere',
    @maxfilesize,
    NULL
if (@rc != 0) goto finish

-- Set the events
declare @on bit
set @on = 1
exec sp_trace_setevent @TraceID, 12, 15, @on
exec sp_trace_setevent @TraceID, 12, 16, @on
exec sp_trace_setevent @TraceID, 12, 1, @on
exec sp_trace_setevent @TraceID, 12, 9, @on
exec sp_trace_setevent @TraceID, 12, 17, @on
exec sp_trace_setevent @TraceID, 12, 6, @on
exec sp_trace_setevent @TraceID, 12, 10, @on
exec sp_trace_setevent @TraceID, 12, 14, @on
exec sp_trace_setevent @TraceID, 12, 18, @on
exec sp_trace_setevent @TraceID, 12, 11, @on
```

```
exec sp_trace_setevent @TraceID, 12, 12, @on
exec sp_trace_setevent @TraceID, 12, 13, @on
-- Set the Filters
declare @bigintfilter bigint

set @bigintfilter = 10000
exec sp_trace_setfilter @TraceID, 13, 0, 4, @bigintfilter

-- Set the trace status to start
exec sp_trace_setstatus @TraceID, 1

-- display trace id for future references
select TraceID=@TraceID

finish:
go
```

注意：

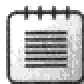
SQL Server 2000 还包含编辑跟踪定义的选项。SQL Server 2000 和 SQL Server 2005 之间的 SQL 跟踪存储过程变化不大（SQL Server 2005 和 SQL Server 2008 之间的 SQL 跟踪存储过程没有任何变化），但产品中添加了几个新事件和列。SQL Server 2000 脚本删除了所有无法向后兼容的脚本。

该脚本非常简单，完整定义了使用文件提供程序的跟踪。需要修改一些占位符的值，大部分都是一些功能性的修改。如果工作的复杂度与 SQL 跟踪存储过程直接相关，我们通常使用 SQL Profiler 的用户界面定义跟踪，然后从编辑脚本开始工作。这样可以获取两方面的优点：操作的简易性和服务器端跟踪的高效性（使用文件提供程序）。

该脚本实现了很多不同的功能，因此我们将详细介绍每个步骤。

（1）该脚本定义了很多过程中需要使用的变量。*@rc* 变量用于从 *sp_trace_create* 中获取返回代码。*@TraceID* 变量用于存储新创建的跟踪的句柄。最后的*@maxfilesize* 变量用于定义每个跟踪文件的最大体积（单位为兆字节）。运行服务器端跟踪时，在主跟踪文件已满的情况下，可以将文件提供程序配置为自动创建*滚动更新文件*（rollover file）。如果您使用的驱动器空间有限，该配置将非常有用，因为您可以将以前的填充文件转移到其他驱动器上。另外，小的文件也便于操作收集的数据子集。最后，滚动更新文件在高负载方案中也有它们的实用性。但在大多数情况下都不需要这些配置，并且对大多数场景而言，将@maxfilesize 的值设为 5 也有些偏小。

（2）该脚本调用 *sp_trace_create* 存储过程，该存储过程将初始化（但不启动）跟踪。此处指定的参数是新创建跟踪的跟踪 ID 的输出参数；选项参数为 0，说明不使用滚动更新文件；使用该脚本之前，需要更改服务器端文件路径的占位符；最大文件大小由*@maxfilesize* 变量定义；NULL 是停止日期（该跟踪仅在被通知时才停止）。注意，*sp_trace_create* 中还有最后一个参数，它允许用户设置滚动更新文件数的最大值。该参数在 *sp_trace_create* 文档中称为*@filecount*，SQL Server 2005 添加了该参数，但使用 Script Trace Definition 选项创建的跟踪定义脚本中没有自动添加该参数。此处没有使用*@filecount* 参数，因为该选项参数被设置为 0，因此没有创建滚动更新文件。但在很多其他案例中，该参数非常有用。注意，因为禁用了更新滚动文件，如果跟踪数据达到了最大文件大小，跟踪将自动停止并关闭。

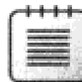


注意：

文件扩展名.trc 将自动添加到输出跟踪文件指定的文件路径中。如果在文件名中使用.trc 扩展名（如 C:\mytrace.trc），那么磁盘上的文件将变成 C:\mytrace.trc.trc。

（3）*sp_trace_setevent* 用于定义跟踪使用的事件/列组合。在本例中，为简单起见，仅使用事件 12（*SQL:BatchCompleted*）。跟踪中使用的每个事件/列组合都需要 1 个 *sp_trace_setevent* 调用。作为预留位，注意@*on* 参数必须占 1 位。因为 SQL Server 2005 及以前版本中的数字文本在默认方式下都隐式转换为整数形式，所以本地@*on* 变量需要强制转换为 SQL Server 2005 及以前版本中存储过程能识别的值。

（4）设置事件后，筛选器就被定义了。在本例中，使用 *and* 逻辑运算符（第三个参数，值为 0）和 *greater than or equal to* 比较运算符（第四个参数，值为 4）筛选列 13（*Duration*）。实际值将作为最后一个参数进行传递。注意，脚本中使用的单位是微秒；SQL 跟踪使用微秒作为其持续时间的单位，而 SQL Profiler 中的默认时间标准是毫秒。为了更改 SQL Profiler 默认设置，依次单击 Tools、Options，然后选中 Show Values In Duration Column In Microseconds 复选框（注意，只有在 SQL Server 2005 和 SQL Server 2008 中才能使用以微秒为单位的持续时间）。



注意：

SQL 跟踪可以同时提供 *and* 和 *or* 逻辑运算符，如果使用多个筛选器，还可以组合使用这两个逻辑运算符。但是，提供相应的方式来表示括号或其他分组结构意味着运算只能按从左向右的顺序计算。这说明，诸如 *A and B or C and D* 的表达式理论上在 SQL 跟踪中计算为(((*A and B*) *or C*) *and D*)。但是，SQL 跟踪根据筛选的列在内部将筛选器分解成不同的组。因此，表达式 *Column1=10 or Column1=20and Column3=15 or Column3=25* 实际计算为 *(Column1=10 or Column1=20) and (Column3=15 or Column3=25)*。这样不仅易于混淆，还使某些条件很难或无法表达。注意，在某些情况下，可能必须分解筛选器标准并创建多个跟踪，以便按需要的方式捕获事件。

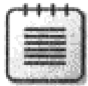
（5）现在已经创建了跟踪，设置了事件和列跟踪，并定义了筛选器。最后要进行的步骤就是启动跟踪。这可通过调用 *sp_trace_setstatus* 完成，同时将第二个参数的值设为 1。

2. 查询服务器端跟踪元数据

正确修改了文件名占位符并在服务器上运行测试脚本后，我收到的跟踪 ID 值为 2。使用跟踪 ID，可以从 *sys.traces* 目录视图中检索各种有关跟踪的元数据，这可以通过以下查询来实现：

```
SELECT
    status,
    path,
    max_size,
    buffer_count,
    buffer_size,
    event_count,
    dropped_event_count
FROM sys.traces
WHERE id = 2;
```

该查询将返回跟踪状态，它的值为 1（启动）或 0（停止）；返回服务器端跟踪文件的路径（如果跟踪使用行集提供程序，则返回 NULL）；返回最大文件大小（同样，如果跟踪使用行集提供程序，则返回 NULL）；返回有关缓冲区用于处理 I/O 的大小信息；返回捕获的事件数；返回删除事件数（在本例中，如果跟踪使用行集提供程序，则返回 NULL）。



注意：

对于从 SQL Server 2000 迁移过来的读者而言，需要注意 *sys.traces* 视图替换了旧 *fn_trace_getinfo* 函数。旧函数仅返回 *sys.traces* 视图返回数据的一小部分，因此当然应该优先使用视图模式。

除了 *sys.traces* 目录视图之外，SQL Server 还附带了很多其他视图和函数，以便获取在服务器运行的跟踪信息。下面介绍这些视图和函数。

***fn_trace_geteventinfo*。**该函数返回跟踪选择的事件和列的数字组合，以表格形式显示。以下 T-SQL 代码将返回跟踪 ID 为 2 的数据：

```
SELECT *
FROM fn_trace_geteventinfo(2);
```

下面是在先前脚本中创建的跟踪上运行该查询的输出结果：

eventid	columnid
12	1
12	6
12	9
12	10
12	11
12	12
12	13
12	14
12	15
12	16
12	17
12	18

***sys.trace_events* 和 *sys.trace_columns*。**跟踪事件和列的数字表示形式本身并不重要。为了能正确查询数据，需要使用它们的文本表示形式。*sys.trace_events* 和 *sys.trace_columns* 不仅分别包含了说明事件和列的文本，还包含了其他信息（如列的数据类型和它们是否可筛选）。将这些视图和前面的查询结合起来使用，与上面的 *fn_trace_geteventinfo* 函数相比，我们可以获取更易于阅读的相同输出版本：

```
SELECT
    e.name AS Event_Name,
    c.name AS Column_Name
FROM fn_trace_geteventinfo(2) ei
```

```
JOIN sys.trace_events e ON ei.eventid = e.trace_event_id
JOIN sys.trace_columns c ON ei.columnid = c.trace_column_id;
```

下面是该查询的输出结果：

Event_Name	Column_Name
SQL:BatchCompleted	*TextData*
SQL:BatchCompleted	*NTUserName*
SQL:BatchCompleted	*ClientProcessID*
SQL:BatchCompleted	*ApplicationName*
SQL:BatchCompleted	*LoginName*
SQL:BatchCompleted	*SPID*
SQL:BatchCompleted	*Duration*
SQL:BatchCompleted	*StartTime*
SQL:BatchCompleted	*EndTime*
SQL:BatchCompleted	*Reads*
SQL:BatchCompleted	*Writes*
SQL:BatchCompleted	*CPU*

***fn_trace_getfi lterinfo*。**为了获取有关跟踪设置了哪些筛选值的信息，可以使用*fn_trace_getfi lterinfo*函数。该函数将返回被筛选的列ID（需要获取更多信息，可以将它连到*sys.trace_columns*视图中）、逻辑运算符、比较运算符和筛选器的值。以下代码展示了该函数的使用示例：

```
SELECT
    columnid,
    logical_operator,
    comparison_operator,
    value
FROM fn_trace_getfilterinfo(2);
```

3. 从服务器端跟踪检索数据

跟踪启动后，下一步显然是读取收集的数据。这由*fn_trace_gettable*函数来完成。该函数有两个参数：从中读取数据的第一个文件名和读取的最大滚动更新文件数（应该存在）。以下T-SQL读取的跟踪文件位于C:\sql_server_internals.trc下：

```
SELECT *
FROM fn_trace_gettable('c:\sql_server_internals.trc', 1);
```

在任何时候都可以读取跟踪文件，甚至当跟踪正在将数据写入跟踪文件时也可以读取跟踪文件。注意，大多数情况下这种方法并不可取，因为它增加了磁盘争用的可能性，因此降低了将事件写入表中的速度，并且增加了阻塞的可能性。但是，如果只是需要偶尔收集数据（例如，为某个不常调用的特定的存储过程模式筛选时），这种方式易于发现截至目前为止已经收集了哪些数据。

由于*fn_trace_gettable*是一个表值函数，它在T-SQL中的使用几乎不受限制。它可用于请求查询，或者可以插入表中来创建索引。在后一种情况下，最好使用*SELECT INTO*语句，以便利用最小记录：

```
SELECT *
```

```
INTO sql_server_internals
FROM fn_trace_gettable('c:\sql_server_internals.trc', 1);
```

数据载入表中后，可以按多种方式操作数据，以进行故障排除或回答问题。

4. 停止和关闭跟踪

跟踪在开始创建时，它的状态为 0，表示停止（或还没有启动）。在任何时候使用 *sp_trace_setstatus* 函数都可以将跟踪恢复为停止状态。为了将跟踪 ID 为 2 的跟踪设置为停止状态，可以使用以下 T-SQL 代码：

```
EXEC sp_trace_setstatus 2, 0;
```

除了跟踪不再收集数据这个明显的优点之外，这样做还有其他优点：跟踪处于停止状态后，可以使用合适的存储过程修改事件/列选择项和筛选器，而不必重新创建跟踪。如果只需要对跟踪稍作调整，该函数非常有用。

如果实际已经完成了跟踪，并且之后不需要继续使用该跟踪，只需将跟踪的状态值设为 2，就可从系统中删除该跟踪定义：

```
EXEC sp_trace_setstatus 2, 2;
```

提示：

SQL Server 服务重新启动时，将自动删除跟踪定义。因此，如果以后需要再次运行相同的跟踪，可以将该跟踪保存为 Profiler 模板，或者保存用于启动该跟踪的脚本。

5. 研究行集提供程序

本节大部分内容讲述了如何使用服务器端的跟踪来处理文件提供程序，有些读者必定会问自己，SQL Server Profiler 如何与行集提供程序进行连接呢？行集提供程序及其接口没有完全公开，但是，由于 Profiler 只调用隐藏的存储过程，因此揭示 Profiler 的工作过程并不难。实际上，可以使用某种意义上的递归过程来实现：使用 Profiler 来跟踪由自身产生的跟踪活动。

指定的跟踪会话无法捕获自身的所有事件（当某些事件发生时，跟踪还没有运行），因此，为了查看 Profiler 如何工作，我们需要建立两个跟踪：配置第一个跟踪来监视 Profiler 的活动；配置第二个跟踪来产生活动供第一个跟踪捕获。首先，打开 SQL Profiler 并使用默认模板创建一个新跟踪。在 Edit Filter 对话框中，删除 *ApplicationName* 上默认的 Not Like 筛选器，并使用 Like 筛选器替换它。Like 筛选器将捕获由任何 SQL Server Profiler 会话产生的所有活动。

启动该跟踪，然后使用默认模板载入另一个跟踪并启动它。第一个跟踪窗口将显示对 various *sp_trace* 存储过程的调用，该调用由 *RPC:Completed* 事件激活。使用行集提供程序时，首先需要注意的不同点是对 *sp_trace_create* 函数的调用：

```
declare @p1 int;
exec sp_trace_create @p1 output,1,NULL,NULL,NULL;
select @p1;
```

该函数的第二个参数选项设置为 1，该值在 *SQL Server 联机丛书*中没有公开，它用于打开行集提供程序。其余参数用于处理输出，使用 NULL 进行填充。

提示：

sp_trace_create 选项参数实际是一个位掩码，可用来同时设置多个选项。为此，只需将需要的每个选项的值相加即可。该函数的选项参数只有 3 个记录的值和 1 个未记录的值，因此可能存在的组合不多，不过仍然需要注意。

其余大部分捕获活动都很常见，对 *sp_trace_setevent*、*sp_trace_setfilter* 和 *sp_trace_setstatus* 进行了正常调用。但是，如果需要查看完整情况，必须停止第二个跟踪（实际产生被捕获跟踪活动的跟踪）。只要第二个跟踪一停止，第一个跟踪就可以捕获以下 *RPC:Completed* 事件：

```
exec sp_executesql N'exec sp_trace_getdata @P1, 0',N'@P1 int',3;
```

在本例中，3 是系统上第二个跟踪的跟踪 ID。给定输入参数设置，*sp_trace_getdata* 存储过程以表格格式返回事件数据，并在跟踪停止时停止返回数据。

遗憾的是，由 *sp_trace_getdata* 产生的表格格式的可识别性非常差，不是标准的跟踪表格式。通过修改前面基于文件的跟踪，我们可以使用以下 T-SQL 代码创建基于行集的跟踪：

```
declare @rc int
declare @TraceID int

exec @rc = sp_trace_create
    @TraceID output,
    1,
    NULL,
    NULL,
    NULL
if (@rc != 0) goto finish

-- Set the events
declare @on bit
set @on = 1
exec sp_trace_setevent @TraceID, 12, 15, @on
exec sp_trace_setevent @TraceID, 12, 16, @on
exec sp_trace_setevent @TraceID, 12, 1, @on
exec sp_trace_setevent @TraceID, 12, 9, @on
exec sp_trace_setevent @TraceID, 12, 17, @on
exec sp_trace_setevent @TraceID, 12, 6, @on
exec sp_trace_setevent @TraceID, 12, 10, @on
exec sp_trace_setevent @TraceID, 12, 14, @on
exec sp_trace_setevent @TraceID, 12, 18, @on
exec sp_trace_setevent @TraceID, 12, 11, @on
exec sp_trace_setevent @TraceID, 12, 12, @on
exec sp_trace_setevent @TraceID, 12, 13, @on

-- Set the Filters
declare @bigintfilter bigint

set @bigintfilter = 10000
```

```
exec sp_trace_setfilter @TraceID, 13, 0, 4, @bigintfilter

-- Set the trace status to start
exec sp_trace_setstatus @TraceID, 1

-- display trace id for future references
select TraceID=@TraceID

exec sp_executesql
    N'exec sp_trace_getdata @P1, 0',
    N'@P1 int',
    @TraceID

finish:
go
```

运行以上代码，然后在其他窗口发布 *WAITFOR DELAY '00:00:10'*，将产生以下输出结果（为简单起见，对输出结果进行了截断和编辑）：

ColumnId	Length	Data
65526	6	0xFEFF63000000
14	16	0xD707050002001D001
65533	31	0x01010000000300000
65532	26	0x0C000100060009000
65531	14	0x0D000004080010270
65526	6	0xFAFF00000000
65526	6	0x0C000E010000
1	48	0x57004100490054004
6	8	0x4100640061006D00
9	4	0xC8130000
10	92	0x4D006900630072006

ColumnId 列中的每个值对应于跟踪数据的列 ID。顾名思义，*Data* 是对应于收集列数据的二进制编码值，*Length* 是 *Data* 列使用的字节数。每个输出行对应于一个事件的一列。SQL Server Profiler 通过调用 *sp_trace_getdata* 将这些事件从行集提供程序中取出，并执行一次数据透视操作，产生易读的输出结果。另外，行集提供程序的效率比不上文件提供程序的效率还表现在，发送太多的行将产生巨大的网络流量。

所幸，如果确实需要使用类似于行集提供程序的行为来满足监视需求，则不需要知道如何操作该数据。SQL Server 2008 在 *Microsoft.SqlServer.Management.Trace* 命名空间中发布了一系列管理类，帮助构建和使用行集提供程序。这些类的使用方法超出了本章的范围，但在 SQL Server TechCenter 中对此有详细介绍，通过阅读其中的内容，读者可以了解如何使用管理类。

2.4　扩展事件

对于需要在 SQL Server 中调试复杂场景的 DBA 和开发人员而言，SQL 跟踪非常有用，但它也有一些关键的限制。首先，SQL 跟踪使用基于列的架构，如果新事件的列与现有输出列的设置不一致，那么

添加新事件将非常困难；其次，大型跟踪对系统性能的影响超出了许多 DBA 的预期范围；最后，SQL 跟踪只是一种跟踪结构，无法将它扩展成通用事件系统可以使用的其他领域。

扩展事件（可简写为 XE、XEvents 或 X/Events，具体取决于阅读的文章或书，后面我们将统一简写为 XE）可以解决所有这些问题。与 SQL 跟踪不同，XE 设计成一种通用的事件系统，不仅用于实现跟踪需求，还适用于其他各种场合（包括引擎的内部和外部）。与 SQL 跟踪事件不同的是，XE 中的事件没有绑定到输出列常用设置中。相反，每个 XE 事件使用自身唯一的架构发布数据，增强了系统的灵活性。另外，XE 还能解答一些和 SQL 跟踪有关的性能问题。在着手设计 XE 系统时就已经充分考虑了性能因素，因此在大多数情况下，XE 事件对整体系统性能的影响是最小的。

由于 XE 的通用功能，它比 SQL 跟踪更大更复杂，了解该系统需要 DBA 理解许多新概念。另外，由于该系统是 SQL Server 2008 的一个新功能，因此还没有 Profiler 或类似工具形式的 UI 支持。学习 XE 的过程非常艰难，因此许多 DBA 都不够主动。但在本章的后面您将看到，XE 是一个功能非常强大的工具，当前的确值得学习了解。在以后的 SQL Server 版本中，您将看到各种形式的 XE 扩展和使用，现在了解它的基础背景，对将来也会大有裨益。

2.4.1 XE 体系结构的组件

XE 系统的主体位于 SQL Server 的总体层中，这种结构类似于 SQL 操作系统（SQLOS）的角色。作为一种通用的事件和跟踪系统，它必须能与各级 SQL Server 主机进程（从查询处理 API 到存储引擎）进行交互。为了完成目标，XE 需要集成许多不同的组件来形成完整的系统。

1. 包

包是一种基本单位，XE 对象都是通过包发布的。每个包都是类型、谓词、操作、映射和事件的集合（您与系统交互时实际使用的是 XE 用户可配置组件）。SQL Server 2008 发行了 4 个包，可以在 *sys.dm_xe_packages* DMV 中查询它们，如下所示：

```
SELECT *
FROM sys.dm_xe_packages;
```

包可以与其他包进行交互，以避免在多个上下文中发布相同的代码。例如，如果一个包实现了可以绑定到其他事件中的某个特定操作，那么其他包中的任何事件都可以使用该包。为了利用这种灵活性，Microsoft 在 SQL Server 2008 中发行了一个名为 *package0* 的包。*package0* 包是一个基础，它包含了与 SQL Server 一起发行的所有其他包需要使用的对象，以及将来可能需要使用的对象。

除了 *package0*，SQL Server 还发行了其他 3 个包。*Sqlos* 包包含的对象旨在帮助用户与 SQL 系统进行交互；而 *sqlserver* 包包含特定于其余 SQL Server 系统的对象；*SecAudit* 包略有不同，它包含的对象供 SQL Audit 使用，SQL Audit 是构建在扩展事件之上的一种审核技术。通过查询 *sys.dm_xe_packages* DMV，可以发现 *SecAudit* 包在 *capabilities_desc* 列中标识为 *private* 属性。这意味着非系统使用者将无法直接使用它所包含的对象。

要查看系统使用的所有系统列表，查询 *sys.dm_xe_objects* DMV：

```
SELECT *
FROM sys.dm_xe_objects;
```

以上 DMV 展示了一些关键列，这对某些热衷于开发该对象的人非常重要。*package_guid* 列使用相

同的GUID进行填充，可以在*sys.dm_xe_packages* DMV的*guid*列查找该GUID。*object_type*列可用于筛选特定类型的对象。正如*sys.dm_xe_packages*一样，*sys.dm_xe_objects*展示了*capabilities_desc*列，如果外部对象无法使用特定的对象，则特定对象的该列通常被设置为*private*属性。另外，还有一个称为*description*的列，它用于提供易读的文本来说明每个对象，目前SQL Server 2008 RTM还在进一步完善该功能，因此许多对象说明还不够完整。

接下来我们将详细介绍*sys.dm_xe_objects*中出现的各种对象类型。

2. 事件

类似于SQL跟踪，SQL Server处理自身任务时，XE展示了不同预期时间激活的多个事件。另外，正如SQL跟踪一样，XE通过调用来检测整个产品中的各种代码路径，这些调用在合适的时候将激活事件。XE新用户几乎可以发现SQL跟踪展示的所有相同事件，并且还可以发现更多事件。SQL跟踪在SQL Server 2008中提供了180个事件，XE提供了254个事件。很多XE事件的级别比SQL跟踪事件的级别更低，因此XE可以提供更多事件。例如，XE包含每次发生页拆分就激活的事件。这样，用户可以在查询级别跟踪拆分，这在先前的SQL Server版本中是无法实现的。

与SQL跟踪展示的对象相比，XE事件最重要的区别在于，每个XE事件还可以展示自身的输出架构。这些架构将展示在*sys.dm_xe_object_columns* DMV中，如下所示，可以查询*sys.dm_xe_object_columns* DMV获取输出列列表：

```
SELECT *
FROM sys.dm_xe_object_columns
WHERE
    object_name = 'page_split';
```

除了列的列表名称和列序号位置之外，以上查询还将返回每个列的数据类型列表。正如XE系统中定义的所有其他对象一样，XE系统也定义了数据类型，并且每个数据类型在*sys.dm_xe_objects* DMV中拥有自身的条目。如果某些列在*column_value*中定义了值，可以将这些列标记为*readonly*(每个*column_type*列）属性，或者可以将它们标记为*data*属性，这表明它们的值将在运行时进行填充。*Readonly*列是元数据，用于存储各种信息（包括激活事件类型的唯一标识符和版本号），以便独立跟踪和使用每个事件的不同架构版本。

事件的CHANNEL属性是与每个事件都有关的少数*Readonly*属性之一，它反映了XE的设计目标之一是为了与Event Tracing for Windows（ETW）系统相统一。SQL Server 2008中的事件可以归类为：管理员事件、分析事件、调试事件和操作事件。以下是有关事件通道的说明。

- *管理员事件*是那些系统管理员使用最为频繁的事件，该通道包括错误报告和Deprecation Announcement之类的事件。
- *分析事件*是那些定期激活的事件（繁忙的系统上每秒可能发生上千次），积累这些事件有助于分析系统性能和运行状况。以锁定获取数据和SQL语句的启动、完成为主题的事件都属于分析事件。
- *调试事件*是那些供DBA和支持工程师使用的事件，以协助诊断和解决有关引擎的问题。该通道包括线程和进程启动和停止时激活的事件、贯穿计划程序生命周期不同时期的事件，以及其他类似主题的事件。
- *操作事件*是那些操作DBA使用最为频繁的事件，用于管理SQL Server服务和数据库。该通道的事件涉及数据库绑定、分离、启动和停止，以及诸如检测数据库页损坏的有关问题。

提供灵活的事件负载系统可以确保任何使用者都可以使用任何公开的事件，只要使用者知道如何读取架构即可。设计事件遵循的目标是：每个事件实例的输出始终包含相同的属性，并且事件的展示顺序与架构定义的顺序完全相同。这种设计目标可以最小化使用者用于处理绑定事件所需的工作量。事件使用者也可以使用该排序更加轻松地忽略那些不需要关注的数据。例如，如果使用者知道给定事件的前 16 字节包含一个与使用者需求无关的标识符，那么可以简单地忽略这些字节，而没必要进行处理。

虽然每个事件的架构在运行之前已经预定了，但每个事件实例的实际大小并没有预定。除了非系统化元素由操作（更多信息请参考本章后面的“操作”部分）进行填充之外，事件负载既可以包含长度固定的数据，也可以包含长度可变的数据。为了降低事件过度使用内存和其他资源的概率，XE 扩展事件系统对长度可变元素的数据设置了 32MB 的硬上限。

您可能已经注意到，与 SQL 跟踪中每个事件可用的列数相比，此处每个事件返回的列的列表要少一些。例如，XE *sql_statement_completed* 事件仅展示 7 个列：*source_database_id*、*object_id*、*object_type*、*cpu*、*duration*、*reads* 和 *writes*。SQL 跟踪用户可能会觉得奇怪，不知道其他常见属性（会话 ID、登录名称）在何处，甚至实际的 SQL 文本也可能用于激活事件。这些功能是通过绑定到“操作”（该内容将在本章的“操作”部分进行讲述）来实现的，而不是使用默认方式下的架构进行填充。这种设计方法进一步增加了 XE 体系结构的灵活性，并使事件本身保持尽可能小，因此提高了整体系统的性能。

与 SQL 跟踪事件一样，默认情况下 XE 事件是禁用的。XE 事件在事件会话（XE 跟踪的等效物，本章后面将介绍它）启动之前，几乎没有开销。与 SQL 跟踪事件一样，可以对 XE 事件进行筛选，并将它们路由到各种后期事件提供程序中进行收集。此处的术语也略有区别，XE 中的筛选器称为*谓词*，而后期事件提供程序称为*目标*。本章将在后面分别介绍“谓词”和“目标”。

3. 类型和映射

在前面一节，我们已经看到了每个事件公开自身的架构，包括列名和类型信息。还提到了在 XE 包中定义架构中包含的各种类型。

XE 包中可以定义两种数据类型：标量类型和映射类型。标量类型是单个值，类似于整数、单个 Unicode 字符或二进制大型对象。而映射类似于大部分面向对象系统中的枚举类型。如果许多事件需要向使用者传递一些易读文本，而不只是机器可读的值集合，它们的值将更大，这时可以使用枚举类型。另外，还可以对很多文本进行预定义（例如，SQL Server 支持等待类型列表），并且可以使用整数在表索引中存储这些文本。在事件激活时，事件可以只存储整数，而不是收集实际文本，因此节省了大量的内存和处理资源。

类似于事件，可以在 *sys.dm_xe_objects* DMV 中查看类型和映射。要查看系统支持的类型和映射，可使用以下查询：

```
SELECT *
FROM sys.dm_xe_objects
WHERE
    object_type IN ('TYPE', 'MAP');
```

虽然类型可以对自身的信息进行适当说明，但映射必须展示它们的相关值，以便使用者在适当的时候可以显示易读文本。可以在 *sys.dm_xe_map_values* 的 DMV 中查看该信息。以下查询将返回 SQL Server 引擎展示的所有等待类型，以及 XE 事件中用于说明等待使用的映射关键字（整数表示类型）：

```
SELECT *
FROM sys.dm_xe_map_values
WHERE
    name = 'wait_types';
```

目前在 SQL Server 2008 RTM 中，使用 *package0* 包几乎可以公开所有类型，但这 4 个包分别包含了许多自身的映射值。因此标量类型（如整数）不需要重复定义，而映射则适合于特定的用途。

还需要注意的是，从结构角度看，其中某些思想已经开始应用于优化类型系统。例如，根据对象的大小选用值传递和引用传递语法。数据流通过系统时，任何 8 字节或更小体积的对象都通过值进行传递，而较大对象则使用特殊的特定于 XE 的 8 字节指针类型通过引用进行传递。

4. 谓词

与 SQL 跟踪事件一样，通过筛选 XE 事件可以仅记录那些有用的事件。例如，您可能希望仅记录在特定数据库中发生的事件，或者记录激活特定会话 ID 的事件。为了提供尽可能灵活的体验目标，XE 谓词分配是基于每个事件的，而不是向所有会话都分配谓词。这与 SQL 跟踪是不同的，SQL 跟踪中的筛选器是基于整个跟踪粒度定义的，因此跟踪中使用的每个事件都必须遵守总体筛选器的设置。但在 XE 中，谓词仅筛选某些事件，而不筛选其他事件（或使用不同的设置标准筛选其他事件），这样做是非常合理的。

从元数据的角度看，谓词在 *sys.dm_xe_objects* 中代表两个不同的对象类型：*pred_compare* 和 *pred_source*。*pred_compare* 对象是比较函数，用于比较特定数据类型的实例，而 *pred_source* 对象是扩展属性，可以在谓词中使用。

我们首先介绍 *pred_compare* 对象。以下查询通过筛选 *pred_compare* 对象类型，通过 *sys.dm_xe_objects* DMV 返回所有可用的“>=”比较函数：

```
SELECT *
FROM sys.dm_xe_objects
WHERE
    object_type = 'pred_compare'
    AND name LIKE 'greater_than_equal%';
```

运行以上查询可以看到，比较函数主要用于一些基本数据类型（整数、浮点数和各种字符串类型）。XE 用户可以显式使用这些函数，但常用运算符中已经重载了创建事件会话的 DDL 函数，因此在大多数情况下都不需要使用该函数。例如，如果使用>=运算符定义基于两个整数的谓词，XE 引擎将自动把该调用映射到 *greater_than_equal_int64* 谓词中。在 DMV 中查看 *greater_than_equal_int64* 谓词。目前，运算符中只有一个谓词还没有重载，即取模运算符，取模运算符用于检测一个输入是否被另一个输入整除。更多信息请参阅本章后面的“扩展事件 DDL 和查询”一节，学习如何使用比较函数。

另一个谓词对象类型（*pred_source*）需要一些背景说明。在 XE 系统中，事件谓词可以筛选两种类型的属性：事件自身展示的列（如 *sql_statement_completed* 事件的 *source_database_id* 列）和 *sys.dm_xe_objects* DMV 中定义成 *pred_source* 的任何外部属性（谓词源）。以下查询将返回可用的谓词源：

```
SELECT *
FROM sys.dm_xe_objects
WHERE
    object_type = 'pred_source';
```

以上每种属性（目前 SQL Server 2008 RTM 提供 28 种属性）都可以绑定到 XE 系统中的任何事件，并且可以随时筛选某个属性（如果事件自身的系统化负载未执行筛选该属性）。因此，可以请求用于获取特定会话 ID 或特定用户名称而激活的事件；如果需要在更深级别进行调试，还可以请求特定线程或工作线程地址上的事件。需要重点注意的是，默认情况下，任何事件都不执行这些谓词源，使用谓词源将强制 XE 引擎在事件处理过程中通过附加步骤获取数据。对大多数谓词而言，获取数据的开销非常小，但是如果使用多个谓词，这种开销将累加。

我们将在本章后面的“事件的生命周期”部分介绍何时和如何激活谓词。

5. 操作

事件激活是事件系统的一个功能，它还可用于执行某些外部代码。例如，激活 DML 触发器事件可用于响应 DML 操作，并执行触发器主体中的代码。除了完成某些工作之外，外部代码还可以检索附加信息，这些信息对于事件来说可能非常重要。例如，触发器可以从系统中的其他表中选择数据。

在 XE 中，一种称为 *action* 的对象承担着这种双重作用。如果将操作绑定到某个事件中，计算出谓词为真之后，操作将被同步调用；操作既可以执行代码，也可以将数据写回事件负载中，因此增加了附加属性。在本章前面的“事件”部分我们已经提到，设计 XE 事件时，总是使它尽可能小，因此每个 XE 事件在默认情况下仅包含少量属性。处理谓词时，使用谓词源可以解决缺少完整属性集的问题，但这仅适用于筛选。使用谓词源不会使谓词源的值和其余的事件数据存储在一起。在谓词源中，最常用的操作是收集指定事件在默认方式下未给出的附加属性。

需要查看可用的操作列表，用户可以查询 *sys.dm_xe_objects*，如下所示：

```
SELECT *
FROM sys.dm_xe_objects
WHERE
    object_type = 'action';
```

目前在 SQL Server 2008 RTM 中，XE 使用了 37 种操作，其中几乎包括了映射到所有谓词源的属性。在实际操作中，您可能需要筛选某个指定的谓词源，并且在事件输出中包含实际值。输出列表还包含各种其他属性，以及少量仅执行代码而不向事件负载返回任何数据的操作。

在计算出谓词之后，且在向引起事件激活的代码返回控制之前（更多信息请参阅本章后面的“事件的生命周期”），操作将立即同步激活事件。这是为了确保在进行同步时，操作能及时收集信息。在服务器状态改变之前，操作的同步激活可能是一个潜在的问题。

由于操作的同步设计，操作将承担一些性能开销。与检索相比，其中大部分操作（如映射可用的谓词）的开销相对廉价，但有些操作的开销也可能非常高。例如，*tsql_stack* 操作是一个非常有趣的适用于调试目的的操作，该操作将返回存储过程中的整个嵌套堆栈和/或导致事件激活的函数调用。虽然返回的信息非常有用，但是，如果不短暂停止执行当前的线程和活动堆栈，*tsql_stack* 操作将无法在引擎中获取该信息。因此与检索当前会话 ID 相比，*tsql_stack* 操作将承担更大的性能开销。

有些操作不返回任何数据，仅执行外部代码。需要查看这些操作列表，可以筛选 *sys.dm_xe_objects* 的 *type_name* 列的“null”返回值，如以下查询：

```
SELECT *
FROM sys.dm_xe_objects
WHERE
```

```
        object_type = 'action'
        and type_name = 'null';
```

注意，本示例中的“null”实际是一个字符串，不同于 SQL NULL；*null* 是 *package0* 中定义的类型名称，并显示在 *type* 类型的对象列表中。有 3 种操作不返回附加数据：其中两个用于执行小型转储，另一个用于激活调试程序的断点。只有产品支持这些操作时，它们才会发挥最佳作用。尤其是调试断点事件，它将停止断点上触发的活动线程，并可能根据断点的触发位置阻塞整个 SQL Server 进程。

与谓词非常相似，操作是基于事件绑定的，而不是基于事件会话级别的。因此，只有在大型会话中激活特定事件时，使用者才可以选择调用操作。某些操作可能不适用于系统中的每个事件，如果用户试图将某些操作绑定到不兼容的事件中，在创建会话时将产生绑定错误信息。

从性能的角度看，除了操作的同步功能之外，还需要重点关注的是，将数据写回事件的操作将增加每个事件实例的体积。这意味着不仅需要花费更长的时间来激活事件和向调用者返回控制（因为操作被同步执行），而且激活事件后，事件也将消耗更多的内存并需要更多的处理时间将数据写入目标中。

与性能有关的问题很常见，解决此类问题的关键是，在使用者的数据需求和服务器的性能需求之间保持总体平衡。记住，操作并不是免费帮助您创建 XE 会话，它对主机服务器的性能将产生微小的影响。

6. 目标

到目前为止我们已经知道，遇到检测代码路径时将激活事件；谓词可以筛选事件，以便收集感兴趣的数据；操作可以向事件负载中添加附加数据。所有这一切发生后，事件数据的最终包需要在某处被收集。事件数据的目的地是一个或多个目标，系统通过目标来使用 XE 事件。

目标是最终的对象类型，该对象类型在 *sys.dm_xe_objects* 中展示元数据。通过运行以下查询可以查看可用的目标列表：

```
SELECT *
FROM sys.dm_xe_objects
WHERE
    object_type = 'target';
```

SQL Server 2008 RTM 使用了 13 个目标（7 个公用目标和 6 个专用目标），仅供 SQL 审核使用。在 7 个公用目标中，3 个在 *capabilities_desc* 列中标记了 *synchronous* 属性。在向引起事件激活的代码返回控制之前，公用目标需要同步收集事件数据（与操作非常相似）。相比之下，其他 5 个事件是异步的，这意味着事件激活之后，目标在收集数据之前要对数据进行缓冲。缓冲能使引起事件激活的代码表现出更好的性能，但它也给进程带来了延迟，因为目标在某些时候可能不收集事件。

XE 目标包含各种类型，它们与 SQL 跟踪提供的 I/O 提供程序既有相似的地方，也有一定的区别。与 SQL 跟踪的文件提供程序相似的是 XE 的 *asynchronous_file_target* 目标，它在写入专用的二进制文件格式之前需要缓冲数据。另一个基于文件的选项是 *etw_classic_sync_target* 目标，它将数据同步写入适合任何启用 ETW 读取程序使用的文件格式中。没有 XE 等效物对应于 SQL 跟踪流行集提供程序。

其余 5 个目标与 SQL 跟踪提供的功能有很大的区别，它们都在内存中存储消费数据，而不是将数据保存到文件中。其中最简单的是 *ring_buffer* 目标，它将数据存储在一个用户可配置大小的环形缓冲区中。当环形缓冲区填充和准备覆盖以前收集的数据时，它将重新回到缓冲区的起点位置。这意味着缓冲区可以使用无限的数据量，而无需使用系统内存，但只有在特定时间才可以获取最新数据。

另一个目标类型是 *synchronous_event_counter* 目标，它将同步计数事件激活的次数。处于同一层次

的是两个 bucketizer 目标（一个是同步的，另一个是异步的），它们基于用户定义的列创建储存桶，并计数每个存储桶中事件发生的次数。例如，用户可以基于会话 ID 创建存储桶，并且目标可以计数激活每个 SPID 的事件数。

最后一个目标类型称为 *pair_matching* 目标，设计该目标是为了帮助查找预期成对发生的事件（由于某些原因一个或另一个事件没有激活）。*pair_matching* 目标的工作方式是：首先异步收集用户定义的开始事件，然后用它们匹配用户定义的结束事件。当成功找到一对匹配的事件时，将删除匹配的事件，仅留下那些没有匹配的事件。例如，考虑存储引擎中的锁定获取数据。为了避免阻塞，我们希望每个锁定都在相对较短的时间内获取和释放。如果出现了阻塞问题，可能是由于锁定的获取和持续时间超出了预期的需要。*pair_matching* 目标通过联合使用锁定获取和锁定释放事件，很容易识别那些已经被锁定但未释放的事件。

目标通常可以与其他目标一起使用，因此可以向单个会话中绑定多个目标，而无需创建多个会话来收集相同的数据。例如，用户可以创建多个存储桶目标，以便根据不同的存储桶标准来同步保留元数据计数，同时在某个文件中记录未聚合的数据以便稍后计算。

与 SQL 跟踪提供程序一样，在一定时间内进入系统的数据条目超出了系统的处理能力时，必须采用某些操作进行处理。使用同步目标时，方法非常简单：在目标返回控制之前，调用代码一直处于等待状态，在事件数据完全使用之前，目标一直处于等待状态。然而，对于异步目标而言，可以通过一些配置选项来指示如何处理该情况。

事件数据缓存区开始填满时，引擎可以采用 3 种可能的操作，具体情况取决于用户如何配置会话。以下是 3 种可能的操作。

- **阻塞、等待缓冲区空间变为可用（无事件丢失）**。这种行为与 SQL 跟踪的文件提供程序相同。
- **删除等待事件（允许单个事件丢失）**。在这种情况下，系统在等待释放更多缓存区空间时仅释放单个事件。这是默认模式。
- **删除满缓冲区（允许多个事件丢失）**。每个缓冲区都可能包含大量事件，事件的丢失数取决于事件的大小和缓冲区的大小（我们将对此进行简要介绍）。

以上各选项对总体系统性能的影响呈递减次序，事件等待缓冲区释放空间时可能丢失的事件数呈递增次序。选择的选项能反映出可接受的数据丢失量，同时需要注意的是，阻塞的出现会对选项的使用产生过多的限制。自由使用谓词、仔细关注绑定到每个事件的操作数和关注其他配置选项都能协助用户避免产生缓冲区填充问题，并协助用户确定是否选用这些选项。

除了能够指定出现缓冲区填满时如何进行处理之外，用户还可以指定分配多少内存、内存如何在 CPU 或 NUMA 节点边界之间进行分配，以及多长时间清除一次缓冲区。

在默认方式下，每个 XE 会话（将在下一节讲述）都将创建一个中心缓冲集合（最多使用 4MB 内存）。中心组缓冲区集合始终包含 3 个缓冲区，每个缓冲区使用的空间占最大指定内存量的 1/3。用户可以重写这些默认设置，为每个 CPU 创建一个缓冲区集合或为每个 NUMA 节点创建一个缓冲区集合，以及增加或减少每个缓冲区集合使用的内存量。另外，用户还可以指定允许激活大于最大分配缓冲区内存的事件。在这种情况下，这些事件将存储在特殊的大型内存缓冲区中。

另一个默认选项是，每隔 30 秒或缓冲区填满时将清除缓冲区。通过用户和最大延迟设置可以重写该选项。这将导致缓冲区在特定的时间间隔（指定为若干秒）和它们填满时将检测缓冲区。

需要重点注意的是，这些设置的应用不是基于每个目标的，而是针对绑定到会话的所有目标。我们将在下一节介绍这些设置如何工作。

2.4.2 事件会话

现在，我们已经学习了构成核心 XE 体系结构的每个元素。在运行时将这些元素汇集成一个整体单元就形成了会话。SQL 跟踪语法中的跟踪是 XE 中会话的等效物。会话用于说明用户需要收集的事件；谓词用于解决应该筛选哪些事件；需要激活的操作与事件一起协同工作；最后，目标在循环的结尾用于数据收集。

在足够的服务器级别权限下，用户可以创建许多会话。与 SQL 跟踪一样，大多数会话相互独立。连接多个会话的主要线程是中心位图，它用于指示是否启用或禁用指定事件。在许多会话中可以同步启用事件，但全局位图用于避免在运行时必须检测所有会话。甚至有些会话之间是完全独立的，每个会话使用自身的内存，并且拥有自身的定义对象集合。

1. 会话作用域的目录元数据

除了在会话级别定义一组事件、谓词、操作和目标之外，各种 XE 配置选项的设置也是在会话级别完成的。与定义 XE 的基础对象一样，SQL Server 元数据存储库中已经添加了很多视图，以支持有关会话的元数据查询。

sys.server_event_sessions 目录视图是有关 XE 会话信息的中心元数据存储。该视图展示了 SQL Server 实例上定义的每个会话的一行。如同 SQL 跟踪中的跟踪一样，可以随意启动和停止 XE 会话。但不同于跟踪的是，服务重新启动后，XE 会话是永久存在的。因此除非显式删除某个会话，否则，在重新启动服务前后查询视图都将显示相同的结果。SQL Server 实例启动时，可以将会话配置为自动启动；通过 *startup_state* 视图列可以查看该设置。

与中心 *sys.server_event_sessions* 视图在一起的还有很多其他视图，用于详细说明如何配置会话。*sys.server_event_session_events* 视图展示了绑定到每个会话中的每个事件的一行，并包含一个谓词列。如果已经设置了一个谓词列，该列将包含用于筛选事件的谓词定义。还存在一些与操作和目标类似的视图，即 *sys.server_event_session_actions* 和 *sys.server_event_session_targets* 视图。最后一个视图 *sys.server_event_session_fields* 包含指定事件或目标的有关自定义设置信息。例如，用户可以将环形缓冲区目标的内存使用量设置为指定数量；如果使用了目标，该视图将显示内存设置。

2. 会话作用域的配置选项

在本章前面的“目标”部分我们已经提到，某个会话的一些设置是全局性的，因此它将影响组成该会话的对象的运行时行为。

第一个会话作用域的选项设置包括我们已经讨论的主题：从内存和延迟角度考虑，确定如何配置异步目标缓冲区。这些设置将影响一个名为 *dispatcher* 的进程，该进程负责定期从缓冲区收集数据，并将收集的数据发送给绑定到会话的每个异步目标中。激活调度程序的频率取决于如何配置内存和延迟设置。如果延迟值指定为 *infinite*，调度程序将不收集数据，除非缓冲区满了。否则，调度程序将按设置确定的时间间隔收集数据，时间间隔通常为每秒 1 次。

可以使用 *sys.dm_xe_sessions* DMV 来监视是否存在调度异步缓冲区问题。该 DMV 展示了已经启动的每个 XE 会话的一行，并展示了一些列，可以让用户了解如何处理缓冲区。以下是其中最重要的列。

- *regular_buffer_size* 和 *total_regular_buffers*。这些列展示了已创建的缓冲区数（基于最大内存和内存分区设置）和每个缓冲区的大小。通过了解这些数字并估计每个事件的近似大小，可以知

道在满缓冲区时可能丢失多少事件，以及是否应该使用允许多个事件丢失选项。

- *dropped_event_count* 和 *dropped_buffer_count*。这些列展示了事件数和/或由于没有足够的空闲缓冲区空间来容纳传入的事件数据而删除的缓冲区数。
- *blocked_event_fire_time*。如果使用了无事件丢失选项，该列展示了阻塞发生的时间量。

另一个可以启动的会话作用域选项称为*因果跟踪*。当相同线程上的任务之间存在主—从关系时，或者一个线程在另一个线程上引起活动时，该选项允许用户使用 SQL Server 引擎功能来帮助关联事件。在引擎代码中，每个定义 GUID（即通常所说的活动 ID）的任务将跟踪这些关系。调用任务时，该 ID 将按顺序传递；当调用随后的任务时，该 ID 将继续传到堆栈中。如果活动需要传递到其他线程中，该 ID 将被传递到一个称为*传送块*的结构中，然后按相同的逻辑继续执行后面的步骤。

通过两个 XE 操作公开这些标识符：*package0.attach_activity_id* 和 *package0. attach_activity_id_xfer*。但是，这些操作无法通过用户创建会话来绑定到某个事件中。相反，用户必须在会话级别启用因果跟踪选项，该选项将自动向该会话定义的每个事件中绑定操作。操作启用后，每个事件的负载中都将添加活动 ID 和活动传送 ID。

3. 事件的生命周期

从本质上说，激活事件就意味着在 SQL Server 代码中遇到了一个潜在的“兴趣”点。代码中的该兴趣点将调用处理事件逻辑的中心函数，并且将产生不同的情况。本节将介绍事件的生命周期，如图 2-13 所示。

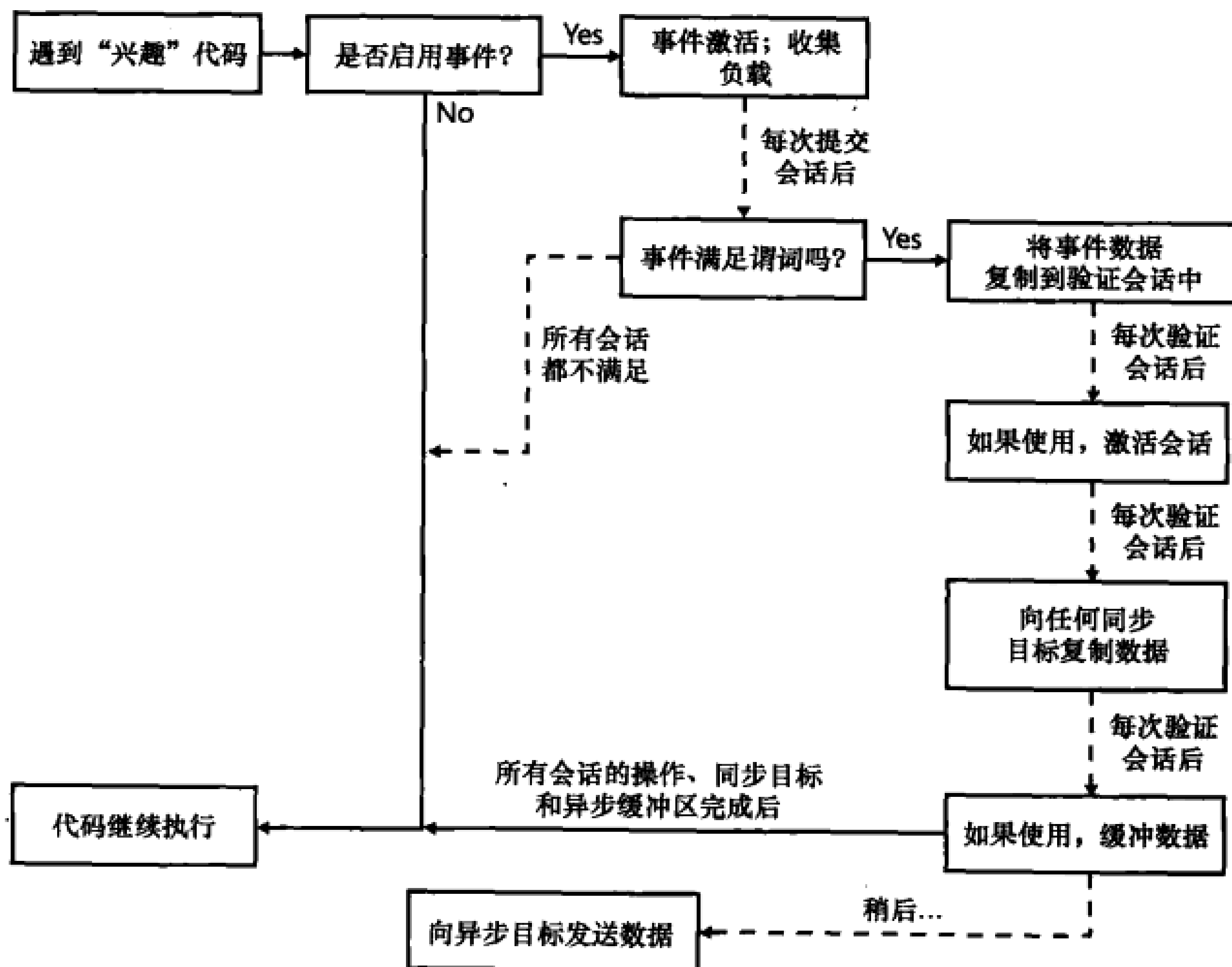


图 2-13 扩展事件的生命周期

至少在一个会话中定义了一个事件后，将设置全局位图来指示：在遇到引用该位图的代码时是否应该激活事件。无论是否启用事件，代码始终必须执行该检测；对于没有启用的事件，检测将进入单个代码分支，几乎没有任何系统开销。如果没有启用事件，进程将结束，代码将继续其正常的执行路径。只有在一个或多个会话中启用某个事件时，特定于事件的代码才会继续处理。

此时如果启用了事件，该事件将被激活，系统将收集与事件架构有关的所有数据元素并将其打包。接下来，XE 引擎将查找每个启用该事件的会话，并同步执行以下步骤。

（1）检查该事件是否满足在会话中为该事件定义的谓词。如果不满足，引擎将转向下一个会话，无需进一步操作。

（2）如果谓词满足条件，引擎将把事件数据复制到会话的上下文中。然后，将激活在会话中为事件定义的任何操作，随后将事件数据复制到所有同步目标中。

（3）最后，对于会话中使用所有异步目标的事件而言，引擎将根据需要缓冲事件数据。

每个会话执行以上步骤后，代码将继续执行。需要重点强调的是，在执行代码块的同时，以上这些步骤都是同步发生的。虽然所有的步骤和整个系统在设计时都考虑了性能问题，如果用户定义了过多的会话、过多的操作或同步目标，以及特别活跃的事件（如分析通道中的事件），用户仍然会遇到问题。应该注意避免过度使用同步功能，以免出现运行时阻塞问题。

事件数据被缓冲后（取决于事件的延迟和会话的内存设置），事件数据将再次传递给所有异步目标。此时，事件数据将从缓冲区中删除，以便为新进入的数据释放空间。

如果目标需要花费过长的时间使用数据，将会引起等待问题，为了帮助跟踪这些问题，可以使用 *sys.dm_xe_session_targets* DMV。该 DMV 公开由每个活动 XE 会话定义的每个目标的一行，还公开名为 *execution_duration_ms* 的列。该列用于指示目标处理最新事件或缓存（取决于该目标）所需的事件量。如果看到该数据在逐渐增加，说明 SQL Server 代码路径中必然会出现等待问题。

2.4.3 扩展事件 DDL 和查询

为了全面了解 XE，我们将快速学习创建 DDL，并理解创建实际会话时，所有对象是如何使用的。另外，我们还给出一个示例，介绍如何查询一些由 XE 会话收集的数据。

1. 创建事件会话

XE 的主要 DDL 挂钩是 *CREATE EVENT SESSION* 语句。该语句允许用户创建会话和映射各种 XE 对象。另外，还有 *ALTER EVENT SESSION* 语句，允许用户修改已创建的会话。为了修改某个现有会话，该会话不能处于活动状态。

以下 T-SQL 语句将创建一个会话，并展示如何配置本章介绍的所有 XE 功能和选项：

```
CREATE EVENT SESSION [statement_completed]
ON SERVER
ADD EVENT
     sqlserver.sp_statement_completed,
ADD EVENT
     sqlserver.sql_statement_completed
      (
         ACTION
         (
              sqlserver.sql_text
```

```
        )
        WHERE
        (
            sqlserver.session_id = 53
        )
    )
ADD TARGET
    package0.ring_buffer
    (
        SET
            max_memory=4096
    )
WITH
(
    MAX_MEMORY = 4096KB,
    EVENT_RETENTION_MODE = ALLOW_SINGLE_EVENT_LOSS,
    MAX_DISPATCH_LATENCY = 1 SECONDS,
    MEMORY_PARTITION_MODE = NONE,
    TRACK_CAUSALITY = OFF,
    STARTUP_STATE = OFF
);
```

该会话称为 *statement_completed*，绑定了两个事件 *sp_statement_completed* 和 *sql_statement_completed*，并通过 *sqlserver* 包公开这两个事件。*sp_statement_completed* 事件没有定义操作或谓词，因此，每次该事件激活范围内的实例时，其默认属性设置将发布到会话的目标中。另一方面，*sql_statement_completed* 事件包含配置（WHERE 选项）谓词，因此，它仅为 ID 为 53 的会话发布属性设置。注意，该谓词使用等于（=）运算符比较两个整数，而不是调用 *pred_compare* 函数。以上代码中定义了标准比较运算符；目前需要直接调用函数的唯一原因是使用了 *divides_by_uint64* 函数，该函数用于确定一个整数是否被另一个整数整除（使用计数器谓词源时有用）。还需要注意的是，WHERE 语句支持 AND、OR 和括号（如果需要，您可以创建许多不同条件的复杂谓词组合）。

为 ID 为 53 的会话激活 *sql_statement_completed* 事件时，事件会话将调用 *sql_text* 操作。该操作用于收集引起事件激活的 SQL 语句文本，并将它添加到事件的数据中。完成事件数据的收集后，事件数据将被推入 *ring_buffer* 目标中，*ring_buffer* 目标可以使用的最大内存是 4 096KB。

另外，我们还配置了一些会话级别的选项。该会话的异步缓冲区无法使用大于 4 096KB 的内存，如果缓冲区填满，我们将允许删除事件。但这不可能发生，因为我们已经将调度程序配置为每秒钟清空一次缓冲区。CPU 上的内存没有分区（因此最终我们将有 3 个缓冲区），并且我们没有使用因果跟踪。最后，会话创建后，它仅作为元数据存在；直到当我们发布以下语句时才启动这个会话：

```
ALTER EVENT SESSION [statement_completed]
ON SERVER
STATE=START;
```

2. 查询事件数据

会话启动后，环形缓冲区目标随着新事件（假设存在一些新事件）的出现每秒更新一次。每个驻内存目标（环形缓冲区、bucketizer 和事件计数目标）都在 *sys.dm_xe_session_targets* DMV 的 *target_data* 中公开 XML 格式的数据。假设该数据是 XML 格式，很多不熟悉 XQuery 的 DBA 可能希望尝试它；假设使用 XE 可以检索大量信息，我们极力推荐您学习如何查询数据。

使用表格格式的 XML 需要了解当前存在哪些节点。在环形缓冲区目标中，称为 *RingBufferTarget* 的根节点为每个激活的事件包含 1 个事件节点。事件节点为事件数据中包含的每个属性存放 1 个数据节点，并为绑定到该事件的操作存放 1 个“操作”节点。这些数据和操作节点都包含 3 个节点：一个节点称为 *type*，用于指示数据类型；一个节点称为 *value*，大多数情况下用于存放值；还有一个节点称为 *text*，用于存放较长的文本值。

解释如何查询每个可能的事件和目标的内容超出了本书的范围，但以下代码提供了一个基于 *statement_completed* 会话的快速示例查询；使用环形缓冲区目标时，可以把该查询作为基础，从中激活基于其他事件和操作的查询。

```
SELECT
    theNodes.event_data.value('(data/value)[1]', 'bigint') AS source_database_id,
    theNodes.event_data.value('(data/value)[2]', 'bigint') AS object_id,
    theNodes.event_data.value('(data/value)[3]', 'bigint') AS object_type,
    theNodes.event_data.value('(data/value)[4]', 'bigint') AS cpu,
    theNodes.event_data.value('(data/value)[5]', 'bigint') AS duration,
    theNodes.event_data.value('(data/value)[6]', 'bigint') AS reads,
    theNodes.event_data.value('(data/value)[7]', 'bigint') AS writes,
    theNodes.event_data.value('(action/value)[1]', 'nvarchar(max)') AS sql_text
FROM
(
    SELECT
        CONVERT(XML, st.target_data) AS ring_buffer
    FROM sys.dm_xe_sessions s
    JOIN sys.dm_xe_session_targets st ON
        s.address = st.event_session_address
    WHERE
        s.name = 'statement_completed'
) AS theData
CROSS APPLY theData.ring_buffer.nodes('//RingBufferTarget/event')theNodes (event_data);
```

以上查询将环形缓冲区数据转换成 XML 实例，然后使用 *nodes* XML 函数为每个找到的事件节点创建一行。最后，它使用事件节点中各种数据元素的序号位置来向输出列映射数据。当然，更高级的会话需要更高级的 XQuery 来确定每个事件的类型；如果会话中涉及的事件有不同的架构，可能还需要实现某些案例逻辑。所幸，本例中的两个会话不需要这样做。获取查询数据后，由于该数据是标准的表格数据，因此可以将它聚集、连接或插入表格中，或者使用它完成任何其他需要的操作。

另外，也可以通过在 T-SQL 中使用 *sys.fn_xe_file_target_read_file* 表值函数从异步文件目标中读取数据。该函数将返回每个事件的一行，但您仍然需要获取合适的 XML 格式；类似于环形缓冲区目标中的数据，在 *event_data* 列展示的事件的数据也是 XML 格式。最后，我们可以期待某个用户界面为我们分担一些 XML 负担。但与 SQL 跟踪一样，当需要进行复杂的分析时，即使最强大的用户界面也有不足之处。因此，此处的 XML 是为那些希望成为 XE 高级用户的 DBA 设计的。

3. 停止和删除事件会话

完成从事件会话中读取数据后，可以使用以下代码停止该会话：

```
ALTER EVENT SESSION [statement_completed]
ON SERVER
STATE=STOP;
```

停止事件会话不能删除元数据；为了从服务器完全删除会话，必须使用以下语句：

```
ALTER EVENT SESSION [statement_completed]
ON SERVER;
```

2.5 小结

SQL Server 包含许多事件系统：从简单的事件系统（如触发器和事件通知）到复杂的事件系统（如XE）。这些系统都是为了在数据库引擎中发生特定操作时，通过启用任意代码或收集数据来帮助用户和SQL Server 本身更好地工作。本章我们介绍了更改跟踪用于支持同步化应用程序所使用的各种隐藏和内部对象、无处不在的 SQL 跟踪基础结构的内部工作机制、复杂的 XE 架构，以及 SQL Server 中事件的发展。SQL Server 中事件的功能极其强大，我们希望本章能为您提供丰富的有关这些系统的内部知识，让您足以了解如何在日常操作中更好地使用事件功能。

第 3 章
数据库和数据库文件

Kalen Delaney

简单地说，Microsoft SQL Server 数据库是保存和操作数据对象的集合。虽然典型的 SQL Server 实例只有少量数据库，但是单个实例包括几十个数据库的情况也不罕见。每个 SQL Server 实例的技术限制是 32 767 个数据库。但实际上，可能永远都不会达到这个限制。

更详细点说，可以认为 SQL Server 数据库具有以下属性和功能。

- 是许多对象（如表、视图、存储过程和约束）的集合。技术限制是 $2^{31}-1$（超过 20 亿）个对象，对象的数量通常在几百和几万之间。
- 由一个 SQL Server 登录账户所有。
- 维护自己的用户账户、角色、架构和安全集合。
- 有自己的系统表集合，用于保存数据库目录。
- 是恢复的主要单元，而且维护其中对象之间的逻辑一致性（例如，主键和外键的关系总是会引用相同数据库中的其他表，而不是其他数据库中的表）。
- 有自己的事务日志，并管理自己的事务。
- 能够跨多个磁盘驱动器和操作系统文件。
- 大小在 2MB 和技术限制 524 272 太字节之间。
- 能自动或手动增大和收缩。
- 其对象可以和来自同一 SQL Server 实例或链接服务器上其他数据库中的对象进行联合查询。
- 可以启用或禁用它的某些属性（例如，可以把数据库设为只读或设为复制中被发布数据的源）。

SQL Server 数据库：

- 和整个 SQL Server 实例不同；
- 不是单个 SQL Server 表；
- 不是特定的操作系统文件。

虽然数据库与操作系统文件不同，但它总是在两个或更多的操作系统文件中。这些文件称为 SQL Server *数据库文件*，在创建数据库时用 *CREATE DATABASE* 命令指定，或是创建后用 *ALTER DATABASE* 命令指定。

3.1 系统数据库

新的 SQL Server 2008 安装总是包括 4 个数据库：*master*、*model*、*tempdb* 和 *msdb*。它还包含第 5 个“隐藏”数据库，用能够列出所有数据库的任何正常 SQL 命令都看不到它。这个数据库称为*资源数据库*，它的实际名称是 *mssqlsystemresource*。

3.1.1 master

master 数据库包括系统表，系统表记录整个服务器安装和在这之后创建的所有其他数据库。虽然每个数据库都有维护数据库中对象信息的系统目录集，但是 *master* 数据库的系统目录保存了关于磁盘空间、文件分配和使用、在系统范围配置设置、终点、登录账户、当前实例上的数据库和其他运行 SQL Server 的服务器（对于分布式操作来说）。

master 数据库对系统来说很关键，因此总是要保存它的当前备份副本。像创建另一个数据库、改变配置值、修改登录账户这样的操作都会修改 *master*，所以总是应该在完成这些操作之后备份 *master*。

3.1.2 model

model 数据库只是模板数据库。每次创建新数据库时，SQL Server 都会生成 *model* 的副本作为新数据库的基础。如果想让每个新数据库开始就具有某些对象或权限，可以把它们放在 *model* 中，这样所有新数据库就会继承它们。也可以通过 *ALTER DATABASE* 命令改变 *model* 数据库的大多数属性，之后创建的任何新数据库都可以使用这些属性值。

3.1.3 tempdb

tempdb 数据库被当成工作空间使用，这在 SQL Server 数据库中是很特别的，因为每次重新启动 SQL Server 时，都会重建而不是恢复它。由用户明确创建的临时表、保存由 SQL Server 在查询处理和排序过程中在内部创建的中间结果的工作表、维护在快照隔离和某些其他操作中使用的行版本，以及显现静态游标和键集游标的键都会使用 *tempdb* 数据库。因为 *tempdb* 数据库是会重建的，所以下次启动 SQL Server 实例时，在数据库中创建的任何对象或权限都会丢失。另一种方法是在 *model* 数据库中创建对象，从 *model* 数据库复制出 *tempdb*（记住，在 *model* 数据库中创建的任何对象都会被添加到之后创建的任何新数据库中。如果想让对象只存在于 *tempdb* 中，可以创建启动存储过程，在每次 SQL Server 实例启动时创建对象）。

tempdb 数据库的大小和配置对 SQL Server 的优化运行和性能很关键，所以我会在本章后面的"*tempdb* 数据库"一节中更详细地讨论。

3.1.4 资源数据库

如前所述，*mssqlsystemresource* 数据库是个隐藏数据库，通常称为*资源数据库*。可执行系统对象（如系统存储过程和功能）都保存在这里。Microsoft 创建这个数据库是为了快速和安全地升级。如果没人能访问此数据库，也就没人能改变它。简单地用新资源数据库更换旧的，就可以升级到新的、包括新系统对象的服务包。记住，不能用查看数据库的任何正常方法查看此数据库，例如从 *sys.databases* 选择或者执行 *sp_helpdb*。它不会出现在 SQL Server Management Studio"对象资源管理器"面板中的系统数据库树中，也不会出现在可以从查询窗口访问的数据库下拉列表中，但这个数据库仍然需要磁盘空间。

可以用 Microsoft Windows 资源管理器在默认的 *binn* 目录中看到这些文件。我的数据目录是 C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL\Binn，能看到名为 mssqlsystemresource.mdf 文件，大小为 60.2MB，还有大小为 0.5MB 的 mssqlsystemresource.ldf 文件。这两个文件的创建和修改时间是固定当前编译代码的日期，这应该与运行 SELECT @@version 时看到的日期相同。对于 SQL Server 2008 的 RTM build 来说，它是 10.0.1600.22。

如果您需要迫切"看到"*mssqlsystemresource* 的内容，有几种方法。如果只想查看里面有什么，那最

容易的方法就是停止 SQL Server、为资源数据库生成这两个文件的副本、重新启动 SQL Server，然后把被复制的文件附加在上面并创建一个具有新名称的数据库。通过使用 Management Studio 中的“对象资源管理器”或者 *CREATE DATABASE FOR ATTACH* 语法来创建副本数据库，如下所示：

```
CREATE DATABASE resource_COPY
ON (NAME = data, FILENAME = 'C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\binn
        \mssqlsystemresource_COPY.mdf'),
    (NAME = log, FILENAME =
   'C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\binn\mssqlsystemresource_COPY.ldf')
   FOR ATTACH;
```

SQL Server 对待这个新 *resource_COPY* 数据库与对待其他用户数据库一样，并不会特殊对待其中的对象。如果想改变资源数据库中的任何内容（如提供的系统存储过程的文本），在 *resource_COPY* 中改变它并不会影响实例上的任何其他内容。但是如果以单一用户模式启动 SQL Server 实例，就可以与 SQL Server 进行单一连接，而且该连接可以使用 *mssqlsystemresource* 数据库。以单一用户模式启动实例和将数据库设置为单一用户模式不同。关于如何以单一用户模式启动 SQL Server 的详细情况，请看 *SQL Server 联机丛书*关于 sqlservr.exe 应用程序的部分。我在第 6 章讨论数据库对象时，会讨论一些资源数据库中的对象。

3.1.5 msdb

msdb 数据库由 SQL Server Agent（SQL 服务代理）服务和其他伴随服务使用，这些服务完成预定的活动（如备份和复制任务），为 SQL Server 提供排队和可靠消息传送的 Service Broker 也使用 *msdb* 数据库。除了备份，*msdb* 中的对象还支持作业、警告、日志传送、策略、数据库邮件和损坏页面的恢复。如果不能在这个数据库上频繁地进行这些活动，基本上可以忽略 *msdb*（但是可能要查看备份历史和保存在那里的其他信息）。用 Management Studio 中的“对象资源管理器”可以访问 *msdb* 中的所有信息，所以通常不需要直接访问这个数据库中的表格。可以把 *msdb* 表想成是系统表的另一种形式：就像从来不直接修改系统表一样，也不应该从 *msdb* 的表中添加或删除数据，除非很清楚自己在做什么或者有 SQL Server 技术支持工程师的指导。在 SQL Server 2005 之前，可以删除 *msdb* 数据库，而且删除后还可使用 SQL Server 实例，但是不能维护任何备份历史，也不能定义任务、警告、作业或设置复制。有一个可以用来删除 *msdb* 数据库的未公开跟踪标志，但由于默认的 *msdb* 数据库非常小，即便您认为可能永远都不再需要它，我也推荐不要去管它。

3.2 样例数据库

在 SQL Server 2005 之前，安装程序会自动安装样例数据库，这样就有一些实际的数据来探索 SQL Server 的功能。SQL Server 2008 不会自动安装任何样例数据库，这是 Microsoft 为加强安全所做的努力之一。但是，有几个样例数据库广泛可用。

3.2.1 AdventureWorks

AdventureWorks 实际上包括样例数据库族，是 Microsoft 用户培训组创建的，“真实”数据库可能与此相似。这个族包括：*AdventureWorks2008*、*AdventureWorksDW2008* 和 *AdventureWorksLT2008*，以及与它们相对应的、为 SQL Server 2005 所创建的族——*AdventureWorks*、*AdventureWorksDW* 和 *AdventureWorksLT*。

可以从 Microsoft 的 codeplex 站点 http://www.codeplex.com/SqlServerSamples 下载这些数据库。这个数据库是为了展示 SQL Server 的功能而设计的，包括按不同的架构组织对象。这些数据库都基于虚构的 Adventure Works Cycle 公司的数据。*AdventureWorks* 和 *AdventureWorks2008* 数据库是为支持 OLTP 应用程序而设计的，*AdventureWorksDW* 和 *AdventureWorksDW2008* 是为支持 SQL Server 的业务智能功能而设计的，而且基于完全不同的数据库架构。两个设计都是高度标准化的。虽然标准化的数据和许多独立的架构可能会与真实的产品数据库设计密切对应，但是它们也会让编写、测试简单的查询和学习基本的 SQL 变得十分困难。

数据库设计不是本书的重点，所以大部分例子都使用我创建的简单表。如果需要很多行数据的话，我有时会把数据从一个或多个 *AdventureWorks2008* 表复制到我自己的表中。最好熟悉 *AdventureWorks* 数据库族的设计，因为 *SQL Server 联机丛书*和 Microsoft 网站（http://www.microsoft.com/sqlserver/2008/en/us/white-papers.aspx）上发表的白皮书中的很多例子都使用这些数据库中的数据。

注意，也可以安装 *AdventureWorksLT2008*（或 *AdventureWorksLT*）数据库，它是 *AdventureWorks* OLTP 数据库的高度简化又不太标准化的版本，它集中于具有单一架构的单一销售情况。

3.2.2 pubs

pubs 数据库是样例数据库，在 SQL Server 的早期版本中大量使用。许多带 SQL Server 例子的旧书都会假设您有这个数据库，因为它在 SQL Server 2005 之前的 SQL Server 版本上都是自动安装的。可以从 Microsoft 的网站下载用于编译此数据库的脚本，我已经在本书的补充内容中包括了这个脚本，网址为 http://www.SQLServerInternals.com/companion。

pubs 数据库公认很简单，这是它的特点，而不是缺点。它提供好例子，但又没有舍本逐末。在测试 SQL Server 功能时，不用担心在 *pubs* 数据库中做改动，因为可以运行所提供的脚本重新编译 *pubs* 数据库。在查询窗口中，打开名为 *Instpubs.sql* 的文件并执行它。确保当前没有与 *pubs* 的连接，因为在创建新数据库之前，会删除当前的 *pubs* 数据库。

3.2.3 Northwind

Northwind 数据库最初是为使用 Microsoft Office Access 开发的样例数据库。处理应用程序编程的大部分 SQL Server 2005 之前的文档使用 *Northwind*。*Northwind* 接近 4MB，比 *pubs* 稍大，也更复杂一些。就像 *pubs* 那样，可以从 Microsoft 站点下载脚本来编译它，或者使用随书资料提供的脚本，文件名为 Instnwnd.sql。而且，本书的一些样例脚本使用 *Northwind* 的修改版本，名为 *Northwind2*。

3.3 数据库文件

数据库文件只不过是操作系统文件（除了数据库文件，SQL Server 还有*备份设备*，这是映射到操作系统文件或物理设备如磁带驱动器的逻辑设备，在本章中，我不会讨论用于存储备份的文件）。数据库至少跨越两个、也可能多个数据库文件，这些文件都是在创建或修改数据库时指定的。每个数据库必须至少跨越两个文件，一个是数据文件（再加上索引和分配页面），另一个是事务日志文件。

SQL Server 2008 允许有以下 3 种类型的数据库文件。

- **主数据文件**。每个数据库都有一个主数据文件，除了存储数据，它还记录数据库中的所有文件。按照约定，主数据文件具有.mdf 扩展名。

- **辅助数据文件**。数据库可以没有或者有多个辅助数据文件。按照惯例，辅助数据文件具有.ndf扩展名。
- **日志文件**。每个数据库至少有一个日志文件，它包括恢复数据库中所有事务的必要信息。按照惯例，日志文件具有扩展名.ldf。

而且，SQL Server 2008 数据库还可以具有文件流数据文件和全文本数据文件。文件流数据文件将在本章后面的“文件流文件组”一节和第 7 章中讨论。全文本数据文件是和其他数据库文件分别创建和管理的，超出了本书的范围。

每个数据库文件有 5 个属性，可以在创建文件时指定：逻辑文件名、物理文件名、初始大小、最大大小和增长增量（文件流数据文件只有逻辑和物理名称属性）。这些属性的值和每个文件的其他信息都可以通过元数据视图 *sys.database_files* 查看，它包含数据库所使用的每个文件的一行。表 3-1 中列出了 *sys.database_files* 中显示的大部分列。这里没提到的列包含处理与特定文件相关的、处理事务日志备份的信息，事务日志将在第 4 章讨论。

表 3-1 ***sys.database_files* 目录视图**

列	说 明
fileid	文件标识号（对于每个数据库是唯一的）
file_guid	文件的 GUID NULL＝从 SQL Server 的早期版本升级的数据库
type	文件类型 0＝行（包括升级到 SQL Server 2008 或者在 SQL Server 2008 中创建的全文本目录） 1＝日志 2＝文件流 3＝为将来保留 4＝全文本（包括 SQL Server 2008 以前版本的全文本目录）
type_desc	文件类型描述 行 日志 文件流 全文本
data_space_id	包括该文件的数据空间的 ID，数据空间是文件组 0＝日志文件
name	文件的逻辑名称
physical_name	操作系统文件名称
state	文件状态 0＝联机 1＝正在复原 2＝正在恢复 3＝等待恢复 4＝可疑 5＝为将来保留 6＝脱机 7＝不再使用

续表

列	说明
state_desc	文件状态的描述 联机 正在复原 正在恢复 等待恢复 可疑 脱机 不再使用
size	文件的当前大小，以 8KB 页面为单位 0＝不适用 对数据库快照来说，大小反映快照文件能够使用的最大空间
max_size	最大文件大小，以 8KB 页面为单位 0＝不允许增长 －1＝文件会增长，到磁盘满为止 268435456＝日志文件最多增长到 2 太字节
growth	0＝文件的大小固定，不能增长
is_media_read_only	>0＝文件会自动增长 如果 is_percent_growth＝0，那么增长增量的单位是 8KB 页面，四舍五入到最近的 64KB 如果 is_percent_growth＝1，那么增长增量以整数百分比表示
is_read_only	1＝文件在只读媒体上 0＝文件在读/写媒体上
is_sparse	1＝文件是稀疏文件 0＝文件不是稀疏文件 （稀疏文件是和数据库快照一起使用的，在本章后面讨论）
is_percent_growth	请看上面 growth 列的说明
is_name_reserved	1＝只有在下一次日志备份之后才可以重新使用被删除的文件名（名称或 physical_name）。从数据库中删除文件时，在下一次日志备份之前，逻辑名称都会保持“被保留”的状态。这一列只对完全恢复模型和批量记录恢复模型有效

3.4 创建数据库

创建数据库最简单的方法是使用 Management Studio 中的“对象资源管理器”，它为创建数据库并设置数据库属性的 T-SQL 命令提供图形前端。图 3-1 所示为“新建数据库”窗口，它显示创建新用户数据库的 T-SQL *CREATE DATABASE* 命令。只有具有合适权限的人才能通过“对象资源管理器”或 *CREATE DATABASE* 命令创建数据库，这些人包括 *sysadmin* 角色中的任何人、被授予服务器“控制”或“修改”权限的任何人和被具有 *sysadmin* 或 *dbcreator* 角色的人赋予 CREATE DATABASE 权限的任何人。

在创建新数据库时，SQL Server 会复制 *model* 数据库。如果想在之后每个用户数据库中创建某个对象，应该先在 *model* 中创建它。也可以用 *model* 在所有之后创建的数据库中设置默认的数据库选项。*model* 数据库包括 53 个对象——45 个系统表、6 个用于 SQL Server 查询通知和 Service Broker 的对象、1 个用

于帮助管理文件流数据的表和 1 个帮助管理变更跟踪的表。从系统表 *sys.objects* 中选择对象，可以看到它们。但是如果在 *model* 数据库中运行过程 *sp_help*，就会列出 1978 个对象，它们中的大多数都不会真正存储在 *model* 数据库中，但是可以通过 *sp_help* 进行访问。在第 5 章中，我会讲述其他的对象，以及如何分辨对象是否真的存储在某个数据库中。在任何数据库中运行 *sp_help* 时，都会显示在 *model* 中看到的大部分对象，但是这个列表可能还有用户数据库中的更多对象，*model* 的内容只是起点。

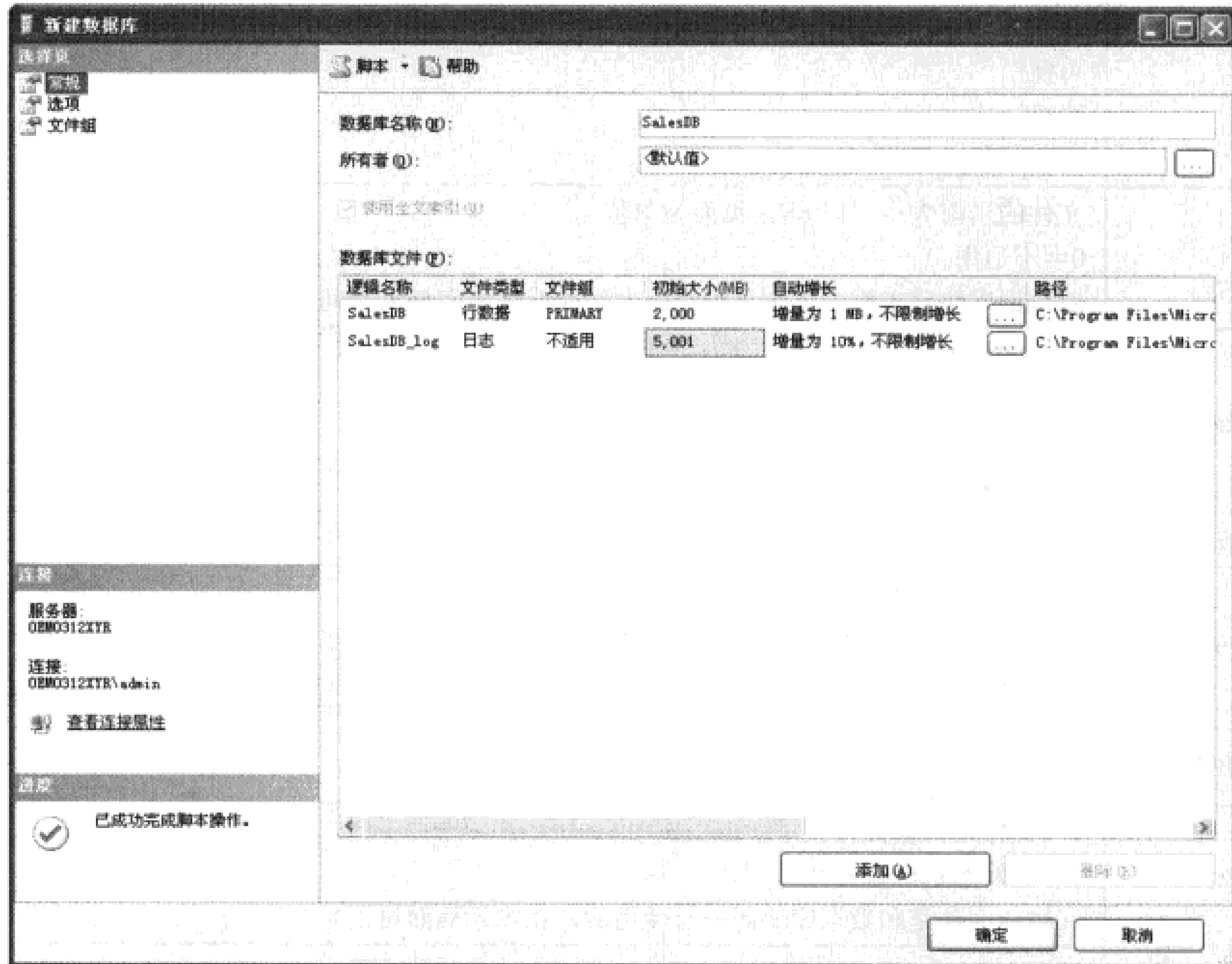


图 3-1 “新建数据库”窗口，在此可以创建新数据库

新建的用户数据库必须是 3MB 或者更大（包括事务日志），主数据文件大小必须至少和 *model* 数据库的主数据文件一样大（*model* 数据库只有一个文件，不能为了添加更多的文件而修改它。因此对 *model* 来说，主数据文件和数据库的大小基本上是相同的）。几乎 *CREATE DATABASE* 命令的所有参数都具有默认值，因此可以用 *CREATE DATABASE* 的简单形式来创建数据库，例如：

```
CREATE DATABASE newdb;
```

这个命令在两个文件上创建具有默认大小的 *newdb* 数据库，文件的逻辑名称（*newdb* 和 *newdb_log*）源于数据库的名称。在默认的数据目录中创建相应的物理文件（newdb.mdf 和 newdb_log.ldf），通常在安装 SQL Server 时确定默认目录。

创建数据库的 SQL Server 登录账户称为*数据库所有者*，该信息与数据库属性的信息一起存储在 *master* 数据库中。数据库只能有一个真正的所有者且总是与登录名相对应。使用任何数据库的任何登录都在该数据库中有用户名，它可能会与登录名相同，但并不是必须的。在使用所拥有的数据库时，数据库所有者的登录总是有特别的用户名 *dbo*。我会在本章后面讨论数据库用户，那时我会讲述数据库安全的

基础知识。数据文件的默认大小是 *model* 数据库的主数据文件的大小（默认是 2MB），日志文件的默认大小是 0.5MB。数据库名称 *newdb* 是否区分大小写取决于在安装过程中所选择的排列顺序。如果接受默认值，名称就不区分大小写（注意，不管为数据所选的大小写敏感性如何，命令 *CREATE DATABASE* 都不区分大小写）。

其他默认的属性值会应用到新数据库及其文件。例如，如果没有指定 LOG ON 语句，但是指定了数据文件，那么 SQL Server 会以所有数据文件大小之和的 25%创建一个日志文件。

如果没有为文件指定 MAXSIZE 语句，那么文件就会增长，直到磁盘满为止（换言之，认为文件的大小是无限的）。可以用 TB、GB 和 MB（默认单位）为单位，为 *SIZE*、*MAXSIZE* 和 *FILEGROWTH* 设置值，也可以用百分比指定 *FILEGROWTH* 属性。*FILEGROWTH* 为 0 表示没有增长。如果没有指定 *FILEGROWTH* 的值，数据文件的默认增长增量是 1MB。日志文件 *FILEGROWTH* 默认被指定为 10%。

3.4.1 CREATE DATABASE 例子

以下是 *CREATE DATABASE* 命令的完整例子，指定 3 个文件和每个文件的所有属性。

```
CREATE DATABASE Archive
ON
PRIMARY
( NAME = Arch1,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\archdat1.mdf',
SIZE = 100MB,
MAXSIZE = 200MB,
FILEGROWTH = 20MB),
( NAME = Arch2,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\archdat2.ndf',
SIZE = 10GB,
MAXSIZE = 50GB,
FILEGROWTH = 250MB)
LOG ON
( NAME = Archlog1,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\archlog1.ldf',
SIZE = 2GB,
MAXSIZE = 10GB,
FILEGROWTH = 100MB);
```

3.5 扩展或收缩数据库

可以手动或自动扩展和收缩数据库。自动扩展架构与自动收缩架构完全不同，手动扩展的处理也和手动收缩的处理不同。日志文件有自己的增长和收缩规则，我会在第 4 章中讨论改变日志文件的大小。

警告：

收缩数据库或任何数据文件都是极为消耗资源的操作，只有一定要收回磁盘空间时再这么做。收缩数据文件也会在数据库中产生过量的逻辑碎片。我们将在第 6 章讨论碎片，在第 11 章讨论收缩。

3.5.1 自动文件扩展

数据库中的任何文件已满后，都能自动进行扩展。文件属性 *FILEGROWTH* 决定如何进行自动扩展。最初定义文件时所指定的 *FILEGROWTH* 属性可以用后缀 TB、GB、MB、KB 或%来量化，而且总是四舍五入到最近的 64KB。如果按百分比指定值，那么发生扩展时，增长增量就是所指定的文件大小的百分比。文件属性 *MAXSIZE* 设置文件大小的上限。

让 SQL Server 自动增长数据文件并不能代替在建立和生成任何表之前进行好的容量规划。启用自动增长可能会防止一些意外的数据量增长而造成的失败，但是也会带来问题。如果数据文件已满，假设自动增长百分比设置为以 10%增长，应用程序尝试插入一行但是没有空间，数据库可能就会大量增长（10GB 的 10%是 1GB）。如果没有使用快速文件初始化（在下一节讨论），增长可能需要大量时间。增长需要的时间可能太长，以至于超过客户端应用程序的超时值，即插入查询失败。如果不设置自动增长，反正查询会失败的，但是若启用了自动增长，SQL Server 会花大量时间增长文件，您也不会立即知道这个问题。而且，文件增长会导致磁盘上有物理碎片。

即使启用自动增长，数据库文件还是无法将数据库增长到超过文件所在磁盘的可用磁盘空间的限制，或者超过 *MAXSIZE* 文件属性所指定的大小。所以如果依靠自动增长功能定义数据库的大小，还是必须独立检查可用硬盘空间或整个文件的大小（未公开的扩展过程 *xp_fixeddrives* 返回每个本地卷的可用磁盘空间大小的列表）。为了降低空间不够的可能性，可以监视 Performance Monitor counter SQL Server（SQL Server 性能监视器计数器）；Databases Objects（数据库对象）；Data File Size（数据文件大小），然后设置数据库文件超过某个大小时发出性能警告。

3.5.2 手动文件扩展

可以用 *ALTER DATABASE* 命令的 MODIFY FILE 选项改变一个或多个文件的 *SIZE* 属性，来手动扩展数据库文件。修改数据库时，新文件大小必须比当前的大。要想减小文件大小，可使用 *DBCC SHRINKFILE* 命令，稍后会讲到。

3.5.3 快速文件初始化

SQL Server 2008 数据文件（但不是日志文件）的初始化可以立即完成，这样可以快速地创建和增长文件。即时文件初始化向数据文件中添加空间，而不是用零填充新添加的空间。只有新数据写入文件时才会覆盖实际的磁盘内容。在覆盖数据之前，总是有可能让使用外部文件阅读器工具的黑客看到以前磁盘上的数据。虽然 SQL Server 2008 文档说即时文件初始化功能是个“选择”，但这在 SQL Server 中并不是真的选择，它实际上是通过被默认授予 Windows 管理员的、称为 SE_MANAGE_VOLUME_NAME 的 Windows 安全设置来控制的（也可以将其他的 Windows 用户添加到 Perform Volume Maintenance Tasks 安全策略中，将这个权利授予他们）。如果您的 SQL Server 服务账户是 Windows 管理员角色且 SQL Server 运行在 Windows XP、Windows Server 2003 或 Windows Server 2008 文件系统上，就会使用即时文件初始化。如果想在创建和扩展数据文件时，确保它们被清零，可以使用跟踪标记 1806 或拒绝运行 SQL Server 服务的账户的 SE_MANAGE_VOLUME_NAME 权利。

3.5.4 自动收缩性

数据库属性 *autoshrink* 可让数据库自动收缩，其效果与 *DBCC SHRINKDATABASE（dbname, 25）*相

同，这个选项会在收缩后在数据库中保留25%的可用空间，任何超过它的可用空间都会归还给操作系统。进行自动收缩的线程以很高的频率收缩数据库，有时候甚至每30分钟一次。收缩数据文件很耗资源，所以应该只在没有其他办法收回所需的磁盘空间时才这么做。

重要提示：

永远不推荐使用自动收缩。实际上，Microsoft公司宣布会在以后的SQL Server版本中删除自动收缩选项，所以您应该避免使用它。

3.5.5 手动收缩

可以使用以下DBCC命令之一手动收缩数据库。

```
DBCC SHRINKFILE ( {file_name | file_id }
[, target_size][, {EMPTYFILE | NOTRUNCATE | TRUNCATEONLY} ] )

DBCC SHRINKDATABASE (database_name [, target_percent]
[, {NOTRUNCATE | TRUNCATEONLY} ] )
```

1. DBCC SHRINKFILE

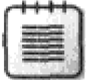
DBCC SHRINKFILE 可以收缩当前数据库中的文件。指定 *target_size* 时，*DBCC SHRINKFILE* 就会尝试把指定文件收缩到指定的大小（MB）。即将释放的空间中的已用页面会被移动到被保留部分的可用空间中。例如，对于一个15MB数据文件来说，带有 *target_size* 为12的 *DBCC SHRINKFILE* 语句会让文件最后3MB中的所有已用页面重新放到文件前12MB中的任何可用空间中。*DBCC SHRINKFILE* 不会将文件收缩到比存储文件所需的大小还小。例如，如果一个10MB数据文件中70%的页面都已经被使用，那么带有 *target_size* 为5的 *DBCC SHRINKFILE* 语句只会把文件收缩为7MB，而不是5MB。

2. DBCC SHRINKDATABASE

DBCC SHRINKDATABASE 收缩数据库中的所有文件，但是不允许任何文件收缩到比最小值还小。数据库文件的最小值是文件的初始大小（在创建数据库时指定的）或者是用 *ALTER DATABASE* 和 *DBCC SHRINKFILE* 命令明确扩展或减小文件的大小。如果需要把数据库收缩到比最小值还小，应该用 *DBCC SHRINKFILE* 命令将单个数据库文件收缩到指定大小。将文件收缩后的大小将成为新的最小值。

传递到 *DBCC SHRINKDATABASE* 命令的数值参数 *target_percent* 是在数据库的每个文件中所保留的可用空间的百分比。例如，如果已经使用了100MB数据库文件中的60MB，那么可以指定收缩百分比为25%。然后SQL Server会将文件收缩到80MB，除了最初的60MB数据之外，还会有20MB的可用空间。换言之，80MB文件具有25%的可用空间。但是如果已经使用了100MB数据库文件的80MB或者更多，那么SQL Server就没办法将这个文件收缩到留出25%的可用空间。在这种情况下，文件大小就不会改变。

DBCC SHRINKDATABASE 是以文件为单位收缩数据库的，因此用于进行数据文件收缩的架构与 *DBCC SHRINKFILE* 指定数据文件时所用的架构相同。SQL Server首先会把页面移动到文件的前面，释放结尾处的空间，然后将相应的已释放的那些页面释放给操作系统。如何收缩数据文件的内部细节将在第11章中讨论。



注意：

收缩日志文件与收缩数据文件的区别很大，可以多大程度收缩日志文件和收缩时会发生什么情况都需要了解如何使用日志。因此，我把收缩日志文件的讨论推迟到第4章再讲。

正如本节开始的警告所说明的，收缩数据库或任何数据文件都是耗费资源的操作。如果一定要从数据库恢复磁盘空间，应该仔细计划收缩操作，并且在对系统的其他部分影响最小时进行。永远都不应该启用 AUTOSHRINK 选项，它会以固定时间间隔收缩*所有*数据文件，会严重破坏系统的性能。因为收缩数据文件会围绕文件移动数据，所以会产生碎片，您可能在这之后又想删除碎片。因为清理数据文件的碎片要使用系统资源，所以会影响生产力。碎片和清理碎片会在第6章讨论。

收缩操作有可能会被启用了快照隔离级别之一的事务所阻挡。如果发生这种情况，*DBCC SHRINKFILE* 和 *DBCC SHRINKDATABASE* 就会在第一个小时内的每5分钟和之后的每个小时向错误日志写入具有信息性的消息。SQL Server 还通过 *sys.dm_exec_requests* 视图为 *SHRINK* 命令提供进程报告。进程报告将在第11章中讨论。

3.6 使用数据库文件组

出于分配和管理的目的，可以将数据库的数据文件分为文件组。在某些情况下，可以把数据和索引放在特定的文件组、特定的驱动器或卷上来提高性能。包含主数据文件的文件组称为*主要文件组*。只有一个主要文件组，如果创建数据库时没有特别说明把文件放在其他的文件组中，*所有*的数据文件都会放在主要文件组中。

除了主要文件组，数据库还可以有一个或多个用户自定义的文件组。可以通过在 *CREATE DATABASE* 或 *ALTER DATABASE* 命令中使用 *FILEGROUP* 关键字创建用户定义文件组。

不要混淆主要文件组和主文件，下面是它们的区别。

- 主文件总是在创建数据库时第一个列出的文件，而且通常具有文件扩展名.mdf。主文件的一个特殊功能是它具有指向 *master* 数据库中表的指针（可以通过目录视图 *sys.database_files* 访问），*master* 数据库包含关于所有属于该数据库的文件的信息。
- 主要文件组总是包含主文件的文件组，这个文件组包含主数据文件和没有放到其他特定文件组中的所有文件。系统表中的所有页面总是由主要文件组中的文件分配的。

3.6.1 默认文件组

文件组总是具有 *DEFAULT* 属性。注意 *DEFAULT* 是文件组的属性，而不是名称。每个数据库中只有一个文件组可以是默认文件组。默认情况下，主文件组也是默认文件组。数据库的所有者可以用 *ALTER DATABASE* 命令改变默认文件组。创建表或索引时，如果没有指定特定的文件组，就会在默认文件组中创建表或索引。

大多数 SQL Server 数据库在一个（默认）文件组中都有一个数据文件。实际上，大多数用户可能从来都不够了解 SQL Server 如何工作，因而也不知道文件组是什么。随着用户数据库水平的提高，可能会决定用多个设备为数据库分担 I/O。这么做最容易的方法是在 RAID 设备上创建数据库文件，但是可能还是不需要使用文件组。水平和复杂性更高一些时，用户可能会决定是否真的想要多个文件——也许要创

建的数据库需要的空间比单个驱动器上的可用空间还多。在这种情况下，仍然不需要文件组，可以用 *CREATE DATABASE* 和独立磁盘上的文件列表达到目的。

经验更丰富的数据库管理员可能会决定把不同的表分配给不同的驱动器，或者使用 SQL Server 2008 中的表和索引分区功能。只有这时才需要文件组，然后可以用 Management Studio 中的“对象资源管理器”在多个文件组上创建数据库。之后可以在“对象资源管理器”中右键单击数据库名称，并创建 *CREATE DATABASE* 命令的脚本，它在合适的文件组中包括所有文件。当需要重建数据库或重建类似的数据库时，可以保存和重用这个脚本。

为什么使用多个文件?

您可能奇怪为什么想要在一个物理驱动器上的多个文件上创建数据库。通常这么做没有性能上的优势，但是会在两个重要方面具有灵活性。

首先，如果磁盘损坏，需要从备份还原数据库，那么新数据库必须包含与原始数据库相同数目的文件。例如，如果原始数据库是一个 120GB 的大文件，就需要将它恢复到这么大的数据库中。如果没有另外可以立即使用的 120GB 驱动器，就无法还原数据库。但是如果最初在几个小文件上创建数据库，那么还原的过程就会更灵活。有几个 40GB 的驱动器比有一个 120GB 的驱动器更有可能。

第二，如果将来修改硬件配置的话，将数据库分为多个文件（即使在相同的驱动器上）能够更灵活地将数据库移动到独立的驱动器上（详情请看本章后面的“移动或复制数据库”一节）。

已经被分配了空间的对象（即表和索引）是在特定的文件组上被创建的（也可以在分区架构上创建它们，分区架构是文件组的集合。我会在第 7 章讨论分区和分区架构）。如果没有指定文件组（或分区架构），就会在默认的文件组上创建对象。向存储在特定文件组中的对象添加空间时，会以*成比例填充*的方式存储数据，就是说如果文件组中的一个文件的可用空间是另一个文件中可用空间的两倍，那么为第一个文件分配两个扩展（或者说空间单位），而只为第二个文件分配一个扩展（我会在本章后面的“空间分配”部分更详细地讨论扩展）。推荐把所有的文件都创建为相同大小，避免成比例填充的问题。

用文件组还可以备份部分数据库内容。因为表是在单个文件组上创建的，所以可以通过备份这些表所在的文件组，来备份某些关键表。也可以用两种方法恢复单个文件或文件组。首先，可以部分还原数据库且只还原文件组的一个子集，文件组必须总是包括主要文件组。只要还原了主要文件组，数据库就会联机，但是只有在还原的文件组上创建的对象才可用。只还原一部分文件组会在强制的时间段内有大量的数据库可用。另外，如果创建数据库的磁盘中有几个损坏，在还原现有数据库的同时，还可以还原损坏磁盘上的文件组备份。这种还原方法也需要有日志备份，所以我会在第 4 章中更详细地讨论这个话题。

3.6.2 FILEGROUP CREATION 例子

这个例子用 3 个文件组创建名为 *sales* 的数据库。

- 主要文件组，具有文件 *salesPrimary1* 和 *salesPrimary2*。这两个文件的 *FILEGROWTH* 都被指定为 100MB。
- 名为 *SalesGroup1* 的文件组，具有文件 *salesGrp1File1* 和 *salesGrp1File2*。

- 名为 *SalesGroup2* 的文件组，具有文件 *salesGrp2File1* 和 *salesGrp2File2*。

```
CREATE DATABASE Sales
ON PRIMARY
( NAME = salesPrimary1,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\salesPrimary1.mdf',
SIZE = 100,
MAXSIZE = 500,
FILEGROWTH = 100 ),
( NAME = salesPrimary2,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\salesPrimary2.ndf',
SIZE = 100,
MAXSIZE = 500,
FILEGROWTH = 100 ),
FILEGROUP SalesGroup1
( NAME = salesGrp1File1,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\salesGrp1File1.ndf',
SIZE = 500,
MAXSIZE = 3000,
FILEGROWTH = 500 ),
( NAME = salesGrp1File2,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\salesGrp1File2.ndf',
SIZE = 500,
MAXSIZE = 3000,
FILEGROWTH = 500 ),
FILEGROUP SalesGroup2
( NAME = salesGrp2File1,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\salesGrp2File1.ndf',
SIZE = 100,
MAXSIZE = 5000,
FILEGROWTH = 500 ),
( NAME = salesGrp2File2,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\salesGrp2File2.ndf',
SIZE = 100,
MAXSIZE = 5000,
FILEGROWTH = 500 )
LOG ON
( NAME = 'Sales_log',
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\saleslog.ldf',
SIZE = 5MB,
MAXSIZE = 25MB,
FILEGROWTH = 5MB );
```

3.6.3 文件流文件组

我在第 1 章中谈到配置选项时，简要提到了文件流存储。创建数据库时，可以像创建普通文件组那样创建文件流文件组，但是必须用 CONTAINS FILESTREAM 语句指定该文件组是为文件流数据所使用

的。和普通的文件组不同，每个文件流文件组只能包括一个文件引用，该文件被指定为操作系统文件夹，而不是特定文件。到最后一个文件夹的路径必须存在，但是最后一个文件夹不能存在。所以在我的例子中，必须存在路径 C:\Data，但是在执行 CREATE DATABASE 命令时不能存在 *Review_FS* 子文件夹。也不像普通的文件组那样，不会为文件组预分配空间，也不会为文件组内的文件指定大小和增长信息。随着数据加入到用文件流列所创建的表中，文件和文件组会增长。

```
CREATE DATABASE MyMovieReviews
ON
PRIMARY
  ( NAME = Reviews_data,
    FILENAME = 'c:\data\Reviews_data.mdf'),
FILEGROUP MovieReviewsFSGroup1 CONTAINS FILESTREAM
  ( NAME = Reviews_FS,
    FILENAME = 'c:\data\Reviews_FS')
LOG ON ( NAME = Reviews_log,
    FILENAME = 'c:\data\Reviews_log.ldf');
GO
```

如果运行之前的代码，就会在 C:\Dada\Reviews_FS 文件夹中看到 Filestream.hdr 文件和$FSLOG 文件夹。Filestream.hdr 文件是 FILESTREAM 容器头文件，不应该修改或删除这个文件。对于现有数据库，可以用 *ALTER DATABASE* 添加文件流文件组，我会在下一节讲这个命令。*MovieReviewsFSGroup1* 中的所有列中的所有数据都与 *Reviews_FS* 文件夹中创建的单个文件一起维护和管理。我会在第 7 章讲述特殊存储格式时，更多地讲解此文件夹中的文件组织。

3.7 修改数据库

使用 *ALTER DATABASE* 命令可以改变以下数据库的定义。

- 改变数据库的名称。
- 向数据库添加一个或多个新数据文件。可以把这些文件放在用户自定义的文件组中。所有用同一个 *ALTER DATABASE* 命令添加的文件都必须在相同的文件组中。
- 向数据库添加一个或多个新日志文件。
- 从数据库删除文件或文件组。只有当文件或文件组完全空白时，才能这么做。删除文件组会删除其中的所有文件。
- 向数据库添加新文件组（必须用单独的 *ALTER DATABASE* 命令向这些文件组添加文件）。可用以下方法之一修改现有文件。
  - 增加 *SIZE* 属性的值。
  - 改变 *MAXSIZE* 或 *FILEGROWTH* 属性。
  - 通过指定 *NEWNAME* 属性，改变文件的逻辑名称。然后将 *NEWNAME* 的值用做将来引用该文件的 *NAME* 属性。
  - 改变文件的 *FILENAME* 属性，这会有效地将文件移动到新位置。新名称或新位置在重新启动 SQL Server 时才会生效。对于 *tempdb* 来说，SQL Server 会自动在新位置用新名称创建文件。对于其他数据库，必须在停止 SQL Server 实例之后手动移动文件，然后 SQL Server 重新启动时会找到新文件。

- 将文件标记为 OFFLINE。应该在物理文件已损坏且还原文件可以使用文件备份时将文件设置为 OFFLINE（还有一个选项将整个数据库标记为 OFFLINE，我稍后讲数据库属性时再讨论它）。可以通过将文件标记为 OFFLINE，让 SQL Server 重新启动时不还原该文件。可用以下方法之一修改现有的文件组。
    - 将文件组标记为 READONLY（只读），从而不允许对文件组中的对象进行更新。不能把主文件标记为 READONLY。
    - 将文件组标记为 READWRITE，它会反转 READONLY 属性。
    - 将文件组标记为数据库的默认文件组。
    - 改变文件组的名称。
- 改变一个或多个数据库选项（我会在本章后面讨论数据库选项）。

每次执行 *ALTER DATABASE* 命令时只能做出一个改变。注意，不能把文件从一个文件组移动到另一个文件组。

3.7.1 ALTER DATABASE 例子

下面的例子说明了能够用 *ALTER DATABASE* 命令做出的改变。

这个例子增加数据库文件的大小：

```
USE master
GO
ALTER DATABASE Test1
MODIFY FILE
( NAME = 'test1dat3',
SIZE = 2000MB);
```

下面的例子在数据库中创建了新文件组，并向文件组添加了两个 500MB 的文件，还让新文件组成为默认文件组。需要 3 条 *ALTER DATABASE* 语句：

```
ALTER DATABASE Test1
ADD FILEGROUP Test1FG1;
GO
ALTER DATABASE Test1
ADD FILE
( NAME = 'test1dat4',
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\t1dat4.ndf',
SIZE = 500MB,
MAXSIZE = 1000MB,
FILEGROWTH = 50MB),
( NAME = 'test1dat5',
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\t1dat5.ndf',
SIZE = 500MB,
MAXSIZE = 1000MB,
FILEGROWTH = 50MB)
TO FILEGROUP Test1FG1;
GO
ALTER DATABASE Test1
MODIFY FILEGROUP Test1FG1 DEFAULT;
GO
```

3.8 数据库剖析

数据库包括永久存储用户对象（如表和索引）的用户定义空间，这个空间位于一个或多个操作系统文件中。

数据库被分成逻辑页面（每个 8KB），在每个文件中，页面从 0 到 *x* 连续编号，*x* 定义为文件的大小。可以通过指定数据库 ID、文件 ID 和页面号引用任何页面。使用 *ALTER DTABASE* 命令扩大文件时，会在文件结尾处添加新空间，即新分配空间的首页是正在扩大的文件上的 *x*+1 页。使用 *DBCC SHRINKDATABASE* 或 *DBCC SHRINKFILE* 命令收缩数据库时，从数据库中编号最高的页面（在结尾处）开始删除页面，向编号低的页面移动。这样能保证文件中的页面编号总是连续的。

用 *CREATE DATABASE* 命令创建新数据库时，会给它赋予唯一的数据库 ID，可以在 *sys.databases* 视图中看到新数据库的行。在 *sys.databas es* 中返回的行包括每个数据库的基本信息，如它的名称、*database_id*、创建时间及可以用 *ALTER DATABASE* 命令设置的每个数据库选项。我会在本章后面更详细地讨论数据库选项。

3.8.1 空间分配

数据库中的空间用于存储表和索引。以单位管理的空间称为*扩展*。扩展是由 8 个逻辑连续的页面（或 64KB 空间）组成的。为了更有效地分配空间，SQL Server 2008 不会将整个扩展分配给含有少量数据的表。SQL Server 2008 有两种类型的扩展。

- **统一扩展**。由单个对象拥有，只有拥有扩展的对象才能使用扩展中的 8 个页面。
- **混合扩展**。它们最多由 8 个对象共享。

SQL Server 从混合扩展为新表或索引分配页面。当表或索引增长到 8 页时，将来所有的分配都会使用统一扩展。

当表或索引需要更多空间时，SQL Server 需要找到可以被分配的空间。如果表或索引仍然少于 8 个页面，SQL Server 就必须在可用空间内找到混合扩展。如果表或索引是 8 页或者更大，SQL Server 就必须寻找可用的统一扩展。

SQL Server 用两种特殊类型的页面记录已经分配了哪些扩展及扩展可以用于哪种类型（混合或统一）。

- **全局分配映射（GAM）页面**。这些页面记录的扩展可以分配给任何用途使用。GAM 对它所涵盖的区间中的每个扩展都有一个位。如果该位是 0，则相应的扩展正在使用中；如果该位是 1，说明该扩展是可用的。考虑了头文件和其他开销之后，页面上有 8 000 字节（或者说 64 000 位）可用，所以每个 GAM 可以涵盖 64 000 个扩展或者 4GB 数据。这说明文件中每 4GB 文件大小都会有一个 GAM 页面。
- **共享全局分配映射（SGAM）页面**。这些页面记录的扩展现在用做混合扩展，至少有一个未使用的页面。就像 GAM 一样，每个 SGAM 涵盖约 64 000 个扩展或者约 4GB 数据。SGAM 对它所涵盖的区间中的每个扩展都有一个位。如果该位是 1，那么被使用的扩展是混合扩展且是可用的页面；如果该位是 0，那么扩展就没有用做混合扩展，或者是页面都是被使用的混合扩展。

表 3-2 显示了每个扩展根据其当前用途在 GAM 和 SGAM 页面中设置的位模式。

表 3-2 GAM 和 SGAM 页面中的位设置

扩展的当前使用	GAM 位设置	SGAM 位设置
可用，没有使用	1	0
统一扩展或完全混合扩展	0	0
带可用页面的混合扩展	0	1

有几个工具可以检查 GAM 和 SGAM 中的位。第 5 章讨论 *DBCC PAGE* 命令，可以使用它在查询窗口中查看 SQL Server 数据库页面的内容。因为 GAM 和 SGAM 的页面数已知，所以我们只看页面 2 或页面 3。如果使用格式 3（它提供大部分详细信息），可以看到输出显示已经被分配哪些扩展、没有分配哪些扩展。图 3-2 是使用 *DBCC PAGE*、以格式 3 显示我的 *AdventureWorks2008* 数据库中第一个 GAM 页面后输出的最后一段。

```
(1:0)       - (1:24256)  =     ALLOCATED
(1:24264)   -            = NOT ALLOCATED
(1:24272)   - (1:29752)  =     ALLOCATED
(1:29760)   - (1:30168)  = NOT ALLOCATED
(1:30176)   - (1:30240)  =     ALLOCATED
(1:30248)   - (1:30256)  = NOT ALLOCATED
(1:30264)   - (1:32080)  =     ALLOCATED
(1:32088)   - (1:32304)  = NOT ALLOCATED
```

图 3-2 GAM 页面内容显示文件中扩展的分配状态

这个输出表示分配了从 24 256 页开始的所有扩展，这对应于文件的前 189MB。没有分配从 24 264 页开始的扩展，但是已经分配了后面的 5 480 个页面。

也可以使用称为 *SQL Inernals Viewer* 的图形工具查看已分配的扩展。SQL Inernals Viewer 是 *http://www.SQLInternalsViewer.com* 上的免费工具，本书的随附网站上也有。图 3-3 显示了我的 ***master*** 数据库的主要分配页面。GAM 和 SGAM 已经被合在一个显示区，显示每个页面而不只是每个扩展的状态。绿色方块表示正在使用该 SGAM，但是没有使用扩展，所以还有页面可以用于单页分配。蓝色方块显示已经设置了 GAM 位和 SGAM 位，所以相应的扩展完全不可用。灰色方块显示扩展可用。

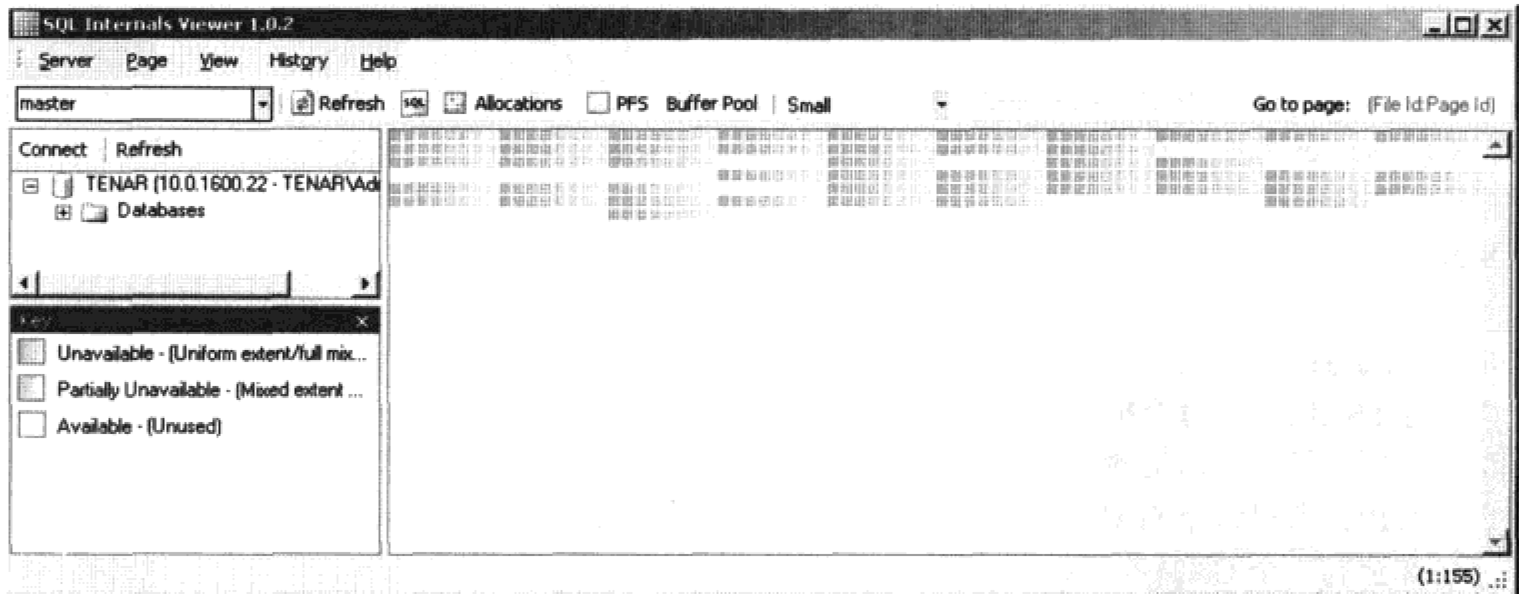


图 3-3 SQL Internals Viewer 显示每个页面的分配状态

如果 SQL Server 需要找到新的、完全未使用的扩展，它可以使用 GAM 页面中相应位值为 1 的任何扩展。如果 SQL Server 需要找到包含可用空间（一个或多个可用页面）的混合扩展，它会在 SGAM 中寻找值为 1（GAM 中的值总是 0）的扩展。如果没有具有可用空间的混合扩展，SQL Server 会使用 GAM 页面找到一个全新扩展，将它分配为混合扩展，然后使用其中一个页面。如果根本没有可用扩展，文件就

是满的。

SQL Server 能够快速找到文件中的 GAM，因为 GAM 总是任何数据库文件的第 3 个页面（即页 2）。SGAM 是第 4 个页面（即页 3）。在页 2 之后，每 511 230 个页面就会有一个 GAM，页 3 之后每 511 230 个页面会有一个 SGAM。任何文件的页 0 都是“文件头”页面，每个文件只有一个页 0。页 1 是“页面可用空间（PFS）”页面。我会在第 5 章讲述表中每个页面的内容及 PFS 页面的详细信息。因为现在讲的是空间分配，所以我会讲如何记录哪些页面属于哪些表。

IAM 页面记录分配单元使用的数据库文件中 4GB 区段中的扩展。分配单元是属于表或索引中单个分区的页面集，它包括 3 种存储类型之一的页面：保存一般行中数据的页面、保存大型对象（LOB）数据的页面或者保存行溢出数据的页面。我会在第 5 章讨论这些一般的行中存储，在第 7 章讨论 LOB、行溢出存储和分区。

例如，具有 3 种数据类型（行中存储、LOB 和行溢出）的 4 个分区上的表至少有 12 个 IAM 页面。单个 IAM 页面还是只包括单个文件的 4GB 区段，所以如果分区跨越几个文件，就会有多个 IAM 页面。如果文件超过 4GB 大小且分区使用的页面超过 4GB 区段，就会有额外的 IAM 页面。

就像后面有 IAM 页面头文件的所有其他页面那样，IAM 页面也包含 96 字节页面头文件，它包括 8 个页指针槽。最后，IAM 页面包括将扩展范围映射到一个文件的位集，该文件不必是 IAM 页面所在的文件。头文件含有由 IAM 映射的扩展范围内第一个扩展的地址。8 个页指针槽可能会包含指向混合扩展中相关对象的页面的指针；只有对象的第一个 IAM 有一些值在这些指针中。一旦对象超过 8 页，那么所有其他扩展都是统一扩展，就是说一个对象从来不需要超过 8 个指针指向混合扩展中的页面。如果从表中删除行，那么表实际上使用的指针少于 8 个。不管是否把扩展分配给拥有 IAM 的对象，位图的每一位都代表扩展范围中的一个扩展。如果位打开，扩展范围中的相关扩展就被分配给拥有 IAM 的对象。如果位关闭，相关的扩展就不会分配给拥有 IAM 的对象。

例如，如果 IAM 的第一个字节中的位模式是 1100 0000，那么就会把该范围中的第一个和第二个扩展分配给拥有 IAM 的对象，不会把扩展 3～8 分配给拥有 IAM 的对象。

IAM 页面是按每个对象的需要进行分配的，且在数据库文件中随意放置。每个 IAM 都可能会包括大约 512 000 个页面的范围。

名为 *sys.system_internals_allocation_units* 的内部系统视图有一列指向分配单元的第一个 IAM 页面，该列称为 *first_iam_page*。该分配单元的所有 IAM 页面都链接在一个链中，每个 IAM 页面都包含一个指针，指向链中的下一个 IAM 页面。可以在第 5 章、第 6 章、第 7 章中讨论对象和索引存储时找到更多关于 IAM 和分配单元的信息。

除了 GAM、SGAM 和 IAM，数据库文件还有其他 3 种类型的特殊分配页面。PFS 页面记录如何使用文件中的每个特定页面。文件的第 2 页（页 1）是 PFS 页面，在这之后每 8 088 个页面就是一个 PFS 页面。我会在第 5 章更多地讨论它们。第 7 页（页 6）称为差分更改映射（Differential Changed Map，DCM）页面，它记录自上一次完全数据库备份后文件中被修改过的扩展。第 8 页（页 7）称为批量更改映射 Bulk Changed Map（BCM）页面，在文件中的扩展在最少操作或批量记录操作时使用。我会在第 4 章讲备份和还原操作的内部时，更多讲述这两种页面。DCM 和 BCM 页面与 GAM 和 SGAM 页面类似，其中有一位对应于它们所代表的文件区段中的每个扩展。它们以固定间隔出现，即每 511 230 页出现一次。

可以用 *DBCC PAGE* 或 SQL Internals Viewer 查看 IAM 和 PFS 页面及 DCM 和 BCM 页面的详情。我会在后几章中涵盖不同类型分配页面的细节时展示更多的 *DBCC PAGE* 输出例子。

3.9　设置数据库选项

可以为数据库设置几十个选项或属性来控制该数据库中的某个行为。一些选项必须设为 ON（开）或 OFF（关），另一些必须设为可能值之一，其他的只能通过指定名称才能启用。默认情况下，除非在 *model* 数据库中将需要开或关的选项设为开，否则所有选项的初始值都是关。所有在 *mode* 中选项发生改变之后创建的数据库都与 *model* 有相同的值。可以通过 Management Studio 轻松改变一些选项的值。用 *ALTER DATABASE* 命令可以直接设置所有选项（也可以用 *sp_dboption* 系统存储过程设置一些选项，但是该过程只是为向后兼容性准备的，已经计划在 SQL Server 的下一版本中去除）。

检查 *sys.dtabases* 目录视图，查看所有选项的当前值。视图也包含其他有用的信息，如数据库 ID、创建日期和数据库所有者的安全 ID（SID）。下列查询从 *sys.databases* 检索 SQL Server 全新默认安装后就有的 4 个数据库的一些最重要的列。

```
SELECT name, database_id, suser_sname(owner_sid) as owner,
     create_date, user_access_desc, state_desc
FROM sys.databases
WHERE database_id <= 4;
```

虽然创建日期可能不同，但是查询会得到以下输出：

```
name     database_id  owner create_date             user_access_desc  state_desc
------   -----------  ----- ----------------------- ----------------  ----------
master   1            sa    2003-04-08 09:13:36.390 MULTI_USER        ONLINE
tempdb   2            sa    2008-04-19 12:02:35.327 MULTI_USER        ONLINE
model    3            sa    2003-04-08 09:13:36.390 MULTI_USER        ONLINE
msdb     4            sa    2008-03-21 01:54:05.240 MULTI_USER        ONLINE
```

sys.databases 视图实际上包括 *user_access* 和 *state* 信息的数字和名称。从 *sys.databases* 选择所有的列会显示 MULTI_USER 的 *user_access_desc* 的相应 *user_access* 值为 0，ONLINE 的 *state_desc* 的 *state* 值为 0。*SQL Server 联机丛书*包括 *sys.databases* 中列的数字和名称关系的完整列表。这些只是显示在 *sys.databases* 视图中的两个数据库选项。数据库选项的完整列表主要分为 7 个类别：状态选项、游标选项、自动选项、参数化选项、SQL 选项、数据库还原选项和外部访问选项。还有一些 SQL Server 可以使用针对特定技术的选项，包括数据库镜像、Service Broker 活动、改变跟踪、数据库加密和快照隔离。一些选项（尤其是 SQL 选项）具有相应的 SET 选项，可以为特定的连接打开或关闭这些选项。注意，ODBC 或 OLE DB 驱动器默认会打开一些这样的 SET 选项，所以应用程序进行操作时，就好像已经设置了相应的数据库选项。

下面是按类别列出的选项。每行列出的选项和值以竖线分隔，而且是互斥的。

状态选项

（1）SINGLE_USER | RESTRICTED_USER | MULTI_USER

（2）OFFLINE | ONLINE | EMERGENCY

（3）READ_ONLY | READ_WRITE

游标选项

（1）CURSOR_CLOSE_ON_COMMIT { ON | OFF }

（2）CURSOR_DEFAULT { LOCAL | GLOBAL }

自动选项

（1）AUTO_CLOSE { ON | OFF }

（2）AUTO_CREATE_STATISTICS { ON | OFF }

（3）AUTO_SHRINK { ON | OFF }

（4）AUTO_UPDATE_STATISTICS { ON | OFF }

（5）AUTO_UPDATE_STATISTICS_ASYNC { ON | OFF }

参数化选项

（1）DATE_CORRELATION_OPTIMIZATION { ON | OFF }

（2）PARAMETERIZATION { SIMPLE | FORCED }

SQL 选项

（1）ANSI_NULL_DEFAULT { ON | OFF }

（2）ANSI_NULLS { ON | OFF }

（3）ANSI_PADDING { ON | OFF }

（4）ANSI_WARNINGS { ON | OFF }

（5）ARITHABORT { ON | OFF }

（6）CONCAT_NULL_YIELDS_NULL { ON | OFF }

（7）NUMERIC_ROUNDABORT { ON | OFF }

（8）QUOTED_IDENTIFIER { ON | OFF }

（9）RECURSIVE_TRIGGERS { ON | OFF }

数据库还原选项

（1）RECOVERY { FULL | BULK_LOGGED | SIMPLE }

（2）TORN_PAGE_DETECTION { ON | OFF }

（3）PAGE_VERIFY { CHECKSUM | TORN_PAGE_DETECTION | NONE }

外部访问选项

（1）DB_CHAINING { ON | OFF }

（2）TRUSTWORTHY { ON | OFF }

数据库镜像选项

（1）PARTNER { = 'partner_server' }

（2）| FAILOVER

（3）| FORCE_SERVICE_ALLOW_DATA_LOSS

（4）| OFF

（5）| RESUME

（6）| SAFETY { FULL | OFF }

（7）| SUSPEND

（8）| TIMEOUT integer

（9）}

（10）WITNESS { = 'witness_server' }| OFF }

Service Broker 选项

（1）ENABLE_BROKER | DISABLE_BROKER

（2）NEW_BROKER

（3）ERROR_BROKER_CONVERSATIONS

改变跟踪选项

CHANGE_TRACKING {= ON [<change_tracking_settings> | = OFF}

数据库加密选项

ENCRYPTION {ON | OFF}

快照隔离选项

（1）ALLOW_SNAPSHOT_ISOLATION {ON | OFF }

（2）READ_COMMITTED_SNAPSHOT {ON | OFF } [WITH <termination>]

3.9.1　状态选项

状态选项控制谁可以使用数据库和用数据库进行什么操作。可用性有3个方面：用户访问状态决定哪个用户可以使用数据库；状态选项的状态决定数据库是否可以供每个人使用；可更新状态决定可以对数据库进行什么操作。用 *ALTER DATABASE* 命令为数据库启用一个选项来控制其中一个方面。这些状态选项都不能使用关键字 *ON* 或 *OFF* 来控制状态值。

1. SINGLE_USER | RESTRICTED_USER | MULTI_USER

3个选项 SINGLE_USER（单用户）、RESTRICTED_USER（受限用户）和 MULTI_USER（多用户）描述数据库的用户访问属性。它们互斥，设置其中任何一个选项就会取消对其他选项的设置。要想为数据库设置其中一个选项，只需要使用选项名称。例如，要想把 *AdventureWorks2008* 数据库设置为单一用户模式，使用以下代码：

```
ALTER DATABASE AdventureWorks2008 SET SINGLE_USER;
```

SINGLE_USER 模式的数据库每次只能有一个连接。RESTRICTED_USER 模式的数据库可以有多个连接，但是只能来自被认为是“合格”用户（*dbcreateor* 或 *sysadmin* 服务角色或该数据库的 *db_owner* 角色）的连接。数据库默认是 MULTI_USER 模式，就是说在数据库中具有有效用户名的任何人都能连接它。如果尝试将数据库的状态改为与当前状态不兼容的模式（例如，当存在其他连接时，尝试把数据库改为 SINGLE_USER 模式），那么 SQL Server 的行为就由所指定的 TERMINATION 选项决定，我稍后会讨论这个选项。

要确定为数据库设置了哪个用户访问值，可以检查 *sys.databases* 目录视图，如下所示：

```
SELECT USER_ACCESS_DESC FROM sys.databases
WHERE name = '<name of database>';
```

该查询会返回 MULTI_USER、SINGLE_USER 或 RESTRICTED_USER 之一。

2. OFFLINE | ONLINE | EMERGENCY

使用 OFFLINE、ONLINE 和 EMERGENCY 选项描述数据库的状态，它们互斥。数据库的默认值是 ONLINE。与其他用户访问选项一样，在使用 *ALTER DATABASE* 将数据库设置为这些模式之一时，不指定 ON 或 OFF 的值，只是使用选项的名称。数据库被设为 OFFLINE 时，它就关闭而且完全关掉，并标记为脱机。数据库脱机时不能被修改。如果在数据库中有任何连接，数据库都不能被设置为 OFFLINE 模

式。SQL Server 是等待其他连接终止还是生成错误信息是由所指定的 TERMINATION 选项决定的。

下列代码样例显示如何将数据库的状态值设置为 OFFLINE，以及如何确定数据库的状态：

```
ALTER DATABASE AdventureWorks2008 SET OFFLINE;
SELECT state_desc from sys.databases
WHERE name = 'AdventureWorks2008';
```

可以将数据库显式设置为 EMERGENCY 模式，这个选项会在第 11 章中和 DBCC 命令一起讨论。

正如前面查询中显示的那样，可以通过检查 *sys.databases* 视图的 *state_desc* 列确定数据库的当前状态。这一列可能会返回除 OFFLINE、ONLINE 和 EMERGENCY 以外的状态值，但是不可以用 *ALTER DATABASE* 直接设置这些值。从备份还原数据库的过程中，数据库能具有状态值 RESTORING。重新启动 SQL Server 时，数据库能具有状态值 RECOVERING。从某时刻在数据库上进行还原过程，直到 SQL Server 完成还原数据库，数据库都具有 RECOVERING 状态。如果出于某种原因无法完成还原过程（最有可能是数据库的一个或多个日志文件不可用或不可读），SQL Server 会让数据库的状态变为 RECOVERY_PENDING。如果 SQL Server 在回滚还原的过程中用完日志或数据空间，或者 SQL Server 在启动过程中用完内存或锁，数据库也可能被设置为 RECOVERY_PENDING 模式。我会在第 4 章更详细地解释回滚恢复与启动恢复之间的区别。

如果所有所需资源（包括日志文件）都可用，但是却在恢复过程中检测到损坏，那么可能会把数据库设置为 SUSPECT 状态。可以通过查看 *sys.dtabases* 视图中的 *state_desc* 列确定状态值。如果数据库是 SUSPECT 状态，那么数据库就完全不可用，运行 *sp_helpdb* 甚至不会列出数据库，但是仍然会在 *sys.dtabases* 视图中看到可疑数据库的状态。很多情况下，可将可疑数据库设置为 EMERGENCY 模式，从而只能对它进行只读操作。如果真的丢失数据库的一个或多个日志文件，在将数据库复制到新位置时，用 EMERGENCY 模式还可以访问数据。从 RECOVERY_PENDING 切换成 EMERGENCY 时，SQL Server 会关闭数据库，然后用特殊标志重新启动它，这样会跳过恢复过程。跳过恢复意味着可能会有逻辑上或物理上不一致的数据——丢失索引行、断开的页面链接或错误的元数据指针。将数据库明确设置为 EMERGENCY 模式就是承认数据可能不一致但是不管怎样还是想访问它。

3. READ_ONLY | READ_WRITE

这些选项描述数据库的可更新性，它们互斥。数据库的默认选项是 READ_WRITE。就像其他的用户访问选项那样，用 *ALTER DATABASE* 将数据库设置为这些模式之一时，不是指定 ON 或 OFF 值，而是使用选项的名称。数据库在 READ_WRITE 模式时，任何具有相应权限的用户都能进行数据修改操作。在 READ_ONLY 模式，不能执行 INSERT（插入）、UPDATE（更新）或 DELETE（删除）操作。而且因为数据库在 READ_ONLY 模式时不能修改，所以当 SQL Server 重新启动时，没有在这个数据库上运行自动更新，而且在任何 *SELECT* 操作过程中也不需要获得锁。不可能在 READ_ONLY 模式收缩数据库。

如果数据库有任何连接，都不能把它放在 READ_ONLY 模式。SQL Server 是等待其他连接终止还是生成错误信息是由所指定的 TERMINATION 选项决定的。

以下代码显示如何将数据库的可更新值设置为 READ_ONLY，以及如何确定数据库的可更新值：

```
ALTER DATABASE AdventureWorks2008 SET READ_ONLY;
SELECT name, is_read_only FROM sys.databases
WHERE name = 'AdventureWorks2008';
```

为数据库启用 READ_ONLY 之后，*is_read_only* 列会返回 1，否则数据库启用 READ_WRITE，返回 0。

终止选项

正如我刚刚讲过的，当数据库正在使用或者正在被不合格的用户使用时，有几个状态选项不能设置。通过在 *ALTER DATABASE* 命令中加上终止选项，可以指定 SQL Server 如何处理这种情况。可以让 SQL Server 等待情况改变、生成错误信息或终止不合格用户的连接。在下面的情况下，终止选项决定 SQL Server 的行为。

- 尝试将数据库改成 SINGLE_USER 时，数据库有多个当前连接。
- 尝试将数据库改成 RESTRICTED_USER 时，不合格的用户正在连接它。
- 尝试将数据库改成 OFFLINE 时，数据库有当前连接。
- 尝试将数据库改成 READ_ONLY 时，数据库有当前连接。

在以上任何一种情况下，SQL Server 的默认行为都是无限等待。下面的 TERMINATION 选项会改变这个行为。

- **ROLLBACK AFTER 整数**（秒）。此选项让 SQL Server 等待指定秒数，然后断开不合格的连接。不完整的事务会回滚。过渡到 SINGLE_USER 模式时，除了发出 *ALTER DATABASE* 命令的那个连接之外，所有的连接都是不合格的。过渡到 RESTRICTED_USER 模式时，不合格的连接是那些非 *db_owner* 固定数据库角色或非 *dbcreator* 和 *sysadmin* 固定服务器角色的用户。
- **ROLLBACK IMMEDIATE。** 这个选项立即断开不合格的连接。所有不完整的事务都会回滚。记住，虽然可能立即断开了连接，但是回滚可能需要一段时间才能完成。所有由事务完成的工作必须回到未完成的状态，因此对于某些操作（如批量更新几百万行或重新编译大索引）来说，可能需要漫长的等待。不合格的连接与之前描述的相同。
- **NO_WAIT。** 这个选项让 SQL Server 在尝试改变数据库状态之前检查连接，如果存在特定的一些连接，则 *ALTER DATABASE* 命令失败。如果数据库被设置为 SINGLE_USER 模式，存在任何其他的连接，*ALTER DATABASE* 命令都会失败。如果过渡到 RESTRICTED_USER 模式且存在任何不合格的连接，*ALTER DATABASE* 命令都会失败。

以下命令将 *AdventureWorks2008* 数据库的用户访问选项改成 SINGLE_USER。存在到 *AdventureWorks2008* 数据库的任何连接，都会产生错误。

```
ALTER DATABASE AdventureWorks2008 SET SINGLE_USER WITH NO_WAIT;
```

3.9.2 游标选项

游标选项控制服务器端游标的行为，这些游标是用下列定义和控制游标的 T-SQL 命令之一来定义的：*DECLARE*、*OPEN*、*FETCH*、*CLOSE* 和 *DEALLOCATE*。

- **CURSOR_CLOSE_ON_COMMIT {ON|OFF}。** 如果这个选项设置为 ON，在提交事务或回滚时，会关闭任何打开的游标（与 SQL-92 兼容）。如果指定了 OFF（默认的），游标会在提交事务后继续打开。除非将游标定义为 *INSENSITIVE* 或 *STATIC*，否则回滚事务会关闭任何游标。
- **CURSOR_DEFAULT {LOCAL|GLOBAL}。** 如果将这个选项设置为 LOCAL（本地）而且创建游标时没有将它指定为 GLOBAL，那么任何游标的范围对于所在的批、存储过程或触发器来说都是本地的。游标的名称只在这个范围内有效。批、存储过程、触发器或存储过程输出参数中的本地游标变量可以引用游标。将这个选项设置为 GLOBAL 而且创建游标时没有将它指定为 LOCAL，那么游标的范围对于连接就是全局的。可以在连接执行的任何存储过程或批中引用游

标名称。

3.9.3 自动选项

自动选项影响 SQL Server 可能会自动进行的操作。所有这些选项都是布尔选项，值为 ON 或 OFF。

- **AUTO_CLOSE。** 这个选项设置为 ON 时，数据库就在最后一个用户退出后关闭和完全关掉，因而会冻结所有资源。所有文件句柄被关闭、删除内存中的所有结构，这样数据库就不会使用任何内存。当用户尝试再次使用数据库时，数据库就会重新打开。如果没有完全关掉数据库，那么直到下次 SQL Server 重新启动后用户尝试使用数据库时，数据库才会初始化（重新打开）。因为用 AUTO_CLOSE 选项可以像管理正常文件那样管理数据库文件，所以它对个人 SQL Server 数据库来说很方便。可以移动数据库、复制数据库做备份甚至将数据库通过电子邮件发给其他用户，但是不应该为那些被反复连接和断开与 SQL Server 的应用程序所访问的数据库使用这个选项。为每个连接关闭和重新打开数据库的开销可能会损害性能。
- **AUTO_SHRINK。** 这个选项设置为 ON 时，所有的数据库文件都可能会被定期收缩。SQL Server 可以自动收缩数据文件和日志文件。释放日志文件中的空间进而能收缩它们以便备份的唯一方法是备份事务日志或者将还原模型设置为 SIMPLE。日志文件会在备份或截断日志时收缩，从不推荐使用这个选项。
- **AUTO_CREATE_STATISTICS。** 这个选项设置为 ON 时（默认），SQL Server 查询优化器会在查询的 WHERE 或 ON 语句中创建列的统计数字。添加统计数字改善了查询性能，因为 SQL Server 查询优化器能更好地确定如何估计查询。
- **AUTO_UPDATE_STATISTICS。** 这个选项设置为 ON 时（默认），如果表中的数据已经改变，那么现有的统计数字也会更新。SQL Server 保留对表做出修改的计数器，并用它决定统计数字过期的时间。将这个选项设置为 OFF 时，不会自动更新现有的统计数据（可以手动更新）。在第 6 章和第 8 章中会更详细地讨论统计信息。

3.9.4 SQL 选项

SQL 选项控制如何解释各种 SQL 语句，它们全部都是布尔选项。对于 SQL Server 来说，这些选项的默认值都是 OFF，但是许多工具（如 Management Studio）和许多编程界面（如 ODBC）都启用某种会话级别的选项，这些选项会覆盖数据库选项，看起来好像默认就是 ON 行为一样。

- **ANSI_NULL_DEFAULT。** 这个选项设置为 ON 时，列遵循 ANSI_SQL-92 规则中的列可为空性。即如果没有明确说明表中的列是否允许 NULL 值，就会允许为 NULL。这个选项设置为 OFF 时，如果没有指定可为空的约束，新创建的列就不允许为 NULL。
- **ANSI_NULLS。** 正如 ANSI-92 标准所说明的那样，这个选项设置为 ON 时，任何与 NULL 值的比较都会得到 UNKNOWN。如果该选项设置为 OFF，如果两个被比较的值都是 NULL，那么非 Unicode 值与 NULL 的比较就会得到 TRUE 值。
- **ANSI_PADDING。** 这个选项设置为 ON 时，被互相比较的字符串会在比较之前被设置为相同的长度。如果选项设置为 OFF，就不会自动补上。
- **ANSI_WARNINGS。** 这个选项设置为 ON 时，像除以 0 或数字溢出这样的情况发生时，会发出错误或警告。
- **ARITHABORT。** 这个选项设置为 ON 时，在执行查询的过程中，发生数字溢出或除以 0 错误

时，会终止查询。如果选项设置为 OFF，查询会返回 NULL 作为操作结果。

- **CONCAT_NULL_YIELDS_NULL。** 这个选项设置为 ON 时，如果任意一个字符串为空，连接两个字符串会得到 NULL 字符串。如果选项设置为 OFF，为了连接字符串，NULL 字符串会被作为空（长度为 0）字符串来对待。
- **NUMERIC_ROUNDABORT。** 这个选项设置为 ON 时，如果表达式会导致精度降低，就会产生错误。如果选项设置为 OFF，会将结果四舍五入。ARITHABORT 的设置决定错误的严重性。如果 ARITHABORT 为 OFF，那么只会发出警告，表达式会返回 NULL。如果 ARITHABORT 为 ON，会产生错误，不会返回结果。
- **QUOTED_IDENTIFIER。** 这个选项设置为 ON 时，可以通过双引号来分隔标识符（如表和列名称），用单引号来分隔叙述文字。所有用双引号分隔的字符串都被理解为对象标识符。如果 QUOTED_IDENTIFIER 是 ON，引号内的标识符不必遵守标识符的 T-SQL 原则。它们可以是关键字，也可以包括一般在 T-SQL 标识符中不允许的字符，如空格和破折号。不能用双引号分隔叙述字符串表达式，必须用单引号。如果单引号是叙述字符串的一部分，可以用两个单引号（”）表示。如果数据库中的对象名称使用了保留的关键字，就必须把这个选项设置为 ON。如果选项设置为 OFF，不能在引号中出现标识符，而且必须遵守标识符的所有 T-SQL 规则。
- **RECURSIVE_TRIGGERS。** 这个选项设置为 ON 时，可以直接或间接地递归触发触发器。触发器触发并进行操作让另一个表上的触发器触发，从而导致原始表格更新，让原始的触发器再次触发，这样就会发生间接递归。例如，一个应用程序更新表 *T1*，这会让触发器 *Trig1* 触发，*Trig1* 更新表 *T2*，又导致触发器 *Trig2* 触发。之后 *Trig2* 更新表 *T1*，这会让 *Trig1* 再次触发。触发器触发并进行让同一个触发器再次触发的操作，会发生直接递归。例如，应用程序更新表 *T3*，这会导致触发器 *Trig3* 触发。*Trig3* 再次更新表 *T3*，这会导致触发器 *Trig3* 再次触发。如果选项设置为 OFF（默认），触发器就不能递归触发。

3.9.5 数据库恢复选项

数据库还原选项 RECOVERY（FULL、BULK_LOGGED 或 SIMPLE）决定在 SQL Server 数据库上所恢复的量，它还控制记录多少信息量，以及有多少日志可用于备份。我会在第 4 章更详细地讲述这个选项。

恢复数据库时，也可以应用另外两个选项。在 SQL Server 2008 中将 TORN_PAGE_DETECTION 选项设置为 ON 或 OFF 是可能的，但这个特殊选项不会出现在将来的版本中。推荐的做法是将 PAGE_VERIFY 选项设置为 TORN_PAGE_DETECTION 或 CHECKSUM（所以现在 TORN_PAGE_DETECTION 应该被看成是一个值，而不是选项的名称）。

PAGE_VERIFY 选项能发现由磁盘 I/O 路径错误造成的、损坏的数据库页面，这可能会导致数据库损坏问题。I/O 错误本身通常是在页面写入到磁盘时，由断电或磁盘损坏引起的。

- **CHECKSUM。** PAGE_VERIFY 选项设置为 CHECKSUM 时，SQL Server 会检查每个页面内容的校验和，并在页面写入到磁盘时，在页面头文件中存储值。从磁盘中读取页面时，重新计算校验和，并与页面头文件中存储的值相比较。如果值不相符，就会报告错误信息 824（表示校验和失败）。
- **TORN_PAGE_DETECTION。** PAGE_VERIFY 选项设置为 TORN_PAGE_DETECTION 时，无论页面何时写入到磁盘，都会使数据库页面（8KB）中的每个 512 字节扇区中的一位进行翻转。

这样 SQL Server 就可以检测由断电或其他系统中断导致的不完整 I/O 操作。如果后来 SQL Server 读页面时，有一位是错误的状态，就表示没有正确写入页面（检测到破损页）。虽然 SQL Server 数据库页面是 8KB，但是磁盘用 512 字节的扇区进行 I/O 操作，因此每个数据库页面上写 16 个扇区。如果在操作系统将第一个 512 字节扇区写到磁盘和完成 8KB I/O 操作之间，系统崩溃（例如由于断电），就可能出现破损页。从磁盘读出页面时，会将存储在页面头文件中的破损位与实际的页面扇区信息相比较。如果值不符合，表示只有页面的一部分写入到磁盘。在这种情况下，会报告错误信息 824（表示破损页错误）。如果真的是不完全写入页面，通常数据库还原会检测到破损页。但是，其他的 I/O 路径失败随时都可能会导致破损页。

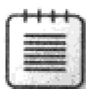
- **NONE（无页面检验选项）**。可以指定写页面时，不生成 CHECKSUM 或 TORN_PAGE_DETECTION 的值，读页面时也不会检验这些值。

校验和和破损页错误都生成错误 824，这会写入 SQL Server 错误日志和 Windows 事件日志中。对于读时生成 824 错误的任何页面来说，SQL Server 都会向 *msdb* 数据库的系统表 *suspect_pages* 中插入一行（*SQL Server 联机丛书*有更多关于“理解和管理 suspect_pages 表”的信息）。

SQL Server 会重新尝试校验和、破损页或其他 I/O 错误 4 次。如果在任何一次尝试中成功读取，就会在错误日志中写入消息，触发读的命令会继续。如果尝试失败，命令就会以错误信息 824 告终。

如果失败仅限于索引页面，可以通过还原数据或重新编译索引来“解决”错误。如果遇到校验和错误，可以运行 *DBCC CHECKDB* 确定受影响的数据库页面的类型。也应该确定错误的根本原因，尽可能快地解决问题，以防止其他或当前的错误。找出根源需要研究硬件、固件驱动器、BIOS、过滤器驱动器（如病毒软件）和其他的 I/O 路径组件。

在 SQL Server 2008 和 SQL Server 2005 中，默认是 CHECKSUM。在 SQL Server 2000 中，默认是 TORN_PAGE_DETECTION，CHECKSUM 不可用。如果从 SQL Server 2000 升级数据库，那么 PAGE_VERIFY 的值就会是 NONE 或 TORN_PAGE_DETECTION。应该总是考虑使用 CHECKSUM。虽然 TORN_PAGE_DETECTION 使用的资源更少，但是它比 CHECKSUM 提供的保护也更少。记住，如果对从 SQL Server 2000 更新的数据库启用了 CHECKSUM，那么校验和的值只是在修改的页面上计算。



注意：

在 SQL Server 2008 以前，*tempdb* 数据库中 CHECKSUM 和 TORN_PAGE_DETECTION 都不可用。

3.9.6 其他数据库选项

后面几章还会讲其他数据库选项。快照隔离选项将在第 10 章讨论，改变跟踪选项已经在第 2 章讲过了，其他的则不在本书讨论范围内。

3.10 数据库快照

添加到 SQL Server 2005 企业版产品中的一个有趣功能是数据库快照，可以用它创建数据库的即时只读副本。实际上，可以在不同时间为相同的源数据库创建多个快照。每个快照所需的实际空间通常比实际数据库所需的空间少很多，因为快照只保存已经改变的页面，这将在稍后讨论。

数据库快照可以做以下事情。

- 将数据库镜像到汇报服务器（不能从数据库镜像读取，但是可以从镜像创建快照再读取）。
- 生成报告，不会阻挡生产运行，也不会被生产运行所阻挡。
- 防止管理或用户错误。

随着使用快照的经验增多，您可能会想出更多使用快照的方法。

3.10.1 创建数据库快照

创建快照的架构很直接，为 *CREATE DATABASE* 命令指定一个选项即可。没有通过“对象资源管理器”创建数据库的图形界面，所以必须使用 T-SQL 语句。创建快照时，必须在 *CREATE DATABASE* 命令中包括源数据库的每个数据文件、原始逻辑名称、新物理名称和路径。不能指定文件的其他属性，也不使用日志文件。

下面是创建 *AdventureWorks2008* 数据库快照的语法，把快照文件放在 SQL Server 2008 默认数据目录中：

```
CREATE DATABASE AdventureWorks_snapshot ON
( NAME = N'AdventureWorks_Data',
  FILENAME =
 N'C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL\
   Data\AW_data_snapshot.mdf')
AS SNAPSHOT OF AdventureWorks2008;
```

快照中的每个文件都被创建为稀疏文件，这是 NTFS 文件系统的功能（不要将稀疏文件与 SQL Server 2008 中的稀疏列相混淆）。最初，稀疏文件不包括用户数据，用户数据的磁盘空间没有分配给稀疏文件。随着数据写入到稀疏文件中，NTFS 逐渐分配磁盘空间。稀疏文件可能会增长到很大。稀疏文件是以 64KB 为增量增长的，所以磁盘上的稀疏文件大小总是 64KB 的倍数。

快照文件只包括修改过的源数据。SQL Server 为每个文件创建位图并将它保存在缓存中，文件的每个页都有一位，表示该页是否已经复制到快照中。每次源中的页面更新后，SQL Server 都会检查文件的位图，看是否已经复制了页面。如果还没有复制，就会在这时进行复制。这个操作称为*写时复制*（copy-on-write）操作。图 3-4 显示了包括 10%源数据（一页）的快照。

进程在读取快照时，它首先访问位图，看想要的页面是在快照文件中还是仍在源中。图 3-5 显示与图 3-4 相同的数据库读操作。可以从源数据库访问其中的 9 页，其中 1 个可以从快照访问，因为已经在源上更新它了。从快照数据库读过程时，无论隔离级别是多少，都不会上锁。不管是从稀疏文件还是从源数据库中读页面，都是这样。这是使用数据库快照的一大优点。

如前所述，位图存储在缓存中，而不是和文件存在一起，所以总是即时可用的。SQL Server 关闭或数据库关闭时，位图就会丢失，需要在数据库启动时重新构建位图。SQL Server 决定每页是否在稀疏文件中，然后它会记录位图中的信息，以便将来使用。

快照会反映发出 *CREATE DATABASE* 命令的时刻，即创建操作发生的时候。SQL Server 检查源数据库并记录源数据库的事务日志中的同步日志序列号（LSN）。在第 4 章讲事务日志时，就会看到 LSN 是确定数据库中特定点的方法。然后 SQL Server 在源数据库上运行还原，这样任何未提交的事务都能在快照中回滚。所以，虽然快照的稀疏文件开始是空的，但是它可能不会很久都是空的。如果在创建快照时，事务正在进行，那么还原过程会在可以使用快照数据库之前撤销未提交的事务，所以快照会有包含修改数据的源中的任何页面的原始版本。

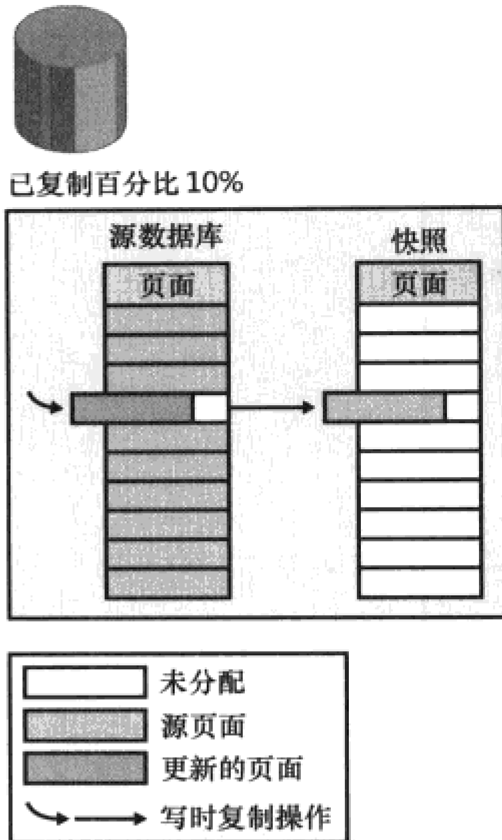


图 3-4 包含源数据库中一个页面的数据库快照

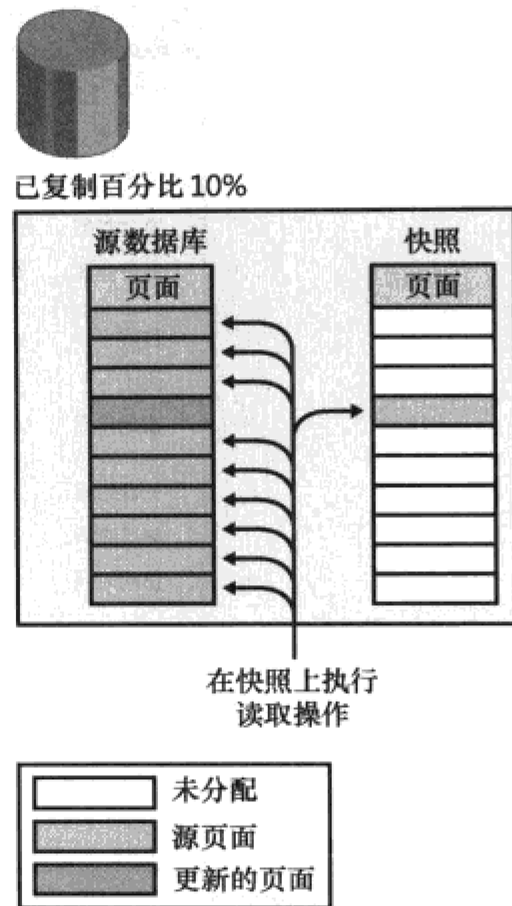


图 3-5 从数据库快照中读操作，从快照读出改变的页面，从源数据库中读出未改变的页面

因为 NTFS 卷是唯一支持稀疏文件技术的卷，所以只能在它上面创建快照。如果尝试在 FAT 或 FAT32 卷上创建快照，就会得到类似的错误：

```
Msg 1823, Level 16, State 2, Line 1
A database snapshot cannot be created because it failed to start.

Msg 5119, Level 16, State 1, Line 1
Cannot make the file "E:\AW_snapshot.MDF" a sparse file. Make sure the file system supports
sparse files.
```

第一个错误基本上是常规的失败消息，第二个消息给出了操作为什么失败的更多详细信息。

3.10.2 数据库快照使用的空间

可以通过查看动态管理函数 *sys.dm_io_virtual_file_stats*（它在 *size_on_disk_bytes* 列返回文件中的当前字节数）找出快照的每个稀疏文件正在使用的字节数。这个函数使用参数 *database_id* 和 *file_id*。快照数据库的数据库 ID 和每个稀疏文件的文件 ID 显示在 *sys.master_files* 目录视图中。也可以右键单击文件名并查看属性，在 Windows 资源管理器中查看大小，如图 3-6 所示。Size 的值是最大值，磁盘上的大小应该和使用 *sys.dm_io_virtual_file_stats* 所看到的相同。创建快照时，最大大小应该与源数据库的大小差不多。

因为相同的数据库可能有多个快照，所以需确保有足够可用的磁盘空间。快照开始时较小，但是随着源数据库的更新，每个快照都会增长。稀疏文件以片段来分配，称为*区域*，它的单位是 64KB。分配了区域以后，除了已经改变的那个页面之外，所有的页面都会清零。在相同的区域内还有 7 个可以改变的页面，直到用完 7 个页面才会分配新区域。

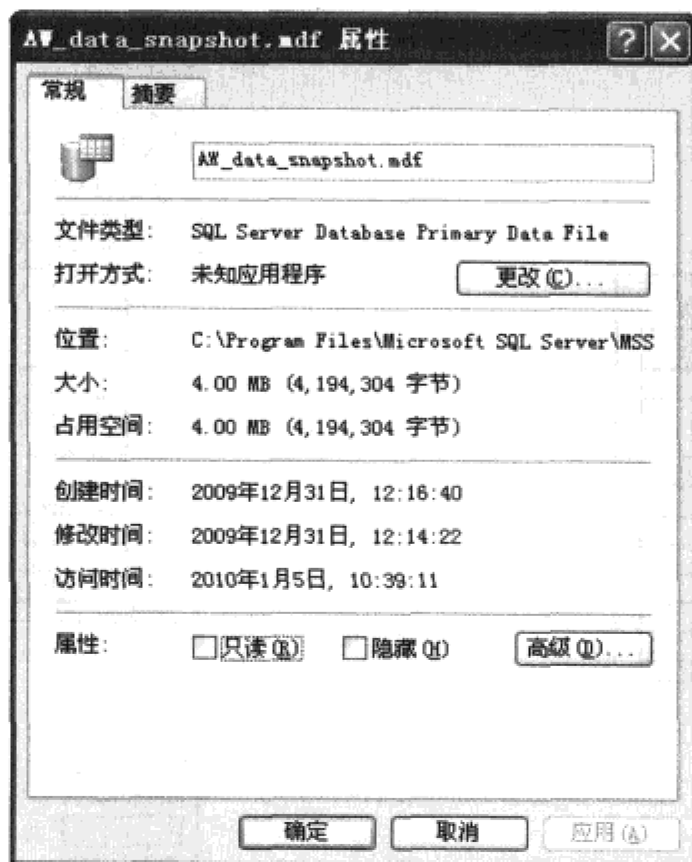


图 3-6 Windows 资源管理器中快照文件的“属性”对话框，显示稀疏文件的当前大小和在磁盘上的大小

有可能会过度提交存储，即在正常情况下，快照可能会比物理上能存储的量多许多倍，但是如果快照增长，可能会用完物理卷上的空间（注意，运行联机 *DBCC CHECKDB* 和运行在处理过程中使用隐藏快照的相关命令时，可能会发生这种事。无法控制命令所使用的隐藏快照的放置，它们被放在与父数据库相同的卷上。如果这样的事发生，DBCC 就会使用源数据库，获得表锁。可以在第 11 章阅读 DBCC 命令内部的更多细节）。一旦物理卷的空间用完，写入源的操作就不会将“之前”的图像复制到稀疏文件中。不能写入页面的快照被标记为可疑和不可使用的，但是源数据库会继续正常工作。没办法“解决”可疑快照，必须删除快照数据库。

3.10.3 管理快照

如果源数据库上存在任何快照，就不能删除、分离或还原源数据库。此外，基本上可以通过将源数据库还原为制作快照时候的样子，用快照之一代替源数据库。使用 *RESTORE* 命令进行此操作：

```
RESTORE DATABASE AdventureWorks2008
FROM DATABASE_SNAPSHOT = 'AdventureWorks_snapshot';
```

在还原操作时，快照和源数据库都不可用，且被标记为“在还原中”。如果在还原操作过程中出现错误，在数据库重新启动时，操作就尝试完成还原。如果存在多个快照，就不能还原快照，所以应该先删除想还原那个之外的所有快照。删除快照就像使用任何其他 *DROP DATABASE* 操作一样。删除快照时，也会删除所有的 NTFS 稀疏文件。

记住关于数据库快照的其他注意事项。

- 不能为 *model*、*master* 或 *tempdb* 数据库创建快照（可以在内部创建快照，从而在 *master* 数据库上运行联机 DBCC 检查，但是无法明确创建它们）。

- 快照继承它源数据库的安全约束，因为它是只读的，所以无法改变权限。
- 如果从源数据库删除用户，该用户仍会在快照中。
- 不能备份或还原快照，但是可以备份源数据库，这不受数据库快照的影响。
- 不能附加或分离快照。
- 数据库快照不支持全文本索引，不会从源数据库传送全文本目录。

3.11 tempdb 数据库

在一些情况下，*tempdb* 数据库就像是其他所有数据库一样，但是它有一些独特的行为。不是所有的 *tempdb* 数据库都与本章的话题有关，所以我会给出对其他章的引用，在那里可以找到其他信息。

正如之前所提到的，*tempdb* 数据库和 SQL Server 实例中的所有其他数据库之间最大的区别是每次重新启动 SQL Server 时，重新创建而不是还原 *tempdb*。可以把 *tempdb* 想成是 SQL Server 本身为临时用户对象和内部对象明确创建的工作空间。

每次重新创建 *tempdb* 时，它会从 model 数据库继承大多数数据库选项，但是不会复制还原模型，因为 *tempdb* 总是使用简单还原，这将在第 4 章详细讨论。不能为 *tempdb* 设置特定的数据库选项，如 OFFLINE 和 READONLY。也不能删除 *tempdb* 数据库。

在 SIMPLE 恢复模型中，会一直截断 *tempdb* 数据库的日志，永远不会备份它。不需要恢复信息是因为每次启动 SQL Server 时，都会完全重建 *tempdb*，用户之前创建的任何临时对象（即所有的表和数据）都会消失。

为 *tempdb* 记录日志也和其他的数据库不同（第 4 章将会讨论正常的日志记录）。许多人假设 *tempdb* 中没有日志记录，这是错的。*tempdb* 中的操作会被记录下来，这样才会回滚临时对象上的事务，但是日志中的记录只包括用于回滚事务的信息，不会恢复（或重复操作）它。

以前我提到过，运行在数据库上的恢复是创建快照的首要步骤之一。因为无法还原 *tempdb*，所以无法为它创建快照，这意味着无法用快照运行 *DBCC CHECKDB*（或者实际上大多数 DBCC 校验命令）。在 *tempdb* 中运行 DBCC 的另一个区别是 SQL Server 会跳过所有的分配和目录检查。在 *tempdb* 中运行 *DBCC CHECKDB*（或 *CHECKTABLE*）需要在选择的每个表上加“共享表”锁（锁将在第 11 章讨论）。

3.11.1 tempdb 中的对象

存储在 *tempdb* 中的 3 种对象是：用户对象、内部对象和主要为快照隔离使用的版本存储。

1. 用户对象

所有用户都有创建和使用 *tempdb* 中本地和全局临时表的权利（本地和全局表的名称分别具有#或##前缀，但在默认情况下，除非表名的前缀是#或##，否则用户没有使用 *tempdb* 并在其中创建表的权利）。但是可以轻易地在每次重启 SQL Server 时自动启动的过程中授予权利。

其他在 *tempdb* 中需要空间的用户对象包括表变量和表值函数。很多时候对待 *tempdb* 中创建的用户对象就像对待任何其他数据库中的用户对象一样。生成 *tempdb* 中创建的用户对象时，必须为它们分配空间，也需要管理元数据。可以通过检查系统目录视图（如 *sys.objects*）查看用户对象，也可以使用 *sys.partitions* 和 *sys.allocation_unit* 中的信息查看用户对象占用多少空间。我会在第 5 章和第 7 章讨论这些视图。

2. 内部对象

使用一般的工具无法查看 *tempdb* 中的内部对象，但是它们仍然占用数据库的空间。因为只在内存中存储它们的元数据，所以没有在目录视图中列出它们。内部对象的 3 个基本类型是工作表、工作文件和排序单元。

工作表是在下列操作过程中由 SQL Server 创建的。

- 后台处理，在大量查询时保存中间结果。
- 运行 *DBCC CHECKDB* 或 *DBCC CHECKTABLE*。
- 处理 XML 或 *varchar（MAX）*变量。
- 处理 SQL Service Broker 对象。
- 处理静态或键集游标。

当 SQL Server 处理使用哈希操作符的查询时，用工作文件合并或聚合数据。

进行排序操作时，会创建排序单位，这在包含 ORDER BY 语句的查询之外的很多情况下也会发生。SQL Server 使用排序建立索引，也可能会使用排序来处理涉及查询的分组。特定类型的加入可能需要 SQL Server 在进行加入之前排序数据。排序数据时，在 *tempdb* 中创建排序单元以保存数据。除了在 *tempdb* 中，SQL Server 也可以在用户数据库中（尤其是创建索引时）创建排序单元。就像在第 6 章中看到的那样，创建索引时，可以选择在当前用户数据库或者在 *tempdb* 中进行排序。

3. 版本存储

版本存储支持数据的行级版本划分技术。在以下情况下，旧版本的更新行保存在 *tempdb* 中。

- 触发 AFTER 触发器后。
- 在允许快照事务的数据库中执行数据修改语言（DML）命令后。
- 从客户端应用程序调用多重活动结果集（MARS）后。
- 索引上有并发的 DML 时，在联机索引编译或重新编译的过程中。

版本划分和快照事务会在第 10 章详细讨论。

3.11.2 tempdb 中的优化

因为 SQL Server 2008 中许多内部操作都使用 *tempdb*，这比以前的版本多，所以要注意监视和管理它的问题。下节给出最佳实践和监视建议。本节我会讲一些 SQL Server 中的内部优化，让 *tempdb* 更有效地管理对象。

1. 日志优化

大家知道，影响用户数据库的任何操作都会被记录下来。但是在 *tempdb* 中，并不完全是这样。例如，使用日志更新操作只会记录原始数据（之前的图像），而不会记录新值（之后的图像）。而且，正如其他数据库中承诺的操作和提交的日志记录不会同步刷新到磁盘，*tempdb* 中承诺的操作和提交的日志记录也不会同步刷新到磁盘。

2. 分配和缓存优化

所有的数据库都使用许多分配优化，而不只是 *tempdb* 使用。但是 *tempdb* 最有可能是在生产操作中

创建和删除新对象数量最多的数据库，所以优化对 *tempdb* 的影响比在用户数据库上的影响更大。在 SQL Server 2008 中，高效地访问分配页面能确定哪里有可用的可用扩展。您应该会看到分配页面中的争用情况比以前版本少得多。SQL Server 2008 具有非常有效的搜索算法，可以从混合扩展中找出一个可用页。当数据库有多个文件时，SQL Server 2008 具有高效的比例填充算法，可将空间分配给多个数据文件，与每个文件中的可用空间成比例。

tempdb 特有的另一个优化可以免去为一些对象分配任何新空间。如果删除工作表，会保存一个 IAM 页和一个扩展（共 9 页），因此，如果要重新创建相同的工作表，则无需先解除分配再重新分配空间。这个被删除的工作表缓存不是很大，只能容纳 64 个对象。如果从内部截断工作表且使用工作表的查询计划仍在计划缓存中，那么还是会保存第一个 IAM 页面和第一个扩展。对于截断的表来说，被缓存的对象数量没有具体限制，仅取决于可用的内存空间。

如果删除 *tempdb* 中的用户对象，也可以缓存这些对象的一些空间。对于小于 8MB 的小表来说，在 *tempdb* 中删除用户对象会保存一个 IAM 页和一个扩展。但是如果表进行了任何其他的 DDL（如创建索引或约束）或者用动态 SQL 创建表格，就不会进行缓存。

对于大表来说，整个删除都会作为延迟操作进行。实际上，在每个数据库中都用延迟的删除操作作为改善整体吞吐量的一种方式，因为线程不需要在处理下一项任务之前等待删除完成。就像所有数据库中其他可用的分配优化一样，延迟删除可能会给 *tempdb* 带来最大的好处，*tempdb* 是在生产工序过程中最有可能删除表的地方。背景线程最终会清除为被删除的表所分配的空间，但是在这之前会一直保留被分配的空间。可以用 *type* 的值 0（表示是被删除的对象），通过 *sys.allocation_units* 系统视图查看行来检测这个空间，还会看到名为 *container_id* 的列是 0（表示被分配的空间不属于任何对象）。我会在第 5 章讲解 *sys.allocation_units* 和其他记录空间使用的系统视图。

3.11.3 最佳实践

默认情况下，只在一个数据文件上创建 *tempdb* 数据库。您可能会发现多个文件的 I/O 性能更好，在全局分配结构（GAM、SGAM 和 PFS 页面）上的争用也更少。每个 CPU 都有一个文件，但是根据您的数据和使用情况进行的测试，系统可能会提示显示需要更多或更少的文件。为了获得使用成比例填充算法的最大效率，文件都应该大小相同。多个文件的缺点是每个对象都有多个 IAM 页面，访问对象时的切换成本更高，仅管理这些文件就需要更多的努力。不管有多少文件，都应该把它们放在能够买得起的最快速磁盘上。一个日志文件应该就足够用了，也应该把它放在快速磁盘上。

为了确定 *tempdb* 的最优大小，必须用数据卷先测试自己的应用程序，但是知道何时以及如何使用 *tempdb* 也能帮您做出初步估计。记住每个 SQL Server 实例只有一个 *tempdb*，所以有问题的应用程序会影响到所有其他应用程序中的所有其他用户。在第 10 章中，我会解释如何确定版本存储的大小。所有这些因素都影响 *tempdb* 所需的空间。最后，在第 11 章中，我会讲解 DBCC 一致性检查命令是如何使用 *tempdb* 和如何确定 *tempdb* 空间需要的。

尽量不要修改 *tempdb* 的数据库选项，一些选项不能应用于 *tempdb*。尤其是自动收缩选项在 *tempdb* 中不起作用。在任何情况下，除非已经大大地改变了工作负荷的模式，否则都不推荐收缩 *tempdb*。如果需要收缩 *tempdb*，最好单独收缩每个文件。记住，如果需要移动任何内部对象或版本存储页面，可能不能收缩文件。收缩 *tempdb* 的最好方法是 ALTER 数据库、改变文件大小，然后停止并重新启动 SQL Server，从而将 *tempdb* 重新编译成想要的大小。只能让 *tempdb* 文件自动增长作为最后的手段，而且只能防止因没有空间引起的错误。不应该依靠自动增长来管理 *tempdb* 文件的大小。虽然在能够承受自动增长时，自

动增长造成的影响会比使用立即文件初始化小一些，但是会造成处理延迟。应通过测试和计划确定 *tempdb* 的大小，这样 *tempdb* 就可以在开始的时候具有它所需要的空间，而无需在应用程序运行时增长。

下面是最优化使用 *tempdb* 的一些技巧，后面几章会详细解释为什么这些建议是最佳实践。

- 利用 *tempdb* 对象缓存。
- 保持简短的事务，尤其是那些使用快照隔离、*MARS* 或触发器的事务。
- 如果预计会有大量的分配页争用，强制使用 *tempdb* 少的查询计划。
- 通过让列以固定大小而不是使大小可变（可以在 INSERT 后面加上 *UPDATE*，作为 *DELETE* 来执行）更新来避免页分配和释放。
- 如果使用版本控制，不要混合来自不同数据库（在同一实例中）的长短操作。

3.11.4 tempdb 空间监视

正如在第 5 章和第 7 章中讨论的，很多工具、存储过程、系统视图都会报告对象空间的使用情况。但是，系统视图的一个集合只报告 *tempdb* 的信息。最简单的视图是 *sys.dm_db_file_space_usage*，它返回 *tempdb* 中每个数据文件的一行。它返回以下列：

- *database_id*（虽然只使用 *DBID 2*）；
- *file_id*；
- *unallocated_extent_page_count*；
- *version_store_reserved_page_count*；
- *user_object_reserved_page_count*；
- *internal_object_reserved_page_count*；
- *mixed_extent_page_count*。

这些列显示 3 种存储类型（用户对象、内部对象和版本存储）如何使用 *tempdb* 中的空间。

另外，两个系统视图是相似的。

- *sys.dm_db_task_space_usage*。这个视图返回每个活动任务的一行，显示任务已经为用户对象和内部对象分配和释放的空间。如果会话没有运行任何任务，这个视图仍然会为会话显示一行，其所有的空间值显示为 0。因为该空间没有与任何特定任务或会话相关联，所以不会报告版本存储信息。对于所有的空间分配和释放值来说，每个正在运行的任务都是从 0 开始的。
- *sys.dm_db_session_space_usage*。这个视图为每个会话返回一行，其中包括会话对所有已经完成的任务为用户对象和内部对象分配和释放空间的累计值。通常，分配的空间值应该与释放的空间值相同，但是如果存在延迟删除操作，分配的值就会比释放的值大。记住，并不是所有用户都可以看到这个信息，需要从这个视图中选择特殊的权限 *VIEW SERVER STATE*。

3.12 数据库安全性

安全性是影响每个 SQL Server 用户（包括管理员和开发人员）的每个操作的重大话题，它本身就够写本书了。但是 SQL Server 安全性架构的一些方面对于理解如何处理数据库或 SQL Server 数据库中的任何对象是非常重要的，所以我在本书中还得稍微讲一下这个话题。

SQL Server 管理实体的架构。这些实体中最显著的是服务器和服务器中的数据库。在数据库级别下面是对象。服务器级别下面的每个实体都是由个体或个体组所拥有。SQL Server 安全性框架控制对 SQL

Server 实例内部实体的访问。就像任何资源管理器那样，SQL Server 安全性模型有两个部分：身份验证和授权。

*身份验证*是 SQL Server 验证和建立想访问资源的个人身份的过程。身份验证是 SQL Server 决定是否允许给某人访问资源的过程。

我在本节中会讨论数据库访问的基本问题，然后描述存储数据库访问信息的元数据。我还会讲架构的概念，说明如何使用架构访问对象。

以下两个名词构成了说明 SQL Server 2008 中安全性控制的基础。

- **安全性对象**。*安全性对象*是可以被授予权限的实体。安全性对象包括数据库、架构和对象。
- **主体**。*主体*是可以访问安全性对象的实体。*主要主体*代表单一用户（如 SQL Server 登录或 Windows 登录），*次要主体*代表多个用户（如一个角色或 Windows 组）。

3.12.1 数据库访问

在 SQL Server 中，身份验证在两个不同的级别进行。首先，必须在服务器级对想访问任何 SQL Server 资源的任何人进行身份验证。SQL Server 2008 安全性提供两种验证登录身份的基本方法：Windows 身份验证和 SQL Server 身份验证。在 Windows 身份验证中，SQL Server 登录安全性直接与 Windows 安全性集成，让操作系统可以验证 SQL Server 用户的身份。在 SQL Server 身份验证中，管理员在 SQL Server 中创建 SQL Server 登录账户，任何连接到 SQL Server 的用户必须申请有效的 SQL Server 登录名和密码。

Window 身份验证使用*可信任连接*，它依赖于 Windows 的模拟功能。通过模拟，SQL Server 可以拿到初始化连接的那个 Windows 用户账户的安全性内容，并测试该 SID 是否具有有效的权限级别。连接到 SQL Server 时，任何可用的网络库都支持 Windows 模拟和可信任连接。

在 Windows Server 2003 和 Windows Server 2008 中，SQL Server 可以使用 Kerberos 支持客户端和服务器之间的相互身份验证，还能在计算机之间传递客户端的安全性凭据，从而可以使用模拟客户端的凭据处理远程服务器上的工作。Windows Server 2003 和 Windows Server 2008 用 Kerberos 和委派来支持 Windows 身份验证和 SQL Server 身份验证。

SQL Server 的身份验证方法（或方法集）是由其安全性模式决定的。SQL Server 能以两种安全性模式运行：Windows 身份验证模式（只使用 Windows 身份验证）和混合模式（客户端可以选择使用 Windows 身份验证或 SQL Server 身份验证）。连接到为 Windows 身份验证模式配置的 SQL Server 实例时，不能提供 SQL Server 登录名称，Windows 用户名决定访问 SQL Server 的级别。

Windows 身份验证的一个优点是总是允许 SQL Server 利用操作系统的安全性功能，如密码加密、密码老化及密码的最小和最大长度限制。在 Windows Server 2003 或 Windows Server 2008 上运行时，SQL Server 身份验证也能利用 Windows 密码策略。关于详细信息，请参阅 *SQL Server 联机丛书*中的 *ALTER LOGIN* 命令。还要注意的是，如果在安装过程中选择 Windows 身份验证，就会禁用默认的 SQL Server *sa* 登录。如果在安装后切换到混合模式，就能用 *ALTER LOGIN* 命令启用 *sa* 登录。可以在 Management Studio 中右键单击服务器名称，选择“属性”，然后选择“安全性”页面，改变身份验证模式。在“服务器身份验证”下面，选择新服务器身份验证模式，如图 3-7 所示。

在混合模式下，基于 Windows 的客户端可以用 Windows 身份验证连接，非 Windows 客户端或者通过 Internet 的连接都可以用 SQL Server 身份验证。而且，用户连接到已经以混合模式安装的 SQL Server 实例时，连接总是会明确提供 SQL Server 登录名称。这样可以使用与 Windows 中的用户名不同的登录名进行连接。

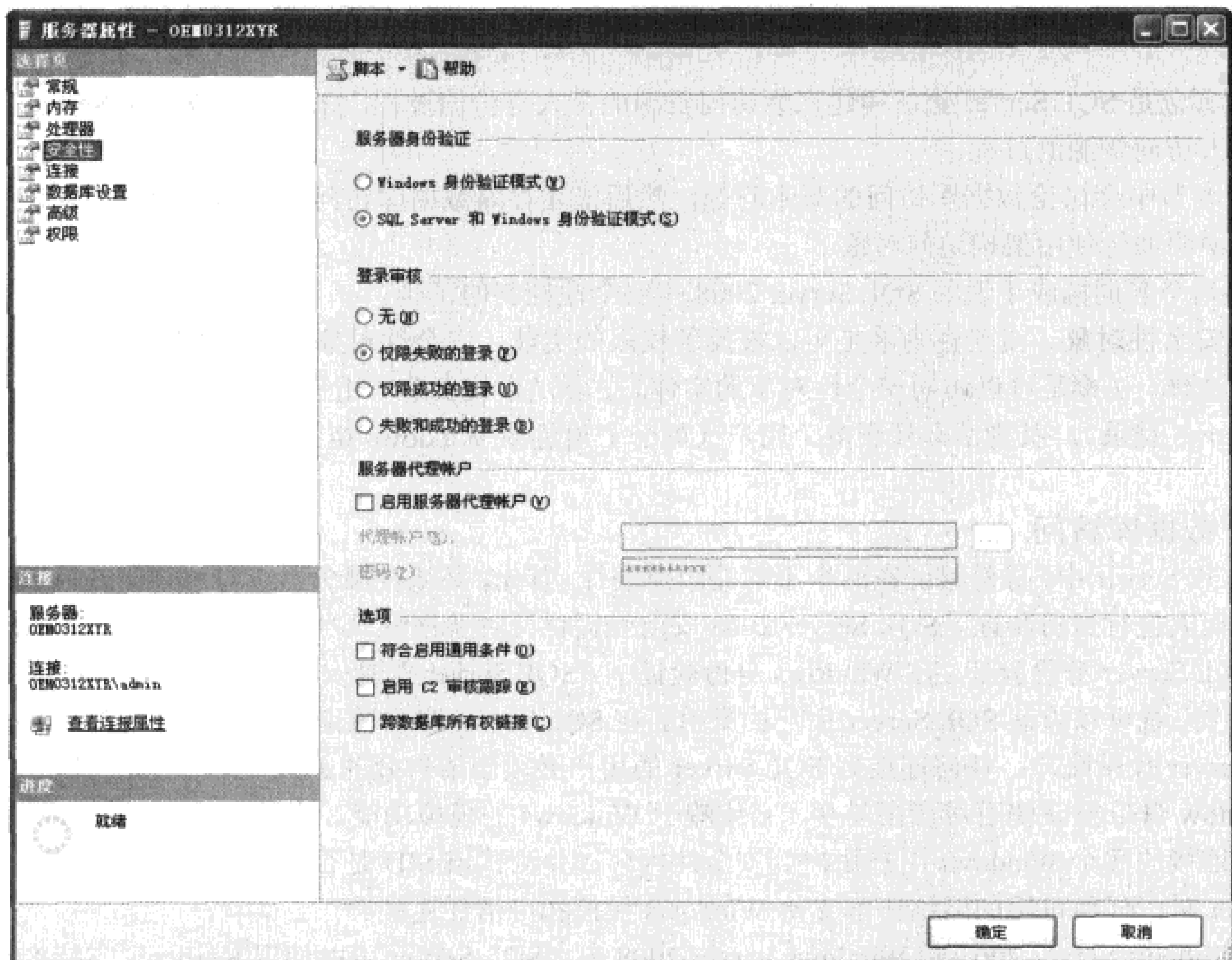


图 3-7　在“服务器属性”对话框中为 SQL Server 实例选择身份验证模式

所有的登录名，不管是 Windows 还是 SQL Server 身份验证，都可以在 *sys.server_principals* 目录视图（它也包括每个服务器主体的 SID）中看到。如果主体是 Windows 登录，那么 SID 与 Windows 用于验证用户对 Windows 资源的访问所用的相同。视图包含对应于服务器角色、Windows 组、映射到证书和非对称键的行，我就不在这里讨论这些主体了。

3.12.2　管理数据库安全性

正如在 *sys.databases* 视图中看到的那样，它有一列是拥有数据库的登录 SID，登录名可以是数据库的所有者。数据库是登录名拥有的唯一资源。后面将会看到，数据库中的所有对象都由数据库主体所有。

主体使用的 SID 决定主体能访问的数据库。每个数据库都有 *sys.databases_principals* 目录视图，可以把它想成是将登录名称映射到该数据库的映射表。虽然登录名和用户名可以相同，但是它们截然不同。下面的查询显示 *AdventureWorks2008* 数据库中的用户映射到登录名，还显示了每个数据库用户的默认架构（稍后讨论）：

```
SELECT s.name as [Login Name], d.name as [User Name],
     default_schema_name as [Default Schema]
   FROM sys.server_principals s
      JOIN sys.database_principals d
   ON d.sid = s.sid;
```

这是在我的 *AdventureWorks2008* 数据库中得到的结果：

```
Login Name User Name  Default Schema
---------- ---------- --------------
sa         dbo        dbo
sue        sue        sue
```

注意登录 *sue* 与数据库中的用户名具有相同的值。不能保证 *sue* 能访问的其他数据库也使用相同的用户名。登录名 *sa* 的用户名为 *dbo*，该名称是特别登录，*sa* 登录、*sysadmin* 角色中的所有登录和 *sys.databases* 中列出的所有作为数据库所有者的登录都使用它。在数据库中，拥有对象的是用户而不是登录，权限授予的是用户而不是登录。

之前的结果也显示了 *AdventureWorks2008* 数据库中每个用户的默认架构。在这种情况下，默认架构与用户名相同，但并不总是这样，在下一节可以看到。

3.12.3 数据与架构

在 ANSI SQL-92 标准中，*架构*的定义是由单个用户所有的、构成单个命名空间的数据库对象的集合。*命名空间*是对象的集合，这些对象不能重名。例如，除非两个表在不同的架构中，否则它们的名称不能相同，因此相同架构中的任何两个表格的名称都不能相同。可以把架构想成是对象的容器（在数据库工具的上下文中，架构也指描述架构或数据库中对象的目录信息。在 SQL Server 分析服务中，架构是多维对象如立方体和尺寸的描述）。

3.12.4 主体与架构

在 SQL Server 2005 以前，有一个 *CREATE SCHEMA* 命令，但它其实不做什么事，因为在用户和想要改变或删除的架构之间具有隐含的关系。实际上，这个关系很紧密，以至于这些早期 SQL Server 版本的许多用户都没有意识到用户和架构是不同的。每个用户都是同名架构的所有者。例如，如果创建了用户 *sue*，那么 SQL Server 2000 就会创建名为 *sue* 的架构，*sue* 是 *sue* 的默认架构。

在 SQL Server 2005 和 SQL Server 2008 中，用户和架构是两件不同的事。要了解用户和架构之间的区别，想一下：权限授予用户，但是对象放在架构中。

命令 *GRANT CREATE TABLE TO sue* 引用用户 *sue*，假设 *sue* 又创建了表，如下：

```
CREATE TABLE mytable (col1 varchar(20));
```

这个表被放在 *sue* 的默认架构中，这可能是架构 *sue*。如果另一个用户想从这个表获取数据，他可以发出以下命令：

```
SELECT col1 FROM sue.mytable;
```

在这个命令中，*sue* 引用了包含表的架构。

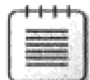
主要主体或次要主体可以拥有架构。虽然 SQL Server 2008 数据库中的每个对象都由用户所有，但是千万别用它的所有者引用对象，而是通过包含它的架构引用它。大多数情况下，架构的所有者与架构中所有对象的所有者相同。元数据视图 *sys.objects* 包含列 *principle_id*，如果 *principle_id* 和对象架构的所有者不同，它就包括对象所有者的 *user_id*。而且，从来不会把用户加到架构中，包含对象的是架构而不是用户。对于向后兼容性来说，如果执行 *sp_adduser* 或 *sp_grantdbaccess* 过程将用户添加到数据库中，SQL

Server 2008 会创建具有相同名称的用户和架构，它还会让架构成为新用户的默认架构。但是，应该习惯使用新命令 *DDL CREATE USER* 和 *CREATE SCHEMA*，因为 *sp_adduser* 和 *sp_grantdbaccess* 已经被取代了。创建用户时，如果想指定默认的架构就可以这么做，但是默认架构是 *dbo* 架构。

3.12.5 默认架构

在 SQL Server 2008 中创建新数据库时，其中包括了几个架构。它们包括 *dbo*、*INFORMATION_SCHEMA* 和 *guest*。而且每个数据库都有一个称为 *sys* 的架构，它提供访问所有系统表和视图的方法。最后，在 SQL Server 2008 中，除 *public* 以外的每个固定数据库角色都有同名架构。

创建用户时，可以给用户分配默认架构，它有可能不存在。任何时候用户最多有一个默认架构。如前所述，如果没有为用户指定默认架构，用户的默认架构就是 *dbo*。在创建对象或引用对象的过程中，名称解析会使用用户的默认架构，这对向后兼容来说有好有坏。好的是如果从 SQL Server 2000 升级数据库，该数据库可能在 *dbo* 架构中有很多对象，您的代码仍然可以继续引用那些对象，而不用明确指定架构。坏的是对于创建对象来说，SQL Server 尝试在 *dbo* 架构中而不是在由创建表的用户所有的架构中创建对象。用户可能没有在 *dbo* 架构(即使它是用户的默认架构)中创建对象的权限。为了避免混淆，在 SQL Server 2008 中，总是应该为所有的对象访问和对象管理指定架构名称。



注意：

当 sysadmin 角色中的登录用单个部件名创建对象时，架构总是 dbo。但是 sysadmin 可以明确指定在其中创建对象的其他架构。

要想在架构中创建对象，必须满足以下条件。

- 必须存在架构。
- 创建对象的用户必须具有直接或通过角色成员创建对象的权利(通过 *CREATE TABLE*、*CREATE VIEW*、*CREATE PROCEDURE* 等)。
- 创建对象的用户必须是架构的所有者、拥有架构的角色的成员，或者用户必须具有在架构上的 ALTER 权利，或者在数据库中具有 ALTER ANY SCHEMA 权限。

3.13 移动或复制数据库

有可能需要在系统上进行维护前、硬件失败后或者用更新更快的系统更新硬件时，移动数据库。复制数据库是创建二次开发或测试环境的常用方法。可以使用*分离*和*附加*技术或者备份数据库并在新位置还原它，来移动和复制数据库。

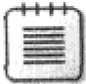
3.13.1 分离和重新附加数据库

可以使用简单的存储过程从服务器上分离数据库。分离数据库需要没人使用数据库时。如果发现当前有无法断开的连接，可以使用 *ALTER DATABASE* 命令，并用断开当前连接的终止选项之一把数据库设置为 SINGLE_USER 模式。分离数据库能保证数据库中没有未完成的事务，而且在内存中没有该数据库的脏页。如果不能满足这些条件，分离操作就会失败。一旦分离了数据库，就会从 *sys.databases* 目录视图和潜在的系统表中删除该项。

下面是分离数据库的命令：

```
EXEC sp_detach_db <name of database>;
```

一旦分离了数据库，从 SQL Server 的角度来看，就好像数据库被删除了一样。该数据库的元数据不会存留在 SQL Server 实例中，唯一有可能跟踪它的情况是当 *msdb* 数据库包含未删除数据库的备份和还原历史。但是完成备份和还原时的历史不会提供任何关于数据库的结构或内容的信息。如果计划以后重新附加数据库，应该记录属于数据库的所有文件的属性。



注意：

DRDOP DATABASE 命令还会从实例中删除数据库的所有跟踪记录，但是删除数据库比分离数据库更严重。SQL Server 在删除数据库之前要确定没人连接到它，但它不会检查脏页或打开的事务。删除数据库也会从操作系统中删除物理文件，所以除非有备份，否则数据库就真的没了。

要附加数据库，可以使用 *CREATE DATABASE* 命令和 FOR ATTACH 选项（有个存储过程 *sp_attach_db* 已经被取代，而且在 SQL Server 2008 中已经废弃，不推荐使用它）。*CREATE DATABASE* 命令可以控制所有文件及其放置，而且不仅限于 16 个文件（如 *sp_attach_db*）。*CREATE DATABASE* 没有这样的限制，实际上，可以为每个数据库指定最多 32 767 个文件和 32 767 个文件组。显示附加选项的 *CREATE DATABASE* 命令的语法总结如下：

```
CREATE DATABASE database_name
    ON <filespec> [ ,...n ]
    FOR { ATTACH
        | ATTACH_REBUILD_LOG }
```

注意<filespec>项只需要主文件，因为主文件包含关于所有其他文件位置的信息。如果使用与首次创建或最后附加数据库不同的路径附加现有文件，就必须有额外的<filespec>项。在任何情况下，不论是否在 *CREATE DATABASE* 命令中指定了数据库的所有数据文件，它们都必须可用。如果有多个日志文件，它们必须全部可用。

但是，如果读/写数据库有一个当前不可用的日志文件，而且数据库关闭且在附加操作之前没有打开的事务，那么 FOR ATTACH 会重新编译日志文件并在主文件中更新关于日志的信息。如果数据库是只读的，就不能更新主文件，所以不能重新编译日志。因此，附加只读数据库时，必须在 FOR ATTACH 语句中指定日志文件或文件。

或者使用 FOR ATTACH_REBUILD_LOG 选项，它指定通过附加操作系统文件的现有集合来创建数据库，这个选项限于读/写数据库。如果一个或多个事务日志文件丢失，会重建日志。必须有一个指定主文件的<filespec>项，而且如果日志文件可用，SQL Server 会使用这些文件，而不是重新编译日志文件，所以 FOR ATTACH_REBUILD_LOG 会像使用 FOR ATTACH 那样工作。

如果通过附加数据库重新编译事务日志，使用 FOR ATTACH_REBUILD_LOG 会断开日志备份链。应该考虑在执行这个操作之后进行完全备份。

通常将带有大型日志的读/写数据库复制到另一台服务器上（在这台服务器上几乎或者只用该副本用做读操作，因此比原始数据库需要的日志空间少）时，使用 FOR ATTACH_REBUILD_LOG。

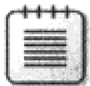
虽然文中说只应该在以前用 *sp_detach_db* 命令附加的数据库上使用 *CREATE DTABASE FOR ATTACH*，但有时候不必遵循这个建议。如果关闭 SQL Server 实例，文件也会关闭，就像已经断开数据库一样。但是不能保证在关闭之前把数据库的所有脏页都写入磁盘。如果日志可用的话，附加这样的数据库不会产生问题。日志文件具有所有已完成事务的记录，但是当数据库被附加时进行完全备份能让数据库保持一致。使用 *sp_detach_db* 过程的一个好处是 SQL Server 记录数据库完全关闭的事实，而且日志文件不是必须可用才能附加数据库。SQL Server 会编译一个新的日志文件，因为 *sp_detach_db* 创建的新日志文件会很小——小于 1MB，所以这是收缩已经远远大于想要的日志文件的快速方法。

3.13.2 备份和还原数据库

除了分离和附加，还可以用备份和还原将数据库移动到新位置。这种方法的一个好处是数据库根本不需要脱机，因为备份是完全联机的操作。本书不是教数据库管理员如何使用的书籍，您应该查询随附内容中的参考文件，那里有一些关于备份和还原数据库机理的好书推荐，还可以学习为您的机构设置备份和还原计划的最佳实践。尽管如此，与备份和还原过程有关的一些问题还是有助于理解为什么一个备份计划比另外一个更适合您的需要，所以我会在第 4 章简要讨论备份和还原。大部分问题涉及备份和还原操作中的事务日志的角色。

3.13.3 移动系统数据库

作为计划中的移动或维护操作的一部分，可能需要移动系统数据库。如果移动系统数据库，之后重新编译 *master* 数据库，就必须再次移动系统数据库，因为重新编译操作会把所有的系统数据库安装到它们的默认位置。移动 *tempdb*、*model* 和 *msdb* 的步骤与移动 *master* 数据库稍有不同。



注意：

在 SQL Server 2008 中，不能移动 *mssqlsystemresource* 数据库。如果移动此数据库的文件，就不能重新启动 SQL Server 服务。这在 *SQL Server 2008 Books Online* 的 RTM 版中写错了，那本书说可以移动 *mssqlsystemresource* 数据库，这个错误信息可能会在以后的版本中纠正。

下面是移动未损坏系统数据库（即不是 *master* 数据库）的步骤。

（1）对于即将移动数据库中的每个文件，使用 *ALTER DATABASE* 命令和 MODIFY FILE 选项指定新的物理位置。

（2）停止 SQL Server 实例。

（3）在物理上移动文件。

（4）重新启动 SQL Server 实例。

（5）通过运行以下查询，验证改变：

```
SELECT name, physical_name AS CurrentLocation, state_desc
FROM sys.master_files
WHERE database_id = DB_ID(N'<database_name>');
```

如果由于硬件失败，需要移动系统数据库，这样做就有点问题了，因为可能不能访问服务器再运行 *ALTER DATABASE* 命令。下面是移动损坏系统数据库的步骤（不是 *master* 数据库或资源数据库）。

（1）如果已经启动 SQL Server 实例，停止它。

（2）在命令行提示符处输入以下命令之一，在 *master-only* 模式下（通过指定跟踪标志 3608）启动 SQL Server 实例。

