



Professional Microsoft SQL Server 2008 Programming

SQL Server 2008

高级程序设计

(美) Robert Vieira 著
杨华 腾灵灵 译



清华大学出版社

SQL Server 2008高级程序设计

Professional Microsoft SQL Server 2008 Programming

Microsoft SQL Server 2008极大地完善了数据库引擎的核心组件，也改变了数据库应用程序的构建方式。《SQL Server 2008高级程序设计》由世界顶尖SQL Server权威专家Robert Vieira编写，旨在指导您熟练运用一系列日趋复杂的功能，助您更高效地管理数据。

本书首先介绍SQL Server 2008的新功能，然后在更详实的示例代码的引导下全面深入地展开论述，讨论了如何编写复杂查询、构建各种数据结构以及提高应用程序性能，还讲述了如何管理高级脚本和数据库以及如何确定和改正脚本错误。

本书提供了快速创建和部署数据驱动的解决方案来满足业务需求的信息，介绍了新数据类型、索引结构、管理功能和高级时区处理等重要内容，掌握这些知识后，您将使自己的数据库发挥出最大功效。

主要内容

- ◆ 除规范化外的数据设计技巧
- ◆ 尽量提高应用程序运行速度的方法
- ◆ 有关存储过程和用户定义函数的全部内容
- ◆ 存储过程的高级处理方法
- ◆ 报表服务和集成服务的用法
- ◆ 提高数据库安全性的提示信息
- ◆ 如何利用XML和XQuery支持
- ◆ 通过修改特定数据值进行推理分析的步骤

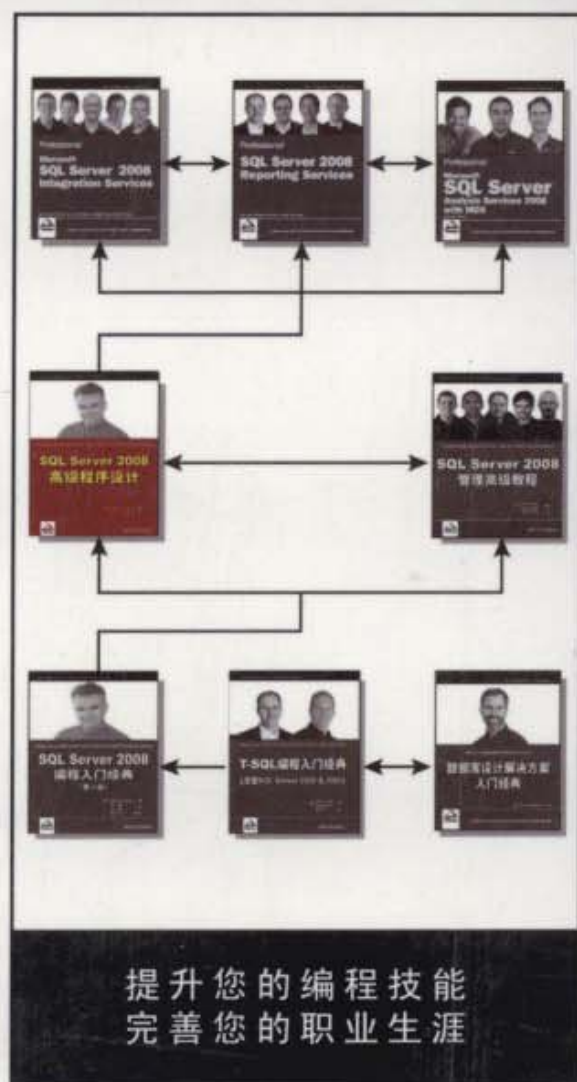
读者对象

本书面向想要学用所有SQL Server 2008功能的有经验的开发人员。

源代码下载及技术支持

<http://www.wrox.com>

<http://www.tupwk.com.cn/downpage>



Wrox Professional guides are planned and written by working programmers to meet the real-world needs of programmers, developers, and IT professionals. Focused and relevant, they address the issues technology professionals face every day. They provide examples, practical solutions, and expert education in new technologies, all designed to help programmers do a better job.

p2p.wrox.com
The programmer's resource center

www.wrox.com



图书上架
分类建议

数据库

SQL Server开发、软件开发

读者信箱: wkservice@vip.163.com
投稿信箱: bookservice@263.net

ISBN 978-7-302-22272-9



9 787302 222729 >

定价: 98.00元

SQL Server 2008

高级程序设计

(美) Robert Vieira 著

杨华 腾灵灵 译

清华大学出版社

北 京

Robert Vieira

Professional Microsoft SQL Server 2008 Programming

EISBN: 978-0-470-25702-9

Copyright © 2009 by Wiley Publishing, Inc.

All Rights Reserved. This translation published under license.

本书中文简体字版由 Wiley Publishing, Inc. 授权清华大学出版社出版。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。

北京市版权局著作权合同登记号 图字: 01-2009-3236

本书封面贴有 Wiley 公司防伪标签, 无标签者不得销售。

版权所有, 侵权必究。侵权举报电话: 010-62782989 13701121933

图书在版编目(CIP)数据

SQL Server 2008 高级程序设计/(美)维埃拉(Vieira, R.)著; 杨华, 腾灵灵 译.

—北京: 清华大学出版社, 2010.4

书名原文: Professional Microsoft SQL Server 2008 Programming

ISBN 978-7-302-22272-9

I. S… II. ①维… ②杨… ③腾… III. 关系数据库—数据库管理系统, SQL Server 2008—程序设计

IV. TP311.138

中国版本图书馆 CIP 数据核字(2010)第 048739 号

责任编辑: 王 军 韩宏志

装帧设计: 孔祥丰

责任校对: 成凤进

责任印制: 王秀菊

出版发行: 清华大学出版社

地 址: 北京清华大学学研大厦 A 座

<http://www.tup.com.cn>

邮 编: 100084

社 总 机: 010-62770175

邮 购: 010-62786544

投稿与读者服务: 010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者: 清华大学印刷厂

装 订 者: 三河市金元印装有限公司

经 销: 全国新华书店

开 本: 185×260 印 张: 46.5 字 数: 1250 千字

版 次: 2010 年 4 月第 1 版 印 次: 2010 年 4 月第 1 次印刷

印 数: 1~4000

定 价: 98.00 元

产品编号: 030073-01

内容简介

本书由世界顶尖SQL Server权威专家Robert Vieira撰写，它介绍SQL Server 2008的新功能，讨论如何设计性能卓越的应用程序，分析如何提高应用程序安全性，讲述如何管理高级脚本和数据库，并陈述新数据类型、高级查询、XML集成、索引结构、高级时区处理和数据仓库等重要内容。

作者简介

当1978年的计算机热潮席卷而来时，Robert Vieira对计算机技术萌生了浓厚兴趣。他从1983年开始攻读计算机信息系统学位，于1985年后期涉足自己钟爱的“计算机程序故障”领域，于1990年获得商业管理学位。丰富的商业知识和计算知识为保障他的事业取得成功奠定了坚实基础。获得学士学位后，他又相继获得CMA、MCSD、MCT、MCDBA以及EMT等认证。

Robert目前担任Huron Consulting Group的Stockamp公司的数据库团队领导，迄今已出版了6本有关SQL Server开发的书籍。

数字图书馆
PDF

前言

我们将在新的起点上重新启航。我们等待 SQL Server 2005 用了长达 5 年的时间，但 Microsoft 给我们带来了惊喜，SQL Server 2008 的到来只用了 3 年时间。

我对 SQL Server 2008 在这么短的时间里引入这么多功能并不感到惊讶。真正使我感到惊讶的是这个新版本中添加了大量“袖珍功能”。尽管 Microsoft 将一些主要功能(如基于管理的策略)作为市场宣传的噱头，但该产品最突出的新功能其实都是相当细微的。

SQL Server 2008 包括了大量看似微小但却非常有用的功能，例如：

- 新增的且非常有用的数据类型(分离的日期和时间数据类型以及对地理空间数据和分层数据表示的支持)。
- 新增的 MERGE 命令(将 INSERT、UPDATE 和 DELETE 选项结合到一条语句中，其他产品有时将其称作 UPSERT 语句)。
- 改进了 Reporting Service，可提供更优雅的报表。
- 跟踪和提供了“缺少”索引信息(在优化时，会关注可提高性能的“缺少”索引信息)。

这些都是面向初学者的。

对于已阅读过我所写的 2005 版的图书的读者来说，会看到本书继续对初级和高级内容作了区分。真正“初级”内容的讨论已经包括在《SQL Server 2008 编程入门经典》(由清华大学出版社引进并出版)一书中。一些介于初级、中级或高级主题之间的内容，我将作较为深入的介绍，但对于此版本来说，大部分初级内容都罗列在入门书籍中(本书最后添加的一个新附录给出了简短的语法列表和一些示例，但数百页的内容现在浓缩成了数十页)。

好消息是，这让本书可以涵盖更广泛的主题。这就让我可以更接近最初为“高级”篇设定的目标：提供产品的大部分基础信息，即使你无法在每个领域都做到专业级，你也可从整体上理解 SQL Server 并构建一个更好的系统，了解许多 SQL Server 功能领域所涉及的内容，并准备在必要时获取更多的信息。

除此之外，本书保持了作者一贯的写作风格。书中涵盖了大部分的附加服务、高级编程结构(如.NET 程序集)以及一些支持的对象模型(可用于管理 SQL Server 及其各种引擎)。

0.1 版本问题

本书是针对 SQL Server 2008 编写的，不过，书中也追溯了一些之前版本的内容，并且密切关注了与 SQL Server 2005 甚至 SQL Server 2000 相关的向后兼容性问题。之前的版本太陈旧了(简言之，在 SQL Server 2008 发布时，SQL Server 6.5 和 7.0 已经面世 10 年之久了)，书中几乎不会涉及它们。

0.2 读者对象

本书假定你已经具有一些 SQL Server 经验,并准备并具有介于中级到高级的编程水准。此外,本书主要面向较高层次的开发人员。

除了方便参考的附录外,本书很少涉及初级内容。它假定你有编写 DML 语句的经验,了解所有主流 SQL Server 对象(如视图、存储过程、用户自定义函数以及触发器)的基础知识。如果你准备好接受高级主题,但又想要温习初级知识,那我强烈建议你学习《SQL Server 2008 编程入门经典》一书,因为这两本书的内容是经过特别设计的,内容较少重叠。

0.3 本书内容

本书介绍的是关于 SQL Server 的内容。更准确地说,本书是围绕 SQL Server 开发展开的。大部分的概念都与所使用的客户端语言无关,尽管本书中利用了客户端语言的示例一般都使用 C# 语言(有些示例采用了多种语言)。

对于那些从 SQL Server 早期版本过渡而来的读者,我们将对因产品有多个版本而导致的一些“陷阱”作相关讨论。

0.4 本书结构

与我所编写的其他书籍一样,本书采用自由的写作风格。其结构较松散。每一章首先概括全章讲述的内容,然后再作详细讲述。对于每一个主题,先给出一些背景知识,然后在适当的时候提供例子。本书中的例子一般都比较简短,但它们可帮助你迅速掌握所讨论主题中涉及的概念。

由于本书是属于中高层次的,为了与初级内容相衔接,我们首先回顾了工具和数据类型(因为它们都有变化),但随即就进入到高级主题的介绍中。

为了合理运用本书,你需要一台运行 SQL Server 2008 的计算机。尽管我强烈建议你使用开发人员版本,但本书的大部分示例和建议都适用于所有 SQL Server 版本。不过,我还是建议用 SQL Server 完整版,而不是速成版。

0.5 源代码

在读者学习本书中的示例时,可以手工输入所有的代码,也可以使用本书附带的源代码文件。本书使用的所有源代码都可以从本书合作站点 <http://www.wrox.com/> 或 www.tupwk.com.cn/downpage 上下载。登录到站点 <http://www.wrox.com/>, 使用 Search 工具或使用书名列表就可以找到本书。接着单击本书细目页面上的 Download Code 链接,就可以获得所有源代码。

注释:

由于许多图书的标题都很类似,所以按 ISBN 搜索是最简单的,本书英文版的 ISBN 是

978-0-470-25702-9。

下载代码后，只需用自己喜欢的解压缩软件对它进行解压缩即可。另外，也可以进入 <http://www.wrox.com/dynamic/books/download.aspx> 上的 Wrox 代码下载主页，查看本书和其他 Wrox 图书的所有代码。

0.6 勘误表

尽管我们已经尽了各种努力来保证文章或代码中不出现错误，但是错误总是难免的，如果你在本书中找到了错误，例如拼写错误或代码错误，请告诉我们，我们将非常感激。通过勘误表，可以让其他读者更方便地学习，当然，这还有助于提供更高质量的信息。

请给 wkservice@vip.163.com 发电子邮件我们就会检查你的反馈信息，如果是正确的，我们将在本书的后续版本中采用。

要在网站上找到本书英文版的勘误表，可以登录 <http://www.wrox.com>，通过 Search 工具或书名列表查找本书，然后在本书的细目页面上，单击 Book Errata 链接。在这个页面上可以查看到 Wrox 编辑已提交和粘贴的所有勘误项。完整的图书列表还包括每本书的勘误表，网址是 www.wrox.com/misc-pages/booklist.shtml。

0.7 P2P.WROX.COM

要与作者和同行讨论，请加入 p2p.wrox.com 上的 P2P 论坛。这个论坛是一个基于 Web 的系统，便于你张贴与 Wrox 图书相关的消息和相关技术，与其他读者和技术用户交流心得。该论坛提供了订阅功能，当论坛上有新的消息时，它可以给你传达感兴趣的论题。Wrox 作者、编辑和其他业界专家和读者都会到这个论坛上来探讨问题。

在 <http://p2p.wrox.com> 上，有许多不同的论坛，它们不仅有助于阅读本书，还有助于开发自己的应用程序。要加入论坛，可以遵循下面的步骤：

- (1) 进入 p2p.wrox.com，单击 Register 链接。
- (2) 阅读使用协议，并单击 Agree 按钮。
- (3) 填写加入该论坛所需要的信息和自己希望提供的其他信息，单击 Submit 按钮。
- (4) 你会收到一封电子邮件，其中的信息描述了如何验证帐户，完成加入过程。

提示：

不加入 P2P 也可以阅读论坛上的消息，但要张贴自己的消息，就必须加入该论坛。

加入论坛后，就可以张贴新消息，响应其他用户张贴的消息。可以随时在 Web 上阅读消息。如果要想让该网站给自己发送特定论坛中的消息，可以单击论坛列表中该论坛名旁边的 Subscribe to this Forum 图标。

关于使用 Wrox P2P 的更多信息，可阅读 P2P FAQ，了解论坛软件的工作情况以及 P2P 和 Wrox 图书的许多常见问题。要阅读 FAQ，可以在任意 P2P 页面上单击 FAQ 链接。

0.8 www.professionalsql.com

从 <http://www.professionalsql.com> 中可找到对本书的一些支持和相关博文。尽管一般支持请求应通过 p2p.wrox.com 网站发送，但 professionalsql.com 也提供了重要的镜像文件下载，以及我个人对当前开发现状的一些说明。可以通过 robv@professionalsql.com 与我联系，但我有几点说明：

- 不要发送那些从课堂测验或期中考试中摘取的问题。
- 尽量发送那些无法从一般地方([google](http://google.com)、p2p.wrox.com 网站、许多主流 SQL Server 网站或是简单的联机丛书)找到答案的问题。
- 我可能无法对所有帮助、建议或其他问题作答。
- 公布我的电子邮件是对读者的信任，请勿滥用。

我非常乐意听到你们有关使用 SQL 的经验，请不吝与我分享。

目 录

第 1 章 回顾 SQL Server 中的对象	1
1.1 数据库的构成	1
1.2 数据库对象概述	2
1.2.1 数据库对象	2
1.2.2 事务日志	5
1.2.3 最基本的数据库对象: 表	5
1.2.4 模式	6
1.2.5 文件组	7
1.2.6 图表	7
1.2.7 视图	8
1.2.8 存储过程	9
1.2.9 用户自定义函数	9
1.2.10 用户和角色	10
1.2.11 规则	10
1.2.12 默认值	10
1.2.13 用户自定义数据类型	10
1.2.14 全文目录	10
1.3 SQL Server 数据类型	11
1.4 SQL Server 对象标识符	15
1.4.1 需要命名的对象	15
1.4.2 命名规则	15
1.5 小结	16
第 2 章 工具	17
2.1 联机丛书	17
2.2 SQL Server 配置管理器	18
2.2.1 服务管理	18
2.2.2 网络配置	19
2.2.3 协议	20
2.2.4 客户端配置	21
2.3 SQL Server Management Studio	23
2.3.1 启动 Management Studio	23
2.3.2 查询编辑器	25

2.4 SQL Server Business Intelligence Development Studio	29
2.5 SQL Server 集成服务(SSIS)	29
2.6 Reporting Services	29
2.7 Bulk Copy Program(bcp)	30
2.8 SQL Server Profiler	30
2.9 sqlcmd	30
2.10 小结	31
第 3 章 提出更好的问题: 高级查询	33
3.1 子查询概述	34
3.2 构建嵌套子查询	34
3.2.1 使用单值 SELECT 语句的 嵌套查询	35
3.2.2 使用返回多个值的子查询的 嵌套查询	35
3.2.3 ANY、SOME 和 ALL 运算符	37
3.3 相关子查询	37
3.3.1 相关子查询的工作方式	37
3.3.2 WHERE 子句中的相关子查询	38
3.3.3 SELECT 列表中的相关子查询	39
3.4 派生表	40
3.5 EXISTS 运算符	42
3.6 INTERSECT 和 EXCEPT 运算符	44
3.6.1 EXCEPT	45
3.6.2 INTERSECT	45
3.6.3 比较 EXCEPT 和 INTERSECT 与相应的 EXISTS 和 NOT EXISTS 语句	46
3.7 通用表表达式(CTE)	49
3.8 递归查询	50
3.9 合并	53

3.10 利用外部调用完成复杂操作	57	5.7 数据库重用	122
3.11 性能考虑	57	5.7.1 可重用数据库的候选	122
3.12 小结	59	5.7.2 如何分解事物	122
第4章 XML 集成	61	5.7.3 可重用性的高昂代价	123
4.1 XML 数据类型	62	5.8 反规范化	123
4.1.1 定义 XML 数据类型的列	62	5.9 通过分区方法进行扩展	124
4.1.2 XML 模式集合	63	5.10 SQL Server 关系图工具	125
4.1.3 创建、修改和删除 XML 模式集合	65	5.10.1 表	126
4.1.4 XML 数据类型方法	67	5.10.2 处理约束	128
4.1.5 施加超出模式集合范围的约束	72	5.11 关于日期列	129
4.2 提取 XML 格式的关系数据	73	5.12 小结	130
4.2.1 FOR XML 子句	73	第6章 核心存储和索引结构	131
4.2.2 OPENXML	98	6.1 SQL Server 存储	131
4.3 有关 XML 索引的提示	102	6.1.1 数据库	132
4.4 层次数据概述	102	6.1.2 文件	132
4.5 小结	103	6.1.3 区段	132
第5章 细心推敲，大胆设计	105	6.1.4 页	133
5.1 进一步了解规范化	105	6.1.5 行	135
5.1.1 入手点	106	6.1.6 全文目录	135
5.1.2 达到第三范式	107	6.1.7 文件流	136
5.1.3 其他的规范形式	107	6.2 理解索引	136
5.2 关系	108	6.2.1 “B”还是非“B”：B 树	137
5.3 图表	108	6.2.2 如何在 SQL Server 中访问数据	140
5.3.1 几种关系类型	109	6.2.3 索引类型和索引导航	141
5.3.2 实体框	109	6.3 创建、修改和删除索引	146
5.3.3 关系线	110	6.3.1 CREATE INDEX 语句	147
5.3.4 终止符	110	6.3.2 随约束隐含创建的索引	152
5.4 逻辑设计与物理设计	112	6.3.3 ALTER INDEX	152
5.4.1 逻辑模型的用途	112	6.3.4 DROP INDEX	154
5.4.2 逻辑模型的组成	113	6.4 明智地决定何时何地使用 何种索引	154
5.5 通过经典的 BLOB 处理基于 文件的信息	114	6.4.1 选择性	155
5.6 子类别	117	6.4.2 注意代价	155
5.6.1 子类别的类型	119	6.4.3 选择聚集索引	156
5.6.2 明确概念——实现子类别	119	6.4.4 列排序问题	158
5.6.3 子类别的物理实现	121	6.4.5 删除索引	158
5.6.4 通过子类别增加可扩展性	121	6.4.6 使用数据库引擎优化顾问	158
		6.5 维护索引	158

6.5.1 碎片	159	8.4 删除视图	215
6.5.2 检测碎片	159	8.5 审核: 显示现有代码	215
6.6 小结	166	8.6 保护代码: 加密视图	217
第 7 章 更高级的索引结构	167	8.7 关于模式绑定	218
7.1 XML 索引	167	8.8 使用 VIEW_METADATA 使自己的视图看起来像一个表	219
7.1.1 主 XML 索引	168	8.9 索引(物化)视图	219
7.1.2 辅助 XML 索引	169	8.10 分区视图	221
7.1.3 创建 XML 索引	170	8.11 小结	223
7.2 用户定义的数据类型	172	第 9 章 脚本和批处理	225
7.2.1 经典 UDT	172	9.1 脚本的基本概念	225
7.2.2 .NET UDT	173	9.2 批处理	226
7.2.3 表格式 UDT	173	9.2.1 批处理错误	228
7.2.4 删除用户定义的类型	176	9.2.2 使用批处理的时机	228
7.3 层次数据	176	9.3 SQLCMD	231
7.3.1 理解深度与输出	176	9.4 动态 SQL: 使用 EXEC 命令生成即时代码	232
7.3.2 HierarchyID 类型结构	177	9.5 流控制语句	237
7.3.3 处理 HierarchyID 值——HierarchyID 方法	178	9.5.1 IF...ELSE 语句	238
7.3.4 索引层次数据	188	9.5.2 CASE 语句	242
7.3.5 性能考虑	189	9.5.3 使用 WHILE 语句进行循环	245
7.4 空间数据	190	9.5.4 WAITFOR 语句	246
7.4.1 空间概念	190	9.5.5 TRY/CATCH 块	246
7.4.2 平面数据描述的实现——GEOMETRY 数据类型	194	9.6 小结	249
7.4.3 测量数据描述的实现——GEOGRAPHY 类型	199	第 10 章 高级编程	251
7.5 文件流	201	10.1 细看存储过程	251
7.6 启用文件流	202	10.1.1 输出参数	252
7.6.1 为数据库启用文件流	203	10.1.2 处理错误	254
7.6.2 创建一个启用文件流的表	203	10.2 表值参数(TVP)	264
7.6.3 在 T-SQL 中使用文件流	204	10.3 调试	267
7.6.4 在 .NET 中使用文件流	205	10.3.1 启动调试器	267
7.7 表压缩	205	10.3.2 调试器的组件	268
7.8 小结	206	10.3.3 启动后使用调试器	271
第 8 章 视图	209	10.4 理解 SQLCLR 及 SQL Server 中的 .NET 编程	273
8.1 回顾视图语法	209	10.4.1 程序简介	273
8.2 更复杂的视图	210	10.4.2 编译程序集	273
8.3 使用 T-SQL 编辑视图	214		

10.4.3 将程序集上载到 SQL Server 上	276	11.4.1 READ COMMITTED	318
10.4.4 创建基于程序集的存储过程	277	11.4.2 READ UNCOMMITTED	319
10.4.5 从程序集创建标量用户 定义函数	278	11.4.3 REPEATABLE READ	319
10.4.6 创建表值函数	281	11.4.4 SERIALIZABLE	319
10.5 创建聚集函数	284	11.4.5 SNAPSHOT	320
10.6 自定义数据类型	293	11.5 处理死锁(也称作“A 1205”)	320
10.6.1 从程序集创建自己的 数据类型	294	11.5.1 SQL Server 判断死锁的方式	321
10.6.2 访问复杂数据类型	294	11.5.2 如何选择死锁牺牲品	321
10.6.3 删除数据类型	295	11.5.3 避免死锁	321
10.7 小结	295	11.6 小结	323
第 11 章 事务和锁	297	第 12 章 触发器	325
11.1 事务	297	12.1 触发器的含义	326
11.1.1 BEGIN TRAN	298	12.1.1 ON	327
11.1.2 COMMIT TRAN	299	12.1.2 WITH ENCRYPTION	327
11.1.3 ROLLBACK TRAN	299	12.1.3 FOR/AFTER 与 INSTEAD OF 子句	327
11.1.4 SAVE TRAN	300	12.1.4 WITH APPEND	330
11.2 SQL Server 日志的工作方式	304	12.1.5 NOT FOR REPLICATION	330
11.2.1 使用 CHECKPOINT 命令	305	12.1.6 AS	330
11.2.2 在服务器正常关机时执行	305	12.2 为数据完整性规则使用 触发器	330
11.2.3 在更改数据库时执行	306	12.2.1 处理源自其他表的要求	331
11.2.4 在启用 Truncate on Checkpoint 选项时执行	306	12.2.2 使用触发器检查更新的差异	333
11.2.5 在恢复时间超过设置的恢复 间隔时执行	306	12.2.3 使用触发器实现自定义 错误消息	335
11.2.6 失败与恢复	306	12.3 触发器的其他常见用途	335
11.2.7 隐式事务	307	12.3.1 更新摘要信息	336
11.3 锁和并发	308	12.3.2 向反规范化的表输入数据 以用于报告	336
11.3.1 通过锁可以防止的问题	309	12.3.3 设置条件标志	336
11.3.2 可锁的资源	312	12.4 其他触发器问题	339
11.3.3 锁升级以及锁对性能的影响	312	12.4.1 嵌套触发器	339
11.3.4 锁模式	313	12.4.2 递归触发器	339
11.3.5 锁的兼容性	314	12.4.3 触发器调试	340
11.3.6 指定特定的锁类型——优化器 提示	315	12.4.4 触发器不妨碍架构的修改	340
11.4 设置隔离级别	318	12.4.5 不必删除就可以禁用触发器	340
		12.4.6 触发器的触发顺序	341
		12.5 INSTEAD OF 触发器	342

12.5.1	INSTEAD OF INSERT 触发器	344	第 14 章	Reporting Services	385
12.5.2	INSTEAD OF UPDATE 触发器	346	14.1	报表服务概述	385
12.5.3	INSTEAD OF DELETE 触发器	346	14.2	Reporting Services 入门	386
12.6	IF UPDATE()和 COLUMNS_UPDATED()	348	14.2.1	管理 Reporting Services 的工具	386
12.6.1	UPDATE()函数	348	14.2.2	访问 Reporting Services 的其他方法	387
12.6.2	COLUMNS_UPDATED() 函数	348	14.3	报表服务器项目	387
12.7	性能考虑	350	14.3.1	数据源	388
12.7.1	触发器不是主动的而 是被动的	350	14.3.2	使用报表向导	390
12.7.2	触发进程与触发器之间不存 在并发问题	350	14.3.3	编辑报表	394
12.7.3	简洁明了	351	14.3.4	参数化报表	397
12.7.4	选择索引时不要忘记触发器	351	14.3.5	提供参数值并控制其使用	398
12.7.5	不要尝试在触发器中回滚	351	14.3.6	添加图表	403
12.8	删除触发器	351	14.3.7	链接报表	404
12.9	小结	351	14.3.8	部署报表	405
第 13 章	SQL 游标	353	14.4	有关 RDL 的简注	405
13.1	游标的含义	353	14.5	小结	406
13.2	游标的生命期	354	第 15 章	bcp 和其他基本的大容量 操作	407
13.3	游标的类型和扩展的声明 语法	358	15.1	bcp 实用工具	408
13.3.1	作用域	359	15.1.1	bcp 语法	408
13.3.2	可滚动性	363	15.1.2	bcp 导入	411
13.3.3	游标类型	365	15.1.3	bcp 导出	415
13.3.4	并发性选项	375	15.2	格式化文件	416
13.3.5	检测游标类型转换: TYPE_WARNING	378	15.2.1	如果列不匹配	418
13.3.6	FOR <SELECT>	380	15.2.2	使用格式化文件	421
13.3.7	FOR UPDATE	380	15.2.3	尽量提高导入性能	421
13.4	在游标中导航: FETCH 语句	380	15.3	BULK INSERT	422
13.5	在游标中修改数据	381	15.4	OPENROWSET (BULK)	423
13.6	小结	383	15.4.1	ROWS_PER_BATCH	423
			15.4.2	SINGLE_BLOB、SINGLE_CLOB 或 SINGLE_NCLOB	423
			15.5	小结	424
			第 16 章	开始集成	425
			16.1	理解问题	425
			16.2	包的综述	426
			16.2.1	任务	427

16.2.2 主窗口	429	第 18 章 全文搜索	471
16.2.3 解决方案资源管理器	431	18.1 全文搜索的体系结构	472
16.2.4 属性窗口	431	18.2 设置全文索引和目录	473
16.3 创建简单的包	431	18.2.1 为数据库启用全文功能	474
16.4 执行包	435	18.2.2 创建、修改、删除和操作 全文目录	474
16.4.1 使用执行包实用工具	436	18.2.3 创建、修改、删除和操作 全文索引	476
16.4.2 在 Management Studio 中执行	438	18.2.4 针对旧语法的说明	481
16.5 小结	438	18.3 更多有关索引填充的内容	481
第 17 章 复制	439	18.4 全文查询语法	482
17.1 复制的基础知识	440	18.4.1 CONTAINS	483
17.1.1 计划复制时需要考虑的事项	440	18.4.2 FREETEXT	484
17.1.2 复制角色	442	18.4.3 CONTAINSTABLE	484
17.1.3 订阅	443	18.4.4 FREETEXTTABLE	486
17.1.4 订阅服务器的类型	443	18.4.5 处理短语	486
17.1.5 筛选数据	443	18.4.6 布尔操作	487
17.2 复制模型	443	18.4.7 邻近词	487
17.2.1 快照复制	444	18.4.8 权重	488
17.2.2 合并复制	447	18.4.9 屈折性	489
17.2.3 事务复制	449	18.5 停止词	489
17.2.4 立即更新的订阅服务器	452	18.6 小结	491
17.2.5 混合复制类型	452	第 19 章 安全性	493
17.3 复制的拓扑结构	453	19.1 安全性基础知识	494
17.3.1 简单模型	453	19.1.1 一个人, 一个登录名, 一个密码	494
17.3.2 混合模型	455	19.1.2 密码过期	495
17.4 制定复制计划	457	19.1.3 密码长度和组成	496
17.4.1 涉及的数据	457	19.1.4 尝试登录的次数	497
17.4.2 移动设备	458	19.1.5 用户和密码信息的存储	497
17.5 在 Management Studio 中设置 复制	458	19.2 安全性选项	498
17.5.1 为复制配置服务器	458	19.2.1 SQL Server 安全性	499
17.5.2 配置发布	462	19.2.2 创建和管理登录	499
17.5.3 通过 Management Studio 设置订阅服务器	465	19.2.3 Windows 身份验证	505
17.5.4 使用复制数据库	468	19.3 用户权限	505
17.6 复制管理对象(RMO)	469	19.3.1 授予访问特定数据库的权限	506
17.7 小结	470	19.3.2 授予访问数据库中对象 的权限	507

19.3.3 用户权限和语句级别的权限	512	20.8 注意细节问题	533
19.4 服务器和数据库角色	513	20.9 硬件考虑事项	533
19.4.1 服务器角色	514	20.9.1 独占式使用服务器	534
19.4.2 数据库角色	515	20.9.2 I/O 密集与 CPU 密集的对比	534
19.5 应用程序角色	518	20.9.3 OLTP 和 OLAP 的对比	538
19.5.1 创建应用程序角色	519	20.9.4 现场和非现场的对比	538
19.5.2 向应用程序角色添加权限	519	20.9.5 宕机的风险	539
19.5.3 使用应用程序角色	519	20.9.6 丢失数据	540
19.5.4 删除应用程序角色	521	20.9.7 性能就是全部吗	540
19.6 更高级的安全性	521	20.9.8 厂商支持	540
19.6.1 如何处理 guest 帐户	521	20.9.9 理想的系统	541
19.6.2 TCP/IP 端口设置	522	20.10 小结	541
19.6.3 不要使用 sa 帐户	522	第 21 章 性能优化工具	543
19.6.4 保持 xp_cmdshell 的隐秘性	522	21.1 优化时机(第二部分)	543
19.6.5 不要忘记把视图、存储过程 和 UDF 作为安全性工具	522	21.2 日常维护	544
19.7 证书和非对称密钥	523	21.3 故障排除	544
19.7.1 证书	524	21.3.1 数据收集器	545
19.7.2 非对称密钥	524	21.3.2 各种显示计划和 STATISTICS	545
19.7.3 数据库加密	524	21.3.3 数据库控制台命令(DBCC)	550
19.8 小结	524	21.3.4 动态管理视图	551
第 20 章 设计性能卓越的数据库	525	21.3.5 活动监视器	551
20.1 优化时机	526	21.3.6 SQL Server Profiler	554
20.2 选择索引	527	21.3.7 性能监视器(PerfMon)	556
20.3 客户端和服务端处理的 对比	528	21.4 小结	558
20.4 策略上的反规范化	529	第 22 章 管理	559
20.5 合理组织存储过程	529	22.1 计划作业	560
20.5.1 保持事务短小	529	22.1.1 创建操作员	561
20.5.2 尽可能使用限制性最少的 事务隔离级别	530	22.1.2 创建作业和任务	564
20.5.3 必要时部署多个解决方案	530	22.2 备份和恢复	580
20.5.4 尽可能避免使用游标	530	22.2.1 创建备份——也叫做 “转储”	580
20.6 使用临时表	531	22.2.2 恢复模型	585
20.6.1 使用临时表分解复杂问题	531	22.2.3 恢复	586
20.6.2 使用临时表以允许在工作 数据上创建索引	532	22.3 索引维护	590
20.7 及时更新代码	532	22.3.1 ALTER INDEX	590
		22.3.2 索引名	591
		22.3.3 表名或视图名	591

22.3.4	REBUILD	591
22.3.5	DISABLE	592
22.3.6	REORGANIZE	592
22.4	数据存档	593
22.5	PowerShell	593
22.5.1	尝试 PowerShell	594
22.5.2	在 PowerShell 中导航	598
22.5.3	关于 PowerShell 的最后一点说明	599
22.6	基于策略的管理	600
22.7	小结	600
第 23 章	SMO: SQL 管理对象	601
23.1	SQL Server 管理对象模型的发展历程	601
23.1.1	SQL 分布式管理对象	602
23.1.2	SQL 名称空间	602
23.1.3	Windows Management Instrumentation	602
23.1.4	SMO	603
23.2	SMO 对象模型	604
23.3	实例演练	605
23.3.1	开始	605
23.3.2	创建数据库	606
23.3.3	创建表	607
23.4	删除数据库	610
23.5	备份数据库	611
23.6	生成脚本	612
23.7	完整的代码	614
23.8	小结	618
第 24 章	数据仓库	619
24.1	考虑不同的需求	619
24.1.1	联机事务处理(OLTP)	620
24.1.2	联机分析处理(OLAP)	620
24.1.3	数据挖掘简介	621
24.1.4	OLTP 与 OLAP	621
24.2	维度数据库	622
24.2.1	事实表	622
24.2.2	维度表	623
24.2.3	星形和雪花模式	624
24.2.4	数据立方体	624
24.3	数据仓库的概念	625
24.3.1	数据仓库的特点	625
24.3.2	数据市场	626
24.4	SQL Server 的集成服务	627
24.4.1	数据验证	627
24.4.2	数据清洗	627
24.5	创建分析服务解决方案	627
24.6	访问立方体	633
24.7	小结	635
第 25 章	保证良好的连接性	637
附录 A	系统函数	639
附录 B	分析元数据	691
附录 C	基础知识	713

第 1 章

回顾 SQL Server 中的对象

如果你以前阅读过我所写的“高级”篇，那会发现我将沿袭与 *Professional SQL Server 2005 Programming* 一致的路线，只不过本书要比以前更“高级”一些。虽说如此，我仍希望稍提及一下所有的基本对象，以及对 SQL Server 2008 新提供的数据类型和对象作一下说明。

1.1 数据库的构成

数据库包括哪些内容？当然包括数据(不保存任何数据的数据库又有什么作用呢？)。我希望你目前(指准备学习高级篇时)已经意识到关系数据库管理系统(RDBMS, Relational Database Management System)不只是数据。目前高级的 RDBMS 不仅可以保存数据，而且可以管理数据，如限定输入到系统内的数据类型，还能方便地从系统中读取数据。如果仅仅是想将数据安全保存，则可使用任意数据存储系统。RDBMS 不仅可以按定义好的方式保存数据，还提供了更多的用途——SQL Server 2008 就是如此。改进后的对层次结构的支持意味着可以采用更自然的方式存储分层的数据，并仍然可以有效地访问它。新的“基于策略的管理”功能允许使用规则驱动的方法控制有关数据管理的多个要素。SQL Server 还通过 SQL Server 代理、Integration Services、Notification Services 以及逐步占据主流地位的 Reporting Services 等强大功能，提供了帮助用户数据与来自其他系统的数据进行自动交互的服务。

本章概述了 SQL Server 中使用的核心对象。对于现阶段的你来说，本章要讨论的大部分内容可能并不新鲜，因此本书也只有本章会对它们之间的联系作大体介绍。我将假定你对这里要讨论的大部分对象已有所熟悉，但本章的目的还是为了填补你的知识空白，以便更好地投入更高级内容的讨论中。

本章将概括介绍以下内容：

- 数据库对象
- 数据类型(包括 SQL Server 2008 提供的一些新类型)
- 确保数据完整性的其他数据库概念

1.2 数据库对象概述

RDBMS(如 SQL Server)包含许多对象。对于 Microsoft 所认定的可称为对象(或不能称为对象)的事物是否符合对象的标准定义,这里不做深究,但是对于 SQL Server 来说,它常包括以下重要的数据库对象:

- 数据库
- 事务日志
- 表
- 文件组
- 图表
- 视图
- 存储过程
- 用户自定义函数
- 索引
- CLR 程序集
- 报表
- 全文目录
- 用户自定义数据类型
- 角色
- 用户
- 加密密钥

这个列表并不完整,也没有按特定顺序排列,但它展示了 SQL Server 可管理的对象的广度。

1.2.1 数据库对象

在特定 SQL Server 中,数据库实际上是最高层对象(从技术角度来说,服务器本身可以看作一个对象,但从实际“编程”的角度看,不能称其为对象,因此此处不这样看)。在 SQL Server 中,大部分其他对象(但不是所有)为数据库对象的子对象。

提示:

如果你熟悉 SQL Server 的旧版本,你可能要问,“登录过程中发生了什么情况?远程服务器或 SQL 代理任务又发生了什么情况?”SQL Server 有其他一些对象(如前面所列的)用于支持数据库。除了链接服务器和 Integration Services 包之外,这些对象主要由数据库管理员管理,因此在设计和编程时通常不需要过多考虑。它们通过 SQL 管理对象(SQL Management Object, SMO)进行编程,不过这是很特殊的情形,此处不作考虑。第 26 章将对 SMO 作更全面的介绍。

数据库通常至少包括一组表对象,一般也包括其他一些对象,如存储过程和视图。存储过程和视图与保存在数据库表中的数据相关。

安装好的 SQL Server 第一次加载时包括以下 4 个系统数据库:

- master
- model
- msdb
- tempdb

安装了以上所有数据库的服务器才能正常运行(事实上,如果缺少其中任一数据库,服务器都无法运行)。根据安装选项的不同,安装的数据库也有所不同,可能包括以下示例数据库:

- AdventureWorks 或 AdventureWorks2008 (可从 codeplex.com 下载的示例数据库)
- AdventureWorksLT 或 AdventureWorksLT2008 (主要示例数据库的轻量级版本)
- AdventureWorksDW 或 AdventureWorksDW2008 (用于 Analysis Services 的示例)

除 Microsoft 支持的主要示例数据库外, 还可安装一些旧版示例数据库(通过网络搜索或是使用其他参考指导)。

- pubs
- Northwind

1. master 数据库

任一 SQL Server 都有 master 数据库, 而不管其是哪一版本或是定制安装。master 数据库保存一组特殊的表(系统表)用于跟踪整个系统。例如, 在服务器上新建数据库, 则在 master 数据库的 sysdatabases 表(如果你对 sysdatabases 中的数据感兴趣, 只可通过 sys.databases 元数据视图访问)中将加入该项。所有扩展的存储过程和系统存储过程都存储在 master 数据库中, 而不论该存储过程是用于哪一数据库。既然几乎所有描述服务器的信息都存储于 master 数据库, 显然, 该数据库对于系统至关重要, 不能删除它。

系统表(包括 master 数据库中的表)总在必要时才显得很重要。随着 Microsoft 不断地提供越来越多的其他方式来访问系统级信息, 直接使用系统表的情形变得越来越少。

注意:

我过去经常使用系统表, 如今却不常使用。

注意:

Microsoft 在 7.0 版本(1998 年左右)之前就建议不用使用系统表。他们不能绝对保证不同版本的 master 数据库的兼容性——事实上, master 数据库几乎都被更改过。对 master 数据库中的对象进行升级是最大的过错。要记住, 用任何方式对这些表进行改动都会让 SQL Server 失去作用。我这样做曾经挽救过一个系统, 也曾经使一个系统瘫痪, 我不希望在职业生涯中冒如此大的风险。

注意:

Microsoft 提供了一些替代方法(如系统函数、系统存储过程、information_schema 视图和一组系统元数据函数)来检索存储在系统表中的大量元数据。应该使用这些替代方法。

2. model 数据库

顾名思义, model 数据库是指可以基于该模型得到一个副本。model 数据库是新建数据库的模板。也就是说, 如果想要改变新建标准数据库的样式, 则可以根据需要更改 model 数据库。例如, 可以向新建的每一数据库中加入一组审计表。也可以将一些用户组复制到系统新建的每一个数据库中。注意: 由于 model 数据库用作其他任意数据库的模板, 因此系统中必须保留该数据库, 禁止删除它。

在更改 model 数据库时要注意的是: 首先, 任意新建的数据库至少要比 model 数据库大。也

就是说, 如果将 model 数据库大小更改到 100MB, 就不能新建比 100MB 小的数据库。另外, 更改 model 数据库还会引起其他问题。同样, 对于绝大部分 SQL Server 系统的安装, 强烈建议不要对其进行修改。

3. msdb 数据库

msdb 数据库是 SQL 代理进程保存任意系统任务的地方。如果计划对一个数据库每夜进行备份, 则在 msdb 数据库中有一个记录项。要调度存储过程一次执行, 则在 msdb 数据库中也会有一个记录项。SQL Server 中的其他大部分子系统以类似的方式使用 msdb。SSIS 包和基于策略的管理的定义就是使用 msdb 的其他进程的示例。

4. tempdb 数据库

tempdb 数据库是服务器的一个主要工作区。在执行复杂或者大型的查询操作时, 如果 SQL Server 需要创建一些中间表来完成, 那它就在 tempdb 数据库中进行。在创建临时表时, 即使你是在当前数据库中创建这些表的, 但实际也是在 tempdb 数据库中创建的。只要需要临时保存数据, 就很可能是将数据保存在 tempdb 数据库中。

tempdb 数据库与其他任意数据库大相径庭。不仅数据库中的对象是临时的, 连数据库本身也是临时的。每次启动 SQL Server 时, tempdb 数据库是系统中唯一完全重建的数据库。

注意:

从技术角度讲, 可以在 tempdb 数据库中创建自己的对象, 但我强烈反对进行该操作。可以在系统中你有权使用的任意数据库中创建临时对象, 该对象也会存储在 tempdb 数据库中。直接在 tempdb 数据库中创建对象没有任何意义, 只会增加数据库之间的混乱程度。这也是不要尝试做的一件事情。

注意:

每次重启 SQL Server 时, 都会删除并重建 tempdb 数据库。

5. AdventureWorks/AdventureWorks2008 数据库

在这些数据库出现之前, SQL Server 就包括了其他示例数据库。过去的示例数据库有不足之处, 如它们包含了一些不良的设计实践(至于 AdventureWorks 数据库是否有同样的问题, 我不打算作评论, 但 AdventureWorks 正试图解决这一问题)。此外, 过去的例子过于简单, 只是集中阐明某些数据库概念, 而不是将 SQL Server 作为一个产品或将数据库视为一个整体。

在 Yukon(这是现在的 SQL Server 2005 的内部代码)发展之初, Microsoft 就想要一个更加健壮的示例数据库, 使其尽可能作为其产品的一个示例。AdventureWorks 数据库就是这一想法的实际结果。尽管该数据库对于初学者过于复杂, 但是其作为示例数据库确实是一大杰作。虽然该数据库不包括所有的内容, 但还是一个相当完整的示例, 具有更接近实际的数据量、复杂的结构和用

于展示产品的大部分功能的部件。从这一方面看,该数据库是非常不错的。AdventureWorks 2008 是最初的 AdventureWorks 数据库自然演变的结果,它修改和扩展了相应的模型,以便使用 SQL Server 2008 中的新功能。

本书中将把 AdventureWorks2008 作为核心示例数据库使用。

6. AdventureWorksLT/AdventureWorksLT2008 数据库

这里的 LT 表示轻量级(lite)。它只是 AdventureWorks 数据库完整版的极小一部分。它提供了更简化的示例集,便于理解基本概念和做简单练习。尽管我并不知晓推出这一新示例集的真正原因,但我猜测是为了取代较老的 Northwind 和 Pubs 示例集(相较于 AdventureWorks 示例集,许多培训者更喜欢采用 Northwind 和 Pubs 示例集,因为 AdventureWorks 数据库对于初级培训来说通常过于复杂和麻烦)。不过,我在最近听闻有对 Northwind 示例作一些更新和开发的计划,因此可能 Microsoft 并不准备完全舍弃这些老示例集。

7. AdventureWorksDW/AdventureWorksDW2008 数据库

AdventureWorksDW2008 数据库是 Analysis Services(分析服务)的示例数据库(DW 表示数据仓库,大部分 Analysis Services 项目建立在数据仓库的基础上)。关于示例数据库最棒的是,Microsoft 将分析示例与事务示例数据库联系在一起,以提供展示两者协同运行的完整示例数据库。

在回顾 OLAP 概念和讨论 Analysis Services 时才会用到该数据库。比较一下两个数据库之间的不同。它们服务于同一个虚构的公司,但是它们具有不同的目的。

1.2.2 事务日志

如果你学习 SQL Server 已有一段时间,那对日志应该会有一个基本的了解。虽说如此,日志还是 SQL Server 中最可能引起误解的一个对象。尽管数据要从数据库文件读取,但实际上,任意数据库的更改起初不在数据库中发生,而是被连续写入到事务日志(transaction log)中。在后面的某个时间点上,数据库发出检查点(checkpoint),就是在该时间点,日志中所有的更改才被传到实际的数据库文件中。

数据库是随机访问的,但是日志却是连续的。数据库文件的随机访问特点允许快速访问,而日志的连续特征又使得发生的事件可按照顺序来跟踪。日志不断积累已经提交的更改,而服务器将在后面的某个时间将修改写入到物理数据库文件。

第 11 章中将深入讨论如何将更改写入日志,现在要记住的是:日志是数据进入磁盘上的最先位置,且在稍后的时刻被传到实际的数据库。需要数据库文件和事务日志一起来完成数据库功能。

1.2.3 最基本的数据库对象:表

数据库包括许多对象,但最重要的对象非表莫属。表由域数据(列)和实体数据(行)构成。数据

库中的实际数据都存储在表中。每个表的定义也包含了描述表中包含数据的类型的元数据(metadata, 数据描述信息)。每一列具有该列可存储什么数据类型的一组规则。表中任一列中的数据违反了规则, 系统会拒绝插入一行, 或拒绝对已有行进行更新, 或者禁止删除行。

表可以有与之相关联的附加对象——这些对象只在特定表的结构中(或在某些极少见的情形中, 如视图)存在。下面就分别来看看这些对象。

1. 索引

索引是仅在特定表或视图架构内存在的对象。索引的功能非常类似于百科全书中的目录。索引中有以某一特定方式排序的查找(或键)值, 使用索引是快速查找数据库中实际信息的一种方法。

利用索引可以加速信息的查找。索引分为两类:

- **聚集索引**——每个表只能有一个聚集索引。如果是聚集索引, 其含义为: 聚集索引对应的表按照其索引进行物理排序。如果为一本百科全书做索引, 则聚集索引是书的页码; 按页码顺序保存百科全书中的信息。
- **非聚集索引**——每个表可以有多个非聚集索引。非聚集索引的含义与你在平时听到的“索引”的含义更接近。非聚集索引是让你找到数据的一些其他值。对于百科全书, 非聚集索引指的是百科全书后面的关键字目录。

注意, 具有索引的视图(索引视图)必须在有非聚集索引之前至少有一个聚集索引。

2. 触发器

触发器是存在于表架构内的对象。触发器是在表操作时(如进行插入、更新或删除等)自动执行的一段逻辑代码。触发器有多种用途, 但主要用于在插入时复制数据或更新时检查数据, 确保数据满足相应条件。

有种特殊类型的触发器——即事前触发器(before trigger)——可与视图相关联。第 12 章将深入介绍它。

3. 约束

约束是仅在表的范围中存在的另一对象。顾名思义, 约束就是限制表中数据满足某种条件。约束在某些方面类似于触发器, 尽可能解决数据完整性问题。但它们也有所不同, 各自具有不同的优点。

与触发器和索引不同的是, 约束只能与表相关联(而不能与视图相关联)。

1.2.4 模式

模式(schema)为数据库和其所包含的其他对象之间提供了中间名称空间。任意数据库的默认模式都是 dbo(表示数据库所有者)。每个用户都有一个默认模式, SQL Server 将自动在用户的默认

模式中搜索对象。不过,如果对象所在的名称空间非用户默认的,那必须以<schema name>.<object name>这种两部分的形式来引用该对象。

注意:

模式取代了 SQL Server 以前版本中“所有者”的概念。尽管 Microsoft 如今在大力宣传其作用(其思想是通过模式引用一组表,而不是将它们一一列出),但我还是表示怀疑。简而言之,我认为它们带来的问题要多于解决的问题,所以我通常建议不要使用它(也有例外,但是在特殊情况下)。

1.2.5 文件组

默认情况下,数据库中所有的表及其他对象(日志除外)要存储在一个文件中。该文件是一些所谓的主要文件组的成员。不过,并不一定要坚持这样的配置。

SQL Server 允许定义的辅助文件数量稍高于 32 000(如果需要定义更多的辅助文件,则可能不是 SQL Server 本身的问题)。这些辅助文件可加入到主要文件组中,或者作为一个或多个辅助文件组的一部分创建。尽管仅能有一个主要文件组(该文件组实际被称为 Primary),但可以有多达 255 个辅助文件组。通过 CREATE DATABASE 或 ALTER DATABASE 命令的选项可创建辅助文件组。

文件组的概念主要是允许用类似分段的形式管理数据的物理存储。可将文件备份到一个指定的文件组中(而不是整个数据库)。可使用单独的文件使数据分布到多个物理存储设备中(可能提供更大的 I/O 带宽)。

1.2.6 图表

数据库图表是数据库设计的可视表示,它包括了各种表、每一张表的列名以及表之间的关系。开发人员应该听过实体/关系图(entity-relationship diagram, ERD)。在 ERD 中,数据库被分成两部分:实体(如“供应商”和“产品”)和关系(如“供应”和“购买”)。

提示:

SQL Server 2008 内含的数据库设计工具保留的很少。实际上工具使用的图表方法并未遵循 ER 图表中公认的标准。但这些图表工具事实上提供了所有的“必要的”内容,因此也可从它们入手。

如图 1-1 所示的图表表示 AdventureWorks 数据库中的各表之间的关系。该图表也描述了数据库的许多其他特征(如果你是第一次接触它,可能会注意不到)。注意小钥匙图标和无穷大符号。这些符号描述了两张表之间的关系。

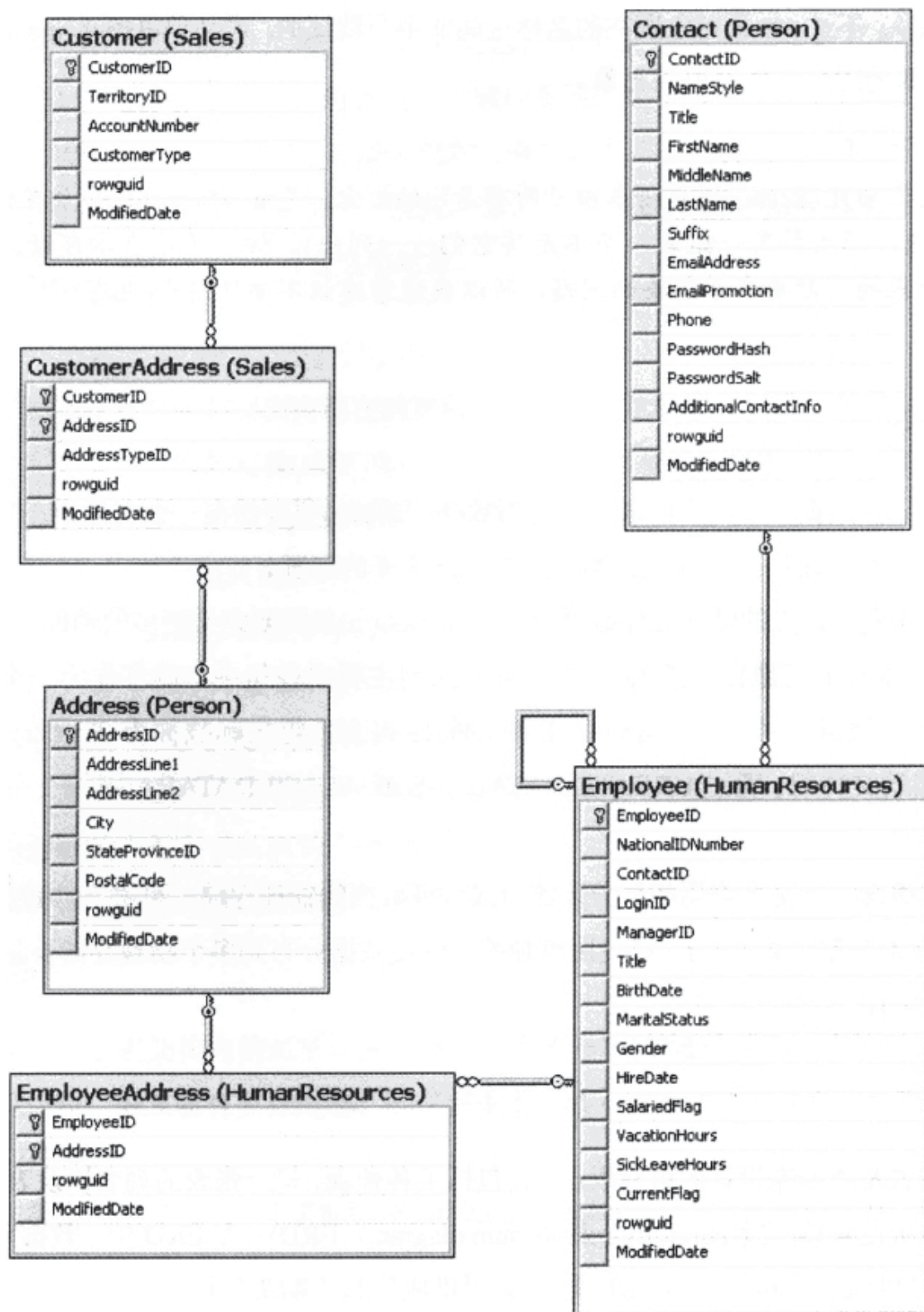


图 1-1

1.2.7 视图

视图是一种虚拟表。除了视图本身不包含任意数据外，视图的使用基本与表的使用类似。事实上视图仅仅是存储在表中的数据预先设计好的映射和表示。视图以查询形式存储在数据库中。这种查询需要从一个或多个表中获取一些列(但不一定是所有列)的数据。为了显示视图中的数据，读取的数据可能要或不要(取决于视图的定义)满足特定标准。对于大部分视图来说，这有两方面的原因：安全和易于使用。使用视图可以控制用户有权查看的内容，因此，如果某些人只能看到表的一部分数据(如工资详细资料)，则可以创建仅包括任何用户都可以访问的这两列数据的视图。此外，可以裁剪视图，使得用户不必搜索所有不必要的信息。

除了这个最基本的应用之外，还可以创建所谓的索引视图(indexed view)。索引视图除了针对视图创建索引之外，别的与其他视图相同。这一点对使用性能有一定影响(包括正面影响和负面影响)：

- 对于引用多个表的视图来说，执行索引视图会快很多，因为它预先构造了表之间的联结。
- 视图中执行的聚合(aggregation)是预先计算好，并作为索引的一部分存储；因此聚合只要执行一次(在插入或更新行时)，然后就可从索引信息中直接读入。
- 由于视图上的索引必须及时更新，因此插入和删除需要更多的系统开销；如果更新影响索引的键列，则更新还需要更大的系统开销。

本书第 8 章将深入探讨这些性能问题以及视图的其他特殊用途。

提示：

尽管创建索引视图的代码在所有版本中都能起作用，但只有在企业版中使用时，查询优化器才考虑它。

1.2.8 存储过程

存储过程(或 sprocs)是 SQL Server 编程功能的基础。存储过程通常是组成一个逻辑单元的 Transact-SQL(用于查询 Microsoft SQL Server 的语言)语句的有序集合。存储过程允许使用变量和参数，也可使用选择和循环结构。与发送单条语句到服务器相比，使用存储过程具有以下几个优点：

- 调用存储过程不使用长文本串而使用短存储过程名，可减少运行存储过程中的代码所要的网络流量。
- 预先优化和预编译，能将存储过程每次运行的时间缩短一点。
- 通常考虑安全原因，或仅仅是简化数据库的复杂性，将过程封装。
- 可从其他存储过程调用，使得它们在一定意义上重用。

此外，可利用任何 .NET 语言创建程序集，并向存储过程中添加 T-SQL 之外的程序结构。

1.2.9 用户自定义函数

用户自定义函数(UDF)与存储过程非常相似，不同之处有以下几点：

- 返回值的数据类型包括大部分 SQL Server 数据类型。不包括的返回类型是：text、ntext、image、cursor 和 timestamp。
- 没有“副作用”，即用户自定义函数不能完成在其范围之外的功能，如更改表、发送电子邮件或更改系统或数据库参数。

UDF 类似于标准编程语言(如 VB.NET 或 C++)中使用的函数。函数可以有多个输入变量，可以有一个返回值。但 SQL Server 的 UDF 与许多过程语言中的函数也有所不同，传送到函数的所有变量都是按值传递。这与 VB 中按引用传递变量或是 C++中指针的传递不同。但是 UDF 可以返回一种特殊的数据类型——表。第 10 章将深入讲述该部分内容。

1.2.10 用户和角色

用户和角色关系密切。用户(user)几乎等价于登录名。简言之,该对象表示登录 SQL Server 的用户的标识符。登录 SQL Server 的任何人都映射(直接或间接映射,这取决于使用的安全模型)到一个用户。用户属于一个或多个角色(role)。SQL Server 中可以直接赋予用户或角色执行某种操作的权限,一个或多个用户可属于同一角色。

1.2.11 规则

规则和约束都是限制插入到表中的数据类型的信息。如果更新或插入记录违反规则,则插入或更新操作被拒绝。此外,规则可用于定义用户自定义数据类型上的限制。与规则不同,约束本身不是实际对象,只是描述特定表的元数据片段。

注意:

尽管 Microsoft 没有说明将从哪一个版本开始,但他们已警告说后续版中将删除规则。只有在向后兼容性问题才考虑规则,在新的开发中应该避免使用。而且,应考虑逐步淘汰当前数据库使用的规则。

1.2.12 默认值

SQL Server 中有两种类型的默认值,包括对象本身的默认值,以及描述表中特定列的元数据的默认值(非真正对象)。和约束与规则类似,规则是对象,而约束不是对象,是元数据。当插入一条记录时,如果没有提供该列的值,且该列具有其默认值,则自动插入默认值。

注意:

与规则十分相似,作为其自身对象的默认值应视为遗留对象,在新开发中应避免使用,并从已有代码中删除。不过,默认值约束仍非常有效。

1.2.13 用户自定义数据类型

用户自定义数据类型是系统定义数据类型的扩展。其潜力几乎是无穷无尽的,但要注意向后兼容性。虽然 SQL Server 2000 及其更早版本也有用户自定义数据类型,但是它们仅限于现有数据类型的不同筛选。自 SQL Server 2005 后,可以将 .NET 程序集绑定到你自己的数据类型上,即能够定义一种数据类型,用于在合理范围内存储 .NET 对象可以存储的任意内容。

提示:

要谨慎使用自定义数据类型!你所使用的数据类型对于数据及数据存储来说非常重要。虽然能够定义任意类型的功能很优秀,但这几乎肯定会带来大量性能开销,也很可能需要付出安全方面的代价。所以要考虑清楚,只定义确实需要的自定义数据类型,然后再进行多次测试!

1.2.14 全文目录

全文目录是数据映射,用于加速对启用了全文搜索功能的列中特定文本块的搜索。尽管这些

对象与映射的表和列关系紧密，但它们还是单独的对象，当数据库发生改变时，它们不一定会自动更新(默认是自动更新，但可改为手动更新)。

1.3 SQL Server 数据类型

与前面的一两个版本相比，这个版本在数据类型方面发生了较大的变化。SQL Server 2005 开始改变与 BLOB 相关的数据类型(如 text 和 ntext 变为 varchar(max)和 nvarchar(max)，image 变成 varbinary(max))。现在，SQL Server 2008 添加了一些新的与时间和日期相关的数据类型，以及处理分层数据的特殊数据类型。

提示：

由于本书面向开发人员，而几乎不存在还不清楚数据类型的开发人员，所以这里假设你已经清楚数据类型的作用，而只需了解 SQL Server 数据类型的具体细节。

SQL Server 2008 所有自带的数据类型如表 1-1 所示。

表 1-1

数据类型名称	类 型	长度(以字节为单位)	数 据 特 点
Bit	整型	1	这个数据的大小容易让人误解。表中的第 1 个 bit 数据类型占 1 个字节，其余 7 个 bit 数据类型使用同一个字节。如果允许使用 null，则会多占用一个字节
Bigint	整型	8	可处理日常用到的越来越大的数，其取值范围为 $-2^{63} \sim 2^{63} - 1$
Int	整型	4	取值范围为 $-2\,147\,483\,648 \sim 2\,147\,483\,647$
SmallInt	整型	2	取值范围为 $-32\,768 \sim 32\,767$
TinyInt	整型	1	取值范围为 $0 \sim 255$
Decimal 或 Numeric	小数/数字型	可变	固定精度，取值范围为 $-10^{38} - 1 \sim 10^{38} - 1$ 。两者含义相同
Money	货币型	8	货币单位，取值范围为 $-2^{63} \sim 2^{63}$ ，精确到 4 个小数位。注意货币单位可以是任意货币，不限于美元
SmallMoney	货币型	4	货币单位，取值范围为 $-214\,748.3648 \sim +214\,748.3647$
Float (也是 ANSI Real 的同义词)	近似数字型	可变	由一参数(如 Float(20))决定其长度与精度。注意参数值表示位数，不是字节数。取值范围为 $-1.79E + 308 \sim 1.79E + 308$
DateTime	日期/时间型	8	日期与时间，取值范围为 1753 年 1 月 1 日到 9999 年 12 月 31 日，精确到 0.03 秒

(续表)

数据类型名称	类 型	长度(以字节 为单位)	数 据 特 点
DateTime2	日期/时间型	可变(6~8)	新扩展的 DateTime 数据类型。支持更大的日期范围和更高的时间部分精度(精确到 100 毫秒)。和 DateTime 一样, 它不包含时区信息, 但与 .NET DateTime 数据类型相对应
SmallDateTime	日期/时间型	4	日期与时间, 取值范围为 1900 年 1 月 1 日到 2079 年 6 月 6 日, 精确到分钟
DateTimeOffset	日期/时间型	可变(8~10)	类似于 DateTime 数据类型, 但有一个相对于 UTC 时间的 -14:00~+14:00 的偏移量。时间在内部存储为 UTC 时间, 任何比较、排序或索引将基于该统一的时区
Date	日期/时间型	3	只存储 Gregorian 日历定义的 0001 年 1 月 1 日到 9999 年 12 月 31 日的日期数据。采取 ANSI 标准日期格式 (YYYY-MM-DD), 但会从其他一些格式隐式转换
Time	日期/时间型	可变(3~5)	只存储 Time 数据, 使用用户可选的精度, 粒度可以为 100 纳秒(默认)
Cursor	特殊数字型	1	指向游标的指针, 尽管指针只占用一个字节, 记住组成实际游标的结果集也占用内存, 占用内存的大小取决于结果集
Timestamp/ rowversion	特殊数字型 (二进制)	8	给定数据库的唯一特定值。即使 UPDATE 语句没有引用 timestamp 列(时间标记), 但其值在插入或更新记录的时间自动由数据库本身设定(不允许直接更新 timestamp 字段)
UniqueIdentifier	特殊数字型 (二进制)	16	特殊的全局唯一标识符(GUID), 必须保证在内存空间和内存内的唯一
Char	字符型	可变	定长字符数据。比设定长度短时使用空格填充, 为非 Unicode 数据, 最大长度为 8 000 字符
VarChar	字符型	可变	变长字符数据。比设定长度短时不使用空格填充, 为非 Unicode 数据。允许最大长度为 8 000 字符, 但如果使用 max 关键字, 则表示它是非常大的字符字段(数据长度可达 2^{31} 字节)
Text	字符型	可变	从 SQL Server 2005 起用作向后兼容。可使用 varchar(max) 代替
NChar	Unicode 型	可变	定长 Unicode 字符数据。比设定长度短时使用空格填充。最大长度为 4 000 字符
NVarChar	Unicode 型	可变	变长 Unicode 字符数据。比设定长度短时不使用空格填充。允许最大长度为 4 000 字符, 但如果使用 max 关键字, 则表示它是非常大的字符字段(数据长度可达 2^{31} 字节)

(续表)

数据类型名称	类 型	长度(以字节为单位)	数 据 特 点
Ntext	Unicode 型	可变	变长 Unicode 字符数据。类似 Text 数据类型, 仅用作向后兼容。可使用 <code>nvarchar(max)</code> 代替
Binary	二进制	可变	定长二进制数, 最大长度为 8 000 字节
VarBinary	二进制	可变	变长二进制数, 最大特定长度为 8 000 字节, 但如果使用 <code>max</code> 关键字, 则指示它实际上是一个 LOB(大对象) 字段(数据长可达 2^{31} 字节)
Image	二进制	可变	从 SQL Server 2005 起用作向后兼容。可使用 <code>varbinary(max)</code> 代替
Table	其他	特殊	主要用于结果集, 通常作为用户自定义函数(UDF)的结果返回或作为存储过程的参数。在表的定义中不作为可用的数据类型(不能嵌套表)
HierarchyID	其他	特殊	维护层次结构位置信息的特殊数据类型。提供建立层次结构所需的特殊功能。允许作深度、父/子关系和索引比较。实际尺寸随层次结构中节点数和平均深度而变
Sql_variant	其他	特殊	与 VB 和 C++ 中的 Variant 有些细微的关联。其实质是用于保存其中的其他大多数 SQL Server 数据类型的容器。当一个列或函数需要处理多种数据类型时可使用这种数据类型。与 VB 不同, 使用这种数据类型要将按顺序其显式转换为更具体的数据类型
XML	字符型	可变	定义一个字符字段用作 XML 数据。用于针对 XML 模式的数据验证和使用特殊的面向 XML 的函数

注意, 与 .NET 数据类型的兼容性要比以前更强。例如, 新的 `date` 和 `time` 数据类型在 .NET 中是兼容的; 而且相比于以前的 `datetime` 数据类型, 新的 `datetime2` 数据类型与 .NET 的兼容性更好。

注意:

遗憾的是, SQL Server 中没有无符号数值数据类型的概念。如果要用到比有符号数据类型允许的值更大的数, 应考虑使用更大的有符号数据类型。如果需要防止使用负数, 可以使用 `CHECK` 约束, 限制有效数据为大于或等于 0。

通常, SQL Server 数据类型与其他现代编程语言中的用法类似。数的相加为和, 而字符串的相加为字符串的连接。如果混合了不同类型变量或字段的使用或赋值, 那许多数据类型会隐式(或自动)转换。大部分其他数据类型可以显式转换(只要具体指定想要转换成何种数据类型)。有些数据类型之间不能相互转换。图 1-2 表示了所有可能的不同转换。

	binary	varbinary	char	varchar	nchar	nvarchar	datetime	smalldatetime	date	time	datetimeoffset	datetime2	decimal	numeric	float	real	bigint	int(INT4)	smallint(INT2)	tinyint(INT1)	money	smallmoney	bit	timestamp	uniqueidentifier	image	ntext	text	sql_variant	xml	CLR UDT	hierarchyid
binary	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
varbinary	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
char	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
varchar	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
nchar	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
nvarchar	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
datetime	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
smalldatetime	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
date	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
time	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
datetimeoffset	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
datetime2	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
decimal	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
numeric	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
float	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
real	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
bigint	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
int(INT4)	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
smallint(INT2)	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
tinyint(INT1)	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
money	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
smallmoney	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
bit	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
timestamp	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
uniqueidentifier	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
image	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
ntext	○	○	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
text	○	○	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
sql_variant	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
xml	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
CLR UDT	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
hierarchyid	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●

● 显式转换

● 隐式转换

○ 不允许转换

* 要求显式执行强制转换,防止隐式转换中可能出现的精度或小数位数丢失。

● 只有源或目标是非类型化 XML 时才支持 XML 数据类型间的隐式转换。否则,必须显式转换。

图 1-2

简而言之,SQL Server 中的数据类型与其他编程环境中的数据类型的功能大多相同。数据类型可以避免出现编程故障,确保提供的数据和定义的数据类型一致(记住日期 1/1/1980 与数字 1/1/1980 不同),而且确保执行所希望的操作。

NULL 数据

假设表中有一行的某一列没有任何数据,也就是你不知道该列的值。比如,让一记录保存某一年的公司业绩信息,假设字段之一为相对于前一年增长的公司业绩百分比,但是在数据库中沒有上一年的业绩信息记录,则可能想在 PercentGrowth 列中输入 0。那么这样做是否会提供正确的信息呢?不知道的人可能认为 0 表示增长 0%。而实际情况是不知道上一年的业绩信息。

不确定的值可设置为 NULL。几乎在我每一次讲授编程课时,都至少有一个学生问我怎么定义 NULL 值。这是一个难题,因为根据定义, NULL 值也就是不确定的值,可能是 1,也可能是 347,还可能是-294 等。简而言之, NULL 值是未定义,或者是不可用的值。

1.4 SQL Server 对象标识符

至此，本书罗列了 SQL Server 中与对象相关的各类信息。下面进一步分析 SQL Server 中对象的命名。

1.4.1 需要命名的对象

基本上 SQL Server 中的所有对象都需要命名，下面是部分命名对象的列表。

- 存储过程(Stored procedure)
- 视图(View)
- 默认值(Default)
- 触发器(Triiger)
- 用户自定义函数(User-defined Functions)
- 全文目录(Full-text catalog)
- 模式(Schema)
- 约束(Constraint)
- 服务器(Server)
- 用户自定义类型(User-defined Type)
- 表(Table)
- 规则(Rule)
- 索引(Indexe)
- 数据库(Database)
- 登录名(Login)
- 文件(File)
- 列(Column)
- 文件组(Filegroup)
- 角色(Role)

这个列表还可不断扩展。除了行(行不是真正的对象)之外的所有对象都具有名称。命名的技巧是使得名称有用且符合实际需要。

1.4.2 命名规则

如本章前面所述，SQL Server 中的命名规则非常随意，命名允许名称中内嵌空格，甚至是关键字。但是命名过于自由也容易带来麻烦。

主要的命名规则如下：

- 对象的名称必须以 Unicode 3.2 规范定义的任意字母开头。包括大部分西方人使用的字母，即 A-Z 和 a-z。“A”与“a”是否相同取决于服务器配置的方式，但不管 A 或 a 作为对象名的开头都符合规则。名称中除了开头是字母之外其他几乎可以是任意字符。
- 正常对象的名称可多达 128 个字符，而临时对象的名称可多达 116 个字符。
- 与 SQL Server 关键字相同或包含内嵌空格的名称必须使用双引号(“”)或方括号([])。哪一个视为关键字取决于为数据库设置的兼容级别。

提示：

只有设置了 QUOTED_IDENTIFIER 为 ON，双引号才可以作为列名中的分界符。使用方括号([])避免了用户进行其他错误设置，但不像双引号那样具有平台中立性。我强烈建议你不要将 QUOTED_IDENTIFIER 设置为 ON，因为它会给索引视图带来问题。

这些规则通常是指标识符以及 SQL Server 中的任意对象的命名规则。特定对象类型存在其他规则。

注意:

尽量避免在名称中使用 SQL Server 关键字或嵌入空格! 虽然只要加以限定, 这两者在技术上都是合法的, 但这样命名等于自找麻烦。

1.5 小结

与其他大部分编程环境一样, 数据库数据也具有数据类型。在 SQL Server 中进行的大多数操作都要考虑到数据类型。尽管在 SQL Server 2008 中, 基本的对象几乎没有什么变化, 但新增了几种数据类型。记着温习这些新类型(日期和时间方面以及层次数据支持方面)。复习一下可用的数据库数据类型, 比较这些数据类型与其他你熟悉的任何编程环境中的数据类型是如何相对应的。

考虑一下 SQL Server 2008 中可用的对象。尽管在学习本书之前, 你可能就已经熟悉了表以及视图和脚本的概念(如果不是, 那要阅读一下《SQL Server 2008 编程入门经典》(由清华大学出版社引进并出版)这本书), 但我希望你认识到仅是少量表和一两个存储过程不能构成真正的数据库。目前的 RDBMS 之所以如此强大, 是因为众多数据库对象使你能将与数据相关的功能和业务规则连同数据放到数据库中。



第 2 章

工 具

如果你已经熟悉了 SQL Server Management Studio(这说明你以前在 SQL Server 2005 环境下工作或是已使用 SQL Server 2008), 那对于本章来说, 你只需对其新的内容浏览一下。如果是这样的话, 你可能希望详细了解一下一些较不常用的工具, 如配置管理器(Configuration Manager)和 Net-Libraries(通常写作 NetLibs)。如果你是位新手, 我建议你先学习《SQL Server 2008 编程入门经典》(由清华大学出版社引进并出版)一书, 该书对基础知识作了更详细的介绍。对于本书来说, 在前几章包含这些内容主要是为了提供一个参考, 同时简单介绍一下一些新内容。

接下来, 我们将讲述工具集。如果你之前使用的是 SQL Server 2000 或更早的版本, 那要对本章内容特别留意。在 SQL Server 2005 中, 工具集发生了很大的变化。SQL Server 2008 在 SQL Server 2005 的基础上添加了一些新的工具并作了些调整。

对于像我这样固执的人来说, 新的工具让人有些不知所措。但对于 SQL Server 的新用户来说, 我认为设计团队已经基本达到了 SQL Server 2005 追求的大大简化工具的设计目标。总的来说, 几乎所有工具都不用费劲去找, 大部分工具集作了合理的组合。

本章要讲述的工具包括:

- SQL Server 联机丛书
- SQL Server 配置管理器
- SQL Server Management Studio
- SQL Server Business Intelligence Development Studio
- SQL Server 集成服务(SSIS), 包括导入/导出向导(Import/Export Wizard)
- Reporting Services
- Bulk Copy Program(bcp, 大容量复制程序)
- Profiler
- Sqlcmd

2.1 联机丛书

联机丛书也被视为工具吗? 本书确实是这样认为的。无论你学习本书或者其他有关 SQL Server 的

书多少遍，你都不可能记住 SQL Server 中所有需要记住的内容。SQL Server 是我经常使用的软件产品之一，但我也无法记住所有内容。联机丛书是 SQL Server 中最重要的工具之一。

提示：

这里有条建议：不要试图记住所有的内容。记住看过的内容是有可能的。记住那些编程的基础知识。记住每天要使用的内容。然后记得为其余内容建立一个参考资料库(从本书开始)。

联机丛书的一切都非常符合我们的使用习惯，因此，这里不准备对如何使用帮助系统作详细说明。只要知道，无论使用哪台计算机，SQL Server 联机丛书都是可快速得到的参考手册。联机丛书还有一个好处，与书面的文档资料相比，它能提供更新的信息。

注意：

从技术上来说，不是每一个系统都安装有联机丛书(BOL)。因为可在安装时手动取消选择 BOL。即使在磁盘空间不多的情况下，我们也强烈推荐安装 BOL。考虑一下如今磁盘每 MB 的成本，BOL 其实不会占用太多的空间，但在运行 SQL Server 时，却可以快速找到参考，从而能节省大量时间(在计算机上安装联机丛书约占用 100MB 磁盘空间)。

2.2 SQL Server 配置管理器

配置进行数据库访问的计算机管理员要经常使用本工具，但理解本工具的用途也很重要。

SQL Server 配置管理器只是将之前版本中的多个设置工具集组成为一个设置工具。配置管理器的管理工作分为两部分：

- 服务管理
- 网络配置

2.2.1 服务管理

这里可管理的服务包括：

- 集成服务(Integration Services)——支持 Integration Services 功能集。
- 分析服务(Analysis Services)——支持 Analysis Services 引擎。
- Full-Text Filter Daemon——支持全文搜索引擎。
- Reporting Services——支持 Reporting Services 的底层引擎。
- SQL Server 代理(SQL Server Agent)——SQL Server 中作业调度的主引擎。利用该服务，可以按照不同时间表安排作业。这些作业可有多个任务，甚至可以根据先前任务的结果来分解成不同子任务。SQL Server Agent 运行的示例包括备份以及日常导入与导出任务。
- SQL Server——核心数据库引擎，其功能包括 SQL Server 数据存储、查询和系统配置。
- SQL Server 浏览器——支持服务器广播，这样用户浏览你的本地网络时可以确认系统是否安装了 SQL Server。

2.2.2 网络配置

很多时候,网络连接问题是因客户端网络配置不合理,或者是客户端网络配置与服务器端配置不匹配引起的。

SQL Server 提供了几种网络库(Net-Libraries, NetLib)。NetLib 作为客户应用程序与网络协议之间的“绝缘体”。在服务器端,它们的功能相同。SQL Server 2008 提供的 NetLib 包括:

- 命名管道(Named Pipes)
- 默认协议(TCP/IP)
- 共享内存(Shared Memory)
- VIA(专用虚拟适配器,常用于服务器之间基于特殊硬件的通信)

在客户端与服务器端计算机上,相同的 NetLib 必须都可用,这样它们可以通过网络协议彼此进行通信。选择在服务器上不支持的客户端 NetLib 会导致通信连接失败,出现一条错误消息,指出无法连接到指定的 SQL Server。

无论采用哪种数据访问方法和驱动程序类型(SQL Native Client、ODBC、OLE DB 或 DB-Lib),通常是驱动程序与 NetLib 通信。通信过程原理如图 2-1 所示。步骤如下:

- (1) 客户应用程序与驱动程序通信(SQL Native Client、ODBC、OLE DB 或 DB-Lib)。
- (2) 驱动程序调用客户端 NetLib。
- (3) NetLib 调用相应的网络协议,并将数据传送给服务器 NetLib。
- (4) 服务器 NetLib 将客户端的请求传送到 SQL Server。

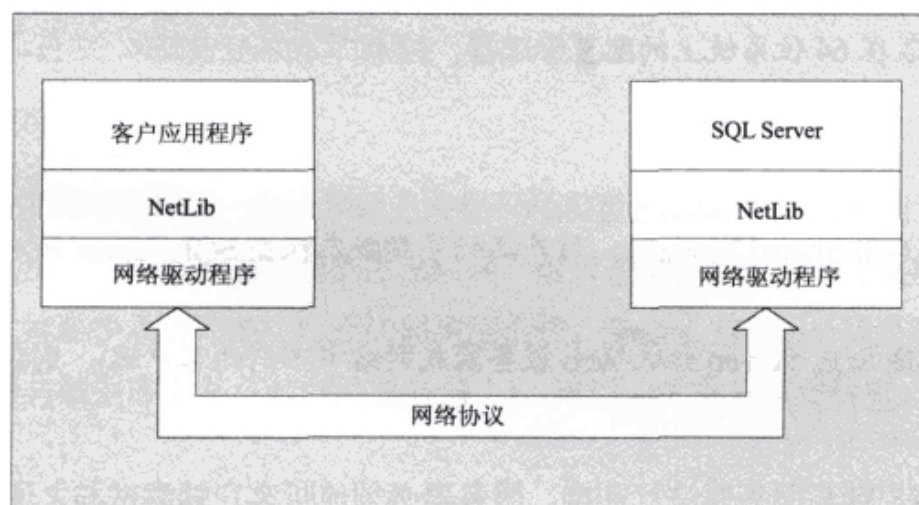


图 2-1

从 SQL Server 到客户端的回复按照同样的顺序进行,只是方向相反。

注意:

如果你熟悉 TCP/IP,那就知道 IP NetLib 将侦听的默认端口是 1433。可将它重新配置为另一端口(有时是因为安全原因),但需要确保客户端系统知道通过哪个端口与服务器通信。通常,我并不赞成作这一修改,因为我相信大部分扫描 SQL Server 端口的黑客所扫描的范围还是较宽的,他们会竭尽全力找到使用的端口。虽说如此,大多数人还是认为这是个正确的做法,因此要谨慎作安装选择。

2.2.3 协议

首先看一下可用的选项。如果启动 SQL Server Configuration Manager, 展开“SQL Server 网络配置”下的“MSSQLSERVER 的协议”树, 将显示如图 2-2 所示的窗口。

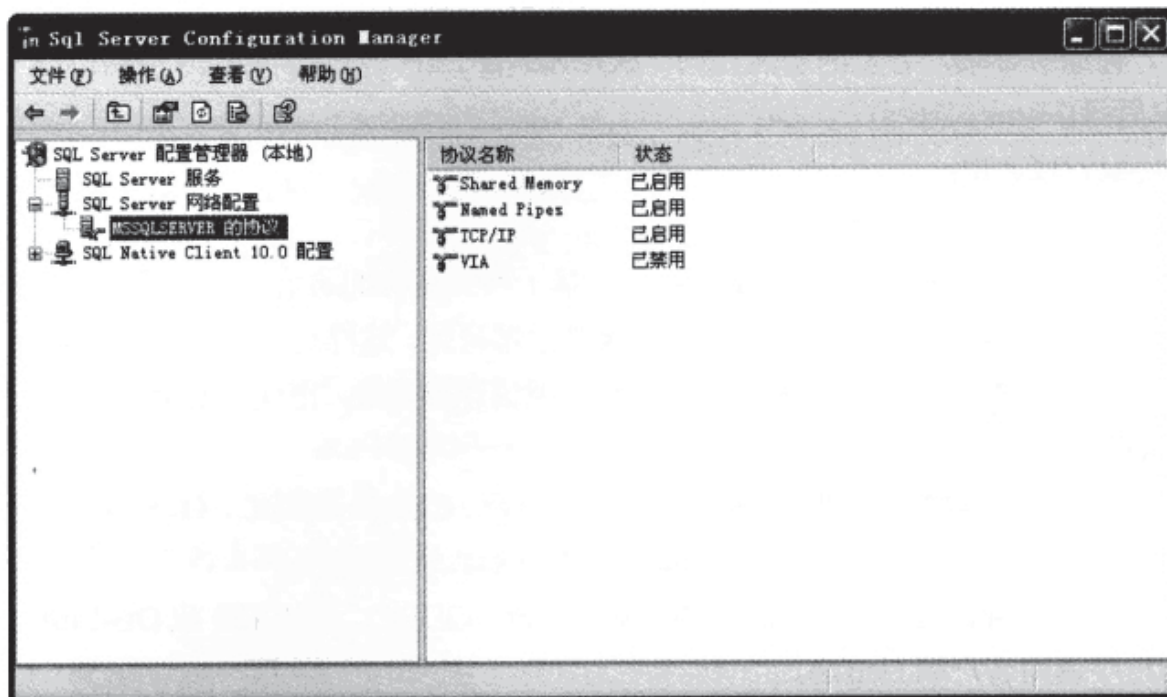


图 2-2

提示:

图 2-2 显示了安装在 64 位系统上的配置管理器。32 位节点不会出现在 32 位安装中, 因为它们默认是默认节点。

注意:

默认情况下, 只启用 Shared Memory。该产品的早期版本根据 SQL Server 和 O/S 的版本默认启用不同的 NetLib。

如果想要远程连接 SQL Server(如从 Web 服务器或网络中不同的客户端), 则需要至少启用一个其他的 NetLib。

记住, 为了让客户端与服务器进行连接, 服务器必须侦听客户端尝试与之通信的协议并在同一端口上侦听。因此, 如果是在 Named Pipes 环境中, 那么可能需要新添加一个网络库。为此, 回到“MSSQLSERVER 的协议”树, 右击“Named Pipes”协议并选择“启用”(在图 2-2 中, 已启用了除 VIA 之外的所有 NetLib)。

提示:

此时你可能要问: “为什么不启用所有的 NetLib? 那样的话, 不就高枕无忧了吗?” 无论将什么添加到服务器, 都会增加开销。在服务器上启用所有的 NetLib 时, 会占用更多的系统开销。这样会降低服务器性能(不是很明显, 但有一点), 而且会降低系统安全性(为何留一扇无人会穿行的门呢?)。

接下来看一下可支持的协议以及选择特定协议的原因。

1. Named Pipes

在没有 TCP/IP 或者没有域名服务(Domain Name Service, DNS)服务器用于将服务器名解析为 TCP/IP 下的地址时, Named Pipes 非常有用。Named Pipes 的使用正逐步减少,这是我喜欢的。因为既然无论怎样都要启用 TCP/IP,那就不要再加入另一个协议(特别是这样就打开了另一条会被黑客用于潜入系统的通路)。

注意:

从技术上讲,可以使用 IP 地址代替名称连接到运行 TCP/IP 的 SQL Server 上。即使没有 DNS 服务,只要确定一条从客户端到服务器的路由,该方法始终有效(如果它有 IP 地址,则不需要名称)。

2. TCP/IP

TCP/IP 协议已经成为事实上的标准网络协议。如果希望通过 Internet(当然,只使用 IP)直接连接到 SQL Server, TCP/IP 协议也是唯一的选项。

提示:

不要将通过 Web 服务器连接数据库服务器与通过 Internet 直接连接数据库服务器两者混淆。可以将 Web 服务器直接连接到 Internet,但不可以将数据库服务器直接连接到 Internet (数据服务器连接到 Internet 的唯一方法是通过 Web 服务器)。

直接将数据服务器连接到 Internet 存在巨大的安全隐患。如果非要直接连接到 Internet(可能有其他原因),则要注意数据服务器的安全防范。

3. Shared Memory

Shared Memory 使得运行在同一台计算机上的客户端和服务端之间不需进行进程间编组(编组是指在跨进程边界传送信息之前将信息打包的方法)。客户端可直接访问服务器存储数据的同一内存映射文件。这样可以极大地减少系统开销,而且访问速度很快。Shared Memory 只是在本地访问服务器时有用(即 Web 服务器与数据库安装在相同的服务器上),可以大大提高系统性能。

4. VIA

VIA 表示虚拟接口适配器(Virtual Interface Adapter),其具体的实现因供应商而异。它通常是一种网络接口,也是两个系统间高性能的、专用的连接。这一高性能部分是由于特殊的专用硬件知道它有一个专用的连接,因此,无需处理普通的网络寻址问题。

2.2.4 客户端配置

只要知道服务器提供的内容,就可以配置客户端。多数情况下,默认的设置就能正常工作。在配置管理器中,展开 SQL Native Client 配置树,然后选择“客户端协议”节点,打开如图 2-3 所示的窗口。

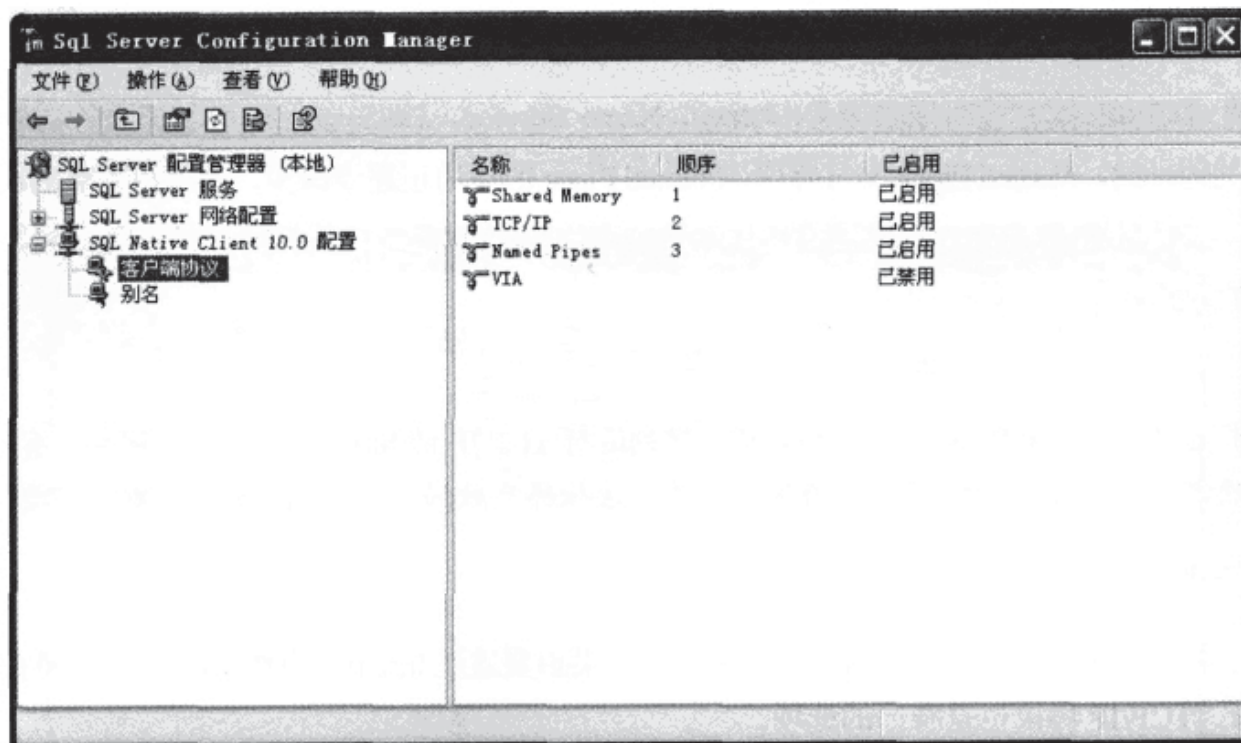


图 2-3

从 SQL Server 2000 开始, Microsoft 提供了一项功能: 即客户端可以先使用一个协议, 如果该协议不起作用, 则再使用另一协议。在图 2-4 中, 按照列顺序, 首先使用 Shared Memory 协议, 然后使用 TCP/IP 协议, 如果 TCP/IP 协议不起作用, 最后使用 Named Pipes 协议。除非更改默认值(使用上下箭头来更改使用协议的先后顺序), Shared Memory 是用于连接到不在别名列表上的任一服务器的第一个协议(“SQL Native Client 配置”的下一节点), 然后是 TCP/IP 协议, 依此类推。