```
-- If the instance is the default instance:
NET START MSSQLSERVER /f /T3608

-- For a named instance:
NET START MSSQL$instancename /f /T3608
```

（3）对于数据库中每个要被移动的文件，使用 *ALTER DATABASE* 命令和 MODIFY FILE 选项指定新的物理位置。可以使用 Management Studio 或 SQLCMD 实用工具。

（4）退出 Management Studio 或 SQLCMD 实用工具。

（5）停止 SQL Server 实例。

（6）在物理上将文件移动到新位置。

（7）不用跟踪标志 3608，重新启动 SQL Server 实例。例如，运行 *NET START MSSQLSERVER*。

（8）运行以下查询检验改变：

```
SELECT name, physical_name AS CurrentLocation, state_desc
FROM sys.master_files
WHERE database_id = DB_ID(N'<database_name>');
```

3.13.4 移动 master 数据库

在 *SQL Server 联机丛书*中能找到移动 *master* 数据库的全部细节，但是我要在这里总结步骤。移动这个数据库和移动其他的系统数据库之间的最大区别是必须通过 SQL Server 配置管理器。

要想移动 *master* 数据库，可按以下步骤操作。

（1）打开 SQL Server 配置管理器，用右键单击目标 SQL Server 实例，选择“属性”，然后单击“高级”选项卡。

（2）将“启动参数”的值编辑为 *master* 数据库数据和日志文件的新目录位置。如果需要，还可以移动 SQL Server 错误日志文件。数据文件的参数值必须遵守-*d* 参数，日志文件的值必须遵守-*l* 参数，错误日志的值必须是-*e* 参数，如下所示：

```
-dE:\SQLData\master.mdf;
-lE:\SQLData\mastlog.ldf;
-eE:\ SQLData\LOG\ERRORLOG
```

（3）停止 SQL Server 实例，在物理上将文件移动到新位置。

（4）重新启动 SQL Server 实例。

（5）通过运行以下查询检验 *master* 数据库的文件改变：

```
SELECT name, physical_name AS CurrentLocation, state_desc
FROM sys.master_files
WHERE database_id = DB_ID('master');
```

3.14 兼容性级别

SQL Server 的每个新版本都包括大量新功能，许多都需要新关键字，也改变了早期版本中存在的某

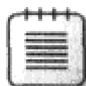
些行为。为了具有最高的向后兼容性，Microsoft允许用户将运行在SQL Server 2008实例上的数据库兼容性级别设置为下列模式：100、90或80。除非为*model*数据库改变级别，否则SQL Server 2008中所有新建数据库的兼容性级别都是100。已经被升级或从旧版本中附加的数据库的兼容性级别都被设置为数据库升级之前的版本。

除非特别说明，本书中所有例子和解释都假设用户在100兼容模式下使用数据库。如果发现SQL语句得到的结果和本书中的不同，应该首先执行下面的命令以检查数据库是否为100兼容模式：

```
SELECT compatibility_level FROM sys.databases
WHERE name = '<database name>';
```

要改变兼容级别，使用*ALTER DATABASE*命令：

```
ALTER DATABASE <database name>
SET COMPATIBILITY_LEVEL = <compatibility-level>;
```



注意：

兼容性级别选项是为了将数据库或应用程序升级到SQL Server 2008过程中提供过渡期。强烈建议您修改应用程序，以便不再需要兼容性选项。Microsoft不保证这些选项在以后的SQL Server版本中还会继续有效。

不是所有SQL Server旧版本的行为都可以通过兼容性级别重现。最重要的是，这些区别与是否识别新的保留关键字和新语法有关，它们不会影响内部如何处理查询。例如，如果将兼容性级别改为80，就会让系统表不可见或者不用架构。但是因为*MERGE*这个字在SQL Server 2008中是新的保留字（兼容性级别100），通过兼容性级别设置为80或90，就可以不用任何特殊的分隔符创建称为*MERGE*的表；或者如果数据库停在90兼容级别，就可以继续访问SQL Server 2005数据库中已经有的表。

关于兼容性级别和新保留字之间的行为区别的完整列表，请参阅*SQL Server联机丛书*中的*ALTER DATABASE兼容性级别*。

3.15　小结

数据库是对象（如表、视图和存储过程）的集合。虽然典型的SQL Server安装包括很多数据库，但它总是包括以下3个数据库：*master*、*model*和*tempdb*。安装时通常也包括*msdb*，但是可以删除该数据库（删除*msdb*数据库需要特殊的跟踪标记，几乎不推荐）。SQL Server实例还包括使用一般的工具无法看到的*mssqlsystemresource*数据库。每个数据库都有自己的事务日志，对象之间的整合性约束让数据库在逻辑上是一致的。

数据库是以一对多的关系存储在操作系统文件中的。每个数据库都至少有一个数据文件和一个事务日志文件。您能够轻易手动、自动增大或减小数据库及其文件的大小。

第 4 章

日志记录和恢复

Kalen Delaney

在第 3 章中，讨论了在 Microsoft SQL Server 数据库中所创建的、用于存储信息的数据文件。每个数据库还至少有一个存储自己事务日志的文件。第 3 章谈到 SQL Server 事务日志和日志文件，但并未详细讲解日志文件和数据文件的区别，也没讲 SQL Server 如何使用日志文件。本章会讲述 SQL Server 日志文件的结构，以及如何在记录事务信息时管理这些日志文件。我会解释 SQL Server 日志文件如何增长及何时、如何减小日志文件。最后讲解如何在 SQL Server 备份和还原期间使用日志文件，以及数据库的恢复模型如何影响日志文件。

4.1 事务日志基础

事务日志记录对数据库所做的更改并存储足够的信息，让 SQL Server 可以恢复数据库。每次启动 SQL Server 实例时进行恢复操作，每次 SQL Server 从备份还原数据库或日志时也可以进行恢复。*恢复*是使数据文件和日志一致的过程。任何在日志中指示已经提交的数据更改必须出现在数据文件中，任何未标记为提交的更改不能出现在数据文件中。如果 SQL Server 收到来自客户端的回滚操作请求（通过 *ROLLBACK TRAN* 命令），或者发生错误（如死锁）生成内部 *ROLLBACK*，日志也会存储需要回滚操作的信息。

从物理上讲，事务日志是在创建或更改数据库时，与数据库关联的一个或多个文件。执行数据库修改的操作在描述以下内容的事务日志中写记录：所做更改（包括操作修改的数据页的页码），已添加或删除的数据值，关于修改所属事务的信息、事务开始和结束日期及时间。发生特定的内部事件（如检查点）时，SQL Server 也会写日志记录。每个日志记录都用一个保证唯一性的日志序列号（LSN）标记。所有属于相同事务的日志项都会链接起来，这样撤销操作（就像回滚）和重做操作（在系统恢复过程中）都能够轻易找到事务的所有内容。

“缓冲区管理器”保证在将更改写入数据库之前，先写入事务日志（这称为*预写日志记录*）。因为 SQL Server 通过 LSN 在日志中记录当前位置，所以可以实现这个保证。每次页面发生更改后，就会把与该更改日志项对应的 LSN 写入数据页的页眉中。只有当页面上的 LSN 小于等于写入日志最后一条记录的 LSN 时，才能将脏页写入磁盘。缓冲区管理器还保证按特定的顺序写入日志页，不管系统何时发生故障，都能清楚地显示发生故障后必须处理的日志块。

事务的日志记录是在将提交确认发送到客户端进程之前写入磁盘的，但其实可能还没有将实际更改的数据写入数据页。虽然写日志是异步的，但是在提交时，线程必须等待在事务日志中完成写提交记录（SQL Server 必须等待提交记录写完，才能知道相关的日志记录在磁盘上是安全的）。写数据页是完全异步的，即写数据页只需告知操作系统，SQL Server 可以在以后检查是否完成了写操作。因为日志包括需要撤销操作的所有信息，所以即使在完成写操作之前发生断电或系统崩溃，都不必立即完成写操作。如果系统等待每个 I/O 请求完成才能继续，那么就会慢得多。

记录涉及到标定每个事务的起始时间（如果事务使用保存点的话，还要记录保存点）。起始和结束标定之间是有关对数据所做更改的信息，这个信息可能是实际的“之前和之后”数据形式，也可以指已经完成的操作，以便提取这些值。典型事务的结尾处被标记为“提交”记录，这表示必须在数据库的数据文件中反映事务，或者重做（如果需要）。在正常的运行时（非系统重启）过程中，由明确回滚或者资源错误（例如，内存不足错误）这样的事导致放弃的事务实际上会通过应用撤销原始数据修改的更改来撤销操作。这些更改的记录会被写入日志，并标记为“补偿日志记录”。

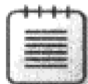
如前所述，恢复有两种类型，它们的目的都是确保日志和数据吻合。每次启动 SQL Server 时都会运行*重启恢复*，因为每个数据库都有自己的事务日志，所以这个进程会在每个数据库上运行。SQL Server 错误日志报告重启恢复的进度，错误日志会说明每个数据库已经前滚和回滚了多少日志。这种类型的恢复有时称为*崩溃恢复*，因为崩溃（或者说 SQL Server 服务的意外停止）需要在重启服务时运行恢复进程。如果服务完全关闭且任何数据库中都没有打开的事务，那么在系统重启时，只需要最小恢复。在 SQL Server 2008 中，可以在多个数据库上并行运行重启恢复，每个由不同的线程来处理。

另一种类型的恢复叫*还原恢复*（或称为*媒体恢复*），在执行恢复操作时根据请求运行。这个进程确保在数据中反映事务日志备份中的所有已提交事务，还确保不会在数据中显示任何未完成的事务。我会在本章后面讲述更多关于还原恢复的内容。

两种恢复类型都必须处理两种情况：日志中的事务被记录为已提交但还没有写入数据文件、数据文件的更改与已提交事务不对应。因为每次提交事务时都会把已提交事务日志写入磁盘上的日志文件，所以会发生这两种情况。每次数据库中出现检查点时，更改的数据页就会完全异步地写入磁盘上的数据文件。正如第 1 章中提到的，也可以在其他的时刻将数据页写入磁盘，但是定期发生的检查点操作让 SQL Server 知道何时*所有*已更改页（或者说脏页）已经写入磁盘。因为缓存的日志记录也被认为是脏的，所以检查点操作也将来自正在进行的事务的日志记录写入磁盘。

如果事务提交之后、将数据写到数据页之前，SQL Server 服务停止，那么当 SQL Server 启动并运行恢复时，就必须前滚事务。SQL Server 基本上通过重新应用事务日志中指示的更改重做事务。所有需要重做的事务（虽然其中一些可能在下一阶段重做）会先处理，这称为恢复的*重做*阶段。

如果检查点出现在提交事务之前，它会将未提交更改写入磁盘。如果 SQL Server 服务在进行提交之前停止，那么恢复进程会在数据文件中找出对未提交事务做出的更改，通过重做事务日志中反映的更改来回滚事务。回滚所有未完成的事务称为恢复的*撤销*阶段。



注意：

我会继续将恢复称为系统启动功能，这是它迄今为止最常见的角色。但要记住，恢复会从备份或附加数据库的最后一步中运行，还可以手动强制执行恢复。而且在创建数据库快照、在数据库镜像或数据库镜像失败时，也会运行恢复。

本章后面会涵盖在数据库恢复过程中与恢复有关的一些特殊问题，包括可以用 *ALTER DATABASE* 语句设置的 3 个恢复模型，以及在日志中放置命名标记以指示要恢复到的特定点的能力。之后讨论关于恢复的一般问题：是在 SQL Server 服务重启时执行，还是从备份还原数据库时进行恢复。

4.1.1 恢复阶段

在恢复过程中，只会评估那些自最后一个检查点之后发生的更改或那时正在进行的更改，以确定是

否需要重做或撤销这些更改。在最后一个检查点之前完成的任何操作（无论是正在提交还是已经回滚）都会精确地反映在数据页中，恢复过程中不需要做其他工作。

恢复算法有 3 个阶段，都是围绕事务日志中的最后一个检查点记录。图 4-1 说明了这 3 个阶段。

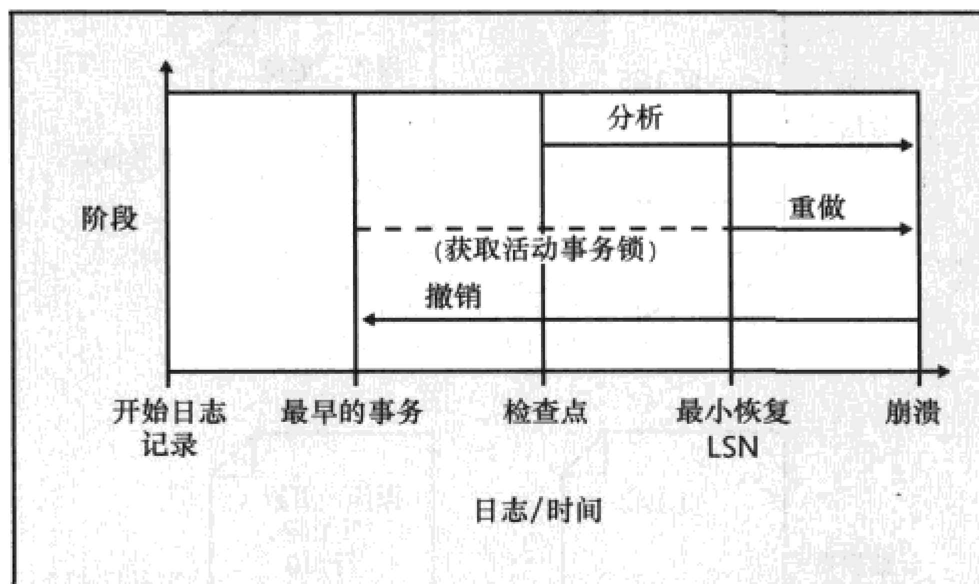


图 4-1 SQL Server 恢复过程的 3 个阶段

阶段 1：分析。第一阶段是向前的步骤，从事务日志的最后一个检查点记录开始。这个步骤确定和构造脏页表（DPT），脏页表包括可能在 SQL Server 停止时是脏的页面。SQL Server 停止时，也会构造包含未提交操作的活动事务表。

阶段 2：重做。这个阶段将数据库恢复到 SQL Server 服务停止时的状态。这个向前步骤的起点是最早未提交事务的开始。DPT 中的最小 LSN 是 SQL Server 预期在页面上进行的第一次重做操作，但是它需要重做从最早打开的事务的起始部分开始的、已经记录的操作，这样可以获取必要的锁（在 SQL Server 2005 之前，只需重新获取分配锁。在 SQL Server 2005 之后，需要重新获取所有打开事务的锁）。

阶段 3：撤销操作。这个阶段使用在第一阶段（分析）中找到的活动事务列表（在 SQL Server 崩溃时未提交的），它单独回滚每个活动事务。SQL Server 执行每个事务的事务日志条目之间的链接。在 SQL Server 停止时，任何未提交的事务都会被撤销，因此数据库中实际上不会反映任何更改。

SQL Server 用日志记录数据更改和任何已应用到被更改对象上的锁，这能让 SQL Server 重启时，支持一个称为*快速恢复*的功能（只在企业版和开发版中可用）。如果使用快速恢复，重做阶段结束后就可以使用数据库。可以重新获得在原始更改过程中获得的那些锁，以防止其他进程访问需要撤销更改的数据；数据库中的所有其他数据仍然可用。在媒体恢复的过程中不能进行快速恢复，但数据库镜像恢复能进行快速恢复，镜像恢复使用的是媒体恢复和重启恢复的混合体。

此外，SQL Server 用多个线程在不同的数据库上同时处理恢复操作，因此 ID 号高的数据库不必在它们自己的恢复进程开始之前，等待所有 ID 号低的数据库完全恢复。

页 LSN 和恢复

每个数据库页的页眉中都有 LSN，它反映修改该页中一行的最后一个日志项在事务日志中的位置。对应于数据页更改的每个日志记录都有两个与其关联的 LSN，除了实际日志记录的 LSN，还记录此日志记录所记录的更改发生之前、数据页上的 LSN。在事务的重做操作过程中，会将每个日志记录上的 LSN 与日志项所修改的数据页的页 LSN 进行比较。如果页 LSN 等于日志记录中的上一个页 LSN，就会重做

日志项所指示的操作。如果页上的 LSN 等于或高于此日志记录的实际 LSN，SQL Server 就会跳过 *REDO* 操作。图 4-2 说明了这两种可能。页上的 LSN 不能介于日志记录的上一个值和当前值之间。

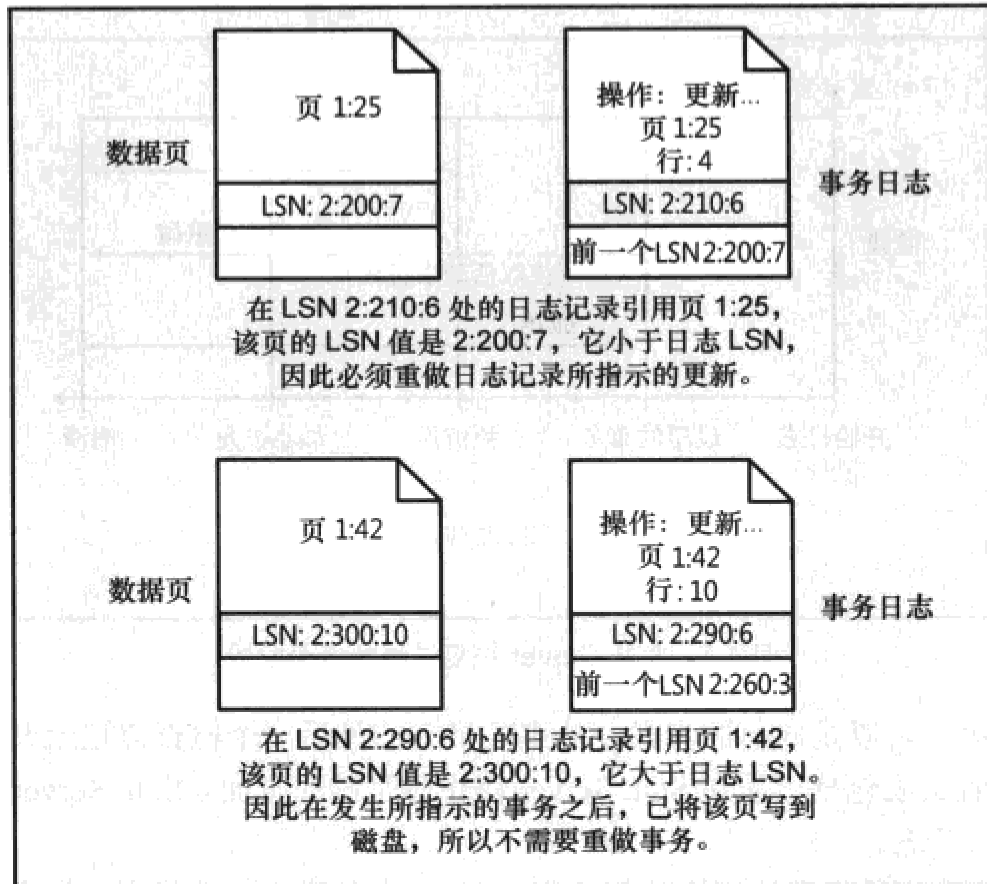


图 4-2　比较 LSN 以决定是否在恢复期间处理日志项

因为恢复会找出日志中的最后一个检查点记录（以及在检查点时刻仍然活动的事务）并从这里开始恢复，所以恢复时间短。可以从日志或存档中清除检查点以前提交的所有更改，不然可能要花很长时间恢复，事务日志可能也会变得过大。不管事务开始之后有多少个检查点，也不管开始或完成了多少其他事务，都不能把事务截断到最早的打开事务之前的时刻。如果事务仍然打开着，就必须保留日志，因为仍不清楚事务是否已经完成或将要完成。可能最后还要回滚或前滚事务。

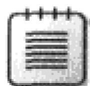
注意：

截断事务日志是逻辑操作，只是把日志的一些部分标记为“不再需要”，所以还可重用这部分空间。截断不是物理操作，不会减少磁盘上的事务日志文件的大小。要想减少物理大小，必须执行缩小操作。

一些 SQL Server 管理员已经注意到，甚至备份日志以后都不能截断事务日志，这个问题通常是用户打开事务却忘记这件事引起的。出于这个原因，从应用程序开发的角度考虑，应确保事务简短。不能截断日志的另一个可能的原因与日志读取器没有处理完所有相关的日志记录时使用事务进行复制的表有关，但这种情况不常见，因为通常日志读取器工作时，只会有几秒钟的延迟。可以用 *DBCC OPENTRAN* 查找最早的、打开的事务或者最早被复制却还没有处理的事务，然后再采取正确的措施（如终止有问题的进程或运行 *sp_repldone* 存储过程，以清除重复事务）。第 10 章将讨论事务管理的问题和一些可能的解决方法。我会在下一节中讨论压缩日志。

4.1.2 读日志

虽然日志包括对数据库所做的每个更改的记录，但它本不是用做审核工具的。事务日志用于保证 SQL Server 在语句或系统出现故障时的可恢复性，并允许系统管理员把更改的备份放到 SQL Server 数据库上。如果想保持对数据库所做更改的可读记录，得自己做审核。可以用 SQL Server Profiler 或在第 2 章中讨论的 SQL Server 中的跟踪机制之一，通过为 SQL Server 的活动建立跟踪来进行审核。



注意：

您可能知道有一些第三方工具可以读取事务日志并显示数据库中进行的所有操作，让您可以回滚这些操作中的任何操作。这些工具的开发人员在研究事务日志文件的位级转储及将该信息与未记录 *DBCC LOG* 命令的输出相关联上面花了上万个小时。一旦他们有产品投放市场，微软公司就和他们一起工作，让他们更容易开发后续版本，但是 SQL Server 2008 还没有这样的工具。

虽然您可能会假设直接读取事务日志可能是有趣甚至有用的事，但是这样通常会有太多信息。如果事先知道想要记录运行 SQL Server 的服务器的活动，那么最好用合适的筛选器定义跟踪，以捕获对您有用的信息。

4.2 更改日志大小

不管为事务日志定义多少个物理文件，SQL Server 总是把日志当成连续流来对待。例如，当 *DBCC SHRINKDATABASE* 命令（第 3 章中讨论过）确定日志可以缩小多少时，它不是单独考虑每个日志文件，而是根据整个日志确定可压缩大小。

4.2.1 虚拟日志文件

任何数据库的事务日志都作为虚拟日志文件（VLF）集来管理，它的大小由 SQL Server 在内部根据所有日志文件的总大小和增大日志所使用的增长量决定。首次创建日志文件时，大小总是在 2 个和 16 个 VLF 之间。如果日志为 1MB 或更小，SQL Server 会将日志文件的大小除以最小的 VLF 大小（31KB×8KB），确定 VLF 的个数。如果日志文件在 1MB 和 64MB 之间，SQL Server 会将日志分成 4 个 VLF。如果日志文件大于 64MB 且小于或等于 1GB，会创建 8 个 VLF。如果日志超过 1GB，就会有 16 个 VLF。日志文件增长时，用相同的方法确定添加多少新 VLF。日志总是以整个 VLF 为单位增长，而且缩小也只能到 VLF 边界为止（图 4-3 说明了一个物理日志文件和几个 VLF）。

VLF 可以是以下 4 种状态之一。

活动的。日志的活动部分从代表活动（未提交）事务的最小 LSN 开始，结束于最后一个写入的 LSN。任何包含活动日志任何部分的 VLF 都被认为是活动的 VLF（物理日志中的未使用空间不属于任何 VLF）。图 4-3 包括两个活动 VLF。

可恢复的。在最早的活动事务之前的那部分日志仅用于维护日志备份序列，从而将数据库恢复到前一个状态。

可重用的。如果没有维护事务日志备份或者已经备份了日志，就不需要最早的活动事务之前的 VLF，而且可以重用这些空间。截断或备份事务日志会将可恢复 VLF 转变为可重用 VLF。为了确定哪些 VLF

可以重用，活动事务不仅仅包括打开的事务。最早的活动事务可能是标记为复制但还未被处理的事务、日志备份操作的起点或 SQL Server 定期进行内部诊断扫描的起点。

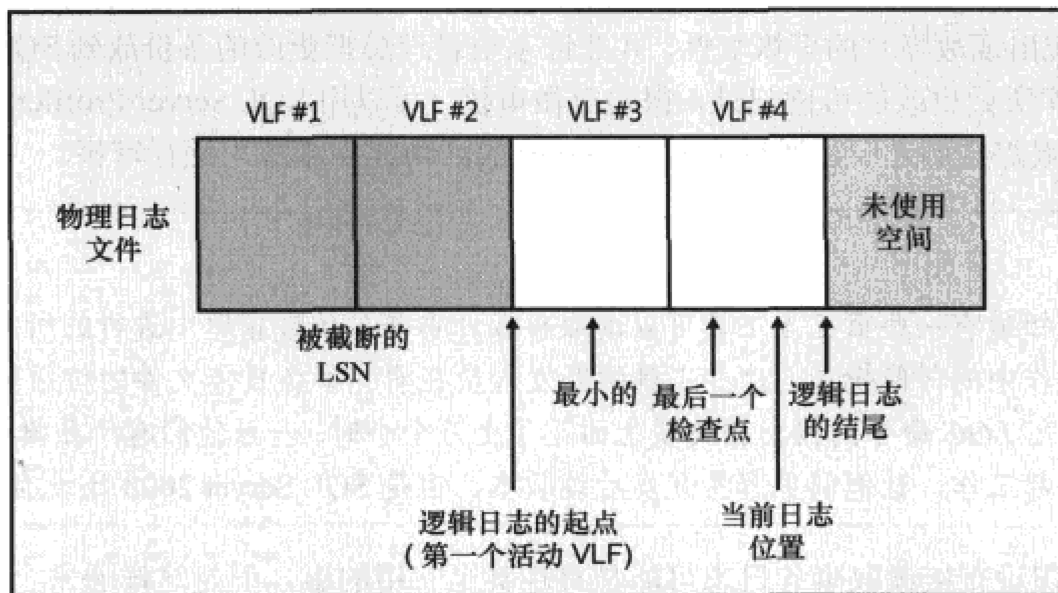


图 4-3 构成物理日志文件的多个 VLF

未使用的。如果已记录的活动不足或者已经把早期的 VLF 标记为可重用并已经重用，就可能还未使用日志文件结尾处的一个或多个 VLF。

4.2.2 观察虚拟日志文件

通过执行未记录命令 *DBCC LOGINFO*，可以观察虚拟日志文件的相同关键属性。这个命令不需要任何参数，所以必须在想要获取信息的数据库中运行，它为每个 VLF 返回一行。在我的 *AdventureWorks2008* 数据库中运行这个命令时，得到下面 8 行（没有显示所有列）：

```
FileId   FileSize       StartOffset    FSeqNo      Status      CreateLSN
-------- -------------- -------------- ----------- ----------- -------------------
2        458752         8192           42          2           0
2        458752         466944         41          0           0
2        458752         925696         43          2           0
2        712704         1384448        44          2           0
2        4194304        2097152        47          2           44000000085601161
2        4194304        6291456        46          2           44000000085601161
2        4194304        10485760       40          2           44000000085601161
2        4194304        14680064       0           0           44000000085601161
```

行数显示我的数据库中有多少个 VLF，*FileID* 列显示包含 VLF 的那些日志的物理文件。我的 *AdventureWorks2008* 数据库只有一个物理日志文件。*FileSize* 和 *StartOffset* 是以字节为单位表示的，所以可以看到第一个 VLF 从 8192 字节之后开始，8192 是一个页面的字节数。日志文件的第一个物理页包含页眉信息（不是日志记录），所以 VLF 被认为是从第二页开始。*FileSize* 列其实对大多数行来说是多余的，因为可以通过对两个连续 VLF 的 *StartOffset* 值相减算出大小。行是以物理顺序列出的，但这并不总是使用 VLF 的顺序。使用顺序（逻辑顺序）反映在 *FSeqNo*（这表示文件序列号）列中。

在前面显示的输出中，可以看到行是用 *StartOffset* 按物理顺序列出的，但与逻辑顺序不符。*FSeqNo* 值表示第 7 个 VLF 实际上是用户（逻辑）顺序中的第一个，用户顺序中的最后一个是物理顺序的第 5 个

VLF。*Status* 列指示是否可以重用 VLF。状态 2 表示 VLF 既是活动的又是可恢复的，状态 0 表示可重用或完全未使用（完全未使用的 VLF 的 *FSeqNo* 值为 0，就像输出的第 8 行那样）。正如我在前面提到的，截断或备份事务日志将可恢复的 VLF 变成可重用的 VLF，所以不包括活动日志记录的所有 VLF 的状态都会从 2 变到状态 0。实际上，有种方法可以判断哪个 VLF 是活动的：在日志备份或截断之后状态仍然是 2 的 VLF 一定包含活动事务的记录。具有状态 0 的 VLF 可以被新日志记录重用，不需要为记录数据库中的活动而增长日志。另一方面，如果日志中所有 VLF 的状态都是 2，那么 SQL Server 需要向日志中添加新 VLF，以记录新的事务活动。以前显示过的 *DBCC LOG* 输出的最后一列称为 *CreateLSN*，该列包含一个 LSN 值，实际上它是在将 VLF 添加到事务日志时的当前 LSN。如果 *CreateLSN* 的值是 0，就表示创建数据库时，VLF 是所创建的原始日志文件的一部分。也可以通过查看哪个 VLF 具有与 *CreateLSN* 相同的值，知道在任何一个操作中添加了多少个 VLF。在我的输出中，*CreateLSN* 的值表示我的日志文件只增长过一次，同时添加了 4 个新 VLF。

多个日志文件

在前面讲过，SQL Server 将多个物理日志文件当做一个序列流来对待，就是说用完一个物理文件中的所有 VLF 之后才会用第 2 个文件中的任何 VLF。如果具有管理良好的、定期备份或截断的日志，可能永远都不会使用除第一个文件之外的任何日志文件。如果需要新 VLF 时，多个物理日志文件中都没有可以重用的 VLF，SQL Server 就会以循环的方式把新 VLF 添加到每个物理日志文件中。

其实可以通过检查 *DBCC LOGINFO* 的输出，查看不同物理文件的使用顺序。第一列是物理文件的 *file_id*。如果我们可以将 *DBCC LOGINFO* 的输出捕获到表中，就能让它按照对我们有用的方式排序。下面的代码创建可以保存 *DBCC LOGINFO* 输出的表，称为 *sp_loginfo*。因为该表是在 master 数据库中创建的，并且以 3 个字母“sp_”开始，所以能在任何数据库中访问和修改它：

```
USE master
GO
IF EXISTS (SELECT 1 FROM sys.tables
                        WHERE name = 'sp_LOGINFO')
        DROP TABLE sp_loginfo;
GO
CREATE TABLE sp_LOGINFO
(FileId tinyint,
 FileSize bigint,
 StartOffset bigint,
 FSeqNo int,
 Status tinyint,
 Parity tinyint,
 CreateLSN numeric(25,0) );
GO
```

下面的代码创建了一个名为 *TWO_LOGS* 的新数据库，然后从 *AdventureWorks2008* 样例数据库中复制一个大表到 *TWO_LOGS* 中，导致日志增长：

```
USE Master
GO
IF EXISTS (SELECT * FROM sys.databases
                        WHERE name = 'TWO_LOGS')
    DROP DATABASE TWO_LOGS;
```

```
GO
CREATE DATABASE TWO_LOGS
  ON PRIMARY
  (NAME = Data ,
    FILENAME =
    'C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL\DATA\TWO_LOGS.mdf'
         , SIZE = 100 MB)
 LOG ON
(NAME = TWO_LOGS1,
  FILENAME =
  'C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL\DATA\TWO_LOGS1.ldf'
       , SIZE = 5 MB
       , MAXSIZE = 2 GB),
(NAME = TWO_LOGS2,
 FILENAME =
 'C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL\DATA\TWO_LOGS2.ldf'
      , SIZE = 5 MB);
GO
```

如果运行 *DBCC LOGINFO*，就会注意到它返回按 *FileID* 排序的 VLF，而且最初文件序列号的值（*FSeqNo*）也是按顺序的：

```
USE TWO_LOGS
GO
DBCC LOGINFO;
GO
```

现在可以从另一个表复制数据向数据库添加一些行：

```
SELECT * INTO Orders
    FROM AdventureWorks2008.Sales.SalesOrderDetail;
GO
```

如果再次运行 *DBCC LOGINFO*，就会看到在 *SELECT INTO* 操作之后，即使每个 *FileID* 有更多的行，但输出还是会按 *FileID* 排序，根本与 *FSeqNo* 的值不相关。可以改成在 *sp_loginfo* 表中保存 *DBCC LOGINFO* 的输出，再按 *FSeqNo* 排序：

```
TRUNCATE TABLE sp_LOGINFO;
INSERT INTO sp_LOGINFO
   EXEC ('DBCC LOGINFO');
GO
-- Unused VLFs have a Status of 0, so the CASE forces those to the end
SELECT * FROM sp_LOGINFO
ORDER BY CASE FSeqNo WHEN 0 THEN 9999999 ELSE FSeqNo END;
GO
```

SELECT 的输出如下所示：

```
FileId  StartOffset           FSeqNo       Status   CreateLSN
------  --------------------  -----------  ------   -----------------
2    8192                     43           0        0
2    1253376                  44           0        0
2    2498560                  45           0        0
2    3743744                  46           0        0
```

```
3    8192        47    0    0
3    1253376     48    0    0
3    2498560     49    0    0
3    3743744     50    0    0
2    5242880     51    0    50000000247200092
2    5496832     52    0    50000000247200092
3    5242880     53    0    51000000035600288
3    5496832     54    0    51000000035600288
2    5767168     55    0    53000000037600316
2    6021120     56    0    53000000037600316
3    5767168     57    0    56000000010400488
3    6021120     58    0    56000000010400488
2    6356992     59    0    58000000007200407
2    6610944     60    0    58000000007200407
3    6356992     61    0    60000000025600218
3    6610944     62    0    60000000025600218
2    7012352     63    0    62000000023900246
2    7266304     64    0    62000000023900246
3    7012352     65    0    64000000037100225
3    7266304     66    0    64000000037100225
2    7733248     67    0    66000000037600259
2    7987200     68    0    66000000037600259
2    8241152     69    0    66000000037600259
3    7733248     70    0    68000000035500288
3    7987200     71    0    68000000035500288
3    8241152     72    0    68000000035500288
2    8519680     73    0    71000000037300145
2    8773632     74    0    71000000037300145
2    9027584     75    0    71000000037300145
3    8519680     76    0    75000000018700013
3    8773632     77    2    75000000018700013
3    9027584     0     0    75000000018700013
```

现在注意到使用了最初的 8 个初始 VLF（*CreateLSN* 值是 0 的那些 VLF）之后，VLF 会在物理文件之间交替出现。因为每次日志都有增长，所以创建了几个新 VLF，先从 *FileID 2*，再从 *FileID 3* 开始。还未使用添加到 *FileID* 的最后一个 VLF。

所以，如果完成了彻底测试并确定数据库事务日志的最优大小，就真的没有理由使用多个物理日志文件了。但是如果发现日志的增长超过预期，而且如果包含日志的卷没有足够的可用空间让日志增长，就可能需要在另一个卷上创建第 2 个日志文件。

4.2.3 自动截断虚拟日志文件

如果下列之一为真，SQL Server 则假设您没有维护日志备份序列。

- 您已经通过将恢复模式设置为 SIMPLE，从而将数据库配置为定期截断日志。
- 您从未进行过完全数据库备份。

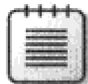
在以上任何一种情况下，每次数据库的事务日志“完全满”（稍后讲解）时，SQL Server 都会截断它。会认为数据库在自动截断模式中。

记住，截断表示在最早的活动事务以前的所有日志记录都无效，而且把不包含活动日志任何部分的所有 VLF 都标记为可重用，但这不表示物理日志文件压缩。而且，如果复制时数据库是发布者，那么最早的打开事务可能是标记为复制但是还没有被复制的事务。

“完全满”表示日志记录的数量比在系统启动过程中、在合理的时间（*恢复间隔*）内能够重做的数量多。可以用 *sp_configure* 存储过程或 SQL Server Management Studio（在第 1 章中讨论的）手动修改恢复间隔，但最好让 SQL Server 自动调节这个值。大多数情况下，将恢复间隔的值设为 1 分钟。默认情况下，*sp_configure* 显示 0 分钟，表示 SQL Server 会自动调节这个值。SQL Server 根据 1 分钟内可恢复 10MB 事务的估算来确定恢复间隔。

实际的日志截断是由检查点过程激活的，该过程通常呈睡眠状态，只在需要时才唤醒。每次用户线程调用日志管理器时，日志管理器都会检查日志的大小。如果大小超过了可以在恢复间隔内恢复的工作量，就会唤醒检查点线程。检查点线程为数据库生成检查点，然后截断日志的非活动部分。

此外，如果数据库在自动截断模式时日志已满 70%，那么日志管理器就会唤醒检查点线程，强制生成检查点。让日志增长比截断日志昂贵得多，所以 SQL Server 在任何可能的时候都会截断日志。



注意：

如果从不需要日志管理器，就不会激活检查点进程，也就永远不会发生截断。如果数据库在自动截断模式，即事务日志有状态为 2 的 VLF，那么直到数据库需要一些日志记录活动时，才会看到状态变为 0。

如果定期截断日志，到达物理日志文件的结尾时，SQL Server 就可以通过回到较早的 VLF 来重用物理文件中的空间。事实上，SQL Server 回收日志文件中恢复或备份不再需要的空间。我的 *AdventurWorks2008* 数据库就是这个状态，因为我从来没有做过完全数据库备份。

4.2.4 维护可恢复日志

如果*正在*维护日志备份序列，那么直到完全备份完最小 LSN 之前的那部分日志以后，才能覆盖这些日志记录。在进行日志备份以前，VLF 状态都保持为 2。日志备份以后，状态变成 0，SQL Server 可以循环到文件的起始处。图 4-4 用简化的方式描述了这个循环。从以前 *AdventureWorks2008* 数据库输出中的 *FSeqNo* 值可以看到，SQL Server 不总是以物理顺序重用日志文件。

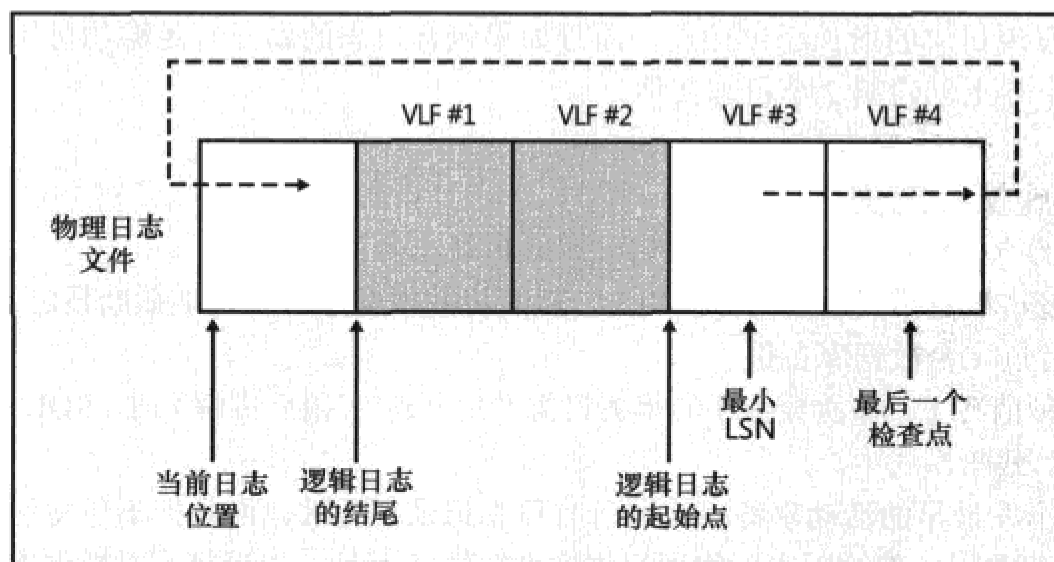
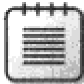


图 4-4 日志的活动部分循环到物理日志文件的起始点



注意：

如果数据库不处于自动截断模式，也没有进行定期日志备份，就永远不截断事务日志。如果只进行完全数据备份，则必须手动截断日志，使日志保持在可管理的大小。

判断数据库是否在自动截断模式的最简单方法是使用名为*sys.database_recovery_statur*的目录视图并查看称为*last_log_backup_lsn*的列。如果该列的值为空，数据库就是处于自动截断模式。

其实可以在*pubs*数据库（在本段结尾处显示）中运行一个简单的脚本，观察在和不在自动截断模式的数据库之间的区别。只要从来没有完全备份*pubs*数据库，这个脚本就能工作。如果从未对*pubs*做过任何修改而且是用*Instpubs.sql*脚本安装*pubs*的，事务日志文件的大小就只有约0.75MB，这是创建时的大小。下列脚本在*pubs*数据库中创建一个新表、插入3条记录，然后更新这些记录1000次。每次更新都是单独的事务，每次都写入事务日志中，但是应注意到日志根本不增长，而且甚至写了3000个更新记录以后，VLF的数量也没有增长（如果已经备份了*pubs*，可能想要在尝试此例子之前重新创建该数据库，做法是再次运行*Instpubs.sql*脚本，可以从随附网站*http://www.SQLServerInternals.com*下载该脚本）。虽然VLF的数量不会改变，但是会看到*FSeqNo*的值发生改变。在生成日志记录的过程中，随着每个VLF的重用，日志会得到新的*FSeqNo*值：

```
USE pubs;
-- First look at the VLFs for the pubs database
DBCC LOGINFO;
-- Now verify that pubs is in auto truncate mode
SELECT last_log_backup_lsn
FROM master.sys.database_recovery_status
WHERE database_id = db_id('pubs');
GO
CREATE TABLE newtable (a int);
GO
INSERT INTO newtable VALUES (10);
INSERT INTO newtable VALUES (20);
INSERT INTO newtable VALUES (30);
GO
SET NOCOUNT ON
DECLARE @counter int;
SET @counter = 1 ;
WHILE @counter < 1000 BEGIN
   UPDATE newtable SET a = a + 1;
   SET @counter = @counter + 1;
END;
```

确保数据库不在SIMPLE恢复模式以后，现在来做*pubs*数据库的备份。我会在本章后面讨论恢复模式，但现在只能通过执行以下命令来保证*pubs*在合适的恢复模式下。

```
ALTER DATABASE pubs SET RECOVERY FULL;
```

可以把下面语句中的路径改成SQL Server安装路径或任何备份位置的路径来进行备份：

```
BACKUP DATABASE pubs to disk =
'c:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\backup\pubs.bak';
```

只要做了完全备份，就能验证数据库不在自动截断模式，还是通过查看 *database_recovery_status* 视图：

```
SELECT last_log_backup_lsn
FROM master.sys.database_recovery_status
WHERE database_id = db_id('pubs');
```

这次应该会得到非空的 *last_log_backup_lsn* 值，表示预期的日志备份。然后从 *DECLARE* 语句开始，再次运行更新脚本。应该会看到物理日志文件已经增长，以包括所添加的日志记录，而且有更多的 VLF。不能重用日志中的初始空间，因为 SQL Server 假设您会为了备份事务日志而保留该信息。

现在可以尝试再次压缩日志。需要做的第一件事是截断日志，可以通过将恢复模型设置为 SIMPLE 来实现，如下所示：

```
ALTER DATABASE pubs SET RECOVERY SIMPLE;
```

如果在这之后发出以下命令，或者如果向 *pubs* 数据库发出 *DBCC SHRINKDATABASE* 命令，SQL Server 就会压缩日志文件：

```
DBCC SHRINKFILE (2);
```

此时应该注意到日志文件的物理大小减小了。如果没有发出任何收缩命令而截断日志，那么 SQL Server 会把被截断的日志所用的空间标记为可重用，但并不更改物理文件的大小。

SQL Server 7.0 中首次引入了日志架构，完全按照指定方法运行之前的命令并不总能压缩物理日志文件。如果日志文件不压缩，那是因为日志的活动部分位于物理文件的结尾处。物理压缩只能在日志的结尾处发生，活动的部分永远不能压缩。为了解决这个问题，可以在截断日志之后输入一些空事务，强制将日志的活动部分移到文件的起始处。SQL Server 7.0 以后的版本则不需要这个过程。如果已经发出压缩命令，在内部截断日志会生成一系列 NO-OP（虚拟）日志记录，它们强制活动日志从文件的物理结尾处移开。只要不再需要日志，就会发生压缩。

4.2.5　自动压缩日志

记住，截断并不是压缩。应该截断数据库，这样数据库才具有最大可压缩性。如果日志处于自动截断模式，而且设置了自动压缩选项，就会按固定间隔实际压缩日志。

如果已经打开数据库的自动压缩选项，那么每 30 分钟就会启动一个自动压缩进程（在第 3 章讨论的），而且会确定应该把日志压缩到多大。日志管理器累计自动压缩进程之间的 30 分钟间隔内所使用的最大日志空间量的统计信息。自动压缩进程将日志的压缩点标记为实际所使用的最大日志空间的 125%和日志的最小大小这两个数中较大的那个（最小大小是创建日志时的大小或者已经被手动增大或压缩过的大小）。如果有机会（被截断或被备份时）的话，日志就会压缩到这个大小。数据库不用运行在自动截断模式也可能会自动压缩，虽然无法保证日志真的会压缩。例如，如果从未备份过日志，那么没有任何 VLF 被标记为可重用，也不会发生压缩。

最后还需要注意的是，正因为数据库在自动截断模式，所以无法保证日志不增长（这与可确定的“如果数据库不在自动截断模式，日志*就会*增长”的观点正相反）。*自动截断*的意思只将那些被认为是可恢复的 VLF 标记为在定期间隔内可重用，但不影响处于活动状态的 VLF。如果有长时间一直运行的事务（可能是某人忘记提交的事务），那么从该事务开始以后、包括任何日志记录的所有 VLF 都被认为是活动的，都永远不能重用。未提交事务可能会使本来很容易管理的事务日志的大小发生变化，并且日志使用的空

间可能会比数据库本身更大，而且它还会继续增长。

4.2.6 日志文件大小

可以通过运行命令 *DBCC SQLPERF*（'*logspace*'）查看所有数据库日志文件的当前大小，以及已经使用的日志文件空间的百分比。但因为这是 DBCC 命令，所以很难通过筛选行来仅得到单个数据库的行。可以改为使用动态管理视图 *sys.dm_os_performance_counters*，然后取得每个数据库日志已满的百分比数：

```
SELECT instance_name as [Database],
       cntr_value as "LogFullPct"
FROM sys.dm_os_performance_counters
WHERE counter_name LIKE 'Percent Log Used%'
    AND instance_name not in ('_Total', 'mssqlsystemresource')
    AND cntr_value > 0;
```

筛选掉未报告日志文件大小的数据库还需要最后一个条件，这些数据库包括任何由于没有被恢复或者在可疑状态而不可用的数据库，以及没有事务日志的任何数据库快照。

4.3 备份和还原数据库

现在您可能已经知道了，本书不是为数据库管理员写的“如何做”书籍。补充内容中的参考文献列出了几本非常好的书，介绍了备份和还原数据库的机制，还为您的组织制定备份和还原计划提供了最佳实践。尽管如此，一些与备份和还原过程有关的重要问题能帮您理解为什么一个备份计划可能比另一个更能满足您的需要。这些问题中的大部分都涉及事务日志在备份和还原操作中所扮演的角色，所以我会在本节讨论一些主要问题。

4.3.1 备份类型

不管在数据库系统上实现了多少容错，都不能代替定期备份。备份可以为意外或错误的数据修改、编程错误和自然灾害（如果在远程位置存储备份）提供解决方案。如果选择以容错为代价获得可能的最快数据文件访问速度，备份会在数据文件损坏的情况下提供保证。而且，备份也是管理复制数据库到其他机器或其他实例的首选方法。

如果用备份恢复丢失的数据，潜在的可恢复数据量取决于备份类型。SQL Server 2008 有 4 种主要的备份类型（和这些类型的几种变形）。

完全备份。完全数据库备份基本上是把所有的页从数据库复制到备份设备上，备份设备可以是本地或网络磁盘文件，也可以是本地磁盘驱动器。

差分备份。差分备份只复制最后一次完全备份之后所更改的那些内容。内容会被复制到指定的备份设备上。SQL Server 通过检查数据库中每个数据文件的差分更改映射（DCM）页面上的位，快速判断需要备份哪些扩展。DCM 页是大位图，其中每位代表文件的一个扩展，就像我在第 3 章中讨论的全局分配映射（GAM）和共享全局分配映射（SGAM）页面那样。每次进行完全备份时，DCM 中的所有位都被清零。扩展中任何页面发生更改时，它在 DCM 页中的相应位就变成 1。

日志备份。大多数情况下，日志备份会复制最后一次完全备份或日志备份之后写入到事务日志中的所有日志记录，但是 *BACKUP LOG* 命令的确切行为取决于数据库的恢复模式设置。稍后我会简要介绍恢复模式。

文件和文件组备份。尤其是对非常大的数据库来说，要用文件和文件组备份来增加时间安排和媒体处理的灵活性（与完全备份相比）。文件和文件组备份也对包含具有各种更新特性的大数据库有用，就是说一些文件组允许读和写，而有一些是只读的。

更多信息：

对于定义备份设备、做备份和安排定期备份的机制的全部详细信息，请参考 *SQL Server 联机丛书*或联机补充内容的参考文献中所列的 SQL Server 管理书籍之一。

正在使用数据库时可以进行完全备份，这被认为是“模糊”备份——它不是任何特定时刻数据库状态的确切映像。备份线程只复制扩展，如果在备份的过程中有其他进程需要更改这些扩展，备份线程也可以这么做。

为了保持完全、差分或文件备份的一致性，SQL Server 会记录备份开始和结束时的当前日志序列号（LSN），这也能让备份获得日志的相关部分。相关的部分从最早的活动事务（第一个记录的 LSN）开始，到第二个记录的 LSN 结束。

正如之前讲的，和日志备份一起记录的内容取决于所使用的恢复模型。所以我会在详细讨论日志备份之前讲解恢复模型。

4.3.2　恢复模型

正如在第 3 章讨论数据库选项时讲到的，RECOVERY 选项有 3 个可能的值：FULL、BULK_LOGGED 和 SIMPLE。所选的值决定事务日志的大小、事务日志备份的速度和大小（或者是否确实能够进行日志备份），以及在媒体失败时能够在多大程度上承受丢失已提交事务的风险。

1. FULL 恢复模型

在存在受损数据文件的情况下，FULL 恢复模型的丢失工作风险最低。如果数据库处于此模式，那么会完全记录所有操作。就是说除了记录用 *INSERT* 操作添加的每个行、用 *DELETE* 操作删除的每个行、用 *UPDATE* 操作更改的每个行之外，SQL Server 还用 *bcp* 或 *BULK INSERT* 操作在事务日志中写入被添加的每一行。如果遇到数据库文件媒体故障并需要恢复 FULL 恢复模式的数据库，而且在完全备份后一直定期备份事务日志，就可以恢复到最后一次日志备份为止的任何指定时刻。而且如果数据文件失败后，日志文件还可用，就可以在失败之前还原到最后一个提交的事务。SQL Server 2008 还支持一个称为*日志标记*的功能，它能让您在事务日志中放置参考点。如果数据库处于 FULL 恢复模式，可以选择恢复到这些日志的标记之一。

在 FULL 恢复模式下，SQL Server 还完全记录 *CREATE INDEX* 操作。从包括索引创建的事务日志备份进行恢复时，恢复操作会更快，因为不需要重新编译索引——已经捕获了所有的索引页并作为数据库备份的一部分。在 SQL Server 2000 之前，SQL Server 只记录已经编译索引这件事，所以从日志备份恢复时就要从头编译整个索引。

所以 FULL 恢复模式看起来很好，对吗？毫无例外，这需要权衡。最大的权衡是事务日志文件可能会大得惊人，因此可能要比 SQL Server 2000 以前的版本花费更长时间进行日志备份。

2. BULK_LOGGED 恢复模型

BULK_LOGGED 恢复模型允许在媒体失败时完全恢复数据库，带来最佳性能，而且对于某些批量操

作还能使用最少的日志空间。在 FULL 恢复模式下会完全记录这些操作，但是在 BULK_LOGGED 恢复模式下只会记录最少的部分。这比正常的日志记录更有效，因为通常将数据写到用户数据库时，必须写入磁盘两次：一次写到日志中，另一次写到数据库本身。这是因为数据库系统使用一个撤销/重做日志，所以能够在需要时回滚或重做事务。最小日志记录只记录那些需要回滚事务的信息，但不支持时间点恢复。这些批量操作如下。

- *SELECT INTO*
  - 这个命令总是在默认文件组中创建新表。
- 批量导入操作，包括：
  - *BULK INSERT* 命令；
  - *bcp* 可执行文件。
- 用 *OPENROWSET*（*BULK*…）函数选择数据时，使用 *INSERT INTO*…*SELECT* 命令。
- 当把超过相当于一个扩展的数据插入到非聚集索引且使用了 TABLOCK 提示的表中时，使用 *INSERT INTO*…*SELECT* 命令。如果目标表为空，它可以有聚集索引。如果已经填充了目标表，就不能有聚集索引（这个选项可能对于在具有最小化日志记录的非默认文件组中创建新表有用。*INSERT INTO* 命令不允许指定文件组）。
- 部分更新具有大值数据类型的列（这将在第 7 章中讨论）。
- 当插入或追加新数据时，在 *UPDATAE* 语句中使用.WRITE 语句。
- 在将新数据插入或追加到 LOB 数据列（文本、ntext 或图像）时使用 *WRITETEXT* 和 *UPDATETEXT* 语句。
  - 更新现有数据时，在这些情况下不使用最小日志记录。
- 索引操作。
  - *CREATE INDEX*，包括视图上的索引。
  - *ALTER INDEX REBUILD* 或 *DBCC DBREINDEX*。
  - *DROP INDEX*。创建新堆是记录最小化操作，但页面释放总是会被完全记录下来。

在执行这些批量操作之一时，SQL Server 只记录操作发生的事实和关于空间分配的信息。SQL Server 2008 数据库中的每个数据文件都至少有一个特殊页，称为批量更改映射（Bulk Changed Map，BCM）页，或者称为最小记录映射（ML Map）页，这和第 3 章中讲的 GAM 和 SGAM 页的管理很像，也和之前提到的 DCM 页面很像。BCM 上的每位代表一个扩展，如果该位为 1，就表示在最后一次事务日志备份之后，最小记录批量操作更改过该扩展。BCM 页位于每个数据文件的第 8 页，之后每 511 230 页有一个 BCM 页。每次备份日志时，BCM 页中的所有位都重置为 0。

因为能够最小化记录日志批量操作，所以在 FULL 恢复模式下执行操作本身可能会快得多，但不能保证速度一定会提高。最小记录操作唯一保证的是日志本身更小。其实在某些情况下，最小记录可能会比完全记录操作慢。虽然不需要写那么多日志记录，但使用最小记录以后，SQL Server 会强制在事务提交之前将数据页刷到磁盘。尤其在这些页的 I/O 为随机时，强制刷数据页可能会非常昂贵。可以将它与完全记录相比较，完全记录总是顺序 I/O。如果没有快速 I/O 子系统的话，可能会很明显感觉最小记录比完全记录慢。

通常，最小记录不表示不记录，也不最低限度记录所有操作。最小记录是前面所讲最小化的操作记录量的功能。如果您有高性能的 I/O 子系统，性能可能也会提高。但是在低端电脑上，最小记录的操作会比完全记录操作慢。

如果数据库在批量记录模式，而且没有进行任何批量操作，那么可以将数据库恢复到任何一个时间点或命名的日志标记，因为日志包含对数据所做的所有更改的完整顺序记录。

在备份日志的过程中需要权衡使用小一些的日志。SQL Server 除了将事务日志的内容复制到备份媒体，还扫描 BCM 页并备份所有被修改的扩展和事务日志本身。日志文件本身很小，但日志备份可能会大很多倍，所以备份日志会花更多时间，可能会比 FULL 恢复模式占用的空间多得多。还原在 BULK_LOGGED 恢复模式下备份的日志与还原在 FULL 恢复模式下备份的日志所用的时间差不多。不需要重做操作，恢复所有数据和索引结构所需的所有信息都在日志备份中。

3. SIMPLE 恢复模型

SIMPLE 恢复模型提供了最简单的备份和还原策略。无论何时发生检查点（以固定的、频繁的时间间隔），都会截断事务日志。因此，唯一可以备份的类型是那些不需要日志备份的类型。这些备份类型是完全数据库备份、差分备份、部分完全和差分备份，以及为只读文件组所做的文件组备份。在 SIMPLE 恢复模式，尝试进行备份会出错。因为备份不需要日志，所以日志包括的所有事务提交或者回滚以后，就能立即重用日志的一些部分，而且服务器或事务失败也不再需要用事务进行恢复。实际上，只要将数据库改成 SIMPLE 恢复模型，就会截断日志。

记住，SIMPLE 记录并不等于不记录。因为从不需要担心日志备份，所以“简单”的是备份策略。即使一些操作在 SIMPLE 模式下记录可能比在 FULL 模式下记录的日志少一些，但所有操作都是在简单 SIMPLE 模式下记录。SIMPLE 模式下，数据库日志可能不会像在 FULL 模式下增长得那么快，因为本章前面讨论的批量操作也是 SIMPLE 模式下最小化记录的。但这并不表示不需要担心 SIMPLE 模式下日志的大小。就像在任何恢复模式下那样，不能截断活动事务的日志记录，也不能截断在最先打开的事务之后启动的任何事务的日志记录。因此如果有大的或长时间运行的事务，还是需要很多日志空间的。

4. 与数据库选项的兼容性

微软公司在 SQL Server 2000 中引入了这些恢复模型，并想让它们代替 *select into/bulkcopy* 和 *trunc. log on chkpt* 数据库选项。SQL Server 7.0 和更早的版本需要设置 *select into/bulkcopy* 选项才能进行 SELECT INTO 或批量复制操作。*trunc. log on chkpt* 选项强制在每次数据库中出现检查点时截断事务日志。只推荐使用这个选项进行测试或开发系统，而不是生产服务器。仍然可以用 *sp_dboption* 过程设置这些选项，但是不能用 *ALTER DATABASE* 命令设置这些选项。您将会看到，在 SQL Server 7.0 以后的版本中，用 *sp_dboption* 更改这些选项之一也会更改恢复模型，更改恢复模型又会更改这些选项之一或全部的值。更改数据库恢复模式的推荐做法是使用 *ALTER DATABASE* 命令。

```
ALTER DATABASE <database_name>
    SET RECOVERY [FULL | BULK_LOGGED | SIMPLE]
```

要了解数据库所处的模式，可以查看 *sys.databases* 视图。例如，这个查询返回恢复模式和 *AdventureWorks2008* 数据库的状态：

```
SELECT name, database_id, suser_sname(owner_sid) as owner ,
        state_desc, recovery_model_desc
FROM sys.databases
WHERE name = 'AdventureWorks2008'
```

正如我提到的，可以通过更改数据库选项来更改恢复模式。例如，如果数据库处于 FULL 恢复模式，并将 *select into/bulkcopy* 选项改为 *true*，那么数据库恢复模式就会更改为 BULK_LOGGED。相反地，如果用 *ALTER DATABASE* 强制数据库改回完全模式，那么 *select into/bulkcopy* 选项的值就会更改。如果使用 SQL Server 2008 标准版或企业版，*model* 数据库在 FULL 恢复模式启动，因此所有的新数据库也会在 FULL 模式。可以用 *ALTER DATABASE* 命令更改 *model* 数据库或任何其他的用户数据库的模式。

为了最大限度地利用事务日志，可以在 FULL 和 BULK_LOGGED 模式之间切换，而不用担心日志脚本失败。在 SQL Server 2000 以前，一旦执行 *SELECT INTO* 命令或批量复制，就不能再备份事务日志。所以如果已经安排好自动让日志备份脚本以固定的间隔运行，备份脚本就会停止，还会出错。不会再发生这样的事了。可以在任何恢复模式下运行 *SELECT INTO* 或批量复制，也可以在 FULL 或 BULK_LOGGED 模式备份日志。如果经常在 FULL 模式操作，偶尔需要进行快速批量操作的话，可能需要在 FULL 和 BULK_LOGGED 模式之间切换。可以更改为 BULK_LOGGED，然后在备份日志时再付出代价：备份会更大且需要更长的时间。

如果想维护日志备份序列，就不能再轻松地来回切换到 SIMPLE 模式。切换到 SIMPLE 模式没问题，但是切换回 FULL 或 BULK_LOGGED 就需要计划备份策略，并知道到这一时刻为止都没有日志备份。因此用 *ALTER DATABASE* 命令从 SIMPLE 改成 BULK_LOGGED 时，应该首先做一个完全数据库备份，让行为中的变化得以完成。记住在 SIMPLE 恢复模式下，会以固定的时间间隔截断事务日志。不推荐对生产数据库使用 SIMPLE 恢复模式，因为生产数据库需要最大的事务可恢复性。SIMPLE 模式真正有用的唯一一种情况是测试和开发，或者对于主要为只读的小数据库。我建议您为生产数据库使用 FULL 或 BULK_LOGGED，无论何时需要在这些模式之间切换都可以这么做。

4.3.3 选择备份类型

如果您负责为数据创建备份计划，那么不仅需要选择恢复模型，还要选择所做备份的类型。我提到了 3 种主要类型：完全、差分和日志。实际上，这 3 种类型可以一起使用。要想对数据库进行任何类型的完全恢复，必须偶尔进行完全数据库备份，用这些备份作为其他备份类型的起点。而且，可能要在差分备份、日志备份或两者的组合之间做出选择。下面是后面两种类型的特点，可根据这些特点进行选择。

差分备份

- 如果环境需要大量更改相同的数据，用差分备份更快。差分备份只备份最近的更改，日志备份则捕获每次单独的更新。
- 差分备份为新索引捕获整个 B 树结构，日志备份则捕获构造索引的每个单独步骤。
- 差分备份是累积的。从媒体失败恢复时，只需恢复最近的差分备份，因为它包括自最后一次完全数据库备份之后所做的所有更改。

日志备份

- 允许备份到任何时刻，因为它顺序记录所有更改。
- 只要日志可用，就能在数据库媒体失败后进行日志备份，这样正好可以恢复到失败的那个时刻。如果数据库本身不可用，那么最后一次日志备份（称为日志结尾）必须在 *BACKUP LOG* 命令中指定 WITH NO_TRUNCATE 选项。
- 日志备份是顺序且离散的。每个日志备份都包括完全不同的日志记录。媒体失败后使用日志备份恢复数据库时，必须以生成备份的顺序应用所有的日志备份。

记住第 1 章中简要讨论过，备份可以创建为压缩备份，这样可以大大减少在备份设备上实际创建备

份（完全、差分或日志）所需的时间和空间量。压缩备份的算法和用于行或页数据压缩的算法大大不同。我会在第 7 章详细说明这些区别。

4.3.4 还原数据库

进行每种备份的频率决定两件事：能够还原数据库的速度和对被还原的事务的控制程度。思考图 4-5 中的时间安排，它显示每周日完全备份数据库。日志为每天备份，差分备份在每周二和每周四进行。周五发生驱动器失败。如果失败不包括日志文件，或者如果已经用 RAID 1 镜像了它们，就应该用 NO_TRUNCATE 选项备份日志的尾部。

> **警告：**
>
> 如果在 BULK_LOGGED 恢复模式下运行，备份日志也会备份用 *BULK_LOGGED* 操作更改的任何数据，所以备份日志的尾部可能不仅需要日志文件，还需要任何包括由最小记录操作插入的数据的文件组。

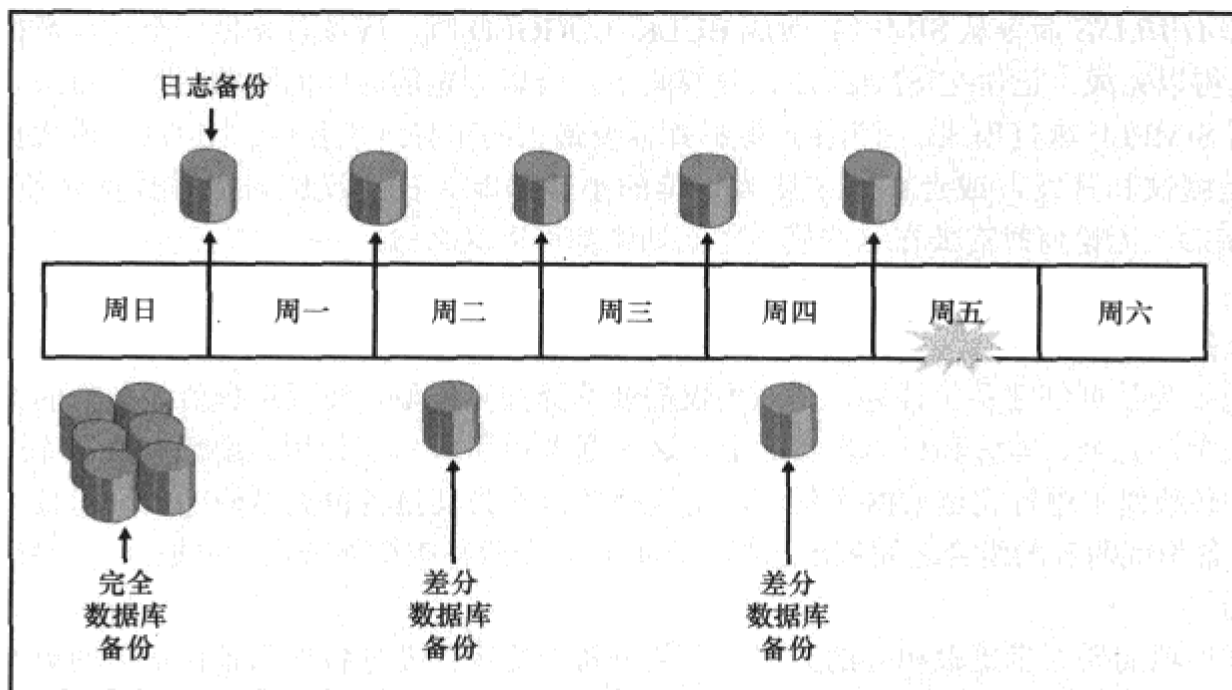


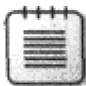
图 4-5 结合使用日志和差分备份可以减少总还原时间

为了在失败后还原此数据库，必须从恢复周日所做的完全备份开始。这样会做两件事：从备份媒体把所有数据、索引扩展和所有日志块复制到数据库文件，并在日志中应用所有事务。您必须决定是否回滚不完全事务。可以选择使用 *RESTORE* 命令的 WITH RECOVERY 选项恢复数据库，这会回滚任何不完整的事务并打开数据库供使用。除此之外不能进行更多还原。如果通过指定 WITH NORECOVERY 选项选择不回滚不完整事务，数据库就不一致且不可用。

如果选择 WITH NORECOVERY，则可以应用下一次备份。在图 4-5 描述的情况下，会恢复周四所做的差分备份，这会将所有已更改扩展复制回数据文件。差分备份还包括跨过生成差分备份时间段的日志记录，所以要决定是否恢复数据库。总是会回滚已完成的事务，但是您可以决定是否回滚未完成的事务。

还原差分备份后，必须按顺序还原差分备份之后所做的所有日志备份。如果能够完成最后这个备份，

它会包括失败之后备份的那个日志的尾部。



注意：

还原恢复（媒体恢复）与本章前面说明的重启恢复类似，但还原恢复是个 *REDO-only*（只重做）操作。还原恢复包括一个确定需要完成多少工作量的分析步骤，还包括一个前滚步骤，重做已完成的事务并将数据库还原到备份完成时的状态。和重启还原恢复不一样的是，您能控制回滚步骤何时完成。不应该在应用所有备份的所有前滚操作之前进行还原恢复。一旦指定了 *RESTORE WITH RECOVERY*，就会在重做步骤后重启数据库，SQL Server 会运行重启恢复以撤销未完成的事务。而且，SQL Server 可能需要在恢复完成后对元数据做些调整，所以直到完成恢复的所有阶段，才允许访问数据库。换言之，*RESTORE* 没有“快速”恢复这个选择。

1. 备份与还原文件和文件组

SQL Server 2008 允许备份单个文件或文件组，这对包括极大数据库的环境有用处。因为可以选择每天只备份一个文件或文件组，不需要经常备份整个数据库。当单个驱动器上有隔离的媒体失败而且认为还原整个数据库会花很长时间时，这可能也有用。

下面是需要记住的关于备份和还原文件和文件组的一些细节。

- 因为必须在恢复文件或文件组之后应用日志备份，所以具有*读—写*属性的单个文件和文件组只能在数据库是 FULL 或 BULK_LOGGED 恢复模式才能备份，而且也不能在 SIMPLE 模式进行日志备份。可以在 SIMPLE 模式备份只读文件组和其中的文件。
- 可以从完全数据库备份还原单个文件或文件组备份。
- 必须在快要还原单个文件或文件组之前备份事务日志。从进行文件或文件组备份时起，就必须具有不间断的日志备份链。
- 还原文件或文件组备份后，必须还原在备份和还原文件或文件组之间的所有事务日志。这保证所有被还原的文件与数据库的其余部分同步。

例如，假设周一上午 10 点备份文件组 *FG1*。数据库仍然在使用中，*FG1* 中的数据在改变，还在处理更改 *FG1* 和其他文件组中数据的事务。您下午 4 点备份日志，然后处理了更多更改 *FG1* 和其他文件组中数据的事务。下午 6 点发生媒体失败，丢失了构成 *FG1* 的一个或多个文件。

要想还原，必须先备份包含下午 4 点和 6 点之间发生的所有更改的日志尾部。日志尾部是用特殊的 WITH NO_TRUNCATE 选项备份的，但也可以使用 NORECOVERY 选项。用 WITH NO_TRUNCATE 选项备份日志尾部时，会把数据库设置为 RESTORING 状态，可以防止意外的后台更改干扰还原顺序。

然后可以用 *RESTORE DATABASE* 命令指定文件组 *FG1* 来还原 *FG1*。因为已还原的 *FG1* 只有到上午 10 点的更改，所以从 10 点到下午 6 点数据库还有变化。SQL Server 知道最后一次更改什么时候写入数据库，因为数据库中的每个页面存储更改该页面的最后一个日志记录的 LSN。恢复文件组时，SQL Server 记录数据库的最大 LSN。必须等日志至少达到数据库中的最大 LSN 再恢复日志备份，直到应用下午 6 点的日志备份时才能到达这个点。

2. 部分备份

部分备份可以基于完全或差分备份，但是部分备份不包含所有文件组。部分备份包含主要文件组和

所有读—写文件组中的所有数据，而且也可以指定备份任何只读文件。如果将整个数据库标记为只读的话，那么部分备份只包括主要文件组。部分备份对于使用简单恢复模型的超大型数据库（VLDB）来说特别有用，因为可以只备份特殊的文件组，甚至可以不备份日志。

3. 页面还原

也可以用 SQL Server 2008 还原单个页面。SQL Server 检测到损坏页时，会把它标记为可疑，并将关于该页的信息存储在 *msdb* 数据库中的 *suspect_pages* 表中。

发生下述活动时，会检测到损坏页：

- 需要读页面的查询；
- 正在运行 *DBCC CHECKDB* 或 *DBCC CHECKTABLE*；
- 正在运行 *BACKUP* 或 *RESTORE*；
- 正在尝试用 *DBCC DBREPAIR* 修复数据库。

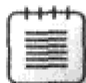
有几种类型的错误需要将页标记为可疑，并输入 *suspect_pages* 表。这些错误包括校验和错误、残缺页错误及内部一致性问题（如页眉中有坏页 ID）。*suspect_pages* 表中的 *event_type* 列指示页状态的原因，它通常反映该页被输入到 *event_type* 表的原因。*SQL Server 联机丛书*列出了 *event_type* 列的可能值。

event_type 值	说　明
1	由操作系统 CDC 错误引起的 823 错误，或者除坏校验和与残缺页（例如，坏页 ID）之外的 824 错误
2	坏校验和
3	残缺页
4	已还原（在标记该页为坏之后已经还原）
5	已修复（DBCC 修复了改页）
7	被 DBCC 释放

suspect_pages 表中记录的一些错误可能是瞬态错误，如由于电缆断开所产生的 I/O 错误。具有合适权限的人（如 *sysadmin* 服务器角色中的人）可以从 *suspect_pages* 表中删除行。而且，不是所有插入 *suspect_pages* 表中的错误都需要还原该页。缓存数据中出现的问题（如未聚集索引）可以通过重新编译索引来解决。如果 *sysadmin* 删除未聚集索引并重新编译它，那么尽管损坏数据已经修复，也不会在 *suspect_pages* 表中显示为已经修复。

页面还原是专门用来替换那些因无效的校验和或残缺写操作而标记为可疑的页。虽然可以一次还原多个数据库页，但是也不要大量更换页。如果确实有很多损坏页，应该考虑还原整个文件或数据库。此外，应该尝试确定错误的原因，如果发现挂起设备出现故障，应该将全部文件或数据库还原到新位置。必须在页面还原（让新页与数据库的其余部分一样新）之后进行日志还原。就像文件还原那样，把日志备份应用到包含被恢复页的数据库文件上。

在联机页面还原时，数据库在还原的过程中是联机的，而且只有被还原的数据脱机。注意，不是所有的已损坏页都可以用数据库联机来还原。



注意：

只有 SQL Server 2008 企业版才允许联机页面还原。

*SQL Server 联机丛书*列出了以下基本页面还原步骤。

（1）获得要还原的已损坏页面的页 ID。校验和或残缺页写错误会返回页 ID，页 ID 是指定页所需的信息，也可以从 *suspect_pages* 表得到页 ID。

（2）用包含即将被还原页面的完全、文件或文件组备份来还原页。在 *RESTORE DATABASE* 语句中，用 PAGE 语句列出所有即将被存储的页 ID。在单个文件中可以还原的最多页数为 1 000。

（3）应用被还原页所需的任何可用差分备份。

（4）应用后续的日志备份。

（5）创建一个包括被还原页的最终 LSN（即最后一个被还原页脱机的时刻）的新数据库日志备份。最终 LSN（它设置为顺序中的第一个还原的一部分）是重做目标 LSN。联机前滚包括该页面的文件可以在重做目标 LSN 处停止。要想知道文件的当前重做目标 LSN，可查看 *sys.master_files* 的 *redo_target_lsn* 列。

（6）还原新的日志备份。一旦应用了这个新的日志备份，就完成了页面还原，并能够使用页。所有的坏页都受日志还原的影响。所有其他页的页眉中都有最新的 LSN，没什么可以重做的。而且，页面级还原不需要 UNDO 语句。

4. 部分还原

SQL Server 2008 允许在紧急情况下对数据库做部分备份。虽然说明和语法与文件和文件组的备份和还原相似，但还是有很大的区别。用文件和文件组还原，从一个完整的数据库开始，然后用之前备份的版本替换一个或多个文件或文件组。用部分数据库还原的话，不是从整个数据库开始，而是将单个的文件组（必须包含所有系统表的主文件组）还原到新位置。在尝试引用存储在不还原的文件组上的数据时，这些文件组被当成脱机来对待。然后可以还原日志备份或差分备份，从而将这些文件组中的数据还原到后来的某个时刻。这样您可以选择在意外删除或修改表数据之后，从一个表集还原数据。可以用部分还原数据库从丢失的表中提取数据，然后将其复制回原始数据库中。

5. 备用还原

在正常的还原操作中，可以选择运行还原来回滚未完成的事务或者根本不运行还原。如果运行还原，就不能还原更多的日志备份，数据库也完全可用。如果不运行还原，数据库则是不一致的，而且 SQL Server 根本不会让您使用。由于生成日志备份的方式是这样，所以您得二选一。

例如，在 SQL Server 2008 中，日志备份不会重叠——每个日志备份都从前一个日志备份结束的地方开始。思考对一个表进行几百次更新的事务。如果在更新的过程中备份日志，那么在这之后，第一个日志备份包括事务的起点和一些更新，而第二个日志备份包括更新的其余部分和提交部分。假设在还原整个数据库以后需要还原这些日志备份。如果在还原第一个日志备份之后运行还原，那么会回滚事务的第一部分。然后如果尝试还原第二个日志备份，由于第二个备份从事务的中部开始，所以 SQL Server 没有关于事务起点的信息。您当然不能从这一时刻还原事务，因为这些操作可能会依赖于已经部分丢失的更新。因此 SQL Server 不允许进行更多的还原。另一种做法是不运行还原、不回滚到事务的第一部分，而是让事务就这样不完整。SQL Server 考虑到数据库是不一致的，所以最终在其上运行还原以前都不允许用户访问的数据库。

如果想结合这两种方法，那怎么办呢？尤其是如果在尝试生成时间点还原但又不知道正确的时间点时，如果能还原一个日志备份并在还原更多的日志备份以前能查看数据就好了。SQL Server 提供了一个称为 STANDBY 的选项来还原数据库，还能还原更多日志备份。如果还原日志备份并指定 WITH

STANDBY = '<*some filename*>'，SQL Server 就会回滚未完成的事务，但是在指定文件（称为*备用文件*）中记录回滚工作。下一个还原操作读取备用文件的内容，并重做已回滚的操作，然后还原下一个日志。如果该还原还指定 WITH STANDBY，那么会再次回滚未完成的事务，但会保存这些已回滚事务的记录。记住，用 WITH STANDBY 还原后，就不能修改任何数据（如果尝试修改数据的话，SQL Server 会生成错误消息），但是可以读数据并继续还原更多的日志。必须用 WITH RECOVERY 还原最后一个日志（不会保存备用文件），才能让数据库完全可用。

4.4　小结

除了一个或多个数据文件，SQL Server 实例中的每个数据库都有一个或多个日志文件，记录对数据库所做的更改（记住，数据库快照没有日志文件，因为从来都不会直接修改快照）。SQL Server 用事务日志来保证数据在逻辑上和物理上的一致性。而且，管理员可以备份事务日志，从而使还原数据库更有效。管理员或数据库的所有者也可以设置数据库的恢复模式，以确定存储在事务日志中的细节详细到什么程度。

第 5 章 表

Kalen Delaney

在本章中，我们首先对表做一个基本介绍，然后对表的内部结构进行非常详细的分析。简单说来，*表*就是具有很多不同命名*属性*（如数量或类型）的特殊*实体*（一个人、一个地方或一件事）的一个数据集合。表通常是 Microsoft SQL Server 和关系模型的核心。在 SQL Server 中，一张表通常是指一个*基表*，用于强调数据存储的地方。称之为基表也可以使之与*视图*相区分，视图是一张虚拟表，是参照一张或多张基表或其他视图的一个内部查询。

表数据的属性（如颜色、尺寸、数量、订购日期和供应商名称）在表中以命名列的形式存在。表中数据的每个实例都由单个条目或*行*（正式的名称是 *tuple*）表示。在一个真实的关系数据库中，表中的每一行都是唯一的并具有一个被称为*主键*的唯一标识符（与 ANSI SQL 标准相一致，SQL Server 不需要您使一行唯一或声明一个主键。但是由于这两个概念对关系模型都非常关键，因此建议您总是这样做）。

大部分表都与其他表有一些关联。例如，在一个典型的订购目录系统中，*order* 表有一个 *customer_number* 列记录某一订单的顾客数量，*customer_number* 还显示在 *customer* 表中。假设 *customer_ number* 是 *customer* 表的一个唯一标识符或主键，则会在 *order* 和 *customer* 表中建立一种外键关联，从而将这两个表连接起来。

30 秒的数据库设计入门介绍就到这里。您可以找到很多介绍逻辑数据库和表设计的书，但是这本书则与众不同。我们假设您已经了解了基本的数据库理论和设计知识，同时您大致知道了表的样子。本章剩下部分将介绍 SQL Server 2008 中表的内部特性。

5.1 创建表

为了创建一张表，SQL Server 使用 ANSI SQL 标准 *CREATE TABLE* 语法。SQL Server 管理工具提供了一种前端填空表设计器，有时能够简化工作。最后，SQL Server 语法总会被发送到 SQL Server 中来创建一张表，不管您使用哪种界面。在本章中，我们将强调直接使用数据定义语言（Data Definition Launguage，DDL）而不是讨论 GUI 工具。您应该在一个脚本中编写所有的 DDL 命令，以便以后更轻松地运行它们来重新创建表（即使您使用一种友好的前端工具，这一点对于以后重新创建表也很重要）。Management Studio 和其他前台工具可以利用创建对象所必需的 SQL DDL 命令创建并保存操作系统文件。这种 DDL 是基本的源代码，而且您也应该这样看待 DDL。您还应该考虑利用一种源控制产品（如 Microsoft Visual SourceSafe）将这些文件保存在版本控制下。

从基本层面上来说，创建一张表只需要知道如何命名、它包含哪些列及每一列可以存储哪些值（域）。这是在 *dbo* 架构中创建 *customer 表*的基本语法，使用 3 个固定长度的字符（*char*）列（注意这种表的定义不一定是存储数据最有效的方式，因为每条数据总是需要 46 字节同时还有几个一般性的字节，不论数据的实际长度如何）。

```
CREATE TABLE dbo.customer
(
```

```
name          char(30),
phone         char(12),
emp_id        char(4)
);
```

这个例子在单独一行上显示一列，从而增强了可读性。就 SQL Server 的分析程序而言，由 tab、回车符和空格键创建的空格是一样的。从系统的角度而言，下面的 *CREATE TABLE* 示例与前面的示例是一样的，但是从用户的角度来说却很难阅读。

```
CREATE TABLE customer (name char(30), phone char(12), emp_id char(4));
```

5.1.1 命名表和列

一张表总是在一个数据库的一个架构中创建的。表也有所有者，但是与 SQL Server 2005 以前的版本不同的是，2008 版表的所有者不用于访问表。架构用于所有对象访问。正常情况下，一张表会在正在创建表的用户的默认架构中创建，但是 *CREATE TABLE* 语句可以指定将被创建的对象位于哪个架构中。用户可以仅在该用户有 *ALTER* 权限的一个架构中创建一张表。具有 *sysadmin*、*db_ddladmin* 或 *db_owner* 角色的任何用户都可以在任何架构中创建表。表的完整名称由 3 部分组成，形式如下：

```
database_name. schema_name.table_name
```

3 部分命名规范中的前两部分具有默认值。数据库名称的默认值是您当前正在操作的数据库上下文。schema_name（架构名称）在查询时实际上有两种可能默认值。如果引用一个对象时没有指定架构名称，则 SQL Server 首先会检查默认架构中的一个对象。如果在默认架构中没有这样的表，那么 SQL Server 接下来会检查 *dbo* 架构中是否有一个指定名称的对象。

注意：

要访问默认架构或 *dbo* 架构之外的某个架构中的一张表或其他对象，必须在表名称中包含架构名称。事实上，您应该养成提及 SQL Server 2008 中的任何对象时总包含模式名称的习惯。这样不仅可以避免出现架构混淆，而且能够带来某些性能上的优势。

sys 架构是一个特例。对于兼容性视图（如 *sysobjects*），SQL Server 在访问可能已经创建的同名对象之前先访问 *sys* 架构中的对象。显然，将自己创建的某个对象命名为 *sysobjects* 不好，因为您永远都不能访问这个对象。兼容性视图也可以通过 *dbo* 架构访问，因此 *sys.sysobjects* 和 *dbo.sysobjects* 是同一个对象。对于分类视图和动态管理对象来说，必须指定 *sys* 架构来访问对象。

您应该使列名称具有描述性，同时由于您会反复使用列名称，因此应该避免使用冗长的列名称。列名称（或 SQL Server 中的任何对象，如一张表或一个视图）可以是您选择的任意字符串，只要符合 SQL Server 标准标识符的命名规则就可以：必须是由 1 到 128 个字母、数字、#、$、@或_组合而成。

更多信息：

也可以选择使用包括您喜欢的任何字符在内的分隔标识符。有关标识符命名规则的更多信息，请参阅 *SQL Server 联机丛书*中的“标识符”部分。这里的规则适用于所有 SQL Server 对象名称，不仅仅是列名称。

在某些情况下，您可以使用 4 部分名称访问表，其中第一部分是 SQL Server 实例的名称。但是，只有 SQL Server 实例被定义为一个链接服务器时，才需要使用 4 部分名称来访问表。可以在 *SQL Server 联机丛书*上阅读更多关于链接服务器的知识，这里我们不做过多的介绍。

5.1.2 保留关键字

某些保留关键字（如 *TABLE*、*CREATE*、*SELECT* 和 *UPDATE*）对于 SQL Server 解析器来说具有特殊意义，它们共同构成了 SQL 语言实现。您应该避免使用保留关键字作为对象名称。除 SQL Server 保留关键字之外，SQL-92 标准有自己的保留关键字列表。在某些情况下，这个列表比 SQL Server 列表的限制更严格，但有时也没有 SQL Server 列表限制的那样严格。*SQL Server 联机丛书*包括这两个列表。

注意 SQL-92 的保留关键字。有些字还不是 SQL Server 中的保留关键字，但是它们可能在将来的 SQL Server 版本中成为保留关键字。如果您使用一个 SQL-92 的保留关键字，而该字成为 SQL Server 的一个保留关键字，那么您可能必须要在升级之前修改应用程序。

5.1.3 分隔标识符

不能在对象名称中使用关键字，除非您使用一个分隔标识符。事实上，如果您使用一个分隔标识符，则不仅可以利用关键字作为标识符，还可以使用其他任何字符串作为对象名称——不论该字符串是否符合标识符规则。其中包括空格和其他通常不允许的非字母数字字符。分隔标识符有两种类型：

- 带括号的标识符，由方括号分隔（[*对象名称*]）；
- 带引号的定界符，由双引号分隔（"*对象名称*"）。

在任何环境中都可以使用带括号的标识符，但如果要使用带引号的标识符，则必须利用 SET QUOTED_IDENTIFIER ON 语句启动一个特殊的选项。如果开启 QUOTED_IDENTIFIER，则双引号会被解释为引用一个对象。为了分隔字符串或日期常量，必须使用单引号。

现在让我们来看一些实例。由于 *column* 是一个保留关键字，因此下面的第一条语句在所有情况下都是不合法的。第二条语句在 QUOTED_IDENTIFIER 被开启之前都是不合法的。第三条语句在所有情况下都合法。

```
CREATE TABLE dbo.customer(name char(30), column char(12), emp_id char(4));

CREATE TABLE dbo.customer(name char(30), "column" char(12), emp_id char(4));

CREATE TABLE dbo.customer(name char(30), [column] char(12), emp_id char(4));
```

SQL Native Client ODBC 驱动程序和 SQL Native Client OLE DB Provider for SQL Server 自动在连接时将 QUOTED_IDENTIFIER 设置为 ON。可以在 ODBC 数据源、ODBC 连接属性或 OLE DB 连接属性中对这一项进行配置。可以通过执行如下查询语句为会话确定是开启还是关闭该选项：

```
SELECT quoted_identifier
FROM sys.dm_exec_sessions
WHERE session_id = @@spid;
```

返回值为 1 表示 QUOTED_IDENTIFIER 被开启。如果正在使用 Management Studio，可以通过在一个查询窗口中运行前面的命令，或者从"工具"菜单中选择"选项"，然后展开查询执行 SQL Server 节点并检查 ANSI 属性信息的方式来检查这项设置，如图 5-1 所示。

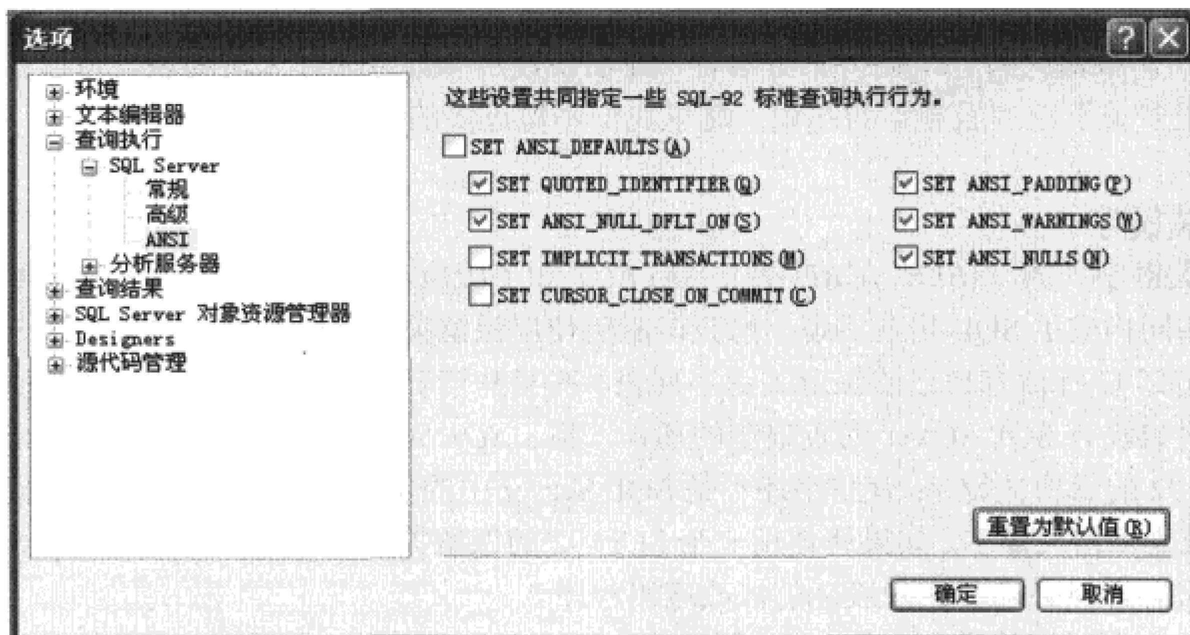


图 5-1 在 Management Studio 中检查某个连接的 ANSI 属性

提示：

从技术上讲，您可以对所有对象和列名应用分隔标识符，因此永远不必担心保留关键字。但是我不推荐这样做。很多为 SQL Serer 提供的第三方工具不能很好地处理引号标识符，并且会使代码很难阅读。使用带引号的标识符也可能使升级为 SQL Server 将来的版本变得更困难。

不要用分隔标识符来解决保留关键字的问题，您应该简单地采取一些简单的命名规范。例如，可以在列名称之前加上表名称和一个下划线作为前几个字母。这种命名风格可以使列或对象名称更容易阅读，同时也会大大地降低关键字冲突或保留字冲突的可能性。

5.1.4 命名约定

很多组织及多用户开发的项目都采用标准命名约定。这一般来说是一种好的习惯。例如，分配一个标准的名称 *cust_id* 来表示每个表中的客户编号可以清晰地说明所有表都共享公共数据。如果一个组织在表中使用几个名称来表示客户编号，如 *cust_id*、*cust_num*、*customer_number* 和 *customer_#*，这些名称就不会明显地表示公共的数据了。

一种命名约定是匈牙利风格的列名称表示法。匈牙利风格的表示法是 C 语言中广泛采用的一种惯例，其中变量名称包括数据类型信息。这种表示法使用 *sint_nn_custnum* 这样的名称来说明 *custnum* 列是一种短整型（2 字节的 *smallint* 类型）而且是 NOT NULL（不允许为空）的。虽然这种习惯在 C 语言编程中很不错，但是却破坏了 SQL Server 提供的数据类型独立性。因此建议您不要采用这种命名约定。

5.1.5 数据类型

SQL Server 提供了很多数据类型，其中大部分都是很简单的。选择合适的数据类型可以很容易地将要存储的值域映射到相应的数据类型。在选择数据类型时，需要在避免浪费存储空间的同时为应用程序整个生命周期内一系列可能值提供足够的空间。讨论使用各种数据类型进行编程的所有可能情况的细节不是本书所讨论的范围。在很大程度上，我们只会介绍与处理各种数据类型相关的基本问题。

1. 选择一种数据类型

每一列使用什么类型的数据主要取决于该列存储的数据的特性及要在数据上执行的操作。SQL Server 2008 中的 5 种基本数据类型分别是数值、字符、日期和时间、大型对象（Large Object，LOB）和杂项。SQL Server 2008 还支持一种称为 *sql_variant* 的可变数据类型。存储在 *sql_variant* 列中的值几乎可以是任何数据类型。我们将在第 7 章中介绍 LOB 列，因为它们的存储格式与本章讨论的其他数据类型不同。在这一部分我们将介绍与存储不同数据类型的数据相关的问题。

2. 数值数据类型

您应该对希望执行数值比较或算数操作的数据使用数值数据类型。您要确定的主要是希望能够存储的最大范围值及所需的精确性。这种方案的缺点是存储更大值范围的数据类型将占用更多的空间。

数值数据类型也能够被分成精确或近似两类。精确数值保证存储数字的精确表示。近似数值具有更大的值范围，但是值不一定被精确地存储。精确数值可以存储的最大数据范围是 - 10^38 + 1～10^38 - 1。除非您需要更大的数字，否则建议您不要使用近似数值数据类型。

精确数值数据类型可以分为两组：整数和小数。整数类型的长度为 1～8 字节，根据可能取值的范围而相应地增加。整数类型中经常包括 *money* 和 *smallmoney* 数据类型，因为在内部它们是以相同的方式存储的。对于 *money* 和 *smallmoney* 数据类型来说，认为最右面的 4 个数字位于小数点之后。对于其他整数类型来说，没有数字位于小数点之后。表 5-1 列出了整数数据类型及其存储容量和值范围。

表 5-1　整型数据类型的范围和存储需求

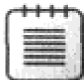
数据类型	范　围	存储（字节）
bigint	-2^{63}～$2^{63}-1$	8
Int	-2^{31}～$2^{31}-1$	4
Smallint	-2^{15}～$2^{15}-1$	2
Tinyint	0～255	1
Money	–922 337 203 685 477.5808～922 337 203 685 477.5807，精确度为一个货币单位的万分之一	8
Smallmoney	–214 748.3648～214 748.3647，精确度为一个货币单位的万分之一	4

小数和数值数据类型允许高精确度和较大的值范围。对于这两种统一的数据类型来说，您可以指定*精度*（存储的总位数）和*小数位数*（小数点右边的最大位数）。可以存储在小数点左边的数字最大位数是精度–小数位数（即从精度中减去小数位数得到的位数）。两种不同的小数值可以有相同的精度和完全不同的范围。例如，定义为小数的某一列（8, 4）可以存储–9 999.9999～9 999.9999 之间的数值，定义为小数（8, 0）的某一列可以存储–99 999 999～99 999 999 之间的值。

表 5-2 显示了已定义精度的小数和数值数据所需的存储空间。

表 5-2　小数和数值数据类型的存储需求

精　度	存储（字节）
1～9	5
10～19	9
20～28	13
29～38	17



注意:

SQL Server 2005 SP2 增加了一项功能，允许小数数据存储在可变空间中。当列中有一些值需要高精度但大部分值只需要几字节、0 或 NULL 时，这一项功能非常有用。与 *varchar* 不同的是，*vardecimal* 不是一种数据类型，而是利用 *sp_tableoption* 程序设置的表的一个属性，在 SQL Server 2005 中，必须为数据库启用这一属性。在 SQL Server 2008 中，除 *master*、*model*、*tempdb* 和 *msdb* 之外的所有数据库都允许表启用 *vardecimal storage format* 属性。

虽然 *vardecimal* 存储格式可以降低数据的存储容量，但是却以增加额外的 CPU 开销为代价。一旦启用某个表的 *vardecimal* 属性，则表中所有 *decimal* 数据都存储为可变长度数据。这包括关于 *decimal* 数据的所有索引及包括 *decimal* 数据的所有日志记录。

修改表的 *vardecimal storage format* 属性的值是一种脱机操作，SQL Server 独占锁定正在被修改的表，直到所有 *decimal* 数据转换成新的格式。*Vardecimal* 存储格式已经不推荐使用了，因此我们不会介绍 *vardecimal* 数据的内部存储。对于新的开发，建议您使用 SQL Server 的压缩功能使需要可变字节的数据的存储需求降到最低。我们将在第 7 章中讨论数据压缩。

3. 日期和时间数据类型

SQL Server 2008 支持 6 种存储日期和时间信息的数据类型：从第一版 SQL Server 开始就一直可用 *datetime* 和 *smalldatetime*，以及在 SQL Server 2008 中新增的 4 种类型（*date*、*time*、*datetime2* 和 *datetimeoffset*）。这些类型之间的区别在于可用的日期范围、所需存储空间的字节数、是否同时存储日期和时间(或者只是日期或者只是时间)及时区信息是否整合到存储值中。表 5-3 取自 *SQL Server 2008 Books Online*，显示了每种日期和时间数据类型的范围和存储需求。

表 5-3 **SQL Server 日期和时间数据类型范围和存储需求**

类型	格　式	范　围	精确度	存储容量(字节)	用户定义秒小数精度
time	*hh:mm:ss [.nnnnnnn]*	00:00:00.0000000～23:59:59.9999999	100 纳秒	3～5	是
date	*YYYY-MM-DD*	0001-01-01～9999-12-31	1 天	3	否
smalldatetime	*YYYY-MM-DD hh:mm:ss*	1900-01-01～2079-06-06	1 分钟	4	否
datetime	*YYYY-MM-DD hh:mm:ss [.nnn]*	1753-01-01～9999-12-31	0.00333 秒	8	否
datetime2	*YYYY-MM-DD hh:mm:ss [.nnnnnnn]*	0001-01-01 00:00:00.0000000 ～9999-12-31 23:59:59.9999999	100 纳秒	6～8	是
datetimeoffset	*YYYY-MM-DD hh:mm:ss [.nnnnnnn] [+|-]hh:mm*	0001-01-01 00:00:00.0000000 ～9999-12-31 23:59:59.9999999 (in UTC)	100 纳秒	8～10（2 字节时区数据）	是

如果没有提供日期，则默认设置为 1900 年 1 月 1 日。如果没有提供时间，则默认设置为 00:00:00.000（半夜）。

注意：

如果您对 SQL Server 日期和时间数据很陌生，那么您可能会惊奇地发现，对于原始 *datetime* 数据类型来说，可以存储的最早日期是 1753 年 1 月 1 日。这是历史原因造成的，从原始 Sybase 的 *datetime* 数据类型规范开始。在我们有时所说的“西方世界”中，有两种日历，分别是公历和阳历。这种日历是很多天的分离（根据您所看的世纪的不同），因此当一种文化由公历转为日历时，它们要从日历中减去 10 到 13 天。大英帝国在 1752 年进行了这种转换，这一年的 9 月 2 号之后就是 9 月 14 号。Sybase 决定不存储 1753 年之前的日期，因为日期的算术运算是模糊的。但是，其他国家/地区在其他时间进行日历转换，土耳其到 1927 年才进行这种转换。

就内部而言，所有日期和时间数据类型值的存储方式与您输入数据的方式或数据显示的方式完全不同。日期和时间总是作为两个独立的部分存储：一个日期部分和一个时间部分。

对于原始的 *datetime* 数据类型 *datetime* 和 *smalldatetime* 来说，数据在系统内部是作为两部分来存储的。对于 *datetime* 值来说，数据存储为两个 4 字节的值，第一部分（日期）是基准日期 1900 年 1 月 1 日之前或之后的天数，第二部分（时间）是半夜后的时钟嘀嗒数，每嘀嗒一次表示 3.33 微秒，即 1/300 秒。可以将一个 *datetime* 值转换成一个 8 位十六进制字节的二进制串。对于 *smalldatetime* 值来说，每一部分都存储在 2 字节中。日期存储为自 1900 年 1 月 1 日的天数，时间存储为半夜后的分钟数。

下面的实例显示了如何查看存储在一个 *datetime* 类型的变量中、利用无参系统函数 CURRENT_TIMESTAMP 检索的当前日期和时间的组成部分。第 1 个 CONVERT 操作表示存储 *datetime* 值的完整十六进制字节串。第 2 个 CONVERT 显示第 1 个 4 字节转换成一个整数，第 3 个 CONVERT 显示转换成一个证书的第 2 个 4 字节。由于我们是在一个本地变量中存储当前日期和时间的，因此可以确定所有 CONVERT 操作使用相同的值：

```
DECLARE @today datetime
SELECT @today = CURRENT_TIMESTAMP
SELECT @today AS [CURRENT TIMESTAMP];
SELECT CONVERT (varbinary(8), @today) AS [INTERNAL FORMAT];
SELECT CONVERT (int, SUBSTRING (CONVERT (varbinary(8), @today), 1, 4))
        AS [DAYS AFTER 1/1/1900];
SELECT CONVERT (int, SUBSTRING (CONVERT (varbinary(8), @today), 5, 4))
        AS [TICKS AFTER MIDNIGHT];
These are the results when the code runs on July 10, 2008:
CURRENT TIMESTAMP
-----------------------
2008-07-10 17:29:11.967
INTERNAL FORMAT
------------------
0x00009AD501202BD6

DAYS AFTER 1/1/1900
-------------------
39637
```

```
TICKS AFTER MIDNIGHT
--------------------
18885590
```

Microsoft利用在SQL Server 2008中添加新的*date*和*time*数据类型的机会彻底修改了日期和时间的内部表示。日期现在存储为一个3字节正数，表示0001年1月1日之后的天数。对于*datetimeoffset*类型来说，另外两字节用于存储一个自UTC以来的时间偏移（以小时和分钟的形式）。注意，虽然系统内部新日期和时间类型的基准日期是0001年1月1日，但是当SQL Server解释一个没有指定的真实日期值时，默认值还是1900年1月1日。例如，如果您试图向一个*datetime2*类型的列插入字符串'01:15:00'，那么SQL Server会将其解释为1900年1月1日1：15分。

所有包含时间信息的新数据类型（*time*、*datetime2*和*datetimeoffset*）都允许通过在数据类型名称后面跟随一个1～7之间的数字来表示所需保留字的方式来指定时间部分的精度。如果不指定保留字，则默认值为7。表5-4显示了每个可能的小数位数值在存储数据值的精确度和存储需求方面的含义。

表5-4　　具有存储存储需求和精度的时间日期的小数位数值

指定的小数位数	结果（精度,小数位数）	列长度（字节）	秒的小数位（精度）
无	(16,7)	5	7
(0)	(8,0)	3	0～2
(1)	(10,1)	3	0～2
(2)	(11,2)	3	0～2
(3)	(12,3)	4	3～4
(4)	(13,4)	4	3～4
(5)	(14,5)	5	5～7
(6)	(15,6)	5	5～7
(7)	(16,7)	5	5～7

要快速查看表5-4中信息的含义，可以运行下面的3条转换语句：

```
SELECT CAST(CURRENT_TIMESTAMP AS time);
SELECT CAST(CURRENT_TIMESTAMP AS time(2));
SELECT CAST(CURRENT_TIMESTAMP AS time(7));
```

结果如下，注意小数位数值决定了小数位数，*time*的该值与*time(7)*一致。

```
17:39:43.0830000
17:39:43.08
17:39:43.0830000
```

在系统内部，时间是利用下面的公式计算的，假设*H*代表小时，*M*代表分钟，*S*代表秒，*F*是小数部分，*D*是小数位数：

```
(((H * 60) + M) * 60 + S) * 10D + F
```

例如，格式为*time(2)*的值17:39:43.08在系统内部将存储为：

```
(((17 * 60) + 39) * 60 + 43) * 102 + 083, or 6358383
```

同样，存储为 *time(7)*将是：

```
(((17 * 60) + 39) * 60 + 43) * 107 + 083, or 635830000083
```

在本章后面标题为“内部存储”这一节中，我们将看到数据存储在数据行时的显示效果。

SQL Server 2008 提供了很多操作 *date* 和 *time* 时间数据的函数及用于解释和显示日期和时间值的很多不同格式。在此级别讨论日期和时间数据超出了本书的范围。但是，理解这些类型最重要的就是要知道所看到的内容不是实际存储在磁盘上的内容。不论是使用原有的 *datetime* 和 *smalldatetime* 类型，还是使用任何其他新增类型，这种磁盘格式都是完全明确的，但是却不是非常友好。您需要确定所提供的输入数据的格式也是明确的。例如，'3/4/48'值就不明确。它是表示 3 月 4 日还是 4 月 3 日，是 1948 年还是 2048 年或 48 年（差不多是 2 000 年之前了）？ISO 8601 格式是一种具有明确规范的国际标准。此外，这种格式还不受 *SET DATEFORMAT 或 SET LANGUAGE* 会话设置的影响。使用这种格式可以将 1948 年 3 月 4 日表示成 19480304 或 1948-03-04。

4. 字符数据类型

字符数据类型有 4 种。它们可以是固定长度或长度可变的单字节字符组成的字符串（*char* 和 *varchar*），也可以是固定长度或长度可变的 Unicode 字符组成的字符串（*nchar* 和 *nvarchar*）。Unicode 字符串需要为每个被存储的字符提供 2 字节，当您需要表示不能以单字节字符（足以存储英语和欧洲字母表中的大部分字符）存储的字符时可以使用 Unicode 字符。单字节字符串最多能存储 8 000 个字符，Unicode 字符串最多能够存储 4 000 个字符。您应该知道正在处理的数据的类型，从而决定是使用单字节还是使用双字节字符串。注意目录视图 *sys.types* 以字节数形式而不是字符数形式报告长度。在 SQL Server 2005 和 SQL Server 2008 中，还可以定义一个具有最大长度的可变长度字符串。当实际长度小于或等于 8 000 字节时，定义为 *varchar(max)*的列被作为标准可变长度列对待，当实际长度大于 8 000 字节时，它们被作为一个大对象值（本章后面会提到，将在第 7 章中做详细介绍）对待。

决定是使用长度可变的数据类型还是固定长度的数据类型是比较困难的，有时可能不是很直观或不明显。一般来说，当您希望某一列中数据的大小有很大变化并且列中数据不会被频繁更改时，可变长度数据类型比较适合。

使用可变长度数据类型可以大大节省存储空间。有时可能会使性能稍微降低，但是在其他情况下可以提高性能。具有可变长度列的一行需要在系统内部维持特殊的偏移条目。这些条目跟踪列的实际长度。计算和维护偏移量比计算和维护一个纯固定长度的行（不需要这样的偏移）所需的系统开销稍多一些。这一任务需要进行一些加减操作来维持偏移值。不过，维护这些偏移的额外系统开销一般是无关紧要的，而且这种开销本身也不会对大部分系统造成影响（如果有的话）。

使用可变长度字段的另一个潜在性能问题是在一个几乎写满的页面上增加一行所带来的开销。如果具有可变长度列的某一行只使用其最大长度的一部分，并且以后会更新为更长的长度，则扩大后的行可能不再匹配原来的页面。如果表有一个聚集索引，则行必须位于相对于其他行的同一位置，因此解决方法就是将拆分页面分隔，并将具有扩大行的页面上的部分行移动到一个新链接页面上。这可能是一种代价非常高的操作。第 6 章将介绍拆分页面和移动行的详细内容。如果表没有聚集索引，则行可以移动到一个新位置并在原始位置留下一个转发指针。我们将在本章后面讨论转发指针。

另一方面，使用可变长度列有时会提高性能，因为可变长度列允许在一页上放置更多的行。但是效率不仅仅是简单地靠需要更少的磁盘空间来获取的。SQL Server 的一个数据页是 8KB（8 192 字节），其

中 8 096 个字节可以用于存储数据（其他字节留作系统内部使用，用于跟踪它所在页面和对象的结构信息）。一次 I/O 操作能够返回整个页面。如果可以在一个页面上放 80 行，则一次 I/O 操作返回 80 行。但是如果能在一页上放 160 行，则一次 I/O 操作的效率基本会提高两倍。在浏览数据及返回很多相邻行的操作中，这意味着可以明显提高性能。每一页上放的行越多，I/O 和缓冲区命中率就越高。

例如，假设有一个简单的 customer 表，而且您可以以固定长度和可变长度两种方式定义，如图 5-2 和图 5-3 所示。

```
USE testdb
GO

CREATE TABLE customer_fixed
(
cust_id                     smallint         NULL,
cust_name                   char(50)         NULL,
cust_addr1                  char(50)         NULL,
cust_addr2                  char(50)         NULL,
cust_city                   char(50)         NULL,
cust_state                  char(2)          NULL,
cust_postal_code            char(10)         NULL,
cust_phone                  char(20)         NULL,
cust_fax                    char(20)         NULL,
cust_email                  char(30)         NULL,
cust_web_url                char(100)        NULL,
)
```

图 5-2　具有固定长度列的一张 customer 表

```
USE testdb
GO

CREATE TABLE customer_var
(
cust_id                     smallint             NULL,
cust_name                   varchar(50)          NULL,
cust_addr1                  varchar(50)          NULL,
cust_addr2                  varchar(50)          NULL,
cust_city                   varchar(50)          NULL,
cust_state                  char(2)              NULL,
cust_postal_code            varchar(10)          NULL,
cust_phone                  varchar(20)          NULL,
cust_fax                    varchar(20)          NULL,
cust_email                  varchar(30)          NULL,
cust_web_url                varchar(100)         NULL
)
```

图 5-3　具有可变长度列的 customer 表

包含 addresse、name 或 URLs 的列，每一列数据的长度都有很大变化。让我们来看一下选择固定长度列和可变长度列的区别。图 5-2 中使用的都是固定长度的列，不管在行中实际插入的字符数是多少，每一行都占用 384 字节。SQL Server 还需要额外为表中的每一行提供 10 字节的系统开销，因此每一行都需要 394 字节的存储空间。但是即使表必须容纳指定大小的 addresse 和 name，平均行也只是最大大小的一半。

图 5-3 中假设对于所有可变长度（*varchar*）列来说，平均输入数据实际上只是最大长度的一半。一行长度不是 394 字节，平均长度是 224 字节。长度的计算方法如下：*smallint* 和 *char(2)*列的总长度是 4 字节。*Varchar* 列的总长度最大是 380，一半是 190 字节。每 9 个 *varchar* 列需要 2 字节的系统开销。为所有有一个或多个可变长度列的行再添加两个字节。此外，这些行同样需要 10 字节的额外开销，这是图 5-2 中固定长度的行所需要的，不论可变长度字段的情况如何。因此总共是 4+190+18+2+10=224 字节（我们将在本章后面讨论系统开销占用的每一个字节的实际意义）。

在图 5-2 中的固定长度示例中，一个数据页上总会有 20 行（8 096/394，不计余数）。在图 5-3 的可变长度示例中，每一页平均有 36 行（8 096/224）。使用可变长度列的表会占用大约一半的存储页，一次 I/O 操作几乎检索两倍的行数，而且在内存中缓存的页的命中率将提高两倍。

更多信息：

如果您正在使用快照分离，则需要为每一行提供额外的系统开销字节。我们将在第 10 章讨论这一并发选项，同时还会讨论支持这一选项所需的额外的行开销。

当您为列选择长度时，请不要浪费，但是也不要低估。为将来留出余地，并要意识到如果额外长度不改变每一页上的行数，则额外大小是自由的。然后再考虑一下图 5-2 和图 5-3 的示例。*cust_id* 被声明为 *smallint* 类型，意味着它能存储的最大正数是 32 767（但遗憾的是，SQL Server 不提供无符号 *int* 或无符号 *smallint* 数据类型），占用 2 字节的存储空间。虽然 32 767 个用户对于一个新公司来说看起来很多，但是公司可能会对自己的成功感到惊讶，而且几年之后发现 32 767 太有限了。

数据库设计人员可能对他们试图保存 2 字节及没有简单地使用一个 *int* 数据类型而感到遗憾，使用 4 字节但是最大正数值是 2 147 483 647。当他们意识到自己没有真正节省空间时，将会特别失望。如果您正在计算刚刚讨论过的每一页的行数，行大小增加 2 字节，您会发现每一页仍然具有相同的行数。额外 2 个字节是免费的——它们是以前浪费的空间。它们从来不会使固定长度示例中的每一页的行数更少，它们也很少在可变长度情况下使每页的行数更少。

哪种策略更好呢？会提供更好的更新性能？或者每页的行数更多？像这样的问题是没有一个正确答案的，具体情况要根据应用程序而定。如果权衡得当，则可以做出最好的选择。知道存在的问题后，值得重复一下通用规则：当您希望列中数据的长度有明显变化而且不会频繁更新时，则适合使用可变长度数据类型。

5. 字符数据排序规则

对于很多数据类型来说，比较和排序的规则是很简单的。不论您问谁，答案都是 12 大于 11。即使人们以不同的方式存入日期，2008 年 8 月 20 号永远与 2007 年 8 月 21 号不同。但是对于字符数据来说，这一原则就不适用。大多数人都会讲 *csak* 排在 *cukor* 的前面，但是在匈牙利词典中，顺序是相反的。*STREET* 等于 *Street* 吗？同样，带有附加符号（如重点符号或变音符号）的字符又如何排序？

由于不同用户有不同的喜好和需求，因此 SQL Server 中的字符数据总是与一种*排序规则*相关。排序规则就是一组定义字符数据如何排序和比较及语言依赖函数（如 *UPPER* 和 *LOWER*）如何工作的规则。排序规则还决定了单字节数据类型 *char*、*varchar* 和 *text* 的字符指令系统。SQL Server 中（即表名、变量等）的元数据也遵守排序规则。

确定使用哪种排序规则。您可以定义在 SQL Server 的多种级别上使用哪种排序规则。创建一张表时，

可以为每个字符列定义排序规则。如果不应用任何排序规则，则会使用数据库排序规则。

数据库排序规则还决定数据库中元数据的排序规则。因此在排序规则不区分大小写的数据库中，可以使用 *MyTable* 或 *MYTABLE* 引用已创建的名为 *mytable* 的表，但是在排序规则区分大小写的数据库中，必须以 *mytable* 形式引用。数据库排序规则还决定字符串常量和字符变量中数据的排序规则。

创建数据库时可以指定数据库排序规则。如果不指定，则会应用服务器排序规则。在一些限制很严格的情况下，允许用 *ALTER DATABASE* 语句修改数据库排序规则(一般来说，如果数据库中有任何 CHECK 约束，都不能修改排序规则)。这样将重新建构系统表以反映元数据中新的排序规则。但是，用户表中的列保持不变，您需要自己进行修改。对于所有限制的详细内容，请参阅 *SQL Server 联机丛书*中的 *ALTER DATABASE* 主题。

服务器排序规被系统数据库 *master*、*model*、*tempdb* 和 *msdb* 使用（另一方面，资源数据库总有相同的排序规则：Latin1_General_CI_AI）。服务器排序规则也是变量名称的排序规则，因此在一个排序规则不区分大小写的服务器上，@a 和@A 是同一个变量，但是如果服务器排序规则是区分大小写的，则这两个变量就是不同的。因此在启动时要选择服务器排序规则。

最后，可以利用 COLLATE 子句强制一个表达式中的排序规则。需要这样操作的一种情况是当同样的表达式包括具有不同排序规则的两个列时。这会导致*排序规则冲突*，SQL Server 要求您使用 COLLATE 子句进行解决。

可用的排序规则。要查看可用的排序规则，可以运行如下查询：

```
SELECT * FROM fn_helpcollations();
```

在 SQL Server 2008 的某个实例上运行这一查询时，结果将包含 2 397 种排序规则。还有 112 种排序规则已经被废弃，没有在 fn_helpcollations 中列出来。

排序规则主要分为两组：Windows 排序规则和 SQL Server 排序规则。SQL Server 排序规则主要是考虑到兼容问题而保留的原有排序规则。不过排序规则 SQL_Latin1_General_CP1_CI_AS 是最经常使用的一种，因为这是当您在将英语（美国）作为系统区域设置的机器上安装 SQL Server 时使用的默认排序规则。

Windows 排序规则。Windows 排序规则是由 Microsoft Windows 定义的。SQL Server 没有向外查询 Windows 的排序规则，而是 SQL Server 团队将排序规则的定义复制到了 SQL Server 中。Windows 中的排序规则一般用新版本的 Windows 修改，从而适应 Unicode 标准中的变化，同时由于排序规则决定数据在索引中出现的顺序，因此 SQL Server 不能接受由于您将数据库移动到不同的 Windows 版本上而造成的对排序规则变化的定义。

排序规则名称的分析。Windows 排序规则以族的形式出现，每族有 18 种排序规则。同一族中的所有排序规则都是以相同的排序规则指示符开始的，用于表示排序规则族所支持的语言或语言组。

排序规则指示符后面紧跟表示排序规则本质的标记。排序规则可以是一种二进制排序规则，此时标记是 BIN 或 BIN2。对于其他 16 种排序规则，标记是 CI/CS，表示区分大小写或不区分大小写。AI/AS 表示区分/不区分重音。KS 表示区分假名类型。WS 表示区分全半角。

如果 CI 是排序规则名称的一部分，则字符串 *smith* 和 *SMITH* 是相等的，但是如果排序规则名称中有 CS，则 *smith* 和 *SMITH* 是不同的。同样，如果排序规则是 AI，则 *cote*、*coté*、*côte* 和 *côté* 都是相同的，但是在一个 AS 排序规则中，它们是不同的。假名类型只与日文文字相关，在一个区分假名类型的排序规则中，katakana 和 hiragana 的对应部分被认为是不同的。区分全半角指东亚语言（其中存在半角和全角格式的一些字符）。KI 和 WI 标记不存在，但是如果没有 KS 和 WS，则暗示着不区分假名类型和全半角。

下面是排序规则名称的一些示例。

- **Latin1_General_CI_AS。**针对于英语、德语和意大利语等西欧语言提供的一种区分大小写、重音的排序规则。
- **Finnish_Swedish_CS_AS。**针对芬兰语和瑞典语的一种区分大小写和重音的排序规则。
- **Japanese_CI_AI_KS_WS。**一种不区分大小写和重音但区分假名类型和全半角的排序规则。
- **Turkish_BIN2。**针对土耳其语的一种二进制排序规则。

相同排序规则的不同版本。一种排序规则指示符可能包含一个指示排序规则被添加到的 SQL Server 版本的版本号信息。缺少版本号表示排序规则是 SQL Server 2000 中的原始排序规则；90 表示排序规则被添加到 SQL Server 2005 中；100 表示被添加到 SQL Server 2008 中。

SQL Server 2008 为已经存在排序规则的语言和语言组添加了新的排序规则。因此现在有 Latin1_General 和 Latin1_General_100、Finnish_Swedish 和 Finnish_Swedish_100 等排序规则对。

这些额外的排序规则反映了 Windows 中的变化。旧的排序规则是建立在 Windows 2000 排序规则基础上的，而新的_100 排序规则是建立在 Windows 2008 中的排序规则基础上的。

> **警告：**
>
> 如果打算从 SQL Server 2005 中以一个链接服务器的方式访问 SQL Server 2008，则应该避免使用新的_100 排序规则，因为如果试图从 SQL Server 2005 中访问这样一列时，将得到“遇到非法表格数据流（TDS）排序规则”的错误信息。

单字节字符类型。单字节字符数据类型 char、varchar 和 text 仅能表示 255 个字符，排序规则的代码页决定哪 255 个字符可用。在大部分代码页中，从 32 到 127 的字符都是相同的，取自 ASCII 标准，其余字符是根据特定语言区域选择的。例如，CP1252（通常所说的 Latin-1）支持英语、法语、瑞典语等西欧语言。CP1250 是针对西里尔脚本的，CP1251 是针对东欧语言的，等等。

对于 Windows 排序规则中的排序、比较、大小写等其他操作，规则与单字节数据类型和双字节 Unicode 数据类型完全相同。其中有一个特例：在一种二进制排序规则中，排序是按照字符编码进行的，单字节代码页中的排序可能与 Unicode 中的排序不同。例如，在一种波兰语排序规则中，*char(209)*打印输出 Ń（一个带有重音符的大写字母 N），而 unicode（N'Ń'）则打印输出 323，这是该字符的 Unicode 代码点（Unicode 中的代码点与 Latin 中的代码点一致，但是只适用于 160～255 之间的范围）。Microsoft 向自己的 Latin-1 版本中添加了一些额外的字符。其中的一个实例是 Euro(€)字符，这是基于 CP1252 排序规则中的 *char(128)*，但是 Unicode 代码点 128 是一个非打印字符，unicode(N'€')打印 8364。

有一些排序规则没有映射为单字节代码页。只能对 Unicode 数据类型使用这些排序规则。例如，如果运行代码

```
CREATE TABLE NepaleseTest
    (abc char(5) COLLATE Nepali_100_CI_AS NOT NULL);
```

则会得到如下错误消息：

```
Msg 459, Level 16, State 2, Line 1
Collation 'Nepali_100_CI_AS' is supported on Unicode data types only and cannot be applied
to char, varchar or text data types.
```

为了查看某种排序规则的代码页，可以使用 *collationproperty* 函数，如下面的示例中：

```
SELECT collationproperty('Latin1_General_CS_AS', 'CodePage');
```

将返回 1252。对于仅支持 Unicode 的某种排序规则来说，返回值为 0（如果返回结果为 NULL，则说明排序规则名称或 *CodePage* 单词拼写错误）。

不能使用仅支持 Unicode 的排序规则作为服务器排序规则。

排序次序。 排序规则决定排序次序。当一种 Windows 排序规则不敏感时（如不区分大小写或重音），则这种不敏感同样会应用到排序次序上。例如，在一种不区分大小写的排序规则中，大小写的区分不会影响数据的排序方式。在一种敏感的排序规则中，大小写、重音、假名类型及全半角都会影响排序，但是只是次级影响。也就是说只有不存在其他差异时，这些属性才会影响排序。

为演示这种效果，考虑下面的表：

```
CREATE TABLE #words (word nvarchar(20) NOT NULL,
                     wordno tinyint PRIMARY KEY CLUSTERED);
INSERT #words
  VALUES(N'cloud',  1), (N'CSAK',    6), (N'cukor',    11),
        (N'Oblige', 2), (N'Opera',   7), (N'Öl',       12),
        (N'résumé', 3), (N'RESUME',  8), (N'RÉSUMÉ',   13),
        (N'resume', 4), (N'resumes', 9), (N'résumés', 14),
        (N'OEIL', 5), (N'oeil', 10);
```

要查看一种排序规则的工作方式，我们使用这里显示的查询。首先查看常用的排序规则 Latin1_General_CI_AS：

```
WITH collatedwords (collatedword, wordno) AS (
  SELECT word COLLATE Latin1_General_CI_AS, wordno
  FROM #words
)
SELECT collatedword, rank = dense_rank() OVER(ORDER BY collatedword),
       wordno
FROM collatedwords
ORDER BY collatedword;
```

执行此查询将得到如下结果：

```
collatedword      rank    wordno
--------------    ------  ------
cloud             1       1
CSAK              2       6
cukor             3       11
Oblige            4       2
OEIL              5       5
oeil              5       10
Öl                6       12
Opera             7       7
RESUME            8       8
resume            8       4
résumé            9       3
RÉSUMÉ            9       13
resumes           10      9
résumés           11      14
```

rank 列给出排序次序的级别。我们可以看到，对于只有大小写不同的单词来说，级别是一样的。我们还可以从输出结构中看到，有时大写的显示在前面，有时小写的显示在前面。这完全是随意的，如果您自己运行此查询，则很有可能看到大小写单词的不同排序次序。

如果将排序规则修改为 Latin1_General_CS_AS，则会得到如下结果：

```
collatedword      rank    wordno
--------------    ------  ------
cloud             1       1
CSAK              2       6
cukor             3       11
Oblige            4       2
oeil              5       10
OEIL              6       5
Öl                7       12
Opera             8       7
resume            9       4
RESUME            10      8
résumé            11      3
RÉSUMÉ            12      13
resumes           13      9
résumés           14      14
```

现在所有输入的数据都具有不同的级别。在不存在其他差异时，小写形式显示在大写形式之前，因为在 Windows 排序规则中，小写总是比大写具有更低的二级权值。

现在让我们来看一下不同语言的情况。这里测试一下 Hungarian_CI_AI 排序规则：

```
collatedword      rank    wordno
--------------    ------  ------
cloud             1       1
cukor             2       11
CSAK              3       6
Oblige            4       2
OEIL              5       5
oeil              5       10
Opera             6       7
Öl                7       12
RÉSUMÉ            8       13
RESUME            8       8
résumé            8       3
resume            8       4
resumes           9       9
résumés           9       14
```

CSAK 和 *ÖL* 现在排在 *cukor* 和 *Opera* 之后。这是因为在匈牙利字母表中，CS 和 Ö 是独立的字母。您还会发现在这种 CI_AI 排序规则中，所有这 4 种 *résumé* 格式都具有相同的级别。

在这些实例中，列的数据类型是 *nvarchar*，但是如果将表改为 *varchar* 并重新运行实例，则会取得同样的结果。

字符范围和排序规则。排序次序不仅可以应用到 ORDER BY 子句，而且还可以应用到操作符（如>）及 LIKE 表达式中的范围。例如，查看下面的代码：

```
SELECT * FROM #words
```

```
WHERE word COLLATE Latin1_General_CI_AS > 'opera';
SELECT * FROM #words
WHERE word COLLATE Latin1_General_CS_AS > 'opera';
```

第一个 *SELECT* 列出了 6 个单词，而第二个 *SELECT* 列出了 7 个（因为在一种区分大小写的排序规则中，*Opera*>*opera*）。

如果您习惯使用其他语言中正则表达式的字符范围，那么在试图选择大写字母开头的单词时可能陷入如下陷阱：

```
SELECT * FROM #words WHERE word LIKE '[A-Z]%';
```

但是即使是在区分大小写的排序规则中，这行代码通常也会列出所有 14 个单词（在某些语言中，*Ö* 作为一个独立的字母排在 *Z* 之后，因此不会归为指定的范围）。*A*～*Z* 也受排序规则的限制。还有另一种结果：在列表中将 *cloud* 修改为 *aloud*。使用一种区分大小写的排序规则，*SELECT* 现在仅返回 13 行。由于 *a* 排在 *A* 的前面，因此 *A*～*Z* 不包括 *a*。

如您所见，结果可能有点让人疑惑。建议您使用字符数据时要非常小心。如果确实需要使用字符数据，一定要保证实际测试边界情况从而保证排除任何数据。

二进制排序规则。在一种二进制排序规则中，不存在二级权值，字符按照字符集中的代码点排序。因此使用前面实例中的 Latin1_General_BIN2 可以得到：

```
collatedword      rank    wordno
--------------    ------  ------
CSAK              1       6
Oblige            2       2
Opera             3       7
RESUME            4       8
RÉSUMÉ            5       13
cloud             6       1
cukor             7       11
resume            8       4
resumes           9       9
résumé            10      3
résumés           11      14
Öl                12      12
OEIL              13      5
oeil              14      10
```

现在大写首字母是 *C*、*O* 和 *R* 的单词排在小写首字母是 *c*、*o* 和 *r* 的单词前面，因为这是 ASCII 标准中的顺序。*ÖI* 和两种形式的 *oeil* 的代码点都在前 127 个 ASCII 代码以外，因此排在列表的后面。

由于二进制排序规则是基于代码点的，同时它们在单字节代码页和 Unicode 中可能有所不同，因此单字节 Unicode 数据类型的顺序可能不同。例如，如果将#words 数据类型修改为 *varchar*，并再次运行 Latin1_General_BIN2 排序规则示例，则会发现 *ÖI* 现在排在最后。

在前面的讨论中介绍了两种类型的二进制排序规则，分别是 BIN 和 BIN2。其中 BIN 排序规则是早期的排序规则，如果需要使用一种新开发的二进制排序规则，应该使用 BIN2 排序规则。为了理解 BIN 和 BIN2 之间的区别，需要查看二进制表示法中的一个 Unicode 字符串。例如，考虑：

```
SELECT convert(varbinary, N'ABC');
```

这段代码返回 0x410042004300。A 的 ASCII 代码是 65，十六进制形式为 41。在 Unicode 中，A 是 U+0041(Unicode 字符通常写为 U+XXXX，而 XXXX 是十六进制表示法中的代码点)。但是转换成 *varbinary* 后将显示为 4100。这是因为 PC 架构是从小到大的，即最不重要的字节会存储在最前面（具体原因不是本书讨论的范围)。

因此，为了按照代码点正确地排序 *nvarchar* 数据，SQL Server 不应该只看字节串，而应该既切换每个单词来获得正确的代码点。这正是 BIN2 排序规则所做的事情。旧的 BIN 排序规则仅对第一个字符执行这种交换，接下来对剩下的字符进行逐个字节的比较。为了说明两种类型二进制排序规则之间的区别及真正的字节排序，我们选择使用带有字符 Z（U+005A）和 Ń（带有重音的 N；U+0143）的如下示例：

```
SELECT n, str, convert(binary(6), str) AS bytestr,
        row_number() OVER(ORDER BY convert(varbinary, str))
           AS bytesort,
        row_number() OVER(ORDER BY str COLLATE Latin1_General_BIN)
           AS collate_BIN,
        row_number() OVER(ORDER BY str COLLATE Latin1_General_BIN2)
          AS collate_BIN2
FROM (VALUES(1, N'ZZZ'), (2, N'ZN´N´'), (3, N'N´ZZ'), (4, N'N´N´N´'))
     AS T(n, str)
ORDER BY n;
```

下面是结果：

```
n           str     bytestr          bytesort   collate_BIN  collate_BIN2
----------- ------  ---------------- ---------- ------------ ------------
1           ZZZ     0x5A005A005A00   4          2            1
2           ZN´N´   0x5A0043014301   3          1            2
3           N´ZZ    0x43015A005A00   2          4            3
4           N´N´N´  0x430143014301   1          3            4
```

您会发现在 *collate_BIN2* 列中，行是按照它们在 Unicode 中的代码点进行编号的。另一方面，在 *bytesort* 列中，它们是按照相反的顺序编号的，因为在字符代码中最不重要的字节优先级最高。最后，在 *collate_BIN* 列中，以 *Z* 开头的两条输入数据排在最前，但是就 *collate_BIN2* 而言，顺序则相反。

SQL Server 排序规则。SQL Server 排序规则（通常所说的 SQL 排序规则）是一种比 Windows 排序规则更小的组。总共有 76 种 SQL 排序规则，其中 1 种已经废弃。

SQL 排序规则使用两种不同的规则集。一种是针对单字节数据类型的，另一种是针对 Unicode 数据类型的。单字节数据类型的规则是由 SQL Server 本身定义的，从 SQL Server 不支持 Unicode 那天开始。使用 Unicode 数据时，SQL 排序规则使用相同的规则作为匹配的 Windows 排序规则。为了查看某一 SQL 排序规则与哪种 Windows 排序规则相匹配，可以从 *fn_helpcollations()*函数的输出结果中查看 *description* 列的信息。

SQL 排序规则的名称总是以 *SQL_* 开始并且后面紧跟一种语言指示符，与 Windows 排序规则的名称类似。同样，SQL 排序规则的名称还包括 CI/CS 和 AI/AS 来指示区分大小写和重音。也有一些二进制 SQL 排序规则。与 Windows 排序规则相比，SQL 排序规则总是在名称中包括单字节字符的代码页。由于某些原因，CP1252、Windows Latin-1 在名称中总是显示为 CP1。

很多 SQL 排序规则都是与美国国家标准委员会（ANSI）代码页相关的，也就是非 Unicode Windows 应用程序使用的代码页。但是也有针对 OEM 代码页 CP437 和 CP850 的 SQL 排序规则，即命令行窗口中

使用的代码页。甚至还有几种针对 EBCDIC 的 SQL 排序规则。

排序次序。使用 SQL 排序规则，根据数据类型的不同可能获得不同的结果。例如，在具有 14 个单词的实例中，如果以单词为 *nvarchar* 类型并且使用最常用的 SQL 排序规则 SQL_Latin1_General_CP1_CI_AS 运行，则结果与使用 Latin1_General_CI_AS 时相同。但是如果将单词修改为 varchar，则会得到如下结果：

```
collatedword      rank    wordno
--------------    ------  ------
OEIL              1       5
oeil              2       10
cloud             3       1
CSAK              4       6
cukor             5       11
Oblige            6       2
Ö                 1       12
Opera             8       7
RESUME            9       8
resume            9       4
résumé            10      3
RÉSUMÉ            10      13
resumes           11      9
résumés           12      14
```

现在两种形式的 *oeil* 都排在前面并且具有不同的级别，虽然排序规则是区分大小写的。在这种排序规则中，一些加重音的字母排序时好像是标点字符（其他的是 S、Y 和 Z）。在 SQL_Latin1_General_CP1_CI_AS 排序规则中，单字节和 Unicode 数据类型之间的其他差异包括标点字符的排序方式。不过，只要数据主要是由数字 0～9 和英文字母 A～Z 组成的，则这些差异的影响不会很大。

三级排序规则。与 Windows 排序规则一样，SQL 排序规则有主要和二级权值，但是并不是到此为止。总共有 32 种 SQL 排序规还有三级权值。除一个特例之外，其他所有三级排序规则都是区分大小写的。三级权值的目的是优先大写，因此当整个 ORDER BY 子句中的其他内容都相同时，大写单词排在前面。在一些三级排序规则中，是通过名称中出现的 Pref 指出的，而在其他三级排序规则中则是隐含的。可以在 *SQL Server 联机丛书*中关于内置函数 *TERTIARY_WEIGHTS* 的主题中看到三级排序规则的完整列表。

为了分析三级排序规则，我们使用具有不同单词的另一个表，如下所示：

```
CREATE TABLE #prefwords
           (word char(3) COLLATE SQL_Latin1_General_Pref_CP1_CI_AS
                         NOT NULL,
            wordno int NOT NULL PRIMARY KEY NONCLUSTERED,
            tert AS tertiary_weights(word));
CREATE CLUSTERED INDEX word_ix ON #prefwords (word);
--CREATE INDEX tert_ix on #prefwords(word, tert)
go
INSERT #prefwords (word, wordno)
   VALUES ('abc', 1), ('abC', 4), ('aBc', 7),
          ('aBC', 2), ('Abc', 5), ('ABc', 8),
          ('AbC', 3), ('ABC', 6);
go
SELECT word, wordno, rank = dense_rank() OVER(ORDER BY word),
       rowno = row_number() OVER (ORDER BY word)
FROM #prefwords
```

```
ORDER BY word--, wordno;
```

该查询的输出如下：

```
word      wordno   rank     rowno
------    -------- ------   -----
ABC       6        1        8
ABc       8        1        6
AbC       3        1        7
Abc       5        1        5
aBC       2        1        4
aBc       7        1        3
abC       4        1        2
abc       1        1        1
```

您可以发现所有单词都有相同的级别，但是，大写字母总是出现在小写字母之前。在 *rowno* 列中，列以相反的顺序进行编号，这好像是偶然的。也就是说，三级权值仅影响查询末尾的 ORDER BY，而不是 *dense_rank* 和 *row_number* 函数的 ORDER BY。

现在，如果您查看该查询的查询计划，在单词上有一个聚集索引，则您会发现一个 Sort 操作符，这是很让人吃惊的。如果在计划中返回上一步，则会发现一个 Compute Scalar 操作符，如果按 F4 键，则会看到操作符定义为[Expr1005]=Scalar Operator (tertiary_weights([tempdb].[dbo].[#prefwords].[word]))，如果查看 Sort 操作符，则会看到它是按照 *word* 和 Expr1005 排序的。也就是说，三级权值没有在索引中存储，而是在运行时计算得到的。

这就是函数 *TERTIARY_WEIGHTS* 起作用的地方。该函数接受 *char*、*varchar* 和 *text* 类型的参数，而且当输入值不是取自一个三级排序规则时将返回一个非空值。SQL Server 联机丛书指出您可以添加一个利用该函数计算得到的列，然后在字符列和计算得到的列上添加一个索引，如以前脚本中的 *tert_ix*。如果在以前的脚本中没有创建 *tert_ix* 并且从 *SELECT* 语句中也注释掉了 *rank* 和 *rowno* 列，则会看到一个没有 Sort 运算符的计划。这样函数 *TERTIARY_WEIGHTS* 就可以帮助您提高使用三级排序规则的性能了。

现在来看一下如果我们没有从 ORDER BY 子句中注释掉 wordno 会得到什么结果，具体的查询语句如下：

```
SELECT word, wordno
FROM #prefwords
ORDER BY word, wordno;
```

输出结果如下：

```
word      wordno
------    ------
abc       1
aBC       2
AbC       3
abC       4
Abc       5
ABC       6
aBc       7
ABc       8
```

也就是说，三级权值仅在整个 ORDER BY 子句没有其他差异时才会起作用。不用说，查询计划还是

会包含 Sort 运算符。

在 SQL Server 安装期间定义的排序规则。安装 SQL Server 时，需要选择一种服务器排序规则。这是一项很重要的选择，因为如果您做出了一种错误的选择，以后就不容易修改了。一般来说必须要重新安装 SQL Server。

SQL Server 安装提供了一种默认排序规则，这种默认排序规则总是 CI_AS——一种区分重音但是不区分大小写、假名类型和全半角的排序规则。安装程序选择排序规则指示符作为 *system locale* 的默认排序规则——应用在系统级别上的区域设置可能与您自己 Windows 用户的区域设置有所不同。默认值总是一种 Windows 排序规则，除了一种非常值得注意的情况：如果系统区域设置的是英语（美国），则默认为 SQL_Latin1_General_CP1_CI_AS。设置这种表面上看起来奇怪的默认值的原因是为了向后兼容。

当同一种语言存在不同版本时，默认值由系统区域设置是在以前 Windows 版本中就存在还是在 Windows 2008 中添加的来决定。例如，对于英语（英国）和德语（德国）来说，默认值为 Latin1_General_CI_AS，而对于英语（新加坡）和斯瓦希里语（肯尼亚）来说，默认值是 Latin1_General_100_CI_AS。同样，这种变化是为了实现向后兼容。要查看默认排序规则的整个列表，请参阅 *SQL Server 联机丛书*中的“安装时的排序规则设置”这一主题。

虽然安装程序提出了一种默认排序规则，但是这种默认排序规则对于您的服务器来说不一定是最好的。您应该做出一个有意识的、精心的决定。如果您安装一台服务器来运行一个第三方产品，则应该翻阅供应商的文档查看是否对应用程序有任何建议或要求。如果打算从早期版本的 SQL Server 中移植数据库，则应该为新服务器选择与现有服务器相同的排序规则。正如我原来提到的那样，如果打算将一台服务器作为 SQL Server 2005 的一台链接服务器来访问，则应该避免使用新的_100 排序规则。

另一件要注意的事情就是 Windows 管理员可能已经安装了一个 U.S. 英语版本的 Windows，那么即使本地语言不同，仍然要将系统区域设置为英语（美国）。如果这种情况发生在您的服务器上，并且您在安装 SQL Server 时没有注意，则最终会发生排序规则不适合您所在国家/地区的语言的情况。

一些语言有多种适合的选择。例如，对于德语来说，默认的选择是 Latin1_General_CI_AS，但是您也可以使用任意一种 German_Phonebook 排序规则（其中 ä、ö 和ü排序为 ae、oe、和 ue）。

运行安装向导。运行 SQL Server 2008 安装向导时，需要注意观察，因为排序规则的选项不在单独的一页上而是在“服务器配置”页的第二个选项卡上。您必须认真观察，因为进入“服务器配置”页面时不会显示排序规则选项卡。您会看到一个询问服务账号信息的页面。当您选择屏幕上的“排序规则”选项卡时，效果如图 5-4 所示。

图 5-5 所示为单击“自定义”按钮后显示的对话框。

可以使用一个选项按钮选择是使用一种 Windows 排序规则还是使用一种 SQL 排序规则。如果选择一种 Windows 排序规则，则会显示一个下拉列表用于选择排序规则指示符。列表下面有一些复选框用于选择区分大小写和其他特性。其中 *Binary* 提供一种 BIN 排序规则，而 *Binary-code point* 则提供一种 BIN2 排序规则。如果选择使用一种 SQL 排序规则，则只会有一个列表框列出所有 SQL 排序规则。

性能方面的考虑。排序规则的选择影响性能吗？是的，但很多情况下只是最低限度地影响，最重要的标准应该是选择能够最好地满足用户需求的排序规则。但是，有一些情况下排序规则可能有非常极端的效果。

一般来说，二进制排序规则的性能最好，但是在大部分应用程序中给用户带来的体验都不是很好。

只要您使用 *varchar* 数据，SQL 排序规则就能很好地作用。SQL 排序规则仅包含针对规则涉及的代码页中的 255 个字符的规则。一种 Windows 排序规则总是对系统内部的所有 Unicode 规则有效，即使是

单字节数据也是如此。因此，SQL 排序规则的内部例程比 Unicode 的简单得多。

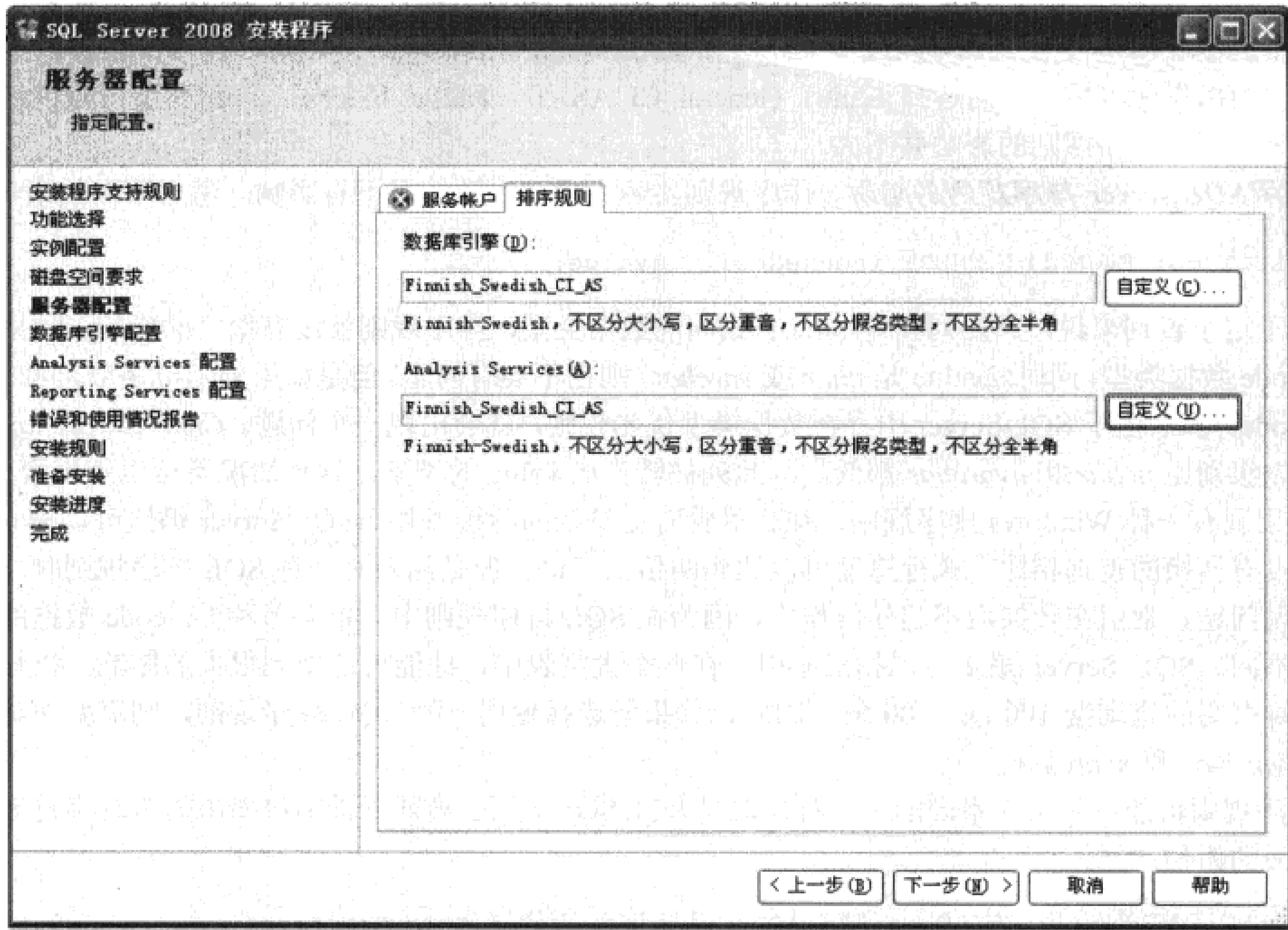


图 5-4 设置服务器配置

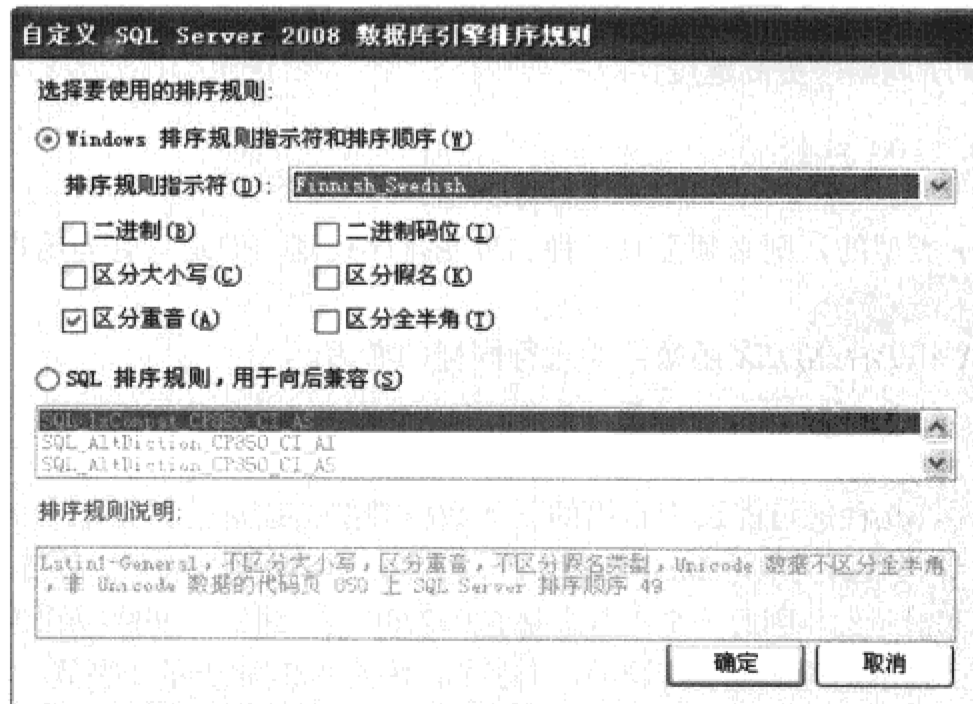


图 5-5 自定义排序规则属性

Windows 排序规则在排序规则族之间有一些差异，其中一些比另一些更快。一种特殊情况是不区分

大小写的 Latin1_General 和 Latin1_General_100 排序规则，操作 Unicode 数据时，这两种规则比其他任何排序规则族执行速度都快。与您想象的相反，区分大小写的排序规则的性能并不是很好，而是在很多操作中速度更慢。但是，您不必对此过多关注。如果用户希望看到按照丹麦字母表排序的数据，那么就没有必要因为操作速度快一点而选择 Latin1_General_CI_AS 了。同样，请记住：一种典型的查询包括很多其他部分，因此排序规则的影响并不大。

使用 SQL Server 排序规则的陷阱。排序规则实际上只在少数情况下有影响。考虑下面的语句：

```
SELECT col FROM tbl WHERE indexedcol = @value;
```

对于这个查询来说，只要列和@value 有相同的数据类型，排序规则就没有很大的影响。如果列有一个 Unicode 数据类型，同时@value 是 *char* 或 *varchar*，则也不会有问题。但是如果列是单字节，同时@value 是 Unicode，那么由于 SQL Server 中存在数据类型优先规则，就会出现一个问题。*Char* 和 *varchar* 数据类型的优先级别比 *nchar* 和 *nvarchar* 都低，因此列被转换成 value 的类型，这种情况下可以使用索引。

如果列有一种 Windows 排序规则，那么尽管有更复杂的筛选条件，SQL Server 仍然可以搜索索引，因此与没有转换的查询相比，执行速度可以达到两倍到三倍。但是当列有一种 SQL 排序规则时，该查询就会出现问题。索引在转换后不起任何作用，因为在 SQL 排序规则中，单字节和 Unicode 数据的排序规则完全不同。SQL Server 最多可以扫描索引。在一个大型表中，性能可能受到很大的影响，执行时间可能比正确书写的查询慢 100 或 1000 倍。因此，如果您选择使用一种 SQL 排序规则，则需要注意不要随便混用 *varchar* 和 *nvarchar*。

排序规则可能产生很大不同的另一种情况是 SQL Server 几乎必须查看字符串中的所有字符时。例如，请看下面的语句：

```
SELECT COUNT(*) FROM tbl WHERE longcol LIKE '%abc%';
```

该语句使用一种二进制排序规则比非二进制 Windows 排序规则的查询速度快 10 倍或更多。对于 *varchar* 数据，SQL 排序规则的执行速度比 Windows 排序规则快 7～8 倍。如果有一个 *varchar* 列，则可以通过强制使用如下排序规则来提高速度：

```
SELECT COUNT(*) FROM tbl
WHERE longcol COLLATE SQL_Latin1_General_CP_CI_AS LIKE '%abc%';
```

如果列是 *nvarchar* 类型的，则必须强制一种二进制排序规则，但是只有当用户能够接受一种区分大小写的查询时才可行。

对于 *CHARINDEX* 和 *PATINDEX* 函数需要进行同样的考虑。

6. 特殊数据类型

最后，我们通过介绍几种您可能认为有用的其他数据类型结束对数据类型这一部分的介绍。

二进制数据类型。这些数据类型是 *binary* 和 *varbinary* 的，用于存储比特串，其值是通过十六进制表示法（用前缀 *0x* 表示）输入和显示的。因此一个十六进制值 0x270F 与十进制值 9999 及比特串 0010011100001111 相对应。在十六进制中，显示的每两个字符表示一个字节，因此 0x270f 表示 2 字节。您需要确定数据是固定长度的还是可变长度的，同时您可以根据前面讨论的选择 *char* 和 *varchar* 的方法来确定数据类型。*binary* 或 *varbinary* 类型数据的最大长度都是 8000 字节。

***bit* 数据类型**。bit 数据类型可以存储一个 0 或一个 1，而且仅占用一个比特的存储空间。但是，如果

表中只有一个 bit 类型的列，则该列会占用一整个字节。一个字节中最多可以存储 8 个 bit 类型的列。

LOB 数据类型。SQL Server 2008 允许您定义具有 MAX 属性的列：*varchar(MAX)*、*nvarchar(MAX)* 和 *varbinary(MAX)*。如果实际插入到这些列中的字节数超过了最大值 8000，则这些列会使用一种为 LOB 数据准备的特殊存储格式进行存储。这种特殊存储格式与 *text*、*ntext* 和 *image* 数据类型使用的存储格式相同，但是由于 *text*、*ntext* 和 *image* 等数据类型在将来的 SQL Server 版本中不再继续使用，因此建议您在进行新开发时使用带有 MAX 限定符的可变长度数据类型。*varchar(MAX)*（或 *text*）数据类型最多可以存储 2^31–1 个非 Unicode 字符，*nvarchar(MAX)*（或 *text*）最多可以存储 2^30–1（一半）个 Unicode 字符，而 *varbinary(MAX)*（或 *image*）最多可以存储 2^31–1 字节的二进制数据。此外，*varbinary(MAX)*数据可以存储为文件流数据。我们将在第 7 章中更详细地介绍文件流数据及 LOB 数据存储结构。

***cursor* 数据类型**。*cursor* 数据类型可以存储对游标的引用。虽然您不能将一个表的某一列声明为 *cursor* 类型，但是这种数据类型可以用于输出参数和本地变量。这里为了介绍全面，我们简单提一下 cursor 数据类型，但是不对这一数据类型做过多的介绍。

***rowversion* 数据类型**。*rowversion* 数据类型是原来被称为 *timestamp* 的数据类型的一个同义词。当使用 *timestamp* 数据类型名称时，很多人可能会认为数据与日期或时间有关，但事实并非如此。类型为 *rowversion* 的列保留一个每次行被修改时 SQL Server 自动更新的内部顺序号。任何 *rowversion* 列的值在整个数据库中实际上都是唯一的，而且一个表中只能有一个类型为 rowversion 的列。修改数据库中 *rowversion* 列的任何操作都会生成下一个序列值。存储在 *rowversion* 列中的实际值本身并不重要。该列通过检查 *rowversion* 值是否被修改过来检测自最后一次访问以来某一行是否被修改过。

***sql_variant* 数据类型**。*sql_variant* 数据类型允许列保存除 *text*、*ntext*、*image*、*XML*、用户定义数据类型、带有 MAX 限定符的可变长度数据类型及 *rowversion*（*timestamp*）之外的任何数据类型的值。我们将在本章后面介绍 *sql_variant* 数据的内部存储问题。

空间数据类型。SQL Server 2008 提供了两种数据类型用于存储空间数据。*geometry* 数据类型支持平面或欧几里得数据。*geometry* 数据类型与 SQL 标准版本 1.1.0 的开放地理信息联盟（Open Geospatial Consortium，OGC）简单特性相一致。*geography* 数据类型存储椭圆数据，如全球定位卫星（Global Positioning Satellite，GPS）经度和纬度坐标。这些数据类型具有自己访问和操作数据的方法，也有自己特殊的扩展索引结构，这一点与常见的 SQL Server 索引不同。关于空间数据访问方法和存储的具体知识不是本书讨论的范畴。

***table* 数据类型**。*table* 数据类型用于存储函数的结果，也可以当做本地变量数据类型来使用。表中的列不能是 *table* 类型。

***xml* 数据类型**。*xml* 数据类型用于存储 XML 文档和 SQL Server 数据库中的片段。您可以在创建表时使用 *xml* 数据类型作为某一列的数据类型，或者作为变量、参数和函数返回值的数据类型。XML 数据有自己的检索和操作方法。本书对使用 *xml* 数据不做详细介绍。

***uniqueidentifier* 数据类型**。*uniqueidentifier* 数据类型有时指一个全球唯一标识符（GUID）或通用唯一标识符（UUID）。一个 GUID 或 UUID 是一个 128 位（16 字节）值，出于实用目的的考虑，其生成方式可以保证世界范围内每一台联网的计算机的值唯一。现在这种方式已经成为标识分布式系统中的数据、对象、软件应用程序及 applet 的一种重要方式。由于 *uniqueidentifier* 数据类型的生成及操作方式有一些地方非常有趣，因此这里我们再详细介绍一下。

T-SQL 语言支持 *NEWID* 和 *NEWSEQUENTIALID* 两个系统函数，您可以利用这两个函数生成 *uniqueidentifier* 值。一个 *uniqueidentifier* 数据类型的列或变量的值可以按照以下两种方式中的一种进行初

始化。

- 使用系统函数 *NEWID* 或 *NEWSEQUENTIALID* 作为默认值。
- 使用如下格式（用连字符分隔的 32 个十六进制数）的字符串常量：*xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx*（每个 x 是一个 0 到 9 或 a 到 f 之间的十六进制数）。

这种数据类型使用起来可能非常繁琐，对 *uniqueidentifier* 值允许进行的唯一操作是比较运算（=，<>，<，>，<=，>=）和检查空值。但是，使用这种数据类型从系统角度来说有一些好处。

使用 *uniqueidentifier* 数据类型的一个原因就是由 *NEWID* 或 *NEWSEQUENTIALID* 生成的值对网络上的所有计算机来说都是全球唯一的，因为 *uniqueidentifier* 值的最后 6 个字节构成该机器的节点号。当 SQL Server 机器没有一个以太网/令牌环（IEEE802.x）地址时，将没有节点编号，同时生成的 GUID 肯定是在该机器上生成的所有 GUID 中唯一的。但是，存在另一台没有以太网/令牌环地址的计算机会生成相同 GUID 的可能。在具有网络地址的计算机上生成的 GUID 肯定是全球唯一的。

SQL Server 需要生成一个 GUID 的主要原因是为了合并复制，同一个表的标识符值可以在很多不同的 SQL Server 机器中的任意一台上生成。需要一种方式来确定两行实际上是否是同一行，而且指向同一个实体的两行不可能具有相同的标识符。使用 GUID 值可以提供这种功能。具有相同 GUID 值的两行必须指出它们实际上是同一行。

NEWSEQUENTIALID 和 *NEWID* 函数之间的区别在于 *NEWSEQUENTIALID* 创建比原来在一台指定计算机上通过该函数生成的所有 GUID 都更大的 GUID，而且可以用于产生一系列 GUID 值。这样可以通过合并复制大大提高系统的可伸缩性。如果 *uniqueidentifer* 值用做被复制表的聚集键，则会在随机的磁盘页上插入新行（您将在第 6 章介绍聚集索引时看到详细介绍）。如果涉及的机器正在执行大量的 I/O 操作，则由 *NEWID* 函数生成的非顺序 GUID 会引起很多随机 B 树查找及低效率的插入操作。新函数 *NEWSEQUENTIALID*（该函数是 Windows 函数 *UuidCreateSequential* 的一个封装函数）会进行字节争夺并为生成的 UUID 值创建一个顺序。

uniqueidentifier 值的列表不可能被用完。这不是其他数据类型通常被用做唯一标识符的情况。事实上，SQL Server 使用这种数据类型在系统内部进行行级合并复制。一个 *uniqueidentifier* 列可以有一个被称为 *ROWGUIDCOL* 的特殊属性，每个表只允许有一个 *uniqueidentifier* 列有该属性。*ROWGUIDCOL* 属性可以被指定作为 *CREATE TABLE* 和 *ALTER TABLE ADD column* 列定义语句的一部分，也可以使用 ALTER TABLE ALTER COLUMN 语句在现有列中被添加或删除。

可以在查询中使用关键字 *ROWGUIDCOL* 引用具有 *ROWGUIDCOL* 属性的 *uniqueidentifier* 列。这与使用 *IDENTITYCOL* 关键字引用标识列类似。*ROWGUIDCOL* 属性没有暗示自动生成任何值，如果需要自动生成值，则需要将 *NEWID* 函数定义为列的默认值。每个表中可以有多个 *uniqueidentifier* 列，但是只有一列可以有 *ROWGUIDCOL* 属性。您可以根据自己想出的任何理由使用 *uniqueidentifier* 数据类型，但是如果您正使用这种数据类型标识现有行，那么应用程序必须有一种通用方式，能在无需知道列名称的情况下访问它。这就是 *ROWGUIDCOL* 属性的功能所在。

5.1.6 关于 NULL

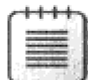
是否允许 NULL 的问题已经成为业内人士讨论的热点，这里的讨论可能会激怒一些人。但是，我并不是要参与一场哲学上的争论。实事求是地说，处理 NULL 会给存储引擎带来额外的复杂性，因为 SQL Server 在每一行中都保留了一个特殊的位图，用于指示哪些可以为空的列实际上是 NULL。如果不允许 NULL，则 SQL Server 必须为访问的每一行解码该位图。允许 NULL 也会增加应用程序代码的复杂性，

这经常会导致 bug。您必须为说明 NULL 情况添加特殊的逻辑。

作为数据库设计者，您可以在执行合并及按值搜索时理解 NULL 和聚合函数中三值逻辑的微小差异。此外，您还必须要考虑开发人员是否真的知道如何使用 NULL。如果可能的话，我们建议您使用 NOT NULL 列并为缺省或未知数据定义*默认*值（如果默认值大小与通常输入的值明显不同，则创建这样的字符列）。

在任何情况下，创建表时最好明确声明 NOT NULL 或 NULL。如果没有进行这样的声明，则 SQL Server 会认为是 NOT NULL（换言之，不允许存在声明）。这可能让那些认为 SQL Server 默认允许声明的人感到惊讶。造成这种误解的原因在于操作 SQL Server 的大部分工具和界面都启用了一个会话设置将默认值设置为允许 NULL。但是，您可以通过使用一个会话或一个数据库选项将默认值设置为允许 NULL，正如刚刚提到的那样，这就是大部分工具和用户界面所采取的做法。如果您将 DDL 编写成脚本，然后在另一台具有不同默认设置的服务器上运行脚本，并且在列定义中没有明确指定 NULL 或 NOT NULL 时，您会得到不同的结果。

有几个数据库选项和会话设置可以根据 NULL 值控制 SQL Server 的行为。正如我们在第 3 章提到的那样，您可以利用 *ALTER DATABASE* 命令设置服务器选项。同时也可以利用 *SET* 命令在某一时刻为某一连接启动会话设置。



注意:

数据库选项 *ANSI null default* 与 ANSI_NULL_DFLT_ON 和 ANSI_NULL_DFLT_OFF 两种会话设置相对应，当 *ANSI null default* 数据库选项设置为 false 时（SQL Server 的默认设置），如果使用 *ALTER TABLE* 和 *CREATE TALBE* 命令创建的新列的空值状态没有明确指定，则默认为 NOT NULL。SET ANSI_NULL_DFLT_OFF 和 SET ANSI_NULL_DFLT_ON 是两种互斥选项，用于指出数据库选项是否应该被替换。设置为 on 时，每个选项强制对立选项设置为 off。设置为 off 时，两种选项都不会将对立选项设置为 on—— 只会终止当前设置。

使用 *GETANSINULL* 函数可以确定当前会话的默认为空性。当新列允许空值或者该列或数据类型为空性且在表被创建或修改没有明确定义时，该函数返回值为 1。强烈建议您在创建列时明确声明 NULL 或 NOT NULL。这样可以消除二义性，同时可以保证您在控制着表的构建方式，不管默认为空性设置如何。

数据库选项 *concat null yields null* 与会话设置 *SET CONCAT_NULL_YIELDS_NULL* 相对应。当 *CONCAT_NULL_YIELDS_NULL* 为 on 时，将一个 NULL 值与一个字符串连接将产生一个 NULL 结果。例如，SELECT 'abc' + NULL 会产生 NULL。当 *SET CONCAT_NULL_YIELDS_NULL* 设置为 off 时，将一个 NULL 值与一个字符串连接的结果是字符串本身。换言之，NULL 值被作为一个空串看待。例如，SELECT 'abc' + NULL 的结果是 abc。如果会话级设置没有被指定，则会应用数据库选项 *concat null yields null* 的值。

数据库选项 *ANSI nulls* 与会话设置 *SET ANSI_NULLS* 相对应。当该选项设置为 ON 时，所有与 NULL 值的比较结果都是 UNKNOWN。当该选项设置为 OFF 时，如果一个值与一个 NULL 相比较时两个值都是 NULL，则比较结果为 TRUE。此外，当该选项设置为 ON 时，代码必须使用 IS NULL 条件确定某一列是否有一个 NULL 值。当该选项设置为 OFF 时，SQL Server allows=NULL 作为 IS NULL 的同义字，<> NULL 作为 IS NOT NULL 的同义字。

第4种会话设置是*ANSI_DEFAULTS*。将其设置为ON是启用ANSI_NULLS和ANSI_NULL_DFLT_ON及其他与NULL操作无关的会话设置的快捷方式。SQL Server ODBC驱动程序及SQL Server OLE DB服务程序自动将*ANSI_DEFAULTS*设置为ON。当您定义数据源名称（DSN）时，可以修改ANSI_NULLS设置。您应该知道用于连接SQL Server的工具会将某些选项设置为ON或OFF。

下面的查询语句显示了当前会话中所有SET选项的值，而且如果您具有VIEW SERVER STATE权限，则可以修改或删除WHERE子句从而返回关于其他会话的信息，语句如下：

```
SELECT * FROM sys.dm_exec_sessions
WHERE session_id = @@spid;
```

正如您所见，您可以按照多种方式配置和控制NULL值的处理和行为，同时您可以认为不可能跟踪所有变化。如果试图控制在每个会话中单独处理NULL的各个方面，那么可能会引起不可估量的混淆甚至是灾难。但是，如果您遵循如下一些基本建议，则大部分问题都会解决。

- 表中不允许NULL值。
- 在表定义中包括一种特殊的NOT NULL约束。
- 不要依靠数据库属性来控制NULL值的行为。

如果在某些情况下一定要使用 NULL，可以遵循同样的规则使问题降到最少，最简单的规则就是ANSI中已经制定的规则。

此外，某些数据库的设计允许在很多列和很多行中存在NULL值。SQL Server 2008中引入了稀疏列的概念。稀疏列通过利用更多的系统开销来检索NOT NULL值，从而降低NULL值的空间需求。因此当大部分数据都为NULL时，就可以体现出稀疏列的强大优势。我们将在第7章讨论稀疏列存储。

当允许列为NULL时，需要注意其他存储注意事项。对于固定长度的列来说（没有定义为稀疏列），即使在存储NULL值时也一直使用定义的整个长度。例如，一个定义为*char(200)*的列总是占用200字节，不管存储的值是否为空。可变长度列则不同，它们不会为实际存储的NULL占用任何空间。这并不是说根本就不需要空间，这一点将在本章后面介绍内部存储机制时介绍。

5.2 用户定义数据类型

用户定义数据类型（UDT）为您提供一种简便方式，用于保证具有相同可能值域的列一直使用基础本机数据类型。例如，您的数据库可能在很多张表中存储了各种电话号码。虽然没有一种单一、明确的方式存储电话号码，但是在这个数据库中一致性很重要。您可以创建一种*phone_number*用户定义数据类型并使用这种数据类型作为所有表中保存电话号码的列的类型，从而保证各个表中的电话号码列具有相同的数据类型。下面是创建这种用户定义数据类型的语句：

```
CREATE TYPE phone_number FROM varchar(20) NOT NULL;
```

下面是创建表时使用这种新定义的用户定义数据类型的方式：

```
CREATE TABLE customer
(
cust_id             smallint     NOT NULL,
cust_name           varchar(50)  NOT NULL,
cust_addr1          varchar(50)  NOT NULL,
cust_addr2          varchar(50)  NOT NULL,
```

```
cust_city              varchar(50)  NOT NULL,
cust_state             char(2)      NOT NULL,
cust_postal_code       varchar(10)  NOT NULL,
cust_phone             phone_number NOT NULL,
cust_fax               varchar(20)  NOT NULL,
cust_email             varchar(30)  NOT NULL,
cust_web_url           varchar(100) NOT NULL
);
```

创建表时，系统内部将 *cust_phone* 数据类型当做 *varchar* (*20*)。注意 *cust_phone* 和 *cust_fax* 都是 *varchar(20)*，虽然 *cust_phone* 是通过定义为一种 UDT 的方式进行声明的。

表中列的信息是通过目录视图 *sys.columns* 获得的，关于 *sys.columns* 的更多内容，我们将在本章后面的“内部存储”一节进行介绍。现在我们只是通过一个基本查询来查看 *sys.columns* 中的两个列，其中一列包含很多表示基本系统的数据类型，另一列包含很多代表创建表时使用的数据类型。下面的查询选择 *sys.columns* 中的所有列并显示 *column_id*、列名、数据类型值及最大长度并显示结果：

```
SELECT column_id, name, system_type_id, user_type_id,
        type_name(user_type_id) as user_type_name, max_length
FROM sys.columns
WHERE object_id=object_id('customer');
```

```
column_id   type_name        system_type_id   user_type_id user_type_name max_length
----------- ---------------- ---------------- ------------ ------------ ----------
1           cust_id          52               52           smallint     2
2           cust_name        167              167          varchar      50
3           cust_addr1       167              167          varchar      50
4           cust_addr2       167              167          varchar      50
5           cust_city        167              167          varchar      50
6           cust_state       175              175          char         2
7           cust_postal_code 167              167          varchar      10
8           cust_phone       167              257          phone_number 20
9           cust_fax         167              167          varchar      20
10          cust_email       167              167          varchar      30
11          cust_web_url     167              167          varchar      100
```

您可以看到，*cust_phone* 和 *cust_fax* 列都具有相同的 *system_type_id* 值，虽然 *cust_phone* 列显示 *user_type_id* 是一种 UDT（*user_type_id*=257)。当表被创建时该类型将被检索，同时只要某张表正在使用这种 UDT，它就不能被修改或删除。某种 UDT 一旦被声明，它就是静态和不可变的，因此在使用一种 UDT 代替本地数据类型时不会出现固有的性能损失。

使用 UDT 可以使数据库更一致、清楚。SQL Server 隐式在不同类型的兼容列之间进行转换（不是本机类型就是不同类型的 UDT)。

现在，UDT 不支持子类型化或继承的概念，也不允许一种 DEFAULT 值或 CHECK 约束被声明为 UDT 本身的一部分。这些强大的面向对象概念很可能成功地进入 SQL Server 未来的版本。虽然有这些限制，但 UDT 功能是一种动态而且通常是 SQL Server 不经常使用的功能。

5.3 IDENTITY 属性

通常为没有一种自然或有效主键的表提供简单的计数器类型的值。*cust_id* 等列通常是简单的计数器

字段。*IDENTITY* 属性使得生成唯一的数字值非常简单。*IDENTITY* 不是一种数据类型，而是一种可以定义在 *tinyint*、*smallint*、*int*、*bigint* 或 *numeric/decimal*（只有一定数量的 0 起作用）等各种数值数据类型上的列属性。每张表都只能有一列具有 *IDENTITY* 的属性。表的创建者可以指定起始数字（种子）及该值增加或减少的数量。如果没有明确指定，则种子值从 1 开始，增量为 1，如下面的示例所示：

```
CREATE TABLE customer
(
cust_id       smallint      IDENTITY NOT NULL,
cust_name     varchar(50)   NOT NULL
);
```

为了找出某个表中定义的种子和增量值，可以使用 *IDENT_SEED(表名)*和 *IDENT_INCR(表名)*函数。我们来看下面的语句：

```
SELECT IDENT_SEED('customer'), IDENT_INCR('customer')
```

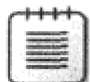
由于值没有明确声明并且使用了默认值，因此这条语句在 customer 表上产生的结果如下。

```
1 1
```

下面的示例明确说明种子的起始值为 100，增量为 20：

```
CREATE TABLE customer
(
cust_id       smallint      IDENTITY(100, 20) NOT NULL,
cust_name     varchar(50)   NOT NULL
);
```

由 *IDENTITY* 属性自动生成的值通常是唯一的，但是这并不是由 *IDENTITY* 属性本身提供的，而且 *IDENTITY* 值也不一定是连续的（我们将在这一部分的后面展开介绍非唯一和非连续 *IDENTITY* 值的问题）。就效率而言，一个值在传给客户端执行一项 INSERT 操作时就认为被使用了。如果该客户端最终没有提交 *INSERT* 命令，该值也不会再出现，因此在连续的数字中就出现了中断。如果下一个数字在前一个数字真正提交或回滚之前不能被分配，则会存在一种不可接受的序列化（甚至在这时，一旦某一行被删除，其值将不再连续。间隔是不可避免的）。



注意：

如果您需要没有间隔的精确序列值，则不适合使用 *IDENTITY* 功能。您应该使用一个 *next_number-type* 表，在这个表中您可以执行替代表中包含的更大事务记录数的操作（从而引起值的序列化）。

为了在一个标识列中临时禁用值的自动生成，可以使用 SET IDENTITY_INSERT tablename ON 选项。除了在标识序列中填充间隔，该选项对于大批加载数据（其中以前的值已经存在）的任务很有用。例如，可能您正在从原有系统加载一个具有客户数据的新数据库。您可能希望保留原有的客户编号但是使新编号自动使用 *IDENTITY* 分配。SET 选项就是为这样的情况创建的。

由于 SET 选项允许您确定一个 *IDENTITY* 列的值，因此 *IDENTITY* 属性本身不强制表中值的唯一性。虽然在 IDENTITY_INSERT 从来没有被启动时，*IDENTITY* 会生成一个唯一的数字，但是一旦您使用过

SET 选项，就不能保证唯一性了。为了强制唯一性（使用 *IDENTITY* 时几乎总是要这样操作），还应该在列上声明 UNIQUE 或 PRIMARY KEY 限制。如果您为某个标识列插入一个值（使用 SET IDENTITY_INSERT），那么在恢复自动生成时，下一个值是表中现有最高值的下一个增量值，不管这个最高值是以前生成的还是显式插入的。

提示：

如果使用 bcp 实用工具大量加载数据，当数据已经分配希望为具有 IDENTITY 属性的某一列保留的值时，请注意-*E*（大写）属性。您也可以使用带有 KEEPIDENTITY 选项的 *T-SQL BULK INSERT* 命令。要了解更多信息，请参阅 SQL Server 文档关于 *bcp* 和 *BULK INSERT* 的内容。

关键字 *IDENTITYCOL* 自动参照表中具有 *IDENTITY* 属性的特殊列，不论该列的名称如何。如果该列是 *cust_id*，则可以在不知道或不使用列名的情况下通过 IDENTITYCOL 引用该列，或者明确地通过 *cust_id* 引用。例如，下面两条语句的工作方式相同并且返回相同的数据：

```
SELECT IDENTITYCOL FROM customer;
SELECT cust_id FROM customer;
```

两种情况下返回给调用者的列名称都是 *cust_id*，而不是 *IDENTITYCOL*。

当插入列时，必须从列名和 VALUES 部分忽略一个标识列（唯一的特殊情况是当 IDENTITY_INSERT 选项设置为 on 时）。如果您真的提供了一个列的列表，则必须忽略值将被自动提供的列。下面是针对前面显示的 *customer* 表的两条有效的 *INSERT* 语句：

```
INSERT customer VALUES ('ACME Widgets');
INSERT customer (cust_name) VALUES ('AAA Gadgets');
```

选择这两行将得到如下输出结果：

```
cust_id  cust_name
-------  ---------
1        ACME Widgets
2        AAA Gadgets
```

在应用程序中，有时为了后续使用需要立即知道 *IDENTITY* 生成的值。例如，一项事务可能首先添加一个新顾客并接下来为该顾客添加一个订单。为了添加订单，您可能需要使用 *cust_id* 列。不是从 *customer* 表中选择值，而是简单地选择特殊的系统函数@@*IDENTITY*，该函数包含该连接使用的最后标识值。但是不必提供表中插入的最后值，因为另一个用户可能随后已经插入了数据。如果在相同或不同表上一次执行多条 INSERT 语句，那么变量的值将是最后一条语句赋予的值。此外，如果在您插入一个新行后触发了一个 INSERT 触发器，并且该触发器向某个具有标识列的表中插入行，则@@IDENTITY 不具有由原始 INSERT 语句插入的值。对于您来说，好像您正在插入然后立刻检查值，语句如下：

```
INSERT customer (cust_name) VALUES ('AAA Gadgets');
SELECT @@IDENTITY;
```

然而，如果一个触发器触发了 *INSERT*，则@@*IDENTITY* 的值可能已经被修改。

使用标识列时您可能会发现另外两个有用的函数：*SCOPE_IDENTITY* 和 *IDENT_CURRENT*。

SCOPE_IDENTITY 返回插入到同一作用域的表中的最后一个标识值，该作用域可以是一个存储过程、触发器或批处理。因此，如果在先前的代码段中用 *SCOPE_IDENTITY* 函数替代@@*IDENTITY*，就会看到标识值被插入到 *customer* 表中。如果一个 *INSERT* 触发器还插入了一个包含标识列的行，则属于另一种不同的作用域，如下面的语句所示：

```
INSERT customer (cust_name) VALUES ('AAA Gadgets');
SELECT SCOPE_IDENTITY();
```

在其他情况下，您可能希望知道插入到某个来自任何用户或应用程序的特定表中的最后一个标识值。可以利用 *IDENT_CURRENT* 函数获得该值，*IDENT_CURRENT* 函数取一个表名作为参数：

```
SELECT IDENT_CURRENT('customer');
```

这并不能保证您可以预测将被插入的下一个标识值，因为另一个进程可以在您检查 *IDENT_CURRENT* 和执行 *INSERT* 语句之间插入一行。

不能将 *IDENTITY* 属性定义为一种 UDT 类型，但是可以在使用 UDT 的列上声明 *IDENTITY* 属性。具有 *IDENTITY* 属性的列必须被声明为 NOT NULL（显式地或隐式地）；否则，CREATE TABLE 语句会产生 8147 错误同时 *CREATE* 不会成功。同样，您不能在同一列上声明 *IDENTITY* 属性和一个 *DEFAULT*。要根据表中当前的最大值检查当前标识值是否有效，同时为了在发现一个非有效值时（应该永远不发生这种情况）进行重新设置，应使用 *DBCC CHECKIDENT(表名)*语句。

标识值是完全可恢复的。如果在具有标识列的表上发生插入活动时出现系统断电，那么在 SQLServer 重启时正确值会被恢复。这是 SQL Server 通过在检查点处理期间刷新所有表的当前标识值来完成的。对于最后检查点之外的活动，随后的值在标准数据库恢复过程中从事务日志中重新构建。向有 *IDENTITY* 属性的表中进行的任何插入都已经修改了该值，同时从事务日中每个表的最后一条 INSERT 语句中检索当前值。最终结果是当数据库被恢复时，正确的当前标识值也会被恢复。

在很少的情况下，标识值可能会不同步。如果出现这种情况，可以利用 *DBCC CHECKIDENT* 命令将标识值重新设置为正确的编号。此外，该命令的 RESEED 选项允许您为标识序列设置一个新的起始值。参阅在线文档可以查看详细信息。

5.4 内部存储

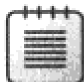
本节介绍 SQL Server 实际如何存储表中的数据。此外，还将分析跟踪数据存储信息的基本系统元数据。虽然您可以在不了解数据存储内部机制的情况下有效地使用 SQL Server，但是知道 SQL Server 如何存储数据可以帮助您开发有效的应用程序。

创建一张表时，会向很多张系统表中插入一行或多行数据用于管理该表，同时 SQL Server 还提供了建立在系统表上的目录视图，从而使您可以分析表中的内容。至少可以在 *sys.tables*、*sys.indexes* 和 *sys.columns* 目录视图下查看新建表的元数据。当您定义具有一种或多种约束的新表时，还可以查看 *sys.check_constraints*、*sys.default_constraints*、*sys.key_constraints* 或 *sys.foreign_keys* 视图中的信息。对于创建的每一张表来说，包含名称、对象 ID 及新表架构的 ID 的某一行都可以通过 *sys.tables* 视图得到。记住，*sys.tables* 视图继承 *sys.objects*（显示与所有类型对象相关的信息）中的所有列并且包括仅属于表的额外列。*sys.columns* 视图显示新建表中每一列中的一行，同时每一行都包含列名、数据类型及长度等信息。

每一列都会收到一个列 ID，该值初始时与您创建表时指定的列顺序相对应——也就是说，*CREATE TABLE* 语句中列出的第一列具有的列 ID 是 1，第二列的列 ID 是 2，依此类推。图 5-6 显示了创建一张表时 *sys.tables* 和 *sys.columns* 视图返回的行（并不是每个视图的所有列都被显示）。

```
CREATE TABLE dbo.employee (
                  emp_lname   varchar(15)   NOT NULL,
                  emp_fname   varchar(10)   NOT NULL,
                  address     varchar(30)   NOT NULL,
                  phone       char(12)      NOT NULL,
                  job_level   smallint      NOT NULL
)

sys.tables        object_id   name                 schema_id   type_desc
                  ----------- -------------------- ----------- ------------------
                  917578307   employee             1           USER_TABLE

sys.columns       object_id   column_id   name           system_type_id max_length
                  ----------- ----------- -------------- -------------- ----------
                  917578307   1           emp_lname        167             15
                  917578307   2           emp_fname        167             10
                  917578307   3           address          167             30
                  917578307   4           phone            175             12
                  917578307   5           job_level        52              2
```

图 5-6 创建表后存储的基本目录信息



注意：

如果表被修改为删除列，则列 ID 序列中可能有间隔。不过信息架构视图（*INFORMATION_SCHEMA.COLUMNS*）为您提供一个值 *ORDINAL_POSITION*，因为这是 ANSI SQL 标准所需要的。原始位置是您查询表中的所有列时列出的顺序。因此 *column_id* 不一定是该列的顺序位置。

5.4.1 sys.indexes 目录视图

除 *sys.columns* 和 *sys.tables* 之外，*sys.indexes* 视图至少为每个表返回一行。在 SQL Server 2005 以前的版本中，*sysindexes* 表包含表和索引的所有物理存储信息，这是唯一实际占用存储空间的对象。*sysindexes* 表用列跟踪所有表和索引使用的空间、每个索引根页的物理位置及每张表和索引的第一页（在第 6 章中，我们将介绍根页和首页的含义）。在 SQL Server 2008 中，兼容性视图 *sys.sysindexes* 包含很多相同的信息，但是由于 SQL Server 2005 中引入的存储结构变化，因此它是不完整的。*sys.indexes* 目录视图仅包含有关索引的基本属性信息，如索引是聚集还是非聚集的、唯一还是非唯一的及将在第 6 章介绍的其他属性。为了获得以前版本在 *sysindexes* 表中提供的 SQL Server 2005 或 SQL Server 2008 的所有存储信息，我们必须查看除 *sys.indexes* 之外的另外两个目录视图：*sys.partitions* 和 *sys.allocation_units*（或者选择未正式记录的 *sys.system_internals_allocation_units*）。我们将简要地介绍这些视图的基本内容，但是我们首先关注的是 *sys.indexes*。

您应该知道如果一张表有一个聚集索引，那么表的数据实际上被认为是索引的一部分，因此数据行实际上是索引行。对于具有一个聚集索引的表来说，SQL Server 在 *sys.indexes* 中有一个 *index_id* 值为 1

的行，同时 *sys.indexes* 中的名称列包含该索引的名称。与该索引相关的表的名称可以由 sys.indexes 中的 object_id 列决定。如果一张表没有聚集索引，则数据本身没有结构，我们称这样的表为*堆*。*sys.indexes* 表中的一个堆的 *index_id* 值为 0，同时名称列包含 NULL。每一个附加索引在 *sys.indexes* 中都有一行 *index_id* 值在 2～250 之间或者在 256～1005 之间（251～255 的值被保留）。由于在一张表上最多可以有 999 个非聚集索引，其中一行用于堆或聚集索引，因此每张表在 *sys.indexes* 视图中都有 1～1000 行用于相关索引。一张表可以在 *sys.indexes* 中有用于 XML 索引的附加行。用于 XML 索引的元数据在 *sys.xml_indexes* 目录视图中可用，它是从 *sys.indexes* 视图中继承列的。SQL Server 2008 中的两个主要特性使得使用多个目录视图跟踪存储信息最有效。首先，SQL Server 具有在多个分区上存储一张表或索引的能力，因此每个分区使用的空间及分区的位置必须被独立跟踪。其次，表和索引数据可以以 3 种不同的格式来存储，通常是标准行数据、行溢出数据和 LOB 数据。行溢出数据和 LOB 数据都可以是某个索引的一部分，因此每个索引都必须单独跟踪特殊格式的数据。这样，每张表就可以有多个索引，同时每个表和索引可以存储在多个分区上，而且每个分区需要以 3 种格式跟踪数据。我们将在第 6 章讨论索引，在第 7 章讨论行溢出数据和 LOB 数据的存储及分区表和索引。

5.4.2 数据存储元数据

每个堆和索引在 *sys.indexes* 中都有一行，同时 SQL Server 2008 数据库中的每张表和索引都可以存储在多个分区上。*sys.partitions* 视图为每个堆或索引的每个分区包含一行数据。每个堆或索引都至少有一个分区，即使没有专门对结构进行分区，每张表或索引最多也可以有 1000 个分区。因此在 *sys.indexes* 和 *sys.partitions* 之间存在一种一对多的关系。*sys.partitions* 视图包含一个称为 *partition_id* 的列，还有 *object_id* 和 *index_id* 列，因此您可以在 *object_id* 和 *index_id* 列上将 *sys.indexes* 与 *sys.partitions* 合并，从而检索某个特殊表或索引上的所有分区 ID 值。SQL Server 2008 中用于描述某个分区上某张表或索引子集的术语是 hobt，代表 Heap Or B-Tree（堆或 B 树），发音（您可以猜到）为“hobbit”（B 树是索引使用的存储结构）。*sys.partitions* 视图包括一个称为 hobt_id 的列，在 SQL Server 2008 中，*partition_id* 和 *hobt_id* 之间总有一种一对一的关系。事实上，您会发现 *sys.partitions* 表中的这两列始终具有相同的值。

每个分区（不论是用于存储堆的还是用于存储索引的）都可以有 3 种类型的行，每一行存储在自己的页面上。这些类型被称为行内数据页（用于我们的“标准”数据或索引信息）、行溢出页和 LOB 数据页。某个特殊分区的一种特殊类型的一组页面称为一个*分配单位*，因此我们需要介绍的最后一种目录视图是 *sys.allocation_units*。*sys.allocation_units* 视图的每个分区包含一行、两行或三行，因为每个分区上的每个堆或索引可以有 3 个分配单位。标准行内数据总有一个分配单位，但是也可能有一个分配单位用于 LOB 数据，一个分配单位用于行溢出数据。图 5-7 显示了 *sys.indexes*、*sys.partitions* 和 *sys.allocation_units* 之间的关系。

查询目录视图

现在我们来看一个具体的示例，以查看这 3 种目录视图中的信息。首先创建原来在图 5-6 中的表。可以在任何数据库中创建，但是我们建议要么使用 *tempdb*，这样在下次重启 SQL Server 实例时自动删除表，要么创建一个新数据库进行测试。我们的很多示例都假设有一个名为 *test* 的数据库：

```
CREATE TABLE dbo.employee(
            emp_lname   varchar(15)   NOT NULL,
            emp_fname   varchar(10)   NOT NULL,
```

```
                  address     varchar(30)  NOT NULL,
                  phone       char(12)     NOT NULL,
                  job_level   smallint     NOT NULL
);
```

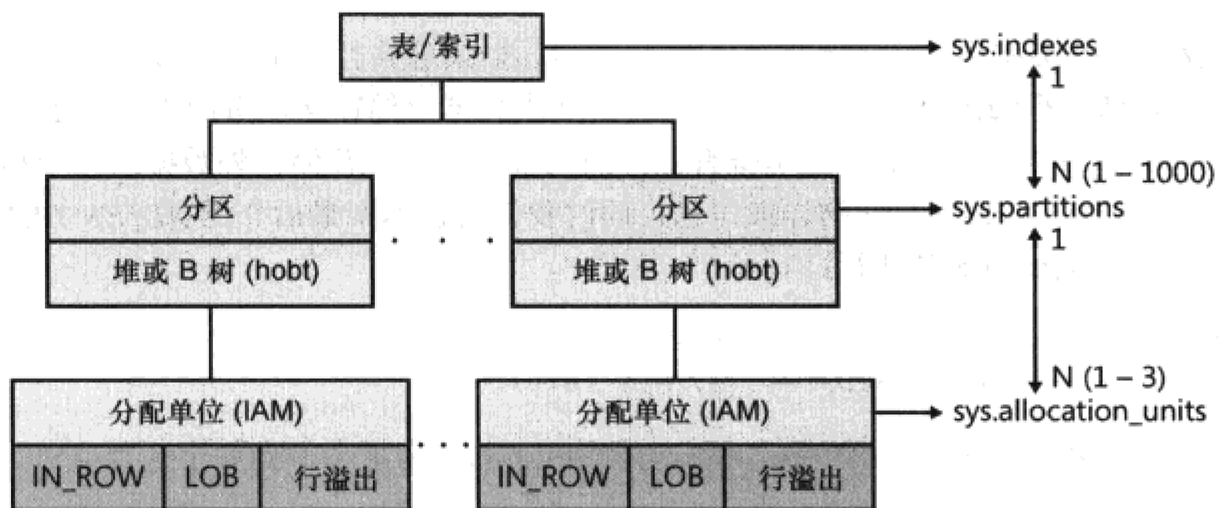


图 5-7 *sys.indexes*、*sys.partitions* 和 *sys.allocation_units* 之间的关系

这张表在 *sys.indexes* 和 *sys.partitions* 中分别有一行，这一点可以在运行下面的查询时看到。我们只需包含 *sys.indexes* 中的几列，但是 *sys.partitions* 仅有 6 列，因此我们检索到了所有列：

```
SELECT object_id, name, index_id, type_desc
FROM sys.indexes
WHERE object_id=object_id('dbo.employee');

SELECT *
FROM sys.partitions
WHERE object_id=object_id('dbo.employee');
```

下面是运行得到的结果（您运行得到的结果可能有点不同，因为您的 ID 值可能不同）：

```
object_id    name     index_id type_desc
-----------  -------  ----------  ------------
5575058      NULL     0           HEAP

partition_id      object_id    index_id partition_number hobt_id           rows
----------------- ----------   --------      -----------------  ------------------ -----
72057594038779904 5575058      0             1                  72057594038779904 0
```

sys.allocation_units 视图中的每一行都有一个 *unique allocation_unit_id* 值。每一行还有一个 *container_id* 列值，该列可以在 *sys.partitions* 中与 *partition_id* 合并，如下面的查询所示：

```
SELECT object_name(object_id) AS name,
    partition_id, partition_number AS pnum, rows,
    allocation_unit_id AS au_id, type_desc as page_type_desc,
    total_pages AS pages
FROM sys.partitions p JOIN sys.allocation_units a
   ON p.partition_id = a.container_id
WHERE object_id=object_id('dbo.employee');
```

同样，对于这种简单的表来说，我们只得到一行，因为只有一个分区，没有聚集索引，并且只有一

种类型的数据（(IN_ROW_DATA)。下面是运行的结果：

```
name     partition_id      pnum   rows   au_id             page_type_desc  pages
-----    ---------------   ------ ------ ----------------- --------------  -----
employee 72057594038779904 1      0      72057594043301888 IN_ROW_DATA     0
```

现在在存储在其他类型页面上的表中添加一些新列。如果总共行大小超出最大值 8060 字节，那么 *varchar* 数据可以存储在行溢出页面上。默认情况下，文本数据存储在文本页面上。对于存储在行溢出页面上的 *varchar* 数据及文本数据来说，行本身有额外的系统开销用于存储行外数据的一个指针。我们将在本节后面介绍行溢出和文本数据存储的详细信息，同时我们将在本章最后介绍 *ALTER TABLE*，现在只研究一下 *sys.allocation_units* 中的其他行：

```
ALTER TABLE dbo.employee ADD resume_short varchar(8000);
ALTER TABLE dbo.employee ADD resume_long text;
```

如果运行前面连接 *sys.partitions* 和 *sys.allocation_units* 的查询，会得到如下 3 行：

```
name     partition_id        pnum   rows   au_id               page_type_desc     pages
-------  ------------------- ------ ------ ------------------- ------------------ -----
employee 72057594038779904   1      0      72057594043301888   IN_ROW_DATA        0
employee 72057594038779904   1      0      72057594043367424   ROW_OVERFLOW_DATA  0
employee 72057594038779904   1      0      72057594043432960   LOB_DATA            0
```

您可能还希望添加一个或两个索引并再次检查这些目录视图的内容。您应该注意只添加一个聚集索引不会修改 *sys.allocation_units* 中的行数，但却会修改 *partition_id* 的编号，因为创建一个聚集索引时，整张表会在系统内部重建。添加一个非聚集索引至少会向 *sys.allocation_units* 中再添加一行从而跟踪该索引的页面。下面的查询链接有 3 个视图 *sys.indexes*、*sys.partitions* 和 *sys.allocation_units*，从而显示表的名称、索引名和类型、页面类型及 *dbo.employee* 表的空间使用信息：

```
SELECT convert(char(8),object_name(i.object_id)) AS table_name,
    i.name AS index_name, i.index_id, i.type_desc as index_type,
    partition_id, partition_number AS pnum, rows,
    allocation_unit_id AS au_id, a.type_desc as page_type_desc,
    total_pages AS pages
FROM sys.indexes i JOIN sys.partitions p
        ON i.object_id = p.object_id AND i.index_id = p.index_id
    JOIN sys.allocation_units a
        ON p.partition_id = a.container_id
WHERE i.object_id=object_id('dbo.employee');
```

由于我们没有向表插入任何数据，因此会发现行和页的值都是 0。当我们讨论实际的页面结构时，会向表中插入数据从而可以看到数据的内部存储。到目前为止，我们所运行的查询都没有为我们提供关于各种分配单位中页面位置的任何信息。在 SQL Server 2000 中，*sysindexes* 表包含只是数据存储位置的 3 个列，这些列被称为 *first*、*root* 和 *firstIAM*。这些列在 SQL Server 2008 中仍然可用（但是名称稍微有所不同，分别是 *first_page*、*root_page* 和 *first_iam_page*），但是只能在被称为 *sys.system_internals_allocation_units* 的未记录视图中看到。这个视图除这 3 个附加列的附加内容不同外，其他均与 *sys.allocation_units* 相一致，因此您可以在前面的分配查询中用 *sys.system_internals_allocation_units* 代替 *sys.allocation_units* 并向选择列表中添加这 3 个额外的列。请记住，作为一个未记录的对象，该视图只在内部使用并且易于修改（这

与以 *system_internals* 开头的其他视图一样）。不保证向前兼容。

5.4.3 数据页

数据页是包含已经被添加到某个数据库表中的用户数据的结构。正如我们在前面看到的那样，有 3 种数据页面，每个页面都以一种不同的格式存储数据。它们分别是用于行内数据、行溢出数据和 LOB 数据的页面。与 SQL Server 中所有其他类型的页面一样，数据页面具有 8KB 或 8192 字节的固定长度。它们由 3 个主要部分组成，分别是页眉、数据行和行偏移数组，如图 5-8 所示。

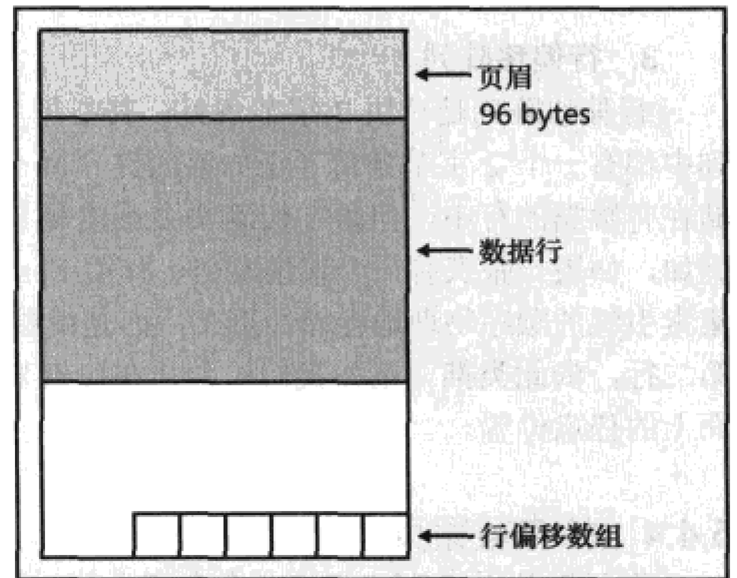


图 5-8 一个数据页的结构

1. 页眉

正如在图 5-8 中看到的那样，页眉占用每个数据页前面的 96 字节（剩下 8096 字节用于数据、行系统开销和行偏移）。表 5-5 所示为我们检查页眉时显示的一些信息。

表 5-5 检查页眉时显示的信息

Field（字段）	含 义
pageID	数据库中该页的页编号和文件编号
nextPage	如果该页位于一个页链中，则该字段表示下一页的文件编号和页编号
prevPage	如果该页位于一个页链中，则该字段表示上一页的文件编号和页编号
Metadata: ObjectId	该页所在对象的 ID
Metadata: PartitionId	该页所在分区的 ID
Metadata: AllocUnitId	包含该页的分配单元的 ID
LSN	与修改该页的最后日志目录相对应的日志序列号（LSN）
slotCnt	该页上所用总的槽（行）数
Level	在索引页中的级别（0 为子叶节点）
indexId	该页的索引 ID（0 为数据页）
freeData	该页面中的第一个可用空间的字节偏移量
Pminlen	行的固定长度部分的字节数
freeCnt	页面上可用字节数
reservedCnt	由所有事务保留的字节数
Xactreserved	由最近启动的事务保留的字节数
tornBits	每个扇区 1 位，用于检测残缺页写入（或者 *torn_page_detection* 没有开启时的校验和信息）
fagBits	包含关于页面其他信息的 2 字节位图

2. 行内数据的数据行

页眉下面是表的真实数据行的存储区域。单个数据行的最大容量是 8060 字节的行内数据。行也可能

有行溢出和存储在独立页上的LOB数据。存储在给定页上的行数随着表结构和存储数据的不同而不同。所有列都是固定长度的，表的每一页存储的行数总是相同的，可变长度行可以根据输入数据的实际长度存储。使行的长度尽可能短可以使一页上能够存储更多的行，这样可以降低I/O并提高缓冲区命中率。

3. 行偏移阵列

行偏移阵列是一块2字节条目，其中每个目录都表示相应数据行起始页的偏移量。每一行在这个阵列中都有一个2字节条目（正如前面提到每一行需要10字节的系统开销时那样）。虽然这些字节不是存储在有数据的行中，但是它们确实会影响每一页所适合的行数。行偏移阵列表示一页上行的逻辑顺序。例如，如果一张表有一个聚集索引，SQL Server会按照聚集索引键的顺序存储行。这并不是说行按照聚集索引键的顺序物理地存储在页上，而是偏移阵列中的槽0引用聚集索引键顺序中的第一行，槽1引用第二行，依此类推。正如我们一会儿在检查某一真实页面时将看到的那样，这些行的物理位置可以是页面上的任意位置。

5.4.4 检查数据页

您可以利用*DBCC PAGE*命令查看某一数据页上的内容，利用该命令可以查看数据库中任意给定页面的页眉、数据行和行偏移表。只有系统管理员可以使用*DBCC PAGE*命令。但是由于您一般不需要查看数据页的内容，因此不会在SQL Server文档中看到*DBCC PAGE*的信息。不过，当您需要使用*DBCC PAGE*时，可以按照如下语法进行操作：

```
DBCC PAGE ({dbid | dbname}, filenum, pagenum[, printopt])
```

*DBCC PAGE*命令包含的参数如表5-6所示。表5-6中的代码和结果显示了*printopt*值设置为1时*DBCC PAGE*命令的输出样例。注意*DBCC TRACEON(3604)*指示SQL Server将结果返回给客户。没有这一跟踪标志时，*DBCC PAGE*命令不返回任何输出结果。

表5-6 **DBCC PAGE命令的参数**

Parameter（参数）	说　明
Dbid	包含该页的数据库的ID
Dbname	包含该页的数据库的名称
Filenum	包含该页的文件的编号
Pagenum	文件内的页编号
Printopt	一个可选的打印选项，可以取如下值。 ■ 0。默认值；打印缓冲区标题和页眉。 ■ 1。输出缓冲区的标题、页眉(分别输出每一行)，以及行偏移量表。 ■ 2。输出缓冲区的标题、页眉(整体输出页面)，以及行偏移量表。 ■ 3。输出缓冲区的标题、页眉(分别输出每一行)，以及行偏移量表；每一行后分别列出它的列值

```
DBCC TRACEON(3604);
GO
DBCC PAGE (pubs, 1, 157, 1);
GO
PAGE: (1:157)
BUFFER:
```

```
BUF @0x038E697C
bpage = 0x0C3AA000                bhash = 0x00000000           bpageno = (1:157)
bdbid = 11                        breferences = 0              bUse1 = 60722
bstat = 0xc00009                  blog = 0x3212159             bnext = 0x00000000
PAGE HEADER:
Page @0x0C3AA000
m_pageId = (1:157)                m_headerVersion = 1          m_type = 1
m_typeFlagBits = 0x4              m_level = 0                  m_flagBits = 0x200
m_objId (AllocUnitId.idObj) = 27 m_indexId (AllocUnitId.idInd) = 256
Metadata: AllocUnitId = 72057594039697408
Metadata: PartitionId = 72057594038779904 Metadata: IndexId = 1
Metadata: ObjectId = 2105058535 m_prevPage = (0:0) m_nextPage = (0:0)
pminlen = 24                      m_slotCnt = 23               m_freeCnt = 6010
m_freeData = 2136                 m_reservedCnt = 0            m_lsn = (18:350:2)
m_xactReserved = 0                m_xdesId = (0:0)             m_ghostRecCnt = 0
m_tornBits = 1967525613

Allocation Status
GAM (1:2) = ALLOCATED           SGAM (1:3) = NOT ALLOCATED
PFS (1:1) = 0x60 MIXED_EXT ALLOCATED 0_PCT_FULL          DIFF (1:6) = CHANGED
ML (1:7) = NOT MIN_LOGGED

DATA:
Slot 0, Offset 0x631, Length 88, DumpStyle BYTE
Record Type = PRIMARY_RECORD  Record Attributes = NULL_BITMAP VARIABLE_COLUMNS
Record Size = 88
Memory Dump @0x6292C631
00000000: 30001800 34303820 3439362d 37323233 †0...408 496-7223
00000010: 43413934 303235ff 09000000 05003300 †CA94025ÿ .....3.
00000020: 38003f00 4e005800 3137322d 33322d31 †8.?.N.X.172-32-1
00000030: 31373657 68697465 4a6f686e 736f6e31 †176WhiteJohnson1
00000040: 30393332 20426967 67652052 642e4d65 †0932 Bigge Rd.Me
00000050: 6e6c6f20 5061726b ††††††††††††††††††nlo Park

Slot 1, Offset 0xb8, Length 88, DumpStyle BYTE
Record Type = PRIMARY_RECORD       Record Attributes = NULL_BITMAP VARIABLE_COLUMNS
Record Size = 88
Memory Dump @0x6292C0B8
00000000: 30001800 34313520 3938362d 37303230 †0...415 986-7020
00000010: 43413934 363138ff 09000000 05003300 †CA94618ÿ .....3.
00000020: 38004000 51005800 3231332d 34362d38 †8.@.Q.X.213-46-8
00000030: 39313547 7265656e 4d61726a 6f726965 †915GreenMarjorie
00000040: 33303920 36337264 2053742e 20233431 †309 63rd St. #41
00000050: 314f616b 6c616e64 ††††††††††††††††††1Oakland

Slot 2, Offset 0x110, Length 85, DumpStyle BYTE
Record Type = PRIMARY_RECORD       Record Attributes = NULL_BITMAP VARIABLE_COLUMNS
Record Size = 85
Memory Dump @0x6292C110
00000000: 30001800 34313520 3534382d 37373233 †0...415 548-7723
00000010: 43413934 373035ff 09000000 05003300 †CA94705ÿ .....3.
00000020: 39003f00 4d005500 3233382d 39352d37 †9.?.M.U.238-95-7
00000030: 37363643 6172736f 6e436865 72796c35 †766CarsonCheryl5
00000040: 38392044 61727769 6e204c6e 2e426572 †89 Darwin Ln.Ber
00000050: 6b656c65 79††††††††††††††††††††††††††keley
```

```
/* Data for slots 3 through 20 not shown */

Slot 21, Offset 0x1c0, Length 89, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes = NULL_BITMAP VARIABLE_COLUMNS
Record Size = 89
Memory Dump @0x6292C1C0
00000000: 30001800 38303120 3832362d 30373532 †0...801 826-0752
00000010: 55543834 313532ff 09000000 05003300 †UT84152ÿ .....3.
00000020: 39003d00 4b005900 3839392d 34362d32 †9.=.K.Y.899-46-2
00000030: 30333552 696e6765 72416e6e 65363720 †035RingerAnne67
00000040: 53657665 6e746820 41762e53 616c7420 †Seventh Av.Salt
00000050: 4c616b65 20436974 79††††††††††††††††Lake City

Slot 22, Offset 0x165, Length 91, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes = NULL_BITMAP VARIABLE_COLUMNS
Record Size = 91
Memory Dump @0x6292C165
00000000: 30001800 38303120 3832362d 30373532 †0...801 826-0752
00000010: 55543834 313532ff 09000000 05003300 †UT84152ÿ .....3.
00000020: 39003f00 4d005b00 3939382d 37322d33 †9.?.M.[.998-72-3
00000030: 35363752 696e6765 72416c62 65727436 †567RingerAlbert6
00000040: 37205365 76656e74 68204176 2e53616c †7 Seventh Av.Sal
00000050: 74204c61 6b652043 697479†††††††††††††t Lake City

OFFSET TABLE:
Row - Offset
22 (0x16) - 357 (0x165)
21 (0x15) - 448 (0x1c0)
20 (0x14) - 711 (0x2c7)
19 (0x13) - 1767 (0x6e7)
18 (0x12) - 619 (0x26b)
17 (0x11) - 970 (0x3ca)
16 (0x10) - 1055 (0x41f)
15 (0xf) - 796 (0x31c)
14 (0xe) - 537 (0x219)
13 (0xd) - 1673 (0x689)
12 (0xc) - 1226 (0x4ca)
11 (0xb) - 1949 (0x79d)
10 (0xa) - 1488 (0x5d0)
9 (0x9) - 1854 (0x73e)
8 (0x8) - 1407 (0x57f)
7 (0x7) - 1144 (0x478)
6 (0x6) - 96 (0x60)
5 (0x5) - 2047 (0x7ff)
4 (0x4) - 884 (0x374)
3 (0x3) - 1314 (0x522)
2 (0x2) - 272 (0x110)
1 (0x1) - 184 (0xb8)
0 (0x0) - 1585 (0x631)

DBCC execution completed. If DBCC printed error messages, contact your system administrator.
```

正如您所见，*DBCC PAGE* 的输出结果分成 4 个主要部分，分别是 BUFFER、PAGE HEADER、DATA 和 OFFSET TABLE（实际上是偏移阵列）。BUFFER 部分显示给定页面的缓冲区信息。在这里，*缓冲区*是

指管理该页面的内存中的结构，同时只有当该页位于内存中时该区域的信息才是有关系的。

DBCC PAGE 输出结果中的 PAGE HEADER 部分显示该页上所有标题字段的数据（表 5-5 显示了这些字段中大部分字段的含义）。数据区域包含每一行的信息。当 *DBCC PAGE* 的 *printopt* 参数值设置为 1 或 3 时，*DBCC PAGE* 会显示每一行的槽位置、页面上行的偏移及行的长度。行数据被分成 3 个部分。左边的列显示被显示数据所在行的字节位置。下一个区域包含存储在该页上的实际数据，以 4 列显示，其中每一列是 8 个十六进制数。右边的列包含该数据的 ASCII 字符表示法。虽然可以显示其他一些数据，但是这一列中只有字符数据是可读的。OFFSET TABLE 区域显示页面结尾处行偏移阵列的内容。在 *DBCC PAGE* 的输出结果中，您可以看到该页包含 23 行，其中第一行（用 0 槽表示）的偏移量是 1585（0×631）。物理上存储在该页上的第一行实际上是第 6 行，行偏移阵列的偏移量为 96。*printopt* 参数值设置为 1 的 DBCC PAGE 命令按照槽编号显示行，即使如此，正如您可以通过每一个槽的偏移量所看到的那样，这并不是行在页面上的物理存储顺序。如果将 *DBCC PAGE* 的 *printopt* 参数值设置为 2，则会发现该页面上（页眉后面）的 8096 字节按照在页面上存储的顺序显示。

5.4.5 数据行的结构

表的数据行具有如图 5-9 所示的常规结构（只要数据是以非压缩形式存储的）。我们将这种格式称为 FixedVar 格式，因为所有固定长度列的数据首先被存储，接下来存储所有可变长度列的数据。表 5-7 显示了存储在 FixedVar 行中的信息（在第 7 章中，我们会看到以不同格式存储的行格式，当页面上的数据被压缩时使用）。状态位 A 包含指示行属性的一个位图。状态位具有如下意义。

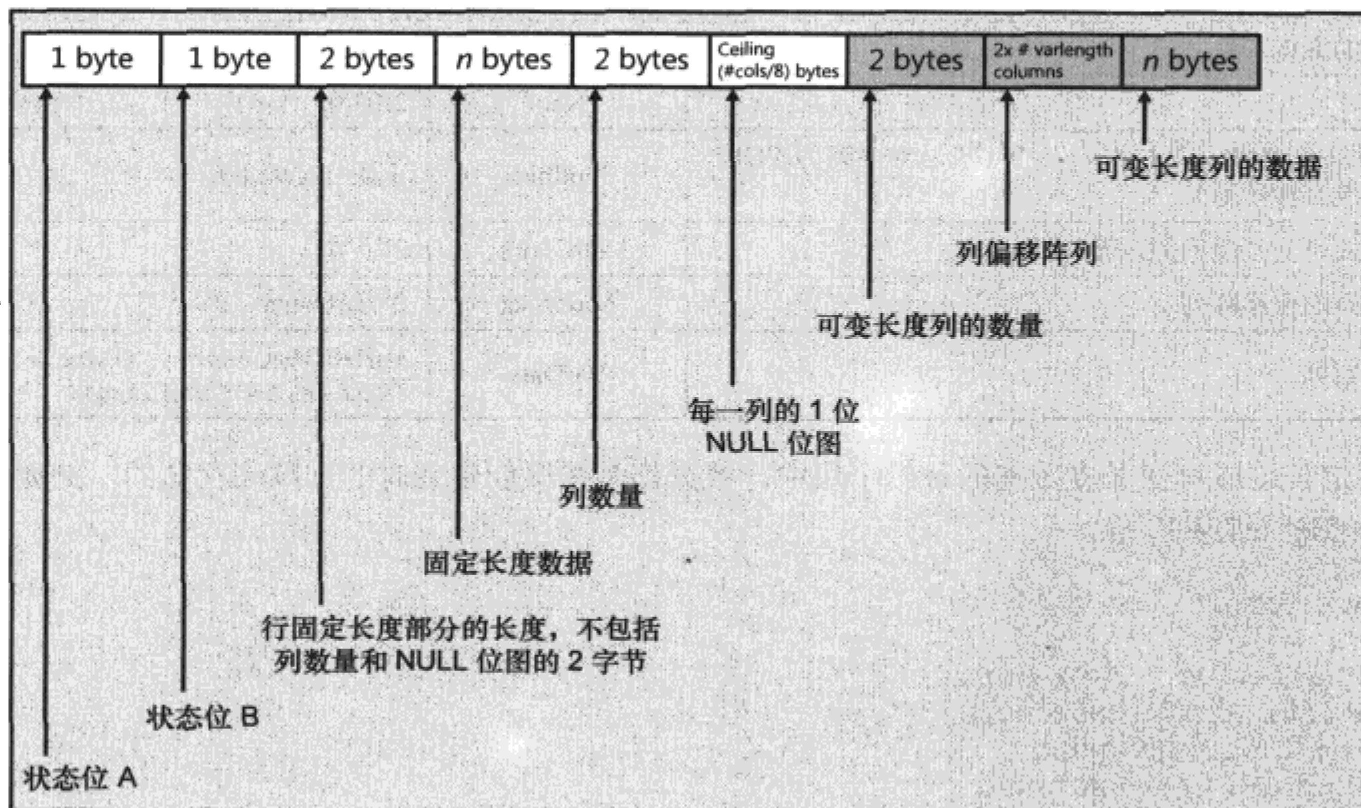


图 5-9 数据行的结构

- **位 0**。版本信息。在 SQL Server 2008 中，该值始终为 0。
- **位 1 到位 3**。被当做一个 3 位值，0 表示一条原始记录，1 表示一条转发记录，2 表示一个转发存根，3 表示一条索引记录，4 表示一条 blob 段或行溢出数据，5 表示一条备份索引记录，6 表

示一条备份数据记录，7 表示一条备份版本记录（我们将在本章后面的“移动行”部分及第 6 章备份记录部分介绍转发记录）。

- **位 4**。表示存在一个 NULL 位图。在 SQL Server 2008 中，总有一个 NULL 位图，即使所有列中都不允许 NULL 值。
- **位 5**。表示行中存在可变长度列。
- **位 6**。表示包含版本信息的行。
- **位 7**。在 SQL Server 2008 中不使用。

状态位 B 字段中只使用一位，表示该记录是一条备份转发记录。

可以在图 5-9 和表 5-7 中看到，第 3 和第 4 字节表示行固定长度部分的长度。正如图 5-9 所示那样，这是不包括 2 字节的列数量以及 NULL 位图的长度，是根据表中列的总体长度确定的可变长度。解释这些位数据的另一种方法是，将它看做在行中查看列数量的位置。例如，如果第 3 和第 4 字节（2～3 字节）包含 0×0016（即十进制的 22）值，则不仅表示行中列数量值之前有 22 字节，而且表示列的数量值位于第 22 字节处。在本章后面章节中的一些图示中，2～3 字节可能表示列数量的位置。

表 5-7 表数据行中存储的信息

信息	助记符	大小
状态位 A	TagA	1 字节
状态位 B	TagB	1 字节
固定长度的大小	Fsize	2 字节
固定长度的数据	Fdata	Fsize–4
列数量	Ncol	2 字节
NULL 位图(表中每一列 1 字节；一个 1 表示对应列为 NULL 或这一位没有被使用)	Nullbits	上限（Ncol/8）
行中存储的可变长度列的数量	VarCount	2 字节
可变长度列的偏移阵列	VarOffset	2*VarCount
可变长度数据	VarData	VarOff[VarCount] – (Fsize + 4 +上限 (Ncol / 8) + 2 * VarCount)

在固定长度或可变长度数据的每一个块中，数据都是按照创建表时的列顺序存储的，例如，假设用如下语句创建一张表：

```
CREATE TABLE Test1
(
Col1 int          NOT NULL,
Col2 char(25)     NOT NULL,
Col3 varchar(60)  NULL,
Col4 money        NOT NULL,
Col5 varchar(20)  NOT NULL
);
```

该行中固定长度数据部分包含 *Col1* 的数据，接下来是 *Col2* 的数据和 *Col4* 的数据。可变长度数据部分包含 *Col3* 的数据，接下来是 *Col5* 的数据。对于仅包含固定长度数据的行来说，下面的规律总是适用的。

- 数据行第一字节的第一个十六进制数是 1，表示没有变长列存在（第一个十六进制数包含 4～7 位；位 6 和 7 总为 0；如果没有可变长度列，则位 5 总是 0，位 4 总是 1，因此 4 个位的值显示为 1）。
- 数据行在 NULL 位图后结束，位于固定长度的数据之后（也就是说，图 5-9 中显示的阴影部分在只有固定长度数据的行中是不存在的）。
- 每个数据行的整体长度是相同的。

一个具有可变长度列的数据行在具有 2 字节目录（用于每个非 NULL 可变长度列）的数据行中有一个列偏移阵列，表示行中列的结束位置（*偏移*和*位置*是不同的。*偏移*是基于 0 的，而*位置*则基于 1。偏移量为 7 的一个字节位于行中的第 8 个字节处）。存储具有一个 NULL 值的可变长度列存在一些特殊问题，我们将在本章后面的“NULL 和可变长度列”部分对这一问题进行介绍。

5.4.6 查找一个物理页面

在检查特殊数据之前，需要先补充一点。下面的示例利用 *DBCC PAGE* 命令检查物理数据库页面。为了运行该命令，我们需要知道哪些页编号用于存储表中的行。前面我们曾经提到过 first_page 的值被存储在一个称为 *sys.system_internals_allocation_units* 的未记录视图中，该视图与 *sys.allocation_units view* 视图相同。首先让我们创建一张表（该表将在后面部分使用到）并向其中插入一行数据：

```
USE tempdb;
CREATE TABLE Fixed
(
Col1 char(5) NOT NULL,
Col2 int NOT NULL,
Col3 char(3) NULL,
Col4 char(6) NOT NULL
);
INSERT Fixed VALUES ('ABCDE', 123, NULL, 'CCCC');
```

通过下面的查询得到 *Fixed* 表中 *first_page* 的值：

```
SELECT object_name(object_id) AS name,
     rows, type_desc as page_type_desc,
     total_pages AS pages, first_page
FROM sys.partitions p JOIN sys.system_internals_allocation_units a
    ON p.partition_id = a.container_id
WHERE object_id=object_id('dbo.Fixed');
RESULTS:
name     rows     page_type_desc     pages     first_page
-----    ----     --------------     -----     --------------
Fixed    1        IN_ROW_DATA        2         0xCF0400000100
```

接下来我们可以从 *preceding sys.system_internals_ allocation_units* 输出结果取出 *first_page* 值并将其转变成一个文件和页面地址（您得到的 *first_page* 值很可能与我们得到的不同）。在十六进制表示法中，两个十六进制数中的每一组都表示一个字节。我们必须先转换这些字节来得到 00 01 00 00 04 CF。前两组表示 2 字节文件编号，后 4 组表示页编号。因此文件是 0x0001（即 1），页编号是 0x000004CF（即十进制的 1231）。

除非您特别喜欢进行十六进制转换，否则使用其他 3 种选项的一种来确定与 SQL Server 表相关的实

际页编号。首先，您可以创建一个函数来实现 6 字节十六进制页编号值（如 0xCF0400000100）到 file_number:page_number 格式的转换：

```
CREATE FUNCTION convert_page_nums (@page_num binary(6))
   RETURNS varchar(11)
AS
  BEGIN
   RETURN(convert(varchar(2), (convert(int, substring(@page_num, 6, 1))
          * power(2, 8)) +
             (convert(int, substring(@page_num, 5, 1)))) + ':' +
               convert(varchar(11),
   (convert(int, substring(@page_num, 4, 1)) * power(2, 24)) +
   (convert(int, substring(@page_num, 3, 1)) * power(2, 16)) +
   (convert(int, substring(@page_num, 2, 1)) * power(2, 8)) +
   (convert(int, substring(@page_num, 1, 1)))) )
END;
```

接下来可以执行如下的 SELECT 语句来调用该函数：

```
SELECT dbo.convert_page_nums(0xCF0400000100);
```

您会得到结果 *1:1231*。

警告：

在 SQL Server 中，*sys.system_internals_allocation_units* 中的 *first_page* 列不总表示表的第一页（毕竟该视图是非正式文件的）。我们已经发现 *first_page* 是可靠的，除非您对表中数据执行删除和更新操作。

确定实际页编号的第二个选项是使用另一个非正式文件的命令 *DBCC IND*。由于返回的大部分信息都只与索引相关，因此我们将在第 6 章详细介绍该命令。但是，为了试验，您可以运行如下命令并注意输出结果（标记为 *PageFID* 和 *PagePID*）中行 *PageType=1*（表示该页是一个数据页）中前两列的值：

```
DBCC IND(tempdb, Fixed, -1);
```

如果您不是位于 *tempdb* 中，则应该用创建该表时所在的数据库的名称来替换这里的 *tempdb*。*PageFID* 和 *PagePID* 的值应该与您转换 *first_page* 的十六进制字符串值时使用的值相同。在我的运行里，我看到的 *PageFID* 值是 1，*PagePID* 值是 1231。因此这些就是我调用 *DBCC PAGE* 时所使用的值。

获得文件和页编号信息的第 3 种方法涉及使用一个未记录文件函数 *sys.fn_PhysLocFormatter* 和一个未记录值 *%%physloc%%* 来返回结果行中的物理行位置及表中的数据值。如果您希望找到表中的哪一页包含特殊值，那么这种方法很有用。*DBCC IND* 可用于查找表中的所有页，而不是包含一个特殊行的特殊页。但是，*sys.fn_PhysLocFormatter* 只显示 SELECT 语句中返回的数据的数据页。我们可以使用该函数获得表 *Fixed* 中数据使用的页面，语句如下：

```
SELECT sys.fn_PhysLocFormatter (%%physloc%%) AS RID, * FROM Fixed;
GO
```

下面是我得到的结果：

```
RID           Col1     Col2             Col3    Col4
------------ ------- -------------     ------  ------
(1:1231:1)    ABCDE    123              NULL    CCCC
```

得到 *FileID* 和 *PageID* 值后，就可以使用 *DBCC PAGE*。对于一个更大型的表来说，我们可以使用 *sys.fn_PhysLocFormatter* 来获得 WHERE 子句条件中返回的特殊行的页面。

警告：

*%%physloc%%*值是关系引擎所不能理解的，也就是说，如果在一个 WHERE 子句中使用*%%physloc%%*，则 SQL Server 必须要检查每一行才能知道哪些行位于*%%physloc%%*所指示的页面上。不能使用*%%physloc%%*来查找行。看待这一问题的另一种方式是*%%physloc%%*可以作为报告某一物理行位置的输出结果而返回，但是不能用做查找表中某一特殊位置的输入。*%%physloc%%*值被作为 SQL Server 产品开发小组的一种调试功能而引入，不打算在产品应用程序中使用（也不被支持）。

下面两个实例说明了如何存储固定长度和可变长度数据行。

5.4.7 固定长度行的存储

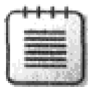
首先，让我们利用刚刚在上一节中建立的表来看一下长度都是固定的行的简单示例：

```
CREATE TABLE Fixed
(
Col1 char(5) NOT NULL,
Col2 int NOT NULL,
Col3 char(3) NULL,
Col4 char(6) NOT NULL
);
```

创建该表时，您应该能够根据 *sys.indexes* 和 *sys.columns* 视图执行如下查询，获得与下面结果类似的信息：

```
SELECT object_id, type_desc,
    indexproperty(object_id, name, 'minlen') as min_row_len
    FROM sys.indexes where object_id=object_id('Fixed');

SELECT column_id, name, system_type_id, max_length as max_col_len
FROM sys.columns
WHERE object_id=object_id('Fixed');

RESULTS:
object_id        type_desc     minlen
-------------    ------------ -------
53575229         HEAP          22

column_id        name                    system_type_id   max_length
-------------    --------------------    ---------------  ----------
1                Col1                    175              5
2                Col2                    56               4
```

```
3        Col3        175        3
4        Col4        175        6
```



注意：

sysindexes 兼容性视图包含 *minlen* 和 *xmaxlen* 列，用于存储某一行的最大和最小长度。在 SQL Server 2008 中，这些值在所有目录视图中均不可见，但是您可以利用 *indexproperty* 函数的未记录文件参数获取这些值。与所有未记录文件的功能一样，请记住它们不受 Microsoft 的支持同时也不保证将来的兼容性。

对于仅包含固定长度列的表来说，利用 *indexproperty* 函数返回的 *minlen* 值等于列长度（取自 *sys.columns.max_length*）加上 4 字节的和。其中不包括用于存储列数目的 2 字节，也不包括用于 NULL 位图的字节。

为了查看该表中某一具体的数据行，您必须首先插入一个新行。如果在前面的操作中还没有插入行，可以利用下面的语句进行插入：

```
INSERT Fixed VALUES ('ABCDE', 123, NULL, 'CCCC');
```

图 5-10 显示了数据页上该行的实际内容。

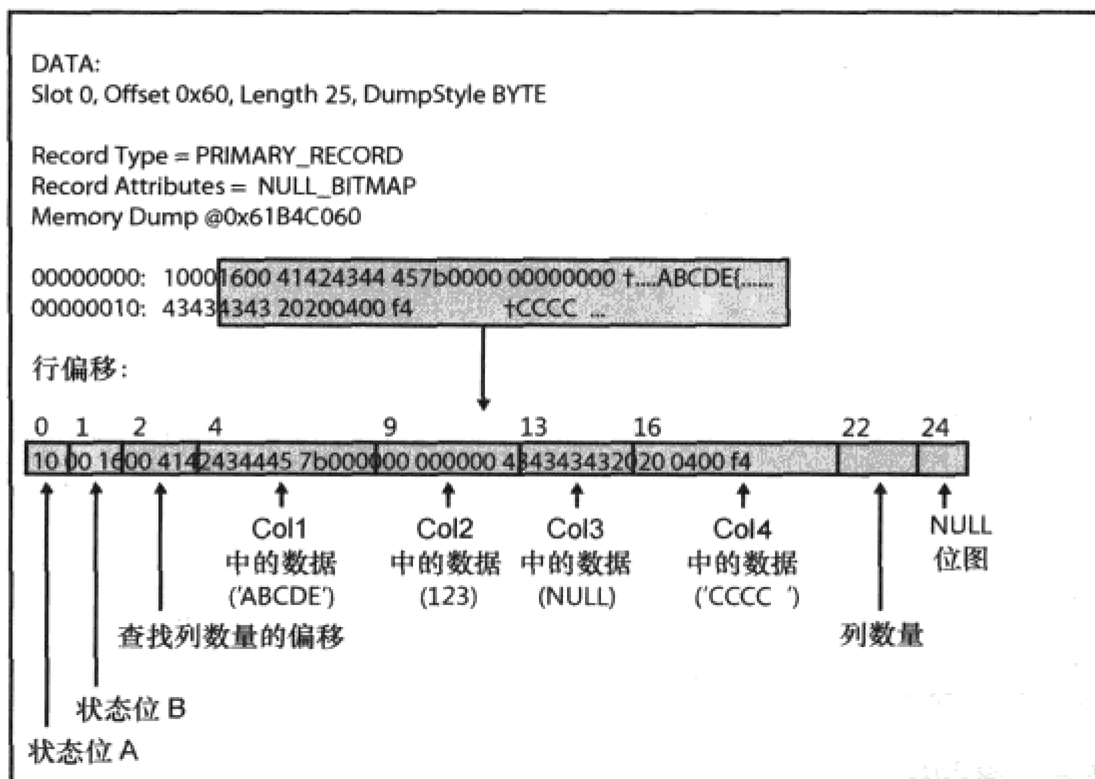


图 5-10 包含的所有列都是固定长度的一个数据行

通过前面介绍的任意一种方法获得的文件和页编号，然后通过运行 *DBCC PAGE* 命令获得页面内容：

```
DBCC PAGE(tempdb, 1, 1231, 1);
```

读取 *DBCC PAGE* 的输出结果需要一些实践。首先，注意到输出结果每次显示多组 4 字节的数据行。图 5-10 中的阴影区域已经被扩展，从而以扩展的形式显示字节。

第一个字节是状态位 A，其值（0x10）表示只有第 4 位开启，同时由于第 5 位没有开启，因此我们

知道该行没有可变长度列。该行中的第 2 个字节（状态位 B）没有被使用。第 3 和第 4 个字节（1600）表示固定长度字段的长度，这也是 *Ncol* 值所在的列偏移（作为一个多字节数字值，该信息以一种字节交换的形式存储，因此该值实际上是 0x0016，即十进制的 22）。为了知道行中偏移量 4 和 22 之间每一列的真实位置，我们需要知道每一列的偏移值。在 SQL Server 2000 中，*syscolumns* 系统表中有一列表示行内的偏移量。虽然还可以从 SQL Server 2005 的 *syscolumns* 兼容性视图中进行查询，但是获得的结果是不可靠的。偏移量可以在一个名为 *sys.system_internals_partition_columns* 的未记录文件视图中找到，接下来可以将 *sys.system_internals_partition_columns* 和 *sys.partitions* 连接来获得参照对象的信息，也可以将 *sys.system_internals_partition_columns* 与 *sys.columns* 连接来获得每一列的其他信息。

下面是一个返回基本列信息的查询，其中包括每一列的行内偏移量。我们将在本章后面对其他表使用相同的查询，同时我们将它称为 “列细节查询”。

```
SELECT c.name AS column_name, column_id, max_inrow_length,
        pc.system_type_id, leaf_offset
 FROM sys.system_internals_partition_columns pc
    JOIN sys.partitions p
      ON p.partition_id = pc.partition_id
    JOIN sys.columns c
      ON column_id = partition_column_id
         AND c.object_id = p.object_id
WHERE p.object_id=object_id('Fixed');

RESULTS:
column_name      column_id       max_inrow_length       system_type_id  leaf_offset
-------------    -------------   -------------------    --------------- -----------
Col1             1               5                175                   4
Col2             2               4                56                    9
Col3             3               3                175                   13
Col4             4               6                175                   16
```

因此，现在我们可以简单地利用前面结果中的偏移值查找行中每一列的数据：列 *Col1* 的数据从偏移量 4 开始，列 *Col2* 的数据从偏移量 9 开始，依此类推。作为一个 int 型数据，*Col2*（7b000000）中的数据一定要经过字节交换，以获得值 0x0000007b，即等于十进制的 123。

注意 *Col3* 的 3 个字节的数据都是 0，表示列中的一个实际 NULL。由于该行没有可变长度列，因此该行在 *Col4* 列数据后的 3 个字节后结束。偏移量 22 处固定长度数据后的 2 个字节（0400，字节交换后产生 0x0004）表示行内有 4 列。最后的字节是 NULL 位图。0xf4 值的二进制形式是 11110100，位是按照由高到低的顺序显示的。低位的 4 位表示表中有 4 列，0100 表示只有第 3 列实际上是空值。高 4 位是 1111，因为这些位没有被使用。NULL 位图一定有多个 8 位，同时如果列的数目不是多个 8 的倍数，那么一定有某些位未被使用。

5.4.8 可变长度行的存储

现在我们来看一下一个具有可变长度数据的表的更复杂情况。每一行都有 3 个 *varchar* 列和 2 个固定长度列：

```
CREATE TABLE Variable
(
Col1 char(3) NOT NULL,
```

```
Col2 varchar(250) NOT NULL,
Col3 varchar(5) NULL,
Col4 varchar(20) NOT NULL,
Col5 smallint NULL
);
```

当该表被创建时，您应该能够根据 *sys.indexes*、*sys.partitions*、*sys.system_internals_partition_columns* 和 *sys.columns* 视图执行如下查询，从而获得与下面结果类似的信息：

```
SELECT object_id, type_desc,
    indexproperty(object_id, name, 'minlen') as minlen
    FROM sys.indexes where object_id=object_id('Variable');

SELECT name, column_id, max_inrow_length, pc.system_type_id, leaf_offset
 FROM sys.system_internals_partition_columns pc
    JOIN sys.partitions p
          ON p.partition_id = pc.partition_id
    JOIN sys.columns c
          ON column_id = partition_column_id AND c.object_id = p.object_id
WHERE p.object_id=object_id('Variable');

RESULTS:
object_id     type_desc     minlen
-----------   -----------   -----------
69575286      HEAP          9
column_name        column_id      max_inrow_length  system_type_id    leaf_offset
-------------      ------------   ----------------  ----------------  -----------
Col1               1              3                 175               4
Col2               2              250               167               -1
Col3               3              5                 167               -2
Col4               4              20                167               -3
Col5               5              2                 52                7
```

现在您可以利用如下语句向表中插入一行数据：

```
INSERT Variable VALUES
   ('AAA', REPLICATE('X', 250), NULL, 'ABC', 123);
```

这里使用 *REPLICATE* 函数来简化填充列的操作；该函数建立一个 *250 X* 字符串插入到 *Col2* 中。

您可以查看该行的详细信息，如图 5-11 所示，*DBCC PAGE* 输出结果中存储的页面。固定长度列的位置可以利用前面查询结果中 *sys.system_internals_partition_columns* 的 *leaf_offset* 值找到，在这张表中，*Col1* 的偏移量从 4 开始，*Col5* 的偏移量从 7 开始。可变长度列没有在具有特殊字节偏移的查询结果中显示，因为每一行的偏移量可能不同。相反，行本身在被称为 *Column Offset Array* 的部分保存该行中每个可变长度列的结束位置。查询结果显示 *Col2* 中 *leaf_offset* 的值为–1，这表示 *Col2* 是第一个可变长度列；*Col3* 的偏移量是–2，表示 *Col3* 是第 2 个可变长度列，*Col4* 的偏移量是–3，表示 *Col4* 是第 3 个可变长度列。

为了查找数据行本身的可变长度列，首先定位行中的列偏移阵列。在 2 字节字段之后表示列的总数量（0x0500）及 NULL 位图的值 0xe4，2 字节字段的值为 0x0300（或十进制的 3）表示存在 3 个可变长度字段。接下来是列偏移阵列。3 个 2 字节值表示 3 个可变长度列中每一列的结束位置：0x0e01 字节交换为 0x010e，因此第一个变量字节列的结束位置是 270。接下来的 2 字节偏移量也是 0x0e01，因此该列没有长度同时也没有任何数据存储在变量数据区域中。与固定长度字段不同的是，如果一个可变长度字

段有一个 NULL 值，则该数据行不占用任何空间。SQL Server 通过确定 NULL 位图中该字段的位的值是 0 还是 1 来区分包含 NULL 值的 *varchar* 和空字符串。第 3 个 2 字节偏移量是 0x1101，字节交换后得到的值是 0x0111。这表示该行的结束位置是 273（并且长度是 273 字节）。

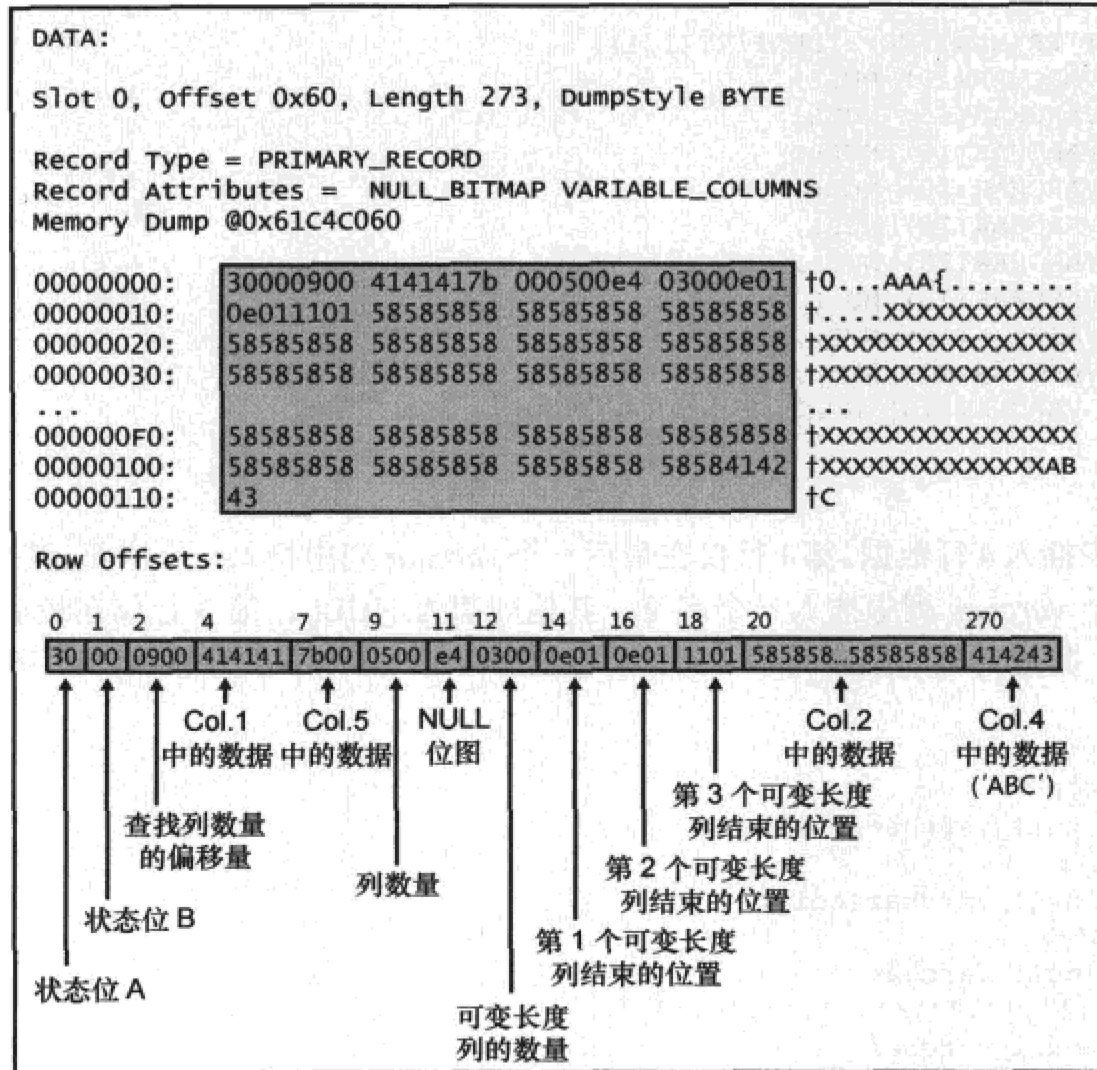


图 5-11 具有可变长度列的一个数据行

一行所需的全部存储空间是由很多因素决定的。可变长度列向一行添加了更多的系统开销，同时列的实际长度可能是不可预测的。即使是固定长度的列，系统开销的字节数量也是可以根据表中列的数量而变的。在本章前面的图 5-2 所示的例子中，我们曾经提到过，如果一行包含的都是固定长度列，则会有 10 字节的系统开销。对于该表来说，10 是正确的数字。NULL 位图的大小需要足够长才能存储行中每一列的一位。在图 5-2 的示例中，表有 11 列，因此 NULL 位图应该是 2 字节。在图 5-10 和图 5-11 所示的例子中，表的列数少于 8 个，因此 NULL 位图只需要 1 字节。请不要忘记整行的系统开销还必须包括 2 字节，用于表示页尾行偏移表中的每一行。

NULL 和可变长度列

正如前面我们提到的那样，固定长度的列始终具有相同的长度，即使列中包含 NULL。对于可变长度列来说，NULL 不占用行中可变长度数据的任何空间。但是，正如我们在图 5-11 中看到的那样，每个可变长度列还有一个 2 字节的列偏移量目录，因此我们不能说它们不占用任何空间。但是，如果一个零长度的值被存储在可变长度数据列的列表结尾处，则 SQL Server 不会存储关于它的任何信息，也不会在

列偏移阵列中包含2字节。现在让我们来看一个例子。

下表允许每个字符列中存在NULL，并且它们都是可变长度的。唯一的固定长度列是integer标识列：

```
CREATE TABLE dbo.null_varchar
   (
     id INT PRIMARY KEY IDENTITY(1,1),
     col1 VARCHAR(10) NULL,
     col2 VARCHAR(10) NULL,
     col3 VARCHAR(10) NULL,
     col4 VARCHAR(10) NULL,
     col5 VARCHAR(10) NULL,
     col6 VARCHAR(10) NULL,
     col7 VARCHAR(10) NULL,
     col8 VARCHAR(10) NULL,
     col9 VARCHAR(10) NULL,
     col10 VARCHAR(10) NULL
   );
GO
```

我们将向该表插入4行数据。第1行仅在最后一个*varchar*列中插入一个字符，其他列都为NULL。第2行仅在第一个*varchar*列中插入一个字符，其他列都为NULL。第3行仅在最后一个*varchar*列中插入一个字符，其他列都为空字符串。第4行仅在第一个*varchar*列中插入一个字符，其他列都为空字符串：

```
SET NOCOUNT ON
INSERT INTO null_varchar(col10)
   SELECT 'a';
INSERT INTO null_varchar(col1)
   SELECT 'b';
INSERT INTO null_varchar
   SELECT '','','','','','','','','','c';
INSERT INTO null_varchar
   SELECT 'd','','','','','','','','','';
GO
```

现在可以利用*DBCC IND*和*DBCC PAGE*（如前所述）查看包含这4行数据的页面。

下面是第一行（列偏移阵列以阴影形式显示）：

```
Slot 0, Offset 0x60, Length 35, DumpStyle BYTE
Record Type = PRIMARY_RECORD Record Attributes = NULL_BITMAP VARIABLE_COLUMNS
Record Size = 35
Memory Dump @0x66B4C060
00000000: 30000800 01000000 0b00fe03 0a002200 †0.........þ...".
00000010: 22002200 22002200 22002200 22002200 †".".".".".".".".
00000020: 230061†††††††††††††††††††††††††††††††#.a
```

在列偏移阵列中有9个条目的值（字节交换后）为十六进制的22（或十进制的18），一个条目的值为十进制的19。前9个位置的值18表示数据与列偏移阵列结束的位置相同，同时SQL Server确定这表示9列的值都为空。但是空可以意味着NULL，也可以表示一个空字符串。通过查看位置11和12处的NULL位图，我们得到fe03，即字节交换后的十六进制03fe。它的二进制形式表示为0000001111111110。列位置从右向左显示。该表只有11列，因此NULL位图中的最后5位被忽略。剩下的字符串表示第一列

和最后一列不为 NULL，但是其他所有列都为 NULL。

列偏移阵列中的第 10 个值是十六进制的 23（或十进制的 19），这表示数据在偏移量 19 处结束，包含的 ASCII 码是 61（代表 a）。

下面是第 2 行（列偏移阵列用阴影表示）：

```
Slot 1, Offset 0x83, Length 17, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes = NULL_BITMAP VARIABLE_COLUMNS
Record Size = 17
Memory Dump @0x66B4C083
00000000: 30000800 02000000 0b00fc07 01001100 †0.........ü.....
00000010: 62†††††††††††††††††††††††††††††††††††b
```

这一行与第一行有几点重要的不同。首先，列偏移阵列仅包含一个值，这是第一个可变长度列的结束位置。1100 字节交换后为 0011，即十进制的 17，这是 ASCII 代码 b（即 62）所在的偏移量。在列偏移阵列之前是表示变长列数量的 2 字节值。这里第 1 行的十六进制值是 000a，表示 10 个可变长度列。第 2 行有 0001，这表示可变长度列中只有一列存储在该行。我们刚发现在包含数据的列之前的零长度列正好使用列偏移阵列，但是在这里，由于所有长度为零的列都位于非 NULL 列之后，因此这里只表示非 NULL 列。如果您查看 NULL 位图，会看到 fc07，字节交换后为 07fc。它的二进制形式表示为 0000011111111100，表示前两列不为 NULL，其他所有列都为 NULL。

如果查看包含空字符串而不是 NULL 的行，则输出结果除 NULL 位图之外应该是完全相同的。这里是第 3 行（槽 2）和第 4 行（槽 3），NULL 位图用阴影表示：

```
Slot 2, Offset 0x94, Length 35, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes = NULL_BITMAP VARIABLE_COLUMNS
Record Size = 35
Memory Dump @0x66B4C094
00000000: 30000800 03000000 0b000000 0a002200 †0.............".
00000010: 22002200 22002200 22002200 22002200 †".".".".".".".".
00000020: 230063†††††††††††††††††††††††††††††††#.c

Slot 3, Offset 0xb7, Length 17, DumpStyle BYTE
Record Type = PRIMARY_RECORD Record Attributes = NULL_BITMAP VARIABLE_COLUMNS
Record Size = 17
Memory Dump @0x66B4C0B7
00000000: 30000800 04000000 0b000000 01001100 †0..............
00000010: 63†††††††††††††††††††††††††††††††††††d
```

对于第 3 行和第 4 行来说，NULL 位图都是 0，这表示没有列为 NULL。第 1 行和第 3 行只有存储的实际字符值及 NULL 位图有所不同。第 2 行和第 4 行也是只有存储的实际字符值及 NULL 位图有所不同。

如果我们在 *varchar* 列中插入一行 NULL，则该行的存储会发生更多的变化。下面是可能出现的结果：

```
Slot 4, Offset 0xc8, Length 12, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes = NULL_BITMAP
Record Size = 12
Memory Dump @0x66B4C0C8
00000000: 10000800 05000000 0b000000 †††††††††.......
```

该行看起像是一个所有列都是固定长度的行而且刚好在 NULL 位图后结束。第一个字节中的第 5 位

（状态位 A）被设置为 0，表示该行中没有可变长度列。

5.4.9 日期和时间数据的存储

在本章前面部分，我们介绍过日期和时间数据类型的存储，现在我们已经有了一些查看实际磁盘存储的经验，就让我们来看一些日期和时间数据。下表在一行中存储了所有不同的日期和时间数据类型，以及时间数据的所有不同的可能范围（记住，*datetime2* 和 *datetimeoffset* 也可以表示时间部分的小数部分，但是时间值与用简单 *time* 数据类型存储的时间值看起来没有区别）。该表还包括单列字符值，这是我刚刚使用的值，因此我可以在 *DBCC PAGE* 提供的一行十六进制数据中轻松找到其他值：

```
CREATE TABLE times (
        a char(1),
        dt1 datetime,
        b char(1),
        sd smalldatetime,
        c char(1),
        dt2 datetime2,
        d char(1),
        dt date,
        e char(1),
        dto datetimeoffset,
        f char(1),
        t time,
        g char(1),
        t0 time(0),
        h char(1),
        t1 time(1),
        i char(1),
        t2 time(2),
        j char(1),
        t3 time(3),
        k char(1),
        t4 time(4),
        l char(1),
        t5 time(5),
        m char(1),
        t6 time(6),
        n char(1),
        t7 time(7));
GO
```

现在我们再插入一行数据，其中时间值与每个日期或时间列提供的值相同。需要日期部分的数据类型的默认日期为 1990 年 1 月 1 日：

```
INSERT INTO times
SELECT
    'a', '01:02:03.123',
    'b', '01:02:03.123',
    'c', '01:02:03.123',
    'd', '01:02:03.123',
    'e', '01:02:03.123',
    'f', '01:02:03.123',
    'g', '01:02:03.123',
```

```
'h', '01:02:03.123',
'i', '01:02:03.123',
'j', '01:02:03.123',
'k', '01:02:03.123',
'l', '01:02:03.123',
'm', '01:02:03.123',
'n', '01:02:03.123';
```

下面是该行的 *DBCC PAGE* 输出结果。我们已经将单字符列数据用阴影表示以作为分隔：

```
00000000: 10005800 61090b11 00000000 00623e00 †..X.a .......b>.
00000010: 00006330 7c27ab08 5b950a64 5b950a65 †..c0|'".[?.d[?.e
00000020: 307c27ab 085b950a 00006630 7c27ab08 †0|'".[?...f0|'".
00000030: 678b0e00 686f9100 6958ae05 6a73cf38 †g?..ho?.iX®.jsÏ8
00000040: 006b7e1a 38026cec 08311600 6d3859ea †.k~.8.lì.1..m8Yê
00000050: dd006e30 7c27ab08 1c000000 0000††††††Ý.n0|'".......
```

表 5-8 显示了其中每个值与十进制格式的转换。这里有几点需要注意。

- 对于 *datetime* 和 *smalldatetime* 数据类型来说，日期值均被存储为 0，这表示日期是基于 1990 年 1 月 1 日的。对于存储一个日期的其他类型来说，日期值被存储为 693595，这表示内部基准日期 0001 年 1 月 1 日后的天数。为了计算相应的日期，可以使用 *dateadd* 函数：

  ```
  SELECT DATEADD(dd, 693595, CAST('0001/1/1' AS datetime2));
  ```

- 这将返回值'1900-01-01' 00:00:00.00'，这是不指定日期时的默认值。
- 秒的小数形式是时间部分的最后 N 个数字，其中的 N 是时间数据的小数位数，在表定义中列出。因此对于 *time(7)*值来说，秒的小数形式是.1230000；对于 *time(4)*值来说，秒的小数形式是.1230；对于 *time(1)*值来说，秒的小数形式是.1；对于 *time(0)*值来说，没有秒的小数形式。
- 不论小时、分钟和秒值的小数位数被删除相应的位数后剩下的时间部分是什么，由于相同的时间值用于表中的所有列，因此时间值都是以相同的四位数 3723 开始的；这里我们将使用取模操作符%和整除法来进行取反。SQL Server 利用下面的转换来确定 3723 的小时、分钟和秒部分：

  ```
  SELECT hours = (3723 / 60) / 60;
  SELECT minutes = (3723 / 60) % 60;
  SELECT seconds = 3723 % 60;

  RESULT:
  hours
  -----------
  1

  minutes
  -----------
  2

  seconds
  -----------
  3
  ```

- 存储 *datetimeoffset* 数据的列用 2 个额外的字节来存储 *timezone* 偏移。之所以需要 2 字节是因

为偏移量被存储为协调世界时（Coordinated Universal Time，UTC）的小时和分钟数（每部分 1 个字节）。

表 5-8 **各种日期和时间值的转换**

列名称	使用的数据类型和字节	存储在行中的值	字节交换日期	时　间	十进制值日期	时　间
dt1	*Datetime* -8-	090b110000 000000	00 00 00 00	00 11 0b 09	0	1116937
sd	*Small datetime* -4-	3e000000	00 00	00 3e	0	62
dt2	*datetime2* -8-	307c27ab0 85b950a	0a 95 5b	08 ab 27 7c 30	693595	37231230000
dt	*date* -3-	5b950a	0a 95 5b	无	693595	无
dto	*datetime offset* -10-	307c27ab08 5b950a00 00	0a 95 5b	08 ab 27 7c 30	693595	37231230000
t	*time* -5-	307c27ab08	无	08 ab 27 7c 30	无	37231230000
t0	*time(0)* -3-	8b0e00	无	00 0e 8b	无	3723
t1	*time(1)* -3-	6f9100	无	00 91 6f	无	37231
t2	*time(2)* -3-	58ae05	无	05 ae 58	无	372312
t3	*time(3)* -4-	73cf3800	无	00 38 cf 73	无	3723123
t4	*time(4)* -4-	7e1a3802	无	02 38 1a 7e	无	37231230
t5	*time(5)* -5-	ec08311600	无	00 16 31 08 ec	无	372312300
t6	*time(6)* -5-	3859eadd00	无	00 dd ea 59 38	无	3723123000
t7	*time(7)* -5-	307c27ab08	无	08 ab 27 7c 30	无	37231230000

5.4.10 sql_variant 数据的存储

sql_variant 数据类型为包含任何或所有 SQL Server 基本数据类型的列提供支持，除了 LOB 和具有 MAX 修饰符的可变长度列、*rowversion*（*timestamp*）、*XML* 及不能为表中每一列定义的类型（即指针和表）。例如，一列可以在某些行中包含一个 smallint 值，在其他行中包含一个 float 值，剩下部分是一个 *char* 值。

该功能用于支持基于 SQL Server 产品中的半结构化数据。这种半结构数据存在于概念表中，概念表具有已知数据类型列的固定数量及一个或多个数据类型事先不知道的可选列。Microsoft Office Outlook 和 Microsoft Exchange 中的电子邮件信息就是一个例子。您可以利用 *sql_variant* 数据类型将一个概念表转换成一个具有多组属性值对的真实且更简洁的表。下面是一个图形化示例：表 5-9 所示的概念表有 3 行数据。固定列是每一行中都有的。每一行还可以有 3 个具有不同数据类型的不同属性

中的一个或多个值。

表 5-9　　一张具有任意数量列和数据类型的概念表

行	固定列	属性 1	属性 2	属性 3
第 1 行	XXXXXX	值 11		值 13
第 2 行	YYYYYY	值 22		
第 3 行	ZZZZZZ	值 31	值 32	

该表可以转换成表 5-10，这里的固定列针对具有这些列的不同属性重复出现。*value* 列可以用 *sql_variant* 数据表示，同时可以是每种不同属性的一个不同数据类型。

表 5-10　　使用 ***sql_variant*** 数据类型存储的半结构化数据

固　定　列	属　　性	值
XXXXXX	属性 1	值 11
XXXXXX	属性 3	值 13
YYYYYY	属性 2	值 22
ZZZZZZ	属性 1	值 31
ZZZZZZ	属性 2	值 32

从系统内部来看，*sql_vairant* 类型的列总被认为是长度可变的。它们的存储结构由数据类型决定，但是每个 *sql_variant* 字段的第一个字节都表示该行中使用的实际数据类型。

我们将创建一个具有 *sql_variant* 列的简单表并向其中插入几行数据，从而使我们可以查看 *sql_variant* 存储的结构。

```
USE testdb;
GO
CREATE TABLE variant (a int, b sql_variant);
GO
INSERT INTO variant VALUES (1, 3);
INSERT INTO variant VALUES (2, 3000000000);
INSERT INTO variant VALUES (3, 'abc');
INSERT INTO variant VALUES (4, current_timestamp);
```

SQL Server 根据提供的数据确定每一行中使用哪种数据类型。例如，第一条插入语句中的 3 被认为是 integer 类型。在第二条插入语句中，*3000000000* 比最大的 integer 值还大，因此 SQL Server 认为它是一个精确度为 10、保留字为 0 的小数（本来可以使用 *bigint*，但是那样会需要更多的存储空间）。现在可以利用 *DBCC IND* 查看表中的第一个页面并利用 *DBCC PAGE* 查看页面中的内容，具体语句如下：

```
DBCC IND (testdb, variant, -1);
-- (I got a value of file 1, page 2508 for the data page in this table)
GO
DBCC TRACEON (3604);
DBCC PAGE (testdb, 1, 2508, 1);
```

图 5-12 显示了 4 行的内容。我们不会讨论每一个字节的细节，因为其中的大部分含义已经在前面介绍过了。

```
system_type_id   name
---------------- ----------------------
34               image
35               text
36               uniqueidentifier
40               date
41               time
42               datetime2
43               datetimeoffset
48               tinyint
52               smallint
56               int
58               smalldatetime
59               real
60               money
61               datetime
62               float
98               sql_variant
99               ntext
104              bit
106              decimal
108              numeric
122              smallmoney
127              bigint
165              varbinary
167              varchar
173              binary
175              char
189              timestamp
231              nvarchar
231              sysname
239              nchar
240              hierarchyid
240              geometry
240              geography
241              xml
```

3 行之间的区别在于起始字节为 13 到 14，这表示第一个可变长度列的结束位置。由于只有一个可变长度列，因此这也是行的长度。*sql_variant* 数据从字节 15 处开始。字节 15 是这种数据类型的代码。可以在 *sys.types* 目录视图的 *system_type_id* 列中找到这些代码。我们在这里重现该视图的相关部分。

在我们的表中，有数据类型 38 hex（即十进制的 56，int 型）、6C hex（即十进制的 108，numeric 类型）、A7 hex（即十进制的 167，*varchar* 类型）和 3D hex（即十进制的 61，*datetime* 类型）。数据类型后的字节是一个表示 *sql_variant* 格式版本的字节，而且在 SQL Server 2008 中该字节的值总是 1。在版本后面可能是下面 4 组字节中的一组。

- 对于 numeric 和 decimal：1 个字节表示精确度，1 个字节表示保留字。
- 对于 string：2 字节表示最大长度，4 字节表示排序规则 ID。
- 对于 *binary* 和 *varbinary*：2 字节用于表示最大长度。
- 对于所有其他类型：没有额外字节。

这些字节之后是 *sql_variant* 列中的实际数据。

```
DATA:

Slot 0, Offset 0x60, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes =  NULL_BITMAP VARIABLE_COLUMNS
Record Size = 21

Memory Dump @0x62B7C060
00000000: 30000800 01000000 02000001 00150038 †0..............8
00000010: 01030000 00†††††††††††††††††††††††††††.....

Slot 1, Offset 0x75, Length 24, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes =  NULL_BITMAP VARIABLE_COLUMNS
Record Size = 24
Memory Dump @0x62B7C075

00000000: 30000800 02000000 02000001 0018006c †0..............l
00000010: 010a0001 005ed0b2 †††††††††††††††††††.....^Ð²

Slot 2, Offset 0x8d, Length 26, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes =  NULL_BITMAP VARIABLE_COLUMNS
Record Size = 26
Memory Dump @0x62B7C08D

00000000: 30000800 03000000 02000001 001a00a7 †0..............§
00000010: 01401f08 d0003461 6263†††††††††††††††.@..Ð.4abc

Slot 3, Offset 0xa7, Length 25, DumpStyle BYTE

Record Type = PRIMARY_RECORD        Record Attributes =  NULL_BITMAP VARIABLE_COLUMNS
Record Size = 25
Memory Dump @0x62B7C0A7

00000000: 30000800 04000000 02000001 0019003d †0..............=
00000010: 0183de14 01299b00 00†††††††††††††††††..Þ..)...

OFFSET TABLE:

Row - Offset
3 (0x3) - 167 (0xa7)
2 (0x2) - 141 (0x8d)
1 (0x1) - 117 (0x75)
0 (0x0) - 96 (0x60)
```

图 5-12 包含 *sql_variant* 数据的行

5.5 约束

约束提供一种有效而简单的方式来增强数据库中数据的完整性。数据完整性有 3 种形式。

实体完整性。保证一个表有一个主键。在 SQL Server 2008 中，通过定义 PRIMARY KEY（主键）、UNIQUE（唯一性）约束或创建唯一索引来保证实体完整性。也可以选择创建一个触发器从而增强实体完整性，但是这种方法通常来说效率较低。

域完整性。保证数据值符合特定的条件。在 SQL Server 2008 中，域完整性可以通过多种方式得以保证。选择适当的数据类型可以保证一个数据值满足特定的条件，例如，数据表示一个有效的日期。其他

方法还包括定义 CHECK（检查）约束、FOREIGN KEY（外键）约束或创建一个触发器。您也可以认为 DEFAULT（默认）约束是强制域完整性的一个方面。

引用完整性。强制两个表（一个是被参照表，一个是参照表）之间的关系。SQL Server 允许您定义 FOREIGN KEY（外键）约束来强制引用完整性，同时您也可以创建触发器实现强制。需要提示的重要一点是引用完整性限制具有两面性。如果被参照的表中的数据被更新或删除，则引用完整性能够保证参照表中参照被修改或被删除的数据也以某种方式得到处理。另一方面，如果数据被更新或插入到参照表中，则引用完整性可以保证新数据与被参照表中的值相匹配。

在这一部分，我们将简要介绍管理约束的一些内在方面。约束也被称为*声明性数据完整性*，因为它们是实际表定义的一部分。这与*程序化数据完整性*（使用存储过程或触发器）相反。

下面是 5 种类型的约束：

- PRIMARY KEY（主键）；
- UNIQUE（唯一）；
- FOREIGN KEY（外键）；
- CHECK（检查）；
- DEFAULT（默认值）。

有时您也可以看到 *IDNETITY* 属性及被描述为约束的某一列的为空性。我们一般不认为这些属性是约束，而是认为它们是某一列的属性，这样做的原因有两个。首先，每一种约束在 *sys.objects* 目录视图中都有自己对应的一行，而 *IDENTITY* 和为空性信息仅在 *sys.objects* 中的 *sys.columns* 和 *sys.identity_columns* 中不可用。这使我们想到这些属性更像数据类型，数据类型也可以利用 *sys.columns* 查看。其次，当您使用 SELECT INTO 命令复制一张表时，所有列名称、数据类型、*IDENTITY* 信息和列为空性也会被复制，但是约束不会被复制到新表中。这使我们想到 *IDENTITY* 和为空性比约束更可能是真实表结构的一部分。

5.5.1　约束名称和目录视图信息

下面这条简单的 CREATE TABLE 语句包括表中的一个主键，用于创建一个具有 PRIMARY KEY 约束的表，同时约束有一个非常晦涩的名称：

```
CREATE TABLE customer
(
cust_id        int IDENTITY NOT NULL PRIMARY KEY,
cust_name      varchar(30)  NOT NULL
);
```

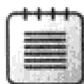
如果没有在定义约束的 *CREATE TABLE* 或 *ALTER TABLE* 语句中提供一个约束名称，那么 SQL Server 会为您提供一个名称。

前面的简单语句中创建的约束有一个与非直观名称 *PK_ _customer_ _3BD0198E35BCFE0A* 非常类似的名称（名称结尾的十六进制数与您创建的 *customer* 表的十六进制数有所不同）。所有类型的单列约束都使用这种命名机制，这里我们对此进行简要介绍。显式命名约束而不是使用系统生成的名称的好处是可以更清楚。约束名称在所有违反约束的错误信息中被使用，因此创建一个 *CUSTOMER_PK* 这样的名称对于用户来说可能比 *PK_ _customer_ _0856260D* 这样的名称更有意义。如果这样的错误信息对用户来说是可见的，那么您应该选择自己的约束名称。前两个字符（*PK*）表示约束类型：*PK* 表示 PRIMARY KEY，*UQ* 表示 UNIQUE，*FK* 表示 FOREIGN KEY（外键），*CK* 表示 CHECK，*DF* 表示 DEFAULT。接下来是

两个下划线，用做分隔符。

提示：

您可能希望使用一个下划线来保留字符同时避免必须这样截断。但是，在一个表名称或一个列名称中（它们都显示在约束名称中），使用下划线是很常见的。可以利用两个下划线来区分名称和分隔符。



注意：

约束名称是架构范围内的，这表示这些名称共享同一个命名空间，因此在同一架构内一定是唯一的。在一个架构内不能有两个约束名称相同的表。

接下来是表名称（*customer*），它的 PRIMARY KEY 约束被限制为 116 个字符，比所有其他约束名称的字符少一些。对于除 PRIMARY KEY 和 UNIQUE 之外的所有约束来说，一系列字符（即列名称）之前还有两个下划线字符用做分隔符。如果必要，可以将列名称截断为 5 个字符。如果列名称不足 5 个字符，则表名称部分的长度可以稍长一些。

最后，另一个分隔符后面是用做约束的对象 ID 十六进制表示。该值用在 *sys.objects* 目录视图的 *object_id* 列中。在 SQL Server 2008 中，对象名称被限定为 128 个字符，因此约束名称所有部分的总长度一定小于或等于 128。

多个目录视图包含约束信息。它们都从 *sys.objects* 视图中继承了列并且包括特定于约束类型的附加列。这些视图是：

- *sys.key_constraints*；
- *sys.check_constraints*；
- *sys.default_constraints*；
- *sys.foreign_keys*。

parent_object_id 列（表示哪个对象包含约束）实际上是基本 *sys.objects* 视图的一部分，但是对于没有“双亲”的对象来说，该列为 0。

5.5.2 视图和多行数据修改中出现的约束故障

应用程序代码中出现的很多故障都是因为开发人员不理解约束故障会如何影响用户声明的多语句事务。最大的误解在于，约束故障等错误会自动终止和回滚整个事务。相反，在出现错误之后，由事务决定是继续并最终提交事务还是回滚。这一特性为开发人员提供了确定处理错误方式的灵活性（语义也与 ANSI SQL-92 标准的 COMMIT 行为相一致）。

由于很多开发人员已经错误地处理了事务错误，同时由于在每条命令之后添加一条错误检查是很麻烦的，因此 SQL Server 包含了一种称为 XACT_ABORT 的 SET 选项，该选项可以使 SQL Server 在事务执行期间遇到错误时终止这项事务。默认设置是 OFF，这与 ANSI 标准行为相一致。

对约束错误和事务的最后一项解释：影响多行的一条数据修改语句（如一条 *UPDATE* 语句）是一项自动的原子操作，即使该语句不是显式事务的一部分。如果这样的一条 *UPDATE* 语句找到 100 行满足 WHERE 子句条件但是其中一行由于违反约束而出现故障，则所有行都不会被更新。我们将在后面的第

10 章介绍显式和隐式事务。

完整性检查的顺序

如果违反任何约束或者如果一个触发器回退该操作，则对某一特定行的修改会失败。只要约束中出现故障，操作就会终止，该行的后续检查不会再被执行，同时也不会对该行触发任何触发器。因此，这些检查的顺序可能非常重要，如下面所列出的那样。

（1）应用适当的默认值。

（2）违反 NOT NULL 限制。

（3）判断 CHECK 约束。

（4）对*引用*表应用 FOREIGN KEY 检查。

（5）对*被引用*表应用 FOREIGN KEY 检查。

（6）检查 UNIQUE 和 PRIMARY KEY 约束的正确性。

（7）触发触发器。

5.6 修改表

SQL Server 2008 允许现有表以多种方式被修改。您可以使用 *ALTER TABLE* 命令对某个现有表进行如下类型的修改。

- 修改某一列的数据类型或 *NULL* 属性。
- 添加一个或多个新列，定义或不定义这些列的约束。
- 添加一条或多条约束。
- 删除一条或多条约束。
- 删除一列或多列。
- 启用或禁用一条或多条约束（仅应用于 CHECK 和 FOREIGN KEY 约束）。
- 启用或禁用一个或多个触发器。
- 重建一张表或一个分区以修改压缩设置或删除碎片（碎片将在第 6 章介绍，压缩将在第 7 章介绍）。
- 修改表的锁升级事件（锁和锁升级将在第 10 章介绍）。

5.6.1 更改数据类型

可以利用 *ALTER TABLE* 的 *ALTER COLUMN* 子句修改数据类型或现有列的 NULL 属性。但是请注意如下的限制。

- 被修改的列不能是 *text*、*image*、*ntext* 或 *rowversion*（*timestamp*）列。
- 如果被修改的列是表的 *ROWGUIDCOL*，则只允许 *DROP ROWGUIDCOL*，不允许对数据类型进行修改。
- 被修改的列不能是计算或复制得到的列。
- 被修改的列不能有 PRIMARY KEY 或 FOREIGN KEY 约束。
- 被修改的列不可以在一个计算得到的列中引用。
- 被修改的列不能将数据类型修改为 *timestamp*。
- 如果被修改的列包含在某个索引中，则唯一允许的类型修改是增加可变长度类型的长度（如将

*varchar(10)*变为 *varchar(20)*）或修改列的为空性。

- 如果被修改的列有 UNIQUE 或 CHECk 约束，则唯一允许的修改是修改可变长度列的长度。对于 UNIQUE 约束来说，新长度必须比旧长度大。
- 如果被修改的列上定义了一个默认值，则允许的唯一修改是增加或减少可变长度类型的长度、修改为空性、修改精确度或保留字。
- 列的原有类型到新类型应该有一种允许的隐式转换。
- 不论当前设置如何，如果合适的话，新类型总有 ANSI_PADDING 语义。
- 如果原有类型到新类型的转换会造成一种溢出（四则运算或大小），那么 *ALTER TABLE* 语句被终止。

下面是使用 *ALTER TABLE* 语句的 *ALTER COLUMN* 子句的语法和一个示例：

```
SYNTAX:

ALTER TABLE table-name ALTER COLUMN column-name
       { type_name [ ( prec [, scale] ) ] [COLLATE <collation name> ]
         [ NULL | NOT NULL ]
         | {ADD | DROP} {ROWGUIDCOL | PERSISTED} }
EXAMPLE:

/* Change the length of the emp_lname column in the employee
   table from varchar(15) to varchar(30) */
ALTER TABLE employee
   ALTER COLUMN emp_name varchar(30);
```

5.6.2 添加一个新列

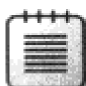
您可以在指定或不指定列级约束的情况下添加一个新列。如果新列不允许 NULL、不是一个标识列并且不是一个 *rowversion*（或 *timestamp*）列，则新列必须有一个已经定义的默认约束（除非表中还没有数据）。SQL Server 用 NULL、合适的标识值、*rowversion*（行版本）值或指定的默认值填充每一行的新列。如果新添加的列可以为空，并且有一个默认约束，则表的现有行不会用默认值填充，而是用 NULL 填充。您可以利用 WITH VALUES 子句覆盖这一约束，从而使表的现有行用指定的默认值填充。

5.6.3 添加、删除、禁用或启用约束

您可以利用 *ALTER TABLE* 语句添加、删除、启用或禁用约束。使用 *ALTER TABLE* 控制约束时，最复杂的地方在于 CHECK 可以有 3 种不同的方式。

- 指定一个 CHECK 约束。
- 延迟新添加的约束的检查。在下面的示例中，我们将添加一个约束来检查 *orders* 表中的 *cust_id* 与 *customer* 中的 *cust_id* 是否匹配，但是我们不希望约束应用到现有数据上。

```
ALTER TABLE orders
  WITH NOCHECK
  ADD FOREIGN KEY (cust_id) REFERENCES customer (cust_id);
```



注意：

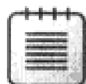
不要使用 WITH NOCHECK，应该使用 WITH CHECK 来强制应用到现有数据上的约束，但是由于这是默认行为，因此不是必需的操作。

- 启用或禁用某一约束。在这个示例中，我们启用 *employee* 表上的所有约束：

```
ALTER TABLE employee
   CHECK CONSTRAINT ALL;
```

可以被禁用的约束类型是 CHECK 约束和 FOREIGN KEY 约束，禁用的意思是告诉 SQL Server 不要检查添加或更新的新数据。在禁用和重新启用约束时需要小心。如果某一约束是表被创建时的一部分，或者是使用 WITH CHECK 选项添加到表中的，则 SQL Server 会知道数据与约束的数据完整性需求相一致。SQL Server 查询优化器接下来可以在某些情况下使用这一知识。例如，如果您需要求 *col1* 的值大于 0 的约束，当应用程序提交一个所有 *col1<0* 的查询时，如果约束始终有效，则优化器会知道没有任何行满足这一查询，同时这一计划是一项非常简单的计划。但是，如果约束已经在没有使用 WITH CHECK 选项的情况下被禁用和重新启用，则不能保证表中的某些数据满足完整性需求。您可能没有任何小于或等于 0 的数据，但是优化器在设计计划时却不知道这一点。优化器所知道的是约束不能被信任。目录视图 *sys.check_constraints* 和 *sys.foreign_keys* 都有一个被称为 *is_not_trusted* 的列。如果您重新启用某一项约束并且不使用 WITH CHECK 选项来通知 SQL Server 使所有现有数据重新生效，则 *is_not_trusted* 列被设置为 1。

虽然不能使用 *ALTER TABLE* 来禁用或启用一个 PRIMARY KEY 或 UNIQUE 约束，但是可以使用 *ALTER INDEX* 命令禁用相关索引。我们将在第 6 章介绍 *ALTER INDEX*。您可以使用 *ALTER TABLE* 删除 PRIMARY KEY 和 UNIQUE 约束，但是需要知道删除这些约束中的一项会自动删除相关索引。实际上删除这些索引的唯一方法是通过修改表来删除约束。



注意：

您不能用 *ALTER TABLE* 修改某一约束的定义。必须使用 *ALTER TABLE* 删除该约束，然后使用 *ALTER TABLE* 添加一个具有新定义的新约束。

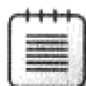
5.6.4 删除列

您可以使用 *ALTER TABLE* 删除表中的一列或多列，但是不能删除如下列。

- 一个被复制的列。
- 在某个索引中使用的某一列。
- 在一个 *CHECK*、*FOREIGN KEY*、*UNIQUE* 或 *PRIMARY KEY* 约束中使用的某一列。
- 与使用 *DEFAULT* 关键字定义或者绑定到默认对象的默认值相关的一列。
- 绑定了某一规则的一列。

您可以利用下面的语法删除某一列：

```
ALTER TABLE table-name
   DROP COLUMN column-name [, next-column-name]...
```



注意：

注意删除某一列和添加新列之间的语法差异：删除一列时需要 *COLUMN* 关键字，但是向表中添加一个新列时则不需要 *COLUMN* 关键字。

5.6.5 启用或禁用一个触发器

利用 *ALTER TABLE* 命令可以启用或删除表上的一个或多个（或所有）触发器。

5.6.6 修改表的内部

注意发布 *ALTER TABLE* 命令时，并不是所有 *ALTER TABLE* 变动都需要 SQL Server 修改每一行。SQL Server 可以按照 3 种基本方式执行 *ALTER TABLE* 命令。

- 仅需要修改元数据。
- 需要检查所有现有数据以保证它们与修改相兼容但是仅需要修改元数据。
- 可能需要从物理上修改每一行。

在很多情况下，SQL Server 只能修改元数据（主要是通过 *sys.columns* 看到的数据）以反映新结构。具体来说，当某一列被删除时、当一个新列被添加并且 NULL 被认为是所有行的新值时、当可变长度列的长度被增加时，或者当不允许为空的列被修改为允许为 NULL 时，数据不会受到影响。删除某一列时数据不会受到影响的意思是列的磁盘空间不会被收回。当表的行大小达到或超过限制时，您可能必须收回被删除列的磁盘空间。您可以通过在表上创建一个聚集索引或者通过 *ALTER INDEX* 重建现有聚集索引的方式收回空间，具体内容将在第 6 章介绍。

对表结构的某些修改要求数据被检查但不被修改。例如，当您将为空性属性修改为不允许 NULL 时，SQL Server 必须首先保证现有行中没有 NULL。当所有现有数据都在新限制范围之内时，某个可变长度列可以被缩短，因此现有数据必须被检查。如果任意行的数据长度比 *ALTER TABLE* 中指定的新限制长度长，则 *ALTER TABLE* 命令执行会失败。因此您要知道对于一个大型表来说，这可能需要花一定的时间。将一个固定长度的列修改为某种更短的类型（如将 *int* 列修改为 *smallint* 或将 *char(10)*修改为 *char(8)*）还需要检查所有数据，以校验所有现有值可以存储在新类型中。但是，即使新数据类型占用更少的字节，物理页上的行也不会被修改。如果您已经创建了一个具有 *int* 列（每一行中需要 4 字节）的表，所有行将使用整个 4 字节。在将表由 *int* 修改为 *smallint* 后，我们会受到可以插入的数据值范围的限制，但是每一行中该列将继续使用 4 字节，而不是 *smallint* 所需要的 2 字节。您可以通过 *DBCC PAGE* 命令进行验证。将 *char(10)*修改为 *char(8)*的操作类似，行继续为该列使用 10 字节，但是新插入的数据只允许 8 字节。直到通过创建或重新创建聚集索引重建一张表时 *char(10)*列才会真正地重建成 *char(8)*。

对表结构的其他修改需要 SQL Server 在物理上修改每一行，同时当 SQL Server 进行修改时，它必须向事务日志写入适当的记录，因此对于大型表来说，这些修改可能是资源密集型的。这种修改类型的一个示例是添加一个不允许为 NULL 的新列，此时您必须指定一个默认的列值。SQL Server 在物理上为每一行添加具有默认值的列。注意添加一个允许 NULL 的新列时，修改只能是一种元数据操作。

修改表时出现的另一种负面影响是当某一列被修改为增加长度时。此时，原有列不会被真正替换，而是将新列添加到该表上，同时 *DBCC PAGE* 显示为原有数据仍然存在那里。现在看一下这种情况下的页转储，我们可以通过查看列偏移（利用我们在本章前面介绍的列详细信息查询）看到其中一部分不可预期的操作。

首先创建所有列都是固定长度的一张表，包括第一个位置的 *smallint*：

```
CREATE TABLE change
(col1 smallint, col2 char(10), col3 char(5));
```

现在我们来看一下列偏移：

```
SELECT c.name AS column_name, column_id, max_inrow_length, pc.system_type_id, leaf_offset
 FROM sys.system_internals_partition_columns pc
    JOIN sys.partitions p
      ON p.partition_id = pc.partition_id
    JOIN sys.columns c
         ON column_id = partition_column_id
            AND c.object_id = p.object_id
WHERE p.object_id=object_id('change');
```

```
RESULTS:
column_name      column_id    max_inrow_length      system_type_id      leaf_offset
-------------    -----------  ------------------    --------------      -----------
col1             1            2                     52          4
col2             2            10                    175         6
col3             3            5                     175         16
```

现在将 *smallint* 修改为 *int*：

```
ALTER TABLE change
   ALTER COLUMN col1 int;
```

最后，再次运行列详细信息查询，看到 *col1* 的起始位置现在在行的更后面，而且在行标题信息后面偏移量 4 处没有任何行。*ALTER TABLE* 对新列的创建甚至会在所有数据插入表之前发生：

```
column_name      column_id    max_inrow_length      system_type_id      leaf_offset
-------------    -----------  ------------------    ----------------    -----------
col1             1            4                     56                  21
col2             2            10                    175                 6
col3             3            5                     175                 16
```

SQL Server 中不真正删除原有列的行为的另一个缺点是我们现在更多地受限于行的大小。行大小现在包括不再可用和可见（除非使用 *DBCC PAGE*）的旧列。例如，在创建一个具有几个大型固定长度的字符列时，如下所示，我们接下来可以利用 *ALTER* 将 *char(2000)*列修改为 *char(3000)*：

```
CREATE TABLE bigchange
(col1 smallint, col2 char(2000), col3 char(1000));

ALTER TABLE bigchange
   ALTER COLUMN col2 char(3000);
```

在这里，行的长度应该刚好超过 4000 字节，因为有一个 3000 字节的列、一个 1000 字节的列和一个 *smallint*。但是，如果试图再添加另一个长度为 3000 字节的列，则会出现错误：

```
ALTER TABLE bigchange
   ADD col4 char(3000);

Msg 1701, Level 16, State 1, Line 1
Creating or altering table 'bigchange' failed because the minimum row size would be 9009,
including 7 bytes of internal overhead. This exceeds the maximum allowable table row size
of8060 bytes.
```

但是，如果只创建一个有两个 3000 字节的列和一个 1000 字节的列的表时，则不会出现任何问题：

```
CREATE TABLE nochange
(col1 smallint, col2 char(3000), col3 char(1000), col4 char(3000));
```

注意，没有办法修改表的逻辑列的顺序或在表中的特殊位置添加一个新列。新添加的列总是获得接下来最高的 *column_id* 值。当您在某张表上执行 *SELECT* *或者利用 *sp_help* 查看元数据时，列总是按照 *column_id* 的顺序返回。如果需要一种不同的顺序，可以利用如下几个选项。

- 不要使用 *SELECT* *，总是按照希望列出现的顺序在 *SELECT* 后面书写列的列表。
- 按照希望列出现的顺序创建表的一个视图，接下来可以使用 *SELECT * from* 视图或在视图上运行 *sp_help*。
- 创建一张新表，然后将旧表中的数据复制到新表中，删除旧表，最后将新表重命名为旧表的名称。不要忘记重新创建所有约束、索引和触发器。

您可能认为，Management Studio 可以在某一特殊位置添加一个新列或重新排列列的顺序，但事实并非如此。在后台上，该工具实际上是利用前面的 3 个选项并利用新索引、约束和触发器创建一张全新的表。如果您想知道为什么仅向一个现有（大型）表添加一个新列会花费很长的时间，原因可能就在于此。

5.7 堆修改内部

我们已经看到 SQL Server 是如何在堆中存储数据的。现在来看一下当堆中数据被修改时，SQL Server 在内部实际做了哪些操作。修改索引中的数据（包括具有聚集索引的表）是完全不同的话题，我们将在第 6 章进行详细介绍。一般来说，表上应该始终有一个聚集索引。有些情况下使用堆可能更好，如最重要的因素是 *INSERT* 操作的速度时，但直到执行完整测试以确定具有其中的一种情况时，最好有一个聚集索引，而不是使数据一点结构都没有。在第 6 章中，您将看到聚集索引和非聚集索引的优点和问题，同时还会检查使用它们的一些准则。现在我们只关注 SQL Server 是如何在没有聚集索引的表上处理数据修改的。

5.7.1 分配结构

正如在第 3 章介绍的那样，SQL Server 为每个对象分配一个或多个 IAM 页面，用于跟踪每个文件中哪些范围属于该对象。如果表是一个堆，则 IAM 是 SQL Server 查找属于该表的所有范围的唯一方法，因为表的单个数据页不是在一种双链接列表中连接的，这是表有聚集索引时它们存在的方式。每一级索引的页面都是链接的，同时由于数据被认为是聚集索引的页级，因此 SQL Server 保持这种链接。但是，对于一个堆来说，没有这样的链接表连接页面。SQL Server 确定哪一页属于表的唯一方法是通过检查表的 IAM。

另一种特殊的分配结构在 SQL Server 执行数据修改操作时尤其有用，这就是页面可用空间（Page Free Space，PFS）结构。PFS 页跟踪每一个页面上有多少空间，这样在堆中执行 *INSERT* 操作就可以知道哪些地方可以用来存储新数据，执行 *UPDATE* 操作可以知道哪一行可以被删除。我们在第 3 章曾经简要提到了 PFS 页面，那时我们说这些页面为一个文件中的 8088 个页面中的每一个页面保留 1 字节。这比全局分配地图（GAM）、共享全局定位地图（SGAM）和 IAM 要稀疏得多，它每个范围内只包含一位。图 5-13 显示了一个 PFS 页面上一个字节的结构。只有最后 3 位用于表示页面的完整性，其他 5 位中的 4 位都有自己的含义。

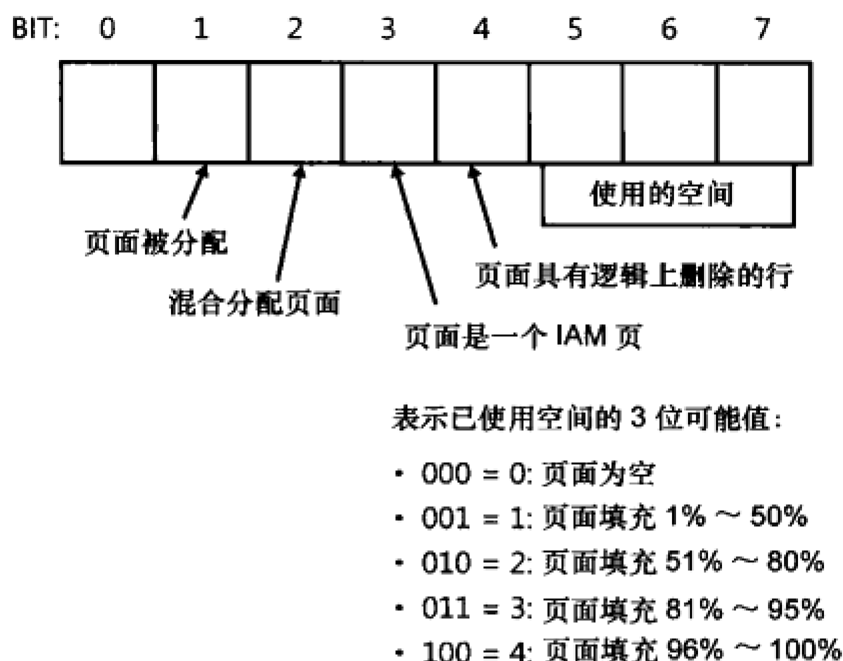


图 5-13 一个 PFS 字节中位的含义

下面是解释这些位的方式。

- **位 1**。该位表示页面实际上是否被分配。例如，可以为某个对象分配一个一致扩展，但是并不是该扩展内的所有页面都可以被分配。为了说明一定分配扩展内的哪些页面可以实际被使用，SQL Server 需要查看 PFS 页面中合适字节上的这一位。
- **位 2**。表示相应页面是否来自一个混合扩展。
- **位 3**。表示该页面是一个 IAM 页面。记住 IAM 页面不是某个文件的已知位置。
- **位 4**。表示该页包含备份记录。正如我们将看到的那样，SQL Server 使用一个背景清除线程来删除备份记录，PFS 页面上的这些位可以帮助 SQL Server 查找需要被清除的页面（备份记录只在索引中或者使用行级版本控制时出现，因此在本章不做进一步的讨论）。
- **位 5~7**。作为一个 3 位值，值 0~4 表示页面的完整性，具体如下。

  0：页面为空。

  1：页面填充 1%~50%。

  2：页面填充 51%~80%

  3：页面填充 81%~95%。

  4：页面填充 96%~100%。

PFS 页面位于每个数据文件的已知位置。文件中的第 2 个页面（页面 1）是一个 PFS 页面，此后每隔 8088 页面就是一个 PFS 页。

5.7.2 插入行

向表中插入一个新行时，SQL Server 必须确定新行的存放位置。当一张表没有聚集索引时（即该表是一个堆时），新行总是被插入到表中任意可用的位置。我们已经介绍过 IAM 和 PFS 页面如何跟踪一个表中的哪些扩展属于一张表，以及这些扩展中的哪些页面还有空间可用。即使没有聚集索引，空间管理也是很高效的。如果页面上没有空间可用，则 SQL Server 会试着从已经属于该对象的现有一致扩展中查找未分配的页面。如果没有，则 SQL Server 必须为表分配一个新的扩展。第 3 章介绍了如何使用 GAM 和 SGAM 来查找可以分配给某个对象的扩展。

5.7.3 删除行

从表中删除行时，必须考虑到数据页和索引页发生的情况。请记住数据实际上是一个聚集索引的页级，从一张具有聚集索引的表上删除行与从非聚集索引的页级上删除行的方式是相同的。从一个堆中删除行的方式则不同，好像从一个索引的节点页删除行。

1. 从堆中删除行

当一行被删除时，SQL Server 2008 不会自动重新组织页面上的空间。作为一种性能优化，只有在某一页面需要额外连续空间来插入新行时才执行压缩。您可以在下面的示例中看到这一点，下面的示例从页面中间删除一行，然后利用 *DBCC PAGE* 检查该页：

```
USE testdb;
GO

CREATE TABLE smallrows
(
    a int identity,
    b char(10)
);
GO

INSERT INTO smallrows
    VALUES ('row 1');
INSERT INTO smallrows
    VALUES ('row 2');
INSERT INTO smallrows
    VALUES ('row 3');
INSERT INTO smallrows
    VALUES ('row 4');
INSERT INTO smallrows
    VALUES ('row 5');
GO

DBCC IND (testdb, smallrows, -1);
-- Note the FileID and PageID from the row where PageType = 1
--  and use those values with DBCC PAGE (I got FileID 1 and PageID 4536)

DBCC TRACEON(3604);
GO
DBCC PAGE(testdb, 1, 4536,1);
```

下面是 *DBCC PAGE* 的输出结果：

```
DATA:

Slot 0, Offset 0x60, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes = NULL_BITMAP
Memory Dump @0x61D9C060
00000000: 10001200 01000000 726f7720 31202020 †........row 1
00000010: 20200200 fc††††††††††††††††††††††††††† ...
```

```
Slot 1, Offset 0x75, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD       Record Attributes = NULL_BITMAP
Memory Dump @0x61D9C075
00000000: 10001200 02000000 726f7720 32202020 †........row 2
00000010: 20200200 fc†††††††††††††††††††††††††††† ...

Slot 2, Offset 0x8a, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD Record Attributes = NULL_BITMAP
Memory Dump @0x61D9C08A
00000000: 10001200 03000000 726f7720 33202020 †........row 3
00000010: 20200200 fc†††††††††††††††††††††††††††† ...

Slot 3, Offset 0x9f, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD       Record Attributes = NULL_BITMAP
Memory Dump @0x61D9C09F
00000000: 10001200 04000000 726f7720 34202020 †........row 4
00000010: 20200200 fc†††††††††††††††††††††††††††† ...

Slot 4, Offset 0xb4, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD       Record Attributes = NULL_BITMAP
Memory Dump @0x61D9C0B4
00000000: 10001200 05000000 726f7720 35202020 †........row 5
00000010: 20200200 fc†††††††††††††††††††††††††††† ...

OFFSET TABLE:
Row - Offset
4 (0x4) - 180 (0xb4)
3 (0x3) - 159 (0x9f)
2 (0x2) - 138 (0x8a)
1 (0x1) - 117 (0x75)
0 (0x0) - 96 (0x60)
```

现在我们将删除中间行（WHERE a=3）并再次查看该页面：

```
DELETE FROM smallrows
WHERE a = 3;
GO

DBCC PAGE(testdb, 1, 4536,1);
GO
```

下面是第二次执行 *DBCC PAGE* 得到的结果：

```
DATA:
Slot 0, Offset 0x60, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD       Record Attributes = NULL_BITMAP
Memory Dump @0x61B6C060
00000000: 10001200 01000000 726f7720 31202020 †........row 1
00000010: 20200200 fc†††††††††††††††††††††††††††† ...

Slot 1, Offset 0x75, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD       Record Attributes = NULL_BITMAP
Memory Dump @0x61B6C075
00000000: 10001200 02000000 726f7720 32202020 †........row 2
00000010: 20200200 fc†††††††††††††††††††††††††††† ...
```

```
Slot 3, Offset 0x9f, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes = NULL_BITMAP
Memory Dump @0x61B6C09F
00000000: 10001200 04000000 726f7720 34202020 †........row 4
00000010: 20200200 fc†††††††††††††††††††††††††† ...

Slot 4, Offset 0xb4, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes = NULL_BITMAP
Memory Dump @0x61B6C0B4
00000000: 10001200 05000000 726f7720 35202020 †........row 5
00000010: 20200200 fc†††††††††††††††††††††††††† ...

OFFSET TABLE:
Row - Offset
4 (0x4) - 180 (0xb4)
3 (0x3) - 159 (0x9f)
2 (0x2) - 0 (0x0)
1 (0x1) - 117 (0x75)
0 (0x0) - 96 (0x60)
```

注意在堆中，页面尾部的行偏移阵列表示第 3 行（槽 2）的偏移量是 2（这表示没有任何行使用槽 2），同时使用槽 3 的行的偏移量与删除前相同。当发生 *DELETE* 时，页面上没有任何数据被删除。使用 *printopt* 1 或 *printopt* 3 作为 *DBCC PAGE* 的选项时，该行不会在该页面上显示。但是，如果利用 *printopt 2* 转储这一页，则还会看到“行 3”的字节。它们没有从页面上物理删除，行偏移阵列中的 0 表示空间现在没有被使用而且不能被新行使用。

除了页面上没有被收回的空间之外，堆中的空页面也经常不能被收回。即使您不从堆中删除所有行，SQL Server 也不会将空页面标记为未分配，因此空间不能被其他对象使用。动态管理视图（DMV）*sys.dm_db_partition_stats* 仍然显示该控件属于该堆表。避免这一问题的一种方法是当执行删除操作时请求一次表锁定，我们将在第 19 章介绍锁的注意事项。如果这一问题已经出现，而且属于某张表的空间比该表实际拥有的空间更多时，可以在表上建立一个聚集索引来重新组织空间，然后再将索引删除。

2. 回收页面

当数据页的最后一行被删除时，整个页面将被释放。除非该表是一个堆，正如我们前面介绍的那样（如果该页是表中剩余的唯一一页，则不会被释放。一张表至少包含一个页面，即使该表是空的）。数据页释放的结果是：指向被释放数据页的索引页中的行被删除。如果一个索引行被删除（这可能会再次作为删除/插入更新策略的一部分出现），那么索引页会被释放，从而使索引页中只有一项。该项被移动到它相邻的页面，然后空页面被释放。

到目前为止，我们所讨论的都是删除一行所必需的页面操作。如果在一次 *DELETE* 操作中删除了多行，则必须注意一些其他问题。

5.7.4 更新行

SQL Server 可以以多种不同的方式更新行，自动和不可见地为特殊操作选择最快的更新策略。在确定策略时，SQL Server 会估算受影响的行数、行被访问的方式（通过扫描或索引检索，以及通过哪个索引）及是否会出现对索引键的修改。更新可以在适当的位置进行（通过将原始行中某一列的值修改为一个新值）或者作为插入后的一种删除。此外，更新可以通过查询处理程序或存储引擎进行管理。在这一

部分，我们将只检查更新是在适当的位置发生还是 SQL Server 将其作为两次独立的操作：删除旧行并插入一个新行。

1. 移动行

如果一行必须移动到表中的一个新位置会出现什么情况呢？在 SQL Server 2008 中，这种情况出现的原因有很多。在第 6 章中，我们将查看索引的结构并发现表的聚集索引的值决定行的位置。因此，如果聚集键的值被修改，则行很可能要在表中移动。

如果行仍然具有相同的行定位器（换言之，行的聚集键保持不变），则所有非聚集索引不必被修改。如果一张表没有聚集索引（换言之，如果表是一个堆），则行可以移动，因为行不再匹配原始页面。当具有可变长度列的一行被更新为一个新的更大容量的行时会出现行移动的情况，这样行不再匹配原始位置。正如我们在第 6 章介绍索引结构时所说的那样，堆上的非聚集索引包含指向行的实际物理位置的数据行指针，包括文件编号、页编号和行编号。为了使非聚集索引不必仅由一行移动到一个不同的物理位置上而必须被全部更新，SQL Server 会在行必须移动时在原始位置留下一个转发指针。

让我们来看一个关于转发指针的例子。我们将创建一张与我们在前面执行 *DELETE* 操作时创建的表非常相似的表，但是该表三分之一的列都是可变长度列。当我们为表填充 5 行数据（填充页面）之后，我们更新其中的一行，使第 3 列更长。该行不再适合原始页面，必须移动。使用 *DBCC IND* 获得表使用的页编号，具体语句如下：

```
USE testdb;
GO
DROP TABLE bigrows;
GO
CREATE TABLE bigrows
(   a int IDENTITY ,
    b varchar(1600),
    c varchar(1600));
GO
INSERT INTO bigrows
    VALUES (REPLICATE('a', 1600), '');
INSERT INTO bigrows
    VALUES (REPLICATE('b', 1600), '');
INSERT INTO bigrows
    VALUES (REPLICATE('c', 1600), '');
INSERT INTO bigrows
    VALUES (REPLICATE('d', 1600), '');
INSERT INTO bigrows
    VALUES (REPLICATE('e', 1600), '');
GO
UPDATE bigrows
SET c = REPLICATE('x', 1600)
WHERE a = 3;
GO

DBCC IND (testdb, bigrows, -1);
DBCC IND (testdb, bigrows, -1);
-- Note the FileID and PageID from the rows where PageType = 1
--  and use those values with DBCC PAGE (I got FileID 1 and
--   PageID values of 2252 and 4586.
```

```
RESULTS:
PageFID PagePID
------- -----------
1       2252
1       4586

DBCC TRACEON(3604);
GO
DBCC PAGE(testdb, 1, 2252, 1);
GO
```

这里不显示 *DBCC PAGE* 命令的整个输出结果，而是显示前面显示的 a=3 这一行的槽中显示的内容：

```
Slot 2, Offset 0x1feb, Length 9, DumpStyle BYTE
Record Type = FORWARDING_STUB       Record Attributes =
Memory Dump @0x61ADDFEB
00000000: 04ea1100 00010000 00†††††††††††††††††.........
```

第一个字节中的值 4 表示这只是一个转发存根。接下来的 3 个字节 0011ea 是被移动的行的页编号。由于这是一个十六进制值，因此需要将其转换成十进制的 4586。下一组的 4 个字节告诉我们该页位于槽 0 文件 1 中。如果接下来利用 *DBCC PAGE* 查看该页面（页面 4586），则可以看到转发记录的内容，同时您可以看到 Record Type（记录类型）表示 FORWARDED_RECORD。

2. 管理转发指针

转发指针允许您修改堆中的数据，不必担心必须对非聚集索引进行重大修改。如果已经被转发的某一行必须再次移动，那么原始的转发指针会指向新的位置。您永远不会使一个转发指针指向另一个转发指针。此外，如果被转发的行缩短到能够适应其原始位置，则该记录可以移回到其原始位置（如果该页面上还有空间），同时转发指针会被删除。

SQL Server 的未来版本可能包括对堆中数据执行物理重组的某种机制，这样可以删除转发指针。注意转发指针只存在于堆中，并且重新组织表的 *ALTER TABLE* 选项不会对堆进行任何改变。您可以重组堆中的非聚集索引而不是表本身。目前，当一个转发指针被创建时，它会永远保留在那里——几种特殊情况除外。第一种特例我们已经提到过，即一行缩短并返回其原始位置。第二种特例是整个数据库缩短。当一个文件被缩短时，书签实际上会被重新分配。收缩过程永远不会产生转发指针。对于由于收缩过程而被删除的页面来说，它们包含的任何指针或存根都会被有效地“转发”。转发指针被删除的其他情况是明显的示例：如果被转发的行被删除，或者在表上创建一个聚集索引从而使其不再是一个堆。

更多信息：

为了获得表中转发记录的数量，可以查看 *sys.dm_db_ index_physical_stats* 函数的输出结果。我们将在第 6 章对此内容进行介绍。

3. 在位更新

在 SQL Server 2008 中，在位更新某一行是规则而不是特例。这表示行仍然保持在同一页的同一位置并且只有受影响的字节被修改。此外，每条被更新的记录在日志中包含一条记录，除非表有一个更新触

发器或者标记为复制。在这些情况下，更新仍然是在位更新，但是日志包含一条删除记录，接着是一条插入记录。

在某一行不能进行在位更新的情况下，由于非聚集索引存储方式及转发指针的使用（我们在前面已经介绍过），非在位更新的花费是很小的。实际上您可以对原始页面上的行进行一种非在位更新。如果正在更新一个堆（并且不需要转发指针）或者一个具有聚集索引的表在不改变聚集键的情况下被更新，则发生的是在位更新。如果聚集键改变但是行根本不需要移动，那么也可以实现在位更新。例如，如果某一行名字列上有一个聚集索引包含连续键值 *Able*、*Becker* 和 *Charlie*，您可能希望将 *Becker* 更新为 *Baker*。由于行保持在相同的位置，即使在聚集索引键值被修改之后也是如此，因此 SQL Server 执行在位更新。另一方面，如果将 *Able* 更新为 *Buchner*，那么这种更新就不是在位更新，但新行可能还是保持在相同的页面上不变。

4. 非在位更新

如果由于正在更新聚集键而造成不能进行在位更新，则更新表现为在一条插入语句后跟随一条删除语句。在某些情况下，您将实现一种混合更新：某些行是在位更新，某些行是非在位更新。如果您正在更新索引键，则 SQL Server 会构建一个 *DELETE* 和 *INSERT* 操作需要修改的所有行的列表。如果列表非常小，则会存储在内存中，必要时会写入 *tempdb* 数据库。该列表接下来会按照键值和操作符（*DELETE* 或 *INSERT*）进行排序。如果被修改的索引键值不唯一，则 *DELETE* 和 *INSERT* 步骤会被应用到表上。如果索引是唯一的，则会执行一个额外步骤将相同键值上的 *DELETE* 和 *INSERT* 操作变成一条更新操作。

更多信息：

查询优化器确定这种特殊的 *UPDATE* 方法是否合适，被种称为 *Split/Sort/Collapse* 的内部优化将在第 8 章详细介绍。

5.8 小结

表通常是关系数据库的核心，对于 SQL Server 来说尤为如此。在这一章中，我们介绍了各种数据类型的内部存储问题，对固定长度数据类型和可变长度数据类型进行了对比。我们发现，SQL Server 2008 提供了存储可变长度数据的多种选项，包括太长而不适合在一个数据页上存储的数据，同时您还看到不能简单地认为使用可变长度数据类型总是好的或者总是不好的。SQL Server 提供了用户定义数据类型支持域的操作，同时还提供了 *IDENTITY* 属性来使一列生成自动连续的数值。还看到了数据是如何物理地存储在数据页上的，同时我们查询了一些提供基本（不可访问）系统表信息的元数据视图。SQL Server 还提供了约束，从而为保证数据的逻辑完整性提供了一种有效的方式。

第 6 章
索引：内部和管理

Kalen Delaney、*Kimberly L. Tripp* 和 *Paul S. Randal*

Microsoft SQL Server 没有使其运行速度更快的配置选项或开关，也没有魔术棒。但是，如果适当地创建和设计索引，则可以使索引发挥魔术棒的作用。适当的索引（为适当的查询而创建的）可以使查询执行时间从几小时减少到几秒钟。没有其他方法可以达到这种效果，添加硬件或修改配置选项通常只能获得有限的改善。为什么索引能够使一个查询请求的执行时间从成千上万次的 I/O 操作减少到几次呢？任何索引都能提高性能吗？遗憾的是，良好的性能不是自然而然发生的。不是所有的索引都是等价的，也不是任何索引都能提高性能。实际上，过多的索引通常比索引不足的效果还差。您不能指望通过对每一列建立索引来提高 SQL Server 的性能。

那么您怎么知道如何创建最合适的索引呢？坦白地说，这由多个部分组成：了解您的数据，了解您的工作负载，同时还要了解 SQL Server 的工作方式。就 SQL Server 的工作方式而言，有多个组成部分，分别是索引内部、统计信息、查询优化和维护。本章主要介绍索引内部和维护，展开讨论这些话题，使您获得最好的创建实践及最佳的基本索引策略。了解 SQL Server 存储索引的物理方式及存储引擎访问和操纵这些物理结构的方式，就知道如何根据工作负载创建*正确的*索引了。此外，由于您可以设想 SQL Server 可以选择的选项及为什么对于某些请求来说有些结构比其他结构更有效，因此这些信息可以帮助您为第 8 章的的学习做准备。

本章分成多个部分。第一部分介绍索引的作用和概念及内部。在这一部分，学习索引如何被存储及索引如何对数据检索起作用。第二部分介绍数据被修改时发生的情况——如何发生及 SQL Server 是如何保证一致性的。在这一部分，您还将了解在索引上进行数据修改的潜在影响，如碎片等。最后，第三部分介绍索引管理和维护。

6.1 概述

想一想日常生活中我们每天都会接触到的索引例子：图书及其他文档中的索引。假设您正在试图利用 *CREATE INDEX* 语句在 SQL Server 中创建索引，同时正在使用两个 SQL Server 引用来查明如何编写该语句。假设一种引用是《*Microsoft SQL Server Transact-SQL Language Reference Manual*》，即我们所说的“T-SQL 引用”。假设该书只是所有 SQL Server 关键字、命令、过程和函数的一个按字母顺序排列的列表。另一种引用是本书。您可以在这两本引用资料中迅速地找到关于索引的信息，虽然这两本书是按照不同的方式组织的。

在 T-SQL 引用中，所有命令和关键字都是按照字母表顺序组织的。您会发现 *CREATE INDEX* 位于所有其他 *CREATE* 语句的前面，因此您不必去看书上的其他大部分内容。这样就可以快速查看几个页面并找到 *CREATE DATABASE* 所在的页面，之后您会发现 *CREATE INDEX*。现在，如果继续向前翻阅到 *CREATE VIEW* 时还没有看到 *CREATE INDEX*，就说明书中漏掉了 *CREATE INDEX* 的内容，因为命令和关键字是

按照字母顺序组织的（当然，这只是一个例子，*CREATE INDEX* 肯定是在 T-SQL 引用中存在的）。

接下来，您可以试着在本书中查找 *CREATE INDEX*。该书不是按照字母表顺序组织命令和关键字的，而是在书的后面有一个目录，所有目录项都是按照字母顺序组织的。因此，您可以根据 *CREATE INDEX* 位于字母表前面这一常识迅速找到该目录的位置。但是，与 T-SQL 引用不同的是，找到关键字 *CREATE INDEX* 后，您不会看到良好且有条理的示例。索引仅为您提供指针，告诉您内容位于哪个页面上。事实上可以在书中列出很多页面。如果在书的目录中查找 *CREATE TABLE*，会发现列出了很多页面。最后，如果查找存储过程 *sp_addumpdevice*（一条彻底被废弃的命令），则根本不会在目录中找到，因为书中没有对这一内容做介绍。

这两种搜索与使用聚集索引（书的内容按序排列）和非聚集索引（按照书中的目录查找）类似。如果一个表是聚集的，则表数据逻辑上按照聚集键值顺序存储，如同 T-SQL 引用依次排列所有主题。当您找到正在查找的内容时，搜索就会结束。在非聚集索引中，索引与数据本身是完全独立的结构。当您在索引中找到正在查找的内容时，必须按照一定的参照指针获取实际数据。虽然 SQL Server 中的非聚集索引与书后面的目录非常类似，但并不是完全相同的。

6.1.1 SQL Server 索引 B 树

在 SQL Server 中，索引是按照 B 树结构组织的，如图 6-1 所示。B 树是指“平衡树”，SQL Server 使用被称为 *B+树*（读作 B 加树）的一种特殊 B 树，*这种树不是在所有时候所有情况下都保持严格平衡*。与标准树不同的是，*B 树*总是倒着的，树根（一个页面）位于顶部，叶级位于底部。中间级别是由多种因素决定的。B 树是很多人以不同方式过多使用的一个术语，或者表示整个索引结构，或者只表示非叶级结构。在本书中，B 树代表整个索引结构。

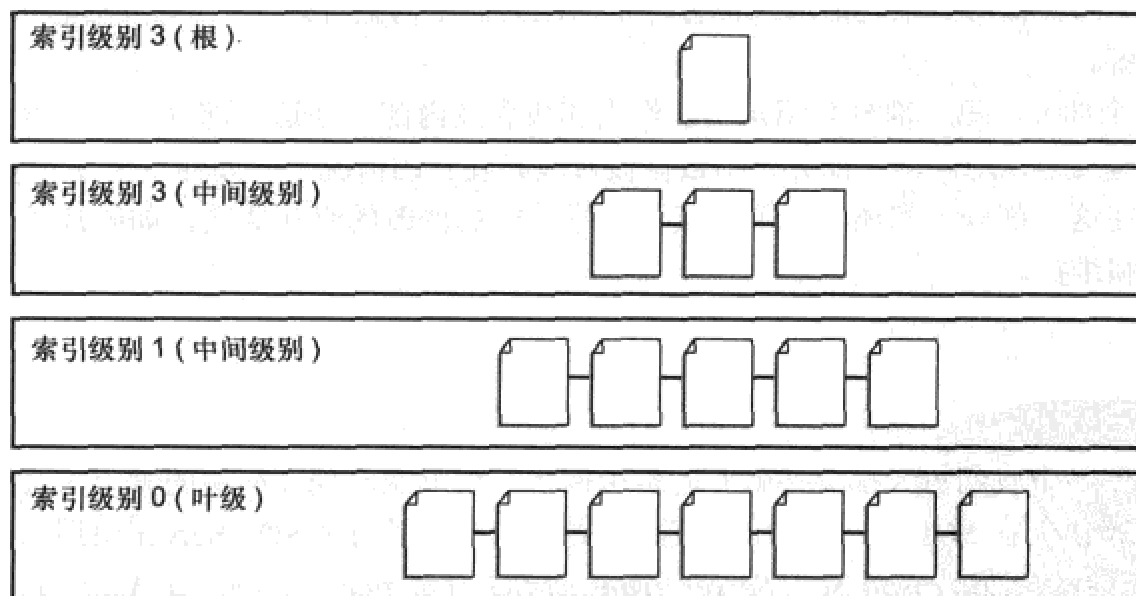


图 6-1 SQL Server 索引的一棵 B 树

SQL Server 中 B 树吸引人的地方在于它们的构建方式及每一级包含的内容。从结构上来说，索引可能根据是否使用多个 CPU 创建或重建索引而有所不同（我们将在本章后面的“MAXDOP”一节对这一内容做详细解释），但是大多数情况下，树的大小和宽度是由表中索引的定义及行的数量和大小决定的。为了说明这一点，我们给出了一些常用术语和定义的示例。首先，索引有两个基本组成部分，一个叶级和一个或多个非叶级。了解和讨论非叶级是很有用的，但是简单地说，非叶级就是用于导航的（大部分是

用于导航到叶级）。但是，第一个中间级也用于碎片分析，并且在大范围扫描索引时驱动预读。

为了理解这些结构，我们从定义一般术语（意思是这些基本概念同时应用到聚集和非聚集索引）中的叶级开始。索引的叶级按照被索引顺序包含表中每行数据的一些内容（我们将在本章后面介绍物理索引结构时具体讨论）。这里我们只关注传统索引及创建时没有使用筛选器的索引（指 SQL Server 2008 中被称为*筛选索引*的新功能）。

非叶级用于帮助导航到叶级结构非常简单的某一行上。每个非叶级都为该级别下面的每个页面存储一些内容，级别不断增加，直到索引建立到页面的根。索引中每个更高的非叶级小于它下面的非叶级，因为该级别上的每一行只包含该级别下面每个页面上可以包含的最小键值，以及该页面的一个指针。虽然这可能会产生很多级（即一棵高树），但是 SQL Server 中键大小（最多 900 字节或 16 列——哪一种都行）的限制帮助使索引树保持相对很小。实际上在我们接下来要介绍的示例中（有一个行非常宽的索引和一个最大大小的键定义），该示例索引（索引被创建时）的树深度只有 8 层。

为了查看这棵树（通过计算确定其大小），我们使用一个索引叶级包含 1 000 000“行”的示例。之所以对行加引号是因为它们不一定是数据行，它们只是*任何*索引的叶级行。在本章后面（在我们讨论每种特殊索引的物理结构时），您会确切地看到什么是叶级行及它们是如何组织的。对于这个示例来说，我们所关注的是一个抽象的“索引”，我们只关注叶级和非叶级，以及它们在 SQL Server 页面的限制（8KB 页面）范围内是如何组织的。在这个示例中，叶级行是 4000 字节，这表示我们只能在每个页面上存储两行。对于一个有 1 000 000 行的表来说，索引的叶级有 500 000 个页面。相对来说，这是一个非常宽的行结构。然而，我们没有在页面上浪费太多的空间。如果叶级页面有两个 3000 字节行，仍然只能在每个页面上放两行，但是这样就会有 2000 字节的空间被浪费（这将是内部碎片的一个示例，我们将在本章后面的“碎片”一节进行介绍）。

现在我们来看为什么这些只是“行”而不是具体的数据行呢？原因在于该叶级可能是某个聚集索引的叶级（因而是数据行），或者这些叶级行可能是某个非聚集索引中的行，该非聚集索引使用 *INCLUDE*（在 SQL Server 2005 中引入的）向索引的叶级添加非键列。当使用 *INCLUDE* 时，叶级页面可以包含更宽的行（比 900 字节或 16 列键的最大值还要宽）。同样，现在听起来似乎没什么用，但我们将在本章后面介绍为什么这是非常有意义的。在这个示例中，该索引的叶级在创建时是 4GB（500 000 个 8KB 页面）。这种结构可能变得更大（由它的定义决定），也可能非常零碎（如果很多数据被添加）。但是（还是由它的定义决定），我们有办法控制数据经常变动时该索引变成碎片的程度（我们将在本章后面的多个小节中进一步介绍这一主题）。此时，由于“行”宽度，索引的叶级很大。同时，使用 900 字节的最大值表示您之后可以在每个非叶级页面上放 8 行数据（每页 8096 字节，每行 900 字节）。但是，使用这个最大值，结果树（达到页面的根）会比较小，而且只有 8 层（如下面所示）。实际上，提高可伸缩性是将索引键值限定为 900 字节或 16 列（无论哪一种情况）的主要原因。

- 非叶级的根页（级别 7）=2 行=1 页（每页 8 行）。
- 中间非叶级（级别 6）=16 行=2 页（每页 8 行）。
- 中间非叶级（级别 5）=123 行=16 页（每页 8 行）。
- 中间非叶级（级别 4）=977 行=123 页（每页 8 行）。
- 中间非叶级（级别 3）=7 813 行=977 页（每页 8 行）。
- 中间非叶级（级别 2）=62 500 行=7 813 页（每页 8 行）。
- 中间非叶级（级别 1）=500 000 行=62 500 页（每页 8 行）。
- 叶级（级别 0）=1 000 000 行=500 000 页（每页 2 行）。

具有更小键大小的索引的伸缩更快。假设有如前所示的相同叶级页面（每页 2 行，共 1 000 000 行），但索引键更小了，因此只有 20 字节的非叶级（包括一些额外开销的空间）中有一个更小的行大小，您可以在每个非叶级页面上存放 404 行。

- 非叶级的根页（级别 3）=4 行=1 页（每页 404 行）。
- 中间非叶级（级别 2）=1 238 行=4 页（每页 404 行）。
- 中间非叶级（级别 1）=500 000 行=1238 页（每页 404 行）。
- 页级（级别 0）=1 000 000 行=500 000 页（每页 2 行）。

在第二个示例中，原始索引不仅只有 4 个级别，而且在请求另一个级别之前，它还可以添加 120 878 528 行（行的最大可能数量是 404*404*404*2（即 131 878 528）减去已经存在的行数（1 000 000））。可以这样理解，根页当前允许放 404 个条目；但是，我们现在只存储了 4 个（现有的非叶级没有完全填充满）。这只是一种理论上的最大值，在没有其他因素的情况下（如碎片），一个 4 级树将能够搜索一个拥有超过 1.31 亿行记录的表（同样，使用这一很小的索引键大小）。这表示查找使用此树导航至对应行的索引只需要 4 次 I/O 操作。同时由于树是平衡的，因此查找任何记录都需要相同的资源数。检索速度不变，因为索引具有相同的深度遍历。索引可以是零碎的，页面可以更稀疏，但是这些树不会失去平衡。我们将在本章后面介绍索引维护时对这一内容进行介绍。

记住用于显示这些示例的所有数学函数并不重要，但是理解索引（尤其是具有合理键的）可伸缩性的真谛意味着能够创建更有效的索引（即更有效更窄的键）。此外，SQL Server 内部有帮助您查看实际结构的工具（不需要计算）。最重要的是，索引的大小（级数）取决于 3 个因素：索引定义、基表是否有聚集索引，以及索引叶级页面的数量。叶级页面的数量直接与表中的行大小和行数量相关。这并不是说定义索引的目标是为了仅拥有非常窄的索引（实际上，极其窄的索引的用途通常比宽索引更少）。这只是说明您应该理解各种索引选择和决定的含义。此外，*INCLUDE* 和筛选索引等功能能够深刻影响索引的大小和用途。但是，知道 SQL Server 的工作方式及索引的内部结构是确定合适索引数量，使索引既不太多也不太少的主要因素，不过最重要的是建立正确的索引。

6.2 分析索引的工具

为了完整地展示和理解索引结构，我们会用到几种工具。为了更易于理解，我们需要知道哪个工具最适合使用及什么时候最适合使用。此外，这部分主要概述执行选项及一些提示和技巧。本章后面会详细介绍有关分析输出各方面的信息。

6.2.1 使用 dm_db_index_physical_stats 动态管理视图

sys.dm_db_index_physical_stats 函数是用于确定表结构的最有用的函数之一。动态管理视图（DMV）可以让您知道表是否有聚集索引、有多少个非聚集索引及表是否有行溢出或大型对象（LOB）数据。最重要的是，DMV 还能向您展示整个结构及其健康状态。这个特殊的 DMV 函数需要 5 个参数（都使用默认值）。如果将所有参数都设置为默认值并且不筛选行或列，那么该函数几乎会为当前 SQL Server 实例中每个数据库每个分区上每个表每个索引的每个级别返回 21 列数据。按照如下方式请求该信息：

```
SELECT * FROM sys.dm_db_index_physical_stats (NULL, NULL, NULL, NULL, NULL);
```

在一个非常小的 SQL Server 实例（除系统数据库之外只有 *AdventureWorks2008*、 *pubs* 和 *Northwind* 数

据库）上执行时，会返回超过 390 行。显然，21 列和 390 行这么多的输出无法在这里列出，因此这是一条您应该练习以获得经验的命令。但是，不太可能真的看到每个数据库每个表上的每个索引（虽然这可能对较小的实例如一个开发实例有益）。为了将其改进成一个目的性更强的操作，我们来看一下下面的参数。

- ***Database_id***。第一个参数必须被指定为一个数字，但是如果您希望通过名称指定数据库，则可以插入 *DB_ID* 函数作为一个参数。如果指定 NULL（这是默认值），该函数将返回所有数据库的信息。如果数据库 ID 是 NULL，则接下来的 3 个参数一定也是 NULL（这些参数的默认值）。此外，该函数必须在至少具有 90 个兼容模式的数据库（表示 SQL Server 2005）中执行。如果由于某些原因数据库不是在至少兼容模式 90 中运行，则从 *master* 中执行该查询并指定一个数据库名称（DB_ID('databasename')）或具体的 ID，表示可以在不更改目标数据库兼容模式的情况下执行查询。
- ***object_id***。第二个参数是对象 ID，也必须是一个数字，而不是一个名称。同样，NULL 默认值表示您希望获得所有对象的信息，如果是这样，接下来的两个参数 *index _id* 和 *partition_id* 一定也是 NULL。与数据库 ID 一样，如果您知道对象名称，也可以使用一个嵌入函数（*OBJECT_ID*）获得对象 ID。需要提醒的是，如果您正在从另一个不同的数据库而不是当前数据库执行该操作的话，应该对 OBJECT_ID 函数使用一个由三部分组成的对象名称，包括数据库名称和架构名称。
- ***index_id***。第三个参数允许您指定一个特殊表中的索引 ID，同样，默认值 NULL 表示您需要获得所有索引。这里需要记住的一个有用事实是表上聚集索引的 *index_id* 总是 1。
- ***partition_number***。第 4 个参数表示分区号，NULL 表示您需要获得所有分区的信息。记住，如果您没有明确地在某个分区架构中创建一个表或索引，则 SQL Server 会认为它将在一个分区上创建。
- ***mode***。第 5 个也是最后一个参数是唯一一个默认值（NULL）不表示返回所有信息的参数。最后一个参数表示查询该函数时您希望返回的信息级别（因此直接影响执行速度）。调用该函数时，SQL Server 遍历页链查找分配页表或索引的指定分区。与 SQL Server 2000 中的 *DBCC SHOWCONTIG* 不同的是，该函数通常需要一个共享（S）表锁，*sys.dm_db_index_physical_stats*（和 SQL Server 2005 中的 *DBCC SHOWCONTIG*）只需要一个意向共享（IS）表锁，该锁与其他大部分锁兼容，我们将在第 10 章对此进行介绍。有效的输入是 DEFAULT、NULL、LIMITED、SAMPLED 和 DETAILED。默认值是 NULL，与 LIMITED 相对应。下面是后三个值的含义。
  - **LIMITED**。LIMITED 模式是最快的，同时扫描的页数最少。对于索引，这种模式只扫描索引的第一个非叶级。对于堆，通过表的 IAM 表避免扫描，然后关联页面可用空间（PFS）页面定义表的分配。这允许 SQL Server 根据页顺序（本章后面将做进一步介绍）而不是页密度（或者只能根据实际读取叶级页面计算得到的其他详细信息）获得关于碎片的详细信息。换言之，它很快但信息不够详细。更具体地说，这与现在已经废弃的 *DBCC SHOWCONTIG* 命令的 WITH FAST 选项相对应。
  - **SAMPLED**。SAMPLED 模式根据索引或堆中所有页面上 1%的样例及通过读取第一个中间级别的页面所得到的页面顺序返回物理特性。但是，如果索引拥有的总页面不足 10 000 页，则 SQL Server 会将 SAMPLED 转换成 DETAILED。
  - **DETAILED**。DETAILED 模式扫描所有页面并返回所有级别索引的所有物理特性（页面顺序和页密度）。分析一个小表时该模式非常有用，但是对于较大的表来说会花费相当多的时

间。如果正在被处理的索引比缓冲池大，则该模式还会“刷新”缓冲池。

使用内置 *DB_ID* 或 *OBJECT_ID* 功能时必须要小心。如果您指定的不是一个有效的名称或者只是写错了名称，则不会收到一条错误消息并且返回的值为 NULL。但是，由于 NULL 是一个有效的参数，因此 SQL Server 只是假设这是您要使用的参数。例如，为了查看所有以前介绍过的信息，但是只针对 *AdventureWorks2008* 数据库，那么您可能会错误地输入如下名称：

```
SELECT * FROM sys.dm_db_index_physical_stats
        (DB_ID ('AdventureWorks208'), NULL, NULL, NULL, NULL);
```

由于没有 *AdventureWorks2008* 之类的数据库，因此 *DB_ID* 函数返回 NULL，同时认为您调用该函数时使用的参数都是 NULL。不给出任何错误或警告消息。

您也许从返回的行数猜测到自己哪里出现了错误，但是，如果您不知道会有多少结果输出，那结果可能就不是很明显了。*SQL Server 联机丛书*建议您通过将 ID 捕获到变量中并且在调用 *sys.dm_db_index_physical_stats* 函数之前对变量中的值进行错误检查来避免这一问题，具体代码如下所示：

```
DECLARE @db_id SMALLINT;
DECLARE @object_id INT;

SET @db_id = DB_ID (N'AdventureWorks2008');
SET @object_id = OBJECT_ID (N'AdventureWorks2008.Person.Address');

IF (@db_id IS NULL OR @object_id IS NULL)
BEGIN
    IF @db_id IS NULL
    BEGIN
        PRINT N'Invalid database';
    END;
    ELSE IF @object_id IS NULL
    BEGIN
        PRINT N'Invalid object';
    END
END
ELSE
SELECT *
FROM sys.dm_db_index_physical_stats
     (@db_id, @object_id, NULL, NULL, NULL);
```

另一种隐蔽性更强的问题是在调用 *sys.dm_db_index_physical_stats function* 之前，*OBJECT_ID* 函数根据当前数据库进行调用。因此如果您正在操作 *AdventureWorks2008* 数据库，但是希望获得 *pubs* 数据库中某个表的信息，那么可以试着运行如下代码：

```
SELECT *
FROM sys.dm_db_index_physical_stats
     (DB_ID (N'pubs'), OBJECT_ID (N'dbo.authors'), NULL, NULL, NULL);
```

但是，由于当前数据库（*AdventureWorks2008*）中没有 *dbo.authors* 表，因此@*object_id* 传递的值为 NULL，同时您从 *pubs* 中的所有对象上获得所有信息。

如果两个数据库中具有同名的对象，则问题更难检测。如果 *AdventureWorks2008* 中有一个 *dbo.authors* 表，该表的 ID 将被用于从 *pubs* 数据库中检索数据——即使两个数据库中都存在 *authors* 表，两个表也不

可能具有相同的 ID。如果 *object_id()*返回的 ID 与指定数据库中的任何对象都不匹配，则 SQL Server 会返回一个错误，但是如果与另一个表中的对象 ID 相匹配，则会生产该表的详细信息，可能会引起更多的混淆。下面的脚本显示了这一错误：

```
USE AdventureWorks2008;
GO

CREATE TABLE dbo.authors
   (ID CHAR(11), name varchar(60));
GO

SELECT *
FROM sys.dm_db_index_physical_stats
     (DB_ID (N'pubs'), OBJECT_ID (N'dbo.authors'), NULL, NULL, NULL);
```

运行上面的 *SELECT* 时，*dbo.authors* ID 的值将由当前环境（仍然是 *AdventureWorks2008*）所决定。但是当 SQL Server 希望使用 *pubs* 中的 ID（该值不存在）时，会产生如下错误：

```
Msg 2573, Level 16, State 40, Line 1
Could not find table or object ID 295672101. Check system catalog.
```

最好的解决办法是完全限定表名，不论是调用 *sys.dm_db_index_ physical_stats* 函数本身还是利用变量获取完全限定表名的 ID（如在前面的代码实例中显示的那样）。如果使用包装器程序调用 *sys.dm_db_index_physical_stats* 函数，那么可以在检索对象 ID 之前将数据库名称与对象名称进行连接，从而避免问题。由于该函数的输出结果有点隐蔽，因此您可能会发现自己编写程序来访问该函数同时以一种更友好的方式返回信息会更有帮助。

总之，DMV 对于确定索引的大小及健康性极为有用。然而，您需要知道如何使用它来获得您感兴趣的特定信息。但是，即使对于表或索引的子集，而且小心地使用可用参数，仍然可能获得比需要信息更多的信息。由于 *sys.dm_db_index_ physical_stats* 是一个表值函数，因此您可以在返回的结果中添加自己的筛选器。例如，可以选择只查看非聚集索引的结果。使用可用的参数，您唯一的选择是查看所有索引或者只查看一个特殊的索引。如果使用第 3 个参数 NULL 指定所有索引，那么接下来可以在 WHERE 子句中添加一个筛选器来表示只想要非聚集索引行（WHERE *index_id*>1）。注意虽然一个 WHERE 子句可以限制返回的行数，但它不一定限制被分析的表和索引。

6.2.2 使用 DBCC ID

DBCC ID 命令（在第 5 章介绍过）未记录但被广泛了解和使用。在生产系统上使用很安全。该命令有 4 个参数，但是只需要前 3 个参数。下面的代码显示了该命令的语法：

```
DBCC IND ( { 'dbname' | dbid }, { 'objname' | objid },
      { nonclustered indid | 1 | 0 | -1 | -2 } [, partition_number] )
```

第 1 个参数是数据库名称或数据库 ID。第 2 个参数是数据库中的一个对象名称或对象 ID。对象可以是一个表或一个索引视图。第 3 个参数是一个特定的非聚集索引 ID（2～250 或 256～1005）或值 1、0、–1 或–2。该参数值的含义如下。

- **0**。显示行内数据页和指定对象行内 IAM 页的信息。
- **1**。显示所有页面的信息，包括 IAM 页面、数据页及被请求对象的任何现有 LOB 页或行溢出页。

如果被请求的对象有一个聚集索引，则包含索引页。

- **–1**。显示所有 IAM、数据页及指定对象上所有索引的索引页信息。包括 LOB 和行溢出数据。
- **–2**。显示指定对象所有 IAM 的信息。
- **非聚集索引 ID**。显示所有 IAM、数据页和某一索引的索引页信息。包括 LOB 和可能是索引中所包含列的一部分的行溢出数据。

最后的参数对于 SQL Server 2005 来说是一个新的参数，同时也是一个可选的参数（为了向后兼容可能使用 SQL Server 2000 中的 DBCC IND 的脚本）。它指定一个特殊的分区号，同时如果没有指定值或者给定值 0，则会显示所有分区的信息。

与 *DBCC PAGE*（在第 5 章介绍过）不同的是，SQL Server 不需要在运行 *DBCC IND* 之前启用追踪标志 3604。但是，由于您可能希望在确定索引拥有的页面之后使用 *DBCC PAGE* 分析页面，因此最好在开始编写程序之前开启追踪标志。

结果集中的列如表 6-1 所示。注意所有的页面参照将两列之间的文件和页面组件方便地分离，因此您不必进行任何转换。

表 6-1　*DBCC IND* 输出结果列的说明

列	含　义
PageFID	包含该页面的文件 ID
PagePID	文件内的页码
IAMFID	包含管理该页面的 IAM 的文件 ID
IAMPID	管理该页面的 IAM 文件内的页码
ObjectID	对象 ID
IndexID	索引 ID，有效值为 0～250 和 256～1005（将在后面介绍）
PartitionNumber	该页内表或索引的分区号
PartitionID	包含该页面的分区 ID（在数据库内是唯一的）
Iam_chain_type	该页面所属的分配单元类型：行内数据、行溢出数据或 LOB 数据
PageType	页类型：1=数据页，2=索引页，3=TEXT_MIXED_PAGE，4=TEXT_TREE_PAGE，10=IAM 页
IndexLevel	索引级别：0 为叶级，级别由叶向根页计数（IndexID 为 1～1005 的一种索引结构）
NextPageFID	包含该级别中下一个页面的文件 ID
NextPagePID	该级别中下一页所在文件的页码
PrevPageFID	包含该级别中上一个页面的文件 ID
PrePagePID	该级别上一个页面所在文件的页码

其中一些返回值已经在第 5 章介绍过了，因为这些值也与堆相关。在处理索引时，还可以查看 *IndexID* 列，0 表示堆，1 表示聚集索引页，2～1005 之间的数字表示非聚集索引页的页面。在 SQL Server 2008 中，一个表最多可以有 1000 个索引（1 个聚集索引和 999 个非聚集索引）。虽然 1005 超出了我们的需求（2～1000 足够 999 个非聚集索引使用了），但是非聚集索引 ID 跳过了 251～255，因为 255 在早期的版本中有特殊的含义，而 251～254 没有被使用。为了简化向后兼容问题，SQL Server 2008 也跳过了 251～255。

IndexLevel 值允许我们查看页面所在的索引树级别，0 值表示叶级。因此任何特殊索引的最大值是该

索引的根页，同时应该能够验证根页与从 *sys.system_internals_allocation_units* 视图的 *root_page* 列中获得的值相同。剩下的 4 列表示每个索引每一级别上的页链。对于每个页面来说，都有下一页的文件 ID 和页 ID 及上一页的文件 ID 和页 ID。当然，对于根页来说，所有这些值都是 0。也可以通过查找上一页值为零的页来确定第一页，同时可以找到最后一页，因为最后一页的下一页是零。DBCC 命令的输出结果太宽，书的一页显示不下，而且您很可能希望重新排序结果集，因此这里不显示输出结果了。如果您希望查看输出结果，可以使用一个脚本将该命令的输出存储到一个表中。在表中存储这一信息后，我们就可以从中查询和检索自己感兴趣的列了。下面是创建一个名为 *sp_tablepages* 的表的语句，表中存储 *DBCC IND* 函数返回的所有信息列。注意 *master* 数据库中所有以 *sp_*开头的对象都可以从任何数据库中访问，不必限定数据库的名称：

```
USE master;
GO
CREATE TABLE sp_tablepages
(PageFID tinyint,
  PagePID int,
  IAMFID tinyint,
  IAMPID int,
  ObjectID int,
  IndexID tinyint,
  PartitionNumber tinyint,
  PartitionID bigint,
  iam_chain_type varchar(30),
  PageType tinyint,
  IndexLevel tinyint,
  NextPageFID tinyint,
  NextPagePID int,
  PrevPageFID tinyint,
  PrevPagePID int,
  Primary Key (PageFID, PagePID));
```

下面的代码截断 *sp_tablepages* 表并用从 *AdventureWorks2008* 数据库中 *Sales.SalesOrderDetail* 表获取的 *DBCC IND* 结果进行填充：

```
TRUNCATE TABLE sp_tablepages;
INSERT INTO sp_tablepages
    EXEC ('DBCC IND (AdventureWorks2008, [Sales.SalesOrderDetail], -1)');
```

当表中有 *DBCC IND* 的结果后，可以选择自己感兴趣的行或列的任何子集。我们使用 *sp_tablepages* 报告本章很多示例的 *DBCC IND* 信息，然后便可以利用 *DBCC PAGE* 检查索引页，与对数据页所做的操作一样。但是，如果具有样式 3 的 *DBCC PAGE* 输出索引页上每行中每一列的详细信息，则输出结果看起来会完全不同。接下来在我们分析索引的物理结构时将会看到一些示例。

6.3 理解索引结构

正如我们在本章前面介绍过的那样，索引结构被分成索引的两个基本部分：叶级和非叶级。这一部分的详细内容将帮助您更好地理解索引这部分中具体存储了哪些内容及它们如何根据索引类型的不同而变化。

6.3.1 聚集键的依赖关系

聚集索引的叶级不仅包含索引键，还包括数据。因此对于“除键值外，聚集索引的叶级中还有哪些内容”这样的问题来说，答案就是“所有内容”——表中每一行的所有列都在聚集索引的叶级中。用另一种表达方式来说就是当一个聚集被创建时，数据成为聚集索引的叶级。在聚集索引键被创建时，表中的数据会按照聚集键的顺序排列和复制。一旦被创建，聚集索引就会从逻辑上而非物理上进行维护。该顺序通过一个被称为*页链*的双链表进行维护（注意堆中的页没有以任何方式进行相互链接）。页链中页面的顺序及数据页中行的顺序是由聚集索引的定义决定的。确定在哪一列上建立聚集索引是影响性能的重要因素。

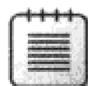
由于数据页的实际页链只能以一种方式排序，因此表只能有一个聚集索引。同时，一般来说大部分表在建立聚集索引后的执行效果会更佳。但是需要明智地选择聚集键。为了适当地选择聚集键，您必须知道聚集索引的工作方式及聚集键的内部依赖关系（尤其是对非聚集索引来说）。

非聚集索引对聚集键的依赖关系自存储引擎在 SQL Server 7.0 中被重建之后就一直在 SQL Server 中存在了。这是在使用一个非聚集索引参照表中的一个相应行时，行是如何被标识（和查找）开始的。如果一个表有一个聚集索引，则行通过它们的聚集键进行标识（和查找）。如果表没有聚集索引，则行通过它们的物理行标识符（RID）进行标识（和查找），我们将在本章后面对此进行详细介绍。在基表中查找相应数据行的过程被称为*[书签]查找*，这是类比非聚集索引引用书中某个位置（如同书签那样操作）进行命名的。

非聚集索引只包括由索引定义的数据。当查找非聚集索引中的一行时，经常需要定位真实的数据行，查找不是非聚集索引一部分的其他数据。为了检索这些其他数据，必须在表中查找这些数据。这里我们只关注表有聚集索引时书签查找是如何执行的。

首先也是最重要的，所有聚集索引必须是唯一的。聚集索引必须唯一的主要原因是使非聚集索引项可以精确地指向某一特定的行。思考一下表按照一个非唯一的值建立聚集索引时会出现什么问题。如果一个非聚集索引存在一个唯一值（例如社会安全号），同时某个查询在索引中查找 123-45-6789 的特定社会安全号，发现其聚集键为'Smith'，此时如果存在多个具有姓 Smith 的行，那么就无法定位是哪一行。如何才能有效地定位到社会安全号为 123-45-6789 的特定行上？

为了有效地使用聚集键，所有非聚集索引项必须精确地指向某一行。由于指针在 SQL Server 中是聚集键，因此聚集键必须唯一。如果在不指定 *UNIQUE* 关键字的情况下构建一个聚集索引，则 SQL Server 可以在必要的时候通过向行添加一个隐藏的唯一标志列来保证内部的唯一性。



注意：

在 *SQL Server* 联机丛书中，***uniquifier*** 写作 uniqueifier。但是，内部工具（如 *DBCC PAGE*）会写成 uniquifier。

唯一标志是在行的聚集键不唯一时添加到数据行上的一个 4 字节整数值。一旦被添加，该值将成为密钥的一部分，也就是说将被复制到每个非聚集索引中。当您查看索引行的实际结构时，可以查看某一特定行是否有这一额外值，我们将在本章后面进行介绍。

其次，如果一个聚集键用于查找某个非聚集索引中聚集索引（数据）的相应数据行，则聚集键是表中最过度复制的数据。构成聚集键的所有列除包括在真实数据行中之外，还包括在每个非聚集索引中。

因此，聚集键的宽度很重要。考虑一下某个具有 12 个非聚集索引和 1 百万行数据的表上有一个 64 字节聚集键的聚集索引，不计算内部和结构开销，仅存储每个非聚集索引中的聚集键的开销就是 732MB，而聚集键只有 8 字节时开销只有 92MB，聚集键只有 4 字节时开销只有 46MB。虽然这只是一种大致估算，但却说明了聚集键过宽会浪费很多空间（可能是缓冲池的内存）。但是，这不仅是空间的问题，还会转化成非聚集索引的性能和效率问题。一般来说，不应该使非聚集索引过宽。

再次，由于聚集键是整个表中最冗余的数据，因此您应该选择一种不易变的列作为聚集键。聚集键改变会产生很多副作用。首先，它会引起聚集索引中的记录重新定位（这可能导致页面分离和碎片，我们将在本章后面对此进行详细介绍）。其次，它会导致每个非聚集索引被修改（这样才能保证相关非聚集索引行的聚集键值是正确的）。这会浪费时间和空间，产生碎片（碎片需要维护），并会由于对构成聚集键的列的修改而增加不必要的开销。

这三个属性（唯一性、窄和静态）也会（但不一定）应用到一个适当的主键上，同时由于只能有一个主键（并且只能有一个聚集键），因此 SQL Server 使用唯一聚集索引来强制主键约束（主键定义中没有定义索引类型时）。但是，表的创建者并不一定知道这一点。如果主键不遵守这些标准（例如，主键是从数据的自然键中选择的，例如，一个宽 7 列的 100 字节组合（只有 7 列组合才唯一）），那么使用一个聚集索引强制唯一性并复制每个非聚集索引中的整个 100 字节的列组合可能会产生非常严重的负面影响。因此，对于一些可信任的数据库开发人员来说，一个非常宽的聚集键可能是由它们的默认值创建的。幸运的是，您可以将主键定义为非聚集索引，并且可以很容易地在不同列上（或一组列上）创建非聚集索引。但是，您必须知道什么时候及如何创建。

最后，还应该选择表的一个聚集键，从而使插入产生的碎片最少（本章后面将对碎片进行深入介绍）。虽然聚集索引被创建后只有它的逻辑顺序被维护，但是对这一结构的维护会占用系统开销。如果行始终需要在随机的入口点输入到表中（例如，按照姓氏的顺序插入到表中），那么维护表的逻辑顺序比向表末尾添加行的代价要大得多（例如，向按照订购编号排序的表中插入数据，这是也应该是一个不断增长的标识列）。

更多信息将在本章后面介绍索引内部结构时看到，但是为了对目前为止所做的讨论进行总结，不仅应该根据表的用途（同时，就聚集键而言，真的很难说“一直”或“从不”）而且应该根据 SQL Server 对聚集键的内部依赖关系选择聚集键。对于后者来说，聚集键应该是唯一的、窄的和静态的，最好是不断增长的。

良好的聚集键的示例如下。

- 用一个不断增长的标识列定义一个列键（例如，一个 4 字节的 *int* 或 8 字节的 *bigint*）。
- 用一个不断增长的日期列（第一列）和唯一标识行的第二列（如一个标识列）定义一个组合键。这对于基于日期分区的表及按照日期递增顺序插入数据的表来说可能非常有用，因为可以为基于日期的范围查询提供其他优势。这样的示例包括一个由 *SalesDate*（8 字节）和 *SalesNumber*（4 字节 *int*）组成的 12 字节组合键，或者 SQL Server 2008 中一个不包括时间的日期列。但是，日期不是一个很好的聚集键，因为日期不唯一（并且需要唯一标志）。
- 一个 GUID 列可以成功地用做一个聚集键，因为它明显是唯一的，相对较窄（16 字节宽）并且可能是静态的。但是，作为一个聚集键，只有当 GUID 遵循一种不断增长的模式时才是适合的。某些 GUID（由其生成方式决定）可能会造成很多碎片。如果 GUID 是在 SQL Server 外部生成或者是在 SQL Server 内部利用 *NEWID ()*函数生成的，则碎片会降低该列作为聚集键的有效性。如果可能，可以考虑使用 *NEWSEQUENTIALID ()* 函数（用于不断增长的 GUID）或选择另一

种聚集键。如果仍然希望使用一个GUID作为主键并且它不是不断增长的，则可以使其成为一个非聚集索引而不是聚集索引。

总之，选择聚集键没有一个绝对的标准，只有对大部分表来说效果很好的一般实践。但是，如果一个表只有一个索引（并且没有非聚集索引），则非聚集索引对聚集键的依赖关系不再重要，同时聚集索引可以采取任何形式。但是，大部分表一般都至少有几个非聚集索引，同时大部分表有聚集索引的时候性能更佳。由于这一实际情况，因此实现更佳性能的第一步是拥有适当聚集键的聚集索引。第二步是通过选择合适的（通常是一个比较小的）非聚集索引在非聚集索引中“找到适当的平衡”。

6.3.2 非聚集索引

正如前面介绍的那样，所有索引都有两个主要部分：叶级和非叶级。对于聚集索引来说，叶级是数据。对于非聚集索引来说，叶级是一个独立和附加的结构，其中存储某些数据的副本。具体来说，一个非聚集索引依赖于它的定义来形成叶级。非聚集索引的叶级由索引键（按照索引的定义）、包含的所有列（使用SQL Server 2005中添加的*INCLUDE*功能）及数据行的书签值（如果表是聚集的则是聚集键，如果表是一个堆则是行的物理RID）构成。一个非聚集索引中行的数量与表中行的数量是完全相同的，除非在定义索引时使用了一种筛选器谓词。筛选索引是SQL Server 2008中的新增内容，我们将在本章后面对这一内容进行详细介绍。

就非聚集索引*如何*被使用而言，实际上有两种方式：帮助指向数据（与树后面的索引类似，如前面介绍的那样使用书签查找）或直接回答一个查询。当一个非聚集索引有查询请求的所有数据时称为*查询覆盖*，同时索引被称为*覆盖索引*。当一个非聚集索引覆盖一个查询时，非聚集索引可用于直接回答一个查询，同时可以避免*书签查找*（书签查找对于非选择性查询来说代价很大）。这可能是提高范围查询性能的最有效方式。

当非聚集索引中没有查询所需的所有数据，但是查询是由索引可能帮助查找的谓词驱动时，会出现对行的书签查找。如果一个表有一个聚集索引，则非聚集索引用于通过聚集键驱动查询查找相应的数据行。如果表是一个堆（换言之，表没有聚集索引），则查找值是一个8字节RID，这是一个*FileID:PageID:SlotNumber*形式的真实行定位器。在8字节行标识符中，2字节用做*FileID*，4字节用做*PageID*，2字节用做*SlotNumber*。在本章后面介绍数据访问时会确切看到这些查找值是如何被使用的。

非聚集索引的存在与否不会影响数据页的组织方式，因此每个表中可以有多个非聚集索引，而聚集索引只能有一个。在SQL Server 2008中，每个表最多可以包含999个非聚集索引（SQL Server 2005中是249个），但是您通常只需要其中的一小部分（一些特例除外，如筛选索引）。

总之，非聚集索引不会影响基表。但是，基表的结构（或者是一个堆或者是一个具有聚集索引的表）会影响非聚集索引的结构。如果您希望减少系统开销并实现最好的性能，则需要考虑和理解这些问题。

6.3.3 约束和索引

正如前面所提到的那样，一个被信任的数据库开发人员可能在创建一个PRIMARY KEY约束时不小心在表上创建了一个聚集索引。使用约束的概念源自关系理论，其中表定义了实体标识符（用于理解表的关系并帮助加入到规范化架构中的表）。当在SQL Server中的表上定义约束时，PRIMARY KEY和UNIQUE KEY约束可以强制数据库中实体完整性的某些方面。

对于PRIMARY KEY约束来说，SQL Server强制两件事情：首先，PRIMARY KEY中包含的所有列都不允许为空；其次，PRIMARY KEY值在表中是唯一的。如果任意一列允许NULL值，则不能建立

PRIMARY KEY 约束。为了强制唯一性，SQL Server 在构成 PRIMARY KEY 约束的列上创建了一个唯一索引。如果不明确指定，则默认的索引类型是唯一聚集索引。

对于 UNIQUE 约束来说，SQL Server 允许构成 UNIQUE 约束的列为空，但是不允许有一行以上的所有列都为空。为了强制 UNIQUE 约束的唯一性，SQL Server 在构成约束的列上创建一个唯一索引。如果不明确指定，则默认索引类型为唯一非聚集索引。

当您声明一个 PRIMARY KEY 和 UNIQUE 约束时，创建的基础索引结构与直接使用 *CREATE INDEX* 命令创建的相同。但是在用途和功能方面会有一些区别。例如，一个基于约束的索引不能添加其他功能，但是一个 UNIQUE 索引在强制索引键定义唯一性的同时可以有这些功能。在引用一个 UNIQUE 索引（不支持约束）时，不能参照具有筛选器的索引。但是，不使用筛选器或者使用包含列的索引可以被参照。这些可能是用于减少索引总数量同时仍能创建一种具有 FOREIGN KEY 约束的参照的有效选项。

用于支持这些约束的索引名称与约束名称相同。就内部存储及索引的工作方式而言，使用 *CREATE INDEX* 命令创建的唯一索引与为支持约束而创建的索引之间没有区别。查询优化器根据唯一索引（而不是列）是否声明作为一个约束而做出决定。换言之，索引的创建方式与查询优化器无关。

6.4 索引创建选项

就创建索引而言，*CREATE INDEX* 命令比较简单：

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
   ON <object> ( column [ ASC | DESC ] [ ,...n ] )
    [ INCLUDE ( column_name [ ,...n ] ) ]
    [ WHERE <filter_predicate> ]
```

索引所需的部分是索引名称、键定义及定义索引的表。一个索引可以利用 *INCLUDE* 在索引的叶级上包括非键列。索引可以在表的整个行集上定义（这是默认情况），在 SQL Server 2008 中，索引可以被限制为只能是通过筛选器使用 WHERE <filter_predicate>定义的行。我们将在分析非聚集索引的物理结构时讨论这两种情况。

但是，*CREATE INDEX* 还有一些附加选项可以在特殊情况下使用。您可以为 *CREATE INDEX* 命令添加一个 WITH 子句：

```
 [WITH
([FILLFACTOR = fillfactor]
[[,] [PAD_INDEX] = { ON | OFF }]
[[,] DROP_EXISTING = { ON | OFF }]
[[,] IGNORE_DUP_KEY = { ON | OFF }]
[[,] SORT_IN_TEMPDB = { ON | OFF }]
[[,] STATISTICS_NORECOMPUTE = { ON | OFF }]
[[,] ALLOW_ROW_LOCKS = { ON | OFF }]
[[,] ALLOW_PAGE_LOCKS = { ON | OFF }]
[[,] MAXDOP = max_degree_of_parallelism]
[[,] ONLINE = { ON | OFF }] )]
```

FILLFACTOR、PAD_INDEX、DROP_EXISTING、SORT_IN_TEMPDB 和 ONLINE 索引创建选项主要是为维护索引而定义和使用的。为了适当地使用这些选项，您必须更好地了解索引的物理结构及数据修改方式。我们将在本章后面 “管理索引结构”一节对这些选项进行详细介绍。下面介绍剩下的选项。

6.4.1 IGNORE_DUP_KEY

可以通过将索引定义为唯一索引，或者通过定义主键或唯一约束来保证索引键的唯一性。如果一条 *UPDATE* 或 *INSERT* 语句会影响多行，或者即使发现一行会引起定义为唯一的键重复，那么整条语句都会被中止，同时所有行不会受到影响。创建唯一索引时，可以选择使用 IGNORE_DUP_KEY 选项，从而使多行 *INSERT* 的重复键错误不会使整条语句回滚。非唯一行将被抛弃，其他所有行都会被插入。IGNORE_DUP_KEY 不允许违反索引的唯一性，而是使对多行数据的错误修改对所有有效行不构成致命威胁。

6.4.2 STATISTICS_NORECOMPUTE

STATISTICS_NORECOMPUTE 选项决定索引上的统计信息数据是否应该自动更新。每个索引都维护一个柱状图，该图表示索引引导列值的分布情况。同时查询优化器在选择一种查询计划时使用这些统计信息确定特殊索引的用途。当数据被修改时，统计信息逐渐过期，如果统计信息不被更新则会使查询计划的优化性降低。在第 3 章中，您已经学习了数据库选项 AUTO_UPDATE_STATISTICS，该选项可以使数据库中的所有统计信息在必要时被自动更新。一般来说，数据库的该选项应该被启用。但是，根据需要可以利用 AUTO_UPDATE_STATISTICS 选项将特殊的统计信息或索引设置为不自动更新。添加该子句会将 AUTO_UPDATE_STATISTICS 数据库选项设置为 *ON*。如果数据库选项设置为 *OFF*，则不能覆盖特殊索引的这一行为，此时数据库中的所有统计信息必须使用 *UPDATE_STATISTICS* 或 *sp_updatestats* 手动更新。为了查看某一给定表的统计信息数据是否设置为自动更新及被更新的最后时间，可以使用 *sp_autostats<table_name>*。

6.4.3 MAXDOP

MAXDOP 选项控制可用于创建索引的最大处理器数，它可以为索引构建覆盖服务器配置选项 *max degree of parallelism*。允许使用多个处理器进行索引创建大大增强了索引构建操作的性能。与其他并行操作一样，查询优化器根据系统的当前负载确定运行时实际使用的处理器数量。MAXDOP 值只是设置一个最大值。只有运行 SQL Server 企业版或 SQL Server 开发人员版本时多处理器才可用于索引创建。使用多处理器时，每个处理器都建立一个大小相等的平行索引块。此时树可能不是完全平衡的，同时用于确定所需页面理论上最小值的数学结果也与实际值不符，因为每个并行线程都会建立一个独立的树。当每个线程都结束时，树基本上被连在了一起。SQL Server 可以使用这种在并行处理期间为以后修改而保留的任何额外页面空间。

6.4.4 索引放置

CREATE INDEX 命令中的最后一个子句用于指定索引的位置：

```
[ ON { partition_scheme_name ( column_name )
       | filegroup_name } ]
```

您可以指定一个索引是应该放在某个特殊的文件组上，还是应该按照预定义分区方案进行分区。默认情况下，如果没有指定文件组或分区机制，索引会与基表放在同一个文件组中。我们在第 3 章讨论了文件组，同时您将在第 7 章学习表和索引分区。

6.4.5 约束和索引

应该使用 UNIQUE 或 PRIMARY KEY 约束还是应该使用 *CREATE INDEX* 命令定义唯一索引是一个普遍关注的问题，也是一个经常容易混淆的问题。正如前面所述，使用 *CREATE INDEX* 命令或使用为支持主键约束而建立的唯一聚集索引在内部结构或查询优化器的选择上没有任何区别。区别实际上是一个设计问题，设计问题不是本书讨论的范围，本书介绍 SQL Server 的内部。但是可以进行简单的区分：约束是逻辑上的概念，而索引是物理上的概念。建立索引时，是在要求 SQL Server 创建一个占用存储空间并且确定在数据修改期间必须被维护的物理结构。定义约束时，实际上是在定义数据的一种属性并希望 SQL Server 强制该属性，但没有告诉 SQL Server *如何*进行强制。在当前的版本中，SQL Server 通过创建唯一索引强制 PRIMARY KEY 和 UNIQUE KEY 约束，但是不需要进行这样的操作。在将来的版本中，SQL Server 可能通过另一种机制进行强制，不过由于向后兼容性的问题，Microsoft 不太可能进行强制。

6.5 物理索引结构

索引页与数据页的结构几乎完全相同，只是索引页存储的是索引记录，而数据页存储的是数据记录。与 SQL Server 中所有其他类型的页面一样，索引页使用 8KB 或 8192 字节的固定大小。索引页也有一个 96 字节的标题，同时页结尾处每一行有 2 字节的偏移阵列表示页面上该行的偏移。一个非聚集索引可以使所有 3 个分配单元（IN_ROW_DATA、 ROW_OVERFLOW_DATA 和 LOB_DATA）与之相关。每个索引在 *sys.indexes* 目录视图中都有一行，其 *index_id* 值或者为 1（用于聚集索引），或者是 2～250 或 256～1005 之间（表示一个非聚集索引）的一个数字。注意 SQL Server 保留了 251～255 之间的值。

6.5.1 索引行格式

索引行的结构与数据行基本相同，除了两种特殊的情况。首先，一个索引行不能有 *SPARSE* 列。如果一个 *SPARSE* 列在一个索引定义中使用（对于索引中哪里可以使用 *SPARSE* 列有一些限制，如不能用于 PRIMARY KEY 中），则该列在索引行中创建时好像没有被定义为 *SPARSE*。其次，如果一个聚集索引被创建并且索引没有定义为唯一索引，则重复键值会包括一个唯一标志。

索引和数据行在结构上还有很多差异。索引行不使用 *TagB* 或 *Fsize* 行标题值。替代 *Fsize* 字段（该字段表示一行的固定长度部分在哪里结束），页眉 *pminlen* 值用于解码索引行。*pminlen* 值表示行的固定长度数据部分结束位置的偏移量。如果索引行没有变长或可为空的列，则 *pminlen* 就是行的结束位置。只有索引行有可为空的列时，被称为 *Ncol* 的字段和空位图才同时存在。*Ncol* 字段有一个表示索引行中包含多少列的值，该值用于确定空位图中有多少位。数据行有一个 *Ncol* 字段和空位图，不管是否有行允许为空，如果索引中的任意一列允许为空，则索引行只有一个空位图和一个 *Ncol* 字段。表 6-2 显示了索引行的标准格式。

表 6-2　　索引行中存储的信息

信　息	助 记 符	大　小
状态位 A	*TagA* 一些相关的位如下。 ■ 位 1～位 3：认为是一个 3 位值。0 表示一个主记录。3 表示一个索引记录。5 表示一个备份索引记录（备份索引记录将在本章后面介绍）。 ■ 位 4：表示存在一个 NULL 位图。 ■ 位 5：表示行中存在变长列	1 字节

续表

信 息	助 记 符	大 小
固定长度的数据	*Fdata*	*Pminlen*——1
列数量	*Ncol*	2字节
NULL位图（表中的每一列1位；1表示相应列为空）	*Nullbits*	Ceiling(*Ncol/8*)
变长列的数量；只有if>0时存在	*VarCount*	2字节
可变列偏移阵列；只有*VarCount*>0时存在	*VarOffset*	2**VarCount*
变长数据，如果有的话	*VarData*	

存储在索引行中的特定列数据由索引类型及索引行所在的级别决定。

6.5.2 聚集索引结构

聚集索引的叶级是数据本身。当一个聚集索引被创建时，数据会根据聚集键进行物理上的复制和排序（如本章前面介绍的那样）。聚集索引的行结构和堆的行结构之间除一种特殊情况之外没有区别，这种特殊情况就是聚集键没有定义UNIQUE属性时，此时SQL Server必须从系统内部保证唯一性，为此，每个复制行需要一个附加的唯一标志值。

具有唯一标识的聚集索引行

正如前面提到的那样，如果聚集索引创建时没有*UNIQUE*属性，则SQL Server会添加一个4字节的integer来使每个非唯一键值唯一。由于聚集键用于表示被非聚集索引参照的基行（书签查找），因此需要一种引用聚集索引中每一行的唯一方式。

SQL Server只有在必要时才添加唯一标志，即重复键被添加到表中时。举一个例子，我们创建所有列的长度都固定的一个小表，然后向表中添加一个聚集且非唯一的索引：

```
USE AdventureWorks2008;
GO

CREATE TABLE Clustered_Dupes
   (Col1 CHAR(5) NOT NULL,
    Col2 INT NOT NULL,
    Col3 CHAR(3) NULL,
    Col4 CHAR(6) NOT NULL);
GO

CREATE CLUSTERED INDEX Cl_dupes_col1 ON Clustered_Dupes(col1);
```

如果查看*sysindexes*兼容性视图中该表对应的一行，您会发现某些不可预期的结果：

```
SELECT indid, keycnt, name FROM sysindexes
WHERE id = OBJECT_ID ('Clustered_Dupes');

RESULT:
indid    keycnt   name
------   ------   --------------
```

```
1          2          Cl_dupes_col1
```

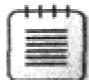
被称为 *keycnt* 的列（表示索引拥有的键数量）的值为 2（注意该列只有在兼容视图 *sysindexes* 中可用，在 *sys.indexes* 目录视图中不可用）。如果该索引有 *UNIQUE* 属性，则 *keycnt* 值将为 1。由于不推荐在非唯一键上创建聚集索引（使行唯一的过程浪费时间和空间），因此我们不对该结构进行完整分析。但是，在本章配套内容的资源素材（可以通过 http://www.SQLServerInternals.com/companion 访问）中包含一个名为 ExaminingtheClusteredIndexUniquifier.sql 的脚本。该脚本在聚集键没有定义为 *UNIQUE* 时创建和分析聚集索引行结构。

6.5.3 聚集索引的非叶级

为了导航到某个索引的叶级，创建一棵 B 树（其中包括叶级中的数据行）。非叶级中的每一行都为该级别下面的每一页提供一个目录（本章后面将进一步查看每一种索引类型的具体情况），该目录包括一个索引键值和一个引用该页面的 6 字节指针。此时，页面指针的格式是 2 字节用于 *FileID*，4 字节 *PageNumberInTheFile*。SQL Server 不需要一个 8 字节 RID，因为不需要存储槽编号。该目录的索引键部分总是表示被指向页面上可能存在的最小值。注意它们不一定表示实际上的最小值，而是表示该页上最小的可能值（当页面上具有最小键值的行被删除时，上级的索引行不会被更新）。

6.5.4 分析聚集索引结构

为了更好地说明聚集索引的存储及遍历方式，我们查看在名为 IndexInternals 的样例数据库中创建的特定结构。对于这个示例来说，我们查看在 PRIMARY KEY 上使用聚集索引创建的一个 *Employee* 表。



注意：

IndexInternals 样例数据库可以从网上下载。该数据库中已经有几个表了。查看 *EmployeeCaseStudy-TableDefinition.sql* 脚本获得表的定义，然后利用 *EmployeeCaseStudy-AnalyzeStructures.sql* 脚本分析结构。可以在配套内容中找到该数据库的一个备份和一个包含解决方案的 zip 文件。

下面是表 *Employee* 的结构定义语句，该表已经在 *IndexInternals* 数据库中存在。

```
CREATE TABLE Employee
  (EmployeeID       INT           NOT NULL     IDENTITY,
   LastName         NCHAR(30)     NOT NULL,
   FirstName        NCHAR(29)     NOT NULL,
   MiddleInitial    NCHAR(1)      NULL,
   SSN              CHAR(11)      NOT NULL,
   OtherColumns     CHAR(258)     NOT NULL     DEFAULT 'Junk');
GO
```

Employee 表是利用与标准最佳方法有些不同的方法创建的，这样可以使其结构稍具可预测性（例如，更简单的数学函数和更简单的可视化）。首先，所有列都有固定的宽度，但是数据值可变。不是所有列都应该因为数据值可变而可变，而是当列超过 20 个字符并且数据变化（不是非常不稳定）时，才考虑使用变长字符列而不是宽度固定的列，这样可以节省空间同时实现更好的 *INSERT* 性能（可能包含 *UPDATE* 性能，尤其当更新使宽度可变的列更大时）。我们将在本章后面介绍碎片时对这一内容做进一步介绍。在这些特殊的表中，宽度固定的列用于保证行大小可预测，同时帮助更好地使数据结

构可视化。

此时（同时包括系统开销），*Employee* 表中的数据行每行都是 400 字节（使用一个称为 *OtherColumns* 的填充列，这样将在数据行的末尾添加 258 个废弃字节）。400 字节的行大小表示每页有 20 行（8096 字节每页/400 字节每行=20.24，转换成每页 20 行，因为数据行的 IN_ROW 部分不能跨页）。为了计算表的大小，我们需要知道这些表中包含了多少行。同时在 *IndexInternals* 数据库中，该表已经精确地设置为 80 000 行。每页 20 行，该表需要 4000 个数据页来存储 80 000 行。

在当前的表定义中，该表是一个堆。对于 *Employee* 表来说，我们使用一个 PRIMARY KEY 约束定义聚集索引：

```
-- Add the CLUSTERED PRIMARY KEY for Employee
ALTER TABLE Employee
  ADD CONSTRAINT EmployeePK
    PRIMARY KEY CLUSTERED (EmployeeID);
GO
```

为了进一步研究 *Employee* 表，我们利用 *sys.dm_db_index_physical_stats* 来确定表中的页数及索引中的级数。可以使用 DMV 确定索引结构以查看级数和每一级的页数：

```
SELECT index_depth AS D
     , index_level AS L
     , record_count AS 'Count'
     , page_count AS PgCnt
     , avg_page_space_used_in_percent AS 'PgPercentFull'
     , min_record_size_in_bytes AS 'MinLen'
     , max_record_size_in_bytes AS 'MaxLen'
     , avg_record_size_in_bytes AS 'AvgLen'
FROM sys.dm_db_index_physical_stats
      (DB_ID ('IndexInternals')
     , OBJECT_ID ('IndexInternals.dbo.Employee')
     , 1, NULL, 'DETAILED');
GO

RESULT:
D   L   Count   PgCnt   PgPercentFull       MinLen  MaxLen  AvgLen
--- --- ------  ------- ------------------  ------- ------- ------
3   0   80000   4000    99.3081294786261    400     400     400
3   1   4000    7       91.7540400296516    11      11      11
3   2   7       1       1.09957993575488    11      11      11
```

假设共有 80 000 行，每一页 20 行，那么该表的聚集索引有一个 4000 页的叶级。从 *MinLen* （*min_record_size_in_bytes*）列中，可以看到叶级中的行长度是 400 字节。但是，非叶级行长度只有 11 字节。该结构很容易划分：4 字节用于定义聚集索引的 integer 列（*EmployeeID*），6 字节用于页指针，1 字节用于行的系统开销。由于我们的索引行只包含宽度固定的列，同时这些列中的任意列都不允许 NULL（因此我们不需要在索引页中包含 NULL 位图），因此只需要 1 字节的系统开销。此外，可以看到叶级上面的第一级中有 4000 行，因为级别 1 有一个 4000 的 *Count*（*record_count*）。实际上在级别 1 中只有 7 页（如 *PgCnt(page_count)*所示），同时在级别 2 中，您可以看到 *Count* 显示为 7。这里要引用本章前面（我们介绍树上的每一级都包含级别下面每一页的一个指针）的内容。如果一个级别有 4000 页，则上面的一个级别有 4000 行。可以在下面的图 6-2 中详细查看这一结构。

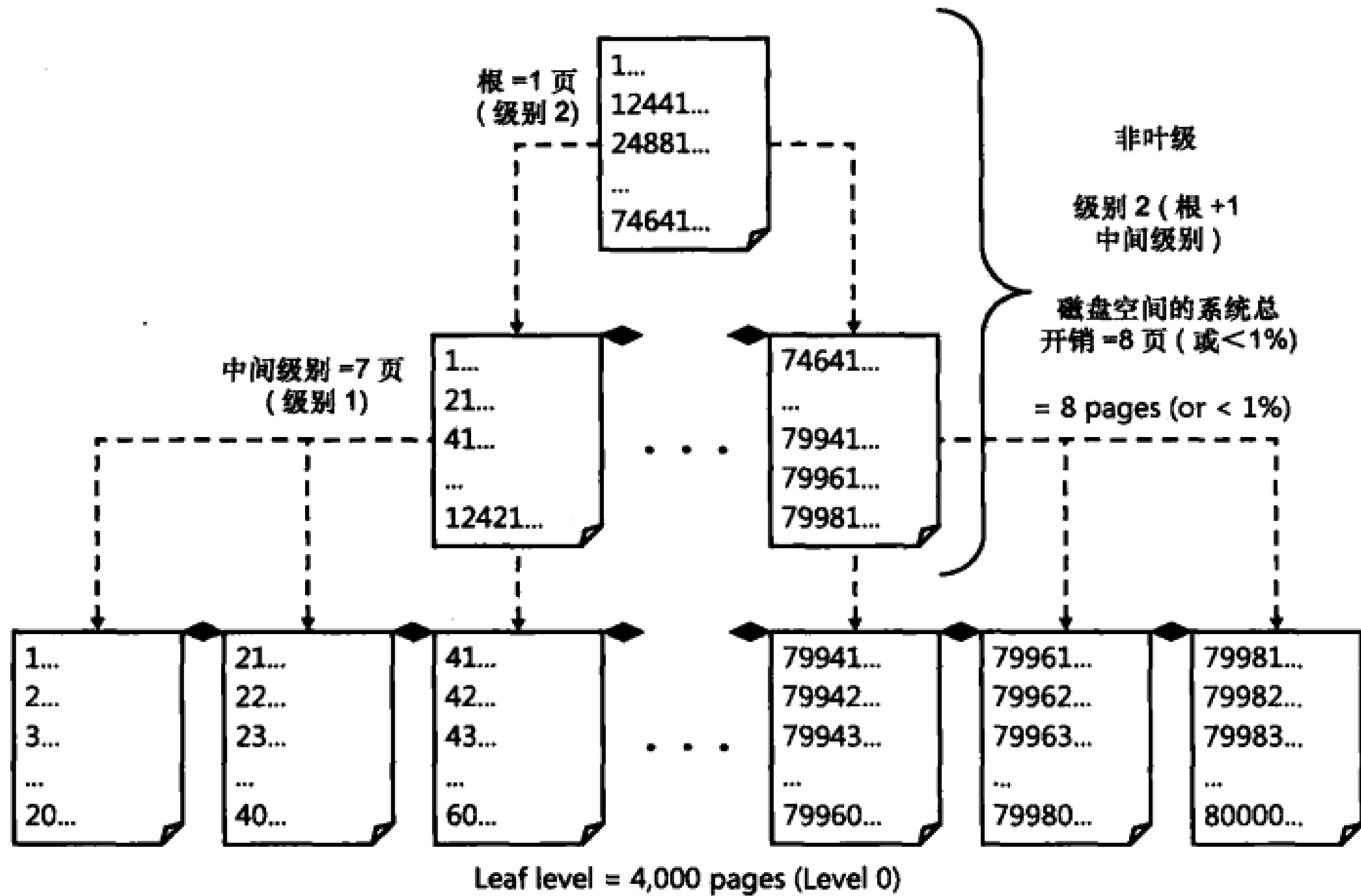


图 6-2 多重索引级别的页面细节

为了进一步理解遍历及连接，可以利用 *DBCC IND* 命令查看哪些页具有哪些数据，以及哪些页位于索引所有级别中各种页面的前面和后面。这里将 *DBCC IND* 的结果插入到 *master* 数据库的 *sp_tablepages* 表中，从而使我们可以只访问特殊列的信息：

```
TRUNCATE TABLE sp_tablepages;
INSERT sp_tablepages
        EXEC ('DBCC IND (IndexInternals, Employee, 1)');
GO

SELECT IndexLevel
     , PageFID
     , PagePID
     , PrevPageFID
     , PrevPagePID
     , NextPageFID
     , NextPagePID
FROM sp_tablepages
ORDER BY IndexLevel DESC, PrevPagePID;
GO

RESULT (abbreviated):
IndexLevel   PageFID  PagePID      PrevPageFID  PrevPagePID  NextPageFID  NextPagePID
----------   -------  -----------  -----------  -----------  -----------  -----------
2            1        234          0            0            0            0
1            1        232          0            0            1            233
1            1        233          1            232          1            235
1            1        235          1            233          1            236
```

```
1           1       236       1         235       1        237
1           1       237       1         236       1        238
1           1       238       1         237       1        239
1           1       239       1         238       0        0

0           1       168       0         0         1        169
0           1       169       1         168       1        170
<snip>
0           1       4230      1         4229 1 4231
0           1       4231      1         4230 0 0

NULL        1       157       0         0 0 0
```

由于该表是在数据库为空的时候创建的，而且聚集索引是在将数据加载到一个临时区域后（这完全是为临时存储数据而设的一个独立位置，本例中是一个不同的文件组）创建的，因此表的聚集索引能够使用文件 1 内完全相邻的页。但是，页面从根向下不是完全相邻的，因为索引是随着行对每一级排序从叶级向上到根的顺序构建的。但是，您需要知道的最重要的事情是导航。假设有下面的查询：

```
SELECT e.*
FROM dbo.Employee AS e
WHERE e.EmployeeID = 27682;
```

为了查找 *EmployeeID* 为 27682（请记住这是聚集索引值）的所有数据行，SQL Server 从根页开始向下导航到叶级。根据前面显示的输出结果，根页是 *FileID* 1 中的第 234 页（由于根页是最高索引级别中[*IndexLevel*=2]的唯一页，因此您可以看到该页）。为了分析根页，我们使用输出格式为 3 的 *DBCC PAGE* 函数，同时我们要保证 SQL Server Management Studio 中的查询窗口设置为返回网格结果。原因就是使用输出格式 3 时，非叶页面的表格系列会返回给网格结果，将行从页眉中分离出来，行被返回到信息窗口中：

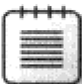
```
DBCC PAGE (IndexInternals, 1, 234, 3);
GO

RESULT:
FileId   PageId   Row      Level ChildFileId ChildPageId EmployeeID (key) KeyHashValue
------   -------  -------  ----- ----------- ----------- ---------------- ------------
1        234      0        2     1           232         NULL             NULL
1        234      1        2     1           233         12441            NULL
1        234      2        2     1           235         24881            NULL
1        234      3        2     1           236         37321            NULL
1        234      4        2     1           237         49761            NULL
1        234      5        2     1           238         62201            NULL
1        234      6        2     1           239         74641            NULL
```

查看根页的 *DBCC PAGE* 输出结果，可以在 *EmployeeID*（键）列每个“子页”的开始部分看到 *EmployeeID* 值。由于这些值是根据该级别下面的排序行决定的，因此我们只需要找到合适的范围。对于第 3 页来说，可以看到一个较小的值 24881，对于第 4 页，看到一个比较小的值 37321。因此，如果值 27682 存在，则一定位于该特殊范围所定义的索引区域。对于导航来说，我们必须使用 *FileID*（*ChildPageId*）1 中的第 235 页（*ChildFileId*）向下导航树。为了查看该页的内容，我们可以再次使用输出格式为 3 的 *DBCC PAGE* 函数：

```
DBCC PAGE (IndexInternals, 1, 235, 3);
GO
RESULT (abbreviated):
FileId  PageId Row    Level ChildFileId ChildPageId EmployeeID (key)  KeyHashValue
------- ------ ------ ----- ----------- ----------- ----------------  ------------
1        235    0      1     1           1476        24881             NULL
...
1        235    139    1     1           1615        27661             NULL
1        235    140    1     1           1616        27681             NULL
1        235    141    1     1           1617        27701             NULL
...
1        235    621    1     1           2097        3730              NULL
```

最后，如果存在一个 *EmployeeID* 为 27682 的行，则该行一定位于 *FileID* 1 的第 1616 页上。我们来看是否是这样的结果：

```
DBCC TRACEON(3604);
GO
DBCC PAGE (IndexInternals, 1, 1616, 3);
GO


...

Slot 1 Column 1 Offset 0x4 Length 4 Length (physical) 4
EmployeeID = 27682

Slot 1 Column 2 Offset 0x8 Length 60 Length (physical) 60
LastName = Arbariol

Slot 1 Column 3 Offset 0x44 Length 58 Length (physical) 58
FirstName = Burt

Slot 1 Column 4 Offset 0x7e Length 2 Length (physical) 2
MiddleInitial = R

Slot 1 Column 5 Offset 0x80 Length 11 Length (physical) 11
SSN = 373-00-8368

Slot 1 Column 6 Offset 0x8b Length 258 Length (physical) 258
OtherColumns = Junk
...
```



注意：

DBCC PAGE 返回页面的所有细节，即标题和所有数据行。在简化的输出结果中，我们只能看到从输出格式 3 得到的、我们感兴趣的 *EmployeeID*（27682）的转换行值。标题和所有其他行都被删除了。

通过遍历行的结构，我们查看了两点：索引内部及利用聚集键值找到单个数据行的过程。该方法在从一个非聚集索引中执行一次书签查找，检索建立聚集表时的数据。为了完全理解非聚集索引的使用方式，我们还需要知道非聚集索引的存储方式及遍历数据的方式。

6.5.5　非聚集索引结构

非聚集索引叶级的内容是由很多因素决定的：非聚集索引键的定义、基表的结构（堆或聚集索引）、是否存在非聚集索引功能（如包含列或筛选索引）及非聚集索引是否定义为唯一。

为了更好地理解非聚集索引，我们继续使用 *IndexInternals* 数据库。但是这一次我们查看两个表上的非聚集索引：*Employee* 表（按照 *EmployeeID* 列上的 PRIMARY KEY 约束建立聚集）和 *EmployeeHeap* 表（没有聚集索引）。*EmployeeHeap* 表实际上是 *Employee* 表的一个副本，但是，它使用 *EmployeeID* 列上的非聚集主键约束而不是聚集约束。这是我们查看的第一个结构。

1. 堆上的非聚集索引行

EmployeeHeap 表与前面示例中使用的 *Employee* 表具有完全相同的定义和数据。下面是 *EmployeeHeap* 表的定义：

```
CREATE TABLE EmployeeHeap
  (EmployeeID     INT           NOT NULL     IDENTITY,
   LastName       NCHAR(30)     NOT NULL,
   FirstName      NCHAR(29)     NOT NULL,
   MiddleInitial  NCHAR(1)      NULL,
   SSN            CHAR(11)      NOT NULL,
   OtherColumns   CHAR(258)     NOT NULL     DEFAULT 'Junk');
GO
```

与 *Employee* 表一样，*EmployeeHeap* 表中的数据行有 80 000 行，每行有 400 字节，该表还需要 4000 个数据页。为了查看数据的物理大小，可以使用本章前面介绍的 *sys.dm_db_index_physical_statsDMV*。我们可以肯定的是，该表与使用 DMV 查看页数的聚集索引叶级完全相同（就数据而言），与 *index_id* 为 0（第 3 个参数为 DMV）的索引行长度也完全相同。

```
SELECT index_depth AS D
     , index_level AS L
     , record_count AS 'Count'
     , page_count AS PgCnt
     , avg_page_space_used_in_percent AS 'PgPercentFull'
     , min_record_size_in_bytes AS 'MinLen'
     , max_record_size_in_bytes AS 'MaxLen'
     , avg_record_size_in_bytes AS 'AvgLen'
FROM sys.dm_db_index_physical_stats
      (DB_ID ('IndexInternals')
     , OBJECT_ID ('IndexInternals.dbo.EmployeeHeap')
     , 0, NULL, 'DETAILED');
GO

RESULT:
D   L   Count  PgCnt   PgPercentFull      MinLen  MaxLen  AvgLen
--- --- ------ ------- ------------------ ------- ------- ------
1   0   80000  4000    99.3081294786261   400     400     400
```

对于 *EmployeeHeap* 表来说，所有约束都是使用非聚集索引创建的。下面的命令在 *EmployeeID* 列上以非聚集索引方式创建 PRIMARY KEY，同时在 *SSN* 列上以非聚集索引方式创建一个 UNIQUE KEY 键。

```
-- Add a NONCLUSTERED PRIMARY KEY for EmployeeHeap
ALTER TABLE EmployeeHeap
   ADD CONSTRAINT EmployeeHeapPK
     PRIMARY KEY NONCLUSTERED (EmployeeID);
GO

-- Add the NONCLUSTERED UNIQUE KEY on SSN for EmployeeHeap
ALTER TABLE EmployeeHeap
   ADD CONSTRAINT SSNHeapUK
     UNIQUE NONCLUSTERED (SSN);
GO
```

为了确定在堆上构建的非聚集索引叶级中的内容，我们首先利用 DMV 查看非聚集索引的结构。对于非聚集索引来说，我们为参数 3 提供特殊的索引 ID。为了查看分配的索引 ID，对 *sys.indexes* 进行查询：

```
SELECT index_depth AS D
     , index_level AS L
     , record_count AS 'Count'
     , page_count AS PgCnt
     , avg_page_space_used_in_percent AS 'PgPercentFull'
     , min_record_size_in_bytes AS 'MinLen'
     , max_record_size_in_bytes AS 'MaxLen'
     , avg_record_size_in_bytes AS 'AvgLen'
FROM sys.dm_db_index_physical_stats
     (DB_ID ('IndexInternals')
     , OBJECT_ID ('IndexInternals.dbo.EmployeeHeap')
     , 2, NULL, 'DETAILED');
GO

RESULT:
D   L   Count   PgCnt    PgPercentFull        MinLen   MaxLen   AvgLen
--- --- ------  -------  -------------------  -------  -------  ------
2   0   80000   149      99.477291821102      13       13       13
2   1   149     1        23.9065974796145     11       11       11
```

此时，非聚集索引的叶级（级别 0）显示记录数为 80 000（根据表中有 80 000 行这一事实），以及一个最大值、最小值和平均长度 13（这些是宽度固定的索引行）。分析简单明了：非聚集索引定义在 *EmployeeID* 列上（一个 4 字节的 integer）；该表是一个堆，因此数据行的书签（物理 RID）是 8 字节。同时由于这是一个宽度固定的行并且所有行都不允许为 NULL 值，所以行的系统开销是 1 字节（4+8+1=13 字节）。为了更具体地查看存储的数据，可以使用 *DBCC IND* 查看索引的叶级页面：

```
TRUNCATE TABLE sp_tablepages;
INSERT sp_tablepages
       EXEC ('DBCC IND (IndexInternals, EmployeeHeap, 2)');
GO

SELECT IndexLevel
     , PageFID
     , PagePID
     , PrevPageFID
     , PrevPagePID
     , NextPageFID
     , NextPagePID
```

```
FROM sp_tablepages
ORDER BY IndexLevel DESC, PrevPagePID;
GO

RESULT (abbreviated):
IndexLevel  PageFID  PagePID     PrevPageFID PrevPagePID NextPageFID NextPagePID
----------  -------  ----------- ----------- ----------- ----------- -----------
1           1        8608        0           0           0            0
0           1        8544        0           0           1           8545
0           1        8545        1           8544        1           8546
...
0           1        8755        1           8754        1           8756
0           1        8756        1           8755        0           0
NULL        1        254         0           0           0           0
```

根页是 *FileID* 1 的第 8608 页。叶级页面的 *IndexLevel* 为 0，因此叶级的第一个页面是 *FileID* 为 0 的第 8544 页。为了查看该页面上的数据，使用输出格式为 3 的 *DBCC PAGE*（该叶级索引页的输出结果仅显示 539 行中的前 8 行和最后 3 行）。

```
DBCC PAGE (IndexInternals, 1, 8544, 3);
GO
RESULT (abbreviated):
FileId  PageId  Row    Level   EmployeeID (key) HEAP RID           KeyHashValue
------  ------- ------ ------  ---------------- ----------------   -------------
1       8544    0      0       1                0xF500000001000000 (010086470766)
1       8544    1      0       2                0xF500000001000100 (020068e8b274)
1       8544    2      0       3                0xF500000001000200 (03000d8f0ecc)
1       8544    3      0       4                0xF500000001000300 (0400b4b7d951)
1       8544    4      0       5                0xF500000001000400 (0500d1d065e9)
1       8544    5      0       6                0xF500000001000500 (06003f7fd0fb)
1       8544    6      0       7                0xF500000001000600 (07005a186c43)
1       8544    7      0       8                0xF500000001000700 (08000c080f1b)
...
1       8544    536    0       537              0xD211000001001000 (190098ec2ef0)
1       8544    537    0       538              0xD211000001001100 (1a0076439be2)
1       8544    538    0       539              0xD211000001001200 (1b001324275a)
```

从 *DBCC PAGE* 的输出结果中，可以看到堆上非聚集索引叶级页面有索引键列值（这里是 *EmployeeID*）及真实的数据行 RID。最后显示的值被称为 *KeyHashValue*，不真正存储在索引行中。它是在所有键列上使用哈希公式得到的固定长度字符串。该值用于表示某些其他工具中的行。其中的一个工具将在第 10 章介绍，即 *sys.dm_tran_locks DMV*，用于显示正在被使用的锁。当在某个索引行上保持一个锁时，锁列表显示 *KeyHashValue*，指示哪个键（或索引行）被锁定。

利用如下函数将 RID 转换成 *FileID:PageID:SlotNumber* 格式：

```
CREATE FUNCTION convert_RIDs (@rid BINARY(8))
    RETURNS VARCHAR(30)
AS
    BEGIN
    RETURN (
        CONVERT (VARCHAR(5),
            CONVERT(INT, SUBSTRING(@rid, 6, 1)
            + SUBSTRING(@rid, 5, 1)) )
        + ':' +
```

```
        CONVERT(VARCHAR(10),
            CONVERT(INT, SUBSTRING(@rid, 4, 1)
            + SUBSTRING(@rid, 3, 1)
            + SUBSTRING(@rid, 2, 1)
            + SUBSTRING(@rid, 1, 1)) )
        + ':' +
        CONVERT(VARCHAR(5),
            CONVERT(INT, SUBSTRING(@rid, 8, 1)
            + SUBSTRING(@rid, 7, 1)) ) )
    END;
GO
```

使用该函数可以找出某一行所在的具体行号。例如，*EmployeeID* 为 6 的行的十六进制 RID 为 0xF500000001000500：

```
SELECT dbo.convert_RIDs (0xF500000001000500);
GO

RESULT:
1:245:5
```

利用该函数将其转换成 1:245:5，这是由 *FileID* 1、*PageID* 245 和 *SlotNumber* 5 组成的。为了查看这一特殊页，使用 *DBCC PAGE* 函数然后查看槽 5 上的数据（查看这是否是 *EmployeeID* 为 6 的行）：

```
DBCC PAGE (IndexInternals, 1, 245, 3);
GO

Slot 5 Column 1 Offset 0x4 Length 4 Length (physical) 4
EmployeeID = 6

Slot 5 Column 2 Offset 0x8 Length 60 Length (physical) 60
LastName = Anderson

Slot 5 Column 3 Offset 0x44 Length 58 Length (physical) 58
FirstName = Dreaxjktgvnhye

Slot 5 Column 4 Offset 0x7e Length 2 Length (physical) 2
MiddleInitial =

Slot 5 Column 5 Offset 0x80 Length 11 Length (physical) 11
SSN = 250-07-9751

Slot 5 Column 6 Offset 0x8b Length 258 Length (physical) 258
OtherColumns = Junk
...
```

这里我们已经看到非聚集索引叶级中非聚集索引行的结构，以及利用堆的 RID 如何从非聚集索引到堆上执行书签查找。

就导航而言，假设有如下的查询：

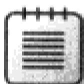
```
SELECT e.*
FROM dbo.EmployeeHeap AS e
WHERE e.EmployeeID = 27682;
```

由于该表是一个堆，因此只有非聚集索引可用于有效地导航数据。这里我们在 *EmployeeID* 上有一个非聚集索引。第一步是定位到根页（如前面的 *DBCC IND* 输出结果所示，根页是 *FileID* 1 的第 8608 页）：

```
DBCC PAGE (IndexInternals, 1, 8608, 3);
GO
RESULT:
FileId  PageId  Row     Level ChildFileId ChildPageId EmployeeID (key) KeyHashValue
------- ------  ------- ----- ----------- ----------- ---------------- -----------
1       8608    0       1     1           8544        NULL             NULL
1       8608    1       1     1           8545        540              NULL
...
1       8608    49      1     1           8593        26412            NULL
1       8608    50      1     1           8594        26951            NULL
1       8608    51      1     1           8595        27490            NULL
1       8608    52      1     1           8596        28029            NULL
1       8608    53      1     1           8597        28568            NULL
1       8608    54      1     1           8598        29107            NULL
...
1       8608    147     1     1           8755        79234            NULL
1       8608    148     1     1           8756        79773            NULL
```

利用该输出结果中的 *EmployeeID* 列，可以看到 *FileID* 1 的子页面 8595 的值为 27 490，下一页的值为 28 029。因此，如果存在 27 682 的 *EmployeeID*，则它一定位于该特殊范围所定义的索引区域中。接下来我们必须利用 *FileID*（*ChildPageId*）1 的第 8595 页（*ChildFileId*）向下导航树。为了查看该页的内容，再次使用输出样式为 3 的 *DBCC PAGE* 函数：

```
DBCC PAGE (IndexInternals, 1, 8595, 3);
GO
RESULT:
FileId  PageId       Row   Level  EmployeeID (key) HEAP RID           KeyHashValue
------  -----------  ----- ------ ---------------- ------------------ ----------------
1       8595         0     0      27490            0x1617000001000900 (6200aa3b160b)
1       8595         1     0      27491            0x1617000001000A00 (6300cf5caab3)
...
1       8595         191   0      27681            0x2017000001000000 (2100fcdaf887)
1       8595         192   0      27682            0x2017000001000100 (220012754d95)
1       8595         193   0      27683            0x2017000001000200 (23007712f12d)
...
1       8595         538  0 28028 0x3117000001000700 (7c00b4675dbf)
```



注意：

输出结果返回 539 行。在这种压缩的输出结果中，我们看到前两行、最后一行及我们感兴趣的值（27682）周围的 3 行。

从这里可以看出导航是如何继续进行的。SQL Server 将数据行的 RID 转换成 *FileID:PageID:SlotNumber* 的格式，然后继续查找堆中相应的数据行。

2. 聚集表中的非聚集索引行

对于有聚集索引的表上的非聚集索引来说，叶级行结构与堆上非聚集索引的行结构类似。非聚集索

引的叶级包含索引键和书签查找值（聚集键）。但是，如果非聚集索引键与聚集键有公共列，则 SQL Server 会在非聚集索引行中只存储公共列一次。例如，如果聚集索引键是 *EmployeeID*，同时您在 *Lastname*、*EmployeeID* 和 *SSN* 上建立了一个非聚集索引，那么索引行不会存储 *EmployeeID* 的值两次。事实上，列的数量及列的顺序是无所谓的。对于这一示例来说（让索引键较宽一般来说不是一种好习惯），假设在 *b*、*e* 和 *h* 列上定义了聚集键。下面的非聚集索引会添加这些列以构成非聚集索引叶级行（被添加到非聚集索引叶级上，列[如果有]用粗斜体显示）：

非聚集索引键	非聚集叶级行
a	a,***b***,***e***,***h***
c,h,e	c,h,e,***b***
e	e,***b***,***h***
h	h,***e***,***b***
b,c,d	b,c,d,***e***,***h***

为了查看在聚集表上创建的非聚集索引的物理结构，查看 *Employee* 表中 *SSN* 列上的 *UNIQUE* 约束：

```
-- Add the NONCLUSTERED UNIQUE KEY on SSN for Employee
ALTER TABLE Employee
    ADD CONSTRAINT EmployeeSSNUK
        UNIQUE NONCLUSTERED (SSN);
GO
```