提示:

如果采用的是 32 位安装, 你所看到的对话框可能会与图 2-3 有所不同。在 64 位环境中, 可分别完成 64 位和 32 位的配置, 以便适应不同类型的应用程序。

注意:

如果网络支持 TCP/IP 协议, 则配置服务器使用它。IP 协议的系统开销更少, 而运行速度更快; 除非网络不支持 TCP/IP 协议, 否则没有理由不使用该协议。但值得注意的是, 对于本地服务器(即该服务器与客户端在同一物理系统中)来说, 使用 Shared Memory NetLib 速度更快, 因为不需要通过网络来查看本地 SQL Server。

别名(Aliases)列表是所有服务器的列表, 当连接到某一服务器时, 要使用在这些服务器上定义的特定 NetLib。这意味着, 可以使用 IP 来连接一台服务器, 而使用 Named Pipes 来连接另一台服务器——使用任何所需的 NetLib 连接特定服务器。图 2-4 显示了一台客户端被配置为使用 Named Pipes NetLib 来连接名为 HOBBS 的服务器, 并配置为使用默认设置连接其他任意 SQL Server 服务器。

记住, 网络计算机上的客户端网络配置必须有一个默认协议与服务器支持的协议匹配, 或者必须在别名列表中有一项, 用来指定服务器支持的 NetLib。

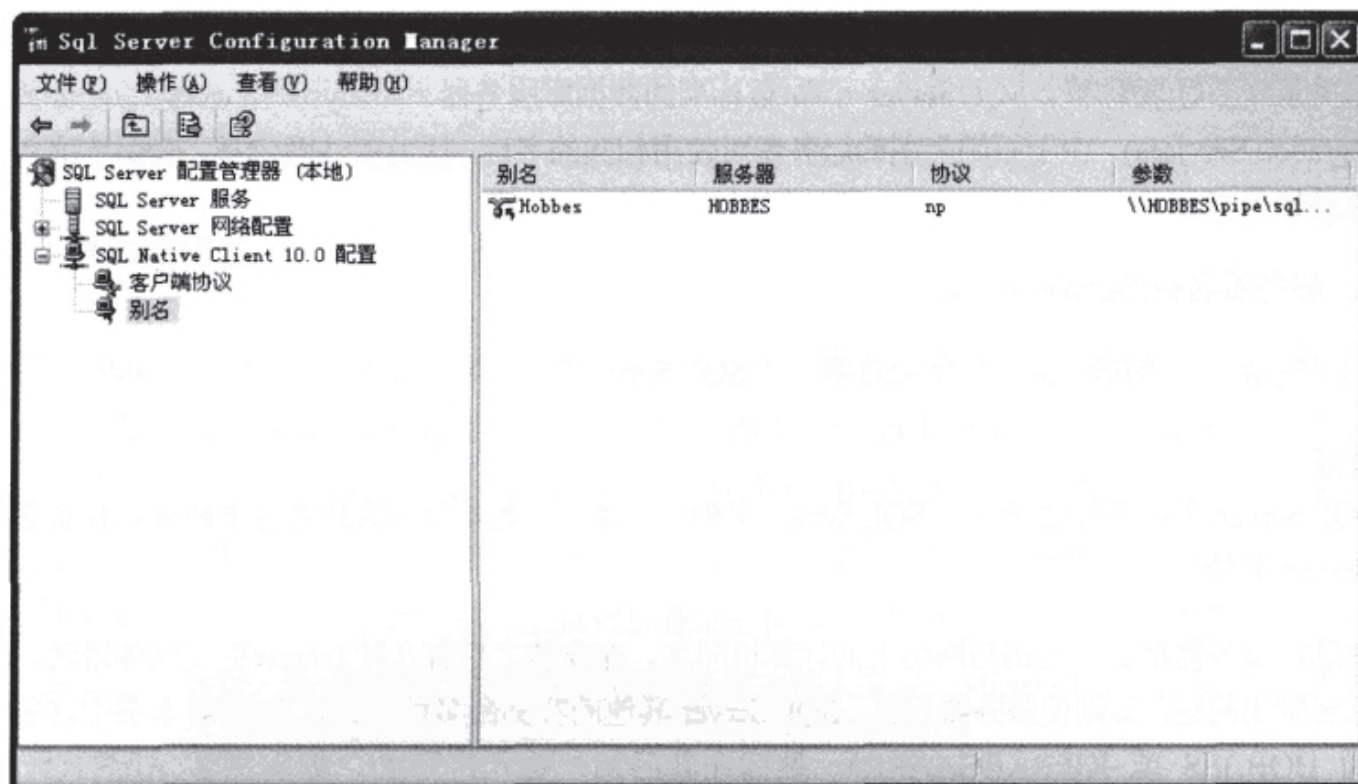


图 2-4

2.3 SQL Server Management Studio

在管理 SQL Server 数据库时, SQL Server Management Studio 是非常优秀的工具。该工具提供了通过较易使用的图形用户界面来管理服务器的各种功能。SQL Server 2005 中首次引入了 Management Studio, SQL Server 2008 对它作了少量修改。通过模仿 Developer Studio IDE 集成环境, 该工具集成了分散在独立工具中的多个功能。

本书不打算介绍 Management Studio 提供的所有功能, 而只是快速浏览以下几点:

- 创建、编辑以及删除数据库与数据库对象
- 管理调度任务(如备份、执行 SSIS 包)
- 显示当前活动(如谁在登录、锁定的对象, 以及在哪个客户端运行), 以及更高级的性能信息
- 管理安全性, 如角色、登录名以及远程服务器和链接服务器等
- 启动并管理 Database Mail Service
- 管理服务器的配置设置
- 建立并管理发布数据库和订阅数据库, 以便执行复制操作

本书中会大量使用 Management Studio 工具, 因此本章详细介绍 Management Studio 的主要功能。

2.3.1 启动 Management Studio

首次启动 Management Studio 时, 会看到一个连接对话框, 要求输入基本的连接信息(服务器类型、服务器名称、身份验证类型, 以及根据身份验证类型, 可能有用户名和密码)。

再次申明, 这里假定你已了解了 SQL Server 的基本知识(假如你不了解, 那需要先阅读《SQL Server 2008 编程入门经典》一书), 因此不会对基本连接作详细介绍。虽说如此, 也有几处与 SQL Server 之前的版本(2000 或更早版本)不同点, 因此下面将作详细说明。

1. 服务器类型(Server Type)

这关系到要登录到哪个 SQL Server 子系统(标准的数据库服务器、Analysis Services、Report Server 或 Integration Services)。由于不同类型的服务器可使用相同的名称,因此要注意选择,确保选项为要登录的服务器类型。

2. 服务器名称(Server Name)

你可能猜到,该选项表示要登录的哪一个 SQL Server。

注意:

SQL Server 允许同时运行多个 SQL Server 实例。不过它们是单独加载到内存中的彼此独立运行的 SQL Server 引擎。

注意,服务器默认的实例与网络上的计算机同名。在安装之后有几种方法来更改服务器名,但这些方法轻则出问题,重则使服务器死机。SQL Server 其他的实例命名可与默认名相同(本书中许多示例命名为 HOBBS 或 KIERKEGAARD),然后在其后加一美元符号(\$),如实例名 ARISTOTLE \$ POMPEII。

如果选择了“.”或(local),则系统使用 Shared Memory NetLib(而不管你连接其他服务器时选择哪一个 NetLib),连接到发起连接同一台计算机上的 SQL Server。这样做有利有弊。不利的是放弃了一小部分控制权(SQL Server 将总是使用 Shared Memory 来连接服务器,而不能选择任意其他协议)。有利的是不用记住登录的服务器,而且由于在本机上运行而能获得高性能。如果使用本地 PC 的实际服务器名,则将通过网络栈进行通信,这将导致与通信相关的系统开销,这与你在与其他系统通信时是一样的,而服务器事实上是位于同一台计算机上(如果是要精确模拟远程系统,这还是不错的选择)。

3. 身份验证类型

可以选择 Windows 身份验证(以前的 NT 身份验证)和混合身份验证。不管如何配置服务器,即使配置为混合身份验证,Windows 身份验证也始终有效。而使用本地 SQL Server 用户名和密码(不是更大的 Windows 网络的一部分)登录仅在专门启用混合身份验证时生效。

Windows 身份验证

从字面意思就可了解 Windows 身份验证。Windows 2000 或更新版本中定义了用户和组。这些 Windows 用户在 Windows 用户配置文件中映射到 SQL Server 登录名。当这些用户想要登录 SQL Server 时,将在整个 Windows 域内对它们进行验证,而且根据登录名映射到角色。角色可以确定允许用户执行哪些操作。

该模型的亮点在于仅有一个密码(如果更改了 Windows 域中的密码,则 SQL Server 登录密码也随之改变)。登录时不需要填写任何用户和密码,仅是从你当前登录 Windows 网络的方式中得到登录信息。此外,管理员必须是仅仅在一个位置管理用户。不利的一面是映射过程很复杂,而且管理 Windows 用户的管理员必须是域管理员。

混合身份验证

数据库的安全性通常不考虑用户在网络中的权限,而只是显式设定用户在 SQL Server 中的权限。验证过程根本不考虑当前登录的网络,用户只需要提供 SQL Server 特定的登录名和密码。

这很有利, 因为对于任一 SQL Server, 管理员不必是域管理员(甚至不需要有一个网络用户名)就能设定 SQL Server 上用户的权限。身份验证过程也比 Windows 身份验证过程简单。而且混合身份验证意味着一个用户可有多个登录名, 并可以对不同对象设定不同权限。

2.3.2 查询编辑器

对于 SQL Server 2008 来说, Management Studio 的查询窗口部分有了一些改变, 它取代了 SQL Server 2005 及之前版本的一个独立工具——查询分析器(Query Analyzer)。它是与指定的 SQL Server 交互式会话的工具, 是执行 Transact-SQL(T-SQL, 我喜欢读为“Tee-Squeal”, 但实际应读为“Tee-Sequel”)语句的地方。T-SQL 是 SQL Server 的本地化语言, 也是结构化查询语言(SQL)的(变体)方言。

过去几年里, 我已习惯于使用这一工具代替查询分析器, 但也发现了其中有些混乱, 因为一个工具要处理太多的事情。虽说如此, 对于那些不沉湎于过去的人来说, Microsoft 希望可以在这样的更大的 Management Studio 中更直观地使用工具。

由于在本书中要经常使用查询编辑器窗口, 因此这里深入介绍该工具, 并帮助你熟悉其使用方法。如果你对 SQL Server 2005 有所熟悉, 那可能希望跳至介绍新的“SQLCMD 模式”一节。

注意:

查询编辑器窗口也存在查询分析器中所设的对结果的字符长度的限制。默认情况下, 对于“以文本格式显示结果”、“将结果保存到文件”或“SQLCMD 模式”下的任何一列, 查询编辑器窗口都返回最多 256 字符。可以通过“工具”|“选项”|“查询结果”|“SQL Server 设置”将其最大值修改为 8092, 但超出的数据(比如来自 blob 列的数据)将被截短。

1. 启动

单击 Management Studio 左上区域的“新建查询”按钮, 或者选择“文件”|“新建”|“使用当前连接查询”菜单项, 打开新的查询编辑器窗口。打开查询编辑器窗口后, 其菜单与早期为单独工具时的查询分析器的菜单非常相似。后面会详细介绍菜单, 但首先来进行一个简单的查询。

首先是以下这个查询:

```
SELECT * FROM INFORMATION_SCHEMA.TABLES;
```

和 SQL Server 2000 的查询分析器中一样, 语句关键字应显示为蓝色。无法确定的项, 如列名和表名(每一服务器上的每一数据库的每一表名都各不相同)显示为黑色。语句参数和连接器显示为红色。你将看到工具栏上的大部分图标都十分类似。例如, 通过工具栏上带有勾号图标的按钮, 可快速分析查询语法, 而不需要实际运行该查询, 这与 Query Analyzer 中的情况相同。

单击工具栏上的“执行”按钮(其旁边有一个红色感叹号图标)。查询编辑器窗口变为如图 2-5 所示。

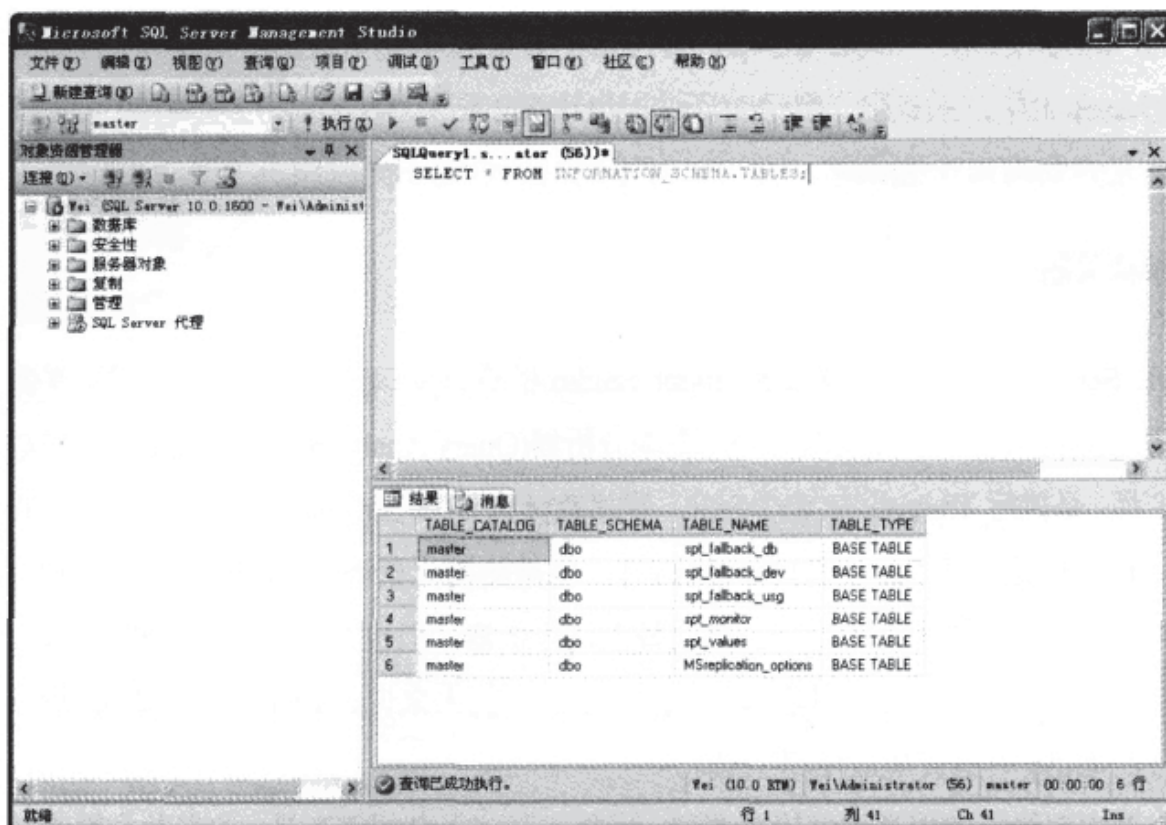


图 2-5

和以前的查询分析器一样，主窗口已经自动划分为两个窗格。上面的窗格是初始查询文本；下面的窗格称为结果窗格。

现在，改变一处或两处设置，查看具体的变化情况。查看查询编辑器窗口上方的工具栏，并且检验由 3 个图标组成的集合，图 2-6 中显示了这 3 个图标。



图 2-6

这些图标可以控制接收输出结果的方法。它们依次是“以文本格式显示结果”、“以网格显示结果”和“将结果保存到文件”。也可以从“查询”菜单的“将结果保存到”子菜单选项选择输出结果的方法。

2. “以文本格式显示结果”选项

“以文本格式显示结果”选项使得查询得到的所有结果以文本页面的方式显示。该页面可以无限长(仅受系统可用内存限制)。

通常在以下几种情况下使用这种输出方法：

- 仅返回一个结果集，且该结果只有很少的列。
- 想使用单个文本文件来保存返回结果。
- 返回多个结果集，但该结果比较小，且不需要使用多个滚动条就可以在同一页面上查看多个结果集。

提示：

正如本章前面提到的，对于这一选项和“将结果保存到文件”选项，查询编辑器窗口可能会截断较长的字符列。默认情况下，它将截断长于 256 字符的字符数据。可以将此设置改成最大值为 8192 字符，不过这对于网格模式不适用。

3. “以网格显示结果”选项

该选项使得返回结果的列和行以网格的形式排列，具有(“以文本格式显示结果”选项不具有)以下几个特点：

- 可以更改列的大小，只要将鼠标指针移到该列标题的右边界，再单击并拖动该列边界到合适的位置。双击右边界使得该列获得适当的大小。
- 如果选择几个单元，然后将其剪切并复制到其他网格(如 Microsoft Excel，或 Open Office 中的 Calc 应用程序)，这几个单元可作为单独的单元处理(如果选择“以文本格式显示结果”选项，剪切的数据会粘贴到一个单元格中)。
- 可以从多行只选择一列或两列(采用“对文本格式显示结果”选项，如果选择中间的几行，所有列都被选中；只能选择第一行和最后一行的中间部分)。

由于上述的优点，该选项最常用。

4. “将结果保存到文件”选项

该选项类似于“以文本格式显示结果”选项，但它是直接将结果输出到文件。我通常使用这一选项生成要用某种实用程序分析的文件或是易于通过电子邮件发送的文件。

5. SQLCMD 模式

SQLCMD 模式是 SQL Server 2008 新提供的，使得查询编辑器窗口的行为与 SQLCMD 实用程序的行为更匹配。其思想就是允许通过用于调试或其他目的的交互式窗口执行本应该用命令行方式运行的批处理文件、查询和脚本。默认情况下，这些 SQLCMD 特定的命令在查询编辑器窗口中不可用。而使用 sqlcmd 模式就可激活这些命令。

6. 显示执行计划

每次运行查询时，SQL Server 要将查询语句解析成其自己的组件部件，然后将它发送到查询优化器(query optimizer)。查询优化器是 SQL Server 的一部分，寻找一种运行查询的最佳方式，以便在快速返回结果以及对其他用户影响最小之间进行平衡。使用“显示估计的执行计划”选项，会返回关于 SQL Server 执行查询的计划的图形表示和辅助信息。同样可以打开“包括实际的执行计划”选项。大多数情况下，实际执行与估计执行的计划相同，但偶尔也有不同，这是因为当再次运行查询语句时优化器决定进行更改，以及运行查询语句时的执行计划的实际花销发生变化。

下面看看简单查询语句中的查询计划，单击“包括实际的执行计划”选项，再执行查询，如图 2-7 所示。

注意，必须实际单击“执行计划”选项卡才能得到实际的返回结果，且以选择的方法显示返回结果。“显示估计的执行计划”选项与“包括实际的执行计划”的输出相同，但以下两种情况除外：

- 立即得到计划，而不是在查询语句执行之后得到计划。
- 虽然查询语句为实际的计划，估计了所有的花销信息，但查询语句实际并未运行。在“包括实际的执行计划”选项下，执行了查询语句，且得到的是实际的花销信息，而不是估计的花销信息。

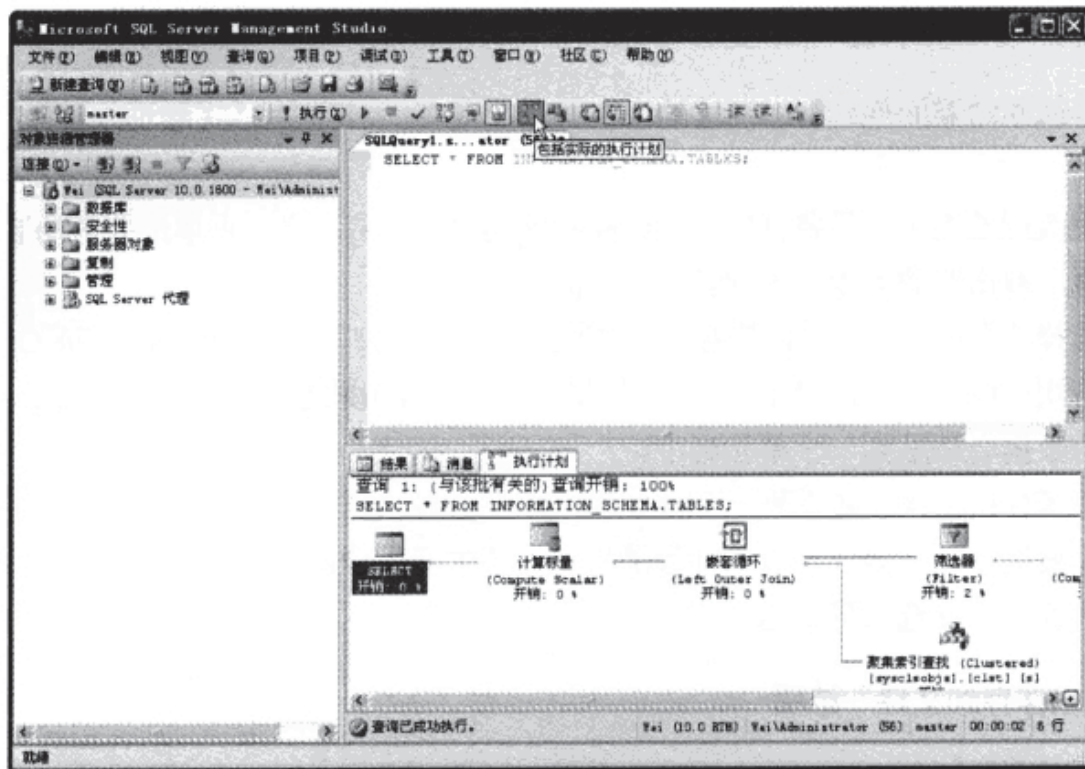


图 2-7

7. DB 组合框

最后介绍一下 DB 组合框。简单地说，你要从该组合框中选择当前窗口中查询语句运行的默认数据库。最初，查询编辑器窗口中为用户登录时默认的数据库(如对于 sa 用户来说为 master 数据库，除非更改了系统)。然后可以更改为当前登录用户有权访问的其他任何数据库。

8. 对象资源管理器

这种实用的小工具可用来浏览数据库，查找对象名，甚至执行一些动作，如编写脚本和查看底层数据。

在图 2-8 的示例中，展开了数据库节点，一直展开到 AdventureWorks2008 数据库中表的列表。可进一步查看单独的列，包括列的数据类型和类似属性。对象资源管理器对于数据库的浏览来说非常方便。

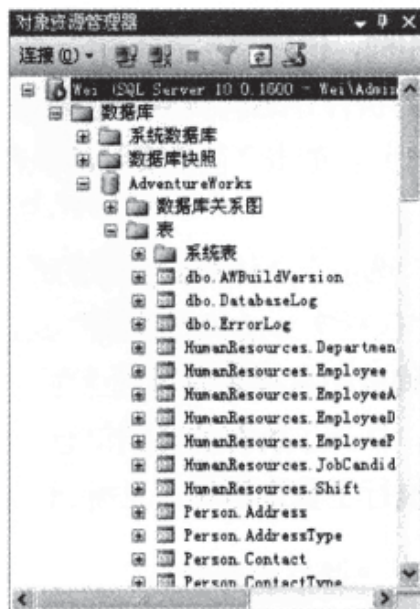


图 2-8

2.4 SQL Server Business Intelligence Development Studio

SQL Server Business Intelligence Development Studio(有时也简称为 BIDS)是一种特殊的 Visual Studio。实际上,这一工具启动后的外观取决于计算机上是否安装了 Visual Studio 2008。如果安装了 Visual Studio 2008,则一组 Visual Studio 菜单将混合到与 SQL Server Analysis Services 和 Integration Services 有关的菜单和模板中。

Business Intelligence Development Studio 是在 SQL Server 2005 中新增的工具,它完全取代了 SQL Server 2000 中的 Analysis Services 和 Integration Services 工具。现在我们所拥有的是更接近于真实的开发环境,而非 SQL Server 2000 中相对有限的多维数据集生成器或 DTS 设计器。

通过 Business Intelligence Development Studio,可以设计 Integration Services 包(稍后将介绍这方面的内容),为 Reporting Services 设计报表,当然还可以直接处理 Analysis Services 项目。有关 Business Intelligence Development Studio 支持的各种服务将在后面的章节中分别讲述。

2.5 SQL Server 集成服务(SSIS)

SSIS 的前身为数据转换服务(Data Transformation Services, DTS),其功能很强大。使用 SSIS,可以简化大量用于完成复杂的数据提取或导入的代码编写工作(通常采用客户端语言)。SSIS 使得可从各种数据源(具有 OLE DB 或 .NET 数据提供程序)取得数据,并将数据注入到 SQL Server 数据表或其他数据目标。

在传送数据时,可以对该数据进行转换。数据转换本质上是指按照某种逻辑规则修改数据。数据的处理可简单,如更改列名,也可复杂,如分析数据完整性并在需要时应用更改规则。要了解其是如何起作用的,可考虑下面这个问题:从一个可为 NULL 值的字段中取得数据并将数据传送到一个不允许使用 NULL 值的表中。使用 SSIS,可在数据传送过程中自动将 NULL 值改成其他所选值(对于数字,可能为 0;对于字符,可能为“unknown”等内容)。第 16 章将介绍 SSIS。

2.6 Reporting Services

Reporting Services 最初是在 SQL Server 2000 后以 Web 下载方式发布的,它在 SQL Server 2005 中成为核心产品的一部分。SQL Server 2008 中对它作了很大调整,添加了新的可伸缩性功能和一些新的设计元素。Reporting Services 提供用于生成报表的框架和引擎。对于 SQL Server 2005 及更早的版本来说,它与内置的 Windows Web 服务器协同工作,在 Web 环境下生成报表。而在 SQL Server 2008 中,Reporting Services 包含了其自身的 Web 服务器,因此不要求配置完整版的 Information Services(IIS)安装。

报表是用基于 XML 的报表定义语言(Report Definition Language, RDL)定义的。Business Intelligence Development Studio 提供一组模板,用于生成简单的或复杂的表。报表被写入 RDL 文件,由 Reporting Services 引擎按需进行处理。第 14 章将更详细地讨论 Reporting Services。

2.7 Bulk Copy Program(bcp)

如果说 SSIS 是新的帮助工具, 那么 Bulk Copy Program 就是老的帮助工具。我们可能很少使用该工具, 但如果使用它, 会发现其作用确实很大。

bcp 是命令行程序, 其主要目的是将格式化数据一起导入和导出 SQL Server。bcp 在 SSIS 很早之前就有, 而 SSIS 在多数输入/输出功能上取代了 bcp, 但 bcp 对于那些习惯命令行工具的人仍能有一定的吸引力。另外在许多 SQL Server 安装过程中, 仍要使用 bcp 工具来快速移动数据。第 15 章将对 bcp 作详细的讨论。

2.8 SQL Server Profiler

很多次在我无计可施的时候, 该工具向我表明了服务器出了什么问题。SQL Server 开发人员(甚至是相关的数据库管理员)并不每天使用该工具, 但当你无计可施的时候, 该工具确实很强大并能解决问题。

SQL Server Profiler 实质上是实时跟踪工具。性能监控器(Performance Monitor)在宏观上(系统配置方面)跟踪系统的运行, 而 SQL Server Profiler 跟踪细节。这两个工具各有利弊。Profiler 可根据跟踪配置, 给出服务器上执行的每一语句的详细语法。假设要对具有 1000 个用户的系统进行性能优化。可以想象得到, 要打印如此多的人执行的语句, 一两分钟就可能用掉大量纸张。幸运的是, Profiler 工具具有很强的筛选功能, 可跟踪更具体的问题, 如长期运行的查询, 或者存储过程中正在运行的查询语句的准确语法。如果存储过程包含一些条件语句, 导致其在不同环境下进行不同的操作, 这时 Profiler 工具就很有用。第 22 章将对 SQL Server Profiler 作一些讨论。

2.9 sqlcmd

在 SQL Server 程序组中不会看到 sqlcmd。实际上, 很多人甚至不知道该工具(或 osql 和 isql)的存在。因为 sqlcmd 是控制台, 而不是 Windows 程序。

有时, 如果想将脚本编写到较大的命令行进程中, sqlcmd 可帮助实现它。sqlcmd 使用起来很方便, 尤其是要使用包含想在 sqlcmd 中运行脚本的文件时更是如此。但是记住: 还有其他工具能更高效地完成通过 sqlcmd 工具完成的功能, 而且具有和其他 SQL Server 工具一致的用户界面。

除了命令行操作, sqlcmd 还支持一些与其他命令行实用工具更具内联性的特殊命令。例如, !!DIR 提供一个目录清单。

提示:

要能够理解 SQL Server 不同的“方言”, sqlcmd 只是该工具的众多名称中的另一个新名称。它最早称为 ISQL, 在 SQL Server 2000 和 7.0 版本中称为 osql。

第 15 章将连同 bcp 一起讨论 sqlcmd 的基本知识。

2.10 小结

本章介绍的大部分工具并不是你以后每天要使用的工具。事实上,对于大部分开发人员来说,每天需要使用的只有 SQL Server Management Studio 工具。然而,了解每一工具的功能还是很有必要的。每一工具都有其重要用途,在学习本书的过程中还会使用到这些工具。

注意,在“开始”菜单中没有其他一些可用的实用工具的快捷方式(如连接工具、服务器诊断与维护实用工具),这些实用工具通常与管理相关。





第 3 章

提出更好的问题：高级查询

在切入本章的主题之前，首先要提醒那些对 SQL Server 的许多功能都已经很熟悉的读者：这部分发生了很大的变化。SQL Server 现在支持诸如 INTERSECT 和 EXCEPT 之类的关键字。使用新的 MERGE 命令可以将一条语句的结果并入另一条语句。这里还需要回顾一下递归查询。

每次写书的时候，我都要为高级查询的主题大伤脑筋。例如，究竟什么是“高级”？过去，我一直与别人争辩应该在游标之前执行高级查询还是在游标之后执行高级查询。这次，我又简单考虑了这个问题，但是却发现自己一直在考虑的是应该和 Beginning 一书所涵盖的那些主题有多少重叠。随后出现了其他的问题，例如，是否要先深入讨论索引？如何讨论 XML？这其实就是一个先有鸡还是先有蛋的问题。在使用本章涉及的主题之前，你不需要了解游标，但是，我们会讨论不使用游标的不同查询方法所具有的优点——而且如果你知道自己要尽力避免的是什麼，那么了解这些好处是非常有益的。在考虑集成关系数据和基于 XML 的数据，或者要讨论不同的查询策略影响查询优化器索引项的方式时也存在类似的问题。

提示：

对我而言，本章中的一些主题代表了 SQL Server 编程中“初学者”与“专业人员”之间的重大区别。显然，它们不仅标志着你成为一名真正的“专业人员”，而且开发人员可以从“我知道可以使用什麼，而且，我将使用其中的一种或两种东西”发展到通过它们解决那些无法解决的查询，这样做意义非常重大。这些主题可以告诉初学者能够使用什麼，并且让他们体验一下能够实现什麼。对于专业人员来说，由于深刻理解概念对于 SQL Server 中更高层次的成功是至关重要的(就这点而言，对于所有主要的 DBMS 几乎都是这样的)，因此，这些主题对他们同样适用。

本章将使用一种读者前所未见的方法讨论查询。我们将讨论使用什麼方法把多个问题合成一个查询。实际上，这里要讨论的是如何把看似很多个的查询放置在一起，以便作为一个完整的单元来执行。我们还会介绍一些比较新的功能(一些是 2008 新增的，还有一些是 2005 新增的)，这些功能可以提供比以前更多的选项。明白了这一点，我们还要讨论查询的性能，以及如何从查询中获得最多信息。编写最好的查询不仅仅是技巧或使它们更加复杂的问题——而是令它们完成任务。

本章将涵盖以下主题：

- 子查询概述

- 相关子查询
- 派生表
- 使用 EXISTS 运算符
- 利用 INTERSECT 和 EXCEPT 运算符
- 通用表表达式(CTE)
- 利用递归查询
- MERGE 命令
- 利用外部调用执行复杂操作
- 优化查询性能

通过学习这些主题可以明白，利用子查询，如何把看似不可能的事情变成完全可能的事情，以及如何局部调整，大大改变查询性能。

3.1 子查询概述

子查询(subquery)是嵌套在其他查询中的常规 T-SQL 查询(用括号括起来)，创建子查询就需要一条 SELECT 语句作为数据部分的基础或者另一个查询中的条件要素。

通常，使用子查询可以满足下列需求之一：

- 将一个查询分解为一系列的逻辑步骤
- 提供一个列表作为 WHERE 子句和[IN|EXISTS|ANY|ALL]的目标对象
- 提供由父查询中的每条记录驱动的查找

要设想和构建一些子查询是很简单的，但是有些子查询却非常复杂——通常取决于内部(子)查询和外部(上层)查询之间关系的复杂程度。

除此之外，要注意的是还可以用联结(join)编写大部分的子查询(当然不会是全部)。在能够使用联结替代子查询的场合下，联结一般是更可取的。

3.2 构建嵌套子查询

嵌套子查询(nested subquery)只朝一个方向进行查询——要么返回一个用于外部查询的值，要么返回一个和 IN 运算符一起使用的值列表。下面给出了在最自由的情况下可能出现的两种查询语法模板：

```
SELECT <SELECT list>
FROM <SomeTable>
WHERE <SomeColumn> =
    SELECT <single column>
    FROM <SomeTable>
    WHERE <condition that results in only one row returned>
```

或

```
SELECT <SELECT list>
FROM <SomeTable>
```

```
WHERE <SomeColumn> IN (
    SELECT <single column>
    FROM <SomeTable>
    [WHERE <condition>])
```

当然，实际的语法将有所不同。不但要替换选择列表和具体的表名，而且在内部或外部的查询(或者两者都有)中还可能有多个表的联结。

3.2.1 使用单值 SELECT 语句的嵌套查询

下面开始详细讨论一个直接的实例以便了解内情。例如，假设我们要知道从系统中购买产品的第一天所出售的所有产品项的 **ProductID**。如果已经知道在系统下订单的第一天的日期，那么一切将迎刃而解。但是如果不知道的话又该怎么办呢？我们可以使用一个嵌套子查询来获得首次销售的日期，然后将其用于外部查询。所有的这些都可以在一条语句中完成：

```
USE AdventureWorks2008;

SELECT DISTINCT soh.OrderDate, sod.ProductID
FROM Sales.SalesOrderHeader soh
JOIN Sales.SalesOrderDetail sod
    ON soh.SalesOrderID = sod.SalesOrderID
WHERE soh.OrderDate =
    (SELECT MIN(OrderDate) FROM Sales.SalesOrderHeader);
```

上面的代码返回 47 行数据：

```
OrderDate                ProductID
-----
2001-07-01 00:00:00.000  707
2001-07-01 00:00:00.000  708
2001-07-01 00:00:00.000  709
...
...
2001-07-01 00:00:00.000  776
2001-07-01 00:00:00.000  777
2001-07-01 00:00:00.000  778

(47 row(s) affected)
```

就是这样的简单便捷。内部查询(`SELECT MIN(...)`)获取一个值，供在外部查询中使用。由于这里使用了等号，所以内部查询只能从一行中返回一个列，否则会导致运行错误。

3.2.2 使用返回多个值的子查询的嵌套查询

或许这些子查询在所有实施的子查询中是最常见的，它们获取某种类型的范围列表并在查询中将其作为一种标准使用。假设要得到应聘了公司另一个职位的所有雇员的列表。在名为 **HumanResources.JobCandidate** 的表中列出求职者，这样我们要获取的就是在求职者表中有记录的 **EmployeeID**(实际上是指表中的 **BusinessEntityID**)列表。当然，**HumanResources.Employee** 表实际

列出了所有的雇员。此外，还必须使用 `Person.Person` 表来获得雇员姓名之类的信息。

下面给出了可能使用的代码：

```
USE AdventureWorks2008;

SELECT e.BusinessEntityID, FirstName, LastName
FROM HumanResources.Employee e
JOIN Person.Person pp
  ON e.BusinessEntityID = pp.BusinessEntityID
WHERE e.BusinessEntityID IN
  (SELECT DISTINCT BusinessEntityID FROM HumanResources.JobCandidate);
```

该代码返回两行数据：

BusinessEntityID	FirstName	LastName
212	Peng	Wu
274	Stephen	Jiang

(2 row(s) affected)

此类的查询几乎总是属于可以使用内部联结而非嵌套的 `SELECT` 来实现的范围。例如，运行下面的简单联结可以得到与前面的子查询相同的结果：

```
USE AdventureWorks2008;

SELECT e.BusinessEntityID, FirstName, LastName
FROM HumanResources.Employee e
JOIN Person.Person pp
  ON e.BusinessEntityID = pp.BusinessEntityID
JOIN HumanResources.JobCandidate jc
  ON e.BusinessEntityID = jc.BusinessEntityID;
```

出于性能上的考虑，如果没有特别的理由必须使用嵌套 `SELECT`，默认的解决方案应该是联结方法——在本章结束前我们会进一步讨论。

使用嵌套 `SELECT` 找出孤立记录

此类嵌套 `SELECT` 类型与前一个例子几乎没有差别，只是这里添加了 `NOT` 运算符。这样做将导致的不同之处是，在转换成联结语法时，会等同于外部联结而非内部联结。

这种嵌套 `SELECT` 查询适用于要了解什么被遗漏的场合。例如，在上一级表中没有父记录的订单明细(在 `AdventureWorks` 数据库中，由于有外键的约束，不会出现这样的情况，但是在其他数据库中可能存在这种情形)。这次的例子稍有不同，本例要找出没有应聘公司其他职位的雇员。在这里出现了“没有”的字样，所以你应该知道做什么了——在查询中添加 `NOT`(但是，这里一定要当心必须要应对的特殊问题)：

```
USE AdventureWorks2008

SELECT e.BusinessEntityID, FirstName, LastName
FROM HumanResources.Employee e
```

```
JOIN Person.Person pp  
ON e.BusinessEntityID = pp.BusinessEntityID
```

```
WHERE e.BusinessEntityID NOT IN  
(SELECT DISTINCT BusinessEntityID  
FROM HumanResources.JobCandidate  
WHERE BusinessEntityID IS NOT NULL);
```

运行代码，当然可以得到一个很大的结果集(除了在前一个例子中看到过的两名雇员之外的所有雇员)。

注意：

像通常一样，在可能包含 NULL 值的集合上进行测试要当心。与 NULL 值比较总是会得到 NULL。在前面的情况下，JobCandidate 表中有的行的 BusinessEntityID 为 NULL。如果允许子查询中再次出现 NULL，那么在与 NOT IN 比较时，外部查询中的所有行都会被确定为错误的——这样将会返回空列表(建议自己试验一下，以保证理解它们之间的差异)。

3.2.3 ANY、SOME 和 ALL 运算符

到目前为止，我所撰写的每一本书都会占用一页的篇幅介绍这些内容。我想这次也要这么做并把自己以前写的那些页码记下来。但是每次读到它的时候，我都会对它感到厌烦，所以这次我决定写得更加简洁一些。

除了 IN 运算符以外，SQL Server 还支持 ANY、SOME 和 ALL 运算符。简言之，它们都是无用的。据我所知，它们主要的目的是占用这本书中的一页篇幅并浪费掉你可能用来学习其他有用知识的十分钟时间。以恰当的方式使用更加通用的运算符(诸如 >=、<=、<>、!= 等)就可以重复所有功能，而且以那样的方式出现在我所见到过的真实例子中，这些功能会变得更加容易读懂。

简言之，知道存在这些内容，但是除此之外不想再浪费你们的时间。因为它们并没有提供任何的新知识。

3.3 相关子查询

你要对本节的讲述特别用心！这又是一个很小的领域，一旦真正“掌握”了它，你就可以鹤立鸡群。这里所说的“掌握”，不仅仅意味着理解它的工作方式，而且要明白它的重要性。

相关子查询可以变不可能为可能。除此之外，它们常常把几行代码变为一行代码，并且往往可以提升相应的性能。相关子查询所伴随的问题是，它们通常需要与常规方式完全不同的思维方式。相关子查询可能是 SQL 中最容易学习的概念，但是由于它与你的思考方式相反，所以它很快就又会被遗忘。如果你可以记住它，那么你将是能够解决难题的少数人之一。此外，在需要从查询中获取最大性能的时候，你将会拥有完备得多的工具集。

3.3.1 相关子查询的工作方式

相关子查询和前面讨论过的嵌套子查询的区别在于：在相关子查询中，信息是双向进行的而

非单向进行的。在嵌套子查询中，内部查询只处理一次，信息就被传出以便用于外部查询，外部查询也只执行一次——实质上，如果愿意亲自输入的话，这里提供的值或列表与你自己输入的完全相同。

但是，在相关子查询中，内部查询根据外部查询提供的信息执行，反之亦然。虽然这样看起来似乎有些混乱(再次出现了先有鸡还是先有蛋的问题)，但是它按照下面的三个步骤执行：

- (1) 外部查询获得一条记录并将其传入内部查询；
- (2) 基于传入的值执行内部查询；
- (3) 内部查询将自己返回的结果值传给外部查询，外部查询利用这些值完成自己的处理。

3.3.2 WHERE 子句中的相关子查询

本小节的内容可能有些难于理解，所以下面通过一个例子来讲解它。

再次查看这个查询，我们希望通过该查询了解第一次在系统中下订单的那一天中都有哪些订单。不过，这一次查询中要加一些新的改动：我们想要知道每一位顾客在系统中的第一份订单的 OrderID 和 OrderDate，即要知道顾客第一天下订单的日期以及订单的 ID。下面开始一步步讨论。

首先，每一个结果中要有 OrderDate、OrderID 和 CustomerID。在 SalesOrderHeader 表中可以找到所有的这些信息，因此，可以知道应该基于该表(至少部分基于该表)进行查询。

其次，想要知道每一位顾客在系统中第一次下订单的日期比较麻烦。在使用嵌套子查询时，我们只是查找整个文件中最早的日期——而现在，需要得到单独顾客的值。如果用两个单独的查询来实现，只需创建一个临时表，然后重新联结，不过需构建两个完全独立的结果集，而且一般会对性能产生负面影响。

有时候，使用两个查询是唯一能不使用游标完成任务的解决方法——但是此类情况并非如此。

如果想要在一个查询中实现，必须找到查询每一个人的方法。我们可以使用内部查询完成基于外部查询中当前的 CustomerID 的查找来解决。然后，需要把值返回给外部查询，以便外部查询能都根据最早下订单的日期来进行查找。

下面给出了它的具体实现：

```
SELECT ol.CustomerID, ol.SalesOrderID, ol.OrderDate
FROM Sales.SalesOrderHeader ol
WHERE ol.OrderDate = (SELECT MIN(o2.OrderDate)
                      FROM Sales.SalesOrderHeader o2
                      WHERE o2.CustomerID = ol.CustomerID)
ORDER BY CustomerID;
```

执行该语句会返回 19 134 行。在该查询中，有一些关键点要注意：

- 这里关于行受影响的说明只有一行——很好地证明了只需要执行一个查询计划；使用不同的查询，可能有两个查询计划。
- 外部查询(本例中)看起来与嵌套子查询非常类似，但是，内部查询对于外部查询有一个显式的引用(注意别名“ol”的使用)。

- 内部查询和外部查询中都使用了别名(尽管看起来外部查询似乎不需要这样)，之所以这样，是因为当要显式地引用另一个查询中的列时需要别名(内部的查询对于外部查询中的列，反之亦然)。

注意：

后面关于是否需要使用别名的观点比较混乱。事实上，有时需要使用别名，有时不需要使用别名。虽然对于本章前面部分讨论的嵌套子查询类型，我倾向于不使用别名，但是，在处理相关子查询时，我建议对所有的事物使用别名。

必须对所有将被另一个查询引用的表(以及表中相关的列)使用别名。问题在于这样可能很快变得非常混乱。安全可靠的方法是对所有的事物使用别名——使你可以明确将从哪一个查询中的哪一个表获取信息。

这里看到 19 134 行受到一次影响。因为它只影响 19 134 行一次。通过观察，我们猜测此版本或许比两个查询的版本执行得更快，而且，事实确实如此。同样，随后会深入研究这部分内容。

在这个特定查询中，外部查询只在 WHERE 子句的引用内部查询——也可以在选择列表中从内部查询中请求数据。

通常，是否使用别名取决于我们是否想要使用，但是相关子查询中必须使用别名。此处的查询很好地展示了原因，即内部查询和外部查询都基于相同的表。由于两个查询都要从对方那里获得信息，若不使用别名，怎么才能知道对该表的哪一项数据感兴趣呢？

3.3.3 SELECT 列表中的相关子查询

可以使用子查询在选择结果中提供一种不同类型的答案。这种情况通常出现在要找寻的消息完全不同于查询中的其他数据的时候(例如，当想要在一个字段上进行聚集，但却不希望因此而影响返回的其他字段)。

为了试验，下面对前一节中用到的查询稍做修改。假设这里只要查找顾客的名字以及首次订购的日期。

这里所做的改变可能比初看上去要大得多。现在要查找顾客的名字，这意味着必须用到 Customer 和 Person 表。此外，我们不需要再构造任何类型的条件——这里要查找所有的顾客(没有任何限制)，我们只是想知道他们第一次订购的日期。

实际上，本次查询比前面的查询要简单一些，代码如下所示：

```
SELECT pp.FirstName, pp.LastName,
       (SELECT MIN(OrderDate)
        FROM Sales.SalesOrderHeader o
        WHERE o.CustomerID = c.CustomerID)
       AS "Order Date"
FROM Person.Person pp
JOIN Sales.Customer c
  ON pp.BusinessEntityID = c.PersonID;
```

得到结果如下所示：

FirstName	LastName	Order Date

Catherine	Abel	2003-09-01 00:00:00.000
Kim	Abercrombie	2001-09-01 00:00:00.000
Humberto	Acevedo	2001-09-01 00:00:00.000
...		
...		
...		
Krystal	Zimmerman	2004-01-29 00:00:00.000
Tiffany	Zimmerman	2003-01-26 00:00:00.000
Jake	Zukowski	2001-09-02 00:00:00.000

{19119 row(s) affected}

注意，日期随顾客的变化而变化——给出的订单日期是特定顾客第一次下订单的日期。

3.4 派生表

有时需要处理查询的结果，但是，有时候不能使用前面讨论的子查询的方式来处理查询的结果。例如，对于给定的表中的每一行，子查询中可能有多个结果，但是这里的查找操作比 IN 运算符提供的查找操作更加复杂。实质上，这里谈论的是在子查询中使用 JOIN 运算符的情形。

此时，偶尔使用 SQL 中不算太出名的结构——派生表(derived table)。派生表(有时称为“内联视图”)由查询结果集的列和行组成(既然它们与常规的表一样，有列、行、数据类型等等，为什么不像使用常规表一样来使用它们?)。

假设要查找订购过某种特定产品的顾客列表——如迷你水泵。很简单！查询如下所示：

```
SELECT pp.FirstName, pp.LastName
FROM Person.Person AS pp
JOIN Sales.Customer sc
    ON sc.PersonID = pp.BusinessEntityID
JOIN Sales.SalesOrderHeader AS soh
    ON sc.CustomerID = soh.CustomerID
JOIN Sales.SalesOrderDetail AS sod
    ON soh.SalesOrderID = sod.SalesOrderID
JOIN Production.Product AS p
    ON sod.ProductID = p.ProductID
WHERE p.Name = 'Minipump'
```

这样的查询很容易。但是现在要对要求做些改动——眼下想要查找既订购过迷你水泵又订购过带有 AWC 标志的棒球帽的所有顾客。注意，这里提到的是订购了这两样产品的顾客——现在麻烦来了。你可能会倾向于使用下面的语句：

```
WHERE p.Name = 'Minipump' AND p.Name = 'AWC Logo Cap'
```

但上面的语句根本不起作用——由于每一行都是一个单独的产品，因而一个产品的名称不可能同时既叫迷你水泵又叫 AWC 棒球帽。上面的代码无法满足我们的要求(事实上，运行这样的代码不会返回任何结果)。

这里真正需要的是，将查找迷你水泵购买者的查询结果与查询棒球帽的购买者的查询结果结

合起来。那么，如何联结查询结果呢？这时候你可能会想到本小节的标题，通过使用派生表来解决这个问题。

要创建派生表，需要做两件事情：

- 用圆括号括住生成结果集的查询；
- 给查询结果赋别名。

语法如下：

```
SELECT <select list>
FROM (<query that returns a regular result set>) AS <alias name>
JOIN <some other base or derived table>
```

下面，用派生表来实现我们的要求。再重复一遍，此处想要查询既订购过迷你水泵又订购过带有 AWC 标志的棒球帽的所有顾客的名字。查询如下所示：

```
SELECT DISTINCT pp.FirstName, pp.LastName
FROM Person.Person AS pp
JOIN (SELECT sc.PersonID
      FROM Sales.Customer sc
      JOIN Sales.SalesOrderHeader AS soh
        ON sc.CustomerID = soh.CustomerID
      JOIN Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
      JOIN Production.Product AS p
        ON sod.ProductID = p.ProductID
      WHERE p.Name = 'Minipump') pumps
ON pp.BusinessEntityID = pumps.PersonID
JOIN (SELECT sc.PersonID
      FROM Sales.Customer sc
      JOIN Sales.SalesOrderHeader AS soh
        ON sc.CustomerID = soh.CustomerID
      JOIN Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
      JOIN Production.Product AS p
        ON sod.ProductID = p.ProductID
      WHERE p.Name = 'AWC Logo Cap') caps
ON pp.BusinessEntityID = caps.PersonID;
```

运行后发现，迷你水泵与带有 AWC 标志的棒球帽似乎是一种常见的订购组合——返回结果有 83 行：

FirstName	LastName
-----	-----
Aidan	Delaney
Alexander	Deborde
Amy	Alberts
...	
...	
...	
Valerie	Hendricks


```

Yale                Li
Yuping              Tian

```

```
(83 row(s) affected)
```

如果想要验证结果，只需要单独运行两个派生表并比较结果。

提示：

对于这个特殊的查询，需要使用 DISTINCT 关键字。如果不使用这个关键字，可能会为每一个顾客返回多行结果。

可以看到，我们不仅可以把看似不可能的查询变成可能，而且性能也很不错。

要记住，派生表并不能解决所有的问题。例如，当结果相当庞大而且有大量的记录要联结时，可能要考虑使用临时表并在其上构建索引(派生表没有索引)。每种解决方法都不同，现在你又多了一种选择。

3.5 EXISTS 运算符

这里称 EXISTS 为运算符，而联机丛书称之为关键字。这或许是因为它在某些意义上不易描述。虽然 EXISTS 运算符类似于 IN 关键字，但是两者看待事物的方法有所不同。

使用 EXISTS 并不会真正返回数据——事实上返回的是满足 EXISTS 语句中所建立的查询条件的数据是否存在的简单的 TRUE/FALSE。

为了解 EXISTS 的应用，下面直接看一个例子。在该例子中，将再次使用前面的嵌套 SELECT 的一个例子——要想得到应征了该公司另一个职位的雇员列表。

```

SELECT e.BusinessEntityID, FirstName, LastName
FROM HumanResources.Employee e
JOIN Person.Person pp
ON e.BusinessEntityID = pp.BusinessEntityID

```

```

WHERE EXISTS
  (SELECT BusinessEntityID
   FROM HumanResources.JobCandidate jc
   WHERE e.BusinessEntityID = jc.BusinessEntityID);

```

该语句的查询结果与更标准的嵌套查询的结果一样，返回两条记录：

EmployeeID	FirstName	LastName
212	Peng	Wu
274	Stephen	Jiang

```
(2 row(s) affected)
```

如果检查嵌套查询实现的版本，可以发现通过联结或许能轻松地实现同样的功能：

```

SELECT e.BusinessEntityID, FirstName, LastName
FROM HumanResources.Employee e

```

```
JOIN Person.Person pp
  ON e.BusinessEntityID = pp.BusinessEntityID
JOIN HumanResources.JobCandidate jc
  ON e.BusinessEntityID = jc.BusinessEntityID;
```

既然使用基于联结的语法也能得到同样的结果(可能存在分类差异),那么,为什么需要新的语法?答案简单明了——性能。

使用 EXISTS 关键字时,SQL Server 不必联结所有的行。实际上,SQL Server 会一直浏览记录直至找到第一个满足条件的记录,然后就地停止。只要有一条记录满足条件,EXISTS 将为真,就不需要继续浏览。与内部联结相比,这里的性能差异更显著。与直接的 EXISTS 语句相反,SQL Server 只是应用了一点相反的逻辑。使用 NOT 时,只要发现了一条满足的记录,SQL Server 依然会停止——唯一的不同是这里的查找会返回 FALSE 而非 TRUE。除了性能之外,其他与查询有关的所有事情都是一样的。

其他使用 EXISTS 的方式

如果经常使用 SQL 创建脚本,会在 CREATE 语句前面看到如下所示的奇怪的东西:

```
IF EXISTS (SELECT * FROM sysobjects WHERE id =
object_id(N'[Sales].[SalesOrderHeader]') AND OBJECTPROPERTY(id,
N'IsUserTable') = 1)
DROP TABLE [Sales].[ SalesOrderHeader]
GO

CREATE TABLE [Sales].[ SalesOrderHeader] (
...
...

```

在风格上可能会有些变化(即,可能使用 sys.objects, sys.databases 或 INFORMATION_SCHEMA 视图),不过概念没有变:在执行 CREATE 之前检查对象是否已经存在。如果表已经存在,有时可能要跳过 CREATE,有时可能要删除表(就像前面例子中所做的那样)。这种想法十分简单——要跳过某个潜在的错误情况(如果表已经存在,CREATE 语句将退出,弹出错误消息,并中止脚本)。

作为一个简单的例子,这里要构建一个小脚本创建数据库对象。由于感兴趣的是 EXISTS 而非 CREATE 命令,因此要用尽可能少的语句实现:

```
USE master

GO

IF NOT EXISTS (SELECT 'True' FROM sys.databases WHERE name = 'DBCreateTest')
BEGIN
    CREATE DATABASE DBCreateTest
END
ELSE
BEGIN
```



```
PRINT 'Database already exists. Skipping CREATE DATABASE Statement'  
END  
GO
```

首次运行上面的语句时，没有名为 `DBCreateTest` 的数据库(除非你在这之前，凑巧创建了名为 `DBCreateTest` 的数据库)，因而数据库得以创建。

再次运行这个脚本，将看到不同的情况发生：

```
Database already exists. Skipping CREATE DATABASE Statement
```

因此，无需过多的修饰，这里加入了一个相当小的脚本以便产品的安装者使用。无论是购买成品的终端用户，还是你自己，会发现它甚至比完整脚本都好。

总之，`EXISTS` 确实是非常有用的关键字。它可以加快一些查询的运行，而且还能简化一些查询和脚本。

提示：

这里要注意的是——此处又是一个容易陷入“传统思维”的地方。尽管在绝大多数可以使用 `EXISTS` 的查询中，`EXISTS` 优于其他选项，但并非一直如此——要记住，规则有时是用来打破的。

3.6 INTERSECT 和 EXCEPT 运算符

特殊关键字 `INTERSECT` 和 `EXCEPT` 对两个结果集的操作类似于 `UNION`。对 SQL Server 来说，它们仍是较新的运算符，但是它们在其他关系数据库管理系统中的使用已经很多年了。

这样说来，它们到底有什么功能呢？`INTERSECT` 和 `EXCEPT` 为我们提供了一种其他的方法显示不同结果集的查找组合。图 3-1 给出了使用不同结果集组合运算符得到的数据，总之，其工作方式如下：

- `UNION`——包括所有结果集中的数据，而且没有重复(所有重复的行都只表现为一个实例)。
- `UNION ALL`——包括两个结果集中的所有数据，不论它们中间是否有重复。
- `EXCEPT`——只包括 `EXCEPT` 关键字左边而且右边的结果集中不存在的那些行。基本上，这等同于“显示 A 中存在、B 中不存在的任意行”。
- `INTERSECT`——只包括两个结果集中都存在的那些行。`INTERSECT` 类似于一个内部联结，不同的是 `INTERSECT` 对结果集中重复的行进行操作而不是对某个特定的联结列进行操作。

我以前著作的那些读者们知道我热衷于举例，而且这里也将给出一些示例，首先要指出的是使用 `EXISTS` 运算符可以轻松地模拟这些语句，所以我们会对两者进行研究。图 3-1 显示了它们各自的语法。

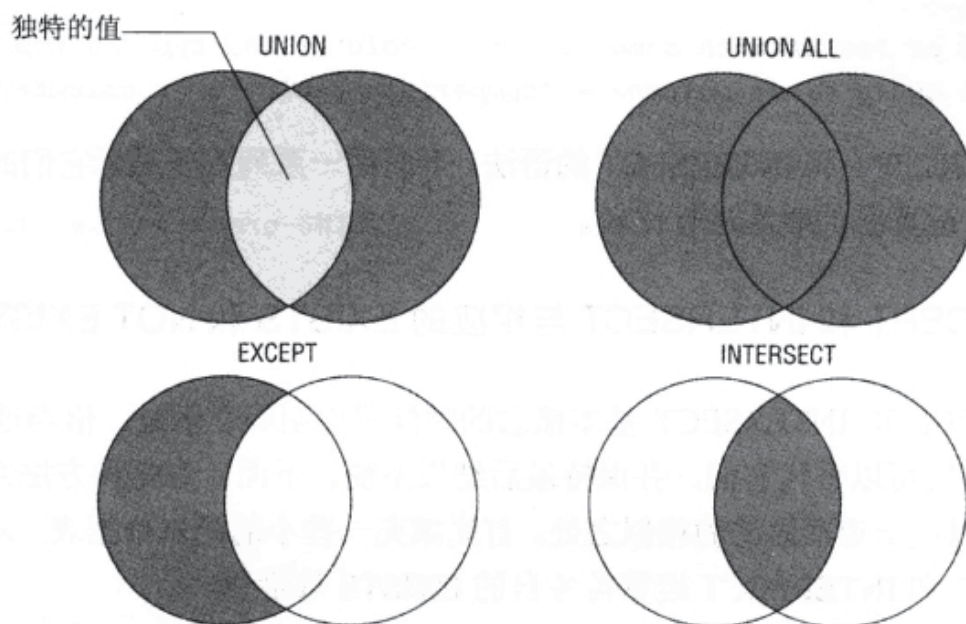


图 3-1

3.6.1 EXCEPT

EXCEPT 运算符提供存在于左边集合同时不存在于右边集合的数据。语法比较直白，而且功能基本类似于 UNION：

```
<table or tabular result>
EXCEPT
<table or tabular result with same number of columns and type as top query>
```

我们之前回顾过 EXISTS 和 NOT EXISTS，这里可以将其转换为 NOT EXISTS 的等价语句，其逻辑如下所示：

```
<base query >
WHERE NOT EXISTS
  (SELECT 1
   FROM <table or result with same number of columns and type as top query>
   WHERE <base query first column> = <comparison table first column> [, ...])
```

在下面介绍 INTERSECT 语法时，将用一个例子介绍它。

3.6.2 INTERSECT

INTERSECT 运算符提供 INTERSECT 左右双方匹配的数据。借助于 EXCEPT，INTERSECT 的语法还很直白而且工作方式类似于 UNION：

```
<table or tabular result>
INTERSECT
<table or tabular result with same number of columns and type as top query>
```

再次，可以将它转换为 EXISTS 语句(这次没有 NOT)，其逻辑如下所示：

```
<base query >
WHERE EXISTS
```



```
(SELECT 1
FROM <table or result with same number of columns and type as top query>
WHERE <base query first column> = <comparison table first column> [, ...])
```

现在看过了 EXCEPT 和 INTERSECT 的语法，下面用一系列例子展示它们的运行过程并将它们与基于 EXISTS 运算符的版本进行比较。

3.6.3 比较 EXCEPT 和 INTERSECT 与相应的 EXISTS 和 NOT EXISTS 语句

在讨论 EXCEPT 和 INTERSECT 基本概念的时候我曾明确表示过，恰当地使用 EXISTS 或 NOT EXISTS 运算符可以替代它们，并保持最后结果不变。下面针对每种方法给出一个带有简单 UNION 的例子，以便于观察语法的相似之处。首先填充一些小的测试数据表，随后观察 UNION，接着查看 EXCEPT 和 INTERSECT 运算符各自的 EXISTS 等价物。

```
SET NOCOUNT ON; -- Eliminate the row counts after each query to save space

-- Create our test tables and populate them with a few relevant rows
CREATE TABLE UnionTest1
(
    idcol int IDENTITY,
    col2 char(3),
);

CREATE TABLE UnionTest2
(
    idcol int IDENTITY,
    col4 char(3),
);

INSERT INTO UnionTest1
VALUES
    ('AAA'),
    ('BBB'),
    ('CCC');

INSERT INTO UnionTest2
VALUES
    ('CCC'),
    ('DDD'),
    ('EEE');

PRINT 'Source and content of both tables: ';
PRINT '';

SELECT 1 AS SourceTable, col2 AS Value

FROM UnionTest1

UNION ALL
```

```
SELECT 2, col4
FROM UnionTest2;

PRINT 'Results with classic UNION';
SELECT col2
FROM UnionTest1

UNION

SELECT col4
FROM UnionTest2;

PRINT 'Results with EXCEPT';
PRINT '-----';

SELECT col2
FROM UnionTest1

EXCEPT

SELECT col4
FROM UnionTest2;

PRINT 'Equivalent of EXCEPT but using NOT EXISTS';
PRINT '-----';

SELECT col2
FROM UnionTest1 ut1
WHERE NOT EXISTS
  (SELECT col4 FROM UnionTest2 WHERE col4 = ut1.col2);

PRINT 'Results with INTERSECT';
PRINT '-----';

SELECT col2
FROM UnionTest1
INTERSECT
SELECT col4
FROM UnionTest2;

PRINT 'Equivalent of INTERSECT but using EXISTS';
PRINT '-----';

SELECT col2
FROM UnionTest1 ut1
WHERE EXISTS
  (SELECT col4 FROM UnionTest2 WHERE col4 = ut1.col2);
```



```
-- Clean up after ourselves
DROP TABLE UnionTest1;

DROP TABLE UnionTest2;

SET NOCOUNT OFF; -- Don't forget to turn this back to the default!
```

下面花点时间看看返回的结果——集中精力观察 EXCEPT 和 INTERSECT 及其各自相关的 EXISTS 等价物返回的结果。

首先，检查 EXCEPT 运算符及其相关的 NOT EXISTS 等价物返回的结果：

```
Results with EXCEPT
-----
col2
----
AAA
BBB

Equivalent of EXCEPT but using NOT EXISTS
-----
col2
----
AAA
BBB
```

可以看到，结果是相同的。但是，值得注意的是查询计划不同。例如，在我的系统中，EXCEPT 的成本(在介绍性能调整的章节中，可以了解更多信息)是 NOT EXISTS 方法的两倍多。如果对性能非常敏感，你也许要在更贴近于自己应用程序的一组数据上对两种方法进行测试，观察两者的结果。

提示：

在研究 INTERSECT 的时候，也会发现 EXISTS 版本比 EXCEPT/INTERSECT 等价物表现得更好。在我个人看到的以及 Web 上的每个例子中，EXISTS 方法要么效率更胜一筹，要么与 EXCEPT/INTERSECT 方法平分秋色，我从未见过 EXCEPT/INTERSECT 方法胜出。

这是不是意味着不应该使用 EXCEPT 和 INTERSECT 方法呢？也许吧。但是我相信事情没有那么简单。例如，在你开发的代码中，哪些更容易阅读？哪些更便于理解？如果你看到性能只差了一点点或者“几乎相差无几”，那么你可能对使用 EXCEPT 和 INTERSECT 更加感兴趣，因为它们与随后阅读代码的那些人所期望的结果更加匹配。EXISTS 和 NOT EXISTS 并不难，只是它们的用法很多，因此不太直观；正确的选择往往只是个人观点问题。

现在观察 INTERSECT 结果：

```
Results with INTERSECT
-----
col2
----
CCC
```

```

Equivalent of INTERSECT but using EXISTS
-----
col2
----
CCC

```

结果仍然很匹配。我们可以使用 EXISTS 运算符再现 INTERSECT 的功能。

提示：

类似于 EXCEPT，EXISTS 表现得更好(大约只需要 EXCEPT 的 30% 的成本)。这个结果会随查找数据的不同而变化。因此我经常说“你的里程数会变化”，意思是一一定要在自己的环境中测试它的影响。

注意：

总之，EXISTS 方法至少不比 EXCEPT/INTERSECT 方法差。但是后者更便于阅读。一定要考虑自己的特殊情况后再在两者之间做出选择。

3.7 通用表表达式(CTE)

SQL Server 2005 中第一次引入了通用表表达式(Common Table Expression, CTE)。它们提供了一种方法按姓名引用临时结果集，并将其视为表加以利用(尽管这在本质上是临时的、虚拟的)。也许最酷的是在使用它们之前先定义它们，这样就不用分几步来完成物理存储和表的重新引用了(因为你可能会处理临时表——或者甚至是一个表变量)。因为 SQL Server 能够平衡 CTE 和查询之间的功能，所以将其看作是逻辑操作的一部分而不是看作是一系列孤立的行为加以利用，可以得到令人满意的性能。在最简单的情况下，CTE 类似于运行中创建的视图，但是 CTE 还可以完成许多视图无法完成的工作(具体实例请参考下一节“递归查询”)

CTE 的基本语法使用后接名称和定义的关键字 WITH:

```

WITH <CTE name> [ ( <column name> [,...n] ) ]
AS
( <query returning tabular data> )
<statement that will make use of the CTE>

```

定义好 CTE 之后，就可以像表一样按名称进行引用了。

注意，虽然 CTE 可以嵌套，而且 CTE 可以引用父 CTE，但是同一时间不能有完全独立的 CTE，同时在嵌套 CTE 中也不能有前向引用。确实，CTE 声明之后必须跟随将要使用 CTE 的语句。

因此，作为使用 CTE 的一个例子，可以用 CTE 引用替代部分早先派生的表。

```

USE AdventureWorks2008;

WITH pumps (BusinessEntityID)
AS
(
    SELECT sc.PersonID AS BusinessEntityID
    FROM Sales.Customer sc
    JOIN Sales.SalesOrderHeader AS soh

```



```

        ON sc.CustomerID = soh.CustomerID
    JOIN Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
    JOIN Production.Product AS p
        ON sod.ProductID = p.ProductID
    WHERE p.Name = 'Minipump'
)

SELECT DISTINCT pp.FirstName, pp.LastName
FROM Person.Person AS pp
JOIN pumps
    ON pp.BusinessEntityID = pumps.BusinessEntityID

JOIN ( SELECT sc.PersonID AS BusinessEntityID
      FROM Sales.Customer sc
      JOIN Sales.SalesOrderHeader AS soh
          ON sc.CustomerID = soh.CustomerID
      JOIN Sales.SalesOrderDetail AS sod
          ON soh.SalesOrderID = sod.SalesOrderID
      JOIN Production.Product AS p
          ON sod.ProductID = p.ProductID
      WHERE p.Name = 'AWC Logo Cap') caps
    ON pp.BusinessEntityID = caps.BusinessEntityID;

```

注意，可以去掉第一个得到的表并用 CTE 引用替代它。但是，因为一次只能生成一个 CTE 引用，所以不能取代 Caps 派生的表。也就是说，可以取代 pumps 或者 caps，但是不能两个都取代。

值得注意的是 CTE 中不能使用某些结构，其中包括：

- COMPUTER 和 COMPUTE BY
- ORDER BY
- INTO
- FOR XML、FOR BROWSE 和 OPTION 查询子句

乍一看，考虑到这么多的限制，CTE 似乎价值很小，但是一旦开始利用递归查询(离开了 CTE，这将是不可行的)，它们就会显示出强大的力量。正如之前说过的，下面开始学习递归查询。

3.8 递归查询

在关系系统中，需要更多技巧的是层次数据。微软在过去发布的两个版本中已经做了很多的努力来减轻这一领域的压力。递归查询就是其中功能非常强大的一部分。当查询或一段代码直接或间接地调用自己时，就可以将其视为递归。我们很久以前就可以使用递归存储过程和函数了，但是递归查询的概念却是在 SQL Server 2005 发布之后才出现的。

在此版本新增内部层次数据类型(第 7 章中将深入讨论这种新的 HierarchyID 数据类型)前，绝大多数的层次数据都被存储在一元关系中——即，父列和子列都在同一个表中。在这种一元的关系中，最需要观察递归，其中层次数据在数据结构中表示为“不规则的”。即，每条树枝的深

度可能不同，所以在找到层次结构的底部之前必须递归——不论它究竟有多深。递归查询使之成为可能。

使用结构合理的 CTE 就可以实现递归查询。递归 CTE 至少需要两个主要部件：一个基础或者“锚”成员，以及一个递归成员。锚成员确立了可以添加其他查询数据的基础。递归成员处理重复的调用并提供递归检查。

下面将看到一个典型的不规则层次——雇员报告链。为此，首先创建一个 AdventureWorks2008 Employees 表，其中报告结构用旧的模式类型表示(2008 版的 AdventureWorks 使用了新的 HierarchyID 数据类型)。我们将使用现有 Employees 表中的数据生成该表，这样我们的数据就可以与 AdventureWorks2008 数据库中的其他地方使用的数据方便地匹配了。

```
CREATE TABLE HumanResources.Employee2
(
    BusinessEntityID int NOT NULL PRIMARY KEY,
    ManagerID int NULL,
    JobTitle nvarchar(50) NULL
);

INSERT INTO HumanResources.Employee2
SELECT hre.BusinessEntityID,
       (SELECT BusinessEntityID
        FROM HumanResources.Employee hre2
        WHERE hre.OrganizationNode.GetAncestor(1) = hre2.OrganizationNode
        ) AS ManagerID,
       JobTitle
FROM HumanResources.Employee hre;
```

这样会向新表 HumanResources.Employee2 返回 290 行数据，我们将在下面的 CTE 例子中使用这些数据。

既然已经知道有些雇员(低级别的“C”级员工)直接向首席执行官汇报工作，而经理们向那些执行官汇报工作，主管们向经理汇报，依此类推，这就表明我们的工作已经开始了。管理链的实际深度随着各个部门和组的不同而不同。我们可以使用一个递归查询来沿着那条链徐徐前进。

首先需要为这个层次建立一个根(或者“锚点”)。这种情况下，很明显就是首席执行官(没有人比他的职位更高)，但是这里这样做的原因是因为他们不用向任何其他雇员汇报：

```
-- Establish the "Anchor Member"
-- This essentially defines the top node of the
-- recursion hierarchy
SELECT hre.ManagerID,
       hre.BusinessEntityID,
       hre.JobTitle,
       hredh.DepartmentID,
       0 AS Level
FROM HumanResources.Employee2 AS hre
JOIN HumanResources.EmployeeDepartmentHistory AS hredh
ON hre.BusinessEntityID = hredh.BusinessEntityID
AND hredh.EndDate IS NULL -- Current employees only!
```



```
WHERE hre.ManagerID IS NULL;
```

现在，我们要加入向这个根节点汇报的那些员工，随后沿着树向下递推，直至到达底部为止。我们将这些结果与那些为根节点得到的结果合并起来：

```
UNION ALL
-- Define the piece that actually recurses
SELECT hre.ManagerID,
       hre.BusinessEntityID,
       hre.JobTitle,
       hredh.DepartmentID,
       r.Level + 1
FROM HumanResources.Employee2 AS hre
JOIN HumanResources.EmployeeDepartmentHistory AS hredh
  ON hre.BusinessEntityID = hredh.BusinessEntityID
AND hredh.EndDate IS NULL -- Current employees only!
JOIN Reports AS r
  ON hre.ManagerID = r.BusinessEntityID
```

现在，把它们放在一起，然后创建一条语句使用我们的 CTE。可以在调用语句中加入 WHERE 子句，这样可以根据此处的需要对汇报信息按照组、部门或职位筛选——例如：

```
USE AdventureWorks2008;
GO

-- Establish the CTE foundation for the recursion
WITH Reports (ManagerID, BusinessEntityID, JobTitle, DepartmentID, Level)
AS
(
    -- Establish the "Anchor Member"
    -- This essentially defines the top node of the
    -- recursion hierarchy
    SELECT hre.ManagerID,
           hre.BusinessEntityID,
           hre.JobTitle,
           hredh.DepartmentID,
           0 AS Level
    FROM HumanResources.Employee2 AS hre
    JOIN HumanResources.EmployeeDepartmentHistory AS hredh
      ON hre.BusinessEntityID = hredh.BusinessEntityID
    AND hredh.EndDate IS NULL -- Current employees only!
    WHERE hre.ManagerID IS NULL
    UNION ALL
    -- Define the piece that actually recurses
    SELECT hre.ManagerID,
           hre.BusinessEntityID,
           hre.JobTitle,
           hredh.DepartmentID,
           r.Level + 1
```

```

FROM HumanResources.Employee2 AS hre
JOIN HumanResources.EmployeeDepartmentHistory AS hredh
    ON hre.BusinessEntityID = hredh.BusinessEntityID
    AND hredh.EndDate IS NULL -- Current employees only!
JOIN Reports AS r
    ON hre.ManagerID = r.BusinessEntityID

```

```

)

-- Code to get it all started.
SELECT ManagerID, BusinessEntityID, JobTitle, Level
FROM Reports r
JOIN HumanResources.Department AS dp
    ON r.DepartmentID = dp.DepartmentID
WHERE dp.GroupName LIKE '%Admin%'
ORDER BY Level, ManagerID, JobTitle;
GO

```

注意 CTE 不控制返回的组名；相反地，这是由调用查询驱动的。但是 WHERE 子句是在执行之前被并入计划的，所以将根据调用查询的特定标记以不同方式对查询进行优化。

下面看一看查询结果：

ManagerID	BusinessEntityID	JobTitle	Level
NULL	1	Chief Executive Officer	0
1	234	Chief Financial Officer	1
1	263	Information Services Manager	1
25	227	Facilities Manager	2
...			
...			
264	266	Network Administrator	3
228	229	Janitor	4
228	230	Janitor	4
228	231	Janitor	4
228	232	Janitor	4

(35 row(s) affected)

你可能会问“这是哪一级？”为了让你感觉到每一行的深度都与整体层次有关，这部分内容是我在这里随便插入的。这里也可以不考虑它。

这里要理解的关键内容是递归查询目前不但是可行的，而且还是比较简单的。诀窍就是要理解自己的根节点和在该锚点上构建层次的方式。

3.9 合并

在以前版本的 SQL Server 中，听到“合并”一词，一般会联想到合并复制。但是在 SQL Server 2008 中，我们对合并一词和 DML 语句有了全新的理解。

有了 MERGE，我们可以在一个整体操作中并入多条 DML 操作语句(INSERT、UPDATE 和

DELETE), 改进性能(他们可以共享许多相同的物理操作)并简化事务。MERGE 使用有点类似于 CTE 的特殊 USING 子句。USING 子句中的结果集在一定条件下可以用于 INSERT、UPDATE 和 DELETE 语句。基本的语法如下所示:

```
MERGE <target table> [AS <alias>]
USING
(
    <source query>
)
WHEN {[NOT] MATCHED | <expression> THEN
    <action statement>
[<additional WHEN clauses>, [...n]]
```

下面将给出一个接收运货清单的例子。假设这里有一个特殊的累计表记录了销售报告。每天都要运行查询, 向月清单中加入新的销售记录。在每个月第一天的晚上执行查询似乎不需要多加考虑, 因为此时没有其他的累计月销售记录, 需要将当天的销售记录累计和插入。但是, 在第二天, 情况发生了变化: 我们需要像第一天一样累计并插入新的记录, 还需要更新现有的记录(更新那些当月已经售出的产品)。

下面看一看 MERGE 如何一步管理所有的操作。在此之前, 首先需要创建一个累计表:

```
USE AdventureWorks2008;

CREATE TABLE Sales.MonthlyRollup
(
    Year smallint NOT NULL,
    Month tinyint NOT NULL,
    ProductID int NOT NULL
    FOREIGN KEY
        REFERENCES Production.Product(ProductID),
    QtySold int NOT NULL,
    CONSTRAINT PKYearMonthProductID
    PRIMARY KEY
        (Year, Month, ProductID)
);
```

这是一个非常简单的月累计表——它可以非常方便地累计某种产品在某年和某月的总销售额。但是, 为了利用它, 我们需要定期用明细表中的累计数据对它进行填充。为此, 我们将要使用 MERGE。

首先, 创建一个结果集, 计算出哪些行是我们累计所需要的。为此, 我们把精力放在 2003 年 8 月, 并且开始查询当月的第一天:

```
SELECT soh.OrderDate, sod.ProductID, SUM(sod.OrderQty) AS QtySold
FROM Sales.SalesOrderHeader soh
JOIN Sales.SalesOrderDetail sod
    ON soh.SalesOrderID = sod.SalesOrderID
WHERE soh.OrderDate >= '2003-08-01'
    AND soh.OrderDate < '2003-08-02'
GROUP BY soh.OrderDate, sod.ProductID;
```

这样便按 ProductID 返回查找范围内每天的总销售额。

提示：

这里存在一处陷阱。设置 GROUP BY 使用 OrderDate，但是 OrderDate 的数据类型是 datetime 而不是 date。如果订单来的时候就带有自己的实际时间，那么它就会破坏我们的假设——所有的订单都会完美地分组为一个日期。如果正处于生产环境中，那么希望将 OrderDate 强制转换为 date 数据类型或者使用 DATEPART 确保按天分组而不是按时间分组。

有了它，我们就可以建立自己的合并了：

```

MERGE Sales.MonthlyRollup AS smr
USING
(
    SELECT soh.OrderDate, sod.ProductID, SUM(sod.OrderQty) AS QtySold
    FROM Sales.SalesOrderHeader soh
    JOIN Sales.SalesOrderDetail sod
    ON soh.SalesOrderID = sod.SalesOrderID
    WHERE soh.OrderDate >= '2003-08-01' AND soh.OrderDate < '2003-08-02'
    GROUP BY soh.OrderDate, sod.ProductID
) AS s
ON (s.ProductID = smr.ProductID)
WHEN MATCHED THEN
    UPDATE SET smr.QtySold = smr.QtySold + s.QtySold
WHEN NOT MATCHED THEN
    INSERT (Year, Month, ProductID, QtySold)
    VALUES (DATEPART(yy, s.OrderDate),
            DATEPART(m, s.OrderDate),
            s.ProductID,
            s.QtySold);

```

注意：

MERGE 语句后必须要有分号。虽然为了向后兼容，绝大多数的 SQL 语句后的分号是可选的，但是你会发现越来越多的语句的末尾处都必须用它作为分隔符，对诸如 MERGE 之类的包含多个部分的语句来说尤其如此。

假设没有修改 AdventureWorks2008 中的数据，当运行它时，应该有 192 行受到影响。现在，既然 Sales.MonthlyRollup 表是空的，那么就不会有匹配，因此所有的数据都会被插入。可以通过查询 Sales.MonthlyRollup 表来验证它：

```

SELECT *
FROM Sales.MonthlyRollup;

```

这样返回期望的 192 行：

Year	Month	ProductID	QtySold
2003	8	707	242

2003	8	708	281
2003	8	711	302
...			
...			
2003	8	997	43
2003	8	998	138
2003	8	999	103

(192 row(s) affected)

基本 SELECT 语句中支持 MERGE 的每一行最终都被插入表中。不过, 接下来看看当月的第二天:

```

MERGE Sales.MonthlyRollup AS smr
USING
(
    SELECT soh.OrderDate, sod.ProductID, SUM(sod.OrderQty) AS QtySold
    FROM Sales.SalesOrderHeader soh
    JOIN Sales.SalesOrderDetail sod
    ON soh.SalesOrderID = sod.SalesOrderID

    WHERE soh.OrderDate >= '2003-08-02' AND soh.OrderDate < '2003-08-03'

    GROUP BY soh.OrderDate, sod.ProductID
) AS s
ON (s.ProductID = smr.ProductID)
WHEN MATCHED THEN
    UPDATE SET smr.QtySold = smr.QtySold + s.QtySold
WHEN NOT MATCHED THEN
    INSERT (Year, Month, ProductID, QtySold)
    VALUES (DATEPART(yy, s.OrderDate),
            DATEPART(m, s.OrderDate),
            s.ProductID,
            s.QtySold);

```

我们更新运行这段程序的时间(在当月的第二天运行), 而且运行后将返回 38 行结果:

(38 row(s) affected)

但是这次有些不同。表中已经有一些与新的销售记录匹配的数据行了。我们知道有 38 行受到影响, 不过我们是怎么影响它们的呢? 针对表重新运行 SELECT:

```

SELECT *
FROM Sales.MonthlyRollup

```

这次只返回 194 行而不是 230 行(192 加 38)。实际上 38 行中有 36 行都是重复的销售记录, 并因此被视为更新处理而不是当作插入处理。有两行(ProductIDs 882 和 928)是当月之前从未出售过产品的销售记录, 因此必须被当作新行插入——一次数据传输, 而运行两条语句的等价语句。

我们还将完成相似的操作, 基于匹配或不匹配的条件来确定删除行。

3.10 利用外部调用完成复杂操作

有时，我们需要获得 SQL Server 外部的信息。事实上，对于绝大多数安装来说，那些信息无法从 SQL Server 内部获得。通常，由一个客户端或中间层组件来挑选哪些是 SQL Server 需要的，哪些是外部源需要的。

在许多方面，这样做还算不错——毕竟，挂起数据库服务器等待外部调用，从好的方面看是有风险的，从坏的方面看是致命的。没有人知道在调用返回前需要等待多久甚至是否有这样的调用？在数据库服务器中挂起进程的风险是相当高的。

前面讲述的是大多数的安装，这就暗示着有一些安装并非如此——而且它们的确有所不同。可以使用几种不同的方法来实现。

首先，这里有一个扩展存储过程的方法。这些是使用特殊的 SQL Server 库可以在 C 中创建的 DLL。它们与 SQL Server 在一个进程中运行，因而速度非常快(假设 DLL 的编写者水平高超)，但有一个问题——外部调用。如果调用的外部过程会实时返回，对我们非常有益。另外还有一个问题是总体安全性。由于运行在 SQL Server 的进程之中，因此，如果 DLL 崩溃，SQL Server 也将崩溃(若你在销售软件，那么当你的产品令客户的 SQL Server 安装出现故障时，你一定能想象客户将做何反应)。最后，也是非常重要的一点，只有极少数人掌握了如何编写 DLL 的诀窍。

另一种解决方法是在 OLE/COM 时代中加入 SQL Server 的。sp_CreateOAMethod 系列存储过程允许实例化一个 COM 对象并对它进行调用。它们使用变量来回传递数据，而且通常都在过程之外运行。它们的灵活性差，而且速度非常慢。

随着 .NET 出现，SQL Server 可以响应 CLR 语言，我们进入一个新的世界。你可以使用任何 .NET 语言编写脚本，并且可以实例化需要的对象以完成工作。可以创建用户定义函数调用外部进程——例如，与其他无法直接连接的在线系统进行交叉通信。想象一下，允许 SQL Server 使用从 Web 服务中搜集的信息，并将数据合并到目的查询中？这令人兴奋吧！

这样做有很多种可能性。但是，你必须意识到，外部调用始终是外部调用！一旦依赖于系统外部的事物，就必须受外部系统支配。对于这样的调用要非常、非常小心。

注意：

应当把外部调用视为一种终极手段。它在安全上存在风险(假设有人冒充外部源的话，风险如何？)，同时在性能上也存在巨大风险。处理这一领域一定要小心谨慎。

3.11 性能考虑

本章前面的讲述已经涉及了一些宏观级别上“什么是最好的”问题，但是，就像生活中大多数事情一样，事情并不都这么简单。这里想要提供查询性能的某种快速参考。我将帮助你根据特定的场合选择恰当的查询。

提示：

现在，又是时候进行老生常谈了。这一次要解决的问题是总括规则(blanket rule)的总括运用。本节将讨论事物通常的工作方式。这里的通常一词非常关键。在 SQL 中，极少有百分之百的

情况下都正确的规则。在充斥着特例的世界里，SQL 无疑是登峰造极的——试图描述 SQL Server 中的性能时，特例随处可见。

简言之，必须评估特定查询的性能有多重要。如果性能是至关重要的，那么别太在意这些规则——而应当只把它们作为一个起点，接下来就是不断地测试、测试、再测试!!!