要收集数据大小和级数信息，可使用 DMV。但是，知道参数 3 的具体索引 ID 后才可以使用 DMV。为了查看分配给非聚集索引的索引 ID，可以对 *sys.indexes* 进行查询：

```
SELECT name AS IndexName, index_id
FROM sys.indexes
WHERE [object_id] = OBJECT_ID ('Employee');
GO

RESULT:
IndexName          index_id
----------------   --------
EmployeePK         1
EmployeeSSNUK      2

SELECT index_depth AS D
    , index_level AS L
    , record_count AS 'Count'
    , page_count AS PgCnt
    , avg_page_space_used_in_percent AS 'PgPercentFull'
    , min_record_size_in_bytes AS 'MinLen'
    , max_record_size_in_bytes AS 'MaxLen'
    , avg_record_size_in_bytes AS 'AvgLen'
FROM sys.dm_db_index_physical_stats
     (DB_ID ('IndexInternals')
    , OBJECT_ID ('IndexInternals.dbo.Employee')
    , 2, NULL, 'DETAILED');
GO
```

```
RESULT:
D L          Count    PgCnt   PgPercentFull    MinLen   MaxLen  AvgLen
-- -------   -------- ------- ---------------- -------- ------- ------
2  0         80000    179     99.3661106992834 16       16      16
2  1         179      1       44.2055843834939 18       18      18
```

这里非聚集索引的叶级（级别0）显示一个记录数80 000（表中有80 000行），以及一个最大值、最小值和平均长度16（是宽度固定的索引行）。分析简单明了：非聚集索引定义在*SSN*列（一个宽度固定的11字节字符列）上，表有一个聚集键*EmployeeID*，因此数据行的书签（聚集键）是4字节，同时由于该行是一个宽度固定的行而且所有列都不能为NULL值，因此行的系统开销是1字节（11+4+1=16字节）。为了更具体地查看存储的数据，可以利用*DBCC IND*查看该索引的叶级页面：

```
TRUNCATE TABLE sp_tablepages;
INSERT sp_tablepages
        EXEC ('DBCC IND (IndexInternals, Employee, 2)');
GO

SELECT IndexLevel
    , PageFID
    , PagePID
    , PrevPageFID
    , PrevPagePID
    , NextPageFID
    , NextPagePID
FROM sp_tablepages
ORDER BY IndexLevel DESC, PrevPagePID;
GO

RESULT (abbreviated):
IndexLevel PageFID PagePID     PrevPageFID PrevPagePID NextPageFID NextPagePID
---------- ------- ----------- ----------- ----------- ----------- -----------
1          1       4328        0           0           0           0
0          1       4264        0           0           1           4265
0          1       4265        1           4264        1           4266
...
0          1       4505        1           4504        1           4506
0          1       4506        1           4505        0           0
NULL       1       158         0           0           0           0
```

FileId 1的第4328页是根页。叶级页的*IndexLevel*为0，因此叶级的第1页在*FileID* 1的第4264页。为了查看该页面上的数据，可以使用格式3的*DBCC PAGE*函数：

```
DBCC PAGE (IndexInternals, 1, 4264, 3);
GO
RESULT (abbreviated):
FileId  PageId      Row     Level   SSN (key)     EmployeeID   KeyHashValue
------  ----------- ------  ------  -----------   -----------  ----------------
1       4264        0       0       000-00-0184   31101        (fd00604642ee)
1       4264        1       0       000-00-0236   22669        (fb00de40fee1)
1       4264        2       0       000-00-0395   18705        (0101d993da83)
...
1       4264        446     0       013-00-5906   44969        (ff00355b1727)
1       4264        447     0       013-00-5982   7176         (03012415a3e8)
```

```
1        4264         448       0        013-00-6001  11932          (f100f75a17a4)
```

从 *DBCC PAGE* 的输出结果中可以看到，聚集表上非聚集索引的叶级页面具有索引键（这里是 *SSN* 列）和数据行书签的实际列值，这里是 *EmployeeID*。这是一个实际值，被复制到非聚集索引的叶级中。如果聚集键更宽，则非聚集索引的叶级也会更宽。

就导航而言，查看如下查询：

```
SELECT e.*
FROM dbo.Employee AS e
WHERE e.SSN = '123-45-6789';
```

为了查找 *SSN* 为 123-45-6789 的所有数据行，SQL Server 从根页开始向下定位到叶级。根据前面显示的输出结果可以看出，根页是 FileID 1 的 4328 页（您可以看到这样的结果是因为根页是最高索引级别 [*IndexLevel*=1] 上的唯一一页）。我们可以像以前那样执行同样的分析并导航整个 B 树，这作为一项练习留给您去完成（如果您感兴趣的话）。

3. 非唯一非聚集索引行

您现在知道非聚集索引的叶级必须有一个书签，因为您希望能够从叶级找到实际的数据行。非聚集索引的非叶级只需要帮助我们向下遍历到更低级的页面。就唯一非聚集索引（如前面实例中的 PRIMARY KEY 和 UNIQUE 约束索引）而言，非叶级行仅包含非聚集索引键值和子页指针。但是，如果索引不是唯一的，则非叶级行包含非聚集索引键值、子页指针和书签值。换言之，书签值被添加到一个非唯一且非聚集的索引中的非聚集索引键上，以保证唯一性（因为按照定义，书签必须是唯一的）。

请记住，为了创建索引行，SQL Server 不关心非唯一索引中的键是否真正包含副本。如果索引没有定义为唯一的，则即使所有值都是唯一的，非页索引行也总会包含书签。

可以通过创建如下的三个索引查看它们的叶级和非叶级行大小，从而验证上面的说法：

```
CREATE NONCLUSTERED INDEX TestTreeStructure
ON Employee (SSN);
GO

CREATE UNIQUE NONCLUSTERED INDEX TestTreeStructureUnique1
ON Employee (SSN);
GO

CREATE UNIQUE NONCLUSTERED INDEX TestTreeStructureUnique2
ON Employee (SSN, EmployeeID);
GO

SELECT si.[name] AS iname
     , index_depth AS D
     , index_level AS L
     , record_count AS 'Count'
     , page_count AS PgCnt
     , avg_page_space_used_in_percent AS 'PgPercentFull'
     , min_record_size_in_bytes AS 'MinLen'
     , max_record_size_in_bytes AS 'MaxLen'
     , avg_record_size_in_bytes AS 'AvgLen'
FROM sys.dm_db_index_physical_stats
     (DB_ID ('IndexInternals')
```

```
    , OBJECT_ID ('IndexInternals.dbo.Employee')
    , NULL, NULL, 'DETAILED') ps
    INNER JOIN sys.indexes si
        ON ps.[object_id] = si.[object_id]
            AND ps.[index_id] = si.[index_id]
    WHERE ps.[index_id] > 2;
GO

RESULT:
iname                   D   L   Count PgCnt PgPercentFull    MinLen MaxLen AvgLen
----------------------- -   -   ----- ----- ---------------- ------ ------ ------
TestTreeStructure       2   0   80000 179    99.3661106992834 16     16     16
TestTreeStructure       2   1   179   1     53.0516431924883 22     22     22
TestTreeStructureUnique1 2  0   80000 179    99.3661106992834 16     16     16
TestTreeStructureUnique1 2  1   179   1     44.2055843834939 18     18     18
TestTreeStructureUnique2 2  0   80000 179    99.3661106992834 16     16     16
TestTreeStructureUnique2 2  1   179   1     53.0516431924883 22     22     22
```

注意这三个索引中的叶级(级别 1)在所有列上都是一致的：*Count(record_count)*、*PgCnt (page_count)*、*PgPercentFull (avg_space_used_in_percent)*和所有三个长度列。对于索引的非叶级（非常小）来说，您可以看到长度是变化的，第一列(*TestTreeStructure*)和第三列(*TestTreeStructureUnique2*)的非叶级完全相同。第一个索引添加了 *EmployeeID*，因为该列是聚集键（因此是书签）。第三个索引中已经包含 *EmployeeID*，因此无需再添加。但是，在第一个索引中，由于没有作为唯一性索引定义，因此 SQL Server 一直沿树向上添加聚集键。对于第二个索引(只有在 *SSN* 上唯一)来说，SQL Server 不一直沿树向上包括 *EmployeeID*。如果您感兴趣，可以利用 *DBCC IND* 和 *DBCC PAGE* 继续分析这些结构来进一步查看物理上的行结构。

4. 具有包含性列的非聚集索引行（使用 *INCLUDE*）

到目前为止的所有非聚集索引中，我们所关注的是通过约束创建的索引的物理特性或用于测试物理结构的索引。在所有地方我们都没有达到索引键大小的极限，即 900 字节或 16 列，不论先出现哪一种情况。这些限制存在的原因是保证索引树的可伸缩性。但是，这也限制了可以被索引的列的最大数量。

在某些情况下，在索引中添加列能够使 SQL Server 在访问范围查询数据时删除书签查找，这称为*覆盖索引*。一个覆盖索引是一个非聚集索引，其中所有要满足某个查询的信息都能在叶级上找到，因此 SQL Server 根本就不必访问数据页。这对于优化某些较复杂的基于范围的查询来说是一种有效的工具。

不是向非聚集索引键添加列或使树更深，一个覆盖索引包含的列可以在不成为键的一部分的情况下利用 *INCLUDE* 语法添加到索引行。这是对 *CREATE INDEX* 命令的一种简单添加：

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX index_name
   ON table_name (column_name [ASC | DESC][,...n])
      [ INCLUDE ( column_name [ ,...n ] ) ]
```

在关键字 *INCLUDE* 之后列出的这些列允许您在非聚集索引的叶级上不受 900 字节或 16 键列的限制。包含性列仅出现在叶级上，不会以任何方式影响索引行的排列顺序。在某些情况下，SQL Server 可能会自动向索引添加包含性列，如在一个分区表上创建索引并且没有 *ON* 文件组或 *ON partition_scheme_name* 被指定的情况下。

5. 具有筛选器的非聚集索引行（筛选索引）

在不使用筛选器的情况下，非聚集索引的叶级为表中每一行数据包含一个索引行，按照索引定义的逻辑顺序。在 SQL Server 2008 中，您可以向非聚集索引定义添加一个筛选谓词，从而使 SQL Server 值为与谓词匹配的数据创建非聚集索引行，来限制非聚集索引的大小。如果遇到如下情形之一，这一功能可能非常有用。

- 当一列包含大部分 NULL 值并且查询仅检索数据为非空的行时。与 *SPARSE* 列结合使用尤为有用。
- 当一列仅包含有限个您感兴趣的值或者您希望为一组值强制唯一性时。例如，如果您希望 *Employee* 表的 *SSN* 列允许 NULL 时会怎样？使用一种约束，SQL Server 仅允许一行为空。但是，使用一个筛选索引可以仅在 *SSN* 不为空的行上创建唯一索引。语法如下：

```
CREATE UNIQUE NONCLUSTERED INDEX SSN_NOT_NULLs
ON Employee (SSN)
WHERE SSN IS NOT NULL;
```

- 当查询仅检索特定范围内的数据并且您希望在这些数据上（而不是整个表上）添加索引时。例如，您有一个表是按照月进行分割的并且包含 3 年的有效数据（2008、2007 和 2006），同时一个小组希望大量分析 2007 年第 4 季度的数据。您不用为所有数据创建非聚集索引，可以针对如下条件创建索引（也可以使用 *INCLUDE*）：

```
WHERE SalesDate > '20071001' AND SalesDate < '20080101';
```

利用筛选器创建的索引的最终结果是：非聚集索引的叶级只有在某一行满足筛选器的定义时才包含这一行。同时定义筛选器的列不需要包含在键中，甚至不需要包含在包含性列中。但是，这可以帮助使索引对于某些查询来说更有效。您可以利用前面介绍过的 *DBCC IND*、*DBCC PAGE* 和 *DMV* 查看具有筛选器的索引的大小和结构。

6.6 特殊索引结构

SQL Server 2008 允许您创建几种特殊类型的索引：计算列上的索引、视图上的索引、空间索引、全文索引及 XML 索引。这部分内容包括创建这些索引的需求及结构上的差异。

6.6.1 计算列上的索引和索引视图

在没有索引的情况下，其中的一些结构（计算列和视图）完全是逻辑上的。不会对包含的数据进行物理存储。一个计算列不是用表存储数据的，而是在每次行被访问时进行重新计算（除非计算列标记为 PERSISTED）。一个视图不保存任何数据，基本上只保存一条 *SELECT* 语句，该语句在视图中的数据每次被访问时执行。利用这些特殊的索引，SQL Server 实际上将逻辑上的数据物化成索引的物理叶级。

1. 必备条件

在计算列或视图上创建索引必须满足一定的先决条件。最大的问题是 SQL Server 必须能够保证如果基表数据相同，则总会为所有计算列或视图中的行返回相同的值（即计算列和视图是*决定性的*）。为了保

证总能生成相同的值，这些特殊的索引有 3 种类型的需求。首先，很多会话级选项必须设置为特定值。其次，对计算列或视图定义中可以使用的函数有一些限制。第 3 种需求（仅应用到被索引的视图）是视图所依据的表必须符合一定的标准。

2. SET 选项

下面的 7 个 SET 选项可以影响一个表达式或谓词的结果值，因此您必须按照下面的语句进行设置，从而在计算列上创建被索引的视图或索引：

```
SET CONCAT_NULL_YIELDS_NULL ON
SET QUOTED_IDENTIFIER ON
SET ANSI_NULLS ON
SET ANSI_PADDING ON
SET ANSI_WARNINGS ON
SET NUMERIC_ROUNDABORT OFF
```

注意，除 NUMERIC_ROUNDABORT 选项（该选项设置为 OFF）外其他所有选项都必须设置为 ON。从技术上来说，ARITHABORT 选项也必须设置为 ON。同时当数据库设置为 90 兼容模式或更高时，将 ANSI_WARNINGS 设置为 ON 会自动将 ARITHABORT 也设置为 ON，因此您不需要单独设置 ARITHABORT。如果其中的任意选项没有设置为指定值，则当您试图创建一个特殊的索引时会收到一条错误消息。此外，如果您已经创建了其中的一个索引，然后又修改了 SET 选项的设置并试图修改视图所依赖的计算列或视图，则会出现错误。如果您发出一条通常需要使用索引的 *SELECT* 语句并且 SET 选项没有指示的值时，则索引将被忽略，但不会产生错误。

有多种方法可以确定创建其中的一种特殊索引之前 SET（设置）选项是否设置了适当的值。可以利用 *SESSIONPROPERTY* 函数测试当前连接的设置。返回值为 1 表示设置为 ON，0 表示设置为 OFF。下面的示例用于检查 NUMERIC_ROUNDABORT 选项的当前会话设置：

```
SELECT SESSIONPROPERTY ('NUMERIC_ROUNDABORT');
```

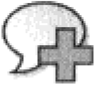
您也可以选择使用 *sys.dm_exec_sessions* DMV 来检查任意连接的 SET 选项。下面的查询返回前面介绍的 6 种 SET 选项中的 5 个的当前会话值：

```
SELECT quoted_identifier, arithabort, ansi_warnings,
       ansi_padding, ansi_nulls, concat_null_yields_null
FROM sys.dm_exec_sessions
WHERE session_id = @@spid;
```

遗憾的是，NUMERIC_ROUNDABORT 不包括在 *sys.dm_exec_sessions* DMV 结果中。没有办法查看当前连接之外的其他连接的设置值。

3. 允许函数

函数要么是*确定性的*要么是*非确定性的*。如果每次用相同的输入值调用函数时函数返回相同的结果，则该函数是确定性的。如果使用相同的输入值调用函数时返回的结果不同，则该函数是非确定性的。对于索引而言，如果所有 SET 选项具有所需的设置时对于相同的输入值函数总会返回相同的结果，则该函数被认为是确定性的。在计算列的定义中使用或在某个索引视图的 *SELECT* 列表或 WHERE 子句中使用的任何函数一定是确定性的。



更多信息：

SQL Server 联机丛书包含一个说明哪些函数是确定性的、哪些函数是非确定性的完整列表。根据它们的使用方式，一些函数可以是确定性的也可以是非确定性的，*SQL Server* 联机丛书也对这些函数进行了介绍。

看起来好像非确定性函数的列表非常严格，但是 SQL Server 必须能够保证存储在索引中的值不变。在某些情况下，这些限制条件可能是过度谨慎了，但是如果不非常小心，被索引的视图或计算列上的索引可能没有意义。同样的限制条件适用于在您自己的用户定义函数（UDF）中使用的函数上，即您自己的函数不能建立在任何非确定性的内置函数上。可以利用 *OBJECTPROPERTY* 函数验证任何函数的确定性属性：

```
SELECT OBJECTPROPERTY (object_id('<function_name>'), 'IsDeterministic')
```

即使某个函数是确定性的，如果它包含 *float* 或 *real* 表达式，则函数的结果也可能随着处理器架构或微代码版本的不同而变化。因此包含 *float* 或 *real* 数据类型值的表达式或函数被认为是不确定的。为了保证从某台机器向另一台机器移动一个数据库（通过分离和附加或通过执行备份和还原）时值保持不变，如果不确定的值被物理存储在数据库中并且没有被重新计算时，它们只能在索引键列中使用。如果不确定值是表中存储列的值或者是被标记为持久的计算列，则可以被使用。我们将在接下来的“计算列上的索引”部分进一步介绍持久列。

4. 架构绑定

要创建索引视图，对表本身的要求是所有基础对象架构的定义不能更改。为了防止对架构定义进行更改，*CREATE VIEW* 语句允许使用 WITH SCHEMABINDING 选项。当您指定 WITH SCHEMABINDING 时，定义视图的 *SELECT* 语句必须包括所有被参照表的两部分名称（*schema.object*）。您不能删除表或修改使用 WITH SCHEMABINDING 子句创建的视图中的列，除非您已经删除了该视图或修改视图使其不再是绑定架构，否则 SQL Server 会出现错误。如果视图所依据的表归创建视图的用户之外的其他用户所有，则视图创建者不会自动具有创建架构绑定视图的权限，因为这样将限制表的所有者对自己的表进行修改。一个用户必须被授予表的 REFERENCES 权限才能在该表上创建一个具有架构绑定的视图。我们一会儿将看到架构绑定的示例。

5. 计算列上的索引

SQL Server 2008 允许您在确定、精确的（和持久的、不精确）计算列上建立索引，否则列的结果数据类型将是可索引的。这说明列的数据类型不能是 LOB 数据类型（如 *text*、*varchar(max)*或 *XML*）。这样的一个计算列可以是一个索引键、包含性列、PRIMARY KEY 和 UNIQUE 约束的一部分。您不能在计算列上定义 FOREIGN KEY、CHECK 或 DEAFAULT 约束，同时计算列总被认为可以为空，除非您在 *ISNULL* 函数中括上表达式。当您在计算列上创建一个索引时，前面提到的 6 种 SET 选项必须首先具有正确的值。

下面是一个示例：

```
CREATE TABLE t1 (a INT, b as 2*a);
GO
```

```
CREATE INDEX i1 ON t1 (b);
GO
```

如果创建表时 SET 选项的任意值不正确，则会在创建索引时收到如下消息：

```
Server: Msg 1935, Level 16, State 1, Line 2
Cannot create index. Object '<tname>' was created with the following SET options off:
    '<option(s)>'.
```

如果有多个选项的值不正确，则错误消息会报告所有不正确的选项。

下面是创建一个具有非确定性计算列的表的一个示例：

```
CREATE TABLE t2 (a INT, b DATETIME, c AS DATENAME(MM, b));
GO
CREATE INDEX i2 ON t2 (c);
GO
```

当您在计算列 *c* 上创建索引时，会得到如下错误：

```
Msg 2729, Level 16, State 1, Line 1
Column 'c' in table 't2' cannot be used in an index or statistics or as a partition key
    because it is nondeterministic.
```

c 列是非确定性的，因为 *DATENAME()*的月份值根据您使用的语言的不同而不同。

使用 *COLUMNPROPERTY* 函数。您可以在某个计算列上创建索引之前利用 *IsDeterministic* 列属性确定该列是否是确定性的。如果您指定该属性，则如果列是确定性的，那么 *COLUMNPROPERTY* 函数将返回 1，否则返回 0。对于既不是计算列也不是视图中的列来说，结果是不确定的，因此您应该考虑在检查 *IsDeterministic* 属性之前检查 *IsComputed* 属性。下面的示例检测到前面示例中表 *t2* 中的 *c* 列是非确定性的：

```
SELECT COLUMNPROPERTY (OBJECT_ID('t2'), 'c', 'IsDeterministic');
```

返回值 0，这表示 *c* 列是非确定性的。注意，*COLUMNPROPERTY* 函数的第一个参数需要一个对象 ID，第二个参数需要一个列名称。

但是，*COLUMNPROPERTY* 函数也有一个 *IsIndexable* 属性。这可能是快速检查的最简单方法，但是如果列不是可索引的，则该函数不会为您提供原因。因此，您应该检查其他属性。

6. 计算列的实现

如果在一个计算列上创建一个聚集索引，则此计算列在表中不再是虚列。计算值将物理存在于表的行中，是聚集索引的叶级。对计算列所依据的列的更新会同时更新表本身的计算列。例如，在前面创建的 *t1* 表中，如果在列 *a* 中插入值为 10 的一行，则该行会根据实际数据行中的值 10 和 20 创建。如果接下来将 10 更新为 15，则第二列会自动更新为 30。

持久列。将某个计算列标志为 PERSISTED（SQL Server 2005 中引入的一种功能）将允许在表中存储计算值，即使在建立索引之前。实际上，在产品中添加该功能是为了允许在基础表 *float* 或 *real* 类型的列的计算值列上建立索引。当您希望在这样的列上建立索引时，可以选择删除和重新建立基础列，在一个大型表中，基础列可能占用很大的系统开销。

下面是一个示例。在 *Northwind* 数据库中，*Order Details* 表有一个被称为 *Discount* 的 *real* 类型。下面的代码添加一个名为 *Final* 的计算列，显示打折后每一项的总价。在 *Final* 上建立索引的语句失败的原

因是：包含 *real* 值的结果列是不确定和不持久的。

```
USE Northwind;
GO
ALTER TABLE [Order Details]
   ADD Final AS
     (Quantity * UnitPrice) - Discount * (Quantity * UnitPrice);
GO

CREATE INDEX OD_Final_Index on [Order Details](Final);
GO

Error Message:
Msg 2799, Level 16, State 1, Line 1
Cannot create index or statistics 'OD_Final_Index' on table 'Order Details'
    because the computed column 'Final' is imprecise and not persisted. Consider removing
    column from index or statistics key or marking computed column persisted.
```

在没有持久计算列的情况下，在包含最终价格的计算列上创建索引的唯一方法是从表中删除 *Discount* 列，然后重新定义该列。*Discount* 上现有的索引也必须被删除，然后重建。在具有持久计算列的情况下，您所要做的是删除计算列（这是一种只针对元数据的操作），然后将其重新定义为一个持久计算列。接下来便可以在该列上创建索引：

```
ALTER TABLE [Order Details]
   DROP COLUMN Final;
GO
ALTER TABLE [Order Details]
   ADD Final AS
   (Quantity * UnitPrice) - Discount * (Quantity * UnitPrice) PERSISTED;
GO
CREATE INDEX OD_Final_Index on [Order Details](Final);
```

当您确定是否必须使用 PERSISTED 选项的时候，请使用 *COLUMNPROPERTY* 函数和 *IsPrecise* 属性来确定某个确定性列是否是精确的：

```
SELECT COLUMNPROPERTY (OBJECT_ID ('Order Details'), 'Final', 'IsPrecise');
```

当定义分区时，也可以使用持久性计算列。用做分区列的计算列必须明确标记为 PERSISTED，不管该列是精确的还是不精确的。我们将在第 7 章介绍分区。

7. 索引视图

SQL Server 中的索引视图与被称为 *materialized views* 的产品类似。索引视图最重要的好处是能够物化大型表的概要聚合。例如，有一个顾客表包含数百万行美国顾客的信息，您希望从中获取每个州的顾客信息。您可以利用 *GROUP BY* 查询创建一个视图，按照州进行分组并且包含每个州的订货数量。标准视图只是命名的存储查询，不存储结果。每次视图被引用时，产生分组结果的聚合必须被重新计算。当您在该视图上创建一个索引时，聚合的数据将被存储在索引的叶级上。因此索引视图中只有 50 行（每个州一行），而不是几百万行的顾客信息。接下来聚合报告查询可以利用索引视图进行处理，不必扫描基础大型表。在视图上建立的第一个索引一定是一个聚集索引，同时由于聚集索引包含其叶级上的所有数据，因此该索引实际上执行视图的物化。视图的数据物理存储在聚集索引的叶级上。

8. 其他要求

除了要求视图中使用的所有函数都必须是确定性的之外，所需的SET选项也必须设置为适当的值，视图定义不能包含下面的内容。

- *TOP*
- LOB列
- *DISTINCT*
- *MIN*、*MAX*、*COUNT(*)*、*COUNT(<expression>)*、*STDEV*、*VARIANCE*、*AVG*
- 对可为空的表达式*求和*（*SUM*）
- 一个已废弃的表
- *ROWSET*函数
- 另一个视图（您只能引用基表）
- *UNION*
- 子查询、OUTER连接或自身连接
- 全文谓词（*CONTAINS*、*FREETEXT*）
- *COMPUTE*、*COMPUTE BY*
- *ORDER BY*

同样，如果视图定义包含*GROUP BY*，则*SELECT*列表一定包括聚合*COUNT_BIG (*)*。*COUNT_BIG*返回一个*BIGINT*，这是一个8字节integer。包含*GROUP BY*的视图不能包含*HAVING*、*CUBE*、*ROLLUP*或*GROUP BY ALL*。同时，所有*GROUP BY*列必须出现在SELECT列表中。注意，如果视图同时包含*SUM*和*COUNT_BIG (*)*，则可以计算*AVG*聚合函数的等价值，即使在索引视图中不允许使用*AVG*。虽然这些限制看起来很严格，但是请记住，它们是应用到视图定义中的，而不是应用到可能使用该索引视图的查询中。

为了验证您已经满足了所有要求，可以使用*OBJECTPROPERTY*函数的*IsIndexable*属性。下面的查询告诉您是否可以在一个名为*Product Totals*的视图上建立索引：

```
SELECT OBJECTPROPERTY (OBJECT_ID ('Product_Totals'), 'IsIndexable');
```

返回值为1表示已经满足所有要求并且能够在视图上建立索引。

9. 创建一个索引视图

在视图上建立索引的第一步是创建视图本身。下面是*AdventureWorks2008*数据库中的一个示例：

```
USE AdventureWorks2008;
GO
CREATE VIEW Vdiscount1
WITH SCHEMABINDING
AS SELECT SUM (UnitPrice*OrderQty) AS SumPrice
    , SUM (UnitPrice * OrderQty * (1.00 - UnitPriceDiscount))
          AS SumDiscountPrice
    , COUNT_BIG (*) AS Count
    , ProductID
FROM Sales.SalesOrderDetail
GROUP BY ProductID;
```

注意WITH SCHEMABINDING子句及表架构名称（*Sales*）的规范性。在这一点上，我们有一个规

范视图——一个不占用存储空间的存储 *SELECT* 语句。实际上，如果我们查看 *sys.dm_db_partition_stats* 中该视图的数据，会发现不返回任何行：

```
SELECT si.name AS index_name,
     ps.used_page_count, ps.reserved_page_count, ps.row_count
FROM sys.dm_db_partition_stats AS ps
   JOIN sys.indexes AS si
      ON ps.[object_id] = si.[object_id]
WHERE ps.[object_id] = OBJECT_ID ('dbo.Vdiscount1');
```

为了创建一个索引视图，必须首先创建一个*唯一性聚集索引*。视图上的聚集索引包含构成该视图定义的所有数据。该语句为视图定义一个唯一聚集索引：

```
CREATE UNIQUE CLUSTERED INDEX VDiscount_Idx ON Vdiscount1 (ProductID);
```

创建索引视图后，就可以重新运行前面的 *SELECT* 语句来查看被视图上索引物化的页面了。

```
RESULT:
index_name       used_page_count       reserved_page_count   row_count
--------------   -------------------   -------------------   ---------
VDiscountIdx     4                     4                     266
```

包含索引视图的数据是持久性的，同时索引视图存储聚集索引叶级上的数据。您可以利用临时表构建类似的内容来存储自己感兴趣的数据。但是临时表是静态的，不反映基础表的变化。相比而言，SQL Server 自动维护索引视图，会在有人修改影响视图的数据时更新聚集索引中存储的信息。

当您创建唯一性聚集索引之后，可以在视图上创建多个非聚集索引。可以利用 *OBJECTPROPERTY* 函数的 *IsIndexes* 属性确定某个视图是否建立了索引。对于 *Vdiscount1* 索引视图来说，下面的语句将返回 1，表示该视图建立了索引：

```
SELECT OBJECTPROPERTY (OBJECT_ID ('Vdiscount1'), 'IsIndexed');
```

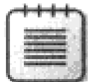
视图建立索引后，有关空间使用率和位置的元数据可以通过目录视图访问，这一点与其他索引一样。

10. 使用索引视图

索引视图最有意义的一个优点是查询不必直接引用某个视图来使用视图上的索引。考虑 *Vdiscount*1 索引视图。假设您发出如下 *SELECT* 语句：

```
SELECT ProductID, total_sales = SUM (UnitPrice * OrderQty)
FROM Sales.SalesOr
```

查询优化器发现每个 *ProductID* 的所有 *UnitPrice*OrderQty* 值的预计算和已经在 *Vdiscount1* 视图的索引中可用。查询优化器估算使用该索引视图处理查询的代价，同时索引视图很可能用于访问满足这一查询所需的信息——*Sales.SalesOrderDetail* 表可能根本不会被涉及。



注意：

虽然您可以在 SQL Server 2008 的任何版本中创建索引视图，但是对于查询优化器来说，为了在查询中没有引用索引视图的情况下使用索引视图，SQL Server 2008 的引擎版本一定要是企业版、开发人员版或评估版。

您有一个索引视图并不表示查询优化器总是为查询执行计划选择索引视图。实际上，即使在 FROM 子句中直接引用索引视图，查询优化器也会直接访问基表。为了保证 FROM 子句中的索引视图没有扩展基础 *SELECT* 语句，可以使用 FROM 子句中的 NOEXPAND 提示。索引选择、查询优化和索引视图用途的一些内部信息将在第 8 章进行更详细的介绍。

6.6.2 全文索引

全文索引是支持全文搜索功能（能够有效搜索表中的字符和二进制列）的特殊索引。创建和使用全文索引的具体内容不是本书讨论的范围，*SQL Server 联机丛书*中有一节全面介绍了全文索引。

全文索引是为了方便起见而存储在数据库内部表中的反转、堆叠和压缩索引。全文索引数据存储在内部表中的标准索引行中，但是大部分行的内容都是不透明的，全文引擎本身（*DBCC PAGE* 等工具不能正确地分开行中的所有字段）除外。

全文索引的存储与标准索引的存储相同，只是前者作为内部表存储，查找它们结构的常规方法不起作用。例如，*AdventureWorks2008* 数据库中的 *HumanResources.JobCandidate* 表拥有一个全文索引。为了查找内部表（存储全文索引的表）的对象 ID，下面的 T-SQL 语句可以用于查询 *sys.internal*_tables 目录视图：

```
USE AdventureWorks2008;
GO
SELECT [name], [object_id] FROM sys.internal_tables
WHERE parent_object_id = OBJECT_ID ('HumanResources.JobCandidate');
GO

RESULT:
name                                              object_id
------------------------------------------------  -----------
fulltext_index_docidstatus_1333579789             2046630334
fulltext_docidfilter_1333579789                   2062630391
fulltext_indexeddocid_1333579789                  2078630448
fulltext_avdl_1333579789                          2094630505
```

接下来可以使用检查索引结构的常规方法，即使用从 *sys.internal_tables* 中返回的对象 ID。同样的方法对于接下来要介绍的空间索引和 XML 索引来说也适用。

正如您所看到的那样，内部表与常规表和索引相比在系统目录中具有不同的根，不过对它们空间使用率的跟踪方式完全相同，而且它们的结构也与常规索引相同。*SQL Server 联机丛书*“内部表”一节包含了对它们的详细解释，包括相关系统目录的一个实体关系图，以及查看它们信息的各种查询。

6.6.3 空间索引

一个空间索引在表的一个空间数据类型列中包含所有值的分解视图。分解的值用于在空间比较操作期间对匹配值进行模糊修剪。就全文搜索而言，创建和使用空间索引的具体内容不是本书讨论的范围，*SQL Server 联机丛书*对此进行了详细的介绍，可参阅“空间索引概述”这一主题。

空间索引是作为内部表存储的聚集索引。除存储分解空间值之外，空间索引与常规索引具有完全相同的结构。

6.6.4 XML 索引

XML 索引通过存储 XML 数据的碎片表示法（可以利用常规的 B 树方法搜索，不必遍历整个[可能

很大的]XML BLOB）为搜索 XML BLOB 值提供一种有效的机制。与全文和空间索引一样，创建和使用 XML 索引的具体内容不是本书讨论的范围，*SQL Server 联机丛书*中对这一内容进行了详细的介绍，可参阅“XML 数据类型上的索引”这一主题。

有两种类型的 XML 索引：主 XML 索引和辅助 XML 索引。主 XML 索引是对 XML 列中被索引的每个值的一种碎片表示法，XML BLOB 中每个节点对应一行。一个主 XML 索引是一个聚集索引并作为内部表进行存储。辅助 XML 索引是主 XML 索引上的一个非聚集索引，并且提供与标准非聚集索引相同的功能；是利用不同的排序顺序的另一种数据访问路径。这些索引的内部结构与标准索引的内部结构相同。

6.7 数据修改的内部

我们已经了解了 SQL Server 存储数据和索引信息的方式。现在我们来看一下数据被修改时 SQL Server 在系统内部实际执行的工作。我们已经看到聚集索引是如何定义数据逻辑顺序的，同时一个堆只不过是未排序页面的一个集合。我们已经看到非聚集索引是如何独立于数据存储的，以及数据是如何成为实际表中数据的副本的（根据索引的定义）。一般来说，您应该在表上建立一个聚集索引。SQL 客户咨询小组在 2007 年年中发布了一本白皮书，对各种表结构进行了比较并且支持这种观点。请参阅 http://www.microsoft.com/technet/prodtechnol/sql/bestpractice/clusivsh.mspx。在这一部分，我们将查看 SQL Server 在处理数据修改语句时是如何处理索引存在的问题的。

注意，对于表上的每个 *INSERT*、*UPDATE* 和 *DELETE* 操作来说，总会对表上的每个非聚集索引进行等价的操作。这一部分介绍的机制对聚集索引和非聚集索引的作用相同。对表所做的任何修改都会首先反映到堆或聚集索引上，接下来反映到每个非聚集索引上。

在 SQL Server 2008 中，这一规则的一种例外情况是筛选索引，这时筛选谓词表示筛选非聚集索引可能没有行与正在被修改的表中的行相匹配。当对表进行修改时，将计算筛选索引谓词以确定是否需要将相同的操作应用到筛选非聚集索引上。

6.7.1 插入行

向表中插入一个新行时，SQL Server 必须确定数据的插入位置，同时还要将相应的行插入到每个非聚集索引中。每个操作都遵循相同的模式：修改合适的数据页（根据表是否有聚集索引确定），然后将相应的索引行插入到每个非聚集索引的叶级上。

当表没有聚集索引时（即该表是一个堆），新行总是被插入到表中任意可用的空间上。在第 3 章中，您已经学习了 IAM 如何跟踪文件中的哪段区间已经属于某个表，同时在第 5 章中，您已经看到 PFS 页面是如何表示这些区间内的哪些页面具有可用空间的。如果所有页面都没有可用空间，SQL Server 会从已经属于该对象的现有统一区间内查找未分配的页面。如果不存在未分配的页面，则 SQL Server 必须为表分配一整段新的区间。第 3 章讨论过如何利用全局分配表（GAM）和共享全局映射表（SGAM）查找将分配给某个对象的可用区间。因此，虽然使用 PFS 和 IAM 定位执行 *INSERT* 的空间相对来说比较有效，但是由于行的位置（对于 *INSERT*）没有定义，因此确定在堆中哪里放置行通常比在有聚集索引的表中放置行效率低。

对于向有聚集索引的表中执行插入和被插入到非聚集索引的索引行来说，行（不管是数据行还是索引行）在索引中被插入的位置是固定的，根据新行在索引键列上的值决定。在新行是一个 *INSERT* 的直接结果，或使行移动或索引键列更改 UPDATE 语句的结果时出现插入。当一行必须移动到一个

新页时，*UPDATE* 语句在内部利用一个 DELETE 语句后跟一个 *INSERT* 语句来执行（*删除/插入策略*）。新行根据它们的索引键位置进行插入，同时如果当前叶级（如果这是一个聚集索引，则是一个数据页；如果这是一个非聚集索引，则是一个索引页）没有空间，则 SQL Server 会通过页面拆分分离新页。由于索引为索引页级中的行指定了一种特殊的顺序，因此每一个新行都有自己所属的特殊位置。如果新行所属的页面上没有空间，则必须分配一个新页面并将该页面连接到 B 树上。如果可能的话，这个新页面会从它所连接的其他页面的同一区间上分配。如果区间已满（通常都会出现这种情况），则为对象分配一个新的区间（8 页或 64KB）。正如在第 3 章介绍的那样，SQL Server 利用 GAM 页查找可用的区间。

6.7.2　拆分页

当 SQL Server 找到新页面后，原始页面必须被拆分；一半的行数（页面槽阵列上的前半部分）被留在了原始页面，另一半则被转移到新页面上（或者尽可能地平均分割）。在某些情况下，SQL Server 发现即使拆分后，新行（由于变长字段的存在，可能比页面上的现有行都大很多）还是没有足够的空间。作为拆分的一部分，SQL Server 必须将每个新页面的一个相应项添加到上级父页面上。如果只需要拆分一次，就添加一行。但是，如果新行在一次拆分后仍然不能匹配，则父页面可能会有多个新页面和多个附加行。例如，假设一个页面有 32 行，SQL Server 希望插入一个大小为 8000 字节的新行。SQL Server 分隔页面一次后新的 8000 字节行仍然不适合，甚至是二次拆分后，新行仍然不适合。最后，SQL Server 发现新行不适合与其他行放在同一页上，因此它为新行单独分配一个新页面。出现相当多的拆分后，会在父页面上形成很多新页面和很多新行。

索引树总是从根向下搜索，因此在一次 *INSERT* 操作过程中，会一直向下拆分。这表示对一个 *INSERT* 执行搜索索引时，索引受到保护，期待可能的更新。保护机制是一个闩锁，您可以把它看成是锁之类的东西（我们将在第 10 章详细介绍锁）。

在磁盘上读取或写入页面时会获得一个闩锁，从而保护页面内容的物理完整性。父节点一直处于锁定状态，直到子节点所需的拆分完成并且当前操作不需要再对父节点进行进一步更新为止。接下来父闩锁可以被安全地释放。

在父节点上的闩锁被释放之前，SQL Server 将决定页面是否能够容纳另外两行。如果不能，则 SQL Server 将拆分页面。这种情况只有在页面以向索引添加一行为目标进行搜索时才出现。目的是保证父页面总有空间存放子页面拆分所得的结果行（这偶尔也会导致不必要的页面拆分——至少现在不会。从长远角度来看，性能优化可以帮助降低索引中的死锁，同时允许添加可用空间以应对将来行的需要）。拆分的类型由被拆分的页面类型决定：索引的根页、中级索引页或叶级页面。同时，当出现一次拆分时，会单独提交引起页面拆分的事务（使用被称为*系统事务*的特殊内部事务）。因此，即使 *INSERT* 事务被回滚，拆分也不会回滚。

1. 拆分索引的根页

如果一个索引的根页由于一个新索引行的插入而被拆分，则会为索引分配两个新页面。根上的所有行会在这两个新页面之间被拆分，同时新的索引行会被插入到这些页面中某个页面的适当位置。原始页面仍然是根，但是现在根页上只有两行，分别指向新分配的页面。保留原始根页面意味着避免对系统目录（包含一个指向索引根页的指针）中索引元数据的更新。一次根页拆分会在索引上创建一个新级别。由于索引通常只有几级的深度并且非常容易扩展，因此这种类型的拆分不会经常发生。

2. 拆分中级索引页

中级索引页的拆分通过简单地定位页面上索引键的中间点、分配一个新页面及将旧索引页的下半部分复制到新页面来完成。一个新行被添加到拆分页面的上级索引页上，与新添加的页面相对应。同样，这也不经常发生，不过比拆分根页更常见。

3. 拆分叶级页

叶级拆分是最有用也是最常见的操作，可能也是您（作为开发人员或DBA）唯一应该关注的一种拆分。该机制对于拆分聚集索引数据页或非聚集索引叶级索引页来说是相同的。

数据页仅在 *INSERT* 操作并且表中存在聚集索引时才拆分。虽然拆分仅由 *INSERT* 操作引起，但是拆分可以是 *UPDATE* 语句的结果，而不仅仅是 *INSERT* 语句的结果。正如您马上要学到的，如果行不能在适当的位置更新或者至少不是在同一页面上更新，则更新会先*删除*原始行，然后*插入*行的新版本。新行的插入可能会引起页面拆分。

拆分叶级（数据或索引）页是一项复杂的操作。与中级索引页拆分类似，拆分叶级页是通过定位页面上索引键的中间点、分配一个新页面并将旧页面的一半复制到新页面完成的。要求索引管理器确定新行所在的页面并对不适应旧页面或新页面的较大行进行处理。当一个数据页被拆分时，聚集索引键值不会修改，因此非聚集索引不会受到影响。

现在我们来看一下页面拆分时发生的情况。下面的脚本创建一个具有大量行的数据——事实上由于行非常大，使得一页上只适合存储5行。表被创建并用5行数据填充后，就可以通过在 *sp_tablepages* 表中插入 *DBCC IND* 输出结果来找到第一页（此时也是唯一的一页），找到没有上一页的数据页的信息，然后利用 *DBCC PAGE* 查看页面的内容。由于我们不需要查看该页上整个8020字节的数据，因此只查看页尾的槽阵列，看看当我们插入6行时这些行所发生的变化：

```
USE AdventureWorks2008;
GO

DROP TABLE bigrows;
GO

CREATE TABLE bigrows
(
    a int primary key,
    b varchar(1600)
);
GO

/* Insert five rows into the table */
INSERT INTO bigrows
    VALUES (5, REPLICATE('a', 1600));
INSERT INTO bigrows
    VALUES (10, replicate('b', 1600));
INSERT INTO bigrows
    VALUES (15, replicate('c', 1600));
INSERT INTO bigrows
    VALUES (20, replicate('d', 1600));
INSERT INTO bigrows
```

```
    VALUES (25, replicate('e', 1600));
GO

TRUNCATE TABLE sp_tablepages;
INSERT INTO sp_tablepages
    EXEC ('DBCC IND ( AdventureWorks2008, bigrows, -1)' );
GO

SELECT PageFID, PagePID
FROM sp_tablepages
WHERE PageType = 1;
GO

RESULTS: (Yours may vary.)
PageFID PagePID
------- -----------
1       742

DBCC TRACEON(3604);
GO
DBCC PAGE(AdventureWorks2008, 1, 742, 1);
GO
```

下面是 *DBCC PAGE* 输出结果中的槽阵列：

```
Row - Offset
4 (0x4) - 6556 (0x199c)
3 (0x3) - 4941 (0x134d)
2 (0x2) - 3326 (0xcfe)
1 (0x1) - 1711 (0x6a)
```

现在我们再插入一行，然后查看槽阵列：

```
INSERT INTO bigrows
    VALUES (22, REPLICATE('x', 1600));
GO
DBCC PAGE (AdventureWorks2008, 1, 742, 1);
GO
```

新页总会包含原始页中的后半部分行，但是根据键值的不同，新行值可能插入到新页面或旧页面中。在这个示例中，聚集键值为 22 的新行应该是被插入到了页面的后半部分。因此当该页面被拆分时，前 3 行会保留在原始的 742 页面上。可以检查页眉查找包含新行的下一个页面位置。

页码通过 *m_nextPage* 字段表示。该值表示为一个*文件编号：页码*对。因此可以通过 *DBCC PAGE* 命令很容易地使用。在这里，*m_nextPage* 返回的值是 1:21912（当前页附近的任意位置）。使用 *DBCC PAGE* 的“下一页”显示那里的行：

```
DBCC PAGE (AdventureWorks2008, 1, 21912, 1);
```

下面是对第 2 页执行插入操作之后的槽阵列：

```
Row - Offset
2 (0x2) - 1711 (0x6af)
1 (0x1) - 3326 (0xcfe)
0 (0x0) - 96 (0x60)
```

注意，当页拆分后，页上的 3 行是：键为 20 和 25 的最后两个原始行及键为 22 的新行。如果检查页面上的真实数据，会发现新行位于槽 1 处，即使行本身物理位于页面的最后一行。槽 1（值为 22）从偏移量 3326 开始，槽 2（值为 25）从偏移量 1711 开始。行的聚集键顺序通过行的槽编号而不是页面上的物理位置来指示。如果一个表有一个聚集索引，则槽 1 处行的键值比槽 2 处行的键值小，同时比槽 0 处行的键值大。只有槽编号被重新编排，数据没有被重新编排。这是一种优化，从而使只有一小部分的偏移量（而不是整个页面的内容）被重新编排。"索引中的行总是与它们的键存储在完全相同的物理位置"是一种荒诞的说法——实际上只要槽阵列提供正确的逻辑顺序，SQL Server 可以将行存储在页面上的任何位置。

行拆分是代价很大的操作，包括对多个页面（正在被拆分的页、新页、作为正在被拆分页 *m_nextPage* 的页面及父页面）的更新，所有更新都会被完全记入日志。因此需要降低对生产系统中的页面拆分频率，尤其是在峰值使用时间内。可以通过减少拆分来避免对性能造成负面影响。通常可以通过选择一种更好的聚集键（新行被插入到表的末尾，而不是随机插入，好像有一个 GUID 聚集键那样）来最小化拆分，或者当拆分是由对宽度可变的列进行更新而引起时，可以通过创建或重建索引时使用 FILLFACTOR 选项保留页面上的某些空间来降低风险。通过在最不忙的操作时间内周期性地利用所需的 FILLFACTOR 重建索引来充分利用这一设置。这样，峰值使用时间内将有额外的空间可用，从而节省了拆分的系统开销。各种维护选项的利弊我们将在本章后面详细介绍。

6.7.3 删除行

从表中删除行时，必须考虑数据页和索引页将发生的变化。请记住数据实际上是聚集索引的叶级，从有聚集索引的表中删除行与从有非聚集索引的叶级删除行的情况相同。从堆中删除行的管理方式有所不同（与从索引的非叶页面上删除行的情况相同）。

1. 从堆中删除行

当某一行被删除时，SQL Server 2008 不会自动对页面上的空间进行压缩。作为一种性能上的优化，只有当页面需要额外的连续空间存储新插入的行时才进行压缩。您将在接下来的示例中看到压缩的情况，首先从页面的中间删除一行，然后利用 *DBCC PAGE* 检查该页面：

```
USE AdventureWorks2008;
GO
CREATE TABLE smallrows
(
    a int identity,
    b char(10)
);
GO

INSERT INTO smallrows
    VALUES ('row 1');
INSERT INTO smallrows
    VALUES ('row 2');
INSERT INTO smallrows
    VALUES ('row 3');
INSERT INTO smallrows
    VALUES ('row 4');
INSERT INTO smallrows
```

```
    VALUES ('row 5');
GO

TRUNCATE TABLE sp_tablepages;
INSERT INTO sp_tablepages
    EXEC ('DBCC IND (AdventureWorks2008, smallrows, -1)' );

SELECT PageFID, PagePID
FROM sp_tablepages
WHERE PageType = 1;

Results:
PageFID  PagePID
-------  -----------
1        4536

DBCC TRACEON(3604);
GO
DBCC PAGE(AdventureWorks2008, 1, 4536,1);
```

下面是 *DBCC PAGE* 的输出结果：

```
DATA:
Slot 0, Offset 0x60, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes = NULL_BITMAP
Memory Dump @0x61D9C060
00000000: 10001200 01000000 726f7720 31202020 †........row 1
00000010: 20200200 fc†††††††††††††††††††††††††† ...

Slot 1, Offset 0x75, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes = NULL_BITMAP
Memory Dump @0x61D9C075
00000000: 10001200 02000000 726f7720 32202020 †........row 2
00000010: 20200200 fc†††††††††††††††††††††††††† ...

Slot 2, Offset 0x8a, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes = NULL_BITMAP
Memory Dump @0x61D9C08A
00000000: 10001200 03000000 726f7720 33202020 †........row 3
00000010: 20200200 fc†††††††††††††††††††††††††† ...

Slot 3, Offset 0x9f, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes = NULL_BITMAP
Memory Dump @0x61D9C09F
00000000: 10001200 04000000 726f7720 34202020 †........row 4
00000010: 20200200 fc†††††††††††††††††††††††††† ...

Slot 4, Offset 0xb4, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes = NULL_BITMAP
Memory Dump @0x61D9C0B4
00000000: 10001200 05000000 726f7720 35202020 †........row 5
00000010: 20200200 fc†††††††††††††††††††††††††† ...

OFFSET TABLE:
Row - Offset
4 (0x4) - 180 (0xb4)
```

```
3 (0x3) - 159 (0x9f)
2 (0x2) - 138 (0x8a)
1 (0x1) - 117 (0x75)
0 (0x0) - 96 (0x60)
```

现在我们来删除中间行（WHERE *a*=3）并再次查看该页面：

```
DELETE FROM smallrows
WHERE a = 3;
GO

DBCC PAGE(AdventureWorks2008, 1, 4536, 1);
GO
```

下面是第二次执行 *DBCC PAGE* 的输出结果：

```
DATA:
Slot 0, Offset 0x60, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes = NULL_BITMAP
Memory Dump @0x61B6C060
00000000: 10001200 01000000 726f7720 31202020 †........row 1
00000010: 20200200 fc†††††††††††††††††††††††††† ...

Slot 1, Offset 0x75, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes = NULL_BITMAP
Memory Dump @0x61B6C075
00000000: 10001200 02000000 726f7720 32202020 †........row 2
00000010: 20200200 fc†††††††††††††††††††††††††† ...

Slot 3, Offset 0x9f, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes = NULL_BITMAP
Memory Dump @0x61B6C09F
00000000: 10001200 04000000 726f7720 34202020 †........row 4
00000010: 20200200 fc†††††††††††††††††††††††††† ...

Slot 4, Offset 0xb4, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes = NULL_BITMAP
Memory Dump @0x61B6C0B4
00000000: 10001200 05000000 726f7720 35202020 †........row 5
00000010: 20200200 fc†††††††††††††††††††††††††† ...

OFFSET TABLE:
Row - Offset
4 (0x4) - 180 (0xb4)
3 (0x3) - 159 (0x9f)
2 (0x2) - 0 (0x0)
1 (0x1) - 117 (0x75)
0 (0x0) - 96 (0x60)
```

在堆上使用格式 1 的 *DBCC PAGE* 函数时，行不会显示在页面上（只显示在槽阵列中）。页面尾部的槽阵列显示第 3 行（槽 2），它现在位于偏移 0 处（表示没有行使用槽 2），而使用槽 3 的行与插入前位于相同的偏移位置。页面上的数据*没有*被压缩。

除了页面上的空间不能被回收外，堆中的空页也经常不能被回收。即使删除堆中的所有行，SQL Server 也不会将空页面标记为未分配，因此空间不能被其他对象所使用。目录视图 *sys.dm_db_partition_*

stats 仍然显示该空间属于该堆。

2. 从B树中删除行

在索引（聚集或非聚集索引）的叶级中，当行被删除时将被标记为*虚影记录*。这表示行保留在页面上，但是行标题中的某一位发生了变化，表明行实际上已经被删除（一个虚影）。页眉同样会反映页面上虚影记录的数量。虚影记录有多种用途，它们可用于使回滚的效率更高——如果行没有被物理删除，则 SQL Server 回滚 *DELETE* 操作所需做的就是将指示行为的一位进行修改。这对键范围锁定（将在第 10 章介绍）及其他锁定模式来说也是一种并发优化。此外，虚影记录还用于支持行级版本控制，这一内容将在第 10 章介绍。

虚影记录迟早会被清除，具体清除时间根据系统的负载情况而定，有时虚影记录可能在您还没有来得及查看之前就被清除。有一个被称为 *ghost-cleanup thread* 的后台线程，它的工作是删除不再需要用于支持活动事务或其他功能的虚影记录。在下面显示的代码中，如果执行删除操作，然后等待一两分钟后开始运行 *DBCC PAGE*，那么虚影记录可能已经消失了。这就是为什么我们要在运行删除操作之前查看表页码的原因，我们可以通过查询窗口的一次单击来执行删除和 *DBCC PAGE* 操作。为了保证虚影记录不被清除，可以将删除操作放到用户事务中，并且在检查页面之前不对事务执行提交或回滚操作。备份清除线程不会清除作为活动事务一部分的虚影记录。您可以选择使用未记录文件的追踪标志 661 禁用备份清除，从而保证执行像该脚本中这样的测试时可以获得相同的结果。请记住，通常未记录文件的追踪标志不一定会在将来发布的版本或服务补丁中继续存在，也不提供对它们的支持。同时，当您进行测试时请保证关闭这一追踪标志。还可以强制 SQL Server 清除虚影记录。*sp_clean_db_free_space* 程序用于从整个数据库中删除所有虚影记录（只要它们不是未提交事务的一部分），同时 *sp_clean_db_fi le_free_space* 程序用于对一个文件的数据库执行同样的操作。

下面的示例用于构建前面删除示例中使用的表，但是这次表声明了一个主键，即建立了一个聚集索引。数据是聚集索引的叶级，因此数据被删除时会被标记为虚影：

```
USE AdventureWorks2008;
GO
DROP TABLE smallrows;
GO
CREATE TABLE smallrows
(
    a int IDENTITY PRIMARY KEY,
    b char(10)
);
GO
INSERT INTO smallrows
    VALUES ('row 1');
INSERT INTO smallrows
    VALUES ('row 2');
INSERT INTO smallrows
    VALUES ('row 3');
INSERT INTO smallrows
    VALUES ('row 4');
INSERT INTO smallrows
    VALUES ('row 5');
GO
TRUNCATE TABLE sp_tablepages;
INSERT INTO sp_tablepages
```

```
    EXEC ('DBCC IND (AdventureWorks2008, smallrows, -1)' );
SELECT PageFID, PagePID
FROM sp_tablepages
WHERE PageType = 1;

Results:
PageFID  PagePID
-------  -----------
1        4568

DELETE FROM smallrows
WHERE a = 3;
GO
DBCC TRACEON(3604);
DBCC PAGE(AdventureWorks2008, 1, 4544, 1);
GO
```

下面是 *DBCC PAGE* 的输出结果：

```
PAGE HEADER:
Page @0x064AE000
m_pageId = (1:4568)               m_headerVersion = 1        m_type = 1
m_typeFlagBits = 0x4              m_level = 0                m_flagBits = 0x8000
m_objId (AllocUnitId.idObj) = 172  m_indexId (AllocUnitId.idInd) = 256
Metadata: AllocUnitId = 72057594049200128
Metadata: PartitionId = 72057594043105280 Metadata:
IndexId = 1
Metadata: ObjectId = 1179867270   m_prevPage = (0:0)         m_nextPage = (0:0)
pminlen = 18                      m_slotCnt = 5              m_freeCnt = 7981
m_freeData = 201                  m_reservedCnt = 0          m_lsn = (233:499:2)
m_xactReserved = 0                m_xdesId = (0:18856)       m_ghostRecCnt = 1
m_tornBits = 0
DATA:
Slot 0, Offset 0x60, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD       Record Attributes = NULL_BITMAP
Memory Dump @0x61B6C060
00000000: 10001200 01000000 726f7720 31202020 †........row 1
00000010: 20200200 fc†††††††††††††††††††††††††† ...

Slot 1, Offset 0x75, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD       Record Attributes = NULL_BITMAP
Memory Dump @0x61B6C075
00000000: 10001200 02000000 726f7720 32202020 †........row 2
00000010: 20200200 fc†††††††††††††††††††††††††† ...

Slot 2, Offset 0x8a, Length 21, DumpStyle BYTE
Record Type = GHOST_DATA_RECORD    Record Attributes = NULL_BITMAP
Memory Dump @0x61B6C08A
00000000: 1c001200 03000000 726f7720 33202020 †........row 3
00000010: 20200200 fc†††††††††††††††††††††††††† ...

Slot 3, Offset 0x9f, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD       Record Attributes = NULL_BITMAP
Memory Dump @0x61B6C09F
00000000: 10001200 04000000 726f7720 34202020 †........row 4
00000010: 20200200 fc†††††††††††††††††††††††††† ...
```

```
Slot 4, Offset 0xb4, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD        Record Attributes = NULL_BITMAP
Memory Dump @0x61B6C0B4
00000000: 10001200 05000000 726f7720 35202020 †........row 5
00000010: 20200200 fc†††††††††††††††††††††††††† ...

OFFSET TABLE:
Row - Offset
4 (0x4) - 180 (0xb4)
3 (0x3) - 159 (0x9f)
2 (0x2) - 138 (0x8a)
1 (0x1) - 117 (0x75)
0 (0x0) - 96 (0x60)
```

注意行仍然显示在页面本身中（使用格式 1 的 *DBCC PAGE*），因为表有一个聚集索引。同样，您可以试着使用不同的输出样式，查看堆和聚集索引是如何处理虚影记录的，但是您仍然会看到空槽、GHOST_DATA_RECORD 类型或同时看到两者（为了进行解释）。行的标题信息表明这实际上是一条虚影记录。页面尾部的槽阵列显示槽 2 处的行与插入操作前具有相同的偏移，同时所有行也都位于相同的位置。此外，页眉还为我们提供了一个页面中虚影记录数量的值（m_ghostRecCnt）。为了查看表中虚影记录的总数量，可以查看 *sys.dm_db_index_physical_stats* 函数。

更多信息：

虚影清除机制和所包含事务日志检查的详细内容可以参阅 Paul Randal 的博客：*http://www.SQLskills.com/BLOGS/PAUL/post/Inside-the-Storage-Engine-Ghost-cleanup-in-depth.aspx*。

3. 删除索引非叶级中的行

从表中删除一行时，一定会维护所有非聚集索引，因为每个非聚集索引都有一个指向行现在所处位置的指针。索引非叶页面中的行在删除时不会成为虚影，而是与堆页面一样：只有当新索引行需要页面中的空间时才会被压缩。

4. 回收页

当从数据页中删除最后一行时，整个页会被虚影清除后台线程重新分配。特例是我们前面介绍过的表是一个堆的情况（如果某页是表中剩下的唯一一个页，则该页不会被重新分配。一个表中至少会包含一个页，即使该表是一个空表）。数据页的重新分配会导致索引页中指向被重新分配数据页的行被删除。如果一个索引行被删除（同样，更新可能作为 *DELETE/INSERT 策略*的一部分出现），那么非叶索引页将会被重新分配。如果有空间，则该项会被移动到相邻的页面上，接下来空页面被重新分配。

到目前为止，我们所关注的都是删除一行记录所必需的页面操作。如果在一次 *DELETE* 操作中多行被删除，那么您必须知道其他一些问题。由于在一次查询中修改多行的问题对于 *INSERT*、*UPDATE 和 SELETE* 来说是相同的，所以我们将在本章后面对应的小节中对这一问题进行讨论。

6.7.4 更新行

SQL Server 会根据具体操作自动无形地从多种更新策略中选择最快速的方式对行进行更新。在决定

策略的过程中，SQL Server 将估算受影响的行数、行的访问方式（通过扫描或索引检索，以及通过哪个索引）及索引键是否发生变化。更新可能通过将一列的值修改为原始行中一个新值，或者在 *INSERT* 操作之后跟上一条 *DELETE* 语句的方式进行。此外，更新可以通过查询处理器或存储引擎进行管理。在这一部分，我们只检查更新是在位更新还是被 SQL Server 看成是两个独立的操作（删除旧行并插入一个新行）。更新是由查询处理器还是由存储引擎控制的问题实际上与所有数据修改操作相关（而不仅仅是更新操作），因此我们将在一个单独的小节中进行介绍。

1. 移动行

如果表中的一行必须移动到一个新的位置，会发生什么情况呢？在 SQL Server 2008 中，当具有变长列的一行必须更新为一个新的更大尺寸并且原始页面不适合的时候，会出现行移动的现象。当聚集索引或非聚集索引列更改时也会出现行移动的现象，因为行在逻辑上是按照索引键排列的。例如，如果 *lastname* 上有一个聚集索引，则 *lastname* 值为 *Abbot* 的行将被存储在表的开始位置附近。如果 *lastname* 值接下来修改为 *Zappa*，则该行必须移动到表尾附近。

在本章前面我们看到过索引的结构并且发现非聚集索引的叶级包含表中每一行的一个行定位器或书签。如果表有一个聚集索引，则行定位器是该行的聚集键。因此如果聚集索引键被更新，则需要对每一个非聚集索引（除筛选非聚集索引）进行修改。当您决定在哪些行上建立聚集索引时请记住这一点。最好在非易失性列（如标识）上建立聚集索引。

如果某一行由于不再适合原始的页面而发生移动，则该行仍然具有相同的行定位器（换言之，就是行的聚集键保持不变），同时非聚集索引也不必修改。即使表被移动到一个新的物理位置（文件组或分区机制）也仍然如此。只有聚集键改变时非聚集索引才会被更新，同时移动表中某一行的物理位置不会修改该行的聚集键。

在我们讨论索引的内部结构时，我们还发现，如果一个表没有聚集索引（换言之，如果表是一个堆），则存储在非聚集索引中的行定位器实际上是行的物理位置。在 SQL Server 2008 中，如果堆中的一行被移动到一个新页面上，则该行会在原始位置留下一个转发指针。非聚集索引不需要被修改，它们仍然指向原始位置并从那里被引导到新位置。此时，如果表移动到一个新位置（文件组或分区机制），则非聚集索引被更新，由于堆中所有记录的物理位置必须修改，因此会使非聚集索引中先前的行定位器失效。

现在我们来看一个示例。我们已经创建了一个与执行插入操作时创建的表很相似的表，只是该表的第 3 列是变长列。当我们为表填充 5 行数据（填满页面）后，我们对其中的一行进行更新，使第 3 列的长度更长。此时该行不再适合原始页面，必须移动。我们接下来将 *DBCC IND* 的输出结果加载到 *sp_tabl-epages* 表中，从而获得该表使用的页码：

```
USE AdventureWorks2008;
GO
DROP TABLE bigrows;
GO
CREATE TABLE bigrows
(   a int IDENTITY ,
    b varchar(1600),
    c varchar(1600));
GO
INSERT INTO bigrows
    VALUES (REPLICATE('a', 1600), '');
INSERT INTO bigrows
```

```
    VALUES (REPLICATE('b', 1600), '');
INSERT INTO bigrows
    VALUES (REPLICATE('c', 1600), '');
INSERT INTO bigrows
    VALUES (REPLICATE('d', 1600), '');
INSERT INTO bigrows
    VALUES (REPLICATE('e', 1600), '');
GO
UPDATE bigrows
SET c = REPLICATE('x', 1600)
WHERE a = 3;
GO

TRUNCATE TABLE sp_tablepages;
INSERT INTO sp_tablepages
    EXEC ('DBCC IND (AdventureWorks2008, bigrows, -1)' );
SELECT PageFID, PagePID
FROM sp_tablepages
WHERE PageType = 1;
RESULTS:

PageFID  PagePID
-------  -----------
1        2252
1        4586

DBCC TRACEON(3604);
GO
DBCC PAGE(AdventureWorks2008, 1, 2252, 1);
GO
```

我们不会向您显示 *DBCC PAGE* 命令的整个输出结果，只显示前面 *a*=3 行所在槽的内容：

```
Slot 2, Offset 0x1feb, Length 9, DumpStyle BYTE
Record Type = FORWARDING_STUB          Record Attributes =
Memory Dump @0x61ADDFEB
00000000: 04ea1100 00010000 00†††††††††††††††††.........
```

第一个字节中的值 4 表示这只是一个转发存根。接下来 3 个字节中的 0011ea 是行已经移动到的新位置页码。由于这是一个十六进制值，因此需要转换成十进制的 4586。下一组字节（4 字节）告诉我们该页面位于文件 1 的槽 0 处。如果您接下来使用 *DBCC PAGE* 查看页面 4586，则会看到转发记录的内容。

2. 管理转发指针

转发指针允许您修改堆中的数据，无需考虑必须彻底修改非聚集索引的情况。如果已经被转发的一行必须再次移动，则原始的转发指针会更新为指向新的位置。永远不会出现一个转发指针指向另一个转发指针的情况。此外，如果转发行缩短到可以匹配原始位置并且原始位置仍然有空间，则记录必须移回原始位置，同时转发指针将被删除。

SQL Server 将来的版本可能包括对堆中数据执行物理重组的某些机制，从而不再使用转发指针。注意转发指针只存在于堆中，同时重组表的 *ALTER TABLE* 选项不会对堆进行任何操作。您可以重组堆中的一个非聚集索引，而不是表本身。目前，当一个转发指针被创建时，它会永远地保留在那里（只有几种特例除外）。第一个特例是我们曾经提到过的行缩短并返回到原始位置的情况。第二个特例是整

个数据库缩短的情况。当某个文件被收缩时，书签实际上会被重新分配。收缩过程不会产生转发指针。对于由于收缩过程而删除的页面来说，它们所包含的任何转发行或存根都会被有效地“取消转发”。转发指针被删除的其他情况很明显：被转发的行被删除，或者在表上建立一个聚集索引（从而使表不再是一个堆）。

为了获得表中转发记录的数量，可以查看 *sys.dm_db_ index_physical_stats* 函数的输出结果。

3. 在位更新

在 SQL Server 2008 中，在位更新行是规则而不是特例。这意味着行精确地保持在相同页面的相同位置上，只有受影响的字节被修改。此外，日志包含每个在位更新操作的一条记录，除非表上有一个更新触发器或者表被标记为复制。在这些情况下，更新仍然是在位更新，只是如果任何索引键列被更新，那么日志中将记录一次 *DELETE* 和一次 *INSERT* 操作。

在行不能进行在位更新的情况下，非在位更新的代价是极小的，这是由非聚集索引的存储方式及转发指针的作用决定的。事实上，您可以使一个更新成为非在位更新，此时行保持在原始页面上。如果堆被更新或者具有聚集索引的表被更新（没有修改聚集键）时，会发生在位更新。如果聚集键修改而行根本不需要移动时，也可以实现在位更新。例如，如果在包含连续键值 *Able*、*Becker* 和 *Charlie* 的 *lastname* 列上建立一个聚集索引，可能希望将 *Becker* 更新为 *Baker*。由于表在聚集索引键修改后行仍然保持在相同位置，所以 SQL Server 对其进行在位更新。另一方面，如果将 *Able* 更新为 *Buchner*，则不会发生在位更新，但新行可能保持在相同的页面上。

4. 非在位更新

如果由于您正在更新聚集键而不能进行在位更新，则会在 *DELETE* 语句之后跟随一条 *INSERT* 语句。在某些情况下，会进行一种混合更新：某些行是在位更新，某些行是非在位更新。如果正在更新索引键，SQL Server 会建立一个所有需要用 *DELETE* 和 *INSERT* 操作进行修改的行列表。如果行非常小，则该列表会存储在内存中，如果必要的话会同时写入 *tempdb* 中。该列表接下来按照键值和操作符（*DELETE* 或 *INSERT*）进行排序。如果正在修改键值的索引不是唯一的，则 *DELETE* 和 *INSERT* 步骤会被应用到表中。如果索引是唯一的，则需要执行其他操作来将具有相同键的 *DELETE* 和 *INSERT* 操作合成一个 *UPDATE* 操作。

6.7.5 表级数据修改与索引级数据修改

我们只讨论过修改具有不超过一个索引的一行或几行所必需的放置和索引操作。如果您正在一次操作（*INSERT*、*UPDATE* 或 *DELETE*）中或者利用 *BCP* 或 *BULK INSERT* 命令修改多行，并且表有多个索引，则必须了解其他一些问题。SQL Server 为维护属于表的所有索引提供了两种策略：表级修改和索引级修改。查询优化器根据每种策略的预期执行代价来确定具体选择哪种策略。

表级修改有时被称为 *row-at-a-time（每次一行）*，索引级修改有时被称为 *index-at-a-time（每次一索引）*。在表级修改中，所有索引都是在行被修改时对每一行进行维护。如果更新流没有以任何方式排序，则 SQL Server 必须进行大量的随机索引访问，每个索引每更新一行就被访问一次。如果更新流被排序，则不能按照多种顺序排序了，因此非随机索引访问最多只能出现在一个索引中。

在索引级修改中，SQL Server 收集所有要被修改的行并分别对每一种索引进行排序。换言之，排序操作的数量与索引的数量相同。接下来，对每个索引来说，更新会被合并到索引中，同时每个索引页被

访问的次数不会超过一次，即使多个更新适合一个索引叶级页也是如此。

显然，如果更新是小型的（少量的行）并且表和表的索引是相当大的，则查询分析器通常认为表级修改是最好的选择。大部分 OLTP 操作使用表级修改。另一方面，如果更新比较大，那么表级修改需要很多随机 I/O 操作，并且可能要多次读取和写入每个索引中的每个叶级页。此时索引级修改会提供更好的性能。两种策略所需的日志数量是一样的。

通过检查查询执行计划确定更新是在表级还是在索引级完成。如果 SQL Server 在索引级执行更新，会为每个受影响的索引生成一个包含 UPDATE 操作符的计划。如果 SQL Server 在表级执行更新，则只会在计划中看到一个更新操作符。

6.7.6 日志记录

标准的 *INSERT*、*UPDATE* 或 *DELETE* 语句总是会被记入日志来保证原子性，您不能禁止这些操作记入日志。在语句或事务的提交被调用程序承认之前，事务日志（预写日志）必须认为修改是安全的磁盘操作。页面分配和重新分配（包括通过 *TRUNCATE TABLE* 执行的操作）也会记入日志。正如在第 4 章中所看到的那样，当数据库处于 *BULK_LOGGED* 恢复模式时，某些操作可以按照最低限度记入日志，但是关于分配和重新分配的信息依然会写入日志，同时执行最低限度的日志操作。

6.7.7 锁定

任何数据修改都必须使用某种形式的排他锁进行保护。通常 SQL Server 在系统内部做出所有锁定决定，用户或程序员不需要请求某种特殊类型的锁。在第 10 章会说明锁的不同类型及其兼容性。但是，由于锁定与数据修改是紧密联系在一起的，因此您应该了解以下内容。

- SQL Server 中执行的每种类型数据修改都需要某种形式的排他锁。对于大部分修改操作来说，SQL Server 把行锁定看做默认锁，但是如果需要很多锁，SQL Server 可以锁定页面甚至整个表。
- 更新锁可用于表示要执行一次更新的目的，同时它们对于避免死锁现象非常重要。但是最终来说，更新操作要求执行一次排他锁。更新锁使访问序列化，从而保证可以获得排他锁，但是更新锁本身是不充分的。
- 排他锁必须始终被保持直到事务结束，以应对事务需要被撤销的情况（与共享锁不同，共享锁在扫描完页面后就会被释放，如 *READ COMMITTED* 隔离起作用时）。
- 如果必须对表进行一次完整的扫描来确定满足 *UPDATE* 或 *DELETE* 操作的行时，SQL Server 必须检查每一行来确定要修改的行。需要查找个别行的其他过程被中断，即使它们最终修改不同的行。不检查行，SQL Server 就没有办法知道该行是否符合修改条件。如果您正在修改表中行的一个子集（通过 WHERE 子句确定），请保证您有可用的索引允许 SQL Server 直接访问所需的行，从而使其不必扫描表中的每一行。

6.7.8 碎片

*碎片*是一个通用术语，用于描述由于数据修改而可能在索引中出现的各种效果。通常有两种类型的碎片：内部碎片和外部碎片。

内部碎片（通常被称为*物理碎片*或*页密度*）是指索引页（叶级和非叶级）上有浪费空间的情况。可能是以下一些或所有因素造成的。

- 页拆分（如前面所述）使被拆分的页面和新分配的页面上出现空白空间。

- 使页面不完整的 *DELETE* 操作。
- 使页面不完整的行大小（例如，某个聚集索引中一个宽度固定的5000字节数据记录使每个聚集索引数据页中有3000字节的空间浪费）。

内部碎片说明索引占用比所需空间更大的空间，从而导致更多磁盘空间的使用，读取更多页面来处理数据，而且缓冲池使用更多的内存来存储页面。有时内部碎片可能有一定的好处，因为它在不引起页面拆分的情况下允许更多的行插入到页面上。可以利用 FILLFACTOR 和 PAD_INDEX 选项故意实现内部碎片，我们将在下一部分对这一内容进行介绍。

外部碎片（通常被称为*逻辑碎片*或*扩展碎片*）是指包含聚集或非聚集索引叶级的页面或扩展没按照最有效的顺序排列。最有效的顺序指页面和扩展的逻辑顺序（与索引键定义的一样，按照页眉的下一页指针）与数据文件中页面和扩展的物理顺序相同。换言之，具有下一个索引键的行的索引叶级页面物理上也是数据文件中的下一个相邻页面。这与*文件系统*级的碎片不同，系统级中真实的数据文件可能由多个物理部分组成。

外部碎片是由页拆分引起的，会造成聚集或非聚集索引部分的有序扫描效率降低。外部碎片越多，存储引擎能够对扫描所需的页面预读效率就越低。

检测和删除碎片的方法将在下一节讨论。

6.8 管理索引结构

SQL Server 自动对您的索引进行维护，从而保证有正确的行。当您添加新行时，SQL Server 会自动将新行插入到具有聚集索引的表中的正确位置，同时为指向新数据行的非聚集索引添加新的叶级索引。当您删除行时，SQL Server 自动从非聚集索引中删除相应的叶级行。

因此，虽然索引继续包含B树中所有正确的索引，从而帮助 SQL Server 找到您正在查找的行，但是偶尔仍然需要对索引执行维护操作，尤其是处理各种形式的碎片时。此外，索引的多种属性都可以被修改。

6.8.1 删除索引

管理使用 *CREATE INDEX* 命令创建的索引及支持约束的索引之间的一个最大区别在于如何删除索引。*DROP INDEX* 命令只允许您删除使用 *CREATE INDEX* 命令创建的索引。为了删除支持约束的索引，必须使用 *ALTER TABLE* 删除约束。此外，为了删除具有 *FOREIGN KEY* 约束引用的 *PRIMARY KEY* 或 *UNIQUE* 约束，必须首先删除 *FOREIGN KEY* 约束。如果目标是删除索引并立即重建（可能使用一个新的填充因子），可能会留下一个易受攻击的窗口。虽然 *FOREIGN KEY* 约束已经不存在，但是 *INSERT* 语句可以向表中添加一个违反参照完整性的行。

避免这一问题的一种方法是使用 *ALTER INDEX*，这种方法允许您在一条语句中删除和重建表中的一个或所有索引，无需删除 *FOREIGN KEY* 约束的辅助步骤。如果您希望在不必删除和重建索引的情况下重建现有索引，可以选择使用带有 *DROP_EXISTING* 选项的 *CREATE INDEX* 命令。虽然您可以正常地使用带有 *DROP_EXISTING* 选项的 *CREATE INDEX* 命令来重新定义一个索引的属性（如键列或包含性列，或者索引是否唯一），但是如果使用带有 *DROP_EXISTING* 选项的 *CREATE INDEX* 命令来重建一个支持某种约束的索引，则不能进行这些类型的修改。索引必须使用相同的列、以相同的顺序并用相同的唯一性和聚集值来进行重建。

6.8.2 ALTER INDEX

SQL Server 2005 引入了 *ALTER INDEX* 命令来允许您使用一条命令激活各种类型的索引修改，这在以前的 SQL Server 版本中可能需要使用一系列不同的命令才能实现，包括 *sp_indexoption*、*UPDATE STATISTICS*、*DBCC DBREINDEX* 和 *DBCC INDEXDEFRAG*。对每个不同索引的维护活动不再是使用单独的命令或程序，而是使用 *ALTER INDEX* 实现全部维护活动，有关 *ALTER INDEX* 所有选项的完整描述，请参阅 *SQL Server 联机丛书*“ALTER INDEX”主题。

一般可以使用 *ALTER INDEX* 进行 4 种类型的修改，其中有 3 种可以在使用 *CREATE INDEX* 命令创建索引时指定对应的选项。

1. 重建索引

重建索引取代了 *DBCC DBREINDEX* 命令，可以被认为是将 *DROP_EXISTING* 选项替换为 *CREATE INDEX* 命令。但是，该选项还允许索引被移动或分区。一个新的选项允许索引进行联机重建，与联机创建索引的方法相同（如本章前面创建索引部分所述）。我们将简要介绍联机索引的创建和重建。

2. 禁用索引

禁用索引将使索引完全不可用，因此不能再为任何操作查找行。禁用索引还表示索引不会随着数据的修改而得到维护。可以用一条命令禁用一个或所有索引。没有 ENABLE 选项。由于索引被禁用时不会执行任何维护，因此必须完全重建索引才能使它们再次有效。重新启用（可以联机或脱机进行）通过 *ALTER INDEX* 的 *REBUILD* 选项实现。该功能主要是为 SQL Server 应用升级和服务补丁的内部目的而引入的，但是禁用索引有几个作用。首先，如果您希望为了进行检测而临时忽略索引，则可以禁用索引。其次，在加载数据之前不是删除非聚集索引，而是禁用非聚集索引。但是不能禁用聚集索引。如果禁用表上的聚集索引，则表的数据将不可用，因为聚集索引的叶级是数据。禁用聚集索引基本上等于禁用了表。但是，如果数据按照聚集索引的顺序（对于一个不断增长的聚集键）被加载从而使所有新数据都添加在表尾，则禁用非聚集索引可以帮助提高加载性能。一旦数据被加载，则不必提供整个索引的定义就可以重建非聚集索引。所有元数据已经在索引禁用时被保存。

3. 更改索引选项

CREATE INDEX 操作期间可以指定的大部分操作也可以使用 *ALTER INDEX* 命令指定。这些选项是 ALLOW_ROW_LOCKS、ALLOW_PAGE_LOCKS、IGNORE_DUP_KEY、FILLFACTOR、PAD_INDEX、STATISTICS _NORECOMPUTE、MAXP_DOP 和 SORT_IN_TEMPDB。IGNOR_DUP_KEY 已经在本章前面的“索引创建选项”一节中进行了介绍。

FILLFACTOR 和 PAD_INDEX。FILLFACTOR 可能是最经常使用的一个选项，该选项使您能够保留索引每个叶级页面上的一部分空间。在一个聚集索引中，由于叶级包含数据，因此可以使用 FILLFACTOR 控制表中留下多少空间。通过保留可用空间，可以避免以后为添加新条目而拆分页面。FILLFACTOR 的一个重要特性是值不被维护，它只表示索引被建立或重建时为现有数据保留多大空间。如果需要，可以使用 *ALTER INDEX* 命令重建索引并重新指定原来指定的 FILLFACTOR。如果使用 *ALTER INDEX* 时不指定一个新的 FILLFACTOR，则使用原来使用的 FILLFACTOR。

FILLFACTOR 应该总是根据一个个的索引进行指定。如果没有指定 FILLFACTOR，则使用服务器级

的默认值。该值是通过带有 *fillfactor* 选项的 *sp_configure* 程序为服务器指定的。该配置值的默认值为 0（与 100 相同），表示索引的叶级页面尽可能地被填充满。这是不修改服务器级设置的最好方式。FILLFACTOR 仅应用于索引的叶级页面。

在专业和使用频率较高的情况下，您可能希望保留中级索引页中的空间，同时避免页面拆分。为此，可以指定 PAD_INDEX 选项，该选项通知 SQL Server 在所有索引级别上使用同一 FILLFACTOR 值。与 FILLFACTOR 一样，PAD_INDEX 只有在创建索引（或重建）时可用。

创建一个包含 PRIMARY KEY 或 UNIQUE 约束的表时，可以指定关联的索引是聚集索引还是非聚集索引，同时还可以指定 *fillfactor*。由于 *fillfactor* 仅在索引被创建时应用，而且第一次创建表时表中没有数据，因此此时指定 *fillfactor* 是完全没用的。但是，如果您决定在填充表之后重建索引，并且没有指定新的 *fillfactor*，将会使用原始值。还可以使用 *ALTER TABLE* 在表中添加一个 PRIMARY KEY 或 UNIQUE 约束时指定一个 *fillfactor*。如果表中已经有数据，则 *fillfactor* 值会在您建立索引来支持新约束时被应用。

DROP_EXISTING。DROP_EXISTING 选项指定一个给定索引应该作为一个事务被删除和重建。该选项在重建聚集索引时特别有用。一般，当开发人员删除一个聚集索引时，SQL Server 必须重建每一个非聚集索引，从而将其书签修改为 RID（而不是聚集键）。接下来，如果开发人员建立（或重建）一个聚集索引，则 SQL Server 必须再次重建所有非聚集索引以更新书签。*CREATE INDEX* 命令的 DROP_EXISTING 选项允许在两次重建非聚集索引的情况下重建聚集索引。如果您正在完全相同的键上建立索引，则非聚集索引根本不需要被重建。如果正在修改键的定义，则非聚集索引在聚集索引被重建后只被重建一次。可以不使用 DROP_EXISTING 选项重建现有索引，而是使用 *ALTRE INDEX* 命令。

SORT_IN_TEMPDB。SORT_IN_TEMPDB 选项允许您控制 SQL Server 在哪里对建立索引所需键值执行排序操作。默认情况下，SQL Server 使用创建索引时所在文件组中的空间。当索引被创建时，SQL Server 通过扫描数据页找到键值，然后在内部排序缓冲区中建立叶级索引行。当这些排序缓冲区被填满时，叶级索引行会被写入磁盘。如果 SORT_IN_TEMPDB 选项被指定，则排序缓冲区将从 *tempdb* 中进行分配，从而使源数据库中所需的空间大大减少。如果您不指定 SORT_IN_TEMPDB，则不仅源数据库需要足够的可用空间来存储排序缓冲及索引的副本（或数据，如果正在建立聚集索引），而且数据库的磁盘头需要在基本表页面和工作区（存储排序缓冲的位置）之间来回移动。相反，如果 *CREATE INDEX* 命令包含 SORT_IN_TEMPDB 选项，并且 *tempdb* 数据库与正在使用的数据库位于不同的物理磁盘上，则性能可以大大提高。可以优化磁盘头移动，让两个独立的磁盘头读取基表页并管理排序缓冲。如果 *tempdb* 数据库位于比用户数据库所在磁盘更快的磁盘上，并且使用 SORT_IN_TEMPDB 选项，则可以进一步加快索引操作。

4. 重组索引

重组索引是唯一在 *CREATE INDEX* 命令中没有相应选项的修改。原因在于当您创建一个索引时，没有任何东西需要重组。REORGANIXE 选项替代 *DBCC INDEXDEFRAG* 命令并从索引中删除一些碎片，但是不能保证删除所有碎片，正如 *DBCC INDEXDEFRAG* 可能不会删除所有碎片一样（不管名称如何）。在我们讨论删除碎片之前，必须首先讨论碎片的检测，下面介绍这一内容。

6.8.3 检测碎片

正如我们已经在很多示例中看到的那样，*sys.dm_db_index_physical_stats* 的输出结果为索引的每个级

别返回一行。但是，当表被分区时，*sys.dm_db_index_physical_stats* 会把每个分区当做一个表来对待，因此该 DMV 实际上为每个索引每个分区的每个级别返回一行。对于只有行内数据（没有行溢出或 LOB 页面）和一个默认分区的一个小型索引来说，可能只会返回两到三行（每个索引级别一行）。但是如果行溢出并且 LOB 数据有多个分区和附加分配单元，则可能会看到更多的行。例如，包含行溢出数据并且建立在 11 个分区两级深度表上的聚集索引在 *sys.dm_db_index_physical_stats* 返回的碎片报告中有 33 行信息(2 级×11 个分区+行溢出分配单元的 11 个分区)。

本章前面的“分析索引的工具”一节对输入参数和输出结果进行了全面的介绍，下面的列给出了不明显的碎片信息。

- ***Forwarded_record_count***。转发记录（本章前面的“数据修改的内部”一节进行了介绍）只可能在堆中存在，并且只有在具有变长列的行由于更新而使长度增加，从而不适应原始位置时才出现。如果表有很多条转发记录，则扫描表可能是非常低效的。
- ***Ghost_Record_Count and version_ghost_record_count***。虚影记录是物理上仍然在页面上存在但逻辑上已经删除的行，如“数据修改的内部”一节所述。SQL Server 中的后台进程清除虚影记录，此前在虚影记录的位置不能插入任何新记录。因此如果有很多虚影记录，则表会有很多内部碎片（即表分散在更多的页面上并且花费更长的时间进行扫描），这样没有任何好处（页面上没有空间插入新行来避免外部碎片）。可以利用 *version_ghost_record_count* 来计算虚影记录的一个子集。该值报告未完成的快照隔离事务所保留的行数。直到所有相关事务被提交或回滚后，虚影才会被清除。快照隔离将在第 10 章介绍。

6.8.4 删除碎片

如果碎片太严重并且影响了查询性能，则可以利用多种选项来删除碎片。您可能想知道到底多严重才算太严重。首先，碎片不一定是一件坏事。只有在应用程序需要对数据执行一种有序扫描时，才出现碎片数据的最大性能代价。逻辑顺序与物理顺序的差别越大，扫描数据所需的代价也就越大。另一方面，如果应用程序只需要一行或几行数据，则表或索引数据是逻辑有序还是物理相邻或者是否遍布在整个磁盘的随机位置上都不重要。如果您有一种良好的索引用于查找自己感兴趣的行，则 SQL Server 可以非常有效地找到一行或多行，不管这些行物理存储在什么位置。

如果您正在对索引进行有序扫描（如对有聚集索引的表进行表扫描，或者对非聚集索引进行叶级扫描)，通常建议 *avg_fragmentation_in_percent* 值在 5～20 之间，您应该重组您的索引以删除碎片。正如不久就会看到的那样，重组一个索引（也称为*整理碎片*）会将叶级页面压缩到它们原来指定的 *fillfactor*，然后重组叶级页面以修正逻辑碎片，使用该索引原来占用的相同页面。不会分配任何新页面，因此这与重建索引相比是一种更节省空间的操作。

如果 *avg_fragmentation_in_percent value* 大于 30，则应该考虑彻底重建索引了。重建索引意味着为索引分配一整套新页面。这样几乎会删除所有碎片，但是不能保证完全删除碎片。如果数据库中的可用空间本身就是碎块，则可能不能够分配足够的相邻空间来删除区域之间的所有间隙。此外，如果索引正在重建时有需要分配新区域的其他工作在进行，则分配给两个进程的区域可能最终会交错。

碎片整理用于从索引的叶级上删除逻辑碎片，同时使索引联机并尽可能地可用。当对某个索引进行碎片整理时，SQL Server 需要在索引 B 树上申请一个意图排他锁。只有当个别页面正在被操作时才会对它们应用排他页面锁，我们将在本章后面介绍碎片整理算法时对这一内容进行介绍。SQL Server 2008 中的碎片整理利用 *ALTER INDEX* 命令进行初始化。该命令删除碎片的一般格式如下：

```
ALTER INDEX { index_name | ALL }
    ON <object>
      REORGANIZE
            [ PARTITION = partition_number ]
            [ WITH ( LOB_COMPACTION = { ON | OFF } ) ]
```

带有 REORGANIZE 选项的 *ALTER INDEX* 命令与 SQL Server 2000 中的 *DBCC INDEXDEFRAG* 相比提供了更强的功能。该命令支持分区索引，因此您可以选择只对某个特殊分区进行碎片整理，同时该命令还允许您控制 LOB 数据是否受碎片整理的影响。

正如前面提到的那样，每个索引都是使用特殊 *fillfactor* 创建的。初始 *fillfactor* 值使用索引元数据进行存储，因此当请求碎片整理时，SQL Server 可以检查该值。在碎片整理期间，如果 *fillfactor* 的值大于叶级页面上当前 *fillfactor* 的值，则 SQL Server 会试图重新建立初始 *fillfactor*。碎片整理用于压缩数据，同时碎片整理可以通过向每个页面中添加更多的行并增加每个页面的填充百分比来实现。SQL Server 可能最终会在碎片整理之后从索引中删除页。如果当前 *fillfactor* 大于初始 *fillfactor*，则 SQL Server 不能通过将行移出页面来*降低*页填充级别。压缩算法会检查相邻页来查看是否有空间用于将第二个页上的行移动到第一个页上。从 SQL Server 2005 开始，通过在一个滚动窗口上查看 8 个逻辑上连续的页使这一过程变得更加有效。这样可以确定是否有足够的行可以在 8 个页面中来回移动，从而允许单个页面被清空和删除，而且只有此时才移动行。

正如前面所述，SQL Server 2005 页引入了压缩 LOB 页面的选项。默认值是 ON。重组指定聚集索引将在压缩叶级页之前压缩聚集索引中包含的所有 LOB 列。重组某个非聚集索引将压缩索引中非键（*INCLUDED*）列的所有 LOB 列。

在 SQL Server 2000 中，用户可以压缩表中 LOB 的唯一方法是卸载和重新加载 LOB 数据。SQL Server 2005 以后的 LOB 压缩具有低密度区（使用率小于 75%）。LOB 压缩会将页面移出这些低密度统一区并将数据放置在该区域之外已经分配给 LOB 分配单元的其他统一区域的可用空间中。该功能允许更好地使用可能被低密度 LOB 区域所浪费的磁盘空间，不会在压缩阶段或下一阶段分配任何新区域。

重组操作的第二个阶段实际上会将数据移动到行内分配单元内的新页面上，目标是使数据的逻辑顺序与物理顺序相匹配。索引保持联机，因为一次操作中只有两个页面被处理，与堆排序或希尔排序（具体内容不是本书讨论的范围）类似。下面的示例是对真实重组过程的一个简化。假设 *datetime* 数据列上有一个索引。星期一的数据逻辑上位于星期二的数据之前，星期二的数据位于星期三的数据之前，星期三的数据位于星期四的数据之前，依此类推。但是，如果星期四的数据位于页面 77 上，则物理和逻辑顺序毫不匹配，同时会出现逻辑碎片。当对一个索引进行碎片整理时，SQL Server 确定第一个物理页面属于叶级（得到的是页 50）并且第一个逻辑页在叶级上（页 88，保存星期一的数据），同时利用一个附加的新页作为临时存储区域交换两个页面上的数据。交换后，存储星期一数据的第一个逻辑页在页面 50 上，该页面是页码最小的物理页面。每个页面都进行交换之后，所有锁和闩锁将被释放，同时最后移动页上的键被保存。该算法的下一个迭代使用保存键来查找下一个逻辑页面（星期二的数据，现在位于页面 88 上）。下一个物理页面是 77，其中存储星期四的数据。因此另一次交换用于将星期二的数据放在页面 77 上，同时将星期四的数据放在页面 88 上。该过程继续执行，直到不需要再进行交换。注意混合区不会对页面进行碎片整理。

您需要知道使用 REORGANIZE 选项的一些限制。当然，如果索引被禁用，则不能进行碎片整理。同样，由于删除碎片的过程需要操作个别页面，因此如果您试图重组一个将 ALLOW_PAGE_LOCKS 选

项设置为 OFF 的索引时，会得到一条错误消息。如果并发联机索引建立在同一个索引上，或者另一个程序当前正在重组同一个索引，则不能进行重组。

您可以利用 *sys.dm_exec_requests* DMV 中的 *percent_complete* 列来观察每个索引的重组进度。该列中的值报告一个索引重组中完成的百分比。如果正在相同的命令中重组多个索引，可能会发现随着每个索引被执行碎片整理，该值会上下波动。

6.8.5 重建索引

可以以多种方式完全重建一个索引。可以使用 *DROP INDEX* 后跟一个 *CREATE INDEX* 的简单组合，但是这种方法可能是最不可取的。尤其是如果您正在以这种方式重建一个聚集索引时，所有非聚集索引必须在删除聚集索引时被重建。这种非聚集索引的重建对于将叶级中的行定位器从聚集键值修改为行 ID 来说是必需的。接下来当您重建聚集索引时，所有非聚集索引必须再次被重建。此外，如果索引支持 PRIMARY KEY 或 UNIQUE 约束，则您根本不能使用 *DROP IDNEX* 命令——除非先删除所有的 FOREIGN KEY。虽然这是可行的，但不是优先选择。

更好的办法是使用 *ALTER INDEX* 命令或结合使用 DROP_EXISTING 子句和 *CREATE INDEX* 。作为一个示例，下面是重建 *Production.TransactionHistory* 表中 *PK_TransactionHistory_TransactionID* 索引的两种方法：

```
ALTER INDEX PK_TransactionHistory_TransactionID
          ON Production.TransactionHistory REBUILD;

CREATE UNIQUE CLUSTERED INDEX PK_TransactionHistory_TransactionID
              ON Production.TransactionHistory
                  (TransactionDate, TransactionID)
              WITH DROP_EXISTING;
```

虽然 *CREATE* 方法需要知道索引架构，但是这种方法实际上更有效，同时您可以指定更多选项。您可以修改构成索引的列、修改唯一性属性或者将非聚集索引修改为聚集索引，只要表中还没有聚集索引。还可以指定重建时要使用的文件组或分区机制。注意，如果确实修改了聚集索引键的属性，则所有非聚集索引必须被重建，但是只重建一次（而不是两次，如果在执行 *DROP INDEX* 后执行 *CREATE INDEX*，则会重建两次）。

使用 *ALTER INDEX* 命令重建聚集索引时，非聚集索引不需要被重建（只起副作用），因为您根本不能修改索引的定义。但是，您可以指定 ALL（而不是一个索引名称）并且请求重建所有索引。*ALTER INDEX* 方法的另一个好处是您可以只重建一个分区——例如，如果 *sys.dm_db_index_physical_stats* 报告的碎片显示只有一个分区或分区的一个子集中有碎片。

联机索引的建立

两种方法重建索引的默认行为是 SQL Server 占用该索引的一个排他锁，因此在索引被重建时完全不可用。如果索引是聚集索引，则整个表不可用。如果索引是非聚集索引，则表上有一个共享锁，这表示不能进行修改但是其他程序可以对表执行查询操作。当然，您不能从正在重建的索引中获益，因此查询可能不是按想象的那样执行。

SQL Server 2005 引入了联机重建一个或所有索引的选项。ONLINE 选项仅对 *ALTER INDEX* 和 *CREATE INDEX* 命令可用，可以有也可以没有 DROP_EXISTING 选项。下面是建立前面索引的语法，要

联机进行：

```
ALTER INDEX PK_TransactionHistory_TransactionID
        ON Production.TransactionHistory REBUILD WITH (ONLINE = ON);
```

联机建立是通过同时维护索引的两个副本（原始索引（源）和新索引（目标））实现的。目标索引只用于记录重建时所做的修改。所有读入的数据都来自源索引，同时修改也会应用到源索引。由于使用的是 SQL Server 行级版本控制，因此从索引中检索信息的任何人都可以读取到一致的数据。图 6-3（来源于 *SQL Server 联机丛书*）显示了源和目标，同时也显示了建立过程所经历的三个阶段。对于每个阶段来说，图示说明了所允许的访问类型、源和目标表中进行的操作及应用了哪些锁。

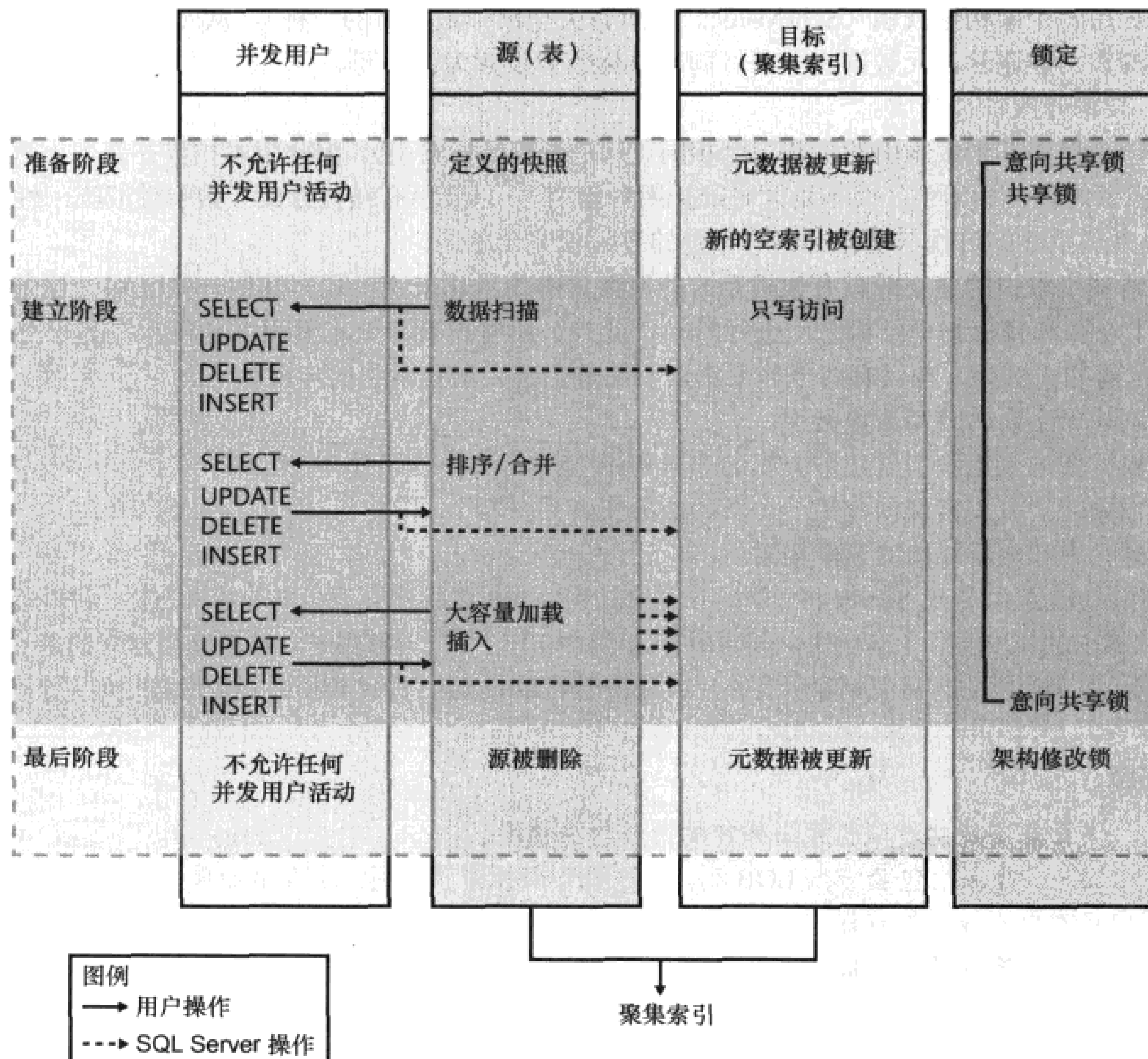


图 6-3 联机索引建立的结构和阶段

实际的过程可能根据索引是初始建立还是重建，以及索引是聚集索引还是非聚集索引等的不同而稍有不同。

下面是重建一个非聚集索引时所包含的步骤。

(1) 索引上的共享锁（S-lock）被占用，从而防止进行任何数据的修改查询，同时表上的一个意向共

享锁（IS-lock）被占用。

（2）索引使用与原始结构相同的结构创建，同时标记为只写。

（3）索引上的共享锁被释放，从而使表上只剩下意向共享锁。

（4）版本扫描（将在第 10 章详细介绍）在原始索引上进行，这表示扫描期间所做的修改将被忽视。被扫描的数据被复制到目标中。

（5）所有后续修改都同时写入源和目标中。读取操作只使用源。

（6）在执行常规操作的同时继续扫描源并向目标进行复制。SQL Server 使用一种专用的方法解决明显的问题，如扫描将记录插入到新索引之前删除该记录。

（7）扫描结束。

（8）占用一个架构修改锁（Sch-M-lock）（所有类型锁中最严格的一种），从而使表完全不可用。

（9）源索引被删除，元数据被更新，同时目标索引设置为读—写。

（10）架构修改锁被释放。

建立一个新的非聚集索引的步骤完全相同（只是没有目标索引），因此版本扫描在基表上进行，同时写操作只需要维护目标索引（而不是同时维护两种索引）。只要没有结构修改（索引键或唯一性属性的修改），聚集索引重建的过程与非聚集索引重建的过程就完全相同。

对于新聚集索引的建立或具有架构修改的聚集索引的重建来说，还有其他一些区别。首先，中级映射索引用于在源和目标物理结构之间进行转换。此外，所有现有非聚集索引都是每次在另一个新基表被建立之后重建的。例如，在具有两个非聚集索引的堆上建立聚集索引的步骤如下。

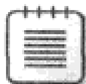
（1）创建一个新的只写聚集索引。

（2）根据新的聚集索引创建一个新的非聚集索引。

（3）根据新的聚集索引创建另一个新的非聚集索引。

（4）删除堆和两个原始的非聚集索引。

在操作完成之前，SQL Server 将一次维护 6 个结构。联机索引建立不会真正被认为是一种性能增强，因为脱机建立索引的速度实际上会更快，同时所有这些结构不需要同时被维护。联机索引建立的是一种有效的功能——您可以重建索引，从而删除所有碎片或重建一个 fillfactor，即使数据必须在任意时刻完全可用。



注意：

下面是不能执行联机索引操作的两种特殊情况。

- 如果索引包含一个 LOB 列，则联机索引操作不可用。这表示如果表包含一个 LOB 列，则聚集索引不能联机重建。只有一个非聚集索引明确地包含一个 LOB 列，联机操作才会被阻止。
- 聚集索引或非聚集索引的单个分区不能被联机重建。

6.9 小结

本章介绍了索引的概念、索引的内部、特殊的索引结构、数据修改及索引管理。我们介绍了很多很好的方法，同时虽然性能优化不是我们的基本目标，但是您对索引的内部工作方式知道得越多，可以创建的最佳结构就越多。此外，通过了解 SQL Server 是如何在磁盘上组织索引的，您可以更善于排除数据库中的问题和管理更改。

第 7 章
特殊存储

Kalen Delaney

在第 5 章和第 6 章中，我们介绍了数据和索引信息“标准行”的存储。我们曾在第 5 章讲过，标准行是以一种被称为 *FixedVar* 的格式存储的。SQL Server 提供了存储数据的另一种格式，被称为*列描述符(CD)*。它还可以以 FixedVar 或 CD 格式存储不适合标准 8KB 页面的特殊值。在本章中，我们将介绍超出标准行大小限制并且作为行溢出或大型对象（LOB）数据存储的数据。我们将向您介绍在实际数据页上存储数据的其他两种方法，这是在 Microsoft SQL Server 2008 中引入的，一种是用具有标准数据行的新类型复杂列(稀疏列)，另一种是用新的 CD 格式(被压缩的数据)。我们还将介绍文件流数据(这是 SQL Server 2008 中的一种新功能)，这种数据允许您像访问关系表一样访问操作系统中的数据。

最后，我们将介绍 SQL Server 将数据分到不同区的功能。虽然这不会改变行中或页上的数据格式，但却会改变用于跟踪空间被分配给哪些对象的元数据。

7.1 大型对象存储

SQL Server 2008 有两种特殊格式用于存储不适合标准 8KB 数据页的数据。这些格式允许行超出 8060 字节的最大行尺寸。正如前面介绍的那样，这个最大行大小包括与行一起存储在物理页面上的几个字节的系统开销，因此所有表已定义列的总大小必须比该值稍微小一些。实际上如果您试图创建一张比允许的最大字节还要大的表，那么所得到的错误消息是非常具体的。如果执行下面列定义正好达到 8060 字节的 *CREATE TABLE* 语句，将会收到错误消息：

```
USE test;
CREATE TABLE dbo.bigrows_fixed
(   a char(3000),
    b char(3000),
    c char(2000),
    d char(60) ) ;

Msg 1701, Level 16, State 1, Line 1
Creating or altering table 'bigrows' failed because the minimum row size would be 8067,
including 7 bytes of internal overhead. This exceeds the maximum allowable table row size of
8060 bytes.
```

在这条消息中，您会看到 SQL Server 希望存储行本身的系统开销字节数量（7）。还有两个额外的字节用于存储页尾的行偏移字节，但是这些字节不包含在总字节中。

7.1.1 长度受限的大型对象数据（行溢出数据）

超出 8060 字节大小限制的一种方式是使用变长列，因为对于变长数据来说，SQL Server 2005 和 SQL Server 2008 可以在特殊的行溢出页中存储这些列，只要所有长度固定的列适合标准行内大小的限制。现

在让我们来看一个所有列都是变长列的表。注意，虽然示例使用的都是 *varchar* 列，但是其他数据类型的列也可以存储在行溢出数据页上。这些其他数据类型包括 *varbinary*、*nvarchar* 和 *sqlvariant* 列，以及使用 CLR 用户定义数据类型的列。下面的代码创建行最大定义长度大于 8060 字节的一张表：

```
USE test;
CREATE TABLE dbo.bigrows
   (a varchar(3000),
    b varchar(3000),
    c varchar(3000),
    d varchar(3000) );
```

实际上，如果在 SQL Server 7.0 中运行这条 *CREATE TABLE* 语句，将会得到一条错误消息，根本不会创建表。在 SQL Server 2000 中，表会被创建，但是如果行大小超过最大值，则会获得一条警告信息，通知您插入或更新操作可能会失败。

在 SQL Server 2005 和 SQL Server 2008 中，不仅先前的 *dbo.bigrows* 表会被创建，而且您可以用一条简单的 *INSERT* 插入列大小加起来不超过 8060 字节的一个行，语句如下：

```
INSERT INTO dbo.bigrows
     SELECT REPLICATE('e', 2100), REPLICATE('f', 2100),
      REPLICATE('g', 2100), REPLICATE('h', 2100);
```

要确定 SQL Server 是否在行溢出数据页中存储某个特殊表中的数据，可以运行第 5 章介绍的如下分配查询语句：

```
SELECT object_name(object_id) AS name,
    partition_id, partition_number AS pnum, rows,
    allocation_unit_id AS au_id, type_desc as page_type_desc,
    total_pages AS pages
FROM sys.partitions p JOIN sys.allocation_units a
   ON p.partition_id = a.container_id
WHERE object_id=object_id('dbo.bigrows');
```

这一查询应该返回与下面结果类似的输出：

```
name    partition_id      pnum rows au_id             page_type_desc     pages
----    ----------------- ---- ---- ----------------- ----------------   -----
bigrows 72057594039238656 1    1    72057594043957248 IN_ROW_DATA        2
bigrows 72057594039238656 1    1    72057594044022784 ROW_OVERFLOW_DATA  2
```

可以看到有两页用于存储标准行内数据的一行，两页用于存储行溢出数据的一行。您也可以选择使用 *DBCC IND*（test,bigrows,-1）单独查看这 4 个页面。使用 *DBCC IND* 只查看 4 个页面不是太难，但是一旦表开始增长并且包含几百或几千个页面（或者更多），则 *DBCC IND* 的输出可能很难使用，因为 *DBCC IND* 每页返回一行。第 6 章为您提供了建立一个名为 *sp_tablepages* 的表的脚本，该脚本用于将 *DBCC IND* 输出收集到一张表中，然后您就可以很容易地找到自己感兴趣的行、计算行的数量、按页类型对页进行分组或者只显示列的一个子集。为了使用 *bigrows* 表中的信息填充该表，运行如下的 *INSERT* 语句：

```
INSERT INTO sp_tablepages
    EXEC ('DBCC IND (test, bigrows, -1)');
```

一旦表被填充，就可以只选择自己感兴趣的列，如下所示：

```
SELECT PageFID, PagePID, ObjectID, PartitionID, IAM_chain_type, PageType
FROM sp_tablepages;
```

您应该看到如下所示的 4 行，每页一行：

```
PageFID  PagePID  ObjectID   PartitionID        IAM_chain_type     PageType
-------  -------  ---------  -----------------  ---------------    --------
1        2252     85575343   72057594039238656  Row-overflow data  3
1        2251     85575343   72057594039238656  Row-overflow data  10
1        2254     85575343   72057594039238656  In-row data        1
1        2253     85575343   72057594039238656  In-row data        10
```

两个页面用于行溢出数据，两个页面用于行内数据。正如您在第 6 章看到的那样，*PageType* 值的含义如下。

- *PageType* = 1，数据页。
- *PageType* = 2，索引页。
- *PageType* = 3，LOB 或行溢出页，TEXT_MIXED。
- *PageType* = 4，LOB 或行溢出页，TEXT_DATA。
- *PageType* = 10，IAM 页面。

我们将在本章后面的"不限长度大型对象数据"一节对不同类型的 LOB 页面做进一步的介绍。

我们可以看到，行内数据有一个数据页和一个 IAM 页，行溢出数据有一个数据页和一个 IAM 页。根据 *DBCC IND* 的结果，可以利用 *DBCC PAGE* 查看页内容。在行内数据的数据页上，我们会看到 4 个 *varchar* 列值中的 3 个，同时第 4 列将被存储在行溢出数据的数据页上。如果为存储行内数据的数据页（本例中是页面 1:2254）运行 *DBCC PAGE*，会发现第 4 列不一定存储在行外的列顺序中。这里不显示行的所有内容，因为一行几乎会占满整个页面。使用 DBCC PAGE 查看行内数据页时，会看到行 *e*、*g* 和 *h*，并且列 *f* 已经被移动到新行上。替代该列的字节如下：

```
65020000 00010000 00290000 00340800 00cc0800 00010000 0067
```

我已经包含了最后一个字节 *e*（ASCII 为十六进制的 65）、第一个字节 *g*（ACSII 为十六进制的 67），以及两者之间的其他 24 字节。这 24 字节中的字节 16～23（第 17 到第 24 字节）被作为一个 8 字节的数值看待：cc08000001000000。我们需要颠倒字节顺序并将其分解：一个 2 字节的十六进制值用做槽编号，一个 2 字节的十六进制值用做文件编号，一个 4 字节的十六进制值用做页码。因此槽 0 的文件编号是 0x0000，因为这个移出列是行溢出页面上的第一条（也是唯一的一条）数据。0x0001 或 1 用做文件编号，0x000008cc 或 2252 用做页码。这也是我们使用 *DBCC IND* 看到的文件和页码。

行中前 16 字节的含义如表 7-1 所示。

表 7-1　行溢出指针的前 16 字节

字　节	十六进制数	十进制数	含　义
0	0x02	2	特殊字段的类型 1=LOB 2=溢出
1～2	0x0000	0	B 树中的级别（0 始终用于溢出）
3	0x00	0	未使用
4～7	0x0000000x	1	序列：乐观并发控制为游标使用的值，每次一个 LOB 或溢出列被更新时该值都会增加
8～11	0x00000029	2 686 976	Timestamp：*DBCC CHECHTABLE* 使用的一个随机值，该值在每个 LOB 或溢出列的生命周期内都保持不变
12～15	0x00000834	2 100	长度

SQL Server 只有在某些情况下才存储行溢出页面上的变长列。决定因素是行本身的长度。SQL Server 试图插入新行的标准页的填充程度无关紧要。SQL Server 正常建构行，同时只要行本身需要超过 8060 字节，SQL Server 就会将其中的某些列存储在溢出页上。

表中的每一列或者完全在行上或者完全不在行上。这表示一个 4000 字节的变长列不能有一半字节在标准数据页上而另一半字节在行溢出页上。如果一行不足 8060 字节并且 SQL Server 要插入该行的页面位置上没有足够的空间，则会应用标准拆分算法（在第 6 章介绍过）。

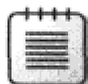
如果一行包含很多大型变长列，则该行可以跨多个行溢出页面。例如，可以创建 *dbo.hugerows* 并向其中插入一行数据：

```
CREATE TABLE dbo.hugerows
  (a varchar(3000),
   b varchar(8000),
   c varchar(8000),
   d varchar(8000));

INSERT INTO dbo.hugerows
     SELECT REPLICATE('a', 3000), REPLICATE('b', 8000),
         REPLICATE('c', 8000), REPLICATE('d', 8000);
```

现在如果运行前面显示的分配查询（用 *hugegrows* 替代 *bigrows*），则会得到如下的结果：

```
name     partition_id      pnum rows au_id             page_type_desc    pages
-------- ----------------- ---- ---- ----------------- ----------------- -----
hugerows 72057594039304192 1    1    72057594044088320 IN_ROW_DATA       2
hugerows 72057594039304192 1    1    72057594044153856 ROW_OVERFLOW_DATA 4
```

行溢出信息有 4 个页面，一个用于行溢出 IAM 页，3 个用于不适合标准行的列。虽然大型变长列非常大，但表中可以包含的大型变长列的数量是不受限制的。任何表中都有 1024 列的限制（使用稀疏列时不受 1024 列的限制，这一内容我们将在本章后面介绍）。但是达到这一限制之前会遇到另一种限制。当一列必须要从一个标准页上移动到一个行溢出页上时，SQL Server 将指向行溢出信息的指针作为原始行的一部分保存，我们在前面的 DBCC 输出中看到该指针是 24 字节，同时该行仍然需要列偏移阵列中的 2 字节用于每个变长列，不论变长列是否存储在该行中。因此 308 是我们可以使用的溢出列的最大数量，同时这样的一行需要 8008 字节用于存储行中每个溢出列的 26 字节系统开销。



注意：

SQL Server 可以在行溢出页面上存储大量大型列，但是并不表示这种做法很好。这一功能可以使您在组织表时变得更灵活，但是如果访问每行数据时都需要很多附加页面，则可能要付出沉重的性能代价。行溢出页面是大部分行完全适合数据页面并且只是偶尔有行溢出数据时的一种选择。SQL Server 可以使用行溢出页面有效地处理额外数据，无需重新设计表。

在某些情况下，如果一个大型变长列缩短，则它可能会被移回标准行。但是，出于效率方面的考虑，如果只是减少几个字节，那么 SQL Server 不会进行检查。只有当存储在一个行溢出页面中的列减少 1000 多字节时，SQL Server 才考虑检查该列现在是否适合标准数据页。如果之前为上一示例创建了 *dbo.bigrows* 表并且只插入每一列中有 2100 字符的一行，可以观察这一行为。

下面的更新使第一列的大小减少 500 字节，行的大小减少到 7900 字节，这样都可以适合一个数据页：

```
UPDATE bigrows
SET a = replicate('a', 1600);
```

但是，如果再次运行分配查询，仍然会看到两个行溢出页面：一个用于存储行溢出数据，另一个用于存储 IAM 页。现在将第一列的大小减少 1000 多字节并再次运行这一分配查询：

```
UPDATE bigrows
SET a = 'aaaaa';
```

现在应该只看到表的 3 个页面，因为不再有行溢出数据页。行溢出数据页的 IAM 页没有被删除，但是不再有存储行溢出数据的数据页。

请记住，行溢出数据存储只应用到变长数据列上，变长数据列每一列的长度不能超过标准变长列的最大值 8000 字节。此外，为了在一个行溢出页面上存储一个变长列，必须满足以下条件。

- 包括系统开销字节在内的所有定长列的长度和不能超过 8060 字节（指向每个行溢出列的指针会在行上添加 24 字节的系统开销）。
- 变长列的实际长度一定不能超过 24 字节。
- 列一定不能是聚集索引键的一部分。

如果个别列需要存储 8000 以上的字节，则应该使用 LOB（*text*、*image* 或 *ntext*）列或使用 *MAX* 数据类型。

7.1.2 不限长度大型对象数据

如果一张表包含旧 LOB 数据类型（*text*、*ntext* 或 *image* 类型），则默认情况下实际数据不存储在标准数据页面上。与行溢出数据一样，LOB 数据存储在自己的页面组内，同时分配查询显示存储 LOB 数据的页面及标准行内数据和行溢出数据的页面。对于 LOB 列来说，SQL Server 在数据行中存储 16 字节的指针指示实际数据的存放位置。虽然默认行为是在数据行之外存储所有 LOB 数据，但是 SQL Server 允许您通过设置表的选项将 LOB 数据存储在数据行内（如果 LOB 数据足够小）来修改存储机制。注意没有对数据页上存储小型 LOB 列的数据库或服务器进行设置的选项，这作为一个表选项进行管理。

默认情况下，没有 LOB 数据存储在数据行中。相反，数据行只包含一个指向所在数据页面（或者一组页面的第一页）的 16 字节指针。这些页面的大小是 8KB，与 SQL Server 中的其他页面一样，个别 *text*、*ntext* 和 *image* 页不限于存储只出现一次的 *text*、*ntext* 或 *image* 列数据。一个 *text*、*ntext* 或 *image* 页可以存储多列和多行中的数据，该页面甚至可以同时有 *text*、*ntext* 和 *image* 数据。但是，每个 *text* 或 *image* 页只能存储一个表中的 *text* 或 *image* 数据（更具体地说，一个 *text* 或 *image* 页只能存储表中某个分区的 *text* 或 *image* 数据，我们将在本章后面讨论分区元数据时详细介绍这一内容）。

构成一个 LOB 列的 8KB 页集合不一定彼此相邻。这些页面逻辑上组织在一棵 B 树中，因此从 LOB 字符串中间开始的操作会非常高效。B 树结构根据数据的数量大于或小于 32KB 而稍微有所不同（参见图 7-1 的一般结构）。B 树已经在第 6 章介绍索引时进行过详细介绍。

如果 LOB 数据的总数不足 32KB，则数据行中的文本指针将指向一个 84 字节的文本根结构。这样就形成了 B 树结构的根节点。根节点指向 *text*、*ntext* 或 *image* 数据块。虽然 LOB 列的数据在 B 树中是按照逻辑顺序组织的，但是根节点和个别数据块在物理上都是扩展到表的整个 LOB 页面。它们被放置在任何可用的位置上。每个数据块的大小由应用程序写入的大小决定。小数据块结合起来填充一个页面。如果数据量不足 64 字节，则数据都会被存储在根结构中。

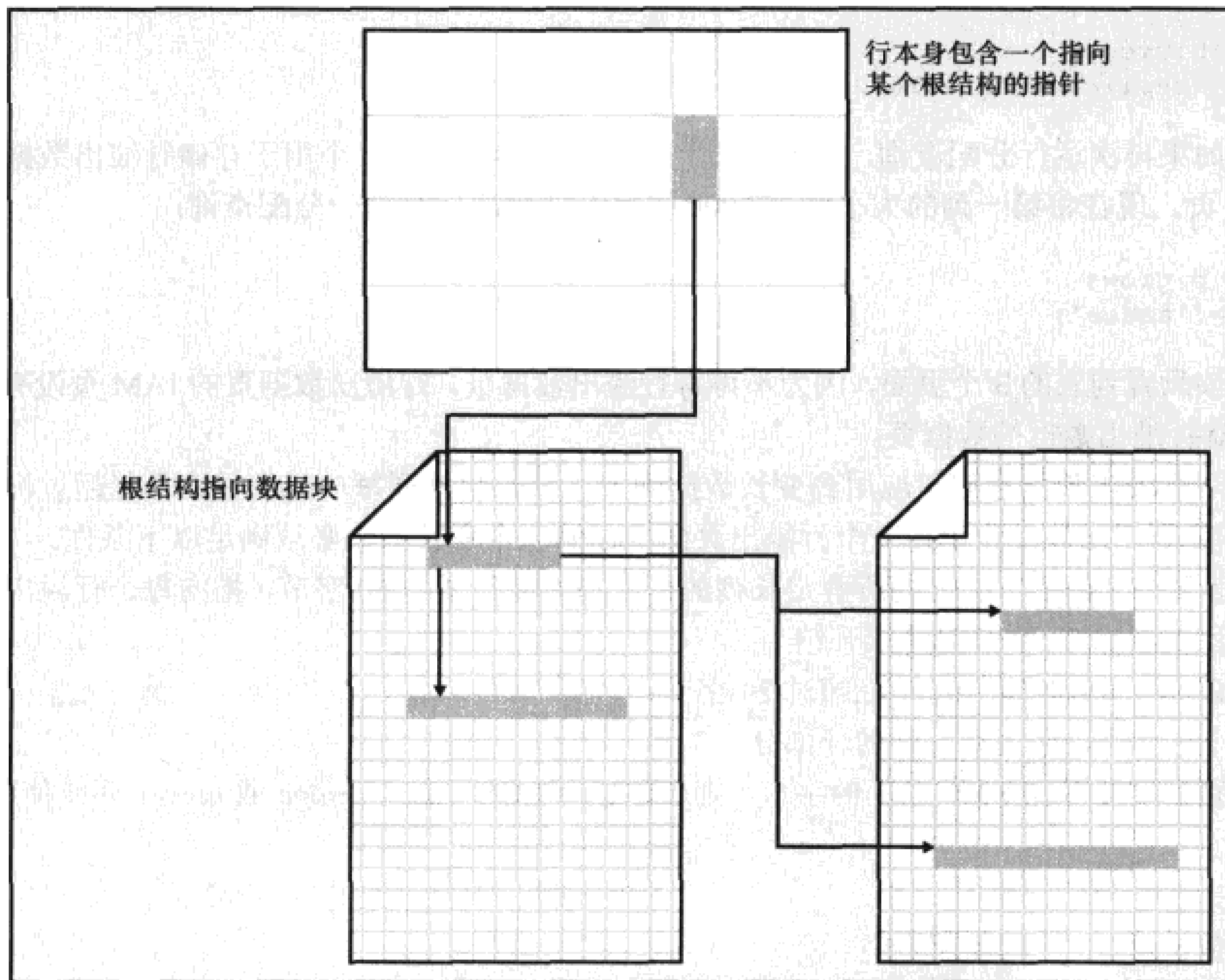


图 7-1 指向包含数据块的一棵 B 树的一个文本列

如果一个 LOB 列一行的数据总量超过 32KB，则 SQL Server 开始在数据块和根节点之间建立中间节点。根结构和数据块在整个 *text* 和 *image* 页上相互交错。但是，中间节点被存储在不能被 *text* 或 *image* 列共享的页面上。存储中间节点的每个页面都只包含某个数据行中一个 *text* 或 *image* 列的中间节点。

SQL Server 可以在两种不同类型的页面上存储 LOB 根及实际的 LOB 数据。其中一种页面（被称为 TEXT_MIXED）允许多行内的 LOB 数据共享相同的页面。但是，一旦文本数据超过 40KB，SQL Server 就会把整个页面专用于一个 LOB 值。这些页面称为 TEXT_DATA 页。

可以通过创建一个具有 *text* 列的表，插入一个小于 40KB 的值，然后插入一个大于 40KB 的值并检查 *DBCC IND* 输出来观察这一行为。下面的脚本使用前面创建的 *sp_tablepages* 表：

```
IF EXISTS (SELECT * FROM sys.tables
                         WHERE name = 'textdata')
    DROP TABLE textdata;
GO
CREATE TABLE textdata
 (bigcol text);
GO
INSERT INTO textdata
   SELECT REPLICATE(convert(varchar(MAX), 'a'), 38000);
GO
TRUNCATE TABLE sp_tablepages;
GO
INSERT INTO sp_tablepages
```

```
  EXEC('DBCC IND(test, textdata, -1)');
GO
SELECT PageFID, PagePID, ObjectID, IAM_chain_type, PageType
FROM sp_tablepages;
GO
INSERT INTO textdata
   SELECT REPLICATE(convert(varchar(MAX), 'a'), 41000);
GO
TRUNCATE TABLE sp_tablepages;
GO
INSERT INTO sp_tablepages
  EXEC('DBCC IND(test, textdata, -1)');
GO
SELECT PageFID, PagePID, ObjectID, IAM_chain_type, PageType
FROM sp_tablepages;
```

第一次对 *sp_tablepages* 进行查询时，应该使 *PageType* 的值为 1、3 和 10。已经插入大小大于 40KB 的数据时，第二次还是应该看到 *PageType* 的值为 4。*PageType* 为 3 表示一个 TEXT_MIXED 页，PageType 为 4 表示一个 TEXT_DATA 页。

数据行中存储的 LOB 数据

如果在标准数据页之外存储所有 LOB 数据类型值，则 SQL Server 需要在每次访问这些数据时对其他页面进行读取，正如对行溢出页的操作那样。在某些情况下，允许某些 LOB 数据存储在数据行中可能会改善性能。可以通过将一个名为 *text in row* 的表选项设置为'ON'（包括引号标记）或者通过指定存储在数据行中的最大字节数来为某个特殊表启用该选项。下面的命令用一个名为 *employee* 的表的标准行数据存储 500 字节的 LOB 数据。

```
EXEC sp_tableoption employee, 'text in row', 500;
```

注意值是字节而不是字符。对于 ntext 数据来说，每个字符需要 2 字节，从而使任何小于或等于 250 字节的 *ntext* 列都可以存储在数据行中。启用 *text in row* 选项后，就不会只获得行中 LOB 数据的 16 字节指针，因为这是该选项不设置为'ON'时的情况。如果 LOB 字段中的数据比指定的最大值大，则该行会存储包含指向单独 LOB 数据块的指针的根结构。根结构的最小大小是 24 字节，并且 *text in row* 可以设置的值范围是 24～7000 字节（如果指定该选项为'ON'而不是一个特殊编号，则 SQL Server 会认为默认值为 256 字节）。

为了禁用 *text in row* 选项，可以将值设置为'OFF'或 0。为了确定表的 *text in row* 属性是否启用，可以利用如下语句检查 *sys.tables* 目录视图：

```
SELECT name, text_in_row_limit
FROM sys.tables
WHERE name = 'employee';
```

Text_in_row_limit 值表示允许在一个数据行中存储的 LOB 数据的最大字节数。如果返回 0，则 *text in row* 选项被禁用。

现在创建一个与查看行结构时创建的表非常类似的表，但是这次将 *varchar（250）*列修改为 *text* 数据类型。我们几乎是用相同的插入语句向表中插入一行：

```
CREATE TABLE HasText
(
```

```
Col1 char(3)        NOT NULL,
Col2 varchar(5)     NOT NULL,
Col3 text           NOT NULL,
Col4 varchar(20)    NOT NULL
);

INSERT HasText VALUES
    ('AAA', 'BBB', REPLICATE('X', 250), 'CCC');
```

现在利用分配查询查找表的基本信息并查看该表的 *DBCC IND* 值：

```
SELECT convert(char(7), object_name(object_id)) AS name,
     partition_id, partition_number AS pnum, rows,
     allocation_unit_id AS au_id, convert(char(17), type_desc) as page_type_desc,
    total_pages AS pages
FROM sys.partitions p JOIN sys.allocation_units a
  ON p.partition_id = a.container_id
WHERE object_id=object_id('dbo.HasText');

DBCC IND (test, HasText, -1);

name     partition_id        pnum  rows   au_id               page_type_desc   pages
-------  -----------------   ----  -----  -----------------   ---------------  -----
HasText  72057594039435264  1      1      72057594044350464   IN_ROW_DATA      2
HasText  72057594039435264  1      1      72057594044416000   LOB_DATA         2

PageFID  PagePID  ObjectID   PartitionID        IAM_chain_type    PageType
-------  -------  --------   -----------------  --------------    --------
1        2197     133575514  72057594039435264  LOB data          3
1        2198     133575514  72057594039435264  LOB data          10
1        2199     133575514  72057594039435264  In-row data       1
1        2200     133575514  72057594039435264  In-row data       10
```

可以看到两个 LOB 页面（LOB 数据页和 LOB IAM 页），以及用于存储行内数据的两个页面（同样是数据页和 IAM 页）。存储行内数据的数据页是 2199，同时 LOB 数据位于页 2197 上。图 7-2 显示页 2199 上运行 *DBCC PAGE* 的输出。行结构与第 5 章图 5-10 中显示的行结构非常类似，只是 *text* 字段本身有所不同。字节 21～36 是 16 字节的文本指针，同时可以看到从偏移位置 29 开始的值 9 508。反转字节时，将变成 0x0895 或十进制的 2197，这是包含文本数据的页面（如我们在 *DBCC IND* 输出中看到的那样）。

```
DATA:

Slot_0,_Offset_0x60,_Length_40,_DumpStyle_BYTE

Record Type = PRIMARY_RECORD       Record Attributes = NULL_BITMAP_VARIABLE_COLUMNS
Record Size = 40
Memory Dump @0x625BC060

00000000:   30000700_41414104_00600300_15002580_†0...AAA..`....%.
00000010:   28004242_420000e1_07000000_00950800_†(.BBB..á.....?..
00000020:   00010001_00434343_††††††††††††††††††.....CCC
```

图 7-2 包含一个文本指针的一行

现在启用行中的文本数据，最多 500 字节：

```
EXEC sp_tableoption HasText, 'text in row', 500;
```

启用该选项不会强制文本数据移动到行中。我们必须更新文本值来实际强制数据的移动：

```
UPDATE HasText
SET col3 = REPLICATE('Z', 250);
```

如果在原始数据页上运行 *DBCC PAGE*，会发现 250z 的文本列现在位于数据行上，同时该行实际上与在图 5-10 中看到的包含 *varchar* 数据的行在结构上是一致的。

虽然启用 *text in row* 不会立即移动数据，但是禁用该选项会立即移动数据。如果关闭 *text in row*，则 LOB 数据立即移动回自己的页面上，因此必须保证在大型操作期间不对大型表关闭该选项。

使用 LOB 数据和 *text in row* 选项的最后一个问题是处理 *text in row* 被启用但是 LOB 却比为某些行配置的最大长度大的问题。如果将 *HasText* 表的 *text in row* 最大长度设置为 50，则会强制具有超过 50 字节 LOB 数据的所有行的 LOB 数据立即移出该页，正如我们完全禁用该选项时一样：

```
EXEC sp_tableoption HasText, 'text in row', 50;
```

但是，将这一限制设置为一个更小的值与禁用该选项不同。首先，某些行可能有小于限制的 LOB 数据，对于这些行来说，LOB 数据完全存储在数据行中。其次，如果 LOB 数据不适合，则存储在数据行自身中的信息就不是简单的 16 字节指针，与关闭 *text in row* 时的情况相同。相反，对于不适合已定义大小的 LOB 数据来说，行包含指向 LOB 数据块的 B 树根结构。只要 *text in row* 选项不是'OFF'（或 0），那么 SQL Server 就不会在行中存储简单的 16 字节 LOB 指针，而是会存储 LOB 数据本身（如果适合）或 LOB 数据 B 树的根结构。

一个根结构必须至少是 24 字节长（这也是为什么 *text in row* 最小大小限制为 24 的原因），并且字节的含义与行溢出指针中 24 字节的含义类似。主要区别在于字节 12～15 不存储任何数据。相反，字节 12～23 由指向独立页面上 LOB 数据块的链接构成。如果多个 LOB 块通过根进行访问，则这里会有多组 12 字节，每一组均指向一个独立页面的 LOB 数据。

正如前面所指出的那样，第一次启用 *text in row* 时，不会对任何数据进行移动，直到文本数据真正被更新。限制被增加时也是如此——也就是说，即使新限制足以容纳存储行外 LOB 数据，LOB 数据也不会自动移动到行上。您必须首先更新实际的 LOB 数据。

另一点需要记住的是，即使 LOB 数据的数量小于限制，数据也不一定存储在行中。您仍然受到一个数据页上一行最多 8060 字节的限制，因此，如果非 LOB 数据的总量非常大，存储在实际数据行中的 LOB 数据总量可能会减少。此外，如果一个变长列需要增长，可能会使 LOB 数据离开该页面，从而实现不超过 8060 字节的限制。变长列的增长总是优先于在行中存储 LOB 数据。如果在一次更新操作期间没有变长 *char* 字段需要增长，则 SQL Server 会检查行内 LOB 数据的增长（按照列偏移的顺序）。如果一个 LOB 需要增长，则其他列会离开该行。

最后，您应该知道 SQL Server 会记录 LOB 数据的所有移动，这表示降低限制或关闭 *text in row* 选项对于一个大型表来说可能是一种非常费时的操作。

虽然使用 LOB 数据类型的大型数据列可以被有效地存储和管理，但是在表中使用这样的列可能有问题。以 *text*、*ntext* 或 *image* 存储的数据不能总利用数据操纵命令进行操纵，同时在很多情况下，您需要借助 *readtext*、*writetext* 和 *updatetext* 进行操作，这些操作需要处理字节偏移和行长度值。在 SQL Server 2005 以前，您必须确定是将列限制为 8000 字节的最大值，还是使用与较短列使用的不同操作符来处理大型数

据列。SQL Server 2005 和 SQL Server 2008 提供了最佳解决方案，我们将在下一节介绍。

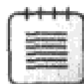
7.1.3 最大长度数据的存储

SQL Server 2005 和 SQL Server 2008 为我们提供了使用 MAX 说明符定义变长字段的选项。虽然该功能通常是通过 *varchar（MAX）*使用的，但是 MAX 说明符也可以用于 *nvarchar* 和 *varbinary*。可以在使用其中一种类型定义列、变量或参数时指定 MAX 说明符（而不是实际大小）。通过使用 MAX 说明符，可以让 SQL Server 决定是将值作为一个标准的 *varchar*、*nvarchar* 或 *varbinary* 值存储还是作为一个 LOB 存储。通常，如果实际长度是 8000 字节或更少，则会被当做一种标准的变长数据类型对待，可能会溢出到行溢出页面。但是，如果 *varchar(MAX)*列确实会溢出该页面时，所需的额外页面被认为是 LOB 页面并且在使用 *DBCC IND* 检查时会显示 *IAM_chain_type* LOB。如果实际长度大于 8000 字节，则 SQL Server 会像存储和对待 *text*、*ntext* 或 *image* 一样存储和对待该值。由于具有 MAX 说明符的变长列要么作为标准变长列对待，要么作为 LOB 列对待，因此无需对它们的存储进行具体的讨论。

使用 MAX 指定的值大小可以达到 LOB 数据所支持的最大值，目前是 2GB。尽管如此，使用 MAX 说明符，正在显示的最大值应该是系统支持的最大值。如果将具有 *varchar(MAX)*列的一张表升级为将来的某个 SQL Server 版本，则 MAX 长度将是新版本中的最大值。

提示：

由于 MAX 数据类型可以存储 LOB 数据和标准行数据，因此建议您在将来的开发中使用这些数据类型来代替 *text*、*ntext* 或 *image* 类型，这几种数据类型会在将来的版本中删除。



注意：

虽然 LOB 这种缩写可以表示“大型对象”，但是将来我们会用这两个术语表示两种不同的事物。当我们希望指图 7-1 中使用特殊存储格式的数据时，我们只使用 LOB 这一术语。当我们希望指任意存储标准数据页无法存储的数据的方法时，我们使用“大型对象”这一术语，这包括行溢出列、实际的 LOB 数据类型、MAX 数据类型及文件流数据。

向 LOB 列追加数据

在存储引擎中，每个 LOB 列都被分成最大长度为 8040 字节的片段。向一个大型对象追加数据时，SQL Server 会查找追加点并查看将添加新数据的当前片段。SQL Server 计算新片段的大小（包括新追加的数据）。如果大小超过 8040 字节，则 SQL Server 会分配新的大型对象页，直到出现一个小于 8040 字节的片段为止，然后找一个有足够空间存储剩余字节的页面。

当 SQL Server 为 LOB 数据分配页面时，可以使用两种分配策略。

（1）对于大小不足 64KB 的数据来说，随机分配一个页面。该页面来自大型对象 IAM 某部分的一个区，但是这些页面不一定是连续的。

（2）对于大小超过 64KB 的数据来说，会使用一次只追加一个扩展的页面分配器并在区内连续写入页面。

因此从性能的角度来看，每次写 64KB 的片段是有好处的。如果知道其大小为 1MB，则提前分配 1MB 可能有好处。但是，您还需要考虑事务日志所需的空间。如果首先创建一个 1MB 的片段（具有任意内容），

则 SQL Server 会将 1MB 记入日志，接下来所有修改也会记入日志。当您指定大型对象数据更新时，不需要分配新页面，但修改仍然需要记入日志。

只要大型对象的值较小，就能存储在数据页中。此时，预分配可能是一种很好的选择，可以使大型对象数据不会太零碎。通常的建议是：如果一次操作插入到一个大型对象列中的数据总量相对较小，则请插入最终所需值的一个大型对象值，然后根据需要替换初始值的字串。对于较大的长度来说，请试着在 8×8040 字节的块中进行追加或插入。这样每次会分配一整个区，同时每个页面上会存储 8040 字节。

如果发现大型对象数据变得很零碎，可以利用 ALTER INDEX REORGANIZE 的一个选项对大型对象数据进行碎片整理。实际上该选项（WITH LOB_COMPACTION）默认处于开启状态，因此您只需保证不将其关闭就可以了。

7.2 文件流数据

虽然 SQL Server 用于在数据库中存储大型对象数据的灵活方法为您在文件系统中存储数据提供了很多优势，但是这些方法也有很多缺点。在数据库中存储大型对象的一些优点如下。

- 可以保证大型对象数据的事务一致性。
- 备份和还原操作包括大型对象数据，允许您对大型对象进行集成的时间点恢复。
- 所有数据都可以使用一种存储和查询环境进行存储。

在数据库中存储大型对象的一些缺点如下。

- 大型对象在缓存中占用非常大的缓冲区。
- 更新大型对象可能会导致大量数据库碎片。
- 数据库文件可能变得非常大。

SQL Server 2008 允许您管理文件系统对象，就好像它们是数据库的一部分，这样就提供了将大型对象放在数据库的好处并使缺点最小化。文件系统中存储的数据称为文件流。当您开始估算文件流数据是否有利于应用程序时，必须综合考虑优缺点。文件流数据的一些优点如下。

- 大型对象数据存储在文件系统中，但是在数据库中却是作为包含文件流数据的一个 48 字节的文件指针值存储在根部。
- 大型对象数据在事务中与结构数据保持一致。
- 大型对象数据可通过 T-SQL 和 NTFS 流 API 访问，从而提供最大性能优势。
- 大型对象大小只受 NTFS 卷大小的限制，不受 LOB 对象的 2GB 限制。

使用文件流对象的一些缺点如下。

- 对包含文件流数据的数据库不能使用数据库镜像。
- 数据库快照不能包括文件流文件组，因此文件流数据不可用。请求文件流列的数据库快照中的 *SELECT* 语句会产生错误。
- 文件流数据不能被 SQL Server 本机加密。

7.2.1 为 SQL Server 启用文件流数据

必须同时在 SQL Server 2008 实例内部和外部启用访问文件流数据的功能，这在第 1 章讨论配置时曾经提到过。通过 SQL Server 配置管理器，必须使 T-SQL 能够访问文件流对象，同时如果启用该选项，也能够启用文件 I/O 流访问。如果允许文件 I/O 流访问，则可以在需要时允许远程客户端访问流数据。一旦

SQL Server 配置管理器被打开，请保证已经选择了左侧窗格中的“SQL Server 服务”。在右侧面板中，用右键单击要配置的 SQL 实例并从下拉菜单中选择“属性”。弹出的“SQL Server 属性”对话框有 4 个选项卡，其中包括一个名为 *FILESTREAM* 的选项卡。可以查看“SQL Server 属性”对话框 *FILESTREAM* 选项卡的详细信息，如图 7-3 所示。

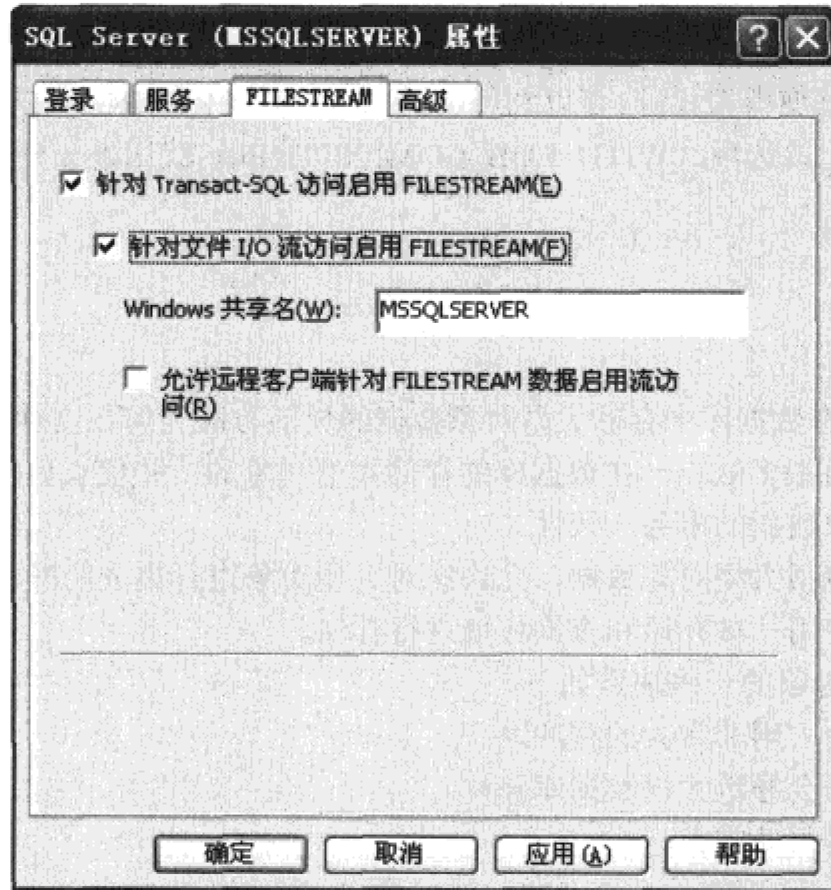


图 7-3 配置一个 SQL Server 实例允许 *FILESTREAM* 访问

配置完服务器实例后，需要使用 *sp_configure* 将 SQL Server 实例设置为所需的文件流访问级别。可选值有 3 个。值 0 表示不允许文件流被访问，值 1 表示可以使用 T-SQL 访问文件流数据，值 2 表示可以使用 T-SQL 和 Win32 API 访问文件流数据。与所有配置选项一样，不要忘记在修改某项设置后运行 *RECONFIGURE* 命令，代码如下：

```
EXEC sp_configure 'filestream access level', 1;
RECONFIGURE;
```

7.2.2 创建一个启用文件流的数据库

为了存储文件流数据，数据库必须至少创建一个文件组来允许文件流数据。创建一个数据库时，允许文件流数据的文件组与包含行数据的文件组的指定方式在很多方面有所不同。

- 文件流文件组中只能有一个文件。
- 为文件流文件组指定的路径最多只能到达最后一个文件夹名称。最后一个文件夹名称不能存在但是将在 SQL Server 创建数据库时创建。
- *size*、*maxsize* 和 *filegrowth* 属性不应用到文件流文件组上。
- 如果没有指定为 DEFAULT 的文件流包含文件组，则列出的第一个文件流包含文件组是默认值（因此，行数据有一个默认文件组，文件流数据有一个默认文件组）。

观察下面的代码，这段代码创建一个具有两个文件流包含文件组的数据库。路径 C:\Data2 必须存在，但是不能包含 *filestream1* 或 *filestream2* 文件夹：

```
CREATE DATABASE MyFilestreamDB
ON
PRIMARY ( NAME = Rowdata1,
    FILENAME = 'c:\Data2\Rowdata1.mdf'),
FILEGROUP FileStreamGroup1 CONTAINS FILESTREAM DEFAULT( NAME = FSData1,
    FILENAME = 'c:\Data2\filestream1'),
FILEGROUP FileStreamGroup2 CONTAINS FILESTREAM ( NAME = FSData2,
    FILENAME = 'c:\Data2\filestream2')
LOG ON ( NAME = FSDBLOG,
    FILENAME = 'c:\Data2\FSDB_log.ldf');
```

当上面的 *MyFilestreamDB* 数据库被创建时，SQL Server 在 C:\Data2 目录下创建 *filestream1* 和 *filestream2* 两个文件夹。这些文件夹被当做文件流*容器*。初始情况下，每个容器包含一个名为 *$FSLOG* 的空文件夹和一个名为 *filestream.hdr* 的标题文件。由于表被创建来使用这一容器中的文件流空间，因此每个分区的一个文件夹或包含文件流数据的每张表都在该容器中创建。

可以向现有数据库添加一个文件流文件组，然后使用 *ALTER DATABASE* 命令向文件流文件组添加一个文件。注意不能向 *master*、*model* 和 *tempdb* 数据库添加文件流文件组。

7.2.3 创建一张表存储文件流数据

可以指定一列来包含文件流数据，该列必须被定义为具有一个 *FILESTREAM* 属性的 *varbinary(MAX)* 类型。包含该表的数据库必须至少有一个文件组是为 *FILESTREAM* 定义的。表创建语句可以指定表的文件流数据存储在哪个文件组中，如果不指定，则使用默认文件流文件组。最后，具有文件流列的表必须有一个指定了 *ROWGUIDCOL* 属性的 *uniqueidentifier* 数据类型的列。该列不允许 NULL 值，同时必须通过指定 UNIQUE 或 PRIMARY KEY 单列约束来保证唯一性。*ROWGUIDCOL* 列作为 *FILESTREAM* 代理可以用于定位表中的实际行来检查权限、获得文件的物理路径并在需要时锁定行。

现在看一下在容器中创建的文件。当表在 *MyFilestreamDB* 数据库中创建时，该表会向 *FileStreamGroup1* 容器添加多个文件夹：

```
CREATE TABLE MyFilestreamDB.dbo.Records
(
        [Id] [uniqueidentifier] ROWGUIDCOL NOT NULL UNIQUE,
        [SerialNumber] INTEGER UNIQUE,
        [Chart_Primary] VARBINARY(MAX) FILESTREAM NULL,
        [Chart_Secondary] VARBINARY(MAX) FILESTREAM NULL)
FILESTREAM_ON FileStreamGroup1;
```

由于该表是在 *FileStreamGroup1* 上创建的，因此 *filestream1* 容器被使用。*filestream1* 会为 *FIleStreamGroup1* 文件组中的每个表或分区创建一个子文件夹，这些文件名称将是 GUID。这些文件中的每个文件有一个子文件夹用于表或分区中的每个列（存储文件流数据），这些文件夹的名称将是 GUID。图 7-4 显示了创建 *MyFilestreamDB.dbo.Records* 表后磁盘上文件的结构。*Filestream2* 文件夹只有$FSLOG 子文件夹，同时没有用于任何表的子文件夹。*Filestream1* 文件夹有一个用于 *dbo.Records* 表的、以 GUID 命名的子文件夹，同时该文件夹内有一个以 GUID 命名的子文件夹用于存储表中的两个 *FILESTREAM* 列。除原始 *filestream.hdr* 文件之外仍然没有任何文件。

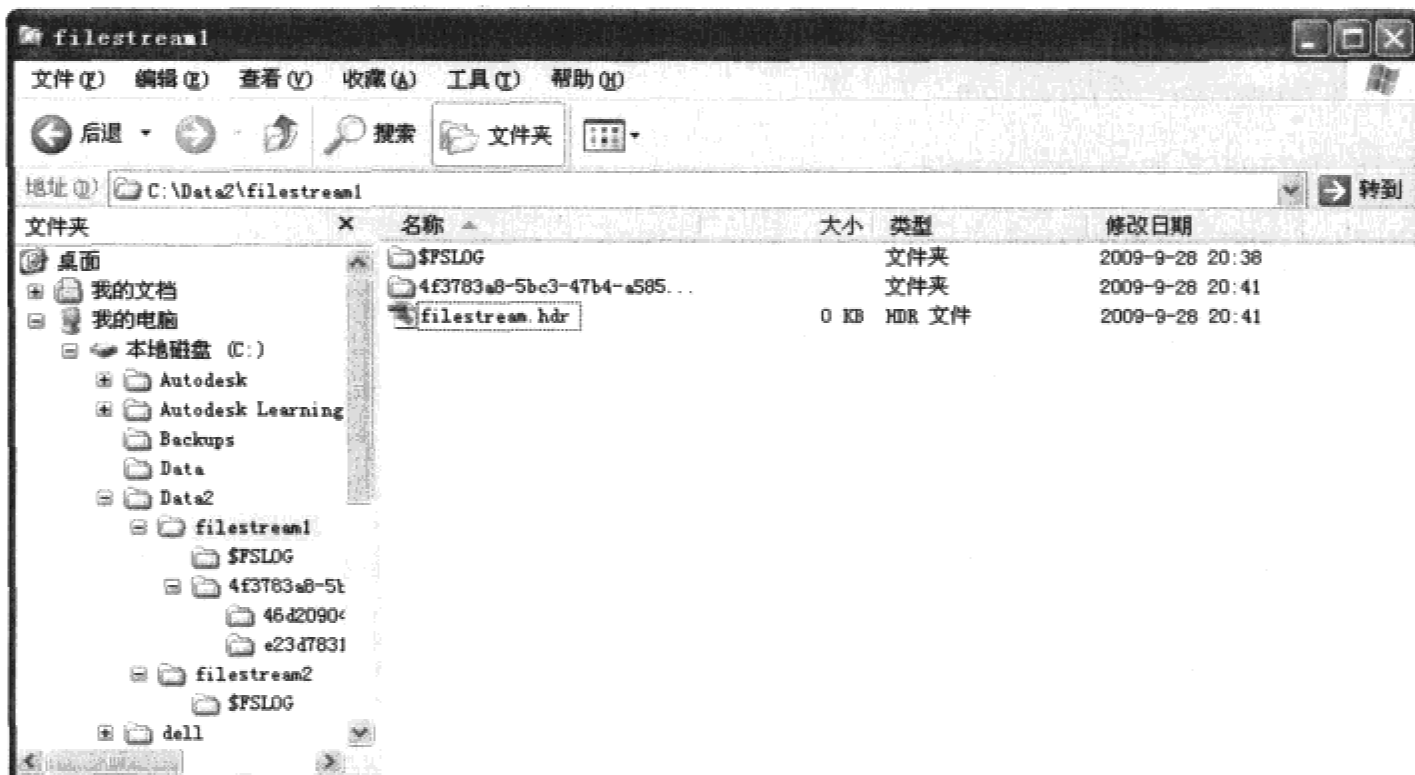


图 7-4　创建一个具有两个文件流数据列的表之后的操作系统文件结构

直到真正向表插入文件流数据时才会添加文件。

警告：

当表被删除时，文件夹、子文件夹和它们所包含的文件不会立即从文件系统中删除。它们是通过垃圾收集线程删除的，垃圾收集线程定期启动，在 SQL Server 服务停止和重启时也会启动。您可以手动删除文件，但是需要谨慎。可以删除某个在数据库中仍然存在的列或行的文件夹，即使该数据库是联机的。接下来对该表的访问会产生一条包含“没有找到路径”的错误消息。您可能认为 SQL Server 应该防止作为数据库一部分的任何文件被删除，但是为了完全阻止文件删除，SQL Server 必须为所有文件流容器中整个数据库使用的每个文件保持开放的文件句柄，对于大型表来说，这是不实际的。

7.2.4　操纵文件流数据

文件流数据通过 T-SQL 或 Win32 API 操作。使用 T-SQL 时，数据的处理方式与 *varbinary(MAX)*完全相同。使用 Win32 API 要求您首先获得文件路径及当前事务的上下文。接下来可以打开 Win32 句柄，使用该句柄读取和写入大型对象数据。这一部分的所有示例使用的都是 T-SQL。您可以从 *SQL Server 联机丛书*上获得有关 Win32 操作的详细信息。

向表添加数据时，文件会被添加到每一列的子文件夹中。出现运行时错误的 *INSERT* 操作仍然会为行中每个文件流列创建一个文件。虽然该行永远都不能被访问，但是它仍然占用文件系统空间。

1. 插入文件流数据

数据可以使用标准的 T-SQL *INSERT* 语句进行插入。文件流数据必须使用 *varbinary(MAX)*数据类型进行插入，但是任何字符串数据都可以在 *INSERT* 语句中转换。下面的语句向原来创建的 *dbo.Records* 表（有

两个文件流列）添加一行。第一个文件流列将一个 90 000 字节的字符串转换成 *varbinary(MAX)*，第二个文件流列获得一个空的二进制字符串。注意，我们首先将 9 个字符的字符串基数据转换成 *varchar(MAX)*，因为一个标准的字符串值不能超过 8000 个字节。*REPLICATE* 函数返回的数据类型与它的第一个参数的类型相同，因此我们希望第一个参数必须是一个大型对象。复制 9 字节字符串 10 000 次会生成一个 90 000 字节的字符串，该字符串接下来被转换成 *varbinary(MAX)*：

```
USE MyFileStreamDB
INSERT INTO dbo.Records
     SELECT newid (), 24,
       CAST (REPLICATE (CONVERT(varchar(MAX), 'Base Data'), 10000)
                            AS varbinary(max)),
       0x;
```

注意值 0x 是一个空二进制字符串，与 NULL 不同。*FILESTREAM* 列中有非 MULL 值的每一行都有一个文件，即使是零长度的值也是如此。

图 7-5 显示了运行前面的代码创建具有两个文件流容器的数据库，并创建一个包含两个 *FILESTREAM* 列的表同时向表插入一行后文件系统的效果。在左侧窗格中，可以看到两个文件流容器（*filestream*1 和 *filestream*2）。

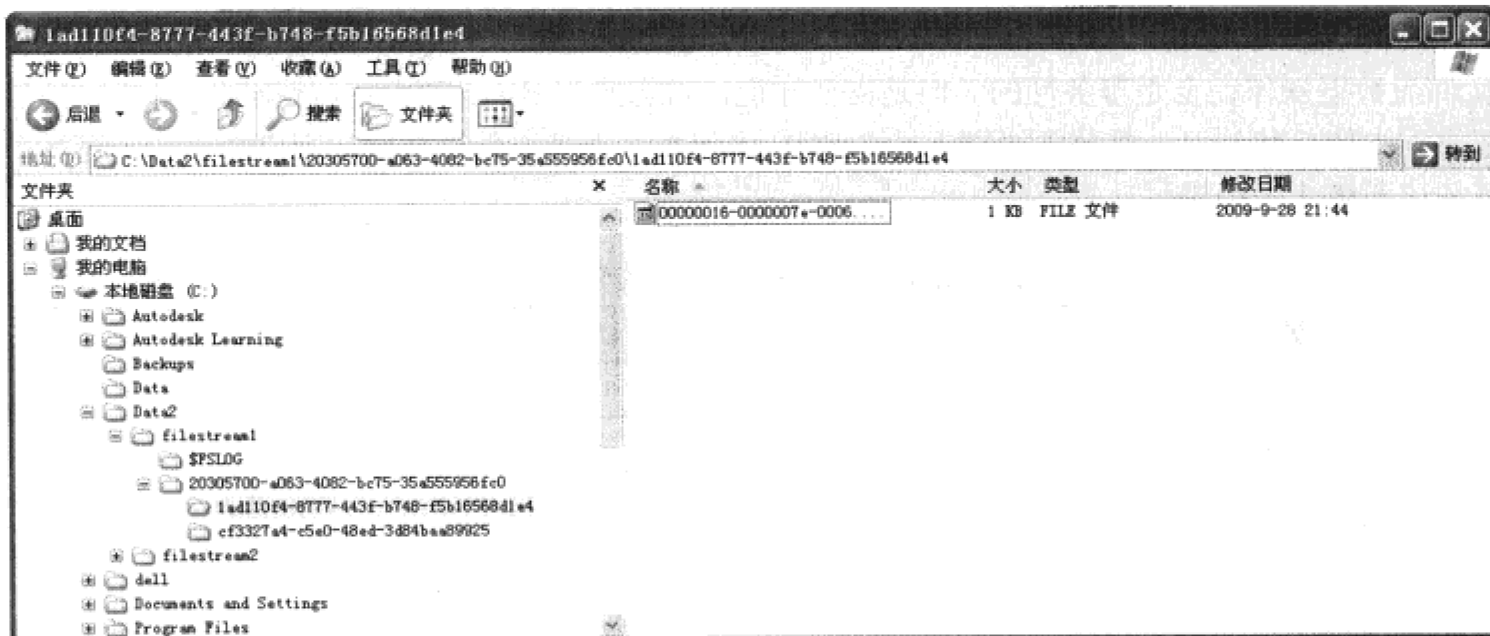


图 7-5 插入文件流数据后操作系统文件的结构

*Filestream*1 容器为我们创建的 *dbo.Records* 表提供了一个具有 GUID 名称的文件夹，同时该文件夹容器有两个具有 GUID 名称的文件夹（用于存储表中的两个列）。右侧窗格显示包含插入到其中一列中的实际数据的文件。

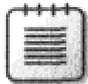
2. 更新文件流数据

当一条 *UPDATE* 语句被用于修改一个文件流列时，包含该数据的文件被修改，同时该文件按照适当的大小进行增减。具体来说，将该列设置为一个空的零长度值会使文件大小为零。同时，在第一个版本中不支持带有 *.WRITE* 子句的 T-SQL“块更新”。建议您使用文件系统流访问文件流数据操作（插入和更新）。对 *FILESTREAM* 数据的更新总是通过 *DELETE* 语句后跟一条 *INSERT* 语句执行的，因此可以看到存储更新列的目录中有一个新行。

当一个文件流单元被设置为NULL时，与该单元相关的文件流文件将在运行垃圾收集线程时被删除（我们将在本章后面介绍垃圾收集）。

3. 删除文件流数据

如果一行是使用*DELETE*或*TRUNCATE* TABLE语句删除的，则与该行相关的任何*FILESTREAM*文件都会被删除。但是，文件的删除与行的删除不同步。文件是通过*FILESTREAM*垃圾收集线程删除的。这同样适合于作为一条更新语句一部分而产生的 *DELETE*。一个新行将被添加，但是旧行不会被物理删除，直到垃圾收集运行。



注意：

数据处理操作（*INSERT*、*UPDATE*、*DELETE* 和 *MERGE*）的 OUTPUT 子句与列修改的OUTPUT子句受支持的方式相同。但是，如果使用OUTPUT子句向具有*varbinary(MAX)*列的表（而不是文件流说明符）进行插入，则需要小心。如果文件流数据大于2GB，则在表中进行文件流数据的插入可能会出现运行时错误。

4. 文件流数据和事务

文件流数据操作完全是事务性的。但是需要知道，当您操作*FILESTREAM*数据时，并不是所有分离级别都得到支持。此外，一些分离级别支持T-SQL访问而不支持文件流访问。表7-2显示了哪种分离级别在哪种访问模式中可用。

表7-2　　文件数据流数据操作支持的分离级别

分 离 级 别	T-SQL 访问	文件系统访问
未提交读	支持	不支持
已提交读	支持	支持
可重复读	支持	不支持
可序列化	支持	不支持
已提交读快照	支持	不支持
快照	支持	不支持

如果试图访问同一*FILESTREAM*数据文件的两个进程处于不兼容的模式，则文件流API会失败并返回ERROR_SHARING_VIOLATION消息而不是仅仅阻塞（使用T-SQL时会发生阻塞）。与所有数据访问一样，同一个事务中的读—写永远不会在同一个文件上发生冲突，但是与非*FILESTREAM*访问不同的是，同一事务中的两个写操作可能最终会由于彼此访问相同的文件而发生冲突，除非文件句柄原来已经被关闭。您可以在第10章了解更多关于事务、分离级别及冲突的内容。

5. 记录文件流更改

正如前面提到的那样，每个 *FILESTREAM* 文件组都有一个跟踪所有涉及该文件组的文件流活动的$FSLOG文件夹。可以在数据库（包含*FILESTREAM*文件组）中执行事务日志备份和还原操作时及在恢复过程中使用该文件夹中的数据。

$FSLOG 文件夹主要跟踪添加到文件流文件组中的新信息。一个文件将被添加到日志文件夹中来反映如下信息。

- 包含创建的文件流数据的一个新表。
- 一个 *FILESTREAM* 列被定义。
- 一个新行被插入，其中包含 *FILESTREAM* 列中的非 NULL 数据。
- 一个 *FILESTREAM* 值被更新。
- 出现一次 *COMMIT*。

下面是一些示例。

- 如果创建一个包含两个文件流列的表，则会有 4 个文件添加到$FSLOG 文件夹——一个用于存储表，两个用于存储列，还有一个用于表示隐含的 *COMMIT*。
- 如果在一个自动提交事务中插入一个包含文件流数据的行，则会有两个文件被添加到$FSLOG 文件夹中——一个用于 *INSERT*，另一个用于 *COMMIT*。
- 如果在一个显式事务中插入 5 行，则会有 6 个文件被添加到$FSLOG 文件夹中。

数据被删除、表被截断或删除时不会向$FSLOG 文件夹添加文件。但是，SQL Server 事务日志将跟踪这些操作，同时有一个新的元数据表包含已经被删除的数据信息。

6. 文件流数据的垃圾收集

文件流数据可以被认为是服务于现场用户数据、对这些数据进行修改的日志及快照操作的行版本(将在第 10 章介绍)。如果任何备份或恢复有可能需要文件流数据文件，则 SQL Server 需要确定文件流数据文件没有被删除。尤其对于日志备份来说，由于事务日志不包含实际的文件流数据，因此所有新文件流内容必须被备份，同时只有文件流数据有实际 *FILESTREAM* 内容的重做信息。一般来说，如果数据库不是处于 SIMPLE 恢复模式，则在垃圾收集从 *FILESTREAM* 文件夹中删除不需要的数据文件之前，需要备份日志两次。现在让我们来看一个实例。我们将从一个新的标记开始，通过删除和重建 *MyFilestreamDB* 数据库实现。一个 *DROP DATABASE* 语句会立即删除所有文件夹和文件，因为现在没有机会执行任何后续的日志备份。这里给出的脚本将重建数据库并创建一个只有一个 *FILESTREAM* 列的表。最后，该脚本将向表插入 3 行数据并对数据库进行备份。如果您检查 *filestream1* 容器，会发现存储列的文件夹包含用于存储 3 行数据的 3 个文件：

```
USE master;
GO
DROP DATABASE MyFilestreamDB;
GO
CREATE DATABASE MyFilestreamDB ON PRIMARY
      (NAME = N'Rowdata1', FILENAME = N'c:\data\Rowdata1.mdf' , SIZE = 2304KB ,
       MAXSIZE = UNLIMITED, FILEGROWTH = 1024KB ),
FILEGROUP FileStreamGroup1 CONTAINS FILESTREAM DEFAULT
   (NAME = N'FSData1', FILENAME = N'c:\data\filestream1' ),
FILEGROUP FileStreamGroup2 CONTAINS FILESTREAM
   (NAME = N'FSData2', FILENAME = N'c:\data\filestream2' )
LOG ON
   (NAME = N'FSDBLOG', FILENAME = N'c:\data\FSDB_log.ldf' , SIZE = 1024KB ,
    MAXSIZE = 2048GB , FILEGROWTH = 10%);
GO
USE MyFilestreamDB;
```

```
GO
CREATE TABLE dbo.Records
(
        Id [uniqueidentifier] ROWGUIDCOL NOT NULL UNIQUE,
        SerialNumber INTEGER UNIQUE,
        Chart_Primary VARBINARY(MAX) FILESTREAM NULL
)
FILESTREAM_ON FileStreamGroup1;
GO
INSERT INTO dbo.Records
    VALUES (newid(), 1,
              CAST (REPLICATE (CONVERT(varchar(MAX), 'Base Data'),
                      10000) as varbinary(max))),
            (newid(), 2,
               CAST (REPLICATE (CONVERT(varchar(MAX), 'New Data'),
                       10000) as varbinary(max))),
             (newid(), 3, 0x);
GO
BACKUP DATABASE MyFileStreamDB to disk = 'C:\backups\FBDB.bak';
GO
```

现在利用下面的语句删除其中的一行：

```
DELETE dbo.Records
WHERE SerialNumber = 2;
GO
```

现在检查磁盘上的文件，您还会看到3个文件。

备份日志并运行一个检查点。注意在实际的系统中，可能会对数据进行大量的修改，数据库的日志满得足以触发一个自动 *CHECKPOINT*。但是，在测试期间，我们没有在日志中放入太多内容，因此我们必须强制 *CHECKPOIT*：

```
BACKUP LOG MyFileStreamDB to disk = 'C:\backups\FBDB_log.bak';
CHECKPOINT;
```

现在如果检查 *FILESTREAM* 数据文件，还会看到3行。等待5秒钟的垃圾收集后，还是看到3行。我们需要备份日志，然后强制另一个 *CHECKPOINT*：

```
BACKUP LOG MyFileStreamDB to disk = 'C:\backups\FBDB_log.bak';
CHECKPOINT;
```

现在在5秒钟之内应该看到其中一个文件消失。在垃圾收集可以获得物理文件之前备份日志两次是为了保证文件空间不被其他文件流操作重新使用，同时仍然可用于还原。

您可以运行一些自己的其他测试。例如，如果想删除 *dbo.Records* 表，注意在 SQL Server 删除表和列的文件夹之前，必须再次执行两次日志备份和 *CHECKPOINT*。

7.2.5 文件流数据的元数据

在 SQL Server 表内，文件流所需的存储不是特别复杂。在行本身内，每个文件流列包含一个40字节大小的文件指针。即使您使用 *DBCC PAGE* 命令查看一个数据页，也没有关于可用文件的太多信息。但是，SQL Server 确实提供了一种函数用于将文件指针转换成路径名。该函数实际上是应用到表中列名称

上的一种方法。因此下面的代码将为包含前面插入行中实际列数据的文件返回一个 UNC 名称。

```
SELECT Chart_Primary, Chart_Primary.PathName()
FROM dbo.Records
WHERE SerialNumber = 24;
GO
```

UNC 值返回的格式如下：

```
\\<server_name>\<share_name>\v1\<db_name>\<object_schema>\<table_name>\<column_name>\<GUID>
```

使用 *PathName* 函数时记住如下几点。

- 函数名称是区分大小写的，即使在一个不区分大小写的服务器上也是如此，因此必须输入 *PathName*。
- 默认的 *share_name* 是 SQL Server 实例的服务名称(因此对于默认实例来说是 MSSQLSERVER)。使用 SQL Server 配置管理器时，可以用右键单击 SQL Server 实例，然后选择“属性”。弹出的“SQL Server 属性”对话框的 *FILESTREAM1* 选项卡允许将 *share_name* 修改为您选择的另一个值。
- *PathName* 函数可以有一个取值为 0、1 或 2 的可选参数，0 为默认值。该参数仅用于控制如何返回 *server_name* 值。UNC 字符串中的所有其他值都不受影响。表 7-3 显示了不同值的含义。

表 7-3　*PathName* 函数的参数值

值	说　明
0	返回已经转换成 BIOS 格式的服务器名称；例如，\\SERVERNAME\MSSQLSERVER\v1\MyFilestream\dbo\Records\Chart_Primary\A73F19F7-38EA-4AB0-BB89-E6C545DBD3F9
1	返回没有转换的服务器名称，例如：\\ServerName\MSSQLSERVER\v1\MyFilestream\Dbo\Records\ Chart_Primary\A73F19F7-38EA-4AB0-BB89-E6C545DBD3F9
2	返回完整的服务器路径，例如：\\ServerName.MyDomain.com\MSSQLSERVER\v1\MyFilestream\Dbo\Records\Chart_Primary\A73F19F7-38EA-4AB0-BB89-E6C545DBD3F9

SQL Server 2005 中添加的一些元数据已经被增强，能够为您提供文件流数据的有关信息。

- ***sys.database_files***。为每个文件流文件返回一行。这些文件的 *type* 值为 2，*type_desc* 值为 *FILESTREAM*。
- ***sys.filegroups***。为每个文件流文件组返回一行。这些文件的 *type* 值为 FD，*type_desc* 值为 *FILESTREAM_DATA_FILEGROUP*。
- ***sys.data_spaces***。为每个数据空间（可以是一个文件组或一种分区方案）返回一行。保存文件流数据的文件组用类型 FD 表示。
- ***sys.tables***。列中有一个值用于 *filestream_data_space_id*，这是文件流数据所使用的文件流文件组或分区方案的数据空间 ID。对于没有文件流数据的表来说，该列值为 NULL。
- ***sys.columns***。*is_filestream* 列中具有 *filestream* 属性的列值为 1。

旧的元数据（如系统程序 *sp_helpdb* <数据库名称> 或 *sp_help*<对象名称>）不显示有关文件流数据的任何信息。

前面提到过，被删除的行或对象不会在$FSLOG 文件夹中生成文件，但是关于已删除数据的数据被存储在一个系统表中。没有任何元数据视图允许您查看这张表，您只能利用专用管理员连接（DAC）进行查看。可以在名为 *sys.internal_tables* 的视图中查看名称中有 *TOMBSTONE* 的对象。接下来可以利用 DAC

查看 *TOMBSTONE* 表内部的数据。如果重新运行上面的脚本但不备份日志，则可以使用如下的脚本：

```
USE MyFilestreamDB;
GO
SELECT name FROM sys.internal_tables
WHERE name like '%tombstone%';

-- I see the table named: filestream_tombstone_2073058421
-- Reconnect using DAC, which puts us in the master database
USE MyFileStreamDB;
GO
SELECT * FROM sys.filestream_tombstone_2073058421;
GO
```

如果该表是空的，则 SQL Server 中的日志和$FSLOG 同步，同时所有不需要的文件已经从磁盘上的 *FILESTREAM* 容器中删除。

7.2.6 文件流数据性能方面的考虑

虽然详细讨论性能调节和检测超出了本书的范围，但是我们希望为您提供关于设置系统以获得文件流数据良好性能方面的一些基本信息。Paul Randal 是本书的作者之一，他曾经写过一本关于 *FILESTREAM* 的白皮书，可以通过 MSDN 站点（*http://msdn.microsoft.com/en-us/library/cc949109.aspx*）进行访问（这一白皮书同样可以在本书的配套站点 *http://www.SQLServerInternals.com/companion* 上找到）。在这一部分，我们只简单地介绍 Paul 关于如何获得良好性能的一些主要观点。所有这些建议在白皮书中都进行了详细的解释。

- 确认以正确的方式存储适当大小的数据。Jim Gray 几年前发表了一篇题为"To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem?"的研究论文。从他的调查结果中可以看出，小于256KB 的大型对象应该存储在数据库中，1MB 或更大的数据应该存储在文件系统中。对于这两种之间的数据来说，具体如何存放由其他因素决定，同时您应该完整地测试应用程序。这里的关键是：如果使用 *FILESTREAM* 存储很多比较小的大型对象，则不会获得很好的性能。
- 为存储 *FILESTREAM* 数据容器的 NTFS 卷使用合适的 RAID 级别。例如，不要为写密集型工作负载使用 RAID-5。
- 使用合适的磁盘技术。SCSI 通常比 SATA/IDE 更快，因为 SCSI 驱动器通常具有更高的旋转速度，因此它们有较小的等待时间和搜索时间。但是，SCSI 驱动也更贵。
- 不论您选择哪种磁盘技术，如果磁盘是 SATA，则请保证它支持 NCQ；如果是 SCSI，则请保证它支持 CTQ。两者都允许驱动器并发处理多个交错的 I/O 操作。
- 使数据容器彼此独立，同时使容器与数据库数据和日志文件彼此独立。这样可以避免竞争磁盘头。
- 在设置 *FILESTREAM* 之前根据需要对 NTFS 卷进行碎片整理，同时定期进行碎片整理以保持良好的扫描性能。
- 使用命令行 *fsutil* 工具关闭 NTFS 卷上的 8.3 名称生成。这是一种 order-N 算法，该算法必须检查生成的新名称与目录中现有的名称不冲突。但是，请注意这将大大降低插入和更新的性能。
- 使用 *fsutil* 关闭对最后访问时间的跟踪。
- 适当设置 NTFS 聚集的大小。对于大小大于 1MB 的大型对象来说，可使用 64KB 的聚集大小来减少碎片。

- 对 *FILESTREAM* 数据的局部更新会生成一个新的文件。将很多小型更新批处理成一个大型更新从而减少改动。
- 当数据流返回客户端时，使用约 60KB 或 60KB 倍数的 SMB 缓冲大小。这样可以帮助防止缓冲区出现太多的碎片，因为传输控制协议/Internet 协议（TCP/IP）缓冲区的大小是 64KB。

考虑这些建议并对应用程序执行完全测试可以在使用非常大型的数据对象时提供最佳性能。

7.3 稀疏列

这一部分将介绍 SQL Server 2008 中新增的另一种特殊的存储格式。稀疏列是用一种优化 NULL 值存储格式的普通列。稀疏列降低 NULL 值的空间需求，允许您在表的定义中存储更多列，只要这些列大部分为 NULL。使用稀疏列的代价是：需要更多的系统开销来存储和检索非 NULL 值。

稀疏列用于存储描述实体（有很多合适属性）的数据表。例如，Microsoft Windows SharePoint Services 之类的目录管理系统可能需要跟踪单个表中很多不同类型的数据。不同的属性应用到表中不同的行字节中。因此对于每个行来说，只有列的一小部分子集填充了值。另一种查看方法适用于任何具体的属性，只有行的一个子集具有该属性的一个值。SQL Server 2008 中的稀疏列允许我们为一行存储非常大的合适列。因此，稀疏列功能有时也被称为*宽表*功能。

7.3.1 稀疏列的管理

除非表至少有 90%的行的某一列值可能为 NULL，否则不建议将该列定义为稀疏列。这不是一种强制限制，并且您几乎可以把任意列定义为稀疏列。稀疏列节省了 NULL 值的空间。

SQL Server 2008 的这种新功能允许您拥有比以往更多的列。表中列数的限度是 30 000 个，其中非稀疏列不能超过 1024 列（计算列被认为是非稀疏列）。显然，并不是所有 30 000 列都可以有值。填充列数量由行中数据的字节数决定。稀疏列优化 NULL 值的存储大小，NULL 值在稀疏列中根本不占用任何空间，这一点与非稀疏列不同（非稀疏列中的 NULL 也需要空间）。正如我们在第 5 章看到的那样，一个固定长度的 NULL 列总会占用整个列宽，一个变长的 NULL 列至少占用列偏移阵列中的 2 字节。虽然稀疏列本身不占用空间，但是需要一些固定的系统开销允许行中存在稀疏列。只要您定义具有 SPARSE（稀疏）属性的列，SQL Server 就会向行末尾添加一个稀疏向量。我们将在本章后面的“物理存储”一节介绍稀疏向量的实际结构，但是首先您应该知道即使有稀疏列，一个数据行的最大长度（不包括 LOB 和行溢出）仍然是 8060（包括系统开销字节）。由于稀疏向量包括附加系统开销，因此其他行的最大字节数会降低。此外，行中所有固定长度的非 NULL 稀疏列的长度最大仍然是 8019 字节。

1. 表的创建

创建一个具有稀疏列的表非常简单，只需向任何数据类型的列（除 *text*、*ntext*、*image*、 *geography*、*geometry*、*timestamp* 或用户定义数据类型）添加 SPARSE 属性。此外，稀疏列不能是聚集索引或主键的一部分。包含稀疏列的表不能被压缩，不论是行级还是叶级（我们将在下一节详细介绍压缩方面的内容）。还有其他几个限制，尤其是在利用稀疏列对表进行分区时，因此您应该查看文档了解所有详细信息。这一部分的示例非常简单，因为打印具有很多列的代码示例从而使稀疏列真正有效是不切实际的。下面的示例显示了两个非常相似的表的创建，其中一个表不允许稀疏列，另一个表允许稀疏列。我们试图向每张表插入相同的行。因为允许稀疏列的行具有更小的最大长度，因此当我们试图插入一行时，有稀疏列

的表会插入失败，而没有稀疏列的表则不会出现问题：

```
USE test;
GO
CREATE TABLE test_nosparse
(
   col1 int,
   col2 char(8000),
   col3 varchar(8000)
);
GO
INSERT INTO test_nosparse
        SELECT null, null, null;
INSERT INTO test_nosparse
        SELECT 1, 'a', 'b';
GO
```

插入这两行时不会出现错误。现在建立第二个表：

```
CREATE TABLE test_sparse
(
 col1 int SPARSE,
 col2 char(8000) SPARSE,
 col3 varchar(8000) SPARSE
);
GO
INSERT INTO test_sparse
        SELECT NULL, NULL, NULL;
INSERT INTO test_sparse
        SELECT 1, 'a', 'b';
GO
```

第二条 *INSERT* 语句会产生如下的错误：

```
Msg 576, Level 16, State 5, Line 2
Cannot create a row that has sparse data of size 8042 which is greater than the allowable
maximum sparse data size of 8019.
```

虽然插入到 *test_sparse* 表中的第二行看起来与成功插入到 *test_nosparse* 表中的行一样，但是从系统内部来看则不同。稀疏列的总大小是：4 字节用于 *int*、8000 字节用于 *char*、24 字节用于行溢出指针，这已经超出了 8019 字节的限制。

2. 修改表

可以将表的一个非稀疏列转换成一个稀疏列，或者将一个稀疏列转换成一个非稀疏列。但是请注意，如果您正在修改没有稀疏列的表中一个非常大的行，则将某一列修改为稀疏列会减小页面上允许的数据字节数。这样可能导致现有列转换成稀疏列时出现错误。例如，下面的代码将创建一个有大量行的表，*INSERT* 语句(不论有没有 NULL)会被接受。但是，当我们希望使某一列成为稀疏列时，即使 8 字节 *datetime* 列这样较小的列也会由于额外的系统开销而使现有行太大，从而导致 *ALTER* 语句执行失败：

```
IF EXISTS (SELECT * FROM sys.tables WHERE name = 'test_nosparse_alter')
                  DROP TABLE test_nosparse_alter;
```

```
GO
CREATE TABLE test_nosparse_alter
(
c1 int,
c2 char(4020) ,
c3 char(4020) ,
c4 datetime
);
GO
INSERT INTO test_nosparse_alter SELECT NULL, NULL, NULL, NULL;
INSERT INTO test_nosparse_alter SELECT 1, 1, 'b', GETDATE();
GO
ALTER TABLE test_nosparse_alter
  ALTER COLUMN c4 datetime SPARSE;
```

我们会得到如下错误消息：

```
Msg 1701, Level 16, State 1, Line 2
Creating or altering table 'test_nosparse_alter' failed because the minimum row size would
be 8075, including 23 bytes of internal overhead. This exceeds the maximum allowable table
row size of 8060 bytes.
```

一般来说，可以像对待其他列一样对待稀疏列，只是稀疏列有一些限制罢了。除前面提到不能定义为 SPARSE 的数据类型的限制外，还要记住如下一些限制。

- 稀疏列不能有默认值。
- 稀疏列不能与某种规则绑定。
- 虽然计算列可以使用一个稀疏列，但是计算列不能被标记为 SPARSE。
- 稀疏列不能是聚集索引或唯一主键索引的一部分。但是，引用稀疏列的持久计算列和非持久计算列都可以是聚集键的一部分。
- 稀疏列不能用做聚集索引或堆的一个分区键。但是，稀疏列可以用做非聚集索引的分区键。

注意，除了稀疏列不能是聚集索引或主键一部分的要求之外，在稀疏列上建立索引时没有其他方面的限制。但是，如果您正在以预期的方式使用稀疏列，并且大多数行中稀疏列的值为 NULL，则稀疏列上的任何常规索引都是非常低效的并且用途有限。稀疏列应该与筛选索引一起使用，我们已经在第 6 章对此做过介绍。

7.3.2 列集和稀疏列操作

如果打算使用稀疏列，则每行中只有几列有值，并且 *INSERT* 和 *UPDATE* 语句比较简单。对于 *INSERT* 语句来说，您可以指定一个列的列表，然后只为该列列表中的几列指定值。对于 *UPDATE* 语句来说，每一行中只能对几列值进行操作。只有在选择没有列出个别列的数据时（即使用 *SELECT* *），才需要关心如何处理可能非常大的列的列表。优秀的开发人员知道使用 *SELECT* *不是一种很好的习惯，但是 SQL Server 需要一种能够处理可能有几千列结果集（或者几万列）的方法。用于帮助处理 *SELECT* *的机制称为 COLUMN_SET 结构。COLUMN_SET 是一种无类型的 XML 表示法，能够将表中的多个列结合到一个结构化的输出中。您可以认为一个 COLUMN_SET 是一种非持久型计算列，因为 COLUMN_SET 物理不存储在表中。在 SQL Server 的版本中，只有 COLUMN_SET 包含表中的所有稀疏列。将来的版本可能允许定义其他 COLUMN_SET 变体。

一个表只能定义一个 COLUMN_SET，同时一旦表定义了一个 COLUMN_SET，*SELECT**将不再返回单个稀疏列。相反会返回一个包含所有稀疏列非空 NULL 值的 XML 片段。现在我们来看一个示例。

下面的代码建立一个包含一个标识列、25个稀疏列和一个列集的表：

```
USE test;
GO
IF EXISTS (SELECT * FROM sys.tables WHERE name = 'lots_of_sparse_columns')
                  DROP TABLE lots_of_sparse_columns;
GO
CREATE TABLE lots_of_sparse_columns
(ID int IDENTITY,
 col1 int SPARSE,
 col2 int SPARSE,
 col3 int SPARSE,
 col4 int SPARSE,
 col5 int SPARSE,
 col6 int SPARSE,
 col7 int SPARSE,
 col8 int SPARSE,
 col9 int SPARSE,
 col10 int SPARSE,
 col11 int SPARSE,
 col12 int SPARSE,
 col13 int SPARSE,
 col14 int SPARSE,
 col15 int SPARSE,
 col16 int SPARSE,
 col17 int SPARSE,
 col18 int SPARSE,
 col19 int SPARSE,
 col20 int SPARSE,
 col21 int SPARSE,
 col22 int SPARSE,
 col23 int SPARSE,
 col24 int SPARSE,
 col25 int SPARSE,
 sparse_column_set XML COLUMN_SET FOR ALL_SPARSE_COLUMNS);
 GO
```

接下来我们向25列中的3列插入值，指定单个列名称：

```
INSERT INTO lots_of_sparse_columns (col4, col7, col12) SELECT 4,6,11;
```

也可以直接在COLUMN_SET中进行插入，指定XML片段中的列值。可以更新COLUMN_SET是区分COLUMN_SET和计算列的另一特征：

```
INSERT INTO lots_of_sparse_columns (sparse_column_set)
                SELECT '<col8>42</col8><col17>0</col17><col22>30000</col22>';
```

下面是对该表进行 *SELECT* *操作时得到的结果：

```
SELECT * FROM lots_of_sparse_columns;
Results:
ID       sparse_column_set
-------  ------------------------------------------------
1        <col4>4</col4><col7>6</col7><col12>11</col12>
2        <col8>42</col8><col17>0</col17><col22>30000</col22>
```

还可以从单个列中进行查询，或者替代或者还选择整个 COLUMN_SET。因此下面的两条 *SELECT* 语句都是有效的：

```
SELECT ID, col10, col15, col20
    FROM lots_of_sparse_columns;
SELECT *, col11
    FROM lots_of_sparse_columns;
```

如果您决定在表中使用稀疏列，请记住以下几点。

- 一旦定义，COLUMN_SET 就不能被修改。为了修改 COLUMN_SET，必须删除并重建 COLUMN_SET 列。
- COLUMN_SET 可以被添加到不包含任何稀疏列的表上。如果稀疏列后来被添加到表上，则它们会显示在列集中。
- COLUMN_SET 是可选的，不需要使用稀疏列。
- 不能在 COLUMN_SET 上定义约束或默认值。
- 包含 COLUMN_SET 的表不支持分布式查询。
- 复制不支持 COLUMN_SET。
- 变更数据捕获功能不支持 COLUMN_SET。
- COLUMN_SET 不能是任何索引类型的一部分，包括 XML 索引、全文索引和索引视图。COLUMN_SET 不能作为任何索引中的包含性列被添加。
- COLUMN_SET 不能在筛选索引或筛选统计信息的筛选器表达式中使用。
- 当一个视图包含一个 COLUMN_SET 时，COLUMN_SET 作为 XML 的一列出现在视图中。
- XML 数据的大小限制是 2GB。如果行中所有非空稀疏列的数据超过这一限制，则操作会产生错误。
- 从具有 COLUMN_SET 的表中复制所有列（使用 *SELECT * INTO* 或 *INSERT INTO SELECT **）不会复制单个稀疏列。只有 XML 数据类型的 COLUMN_SET 被复制。

现在我们来看一下稀疏列的实际存储方式。

7.3.3 物理存储

在一种高级别上，您可以认为稀疏列与使用 COLUMN_SET 显示的存储方式相同，即一组对（列名，值）。因此如果某一特殊列没有值，则该列不会被列出同时也根本不需要任何空间。如果该列有值，则 SQL Server 不仅要存储该值而且还需要存储关于哪一列拥有该值的信息。因此非空稀疏列比空列占用更大的空间。为了形象地说明它们的区别，将表 7-4 和表 7-5 进行比较。

表 7-4 表示一个有非稀疏列的表。您可以看到，当大部分列为空时有很多空间被浪费。表 7-5 表示当除 ID 列之外的所有列都被定义为 SPARSE 时同一表的显示效果。所存储的是所有非空列的名称和它们的值。

表 7-4　用非 SPARSE 列定义并且有很多空值的表的表示

ID	sc1	sc2	sc3	sc4	sc5	sc6	sc7	sc8	sc9
1	1								9
2		2		4					
3						6	7		
4	1				5				
5				4				8	

续表

ID	sc1	sc2	sc3	sc4	sc5	sc6	sc7	sc8	sc9
6			3						9
7					5		7		
8		2						8	
9			3			6			

表 7-5　用 **SPARSE** 列定义并且有很多空值的表的表示

ID	稀疏列	ID	稀疏列
1	(sc1,sc9)(1,9)	6	(sc3,sc9)(3,9)
2	(sc2,sc4)(2,4)	7	(sc5,sc7)(5,7)
3	(sc6,sc7)(6,7)	8	(sc2,sc8)(2,8)
4	(sc1,sc5)(1,5)	9	(sc3,sc6)(3,6)
5	(sc4,sc8)(4,8)		

SQL Server 利用行中一种称为*稀疏向量*的结构跟踪 SPARSE 列的物理存储。稀疏向量仅在至少声明了一个稀疏列并且这些表中的每个数据记录都包含稀疏向量的基表数据记录中存在。

稀疏向量可以作为一个特殊变长列存储在数据记录末尾。它是一种特殊的系统列，*sys.columns* 和其他任何视图中都没有该列的元数据。稀疏向量作为行中最后一个变长列存储。稀疏向量后面只有版本信息，通常与快照分离一起使用，具体内容我们将在第 10 章介绍。NULL 位图中没有任何位用于存储稀疏向量列，但是行中变量列数量包含稀疏向量。这时可能需要查看第 5 章的图 5-10，温习一下数据行的一般结构。

表 7-6 列出了稀疏向量中字节的含义。

表 7-6　稀疏向量中字节的含义

名称	字节数	含义
Complex Column Header（复杂列标题）	2	值 05 表示该复杂列是一个稀疏向量
Sparse Column Count（稀疏列计数）	2	稀疏列数量
Column ID Set（列 ID 集）	2*稀疏列数量	2 字节用于存储表中每一列的列 ID，值存储在稀疏向量中
Column Offset Table（列偏移表）	2*稀疏列数量	2 字节用于存储每个稀疏列结束位置的偏移量
Sparse Data（稀疏数据）	由实际值决定	数据

现在我们来看一下包含 SPARSE 列的行的字节。首先，建立一个包含两个稀疏列的表，并填充 3 行数据：

```
USE test;
GO
IF EXISTS (SELECT * FROM sys.tables WHERE name = 'sparse_bits')
                DROP TABLE sparse_bits;
GO
CREATE TABLE sparse_bits
(
c1 int IDENTITY,
```

```
c2 varchar(4),
c3 char(4) SPARSE,
c4 varchar(4) SPARSE
);
GO
INSERT INTO sparse_bits SELECT 'aaaa', 'bbbb', 'cccc';
INSERT INTO sparse_bits SELECT 'dddd', null, 'eeee';
INSERT INTO sparse_bits SELECT 'ffff', null, 'gg';
GO
```

现在可以利用 *DBCC IND* 查找存储这 3 行的数据页的页码并利用 *DBCC PAGE* 查看该页上的字节：

```
DBCC IND(test, sparse_bits, -1);
GO
-- The output indicated that the data page for my table was on page 289;
DBCC TRACEON(3604);
DBCC PAGE(test, 1, 289, 1);
```

我们没有显示整个页的输出内容，只显示了第一行的输出（3 行输出）：

```
00000000:  30000800 01000000 02000002 00150029 †0..............)
00000010:  80616161 61050002 00030004 00100014 †.aaaa...........
00000020:  00626262 62636363 63†††††††††††††††††.bbbbcccc
```

灰色的字节表示稀疏向量。很容易找到稀疏向量，因为它刚好位于最后非稀疏变长列（包含 aaaa 或 61616161）之后并持续到行尾。图 7-6 根据表 7-6 给出的含义对稀疏列进行了转换。在转换之前不要忘了对数字字段进行字节交换。例如，前两个字节 05 00 需要交换来获得十六进制值 0x0005。接着可以将它转换成十进制。

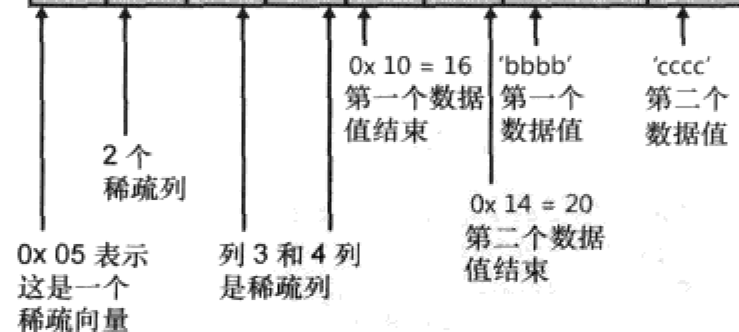


图 7-6　对 *sparse_bits* 表中某一行的稀疏向量值的实际字节进行解释

您可以对该页上其他两行中的字节进行同样的分析。这里有几点需要注意。

- 具有空值的列信息不会出现在稀疏向量中。
- 在稀疏向量中，固定长度和变长字符串的存储没有区别。但是，这并不表示您应该互换使用这两种字符串。不适合 8060 字节的 SPARSE *varchar* 列可以作为行溢出数据存储，而 SPARSE *char* 列则不能。
- 由于只有 2 字节用于存储稀疏列的编号，因此限定了稀疏列的最大数量。
- 复杂列标题的 2 字节表示可能有其他复杂列。此时，可以被存储的唯一另一种复杂类型是存储一个转发指针，与 SQL Server 创建转发记录时执行的操作相同（我们在第 5 章介绍堆更新时介绍过转发记录）。

7.3.4 元数据

只需要非常少的额外数据来支持稀疏列。目录视图 *sys.columns* 包含如下两列用于跟踪表中的稀疏列。其中的每一列都只有两种可能的取值，即 0 或 1。

- *is_sparse*
- *is_column_set*

与 *sys.columns* 中的这些列属性对应，属性函数 *COLUMNPROPERTY()*也有如下两个与稀疏列相关的属性。

- *IsSparse*
- *IsColumnSet*

如果我们要检查已经创建的名称中有“sparse”的所有表并确定其中哪些列是稀疏列、哪些是列集或者两者都不是，可以运行如下查询：

```
SELECT OBJECT_NAME(object_id) as 'Table', name as 'Column', is_sparse, is_column_set
FROM sys.columns
WHERE OBJECT_NAME(object_id) like '%sparse%';
```

如果只想查看所有 COLUMN_SET 列的表和列名称，则可以运行如下查询：

```
SELECT OBJECT_NAME(object_id) as 'Table', name as 'Column'
FROM sys.columns
WHERE COLUMNPROPERTY(object_id, name, 'IsColumnSet') = 1;
```

7.3.5 利用稀疏列节省存储空间

当大部分值为 NULL 时，使用稀疏列可以节省相当大的空间。实际上正如我们前面提到的那样，不为 NULL 但被定义为 SPARSE 的列会比没有定义为 SPARSE 的列占用更多的空间，因为稀疏向量必须存储一些额外的字节来跟踪这些列。为了查看空间区别，可以运行如下脚本，该脚本创建具有比较短的固定长度列的 4 个表。其中两个表有稀疏列，两个表没有稀疏列。利用一个循环向每个表插入 100 000 行数据。具有稀疏列的一个表用 NULL 值填充行，另一个表用非空值填充行。没有稀疏列的表用 NULL 值填充行，另一个表用非 NULL 值填充行：

```
USE test;
GO
SET NOCOUNT ON;
GO
IF EXISTS (SELECT * FROM sys.tables
                         WHERE name = 'sparse_nonulls_size')
                DROP TABLE sparse_nonulls_size;
GO
CREATE TABLE sparse_nonulls_size
(col1 int IDENTITY,
 col2 datetime SPARSE,
 col3 char(10) SPARSE
 );
GO
IF EXISTS (SELECT * FROM sys.tables
                         WHERE name = 'nonsparse_nonulls_size')
                DROP TABLE nonsparse_nonulls_size;
GO
CREATE TABLE nonsparse_nonulls_size
(col1 int IDENTITY,
 col2 datetime,
 col3 char(10)
```

```
);
GO
IF EXISTS (SELECT * FROM sys.tables
                        WHERE name = 'sparse_nulls_size')
                DROP TABLE sparse_nulls_size;
GO
CREATE TABLE sparse_nulls_size
(col1 int IDENTITY,
 col2 datetime SPARSE,
 col3 char(10) SPARSE
 );
GO
IF EXISTS (SELECT * FROM sys.tables
                        WHERE name = 'nonsparse_nulls_size')
                DROP TABLE nonsparse_nulls_size;
GO
CREATE TABLE nonsparse_nulls_size
(col1 int IDENTITY,
 col2 datetime,
 col3 char(10)
 );
GO
DECLARE @num int
SET @num = 1
WHILE @num < 100000
BEGIN
  INSERT INTO sparse_nonulls_size
        SELECT GETDATE(), 'my message';
  INSERT INTO nonsparse_nonulls_size
        SELECT GETDATE(), 'my message';
  INSERT INTO sparse_nulls_size
        SELECT NULL, NULL;
  INSERT INTO nonsparse_nulls_size
        SELECT NULL, NULL;
  SET @num = @num + 1;
END;
GO
```

现在我们来看一下每个表中的页数量。下面的元数据查询为 4 个表中的每一个返回 *sys.allocation_units* 视图中数据页的数量：

```
SELECT object_name(object_id) as 'table with 100K rows', data_pages
FROM sys.allocation_units au
    JOIN sys.partitions p
        ON p.partition_id = au.container_id
WHERE object_name(object_id) LIKE '%sparse%size';
```

下面是运行结果：

```
table with 100K rows       data_pages
------------------------   ----------
sparse_nonulls_size        610
nonsparse_nonulls_size     402
sparse_nulls_size          169
nonsparse_nulls_size       402
```

注意，当表中有值为 NULL 的稀疏列时，需要最少数量的页。如果表没有稀疏列，则空间的使用在列有 NULL 值和没有 NULL 值的情况下是相同的，因为数据被定义为固定长度。这种空间要求是值为 NULL 的稀疏列所需空间的两倍多。最坏的情况是列被定义为 SPARSE 但是不存在 NULL 值。

当然，前面的示例是一种极端情况，即所有数据都为 NULL 或都不为 NULL 并且都是固定长度的数据类型。因此，虽然我们可以说 SPARSE 列比没有定义为 SPARSE 的列需要更多的存储空间用于存储非 NULL 值，但是实际的空间节省情况是由数据类型和空行的百分比决定的。表 7-7 是对 *SQL Server 联机丛书*的复制，显示了每种数据类型的空间使用情况。*NULL 百分比*列表示为了净节省 40%的空间必须有多大百分比的数据为空。表 7-7 显示了 SQL Server 2008 中各种数据类型的空间节省情况。

表 7-7 SPARSE 列的存储需求

数据类型	没有 SPARSE 的存储字节	有 SPARSE 和非 NULL 时的存储字节	NULL 值百分比
固定长度数据类型			
bit	0.125	4.125	98%
tinyint	1	5	86%
smallint	2	6	76%
int	4	8	64%
bigint	8	12	52%
real	4	8	64%
float	8	12	52%
smallmoney	4	8	64%
money	8	12	52%
smalldatetime	4	8	64%
datetime	8	12	52%
uniqueidentifier	16	20	43%
date	3	7	69%
精确度随长度变化的数据类型			
datetime2(0)	6	10	57%
datetime2(7)	8	12	52%
time(0)	3	7	69%
time(7)	5	9	60%
datetimeoffset(0)	8	12	52%
datetimeoffset(7)	10	14	49%
decimal/numeric(1,s)	5	9	60%
decimal/numeric(38,s)	17	21	42%
数据随长度变化的数据类型			
sql_variant	变化		
varchar 或 *char*	4+avg.data	2+avg.data	60%
nvarchar 或 *nchar*	4+avg.data	2+avg.data	60%
varbinary 或 *binary*	4+avg.data	2+avg.data	60%
xml	4+avg.data	2+avg.data	60%
hierarchyid	4+avg.data	2+avg.data	60%

一般性的建议是当您希望提供至少 20%～40%的空间节省时，应该考虑使用稀疏列。

7.4 数据压缩

SQL Server 2008 提供了数据压缩的功能，这是一种只在企业版本中才有的功能。压缩通过去除真实数据中存在的低效性来降低表的大小。这些低效性通常可以分成两类。第一类与单个数据值（当它们存储在利用最大可能大小定义的列中时）的存储有关。例如，表可能需要一个 *int* 类型的 *quantity* 列，因为您偶尔可能存储大于 32 767（*smallint* 类型的最大值）的值。但是，*int* 列总是需要 4 字节，同时如果 *quantity* 值大部分都不足 100 字节，则可以存储在 *tinyint*（只需要 1 字节）列中。SQL Server 的行压缩功能可以压缩数据的单独列，从而只使用所需的最小空间总量。

数据存储中出现的第二种低效性是页面上的数据包含重复值或跨列和行的公有前缀时。这种低效性可以通过只存储重复值一次，然后从其他列引用这些值的方式得到降低。SQL Server 页压缩功能通过维护包含公有前缀或重复值的项来压缩页面上的数据。SQL Server 同样始终应用行压缩。

7.4.1 Vardecimal

SQL Server 2005 SP3 引入了一种简单的压缩格式，这种格式可以只应用到使用 *decimal* 数据类型定义的列上（记住 *numeric* 数据类型与 *decimal* 完全等价，提到 *decimal*，也表示它是 *numeric*）。在 SQL Server 2005 中，该选项是数据库级（使用 *sp_db_vardecimal_storage_format* 程序）和表级（使用 *sp_tableoption* 程序）都要启用的选项。在 SQL Server 2008 中，所有用户数据库都自动对 *vardecimal* 存储格式启用，因此 *vardecimal* 必须只为个别表启用。与 SQL Server 2008 中的数据压缩类似，*vardecimal* 存储格式只在 SQL Server 企业版本中可用，我们将在本部分对压缩进行详细介绍。

在 SQL Server 2005 中，一旦这两个存储过程被运行，表中为 *vardecimal* 启用的 *decimal* 数据将以不同的方式存储。*decimal* 列不作为固定长度数据对待，而是存储在行的可变部分并且只使用所需的字节数量（我们已经在第 5 章了解了固定长度数据和变长数据存储之间的差异）。除使用 *vardecimal* 存储所有 *decimal* 数据的所有表分区之外，表上的所有索引都自动使用 *vardecimal* 格式。

decimal 数据值利用 1～38 之间的一个精确值定义，同时根据定义的精确度不同，*decimal* 数据可能占用 5～17 字节。固定长度的 *decimal* 数据为每行使用相同的字节数，即使实际数据占用的字节数很少。当一个表没有使用 *vardecimal* 存储格式时，表中每一项为每一个定义的 *decimal* 列使用同样的字节数，即使某行的值为 0、NULL 或者是可以用更少字节来表示的某个值，如数字 3。当为表启用 *vardecimal* 存储格式时，每一行中的 *decimal* 列会使用存储指定值所需的最小空间。当然，正如我们在第 5 章看到的那样，每一个变长列都有与之相关的 2 字节额外开销，但是当在定义为具有较大精确度的 *decimal* 列中存储一个非常小的值时，节省的空间完全能够补偿这些附加的 2 字节。对于 *vardecimal* 存储来说，NULL 和 0 作为零长度数据存储并且只占用 2 字节的系统开销。

虽然 SQL Server 2008 支持 *vardecimal* 格式，但是建议您在希望降低数据行所需的存储空间时使用行压缩。用于启用 *vardecimal* 存储的表选项和数据库选项都已经被废弃。

7.4.2 行压缩

您可以认为行压缩是 *vardecimal* 存储格式的一种扩展。很多情况下，SQL Server 可能会使用比存储数据值所需空间更大的空间，在没有 SQL Server 2008 企业版的情况下，您唯一的办法是使用变长数据类

型。任何固定长度的数据类型总会对表中的每一行使用相同的空间长度，即使浪费空间。例如，可以将某列声明为 *int* 类型，因为偶尔可能需要存储大于 32 000 的值。一个 *int* 需要 4 字节的存储空间，不论其中存储的值是什么（即使该列为 NULL）。只有 *character* 和 *binary* 数据可以存储在变长列中（当然还有 *decimal*，如果该选项被启用的话）。行压缩允许 *integer* 值仅使用所需的存储空间，最小值为 1 字节。值 100 只需要 1 字节来存储，值 1000 需要 2 字节。有一种优化允许 0 和 NULL 不使用任何空间来存储数据本身。我们将在本节后面详细介绍这一优化。

1. 启用行压缩

创建表或索引时可以启用压缩，或者使用 *ALTER TABLE* 或 *ALTER* INDEX 命令启用压缩。此外，如果表或索引被分区，则可以选择只压缩分区的一个子集（我们将在本章后面介绍分区）。

下面的脚本在 *AdventureWorks2008* 数据库中创建 *dbo.Employees* 表的两个副本。存储行压缩的数据时，SQL Server 存储小于或等于 8 字节（短列）和大于 8 字节（长列）的值的方式是不同的。因此，脚本会更新新表中的一行，从而使列中不包含超过 8 字节的数据。*Employees_rowcompressed* 表启用了行压缩，*Employees_uncompressed* 表没有被压缩。分配给每个表的元数据查询会针对每个表进行查询，从而比较行压缩前后的大小：

```
USE AdventureWorks2008;
GO
IF EXISTS (SELECT * FROM sys.tables
         WHERE name = 'Employees_uncompressed')
             DROP TABLE Employees_uncompressed;
GO
SELECT e.BusinessEntityID, NationalIDNumber, JobTitle,
         BirthDate, MaritalStatus, VacationHours,
         FirstName, LastName
  INTO Employees_uncompressed
  FROM HumanResources.Employee e
   JOIN Person.Person p
        ON e.BusinessEntityID = p.BusinessEntityID;
GO
UPDATE Employees_uncompressed
SET NationalIDNumber = '1111',
         JobTitle = 'Boss',
         LastName = 'Gato'
WHERE FirstName = 'Ken'
AND LastName = 'Sánchez';
GO
ALTER TABLE dbo.Employees_uncompressed
    ADD CONSTRAINT EmployeeUn_ID
        PRIMARY KEY (BusinessEntityID);
GO
SELECT OBJECT_NAME(object_id) as name,
         rows, data_pages, data_compression_desc
FROM sys.partitions p JOIN sys.allocation_units au
         ON p.partition_id = au.container_id
WHERE object_id = object_id('dbo.Employees_uncompressed');

IF EXISTS (SELECT * FROM sys.tables
         WHERE name = 'Employees_rowcompressed')
```

```
            DROP TABLE Employees_rowcompressed;
GO
SELECT BusinessEntityID, NationalIDNumber, JobTitle,
        BirthDate, MaritalStatus, VacationHours,
        FirstName, LastName
  INTO Employees_rowcompressed
  FROM dbo.Employees_uncompressed
GO
ALTER TABLE dbo.Employees_rowcompressed
   ADD CONSTRAINT EmployeeR_ID
       PRIMARY KEY (BusinessEntityID);
GO
ALTER TABLE dbo.Employees_rowcompressed
REBUILD WITH (DATA_COMPRESSION = ROW);
GO
SELECT OBJECT_NAME(object_id) as name,
        rows, data_pages, data_compression_desc
FROM sys.partitions p JOIN sys.allocation_units au
        ON p.partition_id = au.container_id
WHERE object_id = object_id('dbo.Employees_rowcompressed');
GO
```

我们将在本章后面再次引用 *dbo.Employees_rowcompressed* 表，或者您可以在我详细介绍压缩行存储时亲自检查该表。

现在看一下行压缩的细节，但是请记住以下几点。

- 行压缩只有在 SQL Server 2008 企业版本和开发人员版本中可用。
- 行压缩不会修改表或索引的行最大长度。
- 行压缩不能在有稀疏列的表中启用。
- 如果某个表或索引已经被分区，则行压缩可以在所有分区上启用，也可以在分区的一个字节上启用。

2. 新行格式

在第 5 章中，我们看到从 SQL Server 7.0 开始就已经使用并且在 SQL Server 2008 中仍然在使用（如果没有启用压缩）的行存储格式。这种格式称为 *FixedVar* 格式，因为这种格式有一个独立于变长数据部分的固定长度数据部分。SQL Server 2008 中引入了一种全新的行格式用于存储压缩行，这种格式称为 *CD* 格式。CD 代表“列说明符”，该术语是指每一列中都在行本身内包含说明信息。可能需要重新查看第 5 章中的图 5-10 来回忆 *FixedVar* 的格式并与新的 CD 格式进行比较。图 7-7 显示了 CD 格式的一种抽象。它很难像图 5-10 那样具体，因为除标题外，每个区域中的字节数都是由行中的数据决定的。

图 7-7 CD 记录的一般结构

我们将对每一部分进行详细介绍。

标题：行标题总是一个字节，大致与第 5 章中所说的“状态位 A”对应。每一位的含义如下：

位 0。表示记录类型，值为 1 表示是新的 CD 记录格式。

位 1。表示该行包含版本控制信息。

位 2～位 4。当做一个 3 位值，这些位表示行中存储哪些类型的信息。可能值如下：

- 000——主记录；
- 001——备份空记录；
- 010——转发记录；
- 011——备份数据记录；
- 100——已转发记录；
- 101——备份已转发记录；
- 110——索引记录；
- 111——备份索引记录。

位 5。表示该行包含一个长数据区域（长度大于 8 字节）。

位 6～位 7。SQL Server 2008 中不使用。

CD 区域：CD 区域由两部分组成。第一部分是 1 或 2 字节，表示短列的数量。如果第一个字节中最重要的一位设置为 0，则是一个最大值为 127 的 1 字节字段。如果列数大于 127，则最重要的一位为 1，同时 SQL Server 使用 2 字节表示列数，最多可达 32 767 列。

下面表示列数的 1 或 2 字节是 CD 阵列。CD 阵列使用 4 位表示表中的每一列，以表示有关列长度的信息。4 位可以有 16 种不同的取值，但是在 SQL Server 2008 中，只有其中的 13 种取值被使用。

- 0（0x0）表示对应列为空。
- 1（0x1）表示对应列为一个 0 字节短值。
- 2（0x2）表示对应列是一个 1 字节短值。
- 3（0x3）表示对应列是一个 2 字节短值。
- 4（0x4）表示对应列是一个 3 字节短值。
- 5（0x5）表示对应列是一个 4 字节短值。
- 6（0x6）表示对应列是一个 5 字节短值。
- 7（0x7）表示对应列是一个 6 字节短值。
- 8（0x8）表示对应列是一个 7 字节短值。
- 9（0x9）表示对应列是一个 8 字节短值。
- 10（0xa）表示对应列是一个长数据值并且不占用短数据区域中的任何空间。
- 11（0xb）用于值为 1 的 *bit* 类型列。相应列不占用短数据区域的任何空间。
- 12（0xc）表示相应列是一个 1 字节的符号，表示页目录中的一个值（我们将在本章后面“页压缩”一节介绍目录）。

短数据区域：短数据区域不需要存储每个短数据值的长度，因为这些信息可以在 CD 区域中得到。但是，如果表中有几百列，则访问最后一列的代价可能很大。为了降低代价，列可以被分成聚集，其中每个聚集包含 30 列，并且在短数据区域的开始部分有一个被称为短数据聚集阵列的区域。每个阵列项都是一个单字节的 *integer*，表示短数据区域中上一个聚集中所有数据的长度之和，因此该值基本上是指向聚集第一列的一个指针。短数据的第一个聚集从聚集阵列后开始，因此不需要聚集偏移。但是在一个聚集中可能不是 30 个数据列，因为只有长度小于或等于 8 字节的列才会存储在短数据区域中。

例如，假设一行有 64 列，并且列 5、10、15、20、25、30、40、50 和 60 都是长数据，其他列为短数据。则 CD 区域包含如下内容。

- 1 字节，包含 CD 区域中的列数量 64。
- 一个 4×64 位或 32 字节的 CD 阵列，包含有关每列长度的信息。55 项有值，表示短数据的一个真实数据长度，0xa 的 8 项表示长数据。
- 短数据区域包含如下内容。
  - 一个短数据聚集偏移阵列，包含两个值，每个值包含一个短数据聚集的长度。在本例中，第一个聚集（前 30 列中的所有短数据）的长度为 92，因此偏移阵列中的 92 表示第二个聚集是从第一个聚集后 92 字节处开始的。聚集数量可以用（列数–1）/30 来计算。如果所有 30 列都是长度为 8 字节的短数据，那么聚集阵列中每一项的最大值为 240。
  - 所有短数据值。

图 7-8 显示了 CD 区域及具有示例数据（前面介绍的行使用的）的短数据区域。CD 阵列被完整地显示，同时用一个符号表示每个 64 值的长度。因此阵列可能适合本书的页面，实际的数据值没有显示。第一个聚集在短数据区域中的值是 24（6 是长值），第二个聚集在短数据区域中的值是 27（3 是长值），第三个聚集拥有剩下的 4 列（都是短值）。我们接下来将介绍长值的存储。

<table>
<tr><th colspan="2">CD 区域</th><th colspan="5">短数据区域</th></tr>
<tr><td rowspan="2">列数</td><td>CD 阵列—64 个 4 位值
（'a' 表示长列）</td><td colspan="2" rowspan="2">每 30 列聚集中短数据的长度
(N–1)/30 值</td><td colspan="3" rowspan="2">实际数据
的 3 个聚集</td></tr>
<tr><td rowspan="2">3285a4358a6543a3456a6666a5463a
254372644a745269277a463495736a
5433</td></tr>
<tr><td>N = 64</td><td>92</td><td>106</td><td>24
值</td><td>27
值</td><td>4
值</td></tr>
</table>

图 7-8 CD 记录中的 CD 区域和短数据区域

为了在短数据区域中定位短数据列值对应的项，短数据聚集阵列首先被检查，从而确定包含用于短数据区域中列的聚集的起始地址。

长数据区域：大于 8 字节的列中的所有数据都存储在长数据区域中。其中包括复杂列，不包含实际数据而是包含定位行外存储数据所需的信息。可能包括大型对象数据和行溢出数据指针。与短数据（长度可以简单地存储在 CD 阵列中）不同的是，长数据需要一个实际偏移值来允许 SQL Server 确定每个值的位置。这个偏移阵列与第 5 章中介绍 *FixedVar* 记录时介绍的偏移阵列很像。

长数据区域由 3 部分构成：一个偏移阵列、一个长数据聚集阵列和长数据。

偏移阵列由如下部分组成。

- 一个 1 字节标题。在 SQL Server 2008 中，只有前两位被使用。位 0 表示长数据区域中是否包含 2 字节偏移值，同时在 SQL Server 2008 中该值始终为 1，因为所有偏移都是 2 字节。位 1 表示长数据区域是否包含复杂列。
- 一个表示接下来偏移量的 2 字节值。偏移值第一个字节中的最重要一位用于表示长数据区域中的相应项是否是一个复杂列。阵列项中的其他位/字节存储长数据区域中相应项的结束偏移值。

长数据聚集阵列与短数据聚集阵列类似，用于限制查找一个很长的列的列表末尾位置附近列的代价。每个 30 列的聚集（最后一个聚集除外）有一项。由于我们已经将每个长数据列的偏移存储在偏移阵列中，因此聚集阵列只需要跟踪每个聚集中有多少个长数据值。每个值是一个 1 字节整数，表示该聚集中长数据列的数量。与短数据聚集类似，聚集阵列中的项数可用（表中的列数–1）/30 进行计算。

图 7-9 显示了用于存储前面介绍的行（64 列，其中 9 列是长数据）的长数据区域。由于空间的关系，我们没有实际包含偏移值，长数据聚集阵列有两项，表示值 6 位于第一个聚集中，值 2 位于第二个聚集中。其他值位于最后一个聚集中。

偏移阵列			长数据聚集阵列		长数据								
标题	项 #	偏移项	每个 30 列聚集中的项数 (N–1)/30 值		长数据 1	长数据 2	长数据 3	长数据 4	长数据 5	长数据 6	长数据 7	长数据 8	长数据 9
01	09		06	02									

图 7-9 一条 CD 记录的长数据区域

特殊信息：行末尾包含 3 条可选信息。其中任何一条或所有信息的存在都会在行开始的 1 字节标题中表示出来。3 个特殊的区域如下。

- ***转发指针***。该值在某个堆包含一个转发存根（指向原始行已经被移动到的新位置）时使用。转发指针已经在第 5 章介绍过。转发指针包含 3 个标题字节和一个 8 字节行 ID。
- ***回调指针***。该值在已经被转发的某个行中使用，以指示行的原始位置。作为一个 8 字节的行 ID 存储。
- ***版本控制信息***。当某一行在快照分离模式下被修改时，SQL Server 将向行添加 14 字节的版本控制信息。行版本控制和快照分离将在第 10 章介绍。

现在看一下前面创建的 *dbo.Employees_rowcompressed* 表中某两行的实际字节。DBCC *PAGE* 命令已经增强了关于压缩行和页附加信息的功能。尤其是在行字节显示之前，*DBCC PAGE* 将显示 CD 阵列。对于 *dbo.Employees_rowcompressed* 表中第一个页面返回的第一行来说，所有列都包含短数据。行包含如下数据值：

BusinessEntityID	NationalIDNumber	JobTitle	BirthDate	MaritalStatus	VacationHours	FirstName	LastName
1	1111	Boss	1959-03-02	S	99	Ken	Gato

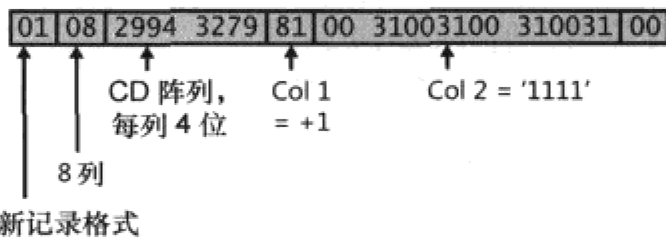
对于短数据来说，CD 阵列包含每一列的实际长度，我们可以在 *DBCC PAGE* 输出中看到第一行的信息：

```
CD array entry = Column 1 (cluster 0, CD array offset 0): 0x02 (ONE_BYTE_SHORT)
CD array entry = Column 2 (cluster 0, CD array offset 0): 0x09 (EIGHT_BYTE_SHORT)
CD array entry = Column 3 (cluster 0, CD array offset 1): 0x09 (EIGHT_BYTE_SHORT)
CD array entry = Column 4 (cluster 0, CD array offset 1): 0x04 (THREE_BYTE_SHORT)
CD array entry = Column 5 (cluster 0, CD array offset 2): 0x03 (TWO_BYTE_SHORT)
CD array entry = Column 6 (cluster 0, CD array offset 2): 0x02 (ONE_BYTE_SHORT)
CD array entry = Column 7 (cluster 0, CD array offset 3): 0x07 (SIX_BYTE_SHORT)
CD array entry = Column 8 (cluster 0, CD array offset 3): 0x09 (EIGHT_BYTE_SHORT)
```

因此第一列的 CD 代码为 0x02，代表一个 1 字节值，正如我们在数据行中看到的那样，这是一个整数 1。第二列包含一个 8 字节的值，并且是 Unicode 字符串 1111。剩下列的代码留给您自己去检查。

图 7-10 显示了行目录的 *DBCC PAGE* 输出，同时我们已经说明了不同字节的含义。

首页上返回的第二行具有如下数据值的几个长列。



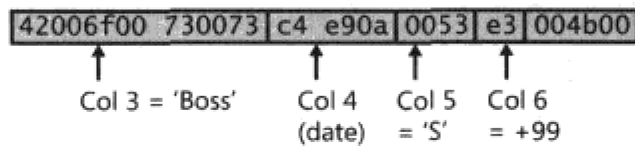


图 7-10 具有 8 个短型数据列的压缩行

Bus...	NationalIDNumber	JobTitle	BirthDate	Marital...	Vacation...	FirstName	LastName
2	245797967	Vice President of Engineering	1961-09-01	S	1	Terri	Duffy

该行的 CD 阵列效果如下：

```
CD array entry = Column 1 (cluster 0, CD array offset 0): 0x02 (ONE_BYTE_SHORT)
CD array entry = Column 2 (cluster 0, CD array offset 0): 0x0a (LONG)
CD array entry = Column 3 (cluster 0, CD array offset 1): 0x0a (LONG)
CD array entry = Column 4 (cluster 0, CD array offset 1): 0x04 (THREE_BYTE_SHORT)
CD array entry = Column 5 (cluster 0, CD array offset 2): 0x03 (TWO_BYTE_SHORT)
CD array entry = Column 6 (cluster 0, CD array offset 2): 0x02 (ONE_BYTE_SHORT)
CD array entry = Column 7 (cluster 0, CD array offset 3): 0x0a (LONG)
CD array entry = Column 8 (cluster 0, CD array offset 3): 0x0a (LONG)
```

注意 8 列中的 4 列是长数据值。

图 7-11 显示了 *DBCC PAGE* 为第二个数据行返回的字节。

```
Record Memory Dump
6294C08B:   2108a24a 23aa8256 ed0a5300 81010400 †!.¢J#ª,Vi.S.....
6294C09B:   12004c00 56006000 32003400 35003700 †..L.V.`.2.4.5.7.
6294C0AB:   39003700 39003600 37005600 69006300 †9.7.9.6.7.V.i.c.
6294C0BB:   65002000 50007200 65007300 69006400 †e. .P.r.e.s.i.d.
6294C0CB:   65006e00 74002000 6f006600 20004500 †e.n.t. .o.f. .E.
6294C0DB:   6e006700 69006e00 65006500 72006900 †n.g.i.n.e.e.r.i.
6294C0EB:   6e006700 54006500 72007200 69004400 †n.g.T.e.r.r.i.D.
```

图 7-11 具有 4 个短数据列和 4 个长数据列的一个压缩行

我们已经强调了长数据部分的字节。下面需要对行的第一部分（长数据区区域之前）进行几点说明。

- 行中第一个字节是 0x21，说明该行不仅是新 CD 记录格式，而且该行包含一个长数据区域。
- 第二个字节表示表中有 8 列，与第一行相同。
- CD 阵列接下来的 8 字节中有 4 个 a 值，说明短数据区域中不包含长值。
- 短数据值在 CD 阵列后按如下顺序列出：
  - *BusinessEntityID* 占 1 字节，值为 0x82 或+2；
  - *Birthdate* 占 3 字节；

- ❑ *MaritalStatus* 占 1 字节，值为 0x0053 或'S'；
- ❑ *VacationHours* 占 1 字节，值为 0x81 或+1。

长数据区域偏移阵列的长度是 10 字节，含义如下：

- 第一个字节是 0x01，表示行偏移位置是 2 字节长；
- 第二个字节是 0x04，表示在长数据区域中有 4 列；
- 接下来的 8 字节是用于 4 个值的 2 字节偏移。注意偏移参照长数据区域本身的结束位置。
  - ❑ 第一个 2 字节偏移是 0x0012 或 18。这表示第一个长值是 18 字节长（是 9 个字符的 Unicode 字符串 245797967，可能需要 18 字节）。
  - ❑ 第二个 2 字节偏移是 0x004c 或 76，表示第二个长值在第一个长值的 58 字节后结束。第二个值是 *Vice President of Engineering*，这是一个 29 字节的 Unicode 字符串。
  - ❑ 第三个 2 字节偏移是 0x0056 或 96，表示第三个值 *Terri* 是 10 字节长。
  - ❑ 第四个 2 字节偏移是 0x0060 或 96，表示第四个值 *Duffy* 是 10 字节长。

由于总共列数少于 30 列，因此没有长数据聚集阵列，同时数据值紧跟在长数据区域偏移阵列后存储。

由于空间限制，这里不对具有多列聚集的行（超过 30 列）进行详细介绍，但是希望您自己能够对这样的行进行研究。

7.4.3 页压缩

除了以某种压缩格式存储行来降低所需空间外，SQL Server 2008 还可以通过分离和重用页上的字节模式来压缩整个页面。

与行压缩不同的是，只有在页被填满并且 SQL Server 认为压缩页可以节省大量空间的情况下才会应用页压缩（我们将在本节后面详细说明节省的空间量）。在计划进行页压缩时应该遵循如下几条原则。

- 只有在 SQL Server 2008 企业版和开发人员版本中才有页压缩功能。
- 页压缩总是包括行压缩（也就是说，如果对某个表启用了页压缩，则会自动启用行压缩）。
- 压缩一棵 B 树时，只有叶级可以进行页压缩。出于性能方面的考虑，节点级不会被压缩。
- 如果某个表或索引已经被分区，则可以在所有分区或某个分区的子集上启用页压缩。
- 新行被添加时，页压缩得到维护。页压缩算法必须被重新应用到整个页面上并且只有在 SQL Server 判定这样的操作能够带来好处时才会进行。同样，我们在本节后面对此进行详细介绍。

下面的代码创建 *dbo.Employees* 表的另一个副本并对该副本进行页压缩。接下来利用 *DBCC IND* 获取 *dbo.Employees_uncompressed*、*dbo.Employees_rowcompressed* 和 *dbo.Employees_pagecompressed* 表的页面位置和链接信息。然后代码利用获取的信息报告每个表中的数据页数量。

```
IF EXISTS (SELECT * FROM sys.tables
        WHERE name = 'Employees_pagecompressed')
              DROP TABLE Employees_pagecompressed;
GO
SELECT BusinessEntityID, NationalIDNumber, JobTitle,
        BirthDate, MaritalStatus, VacationHours,
        FirstName, LastName
  INTO Employees_pagecompressed
  FROM dbo.Employees_uncompressed
GO
ALTER TABLE dbo.Employees_pagecompressed
   ADD CONSTRAINT EmployeeP_ID
```

```
        PRIMARY KEY (BusinessEntityID);
GO
ALTER TABLE dbo.Employees_pagecompressed
REBUILD WITH (DATA_COMPRESSION = PAGE);
GO
SELECT OBJECT_NAME(object_id) as name,
                rows, data_pages, data_compression_desc
FROM sys.partitions p JOIN sys.allocation_units au
                ON p.partition_id = au.container_id
WHERE object_id = object_id('dbo.Employees_pagecompressed');
GO

TRUNCATE TABLE sp_table_pages;
GO
INSERT INTO sp_tablepages
   EXEC ('DBCC IND(AdventureWorks2008, Employees_pagecompressed, -1)');
INSERT INTO sp_tablepages
   EXEC ('DBCC IND(AdventureWorks2008, Employees_rowcompressed, -1)');
INSERT INTO sp_tablepages
   EXEC ('DBCC IND(AdventureWorks2008, Employees_uncompressed, -1)');
GO
SELECT OBJECT_NAME(ObjectID), count(*) as NumPages
FROM sp_tablepages
WHERE pagetype = 1
GROUP BY OBJECT_NAME(ObjectID);
GO
```

如果运行上面的脚本，会在输出中发现行压缩没有降低小型表的大小，而页压缩则将表从 5 个数据页压缩成 3 个数据页。

SQL Server 可以利用公共值以两种不同的方式对页进行压缩，即*列前缀压缩*和*字典压缩*。

1. 列前缀压缩

顾名思义，列前缀压缩对表中被压缩的数据列进行操作，但是它只关注单个页上的列值。对于每个列来说，SQL Server 选择一个可用于降低该列值所需存储空间的公共前缀。包含该前缀的列中的最长值被选作*锚值*。每一列不是作为真实数据值进行存储，而是作为锚值的一个变量进行存储。下面对此举例说明。假如要进行页压缩的表中的某一列具有如下字符值：

```
DEEM
DEE
FFF
DEED
DEE
DAN
```

SQL Server 可能会发现 DEE 是一个有用的公共前缀，因此 *DEED* 被选作锚值。每一列将存储该列值与锚值之间的差异。差异值存储为两部分值的形式：锚使用的字符数，以及要追加的其他字符。因此 *DEEM* 存储为<3><M>，表示该值的前 3 个字符来自公共前缀，同时在公共前缀后面追加一个字符 M。*DEED* 存储为一个空字符串（但不是 null），表示它与前缀完全匹配。*DEE* 存储为<3>，第二部分为空，因为不需要追加其他字符。列值列表用下面的值代替：

```
DEEM -> <3><M>
DEE -> <3><>
```

```
FFF -> <><FFF>
DEED -> <><>
DEE -> <3><>
DAN -> <1><AN>
```

请记住压缩行以 CD 记录格式存储，因此 CD 阵列值用一个特殊的编码表示值实际上为 NULL。如果替代值为<><>并且编码不表示 NULL，则该值与前缀完全匹配。

SQL Server 对每一列应用前缀检测和值替换算法并创建一个被称为*锚记录*的新行来存储列的锚值。如果没有找到有用的前缀，则锚记录中的值为 NULL，同时该列中的所有值按原样存储。

图 7-12 显示了页压缩之前表中的 6 行数据，接下来显示了创建锚记录并且对实际的数据值进行替换后的 6 行数据效果。

原始数据		
ABCD	DEEM	ABC
ABD	DEE	DEE
ABC	FFF	GHI
AAN	DEED	HHH
NULL	DEE	KLM
ADE	DAN	NOP

列前缀压缩后的数据		
锚记录		
ABCD	DEED	NULL
<><>	<3><M>	ABC
<2><D>	<3><>	DEE
<3><>	<><FFF>	GHI
<1><AN>	<><>	HHH
NULL	<3><>	KLM
<1><DE>	<1><AN>	NOP

图 7-12 列前缀压缩前后

2. 字典压缩

在前缀压缩已经被单独应用到每个列之后，页压缩的第二个阶段将查看页面上的所有值，从而查找任意行任意列的副本，即使这些值已经被编码来反映前缀的使用情况。可以在图 7-11 的下面看到其中的两个值多次出现：<3><>出现三次，<1><AN>出现两次。检测重复值的过程是数据类型不可知的，因此完全不同列中的值以二进制形式表示可能是相同的。例如，一个 1 字节的字符以十六进制形式表示为 0x54，可以被看做是 1 字节整数 84（十六进制形式也表示为 0x54）的一个副本。字典存储为一组符号，其中的每个符号与数据页上的重复值相对应。一旦符号和数据值确定，则每个重复值的出现都会用符号来替换。SQL Server 通过检查 CD 阵列中的编码发现实际存储在列中的值是一个符号而不是一个数据值。被符号替换的值的 CD 阵列值为 0xc。图 7-13 显示了图 7-12 中用符号替换 5 个值之后的数据。

符号字典： [S1] = <1><AN> [S2] = <3><>		
<><>	<3><M>	ABC
<2><D>	[S2]	DEE
[S2]	<><FFF>	GHI
[S1]	<><>	HHH
NULL	[S2]	KLM
<1><DE>	[S1]	NOP

图 7-13 利用字典压缩进行压缩的页

不是压缩表中的每个页都有一个存储前缀的锚记录和一个字典。如果没有有用的前缀值，则该页可能只有一个字典。如果

值的重复次数不够（即利用符号替换不会节省空间），则该页面可能只有一个锚记录。当然，如果页面上的数据中根本没有模式，则可能有些页既没有锚记录也没有字典。

3. 物理存储

当某个页进行页压缩时仅对该页进行一次主结构修改。SQL Server 在被称为*压缩信息*（CI）记录的页眉（字节偏移为 96 或 0x96）后面添加一个隐藏行。CI 记录的结构如图 7-14 所示。

标题	PageModCount	偏移	锚记录	字典

图 7-14 CI 记录的结构

CI 记录在页面的槽阵列中没有对应项，但是它始终位于相同的位置。此外，页眉中的一位表示该页被压缩了，因此 SQL Server 会查找 CI 记录。如果利用 *DBCC PAGE* 随便堆放一个页，则该页的页面信息将包含一个称为 *m_typeFlagBits* 的值。如果该值为 0x80，则表示该页已被压缩。

如果已经创建了 *sp_tablepages* 表（按照前面所介绍的方法），则可以成功运行下面的脚本。该脚本获取在这一部分创建的 *Employees_uncompressed*、*Employees_rowcompressed* 和 *Employees_pagecompresse*d 表中的 *DBCC IND* 信息。接下来显示每个表中第一个数据页的页码。您可以利用该信息检查具有 *DBCC PAGE* 的页面。注意只有 *Employees_pagecompressed* 表所在页将 *m_typeFlagBits* 值设置为 0x80：

```
USE AdventureWorks2008;
GO
TRUNCATE TABLE sp_tablepages;
GO
INSERT INTO sp_tablepages
   EXEC ('DBCC IND(AdventureWorks2008, Employees_pagecompressed, -1)');
GO
INSERT INTO sp_tablepages
   EXEC ('DBCC IND(AdventureWorks2008, Employees_rowcompressed, -1)');
GO
INSERT INTO sp_tablepages
   EXEC ('DBCC IND(AdventureWorks2008, Employees_uncompressed, -1)');
GO
SELECT OBJECT_NAME(ObjectID), PageFID, PagePID
FROM sp_tablepages
WHERE pagetype = 1
  AND PrevPagePID = 0;
GO
DBCC TRACEON(3604);
GO
```

利用 *DBCC PAGE* 查看某个已经进行了页压缩的页会提供有关 CI 记录内容的信息，我们将在检查完每一部分的含义（接下来介绍）后查看其中的部分信息。

标题。标题是一个 1 字节值，用于跟踪 CI 值。位 0 表示版本，在 SQL Server 2008 中始终为 0。位 1 表示 CI 是否有一条锚记录，位 2 表示 CI 是否有一个字典。其他位没有使用。

PageModCount。*PageModCount* 值跟踪对特殊页的修改，在确定是否应该对页压缩进行重新评估及构建一条新 CI 记录时使用。我们将在接下来的一节（讨论页压缩分析时）进一步介绍如何使用这一值。

偏移。偏移中包含帮助 SQL Server 查找字典的值。其中包含一个指示锚记录结束位置页面偏移的值和一个指示 CI 记录本身结束位置页面偏移的值。

锚记录。锚记录看起来与页上的标准 CD 记录完全相同，包括记录标题、CD 阵列、一个短数据区域和一个长数据区域。存储在数据区域中的值是每一列的公共前缀值，其中一些可能为 NULL。

字典。字典区域由 3 部分组成。第一部分是一个 2 字节字段，包含表示字典中目录数的一个数值。第二部分是一个 2 字节项的偏移阵列，表示每个字典项相对于字典数据区起始位置的结束偏移。第三部分包含实际的字典数据项。

记住，每一个字典项都是一个字节串，在标准数据行中用一个符号替代。该符号只是一个从 0 到 N 的整数值。此外，字节串是与数据类型无关的，即它们只是字节。在 SQL Server 确定将哪些重复出现的值存储在字典中之后，将首先按照数据长度对列表进行排序，接下来按照数据值进行排序，然后按照顺序分配这些符号。因此，假设存储在字典中的值为：

```
0x 53 51 4C
0x FF F8
0x DA 15 43 77 64
0x 34 F3 B6 22 CD
0x 12 34 56
```

表 7-8 显示了排序后的字典，其中包含每一项的长度和符号。

表 7-8　　页压缩字典中的值

值	长　度	符　号
0x FF F8	2 字节	0
0x 12 34 56	3 字节	1
0x 53 51 4C	3 字节	2
0x 34 F3 B6 22 CD	4 字节	3
0x DA 15 43 77 64	4 字节	4

字典区域的显示效果如图 7-15 所示。

标题	偏移	字典
5	02 00	0x FF F8
	05 00	0x 12 34 56
	08 00	0x 53 51 4C
	0D 00	0x 34 F3 B6 22 CD
	12 00	0x DA 15 43 77 64

图 7-15　压缩信息记录中的字典区域

注意字典永远不会真正存储符号值。符号值只存储在需要使用字典的数据记录中。由于它们是简单

的整数，因此可以用做偏移列表的一个索引来查找适当的字典替换值。例如，如果页面上的某一行包含字典符号[2]，则 SQL Server 会在偏移列表查找第三项，在图 7-14 中，该项在距字典起始位置 0800 处的偏移位置结束。SQL Server 接下来查找在该字节处结束的值，即 0x 53 51 4C。如果该字节串存储在一个 *char* 或 *varchar* 列（即一个单字节的字符串）中，则与字符串 SQL 对应。

本章前面部分曾经介绍过 *DBCC PAGE* 输出会为您显示压缩行的 CD 阵列。对于压缩的页来说，*DBCC PAGE* 显示 CI 记录及其中关于锚记录的详细信息。此外，使用格式 3 时，*DBCC PAGE* 显示字典项的详细信息。当我们利用格式 3 的 *DBCC PAGE* 获取 *Employees_pagecompressed* 表的首页并将其复制到一个 Microsoft Office Word 文档中时，需要 261 页。毫无疑问，我们不会显示所有输出。即使我们只复制 CI 记录信息，也需要 7 页，这对于本书来说已经太多了。对于具有压缩页的表的 *DBCC PAGE* 输出，留给您课后自己完成。

4. 页压缩分析

在这一部分，我们详细介绍 SQL Server 如何确定是否压缩页及使用哪些值作为锚记录和字典等方面的内容。行压缩总是在请求时才执行，而页压缩则由可以节省的空间量决定。但是，压缩行的实际工作只能到执行完页压缩后才能进行。由于两种类型的页压缩（前缀替换和字典符号替换）都是利用编码替换真实的数据值，因此直到 SQL Server 确定使用什么编码替换真实数据时才能对行进行压缩。

第一次对某个表或某个分区进行页压缩时，SQL Server 会检查每一个已填充满的页面，从而确定可能的空间节省（不考虑对没有填充满的页面进行压缩）。压缩分析实际创建锚记录、修改所有列以反映锚值并生成字典。接下来对每一行进行压缩。如果新压缩的页至少能再容纳 5 行或比当前页多容纳 25%的行（取大值），则被压缩的页会替换未压缩的页。如果压缩页不能节省这么多的空间，则压缩的页会被抛弃。

当确定对压缩页上的锚记录使用哪些值时，SQL Server 需要查看每一行中的每一个字节，每次查看一列。当 SQL Server 扫描列时，也会跟踪可能的字典项（可以用于多个列中）。可以在一次扫描过程中确定每一列的锚记录值。也就是说，当第一列的所有行的所有字节被检查一次后，SQL Server 会确定该列的锚记录值，或者确定锚记录值不能节省足够的空间。

当 SQL Server 检查每一列时，它会收集可能出现的字典项的一个列表。正如已经介绍过的那样，字典包含在页上出现次数足够多的值，从而使利用一个符号替换这些值就空间而言是划算的。对于每一个可能出现的字典项来说，SQL Server 会跟踪值、值的大小及出现的次数。如果（数据值的大小–1）*（次数–1）–2 大于零，则表示字典替换会节省空间，同时认为该值适用于字典。通常来说，SQL Server 字典中不超过 300 项，因此如果可能有更多的字典项，则会在分析期间按照数量进行排序，同时只有最经常出现的值会在字典中出现。

5. 重建 CI 记录

如果一个表启用了页或行压缩，则新行总会在插入表之前进行压缩。但是，包含锚记录和字典的 CI 记录是在全有或全无的基础上进行重建的。也就是说，在插入新行时，SQL Server 不是只向字典添加某一个新项。当页已经被修改了很多次之后，SQL Server 会估算是否重建 CI 记录。SQL Server 会跟踪对 CI 记录 *PageModCount* 字段中每个页的修改，同时在每次插入、修改或删除一行时更新该值。如果在一次数据修改操作过程中遇到一个填充满的页，则 SQL Server 会确定 *PageModCount* 值。如果 *PageModCount* 值大于 25 或者 *PageModCount/<页面上的行数>* 大于 25%，则 SQL Server 会像第一次压缩某个页时那样应用压缩分析。只有当 SQL Server 确定重新压缩能够至少再存储 5 行（或者比当前页面多存储 25%的数据行）时，新压缩的页面才会替换旧页面。

在 B 树中进行页压缩和在堆中进行页压缩有几点重要区别。

B 树页的压缩。对于 B 树来说，只有叶级是页压缩的。当向一棵 B 树插入一个新行时，如果被压缩的行适合该页面，则只会插入该行，不会执行其他操作。如果该行不适合该页面，则 SQL Server 会试图根据上一部分介绍的条件重新压缩该页。如果重新压缩成功，则表示 CI 记录被修改，因此新行必须被重新压缩同时 SQL Server 会试图向该页插入新行。同样，如果该行适合页，则只是简单地插入。如果新压缩的行不适合该页，那么即使在重新压缩页之后，该页仍然需要拆分。拆分某个已压缩页时，CI 记录会被复制到一个新页上，除 *PageModCount* 值被设置为 25 外，其他值与原页的完全相同。这表示该页第一次被填充满时，会进行一次彻底的分析以确定是否应该进行重新压缩，同时检查 B 树页以确定索引重建和收缩操作期间可能出现的重新压缩。

堆页压缩。只有在重建和收缩操作期间才会检查堆中的页是否需要压缩（注意 SQL Server 2008 为此提供了一个重建表并指定压缩级别的选项）。同样，在删除表上的一个聚集索引使表成为一个堆时，SQL Server 会在所有填充满的页上运行压缩分析。为了保证 RowID 值保持不变，在标准的数据修改操作期间不重新压缩堆。虽然 *PageModCount* 值得到维护，但是 SQL Server 永远不会试图根据 *PageModCount* 值重新压缩某个页面。

6. 压缩元数据

与数据压缩相关的元数据信息不是很多。目录视图 *sys.partitions* 有一个 *data_compression* 列和一个 *data_compress_desc* 列。*data_compression* 列的可能取值为 0、1 和 2，与 *data_compress_desc* 的值 NONE、ROW 和 PAGE 相对应。请记住，虽然总是执行行压缩，但是页压缩不会总执行。即使 *sys.partitions* 表示一个表或一个分区进行了页压缩，但是这只表示启用了页压缩。每个页被单独分析，如果某个页没有填充满，或者压缩不能节省足够的空间，则该页不会被压缩。

也可以检查动态管理函数 *sys.dm_db_index_operatin_stats*。这一表值函数返回如下压缩相关列。

- ***page_compression_attempt_count***。为某个表、索引或索引视图的特殊分区进行页级压缩所估算的页数量。包括没有压缩的页，因为不能明显节省空间。
- ***page_compression_success_count***。对某个表、索引或索引视图的特殊分区进行页压缩时所压缩的数据页数量。

SQL Server 2008 还提供了一个被称为 *sp_estimate_data_compression_savings* 的存储过程，该存储过程为您提供压缩是否能够节省大量大型空间的一些建议。

该存储过程使用表的 5000 个页，同时还在 *tempdb* 数据库中创建了具有样例页的一张等价表。SQL Server 可以利用这一临时表对请求的压缩状态（NONE、ROW 或 PAGE）估算新表大小。可以对整个表或表的一部分进行压缩估算。这包括堆、聚集索引、非聚集索引、索引视图及表和索引分区。

请记住，该结果只是一种估算，真正节省的空间根据行的大小和填充因子的不同而有很大的不同。如果该存储过程表示您可以降低行大小的 40%，则对于整个表来说，可能实际不会获得 40%的空间节省。例如，如果某一行的长度是 8000 字节，您将该行的大小降低 40%，那么仍然只能在一个数据页上存储一行，同时表仍然需要相同数量的页。

通过运行 *sp_estimate_data_compression_savings* 来获得表将增长的结果。这可能会在表中的很多行几乎占用数据类型的整体最大长度并且压缩信息所需的额外系统开销比压缩所节省的空间还要大时发生。

如果表已经被压缩，则可以利用该存储过程估算表（或索引）未被压缩时的大小。

7. 性能问题

压缩数据的主要目的是节省大型表（如数据仓库实际表）的空间。第二个目的是提高扫描表（出于

汇报目的）时的性能，因为需要读取的页会少很多。要记住压缩是需要一定代价的：空间的节省与压缩存储数据及需要数据时进行解压缩所需要的额外 CPU 系统开销之间有一种平衡。在 CPU 受限的系统上，可能发现压缩数据实际上可能会在很大程度上降低系统运行速度。

页压缩为 I/O 受限的系统提供最佳性能，其中数据只写入一次，然后被频繁地读取，正如我们在上一段中提到的数据仓库和报表时的情况一样。对于频繁读取和写入的情况（如 OLTP 应用程序），可能希望只启用行压缩并避免分析页和重建 CI 记录所需的代价。此时，CPU 额外开销是很小的。实际上行压缩是高度优化的，从而使其只在存储引擎层上可见。关联引擎（查询处理器）根本不需要处理压缩行。关联引擎向存储引擎发送未压缩行，然后存储引擎根据需要对其进行压缩。将行返回给关联引擎时，存储引擎会等待尽可能长的时间才对行进行解压缩。在存储引擎中，会对压缩的数据进行比较，因为在比较表中数据之前的内部转换可以将某种数据类型转换成压缩格式。此外，只有关联引擎请求的列需要进行解压缩，这与解压缩整行不同。

压缩和日志记录。通常来说，SQL Server 只记录未压缩的数据，因为日志需要以一种未压缩的格式读取。这表示对压缩记录的日志修改会严重影响性能，因为每一行都需要在写入日志之前进行解压缩和解码（来自锚记录和字典）。这是压缩之所以对基本上只读的系统（日志很少）提供更多优势的另一个原因。

SQL Server 在几种情况下会向日志写入压缩数据。最常见的情况是页面被拆分时。SQL Server 会在拆分操作期间记录数据移动日志的同时写入压缩行。

压缩和版本存储。我们将在第 10 章讨论快照分离时介绍版本存储，这里我们简要介绍版本存储是如何与压缩交互作用的。SQL Server 可以向版本存储中写入压缩行，同时版本存储处理可以以压缩格式遍历旧版本。但是，版本存储不支持页压缩，因此版本存储中的行不能包含锚记录前缀和页面字典的编码。因此在压缩页中任何行需要显示版本信息时，该页必须首先被解压缩。

版本存储用于两种快照分离（完整快照和已提交读快照），同时也可以用于触发触发器时存储已修改数据的前后影像（这些影像在插入和删除的逻辑表中可见）。在估算压缩代价时应该记住这一点。快照分离已经有很多额外开销，向混合快照中添加页压缩会进一步影响性能。

8. 备份压缩

我们在第 1 章介绍配置选项时曾经简要地提到过备份压缩。我认为值得一提的是压缩备份所使用的算法，它与本章介绍的数据库压缩算法完全不同。备份压缩使用一种非常类似于在集群上压缩的算法，这种算法只查找数据中的模式。即使表和索引已经使用数据压缩技术进行了压缩，仍然可以进一步使用备份压缩算法进行压缩。

页压缩只查找前缀模式，仍然可能有没有被压缩的其他模式。页压缩会删除冗余字符串，但是仍然有很多字符串在大部分情况下不是冗余的，使用 zip 类型的算法可以很好地压缩字符串数据。

此外，数据库中有相当多的空间造成了开销，如页上未分配的槽及已分配范围内的未分配页面。根据是否使用即时文件初始化，以及磁盘上原先是什么内容，背景数据实际上可以被很好地压缩。

因此对有很多压缩表和索引的数据库进行压缩备份可以为备份设置提供额外的空间。

7.5 表和索引分区

正如我们在查看表和索引存储的元数据时所看到的那样，分区是 SQL Server 空间管理的一种内部特性。第 5 章中的图 5-7 显示了表和索引、分区及分配单元之间的关系。没有参照任何分区而构建的表和索引可以被认为存储在单个分区上。检索数据存储相关信息的一种更有效的元数据对象是被称为 *sys.dm_db_partition_stats*

的动态管理视图，该视图结合了 *sys.partitions*、*sys.allocation_units* 和 *sys.indexes* 中的信息。

一个分区对象是内部拆分成可以存储在不同位置的独立物理单元的对象。分区对于用户和程序员（可以利用 T-SQL 代码从一个分区表中进行查询的人，方法与从一个非分区表中进行查询的方式相同）来说是不可见的。在多个分区上创建大型对象可以提高数据库系统的可管理性和可维护性，同时能够大大提高清除历史数据和加载大量数据等活动的性能。在 SQL Server 2000 中，分区只能通过手动创建一个结合了多个表的视图来获得。这一功能被称为*分区视图*。SQL Server 2005 和 SQL Server 2008 内置表和索引的分区与分区视图相比有很多好处，包括提高的执行计划和更少的执行需求。

在这一部分，我们主要关注分区对象的物理存储和分区元数据。在第 8 章，我们将介绍包含分区表和分区索引的查询计划。

7.5.1 分区函数和分区方案

为了认识分区元数据，需要了解一点如何定义分区的一些背景知识。使用一个基于 SQL Server 示例的例子。您可以在随附网站上看到名为 *Partition.sql* 的脚本。该脚本定义 *TransactionHistory* 和 *TransactionHistoryArchive* 两个表，每一个表上都有一个聚集索引和两个非聚集索引。两个表都在 *TransactionDate* 列上进行分区，在一个独立的分区中存储每个月的数据。初始情况下，*TransactionHistory* 中有 12 个分区，*TransactionHistoryArchive* 中有两个分区。

创建一张分区表或一个分区索引之前，必须先定义一个分区函数。分区函数用于在逻辑上定义分区边界。当一个分区函数被创建时，您必须指定该分区是否会使用基于 LEFT 或基于 RIGHT 的边界点。简单地说，就是定义边界值本身是否是左侧或右侧分区的一部分。考虑这一问题的另一种方式是回答“这是某个分区（此时是 LEFT）的上边界还是下一个分区（此时是 RIGHT）的下边界点？”利用一个具有 *n* 个边界的分区函数创建的分区数量是 *n*+1。下面是我们对这个示例使用的分区函数：

```
CREATE PARTITION FUNCTION [TransactionRangePF1] (datetime)
AS RANGE RIGHT FOR VALUES ('20081001', '20081101', '20081201',
               '20090101', '20090201', '20090301', '20090401',
               '20090501', '20090601', '20090701', '20090801');
```

注意表名没有在函数定义中提到，因为分区函数没有与任何特殊表相连接。函数 *TransactionRangePF1* 将数据分成 12 个分区，因为有 11 个 *datetime* 边界。关键字 RIGHT 指定与其中一个边界点相等的值检查分区的终点。因此对于该函数来说，所有小于（October 1, 2008）的值都存储在第一个分区中，所有大于或等于（October 1, 2008）并且小于（November 1, 2008）的值都会存储在第二个分区中。我们也可以指定 *LEFT*（这是默认值），此时等于终点的值会存储在左侧分区中。定义分区函数后，就定义了一种分区方案（将列出一组每个数据系列所存储的文件组）。下面是本示例的分区架构：

```
CREATE PARTITION SCHEME [TransactionsPS1]
AS PARTITION [TransactionRangePF1]
TO ([PRIMARY], [PRIMARY], [PRIMARY]
, [PRIMARY], [PRIMARY], [PRIMARY]
, [PRIMARY], [PRIMARY], [PRIMARY]
, [PRIMARY], [PRIMARY], [PRIMARY]);
GO
```

为了避免必须创建 12 个文件和文件组，将所有分区放在 PRIMARY 文件组上，但是为了充分发挥分区的优势，应该使每个分区位于自己的文件组上。*CREATE PARTITION SCHEME* 命令一定列出至少和分

区数一样多的文件组，但是还可以再有一个。如果列出了一个额外的文件组，则被认为是“下一个使用”的文件组。如果分区函数拆分，则新边界点会被添加在使用的下一个文件组中。如果没有在创建分区方案时指定额外的文件组，则可以在修改该函数之前将分区方案修改为下一个使用的文件组。

如您所见，列出的文件组不一定是唯一的。实际上，如果您希望使所有的分区位于同一个文件组上，可以使用如下快捷语法：

```
CREATE PARTITION SCHEME [TransactionsPS1]
AS PARTITION [TransactionRangePF1]
ALL TO ([PRIMARY]);
GO
```

注意将所有分区放在同一个文件组上通常只在测试代码时使用。

附加文件组在添加更多分区时按顺序使用，当一个分区函数修改为将一个现有系列拆分成两个系列时，可能会出现添加分区的情况。如果在创建分区方案时没有指定额外的文件组，则可以将分区方案修改为添加另一个文件组。

用于两个表的分区函数和分区方案如下：

```
CREATE PARTITION FUNCTION [TransactionArchivePF2] (datetime)
AS RANGE RIGHT FOR VALUES ('20080901');
GO

CREATE PARTITION SCHEME [TransactionArchivePS2]
AS PARTITION [TransactionArchivePF2]
TO ([PRIMARY], [PRIMARY]);
GO
```

脚本接下来创建两个表并向其中加载数据。这里我们不做详细介绍。为了对一个表进行分区，必须在 *CREATE TABLE* 语句中指定一种分区方案。我们创建一个名为 *TransactionArchive* 的表，该表中包含下面的一行作为 *CREATE TABLE* 语句的最后一部分：

```
ON [TransactionsPS1] (TransactionDate)
```

第二个表 *TransactionArchiveHistory* 是利用 *TransactionsPS1* 分区方案创建的。

脚本接下来将数据加载到两个表中，同时由于已经定义了分区方案，因此会在数据加载时将每一行放在适当的分区上。表被加载后，就可以检查元数据了。

7.5.2 分区的元数据

图 7-16 显示了检索分区信息的大部分目录视图。除了左边和底边界，还可以查看本章前面介绍过的 *sys.tables*、*sys.indexes*、*sys.partitions* 和 *sys.allocation_units* 目录视图。

在一些查询中，我们利用未形成文档的 *sys.system_internals_allocation_units* 视图（而不是 *sys.allocation_units*）来检索页面的地址信息。这里介绍其中与每一个视图最相关的列。

- ***sys.data_spaces***。有一个名为 *data_space_id* 的主键，该主键可以是一个分区 ID，也可以是一个文件组 ID，同时有一行用于存储每一个文件组，还有一行用于存储每个分区方案。*sys.data_spaces* 中某一列指定该行所引用的数据空间类型。如果该行引用一种分区方案，则 *data_space_id* 可以和 *sys.partition_schemes.data_space_id* 相连接。如果该行引用一个文件组，则 *data_space_id* 可以与 *sys.filegroups.data_space_id* 相连接。*sys.indexes* 视图也有一个 *data_space_id* 列用于表示 *sys.indexes* 中存

储的每个堆或 B 树是如何存储的。因此，如果我们知道某个表被分区了，则可以直接将该表与 *sys.partition_schemes* 相连接，不必检查 *sys.data_spaces*。也可以选择利用如下查询来确定某个表是否利用我们所感兴趣的表名来替换 *Production.TransactionHistoryArchive* 的方式进行分区：

```
SELECT DISTINCT object_name(object_id) as TableName,
        ISNULL(ps.name, 'Not partitioned') as PartitionScheme
    FROM (sys.indexes i LEFT JOIN sys.partition_schemes ps
                ON (i.data_space_id = ps.data_space_id))
    WHERE (i.object_id = object_id('Production.TransactionHistoryArchive'))
            AND (i.index_id IN (0,1));
```

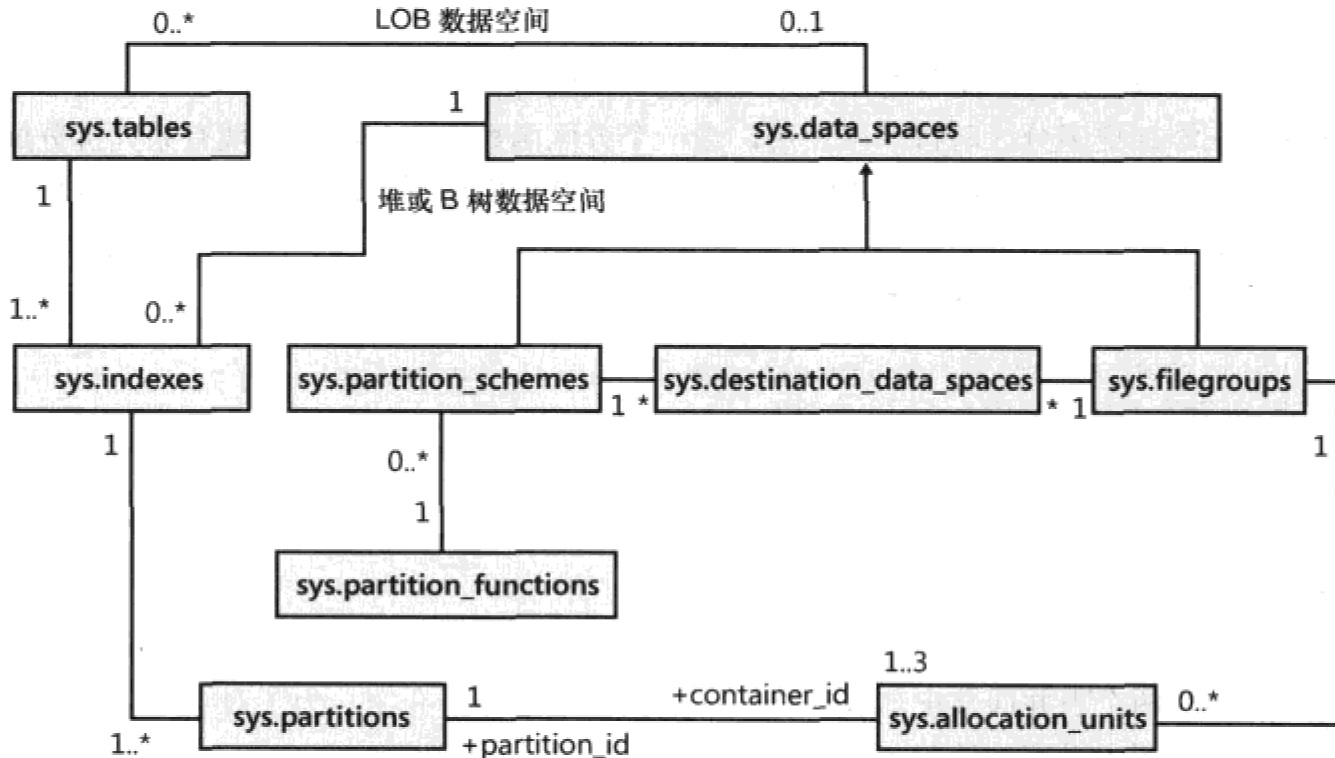


图 7-16 包含分区和数据存储元数据的目录视图

- ***sys.partition_schemes***。为每种分区方案提供一行。除 *data_space_id* 和分区方案名称之外，还将 *function_id* 列与 *sys.partition_functions* 进行连接。
- ***sys.destination_data_space***。是一张链接表，因为 *sys.partition_schemes* 和 *sys.fi legroups* 之间有一种多对多的关系。对于每一种分区方案来说，每个分区有一行。分区编号存储在 *destination_id* 列中，文件组 ID 存储在 *data_space_id* 列中。
- ***sys.partition_functions***。为每个分区函数包含一行，其主键 *function_id* 是 *sys.partition_schemes* 中的一个外键。
- ***sys.partition_range_values***。（没有显示）为每个分区函数的每个终点包含一行。它的 *function_id* 列与 *sys.partition_functions* 相连接，同时它的 *boundary_id* 列可以与 *sys.partitions* 中的 *partition_id* 或 *sys.destination_data_spaces* 中的 *destination_id* 相连接。

这些视图中还有其他一些我们没有提到过的列，同时还有提供信息的一些其他视图，如分区所依据的列和列的数据类型。但是，前面提到的信息对于理解图 7-15 和下面的代码段中所显示的视图来说应该已经足够了。该视图返回每个分区表每个分区的信息。*WHERE* 子句过滤掉分区索引（而不是聚集索引），但是你可以修改你希望的条件。当从该视图中进行查询时，可以添加自己的 *WHERE* 子句，从而只查找自己感兴趣表的信息：

```
CREATE VIEW Partition_Info AS
```

```
SELECT OBJECT_NAME(i.object_id) as Object_Name,
       p.partition_number, fg.name AS Filegroup_Name, rows, au.total_pages,
       CASE boundary_value_on_right
                    WHEN 1 THEN 'less than'
                    ELSE 'less than or equal to' END as 'comparison', value
FROM sys.partitions p JOIN sys.indexes i
     ON p.object_id = i.object_id and p.index_id = i.index_id
       JOIN sys.partition_schemes ps
                ON ps.data_space_id = i.data_space_id
       JOIN sys.partition_functions f
                    ON f.function_id = ps.function_id
       LEFT JOIN sys.partition_range_values rv
ON f.function_id = rv.function_id
                     AND p.partition_number = rv.boundary_id
       JOIN sys.destination_data_spaces dds
               ON dds.partition_scheme_id = ps.data_space_id
                     AND dds.destination_id = p.partition_number
       JOIN sys.filegroups fg
                    ON dds.data_space_id = fg.data_space_id
       JOIN (SELECT container_id, sum(total_pages) as total_pages
                        FROM sys.allocation_units
                        GROUP BY container_id) AS au
                    ON au.container_id = p.partition_id
WHERE i.index_id <2;
```

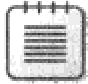
LEFT JOIN 操作符需要获得所有分区，因为 *sys.partition_range_values* 视图仅有一行用于每个分界值（而不是每个分区）。*LEFT* JOIN 将最后一个分区的边界值设置为 NULL，这表示最后一个分区的值没有上限。一个继承的表将某个分区 *sys.allocation_units* 中的所有行都分组到一起，因此所有存储类型（行内、行溢出和LOB）所使用的空间都聚集成一个值。该查询使用前面的视图获取关于 *TransactionHistory* 表的分区信息：

```
SELECT * FROM Partition_Info
WHERE Object_Name = 'TransactionHistory';
```

下面是得到的结果：

Object_Name	Partition _number	Filegroup_Name	行数	总页数	比较	值
TransactionHistory	1	PRIMARY	11155	209	小于	2008-10-01
TransactionHistory	2	PRIMARY	9339	107	小于	2008-11-01
TransactionHistory	3	PRIMARY	10169	185	小于	2008-12-01
TransactionHistory	4	PRIMARY	12181	225	小于	2009-01-01
TransactionHistory	5	PRIMARY	9558	177	小于	2009-02-01
TransactionHistory	6	PRIMARY	10217	193	小于	2009-03-01
TransactionHistory	7	PRIMARY	10703	201	小于	2009-04-01
TransactionHistory	8	PRIMARY	10640	193	小于	2009-05-01
TransactionHistory	9	PRIMARY	12508	225	小于	2009-06-01
TransactionHistory	10	PRIMARY	12585	233	小于	2009-07-01
TransactionHistory	11	PRIMARY	3380	73	小于	2009-08-01
TransactionHistory	12	PRIMARY	1008	33	小于	NULL

该视图包含每个分区边界点的详细信息，以及每个分区上存储的文件组、每个分区中的行数和使用的空间数量。注意，虽然比较结果表示某个特殊分区中行在分区列中的值小于指定值，但是应该知道该值也表示它大于或等于上一个分区中的指定值。但是，该视图不提供数据所在的特殊文件组的相关信息。我们将在下一部分介绍提供位置信息的元数据查询。



注意：

如果一张分区表中包含文件流数据，则建议您使用与非文件流数据相同的分区函数对文件流数据进行分区。因为标准数据和文件流数据位于独立的文件组上，因此文件流数据需要自己的分区方案。但是，文件流数据的分区方案可以使用相同的分区函数，从而保证文件流和非文件流数据使用相同的分区。

7.5.3 分区的滑动窗口优势

对数据进行分区的一个最主要的好处就是可以通过只针对元数据的操作将数据从一个分区移动到另一个分区上。数据本身不必移动。正如我们曾经提到的那样，我们不会对 SQL Server 2008 分区的操作方式进行全面介绍，只介绍分区信息的内部存储方案。但是，为了说明重新排列分区的内部，我们需要了解一些额外的分区操作。

操作分区时使用的主要操作是 *ALTER TABLE* 命令的 SWITCH 选项。该选项允许您：

- 为某张已有分区的表分配一个表作为一个分区；
- 将某个分区在两个分区表中进行交换；
- 重新分配一个分区以建立一个单独的表。

在所有这些操作中，数据不会被移动，而是更新 *sys.partitions* 和 *sys.system_internals_allocation_units* 视图中的元数据，说明某个给定分配单元现在是另一个分区的一部分。接下来我们来看一个示例。下面的查询返回 *TransactionHistory* 和 *TransactionHistoryArchive* 表前两个分区中每个分配单元的信息，包括行数量、分配单元的数据类型及分配单元的起始页：

```
SELECT convert(char(25),object_name(object_id)) AS name,
    rows, convert(char(15),type_desc) as page_type_desc,
    total_pages AS pages, first_page, index_id, partition_number
FROM sys.partitions p JOIN sys.system_internals_allocation_units a
     ON p.partition_id = a.container_id
WHERE (object_id=object_id('[Production].[TransactionHistory]')
   OR object_id=object_id('[Production].[TransactionHistoryArchive]'))
  AND index_id = 1 AND partition_number <= 2;
```

下面是得到的数据（没有显示 *page_type_desc*，因为所有行都是 *IN_ROW_DATA* 类型）。

```
name                      rows    pages     first_page     index_id partition_number
------------------------- ------- --------- -------------- -------- ----------------
TransactionHistory        11155   209       0xD81B00000100 1        1
TransactionHistory        9339    177       0xA82200000100 1        2
TransactionHistoryArchive 89253   1553      0x981B00000100 1        1
TransactionHistoryArchive 0       0         0x000000000000 1        2
```

现在我们来移动某个分区。最终目标是向 *TransactionHistory* 添加一个分区来存储一个新月份的数据并将最早月份的数据移到 *TransactionHistoryArchive* 中。*TransactionHistory* 表所使用的分区函数将数据分

到 12 个分区中，最后一个分区中包含所有大于或等于（August 1,2009）的日期，因此最后一个分区被拆分。在执行此操作之前，必须保证使用该函数的分区方案知道为新创建的分区使用哪个文件组。使用该命令，某些数据会移动，同时使用这种分区方案的所有表中最后一个分区中的所有数据都会移动到一个新的分配单元中。参阅 *SQL Server 联机丛书*查看如下每种命令的完整信息：

```
ALTER PARTITION SCHEME TransactionsPS1
NEXT USED [PRIMARY];
GO

ALTER PARTITION FUNCTION TransactionRangePF1()
SPLIT RANGE ('20090901');
GO
```

接下来进行与 *TransactionHistoryArchive* 使用的函数和分区方案相类似的操作。这里将为（October 1, 2008）添加一个新的边界点：

```
ALTER PARTITION SCHEME TransactionArchivePS2
NEXT USED [PRIMARY];
GO

ALTER PARTITION FUNCTION TransactionArchivePF2()
SPLIT RANGE ('20081001');
GO
C07626249.indd 44C07626249.440 2/16/2009 1:34:42 PM
```

我们希望将 *TransactionHistory* 中所有早于（October 1, 2008）的数据移动到 *TransactionHistoryArchive* 的第二个分区中。但是，从技术角度来说，*TransactionHistory* 的第一个分区没有下限，该分区包含所有早于（October 1, 2008）的数据。*TransactionHistoryArchive* 的第二个分区有一个下限，该下限是第一个边界点（September 1, 2008）。为了将某个分区从一个表交换到另一个表上，需要保证所有被移动的数据满足新位置的需求。因此我们添加一项 *CHECK* 约束保证 *TransactionHistory* 的所有数据都不会早于（September 1, 2008）。添加完 *CHECK* 约束后，运行带有 SWITCH 选项的 *ALTER TABLE* 命令，将 *TransactionHistory* 的分区 1 中的数据移动到 *TransactionHistoryArchive* 的分区 2 上（出于测试目的，可以省略添加约束的下一个步骤，只执行 *ALTER TABLE/SWITCH* 命令，这样会得到一条错误消息。此后，可以添加约束并再次运行 *ALTER TABLE/SWITCH* 命令）。

```
ALTER TABLE [Production].[TransactionHistory]
ADD CONSTRAINT [CK_TransactionHistory_DateRange]
CHECK ([TransactionDate] >= '20080901');
GO
ALTER TABLE [Production].[TransactionHistory]
SWITCH PARTITION 1
TO [Production].[TransactionHistoryArchive] PARTITION 2;
GO
```

现在我们运行检查每个表前两个分区的位置和大小的元数据查询语句：

```
SELECT convert(char(25),object_name(object_id)) AS name,
    rows, convert(char(15),type_desc) as page_type_desc,
    total_pages AS pages, first_page, index_id, partition_number
FROM sys.partitions p JOIN sys.system_internals_allocation_units a
```

```
      ON p.partition_id = a.container_id
WHERE (object_id=object_id('[Production].[TransactionHistory]')
    OR object_id=object_id('[Production].[TransactionHistoryArchive]'))
  AND index_id = 1 AND partition_number <= 2;

RESULTS:
name                 rows    pages      first_page      index_id    partition_number
-------------------- ------- ---------- --------------  ----------- ----------------
TransactionHistory   0       0          0x000000000000  1           1
TransactionHistory   9339    177        0xA82200000100  1           2
TransactionHistoryAr 89253   1553       0x981B00000100  1           1
TransactionHistoryAr 11155   209        0xD81B00000100  1           2
```

您会发现*TransactionHistoryArchive*的第二个分区的现有信息与*TransactionHistory*第一个分区在第一个结果集中的信息完全相同。具有相同的行数（11，155）、相同的页数（209）及相同的起始页（0xD81B00000100 或文件 1，页 7128）。不会有任何数据被移动。唯一的变化是起始位置为文件 1 页面 7128 的分配单元不再属于 *TransactionHistoryArchive* 表的第二个分区。

虽然分区脚本使用表本身所使用的相同分区方案为分区表创建索引，但是这不一定是必要的。某个分区表的索引可以使用相同的分区方案或不同的分区方案进行分区。如果在某个分区表上创建索引时没有指定一种分区方案或文件组，则会使用相同的分区列将索引放在与基表相同的分区方案中。在与基表相同的分区方案上构建的索引被称为*对齐索引*。

但是，内部存储组件自动与对齐索引相联。正如前面所述，如果您在一张分区表上创建一个索引并且不指定放置索引的文件组或分区方案，SQL Server 会利用表使用的分区方案创建索引。但是，如果分区列不是索引定义的一部分，则 SQL Server 会添加分区列作为索引中一个额外的包含性列。如果是聚集索引，则添加一个包含性列是不必要的，因为聚集索引已经包含所有列。SQL Server 不自动添加一个包含列的另一种情况是创建一个唯一索引（不论是聚集索引还是非聚集索引）。由于唯一分区索引要求分区列包含在唯一键中，因此没有明确包含分区键的唯一索引页不会被自动分区。

7.6 小结

本章介绍了 SQL Server 2008 如何存储没有使用标准 *FixedVar* 记录格式的数据及不适合标准 8KB 数据页的数据。

我们讨论了行溢出和大型对象数据，该数据被存储在自己的独立页上。同时还讨论了文件流数据，该数据存储在 SQL Server 外部文件系统的文件中。

SQL Server 2008 中的某些新存储功能要求以一种完全不同的方式看待行存储。稀疏列允许表最多可以有 30 000 列，只要其中大部分列、大部分行的数据为 NULL。包含稀疏列的表中的每一行都有一个特殊的描述字段提供该特殊行的哪一列非 NULL 的信息。

我们还介绍了一种全新的使用压缩数据的行存储格式。数据可以在行级或页级上进行压缩，同时行和页本身说明其中包含的数据。这种行格式被称为 CD 格式。

最后介绍了表和索引的分区。虽然分区实际上不要求行和页面有特殊格式，但是需要以一种特殊的方式访问元数据。

第 8 章 查询优化器

Conor Cunningham

Microsoft SQL Server 2008 中的查询优化器确定指定 SQL 语句将要执行的查询计划。由于查询优化器没有很多公开功能，因此不像 SQL Server 引擎中的其他组件那样得到广泛的理解。本章介绍查询优化器及其工作方式。学完本章，您应该能够理解高级优化器的架构，而且应该能明白为什么某个特殊计划会被查询优化器选中。作为一种能力扩展，您应该能够在查询优化器没有选择所需查询计划并不知道如何影响选择时排除有问题的查询计划。

本章分为两部分。第一部分解释查询优化器的基本机制，这包括使用的高级结构及如何为每个计划定义替代选择集。第二部分介绍查询分析器中的特殊区域及它们是如何适应这种框架的。例如，"如何选择索引"、"如何使用统计"及"我如何理解更新计划"等主题。

8.1 概述

单个查询的基本编译"管道"如图 8-1 所示。

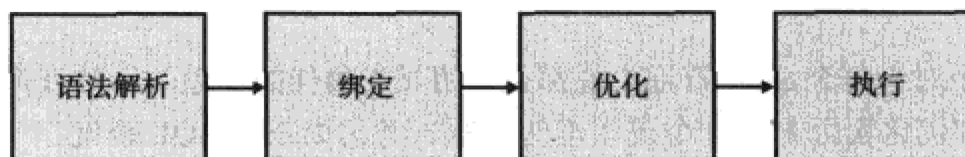


图 8-1 查询处理器管道

当一个查询被编译时，SQL 语句首先被*解析*成一棵等价的树表达式。对于具有有效 SQL 语法的查询来说，下一阶段对查询执行一系列验证步骤（通常称为*绑定*），此时树中的列和表与数据库元数据进行比较，从而保证这些列和表存在并且对当前用户可用。这一阶段还对查询进行语义检查，从而保证查询是有效的，例如保证绑定到某个 *GROUP BY* 操作的列是有效的。一旦查询树被绑定并验证为是一个有效的查询，查询优化器就会获取该查询并开始评估可能存在的不同查询计划。查询优化器执行搜索，然后选择要执行的查询计划并将其返回给系统来执行。执行组件运行查询计划并返回查询结果。

SQL Server 2008 查询优化器有很多扩展这一图表的其他功能，从而使其对数据库开发人员和 DBA 更有用。例如，查询计划被缓存，因为查询计划的生成开销很高并且经常被重复使用。如果基础数据被明显修改，则旧查询计划会被重新编译。SQL Server 还支持 T-SQL 语言，也就是说可以在一次 SQL Server 引擎中处理多条语句。查询优化器不考虑批处理编译或工作负载分析，因此本章主要介绍单条查询的编译情况。

8.1.1 树格式

向查询处理器提交一条 SQL 查询时，SQL 串会被解析成一种树表达式。树中的每个结点代表一条要执

行的查询操作。例如，FROM 子句中的每个表都有自己的运算符。一个 WHERE 子句也以一种单独的运算符来表示。联接用每个表有一个输入的运算符来表示。例如，查询语句 *SELECT * FROM Customers C INNER JOIN Orders O on C.cid = O.cid WHERE O.date = '2008-11-06'*在系统内部可以表示成图 8-2 所示的形式。

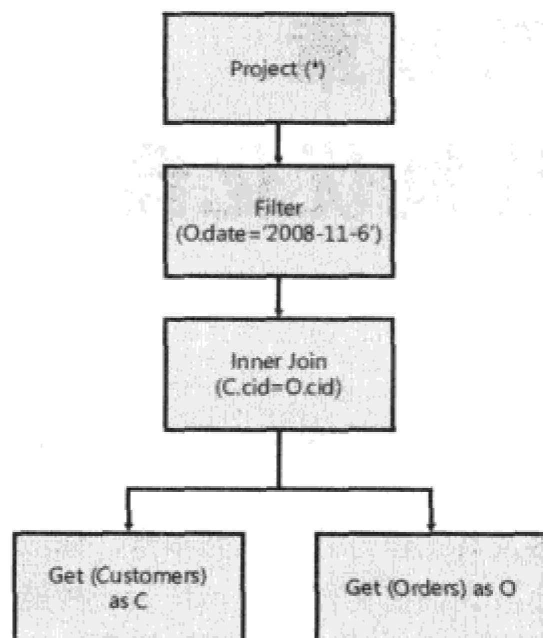


图 8-2　查询树格式实例

查询处理器在编译过程中实际上使用的是不同的树格式。例如，查询优化器执行的一项工作是将一棵树从所需的逻辑描述转换成一项可以被执行的具有真实物理运算符的计划。可能出现这种选择的最明显情况是查询优化器选择一种联接算法时，将一种逻辑联接（例如，*INNER JOIN*）转换成一种物理联接（一种哈希联接、合并联接或嵌套循环联接）。大部分树格式彼此都非常类似。在本章的很多示例中，优化器输出树都用于描述查询优化器原来执行的特殊优化，即内部树格式。

8.2　什么是优化

到目前为止，我们只介绍了从一种逻辑查询树到一种等价物理查询计划的基本转换。查询优化器的另一项主要工作是查找一种有效的查询计划。通常有多种方法来评估一个给定的查询，并且有些计划会比其他计划慢很多。两种不同计划之间的速度差异非常明显，因此选择一种较差的查询计划可能会使一种数据库应用程序执行速度非常慢，对于用户来说则显示为不连续。因此，查询优化器选择一种有效的计划是非常重要的。

首先，似乎有一个对于每个 SQL 查询来说都是“明显”最佳的计划，而且查询优化器应该尽快选择它。但遗憾的是，查询优化实际上是一个更困难的问题。考虑如下的 SQL 查询：

```
SELECT * FROM A
INNER JOIN B ON (A.a = B.b)
INNER JOIN C ON (A.a = C.c)
INNER JOIN D ON (A.a = D.d)
INNER JOIN E ON (A.a = E.e)
INNER JOIN F ON (A.a = F.f)
INNER JOIN G ON (A.a = G.g)
INNER JOIN H ON (A.a = H.h)
```

这条查询有很多可能的实现计划，因为内部联接可以按照不同的顺序进行计算。实际上如果使用相同的模式向该查询添加更多的表，则查询会有很多可选计划，因此考虑所有计划是不可能的。因为内部联接可以以任意顺序（ABCD…, ABDC…, ACBD…）和不同的拓扑[(A join B) join (B join C)]进行评估，因此该查询的可能查询计划数量实际上大于 N! [N x (N-1) x (N-2) x…]。随着查询中表数量的增加，可以考虑的选项迅速超过任何计算机的可计算能力。所有可用查询计划的存储也成为一个问题。在 32 位的 Intel x86 计算机上，SQL Server 通常有一个可用于编译查询的约 1.6GB 的内存，但是在内存中存储所有可能的选项是不可能的。即使一台计算机能够存储所有这些选项，用户也不希望等待那么久才能查看所有这些选项。查询优化器利用探索法和指导这些探索法的统计信息来解决这一问题，本章将对这些概念进行介绍。

很多人都认为为给定查询选择绝对最好的查询计划是查询优化器的工作。您现在发现这是不可能的，

如果您不能考虑每种计划的执行情况，则很难证明某项计划是最优的。但是，查询优化器可以迅速找到一种“足够好”的计划，并且这通常也是最佳的执行计划或接近最佳的计划。

8.3 查询优化器如何研究查询计划

查询优化器使用一种框架来有效地搜索和比较多种不同的可用计划替代项，该框架允许 SQL Server 考虑用复杂、不明显的方式来实现给定的查询。跟踪所有这些不同的替代项来查找一种有效运行的计划是一件困难的事情。SQL Server 的搜索框架包含一些组件能帮助它有效并可靠地完成工作。虽然大部分都是内部的，但是我们在这里介绍这些组件是为了让您了解一条查询是如何被优化的，并且使您能够利用这些功能更好地设计自己的应用程序。

8.3.1 规则

查询优化器是一种搜索框架。对于给定的一棵查询树来说，查询优化器会考虑将该树从当前状态转换成另一种已经存储在内存中的等价状态。在 SQL Server 使用的框架中，转换是通过*规则*实现的。这些规则与您在学校中学到的数学定理非常类似。例如，我们知道 *A INNER JOIN B* 等价于 *B INNER JOIN A*，因为两种查询对所有可能的表数据集来说都返回相同的结果。这是一种交换形式，在标准的整型算术中，这表示（1+2）和（2+1）是等价的，意味着这种操作可以以任意顺序执行并产生相同的结果（或者就数据库来说，返回相同的行集）。规则被匹配到树模式中并且如果适合生成新的选项（这接下来可能还会产生更多的规则匹配）则会被应用。这些规则形成了查询优化器的工作基础，帮助在一段合理的时间内对搜索所需的一些探索法进行编码。

查询优化器有不同种类的规则。探索性地将一棵查询树重新写入一种新形式称为*替代规则*。考虑数学等价性的规则称为*探索规则*。这些规则生成新的树形状但是不能直接被执行。将逻辑树转换成将被执行的物理树的规则称为*实现规则*。在这些生成的物理替代项中，最好的是最终由查询优化器作为最终查询执行计划输出的选项。

更多信息：

本章包含很多查询执行计划的示例，用于说明查询优化器的行为。如果希望了解如何解释查询执行计划及各种运算符含义的更多背景信息，可以参阅《Inside Microsoft SQL Server 2005: Query Tuning and Optimization》（Microsoft Press，2007）的第 3 章。除 SQL Server 2008 中显示的图形化查询计划有一些细微变化外，本章中的所有内容几乎都可应用到 SQL Server 2008。本章内容也可通过随书网站 *http://www.SQLServerInternals.com/ companion* 获得。

8.3.2 属性

搜索框架以一种使规则更容易工作的方式收集查询树的信息。这些被称为*属性*，同时它们从子树中收集信息，从而帮助确定哪些规则可以在树中更高点上处理。例如，SQL Server 中使用的一种属性是构成唯一键的列。考虑如下的查询：

```
SELECT col1, col2, MAX(col3) FROM Table1 GROUP BY col1, col2;
```

这条查询在系统内部表示成图 8-3 所示的一棵树。

如果列（*col1*，*col2*）构成表 *groupby* 的一个唯一键，则根本不需要执行分组——每组只有一行。一组大小的 *MAX*（）是元素本身。因此，可以写一条规则从查询树中完全删除 *groupby*。可以在图 8-4 中看到此规则的运用。

```
CREATE TABLE groupby (col1 int, col2 int, col3 int);
ALTER TABLE groupby ADD CONSTRAINT unique1 UNIQUE(col1, col2);

SELECT col1, col2, MAX(col3) FROM groupby GROUP BY col1, col2;
```

如果查看最终的查询计划，可以看到，即使查询中有 *GROUP BY* 子句，查询优化器也不执行分组操作。优化期间收集到的属性使得可以用这一规则执行树转换来使结果查询计划更快地完成。

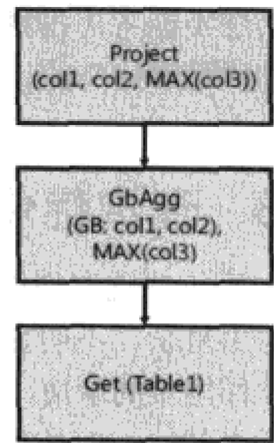


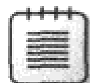
图 8-3 GROUP BY 树实例

SQL Server 也会在优化期间收集很多属性。与在大部分现代编译器中执行的工作相同，查询优化器收集查询中引用的每一列的域约束条件信息。查询优化器从谓词、联接条件、分区信息和检查约束中收集信息，分析所有这些谓词是如何用于优化查询的。这种标量属性的一个有用的应用程序是*冲突检测*。查询优化器可以决定该查询是否是以一种根本不会返回任何行的形式编写的。当查询优化器检测到一种冲突时，它实际上会重写查询来删除包含冲突的部分。图 8-5 所示包含了在优化期间检测到的一个冲突示例。

```
CREATE TABLE DomainTable(col1 int);
GO
SELECT *
FROM DomainTable D1
INNER JOIN DomainTable D2
ON D1.col1=D2.col1
WHERE D1.col1 > 5 AND D2.col1 < 0;
```

图 8-4 删除了聚合操作的查询计划　　　　图 8-5 通过冲突简化的查询计划

最终的查询计划实际上根本不引用该表，它被一个不访问存储引擎并且返回零行（在本例中）的特殊常量扫描运算符所替换。这说明查询运行速度更快、占用更少的内存，同时不需要对本部分引用的、执行时包含冲突的资源获取锁。



注意：

本章通过创建可以运行的示例，让您根据这些示例操作了解系统的运行方式。但遗憾的是，在某些情况下不同的功能相互作用，从而很难明确向您单独显示某项功能的运行方式。在这个示例中，我们添加一个联接，从而避免了另一种被称为*琐碎计划*的优化，这种优化有时会覆盖冲突检测。由于不同版本的功能有所不同，因此我们建议您只使用这些示例浏览查询优化器的当前状态，不能保证不同版本查询优化器的内部工作方式是相通的，因此您不应该在应用程序中包含查询优化器的细节内容。

与规则类似，属性有逻辑属性和物理属性。逻辑属性包含输出列集、键列及某一列是否会输出 null

值之类的信息。这些属性可以应用到所有等价的逻辑和物理计划碎片中。当一种搜索规则被评估时，结果查询树会共享相同的逻辑属性作为该规则使用的原始树。物理属性是针对某个计划的，每个计划运算符都有一组与之相关的物理属性。一种公用的物理属性是结果是否排序。该属性影响查询优化器是否查找一个索引来传送所需的排序。另一种物理属性是来自某个查询可以读取的一个表的列集。这可以确定某个辅助索引是否足够返回查询中所需的所有列，以及每个匹配行是否还需要对基表进行查找。

8.3.3 替代项的存储——“备注”

本章前面我们曾经提到过，对于某些查询来说优化期间所有被考虑的替代项可能很多。查询优化器包含一种避免存储重复信息的机制，这样在编译过程中可以节省内存（和时间）。这种结构被称为 *Memo*（*备注*），它的目标之一是查找以前搜索过的子树并避免重新优化计划的这些区域。它存在于一次优化过程中。

*备注*通过按组存储等价树的方式工作。如果希望在一个组内执行每棵子树，则该子树中的每个替代项将返回相同的逻辑结果。从概念上来说，原始查询树中的每个运算符都是从自己所在的组开始的，也就是说存储在*备注*中时，组会引用其他组而不是直接引用其他运算符。这种模型用于避免在查询优化期间多次存储树，并且可以使查询优化器不多次搜索可能出现的同一种计划替代项。

除了存储等价的替代项外，组还存储属性结构。位于同一组中的替代项有等价的逻辑和标量属性。逻辑属性在 SQL Server 中实际上被称为*组属性*，即使它们没有存储在*备注*中。因此，一个组中的每个选项应该都有相同的输出列、键列、可能的分区等。计算这些属性的开销是很大的，因此该结构也可以帮助避免在优化期间执行不必要的工作。

所有被考虑的计划都存储在备注中。对于大型查询来说，备注可能包含数千个组并且每一组中包含很多替代项。这样就可以表示相当多的替代项。虽然大部分查询在优化期间不会占用大量的内存，但是大型数据仓库查询很有可能在优化期间占用机器上的所有内存。如果查询优化器在搜索计划时会用完内存，则它会包含一种逻辑来选取一种“足够好”的查询计划而不是占用所有内存。

当查询优化器完成对某项计划的搜索后，它会从根开始检查备注，从而从满足查询需求的各组中选择最好的选项。这些运算符被集成到最终查询计划中，然后被转换成一种可以被 SQL Server 中的查询执行组件理解的格式。这种最终的树转换确实包含一小部分运行时优化重写，但是它与为查询计划生成的显示计划输出非常接近。

我们将在本章后面讨论检查查询优化器的结构和管道时介绍备注工作方式的一个示例。

8.3.4 运算符

SQL Server 2008 大约有 40 个逻辑运算符和更多的物理运算符。一些运算符是非常常见的，如 Join 或 Filter。其他运算符（如 Segment、Sequence Project 和 UDX）则很少看到。SQL Server 2008 中的运算符遵循图 8-6 所示的模型。

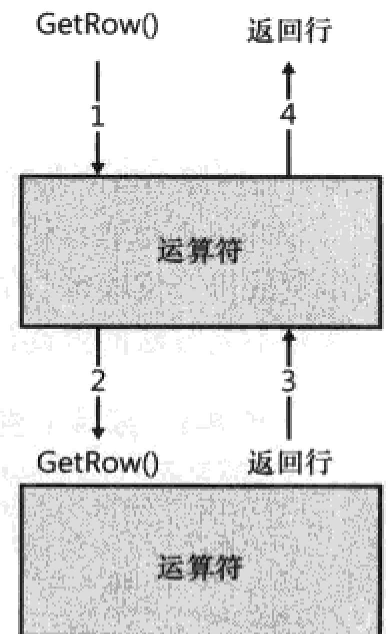


图 8-6 SQL Server 运算符数据流模型

SQL Server 中的每个运算符都是从一个或多个子行中请求行，然后生成行返回给调用程序的方式进行工作。调用程序可以是另一种运算符，也可以发送给用户（如果调用程序是查询树中最上级的运算符）。每个运算符每次都返回一行，也就是说调用程序必须调用每一行。这种设计的一致性允许运算符以多种不同的方式进行结合，也允许新的运算符在不对查询优化器进行重要修改

的情况下被添加到系统中，例如属性框架可以帮助查询优化器选择一种查询计划。

为了保证大家能够理解本章的大部分内容，我们接下来将介绍几种更少见和奇特的运算符，本章后面就会引用它们，同时还介绍如何在系统中表示查询。

1. 计算标量——项目

计算标量在查询优化器中称为 *Project（项目）*，是试图声明一组列、计算某个值或者限制查询树中其他运算符中的列的一种简单运算符。计算标量与 SQL 语言中的 *SELECT* 列表相对应，它们实际上不过度地将运算符与查询优化器关联，查询优化器不需要对它们进行太多的控制。查询优化器在优化期间结束它们与查询树的关联，试图将它们从处理联接顺序、索引选择和其他优化的其他查询优化器逻辑部分进行分离。

2. 计算序列——序列项目

计算序列在查询优化器中称为*序列项目*，该运算符与计算标量相同的地方在于它计算一个将被添加到输出流中的新值。它们的主要区别在于：计算序列操作有序流，同时包含为行保留的状态。例如排序函数使用这一运算符。这通过使用一种不同的物理操作实现，计算序列在查询优化器重新排序时强制附加限制条件。该运算符通常存在于排序和窗口化函数中。

3. 半联接

*半联接*这一术语来自学术数据库，用于描述执行联接但只针对某一输入返回值的运算符。查询处理器使用这种内部机制处理大部分子查询。SQL Server 按这种方式表示子查询是因为这种方式更容易分析查询的可能转换，而且半联接和标准联接的运行时实现方式是相似的。与普遍看法不同的是，子查询不是总在临时表中进行执行和缓存。对待半联接与对待标准联接的方式差不多是相同的。实际上，查询优化器具有可以将标准联接转换成半联接的转换规则。

一种常见的误解是认为使用子查询本身就是错误的。与大多数观点一样，这种观点也是不对的。子查询通常是表达要在 SQL 中执行什么内容的最自然方式，这也是它之所以是 SQL 语言的一部分的原因。有时人们指责子查询造成索引表性能不佳、缺少统计信息，或者谓词的编写方式对于查询优化器来说太生硬以至于不能使用其约束属性框架进行分析。与生活中的每件事一样，系统中可能有太多的子查询，尤其是它们在同一查询中多次复制时。因此，如果公司的开发习惯是“不使用子查询”，则请仔细检查系统——那些被指责为有很多其他问题但表面光鲜的系统。

清单 8-1 是适合使用子查询的一个示例。假设我们需要让销售人员跟踪库存系统，显示在最近 30 天内定货的每一位顾客的信息，从而可以向这些顾客发送电子邮件致谢。图 8-7、图 8-8 和图 8-9 显示了使用 3 种不同方法的查询计划，它们试图提交查询来回答这一问题。

清单 8-1 编写子查询计划时的常见错误

```
CREATE TABLE Customers(custid int IDENTITY, name NVARCHAR(100));
CREATE TABLE Orders (orderid INT IDENTITY, custid INT, orderdate DATE, amount MONEY);
INSERT INTO Customers(name) VALUES ('Conor Cunningham');
INSERT INTO Customers(name) VALUES ('Paul Randal');
INSERT INTO Orders(custid, orderdate, amount) VALUES (1, '2008-08-12', 49.23);
INSERT INTO Orders(custid, orderdate, amount) VALUES (1, '2008-08-14', 65.00);
INSERT INTO Orders(custid, orderdate, amount) VALUES (2, '2008-08-12', 123.44);
```

```
-- Let's find out customers who have ordered something in the last month

-- Semantically wrong way to ask the question - returns duplicate names (See Figure 8-7)
SELECT name FROM Customers C INNER JOIN Orders O ON C.custid = O.custid WHERE
DATEDIFF("m", O.orderdate, '2008-08-30') < 1

-- and then people try to "fix" by adding a distinct (See Figure 8-8)
SELECT DISTINCT name
FROM
Customers C
INNER JOIN
Orders O
ON C.custid = O.custid
WHERE DATEDIFF("m", O.orderdate, '2008-08-30') < 1;
-- this happens to work, but it is fragile, hard to modify, and it is usually not done
properly.

-- the subquery way to write the query returns one row for each matching Customer
SELECT name
FROM Customers C
WHERE
EXISTS (
SELECT 1
FROM Orders O
WHERE C.custid = O.custid AND DATEDIFF("m", O.orderdate, '2008-08-30') < 1
);
-- note that the subquery plan has a cheaper estimated cost result
-- and should be faster to run on larger systems
```

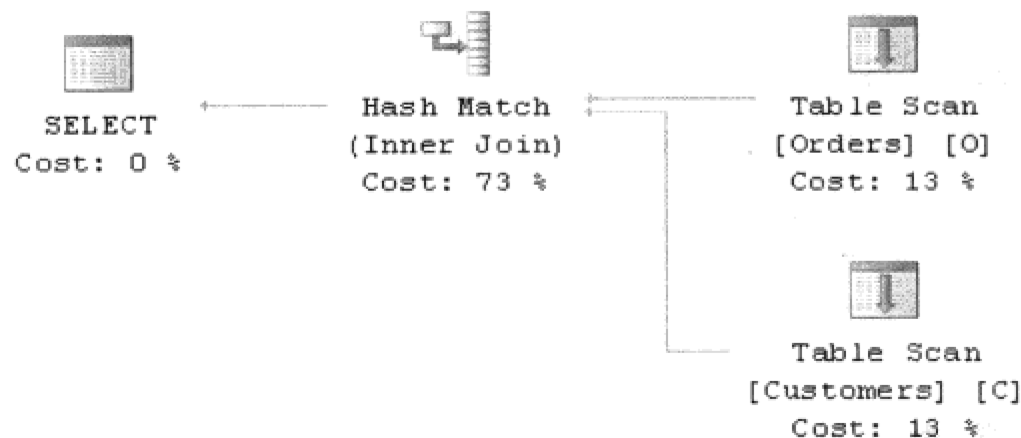


图 8-7 使用 *INNER JOIN* 替代子查询的查询计划

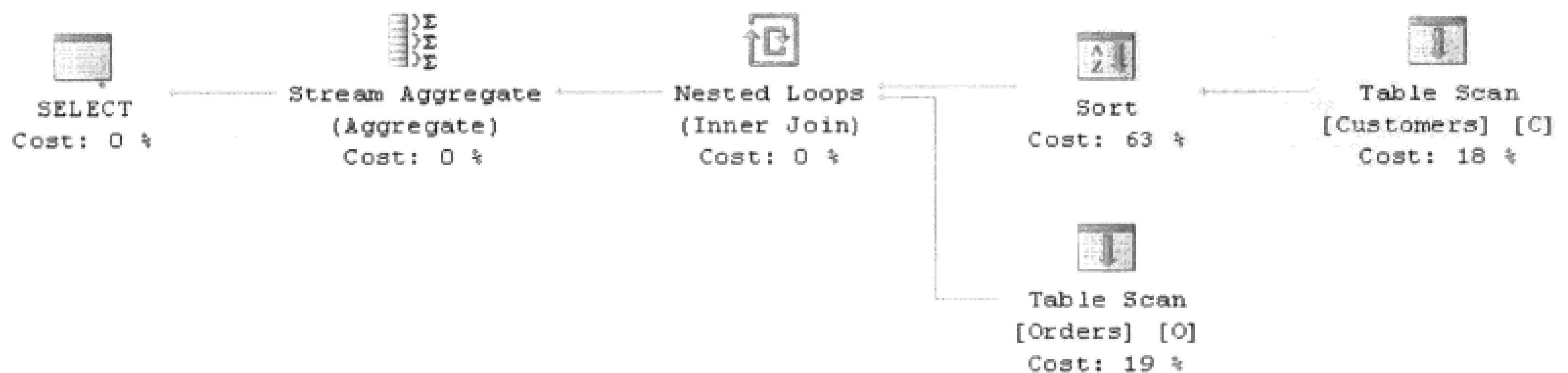


图 8-8 使用 *DISTINCT* 和 *INNER JOIN* 替代子查询的查询计划

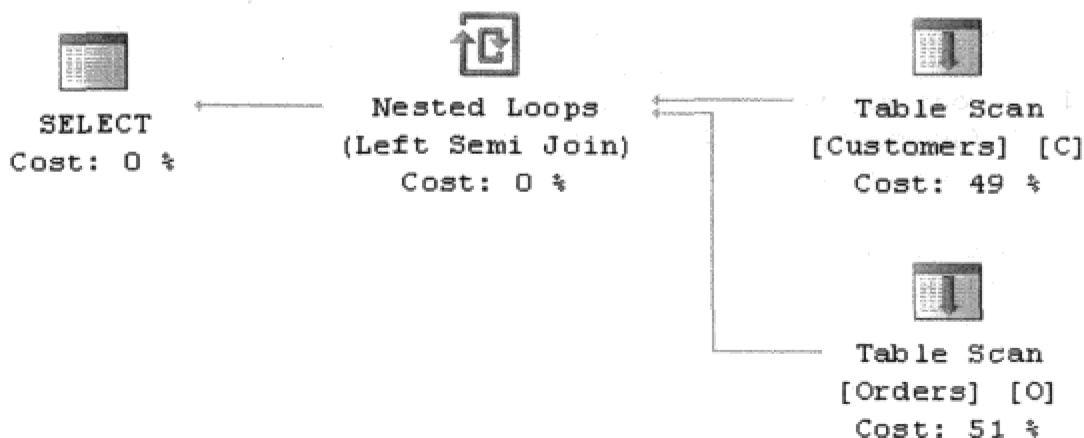
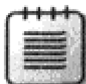


图 8-9 使用子查询的查询计划

在最后一种查询计划中，表 *Customers* 的匹配行被保存并通过 Left Semi-Join 运算符直接返回给用户。



注意：

Left Semi-Join 和 Right Semi-Join 必须处理存储在操作中的子行。对于对这些运算符含义有所疑惑的人来说，遗憾的是 SQL Server Management Studio 和以前工具中的计划表示法被置换了。在置换后的形式中，"左"子行是顶部子行，而"右"子行是底部子行。

4. Apply

CROSS APLLY 和 *OUTER APPLY* 是 SQL Server 2005 中的新增功能，它们表示一种特殊的子查询，其中左输入的值作为一个参数传递给右子行。这有时被称为*相关嵌套循环联接*，表示传递一个参数给子查询。该功能最常见的应用是处理索引查找联接，如清单 8-2 和图 8-10 所示。

清单 8-2 *APPLY* 查询示例

```
CREATE TABLE idx1(col1 INT PRIMARY KEY, col2 INT);
CREATE TABLE idx2(col1 INT PRIMARY KEY, col2 INT);
GO
SELECT *
FROM idx1
CROSS APPLY (
    SELECT *
    FROM idx2
    WHERE idx1.col1=idx2.col1
) AS a;
```

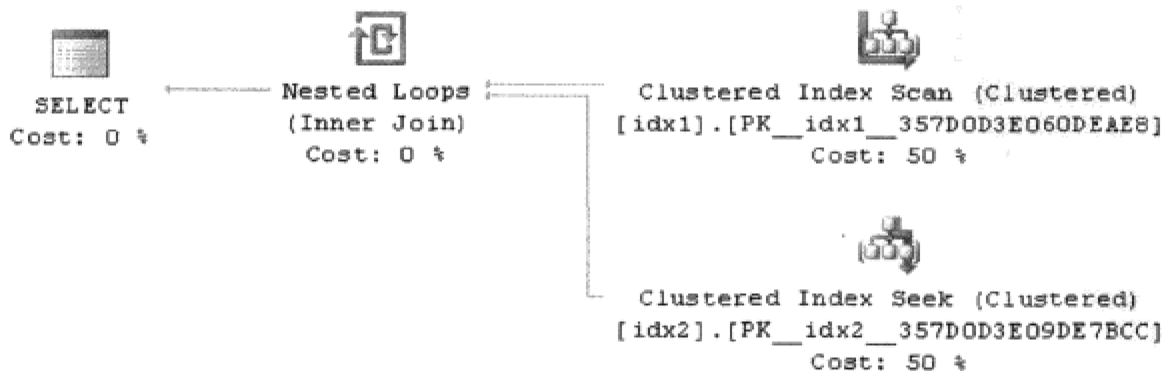


图 8-10 APPLY 查询计划

该查询逻辑上等价于一个 *INNER JOIN*，图 8-11 显示了与 SQL Server 2008 中一致的结果查询计划。

```
SELECT * FROM idx1 INNER JOIN idx2
ON idx1.col1=idx2.col1;
```

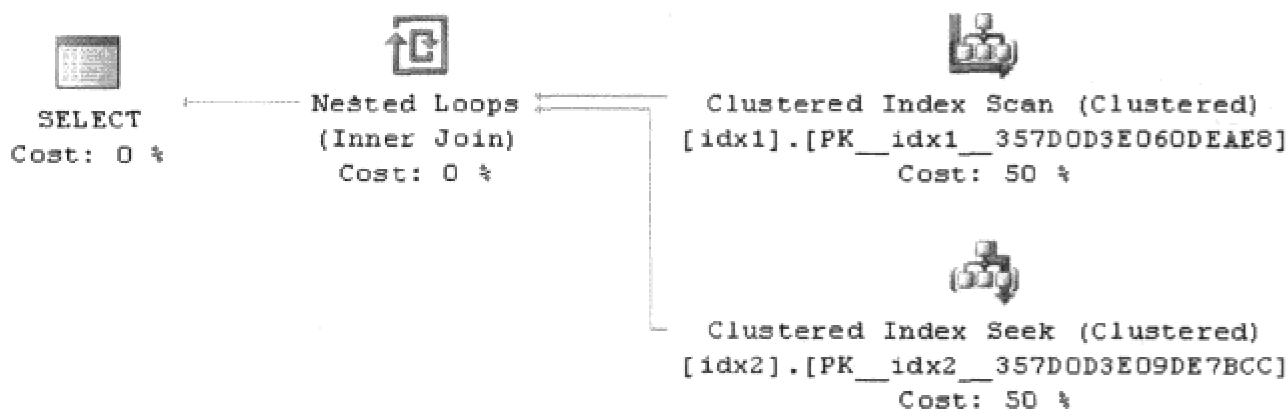


图 8-11 *INNER JOIN* 查询生成嵌套循环和查找计划

在两种情况下，外部表的一个值作为查找内部表的一个参数被引用。注意标准内部联接也能够生成一种查找，也就是说查询优化器会考虑将一个 JOIN 转换成一个 APPLY 作为优化过程的一部分。虽然这里完成的示例非常简单，您不必按照我们的方式编写查询，但是在某些更复杂的情况下这种语法会很有用。首先，有一种针对动态管理视图（DMV，包括本章后面“计划提示”一节中的一个示例）的公共模式，其中一个值通过一个交叉应用传递给一个管理函数。其次，可能有查询优化器的规则引擎不能重写一个简单内部联接来获得索引查找的罕见复杂情况。在这些情况下，重写查询使用 *CROSS APPLY* 对于通过一个不透明运算符手动向下传递一个参数来说非常有用。查询的语义可能随着这样的重写而改变，因此请一定保证在考虑进行这样的重写之前您已经理解了查询的语义。

Apply 运算符与面向过程语言中的函数调用非常类似。对于外部（左）的每一行来说，内部（右）的一些逻辑被评估，同时为右子树的调用返回零行或多行。查询优化器可能有时会删除相关性并将 Apply 转换成一种更通用的联接，在这些情况下，其他联接有时可以被重新排序来检索不同的计划选项。

5. 假脱机

SQL Server 有大量不同的专用假脱机。每一种都用于某些特殊的场合。从概念上来说，所有假脱机都执行相同的工作：从输入数据读取所有行、在内存中存储行或将输入数据输出到磁盘上，然后允许运算符从此缓存中读取行。假脱机用于生成行的副本，这对于某些更新计划中的事务一致性及通过缓存将在某查询中多次使用的复杂子表达式来提高性能而言非常重要。

最神奇的假脱机操作被称为*公共子表达式假脱机*。该假脱机具有一次写入后由查询中的多个不同子查询读取的能力。这是当前在最终查询计划中有多个双亲的唯一运算符。该假脱机在显示计划输出中多次显示而实际上只是*同一个*运算符。公共子表达式假脱机一次只能有一个客户端。因此，第一个实例填充假脱机，后面的每个引用都按顺序从这个假脱机中读取。第一个引用有子查询，而后面的引用出现在查询计划中作为查询树的叶结点。

公共子表达式假脱机大多用在大范围更新计划中，这一点将在本章后面介绍。但是，它们通常用于开窗聚合函数中。这些是不必像常规聚合计算那样折叠行的特殊聚合。清单 8-3 和图 8-12 显示一个公共子表达式假脱机如何用于存储中间查询输入，然后多次使用该中间查询输入作为查询树其他部分的输入。初始表假脱机从 *windows1* 中读取值，同时树中的后续分支为假脱机行提供多个分支。

清单 8-3　具有 OVER 子句的聚合使用公共子表达式假脱机

```
CREATE TABLE windowl(coll INT, col2 INT);
GO
DECLARE @i INT=0;
WHILE @i<100
BEGIN
INSERT INTO windowl(coll, col2) VALUES (@i/10, rand()*1000);
SET @i+=1;
END;

SELECT coll, SUM(col2) OVER(PARTITION BY coll) FROM windowl;
```

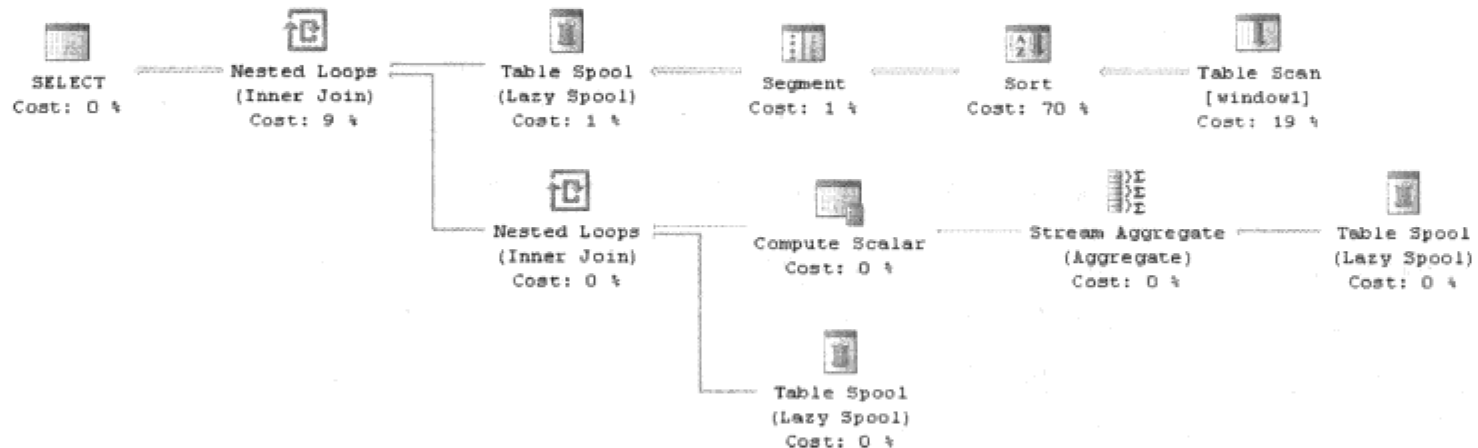


图 8-12　包含公共子表达式假脱机的查询计划

6. Exchange

Exchange 运算符用于表示查询计划中的并行性。根据是从线程中收集数据还是向线程分发行，交换运算符会在显示计划中显示为一种聚合流、重组流或分发流操作。存在多个行分发运算，并且每个运算符根据都有一种基于自己在查询中的上下文的优先算法。在 SQL Server 中，并行性存在于系统试图通过使用额外的 CPU 来加速的区域。图 8-13 显示了一个查询多线程并行扫描一个表的查询。

图 8-13　查询计划中的 Exchange 运算符

更多信息：

可以通过 *http://technet.microsoft.com/en-us/library/ms191158.aspx* 了解其他 SQL Server 2008 运算符的信息。

8.4　优化器架构

查询优化器包含很多执行不同功能的优化阶段。不同阶段帮助查询优化器执行优化过程中最早期的最高值操作。

查询优化中的主要阶段如图 8-14 所示。

- 简化。
- 琐碎计划。
- 自动统计信息创建/更新。
- 探索/实现（阶段）。
- 转换成可执行计划。

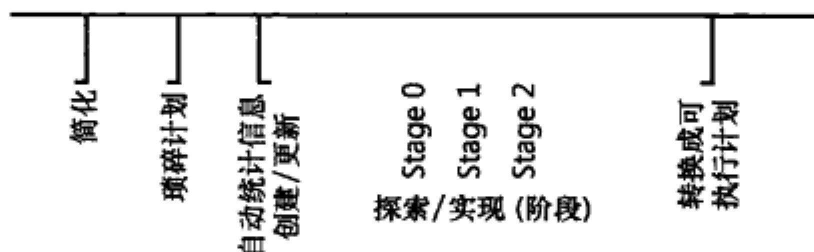


图 8-14 查询优化器管道

8.4.1 优化之前

SQL Server 查询处理器在真正优化过程开始之前要执行多个步骤。这些转换帮助建立容易推理的树形。视图扩展是一种主要的预优化活动。当某个查询编译为引用某个视图时，视图的文本也会从服务器的元数据中读取和解析。这种设计选择的一个后果是多次引用某个视图的查询将在优化之前多次扩展该视图。合并相邻 *UNION* 操作是另一种简化树的预优化转换，它会将语义上的两个 *UNION[ALL]*、*INTERSECT[ALL]*和 *EXCEPT[ALL]*子形式转换成一个可以有两个以上的子形式的单个运算符。这种重写会简化树架构并使查询优化器更容易编写影响 *UNION* 的规则。例如，分组 *UNION* 操作可以使删除重复行的工作变得更简单有效。

8.4.2 简化

在优化初期，树在简化阶段被规范化为从一种与用户语法紧密相关的格式转换为帮助后续处理的格式。例如，查询优化器通检测查询中的语义冲突并通过将查询重写成一种更简单格式的方式来删除语义冲突。此外，在这一部分执行的重写使后续操作（如索引匹配、计算列匹配及统计信息生成）更容易正确执行。

简化阶段也执行很多其他树重写工作，这些活动包括：

- 将联接组合到一起并根据每个表的基数数据选择一种初始联接顺序；
- 查找查询中可能使部分查询不能被执行的冲突；
- 执行重写 *SELECT* 列表来匹配计算列的必要工作。

本章前面的图 8-5 显示了一个冲突检测示例。

8.4.3 琐碎计划/自动参数化

SQL Server 中的主要优化路径是一种非常强大的、基于查询执行时间开销的模型。随着数据库越来越大及对这些数据库的查询变得越来越复杂，这种模型使 SQL Server 能解决越来越多的商业问题。运行这种模式所需的固定启动开销对于不希望执行复杂操作的应用程序来说可能是比较大的。制作一种包括最简单到最大型查询的路径可能颇具挑战性，因为需求和规范可能有很大的不同。

为了能够很好地满足小型查询应用程序的需要，SQL Server 添加了一个快速路径来标识不需要进行基于开销的优化查询。通常来说，该代码标识无需进行任何基于开销的选择查询。也就是说，只有一种

要执行的计划，或者有一种可以被标识的明显最佳计划。在这些情况下，查询优化器直接生成最佳计划并将其返回给将被执行的系统。例如，查询语句 *SELECT col1 FROM Table1* 对于没有任何索引的表来说有一种简单的最佳计划选择：从基表堆中读取行并返回给用户，如图 8-15 所示。

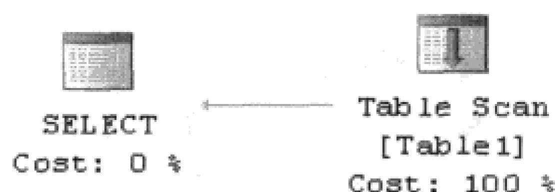


图 8-15　琐碎计划示例——表扫描

```
CREATE TABLE Table1 (col1 INT, col2 INT);
SELECT col1 FROM Table1;
```

SQL Server 查询处理器实际上进一步采纳了这一概念。当简单查询被编译和优化时，查询处理器视图将这些查询变成一种参数化查询。如果计划被确定为是琐碎的，则参数化查询将被变成一项可执行计划。接下来具有相同形状（除查询中明显位置的常量不同外）的查询只运行现有已编译的查询并完全不需要检查查询优化器。这样会大大加快 SQL Server 中小型查询应用程序的速度。

```
SELECT col1 FROM Table1 WHERE col2 = 5;
SELECT col1 FROM Table1 WHERE col2 = 6;
```

如果查看清单 8-4 中程序缓存中的查询文本，会发现实际上只有一种查询计划并且该计划已经被参数化。

清单 8-4　自动参数化的查询文本

```
SELECT text
    FROM sys.dm_exec_query_stats AS qs
    CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) AS st
WHERE st.text LIKE '%Table1%';

----------------------------------------
(@1 tinyint)SELECT [col1] FROM [Table1] WHERE [col2]=@1
```

如果检查该查询计划的 XML 计划，您会发现其中有一行表明该查询是一项琐碎计划。其他选项是完整的，表示执行的是基于开销的优化：

```
. . . <StmtSimple . . . StatementOptmLevel="TRIVIAL"> . . .
```

（XML 计划输出是冗长的，因此出于对空间的考虑，我们忽略了其中大部分内容。粗体代码表示查询优化器所做的选择。）

8.4.4　限制

琐碎计划优化是在 SQL Server 7.0 中引入的。虽然每个 SQL Server 版本的规则都稍有不同，但是所有版本都有完全跳过琐碎计划阶段同时只执行标准优化活动的查询。使用更复杂的功能可能使一项查询不再被认为是琐碎的，因为它们总有一种基于开销的计划选项或者很难标识是否是琐碎的。使一条查询不再被认为是琐碎查询的查询功能包括分布式查询、批量插入、XPath 查询、具有联接或子查询的查询、具有提示的查询、某些游标查询及对包含筛选索引的表的查询。

SQL Server 2005 添加了另一项功能*强制参数化*来进一步要求自动参数化查询。该功能参数化所有常量，忽略基于开销方面的考虑。该功能对于生成 SQL（您不能生成参数化查询）并且结果查询计划几乎总是一致（或者即使结果查询计划不同，计划也类似地执行）的应用程序来说最有用。尤其是应用程序

不能通过 DBA 管理服务器修改时值得考虑该功能。

该功能的优点是可以减少编译、编译时间及程序缓存中的计划数量。所有这些都能提高系统性能。另一方面，当不同的参数值引起选择不同的参数计划时，该功能的性能可能会降低。这些参数用于查询优化器的基数和属性框架，从而确定从每种可能的计划选项返回多少行，同时强制的参数化会阻止这些优化。因此，如果您认为应用程序会从使用强制的参数化中获益，可以通过试验查看应用程序是否会更好地工作。第 9 章将进一步介绍各种参数化选项。

8.4.5 备注——有效地探索多项计划

查询优化器的核心结构是备注。该结构帮助存储查询优化器中运行的所有规则的结果，同时还帮助引导对可能出现的计划进行搜索以快速地找到一种良好的计划并避免多次搜索一棵子树。这会加快编译过程并降低内存需求。实际上这还可以使查询优化器比其他没有类似机制的优化器运行更高级的优化。虽然该结构是查询优化器内部的，但是这部分基本操作介绍是为了使您能更好地理解查询优化器选择计划的方式。

备注存储一棵查询树中的运算符并利用逻辑指针表示树的边。假设有查询 *SELECT * FROM (A INNER JOIN B ON A.a=B.b) AS D INNER JOIN C ON D.c=C.c*，则可以画出如图 8-16 所示的树。

存储在备注中的同一个查询如图 8-17 所示。

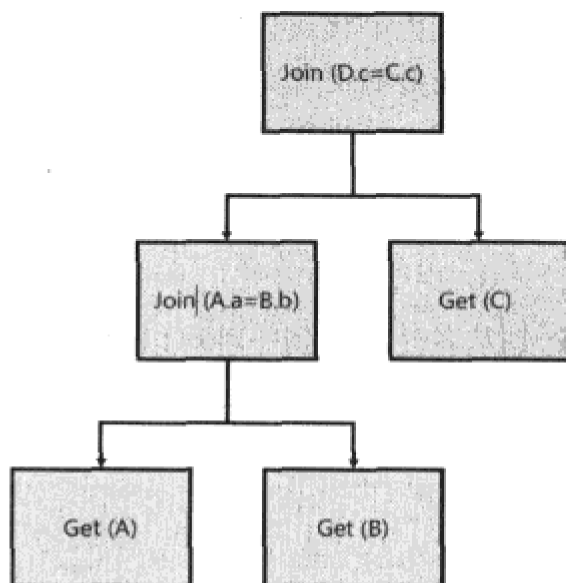


图 8-16　三表联接的树

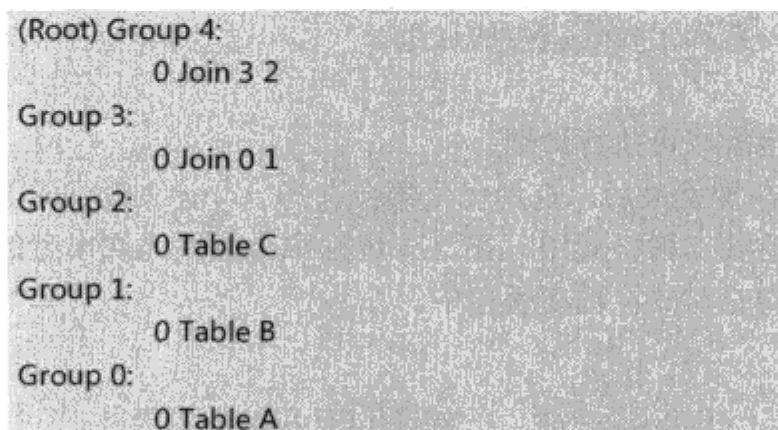


图 8-17　三表联接的初始备注布局

备注由一系列的组构成。当备注第一次被填充时，每一个运算符都被放到自己的组中。运算符之间的引用被修改为对备注中其他组的引用。在这种模型中，可能存储多个在备注的同一组中产生相同结果的替代项。使用这种更改，可以单独搜索最优子树以查看备注中高级组中的内容。逻辑属性存储在每个备注组中，并且组中的每个附加项可以与初始替代项共享该组的属性结构。

查询优化器探索的一类替代项是*联接管理*。[(A join B)join C)]等价于[A join (B join C)]。当该转换被查询优化器所考虑之后，图 8-18 说明了更新后的备注结构（粗体显示的是新内容）。

注意新替代项是如何适应一种结构的。之前备注中没有（B join C），因此创建了一个新组引用 B 和 C 的现有组。这种表示法在考虑多种可能查询计划的同时节省了很多内存，同时使查询优化器可能知道自己是否曾经考虑过一部分搜索空间，从而避免再次执行该项工作。（A join C）将是另一种有效的选项，虽然它没有显示。

规则是允许备注在优化过程中探索新选项的机制。联接关联示例作为一种匹配特殊模式的优化规则实现，然后创建了一种与第一个选项（为查询的该部分返回相同的结果）等价的新选项。按照定义来说，一种规则的结果可以进入与原始模式的根相同的组。

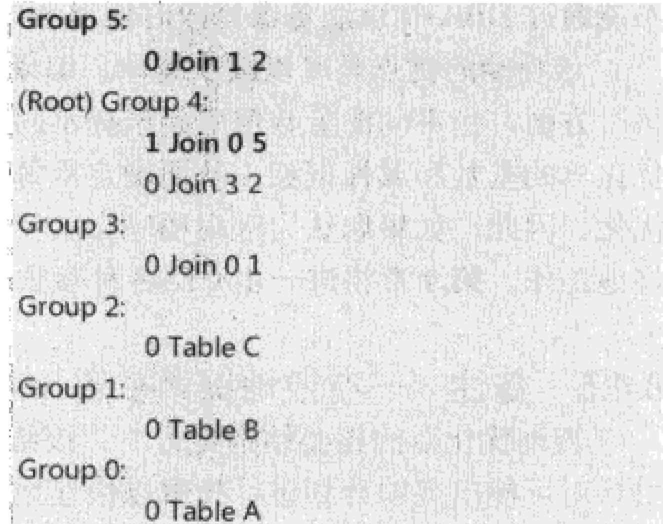

Group 5:
0 Join 1 2
(Root) Group 4:
1 Join 0 5
0 Join 3 2
Group 3:
0 Join 0 1
Group 2:
0 Table C
Group 1:
0 Table B
Group 0:
0 Table A

图 8-18 应用联接关联规则后的三表备注

一次优化搜索步骤分成两部分。在搜索的第一部分中，探索规则与逻辑树相匹配并生成新的等价可选逻辑树，它被插入到备注中。接下来运行实现规则，根据逻辑树生成物理树。生成物理树后，接下来会通过评估开销组件来确定该查询树的开销。得到的开销存储在该替代项的备注中。当为备注中的所有组生成所有物理选项和它们的开销时，查询优化器查找备注中开销最低的查询树并将其复制到一棵单独的树中。选中的物理树与该树的显示计划形式非常接近。

优化过程通过使用多个搜索阶段根据开销及有用性来分离规则，从而进一步优化。共 3 个阶段，每个阶段运行一系列的探索和实现规则。这些阶段配置成使小型查询快速优化，同时使开销更大的查询可能花更长时间来编译的更具挑战性的重写规则。例如，索引匹配在第一阶段执行，而索引视图的匹配通常在后面的阶段才执行。如果某个阶段已经找到了一个很好的计划，则查询优化器可以在该阶段后退出优化。通过比较目前为止发现的最好计划的估计开销与到目前为止所花费的优化时间进行计算，如果当前最好计划仍然开销很大，则会运行下一个阶段来查找一种更好的计划。这种模型允许查询优化器有效地为一系列工作负载生成计划。

转换成可执行计划

在搜索最后，查询优化器会选择一个计划返回给系统。该计划从备注复制到一种可以存储在程序缓存中的独立树格式中。在这一过程中会执行几次小型的物理重写。最后，该计划被复制到一片连续的空间中同时存储在程序缓存中。

8.5 统计信息、基数估计和开销

查询优化器利用一种具有每个运算符估计开销的模型来确定选择哪个计划。开销是由估计每个运算中处理的行数统计信息来决定的。默认情况下，统计信息是在优化过程中自动生成的，用于帮助生成这些基数估计。查询优化器同时还决定每个表中的哪些列需要统计信息。

一旦一组列被标识为需要统计的信息，查询优化器就会试图查找该列已有的统计信息对象。如果没有找到，系统会对表中的数据进行采样，从而创建一个新的统计信息对象。如果已经存在一个统计信息对象，则会对其进行检查，确定样例对于当前查询的编译来说是否足够新。如果认为已存在的统计信息对象已经过期，则使用一个新样例来重建统计信息对象。接下来对需要统计信息的每一列依次执行这一过程。

自动创建和自动更新统计信息的选项默认已经启动。实际上大多数人都会启用查询优化器的这些标志并统计信息行为：

```
ALTER DATABASE . . . SET AUTO_CREATE_STATISTICS {ON | OFF }
ALTER DATABASE . . . SET AUTO_UPDATE_STATISTICS {ON | OFF }
```

这些命令分别修改针对数据库中所有表的自动创建和自动更新统计信息的行为，如果统计信息的自动创建或更新被禁用，则查询优化器会在编译某条查询（查询优化器认为需要该信息）时在显示计划输出中返回一条警告信息。在这种操作模式下，DBA需要保证系统中的统计信息对象最新。

也可以对特殊操作使用提示来控制个别统计信息对象的自动更新行为：

```
CREATE INDEX . . . WITH (STATISTICS_NORECOMPUTE = ON)
CREATE STATISTICS . . . WITH (NORECOMPUTE)
```

虽然这些选项通常被设置为启动状态，但是在下面的情况下需要禁用创建或更新统计信息的功能。

- 当 DBA 已经明确决定更新统计信息而不是在白天自动更新这些对象时，数据库有一个维护窗口。这通常是因为 DBA 有理由相信统计信息改变时查询优化器会选择一种较差的计划。
- 数据库表非常大并且自动更新统计信息的时间太长。
- 数据库表有很多唯一值，用于生成统计信息的采样频率不足以获取生成一个良好的查询计划所需的所有统计信息。DBA 可以利用一个维护窗口以较高的采样频率（而不是默认采样频率，默认采样频率根据表的大小不同而变化）来手动更新统计信息。
- 数据库应用程序定义了一个很短的查询超时并且不希望自动统计信息造成某个查询比平均编译时间明显长很多，因为这样可能会引起查询因超时而被中止。

最后一种情况以一种巧妙的方式证明了这种情况可以中断应用程序。如果将一个 OLTP 应用程序中某个查询的超时时间设置为几秒钟，则通常这已经足够编译所有查询了（甚至是有自动统计信息的查询）。但是，随着数据库表的增长，对表进行采样以创建或更新统计信息的时间也会增加。最终执行该操作的总时间会超出查询超时的限制。由于每个查询都是作为用户事务的一部分进行编译，因此超时会强制事务中断和回滚。当对该表进行的下一个查询被编译时，会再次触发超时同时回滚整个查询。遗憾的是这样会引起应用程序意外失败，并且这通常是在部署应用程序之后发生的，因为这只是根据数据库大小设置的一种时间上的阈值。

为了实现该功能，SQL Server 2005 引入了一种称为*异步统计信息更新*（*ALTER DATABASE . . . SET AUTO_UPDATE_STATISTICS_ASYNC {ON | OFF}*）的功能，从而允许统计信息更新操作在不同事务环境中的后台线程中执行。这种模型的好处是可以避免重复的回滚问题。初始查询继续使用过期的统计信息编译查询并将其作为将被执行的计划返回。当统计信息被更新时，基于这些统计信息对象的计划会设置为失效并在下次使用时重新编译。

8.5.1 统计信息设计

统计信息存储在系统的元数据中，基本上由一个柱状图（表示某一列的数据分布）组成。不要混淆，有时人们说统计信息时实际上指的是柱状图。统计信息对象中的其他元素包括一些标题信息（包括对象被创建时的行采样数量）、检索树（表示字符列的数据分布）和密度信息（跟踪一列或多列上平均数据分布的信息）。

可以为 SQL Server 2008 中的大部分（但不是全部）数据类型创建统计信息。一般的规则是，支持比较（如>、=等）的数据类型都支持创建统计信息。如果它们在语言中不可比较，则查询优化器不需要分析其分布。不支持统计信息的数据类型示例包括旧样式的 BLOB（如 *image*、*text* 和 *ntext*）及一些基于用户定义数据类型（UDT）的新类型（不按字节顺序进行比较）。

此外，SQL Server 还支持对计算列进行统计，从而允许查询优化器对 col1+col2 或地理类型等一些更复杂类型的表达式进行基数估计（主要是在 UDT 上运行函数，而不是直接比较 UDT）。

清单 8-5 在某个持久计算列（在某个不可比较的 UDT 函数上创建的）上创建统计信息。当这种 UDT 方法在后面查询中使用时，查询优化器可以使用该统计信息更精确地估计基数。

清单 8-5 在某个持久计算列上使用 *DBCC SHOW_STATISTICS*

```
CREATE TABLE geog(col1 INT IDENTITY, col2 GEOGRAPHY);
INSERT INTO geog(col2) VALUES (NULL);
INSERT INTO geog(col2) VALUES (GEOGRAPHY::Parse('LINESTRING(0 0, 0 10, 10 10, 10 0, 0 0)'));
ALTER TABLE geog ADD col3 AS col2.STStartPoint().ToString() PERSISTED;
CREATE STATISTICS s2 ON geog(col3);
DBCC SHOW_STATISTICS('geog', 's2');
```

统计信息可以通过使用下面的代码查询系统元数据来进行枚举，结果如图 8-19 所示。

```
SELECT o.name AS tablename, s.name AS statname
FROM sys.stats s INNER JOIN sys.objects o ON s.object_id = o.object_id;
```

结果 消息

	tablename	statname
85	sysidxstats	_WA_...
86	sysidxstats	_WA_...
87	sysiscols	clst
88	sysiscols	nc1
89	sysiscols	_WA_...
90	sysiscols	_WA_...
91	sysiscols	_WA_...
92	sysendpts	clst
93	sysendpts	nc1
94	syswebmet...	clst
95	sysbinobjs	clst
96	sysbinobjs	nc1

图 8-19 查询输出列出统计信息对象

一旦被标识，统计信息对象就可以使用 *DBCC SHOW_STATISTICS* 命令进行查看，如图 8-20 所示。

结果 消息

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Express
1	_WA_Sys_00000001_436BFEE3	10 17 2009 4:21PM	2506	2506	182	0.9637306	4.434158	YES	NULL

	All density	Average Length	Columns
1	0.002717391	4.434158	name

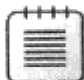
	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	NULL	0	2048	0	1
2		0	3	0	1
3	AF: aggregate function	3	1	3	1
4	AGGREGATE	0	2	0	1
5	ALL SETTABLE OPTIONS	0	3	0	1
6	ANSI null default	0	1	0	1
7	ansi_warnings	7	1	7	1
8	APPLICATION ROLE	2	2	2	1
9	Arabic	0	1	0	1
10	arithabort	0	2	0	1
11	arithignore	0	1	0	1
12	ASSEMBLY	0	2	0	1
13	ASYMMETRIC KEY	0	2	0	1
14	ASYMMETRIC KEY USER	2	2	1	2
15	AU	0	1	0	1

图 8-20 DBCC SHOW_STATISTICS 输出

一旦查询优化器确定需要创建一个新的统计信息对象或更新某个过期的现有对象，系统就会创建一个内部查询来生成一个新的统计信息对象。图 8-21 显示了在 SQL Server Profiler 输出中构建统计信息对象的查询。

```
Showplan All For Query Compile                                                    Micr

StmtText                                                                          StmtId
--------                                                                          ------
Stream Aggregate(DEFINE:([Expr1004]=STATMAN([s1].[dbo].[trace].[col1])))          0
  |--Sort(ORDER BY:([s1].[dbo].[trace].[col1] ASC))                               0
       |--Table Scan(OBJECT:([s1].[dbo].[trace]))                                 0
```

图 8-21 为柱状图生成的 SQL Profiler 显示计划输出



注意：

STATMAN 是一种与系统其他聚合函数工作方式类似的特殊内部聚合函数——很多行被按运算符分组的数据流组消费并传递给 *STATMAN* 聚合。*STATMAN* 过程存储柱状图、密度信息及在该操作期间创建的所有检索树的 BLOB。完成后，统计信息例子将被存储在数据库元数据中并被查询所使用（包括原来发布命令的查询，非对称统计信息的更新除外）。

优化器对数据库页进行采样来生成统计信息，其中包括每个采样页的所有行。对于小型表来说，所有页均被采样（也就是说构建柱状图时考虑所有行）。对于大型表来说，会对更小比例的页进行采样。为了保持柱状图的大小合理，限定总共有 200 个等级。如果在构建柱状图的同时检查 200 多个唯一值，则查询优化器会利用逻辑基于保留尽可能多分布信息的算法来降低等级。由于柱状图对于获取某个系统的非均匀数据分布来说最有用，因此它会试图保留获取最频繁使用的值的信息及它们比数据中最少使用的值的频度高多少的信息。

8.5.2 密度/频度信息

除柱状图之外，查询优化器还跟踪一组列唯一值的数量。当与创建表时看到的行的总数量相结合之后，还可以计算列中重复值的平均数量。该信息被称为*密度信息*，被存储在柱状图中。密度是利用 1/*频度*这一公式计算得到的，*频度*是表中每个值的副本的平均数量。该信息还会在调用 *DBCC SHOW_STATISTICS* 时返回。对于多行统计信息来说，统计信息对象为统计信息对象中列的每种组合（按照在 *CREATE STATISTICS* 语句中指定的顺序）存储密度信息。这样将存储多个组织的副本数量的信息。

在清单 8-6 中，我们将创建一个具有 30 000 行的两列表。

清单 8-6 多列统计信息

```
CREATE TABLE MULTIDENSITY (col1 INT, col2 INT);
go
DECLARE @i INT;
SET @i=0;
WHILE @i < 10000
BEGIN
        INSERT INTO MULTIDENSITY(col1, col2) VALUES (@i, @i+1);
        INSERT INTO MULTIDENSITY(col1, col2) VALUES (@i, @i+2);
        INSERT INTO MULTIDENSITY(col1, col2) VALUES (@i, @i+3);
        set @i+=1;
END;
GO
-- create multi-column density information
CREATE STATISTICS s1 ON MULTIDENSITY(col1, col2);
GO
```

在 *col1* 中有 10 000 个唯一值，每个值被复制了 3 次。在 *col2* 中，实际上有 10 002 个唯一值。对于多列密度来说，表中的每一组（*col1*，*col2*）都是唯一的。图 8-22 显示了为多列统计信息对象存储的数据。

```
DBCC SHOW_STATISTICS ('MULTIDENSITY', 's1')
```

	Name	Updated	Rows	Rows Sampled	Steps	Density
1	s1	Nov 27 2008 10:16AM	30000	30000	3	0.3333333

	All density	Average Length	Columns
1	0.0001	4	col1
2	3.333333E-05	8	col1, col2

图 8-22 *统计信息*对象中的多列密度信息

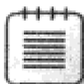
Col1 的密度信息是 0.0001. 1/0.0001 = 10 000，这是 col1 的唯一值数量。（*col1*，*col2*）的密度信息大约是 0.00003（数字以浮点形式存储并且是不精确的）。

现在我们利用与图 8-23 中密度信息相匹配的 *GROUP BY* 清单来检查 *GROUP BY* 操作的基数估计。对于这个查询来说，实际基数和估计基数完全匹配。

```
SET STATISTICS PROFILE ON
SELECT COUNT(*) AS CNT FROM MULTIDENSITY GROUP BY col1
```

	Rows	Executes	StmtText	EstimateRows
1	10000	1	SELECT COUNT(*) AS CNT FROM MULTIDENSITY GROUP ...	10000
2	0	0	|--Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(i...	10000
3	10000	1	|--Hash Match(Aggregate, HASH:([s1].[dbo].[MULTIDENSI...	10000
4	30000	1	|--Table Scan(OBJECT:([s1].[dbo].[MULTIDENSITY]))	30000

图 8-23 哈希聚合的 *STATISTICS PROFILE* 输出



注意：

我们已经记录了 *STATISTICS PROFILE* 输出中的列来显示实现 *GROUP BY* 操作的哈希匹配的 *EstimateRows* 列。

对于两个列上的一个查询组来说，可以看到估计值与密度计算中的值相匹配。图 8-24 中的 *STATISTICS PROFILE* 输出显示将估计值改为 30 000 行。

```
SET STATISTICS PROFILE ON
SELECT COUNT(*) AS CNT FROM MULTIDENSITY GROUP BY col1, col2
```

Rows	Executes	StmtText	EstimateRows
30000	1	SELECT COUNT(*) AS CNT FROM MULTIDENSITY GROUP ...	30000
0	0	|--Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(i...	30000
30000	1	|--Hash Match(Aggregate, HASH:([s1].[dbo].[MULTIDENSI...	30000
30000	1	|--Table Scan(OBJECT:([s1].[dbo].[MULTIDENSITY]))	30000

图 8-24 一个两列聚合的 *STATISTICS PROFILE* 输出

查询优化器实际上在计算某个运算符的输出基数时必须执行一个额外的步骤。由于统计信息通常是在使用它们的查询被编译之前创建的并且经常只使用数据的样例，因此存储在统计信息对象中的值通常

与查询被编译时的行数不完全匹配。因此查询优化器使用这两个值来计算满足该操作的行部分，接下来缩放到查询被编译时表中值的真实数量。

查询优化器不会精确公开基数估计的每一部分是如何计算的。但是，如果发现某个查询具有与运行查询时的真实情况相差很大的估计，统计信息数据配置文件可以帮助您确定查询优化器是否显示了错误信息。您可能需要更新统计信息以获得表中的新数据、创建具有更高采样频率的统计信息或者保证编译期间使用的信息是准确的。虽然 SQL Server 大部分情况下都会自动执行此操作，但是这通常是查找和修正计划选择不佳问题的好方法。

8.5.3 筛选的统计信息

作为 SQL Server 2008 中新增筛选索引功能的一部分，筛选的统计信息功能也同时被添加。这表示*统计信息*对象是根据某个筛选谓词在表中行的某个子集上创建的。创建一个筛选索引会自动创建一个与非筛选索引的行为匹配的筛选的统计信息对象。该信息是通过图 8-25 所示的 *sys.stats* 元数据视图公开的。

```
SELECT * FROM SYS.STATS
```

	object_id	name	stats_id	auto_created	user_created	no_recompute	has_filter	filter_definition
95	26157...	s1	2	0	1	0	0	NULL
96	26157...	_WA_Sys_00000002_0F975522	3	1	0	0	0	NULL
97	26157...	s3	4	0	1	0	1	([col2]>(5))
98	19930...	queue_clustered_index	1	0	0	0	0	NULL

图 8-25 SQL Server 2008 统计信息中的 *filter_definition* 表达式

筛选的统计信息的使用方式与传统统计信息类似——在查询编译早期确定需要分布的一组列。在表上为该查询定义的筛选谓词必须是要考虑的统计信息的统计信息对象 *filter_definition* 的一个子集。如果存在多种统计信息，则会使用其中限制最严格的一个。

筛选的统计信息可以避免基数估计中的一种常见问题，即由于列之间的数据相关性而使估计不准确。例如，如果创建一张名为 *CARS* 的表，其中有一列名为 *MAKE*，另一列名为 *MODEL*。例如，下表显示了 Ford 制作的多个汽车模型。

CAR_ID	*MAKE*	*MODEL*
1	Ford	F-150
2	Ford	Taurus
3	BMW	M3

此外，假设您希望运行如下一个查询：

```
SELECT * FROM CARS WHERE MAKE='Ford' AND MODEL='F-150';
```

当查询处理器试图估计一个 AND 子句中每个条件的可选性时，它通常会假设每个条件都是*独立的*。这样可以使每个谓词的替代项进行相乘，从而为整个 WHERE 子句建立整体可选性。对于这个示例来说，结果是：

```
2/3 * 1/3 = 2/9
```

实际的可选性对于该查询来说是 1/3，因为每个 F-150 都是一个 Ford。在某些数据集中这种估计错

误可能是非常大的。检测这样的统计信息相关性就计算性而言是一个开销很大的不可行问题，因此默认行为假定它们是独立的，即使这样也可能会使基数估计过程中产生某些错误。

筛选的统计信息通过在 *MAKE* 值为 *Ford* 时为 *MODEL* 列获取条件概率来解决这一问题。在使用这种方案获取大量统计信息对象时，可以有效地修正由于基数估计错误而引起的查询优化器选择执行效果很差的计划的大部分问题，尤其是当 WHERE 子句具有比较少的不同值时。

除了独立性假设之外，查询优化器还包含同时用于简化估计过程及如何使估计在所有运算符之间保持一致的其他假设。查询优化器中的另一种假设是一致性假设，即如果一系列值被考虑但是它们是未知的，则会假设它们均匀分布在它们所在的范围内。例如，如果一个查询的每个值具有不同参数的 IN 列表，则参数的值被假设为不分组。查询优化器中的最后一种假设是包含假设，即如果一系列值与另一系列值联接，则默认的假设是该查询正在被询问，因为这些范围重叠并且限制行。如果没有这种假设，则很多公共查询将会被低估并且会选择较差的查询计划。

8.5.4　字符串统计信息

SQL Server 2005 引入了一种被称为*字符串统计信息*或*检索树*的功能，用于改进字符串的基数估计。SQL Server 柱状图最多可以有 200 个等级或唯一值来存储某个表中整体分布的信息。虽然这对于很多数值类型来说可以很好地工作，但是字符数据类型通常有更多唯一值及更多依赖于对该类型有更深层次统计信息方面理解的大量函数，如 *LIKE*。200 个唯一值通常不足以为字符串提供精确的基数估计，同时存储表外部的大量字符串可能会使用大量的空间。检索树用于以一种空间有效方式存储列中字符串示例。

检索树没有被记录，但是检索树通常按照如下方式工作。

如果我们有一个包含如下值的列：

```
ABC
AAA
ABCDEF
ADAD
BBB
```

该结构的检索树如图 8-26 所示。

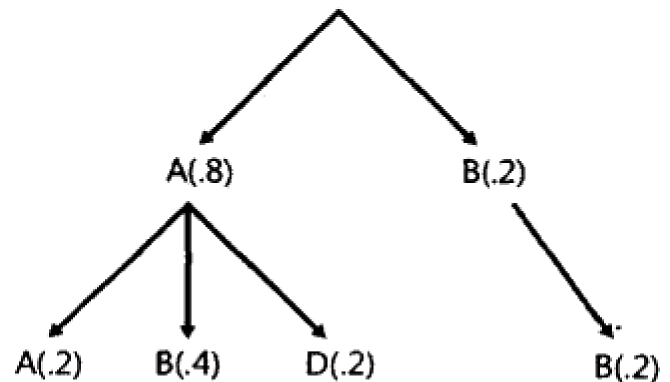


图 8-26　一棵检索树示例

SQL Server 自动存储列中字符串的一个*示例*，即使这样也一定不会占用太多的空间。SQL Server 还有一些关于检索树中列出每个子串相对频率的概念。总的来说，这提供了存储远远超过 200 个需要频率信息的唯一子字符串的能力。

8.5.5　基数估计细节

优化期间，查询中的每个运算符都会被评估来估计该运算符处理的行数。这样可以帮助查询优化器根据不同查询计划的开销来进行适当的调整。该过程是从下到上执行的，基表基数和统计信息用做它上面树结点的输入。该过程继续在查询树上向上执行，这一计算确定从显示计划信息中某个查询返回的预计行数。

清单 8-7 包含用于解释基数偏离过程工作方式的一个示例。

清单 8-7　基数估计示例

```
CREATE TABLE Table3(col1 INT, col2 INT, col3 INT);
GO
```

```
SET NOCOUNT ON;
BEGIN TRANSACTION;
DECLARE @i INT=0;
WHILE @i< 10000
BEGIN
INSERT INTO Table3(col1, col2, col3) VALUES (@i, @i,@i % 50);
SET @i+=1;
END;
COMMIT TRANSACTION;
GO
SELECT col1, col2 FROM Table3 WHERE col3 < 10;
```

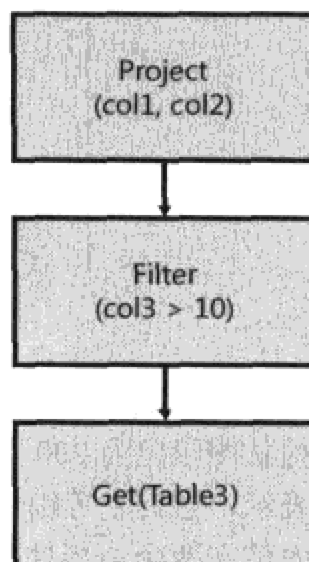


图 8-27 基数估计的一棵逻辑查询树示例

该查询在查询处理器中使用图 8-27 所示的树表示。

对于这一查询来说，Filter 运算符请求参与谓词（在这一查询中是 *col3*）运算的每一列上的统计信息。请求向下传递给 *Table3*，在 *Table3* 上创建或更新一个合适的统计信息对象。该统计信息对象接下来传递给筛选器以确定运算符的可选性。可选性是预期满足谓词并接下来传给用户的行。可选性用于将估计适当地由样例改为当前行数，因为当前行数可能与统计信息对象被创建时有所不同并且统计信息对象可能只是根据行的一个样例创建的。

一旦某个运算符的可选性被计算，则会与查询的当前行数相乘。筛选操作的可选性由为列 *col3* 加载的柱状图决定，如图 8-28 所示。

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows
1	_WA_Sys_00000003_1088795B	Nov 27 2008 10:30AM	10000	10000	50	0	4	NO	NULL	10000

	All density	Average Length	Columns
1	0.02	4	col3

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	0	0	200	0	1
2	1	0	200	0	1
3	2	0	200	0	1
4	3	0	200	0	1
5	4	0	200	0	1
6	5	0	200	0	1
7	6	0	200	0	1
8	7	0	200	0	1
9	8	0	200	0	1
10	9	0	200	0	1
11	10	0	200	0	1

图 8-28 使用一个柱状图来估计基数

我们已经对该示例使用了一个综合数据分布来更轻松地跟踪计算。由于我们已经在 *col3* 上创建了从 0 到 49 的均匀分布，因此 10/50 的值小于行的 10%或 20%。因此，该查询中这一筛选条件的可选性是 0.2，从该筛选器得出的行数结算结果是：

（#下面操作符中的行）×（该运算符的可选性）

10000×0.2=2000 行

我们可以通过查看图 8-29 所示的显示计划信息验证这一计算。

该运算符的估计通过查看柱状图、计算与标准匹配的采样行数（在这里，我们有 10 个具有 200 个与筛选条件相配的相等行的柱状图等级）实现。接下来符合条件的行数（2000）按照柱状图创建时采样到的行数进行规范化以创建运算符的可选性（0.2）。然后与表中的当前行数（10 000）相乘得出估计的查询输出基数。继续对其他筛选器条件进行基数估计，通常只是将结果相乘以估计每种条件的整体可选性。

Misc	
Defined Values	[s1].[dbo].[Table3].col1, [s1].[dbo].[Table
Description	Scan rows from a table.
Estimated CPU Cost	0.0110785
Estimated I/O Cost	0.0232035
Estimated Number of Executions	1
Estimated Number of Rows	2000
Estimated Operator Cost	0.034282 (100%)

图 8-29　运算符行估计基数示例

柱状图另一个有用的方面是 *RANGE_ROWS*、*DISTINCT_RANGE_ROWS* 和 *AVG_RANGE_ROWS*。因为柱状图限定为 200 个等级，因此被查询的某些值可能没有在柱状图等级中显示出来。这些值由 *RANGE* 值表示，它们是等级值之间的行数。对于与柱状图中等价（EQ）行不匹配的查询条件来说，会假设范围内的值在两个边界柱状图等级之间的区域内均匀分布。这一部分由假设决定并用于生成可选性，与前面的示例一样。

虽然大部分运算符的工作机制与筛选器类似，但是其他一些运算符需要其他机制来进行更好的基数估计。例如，*GROUP BY* 实际上不会试图确定应该使用柱状图的哪些部分来估计运算符的可选性。相反，它需要确定一组列唯一值的数量，如图 8-30 所示。这一信息可以通过查看柱状图进行估计，但是在统计信息对象中有另一种机制来帮助您快速执行这一计算。密度信息存储在第二个结果集的柱状图中，对于 col3 来说这里是 0.02。

```
SELECT COUNT(*) FROM Table3 GROUP BY col3;
```

Misc	
Build Residual	[s1].[dbo].[Table3].[col3] = [s1].[dbo].[Tab
Defined Values	[Expr1007] = Scalar Operator(COUNT(*))
Description	Use each row from the top input to build a
Estimated CPU Cost	0.0834958
Estimated I/O Cost	0
Estimated Number of Executions	1

图 8-30　*GROUP BY* 基数估计

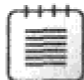
这是表中任意值平均重复数的一种表示法。换言之，这可以告诉我们如何使用总行数计算组数。对于这一简单的 *GROUP BY* 查询来说，行的预计值是(1/0.02)×(10 000/10 000) = 50，与我们创建脚本中预期的组数相匹配：

```
GROUP BY Selectivity = (1/density)
GROUP BY Card. Estimate = (Input operator) * (selectivity)
```

当一个多列统计信息对象被创建时，它会按照统计信息对象的顺序对正在被估计的列计算密度信息。因此在(*col1, col2, col3*)上创建的一个统计信息对象存储了(*(col1)*、*(col1, col2)和 (col1, col2,col3)*)的密度信息。这可以用于对在该表上进行 *GROUP BY col1*、*GROUP BY col1, col2 或 GROUP BY col1, col2, col3* 的查询计算基数估计。

我们可以在图 8-31 所示的 *DBCC SHOW_STATISTICS* 结果中看到这一计算。

```
CREATE TABLE Table4(col1 int, col2 int, col3 int)
GO
DECLARE @i int=0
WHILE @i< 10000
BEGIN
INSERT INTO Table4(col1, col2, col3) VALUES (@i % 5, @i % 10,@i % 50);
SET @i+=1
END
CREATE STATISTICS s1 on Table4(col1, col2, col3)
DBCC SHOW_STATISTICS (Table4, s1)
```



注意：

SQL Server 不会自动为这样的多列情况创建统计信息（除非在索引创建中），因此我们需要手动为该示例创建统计信息对象。

多列密度信息很重要，因为利用它可以获取同一表中列之间的相关性数据。默认情况下，如果假设每一列都是完全独立的，则可能会期望返回很多不同的组，因为添加到分组列中的每一列都会添加越来越多的唯一性（以及对 *GROUP BY* 运算符越来越少的可选性）。但是，这里我们会发现 *GROUP BY* 的可选性与前面的示例相同，都是 50 组。多列密度中获取的数据可以用于使基数估计更准确。

如果我们创建具有前两列中随机数据的一个类似表，则密度看起来会完全不同，如图 8-32 所示。

	All density	Average Length	Columns
1	0.2	4	col1
2	0.1	8	col1, col2
3	0.02	12	col1, col2, col3

图 8-31 多列密度信息

	All density	Average Length	Columns
1	0.01	4	col1
2	0.0001579031	8	col1, col2
3	0.0001009387	12	col1, col2, col3

图 8-32 随机数据分布的多列密度

这表示 *col1*、*col2* 和 *col3* 的每种组合实际上都是唯一的。通过检查基数估计过程中的各种输入，确定编译期间计划是否使用了适当的信息。

关于基数估计的详细信息有很多，我们不能在这里一一介绍。大部分详细信息在不同版本中都稍有不同，同时大部分信息没有公开或记录，从而不能有效地跟踪精确的计算。但是理解统计信息和基数估计机制仍然非常有用，这样可以执行计划调试和提示（本章后面将对此进行解释）。

8.5.6 限制

SQL Server 的基数估计通常都是非常好的。但遗憾的是，很难制作一种适合所有应用程序每个查询的模型。虽然大部分都是内部细节，但是其中一些信息还是值得关注的，这些信息可以使您知道前面介绍的计算不是在每个查询中都能很好地工作。

- **一个运算符中有多个谓词。**多个谓词的可选性在基数估计期间进行相乘，从而确定整个运算符的结果估计。也就是说假设谓词在统计上是独立的。实际上大部分数据在列之间具有某种统计上的依赖关系。随着查询中谓词数量的增加，查询优化器实际上不会直接对所有可选性进行相乘并假设这些不同的谓词是相关的。因此具有很多谓词的运算符的可选性可能比预期效果好很多。
- **深度查询树。**基于树的基数估计过程是好的，但是查询树底部出现的任何错误都会随着在查询树继续向上进行计算而扩大到更多的运算符中。最终在这些计算中产生的错误会代替使用柱状图的值来计算基数估计。因此，非常深的查询树最终停止使用柱状图作为查询树的高级部分，

同时可能使用更简单的试探法使基数估计避免在很可能无效的数据上进行信息假设。

- **不常见的运算符**。查询优化器有很多运算符。其中大部分常用的运算符在产品的多个版本中高度支持。但是，一些使用很少的运算符不必对每一种情况具有相同的支持度。因此如果您正在使用一种不常见的运算符或者最近版本才引入的运算符，则基数估计可能没有像使用大部分核心运算符那么好。在这种情况下，可以使用 *SET STATISTICS PROFILE ON* 对估计进行双重检查，查看估计是否非常接近于预期值。在很多情况下估计值都是不正确的，影响通常会被缓解，因为专用运算符不会有很多计划选项，同时基数估计中错误的影响可能会被降低。

8.5.7 成本计算

估计基数的过程利用逻辑查询树完成。*成本计算*是确定每一种潜在计划选项运行所需的时间的过程，并且对于考虑的每种物理计划单独执行。假设查询优化器考虑返回相同结果的多个不同物理计划，则这样做会很有意义。成本计算是在哈希联接和循环联接之间或一种联接顺序和另一种联接顺序之间进行选择的组件。

成本计算的理论实际上非常简单。使用基数估计及关于表中每一列最大宽度和平均宽度等一些其他信息可以确定每一个数据库页上适合存储多少行。该值接下来被转化成某个查询所需完成的大量的磁盘读取操作。每个运算符的整体开销接下来被添加进来以确定整体查询开销，同时查询优化器能够从优化期间被考虑的计划组中选择最快（开销最低）的查询计划。

实际上，成本计算不是这么简单。顺序 I/O（磁盘块顺序地存储在磁盘上，因此不需要等待磁盘头移动到一个新的轨道或等待磁盘片的一次完整旋转）和随机 I/O（不能保证这两种情况是正确的）之间有开销差异。此外，一些查询非常大，因此数据可以被读取到内存中并且在一次查询期间被读取多次。这些额外的读取通常能够从基于内存的页面缓冲池中读取页面，从而避免从磁盘上读取。而且，一些查询可能占用比服务器为该查询提供的内存更多的内存——此时开销组件需要确定将某些页替换出缓冲池并且必须进行重新读取，可能是随机也可能是顺序进行。优化器利用逻辑来考虑所有这些条件，同时确定某个运算符实际开销的过程可能会花一些时间进行计算。所有这些考虑可以保证 SQL Server 更好地为每个查询选择一种良好的查询计划。

为了保证查询优化器更加一致，开发团队在创建开销模型时使用了多种假设。首先，假设查询从一个冷缓存开始，也就是说查询处理器假设查询的每个初始 I/O 需要从磁盘读取数据。在很少情况下（通常是小型 OLTP 查询），这可能会使查询优化器选择一种为完成查询所需的初始 I/O 数量进行优化的稍慢计划。冷缓存假设是允许查询处理器更一致地生成计划的一种简化，但是它与用于比较计划和现实的数学模型稍有不同。其次，假设随机 I/O 均匀地分散在表或索引的页上。如果一个基于非索引的表（一个堆）有 100 个磁盘页并且查询正在对堆进行 100 次基于书签的随机查找，则查询优化器假设在该查询中出现 100 次随机 I/O，因为查询优化器假设每个目标行位于一个单独的页面上。与本章前面介绍的统计信息列相关性示例类似，这种假设不一定总成立。实际行可能物理聚集在相同页上（可能它们都在相同的时间被插入并且在相邻的页上结束），同时可能只需要 5 次 I/O 来读需要的行。此时查询优化器会过多使用该查询。这种情况也很少发生，但是有必要知道用于成本估计的数学模型只是一个模型。在模型不能适当工作的极少情况下，查询提示可以帮助强制一种不同的查询计划。

查询优化器在成本估计模型中内建了其他假设。一种假设与客户端如何读取查询结果有关。成本估计假设每个查询读取查询结果中的每一行。但是，有些客户端只读取几行后就结束查询。例如，如果您正在使用某个应用程序在屏幕上显示行的页，接下来该应用程序可能读取 40 行，虽然原始查询可能返回

10 000 行。如果查询优化器知道用户使用的行数，则它可能对计划选择过程中的数量进行优化来选择一种更快的计划。一般来说，这会使查询优化器从使用运算符（如哈希联接，这种联接在查询开始时有一种更大的启动开销）向嵌套循环联接（启动开销更低但是每行的开销更大）转换。

SQL Server 为这种情况提供了一种被称为 FAST N 的提示。如果某个用户在查询中一般只读取行的一个子集，则可以向查询传递一个 OPTION（FAST N）来通知查询优化器估计返回 N 行（而不是整个结果集）的成本。清单 8-8 包含一个示例显示 FAST N 提示，这样会选择一种没有 FAST N 提示的哈希联接。图 8-33 显示了应用提示时选择的一个循环联接。

清单 8-8 FAST N 示例

```
CREATE TABLE A(col1 INT);
CREATE CLUSTERED INDEX i1 ON A(col1);
GO
SET NOCOUNT ON;
BEGIN TRANSACTION;
DECLARE @i INT=0;
WHILE @i < 10000
BEGIN
INSERT INTO A(col1) VALUES (@i);
SET @i+=1;
END;
COMMIT TRANSACTION;
GO
SELECT A1.* FROM A as A1 INNER JOIN A as A2 ON A1.col1=A2.col1;
SELECT A1.* FROM A as A1 INNER JOIN A as A2 ON A1.col1=A2.col1 OPTION (FAST 1);
```

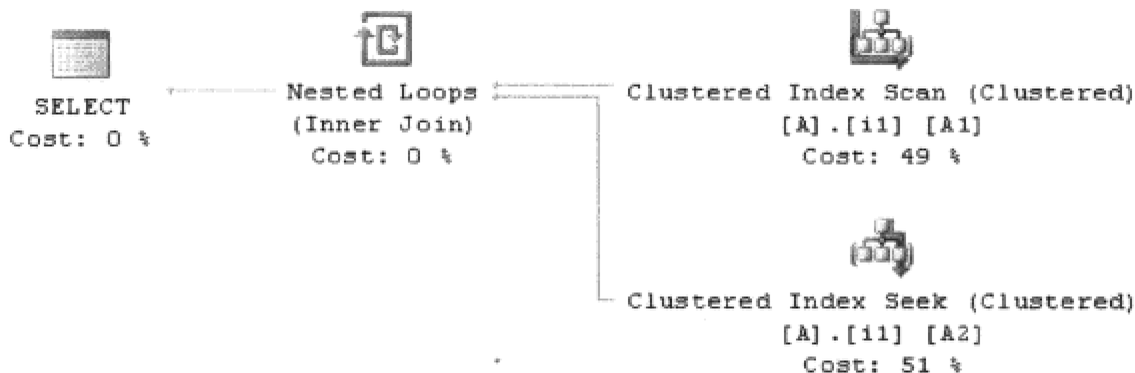


图 8-33 循环联接计划（具有 FAST 1 提示）

8.6 索引选择

索引选择是查询优化最重要的方面之一。索引匹配的基本思想是从一个 WHERE 子句、联接条件或查询中的其他限制操作中选择谓词，并将可以在某个索引上执行的操作进行转换。在索引上可以进行的两个基本操作是：

- 查找（索引键上的某个值或一系列值）；
- 扫描索引（向前或向后）。

对于查找来说，初始查询从 B+树的根开始，并根据索引键在树上向下导航直到找到索引所需的位置。一旦完成，查询处理器可能会对匹配谓词的所有行进行迭代，或者直到找到序列中的最后值为止。由于 B+树中的叶结点在 SQL Server 中是链接在一起的，因此如果中间的 B+树结点已经遍历完，则可以使用

该结构顺序对行进行扫描。

查询优化器的工作是计算哪些谓词可以被应用到索引上以快速地返回行。某些谓词可以被应用到索引上，而其他谓词则不能。例如，*SELECT col1, PKcol FROM MyTable WHERE col1=2* 查询有一个<列> = <常量>这种形式的谓词。如果该列上有一个索引，则这种模式可能与一次检索操作相匹配。生成的结果替代项用于在非聚集索引上执行一次检索并返回匹配行（如果有的话）。图 8-34 显示了查询优化器生成的一个基本检索计划。

```
CREATE TABLE idxtest2(col2 INT, col3 INT, col4 INT);
CREATE INDEX i2 ON idxtest2(col2, col3);

SELECT col2, col3 FROM idxtest2 WHERE col2=5
```

图 8-34　索引检索计划

查询优化器也可以对多列索引应用复杂的谓词，只要该操作可以被转换成开始和结束索引键。图 8-35 显示了一个多列索引计划，您可以在 Management Studio 中查看该运算符的属性时查看使用的谓词。

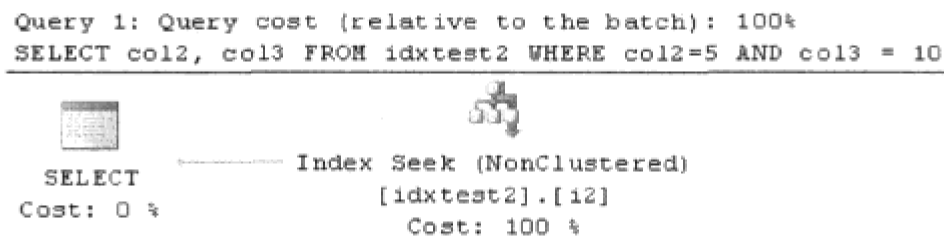


图 8-35　多列索引查找计划

可以转换成索引操作的谓词通常被称为*可求值（sargable）*。这表示谓词的形式可以转换成一个索引操作。永远不会匹配或不匹配选中的索引谓词称为*非可求值谓词*。非可求值谓词将会在索引检索或序列扫描操作之后应用，使查询可以返回与所有谓词相匹配的行。比较容易混淆的地方在于，SQL Server 通常在查询树的检索/扫描运算符中估计非可求值谓词。这是一种性能优化——如果不进行这样的工作，那么执行的一系列步骤将如下。

（1）检索运算符：在一棵索引的 B+树中检索一个键值。

（2）闩上页。

（3）读取行。

（4）释放页面上的闩锁。

（5）将行返回给筛选运算符。

（6）筛选器：估计行上的非可求值谓词。如果符合条件，则将行传递给父运算符。否则回到第 2 步获得下一个候选行。

这比最优化慢，因为将行返回给不同的运算符需要将不同的指令和数据加载到 CPU 中。通过在某个地方保存逻辑性，估计该查询所需的整体 CPU 开销会降低。SQL Server 中的实际操作如下。

（1）检索运算符：在一棵索引的 B+树中检索一个键值。

（2）闩上页。

（3）读取行。

（4）应用非可求值谓词筛选条件。如果该行没有传递筛选条件，则回到第 3 步，否则转到第 5 步。

（5）释放页上的闩锁。

（6）返回行。

这称为*推式非可求值谓词*（该谓词从筛选器中被推入到查找/扫描）。这是一种物理上的优化，但是可以在处理多行的查询中出现。

并不是所有谓词都可以在查找/扫描运算符中被估计。因为闩操作将防止其他用户查看系统中的页，因此这种优化为执行开销很低的谓词所保留。这被称为*非可推入非可求值谓词*。这样的示例包括：

- 大型对象上的谓词（包括 varbinary(max)、varchar(max)、nvarchar(max)）；
- CLR 函数；
- 一些 T-SQL 函数。

谓词可求值性是数据库应用程序设计中要考虑的一个重要方面。系统执行效果很差的一个原因就在于针对此数据库的应用程序的编写方式使谓词不可求值。在很多情况下，如果问题很早就被发现，就可以避免这种现象，修正这一问题有时还可以提高数据库应用程序性能。

SQL Server 在试图对查询中的可求值谓词应用索引时会考虑很多公式。对于 AND 条件（WHERE *col1=5* AND *col2=6* AND …）来说，SQL Server 会试图执行如下工作。

（1）给定一个所需查找相等列、查找非相等列及需要满足查询但是没有谓词的列的列表，首先试图查找一个与该请求完全匹配的索引。如果存在一个这样的索引，则使用该索引。

（2）试图查找满足相等条件的一组索引并对所有这些索引执行一次内联接。

（3）如果第 2 步没有找到所有需要的列，则考虑根据目前为止解决方案中包含的索引中的列与其他所有索引进行联接。

（4）最后，与基表进行一次联接以获得剩余的所有列。

在所有情况下，每种方案的开销都会被考虑并且只返回开销最低的方案。因此，与很多索引联接在一起的某个方案只有在被认为是比扫描基表中的所有行开销更低时才会被使用。其次，这种算法只在查询树本地执行。即使查询优化器利用该过程生成了一种特殊的替代项，最终也可能不是最终查询计划的一部分。成本计算用于确定开销最低的完整查询计划。因此，对于选择索引来说这不是一种基于规则的机制。这是一种探索法，是用于帮助选择有效查询计划的广义成本计算基础结构的一部分。

8.6.1 筛选索引

SQL Server 2008 引入了创建具有简单谓词（限制索引中包含的行）的索引功能。初看起来，该功能是已经包含在索引视图中的功能的一个子集。但是，该功能有其存在的价值。首先，索引视图使用和维护起来开销更大。其次，索引视图功能的匹配功能并不是在所有 SQL Server 版本中都被支持。再次，很多不同的 SQL Server 用户有时只是使用比标准索引功能稍微复杂一些的功能，因此他们实际上对于一个完整的索引视图方案不感兴趣。因此，虽然索引视图仍然是一种非常有用的功能，但是它们一般对更经典的关系查询预计算更有用。

筛选索引在对 *CREATE INDEX* 语句使用一个新的 WHERE 子句时创建。

清单 8-9 显示了如何创建一个索引及如何在查询中使用一个索引。图 8-36 和图 8-37 显示了某个查询的结果查询计划（分别包括有筛选索引和没有筛选索引的情况）。

清单 8-9 筛选索引示例

```
CREATE TABLE testfilter1(col1 INT, col2 INT);
go
DECLARE @i INT=0;
SET NOCOUNT ON;
BEGIN TRANSACTION;
WHILE @i < 40000
BEGIN
INSERT INTO testfilter1(col1, col2) VALUES (rand()*1000, rand()*1000);
SET @i+=1;
END;
COMMIT TRANSACTION;
go

CREATE INDEX i1 ON testfilter1(col2) WHERE col2 > 800;

SELECT col2 FROM testfilter1 WHERE col2 > 800;
SELECT col2 FROM testfilter1 WHERE col2 > 799;
```

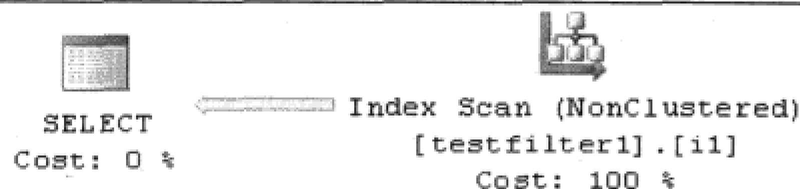


图 8-36 查询计划中使用的筛选索引

图 8-37 由于没有筛选条件而没有使用的筛选索引

第一个选择查询的开销是 0.0141293，而第二个查询的估计开销是 0.112467。筛选索引获益于具有更少的行并且也比基表更窄，因此也有更少的页面。当您知道在大型表（此时空间是一个问题）上使用查询的特定约束时，这种索引可能非常有用。

SQL Server 对标量结构（可用于表示 *CREATE INDEX* 命令的筛选条件）强制了很多限制。这些限制条件在很大程度上是由查询优化器的域属性框架在匹配索引时可以轻松使用的内容所决定的。因此，系统中某些更复杂的部分在这一版本中不被支持，因为没有一种有效匹配这些索引的方法。

筛选索引可以处理以下几种情况。

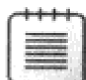
- 不是所有数据都能轻松匹配具有为每一行设置的小型固定列的关系数据库模型。通常，这些字段只是偶尔使用，从而使该列上有很多 NULL 项。一个传统的索引存储很多 NULL 并且浪费大量的存储空间。对表的更新必须维护每一行的这一索引。
- 如果您正在查询一个具有很少不同值的表并且正在使用一个某些元素固定的多列谓词，则可以创建一个筛选索引来加速这一特殊查询。这对于一种常规的只向老板进行汇报用的运行来说非

常有用——可以在加快小型查询速度的同时不明显降低其他人的更新速度。

■ 正如在原始示例中看到的那样，索引可以在对某个大型表进行一次开销很大的查询上有一个已知查询条件时使用。

8.6.2 索引视图

传统的非索引视图已经被用于简化 SQL 查询、从用户模中抽取数据模型及强制用户安全性等。从优化方面来说，SQL Server 不会对这些视图执行很多操作，因为它们在优化开始之前是扩展的或行内的。这使得查询优化器可以在全局优化查询，但是同时也使查询优化器更难考虑首先执行视图评估然后处理查询剩余部分的计划。任意树匹配是一个复杂的计算问题，并且视图的功能集太多以至于不能有效地执行这一操作。



注意：

只有 SQL Server 2008 企业版支持索引视图的匹配。

索引视图功能允许 SQL Server 公开视图物化的一些优点，同时保留全局分析查询操作的优点。SQL Server 在创建查询结果的一个物化视图上显示一个 *CREATE INDEX* 命令。得到的结构在物理上与有聚集索引的表一致。这种结构也支持非聚集索引。查询优化器可以使用这种结构更有效地将结果返回给用户。查询优化器包含在原始查询文本显式引用视图及用户提交一个与该视图使用相同组件（以任意相等顺序）的查询时使用这一索引的逻辑。实际上查询处理器原来在查询管道中扩展了索引视图并对两种情况始终使用相同的匹配代码。WITH(NOEXPAND)提示告诉查询处理器不要扩展视图定义。清单 8-10 包含一个利用 3 种不同路径获取 SQL Server 来匹配该视图。匹配计划如图 8-38、图 8-39 和图 8-40 所示。

清单 8-10 索引视图匹配示例

```
-- Create two tables for use in our indexed view
CREATE TABLE table1(id INT PRIMARY KEY, submitdate DATETIME, comment NVARCHAR(200));
CREATE TABLE table2(id INT PRIMARY KEY IDENTITY, commentid INT, product NVARCHAR(200));
GO
-- submit some data into each table
INSERT INTO table1(id, submitdate, comment) VALUES (1, '2008-08-21', 'Conor Loves Indexed
Views');
INSERT INTO table2(commentid, product) VALUES (1, 'SQL Server 2008');
GO
-- create a view over the two tables
CREATE VIEW dbo.v1 WITH SCHEMABINDING AS
SELECT t1.id, t1.submitdate, t1.comment, t2.product FROM dbo.table1 t1 INNER JOIN dbo.table2
t2 ON t1.id=t2.commentid;
go
-- indexed the view
CREATE UNIQUE CLUSTERED INDEX i1 ON v1(id);
-- query the view directly --> matches
SELECT * FROM dbo.v1;

-- query the statement used in the view definition --> matches as well
SELECT t1.id, t1.submitdate, t1.comment, t2.product
FROM dbo.table1 t1 INNER JOIN dbo.table2 t2
    ON t1.id=t2.commentid;
```

```
-- query a logically equivalent statement used in the view definition that
-- is written differently --> matches as well
SELECT t1.id, t1.submitdate, t1.comment, t2.product
FROM dbo.table2 t2 INNER JOIN dbo.table1 t1 ON t2.commentid=t1.id;
```

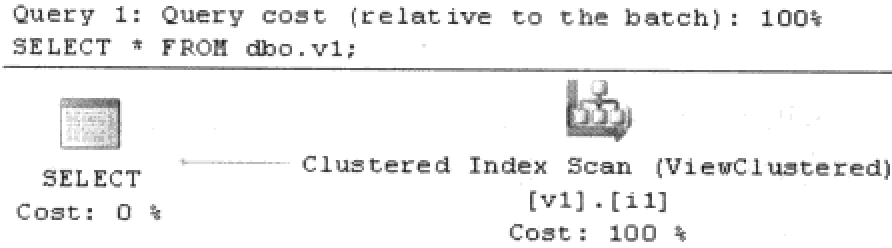


图 8-38　索引视图的一个直接引用匹配

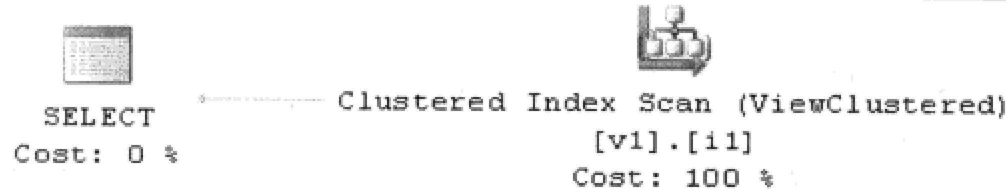


图 8-39　查询与视图定义匹配的一个索引视图匹配

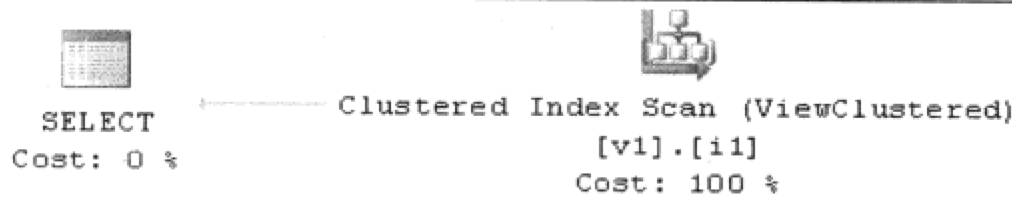


图 8-40　当查询与视图定义不完全匹配时的一种索引视图匹配

这是查询优化器不匹配该视图的情况。首先，请记住索引视图被插入到备注中并且按照其他计算选项进行评估。虽然它们通常是最好的计划选项，但是却不一定总是如此。在清单 8-11 中，查询优化器可以检测视图定义和引用该视图的查询之间的逻辑冲突。图 8-41 显示了直接引用基表（而不是视图）的查询计划。

清单 8-11　索引视图不匹配时的示例

```
CREATE TABLE table3(col1 INT PRIMARY KEY IDENTITY, col2 INT);
INSERT INTO table3(col2) VALUES (10);
INSERT INTO table3(col2) VALUES (20);
INSERT INTO table3(col2) VALUES (30);
GO
-- create a view that returns values of col2 > 20
CREATE VIEW dbo.v2 WITH SCHEMABINDING AS
SELECT t3.col1, t3.col2 FROM dbo.table3 t3 WHERE t3.col2 > 20;
GO
-- materialize the view
CREATE UNIQUE CLUSTERED INDEX i1 ON v2(col1);
GO

-- now query the view and filter the results to have col2 values equal to 10.
```

```
-- The optimizer can detect this is a contradition and avoid matching the indexed view
-- (the trivial plan feature can "block" this optimization)
SELECT * FROM dbo.v2 WHERE col2 = CONVERT(INT, 10);
```

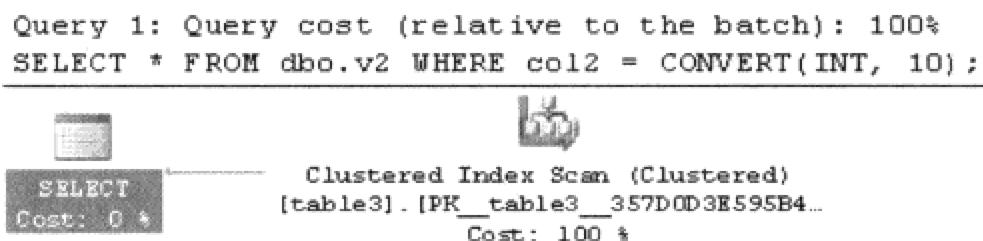
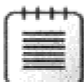


图 8-41 索引视图不匹配时的一个查询计划



注意：

这一示例中的谓词是*[v1].[dbo].[table3].[col2] as [t3].[col2]=[@1] AND [v1].[dbo].[table3].[col2] as [t3].[col2]>(20)*。我们已经在本章创建了尽可能简单的示例，查询优化器在这里使用逻辑来检测我们将这一查询示例设计得太简单。因此，它将该查询作为一个琐碎计划来对待并且对其进行自动参数化，从而使所有将来类似这个按常量（10）变化的查询都可以使用。虽然琐碎计划的复杂性没有正式记录并且在每个版本中都可能有所修改，但图 8-42 显示了稍微修改该查询以避免琐碎计划时可能发生的情况（我们使用了一个查询提示，但是代码中没有显示）。

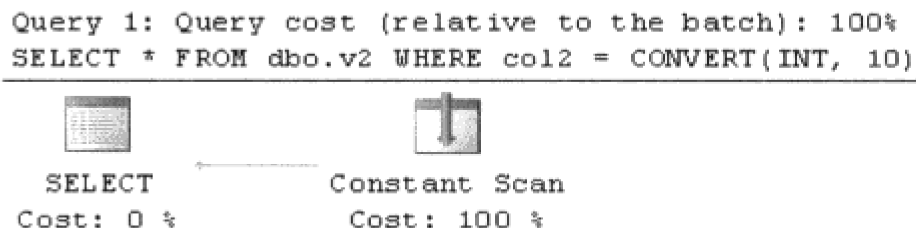


图 8-42 非琐碎计划冲突检测所引起的一个常量扫描计划

这是一个零行扫描，因为查询优化器发现 *col3=10* 和 *col2>20* 没有返回任何行。这一查询计划甚至不会试图扫描 *table3* 或 *v2*。

提示：

遗憾的是，有些情况下查询优化器不会识别某个索引视图，即使该视图可能是一个很好的计划选项。通常这些情况处理查询处理器内高级功能之间的复杂交互（如计算列匹配及探索联接顺序的算法）。虽然 SQL Server 确实通过警告和显示计划提供一些信息可以帮助您查看该级别上系统的性能，但是需要很多内部知识才能完全理解。如果碰巧发现自己遇到认为索引视图应该匹配但是没有匹配的情况，则考虑使用 WITH(NOEXPAND)提示来强制查询处理器选择该索引视图。这对于获取该计划以包括索引视图来说通常已经足够了。

SQL Server 在超出查询文本与视图定义的精确匹配之外的情况下同时支持匹配索引视图，也支持对不精确的匹配（视图的定义比用户提交的查询更宽泛）使用一个索引视图。SQL Server 接下来应用剩余的筛选器、投影（查询列表中的列）甚至聚合来使用该视图作为查询结果的一个局部预计算。

清单 8-12 显示了与筛选条件和投影剩余部分相匹配的视图。创建了一个比我们最终的查询有更多行

和更多列的视图，但是索引视图仍然由查询优化器进行匹配。结果查询计划如图 8-43 所示。

清单 8-12 索引视图匹配示例（行和列的一个子集）

```
-- base table
CREATE TABLE basetbl1 (col1 INT, col2 INT, col3 BINARY(4000));
CREATE UNIQUE CLUSTERED INDEX i1 ON basetbl1(col1);
GO
-- populate base table
SET NOCOUNT ON;
DECLARE @i INT =0;
WHILE @i < 50000
BEGIN
INSERT INTO basetbl1(col1, col2) VALUES (@i, 50000-@i);
SET @i+=1;
END;
GO
-- create a view over the 2 integer columns
CREATE VIEW dbo.v2 WITH SCHEMABINDING AS
SELECT col1, col2 FROM dbo.basetbl1;
GO
-- index that on col2 (base table is only indexed on col1)
CREATE UNIQUE CLUSTERED INDEX iv1 on dbo.v2(col2);

-- the indexed view still matches for both a restricted
-- column set and a restricted row set
SELECT col1 FROM dbo.basetbl1 WHERE col2 > 2500;
```

图 8-43 一个索引视图与行和列的一个子集相匹配

投影没有明确地作为查询中的一个单独计算标量运算符列出，因为 SQL Server 2008 有一种特殊的逻辑用于删除没有计算表达式的投影。索引匹配代码中的筛选运算符被翻译成一个针对该视图的索引查找。如果我们修改该查询来计算一个表达式，则图 8-44 显示了添加到计划中的剩余计算标量：

```
SELECT col1 + 1 FROM dbo.basetbl1 WHERE col2 > 2500 AND col1 > 10;
```

图 8-44 计算标量，只有在计算新值时才需要

与查询优化器考虑的所有选项类似，索引视图选项是在备注中生成和存储的，并且利用开销等式与其他可能出现的计划进行比较。包括局部匹配的替代项也估计剩余操作的成本，这表示在查询优化器认为其他计划具有更低开销时可能会生成一个索引视图计划但是不会选择该计划。

索引视图作为视图所基于的表的更新过程一部分进行维护。这样可以保证视图被查询优化器选择查

询计划时提供一致的结果。某些查询操作与这种设计保障不兼容。因此，SQL Server 在索引视图中的支持结构上进行了一些限制，以保证尽可能高效地创建、匹配和更新视图。*SQL Server 联机丛书*中的限制描述非常长并且非常详细，这使得理解高级规则非常困难。

对于更新索引视图来说，限制的核心问题是“查询处理器能够在不重新计算整个索引视图的情况下计算索引视图聚集和非聚集索引的必要修改吗”。如果可以，则查询处理器可以有效地执行这些修改作为视图中应用的基表维护的一部分。该属性对于筛选条件、投影（计算标量）和键上的内部联接来说相对比较容易。删除或创建数据的运算符更难维护，因此这些通常不能在索引视图中使用。

索引视图在更新计划中如何表示的问题将在本章后面的“更新”一节进行介绍。

8.7 分区表

随着 SQL Server 用于存储越来越多的数据，管理非常大型的数据库就成为 DBA 越来越关注的问题。首先，执行像重建索引这样操作的时间会随着数据的增加而增长，最终这可能会影响系统的可用性。其次，大型表的大小使得执行操作非常困难，因为系统通常受到临时空间、日志空间和物理内存等资源的限制。表和索引分区可以帮助您更好地管理大型数据库，同时将故障时间降到最低。

从物理上来说，分区表和索引实际上是存储部分行的 *N* 个表或索引。当与非分区情况相比较时，计划中的差别通常是分区情况下需要反复使一系列表或索引返回所有行。在 SQL Server 2005 中，用一个 APPLY 运算符表示，这基本上是一个嵌套循环联接。在 2005 的表示法中，一个特殊的分区 ID 表作为参数被传递给某个联接中的查询执行组件从而反复对每个分区进行操作。虽然这在大部分情况下都能够很好地工作，但是在一些非常重要的情况下不能很好地使用这一模式。例如，对要求并行表或索引扫描功能（多个线程从表中立即读取行以提高性能）在嵌套循环联接的内部不工作的并行查询计划有一种限制，这在 SQL Server 2005 之前是不可能实现的。遗憾的是，这是表分区的大部分情况。此外，APPLY 表示法启用了联接组合，此时以相同方式分区的两个表可以非常有效地联接。遗憾的是，事实证明这比设计该功能时所预想的情况更少见。由于类似的原因，在 2008 版本中进一步对表示法进行了改进。

SQL Server 2008 在大部分情况下通过在访问分区表或索引的运算符内存储分区的方式来表示分区。这种方式提供了很多优点，如使得并行扫描能够适当地工作。还删除了查询优化器中分区和非分区情况之间的很多其他差异，这些差异表明它们本身是未达到的性能优化。希望这可以使得在非分区的应用程序中部署分区更加容易。

清单 8-13 包含了显示这种新型设计的示例。图 8-45 包含了针对 SQL Server 2008 中分区表的结果查询计划。

清单 8-13 SQL Server 2008 分区示例——不需要应用

```
CREATE PARTITION FUNCTION pf2008(date) AS RANGE RIGHT
       FOR VALUES ('2008-10-01', '2008-11-01', '2008-12-01');
CREATE PARTITION SCHEME ps2008 AS PARTITION pf2008 ALL TO ([PRIMARY]);

CREATE TABLE ptnsales(saledate DATE, salesperson INT, amount MONEY) ON ps2008(saledate);
INSERT INTO ptnsales (saledate, salesperson, amount) VALUES ('2008-10-20', 1, 250.00);
INSERT INTO ptnsales (saledate, salesperson, amount) VALUES ('2008-11-05', 2, 129.00);
INSERT INTO ptnsales (saledate, salesperson, amount) VALUES ('2008-12-23', 2, 98.00);
INSERT INTO ptnsales (saledate, salesperson, amount) VALUES ('2008-10-3', 1, 450.00);

SELECT * FROM ptnsales WHERE (saledate) NOT BETWEEN '2008-11-01' AND '2008-11-30';
```

图 8-45　新 SQL Server 2008 分区模型的查询计划

您可以看到基本情况不需要与一种常量扫描进行额外的联接，这使查询计划更多情况下好像是非分区的，而且应该可以使查询计划更容易理解。

这种模型的一个优点是现在可以对分区表进行并行扫描。下面的示例创建一个大型分区表，然后执行一项生成并行扫描的 *COUNT(*)*操作。在 SQL Server 中，一些聚合函数可以被分成两个部分，其中一部分与表处于同一执行线程。这样可以加快大型查询的执行时间并降低需要在线程之间传递的行数。清单 8-14 显示了 SQL Server 2008 现在是如何对分区表进行并行扫描从而计算聚合的。图 8-46 包含了结果查询计划。

清单 8-14　分区并行扫描示例——SQL Server 2008

```
CREATE PARTITION FUNCTION pfparallel(INT) AS RANGE RIGHT FOR VALUES (100, 200, 300);
CREATE PARTITION SCHEME psparallel AS PARTITION pfparallel ALL TO ([PRIMARY]);
GO
CREATE TABLE testscan(randomnum INT, value INT, data BINARY(3000)) ON
psparallel(randomnum);
GO
SET NOCOUNT ON;
BEGIN TRANSACTION;
DECLARE @i INT=0;
WHILE @i < 100000
BEGIN
INSERT INTO testscan(randomnum, value) VALUES (rand()*400, @i);
SET @i+=1;
END;
COMMIT TRANSACTION;
GO
-- now let's demonstrate a parallel scan over a partitioned table in SQL Server 2008
SELECT COUNT(*) FROM testscan;
```

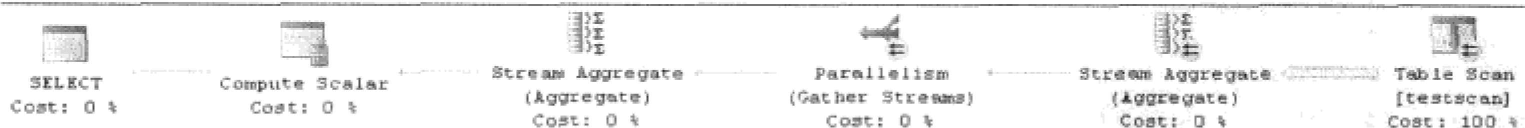


图 8-46　对 SQL Server 2008 中分区表的并行扫描

SQL Server 2005 对于如何能够对分区表进行并行查询进行了一些限制。使用 APPLY 运算符来扫描每个分区与系统上其他一些允许 SQL Server 2005 在每个分区上只能运行一个线程的限制交互性很差。虽然这样可以允许查询在扫描很多分区时能够并行运行，但是这种模式在查询访问单个分区时不能很好地运行。在访问单个分区时，只有一个线程可以访问该分区，基本上忽略了并行扫描的功能。遗憾的是，SQL Server 排列分区的一个主要原因是要在一个日期范围内访问最通用的分区。此外，APPLY 模式使得有效地处理分区扭曲（此时一个分区比其他分区大很多）很困难。虽然 SQL Server 2005 在使用这种模式

对查询进行成本估计时会考虑最大分区的大小，但是仍然有一个线程在其他线程之后完成。

查询优化器已经提高了分区表计划生成过程中端到端的体验。以相同方式表示分区表访问和非分区表访问的能力保证分区表和非分区表之间的性能差异被降到最低。尤其是被考虑的并行计划选项更加一致。查询执行组件可以在一个分区使用一个线程及一个分区使用多个线程之间进行动态地调整，这应该分配线程来更有效地完成查询处理。

在 SQL Server 2008 中，联接排列仍然用应用/嵌套循环联接来表示，但是其他情况使用传统表示法。这与查询处理器中的其他功能一起保证它们的行为方式与非分区表完全一样。下面的示例建立在最后一个示例基础上，用于说明联接具有相同分区机制的两个表可以通过排列联接技术完成。这种情况在 SQL Server 2005 中（当您希望将两个分区表或索引联接到一起时）是一样的。通常，这将是一个事实数据表和一个大规模的表索引（与实际表的分区方式相同）。图 8-47 显示了原始 SQL Server 2005 分区逻辑可见时的一个每分区联接。