使用联结、子查询或其他方法

我在本章前面提到过曾与一位同事激烈争论过决定使用联结还是子查询的问题。而且，可以想见，当两个人对自己的观点深信不疑时，在一定程度他们都是正确的(在一定程度上也都是错误的)。

由来已久的传统观念认为子查询应该尽可能地使用联结。这个观点在某些情况下是十分正确的。事实上，它与大量的因素有关。表 3-1 中讨论了决定性能权衡的某些因素，以及倾向于何种解决方案。

表 3-1

情 况	比较适当的方式
对于外部查询中的每一行，从子查询返回的值是相同的	预查询。声明变量，然后选中需要的值放入变量中，这样可以只先执行一次即将形成的子查询，而不用对外部表中的每一条记录执行一次子查询。事实上，SQL Server 中的优化器智能程度很高，它一旦察觉到处于这样的情况，就会为你执行预查询，但是别依赖它。为了更加可靠，在你意识到出现这样的情形的时候，请执行自己的预查询
两个表都较小(例如，只有 10 000 条记录或者更少)	子查询。我不清楚确切的原因，不过我已经对它做过多次测试，几乎每次都可以得到这样的结果。我怀疑这是由于查找到的所有数据局限于一两个数据页的时候，较低的查找开销与连接对比的结果。每个版本的优化器都意识到了这一点，智能程度不断提高。所以，有时候你会发现这两种选项返回的查询计划是相同的
考虑了所有的条件后，匹配将只返回一个值	子查询。同样，与必须联结整张表相比，只找寻一条记录并替换它花费的开销要少很多
考虑了所有条件后，匹配将只返回相当少的值，而且查询列上没有索引	子查询。通常，单独的一次或者甚至数次查找所花费的开销都比散列联结少
查找表较小，但基表很大	如果可以的话，请使用嵌套子查询。如果相对于子查询来说，则使用联结。使用子查询的时候，只需要查找一次，所以开销相对较小。但是，如果使用的是相关子查询，查找会循环许多次——此时，大多数情况下联结会是更好的选择
相关子查询与联结	联结。本质上，相关子查询将造成嵌套循环的情形。这样的开销很大。在多数情况下，它远远快于游标，但是它比其他可能的选择方案要慢一些
派生表与其他选择	派生表通常会带来大量的开销，因此使用时必须谨慎。要知道，派生表只运行一次，随后就会驻留在内存中，因而，多数的开销存在于首次创建派生表的过程中以及没有索引的较大结果集上。它们可能很快，也可能很慢，这要视具体情况而定。在编码前要仔细考虑

(续表)

情 况	比较适当的方式
EXISTS 与其他选择	EXISTS。不必针对同样的条件多次查找。一旦找到了特定行的一个匹配值，将进入到下一个查找中，这可以大大减少开销
使用 CTE	CTE 被并入调用查询的查询计划中。一般来说，这意味着基本 CTE 对终端性能不会有太大影响
MERGE 与多语句	MERGE 允许在一次传输数据中完成分离的操作语句，并在适当的时候利用同样的锁。这样做一般会提高性能。不过要记住，对很多用户来说，它可能会产生一些很难阅读的代码

上面列出的只是一些突出的情况。必定存在不同的混合形式以及不计其数的特例。

注意：

有一点无论强调多少次都不过分，当你犹豫不决(或者甚至你并没有什么疑问，但是性能至关重要)时，应当对所有有竞争力的解决方案进行合理的测试。所谓的合理测试是指测试必须囊括用户执行代码可能遇到的所有典型场景。此外，对数据库和负载的测试等效于生产环境中经常出现的事物。总括的规则适合大多数情况，但是，也有例外。通过合理的测试，就可以确保做出正确的抉择。

3.12 小结

在使用 SQL 中所学到的基本查询或许覆盖了 80% 甚至更多你所遇到的查询情况，但是，这剩下的 20% 却可能让你不知所措。有时面临的问题是，是否能够找到一个可以完成任务的查询。有时出现的状况是，所拥有的特定查询或存储过程的性能无法让人满意。无论情况如何，都将遭遇到大量这样简单的查询和联结并不适用的情况。于是，你需要了解更多的知识，但愿本章中涵盖的选项能充实你的知识，使你能够应付那些棘手的情况。



第 4 章

XML 集成

对我来说，回顾 XML 的历史是非常有意思的。它之所以强大，在某种程度上是因为它的简洁性，因此它看上去变化不大。事实上，它的基本规则从来没有改变过——但是所有 XML 周边的事物(例如如何访问以 XML 格式存储的数据)都发生了很大的变化。同样，从 SQL Server 首次为 XML 提供支持特性至今，SQL Server 对 XML 的支持方式也已经发生了非常大的变化。

我在一本书中曾经将 XML 的支持作为“额外”的知识而讨论——这样做是很愚蠢的。我之所以犯这样的错误，只是因为 SQL Server 运行并不真正需要支持 XML。现在我真正的意识到，在 SQL Server 工作时不需要 XML 支持的情况其实并不多见。正是因为这样，为了回顾 XML 集成变为产品的过程，本书在以前版本的基础上(那里只是一种回想)，大大扩展了 XML 的覆盖范围。

近十年来，XML 得到了广泛的应用，成为数据设计的基本主流方案。当然，也有许多经过慎重考虑后设计出来的系统并没有使用任何的 XML 代码，但是大多数人在设计这些系统的时候都曾经考虑过“是否应该使用 XML”。网站中使用 XML 交换数据并简单存储诸如层次一类的数据——如果你不曾考虑过在自己的数据库应用程序中使用 XML，那可能你对自己的数据应用程序考虑得还不够周全。

总而言之，在本章中可以看到：

- XML 数据类型；
- XML 模式集合；
- 把关系数据表示为 XML 的方法；
- 对原来以 XML 格式存储的数据进行查询的方法(XQuery，微软的“XDL”语言，以及其他的方法)，微软的“XDL”语言是 XQuery 的变体。

它们中间的一些内容是相互嵌套的，因此下面看一看它们是如何混合使用的。

注意：

本章假设你已经具备基本的 XML 规则和结构方面的知识。如果没有这些基本知识，我强烈建议你在继续阅读本章之前，先阅读《XML 入门经典》(由清华大学出版社引进并出版)或者其他有关 XML 的书籍。记住，其他章节可能偶尔会引用本章的内容。

4.1 XML 数据类型

XML 数据类型是由 SQL Server 2005 率先引入的。这是将关系数据与 XML 数据结合在一起的历史分水岭。SQL Server 以 XML 的格式存放数据，并将它真正作为 XML 数据而识别。在以前的版本中，有很多处理 XML 数据的方式，但是所有方法都是基于字符数据的。现在 XML 终于作为 XML 而被识别了，同时也开启了从索引到数据验证的许多新的可能性。

这里的差异很大。在讨论 XML 数据类型所包含内容的时候，需要介绍包含以下内容在内的相关事物：

模式集合(Schema Collection)——XML 的一个核心概念是允许 XML 同模式文档关联。XML 模式定义了允许用户确认 XML 是否“有效”的规则(即，是否与特定类型的 XML 文档的规则匹配)。SQL Server 中的 XML 模式集合是一种存储模式而且允许 SQL Server 验证文档的方式。可以将 XML 数据实例(例如列数据或变量)与 XML 模式相关联，为了确定这些 XML 是否有效，SQL Server 会在每一个 XML 实例中应用相应的模式。

强约束(enforcing constraint)——这样的概念已经讨论过了，在关系数据系统中，将数据插入数据表之前，首先要求列满足某种条件。那么应该怎么处理 XML 呢？XML 允许将多块离散的数据存储在一列中——应该如何验证这些离散的数据块呢？XML 数据类型理解 XML，而且不允许直接的约束定义，针对 XML 中的特定节点可以使用封装函数(存储过程或触发器形式)定义约束。

XML 数据类型方法(XML data type method)——在使用类型为 XML 的列或变量的时候，可以使用该数据类型固有的几种方法。例如，可以测试是否存在某个节点或某个属性，可以执行 XDL(微软定义的一种 Xquery 扩展，Xquery 允许修改数据)，或对特定的节点或者属性的值进行查询。

下面更加具体地阐述这部分内容。

4.1.1 定义 XML 数据类型的列

前面已经学习了 XML 列的最基本定义。例如，查看 AdventureWorks2008 数据库中 Production.ProductModel 表的最基本定义，会发现它类似于下面的一段代码：

```
CREATE TABLE Production.ProductModel (
    ProductModelID      int IDENTITY(1,1)          NOT NULL,
    Name                 dbo.Name                   NOT NULL,
    CatalogDescription xml                          NULL,
    Instructions         xml                        NULL,
    rowguid              uniqueidentifier ROWGUIDCOL NOT NULL,
    ModifiedDate         datetime NOT NULL,
    CONSTRAINT PK_ProductModel_ProductModelID PRIMARY KEY CLUSTERED
    (
        ProductModelID ASC
    );
```

那么，自己思考一下我们从这里的两个 XML 列中得到了什么。

(1) 已经将它们定义为 XML，所以可以使用相应的 XML 数据类型方法(很快就会介绍它)。

(2) 已经允许它们为 NULL, 但是也可以轻易地选择 NOT NULL 来约束它们。不过, 要注意, 如果设置了 NOT NULL, 会强制相应行的列中必须包含数据(而非数据是否有效)。

(3) 此处的 XML 被视为“未类型化的 XML(non-typed XML)”。也就是说, 因为没有将其与任何模式相关联, SQL Server 不知道此 XML 的行为是否“有效”。

第一条提示在任何被定义为 XML 数据类型的列中, 保存的数据不再仅仅是纯文本数据。我们将在随后的 XML 数据类型一节中看到更多的这方面的内容。

其次, 对于 SQL Server 中的任何数据类型, 都可以指定是否允许相应的列为 NULL。

这里真正改变的是, 在定义的时候就可以确定将 XML 列定义为类型化的 XML 还是非类型化的 XML。在前面的例子中定义的非类型化, 意味着 SQL Server 对列中存储的任何 XML 数据都知之甚少。因此, 也不能做更多的事情来验证其有效性。如果将该列设置为定义类型化的 XML, 就可以提供更多的定义, 提供确认方法验证列中的 XML 数据是否“有效”。

AdventureWorks2008 数据库中已经存在模式集合, 它与已经放在表中的这两个 XML 列的验证相匹配, 所以, 下面看一看如何修改 CREATE 语句使其支持类型化 XML:

```
CREATE TABLE Production.ProductModel (
    ProductModelID      int      IDENTITY(1,1) NOT NULL,
    Name                dbo.Name      NOT NULL,
    CatalogDescription xml
        (CONTENT Production . ProductDescriptionSchemaCollection ) NULL,
    Instructions         xml
        (CONTENT Production . ManuInstructionsSchemaCollection ) NULL,
    rowguid             uniqueidentifier ROWGUIDCOL NOT NULL,

    ModifiedDate        datetime NOT NULL,
    CONSTRAINT PK_ProductModel_ProductModelID PRIMARY KEY CLUSTERED
    (
        ProductModelID ASC
    );
```

在真实的 AdventureWorks2008 例子中, 定义了这里演示的方法。为了向 Production.ProductModel 插入一条记录, 必须将 CatalogDescription 和 Instructions 字段设置为空, 要么必须提供可以通过相关模式测试证明为有效的 XML 数据。

4.1.2 XML 模式集合

XML 模式集合(XML Schema Collection)是数据库中的一个或多个模式文档的有名称的持久存储形式。该名称用于处理模式集。引用这个集合可以指示类型化 XML 列或者变量必须与该集合中所有的模式相匹配才是有效的。

这里可以使用内置的 XML_SCHEMA_NAMESPACE()函数查看现有的模式集合。下面给出了其语法:

```
XML_SCHEMA_NAMESPACE( <SQL Server schema> , <xml schema collection> , [<namespace>] )
```

这里有一处地方容易混淆, 所以表 4-1 给出了这些参数的含义:

表 4-1

参 数	说 明
SQL Server 模式	这是你的关系数据库模式(不要和 XML 模式混为一谈)。例如, 对于表 Production.ProductModel 而言, Production 是其关系模式, 对于 Sales.SalesOrderHeader 表, Sales 是其关系模式
XML 模式集合	创建 XML 模式集合的时候会使用这个名称。在前面创建表的例子中, 使用 XML 模式集合 ProductDescriptionSchemaCollection 和 ManuInstructionSchemaCollection
名称空间	这是一个可选项, 在 XML 模式集合中指定的名称空间。记住 XML 模式集合中可以包含多个模式文档——这样会返回任何属于该指定名称空间中的内容

因此, 必须编写下面的查询才可以针对 Production.ProductDescriptionSchemaCollection 模式集合使用这些参数:

```
SELECT XML_SCHEMA_NAMESPACE('Production','ManuInstructionsSchemaCollection')
```

这样会得到一大堆未经格式化的 XML:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:t="http://schemas.microsoft.com/sqlserver/2004/07/adventureworks/
ProductModelManuInstructions"
targetNamespace="http://schemas.microsoft.com/sqlserver/2004/07/adventureworks/
ProductModelManuInstructions"
elementFormDefault="qualified"><xsd:element name="root"><xsd:complexType
mixed="true"><xsd:complexContent mixed="true"><xsd:restriction
base="xsd:anyType"><xsd:sequence><xsd:element name="Location"
maxOccurs="unbounded"><xsd:complexType mixed="true"><xsd:complexContent
mixed="true"><xsd:restriction base="xsd:anyType"><xsd:sequence><xsd:element
name="step" type="t:StepType" maxOccurs="unbounded"
/></xsd:sequence><xsd:attribute name="LocationID" type="xsd:integer"
use="required" /><xsd:attribute name="SetupHours" type="xsd:decimal"
/><xsd:attribute name="MachineHours" type="xsd:decimal" /><xsd:attribute
name="LaborHours" type="xsd:decimal" /><xsd:attribute name="LotSize"
type="xsd:decimal"
/></xsd:restriction></xsd:complexContent></xsd:complexType></xsd:element></xsd:
sequence></xsd:restriction></xsd:complexContent></xsd:complexType></xsd:eleme
nt><xsd:complexType name="StepType" mixed="true"><xsd:complexContent
mixed="true"><xsd:restriction base="xsd:anyType"><xsd:choice minOccurs="0"
maxOccurs="unbounded"><xsd:element name="tool" type="xsd:string"
/><xsd:element name="material" type="xsd:string" /><xsd:element
name="blueprint" type="xsd:string" /><xsd:element name="specs"
type="xsd:string" /><xsd:element name="diag" type="xsd:string"
/></xsd:choice></xsd:restriction></xsd:complexContent></xsd:complexType></xsd:
schema>
```

SQL Server 删除了标记之间的所有空格, 因此如果你使用各种各样美观的缩进效果创建模式集合以便提高可读性, SQL Server 会为了更高的存储效率而删除它们。

注意:

在 Management Studio 中, 默认返回的文本结果只有 256 个字符。如果使用的是文本视图,

可以通过“工具”|“选项”|“查询结果”|“SQL Server”|“以文本格式显示结果”，改变可以显示的最大字符数。

4.1.3 创建、修改和删除 XML 模式集合

XML 模式集合的 CREATE、ALTER 和 DROP 的工作方式，同迄今为止使用 SQL Server 中其他对象中这类语句的方式基本相同。这里运行它们，但是请特别留心 ALTER 语句，因为它不同于我们使用过的其他 ALTER 语句。

1. 创建 XML 模式集合

同样地，下面的 CREATE 操作使用的是在本书中多次用到过的典型 CREATE<对象类型><对象名>语法，同时还使用了在创建存储过程、视图和其他结构化程度不高的对象时所用到的关键字 AS：

```
CREATE XML SCHEMA COLLECTION [<SQL Server schema>.] <collection name>
AS { <schema text> | <variable containing the schema text> }
```

例如，可以执行下面的语句创建一个模式集合，使其类似于 AdventureWorks2008 数据库中的 Production.ManuInstructionsSchemaCollection：

```
CREATE XML SCHEMA COLLECTION ProductDescriptionSchemaCollectionSummaryRequired
AS
    '<xsd:schema
targetNamespace-"http://schemas.microsoft.com/sqlserver/2004/07/adventureworks/
ProductModelWarrAndMain"
    xmlns-"http://schemas.microsoft.com/sqlserver/2004/07/adventureworks/ProductM
odelWarrAndMain"
    elementFormDefault-"qualified"
    xmlns:xsd-"http://www.w3.org/2001/XMLSchema" >
    <xsd:element name-"Warranty" >
    <xsd:complexType>
    <xsd:sequence>
    <xsd:element name-"WarrantyPeriod" type-"xsd:string" />
    <xsd:element name-"Description" type-"xsd:string" />
    </xsd:sequence>
    </xsd:complexType>
    </xsd:element>
    </xsd:schema>
    <xs:schema
targetNamespace-"http://schemas.microsoft.com/sqlserver/2004/07/adventureworks/Prod
uctModelDescription"
    xmlns-"http://schemas.microsoft.com/sqlserver/2004/07/adventureworks/ProductMod
elDescription"
    elementFormDefault-"qualified"
    xmlns:mstns-"http://tempuri.org/XMLSchema.xsd"
    xmlns:xs-"http://www.w3.org/2001/XMLSchema"
    xmlns:wm-"http://schemas.microsoft.com/sqlserver/2004/07/adventureworks/Product
ModelWarrAndMain" >
```



```

    <xs:import
namespace-"http://schemas.microsoft.com/sqlserver/2004/07/adventureworks/ProductModelWar
rAndMain" />
    <xs:element name-"ProductDescription" type-"ProductDescription" />
    <xs:complexType name-"ProductDescription">
        <xs:sequence>
            <xs:element name-"Summary" type-"Summary" minOccurs-"1" />
        </xs:sequence>
        <xs:attribute name-"ProductModelID" type-"xs:string" />
        <xs:attribute name-"ProductModelName" type-"xs:string" />
    </xs:complexType>
    <xs:complexType name-"Summary" mixed-"true" >
        <xs:sequence>
            <xs:any processContents-"skip"
namespace-"http://www.w3.org/1999/xhtml" minOccurs-"0" maxOccurs-"unbounded"
/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>'

```

注意:

名称空间声明的 URL 部分必须写在一行内。因为在印刷的时候，每行的字数受到了限制，所以本书中将它们显示为多行。请确保在你的代码中，将整个 URL 写在一行内。

这时的结果就像模式集合 `Production.ManuInstructionsSchemaCollection` 一样，但是我已经修改了模式，使得 `Summary` 元素是必需的而不是可选的。因为其基本结构是相同的，所以这里使用了相同的名称空间。

2. 修改 XML 模式集合

与其他的 `ALTER` 语句略有不同，在这里，仅限于向集合中添加新的片段。下面给出了它的语法：

```

ALTER XML SCHEMA COLLECTION [<SQL Server schema>.] <collection name>
    ADD { <schema text> | <variable containing the schema text> }

```

提示:

如果将其功能在随后的一个服务包中功能得到改进，我对此根本不会感到惊讶，不过，此时再次强调一下，这个工具是要向模式集合添加的，而不是从那里更改或删除的。

3. 删除 XML 模式集合

这是一个典型的“顾名思义”的事物，就像其他的 `DROP` 一样：

```

DROP XML SCHEMA COLLECTION [<SQL Server schema>.] <collection name>

```

另外，可以执行以下语句，以便删除之前创建的模式集合 `ProductDescriptionSchemaCollectionSummaryRequired`：

```

DROP XML SCHEMA COLLECTION ProductDescriptionSchemaCollectionSummaryRequired;

```

这样就可以用了。

4.1.4 XML 数据类型方法

XML 数据类型带有几个内置的方法。这些方法是 XML 数据类型特有的，当前其他的数据类型都没有类似的方法。因为这些方法基于不同的、但又是主流行业标准的 XML 访问方法，所以它们的语法稍有不同。调用这些方法的标准语法是：

```
<instance of xml data type>.<method>
```

可以利用的方法一共有 5 种：

- **.query** — 这是行业标准 XQuery 语言的一种实现。它允许用户通过运行 XQuery 格式的查询访问自己的 XML。XQuery 允许返回多个数据片段，而不是一个离散的值；
- **.value** — 它允许你访问指定元素或属性中的离散值；
- **.modify** — 这是微软自己的 XQuery 扩展。尽管 XQuery 被限制为只能请求数据(这不是一种修改语言)，但是 modify 方法扩展了 XQuery，进而允许对数据进行修改；
- **.nodes** — 用于将 XML 数据拆分成单独的，更关系化的行；
- **.exist** — 它非常类似于在标准的 SQL 中广泛使用的 IF EXISTS 子句，XML 数据类型方法 exist() 测试是否存在指定类型的数据。使用 exist() 时，该测试用于查看特定的元素或属性在 XML 的实例中是否拥有项。

1. .query(SQL Server 中的 XQuery 实现)

.query 是行业标准的 XQuery 语言的一种实现。其工作结果类似于 SQL 查询，只是结果匹配于 XML 数据节点，而不是关系化的行和列。

.query 需要一个可以在 XML 数据实例上运行的有效 XQuery 作为自己的参数。例如，可以执行下面的代码提取出 ProductID 为 66 的产品文档：

```
SELECT ProductModelID, Instructions.query('declare namespace
PI-"http://schemas.microsoft.com/sqlserver/2004/07/adventureworks/
ProductModelManuInstructions";
/PI:root/PI:Location/PI:step') AS Steps
FROM Production.ProductModel
WHERE ProductModelID = 66;
```

提示：

名称空间声明的 URL 部分必须写在一行内。因为在印刷的时候，每行的字数受到了限制，所以本书中将它们显示为多行。请确保在你的代码中，将整个 URL 写在一行内。

结果非常冗长。因此这里截断了结果右边的内容，不过还是可以看到由于截断了一部分内容，所以这里只得到了在 XML 层次里位于 Step 级或其下级的那些节点。

```
ProductModelID Steps
-----
66          <PI:step xmlns:PI-"http://schemas.microsoft.com/sqlser...
              Put the <PI:material>Seat post Lug (Product N...
```



```

</PI:step><PI:step xmlns:PI-"http://schemas.micro...
    Insert the <PI:material>Pinch Bolt (Product N...
</PI:step><PI:step xmlns:PI-"http://schemas.micro...
    Attach the <PI:material>LL Seat (Product Numb...
</PI:step><PI:step xmlns:PI-"http://schemas.micro...
    Inspect per specification <PI:specs>FI-620</P...
</PI:step>

```

(1 row(s) affected)

值得指出的是，所有的 XML 仍然是基于数据库中每个数据行的一行一列而生成的。

注意：

需要重复指出的是，.query 不能修改数据——它是只读操作。

注意，顺便提一句，这里需要声明名称空间。因为名称空间被声明为引用的模式集合的一部分，可以看到它扩展查询的方式以及破坏查询易读性的方式。通过使用声明 with XMLNAMESPACES () 可以修正这个问题：

```

WITH XMLNAMESPACES ('http://schemas.microsoft.com/sqlserver/2004/07/adventureworks/
ProductModelManuInstructions' AS PI)

SELECT ProductModelID, Instructions.query('/PI:root/PI:Location/PI:step') AS Steps

FROM Production.ProductModel
WHERE ProductModelID = 66;

```

提示：

名称空间声明的 URL 部分必须写在一行内。因为在印刷的时候，每行的字数受到了限制，所以本书中将它们显示为多行。请确保在你的代码中，将整个 URL 写在一行内。

这里的查询可读性稍强一些，但是生成的结果集是相同的。

2. .value

.value 方法用来查询离散数据。它使用 XPath 的语法定位指定的节点并提取标量值，其语法如下：

```
<instance of xml data type>.value (<XPath location>, <non-xml SQL Server Type>)
```

这里使用的技巧是为了确保指定的 XPath 能返回一个离散值。

例如，如果希望知道 ProductModelID 为 66 的第一个 Location 元素中的 LaborHours 属性值，那么可以进行如下操作：

```

WITH XMLNAMESPACES ('http://schemas.microsoft.com/sqlserver/2004/07/adventureworks/
ProductModelManuInstructions' AS PI)

SELECT ProductModelID,
    Instructions.value('/PI:root/PI:Location/@LaborHours[1]',
        'decimal (5,2)') AS Location

```

```
FROM Production.ProductModel
WHERE ProductModelID = 66
```

提示:

名称空间声明的 URL 部分必须写在一行内。因为在印刷的时候, 每行的字数受到了限制, 所以本书中将它们显示为多行。请确保在你的代码中, 将整个 URL 写在一行内。

检查结果:

```
ProductModelID Location
-----
```

```
66                1.50
```

```
(1 row(s) affected)
```

注意 SQL Server 提取了刚才指定的属性值(这种情况下, 是 Location 节点的 LaborHours 属性), 把它视为一个离散数据片段。返回值的数据类型必须可以转换为 SQL Server 中的非 XML 类型值, 并且必须返回标量值——即, 不能得到多个行。

3. .Modify

哈, 现在这里变得有些趣味了。

标准 W3C 格式的 XQuery 是只读的——即, 对于选择数据来说它非常好用, 但是对于 INSERT、UPDATE 或 DELETE 来说, 却没有提供任何等价的操作。微软显然也对此不满, 为了修改 XQuery 数据扩展了自己的 XQuery。这种对 XQuery 的扩展被称为 XML 数据操纵语言(XML Data Manipulation Language), 或 XML DML。XML DML 向 Xquery 添加了 3 条新命令:

- insert
- delete
- replace value of

注意:

像所有的 XML 关键字一样, 这些新命令是区分大小写的。

这些命令的名称暗示了它们各自的功能, 其中, replace value of 替代了 SQL 的 UPDATE 语句。

例如, 可以编写下面的代码增加.value 例子中原先为 1.5 小时的劳动时间:

```
WITH XMLNAMESPACES ('http://schemas.microsoft.com/sqlserver/2004/07/adventureworks/
ProductModelManuInstructions' AS PI)
```

```
UPDATE Production.ProductModel
SET Instructions.modify('replace value of
(/PI:root/PI:Location/@LaborHours)[1] with 1.75')
WHERE ProductModelID = 66;
```

提示:

名称空间声明的 URL 部分必须写在一行内。因为在印刷的时候, 每行的字数受到了限制,

所以本书中将它们显示为多行。请确保在你的代码中，将整个 URL 写在一行内。

现在如果重新运行.value 命令：

```
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/adventureworks/ProductModelManuInstruc
tions' AS PI)
```

```
SELECT ProductModelID, Instructions.value('/PI:root/PI:Location/@LaborHours')[1]',
'decimal (5,2)') AS Location
FROM Production.ProductModel
WHERE ProductModelID = 66
```

提示：

名称空间声明的 URL 部分必须写在一行内。因为在印刷的时候，每行的字数受到了限制，所以本书中将它们显示为多行。请确保在你的代码中，将整个 URL 写在一行内。

就可以得到新的值：

```
ProductModelID Location
```

```
-----
66                1.75
```

```
(1 row(s) affected)
```

提示：

这本质上是在 UPDATE 中进行 UPDATE 的一种方法。因为正在修改 SQL Server 的行，所以必须使一条 UPDATE 语句告诉 SQL Server 即将更新这行关系数据(这只出现在其中有 XML 的情况下)。这里还必须使用 replace value of 关键字指定要更新的 XML 部分。

4. .nodes

使用.nodes 可以获取 XML 块，并按照其在关系表中的存储方式将其拆分为多个数据行。这种获取一个 XML 文档并将其拆分成独立部分的方法，被称作拆分(shredding)文档。

利用.nodes 完成的工作，实质上是将 XML 数据实例拆分为它们自己的表(行数与符合 XQuery 条件的数据实例数相同)。可以预料，这意味着必须将.nodes 的结果看作一个表而不是一个列。.nodes 和典型表之间的主要差别是必须将.nodes 的结果交叉应用(cross apply)在指定的 XML 数据来源表中。因此，除了.nodes 外，.nodes 实际上还涉及到了更多的语法——可以把它看作与联接类似的东西，但使用特殊的 CROSS APPLY 关键字替代 JOIN，使用.node 替代 ON 子句。具体语法如下：

```
SELECT <column list>
FROM <source table>
CROSS APPLY <column name>.nodes(<XQuery>) AS <table alias for your .nodes results>
```

这样非常容易引起混淆。所以下面回顾一下先前的.value 例子。可以看到，寻找某个特定表的查询只返回一个结果：

```
WITH XMLNAMESPACES ('http://schemas.microsoft.com/sqlserver/2004/07/adventureworks/
ProductModelManuInstructions' AS PI)
SELECT ProductModelID,
       Instructions.value('(//PI:root/PI:Location/@LaborHours)[1]',
                          'decimal (5,2)') AS Location
FROM Production.ProductModel
WHERE ProductModelID = 66;
```

提示:

在名称空间声明的 URL 部分, 必须写在一行内。本书中将它们显示为多行, 是因为在印刷的时候, 每行的字数受到了限制。请确保在你的代码中, 将整个 URL 写在一行内。

.value 期望返回的结果是标量。因此需要确认对 XML 的每一行 XQuery 只返回单个值。.nodes 告诉 SQL Server 使用 XQuery 映射到指定的位置上, 并将在 XQuery 中找到的每项都作为单独的行来对待。

下面修改.value 的例子, 以返回所有 LocationID 值以及各自的劳动时间。这里希望能够将自己的 XML 数据看作关系数据进行查询, 因此必须将 LocationID 和 LaborHours 信息分成两列, 就好像它们在关系表中的排列方式一样。

```
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/adventureworks/ProductModelManuInstruc
tions' AS PI)

SELECT pm.ProductModelID,
       pmi.Location.value('..@LocationID', 'int') AS LocationID,
       pmi.Location.value('..@LaborHours', 'decimal(5,2)') AS LaborHours
FROM Production.ProductModel pm
CROSS APPLY pm.Instructions.nodes('/PI:root/PI:Location') AS pmi(Location);
```

提示:

名称空间声明的 URL 部分必须写在一行内。因为在印刷的时候, 每行的字数受到了限制, 所以本书中将它们显示为多行。请确保在你的代码中, 将整个 URL 写在一行内。

这里使用.nodes 方法基本上将一张表(ProductModel)分成了两张表(源表和 ProductModel 表中由 Instructions 列返回的.nodes 结果)。观察下面的结果:

ProductModelID	LocationID	LaborHours
7	10	2.50
7	20	1.75
7	30	1.00
7	45	0.50
7	50	3.00
7	60	4.00
10	10	2.00
10	20	1.50
10	30	1.00
10	4	1.50
10	50	3.00

10	60	4.00
43	50	3.00
44	50	3.00
47	10	1.00
47	20	1.00
47	50	3.50
48	10	1.00
48	20	1.00
48	50	3.50
53	50	0.50
66	50	1.75
67	50	1.00

(23 row(s) affected)

正如你所看到的，原先在 `ProductModel` 表中的每一行都返回了多行结果。例如，`ProductModelID7` 有 6 个不同的 `Location` 元素实例，因此返回的结果是 6 行数据，而不是 `ProductModel` 表现有的这一行。

也许，这在 XML 数据类型的各种方法中是最复杂的，它赋予了将 XML 数据转换为关系数据的无限可能性。

5. .exist

`.exist` 的工作有点像 SQL 中的 `EXISTS` 语句。它接受一个表达式(此处，是一个 XQuery 表达式，而不是 SQL 表达式)，并返回一个布尔值表示表达式为真或假(也可能输出 `NULL`)。

在 `.modify` 例子中，可以使用 `.exist` 查看那些包含 `step`(`step` 又包含 `specs` 元素)的行：

```
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/adventureworks/ProductModelManuInstructions' AS PI)
```

```
SELECT ProductModelID, Instructions
FROM Production.ProductModel
WHERE Instructions.exist('/PI:root/PI:Location/PI:step/PI:specs') = 1
```

提示：

要特别注意应用测试条件的场合。

例如，该代码会显示在至少有一个 `step` 包含 `specs` 元素的行——不需要显示每个拥有 `specs` 元素的 `step`。如果要测试每个元素，或许可以将这些元素分行输出(使用 `.nodes`)，或者在 XQuery 中设置测试条件。

注意，名称空间声明的 URL 部分必须写在一行内。因为在印刷的时候，每行的字数受到了限制，所以本书中将它们显示为多行。请确保在你的代码中，将整个 URL 写在一行内。

4.1.5 施加超出模式集合范围的约束

在没有阅读本书之前，你应该已经对关系数据库的约束有了一定的了解。如果需要约束关系数据库，它会遵循 XML 数据的方式执行。实际上，通过使用模式集合，已经实现了这一思想的

绝大部分。但如何施加超出模式集合范围的约束呢？

4.2 提取 XML 格式的关系数据

在发布 SQL Server 2005 之前，就已经有很多人指出了这一领域。我们有很多不同的选择，而且还可能有更多的选择——在相当一段时间内，这些选择是非常灵活的。下面看一看详细内容。

4.2.1 FOR XML 子句

该子句可以用在大多数不同的集成模型的根部。除了 XML 映射模式(非常高级，本章随后会简要介绍它)及使用 XPATH，FOR XML 可以告诉 SQL Server 希望返回的是 XML，而不是较典型的结果集。实质上，它只是可以被添加到现有 SQL Server 语句末尾一个选项。

下面看看 SELECT 语句的语法：

```
SELECT <column list>
[FROM <source table(s)>]
[WHERE <restrictive condition>]
[GROUP BY <column name or expression using a column in the SELECT list>]
[HAVING <restrictive condition based on the GROUP BY results>]
[ORDER BY <I>]
[FOR XML {RAW|AUTO|EXPLICIT|PATH}
    [, XMLDATA][, ELEMENTS][, BINARY base64]]
[OPTION (<query hint>, [, ...n])]
```

迄今为止，这看上去似乎并不难——毕竟这是高级版本——不过，这一次关注的是 FOR XML 行。

为了确定在结果中如何对 XML 进行格式化，FOR XML 提供了 4 个不同的初始化选项。

- **RAW**——以单个数据元素的形式返回结果中的每个数据行，使用“行”元素名，每个被列出的列都作为“行”元素的属性。即使联接了多个表，RAW 输出结果的元素数量也等于标准的 SQL 查询中行的数量。
- **AUTO**——该选项将使用数据源中的表名或表的别名来标识每个元素。如果查询会从一个或多个表中输出数据，那么来自每个表中的数据被拆分成分离的、嵌套的元素。如果使用 AUTO 选项，同时希望将列数据表现为元素而不是属性，那么也会支持另一个选项 **ELEMENTS**。
- **EXPLICIT**——该选项肯定是对查询进行格式化最复杂的选项，不过所得到的最终结果是一个看上去高度可控制的 XML。利用该选项可以为返回的数据定义某些层次结构，然后对查询进行格式化，以便于每个数据块按照需要归属于一个指定的层次级别(并分配相应的标签)。该选项基本被 PATH 选项所替代，这里保留它只是为了实现向后的兼容性。
- **PATH**——在 SQL Server 2005 中加入该选项，是为了以更可用的格式实现 EXPLICIT 选项级别的灵活性——需要对输出的结果的格式进行高度控制的时候应该会需要使用这个选项。

提示：

这些选项都没有提供所需要的根元素。如果希望 XML 文档被认为是“良好格式化”的，那

么必须为根元素设置合适的起始、结束标签对结果进行封装。虽然这样会带来一些麻烦，不过也会带来好处——它意味着可以将多个 XML 查询结合在一起，并将不同的结果封装到一个 XML 文件中，构建更复杂的 XML。

除了主要的格式化选项之外，还可以使用其他一些可选参数来进一步对 SQL Server 在 XML query 提供的输出结果进行修改：

- **XMLDATA**——它告诉 SQL Server 希望在结果前部应用 XML 模式。该模式会定义结构(包含数据类型)和 XML 数据规则。
- **ELEMENTS**——只在使用 AUTO 格式化选项的时候才能使用该选项。它告诉 SQL Server 以嵌套元素(而不是属性)的方式返回数据中的列。
- **BINARY BASE64**——它告诉 SQL Server 以 base64 格式对任何的二进制列(binary、varbinary 和图像)进行编码。如果也使用 AUTO 选项，则该选项也被隐式应用(使没有指定它，SQL Server 也会使用它)。对于 EXPLICIT 和 RAW 查询来说，没有隐式使用这个选项，但这却是当前唯一有效的选项——最终的目标是自动让两个选项向二进制数据提供 URL 链接(除非要使用 base64 编码)，但是现在还没有实现。
- **TYPE**——通知 SQL Server 返回报告 XML 数据类型的结果，而不是返回报告默认 Unicode 字符类型的结果。
- **ROOT**——如果没有添加根节点，该选项会令 SQL Server 添加根节点。可以为根节点取一个名称，或是使用默认的名称(根)

下面更深入地了解一下这些选项。

1. RAW

这是一种很清晰的选项。它的目的就是让事情完成(没有虚张声势，也根本没有特别的格式化)只是花费最少的气力将关系行转换成 XML 数据元素。元素被命名为“行”(真有创意，不是吗?)，在选择列表中的每个列都被作为属性而添加，属性的名称使用显示的列名，如果你运行过传统的 SELECT 语句，你就会明白它的名称由来。

提示：

这种方式不利的一面是在对属性的命名上，必须确保每个列都有名称。通常，如果完成了一个聚集操作或是计算列操作并且没有提供别名，那么 SQL Server 不会显示列标头——在进行 XML 查询的时候，所有的结果都必须有名称，所以不要忘记为计算列设置别名。

下面开始进行些较简单的操作。假设经理要求提供一个查询，列出一些客户的订单——假设 CustomerID 为 1 和 2。在读过本书的前 5 章之后，你可能会说“没问题!”，同时会给出类似下面的代码：

```
SELECT sc.CustomerID,
       pp.LastName,
       pp.FirstName,
       soh.SalesOrderID,
       soh.OrderDate
FROM Person.Person pp
JOIN Sales.Customer sc
```

```

ON pp.BusinessEntityID - sc.PersonID
JOIN Sales.SalesOrderHeader soh
ON sc.CustomerID - soh.CustomerID
WHERE sc.CustomerID - 29484 OR sc.CustomerID - 29485;

```

这样，就可以将结果交给老板了。

```

29484 Achong      Gustavo      44132      2001-09-01 00:00:00.000
29484 Achong      Gustavo      45579      2002-03-01 00:00:00.000
...
...
29485 Abel        Catherine    65157      2004-03-01 00:00:00.000
29485 Abel        Catherine    71782      2004-06-01 00:00:00.000

```

很简单，对吗？好，现在老板又回过头说，“很好——现在我要让 Billy Bob 写点什么把这个结果转换成 XML——真不幸，看上去可能要花一两天的时间”。这时你不失时机地说，“哦，为什么这样说呢？”然后简单地加上了 3 个关键字：

```

SELECT sc.CustomerID,
       pp.LastName,
       pp.FirstName,
       soh.SalesOrderID,
       soh.OrderDate
FROM Person.Person pp
JOIN Sales.Customer sc
ON pp.BusinessEntityID - sc.PersonID
JOIN Sales.SalesOrderHeader soh
ON sc.CustomerID - soh.CustomerID
WHERE sc.CustomerID - 29484 OR sc.CustomerID - 29485

```

FOR XML RAW;

你已经让老板非常高兴了。输出的结果是一对一匹配的(与运行一个标准的 SQL 查询中所能看到的结果集相对)。

```

<row CustomerID="1" LastName="Achong" FirstName="Gustavo" SalesOrderID="44132"
OrderDate="2001-09-01T00:00:00"/>
<row CustomerID="1" LastName="Achong" FirstName="Gustavo" SalesOrderID="45579"
OrderDate="2002-03-01T00:00:00"/>
<row CustomerID="1" LastName="Achong" FirstName="Gustavo" SalesOrderID="46389"
OrderDate="2002-06-01T00:00:00"/>
<row CustomerID="1" LastName="Achong" FirstName="Gustavo" SalesOrderID="47454"
OrderDate="2002-09-01T00:00:00"/>
<row CustomerID="1" LastName="Achong" FirstName="Gustavo" SalesOrderID="48395"
OrderDate="2002-12-01T00:00:00"/>
<row CustomerID="1" LastName="Achong" FirstName="Gustavo" SalesOrderID="49495"
OrderDate="2003-03-01T00:00:00"/>
<row CustomerID="1" LastName="Achong" FirstName="Gustavo" SalesOrderID="50756"
OrderDate="2003-06-01T00:00:00"/>
<row CustomerID="2" LastName="Abel" FirstName="Catherine" SalesOrderID="53459"
OrderDate="2003-09-01T00:00:00"/>

```



```
<row CustomerID-"2" LastName-"Abel" FirstName-"Catherine" SalesOrderID-"58907"
  OrderDate-"2003-12-01T00:00:00"/>
<row CustomerID-"2" LastName-"Abel" FirstName-"Catherine" SalesOrderID-"65157"
  OrderDate-"2004-03-01T00:00:00"/>
<row CustomerID-"2" LastName-"Abel" FirstName-"Catherine" SalesOrderID-"71782"
  OrderDate-"2004-06-01T00:00:00"/>
```

这里提醒一下,如果列的长度超出了通过打开“工具”|“选项”菜单,在“查询结果”的“以文本格式显示结果”节点中所设置的长度(最大是 8192), Management Studio 会自动将该列截断。如果直接输出到文件,那么结果窗口中(网格或文本,尽管如果数据是 XML,网格所允许的字符会更多)也存在这样的问题。这不是 SQL Server 自身的问题,而是工具本身的问题。使用其他的方法提取结果(例如 ADO.NET)就不会遇到这样的问题。

提示:

还有,请留意为了显示得更加清晰,我在上面的结果中加入了回车——SQL Server 只会将所有的这些元素一起返回,以使它们更加紧凑。

查询生成的每行数据都有对应的一个 XML 元素。不管数据源来自哪个表,所有的列信息都表现为“行”元素的属性。这样做的弊端是无法表示数据的真正层次——排列的顺序只由用户决定。不过,好处是 XML 文档对象模型(Document Object Model)(如果这是你正在使用的模型)会比较浅,因而占用的内存稍少些,并得到较好的性能。具体程度和你自己正在做的工作有关。

2. AUTO

AUTO 采用的数据处理方式与 RAW 稍有不同。它试图格式化得更好一些——根据表命名元素(如果使用了表的别名,还可以根据表的别名命名)。此外, AUTO 可以识别出数据是否可能有某些能够用 XML 来表示的下级层次。

下面回到上一节客户订单的例子中来。这次使用 AUTO 选项,所以可以看到输出结果与采用 RAW 模式输出的普通形式之间的不同。

```
SELECT sc.CustomerID,
  pp.LastName,
  pp.FirstName,
  soh.SalesOrderID,
  soh.OrderDate
FROM Person.Person pp
JOIN Sales.Customer sc
  ON pp.BusinessEntityID = sc.PersonID
JOIN Sales.SalesOrderHeader soh
  ON sc.CustomerID = soh.CustomerID
WHERE sc.CustomerID = 29484 OR sc.CustomerID = 29485
```

第一个明显的区别是元素的名称变为数据来源表的名称或表的别名——在 FOR XML AUTO 查询中选择表的别名时,需要考虑到这一点。也许更大的区别是可以更加全面地观察 XML。此处再次整理了输出结果,使其看起来更加清晰:

```
<sc CustomerID-"29484">
  <pp LastName-"Achong" FirstName-"Gustavo">
```

```

    <soh SalesOrderID-"44132" OrderDate-"2001-09-01T00:00:00"/>
    <soh SalesOrderID-"45579" OrderDate-"2002-03-01T00:00:00"/>
    <soh SalesOrderID-"46389" OrderDate-"2002-06-01T00:00:00"/>
    <soh SalesOrderID-"47454" OrderDate-"2002-09-01T00:00:00"/>
    <soh SalesOrderID-"48395" OrderDate-"2002-12-01T00:00:00"/>
    <soh SalesOrderID-"49495" OrderDate-"2003-03-01T00:00:00"/>
    <soh SalesOrderID-"50756" OrderDate-"2003-06-01T00:00:00"/>
  </pp>
</sc>
<sc CustomerID-"29485">
  <pp LastName-"Abel" FirstName-"Catherine">
    <soh SalesOrderID-"53459" OrderDate-"2003-09-01T00:00:00"/>
    <soh SalesOrderID-"58907" OrderDate-"2003-12-01T00:00:00"/>
    <soh SalesOrderID-"65157" OrderDate-"2004-03-01T00:00:00"/>
    <soh SalesOrderID-"71782" OrderDate-"2004-06-01T00:00:00"/>
  </pp>
</sc>

```

源自第二个表(由 SELECT 表确定)的数据被嵌套到在源自第一个表的数据中间。在这种情况下, soh 元素被嵌套在 pp 元素中, 而 pp 元素由嵌套在 c 元素内。如果选择列表中首先列出了 SalesOrderHeader 表中的列, 那么 Person 和 Customer 会被嵌套在 SalesOrderHeader 内。

提示:

注意 SELECT 列表中业务的顺序。要考虑的主要问题是你的 XML 查询必须返回结果。调整 SELECT 列表, 以便让它生成类型与你的 XML 目标相吻合。当然, 也可以自己设置它的风格使其呈现不同的形式——不过既然 SQL Server 已经生成了它, 你为什么不利用它呢?

使用 AUTO 的弊端是最终的 XML 数据模型结果稍微有点复杂。使用 AUTO 的优点是数据被明确地分解成层次模式。这样可以简化元素被作为更重要断点时的情况——比如使用双排序报表时(例如, 在 Contact 行中排序 SalesOrderHeader 行)。

3. EXPLICIT

对于该选项, 使用 EXPLICIT 将是一个很有趣的选择——它大致描述了在试图创建查询时所使用的语言类型。EXPLICIT 选项做了很多准备工作, 不过这种努力也带来了回报, 它可以对什么是元素、什么是属性, 以及什么元素被嵌套到其他元素中, 进行非常好的粒度控制。

提示:

目前, EXPLICIT 所能完成的大部分工作都可以用 PATH 替代。但是顾名思义, EXPLICIT 可以很好的控制输出。一般来说, 应该使用 PATH, 但当 PATH 不能满足需要的时候, 我建议你可以考虑一下 EXPLICIT。

EXPLICIT 允许对层次中的每个级别以及每个级别的外观进行定义。为了定义层次, 需要创建在内部被称作通用表的对象。在大多数情况下, 通用表类似于 SQL Server 中生成的其他结果集。它通常是使用 UNION 语句将数据片段一次结合到一个级别中生成的, 不过也可以在 UDF 中构建多个数据, 然后使用 SELECT 生成最终的 XML。通用表与传统结果集之间的不同在于: 必须在结果集中提供足够的元数据, 这样 SQL Server 可以根据你的需要, 将结果集转换成模式中的 XML

文档。

这里所说的“足够的元数据”是指什么呢？嗯，为了让你对它的复杂性有所认识，表 4-2 观察一个真正的通用表——在本节稍后的例子中将要分析的一段示例代码会用到它：

表 4-2

Tag	Parent	scl1! CustomerID	pp12! LastName	pp12! FirstName	soh13! SalesOrderID	soh13! OrderDate
1	NULL	29484	NULL	NULL	NULL	NULL
2	1	29484	Achong	Gustavo	NULL	NULL
3	2	29484	Achong	Gustavo	44132	9/1/2001
3	2	29484	Achong	Gustavo	45579	3/1/2002
3	2	29484	Achong	Gustavo	46389	6/1/2002
3	2	29484	Achong	Gustavo	47454	9/1/2002
3	2	29484	Achong	Gustavo	48395	12/1/2002
3	2	29484	Achong	Gustavo	49495	3/1/2003
3	2	29484	Achong	Gustavo	50756	6/1/2003
1	NULL	29485	NULL	NULL	NULL	NULL
2	1	29485	Abel	Catherine	NULL	NULL
3	2	29485	Abel	Catherine	53459	9/1/2003
3	2	29485	Abel	Catherine	58907	12/1/2003
3	2	29485	Abel	Catherine	65157	3/1/2004
3	2	29485	Abel	Catherine	71782	6/1/2004

这似乎就是我们要构建的通用表。构建它的目的是为了让我们 EXPLICIT 的返回结果与本例的 AUTO 查询提取的结果相同。

提示：

你可能会说，“如果它只是生成和 AUTO 一样的东西，那为什么要使用它？”好的，对于这个特定的例子，它碰巧也可以用 AUTO 来实现——这里使用它的目的是为了阐述与你之前所看到的事物进行比较所呈现的一些功能差异。不过，我们会在本节后面看到，EXPLICIT 允许我们进行“额外”的格式化，这是 AUTO 或 RAW 不能完成的(但是 PATH 可以完成)——所以请先容忍我在这里的介绍。

关于结果集，应该注意到如下几件事情：

- 其中添加了两个特殊的元数据列(Tag 和 Parent)，否则，不能使用数据(它们不是来自于表的列)；
- 真正的列名遵循特殊的格式(它提供了额外的元数据)；
- 数据已经基于层次而被排序。

这里的每个项目对最终结果来说都是至关重要的，因此，在开始进行复杂的例子之前，先看一下为了构建它需要知道那些知识。

Tag 和 Parent

XML 本质上是自然分层的(元素包含在其他元素内, 这实质上创建了一种父子关系)。Tag 和 Parent 列定义了每一行与元素层次之间的关系。每个行被指派到一个特定的标签级别上(随后会为其指派元素名)——不难预料, 该级别位于 Tag 列内。然后 Parent 提供引用信息, 指示体系结构中下一个最高的级别。通过执行这样的操作, SQL Server 知道了行作为属性需要被嵌套或指派到什么级别(将会基于列名指定它成为元素还是属性, 下一节中将会介绍这部分内容)。如果 Parent 为 NULL, 那么 SQL Server 就会知道该行应该是顶级元素还是该元素的属性。

因此, 如果得到与表 4-3 类似的数据:

表 4-3

Tag	Parent
1	NULL
2	1

那么第一行将被关联到顶级元素(外部元素的属性, 或者元素本身), 而且第二行将与顶级元素中嵌套的某个元素相关联(它的 Parent 值为 1, 同第一行的 Tag 值相匹配)。

列的命名

坦白地说, 第一次开始研究 EXPLICIT 时, 这是最让我困惑的部分。Tag 和 Parent 都有非常整洁的分界点(他们都分别有自己的列), 名称选取元数据中的几个片段, 并将它们组合为一个整体——标明起始位置的唯一方法是使用惊叹号(!)进行分隔。

命名的格式如下所示:

```
<element name>![<tag>![<attribute
name>]][!{element|hide|ID|IDREF|IDREFS|xml|xmltext|cdata}]
```

当然, 这里的元素名, 正是你希望指定给 XML 元素的名称。对于任何给定的标签级别, 一旦定义了一个列名, 拥有相同标签的所有其他列都必须拥有同前一个具有相同标签号的列同名。因此, 如果一个列已经被定义为[MyElement!2!MyCol], 那么其他的列可以被命名为[MyElement!2!MyOtherCol], 而不能被命名为[SomeOtherName!2!MyOtherCol]。

标签使用匹配的标签号将列同行关联起来。SQL Server 看到通用表的时候, 它读取该标签号, 并分析拥有相同标签号的列。这样, 当 SQL Server 看到表 4-4 所示的行:

表 4-4

Tag	Parent	c!1!	c!1!	c!1!	soh!2!	soh!2!OrderDate
		ContactID	LastName	FirstName	SalesOrderID	
1	NULL	1	Achong	Gustavo	NULL	NULL

它可以看到标签号为 1, 并知道它应该处理 sc!1!ContactID、c!1!LastName 和 c!1!FirstName, 不过它不用处理其他的名称, 如 pp!2!LastName 或者 soh!3!SalesOrderID。同样, 它查看下一行的标签号, 看到标签号为 2 就知道自己应该处理 sc!1!ContactID、pp!2!LastName 和 pp!2!FirstName, 不过它不用处理 soh!3!SalesOrderID。

从属性名开始,事情变得更加复杂(此后还有更多的任务)。如果没有指定指令(随后介绍)那么就必须提供属性,其名称就是该列提供的 XML 属性名。在 XML 中,属性将成为列名中指定的元素的一部分。

如果指定了指令,那么属性将分成 3 种不同的形式:

- 它是被禁止的——必须令属性保持空白(尽管还是可以使用一个惊叹号!来标记它的位置)。这种情况发生在使用 CDATA 指令的时候。
- 它是可选的——可以提供该属性,不过这并不是必需的。接下来出现的情况与所选择的指令有关。
- 它仍是必需的——这适用于 elements 和 xml 指令。此时,属性的名称会变成新元素的名称,该新元素是作为 element 或 xml 指令的结果创建的。

现在已经对命名有了足够的认识,可以满足查询的最小需求了。下面通过一个例子,了解一下什么类型的查询可以生成什么类型的结果。

首先从查询开始,它生成的基本数据与我们在 RAW 和 AUTO 例子中生成的基本数据相同。你可能会注意到 EXPLICIT 对代码的影响比 RAW 和 AUTO 大很多。在使用 RAW 和 AUTO 时,在绝大多数情况下都要在最后加入 FOR XML 子句。而使用 EXPLICIT 时,需要重新考虑查询的组合方法。

其语法如下所示:

```
USE AdventureWorks2008
```

```
SELECT 1
        NULL          as Tag,
        NULL          as Parent,
        sc.CustomerID as [sc!1!CustomerID],
        NULL          as [pp!2!LastName],
        NULL          as [pp!2!FirstName],
        NULL          as [soh!3!SalesOrderID],
        NULL          as [soh!3!OrderDate]
FROM Person.Person pp
JOIN Sales.Customer sc
  ON pp.BusinessEntityID = sc.PersonID
WHERE sc.CustomerID = 29484 OR sc.CustomerID = 29485
```

```
UNION
```

```
SELECT 2,
        1,
        sc.CustomerID as [sc!1!CustomerID],
        pp.LastName   as [pp!2!LastName],
        pp.FirstName  as [pp!2!FirstName],
        NULL          as [soh!3!SalesOrderID],
        NULL          as [soh!3!OrderDate]
FROM Person.Person pp
JOIN Sales.Customer sc
  ON pp.BusinessEntityID = sc.PersonID
JOIN Sales.SalesOrderHeader soh
  ON sc.CustomerID = soh.CustomerID
WHERE sc.CustomerID = 29484 OR sc.CustomerID = 29485
```

```
UNION ALL
```

```

SELECT 3,
        2,
        sc.CustomerID          as [sc!1!CustomerID],
        pp.LastName             as [pp!2!LastName],
        pp.FirstName            as [pp!2!FirstName],
        soh.SalesOrderID,
        soh.OrderDate
FROM Person.Person pp
JOIN Sales.Customer sc
  ON pp.BusinessEntityID = sc.PersonID
JOIN Sales.SalesOrderHeader soh
  ON sc.CustomerID = soh.CustomerID
WHERE sc.CustomerID = 29484 OR sc.CustomerID = 29485

ORDER BY [sc!1!CustomerID], [pp!2!LastName], [pp!2!FirstName], [soh!3!SalesOrderID]
FOR XML EXPLICIT

```

注意:

这里只用了一次 FOR XML——在 UNION 的第一次查询之后。

重申了一遍——真讨厌！不过，通过完成少量的修改，可以将 XML 改变成 AUTO 不能带来的形式。

为了做一个非常简单的说明，下面针对查询的需求做一点小小的修改。如果希望 LastName 信息成为 soh 的属性，而不是(或者，正如它发生的那样，除了)pp 元素的属性，应该怎么做呢？为此，在使用 AUTO 时需要一些小技巧(对于每一行，都需要使用相关子查询再次查看 Customer——AUTO 不允许在两个位置使用相同的值)。如果进行多个查询，代码会变得非常复杂——实际上，你可能根本无法得到需要的结果。使用 EXPLICIT，情况就会变得相对简单(至少，EXPLICIT 的定义相对简单)。

要使用 EXPLICIT 完成任务，只需要在 SELECT 列表中再次引用 LastName。不过需要将它的新实例与 soh 相关联而不是与 c 相关联：

```
USE AdventureWorks2008
```

```

SELECT 1          as Tag,
        NULL       as Parent,
        sc.CustomerID as [sc!1!CustomerID],
        NULL       as [pp!2!LastName],
        NULL       as [pp!2!FirstName],
        NULL       as [soh!3!SalesOrderID],

        NULL       as [soh!3!OrderDate],
        NULL       as [soh!3!LastName]

```

```

FROM Person.Person pp
JOIN Sales.Customer sc
  ON pp.BusinessEntityID = sc.PersonID
WHERE sc.CustomerID = 29484 OR sc.CustomerID = 29485

```

```
UNION
```



```

SELECT 2,
       1,
       sc.CustomerID      as [sc!1!CustomerID],
       pp.LastName        as [pp!2!LastName],
       pp.FirstName       as [pp!2!FirstName],
       NULL               as [soh!3!SalesOrderID],

```

```

       NULL               as [soh!3!OrderDate],
       NULL               as [soh!3!LastName]

```

```

FROM Person.Person pp
JOIN Sales.Customer sc
  ON pp.BusinessEntityID = sc.PersonID
JOIN Sales.SalesOrderHeader soh
  ON sc.CustomerID = soh.CustomerID
WHERE sc.CustomerID = 29484 OR sc.CustomerID = 29485

```

```

UNION ALL

```

```

SELECT 3,
       2,
       sc.CustomerID      as [sc!1!CustomerID],
       pp.LastName        as [pp!2!LastName],
       pp.FirstName       as [soh!2!FirstName],
       soh.SalesOrderID,

```

```

       soh.OrderDate,
       pp.LastName

```

```

FROM Person.Person pp
JOIN Sales.Customer sc
  ON pp.BusinessEntityID = sc.PersonID
JOIN Sales.SalesOrderHeader soh
  ON sc.CustomerID = soh.CustomerID
WHERE sc.CustomerID = 29484 OR sc.CustomerID = 29485

```

```

ORDER BY [sc!1!CustomerID], [pp!2!LastName], [pp!2!FirstName], [soh!3!SalesOrderID]
FOR XML EXPLICIT

```

执行它，可以得到同以前完全相同的结果，只不过这一次还获得了在 soh 元素中要查找的额外属性：

```

<sc CustomerID="29484">
  <pp LastName="Achong" FirstName="Gustavo">
    <soh SalesOrderID="44132" OrderDate="2001-09-01T00:00:00"
    LastName="Achong"/>
    <soh SalesOrderID="45579" OrderDate="2002-03-01T00:00:00"
    LastName="Achong"/>
    <soh SalesOrderID="46389" OrderDate="2002-06-01T00:00:00"
    LastName="Achong"/>
    <soh SalesOrderID="47454" OrderDate="2002-09-01T00:00:00"
    LastName="Achong"/>
    <soh SalesOrderID="48395" OrderDate="2002-12-01T00:00:00"
    LastName="Achong"/>
  </pp>
</sc>

```

```

    <soh SalesOrderID="49495" OrderDate="2003-03-01T00:00:00"
    LastName="Achong"/>
    <soh SalesOrderID="50756" OrderDate="2003-06-01T00:00:00"
    LastName="Achong"/>
  </pp>
</sc>
<sc CustomerID="29485">
  <pp LastName="Abel" FirstName="Catherine">
    <soh SalesOrderID="53459" OrderDate="2003-09-01T00:00:00"
    LastName="Abel"/>
    <soh SalesOrderID="58907" OrderDate="2003-12-01T00:00:00"
    LastName="Abel"/>
    <soh SalesOrderID="65157" OrderDate="2004-03-01T00:00:00"
    LastName="Abel"/>
    <soh SalesOrderID="71782" OrderDate="2004-06-01T00:00:00"
    LastName="Abel"/>
  </pp>
</sc>

```

这个例子是为入门者准备的。你可以使用指令来获得更大的灵活性——设计并控制数据和模式的输出(如果使用 XMLDATA 选项)。

指令很难理解。不过一旦你理解了,它们就不难使用了,尽管偶尔会感到困惑(有些指令的工作方式并不直观,而且在不同的情形下行为也不同)。我个人认为(而且我所认识的那个团队的成员会因为我下面将要说的话而要揍我)微软公司人经历了倒霉的一天后,决定做些事,也让其他人来体会同样的痛苦。这样做的确很酷,因为人们没有其他选择,只能使用它。

总共可以使用的指令有 8 条。有些可以在同一层次级别中使用——而另一些则在给定的层次级别中相互排斥。

指令的目标是允许对结果进行调整。没有指令的话,EXPLICIT 选项就几乎或完全丧失价值了(如果不使用指令,AUTO 可以完成 EXPLICIT 指令的大部分功能,尽管有时候必须使用一些技巧)。因此,将这个牢记在心,下面看一看可以使用哪些指令。

element

这可能是所有指令中最容易理解的指令。它要做的就是标明希望将相关列作为元素添加而不是作为属性添加。元素将作为当前标签的孩子添加。例如,假设前面例子中提到的经理指示自己需要将 OrderDate 表示为自己的元素。为此,只需在 OrderDate 字段末添加 element 指令。

```

SELECT 1          as Tag,
        NULL      as Parent,
        sc.CustomerID as [sc!1!CustomerID],
        NULL      as [pp!2!LastName],
        NULL      as [pp!2!FirstName],
        NULL      as [soh!3!SalesOrderID],
        NULL      as [soh!3!OrderDate!element]

FROM Person.Person pp
JOIN Sales.Customer sc
ON pp.BusinessEntityID = sc.PersonID

```



```

WHERE sc.CustomerID = 29484 OR sc.CustomerID = 29485

UNION

SELECT 2,
       1,
       sc.CustomerID      as [sc!1!CustomerID],
       pp.LastName        as [pp!2!LastName],
       pp.FirstName       as [pp!2!FirstName],
       NULL,
       NULL
FROM Person.Person pp
JOIN Sales.Customer sc
  ON pp.BusinessEntityID = sc.PersonID
JOIN Sales.SalesOrderHeader soh
  ON sc.CustomerID = soh.CustomerID
WHERE sc.CustomerID = 29484 OR sc.CustomerID = 29485

UNION ALL

SELECT 3,
       2,
       sc.CustomerID      as [sc!1!CustomerID],
       pp.LastName        as [pp!2!LastName],
       pp.FirstName       as [pp!2!FirstName],
       soh.SalesOrderID,
       soh.OrderDate
FROM Person.Person pp
JOIN Sales.Customer sc
  ON pp.BusinessEntityID = sc.PersonID
JOIN Sales.SalesOrderHeader soh
  ON sc.CustomerID = soh.CustomerID
WHERE sc.CustomerID = 29484 OR sc.CustomerID = 29485

ORDER BY [sc!1!CustomerID], [pp!2!LastName], [pp!2!FirstName], [soh!3!SalesOrderID]
FOR XML EXPLICIT

```

突然间，一个额外的元素替代了属性：

```

<sc CustomerID="29484">
  <pp LastName="Achong" FirstName="Gustavo">
    <soh SalesOrderID="44132">
      <OrderDate>2001-09-01T00:00:00</OrderDate>
    </soh>
    <soh SalesOrderID="45579">
      <OrderDate>2002-03-01T00:00:00</OrderDate>
    </soh>
    <soh SalesOrderID="46389">
      <OrderDate>2002-06-01T00:00:00</OrderDate>
    </soh>
  </pp>
</sc>

```

```

<soh SalesOrderID="47454">
  <OrderDate>2002-09-01T00:00:00</OrderDate>
</soh>
<soh SalesOrderID="48395">
  <OrderDate>2002-12-01T00:00:00</OrderDate>
</soh>
<soh SalesOrderID="49495">
  <OrderDate>2003-03-01T00:00:00</OrderDate>
</soh>
<soh SalesOrderID="50756">
  <OrderDate>2003-06-01T00:00:00</OrderDate>
</soh>
</pp>
</sc>
<sc CustomerID="29485">
  <pp LastName="Abel" FirstName="Catherine">
    <soh SalesOrderID="53459">
      <OrderDate>2003-09-01T00:00:00</OrderDate>
    </soh>
    <soh SalesOrderID="58907">
      <OrderDate>2003-12-01T00:00:00</OrderDate>
    </soh>
    <soh SalesOrderID="65157">
      <OrderDate>2004-03-01T00:00:00</OrderDate>
    </soh>
    <soh SalesOrderID="71782">
      <OrderDate>2004-06-01T00:00:00</OrderDate>
    </soh>
  </pp>
</sc>

```

xml

该指令在本质上与 `element` 指令相同。它将要生成的相关列表现为元素而不是属性。只有在需要对一些特殊字符进行编码的时候，才可以看到 `xml` 和 `element` 指令之间的差别——例如，等号是 XML 中的保留字，如果需要在 XML 中表示等号，就需要对它进行“编码”（“=”应该被编码为“&eq”）。使用 `element` 指令可以自动对元素的内容进行编码。使用 `xml` 指令，内容被直接发送到结果 XML 中，而不进行编码。如果使用 `xml` 指令，在该级别的其他项目(号码)中，除了 `hide` 之外，不能再有其他的指令。

hide

`hide` 是另一条简单指令，顾名思义它的功能就是——隐藏指定列的结果。

为什么会希望做这类事情呢？有时候，出于某种原因，会希望包含某些列，但是不希望输出它。例如，在常规查询中，可以对 `SELECT` 列表中没有出现的列执行 `ORDER BY` 操作。但是，对于 `UNION` 查询，就不能这样做——必须在 `SELECT` 列表中指定该列，因为只有它可以合并 `UNION` 中执行的所有查询。

下面用一个小例子来追踪某些产品的销售额。假设希望得到所有产品的清单，以及已出货订单的 SalesOrderID 和它们的出货时间。这里只需要 ProductID，但是希望按照相似产品在一起的顺序对 ProductID 排序——这意味着需要基于 ProductSubcategoryID 排序，但是又不希望最终结果包含 ProductSubcategoryID。

可以首先构建一个没有指令的查询——这样可以看到排序的工作过程：

```
SELECT 1
        NULL
        p.ProductID
        p.ProductSubcategoryID
        NULL
        NULL
        as Tag,
        as Parent,
        as [Product!1!ProductID],
        as [Product!1!ProductSubcategoryID],
        as [Order!2!OrderID],
        as [Order!2!OrderDate]
FROM Production.Product p
JOIN Sales.SalesOrderDetail AS sod
    ON p.ProductID = sod.ProductID
JOIN Sales.SalesOrderHeader AS soh
    ON sod.SalesOrderID = soh.SalesOrderID
WHERE soh.OrderDate BETWEEN '2003-03-27' AND '2003-03-27'

UNION ALL

SELECT 2,
        1,
        p.ProductID,
        p.ProductSubcategoryID,
        soh.SalesOrderID,
        soh.OrderDate
FROM Production.Product AS p
JOIN Sales.SalesOrderDetail AS sod
    ON p.ProductID = sod.ProductID
JOIN Sales.SalesOrderHeader AS soh
    ON sod.SalesOrderID = soh.SalesOrderID
WHERE soh.OrderDate BETWEEN '2003-03-27' AND '2003-03-27'
ORDER BY [Product!1!ProductSubcategoryID], [Product!1!ProductID],
        [Order!2!OrderID]
FOR XML EXPLICIT
```

提示：

请注意这里处理 OrderDate 的方式。尽管需要从 SalesOrderHeader 表中取回该信息，但是将该信息与 SalesOrderDetail 表中的 SalesOrderID 信息合并却是非常简单的(因为使用了 EXPLICIT)。当它发生时，也可以只从 SalesOrderHeader 表中取出 SalesOrderID 信息，不过有时候需要将多个表的数据混合到一个元素里，此查询是对我们可以执行操作的另一种演示。

从结果中可以看到，它的确按照我们的需要进行排序。

```
<Product ProductID="779" ProductSubcategoryID="1">
    <Order OrderID="49775" OrderDate="2003-03-27T00:00:00"/>
</Product>
<Product ProductID="782" ProductSubcategoryID="1">
```

```

    <Order OrderID="49774" OrderDate="2003-03-27T00:00:00"/>
</Product>
<Product ProductID="764" ProductSubcategoryID="2">
    <Order OrderID="49776" OrderDate="2003-03-27T00:00:00"/>
</Product><
Product ProductID="766" ProductSubcategoryID="2">
    <Order OrderID="49777" OrderDate="2003-03-27T00:00:00"/>
</Product>

```

现在添加 **hide** 指令以便隐藏分类信息:

```

SELECT 1                                as Tag,
      NULL                                as Parent,
      p.ProductID                        as [Product!1!ProductID],

      p.ProductSubcategoryID as [Product!1!ProductSubcategoryID!hide],

      NULL                                as [Order!2!OrderID],
      NULL                                as [Order!2!OrderDate]
FROM Production.Product p
JOIN Sales.SalesOrderDetail AS sod
  ON p.ProductID = sod.ProductID
JOIN Sales.SalesOrderHeader AS soh
  ON sod.SalesOrderID = soh.SalesOrderID
WHERE soh.OrderDate BETWEEN '2003-03-27' AND '2003-03-27'

UNION ALL

SELECT 2,
      1,
      p.ProductID,
      p.ProductSubcategoryID,
      soh.SalesOrderID,
      soh.OrderDate
FROM Production.Product AS p
JOIN Sales.SalesOrderDetail AS sod
  ON p.ProductID = sod.ProductID
JOIN Sales.SalesOrderHeader AS soh
  ON sod.SalesOrderID = soh.SalesOrderID
WHERE soh.OrderDate BETWEEN '2003-03-27' AND '2003-03-27'

ORDER BY [Product!1!ProductSubcategoryID!hide], [Product!1!ProductID],
         [Order!2!OrderID]

FOR XML EXPLICIT

```

这里得到了相同的结果,但只有在这一次,Category 信息确实被隐藏了。

```

<Product ProductID="779">
    <Order OrderID="49775" OrderDate="2003-03-27T00:00:00"/>
</Product>

```



```

<Product ProductID="782">
    <Order OrderID="49774" OrderDate="2003-03-27T00:00:00"/>
</Product>
<Product ProductID="764">
    <Order OrderID="49776" OrderDate="2003-03-27T00:00:00"/>
</Product>
<Product ProductID="766">
    <Order OrderID="49777" OrderDate="2003-03-27T00:00:00"/>
</Product>

```

id、idref 和 idrefs

除非还使用 XMLDATA 选项(位于 FOR 子句的 EXPLICIT 后)或针对其他一些具有适当声明的模式进行验证, 否则这三个指令不会产生任何影响。在考虑它们所做的事情时, 你会觉得这样做是合理的——它们向模式添加命令以强制执行某些操作, 不过, 如果没有模式, 那应该修改什么呢?

XML 中包含 id 的概念。id 在 XML 中的作用类似于关系数据中的主键——它为 XML 文档中的元素名称指派一个唯一的标识。对于任意元素名称, 在 id 中只能指定一个属性。XML 的 id 属性是在 XML 的模式中定义的。一旦元素中的 id 属性给定了值, 那么同名的其他元素就不再允许拥有该属性。

注意:

不同于 SQL 中的主键, XML 中的 id 不能由多个属性组成(此处没有复合键的概念)。

因为 XML 拥有的概念同主键类似, 所以 XML 拥有类似外键的概念也就顺理成章了——即 idref 和 idrefs。可以使用它们创建从一个元素中的一个属性到另一个元素中的 id 属性之间的引用。

这对我们有什么好处呢? 如果没有它们, 在两个元素之间建立关系的方法就只有一种——嵌套它们。通过为元素指定一个 id, 然后在声明为 idref 或 idrefs 属性中引用它, 我们就可以无视两个元素在文档中的位置把它们连接起来。

这会带来一个问题, “那为什么有两个这样的指令呢?” 可以在它们的名称上找到答案: idref 提供的单个值与现有元素的 id 值相匹配。Idrefs 则提供多值的、通过空白分隔的列表——同样, 这些值都必须与现有元素的 id 值相匹配。结果是, 如果要建立一对多的关系(每个 id 值只有一个, 但是在 idref 属性中拥有该值的元素可能有多个)则使用 idref。可以使用 idrefs 来尝试创建多对多的关系(每个带有 idrefs 的元素都可以引用多个 id, 这些值可以被多个 id 引用)。

为了说明这个问题, 需要对前面的查询做一点小小的修改。这里从 idref 指令开始:

```

SELECT 1                                as Tag,
      NULL                               as Parent,

      p.ProductID                        as [Product!1!ProductID!ID],

      p.ProductSubcategoryID as [Product!1!ProductSubCategoryID!hide],
      NULL                               as [Order!2!OrderID],

      NULL                               as [Order!2!ProductID!idref],

```

```

        NULL                                as [Order!2!OrderDate]
FROM Production.Product AS p
JOIN Sales.SalesOrderDetail AS sod
    ON p.ProductID = sod.ProductID
JOIN Sales.SalesOrderHeader AS soh
    ON sod.SalesOrderID = soh.SalesOrderID
WHERE soh.OrderDate BETWEEN '2003-03-27' AND '2003-03-27'

UNION ALL

SELECT 2,
    1,
    p.ProductID,
    p.ProductSubcategoryID,
    sod.SalesOrderID,
    sod.ProductID,
    soh.OrderDate
FROM Production.Product AS p
JOIN Sales.SalesOrderDetail AS sod
    ON p.ProductID = sod.ProductID
JOIN Sales.SalesOrderHeader AS soh
    ON sod.SalesOrderID = soh.SalesOrderID
WHERE soh.OrderDate BETWEEN '2003-03-27' AND '2003-03-27'

ORDER BY [Product!1!ProductSubCategoryID!hide], [Product!1!ProductID!ID],
    [Order!2!OrderID]
FOR XML EXPLICIT, XMLDATA

```

在结果中可以看到，这里实际上只有两块内容是我们感兴趣的——模式和产品元素：

```

<Schema name="Schema1" xmlns="urn:schemas-microsoft-com:xml-data"
    xmlns:dt="urn:schemas-microsoft-com:datatypes">
  <ElementType name="Product" content="mixed" model="open">
    <AttributeType name="ProductID" dt:type="id"/>
    <attribute type="ProductID"/>
  </ElementType>
  <ElementType name="Order" content="mixed" model="open">
    <AttributeType name="OrderID" dt:type="i4"/>
    <AttributeType name="ProductID" dt:type="idref"/>
    <AttributeType name="OrderDate" dt:type="dateTime"/>
    <attribute type="OrderID"/>
    <attribute type="ProductID"/>
    <attribute type="OrderDate"/>
  </ElementType>
</Schema>

```

在模式中，可以看到一些非常具体的类型信息。Product 被声明为元素类型，还可以看到 ProductID 声明为该元素类型的 id。类似地，Order 元素的 ProductID 被声明为 idref。

下面一段是我们在 Product 元素中所感兴趣的内容：


```
<Product xmlns="x-schema:#Schema1" ProductID="779">
  <Order OrderID="49775" ProductID="779" OrderDate="2003-03-27T00:00:00"/>
</Product>
```

这种情况下, 注意 SQL Server 已经在 Product 元素中引用了内联模式。这声明了 Product 元素和其中的所有元素都必须遵守该模式——因此确保实施 id 和 idrefs。

使用 idrefs 指令时必须使用一点小技巧。SQL Server 要求构建 idrefs 列表的查询, 必须同构建带有 ids 元素的查询分隔开。这意味着必须在 UNION 中添加另一个查询, 以提供 idrefs(在构建 idrefs 列表之前, 必须知道可能的 ids 列表——但是实际的 ids 出现在 id 列表之后)。必须正好在执行生成 ids 的查询之前, 执行生成 idrefs 的查询。这样做令查询看起来非常复杂:

```
SELECT 1
        NULL
        p.ProductID
        NULL
        NULL
        NULL
        as Tag,
        as Parent,
        as [Product!1!ProductID],
        as [Product!1!OrderList!idrefs],
        as [Order!2!OrderID!id],
        as [Order!2!OrderDate]
FROM Production.Product p
JOIN Sales.SalesOrderDetail AS sod
    ON p.ProductID = sod.ProductID
JOIN Sales.SalesOrderHeader AS soh
    ON sod.SalesOrderID = soh.SalesOrderID
WHERE soh.OrderDate BETWEEN '2003-03-27' AND '2003-03-31'

UNION ALL

SELECT 1,
        NULL,
        p.ProductID,
        soh.SalesOrderID,
        NULL,
        NULL
FROM Production.Product AS p
JOIN Sales.SalesOrderDetail AS sod
    ON p.ProductID = sod.ProductID
JOIN Sales.SalesOrderHeader AS soh
    ON sod.SalesOrderID = soh.SalesOrderID
WHERE soh.OrderDate BETWEEN '2003-03-27' AND '2003-03-31'

UNION ALL

SELECT 2,
        1,
        p.ProductID,
        soh.SalesOrderID,
        soh.SalesOrderID,
        soh.OrderDate
FROM Production.Product AS p
JOIN Sales.SalesOrderDetail AS sod
```

```

    ON p.ProductID = sod.ProductID
JOIN Sales.SalesOrderHeader AS soh
    ON sod.SalesOrderID = soh.SalesOrderID
WHERE soh.OrderDate BETWEEN '2003-03-27' AND '2003-03-31'
ORDER BY [Product!1!ProductID], [Order!2!OrderID!id],
    [Product!1!OrderList!idrefs]
FOR XML EXPLICIT, XMLDATA

```

提示:

这里将日期范围扩展了一点,以确保对于给定的范围内存在多个产品 ID,这样你才能看到正确的多对多的关系。

模式看上去像我们在 idref 上所得到的—样糟糕:

```

<Schema name="Schema4" xmlns="urn:schemas-microsoft-com:xml-data"
    xmlns:dt="urn:schemas-microsoft-com:datatypes">
  <ElementType name="Product" content="mixed" model="open">
    <AttributeType name="ProductID" dt:type="i4"/>
    <AttributeType name="OrderList" dt:type="idrefs"/>
    <attribute type="ProductID"/>
    <attribute type="OrderList"/>
  </ElementType>
  <ElementType name="Order" content="mixed" model="open">
    <AttributeType name="OrderID" dt:type="id"/>
    <AttributeType name="OrderDate" dt:type="dateTime"/>
    <attribute type="OrderID"/>
    <attribute type="OrderDate"/>
  </ElementType>
</Schema>

```

但是元素没有太多的不同:

```

<Product xmlns="x-schema:#Schema4" ProductID="763" OrderList="49790 49797">
  <Order OrderID="49790" OrderDate="2003-03-28T00:00:00"/>
  <Order OrderID="49797" OrderDate="2003-03-29T00:00:00"/>
</Product>

```

使用 id、idref 和 idrefs 是非常复杂的。不过它们允许你创建强类型的输出。在绝大多数情况下,不需要这种控制级别以及随之而来的麻烦,不过,如果确实必要,有三个指令可能会成为我们的救兵。

xmltext

xmltext 期待列的内容是 XML,并试图将它作为所创建的 XML 文档的一部分而插入。

表面上,这似乎听起来非常简单(它们将一些文本插入到表中,行了!)。但是在哪里插入,什么时候插入,以及如何插入数据的规则,却有一些奇特之处:

- 只要插入的 XML 格式正确,根元素将被剥离——不过会遵循下面的几条规则保留并应用该元素的属性。

- 如果在使用 `xmltext` 指令时, 没有指定属性名称, 那么从被剥离元素中保留下来的属性将被添加到包含 `xmltext` 指令的元素中。在组合的元素中会使用保留属性的名称。如果任何被保留属性数据中的属性名称同组合元素中的其他属性信息相冲突, 那么会从被保留数据中剔除冲突的属性。
- 任何嵌套在被剥离元素中的元素, 将成为组合元素的嵌套元素。
- 如果提供了一个带有 `xmltext` 指令的属性名称, 则被保留的数据将被放入命名该名称的某个元素中。新的元素成为创建该指令的元素的子元素。
- 如果生成的 XML 格式不正确, 则没有确定的行为。基本上, 行为会依赖于最终结果, 不过我认为这样会报错(我还没有看到过使用格式不对的数据, 而不报错的例子)。

cdata

`cdata` 是 DTD 和 SGML 的继任者(SGML 是旧的标记语言, 这种用于图形业的语言是 HTML 和 XML 的前身。DTD 是类型定义文档, 它为 SGML [以及随后的 HTML 和 XML] 文档定义结构规则)。基本上, `cdata` 本身代表的是字符数据。XML 认为 `cdata` 区域是无法触及的——它完全忽视标记为 `cdata` 区域内包含的所有内容。因为不会验证 `cdata` 区域中的数据, 所以也不需要对其进行编码。如果保持某些数据完全不被修改(不能让编码修改该数据), 或者如果要移动数据, 但是又不知道数据是什么(因此你不知道它是否会引发问题), 可以使用 `cdata`。

为此, 下面用一个简单的例子说明一下——AdventureWorks2008 Production.Document 表。该表包含一个 `nvarchar(max)` 数据类型的字段。该字段的内容基本上是未知的。下面给出了一个将雇员的附注生成到 XML 中的查询:

```
SELECT 1                                as Tag,
       NULL                               as Parent,
       DocumentNode                      as [Document!1!DocumentNode],
       DocumentSummary                   as [Document!1!!cdata]
FROM Production.Document Document
WHERE DocumentSummary IS NOT NULL
ORDER BY [Document!1!DocumentNode]
FOR XML EXPLICIT
```

输出结果是非常直观的:

```
<Document DocumentNode="/1/2/">
  <![CDATA[It is important that you maintain your bicycle and keep it in good
repair. Detailed repair and service guidelines are provided along with
instructions for adjusting the tightness of the suspension fork.

]]>
</Document>
<Document DocumentNode="/2/2/">
  <![CDATA[Guidelines and recommendations for lubricating the required
components of your Adventure Works Cycles bicycle. Component lubrication is
vital to ensuring a smooth and safe ride and should be part of your standard
maintenance routine. Details instructions are provided for each bicycle
component requiring regular lubrication including the frequency at which oil
or grease should be applied.
```

```
]]>
</Document>
<Document DocumentNode="/3/2/">
  <![CDATA[Reflectors are vital safety components of your bicycle. Always
ensure your front and back reflectors are clean and in good repair. Detailed
instructions and illustrations are included should you need to replace the
front reflector or front reflector bracket of your Adventure Works Cycles
bicycle.

]]>
</Document>
<Document DocumentNode="/3/3/">
  <![CDATA[Detailed instructions for replacing pedals with Adventure Works
Cycles replacement pedals. Instructions are applicable to all Adventure Works
Cycles bicycle models and replacement pedals. Use only Adventure Works Cycles
parts when replacing worn or broken components.

]]>
</Document>
<Document DocumentNode="/3/4/">
  <![CDATA[Worn or damaged seats can be easily replaced following these simple
instructions. Instructions are applicable to these Adventure Works Cycles
models: Mountain 100 through Mountain 500. Use only Adventure Works Cycles
parts when replacing worn or broken components.

]]>
</Document>
```

基本上，这是很容易使用的一个指令。

4. PATH

下面稍微调整一下方向，使用“真正”的 XML 方法来获得数据。

虽然目前还没有废弃 EXPLICIT，不过别搞错了——PATH 是真正能够完成那些最初只有 EXPLICIT 才能完成的操作的最好方法。PATH 以多种方法实现不同的目标，这就是我建议绝大多数情况下输出复杂 XML 的原因。

提示：

这种推荐远比它看上去的要复杂得多。微软声称 PATH 更易用。不错，PATH 在很多方面的确是更加简单，不过，正如我们将要看到的，它也有自己的“例外”，这会让你对它到底能做什么感觉困惑。简言之，在某些情况下，如果你不知道 XPath，那么 EXPLICIT 确实是比较简单的。如果要处理 XML，那么 XPath 应该是最先学习的东西，如果你要了解它，就应该知道基于 XPath 的方法更有用。

不过要注意，如果要将兼容性退回到 SQL Server 2000，那么就必须坚持使用 EXPLICIT。

不难看出，PATH 选项一点都不差。因此，下面从 PATH 的基本使用方法开始介绍。为了展示 PATH 到底能提供些什么，接着会进行更加复杂的操作。

PATH 简介

使用 PATH, 可以创建一个模型塑造一个现有的标准来获取数据——Xpath。Xpath 有一个得到认可的标准, 并提供一种在 XML 模式中指向特定点的方法。对于 PATH, 为了说明如何以本地 XML 的方式处理数据, 这里使用了许多相同的规则和想法。

PATH 如何处理所引用的数据, 这要依赖于不同的规则, 包括列是否已命名(像 EXPLICIT 一样, 如果使用别名, 该别名就作为列的名称)。如果列确实有名称, 那么在需要的时候还可以使用其他许多规则。

下面看一些可能性。

未命名的列

未命名列中的数据会被视为行元素中的原始文本。为了说明这种情况, 这里修改了在 XML RAW 一节中使用过的例子。这里要做的是列出感兴趣的两个客户, 以及他们所下的多个订单:

```
SELECT sc.CustomerID,  
       COUNT(soh.SalesOrderID)  
FROM Person.Person pp  
JOIN Sales.Customer sc  
  ON pp.BusinessEntityID = sc.PersonID  
JOIN Sales.SalesOrderHeader soh  
  ON sc.CustomerID = soh.CustomerID  
WHERE sc.CustomerID = 29484 OR sc.CustomerID = 29485  
GROUP BY sc.CustomerID  
FOR XML PATH;
```

其输出为:

```
<row><CustomerID>29484</CustomerID>7</row>  
<row><CustomerID>29485</CustomerID>4</row>
```

针对查询中的每一行都创建了一个行元素(这很像在使用 RAW 时得到的结果)。但是要注意它在处理列数据上的区别。

既然 CustomerID 列已经被命名, 它被放置到自己的元素内(下一节将对此展开介绍)——但是, 要注意结果中的数字 7。它只是内嵌在行元素中的自由文本——它甚至根本没有同 CustomerID 直接关联, 因为它在 CustomerID 元素的外部。

提示:

注意在你的系统中准确的数量(在这里是 7)可能会发生变化, 这与你数据进行了多少次操作有关。重要的是要看到这个数量没有同 CustomerID 关联, 只是与行相关联的原始文本。

对此我个人认为, 在顶级元素上使用自由文本的场合是非常有限的。规则的确允许你这么做, 但是我相信这适用于不太清楚的数据。而且, 这就是它的工作方式——只在需要满足特定系统的要求时使用。

命名列

命名列会让事情变得相当复杂而不是变得更快捷。在最简单的形式里, 命名列就像未命名列一样简单——实际上, 在以前的例子中已经看到过一个这样的实例了。如果使用 PATH 对列进行

简单命名, 那么它只是作为一个额外的元素被添加入行内。

```
<row><CustomerID>29484</CustomerID>7</row>
```

其中 **CustomerID** 列是一个简单命名的列。

不过可以在列名称中使用特殊的字符, 表示要对该列进行怎样的特殊操作。下面看一看其中最重要的一些例子。

@

不, 这不是拼写错误——@符号就是这一节的标题。如果在列名中加入了一个@符号, 那么 SQL Server 会把该列看作前一列的属性。下面将 **CustomerID** 设为该行顶级元素的一个属性:

```
SELECT sc.CustomerID AS '@CustomerID',

COUNT(soh.SalesOrderID)
FROM Person.Person pp
JOIN Sales.Customer sc
  ON pp.BusinessEntityID = sc.PersonID
JOIN Sales.SalesOrderHeader soh
  ON sc.CustomerID = soh.CustomerID
WHERE sc.CustomerID = 29484 OR sc.CustomerID = 29485
GROUP BY sc.CustomerID
FOR XML PATH;
```

会生成:

```
<row CustomerID="29484">7</row>
<row CustomerID="29485">4</row>
```

注意, 订单数量仍然是行的文本元素——只有标识为属性的列移进去了。下一步, 我们可以对数量命名, 为之设置前缀, 也令它成为属性:

```
SELECT sc.CustomerID AS '@CustomerID',

COUNT(soh.SalesOrderID) AS '@OrderCount'

FROM Person.Person pp
JOIN Sales.Customer sc
  ON pp.BusinessEntityID = sc.PersonID
JOIN Sales.SalesOrderHeader soh
  ON sc.CustomerID = soh.CustomerID
WHERE sc.CustomerID = 29484 OR sc.CustomerID = 29485
GROUP BY sc.CustomerID
FOR XML PATH;
```

采用这样的方法, 就不再有元素的自由文本了:

```
<row CustomerID="29484" OrderCount="7"/>
<row CustomerID="29485" OrderCount="4"/>
```

还要注意, SQL Server 的智能程度较高, 它能够意识到所有的内容都包含在属性内——没有更低级别的元素或简单文本时, 它会选择生成自闭合标记(在元素的末尾可以看到“/”符号)。

既然如此, 为什么还说这里需要技巧呢? 嗯, 这里有很多不同的“只有当……时才能工作”这样类型的规则。要说明这些规则, 要先简单修改一下原先的查询。这次看上去它是能够工作的,

不过如果试图运行它，SQL Server 会产生相应的警示：

```
SELECT sc.CustomerID,

        COUNT(soh.SalesOrderID) AS '@OrderCount'
FROM Person.Person pp
JOIN Sales.Customer sc
    ON pp.BusinessEntityID = sc.PersonID
JOIN Sales.SalesOrderHeader soh
    ON sc.CustomerID = soh.CustomerID
WHERE sc.CustomerID = 29484 OR sc.CustomerID = 29485
GROUP BY sc.CustomerID
FOR XML PATH;
```

这里所完成的是让 CustomerID 回到自己的元素中。乍一看，似乎应当得到带有 OrderCount 属性的 CustomerID 元素，但是结果却不是这样的：

```
Msg 6852, Level 16, State 1, Line 1
Attribute-centric column '@OrderCount' must not come after a non-attribute-centric
sibling in XML hierarchy in FOR XML PATH.
```

哪里出错了？简单答案就是系统不知道属性属于哪个元素——它是行的属性，还是 CustomerID 的属性？

/

是的。一个反斜杠。类似于以@这个特殊字符，表示你要完成的特殊任务。实际上，使用它来定义路径——一个将元素及其他归属于该元素的内容相关联的层次。除首字符外，列名称的任何位置都可以使用此符号。为了说明这一点，下面使用最后这个(失败的)例子，按错误指示修改一下，使其成为我们希望得到的程序。

首先需要修改 OrderID，指示它归属于什么元素。

```
SELECT sc.CustomerID,

        COUNT(soh.SalesOrderID) AS 'CustomerID/OrderCount'

FROM Person.Person pp
JOIN Sales.Customer sc
    ON pp.BusinessEntityID = sc.PersonID
JOIN Sales.SalesOrderHeader soh
    ON sc.CustomerID = soh.CustomerID
WHERE sc.CustomerID = 29484 OR sc.CustomerID = 29485
GROUP BY sc.CustomerID
FOR XML PATH;
```

添加“/”，将 CustomerID 放置在斜线之前，就可以告诉 SQL Server，OrderCount 在层次中位于 CustomerID 之后。现在有很多种方法构建 XML 层次结构，下面看一看 SQL Server 是如何利用这些方法的：

```
<row><CustomerID>29484<OrderCount>7</OrderCount></CustomerID></row>
<row><CustomerID>29485<OrderCount>4</OrderCount></CustomerID></row>
```

现在，我们希望 OrderCount 成为 CustomerID 的属性，虽然 OrderCount 在层次中位于

CustomerID 之下,但是它还没有处于我们所希望的位置上。为此,可以组合使用@和/符号,不过需要定义完整的层次。因为我觉得这里可能有点让人困惑,所以下面分两步走——首先,尝试使用这个方法,但是这会生成与早先的例子类似的错误:

```
SELECT sc.CustomerID,
COUNT(soh.SalesOrderID) AS 'CustomerID/@OrderCount'

FROM Person.Person pp
JOIN Sales.Customer sc
ON pp.BusinessEntityID = sc.PersonID
JOIN Sales.SalesOrderHeader soh
ON sc.CustomerID = soh.CustomerID
WHERE sc.CustomerID = 29484 OR sc.CustomerID = 29485
GROUP BY sc.CustomerID
FOR XML PATH;
```

错误如下:

```
Msg 6852, Level 16, State 1, Line 1
Attribute-centric column 'CustomerID/@OrderCount' must not come after a nonattribute-
centric sibling in XML hierarchy in FOR XML PATH.
```

为了修正这个错误,我们需要了解一下构建 XML 标签的原理。重要的是,标签实质上是按照列出顺序被构建的。因此,如果要向一个元素添加属性,就需要留意它们也是元素标签的一部分——这意味着你需要在定义该元素的任何其他内容(子元素或原始文本)之前必须先定义属性。

在这个例子中, CustomerID 被设置成原始文本,而将 OrderCount 设置为一个属性(这可能看起来和现实世界中相反,但是这里先这样假设吧)。这意味着我们告诉 SQL Server 的事情是反着的。在它看到 OrderCount 信息的时候,它已经处理好了 CustomerID 的属性而且已经不能回头了。

因此,要修正这个问题,只需在告诉它更多的元素或原始文本之前,首先要告诉它有关属性的信息:

```
SELECT COUNT(soh.SalesOrderID) AS 'CustomerID/@OrderCount',
sc.CustomerID

FROM Person.Person pp
JOIN Sales.Customer sc
ON pp.BusinessEntityID = sc.PersonID
JOIN Sales.SalesOrderHeader soh
ON sc.CustomerID = soh.CustomerID
WHERE sc.CustomerID = 29484 OR sc.CustomerID = 29485
GROUP BY sc.CustomerID
FOR XML PATH;
```

这看上去可能不够直观,不过也应该考虑事物被写入的顺序。必须首先写入属性,然后才可以写入 CustomerID 元素的较低级别信息。运行它,就可以看到相应的结果:

```
<row><CustomerID OrderCount="7">29484</CustomerID></row>
<row><CustomerID OrderCount="4">29485</CustomerID></row>
```

OrderCount 现在已经被移到属性的位置上了,正如我们所希望的,真正的 CustomerID 仍然是

嵌在元素的原始文本。

提示:

按照这种顺序逻辑进行操作,因为它适用于几乎所有的情况。因此,如果希望 CustomerID 也成为属性,而不是原始文本,同时还希望它位于 OrderCount 之后,那么可以这样做——只需要确保它出现在 OderCount 定义之后即可。

更多内容...

前面已经说过,XPath 本身很复杂,值得写成一本书。不过并不是所有的内容都已经在前面章节中介绍过了。

@和 / 为你提供了非常好的灵活性,允许你选择自己期望的方式,构建 XML 输出。而且几乎可以满足大多数简单应用的需求。但是,如果还要进行更多的工作,还可以使用其他更多的功能。例如:

- 将数据“通配化”,这样就不用将数据分成一列一列,而且将全部数据当作文本数据处理;
- 嵌入 XML 数据类型列中的本机 XML 数据;
- 使用 XPath 节点测试——这些是一组特殊的 XPath 指令,它们可以改变数据的行为;
- 使用 data()指令允许将多个值看作 XML 中的一个数据点而一起运行;
- 使用名称空间。

4.2.2 OPENXML

这里已经占用了大量篇幅来说明如何将关系数据转换成 XML。SQL Server 还必须允许你打开一个 XML 字符串,并按照 SQL 所期待的那样,用表的形式来表示它,这样似乎是很直观的。

作为一个行集函数,OPENXML 可以像其他行集函数(例如,OPENQUERY 和 OPENROWSET)一样打开字符串。这意味着你可以连接到一个 XML 文档中,甚至借助 INSERT...SELECT 或者 SELECT INTO 将其作为输入数据的源头使用。主要的不同之处在于,它要求使用几个系统存储过程来准备文档,并在使用后清空内存。

可以使用 sp_xml_preparedocument 设置自己的文档。它将字符串移动至内存,并对其进行预解析以优化查询性能。XML 文档将保留在内存中,直至显式地删除它或断开调用 sp_xml_preparedocument 时所使用的连接。

提示:

下面说点别的,我并没有要系统来为我们清理内存的习惯。如果你是这样做的,那么应该在完成操作之后主动清理(我最小的孩子乱丢自己的玩具时,我就会这样教育她)。

与 VisualBasic、C#以及其他大多数支持对象离开作用域后自动被清除的语言类似,SQL Server 也支持清理准备好的文档。请不要懒惰地依赖这种方法——应该在操作完成后自己清理!通过显式释放它(使用 sp_xml_removedocument),就可以确保执行清理操作,在最快的时间内从内存中清理它们,同时也让执行该操作的代码更加清晰。

其语法非常简单:

```
sp_xml_preparedocument @hdoc -<integer variable> OUTPUT,
[, @xmltext -<xml>]
```

```
[, @xpath_namespaces - <url to a namespace>]
```

注意:

如果要提供名称空间 URL，必须将其写在 < 和 > 符号内（例如，<root xmlns:sql="run:schemas-microsoft-com:xml-sql">）

该存储过程中参数的含义是不言自明的：

- @hdoc——如果你曾经进行过 WindowsAPI 编程（以及其他类似的操作，不过这是最常见的），那么你以前就会见过“h”——这是匈牙利命名法中对句柄的表述。句柄实际上就是一个指向存储着某些内容（任何事物都可以）的内存块指针。在本例中，这是我們要求 SQL Server 保持并进行解析的 XML 文档句柄。这是一个输出变量——存储过程返回后，此处引用的变量会包含 XML 的句柄。因为在使用 OPENXML 的时候需要用到它，所以请确保将它另外保存。
- @xmltext——顾名思义，这就是要进行解析并使用的实际 XML。
- @xpath_namespaces——使 XML 正确运行所需的任何名称空间引用。

在调用存储过程并将文档的句柄保存妥当之后，就可以使用 OPENXML 了。其语法稍微有点复杂：

```
OPENXML(<handle>,  
        <XPath to base node>  
        [, <mapping flags>])  
[WITH (<schema Declaration>|<table Name>)]
```

这里已经讨论过这个句柄了——它会是一个整数值，是调用 sp_xml_perparedocument 接收到的输出参数。

在调用 OPENXML 的时候，必须提供节点的 Xpath，这将作为所有查询的起点。通过相对于这里设置的基节点进行导航，模式声明可以引用 XML 文档中的所有部分。

下一个要讲的是映射标志。它帮助我们确定希望在 OPENXML 的结果返回元素还是属性。表 4-5 列出了选项：

表 4-5

字节值	说 明
0	除了不能将它与 2 或 8 来进行组合(2+0 还是等于 2)之外，它与 1 相同。这是默认值。
1	除非和 2 组合(接下来说明)，否则只使用属性。如果没有带有指定名称的属性，那么返回 NULL。它也可以与 2 或(和)8 相加来组合行为，但是该选项比选项 2 优先级更高。如果 Xpath 发现属性和元素具有相同的名称，那么会选中属性。
2	除非与 1 组合(前面已经说明过)，否则只选择元素。如果没有指定名称的元素，那么返回 NULL。它也可以与 1 或(和)8 结合来组合行为。如果与 1 组合且存在属性，那么映射到属性，如果不存在属性，那么使用元素。如果元素不存在，则返回 NULL。
8	可以与 1 或 2(前面已经说明过)组合。使用的数据不应该被复制到溢出属性 @mp:xmltext(你必须使用元属性模式项来获取)。如果不使用元属性(绝大多数情况下你不会这样做)，建议使用这个选项。它会减少(非常小的)一部分操作量。

最后是模式或表。如果定义了一个模式，但是又不熟悉 Xpath，这部分就有点麻烦。幸运的是，这个特殊 Xpath 的使用不是非常复杂，很快你就会习惯(它的工作方式类似于 Windows 中的目录，不过功能要强大许多)。

模式会随着声明方式的不同而变化。定义声明如下：

```
WITH (
    <column name> <data type> [{<column XPath>|<metaproperty>}]
    [,<column name> <data type> [{<column XPath>|<metaproperty>}]
```

- 列名就是要获取的元素或属性的名称。也可以用它构建 SELECT 列表以及完成 JOIN 等操作。
- 数据类型可以是任何有效的 SQL Server 数据类型。因为 XML 中有一些数据类型在 SQL Server 中并不能找到对等物，必要的时候会自动进行强制类型转换，不过这通常都是可以预见的。
- Xpath 列就是 Xpath 模式(与你为 OPENXML 函数创建的起始节点相关)，用于获得针对列的节点——获得元素还是属性取决于前面所描述的标志(flag)参数。如果没有使用它，那么 SQL Server 就认为你希望将当前的节点定义为 OPENXML 语句的起始点。
- 元属性是可以在 OPENXML 查询中引用的一系列特殊参数。它们描述了你在 XML DOM 中所感兴趣部分的各个方面。要使用它们，在它们外面加上单引号，并将它们放置到 Xpath 列中。可供使用的元属性包括：

- @mp:id——不要同我们在 EXPLICIT 中看到的 XML id 混为一谈。当类似的函数使用该属性时，它是 DOM 节点的唯一标识(在文档的作用域内)。不同之处它是系统生成的——因此，可以肯定它是存在的。只要文档被保留在内存中，就可以确保引用相同的 XML 节点。如果 id 为 0，则为根节点(因为要指向更低的位置，所以它的 @mp:parentid 属性为 NULL)
- @mp:parentid——与之前描述的一样，只针对父级。
- @mp:localname——为节点提供不完全限定的名称。它带有一个前缀和一个名称空间 URI(统一资源标识符——你会发现它从 URN 开始)来命名元素或属性节点。
- @mp:parentlocalname——与之前描述的一样，只针对父级。
- @mp:namespaceuri——为当前元素提供名称空间 URI。如果该属性值为 NULL，则不指定名称空间。
- @mp:parentnamespaceuri——与之前描述的一样，只针对父级。
- @mp:prefix——存储当前元素名的名称空间前缀。
- @mp:parentprefix——与之前描述的一样，只针对父级。
- @mp:prev——存储某个节点前一个节点的 mp:id。使用它，可以说明层次中当前级别里的元素排序信息。例如，如果 @mp:prev 值为 NULL，那么它将是树中当前级别里的第一个节点。
- @mp:xmltext——该元属性用于处理目的，包含当前元素的真正 XML。

当然，可以绕过所有这些参数，为自己节省大量的工作。如果有表与在 XML 中指定的 Xpath 起始点直接关联(名称和数据类型)，就必须这样做。如果确实存在这样的表，那么只需要对它进行命名，SQL Server 会自行转换。

好吧，这里要处理的事情已经很多了，不过还没有结束。你可以看到，完成了 XML 的所有操作时，需要调用 `sp_xml_removedocument` 清理存储了 XML 文档的内存。值得庆幸的是，它的语法非常简单：

```
sp_xml_removedocument [hdoc = ]<handle of XML doc>
```

提示：

同样，我又得再次强调一下，养成在操作完毕后清理内存习惯非常的重要。我知道，我这样说可能有点啰嗦。好吧，再啰嗦一下，SQL Server 会在你完成操作之后为你清理内存，不过，你不能每次都指望 SQL Server 为你清理。断开连接的时候，SQL Server 会进行清理，但是如果使用了连接池怎么办呢？如果系统处于加载中，那么某些连接可能从不会断开。使用存储过程可以方便地完成清理，所以动手去做吧——每次都这么做！

很好，我确信你已经在等我介绍如何真正使用它了——所以现在可以介绍所有重要的例子了。

假设你同其他公司合并了，而且需要将它们的一些数据导入到自己的系统中。在本例子中要做的就是导入一些他们公司的出货供应商，这些供应商是我们公司没有的。下面给出了一个从 XML 文档中导入这些信息的脚本实例：

```
USE AdventureWorks2008;

DECLARE @idoc          int ;
DECLARE @xmldoc        nvarchar(4000);

-- define the XML document
SET @xmldoc = '
<ROOT>
<Shipper CompanyName="Billy Bob's Pretty Good Shipping" Base="4.50"
Rate="1.05"/>
<Shipper CompanyName="Fred's Freight" Base="3.95" Rate="1.29"/>
</ROOT>
';

PRINT @xmldoc;
--Load and parse the XML document in memory
EXEC sp_xml_preparedocument @idoc OUTPUT, @xmldoc;

--List out what our shippers table looks like before the insert
SELECT * FROM Purchasing.ShipMethod;

--See our XML data in a tabular format

SELECT * FROM OPENXML (@idoc, '/ROOT/Shipper', 0) WITH (
    CompanyName      nvarchar(40),
    Base              decimal(5,2),
    Rate              decimal(5,2)) ;

--Perform and insert based on that data
INSERT INTO Purchasing.ShipMethod
```



```

(Name, ShipBase, ShipRate)
SELECT * FROM OPENXML (@idoc, '/ROOT/Shipper', 0) WITH (
    CompanyName      nvarchar(40),
    Base              decimal(5,2),
    Rate              decimal(5,2));

--Now look at the Shippers table after our insert
SELECT * FROM Purchasing.ShipMethod;

--Now clear the XML document from memory
EXEC sp_xml_removedocument @idoc;

```

看来最终得到的结果集正像我们所希望的那样(注意, 为了简洁起见, 这里删去了最后的两列):

ShipMethodID	Name	ShipBase	ShipRate
1	XRQ - TRUCK GROUND	3.95	0.99
2	ZY - EXPRESS	9.95	1.99
3	OVERSEAS - DELUXE	29.95	2.99
4	OVERNIGHT J-FAST	21.95	1.29
5	CARGO TRANSPORT 5	8.99	1.49
6	Billy Bob's Pretty Good Shipping	4.50	1.05
7	Fred's Freight	3.95	1.29

这并不完美, 不过的确有效——XML 被转换成了关系数据。

4.3 有关 XML 索引的提示

在讨论其他一些 SQL Server 索引结构之前, 不会讨论 XML 索引。不过这里要花费点时间提醒你可以在 XML 数据上构建索引。第 7 章会更加全面地讨论这些问题。不过, 现在我要确保你在设计和性能期望中考虑了 XML 索引。

4.4 层次数据概述

XML 实际是分层次的。根以及相应的分支元素和属性就可以说明这一切: 某个元素位于另一个元素的上层。自 SQL Server 2005 之后, XML 就具备了索引能力, XML 数据类型内部并不允许以真正分层的方式处理 XML。

从 SQL Server 2008 开始, 为了处理层次数据, 明确创建了一种新的数据类型——HierarchyID 数据类型。我要确保你已经意识到这种新数据类型是可以按照关系格式跟踪层次数据的工具。当你需要以 XML 格式而不是传统的数据存储格式存储数据的时候, 这样做就是非常有深意的。

第 7 章会继续讨论 HierarchyID 和其他的层次设计问题, 不过你要牢记它们之间的关联。你可能会发现, 为了更快地响应层次问题, 需要存储 XML 数据在树型体系结构中的深度的信息。

4.5 小结

自 XML 先于 SQL Server 2000 在网络发布之后, SQL Server 的 XML 部分已经变得很大, 而且还会越来越大。XML 是近 20 年来对行业形成冲击的最重要技术之一。它提供了一种灵活的、极其移植性的方式描述数据, 而且 SQL Server 现在已经有越来越多的方式可以满足你的 XML 需求。

本章回顾了将关系数据设为 XML 格式的方式, 以及如何将 XML 数据设为关系结构。而且还介绍了 SQL Server 可以直接使用 XML 方法提供 Web 服务数据。





第 5 章

细心推敲，大胆设计

这里必须再一次考虑你究竟了解了多少相关知识。“规范化，还是不要规范化——这是一个问题！”真正的问题是你是否已经理解设计关系数据库的最基本准则。既然你已经具备了一定的经验可以学习这本书，那么这里假设你已经听说过这里使用的方法，而且知道它很重要，甚至还已经掌握了它的要领。这里假设你只需要填补知识空缺，而不必从头学起。

除了第 3、4 章以外，本书都采用联机事务处理(OLTP)来展开示例。请不要误会我的意思，这里偶尔会指出 OLTP 与面向分析的 OLAP 之间的区别。重点是，在大多数示例中你会看到为最常见的数据库类型——OLTP 优化的表设计。因此，表示例的数据库布局在绝大多数情况下都被规范化为第三范式。

本章首先简要介绍“什么是范式”这个问题，随后立即叙述一些更加高级的概念。不过在当前阶段，假设已经把数据分开归类为符合逻辑的、不重复的形式，而且可以轻易地重新组合这种形式。除了规范化(规范化数据库的过程)，这里还会研究 OLTP 和 OLAP 数据库的特征。而且，由于这两个主题之间似乎没有太多的内容可以探究，所以这里还研究了很多例子，以便了解如何在整体解决方案中实现那些前面已经学习过的约束。

5.1 进一步了解规范化

如果你已经阅读了由清华大学出版社引进并出版的《SQL Server 2008 编程入门经典》，那么完全可以跳过本节继续阅读下面的更加深刻的内容。

首先要说明，有 6 种规范形式(根据学者观点的不同，数量上可能会有增减)。这个问题留待学术界专家考虑吧。实际上，通常只使用三种规范形式。一个完全规范化的数据库通常是指规范化为第三范式的数据库。

提示：

在编程中，规范化是一个被大量引用但是仍然被误解的概念。所有人都认为自己理解规范化，而且很多人至少确实理解了它的理论知识。但遗憾的是，很多数据库设计者往往倾向于把它当作一种标签——只是一种符号，显示他们是“真正的”数据库工程师。但是，它实际上只是一种符号——数据库设计者知道规范形式的含义——仅此而已。规范化其实是数据库设计宏图中的

一部分。有时，需要规范化自己的数据——虽然有时候也需要特地对数据反规范化。甚至在规范化的过程中，也经常存在很多方法实现一个规范化的数据库。

我的观点是，规范化是一种理论，而且这就是规范化的全部。当选择执行或者不执行规范化的策略时，你所拥有的只是一个数据库——希望它是可能设计出的最好的数据库。不要局限于书中(包括本书)所描述的处理方法——要针对自己所处的情况做出正确的处理。作为本书的作者，我所能做的就是将概念介绍给你——我不能为你实现它们，而且其他作者也不能这么做(至少不能使用写出的文字)。你必须在这些概念中挑选出最好的匹配和最佳方案。

到目前为止，在你的数据库开发背景中，我希望你已经理解了如何创建主键以及在表中使用主键的原因——如果只想对一行进行操作，那么，必须能唯一标识出那个行。规范化的概念很大程度上依赖于围绕主键定义的问题，以及哪些列依赖于主键。在规范化中，可能经常会听到下面的短语：

键，整个键，没别的，只有键。

下面还有一个非常有趣的扩充：

键，整个键，没别的，只有键，所以，Codd 请帮帮我！

这是对力图达到第三范式的规范化给出的一个超级简短的概括(对于那些不知道 Codd 的人，这里说明一下，Codd 被认为是关系设计之父)。当所有的列仅依赖于整个键时，就达到了第三范式。

下面将回顾各种不同的规范形式，并说明每一种规范形式的作用。

5.1.1 入手点

关系数据库的设计建立在“实体(entity)”和“关系(relation)”的概念上。如果你熟悉面向对象的编程，那么就可以把顶层实体看成是对象模型中的对象。就像父对象可能包含进一步描述它的其他对象，表可以有子表或其他表，后者进一步说明原始表中的行。

一个实体一般依赖于一个“父”表。通常，对于正在描述的实体的每个实例，表中有且仅有一个行与其相对应(例如，在系统中跟踪订单的上层表中，每一个单独的订单只有一行)。但是，一个实体可能需要多个表来提供附加的描述信息(例如，用一个细节表或项目表记录在特定订单中购买的所有商品清单)。

关系表述了两个实体相互之间进行逻辑相关的方式。例如，顾客与订单是不同的实体，但它们之间是相关的。例如，一个订单中至少要有一位顾客。此外，订单只能与一位顾客相关。

在开始对这些实体和关系进行“规范化”以形成表的时候，即使在达到第一范式之前，就已经形成了下面的一些关于数据的假设：

- 表应当描述一个实体，而且只能描述一个实体(不要企图走捷径合并事物！)；
- 所有的行必须是唯一的，而且这里必须存在主键；
- 列和行的顺序必须是无关紧要的。

提示：

随着经验的积累，这将不再是一个“过程”，而更像是设计自己的表时自然的起点。你会发

现, 创建一组规范化的表就是从自己脑中自然流露的想法, 而不是必须完成的特别的事情。

5.1.2 达到第三范式

正如之前提过的, 从实践的观点看, 有三种规范形式:

- 第一范式(1NF)是为了消除重复的数据组同时保证原子性(atomicity, 即数据是独立的)。在较高层次上, 通过下面的步骤可以实现第一范式: 创建主键(已经有主键), 然后将所有重复的数据组移至新表中, 为这些表创建新键, 依此类推。另外, 把那些将不同数据组合形成单独的行的那些列进行拆分。
- 第二范式(2NF)进一步减小重复数据的发生率(不必要是数据组)。第二范式有两条准则:
 - 表必须满足第一范式(规范化是一种类似于堆积木的过程——如果前两块没有建好, 则无法叠放上第三块)。
 - 每一列必须依赖于整个键。
- 第三范式(3NF)处理的问题包括: 表中所有的列不但要依赖于某个事物, 而且所依赖的还必须是正确的事物。第三范式有三条准则:
 - 表必须是满足第二范式(前面说过这是一个类似堆积木的过程)。
 - 任何列不能与任何非主键列有依赖关系。
 - 不能有派生的数据(即可以从表中的其他数据推导出该数据)。

5.1.3 其他的规范形式

从学术角度讲, 还有一些其他的形式被视为规范化模型的一部分。它们包括:

- Boyce-Codd(实际上被认为是第三范式的变体)——该规范形式试图处理多个候选键相互交叠的情况。这种情况只可能出现在:
 - a. 所有候选键都是组合键(即, 键由一个以上的列组成);
 - b. 候选键不止一个;
 - c. 每个候选键至少有一列与另一个候选键一致。

通常情况下, 这是一种有许多种可行解决方案的情形, 而且学术界之外几乎从来不考虑它。

- 第四范式(4NF)——该规范形式试图处理与多值依赖有关的问题。在这种情形中, 在每一行中, 没有任何列依赖于主键之外的列并且依赖于整个主键(满足第三范式)。不过, 可能出现非常奇怪的情形, 主键中的一个列可以单独依赖于主键中的其他列。这种情况很少见, 而且通常在实际中不会引起任何问题。因此, 数据库领域通常会忽略它们, 因此这里也不会对其进行更深入的介绍。
- 第五范式(5NF)——处理无损分解和有损分解的问题。本质上, 必然存在某种情形: 可以对关系进行分解, 但是这样的分解在逻辑上不能够再重新组合回到其原先的形式。同样, 这种情况非常少见的, 普遍上是学术性的, 所以这里不会做进一步的介绍。

当然, 这里只是匆匆带过这些规范形式——这里是故意这样处理的。在现实世界里, 了解它们的主要目的要么是为了给你的朋友留下深刻印象(或者证明你“见多识广”), 要么是为了当某个数据库权威在场并开始谈论它们的时候, 你不会因为不知道而令自己显得像个傻瓜。但是我建议你不要企图利用这些知识来获取约会。

5.2 关系

我经常听女人们说起如果提到“关系(relationship)”这个词，男人们会立刻离开房间。有了这样的一种想法，我希望自己不会失去一半的读者。

当然，我只是在开玩笑——不过并不完全像你所想的那样。专家们说，人际关系成功的关键在于，你清楚双方的任务而且每个人都理解自己所处关系的边界和规则。我可以用陈述人们之间的关系的的方式来谈论数据库中的关系。

不同的主要关系有 3 种：

- 一对一——该关系就像它所声明的那样。一对一关系是说：一个表中有一条记录意味着在另一个表中刚好有一条与之匹配的记录。
- 一对多——这是一种普通的、一般的、日常可见的外键类型的关系。通常情况下，它存在于某种主/细关系中，而且一般会采用父对子的层次结构。例如，对于每一位顾客，可以有好几个订单。
- 多对多——在这种类型的关系中，关系的双方可以有几条匹配的记录。产品与订单的关系就是这种关系的一个例子——一个订单可能包含几种产品，类似地，一种产品也可以出现在多个订单中。SQL Server 无法直接构建物理的多对多关系，因此，可以使用一个中间表来建立这种关系。

根据关系中的一方是否允许为空值，上述关系中的每一种都有一些变体。例如，可能并非是一对一关系，而是一对零或者一对一关系。

5.3 图表

实体关系图(ERD)在良好数据库设计中是一个重要工具。通常，不需要事先拟订计划就可以很轻松地由若干脚本中创建小型数据库并直接执行它。但是，随着数据库的增大，只是“在脑中想想”这样的方式很快变得问题重重。因为 ERD 允许快速查看实体及其之间的关系并理解它们，所以 ERD 解决了很多问题。

提示：

在本书中，我决定用一种与之前相反的方法进行讲述。SQL Server 中包含非常基本的关系图工具，可以使用它着手创建 ERD 的雏形。但是，它使用一套专有的关系图方法，远远看上去，与我所知道的任何标准都不相似。此外，它不允许使用逻辑模型——我认为这个概念相当重要。因此，这里首先会介绍更为标准的关系图方法学——在本章的后面，将讨论 SQL Server 的内置工具以及如何使用该工具。

有两种很常见的关系图范型——IE 和 IDEF1X。这两种范型都应用广泛，不过这里主要学习的是 IE(也就是所谓的信息工程，Information Engineering)的基础知识。据资料显示，IDEF 是一种由美国空军首次推行的、极其完美的关系图范型。但是，IE(再说明一下，它是指信息工程，而非 Internet 浏览器)是我个人正在使用的方法，我使用它只有一个原因——对于尚无经验的看图人来说，它直观得多。我还发现在两种方法中它更为普及。

注意:

关于选择正确工具的重要性我所说的还远远不够。虽然内置的工具至少向你提供了“可用的一些东西”,但是它们距离你所需要的还很远。

ER 工具绝不廉价——可以高达 1000 美金到 5000 美金不等(这是每个席位的价格)。它们本身也是一种语言。不要妄想坐下就可以使用任何主要的 ER 工具——你最好花费一定的时间使它能够完成你期望的任务。

不要因为这些工具的高昂价格而放弃建立逻辑模型。虽然 Visio 还是不能回答世界上所有的数据库设计问题,但是它可以胜任逻辑建模方面的工作而且可以在某种程度上完成同步和物理建模。也就是说,如果你对数据库设计确实感兴趣,并且将要完成大量的数据库设计工作,那么你确实需要购买真正的 ER 工具。

抛开费用的因素,第三方工具与内置工具的生产力根本无法相提并论。借助所选择的 ER 工具,可以完成下面的工作:

- 创建逻辑模型,随后在逻辑模型和物理模型之间来回切换;
- 离线对关系图进行操作——一次性(准备就绪时,与之相对的是需要注销时)将所有的改动传送到物理数据库中;
- 若系统是一种主流的关系数据库管理系统(甚至是 ISAM 数据库),可以对来自该系统的数据实施逆向工程,然后把它们正向工程到完全不同的关系数据库管理系统中;
- 在众多不同的系统上创建物理模型。

实际上,这并不是核心内容,只是一些皮毛。

5.3.1 几种关系类型

在更深入了解关系图概念之前,这里先讨论两种类型的关系:标识关系和非标识关系。

1. 标识关系

对某些读者来说,我相信“标识关系(identifying relationship)”一词会令你想起过去结识过的一些占有欲比较强的男女朋友——这里不是指那种关系。实际上,这里要处理由外键定义的关系。

标识关系是指:(父表中)被引用的列(可以多于一个)是引用(子)表的部分或全部主键。既然主键标识了表中的所有行,而且子表中的部分或全部主键依赖于父表——因此可以说子表(至少部分)由父表“标识”。

2. 非标识关系

非标识关系是在设立外键时创建的,此时的外键并不属于引用(子)表的主键。在引用域表的时候,经常可以见到这种关系——实际上,被引用表的唯一目的是将引用字段限制在一组可能的选项范围内。

5.3.2 实体框

实体框是 IE 和 IDEF1X 与 SQL Server 关系图之间最主要的差别之一。根据当前处理的是逻辑模型还是物理模型,实体框大致等同于表。通过查看实体框,可以轻易地确定实体的名称、主

键以及该实体的所有属性(实际上是列)。此外,关系图可以显示其他信息,比如属性的数据类型或者该属性上是否定义了外键。图 5-1 中的实体框给出了一个具体的实例:

实体的名称位于框外部的顶端。在整个框的顶部区域,有一个单独的主键框(不久就可以看到主键不止一例的例子),实体的属性虽然在最后但仍然很重要。

下面来看一个稍有点不同的实体(图 5-2):

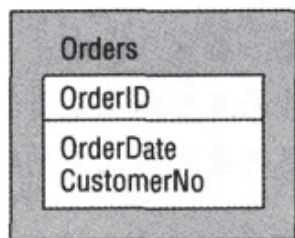


图 5-1

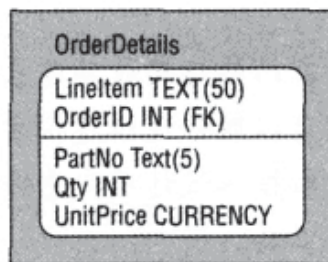


图 5-2

这里出现了一些新的东西:

- 数据类型(我已经打开了适当选项);
- 外键(如果有外键的话——类似地,我已经打开了显示外键的选项);
- 主键中有多个列(实体框中分隔线的上部都是主键);
- 实体框的边角这次是圆形的,这说明至少有一个别的表引用了该表(还记得前面讲过的标识关系吗?)。

可以直接在 ER 图中定义数据类型,具体取决于 ER 工具。在绘制构成关系的线条时(很快就可以看到它们),还可以定义外键,也可以显示这些外键。对于大多数 ER 工具而言,可以使工具在外键关系中自动把被引用字段定义为引用表中的部分(或者可能是全部)主键。

5.3.3 关系线

关系线有两种,它们与关系类型完全匹配。

实线表示标识关系:

虚线表示非标识关系:

再强调一下,在标识关系中,对另一个表进行引用的列是引用表中的部分或全部主键。在非标识关系中,外键列与引用表中的主键没有任何关系。

5.3.4 终止符

这里事情变得更加有意思了。当然,此处谈论的终止符(terminator)并非阿诺德·施瓦辛格在电影里扮演的那种终结者——这里的终止符是指放置在关系线上的端点。

关系线上的终止符传达的关于数据库性质的信息与实体本身所传达的信息差不多或者更多。它们将告知你包括关系的基数在内的关系真实本质的信息。

最基本的基数是指关系两端的记录数。在提到一对多关系的时候,就是在说明基数。但是,基数能够比 0、1 或许多通用的命名规则更加具体。基数可以表明具体细节,在关系图中,基数常常扩充为两个数字和一个冒号的形式,如:

- 1:M;

- 1:6(该表达式不但符合一对多的标准, 而且描述更加详细, 说明了在关系的那一边记录的最大值为 6)。

下面看一看终止符的组成部分, 并查看其含义:

注意:

提醒一下, 下面的终止符来自 IE 关系图方法论。我之前说过, 还有另一种广泛使用(虽然我在 IE 中很少见到)的关系图标准——IDEFIX。虽然它的实体框与 IE 的类似, 但是关系线上的终止符却完全不同。

图 5-3 中终止符的上半部分指示出了关系的第一部分。此处为零方。图的下半部分说明了关系的第二部分——此处为多方。在本例中, 我们得到了关系的零、一或者多方。

在图 5-4 中, 关系中的这一端不允许为空——这是关系中的一方或多方。

在图 5-5 中, 再次允许关系的这一端为零, 但是现在不允许大于 1。这是关系的零方或者一方。

最后是图 5-6。这个图的限制非常严格——它仅仅是关系的“一”方(不能多也不能少)。

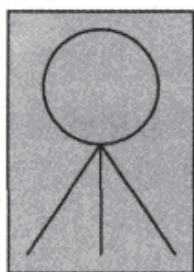


图 5-3

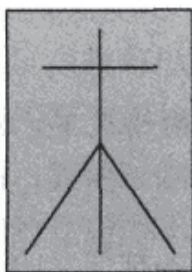


图 5-4

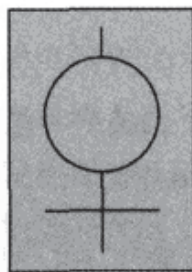


图 5-5



图 5-6

因为只观察终止符可能不太好理解, 所以下面来看一些示例表和关系的实例(图 5-7 所示)

关系图 5-7 显示了支持一个逻辑实体(订单)的两个表。Orders 表用来跟踪订单的整体信息(此处只有一个 CustomerNo, 但是还可能包含运送地址、订单日期、截至日期等信息)。另外, 还有一个 OrderDetails 表, 用来跟踪订单中的每一项。该关系图不仅描述了 Orders 表和 OrderDetails 表, 而且还显示了两个表之间的一(Orders 方)对零、一或多(OrderDetails 方)关系。该关系是标识关系(图中是实线而非虚线), 并且关系名为 OrderHasDetails。

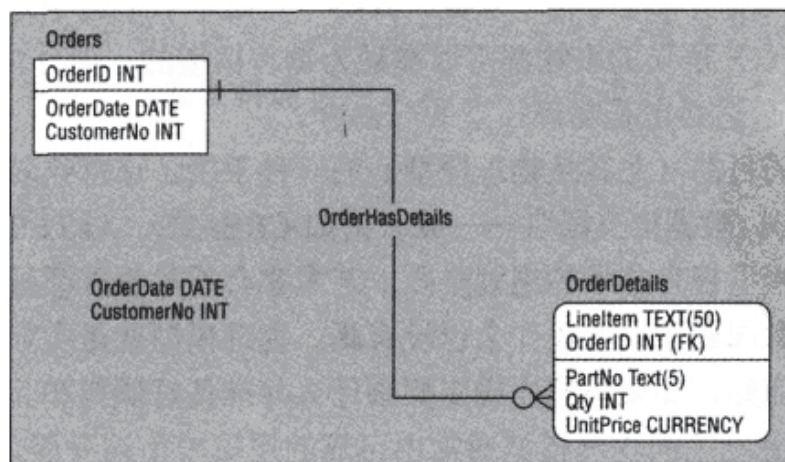


图 5-7

图 5-8 中加入了一个 Products 表:

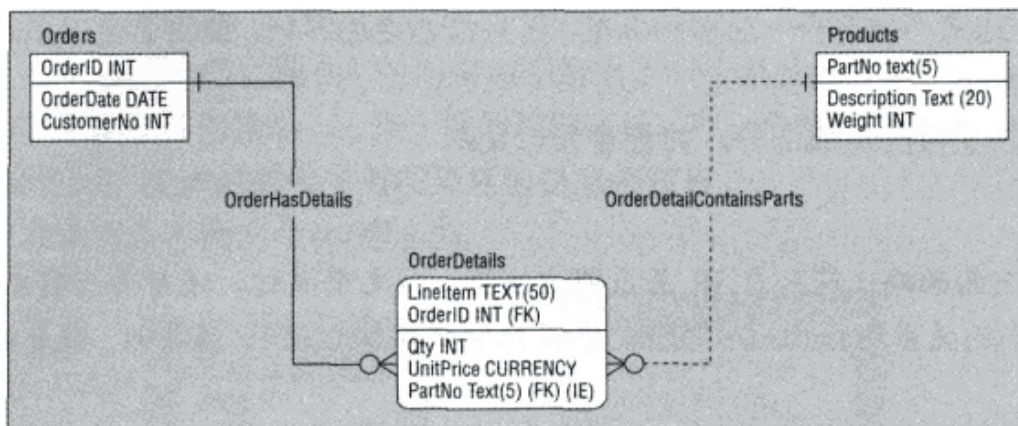


图 5-8

图中新加入的关系与前面已经看到的关系非常类似。它也是一(这次是 **Products** 方)对零、一或多(**OrderDetails** 方)关系, 不过这里的关系是非标识关系(以虚线表示)。IE 表明, 对于本表来说 **PartNo** 是一个倒置项或者只与外键有关的索引。通常加入倒置项是为了在外键字段(由于它是经常查找的目标)上有索引。

综合考虑三个表, 会发现通过 **OrderDetails** 表, **Orders** 与 **Products** 之间存在一种多对多关系。

提示:

注意倒置项可以不和任何项相关——在这种特殊的情况下, 它只是恰好与外键相关。倒置项实质上是不唯一的或者与主键相关的。

前面已经指出, 这里只涉及到了 ER 图能传递的各种信息的皮毛。尽管如此, 当在本章后面讲述 SQL Server 关系图工具时, 你将会发现, 较之内置工具所提供的信息, 使用更常用的方法学可以传达更多的信息。此外, 表显示方式的本质使得信息(如键)更一目了然、便于阅读。

5.4 逻辑设计与物理设计

在数据库的工作中, 你可能已经听说过逻辑模型与物理模型的概念。本节将探讨二者之间的差异。

在两种模型中, 物理模型可能更易于掌握。其实, 它是本书中迄今为止一直在使用的东西。任何能够对它执行 CREATE 语句的事物都可以被视为物理模型的一部分。实际上, 只要能对它运行 SQL Server 语句, 那么它必定是物理模型的一部分。

逻辑模型是达成不同目的(尤其是物理模型)的一种手段。这意味着, 对逻辑模型进行操作时, 就是在朝着生成 DDL(数据定义语言——或者诸如 CREATE、ALTER 和 DROP 此类的语句)的方向努力。可将逻辑模型视为艺术的规划阶段。艺术家心理盘算着要画什么, 准备好颜料, 找一块大小合适的画布, 拿起刷子, 但还什么也没有画。物理模型就是实际的绘画作品。每个人都可以欣赏画作, 但可想而知, 如果不预先确定要画什么并准备好颜料和其他必需品, 就不会有最终的作品。与此类似, 最出色的物理模型通常由可靠的逻辑模型演变而来。

5.4.1 逻辑模型的用途

首先要理解的一点是, 逻辑模型与物理模型有着不同的目的。逻辑模型能完成下列工作:

- 允许着手创建复杂的、与数据有关的业务问题的抽象,并在高层次上标识实体;
- 允许使用这些抽象有效地传达这些与数据有关的业务规则和意图;
- 代表了最纯粹的数据形式(在开始加入真正起作用的因素之前);
- 在项目的数据需求部分中,充当主要的文档。

由于逻辑模型并不完全符合创建数据库的确切语法,因此,它们提供的灵活性是物理模型不能提供的。无论你的关系数据库管理系统是否支持这些规则,都可以在逻辑模型中添加对话和规则。简短来说,在开始把设计细化为特定的具体实现方案之前,你可以向逻辑模型中添加各种事实。

这样做的好处是,逻辑模型允许在同一个地方保存所有的数据规则,而不理会将在什么地方实现这些规则。你可能常常会遇到规则不能在数据库中实现的情况。这些有问题的规则或许是与数据有关的,但是,由于某些约束或要求,需要在客户端或某种中间层中使用更程序化的代码来实现它们。有了逻辑模型,就可以继续前进并将数据规则包括进来。

无论来源如何,逻辑模型中包含了所有与数据有关的信息,创建系统数据的一个或多个抽象。可以把这些抽象用作给客户的说明,告诉他们你真正准备存储的内容以及你认为捕获了什么规则。可以通过打开额外的沟通途径尽早(以及经常)使用这样的说明,为项目省下宝贵的时间和金钱。即使是那些没有什么数据技能的客户也可以经常查看最高级别的关系图,并谈论“哪些地方提出购买请求?”这样的问题。一般情况下,你对于它们的称呼总是有很多现成的解释,这样你就可以在关系图上一一指出——然而,其他时候你可能会发出最令人恐惧的感叹“哎哟!”。我不了解你,但是我个人情愿在项目最初的几周而不是部署开始的几周发出这样的声音。在与客户正确分享逻辑模型时,逻辑模型可以避免在部署的时候狼狈地发出“哎哟”的声音。

提示:

对客户尽早尽可能经常共享逻辑设计(最好是有逻辑设计)的重要性的强调还远远不够。对客户培训一些关于如何阅读自己的逻辑模型的知识(还应该包括良好记录模型中的实体和关系的原因与用途),可以节省宝贵的时间和金钱。

我所遇到的具有实际经验的开发人员都一次或多次地认识到后期改动系统带来的巨大开销。改动代码花费不菲,但是比起在项目后期改动自己的数据库所花费的开销仍是望尘莫及。如果没有很好的抽象自己的数据库,那么对数据库的每一处改动都会影响到许多的代码。换句话说,数据库的一点小改动可能引发访问数据库代码的成百上千次变化(这与系统的大小有关)。

简言之,沟通无所不在,而且逻辑建模应该是与客户沟通的重要工具。

5.4.2 逻辑模型的组成

逻辑模型包含3个主要部分:

- 结构
- 约束
- 规则

这三者的结合应该能够完整地描述系统中的数据需求,不过它们可能不会全部转变为物理模型。逻辑模型中确定的一些问题可能需要以某种过程的形式来实现(例如在中间层组件中)。其他时候,可以通过关系数据库管理系统的各种功能实现完整的逻辑模型。

注意:

这一点确实非常重要,所以我想再强调一下——这样做只是因为你的逻辑模型里有它,并不意味着它存在于你的物理数据库中。逻辑模型会考虑数据的所有需求——甚至包括你的关系数据库管理系统不能实现的那些数据需求(例如,你可以从第三方——也许在 XML 文档中或其他存储媒介中——获得的那些数据)。逻辑模型包含所有的一切,允许你按照一定的方式规划物理设计,以确保解决所有的数据问题——不止是那些在物理上会留在数据库中的问题。

1. 结构

结构是逻辑设计中涉及实际数据存储概念的部分。在处理数据库结构时,你要讨论的是实体(大部分实体将转变为存储数据的表)以及为维护数据的原子性所需要的特定列。

2. 约束

从逻辑模型的观点看,约束(Constraint)比迄今为止你一直使用的约束一词的使用范畴更广。截至目前为止,使用约束一词的时候,表示你在讨论一些具体的特征将数据限定为某些特定值。从逻辑的观点看,约束为数据定义“什么”问题——即,什么数据是有效的。逻辑模型包含约束,这是指它包含如下的内容:

- 数据类型(注意,这的确不同于需要列的原因或者列的名称应该是什么这样的想法);
- 截至目前为止所使用的约束形式——即 CHECK 约束、外键约束、或者甚至主键约束和 UNIQUE 约束(备用键)。每一种约束都提供了什么数据能存在于数据库中的逻辑定义中。该领域也包含域表(使用外键可以对它进行引用)——用来把列中的值限制在特定的“域”列表中。

3. 规则

如果说约束是数据中的“什么”,那么规则将是数据中的“何时以及多少”。

在定义逻辑规则时,我们是在定义:“它需要一个值吗?”(等同于“允许空值吗?”)以及“允许有多少?”(定义了数据的基数——接受一个或是多个?)

值得再次注意的一点是,数据库的物理部分可能不会实现这里所有的操作(我们可能决定完全在客户端处理施加的规则)。无论这个要求在哪里实现,它仍是我们广泛的逻辑数据模型的一部分。只有获得了完整的数据模型,才能知道已经处理了所有的问题(无论是在哪里进行处理的)。

5.5 通过经典的 BLOB 处理基于文件的信息

BLOB(二进制大型对象)。你可能对它们见得还不多,也不讨厌它们。你是否“已经”讨厌它们了,这在很大程度上取决于你是否需要支持向后兼容。

从 SQL Server 2005 开始,微软加入了一些新的数据类型(varchar(max)、nvarchar(max)和 varbinary(max)),而且这些数据类型可以极大地简化 BLOB 的处理。SQL Server 2008 还支持一种称之为文件流的特殊文件级存储选项的形式加入了另一个可供混合的选项(使用自己的客户端编码器会更加复杂,而且需要持之以恒的努力以及对网络进行特殊的考虑)。对于这部分内容,我们不会匆匆一笔带过,下一章(高级数据结构)和更高级性能设计一章(第 21 章)我们会用很大的篇

幅讨论文件流。

当与兼容的数据访问模型一起使用时(ADO.NET 2.0 或更高版本), 可以像访问更小的基本数据类型(varchar, nvarchar 或 varbinary)一样, 通过更标准的方法(也就是不使用文件流)访问 BLOB 数据。遗憾的是, 由于许多人还需要面对向后兼容的问题, 因此不得不使用较早的(并且也是更慢的)“分块(chunking)”方法访问数据。无论使用何种访问方法, BLOB 都很慢——非常慢且大。使用新的访问方法确实可以帮助 BLOB 处理性能, 所以我会鼓励你尽快移植至 SQL Server 2005, 并将其作为自己支持的底线。

注意:

在使用较新的 BLOB 数据类型时, 你支持的最老版本的 SQL Server 是关键因素(而非数据访问方法)。在处理旧的连接方法时, SQL Server 会自动将新的数据类型转换为旧的数据类型。不过要注意, 使用文件流不需要特定的客户端代码。

BLOB 使你能突破行大小 8KB 的限制(BLOB 能够达到 2GB)。在这一点上 BLOB 很不错。首要的问题是在旧的数据类型和访问方法中使用它们很笨拙。不过, 更大的问题或许是它们非常慢(我知道自己又在重复了, 但是我在这里也是表达了一个观点)。如果 BLOB 与乌龟赛跑(龟兔赛跑的续集) 只有当乌龟停下来休息时 BLOB 才能够赢。

提示:

好吧, 我就不再谈论慢了。确实, 这些年 BLOB 的处理性能得到了很大的改进, 变化之处不在于它过去怎么样, 而是要冒着过分重复的危险来说明, BLOB 依然相对较慢。

那么, 你已经听到我说 BLOB 很慢, 不过你仍然需要存储巨大的文本块或二进制信息。通常你会使用 BLOB 来完成这个任务(由于近来 BLOB 处理性能得到改进, 它可能是最合适的——不过你确实还有另一种选择, 能够以不同的方式完成任务。可以通过存储文件来解决这个问题。

注意:

现在你们中可能有人要问“通过数据库访问数据是否比文件系统更快?” 我的答案很简单——“一般不会”。

不使用文件流, 还有两种方式实现。我们首先会讲述传统的实现方法, 随后再讨论另一种 .NET 时代可能的实现方式(它需要自己设计特定的应用程序)。

在此我要先告诫, 为了用典型的方式实现, 必须在客户端预先做准备——在这里数据库服务器并非唯一要做的事情。实际上, 我们将从数据库服务器上去除大部分要做的工作, 然后把它们放到中间层和文件系统中。可以先考虑要在服务器的文件系统上完成的事情。这里唯一要做的事情是, 确保至少有一个目录可以用来存放信息。根据应用程序的类别, 可能还需要中间层对象的逻辑, 以便在需要时允许创建额外的目录。

所有 Windows 操作系统对于一个目录中能够存放的文件数目都是有限制的。随着 64 位操作系统的推出, 增大了每个目录中的文件数目的最大值, 最大值不像原始性能那样问题大(当特定目录中的文件数目增加时, Windows 仍然会在文件访问上变得很慢)。然而, 你还需要考虑要存储多少文件的问题。如果要存储的文件数目很大(例如超过 500), 那么, 可能要在存储 BLOB 的对象中创建一种机制, 以便能够根据需要或者其他一些合理标准创建新的目录。

业务组件将负责将 BLOB 信息复制到用来存储该信息的文件中。如果它已经是某种定义过的文件格式,你的任务将很轻松——只需运行语言中用于复制的命令(带有一点改动,这点我们很快就可以看到),就大功告成。如果它是数据流,那么只有把逻辑放在组件中才能把信息存储为可供日后取用的逻辑格式。

注意:

这种实现存在一种很大的安全问题。既然你要把信息存储在 SQL Server 外的文件中,那么它也就位于 SQL Server 的安全保护范围之外。因此,你必须依赖自己的网络安全性。

这里有好几处地方都可以使你在头脑中冒出“很可怕”的念头。首先,如果有人要从存储所有数据的目录中读出数据,这不就意味着他们可以看到那里保存的其他文件吗?是的,他们可以看到(如果你希望把事情变得很复杂的话,可以通过更改每个文件的 Windows 安全设置来避免这个问题,不过这样做实际上是让人厌烦的——在 Web 应用程序中,你可能需要完成一些操作,比如在自己的 Web 服务器上实现 DLL)。其次,既然你必须给别人权限将文件复制到目录中,那么那里可能有人不使用数据库而直接修改文件,这样做不是很危险么(可能会导致自己的数据库与文件失去同步)?当然。

这些问题和你可能遇到的其他问题的答案就在你自己的数据访问层中(这里假设使用一种 n 层的方法)。例如,可以在不同的安全背景下而不是在终端用户处运行访问组件。这表明你可以创造一种环境,使所有的用户都可以访问自己的数据——不过只有在他们使用数据访问组件完成这项工作的时候才可以(他们没有权限访问目录本身——确实,他们可能并不知道那些文件存储在什么地方)。

在整个过程中,SQL Server 在哪里发挥作用?SQL Server 将记录当前讨论的信息存放的位置。理论上,首先要尽力首先把信息存放在数据库中,这是因为,当把信息存储为行的一部分时,它与行中一些其他信息有关系。不过这里要做的是存储一个你所保存的文件的路径,而不是把实际的数据全部保存在行中。下面给出了具体的存储过程:

- (1) 决定它的存储名称;
- (2) 把文件复制到要存储的位置;
- (3) 把完整的名称和路径以 `varchar` 的形式与行中其他数据存储在起;
- (4) 要获取数据,类似于直接从表中获取数据,直接运行查询,只是眼下获取的是到实际 BLOB 数据存储位置的路径;
- (5) 从文件系统中获取数据。

一般来说,这种方法执行起来比使用 BLOB 快一些。不过需要使用该方法的规则也有一些例外情况:

- 保存的 BLOB 的大小一直比较小(少于 8KB)。
- 数据为文本或者某些格式,MS Search 有适用于这种格式的筛选器,而且你可能希望在其上执行全文搜索。

如果 BLOB 的大小总是小于 8KB,那么数据能够全放在一个数据页上。这样就把处理 BLOB 的开销减到了最小。尽管文件系统方法可能还是会快一些,不过也不会快得很多,因此极大地降低了它的优势。当在你所处的情形中速度是至关重要的,那么我所能提供的建议就是进行试验。

如果要执行全文搜索,那么,或许直接把大型的文本块以 TEXT 数据类型(BLOB)存储在 SQL Server 中会更好一些。如果用带有可用 MS Search 筛选器的二进制格式(或者,若强烈需要的话,

也可以自己写一个)存储文本,那么,可以用图像数据类型存储文件,而且 MS Search 会自动使用筛选器构建全文索引。不要误解我的意思;对文件中的文本进行全文搜索也是行得通的,只不过当你想要从同一个行中获得非 BLOB 数据的时候,必须编写更多的代码以保持关系的完整。此外,你很可能必须规划中间层以使用索引服务器。

如果必须在基于文件系统的信息上做全文搜索,你可能需要尝试一下通过查询直接访问索引服务器。SQL Server 能够发出远程查询,这样你就可以潜在地访问任何 OLE DB(微软发布的数据访问 API——第 25 章会查看其中的细节)数据源。MS Search 服务有一个 OLE DB 提供程序,可以用作链接服务器或用在 OPENQUERY 中。不过,如果索引服务器与你的 SQL Server 不在同一个物理区域,则基于该索引服务器执行的索引服务器查询根本无效。唯一的工作区是让索引服务器位于本地系统的 SQL Server 上,令它的目录文件存储在另外的系统中。这样做的问题是处理目录时出现的网络通话(chatter)以及无法卸下目录工作(这会影晌扩展性)。

以上就是第一种实现方法(你或许还记得我说过有两种方法)。第二种方法将利用 .NET 程序集体系结构(它是 SQL Server 2005 的一部分)。由于我们尚未开始讨论 .NET 集成,因此仅做高度概括。

实际上,这种方法利用了许多在中间层文件访问方法中使用过的相同的概念。唯一真正改变的是什么服务器或组件中负责文件访问。

随着公共语言运行时(CLR)集成的出现,我们可以创建比已经复杂得多的用户定义函数。因此可以定义表值函数,该函数能从几乎所有的基础源中获取数据。实际上,我们将在第 10 章中讨论如何列举目录中的文件并把它们以表值函数的形式返回,不过我们也可以很轻松地返回包含该文件的 varbinary(max)列。在这种模型中,所有的文件访问都将在网络安全上下文中进行,该网络安全上下文是我们为了运行程序集而设立的。不过,它只是作为表值函数的一部分执行。

提示:

必须注意:之前提到的基于文件系统方法可以被视为文件流特征的前身。文件流实现的是这种方法的高级版本——包含协同备份等。也就是说,文件流比这种方法复杂得多——这也就是,我将对这些内容的详细讨论放在高级数据结构和性能章节中。

5.6 子类别

子类别(subcategories)是一种逻辑结构,为你提供另一种可以使用的关系类型(有时称为“超类型”或“子类型”关系)。在物理模型一边,使用已经讨论过的关系类型的混合来实现子类别(在结束前你会看到这方面的细节)。

子类别处理这样的情况,在这里,许多实体初看上去可能不同,但它们在—些(不是全部)地方有共同点。

我认为,理解子类别的概念最好的方法是看一个例子。因此,我们以某个公司的文档为例。这个文档与任何类型的文档之间有许多共同的属性。例如:

- 标题;
- 作者;
- 创建日期;
- 最后修改日期;

- 存储位置。

我相信还有更多的共同属性。注意，这里并不是说所有的文档有相同的标题，而是说所有的文档都有一个标题。所有的文档都有一个作者(实际上可能不止一个，但是，本例中将假设限定为一个作者)。所有的文档都是在某个日期被创建的。明白了吧——我们讨论的是文档这一概念的属性，而不是具体的文档实例。

不过，存在很多不同类别的文档。从法律形式(抵押文档)的东西到办公备忘录，再到成绩报告单——有许多文档类型。尽管如此，每一种文档仍然能被认为是一个文档——或文档的子类别。考虑下面的几个例子。

第一个例子，来看一个租约。租约具有文档类别的所有特征，不过它还包含有其独特的信息。租约中有如下的一些信息：

- 出租人；
- 承租人；
- 期限(租期为多长)
- 费用(每月或每周多少钱)；
- 押金；
- 起始日期；
- 到期日期；
- 选择权(常常用来提供租期的延续，按预先议定的价格为额外的期限支付租金)

租约具有的以上所有这些属性并不会妨碍这样一个事实：它依然是一个文档。

你还可以提出更多的例子，而我将继续给出法律文档的例子——离婚证明。它有如下的属性：

- 起诉人(提出离婚请求的一方)；
- 抗辩人(起诉人的配偶)；
- 分居日期；
- 起诉人提出离婚请求的日期；
- “最终”认定离婚的日期；
- 赡养费(如果有的话)；
- 子女抚养费(如果有的话)

我们可能还有一个销售清单——我们的清单可能包含下面的属性：

- 出售日期；
- 出售金额；
- 卖方；
- 买方；
- 担保期(如果有的话)

同样，离婚证明和买卖契约都有自己的属性，而且这个事实并不会妨碍它们是一个文档的事实。

实际上，我们所拥有的每一种形式(租约、离婚证明和买卖契约)都是“文档”类别的子类别。如果不属于某个子类别的话，文档几乎没有什么意义或完全没有意义。同样，任何子类别的实例如果没有只在超类别(文档)中存在的父信息，也几乎没有多少意义了。

5.6.1 子类别的类型

子类别本身有两种不同的分类——独占的和非独占的。

当仅仅只用“子类别”来提及子类别时,通常指的是这样的子类别安排:其中,一个表中的记录代表超类别(在我们前面的例子中是文档),并且匹配的记录至少属于一个子类别。

与迄今为止见过的符号相比,代表这种类型的子类别的符号似乎有点奇怪(如图 5-9 所示)。

尽管这里以及前面的文档例子中都展示了 3 个子类别,但不要误以为这是对子类别数目的正式限制——这里没有限制。既可以有一个子类别,也可以有十个——实际上,这并不会有什么区别。

更常见的是拥有独占子类别的情形。独占子类别与类别的作用完全一样,只有一个例外——对于超类别中的所有记录,在任何子类别中只有一条匹配的记录。可以认为每一个子类别都是互相排斥的,因此匹配超类别的记录只能在一个子类别表中以一行的形式存在。

独占子类别的图看上去更奇怪一些(如图 5-10 所示)。

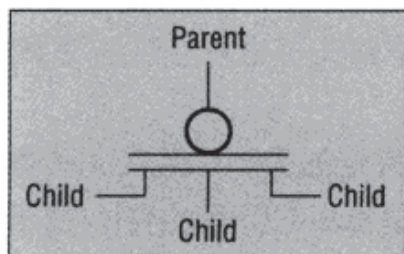


图 5-9

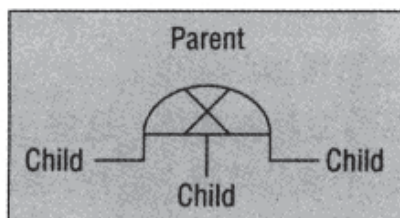


图 5-10

5.6.2 明确概念——实现子类别

关于子类别真正很酷的事情是,它们允许你把所有具有相似构造的东西存储在一个地方。在学习概念前,你可能使用过两种方法中的任意一种来实现我们的文档模型:

- 把所有的属性添加到一个列中,并且如果某个给定记录的信息不符合你感兴趣的特定的文档类型,只需让列保持空值即可;
- 对每一种类型的文档使用一个单独的表,每一个表中都会重复文档类型之间本质上相同的列(每一个表存储自己的一份文档信息,因为该信息要应用在那个特定的表中的记录上)。

不论文档的类型如何,使用子类别概念可以令你能够存储所有的文档,就好像它们都来自于同一个地方。现在,对于系统中所有文档的信息的任何查询,都只能运行在一个表上,而不必在 3 个(或 3 个以上,或 3 个以下)不同表上使用 UNION 运算符进行联结。于是下面这一点不言而喻:比起其他的选择,使用子类别实现的这种情形可以显著地增强执行性能。

不过,这里有一个问题(你已经知道这里会有问题,是吧?)——必须提供某种机制,以指向那个文档的其余信息。查询所有的文档可以提供你在寻找的特定文档的基本信息,但是,当你想要得到该文档的其余信息(该文档类型所特有的信息)时,如何让应用程序知道匹配的信息在哪个子类别表中?要实现这个任务,只需向超类别表中添加一个字段,说明该记录属于的哪种子类别。在我们的例子中,可能需要在文档表中实现一个称为 **DocumentType** 的不同的列。从该类型列中你可以知道,应该在其他的哪一个表中查找带有更多信息的匹配记录。此外,可能要使用一个域表(该表用于把 **DocumentType** 列中的值限制在有子类别的类型中)和表的一个外键来实现。

提示:

要谨记, 尽管这里讨论的是物理存储和数据检索。但是, 完全能够使用存储过程或一连串的视图(或者二者)进行抽象。例如, 可以让存储过程调用从 Document 表中取出信息然后联结到适当的子类别。

一些读者会说, “等等, 为什么我在其他关于 n 层架构的书中读到不要使用存储过程?” 好。以我不算太拙的拙见, 那是个糟糕的建议(第 10 章将更详细地讲述存储过程)。有可用的性能工具却不去使用, 那真是很傻——不过要记得只通过数据访问层去使用它们, 甚至不要让中间层或客户组件知道存储过程的存在。听从了这一建议, 将获得更好的性能、改善整体封装级别、缩短开发时间, 而且具备了上面的所有优点之后, 你依然符合单独的数据访问层这一理论要求, 这对 n 层设计十分重要。

除了要建立指向文档类型的指针, 还需要确定正在处理的是普通的子类别还是独占的子类别。在我们的文档例子中, 应该把它设计为独占的子类别。虽然, 你可能有许多文档, 但是, 不可能有既是租约又是离婚证明的文档(非独占子类别允许子类别之间进行任意混合)。即使一个租约带有购买选择权, 但还是要创建一个单独的买卖契约文档来行使租约选择权。

图 5-11 显示了逻辑模型的实现。

现在, 有一个名为文档的实体。这些文档属于特定的类型, 且类型被限定在一个域内——DocumentType 设定这个域的范围。另外, 每一种类型都由它特有的实体(或子类别)来描述。位于这些实体中间的符号(中间有“X”的半圆)表示这 3 个子类别本质上是独占子类别(对于文档的每一个实例, 有且仅有一个子类别)。

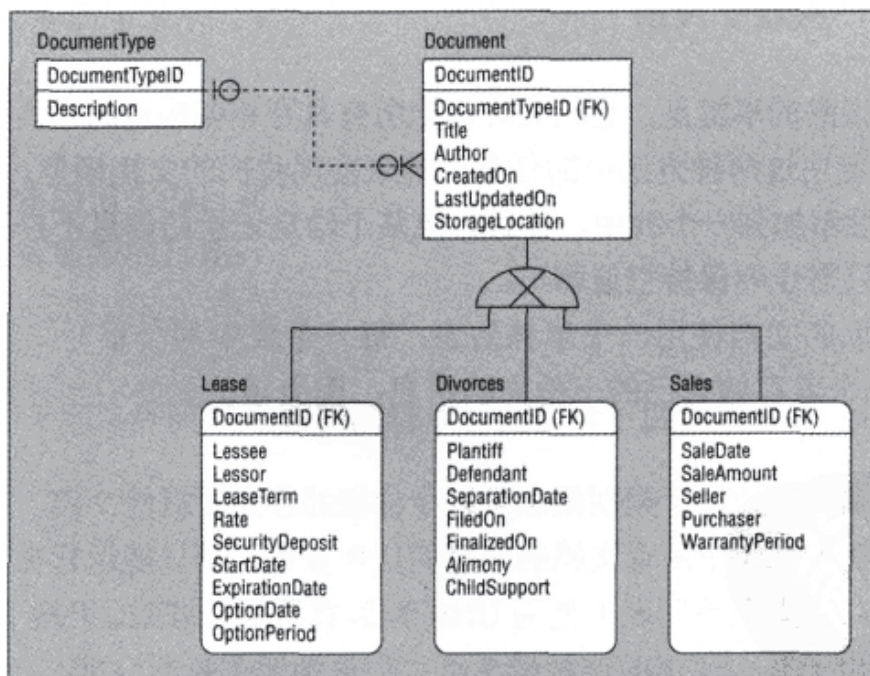


图 5-11

至此, 应当回过头仔细想一下逻辑模型能为你完成的操作。我在本章前面讲过, 逻辑模型为你提供了一种交流业务规则和数据需求的途径。因此, 只需稍加解释, 某人(或许是客户?)就能审视逻辑模型并理解你说明的这一概念: Leases、Divorces 和 Sales 都是同一个主题的变种——它们实际上是相同的事物。查看的人可能会说, “不, 等等, 它们其实是不同的事物。” 或者说, “哦, 我明白了——你看, 还有遗嘱和授权书——它们几乎是同样的东西, 不是吗?” 眼下这些少量的

信息能在以后为你省下大量的时间和金钱。

5.6.3 子类别的物理实现

从物理的角度考虑事物, 没有什么能像在逻辑模型中那样看起来整洁清爽。事实上, 在物理上做的所有事情都是实现一系列的一对零或一对一关系。不过, 你的确把它们作为单独的多表关系的一部分表达出来(图 5-12)。

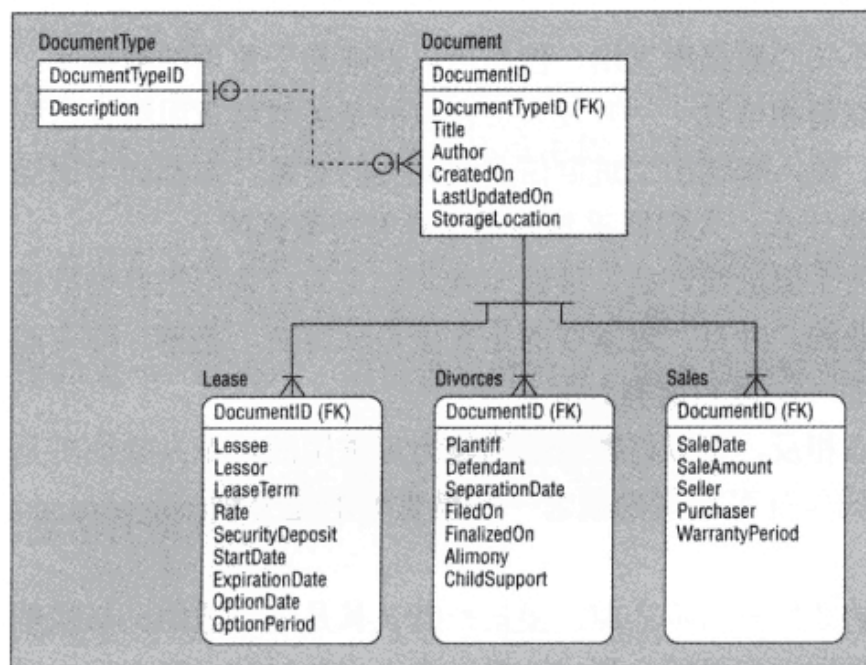


图 5-12

只有在具有独占的子类别时(实际上这是更经常出现的一种情形), 你才会被欺骗。在这种情况下, 还需要向子类别表中加入一些逻辑(用触发器的形式)以确保: 在插入任何行的时候, 在其他的任一子类别中不存在另一个匹配的行。例如, 可能需要在 Leases 中放置一个插入触发器, 该触发器在 Divorces 和 Sales 表中查询具有相同 DocumentID 的记录。如果发现已有一条这样的记录, 则触发器会拒绝插入记录, 发出适当的错误消息并回滚(ROLLBACK)。

5.6.4 通过子类别增加可扩展性

子类别这个概念会极大地影响到数据库设计的成功。如果运用得当, 能够显著减少查询的时间, 并极大简化为相关但不同的信息而聚集的信息。不过这些还不是子类别仅有的好处。

子类别能够提供一种方法让数据库更具可扩展性。在加入另一个子类别时, 唯一要处理的查询是那些针对新子类别的查询。只与父表有关的查询仍将正常工作——而且, 它们无需任何改变就能获得新子类别上的信息!

总之, 在可扩展性上获得的两个主要的好处是:

- 只扫描一个表就能获得适用于超类别(在本例中是文档)的信息, 而不必使用 UNION 运算符。这意味着更少的联结和相对更快的查询执行速度——尤其是当表变得更大或者有越来越多的子类别时。
- 添加新的子类别花费的开发时间通常不会像从零开始开发类别的框架那样长。

像大多数事情一样, 你必须紧记子类别的一个弊端——子类别会在父表上造成瓶颈。对整个集合中涉及到所有表和数据进行查询可能要访问父表。想一下这里涉及的锁(若你初次接触有关锁

的事项,第12章会详细讨论它们)。如果在索引和查询策略上不小心,可能导致一些很糟糕的阻塞和(或)死锁问题。话虽如此,只要小心地计划和编写查询,通常不会出现这方面的问题。另外,如果父表的大小成了一大难题,现在 SQL Server 给了我们一个使用分区表的选择,以便能处理更大的大小。

5.7 数据库重用

人们之前几乎从不关心数据库重用,但是可以创建易于重用的数据库。为什么我会说几乎没有人关心?关于这一点请相信我——开发人员在意的类似可重用组件这样的东西。诸如用来验证信用卡、分发邮件、输入和输出二进制信息流此类的对象,你都会考虑储藏将其起来以便反复使用。但是,不知道为什么,人们似乎却并不这样看待数据库。

出现这种情况的一个原因或许是,数据库就其定义而言是用来存储数据的。通常,认为数据是属于一个公司或企业的,并且,最重要的是数据是私有的。我猜,你可能自然而然地认为存储数据的容器也是私人的——谁知道啊?

与人们普遍的观点相反,可以将数据库构建为可重用的。令人吃惊的是,要实现数据库重用必须应用许多使得代码组件可重用的概念——最重要的是分割(compartmentalization)和使用通用的接口。

在试图重用现有的数据库结构之前,要记得确认其是很合适的。与我曾见过很多在编程中重用的事情十分相似,重用很有可能会变成这样:你工作的时候一直在使用错误的工具,实际上,事情可能变得更加昂贵,因而,反倒认为从零开始编写会更好些。

5.7.1 可重用数据库的候选

最有可能成为可重用数据库的是那些能被分解为单独的主题域的数据库(类似于组件常常被分解为功能组),每一个主题域保持通用的可行性。例如一个帐目清算数据库可能有与帐目清算中的功能域匹配的主题域:

- 购买;
- 应收款(可能会依次分解为发票和现金收据);
- 存货清单;
- 应付款;
- 总分类帐;
- 现金管理。

上面的列表还可以继续。另外,也可以把分解进行到更细的粒度,创建很多很多的数据库,细到人员、商业实体(你是否注意到顾客和售卖者之间有多相似?)、订单——有很多事物都具有不断重复使用的基本构造。你可以把这些累积为它们自己的“微型数据库”,然后将它们添加到更大的逻辑模型中(用存储过程、视图或数据访问层的其他组件联系在一起)。

5.7.2 如何分解事物

在这里,逻辑建模和物理建模才开始展示自己真正的实力。在努力使数据库可重用的时候,

常常会有一个包含许多物理数据库的逻辑数据库(包含所有不同的主题域)。有时候你会选择直接引用每一个物理实现来实现逻辑设计。另一些时候, 可能要选用能够更好地隐藏数据库实现方式的方法——可以创建“虚拟的”数据库, 因为该数据库中只有视图而没有别的东西, 视图从适当的物理数据库中引用数据。

我们先脱离主题, 说明一下这个过程在本质上类似于面向对象编程中的封装。通过使用视图, 能够对视图的使用者隐藏数据库的真正实现。这意味着可以在数据库中移除一个主题域, 并用完全不同的设计来替换它(唯一的关键是要把新的设计映射到视图上), 从此以后, 客户端应用程序和用户就不会注意到实现中的变化。

分成几个独立物理数据库和/或虚拟化数据库会造成一定的限制, 但其中的许多限制都可以帮助你将一个主题区域从整体中分割出来并在其他环境中重新使用它。

分解事物包括:

- 尽量减少或者消除对其他功能域的直接引用。如果已经实现了视图方法, 就可以只通过视图把每一个物理上分离的数据库部分连接成逻辑整体。
- 不要使用外键约束——必要时, 可以改用触发器。触发器能够跨越数据库, 而外键约束却不能。

5.7.3 可重用性的高昂代价

所有的这些重用都是要付出代价的。为了便于重用, 你为自己设计的许多调整都会给系统性能带来负面影响。它们包括:

- 总的说来, 外键约束比触发器快, 不过, 触发器是跨越数据库边界实施引用完整性的唯一方法;
- 使用视图意味着在所有的查询上进行两级优化(一次是到达底层的查询并与原始的查询紧密协调, 另一次是找到给出最终结果的最佳方案), 这样做开销更大, 而且会减慢速度;
- 如果不使用虚拟数据库方法(一个数据库中的视图被映射到所有其他数据库中), 则跨越多个数据库维护用户的权限可能会困难重重。

总之, 别指望速度能像在单一数据库模型下那样快, 除非把数据分散到更多的服务器上处理。

重用数据库在缩减开发时间和费用上极具意义, 但是, 你必须平衡这样做的优势和可能在性能方面招致的某种程度的损害。

5.8 反规范化

由于这里的讲述很容易陷入到高级的概念中, 因此我将尽量简短地说明, 不过要记住别沉迷在数据的规范化中。

正如本章前面讲过的, 有时候数据库设计人员会把一些东西当作一种象征, 规范化就是其中之一。不知何故, 规范化成为了他们的信仰, 他们开始为了规范化而规范化数据, 而不是为了通过规范化做有利于数据库的事情。下面是在这方面要考虑的一些事情:

- 如果声明计算列或存储一些派生的数据能够使报表的运行更加有效, 那么当然可以加入它。只是要记得把好处和风险都考虑进来。(例如, 如果“摘要”数据与派生它的数据不同步会怎么样? 你如何确定出现了不同步, 以及一旦出现不同步你将如何修正?)

- 有时候,只要在表中包含进一个(或多个)非规范化的列,就能够消除或者显著减少检索信息必须进行的联结的数目。要注意这些情形——实际上它们发生得相当频繁。我曾经遇到过这样的情况,在经常用到的一个基表中加入一列,就能够把 9 个表联结减至 3 个,并在处理中将查询时间减少了大约 90%。
- 如果存储的是历史数据(数据很大程度上不会有变化,而且只是用于报表),那么完整性问题将会变成一个较小的考虑因素。一旦把数据写入到只读区域并进行了校验,就可以确定不会遇到这种“不同步”的问题,这类问题是数据规范化要解决的主要问题之一。在那种情况下,或许在一些表中“平展”(反规范化)数据会更好些,并且可以提高速度。
- 必须进行联结的表越少,要自己做报表的用户就会越高兴。这些用户对于他们使用的工具越来越得心应手。渐渐地,用户会找到他们的数据库管理员,并要求直接访问数据库,以便能够完成他们自己定制的报表。对于这些用户来说,高度规范化的数据库会令他们很迷惑,且对他们几乎没什么用处。非规范化数据能够让这些用户觉得更轻松。

话虽如此,当有什么疑问时,还是要规范化数据。一般以这种方式设计关系系统是有原因的。在规范化上犯错是在更好的数据完整性上犯错,也是在事务环境中更好的性能上犯错。

5.9 通过分区方法进行扩展

从 SQL Server 2000 开始,SQL Server 就被赋予了非凡的能力,可以由多个物理表(分区视图)创建一个逻辑表。即,对来自一个逻辑表的数据进行分区,把数据存储在一组定义明确的一单独的物理表中。不过,数据分区概念比分区视图出现得要早得多。实际上,在一台服务器上放置主帐目系统而在另一台服务器上放置订单输入及库存系统,这就是某种形式的分区——这样做是在确保处理这两项活动的负载分散在多台服务器上。SQL Server 2005 则更进一步,添加了分区表(partitioned table)。

分区表与分区视图的名称暗示它们在某种程度上是不同的——尽管它们确实都是表。分区视图不能支持某些表的功能(约束、默认值、标识列等等),而分区表则支持所有的这些功能。

当然,麻烦总是会有——分区表被限制在仅仅一台服务器上(这是把表分散在多个文件组、进而分散在多个驱动器上的一种方法)。注意,服务器的限制并不表明只局限于一个物理存储设备——没有什么可以阻止你将多个存储设备(包括多个 SAN)链接到一个服务器上。

注意:

分区表不允许那些不属于分区键的列使用唯一索引——当你要分区的列不是你要用作主键的列时,或者需要对其他的列施加唯一约束的时候,这是至关重要的。

必须在多个服务器上分担负载时,仍然需要使用分区视图。鉴于本章的目的,这里只讲述既适合视图又适合表模型的分区的基本概念。

无论使用哪一种分区方法,分区的理念都是相同的。用逻辑表中的一个或多个列作为划分的依据,在物理上分散数据。这使得你可以用多个输入输出通道甚至多个服务器处理查询。只是如何划分数据是个很大的问题。使用极其简单的方法,仅仅根据分区列上可能的值来均等地分割数据将是大势所趋。这个方法或许不错,但是也有些弊端,两个原因如下:

- 数据很少能分成刚好均等分布的堆,通常需要大量的研究以及预先取样来预测分布状态;

- 一旦存储了数据, 就不考虑到实际使用数据的方式。

数据分区的方式不仅是决定每一个分区将接收数据的数量——更重要的是, 它在整个系统执行得有多好的问题上会造成非常巨大的差异。要记住:

- 表很少是孤立的。多数时候, 需要把给定表中的数据与系统中的其他数据进行联结——“其他”数据分区的方式兼容吗(从性能的观点看)?
- 网络带宽通常是系统性能最大的瓶颈——在设计分区时将如何考虑这一点呢? 处理分区表方案(将只在一台服务器上)时, 这不算大问题, 但是, 对于分区视图模型来说, 这是非常重大的问题。

那么, 记住了这些后, 下面列出分区的一些规则:

- 使用分区视图在服务器间分散数据时, 把将要一起使用的数据存储在一起。即, 如果会频繁地在查询中一起使用某些表, 那么对这些表进行分区时, 要尽量将那些可能作为部分查询结果返回的数据放在同一台服务器上。很明显, 这种情况不可能 100% 保证实现, 但是, 经过仔细思考并确定将如何使用数据后, 会发现可以令大多数查询只发生在一台服务器上。例如, 对于一个给定的订单, 所有相关的订单详情请行应该位于同一台服务器上。
- 在设计应用程序的时候, 理论上应当让它意识到存在分区——也就是说, 应当让执行查询的程序了解它们的数据最可能在哪一台服务器上。数据可能散布在多个计算机上以供使用——如果应用程序一开始查询数据库服务器恰好就是正确的服务器, 那么就无需把查询转送到另外的服务器上了, 这难道不好吗?

如果确定需要使用分区系统, 那么, 一定要真正有一个打算处理的负载。如何对数据进行分区, 将对服务器能否成功处理负载造成重大的影响。要记住花点时间充分策划好分区计划。当认为已经决定好要做什么时, 请测试! 测试! 再测试!

5.10 SQL Server 关系图工具

可以通过导航到要为其构建关系图的数据库的关系图节点来打开 SQL Server 内置的工具(先展开自己的服务器, 随后展开数据库)。你会看到一些自己很熟悉的东西——一些与第4章中创建表时的对话框相同的对话框。SQL Server 关系图工具不会提供太多的选择, 因此你会发现自己很快就可以了解它。

可以从创建第一个关系图开始入手。右击 AdventureWorks 数据库下面的“数据库关系图”节点, 然后选择“新建数据库关系图”选项, 这样就能够创建新的关系图了。

提示:

你可能(如果这是你第一次创建关系图)会看到一个对话框, 警告自己这个数据库缺少一个或多个使用数据库关系图所需的支持对象, 并询问你是否要创建它们——请选择“是”。

SQL Server 会弹出一个“添加表”对话框(如图 5-13 所示), 这个对话框会列出可以添加到关系图中的所有表。

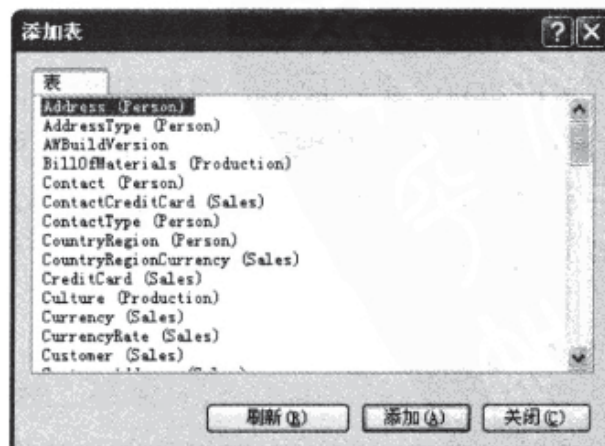


图 5-13

选择下面的表(记住按下 Ctrl 键以选择多个表):

- Address
- Customer
- CustomerAddress
- SalesOrderHeader
- SalesOrderDetail

然后,单击“添加”。SQL Server 在很快绘制选中的所有表后,单击“关闭”按钮。这时,SQL Server 已经把表添加到关系图中,如图 5-14 所示。

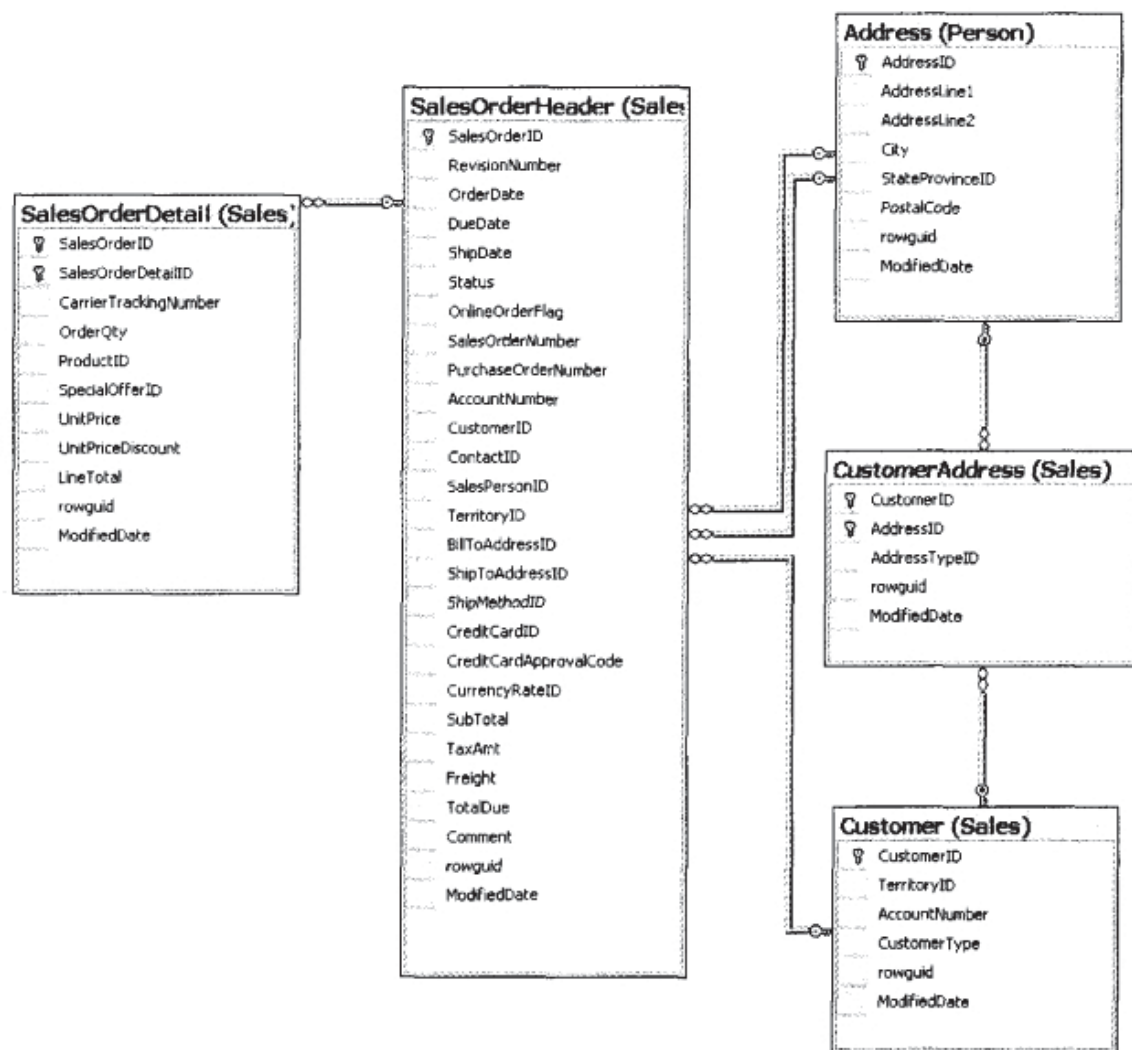


图 5-14

这里,我重新调整了 SQL Server 给出的默认图,以便适合本书的尺寸。如果你的屏幕采用不同的分辨率,由于缩放的原因可能很难一下看到整个关系图。要在视图中看到更多的表,请在工具栏中改变缩放设置。

SQL Server 会列出你要添加的每一个表,并分析其他哪些对象与这些表关联。除了表本身,你还将看到不同的其他项,它们是连接到表中的许多其他对象中的一部分——主键和外键。

那么,有了这样的开始后,我们可以把这个关系图看作一个起点,以此来阐释关系图工具的工作方式并在不同的地方创建一些表。

5.10.1 表

在关系图中,每一个表都有它自己的窗口,可以四处移动这个窗口。如果列是主键,那么在

列名称的左边显示一个小的钥匙符号, 就像 CustomerID 旁边显示的那样。这只是表的默认视图, 可以从若干其他选项中进行选择, 这些选项允许你对表的特定组成部分进行编辑。可以在感兴趣的表上右键单击就可以了解表视图的选项。默认的表现图是只显示列名, 但是, 你也可能对“自定义”的选项有兴趣。使用该选项或“标准”选项就可以直接在关系图中编辑表(这真不错!)

1. 添加表

可以按照下面的两种方式向关系图中添加新表:

- 如果要添加的表已经存在于数据库中(但不在关系图里), 不过现在想要把它加入到关系图里, 只需在数据库关系图工具栏里单击“添加表”按钮, 或右击关系图的任何地方并选择“添加表”选项。随后会列出数据库中所有的表——选择想要添加的表即可, 关系图中会显示这个表以及它与其他表之间的所有关系。
- 如果要添加全新的表, 单击数据库关系图工具栏里的“新建表”按钮, 或右击关系图并选择“新建表”, 随后会询问你新表的名称, 然后会在关系图中加入这个表, 该表将以“列属性”视图的方式显示。编辑属性, 使列具有列名、数据类型等希望的东西, 这样数据库中就有了一个新表。

注意:

下面花一点时间指出这个过程中的一些难点。

首先不要忘了向自己的表添加一个主键。SQL Server 不会自动完成这些操作, 也不会提示你这么。这不是一个靠直觉完成的过程。为了加入一个主键, 你必须选择希望在键中保留的那些列。随后右击并选中“设置主键”按钮。

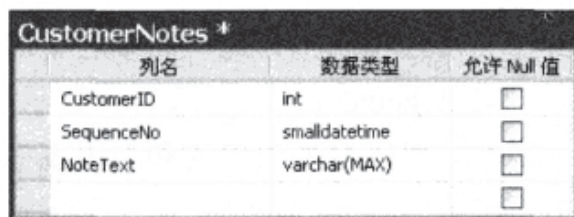
接下来, 意识到除非选择保存, 否则自己的新表实际上不会被加入数据库——这同样适用于你一直以来所做的所有编辑操作。

接下来快速添加一个表, 以了解它的工作方式并设置随后的例子。

首先, 右击关系图窗格中的任何地方和键, 然后选择“新建表”。接着会提示你输入表名——这里把表命名为 CustomerNotes。现在只加入三列, 如图 5-15 所示。

注意, 在该表的标题栏中有一个星号——这说明在该表上有未保存的变动(明确地讲, 该表根本就没有保存过)。接着保存关系图, 这也会在物理数据库中创建表。

现在, 有了一个包含三个 NOT NULL 列的表。截至目前为止该表还没有主键(我们将在添加约束的小节中处理主键)。



列名	数据类型	允许 Null 值
CustomerID	int	<input type="checkbox"/>
SequenceNo	smalldatetime	<input type="checkbox"/>
NoteText	varchar(MAX)	<input type="checkbox"/>

图 5-15

2. 从数据库或关系图中删除表

删除表有一点让人迷惑, 因为从关系图中删除表和从数据库中删除表略有不同。从关系图中删除表, 有以下两种方式:

- 选中表, 然后按下删除键;
- 选中表, 然后从工具栏中选择“从关系图中移除”按钮。

要彻底从数据库中删除表, 有以下三种方式:

- 选中表, 然后选择“编辑”|“从数据库中删除表”菜单选项;

- 选中表，然后单击工具栏上的“从数据库中删除表”按钮；
- 在表的标题上右键单击，选择“从数据库中删除表”菜单选项。

注意：

虽然只有对关系图进行保存，才能将表从关系图中永久删除，不过在确认删除之后，就会立即将它从数据库中删除。

5.10.2 处理约束

如果必须使用关系图工具，你就必须做更多的工作而不是只创建基本的表——你可能还要确定约束。不过有了关系图工具，这个过程会变得比较简单。

1. 主键

这一任务实在是非常简单。要创建主键，只需选中想要参与到主键中的列(再说一次，如果要选中多个列，必须按住 Ctrl 键)，右键单击并选择“设置主键”，如图 5-16 所示。

我向在前一节中创建的 CustomerNotes 表添加一个主键。在选择“设置主键”的选项时，你会看到你列中的所有字段都加了一个键图标。为了改变主键，只需要选择一组新的列然后再次选择“设置主键”选项。要删除主键，只需要从同样的菜单中选择“删除主键”项(我们的图中并没有显示它，因为还没有设置主键)。



图 5-16

2. 外键

外键和主键一样容易使用——它们使用简单的拖放模型。

在我们的 CustomerNotes 例子中，你会注意到我使用了 CustomerID——这个和 AdventureWorks 数据库在其他地方使用的 CustomerID 一样，所以你要为 CustomerID 的基础表(Customer)创建一个外键是非常合理的。为此，简单地单击 Customer 表的 CustomerID 列，将其拖到 CustomerNotes 表上。Management Studio 随后会显示图 5-17 中的对话框确认你要添加的外键。

现在，可以对被引用的和引用表中的列进行修改，如果需要的话，甚至可以加入额外的列。单击“确定”，就会看到图 5-18 所示的对话框，它允许你设置外键定义的其他属性，包括级联操作和是否需要将这个外键传递给其他的复制数据库。

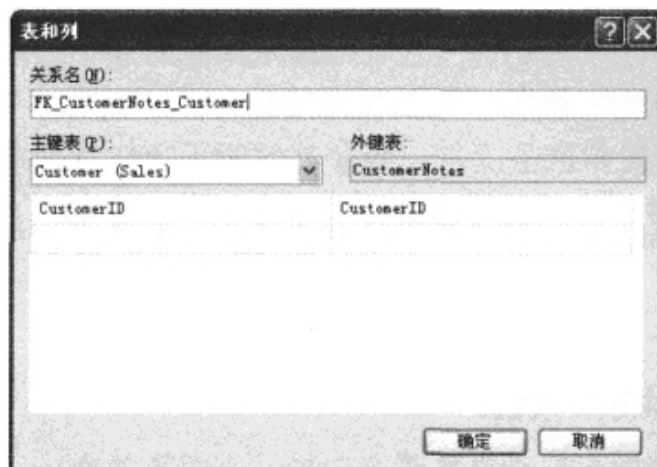


图 5-17

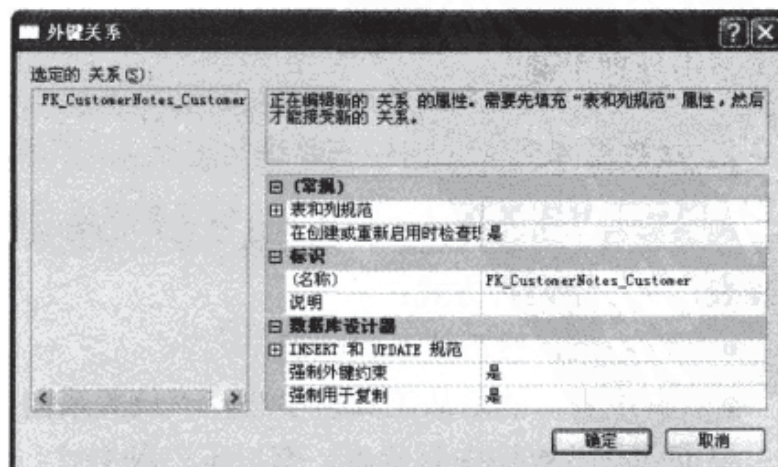


图 5-18

在创建了外键之后, 若要编辑它, 需先选中它(通过单击外键), 随后会在屏幕右边的窗格中看到外键的属性。

提示:

属性窗格是可移动的窗口, 因此, 有可能已经把它从默认屏幕右边移开了。

要删除外键, 只需右键单击关系, 并选择“从数据库中删除关系”。

3. CHECK 约束

只需右键单击该表就可以为表定义 CHECK 约束, 然后选择“Check 约束”。这样会打开一个对话框, 允许你创建新的约束, 或者从已经为表定义的约束中进行选择。创建了一个新的约束或者从现有约束中选定了—一个后, Management Studio 会打开一个对话框, 该对话框与用于外键约束的对话框并没有太大的不同。

在创建表的时候, 在 CK_CustomerNotes 名称的旁边有一个星号——告诉你这里有没有未保存的更改。在该对话框中, 首先要注意的是“表达式”字段, 也就是要输入约束条件的地方。

提示:

不要把对话框中的“标识”框与 IDENTITY 列混为一谈——对话框中的这一部分只是用来提供名称和对约束的描述(可选)。

只需按照自己的需要改变属性就可以编辑现有的约束。要删除 CHECK 约束, 只需选中感兴趣的约束并单击“删除”按钮。

5.11 关于日期列

一般情况下, 我们不会在特定数据类型上花费太多的时间, 不过使用 SQL Server 2008 的时候必须特别留意此类新的数据类型。特别的要注意的问题是新的 Date 和 Time 数据类型修改事物的方式。这里我们暂时不讲性能一章(第 21 章)中讨论的设计性能和空间细节, 不过需要简短地讨论一下新的 Date 数据类型。

之前版本的 SQL Server 只支持日期和时间, 并将它们看做一种组合数据类型。Datetime 数据类型占据了整个 8 比特, 而且这种组合往往会在开发过程中导致冲突——其中包括:

- 不需要跟踪特定时间的时候就会浪费空间(或者你需要的就是时间)。
- 比较带有时间的日期很麻烦(你想看一看它们是否在同一天, 不过由于它们在同一天的发生时间不同所以两者不可能完全相同, 你可以绕开这个问题, 不过它很麻烦而且会弄乱你的代码)。
- 在处理只希望返回日期或时间的客户数据类型的时候, 偶尔会遇到这种兼容困难。

为了解决这个问题, 新的日期和时间数据类型将日期和时间分开并为每个类型加入一定的灵活性(你可以设置精度)。你现在可以轻松地比较日期, 不但可以轻松地比较不同的时间, 而且还可以设置精度以节省空间或将时间的分辨率设置为纳秒级(之前的精度只能达到 3ms)。

此外, 我们有了新的数据类型以便满足日益增长的标准化时间需求。允许记录相对于协调世界时(UTC, 法语 Temps Universel Coordonné 的简称)的时间偏移。这表明你可以接受世界上任何

地方提交的时间并比较它们以便更加好地比较时间。

本书会继续深入讲解这些新数据类型，不过为了保持 `datetime` 数据类型的一贯性，必须认识这些新的数据类型并计划它们对自己应用程序的影响。

5.12 小结

数据库设计是一个庞大的概念，有很多优秀的图书将其作为唯一的主题专门讲授。根本不可能只用一两章就理解所有的数据库设计理念。

尽管如此，本章开了个好头。你已经了解到一些规范化的知识，而且也看到了规范化并不总是正确的答案——有策略地对数据进行反规范化能够简化数据库，并加快报表的执行。最后，还讨论了数据库设计中一些与规范化无关的概念，并讲述了如何利用关系图工具来设计数据库。

接下来的一章将详细讲述 SQL Server 存储信息的方式，以及使用索引的最佳方式。

第 6 章

核心存储和索引结构

索引可能是数据库规划和系统维护中第二个至关重要的部分(对表来说)。但是为什么在许多设计中都常常在事后才考虑索引呢?

思考一下这个问题。许多数据库系统都是基于快速、有效获取和维护数据的。索引为数据库系统提供了额外的方式查找数据并快速找到数据的物理位置。正确的索引可以节省大量的查询执行时间。所以,如果高效地获取和维护数据是我们建立数据库的原因,那么索引就是数据库中高效访问和维护数据的关键。为什么许多软件工程师确定好表的布局之后就直接进入存储过程或客户机代码呢?真是愚蠢。

这里不要误解我的意思:考虑存储过程、客户端代码和其他非表元素是非常重要的,而且许多开发人员不会设计出没有索引的数据库。实际上,不用你指定,数据库中也会至少出现一些索引(创建一个主键或者唯一的约束条件会创建一个隐含的索引来强制执行那些约束)。但是在几分钟的猜测中或者只是为了试图消除 QA 中出现的某个性能故障而应用索引(甚至是一个已经发布的产品的修补程序)的情况屡见不鲜。在其他的场合下,开发人员会采用一种“索引一切”的方法,而没有意识到这样做需要额外的存储,或者那么多规划不佳的索引实际上会增加查询的运行时间。

本章将从开发人员和管理人员的角度出发,重点介绍 SQL Server 中的核心索引结构。这里也会介绍 SQL Server 中数据的存储方式以便于更好地理解 SQL Server 进行优化选择的方式,以及不同索引的使用场合。

6.1 SQL Server 存储

提示:

存储在 SQL Server 2008 中做了一些细微的改动(技术上,它们曾经在 SQL Server 2005 中以服务包的形式出现过)。这些变动主要集中在定长存储类型的压缩上,这部分内容我们会在下一章讨论。

可以把 SQL Server 中的数据视为存储在某种层次结构中的数据。这种层次非常简单。层次中的一些对象是你将要直接处理的,因此很容易了解。许多其他的对象存在于保护之下,虽然在一些情况下可以直接处理它们,但是通常是不可以这么做的。下面来逐一讲解。

6.1.1 数据库

数据库是比较容易理解的概念。也许有读者会说“我已经知道了”。是的，大家很可能已经知道。不过由于它是存储定义的最高级别(对于一个指定的服务器而言)，因而这里作为一个独立的实体指出它。尽管无法显式创建数据库级别的锁，但是数据库是可以建立锁的最高级别。

提示：

锁是系统使用的某种控制和位置标记。第 11 章将全面讲述锁，不过在我们讲述存储的时候，会顺便讨论 SQL Server 中对象的可锁定性。

6.1.2 文件

在默认情况下，与数据库有关的文件有两种：

- 第一种是主物理数据库文件——它是最终存储数据的地方。应当以*.mdf 为扩展名命名这种文件(推荐使用这种方法，不过并非必须如此——但是你会发现以其他方式命名会随着时间的推移而变得让人迷惑)。可以添加“次”文件(而且应该使用*.ndf 扩展名)，而且不需要与主文件位于同一主物理驱动器上(这意味着你可以使用它们分配 I/O 负载——第 21 章会深入讨论这个问题)。
- 第二种是数据库文件的某种分支——日志。在第 11 章中讲述事务和锁时，会深入研究日志，不过你应该意识到它驻留在自己的文件中(应该以扩展名*.ldf 结尾)，而且没有它，你的数据库不能工作。日志是从最近一次向“数据库”提交数据之后数据库中发生事件的顺序记录。数据库实际上不是一组完整的数据。日志也不是完整的数据集。只有从数据库入手，并“应用”(从中添加所有的活动以同步)日志，才能拥有完整的数据集。

对于这些文件相互之间如何放置没什么限制。可以把每一个文件放在单独的物理设备上(实际上，这样做甚至是非常必要的)。这不仅允许一个文件中的活动不会影响到另一个文件中的活动，而且还创造出这样的情形：当数据库文件受损时，不会损失所做的工作——可以恢复备份然后重新应用日志(它安全地存放在另一个驱动器上)。同样地，如果损失的是存放日志的驱动器，仍然能获得到最近的检查点时间的有效的数据库(第 11 章会全面介绍检查点)。

6.1.3 区段

区段(extent)是用来为表和给定文件中的索引分配空间的基本存储单元。它由 8 个连续的 64KB 数据页组成。

基于区段而不是实际使用的空间来分配空间的概念，对习惯于操作系统存储原则的人来说，多少有点难以理解。关于区段的要点包括：

- 一旦一个区段已满，下一条记录将占用一个完整的新区段大小，而不是记录本身的大小。许多新接触 SQL Server 的人在空间估算上失误，部分原因是由于一次分配的是一个区段而非一条记录的大小。
- 通过预先分配空间，SQL Server 省下了为每一条记录分配新空间的时间。

仅仅因为要添加的行比现在分配的区段所能容纳的行多了一行，就要另外占用整个区段，这似乎是一种浪费。但是，以这种方式浪费的空间总量通常并不那么多，只占整个数据库空间的很

小的百分比。不过它们会积少成多(特别是在高度碎片化的环境中),因此你一定要紧记。

占用整个区段空间的好处是 SQL Server 省去了一些分配空间的时间上的开销。SQL Server 不必每写入一行就考虑分配问题,而是在需要新的区时才处理额外空间的分配。

提示:

不要把区段所占用的空间与数据库占用的空间混为一谈。分配给数据库的空间是从硬盘可用空间中消失的空间。区段的空间则仅仅是在单独数据库文件获得的整个空间中分配的空间。

6.1.4 页

与区段是数据库中的分配单元类似,页是特定区段中的分配单元。每一个区段有 8 个页。

页是到达真正的数据行的最后一个级别。尽管每个区段中页的数目是固定的,但是页中的行数是不固定的——这完全取决于行的大小,这是可变的。可以把页想成是表和索引行数据的某种容器。行不允许跨页。

图 6-1 展示了数据是如何放置在页中的。对于插入的每一行,为了表明特定行的数据开始于页中的何处,注意我们把行偏移量放置在页末尾。

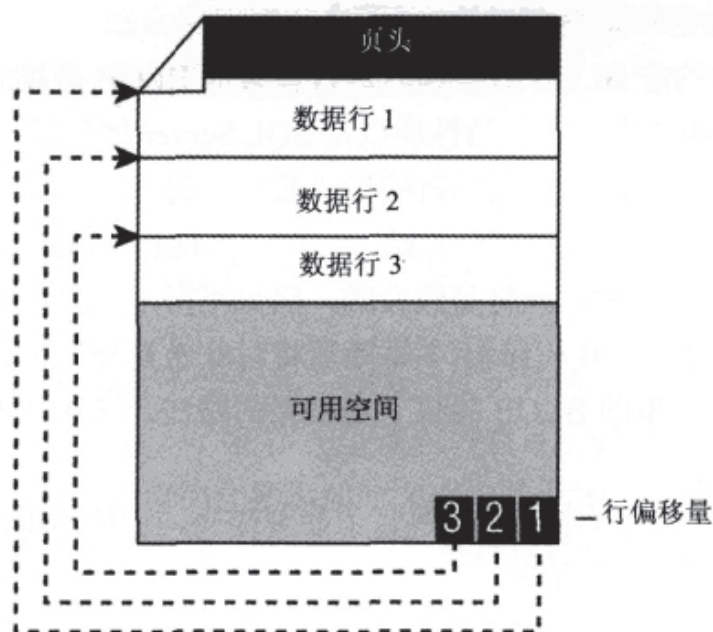


图 6-1

有很多种不同的页类型。鉴于本书的目的,这里关心的类型是:

- 数据;
- 索引;
- 二进制大型对象(BLOB)(用于图像,大多数的 Text、Ntext 数据和大于 8K 的 varchar(max)/nvarchar(max)数据);
- 全局分配映射表(GAM)和共享全局分配映射表(SGAM);
- 页可用空间(PFS);
- 索引分配映射(IAM);
- 批量更改映射;
- 差异更改映射。

1. 数据页

数据页是不言自明的——它们是表中的实际数据，除了所有没有存储在行内的 BLOB 数据之外(在“BLOB 页”一节中有更详细的描述)。当行中有一个列包含 BLOB 数据时，常规的数据存储在数据页中，而且 BLOB 数据可能被存储在一页中(如果足够小可以放下)。如果页放不下 BLOB 数据，那么使用一个 16 字节的指针显示包含 BLOB 起点的 BLOB 页。

2. 索引页

索引页也非常简单明了：它们保存非聚集索引的非叶级页和叶级页(在本章的后面会考查它们是什么)以及聚集索引的非叶级页。当继续本章的学习时，这些索引类型会变得越来越清晰。

3. BLOB 页

BLOB 页用于存储二进制大对象。对于 SQL Server 来说，它们是所有存储在 `varbinary(max)`、`varchar(max)` 或 `nvarchar(max)` 列中的数据。在数据存储页中 BLOB 页是特殊的，因为它们并没有那么多行。既然 BLOB 的大小可以达到 2GB，因而它们必须能够分布于多页——就这部分，版本并不重要。为了存储整个 BLOB，SQL Server 将分配它所需的足够多的页，但不保证这些页是连续的——页可以位于数据库文件中的任何地方。

正如前面说过的，在行中的非 BLOB 数据和与该行有关的 BLOB 数据之间，以指针的形式联系在一起。在 SQL Server 7.0 版本中，指针的性质以及 SQL Server 如何导航到 BLOB 数据的方式发生了改变。在 6.5 版本及以前，BLOB 页以链的形式放在一起——类似于链表。要找到任何是 BLOB 的一部分的页，必须从链表的起始部分开始，一页一页地在 BLOB 中浏览。如果试图进行某种文本或二进制形式的搜索，这种排列将是致命的，因为不得不陷入数据的串行扫描中。不过从 7.0 版开始有了改变，页被组织为 B 树结构(本章随后将对此做充分的讨论)。B 树提供了一种有更多分支的结构，因此，为更大型的 BLOB 提供了更直接的路线。这在文本操作可以执行得多快上产生了很大的不同。

即使 BLOB 的执行性能在几个版本的发展中有了显著提高，它依然很慢，因此，当后面讲述高级的设计问题时，会讨论其他存储方法。

4. 全局分配映射表、共享全局分配映射表和页可用空间页

全局分配映射表(GAM)、共享全局分配映射表(SGAM)和页可用空间(PFS)页类型涉及确定哪些区段和页在使用，哪些没在使用。从本质上讲，这些页存储的是说明哪些空间可用的记录。其实，完成高质量的开发或系统管理并非一定要理解这些页的类型，而且这些内容已经超越了本书的讲述范围。尽管如此，如果你非常想了解它们，可以在联机从书中找到更多关于它们的信息(或者你受到失眠的困扰)——在索引中查找 GAM 即可。

5. 大容量更改映射表

由于现在还没有讨论过大容量操作，我们该怎么讲述这一主题呢。

SQL Server 中有“大容量操作”的概念。大容量操作是对数据库非常快速的更改(通常是大规模的数据导入或截断表)。这样的速度部分来自于大容量操作不“记录”它们所做的每一件事情。日志对于备份和恢复系统是至关重要的，而大容量操作意味着数据库中发生了没有记入日志的活

动(它会记录进行了某些操作,但不记录操作的详情,因此日志无法重建所做的事情)。

大容量更改映射表(BCM)是一组用于跟踪哪些区段通过大容量操作进行了修改的页。它对于修改的细节不感兴趣——只关心你对特定的区段作了改动。因为知道该区段发生了改变,在备份数据库时,它能提供更多的选择。更确切地讲,在进行日志备份时,可以对受大容量操作影响的区段上的物理数据进行了备份,以此作为对日志备份的补充。

6. 差异更改映射表

它与大容量更改映射表几乎一样,不过,它不是只跟踪被大容量操作更改的区段,它跟踪从最后一次完整备份数据库以来发生了更改的所有区段。

请求进行差异备份时,差异更改映射表(DCM)提供了需要备份区段的信息。由于只包含自上一次备份以来发生过改动的区段,所以该备份更小且运行速度更快(虽然只是部分的)。

7. 页拆分

页满时,会对其做拆分。这意味着不仅要分配一个新页,还要把现有页上大约一半的数据移动到新页上。

关于这一过程的例外情况是使用聚集索引的时候。如果存在聚集索引,而且下一个插入的行物理上位于表的最后一行,那么将创建一个新页,并把新行添加到新的页上,而不会重新部署现有的数据。在研究索引时,会更深入地讨论页拆分。

6.1.5 行

鉴于你将听到大量关于“行级锁”的说法,因此,对于行这一术语应该不会太意外。通常,行的大小可以达到 8KB。

除了行的大小限制为 8060 个字符外,行中的最大列数限制为 1024 列。在实际处理中,在列的大小达到 8060 字符的限制之前,很少会出现列的数目不够用的情况。当是 1024 列的时候,平均列的大小为 8 字节。多数应用中会超出这一大小。关于这一点的例外是在测量和统计信息中——要存储大量不同的数字样本。尽管如此,即便是在这样的应用中,达到 1024 的限制也是很罕见的。

你可能已经注意到,在提及 8KB 的限制时,我使用了“通常”一词。这个限制的依据是一行限制在一页中,且页的大小为 8KB。但是,在个别情况下可以超出这个限制——具体地说,是在使用 `varchar(max)` 或 `varbinary(max)` 以及传统的 BLOB 数据类型如 `image` 和 `text` 时。如果行在这些数据类型中的一种中包含的数据过多,无法放置在一页里,那么,这些特殊的数据类型知道如何让数据跨越多个页(一行可以达到 2GB)。这种情况下,原始的行用来跟踪该列中实际的数据存放在何处(所有其他的列依然存放在原始的行中)。

6.1.6 全文目录

在 SQL Server 2008 之前,在标准的数据库之外就有独立的存储机制。虽然可以把全文目录默认关联到指定的数据库上,并且甚至可以与数据库一起备份全文目录(在 2005 版中——即使分离了备份),它们完全是独立存储的。在 SQL Server 2008 中,全文目录与存储单元无关——而是全

文索引的逻辑分组。这里提到它们只是做一下历史参考(第 18 章将进行详细讨论)。

注意:

在 SQL server 2005 之前,不存在全文索引文件和核心数据库之间的协调备份。如果要解决与之前版本的后向兼容性问题的话,请记住这一点。

6.1.7 文件流

文件流是用来解决非常大的 BLOB 存储性能问题的一种特殊的存储方法。文件流没有被存储在在一组 BLOB 页上,而是被存储在 NTFS 目录中,这一目录是由你存储数据的特定 SQL Server 数据库显式创建的。不同于文件系统中存储二进制数据的客户控制系统(以及数据库中的指针),SQL Server 会自动协调文件版本——允许文件流参与事务和备份。

提示:

文件流是一个很特殊的领域,但确是相当重要的。后一章以及第 21 章将更详细地对其进行介绍。

6.2 理解索引

在韦伯斯特词典中,将索引定义为:

通常以字母顺序排列的某种指定资料(如作者、主题或关键字)的列表(如书目信息或对著作正文的引用)。

在数据库中,我将用更简单的方法来说明,称索引是可以以很快的速度到达数据的一种手段。尽管如此,韦氏词典的定义也还不错——即便是对于当前特定的用途而言。

或许在韦氏词典的定义中,要指出的关键事情是使用的“通常”一词。在很多规则下,“字母顺序”的定义会发生改变。例如,在 SQL Server 中,有许多不同的排序选项可用。这些选项包括:

- **二进制:** 根据字符的数字表示(例如,以 ASCII 表示时,空格用数字 32 表示,字母 D 是 68,而字母 d 是 100)进行排序。由于所有的事物都是数字的,因此这是最快的一种排序选择;遗憾的是,它不是人们思考问题的方式,因而实际上在 WHERE 子句中,可能会带来麻烦。
- **字典顺序:** 这种排序方式就是你在字典中看到的排序方式,不过有一点改变,可以设置许多不同的附加选项,以确定区分大小写、区分重音以及字符集选项。要记住,每种语言都会加入自己的字典排序方式,因此如果选择对面向非英语的语言进行排序,可能会看到不同的排列顺序。

理解这样的内容是相当容易的:如果告诉 SQL Server 注意大小写,那么 A 将不等于 a。同样地,如果告诉 SQL Server 不区分大小写,那么 A 将等同于 a。当添加区分重音选项时,事情变得有点不易理解——即是说,这时 SQL Server 会留意变音符号,因此 a、á 和 à 都不同。排序规则不仅会影响数据是否相等,而且会影响排序的顺序(因此,会影响到在索引中存储的方式),这一点

对于许多人而言则更加不易理解。

通过下面的例子，考查表 6-1 中列出了几种排序规则的性质，并说明了它们对于排序顺序和相等信息有何影响。

表 6-1

排 序 规 则	比 较 值	索引存储顺序
字典顺序，不区分大小写，不区分重音(默认值)	$A = a = \grave{a} = \acute{a} = \hat{a} = \ddot{A} = \ddot{a} = \text{\AA} = \text{\aa}$	$a, A, \grave{a}, \acute{a}, \hat{a}, \ddot{A}, \ddot{a}, \text{\AA}, \text{\aa}$
字典顺序，不区分大小写，不区分重音，大写字母优先	$A = a = \grave{a} = \acute{a} = \hat{a} = \ddot{A} = \ddot{a} = \text{\AA} = \text{\aa}$	$A, a, \grave{a}, \acute{a}, \hat{a}, \ddot{A}, \ddot{a}, \text{\AA}, \text{\aa}$
字典顺序，区分大小写	$A \neq a, \ddot{A} \neq \ddot{a}, \text{\AA} \neq \text{\aa}, a \neq \grave{a} \neq \acute{a} \neq \hat{a} \neq \ddot{A} \neq \ddot{a}, A \neq \ddot{A} \neq \text{\AA}$	$A, a, \grave{a}, \acute{a}, \hat{a}, \ddot{A}, \ddot{a}, \text{\AA}, \text{\aa}$

这里的要点是，索引中的操作依赖于为数据确定的排序信息。排序可以在数据库级别和列级别设置。因此，在控制的级别上可以达到相当精细的粒度。如果要服务器采用不区分大小写的排序规则，那么，必须确定系统文档能处理此事，否则，就准备着接到许多技术支持电话——尤其是如果是在美国之外出售。假设你是独立软件开发商(ISV)，购买你产品的客户把产品安装在现有的服务器上(对于客户来说，这个服务器上所有的事情都像是合理的)。可是，那个现有的服务器刚好是设置为区分大小写的。那么，你就等着接到怒气冲冲的客户打来的支持电话吧。

注意：

一旦设置了排序顺序，要更改它很不容易(但也是可能的)，因此要在设置前就确定好想要的排序规则。

6.2.1 “B” 还是非“B”：B 树

显然，平衡树(或 B 树)的概念不是 SQL Server 创建的。实际上，在数据库领域之内或之外大量的索引系统中，都在广泛地使用着 B 树。

B 树只是试图提供一种一致的、成本相对较低的方法，以找到一条特定的信息。名称中的平衡非常有自我描述性——B 树是自平衡的，极少有例外，这意味着每当树分叉的时候，将近一半的数据在一边，一半的数据在另一边。在这一点上，名称中的“树”也很显而易见(提示：树、分枝——看到这里的趋势了吗？)在这里使用“树”是因为在画出结构并将其倒转时，它具有树的一般形状。

B 树从根节点(这是关于树的又一个类比，但不是最后一个)开始。如果数据量较少，根节点可以直接指向数据的实际位置。在这种情况下，整个结构看起来将像图 6-2 所显示的那样。

那么，我们从根节点出发，开始浏览记录，直至找到最后一个起点小于我们要找寻的值的页。然后，获取到那个节点的指针，并浏览该节点，直至找到要找寻的行为止

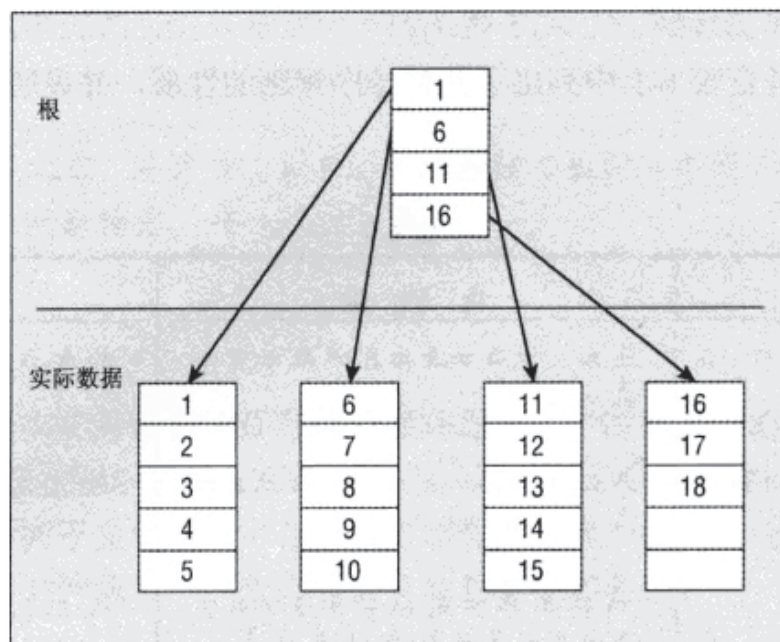


图 6-2

可是，在多数情况下，要从根节点引用的数据实在太多了，因此，根节点指向中间节点——或者称为非叶级节点。非叶级节点位于根节点和节点(说明数据在物理上存放于何处)之间。随后非叶级节点可以指向其他的非叶级节点，或者指向叶级节点(最后的树类比引用——我保证)。叶级节点是获得真实物理数据的实际引用节点。与叶子沿着树的轮廓所达到的末梢很类似，顺着索引的线一直走到尽头就到达了叶级节点——从这里，可以直接到达存放了数据的数据节点)。

在图 6-3 中可以看到，我们和之前一样也是从根节点开始，随后移动到下一级中具有等于或小于我们正在寻找的值的最大开始值的节点。然后重复这个过程——查找等于或低于我们要查找的值的最高起始值节点。沿着树一直这样一级一级地做下去，直到到达叶级——此时我们知道数据的物理位置而且可以快速地浏览它了。

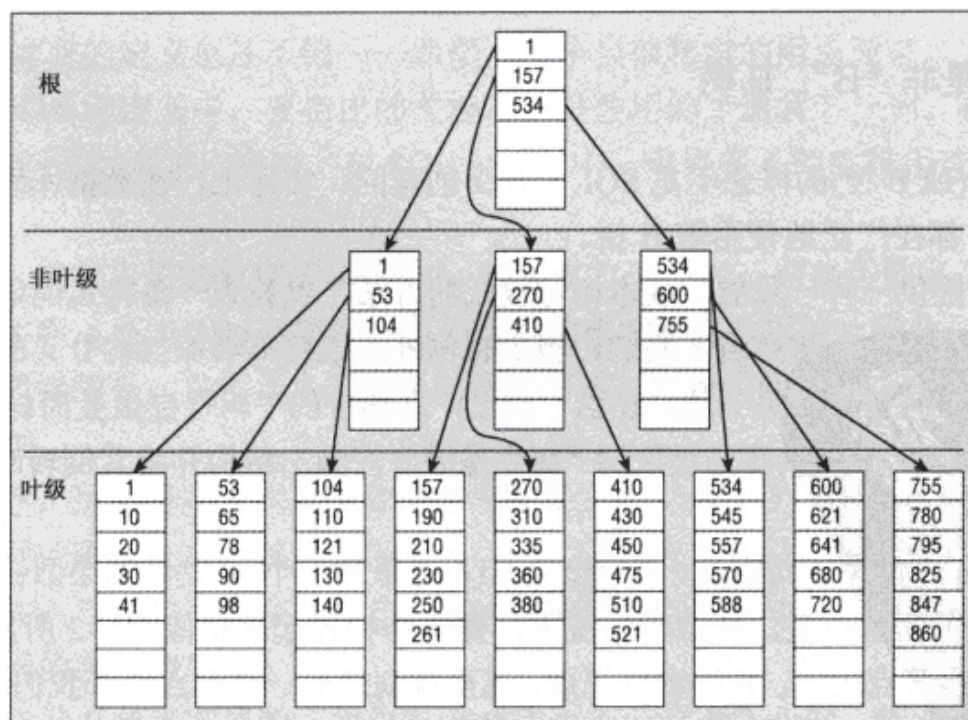


图 6-3

1. 页拆分——深入研究

所有读取方面的工作似乎都进行得很顺利，只是插入操作有点麻烦。回忆一下 B 树中的 B 代

表的是平衡。你可能还记得我提到过 B 树是平衡的，因为每次遇到一个树枝，两边都有一半的数据。有时候，B 树指的是自我平衡，因为向树添加新数据的方式一般会避免向一边倾斜。

当向树中加入数据时，节点最终会被填满，进而需要拆分。因为在 SQL Server 中，节点等同于页，所以这种拆分被称为页拆分，如图 6-4 所示。

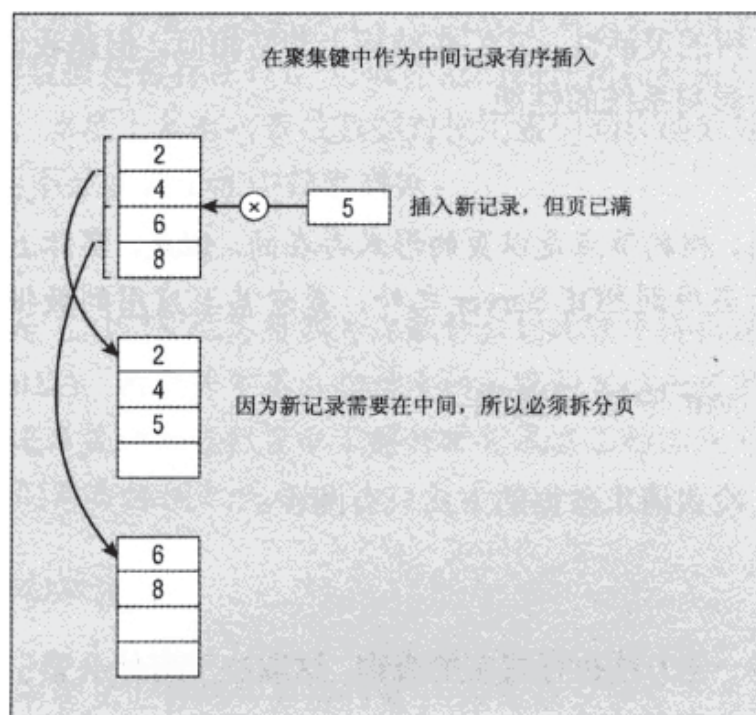


图 6-4

提示：

在拆分页时，数据被自动移动以保持平衡。前面的一半数据留在旧的页中，而剩余的数据被加入新页——这样就有了一个差不多对半的拆分，因而树依然保持平衡。

考虑一下拆分的过程，会意识到在拆分时增加了大量的开销。不仅仅是插入一个页，而是：

- 创建新页；
- 把数据从现有页上移动到新页；
- 向其中一页加入新行；
- 向父节点中加入另一个条目。

不过开销不止这些。由于是在树的排列中，有可能会产生一些级联的操作。创建一个新页时(因为拆分)需要在父节点中另外加入一个条目。父节点中的条目也可能导致那一级别的页拆分，拆分的过程又全部重头开始。实际上，可能性会一直向上延伸，甚至会影响到根节点。

如果根节点发生拆分，那么，实际上最终将创建两个额外的页。因为只能有一个根节点，原来的根节点页拆分成两个页，进而成为了树中新的中间节点。然后，创建一个全新的根节点，该节点中有两个条目(一个指向旧的根节点，另一个是指向拆分页)。

勿庸讳言，页拆分会对系统的性能造成非常负面的影响。并且，在进行页拆分处理的服务器上会表现出停顿好几秒的行为(在对页进行拆分和重写时)。

在结束本章前，我们将讨论如果防止页拆分。

提示：

叶级的页拆分是极其平常的，中间节点的页拆分则少得多。随着表的增大，索引的每一级都将经历页拆分，但是，由于中间节点中的一个条目对应着下一级节点中的几个条目，所以，随着

在树中级别的升高，页拆分的发生频率也越来越少。尽管如此，在叶级的上一级发生了拆分，那么下一级节点必定也已经进行了拆分——实际上，这意味着沿着树向上页拆分是累积的(并且在性能上的开销是昂贵的)。

SQL Server 有许多不同类型的索引(将在后面讨论)，不过它们都以这样或那样的方法使用 B 树方法。事实上，由于 B 树的灵活性，它们在结构上非常相似。但是我们仍会看到确实存在一些显著的区别，而且它们会影响系统的性能。

提示：

对于 SQL Server 索引，树的节点是以页的形式存在的，但是，实际上可以将根节点、非叶级、叶级以及树的结构等概念应用到 SQL Server 之外，或者甚至应用到数据库之外的领域。

6.2.2 如何在 SQL Server 中访问数据

大体上，SQL Server 检索请求数据的方式只有两种：

- 使用表扫描
- 使用索引

SQL Server 将使用哪一种方法执行特定的查询，这取决于有什么索引可用、询问的是什么列、在进行什么类型的联结以及表的大小。

1. 使用表扫描

表扫描是非常直观的过程。在执行表扫描时，SQL Server 从表的物理起点开始，浏览表中的每一行。当发现符合查询条件的行时，把这些行包含在结果集中。

你可能听到过很多关于表扫描不好的事情，一般来说，这是真的。不过，在某些实例中，表扫描实际上可能是最快的访问方法。典型的情况是，当从相当小的表中检索数据时就是如此。在表的大小具体为多少时表扫描会成为最快的访问方法，这将随表的宽度和查询的特定性质的变化而变化。

提示：

看看你是否可以说明，为什么在适当的时候在查询的 WHERE 子句中使用 EXISTS 可以极大地提升性能。使用 EXISTS 运算符时，SQL Server 只要一找到匹配查询条件很快就会主动停止查找。如有一个有一百万条记录的表，表中第三个记录就是匹配的记录，那么，使用 EXISTS 选项会省去读取 999 997 条记录的工作量！NOT EXISTS 也以同样的方式起作用。

2. 使用索引

当 SQL Server 决定使用索引时，实际上，其处理过程与表扫描多少有些相似，只是这里有一些捷径。

在查询优化过程中，优化器查看所有可用的索引，然后选择最佳的一个(这主要基于在联结和 WHERE 子句中指定的信息，结合 SQL Server 保存的关于索引组成的统计信息)。一旦选定了索引，SQL Server 在索引的树结构中导航，到达匹配查询条件的数据，再提取需要的记录。区别之处在于，由于数据是排序的，因此查询引擎知道何时到达当前找寻范围的终点。于是，它可以结束查

询,或者在需要时继续移动到下一级的数据范围。

思考一下迄今为止学过的查询主题,你可能会注意到,索引与 EXISTS 选项在作用方式上有惊人的相似之处。EXISTS 关键字允许在找到匹配值后立即退出查询运行。索引利用类似的方式搜索数据的过程,因此使用索引能同样或者甚至更好地改进性能——即,服务器可以了解什么时候不再存在任何相关的数据,并能在这里停止。不过使用索引甚至能更好,因为无需局限于布尔逻辑的情形中(要寻找的数据是否存在?),可以在范围的开始和结束处应用同样的概念——可以把数据范围聚集在一起,本质上具有与查找数据时使用索引同样的好处。此外,可以非常快速地查找数据(称为 SEEK)而不是在整个表中搜索数据。

提示:

不要从我对于索引与 EXISTS 运算符能为你做什么的比较中得出这样的印象:索引会完全取代 EXISTS 运算符(或相反)。两者并不是互相排斥的,它们可以一起使用,并且常常这样使用。这里把它们放在一起讲述只是因为它们可以了解何时完成自己的工作并能在抵达表的物理末尾之前退出,在这一点上它们具有相似之处。

6.2.3 索引类型和索引导航

虽然在 SQL Server 中名义上有两种类型的索引(聚集索引和非聚集索引),但实际上,内部说来,有 3 种索引类型。

- 聚集索引;
- 非聚集索引,其中包含:
 - 堆上的非聚集索引;
 - 聚集索引上的非聚集索引。

物理数据的存储方式在聚集索引和非聚集索引之间有所不同。SQL Server 遍历 B 树到达最终数据的方式在 3 种索引类型之间各不相同。

所有 SQL Server 索引都有叶级页和非叶级页。在讨论 B 树时曾说过,叶级是保存标识记录的“键”的,非叶级是引导到叶级的。

索引要么构建在聚集表(有聚集索引的表)上,要么构建在堆(没有聚集索引的表)上。

- 聚集表是任何有聚集索引的表。很快将详细讨论聚集索引,不过它们对于表的意义是按照指定的顺序物理地存储数据。通过使用聚集键(定义聚集索引的列)唯一标识单独的行。

提示:

你可能会产生这样的疑问,“如果聚集索引不是唯一的,将会怎样呢?”即是说,如果聚集索引不是唯一索引,怎样才能用聚集索引唯一标识一行呢?答案是隐蔽的:SQL Server 强制所有的聚集索引唯一,即使没有把聚集索引定义成唯一的,也是如此。幸运的是,它强制索引唯一的方式不会改变索引的使用方式。如果你愿意,仍然可以插入重复的行,只是 SQL Server 会在内部为键添加后缀,以确保该行具有唯一的标识符。

- 堆是任何没有聚集索引的表。这种情况下,将基于该行的区段、页和行偏移量(从页首到该行的距离)的组合来创建唯一标识符或行 ID(RID)。只有当没有聚集键可用时(没有聚集索引),RID 才是必需的。

1. 聚集索引

对任何指定的表来说，聚集索引是唯一的——每个表只能有一个聚集索引。并非一定要有聚集索引。但是，你会发现它是最经常作为第一个索引选用的索引类型，由于各种各样的原因，查看索引类型的时候，那会是非常明显的。

聚集索引的特殊之处在于，它的叶级是真正的数据——即数据按照索引或相关键命令中定义的物理顺序进行排序存储。这意味着，一旦到达索引的叶级，就到达了终点——这里是真正的数据。新记录根据它在聚集索引里正确的物理顺序进行插入。创建新页的方式会随着需要插入记录的位置的变化而变化。

当新记录必须插入到索引结构的中间时，会发生常规的页拆分。来自原来页中后半部分的记录被移动到新页上，新记录适当地插入到新的或旧的页中。

当新记录在逻辑上位于索引结构的末尾时，将创建一个新页。但是，只有新记录添加到新页中，如图 6-5 所示。

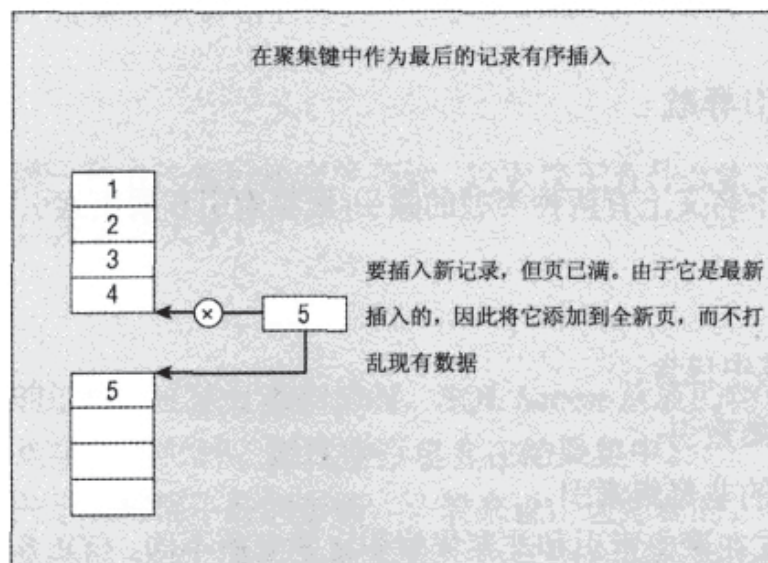


图 6-5

在树中导航

我在前面说过，SQL Server 中的索引以 B 树结构存储。在理论上，在 B 树分叉的每一个可能的方向上，总是有一半的剩余信息。下面来看一下聚集索引的 B 树的图示(如图 6-6 所示)。

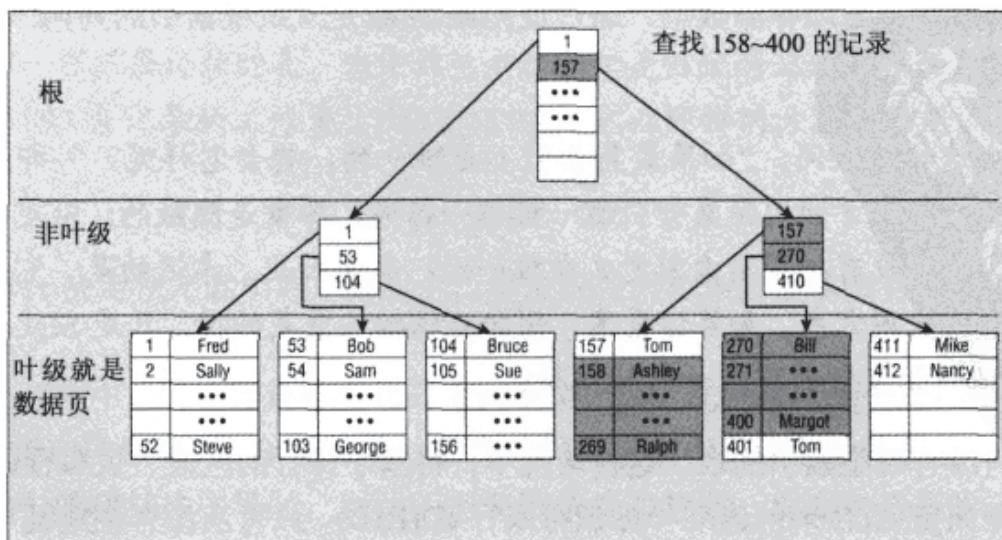


图 6-6

可以看出,它实际上与本章前面讨论的更一般的B树是一样的。在这里进行的是范围搜索(有时聚集索引尤其擅长此类事情),搜索158~400的数字。只需按照如下步骤进行:

导航到第一条记录,并包含该页中所有的其余记录(之所以知道需要该页中其余的记录,是因为我们从上一级节点的信息中得知还需要来自其他页中的数据。因为这是一个有序的列表,所以可以确定它是连续的),这意味着如果下一页中有需要包含进来的记录,那么本页中其余的记录必定都要包含进来。我们可以开始从这些页中提取记录,而不需要完成确认方面的工作。

我们从导航到根节点开始。SQL Server可以基于一个保存为系统表的条目定位根节点。可以通过查询sys.indexes查看那个表的逻辑内容。

注意:

数据库中的每一个索引在sys.indexes中都有一个条目。系统视图是数据库的一部分(与主数据库中相反)而且会显示数据库中所有索引的存储位置信息,以及它们基于哪一行。

在版本较旧的SQL Server中,可以查询基础表(在技术上你可以这么做,不过我强烈推荐你此时不要直接查询),即所谓的sysindexes表。

浏览作为根节点的页,可以知道接下来要检查的页是什么(正如图中所画出的,我们要查看第二级的第二页)。然后,继续处理。随着我们沿着树一步步向下,将得到越来越小的数据子集。

最终,我们会到达索引级别的叶级。在聚集索引中,到达索引的叶级意味着也到达了要找寻的行和要找寻的数据。

注意:

关于区别重要性,我已经强调过很多次了。有了聚集索引,在完整浏览索引的时候就已经完整地浏览了自己的数据。在你观察非聚集索引的时候就会发现这样做对性能的影响差别有多大——特别是在聚集索引上建立非聚集索引的时候。

2. 堆上的非聚集索引

堆上的非聚集索引在各方面都与聚集索引的作用方式很类似。不过它们确实有几个显著的区别:

叶级不是数据,而是你可以获得数据指针的级别。指针以索引所指向的特定行的行标识符(RID)的形式出现,本章前面介绍过RID,它由索引指向的特定行的区段、页和行偏移量组成。尽管叶级并非真正的数据(而是具有RID)。不过,这里只比使用聚集索引多了一步而已——因为RID包含行位置的完整信息,因而可以直接访问到数据。

不过不要误以为“多了一步”意味着在开销上只有一点点的差别,进而误以为堆上的非聚集索引与聚集索引运行得一样快。使用聚集索引时,数据的物理顺序是索引顺序。这意味着,对于一定的数据范围,当找到该范围上数据起点所在的行时,很可能该页上其他行也和它一起在这一页上(也就是说,因为它们存储在一起,所以在物理上你基本上已经到下一条记录了)。使用堆的时候,数据没有链接在一起,唯一的连接就是通过索引。从物理的观点看,这里完全没有任何形式的排序。就是说,从物理读取的观点看,系统可能要在文件中四处检索数据。实际上,十分可能(或者很可能)要花上单独的几次从同一个页中获取数据,因为数据间没有链接,所以SQL Server无法知道它将返回这个物理位置。使用聚集索引时,因为知道物理的顺序,所以可以只访问页一次就获取到所有的数据。

提示:

为了公平对待堆上的非聚集索引与聚集索引,需要说明,任何已经读过一次的页极有可能依然存在于内存缓存中,因而对检索它应该是非常快的。不过,它在检索数据上还是增加了另外的逻辑操作。

图 6-7 显示了曾在聚集索引上进行过的同样搜索,不过,这一次使用的是堆上的非聚集索引。

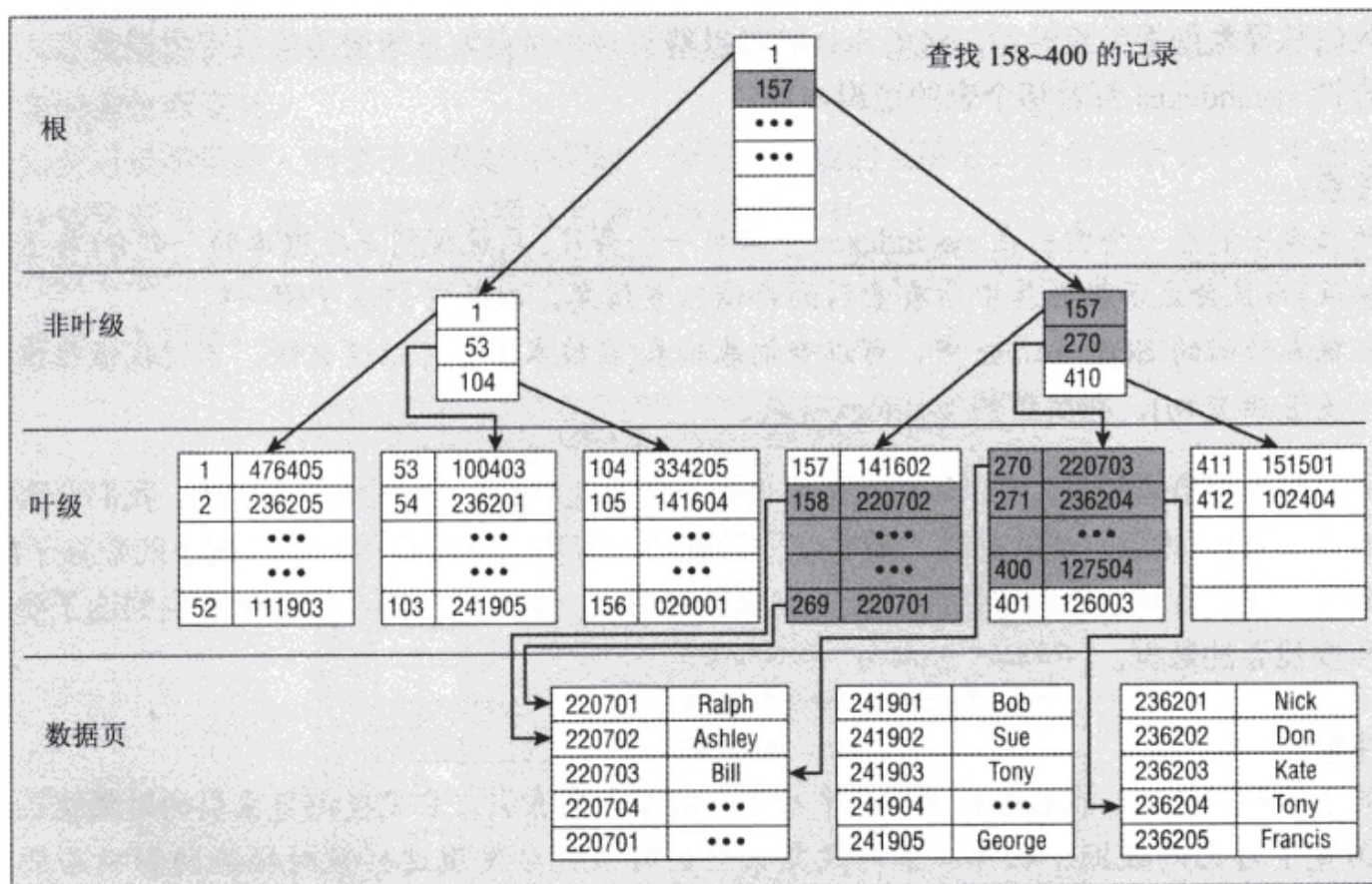


图 6-7

在大多数索引导航中,这里所做的事情与以前完全一样。从同一根节点开始,遍历树处理越来越集中的页,直至到达索引级别的叶级。此时,事情开始与以前有所不同。使用聚集索引时,就到此为止了,而使用非聚集索引时,要做更多的事情。如果非聚集索引在堆上,那么还需要到达一个级别。从叶级页获取行标识符,然后导航到该行;一直到这时方才找到实际的数据。

3. 聚集表上的非聚集索引

使用聚集表上的非聚集索引时,同样有相似之处,但也同样有不同之处。与堆上的非聚集索引一样,索引的非叶级看起来和聚集索引非常类似。只有到达叶级才可以看到不同之处。

提示:

在叶级,它与其他两种索引结构有显著的区别——这里还要查看一个索引。使用聚集索引,到达了叶级便找到了真正的数据。使用堆上的非聚集索引时,到达叶级还没有得到真正的数据,但是,找到了引领你直达数据的标识符(只剩最后一步要走)。使用聚集表上的非聚集索引,在叶级得到的是聚集键。这就是说,在这里找到了足够的信息来使用聚集索引。

最后你会看到类似图 6-8 所示的情形。

最终得到的是两个类型完全不同的查找。

在图中的例子里，从范围查找开始——在索引中进行一个查找，并可以浏览非聚集索引找到满足条件(LIKE'T%')的连续数据范围。这种可以直接到具体的索引所在地点的查找称为查找(seek)。

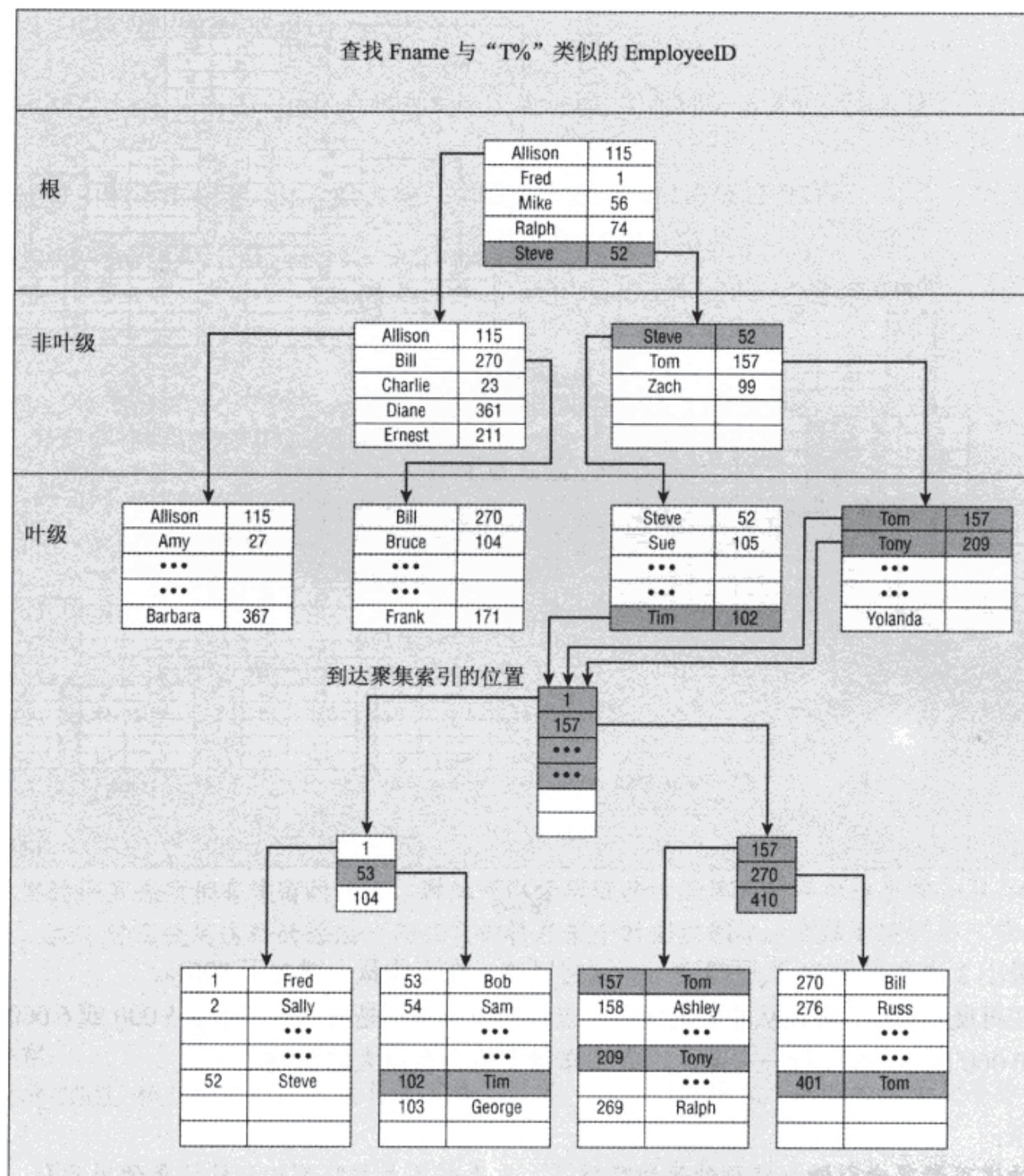


图 6-8

然后，开始了第二种类型的查找——使用聚集索引的查找。第二种查找非常快，这里问题是它将发生多次。我们来看，SQL Server 从第一个索引查找中获取了一个列表(所有以“T”开头的名字列表)，不过这个列表在逻辑上不以任何连续的方式与索引键匹配——必须单独查找每一条记录，如图 6-9 所示。

显然，这种多次查找的情形比起从一开始就使用聚集索引的情形，引入了更多的管理开销。第一个索引搜索(通过非聚集索引进行的搜索)只需要很少的逻辑读。

例如，如果有一个表，表中每一行为 1000 字节，要对该表进行与前面图中类似的查找(比如要返回 5、6 行的查找结果)，从非聚集索引中获得信息将只需要完成 8~10 次逻辑读。不过到目前

为止，只是准备好了在聚集索引中查找行。每一次查找都将花费大约 3~4 次逻辑读，或 15~24 次附加读取。乍一看，这似乎不是什么大问题，但是，如果像下面这样考虑：

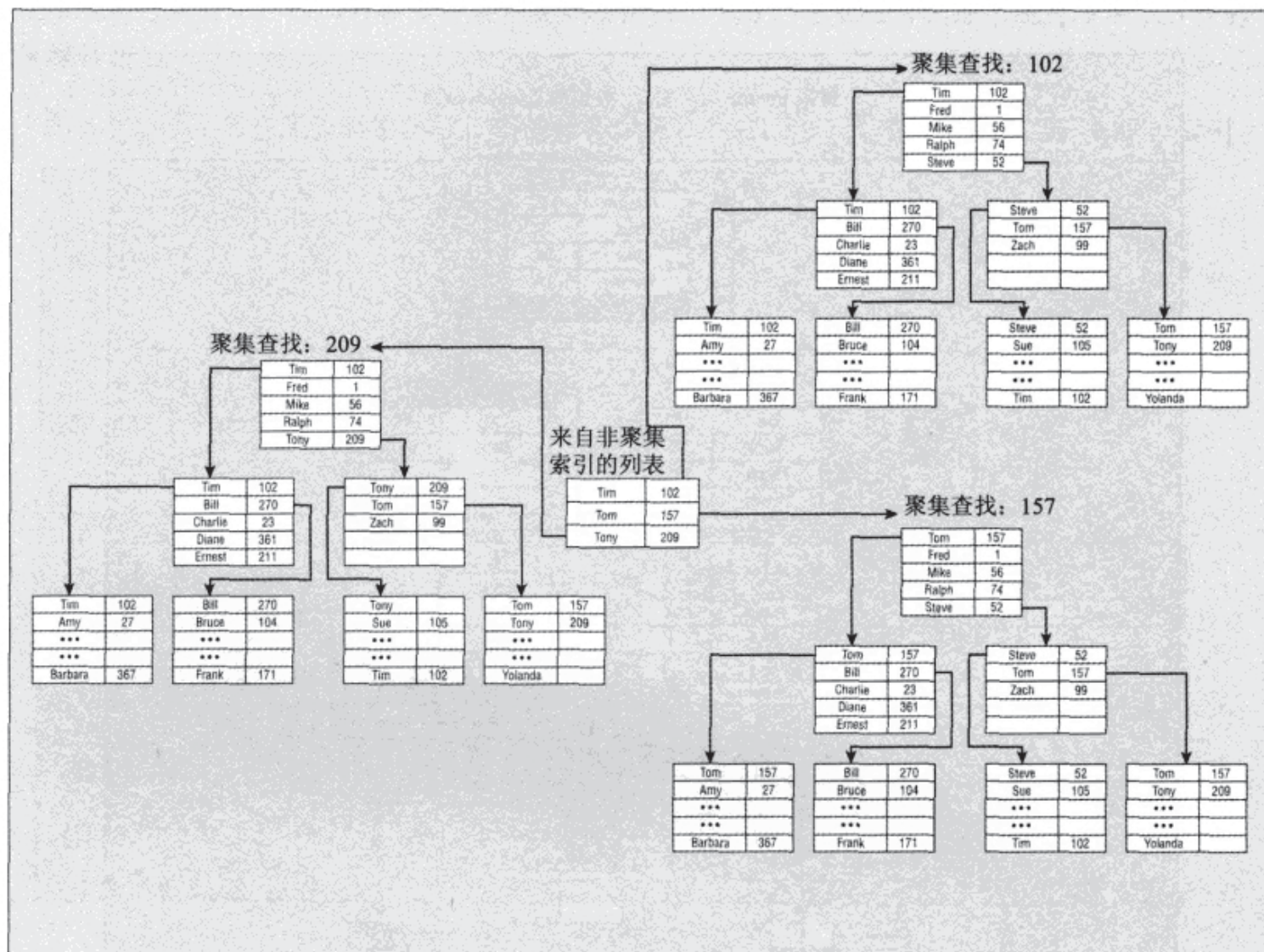


图 6-9

从最小 3 次到最大 24 次逻辑读——比起以前，在工作量上增加了 800%。

现在再展开想象，假设从非聚集索引中查找的值范围不是五六行，而是 5 000 或 6 000 行，或者是 500 000 或 600 000 行——那么，在工作量上将会有巨大的影响。

提示：

别被这里聚集索引增加的额外开销吓倒了。重点并不是要吓倒你让你远离使用索引，而是要让你意识到，从读的观点来看，非聚集索引不会像聚集索引那样有效(实际上，在某些情况下，在进行插入的时候它可以成为更好的选择)。通常(有例外的时候)，任何类型的索引都是进行查找的最快的方法。本章后面将说明使用什么索引以及为什么要使用这种索引。

6.3 创建、修改和删除索引

索引的创建、修改和删除与其他对象(例如表)几乎一样。下面分别来看这些操作，从 CREATE 开始。

可以按照下面的两种方式创建索引：

- 通过显式的 CREATE INDEX 命令创建;
- 在创建约束时作为隐含对象创建。

每一种方式在能做什么和不能做什么上都有自己特异之处, 因此, 接下来分别讨论它们。

6.3.1 CREATE INDEX 语句

CREATE INDEX 语句完成的事情与它听上去一样。它在特定的表或视图上基于声明的列创建索引。

创建索引的语法如下, 这里会介绍几个迄今为止尚未讨论过的项目:

```
CREATE [UNIQUE] [CLUSTERED|NONCLUSTERED]
INDEX <index name> ON <table or view name>(<column name> [ASC|DESC] [,...n])
INCLUDE (<column name> [, ...n])
[WITH
[PAD_INDEX = { ON | OFF }]
[[,] FILLFACTOR = <fillfactor>]
[[,] IGNORE_DUP_KEY = { ON | OFF }]
[[,] DROP_EXISTING = { ON | OFF }]
[[,] STATISTICS_NORECOMPUTE = { ON | OFF }]
[[,] SORT_IN_TEMPDB = { ON | OFF }]
[[,] ONLINE = { ON | OFF }
[[,] ALLOW_ROW_LOCKS = { ON | OFF }
[[,] ALLOW_PAGE_LOCKS = { ON | OFF }
[[,] MAXDOP = <maximum degree of parallelism>
]
[ON {<filegroup> | <partition scheme name> | DEFAULT }]
```

提示:

这里的许多选项都有遗留的语法, 因此可以使用这些语法来支持以前版本的 SQL Server。尽管如此, 我不赞成使用这样的语法, 而且它们将在某个时刻被删除。我强烈建议尽可能使用较新的语法。

注意:

创建 XML 和空间索引的语法与这有点儿类似, 但区别也很大。本节的后面会单独讲述这个语法。

宽泛地讲, 这里的语句遵循你已经见过很多次(并将见到更多)的 CREATE<object type><object name>语法。主要的障碍是在其他地方没有见过的参数。

就像将在后面一章的视图中看到的那样, 必须在 CREATE 语句中加入另外的子句, 以处理索引其实不是一个孤立对象这个事实。索引必须伴随表或视图出现, 因而必须声明索引所在列为“ON”的表。

在 ON<table or view name>(<column name>)子句后, 所有的事物都是可选的。可以搭配使用这些选项。这里的许多选项都很少用到, 但是, 某些选项(诸如 FILLFACTOR)可以对系统的性能和行为产生重大的影响, 因此, 下面来逐一讨论它们。

1. ASC/DESC

这两个参数允许为索引选择升序和降序排序顺序。默认值是 ASC，正如你所猜想的，默认是升序。

这里可能会有点疑惑，为什么升序与降序与索引有关系呢——如果 SQL Server 需要相反的排序顺序，只需反向查看索引就可以了。不过事情不总是那样简单。如果只涉及一列，或者如果所有的列总是以同样的方式排序，那么，可以反序查看索引。但是，如果想在索引中混合升序、降序的排序方式呢？即，如果需要一列按升序排序，而另一列按降序排序，这又将如何呢？由于索引列是存储在一起的，因此对一列反向查看索引也将倒转另一列的顺序。如果显式地说明一个列为升序，而另一列为降序，则会直接在物理数据中倒转第二列——在不经意间改变了访问数据的方式。

下面举一个简短的例子，设想一个生成报表的场景。想要把雇员列表按照雇佣日期排序，日期最近的排在前面(降序)。但是，还想要按照姓氏进行排序(升序)。在以前的版本下，SQL Server 必须完成两个操作：一个用于第一列另一个用于第二列。通过允许控制数据的物理排序顺序，可以在组合列上获得灵活性。

一般来说，将不理睬这里(再次说明，记住向后兼容性)。可能的一些例外情况是：

- 需要在多个列上混合升序和降序；
- 向后兼容性不是问题。

2. INCLUDE

这个参数是新加入 SQL Server 2005 的。它的目的是为了更好地支持覆盖查询(covered query)。当使用的索引包含查询要用到的所有数据时，认为该查询是“覆盖的”。如果需要的全部数据都在索引中，那么就没必要进入到实际的数据页中，只要到达索引的叶级就得到了所有需要的数据，因此可以停止在这里(省下了大量的 I/O 操作)。

当包含列而不是把它们放在 ON 列表中时，SQL Server 只将它们添加到索引的叶级。由于索引叶级的每一行都对应一个数据行，因此，本质上这里所做的是把更多的原始数据包含在索引内。如果你考虑了这个问题，那么你可能已经猜到 INCLUDE 实际上只适用于非聚集索引(聚集索引已经是叶级数据，所以与此无关)。

这样做有什么影响呢？本书随后会深入讨论这个问题。SQL Server 一旦得到自己真正需要的东西就会立刻停止工作。因此，如果在遍历索引时，没继续访问实际的数据行就得到了所有需要的数据，那么就不用再前往数据行(这样做有什么意义？)。通过在索引中包含特定的列，可以利用那个特定索引在叶级“覆盖”查询，从而节省了与使用索引指针到达数据页有关的 I/O。

注意：

要小心，不要滥用该参数！在包含列的时候，将增加索引页在叶级的大小。这意味每一页中的行数将减少，因而要看到同样数目的行可能需要更多的 I/O。结果可能是，加快一个查询可能会减慢其他的查询。套用一部上个世纪八十年代的老电影，“Balance Danielson——balance!” 要考虑对系统各部分的影响，而不要只考虑某个时候正在使用的特定的查询。

3. WITH

WITH 是一个易于使用的参数——它只是用来告诉 SQL Server 你将要提供一个或多个后面的

选项。

4. PAD_INDEX

这个参数在语法列表中第一个出现——不过，在理解了 PAD_INDEX 的作用后，会觉得这样有点古怪。简言之，这个参数决定首次创建索引时，索引的非叶级页的满度(以百分数表示)。在 PAD_INDEX 中不声明百分比，因为它将使用随后在 FILLFACTOR 选项中指定的百分比。如果没有 FILLFACTOR，设置 PAD_INDEX=ON 是毫无意义的(这就是为什么先出现这个参数会有些古怪的原因)。

5. FILLFACTOR

SQL Server 首次创建索引时，在默认情况下，将尽可能满地填充页，仅留下两条记录的空间。可以将 FILLFACTOR 设置为 1 到 100 之间的任何值。这里的数字是一个百分比，它表明一旦索引构造完成，页将有多满。但是要记住，拆分页的时候，数据在两个页之间以平分的方式分布。除了定期重新创建索引(你应该做的事情——设置维护计划，这部分内容将在第 23 章中讨论)外，不能在日常操作中控制填充的百分比。

当需要调整页的信息密度时会使用 FILLFACTOR。请按照下面的方式来考虑：

- 如果是 OLTP 系统，希望 FILLFACTOR 较低；
- 如果是 OLAP 或其他非常稳定的(从变化方面讲，几乎没有添加和删除)系统，则希望 FILLFACTOR 尽可能高；
- 如果系统中事务比例中等，且有许多基于它的报表类型的查询，那么可能希望选取中等值的 FILLFACTOR(不太低，也不太高)。

如果未提供值，SQL Server 会在每页最少有一行的情况下，把页填充到差两行满(例如，如果行是 8 000 字符宽，那么每页只能放入一行，因此无法达到差两行满)。

6. IGNORE_DUP_KEY

IGNORE_DUP_KEY 选项差不多算是一种回避系统的方法。简言之，它导致 UNIQUE 约束与其应有的行为方式有些不同。

通常，唯一约束或唯一索引不允许有任何种类的重复。如果事务试图基于一个定义为唯一的列创建重复值，则事务将被回滚和拒绝。然而，当设置了 IGNORE_DUP_KEY 选项后，将得到某种混合的行为。虽然仍然会收到错误消息，但是，错误只是警告级别的错误。记录依然没有被插入。

从 IGNORE_DUP_KEY 的角度看，最后一行的“记录依然没有被插入”是一个重要的概念。没有为事务发出回滚(错误是一个警告错误而非重大错误)，但重复行将被拒绝。

为什么要这样做？这是一种不打扰试图插入重复值的事务来存储唯一值的方法。对于任何正在插入重复值的过程；或许，关于要插入的行是一个重复行这一点完全无关紧要(没有源自它的逻辑错误)。相反，这个过程的态度更可能是，“嗯，只要我知道那里已经存在一个这样的行就够了。我并不在乎行是否是试图插入的特定行”。

7. DROP_EXISTING

如果指定 DROP_EXISTING 选项，任何具有讨论中的索引名的现有索引将被删除，以便创建

新的索引。当该选项与聚集索引一起使用时，与简单地删除并重新创建现有索引相比，该选项更加有效。如果重建与现有索引完全匹配的索引，SQL Server 就会知道它无需触及非聚集索引。但是，为了适应不同的行位置，显式的删除和创建将导致所有的非聚集索引重建两次。如果使用 DROP_EXISTING 修改了索引的结构，只会重建 NCI 一次而非两次。此外，不能简单地删除和重建由约束创建的索引，例如，要实现一个确定的填充因子。可以使用 DROP_EXISTING 完成这一使命。

8. STATISTICS_NORECOMPUTE

在默认情况下，SQL Server 尝试自动更新表和索引上的统计信息。通过选择 STATISTICS_NORECOMPUTE 选项，表明将由你负责统计信息的更新。要关闭该选项，需要运行 UPDATE STATISTICS 命令，但不使用 NORECOMPUTE 选项。

我强烈建议不要使用该选项。为什么呢？查询优化器将使用索引上的统计信息来确定索引对于给定的查询有多大的用处。由于表中的特定数据增多和减少的数额很大，而且由于列中特定的值发生改变，索引上的统计信息在持续地变化着。结合这两个事实，应该能看出，不更新统计信息意味着查询优化器将基于过时的信息运行查询。保持自动更新统计信息功能启用意味着统计信息将定期更新(多长时间更新一次取决于对表进行更新的频率和类型)。相反，关闭统计信息自动更新意味着要么你的信息会过时，要么你必须设定计划以便手工运行 UPDATE STATISTICS 命令。

9. SORT_IN_TEMPDB

只有当存储 tempdb 的驱动器与包含新索引的数据库所在的驱动器在物理上隔离时，该选项才有意义。它在很大程度上是一种管理功能，因此，在简单概述了它是什么以及为什么只有当 tempdb 在单独的物理设备上时它才有意义后，我不会对这一主题做过多的探讨。

SQL Server 建立索引时，必须完成多次读取以完成不同的索引构造步骤：

(1) 通读所有数据，构造与实际数据的每一行相对应的叶行。正像实际的数据和最终的索引一样，这些会进入到用于临时存储的页中。这些中间的页不是最终的索引页，而是每当排序缓存填满时用于临时存储的地方。

(2) 在中间页中独立运行，以便将其融入最终的索引叶级页中。

(3) 在构建叶级页的时候，构造了非叶级页。

如果不使用 SORT_IN_TEMPDB 选项，那么中间页将被写到用于存储数据库的同一个物理文件中。这意味着对实际数据的读操作不得不与构建过程的写操作竞争。二者导致磁盘磁头从一个(读与写)需要的地方移动至另一个不同的地方。结果是磁头持续地来回移动，这样会花费很多时间。

而如果使用了 SORT_IN_TEMPDB，那么中间页将写入到 tempdb 中，而非数据库自己的文件中。如果它们在不同的物理驱动器上，就说明构建索引的读和写操作之间没有竞争。然而，要记住，只有当 tempdb 所在的物理驱动器和数据库文件所在的驱动器不同时，该选项才有效；否则，只是名义上发生了改变，而实际上 I/O 争用依然存在。

提示：

如果要使用 SORT_IN_TEMPDB，请确保 tempdb 中有足够的空间来支持大型文件。

10. ONLINE

如果将该选项设置为 ON, 将强制表对于一般性的访问保持可用, 且不创建任何阻止用户使用索引和(或)表的锁。默认情况下, 完全的索引操作将夺取它所需的锁(最终得到表锁)以便于完整且有效地访问表。不过, 这样做的副作用是把用户阻挡在外(是的, 它有点自相矛盾。为了让数据库更好用, 可能要创建索引, 而在创建索引的过程中, 实际上却让表不可用了)。

这样一来你可能会想: “如果随时随地构建索引, 那用户就不会受影响了, 这似乎是个不错的主意吧?” 这是糟糕的想法。要记住, 任何像这样的索引构建都很可能是非常密集的 I/O 操作, 因此会或多或少地影响到用户。再加上构建索引时为了确保不会妨碍到任何用户, 还需要许多额外的开销。如果构建索引时令 SQL Server 对表有自由的支配权, 则索引的构建将快很多。而且因构建索引而对系统产生影响的总时间也会少很多。

注意:

只有企业版的 SQL Server 才支持 ONLINE 索引操作。你可以在其他版本中使用 ONLINE 指令执行索引命令, 不过它会被忽略, 所以如果正在使用的是较低版本的 SQL Server, 你使用 ONLINE 查找并发现自己的用户仍然被索引操作拒之门外, 也不必惊讶。

11. ALLOW ROW/PAGE LOCKS

这个指令的历史比 ONLINE 更久, 也是一个非常、非常高级的话题。鉴于本书的目的以及眼下对于锁的介绍, 这里只简单解释一下。

迄今为止, 本书的讲述中不断重复使用的一个术语是“锁”。如之前讲过的, 这是某种避免在数据完整性上有冲突的占位符。这里看到的 ALLOW 设置是关于索引是否允许那些锁类型的设置指令。该选项属于极度地调整性能的内容。

12. MAXDOP

这个选项为索引的构建覆盖系统关于最大并行度的设置。并行并不是本书中要谈论的内容, 因此这里只稍作讲述。

简短来说, 并行度是指一个数据库操作(这里是索引构建操作)可以使用多少个进程。有一个称为最大并行度的系统设置, 允许你设置每个操作可以使用多少个进程。索引创建中的 MAXDOP 选项允许设置任何你认为合适的并行度, 它可以比基本系统设置的并行度更高, 也可以更低。

13. ON

在 SQL Server 中, 可以选择不把索引与数据存放在一起, 通过使用 ON 选项单独存储索引。从以下几个方面来看, 这样做的效果不错:

- 索引所需的空间可以分散在其他驱动器上;
- 用于索引操作的 I/O 不会加重物理数据检索的负担。

关于这个选项还有更多可讲述的东西, 但这涉及非常高级的内容。它高度依赖于数据和使用, 因而我们认为它超出了本书的范畴。

6.3.2 随约束隐含创建的索引

我称这样的索引为“意外的索引”。这并不是说索引不应该在那里。当你需要创建索引的约束时，索引无疑是在那里的。只是我见过很多这样的情形：系统中仅有的索引是以这种方式产生的索引。通常，这说明系统的管理员和(或)设计者几乎忘记了索引的概念。

另一方面，你还会在这上面发现一种奇特的倾向——管理员或设计者知道如何创建索引，但却不是真正了解如何说明系统中已经有了什么索引，以及这些索引的作用是什么。这种状况的典型代表是重复的索引。只要索引具有不同的名称，SQL Server 就会很乐意为你创建。

当向表中加入下列两种约束之一时，会创建隐含的索引：

- PRIMARY KEY 约束；
- UNIQUE 约束(也称备用键)。

到目前为止，已经学习了大量的 CREATE 语法，因此这里就不再赘述了。不过要注意，当作为约束的隐含索引来创建索引时，除 {CLUSTERED|NONCLUSTERED} 和 FILLFACTOR 之外，不允许使用其他任何的选项。

6.3.3 ALTER INDEX

ALTER INDEX 命令多少有些欺骗性。截至目前为止，ALTER 命令总是与修改对象的定义有关。例如，ALTER(修改)表以添加或禁用约束和列。ALTER INDEX 是不同的——该命令只与维护有关，而与结构无关。如果需要修改索引的组成，只能要么 DROP(删除)然后 CREATE(创建)索引，要么使用带 DROP_EXISTING=ON 选项的索引。

正如在本章前面看到的，SQL Server 提供了一个选项，用以控制叶级页可以填充到多满，并且，如果选择使用，还有另一个选项用来处理非叶级页。但它们是主动选项。它们只应用一次，然后，在需要时，必须通过重新创建索引并重新应用这些选项来再度应用它们。

在随后关于维护的一节中，将了解到更多关于在哪里以及为什么要使用这一命令的内容，但是现在，只需无条件相信你将在日常维护过程中使用像 ALTER INDEX 这样的维护命令。

ALTER INDEX 语法如下所示：

```
ALTER INDEX { <name of index> | ALL }
    ON <table or view name>
    { REBUILD
        [ [ WITH (
            [ PAD_INDEX = { ON | OFF } ]
            | [[,] FILLFACTOR = <fillfactor>
            | [[,] SORT_IN_TEMPDB = { ON | OFF } ]
            | [[,] IGNORE_DUP_KEY = { ON | OFF } ]
            | [[,] STATISTICS_NORECOMPUTE = { ON | OFF } ]
            | [[,] ONLINE = { ON | OFF } ]
            | [[,] ALLOW_ROW_LOCKS = { ON | OFF } ]
            | [[,] ALLOW_PAGE_LOCKS = { ON | OFF } ]
            | [[,] MAXDOP = <max degree of parallelism>
            ) ]
        | [ PARTITION = <partition number>
            [ WITH ( <partition rebuild index option>
```

```

[ ,...n ] ) ] ] ]
| DISABLE
| REORGANIZE
| [ PARTITION = <partition number> ]
| [ WITH ( LOB_COMPACTION = { ON | OFF } ) ]
| SET ( [ ALLOW_ROW_LOCKS = { ON | OFF } ]
| [ [,] ALLOW_PAGE_LOCKS = { ON | OFF } ]
| [ [,] IGNORE_DUP_KEY = { ON | OFF } ]
| [ [,] STATISTICS_NORECOMPUTE = { ON | OFF } ]
)
) [ ; ]

```

一些选项与 CREATE INDEX 命令相同，因而这里将略去这些选项的重新定义。除此之外，很多特定的 ALTER 选项十分精细，且与处理碎片之类的事情有关(你很快就会看到碎片和维护)或者更加面向数据库管理员并且通常是即兴使用的，以处理非常特殊的问题。尽管如此，这里的核心内容应该是常规维护计划的部分。

下面从最先的几个参数开始讲述，然后再谈论更大的维护计划所需要的选项。

1. 索引名称

如果想维护一个特定的索引，你可以指定特定索引的名称，或者使用 ALL 表明想要维护的与指定表相关联的所有索引。

2. 表名或视图名

顾名思义——想要维护的特定对象(表或视图)的名称。注意，必须是一个特定的表(可以给它提供一个列表，然后说，“请处理所有这些!”)

3. REBUILD

这是用来调整索引的“行业领先”的方法。如果使用该选项运行 ALTER INDEX，将完全丢弃旧的索引并从头重新生成新的索引。最后的结果是，得到了真正最优化的索引，索引中所有叶级和非叶级的页都按照定义的那样进行了重新构建(或者使用默认值，或者使用开关修改填充因子等)。如果讨论中的索引是聚集索引，那么也会重新组织物理数据。

默认情况下，页将被构建成差两条记录填满。正如在 CREATE TABLE 语法中那样，可能把 FILLFACTOR 设置为 0 到 100 之间的任何值。这个值是在数据库完成重新组织后页被填满的程度，以百分比表示。要记住，尽管在页拆分时，数据将会在两个页上平分；但是，除了定期地重建索引外，无法在进行过程中控制填充的百分比。

提示：

在该选项上要小心，一旦开始 REBUILD，在完成索引重建之前，实质上，正在使用的索引就没有了。所有依赖该索引的查询可能会变得异乎寻常地慢(可能要以数量级计)。对于这类事情，首先，需要在脱机系统上测试，以了解整个过程将花费多少时间。然后，安排它在非高峰时段运行(最好有人监控，以确保当高峰时段来临时，它已经回复联机状态)。

运行它可能有许多副作用，因此，以我之拙见，这应当是数据库管理员的领域。

4. DISABLE

该选项所做的事情如同它声称的那样，只是方式多少有点过激。如果命令只是为了让索引脱机，直至你决定了下一步要做什么，则它是个不错选择，但本质上它把索引标记为不可用。一旦禁用了某个索引，在重新激活之前，必须重建索引(不是重新组织，而是重建)。

你很少会自己使用这个选项(你更可能做的是删除索引)——它很有可能在 SQL Server 升级的过程中或者其他一些古怪的情况下用到。

注意：

还有一点要注意！关于这一点的警告。如果你禁用了自己表的聚集索引，它就有禁用表的效果。数据还在，不过，在你重建聚集索引前，所有的索引都不能访问它(因为它们都和聚集索引有关)。

5. REORGANIZE

从开发人员的角度看，这个选项真太棒了!!! 有了 REORGANIZE 就仿佛找到了非常适当的方法。重新组织索引的时候，得到了比彻底重建索引的全面性略逊一筹的优化方案，可是这种方法可以联机进行(用户仍然能使用索引)。

如果留意的话，上面的说法可能会让你产生一个问题，“前面说全面性‘略逊一筹’到底是指什么？”其实是这样的，REORGANIZE 只对索引的叶级起作用，对于非叶级不予理会。这意味着没有完全获得彻底的优化。但是，对于大部分的索引而言，那不是碎片真正开销的所在(尽管可能会发生，而且各人遭遇的事情也可能不尽相同)。

由于它对用户的影响非常小，通常会想要把该工具作为常规维护计划的一部分来使用。后面讨论碎片时，会更深入地考查它。

6.3.4 DROP INDEX

这条语句恢复了前面的 DROP 语句大部分的简洁性。关于它唯一真正麻烦的是，由于索引不是一个孤立的对象(本质上，它包含在表的定义当中)。所以删除索引时，不但必须指定索引，而且还要指定索引所属的表。其语法如下所示：

```
DROP INDEX <table name>.<index name>
```

正如你所看到的，这里其实没有太多可讲的内容。如果需要，可以使用四部分命名(如果包含索引的话，我想应该是五部分)。

6.4 明智地决定何时何地地使用何种索引

到现在，你心中可能会想，“哎呀，我总是准备创建聚集索引！”这样想有很多很好的理由。只是要知道，同样还有一些理由不这样想。

选择包含什么索引以及不包含什么索引可能是个困难的过程，而且，如果那样做还不够，那么必须做一些关于想让它们成为什么类型的决定。由于只能有一个聚集索引，因此后者的决定同时变得更简单也更难。这说明必须明智地进行选择，以便最有效地使用它。

6.4.1 选择性

索引，特别是非聚集索引，主要的好处是在某些情况下索引中有相当高级别的选择性。这里的选择性是指列中为唯一值的百分比。列中唯一值的百分比越高，就认为选择性也越高，因而索引的好处就越大。

回顾前面关于非聚集索引的小节(特别是关于聚集索引上的非聚集索引的部分)，会想起在非聚集索引中的查找其实只是开始。要找到真正的数据，还需要再一次浏览聚集索引。即使使用堆上的非聚集索引，仍然还需要执行多个单独的物理读操作。

如果非聚集索引中的查找将产生聚集索引中的多个另外的查找，那么，进行表扫描可能会更好。事实上，这里可能产生的作用效果很多，这非常令人吃惊。只有当被索引列中的唯一性约为90-95%时，非聚集索引所创造的循环过程才是值得的。

这对于聚集索引的影响微乎其微，因为一旦位于数据(唯一的或不唯一的)范围的起点，就已经到达目的地了。这里不必再读额外的索引页。另外，聚集索引所拥有的其他事物很可能更有用处。

提示：

选择性规则的一个例外与外键有关。如果表中的一列是外键，那么，在该列上有一个索引，这十有八九是有益的。为什么是外键而非其他列呢？外键通常是它们引用的表要联结的目标。无论索引的选择性如何，在联结的执行中它都是非常帮助的，因为它们允许合并联结(merge join)。合并联结从每一个表中获取一行并进行比较，以确定它们是否满足联结条件(在什么上进行联结)。由于在两个表的相关列上有索引，因而查找两个行会非常快。

这里的要点是，虽然选择性不是一切，但它是需要考虑的大问题。如果讨论中的列不是外键，那么，在你必须考虑的问题中，选择性几乎仅次于“这将用得有多频繁？”这样的问题。

6.4.2 注意代价

要记住，虽然索引在读取数据时加快了执行速度，但事实上在修改数据时，它们的开销非常大。索引不是靠魔法来维护的。每对数据修改一次，就必须更新任何与数据相关的索引。

当插入一个新行时，必须在表中所有的索引上加入新的条目。另外，也要记住，在更新行时，是作为删除和插入来处理的，所以需要再次更新索引。但是，等等！还有更多！（感觉这里就像深夜播出的电视购物节目）。删除记录的时候，同样也必须更新所有的索引，而不仅仅只涉及数据。每创建一个索引，就又创建了一组必须更新的条目。

顺便说一下，我前面提及条目时使用的是复数形式——不是单个。记得B树有许多级。每当在叶级进行修改时，可能会发生页拆分，这样一来，为了引用到正确的叶级页，也必须修改一个或多个合适的非叶级页。

有时(实际上很经常)要做的事情是不创建额外的索引。有时，最好是根据对系统至关重要的事务选择索引，并使用讨论中的表。用于这个事务的代码中有WHERE子句吗？它使用了什么列？这里需要排序吗？

6.4.3 选择聚集索引

记住，只能有一个聚集索引，因此必须明智地选择。

在默认情况下，主键与聚集索引一起创建。通常，这是聚集索引最好的地方，但并不总是这样(甚至在一些情况下，这会对你极其不利)，如果你顺其自然，则无法在其他任何地方使用聚集索引。此处的要点是，不要这样接受默认安排。在定义主键时请仔细考虑一下：真的想要它成为聚集索引吗？

如果决定确实不想接受默认安排(即不希望声明为聚集索引)，只需在创建表时添加 `NONCLUSTERED` 关键字即可。例如：

```
CREATE TABLE MyTableKeyExample
(
    Column1 int IDENTITY
        PRIMARY KEY NONCLUSTERED,
    Column2 int
)
```

一旦创建了索引，改变它的唯一办法是删除并重建它，因此，希望事情从一开始就是正确的。

要记住，如果修改了聚集索引所在的列，SQL Server 将必须对整个表进行彻底的重新排序(记住对于聚集索引而言，表的排序顺序与索引的顺序相同)。现在，考虑一个有 5 000 字符宽的表，该表有一百万行。这是必须要进行重新排序的庞大的数据。从这里应当想到下面几个问题：

- 这会耗时多久？时间可能会很长，而且，实在没有一个好的方法来估计这一时间。
- 有足够的空间吗？为了在聚集索引上重新排序，平均来说，另外需要的空间是表实际占用空间的 1.2 倍(运行空间加上新的索引)。在处理一个大型的表时，这将会是非常巨大的空间。确定你有足够的空间来进行重新排序。顺便说一下，所有这些活动都将会自己在数据库中发生。因此，所需要的空间还会受到为数据库设置的最大大小和增长选项的影响。
- 应该使用 `SORT_IN_TEMPDB` 选项吗？如果 `tempdb` 单独位于与数据库不同的物理阵列上，并且有足够的空间，那么答案很可能是肯定的。

1. 赞成

相关列经常作为范围查询的对象时，聚集索引对于这样的查询是很有用的。这类查询以使用 `BETWEEN` 语句或者 `<or>` 符号为代表。使用了 `GROUP BY` 并利用了 `MAX`、`MIN` 和 `COUNT` 聚集的查询，也是使用范围并偏好聚集索引查询的重要例子。聚集在这里非常恰当，因为搜索可以直接到达物理数据中的特定位置，一直读取数据，直至到达范围的末尾，然后停止。它非常高效。

当想要数据基于聚集键排序时(使用 `ORDER BY`)，聚集的表现也十分出色。

2. 反对

在两种情况下不希望创建聚集索引。第一种非常显而易见——当有更好的地方使用聚集索引时。是的，听上去我有些重复，但不要因为列看上去该使用聚集索引就使用聚集索引(主键一般是罪魁祸首)。要先确定是否有另外的更适合的列。

尽管如此，或许关于聚集索引更大的使用禁忌是：如果将要以不连续的顺序进行大量的插入，请不要使用。记得页拆分概念吗？是的，这里会发生页拆分并消耗大量的时间。

设想一下这样的场景：你正在创建帐目清算系统。想要在交易文件中使用交易号码的概念作为主键，但也想要交易号码能反应出交易的类型(它对于会计人员的排错确实很有帮助)。那么，你有了一种算是方案的想法：在所有的事务上添加前缀，以表明它们源自什么子系统。它们看起来像下面这样：

```
ARXXXXXX    Accounts Receivable Transactions
GLXXXXXX    General Ledger Transactions
APXXXXXX    Accounts Payable Transactions
```

其中 XXXXXX 是连续的数值。

这好像是个不错的主意，所以你开始实现它，将主键默认设为聚集索引。

乍一看，这样的设置似乎很好。你将会有唯一的值，会计人员可以基于交易号码推断出交易来自何处，他们也会很满意。由于经常使用聚集索引查询交易号码的范围，看起来聚集索引是有意义的。

如果真的只是那样简单就好了。考虑一下进行插入的情形。使用聚集索引时，最初有一个良好的机制避免大量页拆分的开销。当插入一条新记录时，将把它接在表中最后一条记录的后面，那么，即使发生了页拆分，也只是让新插入的记录插入新的页中，SQL Server 不会试图移动旧的数据。不过现在，却陷入到麻烦中了。

从总分类帐(General Ledger)中插入的记录还好，将接续在表的最后(按字母顺序 GL 在后面，而且号码是连续的)。可是，AR 和 AP 交易则有大问题：要进行不连续插入。当插入的 AP000025 页上没有空间时，SQL Server 将在表中发现 AR000001 并知道这不是连续的插入。在插入 AP000025 前，原来的页中一半的记录将被复制到新页上。

这样做产生的开销可能是惊人的。要记住正在处理的是聚集索引，而聚集索引就是数据。数据的顺序是索引的顺序。这意味着，当把索引移动到新页时，也在移动数据。现在，设想一下在典型的 OLTP 环境中运行这样的帐目清算系统(再没有比帐目清算系统更具有 OLTP 特征的了)，该环境中有一群数据录入员正在尽可能快地输入供应商发票或客户订单。你的系统将会经常发生页拆分，每次发生页拆分时，该表的用户将在系统移动数据时出现短暂的停滞。

幸运的是，有两种方法可以避免这种情形：

- 选择在进行插入时是连续的聚集键。可以为此创建标识列，或者也可以使用另一个对任何输入交易(无论来自那个系统)其逻辑上都是连续的列。
- 选择在该表上不使用聚集索引。对于类似本例的情形，这常常是最好的选择，因为在堆上的非聚集索引中插入比聚集键上插入快。

提示：

尽管前面说过，为避免页拆分，倾向于使用连续的聚集键，但必须意识到这里也要付出代价。连续聚集键的弊端之一是并发性(两个或更多的人试图在同一时间获得同样的对象)。在需要什么、正在做什么以及这样做在别的地方付出的代价是什么这个问题上，全都是要权衡的。

为什么要如此深入地研究有关事情是如何进行的，这可能是最好的例子之一。在明白要使用(或不使用)的正确的索引是什么之前，需要思考事情真正是怎样完成的。

6.4.4 列排序问题

只因为索引中有两个列，所以这不意味着索引对所有引用任一列的查询都是有用的。

仅当查询中使用了索引中第一个列出的列时，才考虑使用索引。好的一面是，不必在所有的列上正好一对一地匹配——只需第一个列匹配即可。当然，匹配的列越多(按顺序)越好，但是只需第一列就能产生明确的“请别使用”的情况。

按照这样的方式思考问题，假设在使用电话簿，所有的条目都按照先姓氏后名字的方式索引。如果只知道要通电话的人的名字是 Fred，这样的排序方式能为你带来什么好处吗？另一方面，如果只知道他的姓是 Blake，索引能为你缩小查找范围。

一种在索引构建上最常犯的错误是，认为一个包含所有列的索引将对任何情况都有帮助。事实上，这样做只是把所有的数据再存储了一次而已。如果在查询的 JOIN、ORDER BY 或 WHERE 子句中没有提及索引中的第一个列，那么就会完全忽略索引。

6.4.5 删除索引

如果不断地重新分析情况并添加索引，也请别忘记删除索引。要记住插入时的开销。如果只考虑需要的索引，而不考虑哪些索引是不需要的，那将没有多大意义。你要一直问自己：“我能从这些索引中去除掉哪些索引？”

删除索引的语法与删除表的非常相似。唯一的不同是这里需要用索引所附着的表或视图来限定索引名：

```
DROP INDEX <table or view name>.<index name>
```

运行该语句即可删除索引。

6.4.6 使用数据库引擎优化顾问

希望你对于索引的了解已经足够多，可以不需要使用数据库引擎优化顾问，尽管如此，它还是非常便利的。它的运行方式如下：取得一个工作负荷文件来考查信息，以确定什么索引对系统最有利，工作负荷文件是使用 SQL Server Profiler(将在第 22 章中讨论)生成的。

可以在 SQL Server Management Studio 的“工具”菜单中找到“数据库引擎优化顾问”。也可以从 Windows 的“开始”菜单中通过一个单独的程序项来访问它(在“Microsoft SQL Server 2008”|“Performance Tools”下)。像使用大多数其他性能调试工具一样，不建议把使用该工具作为决定创建什么索引的唯一方法，但是，它可以在你可能考虑不到的地方提出建议，在这一点上它是非常便利的。

6.5 维护索引

作为开发人员，在一个产品交付之后，我们常常会倾向于忘记它。对于许多类型的软件来说，就这样忘掉也没什么不好——你售出它，然后继续开发下一个产品或下一个版本。不过，对于数据库驱动的项目来说，实际上不可能就这样摆脱它。在交付日期之后，还必须担负起让产品工作

良好的责任。

请不要以为我的意思是你必须在技术支持部门分担一份工作。其实我讨论的是更重要的事情：维护计划。

实际上，在索引维护方面需要处理两个问题：

- 页拆分；
- 碎片。

这两个问题都与页上的信息密度有关，虽然二者的症状极为不同，但故障排除工具是相同的，处理方式也是相同的。

6.5.1 碎片

前面已经讨论过很多关于页拆分的内容，但实际上还不曾谈到碎片。这里谈论的不是你可能听说过的操作系统文件的碎片以及所使用过的碎片整理工具，因为它们对数据库碎片没有什么帮助。

在数据库增长、页拆分然后最终删除数据时，会产生碎片。虽然从增长的视角看，B 树机制在保持平衡上确实没那么差。但是，在删除数据时，它的确没做多少贡献。最后，可能会陷入这样的情况：这一页上有一条记录，那一页上有几条记录——在这种情况下，许多数据页上的数据量是它们可以保存的总数据量的一小部分。

或许，关于碎片首先会想到的第一个问题是——浪费的空间。记得前面说过，SQL Server 一次分配一个区段的空间。只要一页上有一条记录，那么，整个区段仍然被分配了。当区段中有空页时，SQL Server 把这些页视为可以在同一个表或索引中重新使用，但是，如果那个表或索引的大小减小了，那么将保持不使用区段中空闲的页。

第二个问题是碎片很容易带来麻烦——记录散布在各处会在检索数据时带来额外的开销。SQL Server 为了获取 10 行记录可能必须载入 10 个不同的页，而不是只载入一个页来获取相同的信息。并不只是读取行会产生这种结果——SQL Server 必须先读取该页。更多的页等于更多的读取工作量。

话虽如此，数据库碎片也的确有其好的一面。OLTP 系统显然偏好碎片。其原因何在？页拆分。没有多少数据的页可以在插入数据时几乎或者完全不担心页拆分。

因此，高碎片等于低读取性能，但也等于卓越的插入性能。正如你可以想见的，这意味着 OLAP 系统的确不喜欢碎片，但是 OLTP 系统却很喜欢。

6.5.2 检测碎片

SQL Server 一直有一条命令帮助确定数据库中的页和区有多满。在 SQL Server 2005 中，微软极大地扩展了这一选项，特别是扩展了用于索引的管理工具的可用性。而且进入 SQL Server 2008 的时代之后，这些增加的选项继续变成主流选项而且慢慢变得不太关注与 SQL Server 2000 及之前版本的兼容性了。我们可以利用由这些命令和工具提供的信息，在需要做些什么来维护数据库的问题上进行决策。

1. sys.dm_db_index_physical_stats

sys.dm_db_index_physical_stats 函数是一种重新加入 SQL Server 2005 的元数据函数(附录 B 讨

论了这些函数)。这些元数据函数和其他类似元数据函数背后的思想是允许开发人员和管理员更灵活地访问服务器、数据库、表和表内部的索引上的数据。数据库一致性检验程序(DBCC)中不同函数会给出非常灵活的输出结果,但我们难以使用编程方式利用这些结果(你可能一直坚持对结果进行解析以便于找到你所需要的信息)。我们现在有了标量函数和表值函数,它们会向我们返回可用的数据,我们可以利用这些数据创造环境,获取变量中的数值,以及将其作为离散数据片断进行其他操作。谈到索引,我们最感兴趣的元数据函数是 `sys.dm_db_index_physical_stats`。这是一个需要多个参数的表值函数,其语法如下所示:

```
sys.dm_db_index_physical_stats (
    { <database id> | NULL | 0 | DEFAULT }
    , { <object id> | NULL | 0 | DEFAULT }
    , { <index id> | NULL | 0 | -1 | DEFAULT }
    , { <partition number> | NULL | 0 | DEFAULT }
    , { LIMITED | SAMPLED | DETAILED | NULL | DEFAULT }
)
```

再说一遍,这是一个表值函数。所以你必须将它和一条 SELECT 语句联合使用。表 6-2 逐一列出了输入参数。

表 6-2

参 数	描 述
Database ID	SQL Server 的内部标识符,表示的数据库包含了物理统计数据针对表和索引。使用 <code>DB_ID()</code> 函数可以方便地检索自己的数据库 ID。默认参数是 NULL(在技术上等同于使用的是 0),这意味着它为所有的数据库提供信息
Object ID	物理统计数据针对的特定对象的内部标识符。使用 <code>OBJECT_ID()</code> 函数可以方便地检索你感兴趣的表或视图的对象 id。默认值为 NULL(它在功能上等价于 0),它意味着你要得到数据库中所有对象的数据
Index ID	你对其物理统计数据感兴趣的特定索引的内部标识符。取回特定索引的标识符很有挑战性,因为没有系统函数取回它(你必须使用它所属的名称和对象 id 从 <code>sys.indexes</code> 中查询)。与前面的其他参数类似,目前的默认值是 -1。不同于之前的参数,它在功能上不等于 0(只有在堆上建立表的时候才是有效的,而且这意味着你需要堆上的数据)
分区号	对绝大多数的表来说,只有一个分区(因此它的号码是 1)。默认值为 NULL,会返回所有的分区而且在功能上等于 0
模式	确定要完成的扫描等级,以确定要返回的统计值。扫描模式包括 LIMITED、SAMPLED 和 DETAILED,按照这种排序方式准确度递增,不过要花费的开销也递增(但是响应更慢)

返回的是一个很大的表,带有索引或表的一组不同物理统计值。我们不能一次性解决所有问题,不过可以看一看表 6-3 中列出的一些要点:

表 6-3

列	描 述
Index_type_desc	表示这一行相关的索引。如果结果是 HEAP 或 CLUSTERED INDEX, 那么它与表的物理数据有关。其他可能的结果包括 NONCLUSTERED INDEX、PRIMARY XML INDEX、XML INDEX 和 SPATIAL INDEX
Index_depth	索引的级数。如果它是堆或一组 LOB 页, 那么这个值始终是 1, 否则, 它会代表索引中有多少级(例如, 回到图 8-7, 非聚集索引有三个层次)
Index_level	这个有点不太直观, 因为它是从索引的底部开始数的。索引的叶级是 0(堆或 LOB 也是 0), 沿着树向上数, 这个数字会增加。只有在模式为 DETAILED 的时候, 才会提供这个值
Avg_fragmentation_in_Percent	这表示基于无序页或区段的索引树的碎片等级(逻辑下一页的指针与物理下一页的指针不同)。虽然究竟有多低还要取决于行的构成细节和索引的目的, 一般你在这里要查找的一个低的号码
Avg_record_Size_in_bytes	名副其实。索引中记录的平均大小。在进行空间规划的时候, 这个数字非常有用(如果另外加 100 000 行, 它又要占多少空间呢?)
Record_count	这又是一个难题。这个值一般会和你要从 SELECT COUNT(*)中得到的值一致。在处理一个有前向记录的堆的时候会出现例外情况。在向页写入记录的时候会出现前向记录, 随后对一个给定的不再适合这个页的列进行升级(这样它们就存储了数据的位置指针)

下面看一下一个使用这个系统函数的简单例子。想象一下我们要看到 Sales.SalesOrderDetail 表的聚集索引分段信息。我们可以用下面的查询语句获得一些关键信息:

```
SELECT index_type_desc AS Type,
       index_id,
       avg_fragmentation_in_percent,
       forwarded_record_count
FROM sys.dm_db_index_physical_stats(
    DB_ID(),
    OBJECT_ID('Sales.SalesOrderDetail'),
    DEFAULT,
    DEFAULT,
    'DETAILED' )
WHERE index_id = 1
      AND index_level = 0;
```

生成一个相当简单的结果集:

```
Type          index_id avg_fragmentation_in_percent forwarded_record_count
-----
CLUSTERED INDEX 1          0.0810372771474878          NULL

(1 row(s) affected)
```


注意，为了强迫它必须是一个聚集索引，我的 WHERE 子句中已经使用了 index_id=1。(如果在堆上，我会选择 0)。我还选择了 index_level=0，以强迫它提供索引的页级信息。

通过在碎片百分比上放置额外的 WHERE 条件，我会使用这里提供的信息建立一个需要维护的索引的列表(更多的细节请参考第 23 章)

2. 后向兼容性

我们现在看到了获得信息的元数据方式，不过我们利用的是旧版本(2005 之前的版本)那应该怎么办呢？“旧版本备用(old Standby)”命令实际上是一个 DBCC 选项。现在和今后每次安装都会以某种形式利用这条命令。这是 2005 版之前解决问题的方法，要维护的 2005 版之前的所有数据库安装都会利用它。甚至，还有成千上万的文章和网页上的“如何做”之类的文章告诉你这种工具的使用方式。

提示：

在我盛赞 DBCC SHOWCONTIG 之前，我要再提醒你一下这是一种“老”方法，而且我可以说这是一种“不灵活”的方法。系统视图为我们提供了更多的可能性，可以更有针对性地查询数据并在更加全局的层次上管理索引。本书后的附录 B 对这种功能进行了更深入的研究。我已经说过了，DBCC 多年前就已经完成了这项工作，如果你要对包含 SQL Server 2005 预安装的服务器环境中的索引进行监控，那么就该使用它，而且你可以在现有的管理代码中找到它。

语法非常简单：

```
DBCC SHOWCONTIG
    [({<table name>|<table id>|<view name>|<view id>}
    [, <index name>|<index id>})]
    [WITH { [ ALL_INDEXES ]
    | [, FAST ]
    | [, TABLERESULTS ]
    | [, ALL_LEVELS ] ]
    | [, NO_INFOMSGS ]
```

有些东西是不言自明的(例如表名)。不过，我想要说明名字之外的一些项，如表 6-4 所示：

表 6-4

Table id/view id/index id	这是表、视图或索引的内部对象 id。在较早的 SQL Server 版本中，DBCC SHOWCONTIG 完全使用标识符，由此在调用 DBCC 之前，必须通过 OBJECT_ID() 函数进行查找
ALL_INDEXES	这是一种“像它听上去那样”的事物。如果指定该选项，就会忽略提供指定的索引，因为它会分析所有的索引并返回数据
FAST	这是关于尽可能快地返回结果的选项，因而它将不分析索引的实际页，并且只输出最少的信息
TABLERESULTS	这个功能非常棒——它把结果作为表返回而非文本，这意味着分析结果和使用自动化操作将更加容易

(续表)

ALL_LEVELS	这个选项与 SQL Server 2005 中的选项只有一个相关性，因为它过去做的事情现在已经忽略了。相关性是向后兼容的。基本上，可以包含该选项，此时命令仍将执行，只是不会有任何不同
NO_INFOMSGS	这个选项修剪信息性消息。基本上，当表中有任何重大的错误时(错误严重级别为 11 或更高)，仍会传出消息，但错误级别 10 或更低的消息将被排除

作为一个例子，为了再次从 Sales.SalesOrderDetail 表的 PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID 索引中获得信息，应该运行：

```
USE AdventureWorks2008;
GO

DBCC SHOWCONTIG ('Sales.SalesOrderDetail',
    PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID);
```

提示：

注意包围着表名的单引号。因为使用了两部分命名，所以才必须这样做——如果只指定表名 (SalesOrderDetail)，那么不需要引号。这里的问题是，省去模式名可能会产生错误，或者在不同于你预期的表上执行操作，这取决于在使用不同模式时或在不同模式中存在同名的其他表时，用户是如何设置的。

输出实际上不全是自描述的：

```
DBCC SHOWCONTIG scanning 'SalesOrderDetail' table...
Table: 'SalesOrderDetail' (898102240); index ID: 1, database ID: 7
TABLE level scan performed.
- Pages Scanned.....: 1234
- Extents Scanned.....: 155
- Extent Switches.....: 154
- Avg. Pages per Extent.....: 8.0
- Scan Density [Best Count:Actual Count].....: 100.00% [155:155]
- Logical Scan Fragmentation .....: 0.08%
- Extent Scan Fragmentation .....: 3.23%
- Avg. Bytes Free per Page.....: 28.5
- Avg. Page Density (full).....: 99.65%
DBCC execution completed. If DBCC printed error messages, contact your
system administrator.
```

其中的一些含义可能非常简单明了，不过表 6-5 会带领你了解所有的输出。

表 6-5

统 计 数 据	说 明
扫描页数(Pages Scanned)	表(用于聚集索引)或索引中的页数
扫描区段数 (Extents Scanned)	表或索引中的区段数。页数的最小值除以 8，然后取整。区段数越多，碎片就越多

(续表)

统 计 数 据	说 明
区 段 切 换 次 数 (Extent Switches)	遍历表或索引的页时, DBCC 从一个区段移动到另一个区段的次数。这是另一个说明碎片状况的信息——为了看到同样数目的页, 所做的切换越多, 碎片也越多
每个区段的平均页数(Avg. Pagesper Extent)	每个区段的平均页数。完全占用的区段有 8 个页
Scm Deusity[Best Count: Actual Count]	最佳计数是指在一切都完美链接的情况下, 区更改的理想数目。实际计数是指区段更改的实际数目。扫描密度是最佳计数与实际计数相除的百分比
逻辑扫描碎片(Logical ScanFragmentation)	扫描索引的叶级页时检查到的无序页的百分比。它只与聚集表上的扫描有关。对于无序页而言, 在索引分配映射(IAM)中指示的下一页与叶级页中下一页指针指向的页不同
区段扫描碎片(Extent Scan Fragmentation)	该信息表示, 一个区段在物理上的下一个区段是否是它在逻辑上的下一个区段。这只是说明索引的叶级页在物理上是无序的(尽管它们在逻辑上是正常的), 并给出问题区段所占的百分比
页中平均可用字节数(Avg. Bytes free per page)	扫描的页上平均可用的字节数。如果行比较大的话, 该数值可能虚高。例如, 假设行的大小是 4040 字节, 那么每页只能保存一行, 因而平均可用字节数将总是 4020 字节左右。这个数值看似很多, 但考虑到行大小, 它不能比这更低
页的平均密度(全部)(Avg. Page density (full))	页的平均密度(以百分比表示)。该值会考虑行的大小, 因而可以更准确地指示页的满度。百分比越高越好

现在的问题是, 我们一旦得到这些信息, 该如何使用呢? 答案自然是视情况而定。

使用 SHOWCONTIG 的输出, 可以很好地认识到数据库是否满、是否碎片化或者介于两者中间(后者很可能是我们想看到的)。如果运行的是 OLAP 系统, 那么将愿意看到页较满, 因为碎片化会让人沮丧。而对于 OLTP 系统, 我们需要的结果完全相反(尽管只在一定程度上如此)。

那么, 如何解决这一问题呢? 要回答它, 必须了解重新生成索引和填充因子的概念。

3. DBREINDEX——维护索引的另一种方法

在本章前面讨论过 ALTER INDEX 命令。这是你用来执行索引的重新组织和管理碎片化程度的第一条命令。虽然我强烈推荐使用 ALTER INDEX, 但 DBREINDEX 曾经是过去一直使用的方法, 而且与 DBCC SHOWCONTIG 类似, 这里已经忽略了太多的代码和用法。

DBRE INDEX 是另一条 DBCC 命令, 其语法如下所示:

```
DBCC DBREINDEX (<'database.owner.table_name'>[, <index name>
[, <fillfactor>]]) [WITH NO_INFOMSGS]
```

执行这条命令会重新生成被请求的索引。如果提供的是不带索引名的表名, 则会重新生成该表所有的索引。没有单独的命令来重新生成数据库中的所有索引。

重新生成索引将重新构造索引中的所有信息, 并重建页填充度的基础百分比。如果讨论中的索引是聚集索引, 那么也将会重新组织物理数据。

像使用 ALTER INDEX 一样,在默认情况下,会将页重建为差两条记录填满。正像使用 CREATE TABLE 语法一样,可以把 FILLFACTOR 设置为 0 到 100 之间的任何值。这个值是完成数据库重新组织时的页填充度百分比。记得之前说过,在页拆分时,数据会在两页间平分,但除了定期重新生成索引外,不能在动态的基础上控制填充度百分比。

注意:

如果使用 0 作为要填充的百分比,与填充比例匹配的数字就会出现例外。它将是一页差两行(带有一定的欺骗性——你不这么认为吗?)。

在需要调整页密度时,可以使用 FILLFACTOR。前面已经讨论过,较低的页密度(因而具有较低的 FILLFACTOR)对于有很多插入操作的 OLTP 系统是理想的——这有助于避免页拆分。较高的页密度是 OLAP 系统必需的(要读取的页较少,因为极少或几乎没有插入,所以实际上没有页拆分的风险)

对于前面查看过的 Order Detail 表,如果想要以填充因子 65 重新生成在该表中作为主键的索引,可以发出下面的 DBCC 命令:

```
DBCC DBREINDEX ('Sales.SalesOrderDetail',
    PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID, 65)
```

然后再次运行 DBCC SHOWCONTIG,以便观察效果:

```
DBCC SHOWCONTIG scanning 'SalesOrderDetail' table...
Table: 'SalesOrderDetail' (898102240); index ID: 1, database ID: 7
TABLE level scan performed.
- Pages Scanned.....: 1883
- Extents Scanned.....: 236
- Extent Switches.....: 235
- Avg. Pages per Extent.....: 8.0
- Scan Density [Best Count:Actual Count].....: 100.00% [236:236]
- Logical Scan Fragmentation .....: 0.05%
- Extent Scan Fragmentation .....: 1.27%
- Avg. Bytes Free per Page.....: 2809.1
- Avg. Page Density (full).....: 65.29%
DBCC execution completed. If DBCC printed error messages, contact your
system administrator.
```

这里需重点注意平均页密度(Avg.Page Density)上的变化。由于 SQL Server 必须处理页和行的大小问题,所以这个值并非完全等于 65%,但它会尽可能地接近。

关于 DBREINDEX 和 FILLFACTOR 要注意如下几件事情:

- 如果没有提供 FILLFACTOR,那么 DBREINDEX 会使用以前创建索引时设置的填充因子。如果从未指定过填充因子,那么就会把页填充至差两条记录满为止(这种填充度对大多数情况来说都太满了)。
- 如果提供了 FILLFACTOR,那么这个值将成为相应索引的默认 FILLFACTOR。
- 虽然可以当场使用 DBREINDEX,但我强烈建议不要使用它——它会锁定资源并带来许多问题。最起码应考虑在非高峰时段使用。如果要在联机的情况下完成该任务,更好的方法是使用 ALTER INDEX,且只进行 REORGANIZE 而不是重新生成。

- 关于这一点，我以前曾经说过，不过需要再重复一次：现在不建议使用 DBREINDEX，在不需要向后兼容的情况下，应当避免使用它(请改用 ALTER INDEX)。

6.6 小结

在 SQL Server 或任何其他数据库环境中，索引是一种基础性的主题，因而绝不要轻视它。它们既能导致优良的性能，也能导致糟糕的性能。

关于索引要考虑的首要问题是：

- 聚集索引通常比非聚集索引要快(通常可以这么说，不过也有例外的情况)；
- 只在将获得高级别选择性(95%或更多的行是唯一的)的列上放置非聚集索引；
- 索引可以为所有的数据操作语言(DML: INSERT、UPDATE、DELETE、SELECT)语句带来好处，不过索引会降低插入、删除和更新(记住，它们使用一种删除或插入的方法)的速度。索引可以帮助查询的查找部分，不过所有对数据的修改都需要完成额外的操作(用于维护索引和实际数据)。
- 索引会占用空间；
- 只有当查询与索引的第一列相关时，才会使用索引；
- 索引既能有助益，也能带来麻烦——要了解为什么要创建索引，并且不创建不需要的索引；
- 索引能为非结构化的 XML 数据提供结构化的数据性能，不过要记住，类似其他索引，这会涉及开销的问题。

在考虑索引的时候，问一下自己表 6-6 中列出的这些问题：

表 6-6

问 题	答 案
是否要对表做很多修改或者插入很多内容？	如果是的话，最好少用索引。这种类型的表通常是在主键的单个记录查找中修改的——通常，这是你唯一希望的表索引。 如果插入操作是不连续的，那么，考虑不使用聚集索引
这是一个报表吗？也就是说，这里没有多少插入操作，但会以多种不同的方法运行报表吗？	应使用较多的索引。把聚集索引确定在经常使用的可能会在范围中提取的信息上。OLAP 安装中的索引数目常常是 OLTP 环境中的数倍
数据上有高级别的选择性吗？	如果是，并且它常常是 WHERE 子句的目标，那么添加索引
已经删除了不再需要的索引了吗？	如果没有，为什么不删除？
我是否已经确立了维护策略呢？	如果没有，为什么不这样做呢？

第 7 章

更高级的索引结构

之前已经介绍了设计的基本概念。我们甚至已经看过了高级的传统索引。不过在索引和其他存储中还有一些更加高级的事物要思考。它们之中有一些非典型的索引和存储结构：

- XML 索引
- 空间数据及其相关的索引
- 用户定义数据类型
- 文件流
- 表压缩
- 层次数据

在本章中，我们将看到所有的这些问题。其中一些问题构建在你已经知道的一些概念之上(类似我们已经广泛讨论过的 XML 数据类型和方法)，还有一些可能是全新的概念(确实，其余的项目对 SQL Server 2008 来说都很新)。

选择将这些具体内容放在一章之内介绍可能看起来有些不可思议(甚至对我来说也是这样)，不过它们的共同之处就是非常简单。它们都在主流之外，而且需要再思考一下才能知道它们的工作方式。

7.1 XML 索引

XML 索引第一次出现在 SQL Server 2005 中，而且必须承认我一直很惊讶微软会使用它。迄今为止，这个团队中的一些成员我已经认识很久了，而且我对他们很有信心，不过，虽然很多人尝试为类似 XML 的非结构化数据编制索引，但很少有人取得实质上的成功。引入这个应该为 SQL Server 小组带来了一定的声誉。尽管情绪过于激动，我还是想平静下来研究一下 XML 索引到底是什么。

也许最令人惊讶的是 XML 索引与更加典型的关系数据索引并非全无共同之处。实际上，XML CREATE 语法支持你在之前章节中 CREATE INDEX 语句看到的所有相同的选项，其中的 CREATE INDEX 语句带有 IGNORE_DUP_KEY 和 ONLINE 例外。这为什么是意义重大的？虽然索引可能是支持任何事物的基本结构，不过索引事物的本质可能会严重影响传统索引支持基本数据的方式。不同于你已经熟悉的关系数据，XML 倾向于非结构化。它利用标签识别数据，而且在本质上比典

型关系数据更加多变。XML 的非结构化本质需要“导航”或“路径”信息以便于在 XML 文档中查找一个数据“节点”。现在索引试图为数据提供特定的结构和顺序。这样会引发一定的冲突。

可以在 SQL Server 中类型为 XML 的列上创建索引。这样做需要：

- 包含你要索引的 XML 的表必须有一个聚集索引，而且这个聚集索引必须在表的主键上，因此，主键不能多于 15 列。
- 在创建“辅助”索引之前，XML 数据列上必须存在“主”XML 索引(很快就可以看到其中的细节)。
- 只能在 XML 类型的列上创建 XML 索引(而且 XML 索引是你可以在这种类型的列上创建的唯一一种索引)。
- XML 列必须是基表的一部分。不能在视图、表变量或表用户定义数据类型上创建索引。

在一个表上创建一个或多个 XML 索引也意味着对自己的表施加了一个严格的限制：在表上存在任何 XML 索引时，不能修改主键或(作为一个结果)聚集索引。如果要修改主键，必须首先删除所有的 XML 索引(在完成主键的修改之后，可以重建它们)。

7.1.1 主 XML 索引

在 XML 索引上创建的第一个索引必须被声明为“主”索引。在创建主索引的时候，SQL Server 将 XML “撕碎”(将其转变为表形式)并创建一个新的聚集索引，它将基表的聚集索引与指定的 XML 节点中的所有数据结合起来。除了聚集键信息，主 XML 索引还会存储下面的信息：

- 要索引的节点的标签名(它的元素或属性名)；
- 节点值；
- 节点类型(元素、属性或文本)；
- 内部节点标识符(排序信息)；
- 节点到文档根部的路径。

所有这些都是将 XML 分解为内部表的结果。XML 数据被保存在这个内部表中，而且允许使用传统的索引模型。可以通过查询 sys.internal_tables 或 sys.xml_indexes 查看系统中存储的内部表(也显示了其他类型的内部表)。例如，我们可以检查 AdventureWorks2008 数据库中的 XML 索引：

```
SELECT * FROM sys.xml_indexes;
```

这样会生成几个主 XML 索引和一些辅助 XML 索引(很快我们就会看到辅助 XML 索引)。

object_id	name	index_id	type
162099618	PXML_ProductModel_CatalogDescription	256000	3 ...
162099618	PXML_ProductModel_Instructions	256001	3 ...
270624007	PXML_Store_Demographics	256000	3 ...
1509580416	PXML_Person_AddContact	256000	3 ...
1509580416	PXML_Person_Demographics	256001	3 ...
1509580416	XMLPATH_Person_Demographics	256002	3 ...
1509580416	XMLPROPERTY_Person_Demographics	256003	3 ...
1509580416	XMLVALUE_Person_Demographics	256004	3 ...

(8 row(s) affected)

为了能在这里显示，右边的结果被截断了，不过如果你自己运行查询，就会看到列出的每个

XML 索引的更详细的信息。

我们随后会讨论分解过程，现在先暂时讨论一下辅助 XML 索引和它们与主键的差别。

7.1.2 辅助 XML 索引

类似于非聚集索引指向聚集索引的聚集键，辅助 XML 索引指向主 XML 索引的内部表中的不同列。辅助 XML 索引是独立的，而且比其依赖的任何主 XML 索引或其他任何索引更具有针对性。任何一个给定的列最多只能有 248 个辅助 XML 索引。

辅助 XML 索引是特殊的，因为它们具有三个不同的子类型：

- **PATH**：次索引类型主要根据基于路径的搜索准则提供快速访问。这种索引基于内部表的反路径和值。
- **VALUE**：正如名称所暗示的，这种索引类型提供的索引面向搜索某个特定节点值的场景。它可以被视为 **PATH** 次索引类型的反转，首先索引值，然后索引反路径。
- **PROPERTY**：类似于 **VALUE**，不过面向多值场景。

关于辅助 XML 索引要重点理解的是，你选择的辅助 XML 索引并不仅是针对你要索引的数据的，还针对你要对数据执行的某种类型的查询。

下面分别看一看这三种类型。

1. PATH XML 索引

第一种辅助 XML 索引类型针对的是那些基于特定路径的搜索。如果绝大多数的查询在自己的 **WHERE** 子句中包含了一个特定的路径，那么就应该使用 **PATH** 类型的次索引。虽然主 XML 索引对查找某个特定路径的搜索帮助很大(可能是通过 `.exist()`)，不过它还会引入识别 blob 信息(我们之前讨论的节点信息)的开销。作为一个次索引，基于 **PATH** 的索引的主要焦点是路径信息，而且更加简洁(因此在搜索的时候更加有效)。

要确保使用 **PATH** 索引的效率，最关键的就是要确保指定了某个特定的路径。你指定路径的 **Xpath** 可以包含某个值(如果你这样选择的话)，不过包含一个路径会导致使用这种索引。

例如，观察一下 **AdventureWorks2008** 数据库中的 **Person.Person** 表。我们可以对表的 **Demographics** 列发起一个相对直接的面向 **Xpath** 的查询：

```
WITH XMLNAMESPACES
  ('http://schemas.microsoft.com/sqlserver/2004/07/adventureworks/
  IndividualSurvey' AS "IS")

SELECT Demographics.query('
  /IS:IndividualSurvey/IS:TotalPurchaseYTD
') AS Result
FROM Person.Person
WHERE Demographics.exist ('/IS:IndividualSurvey/IS:TotalPurchaseYTD')=1;
```

搜索某个特定的路径会出现这样一种情况，它非常适合 **PATH** 次索引类型。要查看 SQL Server 是否确实在使用它，检查查询计划，如图 7-1 所示。

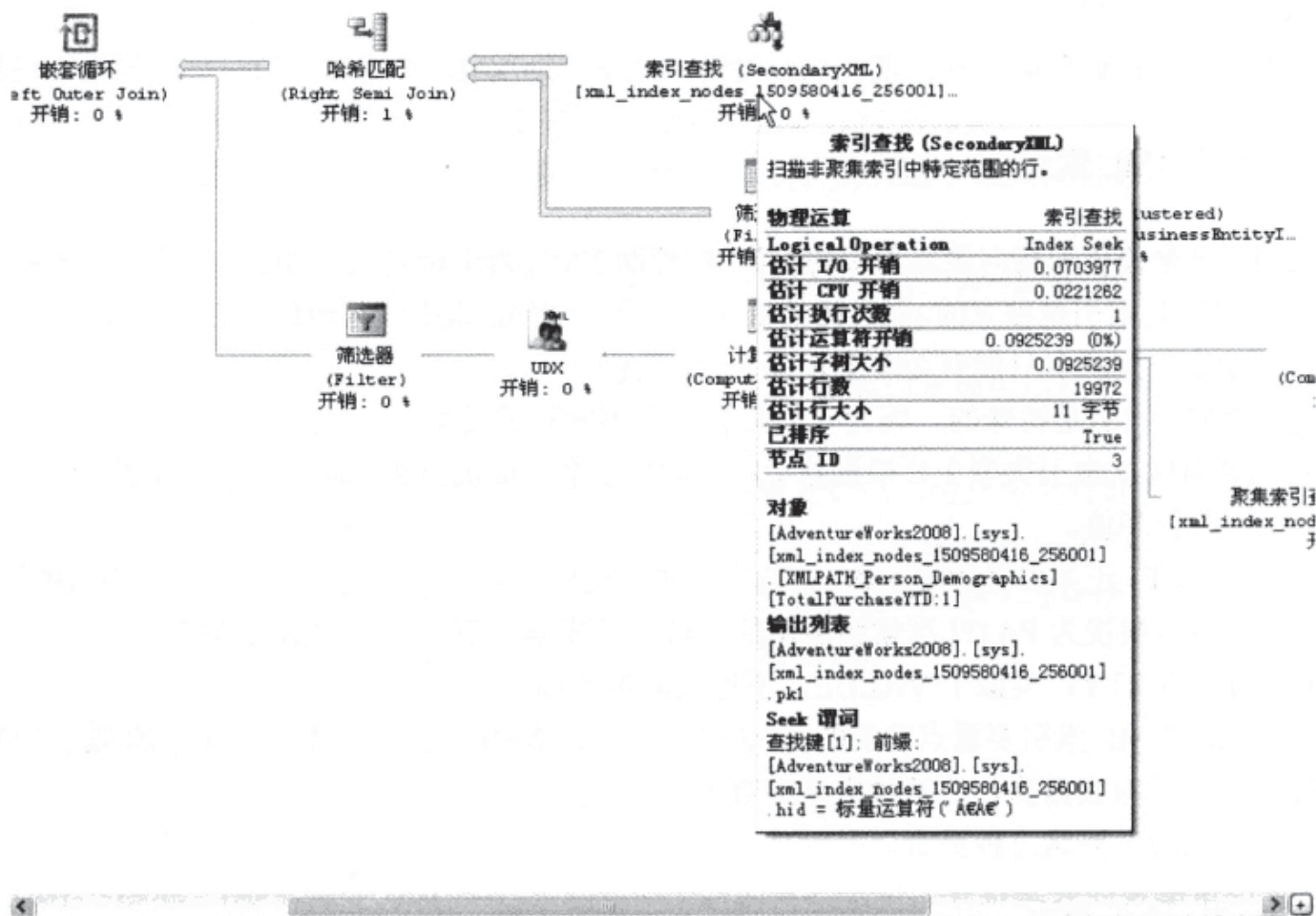


图 7-1

实际上，我们可以看到使用中的 XMLPATH_Person_Demographics 索引。

2. VALUE XML 索引

这是有关于按照值排序的。

这种索引使用的列都是基于主节点的值和路径的。值的类型不重要。重要的是要记住，在考虑这个问题的时候，你可能还不知道整个路径。特别是你可能只知道实际包含值的元素和(或)属性。

既然此类索引主要关注的是值，那么首先它会发现那里有匹配的值，然后再考虑自己是否与路径匹配。路径实际上是反向存储的，这样就允许你发现路径的叶级部分匹配(不考虑你提供的部分路径的父部分)。

3. PROPERTY XML 索引

PROPERTY 索引要将两种不同的列(不管是主键还是 XML 列)的值组合起来。PROPERTY 索引首先面向行的主键，然后再面向路径(再次声明一下，是反向存储的)和独立的 XML 节点值。你可能已经从该行的主键的第一个值中猜测到，这个索引只适用于主键已知的情况。此外，它的行为类似于 PATH 辅助索引类型。

7.1.3 创建 XML 索引

现在我们研究了所有不同类型的 XML 索引，你可能已经迫不及待地要了解创建它们的方法了。它的语法与创建标准的索引的语法差别不大，不过其中有一些调整。下面给出了整个语法的

模板:

```
CREATE [ PRIMARY ] XML INDEX <index name>
ON <table> (<name of the xml column to index> )
[ USING XML INDEX <name of primary xml index if creating a secondary>
  [ FOR { PATH | VALUE | PROPERTY } ] ]
[ WITH ( PAD_INDEX = { ON | OFF }
  | FILLFACTOR = <fill factor>
  | SORT_IN_TEMPDB = { ON | OFF }
  | IGNORE_DUP_KEY = OFF
  | STATISTICS_NORECOMPUTE = { ON | OFF }
  | DROP_EXISTING = { ON | OFF }
  | ONLINE = OFF
  | ALLOW_ROW_LOCKS = { ON | OFF }
  | ALLOW_PAGE_LOCKS = { ON | OFF }
  | MAXDOP = <max degree of parallelism>
  [ ,...n ]
) ] [ ; ]
```

注意 IGNORE_DUP_KEY 和 ONLINE 选项都只有一种设置。必须承认,我无法告诉你微软决定把它们放在那里的原因(或许是想与基本 CREATE INDEX 语句更趋一致,不过这样看起来还是很奇怪),不过它们现在确实在那儿(也许它们随后就有其他的选项了)。你可以看到,绝大多数的其他选项都是相同的,索引下面集中精力研究那些主要的语法项目。

首先,必须在 CREATE INDEX 行内通过 XML 关键字显式调用 XML 索引。只有主 XML 索引才需要 PRIMARY 关键字。否则 XML 索引只被假设为一个辅助索引。

接下来,要注意我们没有选项提供多列。相反,我们只命名自己计划索引的 xml 类型。

USING 子句和 PRIMARY 关键字是互斥的,而且只适用于辅助索引(而且在这种情况下必须这样)。使用这些带 FOR 关键字的子句可以表示你要创建的次索引类型(PATH、VALUE 或 PROPERTY)。

如果要使用上面的内容,我们可能要在 Production.ProductModel 表上创建一个主 XML 索引:

```
CREATE PRIMARY XML INDEX PXProductModelInstructions
ON Production.ProductModel (Instructions)
WITH (PAD_INDEX - OFF,
      SORT_IN_TEMPDB - OFF,
      DROP_EXISTING - OFF,
      ALLOW_ROW_LOCKS - ON,
      ALLOW_PAGE_LOCKS - ON
);
```

提示:

如果要运行之前的脚本,可能需要删除 AdventureWorks 2008 示例中已有的 XML 索引。

或者使用我们刚刚创建的主键来创建一个辅助索引,我们应该完成下面的操作:

```
CREATE XML INDEX SXProductModelInstructionsPATH
ON Production.ProductModel (Instructions)
USING XML INDEX PXProductModelInstructions
FOR PATH
WITH (PAD_INDEX - OFF,
      SORT_IN_TEMPDB - OFF,
```



```

DROP_EXISTING - OFF,
ALLOW_ROW_LOCKS - ON,
ALLOW_PAGE_LOCKS - ON
);

```

提示:

在 AdventureWorks2008 中, 之前的 CREATE XML INDEX 语句都会失败, 因为示例已经有一个默认的主 XML 索引了。如果你将 USING 子句修改为引用现有的主 XML 索引 (PXML_ProductModel_Instructions), 那么就会运行两个例子中的第二个例子。

再次说明, 它和标准索引的语法在本质上只有一些细微的差别。

7.2 用户定义的数据类型

用户定义数据类型(或 UDT)的潜力是非常惊人的。

这有点类似于经典的“先有鸡还是先有蛋?”的问题。将加入 .NET 对象视为一个可能的 UDT 来源应该就是 SQL Server 2005 之后 UDT 变得非常有趣的原因。不过, 出于不同的原因, 在涉及第 10 章将谈论的过程方面的内容前, 我不愿意讨论 .NET 的加入。

有了这些想法, 我决定做出一些让步。这里我们开始讨论 UDT, 并在第 10 章中讨论 .NET 方面(称之为两个不同的世界...).

在处理掉所有的组织性事物后, 下面开始看一看什么是真正的用户定义数据类型。如果你是一个真正的 SQL Server “专业人员”, 那么 UDT 的基本概念就不再新鲜了。毕竟, 在所有人没有思考过 .NET 之前, UDT 就已经是 SQL Server 的一部分了。在 .NET 时代之前, 它们的价值很小。即使是 .NET 到来之后, .NET 数据类型的灵活性需要高昂的复杂性作为代价, 而且需要你开启服务器配置中那些可能会违背某些安全性策略的设置(许多数据库管理员在为 SQL Server 启用 SQLCLR 和 .NET 时都会面临很多危险)。你可能完全忽略了 UDT, 但考虑到 SQL Server 2008 对 UDT 的修改, 现在的 UDT 还是值得一看的。

7.2.1 经典 UDT

经典的 UDT 建立在现有的数据类型之上。实际上, 它可以被视为 SQL Server 中已有基本类型的一个别名。在历史上, 它主要被用来保持频繁使用的属性的一致性, 或者令其符合规则和默认值(它可以直接被绑定在 UDT 上并适用于任何使用经典 UDT 的场景)。

下面从一个非常简单的例子开始——帐号。AdventureWorks2008 数据库利用一个名为 AccountNumber 的用户定义数据类型, 它是用 nvarchar 的基类创建的(此时, 是一个 nvarchar(15))。使用 AccountNumber 用户定义数据类型而不是直接使用 Nvarchar(15)基类可以保证要利用帐号概念的所有实例都保持一致性。

创建经典 UDT(源于内联数据类型的 UDT, 不过本质上不是表式的)的语法非常简单:

```

CREATE TYPE [<schema name>.<type name>
FROM <base type>
[ ( precision[, scale] ) ]
[ NULL | NOT NULL ]

```

因此，下面给出了 AdventureWorks2008 中使用的 AccountNumber 数据类型：

```
CREATE TYPE dbo.AccountNumber  
FROM nvarchar(15) NULL;
```

正如你看到的，为现有的对象创建一个简单的别名并不会花费很大气力。你可以看到这些通过使用规则和默认值扩展的相对简单的类型，不过我建议不要使用它，因为微软已经在之前的四次发布中说过，他们不赞成使用这些规则和默认值，而且迟早会把它从产品中除去。

提示：

可能还值得注意的是：该小组中的成员已经告知我微软现在越来越关注删除这些不赞成使用的特征之后要完成的事情。等到最终的 SQL Server 11(当前的代码为 Kilimanjaro)问世的时候，大家长久以来不赞成使用的一些特征会从产品中消失。

7.2.2 .NET UDT

在第 10 章全面研究基于.NET 的开发之前，我不会花太多的精力讨论基于.NET 的 UDT，不过为了给出相关的背景，这里还是会快速地了解一下基于.NET 的 UDT。

正如你可以猜到的，.NET UDT 使用.NET 程序集实现自定义数据类型。从.NET 第一次作为 SQL Server 2005 的一部分出现在 SQL Server 中开始，它们就已经被使用了，而且可以实现一些非常复杂的自定义类型。究竟有多么复杂？复杂到可以使用.NET 实现新的地理空间数据类型，我们在本章后面会讨论这个新类型。实际上，它们在本质上与你可能开发和部署的任何.NET 数据类型相同，只是被标记为系统类型，不需要显式启用.NET 以保证使用地理空间数据类型。

为了复制相同的语法例子，这里给出了.NET 的 UDT 语法模板：

```
CREATE TYPE [<schema name>.<type name>  
EXTERNAL NAME <assembly name>[.<class name>]
```

作为第 10 章中将会用到的例子的预览(不要执行这段代码——我们只在适当的时候使用它)，我们会通过代码添加一个称为 ComplexNumber 的.NET 程序集：

```
CREATE TYPE ComplexNumber  
EXTERNAL NAME [ComplexNumber].[Microsoft.Samples.SqlServer.ComplexNumber];
```

再说明一下，我们在第 10 章中会全面研究基于.NET 的 UDT，包括创建自己的类型。

7.2.3 表格式 UDT

这部分是新加入 SQL Server 2008 的，而且它们开启了一件重要的事情，我认为它会在接下来的几次发布中有所变化。

它们是什么？它们是：接受表数据的用户定义数据类型。创建它们的语法与表值变量或在 CREATE TABLE 命令中使用的语法最为匹配。创建成功后，就可以在脚本中使用它们，或在存储过程中作为表值参数使用它们，后者或许更加重要。

注意我没有提到将其作为可以在表中使用的类型使用它。不同于其他用户定义数据类型，在其他表类型对象(即，表对象或表变量)中不能使用基于表的用户定义数据类型。

注意:

不同于其他的用户定义数据类型, 表格式 UDT 不能被嵌入其他表对象, 例如表变量或表对象中。

到编写本书时为止, 微软还没有承认自己在表格式 UDT 上究竟已经走了多远。目前, 可以看到的是他们正在路上, 越来越靠近在竞争产品(比如, Oracle)中发现的类似表格式 UDT 的功能, Oracle 很久以前就可以在一个表中嵌入另一个表了。

现在, 我们要集中精力研究如何准确地创建表格式 UDT。在第 10 章中, 我们会分析表格式 UDT 最常见的用法: 存储过程的表值参数。

1. 创建一个表格式用户定义数据类型

表格式用户定义数据类型的创建方式类似于将典型的 CREATE TYPE 和表变量语法组合在一起。表的 CREATE TYPE 语法如下所示:

```
CREATE TYPE [<schema name>.]<type name>
AS TABLE
(
    { <column name> <data type>
        [ COLLATE <collation name> ]
        [ NULL | NOT NULL ]
        [ DEFAULT <expression> ]
        [ IDENTITY [ ( <seed>, <increment> ) ] ]
        [ ROWGUIDCOL ] [ <column constraint> [ ...n ] ]
        [ <table constraint> ] }
    | <computed column definition> }
);
```

作为一个例子, 我们将创建一个用户定义的表类型来代表这些地址。本书的后面部分(在第 10 章中)会看到如何将这种数据类型的一个实例传递给一个存储过程或函数以便于继续处理。

很久以前, 对绝大多数人来说一个地址似乎就足够了。绝大多数的系统为自己要处理的每个业务实体存储了一个地址。不过, 现在一个地址似乎已经不够了。我们要处理有多个地址的公司, 甚至要在一个地点接收帐单, 然后再将这些帐单分发到不同的地点, 我们要处理的很多商业实体和个人也有多个地址。AdventureWorks2008 数据库用在不同表(Person.Address)中的独立地址进行表示。我们已经决定在系统中使用一种统一的方式表示地址, 因此我们创建了自己的自定义类型:

```
USE AdventureWorks2008;
GO

CREATE TYPE Person.Address
AS TABLE(
    AddressID int NULL,
    AddressLine1 nvarchar(60) NOT NULL,
    AddressLine2 nvarchar(60) NULL,
    City nvarchar(30) NOT NULL,
    StateProvinceID int NOT NULL,
    PostalCode nvarchar(15) NOT NULL,
    SpatialLocation geography NULL
);
```

关于这个脚本，要注意下面的一些项目：

- 我使用了数据库中现有的一个对象名(存在一个称之为 `Person.Address` 的表)。这个类型可以被视为类似于类和对象之间的差别——即，类型是一个定义，表实际上是一个实例(尽管表不是类型定义的实例，而对象是类的实例)。
- 创建实际定义的语法与 `CREATE TABLE` 的语法非常类似。
- 为了在两者之间轻松地移动数据，布局图非常接近 `Person.Address` 表。

提示：

为了证明这一点，我们用与表相同的名称创建了自己的用户定义类型。我不推荐在实践中使用重复的名称，因为它可能会导致严重的混乱。

有了现在创建的类型，就可以将其作为变量声明、函数或存储过程参数的有效数据类型使用(更多的细节请参考第10章的最后两节)。

下面利用这个新的类型继续上面的例子：

```
DECLARE @Address Person.Address;
```

```
INSERT INTO @Address
(AddressID,
 AddressLine1,
 City,
 StateProvinceID,
 PostalCode
)
VALUES
(
  1,
  'My first address',
  'MyTown',
  1,
  '21212'
),
(
  1,
  'My second address',
  'OtherTown',
  5,
  '43434'
),
(
  1,
  'My third address',
  'MyTown',
  1,
  '21214'
);
```

```
SELECT *
FROM @Address;
```

注意，简单地声明一个 `Person.Address` 用户定义类型，我们就可以访问该表类型的所有列。

我们还能插入行并选中它们：

```
(3 row(s) affected)
AddressID  AddressLine1
-----
1          My first address ...
1          My second address ...
1          My third address ...

(3 row(s) affected)
```

第 10 章会进一步研究它的使用方法，在讨论表值参数时予以介绍。

7.2.4 删除用户定义的类型

这样做非常直观，不过只是为了确保这里对这一点做出了说明。使用通用的 `DROP<object type><object name>` 语法可以删除所有的 UDT：

```
DROP TYPE [<schema name>.<type name>];
```

7.3 层次数据

这个领域比我们在这一章中讨论的其他任何一个项目都更具革命性。虽然其他产品早就已经支持空间数据了(因为缺乏空间数据的支持，所以 SQL Server 经常受到耻笑)，新增的 HierarchyID 数据类型及其嵌入函数还是推动数据库进入了一种前所未有的功能境界。

什么是 HierarchyID 呢？简单来说，它是一种特殊的数据类型，为表示层次结构(通常是树形结构)中的单个节点而进行优化的。这里真正的威力是它能够分析层次祖先(父子关系)的概念，理解深度和兄弟节点的概念(例如，所有的部门经理与部门工作人员或执行人员之间)。

注意：

某个给定的 HierarchyID 数据实例并不代表一棵树。实际上它只是关于一棵树的某个节点的属性信息，包括节点的祖先。只有利用一组相关节点的集合，它才能表示一个真正的层次树。

对数据层次描述的需求并不是什么新鲜事。实际上，SQL Server 2005 对应的 AdventureWorks 版本已经包含了一个相当经典的模型描述一个或多个常见的层次问题：员工汇报结构(实际上，在第 3 章创建 Employee2 表介绍 CTE 的时候我们已经创建了一个类似的映射)。经典的解决方案是所谓的一元关系——即，表有一个指向自身的外键。另一个常见的层次问题是“工具包”(例如，某一部分只是其他部分的一个集合，也许其中的某些部分是其他部分的工具包)。XML 是层次化的，而且还是解决存储层次问题的常用方法，即使是对于非 XML 应用程序来说，也是如此。

下面看一下它的工作方式，随后研究数据类型及其相关方法带来的一些功能性问题。

7.3.1 理解深度与输出

在深入讨论 HierarchyID 类型的结构和支持它的索引类型和方法之前，理解深度(或等级)概念和称之为输出(fanout)的思想是很重要的(此时，将其视为水平的)。

层次节点的深度(深度等级)是基于直接或节点的祖先节点的。注意这样会产生一个基于零的集合——即, 层次的根节点的等级为 0 级, 它的直接后代是 1 级, 以此类推。例如, 图 7-1 中标号为 E 的节点等级为 2。图 7-1 中标记为 A 的根节点的等级为 0。

HierarchyID 类型为我们提供了一种特殊的方法调用, 通知我们某个节点位于层次中的哪个等级(称其为 GetLevel 并不奇怪)。等级主要用于兄弟节点之间的比较, 而且它在随后讨论 HierarchyID 列上的宽度优先索引的时候变得尤其重要。

层次的输出是指某个父节点上有多少个子节点。在如图 7-1 所示的树形表示法中, 你可以认为输出控制着层次的宽度。在图 7-1 中, E 节点的输出为 3, B 节点的输出为 4, 而且 A 节点输出为 11。在下面一节中, 我们会看到存储这些信息的方式。

7.3.2 HierarchyID 类型结构

HierarchyID 数据类型在内部是以节点的变长二进制表示法存储的。实际上, 如果取回了 HierarchyID 类型数据, 它就会返回 16 进制的表示法。例如, 如果执行:

```
SELECT e.OrganizationNode
FROM HumanResources.Employee e;
```

它会返回一些 16 进制数:

```
OrganizationNode
-----
0x
0x58
0x68
...
...
0x85EBA6
0x85EBAA
0x85EBAE

(290 row(s) affected)
```

可以使用 ToString 方法(下一节中会研究各种方法的调用)使得它更具可读性:

```
SELECT e.OrganizationNode.ToString() AS OrganizationNode
FROM HumanResources.Employee e;
```

返回的结果, 乍一看可能并不容易读懂:

```
OrganizationNode
/
/1/
/2/
/3/
/4/
/5/
/6/
/1/1/
/2/1/
```



```

/2/2/
...
...
/4/3/1/9/
/4/3/1/10/
/4/3/1/11/

(290 row(s) affected)

```

在这种字符串表示法中，要注意下面的几点：

- 每一个斜线(/)用来分割当前节点路径上的节点。
- 数字是任意的。这些数字可以自己分配，也可以由 SQL Server 代你插入。如果令 SQL Server 为你生成数字，那么这个数字会是下个可用的整数，除非明确声明自己想要现有节点之间的新值。此时你必须提供值的取值区间(在下一节介绍 GetDescendant 方法的时候会看到更多的有关内容)。只有明确地管理某个给定等级的值，你才能提供某个给定等级的所有分级形式。
- 在某个节点内，数字排序并不重要，起作用的只是序号中的位置。每一组数字都只在该特定等级上起作用(注意/1/1/中的第一个 1 与第二个 1 不同。层次中的每一级的号码序列都是分别维护的)。
- 单独一个斜线(/)表示根节点(等值的 16 进制数是 0x0)。不过，这也是任意的，因为允许有多个根节点。

注意：

HierarchyID 数据类型本身不能确保只有一个根节点。实际上，除非你明确施加那样的约束(通过主键或唯一约束)，否则唯一性根本就没有保证。

之前已经指出，HierarchyID 类型在内部代表变长位字段(所有的 16 进制输出)的节点。不同于其他的变长数据类型，这里没有明确定义长度。相反，SQL Server 会根据需要调整长度，以表示在不同节点的深度和输出。微软并没有真正说明如何对每个位进行操作的细节，不过根据微软公开发表的声明，可以算出在绝大多数的安装中每个节点的平均长度为 5-6 比特。

7.3.3 处理 HierarchyID 值——HierarchyID 方法

所有这些概念和理论都很棒，不过读过我的书的人应该已经知道我非常热衷于展示一些具体的例子。有了这种想法，下一节中就要使用一个例子来涵盖该类型支持的各种不同方法。这些工作方法类似于对 XML 数据使用的方法。我们会根据它们帮助我们完成操作(插入、定位、嫁接等等诸如此类)的函数介绍每种方法，不过这里会把它们放在一个地方以便于参考，下面简要列出了所有的方法及其使用的对象：

- GetAncestor(n)：取回树形结构中高 n 级的父节点的节点值，例如 GetAncestor(1)会取回自己的上一级父节点。
- GetDescendant(<child1>,<child 2>)：其行取决于为子参数提供的具体值，不过这个名称带有一定的误导性。不同于你的猜测，GetDescendant()不用于取回特定的子节点，这是因为层次中某个等级的子节点可能会很多，要取回某个特定的子节点可能比较困难。实际上，它用于计算向层次结构中插入新节点时可能会用到的值。

- **GetLevel()**: 返回当前节点的等级, 此时认为根节点的等级为 0, 而且根节点下的每一个子节点的等级都比其上的节点等级大 1。
- **GetReparentedValue(<old root>,<new root>)**: 这个名称也带有一定的欺骗性。尽管名称中使用了 **Get**, 不过 **GetReparentedValue()** 实际上完成的是一个任务——即, 从某个给定的父节点(旧节点)上剪掉某个给定的节点或一组子节点并将它们嫁接在新的父节点(新的根节点)上。不要被这里使用的术语“根”所迷惑。它并不必要是整个层次的主根, 可以是所有嫁接节点共享的通用父节点。
- **GetRoot()**: 提供层次结构中的根节点的常量值(始终是 0x0)。不同于这里讨论的其他大多数方法, **GetRoot()** 是一种静态方法, 因此只能在基本类型上调用这种方法, 而不能在某个节点的实例上调用。后面讨论取回根节点的时候, 我们会研究这方面的细节。
- **IsDescendantOf(<node>)**: 提供真/假标识, 表示当前的节点是否是某个给定节点的子孙。注意父节点被视为自己的一个子节点(因此, 如果在某个引用自己的节点上执行 **IsDescendantOf()** 操作, 返回结果应该是真)。
- **Parse()**: 宽泛地讲, 可以认为这种方法是与 **ToString()** 方法对立的。它接收用基于字符串的节点表示形式并将其转换为内部的二进制表示形式。类似于 **GetRoot()**, 这也是一个静态方法, 只能在基本类型上调用它(例如, **HierarchyID::GetRoot()**)。
- **Read()**: 这是一个 CLR 或仅限于客户端语言的函数(在 T-SQL 内不能调用它), 而且可以用于接收本地二进制形式的 **HierarchyID** 实例流。一般来说, 数据库开发人员只有在用客户端语言完成极其复杂的 CLR 编程或对层次进行操作的时候才会用到它。
- **ToString()**: 它的功能名副其实——即, 它将二进制表示法转换为更具可读性的字符串。
- **Write()**: 它在功能上与 **Read()** 相反。类似于 **Read()**, 也仅限于在 CLR/.NET 中使用, 而且不能在 T-SQL 中调用它。它用于接收二进制形式的客户端 **HierarchyID** 实例并将其直接写回 SQL Server, 不需要完成字符串的转换。

介绍完这些内容之后, 下面用一种更加功能化的角度查看这些内容, 并讨论一下希望利用 **HierarchyID** 数据类型实例完成的一些工作。

1. 与取回给定级别的层次数据有关的方法

在 Adventure Works 2008 数据库中有一些 **HierarchyID** 数据类型的例子(**Address**、**Document**、**Employee** 和 **ProductDocument** 表中各有一个例子)。这里会集中精力研究 **Employee** 表, 因为它最容易理解, 不过 **HierarchyID** 其他用法的例子更加深入。

首先从 **Employee** 表中取回一个简单的用户可读选择结果开始。为此, 使用上一节看到的 **ToString()** 方法。**ToString()** 没有输入参数而且是针对 **HierarchyID** 类型的数据实例(通常是一行数据或变量)使用的。因此其正式语法如下所示:

```
<instance of hierarchical data>.ToString()
```

为了便于管理, 这里会使用 **OrganizationLevel** 列限制结果:

```
SELECT e.BusinessEntityID,  
       p.LastName + ', ' + p.FirstName AS Name,  
       e.OrganizationNode.ToString() AS Hierarchy  
FROM HumanResources.Employee e  
JOIN Person.Person p
```



```
ON e.BusinessEntityID = p.BusinessEntityID
WHERE e.OrganizationLevel BETWEEN 1 AND 2;
```

这样会返回一组父节点及其各自的子节点:

BusinessEntityID	Name	Hierarchy
2	Duffy, Terri	/1/
16	Bradley, David	/2/
25	Hamilton, James	/3/
234	Norman, Laura	/4/
263	Trenary, Jean	/5/
273	Welcker, Brian	/6/
3	Tamburello, Roberto	/1/1/
17	Brown, Kevin	/2/1/
18	Wood, John	/2/2/
19	Dempsey, Mary	/2/3/
20	Benshoof, Wanida	/2/4/
21	Eminhizer, Terry	/2/5/
22	Harnpadoungsataya, Sariya	/2/6/
23	Gibson, Mary	/2/7/
24	Williams, Jill	/2/8/
26	Krebs, Peter	/3/1/
211	Abolrous, Hazem	/3/2/
222	Wright, A. Scott	/3/3/
227	Altman, Gary	/3/4/
235	Barreto de Mattos, Paula	/4/1/
241	Liu, David	/4/2/
249	Kahn, Wendy	/4/3/
262	Barber, David	/4/4/
264	Conroy, Stephanie	/5/1/
267	Berg, Karen	/5/2/
268	Meyyappan, Ramesh	/5/3/
269	Bacon, Dan	/5/4/
270	Ajenstat, François	/5/5/
271	Wilson, Dan	/5/6/
272	Bueno, Janaina	/5/7/
274	Jiang, Stephen	/6/1/
285	Abbas, Syed	/6/2/
287	Alberts, Amy	/6/3/

(33 row(s) affected)

注意 HierarchyID 列中使用的编号与其他列无关。BusinessEntityID 是表的主键，但是层次表示中根本就没有利用它。下面看一看 Roberto Tamburello，我们可以看到他向 Terri Duffy 汇报工作。每一级层次都重复使用了数字“1”，暗示着这与在层级结构的其他层中使用数字“1”的方式无关。这里看到每一层的数字序列都是按顺序排列的，不过这是这个特定数据集的特殊情形。没有要求说一定要按照顺序排列(可以出现小数和负数)。

接下来，注意之前查询中使用的 OrganizationLevel 列。如果查看数据库中这一列的定义，你会发现这一列是计算出来的。实际上它利用的方法是 GetLevel()，下面会查看这种方法。

GetLevel()没有输入参数(它在你使用 GetLevel 方法的层次数据实例上进行操作，而且会返回该节点在层次中的深度，根节点的深度为 0，根节点的第一级子节点是 1 层，它们的子节点是 2

层, 依此类推)。因此, 可能的语法是:

```
<instance of hierarchical data>.GetLevel()
```

如果要比之前查询中使用的 `OrganizationLevel` 的结果和直接使用 `GetLevel()` 的结果, 它可以重新写为:

```
SELECT e.OrganizationNode.ToString() AS Hierarchy,
       OrganizationLevel,
       e.OrganizationNode.GetLevel() AS ComputedLevel
FROM HumanResources.Employee e
WHERE e.OrganizationLevel BETWEEN 1 AND 2;
```

不出所料, 这样做会返回相同的 `OrganizationLevel` 值而且使用 `GetLevel()` 方法:

Hierarchy	OrganizationLevel	ComputedLevel
-----	-----	-----
/1/	1	1
/2/	1	1
/3/	1	1
...		
...		
/6/1/	2	2
/6/2/	2	2
/6/3/	2	2

(33 row(s) affected)

这里可以使用很多方法, 不过一定要注意下面的这些要点:

- 返回与层次结构中某级相关的所有数据行。例如, 查找 1 层或 2 层可能会找到 CxO 层的雇员, 在 3 层中可能会找到区域经理。这只与设置层次的方法有关。
- 编制水平对比的索引。

2. 与取回父或子层次数据有关的方法

能够观察到层级结构中的某一级或节点的信息固然不错, 但是事实上这样并不能彰显 `HierarchyID` 数据类型的威力。为此, 必须进一步扩展到父/子关系, 这是层次数据的基础。实际上, 这个功能的焦点就是 `GetAncestor()` 和 `IsDecendantOf()` 方法。

下面从 `GetAncestor()` 的模板开始, 它只接受一个参数。下面给出了它的语法:

```
<instance of HierarchyID data>.GetAncestor(n):
```

这个方法可以对层次数据的实例执行操作, 而且使用单个参数来表示要向上通过的树的级数。

注意:

`GetAncestor()` 返回的值是 `HierarchyID` 类型, 这表示你可以使用其他的 `HierarchyID` 方法进一步扩展 `GetAncestor()` 调用。

下面看一看, 在取出自己在第一个层次查询例子中那个名为 Roberto Tamburello 的雇员的不同上级级别时会看到哪些内容。回忆一下, 你可能会看到与下面类似的层次节点:

/1/1/

运行几个 `GetAncestor()` 方法实例，观察返回结果：

```
SELECT e.BusinessEntityID,
       p.LastName + ', ' + p.FirstName AS Name,
       e.OrganizationNode.ToString() AS Hierarchy,
       e.OrganizationNode.GetAncestor(0).ToString() AS Self,
       e.OrganizationNode.GetAncestor(1).ToString() AS OneUp,
       e.OrganizationNode.GetAncestor(2).ToString() AS TwoUp,
       e.OrganizationNode.GetAncestor(3).ToString() AS TooFar
FROM HumanResources.Employee e
JOIN Person.Person p
  ON e.BusinessEntityID = p.BusinessEntityID
WHERE e.BusinessEntityID = 3
```

如果仔细观察，会看到好几次都取回了相同的节点，不过对于每一列，在超过特定数据所在的层级之前，我会一直沿着层次等级向上走。运行它，会得到单行返回结果：

BusinessEntityID	Name	Hierarchy	Self	OneUp	TwoUp	TooFar
3	Tamburello, Roberto	/1/1/	/1/1/	/1/	/	NULL

(1 row(s) affected)

关于这个结果，要注意下面的几点：

- 尽管它提供的值很少，但是 0 是有效参数(它返回的是调用节点本身)。
- `GetAncestor()` 的参数每增加 1，就沿着层次树向上前进一步。
- 使用高于层次树中根节点的数值，就会返回 NULL。

这对沿层次树向上前进非常重要，不过如果要返回子节点，或只是要知道某个特定的子节点在其祖先集合中是否有一个特定的父节点，该怎么办呢？这个问题的正确答案取决于我们是否知道自己要在层次链中走多远(所有的汇报者或只是直接汇报者)。如果是所有的汇报者，那么使用 `IsDecendantOf()`。这个函数接受单个节点作为输入参数，并返回一个简单的真/假布尔结果。正如你所预料到的，这个结果表示传递给方法的节点是否(直接或间接地)包含调用方法的子节点。下面给出了它的语法：

```
<instance of HierarchyID data>.IsDescendantOf(n):
```

先看看如何在不同的方向上使用它。例如，假设要返回 Tamburello 先生的所有上级主管。这就是要返回将 Tamburello 节点视为子孙的节点的行。例如：

```
DECLARE @ChildNode HierarchyID

SELECT @ChildNode = OrganizationNode
FROM HumanResources.Employee e
WHERE e.BusinessEntityID = 3

SELECT e.BusinessEntityID,
       p.LastName + ', ' + p.FirstName AS Name,
       e.OrganizationNode.ToString() AS Hierarchy
FROM HumanResources.Employee e
JOIN Person.Person p
```

```
ON e.BusinessEntityID = p.BusinessEntityID
WHERE @ChildNode.IsDescendantOf(e.OrganizationNode) = 1;
```

首先, 注意我们可以将节点移动到某个 `HierarchyID` 类型的变量上, 而且我们还能从这个变量调用方法。为什么要使用它而不使用 `GetAncestor()` 呢? 如果你思考一下这个问题, 我猜你会发现它和这个问题的限制有关。 `GetAncestor()` 确实希望你知道自己有多少个上级。你可以使用 `GetLevel()` 方法或配备一个针对 `NULL` 值的测试, 计算出这个值, 不过这样做比简单返回所有 `IsDescendant()` 为 `True` 的行要复杂得多。

BusinessEntityID	Name	Hierarchy
1	Sanchez, Ken	/
2	Duffy, Terri	/1/
3	Tamburello, Roberto	/1/1/

(3 row(s) affected)

注意:

一个节点可以认为自己既是自己的上级节点(输入层级为 0), 也可以是自己的子节点。

上面展示了检查自己上级节点的方法, 不过如何检查自己的子节点呢? 为此, 我们提出一个更加开放的问题。例如, 要列出所有直接或间接向 Tamburello 先生汇报工作的人只需要将之前查询中的 `WHERE` 条件反转一下就可以了:

```
DECLARE @ChildNode HierarchyID

SELECT @ChildNode = OrganizationNode
FROM HumanResources.Employee e
WHERE e.BusinessEntityID = 3

SELECT e.BusinessEntityID,
       p.LastName + ', ' + p.FirstName AS Name,
       e.OrganizationNode.ToString() AS Hierarchy
FROM HumanResources.Employee e
JOIN Person.Person p
  ON e.BusinessEntityID = p.BusinessEntityID
WHERE e.OrganizationNode.IsDescendantOf(@ChildNode) = 1;
```

很快就可以看到 Tamburello 的所有汇报者了:

BusinessEntityID	Name	Hierarchy
3	Tamburello, Roberto	/1/1/
4	Walters, Rob	/1/1/1/
5	Erickson, Gail	/1/1/2/
6	Goldberg, Jossef	/1/1/3/
7	Miller, Dylan	/1/1/4/
8	Margheim, Diane	/1/1/4/1/
9	Matthew, Gigi	/1/1/4/2/
10	Raheem, Michael	/1/1/4/3/
11	Cracium, Ovidiu	/1/1/5/
12	D'Hers, Thierry	/1/1/5/1/
13	Galvin, Janice	/1/1/5/2/

14	Sullivan, Michael	/1/1/6/
15	Salavaria, Sharon	/1/1/7/

(13 row(s) affected)

为了获得他的直接汇报者, 使用相同的查询, 不过返回给 `GetAncestor()` 方法:

```
DECLARE @ChildNode HierarchyID;

SELECT @ChildNode = OrganizationNode
FROM HumanResources.Employee e
WHERE e.BusinessEntityID = 3;

SELECT e.BusinessEntityID,
       LEFT((p.LastName + ', ' + p.FirstName), 30) AS Name,
       LEFT(e.OrganizationNode.ToString(), 10) AS Hierarchy
FROM HumanResources.Employee e
JOIN Person.Person p
  ON e.BusinessEntityID = p.BusinessEntityID
WHERE e.OrganizationNode.GetAncestor(1) = @ChildNode;
```

这样做会被限制在自己下面的某个特定层级上(或者 `GetAncestor()` 认为我们目前比它高一级)。

3. 插入新的层次数据

基本上, 插入新的层次数据与在 SQL Server 中插入其他数据类似。实际上, 插入新层次节点的关键是理解应该用什么表示法表示新行。

记住, SQL Server 并不预先了解你的层次。SQL Server 不一定认为它是一颗树或给定节点是唯一的树或节点。虽然 SQL Server 不能为你建立层次, 不过它可以根据你提供的信息帮助你生成值。这个功能是由 `GetDescendant()` 方法提供的。

如果 `GetDescendant()` 称为 “`GenerateHierarchyNodeRepresentation()`” 或诸如此类的方法, 那么它的名称应该更准确一些。这样做的目的是生成落在两个可选设置参数之间的层次节点的有效表示。其语法如下所示:

```
<parent node>.GetDescendant({<Low Child> | NULL}, {<High Child> | NULL})
```

低级和高级子节点指定了一个范围, 生成的值必须落在这个范围内(不包含两端)。只要落入指定的范围内, 生成的数值可以包含小数或负数。虽然两个参数都是必需的, 不过可以明确地指定 NULL 作为低级和高级子节点的数值, 这样就等于没有设定生成边界。

- 如果父节点的值为 NULL, 返回 NULL 值;
- 如果父节点的值不为 NULL, 而且低级和高级子节点均为 NULL, 那么返回父节点的一个子节点;
- 如果父节点和低级子节点不为 NULL, 而且高级子节点为 NULL, 那么返回比低级子节点更大的子节点;
- 如果父节点和高级子节点不为 NULL, 而且低级子节点为 NULL, 那么返回比高级子节点更小的子节点;
- 如果父节点、低级子节点和高级子节点不为 NULL, 那么返回一个比低级子节点大, 但是比高级子节点小的子节点。

- 如果低级子节点不为 NULL 而且也不是父节点的子节点, 那么将产生异常;
- 如果高级子节点不为 NULL 而且也不是父节点的子节点, 那么将产生异常;
- 如果低级子节点等于或大于高级子节点, 那么将产生异常。

这里会使用下面的脚本生成一个自定义例子来说明这个特殊的调用方法:

```
CREATE TABLE NodeTest
(
    NodeID      int NOT NULL IDENTITY PRIMARY KEY,
    Node        hierarchyid NOT NULL,
    NodeLevel AS Node.GetLevel(),
    Name        varchar(50) NOT NULL
);
INSERT NodeTest
VALUES
    ('/', 'Manager');

DECLARE @Manager hierarchyid;
SELECT @Manager = Node
FROM NodeTest
WHERE NodeID = 1;

INSERT NodeTest
VALUES
    (@Manager.GetDescendant(NULL, NULL), 'ReportAAA'),
    (@Manager.GetDescendant(NULL, NULL), 'ReportBBB'),
    (@Manager.GetDescendant(NULL, '/1000/'), 'ReportCCC'),
    (@Manager.GetDescendant(NULL, '/1000/'), 'ReportDDD'),
    (@Manager.GetDescendant('/1000/', NULL), 'ReportEEE'),
    ('/547/', 'ReportFFF'),
    (@Manager.GetDescendant('/3/', '/547/'), 'ReportGGG'),
    (@Manager.GetDescendant('/1/', '/2/'), 'ReportHHH'),
    (@Manager.GetDescendant('/-10/', '/-1/'), 'ReportIII'),
    ('/547/345/', 'SecondLevelAA'),
    ('/547/346/', 'SecondLevelBB'),
    ('/547/345/1/', 'ThirdLevelAA'),
    ('/785/294/386/925/', 'RandomEntry');

SELECT NodeID,
       Node.ToString(),
       Name
FROM NodeTest;
```

有了这个脚本, 可以插入很多数据, 不过有几处的输出有些令人吃惊:

NodeID		Name
1	/	Manager
2	/1/	ReportAAA
3	/1/	ReportBBB
4	/999/	ReportCCC
5	/999/	ReportDDD
6	/1001/	ReportEEE

7	/547/	ReportFFF
8	/4/	ReportGGG
9	/1.1/	ReportHHH
10	/-9/	ReportIII
11	/547/345/	SecondLevelAA
12	/547/346/	SecondLevelBB
13	/547/345/1/	ThirdLevelAA
14	/785/294/386/925/	RandomEntry

(14 row(s) affected)

注意可以随机插入数据。例如，这里有一个称之为 **RandomEntry** 的四级节点——它是随机的。它没有父节点。**SQL Server** 没有强迫你使用树表示法或强迫你的层次必须是有效的，它只提供工具，使节点能够按照创建层次树一样的方式协同工作。

接下来，注意插入了一个小数值。这里的第九项是插入 1 到 2 之间的一个数，所以不使用小数的话就没有办法把它插进去(这就是 **SQL Server** 完成的工作)。

这里还有负数。我们向 **SQL Server** 提供了非实数的选项，因为这里的低级子节点和高级子节点都是负数。

最后，插入了重复的行。**HierarchyID** 列和其他数据类型一样并非是无二一的。如果要避免重复节点数值，必须利用一个唯一性或主键约束。还要注意，对于给定的某个高级子节点或低级子节点，**GetDescendant()** 会多次生成相同的数值，而不管是否有重复(同时无视是否存在一个唯一性或主键约束)。必须盘算一下自己要插入的数值。对于大多数的层次来说，水平位置并不重要，因此通常可以使用自己要插入的层次中的最大节点。

4. 在父节点之间移动子树

HierarchyID 数据类型还提供了使用 **GetReparentedValue()** 方法将一个节点及其子节点嫁接到新的父节点的概念。类似于 **GetDescendant()**，**GetReparentedValue()** 的名称似乎暗示了它的主要功能是取回数据。虽然它在技术上就是要返回数据，不过 **GetReparentedValue()** 主要还是要移动数据。它需要两个参数：旧的“根节点”和新的“根节点”。下面给出了它的基本语法：

```
<node to be moved>.GetReparentedValue(<old root>, <new root>)
```

注意此时的“根节点”并不是整个层次的顶级根节点。它只是你要移动的某个特定子树的根节点。

注意：

GetReparentedValue() 并不必完成这次移动操作。它只是演示“如果...那情况会怎么样”这种场景的一种方法。在使用 **UPDATE** 语句的时候，它会完成实际的移动操作。

下面回到前一例子中创建的 **NodeTest** 表。这里要看一下如果将节点/547/的子节点移动到节点/1001/下，会出现什么情况。将 **GetReparentedValue()** 方法和 **IsDescendantOf()** 方法结合起来，可以完成这个任务：

```
SELECT NodeID,
       Node.GetReparentedValue('/547/', '/1001/').ToString() AS New,
       Node.ToString() AS Old,
```

```

Name
FROM NodeTest
WHERE Node.IsDescendantOf('/547/') = 1;

```

这段代码给出了如果剪掉节点/547/子树(包括/547/节点本身)并将所有相关的节点嫁接到节点/1001/下会出现的情况。下面看一看结果:

NodeID	New	Old	Name
7	/1001/	/547/	ReportFFF
11	/1001/345/	/547/345/	SecondLevelAA
12	/1001/346/	/547/346/	SecondLevelBB
13	/1001/345/1/	/547/345/1/	ThirdLevelAA

(4 row(s) affected)

乍一看, 结果似乎很好, 不过还有一个潜在的问题: 实际的/547/节点。在最初的数据中, 已经有一个/1001/节点了。如果这里允许重复(使节点看上去有两个父节点), 那么这里就不会有问题。不过在绝大多数情况下, 一个节点会有一个父节点而且也只能有一个父节点。为了改变这种情况, 以便只移动/547/的子节点, 这里只要使用 WHERE 子句将其从结果集中排除就可以了:

```

SELECT NodeID,
       Node.GetReparentedValue('/547/', '/1001/').ToString() AS New,
       Node.ToString() AS Old,
       Name
FROM NodeTest
WHERE Node.IsDescendantOf('/547/') = 1
      AND Node.ToString() != '/547/';

```

并且这里已经快速地将错误节点从结果中移除了:

NodeID	New	Old	Name
11	/1001/345/	/547/345/	SecondLevelAA
12	/1001/346/	/547/346/	SecondLevelBB
13	/1001/345/1/	/547/345/1/	ThirdLevelAA

(3 row(s) affected)

计算出所有的结果, 这样就可以用 UPDATE 语句真正移动数据了:

```

UPDATE NodeTest
SET Node = Node.GetReparentedValue('/547/', '/1001/')

WHERE Node.IsDescendantOf('/547/') = 1
      AND Node.ToString() != '/547/';

```

执行它, 随后从 NodeTest 表中重新选中所有的数据:

NodeID	Name
1	/
2	/1/

3	/1/	ReportBBB
4	/999/	ReportCCC
5	/999/	ReportDDD
6	/1001/	ReportEEE
7	/547/	ReportFFF
8	/4/	ReportGGG
9	/1.1/	ReportHHH
10	/-9/	ReportIII
11	/1001/345/	SecondLevelAA
12	/1001/346/	SecondLevelBB
13	/1001/345/1/	ThirdLevelAA
14	/785/294/386/925/	RandomEntry

(14 row(s) affected)

正如之前计划好的, 之前/547/节点的所有子节点都被移动到/1001/节点下面。/547/节点仍然保持原来的状态。

5. 获得层次的根节点

我认为这一点值得提及, 不过它可能有点虎头蛇尾。这里要介绍的最后一种方法(这里只局限于 T-SQL 可以解决的那些方法)是取回层次的根节点。这个方法的奇特之处在于它会返回一个常量。既然它是一个 HierarchyID 类型的静态成员, 可以使用 HierarchyID 类型而不是某个特定的实例来引用它。如果这样选择的话, 可以跳过它, 因为值总是相同的(如果针对其执行 ToString(), 则为 “/”)。语法是很好理解的, 而且不会随着特定实现方式的变化而变化:

```
HierarchyID::GetRoot()
```

之前说过, 这种方法没有什么魔法。可以选中它:

```
SELECT HierarchyID::GetRoot().ToString();
```

这样会返回一个熟悉的斜线:

```
-----  
/
```

(1 row(s) affected)

7.3.4 索引层次数据

要索引自己的层次数据, 有两种方法可供使用:

- **垂直地**(也就是“深度优先”的方法): 这是索引 HierarchyID 列的基本索引方法。它从自己可以查找到的最高节点(假设你有一个根节点)开始并且沿着树下降。如图 7-2 所示, 当它到达节点底部的时候, 它会索引该层的所有节点, 然后返回未被索引的相同树枝的最低节点, 然后又开始沿着树上升。使用前面一章介绍过的标准索引语法可以创建索引。如果要将 HierarchyID 列作为自己的第一列进行索引, 那么你一定获得一个深度优先索引。

- 水平地(通常所说的“宽度优先”索引方法): 创建一个宽度优先的索引需要付出额外的代价, 在为此担心之前, 这里先集中精力研究一下它要完成的工作。

宽度优先的索引存储兄弟节点(如图 7-3 所示)。创建它的目的是用于围绕 `GetAncestor()` 之类的方法的比较。为了使用这种处理顺序创建一个索引, 必须基于 `GetLevel()` 方法创建一个计算列, AdventureWorks2008 数据库中的 `Employee` 表就有一个类似的 `OrganizationLevel` 列(就像在 `NodeTest` 中创建的一样)。随后, 可以索引 `Level` 列和之后的 `HierarchyID` 列, 使其按照深度优先的方式进行索引。

不考虑深度优先和宽度优先索引的区别, `HierarchyID` 索引的工作方式与其他 SQL Server 索引的工作方式相差无几。

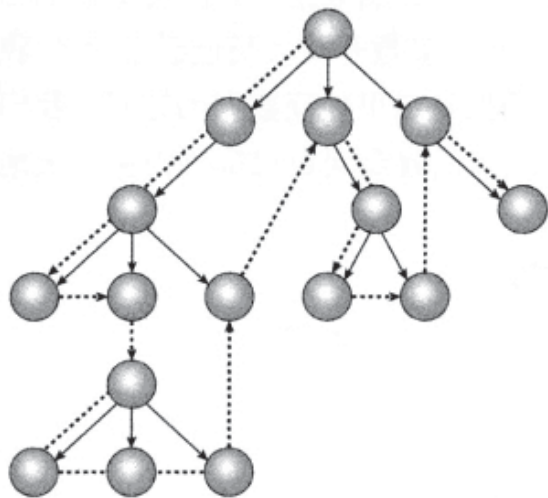


图 7-2

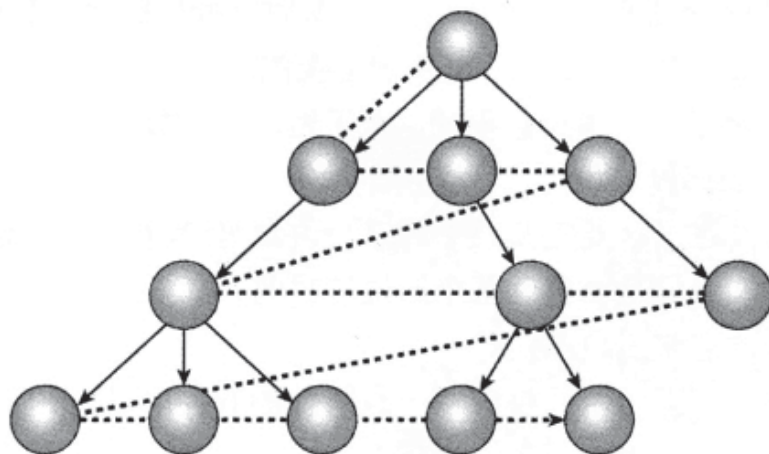


图 7-3

7.3.5 性能考虑

总的来说, `HierarchyID` 方法会提供最好的层次数据的整体性能和功能。不过因为软件开发中存在许多其他环节, 索引的最好性能规则还存在一些其他方法和例外。在开始讨论层次的时候, 我们就讨论了一些备用方案, 下面利用一种表方法研究每种选项的性能效果, 如表 7-1 所示。

表 7-1

情 况	可能是最好的方法		
	HierarchyID	父/子	XML
子树查询很常见(这是最普遍的场景)	×		
绝大多数的子树查询只是针对直接后代		×	
绝大多数的查询是针对层次中的某个已知、独立节点		×	
父节点经常变化		×	
一次消耗掉整个数据集(在理想情况下, 是 XML 格式的)			×

提示:

不要因为看到认为“父/子”关系最适合的方框数最多而产生困惑, “父/子”关系仍然可能是最好的解决方案。分别对待每种情况, 并认识到子树查询和父辈查询在层次中非常常见, 而且这些是 `HierarchyID` 数据类型最擅长的查询。

7.4 空间数据

新加入的空间数据处理是 SQL Server 2008 极力炒作的功能。我觉得最有趣的可能是可以向一个基本不理解该功能的听众极力炒作这一功能。

这里要得到的是什么呢？人们在大约十年前就开始强烈要求 SQL Server 加入这个新的地理空间数据类型了(这是 Oracle 团队经常关注的焦点，因为 Oracle 很久以前就可以处理地理空间数据了)。虽然它的功能十分强大，不过许多数据库开发人员都没有意识到它所指出的这个领域是自己需要的，更不用说真正理解了。

地理空间数据类型需要理解的数据类型不同于我们处理过的其他数据的数据类型。例如，在处理之前看到的新 HierarchyID 类型时，处理的数据类型是绝大多数开发人员已经了解的(我们很多年前就开始处理诸如 org 图表之类的层次了)。所以不同的只是这里处理数据的方式，我们之前已经了解了它的本质。不过，有了地理空间数据，许多开发人员就会向自己提出很多有关地理空间数据的问题。例如：

- 这只是要定义某个特定的位置吗(例如，一个地址)？
- 这只是要定义某个属性边界吗？
- 它映射道路吗？
- 有多少客户住在这一点附件的五英里以内？
- 麦迪逊郡有多少座桥？

现实比上述任一问题都要复杂。实际上它包含了上面列出的所有概念。过去是如何设计这些问题的呢？我们用一些相对简单(而且功能比较差)的方法解决了其中的一些问题，例如包含一个简单地址。我们甚至可能将地址传递给一个外部应用程序，令其保存地理空间数据并利用该应用程序的反馈来提出更大的问题。但是，目前的终端用户希望得到更多的信息。例如，几乎不可能找到一个网站没有包含“查找附近的商店”特征的零售商店或餐馆的链接。它们使用地理空间数据支持这一功能。

考虑到这些需求，下面研究一下两种地理空间数据(平面或测量的)和每种类型支持的功能。

7.4.1 空间概念

为了找出要处理的某种地理空间数据的特征，这里首先要了解一些更加通用的表示空间数据的方法。正如可以想象到的，表示空间数据的方法有一定的标准。不过很遗憾，这种标准不止一个(SQL Server 支持若干“模型”)。

要开始理解地理空间数据，必须首先要知道表示地理空间数据的主要方法有两个：平面(平地球)和测量(圆地球)。这两种方法有着相同的目标：使用一组数据点(点、直线、曲线)表示空间。平面表示法一般更加简单，因而更容易理解和管理。平面数据经常用于描述相对来说比较“局部”的数据——即，数据不需要覆盖一个特别大的区域而且不需要针对地球表面的弯曲度调整精度。测量表示法“更真实地”描述了这个世界，而且一般在需要表示更容易受到地球弯曲度影响的比较大的区域的时候使用它。

1. 平面(平地球)数据

平面(planar)数据有若干个名称,例如地平线、几何学或平地球。可以将其自然地映射为自己在高中学过的欧几里德几何学。利用平面数据可以表示一个平面或一组平面上的所有对象。假设表示的空间是平面的。对于更小的区域来说,这是一种实用的查看空间数据的方法,因为它可以方便地实现可视化功能,绝大多数的功能不需要特别复杂的数学知识(例如,距离是一条直线)。使用几何类的图形中使用的那种 x 、 y 、 z 数据点(通过将数据点集合连接成线和多边形),就可以表示平面数据了。可以使用基本的几何学表示复杂的形状,而且可以处理重叠对象。

不管这里是否很好地描绘了平面图,通常用平面上的点表示的不是真正的平面。这样会引入一些问题。有一些方法可以将用平面表示法表示圆地球的负作用降到最低。图 7-4 和图 7-5 给出了一些常见的地球投影图。用平面的方法表示地球使用了“投影”的概念——即,将圆地球投影到一个平面上。



图 7-4



图 7-5

对许多空间数据来说,这些投影都很好。实际上,政府用来定位地产、道路和满足其他需求的绝大多数本地地图都是利用诸如经度和纬度的平面模型完成的。

提示:

要注意这里有关经度和纬度的假设。虽然它们可能看起来很好理解而且也符合概念,实际上目前世界上使用的经度和纬度的映射方法有很多。例如,全球定位系统(GPS)中使用的纬度与绝大多数的其他经度表示法(一般都基于皇家天文台的零经度定义)差距很大(根据你测量的时候站在地球的哪一部分,这个差距超过 100 米或更大)。

SQL Server 通过 GEOMETRY 数据类型支持平面数据(这是随后给出的绝大多数例子中的核心类型)。

注意:

存在多个被认可的地球模型。确保在提供或接收空间数据的时候,使用的模型是兼容的或者你知道如何调整两者之间的差别。

2. 测量(圆地球)数据

如图 7-5 所示, 测量数据表示的是一种基于圆地球的更加真实(也复杂得多)的模型。GEOGRAPHY 数据类型支持数据的测量表示法。

在平面数据模型中, 假设地球的表面是平的。这适用于在相对小的距离内测量的区域(几英里之内), 不过随着距离的增大, 这种方法就失效了。例如, 在测量俄勒冈州的波特兰和中国北京之间的距离时, 平面模型中使用的直线所代表的距离与实际距离可能相差好几英里。为什么? 在平面模型中, 距离是直线而不是弧度(该弧度沿着地球表面的曲度)。实际的问题可能会变得更加复杂, 因为地球不是完美的球(在某些地方, 它会突出), 沿着不同的方向测量, 圆的周长可能会相差几百英里。测量数据对地球的曲度进行建模, 而且在 SQL Server 中通过 GEOGRAPHY 数据类型支持它。

注意:

值得注意的是, SQL Server 只能表示单个半球内驻留的地理数据。半球可以被视为一个球的任一半——不管沿着哪个平面切割球体。

3. 表示空间数据

有一些通用的关键概念可以表示平面和测量数据, 它们合在一起允许你使用不同的方法表示某种数据类型。开发地理信息联盟(Open Geospatial Consortium, 简称 OGC, 是一个专门指定几何数据标准的组织)定义了若干的格式, 可以使用这些格式表示空间数据。SQL Server 2008 实现了其中的三种格式:

- **已知文本(WKT):** 这个看起来类似纯文本, 而且只是对一组对象顺序命名(例如一个点或一条线), 后面跟随的是每个对象的坐标信息。
- **已知二进制(WKB):** 实现与 WKT 相同的一般概念, 这种表示方法按照二进制流的方式(而不是按照纯文本的方式)对同样的信息进行编码。
- **地理标记语句(GML):** 这是一种表示几何数据的 XML 模式。GML 利用 XML 数据的自定义本质, 允许随同坐标数据其他信息(非坐标信息)。GML 数据可能包含扩充信息的例子有在某位置确定的信息或传感器信息(例如, 在加利福尼亚州洛杉矶的某个地方测量到的臭氧值以及在葡萄牙的里斯本完成类似的测量)。

注意:

在本书的例子中会利用 WKT, 不过这在很大程度上是一个可读性决定, 而且并不表示在一般性的使用中 WKT 是一个更好的选择(正确的选择会随着情况的变化而变化)。

不管利用的数据采用何种表示方式, 需要的一般对象必须相同。每种格式识别三个基本对象, 可以分别使用它们或将其结合在一起使用来表示空间数据。这些对象是:

- **点:** 这是空间内的一个特殊点。它没有长度、没有宽度也没有高度。它等同于你在地图上用图钉做出的标记, 代表你自己目前或之前的位置。一个点需要 X、Y 标记。
- **线:** 在 SQL Server 识别的每种格式中, 线是使用 LINESTRING 对象表示的。注意此处嵌入的 STRING 项的相关性。它识别线是由两个或多个点表示的。在线的定义中使用多个

点允许出现非直线。既然线串的每个部分都是两点之间的最短路径，增加表示相同概念线的点数会增加线条表示法的精确性。

- 如果线条没有自我交叉，那么可以将其视为“简单的”，而且可以将其视为一维对象，即使它被弯曲或形成一个环(一条线有着相同的起点和终点)，也是如此。

注意：

环并不意味着这个多边形是圆的——它只表示某种形式的闭合空间。

- **多边形：**尽管它是用一个或多个环(也是一条线有着相同的起点和终点)定义的，这个环定义了构成多边形的线串，没有改变线串的处理方式。不同于基本的环定义，基本的环定义是一维的而且没有面积，而多边形有面积。此外，定义多边形外部边界的环可以包含额外的多边形(可以定义外部多边形的面积为空)。我们认为内部的空多边形定义的空间不是父多边形的一部分。
- **集合：**这是其他三种对象的集合(线、点、多边形)。

不管在 SQL Server 中使用的是哪种空间数据类型(GEOMETRY 还是 GEOGRAPHY)，所有的这三种基本对象(或者它们的集合)都是可用的而且可以在给定的表中混合使用。例如一个关于地标的表可能存储了复杂的多边形代表黄石公园，一条直线代表赤道，一个简单的点代表地球上的最高点。每一种方法(或者它们的集合)都可以在同一个表的同一列中使用。

除了这些基本概念之外，OGC 还定义了一组方法，要使用空间数据就必须支持这些方法。我们会在例子中研究 SQL Server 支持的这些方法，不过值得注意的是每种空间数据都存在许多方法(使用相同的名称或只是稍微改一下名称)，而且在各种类型之间都具有相同的一般功能。OGC 函数都带有一个 ST 前缀，随后是一个表示函数功能的动词。它们是作为空间数据的每个实例的方法实现的。表 7-2 表讨论了一些重要的例子。

注意：

为了完成有效的比较，调用每种 ST 方法的时候，空间引用 id(或 SID)必须匹配。SRID 表示这个特定的空间实例正在引用的空间模型已经被(被欧洲石油调查小组)识别。如果两个实例的 SRID 不匹配，那么所有的比较都会返回 NULL。

表 7-2

方 法	用 法
.STArea()	计算多边形空间实例的面积，并考虑所含多边形的“空”空间
.STContains(<spatial instance>)	返回一个位，表示提供的实例是否被完整地包含在调用实例中
.STDistance(<spatial instance>)	提供一个数值表示提供的实例和调用实例之间的距离
.STEquals(<spatial instance>)	返回一个位，表示提供的实例与调用的实例是否相等。注意它们不必采用相同的定义方式，不过最终的结果必须相同(例如，如果定义一个包含 8 条线段的方块，另一个方块有四条线段，不过得到的边长和位置相同，则返回 1)
.STIntersects(<spatial instance>)	返回一个位，表示提供的实例与调用实例是否有交点
.STOverlaps(<spatial instance>)	返回一个位，表示提供的实例是否与调用实例重叠(例如，一条线起点在多边形内部，终点在多边形外部)

(续表)

方 法	用 法
.STTouches(<spatial instance>)	返回一个位, 表示提供的实例是否与调用的实例相切
.STWithin(<spatial instance>)	返回一个位, 表示提供的实例是否整个位于调用实例的内部, 如果提供实例的任何部分落在调用实例的外部, 那么 STWithin 会返回 0

OGC 函数列表实际上要长很多, 而且在 GEOMETRY 和 GEOGRAPHY 类型上有一定的不同, 不过这些列出了可用的方法, 其中包括索引空间数据的方法。可以在联机丛书中查看每种空间类型(GEOMETRY 和 GEOGRAPHY), 找到一个更完整的列表。

7.4.2 平面数据描述的实现——GEOMETRY 数据类型

之前提到过, 实现平面或平地球的数据类型称之为 GEOMETRY。使用 GEOMETRY 数据类型不但可以提供一种方式包含之前提过的几何对象定义的类型(点、线形和多边形), 而且还提供了可以在数据上使用的一系列方法。类似于之前在本章中讨论过的 HierarchyID 数据类型(以及之后会讨论的 GEOGRAPHY 类型), GEOMETRY 是利用一个 CLR 用户定义函数实现的(然后被标识为系统的, 这样就不需要考虑它的安全性了, 而这种考虑是真正的 CLR 用户定义类型所必需的)。类似于其他的 .NET 类, 可以利用类的许多属性和静态成员。

GEOMETRY 类型可以接受所有讨论过的几何类型。下面用一个简单的例子来检验一下, 这个例子不但对一个几何数据类型进行实例化, 而且还为它装载了数据。

注意:

SQL Server 可能试图在 Management Studio 中表示一些空间数据。不过, 只有你在查询编辑器窗口的“结果显示为网格”模式中, 才可以看到这种表示法。

首先检查将 WKT 数据转为这里的数据类型的方法:

```
DECLARE @MyGeometry GEOMETRY;

SET @MyGeometry = Geometry::STGeomFromText('LINESTRING(-3 3, 3 3, 3 -3, -3
-3, -3 3)', 0)
SET @MyGeometry = Geometry::Parse('LINESTRING(-3 3, 3 3, 3 -3, -3 -3, -3
3)')
SET @MyGeometry = 'LINESTRING(-3 3, 3 3, 3 -3, -3 -3, -3 3)'

SELECT @MyGeometry;

SET @MyGeometry = 'POLYGON((-3 3, 3 3, 3 -3, -3 -3, -3 3))'