```
-- SQL Server 2008 join collocation still uses the constant scan + apply model
SELECT * FROM testscan t1 INNER JOIN testscan t2 ON t1.randomnum=t2.randomnum;
```

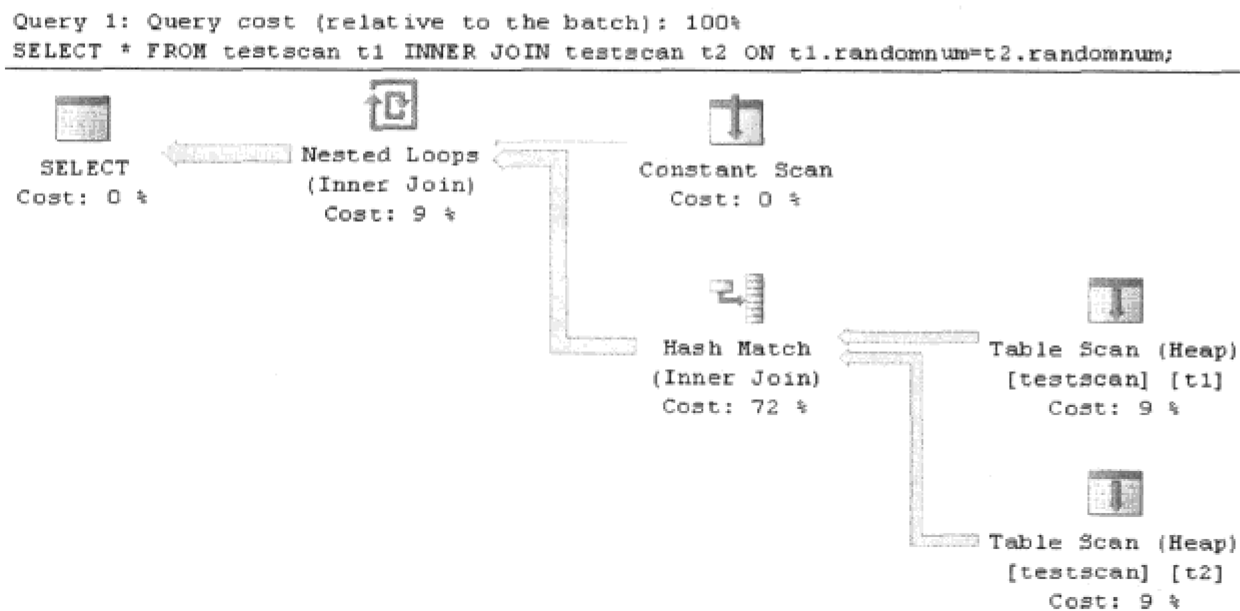


图 8-47 对分区表每分区联接的查询计划

SQL Server 2008 中分区表的实现方面有一点值得注意，因为这一点最初可能使您感到很惊讶。如果仔细查看图 8-48 中最后一个查询计划的显示计划输出，可能会注意到该分区表的堆扫描有一个查找谓词。

⊞ Output List	[s1].[dbo].[testscan].randomnum, [s1].[dbo].[testscan].value, [s1].[dbo].[testscan].data
Parallel	False
Partitioned	True
Physical Operation	Table Scan
Scan Direction	FORWARD
⊞ Seek Predicates	Seek Keys[1]: Prefix: PtnId1001 = Scalar Operator([Expr1008])
TableCardinality	100000

图 8-48 分区堆的查找谓词

虽然 SQL Server 2005 公开了查询计划中的分区 ID，但是 SQL Server 2008 在很大程度上将分区 ID 从视图中隐藏起来。分区 ID 仍然存在于查询计划中，但是在大部分情况下都是与索引更紧密地结合在一起。SQL Server 2008 中的每个分区访问结构都被建模成一个索引，其中第一列是分区列。由于分区 ID（继

承自分区键）需要执行查找，因此实际上与SQL Server 2005中看到的有效行为相匹配。唯一奇怪的地方在于分区堆现在显示为有一个索引。您可以从前面的示例属性中看到这一点。

8.7.1 分区对齐索引视图

SQL Server 2008允许跨*SWITCH*操作的分区对其索引视图存在。在SQL Server 2005中，必须删除这些视图后才能执行*SWITCH*，这一点限制了在索引视图被禁用和重建时保持一个系统作为生产系统运行的能力。现在，分区表（尤其是大型数据仓库）具有在保证数据库完全可用的同时对数据库进行维护的一种方式。

8.8 数据仓库

SQL Server包含很多加快数据仓库查询执行速度的特殊优化。一个数据仓库通常是具有一个大型事实数据表和包含被事实数据表引用的详细信息的很多小型维度表的一个大型数据库。这通常被称为*星型架构*或*雪花架构*（*雪花*应用到引用其他维度表的维度表）。这些类型的架构通常用于存储大量的原始数据，这些数据接下来用于帮助查找信息以帮助公司学习业务方面的知识。

数据仓库通常试图在事实数据表中使每一行尽可能小，因为事实数据表非常大。大数据（如字符串）被移动到维度表中以降低行内空间。事实数据表通常很大，以至于由于存储这些结构需要大量存储而使非聚集索引的使用受到限制。维度表经常被索引。这种模式与一般的事务处理系统不匹配，在事务处理系统中，每个表都是根据对系统的查询而被访问的。

对数据仓库进行优化查询时，不对事实数据表进行不必要的扫描非常重要，因为扫描通常是占用执行时间最多的操作。SQL Server可以识别星型和雪花架构并应用特殊的优化来提高查询性能。首先，SQL Server在数据仓库中按照不同的方式对联接进行排序，从而在对事实数据表执行扫描之前对维度表执行尽可能多的限制操作。这甚至可以包括在维度表之间执行完全交叉乘积，从而使事实数据表的扫描可以被消除。

SQL Server 2008还包含对位图运算符的改进，帮助降低在并行查询的线程之间对数据的移动。位图可用于将每一行的大小降低到一位。因为两个位图可以有效地相交或联合，因此这种模式允许SQL Server简单地通过执行一次位图操作来联接两个表。这样将允许每个维度表被查询以标识满足条件的行，创建接下来被发送到线程中扫描事实数据表的一个位图。这些位图通过使用一种探索筛选（作为事实数据表的非可求值谓词）而被应用。这有点像一种特殊的快速索引（专门为这种模式的数据仓库查询创建）。

SQL Server中影响大型表（如一个数据仓库配置中的真实表）的一个限制是在柱状图中只能有200个等级。非常大型的表通常有超过200个等级的有用数据分布信息，因此有时查询可能受这一限制的影响（即使有完全扫描统计信息）而被低估或高估，幸运的是，筛选统计信息可在一定程度上减轻这一问题——可以为该范围创建筛选统计信息定义的一个分区并使用该分区估计基数。由于分区表上的很多查询都是在一定日期范围内当前分区上进行的，因此这就包含了很多在SQL Server 2005中没有包含的情况。

8.9 更新

更新是查询处理中的一个重要方面。除在优化传统*SELECT*查询时遇到的很多挑战之外，更新优化还考虑物理优化，如每行需要接触多少索引、某一个时刻只处理一个索引上的更新还是同时处理所有索

引上的更新，以及如何尽可能快地处理修改同时避免不必要的死锁等。优化器包含很多专门针对更新的功能，用于尽可能快地完成查询。在这一部分，我们将讨论其中的很多优化。

在这一部分，术语*更新处理*实际上包括更改数据的所有顶级命令。其中包括 *INSERT*、*UPDATE*、*DELETE* 和（SQL Server 2008 中的）*MERGE*。正如您在这一部分中看到的那样，SQL Server 几乎同等对待这些命令。SQL Server 中的每个更新查询都由同样的基本操作组成：

- 确定哪些行被更改（插入、更新、删除、合并）；
- 计算所有更改列的新值；
- 将更改应用到表和所有非聚集索引结构。

图 8-49 说明了 *INSERT* 是如何使用这种模式工作的。

```
CREATE TABLE update1 (col1 INT PRIMARY KEY IDENTITY, col2 INT, col3 INT);
INSERT INTO update1 (col2, col3) VALUES (2, 3);
```

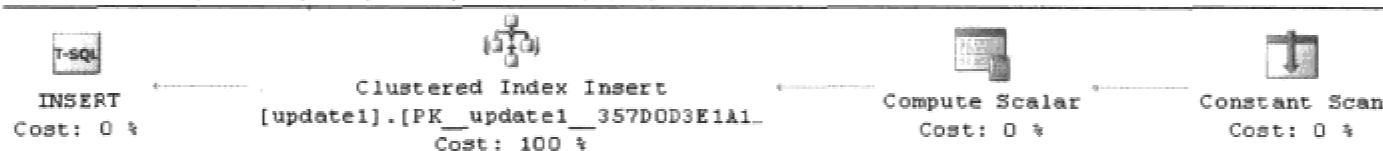


图 8-49 基本 *INSERT* 查询计划

INSERT 查询有一个被称为 *Constant Scan* 的运算符。Constant Scan 是关系代数中的一种特殊运算符，无需从表中读取行就能生成行。如果正在向某个表插入一行，而实际上不存在一个现有表，那么该运算符会为插入运算符创建一行进行处理。*计算标量*操作会估计将被插入的值。在我的示例中，这些是常量，但是它们可以是任意标量表达式或标量子查询。最后，插入运算符会在物理上更新主键聚集索引。

图 8-50 显示了 *UPDATE* 计划的表示方法。

图 8-50 *UPDATE* 查询计划

UPDATE 查询从聚集索引中读取值，执行 *Top* 操作，接下来更新同一个聚集索引。*Top* 操作实际上是处理 *ROWCOUNT* 的一个占位符，除非您在会话中执行了一次 SET *ROWCOUNT N* 操作，否则 *Top* 操作不执行任何操作。同时需要注意的是，在这个示例中，*UPDATE* 命令没有修改聚集索引的键，因此索引中的行不需要在索引内部移动。最后，好像没有一个运算符为 *col2* 计算新值 5。这显然不对——实际上 *col2* 是被处理了，但是一种物理优化将该命令融合成处理过程中的更新运算符。如果检查更新运算符的属性（如图 8-50 所示），会发现查询也已经被自动参数化并且目标值直接被应用到 *Update* 运算中。图 8-51 显示了 *DELETE* 查询计划模式。

```
DELETE FROM update1 WHERE col3 = 10;
```

DELETE 查询与 *UPDATE* 查询非常类似——唯一区别在于行最后被删除。唯一重要的区别在于 WHERE 子句用做源表查找操作的一个条件。

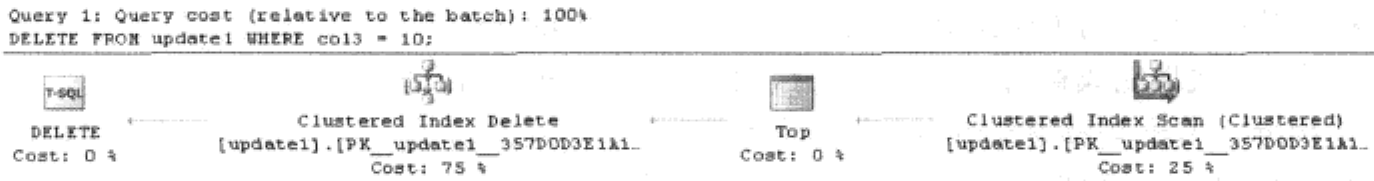


图 8-51 *DELETE* 查询计划

SQL Server 根据表、索引和其他辅助结构的物理布局生成不同的计划。例如，如果我们考虑一个没有主键聚集索引的类似示例，则会发现结果计划状态变为图 8-52 所示的效果。

```
CREATE TABLE update2 (col1 INT , col2 INT, col3 INT);
INSERT INTO update2 (col2, col3) VALUES (2, 3);
```

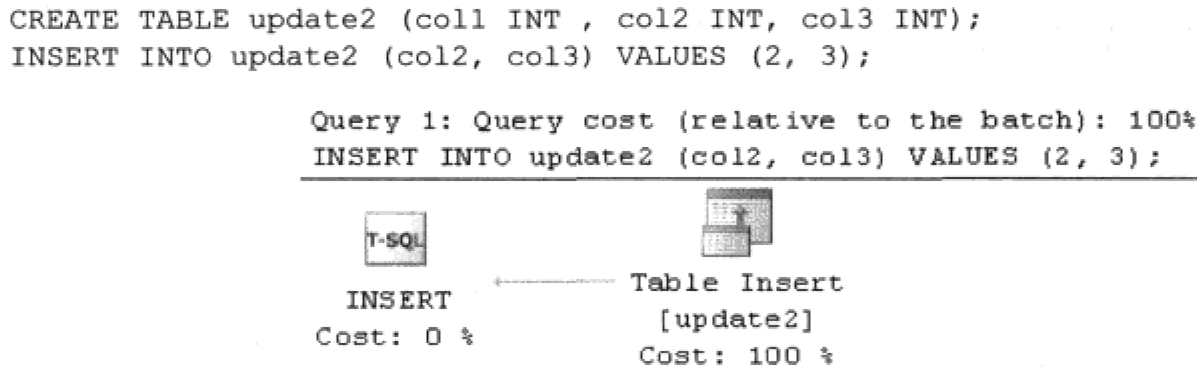


图 8-52 简单的 *INSERT* 查询计划

当表是一个堆（没有聚集索引）时，会出现一种特殊的优化将操作折叠成一种更小的形式。这被称为*简单更新*（*更新*这一术语通常指插入、更新、删除和合并计划），同时速度明显更快。这是完成向堆进行插入的所有工作的一个单一运算符，但不是对更新中的每一种功能都提供支持。

图 8-53 说明了如何在有多个索引的表中执行插入操作。

```
CREATE TABLE update3 (col1 INT , col2 INT, col3 INT);
CREATE INDEX i1 ON update3(col1);
CREATE INDEX i2 ON update3(col2);
CREATE INDEX i3 ON update3(col3);
INSERT INTO update3(col1, col2, col3) VALUES (1, 2, 3);
```

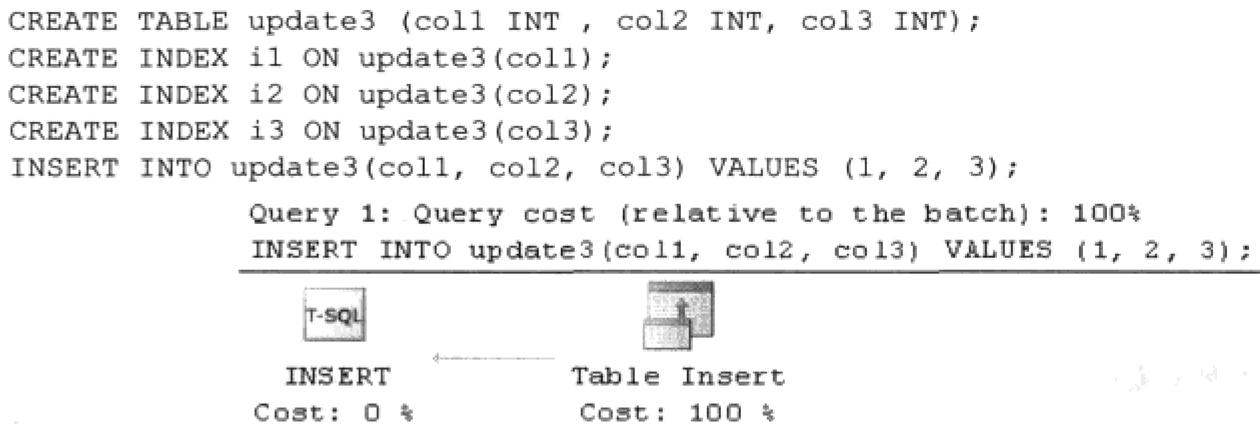


图 8-53 一体化 *INSERT* 查询计划

该查询需要更新所有索引，因为已经创建了一个新行。但是，图 8-53 说明了该计划只有这样一个运算符。如果在 Management Studio 中查看该运算符的属性，如图 8-54 所示，则可能会发现实际上更新某个运算符中的所有索引。

⊟ Object	[s1].[dbo].[update3], [s1].[dbo].[update3].[i1]
⊞ [1]	[s1].[dbo].[update3]
⊞ [2]	[s1].[dbo].[update3].[i1]
⊞ [3]	[s1].[dbo].[update3].[i2]
⊞ [4]	[s1].[dbo].[update3].[i3]

图 8-54 单个运算符更新的多个索引

有另一种物理优化用于提高常用更新情况的性能。这种插入被称为*一体化*或*每行*插入。

使用同一个表，我们试着使用 UPDATE 命令更新某些索引（而不是全部）。图 8-55 包含了结果查询计划。

```
UPDATE update3 SET col2=5, col3=5;
```

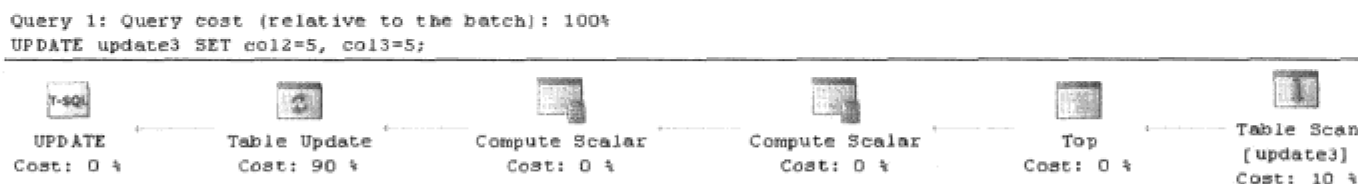


图 8-55 修改表上某些索引的一个查询计划

现在问题变得有点更复杂了。该查询扫描 Table Scan 运算符中的堆，执行 *ROWCOUNT Top*、两个 Compute Scalar 和一个 Table Update。如果检查表更新的属性，会发现只列出了索引 *i2* 和 *i3*，因为查询优化器可以通过统计信息确定该命令不会修改 *i1*。一个 Compute Scalar 计算列的新值，另一个 Compute Scalar 帮助计算每一行是否需要修改每个索引的另一种物理优化。SQL Server 包含处理非更新所更新的逻辑。在这种情况下，用户调用一次更新但实际上提交该行的现有值。查询优化器可以识别这种情况并在某个值被更新为相同值时避免某些内部步骤（如日志修改）。由于很多预定义的 SQL 应用程序和工具允许用户检索一行、修改某些列，然后将对所有列的完整更新写回数据库（而不只是修改的列），因此这实际上是加快查询速度的一种必要的有效方式。这种优化并不总被应用——SQL Server 使用逻辑对这种优化的可能性和有效性进行一种有依据的推测，但是不会在应用时降低写入通信量和逻辑通信量。

8.9.1 Halloween 保护

*Halloween 保护*描述了关系数据库中用于保证更新计划正确性的一种功能。之所以需要这种方案是由一个更新计划的一次单纯实现中所发生的情况决定的。执行一次更新的一种简单方法是使一个运算符遍历一棵 B+树索引并更新满足筛选条件的每一个值。这种方法的效果很好，只要人们给一个常量分配一个值或没有应用到筛选器中的某个值。如果不小心，则迭代器可能会发现已经在前面扫描过程中处理过的行，因为原来的更新将行移动到了指针的前面循环访问 B+树。

不是每个查询都需要担心这一问题，但是这却是查询计划在某些情况下会出现的一个问题。对这一问题的一般保护方法是将所有行扫描到一个缓冲中，接下来处理缓冲中的行。在 SQL Server 中，这通常是使用 Spool 或 Sort 运算符实现的，其中每种运算符都具有在查询树中产生输出行给下一个运算符之前读取所有输入行的某种保障。SQL Server 还能够使用某种特殊形式的计算标量运算符在某些限定情况下提供 Halloween 保护，但是显示计划没有公共信息说明这种情况正在发生（除计划中有一个额外的计算标量之外）。在所有情况下，副本防止同一行被查看两次。

8.9.2 拆分/排序/折叠

SQL Server 包含一种被称为*拆分/排序/折叠*的物理优化，用于使大范围更新计划更有效。该功能检查批处理中所有要被更改的更改行并确定这些更改对某个索引的净效应。这样避免了不必要的更改，从而降低完成查询的 I/O 需求。这种更改还允许进行一种单一线性传递，将更改应用到每个索引上，这比一系列随机 I/O 操作更有效。图 8-56 包含了结果查询计划。

```
CREATE TABLE update5(col1 INT PRIMARY KEY);
INSERT INTO update5(col1) VALUES (1), (2), (3);
```

```
UPDATE update5 SET col1=col1+1;
```

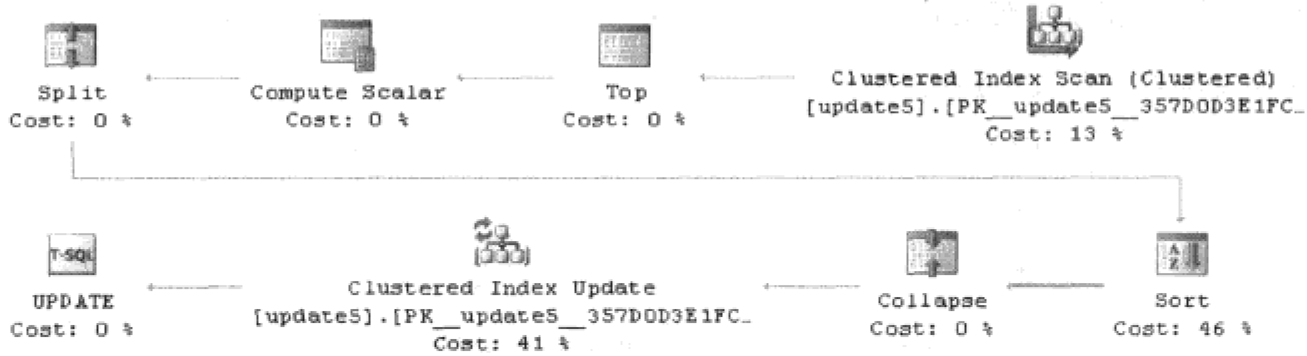


图8-56　拆分/排序/折叠 *UPDATE* 查询计划

这一查询正在修改一个聚集索引，该索引具有值为1、2和3的3行。在这一查询之后，这3行的值改为2、3和4。不是修改这3行，很可能是只删除1并插入4来对该查询进行修改。对于琐碎示例来说，可以避免修改一行，但是对于大型表来说，这种节省的规模可能是很大的。

这种优化通过一种被称为*操作列*的内部列实现。其中包含一个值表示每一行是否是一个 *INSERT*、*UPDATE*、*DELETE* 或 *MERGE*。更新运算符使用这种操作列确定应该对该索引应用怎样的修改。虽然显示计划根据提交查询的不同为该更新运算符显示不同的名称，但是在系统内部它们是同一个运算符并且通过操作列进行修改。遗憾的是，您不可以查看该列的值，因为该列只是查询处理器中的一种结构。

操作列也可以被查询处理器用于帮助确定应用到某个索引上的净更改。也可以被拆分/排序/折叠逻辑用于确定对该索引要进行的下一步更改。现在让我们大致看一下每一步骤中的情况。在拆分之前，行数据如表8-1所示。

表8-1　　预拆分更新数据表示

操　作	旧　值	新　值
UPDATE	1	2
UPDATE	2	3
UPDATE	3	4

拆分将每条 *UPDATE* 转换成一条 *DELETE* 和一条 *INSERT*。拆分之后，现在行的显示效果如表8-2所示。

表8-2　　拆分后的数据表示

操　作	值
DELETE	1
INSERT	2
DELETE	2
INSERT	3
DELETE	3
INSERT	4

按照（值，操作）进行排序，其中 *DELETE* 排在 *INSERT* 之前。排序之后，行的内容如表8-3所示。

Collapse 运算符在（*DELETE*，*INSERT*）对中查找相同的值并将它们删除。在这个示例中，Collapse 运算符将值为2和3的行用 *UPDATE* 替换了 *DELETE* 和 *INSERT* 行。*UPDATE* 减少了必要的B+树维护操作的数

量，同时存储引擎能够不记录B+树更新的相同值（但是，锁用于更正）。折叠后行的最终格式如表8-4所示。

表 8-3 排序后的数据表示

操　作	值
DELETE	1
DELETE	2
INSERT	2
DELETE	3
INSERT	3
INSERT	4

表 8-4 折叠后的数据表示

操　作	值
DELETE	1
UPDATE	2
UPDATE	3
DELETE	3
INSERT	4

结果是需要对索引进行净更改。从技术上来说，每个索引都包含主键索引或堆的行标识符，甚至索引中不存在的行也会被更新，从而使引用适应堆或聚集索引。日志流量也会在标准更新路径中减少，并且也可以获得I/O排序的优点。

虽然拆分/排序/折叠逻辑是一种性能优化，但是它也会帮助在修改一种唯一索引（例如主键）时避免失败故障。如果原始计划在没有拆分/排序/折叠的情况下被执行，则它会试图将值为1的行修改为2。这可能会与索引中值为2的现有行相冲突。虽然这对于该查询来说可以通过向后循环行的方式避免，但是有时不能通过选择一种扫描顺序的方式来避免这一问题。拆分/排序/折叠允许SQL Server在不返回错误的情况下支持该示例之类的查询。

8.9.3 合并

SQL Server 2008引入了一种被称为*MERGE*的新型更新操作。*MERGE*是其他更新操作的混合，用于对表执行条件更改。该操作的商业价值在于它可以将多个T-SQL操作折叠成一个查询。这样会简化必须编写修改表的代码、提高性能并在实际上帮助对大型表（可能太大以至于有效地进行多步操作太慢而失去意义）的操作。

看到其他更新操作如何被处理后，可能已经知道*MERGE*实际上不是对其他操作中使用的操作列技术的一种复杂扩展。与其他查询一样，元数据被扫描、筛选和修改。但是，在*MERGE*运算中，要被修改的行与目标资源相联接以决定对每一行所应执行的操作。根据这一联接，每一行的操作列将会被修改以通知*STREAM UPDATE*操作对每一行要进行的操作。

在清单8-15中，对一个现有表使用新数据进行更新，这些新数据中可能有一些已经在表中存在。因此，*MERGE*用于确定不存在的行。图8-57包含了结果*MERGE*的查询计划。

清单 8-15　一个 *MERGE* 示例

```
CREATE TABLE AnimalsInMyYard(sightingdate DATE, Animal NVARCHAR(200));
GO
```

```
INSERT INTO AnimalsInMyYard(sightingdate, Animal) VALUES ('2008-08-12', 'Deer');
INSERT INTO AnimalsInMyYard(sightingdate, Animal) VALUES ('2008-08-12', 'Hummingbird');
INSERT INTO AnimalsInMyYard(sightingdate, Animal) VALUES ('2008-08-13', 'Gecko');
GO
CREATE TABLE NewSightings(sightingdate DATE, Animal NVARCHAR(200));
GO
INSERT INTO NewSightings(sightingdate, Animal) VALUES ('2008-08-13', 'Gecko');
INSERT INTO NewSightings(sightingdate, Animal) VALUES ('2008-08-13', 'Robin');
INSERT INTO NewSightings(sightingdate, Animal) VALUES ('2008-08-13', 'Dog');
GO

-- insert values we have not yet seen - do nothing otherwise
MERGE AnimalsInMyYard A USING NewSightings N
    ON (A.sightingdate = N.sightingdate AND A.Animal = N.Animal)
WHEN NOT MATCHED
    THEN INSERT (sightingdate, Animal) VALUES (sightingdate, Animal);
```

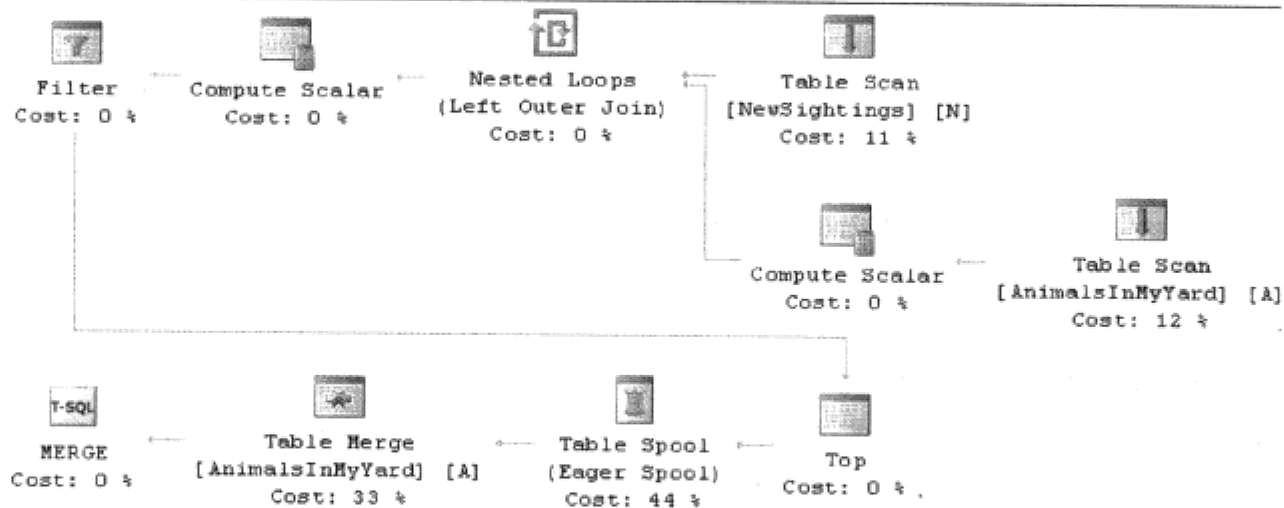


图 8-57 合并查询计划

由于 *MERGE* 计划会变得越来越大，因此我们将其分成几部分并介绍查询计划的每一部分。该计划的第一部分如图 8-58 所示。

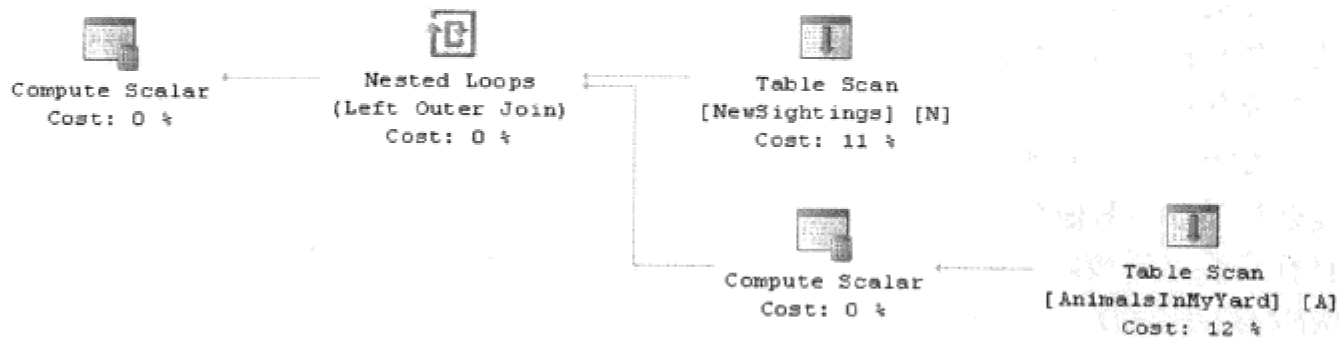


图 8-58 MERGE 计划——初始联接以查找已存在的行

首先读取源表 *NewSighting*，同时查询处理器探测目标表 *AnimalsInMyYard*，查看该行是否已经存在。Left Outer Join 隐含的 Computer Scalar 仅仅用于添加一列，如果该值匹配则值为 1，如果没有与源表行相匹配的行（由于 Left Outer Join 的工作原理）则返回 NULL 值。联接上的 Computer Scalar 会生成 *Action* 列：

```
[Action1008] = Scalar Operator(ForceOrder(CASE WHEN [TrgPrb1006] IS NOT NULL THEN NULL
ELSE(4) END))
```

在该计划的上半部分（如图 8-59 所示），筛选器删除有一次 null 操作的行（谓词：[Action1008] IS NOT NULL），因为该 *MERGE* 语句只有一项操作（可能在一条 *MERGE* 语句中有多项操作）。导出查询命令的脚本能够提供 Halloween 保护，这表示它会在视图将值写回 *AnimalsInMyYard* 表之前使用输入中的所有行。表 *MERGE* 实际上只是一项更新操作，但是显示计划输出已经被更改，以避免出现混淆。

图 8-59 *MERGE* 计划——更新、Halloween 保护假脱机和行筛选器

8.9.4 大范围更新计划

SQL Server 还有用于加快对表进行大型批处理更改的速度的特殊优化逻辑。如果表的大部分内容正在被某个查询更改，则 SQL Server 可以创建一种计划避免利用很多次单独更新来修改每棵 B+树，它生成一种预索引计划，确定需要被更改的所有行，将其按索引顺序排序并在一次传递过程中将更改应用到索引中。这种方法可能比单独更新每一行更有效。这些计划被称为*每索引*或*大范围更新*计划，您可以在它们的计划形式中看到。

下面的示例显示了一个大范围更新计划。图 8-60 包含结果计划。

```
CREATE TABLE dbo.update6(col1 INT PRIMARY KEY, col2 INT, col3 INT);
CREATE INDEX i1 ON update6(col2);
GO
CREATE VIEW v1 WITH SCHEMABINDING AS SELECT col1, col2 FROM dbo.update6;
GO
CREATE UNIQUE CLUSTERED INDEX i1 ON v1(col1);
UPDATE update6 SET col1=col1 + 1;
```

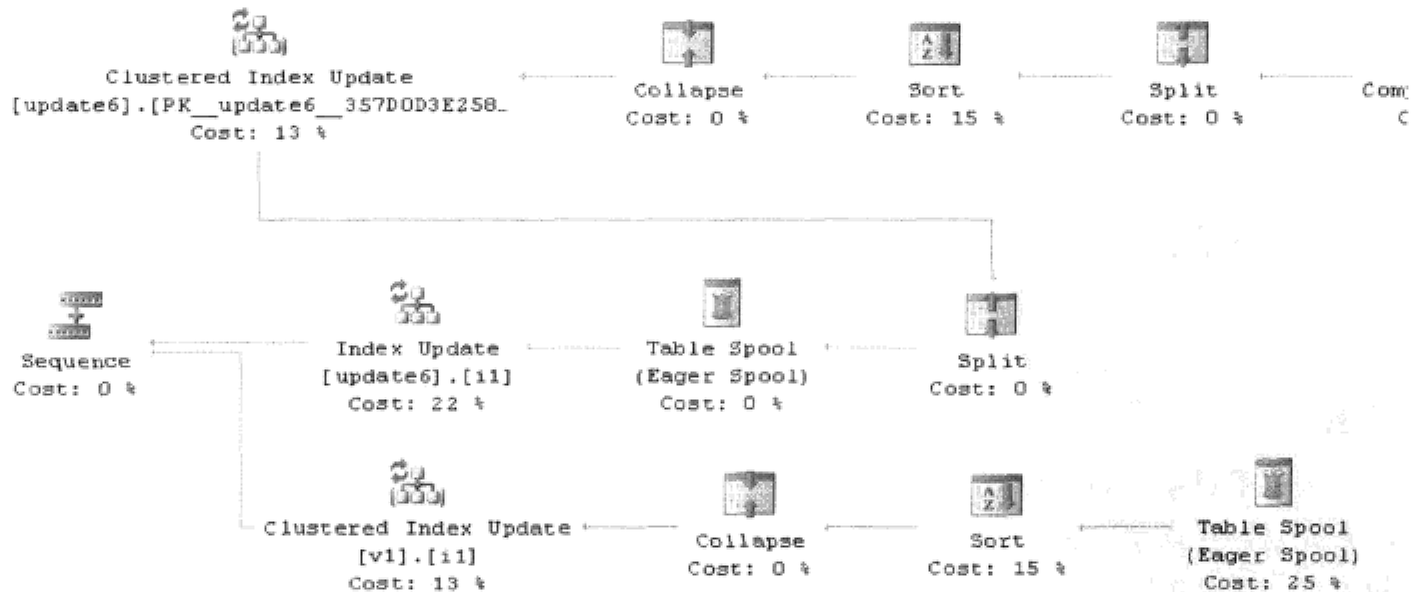


图 8-60 大范围更新查询计划（有删节）

由于该计划比较复杂，因此我们将其拆分成更小的部分，使其能够适合打印页面并且不是非常大。

图 8-61 包含查询计划的第一部分，工作方式与前面的示例类似：净更改只应用到聚集索引上。净更改的内容是所有非聚集索引的一个超集，因为聚集索引包括所有列。

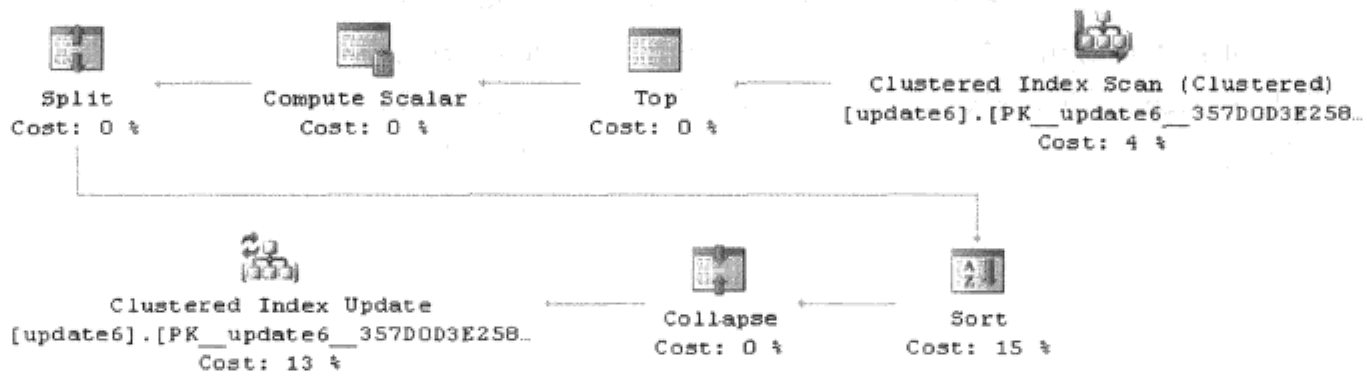


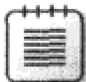
图 8-61 大范围更新计划的聚集索引更新部分

图 8-62 包含查询计划的下一部分。聚集索引更新上的第一个分支完成很多任务。该计划中的导出查询脚本是一种常见的子表达式导出查询脚本（本章前面介绍过）。这是广播行的一种方式，允许每个索引使用这一数据作为输入。Sequence 运算符不改变或修改数据，用于按输入顺序处理数据。这可以驱动大范围更新计划中行的处理进程。最后，由于可以有对这些行进行拆分/排序/折叠的多个客户端，因此 SQL Server 通过在第一个分支上执行操作进行优化，从而使拆分只执行一次而不是多次。

图 8-62 已更新行被拆分并存储在一个多次读取的假脱机中

最后，第二个分支读取前面已导出的查询脚本和拆分的行、为索引排序、折叠及对该索引执行净更改。图 8-63 包含计划的这一部分。如果该查询有其他索引要更新，则它们可能被作为查询中的其他分支被应用，这些分支将依次被处理。

图 8-63 大范围更新计划中的一个非聚集索引



注意：

大范围更新计划是 SQL Server 中最常见和最完整的函数形式的更新计划，并且从结构上来说，所有计划都可以作为 SQL Server 中的一个大范围计划来执行。某些功能（如被索引的视图和查询通知）仅使用大范围更新计划进行更新。由于 SQL Server 中可用的某些优化限定于更传统的功能集，因此请注意使用某些功能会强制 SQL Server 使用大范围更新计划。在大部分情况下，这对应用程序没有什么影响，但是对于有少量数据执行很多次更新的系统来说可能会受到影响。

非更新更新

UPDATE 操作有很多特殊优化，用于提高通常情况下的性能。例如，更新数据库中一行的常见编程范例如下。

（1）运行 *SELECT* 查询，从数据库中检索一行并将值复制到客户端或中间层，如：

```
SELECT col1, col2, col3, . . . FROM Table WHERE primarykey = <constant>
```

（2）允许用户选择性地更新某些列。

（3）客户端完成对行的修改后，试图用下面类似的查询将值写回服务器：

```
UPDATE Table SET col1=@p1, col2=@p2, col3=@p3 . . . WHERE primarykey = constant AND col1
= originalcol1value AND col2 = originalcol2value AND col3 = originalcol3value AND . . .
```

这种模式提供了一种功能但不是最佳的并发控制，无需服务器锁定基表。此外，数据库程序员通常只需实现一次 *UPDATE* 查询就能处理所有被修改列，并且只需向 SET 列表传递原始值，避免必须处理很多查询计划工作。

SQL Server 具有可以根据 *UPDATE* 的 SET 列表中的列来确定需要维护的索引更新逻辑。但是，默认情况下，这里介绍的模式会使 SQL Server 更新每个 *UPDATE* 语句的所有索引，即使只有一列值发生实际变化。为了避免这一问题，SQL Server 实现了一种被称为*非更新更新*功能，这一功能可以动态地检测未更改的值并避免对未更改的索引进行更新。虽然该查询计划仍然引用每个索引，但是可以避免写入不需要值的开销。

这种优化不是在所有情况下都执行，某些逻辑用于将这种优化应用到最有可能提高性能的场合。这种优化对于用户来说是透明的，不过在某些查询计划中可以看到这种优化被作为附加筛选器。

8.9.5 稀疏列更新

SQL Server 引入了一种被称为*稀疏列*的新功能，用于支持在表中创建比原来更多的列，同时也支持创建从技术上来说比数据页大小更大的行。该功能主要用在用户可以动态创建列的灵活架构系统中。通常这些列大部分是 NULL 值，但是有些行的值已经被定义。对于每个稀疏列来说，这种模式在很大程度上是独立的，也就是说某个给定的行有几个（但通常不会很多）非 NULL 稀疏列值。

稀疏列存储在标准数据行的一个复杂列中，这一点已经在第 7 章中介绍过。在使用稀疏列数据时，SQL Server 必须解释复杂列以确定哪些列中有实际的值。为了修改稀疏列，行会被读取、新值会被计算并且接下来行会被写入。主要的区别在于稀疏列需要更多的工作来读取和修改。

8.9.6 分区更新

更新分区表比更新非分区表更复杂。查询处理器必须处理每个分区中的堆或 B+树 [不是一个单一的物理表（堆）或 B+树]。需要弄清楚每一行属于哪个分区，同时行可以在某些更新计划的分区之间进行移动。此外，每个索引都可以使用一种单独的分区函数进行分区。甚至索引的视图也可以被分区，它们也可以用与表相关的其他访问路径进行分区。SQL Server 2008 中的分区更新计划是本章已经介绍过的更新计划形式的一种扩展。因此这里只介绍使用分区时这些计划有何不同。

在本章前面对分区表 *SELECT* 计划的介绍中，您可能会想到在每种访问方法（堆或索引）中分区 ID 在查询处理器中表示为一个虚拟引导列。分区表更新也使用这种表示方法，这种表示方法使得计划看起来很像更新索引使用的计划。引导列也显示在更新计划使用的某些其他运算符中，如拆分/排序/折叠运算

符。下面的示例说明分区是如何适应这些计划的。

在第一个示例中，我们创建了一个分区表并向其中插入了一行。这一计划的效果如图 8-64 所示。

```
CREATE PARTITION FUNCTION pfinsert(INT) AS RANGE RIGHT FOR VALUES (100, 200, 300);
CREATE PARTITION SCHEME psinsert AS PARTITION pfinsert ALL TO ([PRIMARY]);
go
CREATE TABLE testinsert(ptncol INT, col2 INT) ON psinsert(ptncol);
go
INSERT INTO testinsert(ptncol, col2) VALUES (1, 2);
```

Query 1: Query cost (relative to the batch): 100%
INSERT INTO testinsert(ptncol, col2) VALUES (1, 2);

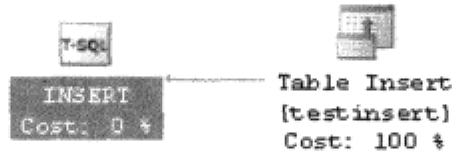


图 8-64　分区插入——单个分区

从查询计划中可以看到，该查询计划与非分区表的处理方式相同。只有一个运算符插入到表中。但是，实际上查询执行运算符必须确定哪个分区需要被更新、加载该分区并设置适当的值。查看 *INSERT* 运算符的属性会发现（如图 8-65 所示）分区特殊的逻辑。首先，*Expr1005* 被计算以确定要使用的目标分区，同时可以看到分区边界被传递给一个名为 *RangePartitionNew* 的内部函数。在 *Predicate* 部分，额外的 *Expr1005* 计算值用于设置 *Ptnld1001* 列，该列是在系统中显示的、用于支持分区的虚拟分区 ID 列。*Predicate* 列表中的剩余部分支持标准列 *ptncol* 和 *col2* 设置的值。

⊟ Misc	
⊞ Defined Values	[Expr1005] = Scalar Operator(RangePartitionNew([@1],(1),(100),(200),(300)))
Description	Insert input rows into the table specified in Argument field.
Estimated CPU Cost	0.000001
Estimated I/O Cost	0.01
Estimated Number of Executi	1
Estimated Number of Rows	1
Estimated Operator Cost	0.0100022 (100%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	9 B
Estimated Subtree Cost	0.0100022
Logical Operation	Insert
Node ID	0
⊞ Object	[s1].[dbo].[testinsert]
⊞ Output List	
Parallel	False
Partitioned	True
Physical Operation	Table Insert
Predicate	[s1].[dbo].[testinsert].[ptncol] = [@1],[s1].[dbo].[testinsert].[col2] = [@2],[PtnId1001] = RaiseIfNullInsert([Expr1005])

图 8-65　查询计划中的分区选择计算

查询处理器可以加载必要的分区来动态地修改每一行。如果在一条语句中插入多行，则可以看到图 8-66 中的查询处理器是如何正确支持每一行更新的。

```
INSERT INTO testinsert(ptncol, col2) VALUES (5, 10),(105,25);
```

该查询视图插入两行，并且查询计划使用 Constant Scan 运算符表示这一查询。Compute Scalar 运算符计算分区行数，决定每一行的目标分区，如图 8-67 所示。

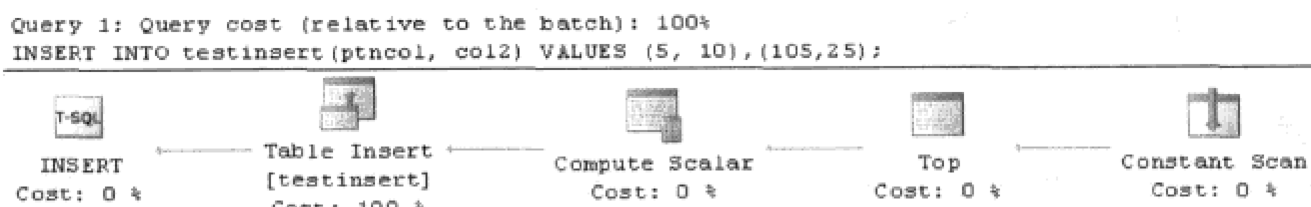


图 8-66 插入计划中的动态分区计算

Defined Values [Expr1007] = Scalar Operator(RangePartitionNew([Union1005],(1),(100),(200),(300)))

图 8-67 显示计划输出中的范围分区计算

而且，Table Insert 运算符使用这种计算标量（如图 8-68 所示），并且对每一行的适当分区进行更改（如果需要）。

图 8-68 插入利用一个范围分区来确定插入目标分区

> **注意：**
>
> 由于实现方面的原因，Compute Scalar 存在于这两行计划中并且不存在于第一个示例中。这些因素对于理解这些计划或者在运行时实际影响计划的性能方面来说是不必要的。

更改分区可能是开销比较大的操作，特别是在一条语句中修改表中的很多行时。拆分/排序/折叠逻辑也可以用于降低发生分区交换的数量，从而提高运行时性能。在下面的示例中，大量行被插入到了分区表中，同时查询优化器选择在插入之前在虚拟分区 ID 列上进行排序以降低运行时分区交换的数量。图 8-69 显示了查询计划中的排序优化。

```
INSERT INTO testinsert SELECT * FROM #nonptn;
```

图 8-69 排序优化以降低分区交换

排序有一种排列需求，如图 8-70 中所定义的那样，排序继承自对早期计划 Compute Scalar 中分区函数的调用，与前面的示例类似。

⊞ Order By Expr1009 Ascending

图 8-70 分区插入排序优化的排列需求

对分区表的更新更复杂，因为它们可能移动行。但是，它们与更新遵循相同的原则。要了解的关键属性是查询优化器必须读取每一行、确定对该行的更改、计算该行的目标分区并执行更改。其中可能包括从一棵 B+树中删除分区并将分区插入到另一棵 B+树中。这与批处理更新的拆分/排序/折叠概念非常接

近，但是对于分区更新来说，甚至可能发生在更小的更改上。

8.9.7 锁定

SQL Server 包含很多用于提高系统内整体性能和更新吞吐量的技巧和优化。虽然很多情况下查询优化器不知道锁定的存在，但是更新中的几种目标功能和锁定模式可以改进并发性（同时避免死锁错误）。有一种特殊的锁定模式，被称为 U（用于更新）锁。这是一种与其他 S（共享）锁兼容的特殊锁类型，但是与其他 U 锁不兼容。在更新查询中使用的很多计划中，SQL Server 都有两种不同的运算符访问同一访问的方法。第一种运算符是源表，只能读取。第二种运算符是更新本身。如果读取运算符中只有一个共享（S）锁，则多个用户可以同时运行查询，两者都为某一行获取 S 锁然后死锁，因为当更新运算符后来查看该行时，两者都不能将 U 锁更新为排他（X）锁。为了防止出现这一种现象，U 锁与其他 S 锁兼容但与其他 U 锁不兼容。这样可以防止其他人读取行，从而避免死锁。

清单 8-16 显示了如何检查某个更新查询计划的锁定行为。图 8-71 包含这一示例中使用的查询计划，图 8-72 包含 *sp_lock* 的锁定输出。

清单 8-16　更新锁定示例

```
CREATE TABLE lock(col1 INT, col2 INT);
CREATE INDEX i2 ON lock(col2);
INSERT INTO lock (col1, col2) VALUES (1, 2);
INSERT INTO lock (col1, col2) VALUES (10, 3);

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION;
UPDATE lock SET col1 = 5 WHERE col1 > 5;
EXEC sp_lock;
ROLLBACK;
```

图 8-71　锁定示例中使用的更新计划

	spid	dbid	ObjId	IndId	Type	Resource	Mode	Status
1	52	21	0	0	DB		S	GRANT
2	52	1	1131151075	0	TAB		IS	GRANT
3	52	21	693577509	0	PAG	1:75676	IX	GRANT
4	52	21	693577509	0	RID	1:75676:1	X	GRANT
5	52	21	693577509	0	TAB		IX	GRANT

图 8-72　更新计划的 sp_lock 输出

使用一种具有用户控制（非自动提交）事务的高级分离模式可以检查查询中每个对象的最终锁定状态。在本例中，可以看到行（*Resource 1:69641:1*）用一个 X 锁锁定。该锁开始是一个 U 锁，后来被 *UPDATE* 提升为一个 X 锁。

我们可以运行一个稍微不同的查询，根据选中查询计划的不同而显示锁的变化。图 8-73 包含一个基于查找的更新计划。在第二个示例中，U 锁被非聚集索引占用，而基表中包含 X 锁。因此，这个 U 锁保护只在检查相同的访问路径时才工作，因为它存在于查询计划的第一种访问路径中。图 8-74 显示了该查询的锁定行为。

```
BEGIN TRANSACTION;
UPDATE lock SET col1 = 5 WHERE col2 > 2;
EXEC sp_lock;
```

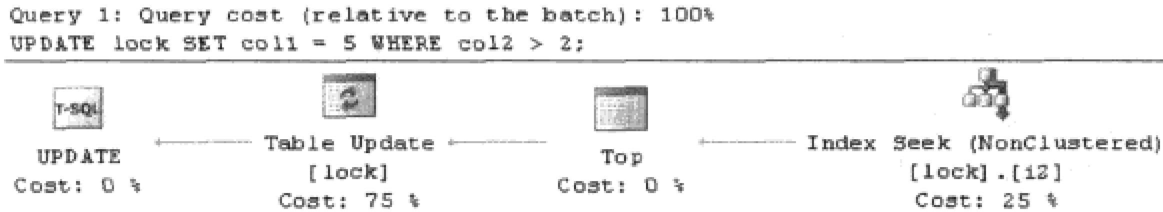


图 8-73 一个具有查找的更新计划的锁定行为

	spid	dbid	ObjId	IndId	Type	Resource	Mode	Status
1	52	21	0	0	DB		S	GRANT
2	52	21	693577509	2	KEY	(a0004dc87aeb)	U	GRANT
3	52	1	1131151075	0	TAB		IS	GRANT
4	52	21	693577509	2	PAG	1:75678	IU	GRANT
5	52	21	693577509	0	PAG	1:75676	IX	GRANT
6	52	21	693577509	0	RID	1:75676:1	X	GRANT
7	52	21	693577509	0	TAB		IX	GRANT

图 8-74 基于查找的更新计划的 sp_lock 输出

分区级锁升级

锁定行为通常不是查询优化器的控制范围。虽然查询优化器确实试图生成降低锁定冲突的计划，但是很大程度上不知道计划的锁定交互作用是好是坏。查询优化器使用很多逻辑来实现分区，包括从查询计划中删除不必要的分区从而不必访问这些分区。SQL Server 2008 产品中的一个很好的附加功能是*分区级锁升级*。该功能允许数据库避免对分区表的锁升级为表级。该功能与删除功能一起使用时，可以提供一种改进应用程序并发性的强大方法，尤其是对大型分区表进行查询可能会花费很长时间来执行的情况。该功能可以通过下面的命令启动：

```
ALTER TABLE TableName SET (LOCK_ESCALATION = AUTO);
```

锁定方面的内容将在第 10 章详细介绍。

8.10 分布式查询

SQL Server 包括一种称为*分布式查询*的功能，可以访问不同 SQL Server 实例上的数据、其他数据源和非数据库表格数据（如 Microsoft Office Excel 文件或逗号分隔文本文件）。分布式查询基于 OLE DB 接口，并且大部分 OLE DB 提供程序的功能非常丰富，可以被分布式查询功能所使用。由于可以在一个查询中引用多个源，因此这是一种与多源数据进行交互的有效机制，无需编写很多特例代码。

分布式查询支持多种不同的使用情况。首先，分布式查询对于从一个源向另一个源移动数据非常有用。虽然分布式查询不是 SQL Server Integration Services 那样的一种提取、转换和加载（ETL）工具，但是通常是将表从一台服务器复制到另一台服务器的一种非常简单的方式。例如，如果某家公司的财务报表组希望获得每个月销售情况的一个副本，则可能会编写一条查询从销售部门的 SQL Server 实例中复制数据到财务报表组的另一台服务器上。分布式查询的另一个应用程序是将非传统的资源集成到一个 SQL Server 查询中。由于有 OLE DB 提供程序用于活动目录域服务、Microsoft Exchange Server 和很多第三方资源的非数据库数据，因此可能编写查询从这些资源中收集信息，然后使用 SQL 语言询问可能不被该数据源支持的数据问题。分布式查询也可以用于报表。由于多个源可以在一条查询中被查询，因此您可以使用一条查询将数据收集到一个源中并生成报表（可以通过报表服务完成）。最后，分布式查询可以用于向外扩展的情况。SQL Server 支持一种被称为*分布式分区视图（DPV）*的特殊 *UNION ALL* 视图。该视图将存储在不同 SQL Server 实例中一个范围的不同部分聚合到一起。特别大的表可以存储在不同的服务器上并且可以引导查询只访问必要的子集以满足特殊的查询。分布式查询功能包括很多种情况并且可以帮助更轻松地解决这些问题。

分布式查询是在查询优化的计划搜索框架中实现的。除直接传递查询（不是在优化期间进行修改的）之外，分布式查询最初都是使用与标准查询相同的运算符表示的。查询优化器树中表示的每个基表都包含使用 OLE DB 元数据接口（如 OLE DB 模式行集）从远程资源中收集到的元数据，收集到的信息与查询处理器为本地表收集的信息非常类似，包括列信息、索引信息和统计信息。收集到的一条附加信息包括关于 SQL 语法构建远程资源支持的信息，这些信息将在后面的优化中使用。一旦收集到元数据，查询优化器就会为操纵远程数据的每个运算符提取特殊属性信息。该属性确定生成一条 SQL 语句标识可以直接发送到远程数据源的整棵查询子树是否可行。某些运算符可以轻松地被远程控制，如 Filter 和 Project。其他运算符只能在本地执行，如用于实现 SQL Server 中部分 XQuery 功能的数据流表值函数运算符。SQL Server 执行探索规则，将查询树转换成允许服务器远程控制更大型树的结构。例如，SQL Server 试图在一棵子树中将同源的所有远程表分组到一起并将聚合拆分成可以被远程控制的本地结构。在这一过程中，SQL Server 的某些更高级规则如果生成防止子树远程控制的选项就会被禁用。虽然 SQL Server 不维护每个远程资源的特殊开销模式，但是该功能用于将大型子树远程控制到该资源上，尽量避免在服务器之间移动数据。这通常提供一种准最佳查询计划。

在这一示例中，我们创建了链接服务器指向远程 SQL Server 实例。接下来，我们利用四部分命名语法生成一条可以完全被远程分配到远程资源的查询（如图 8-75 所示）。

```
EXEC sp_addlinkedserver 'remote', N'SQL Server';
go
SELECT * FROM remote.Northwind.dbo.customers WHERE ContactName = 'Marie Bertrand';
```

图 8-75 一条完整的远程分布式查询

正如您所看到的那样，这条相对简单的查询基本上完全由查询优化器远程控制。远程查询结点的属性信息包含在远程服务器上执行的查询文本。结果被返回到本地服务器并返回给用户。

这里显示的生成查询比原来提交的文本更冗长，但是为了保证远程查询的语义与本地查询文本精确

匹配，这是必需的。

```
SELECT "Tbl1002"."CustomerID" "Col1004","Tbl1002"."CompanyName" ,"Col1005",
"Tbl1002"."ContactName" "Col1006","Tbl1002"."ContactTitle" "Col1007","Tbl1002"."Address"
"Col1008","Tbl1002"."City" "Col1009","Tbl1002"."Region" "Col1010","Tbl1002".
"PostalCode" "Col1011","Tbl1002"."Country" "Col1012","Tbl1002"."Phone"
"Col1013","Tbl1002"."Fax" "Col1014" FROM "Northwind"."dbo"."customers" "Tbl1002" WHERE
"Tbl1002"."ContactName"=N'Marie Bertrand';
```

由于 OLE DB 模式有一个丰富的基于游标的更新模式，因此可以使用 SQL Server 的标准 *UPDATE* 语句更新远程数据源。这些计划与本章讨论的本地计划看起来是一致的，除顶级更新操作是专门针对于远程资源之外。由于 SQL Server 中的存储引擎模式最初是基于 OLE DB 接口的，因此执行本地和远程更新的机制实际上非常类似。在整个更新查询可以被远程控制的情况下，SQL Server 可以并且会生成在远程服务器上执行的完整远程 *INSERT*、*UPDATE* 和 *DELETE* 语句。您应该检查您认为可以完全被远程控制的所有查询，从而保证它们真正能够被远程控制——在某些情况下可能会有某条特殊的语法结构或内部函数是查询所不需要的，它们会阻止查询被完全远程控制。

分布式查询功能是 SQL Server 7.0 中引入的，该功能有一些限制，在设计使用它的情况时要考虑到。首先，该功能依赖于远程提供程序为 SQL Server 提供非常详细的基数和统计信息，从而使其能够比较不同的查询计划。由于大部分 OLE DB 提供程序不提供很多统计信息，因此可能会限制查询优化器生成的查询计划质量。此外，OLE DB 不是由 Microsoft 主动扩展的，因此某些提供程序没有得到积极的维护。同时，不是 SQL Server 中的每项功能都是通过远程查询机制支持的，如某些 XML 和基于 UDT 的功能。SQL Server 2008 没有一种本地机制支持为 CLR 运行时编写的查询管理适配器。最后，SQL Server 中使用的成本计算模型在一般情况下工作良好但是有时会生成一项比最优化慢很多的计划。遗憾的是，分布式查询功能中不远程控制某条查询的影响比本地情况大很多，因为从远程资源中移动行所需的工作会更多。在生产中应用功能之前使用该特性测试该功能时要小心。为常用的开销很大的查询预生成直接传递查询很有用，这样可以保证查询总是能够正确地被远程控制。

8.11 扩展的索引

SQL Server 包含很多特殊的索引用于支持特殊的情况。全文索引支持文档存储、查询和检索。XML 索引用于支持对存储在数据库中的 XML 数据进行 XQuery 操作。在 SQL Server 2008 中引入的空间索引支持对空间数据进行查询。这些索引通常比 B+树索引更适合于特殊的域，但是它们通常是特定于特殊使用情况的。虽然查询优化器包含对它们的支持，但是它们在概念上与 B+树索引不同。

8.11.1 全文索引

在 SQL Server 2008 中，全文索引已经从一种完全的外部结构变成一种基本上是内部的索引类型。SQL 语法中的特殊关键词通知系统隐式选择该索引，并且在优化进程开始之前执行。关于全文索引产生的信息和索引的其他详细信息都是从查询优化器中提取出来的，用于简化该索引的维护并避免重新编译以说明每个新索引的生成。

8.11.2 XML 索引

XML 索引与其他索引有些不同，它实际上的存储方式更像索引视图。它使用与聚集索引相同的物理

存储，并且在各个列上也有辅助索引。但是，这与查询优化器不匹配。与全文索引类似，XQuery 结构在语法中非常特殊，它们用于表示该操作应该始终使用该索引（如果存在）。

一个 XML 索引基本上具有 XML 文档中的每个属性和值并将它们放到自己的行中。除这些值之外，其他列会被添加到该结构中以存储某个值的属性及文档中的相关位置（根节点的路径）等信息。该信息使得特定 XQuery 结构在使用索引时明显更快。

在查询树表示法中，XML 使用很多非传统运算符表示。其中一个是流表值函数（Streaming Table-Valued Function，STVF），用于其他功能，包括 SQL Server 的动态管理对象。该结构允许 XML BLOB 分成几部分，每一部分返回一行。

8.11.3 空间索引

空间索引是 SQL Server 2008 中的新增功能，这些索引与查询优化器相匹配。空间索引分解一个空间同时允许点被索引。空间索引的主模型用于将空间分成区域，然后对每个区域使用边界框。在该计划中，一个 STVF 根据编码函数生成接近于请求点的候选点并且结果用于查询其他部分。图 8-76 包含一个示例空间索引查询计划。

```
CREATE TABLE geo(col1 INT PRIMARY KEY, point GEOGRAPHY);
CREATE SPATIAL INDEX spaidx ON geo(point);
SELECT * FROM geo WITH (INDEX=spaidx)
    WHERE geo.point.STEquals('POINT(24.0 24.0)')=1;
```

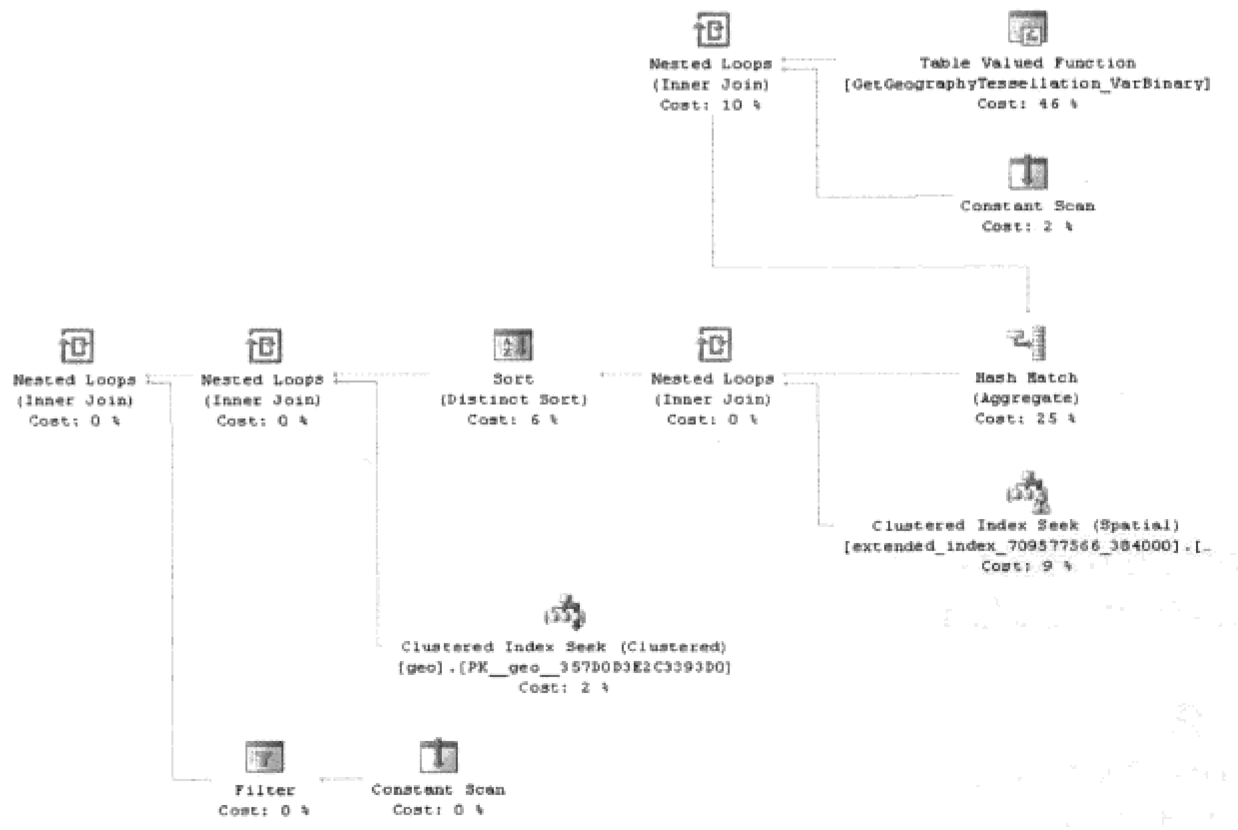


图 8-76　空间索引示例

空间索引在计划中使用比 STVF 更多的运算符。第一个 Constant Scan 和 Hash Match 帮助减少层次结构编码造成的重复。第一个循环联接从索引结构中检索值。该排序需要查找重复的基行（对于一个数据库行来说也许不只有一点）。第二个循环联接从基表中检索其他请求的列。最后，最后的循环联接应用 CLR 函数作为一个剩余谓词。

8.12 计划提示

有很多关于查询提示的错误信息，同时错误地使用提示可能会创建支持或反对在查询中使用提示的全局策略。本章的一个目标是为 DBA 和开发人员提供获得有关设计、实现和维护 SQL Server 应用程序等方面其他会话所需的工具。这部分将介绍查询提示及什么时候使用查询提示。

在本章的前面，我们介绍了工程师团队所面对的查询优化器问题。优化 SQL 查询过程中包含的某些算法非常复杂，以至于不可能有效地查看每个可发送到系统上每一查询的所有可能查询计划。基数估计中对统计信息收集和数学建模问题的延迟限制也对查询优化器的能力有一些限制，就处理器的当前计算能力来说，事实是查询优化器不能为某些查询生成一种理想的计划。

也就是说，查询优化器实际上对大部分查询进行了一项神奇的工作。该组件的开发吸收了多年来一些非常聪明的想法，结果使某个系统总能非常快地找到一种非常好的查询计划。这是通过很多精明的算法、探索及对常见情景的理解来完成的。产品的每个版本都是越来越好并且能够处理越来越多的问题。

人们问我关于提示的问题时，我首先会询问他们的应用程序。很多人没有意识到应用程序的设计对于提示是否恰当或必要具有很大的影响。如果数据库模式有一个经典的、三级范式的数据表并且查询都是使用美国国家标准委员会（ANSI）SQL 语法编写的，则 SQL Server 不用修改就可以对查询执行适当的工作。当在不同方向上改进系统并且加强设计时，可能会发现产品中的算法和探索法不能正常工作。例如，如果数据分布上有很大变化或者有一个应用程序依赖列值的统计信息相关性来选择一种良好的计划，则有时不能获得一种对查询来说接近最佳的联接顺序。因此在考虑提示之前，请保证理解应用程序是如何设计的，尤其是哪些种类的工作使应用程序不像一个常见的数据库应用程序。

如果您已经知道对应用程序来说非常重要的一个查询执行效果很差并且知道查询优化器的问题所在，则适合考虑利用提示来帮助解决问题。一般我会告诉人们不要使用提示，除非您有足够的理由，即系统的标准行为对于业务是不可接受的并且有一种更好的计划选择。因此，如果您知道某个特殊的联接顺序或索引选择会对系统中的其他查询造成死锁，则应该考虑使用提示——锁定不是查询优化器优化查询的一个因素，并且每个查询都是独立于系统中的其他查询被有效地优化的。

现在一些数据库开发团队可能强制“无提示”或“执行 *SELECT* 时总是在表上强制该索引”这样的规则。不是说他们不对——通常这些规则的存在是有充足理由的。建议您通读本章并保证与您的 DBA 交待每项规则存在的原因。当您构建一个新功能并修改数据库应用程序时，可能完全适合使用提示或者修改这些开发习惯。这一部分的目标是帮助您理解每一个提示的目的。我们希望这样可以使您知道什么时候适合使用提示（及什么时候不适合使用提示）。

几乎在所有情况下，查询和表提示实际上都是查询作者在生成一项查询计划时提出的需求。因此，如果一个提示不能得到满足，查询优化器实际上会返回一个错误并且根本就不返回计划。锁定提示是一种特殊情况——它们有时被忽视，用于保存使系统正常工作所需数据控制操作的正确性。虽然名称有点误导人，但是该行为允许您修改查询并知道您对查询生成过程有一定的影响。

8.12.1 调试计划问题

确定何时使用提示需要对查询优化器的工作及做出不正确决定的情况有一定的了解，然后对提示如何修改计划生成过程来定位问题有一定的了解。对于 Microsoft 在关于查询计划的支持电话中遇到的大部分问题来说，具有非正确基数估计的问题是生成较差计划选项的主要原因。换言之，一个良好的基数估

计会生成一种可接受的计划选项。其他情况下，更多关于开销、物理数据布局、锁升级、内存竞争的复杂问题或其他问题都是造成性能下降的因素。这一部分将介绍如何确定基数估计错误，然后利用提示修正较差的计划选项，因为通常有些问题无需购买新硬件或修改主机就可以解决。

识别基数估计错误的主要工具是 SQL Server 中的统计信息配置文件输出。启用该工具时，会为计划中每个查询运算符生成实际的基数。这些实际基数与查询优化器中的估计基数进行比较以找到差异。由于技术估计是从树的底部向上执行的（在显示计划的图形显示中是从右向左），因此错误会传播。通常最低的错误位置表示是考虑使用提示的位置。

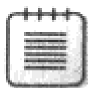
还有其他工具能够跟踪查询计划的性能问题。设置统计信息时间是决定查询运行时信息的一种良好方式。SQL Profiler 是跟踪死锁和其他系统级统计信息问题（可以通过跟踪进行捕获）的一个良好工具。*DBCC MEMORYSTATUS* 是查找系统中哪些组件正在引起 SQL Server 内部内存压力的一种非常好的工具。这些工具中的大部分都超出了本章的讨论范围，不过这些工具对于某些计划问题来说很有帮助。建议在关注计划质量时首先研究基数问题，因为这可能是最常见的问题。

在图 8-77 中，我们对某个目录视图进行一次查询，查看每个运算符的估计和实际基数是如何匹配的。

```
SET STATISTICS PROFILE ON;
SELECT * FROM sys.objects;
```

	Rows	Executes	StmtText	EstimateRows	EstimateExecutions
1	91	1	SELECT * FROM sys.objects;	91	NULL
2	91	1	|-Nested Loops(Left Outer Join, OUTER REFEREN...	91	1
3	91	1	|-Nested Loops(Left Outer Join, OUTER REFER...	91	1
4	91	1	| |-Filter(WHERE:(has_access('CO',[s1].[sys].[sy...	91	1
5	0	0	| | |-Compute Scalar(DEFINE:([Expr1006]=CO...	91	1
6	91	1	| | |-Clustered Index Scan(OBJECT:([s1].[sy...	91	1
7	0	91	| |-Clustered Index Seek(OBJECT:([s1].[sys].[sys...	1	91
8	91	91	|-Clustered Index Seek(OBJECT:([mssqlsystemres...	1	91

图 8-77　统计信息配置文件输出



注意：

EstimateRows 和 *EstimateExecutions* 列已经从屏幕快照显示的实际输出中被删除了。虽然对该查询的估计实际上是完美的，但是估计通常都会与实际基数有所不同，尤其是当查询变得越来越复杂时。通常，您希望估计值与实际值非常接近，这样选项就不用修改了，这几乎总是比树顶部的数量级小。同时注意 *EstimateRows* 数字是每次执行平均行数，而 *Rows* 是总行数。可以用 *Rows* 除以 *Executes* 得到可比较的数字。

如果在查看统计信息配置文件输出时发现错误，就更好确定查询优化器曾经使用错误信息做出决定的地方。通常利用完全扫描更新统计信息可以帮助分离这是一个过期问题还是采样信息不足的问题。如果查询优化器利用最新统计信息做出了一个较差的决定，则可能表示查询优化器使用一种缺乏模型的条件。例如，如果在一个查询的两列之间有很强的数据相关性，则可能会引起查询中出现基数问题。一旦发现缺乏模型是产生较差计划选项的原因，那么提示是改正计划选项并使用更好查询计划的一种机制。

这一部分将介绍大部分查询或表提示是如何适应查询优化器结构上下文的，包括适合使用提示的情况。

8.12.2 {HASH|ORDER}GROUP

SQL Server 实现 *GROUP BY*（和 *DISTINCT*）的方式有两种。可以通过存储行，然后根据同一组中的行物理相邻的事实实现，也可以将每一组散列到不同的内存位置。当其中一个选项被指定时，可以通过关闭其他物理运算符的实现规则实现。注意这会应用到查询中的所有 *GROUP BY* 操作上，包括该查询中包括的视图中的操作。

很多数据仓库查询都有一种聚合操作所遵循的、由很多联接组成的公共模式。如果对联接部分返回的行数的估计是错误的，则聚合操作的估计大小很可能不正确。如果被低估，则可能选择一种排序和流聚合。由于内存是根据估计的基数估计分配给每个处理器的，因此低估可能会引起排序溢出磁盘。在类似这样的情况下，提示一种哈希算法可能是一个很好的选项。同样，如果内存不足或者比预期的不同组值更多，可能使用一个流聚合会更适合。这种提示是影响系统性能的一种良好方式，尤其是在更大型查询及很多查询立即在某个系统上运行的情况下。

8.12.3 {MERGE|HASH|CONCAT}UNION

很多人在查询中应该使用 *UNION ALL* 的时候都错误地使用了 *UNION*，可能是因为 *UNION* 编写更容易吧。通常来说 *UNION ALL* 实际上是一种更快的操作，因为该运算符从每个输入中提取行并简单地返回全部。*UNION* 实际上必须比较两边的行，同时还要保证不返回副本。一般来说 *UNION* 执行一次 *UNION ALL*，然后在所有输出列上进行 *GROUP BY* 操作。在某些情况下，查询优化器可以决定输出列包含一个输入中唯一的键值，并且能够将 UNION 转换成一个 *UNION ALL*，但是一般来说要保证您实际上是在执行正确的查询。这 3 条提示只应用于 *UNION*。

现在假设您有正确的操作，您可以在 3 种联接模式中进行选择，这些提示让您指定使用哪种模式。下面的示例显示了 *MERGE UNION* 提示。

```
CREATE TABLE t1 (col1 INT);
CREATE TABLE t2 (col1 INT);
go
INSERT INTO t1(col1) VALUES (1), (2);
INSERT INTO t2(col1) VALUES (1);

SELECT * FROM t1
UNION
SELECT * FROM t2
OPTION (MERGE UNION);
```

正如您所看到的那样，每种提示都强制使用一种不同的查询计划模式。*MERGE UNION* 在有通用输入大小时有用。*CONCAT UNION* 在低基数计划（一次排序）时效果最好。*HASH UNION* 在有一个小型输入能够使一张哈希表与其他输入进行比较时效果最好。

UNION 提示与 *GROUP BY* 提示的作用大致相同——这两种操作通常在一个查询定义的顶级使用，如果在一个具有多个联接的查询中有基数估计错误，则这两种操作都可能变差。一般来说，人们或者提示 HASH 运算符来处理基数低估，或者提示 CONCAT 运算符来处理高估。

8.12.4 FORCE ORDER, {LOOP | MERGE | HASH } JOIN

联接顺序和算法提示是修正较差错误选项的常见技术。在估计满足一种联接条件的行数时，最好的

算法由输入的基数、这些输入的柱状图（用于估计多少行满足联接条件）、存储哈希表等数据的可用内存及哪些索引可用（可以加快循环联接）等因素决定。如果基数或柱状图不代表输入，则会产生一个较差的联接顺序或算法。此外，在数据与很难用当前技术（SQL Server 2008 中的筛选的统计信息只在一个表内起作用）建模的联接之间可能有一定的相关性。

提示：

如果统计信息配置输出显示基数评估明显不正确，则联接顺序可以通过按照自己希望在输出计划中查看表的顺序重写查询进行强制。这样将修改查询优化器设置初始联接顺序的方式并禁用重新排序联接的规则。一旦被提示，则应该对新查询进行计时并保证该计划比初始计划更快。此外，当输入发生变化时，您需要定期重新检查这些提示从而保证您已经强制执行的计划仍然适合——您一般会说“我比查询优化器更明白”，这与对您自己的汽车进行维护是一样。

我们所发现的适合这些提示的情况如下。

- 小型、类似 OLTP 这样关注锁定的查询。
- 具有很多联接、复杂的数据相关性并且一种固定查询模式已经足够的大型数据仓库系统，您可以分析对所有查询都有意义的联接顺序。例如，“如果我希望首先访问维度表，然后是事实数据表并且所有操作都是哈希联接”。
- 超出传统关系应用程序设计并且使用某种改变查询性能的引擎的系统。包括使用 SQL 作为存储全文本或 XQuery 组件的文档（与传统的关系组件相结合，或者对没有为 SQL Server 查询处理器提供表面统计信息的远程提供程序使用分布式查询）。

遗憾的是，在 SQL Server 2008 中没有显示半联接特定实现提示，虽然这种提示可能受到其他联接提示的间接影响。

8.12.5 INDEX=<indexname> | <indexid>

INDEX=<indexname> | < indexidid>提示已经在产品的很多版本中出现，对于强制查询优化器在编译计划时使用特殊索引来说非常有效。它主要关注的是 OLTP 应用程序，希望对 OLTP 强制一项计划从而避免任何类型的扫描。请记住查询优化器首先仅试图使用该索引来生成一项计划，但是如果您强制的索引不包含该查询（也就是说所有列均包含在索引键、表的主键中，或者作为一个 INCLUDED 列出），则查询优化器也会对表的其他索引添加联接。人们一般会使用可以用于对索引生成一个检索的查询筛选谓词，这种提示对于索引扫描也是有效的（如果您使用索引来缩短行宽以节省查询执行时间）。第二种有效的情况是在一台服务器上开发用于另一台服务器的计划，如对部署服务器的测试或者为一个应用程序创建的一个 ISV，然后将此应用程序发给客户端，让他们在自己的 SQL Server 实例上部署。

8.12.6 FORCESEEK

该提示是 SQL Server 2008 中的新增功能，用于通知查询优化器在使用索引时需要生成一个查找谓词。有几种情况查询优化器可以决定对索引的一次扫描比编译查询时的一次查找更好。例如，如果某个查询在表几乎为空的时候被编译，则存储引擎可能会在一个索引的一个页上存储所有现有行。此时就 I/O 而言，一次扫描比一次查找更快，因为存储引擎是从 B+树的叶级继续进行扫描，一次扫描可以避免使用一个额外的 I/O 页。这种情况可能是短暂的，因为新创建的表经常很快就被填充。这种提示对于避免如下

情况下很有效：您知道表不会有足够的行来触发重新编译，或者这种条件的性能影响对于保证提示的系统来说是很不利的。

使用这样提示的另一种情况是避免在 OLTP 应用程序中使用锁。这种提示排除了索引扫描，因此如果您有一个高级 OLTP 应用程序（缩放比例和并发是锁定关注的内容），那么它将是有效的。这种提示避免使用不必要的计划。因为查询优化器不会明确分析计划选择中的锁定（它不会选择具有更少锁的计划，但是它可能选择执行速度更快同时恰好有更少锁的计划）。只有在必要的时候才应该关注提示高级应用程序，因为一种较差的提示可能会使系统的性能比未使用提示计划时差很多。

8.12.7 FAST <number_rows>

查询优化器假设用户会读取查询生成的每一行。虽然一般情况下确实如此，但是某些用户状态（如手动浏览结果）并不遵循这种模式——在这些情况下，客户读取更少量的行然后关闭查询结果。通常会在不久的将来提交一个类似的查询用于从服务器上检索另一批行。在成本计算组件内，这种假设会影响计划的选择。例如，哈希联接对于大型结果集来说更有效，但是启动开销也更大（为联接的一端构建一个哈希表）。嵌套循环联接没有启动开销，但是每一行的开销会稍微高一些。因此，当一个客户只希望获得几行，但是没有指定仅返回几行的查询时，第一行的延迟可能会更慢，因为像哈希联接、导出查询脚本和排序这种定期被迫停止的运算符有一定的启动开销。

FAST <number_rows>提示提供具有用户希望从查询中读取多少行这样的提示的成本计算结构。在系统内部这被称为*行目标*，简单地为成本计算公式提供一个输入，帮助指定在成本计算函数的哪一点上适合用户查询。

SQL Server 中的 *TOP()*语法也使用了一个行目标。注意如果您提供 *TOP(@param)*，则查询优化器可能没有一个合适的值用于监听 T-SQL 上下文。在这种情况下，可能需要使用 OPTIMIZE FOR 提示（后面会介绍）。

8.12.8 MAXDOP <N>

MAXDOP 代表最大并行度，说明查询运行时（在 SKU 中支持并行查询计划）将被使用的首选展开级别。对于开销很大的查询来说，查询优化器试图使用多线程降低查询的运行时间。在成本计算函数中，这表示查询开销的一部分被划分到多个处理器核心上，与相同的串行计划相比降低了总体开销。非常复杂的查询实际上可能有多个区域并行，也就是说每个区域在执行期间最多可以分配 MAXDOP 个线程。

大型查询可能使用系统可用资源的非琐碎部分。一个并行查询可能使用内存和线程，阻止希望开始执行的其他查询。在某些情况下，这对于系统整体的健壮性是有益的，可以降低一个或多个查询的并行性等级，从而降低运行时间很长的查询所需的资源。这样对不使用资源控制器管理资源的工作负载有所帮助。在需要的时候，混合工作负载服务的服务器通常会考虑这种提示（如果需要）。

8.12.9 OPTIMIZE FOR

查询性能优化器在查询文本内使用标量值帮助估计查询中每个运算符的基数。这最终会帮助选择开销最低的计划，因为基数是成本计算函数的一个主要输入参数。参数化查询可能使该进程更困难，因为每次执行时参数可能都有所变化。假设 SQL Server 也自动参数化查询，则这种设计选项对查询的影响比人们所预期的更大。在估计参数化查询的基数时，查询优化器通常会使用精确度更低的方式估计列中不同值的平均数量或者监听上下文的参数值（遗憾的是通常只用于重新编译）。

窃听值用于基数估计和计划选择，但是不用于简化查询或依赖于特殊的参数值。因此，参数窃听可以帮助选择一种对某种特殊情况效果很好的计划。由于大部分数据集都具有不均匀的列分布，因此窃听的值可能影响查询计划的运行时间。如果表示通常分布的值被选择，则可能在平均情况下效果很好，但是在异常值（比平均值具有大量更多示例的值）情况下欠佳。如果异常值用于窃听值，则选择的计划可能比窃听平均值的效果明显更差。这可能成为 SQL Server 中计划缓冲策略造成的一个问题——参数化的查询被保留在缓冲中，即使不同执行情况的值有所变化。重新编译时，只有特殊上下文中的信息被重新编译。

OPTIMIZE FOR 提示允许查询作者指定编译期间使用的实际值，这可以用于通知查询优化器“这是我在运行时希望看到的一个常见值”，同时可以对参数化查询提供更多的计划预测。该提示对初始编译和重新编译均起作用。虽然指定一个常见值通常是最好的方法，但是请使用该提示以保证它提供所需的行为。

在清单 8-17 中，OPTIMIZE FOR 提示用于强制查询计划说明查询优化中的一个平均值（注意：我只是想说明计划改变，而不是说这两项计划的执行方式不同。这种技术可以用于对提示计划的任意复杂查询）。注意当值 23 被用于编译该查询（如图 8-79 所示）时，会与不是 23 的情况（如图 8-78 所示）选择不同的索引，因为 23 是一个非常常见的值并且不是与 *col2* 上的谓词一样具有选择性。参数值可以使查询计划的索引发生改变、联接顺序发生改变及其他更复杂的更改——强烈推荐测试强制参数。

清单 8-17 参数窃取示例

```
CREATE TABLE param1(col1 INT, col2 INT);
go
SET NOCOUNT ON;
BEGIN TRANSACTION;
DECLARE @a INT=0;
WHILE @a < 10000
BEGIN
INSERT INTO param1(col1, col2) VALUES (@a, @a);
SET @a+=1;
END;
COMMIT TRANSACTION;
go
CREATE INDEX i1 ON param1(col1);
go
CREATE INDEX i2 ON param1(col2);
go
DECLARE @b INT;
DECLARE @c INT;
SELECT * FROM param1 WHERE col1=@b AND col2=@c;
```

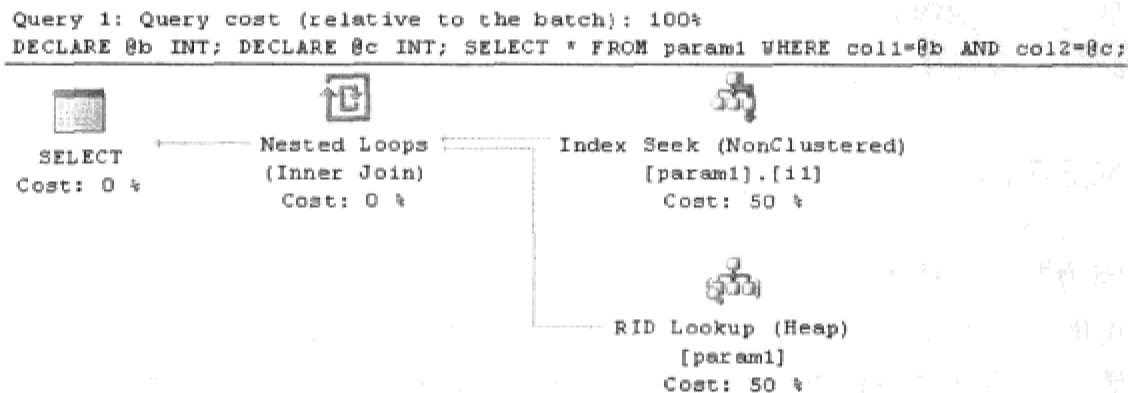


图 8-78 非窃听参数使用索引 i1

```
SELECT * FROM param1 WHERE col1=23 AND col2=5;
```

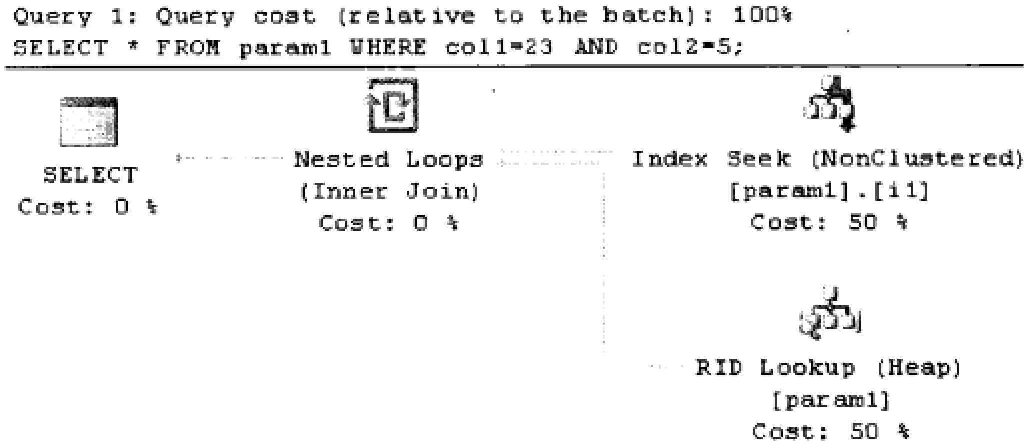


图 8-79 窃听参数使用 i2

```
DECLARE @b INT;
DECLARE @c INT;
SELECT * FROM param1 WHERE col1=@b AND col2=@c
OPTION (OPTIMIZE FOR (@b=22));
```

使用 OPTIMIZE FOR 提示通知查询优化器在生成计划时使用一个已知的常见值，从而使查询优化器能为大范围参数值工作。

8.12.10 PARAMETRIZATION{SIMPLE|FORCED}

SIMPLE 参数化是 SQL Server 很多版本中存在的一种模式，与本章介绍的琐碎计划的概念相对应。FORCED 参数化总是利用参数替换查询中的大部分文本。由于计划质量可能变差，因此使用 FORCED 时应该小心并且应该了解应用程序的全局行为。通常 FORCED 模式应该只用于具有很多几乎是等价查询的 OLTP 系统中（几乎总是产生相同的查询计划）。一般来说，您总认为计划不会在可能的参数值之间进行修改。如果所有的查询都非常小，则这种冒险的风险比较小。对这一提示的推理是某些具有特定查询的 OLTP 系统花费很长时间重复编译相同（或类似）的查询。在可能的情况下，考虑向应用程序查询添加参数是一种好办法。

8.12.11 NOEXPAND

默认情况下，查询处理器在解析和绑定查询树时扩展视图定义。虽然查询优化器通常在优化期间匹配索引视图（索引视图没有被指定时任意查询的一部分），但有时内部查询会被重写，从而不可能再匹配索引视图。NOEXPAND 提示强制查询处理器在最终的查询计划中使用索引视图。在很多情况下，这可以加速查询计划的执行，因为索引视图经常预计算查询开销很高的一部分。但是，并非始终如此——查询优化器也许能够利用完全扩展的查询树信息找到一种更好的计划。

8.12.12 USE PLAN

USE PLAN *N'xml plan'*提示指导查询优化器生成一种看起来与提供的 XML 字符串相似的计划。查询优化器利用该计划的结构作为指导优化进程的一系列提示，从而获取所需的计划结构。注意不能保证 _exact_ same 计划被选中，但是被选中的计划通常是一致的或非常接近。

这种提示的常见用途是 DBA 或数据库开发人员希望在查询优化器中确定计划回归时。如果良好的或

期望的查询计划的基准在开发或初次部署应用程序时被保存，则这些计划可以在后面用于强制一项查询计划修改回所期望的效果（如果查询优化器后来决定修改为一种执行效果不是很好的另一种计划）。这对于强制联接顺序来说非常必要，可以避免锁定死锁，或者仅仅是获得正确的物理计划形状和要选择的算法。在某些情况下，查询优化器没有足够的信息确定查询计划的一部分（例如联接顺序），同时可能产生一个不理想的计划选项。DBA 使用该选项时应该小心——强制原始查询计划实际上可能会进一步降低性能，因为计划可能是为不同的数据卷和分发创建的。如果可能的话，请在部署之前在测试数据库上试用计划提示。

虽然该功能是在 SQL Server 2005 中添加的，但是该功能在 SQL Server 2008 中已经有所改进，包括 Management Studio 对脚本的支持及提示更多类型查询的能力。例如，*INSERT/DELETE/UPDATE/MERGE* 查询现在在 USE PLAN 提示中得到支持，这可能对于强制避免在压力情况下出现死锁的特殊更新计划来说非常有用。

虽然 SQL Server 2008 支持额外的查询类型，但是有些类似还是不支持。这些类型包括：

- 动态的、键集和快速转发指针；
- 包含远程表的查询；
- 全文查询；
- DDL 命令，包括 *CREATE INDEX* 和 *ALTER PARTITION FUNCTION*，它们操纵数据。

在规则、属性和备注的上下文中，USE PLAN 提示被查询优化器用于控制查询树的初始形状（例如，树在早期优化过程中规范化之后的初始联接顺序）和为备注中每个组运行的规则。在联接顺序的情况下，查询优化器只启用使配置在计划提示中指定的联接顺序转换。物理实现规则也被提示，也就是说 XML 计划提示中的一个哈希聚合要求启用哈希聚合的实现规则同时禁用流聚合规则。

下面的示例说明了如何从 SQL Server 中检索计划提示并将其作为一个提示应用到后续编译中来保证查询计划：

```
CREATE TABLE customers(id INT, name NVARCHAR(100));
CREATE TABLE orders(orderid INT, customerid INT, amount MONEY);
go
SET SHOWPLAN_XML ON;
go
SELECT * FROM customers c INNER JOIN orders o ON c.id = o.customerid;
```

SELECT 语句返回包含该查询 XML 计划的 XML 文本的一行和一列。本书很难打印所有返回结果，结果的第一行是：

```
<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan" Version="1.0"
. . .
```

复制 XML 后，需要在 USE PLAN 提示中避免出现单引号。通常我们将 XML 复制到一个编辑器中，然后搜索单引号并用双引号替换单引号。接下来可以利用 OPTION（USE PLAN '<xml . . ./>'）提示将 XML 复制到查询中（该提示同样由于篇幅的关系被缩短）。

```
SET SHOWPLAN_XML OFF;
SELECT * FROM customers c INNER JOIN orders o ON c.id = o.customerid
OPTION (USE PLAN '<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/
showplan" Version="1.0" . . .');
```

利用该技术在操纵提交给服务器的查询时强制一项查询计划。XML 计划格式中使用的名称是逻辑名

称（表名称，而不是 *object_id*），因此应该可以从一个表中选择一个 USE PLAN 提示并只在需要少量修改的情况下将该提示应用到具有相同物理模式（列、索引等）的另一个表上。从一个表中将一项计划复制到具有相同结构的另一个表中（或者从一个数据库或 SQL Server 示例复制到另一个数据库或 SQL Server 示例中）的最简单方式是创建一个*计划向导*合并 USE PLAN 提示。计划向导将在第 9 章介绍。

8.13 小结

查询优化器是具有很多内部功能的一个复杂组件。虽然不可能每次都精确地知道查询优化器为什么选择一个特定计划，但是了解一些查询优化器设计方面的知识可以帮助 DBA 或数据库开发人员检查查询计划并分析问题产生的原因。知道查询优化器的工作方式也可以帮助提高应用程序质量同时减少部署问题。

本章介绍了查询处理和优化中使用的机制，包括树、规则、属性和备注框架。这些概念在优化的不同阶段使用，用于快速找到一种适当的计划。本章的示例介绍了很多运算符及如何使用这些运算符实现用户提交的 SQL 查询。最后，使用统计信息配置文件输出可以帮助确定优化较差的查询并利用统计信息和提示使查询优化器选择一种更好的计划。

第 9 章
计划缓存和重新编译

Kalen Delaney

我们已经介绍了 Microsoft SQL Server 中的查询优化流程和查询执行的细节。因为查询优化可能是一个复杂且费时的流程，所以 SQL Server 通常会重用已经生成并存储在计划缓存中的计划，而不是生成全新的计划。但是，在有些情况下，之前创建的计划可能不适用于当前查询执行，那么创建新计划可以获得更好的性能。

本章将介绍 SQL Server 2008 计划缓存及其组织方式。大部分讨论也与 SQL Server 2005 相关，我将介绍特定于 SQL Server 2008 的行为或功能。然后，我将介绍存储的计划类型，以及在何种情况下 SQL Server 可能会重用它们。还将介绍可能使现有计划重新创建的因素，以及描述计划缓存内容的元数据。最后，介绍在 SQL Server 可能创建新计划时，鼓励 SQL Server 使用现有计划的方法，以及在需要了解哪些最新计划可用时，如何强制 SQL Server 创建新计划。

9.1 计划缓存

SQL Server 2008 中的计划缓存实际上不是一个单独的内存区域，了解这一点很重要。SQL Server 7 之前的版本有两个有效的配置值用于控制计划缓存的大小，那时称为*过程缓存*。一个值指定 SQL Server 中总可用内存的固定大小，而另一个值指定专用于存储过程计划的内存百分比（在满足了固定需求之后）。此外，在 SQL Server 7 之前的版本中，用于即席 SQL 语句的查询计划从未在缓存中存储过，仅存储过程的计划存储在缓存中。这就是在旧版本中称其为过程缓存的原因。在 SQL Server 2008 中，总内存大小默认是动态的，而且用于查询计划的空间也是不固定的。

9.1.1 计划缓存元数据

本章第一部分将探讨计划可以重用的不同机制。要查看此计划重用（或非重用），需要查看几种不同的元数据对象。实际上有几十种不同的元数据视图和功能，它们提供关于计划缓存内容的信息，而且其中不包含有关计划缓存的内存使用信息的元数据。本章稍后将详细介绍计划缓存元数据，但现在我们只介绍一种视图和一种功能。此视图是 *sys.dm_exec_cached_plans*，它包含缓存中每个计划的一行内容，并且我们将查看 *usecounts*、*cacheobjtype* 和 *objtype* 列。*usecounts* 的值用于查看一种计划重用的次数。*cacheobjtype* 和 *objtype* 的可能值将在下一节介绍。当使用 CROSS APPLY 运算符连接 *sys.dm_exec_cached_plans* 视图和表值函数(TVF) *sys.dm_exec_sql_txt* 时，可以将 *plan_handle* 列中的值用做参数。这是我们使用的查询，我们称之为 *usecount 查询*:

```
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
    CROSS APPLY sys.dm_exec_sql_text(plan_handle)
```

```
WHERE cacheobjtype = 'Compiled Plan'
    AND [text] NOT LIKE '%dm_exec_cached_plans%';
```

9.1.2 清除计划缓存

因为 SQL Server 2008 具有缓存几乎所有查询的潜力，所以缓存中的计划数可以非常大。这是在缓存中查找计划的一种十分高效的机制，本章稍后将介绍。拥有大量的缓存计划，除了使用大量内存外，没有直接性能损失。但是，如果有许多非常相似的查询，则 SQL Server 查找正确计划的时间有时可能会非常长。此外，从测试和故障排除的角度来说，拥有大量计划有时可能会使仅查找我们目前感兴趣的计划变得更困难。SQL Server 提供了一种清除缓存中所有计划的机制，有时您可能想要在测试服务器上清除缓存中的计划，使缓存大小易于管理并易于检查。可以使用以下任意一种命令。

- ***DBCC FREEPROCCACHE***。此命令清除内存中的所有已缓存计划。SQL Server 2008 增加了为此命令添加参数的功能，以支持 SQL Server 从缓存中清除某一特定计划、具有相同 *sql_handle* 值的所有计划，或者是某一特定资源调控器资源池中的所有计划。本章稍后将在讨论检查计划缓存时介绍此过程。
- ***DBCC FREESYSTEMCACHE***。此命令除了清除计划缓存外，还清除所有 SQL Server 内存缓存。本章后面的“缓存存储”一节将介绍不同内存缓存的更多信息。
- ***DBCC FLUSHPROCINDB (<dbid>)***。此命令允许您指定某一特定的数据库 ID，然后清除这一特定数据库的所有计划。注意，此部分使用的 *usecount* 查询不会返回数据库 ID 信息，但是 *sys.dm_exec_sql_txt* 函数中含有此信息，所以 dbid 可以被添加到 *usecount* 查询中。

提示：

当然，不建议在生产服务器上使用这些命令，因为它会影响正在运行的应用程序的性能。通常将计划存储在缓存中。

9.2 缓存机制

SQL Server 可以避免编译之前执行的查询，方法是使用 4 种机制使计划缓存在许多情况下都可以访问：

- 即席查询缓存；
- 自动参数化；
- 已准备查询，使用 *sp_executesql* 或通过 API 调用的 prepare 和 execute 方法；
- 存储过程或其他编译对象（触发器和表值函数等）。

要确定缓存中每个计划的使用机制，需要查看 *sys.dm_exec_cached_plans* 视图中 *cacheobjtype* 和 *objtype* 列的值。*cacheobjtype* 列可以是以下 6 个值之一：

- *Compiled Plan*；
- *Compiled Plan Stub*；
- *Parse Tree*；
- *Extended Proc*；
- *CLR Compiled Func*；
- *CLR Compiled Proc*。

在本节中，我们仅查看值 *Compiled Plan* 和 *Compiled Plan Stub*。注意，我筛选了 *usecount* 查询以将结果限制为具有其中一个值的行。

objtype 列可以有 11 个不同的值：

- *Proc*（存储过程）；
- *Prepared*（预定义语句）；
- *Adhoc*（即席查询）；
- *ReplProc*（复制筛选过程）；
- *Triggger*；
- *View*；
- *Default*（默认约束或默认对象）；
- *UserTab*（用户表）；
- *SysTab*（系统表）；
- *Check*（CHECK 约束）；
- *Rule*（规则对象）。

我们主要检查前 3 个值，但是许多应用于存储过程的缓存细节也应用于复制筛选过程和触发器。

9.2.1 即席查询缓存

如果缓存元数据表示 *cacheobjtype* 的 *Compiled Plan* 值和 *objtype* 的 *Adhoc* 值，那么此计划就被认为是即席计划。在 SQL Server 2005 之前，有时会创建即席计划，但是并不能依赖此计划。但是，即使是 SQL Server 缓存即席查询，可能也不能依赖它们的重用。当 SQL Server 缓存即席查询的计划时，缓存的计划只有在后续批处理完全匹配时才能重用。使用此功能不需要任何额外的操作，但是它仅限于*完全文本匹配*。例如，如果在 *Northwind2*（可以在随附网站 http://www.SQLServerInternals.com/companion 中找到）数据库中执行以下三个查询，第一个查询和第三个查询使用相同的计划，但是第二个查询需要生成新计划：

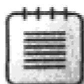
```
SELECT * FROM Orders WHERE CustomerID = 'HANAR';
SELECT * FROM Orders WHERE CustomerID = 'CHOPS';
SELECT * FROM Orders WHERE CustomerID = 'HANAR';
```

可以验证此查询，方法是首先清除计划缓存，然后在单独的批处理中运行三个查询，再运行前面提到的 *usecount* 查询：

```
USE Northwind2;
DBCC FREEPROCCACHE;
GO
SELECT * FROM Orders WHERE CustomerID = 'HANAR';
GO
SELECT * FROM Orders WHERE CustomerID = 'CHOPS';
GO
SELECT * FROM Orders WHERE CustomerID = 'HANAR';
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
    CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
    AND [text] NOT LIKE '%dm_exec_cached_plans%';
```

应该得到两行结果，因为 NOT LIKE 条件筛选出 *usecount* 查询本身的行。两行结果显示如下，其中一个计划仅使用了一次，而另一个计划使用了两次。

usecounts	cacheobjtype	objtype	文 本
1	Compiled Plan	Adhoc	SELECT * FROM Orders Where CustomerID = 'CHOPS'
1	Compiled Plan	Adhoc	SELECT * FROM Orders Where CustomerID = 'HANAR'



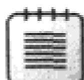
注意：

本节展示的结果使用设置为 0 的“针对即席工作负荷进行优化”配置选项（这是安装 SQL Server 时的默认设置）获得。本章稍后将介绍新的 SQL Server 2008 选项。

上述结果表明：更改了 *CustomerID* 值，相同计划就不能再重用。但是，要利用即席查询计划的重用，需要确保不仅在查询中使用相同的 *CustomerID* 值，而且查询的字符数还要相同。如果一个查询有另一个查询没有的新行或多余空格，则它们就视为不相同。如果一个查询包含另一个查询没有的注释，它们也不相同。此外，如果一个查询为标识符或关键字使用不同的大小写，即使是在不区分大小写的数据库中，查询也不相同。如果运行下面的代码，那么将看到没有查询可以重用相同的计划：

```
USE Northwind2;
DBCC FREEPROCCACHE;
GO
SELECT * FROM orders WHERE customerID = 'HANAR';
GO
-- Try it again
SELECT * FROM orders WHERE customerID = 'HANAR';
GO
SELECT * FROM orders
WHERE customerID = 'HANAR';
GO
SELECT * FROM Orders WHERE CustomerID = 'HANAR';
GO
select * from orders where customerid = 'HANAR';
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
    CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
    AND [text] NOT LIKE '%dm_exec_cached_plans%';
```

结果应该显示 *sys.dm_exec_cached_plans* 中的 5 行内容，每一行的 *usecount* 值都为 1。



注意：

SELECT 语句在各自的批处理中使用 GO 分隔。如果没有 GO，那么将只有一个批处理，并且每个批处理都有其自己的计划，此计划包含批处理中每个单独查询的执行计划。对于即席查询计划的重用，整个批处理必须是相同的。

有几种总是被认为是即席的特殊语句。这些构造包含以下内容：

- EXEC 使用的语句，如 EXEC `('SELECT FirstName, Lastname, Title FROOM Employees WHERE EmployeeID = 6');`
- 使用 *sp_exectuesql* 提交的语句，如果没有提供任何参数的话。

使用 *sp_prepare* 和 *sp_prepexec* 向应用程序提交的查询不被认为是即席的。

9.2.2 即席工作负荷优化

如果大多数查询是即席的且从未重用过，缓存它们的执行计划似乎是在浪费内存。本章稍后将介绍如何确定计划缓存的最大大小。可以肯定地说，对于很少重用的即席查询来说，具有成千上万个缓存计划可能不是 SQL Server 内存的最佳使用情况。基于此原因，SQL Server 2008 增加了一个配置选项，如果希望大多数查询都是即席的，可以使用此选项。启用此选项后，SQL Server 仅缓存第一次编译即席查询时查询计划的存根，并且只有在第二次编译后完整计划才会替代存根。

1. 控制“针对即席工作负荷进行优化”设置

启用“针对即席工作负荷进行优化”选项非常简单，如以下代码所示：

```
EXEC sp_configure 'optimize for ad hoc workloads', 1;
RECONFIGURE;
```

也可以使用 SQL Server Management Studio 启用此选项，此选项位于“服务器属性”对话框中的“高级”页面中，如图 9-1 所示。

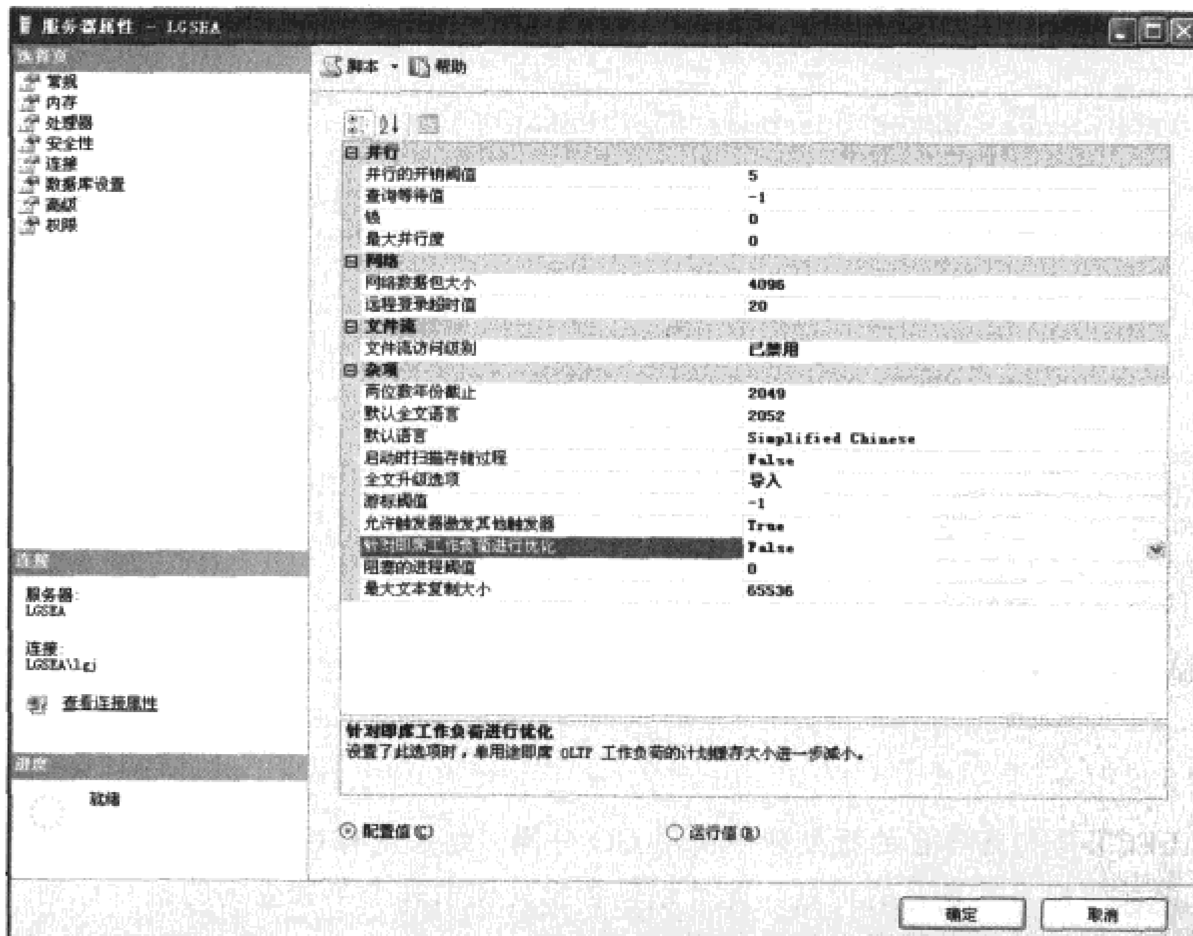


图 9-1 使用 Management Studio 中的“服务器属性”对话框启用“针对即席工作负荷进行优化”选项

2. 编译计划存根

启用“针对即席工作负荷进行优化”选项时，SQL Server 缓存的存根大小仅为 200 字节并且不包含查询执行计划的任何部分。它基本上仅是一个占位符，跟踪某一特定查询先前是否已编译。存根包含缓存键和实际查询文本的指针（存储在 SQL Manager 缓存中）。本章后面的“计划缓存内部”一节将介绍缓存键和 SQL Manager。对于编译的计划存根来说，缓存元数据中的 *usecounts* 值总是 1，因为它们从未被重用过。

当生成的编译计划存根的查询或批处理被重新编译时，存根会被完整的编译计划所代替。最初，*usecount* 值设置为 1，因为不能确保上一个查询具有完全相同的执行计划。所知道的只是查询本身是相同的。在启用“针对即席工作负荷进行优化”选项后，我将执行一些与在上一节使用的相同查询，并查看 *usecounts* 查询展示的内容。我需要稍微修改一下 *usecounts* 查询，并查找以 *Compiled Plan* 开头的 *cacheobjtype* 值，而不是查找 *Compiled Plan* 的值为 *cacheobjtype* 的行：

```
EXEC sp_configure 'optimize for ad hoc workloads', 1;
RECONFIGURE;
GO
USE Northwind2;
DBCC FREEPROCCACHE;
GO
SELECT * FROM Orders WHERE CustomerID = 'HANAR';
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
        CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype LIKE 'Compiled Plan%'
        AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO
SELECT * FROM Orders WHERE CustomerID = 'HANAR';
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
        CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype LIKE 'Compiled Plan%'
        AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO
```

第一次 usecounts 查询返回以下内容：

usecounts	cacheobjtype	objtype	文 本
1	Compiled Plan Stub	Adhoc	SELECT * FROM Orders WHERE CustomerID = 'HANAR'

第二次执行显示了使用编译的计划代替存根：

usecounts	cacheobjtype	objtype	文 本
1	Compiled Plan	Adhoc	SELECT * FROM Orders WHERE CustomerID = 'HANAR'

编译计划不是执行计划时生成此存根，所以如果使用一个“显示计划”选项仅检查了两次查询计划而没有执行查询，则会看到相同的行为。

如果“针对即席工作负荷进行优化”选项设置为 1，然后在将编译计划存根放置到计划缓存中时将其重置为 0，则存根不会立即从缓存中删除。当选项设置为 1 时，相同即席 T-SQL 批处理的任何重新提交将使用编译计划代替存根，并且不会再创建存根。

当工作负荷主要用于即席查询时，即使使用新的 SQL Server 2008 机制改进缓存行为，即席工作负荷也不是一个好方法。有时对提交给 SQL Server 的查询类型没有控制权，在这种情况下，可能会发现此选项很有用。但是，如果您和您的开发人员能够控制提交查询的方式，建议考虑其他选项，如已准备查询或存储过程，本章稍后将介绍该内容。

如果是边阅读文章边运行示例查询，可能想关闭“针对即席工作负荷进行优化”选项，方法如下：

```
EXEC sp_configure 'optimize for ad hoc workloads', 0;
RECONFIGURE;
GO
```

9.2.3 简单参数化

对于某些查询，SQL Server 可以决定将一个或多个常量看做参数。将常量看做参数时，使用相同基本模板的后续查询可以使用相同计划。例如，在 *Northwind2* 数据库中运行的以下两个查询可以使用相同计划：

```
SELECT FirstName, LastName, Title FROM Employees
WHERE EmployeeID = 6;
SELECT FirstName, LastName, Title FROM Employees
WHERE EmployeeID = 2;
```

SQL 在内部将这些查询参数化为以下内容：

```
SELECT FirstName, LastName, Title FROM Employees
WHERE EmployeeID = @1
```

通过运行以下代码并查看 *usecount* 查询的输出，可以查看此行为：

```
USE Northwind2
GO
DBCC FREEPROCCACHE;
GO
SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = 6;
GO
SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = 2;
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
    CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
    AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO
```

应该返回 3 行内容，内容如下所示。

usecounts	cacheobjtype	objtype	文　本
1	Compiled Plan	Adhoc	SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = 2

续表

usecounts	cacheobjtype	objtype	文　本
1	Compiled Plan	Adhoc	SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = 6
2	Compiled Plan	Adhoc	(@1 tinyint)SELECT [FirstName], [LastName], [Title] FROM [Employees] WHERE [EmployeeID] = @1

应该注意到具有不同常量的两个查询作为即席查询进行缓存。但是，这些仅被认为是 *shell* 查询，并且缓存仅是为了在稍后重用具有相同常量的相同查询时，能够更容易地查找查询的参数化版本。这些 *shell* 查询不包含完整执行计划，仅包含相应预定义计划中完整计划的指针。

提示：

不要混淆 shell 查询和计划存根。shell 查询包含查询的完整文本，至少使用 16KB 的内存。SQL Server 仅为那些认为可参数化的计划创建 shell 查询。如前所述，计划存根仅使用 200 字节的内存，并且仅为非参数化的、即席查询创建，并且仅在“针对即席工作负荷进行优化”选项设置为 1 时才可用。

在上面显示的输出中，*sys.dm_exec_cached_plans* 返回的第 3 行的 *objtype* 值为 *Prepared*（不能保证返回行的顺序。应该是两行的 *cacheobjtype* 值为 *Adhoc*，一行的 *cacheobjtype* 值为 *Prepared*）。查询计划与预定义计划关联，而且可以看到此计划使用了两次。此外，*Prepared* 行的文本显示参数代替了常量。

默认情况下，SQL Server 在决定何时自动进行参数化方面很保守。只有查询模板被认为是*安全的*，SQL Server 才会自动参数化查询。即使实际参数值发生更改，而选中的计划没有改变，就认为模板是安全的。这保证了参数化不会降低查询的性能。前面查询中使用的 *employees* 表格有一个唯一的索引，所以具有对 *employeeID* 进行相等比较的任何查询都能确保不会超过一行内容。无论使用什么实际值，对唯一索引使用查找计划可以很有用。

然而，要考虑非唯一列中具有相等比较或不等比较的查询。在这些情况下，一些实际值可能返回许多行，而其他值则返回一行内容或不返回内容。仅返回几行内容时，非聚集索引查找是个好选择，但是当返回多行内容时，则会是一个可怕的选择。所以根据查询中使用的值，具有多个可能最佳计划的查询被认为是不安全的且不能被参数化。默认情况下，SQL Server 对查询重用计划的唯一一种方式是使用上一节介绍的即席计划缓存（如果查询中的常量值不同，则不会发生此情况）。

除了要求每个查询模板仅有一个可能计划外，还有许多查询构造通常会禁止进行简单参数化。类似构造包含具有以下元素的任何语句：

- *JOIN*；
- *BULK INSERT*；
- *IN 列表*；
- *UNION*；
- *INTO*；
- *FOR BROWSE*；
- *OPTION* <查询提示>；
- *DISTINCT*；

- *TOP*；
- *WAITFOR* 语句；
- *GROUP BY*、*HAVING*、*COMPUTE*；
- 全文谓词；
- 子查询；
- *SELECT* 语句的 *FROM* 子句，具有表值方法、全文表或者 *OPENROWSET*、*OPENXML*、*OPENQUERY* 或 *OPENDATASOURCE*。
- 表单 EXPR <>一个非空常量的比较谓词。

简单的参数化还禁止使用以下构造的数据修改语句：

- 使用 FROM 子句进行 *DELETE/UPDATE*；
- 使用具有变量的 SET 子句进行 *UPDATE*。

1. 强制参数化

如果应用程序使用许多相似的查询，这些查询受益于同一计划，但是不会被自动参数化，可能是因为 SQL Server 认为计划不安全，也可能是因为它们使用其中一种禁用构造，此时 SQL Server 2008 会提供一个替代方案。名为 PARAMETERIZATION FORCED 的数据库选项可以使用以下命令启用：

```
ALTER DATABASE <database_name> SET PARAMETERIZATION FORCED;
```

启用此选项后，SQL Server 将常量看做参数，仅有少数几个异常。这些异常（在 *SQL Server 联机丛书*中列出）包含以下几种。

- INSERT ... EXECUTE 语句。
- 存储过程、触发器或用户定义的函数主体中的语句。SQL Server 已为这些例程重用了查询计划。
- 已在客户端应用程序中进行参数化的预定义语句。
- 包含 XQuery 方法调用的语句（如 WHERE 子句），该语句的方法位于其参数通常会被参数化的上下文中。如果方法位于其参数不会被参数化的上下文中，则语句的其他部分就会被参数化。
- T-SQL 游标中的语句（API 游标中的 SELECT 语句被参数化）。
- 不推荐使用的查询构造。
- 在 ANSI_PADDING 或 ANSI_NULLS 设置为 OFF 的上下文中运行的任何语句。
- 包含超过 2097 个字母的语句。
- 引用变量的语句，如 WHERE T.co12 >= @p。
- 包含 RECOMPILE 查询提示的语句。
- 包含 COMPUTE 子句的语句。
- 包含 WHERE CURRENT OF 子句的语句。

当为整个数据库将此选项设置为开时要谨慎，因为假设在优化过程中将所有常量都看做参数，然后重用现有计划通常会使性能降低。另一个选择是仅有选中的、被自动参数化的查询可使用计划指南，这将在本章末尾讨论。此外，如果数据库被设置为 PARAMETERIZATION FORCED，计划指南也可以用于覆盖选中查询的强制参数化。

2. 简单参数化的缺点

如之前 *usecount* 查询的输出所示，自动参数化的一个特点是 SQL Server 自主决定参数的数据类型，

此数据类型可能不是您认为将使用的数据类型。在前面的示例中，查看 *employees* 表，SQL Server 假定参数类型为 *tinyint*。如果重新运行批处理并使用不在 *tinyint* 范围（即小于 0 或大于 255 的值）内的值，则 SQL Server 无法使用相同的自动参数化查询。自动参数化下面的批处理是 SELECT 语句，但是两个查询不可以使用相同的计划。*usecount* 查询的输出应该显示两个即席 shell 查询和两个已准备查询。一个已准备查询的参数类型为 tinyint，另一个已准备查询的参数类型为 *smallint*。这看起来可能很奇怪，即使切换了查询的顺序并首先使用较大的值，仍会得到参数数据类型不同的两个已准备查询：

```
USE Northwind2;
GO
DBCC FREEPROCCACHE;
GO
SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = 6;
GO
SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = 622;
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
    CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
    AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO
```

强制 SQL Server 为不同查询使用同一数据类型的唯一方式是使数据库支持 PARAMETERIZATION FORCED。

如前所述，简单参数化并不总是恰当的，这就是为什么 SQL Server 在选择使用它时很保守的原因。考虑以下示例：*Northwind2* 数据库中的 *BigOrders* 表有 4150 行内容，共有 105 页，所以对于访问 *BigOrders* 表的任何查询来说，可以认为读取 105 页内容的表扫描可能是最大的性能影响因素。*CustomerID* 列有一个非聚集非唯一索引。如果为 *Northwind2* 数据库启用强制参数化，即使常量不同，第一个 SELECT 语句使用的计划也适用于第二个 SELECT 语句。第一个查询返回 5 行内容，第二个查询返回 155 行内容。通常，第一个 SELECT 语句会选择非聚集索引查找，而第二个语句会选择聚集索引扫描，因为符合条件的行数超过了表的页数。但是，启用 PARAMETERIZATION FORCED 不是我们所希望的，运行以下代码就会明白：

```
USE Northwind2;
GO
ALTER DATABASE Northwind2 SET PARAMETERIZATION FORCED;
GO
SET STATISTICS IO ON;
GO
DBCC FREEPROCCACHE;
GO
SELECT * FROM BigOrders WHERE CustomerID = 'CENTC'
GO
SELECT * FROM BigOrders WHERE CustomerID = 'SAVEA'
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
    CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
```

```
    AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO
ALTER DATABASE Northwind2 SET PARAMETERIZATION SIMPLE;
GO
```

运行此代码时，我们会看到，第一个 SELECT 语句需要 12 次逻辑读取，而第二个需要 312 次，几乎是扫描表所需读取次数的 3 倍。这里的 *usecount* 查询输出表明应用了强制参数化并且参数化的预定义计划使用了两次。

usecounts	cacheobjtype	objtype	文 本
1	Compiled Plan	Adhoc	SELECT * FROM Bigorders WHERE CustomerID = 'SAVEA'
1	Compiled Plan	Adhoc	SELECT * FROM Bigorders WHERE CustomerID = 'CENTC'
2	Compiled Plan	Adhoc	(@0 varchar(8000)select) * from Bigorders where CustomerID = @0

在本例中，强制 SQL Server 将常量看做参数不好，并且在最后一步中批处理将数据库重置为 PARAMETERIZATION SIMPLE（默认值）。注意，尽管我们正在使用 PARAMETERIZATION FORCED，但是参数化查询所使用的数量类型可能是最大的常规字符数据类型。

所以，如果有许多不应被参数化的查询和应被参数化的查询，应该怎么做呢？正如我们所见，与应用程序相比，SQL Server 查询处理器在决定模板是否安全方面比较保守。SQL Server 猜测哪些值实际上是参数，而应用程序开发人员实际上知道这些。不应该依赖 SQL Server 自动参数化查询，在知道值时，可以使用其中一种已准备查询机制将值标记为参数。

SQL Server 性能监视器包含名为 *SQLServer:SQL Statistics* 的对象，此对象有几个与自动参数化相关的计数器。可以监视这些计数器，确定是否有许多不安全或失败的自动参数化尝试。如果这些数字很大，可以检查应用程序是否有以下情形，即应用程序开发人员可以负责明确地标记参数。

9.2.4 已准备查询

如前所述，SQL Server 参数化的查询在缓存计划元数据中显示了一个 *objtype* 为 *Prepared* 的类型。还有另外两个拥有预定义计划的构造。这两个构造支持参数控制参数值和非参数值。此外，与简单参数化不同，程序设计器还确定参数使用的数据类型。一个构造是 SQL Server 存储过程 *sp_executesql*，该过程从 T-SQL 批处理内部调用；另一个构造从客户端应用程序使用 prepare 和 execute 方法。

1. *sp_executesql* 过程

存储过程 *sp_executesql* 位于即席缓存和存储过程的中间。使用 *sp_executesql* 需要确定参数及其数据类型，但是不需要存储过程和其他程序对象所需的所有持久对象管理。

下面是过程的基本语法：

```
sp_executesql @batch_text, @batch_parameter_definitions,
   param1,...paramN
```

具有相同值的 ***@batch_text*** 和 ***@batch_parameter_definitions*** 重复调用使用相同的缓存计划，并且指定了新参数值。只要计划没有从缓存中清除并仍然有效，就可以重用此计划。本章后面的“重新编译的原

因”一节讨论了这些情形，在这些情形中，SQL Server 确定计划不再有效。相同的缓存计划可用于以下所有查询：

```
EXEC sp_executesql N'SELECT FirstName, LastName, Title
    FROM Employees
    WHERE EmployeeID = @p', N'@p tinyint', 6;
EXEC sp_executesql N'SELECT FirstName, LastName, Title
    FROM Employees
    WHERE EmployeeID = @p', N'@p tinyint', 2;
EXEC sp_executesql N'SELECT FirstName, LastName, Title
    FROM Employees
    WHERE EmployeeID = @p', N'@p tinyint', 6;
```

与强制参数化相同，使用 *sp_executesql* 强制计划重用并不总是恰当的。如果使用与将数据库设置为 PARAMETERIZATION FORCED 的同一示例，可以看到使用 *sp_executesql* 是不恰当的：

```
USE Northwind2;
GO
SET STATISTICS IO ON;
GO
DBCC FREEPROCCACHE;
GO
EXEC sp_executesql N'SELECT * FROM BigOrders
    WHERE CustomerID = @p', N'@p nvarchar(10)', 'CENTC';
GO
EXEC sp_executesql N'SELECT * FROM BigOrders
    WHERE CustomerID = @p', N'@p nvarchar(10)', 'SAVEA';
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
    CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
    AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO
SET STATISTICS IO OFF;
GO
```

此外，还可以看到，第一个 SELECT 语句需要 12 次逻辑读取，第二个需要 312 次。*usecount* 查询的输出表明参数化查询使用了两次。注意，使用 *sp_executesql*，没有用于即席（非参数化）shell 查询的任何条目。

usecounts	cacheobjtype	objtype	文　本
2	Compiled Plan	Prepared	(@p nvarchar（10)) SELECT * FROM BigOrders WHERE CustomerID = @p

2. prepare 和 execute 方法

最后一种机制与批处理参数中的 *sp_executesql* 类似，它由应用程序确定，但是有一些主要差异。prepare 和 execute 方法不需要每次执行时都发送全文批处理。相反，在预定义时会发送一次全文，返回可在执行时调用批处理的句柄。ODBC 和 OLE DB 通过 *SQLPrepare/SQLExecute* 和 *ICommandPrepare* 展示此功能。在包含游标时，还可以通过 ODBC 和 OLE DB 使用此机制。使用这些功能时，SQL Server 将被

告知此批处理将重复使用。

3. 缓存已准备查询

如果已经使用 prepare 和 execute 方法在客户端对查询进行了参数化，则元数据将显示已准备查询，正如在服务器被自动参数化或使用 *sp_executesql* 进行参数化查询一样。但是，没有被参数化（简单参数化或强制参数化）的查询在缓存中没有相应的即席 shell 查询，也不包含非参数化的实际值，它们仅有已准备计划。检测已准备计划是 SQL Server 使用简单或强制参数化准备的，还是开发人员通过客户端参数化准备的，没有一种确定的方法。如果看到了一个相应的 shell 查询，可以知道查询是被 SQL Server 参数化的，但是相反的情形并不一定总是真的。因为 shell 查询具有零成本，所以在 SQL Server 的内存不足时，会首先将它们清除。因此缺少 shell 查询可能仅意味着已从缓存中清除了即席计划，而不是意味着从来没有进行过 shell 查询。

9.2.5 已编译对象

查看 *sys.dm_exec_cached_plans* 中的元数据时，我们看到了 *objtype* 值为 *Adhoc* 和 *Prepared* 的已编译计划。将讨论的第 3 个 *objtype* 值为 *Proc*，执行存储过程、用户定义标量函数和多语句 TVF 时，将看到此类型。对于这些对象，在执行这些对象时，可以完全控制作为参数的值及其数据类型。

1. 存储过程

存储过程和用户定义标量函数几乎是完全相同的。元数据表明 *objtype* 值为 *Proc* 的编译计划被缓存且可以被重复使用。默认情况下，所有后续执行都会重用此缓存计划，并且与 *sp_executesql* 一样，这并不一定总能令人满意。但是，与使用 *sp_executesql* 缓存和重用的计划不同，当执行对象时，可以使用存储过程和用户定义标量函数强制进行重新编译。此外，对于存储过程，可以创建对象，这样在每次执行计划时，都会创建一个新计划。

要为每次执行强制进行重新编译时，可以使用 EXECUTE...WITH RECOMPILE 选项。下面是一个示例，展示了在 N*orthwind2* 数据库中对存储过程进行强制重新编译：

```
USE Northwind2;
GO
CREATE PROCEDURE P_Customers
  @cust nvarchar(10)
AS
  SELECT RowNum, CustomerID, OrderDate, ShipCountry
  FROM BigOrders
WHERE CustomerID = @cust;
GO
DBCC FREEPROCCACHE;
GO
SET STATISTICS IO ON;
GO
EXEC P_Customers 'CENTC';
GO
EXEC P_Customers 'SAVEA';
GO
EXEC P_Customers 'SAVEA' WITH RECOMPILE;
```

如果查看 STATISTICS IO 的输出，可以看到第二次执行使用了一个非最佳计划，此计划所需读取的页

数比表扫描更多。此情形您可能见过，我们称之为*参数嗅探*（parameter sniffing）。SQL Server 基于第一个实际参数（在本例中是 CENTC）计划过程，并且后续执行假设使用的是相同或相似的参数。第三次执行使用 WITH RECOMPILE 选项强制 SQL Server 创建一个新计划，应该看到逻辑页读取数与表中的页数相同。

如果查看运行 *usecount* 查询的结果，如下所示，应该看到，*P_Customers* 过程的缓存计划的 *usecounts* 值为 2 而不是 3。

usecounts	cacheobjtype	objtype	文 本
2	Compiled Plan	Proc	CREATE PROCEDURE P_Customers @cust nvarchar(10) AS SELECT RowNum, CustomerID, OrderDate, ShipCountry FROM BigOrders WHERE CustomerID = @cust

对于使用 WITH RECOMPILE 选项执行的过程来说，针对此过程开发的计划仅在当前执行中有效，并且不会在缓存中保存以供将来重用。

2. 函数

用户定义标量函数和过程的行为方式可以完全相同。如果使用 EXECUTE 语句而不是表达式的一部分执行函数，还可以强制进行重新编译。下面是一个示例，展示了遮盖部分社保号码的函数。我们在 *pubs* 示例数据库中创建该函数，因为 *authors* 表在 *au_id* 列包含了社保号码：

```
USE pubs;
GO
CREATE FUNCTION dbo.fnMaskSSN (@ssn char(11))
RETURNS char(11)
AS
BEGIN
   SELECT @SSN = 'xxx-xx-' + right (@ssn,4);
   RETURN @SSN;
END;
GO
DBCC FREEPROCCACHE;
GO

DECLARE @mask char(11);
EXEC @mask = dbo.fnMaskSSN '123-45-6789';
SELECT @mask;
GO
DECLARE @mask char(11);
EXEC @mask = dbo.fnMaskSSN '123-66-1111';
SELECT @mask;
GO
DECLARE @mask char(11);
EXEC @mask = dbo.fnMaskSSN '123-66-1111' WITH RECOMPILE;
SELECT @mask;
GO
```

如果运行 *usecounts* 查询，应该注意到，函数的缓存计划的 *objtype* 值为 *Proc*，*usecounts* 值为 2。如

果在一个表达式内使用了标量函数，如本例所示，就没有办法要求进行重新编译：

```
SELECT dbo.fnMaskSSN(au_id), au_lname, au_fname, au_id  FROM authors;
```

根据定义 TVF 的方式，TVF 可能与过程相同或不同。可以将 TVF 定义为内联函数或多语句函数。当调用函数时，任何一种方法都可以进行强制重新编译。下面两个函数处理同样的事情：

```
USE Northwind2;
GO
CREATE FUNCTION Fnc_Inline_Customers (@cust nvarchar(10))
RETURNS TABLE
AS
 RETURN
 (SELECT RowNum, CustomerID, OrderDate, ShipCountry
 FROM BigOrders
 WHERE CustomerID = @cust);
GO

CREATE FUNCTION Fnc_Multi_Customers (@cust nvarchar(10))
RETURNS @T TABLE (RowNum int, CustomerID nchar(10), OrderDate datetime,
 ShipCountry nvarchar(30))
AS
BEGIN
 INSERT INTO @T
  SELECT RowNum, CustomerID, OrderDate, ShipCountry
  FROM BigOrders
  WHERE CustomerID = @cust
  RETURN
END;
GO
```

下面是对函数的调用：

```
DBCC FREEPROCCACHE
GO
SELECT * FROM Fnc_Multi_Customers('CENTC');
GO
SELECT * FROM Fnc_Inline_Customers('CENTC');
GO
SELECT * FROM Fnc_Multi_Customers('SAVEA');
GO
SELECT * FROM Fnc_Inline_Customers('SAVEA');
GO
```

如果运行 *usecounts* 查询，可以看到，仅有多语句函数重用了其计划。内联函数实际上被看做一个视图，并且只有在重新执行了完全相同的查询（即 SELECT 语句调用了具有完全相同参数的函数）之后，才可以重用计划，这是重用计划的唯一方式。

9.2.6 重新编译的原因

目前为止，我们讨论了 SQL Server 自动重用计划的情形，以及计划可能被不恰当地使用，以至于需要强制进行重新编译的情形。但是，也有这样的情形，即现有计划不能被重用，因为基础对象或执行环境发生了变化。产生重新编译的原因可分为两大类：与正确性相关的原因（与正确性相关的重新编译）

和与最优性相关的原因（与最优性相关的重新编译）。

1. 与正确性相关的重新编译

SQL Server 可能选择重新编译计划，如果它有理由怀疑现有计划不再是恰当的。当基础对象有明显更改（如数据类型更改或索引删除）时，会发生这种情况。显然，引用列的任何现有计划假设其以前的数据类型或使用现在不存在的索引访问的数据，都是不正确的。与正确性相关的重新编译分为两大类：架构更改和环境更改。下列更改标志着对象架构的更改：

- 为表或视图添加列，或者从表或视图中删除列；
- 为表添加约束、默认值或规则，或者从表中删除约束、默认值或规则；
- 为表或索引视图添加索引；
- 如果计划使用索引，则删除在表或索引视图中定义的索引；
- 删除在表（导致任何使用该表的查询计划进行与正确性相关的重新编译）中定义的统计信息；
- 为表添加触发器，或者从表中删除触发器。

此外，在表或视图上运行 *sp_recompile* 过程会更改对象的修改日期，可以在 *sys.objects* 的 *modify_date* 中查看此日期。这使 SQL Server 确定发生了架构更改，并在下次执行访问表或视图的任何存储过程、函数或触发器时进行重新编译。在过程、触发器或函数上运行 *sp_recompile* 会清除缓存中可执行对象的所有计划，以确保下次执行时将进行重新编译。

通过更改其中一个 SET 选项更改了环境时，会调用其他与正确性相关的重新编译。某些 SET 选项的更改会使查询返回不同结果，所以当其中一个值更改时，SQL Server 要确定使用的计划是在相似环境中创建的。SQL Server 会跟踪计划执行时设置的 SET 选项，并且您可以使用名为 *sys.dm_exec_plan_attributed* 的 DMF 访问这些 SET 选项的位图。当传入从 *sys.dm_exec_cached_plans* 视图获得的计划句柄值时会调用该函数，并为每个计划属性列表返回一行内容。需要确保在想要检索的列的列表中包含了 *plan_handle*，而不仅仅是先前在 *usecounts* 查询中使用的几列。下面是一个提供 *plan_handle* 值时检索所有计划属性的示例。表 9-1 展示了运行此代码的结果：

```
SELECT * FROM sys.dm_exec_plan_attributes
(0x06001200CF0B831CB821AA05000000000000000000000000)
```

表 9-1　　特定 *plan_handle* 对应的属性

属　性	值	Is_cache_key
set_options	4347	1
objectid	478350287	1
dbid	18	1
dbid_execute	0	1
user_id	-2	1
Language_id	0	1
date_format	1	1
date_first	7	1
Compact_level	100	1
status	0	1

续表

属　性	值	Is_cache_key
required_cursor_options	0	1
acceptable_cursor_options	0	1
merge_action_type	0	1
is_replication_specifi c	0	1
optional_spid	0	1
optional_clr_trigger_dbid	0	1
optional_clr_trigger_objid	0	11
inuse_exec_context	0	1
free_exec_context	1	1
hits_exec_context	0	1
misses_exec_context	0	1
removed_exec_context	0	1
inuse_cursors	0	0
free_cursors	0	0
hits_cursors	0	0
misses_cursors	0	0
removed_cursors	0	0
sql_handle	0x02000000CF0B831CBBE70632EC8A 8F7828AD6E6	0

本章稍后将介绍缓存管理和缓存内部原理，您将学习其中一些值（这些值的含义可能不是很明确），并且还将详细介绍跟踪计划的元数据。要获得与每个 *plan_handle* 一同返回的一行属性，使用 PIVOT 运算符并列出您想要在一行内包含的每个属性。在下一次查询中，我们将从属性列表中检索 *set_options*、*object_id* 和 *sql_handle*。

```
SELECT plan_handle, pvt.set_options, pvt.object_id, pvt.sql_handle
FROM (SELECT plan_handle, epa.attribute, epa.value
      FROM sys.dm_exec_cached_plans
          OUTER APPLY sys.dm_exec_plan_attributes(plan_handle) AS epa
      WHERE cacheobjtype = 'Compiled Plan'
    ) AS ecpa
PIVOT (MAX(ecpa.value) FOR ecpa.attribute
   IN ("set_options", "object_id", "sql_handle")) AS pvt;
```

得到的 *set_options* 值为 4347，这与位字符串 1000011111011 相同。要了解 SET 选项对应的位，可更改一个选项，然后查看位如何更改。例如，如果清除计划缓存并将 ANSI_NULLS 更改为 OFF，则 *set_options* 值将改为 4315 或二进制 1000011011011。4347 和 4315 的差值为 32，两者的不同之处在于第 6 位到第 8 位。如果不清除计划缓存，将得到同一批处理的两个计划，每个 *set_options* 值为一个计划。

尽管对 SET 选项的许多更改都会导致重新编译，但并不是所有更改都会导致重新编译。下面是更改 SET 选项时会导致重新编译的一个 SET 选项列表。

- ANSI_NULL_DFLT_OFF

- ANSI_NULL_DFLT_ON
- ANSI_NULLS
- ANSI_PADDING
- ANSI_WARNINGS
- ARITHABORT
- CONCAT_NULL_YIELDS_NUL
- DATEFIRST
- DATEFORMAT
- FORCEPLAN
- LANGUAGE
- NO_BROWSETABLE
- NUMERIC_ROUNDABORT
- QUOTED_IDENTIFIER

此列表中的两个 SET 选项具有与对象（包括存储过程、函数、视图和触发器）相关的一个特殊行为。ANSI_NULLS 和 QUOTED_IDENTIFIER 的 SET 选项设置实际上与对象定义保存在一起，并且在第一次创建对象时，总是使用实际 SET 值执行过程或函数。通过从 OBJECTPROPERTY 函数进行选择，确定想要为对象使用的两个 SET 选项值，如下所示：

```
SELECT OBJECTPROPERTY(object_id('<object name>'), 'ExecIsQuotedIdentOn');
SELECT OBJECTPROPERTY(object_id('<object name>'), 'ExecIsAnsiNullsOn');
```

返回的值为 0 意味着 SET 选项设置为 OFF，值为 1 则意味着 SET 选项设置为 ON，值为 NULL 则意味着输入有误或者您没有相应的权限。然而，尽管更改其中任何一个选项都不会造成对象执行的差异，但是 SQL Server 可能仍会重新编译访问对象的语句。不会进行重新编译的唯一对象是 *objtype* 值为 *Proc* 的缓存计划，即存储过程和多语句 TVF。对于这些编译对象，*usecounts* 查询展示了重用的相同计划，而不会展示具有不同 *set_options* 值的其他计划。如果更改了这些选项，则内联 TVF 和视图会创建新计划，并且 *set_options* 值表示不同的位图。但是，基础 SELECT 语句的行为不会更改。

2. 与最优性相关的重新编译

SQL Server 可能选择重新编译计划，如果它有理由怀疑现有计划不再是最优的。怀疑非最优计划的主要原因与基础数据的更改有关。如果用于生成查询计划的任何统计信息自计划创建后已经更新，或者如果有任何统计信息被认为是陈旧的，则 SQL Server 就会重新编译查询计划。

更新的统计信息。可以手动或自动更新统计信息。当有人运行 *sp_updatestats* 或 *UPDATE STATISTICS* 命令时会进行手动更新。当 SQL Server 确定现有统计信息已过期或陈旧时会进行自动更新，并且只有在数据库的选项 AUTO_UPDATE_STATISTICS 或 AUTO_UPDATE_STATISTICS_ASYNC 设置为 ON 时才会进行这些更新。如果另一个批处理尝试了使用当前计划使用的其中一个相同的表或索引，检测到统计信息已陈旧，并且启动了 UPDATE STATISTICS 运算，则会进行计划更新。

陈旧的统计信息。当 SQL Server 第一次编译在缓存中没有计划的批处理时，会检测过期的统计信息。它还会检测现有计划的陈旧统计信息。图 9-2 展示了一个步骤流程图，该步骤包括查找现有计划并检查是否需要进行重新编译。可以看到，在查看了是否有更新的统计信息可用后，SQL Server 检查了陈旧信息。如果有陈旧信息，统计信息就会被更新，然后从批处理开始进行重新编译。如果数据库的 AUTO_UPDATE_STATISTICS_ASYNC

为 ON，则 SQL Server 不会等待统计信息更新完成，它仅会基于陈旧信息进行重新编译。

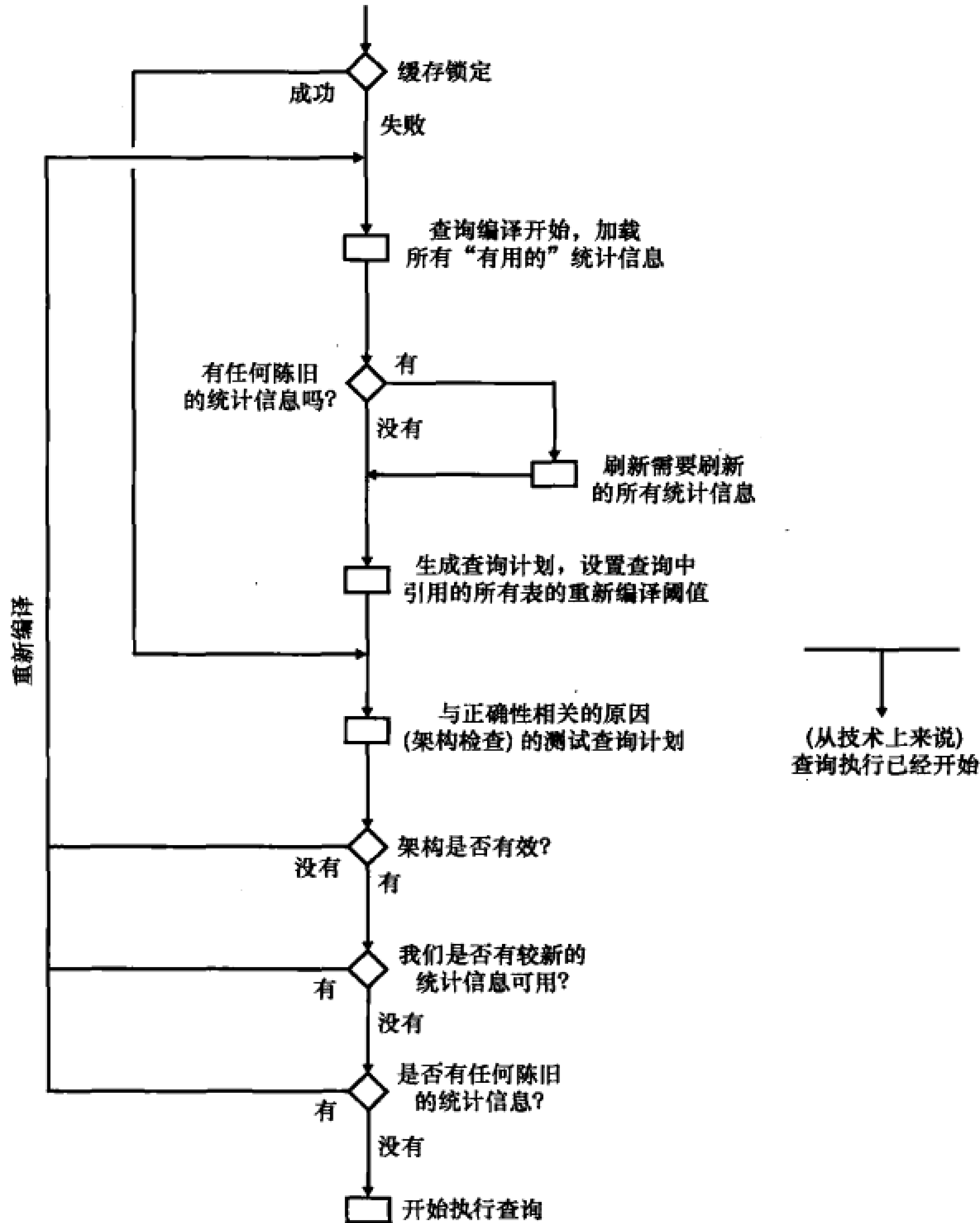


图 9-2 检查现有计划以查看是否需要重新编译

如果支持统计信息的列中已有足够的修改次数，则统计信息就会被认为是陈旧的。每个表都有一个重新编译阈值（RT），该阈值确定在表中的任何统计信息被标记为陈旧之前，更改可以发生的次数。批处理中引用的所有表的 RT 值都与批处理的查询计划保存在一起。

RT 值的类型（是永久的还是临时的）取决于表类型和编译计划时表中的行数。RT 值的正确算法随着服务包的不同而不同，我将介绍 SQL Server 2008 RTM 版本的算法。不同服务包使用的公式与此公式类似，但不能保证完全相同。*N* 表示表的基数。

- 对于永久表和临时表，如果 *N* 小于等于 500，RT 值为 500。这意味着，对于相对小的表，必须至少进行 500 次更改才能触发重新编译。对于较大的表，必须至少进行 500 次更改，外加行数的 20%。

- 对于临时表，算法相同，但有一个例外。如果表非常小或是空的（在任何数据修改操作之前，*N* 小于 6），我们所需做的是进行 6 次更改以触发重新编译。这意味着创建临时表（创建的是空表），然后在此临时表中插入 6 行或更多行的过程。只要此临时表能够访问就可以进行重新编译。

通过使用 KEEP PLAN 查询提示，可以解决（创建临时表的）此批处理的频繁重新编译。使用此提示更改临时表的重新编译阈值，并使其与永久表的阈值相同。所以如果对临时表的更改导致了许多重新编译，并且您认为重新编译影响了整体系统性能，可以使用此提示并查看是否有性能改进。可以在查询中指定提示，如下所示：

```
SELECT <column list>
FROM dbo.PermTable A INNER JOIN #TempTable B ON A.col1 = B.col2
WHERE <filter conditions>
OPTION (KEEP PLAN)
```

- 表变量没有 RT 值。这意味着更改表变量中的行数不会导致重新编译。

修改计数器。这里讨论的 RT 值是 SQL Server 确定陈旧统计信息所需的更改次数。在 SQL Server 2005 之前的 SQL Server 版本中，*sysindexes* 系统表跟踪名为 *rowmodctr* 的表列中实际发生的更改次数。这些计数器记录表或索引中任何行的更改，甚至会记录任何索引或有用统计信息中包含的列的更改。SQL Server 2008 现在使用一组列修改计数器（Column Modification Counter）或 *colmodctr* 值，表中的每列（非持久化计算列除外）都有一个单独计数。这些计数器不是事务性的，这意味着如果一个事务开始，在表中插入上千行，然后回滚，那么对修改计数器的更改*不会*回滚。与 *sysindexes* 中的 *rowmodctr* 值不同，用户看不到 *colmodctr* 值。它们仅在查询优化器内部提供。

使用 *colmodctr* 值跟踪表和索引视图的更改。SQL Server 跟踪的 *colmodctr* 值随着表数据的更改不断变化。表 9-2 描述了何时对 *colmodctr* 值进行修改，以及如何根据数据更改修改 *colmodctr* 值，包括 *INSERT*、*UPDATE*、*DELETE*、*BULK INSERT* 和 *TRUNCATE TABLE* 操作。尽管我们只具体介绍了表修改，但是请记住，还会跟踪索引视图的相同 *colmodctr* 值。

表 9-2　影响内部 *colmodctr* 值更改的因素

语　句	*colmodctr* 值的更改
INSERT	每个插入行的 *colmodctr* 值都增加 1
DELETE	每个删除行的 *colmodctr* 值都增加 1
UPDATE	如果是对非键列进行更新，每个更新行的修改列的 *colmodctr* 值都增加 1。如果是对键列进行更新，表中所有列的每个更新行的 *colmodctr* 值都增加 2
BULK INSERT	像 *N INSERT* 操作一样对待。所有 *colmodctr* 值都增加 *N*，*N* 是大容量插入的行数
TRUNCATE TABLE	像 *N DELETE* 操作一样对待。所有 *colmodctr* 值都增加 *N*，*N* 是表的基数

3. 跳过重新编译步骤

由于最优性原因，在几种情形下 SQL Server 会省略重新编译语句的步骤。这几种情形如下。

- 计划是琐碎计划时。琐碎计划是没有替代方案的计划，它基于查询所引用的表和索引（或缺少索引）。在这种情况下，确实只有一种方式来处理查询，任何重新编译都是浪费资源，无论有多少统计信息进行了更改。记住，没有任何保证能够说明查询将继续拥有一个琐碎计划，只是因为它原本有一个琐碎计划。如果自上次编译查询后，已经增加了新索引，那么现在可能有多种方式来处理查询。

- 如果查询包含OPTION提示KEEPFIXED PLAN，那么由于与最优性相关的任何原因，SQL Server都不会重新编译计划。
- 如果索引的统计信息和在表或索引视图上定义的统计信息的自动更新被禁用了，那么由这些索引或统计信息导致的与计划最优性相关的重新编译都将停止。

> **警告：**
>
> 关闭自动统计信息功能通常不是一个好方法，因为查询优化器对这些对象中的数据更改不再敏感，并且很容易生成非最优查询计划。只有在尝试了所有避免进行重新编译的方法之后，才能考虑使用这种方法。在更改了自动统计信息选项之后，应确保彻底测试应用程序，保证不会损害其他区域的性能。

如果查询中引用的表是只读的，则SQL Server不会重新编译计划。

4. 多次重新编译

在前面讨论的计划外重新编译中，主要介绍了在执行计划之前需要进行缓存计划重新编译的情形。但是，即使SQL Server认为它可以重用现有计划，仍然可能出现在开始执行批处理后，发现陈旧统计信息或架构发生变化的情况，这时就会在执行开始之后进行重新编译。每个批处理或存储过程可以包含多个查询计划，每个可优化语句一个。在SQL Server执行任何单个查询计划之前，它会检查该计划的正确性和最优性。如果其中一个检查失败，相应的语句就会再次进行编译，并且可能会生成不同的查询计划。

在某些情况下，即使批处理的计划没有被缓存，查询计划也可能被重新编译。例如，如果一个批处理包含一个大于8KB的文字，就不会缓存此批处理。然而，如果此批处理创建了一个临时表，然后在此表中插入了几行内容，传入了临时表的重新编译阈值，则插入的第 7 行内容就会重新编译。由于文字值较大，批处理不能进行缓存，因此当前正在执行的计划需要进行重新编译。

在SQL Server 2000中，当一个批处理被重新编译时，该批处理中的*所有*语句都会被重新编译，不会仅重新编译一个批处理。SQL Server 2005引入了语句级重新编译，这意味着仅造成重新编译的语句会创建新计划，而不是整个批处理都会创建新计划。这意味着SQL Server在重新编译时仅占用少量CPU时间和内存。

5. 从缓存中清除计划

如果已经从计划缓存中清除了之前的所有计划，SQL Server除了需要重新编译基于架构或统计信息更改的计划外，还需要为批处理编译计划。有关基于内存压力而清除的计划的内容，将在本章后面的“缓存大小管理”一节介绍。但是，其他操作也会导致从缓存中清除计划。其中一些操作可能会清除某个数据库中的所有计划，而一些操作可能会清除整个SQL Server实例的所有计划。

下列操作刷新整个计划缓存，所以之后提交的所有批处理将需要一个全新的计划。注意，尽管下面一些操作仅影响一个数据库，但整个计划缓存都会被清除。

- 将任何数据库升级到SQL Server 2008。
- 运行DBCC FREEPROCCACHE或DBCC FREESYSTEMCACHE命令。
- 更改以下任意一种配置选项。
  - cross db ownership chaining

- index create memory
- cost threshold for parallelism
- max degree of parallelism
- max text repl size
- min memory per query
- min server memory
- max server memory
- query governor cost limit
- query wait
- remote query timeout
- user options

以下操作清除与某一特定数据库相关的所有计划。

- 运行 *DBCC FLUSHPROCINDB* 命令。
- 断开数据库的连接。
- 关闭或打开 auto-close 数据库。
- 使用 *ALTER DATABASE . . . COLLATE* 命令修改数据库的排序规则。
- 使用以下任意一种命令更改数据库。
  - *ALTER DATABASE . . . MODIFY_NAME*
  - *ALTER DATABASE . . . MODIFY FILEGROUP*
  - *ALTER DATABASE . . . SET ONLINE*
  - *ALTER DATABASE . . . SET OFFLINE*
  - *ALTER DATABASE . . . SET EMERGENCY*
  - *ALTER DATABASE . . . SET READ_ONLY*
  - *ALTER DATABASE . . . SET READ_WRITE*
  - *ALTER DATABASE . . . COLLATE*
- 删除数据库。

从缓存中清除一个计划有几种不同的方式。第一，可以为缓存计划创建一个与 SQL 文本完全匹配的计划指南，然后就会自动清除具有该文本的所有计划。SQL Server 2008 提供了一种从计划缓存创建计划指南的简单方式。本章稍后将详细介绍计划指南。从缓存中清除单个计划的第二种方法是使用 SQL Server 2008 的新功能和 DBCC FREEPROCCACHE 的新选项。语法如以下代码所示：

```
DBCC FREEPROCCACHE [ ( { plan_handle | sql_handle | pool_name } ) ] [ WITH NO_INFOMSGS ]
```

现在此命令支持您指定 3 个参数中的一个，以说明您想从缓存中清除的计划。

- ***plan_handle***。通过指定 *plan_handle*，可以使用该句柄从缓存中清除一个计划（*plan_handle* 对所有现有计划来说都是唯一的）。
- ***sql_handle***。通过指定 *sql_handle*，可以使用该句柄从缓存中清除多个计划。如果任意一个缓存键值（如 SET 选项）发生了更改，那么同一个 SQL 文本可能存在多个计划。以下代码说明了这一点：

```
USE Northwind2;
GO
```

```
DBCC FREEPROCCACHE;
GO
SET ANSI_NULLS ON
GO
SELECT * FROM orders WHERE customerid = 'HANAR';
GO
SELECT * FROM Orders WHERE CustomerID = 'CENTC';
GO
SET ANSI_NULLS OFF
GO
SELECT * FROM orders WHERE customerid = 'HANAR';
GO
SET ANSI_NULLS ON
GO

-- Now examine the sys.dm_exec_query_stats view and notice two different rows for the
-- query searching for 'HANAR'
SELECT execution_count, text, sql_handle, query_plan
FROM sys.dm_exec_query_stats
   CROSS APPLY sys.dm_exec_sql_text(sql_handle) AS TXT
      CROSS APPLY sys.dm_exec_query_plan(plan_handle)AS PLN;
GO
-- The two rows containing 'HANAR'should have the same value for sql_handle;
-- Copy that sql_handle value and paste into the command below:
DBCC FREEPROCCACHE(0x02000000CECDF507D9D4D70720F581172A42506136AA80BA);
GO
-- If you examine sys.dm_exec_query_stats again, you see the rows for this query
-- have been removed
SELECT execution_count, text, sql_handle, query_plan
FROM sys.dm_exec_query_stats
   CROSS APPLY sys.dm_exec_sql_text(sql_handle) AS TXT
      CROSS APPLY sys.dm_exec_query_plan(plan_handle)AS PLN;
GO
```

- ***pool_name***。通过指定资源调控器（Resource Governor）池的名称，可以清除缓存中的所有计划，该缓存与使用指定资源池分配给工作负荷的查询相关（资源调控器、工作负荷组和资源池已在第 1 章介绍过）。

9.3　计划缓存内部

了解何时和如何重用或重新编译计划有助于设计高性能的应用程序。越了解最优查询计划，以及不同实际值和基数需要不同的计划，就越能决定何时必须进行重新编译。当您进行不必要的重新编译，或者认为 SQL Server 需要重新编译而它却没有时，越了解从内部管理计划的方式，故障排除就会越容易。本节将探讨计划缓存的内部结构、元数据可用性、SQL Server 如何在缓存中查找计划、计划缓存大小调整，以及计划清除策略。

9.3.1　缓存存储

SQL Server 中的计划缓存由 4 个单独的内存区域（称为*缓存存储*）组成。实际上在其内存中还有其他存储，这可以在 DMV 中的 *sys.dm_os_memory_cache_counters* 看到，但是只有 4 个区域包含了查询计划。下面括号中的名称是在 *sys.dm_os_memory_cache_counters* 的 *type* 列中看到的值。

- 对象计划（*CACHESTORE_OBJCP*）。对象计划包含存储过程、函数和触发器的计划。
- SQL 计划（*CACHESTORE_SQLCP*）。SQL 计划包含即席缓存计划、自动参数化计划和预定义计划的计划。管理 SQLCP 缓存存储的内存 clerk 也用于 SQL Manager，它管理即席查询中使用的所有 T-SQL 文本。
- 绑定树（*CACHESTORE_PHDR*）。绑定树是 SQL Server 中的 algebrizer 为视图、限制和默认值生成的结构。
- 扩展过程存储（*CACHESTORE_XPROC*）。Extended Procs (Xprocs)是预定义系统过程，如 *sp_executesql* 和 *sp_tracecreate*，它们使用动态链接库（DLL）而不是 T-SQL 语句进行定义。缓存的结构仅包含过程实施的函数名称和 DLL 名称。

每个计划缓存存储都包含一个哈希表，用于记录某个存储中的所有计划。哈希表中的每个存储桶包含 0 个、1 个或多个缓存计划。当决定使用哪个存储桶时，SQL Server 将使用一种非常简单的哈希算法。哈希键将作为(*object_id* database_id*) mod（哈希表大小）进行计算。对于与即席或预定义计划相关的计划，*object_id* 是批处理文本的内部哈希值。DMV *sys.dm_os_memory_cache_hash_tables* 包含有关每个哈希表的信息，包括其大小。您可以查询此视图以获取每个使用以下查询的计划缓存存储的存储桶数量：

```
SELECT type as 'plan cache store', buckets_count
FROM sys.dm_os_memory_cache_hash_tables
WHERE type IN ('CACHESTORE_OBJCP', 'CACHESTORE_SQLCP',
  'CACHESTORE_PHDR', 'CACHESTORE_XPROC');
```

您应该注意到，绑定树存储拥有对象计划和 SQL 计划的存储哈希存储桶数量的 10%（在 64 位系统上，对象计划和 SQL 计划存储的存储桶数量大约为 40 000；在 32 位系统上，该数量大约为 10 000）。扩展过程存储的存储桶数通常设置为 127 项。我们将不再深入介绍绑定树和扩展过程存储。本章的剩余部分仅介绍与对象计划和 SQL 计划相关的计划缓存。

要找出存储本身的大小，可以使用视图 *sys.dm_os_memory_objects*。以下查询返回具有计划的所有缓存存储的大小，以及 SQL Manager（存储所有即席和已准备查询的 T-SQL 文本）的大小。

```
SELECT type AS Store, SUM(pages_allocated_count) AS Pages_used
FROM sys.dm_os_memory_objects
WHERE type IN ('MEMOBJ_CACHESTOREOBJCP', 'MEMOBJ_CACHESTORESQLCP',
  'MEMOBJ_CACHESTOREXPROC', 'MEMOBJ_SQLMGR')
GROUP BY type
```

在缓存中查找计划分为两步。之前介绍的哈希键将 SQL Server 引导到可以找到计划的存储桶，但是如果存储桶中有多个项，SQL Server 就需要更多信息，以确定可以找到精确匹配的计划。第二步需要一个缓存键，它是几个计划属性的组合。之前我们介绍了可以传递 *plan_handle* 的 DMF *sys.dm_exec_plan_attributes*。得到的结果是某个特定计划的一个属性列表，布尔值表明此特定值是否是缓存键的一部分。表 9-1 包含了组成缓存键的 17 个属性，在 SQL Server 确定它在缓存中查找到一个匹配计划之前，它需要确保所有 17 个值都匹配。除了在 *sys.dm_exec_plan_attributes* 中找到的 17 个值以外，*sys.dm_exec_cached_plans.pool_id* 列也是任意计划的缓存键的一部分。

9.3.2 编译计划

在对象和 SQL 计划缓存存储中有两个主要计划类型：编译计划和执行计划。编译计划检查 *sys.dm_exec_cached_plans* 视图看到的对象类型。我们已经介绍了与编译计划（*Adhoc*、*Prepared* 和 *Proc*）对应的

3 种主要 *objtype* 值。根据编译计划拥有的 *objtype* 值，编译计划可以对象存储或 SQL 存储方式进行存储。编译计划被认为是有价值的内存对象，因为重新创建它们的成本昂贵。SQL Server 视图在缓存中保存它们。当 SQL Server 面临重大的内存压力时，用于清除缓存对象的策略确保我们的编译计划不是第一个清除的对象。

编译计划为整个批处理而不是单个语句生成。对于多语句批处理，可以将编译计划看做一个计划数组，数组的每个元素包含单个语句的一个查询计划。编译计划可以在多个会话或用户之间进行共享。但是，您应该明白，并不是执行相同计划的所有用户都会得到相同的结果，即使底层数据没有任何更改。除非编译计划是即席计划，每个用户拥有其自己的参数和本地变量，而且批处理可以针对具体用户构建临时表或工作表。与编译计划的某一特定执行相关的信息存储在名为*可执行计划*的另一个结构中。

9.3.3　执行上下文

可执行计划或执行上下文独立于编译计划，而且不会出现在 *sys.dm_exec_cached_plans* 视图中。执行计划是在执行编译计划时创建的运行时对象。正如编译计划一样，执行计划可以是存储在对象存储中的对象计划，也可以是存储在 SQL 存储中的 SQL 计划。每个执行计划与其依赖的编译计划位于相同的缓存存储中。执行计划包含编译计划执行一次的特定运行时信息，并且包含在运行时创建的对象的实际运行时参数、任意本地变量信息和对象 ID、用户 ID，以及批处理中目前正在执行的语句的信息。

当 SQL Server 开始执行编译计划时，它会从此编译计划生成一个执行计划。编译计划中的每个单独语句都会得到其自己的执行计划，可以将此执行计划看做运行时查询计划。与编译计划不同，执行计划是针对单独对话的。例如，如果 100 个用户正在同时执行相同的批处理，那么同一个编译计划将有 100 个执行计划。执行计划可以从与其相关的编译计划生成，并且创建它们相对比较廉价。本章稍后将介绍 *sys.dm_exec_cached_plan_dependent_objects* 视图，它包含关于执行计划的相关信息。注意，（当“针对即席工作负荷进行优化”配置选项设置为 1 时生成的）编译计划存根没有相关的执行上下文。

9.3.4　计划缓存元数据

当查看 *usecount* 信息以确定是否正在重用计划时，实际上已经查看了 *sys.dm_exec_cached_plans* DMV 中的一些信息。本节将介绍一些其他元数据对象，并讨论元数据中包含的一些数据的意义。

9.3.5　句柄

sys.dm_exec_cached_plans 视图包含每个编译计划名为 *plan_handle* 的值。*plan_handle* 是 SQL Server 从整个批处理的编译计划得到的一个哈希值，并且它对每个目前存在的编译计划来说都是唯一的(*plan_handle* 值可以随着时间的发展而重用)。*plan_handle* 可以作为编译计划的标识符使用。即使批处理中的每个语句因为正确性或最优性原因而进行了重新编译，*plan_handle* 仍然保持不变。

如前所述，根据计划是对象计划还是 SQL 计划，编译计划以两种缓存方式存储。批处理或对象的实际 SQL 文本存储在另一个名为 *SQL Manager Cache (SQLMGR)*的缓存中。与每个批处理相关的 T-SQL 文本存储在其整体中，包括所有注释。SQLMGR 缓存中缓存的 T-SQL 文本可以使用名为 *sql_handle* 的数据值获得。*sql_handle* 包含整个批处理文本的哈希值，而且因为对于每个批处理来说它都是唯一的，所以 *sql_handle* 可以作为 SQLMGR 缓存中批处理文本的标识符。

任意具体的 T-SQL 批处理总是具有相同的 *sql_handle*，但并不一定总是具有相同的 *plan_handle*。如

果缓存键中的任意一个值发生了更改，就会在计划缓存中得到一个新的 *plan_handle*。参见表 9-1，查看组成缓存键的计划属性。*sql_handle* 和 *plan_handle* 之间的关系是 1:*N*。

我们已经看到，*plan_handle* 可以轻易地从 *sys.dm_exec_cached_plans* 视图中获得。可以从先前查看的 *sys.dm_exec_plan_attributes* 函数中获得特定 *plan_handle* 对应的 *sql_handle* 值。下面是前面介绍的返回属性信息的相同查询，对它进行数据透视操作，以便 3 个属性作为 *plan_handle* 值在同一行返回：

```
SELECT plan_handle, pvt.set_options, pvt.object_id, pvt.sql_handle
FROM (SELECT plan_handle, epa.attribute, epa.value
     FROM sys.dm_exec_cached_plans
        OUTER APPLY sys.dm_exec_plan_attributes(plan_handle) AS epa
     WHERE cacheobjtype = 'Compiled Plan'
     ) AS ecpa
PIVOT (MAX(ecpa.value) FOR ecpa.attribute
   IN ("set_options", "object_id", "sql_handle")) AS pvt;
```

sys.dm_exec_query_stats 视图包含 *plan_handle* 和 *sql_handle* 值，以及关于执行每个计划的频率和执行中包含的工作量的信息。*sql_handle* 的值很隐密，有时很难确定每个 *sql_handle* 对应的查询。要获得此信息，可以使用另一个函数。

9.3.6 sys.dm_exec_sql_text

函数 *sys.dm_exec_sql_text* 可以将 *sql_handle* 或者 *plan_handle* 用做参数，返回对应句柄的 SQL 文本。当返回 SQL 时，SQL 文本中可能包含的任何敏感信息（如密码）都会被阻挡。函数输出的文本列包含了即席、预编译和自动参数化查询，以及对象（如触发器、过程和函数）的整个 SQL 批处理文本，它提供了完整的对象定义。

从 *sys.dm_exec_sql_text* 查看 SQL 文本很有用，它能快速确定由于几个不同因素而具有不同预编译的相同批处理。作为示例，考虑以下代码，它执行两个相同的批处理。该示例与先前介绍使用具有 *sql_handle* 的 *DBCC FREEPROCCACHE* 时使用的示例相同。但是这次查看 *sql_handle* 和 *plan_handle* 值，两次连续执行的唯一区别是 SET 选项的值 QUOTED_IDENTIFIER 发生了更改。在第一次执行中选项是 OFF，而第二次执行是 ON。执行完两个批处理后，检查 *sys.sm_exec_query_stats* 视图：

```
USE Northwind2;
DBCC FREEPROCCACHE;
SET QUOTED_IDENTIFIER OFF;
GO
-- this is an example of the relationship between
-- sql_handle and plan_handle
SELECT LastName, FirstName, Country
FROM Employees
WHERE Country <> 'USA';
GO
SET QUOTED_IDENTIFIER ON;
GO
-- this is an example of the relationship between
-- sql_handle and plan_handle
SELECT LastName, FirstName, Country
FROM Employees
WHERE Country <> 'USA';
GO
```

```
SELECT st.text, qs. sql_handle, qs.plan_handle
FROM sys.dm_exec_query_stats qs
    CROSS APPLY sys.dm_exec_sql_text(sql_handle) st;
GO
```

应该看到具有相同文本字符串和 *sql_handle* 的两行内容，但是具有不同的 *plan_handle* 值，如下所示（在我们的输出中，两个 *plan_handle* 值之间的差别仅是一个数，所以很难看到，但是再其他情况下，差别可能会更明显）。

文 本	*sql_handle*	*plan_handle*
-- this is an example of the -- relationship between -- *sql_handle* and *plan_handle* SELECT LastName, FirstName, Country FROM Employees WHERE Country <> 'USA'	0x0200000012330 B0EEA82077439354E7A 5B12E1B7E37A1361	0x0600120012330B0EB82 18705000000000000000 00000000
-- this is an example of the -- relationship between -- *sql_handle* and *plan_handle* SELECT LastName, FirstName, Country FROM Employees WHERE Country <> 'USA'	0x0200000012330 B0EEA82077439354E7A 5B12E1B7E37A1361	0x0600120012330B0EB82 18605000000000000000 00000000

可以看到，有两个计划对应同一个批处理文本，并且此示例应该明确一个重要性，即确保在重复执行同一查询时，影响计划缓存的所有 SET 选项都相同。您应该明确，无论编程界面对 SET 选项做了怎样的更改，都是为了确保不会无意得到不同的计划。例如，OSQL 界面使用 ODBC 驱动程序，它将每次连接的 QUOTED_IDENTIFIER 设置为 OFF，而 Management Studio 使用 ADO.NET，它将 QUOTED_IDENTIFIER 设置为 ON。从两个不同的客户端执行相同的批处理会在缓存中生成多个计划。

9.3.7 sys.dm_exec_query_plan

函数 *sys.dm_exec_query_plan* 是一个表值函数，它采用 *plan_handle* 作为参数，并返回 XML 格式的相关查询计划。如果计划是针对一个对象的，TVF 包含数据库 ID、对象 ID、过程号和对象的加密状态。如果计划是针对即席或已准备查询的，则这些值为 NULL。如果 *plan_handle* 对应于*编译计划存根*，那么查询计划也将是 NULL。我已在前面一些示例中使用过此函数。

9.3.8 sys.dm_exec_text_query_plan

函数 *sys.dm_exec_text_query_plan* 是一个表值函数，它采用 *plan_handle* 作为参数，并返回与 *sys.dm_exec_query_plan* 相同的基本信息。两个函数之间的差别如下。

- *sys.dm_exec_text_query_plan* 可以使用可选输出参数指定批处理语句的开始和结束偏移量。
- *sys.dm_exec_text_query_plan* 的输出将计划作为文本数据而不是 XML 数据返回。
- *sys.dm_exec_query_plan* 返回的查询计划的 XML 输出限制为 128 级的嵌套元素。如果计划超过了此限制，则会返回 NULL。

sys.dm_exec_text_query_plan 返回的查询计划的文本输出没有大小限制。

9.3.9 sys.dm_exec_cached_plans

sys.dm_exec_cached_plans 视图是解决查询计划重新编译问题最常用的一个视图。它也是我在第一节使用的视图，用来说明即席计划（与自动参数化和编译计划）的计划重用行为。该视图的每个缓存计划都有一行内容，并且除了 *plan_handle* 和 usecounts 外（我们已经查看过），该 DMV 还有关于缓存计划的其他有用信息，包括以下内容。

- ***size_in_byte***。该缓存对象使用的字节数。
- ***cacbeobjtype***。缓存对象的类型，即 *Compiled Plan*、*Parse Tree* 或 *Extended Proc*。
- ***memory_object_address***。缓存对象的内存地址，可用于获得缓存对象的内存故障位置。

尽管该 DMV 没有与每个编译计划相关的 SQL 文本，但我们已经看到，通过将 *plan_handle* 传给 *sys.dm_exec_sql_text* 函数，可以找到它。可以使用下面的查询获得编译计划的 *text*、*usecounts* 和 *size_in_bytes*，以及缓存中所有计划的 *cacheobjtype*。结果按频率顺序返回，首先显示最常使用的批处理：

```
SELECT st.text, cp.plan_handle, cp.usecounts, cp.size_in_bytes,
  cp.cacheobjtype, cp.objtype
FROM sys.dm_exec_cached_plans cp
  CROSS APPLY sys.dm_exec_sql_text(cp.plan_handle) st
ORDER BY cp.usecounts DESC;
```

9.3.10 sys.dm_exec_cached_plan_dependent_objects

将有效的 *plan_handle* 作为参数传入时，*sys.dm_exec_cached_plam_dependent_objects* 函数为编译计划的每个依赖对象返回一行内容。如果 *plan_handle* 不是编译计划的一部分，函数返回 NULL。相关对象包括执行计划（如前所述），以及编译计划使用的游标计划。下面的示例使用 *sys.dm_exec_cached_plan_dependent_objects* 和 *sys.dm_exec_cached_plans* 获取所有编译计划的依赖对象、*plan_handle* 和其 *usecounts*。它还调用 *sys.dm_exec_sql_text* 函数返回相关的 T-SQL 批处理：

```
SELECT text, plan_handle, d.usecounts, d.cacheobjtype
FROM sys.dm_exec_cached_plans
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
CROSS APPLY
   sys.dm_exec_cached_plan_dependent_objects(plan_handle) d;
```

9.3.11 sys.dm_exec_requests

sys.dm_exec_requests 视图为 SQL Server 示例中每个当前正在执行的请求返回一行内容，并且除了记录计划缓存信息之外，它还有许多用途。该 DMV 包含当前语句的 *sql_handle* 和 *plan_handle*，以及每个请求的资源使用信息。为了故障排除，可以使用该视图帮助确认长时间运行的查询。记住，*sql_handle* 指向整个批处理的 T-SQL。但是，*sys.dm_exec_requests* 视图包含 *statement_start_offset* 和 *statement_end_offset* 列，这表示在整个批处理中可以找到当前正在执行的语句的位置。开始偏移量设置为 0，而偏移量为-1 则表示批处理的结束。语句偏移量可以与传入 *sys.dm_exec_sql_text* 的 *sql_handle* 结合起来使用，以从整个批处理文本中提取查询文本，如以下代码所示。该查询返回目前正在执行的 10 个运行时间最长的查询：

```
SELECT TOP 10 SUBSTRING(text, (statement_start_offset/2) + 1,
   ((CASE statement_end_offset
       WHEN -1
```

```
            THEN DATALENGTH(text)
        ELSE statement_end_offset
    END - statement_start_offset)/2) + 1) AS query_text, *
FROM sys.dm_exec_requests
    CROSS APPLY sys.dm_exec_sql_text(sql_handle)
ORDER BY total_elapsed_time DESC;
```

注意，SELECT 列表中的“*”表示该查询应该返回 *sys.dm_exec_requests* 视图的所有列。您应该使用特别感兴趣的列（如 *start_time* 和 *blocking_session_id* 等）替换“*”。

9.3.12 sys.dm_exex_query_stats

正如 *sql_handle* 返回的文本是整个批处理的文本一样，编译计划返回的文本也是整个批处理的文本。为了最优化故障排除，可以使用 *sys.dm_exec_query_stats* 返回批处理中单个查询的性能信息。该视图返回查询的性能信息，总计超过同一查询的所有执行。该视图还返回 *sql_handle* 和 *plan_handle*，以及我们在 *sys.dm_exec_requests* 中看到的开始和结束偏移量。以下查询返回占总 CPU 时间最长的 10 个查询，帮助您确定 SQL Server 示例中最昂贵的查询：

```
SELECT TOP 10 SUBSTRING(text, (statement_start_offset/2) + 1,
    ((CASE statement_end_offset
        WHEN -1
            THEN DATALENGTH(text)
        ELSE statement_end_offset
    END - statement_start_offset)/2) + 1) AS query_text, *
FROM sys.dm_exec_query_stats
    CROSS APPLY sys.dm_exec_sql_text(sql_handle)
    CROSS APPLY sys.dm_exec_query_plan(plan_handle)
ORDER BY total_elapsed_time/execution_count DESC;
```

该视图具有批处理中每个查询语句的一行内容，当从缓存中清除计划时，该语句对应的行和累计的统计信息将从该视图中清除。除了 *plan_handle*、*sql_handle* 和性能信息外，该视图还包含 SQL Server 2008 的两个新列，这有助于确定具有不同计划的相同查询。

- ***query_hash***。该值是查询文本的哈希值，可用于确定具有不同计划缓存的相同查询。仅常量值不同的查询具有相同的 *query_hash* 值。
- ***query_plan_hash***。该值是查询执行计划的哈希值，可用于确定基于逻辑和物理操作符及操作符属性子集的相同计划。要查找可能不想实行强制参数化的情形，可以搜索具有相同 *query_hash* 和不同 *query_plan_hash* 的查询。

sys.dm_exec_cached_plans 和 *sys.dm_exec_query_stats* 有两个主要差别。第一，*sys.dm_exec_cached_plans* 拥有已经编译和缓存的每个批处理的一行内容，而 *sys.dm_exec_query_stats* 拥有每个语句的一行内容。第二，*sys.dm_exec_query_stats* 包含摘要信息，累计超过了某一特定语句的所有执行。*sys.dm_exec_query_stats* 返回每个查询的大量性能信息，包含执行的次数、累计 I/O、CPU 和持续信息。记住，只有当查询完成时，才会更新此视图，所以如果服务器上有大量工作负荷时，可能要多次获取信息。

9.3.13 缓存大小管理

我们已经介绍了计划重用和 SQL Server 如何在缓存中查找计划。本节将介绍 SQL Server 如何管理计划缓存大小，以及它在缓存中没有多余空间时如何确定要清除的计划。之前已经介绍了应从缓存中清除

计划的几种情形。这些情形包括全局操作（如运行 *DBCC FREEPROCCACHE* 从缓存中清除所有计划），以及对单个过程的更改（如 *ALTER PROCEDURE*，这会从缓存中清除过程的所有计划）。大多数情况下，仅在检测到内存压力时才会从缓存中清除计划。SQL Server 用于确定何时和如何从缓存中清除计划的算法称为*清除策略*。每个缓存存储都有其自己的清除策略，但是我们仅介绍对象计划存储和 SQL 计划存储的策略。

确定清除计划的依据是计划的成本，这将在下一节介绍。清除何时开始取决于内存压力。当 SQL Server 检测到内存压力时，会从缓存中清除零成本计划，而且所有其他计划的成本都减半。如第 1 章所述，有两种类型的内存压力，而且这两种类型都会从缓存中清除计划。这两种内存压力类型指*本地内存压力*和*全局内存压力*。

当谈到内存压力时，我们指*可见内存*。可见内存是 SQL Server 缓冲池直接可用的可寻址物理内存。在 32 位 SQL Server 实例上，可见内存的最大值是 2GB 或 3GB，这取决于在 boot.ini 文件中是否有*/3GB* 标志集。大于 2GB 或 3GB 的可寻址内存只能通过 *AWE 映射内存*间接使用。在 64 位 SQL Server 实例上，可见内存没有特殊意义，因为所有内存都是直接可寻址的。在下述的任何讨论中，如果提到大于 3GB 的可见目标内存，记住这只可能发生在 64 位 SQL Server 实例上。术语*目标内存*指可用于 SQL Server 流程的最大内存数。目标内存指可用于缓冲池的物理内存，而且它是您为最大服务器内存配置的较小值，也是操作系统可用的最大内存数。所以*可见目标内存*是目标内存的可见部分。查询计划仅可以非 AWE 映射内存的形式存储，这也是可见内存之所以重要的原因。您可以看到可见内存的值，指定为 8 KB 缓冲数，如 *sys.dm_os_sys_info* DMV 中的 *bpool_visible* 列。该视图还包含 *bpool_committed* 和 *bpool_commit_target* 的值。

SQL Server 定义缓存存储压力限制值，这根据正在运行的 SQL Server 版本和可见目标内存数的不同而不同。我们简要阐述如何使用该值。确定计划缓存压力限制值的公式在 SQL Server 2005 SP2 中发生了更改。表 9-3 展示了如何在 SQL Server 2000 和 2005 中确定计划缓存压力限制值，并展示了 SP2 中的更改，这使用较大的内存数降低了内存压力限制值。SQL Server 2008 RTM 使用在 SQL Server 2005 SP2 中新添加的公式。注意，这些公式可能会在未来的服务包中发生更改。

表 9-3　　确定计划缓存压力限制值

SQL Server 版本	缓存压力限制值
SQL Server 2005 RTM 和 SP1	可见目标内存的 75%，为 0GB～8GB；可见目标内存的 50%，为 8GB～64GB；可见目标内存的 25%，大于 64GB
SQL Server 2005 SP2 和 SP3、SQL Server 2008 RTM	可见目标内存的 75%，为 0GB～4GB；可见目标内存的 10%，为 4GB～64GB；可见目标内存的 5%，大于 64GB
SQL Server 2000	计划缓存的最大值为 4GB

举个例子，假设我们使用的是 SQL Server 2005 SP1、64 位的 SQL Server 实例且目标内存为 28GB。计划缓存压力限制值为 8GB 的 75%，目标内存 8GB 的 50%（或者 20GB 的 50%），这是 6GB 加 10GB，即 16GB。

在具有 28GB 目标内存的 64 位 SQL Server 2008 RTM 实例中，计划缓存压力限制值为 4GB 的 73% 加 4GB 的目标内存的 10%（或是 24GB 的 10%），这是 3GB 加 2.4GB，即 5.4GB。

1. 本地内存压力

如果任意单个缓存存储增长得过大，这表示本地内存压力，并且 SQL Server 开始从该存储中清除项。

该行为会防止一个存储占用太多的总系统内存。

如果缓存存储达到了计划缓存压力限制值的 75%，如表 9-3 所述，在单页分配或多页分配的计划缓存压力限制值的 50%中，会触发内部内存压力且会从缓存中清除计划。例如，在如前所述的情形中，我们计算出计划缓存压力限制值为 5.4GB。如果任何缓存存储超过该值的 75%，或者单页分配中为 4.05GB，则就会触发内部缓存压力。如果为缓存添加某个计划，则会导致缓存存储超过该限制值。从缓存中清除其他计划与添加新计划一样，这会增加新查询的响应时间。

除了内存总数达到某个特定限制值时会发生内存压力外，SQL Server 还会显示在存储中的计划数超过该存储中哈希表大小的 4 倍时的内存压力，不管计划的实际大小如何。如先前在缓存存储中所述，在 32 位系统和 64 位系统中的哈希表中各有 10 000 或 40 000 个存储桶。这意味着在 SQL Server 或是对象存储中拥有 40 000 或 160 000 项时，会触发内存压力。下面展示的第一个查询是我们之前看到的查询，它可用于确定对象存储和 SQL 存储的哈希表中的存储桶数；第二个查询返回这些存储中每个存储的项数：

```
SELECT type as 'plan cache store', buckets_count
FROM sys.dm_os_memory_cache_hash_tables
WHERE type IN ('CACHESTORE_OBJCP', 'CACHESTORE_SQLCP');
GO
SELECT type, count(*) total_entries
FROM sys.dm_os_memory_cache_entries
WHERE type IN ('CACHESTORE_SQLCP', 'CACHESTORE_OBJCP')
GROUP BY type;
GO
```

在 SQL Server 2008 之前，因为哈希表中的项数原因，内部内存压力很少被触发，但是几乎总是会触发缓存存储中的计划数。但是，在 SQL Server 2008 中，如果已经启用了“针对即席工作负荷进行优化”，SQL 缓存存储中的实际项可能相当小（每个*编译计划存根*大约为 300 字节），所以在存储数量变得过大之前，项数可能会超过限制值。如果未启用“针对即席工作负荷进行优化”，缓存中的项数会相当大，每个计划最小为 24KB。要查看缓存存储中所有计划的大小，需要检查 *sys.dm_exec_cached_plans*，如下所示：

```
SELECT objtype, count(*) AS 'number of plans',
      SUM(size_in_bytes)/(1024.0 * 1024.0 * 1024.0)
            AS size_in_gb_single_use_plans
FROM sys.dm_exec_cached_plans
GROUP BY objtype;
```

记住，即席计划和编译计划都以 SQL 缓存存储方式存储，所以要监控缓存大小，您需要添加这两个值。

2. 全局内存压力

全局内存压力适用于所有缓存存储使用的内存，并且可以是外部压力也可以是内部压力。由于服务器上的其他流程的计算需求，操作系统确定 SQL Server 流程需要减少其物理内存消耗时，会发生外部内存压力。当发生此种情况时，所有缓存存储大小都会减少。

当虚拟地址空间很低时，会发生内部全局内存压力。在内存 broker 预测到所有缓存存储将使用计划缓存压力限制值的 80%以上时，也会发生内部全局内存压力。再一次说明，当发生这种情况时，所有计划缓存都将清除项。

如前所述，当 SQL Server 检测到内存压力时，会从缓存中清除所有零成本计划，且所有其他计划的成本都会减半。任意特定循环更新每个缓存存储最多 16 项的成本。当已更新项具有零成本值时，它可以

被清除。没有任何可以清除正在使用的项的机制。但是，正在使用的编译计划的未使用的依赖对象都可能被清除。依赖对象包括执行计划和游标，并且当存在内存压力时，这些对象的多半内存都会被清除。记住，依赖对象的重新创建较容易，尤其是与编译计划相比。

更多信息：

有关内存管理和内存压力的更多信息，请参阅第 1 章。

9.3.14 缓存项的成本

从缓存中清除计划的决策取决于计划的成本。对于即席计划，成本可以被认为是零，但是每次重用计划时都会增加 1。对于其他类型的计划，成本是生成计划所需的资源的衡量标准。当重用这些计划中的任意一个时，成本被重置为原始成本。对于非即席查询，成本以刻度单位衡量，最大为 31。成本取决于 3 个因素：I/O、上下文切换和内存。每个最大值都在 31 刻度内。

- I/O：每个 I/O 成本为 1 刻度，最大值为 9。
- 上下文切换：每个为 1 刻度，最大值为 8。
- 内存：每 16 页为 1 刻度，最大值为 4。

当没有内存压力时，成本不会降低，除非所有计划的总大小达到缓冲池大小的 50%。此时，下一计划访问成本以 1 个单位递减。一旦遇到内存压力时，SQL Server 启动一个专用资源监控线程来降低某一特定缓存中的计划对象（本地压力）或所有计划缓存对象（全局压力）的成本。

sys.dm_os_memory_cache_entries DMV 显示任意缓存项的当前成本和原始成本，以及组成这些成本的组件：

```
SELECT text, objtype, refcounts, usecounts, size_in_bytes,
   disk_ios_count, context_switches_count,
   pages_allocated_count, original_cost, current_cost
FROM sys.dm_exec_cached_plans p
   CROSS APPLY sys.dm_exec_sql_text(plan_handle)
   JOIN sys.dm_os_memory_cache_entries e
   ON p.memory_object_address = e.memory_object_address
WHERE cacheobjtype = 'Compiled Plan'
   AND type in ('CACHESTORE_SQLCP', 'CACHESTORE_OBJCP')
ORDER BY objtype desc, usecounts DESC;
```

注意，通过连接 *memory_object_address* 列，可在 *sys.dm_os_memory_cache_entries* 中找到对应 *sys.dm_exec_cached_plans* 中某个计划的特定项。

9.4 计划缓存中的对象：概况

除了迄今为止讨论的 DMV 和 DMF 之外，还有另一个称为 *syscachedobjects* 的元数据对象（实际上仅是一个伪表）。在 SQL Server 2005 之前，没有任何动态管理对象，但我们有一些伪表，包括 *sysprocesses* 和 *syslockinfo*，它们不占用磁盘空间，仅在有人执行访问它们的查询时才会被物化，工作方式类似于动态管理对象。这些对象在 SQL Server 2008 中仍可用。在 SQL Server 2000 中，伪表仅存在于 *master* 数据库中，或者在引用它们时通过使用完全对象限定使它们可用。在 SQL Server 2008 中，可以将 *sys* 架构用做

限定，以便从任意数据库访问 *syscachedobjects*，所以我们使用其架构引用对象。表 9-4 列出了 *sys.syscachedobjects* 对象中的一些更有用的列。

表 9-4 sys.syscachedobjects 视图中有用的列

列名称	说明
bucketid	内部哈希表中该计划的存储桶 ID；存储桶 ID 有助于 SQL Server 更快地定位计划。具有相同存储桶 ID 的两行指同一对象（例如，同一个过程或触发器）
cacheobjtype	缓存中的对象类型：*Compiled Plan*、*Parse Tree* 等
objtype	对象类型：*Adhoc*、*Prepared*、*Proc* 等
objid	用于在缓存中查找对象的其中一个主键。这是存储在 *sysobjects* 中的数据库对象（过程、视图和触发器等）的 ID。对于缓存对象，如 *Adhoc* 或 *Prepared*，*objid* 是一个内部生成的值
dbid	编译缓存对象的数据库 ID
uid	计划（即席查询计划和编译计划）的创建者
refcounts	该缓存对象引用的其他缓存对象数
usecounts	从对象创建之时起，该缓存对象已经使用的次数
pagesused	缓存对象使用的内存页数
setopts	影响编译计划的 SET 选项。该列中值的更改表示用户已经修改了 SET 选项
langid	创建缓存对象的连接的语言 ID
dateformat	创建缓存对象的连接的数据格式
sql	提交的批处理中前 3900 个字符的模块定义

在 SQL Server 2000 中，*syscacheobjects* 伪表还包括可执行计划的项。也就是说 *cacheobjtype* 列中包含*可执行计划*的值。在 SQL Server 2008 中，因为执行计划被认为是依赖对象，且与编译计划分开存储，所以在 *sys.syscacheobjects* 视图中看不到它们。要访问执行计划，需要从 *sys.dm_exec_cached_plan_dependent_objects* 函数中直接选择执行计划，并将 *plan_handle* 作为参数传入。

作为 *sys.syscacheobjects* 视图（这是一种兼容性视图并且在未来的版本中不一定存在）的替代，您可以创建自己的视图，以便从 SQL Server 动态管理对象获取相同的信息。脚本在 *master* 数据库中创建了一个名为 *sp_cacheobjects* 的视图。记住，在主视图中创建的名称以 *sp_* 开头的任意对象，都可以从任意数据库直接访问，无需完全输入对象的名称。除了能从任意位置访问 *sp_cacheobjects* 视图外，创建自己的对象的另一个好处是可以自定义该对象。例如，再进行一次 OUTER APPLY 可能会相对简单，将该视图与 *sys.dm_exec_query_plan* 函数结合起来使用，可以获得缓存中每个计划的 XML 计划。