SELECT @MyGeometry;
```

在这段代码中, 我们已经在称之为 @MyGeometry 的变量中声明了几何数据的实例。随后以三种不同的方式为自己的变量分配 linestring 数据。它们的功能相同, 最后隐式使用 Parse 函数分配数据。

接着选中自己最新分配的数据行。在执行时, Management Studio 不但会显示二进制表示形式,

而且还显示了视觉表示, 如图 7-6 所示。

注意:

为了观察空间数据标签, 必须使用查询编辑器窗口中的“结果显示为网格”模式。

接下来重复分配和选择操作, 不过这次是多边形而不是 `linestring`。最后生成的结果稍有不同(如图 7-7 所示)。

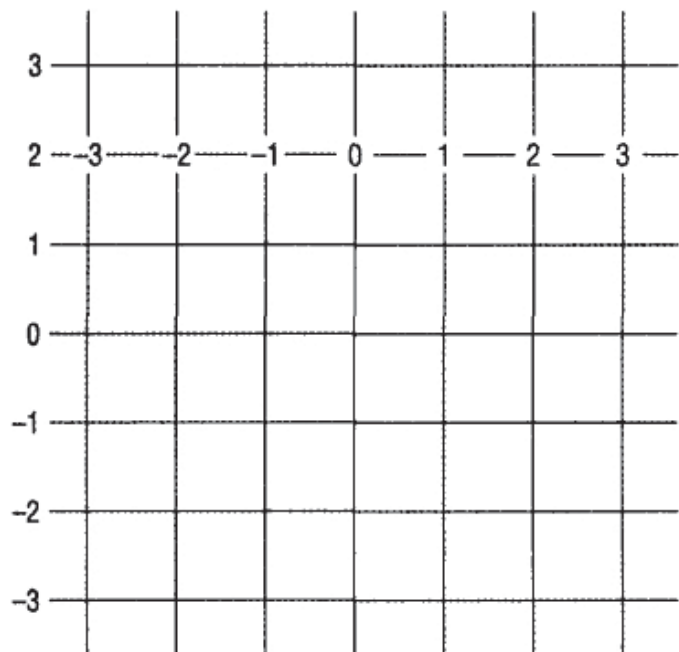


图 7-6

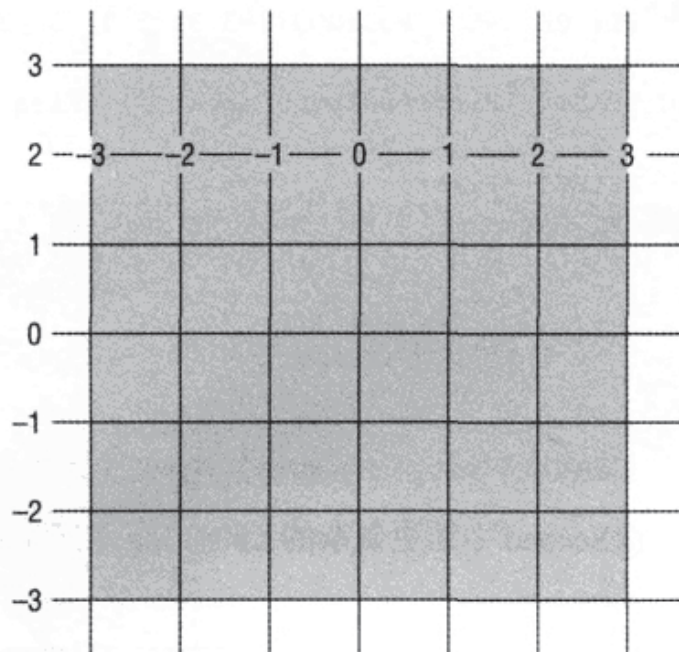


图 7-7

注意这两种基于相同点系列的对象的表示方法之间存在细微的差别。它们为什么不同呢? 将它称之为 `linestring` 通常是将其视为一维。尽管它们可以弯曲而且甚至可以彼此穿越, 它们还是被视为区域不足(需要二维)。SQL Server 用空洞来表示缺少区域的 `linestring` (尽管它构成了一个环)。但是, 对于多边形, SQL Server 将方形填满代表闭合的二维空间。SQL Server 意识到, 尽管基于相同的点, 它们两个之间还是存在可以识别的本质差别。随后差别会越来越明显, 因为 `GEOGRAPHY` 数据类型的各种方法都只是针对特定对象类型的(例如, 计算面积的方法只适用于多边形而不适用于线)。

当然, 这里的多边形并不局限于正方形或矩形。实际上, 只要多边形相对于自己的起点是闭合的, 它们可以是任意形状的(`linestring` 只能自我交叉, 不能形成环和多边形。因为多边形的起点必须和终点重合)。此外, 我们可以使用多边形内嵌入的多边形代表洞。下面来介绍这些概念。

首先, 要比较一些不同的 `GEOMETRY` 数据类型实例。这里还要再建立一个简单的方形, 不过这次要调用 `GEOMETRY` 类型的 `STArea()` 方法计算方形的面积:

```
DECLARE @First GEOMETRY,
        @Second GEOMETRY,
        @Merged GEOMETRY;

SET @First = 'POLYGON((-3 3, 3 3, 3 -3, -3 -3, -3 3))';

SELECT 'First polygon area: ', @First.STArea();

SELECT @First;
```


STArea()方法是空间数据方法 OGC 列表的一部分。执行这段代码，可以得到方形的表示(与图 7-7 中显示的一样)，不过这里得到的计算面积是 36。

继续扩展脚本，加入另一个多边形，不过这次要加入一个带有复杂 Linestring 的事物：

```
DECLARE @First GEOMETRY,
        @Second GEOMETRY,
        @Merged GEOMETRY;

SET @First = 'POLYGON((-3 3, 3 3, 3 -3, -3 -3, -3 3))'

SELECT 'First polygon area: ', @First.STArea();

SELECT @First;

SET @Second = 'POLYGON((-1 .4, -.4 1, .4 1, 1 .4, 1 -.4, .4 -1, -1 -1, -.4
-1, -1 -.4, -1 .4))';

SELECT @Second;

SET @Second= @Second.MakeValid();
SELECT 'Second polygon area: ', @Second.STArea();
```

@Second 中的更复杂的 Linestring 暗示正在显示一种更加复杂的形状：八边形(如图 7-8 所示)。

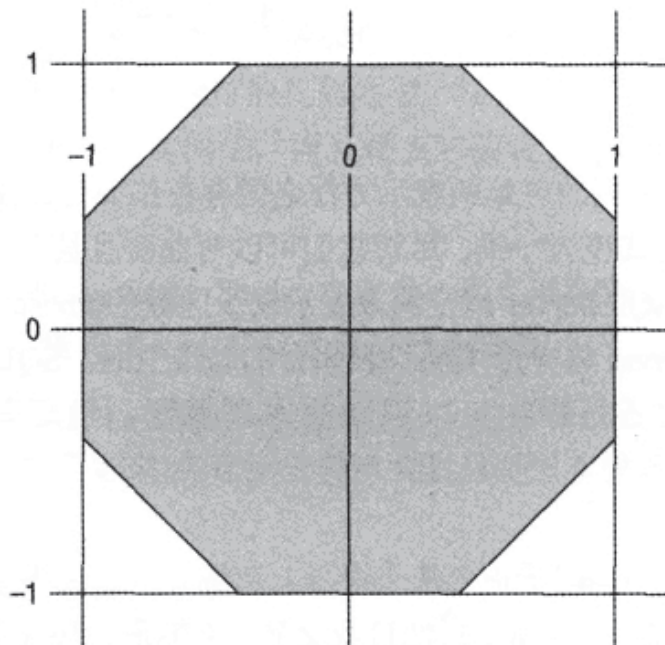


图 7-8

还要注意，这里必须在自己的多边形上完成额外的操作使其有效——一旦完成这些操作，就可以调用面积计算并接收返回的结果(3.280 000 228 881 82)。

继续上面的例子，可以利用 Linestring 构建一个多边形，第一多边形中的第二个变成一个空洞：

```
DECLARE @First GEOMETRY,
        @Second GEOMETRY,
        @Merged GEOMETRY;

SET @First = 'POLYGON((-3 3, 3 3, 3 -3, -3 -3, -3 3))'

SELECT 'First polygon area: ', @First.STArea();
```

```

SELECT @First;
SET @Second = 'POLYGON((-1 .4, -.4 1, .4 1, 1 .4, 1 -.4, .4 -1, -1 -1, -.4
    -1, -1 -.4, -1 .4))';

SELECT @Second;

SET @Second= @Second.MakeValid();
SELECT 'Second polygon area: ', @Second.STArea();

SET @Merged = 'POLYGON((-3 3, 3 3, 3 -3, -3 -3, -3 3),
    (-1 .4, -.4 1, .4 1, 1 .4, 1 -.4, .4 -1, -1 -1, -.4 -1, -1 .4, -1 .4))';

SELECT @Merged;

SET @Merged = @Merged.MakeValid();
SELECT 'Merged polygon area: ', @Merged.STArea();

```

这次 SQL Server 显示了两个多边形——反转了填充颜色以显示空洞区域(如图 7-9 所示)。

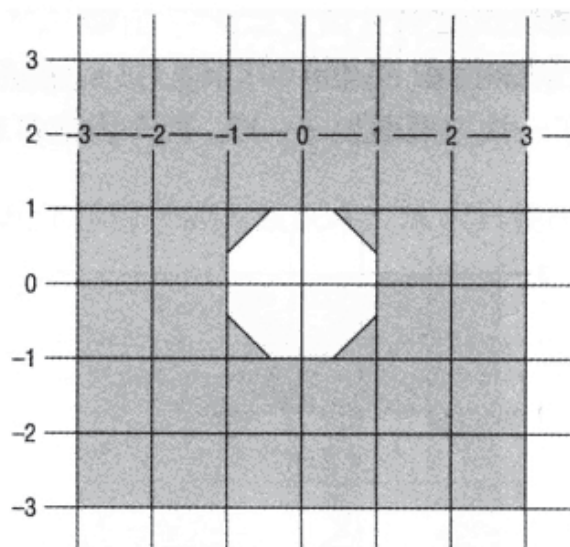


图 7-9

合并的多边形面积中已经考虑了空白区域(即, 从大多边形中减去它)并返回了正确的面积值 32.720 子 000 915 527 6。

最后再给脚本加入最后一部分内容, 这次在混合图中加入另一个多边形, 观察 SQL Server 如何处理重叠区域。这里要向合并的多边形添加另一个八边形:

```

DECLARE @First    GEOMETRY,
        @Second   GEOMETRY,
        @Merged   GEOMETRY;

SET @First = 'POLYGON((-3 3, 3 3, 3 -3, -3 -3, -3 3))';

SELECT 'First polygon area: ', @First.STArea();

SELECT @First;

SET @Second = 'POLYGON((-1 .4, -.4 1, .4 1, 1 .4, 1 -.4, .4 -1, -1 -1, -.4
    -1, -1 -.4, -1 .4))';

SELECT @Second;

```



```

SET @Second= @Second.MakeValid();
SELECT 'Second polygon area: ', @Second.STArea();

SET @Merged = 'POLYGON((-3 3, 3 3, 3 -3, -3 -3, -3 3),
                        (-1 .4, -.4 1, .4 1, 1 .4, 1 -.4, .4 -1, -1 -1, -.4 -1, -1
                        -.4, -1 .4))'

SET @Merged = @Merged.MakeValid();
SELECT 'Merged polygon area: ', @Merged.STArea();
SET @Merged = 'POLYGON((-3 3, 3 3, 3 -3, -3 -3, -3 3),
                        (-1 .4, -.4 1, .4 1, 1 .4, 1 -.4, .4 -1, -1 -1, -.4 -1, -1 -.4, -1 .4),
                        (-2.5 .4, -1.9 1, -1.1 1, -.5 .4, -.5 -.4, -1.1 -1, -2.5 -1, -1.9 -1, -2.5
                        -.4, -2.5 .4))'

SELECT @Merged;

SET @Merged = @Merged.MakeValid();
SELECT 'Second Merged polygon area: ', @Merged.STArea();

```

注意第三张图(如图 7-10 所示)和面积 30.490 001 068 115 8。注意显示了两个多边形(包括它们的重叠区域),得到的面积只减了一次空洞面积——即,两个内部多边形之间的重叠区域只是被删除了一次。

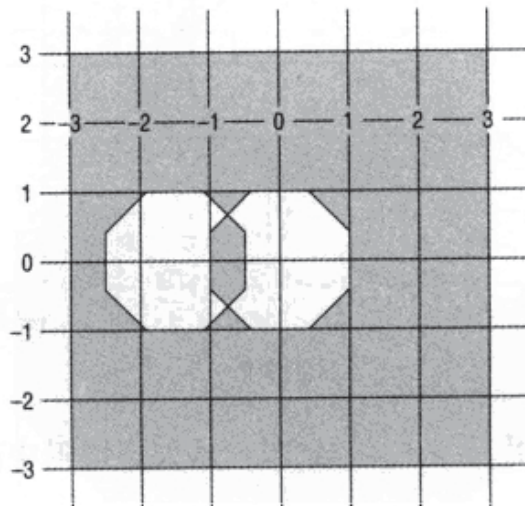


图 7-10

最后,也是比较重要的,这里快速看一下 ToString()方法。我们会使用相同的合并 GEOMETRY,激活 Makevalid()方法,然后输出稍经修改后的结果:

```

DECLARE @Merged GEOMETRY;

SET @Merged = 'POLYGON((-3 3, 3 3, 3 -3, -3 -3, -3 3),
                        (-1 .4, -.4 1, .4 1, 1 .4, 1 -.4, .4 -1, -1 -1, -.4 -1, -1
                        -.4, -1 .4))'

SET @Merged = @Merged.MakeValid();
SELECT 'Merged polygon area: ', @Merged.STArea();

SELECT @Merged;
SELECT @Merged.ToString()

```

注意输出的变化:

```
POLYGON ((-3 -3, 3 -3, 3 3, -3 3, -3 -3), (-0.399999961853027344 -1, -1 -
0.399999961853027344, -1 0.399999961853027344, -0.399999961853027344 1,
0.399999961853027344 1, 1 0.399999961853027344, 1 -0.399999961853027344,
0.399999961853027344 -1, -0.399999961853027344 -1))
```

显示的结果与相对约整数有偏差, 这是 `MakeValid()` 命令的副产品, 不过, 除了此变化之外, 我们可以得到与我们的布置几乎相同的布局。

7.4.3 测量数据描述的实现——GEOGRAPHY 类型

实现测量、圆地球数据的类型被称之为 `GEOGRAPHY`。在绝大多数情况下, `GEOGRAPHY` 数据类型和 `GEOMETRY` 类型的工作方式一样(实际上, 它们共享了许多相同的函数)。和之前讨论过的最后两种数据类型一样, `GEOMETRY` 是用 CLR 用户定义函数实现的。

`GEOGRAPHY` 类型也可以接受之前讨论过的几何类型, 不过它还应用了半球的概念。

注意:

虽然几何数据类型将默认的 `SRID` 应用在空间实例上(默认值为 0), `GEOGRAPHY` 数据类型一般没有默认值(有些地理方法假设 `SRID` 值为 4326), 而且每次重新定义地理实例的时候必须提供它。

首先利用最近的第一个几何学例子的大致副本, 这次只使用 `GEOGRAPHY` 类型:

```
DECLARE @First GEOGRAPHY;

SET @First = GEOGRAPHY::STGeomFromText('LINESTRING(-3 3, 3 3, 3 -3, -3 -3,
-33)', 4326)
SET @First = GEOGRAPHY::Parse('LINESTRING(-3 3, 3 3, 3 -3, -3 -3, -3 3)')
SET @First = 'LINESTRING(-3 3, 3 3, 3 -3, -3 -3, -3 3)'

SELECT @First;
```

它们都很好, 只是 `STGeomFromText()` 函数的工作方式不同于与其对应的几何函数(唯一的区别是它不使用默认值, 需要第二个参数)。

在处理多边形的时候, 事情变得更有意思了, 因为必须将它放在一个给定的半球内。根据字典上的定义, 半球只是球的一半。每个半球的起点和终点会随着自己引用的 `SRID` 的不同而变化, 不过不管你选中了什么, 某个空间实例引用的所有多边形、线和点都必须能放在那个半球内。

提示:

我可以想象会有人提出这样的问题“为什么?” 我知道这个问题是要我回答的。问题的答案是消除所谓的“内部”对“外部”多边形上的模糊性。有函数查看空间实例中是否包含某些内容, 不过如果你不知道定义的环的哪一侧可以被视为内部的(而不是外部的), 那么你怎么可能知道对象内部是否包含某些内容呢?

当然, 解决空间数据的内部和外部问题的方法一般不止一个, 不过 `SQL Server` 团队必须选择一种方法, 而且他们采用的方法要求不超出一个半球的范围。如果必须映射的对象已经超过了半球边界, 那么考虑将其作为两个相邻的对象进行映射(共享半球边界), 并成对使用它们。

为了予以验证, 这里会继续研究下面的例子, 它和之前使用过的几何学例子基本相同, 不过

它被映射到弯曲感知数据类型(GEOGRAPHY)上:

```
DECLARE @First GEOGRAPHY;

SET @First = 'POLYGON((-3 3, 3 3, 3 -3, -3 -3, -3 3))';

SELECT @First;
```

不过, 尝试执行它的时候, 就会遇到自己在 GEOGRAPHY 数据类型中不会遇到的麻烦:

```
Msg 6522, Level 16, State 1, Line 3
A .NET Framework error occurred during execution of user-defined routine or
aggregate "geography":
Microsoft.SqlServer.Types.GLArgumentException: 24205: The specified input does
not represent a valid geography instance because it exceeds a single
hemisphere. Each geography instance must fit inside a single hemisphere. A
common reason for this error is that a polygon has the wrong ring orientation.
Microsoft.SqlServer.Types.GLArgumentException:
at Microsoft.SqlServer.Types.GLNativeMethods.ThrowExceptionForHr(GL_HResult
errorCode)
at Microsoft.SqlServer.Types.GLNativeMethods.GeodeticIsValid(GeoData g)
at Microsoft.SqlServer.Types.SqlGeography.IsValidExpensive()
at Microsoft.SqlServer.Types.SqlGeography.ConstructGeographyFromUserInput
(GeoData g, Int32 srid)
at Microsoft.SqlServer.Types.SqlGeography.GeographyFromText(OpenGisType type,
SqlChars taggedText, Int32 srid)
at Microsoft.SqlServer.Types.SqlGeography.Parse(SqlString s)
.

(1 row(s) affected)
```

.NET 实现的结果是额外的错误行堆栈, .NET 实现是本章提到的所有新数据类型所涉及的内容。不过, 关键项是 `GLArgumentException` 行; 我们在多个半球中。

我第一次开始学习半球问题的时候, 认为它必须与负数和正数有关——但事实并非如此。实际上, 问题并不是简单的测试这里的多边形内部是否可以被放置在一个半球内。我们已经定义了盒子, 它看起来非常简单而且很小, 所以可以方便地看到它在多个半球内为什么会引起混淆。不过, 问题也很简单。这里的内部和外部都是相反的。也就是说, 你所感知到的位于正方形“外部”可能被视为在正方形“内部”(因为这里已经为 SQL Server 定义了盒子)。

为了解决这个问题, 必须以绘制它的环的角度考虑多边形——即, 一系列起点和终点重合的连接线。画线的时候线的左边通常被认为是“内部”的。一般情况下, 这意味着在画对象的时候, 你希望按照逆时针方向绘制围绕它的线。在这里的例子中, 采用的是顺时针方向, 所以这里出现了一种情况, 其中“外部”就是这里的线界定的区域, 而且内部区域是没有边界的。只是改变一下这里绘制多边形的顺序就可以修复这个错误:

```
DECLARE @First GEOGRAPHY;

SET @First = 'POLYGON((-3 3, -3 -3, 3 -3, 3 3, -3 3))';

SELECT @First;
```

现在如果执行它，就会返回结果，同时这个结果看起来和处理 GEOMETRY 类型得到的结果差不多。

用 GEOMETRY 和 GEOGRAPHY 类型实现的一组方法之间有很大的重叠，不过它们也不完全相同(在本章中看到的所有方法都是用这两种类型实现的，除了 MakeValid())。空间数据是一个独立的研究领域，所以我建议要好好地研究 SQL Server 领域外的信息以便于理解每种实现期望的结果。

7.5 文件流

这是 SQL Server 2008 中新加入的一种极新颖的功能。事实上，它在本质上是额外的，而且甚至需要你完成特殊的步骤以便启用它(在默认安装中没有启用它)。还有，虽然我认为这个功能还不够稳固，不过它开辟了一条特殊之路。因此也带来了这样的一个问题：“文件流到底要完成什么工作？”。

在数据库领域，一直有很多和如何存储非结构化数据文件(例如，图像、文档、电子数据表和电影等)有关的问题。这些文件通常是存储在数据库中的大块数据的一个不可分割的部分(例如，一份保险索赔单上的损毁照片和扫描图像)。

在了解了这一点后，你可能想：

- 以一种节省空间的方式将所有数据存储在一起
- 用最大性能读写数据
- 利用事务
- 在一个模型下，并有效地确保数据安全
- 在备份和还原数据的时候保持一致性

解决这些问题的方法会根据哪个问题对于特定安装应作优先考虑而不同。下面给出了某种平衡行为：

- **性能是关键：**数据一般保存在文件系统级别的单独文件中；
- **一致性是关键：**数据一般都会存储在数据库中的二进制大型对象(blobs)中。通常使用文件组将 blob 存储在独立的驱动器阵列中。

具体情况会因安装而不同，这几年来虽然 SQL Server 处理 blob 的性能已经得到了很大的改进，不过一般的安装还是要将文件存储在文件的系统级而且只在 SQL Server 中存储文件路径，这样做有一定的风险，包括：

- 可以在数据库不知情的情况下移动文件，导致数据之间的链接断开，而且没有可用于恢复的历史记录。
- 在数据库没有直接意识到变化的情况下对文件进行更新，轻则会导致审核效率低下。
- 不能在同一事务中同时登记多个数据更改。这意味着可以覆盖一个文件，但回滚相关数据库的更改(反之亦然)，从而破坏数据的正常状态。
- 缺少协调事务，使得文件系统上的更改和数据库备份/恢复工作之间有了时间延迟。

其他的安装遵循 SQL Server 的 blob 路线，可以修正之前的问题，不过却会造成其他的一些问题：

- 存储效率低下，SQL Server 的页存储模型开销(与 blob 数据有关)会造成空间损失。

- 性能受损。一般情况下,性能的损伤会影响所有访问的数据,不只是 blob。
- 与其他数据库数据相比,从数据库访问 blob 数据需要完成特殊的处理——增加了复杂性。更有甚者,访问模型一般看起来比文件系统中文件的较简单流处理更加复杂。

通过将数据库和文件系统中的存储协调成一个内聚性的解决方案,SQL Server 中的文件流实际上可以解决所有的这些问题,令两种系统都只完成自己最擅长的工作(SQL Server 会协调事务和存储结构数据,而文件系统存储非结构数据)。

在文件流模型中,SQL Server 集成了 NTFS(Windows 使用的文件系统)。对于那些已经配置好要完成这些工作的表和列来说,定义为 varbinary(max)类型的列的数据被重定向到文件系统。SQL Server 中的访问相对透明,标准的 T-SQL 语句会对数据进行操作。不过,客户端语言可以利用从 .NET 中的 Stream 类派生的一种特殊的 SqlFileStream 对象,获得那些已经习惯于用 Stream 对象处理文件和作其他流访问的开发人员所熟悉的大部分功能。通过将 SQL Server 和 NTFS 中的最佳部分集成在一起,可以解决一些关键问题:

- **协调 SQL Server 和 NTFS 之间的安全性:** 只能在 SQL Server 授权的上下文中访问用来存储 SQL Server 文件流数据的目录。这意味着在 SQL Server 中没有适当权限访问 varbinary(max)列的人不能访问 NTFS 中的底层文件。
- **完全支持事务:** 任何活动的事务都全面注册了流更新(实际上为了获得数据的访问权限,使用文件流的客户端需要在事务上下文中登记),而且会酌情支持提交行为和回滚行为。这意味着更新现有的文件会酌情回滚,如果事务没有完成,就将文件还原到之前的状态。
- **协调备份:** 这意味着数据库备份包含 NTFS 处理文件(状态与剩余的备份数据一致)。
- **访问文件信息的方式与将文件直接存储在 NTFS 中时几乎完全相同:** 引入最少的协调开销,因此基本上可以忽略它与直接 NTFS 存储的性能之间的差别。

这很好地预示着非结构化数据在其他结构化环境中的前景。下面让我们快速地从开发的角度看一下其中的内容。

7.6 启用文件流

在默认情况下,在安装 SQL Server 的时候会关闭文件流访问。在安装过程中,有一个选项可以建立文件流访问,如果你还记得它的话,我推荐你使用这个选项。不过,如果你忘了(或者只是当时觉得自己不需要),可以使用 SQL Server 配置管理器为服务器启用文件流。找到 SQL Server 服务节点,右击自己实例的 SQL Server 服务(默认的实例是 MSSQLSERVER)。这样会弹出图 7-11 中的对话框(注意我已经切换到了 FILESTREAM 选项卡)。

提示:

安装 AdventureWorks2008 数据库需要为安装它的服务器开启文件流,所以如果你一直按照本书的说明运行至此,那么你一定已经为运行自己实例的服务器开启了文件流访问。虽说如此,你也许有时候要利用的是一个没有开

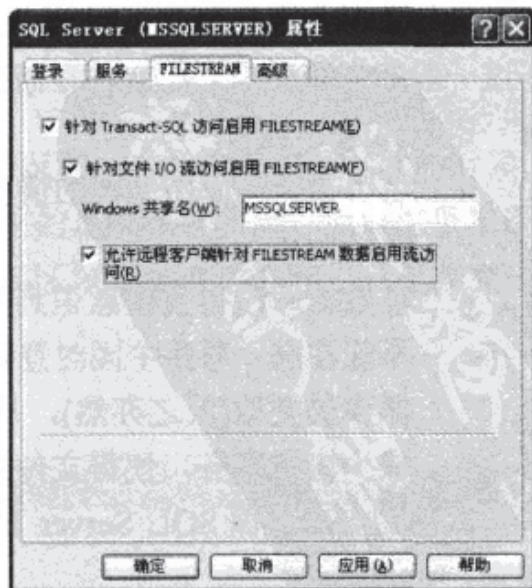


图 7-11

启文件流的系统，这样你就可以理解在事后启用它要涉及哪方面的知识了。

在这个对话框中，可以定义你希望文件流暴露的访问等级。请注意，这里是在针对服务器进行设置，数据库要存储流数据，需要对其作额外的配置。

7.6.1 为数据库启用文件流

只需要使用 `CONTAINS FILESTREAM` 选项创建一个文件组，就可以为数据库启用文件流。这会设置 SQL Server 访问控制下的路径，并将表配置为支持文件流访问。下面通过创建一个在这部分的例子中会使用到的数据库来检验一下这种方法：

```
CREATE DATABASE FileStreamDB
ON
PRIMARY ( NAME = FSDBPrimary, FILENAME = 'C:\FSDB\DB\fsdb.mdf'),
FILEGROUP FSDBStream CONTAINS FILESTREAM
( NAME = FSDBStream, FILENAME = 'C:\FSDB\STREAM')
LOG ON ( NAME = FSDBLog, FILENAME = 'C:\FSDB\fsdb.ldf')
GO
```

注意：

与在运行 `CREATE DATABASE` 语句时必须存在的数据和日志文件路径不同的是，用于文件流的文件组不得存在。SQL Server 创建了一个目录作为数据创建的一部分，在目录的使用权限和所有权方面与 NTFS 协调。

运行它(将文件路径改为在自己的特定系统上工作)就可以得到已经创建了自己的数据库的确认信息。

如果需要为现有的数据库启用文件流访问，可以使用 `ALTER DATABASE` 命令添加文件流分组。

7.6.2 创建一个启用文件流的表

为表启用文件流不需要什么特殊的设置。实际上，你只需要确定自己的表有一个定义为 `rowguidcol` 类型的唯一的约束列(这是一种特殊的数据类型，使用 `uniqueidentifier` 类型，不过也将其定义为 SQL Server 的行标识符)。此后，基于表中任意 `varbinary(max)` 列的选项，在每一列上定义文件流访问。

下面创建一个表，随后会用它在自己的 SQL Server 上存储对象：

```
CREATE TABLE FSTable
(
    FileKey int NOT NULL IDENTITY PRIMARY KEY,
    rowguid uniqueidentifier rowguidcol NOT NULL UNIQUE,
    filedata varbinary(max) FILESTREAM
);
```

同样地，它也会返回一条简单的确认信息，指出命令运行成功，不过创建了它，应该就可以对流数据进行操作了。

7.6.3 在 T-SQL 中使用文件流

文件流数据对 T-SQL 的访问相对透明。例如，可以运行一个简单的 INSERT 语句，就像之前对其他含有二进制数据的行所做的处理一样：

```

DECLARE @Ident int

INSERT FSTable
VALUES
    (NEWID(), 0x0A);

SET @Ident = @@IDENTITY;

SELECT FileKey, filedata
FROM FSTable
WHERE FileKey = @Ident;

UPDATE FSTable
SET filedata = 0x49276D206C6561726E696E672066696C6573747265616D73
WHERE FileKey = @Ident;

SELECT FileKey, filedata
FROM FSTable
WHERE FileKey = @Ident;

DELETE FSTable
WHERE FileKey = @Ident;

SELECT FileKey, filedata
FROM FSTable
WHERE FileKey = @Ident;

```

这样会搜索所有的 SQL 主语句：

```

(1 row(s) affected)
FileKey      filedata
-----
1            0x0A

(1 row(s) affected)

(1 row(s) affected)
FileKey      filedata
-----
1            0x49276D206C6561726E696E672066696C6573747265616D73

(1 row(s) affected)

(1 row(s) affected)

FileKey      filedata
-----

(0 row(s) affected)

```

可以看到，从 T-SQL 的角度来看，这实际上并没有什么。确实，所有的主要语句几乎和处理

非文件流数据的语句一样。在启用文件流的时候,使用添加到 `varbinary(max)` 数据类型的 `PathName()` 属性会看到一些额外的信息,例如:

```
DECLARE @Ident int;

INSERT FSTable
VALUES
    (NEWID(), 0x0A);

SELECT @Ident = @@IDENTITY;

SET @Ident = @@IDENTITY;

SELECT rowguid, fs.filedata.PathName() AS Path
FROM FSTable fs
WHERE FileKey = @Ident;
```

运行它,你会看到返回一行(很遗憾,数据行太宽了,所以本书不能很好地显示)。首先,注意 `rowguid` 列。现在将它和 `Path` 列的最后一部分进行比较,应该可以看到它们之间是匹配的。

正如你所看到的,在为自己的存储文件流设置唯一路径的时候,这里设定为 `rowguidcol` 的那一列是非常重要的。

7.6.4 在.NET 中使用文件流

在第25章讨论连接性时(仅限于 Web 版,所以不用立即跳到这一章),我会花费很多精力讨论.NET 与文件流。不过,我认为尽早理解一些关键的知识点是很重要的,因为它们有很多的设计分支,而且在你真正进入.NET 代码之前可能并不知道这些内容。

文件流的所有工作都需要事务上下文。即使你正在读数据,也需要 SQL Server 方的事务背景来控制自己数据的并发问题和一致性问题。遗憾的是,不能使用 T-SQL 关键字 `BEGIN TRANSACTION`, `BEGIN TRANSACTION` 不支持启用了多个活动结果集(MARS)的连接的一些规则。因此,必须在通过文件流访问数据之前,使用客户端数据访问 API 的登记事务方法。

除此之外,处理文件流相关的 SQL Server 和处理.NET 中更加通用的 `Stream` 对象之间显著的区别很可能是你遇到的情况之一(对 SQL Server 文件流来说, `SqlFileStream` 对象会透明地处理绝大多数区别)。

注意:

再次提醒,第25章 Web 是一个发布章节,而且我希望在本书发行过程中偶尔更新一下,使得它在一定程度上能够与不断变化的连接方法保持同步。

7.7 表压缩

注意:

截至编写本书时为止,SQL Server 的数据压缩特征仅限于企业版。

这也是新加入 SQL Server 2008 的, 不过在 SQL Server 2005 服务包中就已经暗示过它要出现了。从编程的角度来看, 这里要完成的工作很少(主要是关于表的设置)。不过出于下面的三种原因, 在这个“高级”数据结构章节中, 仍必要提一下:

- **规划:** 压缩特征会从根本上修改磁盘上数据的页/行存储格式, 而且可以大大减少自己数据的空间。这是以表-表的方式实现的(这也是一种表级别的设置), 因此需要调整自己对所需存储容量和数据库容量增长的规划。
- **性能:** 在评估表压缩的优缺点时, 必须平衡性能。这要视特定的场合而定。管理压缩需要额外的开销, 不过压缩可能还会大大减少 I/O 需求, 并弥补一些由压缩开销带来的性能损失。
- **结构知识:** 讲了这么多只是为了告诉你传统的页/行存储方法, 也许还应该看一下如何从本质上修改这些默认存储方法的内容。

启用压缩

在之前的一章中, 已经看到了 CREATE INDEX 语法。和 CREATE TABLE 一样, 这里也可以使用 DATA_COMPRESSION 选项。下面的代码中突出显示了 CREATE INDEX 的版本(与其在 CREATE TABLE 语句中的工作方式相同)。

```
CREATE [UNIQUE] [CLUSTERED|NONCLUSTERED]
INDEX <index name> ON <table or view name>(<column name> [ASC|DESC] [,...n])
INCLUDE (<column name> [, ...n])
[WITH
[PAD_INDEX = { ON | OFF }]
[[,] FILLFACTOR = <fillfactor>]
[[,] IGNORE_DUP_KEY = { ON | OFF }]
[[,] DROP_EXISTING = { ON | OFF }]
[[,] STATISTICS_NORECOMPUTE = { ON | OFF }]
[[,] SORT_IN_TEMPDB = { ON | OFF }]
[[,] ONLINE = { ON | OFF }
[[,] ALLOW_ROW_LOCKS = { ON | OFF }
[[,] ALLOW_PAGE_LOCKS = { ON | OFF }
[[,] DATA_COMPRESSION = { NONE | ROW | PAGE }
    [ ON PARTITIONS ( { <partition number expression> | <range> }
[[,] MAXDOP = <maximum degree of parallelism>
]
[ON {<filegroup> | <partition scheme name> | DEFAULT }]
```

之前提到过, 可以向 CREATE INDEX 语句中使用的语句添加同样一行, 将数据压缩作为 CREATE TABLE 语句的一部分启用。

7.8 小结

事实上, 在本章中看到的所有内容都是新加入 SQL Server 2008 的(要注意, XML 索引是个例外)。它们都是高度专业化的, 不过它们都可以很好地处理那些为特定任务优化的数据结构。

如果你正在处理 XML 数据, 请仔细考虑自己的索引, 试验索引并了解到它们可以大大提高

XML 的查询速度。对于层次数据来说，请考虑新的 HierarchyID 数据类型。它不但包括特定于层次的方法，而且对许多开发人员来说，给定节点知道自己的前后代关系的概念比一般父子方法需要的对层次结构的递归调用更长容易掌握。

这里的空间数据最终将 SQL Server 开发人员带入一种他们从未进入过的新世界。它们支持平地模型和圆地模型，而且识别邻近性、不规则形状、交点和类似的特定于空间的概念的能力可以极大地帮助那些还没有意识到自己有某种特别需要的人——更不用说构思如何满足那种需要的人了。

文件流满足了 SQL Server 中的一个长期需求。文件流支持的许多功能已经以某种其他的方式被支持很长时间了，不过文件流以一定的方式集成功能，允许更加协调地执行备份过程。而且更重要的是允许基于事务处理大的二进制文件。虽然文件流访问很大程度上只是一个客户端应用程序——只是过程，但是它需要数据库架构师完成很多细节上的设计和安全性的考虑。

这里的数据压缩是数据库级别的。虽然压缩对应用程序来说是透明的，但是压缩可以提高和破坏系统性能，而且在激活压缩功能之前必须认真考虑它。

下一章会研究 SQL Server 的支柱——视图。



第 8 章

视 图

截至目前，本书一直认为读者已经对 SQL Server 有所了解，所以下面将简化对基本概念的讨论，重点讨论视图的简洁使用方法。即在继续新内容之前，首先要简要回顾一下视图的基本概念。

对视图的使用往往不是过多就是过少——很少使用得恰到好处。在学习完本章内容以后，就可以使用视图完成下面的工作：

- 更加熟悉视图的基本概念；
- 向数据库添加额外的索引以加快查询的执行——即使不使用索引所基于的视图；
- 理解并利用分区视图和联合服务器的概念。

实际上，视图是一种存储查询。可以创建一个简单查询，只从一个列表中选择，并忽略一些列，或者创建一个复杂的查询来联结多个表使它们看上去像是一个表。

8.1 回顾视图语法

视图的基本语法如下所示：

```
CREATE VIEW <view name>
AS
<SELECT statement>
```

它利用绝大多数 SQL Server 对象中存在的 CREATE <对象类型><对象名>基本语法。这是最简单的语法，不过在绝大多数情况下，只需要使用这种形式。下面给出了扩展后的语法：

```
CREATE VIEW [<schema name>].<view name> [(<column name list>)]
[WITH [ENCRYPTION] [, SCHEMABINDING] [, VIEW_METADATA]]
AS
<SELECT statement>
[WITH CHECK OPTION]
```

下面给出了 AdventureWorks2008 数据库中 Person.Person 表上的一个极其简单的视图：

```
USE AdventureWorks2008;
GO

CREATE VIEW Person.PersonView
```



```
AS
SELECT FirstName, MiddleName, LastName
FROM Person.Person;
```

运行下面的代码:

```
SELECT * FROM Person.PersonView;
```

得到的结果与执行下面的代码一致:

```
SELECT FirstName, MiddleName, LastName
FROM Person.Person;
```

实际上这是在对 SQL Server 说:“在运行语句 `SELECT FirstName, MiddleName, LastName FROM Person.Person` 的时候请将所有获得的行和列给我。”

这里创建了某种穿越(pass-through)状况——也就是说,视图其实没有改变任何事情,只是对访问的数据的过滤版本进行了某种形式的穿越。考虑一下,应该可以看出如何利用此概念为缺乏经验的用户简化数据(为了避免引起混淆,这里只显示他们关心的数据)或者不授权用户访问底层表,但授权他们访问不包含敏感数据的视图,从而提前隐藏敏感数据(如利润或薪水数)。

注意:

请记住,在默认情况下,不需要对视图进行特殊处理。运行视图就像从命令行中运行查询一样——不存在任何形式的预先优化。这意味着你在请求数据和传递数据之间加入了又一层开销。这表明如果直接运行 `SELECT` 语句,视图就会运行得更快。也就是说,视图的存在是有原因的(用户的安全性和简化),平衡自己的需求和开销以适应自己的特定情况。

开销有多大?这既与视图的复杂程度有关,又与调用代码有关。根据具体的细节不同,耗费的时间可以是几毫秒或更长(虽然通常都是几毫秒)。

下面继续深入下去。

前面已经讲过了创建简单视图的方法——只需使用简单的 `SELECT` 语句。如何对查询结果进行筛选呢?答案是使用 `WHERE` 子句。视图没什么特别之处。

8.2 更复杂的视图

或许,视图最常见的用途是扁平化数据——即去掉在本章开始处指出的复杂性。假设要提供一个管理视图,以便于检查销售信息。我无意冒犯那些阅读本书的经理们,不过很少有经理可以自己写出复杂的查询——即便在当今这个信息时代。

例如,我们的经理可能需要做些简单的查询,告知他(她)下了什么订单、订购了什么商品、每张订单上售出了多少商品以及相关的价格信息。因此,可以创建一个视图,他(她)这样就可以在视图上执行非常简单的查询:

```
USE AdventureWorks2008;
GO

CREATE VIEW CustomerOrders_vw
AS
```

```

SELECT o.SalesOrderID,
       o.OrderDate,
       od.ProductID,
       p.Name,
       od.OrderQty,
       od.UnitPrice,
       od.LineTotal
FROM Sales.SalesOrderHeader AS o
JOIN Sales.SalesOrderDetail AS od
     ON o.SalesOrderID = od.SalesOrderID
JOIN Production.Product AS p
     ON od.ProductID = p.ProductID;

```

现在执行 SELECT 语句:

```

SELECT *
FROM CustomerOrders_vw;

```

执行后会得到很多行(超过 100 000 行),但是也会得到这样的信息,一般经理都可以很方便地理解和挑选这些信息。再者,无需过多培训,经理(或者可能的任何用户)就能直接把握住自己所要找寻的问题的实质。

```

SELECT ProductID, OrderQty, LineTotal
FROM CustomerOrders_vw
WHERE OrderDate = '5/15/2003';

```

用户不必了解如何对 4 个表进行联结——这隐藏在视图中。实际上,他(她)只需要具备很少的技能(以及有限的想象力)就可以完成自己的工作。

ProductID	OrderQty	LineTotal
791	1	2443.350000
781	1	2071.419600
794	1	2181.562500
798	1	1000.437500
783	1	2049.098200
801	1	1000.437500
784	1	2049.098200
779	1	2071.419600
797	1	1000.437500

(9 row(s) affected)

不过这里可以让查询更有针对性。假设希望视图只返回昨天的销售信息。这里要稍微修改一下查询:

```

USE AdventureWorks2008;
GO

CREATE VIEW YesterdaysCustomerOrders_vw
AS
SELECT o.SalesOrderID,
       o.OrderDate,
       od.ProductID,

```



```

        p.Name,
        od.OrderQty,
        od.UnitPrice,
        od.LineTotal
FROM Sales.SalesOrderHeader AS o
JOIN Sales.SalesOrderDetail AS od
    ON o.SalesOrderID = od.SalesOrderID
JOIN Production.Product AS p
    ON od.ProductID = p.ProductID
WHERE CONVERT(varchar(12),o.OrderDate,101) =
        CONVERT(varchar(12),DATEADD(day,-1,GETDATE()),101)

```

由于 AdventureWorks 数据库中的所有日期都比较陈旧, 所以视图不会返回任何数据, 因此, 为了进行测试, 我们加入一行数据。一次执行下面所有的脚本:

```

USE AdventureWorks2008;

DECLARE @Ident int;

INSERT INTO Sales.SalesOrderHeader
(
    CustomerID,
    OrderDate,
    DueDate,
    BillToAddressID,
    ShipToAddressID,
    ShipMethodID
)
VALUES
(
    1, -- CustomerID
    DATEADD(day,-1,GETDATE()), -- OrderDate (Yesterday)
    GETDATE(), -- Due Date (today)
    1, -- BillToAddressID
    1, -- ShipToAddressID
    1 -- ShipMethodID
);

SELECT @Ident = @@IDENTITY;

INSERT INTO Sales.SalesOrderDetail
(
    SalesOrderID,
    OrderQty,
    ProductID,
    SpecialOfferID,
    UnitPrice,
    UnitPriceDiscount
)
VALUES
(@Ident, 4, 765, 1, 50, 0);

SELECT 'The OrderID of the INSERTed row is ' + CONVERT(varchar(8),@Ident);

```

对初学者来说, 上面的脚本中完成的大多数操作并不神秘, 不过第 9 章会说明这里发生的一切。现在只需要相信, 为了在 AdventureWorks 2008 中有一个能够出现在视图中的值, 必须运行

所有的脚本。在 Management Studio 中应该可以看到类似下面的运行结果：

```
(1 row(s) affected)
```

```
(1 row(s) affected)
```

```
-----
The OrderID of the INSERTed row is 75124
```

```
(1 row(s) affected)
```

提示：

要知道，如果使用的是 Management Studio 的“以网格显示结果”模式，那么，显示在上面代码中的一些信息将只出现在消息选项卡中。还要记住，你的 OrderID 可能与我的略有不同，这取决于你在 AdventureWorks2008 数据库中已经完成了哪些试验。

SalesOrderID 可能会变化，不过其余部分应该都差不多。

现在，在视图上运行查询看看会得到什么结果：

```
SELECT SalesOrderID, OrderDate FROM YesterdaysCustomerOrders_vw
```

会看到确实显示出了 75124：

```
SalesOrderID OrderDate
```

```
-----
75124          2008-12-31 01:00:00.000
```

```
(1 row(s) affected)
```

注意：

不要被自己的 SalesOrderID 号要和我的一样这种说法所迷惑——这是由系统设置的(因为 SalesOrderID 是一种标识列)，而且与已经向表插入了多少行有关。因此，你的编号可能会有所不同。

8.2.1 使用视图修改数据——在 INSTEAD OF 触发器之前

正如前面说过的，从使用的观点来看，视图和表的工作方式基本上是一样的(显然，创建它们的方式差别很大)。不过，现在会遇到一些不同之处。

很多人可能会觉得可以顺利地视图上运行 INSERT、UPDATE 和 DELETE 语句是很奇怪的。不过，在通过视图修改数据时，必须牢记以下几点：

- 如果视图包含联结，除非使用 INSTEAD OF 触发器，否则在大多数情况下不能插入(INSERT)或删除(DELETE)数据。在某些情况下(只更新来自一个表的列时)，不需要 INSTEAD OF 触发器就可以进行 UPDATE，只需要做一些计划，否则很快就会出现问題。
- 如果视图只引用一个表，那么不需要使用 INSTEAD OF 触发器就可以使用视图插入数据(假设表中所有必需的字段暴露在视图中或者具有默认值)。即使是对于单个表的视图，如果没有默认值的列未显露在视图中，那么只有使用 INSTEAD OF 触发器才允许执行 INSERT 操作。
- 可以在一定程度上限制在视图中插入或不插入的内容。

这里已经多次提到 INSTEAD OF 触发器了。INSTEAD OF 触发器是一种特殊的、相当复杂的触发器，我们将在第 12 章中详细地研究它。这里的问题是我们还没有深入讨论过触发器。与在 SQL Server 中经常会发生的事情一样，我们陷入某种古老的鸡与蛋的问题之中（“先有鸡还是先有蛋？”）。因为 INSTEAD OF 触发器与视图有关，所以这里必须讨论 INSTEAD OF 触发器，不过又由于触发器是创建在表和视图上的，所以在谈论 INSTEAD OF 触发器的话题之前，我们必须先理解这两个对象。

本章会按照视图惯常的方式处理问题——在使用 INSTEAD OF 触发器这类事情之前。尽管本章不会涉及 INSTEAD OF 触发器的细节，但我们必须了解何时一定要使用它们。在第 12 章中讨论 INSTEAD OF 触发器的时候，我们会回过头来更全面地处理这里的问题。

提示：

我曾经说过，这里会给出一些相关背景——INSTEAD OF 触发器是一种特殊的触发器，本质上，它“替代”导致触发器触发的语句运行。结果是它能够看到语句原本要完成的工作，随后做出正确的决定，解决期间可能出现的所有冲突和其他问题。它的功能非常强大，不过也相当复杂，这就是现在暂时不谈论它的原因。

1. 修改有联结数据的视图

如果视图涉及的表不止一个，那么在很多情况下，使用视图修改数据已经过时了（可能是这样吧），除非使用 INSTEAD OF 触发器。它在键布局上造成了一些混乱，在默认情况下，在有多个表时微软会禁止这样做。要解决这个问题，可以使用 INSTEAD OF 触发器来检查修改的数据，并明确告知 SQL Server 你要用它做什么。

2. 需要的字段必须出现在视图中或者必须有默认值

在默认情况下，如果要使用视图插入数据（在底层查询中必须是一个单表的 SELECT 语句，或者至少限制插入只影响一个表并显示所有需要的列），必须能够提供所有必需的字段的一些值（不允许为 NULL 的字段）。注意“提供一些值”并不表示要在 SELECT 表中列出这些值——默认很好地涵盖了清单。不过要清楚，为了通过视图进行插入（INSERT），视图中必须显示所有没有默认值的列和所有不接收 NULL 值的列。使用 INSTEAD OF 触发器是摆脱这种问题的唯一办法（你猜到了）。

3. 限制插入视图的内容——WITH CHECK OPTION

与那些完全不为人知的 SQL Server 功能相比，WITH CHECK OPTION 是一种鲜为人知的功能。规则很简单——为了使用视图更新或插入数据，必须保证在视图结果集中出现最终的行。再次重申，插入或更新的行必须满足在视图底层的 SELECT 语句中使用的 WHERE 标准。

8.3 使用 T-SQL 编辑视图

使用 T-SQL 编辑视图时，必须要记住你正在替代整个现有视图。使用 ALTER VIEW 语句和 CREATE VIEW 语句之间仅有的差别是：

- ALTER VIEW 期望找到一个现有的视图，而 CREATE 不是这样；

- ALTER VIEW 要保留所有建立在该视图上的许可权限;
- ALTER VIEW 保留所有的依赖信息。

上面的所有差别中,第二点是最重要的。如果你要执行 DROP 然后再使用 CREATE 语句,这样做的效果与 ALTER VIEW 语句的效果几乎相同。问题是你必须彻底重新创建可以使用视图和不可以使用视图的权限。

8.4 删除视图

没有比这更容易的了:

```
DROP VIEW <view name>, [<view name>,[ ...n]]
```

执行这条语句,可以删除视图。

8.5 审核:显示现有代码

如果你有一个视图却不确定它在做什么,那应该怎么做呢?这时的第一个选择是很容易的——就像要编辑视图一样,直接进入 Management Studio 中。打开“视图”子节点,选择要编辑的视图,右击后,选择设计或脚本视图,随后是需要的特定脚本类型,你会发现视图背后的代码是用不同的颜色分别显示的。

提示:

注意设计功能带来了一种特殊的视图生成器实用工具。虽然对那些缺乏 SQL 经验的人来说,视图生成器效果不错(它类似于 Access 中的一种工具),不过我发现我希望视图形成的格式影响范围过大了,并不可避免地产生了一个更加复杂的视图(因此更难懂),因此,我一般坚持依靠脚本工具和自己编写 SQL 的能力。

遗憾的是,不能一直选择让 Management Studio 帮助我们解决这个问题(可能现在使用的是某种轻型工具,或者我们需要在自己的应用程序中植入实际的请求)。好的一面是得到实际视图定义的方式有几种:

- sp_helptext
- OBJECT_DEFINITION() 系统函数
- sys.comments 系统视图

下面在 AdventureWorks2008 数据库中的某个提供的视图上运行 sp_helptext——vStateProvince-CountryRegion:

```
EXEC sp_helptext 'Person.vStateProvinceCountryRegion';
```

提示:

注意那些引号。这是因为此存储过程只需要一个参数,而句号是某种分隔符——如果不使用引号就传递 Person.vStateProvinceCountryRegion,那么在遇到句号的时候,它就不知道该如何处理了,并因此产生错误。如果视图在默认模式下,就可以只提供视图名(没有模式名)而且不需要把

它放在引号内。

SQL Server 向我们显示了视图的代码：

Text

```
-----

CREATE VIEW [Person].[vStateProvinceCountryRegion]
WITH SCHEMABINDING
AS
SELECT
    sp.[StateProvinceID]
    ,sp.[StateProvinceCode]
    ,sp.[IsOnlyStateProvinceFlag]
    ,sp.[Name] AS [StateProvinceName]
    ,sp.[TerritoryID]
    ,cr.[CountryRegionCode]
    ,cr.[Name] AS [CountryRegionName]
FROM [Person].[StateProvince] sp
    INNER JOIN [Person].[CountryRegion] cr
    ON sp.[CountryRegionCode] = cr.[CountryRegionCode];
```

现在 `sp_helptext` 很好了，不过此时我把它视为陈旧的事物。为什么？既然 `sp_helptext` 是一个存储过程，那么你就不能轻易地把结果集看作更复杂的数据操作。幸运的是，Microsoft 为我们提供了 `OBJECT_DEFINITION()` 处理这个问题。

`OBJECT_DEFINITION()` 应该是你的首选，原因如下：

- 出现新版本的时候，它会根据系统表的变化自动更新(这样你就不用为此担心了)；
- 可以在更多查询中使用返回值(例如，作为一行，返回许多对象的源代码)。

语法如下所示：

```
OBJECT_DEFINITION(<object id>)
```

这样做的负面影响是：不完成特殊的查找就不知道自己的对象 id。幸运的是，SQL Server 为我们提供了一种简单的方法，也就是使用 `OBJECT_ID()` 函数查找对象 id。例如，如果要使用 `OBJECT_DEFINITION()` 获得之前看到的同一视图的代码，我们应该编写下面的代码：

```
SELECT OBJECT_DEFINITION (OBJECT_ID(N'Person.vStateProvinceCountryRegion'));
```

对象 ID 是 SQL Server 跟踪事物的内部方法。它们是整数值而不是可供对象使用的名称。一般来说，它们超出了本书的涵盖范围，不过最好能够认识到它们的存在，因为你会发现你从其他人那里拷贝来的脚本使用了它们，或者随后在学习 SQL 的时候会遇到它们。

尝试一下，你会发现结果与使用 `sp_helptext` 的结果相同(除非在查询定义中提供一个别名，否则它不会为列命名)。

继续下去可以方便地返回数据库中每个视图的代码：

```
SELECT '-----', OBJECT_DEFINITION(so.object_id)
FROM sys.objects so
WHERE so.type = 'V';
```

提示:

为了避免占用过多篇幅,本书在这里忽略了这些结果——它们很长。也就是说,运行之前的查询应该会显示 AdventureWorks2008 数据库中的所有视图。

不使用游标而使用 `sp_helptext` 不能完成这个任务——可以轻易地观察到系统函数 `OBJECT_DEFINITION` 和之前版本的 SQL Server 中的系统存储过程对象的有效性。

现在尝试最后一种方法——使用 `sys.comments`。

提示:

你可能会看到与老版本交替使用的 `sys.comments`(系统视图),而非不如人意的 `syscomments`(系统表)。`syscomments` 是一个系统表,向我们提供之前版本的 SQL Server 的大多数系统信息。微软多年来一直试图不要我们直接调用系统表,而且它们最终为我们提供了一系列符合其愿望的工具。

因为微软已经警告过我们系统表随时都可能改变(即使是服务包,即使我从未看到这种情况),所以即使系统表是获得系统信息的唯一可直接查询的方式,使用它们还是非常危险的。现在微软提供了系统模式的视图以及许多元数据的表值函数(细节请参考附录 B),直接针对系统表其实是很傻的。我强烈建议直接移植那些访问系统表的旧代码,以便于利用等价视图(通常只需要在旧系统表名称中的“sys”后面加上一个句号就可以了)。

`sys.comments` 为自己的基本源代码提供一个实际视图,而且如果你选择这样做的话,还可以提供一些直接可以插入的东西。类似于 `OBJECT_DEFINITION()`,任何使用 `sys.comments` 的方法都需要你知道自己的对象 id。你可以像我在前一例子中做的一样,加入到 `sys.objects` 系统视图,或者像在更早的例子中那样利用 `OBJECT_ID()` 函数。不过要注意,使用 `sys.objects` 的时候,必须分别处理对象名称和模板名称(这意味着还必须调用 `sys.schemas` 系统视图)。例如:

```
SELECT sc.text
FROM sys.syscomments sc
JOIN sys.objects so
    ON sc.id = so.object_id
JOIN sys.schemas ss
    ON so.schema_id = ss.schema_id
WHERE so.name = 'vStateProvinceCountryRegion'
    AND ss.name = 'Person';
```

再次声明,这样做得到的代码块与你在前两个方法中看到的一样。

8.6 保护代码:加密视图

如果你正在构建任何类型的商业软件产品,偶尔可能会对保护源代码感兴趣。你要做的只是用 `WITH ENCRYPTION` 选项加密自己的视图(和绝大多数其他形式的服务器存储代码)。如果你习惯了使用 `WITH CHECK OPTION` 子句,那么下面有一些技巧要记住:

- `WITH ENCRYPTION` 必须在视图名之后,但在 `AS` 关键字之前,
- `WITH ENCRYPTION` 不使用 `OPTION` 关键字。

此外要记住,如果使用了 `ALTER VIEW` 语句,要替换的是整个现有的视图(除了访问权限)。这意味着加密也同样会被取代。必须使用 `ALTER VIEW` 语句中的 `WITH ENCRYPTION` 子句对更改的视图进行加密。

接下来, 对本章前面创建的 CustomerOrders_vw 视图执行 ALTER VIEW。如果还没有创建 CustomerOrders_vw 视图, 只需要把 ALTER 更改为 CREATE(不要忘了在 AdventureWorks 上运行):

```
ALTER VIEW CustomerOrders_vw
WITH ENCRYPTION
```

```
AS
SELECT  o.SalesOrderID,
        o.OrderDate,
        od.ProductID,
        p.Name,
        od.OrderQty,
        od.UnitPrice,
        od.LineTotal
FROM Sales.SalesOrderHeader AS o
JOIN Sales.SalesOrderDetail AS od
    ON o.SalesOrderID = od.SalesOrderID
JOIN Production.Product AS p
    ON od.ProductID = p.ProductID;
```

现在在 CustomerOrders_vw 上应用 sp_helptext:

```
EXEC sp_helptext CustomerOrders_vw;
```

SQL Server 会立刻告知我们它无法完成我们的要求:

```
The text for object 'CustomerOrders_vw' is encrypted.
```

你心有不甘, 很快就看到 sys.comments 视图:

```
SELECT sc.text
FROM syscomments sc
JOIN sys.objects so
    ON sc.id = so.object_id
JOIN sys.schemas ss
    ON so.schema_id = ss.schema_id
WHERE so.name = 'CustomerOrders_vw'
AND ss.name = 'dbo';
```

不过这样也并不会会有太大的改变——SQL Server 可以识别那些被加密的表并且返回一个 NULL 结果。

简言之——你的代码是安全可靠的。即使用其他的读取器进行提取(例如, Management Studio 实际上在加密表上不会提供设计选项), 你也会发现它并无用处。

注意:

在使用 WITH ENCRYPTION 选项之前, 请确保自己已经保存好源代码。一旦完成加密, 就不能回头了。如果没有把代码保存在其他地方而且又需要修改代码, 你可能就会发现自己要从头写代码了。

8.7 关于模式绑定

本质上, 模式绑定(schema binding)把视图所依赖的东西(表或其他视图)“绑定”到视图上。

这样做的意义在于，除非先删除使用了模式绑定视图，否则没人可以修改那些对象(CREATE、ALTER)。

为什么要这么做呢？其中的原因包括：

- 它防止在对底层对象进行修改时“孤立”视图。设想一下，某个人完成了一条 DROP 语句或做了其他修改(即使删除列也可能会给视图带来麻烦)，却没有注意到你的视图。这样实在太糟糕了。如果视图使用了绑定模式，那么就不会发生这种事情了。
- 允许索引视图。必须用 SCHEMABINDING 选项在自己的视图上建立索引(后面会介绍索引视图)。
- 如果要创建引用视图的模式绑定用户定义函数(而且有用户定义函数必须采用模式绑定的实例)，那么你的视图也必须是模式绑定的。

在重建视图时，要紧记这些。

8.8 使用 VIEW_METADATA 使自己的视图看起来像一个表

这个选项可以使 DB-LIB、ODBC 和 OLE-DB 客户觉得视图是一个真正的表。不使用这个选项，交回给客户端 API 的元数据就是视图依赖的基本表。

要更新任何客户端游标(客户端应用程序管理的游标)，必须提供元数据信息。注意，如果要支持这样的游标，还需要使用 INSTEAD OF 触发器。

8.9 索引(物化)视图

在 SQL Server 2000 中，只有企业版支持索引视图(虽然开发版和评估版也支持，但不允许在生产系统中使用测试和开发版本)。不过从 SQL Server 2005 开始，所有的版本都支持索引视图。

引用视图时，实质上调用查询中并入了构成视图的查询中的逻辑。遗憾的是，这意味着调用的查询变得更加复杂。实际上，动态计算视图(以及视图所代表的的数据)带来的影响的额外开销可能变得非常高；再者，引用视图时，常常以组成视图中联结表的形式向查询添加额外的联结。索引视图提供了在运行查询前处理这类问题的方法。

索引视图实质上是把一组唯一值“物化”为聚集索引形式的视图。索引视图的优点是，通过把视图背后的信息收集到一起，提供非常快速的查找。在创建了第一个索引(必须是在一组唯一的值上的聚集索引)之后，SQL Server 也能用第一个索引中的聚集键作为引用点，在视图上创建其他的索引。尽管如此，任何事情都是有限度的——关于何时能或不能在视图上创建索引有一些限制(希望你心理准备——这些限制列出来有很长一串！)：

- 视图必须使用 SCHEMABINDING 选项；
- 如果引用任何用户定义函数(本书后面会详细描述这部分内容)，那么这些函数也必须是模式绑定的；
- 视图不能引用任何其他视图——只能引用表和用户定义函数；
- 视图中引用的所有表和用户定义函数必须使用两部分(甚至不能是三部分或四部分名称)命名约定(例如，dbo.Customers、BillyBob.SomeUDF)，并且必须与视图具有相同的所有者；

- 视图引用的所有对象必须与视图同在一个数据库中;
- 创建视图和所有基本表的时候必须启用 ANSI_NULLS 和 QUOTED_IDENTIFIER 选项;
- 视图引用的所有函数必须是确定的。

接下来给出一个创建索引视图的例子, 首先回顾本章前面创建的 CustomerOrders_vw 对象。这里会使用在加密一节中用过的 ALTER 语句, 不过, 只要在本章很早的时候创建的最初的版本中正确地加入了 WITH SCHEMABINDING, 做起来也是很容易的。

```
ALTER VIEW CustomerOrders_vw
WITH SCHEMABINDING
AS
SELECT o.SalesOrderID,
       o.OrderDate,
       od.ProductID,
       p.Name,
       od.OrderQty,
       od.UnitPrice,
       od.LineTotal
FROM Sales.SalesOrderHeader AS o
JOIN Sales.SalesOrderDetail AS od
  ON o.SalesOrderID = od.SalesOrderID
JOIN Production.Product AS p
  ON od.ProductID = p.ProductID;
```

这里, 要注意以下几点:

- 必须使用 SCHEMABINDING 选项创建自己的视图。
- 为了使用 SCHEMABINDING 选项, 引用的对象(此处是指所有的表)必须使用两部分命名(这里我们是这样做的, 不过并非所有的视图都是如此设置的)。

其实这只是开始——我们还没有索引视图。目前已经有一个能够被索引的视图。创建索引的时候, 在视图上创建的第一个索引必须既是聚集的又是唯一的。

```
CREATE UNIQUE CLUSTERED INDEX ivCustomerOrders
ON CustomerOrders_vw(SalesOrderID, ProductID, Name);
```

一旦执行了上面的命令, 就可以得到一个索引视图。尽管如此, 很快还会出现一个小问题。下面在视图上运行一个简单的 SELECT 来进行测试:

```
SELECT * FROM CustomerOrders_vw;
```

如果执行上面的命令, 会看到如图 8-1 所示的显示计划(“显示估计的查询计划”是一个工具提示, 你可以在工具栏的中间找到它, 也可以在菜单“查询”|“显示估计的查询计划”中找到它), 图中指出使用新索引。

注意:

即使没有显式使用视图, SQL Server 可能也会利用支持索引视图的索引。例如, 如果你要完成的联结类似于那些索引视图支持的联结, SQL Server 可能会识别并利用索引。

查询 1: (与该批有关的) 查询开销: 100%

SELECT * FROM CustomerOrders_vw;

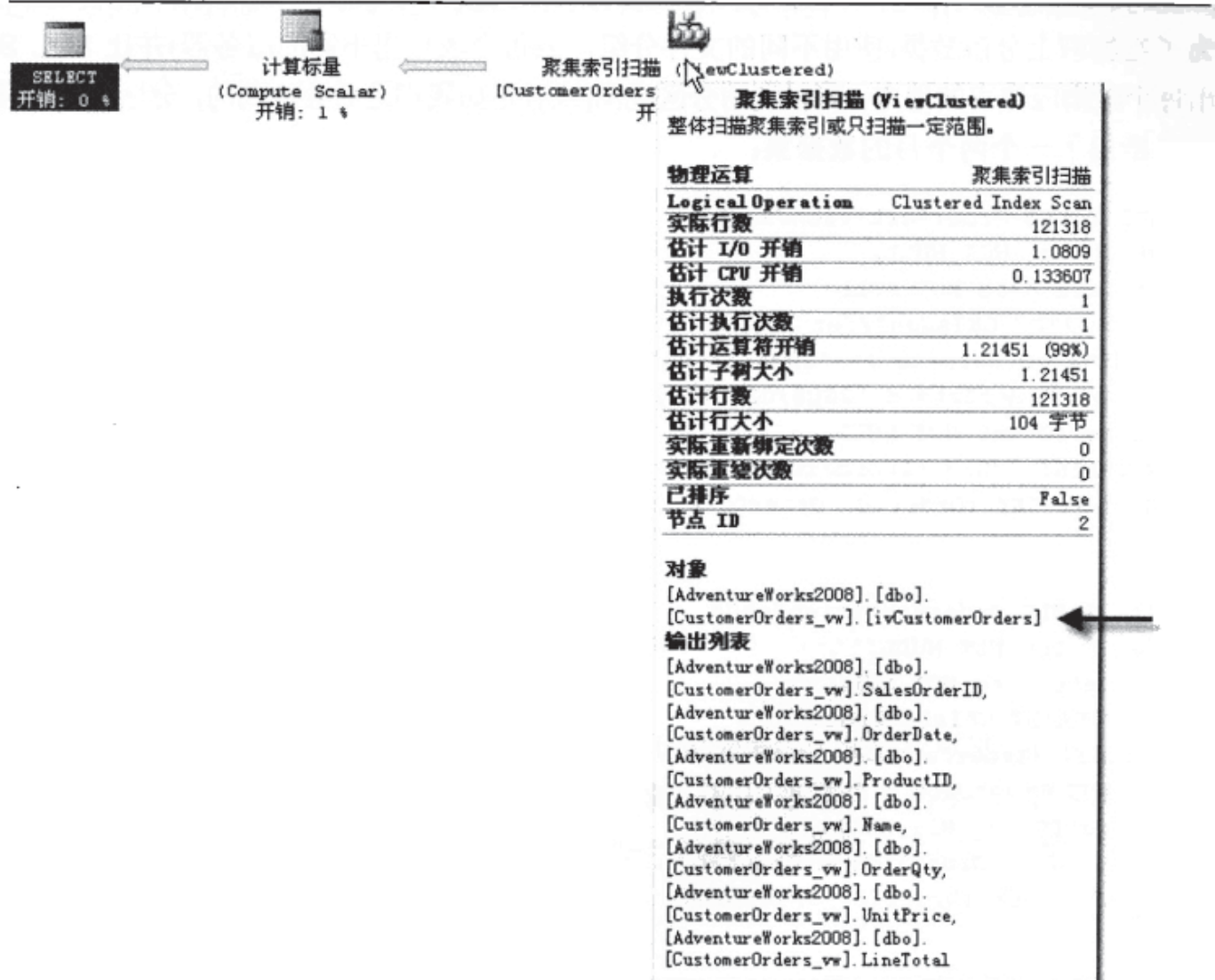


图 8-1

8.10 分区视图

虽然从 SQL Server 2000 开始才使用分区视图，不过微软从 2005 年就开始考虑将分区表视为首选的分区方法。这里提到它们是因为这是微软多年来力推的主要可扩展选项之一，而且你必须在遗留代码中遇到它们的时候了解它们的工作方式。此外，有一些利用分区表难以解决甚至无法解决的分区问题，所以最好了解其他的选项。

分区视图将多个同样的表(按照结构，而非实际数据)统一在一起，使它们看起来如同一个表一样。起初，似乎用简单的 UNION 子句就可以轻易完成这个任务，不过在处理插入和更新情形的时候，这些概念就变得有些棘手了。

使用分区视图时，可以在视图中的一个表上定义约束。随后，在第二个(也可能更多的)表上定义一个相似的、但互相排斥的约束。在创建一个视图统一这些相互排斥的表的时候，SQL Server 能够以合理的方式挑拣出表的唯一特点。这样，SQL Server 就可以准确地确定从哪一个表中获取新数据(通过确定哪个表可以接收数据——如果按照需要把它们创建为互斥的表，那么就可以将数据放入一个表中而不会有冲突)。唯一注意的是所谓的“分区列”必须参与到主键中去。下面创建自己的小例子看一看它是如何工作的。

此时想象一下自己正在运行一个非常大的 Internet 站点，而且每天要接受数千个订单。你的 Orders 表越来越大，而且在运行 DELETE 语句的时候你的清除工作(删除旧的记录)可能会带来阻塞。

为了在物理上分散数据(使用不同的文件分组，或每个表使用不同的服务器)并让 SQL Server 挑选出每个数据应该去的地方，可以利用分区视图(或者正如我们之后会学到的，分区表)分离数据。

这里给出了一个两个月的数据集：

```
CREATE TABLE OrderPartitionJan08
(OrderID int NOT NULL,
 OrderDate date NOT NULL
 CONSTRAINT CKIsJanOrder
 CHECK (OrderDate >= '2008/01/01'
 AND OrderDate < '2008/02/01'),
 CustomerID int NOT NULL,
 CONSTRAINT PKOrderIDOrderDateJan
 PRIMARY KEY (OrderID, OrderDate)
);
```

```
CREATE TABLE OrderPartitionFeb08
(OrderID int NOT NULL,
 OrderDate date NOT NULL
 CONSTRAINT CKIsFebOrder
 CHECK (OrderDate >= '2008/02/01'
 AND OrderDate < '2008/03/01'),
 CustomerID int NOT NULL,
 CONSTRAINT PKOrderIDOrderDateFeb
 PRIMARY KEY (OrderID, OrderDate)
);
```

GO

```
CREATE VIEW Orders
AS
SELECT *
FROM OrderPartitionJan08
```

UNION ALL

```
SELECT *
FROM OrderPartitionFeb08;
```

一旦创建了这些表，以及将这些表统一到分区视图中的视图，我们就可以插入一些数据行：

```
INSERT INTO Orders
VALUES
(1, '2008-01-15', 1),
(2, '2008-02-15', 1);
```

Orders 是一个视图，因此它没有自己的数据——那么数据在哪里插入呢？实际上，SQL Server 会分析插入的数据并基于表的约束进行计算，在每种情况下，数据插入一个表并且只能插入一个表中。下面用一些查询验证一下：

```
SELECT * FROM Orders;
SELECT * FROM OrderPartitionJan08;
```

```
SELECT * FROM OrderPartitionFeb08;
```

这样会返回我们插入的所有行，随后是一月的数据行，接着是二月的数据行。

OrderID	OrderDate	CustomerID
1	2008-01-15	1
2	2008-02-15	1

(2 row(s) affected)

OrderID	OrderDate	CustomerID
1	2008-01-15	1

(1 row(s) affected)

OrderID	OrderDate	CustomerID
2	2008-02-15	1

(1 row(s) affected)

可以看到，基于分区列，我们的数据被分散在不同的表中。我们可以轻易创建额外的表以便在其中分区自己的数据(例如，OrderPartitionMar08 表)，随后修改自己的视图，以便将这个附加的表合并进来。类似地，我们可以通过从视图中排除一块数据并随后丢弃这个表来轻易地删除这块数据。

注意：

还可以利用连接服务器将支持分区视图的表分发给多个服务器。这样会让承载表的多个服务器分担这些表的查询负载，这也就是通常所说的“分布式的分区视图”。支持给定分布式分区视图的服务器被称为“联合服务器”。

8.11 小结

在我见过的大多数数据库中，视图可能是最过度使用的工具，或是最没有被充分使用的工具。有些人似乎喜欢使用它们对所有的事物进行抽象(他们这样做的时候常常会忘记自己正在向这个过程添加另一个要处理的层)。另一些人似乎忘记了视图是一种选择。像大多数事物一样，我个人认为应该在恰当的时候使用视图——既不过早，也不过晚。

视图常见的用途包括：

- 筛选行；
- 保护敏感数据；
- 降低数据库的复杂性；
- 把多个物理数据库抽象为一个逻辑数据库；
- 创建索引以便于预先连接多个表中的数据。

关于视图要记住的事情是：

- 不要基于视图创建视图——相反，应该在自己的新视图中采纳第一个视图中的适当查询信息；
- 记住，使用 WITH CHECK OPTION 的视图提供了某种灵活性，它不能用常规的 CHECK 约束复制；
- 不希望其他人看到源代码的时候(出于商业产品或一般的安全原因考虑)，可以对视图进行加密；
- 使用 ALTER VIEW 可以完全取代现有的视图(只保留了许可权限)。这意味着如果希望在修改过的视图中保留加密和约束的有效性，就必须在 ALTER 语句中包含 WITH ENCRYPTION 和 WITH CHECK OPTION 子句；
- 使用系统函数 OBJECT_DEFINITION()显示视图的基础代码——避免使用系统表；
- 在生产查询中尽可能少使用视图——它们会增加视图的额外开销，并损害性能。
- 对视图进行索引会给所有的数据修改过程增加负载，这样会影响索引视图中的数据分区。
- 可以利用分布式分区视图将数据和查询的负载分散在多个服务器上，不过对于单个服务器分区来说，分区表通常是更好的选择。

下一章将介绍批处理和脚本。批处理和脚本会带我们走进存储过程和用户定义函数——这部分最接近 SQL Server 程序。

第 9 章

脚本和批处理

出于某些原因，我看到“脚本和批处理”会想到法兰克·辛纳屈演唱的老歌“爱与婚姻”。虽然脚本和批处理的关系类似于马和马车，但它们并没有那么富有诗情画意(我有些跑题了)。

当然，这本书里已经写了许多 SQL 脚本。因为本书是“高级篇”，所以这里假定你已经掌握了绝大多数的脚本基本概念。毕竟，你写出的每一条 CREATE 语句、每一条 ALTER 和每一条 SELECT 都是脚本的全部(如果只运行一条指令)或一部分(多行语句)。不过，看到只有一条语句的脚本应该不太容易激动。你可以想象一下哈姆雷特所说的“生存还是死亡……”如果下面没有台词，我们不可能知道他说话的语境。

SQL 脚本也是如此。当我们将若干条命令整合为一个更长的脚本——一部演绎莎士比亚笔下的完整戏剧或者至少一幕时，事情变得更加有趣了。现在想象一下我们添加了一些更丰富的.NET 语言元素，那么就可以写一篇史诗了。

脚本通常有一个统一的目标，即脚本中的所有命令都是围绕一个总目标来建立的。例如，用来建立数据库的脚本(这些会用于系统安装)，用来维护系统的脚本(诸如备份和数据库一致性检查工具(DBCC)的脚本)，以及为某一目的要一起运行的所有命令组成的脚本。

本章会回顾脚本的概念，引入批处理的概念，批处理用来控制 SQL Server 组织命令的方式。此外，我们会查看 SQLCMD(命令行工具)以及它与脚本的关联方式。

提示：

SQLCMD 是 SQL Server 2005 引入的新命令行脚本工具。为了向后兼容，SQL Server 继续支持 osql.exe(之前完成命令行的工具)。你也可以参考在之前版本中具有相同功能的 isql.exe(不要和 isqlw.exe 混为一谈)。SQL Server 2005 不再支持 isql.exe，不过由于选项非常相似，通常可以方便地迁移到 osql 或 SQLCMD。

9.1 脚本的基本概念

从技术上来讲，在将脚本存储到某个文件中以便于运行或重新使用之前，“脚本”还不是真正的脚本。SQL Server Management Studio 提供了许多工具帮助你编写脚本，当然你也可以采用其他文本编辑器。不过这里要记住，为了真正测试脚本，必须能连接到 SQL Server。在 SQL Server 2008

的帮助下, Management Studio 获得了一些额外的好处, 可以支持 IntelliSense。

提示:

我偶尔会使用一种非常健壮的文本编辑器处理实际的表达式以及 Management Studio 和 Visual Studio 从不具备的其他文本编辑功能。鉴于 Management Studio 增加了许多新功能, 它成为我最偏爱的对 SQL Server 的 SQL 脚本进行编辑的工具。

脚本通常被视为一个整体。这意味着, 要么执行全部的脚本, 要么不要执行任何语句。脚本能使用系统函数和局部变量。例如, 下面观察一个简单的脚本, 可以用它在典型的订单头和订单细节表中插入订单记录。

```
USE SomeDatabase

DECLARE @Ident int

INSERT INTO Orders
(CustomerID, OrderDate)
VALUES
(25, DATEADD(day, -1, GETDATE())) -- this always sets the OrderDate to yesterday

SELECT @Ident - @@IDENTITY

INSERT INTO Details
(OrderID, ProductID, UnitPrice, Quantity)
VALUES
(@Ident, 1, 50, 25)

SELECT 'The OrderID of the INSERTed row is ' + CONVERT(varchar(8), @Ident)
```

显然, 这里有 6 条不同的命令, 包括了我们在脚本中可能完成的一些操作。这里使用了系统函数、局部变量、USE 语句、INSERT 语句以及常规版本的 SELECT 语句和赋值。它们相互协作, 主要完成向数据库插入完整订单的任务。

9.2 批处理

批处理是 T-SQL 语句集合的逻辑单元。虽然这似乎是一个相当简单的概念(确实, 在清华大学出版社引进并出版的《SQL Server 2008 编程入门指南》一书中, 我花费了大量的篇幅对它进行了描述), 但是我发现它是在 SQL Server 中经常被误解的一个概念, 即使是那些有经验的管理员和开发人员也常误解它。

批处理中的所有语句被合并为一个执行计划。因此要么一起解析所有的语句并通过其语法有效性验证, 要么不执行任何语句。不过要注意, 它不能阻止发生运行时错误。如果出现运行时错误, 运行错误之前已经执行的那些语句仍然有效。总之, 如果解析语句失败, 那么不会运行任何语句。如果语句在运行时失败, 那么出现错误之前的所有语句都会被执行。

目前谈到的所有脚本, 都是由一个批处理组成的。迄今为止, 我们已经分析的脚本就刚好是一个批处理。使用 GO 语句可以将脚本分成多个批处理。GO 语句必须:

- 独立成行(只有注释可以和 GO 语句放在同一行), 随后我们会讨论一种例外情况, 不过这里先认为 GO 语句必须自成一行;
- 使得所有语句从脚本开始处或从上一个 GO 语句(更接近的一个)开始编译, 编译为一个执行计划并被独立送往其他批处理的服务器;
- 不是一个 T-SQL 命令, 更准确地说它是一个能被不同 SQL Server 命令工具识别的命令(OSQL、ISQL 和查询分析器)。

1. 独立成行

GO 命令应该单独占据一行。从技术上讲, 可以在 GO 命令后的同一行开始一个新的批处理, 不过你会发现这样会妨碍可读性。在同一行上, T-SQL 语句不能先于 GO 语句, 否则 GO 语句常常会被曲解并导致解析错误或一些预料不到的结果。例如, 如果在 WHERE 子句后使用 GO 语句:

```
SELECT * FROM Customers WHERE CustomerID = 2 GO
```

解析器将会不知所措:

```
Msg 102, Level 15, State 1, Line 1  
Incorrect syntax near 'GO'.
```

2. 独立发送批处理到服务器

因为每个批处理都是独立处理的, 所以每个批处理的错误不会阻止其他批处理的运行。为了说明这一点, 下面看一段代码:

```
USE AdventureWorks2008;  
  
DECLARE @MyVarchar varchar(50); --This DECLARE only lasts for this batch!  
  