```
USE master
GO
CREATE VIEW sp_cacheobjects
   (bucketid, cacheobjtype, objtype, objid, dbid, dbidexec, uid,
   refcounts, usecounts, pagesused, setopts, langid, dateformat,
   status, lasttime, maxexectime, avgexectime, lastreads,
   lastwrites, sqlbytes, sql)
AS
   SELECT pvt.bucketid,
      CONVERT(nvarchar(18), pvt.cacheobjtype) AS cacheobjtype,
      pvt.objtype,
      CONVERT(int, pvt.objectid) AS object_id,
```

```
        CONVERT(smallint, pvt.dbid) AS dbid,
        CONVERT(smallint, pvt.dbid_execute) AS execute_dbid,
        CONVERT(smallint, pvt.user_id) AS user_id,
        pvt.refcounts, pvt.usecounts,
        pvt.size_in_bytes / 8192 AS size_in_bytes,
        CONVERT(int, pvt.set_options) AS setopts,
        CONVERT(smallint, pvt.language_id) AS langid,
        CONVERT(smallint, pvt.date_format) AS date_format,
        CONVERT(int, pvt.status) AS status,
        CONVERT(bigint, 0),
        CONVERT(bigint, 0),
        CONVERT(bigint, 0),
        CONVERT(bigint, 0),
        CONVERT(bigint, 0),
        CONVERT(int, LEN(CONVERT(nvarchar(max), fgs.text)) * 2),
        CONVERT(nvarchar(3900), fgs.text)
FROM (SELECT ecp.*, epa.attribute, epa.value
    FROM sys.dm_exec_cached_plans ecp
      OUTER APPLY
         sys.dm_exec_plan_attributes(ecp.plan_handle) epa) AS ecpa
PIVOT (MAX(ecpa.value) for ecpa.attribute IN
    ("set_options", "objectid", "dbid",
    "dbid_execute", "user_id", "language_id",
    "date_format", "status")) AS pvt
OUTER APPLY sys.dm_exec_sql_text(pvt.plan_handle) fgs;
```

您可能已经注意到，几个输出列的值已经被硬编码为 0。对于大部分输出列，这些列是在 SQL Server 2005 或 SQL Server 2008 中已经不存在的数据列。具体地说，这些列是关于缓存计划性能信息的报告。在 SQL Server 2000 中，还为每个批处理保留了该性能数据。在其之后的版本中，仅保留了语句级别的数据，并且仅通过 *sys.dm_exec_query_stats* 使用。要与 *sys.syscacheobjects* 视图兼容，新视图必须返回这些列中的一些内容。如果选择自定义该视图，可以选择清除这些列。

9.5 缓存中的多个计划

SQL Server 试图限制每个查询或过程的计划数。因为计划是重入的，所以可以轻松实现该目的。您应该注意一些情形，在这些情形下会在缓存中存储同一过程的多个查询计划。最有可能的情况是某些 SET 选项的差别，如前所述。

另一个连接问题会影响是否重用计划。如果所有者名称必须进行隐式解析，就无法重用计划。例如，假设用户 *sue* 执行了以下 SELECT 语句：

```
SELECT * FROM Orders;
```

SQL Server 首先尝试解析该对象，方法是在用户 *sue* 默认的架构中查找名为 *Orders* 的对象，如果没有找到这样的对象，则会在 *dbo* 架构中查找名为 *Orders* 的对象。如果用户 *dan* 执行了完全相同的查询，对象可以一种完全不同的方式进行解析（在用户 *dan* 的默认架构中查找表），所以 *sue* 和 *dan* 无法共享为该查询生成的计划。因为在使用不合格的对象名称时可能会存在不明确性，所以查询处理器假设不重用现有计划。但是，如果 *sue* 执行了以下命令，情况就不同了：

```
SELECT * FROM dbo.Orders
```

现在就不存在模糊性了。执行该同一查询的任何人总是会引用同一对象。在 *sys.syscacheobjects* 视图中，列 *uid* 表示生成计划的连接的用户 ID。对于即席查询，仅具有相同用户 ID 值的连接可以使用相同的计划。唯一例外是，用户 ID 值在 *syscacheobjects* 中被记录为-2，这表示提交的查询不依赖于隐式名称解析，并且可以在不同的用户之间进行共享。这是较常使用的一种方法。

提示：

强烈建议总是将对象与其包含的架构名称相匹配，这样就不必依赖于隐式解析，并且计划缓存的重用也会更高效。

9.6　何时使用存储过程和其他缓存机制

在决定是否重用存储过程或其中一种查询机制时，请牢记以下指导原则。

- **存储过程**。在多个连接正在执行参数已知的批处理时应使用这些对象。在需要控制何时重新编译代码块时，这些也很有用。
- **即席缓存**。此选项仅限于几种有限的情形。在设计想要正确控制适当计划重用的应用程序时，该选项不是很可靠。
- **简单或强制参数化**。此选项适用于无法被轻易修改的应用程序。但是，当您使用明确允许的声明参数及其数据类型的方法时，这很有用。比如下面的两个建议。
- ***sp_executesql* 过程**。当同一批处理可能被多次重用和参数已知时，此过程很有用。
- **prepare 和 execute 方法**。当多个用户正在执行参数已知的批处理时，或者当单个用户将多次使用同一批处理时，这些方法很有用。

9.7　计划缓存问题故障排除

要开始解决有关计划缓存使用和管理的问题，您必须确定现有问题确实是由计划缓存问题造成的。由于计划缓存的使用不当或管理不当，以及重新编译不当造成的性能问题，可能仅表现为吞吐量减少或查询响应时间增加。缓存问题也可以表现为内存不足错误或连接超时错误，这可能是由多种不同的原因造成的。

9.7.1　等待统计信息表明存在计划缓存问题

要确定计划缓存行为正在造成问题，其中一件事情就是查看 SQL Server 中的等待统计信息。等待统计信息将在第 10 章详细介绍，这里介绍一些表明计划缓存问题的主要等待类型。

查询 *sys.dm_os_wait_stats* 视图时会显示等待统计信息。下面的查询列出了 SQL Server 服务可能等待的所有资源，并显示了具有最常等待列表的资源：

```
SELECT *
FROM sys.dm_os_wait_stats
ORDER BY waiting_tasks_count DESC;
```

您应该注意到，此视图中展示的值是累计的，所以如果需要查看某个特定时间段正在等待的资源时，

则需要轮询此阶段的视图。如果看到以下任意一种资源具有相对长的等待时间，或者如果这些资源位于列表的顶部（由之前的查询返回的结果），则应该研究计划缓存的使用情况。

- **CMEMTHREAD 等待**。此等待类型表明在分配缓存描述符的内存对象上存在争用现象。向计划缓存插入大量项可能会导致争用问题。类似地，从缓存中清除项时和当资源监控线程关闭时也会发生争用问题。此外，正如我们所述，即席编译计划仅有一个单一缓存存储。

考虑到同一过程会调用几十次或几百次。记住，SQL Server 将缓存即席 shell 查询，即使过程本身仅有一个缓存计划。正如 SQL Server 开始经历内存压力一样，为每个单独过程调用插入项可能会导致过长的等待时间，导致吞吐量中断或者是生成内存不足的错误。

SQL Server 2005 SP2 增加了一些对缓存行为的改进以减轻缓存泛滥问题，在使用不同参数重复调用同一过程或参数查询时会发生这种问题。在 SQL Server 2005 SP2 之后的版本中，包含 SET 语句或事务控制的零成本批处理不会进行任何缓存。唯一例外是，仅包含 SET 和事务控制语句的那些批处理。这并不会成为很大的问题，因为在任何情况下，包含 SET 语句的批处理的计划都不会被重用。此外，对于 SQL Server 2005 SP2，缓存描述符分配的内存对象已在所有 CPU 中进行分区，以减轻内存对象（它可以减少 CMEMTHREAD 等待时间）的争用问题。

- **SOS_RESERVEDMEMBLOCKLIST 等待**。此等待类型可以表明具有大量参数的查询（或者是在 IN 语句中指定了大量值）的缓存计划的出现情况。这些查询类型需要 SQL Server 以较大的单位分配（*多页分配*）。可以查看 *sys.dm_os_memory_cache_counters* 视图，看看多页分配的内存数：

```
SELECT name, type, single_pages_kb, multi_pages_kb,
   single_pages_in_use_kb, multi_pages_in_use_kb
FROM sys.dm_os_memory_cache_counters
WHERE type = 'CACHESTORE_SQLCP' OR type = 'CACHESTORE_OBJCP';
```

使用 *DBCC FREEPROCCACHE* 清除计划缓存可以减轻由太多页分配造成的问题，最少直到重新执行查询和计划再次被缓存时。此外，SQL Server 2005 SP2 中缓存管理的更改也缩短了 SOS_RESERVEDMEMBLOCKLIST 的等待时间。也可以考虑重写应用程序，以使用长参数或长 IN 列表的替代方法。具体地说，长 IN 列表几乎总是可以得到改进，方法是在 IN 列表中创建值列表并连接该表。

- **RESOURCE_SEMAPHORE_QUERY_COMPILE 等待**。此等待类型表明有很多并发编译。要防止对查询内存的低效使用，SQL Server 2008 限制了其他内存使用的并发编译操作数。如果发现 RESOURCE_SEMAPHORE_QUERY_COMPILE 的值很高，可以通过 *sys.dm_exec_cached_plans* 视图检查计划缓存中的项，如下所示：

```
SELECT usecounts, cacheobjtype, objtype, bucketid, text
FROM sys.dm_exec_cached_plans
   CROSS APPLY sys.dm_exec_sql_text(plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
ORDER BY objtype;
```

如果结果中不包含具有 *objtype* 值的 *Prepared* 类型，则表示 SQL Server 不会自动参数化查询。可以尝试将本例中的数据库改为 PARAMETERIZATION FORCED，但是此选项会影响整个数据库，包括不会从参数化获益的查询。要强制 SQL Server 参数化某些查询，可以使用计划指南。下一节将介绍计划指南。

记住，每个批处理都会进行缓存。如果尝试使用 *sp_executesql* 或 prepare 和 excute 方法强制参数化，则批处理中的所有语句都必须参数化将重用的计划。如果批处理有一些参数化语句并且一些语句使用常量，则具有不同常量的每个批处理执行都不同，且在批处理中的参数化没有任何值。

9.7.2 其他缓存问题

除了查看可以表明缓存问题的等待类型外，还有一些其他编码行为对计划重用具有消极影响。

- **验证参数类型（已准备查询和自动参数化）**。对于已准备查询，实际上指定了参数类型，所以确保总是使用相同的类型会比较容易。当SQL Server进行参数化时，它自行决定数据类型。如果您查看Prepared查询类型的参数化表格，则会看到SQL Server已经假设了数据类型。如本章前面所述，值12345与值12是不同的数据类型，而且只有具体值不同的两个相同查询永远不会共享相同的自动参数化计划。

如果传递的参数是数值类型，SQL Server会根据其精确度和小数位数确定数据类型。值8.4的数据类型为*numeric(2,1)*，值8.44的数据类型为*numeric(3,2)*。对*carchar*数据类型，服务器端的参数化在实际值的长度方面不是很可靠。查看*Northwind2*数据库中的两个查询：

```
SELECT * FROM Customers
WHERE CompanyName = 'Around the Horn';
GO
SELECT * FROM Customers
WHERE CompanyName = 'Rattlesnake Canyon Grocery';
GO
```

- **监控计划缓存大小和数据缓存大小**。一般情况下，运行的查询越多，数据页缓存使用的内存数也会随着计划缓存使用的内存数的增加而增加。但是，如在讨论计划缓存大小时所述，在SP2之前的SQL Server 2005中，在内存压力迫使计划被清除之前，计划缓存的最大限制值应该为总缓冲池的80%。这可能会使依赖好的数据缓存行为的那些查询的性能降低。对于大于4GB的任意内存，SQL Server 2005 SP1之后的版本更改了在检测到内存压力之前的计划缓存的限制值。比较计划缓存使用的页数和数据缓存使用的页数的一种最简单的方法是性能计数器。查看以下计数器：SQL Server:Plan Cache/Cache Pages(_Total)和SQLServer:BufferManager/Database页面。

9.7.3 处理编译和重新编译问题

有工具可以检测过量的编译和重新编译。您可以使用系统监视器或第2章中介绍的一种跟踪或事件监控工具检测编译和重新编译。记住，编译和重新编译不是一回事。当现有模块或语句不再有效或不再是最优时需要进行重新编译。所有重新编译都是编译，但并不是所有编译都是重新编译。例如，当缓存中没有任何计划时，或者使用WITH RECOMPILE选项执行过程时，或者是执行使用WITH RECOMPILE选项创建的过程时，SQL Server将其视为编译而不是重新编译。

如果这些工具表明存在过量编译或重新编译，可以考虑以下操作。

- 如果重新编译是由SET选项的更改造成的，在重新编译表明是哪个SET选项发生了更改之前，SQL会立即跟踪T-SQL语句的文本数据。最好在第一次连接时更改SET选项，并且避免在提交了语句之后（或者是在存储过程内部）更改SET选项。
- 如本章前面所述，临时表的重新编译阈值比正常表的低。如果统计信息更改导致了临时表的重新编译，*EventSubclass*列中的数据值表明对临时表的某个操作导致统计信息发生了改变。可以考虑将临时表变为表变量，在表变量中不会维持统计信息。因为没有任何统计信息可维持，所以统计信息的更改不会导致重新编译。但是，缺少统计信息可能会导致这些查询的非最优计划。您自己的测试可确定表变量的受益是否值得。另一个选择是使用KEEP PLAN查询提示，这将设置临时表的重新编译阈值，该阈值与永久表的阈值相同。

- 要避免由于统计信息更改发生的重新编译，无论是永久表还是临时表，都可以指定 KEEPFIXED PLAN 查询提示。使用该提示，仅会在存在与正确性相关的原因时进行重新编译，如前所述。以下情形可能发生重新编译：如果语句引用的表的架构发生更改，或者是表通过使用 *sp_recompile* 存储过程标记为需要重新编译。
- 另一种避免由于统计信息更改而导致的重新编译方法是，关闭索引和列的统计信息自动更新。但是注意，关闭自动统计信息功能通常不是一个好方法。如果这样做，查询优化器不再对数据更改保持敏感，并很可能生成非最优计划。这种方法应该在没有办法时才使用。
- 所有 T-SQL 代码都应该使用两部分对象名称（例如，*Inventory.ProductList*），以明确表示正在引用的对象，这有助于避免进行重新编译。
- 不要在条件构造（比如 IF 语句）内部使用 DDL。
- 查看存储过程是否是使用 WITH RECOMPILE 选项创建的。在许多情况下，存储过程中仅有一个或两个语句会从每次执行的重新编译中获益，而且我们仅可以对这些语句使用 RECOMPILE 查询提示。这比为整个过程使用 WITH RECOMPILE 选项（每次执行过程时都会对过程中的每个语句进行重新编译）要好很多。

9.7.4 计划指南和优化提示

第 8 章介绍了许多不同的执行计划，并讨论了对查询进行优化意味着什么。本章介绍了 SQL Server 重用计划的几种情形，当生成一个新计划可能最好时，我们也见过这样的情形，尽管缓存中已经存在一个非常好的计划，但是 SQL Server 却不重用该计划。鼓励计划重用的一种方式是启用 PARAMETERIZATION FORCED 数据库选项，这已在本章前面介绍过。在其他情况下，我们无法使用优化器重用计划，我们可以使用优化器提示。优化器提示也可用于强制 SQL Server 在使用现有计划时生成一个新计划。有几十种提示都可以用在 T-SQL 代码中以影响 SQL Server 生成的计划，其中一些已经在第 8 章介绍过了。本节将具体介绍那些影响重新编译的提示及所有提示之母(USE PLAN)，这是在 SQL Server 2005 中新增的内容。最后我们将介绍名为*计划指南*的 SQL Server 功能。

1. 优化提示

本节介绍的所有提示在 *SQL Server 联机丛书*中都称为查询提示，以与表提示（在表名之后的 FROM 子句中指定）和连接提示（在词 JOIN 之前的 JOIN 子句中指定）进行区分。但是，我们通常将查询提示称为选项提示，因为它们在名为 OPTION 子句的特殊子句中指定，它仅用于指定此类型的提示。OPTION 子句（如果查询中包含该子句）总是任意 T-SQL 语句的最后一个子句，如下一节中的代码示例所示。

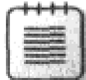
RECOMPILE。RECOMPILE 提示强制 SQL Server 重新编译查询。当批处理中仅有一个语句需要重新编译时，该提示特别有用。您知道 SQL Server 将 T-SQL 批处理编译为单位，确定批处理中每个语句的执行计划，并且它不会执行任何语句，除非整个批处理都进行了编译。这意味着如果批处理包含一个变量声明和指定，该指定不会在编译阶段发生。当以下批处理被优化时，SQL Server 不会为变量生成特定值：

```
USE Northwind2;
DECLARE @custID nchar(10);
SET @custID = 'LAZYK';
SELECT * FROM Orders WHERE CustomerID = @custID;
```

SELECT 语句的计划将显示，SQL Server 正在扫描整个群集索引，因为在优化阶段 SQL Server 不知

道将搜索哪个值，并且无法使用索引统计信息中的算法以获得对行数的好的评估。如果使用常量 LAZYK 替换变量，SQL Server 认为仅有几行内容符合条件且使用 *customerID* 的非聚集索引。此时 RECOMPILE 提示很有用，因为它告诉优化器在语句执行之前为单个 SELECT 语句生成新计划，这是在 SET 语句执行了以后：

```
USE Northwind2;
DECLARE @custID nchar(10);
SET @custID = 'LAZYK';
SELECT * FROM Orders WHERE CustomerID = @custID
OPTION (RECOMPILE);
```



注意：

变量与参数不同，尽管编写它们的方式相同。因为过程仅有在执行时才能编译，所以 SQL Server 总是使用特定的参数值。当之前的编译计划用于不同的参数时就会出现问题。但是，对于本地变量，除非使用 RECOMPILE 提示，否则在使用变量的语句进行编译时永远不会知道本地变量的值。

OPTIMIZE FOR。 OPTIMIZE FOR 提示告诉优化器优化查询，正如某个特定值已经指定为变量或参数。执行使用真正的值。记住，OPTIMIZE FOR 提示不会强制重新编译查询。它仅指示 SQL Server 假设变量或参数具有特定值，在这些情形下 SQL Server 确定查询需要优化。如第 8 章的 OPTIMIZE FOR 提示所述，我们在这里不再赘述。

KEEP PLAN。 KEEP PLAN 提示放宽了查询的重新编译阈值，尤其是访问临时表的查询。正如本章前面所述，当表格有至少 6 处进行了更改后，访问临时表的查询可以重新进行编译。如果查询使用 KEEP PLAN 提示，临时表的重新编译阈值也会发生改变，正如永久表一样。

KEEPFIXED PLAN。 KEEPFIXED PLAN 提示限制所有重新编译，因为最优性问题。使用此提示，只有在强制时或者如果底层表的架构发生了更改时，查询才会进行重新编译。

PARAMETERIZATION。 PARAMETERIZATION 提示重写数据库的 PARAMETERIZATION 选项。如果数据库设置为 PARAMETERIZATION FORCED，使用 PARAMETERIZATION 提示的单个查询只有在它们满足严格的条件列表时才可以避免被参数化。另外，如果数据库设置为 PARAMETERIZATION SIMPLE，则单个查询遇到每种情形进行参数化。但是，注意 PARAMETERIZATION 提示仅可以与计划指南结合使用，稍后将介绍计划指南。

USE PLAN。 USE PLAN 提示已在第 8 章介绍过了，作为强制 SQL Server 使用计划的一种方式，可能不想使用其他提示指定计划。指定的计划必须为 XML 格式，并且可以通过使用选项 SET SHOWPLAN_XML ON 获得使用所需计划的查询。因为 USE PLAN 提示包含查询提示的完整 XML 文档，它们在计划指南中使用最好，下一节将介绍。

2. 计划指南的目的

尽管在大多数情况下，建议您允许查询优化器确定每个查询的最佳计划，然而有时查询优化器想不到最佳计划，您可能发现获得合理性能的唯一方式是使用提示。这通常是应用程序的明显更改，在您验证了所需的提示将产生不同的效果时。但是，在有些环境下，无法控制应用程序代码。在实际 SQL 查询嵌入到无法访问的供应商代码，或者是修改供应商代码会违反许可协议或是会使支持保证无效时，可能无法将提示添加到行为不当的查询中。

计划指南（在 SQL Server 2005 中引入）提供了一种解决方案，为您提供向查询添加提示而无需更改查询本身的机制。从本质上来说，计划指南告诉优化器：如果它试图优化具有特定格式的查询时，它就应该为查询添加一个指定的提示。SQL Server 支持 3 种类型的计划指南：SQL、对象和模板。本章稍后将介绍。

计划指南在 SQL Server 的标准版、企业版、评估版和开发人员版中存在。如果分离包含来自受支持版本中的计划指南的数据库，并将此数据库附加到未支持版本（工作组版或 Express 版）中，SQL Server 不会使用任何计划指南。但是包含计划指南信息的元数据仍然可用。

3. 计划指南类型

有 3 种类型的计划指南可以使用 *sp_create_plan_guide* 过程创建。*sp_create_plan_guide* 过程的通用形式如下所示：

```
sp_create_plan_guide 'plan_guide_name', 'statement_text',
   'type_of_plan_guide', 'object_name_or_batch_text',
   'parameter_list', 'hints'
```

我们将介绍每种计划指南类型，然后介绍使用计划指南的机制，以及记录计划指南信息的元数据。

对象计划指南。对象类型的计划指南表明您对与 SQL Server 对象相关的 T-SQL 语句感兴趣，SQL Server 对象可以是存储过程、用户定义函数或数据库（创建计划指南）中的触发器。举一个例子，假设我们拥有一个名为 *Sales.GetOrdersByCountry* 的存储过程，它以国家/地区为参数，在进行一些错误检查和其他验证之后，它返回了一个特定国家/地区的客户的命令行集。进一步假设测试确定了 US 参数值提供了最佳计划。下面是一个计划指南示例，它告诉 SQL Server 只要在 *Sales.GetOrdersByCountry* 过程中找到指定语句，就使用 OPTIMIZE FOR 提示：

```
EXEC sp_create_plan_guide
    @name = N'plan_US_Country',
    @stmt =
       N'SELECT SalesOrderID, OrderDate, h.CustomerID, h.TerritoryID
            FROM Sales.SalesOrderHeader AS h
            INNER JOIN Sales.Customer AS c
               ON h.CustomerID = c.CustomerID
            INNER JOIN Sales.SalesTerritory AS t
               ON c.TerritoryID = t.TerritoryID
            WHERE t.CountryRegionCode = @Country',
@type = N'OBJECT',
@module_or_batch = N'Sales.GetOrdersByCountry',
@params = NULL,
@hints = N'OPTION (OPTIMIZE FOR (@Country = N''US''))';
```

在 *AdventureWorks2008* 数据库中创建该计划后，每次编译 *Sales.GetOrdersByCountry* 过程时，该计划中的语句就会优化，就好像实际参数传入的是字符串“US”一样。过程中的其他语句不会受到该计划的影响，并且如果指定的查询发生在 *Sales.GetOrdersByCountry* 过程之外，该计划指南就不会被调用（随附网站还包含构建 *Sales.GetOrdersByCountry* 过程的脚本，该网站包含本书示例使用的所有代码）。

SQL 计划指南。SQL 类型的计划指南表示您对具体的 SQL 语句（独立语句或具体批处理中的语句）感兴趣。CLR 对象或扩展存储过程发送给 SQL Server 的 T-SQL 语句，或是使用 EXEC(*sql_string*)构造调用的动态 SQL 的 T-SQL 语句，它们都作为批处理在 SQL Server 上处理。要在计划指南中使用它们，其类

型应该设置为SQL。对于独立语句，*sp_create_plan_guide* 的@*module_or_batch* 参数应设置为NULL，这样SQL Server会认为批处理和语句具有相同的值。如果感兴趣的语句位于较大的批处理中，则整个批处理文本需要在@*module_or_batch* 参数中指定。如果为SQL计划指南指定了批处理，则批处理的文本需要与应用程序中的批处理文本完全相同。但规则不像即席查询计划重用那样严格，它们相近，如本章前面所述。确保使用相同的大小写、相同的空格和应用程序使用的其他特点。

下面是一个计划指南示例，它告诉SQL Server在将某个查询用做独立查询时仅使用一个CPU（没有并行）：

```
EXEC sp_create_plan_guide
@name = N'plan_SalesOrderHeader_DOP1',
@stmt = N'SELECT TOP 10 *
          FROM Sales.SalesOrderHeader
          ORDER BY OrderDate DESC',
@type = N'SQL',
@module_or_batch = NULL,
@params = NULL,
@hints = N'OPTION (MAXDOP 1)';
```

在 *AdventureWorks2008* 数据库中创建该计划后，每次在批处理中遇到指定语句时，都会使用单CPU创建一个计划。如果指定的查询是较大批处理的一部分，就不会调用该计划指南。

模板计划指南。 *模板*类型的计划指南只能使用PARAMETERIZATION FORCED或PARAMETERIZATION SIMPLE提示覆盖PARAMETERIZATION数据库设置。模板指南较难使用，因为您必须使SQL Server构造查询模板的方式与对其进行参数化的方式相同。这并不难，因为SQL Server提供了一个名为 *sp_get_query_template* 的特殊过程，但是要使用模板指南，需要执行几个必备步骤。如果看了前面介绍的两个计划指南示例，就会看到OBJECT和SQL计划指南都有名为@*params* 的参数。仅使用模板计划指南指定@*params* 的值。

要查看使用模板指南和强制参数化的示例，首先要清除过程缓存，然后在 *AdventureWorks2008* 数据库中执行这两个查询：

```
DBCC FREEPROCCACHE;
GO
SELECT * FROM AdventureWorks2008.Sales.SalesOrderHeader AS h
INNER JOIN AdventureWorks2008.Sales.SalesOrderDetail AS d
  ON h.SalesOrderID = d.SalesOrderID
WHERE h.SalesOrderID = 45639;
GO
SELECT * FROM AdventureWorks2008.Sales.SalesOrderHeader AS h
INNER JOIN AdventureWorks2008.Sales.SalesOrderDetail AS d
  ON h.SalesOrderID = d.SalesOrderID
WHERE h.SalesOrderID = 45640;
```

这两个查询非常相似，并且两个查询的计划是相同的，但是因为查询太复杂了，所以SQL Server不会对它们进行自动参数化。执行了两个查询后，查看计划缓存，将会仅看到即席查询。如果已经创建了前面介绍的 *sp_cacheobjects* 视图，则可以使用该视图，否则使用 *sys.syscacheobjects* 视图替换 *sp_cacheobjects* 视图：

```
SELECT objtype, dbid, usecounts, sql
FROM sp_cacheobjects
WHERE cacheobjtype = 'Compiled Plan';
```

要创建强制此类型语句参数化的计划指南，首先需要调用 *sp_get_query_template* 过程，并将两个变量作为输出参数传入。一个参数拥有查询的参数化版本，另一个参数拥有参数列表和参数数据类型。然后以下代码 SELECT 两个输出参数，以便看到其内容。当然，也可以从自己的代码中删除此 SELECT。最后调用 *sp_create_plan_guide* 过程，这将指示优化器在看到与此特定模板匹配的查询时使用 PARAMETERIZATION FORCED。换言之，当查询以上述方式参数化时，它将使用已经缓存的相同计划：

```
DECLARE @sample_statement nvarchar(max);
DECLARE @paramlist nvarchar(max);
EXEC sp_get_query_template
  N'SELECT * FROM AdventureWorks2008.Sales.SalesOrderHeader AS h
    INNER JOIN AdventureWorks2008.Sales.SalesOrderDetail AS d
       ON h.SalesOrderID = d.SalesOrderID
    WHERE h.SalesOrderID = 45639;',
  @sample_statement OUTPUT,
  @paramlist OUTPUT
SELECT @paramlist as parameters, @sample_statement as statement
EXEC sp_create_plan_guide @name = N'Template_Plan',
  @stmt = @sample_statement,
  @type = N'TEMPLATE',
  @module_or_batch = NULL,
  @params = @paramlist,
  @hints = N'OPTION(PARAMETERIZATION FORCED)';
```

创建计划指南后，运行与前面相同的两个语句，然后检查计划缓存：

```
DBCC FREEPROCCACHE;
GO
SELECT * FROM AdventureWorks2008.Sales.SalesOrderHeader AS h
INNER JOIN AdventureWorks2008.Sales.SalesOrderDetail AS d
   ON h.SalesOrderID = d.SalesOrderID
WHERE h.SalesOrderID = 45639;
GO
SELECT * FROM AdventureWorks2008.Sales.SalesOrderHeader AS h
INNER JOIN AdventureWorks2008.Sales.SalesOrderDetail AS d
   ON h.SalesOrderID = d.SalesOrderID
WHERE h.SalesOrderID = 45640;
GO
SELECT objtype, dbid, usecounts, sql
FROM sp_cacheobjects
WHERE cacheobjtype = 'Compiled Plan';
```

现在应该看到一个具有以下参数化形式的编译计划：

```
(@0 int)select * from AdventureWorks2008.Sales.SalesOrderHeader as h
   inner join AdventureWorks2008.Sales.SalesOrderDetail as d
   on h.SalesOrderID = d.SalesOrderID
   where h.SalesOrderID = @0
```

4. 管理计划指南

除了 *sp_create_plan_guide* 和 *sp_get_query_template* 过程外，其他与计划指南一起使用的基本过程是 *sp_control_plan_guide*。该过程允许您使用以下基本语法删除、禁用或启用计划指南：

```
sp_control_plan_guide '<control_option>' [, '<plan_guide_name>']
```

有 6 个可能的 *control_option* 值：DISABLE、DISABLE ALL、ENABLE、ENABLE ALL、DROP 和 DROP ALL。*plan_guide_name* 参数是可选的，因为对于其他所有 *control_option* 值，都不会提供 *plan_guide_name* 值。计划指南是相对于本地特定数据库而言的。此外，计划指南的工作方式在某种程度上类似于架构绑定视图。数据库中任意对象计划指南所设计的存储过程、触发器和函数都无法改变或删除。所以对于我们的示例对象计划指南，只要计划指南存在，*AdventureWorks2008.Sales.GetOrdersByCountry* 过程就无法改变或删除。无论是禁用还是启用了计划指南都是这样，并且除非使用 *sp_control_plan_guide* 删除了所有引用这些对象的计划指南，否则仍然是这样。

在具体数据库中包含计划指南信息的元数据视图是 *sys.plan_guide*。该视图包含 *sp_create_plan_guide* 过程中提供的所有信息，以及其他信息，如创建日期和每个计划指南的最后修改日期。使用该视图中的信息，如果需要，可以手动重新构造计划指南定义。此外，Management Studio 允许您从 Object Explorer 树编写计划指南定义。

5. 计划指南考虑因素

如果 SQL Server 想要确定要使用的恰当的计划指南，计划指南中的语句文本必须与正在编译的查询匹配。这必须是完全字符匹配，包括大小写、空格和注释，正如当 SQL Server 确定是否可以重用即席查询计划时一样，如本章前面所述。如果关闭了语句文本，但是不是完全匹配，这就会导致故障很难排除。当与 SQL 模板匹配时，无论定义是否包含该语句所属的批处理，SQL Server 会为批处理定义留有更多余地。具体地说，关键字大小写、空格和注释都会忽略。

要确保计划指南使用与应用程序提交的文本完全相同的文本，可以使用 SQL Server Profiler 运行跟踪，并捕获 SQL:BathCompleted 和 PRC:Completed 事件。在 Profiler 输出的顶部窗口中显示了相关批处理（想要为计划指南创建的批处理）后，可以右键单击事件并选择“提取事件数据”，将批处理的 SQL 文本存储到文本文件中。仅从 Profiler 的底部窗口复制和粘贴文本是不够的，因为那里的输出可以引入其他换行符。

要验证使用了计划指南，可以查看查询的 XML 计划。如果直接运行查询，可以使用选项 SET SHOWpLAN_XML ON，或者通过跟踪捕获 XML 显示计划。XML 计划特定项，这两个表示查询使用了计划指南。这些项是 *PlanGuideDB 和 PlanGuideName*。如果计划指南是模板计划指南，则 XML 计划还有 *TemplatePlanGuideDB* 和 *TemplatePlanGuideName* 项。

当提交查询进行处理时，如果数据库中有任何计划指南，SQL Server 首先检查语句是否与 SQL 计划指南或对象计划指南匹配。查询字符串被分隔成多个字符串，以使在数据库的现有计划指南中查找任意匹配的字符串变得更容易。如果没有找到任何匹配的 SQL 或对象计划指南，则 SQL Server 会检查模板计划指南。如果找到 TEMPLATE 指南，它会尝试将生成的参数化查询与 SQL 计划指南相匹配。这使您有可能使用强制参数化向查询应用其他提示。图 9-3（从 *SQL Server 联机丛书*复制而来）展示了 SQL Server 用于检查适用的计划指南的流程。

主要步骤如下所示，它按照上述流程图从左到右、从上至下的顺序进行，根据计划指南及其提示修改语句。

（1）对于批处理中的特定语句，SQL Server 尝试匹配语句和基于 SQL 的计划指南，该计划指南的 *@module_or_batch* 参数与进入的批处理文本（包括任意常量文字值）匹配，并且该计划指南的 *@stmt* 参数还与批处理中的语句匹配。如果存在此种类型的计划指南，则匹配成功，语句文本被修改为包含计划指南的查询提示。然后使用指定的提示编译语句。

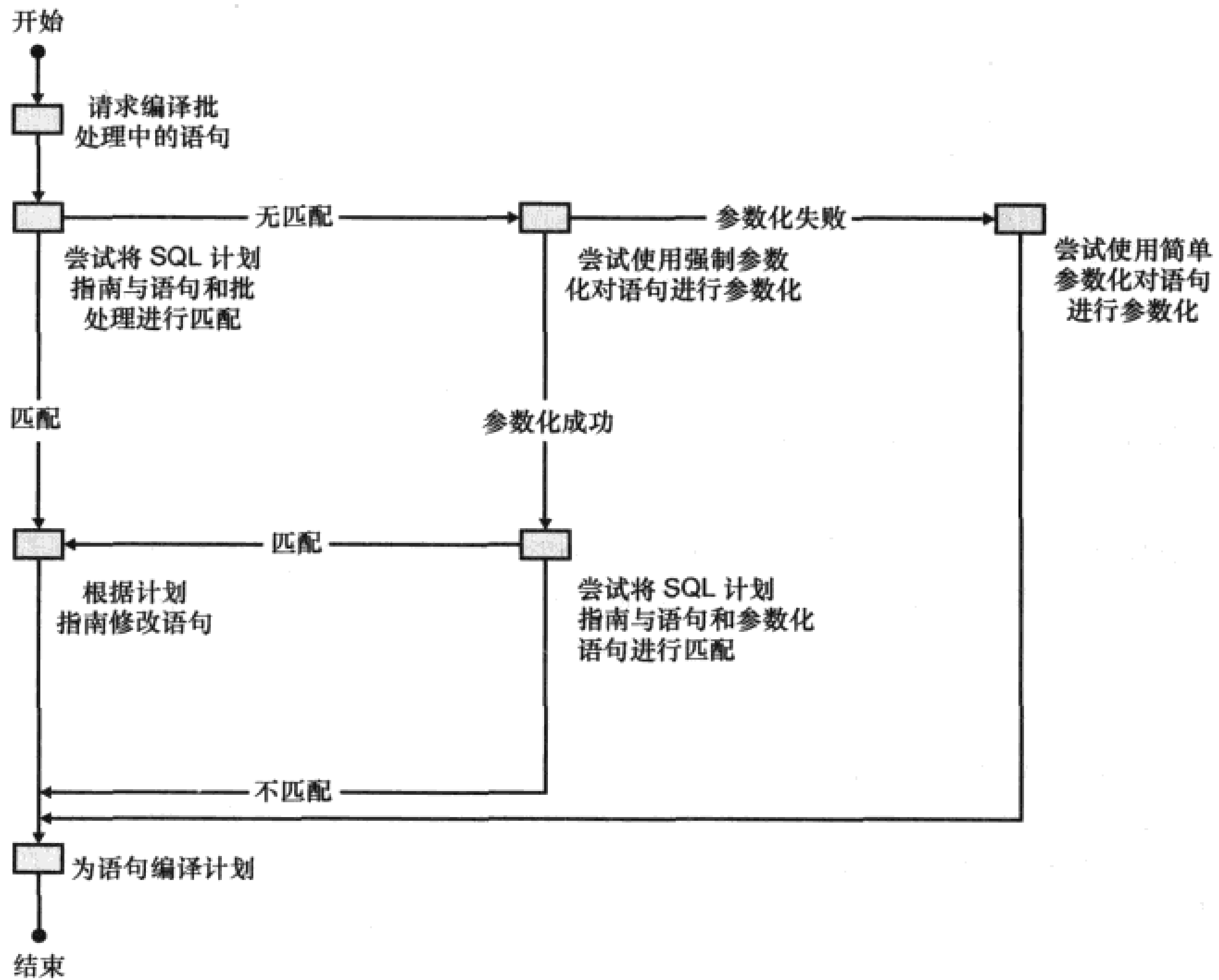


图 9-3 检查适用的计划指南

（2）如果计划指南与步骤（1）中的语句不匹配，SQL Server 尝试使用强制参数化对语句进行参数化。在该步骤中，由于以下任意一种原因，参数化都可能失败。

- 语句已经被参数化或它包含本地变量。
- 应用了 PARAMETERIZATION SIMPLE 数据库 SET 选项（默认设置），并且没有应用于语句并指定了 PARAMETERIZATION FORCED 查询提示的 TEMPLATE 类型的计划指南。
- 存在应用于语句并指定了 PARAMETERIZATION SIMPLE 查询提示的 TEMPLATE 类型的计划指南。

请看以下示例，该示例包含 *AdventureWorks2008* 数据库中 *Sales.SalesOrderDetail* 表格的 *SpecialOfferID* 列的数据分发。有 12 个不同的 *SpecialOfferID* 值，并且大多数值最多会出现几百次（*Sales.SalesOrderDetail* 的 121 317 行内容），如以下脚本和输出所示：

```
USE AdventureWorks2008
GO
SELECT SpecialOfferID, COUNT(*) as Total
FROM Sales.SalesOrderDetail
GROUP BY SpecialOfferID;

RESULTS:
SpecialOfferID   Total
--------------   -----------
1                115884
```

```
2                 3428
3                 606
4                 80
5                 2
7                 137
8                 98
9                 61
11                84
13                524
14                244
16                169
```

表格有 1238 页，对于大多数值，*SpecialOfferID* 上的非聚集索引很有用，下面是构建使用的代码：

```
CREATE INDEX Detail_SpecialOfferIndex ON Sales.SalesOrderDetail(SpecialOfferID);
```

假设很少有查询实际上会搜索值为 1 或值 2 的 *SpecialOfferID*，并且 99%的时间查询都查找不热门的值。我们希望查询优化器自动参数化访问 *Sales.SalesOrderDetail* 表格的查询，为 *SpecialOfferID* 指定一个特定值。所以我们创建一个模板计划指南以自动化此类型的查询：

```
SELECT * FROM Sales.SalesOrderDetail WHERE SpecialOfferID = 4;
```

但是，我们想要确保确定计划的初始参数不是可能使用聚集索引扫描的一个值，也就是说不是值 1 或值 2。所以我们使用 *sp_get_query_template* 过程生成的自动参数化查询，首先使用它创建一个模板计划指南，然后创建一个具有 OPTIMIZE FOR 提示的 SQL 计划指南。该提示强制 SQL Server 在每次重新优化查询时假设值为 4：

```
USE AdventureWorks2008;
-- Get plan template and create plan Guide
DECLARE @stmt nvarchar(max);
DECLARE @params nvarchar(max);
EXEC sp_get_query_template
    N'SELECT * FROM Sales.SalesOrderDetail WHERE SpecialOfferID = 4',
    @stmt OUTPUT,
    @params OUTPUT
--SELECT @stmt as statement -- show the value when debugging
--SELECT @params as parameters -- show the value when debugging

EXEC sp_create_plan_guide N'Template_Plan_for SpecialOfferID',
    @stmt,
    N'TEMPLATE',
    NULL,
    @params,
    N'OPTION (PARAMETERIZATION FORCED)';

EXEC sp_create_plan_guide
    @name = N'Force_Value_for_Prepared_Plan',
    @stmt = @stmt,
    @type = N'SQL',
    @module_or_batch = NULL,
    @params = @params,
    @hints = N'OPTION (OPTIMIZE FOR (@0 = 4))';
GO
```

通过运行如下所示的测试，可以验证正在为使用 DpecialOfferID 的非聚集索引的值进行自动参数化

和优化的计划：

```
DBCC FREEPROCCACHE;
SET STATISTICS IO ON;
SELECT * FROM Sales.SalesOrderDetail
WHERE SpecialOfferID = 3;
GO
SELECT * FROM Sales.SalesOrderDetail
WHERE SpecialOfferID = 4;
GO
SELECT * FROM Sales.SalesOrderDetail
WHERE SpecialOfferID = 5;
GO
```

您应该注意到，在 STATISTICS IO 输出中，每次执行都使用不同的读取数，因为它通过非聚集索引查找到不同的行数。还可以通过检查 STATISTICS XML 输出验证 SQL Server 正在使用编译计划。如果将选项设置为 ON，并运行查询以查找值 5，则应在 XML 文档中看到一个类似这样的节点：

```
<ParameterList>
<ColumnReference Column="@0" ParameterCompiledValue="(4)"
       ParameterRuntimeValue="(5)" />
</ParameterList>
```

计划指南目的不在加快查询编译速度。SQL Server 不仅会首先确定是否存在与正在编译的查询完全匹配的计划指南，还会确定计划指南执行的计划是查询优化器将生成的计划。要了解执行的计划是否有效，查询优化器需浏览大多数优化流程。计划指南的好处是缩短了查询的执行时间，在这些查询中查询优化器不会生成最优计划。

SQL Server 2008 中主要计划指南的改进与使计划指南变得更有用相关。SQL Server 2008 包含 SMO 和 Management Studio 支持，包括将计划指南脚本编写为数据库脚本的一部分。编写了计划指南的脚本之后，它可以复制到运行相同查询的其他 SQL Server 实例中。

6. 计划指南验证

SQL Server 2005 计划指南实现的一个限制是，它可能会更改表的物理设计（例如删除索引），这会使计划指南无效，且会使使用计划指南的任意查询在执行时失败。当更改表格设计会损害计划指南时，SQL Server 2008 可以检测到该情形。现在可以通过跟踪事件（而无需计划指南）重新编译查询并通知管理员。此外，还有一个用于验证计划指南的新系统函数。此函数用于检测损害现有计划指南的物理数据库设计更改，并支持您在它损害系统之前回滚损害性的事务。

要验证系统中的所有现有计划指南，可使用 *sys.fn_validate_plan_guide* 函数：

```
SELECT * FROM sys.plan_guides pg
CROSS APPLY
(SELECT * FROM sys.fn_validate_plan_guide(pg.plan_guide_id)) v;
```

该函数没有为有效计划指南返回任何内容。当指南将生成一个错误时，它会返回一行内容。所以可以将这一函数合并进系统中的任意架构更改中：

```
BEGIN TRANSACTION;
DROP INDEX t2.myindex;
```

```
IF EXISTS(
SELECT * FROM sys.plan_guides pg
CROSS APPLY
  (SELECT * FROM sys.fn_validate_plan_guide(pg.plan_guide_id)) v
)
ROLLBACK TRANSACTION
ELSE
COMMIT TRANSACTION;
```

7. 从计划缓存冻结计划

SQL Server 2008 增加了一个新存储过程，支持您从已缓存的计划自动创建一个计划指南。过程 *sp_create_plan_guide_from_handle* 需要一个 *plan_handle* 和一个计划指南名称作为参数，并使用存储在缓存中的执行计划创建 *plan_handle* 值的计划指南。该功能称为*计划冻结*，因为它确保在每次执行相关查询时重用执行良好的计划。假设我们发现，刚刚为以下查询执行的计划表现很好，我们想要确保后续执行继续使用该计划：

```
SELECT City, StateProvinceID, PostalCode FROM Person.Address ORDER BY PostalCode DESC;
```

通过搜索与查询匹配的文本值，可以在缓存中找到相应的计划：

```
SELECT plan_handle
FROM sys.dm_exec_query_stats AS qs CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) AS st
WHERE st.text LIKE N'SELECT City,%';
```

拥有了 *plan_handle* 之后，可以将其作为参数传给 *sp_create_plan_guide_from_handle* 过程的参数，如下所示：

```
EXEC sp_create_plan_guide_from_handle
     @name = N'Guide1_from_XML_showplan',
     @plan_handle = 0x06000600F19B1E1FC0A14C0A0000000000000000000000000
```

在几种情形中计划指南和计划冻结可能很有用。

- 服务器升级之后，可以使用计划指南为计划回归提供一个解决方案。
- 不支持对执行良好的系统中的关键计划进行更改。
- 通过冻结一个好计划（假设曾使用的好计划）对问题查询进行故障排除。
- ISV 可以创建其应用程序随附提供的已知好计划。
- 可以优化测试系统，然后将计划指南移植到生产系统中。
- 可以在复制的数据库中包含计划指南。

9.8 小结

对于所有缓存机制，重用缓存计划可以避免重新编译和优化。这会节省编译时间，但这意味着无论传入的是什么参数值，都会使用相同的计划。如果指定参数值的最优计划与缓存计划的不同，则不会获得最优执行时间。基于此原因，SQL Server 在自动参数化方面很保守。当应用程序使用 *sp_executesql*、prepare 和 execute 方法，或者是存储过程时，应用程序开发人员负责确定应该参数化的内容。仅参数化其值范围不会明显影响优化选择的常量。

本章介绍了查询优化器生成的计划缓存和重用。SQL Server 不仅可以缓存和重用存储过程的计划，还可以缓存和重用即席和自动参数化查询的计划。因为生成查询计划代价很大，所以它有助于了解如何和为什么重用查询计划，以及何时必须重新生成计划。了解缓存和重用计划的工作方式有助于确定何时使用缓存计划是正确的选择，以及何时需要 SQL Server 生成新计划，以赋予查询和应用程序最佳性能。

第 10 章 事务和并发性

Kalen Delaney

并发性可以定义为多个进程在相同时间访问或更改共享数据的能力。在互不干扰的情况下可以激活的并发用户进程数越多，数据库系统的并发性就越强。

当正在更改数据的进程阻止其他进程读取该数据时，或者当读取数据的进程阻止其他进程更改该数据时，并发性减弱。我们使用术语*读取*或*访问*来描述在数据上使用 *SELECT* 语句的效果。当多个进程试图同时更改相同数据，且无法在不牺牲数据一致性的前提下都能成功时，并发性也会受到影响。我们使用术语*修改*、*更改*或*写入*来描述在数据上使用 *INSERT*、*UPDATE*、*MERGE* 或 *DELETE* 语句的效果（注意，*MERGE* 是 SQL Server 2008 中的一个新数据修改语句，可以把它看做是 *INSERT*、*UPDATE* 和 *DELETE* 语句的组合）。

一般来说，数据库系统管理并发数据访问可以采用两种方法：乐观和悲观。Microsoft SQL Server 2008 对这两种方法都支持。悲观并发是 SQL Server 2005 之前可用的唯一并发模型。到了 SQL Server 2005，通过使用两个数据库选项和一个叫做 TRANSACTION ISOLATION LEVEL（事务隔离级别）的 SET 选项来指定使用哪种模型。

我们首先来看两种模型的基本区别，然后再来看 SQL Server 2008 中 5 种可能的隔离级别，以及 SQL Server 在内部是如何使用每种模型来控制并发访问的。我们还要介绍如何控制隔离级别，并展示 SQL Server 在操作的元数据。

10.1 并发模型

在两种并发模型中，两个进程试图在相同时间修改相同数据时，可能会出现冲突。两种模型之间的区别在于，冲突是在出现之前被避免还是在出现之后用某种方式进行处理。

10.1.1 悲观并发

对于悲观并发，SQL Server 默认的行为是获取锁来阻塞对另一个进程正在使用的数据的访问。悲观并发假设系统中有足够的数据修改操作，因而任何给定的读取操作都可能受到另一个用户的数据修改操作的影响。换言之，系统表现得很悲观，总是假设会出现冲突。悲观并发通过获得正在被读取数据上的锁，使其他进程无法修改该数据而避免冲突。它也获得正在被修改数据上的锁，使其他进程无法访问该数据，无论是读取还是修改。换言之，在悲观并发环境中，读取者阻塞写入者，写入者也阻塞读取者。

10.1.2 乐观并发

乐观并发假设系统中有足够少的冲突数据修改操作，因而任何单个事务都不太可能修改另一个事务正在修改的数据。乐观并发的默认行为是使用行版本控制来允许数据读取者看到修改之前的数据状态。数据行较老的

版本被保存，所以读取数据的进程可以看到进程开始读取时的数据，不会受到对该数据正在做出的任何更改的影响。修改数据的进程不会受到读取数据进程的影响，因为读取者访问的是所保存的数据行版本。换言之，读取者不阻塞写入者，写入者也不阻塞读取者。但是，写入者可以而且也将阻塞写入者，这也是导致冲突的原因所在。SQL Server 在冲突出现时产生一个错误消息，但是由应用程序来负责响应该错误。

10.2 事务处理

不管使用哪种并发模型，了解事务都是至关重要的。事务是 SQL Server 中的基本工作单元。通常，它由几个读取和更新数据库的 SQL 命令组成，但是这里的更新不被看做是最终的，直到发出一个 *COMMIT* 命令为止（至少对于显式的事务是如此）。一般来说，我说到修改操作或读取操作时，是在说执行数据修改或数据读取的事务，不一定是单个 SQL 语句。我说到写入者会阻塞读取者时，是指只要执行写入操作的事务是激活的，其他进程就无法读取所修改的数据。

事务的概念是理解并发控制的基础。从编程角度解释事务控制的机制超出了本书范畴，但是我们要来讨论一些基本的事务属性。我们还将详细介绍事务隔离级别，因为隔离级别将直接影响 SQL Server 如何管理事务中正在被访问的数据。

隐式事务就是任何单个的 *INSERT*、*UPDATE*、*DELETE* 或 *MERGE* 语句（您也可以将 *SELECT* 语句看做是隐式事务，尽管在 *SELECT* 语句被处理时 SQL Server 不写入到日志）。不管影响到多少行，语句必须展示事务的所有 ACID 属性，下一小节将介绍 ACID 属性。显式事务开头标记有 *BEGIN TRAN* 语句，结尾标记有 *COMMIT TRAN* 或 *ROLLBACK TRAN* 语句。本章给出的大部分例子使用显式事务，因为这是展示 SQL Server 在事务中的状态的唯一方式。例如，只有在事务处理过程中，才持有很多类型的锁。我们可以开始一个事务，执行一些操作，查看元数据，看持有哪些锁，然后结束事务。事务结束时，锁被释放，我们再也看不到这些锁了。

10.2.1 ACID 属性

事务处理保证 SQL Server 数据库的一致性和可恢复性。它确保所有的事务都作为单个工作单元执行——即使在出现硬件或常规系统故障时亦是如此。此类事务被称为具有 ACID 属性，其中 ACID 代表 *atomicity（原子性）*、*consistency（一致性）*、*isolation（隔离性）和 durability（持久性）*。除了保证显式的多语句事务具有 ADIC 属性之外，SQL Server 还保证隐式事务也具有 ACID 属性。

下面是一个例子，以伪代码的形式展示了一个显式 ACID 事务：

```
BEGIN TRANSACTION DEBIT_CREDIT
Debit savings account $1000
Credit checking account $1000
COMMIT TRANSACTION DEBIT_CREDIT
```

下面我们详细介绍每一种 ACID 属性。

1. 原子性

SQL Server 保证其事务的原子性。原子性意味着每个事务被作为整体处理或者不被处理——要么全部提交，要么全部撤销。如果一个事务被提交，那么它所有的效果都保留。如果事务被撤销，那么它所有的效果都还原。在前面的 DEBIT_CREDIT 例子中，如果储蓄存款账户借记金额反映到数据库中，但是

支票存款账户贷记金额没有反映到数据库中，那么资金就会彻底从数据库消失——也就是说，资金从储蓄存款账户减少了，却没有被增加到支票存款账户。如果发生相反的情况（贷记了支票存款账户却没有借记储蓄存款账户），那么客户的支票存款账户余额神秘地增多了，却没有发生相应的客户现金储蓄或账户资金转移。由于 SQL Server 的原子性特性，所以借贷必须都完成，否则就都不完成。

2. 一致性

一致性属性确保事务不会允许系统达到不正确的逻辑状态——数据必须总是逻辑上正确。即使出现系统故障，也遵循所有的约束和规则。在 DEBIT_CREDIT 例子中，逻辑规则是，钱不会变多，也不会变少，即每个条目都有对应的平衡条目（一致性隐含在原子性、隔离性和持久性中，在大多数情形下与原子性、隔离性及持久性是冗余的）。

3. 隔离性

隔离性将并发事务与其他未完成事务的更新分离开来。在 DEBIT_CREDIT 例子中，另一个事务无法看到这个正在执行的事务中所进行的工作。例如，如果另一个事务在借记之后读取储蓄存款账户的余额，而后 DEBIT_CREDIT 事务又被撤销了，那么这个事务读取到的则是从来没有在逻辑上存在过的余额。

SQL Server 自动地在事务之间实现隔离。它锁定数据或者创建行版本，以允许多个并发用户在防止出现负面效果的同时处理数据。负面效果是指，如果用户序列化他们的请求（也就是说将请求进行排队，一次响应一个），那么结果会被扭曲，不同于预期的结果。这种可序列化特性是 SQL Server 支持的隔离级别之一。SQL Server 支持多种隔离级别，您可以在锁定多少数据、锁定多长时间及是否允许用户访问行数据的优先版本之间进行权衡。该权衡就是所谓的并发性与一致性之间的平衡。

4. 持久性

在事务提交之后，SQL Server 的持久性属性确保事务的效果持久存在，即使出现系统故障亦是如此。如果在事务进行过程中出现系统故障，那么事务被彻底取消，不对数据产生任何部分效果。例如，如果在事务提交之前，在事务执行当中出现断电，那么整个事务在系统重启时回滚。如果刚好在事务提交确认信息发送到调用程序之后断电，那么保证事务结果已存在于数据库中。预写式日志和 SQL Server 启动的恢复阶段进行的事务的原子回滚及前滚，确保了持久性。

10.2.2 事务依赖性

除了支持所有 4 个 ACID 属性之外，事务可能还会展示几个其他行为。一些人将这些行为称之为“依赖性问题”或“一致性问题”，但是我并不一定把它们看做是问题。它们只是可能的行为，除丢失更新外（因为这种行为从来不是人们所想要的），您可以决定自己想要允许哪些行为，想要避免哪些行为。所选的隔离级别决定了哪些行为是被允许的。

1. 丢失更新

当两个进程读取相同的数据，并且都操纵数据、更改它的值，还试图将原始数据更新到新值时，会出现丢失更新。第二个进程可能会彻底覆盖第一个进程所做的更新。例如，假设收料室的两个店员正在收零件，并将新收到的零件增添到库存数据库中。店员 A 和店员 B 都接收到运来的部件。他们都检查当前库存，并看到存货中当前还有 25 个部件。店员 A 的一车货是 50 个部件，所以他将 50 与 25 相加，并

将当前值更新为75。店员B的一车货是20个部件，所以他将20与他最初读取到的25相加，并将当前值更新为45，彻底覆盖店员A添加的50个新部件。店员A的更新丢失。

丢失更新只是这里描述的您在所有情形下都想要避免的行为之一。

2. 脏读

脏读出现在进程读取未提交数据时。如果一个进程更改了数据，但是还没有提交更改，另一个进程此时读取该数据就是在不一致的状态下读取数据。例如，假设店员A已将原来的25个部件更新为75个，在他提交之前，一个销售人员看到当前值是75并答应第二天给客户发60个部件。如果店员A后来发现部件有缺陷并发回制造商，那么销售人员就进行了一次脏读，并基于未提交数据采取了行动。

默认情况下，脏读是不允许的。记住，更新数据的进程无法控制另一个进程是否可以在第一个进程提交之前读取它的数据。由读取数据的进程来决定它是否要读取不能保证已提交的数据。

3. 不可重复读

如果进程在同一事务两次独立的读操作中读取相同的数据却取回不同的值，那么这样的读是不可重复的。在第一个进程进行的两次读操作之间，另一个进程更改数据时就会发生此情况。在收料室例子中，假设一个经理来检查当前库存。她走向每个店员，询问当天收到的部件总数，并将这个数加到她的计算器中。问完之后，她想要复查结果，又回到第一个店员那里。如果店员A在经理两次询问他之间又收到了部件，总数就变了，这样的读是不可重复的。不可重复读也叫做*不一致分析*。

4. 幻像

幻像出现在一组中的成员人数发生改变时。仅当涉及具有谓词（如WHERE count_of_widgets < 10）的查询时才会出现幻像。如果同一事务中使用相同谓词的两个*SELECT*操作返回不同的行数，就出现了幻像。例如，假设经理还在检查库存。这次她走到收料室，记录拥有部件少于10个的店员。记录完后，她再返回去挨个通知这些人。但是，如果她第一次在收料室记录时，一个拥有少于10个部件的店员休假回来了，但是已经错过了经理的检查，那么该店员就没有出现在经理的清单中，即使他满足谓词中的条件。这个额外的店员（或行）就被认为是一个幻像。

事务的行为依赖于隔离级别。正如前面所提到的，可以通过使用命令SET TRANSACTION ISOLATION LEVEL <isolation_level>设置一个适当的隔离级别，来决定允许前面描述的哪些行为。并发模型（乐观或悲观）确定隔离级别是如何实现的——或者，更具体地来说，就是SQL Server如何保证不会出现您不想要的行为。

10.2.3 隔离级别

SQL Server 2008支持5种隔离级别，用于控制读操作的行为。其中3种只对悲观并发可用，1种只对乐观并发可用，还有1种对两者都可用。我们现在来看这些级别，但是对隔离级别的完全理解还需要了解锁定和行版本控制。在我们的隔离级别描述中，提到了支持该级别的锁或行版本，锁定和行版本控制将在本章后面详细讨论。

1. 未提交读

在未提交读（未提交读）隔离级别中，除了丢失更新之外，前面描述的所有行为都是可能的。查询

可以读取未提交数据，而且不可重复读和幻像都是可能的。未提交读隔离级别通过允许读操作不利用任何锁而实现，因为SQL Server不试图获得锁，它就不会被其他进程获得的有冲突锁阻塞了。您的进程可以读取另外一个进程已经修改但是还未提交的数据。

除了读取还未提交的单个值之外，未提交读隔离级别还会引起其他不想要的行为。当使用该隔离级别并扫描整个表时，SQL Server可以决定进行分配顺序扫描（按页码顺序），而不是进行逻辑顺序扫描（这会遵循页指示器）。如果存在其他进程更改数据和将行移动到表中新位置的并发操作，那么分配顺序扫描可能会两次读取到同一行。当您读取了一个更新之前的行，而后更新将该行移动到比扫描遇到的页码更高的页时，就会发生此种情况。此外，在未提交读隔离级别下执行分配顺序扫描会导致彻底丢失一个行。当较高页码上一个还未读取的行被更新并移动到已经读取过的较低页码时，就会发生这种情况。

尽管该情形通常不是理想的选择，但是利用未提交读不会因为等待锁而被阻塞，并且读操作不获得任何锁，因而不会影响其他正在读取或写入数据的进程。

使用未提交读时，为了使系统实现高并发性，用户相互之间不会被锁定，放弃了保证数据的强一致性。那么何时应该选用未提交读呢？显然，不想将它用于财务事务，在这样的事务中，每个数字都必须平衡。但是它可能适合于某种决策支持系统，比如说当您了解销售趋势时，因为这不需要完全精确，并且较高的并发性使得它是值得的。未提交读对过于阻塞活动的问题，是一个悲观的解决方案，因为它只忽略锁，并不提供事务一致性。

2. 已提交读

SQL Server 2008支持已提交读（Read Committed）隔离级别的两个变种，这是默认隔离级别。该隔离级别可以是乐观的也可以是悲观的，具体取决于数据库设置READ_COMMITTED_SNAPSHOT。由于数据库选项的默认值是关闭的，所以该隔离级别的默认值使用悲观并发控制。除非特意指出，否则本书中提到已提交读隔离级别时，都是指该隔离级别的两个变种。我们将悲观实现称为已提交读（锁定），将乐观实现称之为已提交读（快照）。

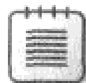
已提交读隔离级别确保操作从来不会读取另一个应用程序已经更改但是还未提交的数据（这就是说，从来不会读取逻辑上不存在的数据）。利用已提交读（锁定），如果另一个事务在更新数据并在数据行上具有独占锁，那么您的事务必须等待这些锁被释放，然后才能使用该数据（不管您是读取还是修改）。此外，您的事务必须在已访问的数据上（至少）放置共享锁，这意味着该数据不能被其他事务使用。共享锁不阻止其他事务读取数据，但是让它们更新数据要等待。默认情况下，共享锁可在数据处理完成后被释放——不必持有到事务结束，或者甚至持有到语句结束（这就是说，如果获得了共享的行锁，那么每个行锁可以在行一处理完就被释放，即使语句可能还需要处理很多的行）。

已提交读（快照）也确保操作从来不读取未提交数据，但是不强迫其他进程等待。在已提交读（快照）中，每次一个行被更新，SQL Server都产生已更改行的一个版本，带有它以前已提交的值。正在更改的数据仍然被锁定，但是其他进程可以看到更新操作开始之前数据的以前版本。

3. 可重复读

可重复读（Repeatable Read）是一种悲观隔离级别。它通过确保事务再次访问数据或者再次发出查询时数据不会改变来增强已提交读的属性。换言之，在一个事务中两次发出相同的查询不会搜出另一个用户的事务对数据值做出的任何更改，因为不会由其他事务做出任何更改。但是，可重复读隔离级别确实允许出现幻像行。

在某些情况下，需要防止不可重复读。但是没有免费的午餐。这重额外保护的成本是，事务中的所有共享锁必须一直持有到事务完成（*COMMIT* 或 *ROLLBACK*）。(独占锁必须总是一直持有到事务结束，不管隔离级别或并发模型如何，所以事务在必要时可以回滚。如果锁很快被释放，就不可能撤销工作，因为其他并发事务可能使用了相同的数据并更改了数据值）只要您的事务是打开的，任何其他用户都不能修改您的事务所访问的数据。显然，这会大大降低并发性和性能。如果事务不是只持续很短的时间，或者应用程序没有编写为了解之类潜在的锁争用问题，那么在进程等待锁被释放时，SQL Server 可能会停止响应。



注意:

通过使用会话选项 LOCK_TIMEOUT，可以控制 SQL Server 在锁被释放之前等待多长时间。这是一个 SET 选项，所以只能针对单个会话对行为进行控制。无法作为一个整体为 SQL Server 设置 LOCK_TIMEOUT 值。可在 *SQL Server 联机丛书*中了解 LOCK_TIMEOUT。

4. 快照

快照隔离级别（有时称为 SI）是一种乐观的隔离级别。跟已提交读（快照）一样，如果已提交数据的当前版本被锁定，那么它允许进程读取较老的版本。快照和已提交读（快照）之间的区别在于，较老版本到底有多老（我们将在本章后面的“行版本控制”一节中了解详细信息)。尽管快照隔离级别阻止的行为与可序列化阻止的行为一样，但是快照并不真正是一种可序列化隔离级别。利用快照隔离级别，可以让两个事务同步执行，得出任何串行执行不可能得到的结果。表 10-1 显示的例子中有两个同步的事务。如果它们并行运行，最终会交换 *pubs* 数据库 *titles* 表中两本书的价格。但是串行执行最终不会交换值，不管是先运行 Transaction 1 再运行 Transaction 2，还是先运行 Transaction 2 再运行 Transaction 1。两种串行顺序最终的结果是两本书的价格相同。

表 10-1　快照隔离级别中的两个不能串行运行的同步事务

时　间	Transaction 1	Transaction 2
1	USE pubs SET TRANSACTION ISOLATION LEVEL SNAPSHOT DECLARE @price money BEGIN TRAN	USE pubs SET TRANSACTION ISOLATION LEVEL SNAPSHOT DECLARE @price money BEGIN TRAN
2	SELECT @price = price FROM titles WHERE title_id = 'BU1032'	SELECT @price = price FROM titles WHERE title_id = 'PS7777'
3	UPDATE titles SET price = @price WHERE title_id = 'PS7777'	UPDATE titles SET price = @price WHERE title_id = 'BU1032'
4	COMMIT TRAN	COMMIT TRAN

5. 可序列化

可序列化也是一种悲观隔离级别。通过确保再次发出查询时查询期间不添加行，可序列化隔离级别增强了可重复读的属性。换言之，同一查询在一个事务中被发出两次时不会出现幻像。因此，可序列化

是最强的悲观隔离级别，因为它阻止前面讨论过的所有可能的非预期行为——也就是说，它不允许未提交读、不可重复读或幻像，并且它也保证事务可串行运行。

阻止幻像是另一个想要的保证。同样，也没有免费的午餐。这重额外保护的成本类似于可重复读的成本——事务中所有的共享锁必须持有到事务完成。此外，实施可序列化隔离级别要求您不仅锁定已读取的数据，还要锁定不存在的数据！例如，假设在一个事务中，我们发出一个 *SELECT* 语句，读取邮编在 98000 和 98100 之间的所有客户，在第一次执行时，没有满足该条件的行。要实施可序列化隔离级别，我们必须锁定邮编在 98000 和 98100 之间的所有潜在行，以便在再次发出相同查询时，还是没有满足该条件的行。SQL Server 通过使用一种特殊的锁（叫做*键范围锁*）来处理该情形。键范围锁要求在定义值范围的列（在本例中，应该是包含邮编的列）上有一个索引。如果该列上没有索引，那么可序列化隔离级别需要一个表锁。我将在关于锁定的一节中详细讨论不同类型的锁。可序列化隔离级别的名称来源于这样的事实，即同时运行多个可序列化的事务等价于一次运行一个事务——串行地运行。

例如，假设事务 A、B 和 C 在可序列化隔离级别下同步运行，并且每个事务都试图更新同一范围的数据。如果事务获得数据范围上的锁的顺序依次是 B、C、A，那么同步运行所有 3 个事务得到的结果与串行地依次运行 B、C、A 的结果相同。表 10-1 演示的情况是，两个事务不能串行地运行并得到相同结果。

表 10-2 总结了每种隔离级别中可能的行为，并指出了用于实现每种级别的并发控制模型。可以看到，已提交读和已提交读（快照）在允许的行为方面是相同的，但是行为的实现方式不同——一种是悲观的（锁定），一种是乐观的（行版本控制）。可序列化和快照也都对所有行为使用 No 值，但是一个是悲观的，一个是乐观的。

表 10-2　每种隔离级别中允许的行为

隔离级别	脏读	不可重复读	幻像	并发控制
未提交读	Yes	Yes	Yes	悲观
已提交读（锁定）	No	Yes	Yes	悲观
已提交读（快照）	No	Yes	Yes	乐观
可重复读	No	No	Yes	悲观
快照	No	No	No	乐观
可序列化	No	No	No	悲观

10.3 锁定

锁定是任何多用户数据库系统（包括 SQL Server 在内）的一个至关重要的功能。悲观和乐观并发模型中都用到了锁，只不过两种模型中，其他进程处理被锁定数据的方式不同。我将已提交读隔离级别的悲观变体称为已提交读（锁定），因为锁定允许并发事务维护一致性。在悲观模型中，写入者总是阻塞读取者和写入者，并且读取者可以阻塞写入者。在乐观模型中，唯一发生的阻塞是写入者阻塞其他写入者。但是要真正理解这些简化的行为摘要的含义，需要详细了解 SQL Server 锁定。

10.3.1 锁定基础

SQL Server 可以使用好几种不同的模式锁定数据。例如，读操作获得共享锁，写操作获得独占锁。在更新操作的初始部分，即 SQL Server 在搜索要更新的数据时，获得更新锁。SQL Server 自动获得并释

放所有这些类型的锁。它也管理锁模式之间的兼容性，解决死锁，以及在必要时让锁升级。它控制表上、表页上、索引键上及单个数据行上的锁。锁也可以持有在系统数据上，即对数据库系统私有的数据上，比如页眉和索引。

SQL Server 提供了两种独立的锁定系统。第一种系统影响所有完全共享的数据，并对表、数据页、大型对象（Large Object，LOB）页和叶级索引页提供行锁、页锁和表锁。第二种系统内部用于索引并发控制，控制对内部数据结构的访问，以及检索单个数据行页面。这第二种系统使用闩锁，闩锁没有锁那么对资源敏感并提供性能优化。可以对所有锁定都使用完全锁，但是由于这种锁比较复杂，如果为所有内部需求都使用它们的话，会减慢系统速度。如果使用 *sp_lock* 系统存储过程或者一种类似的机制（从 *sys.dm_tran_locks* 视图获得信息）来考察锁，则不会看到闩锁——只会看到关于锁的信息。

另一种看到锁与闩锁之间区别的方式是，锁确保数据的逻辑一致性，闩锁确保物理一致性。当在页面上物理地放置一行数据或者以其他方式（比如压缩页面上的空间）移动数据时，会发生闩锁锁定。SQL Server 必须保证，该数据移动过程中不会受到干涉。

10.3.2 旋转锁

针对短期需要，SQL Server 利用旋转锁获得互斥。旋转锁纯粹用于互斥，从不用于锁定用户数据。它们甚至比闩锁更加轻量级，闩锁比用于锁定数据和索引叶级页的完全锁要简便一些。如果锁不是即时可用，旋转锁的请求者就重复它的请求（这就是说，请求者不断地“旋转”锁，直到锁变得可用）。

旋转锁通常针对不太忙的资源，在 SQL Server 中用做互斥体。如果资源很忙，那么旋转锁的持有时间足够短，使得重试比起等待并由操作系统重新调度更好一些，后者会导致线程之间的上下文切换。只要不必旋转太长时间，省去上下文切换会超出弥补旋转成本。旋转锁用于这样一些情形，即希望等待资源的时间较短（或者不希望等待）。*sys.dm_os_tasks* 动态管理视图（DMV）为任何当前使用旋转锁的任务显示了 SPINLOOP 的状态。

10.3.3 用户数据的锁类型

我们考察锁定用户数据的 4 个方面。首先来看锁定模式（锁的类型）。我们已经提到过共享锁、独占锁和更新锁，下面详细地介绍这些模式及其他模式。接下来介绍锁的粒度，即指定一个锁涵盖多少数据，可以是一行、一页、一索引键、一个索引键范围、一个范围、一个分区或者整个表。锁定的第三个方面是锁的持续时间。正如前面所提到的，一些锁只要数据访问完就被释放，一些锁要一直持有到事务提交或回滚。锁定的第四个方面涉及锁的所有权（锁的作用域）。锁可以由会话、事务或游标拥有。

10.3.4 锁模式

SQL Server 使用好几种锁定模式，包括共享锁、独占锁、更新锁和专用锁，以及这些锁的变种。正是由锁的模式决定当前请求的锁是否与已经同意的兼容。我们在本节末尾的图 10-2 中可以看到各种锁之间的兼容性。

1. 共享锁

共享锁在数据被读取时由 SQL Server 自动获得。可在表、页面、索引键或单个行上持有共享锁。很多进程可以在相同数据上持有共享锁，但是进程不能在已经具有共享锁的数据上获得独占锁（除非请求独占锁的进程就是那个持有共享锁的进程）。通常，共享锁只要数据读取完毕就会被释放，但是这可以通

过使用查询提示或不同的事务隔离级别而改变。

2. 独占锁

当数据被 *INSERT*、*UPDATE* 或 *DELETE* 操作修改时，SQL Server 自动获得数据上的独占锁。在一个特定的数据源上，同一时刻只有一个进程可以获得独占锁；事实上，当我们在本章后面讨论锁兼容性时，您可以看到，如果一个进程独占地锁定了请求的数据源，那么别的进程无法在此数据源上获得任何种类的锁。独占锁一直持有到事务结束。这意味着，更改的数据通常不对任何其他进程可用，直到当前事务提交或回滚。通过使用查询提示，其他进程可以决定独占地读取锁定的数据。

3. 更新锁

更新锁实际上并不是一种独立的锁，而是共享锁和独占锁的混合。当 SQL Server 执行数据修改操作却首先需要搜索表以找到需要修改的资源时，会获得更新锁。使用查询提示，进程可以明确地请求更新锁，在这种情况下，更新锁可以阻止本章后面图 10-6 中给出的转换死锁情形。

更新锁提供与数据的其他当前读取者的兼容性，在确保数据自上次读取以后尚未修改的前提下，使进程能够在稍后对数据进行修改。更新锁不足以允许您改变数据——所有修改都要求被修改的数据资源具有独占锁。更新锁的作用就好像一个序列化闸门（serialization gate），将后续申请独占锁的请求压入队列中（许多进程都可以对一个资源持有共享锁，但是只有一个进程能够持有更新锁）。只要有一个进程对资源持有更新锁，其他进程就无法获取该资源的更新锁或独占锁，其他正在申请相同资源上更新锁或独占锁的进程就必须等待。持有更新锁的进程能够将其转换成该资源上的独占锁，因为更新锁阻止与其他进程之间的锁不兼容性。可以将更新锁看做是“意图更新”锁，这在本质上就是它们所扮演的角色。单独使用的话，更新锁还是不足以用于更新数据——实际的数据修改仍然需要用到独占锁。对于独占锁的序列化访问可以避免转换死锁的发生。更新锁会保留到事务结束或者当它们转换成独占锁时为止。

不要被名字误导：更新锁并不只用于 *UPDATE* 操作。SQL Server 对任何需要在进行实际修改之前搜索数据的数据修改操作使用更新锁。这样的操作包括受限更新及删除，也包括在带有聚集索引的表上进行的插入操作。对于后面一种情况，SQL Server 必须先搜索数据（使用聚集索引）以找到正确的位置来插入新的记录。当 SQL Server 只进行到搜索阶段时，它采用更新锁来保护数据，而只有当它找到正确的位置并开始插入以后才将更新锁转换成独占锁。

4. 意向锁

意向锁实际上并不是一种独立的锁定模式，它们是之前讨论过的那些模式的限定词。换言之，可以拥有意向共享锁、意向独占锁甚至意向更新锁。由于 SQL Server 可以在不同级别的粒度上获取锁，因此需要一种机制来指出一个资源上的组件已经被锁定了。举例来说，如果一个进程试图锁定一张表，SQL Server 就需要采用一种方式来确定这张表上的一行（或者一页）是否已经被锁定了。意向锁就是起这个作用的。在介绍锁的粒度时我们还会进一步讨论意向锁。

5. 特殊锁模式

SQL Server 提供 3 种额外的锁模式：架构稳定锁、架构修改锁和大容量更新锁。当查询被编译时，架构稳定锁会防止其他进程获取架构修改锁（在表结构被修改时获得）。在执行 *BULK INSERT* 命令或者使用 bcp 实用工具将数据导入表时会获取大容量更新锁。另外，大容量导入操作必须使用 TABLOCK 查

询提示来请求这个特殊的锁。或者，表的所有者可以将表的 *table lock on bulk load* 选项设为 True，然后任何 *BULK COPY IN* 或者 *BULK INSERT* 操作都会自动请求大容量更新锁。这种特殊的大容量更新表的锁，请求它不一定意味着能被授予。如果其他进程已经持有了表上的锁，或者表上有索引存在，就不会授予大容量更新锁。如果有多个连接已经请求并得到了一个大容量更新锁，那么它们可以执行并行调用将数据导入相同的表中。与独占锁不同的是，大容量更新锁之间不会互相冲突，因此多个连接的并发插入在 SQL Server 中是受支持的。

6. 转换锁

转换锁不会由 SQL Server 直接请求，而是从一种模式转换到另一种模式所造成的。SQL Server 2008 支持的 3 类转换锁是 SIX、SIU 和 UIX。其中最常见的是 SIX，如果事务持有一个资源上的共享锁（S），然后又需要一个 IX 锁，此时出现的锁模式就是 SIX。例如，假设您发出以下批命令：

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRAN
SELECT * FROM bigtable
UPDATE bigtable
    SET col = 0
    WHERE keycolumn = 100
```

如果表很大，*SELECT* 语句会获得一个共享表锁（如果表只有为数不多的几行，SQL Server 就会获得单个行或键上的锁）。*UPDATE* 语句然后会获得单个独占锁以执行单个行的更新，而键级别的 X 锁意味着在页和表级别上存在 IX 锁。当通过 *sys.dm_tran_locks* 查看时，该表上会显示有 SIX 锁。类似地，当一个进程在一张表上拥有共享锁并在该表的一行上拥有更新锁时，就出现了 SIU 锁；而当一个进程在一张表上拥有更新锁并在一行上拥有独占锁时，就出现了 UIX 锁。

表 10-3 显示了大多数的锁模式，以及 *sys.dm_tran_locks* 中使用的缩写。

表 10-3　SQL Server 锁模式

缩　写	锁 模 式	说　明
S	Shared	允许其他进程读取但不能修改锁定的资源
X	Exclusive	防止别的进程修改或者读取锁定资源中的数据
U	Update	防止其他进程获得更新锁或独占锁；在搜索要修改的数据时使用
IS	Intent shared	表示该资源的一个组件被共享锁锁定了。只有在表或页级别才能获得这类锁
IU	Intent Update	表示该资源的一个组件被更新锁锁定了。只有在表或页级别才能获得这类锁
IX	Intent exclusive	表示该资源的一个组件被独占锁锁定了。只有在表或页级别才能获得这类锁
SIX	Shared with intent exclusive	表示一个正持有共享锁的资源还有一个组件（一页或一行）被独占锁锁定了
SIU	Shared with intent *UPDATE*	表示一个正持有共享锁的资源还有一个组件（一页或一行）被更新锁锁定了
UIX	Update with intent exclusive	表示一个正持有更新锁的资源还有一个组件（一页或一行）被独占锁锁定了
Sch-S	Schema stability	表示一个使用该表的查询正在被编译
Sch-M	Schema modification	表示表的结构正在被修改
BU	Bulk Update	在一个大容量复制操作将数据导入表中并且(手动或自动)应用了 TABLOCK 查询提示时使用

7. 键范围锁

另外的锁模式（称为*键范围锁*）只在可序列化隔离级别中为了锁定一定范围内的数据而被获取。大多数锁模式能够应用于几乎任何锁资源上。例如，共享锁和独占锁可以在表、页、行或键上获取。因为键范围锁只能在键上获取，所以我们将在稍后介绍键锁的一节中深入了解键范围锁。

10.3.5 锁粒度

SQL Server 可以在表、页、行等级别锁定用户数据资源（非系统资源，系统资源是用闩锁来保护的）。（如果锁被升级，SQL Server 也可以锁定表或索引的一部分）此外，SQL Server 还可以锁定索引键和索引键范围。图 10-1 显示了资源第一次被访问时表中可以获得的基本锁级别。记住，如果表具有聚集索引，那么数据行就在聚集索引的叶级别并且是被键锁而不是行锁锁定的。

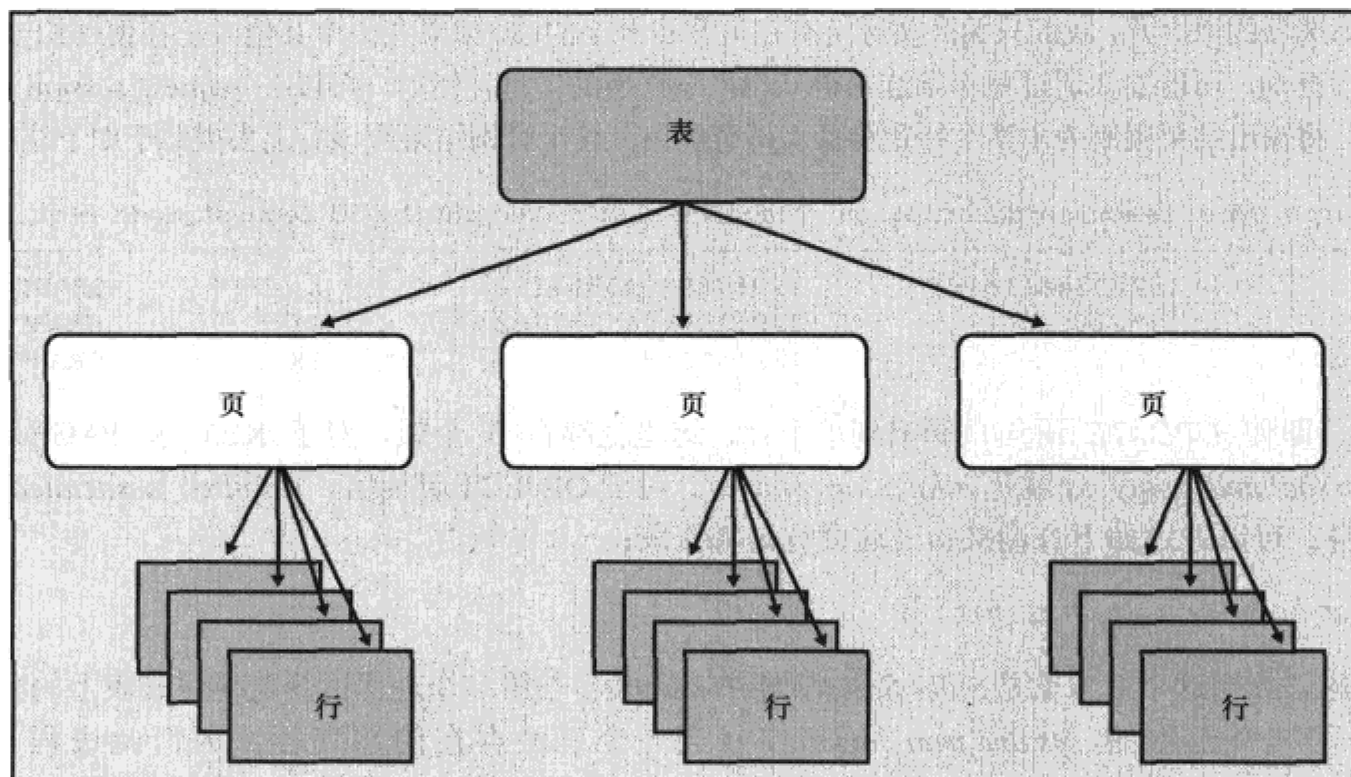


图 10-1 表上的 SQL Server 锁的粒度级别

sys.dm_tran_locks 视图对每个锁都进行追踪并包含关于资源的信息：谁被锁定了（如行、键或页）、锁的模式及特定资源的标识符。记住，*sys.dm_tran_locks* 只是一个用来显示所持有锁的信息的动态视图。实际的信息存储在内部的 SQL Server 结构中，对我们完全不可见。因此，我们谈到 *sys.dm_tran_locks* 视图中的信息时，实际上指的是通过该视图能够看到的信息。

当进程请求一个锁时，SQL Server 会将所请求的锁与已经列出在 *sys.dm_tran_locks* 中的资源进行比较，寻找完全匹配资源类型和标识符的锁。但是，譬如说，如果一个进程在 *Sales.SalesOrderHeader* 表中的某行上具有一个独占锁，那么别的进程可能会尝试获得整个 *Sales.SalesOrderHeader* 表上的锁。由于是两种不同的资源，SQL Server 不会找到一个完全的匹配，除非额外信息已经存在于 *sys.dm_tran_locks* 之中。这就是意向锁存在的理由了。在 *Sales.SalesOrderHeader* 表的一行记录上拥有独占锁的进程在包含该行记录的页上拥有一个意向独占锁，也在包含该行记录的这张表上拥有另一个意向独占锁。通过运行以下代码，可以看到这些锁：

```
USE Adventureworks2008;
BEGIN TRAN
UPDATE Sales.SalesOrderHeader
SET ShipDate = ShipDate + 1
WHERE SalesOrderID = 43666;
```

这条语句应该影响到单个行。由于启动了一个事务且尚未结束，所以获得的独占锁仍旧被持有。我们可以使用 *sys.dm_tran_locks* 视图来看这些锁：

```
SELECT resource_type, resource_description,
       resource_associated_entity_id, request_mode, request_status
FROM sys.dm_tran_locks
WHERE resource_associated_entity_id > 0;
```

本章稍后部分将介绍更多关于 *sys.dm_tran_locks* 视图内的信息，但是就目前来说，要注意在 WHERE 子句中使用筛选器的原因是，我们只关注实际持有在数据资源上的锁。如果在一个其他人正在使用的 SQL Server 实例上执行查询，可能需要通过更多筛选来得到自己感兴趣的信息。例如，可以在 *request_session_id* 字段上加以筛选，将输出结果限制为由某个特定会话所持有的锁。这个查询结果应该看上去类似于如下结果：

```
resource_type  resource_description  resource_associated_entity_id request_mode request_status
-------------  --------------------  ------------------------------ ------------ --------------
KEY            (92007ad11d1d)        72057594045857792              X            GRANT
PAGE           1:5280                72057594045857792              IX           GRANT
OBJECT                               722101613                      IX           GRANT
```

注意，即使 *UPDATE* 语句只影响到单个行，这里也存在 3 个锁。对于 KEY 和 PAGE 锁来说，*resource_associated_entity_id* 就是 *allocation_unit_id*。对于 OBJECT 锁而言，*resource_associated_entity_id* 就是表本身。可以通过如下查询来验证这究竟是哪张表：

```
SELECT object_name(722101613)
```

结果应该表明这个对象就是 *Sales.SalesOrderHeader* 表。当第二个进程试图获得这张表上的独占锁时，它会在相同锁资源上的 *sys.dm_tran_locks* 中找到一个已经存在的冲突行，因而此进程被阻塞。*sys.dm_tran_locks* 视图显示出下面这行结果，表示请求该对象上的独占锁还没有得到允许。这个请求锁的进程正处于 WAIT 状态。

```
resource_type  resource_description  resource_associated_entity_id request_mode request_status
-------------  --------------------  ------------------------------ ------------ --------------
OBJECT                               722101613                      X            WAIT
```

并不是所有已被锁定资源上的锁申请都会导致冲突。当一个进程要在已经被别的进程锁定的资源上申请不相兼容的锁时会产生冲突。例如，两个进程可以在同一资源上获取共享锁，因为共享锁之间是互相兼容的。本章稍后部分将详细介绍锁的兼容性。

1．键锁

SQL Server 2008 支持两种类型的键锁，而它究竟采用哪种类型则取决于当前事务的隔离级别。如果隔离级别是已提交读、可重复读或快照，SQL Server 会在处理查询时尝试锁定实际被访问的索引键。对于具有聚集索引的表而言，数据行就是索引的叶级别，您看到获得的是键锁。如果表是一个堆的话，您

可能看到非聚集索引上的键锁和实际数据上的行锁。

如果隔离级别是可序列化，情况就有所不同了。为了防止幻像，如果在一个事务中扫描了一个范围内的数据，就需要充分锁定该表以确保没人能够插入新值到已扫描的范围内。例如，可以在 *AdventureWorks2008* 数据库中发起一个显式事务并执行如下查询：

```
BEGIN TRAN
SELECT * FROM Sales.SalesOrderHeader
WHERE CustomerID BETWEEN 100 and 110;
```

当使用可序列化隔离时，必须获取锁以确保在事务结束之前没有 *CustomerID* 值介于 100 和 110 的新记录被插入。在 SQL Server 的许多早期版本中（7.0 以前），是通过锁定整个页甚至整张表来保证这一点的。但是在许多情况下，这样的限制太过分了——被锁定的数据比 WHERE 子句所指示的实际数据要多，造成不必要的资源争用。SQL Server 2008 采用键范围锁模式，与索引中的特定键值相关联，并表明在索引中这个键与上一个键之间的所有值都被锁定。

AdventureWorks2008 数据库包含一个在 *Person* 表的 *LastName* 列上的索引。假设当前正处于 TRANSACTION ISOLATION LEVEL SERIALIZABLE 状态，并在一个用户定义的事务中发出以下 *SELECT* 命令：

```
SELECT * FROM Person.Person
WHERE LastName BETWEEN 'Freller' AND 'Freund';
```

如果 *Fredericksen*、*French* 和 *Friedland* 在 *LastName* 列的索引中是连续的叶级别索引键，而其中后面的两个键（*French* 和 *Friedland*）获得键范围锁（尽管只有一行针对 *French* 的记录返回到结果集中）。键范围锁可以防止在以这两个键范围锁为终点的范围内进行任何插入操作。大于 *Fredericksen* 且小于或等于 *French* 的值不能被插入，并且大于 *French* 且小于或等于 *Friedland* 的值不能被插入。注意，键范围锁涵盖了从前一个连续键开始的开区间到放置锁的那个键为止的闭区间。这两个键范围锁可以防止有人插入 *Fremlich* 或者 *Frenkin*（介于 WHERE 子句所指定的范围之内）。然而，键范围锁还会防止有人插入 *Freedman*（*Freedman* 大于 *Fredericksen* 小于 *French*），即使 *Freedman* 并不在查询所指定的范围之内。键范围锁并非完美，但它们的确在保证不会出现幻像的前提下比锁定整个页或表提供了更高的并发能力。

一共有 9 个类型的键范围锁，而其中每一个都有一个二段式的名字：第一段表示相邻索引键之间数据范围上的锁的类型，第二段表示键本身的锁的类型。表 10-4 描述了这 9 种键范围锁。

表 10-4　键范围锁类型

缩　写	说　明
RangeS-S	键之间的范围上是共享锁；范围的终点（键本身）上是共享锁
RangeS-U	键之间的范围上是共享锁；范围的终点上是更新锁
RangeIn-Null	防止在键之间的范围上插入的独占锁；键本身没有锁
RangeX-X	键之间的范围上是独占锁；范围的终点上是独占锁
RangeIn-S	由 S 锁和 RangeIn_Null 锁创建的转换锁
RangeIn-U	由 U 锁和 RangeIn_Null 锁创建的转换锁
RangeIn-X	由 X 锁和 RangeIn_Null 锁创建的转换锁
RangeX-S	由 RangeIn_Null 锁和 RangeS-S 锁创建的转换锁
RangeX-U	由 RangeIn_Null 锁和 RangeS-U 锁创建的转换锁

这些锁模式中很多是非常罕见的或者持续时间非常短暂，因此在 *sys.dm_tran_locks* 中也不是经常能见到。例如，在一个使用可序列化隔离的会话中，RangeIn-Null 锁会在 SQL Server 尝试插入值到键之间的范围内时被获取。这种类型的锁并不常见，因为它的持续时间总是非常短暂。这类锁只会保留到正确的插入位置被发现为止，而紧接着它就会升级为 X 锁。然而，如果在可序列化隔离级别下，一个事务扫描了一个范围内的数据而紧接着另一个事务尝试在这个范围内插入数据，这时第二个事务会发起一个状态为 WAIT、模式为 RangeIn-Null 的锁请求。可以通过查看 *sys.dm_tran_locks* 视图中的 *status* 列来观察这个锁请求，本章稍后将详细介绍 *sys.dm_tran_locks*。

2. 另外的锁资源

除了对象、页、键和行上的锁之外，SQL Server 还可以锁定一些其他的资源。扩展（磁盘空间上具有 64KB 大小的分配单元，由 8 个 8KB 大小的页组成）上也可以加锁。当一个表或索引需要增长且必须分配一个新的扩展时，这类锁定就会自动发生。可以把扩展锁想象成另一种类型的具有特殊用途的闩锁，但是它不会显示在 *sys.dm_tran_locks* 中。扩展可以拥有共享扩展锁和独占扩展锁。

查看 *sys.dm_tran_locks* 的内容时，应该会注意到大多数进程至少在一个数据库上持有锁（*resource_type* = 'DATABASE'）。实际上，在除 *master* 和 *tempdb* 之外的任何数据库上持锁的所有进程都拥有一个该数据库资源上的锁。如果进程只是使用数据库，那么这些数据库锁总是共享锁。SQL Server 在判断一个数据库是否正在被使用时检查这些数据库锁，然后才能判断该数据库能否被删除、还原、修改或关闭。由于很少能对 *master* 和 *tempdb* 数据库进行改变而且它们也是无法删除或关闭的，所以数据库锁对它们而言是没有意义的。此外，*tempdb* 从来不会被还原，而还原 *master* 数据库必须以单用户模式启动整个服务器，所以也说明数据库锁对它们来说没有必要。当尝试执行这些操作时，SQL Server 会申请一个独占的数据库锁，而如果有任何其他进程在该数据库上持有共享锁，这个请求就会被阻塞。一般来说，您不需要关心扩展锁或数据库锁，但如果细读 *sys.dm_tran_locks* 的内容，也会看到它们。

可能偶尔还会在 ALLOCATION_UNIT 资源上看到锁的存在。尽管所有表和索引结构都包含一个或多个 ALLOCATION_UNIT，当出现这些锁的时候，意味着 SQL Server 正在处理这些资源中某个不再与一个特定对象关联的资源。例如，当删除或者重建庞大的表或索引时，实际的页回收操作被推迟，直到事务提交之后才进行。被延迟的删除操作并不立即释放已分配的空间，而且还引入了额外开销，因此被延迟的删除操作只在使用了超过 128 个扩展的表或索引上执行。如果表或索引只使用了 128 个或更少的扩展，那么删除、截断和重建操作就不会被延迟。在延迟操作的第一阶段，表或索引所使用的现有分配单元会被标记为进行回收，并且被锁定到事务提交为止。此时就会在 *sys.dm_tran_locks* 中看到 ALLOCATION_UNIT 锁。还可以在 *sys.allocation_units* 视图中找到 type_desc 值为 DROPPED 的分配单元，看有多少空间被分配单元占用，导致这部分空间既无法被重用又不属于任何对象。分配单元空间的实际物理删除操作会在事务提交以后发生。

最后，偶尔也可以具有单个分区上的锁，这种锁在锁元数据中显示为 HOBT 锁。只有当锁被升级，并且指定了该升级对分区级别是允许的时候（当然，只有当表或索引已经分区时），才会出现这种锁。在本章后面关于锁升级的部分，我们要介绍如何指定允许分区级别锁定。

3. 鉴别锁资源

当 SQL Server 要决定是否可以授予一个申请的锁时，会检查 *sys.dm_tran_locks* 视图，判断是否已经有冲突锁模式的匹配锁存在。它通过查看数据库 ID（*resource _database_ID*）、*resource_description*、

resource_associated_entity_id 列中的值，以及被锁定资源的类型，对锁进行比较。SQL Server 并不了解资源描述中的含义。它只是比较识别锁资源的字符串来寻找一个匹配。如果发现一个匹配且 *request_status* 值是 GRANT，它就知道资源已经被锁定。然后 SQL Server 会利用锁兼容性矩阵来判断当前的锁是否与某个正在被申请锁兼容。表 10-5 列出了许多在 *sys.dm_tran_locks* 视图的第一列中显示的可能的锁资源和 *resource_description* 列中的信息（用来定义实际被锁定的资源）。

表 10-5　SQL Server 中可锁定的资源

资源类型	Resource_description	例　子
DATABASE	无：每个被锁资源的 *resource_database_ID* 列都指明了数据库	12
OBJECT	对象 ID（可以是任何数据库对象，不一定就是表）显示在 *resource_associated_entity_id* 列中	69575286
HOBT	*hobt_id* 显示在 *resource_associated_entity_id* 列中。只有当针对表的分区锁定被允许时才使用	72057594038779904
EXTENT	文件号:扩展（extent）的第一个页的页码	1:96
PAGE	文件号:实际表或索引页的页码	1:104
KEY	由所有键的组成部分及定位符得到的哈希值。对于一个堆上的非聚集索引（*c1* 和 *c2* 是索引列），哈希将包含来自 *c1*、*c2* 和 *RID* 的贡献	ac0001a10a00
ROW	实际数据行的文件号:页码:槽号	1:161:3

注意，键锁和键范围锁具有相同的资源描述，因为键范围被视为一种锁定模式，而不是一种锁定资源。当查看 *sys.dm_tran_locks* 视图的输出结果时，可以通过锁模式列的值来区分各种类型的锁。

另一种类型的可锁定资源是 METADATA。METADATA 资源被划分成多个子类型，由 *sys.dm_tran_locks* 视图中的 *resource_subtype* 列来描述。您可能会发现 METADATA 资源的许多子类型，但其中的大多数都超出了本书的范围。对于其中的一些，即使在 *SQL Server 联机丛书*中将其描述为“仅供内部使用”，但是它们所指的含义还是相当明显的。例如，当改变数据库的属性时，可以看到一个资源类型（*resource_type*）为 METADATA、资源子类型（*resource_subtype*）为 DATABASE 的记录。这条记录的 *resource_description* 列的值是 *database_id* =<*ID*>，表示其元数据当前被锁定的数据库的 ID。

4. 关联实体 ID

对于一个较大实体内的一部分的锁定资源，*sys.dm_tran_locks* 中的 *resource*_associated_entity_id 列显示了数据库中的这个关联实体的 ID。它可以是对象 ID、分区 ID 或分配单元 ID，具体取决于资源类型。当然，对于某些资源，像 DATABASE 和 EXTENT，没有对应的 *resource_associated_entity_id*。对于 OBJECT 类型的资源，该列中指定了对象 ID 值，而对于 ALLOCATION_UNIT 资源，该列中指定了分配单元的 ID。对于资源类型 PAGE、KEY 和 RID，则提供了一个分区 ID。

没有简单的函数可以将分区 ID 值转换成对象名，实际上必须从 *sys.partitions* 视图中选取。下面的查询通过将 *sys.dm_tran_locks* 视图与 *sys.partitions* 视图联接，对当前数据库中全部锁的 *resource_associated_entity_id* 值进行转换。对于 OBJECT 型资源，在 *resource_associated_entity_id* 列上应用 object_name 函数。对于 PAGE、KEY 和 RID 资源，对从 *sys.partitions* 视图中得到的 *object_id* 值使用 *object_name* 函数。对于其他没有 *resource_associated_entity_id* 的资源，代码只返回 n/a。由于 *object_name* 函数代码引用了 *sys.partitions* 视图，而该视图出现在每个数据库中，所以代码进行了筛选，只返回当前数据库中资源的锁信息。其输出是按照 *sp_lock* 存储

过程返回信息的格式来组织的，但是您可以自行添加任何额外的筛选或者所需的列。在本章后面的许多例子中要用到这个查询，所以我们基于这个 *SELECT* 语句新建一个名为 *DBlocks* 的视图：

```
CREATE VIEW DBlocks AS
SELECT request_session_id as spid,
     db_name(resource_database_id) as dbname,
     CASE
   WHEN resource_type = 'OBJECT' THEN
        object_name(resource_associated_entity_id)
      WHEN resource_associated_entity_id = 0 THEN 'n/a'
   ELSE object_name(p.object_id)
    END as entity_name, index_id,
        resource_type as resource,
        resource_description as description,
        request_mode as mode, request_status as status
FROM sys.dm_tran_locks t LEFT JOIN sys.partitions p
   ON p.partition_id = t.resource_associated_entity_id
WHERE resource_database_id = db_id();
```

10.3.6　锁的持续时间

锁的持续时间主要取决于锁的模式和当前起作用的事务隔离级别。SQL Server 的默认隔离级别是已提交读。在该级别下，SQL Server 会在读取和处理完锁定数据以后立刻释放共享锁。对于快照隔离级别，其行为也是相同的——SQL Server 读完数据后立即释放共享锁。如果事务隔离级别是可重复读或可序列化，共享锁就和独占锁的持续时间相同。也就是说，直到事务结束时才会被释放。对于任何隔离级别，独占锁都要一直持续到事务结束为止，无论事务是被提交还是被回滚。更新锁也会持续到事务结束，除非它被升级成了独占锁（这种情况下的独占锁跟所有情况下的独占锁一样，总会保留到事务结束为止）。

除了改变事务隔离级别，还可以使用查询提示来控制锁的持续时间。本章稍后部分会简要讨论锁定的查询提示。

10.3.7　锁的所有权

锁的持续时间也受到锁的所有权的直接影响。锁的所有权与申请锁的进程没有关系，可以将其想象成锁的“作用域”。一共存在 4 种锁所有者（或作用域）类型：事务、游标、事务工作空间（transaction_workspace）和会话。可以通过 *sys.dm_tran_locks* 视图中的 *request_owner_type* 列来查看锁的所有者。

之前讨论的大多数锁定处理的锁，其所有者都是事务。正如我们所见，根据事务的隔离级别和锁定模式，这类锁可以拥有两个不同的持续时间。已提交读隔离下的共享锁的持续时间只是被锁定数据被读取的时间。由事务所拥有的全部其他锁的持续时间要一直持续到事务结束为止。

request_owner_type 的值为 CURSOR 的锁必须在声明游标时显式地申请。如果一个游标是采用 SCROLL_LOCKS 锁定模式打开的话，那么每次被提取的行记录上都会持有一个游标锁，直到下一行记录被提取或者游标被关闭为止。即使事务在下一次提取之前就被提交了，游标锁也不会被释放。

在 SQL Server 2008 中，由一个会话所拥有的锁也必须显式地申请并且只适用于应用程序锁。会话型锁是通过 *sp_getapplock* 存储过程来申请的。它会一直持续到会话断开或者锁被显示地释放为止。

每次数据库被访问，都会获得事务工作空间型锁，与这些锁相关联的资源总是数据库。工作空间为登记进入通用环境中的会话持有数据库锁。通常，每个会话对应一个工作空间，因此会话中获取的所有

数据库锁都保留在同一个工作空间对象中。在分布式事务情况下，多个会话会进入同一个工作空间，因而它们共享数据库锁。

当进程发出 USE 命令时，会在一个数据库上获取一个所有者为 SHARED_ TRANSACTION_WORKSPACE 的数据库锁。例外发生在任何使用 *master* 或 *tempdb* 数据库的进程上，在这种情况下不会获取数据库锁。这个锁要持有到发出另一个 USE 命令或者进程断开时释放。如果一个进程试图修改、还原或删除数据库，那么所获取的数据库锁的所有者就是 EXCLUSIVE_TRANSACTION_WORKSPACE。SHARED_TRANSACTION_WORKSPACE 和 EXCLUSIVE_TRANSACTION_WORKSPACE 锁由同一个工作空间维持，只不过是一个工作空间中的两个不同列表而已。在这种情况下，使用两种不同的所有者名容易引起误解。

10.3.8 查看锁

为了弄清系统当前正处于活动和等待状态的锁，最佳的信息来源是 *sys.dm_tran_locks* 视图。在之前的小节中已经展示了一些基于该视图的查询，而在本小节中还会再介绍一些并进一步解释其输出列的含义。这个视图取代了 *sp_lock* 存储过程。尽管调用存储过程可能比查询 *sys.dm_tran_locks* 视图要少打一些字，但是视图比存储过程灵活得多。*sys.dm_tran_locks* 不但包括了更多提供锁的详细信息的列，而且它作为一个视图，可以在查询过程中只选择需要的列或者符合某些条件的记录。它还能与其他视图联接并聚合，以得到诸如每种类型的锁有多少之类的综述信息。

sys.dm_tran_locks

sys.dm_tran_locks 中的所有列（最后一个名为 lock_owner_address 的列除外）都是以两种前缀中的一种打头。以 *resource_* 打头的列描述的资源是 SQL Server 正在申请该资源上的锁。以 *request_* 打头的字段描述的是正在申请锁的进程。只有当所有的 *resource_* 列内容相同时，才表示两个申请作用于同一个资源上。

***resource_* 列。** 前面已经提到过大多数的 *resource_* 列，但只是简要的介绍。并不是所有资源都有子类型，而有些资源却有很多子类型。例如，METADATA 资源类型有 40 多种子类型。

表 10-6 列出了除 METADATA 类型以外的其他资源类型的所有子类型。

表 10-6 资源的子类型

资源类型	资源子类型	说明
DATABASE	BULKOP_BACKUP_DB	用于同步大容量操作的数据库备份
	BULKOP_BACKUP_LOG	用于同步大容量操作的数据库日志备份
	DDL	用于同步文件组的 DDL 操作（如 *DROP* 操作）
	STARTUP	用于数据库启动同步
TABLE	UPDSTATS	用于表上的统计更新的同步
	COMPILE	用于存储过程编译的同步
	INDEX_OPERATION	用于索引操作的同步
HOBT	INDEX_REORGANIZE	用于堆或索引重组操作的同步
	BULK_OPERATION	用于并发扫描的堆优化大容量加载操作，适用于快照、未提交读和已提交读 SI 隔离级别
ALLOCATION_UNIT	PAGE_COUNT	用于延迟的删除操作期间的分配单元页数统计的同步

如前所述，大多数 METADATA 子类型被描述为 INTERNAL USE ONLY，其含义显而易见。当发生改变时，每种类型的元数据都能分别被锁定。下面是部分 METADATA 子类型的列表。

- INDEXSTATS
- STATS
- SCHEMA
- DATABASE_PRINCIPAL
- DB_PRINCIPAL_SID
- USER_TYPE
- DATA_SPACE
- PARTITION_FUNCTION
- DATABASE
- SERVER_PRINCIPAL
- SERVER

大多数其他未列出的 METADATA 子类型涉及到的 SQL Server 2008 元素本书没有讨论，包括 CLR 例程、XML、证书、全文搜索和通知服务等。

***request_*列**。之前已经提到了 *sys.dm_tran_locks* 中几个最重要的 *request_*列，包括 *request_*mode（申请的锁的类型）、*request_owner_type*（申请的锁的范畴）和 *request_session_id*。这里再列举一些其他列。

- ***request_*type**。在 SQL Server 2008 中，*sys.dm_tran_locks* 中追踪的唯一资源申请类型是 LOCK。后续版本中还将包括其他可申请的资源类型。
- ***request_status***。请求的当前状态可以是以下 3 个值中的一个：GRANT、CONVERT 和 WAIT。CONVERT 状态表示请求者已经被授予了同一资源上的一个不同模式的请求，所以当前正在等待从当前锁模式升级（转换）为授予状态（例如，SQL Server 可以将一个 U 锁转换成 X 锁）。WAIT 状态表示请求者当前还未持有资源上的已授权请求。
- ***request_*reference_count**。该值返回同一请求者已请求该资源的大概次数，并且仅适用于那些在事务结束时不会自动释放的资源。如果该值递减到 0 并且 *request_*lifetime 也是 0 的话，一个已授权资源就不再被视做被某个请求者占有着了。
- ***request_*lifetime**。该值表示资源上的锁何时会释放。
- ***request_session_id***。该值表示申请锁的会话的 ID。对于分布式事务和绑定事务，拥有的会话 ID 可以改变。该值为–2 时，表示请求属于孤立的分布式事务。该值为–3 时，表示请求属于延迟的恢复事务（这些事务因为回滚未能成功完成而延迟恢复）。
- ***request_*exec_context_id**。该值表示当前拥有该请求的进程的执行上下文 ID。如果该值大于 0，表示这是一个用于执行并行查询的子线程。
- ***request_*request_id**。该值是当前拥有该请求的进程的请求 ID（批处理 ID）。只有对于来自使用多活动结果集（MARS）的客户端应用程序的请求，才会填充这一列。
- ***request_*owner_id**。该值目前仅用于所有者为事务的请求，且所有者 ID 就是事务 ID。该列可以与 *sys.dm_tran_active_transactions* 视图中的 *transaction_id* 列联接。
- ***request_*owner_guid**。该值目前仅用于分布式事务，在该值对应于事务的 DTC GUID 时使用。
- ***lock_owner_address***。该值用于跟踪请求的内部数据结构的内存地址。如果请求处于 WAIT 或 CONVERT 状态，该列可以与 *sys.dm_os_waiting_tasks* 中的 *resource_*address 列联接。

10.3.9 锁定示例

下面的例子通过前面介绍的 *DBlocks* 视图演示了之前讨论过的许多锁定类型和模式的输出结果。

示例 1：默认隔离级别下的 *SELECT*

SQL 批处理

```
USE Adventureworks2008;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN TRAN
SELECT * FROM Production.Product
WHERE Name = 'Reflector';
SELECT * FROM DBlocks WHERE spid = @@spid;
COMMIT TRAN
```

DBlocks 输出结果

spid	dbname	entity_name	index_id	resource	description	mode	status
60	Adventureworks2008	n/a	NULL	DATABASE		S	GRANT
60	AdventureWorks2008	Dblocks	NULL	OBJECT		IS	GRANT

Production.Product 表中的数据上没有锁存在，因为批处理只执行获得共享锁的 *SELECT* 操作。默认情况下，共享锁在数据读完之后就立刻被释放了，所以从视图上看，到执行 *SELECT* 语句时，锁已经不存在了。现在只有一个经常存在的数据库锁和一个视图上的对象锁。

示例 2：Repeatable Read 隔离级别下的 *SELECT*

SQL 批处理

```
USE AdventureWorks2008;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRAN
SELECT * FROM Production.Product
WHERE Name LIKE 'Racing Socks%';
SELECT * FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'Product';
COMMIT TRAN
```

DBlocks 输出结果

spid	dbname	entity_name	index_id	resource	description	mode	status
54	AdventureWorks2008	Product	NULL	OBJECT		IS	GRANT
54	AdventureWorks2008	Product	1	PAGE	1:16897	IS	GRANT
54	AdventureWorks2008	Product	1	KEY	(6b00b8eeda30)	S	GRANT
54	AdventureWorks2008	Product	1	KEY	(6a00dd896688)	S	GRANT
54	AdventureWorks2008	Product	3	KEY	(9502d56a217e)	S	GRANT
54	AdventureWorks2008	Product	3	PAGE	1:1767	IS	GRANT
54	AdventureWorks2008	Product	3	KEY	(9602945b3a67)	S	GRANT

这次，将数据库锁及视图和行集上的锁筛选掉了，只关注数据上的锁。由于 *Production.Product* 表上存在聚集索引，数据行就是叶级别的全部索引行。返回的两个数据行上的锁是键锁。该表上还有两个非聚集索引叶级别的键锁，用来寻找相关数据行。在 *Production.Product* 表中，那个非聚集索引存在于 *Name* 列上。可以通过 *index_id* 列的值来区分聚集索引和非聚集索引：数据行（聚集索引的叶级别行）的 *index_id* 值为 1，而非聚集索引行的 *index_id* 值为 3（对于非聚集索引，index_id 值可以是 2～250 或 356～1005 的

任意值)。由于事务隔离级别是可重复读，所以共享锁会持续到事务结束为止。注意，索引行拥有共享锁(S锁)，而数据页、索引页及表本身拥有意向共享锁(IS锁)。

示例3：可序列化隔离级别下的 *SELECT*

SQL 批处理

```
USE AdventureWorks2008 ;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRAN
SELECT * FROM Production.Product
WHERE Name LIKE 'Racing Socks%';
SELECT * FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'Product';
COMMIT TRAN
```

DBlocks 输出结果

```
spid dbname            entity_name index_id   resource description   mode      status
---- ----------------- ----------- ---------- -------- ------------- -------   ------
54   AdventureWorks2008 Product    NULL       OBJECT                 IS        GRANT
54   AdventureWorks2008 Product    1          PAGE     1:16897       IS        GRANT
54   AdventureWorks2008 Product    1          KEY      (6b00b8eeda30) S        GRANT
54   AdventureWorks2008 Product    1          KEY      (6a00dd896688) S        GRANT
54   AdventureWorks2008 Product    3          KEY      (9502d56a217e) RangeS-S GRANT
54   AdventureWorks2008 Product    3          PAGE     1:1767        IS        GRANT
54   AdventureWorks2008 Product    3          KEY      (23027a50f6db) RangeS-S GRANT
54   AdventureWorks2008 Product    3          KEY      (9602945b3a67) RangeS-S GRANT
```

可序列化隔离级别下持有的锁几乎与可重复读隔离级别下持有的完全一样。主要区别在锁定模式上。两部分模式 RangeS-S 表示，除了键本身上的锁以外，还有一个键范围锁。第一部分(RangeS)是加在介于持有锁的键(包含)和索引中的前一个键之间的键范围上的锁。键范围锁阻止其他事务向表中插入新的符合该查询条件的数据行，也就是说，不能够插入产品名以“*Racing Socks*”打头的新数据行。键范围锁存在于 *Name* 列上非聚集索引(*index_id = 3*)中的范围上，因为这就是用来寻找符合条件数据行的索引。非聚集索引中共有3个键锁，因为需要锁定3个不同的范围。两个 *Racing Socks* 行是(*Racing Socks, L*)和(*Racing Socks, M*)。SQL Server 必须锁定从索引中第一个 *Racing Socks* 行往前的一个键到第一个 *Racing Socks* 行之间的范围。还必须锁定两个以 *Racing Socks* 打头的数据行之间的范围，以及从第二个 *Racing Socks* 行往后直到索引中下一个键之间的范围。因此，事实上在 *Racing Socks* 到上一个键 *Pinch Bolt* 及 *Racing Socks* 到下一个键 *Rear Brakes* 的范围之内都不能插入新的数据。例如，我们无法插入一个名为 *Portkey* 或 *Racing Tights* 的新产品。

示例4：更新操作

SQL 批处理

```
USE AdventureWorks2008;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN TRAN
UPDATE Production.Product
SET ListPrice = ListPrice * 0.6
WHERE Name LIKE 'Racing Socks%';
SELECT * FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'Product';
COMMIT TRAN
```

DBlocks 输出结果

```
spid dbname             entity_name index_id  resource  description    mode   status
---- ------------------ ----------- --------- --------- -------------- ------ ------
54   AdventureWorks2008 Product     NULL      OBJECT                   IX     GRANT
54   AdventureWorks2008 Product     1         PAGE      1:16897        IX     GRANT
54   AdventureWorks2008 Product     1         KEY       (6b00b8eeda30) X      GRANT
54   AdventureWorks2008 Product     1         KEY       (6a00dd8966 88) X     GRANT
```

聚集索引叶级别中的两个数据行被 X 锁锁定。页和表则被 IX 锁锁定。我们曾经提到过，在寻找要更新的行时，SQL Server 实际上获得的是更新锁。然而，当实际的更新操作完成时，这些锁被升级成 X 锁，因此当我们查看 DBLocks 视图的时候，更新锁已经不见了。除非通过查询提示强制使用更新锁，否则可能永远无法从 DBLocks 视图的输出结果或者对 *sys.dm_tran_locks* 的直接观察中看到它们。

示例 5：在可序列化隔离级别下使用索引进行更新

SQL 批处理

```
USE AdventureWorks2008;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRAN
UPDATE Production.Product
SET ListPrice = ListPrice * 0.6
WHERE Name LIKE 'Racing Socks%';
SELECT * FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'Product';
COMMIT TRAN
```

DBlocks 输出结果

```
spid dbname             entity_name index_id  resource  description    mode     status
---- ------------------ ----------- --------- --------- -------------- -------- -----
54   AdventureWorks2008 Product     NULL      OBJECT                   IX       GRANT
54   AdventureWorks2008 Product     1         PAGE      1:16897        IX       GRANT
54   AdventureWorks2008 Product     1         KEY       (6a00dd896688) X        GRANT
54   AdventureWorks2008 Product     1         KEY       (6b00b8eeda30) X        GRANT
54   AdventureWorks2008 Product     3         KEY       (9502d56a217e) RangeS-U GRANT
54   AdventureWorks2008 Product     3         PAGE      1:1767         IU GRANT
54   AdventureWorks2008 Product     3         KEY       (23027a50f6db) RangeS-U GRANT
54   AdventureWorks2008 Product     3         KEY       (9602945b3a67) RangeS-U GRANT
```

再次注意，键范围锁存在于用来寻找相关数据行的非聚集索引上。范围间隔本身只需要一个共享锁来防止插入，但是被搜索的键上拥有 U 锁，所以其他进程都不能更新它们。当实际的修改发生时，表本身（*index_id = 1*）的键上会获取独占锁。

下面看一个相同隔离级别下不使用索引来搜索的更新操作。

示例 6：在可序列化隔离级别下不使用索引进行更新

SQL 批处理

```
USE AdventureWorks2008;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRAN
UPDATE Production.Product
SET ListPrice = ListPrice * 0.6
WHERE Color = 'White';
```

```
SELECT * FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'Product';
COMMIT TRAN
```

DBlocks 输出结果（省略版）

spid	dbname	entity_name	index_id	resource	description	mode	status
54	AdventureWorks2008	Product	NULL	OBJECT		IX	GRANT
54	AdventureWorks2008	Product	1	KEY	(7900ac71caca)	RangeS-U	GRANT
54	AdventureWorks2008	Product	1	KEY	(6100dc0e675f)	RangeS-U	GRANT
54	AdventureWorks2008	Product	1	KEY	(5700a1a9278a)	RangeS-U	GRANT
54	AdventureWorks2008	Product	1	PAGE	1:16898	IU	GRANT
54	AdventureWorks2008	Product	1	PAGE	1:16899	IU	GRANT
54	AdventureWorks2008	Product	1	PAGE	1:16896	IU	GRANT
54	AdventureWorks2008	Product	1	PAGE	1:16897	IX	GRANT
54	AdventureWorks2008	Product	1	PAGE	1:16900	IU	GRANT
54	AdventureWorks2008	Product	1	PAGE	1:16901	IU	GRANT
54	AdventureWorks2008	Product	1	KEY	(5600c4ce9b32)	RangeS-U	GRANT
54	AdventureWorks2008	Product	1	KEY	(7300c89177a5)	RangeS-U	GRANT
54	AdventureWorks2008	Product	1	KEY	(7f00702ea1ef)	RangeS-U	GRANT
54	AdventureWorks2008	Product	1	KEY	(6b00b8eeda30)	RangeX-X	GRANT
54	AdventureWorks2008	Product	1	KEY	(c500b9eaac9c)	RangeX-X	GRANT
54	AdventureWorks2008	Product	1	KEY	(c6005745198e)	RangeX-X	GRANT
54	AdventureWorks2008	Product	1	KEY	(6a00dd896688)	RangeX-X	GRANT

这里的锁与上一个例子中的那些锁相似，只不过所有的锁都是存在于表本身上（*index_id = 1*）。必须执行一次聚集索引扫描（整张表上），以便所有键最初都获得 RangeS-U 锁，当 4 个数据行最终都被修改后，键上的锁就被转换成 RangeX-X 锁。从上面的结果中可以看到全部的 RangeX-X 锁，但是空间有限（表中共有 504 行数据），没有列出所有的 RangeS-U 锁。

示例 7：新建表

SQL 批处理

```
USE AdventureWorks2008;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN TRAN
SELECT *
INTO newProducts
FROM Production.Product
WHERE ListPrice between 1 and 10;
SELECT * FROM DBlocks
WHERE spid = @@spid;
COMMIT TRAN
```

DBlocks 输出结果（省略版）

spid	dbname	entity_name	index_id	resource	description	mode	status
54	AdventureWorks2008	n/a	NULL	DATABASE		NULL	GRANT
54	AdventureWorks2008	n/a	NULL	DATABASE		NULL	GRANT
54	AdventureWorks2008	n/a	NULL	DATABASE		S	GRANT
54	AdventureWorks2008	n/a	NULL	METADATA	user_type_id = 258	Sch-S	GRANT
54	AdventureWorks2008	n/a	NULL	METADATA	data_space_id = 1	Sch-S	GRANT
54	AdventureWorks2008	n/a	NULL	DATABASE		S	GRANT
54	AdventureWorks2008	n/a	NULL	METADATA	$seq_type = 0, objec	Sch-M	GRANT

```
54   AdventureWorks2008 n/a          NULL     METADATA  user_type_id = 260 Sch-S  GRANT
54   AdventureWorks2008 sysrowsetcol NULL     OBJECT                       IX     GRANT
54   AdventureWorks2008 sysrowsets   NULL     OBJECT                       IX     GRANT
54   AdventureWorks2008 sysallocunit NULL     OBJECT                       IX     GRANT
54   AdventureWorks2008 syshobtcolum NULL     OBJECT                       IX     GRANT
54   AdventureWorks2008 syshobts     NULL     OBJECT                       IX     GRANT
54   AdventureWorks2008 sysserefs    NULL     OBJECT                       IX     GRANT
54   AdventureWorks2008 sysschobjs   NULL     OBJECT                       IX     GRANT
54   AdventureWorks2008 syscolpars   NULL     OBJECT                       IX     GRANT
54   AdventureWorks2008 sysidxstats  NULL     OBJECT                       IX     GRANT
54   AdventureWorks2008 sysrowsetcol 1        KEY      (15004f6b3486)      X      GRANT
54   AdventureWorks2008 sysrowsetcol 1        KEY      (0a00862c4e8e)      X      GRANT
54   AdventureWorks2008 sysrowsets   1        KEY      (000000aaec7b)      X      GRANT
54   AdventureWorks2008 sysallocunit 1        KEY      (00001f2dcf47)      X      GRANT
54   AdventureWorks2008 syshobtcolum 1        KEY      (1900f7d4e2cc)      X      GRANT
54   AdventureWorks2008 syshobts     1        KEY      (000000aaec7b)      X      GRANT
54   AdventureWorks2008 NULL         NULL     RID      1:6707:1            X      GRANT
54   AdventureWorks2008 DBlocks      NULL     OBJECT                       IS     GRANT
54   AdventureWorks2008 newProducts  NULL     OBJECT                       Sch-M  GRANT
54   AdventureWorks2008 sysserefs    1        KEY     (010025fabf73)       X      GRANT
54   AdventureWorks2008 sysschobjs   1        KEY     (3b0042322c99)       X      GRANT
54   AdventureWorks2008 syscolpars   1        KEY     (4200c1eb801c)       X      GRANT
54   AdventureWorks2008 syscolpars   1        KEY     (4e00092bfbc3)       X      GRANT
54   AdventureWorks2008 sysidxstats  1        KEY     (3b0006e110a6)       X      GRANT
54   AdventureWorks2008 sysschobjs   2        KEY     (9202706f3e6c)       X      GRANT
54   AdventureWorks2008 syscolpars   2        KEY     (6c0151be80af)       X      GRANT
54   AdventureWorks2008 syscolpars   2        KEY     (2c03557a0b9d)       X      GRANT
54   AdventureWorks2008 sysidxstats  2        KEY     (3c00f3332a43)       X      GRANT
54   AdventureWorks2008 sysschobjs   3        KEY     (9202d42ddd4d)       X      GRANT
54   AdventureWorks2008 sysschobjs   4        KEY     (3c0040d00163)       X      GRANT
54   AdventureWorks2008 newProducts  0        PAGE     1:6707              X      GRANT
54   AdventureWorks2008 newProducts  0        HOBT                         Sch-M  GRANT
```

实际上，这些锁中只有很少一些是 *newProduct* 表元素上的锁。从 *entity_name* 列可以看到，绝大多数对象都是未记录的系统表名，而且一般都不可见。当创建新表时，SQL Server 会在 9 个不同的系统表上获得锁，以记录关于这个新表的信息。此外，注意新表上的架构修改锁（Sch-M 锁）和其他元数据锁。

最后一个例子用来查看当表上没有聚集索引时，进行数据行更新操作时持有的锁。

示例 8：行锁

SQL 批处理

```
USE AdventureWorks2008;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRAN
UPDATE newProducts
SET ListPrice = 5.99
WHERE name = 'Road Bottle Cage';
SELECT * FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'newProducts';
COMMIT TRAN
```

DBlocks 输出结果

```
spid dbname             entity_name  index_id  resource   description mode     status
---- ------------------ ------------ --------- ---------- ----------- -------- --------
54   AdventureWorks2008 newProducts  NULL      OBJECT                 IX       GRANT
```

```
54  AdventureWorks2008 newProducts  0          PAGE         1:6708      IX         GRANT
54  AdventureWorks2008 newProducts  0          RID          1:6708:5    X          GRANT
```

newProducts 表上没有索引，所以实际符合搜索标准的行（RID）上存在的是一个独占锁（X 锁）。对于 RID 锁，*description* 列实际上是以“文件号:页码:槽号”的形式来表示特定的行。正如预料中的那样，IX 锁出现在页和表上。

10.4 锁兼容性

如果同一资源上还有别的进程持有的锁时，另一个锁也能被授予，那么这两个锁就是兼容的。如果所请求的针对某个资源的锁与当前持有的锁不兼容，那么请求连接就必须等待这个锁被释放。例如，如果一个共享页锁存在于一页上，另一个进程请求该页上的共享页锁时会被同意，因为两个锁类型是兼容的。但是请求该页上独占锁的进程不会被同意，因为独占锁与已经持有的共享锁不兼容。图 10-2 总结了 SQL Server 2008 中锁的兼容性。上面一行是一个进程可能已经持有的锁模式，左侧一栏是另一个进程可能请求的锁模式。

	NL	SCH-S	SCH-M	S	U	X	IS	IU	IX	SIU	SIX	UIX	BU	RS-S	RS-U	RI-N	RI-S	RI-U	RI-X	RX-S	RX-U	RX-X
NL	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
SCH-S	N	N	C	N	N	N	N	N	N	N	N	N	N	I	I	I	I	I	I	I	I	I
SCH-M	N	C	C	C	C	C	C	C	C	C	C	C	C	I	I	I	I	I	I	I	I	I
S	N	N	C	N	N	C	N	N	C	N	C	C	C	N	N	N	N	N	C	N	N	C
U	N	N	C	N	C	C	N	C	C	C	C	C	C	N	C	N	N	C	C	N	C	C
X	N	N	C	C	C	C	C	C	C	C	C	C	C	C	C	N	C	C	C	C	C	C
IS	N	N	C	N	N	C	N	N	N	N	N	N	C	I	I	I	I	I	I	I	I	I
IU	N	N	C	N	C	C	N	N	N	N	N	C	C	I	I	I	I	I	I	I	I	I
IX	N	N	C	C	C	C	N	N	N	C	C	C	C	I	I	I	I	I	I	I	I	I
SIU	N	N	C	N	C	C	N	N	C	N	C	C	C	I	I	I	I	I	I	I	I	I
SIX	N	N	C	C	C	C	N	N	C	C	C	C	C	I	I	I	I	I	I	I	I	I
UIX	N	N	C	C	C	C	N	C	C	C	C	C	C	I	I	I	I	I	I	I	I	I
BU	N	N	C	C	C	C	C	C	C	C	C	C	N	I	I	I	I	I	I	I	I	I
RS-S	N	I	I	N	N	C	I	I	I	I	I	I	I	N	N	C	C	C	C	C	C	C
RI-U	N	I	I	N	C	C	I	I	I	I	I	I	I	N	C	C	C	C	C	C	C	C
RI-N	N	I	I	N	N	N	I	I	I	I	I	I	I	C	C	N	N	N	N	C	C	C
RI-S	N	I	I	N	N	C	I	I	I	I	I	I	I	C	C	N	N	N	C	C	C	C
RI-U	N	I	I	N	C	C	I	I	I	I	I	I	I	C	C	N	N	C	C	C	C	C
RI-X	N	I	I	C	C	C	I	I	I	I	I	I	I	C	C	N	C	C	C	C	C	C
RX-S	N	I	I	N	N	C	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C	C
RX-U	N	I	I	N	C	C	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C	C
RX-X	N	I	I	C	C	C	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C	C

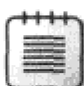
图 10-2 SQL Server 锁兼容性矩阵

持有的锁和请求的锁的交叉点上，有 3 个可能的值。N 表示没有冲突；C 表示会有冲突，请求锁的进程必须等待；/表示永远不会出现的无效组合。表格中的所有“/”都涉及范围锁，而范围锁只适用于键资源，所以任何不适用于键资源的锁类型都是无效匹配。

锁兼容性问题也出现在不同资源上的锁之间，比如表锁和页锁。表和页显然代表了一种隐含的层次关系，因为表是由多个页组成的。如果一个表页上存在独占页锁，那么另一个进程就无法获得该表上的共享表锁。这种层次关系是用意向锁来保护的。进程要获得独占页锁、更新页锁或意向独占页锁，首先得获得表上的意向独占锁。该意向独占表锁防止另外的进程获得该表上的共享表锁（记住，同一资源上的意向独占锁和共享锁是不兼容的）。

类似地，进程要获得共享行锁，必须首先获得该表的意向共享锁，以防止别的进程获得独占表锁。或者，如果独占表锁已经存在，那么意向共享锁就不会被同意，共享页锁必须等待，直到独占表锁被释放。没有意向锁的话，进程 A 可以用独占页锁锁定表中的一页，进程 B 可以在同一表上放置独占表锁，

从而有权修改整个表，包括进程 A 已经独占锁定的那一页。



注意：

显然，锁兼容性只有在锁影响同一对象时才是一个问题。例如，两个或多个进程可以分别同时持有独占页锁，只要锁是加在不同的页或不同的表上。

即使两个锁兼容，如果有一个不兼容的锁在等待，那么第二个锁的请求者也必须等待。例如，假设进程 A 持有一个共享页锁。进程 B 请求一个独占页锁并且必须等待，因为共享页锁与独占页锁不兼容。进程 C 请求一个共享页锁，与已经授予给进程 A 的共享页锁兼容。但是，共享页锁不能被立即授予。进程 C 必须等待它的共享页锁，因为进程 B 在锁队列中排在它的前面并且请求的独占页锁与进程 C 请求的共享页锁不兼容。

通过考虑与已被授予锁的进程的锁兼容性，以及与正在等待的进程的锁兼容性，SQL Server 防止了锁饥饿现象（当对共享锁的请求太多，以至于对独占锁的请求得不到理会时，就会出现锁饥饿）。

10.5 锁定内部架构

锁不是磁盘结构。不会在数据页或表头上直接找到锁字段，跟踪锁的元数据也不会被写到磁盘上。锁是内部存储结构——它们消耗用于 SQL Server 的存储部分。锁由锁资源识别，锁资源描述被锁定的资源（行、索引键、页或表）。为了跟踪数据库、锁的类型和描述被锁定资源的信息，每个锁在 32 位系统上需要 64 字节内存，在 64 位系统上需要 128 字节内存。这种 64 字节或 128 字节的结构叫做锁块。

每个持有锁的进程也必须具有一个*锁所有者*，这代表锁与请求或持有该锁的实体之间的关系。锁所有者在 32 位系统上需要 32 字节内存，在 64 位系统上需要 64 字节内存。这种 32 字节或 64 字节的结构叫做锁所有者块。一个事务可以具有多个锁所有者块；有时一个可滚动游标使用好几个锁所有者块。同样，一个锁也可以具有很多锁所有者块，就跟共享锁的情形一样。前面已经提到，锁所有者代表锁与实体之间的关系，这个关系可以是已授予、在等待或者在*等待转换*。

锁管理程序维护着一个锁哈希表。锁资源包含在锁块中，被进行哈希算法，以确定其在哈希表中的目标哈希槽。所有经过哈希算法确定到同一哈希槽的锁块都同哈希表中的一项链接在一起。每个锁块包含一个 15 字节的字段，用于描述锁定的资源。锁块也包含到锁所有者块列表的指针。在 3 种状态中，分别都有一个单独的锁所有者列表。图 10-3 显示了一般的锁架构。

哈希表中的槽数取决于系统的物理内存，如表 10-7 所示。上限是 2^{31} 个槽。同一台机器上 SQL Server 的所有实例都具有一个槽数相同的哈希表。锁哈希表中的每一项，大小是 16 字节，并且由一个指向锁块列表的指针和一个保证对同一槽进行序列化访问的旋转锁组成。

表 10-7　　内部锁哈希表的槽数

物理内存（MB）	槽　数	使用的内存
<32	$2^{14}=16384$	128 KB
>=32 且 <64	$2^{15}=32768$	256 KB
>=64 且 <128	$2^{16}=65536$	512 KB
>=128 且<512	$2^{18}=262144$	2048 KB

续表

物理内存（MB）	槽　数	使用的内存
>= 512 且 < 1024	$2^{19} = 524288$	4096 KB
>= 1024 且< 4096	$2^{21} = 2097152$	16384 KB
>= 4096 且 < 8192	$2^{22} = 4194304$	32768 KB
>= 8192 且< 16384	$2^{23} = 8388608$	65536 KB
>= 16384	$2^{25} = 33554432$	262144 KB

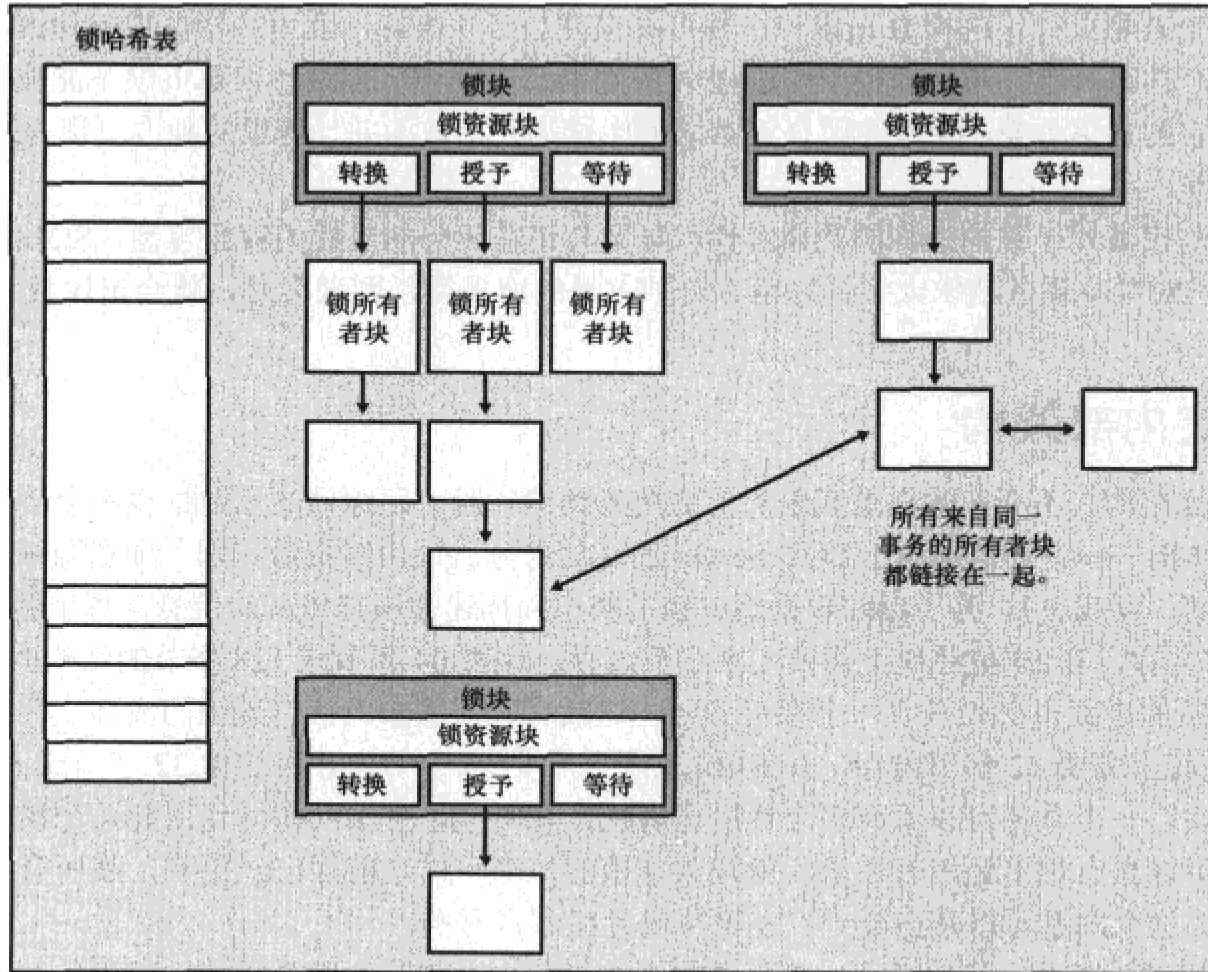


图 10-3　SQL Server 锁定架构

锁管理程序在服务器启动时提前分配很多的锁块和锁所有者块。在 NUMA 配置上，这些锁块和锁所有者块在所有 NUMA 模式之间分配。所以当来一个锁请求时，使用的是本地锁块。如果锁的数量已由 *sp_configure* 进行设置，那么它会分配这个配置好的锁块数和相同数量的锁所有者。如果数量不固定（0 表示自动调整），它会为您的 SQL Server 实例分配 2500 个锁块。它分配两倍的锁所有者块（2 * # 锁块）。最大数量是，静态分配不能消耗超过已提交缓冲池大小的 25%。

当来了锁请求却没有空闲的锁块时，锁管理程序就动态地分配新锁块，而不是拒绝锁请求。锁管理程序与全局内存管理程序合作，协商服务器分配的内存。必要时，锁管理程序可以释放动态分配的锁块。锁管理程序被限制为将缓冲区管理器已提交的目标大小分配给锁块和锁所有者块。

10.5.1　锁分区

对于大型系统，经常引用的对象上的锁会变成性能瓶颈。获得和释放锁的过程会导致内部锁定资源上的争用。锁分区通过将单个锁资源分解成多个锁资源，增强了锁定性能。对于有 16 个或更多 CPU 的

系统，SQL Server 自动将某些锁分解成多个锁资源，每个 CPU 一个。这就叫做锁分区，并且用户无法控制这一过程（不要将锁分区与分区锁混淆，后者将在本章后面“锁升级”一节中讨论)。每当锁分区活动时，就会有一条信息性消息发送到错误日志。该错误消息是“锁分区被启用。这只是一条信息性消息。用户无需采取操作”。锁分区只适用于以下锁模式的全对象锁（例如，表和视图)：S、U、X 和 SCH-M。所有其他模式（NL、SCH_S、IS、IU 和 IX）都在单个 CPU 上获得。SQL Server 在事务开始时给每个事务分配一个默认的锁分区。在这个事务存续期间，遍布所有分区的所有锁请求都使用分配给该事务的分区。通过这种方法，不同事务对同一对象的锁资源的访问分布于不同分区。

sys.dm_tran_locks 中的 *resource*_lock_partition 列指出一个特定的锁位于哪个锁分区上，所以您可以看到针对同一资源的多个锁具有不同的 *resource_lock_partition* 值。对于少于 16 个 CPU 的系统，没有使用锁分区，*resource_lock_partition* 值总是为 0。

例如，考虑一个获得 REPEATABLE READ 级别 IS 锁的事务，所以该 IS 锁在事务存续期间一直被持有。该 IS 锁是在事务的默认分区（例如，分区 4）获得的。如果另一个事务试图获得同一表上的 X 锁，那么该 X 锁必须在所有分区上被获得。SQL Server 成功地在分区 0 到 3 上获得 X 锁，但是它试图在分区 4 上获得 X 锁时被阻塞。在 5 到 15 的分区 ID 上（这些分区还没有获得该表的 X 锁)，其他事务可以继续获得任何不会导致阻塞的锁。

利用锁分区，SQL Server 在多个旋转锁之间分散检查锁的负载，对任何给定旋转锁的大多数访问都来自同一个 CPU（因而实际上总是来自同一个节点)，这意味着旋转锁不应该经常旋转。

10.5.2 锁块

锁块是 SQL Server 的锁定架构中的关键结构，参见前面图 10-3。锁块包含以下信息。

- 锁资源信息，其中包含锁资源名称和有关锁的详细信息。
- 将锁块与锁哈希表连接在一起的指针。
- 指向该资源上已经授予的锁的锁所有者块列表的指针。维护着 4 个*授予列表*，以最小化找到已授予锁所花的时间。
- 指向该资源上正在等待转换成另一个锁模式的锁的锁所有者块列表的指针。这个列表叫做*转换列表*。
- 指向该资源上已经请求但还未授予的锁的锁所有者块列表的指针。这个列表叫做*等待列表*。

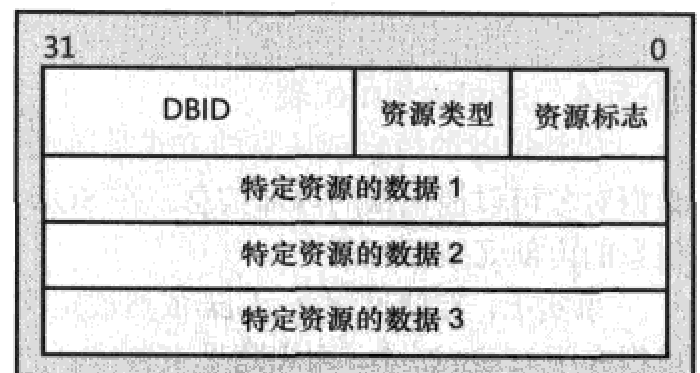


图 10-4 锁资源的结构

锁资源唯一识别被锁定的数据。它的结构显示在图 10-4 中。图中的每一行代表 4 字节或者 32 位。

图 10-4 所示字段的含义描述在表 10-8 中。*资源类型*字节中的值是前面表 10-5 中描述的锁定资源之一。资源类型后面括号中的数字是该资源类型的代码编号（在本章稍后的 *syslockinfo* 表中我们会看到)。3 个数据字段中的值的含义依所描述的资源类型的不同而不同。SR 表示子资源（我们马上就会介绍)。

表 10-8 锁资源块中的字段

资源类型	资源内容数据 1	数据 2	数据 3
数据库（2)	SR	0	0
文件（3)	文件 ID	0	0

续表

资源类型	资源内容数据 1	数据 2	数据 3
索引（4）	对象 ID	SR	索引 ID
表（5）	对象 ID	SR	0
页（6）	页码		0
键（7）	分区 ID	哈希键	
扩展（8）	扩展 ID		0
RID（9）	RID		0

下面是一些可能的 SR（SubResource）值。如果锁是在数据库资源上，那么 SR 表示以下值之一：

- 全数据库锁；
- 批量操作锁。

如果锁是在表资源上，那么 SR 表示以下值之一：

- 全表锁（默认）；
- 更新统计锁；
- 编译锁。

如果锁是在索引资源上，那么 SR 表示以下值之一：

- 全索引锁（默认）；
- 索引 ID 锁；
- 索引名称锁。

10.5.3　锁所有者块

每个由会话拥有或等待的锁都表示在一个锁所有者块中。锁所有者块的列表形成组成锁块的授予、转换盒等待列表。已授予锁的每个锁所有者块与同一事务的所有其他锁所有者块链接在一起，所以它们在事务或会话结束时能一起被释放。

10.5.4　syslockinfo 表

尽管推荐的检索锁信息的方式是通过 *sys.dm_tran_locks* 视图，但是还有另一种叫做 *syslockinfo* 的元数据对象可以提供锁的内部信息。在 SQL Server 2005 中引入 DMV 之前，*syslockinfo* 是唯一用于检索锁信息的内部元数据变量。

事实上，存储过程 *sp_lock* 依然被定义为从 *syslockinfo* 而不是从 *sys.dm_tran_locks* 检索信息。我们将 [illegible] *syslockinfo* [illegible] 该表的 [illegible] 信息都 [illegible] *sys.dm_tran_locks* [illegible]

[illegible] 很多 *syslockinfo* 在 *master* 数据库中可用。您可以看一下 [illegible] 列要 [illegible] 就是 [illegible] 含关于某个被锁定资源的 16 字节的描述。

可以将 *syslockinfo.rsc_bin* 字段作为资源块分析。我们来看一个例子。我使用 REP[illegible] 隔离级别从 *AdventureWorks2008* 中的 *Person* 表选择单个行，所以我的共享锁在该事务[illegible]期间继续持有。我[illegible]然后查看 *syslockinfo* 中的 *rsc_bin* 列，获得关于键锁、页锁和表锁的信息。

```
USE AdventureWorks2008
GO
```

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
GO
BEGIN TRAN
SELECT * FROM Person.Person
WHERE BusinessEntityID = 249;
GO
SELECT rsc_bin, rsc_type
FROM master..syslockinfo
WHERE rsc_type IN (5,6,7);
GO
```

下面是结果中的 3 行：

```
rsc_bin                            rsc_type
---------------------------------- --------
0x805EFA59000000000000000007000500 5
0x19050000010000000000000007000600 6
0x710000000001F900CE79D52507000700 7
```

rsc_bin 中的最后 2 字节是资源模式，所以在字节交换之后，可以在 *rsc_type* 中看到相同的值——例如，0500 经过字节交换后成为 0005，资源模式就是 5（表锁）。往前 2 字节是数据库 ID，对于所有这 3 行，字节交换后的值都是 0007，这就是我的 *AdventureWorks2008* 数据库的数据库 ID。

其余字节会根据资源的类型而不同。对于表来说，前 4 字节表示对象 ID。对象锁（*rsc_type = 5*）的第一行完成字节交换后的值是 59FA5E80，换算成十进制是 1 509 580 416。我可以将该值转换成一个对象名称，如下所示：

```
SELECT object_name(1509580416)
```

结果显示的是 *Person* 表。

对于页来（*rsc_type = 6*）说，前 6 字节是页码，后面是文件号。经过字节交换后，文件号是 0001（十进制就是 1），页码是 00000519（十进制是 9889）。所以，锁是加在文件 1 的第 1305 页上。

最后，对于键（*rsc_type = 7*），前 6 字节是分区 ID，但是转换要稍微复杂一点。我们需要在经过字节交换后的值后面再添加 2 字节的 0，得到 0100000000710000，换算成十进制是 72 057 594 045 333 504。为了知道该分区属于哪个对象，可以查询 *sys.partitions* 视图：

```
SELECT object_name(object_id)
FROM sys.partitions
WHERE partition_ID = 72057594045333504;
```

同样，结果是该分区是 *Person* 表的一部分。键资源的 *rsc_bin* 的下面 6 字节是 F900CE79D525。这是一个字符字段，所以不需要进行字节交换。但是其值不再是可解读的。键锁具有一个根据索引的所有键列而为其生成的哈希值。索引可以很长，所以对于几乎任何可能的数据类型，SQL Server 都需要一种一致的方式来跟踪哪些键被锁定了。哈希函数因此生成一个 6 字节的哈希字符串来表示键。尽管不能对该值进行反向工程，确切判断哪个索引行被锁定了，但是可以使用它来查找匹配项，就像 SQL Server 所做的那样。如果两个 *rsc_bin* 值具有相同的 6 字节哈希字符串，那么它们引用的是同一个锁资源。

除了检测对同一锁资源的引用之外，还可以通过使用*%%lockres%%*值（该值可以返回任何键的哈希字符串）来判断哪些特定的键被锁定了。根据用于检索数据的索引，从表中选择该值和数据会在结果集中的每行返回锁资源。考虑下面这个例子，它在一个小表上创建聚集索引和非聚集索引，然后选择每行的*%%lockres%%*值，首先使用的是聚集索引，然后使用的是非聚集索引：

```
CREATE TABLE lockres (c1 int, c2 int);
GO
INSERT INTO lockres VALUES (1,10);
INSERT INTO lockres VALUES (2,20);
INSERT INTO lockres VALUES (3,30);
GO
CREATE UNIQUE CLUSTERED INDEX lockres_ci ON lockres(c1);
CREATE UNIQUE NONCLUSTERED INDEX lockres_nci ON lockres(c2);
GO
SELECT %%lockres%% AS lock_resource, * FROM lockres WITH (INDEX = lockres_ci);
SELECT %%lockres%% AS lock_resource, * FROM lockres WITH (INDEX = lockres_nci);
GO
```

得到以下结果。第一组的行展示了聚集索引键的锁资源，第二组展示了非聚集索引键的锁资源：

```
lock_resource                     c1          c2
--------------------------------  ----------- -----------
(010086470766)                    1           10
(020068e8b274)                    2           20
(03000d8f0ecc)                    3           30
lock_resource                     c1          c2
--------------------------------  ----------- -----------
(0a0087c006b1)                    1           10
(14002be0c001)                    2           20
(1e004f007d6e)                    3           30
```

可以使用该锁资源找到表中的哪一行匹配锁定的资源。例如，如果 *sys.dm_tran_locks* 表示具有锁资源（010086470766）的行持有 lockres 表中的锁，那么可以利用以下查询找出该资源对应于哪一行：

```
SELECT * FROM lockres
WHERE %%lockres%% = '(010086470766)'
```

注意，如果表是一个堆，并且我们在扫描表的时候查找锁资源，那么锁资源就是实际的行 ID（RID）。返回的值看起来就像是特殊值*%%physloc%%*，这在第 5 章已经介绍过了。

```
CREATE TABLE lockres_on_heap (c1 int, c2 int);
GO
INSERT INTO lockres_on_heap VALUES (1,10);
INSERT INTO lockres_on_heap VALUES (2,20);
INSERT INTO lockres_on_heap VALUES (3,30);
GO
SELECT %%lockres%% AS lock_resource, * FROM lockres_on_heap;
```

下面是结果：

```
lock_resource                     c1          c2
--------------------------------  ----------- ----
1:169:0                           1           10
1:169:1                           2           20
1:169:2                           3           30
```

警告：

在试图找到表中哈希字符串匹配特定锁资源的行时要小心。这些查询必须执行一次完全的表扫描，以找到您关注的行，如果表比较大，这个过程代价会非常大。

10.6 行级别锁与页级别锁

尽管 SQL Server 2008 完全支持行级别锁，但是有些情况下，锁管理者会决定不锁定单个行，而锁定页或整个表。在另外一些情况下，很多较小的锁升级成表锁，下一小节中将会讨论。

在 SQL Server 7.0 之前，SQL Server 可以锁定的最小数据单元是页。尽管很多人说这是无法接受的，并且不可能在锁定整个页的同时保持良好的一致性，但是有很多大型且强大的应用程序都是只使用页级别锁定而编写和部署的。如果能够得到良好的设计和调优，并发性并不是问题，这些应用程序中有一些支持数百个活动用户连接，并且响应时间和吞吐量都还不错。但是对于 SQL Server 7.0，如果页大小从 2 KB 改变为 8 KB，问题就严重了。锁定整个页意味着锁定的数据是以前版本锁定的数据的 4 倍。从 SQL Server 7.0 开始，该软件实现了完全行级别锁定，所以任何由于对较大页大小的低并发性而引起的潜在问题应该都不是问题了。但是，锁定不是免费的。需要资源来管理锁。回想一下，锁是一种 64 或 128 字节（分别对应于 32 位或 64 位机器）的内存结构，还有另外 32 或 64 字节用于每个持有或请求锁的进程。如果每行都需要一个锁，要扫描数百万行，那么仅为这一个进程持有锁就需要使用 64 MB 的 RAM。

除了内存消耗问题，锁定还是一个相当处理密集的操作。管理锁需要缜密的记录功能。回想一下，在内部，SQL Server 使用一个叫做旋转锁的轻量级互斥锁来保护资源，使用闩锁（也比常规锁要轻）来保护非页级别索引页。这些性能优化避免了完全锁定带来的开销。如果一个数据页包含 50 行数据，并且所有行都要用到，那么显然，在页上发出并管理一个锁，比在行上管理 50 个锁要高效多了。这是页锁定的明显优势——减少了必须存在和管理的锁结构的数量。

我们假设有两个进程都需要更新不多的几行数据，尽管数据行不完全相同，但是有些行刚好在同一页上。利用页级别锁定，一个进程必须等待，直到另一个进程的页锁被释放。而如果使用行锁定的话，另一个进程就不需要等待。较小的锁粒度意味着不会一开始就出现冲突，因为每个进程关心的是不同的行。这是行级别锁定的明显优势。谁的明显优势更胜一筹呢？这并不好下结论，跟应用程序和数据都有关系。每种类型的锁定都有适用的应用程序类型和用途。

ALTER INDEX 语句可用于手动控制索引中的锁定单元，有不同的选项用于在索引中禁用页锁或行锁。由于这些选项只对索引可用，因而无法在堆的数据页中控制锁定（但是请记住，如果表具有聚集索引，数据页就是索引的一部分，就会受到用 ALTER INDEX 设置的值的影响）。索引选项分别针对每个表或索引而设定。ALLOW_ROW_LOCKS 和 ALLOW_PAGE_LOCKS 这两个选项最初对每个表和索引都设置为 ON。如果这两个选项对一个表都设置为 OFF，那么就只允许表锁。

正如前面所提到的，在优化过程中，SQL Server 决定最初锁定行、页或整个表。行（或键）的锁定是最受欢迎的。锁定类型的选择取决于要扫描的行和页数、一页上的行数、起作用的隔离级别、所进行的更新活动、系统上需要使用内存的用户数等。

10.6.1 锁升级

SQL Server 可以自动将行锁、键锁或页锁适当升级为表锁或分区锁。这种升级可以保护系统资源（防止系统将过多的内存用于跟踪锁）并提高效率。例如，在一个查询获得很多行锁之后，锁级别会被升级，因为获得并持有单个锁可能比持有很多行锁更有意义。发生锁升级时，较小单元（行或页）上的很多锁被释放，并被较大单元上的一个锁取代。由于可用于锁结构的内存数量是有限的，所以有时候，升级是很必要的，可以确保用于锁的内存保持在合理的限度内。

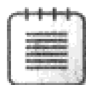
SQL Server 中默认为升级到表锁。但是 SQL Server 2008 能够使用 *ALTER TABLE* 语句升级到单个分区。*ALTER TABLE* 的 *LOCK_ESCALATION* 选项可以指定总是升级到表级别，或者指定为既可升级到表级别，也可升级到分区级别。*LOCK_ESCALATION* 选项也可以用来完全阻止升级。下面有一个例子，就是更改 *TransactionHistory* 表(在第 7 章中运行分区例子时已经创建好了)，以便锁可以升级到表或分区级别：

```
ALTER TABLE TransactionHistory
SET (LOCK_ESCALATION = AUTO);
```

锁升级发生在以下情形。

- 单个语句在一个对象上或者对象的一个分区上持有的锁的数量超出阈值。该阈值当前是 5000 个锁，但是未来的服务包中可能会改变。如果同一语句持有的锁是加在多个对象上（比如说 3000 个锁在一个索引上，还有 3000 个锁在另一个索引上），此时不会发生锁升级。
- 锁资源占用的内存超过非 AWE（32 位）或常规（64 位）已启用内存的 40%并且锁配置选项设置为 0（在这种情况下，锁内存是根据需要自动分配的，所以这个 40%不是常量）。如果锁选项设置为非 0 值，那么为锁保留的内存是在 SQL Server 启动时静态分配的。SQL Server 使用超过 40%的为锁资源保留的锁内存时，就会发生锁升级。

当锁升级被触发时，如果存在冲突锁，升级尝试可能会失败。举个例子，假设 RID 上的 X 锁需要升级，而同一表或分区上有另外一个进程持有的并发 X 锁，那么锁升级尝试就会失败。但是，每当事务在同一对象上获得另外 1250 个锁时，SQL Server 就会继续尝试将锁升级。如果锁升级成功，SQL Server 就会释放索引或堆上的所有行锁和页锁。



注意：

SQL Server 从来不升级到页锁。锁升级的结果总是表锁或分区锁。此外，多个分区锁从来不会升级到表锁。

控制锁升级

锁升级可潜在地导致在对象上的行锁或页锁的事务未来对索引或堆的并发访问。在发出新请求时，SQL Server 不能将锁进行降级。所以，锁升级并不总是对所有应用程序都是一个好主意。

SQL Server 2008 也支持使用 *ALTER TABLE* 语句为单个表禁用锁升级。下面这个例子禁用 *TransactionHistory* 表上的锁升级：

```
ALTER TABLE TransactionHistory
SET (LOCK_ESCALATION = DISABLE);
```

SQL Server 2008 也支持使用跟踪标志禁用锁升级。注意，这些跟踪标志影响一个 SQL Server 实例中所有数据库的所有表上的锁升级。

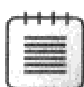
- 跟踪标志 1211 完全禁用锁升级。它指示 SQL Server 忽视锁管理程序获得的内存，直到达到最大静态分配的锁内存（使用锁配置选项指定的）或者非 AWE（32 位）或常规（64 位）动态分配内存的 60%。这时，会产生一个超出锁内存错误。使用该跟踪标志时应该极为小心，因为设计不好的应用程序会耗尽内存，并严重降低 SQL Server 实例的性能。
- 跟踪标志 1224 也基于获得的锁的数量禁用锁升级，但是它允许基于内存消耗进行锁升级。在锁

管理程序获得40%的静态分配内存（按照锁选项）或者40%的非AWE（32位）或常规（64位）动态分配内存时，它启用锁升级。应该注意，如果由于内存被其他组件使用，导致SQL Server不能为锁分配内存，那么锁升级会被提早触发。与跟踪标志1211一样，在分配给锁管理程序的内存超过总静态分配内存或60%的非AWE（32位）或常规（64位）动态分配内存时，SQL Server产生一个超出锁内存错误。

如果两个跟踪标志（1211和1224）同时设置，那么跟踪标志1211优先。记住，这些跟踪标志影响整个SQL Server实例。在很多时候，迫切希望在对象级别控制升级阈值，所以应该尽可能考虑使用 *ALTER TABLE* 命令。

10.6.2 死锁

当两个进程都在等待一个资源，又相互阻止对方获得该资源时，就会发生死锁。一个真正的死锁是Catch-22，不加干涉的话，无论哪个进程都不能前进。发生死锁时，SQL Server会自动干预。我所指的死锁主要是指冲突锁，尽管死锁也可发现于工人线程、内存和并行查询资源上。



注意：

简单的等待锁不是死锁。当持有锁的进程完成时，等待进程就可以获得锁。在多用户系统中，锁等待是正常、希望且必要的。

在SQL Server中，可发生两种主要类型的死锁：循环死锁和转换死锁。图10-5展示了一个循环死锁的例子。进程A开始一个事务，获得 *Product* 表上的一个独占表锁，并请求 *PurchaseOrderDetail* 表上的独占表锁。同时，进程B也开始一个事务，获得 *PurchaseOrderDetail* 表上的一个独占表锁，并请求 *Product* 表上的独占表锁。两个进程被死锁锁住，互不相让。每个进程都持有一个对方需要的资源。谁都无法前进，不加干预的话，就双双永远处于死锁中。像下面这样，可以在SQL Server Management Studio中真正产生死锁。

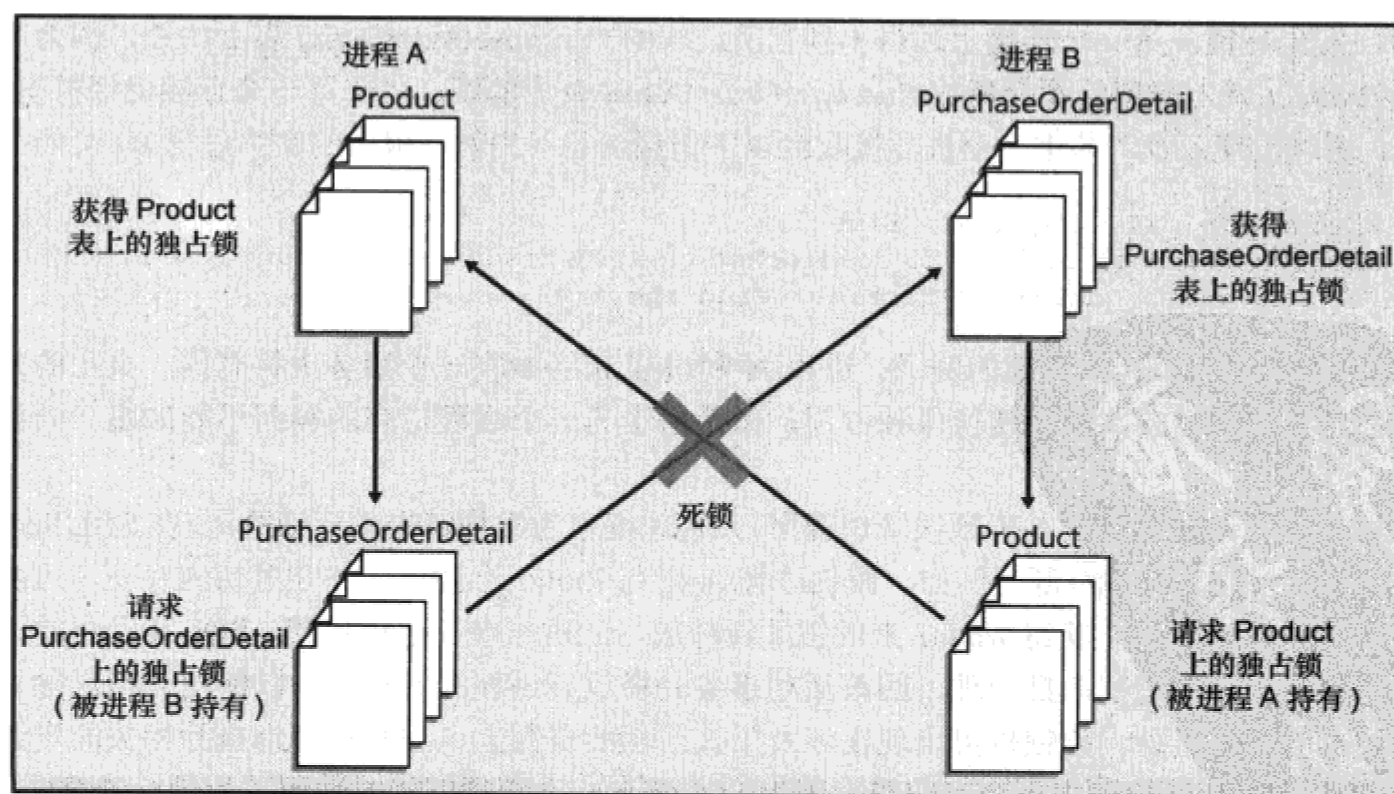


图 10-5 由两个进程产生的死锁，每个进程都持有对方需要的资源

（1）打开一个查询窗口，将数据库上下文更改到 *AdventureWorks2008* 数据库。为进程 A 执行以下批处理：

```
BEGIN TRAN
UPDATE Production.Product
    SET ListPrice = ListPrice * 0.9
WHERE ProductID = 922;
```

（2）打开第二个窗口，为进程 B 执行以下批处理：

```
BEGIN TRAN
UPDATE Purchasing.PurchaseOrderDetail
    SET OrderQty = OrderQty + 200
    WHERE ProductID = 922
    AND PurchaseOrderID = 499;
```

（3）回到第一个窗口，并执行下面这个 *UPDATE* 语句：

```
UPDATE Purchasing.PurchaseOrderDetail
    SET OrderQty = OrderQty - 200
    WHERE ProductID = 922
    AND PurchaseOrderID = 499;
```

此时，查询应该被阻塞，但是还没有死锁。它在等待 *PurchaseOrderDetail* 表上的锁，但是没有理由怀疑它最终得不到该锁。

（4）回到第二个窗口，并执行下面这个 *UPDATE* 语句：

```
UPDATE Production.Product
    SET ListPrice = ListPrice * 1.1
    WHERE ProductID = 922;
```

此时，发生死锁。第一个连接永远得不到它所请求的 *PurchaseOrderDetail* 表上的锁，因为第二个连接没得到 *Product* 表上的锁就不会释放 *PurchaseOrderDetail* 表上的锁。由于第一个连接已经持有 *Product* 表上的锁，所以出现死锁。其中一个进程接收到以下错误消息（当然，报告的实际进程 ID 可能会不同）。

```
Msg 1205, Level 13, State 51, Line 1
Transaction (Process ID 57) was deadlocked on lock resources with another process and has
been chosen as the deadlock victim. Rerun the transaction.
```

图 10-6 展示了一个转换死锁的例子。进程 A 和进程 B 各自在一个事务中持有同一页上的共享锁。每个进程都想要将自己的共享锁转换成独占锁，但是由于另一个进程持有的锁而不能如愿。同样，需要进行干涉。

SQL Server 自动检测死锁并干预锁管理程序，锁管理程序为常规锁提供死锁检测。在 SQL Server 2008 中，死锁也可涉及锁之外的资源。例如，假设进程 A 持有 *Table1* 上的锁，并在等待内存变为可用，而进程 B 占有一些内存，要到它获得 *Table1* 上的锁才能释放，此时出现了进程死锁。SQL Server 检测到一个死锁时，它将终止一个进程的批处理，回滚活动事务并释放该进程的所有锁，以解决死锁。除了锁资源和内存资源上的死锁之外，死锁也可出现在涉及工人线程的资源上、与并行查询执行相关的资源上，以及 MARS 资源上。死锁检测中不会有闩锁，因为 SQL Server 在获得闩锁时使用了死锁检验算法。

在 SQL Server 中，一个叫做 LOCK_MONITOR 的独立线程每 5 秒钟检查一次系统中是否有死锁。当

发生死锁时，死锁检测间隔就会缩短，最短可达到 100 毫秒。事实上，检测到死锁之后得不到满足的前几个锁请求将立即触发死锁搜索，而不是等待下一个死锁检测间隔。如果死锁频率降低，间隔又会回到 5 秒钟。

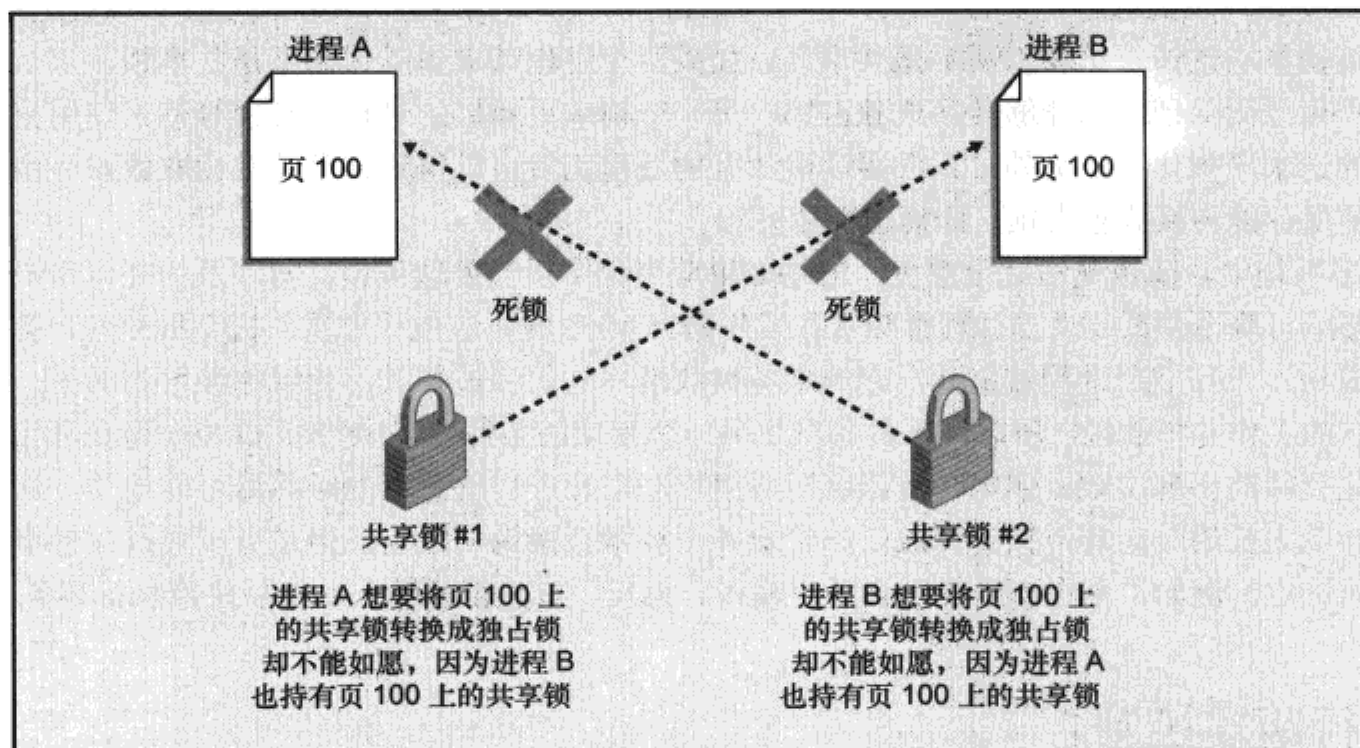
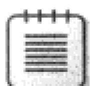


图 10-6 由两个进程产生的转换死锁，每个进程想要在事务中提升其在同一资源上的锁

这个 LOCK_MONITOR 线程通过检测等待锁列表中是否有循环来检测死锁，有循环表示持有锁的进程和等待锁的进程之间存在循环关系。SQL Server 从考虑已经完成的工作进程数出发，试图选择回滚代价最低的进程作为牺牲品。牺牲品进程被杀死，错误消息 1205 被发送到相应的客户端连接。事务回滚，意味着它的所有锁都被释放，所以死锁中涉及的其他进程就可以继续前进。但是，某些操作被标记为黄金或者不可被杀死，不能被选作死锁牺牲品。例如，回滚事务中涉及的进程不能被选作死锁牺牲品，因为被回滚的更改会处于不确定状态，导致数据损坏。

使用 SET DEADLOCK_PRIORITY 语句，可以决定进程在死锁时被选作牺牲品的优先级。一共有 21 种不同的优先级别，从–10 到 10。您也可以指定值为 LOW（等于–5）、NORMAL（等于 0）和 HIGH（等于 5）。具体哪个会话被选作死锁牺牲品取决于每个会话的死锁优先级。对于死锁优先级不同的几个会话，具有最低死锁优先级的会话被选作死锁牺牲品。如果两个会话具有相同的死锁优先级，那么 SQL Server 选择回滚代价低的会话作为牺牲品。



注意：

内部使用的轻量级闩锁和旋转锁不具有死锁检测服务。相反，闩锁和旋转锁上的死锁是事先避免而不是事后解决。避免的手段是 SQL Server 开发团队必须使用严格的编程指导方针。这些轻量级锁必须在一个层次结构中获得，并且进程在持有闩锁或旋转锁时不必等待常规锁。例如，一个编码规则是，持有旋转锁的进程不必直接等待锁或者调用其他必须等待锁的服务，并且不能请求在更高层次中获得的旋转锁。通过为开发团队针对 SQL Server 对象被访问的顺序建立类似的指导方针，可以事先主动避免死锁。

在图 10-5 的例子中，如果进程之间事先协商好（比如说，它们决定总是先访问 *Product* 表再访问 *PurchaseOrderDetail* 表），就可以避免发生循环死锁。然后，其中一个进程得到最先被访问的表上的初始独占锁，另一个进程等待锁被释放。进程等待锁是正常而自然不过的事情。记住，等待不是死锁。

针对进程访问表的顺序，应该总是具有一个标准协议。如果知道进程可能需要在读取行之后更新它，那么进程最初就应该请求更新锁而不是共享锁。如果两个进程请求更新锁而不是共享锁，那么被授予更新锁的进程可以确保锁稍后能够转换成独占锁。另一个请求更新锁的进程则必须等待。使用更新锁让针对独占锁的请求序列化了。另外，只需要读取数据的进程仍然可以得到共享锁并读取数据。由于更新锁的持有者被保证能转换成独占锁，因而避免了死锁。

在很多系统中，死锁无法完全避免，但是如果应用程序适当处理死锁，对涉及的任何用户及系统其余部分的影响可降至最低（适当处理意味着在发生错误 1205 时，应用程序重新提交批处理，第二次尝试大多能够成功。一旦一个进程被杀死，它的事务被取消，它的锁被释放，死锁中涉及到的另一个进程就可以完成它的工作并释放锁，所以就不具备产生另一个死锁的条件了）。尽管不能完全避免死锁，但是总可以最小化死锁的出现次数。例如，应该编写应用程序，让进程持有锁的时间尽可能地短，这样，其他进程就不必花太长的时间等待锁被释放。尽管您并不经常直接调用锁定，但是可以通过保持事务尽可能地短来影响锁定。例如，不在事务中要求用户输入。相反，首先得到输入，然后快速执行事务。

10.7 行版本控制

在本章开头，我描述了 SQL Server 可以使用的两种并发模型。悲观并发根据您所使用的隔离级别，使用锁定来保证适当的事务行为，避免诸如脏读之类的问题。乐观并发使用一种叫做行版本控制的新技术来保证事务。从 SQL Server 2005 中开始，启用 READ_*COMMIT*TED_SNAPSHOT 和 ALLOW_SNAPSHOT_ ISOLATION 这两个数据库属性中的一个或者两个之后，乐观并发就可用了。使用乐观并发时可获得独占锁，所以仍然需要明白与锁模式、锁资源、锁持续时间、跟踪及管理锁所需的资源等相关的所有问题。乐观并发与悲观并发之间的区别是，使用乐观并发，写入者和读取者不相互阻塞。或者，使用锁定术语来说，当请求的资源上当前有一个共享锁时，请求其独占锁的进程不会受阻。反过来说，当请求的资源上当前有一个独占锁时，请求其共享锁的进程也不会受阻。

避免阻塞是可能的，因为只要启用了一个新数据库选项，SQL Server 就开始使用 *tempdb* 来存储所有发生更改的行的副本（版本），并且将这些副本一直保存到没有任何事务需要访问它们为止。*tempdb* 中用于存储已更改行的以前版本的空间叫做版本存储区。

10.7.1 行版本控制概述

在 SQL Server 的早期版本中，并发解决方案的代价是，如果我们愿意冒数据不一致的风险（就是说，如果使用已提交读隔离级别），就可以避免让写入者阻塞读取者。如果我们的结果必须总是基于已提交的数据，那么我需要愿意等待更改被提交。

SQL Server 2005 引入了一种叫做快照隔离级别的新隔离级别和一种新的非阻塞风格的已提交读隔离级别，叫做已提交读快照隔离（RCSI）。这些基于行版本控制的隔离级别允许读取者得到行的一个前面已提交的值，而不会被阻塞，所以系统中的并发性提高了。为此，SQL Server 在行被更新或删除时必须保存行的老版本。如果对同一行进行多个更新，就可能需要维护行的多个较老的版本。正因为如此，行版本控制有时被称作多版本并发控制。

为支持存储行的多个较老版本，*tempdb* 数据库要使用额外的磁盘空间。用于版本存储区的磁盘空间必须被适当监控和管理，本节后面我给出了一些这样的方法。版本控制的工作原理是，让更改数据的任何事务都保存数据的老版本，以便从这些老版本构造出数据库（或数据库的一部分）的快照。

10.7.2 行版本控制细节

当表或索引中的一行被更新时，新行被标注上执行该更新的事务的事务序列号（XSN）。XSN 是一个递增的数字，在每个 SQL Server 数据库中是唯一的。XSN 的概念跟第 4 章中讨论的日志序列号（LSN）并不相同。稍后我将更详细地讨论 XSN。更新一个行时，前一版本存储在版本存储区中，新行包含一个到版本存储区中老行的指针。版本存储区中的老行可能还包含指向更老版本的指针。特定行的所有老版本都连接在一个链表中，SQL Server 可能需要在链表中经过好几个指针，才能到达适当的版本。只要有操作可能需要它们，版本行就必须保存在版本存储区中。

在图 10-7 中，行的当前版本由事务 T3 产生，并存储在普通数据页中。行的以前版本由事务 T2 和事务 Tx 产生，存储在版本存储区中的页中（在 *tempdb* 中）。

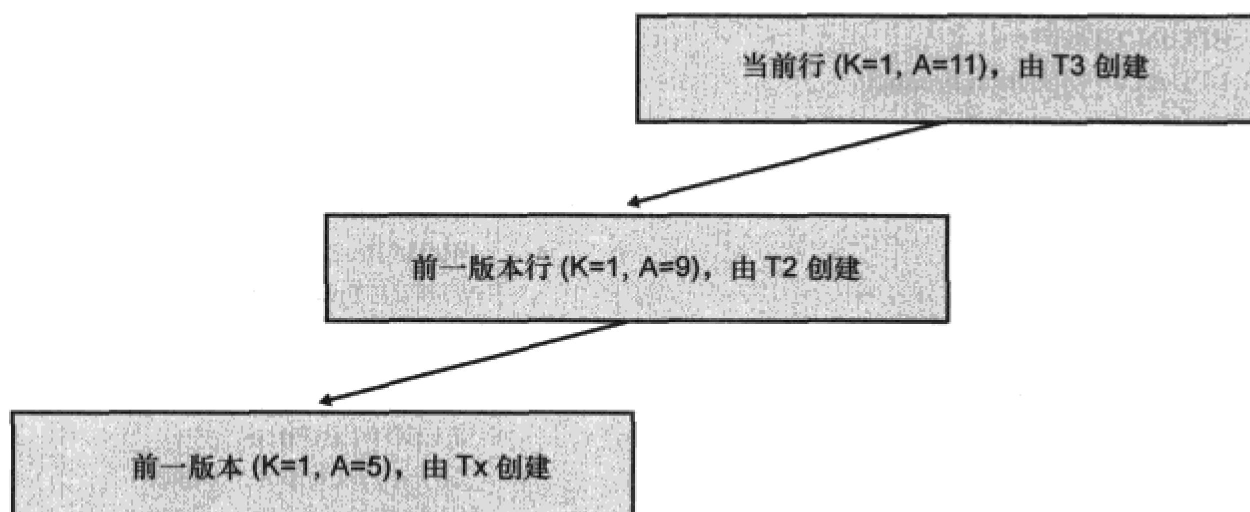


图 10-7 行的版本

行版本控制给 SQL Server 带来了一种乐观并发模型，在应用程序需要时或者使用默认悲观模型会将并发性降低到不可接受的地步时，就可以使用行版本控制。在切换到基于行版本的隔离级别之前，必须慎重考虑使用这个新并发模型的代价。除了需要额外的管理来监控用于版本存储区的 *tempdb* 的使用，由于维护老版本涉及到额外的工作，所以版本控制还降低了更新操作的性能。即使当前没有数据读取者，更新操作也要承担此成本。如果有读取者在使用版本控制，他们必须承担额外的成本，即通过链接指针找到所请求的行的适当版本。

此外，由于快照隔离级别的乐观并发模型（乐观地）假设没有很多更新冲突会发生，所以，如果您希望并发地更新同一数据，就不应该选用快照隔离级别。快照隔离级别在支持读取者不会被写入者阻塞方面做得很好，但是仍然不允许同时有多个写入者。在默认的悲观模型中，第一个写入者将阻塞所有后续写入者，但是通过使用快照隔离级别，后续写入者可以接收到错误消息，应用程序需要重新提交原来的请求。注意，这些更新冲突只发生于完全的快照隔离级别，不会发生于增强的 RCSI。

10.7.3 基于快照的隔离级别

SQL Server 2008 提供两种类型的基于快照的隔离级别，两者都使用行版本控制来维护快照。一种类

型是 RCSI，只要通过设置一个数据库选项就可以启用。一旦启用，就无需做进一步的更改。任何应该在默认已提交读隔离级别下运行的事务都将运行在 RCSI 下。另一种类型是快照隔离级别，必须在两个地方启用。首先必须用 ALLOW_SNAPSHOT_ISOLATION 选项启用数据库，然后每个想要使用 SI 的联接都必须使用 *SET TRANSACTION ISOLATION LEVEL* 命令设置隔离级别。下面我们来比较一下这两种类型的基于快照的隔离级别。

1. 已提交读快照隔离级别

RCSI 是一种语句级别基于快照的隔离级别，这表示任何查询都看到语句开始时最近提交的值。例如，我们来看表 10-9 中的场景。假设两个事务正运行在 *AdventureWorks2008* 数据库中（已经用 *ALLOW_SNAPSHOT_ISOLATION* 选项启用了该数据库），并且在两个事务开始运行之前，产品 922 的 *ListPrice* 值是 8.89。

表 10-9　　运行在 RCSI 下的 *SELECT*

时间	事务 1	事务 2
1	BEGIN TRAN UPDATE Production.Product SET ListPrice = 10.00 WHERE ProductID = 922;	BEGIN TRAN
2		SELECT ListPrice FROM Production.Product WHERE ProductID = 922; -- SQL Server returns 8.89
3	COMMIT TRAN	
4		SELECT ListPrice FROM Production.Product WHERE ProductID = 922; -- SQL Server returns 10.00
5		COMMIT TRAN

我们应该注意到，在 Time=2 时，事务 1 所做的更改仍然未提交，所以锁依然持有在 *ProductID* = 922 这一行上。但是，事务 2 并不阻塞在这个锁上；它可以访问该行的一个老版本，其中最新提交的 *ListPrice* 值是 8.89。在事务 1 提交并释放锁之后，事务 2 看到新的 *ListPrice* 值。这仍然是已提交读隔离级别（只不过是一种非锁定变种），所以不能保证读取操作是可重复的。

可以将 RCSI 看做只是默认隔离级别已提交读的一种变种。允许和不允许的行为相同，参见前面的表 10-2。

RCSI 用 *ALTER DATABASE* 命令启用或禁用，比如说下面这个命令是在 *AdventureWorks2008* 数据库中启用 RCSI：

```
ALTER DATABASE AdventureWorks2008
   SET READ_COMMITTED_SNAPSHOT ON;
```

具有讽刺意味的是，尽管该隔离级别旨在帮助避免锁定，但是如果在执行这个先导命令时，数据库中有任何用户，那么 *ALTER* 会阻塞他（发出 *ALTER* 命令的联接可以在数据库中，但是其他联接都不可以）。

直到更改成功，数据库继续像不是处于 RCSI 模式一样操作。通过为 *ALTER* 命令指定 TERMINATION 子句，可以避免锁定（这在第 3 章中介绍过）：

```
ALTER DATABASE AdventureWorks2008
  SET READ_COMMITTED_SNAPSHOT ON WITH NO_WAIT;
```

如果数据库中有用户，前导的 *ALTER* 会失败，产生以下错误消息：

```
Msg 5070, Level 16, State 2, Line 1
Database state cannot be changed while other users are using
the database 'AdventureWorks2008'
Msg 5069, Level 16, State 1, Line 1
ALTER DATABASE statement failed.
```

也可以指定一种 ROLLBACK 终止选项，主要是断开任何当前数据库连接。

RCSI 最大的优势是，在该模式下可以引入更大的并发性，因为读取者不阻塞写入者，写入者也不阻塞读取者。但是，写入者会阻塞写入者，因为通常的锁定行为适用于所有 *UPDATE*、*DELETE* 和 *INSERT* 操作。无需为任何会话设置 SET 选项，就可以利用 RCSI 的优势，因而我们可以降低阻塞和死锁的并发性影响，无需在应用程序中做任何更改。

2. 快照隔离级别

快照隔离级别需要在会话中使用 SET 命令，就像隔离级别的任何其他更改一样（例如，*SET TRANSACTION ISOLATION LEVEL* SERIALIZABLE）。要一个会话级别的选项生效，必须通过更改数据库以允许数据库使用 SI：

```
ALTER DATABASE AdventureWorks2008
   SET ALLOW_SNAPSHOT_ISOLATION ON;
```

在更改数据库以允许 SI 时，数据库中的用户阻塞不了命令的完成。但是，如果数据库中有活动的事务，*ALTER* 就会被阻塞。这并不意味着直到语句完成才会有效果。更改数据库以允许完全的 SI，可以是一个被延迟的操作。就 ALLOW_SNAPSHOT_ISOLATION 来说，数据库实际上可以处于 4 种状态之一。可以是 ON 或 OFF，也可以是 IN_TRANSITION_TO_ON 或 IN_TRANSITION_TO_OFF。

下面是更改数据库的 ALLOW_SNAPSHOT_ISOLATION 选项时会发生的情况。

- SQL Server 等待所有活动事务的完成，数据库状态被设置为 IN_TRANSITION_TO_ON。
- 任何新的 *UPDATE* 或 *DELETE* 事务开始在版本存储区中生成版本。
- 新的快照事务无法开始，因为已经进行中的事务没有在数据更改时存储新的版本。新的快照事务应该必须具有要读取的数据的已提交版本。执行 *SET TRANSACTION ISOLATION LEVEL SNAPSHOT* 命令时不会出错；错误出现在选择数据时，会得到以下消息：

```
Msg 3956, Level 16, State 1, Line 1
Snapshot isolation transaction failed to start in database 'AdventureWorks2008'
because the ALTER DATABASE command which enables snapshot isolation for this database
has not finished yet. The database is in transition to pending ON state. You must wait
until the ALTER DATABASE Command completes successfully.
```

- 只要在 *ALTER* 命令开始时活动的所有事务一完成，*ALTER* 就可以结束，并且状态更改完成。数据库现在的状态是 ALLOW_SNAPSHOT_ISOLATION。

让数据库脱离 ALLOW_SNAPSHOT_ISOLATION 模式与此类似，同样有一个转换阶段。

- SQL Server 等待所有活动事务的完成，数据库状态被设置为 IN_TRANSITION_TO_OFF。
- 新的快照事务无法开始。
- 现有快照事务仍然执行快照扫描，从版本存储区读取数据。
- 新的事务继续生成版本。

3. 快照隔离级别范围

SI 为我们提供事务上一致的数据视图。任何行读都是事务开始时行的最新已提交版本（对于 RCSI，我们得到语句开始时最新的已提交版本）。要记住的一个关键点是，事务不是开始于 *BEGIN TRAN* 语句；对于 SI 而言，事务开始于事务首次访问数据库中的任何数据时。

作为 SI 的一个例子，我们来看一个类似于表 10-9 中的场景。表 10-10 展示了一个 ALLOW_SNAPSHOT_ISOLATION 设置为 ON 的数据库中的活动。假设两个事务运行在 *AdventureWorks2008* 数据库中，并且在事务开始之前，Product 922 的 *ListPrice* 值为 10.00。

表 10-10 运行在 SNAPSHOT 事务中的 *SELECT*

时间	事务 1	事务 2
1	BEGIN TRAN	
2	UPDATE Production.Product SET ListPrice = 12.00 WHERE ProductID = 922;	SET TRANSACTION ISOLATION LEVEL SNAPSHOT
3		BEGIN TRAN
4		SELECT ListPrice FROM Production.Product WHERE ProductID = 922; -- **SQL Server returns 10.00** -- This is the beginning of -- the transaction
5	COMMIT TRAN	SELECT ListPrice FROM Production.Product WHERE ProductID = 922; -- **SQL Server returns 10.00** -- Return the committed -- value as of the beginning -- of the transaction
6		COMMIT TRAN
7		SELECT ListPrice FROM Production.Product WHERE ProductID = 922; -- SQL Server returns 12.00

即使事务 1 已提交，事务 2 仍继续返回它读取的初始值 10.00，直到事务 2 完成。只有当事务 2 完成后，联接才读取 *ListPrice* 的新值。

4. 查看数据库状态

目录视图 *sys.databases* 包含几列，用于报告数据库的快照隔离级别状态。数据库可以启用为支持 SI

和/或 RCSI。但是，启用一个并不会自动启用或禁用另一个。必须使用单独的 *ALTER DATABASE* 命令，分别启用或禁用它们。

snapshot_isolation_state 列的可能值是 0 到 4，分别代表一种 SI 状态，*snapshot_isolation_state_desc* 列则解释该状态。表 10-11 总结了每种状态的含义。

表 10-11　　数据库选项 ALLOW_SNAPSHOT_ISOLATION 的可能值

快照隔离级别状态	说　明
OFF	快照隔离级别状态在数据库中被禁用。换言之，具有快照隔离级别的事务是不被允许的。数据库版本控制状态最初在恢复期间被设置为 OFF。如果版本控制被启用，那么版本控制状态在恢复之后被设置为 ON
IN_TRANSITION_TO_ON	数据库正在启用 SI。它等待发出 ALTER DATABASE 命令时处于活动状态的所有 *UPDATE* 事务的完成。该数据库中新的 *UPDATE* 事务开始为版本控制付出代价，即产生行版本。使用快照隔离级别的事务无法开始
ON	SI 被启用。新的快照事务可在该数据库中开始。在版本控制状态变成 ON 之前开始的现有快照事务（在另一个启用快照的会话中）无法在该数据库中进行快照扫描，因为这些事务感兴趣的快照未由 *UPDATE* 事务适当生成
IN_ TRANSITION_TO_OFF	数据库正在禁用 SI 状态，不能开始新的快照事务。*UPDATE* 事务仍然要在该数据库中为版本控制付出代价。现有快照事务仍然能够进行快照扫描。不到所有现有事务都完成，IN_TRANSITION_TO_OFF 不会变成 OFF

is_read_committed_snapshot_on 列的值为 0 或 1。表 10-12 总结了每种状态的含义。

表 10-12　　数据库选项 READ_*COMMITTED*_SNAPSHOT 的可能值

READ_*COMMITTED*_SNAPSHOT 状态	说　明
0	READ_*COMMITTED*_SNAPSHOT 被禁用
1	READ_*COMMITTED*_SNAPSHOT 被启用。任何隔离级别为 Read *COMMITted* 的查询都以非阻塞模式执行

利用下面这个查询，可以看到您所有数据库的这些快照状态的值：

```
SELECT name, snapshot_isolation_state_desc,
         is_read_COMMITted_snapshot_on , *
FROM sys.databases;
```

5. 更新冲突

两种乐观并发级别之间的一个主要区别是，当一个进程在事务执行过程中看到相同的数据，并不会仅仅因为另一个进程在更改相同的数据就被阻塞，SI 可潜在地导致更新冲突。表 10-13 演示了两个进程试图更新 *AdventureWorks2008* 数据库 *ProductInventory* 表中同一行的 *Quantity* 值。两个店员都接收到了 ProductID 872 的发货，并试图更新库存。*AdventureWorks2008* 数据库的 ALLOW_SNAPSHOT_ISOLATION 设置为 ON，在两个事务开始之前，Product 872 的 *Quantity* 值是 324。

表 10-13　　快照隔离级别中的更新冲突

时间	事务 1	事务 2
1		`SET TRANSACTION ISOLATION LEVEL SNAPSHOT`

续表

时间	事务 1	事务 2
2		BEGIN TRAN
3		SELECT Quantity FROM Production.ProductInventory WHERE ProductID = 872; -- SQL Server returns 324 -- This is the beginning of -- the transaction
4	BEGIN TRAN UPDATE Production.ProductInventory SET Quantity=Quantity + 200 WHERE ProductID = 872; -- Quantity is now 524	
5		BEGIN TRAN UPDATE Production.ProductInventory SET Quantity=Quantity + 200 WHERE ProductID = 872; -- Quantity is now 524
6	COMMIT TRAN	
7		-- Process receives error 3960

发生冲突是因为事务 2 开始时 *Quantity* 值是 324。当该值被事务 1 更新时，*Quantity* 值为 324 的行版本被保持在版本存储区中。事务 2 继续在事务执行过程中读取该行。如果两个 *UPDATE* 操作都被允许成功，我们就会遇到经典的丢失更新情形。事务 1 将 *Quantity* 值增加 200，然后事务 2 将初始值增加 300 并保存。事务 1 增加的 200 将彻底丢失。SQL Server 不允许这样。

事务 2 首次试图执行 *UPDATE* 时，不会立即得到错误——只会被阻塞。事务 1 具有该行的一个独占锁，所以事务 2 在试图得到独占锁时被阻塞。如果事务 1 回滚它的事务，事务 2 就能够完成 *UPDATE*。但是由于事务 1 已提交，所以 SQL Server 检测到一个冲突，并产生以下错误：

```
Msg 3960, Level 16, State 2, Line 1
Snapshot isolation transaction aborted due to update conflict. You cannot use snapshot
isolation to access table 'Production.ProductInventory' directly or indirectly in database'
AdventureWorks2008' to update, delete, or insert the row that has been modified or deleted
by another transaction. Retry the transaction or change the isolation level for the
update/delete statement.
```

冲突只可能在 SI 模式下出现，因为该隔离级别是基于事务的，而不是基于语句。如果表 10-13 中的例子是在使用 RCSI 的数据库中执行，那么事务 2 执行的 *UPDATE* 语句就不会使用数据的旧值。它在试图读取当前 *Quantity* 时会被阻塞，然后当事务 1 完成之后，它将读取新更新的 *Quantity* 作为当前值，并将之增加 300。无论哪一个更新都不会丢失。

如果选择工作在 SI 模式中，需要知道可能发生冲突。冲突可以最小化，但是就跟死锁一样，无法确保永远不会有冲突。编写应用程序时，必须适当地处理冲突，并且不要假设 *UPDATE* 已经成功。如果偶尔出现冲突，可以将之看成是使用 SI 的代价，但是如果出现的频率过高，则应该采取另外的措施了。

您可能会考虑 SI 是否真正必需，如果确实必需的话，您应该确定基于语句的 RCSI 是否可带来您所

需的行为，却没有检测和处理冲突的代价。另一个解决方案是，使用 UPDLOCK 查询提示来确保，在您准备更新数据之前没有其他进程更新数据。在表 10-13 中，事务 2 应该在它的初始 *SELECT* 上使用 UPDLOCK，如下所示：

```
SELECT Quantity
FROM Production.ProductInventory WITH (UPDLOCK)
WHERE ProductID = 872;
```

UPDLOCK 提示迫使 SQL Server 为事务 2 在被选择的行上获得更新锁。然后事务 1 试图更新该行时被阻塞。事务 1 使用的不是 SI，所以它不会看到 *Quantity* 以前的值。因为事务 1 被阻塞了，所以事务 2 可以执行它的更新并提交。然后事务 1 可以在 *Quantity* 的新值上执行更新，两个更新都没有丢失。

在本章末尾，我们将更详细地讨论锁定提示。

6. 数据定义语言和快照隔离级别

使用 SI 时需要知道，尽管 SQL Server 保存所有已更改数据的版本，但是元数据没有保存老版本。因此，某些 DDL 语句在快照事务中是不被允许的。以下 DDL 语句在快照事务中不被允许。

- *CREATE / ALTER / DROP INDEX*
- *DBCC DBREINDEX*
- *ALTER TABLE*
- *ALTER PARTITION FUNCTION / SCHEME*

另一方面，以下 DDL 语句是允许的。

- *CREATE TABLE*
- *CREATE TYPE*
- *CREATE PROC*

注意，可允许的 DDL 语句都是创建新对象的语句。在 SI 中，任何同步数据修改都无法影响这些对象的创建。表 10-14 展示了一个包含 *CREATE TABLE* 和 *CREATE INDEX* 语句的快照事务的伪代码例子。

表 10-14　SNAPSHOT 事务中的 DDL

时间	事务 1	事务 2
1	SET TRANSACTION ISOLATION LEVEL SNAPSHOT;	
2	BEGIN TRAN	
3	SELECT count(*) FROM Production.Product; -- This is the beginning of -- the transaction	
4		BEGIN TRAN
5	CREATE TABLE NewProducts (<column definitions>) -- This DDL is legal	INSERT Production.Product VALUES (9999, ...) -- A new row is insert into -- the Product table
6		COMMIT TRAN

续表

时间	事务 1	事务 2

7	`CREATE INDEX PriceIndex`
	`ON Production.Product`
	`(ListPrice)`
	`-- This DDL will generate an`
	`-- error`	

即使事务 1 处于 SI 模式，*CREATE TABLE* 语句也会成功，因为它不会受到任何其他进程所做事情的影响。*CREATE INDEX* 语句则不同。事务 1 开始时，ProductID 为 9999 的新行不存在。但是遇到 *CREATE INDEX* 语句时，事务 2 执行的 *INSERT* 已提交。事务 1 应该在索引中包含新行吗？真的是没有办法避免包含新行，但是这会妨碍事务 1 正在使用的快照，所以 SQL Server 产生一个错误，而不是创建索引。

要考虑的并发 DDL 的另一个方面是，当快照事务外的语句更改了快照事务引用的对象时会发生什么。DDL 是被允许的，但是当此情况发生时，会在快照事务中得到一个错误。表 10-15 展示了一个例子。

表 10-15　　SNAPSHOT 事务外的并发 DDL

时间	事务 1	事务 2
1	`SET TRANSACTION ISOLATION` `LEVEL SNAPSHOT;`	
2	`BEGIN TRAN`	

3	`SELECT TOP 10 *`
	`FROM Production.Product;`
	`-- This is the start of`
	`-- the transaction`	

4		`BEGIN TRAN`
		`ALTER TABLE Purchasing.Vendor`
		`ADD notes varchar(1000);`
		`COMMIT TRAN`

5	`SELECT TOP 10 *`
	`FROM Production.Product;`
	`-- Succeeds`
	`-- The ALTER to a different`
	`-- table does not affect`
	`-- this transaction`	

6		`BEGIN TRAN`
		`ALTER TABLE Production.Product`
		`ADD LowestPrice money;`
		`COMMIT TRAN`

7	`SELECT TOP 10 * FROM Production.`
	`Product;`
	`-- ERROR`	

对于前一种情况，在事务 1 中，重复的 *SELECT* 语句应该总是从 *Product* 表返回相同的数据。完全不同的另一个表上的外部 *ALTER TABLE* 对快照事务没有影响，但是事务 2 之后会更改 *Product* 表以添加新的列。由于表示前一个表结构的元数据没有保存旧版本，所以事务 1 无法为第三个 *SELECT* 产生相同

的结果。SQL Server 产生以下错误：

```
Msg 3961, Level 16, State 1, Line 1
Snapshot isolation transaction failed in database 'AdventureWorks2008' because the object
accessed by the statement has been modified by a DDL statement in another concurrent
transaction since the start of this transaction. It is disallowed because the metadata is
not versioned. A concurrent update to metadata can lead to inconsistency if mixed with
snapshot isolation.
```

在该版本中，对快照事务引用的对象的元数据做任何并发更改都会产生该错误，即使根本没有异常。例如，如果事务 1 发出一个 *SELECT count(*)*，这不会受到 *ALTER TABLE* 的影响，SQL Server 依然会产生错误 3961。

7. 基于快照的隔离级别的总结

SI 和 RCSI 的类似之处是，它们都基于数据库中行的版本控制。但是它们之间也有一些关键的区别，比如说这些选项是如何从管理角度启用的，又是如何影响您的应用程序的。这些区别我们已经讨论了很多次，但是为完整起见，表 10-16 列出了两种类型的基于快照的隔离级别之间的类似之处及区别。

表 10-16　快照与已提交读快照隔离级别

快照隔离级别	已提交读快照隔离级别
数据库必须配置为允许 SI，并且会话必须发出 *SET TRANSACTION ISOLATION LEVEL SNAPSHOT* 命令	数据库必须配置为使用 RCSI，并且会话必须使用默认隔离级别。无需代码更改
为数据库启用 SI 是一个在线操作。它允许 DBA 为一个特定的应用程序（比如正在创建大报告的应用程序）打开版本控制。DBA 可以在报告事务开始之后关闭版本控制，以阻止新的快照事务开始。在现有数据库中打开 SI 是同步的。控制权一旦交给了 *ALTER DATABASE* 命令，就要直到所有需要在当前数据库中创建版本的现有更新事务都完成了，才返回给 DBA。此时，ALLOW_SNAPSHOT_ISOLATION 更改为 ON。然后用户才可以在该数据库中开始快照事务。关闭 SI 也是同步的	为数据库启用 RCSI 需要数据库上的 SHARED_TRANSACTION_WORKSPACE 锁。要启用该选项，所有用户都必须被踢出数据库
启用该数据库选项时，对数据库中的活动会话没有任何限制	启用该选项时，数据库中不应该有其他会话是活动的
如果一个应用程序运行的快照事务要访问两个数据库中的表，那么 DBA 必须在两个数据库中都打开 ALLOW_SNAPSHOT_ISOLATION，然后应用程序才能开始快照事务	RCSI 实际上是一个表级别的选项，所以同一个查询中引用的两个不同数据库中的表，可以各自具有自己单独的设置。一个表可能从版本存储区获得数据，另一个表则可能只是读取数据的当前版本。无需两个数据库都启用 RCSI 选项
IN_TRANSITION 版本控制状态不持久。只有 ON 和 OFF 状态才被记忆到磁盘上	这里没有 IN_TRANSITION 状态。只有 ON 和 OFF 状态持久存在
当数据库从服务器崩溃之后恢复，或者在 SQL Server 实例关闭、还原、附加或联线之后，该数据库的所有版本控制历史记录都将丢失。如果数据库版本控制状态为 ON，那么 SQL Server 可以允许新的快照事务访问数据库，但是必须阻止以前的快照事务访问数据库。这些以前的事务可能需要从数据库恢复之前的某个点访问数据	这是一个对象级别的选项；它不是事务级别的，所以它不适用
如果数据库处于 IN_TRANSITION_TO_ON 状态，那么 *ALTER DATABASE SET ALLOW_SNAPSHOT_ ISOLATION OFF* 等待大约 6 秒钟，若数据库状态依然是 IN_TRANSITION_TO_ON，它可能会失败。DBA 可以在数据库状态更改为 OFF 之后重试该命令	该选项只有在数据库中没有其他活动会话时才能被启用，所以没有过渡状态

续表

快照隔离级别	已提交读快照隔离级别
对于只读数据库，版本控制是自动启用的。您仍然可以对只读数据库使用 ALTER DATABASE SET ALLOW_SNAPSHOT_ISOLATION ON。数据库以后变成可读写的话，该数据库的版本控制仍然是启用的	跟 SI 一样，版本控制对于只读数据库是自动启用的
如果有长时间运行的事务，那么 DBA 可能需要等待一段长的时间，版本控制状态更改才能完成。DBA 可以取消等待，那么版本控制状态回滚，设置成前一个值	该选项只有在数据库中没有其他活动会话时才能被启用，所以没有过渡状态
不能在用户事务中使用 ALTER DATABASE 来更改数据库版本控制状态	跟 SI 一样，不能在用户事务中更改数据库版本控制状态
可以更改 *tempdb* 的版本控制状态。在 SQL Server 重启时，*tempdb* 的版本控制状态被保留，尽管 *tempdb* 的内容不被保留	不能为 *tempdb* 打开该选项
可以更改 *master* 数据库的版本控制状态	不能为 *master* 数据库更改该选项
可以更改模型的版本控制状态。如果为模型启用了版本控制，那么创建的每个新数据库也都启用了版本控制。但是，*tempdb* 的版本控制状态并不因您为模型启用了版本控制而被自动启用	类似于 SI 的行为，只是没有针对 *tempdb* 的隐含意思
可以为 *msdb* 打开该选项	不可以为 *msdb* 打开该选项，因为这会潜在地打断建立在 *msdb* 之上、依赖于阻塞已提交读隔离级别的行为的应用程序
SI 事务中的查询看到事务开始之前已提交的数据，而事务中的每个语句看到相同的一组已提交的更改	运行在 RCSI 中的语句看到语句开始之前已提交的所有内容。事务中的每个新语句使用最新提交的更改
SI 可以导致更新冲突，从而引起回滚或取消事务	不可能有更新冲突

8. 版本存储区

数据库只要启用 ALLOW_SNAPSHOT_ISOLATION 或 READ_*COMMIT*TED_SNAPSHOT，所有 *UPDATE* 和 *DELETE* 操作就开始产生以前提交的行的行版本，并将这些版本存储在 *tempdb* 中数据页上的版本存储区内。只要有需要版本行的快照查询，就应该将这些版本行保存在版本存储区中。

SQL Server 2008 提供几个 DMV，其中包含关于活动快照事务和版本存储区的信息。我们不会详细介绍所有这些 DMV，只介绍其中几个比较关键的，帮助您确定版本存储区的利用率有多高及哪些快照事务会影响操作结果。我们介绍的第一个 DMV 是 *sys.dm_tran_version_store*，其中包含关于版本存储区中实际的行的信息。运行下面这个脚本以创建 *Production.Product* 表的副本，然后在 *AdventureWorks2008* 中打开 ALLOW_SNAPSHOT_ISOLATION。最后，验证该选项处于 ON 状态，并且版本存储区中当前没有数据行。您可能需要关闭任何当前在使用 *AdventureWorks2008* 的活动事务。

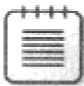
```
USE AdventureWorks2008
SELECT * INTO NewProduct
FROM Production.Product;
GO
ALTER DATABASE ADVENTUREWORKS2008 SET ALLOW_SNAPSHOT_ISOLATION ON;
GO
SELECT name, snapshot_isolation_state_desc,
       is_read_committed_snapshot_on
FROM sys.databases
WHERE name= AdventureWorks2008;
```

```
GO
SELECT COUNT(*) FROM sys.dm_tran_version_store;
GO
```

只要看到该数据库选项处于 ON 状态并且版本存储区中没有数据行，就可以继续。我想要阐明的是，只要一启用 ALLOW_SNAPSHOT_ISOLATION，SQL Server 就开始存储行版本，即使没有快照事务需要读取这些版本。所以，现在我们在 *NewProduct* 表上运行这个 *UPDATE* 语句，再来看看版本存储区：

```
UPDATE NewProduct
SET ListPrice = ListPrice * 1.1;
GO
SELECT COUNT(*) FROM sys.dm_tran_version_store;
GO
```

现在应该看到，版本存储区中有 504 行，因为 *NewProduct* 表中有 504 行数据。每一行在更新之前的版本都被写到 *tempdb* 中的版本存储区。



注意：

只要一启用数据库可以使用某个基于快照的隔离级别，SQL Server 就开始在 *tempdb* 中生成版本。在更新频繁的数据库中，这会影响其他使用 *tempdb* 的查询的行为，还会影响服务器本身。

如前面图 10-7 所示，版本存储区维护着行的链表。当前行指向比它老的行，后者又指向更老的行，依此类推。链表的末尾是这个特定行最老的版本。为了支持行版本控制，一个行需要额外 14 字节的信息来跟踪指针。8 字节分配给指向 *tempdb* 中文件、页和行的实际指针，还有 6 字节用于存储 XSN，以帮助 SQL Server 确定哪些是当前行，或者哪个版本的行是某个特定事务需要访问的。在以后介绍其他快照事务元数据时，还会进一步介绍 XSN。此外，每个数据行的第一个字节（TagA 字节）中的一位被打开，表示这一行中有版本控制信息。

任何在数据使用基于快照的隔离级别时插入或更新的行，都会包含这 14 个额外的字节。以下代码在 *AdventureWorks2008* 数据库（已经启用了 ALLOW_SNAPSHOT_ISOLATION）中创建一个小表，并在其中插入两行数据。我之后使用 *DBCC IND* 找到页码（即页 6709），并使用 DBCC 来看页上的行。输出只显示插入的其中一行：

```
CREATE TABLE T1 (T1ID char(1), T1name char(10));
GO
INSERT T1 SELECT 'A', 'aaaaaaaaaa';
INSERT T1 SELECT 'B', 'bbbbbbbbbb';
GO
DBCC IND (AdventureWorks2008, 'T1',-1); -- page 6709
DBCC TRACEON (3604);
DBCC PAGE('AdventureWorks2008', 1, 6709, 1);
OUTPUT ROW:
Slot 0, Offset 0x60, Length 32, DumpStyle BYTE
Record Type = PRIMARY_RECORD
Record Attributes = NULL_BITMAP VERSIONING_INFO
Memory Dump @0x6207C060
00000000: 50000f00 41616161 61616161 61616102 †P...Aaaaaaaaaaa.
00000010: 00fc0000 00000000 0000020d 00000000 †................
```

我突出显示了新的头部信息（指出这一行包含版本控制信息），我还突出显示了 14 字节的版本控制

信息。该行中的 XSN 全部为 0，因为它没有作为快照隔离级别需要跟踪的事务的一部分被修改。*INSERT* 语句创建没有快照事务需要看到的新数据。如果我更新其中一行，那么该行的更新前版本会被写到版本存储区，并且 XSN 反映在行版本控制信息中：

```
UPDATE T1 SET T1name = '2222222222' where T1ID = 'A';
GO
DBCC PAGE('AdventureWorks2008', 1, 6709, 1);
GO
OUTPUT ROW:
Slot 0, Offset 0x60, Length 32, DumpStyle BYTE
Record Type = PRIMARY_RECORD
Record Attributes = NULL_BITMAP VERSIONING_INFO
Memory Dump @0x61C4C060
00000000: 50000f00 41323232 32323232 32323202 †P...A2222222222.
00000010: 00fc1804 00000100 0100590d 00000000 †..........Y.....
```

正如所提到的，如果为某个基于快照的隔离级别启用了数据库，那么每个新行都被增添了额外的 14 字节，不管这个行是否实际涉及版本控制。每个经过更新的行也都被添加了 14 字节（如果还没有这 14 字节的话），并且更新被作为后跟 *INSERT* 的 *DELETE* 来完成。这意味着，对于完整页上的表和索引，一个简单的 *UPDATE* 会导致页分割。

当为快照启用的数据库中的一行被删除时，一个指针被留在页上作为幻像记录，指向版本存储区中已删除的行。这些幻像记录非常类似于我们在第 6 章中看到的记录，它们作为版本控制清除进程的一部分被清除，我们马上就会介绍。下面是版本控制下的一个示例虚影记录：

```
DELETE T1 WHERE T1ID = 'B';
DBCC PAGE('AdventureWorks2008 ', 1, 6709, 1);
GO
--Partial Results:
Slot 4, Offset 0x153, Length 15, DumpStyle BYTE

Record Type = GHOST_VERSION_RECORD
Record Attributes = VERSIONING_INFO
Memory Dump @0x5C0FC153

00000000: 4ef80300 00010000 00210200 000000†††N........!.....
```

记录头部指出，该行是一个 GHOST_VERSION_RECORD，并且包含版本控制信息。但是，实际的数据不在这一行上，行上是 XSN，所以快照事务知道该行是何时删除的，以及是否应该在快照中访问较老的版本。*sys.dm_db_index_physical_stats* DMV 在第 6 章讨论过，它包含因版本控制而产生的虚影记录的个数（*version_ghost_record_count*）和所有虚影记录的个数（*ghost_record_count*），后者包含前者。如果一个更新作为紧跟着 *INSERT* 的 *DELETE* 执行，那么旧值和新值的虚影必须同时存在，这增加了对象的空间需求。

如果数据库处于基于快照的隔离级别，那么所有对数据行和索引行的更改都必须进行版本控制。遍历索引的快照查询仍然需要访问指向较老行（版本控制存储的行）的索引行。所以在索引级别，我们可能具有与新值同时存在的旧值（作为虚影），因而索引会要求更多的存储空间。

如果数据库更改为非快照隔离级别，就可以删除那额外的 14 字节的版本控制信息。一旦数据库选项更改，每次包含版本信息的行被更新，版本控制字节就会被删除。

9. 版本存储区管理

版本存储区的大小是自动管理的，SQL Server 包含一个清除线程，以确保版本控制存储的行在不再需要时被删除。对于运行在 SI 下的查询，行版本必须保存到事务结束。对于运行在 RCSI 下的 *SELECT* 语句，一个特定的行版本在 *SELECT* 语句执行之后就不需要了，可以被删除。

常规的清除函数作为后台进程每分钟执行一次，以从版本存储区回收所有可重用的空间。如果 *tempdb* 真正用完了所有可用空间，那么在 SQL Server 增加文件大小之前，会调用这个清除函数。如果磁盘太满了，以致文件不能增大，那么 SQL Server 将停止生成版本。发生这样的情况时，快照查询如果需要读取一个因空间限制而没有生成的版本，就会失败。尽管详细讨论这个问题的监控和解决超出了本书范畴，但是我要指出，SQL Server 2008 包含好些性能计数器，用于监控 *tempdb* 和版本存储区。其中有一些计数器用于跟踪使用行版本控制的事务。以下计数器包含在 SQLServer:Transactions 性能对象中。额外的详细信息和其他计数器可在 *SQL Server 联机丛书*中找到。

- **Free Space in *tempdb***。该计数器监控 *tempdb* 数据库中可用空间的总量。可以观察该值来检测 *tempdb* 何时用完空间，用完空间会导致保存所有必需的版本行出现问题。
- **Version Store Size**。该计数器监控版本存储区的大小，单位是千字节。监控该计数器可以帮助估计 *tempdb* 需要的额外空间。
- **Version Generation Rate 和 Version Cleanup Rate**。这两个计数器监控版本存储区获得和释放空间的速率，单位是千字节/秒。
- ***UPDATE* Confl ict Ratio**。该计数器监控更新快照事务具有更新冲突的比率。它是冲突数量与更新快照事务总数的比率。
- **Longest Transaction Running Time**。该计数器监控任何使用行版本控制的事务的最长运行时间，单位是秒。它可用来确定任何事务是否运行了太长的时间，还可以帮助您确定 *tempdb* 中需要为版本存储区用到的最大空间大小。
- **Snapshot Transactions**。该计数器监控活动快照事务的总数。

10. 快照事务元数据

用于观察快照事务行为的最重要的 DMV 是 *sys.dm_tran_ version_store*（本章前面简要介绍过）、*sys.dm_tran_transactions_snapshot* 和 *sys.dm_tran_active_snapshot_database_transactions*。

所有这些视图都包含一个叫做 *transaction_sequence_num*（即前面提到过的 XSN）的列。每个事务在开始一个快照读取或者将数据写入启用快照的数据库时，都会被分配到一个递增的 XSN 值。当 SQL Server 实例重启时，XSN 被重新设置为 0。不生成版本行且不使用快照扫描的事务不会收到 XSN。

另一个列，即 *transaction_id*，也被用于一些快照事务元数据中。事务 ID 是分配给事务的唯一识别号。它主要用于在锁定操作中识别事务。它也可以帮助您识别快照操作中涉及哪些事务。对于整个服务器中的每个事务（包括内部系统事务），事务 ID 值是递增的，所以不管事务是否被涉及任何快照操作中，当前事务 ID 值通常比当前 XSN 大很多。

可以使用 *sys.dm_tran_current_transaction* 视图检查当前事务号信息，该视图返回单个行，行中包括以下列。

- ***transaction_id***。该值显示当前事务的事务 ID。如果是在用户定义的事务中从视图选择，那么每次从视图选择应该会看到相同的 *transaction_id*。如果是在事务外面从 *sys.dm_tran_current_transaction*

运行 *SELECT*，那么 *SELECT* 本身会产生一个新的 *transaction_id* 值，您在每次执行相同的 *SELECT* 时会看到不同的值，即使在相同的连接中。

- ***transaction_sequence_num***。该值是当前事务的 XSN，如果有值的话。否则，该列返回 0。
- ***transaction_is_snapshot***。如果当前事务是在快照隔离级别下启动的，那么该值是 1；否则是 0（也就是说，如果当前会话显式地将 TRANSACTION ISOLATION LEVEL 设置为 SNAPSHOT，则该列是 1）。
- ***first_snapshot_sequence_num***。当前事务启动时，会拍下所有活动事务的快照，而该值是快照中事务的最低 XSN。
- ***last_transaction_sequence_num***。该值是系统产生的最新 XSN。
- ***first_useful_sequence_num***。该值是一个 XSN，表示可被清除而不会影响任何事务的版本存储区行的上限。任何 XSN 小于该值的行都是不再需要的值。

我现在创建了一个简单的版本控制场景，演示快照元数据中的值如何更新。这并不完善，但是应该能够帮助您开始探索自己的查询的版本控制元数据。我使用 *AdventureWorks2008* 数据库（它的 ALLOW_SNAPSHOT_ISOLATION 设置为 ON），并创建一个简单的表：

```
CREATE TABLE t1
(col1 int primary key, col2 int);
GO
INSERT INTO t1 SELECT 1,10;
INSERT INTO t1 SELECT 2,20;
INSERT INTO t1 SELECT 3,30;
```

我将该会话称为 Connection 1。更改会话的隔离级别，启动一个快照事务，并检查一些元数据：

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
GO
BEGIN TRAN
SELECT * FROM t1;
GO
select * from sys.dm_tran_current_transaction;
select * from sys.dm_tran_version_store;
select * from sys.dm_tran_transactions_snapshot;
```

sys.dm_tran_current_transaction 视图应该显示类似于这样的一些内容：当前会话确实具有一个 XSN，并且事务是快照事务。此外，可以注意到，*first_useful_sequence_num* 值与该事务的 XSN 相同，因为现在没有其他快照事务是有效的。我称该事务的 XSN 为 XSN1。

版本存储区应该是空的（除非您在最后关头执行了其他快照测试）。此外，*sys.dm_tran_transactions_snapshot* 应该为空，表示在其他事务进行过程中没有快照事务启动。

在另一个连接中（Connection 2），运行一个更新并检查当前事务的一些元数据：

```
BEGIN TRAN
UPDATE T1 SET col2 = 100
   WHERE col1 = 1;
SELECT * FROM sys.dm_tran_current_transaction;
```

注意：

尽管该事务因为产生版本而具有一个 XSN，但是它不是运行在 SI 中，所以 *transaction_is_snapshot* 值为 0。我称该事务的 XSN 为 XSN2。

现在，在 Connection 3 中启动第三个事务来执行另一个 *SELECT*（不要担心，这是最后一个 *SELECT*，并且我们不会保留它）。几乎与第一个 *SELECT* 相同，但是元数据结果中有一个重要的区别：

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
GO
BEGIN TRAN
SELECT * FROM t1;
GO
select * from sys.dm_tran_current_transaction;
select * from sys.dm_tran_transactions_snapshot;
```

在 *sys.dm_tran_current_transaction* 视图中，看到该事务的一个新的 XSN（XSN3），而看到 *first_snapshot_sequence_num* 和 *first_useful_sequence_num* 的值都与 XSN1 相同。在 *sys.dm_tran_transactions_snapshot* 视图中，看到这个 XSN3 的事务有两行，表示在这个事务启动时有两个事务是活动的。XSN1 和 XSN2 都显示在 *snapshot_sequence_num* 列中。现在可以提交或回滚该事务，然后关闭连接。

回到 Connection 2（在这里启动了 *UPDATE*），提交事务。

现在我们回到 Connection 1 中的第一个 *SELECT* 事务，并重新运行 *SELECT* 语句，留在同一事务中：

```
SELECT * FROM t1;
```

即使 Connection 2 中的 *UPDATE* 已经提交，仍然可以看到初始的数据值，因为我们运行的是快照事务。我们可以用以下查询检查 *sys.dm_tran_active_snapshot_database_transactions* 视图：

```
SELECT transaction_sequence_num, commit_sequence_num,
    is_snapshot, session_id,first_snapshot_sequence_num,
    max_version_chain_traversed, elapsed_time_seconds
FROM sys.dm_tran_active_snapshot_database_transactions;
```

这里不给出输出，因为页面容纳不下，但是有很多列您可能会觉得有用。具体来说，*transaction_sequence_num* 列包含 XSN1，这是当前联接的 XSN。您可以从任何联接实际运行刚才这个查询；它显示 SQL Server 实例中所有活动的快照事务，并且因为它包含 *session_id*，所以您可以将它与 *sys.dm_exec_sessions* 联结，得到关于正在运行该事务的联接的信息：

```
SELECT transaction_sequence_num, commit_sequence_num,
    is_snapshot, t.session_id,first_snapshot_sequence_num,
    max_version_chain_traversed, elapsed_time_seconds,
    host_name, login_name, transaction_isolation_level
FROM sys.dm_tran_active_snapshot_database_transactions t
   JOIN sys.dm_exec_sessions s
    ON t.session_id = s.session_id;
```

另一个要注意的是叫做 *max_version_chain_traversed* 的列。尽管它现在应该是 1，但我们可以更改。回到 Connection 2 并运行另一个 *UPDATE* 语句。即使 *BEGIN TRAN* 和 *COMMIT TRAN* 对于单个语句事务不是必需的，我还是包含了它们，以便清楚地显示该事务是完整的：

```
BEGIN TRAN
 UPDATE T1 SET col2 = 300
   WHERE col1 = 1;
COMMIT TRAN;
```

愿意的话，可以检查版本存储区，查看添加的行：

```
SELECT *
 FROM sys.dm_tran_version_store;
```

当回到 Connection 1 并在初始事务中运行相同的 *SELECT*，再来看 *sys.dm_tran_active_snapshot_database_transactions* 中的 *max_version_chain_traversed* 列时，应该看到数字在增长。重复的 *UPDATE* 操作（不管是 Connection 2 还是新联接中的）导致 *max_version_chain_traversed* 值不断增长，只要 Connection 1 保留在同一个事务中。可将这当做使用快照隔离级别的附加成本。在快照事务需要的数据上执行更多更新时，由于 SQL Server 必须遍历更长的版本链以得到事务需要的数据，所以读取操作花费的时间会更长。

就快照和事务元数据如何被用来检查快照事务的行为来说，我们介绍的不过是冰山一角。

10.7.4 选择并发模型

悲观并发是 SQL Server 2008 中的默认选项，在 SQL Server 2005 之前的所有 SQL Server 版本中是唯一选择。事务行为受到锁定的保障，代价是阻塞。访问相同数据源时，读取者可以阻塞写入者，写入者也可以阻塞读取者。由于 SQL Server 最初是设计和构建来使用悲观并发的，所以您应该考虑使用该模型，除非可以证实乐观并发对您和您的应用程序来说真的可以工作得更好。如果发现阻塞的代价变得过高，可以考虑使用乐观并发。

在大多数情形下，优先推荐 RCSI 而不是快照隔离级别，原因如下。

- RCSI 比 SI 消耗的 *tempdb* 空间少。
- RCSI 适合于分布式事务，SI 不适合。
- RCSI 不产生更新冲突。
- RCSI 不需要在应用程序中做任何更改。需要做的所有更改就是更改数据库选项。做了数据库级别更改之后，任何编写为使用默认已提交读隔离级别的应用程序都自动使用 RCSI。

以下情形可以考虑使用 SI。

- 任何事务因为更新冲突而必须回滚的可能性低。
- 具有需要基于长时间运行、必须保持时间点一致性的多语句查询而生成的报告。快照隔离级别提供可重复读的优势，不会被并发修改操作阻塞。

乐观并发确实具有它的优点，但是您必须了解它的代价。下面总结了它的优点。

- *SELECT* 操作不获得共享锁，所以读取者和写入者不相互阻塞。
- 所有 *SELECT* 操作检索一致的数据快照。
- 相对于悲观并发来说，所需的锁的总数大大减少，所以使用的系统开销也小。
- SQL Server 需要执行较少的锁升级。
- 不太可能出现死锁。

现在再来总结一下缺点。在权衡并发选项时，必须考虑基于快照的隔离级别的代价。

- 当必须扫描长的版本链时，*SELECT* 性能会受到负面影响。快照越老，在 SI 事务中访问所需的行花费的时间就越长。
- 行版本控制需要 *tempdb* 中的额外资源。
- 每当为数据库启用某一种基于快照的隔离级别，*UPDATE* 或 *DELETE* 操作就必须产生行版本（尽管我前面提到过，*INSERT* 操作不产生行版本，但是它们也会在一些情况下产生行版本。具体来说，假设您在一个具有唯一索引的表中插入一行，如果存在该行的一个较老版本，键值与新行的相同，并且这个旧行仍然作为幻像存在，那么新行就会产生一个版本）。

- 行版本控制信息让每个受影响的行的大小增加 14 字节。
- 由于维护行版本要做工作，所以 *UPDATE* 性能会变慢。
- 使用 SI 的 *UPDATE* 操作可能由于发现冲突而必须回滚。应用程序必须编写成能够处理出现的任何冲突。
- *tempdb* 中的空间必须严加管理。如果有运行时间很长的事务，那么更新事务产生的所有版本都必须保存在 *tempdb* 中。如果 *tempdb* 用完了空间，那么 *UPDATE* 操作不会失败，但是需要读取版本控制存储的数据的 *SELECT* 操作可能会失败。

要维护一个使用 SI 的生产系统，应该为 *tempdb* 分配足够的磁盘空间，以便总是至少有 10%的可用空间。如果可用空间低于该阈值，系统性能可能会降低，因为 SQL Server 将更多的资源花在回收版本存储区中的空间上。利用以下公式可以大致估计版本存储区所需的磁盘空间大小。对于长时间运行的事务，可以使用性能监视器来监控版本存储区数据产生和清除率，以估计所需的最大磁盘空间大小。

```
[size of common version store] =
2 * [version store data generated per minute]
* [longest running time (minutes) of the transaction]
```

10.8 控制锁定

SQL Server 查询优化器通常能选择正确类型的锁和锁模式。只有通过彻底的测试表明一种与此不同的方法更可取时，才应该忽视 SQL Server 查询优化器的行为。记住，通过设置隔离级别，会影响所持有的锁、导致阻塞的冲突及锁的持续时间。设置的隔离级别在整个会话期间有效，应该选择提供应用程序所需数据一致性的隔离级别。表级别锁定提示可用于在需要时更改默认锁定行为。不允许锁定级别会反过来影响并发性。

10.8.1 锁提示

T-SQL 语法允许您为单个表指定锁定提示，当 *SELECT*、*INSERT*、*UPDATE* 和 *DELETE* 语句中引用到这些表时，提示告诉 SQL Server 用于特定查询中特定表的锁定类型或版本控制。由于这些提示是在 FROM 子句中指定的，所以称为表级别提示。*SQL Server 联机丛书*中列出了除锁定提示之外的一些其他表级别提示，但是这些提示主要影响锁定行为。只有绝对需要更好地控制对象级别的锁定（相对于会话的隔离级别提供的控制）时，才应该使用这些表级别提示。SQL Server 锁定提示可以覆盖会话的当前事务隔离级别。在本节中，我只提到一些获得想要的并发行为时需要用到的锁定提示。

很多锁定提示只在事务环境中工作。但是，每个 *INSERT*、*UPDATE* 和 *DELETE* 语句都自动在一个事务中，所以唯一要关注的是对 *SELECT* 语句使用锁定提示。要获得在 *SELECT* 查询中使用以下提示的最大好处，必须使用一个显式的事务，以 BEGIN TRAN 开始，以 *COMMIT* TRAN 或 ROOLBACK TRAN 结束。锁提示语法如下：

```
SELECT select_list
FROM object [WITH (locking hint)]

DELETE [FROM] object [WITH (locking hint)
[WHERE <search conditions>]

UDPATE object [WITH (locking hint)
```

```
SET <set_clause>
[WHERE <search conditions>]

INSERT [INTO] object [WITH (locking hint)
<insert specification>
```

> **提示:**
>
> 并不是所有的锁定提示都需要关键字 WITH，但是没有 WITH 的语法在下一版 SQL Server 中将消失。推荐所有的提示都使用 WITH 指定。

可以为锁定提示指定以下关键字之一。

- **HOLDLOCK**。该提示等价于 SERIALIZABLE 提示。使用该提示类似于指定 *SET TRANSACTION ISOLATION LEVEL SERIALIZABLE*，只是 SET 选项影响所有的表，不止影响该提示中指定的表。
- **UPDLOCK**。该提示迫使 SQL Server 在读取表时采用更新锁而不是共享锁，并且一直持有到事务结束。采用更新锁对于消除转换死锁是一个重要技巧。
- **TABLOCK**。该提示迫使 SQL Server 采用表上的共享锁，否则会采用页锁。在您知道需要升级到表锁或者需要得到表的完整快照时，该提示非常有用。如果想要表锁一直持有到事务块结束，那么可以与 HOLDLOCK 一起使用该提示来以可重复读级别操作。如果与 *DELETE* 语句一起在堆上使用该提示，那么该提示允许 SQL Server 取消分配在行被删除时分配的页（如果从堆删除时获得行或页锁，空间将不被取消分配，因而不能被其他对象重新使用）。
- **PAGLOCK**。该提示迫使 SQL Server 在可能采用单个共享表锁的时候采用共享页锁（要请求独占页锁，必须同时使用 XLOCK 提示和 PAGLOCK 提示）。
- **TABLOCKX**。该提示迫使 SQL Server 采用表上的独占锁，一直持有到事务块结束（所有独占锁都持有到事务结束，不管处于何种隔离级别。该提示的效果与同时指定 TABLOCK 和 XLOCK 提示一样）。
- **ROWLOCK**。该提示指定在通常采用单个共享页或表锁时，应该采用共享行锁。
- **READUN*COMMIT*TED | REPEATABLEREAD | SERIALIZABLE**。这些提示指定，SQL Server 应该使用与事务隔离级别被设置为同名级别时相同的锁定机制。但是，该提示控制单个语句中单个表的锁定，而不是控制一个事务中所有语句中所有表的锁定。
- **READCOMMITTED**。该提示指定通过使用锁定或行版本控制，*SELECT* 操作遵循已提交读隔离级别的规则。如果数据库选项 READ_COMMITTED_SNAPSHOT 是 OFF，那么 SQL Server 使用共享锁并在读操作完成时马上释放。如果数据库选项 READ_COMMITTED_SNAPSHOT 是 ON，那么 SQL Server 不获得锁，而是使用行版本控制。
- **READCOMMITTEDLOCK**。该提示指定 *SELECT* 语句使用已提交读隔离级别（SQL Server 默认级别）的锁定版本。不管数据库选项 READ_COMMITTED_SNAPSHOT 的设置如何，SQL Server 都在读取数据时获得共享锁，在读操作完成时释放锁。
- **NOLOCK**。该提示允许未提交读（或脏读）。因为没有请求共享锁，所以在读取持有独占锁的数据时，语句不会阻塞。换言之，检测不到锁定冲突。该提示等价于 READUNCOMMITTED。
- **READPAST**。该提示指定锁定的行被跳过（跳过读）。READPAST 只适用于在 READ COMMITTED 隔离级别运行的事务，并且只跳过行级别锁读取数据。

- **XLOCK**。该提示指定 SQL Server 应该在语句处理的所有数据上采用一直持有到事务结束的独占锁。根据独占锁适用的指定资源，该锁可以用 PAGLOCK 或 TABLOCK 指定。

10.8.2 设置锁超时

设置 LOCK_TIMEOUT 可以让您控制 SQL Server 锁定行为。默认情况下，SQL Server 等待锁时不会超时；它乐观地假设锁最终都会被释放。大多数客户端编程界面允许您为连接设置常规超时限制，以便在等待指定的时间后没有响应时，查询自动被客户端取消。但是，超时的时候返回来的消息并不指出取消的原因；可能是因为锁没有被释放，也可能因为网络太慢，或者只不过是一个运行时间长的查询。

与其他 SET 选项一样，SET LOCK_TIMEOUT 只对当前连接有效。它的值的单位是毫秒，可以使用系统函数@@LOCK_TIMEOUT 进行访问。本例将 LOCK_TIMEOUT 值设置为 5 秒，然后检索该值进行显示：

```
SET LOCK_TIMEOUT 5000;
SELECT @@LOCK_TIMEOUT;
```

连接超过锁超时值时，会收到以下错误消息：

```
Server: Msg 1222, Level 16, State 50, Line 1
Lock request time out period exceeded.
```

将 LOCK_TIMEOUT 值设置为 0 意味着 SQL Server 根本不等待锁。它取消整个语句，进入批处理的下一条语句。这跟 READPAST 提示不同，READPAST 提示跳过单个行。

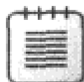
下面这个例子演示了 READPAST、READUNCOMMITTED 和将 LOCK_TIMEOUT 设置为 0 之间的区别。所有这些技术都用于避免阻塞问题，但是各自的行为稍有不同。

(1) 在新的查询窗口中，执行下面的批处理，锁定 *HumanResources.Department* 表中的一行：

```
USE AdventureWorks2008;
BEGIN TRAN;
UPDATE HumanResources.Department
SET ModifiedDate = getdate()
WHERE DepartmentID = 1;
```

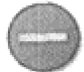
(2) 打开第二个连接，执行以下语句：

```
USE AdventureWorks2008;
SET LOCK_TIMEOUT 0;
SELECT * FROM HumanResources.Department;
SELECT * FROM Sales.SalesPerson;
```



注意：

在收到错误 1222 之后，第二个 *SELECT* 语句被执行，返回 *SalesPerson* 表中的所有 17 行。在遇到错误 1222 时，批处理没有被取消。



警告：

在遇到锁超时错误时，不只有批处理不会被取消，任何活动事务也不会被回滚。如果一个事务中有两个 *UPDATE* 语句，只要其中一个成功，另一个就必须成功，那么其中一个 *UPDATE* 语句的锁超时将仍然允许另一个语句被处理。必须在批处理中包含错误处理，以便遇到错误

（3）打开第三个连接，并执行以下语句：

```
USE AdventureWorks2008 ;
SELECT * FROM HumanResources.Department (READPAST);
SELECT * FROM Sales.SalesPerson;
```

SQL Server 只跳过（跳过读）一行，返回 *Department* 表的其余 15 行，后跟所有的 *SalesPerson* 行。READPAST 提示经常与 TOP 子句一起使用，具体来说是数据 TOP 1 子句，表在该子句中充当工作队列。*SELECT* 必须得到一个包含待处理订单的行，但是具体是哪一行并不重要。所以 *SELECT* TOP 1 * FROM <OrderTable>返回第一个未锁定的行，您可以将该行作为开始处理的行。

（4）打开第四个连接，并执行以下语句：

```
USE AdventureWorks2008 ;
SELECT * FROM HumanResources.Department (READUNCOMMITTED);
SELECT * FROM Sales.SalesPerson;
```

在本例中，SQL Server 不跳过任何行。它读取 *Department* 表的所有 16 行，但是 Department 1 那一行显示您在第一步更改的脏数据。该数据还没有提交，将被回滚。

由于行版本控制的可用性，READUNCOMMITTED 提示可能是最没有用的。事实上，每当您觉得自己需要使用该提示或等价的 NOLOCK 时，都应该考虑自己是否真的能承受使用某个基于快照的隔离级别的代价。

10.9 小结

SQL Server 让您可以同时管理多个用户，并确保事务遵守所选隔离级别的属性。锁定保护数据和内部资源，使得多用户系统能够像单用户系统一样操作。您可以选择让自己的数据库和应用程序使用乐观或悲观并发控制。对于悲观并发，数据修改操作获得的锁阻塞试图检索该数据的用户。对于乐观并发，锁被忽略，读取的是数据较老的已提交版本。在本章中，我们介绍了 SQL Server 中的锁定机制，包括数据和叶级索引页的完全锁定，以及针对内部使用的资源的轻量级锁定机制。我们还详细介绍了乐观并发如何避免锁阻塞却仍然能够访问数据。

如果想要设计和实现高并发性应用程序，理解锁兼容性和升级问题很重要，还需要了解两种并发模型各自的代价和优点。

第 11 章 DBCC 揭秘

Paul Randal

当大家提到检查 SQL Server 数据库的一致性时，首先会想到“DBCC”。在 SQL Server 7.0 中，DBCC 代表数据库一致性检查器，但是 Microsoft 在后续版本 SQL Server 2000 中将定义修改为数据库控制台命令。这一变化反映了 DBCC 命令家族的工作已经不仅仅是检查一致性了，还包括帮助消除“SQL Server 本身会引起中断，所以 SQL Server 数据库需要常规一致性检查”这样一种看法。

虽然 SQL Server 本身不会引起数据库中断，但是 I/O 子系统（SQL Server 缓冲池和磁盘驱动器金属氧化物之间的所有软件和硬件）确实会引起大量的中断。因此，谨慎执行常规一致性检查是一种明智的选择，因为所有数据库服务器都有某种类型的 I/O 子系统。我经常会被问到有关*常规*的定义问题，实际上常规是根据 I/O 子系统的完整性决定的。一般来说，每周执行一次一致性检查是可以接受的。

*一致性检查*是检查数据库物理和逻辑结构的过程，用于保证没有中断可以防止存储引擎处理部分数据库或者导致某些不正确的行为。下面是一些简单的示例：

- 持久值已经被中断而不再与计算结果相匹配的一个持久计算列；
- 页眉中的页 ID 不正确的一个数据页；
- 记录的键顺序不正确的一个索引。

SQL Server 中的一致性检查功能从 SQL Server 7.0 开始增强了许多，当时是脱机运行的（也就是说，需要表锁定）。SQL Server 2000 默认引入了联机检查，使用一种新的高效机制扫描数据库。在 SQL Server 2005 中，存储引擎内部的一致性检查和修改代码明显被重写和改进了。这样可以处理大量 SQL Server 新功能和重写子系统，同时用于提高性能、可靠性及一致性检查和修复本身的功能。SQL Server 2008 添加了新功能并进一步调整了性能和可伸缩性。

最全面的在数据库上执行一致性检查的方式是使用 *DBCC CHECKDB* 命令。使用 *DBCC CHECKDB* 的主要步骤如下。

（1）创建一种事务一致性的数据库静态视图。

（2）对主要的系统目录执行低级一致性检查。

（3）对数据库执行分配一致性检查。

（4）对数据库中的每个表执行一致性检查。

（5）只要在前面的步骤中没有发现问题，就执行下面的跨表一致性检查。

- 对 Service Broker 元数据执行一致性检查。
- 在各种系统目录之间执行一致性检查。
- 对索引视图执行一致性检查。
- 对 XML 索引执行一致性检查。
- 对空间索引执行一致性检查。

（6）输出。

如果需要，修复可以在不同步骤下完成，但是用户指定一个修复选项时才能进行。

在本章中，我们将利用下面列出的步骤介绍 *DBCC CHECKDB* 的内部是如何在 SQL Server 2008 中工作的。对于可以指定的每一个选项，我们将介绍它会如何影响 *DBCC CHECKDB* 的行为。最后，我们将介绍如何修复工作及其他 DBCC 一致性检查命令。

11.1 获得数据库的一致性视图

数据库的一致性视图是必要的，因为 *DBCC CHECKDB* 必须分析数据库中的所有已分配页并检查多个页结构之间的各种连接。也就是说，被分析的页（即整个数据库）在运行一致性检查期间不能更改，否则 *DBCC CHECKDB* 会报告所有类型的非正确结果。由于 *DBCC CHECKDB* 不能同时读取数据库中的所有已分配页，因此必须维护数据库的一致性视图以保证一致性检查的持久性。对于数据库来说，及时、简单地冻结也是不够的——数据库的一致性视图必须也是事务一致的，从而使 *DBCC CHECKDB* 看到视图中没有未完成的更改。

下面是一个示例：假设一个事务要向具有非聚集索引的表中插入一条记录，假定的一致性检查进程并行运行（不强制数据库有一致性视图）。查询处理器的工作方式是首先插入表记录，然后插入匹配的非聚集索引记录。由于假定的一致性检查进程没有一致性视图，因此可能读取表中的记录但不会读取非聚集索引中的记录，从而生成一条非聚集索引与表不一致的报告。

这是如何实现的呢？正如我们将在本章后面会看到的那样，*DBCC CHECKDB* 按特殊的顺序读取数据库页来提高性能。利用这种机制并继续该示例，可能会在非聚集索引记录被插入之前读取非聚集索引页，但是要在表记录插入之后读取表页。接下来可能会得出存在故障的结论，但实际上问题在于 *DBCC CHECKDB* 看到的是正在执行事务的部分结果。

11.1.1 获得一致性视图

在 SQL Server 7.0 中，事务一致性视图通过在数据库中获得各种级别的锁而取得。这对工作负载的性能来说非常不利，因此 SQL Server 2000 引入了联机一致性检查并删除了保持阻塞锁的需要。*DBC CHECKDB* 在扫描数据库之后分析事务日志，同时主要在数据库内部视图上运行恢复，从而生成数据库的事务一致性视图。

SQL Server 2000 解决方案从很多方面来说都是不实用的，因此在 SQL Server 2005 中通过数据库快照所替代，这种机制与 SQL Server 2008 相同。也就是说，*DBCC CHECKDB* 使用复杂性大大降低的标准存储引擎功能。

正如前面第 3 章中介绍的那样，数据库快照在空间上是非常有效的，只包含自数据库快照被创建以来修改过的数据库页。数据库快照的内容和数据库中未修改页相结合提供了一种数据库未更改的、事务一致性视图。

这是 *DBCC CHECKDB* 需要联机运行的功能。创建一个数据库快照并在该数据库快照上运行一致性检查算法从概念上来说与在数据库的一个只读副本上运行一致性检查算法是一样的。

使用 *DBCC CHECKDB* 可以创建不能被用户访问的数据库快照——基本上是隐藏的。隐藏的数据库快照的创建方式与标准数据库快照的创建方式稍有不同。一个标准的数据库快照具有一个与资源数据库中每个数据文件相对应的快照文件，每个文件必须在数据库快照被创建时明确地命名。*DBCC CHECKDB* 不允许任何用户为隐藏的数据库快照指定文件名，因此它为每一个现有源数据库数据文件创建一个 NTFS

可选流。您可以认为一个可选流是可以通过指向用户可见文件的文件系统进行访问的一个隐藏文件。这种机制工作效果很好并且对用户是透明的。

1. 磁盘空间问题

有时当隐藏的数据库快照用完空间时会出现一个问题。由于数据库快照是使用可选现有数据文件流实现的，因此它会在现有数据文件位置使用相同的空间。如果在正在检查的数据库上有较大的更新工作负载，则会有越来越多的页压入到数据库快照中，从而引起数据库快照不断增长。在磁盘上没有太多空间容纳数据库的情况下，可能引起隐藏的数据库快照耗尽空间，同时 *DBCC CHECKDB* 出错而停止运行。下面显示了一个示例（根据数据库快照耗尽空间的具体点不同，错误可能会有所不同）：

```
DBCC CHECKDB ('SalesDB2') WITH NO_INFOMSGS, ALL_ERRORMSGS;
GO
Msg 1823, Level 16, State 1, Line 5
A database snapshot cannot be created because it failed to start.
Msg 1823, Level 16, State 2, Line 1
A database snapshot cannot be created because it failed to start.
Msg 7928, Level 16, State 1, Line 1
The database snapshot for online checks could not be created. Either the reason is given in
a previous error or one of the underlying volumes does not support sparse files or alternate
streams. Attempting to get exclusive access to run checks offline.
Msg 5128, Level 17, State 2, Line 1
Write to sparse file 'C:\SQLskills\SalesDBData.mdf:MSSQL_DBCC20' failed due to lack of disk
space.
Msg 3313, Level 21, State 2, Line 1
During redoing of a logged operation in database 'SalesDB2', an error occurred at log record
ID (1628:252:1). Typically, the specific failure is previously logged as an error in the
Windows Event Log service. Restore the database from a full backup, or repair the database.
Msg 0, Level 20, State 0, Line 0
A severe error occurred on the current command. The results, if any, should be discarded.
```

这里的解决方案是创建您自己的数据库快照，将快照文件放在具有更多磁盘空间的卷上，然后在该卷上运行 *DBCC CHECKDB*。*DBCC CHECKDB* 发现自己已经在一个数据库快照上运行，同时不会试图创建另一个数据库快照。

如果一个数据库快照是由 *DBCC CHECKDB* 创建的，则它会在一致性检查算法完成时自动被抛弃。

在创建一个数据库快照的同时（如果需要），*FILESTREAM* 垃圾收集器进程会在 *DBCC CHECKDB* 运行时被挂起。这样允许一致性检查算法查看所有 *FILESTREAM* 数据容器上 *FILESTREAM* 数据的事务一致性视图，具体内容将在本章后面详细介绍。

2. 使用数据库快照的替代项

在如下情况下不需要数据库快照。

- 指定的数据库是数据库快照本身。
- 指定的数据库是只读的，处于单用户模式或紧急运行方式。
- 服务器是在单用户模式下利用带有-line 选项的-m 命令启动的。

在这些情况下，数据库基本上已经是一致的，因为可能没有其他活动连接打破一致性检查的更改。

数据库快照不能在如下情况下创建。

- 指定的数据库存储在一个非 NTFS 文件系统上（此时，数据库快照不能被创建，因为数据库快

照依赖于 NTFS 稀疏文件技术)。

- 指定的数据库是 *tempdb*(因为数据库快照不能在 *tempdb* 上创建)。
- 指定了 TABLOCK 选项。

如果由于某种原因不能创建数据库快照,则 *DBCC CHECKDB* 会试图使用锁来获取数据库的事务一致性视图。

首先,*DBCC CHECKDB* 获得数据库级排他锁,从而能够在不进行任何更改的情况下执行分配一致性检查。这不能在 *master* 上进行,也就是说脱机一致性检查不能在 *master* 上运行。也不能在 *tempdb* 上进行,也就是说分配一致性检查总是跳过 *tempdb*(在 SQL Server 2000 上通常也是如此)。*DBCC CHECKDB* 不会无限期地(或服务器锁超时周期所设置的时间)等待排他锁,而是会在等待 20 秒(或为会话配置的锁超时时间)后返回如下错误信息并退出:

```
DBCC CHECKDB ('msdb') WITH TABLOCK;
GO
Msg 5030, Level 16, State 12, Line 1
The database could not be exclusively locked to perform the operation.
Msg 7926, Level 16, State 1, Line 1
Check statement aborted. The database could not be checked as a database snapshot could not
be created and the database or table could not be locked. See Books Online for details of
when this behavior is expected and what workarounds exist. Also see previous errors for more
details.
```

如果获取了锁,在分配检查完成后,排他锁会被删除并在标记逻辑一致性检查时获得表级共享锁。同样的超时会应用到这些表级锁上。

DBCC CHECKDB 会以某种方式包含它正在检查的数据库的事务一致性视图。此后,它可以开始处理数据库。

11.2 有效地处理数据库

一个数据库可以被认为是一个巨型的互联结构,其中所有表均被连接到系统目录上,同时所有系统目录连接到存储在 *sys.sysallocunits* 中的最低级分配元数据上,元数据将它的第一个页固定在每个数据库中的页(1:16)上,外加固定位置的分配位图(如页可用空间[PFS]和全局分配表[GAM]页),整个数据库都可以表示为一个单一的实体关系图。

带着这种想法,您可以为这种从页(1:16)和分配位图开始并且逐渐扩展到对象和结构之间的数据库检查链接的元结构设想一个一致性检查算法。不论何时发现一个页链接,都会跟随这个链接以保证链接到正确的页。不论何时一个分配位图将一个页标记为一个 IAM 页,该页都会被检查以保证它实际上是一个 IAM 页。这将是一种*深度优先算法*。

假设一个数据页有 3 条数据记录,每条数据记录都包含到行外存储的两个 8000 字节大型对象(LOB)列的一个链接。使用以前的算法,对一致性检查页的操作包括如下步骤:

(1)从记录 1 中提取包含第一个 LOB 列数据的页 ID。

(2)读取该页,保证存储的是正确的 LOB 列数据。

(3)从记录 1 中提取包含第二个 LOB 列数据的页 ID。

(4)读取该页,保证页上存储的是正确的 LOB 列数据。

(5)根据需要重复步骤 1~步骤 4,直到处理完整个结构为止。

您会发现这种算法非常低效。页根据需要读取并且基本上是按照随机顺序读取。页可能被处理多次，而且页读取的随机性意味着 I/O 子系统不能用做预读取。就算法复杂性而言，该算法的复杂度是 $O(n^2)$。这可以被称做“*n* 的平方级”，即随着操作数的增加，算法的运行时间按指数级增长。这里的 *n* 代表数据库中页的数量。

这种方法不是 SQL Server 2008（实际上是从 SQL Server 2000 开始）中 *DBCC CHECKDB* 的工作方式。这种 $O(n^2)$算法对于大型数据库来说运行代价非常大。相反，*DBCC CHECKDB* 使用一种复杂度为 $O(n*\log(n))$的算法，这种算法提供一种近似线性的缩放比例。接下来介绍这种算法。

11.2.1 事实生成

DBCC CHECKDB 以最有效的方式从正在进行一致性检查的对象中读取所有页——按照分配顺序（即按照它们在数据文件中的存储顺序）而不是按照页连接随机顺序读取。这种机制将在本章后面介绍。

由于页是按照严格的分配顺序读取的，因此没有办法在页被处理时马上验证页间的所有关系。因此，*DBCC CHECKDB* 必须记住它所知道的每个页的情况，从而可以在后面阶段执行这种关系检查。*DBCC CHECKDB* 是通过生成被称为“事实”的页信息位实现的。

继续前面的示例（作为处理数据页的一部分），会生成如下事实。

- 第一条记录连接到一个 LOB 值（每个 LOB 值为一个事实）的两个事实。每个事实包含：
  - 数据记录的页 ID 和槽 ID（即记录编号）;
  - 应该存储 LOB 值的页 ID 和槽 ID（从存储在数据记录中的文本根部提取）;
  - LOB 值的文本时间戳（即分配给 LOB 值的一个唯一 ID）;
  - 页所在的对象 ID、索引 ID、分区 ID 和分配单元 ID。
- 第二条记录连接到一个 LOB 值的两个事实。
- 第三条记录连接到一个 LOB 值的两个事实。

这些事实被称做*父文本*事实。

包含实际 LOB 值的每个文本页被处理时，处理过程会生成遇到 LOB 值的一个事实。每个事实包含：

- 文本记录的页 ID 和槽 ID;
- LOB 值的文本时间戳;
- 页所属的对象 ID、索引 ID、分区 ID 及分配单元 ID。

这些事实被称为*实际文本*事实。

后面的某个时候事实之间会相互检查（称为*聚合*）。只要每个 LOB 值有一个匹配的文本事实和实际文本事实，*DBCC CHECKDB* 就会认为这个特殊的 LOB 值连接是自由删除的。

除实际事实和父事实之外，还有一种被称为*同级*事实的事实，这种事实在检查索引 B 树连接时使用，用于描述索引 B 树每个级别中存在的链接列表。

数据库结构不同部分使用的一致性检查算法使用各种事实类型和事实内容，但是基本算法是相同的。使用的这些事实类型如下。

- 收集对象、索引、分区和分配单元的分配统计信息的事实。
- 跟踪 *FILESTREAM* 数据的事实。
- 跟踪 IAM 链连接的事实。
- 跟踪某个特殊 GAM 间隔的 IAM 页位图的事实。
- 跟踪数据库文件的事实。

- 跟踪扩展分配和所有权的事实。
- 跟踪页分配和所有权的事实。
- 跟踪B树连接的事实。
- 跟踪LOB值连接的事实。
- 跟踪LOB值连接的事实。
- 跟踪堆中转发记录的事实。

在生成和聚合期间，事实存储在内存用于排序操作的查询处理器中。有时这种排序的大小比查询处理器可用的内存更大，因此这种操作会向磁盘进行写入（写入到*tempdb*数据库中），从而产生对*tempdb*的物理读取和写入操作（有时这种操作是很明显的）。由于每个事实基本上是表的一行，因此事实必须分成表的列。每个事实由5列构成。

- **ROWSET_COLUMN_FACT_KEY。**事实描述的页的页ID或*FILESTREAM*文件的LSN。
- **ROWSET_COLUMN_FACT_TYPE。**事实类型。
- **ROWSET_COLUMN_SLOT_ID。**事实描述的记录的槽ID（如果有的话）。
- **ROWSET_COLUMN_COMBINED_ID。**页所在的对象、索引、分区和分配单元ID。
- **ROWSET_COLUMN_FACT_BLOB。**存储任何所需的额外数据的变长列。

如果*tempdb*没有足够的空间存储DBCC排序，则*DBCC CHECKDB*可能会失败。如果发生这种情况，则会显示如下的8921错误：

```
Msg 8921, Level 16, State 1, Line 1
Check terminated. A failure was detected while collecting facts. Possibly tempdb out of
space or a system table is inconsistent. Check previous errors.
```

11.2.2 使用查询处理器

*DBCC CHECKDB*充分利用了查询处理器——既轻松处理事实又容易实现并行一致性检查。

用于生成事实的算法如下。

（1）DBCC代码向查询处理器发出一个查询（利用仅SQL Server内部可用的语法），包含一个指向行集的指针和一个自定义聚合函数的名称。

（2）查询处理器在行集中查询某一行，主要是调用DBCC获取一个要处理的事实。

（3）DBCC返回一个事实作为一个行集行（使用前面介绍的列结构）。如果没有可用的事实，则DBCC读取一个页并完全处理该页，生成所有必要的事实。事实被存储在线程局部的内存中作为一个先进先出（FIFO）的队列，同时一个事实会被返回给查询处理器。从线程局部的事实队列的头部返回一个事实（具有查询处理器的每个后续请求），直到没有可用的事实为止。只有此时才会读取和处理另一个页，从而生成事实，再次填充事实队列。

（4）查询处理器在排序内存的内部存储事实，也可能在*tempdb*数据库中。

一旦为正在进行一致性检查的对象生成了所有对象，则查询处理器会完成在这些对象上的排序操作并调用DBCC提供的自定义聚合函数。事实按照事实键排序并按照该类型之外的所有列进行分组，从而使聚集程序按照正确的顺序获得事实以轻松地匹配连续的事实。

继续前面的LOB连接示例，事实按照如下顺序传回聚集程序。

（1）LOB值的实际文本事实（实际上是数据页上记录1的LOB值1）。

（2）数据页上记录1的LOB值1的父文本事实。

（3）LOB值的实际文本事实（实际上是数据页上记录1的LOB值2）。

（4）数据页上记录 1 的 LOB 值 2 的父文本事实。

如果事实不是按照这样的顺序，那么要再次记住已经看到的内容才能匹配事实。

聚合算法按照如下方式运行。

（1）查询处理器用一个事实调用 DBCC 自定义聚合函数。

（2）事实被合并，直到从查询处理器传递一个与正在被合并的事实不匹配的事实。例如，前面示例中 LOB 值为 1 的事实和父事实被合并。下一个事实用于 LOB 值 2，这是用于数据库结构的一个不同部分。

（3）一旦遇到某个不匹配的事实，则合并事实会形成聚合来确定是否有错误存在。聚合表示事实被检查以查看它们所描述的数据库结构是否存在正确的事实。例如，一个 LOB 值必须有一个实际事实（实际上遇到该值）和一个父事实（某个索引或数据记录连接到 LOB 值）。

（4）如果有错误，则会在错误列表中显示。错误项是利用包含在聚合事实集中的信息生成的。示例错误是没有一个指向它的数据或索引记录的 LOB 值。

（5）事实接下来被抛弃并且一组新的合并事实开始，从触发聚合的不匹配事实开始。

（6）DBCC 代码接下来向已经准备好接收下一个要合并和聚合的事实发出信号。

图 11-1 显示了 *DBCC CHECKDB* 执行时查询处理器和 DBCC 代码的交互方式。

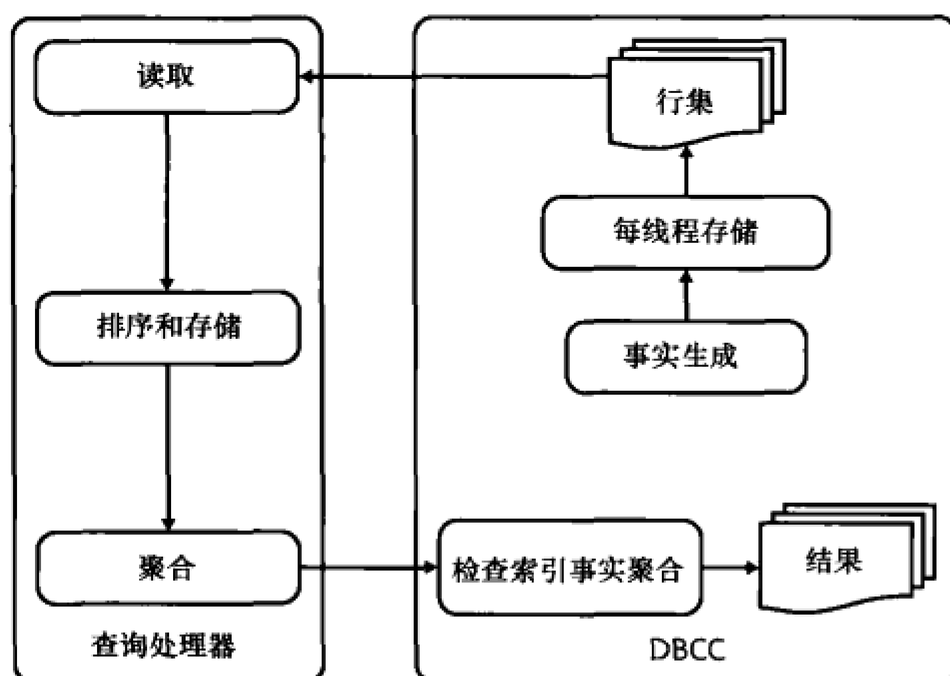


图 11-1 查询处理器和 DBCC 之间的交互

这些算法在 *DBCC CHECKDB* 内部执行查询时开始工作。*DBCC CHECKDB* 运行的查询如下：

```
DECLARE @BlobEater VARBINARY(8000);
SELECT @BlobEater = CheckIndex(ROWSET_COLUMN_FACT_BLOB)
FROM <memory address of fact rowset>
GROUP BY ROWSET_COLUMN_FACT_KEY
>> WITH ORDER BY
     ROWSET_COLUMN_FACT_KEY,
     ROWSET_COLUMN_SLOT_ID,
     ROWSET_COLUMN_COMBINED_ID,
     ROWSET_COLUMN_FACT_BLOB
OPTION(ORDER GROUP);
```

该查询将查询处理器和 *DBCC CHECKDB* 代码结合起来执行事实生成、事实排序、事实存储及事实

聚合算法。该查询的部分如下。

- **@BlobEater**。这是一个哑变量，除使用 CheckIndex 函数的输出之外没有其他作用（除语法需要外，不应该有该变量）。
- ***CheckIndex（ROWSET_COLUMN_FACT_BLOB）***。这是 DBCC *CHECKDB* 内部的一个自定义聚合函数，查询处理器利用排序和分组的事实作为整个事实聚合算法的一部分进行调用。
- **<memory address of fact rowset>**。这是 *DBCC CHECKDB* 为查询处理器提供的 OLEDB 行集的内存地址。查询处理器查询行集（包含生成的事实）作为整个事实生成算法的一部分。
- ***GROUP BY ROWSET_COLUMN_FACT_KEY***。触发查询处理器中的聚合。
- **>> WITH ORDER BY <column list>**。这是为聚合步骤提供有序聚合的内部语法。正如我们在前面介绍的那样，*DBCC CHECKDB* 聚合代码是建立在查询处理器事实聚合流的顺序被强制这一假设基础之上的（也就是说，要求每一组内键的排序顺序就是查询中 4 个键的顺序）。
- **OPTION(ORDER GROUP**。这是强制流聚合的查询优化器暗示。该选项强制查询优化器按照分组列排序同时避免哈希聚合。

这种机制用于分配一致性检查及 *DBCC CHECKDB* 按表一致性检查的各阶段。此外，这也表示 *DBCC CHECKALLOC*、*DBCC CHECKTABLE*、*DBCC* 和 *CHECKFILEGROUP* 命令也使用相同的机制。

如果由于缺少内存而导致内部查询失败，则会报告 8902 错误。如果由于其他原因而使内部查询失败，则会报告 8975 错误。在这两种情况下 *DBCC CHECKDB* 都会终止。

11.2.3　批处理

在按表进行逻辑检查的阶段，*DBCC CHECKDB* 通常不会一起处理数据库中的所有表，也不会一次只处理一个表。而是会将表分成组（批）并在每组中的所有表上运行事实生成和聚合算法。在所有批处理完成后，数据库中的所有表都进行了一致性检查。

DBCC CHECKDB 将数据库分成一系列批处理的原因是要限制 *tempdb* 中存储事实所需的空间数量。生成的每个事实都会占用一定的空间，占用空间的数量根据事实的类型和内容的不同而有所不同。模式越复杂，要求生成的事实就越多（从而允许表模式的各个方面都进行一致性检查）。

您可以想象，对于一个非常大型的数据库来说，如果在一个批处理中要对数据库中的所有表进行一致性检查，则存储所有事实所需的空间数量会迅速超出 *tempdb* 的存储能力。

DBCC CHECKDB 在按表逻辑检查阶段之初扫描表元数据时确定一个批处理中的表。批处理至少有一个表（及所有非聚集索引）并且每个批处理的大小都是由如下规则限制的。

- 如果指定了修复选项，则在包含一个表时构建批处理的过程会停止。这样可以保证修复按照正确的顺序进行。
- 当一个表被添加到一个批处理中并且该批处理中的所有表的索引总数超过 512 时，批处理的构建过程会停止。
- 当一个表被添加到一个批处理中并且该批处理中所有事实的所有表所需的 *tempdb* 空间超过 32MB 时，批处理的构建过程会停止。

一旦批处理构建完成，则会在批处理中的所有表上运行事实生成和事实聚合算法。也就是说，为每个批处理中的表执行一次前面介绍的内部查询。

当一个批处理结束后，可能会触发各种深度优先算法，从而找到不匹配的文本时间戳值或不匹配的非聚集索引记录。此时未检查程序集也会被清除。

如果一个表根据一个 CLR 程序集实现一个 CLR 用户定义数据类型（UDT）或者计算列，并且该程序集接下来利用带有 WITH UNCHECKED DATA 选项的 *ALTER ASSEMBLY* 语句更改，则依赖于该程序集的所有表在系统目录中都被标记为具有未检查程序集。清除这一设置的唯一机制是在受影响的表上运行 DBCC 一致性检查。如果没有发现错误，则未检查程序集设置会被清除。

11.2.4 读取要处理的页

事实生成和事实聚合算法的部分性能源自有效地读取构成批处理中表和索引的页这一事实。正如我们在前面已经介绍过的那样，页不必按照特殊的顺序读取，因为事实会在所有相关页被读取后（并且所有事实生成后）才形成聚合。

从数据文件读取一组页的最快速方式是按*分配顺序*（数据文件中页的物理顺序）读取。这允许磁盘头在磁盘上进行一次扫描，而不是进行所有随机 I/O 并引起额外的磁盘头查找时间系统开销。

构成批处理中每个表和索引的页和扩展都由表或索引中各种分配单元的 IAM 链进行跟踪。一旦构建完批处理，所有这些 IAM 链都合并成一个大型位图（可以通过扫描 *DBCC CHECKDB* 中的对象管理）。这个位图接下来以*有序的物理顺序*表示所有页和扩展（构成批处理中的所有页和扩展）。

使用这一位图时，所有必需页都可以顺序（接近顺序）地读取。扫描对象对页进行预读从而保证 CPU 不必等待要处理的下一个页读取到缓冲池中。预读机制与剩余存储引擎中使用的机制类似，只是在逻辑数据文件被创建的物理卷上以一种轮流的方式进行。这样会在物理卷上传播 I/O 工作负载，好像大量的 I/O 是通过 *DBCC CHECKDB* 完成的。

不论何时下一个页需要处理，都会对扫描对象（接下来为调用程序返回一个页）进行一次调用。返回的页类型或页所在的对象/索引是完全无关的，由事实生成和聚合算法的本性决定。

注意有时随机 I/O 是必要的，因为正在被读取的页上的某些行可能使行的一部分存储在一个不同的页上（由于行溢出特性）。*DBCC CHECKDB* 实现内存中的一整行（除行外 LOB 列外），这可能包含一次随机 I/O 以读取行的行溢出部分。

11.2.5 并行性

DBCC CHECKDB 有能力利用多个处理器内核并行运行，从而更有效地利用系统资源并更快速地处理数据库。

如果下面的所有条件都是真，则可以并行运行当前批处理。

- SQL Server 实例是企业版、Enterprise Eval 版或开发人员版。
- 有多于 64 个页构成当前批处理中的所有表和索引。
- 批处理表中没有基于 T-SQL 或 CLR 的计算列。
- 并行性没有显式禁用跟踪标志 2528。

如果所有这些条件都是真，则 *DBCC CHECKDB* 在处理前面介绍过的内部查询时会向查询处理器发出可以进行并行处理的信号。查询处理器接下来做出是否使用并行线程的最终决定。查询处理器根据影响 SQL Server 中所有其他查询并行性的因素做出是否使用并行线程的决定，如：

- 服务器的 MAXDOP 设置；
- 并行性的预期查询代价；
- 在 *DBCC CHECKDB* 为执行而编译对批处理的查询时服务器上资源的可用性。

每次查询被处理时执行是否并行化内部查询的决定，这表示 *DBCC CHECKDB* 一次执行中的不同批

处理可以利用不同等级的并行性运行。

内部查询并行运行时数据的概念流如图11-2所示。该图显示了并行级别为3时的数据流。

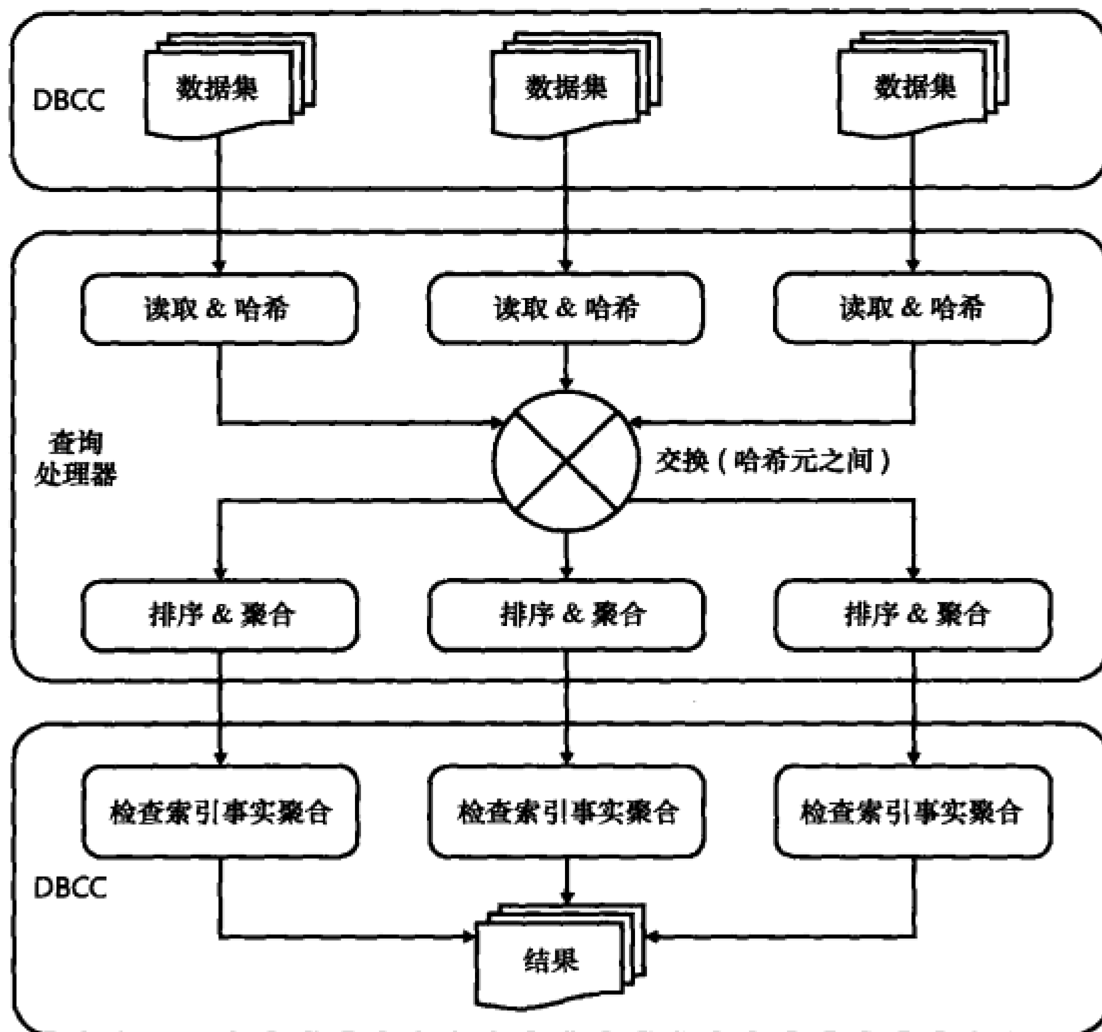
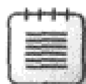


图 11-2 并行级别为3时内部查询的概念数据流

内部查询并行运行时，会为并行性的每个级别创建一个线程。在算法的事实生成期间，每个线程都有责任从扫描对象中请求要处理的页并彻底处理每个页。一个页只由一个线程处理。在算法的事实聚合部分，每个线程负责聚合一个单独的自包含事实流（表示对象结构的一个特殊部分的所有相关事实必须提供给一个线程——没有跨线程的事实聚合）。

对于哪个线程处理哪个页不进行控制，因此包含单个对象的页可以由多个线程处理。在事实聚合阶段，如果真正应该聚合到一起的事实实际上包含在不同线程的排序和聚合事实流中则可能会引起问题。因此在并行执行内部查询时，所有事实都由它们的ROWSET_COLUMN_FACT_KEY元素进行哈希并且通过一个交换运算符在存储之前传递到每个线程的哈希元中。这样可以保证某种对象结构一个特殊部分的所有事实都只提交给一个线程。



注意：

DBCC CHECKTABLE 和 *DBCC CHECKFILEGROUP* 命令也可以以这种方式使用并行性。但 *DBCC CHECKALLOC* 不能并行运行。

如果并行命令给服务器带来了太大的工作负载，则会利用跟踪标志2528禁用所有这些DBCC命令

的并行性。注意禁用并行性会使 DBCC 命令花费更长的时间才能完成。

为了在没有人为瓶颈的情况下允许有效的并行性和可伸缩性，多线程感知（例如，扫描对象和进程报告对象）的 *DBCC CHECKDB* 内部所有部分都是这样设计的，从而从多线程访问它们不会引起伸缩性问题（至少在跨 32 个处理器核心进行并行处理的情况下）。

11.3 早期的系统目录一致性检查

SQL Server 2008 中存储引擎定义 3 个系统目录作为操作的关键目录。这些表是：

- *sys.sysallocunits*
- *sys.sysrowsets*
- *sys.sysrscols*

在 SQL Server 2005 中，还有两个更关键的系统表——*sys.syshobts* 和 *sys.syshobtcolumns*，但是它们已经分别合并到了 *sys.sysrowsets* 和 *sys.sysrscols* 表中。它们与原有的 *sysindexes*、*sysobjects* 和 *syscolumns* 是等价的。它们存储存储引擎需要对表和索引结构进行导航的所有基元数据。*DBCC CHECKDB* 也利用这些表实现这一目的，不过间接地通过关系引擎中的元数据子系统。

这些系统目录都有一个聚集索引，并且有些还有非聚集索引。*DBCC CHECKDB* 需要检查聚集索引叶级没有明显的破坏，从而在调用一个元数据函数从中检索信息时，元数据函数成功的可能性更大。

下面的检查是在这 3 个聚集索引叶级上执行的。

- **每个页被读进缓冲池。**这将检查页没有 I/O 问题（如页校验和失败、无效的页 ID 或 I/O 子系统读取页失败）。在这种操作失败时会为页报告 7985 错误。
- **每个页被审核。**页审核将在本章后面介绍，简单地说，页审核可以保证页结构和页眉看起来是有效的。该页必须是一个数据页并且必须分配给正确的分配单元。在这种检查失败时会为页报告 7984 错误。
- **叶级链表被检查。**索引中某个级别上的所有页都处于一个双向链表中。一旦叶级的所有页都已经被读取到缓冲池并且已经被审核，则会通过叶级的下一页联接检查页联接，同时保证上一页联接真正指向上一页。如果链表中断则会报告 7986 或 7987 错误。
- **检查叶级链表的循环。**这是在链表联接被检查时通过将两个指针指向页链表的方式（一个指针每隔一步向前移动一次，另一个指针每隔两步向前移动一次）实现的。如果在移动速度更快的指针到达叶级的右手边之前，两个指针曾经指向同一个页，则出现一次循环。重要的是没有联接循环，否则一序列扫描可能会变成一个无限的循环。我们从来没有在客户系统上看到过这种情况出现。如果检测到一次循环则会报告 7988 错误。

如果其中任意一种检查失败，则 *DBCC CHECKDB* 都会终止并返回适当的消息，如：

```
DBCC CHECKDB ('TestDB') WITH NO_INFOMSGS, ALL_ERRORMSGS;
GO
Msg 7985, Level 16, State 2, Line 1
System table pre-checks: Object ID 4. Could not read and latch page (1:65) with latch type
SH. Check statement terminated due to unrepairable error.
DBCC results for 'TestDB'.
Msg 5233, Level 16, State 98, Line 1
Table error: alloc unit ID 262144, page (1:65). The test (IS_OFF (BUF_IOERR, pBUF->bstat))
failed. The values are 12584969 and -4.
```

```
CHECKDB found 0 allocation errors and 1 consistency errors not associated with any single
object.
CHECKDB found 0 allocation errors and 1 consistency errors in database 'TestDB'.
```

上面的检查之所以会终止是因为这些主要的系统目录是 *DBCC CHECKDB* 检查其他数据库所需要的。注意在 *DBCC CHECKDB* 输出中没有推荐的修复级别。这些错误不能被修复——唯一的选项是从备份还原。

如果没有发现问题，则下一个阶段是运行数据库级分配一致性检查（这一内容将在下节介绍）。

11.4 分配一致性检查

这些检查验证跟踪数据库中页和扩展分配各种结构之间的内容和关系。包含的结构如下。

- PFS 页，跟踪一个数据文件 64MB 内（一个 PFS 间隔）个别页的分配状态。
- GAM 页，跟踪一个数据文件 4GB 内（一个 GAM 间隔）所有扩展的分配状态。
- SGAM 页，跟踪所有混合扩展（至少有一个页对有一个 GAM 间隔的分配可用）。
- IAM 页，跟踪从一个 GAM 间隔返回的分配给分配单元的所有页和扩展。
- IAM 链，是分配单元所有 IAM 页的一个链表（因此跟踪所有页和扩展分配给所有数据文件所有部分的分配单元）。
- 3 个主要系统表中的存储引擎元数据，如前所述。

分配一致性检查是非常快的——实际上比单表和跨表一致性检查快很多。原因在于必须读取以执行分配一致性检查的数据库页数量比必须被读取以执行所有单表和跨表一致性检查的数量少很多。

11.4.1 收集分配事实

在所有分配一致性检查可以运行之前，所有必要的信息需要从各种分配结构中进行收集并作为事实存储。

对于数据库中每个联机文件组中的每个数据文件来说，会执行如下的操作。

- 数据库的启动页[文件 1 中的页（1:9）]和每个数据文件中的文件标题页被审核。如果这种检查失败并且 *DBCC CHECKDB* 终止，则会报告 5250 错误。
- 所有 PFS 被读取并处理。这会提供文件中所有 IAM 页的位图，同时 PFS 页也跟踪哪些页是 IAM 页。这也提供文件中哪些页是来自混合扩展的位图。每组位图比文件中的所有 PFS 页占用更少的空间，因为 PFS 页为每个数据文件页存储 8 位信息。
- 所有 GAM 页均被读取和处理。这将提供文件中所有分配扩展的一个位图。
- 所有共享全局分配映射（SGAM）页均被读取和处理。这会提供至少有一个可用页的文件内的所有混合扩展的一个位图。
- GAM 扩展被处理时增量更改映射（DCM）页和简单记录映射（ML Map）页被读取，只是为了保证它们可以被正确读取。

在每个 GAM 间隔的开始处是一个被称为 GAM 扩展的特殊扩展，包含该 GAM 间隔的 GAM 页和 SGAM 页。也包含跟踪该 GAM 间隔中扩展的两个其他页——DCM 页和 ML Map 页。DCM 页跟踪自最后一个完整数据库备份以来 GAM 间隔中哪些扩展被修改。ML Map 页跟踪自最后一次事务日志备份以来通过简单记录操作修改了 GAM 间隔中的哪些扩展。

- 所有 IAM 页被读取和处理，这将提供以下信息。
  - 文件中所有混合页（混合扩展分配的页）的一个列表（请记住 IAM 链中第一个 IAM 页包含一个阵列（单页槽阵列），最多可存储它所表示的分配单元的 8 个混合页）。
  - 文件中所有有效 IAM 页的一个列表。这是必要的，因为一个 PFS 页可能是有错误的并且错误地将一个页标记为一个 IAM 页，或者一个真实的 IAM 页可能只是错误的和不可读取的。
  - 文件中所有分配的专用扩展列表。
  - 所有 IAM 链的联接信息。

IAM 链中所有 IAM 页都联接在一个双向链表中。它们也包含一个序列号，链中第一个 IAM 页的序列号从 0 开始，并且添加到链中的每个 IAM 页的序列号按增量 1 增长。

如果由于分配页的标题是错误的而使分配页不能被读取，则会报告 8946 错误（或为错误 IAM 页报告 7965 错误）。这表示数据库的一些列会排除在一致性检查之外。排除的序列是在错误 8998 中报告的。

在所有文件信息收集之后，存储引擎元数据按如下方式处理。

（1）每个 IAM 链中第一个 IAM 页的页 ID 被存储在系统目录中（如果页 ID 没有被存储在某个位置，则存储引擎不知道在哪里找到分配给表或索引的页和扩展列表）。

这些页 ID 用于生成每个 IAM 链中第一个 IAM 页的父事实。这应该与按文件步骤中生成的实际事实表相匹配。

在这一阶段，所有系统目录会被检查以保证它们被存储在数据库的主文件组中。如果有例外，则会报告 8995 错误。

（2）当前等待被“延迟删除”的 IAM 链信息存储在一个内部队列中。

延迟删除是 SQL Server 2005 中引入的一种优化，可以防止事务在删除 IAM 链时用完锁内存。删除具有多于 128 个扩展的 IAM 链（通过删除和重建索引，或者删除和缩短一个表）的进程没有实际页并且在事务提交后扩展解除分配。IAM 链从 *sys.sysallocunits* 上删除并挂到一个内部队列中。

如果 *DBCC CHECKDB* 没有作为分配事实生成进程的一部分扫描内部队列，则可能发现与各种分配位图不一致的各种问题。

分配事实表被传递给查询处理器，如前所述，它们在查询处理器中被排序和分组到一起。它们接下来被传递给 *DBCC CHECKDB*，从而使它们可以被聚合及发现的所有错误。

11.4.2 检查分配事实

分配事实聚合算法执行如下一致性检查。

- 检查每个 GAM 间隔中每个扩展被正确地分配。可能扩展应该：
  - 在 GAM 页中标记为分配可用；
  - 在 SGAM 页中标记为不完整混合扩展；
  - 在覆盖GAM 间隔的某个 IAM 页中进行精确地标记；
  - 不在任何分配位图中标记（其中扩展中的所有页一定是混合页，在各种 IAM 页的单页槽阵列中引用）。

表 11-1 列出了可能的组合，显示了不合法的状态及结果错误编号。

如果有两个 IAM 页分配相同的扩展，则会报告 8904 错误。8904 错误总是伴随 8913 错误，该错误会返回给分配扩展的第二个对象（或分配位图）。如果扩展是一个混合扩展但没有发现混合页，则会报告 8905 错误。

表 11-1 分配位图的可能组合

GAM	SGAM	IAM	合法性	含 义	错 误
0	0	0	Y	所有页被分配的混合扩展	N/A
0	0	1	Y	分配给 IAM 的专用扩展	N/A
0	1	0	Y	具有可用页的混合扩展	N/A
0	1	1	N	不合法	8904
1	0	0	Y	扩展没有被分配	N/A
1	0	1	N	不合法	8904
1	1	0	N	不合法	8903
1	1	1	N	不合法	8904

- 检查每个混合页和 IAM 页的 PFS 字节是正确的。为所有检查出错误的页报告错误 8948。
- 检查由 PFS 页标记为混合页的每个页出现在 IAM 页上单页槽阵列中的某个位置。为所有检查出错误的页报告一个错误 8906，如下所示：

```
Msg 8906, Level 16, State 1, Line 1
Page (1:50139) in database ID 13 is allocated in the SGAM (1:3) and PFS (1:48528), but
was not allocated in any IAM. PFS flags 'MIXED_EXT ALLOCATED 0_PCT_FULL'.
```

- 检查每个混合页只在单个 IAM 页的单页槽阵列中分配。为双重分配的页报告错误 8910。
- 检查 IAM 链中的 IAM 页具有单调递增序列号。如果检查失败则会报告错误 2577。
- 检查同一 IAM 链位图中没有两个 IAM 页具有相同的 GAM 间隔。检查失败时报告错误 8947。
- 检查 IAM 链中所有 IAM 页属于相同的分配单元。检查失败时报告错误 8959。
- 检查 IAM 页位图数据文件的有效部分（例如，不在文件 ID 0 或 ID 2 中）。检查失败时会报告错误 8968。
- 检查 IAM 链中 IAM 页之间链表是正确的，包括从 *sys.sysallocunits* 目录到 IAM 中第一个 IAM 页的指针。如果检查失败则会报告错误 8969、2575 或 2576（根据哪个链接被破坏）。示例如下：

```
Msg 8906, Level 16, State 1, Line 1
Page (1:50139) in database ID 13 is allocated in the SGAM (1:3) and PFS (1:48528), but
was not allocated in any IAM. PFS flags 'MIXED_EXT ALLOCATED 0_PCT_FULL'.
```

- 检查某个 IAM 页将一个 GAM 间隔映射到与自己所在的同一个文件组中的某个位置。如果检查失败，则会报告错误 8996。
- 检查映射到某个文件中的最终 GAM 间隔的所有 IAM、GAM 和 SGAM 页没有将标记扩展为已分配（超出文件的物理终点）。如果检查失败则会报告错误 2579。

分配一致性检查完成（如果指定了修复选项则会执行修复）后，就做好了逻辑一致性检查的准备，接下来对此进行介绍。

11.5 按表进行逻辑一致性检查

这些检查验证表中所有结构及其所有索引的一致性。在这一部分，我们所说的“一个表”是指“堆或聚集索引的所有分区，所有非聚集索引的所有分区及所有行外 LOB 数据”。表可以是一个标准表、一

个索引视图、一个 XML 索引、一个系统目录、一个空间索引、一个 Service Broker 队列，或者是在内部作为一个表存储的其他数据库对象。

利用前面介绍的事实生成和事实聚合算法，单个批处理中的所有表同时被检查。第一个批处理包含主要的系统表，同时指导数据库中的所有表都检查完成后才构建和检查后续批处理。

对每个表执行的一致性检查如下。

（1）提取并检查表的所有元数据。

（2）对表中的每个页执行如下操作。

- 读取并审核页。
- 执行页级一致性检查。
- 对页上的所有记录执行记录级一致性检查。
- 对于数据和索引记录来说，对每个记录中的每一列执行列集一致性检查。

（3）执行如下跨页一致性检查。

- 非聚集索引交叉检查。
- B 树一致性检查。
- 行外 LOB 数据一致性检查。
- *FILESTREAM* 一致性检查。

接下来详细介绍每一个步骤。

11.5.1 元数据一致性检查

DBCC CHECKDB 构建一个说明每个表大部分元数据的内部缓存。这个元数据缓存在各种一致性检查期间被广泛使用，并且对于 *DBCC CHECKDB* 来说访问自己的缓存比连续调用存储引擎元数据子系统的速度快很多。

元数据缓存具有如下的信息层次结构。

- **表元数据对象**。存储说明索引元数据对象的表和链接列表的元数据。
- **索引元数据**。存储表中每个索引的元数据（包括堆或聚集索引）及行集元数据对象的链接列表。
- **行集元数据**。存储描述每个索引每个分区的元数据。

不必列出元数据缓存跟踪的所有信息。相反，我们只列出每个元数据对象中被跟踪的重要项目。

表的元数据缓存对象如下。

- *DBCC CHECKDB* 默认输出中信息性消息中使用的页和记录。
- 在该表中发现的错误数。
- 用于计算持久和索引计算列的预期值的表达式计算器。这是从查询处理器获得的，只要 CLR 没有对实例禁用。
- 状态信息，包括是否发现表包含错误。

如果 CLR 已经被禁用，则不能创建表达式计算器并报告错误 2518。如果 CLR 被启用，但是在初始化表达式计算器时遇到一个问题，则会报告错误 2519。在这两种情况下都不会检查计算列和 UDT。

索引元数据缓存对象如下。

- 有关索引的所有分区函数的所有元数据，每条记录可以被检查，从而保证该记录处于正确的分区中。
- 状态信息，包括是否发现索引包含一个错误。对于非聚集索引来说，如果没有发现错误，则不

会执行非聚集索引交叉检查。

如果正在被关注的索引位于一个文件组上而不是为 *DBCC CHECKFILEGROUP* 指定的文件组上，则不会包含在这些检查中并且会报告错误 2594。

行集元数据缓存对象，包括：

- 各种列和键数量。
- *FILESTREAM* 一致性检查中附加的元数据。

如果一个表或索引在一个脱机或无效的文件组中有一个行集（即索引或表的一个分区），则表或索引不包含在这些检查中。对于脱机文件组来说，会报告错误 2527。对于无效的文件组来说，会报告错误 2522。

在行级元数据缓存对象被构建时，会测试每个分配单元的系统目录页数量以保证它们不是负数。这种情况可能在 SQL Server 2005 以前的 SQL Server 版本中出现。如果发现一个负数，则输出中会报告如下所示的错误 2508：

```
Msg 2508, Level 16, State 3, Line 1
The In-row data RSVD page count for object "Receipts", index ID 0, partition ID
49648614572032, alloc unit ID 49648614572032 (type In-row data) is incorrect. Run DBCC
UPDATEUSAGE.
```

此外，一个独立的哈希表包括表中现有每个分配单元 ID（具有到相关行集元数据缓存对象的链接）。这提供了一种非常快的方式来查找描述特殊页的元数据（因为每个页的页眉中只有一个分配单元 ID），而不必通过元数据缓存执行一种代价很大的搜索。

如果请求对一个特殊的索引进行一致性检查，但是在数据库元数据中找不到该索引，则会报告错误 2591。

在该缓冲被构建时，会检查一致性。如果发现错误（例如，与各种列数量和阵列不匹配），则会输出错误 8901 或 8930，具体取决于错误的严重性。一个错误 8901 防止表被检查，但是一个错误 8930 会引起 *DBCC CHECKDB* 终止。下面是一个示例：

```
Msg 8930, Level 16, State 1, Line 1
Database error: Database 16 has inconsistent metadata. This error cannot be repaired and
prevents further DBCC processing. Please restore from a backup.
```

11.5.2 页审核

所有由 *DBCC CHECKDB* 读取的页（不管页类型如何）都会在进一步处理之前进行审核。审核过程保证页及页上的记录都是正确的，因此进一步的一致性检查算法不会引起 *DBCC CHECKDB* 内部出现问题。

DBCC *CHECKDB* 本身不执行任何物理 I/O 操作——相反，它利用缓冲池读取要处理的所有页。除了降低复杂性之外，还允许 *DBCC CHECKDB* 使用缓冲池的审核。不论缓冲池何时向内存中读入一个页，都会对页进行检查以保证不出现 I/O 错误，同时接下来验证不完整页或页校验和保护。如果发现任何问题，则缓冲池会引发常见的错误 823 或 824，但是错误会由 *DBCC CHECKDB* 所抑制并转换成一条特殊的 DBCC 错误信息。这些通常是错误 8928 和 8939，如下所示：

```
Msg 8928, Level 16, State 1, Line 1
Object ID 1326627769, index ID 1, partition ID 72057594048872448, alloc unit ID
72057594055557120 (type LOB data): Page (1:69965) could not be processed. See other errors
for details.
Msg 8939, Level 16, State 98, Line 1
```

```
Table error: Object ID 1326627769, index ID 1, partition ID 72057594048872448, alloc unit ID
72057594055557120 (type LOB data), page (1:69965). Test (IS_OFF (BUF_IOERR, pBUF->bstat))
failed. Values are 12716041 and -4.
```

如果缓冲池统计失败，则不会对页做进一步处理。否则会执行 DBCC 页审核。这包括如下步骤。

（1）检查页眉中的页 ID 是正确的。这种检查实际上是由缓冲池读取页时执行的，并且在检查失败时会通知 *DBCC CHECKDB*。如果在原始系统目录检查期间审核主要系统目录中的一个页时这种检查失败，则会出现如下所示的错误 5256（不包含元数据信息）：

```
Msg 5256, Level 16, State 1, Line 1
Table error: alloc unit ID 334184954400421, page (1:2243) contains an incorrect page
ID in its page header. The PageId in the page header = (0:0).
```

如果这种检查在其他任何情况下失败，则会产生如下所示的错误 8909：

```
Msg 8909, Level 16, State 1, Line 1
Table error: Object ID 0, index ID -1, partition ID 0, alloc unit ID 844424953200640
(type Unknown), page ID (1:26483) contains an incorrect page ID in its page header.
The PageId in the page header = (0:0).
```

（2）检查页类型对页所在的分配单元是有效的。例如，数据页不应该在非聚集索引的分配单元中存在。如果这种检查失败，则会产生如下所示的错误 8938：

```
Msg 8938, Level 16, State 1, Line 1|
Table error: Page (1:4667), Object ID 1877736499, index ID 1, partition ID
72044394032172426, alloc unit ID 72044394045227020 (type LOB data). Unexpected page
type 1.
```

（3）检查页上的每条记录都有正确的结构并且没有任何不正确的指针（例如，指向另外一条记录或可用空间）。如果任何记录结构审核检查失败，则会产生如下所示的 8940 到 8944 的错误：

```
Msg 8941, Level 16, State 1, Line 1
Table error: Object ID 0, index ID -1, partition ID 0, alloc unit ID 72057613244301312
(type Unknown), page (3:45522). Test (sorted [i].offset >= PAGEHEADSIZE) failed. Slot
114, offset 0x12 is invalid.
Msg 8942, Level 16, State 1, Line 1
Table error: Object ID 0, index ID -1, partition ID 0, alloc unit ID 72057613244301312
(type Unknown), page (3:45522). Test (sorted[i].offset >= max) failed. Slot 0, offset
0x72 overlaps with the prior row.
Msg 8944, Level 16, State 12, Line 1
Table error: Object ID 0, index ID -1, partition ID 0, alloc unit ID 72057613244301312
(type Unknown), page (3:45523), row 0. Test (ColumnOffsets <= (nextRec - pRec))
failed. Values are 25 and 17.
```

作为页审核进程的一部分，页上的所有页压缩信息都会被验证，包括存储前缀（位于嵌入在 CI 记录中的一条锚记录中）和压缩字典（大量偏移正值）的每页压缩信息（CI）记录。该记录中的任何错误都会报告错误 5274。

一旦页通过审核，页眉中的分配单元 ID 就会用于查询前面介绍的元数据哈希表，以查找描述存储在页记录中所有内容的元数据。接下来检查该页以确定自上一次执行完全备份以来是否被修改过。如果被修改过，并且相关的差分位图已经被正确设置用来说明修改，则会报告如下所示的 2515 错误：

```
Msg 2515, Level 16, State 1, Line 1
```

```
The page (1:24), object ID 60, index ID 1, partition ID 281474980642816, allocation unit ID
281474980642816 (type In-row data) has been modified, but is not marked as modified in the
differential backup bitmap.
```

一旦这些通用的检查执行完成，页就会按照自己的类型被进一步处理。

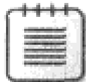
11.5.3 数据和索引页处理

数据页和索引页都是利用相同的高级算法处理的，该算法对每条记录执行如下操作。

- 对于具有行外 LOB 数据的记录来说，完全在内存中实例化记录（放入所有行溢出列）。对于没有行外 LOB 数据的简单记录来说，记录直接在包含它的页中进行处理。
- 检查记录长度是正确的，考虑添加到记录尾部的所有版本信息。
- 如果记录包含数据（即不是备份记录），则对记录中的所有列进行循环处理。
- 检查记录中没有反物质（antimatter）列，表示一个失败的联机索引操作。如果这种检查失败，则会输出如下所示的错误 5228 或错误 5229：

```
Msg 5228, Level 16, State 3, Line 1
Table error: object ID 2073058421, index ID 0, partition ID 72057594038321152, alloc
unit ID 72057594042318848 (type "In-row data"), page (3:23345), row 12. DBCC detected
incomplete cleanup from an online index build operation. (The anti-matter column value
is 14.)
```

- 检查每条记录的版本信息（如果存在）。如果在一条记录上附加版本信息，但是页眉没有说明页上有版本记录，则会输出错误 5260。如果记录有带有一个 NULL 版本时间戳但是没有非 NULL 版本链指针的版本信息，则会输出错误 5262。



注意：

版本存储本身的有效性不是由 DBCC CHECKDB 检查的。

- 根据记录及其内容生成所有必要事实（例如，B 树链接事实和 LOB 链接事实）。

对于没有存储在堆数据页上的记录来说，记录必须按照定义的聚簇或非聚集索引键排序。在对页进行一致性检查时，上一条记录的键值会被记录，从而使它们可以与当前被处理的记录进行比较。如果记录没有被正确地排序，则会产生如下所示的错误 2511：

```
Msg 2511, Level 16, State 1, Line 1
Table error: Object ID 142675606, index ID 1, partition ID 72057594295025664, alloc unit ID
72057594301906944 (type In-row data). Keys out of order on page (1:1124457), slots 59 and 60.
```

对于没有存储在堆数据页中的记录来说，页上的记录也必须具有唯一的键值——任何两条记录没有相同的键值。这甚至应用于已经定义为非唯一的索引——但是只在关系级。在存储引擎级别上，每条记录必须是唯一标识的。如果两条记录具有相同的键，则会报告如下所示的错误 2512：

```
Msg 2512, Level 16, State 2, Line 1
Table error: Object ID 4, index ID 1, partition ID 262144, alloc unit ID 262144 (type In-row
data). Duplicate keys on page (1:4224) slot 9 and page (1:4224) slot 10.
```

页和记录之间各种链接的一致性检查将在本章后面的“跨页一致性检查”一节介绍。

当所有记录被处理之后，会检查页眉中的如下计数器：

- 页上的记录数（槽数）；
- 页上备份记录的数量。

如果记录数不正确，则会报告错误 8919。如果备份记录数不正确，则会报告如下所示的错误 8927：

```
Msg 8927, Level 16, State 1, Line 1
Object ID 29, index ID 1, partition ID 281474978611200, alloc unit ID 281474978611200 (type
In-row data): The ghosted record count in the header (0) does not match the number of
ghosted records (1) found on page (1:309).
```

对于非叶 B 树页来说，页上必须至少有一条记录。如果没有，则会报告错误 2574。

对于堆中的数据页来说，会检查相关 PFS 页中相应字节的可用空间记录。如果两者不匹配，则会报告如下所示的错误 8914：

```
Msg 8914, Level 16, State 1, Line 3
Incorrect PFS free space information for page (1:2511951) in object ID 357576312, index
ID 0, partition ID 72057594040156160, alloc unit ID 72057594044284928 (type In-row data).
Expected value 100_PCT_FULL, actual value 95_PCT_FULL.
```

11.5.4 列处理

对于数据和索引记录来说，每列按照自己的类型进行处理。这里介绍的很多检查都会导致常见的错误信息 2537（“错误记录”），可用添加到错误中的某些特殊文本来标识具体的问题。

对于*复杂列*（即存储 LOB 或 *FILESTREAM* 数据或联接的列）来说，列结构被检查同时提取相关联接事实。如果发现一个错误复杂列，则会产生错误 8960。

被处理的列支持跨页一致性检查的方式也有很多。这些内容将在本章后面的“跨页一致性检查”一节介绍。

1. 计算列

如前面所述，一个表达式计算器会为包含计算列或 CLR UDT 的每个对象进行编译。如果表达式计算器不能被编译，则这些列不能进行一致性检查。

表达式计算器被调用以计算持久计算列，或索引记录中存在的计算列。它返回一个值，该值接下来与数据或索引记录中的持久值进行比较。如果两个值的 NULL 状态不同或者两个值的字节的比较不同，则会返回错误 2537。

对于 UDT 列来说，会在表达式计算器中进行比较。它被传递给被检查的整个记录并根据 UDT 比较返回 *True* 或 *False* 值。

注意表达式计算器对象不是线程安全的。也就是说，*DBCC CHECKDB* 在多个处理器上并行运行时（每个处理器内核上一个线程），只有一个处理器内核可以访问和使用表达式计算器。多个处理器内核能够处理同一表中具有计算列的页，因此所有内核需要访问表达式计算器。当然，有内部同步来防止这一现象，同时不可避免地，一个或多个处理器内核访问时可能必须等待。这是交互排他机制的情况，这可能会影响模式中具有大量计算列或 CLR UDT 的负载过重系统的性能。

2. NULL 和长度检查

执行以下 3 种检查。

- 值为 NULL 的变长列不能有一个非零数据长度。如果这种检查失败，则会报告错误 7961。
- 作为 NOT NULL 创建的列不能有 NULL 值。如果这种检查失败，则会报告错误 8970。
- 一个列不能比元数据中定义的行内长度最大值还长。如果这种检查失败，则会报告错误 2537。

3. 数据纯度检查

数据纯度检查用来检查列值是否在列的数据类型定义范围之内。具有一个“零时”1440 子值或更多子值的错误 *SMALLDATETIME* 列值就是一个示例。

正如 *SQL Server 联机丛书*中记录的那样，在 SQL Server 2005 以前的 SQL Server 版本中，可以向数据库中导入“越界”数据值。在 SQL Server 2005 和 SQL Server 2008 中，则不可能再导入越界值。SQL Server 2005 引入了纯数据库的概念——换言之，数据库中没有“越界”数据值。

纯数据库默认运行数据纯度检查，这种功能不能被禁用。在 SQL Server 2008 上创建的数据库在创建时就是纯的并且在升级到 SQL Server 2008 时仍然如此。

不纯的数据库默认不进行数据纯度检查——必须专门用 WITH DATA_PURITY 选项请求。不纯数据库是在 SQL Server 2005 之前创建的并且升级到 SQL Server 2008 的数据库，在没有错误时不运行数据纯度检查。在没有错误的情况下运行数据纯度检查时，数据库不可避免地会转变成纯的。数据库的纯度状态存储在启动页中。

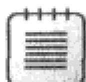
下面的表 11-2 列出了一些 SQL Server 数据类型及为这些类型执行的数据纯度验证。

表 11-2　数据类型进行的数据纯度检查

数据类型	数据纯度检查
TINYINT	None——所有值都是有效的
SMALLINT	None——所有值都是有效的
INT	None——所有值都是有效的
BIGINT	None——所有值都是有效的
MONEY	None——所有值都是有效的
SMALLMONEY	None——所有值都是有效的
UNIQUEIDENTIFIER	None——所有值都是有效的
TIMESTAMP	None——所有值都是有效的
IMAGE	None——所有值都是有效的
TEXT	根据排序规则验证 DBCS 字节
NTEXT	验证长度是 2 的倍数
BIT	保证值为 0 或 1
REAL 或 *FLOAT*	验证浮点值超出合法范围
DATETIME	验证 *DATETIME* 结构中的字段。例如，“days”字段必须小于（December 31, 9999）并且大于（January 1, 1753）
SMALLDATETIME	验证 *SMALLDATETIME* 结构中的字段。例如，“minutes“字段必须小于 1440（即 60×24）
DECIMAL 或 *NUMERIC*	验证值的精确度小于或等于定义的精确度，值的大小等于定义的大小，并且值是合法的
BINARY	验证值有正确的长度
VARBINARY	验证长度小于或等于定义的最大长度

续表

数据类型	数据纯度检查
VARBINARY（MAX）	None——所有值都是有效的
NCHAR	验证长度小于或等于定义的最大长度并且长度是 2 的倍数
NVARCHAR	验证长度小于或等于定义的最大长度并且长度是 2 的倍数
NVARCHAR（MAX）	验证长度是 2 的倍数
CHAR	验证长度等于定义的长度。根据排序规则验证 DBCS 字节
VARCHAR	验证长度小于或等于定义的最大长度。根据排序规则验证 DBCS 字节
VARCHAR（MAX）	根据排序规则验证 DBCS 字节
SQLVARIANT	验证 SQLVARIANT 结构是有效的并且其中包含的值对于它所属的数据类型来说是有效的
UDTs	将值转换成 UDT 并用原始值对结果进行字节比较
XML	执行 XML 值的结构验证。由 XML 子系统执行



注意：

压缩值（通过行压缩、页压缩或 *VARDECIMAL*）必须在检查之前被压缩。如果表、文件组或数据库的大部分被压缩，则会增加 CPU 系统开销及 *DBCC CHECKDB* 的额外运行时间。

如果列值在数据纯度检查时失败，则会返回如下所示的错误 2570：

```
Msg 2570, Level 16, State 3, Line 1
Page (1:152), slot 0 in object ID 2073058421, index ID 0, partition ID 72057594038321152,
alloc unit ID 72057594042318848 (type "In-row data"). Column "c1" value is out of range for
data type "datetime". Update column to a legal value.
```

这些错误不能被修复并且必须手动处理。处理这一问题的方法在知识库文章 923247（http://support.microsoft.com/kb/923247）进行了介绍。

4. 分区检查

正如前面介绍的那样，如果被检查的表或索引被分区，则它的索引元数据缓存对象包含分区函数使用的所有信息。

所有列值检查完成后，页上的每条记录都会被测试以保证处于正确的分区中。用于分区的列是从每条记录中提取的并且传递到查询处理器的帮助函数中。帮助程序对分区函数进行求值并返回该记录应该属于的分区 ID。如果返回的分区 ID 与页所属的分区 ID 不匹配，则会产生如下所示的错误 8984 和 8988。

```
Msg 8984, Level 16, State 1, Line 1
Table error: Object ID 2073058421, index ID 0, partition ID 72057594038452224. A row should
be on partition number 2 but was found in partition number 3. Possible extra or invalid keys
for:
Msg 8988, Level 16, State 1, Line 1
Row (1:162:0) identified by (HEAP RID = (1:162:0)).
```

错误 8984 确定包含错误的分区，错误 8988 确定错误分区记录的物理位置，以及可以用于访问记录的索引键（或堆物理 RID——如果不正确的分区记录是分区堆的一部分）。

5. 稀疏列检查

将列定义为 SPARSE 的功能是 SQL Server 2008 中的一个新功能。值为 NULL 的 SPARSE 列根本没有存储在记录中，甚至没有在 NULL 位图中。也就是说 NULL 值实际上在记录中没有占用空间。当一个 SPARSE 列为非 NULL 时，它被存储在一个特殊的 SPARSE 列阵列中，SPARSE 列阵列作为变长列存储在记录的变长列阵列中。SPARSE 列阵列的一致性检查是由查询处理器执行的，并且错误以常见列错误形式报告。

11.5.5 文本页处理

文本页用于存储 LOB 值（或者是行外存储的实际 LOB 值，或者是已经作为行溢出数据行外存储的非 LOB 变长列）。在包含文本记录或 LOB 连接的所有错误信息中，分配单元类型可以是 *LOB 数据*或*行溢出数据*。

有多种类型的文本记录，以各种方式用于构建存储 LOB 值的松散文本树。文本记录存储在两种类型的文本页上——或者是专用于单个 LOB 值，或者是多个 LOB 值共享的页。这两种文本页都是利用相同的算法处理的，该算法对每条文本记录执行如下操作。

- 实例化记录并检查它是一个有效的文本记录。
- 检查每条记录的版本信息（如果存在）。如果记录有版本信息但是页眉没有指出该页上有版本记录，则会产生错误 5260。如果记录有一个带有 NULL 的版本时间戳但是一个非 NULL 版本链指针的版本信息，则会产生错误 5262。
- 从记录中生成所有必要的事实及其内容（即 LOB 链接事实）。

当检查到一条文本记录是有效的，则具有各种结构的多种类型的文本记录一定是其中的一部分。除标准的记录格式结构检查之外，执行的特殊文本检查如下。

- 具有版本信息的已删除文本记录一定具有合适的行大小。如果这种检查失败，则会报告错误 2537。
- 文本记录至少一定是存储一个文本树叶级结点所需的最小大小。如果这种检查失败，则会报告错误 2537。
- 文本记录一定是适当的文本页类型。如果这种检查失败，则会报告如下所示的错误 8963：

```
Msg 8963, Level 16, State 1, Line 1
Table error: Object ID 1326627769, index ID 1, partition ID 72057594048872448, alloc
unit ID 72057594022622331 (type LOB data). The off-row data node at page (3:23345),
slot 12, text ID 89622642688 has type 3. It cannot be placed on a page of type 4.
```

- 非叶文本记录具有的子节点数不能超过文本记录类型中能够存储的数量，不能具有比子联接阵列大小更多的子节点，也不能有比最大允许的文本树扇出更多的子节点。如果这些检查失败，则会报告错误 2537。
- 文本记录必须有一种有效的类型。如果这种检查失败，则会报告错误 8962。

文本记录中的错误通常伴随一个错误 8929，表示联接到错误文本记录的数据或索引记录，如下所示：

```
Msg 8929, Level 16, State 1, Line 1
Object ID 1326627769, index ID 1, partition ID 72057594048872448, alloc unit ID
72057594055622656 (type In-row data): Errors found in off-row data with ID 89622642688 owned
by data record identified by RID = (1:77754:1)
```

下一部分将介绍页和记录之间各种联接的一致性检查。

一旦所有记录被处理完，则会检查页眉中的各种计数器：

- 页上的记录数（槽数）；
- 页上的备份记录的数量。

如果记录数不正确，则会报告错误 8919。如果备份记录数是不正确的，则会报告错误 8927。

可用空间数对相关 PFS 页中相应字节进行检查。如果两者不匹配，则会报告如下所示的错误 8914：

```
Msg 8914, Level 16, State 1, Line 1
Incorrect PFS free space information for page (1:35244) in object ID 1683128146, index ID 1,
partition ID 223091033422352, alloc unit ID 81405523118118176 (type LOB data). Exected value
0_PCT_FULL, actual value 100_PCT_FULL
```

11.5.6 跨页一致性检查

在各种数据、索引和文本页被处理时，会从页上的记录中提取事实来支持跨页一致性检查。根据数据库中现有模式执行的各种检查如下。

- 堆数据页中转发和已转发记录之间的联接。
- 内部 B 树页和记录联接。
- 数据/索引记录和文本记录之间的联接。
- 内部文本树记录联接。
- 数据/索引记录和 *FILESTREAM* 文件之间的联接。
- *FILESTREAM* 容器结构。
- 基表记录和非聚集索引记录之间的联接。

接下来将介绍这些检查。

1. 堆一致性检查

对堆的跨页一致性检查会验证转发和已转发记录之间的联接。当堆中数据记录的大小增长并且记录的当前页没有空间容纳增长的大小时会出现转发/已转发记录。记录被移动到新位置（成为一条转发记录）并且一条小型存根记录（转发记录）保留在原始位置以指示记录的实际位置。

当转发记录的位置不需要再次修改时，转发记录会指回转发记录——而不是创建的转发记录链。原始转发记录会被更新以指向新位置。

在对堆数据页的标准处理过程中，会从转发和转发记录生成额外事实。

- 转发记录生成父记录。
- 被转发记录生成一条实际事实，有一条由联接回转发记录的联接组成的备注。

当事实被聚合时，会进行如下检查。

- 由转发记录联接的转发记录必须存在。如果这种检查失败，则会报告如下所示的错误 8993：

```
Msg 8993, Level 16, State 1, Line 3
Object ID 357576312, forwarding row page (1:2386712), slot 8 points to page
(1:2621015), slot 18. Did not encounter forwarded row. Possible allocation error.
```

- 由已转发记录联接回的转发记录必须存在。如果这种检查失败，则会报告如下所示的错误 8994：

```
Msg 8994, Level 16, State 1, Line 1
Object ID 1967346073, forwarded row page (1:181506), slot 23 should be pointed to by
```

```
forwarding row page (1:83535), slot 66. Did not encounter forwarding row. Possible
allocation error.
```

■ 转发记录联接的已转发记录必须联接回该转发记录。如果这种检查失败，则会报告错误 8971：

```
Msg 8971, Level 16, State 1, Line 3
Forwarded row mismatch: Object ID 357576312, partition ID 72057594040156160, alloc
unit ID 72057594044284928 (type In-row data) page (1:3491303), slot 18 points to
forwarded row page (1:2506991), slot 22; the forwarded row points back to page
(1:3423966), slot 1
```

■ 已转发记录不能由多条转发记录联接。如果这种检查失败，则会报告如下所示的错误 8972：

```
Msg 8972, Level 16, State 1, Line 3
Forwarded row referenced by more than one row. Object ID 357576312, partition
ID 72057594040156160, alloc unit ID 72057594044284928 (type In-row data), page
(1:2500650), slot 2 incorrectly points to the forwarded row page (1:4361594), slot 4,
which correctly refers back to page (1:3472293), slot 20.
```

2. B 树一致性检查

对 B 树进行跨页一致性检查可以验证 B 树级别内、B 树级别之间的联接，同时键的一致性在级别之间进行变化。

对于索引的叶级页来说，页联接事实是由页眉产生的，附加页的第一条记录和最后一条记录的事实（提供页上包含的键范围）。对于索引的非叶级页来说，所有事实都被生成（以及来自页上每条记录的一个事实），包含指向记录引用的 B 树中下一级别页的指针。

在事实被聚合时，会进行如下检查。

■ 页眉中下一页联接所指向的页必须具有相同的 B 树级别。如果这种检查失败，则会报告错误 2531。

■ 非叶（父）页的子页联接只能联接到 B 树中自己下面一个级别的页。如果这种检查失败，则会报告错误 8931。

■ 上一个页的联接必须与父页中子页联接的顺序一致。如果父页中 A 页的后面是 B 页，则 B 页的页眉中上一页的联接一定到 A 页。如果这种检查失败，则会报告如下所示的错误 8935：

```
Msg 8935, Level 16, State 1, Line 3
Table error: Object ID 1349579846, index ID 1, partition ID 72057594040811520, alloc
unit ID 72057594046382080 (type In-row data). The previous link (1:233719) on page
(1:233832) does not match the previous page (1:275049) that the parent (1:42062), slot
16 expects for this page.
```

■ 如果 A 页页眉中的下一页联接到 B 页，则 B 页的页眉中上一页联接一定回到 A 页。如果这种检查失败，则会报告如下所示的错误 8936：

```
Msg 8936, Level 16, State 1, Line 3
Table error: Object ID 1349579846, index ID 1, partition ID 72057594040811520,
alloc unit ID 72057594046382080 (type In-row data). B-tree chain linkage mismatch.
(1:275049)->next = (1:233832), but (1:233832)->Prev = (1:233719).
```

■ 一个页只能通过 B 树中更高级的某个非叶页联接（也就是说，两个“父”页不能有两个子页联接）。如果这种检查失败，则会报告如下所示的错误 8937：

```
Msg 8937, Level 16, State 1, Line 3
```

```
Table error: Object ID 1349579846, index ID 1, partition ID 72057594040811520, alloc
unit ID 72057594046382080 (type In-row data). B-tree page (1:148135) has two parent
nodes (1:212962), slot 20 and (1:233839), slot 1.
```

- 页只应该被 *DBCC CHECKDB* 扫描一次。如果这种检查失败，则会报告错误 8973。
- 如果“父”页上的某个子页联接向下联接到某个页上，并且同级中的某个页有一个上一页联接也联到该页，则该页一定发生冲突。如果这种检查失败，则会报告如下所示的错误 8976：

```
Msg 8976, Level 16, State 1, Line 1
Table error: Object ID 2073058421, index ID 1, partition ID 72057594038386688, alloc
unit ID 72057594042384384 (type In-row data). Page (1:158) was not seen in the scan
although its parent (1:154) and previous (1:157) refer to it. Check any previous
errors.
```

- B 树中的每个页必须有一个带有指向它子页联接的“父页”。如果这种检查失败，则会报告如下所示的错误 8977：

```
Msg 8977, Level 16, State 1, Line 3
Table error: Object ID 1349579846, index ID 1, partition ID 72057594040811520, alloc
unit ID 72057594046382080 (type In-row data). Parent node for page (1:163989) was not
encountered.
```

- B 树中的每个页一定有一个带有指向它下一页联接的上一页。这包括 B 树左端的页，在这里创建假联接事实以使聚合工作。如果该检查失败，则会报告如下所示的错误 8978：

```
Msg 8978, Level 16, State 1, Line 3
Table error: Object ID 1349579846, index ID 1, partition ID 72057594040811520, alloc
unit ID 72057594046382080 (type In-row data). Page (1:238482) is missing a reference
from previous page (1:233835). Possible chain linkage problem.
```

- 每个 B 树页一定有联接到它的子页联接的“父”页，以及联接到它下一页联接的上一页。其中包括 B 树左端的那些页。如果情况并非如此，则通常是 B 树的根页出现了问题，这是由一个错误的系统目录项引起的。这是产生错误的原因，实际上是上面两个问题的共同作用。如果这种检查失败，则会报告如下所示的错误 8979：

```
Msg 8979, Level 16, State 1, Line 1
Table error: Object ID 768057822, index ID 8. Page (1:92278) is missing references
from parent (unknown) and previous (page (3:10168)) nodes. Possible bad root entry in
sysindexes.
```

- 在 DBCC CHECKDB 对 B 树中的某个有效页进行扫描时，一定会遇到由“父”页中某个子页联接所联接的页。如果这种检查失败，则会报告如下所示的错误 8980：

```
Msg 8980, Level 16, State 1, Line 1
Table error: Object ID 421576540, index ID 8. Index node page (1:90702), slot 17
refers to child page (3:10183) and previous child (3:10182), but they were not
encountered.
```

- 在 *DBCC CHECKDB* 对 B 树中的某个有效页进行扫描时，一定会遇到由页眉中下一页联接所联接的页。如果这种检查失败，则会报告如下所示的错误 8981：

```
Msg 8981, Level 16, State 1, Line 3
```

```
Table error: Object ID 1349579846, index ID 1, partition ID 72057594040811520, alloc
unit ID 72057594046382080 (type In-row data). The next pointer of (1:233838) refers
to page (1:233904). Neither (1:233904) nor its parent were encountered. Possible bad
chain linkage.
```

- 页眉中的下一页联接一定联接到同一棵 B 树中的某个页。如果这种检查失败，则会报告错误 8982。
- 一个页一定联接到同一棵 B 树中的“父”页和上一页。如果这种检查失败，则会报告如下所示的错误 8926：

```
Msg 8926, Level 16, State 3, Line 1
Table error: Cross object linkage: Parent page (0:1), slot 0 in object 2146106686,
index 1, partition 72057594048806912, AU 72057594053394432 (In-row data), and page
(1:16418)->next in object 366624349, index 1, partition 72057594049593344, AU
72057594054246400 (In-row data), refer to page [1:16768] but are not in the same
object.
```

- 页上的最低键值一定大于或等于 B 树中上一级中“父”页的子页联接中的键值。如果这种检查失败，则会报告如下所示的错误 8933：

```
Msg 8933, Level 16, State 1, Line 3
Table error: Object ID 1349579846, index ID 1, partition ID 72057594040811520, alloc
unit ID 72057594046382080 (type In-row data). The low key value on page (1:148134)
(level 0) is not >= the key value in the parent (1:233839) slot 0.
```

- 页上的最高键值一定小于 B 树同级中下一页“父”页的子页联接中的键值。如果这种检查失败，则会报告如下所示的错误 8934：

```
Msg 8934, Level 16, State 3, Line 3
Table error: Object ID 1349579846, index ID 1, partition ID 72057594040811520, alloc
unit ID 72057594046382080 (type In-row data). The high key value on page (1:275049)
(level 0) is not less than the low key value in the parent (0:1), slot 0 of the next
page (1:233832).
```

如果一棵 B 树错误，则其中的很多错误很容易在同一棵 B 树中一起出现。同时，没有遇到期待页时出现的很多错误，同时伴随错误 2533：

```
Msg 2533, Level 16, State 1, Line 1
Table error: Page (3:9947) allocated to object ID 768057822, index ID 4 was not seen. Page
may be invalid or have incorrect object ID information in its header.
Msg 8976, Level 16, State 1, Line 1
Table error: Object ID 768057822, index ID 4. Page (3:9947) was not seen in the scan
although its parent (1:858889) and previous (1:84220) refer to it. Check any previous
errors.
```

如果确定丢失的页被分配给另一个对象，则会报告错误 2534，用于指示页眉中的实际对象。

3. LOB 联接一致性检查

正如前面介绍的那样，LOB 联接事实是由数据或索引目录中的文本记录和复杂列生成的。这允许到行外 LOB 列或行溢出数据值的文本树和联接一致性检查。

在聚合时会执行如下检查。

- 文本记录中的文本时间戳一定与联接到它的数据或索引记录复杂列中的文本时间戳相匹配。如果这种检查失败，则会报告如下所示的 8961 错误：

```
Msg 8961, Level 16, State 1, Line 1
Table error: Object ID 434100587, index ID 1, partition ID 72057594146521088, alloc
unit ID 71804568277286912 (type LOB data). The off-row data node at page (1:2487),
slot 0, text ID 3788411843723132928 does not match its reference from page (1:34174),
slot 0.
```

- 每条文本记录一定有一个来自另一条文本记录或来自数据或索引记录着的一个复杂列的联接。如果这种检查失败，则会报告如下所示的错误 8964：

```
Msg 8964, Level 16, State 1, Line 1
Table error: Object ID 750625717, index ID 0, partition ID 49193006989312, alloc unit
ID 71825312068206592 (type LOB data). The off-row data node at page (1:343), slot 0,
text ID 53411840 is not referenced.
```

同一个文本页上的文本记录可能报告多条 8964 错误。如果整个数据或索引页不能被处理，就会发生这种情况，同时错误页上的每条记录有一个到同一个文本页上某条文本记录的复杂列联接。

- 如果某条数据或索引记录中的一个复杂列联接到一条文本记录，则 *DBCC CHECKDB* 一定会扫描到该文本记录。如果这种检查失败，则会报告错误 8965。这种情况通常在某个文本页由于某种原因不能处理时发生，此时会报告如下所示的错误 8928：

```
Msg 8928, Level 16, State 1, Line 1
Object ID 1993058136, index ID 1, partition ID 412092034711552, alloc unit ID
71906736119218176 (type LOB data): Page (1:24301) could not be processed. See other
errors for details.
Msg 8965, Level 16, State 1, Line 1
Table error: Object ID 1993058136, index ID 1, partition ID 412092034711552, alloc
unit ID 71906736119218176 (type LOB data). The off-row data node at page (1:24301),
slot 0, text ID 1606680576 is referenced by page (1:24298), slot 0, but was not seen
in the scan.
```

- 一条文本记录只能有一个联接指向它。如果这种检查失败，则会报告如下所示的错误 8974：

```
Msg 8974, Level 16, State 1, Line 1
Table error: Object ID 373576369, index ID 1, partition ID
72057594039238656, alloc unit ID 71800601762136064 (type LOB data). The
off-row data node at page (1:13577), slot 13, text ID 31002918912 is
pointed to by page (1:56), slot 3 and by page (1:11416), slot 37.
```

- 一条数据或索引记录中一个复杂列的 LOB 联接一定联接到包含在同一个对象和索引中的某条文本记录。如果这种检查失败，则会报告错误 8925。

这些错误通常一起出现，同时伴随错误 8929，如下所示：

```
Msg 8961, Level 16, State 1, Line 1
Table error: Object ID 434100587, index ID 1, partition ID 72057594146521088, alloc unit ID
71804568277286912 (type LOB data). The off-row data node at page (1:2487), slot 2, text ID
341442560 does not match its reference from page (1:2487), slot 0.
Msg 8929, Level 16, State 1, Line 1
Object ID 434100587, index ID 1, partition ID 72057594146521088, alloc unit ID
72057594151239680 (type In-row data): Errors found in off-row data with ID 341442560 owned
by data record identified by RID = (1:34174:0)
```

8929 错误包含联接到错误文本记录的实际数据或索引记录。这些信息只能通过重新扫描所有数据和索引记录（在批处理结束时查找包含一个文本时间戳的复杂列与已经发现是错误的文本记录中的复杂列相匹配）找到。执行这种重新扫描的过程非常重要，以便数据库修复能够删除错误文本记录和包含指向它的联接的记录。

4. FILESTREAM 一致性检查

SQL Server 2008 中的一个主要新功能是 *FILESTREAM* 存储——在数据库之外的 NTFS 文件系统中存储 LOB 值的能力。这样可以在保持数据库中存储的关系数据事务完整性的同时，非常快速地访问 LOB 值。

与多位置存储系统一样，联接完整性很重要，以便 SQL Server 2008 中的 *DBCC CHECKDB* 对附加到数据库上的 *FILESTREAM* 存储进行严格的一致性检查。*FILESTREAM* 存储结构如下。

- *FILESTREAM* 数据容器的顶级是 NTFS 目录。
- 包含 *FILESTREAM* 数据的每个行集(即表或索引的一个分区)在顶级中都有一个目录(称为*行集目录*)。
- 分区中的每一列在行集目录中都有一个目录（称为*列目录*)。
- 分区中每条记录中该列的 *FILESTREAM* 数据值存储在列目录中。
- 也有一个 *FILESTREAM* 日志目录（认为是 *FILESTREAM* 存储的一个事务日志）存储在 *FILESTREAM* 数据容器的顶级中。

如果 *FILESTREAM* 日志目录已经受到损害，则 *FILESTREAM* 扫描会失败并根据损坏情况使 *DBCC CHECKDB* 以各种方式终止。例如，如果在 *FILESTREAM* 日志目录中创建了一个文件，则 *DBCC CHECKDB* 会失败，具体如下：

```
Msg 8921, Level 16, State 1, Line 1
Check terminated. A failure was detected while collecting facts. Possibly tempdb out of
space or a system table is inconsistent. Check previous errors.
Msg 5511, Level 23, State 10, Line 1
FILESTREAM's file system log record 'badlog.txt' under log folder '\\?\F:\Production\
FileStreamStorage\Documents\$FSLOG' is corrupted.
```

当 *DBCC CHECKDB* 开始时，会避免 *FILESTREAM* 垃圾收集器（GC）干扰 *DBCC CHECKDB* 扫描。*FILESTREAM* 联接事实是根据如下内容生成的。

- 包含 FILESTREAM 列的表和索引记录。
- 内部 GC 表，其中包含关于哪些 FILESTREAM 文件已经被逻辑删除（因此没有来自一条数据/索引记录的联接）但是还没有被物理删除的信息。
- *FILESTREAM* 容器中的每行集和每列目录。
- 实际 *FILESTREAM* 数据文件。

在事实聚合时，会执行如下一致性检查。

- 一个 *FILESTREAM* 文件一定有一个父联接（来自数据/索引记录或 GC 表)。如果这种检查失败，则会报告错误 7903。
- 来自数据/索引记录或 GC 表的 *FILESTREAM* 联接一定指向一个有效的 *FILESTREAM* 文件。如果这种检查失败，则会报告错误 7904。如果已经对一个 *FILESTREAM* 数据文件进行了手动修改，则一起报告这两种错误，如下所示：

```
Msg 7903, Level 16, State 2, Line 1
```

```
Table error: The orphaned file "00000017-00000101-0003" was found in the FILESTREAM
directory ID 988dc26c-ab62-46a4-bc44-c1062b6f6f80 for object ID 2105058535, index ID
0, partition ID 72057594038779904, column ID 3.
Msg 7904, Level 16, State 2, Line 1
Table error: Cannot find the FILESTREAM file "00000017-00000101-0002" for column ID 3
(column directory ID 988dc26c-ab62-46a4-bc44-c1062b6f6f80) in object ID 2105058535,
index ID 0, partition ID 72057594038779904, page ID (1:169), slot ID 1.
```

- *FILESTREAM* 数据容器目录结构中的每个目录一定是 *FILESTREAM* 存储结构的一部分。如果错误目录位于 *FILESTREAM* 数据容器的顶级，则会报告错误 7905，否则报告错误 7909，如下所示：

```
Msg 7907, Level 16, State 1, Line 1
Table error: The directory "\988dc26c-ab62-46a4-bc44-c1062b6f6f80\BadDirectory" under
the rowset directory ID 7e23f5a2-9cc0-462f-82d3-03ff3eaec4c9 is not a valid FILESTREAM
directory.
```

- *FILESTREAM* 数据容器目录结构中的每个文件一定是一个有效的 *FILESTREAM* 数据文件。如果错误文件位于 *FILESTREAM* 数据容器的顶级，则会报告错误 7906，否则报告错误 7908，如下所示：

```
Msg 7908, Level 16, State 1, Line 1
Table error: The file "\988dc26c-ab62-46a4-bc44-c1062b6f6f80\corruptfile.txt" in the
rowset directory ID 7e23f5a2-9cc0-462f-82d3-03ff3eaec4c9 is not a valid FILESTREAM
file.
```

- 每个 *FILESTREAM* 行集或列目录都不应该由 *DBCC CHECKDB* 扫描一次以上。如果这种检查失败，则会报告错误 7931。
- 每个 *FILESTREAM* 行集目录应该位于某个数据库正确的 *FILESTREAM* 容器中。如果这种检查失败，则会报告错误 7932。
- 每个 *FILESTREAM* 行集目录一定映射到数据库中的某个有效分区。如果这种检查失败，则会报告错误 7933。
- 数据库中包含 *FILESTREAM* 数据的某个表或索引的每个分区都必须有一个匹配的 *FILESTREAM* 行集目录。如果这种检查失败，则会报告错误 7934。如果对某个 *FILESTREAM* 数据文件进行了手动修改，则可能同时报告这两种错误（极有可能伴随错误 7937 一起出现，如下所示）：

```
Msg 7933, Level 16, State 1, Line 1
Table error: A FILESTREAM directory ID 6e23f5a2-9cc0-462f-82d3-03ff3eaec4c9 exists for
a partition, but the corresponding partition does not exist in the database.
Msg 7937, Level 16, State 1, Line 1
Table error: The FILESTREAM directory ID 988dc26c-ab62-46a4-bc44-c1062b6f6f80 for
column ID of object ID 2105058535, index ID 0, partition ID 72057594038779904 was not
found.
Msg 7934, Level 16, State 1, Line 1
Table error: The FILESTREAM directory ID 7e23f5a2-9cc0-462f-82d3-03ff3eaec4c9 for
object ID 2105058535, index ID 0, partition ID 72057594038779904 was not found.
```

- 每个 *FILESTREAM* 列目录一定与某个分区中的某一列相匹配。如果这种检查失败，则会报告错误 7935。
- 每个 *FILESTREAM* 列目录一定与分区中的一个 *FILESTREAM* 列相匹配。如果这种检查失败，则会报告错误 7936。

- 一个分区中的每个 *FILESTREAM* 列在合适的 *FILESTREAM* 行集目录中一定有一个匹配 *FILESTREAM* 列的目录。如果父行集目录在某一方面有误，则行集目录中的所有列目录在进行这种检查时会失败（如前所述），同时返回错误 7937。
- 每个 *FILESTREAM* 数据文件在一个 *FILESTREAM* 列目录中仅应该由 *DBCC CHECKDB* 扫描一次。如果这种检查失败，则会显示文件系统错误并报告错误 7938。
- 每个 *FILESTREAM* 数据文件只能由表或索引分区中的一条记录联接。如果这种检查失败，则会报告错误 7941。
- *FILESTREAM* 日志文件不应该有误。如果这种检查失败，则会报告错误 7963。

5. 非聚集索引交叉检查

最后要介绍的跨页一致性检查是有关非聚集索引的。这一直是 *DBCC CHECKDB* 代码基准中我最喜欢的一部分，因为有效执行检查的复杂性。

非聚集索引交叉检查验证：

- 非聚集索引（筛选或非筛选）中每条记录一定映射到基表（堆或聚集索引）中的一个有效记录上；
- 基表中的每条记录一定精确映射到每个非筛选、非聚集索引中的一条记录和每个筛选索引中的一条记录上（此时允许使用筛选器）。

如果一条非聚集索引记录丢失，则会报告错误 8951 和 8955。错误 8951 报告丢失记录的索引名和表名。错误 8955 确定丢失一条索引记录的数据记录及丢失索引记录的索引键。

如果有某个额外的非聚集索引记录，则会报告错误 8952 和 8956。错误 8952 报告具有额外记录的索引的索引名和表名。错误 8956 确定额外索引记录和数据记录（索引记录所联接的记录）的索引键。

通常来说，一个非聚集索引记录是错误的，因此会报告如下所示的 4 条记录：

```
Msg 8951, Level 16, State 1, Line 1
Table error: table 'FileStreamTest1' (ID 2105058535). Data row does not have a matching
index row in the index 'UQ__FileStre__3EF188AC7F60ED59' (ID 2). Possible missing or invalid
keys for the index row matching:
Msg 8955, Level 16, State 1, Line 1
Data row (1:169:0) identified by (HEAP RID = (1:169:0)) with index values 'DocId =
'7E8193B4-9C86-47C0-2207-BF1293BA8292' and HEAP RID = (1:169:0)'.
Msg 8952, Level 16, State 1, Line 1
Table error: table 'FileStreamTest1' (ID 2105058535). Index row in index 'UQ__
FileStre__3EF188AC7F60ED59' (ID 2) does not match any data row. Possible extra or invalid
keys for:
Msg 8956, Level 16, State 1, Line 1
Index row (1:171:1) with values (DocId = '7E8193B4-9C86-47C0-B407-BF2293BA8292' and HEAP RID
= (1:169:0)) pointing to the data row identified by (HEAP RID = (1:169:0)).
```

有效地执行这些检查的机制自 SQL Server 7.0 以来的每个版本中都有所改变——变得越来越有效。在 SQL Server 2008 中，为每个非聚集索引的每个分区创建了两个哈希表——一个哈希表用于存储非聚集索引分区中的实际记录，另一个哈希表用于存储应该存储在非聚集索引的分区中的记录（正如根据表中现有数据记录计算的那样）。

当一条非聚集索引记录被处理时，记录中的所有列都会被散列到一个 *BIGINT* 值中。这包括：

- 到基表的物理或逻辑联接（被称为基表 RID）；
- 所有包含性列（甚至是 LOB 和 *FILESTREAM* 值）都被散列到一个 *BIGINT* 值中。

结果值被添加到非聚集索引分区（记录所在）的实际记录的主哈希值上。

DBCC CHECKDB 知道表存在哪些非聚集索引及每个索引的完整非聚集索引记录组成应该是什么。当一条数据记录被处理时，会对每一条匹配的非聚集索引记录（为数据记录而存在的（考虑筛选非聚集索引的筛选谓词）运行如下算法。

（1）在内存中创建非聚集索引记录（同样包括基表 RID 及包含的列）。

（2）将索引记录中的所有列都散列到一个 *BIGINT* 值中。

（3）将结果值添加到对于索引记录所在的相关非聚集索引分区来说“应该存在”的主哈希值中。

该算法工作的前提是不存在错误，这样实际记录的主哈希值和每个非聚集索引分区的“应该存在”记录在 *DBCC CHECKDB* 批处理结束时应该精确匹配。

如果不匹配，则表示有问题。这里介绍的算法也不是百分之百有效。如果两个主哈希值不匹配（自 SQL Server 2000 以来总会出现这种情况），则没有办法精确说明非聚集索引分区中的哪条记录有误。此时，一定会执行一次深层次检查，对表和它的索引进行比较以精确查找到有错误的记录。

深层次检查一旦被触发则会花费很长时间来运行，这样可能明显增加 *DBCC CHECKDB* 的运行时间。如果触发一次深层次检查，则会向 SQL Server 错误日志中输出错误 5268，同时向被搜索的每个表输出一个错误 5275。下面是一个示例：

```
2008-11-25 15:57:53.95 spid55     DBCC CHECKDB is performing an exhaustive search of 1
indexes for possible inconsistencies. This is an informational message only. No user action
is required.
2008-11-25 15:57:53.96 spid55     Exhaustive search of 'dbo.FileStreamTest1, UQ__
FileStre__3EF188AC7F60ED59' (database ID 17) for inconsistencies completed. Processed 1 of
1 total searches. Elapsed time: 5 milliseconds. This is an informational message only.
No user action is required.
```

深层次检查利用查询处理器在表和关注的索引之间进行匹配，基本上仅利用内部语法执行两个左反半联接（left-anti-semi-join）。包含的查询采用如下形式：

```
SELECT <all information needed for errors 8951 and 8955 for an unmatched data record>
FROM <tablename> tOuter WITH (INDEX = <base table>)
WHERE NOT EXISTS
(
       SELECT 1
       FROM   <tablename> tInner WITH (INDEX = <nonclustered index>)
       WHERE
       (
              (([tInner].<index columns> = [tOuter].<index columns>)
              OR ([tInner].<index columns> IS NULL AND [tOuter].<index columns> IS NULL))
       AND
              (([tInner].<base table RID> = [tOuter].<base table RID>)
              OR ([tInner].<base table RID> IS NULL AND [tOuter].<base table RID> IS NULL))
       )
)
UNION ALL
SELECT <all information needed for errors 8952 and 8956 for an unmatched index record>
FROM <tablename> tOuter WITH (INDEX = <nonclustered index>)
WHERE NOT EXISTS
(
       SELECT 1
       FROM   <tablename> tInner WITH (INDEX = <base table>)
```

```
        WHERE
        (
                (([tInner].<index columns> = [tOuter].<index columns>)
                OR ([tInner].<index columns> IS NULL AND [tOuter].<index columns> IS NULL))
        AND
                (([tInner].<base table RID> = [tOuter].<base table RID>)
                OR ([tInner].<base table RID> IS NULL AND [tOuter].<base table RID> IS NULL))
        )
)
```

对两个哈希值不匹配的每个非聚集索引分区执行完查询后，批处理才真正完成。

11.6 跨表一致性检查

跨表一致性检查包括验证数据库中各种表之间的非物理关系。一些示例如下。

- 表的元数据一定具有描述列的匹配元数据（这两种数据被存储在不同的系统目录中）。
- 一个主要的 XML 索引一定是它索引的 XML 列的一种精确表示法（一个主要的 XML 索引作为一个内部表存储，与包含它索引的 XML 列的表分离）。
- 一个索引视图一定是视图定义的一种精确表示法（一个索引视图作为一个内部表存储，与视图定义中引用的表分离）。

这些跨表一致性检查不能运行，除非包含的表已经被检查并且没有一致性问题（或者已经修复了一致性问题）。

例如，想象一下某个 XML 索引是基于表 *T1* 中的某个 XML 列。表 *T1* 有一个页以这种方式被损坏，它好像是空的（即某些记录不可访问）。如果 XML 索引在表 *T1* 之前被检查，则好像 XML 索引中有额外信息并且是有错误的。但是，实际上 *T1* 表是有错误的——XML 索引需要在对表 *T1* 进行修复之后才能重建。

这好像是一个微不足道的差异，但是 *DBCC CHECKDB* 需要报告这个第一位一致性错误。执行其他跨表一致性检查时使用相同的逻辑，因此根据前面步骤中发现的一致性错误可以跳过某些跨表一致性检查。

11.6.1 Service Broker 一致性检查

Service Broker 功能使用数据库中两种类型的表。

- 数据库中存储 Service Broker 的元数据的系统目录（例如，会话、终点和队）。
- 用于存储 Service Broker 队列的内部表。

两种类型的表与用户表具有相同的物理结构，但是用户通过不同的属性管理它们的行为和可访问性。它们的物理结构作为前面介绍的逻辑一致性检查的一部分被检查。

同时对 Service Broker 系统目录和队列包含的数据执行另一个级别的检查，与本章前面介绍的系统目录一致性检查类似。这些检查验证如下内容。

- 一个会话必须具有两个终点。
- 一个服务必须与一个有效合约相关。
- 一个服务必须与一个有效队列相关。
- 一条信息必须有一个有效的消息类型。

这些检查不能由 *DBCC CHECKDB* 执行——相反是由代表 *DBCC CHECKDB* 的 Service Broker 子系统本身执行的。如果发现任何一致性错误，则会以一个错误 8997 向 *DBCC CHECKDB* 报告最终用户结果

中的内容，格式与 8992 跨目录一致性检查错误一样。

11.6.2 跨目录一致性检查

在 SQL Server 2005 以前的 SQL Server 版本中，对于何时运行 *DBCC CHECKCATALOG* 来验证各种系统目录之间的关系一直不确定。为了消除这种不确定性，SQL Server 2005 以后的 *DBCC CHECKDB* 内部包含了 *DBCC CHECKCATALOG* 功能。

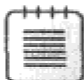
SQL Server 2005 重写了关系引擎中的整个元数据子系统，除 SQL Server 2008 中的一些列外，还写入了一组新的目录一致性检查。这些检查比 SQL Server 2000 及更早版本中相应的检查更广泛、更有效，并由代表 *DBCC CHECKDB* 的元数据子系统执行（也可以利用 *DBCC CHECKCATALOG* 命令执行检查）。

这些检查只在处理关系引擎元数据的系统目录上进行。存储引擎元数据系统目录在前面介绍的按表一致性检查过程中进行检查。这些检查如下。

- 对于所有列元数据来说，相匹配的表元数据必须存在。
- 在一个计算列的定义中引用的所有列必须存在。
- 一个索引定义中包括的所有列必须存在。

如果发现任何一致性错误，则会以如下格式的 8992 错误向 *DBCC CHECKDB* 报告最终用户结果中的内容：

```
Msg 8992, Level 16, State 1, Line 1
Check Catalog Msg 3853, State 1: Attribute (object_id=1977058079) of row
(object_id=1977058079,column_id=1) in sys.columns does not have a matching row
(object_id=1977058079) in sys.objects.
Msg 8992, Level 16, State 1, Line 1
Check Catalog Msg 3853, State 1: Attribute (object_id=1977058079) of row
(object_id=1977058079,column_id=2) in sys.columns does not have a matching row
(object_id=1977058079) in sys.objects.
```



注意：

这些检查不在 *tempdb* 数据库上运行。

11.6.3 索引视图一致性检查

虽然索引视图是数据库中的一流对象，但它存储时就好像是一个具有聚集索引的内部表一样，因此它的物理结构会作为按表一致性检查的一部分进行检查。这些一致性检查不检查索引视图的*目录*与视图定义相匹配（即内部表没有任何额外行或丢失行）。

描述索引视图一致性检查的最简单方式是利用 *indexedview* 定义生成索引视图的一个临时副本。然后利用查询处理器在实际索引视图和临时索引视图之间运行左反半联接。该查询报告实际索引视图中的丢失行或额外行。

索引视图的临时副本实际上不是完整创建的——由运行查询时查询处理器使用的查询计划决定。*DBCC CHECKDB* 使用的查询与前面介绍的非聚集索引深层次交叉检查使用的查询类似。该查询的格式如下：

```
SELECT <identifying columns of missing rows>
FROM <materialize the view temporarily> tOuter WITH (NOEXPAND)
WHERE NOT EXISTS
```

```
(
        SELECT 1
        FROM <actual view> tInner WITH (INDEX = 1)
        WHERE
        (
              ([tInner].<view columns> = [tOuter].<view columns>) OR
              ([tInner].<view columns> IS NULL AND [tOuter].<view columns> IS NULL)
        )
)
UNION ALL
SELECT <identifying columns of extra rows>
FROM <actual view> tOuter WITH (INDEX = 2)
WHERE NOT EXISTS
(
        SELECT 1
        FROM <materialize the view temporarily> tInner WITH (NOEXPAND)
        WHERE
         (
             ([tInner].<view columns> = [tOuter].<view columns>) OR
             ([tInner].<view columns> IS NULL AND [tOuter].<view columns> IS NULL)
        )
)
```

查询中使用的 NOEXPAND 暗示构建查询处理器来执行索引视图的索引扫描而不是将其扩展到它的组件部分。索引视图中的额外行报告为错误 8907，丢失的行报告为错误 8908。

这种检查可能非常费时、费空间。*Indexedview* 定义越复杂，它定义的表就越大，实现索引视图一个临时副本的时间就越长，就越可能占用 *tempdb* 中的空间。在 SQL Server 2008 中默认不执行这种检查，必须利用 EXTENDED_LOGICAL_CHECKS 选项启用。

11.6.4　XML 索引一致性检查

主 XML 索引被存储为带有聚集索引的内部表。辅助 XML 索引被存储为主 XML 索引内部表上的一个非聚集索引。一致性检查必须验证 XML 索引包含用户表中 XML 值的一种精简表示。

实现这种操作的机制与用于索引视图一致性检查的机制类似，而且可以使用相同的查询格式可视化，不过不使用 T-SQL 查询。此时，两个左反半联接（left-anti-semi-joins）可以认为是实际 XML 索引和 XML 子系统生成的 XML 索引的一个临时副本。

XML 索引中的额外行报告为错误 8907，丢失行报告为错误 8908。

这种检查的运行代价可能非常大。XML 架构越复杂、XML 列值越大，则生成 XML 索引临时副本的时间就越长，同时越有可能占用 *tempdb* 中的空间。SQL Server 2008 中默认不执行这种检查，并且必须使用 EXTENDED_LOGICAL_CHECKS 选项才能启用。

11.6.5　空间索引一致性检查

空间索引存储为一个带有聚集索引的内部表。一致性检查必须验证该空间索引包含用户表中空间值的一个精简表示。

实现这种操作的机制与用于索引视图一致性检查的机制类似，而且可以使用相同的查询格式可视化，但是不使用 T-SQL 查询。此时两个左反半联接可以认为是实际空间索引和由空间子系统生成的空间索引的一个临时副本。

空间索引中的额外行报告为错误 8907，丢失行报告为错误 8908。

这种检查可能是非常费时和费空间的，具体由定义空间索引的方式决定。索引边界框中每一级分解的单元数越高，同时按空间值存储的匹配网格数就越高，生成空间索引的临时副本所需时间就越长，同时越有可能占用 *tempdb* 中的空间。在 SQL Server 2008 中默认不执行这种检查，并且必须使用 EXTENDED_LOGICAL_CHECKS 选项启用。

11.7 DBCC CHECKDB 输出

DBCC CHECKDB 以如下 4 种方式输出信息。

- 标准输出，由一系列错误和处理 DBCC CHECKDB 命令的信息组成。
- SQL Server 错误日志中的消息。
- Microsoft Windows 应用程序事件日志中的项。
- *sys.dm_exec_requests* 目录视图中的进度报告信息。

11.7.1 标准输出

默认情况下，*DBCC CHECKDB* 报告如下内容。

- Service Broker 一致性检查的一个摘要。
- 分配错误的一个列表，附加错误数。
- 不能确定受影响表时的错误列表、附加错误数。
- 对于数据库中的每个表（包括系统目录）：
  - 行数和页数；
  - 错误列表，以及错误数量。
- 分配和一致性错误的摘要数量。
- 必须指定修复报告的错误的最低修复级别。

下面是 *DBCC CHECKDB* 输出的一个示例，是一个包含某些错误的数据库：

```
DBCC results for 'CorruptDB'.
Service Broker Msg 9675, State 1: Message Types analyzed: 14.
Service Broker Msg 9676, State 1: Service Contracts analyzed: 6.
Service Broker Msg 9667, State 1: Services analyzed: 3.
Service Broker Msg 9668, State 1: Service Queues analyzed: 3.
Service Broker Msg 9669, State 1: Conversation Endpoints analyzed: 0.
Service Broker Msg 9674, State 1: Conversation Groups analyzed: 0.
Service Broker Msg 9670, State 1: Remote Service Bindings analyzed: 0.
Service Broker Msg 9605, State 1: Conversation Priorities analyzed: 0.
Msg 8909, Level 16, State 1, Line 1
Table error: Object ID 0, index ID -1, partition ID 0, alloc unit ID 0 (type Unknown), page
ID (1:158) contains an incorrect page ID in its page header. The PageId in the page header =
(0:0).
CHECKDB found 0 allocation errors and 1 consistency errors not associated with any single
object.
DBCC results for 'sys.sysrscols'.
There are 637 rows in 8 pages for object "sys.sysrscols".
DBCC results for 'sys.sysrowsets'.
There are 92 rows in 1 pages for object "sys.sysrowsets".
```

```
DBCC results for 'sys.sysallocunits'.
There are 104 rows in 2 pages for object "sys.sysallocunits".
DBCC results for 'sys.sysfiles1'.
There are 2 rows in 1 pages for object "sys.sysfiles1".
DBCC results for 'sys.syspriorities'.
There are 0 rows in 0 pages for object "sys.syspriorities".
DBCC results for 'sys.sysfgfrag'.
There are 2 rows in 1 pages for object "sys.sysfgfrag".

<some results removed for brevity>

DBCC results for 'sys.syssqlguides'.
There are 0 rows in 0 pages for object "sys.syssqlguides".
DBCC results for 'sys.sysbinsubobjs'.
There are 3 rows in 1 pages for object "sys.sysbinsubobjs".
DBCC results for 'sys.syssoftobjrefs'.
There are 0 rows in 0 pages for object "sys.syssoftobjrefs".
DBCC results for 'sys.queue_messages_1977058079'.
There are 0 rows in 0 pages for object "sys.queue_messages_1977058079".
DBCC results for 'sys.queue_messages_2009058193'.
There are 0 rows in 0 pages for object "sys.queue_messages_2009058193".
DBCC results for 'sys.queue_messages_2041058307'.
There are 0 rows in 0 pages for object "sys.queue_messages_2041058307".
DBCC results for 'sales'.
Msg 8928, Level 16, State 1, Line 1
Object ID 2073058421, index ID 1, partition ID 72057594038386688, alloc unit ID
72057594042384384 (type In-row data): Page (1:158) could not be processed. See other errors
for details.
There are 4755 rows in 20 pages for object "sales".
CHECKDB found 0 allocation errors and 1 consistency errors in table 'sales' (object ID
2073058421).
DBCC results for 'sys.filestream_tombstone_2121058592'.
There are 0 rows in 0 pages for object "sys.filestream_tombstone_2121058592".
DBCC results for 'sys.syscommittab'.
There are 0 rows in 0 pages for object "sys.syscommittab".
CHECKDB found 0 allocation errors and 2 consistency errors in database 'CorruptDB'.
repair_allow_data_loss is the minimum repair level for the errors found by DBCC CHECKDB
(CorruptDB).
DBCC execution completed. If DBCC printed error messages, contact your system administrator.
```

虽然这种输出很全面，但是消息报告是冗余的。在标准操作中，重要的信息涉及数据库中可能出现的错误。我们总是推荐使用NO_INFOMSGS选项减少输出，只显示必要的信息。例如，下面是同一个有错误的数据库的*DBCC CHECKDB*输出，但是指定了NO_INFOMSGS选项：

```
Msg 8909, Level 16, State 1, Line 1
Table error: Object ID 0, index ID -1, partition ID 0, alloc unit ID 0 (type Unknown), page
ID (1:158) contains an incorrect page ID in its page header. The PageId in the page header =
(0:0).
CHECKDB found 0 allocation errors and 1 consistency errors not associated with any single
object.
Msg 8928, Level 16, State 1, Line 1
Object ID 2073058421, index ID 1, partition ID 72057594038386688, alloc unit ID
72057594042384384 (type In-row data): Page (1:158) could not be processed. See other errors
for details.
CHECKDB found 0 allocation errors and 1 consistency errors in table 'sales' (object ID
```

```
2073058421).
CHECKDB found 0 allocation errors and 2 consistency errors in database 'CorruptDB'.
repair_allow_data_loss is the minimum repair level for the errors found by DBCC CHECKDB
(CorruptDB).
```

正如您看到的那样，这种输出更容易读取。

当 *DBCC CHECKDB* 在 *master* 数据库上执行时有一种特殊情况。在这种情况下，*DBCC CHECKDB* 也在隐藏的资源数据库 *mssqlsystemresource* 上运行，因此结果中包含两个数据库的结果。

如果 *DBCC CHECKDB* 由于某种原因不得不过早地终止，并且可由 *DBCC CHECKDB* 控制，则会报告错误 5235，包含一种错误状态。错误状态具有如下含义。

- **0**。检测到一个致命的元数据错误。一个或多个 8930 错误（前面介绍过）伴随 5235 错误一起出现。
- **1**。*DBCC CHECKDB* 内部检测到一种无效的内部状态。一个或多个 8967 错误（前面介绍过）伴随 5235 错误一起出现。
- **2**。关键系统表的早期检查失败。一个或多个 7984 到 7988 错误（前面介绍过）伴随 5235 错误一起出现。
- **3**。由于重建事务日志后数据库不能被重启而导致紧急模式修复失败。7909 错误伴随 5235 错误一起出现。这一内容将在本章后面详细介绍。
- **4**。出现一种非法访问或断言（即使 SQL Server 2005 中重新设计了 *DBCC CHECKDB* 来避免这些问题）。
- **5**。引起 *DBCC CHECKDB* 终止的一种未知故障，尽管可能是一种得体的终止。

向 Microsoft 发送错误报告

在 SQL Server 2008 中，不论何时 *DBCC CHECKDB* 发现一个错误，都会在示例 LOG 目录中创建一个故障文件，同时还有一个 XML 形式的错误文本摘要及当前 SQL Server 错误日志文件的一个副本。如果已经将示例配置成向 Microsoft 提供反馈，那么这些文件会被自动上传。其中包含的信息会被 SQL Server 工作组使用以确定各种错误的普遍性。这样可以帮助确定在将来的一致性检查和修复功能中应该将工作重点放在哪里。

11.7.2 SQL Server 错误日志输出

每次 *DBCC CHECKDB* 执行成功后，都会向进行一致性检查的数据库的 SQL Server 错误日志中添加一项。下面是一个示例：

```
2008-11-03 00:51:11.08 spid56       DBCC CHECKDB (CorruptDB) executed by CHICAGO\
Administrator found 2 errors and repaired 0 errors. Elapsed time: 0 hours 0 minutes 0
seconds. Internal database snapshot has split point LSN = 00000044:00000188:0001 and first
LSN = 00000044:00000187:0001. This is an informational message only. No user action is
required.
```

注意其中列出了完成 *DBCC CHECKDB* 所用的时间。这样可以使数据库管理员在不借助手动计时的情况下获得对某个特殊数据库进行 *DBCC CHECKDB* 所需的平均运行时间。该示例还列出指定了哪些选项，这对于确定某个数据库原来是否被修复过很有用。

条目中还列出了关于 *DBCC CHECKDB* 创建的数据库快照的一些元数据信息，在调试错误问题时，这些信息对产品支持可能很有用。

如果 *DBCC CHECKDB* 过早地终止，则会在错误日志中输入一个简短的条目。如果存储引擎中的一

个非常严重的错误引起 *DBCC CHECKDB* 无法控制地终止，则错误日志中不会有条目。生成错误日志条目是 *DBCC CHECKDB* 结束时执行的最后一项工作。也就是说，如果出现一个错误终止运行该命令的联接，则 *DBCC CHECKDB* 不能生成错误日志条目。

11.7.3 应用程序事件日志输出

DBCC CHECKDB 在每次向 SQL Server 错误日志中写入输出时都会生成一个匹配应用程序事件日志条目。

每次 *DBCC CHECKDB* 成功结束时，都会向应用程序事件日志添加一项详细记录发现和确定的错误数量。图 11-3 所示显示了一个示例。

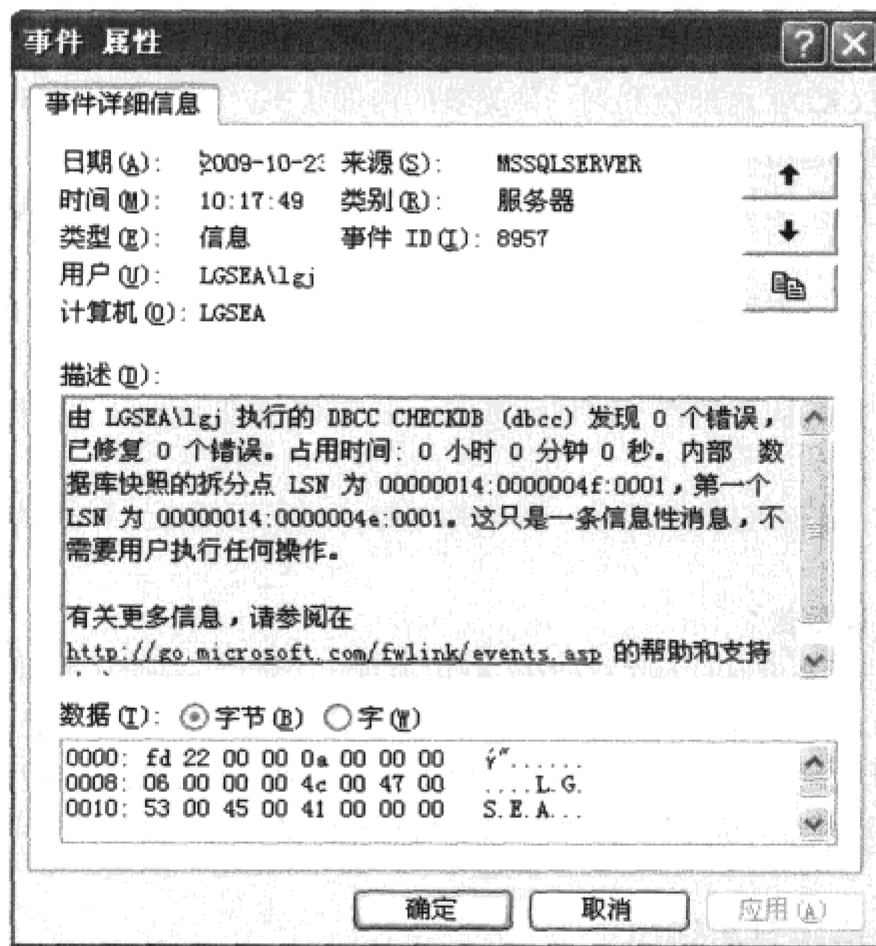


图 11-3 *DBCC CHECKDB* 正常结束时的应用程序事件日志项

如果 *DBCC CHECKDB* 发现错误，那么在包含前面介绍的错误报告故障文件元数据的事件日志中可能还有 3 个附加项。

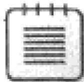
在 *DBCC CHECKDB* 决定过早终止的事件中，一个简短的条目会被添加到事件日志中。如果由于 *DBCC CHECKDB* 不可控制地终止而没有生成 SQL Server 错误日志项，也不会生成应用程序事件日志项。

11.7.4 进度报告输出

DBCC CHECKDB、*DBCC CHECKTABLE* 和 *DBCC CHECKFILEGROUP* 都会在 *sys.dm_exec_requests* 目录视图中报告自己的进度。其中有用的两个列分别是 *percent_complete*（是自解释的）和 *command*（提供正在执行的 DBCC 命令的执行当前阶段）。*DBCC CHECKDB* 的各个阶段按照执行顺序都显示在了表 11-3 中。这与 *SQL Server 联机丛书*主题“DBCC”中的列表类似，但是 *SQL Server 联机丛书*主题“DBCC”中的列表有几个错误和遗漏。

表 11-3 执行阶段的进度报告

阶 段	说 明	报告的粒度
DBCC ALLOC CHECK	分配一致性检查	这一步被认为是工作的一个单位（即进度从 0%开始，在步骤完成时跳到 100%）
DBCC ALLOC REPAIR	分配修复（如果已经指定）	在上一步骤中发现的每个分配错误有一个工作单位，同时在每个修复操作结束时进度被更新。例如，发现 8 个错误时，每个修复会将进度延长 12.5%
DBCC SYS CHECK	主要系统表的按表一致性检查	进度作为必须读取和处理的数据库页总数的一部分进行计算。每处理 1000 个页更新一次进度
DBCC SYS REPAIR	主要系统表修复，如果已经指定并且可能	如 DBCC ALLOC REPAIR 中所述
DBCC TABLE CHECK	所有表的按表一致性检查	如 *DBCC SYS CHECK* 中所述
DBCC TABLE REPAIR	用户表修复，如果已经指定	如 *DBCC ALLOC REPAIR* 中所述
DBCC SSB CHECK	Service Broker 一致性检查（如果已经指定则还会修复）	这一步被认为是一个工作单位
DBCC CHECKCATALOG	跨目录一致性检查	这一步被认为是一个工作单位
DBCC IVIEW CHECK	索引视图、XML 索引和空间索引一致性检查，如果已经被指定	被检查的每个索引视图、XML 索引及空间索引都有一个工作单位
DBCC IVIEW REPAIR	索引视图、XML 索引和空间索引修复，如果已经指定	如 *DBCC ALLOC REPAIR* 所述



注意：

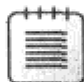
对原始系统表的检查没有报告进度。这一阶段运行速度非常快，以至于在 SQL Server 2005 的 *DBCC CHECKDB* 代码中添加进度报告时，开发组认为不值得包括一个单独的进度报告阶段。

对于 *DBCC CHECKTABLE* 来说会报告如下阶段：

- *DBCC TABLE CHECK*；
- *DBCC IVIEW CHECK*（如果上一个步骤中没有发现错误）；
- *DBCC TABLE REPAIR*（如果发现错误并指定了一个修复选项）。

对于 *DBCC CHECKFILEGROUP* 来说，会报告如下阶段：

- *DBCC ALLOC CHECK*；
- *DBCC SYS CHECK*；
- *DBCC TABLE CHECK*。



注意：

DBCC CHECKFILEGROUP 不报告修复操作。

11.8 DBCC CHECKDB 选项

*SQL Server 联机丛书*中的 *DBCC CHECKDB* 语法如下：

```
DBCC CHECKDB
[
```

```
    [ ( database_name | database_id | 0
        [ , NOINDEX
        | , { REPAIR_ALLOW_DATA_LOSS | REPAIR_FAST | REPAIR_REBUILD } ]
        ) ]
     [ WITH
        {
            [ ALL_ERRORMSGS ]
            [ , EXTENDED_LOGICAL_CHECKS ]
            [ , NO_INFOMSGS ]
            [ , TABLOCK ]
            [ , ESTIMATEONLY ]
            [ , { PHYSICAL_ONLY | DATA_PURITY } ]
        }
    ]
]
```

接下来将介绍这些选项及 *SQL Server 联机丛书*中的附加信息。

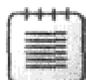
11.8.1 NOINDEX

NOINDEX 选项使 *DBCC CHECKDB* 跳过用户表的非聚集索引交叉检查（也就是说，当该选项被指定时会在系统表上执行非聚集索引交叉检查）。这些检查是 CPU 密集型操作，因此关闭这些检查可以使 *DBCC CHECKDB* 的运行速度更快。该选项很少使用，因为 PHYSICAL_ONLY 选项可以更好地禁用 CPU 密集型检查同时使 *DBCC CHECKDB* 运行速度更快。

11.8.2 修复选项

您可以指定 3 个修复选项，虽然 REPAIR_FAST 选项已经被更改，但是在 SQL Server 2005 和 SQL Server 2008 中它不能完成任何功能，它的存在只是为了保持向后兼容性。

REPAIR_REBUILD 选项试图修复数据不可能丢失情况下的错误。REPAIR_ALLOW_DATA_LOSS 选项试图修复所有错误，包括数据可能丢失时的错误（该选项的名称是精心选择的）。本章后面将进一步详细介绍修复过程和特殊修复。



注意：

这些选项要求数据库处于单用户模式。只有不能从备份中进行还原时才会使用修复功能。

11.8.3 ALL_ERRORMSGS

ALL_ERRORMSGS 选项强制 *DBCC CHECKDB* 输出发现的所有错误，而不是只输出默认情况下的 200 条错误信息。如果错误消息超过 200 条，则会将 8986 错误添加在 *DBCC CHECKDB* 结果末尾，同时需要使用 ALL_ERRORMSGS 选项重新运行该命令来查看所有错误。

也就是说，默认情况获得的是数据库中错误的不完整视图。您可以认为 200 条错误信息足以确定问题所在，但是正如本章后面所介绍的那样，查看所有错误信息是非常重要的，因为第 201 条错误信息可能会使您更改灾难恢复计划。

例如，假设 *DBCC CHECKDB* 发现了 201 个错误，但是您利用默认设置时只看到了前 200 个错误。这 200 个错误可能位于非聚集索引中，也就是说，您可以重建这些索引来修正错误。但是，您不知道的第 201 个错误是聚集索引数据页中的一个错误，该错误使得错误被修正之前重建非聚集索引是没有用的。

重要提示：

如果利用 SQL Server Management Studio 中的 ALL_ERRORMSGS 选项运行 *DBCC CHECKDB*，则会将错误数量限定为 1000。为了解决这一问题，您必须使用 *sqlcmd*。*SQL Server 联机丛书*错误地认为多次运行 *DBCC CHECKDB* 可以使您看到整个错误信息列表。

11.8.4 EXTENDED_LOGICAL_CHECKS

EXTENDED_LOGICAL_CHECKS 选项对索引视图、XML 索引和空间索引启用跨表一致性检查。由于运行这些检查的代价很大，因此默认关闭。

在 SQL Server 2005 中（其中引入了索引视图和 XML 索引检查），这些检查默认是打开的。因此，当数据库设置为第 90 个兼容级别时，会忽略 EXTENDED_LOGICAL_CHECKS 选项，同时始终执行跨表一致性检查。

11.8.5 NO_INFOMSGS

在指定 NO_INFOMSGS 选项时，输出中不会包括信息性消息。这样可以使有错误时更容易读取输出。虽然这不是默认设置，但是我们建议始终指定该选项。

11.8.6 TABLOCK

TABLOCK 选项强制 *DBCC CHECKDB* 获取数据库和表的锁以获取数据库的事务一致性视图（即脱机执行一致性检查，同时并行活动可能被阻止）。指定该选项时的锁定行为已经在本章前面介绍了。

11.8.7 ESTIMATEONLY

ESTIMATEONLY 选项计算在 *tempdb* 中需要多大空间用于存储一致性检查算法生成的事实，考虑所有其他指定选项。

DBCC CHECKDB 查看构建要检查的对象的所有批处理的动机（如本章前面介绍的那样），但实际上它不对它们进行检查。相反，它利用已经收集到的关于每个表和索引的元数据来估计它生成的每种类型事实的数量。各种数字乘以每个事实类型的大小并相加来构成该批处理的总大小。具有最大总大小的批处理就是被报告的那一个。

计算中使用的评估算法是非常保守的，从而保证精确地返回最大大小。占用的 *tempdb* 空间的实际数量可能明显低很多。例如，评估算法通过简单地计算一个堆中的所有记录并乘以 2 来估算跟踪转发记录和被转发记录所需的事实数量。几乎从来不会发生这种情况，但是这是一种充分估计。

当指定 ESTIMATEONLY 选项时，输出中不会包含任何错误或信息性消息。相反，输出格式如下：

```
Estimated TEMPDB space needed for CHECKALLOC (KB)
-------------------------------------------------
32

(1 row(s) affected)

Estimated TEMPDB space needed for CHECKTABLES (KB)
--------------------------------------------------
750

(1 row(s) affected)

DBCC execution completed. If DBCC printed error messages, contact your system administrator.
```

11.8.8 PHYSICAL_ONLY

PHYSICAL_ONLY 选项使 *DBCC CHECKDB* 跳过 CPU 密集型单表和跨表一致性检查。当该选项被指定时，*DBCC CHECKDB* 执行如下操作。

- 创建数据库的一个事务一致性静态视图。
- 对主要的系统目录执行低级一致性检查。
- 对数据库执行分配一致性检查。
- 读取并审核数据库中每个表的所有已分配页。

*SQL Server 联机丛书*错误地认为它也可以检查 B 树连接，但是实际并非如此。它也跳过对 *FILESTREAM* 数据的所有检查。

通过跳过所有 CPU 密集型一致性检查，PHYSICAL_ONLY 选项将 *DBCC CHECKDB* 从 CPU 密集型处理变成 I/O 密集型。这通常会使 *DBCC CHECKDB* 明显更快速地运行。

由于该选项强制将数据库中的所有已分配页读到缓冲池中，因此是测试存在的所有页校验和的一种最好方式。

PHYSICAL_ONLY 选项具有如下约束。

- 与 DATA_PURITY 选项是互斥的。
- 与修复选项是互斥的。
- 启动 NO_INFOMSGS 选项。

11.8.9 DATA_PURITY

DATA_PURITY 选项强制运行单列数据纯度检查（如本章前面介绍的那样）。默认情况下，这些检查在标记为纯的数据库上运行。纯数据库是指在 SQL Server 2005 或 SQL Server 2008 上创建的数据库，或者运行带有 DATA_PURITY 选项的 *DBCC CHECKDB* 命令但没有发现错误的一个升级数据库（将数据库标记为纯的操作不能撤销）。

对于所有其他数据库来说，只有指定该选项时才运行数据纯度检查。DATA_PURITY 和 PHYSICAL_ONLY 选项是互斥的。

11.9 数据库修复

除在数据库上执行一致性检查之外，*DBCC CHECKDB* 可以对找到的大部分错误执行修复。我们说“大部分”而不是“全部”，因为有些错误是 *DBCC CHECKDB* 不能修复的。这包括：

- 主要系统目录聚集索引的叶级中的错误；
- PFS 页眉中的错误；
- 数据纯度错误（错误 2570）；
- 系统目录交叉检查时发现的错误（错误 8992）。

如果出现其中的任意一种错误，*DBCC CHECKDB* 都会尽可能地修复，同时用错误 2540 指示哪些错误不能被修复（系统不能自己修复这种错误）。

本书不是要讨论何时应该使用修复，但是从有效的备份中进行恢复会使数据丢失降到最低，这通常是首选操作。数据库修复功能实际上是在备份不可用时才使用的选项。

修复通常应该被作为最后一招的原因有两点。首先，REPAIR_ALLOW_DATA_LOSS 启用的大部分修复都是删除所有错误并确定错误对象的所有联接。这是最快、最简单的方式，这无疑是删除错误的正确方法。其次，并不是所有错误都可以被修复，正如我们前面介绍的那样。

但是，如果没有可用的备份，则修复可能是必要的。使用的必要修复级别在 *DBCC CHECKDB* 输出的末尾使用错误 8958 进行报告，如下所示：

```
repair_rebuild is the minimum repair level for the errors found by DBCC CHECKDB
(ProductionDB).
```

可修复的错误分为两组——可以在不丢失数据的情况下被修复的错误（例如，非聚集索引、索引视图、XML 或空间索引、转发<>中被转发记录联接中的错误），以及要求数据丢失的错误（包含堆和聚集索引的大部分错误）。

对于第一组中的错误来说，REPAIR_REBUILD 选项已经足够。对于第二组中的错误来说，需要 REPAIR_ALLOW_DATA_LOSS 选项。如果没有指定，则有些错误不能被修复，同时 *DBCC CHECKDB* 会利用一个错误 8923 说明这是因为指定了一个更低的修复级别（"DBCC 语句中的修复级别会避开这种修复"）。

11.9.1 修复机制

如果一个修复选项被指定，然后 *DBCC CHECKDB* 完成每一阶段的一致性检查后，则此时会执行修复。这样可以保证在数据库上执行后续一致性检查和修复（没有低级错误）。

在需要运行修复时，*DBCC CHECKDB* 内部的修复子系统会传递给一致性检查发现的错误列表。错误不是按照它们被发现的顺序修复的——相反，它们按照修复的顺序进行排序。

例如，假设一个非聚集索引有一个错误 IAM 页和一个丢失的页。对错误的 IAM 页的修复是要重建非聚集索引，对丢失的记录的修复只是简单地插入一个新的非聚集索引记录。如果丢失的记录是在错误的 IAM 页之前修正的，新记录的插入是无用的，因为非聚集索引被重建来修复错误的 IAM 页。

因此，通过修复的侵入方式对所有错误进行分级是有意义的，同时首先执行侵入性最强的修复。这通常会跳过侵入性较弱的修复，因为它们可以作为执行侵入性较强修复的一个副作用而被修复。继续前面的示例，非聚集索引 IAM 页错误比丢失的非聚集索引记录的级别更低。当该索引被重建以修复错误的 IAM 页时，新索引包括丢失的记录，因此作为一个副作用修正丢失的非聚集索引记录错误。

修复级别也会防止修复系统无意间引起更多错误。例如，假设一个表有一个错误的非聚集索引，同时在聚集索引的叶级上有一个错误的页。首先执行侵入性更强的修复（重新分配聚集索引叶级页）是必要的。这样可以保证在非聚集索引被重建时，重建利用一个不受错误影响的聚集索引作为它的基准。如果非聚集索引首先被重建，然后聚集索引叶级页被重新分配，那么非聚集索引是错误的。

每个修复在一个单独的事务中执行。这样可以使 *DBCC CHECKDB* 处理修复故障，接下来修复其他错误。

DBCC CHECKDB 的输出包含被执行的修复的细节。具有修复的所有错误的整个列表不是本书讨论的范围，示例修复包括如下。

- 修正一个数据或索引页上不正确记录的数量。
- 修正不正确的 PFS 页字节。
- 从非聚集索引中删除一条额外记录。
- 向非聚集索引中插入一条记录（一种比重建一个大型非聚集索引更有效的修复单条记录的选项）。
- 将一个错误的分区记录移动到正确的分区。
- 重建一个非聚集索引。

- 重建一个索引视图、XML 索引或空间索引。
- 删除孤立的行外 LOB 值或 *FILESTREAM* 文件。
- 重建一个聚集索引。
- 重新分配一条数据记录，同时级联删除该记录所引用的所有非聚集索引记录和行外 LOB 或 *FILESTREAM* 值。
- 重新分配整个数据页，同时级联删除后续页。
- 各种 GAM、SGAM 和 IAM 分配位图中设置或未设置位。
- 修正 IAM 链中上一个页和下一个页的联接。
- 截断错误 IAM 页上的一个 IAM 链，将剩余的 IAM 链联到一起，同时执行后续重建和级联删除。
- 解析分配给多个对象的页或扩展。

包括数据丢失的修复也有其他可能的结果。如果受修复影响的表包含在一个外键关系中，则在运行修复之后，这种关系可能被破坏，因此应该执行 *DBCC CHECKCONSTRAINTS*。如果受到修复影响的表是应用程序发布的一部分，那么修复没有被复制，因此应该在运行修复之后重新初始化订阅。

11.9.2 紧急模式修复

另一种附加的修复功能只有在数据库处于 EMERGENCY 模式时才会触发。当数据库的事务日志已经被破坏并且没有备份可以还原时会使用 EMERGENCY 模式。此时，标准的修复不起作用——修复被完整地记入日志，如果事务日志被破坏则不会出现这种情况。

在 SQL Server 2000 和更早的版本中，EMERGENCY 模式没有记录，用于使用未记录的 *DBCC REBUILD_LOG* 命令使事务日志被重建。遗憾的是，该程序在网上被公布但是通常没有所有必要的步骤。因此，我们决定添加在 SQL Server 2005 中重建事务日志并恢复数据库的一种记录和受支持的方法。该功能被称为*紧急模式修复*并且其机制在 SQL Server 2008 中没有改变。

当数据库处于 EMERGENCY 模式和 SINGLE_USER 模式，并且 *DBCC CHECKDB* 利用 REPAIR_ALLOW_DATA_LOSS 选项运行时，会采取如下步骤。

（1）强制在事务日志（如果存在）上运行恢复。

这通常是带有 CONTINUE_AFTER_ERROR 的恢复，与使用带有 BACKUP 或 RESTORE 的 CONTINUE_AFTER_ERROR 类似。其中的观点是数据库已经不一致，因为事务日志是错误的，或者数据库中某些内容是有错误的，因此不能完成恢复。

假定数据库是不一致的并且事务日志将被重建，这对于在信息被抛弃并且新信息被建立之前从日志中挽救尽可能多的事务信息很有意义。

利用 CONTINUE_AFTER_ERROR 功能进行恢复只有在 *DBCC CHECKDB* 中可行。

（2）如果事务日志是错误的，则会重建事务日志。

（3）利用 REPAIR_ALLOW_DATA_LOSS 选项在数据库上运行完整的一致性检查。

（4）使数据库联机。

提示:

这一操作大多数情况下会执行成功，虽然有数据丢失。但是我们曾经看到过在该操作中失败的情况，尤其是在错误的文件系统上，因此备份是避免数据丢失的推荐方法。

11.9.3 哪些数据可以由修复删除

最坏的情况下您除了使用 REPAIR_ALLOW_DATA_LOSS 选项之外没有其他选择，有些数据不可避免地会丢失，如前所述。您的任务是查明丢失了哪些数据，从而使其可以被重建，或者查明数据库的其他哪些部分被修理以反映这种丢失。

在运行修复之前，您可以试着检查 *DBCC CHECKDB* 报告为错误的一些页来查看页上有哪些数据。假设有如下的错误：

```
Server: Msg 8928, Level 16, State 1, Line 2
Object ID 645577338, index ID 0: Page (1:168582) could not be processed. See other errors
for details.
```

您可以试着利用 *DBCC PAGE* 检查页（1:168582）。根据页中错误的严重程度，您可能能够查看页上的某些记录并在页被修复操作重新分配时查明哪些数据丢失了。

运行修复之后，就可能知道哪些数据已经被删除了。除非您非常熟悉数据库中的数据，否则使用下面两个选项。

- 在运行修复之前对有错误的数据库创建一个备份，使您可以比较预修复和修复后的数据并查看丢失的内容。如果数据库有严重错误则很难处理——您可能需要使用 BACKUP 和 RESTORE 的 WITH CONTINUE_AFTER_ERROR 选项完成此操作。
- 在运行修复之前启动一个明确的事务。一般不能在事务内部运行修复。在修复完成之后，可以检查数据库来查看修复所执行的操作，同时如果您希望撤销修复，可以简单地回滚显式事务。

在修复完成之后，就可以查询修复的数据库来发现已经修复了哪些数据。例如，假设一个修复从带有一个标识列的聚集索引中删除了一个叶级页。可以构建查找被删除记录的查询，例如：

```
-- Start of the missing range is when a value does not have a plus-1 neighbor.
SELECT MIN(salesID + 1) FROM DemoRestoreOrRepair.dbo.sales as A
WHERE NOT EXISTS (
      SELECT salesID FROM DemoRestoreOrRepair.dbo.sales as B
      WHERE B.salesID = A.salesID + 1);
GO
-- End of the missing range is when a value does not have a minus-1 neighbor
SELECT MAX(salesID - 1) FROM DemoRestoreOrRepair.dbo.sales as A
WHERE NOT EXISTS (
      SELECT salesID FROM DemoRestoreOrRepair.dbo.sales as B
      WHERE B.salesID = A.salesID - 1);
GO
```

在运行一项修复之后，至少您应该对错误进行一次完整的备份并分析根本原因来找出引起问题的原因。

11.10 除 DBCC CHECKDB 之外的一致性检查命令

这一部分介绍每种 *DBCC CHECK* 命令的功能。从顺序上来讲，对所有不同的一致性检查 DBCC 命令所完成的功能、应该进行哪些检查及以怎样的顺序进行检查有很多疑惑。*DBCC CHECKDB* 包括所有 *DBCC CHECK...*命令（除 *DBCC CHECKIDENT* 和 *DBCC CHECKCONSTRAINTS*）的功能。

11.10.1 DBCC CHECKALLOC

DBCC CHECKALLOC 执行如下操作：

- 原始的系统目录一致性检查；
- 数据库上的分配一致性检查。

它默认使用一个数据库快照并与 *DBCC CHECKDB* 具有相同的选项，除了以下选项：

- PHYSICAL_ONLY；
- REPAIR_REBUILD；
- DATA_PURITY。

这几个选项对分配一致性检查没有意义。

如果允许信息性消息，则它会输出分配给数据库中每个分配单位的页和扩展数量的各方面信息，还有 IAM 链中的第一个 IAM 页和根页。当分配一致性检查作为 *DBCC CHECKDB* 一部分执行时，不会返回该信息。下面是示例输出：

```
DBCC results for 'CorruptDB'.
***************************************************************
Table sys.sysrscols                 Object ID 3.
Index ID 1, partition ID 196608, alloc unit ID 196608 (type In-row data). FirstIAM (1:188).
Root (1:189). Dpages 8.
Index ID 1, partition ID 196608, alloc unit ID 196608 (type In-row data). 10 pages used in 1
dedicated extents.
Total number of extents is 1.
***************************************************************
<some results removed for brevity>
***************************************************************
Table sys.syscommittab                  Object ID 2137058649.
Index ID 1, partition ID 72057594038583296, alloc unit ID 72057594042580992 (type In-row
data). FirstIAM (0:0). Root (0:0). Dpages 0.
Index ID 1, partition ID 72057594038583296, alloc unit ID 72057594042580992 (type In-row
data). 0 pages used in 0 dedicated extents.
Index ID 2, partition ID 72057594038648832, alloc unit ID 72057594042646528 (type In-row
data). FirstIAM (0:0). Root (0:0). Dpages 0.
Index ID 2, partition ID 72057594038648832, alloc unit ID 72057594042646528 (type In-row
data). 0 pages used in 0 dedicated extents.
Total number of extents is 0.
File 1. The number of extents = 25, used pages = 174, and reserved pages = 195.
           File 1 (number of mixed extents = 18, mixed pages = 139).
    Object ID 3, index ID 1, partition ID 196608, alloc unit ID 196608 (type In-row data),
data extents 1, pages 10, mixed extent pages 9.
    Object ID 5, index ID 1, partition ID 327680, alloc unit ID 327680 (type In-row data),
data extents 0, pages 2, mixed extent pages 2.
    Object ID 7, index ID 1, partition ID 458752, alloc unit ID 458752 (type In-row data),
data extents 0, pages 4, mixed extent pages 4.
    Object ID 7, index ID 2, partition ID 562949953880064, alloc unit ID 562949953880064
(type In-row data), index extents 0, pages 2, mixed extent pages 2.
<some results removed for brevity>

  Object ID 2073058421, index ID 1, partition ID 72057594038386688, alloc unit ID
72057594042384384 (type In-row data), data extents 2, pages 23, mixed extent pages 9.
The total number of extents = 25, used pages = 174, and reserved pages = 195 in this
database.
```

```
        (number of mixed extents = 18, mixed pages = 139) in this database.
CHECKALLOC found 0 allocation errors and 0 consistency errors in database 'CorruptDB'.
DBCC execution completed. If DBCC printed error messages, contact your system administrator.
```

11.10.2 DBCC CHECKTABLE

DBCC CHECKTABLE 执行如下操作：

- 原始的系统目录一致性检查；
- 指定单表上每个表的一致性检查；
- 在引用指定表的索引视图上进行跨表一致性检查。

如果发现指定的表中有错误，则不会执行跨表一致性检查。这一点比 *DBCC CHECKDB* 稍微严格一些。

此外，可以指定一个非聚集索引 ID 限制只对非聚集索引进行非聚集索引交叉检查。如果任何修复选项被使用，则不能指定该项。

它默认使用一个数据库快照并与 *DBCC CHECKDB* 具有相同的选项，包括修复。

该命令的输出限定于被检查的表。

11.10.3 DBCC CHECKFILEGROUP

DBCC CHECKFILEGROUP 执行如下操作：

- 原始系统目录一致性检查；
- 对文件组进行分配一致性检查；
- 对存储在文件组中的所有表进行单表一致性检查；
- 跨表一致性检查，只要索引视图、XML 索引和空间索引被存储在文件组中，同时它们所基于的表也存储在文件组中。

它默认使用一个数据库快照同时与 *DBCC CHECKDB* 具有相同的选项，除了以下选项：

- PHYSICAL_ONLY；
- DATA_PURITY；
- 所有修复选项。

当一个表和它的所有非聚集索引不存储在同一个文件组中时，单表一致性检查有如下一些微妙之处。

- 如果表被存储在指定的文件组中，但是一个或多个非聚集索引存储在其他文件组中，则不对它们进行一致性检查。
- 如果一个非聚集索引存储在指定的文件组中，但是表（堆或聚集索引）存储在另一个文件组中，则不对非聚集索引进行一致性检查。

基本原则是 *DBCC CHECKFILEGROUP* 不执行跨文件组的一致性检查。

该命令的输出与 *DBCC CHECKDB* 相同，但是它不报告任何 Service Broker 详细信息并且只对指定文件组中的表进行检查。

11.10.4 DBCC CHECKCATALOG

DBCC CHECKCATALOG 执行如下操作：

- 原始系统目录一致性检查；
- 跨目录一致性检查。

它默认使用一个数据库快照并且只有 NO_INFOMSGS 选项。如果不能创建数据库快照，则需要运行一个排他数据库锁。假设在 *tempdb* 上不能获得数据库快照和排他锁，则 *DBCC CHECKCATALOG* 不能在

tempdb 数据库上运行（或者作为 *DBCC CHECKDB* 的一个部分或者作为单机命令）。

除非发现错误，否则该命令的输出是空的。

11.10.5 DBCC CHECKIDENT

DBCC CHECKIDENT 检查指定表的标识值是有效的（即比表中包含的最高标识值大）并且在必要时自动重新设置。它通过扫描指定表中的行进行工作来查找最高的标识值，然后将其与存储在表元数据中的下一个标识值进行比较。

该命令也可以用于在需要时手动重设标识值，此时应该小心，防止意外地在标识列中生成重复值。

如果该命令只是在检查标识值，则表用目标共享锁锁定，从而对并发操作产生最小的影响。如果已经指定了一个新值，则在它修改表的元数据时用模式修改锁锁定该表。

更多信息：

*SQL Server 联机丛书*主题针对此命令的主题 “DBCC CHECKIDENT(Transact-SQL)” 介绍了各种命令及其影响，网址为 *http://msdn.microsoft.com/en-us/library/ms176057.aspx*。

11.10.6 DBCC CHECKCONSTRAINTS

DBCC CHECKCONSTRAINTS 检查在数据库中定义的已启用的 FOREIGN KEY 和 CHECK 约束。它可以检查单个约束、表上的所有约束或数据库中的所有约束。如果指定 ALL_CONSTRAINTS 选项，那么还检查禁用的 FOREIGN KEY 和 CHECK 约束。

它的工作方式是创建一个查询来查找违反正在被检查的约束的所有行。该查询利用一个内部查询暗示来通知查询处理器 *DBCC CHECKCONSTRAINTS* 正在运行，并且它不应该基于现有约束而缩短查询。

它不使用数据库快照，在设置的会话孤立级别下运行。您必须将 CONCAT_NULL_YIELDS_NULL 会话选项设置为 ON，否则该命令会失败并报告错误 2507。如果有行违反约束，则行的键会与包含该行的表名和违反的约束名一起输出。

提示：

DBCC CHECKCONSTRAINTS 应该在执行完所有 DBCC 修复后运行，因为这些修复没有考虑任何约束。

11.11 小结

正如本章内容所述，SQL Server 2008 在数据库上可以执行的一致性检查是相当广泛的，就宽度、深度和有效性而言，都比以前的版本有明显的改进。

同时您可以看到，*DBCC CHECKDB* 花费如此长的时间完成一个大型复杂数据库的原因。我们尽量包括 *DBCC CHECKDB* 可以报告的每一条错误的信息，以及一致性检查机制的背景知识（做出决定的原因）。

希望在您遇到实际错误时这些信息会对您有所帮助。