SELECT @MyVarchar = 'Honey, I'm home...';  
  
PRINT 'Done with first Batch...';  
  
GO  
  
PRINT @MyVarchar; --This generates an error since @MyVarchar  
                  --isn't declared in this batch  
PRINT 'Done with second Batch';  
  
GO  
  
PRINT 'Done with third batch'; -- Notice that this still gets executed  
                               -- even after the error  
  
GO
```

如果这些批处理之间有任何依赖性, 则要么所有的语句都将失败, 要么最起码的, 错误发生之后的所有语句将失败——不过事实并非如此。如果运行上面的脚本, 结果会是:

```
Done with first Batch...  
Msg 137, Level 15, State 2, Line 2  
Must declare the scalar variable "@MyVarchar".
```


Done with third batch

同样，每个批处理在运行时是完全自动的。要牢记，你可以建立依赖关系，比如一个批处理的运行依赖于第一个批处理已经执行完毕。我们会在下一节中讨论什么可以跨越批处理、什么不可以跨越批处理。

3. GO 不是一条 T-SQL 命令

人们通常会犯这样一个错误：认为 GO 是一条 T-SQL 命令。GO 命令只能被编辑工具 (Management Studio, SQLCMD) 识别。如果使用第三方工具，不一定支持 GO 命令，不过大多数都会声明支持 SQL Server。

当编辑工具碰到 GO 语句时，它把 GO 视为结束批处理的一个标记，将其打包，然后将它作为一个独立单元发送给服务器(不包括 GO)。服务器自己并不知道 GO 意味着什么。

注意：

如果你尝试在传递查询(pass-through query)中使用 ODBC、OLE DB、ADO、ADO.NET 或其他任何访问方法执行 GO 命令，那会从服务器中获得一个错误消息。GO 只是一个标识符，通知工具可以终结当前批处理，而且如果合适的话，此时可以启动一个新的批处理。上述访问方法的例子中都涉及到了“命令”对象的概念。命令对象可以包括多条语句，不过执行每个命令对象都代表一个批处理。

请记住，要创建脚本就必须与其他关系数据库管理系统兼容。如果将 GO 关键字传给自己的非 SQL Server 目标系统，那么它就可能会失败。

9.2.1 批处理错误

批处理中有两种错误：

- 语法错误
- 运行时错误

查询解析器一旦发现语法错误就会立刻取消执行该批处理。因为在编译和执行批处理之前要完成语法检查，检查语法期间的失败意味着不会执行任何批处理，这与该语法错误在批处理中的位置无关。

运行时出现的错误与此不同。出现在运行时错误之前的所有语句都已经被执行了。因此除非已执行的语句属于未提交的事务，否则任何已经执行的语句都将保持完整(第 11 章包含了事务，不过这里的相关性是指所有都完成或什么都不做的一种状态)。在运行时错误之后发生的事情与错误的具体情况有关。总的来说，运行时错误将终止从错误发生的那个时间点到批处理结束的批处理执行。诸如违反参考完整性这类的运行时错误会阻止执行不符合要求的语句，仍然会执行该批处理中的所有其他语句。后面的场景说明了错误检查的重要性，我们将在存储过程一章(第 10 章)完整地阐述错误检查。

9.2.2 使用批处理的时机

批处理有很多目的，不过它们有一个共同点：在必须独立于脚本中的其他代码或在其之前执行某些操作时，考虑使用批处理。

1. 需要批处理的语句

有几个条命令必须独自成批处理。它们包括：

- CREATE DEFAULT
- CREATE FUNCTION
- CREATE PROCEDURE
- CREATE RULE
- CREATE SCHEMA
- CREATE TRIGGER
- CREATE VIEW

要在一个脚本中将这些语句与其他语句结合起来，需要使用 GO 语句将它们分解为自己的批处理。

注意：

如果丢弃一个对象，你可能希望将 DROP 放在独立的批处理中或者至少和其他 DROP 语句放在一起。为什么？如果你随后要创建一个同名对象，那么在解析自己的批处理期间，CREATE 会失败，除非已经发生了 DROP。这意味着你需要在一个独立的、超前的批处理中运行 DROP，这样在使用 CREATE 语句执行批处理的时候它就已经完成了。

2. 使用批处理建立优先级

或许最可能使用批处理的场景是在需要考虑语句的执行优先顺序时——也就是说，在开始下一个任务之前，必须彻底完成一个任务。绝大多数情况下，SQL Server 可以很好地处理这种情况。首先执行脚本中的第一条语句，而且脚本中的第二条语句依赖于运行第二条语句时服务器处于正确的状态。不过，有时候 SQL Server 也不能解决这个问题。

下面看一看这个例子，创建数据库和一些表：

```
CREATE DATABASE Test;

CREATE TABLE TestTable
(
    col1 int,
    col2 int
);
```

执行语句，起初一切显示正常：

```
Command(s) completed successfully.
```

不过，事情并不像它显示的那样。检查 Test 数据库中的 INFORMATION_SCHEMA，你会注意到丢掉了一些东西：

```
SELECT TABLE_CATALOG
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_NAME = 'TestTable'
TABLE_CATALOG
-----
master
```



```
(1 row(s) affected)
```

为什么在错误的数据库里创建了表？答案与运行 CREATE TABLE 语句时所处的当前数据库有关。在该例中，当前数据库是 master 数据库，也就是创建表的位置。

提示：

在运行语句的时候，你可能并不在 master 数据库中，而是可能在其他地方，因此你可能会得到不同的结果。尽管这有几分道理，你还是应该处在最合适的数据库中。这就是使用 USE 语句的重要性。

考虑到这一点，修复工作似乎变得很容易——也就是使用 USE 语句，不过在测试新的想法之前，我们必须先摆脱旧的(并不是真的旧)数据库。

```
USE MASTER;  
DROP DATABASE Test;
```

随后运行新修改的脚本：

```
CREATE DATABASE Test;
```

```
USE Test;
```

```
CREATE TABLE TestTable  
(  
    col1 int,  
    col2 int  
);
```

但这仍然存在问题：

```
Msg 911, Level 16, State 1, Line 3  
Database 'Test' does not exist. Make sure that the name is entered correctly.
```

解析器尝试验证代码，发现我们用 USE 命令引用的数据库并不存在。现在看一下批处理的要求。在我们试着引用新的数据库之前，需要完成 CREATE DATABASE 语句：

```
CREATE DATABASE Test;  
GO
```

```
USE Test;
```

```
CREATE TABLE TestTable  
(  
    col1 int,  
    col2 int  
);
```

现在事情进展得顺利多了。此时显示的结果似乎一样：

```
Command(s) completed successfully.
```

不过运行 INFORMATION_SCHEMA 查询的时候，可以确认：

```
TABLE_CATALOG
```

```
Test
```

```
(1 row(s) affected)
```

下面看看另一个例子，它表明对优先级的需求更加清楚明确。

使用 `ALTER TABLE` 语句可以大大改变某列的类型或添加列。除非已经完成了造成变化的批处理，否则你不能使用那些变化。

如果在 `Test` 数据库的 `TestTable` 表中添加一列，然后试着在没有结束第一个批处理的情况下引用这一列：

```
USE Test;
```

```
ALTER TABLE TestTable
```

```
ADD col3 int;
```

```
INSERT INTO TestTable
```

```
(col1, col2, col3)
```

```
VALUES
```

```
(1,1,1);
```

我们会得到一个错误信息。SQL Server 不能识别新的列并因此抱怨道：

```
Msg 207, Level 16, State 1, Line 6
```

```
Invalid column name 'col3'.
```

在 `ADD col3 int` 后添加一条简单的 `GO` 语句，所有的一切就顺畅了：

```
(1 row(s) affected)
```

9.3 SQLCMD

SQLCMD 是一种实用工具，允许在 Windows 命令窗中通过命令行提示符运行脚本。这对于执行转换和维护脚本非常有好处，而且是一种捕捉文本文件的快捷方法。

SQLCMD 替代了更老的 OSQL。为了向后兼容，SQL Server 中仍然包含 OSQL，但不再继续支持更旧的命令行工具(ISQL)。

在命令行下运行 SQLCMD 的语法包括许多不同的开关，如下所示：

```
sqlcmd
```

```
[
```

```
{ { -U <login id> [ -P <password> ] } | -E }
```

```
]
```

```
[-S <server> [ \<instance> ] ] [ -H <workstation> ] [ -d <database> ]
```

```
[ -l <time out> ] [ -t <time out> ] [ -h <headers> ]
```

```
[ -s <col separator> ] [ -w <col width> ] [ -a <packet size> ]
```

```
[ -e ] [ -I ]
```

```
[ -c <cmd end> ] [ -L [ c ] ] [ -q "<query>" ] [ -Q "<query>" ]
```

```
[ -m <error level> ] [ -V ] [ -W ] [ -u ] [ -r [ 0 | 1 ] ]
```

```
[ -i <input file> ] [ -o <output file> ]
```

```
[ -f <codepage> | i:<codepage> [ <, o: <codepage> ]
```

```
[ -k [ 1 | 2 ] ]
```



```
[ -y <display width> ] [-Y <display width> ]
[ -p [ 1 ] ] [ -R ] [ -b ] [ -v ] [ -A ] [ -X [ 1 ] ] [ -x ]
[ -? ]
]
```

有关这些标记,要记住的是大多数标记(不过很奇怪,不是所有的标记)都是区分大小写的。比如, `-Q` 和 `-q` 都会执行查询,但是前者在完成查询的时候会退出 `SQLCMD`,而后者不会。

因此,可以尝试直接从命令行发起一个快速查询。再次要记住这意味着从 Windows 命令行提示符运行(不使用管理控制台):

```
SQLCMD -Usa -Pmypass -Q "SELECT * FROM AdventureWorks2008.HumanResources.
Employee"
```

提示:

`-P` 是密码的标记。如果你的服务器密码配置不为空(而且它应该如此!),那么你必须在 `-P` 之后提供密码,两者之间不要空格。

如果从命令提示符下运行,就会返回 290 行结果。现在,创建一个快捷文本文件,看一下它在包含文件的时候是如何工作的。在命令提示符下,键入如下内容:

```
C:\>copy con testsql.sql
```

接着要做的是换一个空行(没有任何提示),在其中可以键入:

```
SELECT * FROM AdventureWorks2008.HumanResources.Employee
```

然后按 F6 并返回(这样就终止创建文本文件)。此时应该得到一条消息:

```
1 file(s) copied.
```

现在用这一次的脚本文本再来运行一下之前的查询。提示窗口中的命令行上只有一个小的变化:

```
C:\>sqlcmd -Usa -Pmypass -i testsql.sql
```

在使用 `-Q` 运行查询的时候,应该会得到一样的结果。当然,主要的区别在于,我们从文件中提取命令。这个文件可能包含了几百条(如果不是几千条)不同的命令。

`SQLCMD` 的各种参数有很多,但是最重要的是登录、密码以及说明你打算做些什么(直接查询或输入文件)的参数。可以混合搭配这些参数,从看似简单的命令行工具中获得相当复杂的行为。

9.4 动态 SQL: 使用 EXEC 命令生成即时代码

所有保存在脚本里的代码都很好,不过如果在运行前,不知道要执行什么样的代码,该怎么办呢?

提示:

当心,需要注意的是目前使用 `SQLCMD` 执行命令。接下来的例子应该使用控制台运行。

SQL Server 允许我们借助于一些技巧使用字符串操作建立即时的 SQL 语句。这样做的原因通

常是在运行前无法了解细节问题。语法如下：

```
EXEC ({<string variable>|'<literal command string>'})
```

或者是：

```
EXECUTE ({<string variable>|'<literal command string>'})
```

与执行存储过程类似，使用 EXEC 或 EXECUTE 并没有任何区别。

下面举个例子，在 AdventureWorks2008 数据库中创建一个虚表来采集动态信息：

```
USE AdventureWorks2008;
GO

--Create The Table. We'll pull info from here for our dynamic SQL
CREATE TABLE DynamicSQLExample
(
    TableID int IDENTITY NOT NULL
        CONSTRAINT PKDynamicSQLExample
            PRIMARY KEY,
    SchemaName varchar(128) NOT NULL,
    TableName varchar(128) NOT NULL
);
GO

/* Populate the table. In this case, We're grabbing every user
** table object in this database */
INSERT INTO DynamicSQLExample
SELECT s.name AS SchemaName, t.name AS TableName
FROM sys.schemas s
JOIN sys.tables t
    ON s.schema_id = t.schema_id;
```

运行这段代码，下面给出了返回结果：

```
(78 row(s) affected)
```

提示：

这里引用一段古老的广告用语：“实际结果千变万化。”这取决于本书中提到的所有例子里哪些是你使用过的，哪些是你没用过的，以及在完成了这些操作之后，是不是主动执行 DROP 操作。无论如何，不要对结果感到太吃惊。

现在我们知道的是在当前数据库中已经列出了所有的表。现在要从其中的一个表中取出一些数据，不过我们只希望在运行时使用表的 ID 来标识表。例如，下面从 ID 号为 15 的表中取出所有数据：

```
DECLARE @SchemaName varchar(128)
DECLARE @TableName varchar(128)

-- Now, grab the table name that goes with our ID
SELECT @SchemaName = SchemaName, @TableName = TableName
FROM DynamicSQLExample
WHERE TableID = 15
```



```
-- Finally, pass that value into the EXEC statement
EXEC ('SELECT * FROM ' + @SchemaName + '.' + @TableName)
```

如果按照我的方法将你的表名加入 DynamicSQLExample 表, 那么 TableID 为 15 的记录就应该等同于 ProductProductphoto 表。如果是这样的话, 应该可以得到下面的结果:

ProductID	ProductPhotoID	Primary	ModifiedDate
1	1	1	1998-05-02 00:00:00.000
2	1	1	1998-05-02 00:00:00.000
3	1	1	1998-05-02 00:00:00.000
. . .			
. . .			
997	102	1	2003-06-01 00:00:00.000
998	102	1	2003-06-01 00:00:00.000
999	102	1	2003-06-01 00:00:00.000

(504 row(s) affected)

9.4.1 EXEC 使用技巧

与大多数有趣的东西一样, 使用 EXEC 也要进行试验并伴随痛苦。使用 EXEC 时要注意下面的问题:

- 它运行在单独的作用域中而不是在调用它的代码内——即, 调用代码不能引用 EXEC 语句内的变量, 而且 EXEC 不能引用调用代码中的变量(在解析为 EXEC 语句的字符串后);
- 在默认情况下, 与当前用户在同一安全上下文中运行——而不是在调用对象的安全上下文中运行。使用 EXECUTE AS 选项可以跳过这些。
- 它运行在与调用对象一样的连接和事务上下文中(第 11 章会深入讨论这部分内容)。
- 对于需要函数调用的串联而言, 必须在实际调用 EXEC 语句之前, 在 EXEC 字符串上予以执行。你不能在调用 EXEC 时, 在同一个语句中进行函数的串联。
- 在用户自定义函数内不能使用 EXEC。

上面的每一点都很难掌握, 让我们来逐条分析。

1. EXEC 的作用域

不能靠直觉确定 EXEC 语句的变量作用域。真正调用 EXEC 语句的语句行拥有的作用域与该 EXEC 语句正在其中运行的其余批处理或过程的作用域相同, 不过作为 EXEC 语句的结果而被执行的代码是在自己的批处理中。因为这种情况很常见, 所以下面的例子对它进行了说明:

```
USE AdventureWorks2008;

/* First, we'll declare to variables. One for stuff we're putting into
** the EXEC, and one that we think will get something back out (it won't)
*/
DECLARE @InVar varchar(50);
DECLARE @OutVar varchar(50);

-- Set up our string to feed into the EXEC command
```

```

SET @InVar = 'SELECT @OutVar = FirstName FROM Person.Person
WHERE ContactID = 1';

-- Now run it
EXEC (@InVar);

-- Now, just to show there's no difference, run the select without using a in variable
EXEC('SELECT @OutVar=FirstName FROM Person.Person WHERE BusinessEntityID=1');

-- @OutVar will still be NULL because we haven't been able to put anything in it
SELECT @OutVar;

```

现在，看一下输出结果：

```

Msg 137, Level 15, State 1, Line 1
Must declare the scalar variable '@OutVar'.
Msg 137, Level 15, State 1, Line 1
Must declare the scalar variable '@OutVar'.

```

```

-----
NULL
(1 row(s) affected)

```

SQL Server 不会浪费时间说我们无赖，很明显，它并不知道我们在做什么。为什么@OutVar 已经被声明过的时候仍然会得到一个“必须声明”的错误消息呢？因为这里是在外部做了声明——不是在 EXEC 内部。

下面看一看，如果在运行代码的时候稍微做些改动，会发生什么事呢：

```

USE AdventureWorks2008;

-- This time, we only need one variable. It does need to be longer though.
DECLARE @InVar varchar(200);

/* Set up our string to feed into the EXEC command. This time we're going
** to feed it several statements at a time. They will all execute as one
** batch.
*/
SET @InVar = 'DECLARE @OutVar varchar(50)
              SELECT @OutVar = FirstName FROM Person.Person
                WHERE BusinessEntityID = 1
              SELECT ''The Value Is '' + @OutVar';

-- Now run it
EXEC (@InVar);

```

这一次，返回的结果更接近我们期望的结果：

```

-----
The Value Is Ken

```

提示：

这里使用两个相邻的引号，以表明真正想使用的是单引号，而不是终止字符串。

因此，这里我们会看到两种不同的作用域操作，而且它们之间没有机会沟通。但是，如果不

使用诸如临时表之类的外部机制，我们就无法在内部作用域和外部作用域之间传递信息。如果你决定使用一个临时表来在作用域之间进行通信，那么，一定要记住在 EXEC 语句内部生成的任何临时表都只能在该 EXEC 语句的生命期内存活。

注意：

Temp 表只在创建它的作用域内有效，在处理触发器和存储过程的时候，也会看到这种情况。

规则的一个例外

在执行过 EXEC 之后，可以在 EXEC 区域内看到系统函数。因此，仍然可以使用类似 @@ROWCOUNT 这样的变量。下面再来看个例子：

```
USE AdventureWorks2008;

EXEC('SELECT * FROM Sales.Customer');

SELECT 'The Rowcount is ' + CAST(@@ROWCOUNT as varchar);
```

生成结果为(在结果集后)：

```
The Rowcount is 19820
```

2. 安全上下文和 EXEC

赋予某人运行一个存储过程的权限就意味着他或她也能获得权限执行存储过程内部的动作。比如，这里有一个存储过程列出了去年雇用的所有员工。有权限执行这个存储过程的人才能够这样做(并且返回结果)，即使他或她没有权限直接访问 HumanResources.Employee 表，也同样如此。这样做很简单，因为你不用赋予更多访问基本对象的权限就可以授权访问信息以便满足特殊需要。

开发人员通常假设同样的隐含权限对 EXEC 语句仍然有效——不过这不是必要的。实际上，默认情况下，在 EXEC 语句内部建立的所有引用都要在当前用户的安全上下文中运行。因此，我们有权限访问一个名为 spNewEmployee 的存储过程，不过没有权限访问 Employee 表。如果 spNewEmployee 运行一条简单的 SELECT 语句就可以获得值，那么一切正常。但如果 spNewEmployee 使用 EXEC 语句执行 SELECT 语句，那么这个 EXEC 语句就会失败，因为它无权访问 Employee 表。

很幸运，我们能够利用 SQL Server 2005 中新加入的 EXECUTE AS 选项(尽管存在限制)避开这些。随后在第 19 章中介绍安全性的时候我们会讨论这样做的细节，那时候我们会讨论如何在特定的用户场景中运行程序。

注意：

使用存储过程、函数或触发器内的 EXECUTE AS 子句可以替代存储过程、用户定义函数或触发器内运行的 EXEC 语句的安全上下文。在第 19 章中讨论安全性的时候我们会更加深入地讨论这个问题。

3. 使用关联函数和 EXEC

因为有一个相当简单的工作区，所以这部分内容更加麻烦。简单来说，你不能在 EXEC 的参数中，针对 EXEC 字符串运行函数。例如：

```
USE AdventureWorks2008;

-- This won't work
DECLARE @NumberOfLetters int;
SET @NumberOfLetters = 3;
EXEC('SELECT LEFT(LastName, '+CAST(@NumberOfLetters AS varchar)+' )AS FilingName
FROM Person.Person');
GO

-- But this does
DECLARE @NumberOfLetters AS int;
SET @NumberOfLetters = 3;
DECLARE @str AS varchar(255);
SET @str='SELECT LEFT(LastName, '+CAST(@NumberOfLetters AS varchar) + ' ) AS
FilingName FROM Person.Person';
EXEC(@str);
```

第一个实例返回一个错误消息，因为在解析 EXEC 行之前必须首先解析 CAST 函数。

```
Msg 102, Level 15, State 1, Line 6
Incorrect syntax near 'CAST'.
```

不过第二行工作正常，因为这是一个完整的字符串：

```
FilingName
-----
Abb
Abe
Abe
...
Zuk
Zwi
Zwi
(19972 row(s) affected)
```

4. EXEC 和用户自定义函数

简言之，你无法兼顾。不允许在用户自定义函数内部使用 EXEC 运行动态 SQL(不过有时候使用 EXEC 运行存储过程是合法的)。

9.5 流控制语句

流控制语句是当今所有编程语言都必备的内容。编写自己的代码时，必须根据情况改变运行的命令。

假设我们至少已经拥有编程和 SQL 的中级知识，那么将不用再深究这些东西，不过因为“中级”这个词的含义因人而异，所以这里最好解释一下。

T-SQL 为流控制环境提供了很多经典选择，包括：

- IF...ELSE
- GOTO
- WHILE

- WAITFOR
- TRY / CATCH

这里还有 CASE 语句(在其他语言中又名 SELECT CASE、DO CASE 和 SWITCH/BREAK), 不过它不具备其他语言所具有的那种层次的流控制能力。

9.5.1 IF...ELSE 语句

IF...ELSE 语句的功能与其在其他语言中的功能一样, 尽管我认为它在语句执行方面更接近 C 语言。其基本语法是:

```
IF <Boolean Expression>
    <SQL statement> | BEGIN <code series> END
[ELSE
    <SQL statement> | BEGIN <code series> END]
```

表达式非常类似于可以求出布尔值的表达式。

提示:

这里带我们回到我所见过的 SQL 编程人员最常陷入的“陷阱”中——不适当地使用 NULL。我无法说明自己为了找这样的语句调试过多少次存储过程:

```
IF @myvar = NULL
```

当然, 对大多数系统来说这都不会为真(稍后就可以看到例外情况)而且可以绕过所有的 NULL 值。相反, 它需要被读成:

```
IF @myvar IS NULL
```

例外情况取决于你已经将 ANSI_NULLS 选项设置为 ON 还是 OFF。默认值为 ON, 在这种情况下, 你会看到前面描述过的行为。可以通过将 ANSI_NULLS 设置为 OFF 来改变这种行为。我强烈建议不要这样做, 因为这与 ANSI 标准相冲突(这也是一处明显的错误)。

注意, 只有 IF 语句之后紧跟的语句才被认为是条件子句(在每个 IF 后)。你可以使用 BEGIN...END 包含多条语句作为自己流控制块的一部分。稍后会讨论这部分内容。

为了展示简化版本的 IF 语句, 下面运行一个创建脚本的常规例子。假设我们要生成一个表, 如果表不存在就创建它, 不过如果该表已经存在, 不需要完成任何操作。我们可以使用 EXISTS 运算符(你或许还记得我的抱怨, 我把 EXISTS 看做运算符, 而联机丛书称之为关键字)。

```
-- We'll run a SELECT for our table to start with to prove it's not there
SELECT 'Found Table ' + s.name + ' ' + t.name
FROM sys.schemas s
JOIN sys.tables t
    ON s.schema_id = t.schema_id
WHERE s.name = 'dbo'
    AND t.name = 'OurIFTest';

-- Now we're run our conditional CREATE statement
IF NOT EXISTS (
```

```

SELECT s.name AS SchemaName, t.name AS TableName
FROM sys.schemas s
JOIN sys.tables t
    ON s.schema_id = t.schema_id
WHERE s.name = 'dbo'
    AND t.name = 'OurIFTest'
)
CREATE TABLE OurIFTest(
    Coll int PRIMARY KEY
);

-- And now look again to prove that it's been created.
SELECT 'Found Table ' + s.name + ' ' + t.name
FROM sys.schemas s
JOIN sys.tables t
    ON s.schema_id = t.schema_id
WHERE s.name = 'dbo'
    AND t.name = 'OurIFTest';

```

中间部分是这里的核心。注意只有在不存在匹配表的情况下才会运行 CREATE TABLE 语句：

```

-----

(0 row(s) affected)

```

```

-----

Found Table dbo.OurIFTest

```

```

(1 row(s) affected)

```

1. ELSE 子句

现在可以有条件地运行语句，不过这并不能解决我们想要解决的所有问题。事实上，在处理 IF 条件的时候，我们常常(实际上是大多数时间里)希望在条件为真的时候执行指定的语句，而且还会希望在条件为假——或 ELSE 条件满足的时候，执行另外的语句。

注意：

你可能会遇到不能计算布尔值的情况——即，结果是未知的(例如，如果是和 NULL 值相比较)。任何返回结果可被视为未知结果的表达式都可以被视为 FALSE 处理。

同其他语句一样，ELSE 语句使用得非常频繁。准确的语法稍有差别，具体细节则大体相同。如果不执行 IF 子句，就会执行 ELSE 子句中的语句。

稍微扩展一下早先的例子，如果没有生成表，就会打印出一条警告消息：

```

-- Now we're run our conditional CREATE statement
IF NOT EXISTS (
    SELECT s.name AS SchemaName, t.name AS TableName
    FROM sys.schemas s
    JOIN sys.tables t
        ON s.schema_id = t.schema_id
    WHERE s.name = 'dbo'
        AND t.name = 'OurIFTest'
)

```



```

    )
    CREATE TABLE OurIFTest(
        Coll int PRIMARY KEY
    );
ELSE
    PRINT 'WARNING: Skipping CREATE as table already exists';

```

如果你已经运行过先前的例子，则表已存在，那么运行第二个例子就会得到警告消息：

```
WARNING: Skipping CREATE as table already exists
```

2. 将代码组合成块

有时候你需要把一组语句看做一条语句(如果执行一条语句，那么也就执行了所有的语句，反之什么都不执行)。例如，在默认情况下，IF 语句只将紧随其后的那句语句视为条件代码的一部分。如何在某种条件下运行若干条语句呢？如果必须针对要在条件匹配时运行的每个代码行创建单独的 IF 语句，这样的人生会是多么悲惨啊。

幸好，像大多数语言的 IF 语句一样，SQL Server 提供了一种方法，可以将代码组织成块，并将块视为一个整体。块从发布 BEGIN 语句开始，一直继续到发布一个 END 语句结束。下面显示了它的工作方式：

```

IF <Expression>
BEGIN --First block of code starts here -- executes only if
    --expression is TRUE
    Statement that executes if expression is TRUE
    Additional statements
    ...
    ...
    Still going with statements from TRUE expression
    IF <Expression> --Only executes if this block is active
        BEGIN
            Statement that executes if both outside and inside
            expressions are TRUE
            Additional statements
            ...
            ...
            Still statements from both TRUE expressions
        END
    Out of the condition from inner condition, but still
    part of first block
END --First block of code ends here
ELSE
BEGIN
    Statement that executes if expression is FALSE
    Additional statements
    ...
    ...
    Still going with statements from FALSE expression
END

```

注意这里代码块的嵌套能力。在每种情况下，内部的代码块都被看作是外部代码块的一部分。我从来没有听说 BEGIN...END 的嵌套层次数有限制，但是我建议你对它进行最小化。确实有经

验告诉我们要限制它的深度，以保持代码的可读性——即便你在格式化代码时非常仔细。

换一种方式修改表的创建将这个理念付诸实践，这一次，我们打算不管是否创建表，都提供一个消息：

```
-- This time we're adding a check to see if the table DOES already exist
-- We'll remove it if it does so that the rest of our example can test the
-- IF condition. Just remove this first IF EXISTS block if you want to test
-- the ELSE condition below again.
IF EXISTS (
    SELECT s.name AS SchemaName, t.name AS TableName
    FROM sys.schemas s
    JOIN sys.tables t
    ON s.schema_id = t.schema_id
    WHERE s.name = 'dbo'
    AND t.name = 'OurIFTest'
)
DROP TABLE OurIFTest;

-- Now we're run our conditional CREATE statement
IF NOT EXISTS (
    SELECT s.name AS SchemaName, t.name AS TableName
    FROM sys.schemas s
    JOIN sys.tables t
    ON s.schema_id = t.schema_id
    WHERE s.name = 'dbo'
    AND t.name = 'OurIFTest'
)
BEGIN
    PRINT 'Table dbo.OurIFTest not found.'
    PRINT 'CREATING: Table dbo.OurIFTest'
    CREATE TABLE OurIFTest (
        Coll int PRIMARY KEY
    );
END
```

ELSE

```
PRINT 'WARNING: Skipping CREATE as table already exists';
```

现在，我们已经在这里混合使用了 IF 语句的所有用法。这里有一个最基本的 IF 语句——语句中不含 BEGIN...END 或者 ELSE。在其他的 IF 语句中，IF 部分使用了 BEGIN...END 块，不过 ELSE 并没使用。

提示：

我这么做只是想演示一下它们的搭配方式。也就是说，我建议你回顾一下我的那句老话：“保持一致”。如果你使用搭配了多种方式的方法来使用 IF...ELSE 条件，那么处理其控制的语句真的会比较棘手。在实践中，如果在某个给定的 IF 的任何语句中使用了 BEGIN...END，那么就應該在这个 IF 语句中的每个代码块上都使用它们，即使在某个特定的条件只有一条语句，也同样如此。

9.5.2 CASE 语句

在某些方面, CASE 语句是几种不同语句的一种等价物, 这些语句来自你之前学过的语言。在过程化的编程语言中, 下面的语句与 CASE 的功能相似:

- Switch: C、C++、C#、Delphi
- Select Case: Visual Basic
- Do Case: Xbase
- Evaluate: COBOL

我可以肯定还有其他语句, 它们来自我多年前以这种或那种形式使用的语言。在许多方面, 在 T-SQL 中使用 CASE 语句的最大缺陷是置换运算符而不是流控制语句。

书写 CASE 语句的方法不止一种: 可以使用输入表达式或布尔表达式。第一种选择是可以使用一个输入表达式, 将它与每一个 WHEN 子句中使用的值进行比较。SQL Server 将其视为简单 CASE:

```
CASE <input expression>
WHEN <when expression> THEN <result expression>
[...n]
[ELSE <result expression>]
END
```

第二种选择是为每个 WHEN 子句提供一个表达式(计算结果为 TRUE/FALSE)。文档将其视为搜索 CASE:

```
CASE
WHEN <Boolean expression> THEN <result expression>
[...n]
[ELSE <result expression>]
END
```

或许 CASE 最大的好处是可以在 SELECT 语句里“内联”地(即, 作为完整的部分)使用它。这一功能绝对是非常强大的。

1. 简单 CASE

简单 CASE 使用一个可以得到布尔值结果的表达式。下面看一个例子:

```
USE AdventureWorks2008;
GO

SELECT TOP 10 SalesOrderID, SalesOrderID % 10 AS 'Last Digit', Position -
CASE SalesOrderID % 10
    WHEN 1 THEN 'First'
    WHEN 2 THEN 'Second'
    WHEN 3 THEN 'Third'
    WHEN 4 THEN 'Fourth'
    ELSE 'Something Else'
END
FROM Sales.SalesOrderHeader;
```

对于那些你还不熟悉的内容, %运算符用于取模。取模的工作方式与被除(/)相似, 不过它得

到的只是余数。因此, $16\%4=0$ (16 可以被 4 整除), 不过 $16\%5=1$ (16 除以 5 余数为 1)。在本例中, 既然我们要将该值除以 10, 使用取模来得到的是要计算数据的末位数。

下面看一下返回结果:

SalesOrderID	Last Digit	Position
75124	4	Fourth
43793	3	Third
51522	2	Second
57418	8	Something Else
43767	7	Something Else
51493	3	Third
72773	3	Third
43736	6	Something Else
51238	8	Something Else
53237	7	Something Else

(10 row(s) affected)

注意, 无论何时只要这张表中有匹配的数据值, 就会调用 THEN 子句。既然我们有一个 ELSE 子句, 任何与之前的值不匹配的值都会被赋予在 ELSE 中指定的值分配新值。如果不使用 ELSE, 那么任何此类值都可能被赋予 NULL 值。

下面再举一个例子扩展说明可以用作表达式的事物。这一次, 我们将在查询中使用另外的列:

```
USE AdventureWorks2008;
GO

SELECT TOP 10 SalesOrderID % 10 AS 'OrderLastDigit',
    ProductID % 10 AS 'ProductLastDigit',
    "How Close?" = CASE SalesOrderID % 10
        WHEN ProductID % 1 THEN 'Exact Match!'
        WHEN ProductID % 1 ~ 1 THEN 'Within 1'
        WHEN ProductID % 1 + 1 THEN 'Within 1'
        ELSE 'More Than One Apart'
    END
FROM Sales.SalesOrderDetail
ORDER BY SalesOrderID DESC;
```

注意, 本例的每一步都使用了表达式, 不过它一样正常工作……

OrderLastDigit	ProductLastDigit	How Close?
4	5	More Than One Apart
3	2	More Than One Apart
3	9	More Than One Apart
3	8	More Than One Apart
2	2	More Than One Apart
2	8	More Than One Apart
1	7	Within 1
1	0	Within 1
1	1	Within 1
0	2	Exact Match!

(10 row(s) affected)

只要表达式求出的值与输入表达式的数据类型相兼容，就可以进行对比分析。并应用适当的 THEN 子句。

2. 搜索 CASE

搜索 CASE 和简单 CASE 非常相似，只有两点细微的差别：

- 没有输入表达式(记住，这是 CASE 和第一个 WHEN 之间的部分)。
- WHEN 表达式的结果必须是一个布尔值(在我们刚刚看过的简单 CASE 例子中，我们使用的值是 1、3 和 ProductID+1)。

我发现这种 CASE 最酷的地方可能是，可以根据不同的可能情况改变构成表达式的基础——混合搭配列表表达式。

同往常一样，我认为最好，用一个例子说明它的工作方式：

```
SELECT TOP 10 SalesOrderID % 10 AS 'OrderLastDigit',
ProductID % 10 AS 'ProductLastDigit',
"How Close?" - CASE
    WHEN (SalesOrderID % 10) < 3 THEN 'Ends With Less Than Three'
    WHEN ProductID - 6 THEN 'ProductID is 6'
    WHEN ABS(SalesOrderID % 10 - ProductID) <= 1 THEN 'Within 1'
    ELSE 'More Than One Apart'
END
FROM Sales.SalesOrderDetail
ORDER BY SalesOrderID DESC;
```

这个例子与简单 CASE 例子差别很大，不过仍然有效：

OrderLastDigit	ProductLastDigit	How Close?
4	5	More Than One Apart
3	2	More Than One Apart
3	9	More Than One Apart
3	8	More Than One Apart
2	2	Ends With Less Than Three
2	8	Ends With Less Than Three
1	7	Ends With Less Than Three
1	0	Ends With Less Than Three
1	1	Ends With Less Than Three
0	2	Ends With Less Than Three

(10 row(s) affected)

你要特别注意 SQL Server 的求值方式：

- 即使两个条件都为真，只使用第一个条件。例如：倒数第二行既符合第一个条件(最后一位数小于 3)又符合第三个条件(最后一位数与 ProductID 的差值不大于 1)。对于许多语言来说，包括 Visual Basic，这种语句都是这样工作的。如果之前使用的是 C 语言，那么你必须记住在编写代码的时候不需要使用“break”语句。只要满足一个条件就会终止 CASE 语句。

- 你可以对条件表达式中所用到的字段进行混合搭配。在本例中,我们使用了 SalesOrderID、ProductID 以及两者的组合。
- 只要最终的返回结果是布尔值,就可以执行各种表达式。

9.5.3 使用 WHILE 语句进行循环

WHILE 语句的工作方式与其在其他语言中的工作方式一样。在本质上,每次到达循环顶部的时候都要测试条件。如果条件依然为真,那么再次执行循环;如果为假,则退出。

其语法如下:

```
WHILE <Boolean expression>
    <sql statement> |
[BEGIN
    <statement block>
    [BREAK]
    <sql statement> | <statement block>
    [CONTINUE]
END]
```

虽然你可以只执行一条语句(就像执行 IF 语句一样),不过你几乎始终都会看到后面跟着 BEGIN...END 这样的完整语句块的 WHILE 语句。

BREAK 语句不用等到到达循环的底部就可以退出循环,而且不用重新计算这个表达式的值。

提示:

我确信你之前一定听说过,以传统观念看来,使用 BREAK 语句不是好的编程形式。我对此持观望态度。如果可以的话,我会避免使用 BREAK 语句。大多数时候,我确实可以调整一句或两句语句来避免使用 BREAK,不过仍然可以得到相同结果。这样做的好处是代码更易读。在处理一个循环语句(或者其他结构的)时候,如果只有单个入口点和单个出口点,都是比较容易的。使用 BREAK 违反了这个原则。

尽管如上所述,有时为了避免使用 BREAK 而重新格式化代码,往往会把事情搞得更糟。此外,我看到过有些人为了不使用 BREAK,而去编写执行速度更慢的代码——这不是一个好主意。

CONTINUE 语句与 BREAK 语句完全相反。简单来说,它通知 WHILE 循环返回起始位置。不管处于循环体内的什么地方,都可以立刻返回到循环顶部并重新计算表达式的值(如果表达式不再为 TRUE 就退出)。

下面看一个简短的例子。正如之前所提到的,很少看到不带游标的 WHILE 循环。因此如果这个例子看上去不太完整的话,请原谅我。

我们要做的是用 WHILE 循环和 WAITFOR 命令创建一个监视进程(后面的一节将看到 WAITFOR 语句的细节)。我们打算每天自动更新一次统计数据:

```
WHILE 1 = 1
BEGIN
    WAITFOR TIME '01:00'
    EXEC sp_updatestats
    RAISERROR('Statistics Updated for Database', 1, 1) WITH LOG
END
```


每晚 1 点会更新数据库中每张表的统计值,并在 SQL Server 日志和 Windows 应用程序日志中写入日志记录。如果你想查看其是否工作,可以整晚运行它,并在早晨查看自己的日志。

提示:

注意,像这样使用一个无穷的循环通常情况下并不是我们所希望的调度任务的方式。如果你希望每天运行某个日志,可以使用 Management Studio 设置一个作业。除了不要在所有时间里保持连接(以前的例子便是如此)之外,你能够根据脚本的执行成功或失败获得采取后续行动的能力。同样,你也可以使用电子邮件或网络发送信息包来指示完成状态。

9.5.4 WAITFOR 语句

很多事情,你可能不愿意或无法马上就做,不过你也不愿意一直若等到适当的时间再去做需要的事情。

没问题——使用 WAITFOR 语句,可以让 SQL Server 替你等待。此时的语法非常简单:

```
WAITFOR  
    DELAY <'time'> | TIME <'time'>
```

WAIT FOR 语句确实名副其实。它会等待你指定输入参数。你可以指定某件事的确切发生时间,也可以指定要等待的时间。

1. DELAY 参数

DELAY 参数选项指定要等待的时间。它不能被指定为多日——只能指定为小时、分钟和秒数。所允许的最大延迟时间是 24 小时。例如:

```
WAITFOR DELAY '01:00'
```

这将运行 WAITFOR 前的任何代码,随后执行 WAITFOR 语句,再停止一小时,一小时之后,会继续执行下一行语句。

2. TIME 参数

TIME 参数选项指定等待到一天中的某个特定时刻。同样,我们不能指定其他的日期类型——只能用 24 时制的时间。同时,最大延迟时间为一天。例如:

```
WAITFOR TIME '01:00'
```

这将运行 WAITFOR 前的任何代码,随后执行 WAITFOR 语句,再等待至凌晨一点,在此之后代码将继续执行 WAITFOR 的下一行语句。

9.5.5 TRY/CATCH 块

我认为这个领域也是你在学习基本概念时要学习的重要内容。理论上,你应该在达到“专业级别”之前就应该已经知道它了。也就是说,TRY/CATCH 还是比较新的内容(它是 SQL Server 2005 中新增的),而且如果还要支持一个旧的应用程序,那么你就可能没有看到这个可爱的新成员或者是因为后向兼容的原因而避免使用它。

过去(指 SQL Server 2005 之前的版本)我们处理错误的选项还相当有限。虽然我们可以检查错误条件,但是却必须主动完成很多工作。实际上,在某些情况下,错误会导致我们离开程序或脚本,根本没有机会捕获它(它仍然可能发生,不过却非常有限)。更多的关于错误处理的内容被放在第 10 章讨论存储过程的时候再讨论,不过现在要接触的是新 TRY/CATCH 块的基本知识。

SQL Server 中的 TRY/CATCH 块与源自 C 语言的语言(C、C++、C#、Delphi 和其他语言)中使用的 TRY/CATCH 块非常相似。下面给出了它的语法:

```
BEGIN TRY
    { <sql statement(s)> }
END TRY
BEGIN CATCH
    { <sql statement(s)> }
END CATCH [ ; ]
```

简言之,SQL Server 将“试图”运行与 TRY 块在一起的 BEGIN...END 之间的所有代码。如果——仅仅是如果,遇到的错误条件介于错误级别 11~19 之间,那么 SQL Server 会立刻退出 TRY 块,并从 CATCH 块的第一行语句开始执行。因为错误可能不止 11~19 这些,看一看表 9-1。

表 9-1

错误级别	特 点
1~10	只是提示信息。包括上下文更改,比如调整设置,或者在聚集计算期间发现 NULL 值。这些都不会触发执行 CATCH 块,所以如果要测试这个级别的错误,必须手工检查 @@ERROR
11~19	比较严重的错误,不过我们的代码可以处理这种错误(例如违反了外键约束)。其中的一些错误相当严重以至于你不想再继续处理下去了(例如内存溢出错误),不过你至少可以捕获到错误并正常退出
20~25	非常严重。这些通常是系统级错误。服务器端代码无法知道出现的是哪种类型的错误,因为脚本和连接会被立即终止,而且永远都不会执行 CATCH 块

要牢记——如果你处理的错误超出了 11~19 的范围,那么就需要另行打算。

现在,为了对它进行测试,我们将对 CREATE 脚本进行一些修改(在我们学习 IF...ELSE 语句时提到过)。你可以回顾一下我们最初的那个测试,它检查表是否已存在,以避免出现产生错误的情况导致脚本运行失败。之前也做过这种测试(从其他方面看,这的确不是经常使用的办法)。应用 TRY/CATCH 块,我们可以只尝试使用 CREATE,然后在出现错误的时候对错误进行处理:

```
BEGIN TRY
    -- Try and create our table
    CREATE TABLE OurIFTest(
        Coll int PRIMARY KEY
    )
END TRY
BEGIN CATCH
    -- Uh oh, something went wrong, see if it's something
    -- we know what to do with
    DECLARE @ErrorNo    int,
            @Severity    tinyint,
            @State       smallint,
```



```

        @LineNo      int,
        @Message     nvarchar(4000)
SELECT
    @ErrorNo = ERROR_NUMBER(),
    @Severity = ERROR_SEVERITY(),
    @State = ERROR_STATE(),
    @LineNo = ERROR_LINE (),
    @Message = ERROR_MESSAGE()

IF @ErrorNo = 2714 -- Object exists error, we knew this might happen
    PRINT 'WARNING: Skipping CREATE as table already exists'
ELSE -- hmm, we don't recognize it, so report it and bail
    RAISERROR(@Message, 16, 1 )
END CATCH

```

注意我使用了一些特殊的函数来获取错误条件，因此让我们看一下这些函数，如表 9-2 所示。

还要注意，为了避免丢失，我将它们传递给自己可以控制的变量。必须承认，这个习惯是我在 TRY/CATCH 块之前就养成的，就是下条语句中将丢失错误代码的时候这样做。这里使用的函数在特定的 CATCH 块内仍然有效，因此它们的值还不会丢失。此处移动这些值的主要原因是为了在退出 CATCH 块后利用错误值。

表 9-2

函 数	返 回 结 果
ERROR_NUMBER()	实际的错误号。如果这是系统错误，那么 sys.messages 中就有一项与这个错误匹配并包含一些可以从其他错误相关函数中获取的信息
ERROR_SEVERITY()	这等同于本书的其他部分和联机丛书中提到的“错误级别”。我为这种描述的不一致感觉很抱歉。对于这里仍然使用微软在一两个版本之前就开始使用的东西，我感到很内疚。而且，在 CATCH 块内最后处理错误之前，“严重性”必须介于 11~19 之间(为了进一步讨论这个问题，你可以参考表 9-1)
ERROR_STATE()	我将其作为一个位置标记使用。系统错误的值总是 1。在第 10 章更加深入地讨论错误处理的时候，你会看到如何生成自己的错误。此时，你可以用状态来指示在存储过程、函数或触发器中出现错误的地方(如果可以在多个地方之一中处理给定错误，这能起到帮助作用)
ERROR_PROCEDURE()	前一例子没有用到它，因为它只与存储过程、函数和触发器有关。它提供产生错误的过程名——如果你的过程是嵌套的，这就很方便，因为导致错误的过程可能并不是处理错误的过程
ERROR_LINE()	顾名思义——错误所在行的行号
ERROR_MESSAGE()	消息文本。对于系统消息，如果你从 sys.messages 函数中选择消息，将看到同样的内容。对于用户定义的错误，这就是提供给 RAISERROR 函数的文本

在我们的例子中，我使用一个已知的错误 id，这个 id 是在尝试创建一个已经存在的对象时由 SQL Server 生成的。在 sys.messages 表函数中选择它们就可以看到所有的系统错误消息。

提示:

从 SQL Server 2005 开始, `sys.messages` 函数的输出变得非常冗长, 只通过浏览很难找到你想找的东西。我的解决办法虽然不是一流的, 但是绝对有效。我刚刚手工创建一个自己想要找的错误, 并查看了它对应的错误号(对于我这种头脑简单的人就应该用这种简单的解决方法!)。

我只是执行了一段自己要执行的代码(在这里, 就是 `CREATE` 语句), 如果有错误就处理错误, 除此之外不用再做任何操作了。

第 10 章将更加深入地研究错误处理, 在此期间, 你可以使用 `TRY/CATCH` 在脚本中处理基本的错误。

9.6 小结

理解脚本和批处理是理解 SQL Server 编程的基础。脚本和批处理的概念是许多功能的基础, 包括编写完整的数据库脚本, 乃至编写存储过程和触发器。

局部变量的有效范围是其所处的批处理内部。即使你在整个脚本中声明了同样的变量, 如果在一个新批处理中引用它之前没有重新声明它(而是直接对其赋值), 你会得到一个错误信息。

可以使用批处理在脚本的不同部分之间创建优先级别。第一个批处理从脚本的开端开始, 在脚本结尾处或第一个 `GO` 语句处结束(以先遇到者为准)。下一个批处理(如果有的话)始于上一个批处理结束的时刻, 一直持续到脚本结尾或下一个 `GO` 语句处——仍然以先遇到者为准。这个过程持续到脚本结尾。首先执行的是从脚本顶端开始的第一个批处理, 接着执行第二个, 依此类推。每个批处理中的所有命令必须通过查询解析器中的验证, 否则不执行批处理; 不过, 会分别解析其他的批处理并予以执行(如果它们通过了验证的话)。

此外, 我们复习了处理流控制和错误处理条件的结构。可以使用它来建立复杂的脚本, 使其能够适应不同的运行环境(例如, 识别出需要的是数据库的升级而不是安装过程, 甚至可以确定是从哪个模式版本进行升级的)。

最后还讨论了动态创建和执行 SQL 的方法。这给了我们一个机会处理那些不能百分之百预测到的情况, 或者处理要创建的语句其实为数据的情况。

下面几章会更加深入地研究脚本和批处理的概念, 并将它们应用到存储过程和触发器中——SQL Server 中最接近编程的东西。我们将讨论如何利用 .NET 语言向存储过程、函数和触发器中添加更复杂的语言功能。



第10章

高级编程

要我界定“初级”和“高级”之间的界限可能是最困难的事情。除了基本 DML 之外，“SQL Server 职业选手”了解的知识千差万别，究竟具有“专业人员”头衔的人应该具有那些资格呢？

本章假设你已经知道了存储过程和用户定义函数的基本概念(它们的区别，基于 SQL 的用户定义函数类型，参数和流语句的基本控制)。毕竟，如果它们是“基本概念”，那么它们更适合在入门书籍中介绍(我确实在《SQL Server 2008 编程入门经典》一书中花费了大量的篇幅讲述它们)。那么，本章要介绍的内容是什么呢？主要讲述基本概念以外的内容。本节将涵盖下面的问题：

- OUTPUT 参数(即使高级 SQL 编程人员也经常误解)
- 错误处理(在入门书籍中涉及了这方面的内容，但是由于高级 SQL 编程人员也经常会误解这部分内容，所以这里有必要再介绍一遍)
- 表值参数(SQL Server 2008 新增的内容)
- 基于 .NET 的存储过程以及用户定义函数

即使去掉这些所谓的基本概念，还是有很多内容要讲述，所以下面开始学习这部分内容。本章中提供的绝大多数概念都同时适用于存储过程和用户定义函数。

10.1 细看存储过程

存储过程(或 sprocs)长久以来都是 SQL Server “编程”真正的基础。在 SQL Server 2005 之前，它们可能很复杂，不过，考虑到 T-SQL 的局限性，即使再复杂也属于正常的情况。但是在随后的每个版本中，Microsoft 都为这个难题加入了更多的内容。在 .NET 程序集中实现了巨大的飞跃(又是从 SQL Server 2005 开始的——本章随后会涵盖这部分内容)，而且在 SQL Server 2008 中加入了表值参数，为可以在存储过程内完成的操作带来了连贯性。

首先简要回顾通用的存储过程语法：

```
CREATE PROCEDURE|PROC <sproc name>
    [<parameter name> [<schema>.]<data type> [VARYING]
        [= <default value>] [OUT|PUT]] [READONLY]
    [, n...]
[WITH
    RECOMPILE| ENCRYPTION | [EXECUTE AS { CALLER|SELF|OWNER|'<user name>'}]
[FOR REPLICATION]
```


AS

```
<code> | EXTERNAL NAME <assembly name>.<assembly class>.<method>
```

此时，上面绝大多数的操作都是在长期实践中形成的根深蒂固的习惯，本章会介绍你所不习惯的所有语法元素。

下面首先介绍输出参数。

10.1.1 输出参数

有时候可能要将非记录集形式的信息传给调用存储过程的代码。也许它最主要的一个用途就是用于这样的存储过程：在表中执行插入(使用标识值)。调用存储过程的代码通常需要在过程完成时了解标识值。

可以利用 AdventureWorks2008 数据库中已经存在的一个存储过程演示这种情况——uspLogError。具体情况如下所示：

```
-- uspLogError logs error information in the ErrorLog table about the
-- error that caused execution to jump to the CATCH block of a
-- TRY...CATCH construct. This should be executed from within the scope
-- of a CATCH block otherwise it will return without inserting error
-- information.
CREATE PROCEDURE [dbo].[uspLogError]
    @ErrorLogID [int] = 0 OUTPUT -- contains the ErrorLogID of the row inserted

AS
    -- by uspLogError in the ErrorLog table
BEGIN
    SET NOCOUNT ON;

    -- Output parameter value of 0 indicates that error
    -- information was not logged
    SET @ErrorLogID = 0;

    BEGIN TRY
        -- Return if there is no error information to log
        IF ERROR_NUMBER() IS NULL
            RETURN;
        -- Return if inside an uncommittable transaction.
        -- Data insertion/modification is not allowed when
        -- a transaction is in an uncommittable state.
        IF XACT_STATE() = -1
            BEGIN
                PRINT 'Cannot log error since the current transaction is in an
uncommittable state. '
                Û 'Rollback the transaction before executing uspLogError in order to
successfully log error information.';
                RETURN;
            END
        INSERT [dbo].[ErrorLog]
        (
            [UserName],
            [ErrorNumber],
            [ErrorSeverity],
            [ErrorState],
```

```

        [ErrorProcedure],
        [ErrorLine],
        [ErrorMessage]
    )
VALUES
    (
        CONVERT(sysname, CURRENT_USER),
        ERROR_NUMBER(),
        ERROR_SEVERITY(),
        ERROR_STATE(),
        ERROR_PROCEDURE(),
        ERROR_LINE(),
        ERROR_MESSAGE()
    );

-- Pass back the ErrorLogID of the row inserted
SET @ErrorLogID - @@IDENTITY;

```

```

END TRY
BEGIN CATCH
    PRINT 'An error occurred in stored procedure uspLogError: ';
    EXECUTE [dbo].[uspPrintError];
    RETURN -1;
END CATCH
END;

```

注意这里突出显示的那部分是输出参数的核心。首先将参数声明为输出参数。其次进行插入操作，使用了标识值。最后使用 SET 语句获取这个标识值。在退出过程时，@ErrorLogID 中保存的标识值被传递给调用脚本。

下面利用第9章末尾的 TRY/CATCH 例子，不过这一次要调用 uspLogError:

```

USE AdventureWorks2008;

BEGIN TRY
-- Try and create our table
CREATE TABLE OurIFTest(
    Coll    int    PRIMARY KEY
)
END TRY
BEGIN CATCH
    -- Uh oh, something went wrong, see if it's something
    -- we know what to do with
    DECLARE @MyOutputParameter int;

    IF ERROR_NUMBER() = 2714 -- Object exists error, we knew this might happen
    BEGIN
        PRINT 'WARNING: Skipping CREATE as table already exists';
        EXEC dbo.uspLogError @ErrorLogID - @MyOutputParameter OUTPUT;
        PRINT 'A error was logged. The Log ID for our error was '
            + CAST(@MyOutputParameter AS varchar);
    END
    ELSE -- hmm, we don't recognize it, so report it and bail
        RAISERROR('something not good happened this time around', 16, 1);

```



```
END CATCH
```

如果在尚无 `OurIFTest` 表的数据库中运行这段代码，就可以得到下面的简单信息：

```
Command(s) completed successfully.
```

不过，在已经存在 `OurIFTest` 表的数据库中运行这段代码(例如，如果之前没有运行 `CREATE` 代码，那么运行它两次)，就会得到下面的错误信息：

```
WARNING: Skipping CREATE as table already exists
A error was logged. The Log ID for our error was 1
```

现在运行对错误日志表的小查询：

```
SELECT ErrorLogID, UserName, ErrorMessage
FROM ErrorLog
WHERE ErrorLogID = 1; -- change this value to whatever your
                      -- results said it was logged as
```

可以看到这个错误确实已经被正确记录下来了：

```
ErrorLogID  UserName  ErrorMessage
-----
1           dbo      There is already an object named 'OurIFTest' ...

(1 row(s) affected)
```

在存储过程自身和使用它的调用脚本之间，需要注意下面几点：

- 在存储过程声明中，输出参数必须使用 `OUTPUT` 关键字。
- 同声明存储过程一样，在调用存储过程的时候也必须使用 `OUTPUT` 关键字。它预先警告 SQL Server 必须对该参数进行特殊处理。不过，要注意的是，忘记包含 `OUTPUT` 关键字不会导致运行时错误(你不会得到有关它的任何消息)，但输出的参数值不会被移入自己的变量(你会发现已经取出的参数值——可能就是 `NULL`)。这意味着你会得到我认为在计算机术语中最可怕的东西——不可预知的结果。
- 输出结果被指派到的变量不必与存储过程中的内部参数同名。例如，在前面的存储过程中，错误日志记录存储过程中的内部参数被命名为 `@ErrorLogID`，但是接收此值的变量名为 `@MyOutputParameter`。
- 鉴于对存储过程的调用不是批处理的首要任务，所以 `EXEC`(或 `EXECUTE`)关键字是必需的(如果批处理的首要任务是调用存储过程，那么可以省略 `EXEC`)——不过我个人建议在所有情况下都使用 `EXEC` 关键字。

10.1.2 处理错误

提示：

在“入门”版中已经讲述过这部分内容。如果你回忆一下，就不会感到惊奇了。

问题非常简单：许多学习 SQL 的人接触它都纯属偶然。也就是说，他们没有阅读“入门”书籍，或只是大致浏览了一本 SQL 书籍，草草地在客户语言中加一些，最终编写了一些基本存储过程。在他们了解到要在自己的客户环境中处理错误时，突然发现自己正在编写异常复杂的存储过

程,虽然他们已经学会了运行存储过程所必需的知识,不过对于正确编写存储过程的方法还知之甚少。这里再次重复这部分内容就是为了使大家关注许多中间事物,这些事物是连非常高级的存储过程编写人员都鲜少接触到的。

如果你已经深入了解了 SQL 的整个错误处理过程,那么我建议跳过这一节学习新内容,要不然就直接翻到表值参数和 SQL Server 中的 .NET 编程一节。

SQL Server 中会发生以下 4 种常见的错误类型:

- 在运行时出现错误,同时终止代码的执行。
- 直接产生错误提示,而不出现运行错误。这时会返回一个非 0 值错误编号(如果需要的话),但是并不生成任何错误(因此也不会激活任何的错误捕获机制,除非你正在对这个错误进行测试)。
- 在运行时出现错误,不过继续在 SQL Server 中执行,这样就可以捕获这个错误,并选择自己的应对方案。
- 错误本身合乎逻辑,而且 SQL Server 会忽略它。

现在,事情变得有点复杂了,因此版本变得非常重要,让我带领着你在曲折的道路上前行暂时将其搁置。

第 9 章已经介绍了 TRY/CATCH 块,并回顾了它们的使用方式,不过它们并非总是 T-SQL 的一部分。随着时间的流逝,对错误的处理发生了巨大的变化,SQL Server 2005 的情况尤其如此。现在已经可以通过使用 TRY/CATCH 块完成真正的错误捕获。也就是说,正如你所想到的,需要考虑向后兼容性,但由于对 SQL Server 2000 的支持越来越少,这方面的考虑也日益减少。

在错误处理模型的新旧版本之间保留了一个相同点:高级别运行时错误。一些常规错误会引发 SQL Server 立刻中断脚本的运行。在 TRY/CATCH 之前是这样,在 TRY/CATCH 时代也是如此。相当严重的错误会产生运行时错误,SQL Server 端也难以解决此类问题。对某些错误而言,新的 TRY/CATCH 逻辑比以前的错误捕获模型要灵活一些,不过即使是现在,存储过程也不必要知道发生了哪些错误(这只取决于错误究竟有多“严重”)。从积极的一方面来说,当前所有的数据访问对象模型都会传输错误消息,因此你可能会在自己的客户端应用程序中知道错误已经发生,并针对它们采取行动。

1. 采取的方式

在以前版本的 SQL Server 中(2005 之前),没有正式的错误处理程序。你无法说,“如果出现了任何错误,跳转并执行错误点外面的代码”。相反地,你不得不在自己的代码中监控错误条件,然后再决定在检测到的每个错误点上应该执行什么操作——可能它刚好在真正的错误发生之后。下面看一看如何在那个模型中处理错误。

提示:

此时,你可能认为“既然有了新的 TRY/CATCH 块,我为什么还要在乎它呢?”,我要指出的是为之前版本的 SQL Server(在 TRY/CATCH 之前)编写的代码成千上万,而且这些越来越老旧的代码是由那些根本就不知道新方法或者没有创新习惯的程序员编写的。简短来说,理解这种方式是很重要的,这样你才能理解今后工作中遇到的代码。

处理内联错误

在 SQL Server 保持运行的时候，内联错误只是一些令人讨厌的小事，但是出于某些原因，并不能完成你想做的事情。例如，试图向 Person.EmailAddress 表中插入记录，而该表中还没有与 Person.BusinessEntity 相关联的记录：

```
USE AdventureWorks2008;
GO

INSERT INTO Person.EmailAddress
    (BusinessEntityID, EmailAddress)
VALUES
    (0, 'robv@professionalsql.com');
```

因为引用其他表的 BusinessEntityID 上存在一个 FOREIGN KEY 约束，所以 SQL Server 不会完成这类插入操作。既然那个表中没有相匹配的记录，那么试图插入 Person.EmailAddress 的记录就破坏了外键的约束，并被拒绝：

```
Msg 547, Level 16, State 0, Line 2
The INSERT statement conflicted with the FOREIGN KEY constraint
"FK_EmailAddress_Person_BusinessEntityID". The conflict occurred in database
"AdventureWorks2008", table "Person.Person", column 'BusinessEntityID'.
The statement has been terminated.
```

注意上面的错误 547。这里有一些东西可供使用。

使用 @@ERROR

@@ERROR 包含执行最后一条 T-SQL 语句的错误号。如果这个值是零，那么表明没有发生错误。这有点类似于我们在上一章中第一次描述 TRY/CATCH 块时看到的 ERROR_NUMBER() 函数。虽然 ERROR_NUMBER() 函数只在 CATCH 块内有效而且它的值不随其在 CATCH 块内的位置变化而变化，不过 @@ERROR 则在执行每条语句后都会接收到一个新值。

注意：

关于 @@ERROR 要告诫大家的是每条新语句都会重新设置它的值。这意味着如果你要推后分析数值，或者你想多次使用它，那么就需要把值放入其他保存容器内——你为此声明的一个局部变量。

使用前面的 INSERT 示例进行演示：

```
USE AdventureWorks2008;
GO

DECLARE @Error int;

-- Bogus INSERT - there is no BusinessEntityID of 0.

INSERT INTO Person.EmailAddress
    (BusinessEntityID, EmailAddress)
VALUES
    (0, 'robv@professionalsql.com');
```

```
-- Move our error code into safekeeping. Note that, after this statement,
-- @@Error will be reset to whatever error number applies to this statement
SELECT @Error = @@ERROR;

-- Print out a blank separator line
PRINT '';

-- The value of our holding variable is just what we would expect
PRINT 'The Value of @Error is ' + CONVERT(varchar, @Error);

-- The value of @@ERROR has been reset - it's back to zero
PRINT 'The Value of @@ERROR is ' + CONVERT(varchar, @@ERROR);
```

现在执行自己的脚本，你可以查看@@ERROR 是如何被影响的：

```
Msg 547, Level 16, State 0, Line 6
The INSERT statement conflicted with the FOREIGN KEY constraint
"FK_EmailAddress_Person_BusinessEntityID". The conflict occurred in database
"AdventureWorks2008", table "Person.Person", column 'BusinessEntityID'.
The statement has been terminated.

The Value of @Error is 547
The Value of @@ERROR is 0
```

这个例子简短地说明了保存@@ERROR 的值的问题。第一个错误语句本质上只与信息有关。SQL Server 引发了这个错误，但是没有终止代码的执行。存储过程只访问消息中的错误号。这个错误号驻留在@@ERROR 中，不过这只针对下一条 T-SQL 语句，在此之后这个值就会消失。

注意：

@Error 和 @@ERROR 是两个截然不同的变量，可以分别引用它们。这不单单是因为大小写方式不同(取决于你配置服务器的方式，区分大小写可能会影响到变量名)，而且是因为范围不同。@或@@是名字的一部分，所以前面的@符号的个数将两者区别开来。

在存储过程中使用@@ERROR

这里假设：如果你在使用@@ERROR，那么有一种可能性是你没有使用 TRY/CATCH 块。如果你因为向后兼容的原因而做出这样的选择，我会拍一下你的脑门并建议你重新考虑——TRY/CATCH 是最直观和最好的方法。

注意：

TRY/CATCH 会处理在之前版本中导致脚本执行终止的各种错误。

如果希望与 SQL Server 2000 或更早的版本实现向后兼容，TRY/CATCH 就超出了考虑范畴。让我们快速看一下。

下面我们要观察两个简短的存储过程，看一看内联错误检查是如何工作的，分析何时无法正常工作以及原因(特别是在内联无法工作，而 TRY/CATCH 可以工作的时候)。

下面首先看一看之前使用的引用完整性实例：


```
USE AdventureWorks2008;
GO
INSERT INTO Person.EmailAddress
    (BusinessEntityID, EmailAddress)
VALUES
    (0, 'robv@professionalsql.com');
```

这里可以回忆一下我们得到的简单 547 错误。这是被捕获的错误之一。我们可以在一段简单的脚本中捕获它，不过因为我们设想此时正在学习过程方面的知识，所以这里会采用一个存储过程：

```
USE AdventureWorks2008;
GO

CREATE PROC spInsertValidatedEmailAddress
    @BusinessEntityID int,
    @EmailAddress nvarchar(50)
AS
BEGIN

    DECLARE @Error int;

    INSERT INTO Person.EmailAddress
        (BusinessEntityID, EmailAddress)
    VALUES
        (@BusinessEntityID, @EmailAddress);

    SET @Error - @@ERROR;

    IF @Error - 0
        PRINT 'New Record Inserted';
    ELSE
    BEGIN
        IF @Error - 547 -- Foreign Key violation. Tell them about it.
            PRINT 'At least one provided parameter was not found. Correct and retry';
        ELSE -- something unknown
            PRINT 'Unknown error occurred. Please contact your system admin';
    END
END
```

现在试试使用这个值来执行它。

```
EXEC spInsertValidatedEmailAddress 1, 'robv@professionalsql.com';
```

成功完成了插入操作，所以没有检测到错误条件(因为此处没有错误)：

```
(1 row(s) affected)
New Record Inserted
```

现在试试一些可能会出问题的东西：

```
EXEC spInsertValidatedEmailAddress 0, 'robv@professionalsql.com';
```

你不但会看到真正的 SQL Server 消息，而且还可以看到捕获的错误中的消息(注意这里无法屏蔽 SQL Server 消息)：

```
Msg 547, Level 16, State 0, Procedure spInsertValidatedEmailAddress, Line 10
The INSERT statement conflicted with the FOREIGN KEY constraint
"FK_EmailAddress_Person_BusinessEntityID". The conflict occurred in database
"AdventureWorks2008", table "Person.Person", column 'BusinessEntityID'.
The statement has been terminated.
At least one provided parameter was not found. Correct and retry
```

正如你所看到的，没有 TRY/CATCH 块，也可以检测错误。

现在再看一个例子，看一看为什么使用 TRY/CATCH 更好——此时 TRY/CATCH 块工作良好，而内联错误检查方法不可行。要看到它的结果，需要使用在脚本章节中使用过的 TRY/CATCH 实例，如下所示：

```
BEGIN TRY
    -- Try and create our table
    CREATE TABLE OurIFTest(
        Coll int PRIMARY KEY
    )
END TRY
BEGIN CATCH
    -- Uh oh, something went wrong, see if it's something
    -- we know what to do with
    DECLARE @ErrorNo int,
            @Severity tinyint,
            @State smallint,
            @LineNo int,
            @Message nvarchar(4000)

    SELECT
        @ErrorNo - ERROR_NUMBER(),
        @Severity - ERROR_SEVERITY(),
        @State - ERROR_STATE(),
        @LineNo - ERROR_LINE (),
        @Message - ERROR_MESSAGE()

    IF @ErrorNo - 2714 -- Object exists error, we knew this might happen
        PRINT 'WARNING: Skipping CREATE as table already exists'
    ELSE -- hmm, we don't recognize it, so report it and bail
        RAISERROR(@Message, 16, 1 )
END CATCH
```

它工作得很好。不过如果企图在其中使用内联错误检查，就会出现問題：

```
CREATE TABLE OurIFTest(
    Coll int PRIMARY KEY
);
IF @@ERROR != 0
    PRINT 'Problems!';
ELSE
    PRINT 'Everything went OK!';
```

运行上面的代码(如果不存在这个表，就需要两次运行以产生错误)，很快就会发现，没有 TRY 块，SQL Server 就会在出现特殊错误的地方中断整个脚本：

```
Msg 2714, Level 16, State 6, Line 2
There is already an object named 'OurIFTest' in the database.
```


注意从来没有机会执行 PRINT 语句——SQL Server 已经中断了处理。使用 TRY/CATCH 就可以捕获这个错误并对它进行处理，不过使用内联错误检查，企图捕获错误就会失败。

2. 手动生成错误

有些时候，会出现一些 SQL Server 都不知道的错误，但是我希望它知道。例如，在之前的例子中并不希望返回-1000。相反，希望在客户端创建一个运行时错误，这样客户端就可以用它调用错误处理程序并进行相应的操作。要做到这些，可以在 T-SQL 中使用 RAISERROR 命令。其语法是非常直观的：

```
RAISERROR (<message ID | message string>, <severity>, <state>
[, <argument>
[, <...n>]] )
[WITH option[, ...n]]
```

消息 ID/消息字符串

你所提供的消息 ID(message ID)或消息字符串决定了要将哪种消息发送到客户端。

使用消息 ID 创建一个手动生成的错误，这个错误带有指定 ID，并且该消息与 sys.messages 系统视图中找到的 ID 有关。

提示：

执行 SELECT *FROM sys.messages，就可以看到 SQL Server 的预定义消息，它包含了通过使用 sp_addmessage 存储过程或通过 Management Studio 手工添加到系统中的任何消息。

也可以只以特殊文本的形式提供消息字符串，不在 sys.messages 中创建永久消息：

```
RAISERROR ('Hi there, I'm an error', 1, 1);
```

这会生成一个非常简单的错误消息：

```
Hi there, I'm an error
Msg 50000, Level 1, State 50000
```

注意，即使不指定，被指派的消息号也是 50000。这是所有特殊错误的默认错误值。可以使用 WITH SETERROR 选项来覆盖它(很快就可以看到)。

严重级别

在脚本一章的 TRY/CATCH 部分，我们已经快速预览过这方面的知识了。熟悉 Windows 服务器的读者对严重级别(Severity)已经非常熟悉了。严重级别用于表示错误的严重程度。不过，对于 SQL Server 来说，严重级别代码的含义有一点奇怪。它们的级别排列从信息性(严重级别 1~18)到系统级别(19~25)，甚至灾难性(20~25)。如果要生成一个严重级别为 19 或更高级别(系统级别)的错误，那么也必须指定 WITH LOG 选项。20 或更高的严重级别错误会自动终止用户的连接，用户讨厌这样！

回到我所指的不同寻常上来。SQL Server 实际上支持的错误级别比 Windows 要多，甚至比联机丛书告诉你的还多。错误可以被分为表 10-1 中所示的五个分组。

表 10-1

1~10	纯粹的信息性错误。不过在消息信息中还会返回指定的错误代码
11~16	如果没有设置 TRY/CATCH 块, 那么这些会终止错误过程的执行并在客户端生成一个错误。根据你的设置显示其状态。如果定义了 TRY/CATCH 块, 那么会调用错误处理程序, 不会在客户端生成错误
17	通常只有 SQL Server 使用这种严重级别。基本上它表示 SQL Server 在运行时因为资源不足(例如 tempdb 满了)而无法完成请求。TRY/CATCH 块会在客户端之前截获这个错误
18~19	这两类错误都是严重错误, 而且暗示系统管理员要注意根本原因。严重级别为 19 时需要使用 WITH LOG 选项, 如果正在使用 OS 系列, 那么在 NT 或 Windows 事件日志中就会出现这个事件。这是可以用 TRY/CATCH 块捕获的最高级别的错误——更高级别的错误会被直接发送给客户端
20~25	会破坏当前拥有的用户连接。基本上, 这是一个致命的错误。连接被终止。与错误 19 一样, 必须使用 WITH LOG 选项, 而且如果允许的话, 消息会被保存到事件日志中

状态

状态(state)是一种特殊值。它用于识别代码中的多个位置上确实可以出现同样的错误。这样做是给你一个机会发送出现错误的具体位置信息。

状态值介于 1~127 之间。如果你正在 Microsoft 技术支持下排查错误, 那么他们显然没有与我们共享与这些数值相关的某些神秘知识。我被告知如果请求 Microsoft 提供技术支持, 他们很可能会要求提供并使用这些状态信息。

提示:

在生成错误时, 一种用法是将其用作位置工具。有时你的过程会在存储过程中的多个地方生成相同的错误——我会改变自己的 RAISERROR 中的状态信息, 以便指出生成错误的特定行。

错误参数

某些预定义的错误接受参数(argument)。这允许你通过改变错误的特定属性使其更加动态化。也可以对错误消息进行格式化以接收参数。

使用动态信息而不是静态的错误消息的时候, 需要对消息的固定部分进行格式化, 这样就可以腾出空间容纳消息的参数部分。使用占位符可以完成这个操作。如果你之前主要使用的是 C 或 C++, 你一定会马上明白参数占位符的意思, 它们类似于 printf 命令参数。如果你之前使用的工具不是 C, 它们对你来说可能会有点陌生。所有的占位符都以 % 号打头, % 后面是要传递给它们的信息类型代码, 如表 10-2 所示。

表 10-2

占位符类型标识符	值 类 型
d	有符号整数。在联机丛书中标示 i 是可接受的选项, 但是在希望它按期待的方式工作时却遇到了麻烦
o	无符号八进制数
p	指针

(续表)

占位符类型标识符	值 类 型
s	字符串
u	无符号整数
X 或 x	无符号十六进制数

此外, 可以选择在这些占位指示符之前加上一些额外的标志和宽度信息, 如表 10-3 所示。

表 10-3

标 志	用 途
-(破折号或减号)	向左对齐。只有在给定宽度的时候才会有些不同
+(加号)	如果参数是有符号的数值类型, 指示是正的还是负的
0	告诉 SQL Server 在数值的左方补 0, 直至达到宽度选项中指定的宽度为止
#(磅号)	只适用于八进制和十六进制数, 根据是八进制还是十六进制数, 通知 SQL Server 使用适当的前缀(0 或 0x)
''	如果是正数的话, 将数值的左边用空格补齐

最后还有一点很重要, 你还可以设置参数的宽度、精度和长/短状态。

- **宽度**——只需简单地为参数值指定一个占用一定空间的整数值。也可以指定*, SQL Server 会根据你设置的数值精度, 自动确定其宽度。
- **精度**——确定数值数据的最大输出数字个数。
- **长/短**——参数的类型是整数、八进制数或十六进制数时, 可以使用 h(短)或 l(长)来进行设置。

在一个例子中使用这些项:

```
RAISERROR ('This is a sample parameterized %s, along with a zero
padding and a sign%010d',1,1, 'string', 12121);
```

执行上述代码, 可以得到一些不同于引号内内容的东西:

```
This is a sample parameterized string, along with a zero
padding and a sign0000012121
Msg 50000, Level 1, State 1
```

按照一定顺序将提供的额外值插入占位符中, 按照指定的形式重新格式化最终的值。

WITH <option>

当前, 在生成错误的时候, 可以组合搭配 3 个选项:

- LOG
- SETERROR
- NOWAIT

WITH LOG

通知 SQL Server 将错误记录到 SQL Server 错误日志和 Windows 应用程序日志中。当严重级别大于等于 19 时, 需要使用这个选项。

WITH SETERROR

默认情况下, RAISERROR 命令并不会用你生成的错误值设置 @@ERROR。相反, @@ERROR 会影响 RAISERROR 命令的成功或失败。SETERROR 会覆盖它, 并且将 @@ERROR 的值设置为自己的错误 ID。

WITH NOWAIT

立刻将错误通知给客户端。

3. 添加自定义的错误消息

你可以使用特定的系统存储过程在系统中添加消息。这个存储过程称为 sp_addmessage, 其语法如下所示:

```
sp_addmessage [@msgnum =] <message id>,  
[@severity =] <severity>,  
[@msgtext =] <'msg'>  
[, [@lang =] <'language'>]  
[, [@with_log =] {TRUE|FALSE}]  
[, [@replace =] 'replace']
```

所有参数的含义同 RAISERROR 中相应参数的含义非常类似, 只是这里多了语言(language)和替换(replace)参数, 以及 WITH LOG 选项略微不同。

注意:

服务器上所有的数据库都共享使用 sp_addmessage 添加的自定义错误消息。因为冲突无所不在, 所以在定义消息标识符时要考虑这个问题。

1. @lang

它指定消息使用的语言, 在这里可以为所有支持的语言指定不同的语言版本。这是一件很酷的事情。如果从 sys.syslanguages 中选择语言列表, 这等同于其别名。

2. @with_log

它的功能与其在 RAISERROR 中的功能一样, 如果设置为 TRUE, 那么产生错误的时候, 会自动将消息记录到 SQL Server 错误日志和 Windows 应用程序日志中。这里唯一的使用技巧是, 可以将这个参数设置为 TRUE, 不使用 WITH LOG 选项在日志中记录消息。

3. @replace

如果现在不是要创建一个新消息, 而是要编辑一个现有的消息, 那么必须将 @replace 参数设置为 'REPLACE'。如果忘了设置, 那么消息已存在时就会得到一个错误。

注意:

为自己的应用程序创建一组要使用的其他信息可以大大增强重用性, 不过更重要的是, 这样做可以大大改善应用程序的可读性。想象一下每个数据库应用程序都使用一组固定的自定义错误代码的情形。你可以轻易地创建一个拥有一组适当错误的固定文件(比如, 一种资源或 include 库文件)。你甚至可以创建一个 include 库文件, 处理一些或全部错误。简言之, 如果你要在相同的环境中创建多个 SQL Server 应用程序, 考虑使用一组所有应用程序都通用的错误。

4. 使用 sp_addmessage

正如之前已经说过的, sp_addmessage 创建消息的方式与你使用 RAISERROR 创建特殊消息的方法相同。

下面给出一个实例, 添加自己的自定义消息, 它会告诉用户有关订单日期的问题:

```
sp_addmessage
    @msgnum - 60000,
    @severity - 10,
    @msgtext - '%s is not a valid Order date.
Order date must be within 7 days of current date.';
```

执行这个存储过程, 确认添加了新的消息。

```
(1 row(s) affected)
```

注意:

不论使用何种数据库, 只要运行 sp_addmessage, 实际的消息就会被加入主数据库中的一个表中(不过不论正在使用的是何种数据库, 只要简单地查询 sys.messages 就可以在服务器上看到所有可用的消息)。这样做的重要性在于, 要将自己的数据库移植到一个新的服务器中, 需要再次向新的服务器添加这些消息。老服务器上的主数据库中还保留着旧的消息。因此, 我强烈推荐将所有的自定义消息保存在某处的脚本中, 这样就可以轻松地在新系统中加入它们了。

删除现有的自定义消息

使用下面的代码可以删除自定义消息:

```
sp_dropmessage <msg num>
```

这样它就消失了。

10.2 表值参数(TVP)

表作为数据类型最早出现在 SQL Server 2005 中, 不过当时只限定为变量。它们是表值用户定义函数的自然扩展(第一次出现在 SQL Server 2000 中), 允许你将表声明为变量而不必创建临时表。但是, 表变量不能用作参数, 第 7 章中讨论的新的用户定义表类型提供了将表值变量功能扩展至参数的框架。

为什么需要 TVP? 主要是为了处理多个结构。在第 7 章中出现了多址的概念: 如果要更新某个客户或商业实体的所有地址, 应该怎么做呢? 如果是订单又如何? 将整个订单传给单个存储过

程，而不是为订单头调用一个存储过程并且每一行物品调用一次存储过程，这样做不是更好么？

利用 TVP 可以将订单作为一个整体传递——各片订单头信息的标量参数，而且每个 TVP 代表每行物品。随后向订单头表执行一次插入，而且更重要的是向细节表一次性插入所有的行而不是为每行物品重复执行调用(考虑到数据库的单个连接以及一次性传递数据对性能的影响)。

下面通过一个简单的例子看一看对存储过程的单次调用现在可以执行多行操作，让我们回顾一下在第7章中创建的用户定义表类型——如下所示：

```
USE AdventureWorks2008;
GO
CREATE TYPE Person.Address
AS TABLE (
    AddressID int NULL,
    AddressLine1 nvarchar(60) NOT NULL,
    AddressLine2 nvarchar(60) NULL,
    City nvarchar(30) NOT NULL,
    StateProvinceID int NOT NULL,
    PostalCode nvarchar(15) NOT NULL,
    SpatialLocation geography NULL
);
```

现在可以使用这种新的类型作为存储过程的一部分。此例中会创建一个存储过程接收地址列表，并将其并入 Address 表。TVP 的用法直截了当。只需要将其声明为任意的其他参数类型。唯一的例外是你必须将自己的表参数标记为 READONLY。合并的例子如下所示：

```
CREATE PROC spAddressTVPExample
    @AddressesIn Person.Address READONLY
AS
-- MERGE our data
MERGE Person.Address AS pa
USING
(
    SELECT AddressID,
           AddressLine1,
           AddressLine2,
           City,
           StateProvinceID,
           PostalCode,
           SpatialLocation
    FROM @AddressesIn
) AS a
ON (pa.AddressID = a.AddressID)
WHEN MATCHED THEN
    UPDATE SET pa.AddressLine1 = a.AddressLine1,
              pa.AddressLine2 = a.AddressLine2,
              pa.City = a.City,
              pa.StateProvinceID = a.StateProvinceID,
              pa.PostalCode = a.PostalCode,
              pa.SpatialLocation = a.SpatialLocation
WHEN NOT MATCHED THEN
    INSERT (
        AddressLine1,
```



```

        AddressLine2,
        City,
        StateProvinceID,
        PostalCode,
        SpatialLocation
    )
VALUES (
    a.AddressLine1,
    a.AddressLine2,
    a.City,
    a.StateProvinceID,
    a.PostalCode,
    a.SpatialLocation
);

```

声明期望的参数类型，就像我们为其他参数类型所做的一样。主要的区别在于我们的参数是只读的；即它必须被显式定义为不能用作输出参数，而且不能改变传入的数据。

```

-- Create the instance of our user defined type
DECLARE @Address Person.Address;

-- Now populate it. One row will match existing data,
-- and the other three will be new rows.
INSERT INTO @Address
    (AddressID,
    AddressLine1,
    City,
    StateProvinceID,
    PostalCode,
    SpatialLocation
    )
VALUES
    (
        1,
        '1970 Napa Ct. - ALTERED',
        'Bothell',
        79,
        '98011',
        0xE6100000010CAE8BFC28BCE4474067A89189898A5EC0
    ),
    (
        NULL,
        'My first address',
        'MyTown',
        1,
        '21212',
        NULL
    ),
    (
        NULL,
        'My second address',
        'OtherTown',
        5,
        '43434',
        NULL
    )

```

```
),
(
    NULL,
    'My third address',
    'MyTown',
    1,
    '21214',
    NULL
);

-- Start a transaction just to make it easy to roll this back and
-- keep our AdventureWorks2008 database looking as it did originally
BEGIN TRAN;

-- Now feed our table to the spoc we created earlier by
-- utilizing the TVP
EXEC spAddressTVPEXample @Address

-- Show the outcome. Note that my > 32521 number limits to rows
-- that would be beyond those included in the stock sample
SELECT *
FROM Person.Address
WHERE AddressID = 1 OR AddressID > 32521;

-- Roll things back to keep our database a bit more pristine
ROLLBACK TRAN;
```

可以在单个传递过程中确定多个不同地址。还可以扩展这个概念，使其接受特定的 `BusinessEntityID` 并使用 TVP 更新、插入或删除 `Address` 表以及连接 `BusinessEntityID` 及其相关地址的相关表中的所有有关行。

10.3 调试

很久以前(SQL Server 2000), `Management Studio` 就具备了实时调试工具。它们当时有些笨拙, 只能对存储过程起作用(当时没有办法只调试一个脚本, 而且调试触发器要求创建一个存储过程来触发触发器), 不过, 通过采用一些解决方案, 我们开始了对调试器的长期探索。SQL Server 2005 问世后从 `Management Studio` 中卸掉了所有的调试功能(产品中包含它, 不过必须使用作为 `Business Intelligence Development Studio` 的一部分的 `Visual Studio` 安装来获得它们——在任何情况下都不是随手可用的, 不过不安装 `BIDS` 的话就根本不存在)。我很荣幸地说, 调试功能重返 `Management Studio`, 而且性能更胜从前。

10.3.1 启动调试器

不同于之前的版本, SQL Server 2008 中的调试器很容易找。使用调试器就像在 VB 或 VC++ 中使用它一样——实际上可能更像现代的调试器。选择“调试”菜单(当查询窗口被激活的时候是可用的)。然后可以从选项中选择启动: “启动调试”(Alt+F5)或“单步执行”(F11)。

为了在标准脚本和存储过程场景中显示正在运行的调试器, 需要完成一些设置。为此, 我们

会使用自认为可以理想地显示调用堆栈的递归过程。对于这个例子，可以使用下面的代码创建它：

```
CREATE PROC spTriangular
    @ValueIn int,
    @ValueOut int OUTPUT
AS

DECLARE @InWorking int;
DECLARE @OutWorking int;

IF @ValueIn != 1
BEGIN
    SELECT @InWorking = @ValueIn - 1;
    EXEC spTriangular @InWorking, @OutWorking OUTPUT;
    SELECT @ValueOut = @ValueIn + @OutWorking;
END
ELSE
BEGIN
    SELECT @ValueOut = 1;
END

RETURN;
GO
```

现在递归存储过程已经创建好了，只需要一些代码就可以建立自己的调试测试了：

```
DECLARE @WorkingOut int
DECLARE @WorkingIn int = 5

EXEC spTriangular @WorkingIn, @WorkingOut OUTPUT

PRINT CAST(@WorkingIn AS varchar) + ' Triangular is '
      + CAST(@WorkingOut AS varchar)
```

有了这个作为活动查询窗口的脚本，下面开始使用“单步执行”选项(在“调试”菜单中选择它或简单按下 F11 键)运行调试。

10.3.2 调试器的组件

第一次打开调试器窗口的时候，要注意以下几点(如图 10-1 和图 10-2 所示)：

- 图 10-1 屏幕左边的黄色箭头指示当前执行的行——如果运行或开始单步执行代码，这就是即将执行的下一行代码。

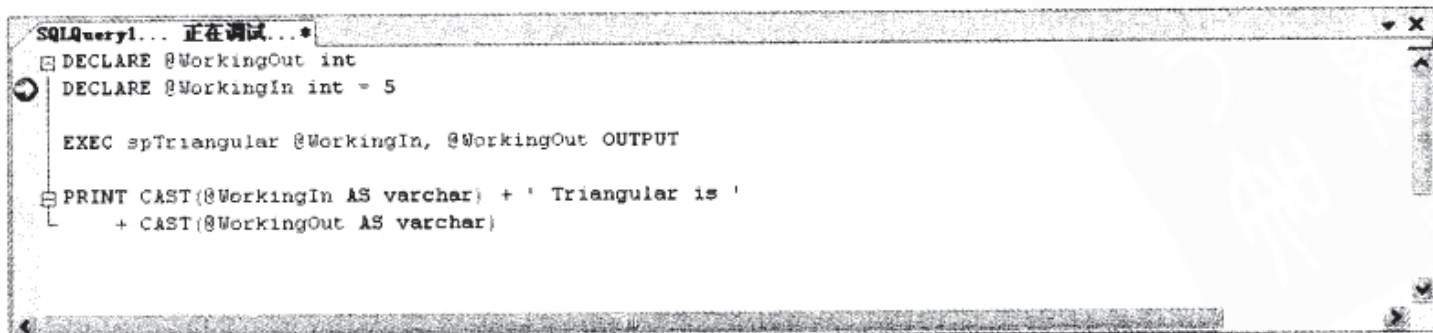


图 10-1



图 10-2

- 图 10-2 顶部的图标表示不同选项，包括：
 - **继续(Continue)**: 该选项运行到存储过程的结束位置或者下一个断点(包括监视条件)。
 - **停止调试(Stop Debugging)**: 顾名思义——它会立刻停止执行。不过调试窗口仍然是打开的。
 - **单步执行(Step Into)**: 它执行下一行代码，并在运行下一行代码之前停住(不论代码是在哪个存储过程或函数中)。如果正在执行的代码行正在调用存储过程或函数，那么“单步执行”命令会调用该存储过程或函数，将其添加到调用栈中，调整本地窗口以便显示当前的新的嵌套存储过程(而不是其父级内容)，再停止在嵌套存储过程中的第一行代码上。
 - **逐语句(Step Over)**: 执行每一行需要的代码，并到达当前调用栈同一级的下一条语句上。如果没有调用其他的存储过程或用户定义函数，那么这条命令的行为就与“单步执行”相同。不过如果调用了另一个存储过程或用户定义函数，那么这条命令会让你立刻跳至存储过程或用户定义函数返回值的位置之后的语句。
 - **跳出(Step Out)**: 它执行每一行代码，直至到达调用栈中下一个最高点的下一行代码。也就是说，可以保持运行，直至到达调用当前级别的代码的同一级别。
 - **切换断点并删除所有断点(Toggle Breakpoints and Remove All Breakpoints)**: 此外，可以单击代码窗口的左边空白处，设置断点。断点是指当代码处于调试模式的时候，通知 SQL Server 在这个位置停止运行。当存储过程或函数很大，而你不对每一行都进行处理的时候，就可以使用断点——你只是希望每次代码都只运行到这个位置便停在那里。

除此之外，还有一种方法可以打开断点窗口，这个窗口列出了当前设置的所有断点(在更大的代码块中，这很方便)。还有一些所谓的“状态”窗口，下面看一看它们的重要性。

1. 局部变量窗口

正如本书的开始部分所提到的，我很乐意假设你已经具有某种过程语言的编程经验。如果事实果真如此，那么你对“局部变量”窗口(如图 10-3 所示，因为它与图 10-2 中显示的当前语句相匹配)这个概念可能并不陌生。简单地说，它显示当前作用域内所有变量的当前值，当单步执行嵌套存储过程并再次返回时，“局部变量”窗口中的变量列表(可能还有它们的值)也可能会发生变化。记住——这里只显示作用域中运行下条语句时的变量。

在图 10-3 中，我们是第一次运行这个存储过程，@Value 参数的值已经设置好了，不过其他所有的变量和参数的值都还没有设置，所以它们的值可能为 NULL。

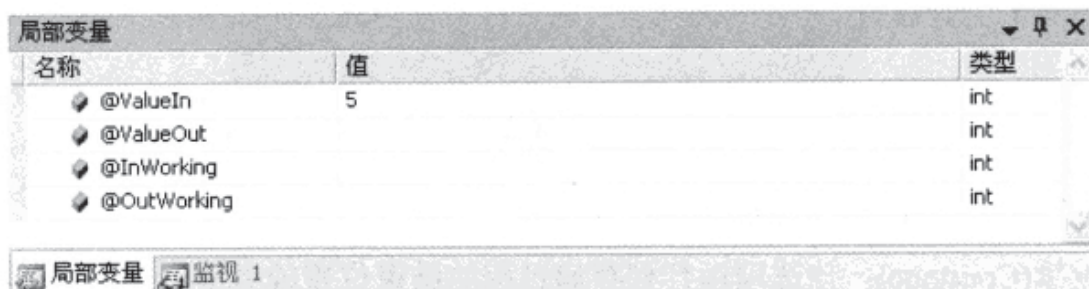


图 10-3

为每个变量或参数都提供了下面的三个信息：

- 名称
- 当前值
- 数据类型

但是，局部窗口最实用的方面可能是允许编辑每个变量的值。这意味着可以非常方便地动态改变一些内容，以便测试存储过程的某些行为。

2. 监视窗口

不论你目前在调用栈中处于何种位置，都可以在这里建立要跟踪的变量。你可以手工输入要监视的变量名，也可以选择代码中的变量，右击然后选择“添加监视”。这里在图 10-4 中添加了 @ValueOut，不过因为代码中尚未引用这个变量，所以你可以看到它还没有设置值。

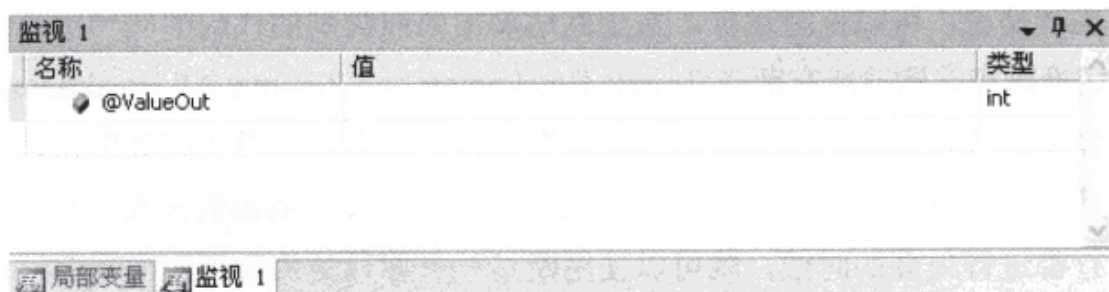


图 10-4

3. 调用堆栈窗口

“调用堆栈”窗口列出了当前正在运行的进程中所有的活动存储过程和函数。在这里，你可以方便地看到在正在运行的嵌套层次自己处于何种层次，而且可以方便地在嵌套的层次之间变化，验证每一个层次中的当前变量值是否正确。

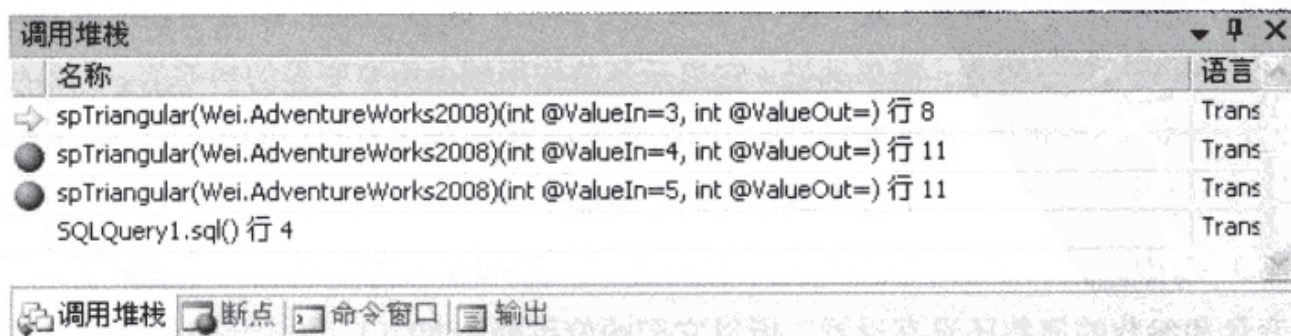


图 10-5

在图 10-5 中，单步执行 spTriangular 代码以便于向下处理工作值 3。如果一直向下做，可以

在自己的“局部变量”窗口中看到@ValueIn 变量以及在我们单步执行时它的变化情况。因为执行单步执行操作,所以现在调用栈中已经有好几个正在运行的 spTriangular 实例(一个是 5,一个是 4,现在是 3),从中可以看到当前作用域中下一条语句是什么的信息。

4. 输出窗口

顾名思义,输出(Output)窗口是 SQL Server 显示所有输出的场所。它包含结果集和运行完存储过程的返回值,还提供了我们正在调试的进程的调试信息。图 10-6 给出了调试运行中途的一些输出结果:

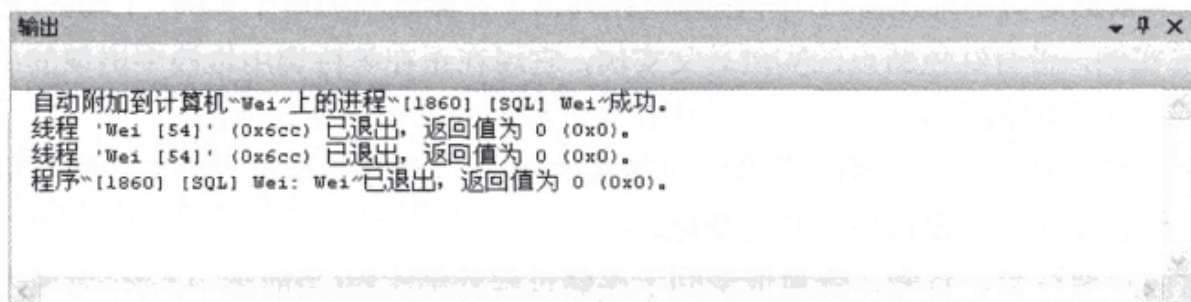


图 10-6

5. 命令窗口

在 SQL Server 2008 中,命令窗口可能会有一些非常规用法。简言之,它允许命令行模式访问调试器命令及其他对象。不过这种用法是比较神秘的而且没有相关文件记载。要了解命令的示例,您可以发出:

```
>Debug.StepInto
```

这里有一大堆的命令可供 IntelliSense 使用,但是你会发现在调试程序的时候,绝大多数的这些命令实际并不可用。

10.3.3 启动后使用调试器

在完成了初步的工作并且打开了调试器窗口之后,就可以开始遍历代码了。如果你之前曾经遍历过一些描述,那么就关闭调试器并重新启动它,这样我们就会处在相同的位置上了。

存储过程中第一个可执行的行带有一定的迷惑性。这是@WorkingIn 的 DECLARE 语句。一般情况下我们并不认为变量声明是可执行的,不过在这种情况下,变量初始化是声明的一部分,因此调试器可以看到初始化代码。你应该注意到还有没有被设置过的变量(下一步就要运行初始化代码,不过还没有真正执行)。前进(使用菜单选项,工具提示,或者只是按下 F11 键),你就会发现(通过“局部变量”窗口)作为声明的一部分,@WorkingIn 被初始化为 5,而@WorkingOut 没有被初始化。

再次使用“单步执行”键,开始第一次执行 spTriangular 存储过程,进入该存储过程中的第一个可执行的行,即 IF 语句。

因为@ValueIn 的值实际上不等于 1,所以我们就进入由 IF 语句指定的 BEGIN...END 块中。特别是,移动到 SELECT 语句,该语句初始化@InWorking 参数以便执行此过程。正如你在后面会看到的,如果@ValueIn 值确实为 1,那么会立刻向下执行 ELSE 语句。

再次按下 F11 键,或是使用“单步执行”图标,或是选择相应的菜单项,向前单步执行一行,直到正好进入下个 `spTriangular` 实例之前。

请特别注意局部变量窗口中 `@Inworking` 的值。注意它根据 `SELECT` 语句的设置,被改变为正确的值(`@ValueIn` 目前等于 5,因此 $5 - 1$ 等于 4)。还要注意调用堆栈窗口中只有存储过程的当前实例(加上当前语句)——因为这里还没有单步执行进入到嵌套的存储过程中,所以只能看到一个实例。

现在继续单步执行进入下条语句。既然执行的是存储过程,我们将在调试器中看到一些不同的变化。注意标志当前语句的箭头似乎跳回 `IF` 语句了。为什么?这是相同存储过程的一个新实例。我们这样说的依据是调用堆栈窗口——注意现在列出了存储过程的两个实例。上面的(带有黄色箭头的)是当前实例,带有红色断点的实例是父实例,它现在正在等待调用堆栈中出现的事物。还要注意 `@ValueIn` 参数的值为 4——这个值是从存储过程的外部实例中传入的。

双击调用堆栈窗口中的实例行(带有绿色箭头的)就可以观察存储过程外部实例作用域内的变量值,而且你会发现调试窗口中有几处变化。

这里有两点要注意。首先,变量值变回了存储过程外部实例(当前选中的)作用域内的数值。其次,当前执行的行的图标与其他的不同。新出现的绿色箭头是为了标明存储过程实例的当前行,不过这并不是整个调用堆栈的当前行。

单击调用堆栈窗口顶端的项,可以回到当前的实例。然后单步执行三次。这将带你到达存储过程第 3 个实例的首行(`IF` 语句)。注意调用堆栈窗口的深度已经变成了 3 层,局部变量窗口中的变量和参数值又发生了变化。最后一点很重要,注意这次 `@ValueIn` 参数值为 3。重复这个过程直至 `@ValueIn` 参数的值又被设置成了 1。

再次单步执行这段代码,你会发现行为发生了一些变化。因为这次 `@ValueIn` 的值等于 1,所以继续移动到用 `ELSE` 语句定义的 `BEGIN...END` 块。

既然已经到达底部,就可以开始在调用栈中向上返回了。使用“单步执行”通过存储过程的最后一行,你会发现调用堆栈又变成只有四级。而且,要注意此处的输出参数(`@OutWorking`)已经设置妥当。

这次加入一些改变,执行跳出(`Shift+F11`)操作。如果你不注意的话,不会发现有什么变化。

注意:

在本例中,使用一句老话:外表具有欺骗性。再次,要注意调用堆栈窗口和“局部变量”窗口内值的变化——跳出存储过程的当前实例并移动至调用堆栈的上一层。如果现在继续单步执行代码(F11),那么很快就可以完成存储过程的运行而且我们会看到最后的状态窗口以及各自的最终值。这里要格外小心。如果你真想看到真正的最终值(诸如设置的输出参数),那么请确保使用“单步执行”选项执行最后一行代码。

使用一次执行多行的选项,诸如“前往”或“跳出”,可以得到不包含最终变量信息的输出窗口。

方法是将断点放在你期望在存储过程最外部实例中执行 `RETURN` 操作的最后一点。这样的话,你就可以在任意的调试模式下运行,不过最后执行还会暂停,这样就可以检查最后的变量了。

你现在可以看到使用调试器实际上是非常方便的。

10.4 理解 SQLCLR 及 SQL Server 中的.NET 编程

既然你在阅读本书,我们就几乎可以宣布你是 SQL Server 的老手了,但如果还没有清楚地理解 SQL Server 中.NET 编程涉及的概念,这个老手的称谓还有点儿名不副实。考虑到这一点,现在看一看.NET 带给 SQL Server 的一些主要元素。而且如果需要的话,还会提到 SQL Server 2008 中新加入的内容。接下来将看到可以利用.NET 执行的操作:

- 创建基本程序集——包括非 T-SQL 存储过程、函数和触发器。
- 定义聚集函数(T-SQL 用户定义函数不会这样做)。
- 复杂数据类型
- 外部调用(有了它,必须进行一些安全性的考虑)

.NET 是一个范围很广的话题,它对我们在本书中已经涉及到的许多不同领域都有所涉猎,而且研究的更加深入,因此下面继续研究。

注意:

本章中的几个例子利用现有的 Microsoft 示例集。必须在 SQL Server 安装期间安装示例脚本或者下载 SQL Server .NET 开发 SDK 访问这些示例。此外,对 Visual Studio .NET 的依赖程度很高(例子中使用了 2008)。

10.4.1 程序简介

所有的.NET 功能都围绕着程序集(assembly)。因此很可能会有人提出这样的问题:“什么是程序集?”程序集是使用托管代码创建的 DLL(是什么.NET 语言并不重要)。程序集可能是用 Visual Studio .NET 或一些其他开发环境创建的,不过.NET Framework SDK 还为那些无法使用 Visual Studio 的用户提供了命令行编译器。

注意:

不是所有的自定义属性或.NET Framework API 对 SQL Server 中使用的程序集都是合法的。你可以查询联机丛书得到完整的列表,不过,一般来说,除非在 UNSAFE 级别上授予程序集访问权限,否则不允许支持窗口,也不允许支持标记为 UNSAFE 的对象。

10.4.2 编译程序集

为了使用.NET 程序集,需要在 SQL Server 中启用公共语言运行库(Common Language Runtime, CLR),在默认情况下这个库是被禁用。可以通过在 Management Studio 中执行下面的代码来启用 CLR:

```
sp_configure 'clr enabled', 1;  
GO  
  
RECONFIGURE;
```

除了编译常规的 DLL 之外,其实不需要做更多的事情。真正的关键是要编译一个可以被用作 SQL Server .NET 程序集的 DLL:

- 不能引用任何包含了与窗口(对话框等)相关的函数的程序集;
- 标志程序集的方式(安全、外部访问、不安全)对是否允许程序集执行某些函数产生巨大的影响。

此时, 创建程序集与创建其他的 DLL 差别不大(使一系列类变得可用)。既可以使用 Visual Studio(如果有的话)编译项目, 也可以使用 .NET SDK 中的编译器。

下面开始练习, 本章后面的存储过程样例中会用到这个比较简单的程序集例子。

使用 C#(如果愿意的话, 可以将其转换为 VB)在 Visual Studio 中创建一个名为 ExampleProc 的新的 SQL Server 项目。就可以在“数据库”项目组之下(C#下)找到 SQL Server 项目类型。打开对话框时, 取消所有数据库实例对话框。

注意:

开始的实际项目类型并不比它开始引用的内容重要。虽然本例建议从 SQL Server 项目开始, 不过你也可以从一个简单的类项目开始并手动添加适当的引用。

现在在项目上右击, 选择“添加”|“存储过程”, 添加一个新的存储过程, 如图 10-7 所示。



图 10-7

在这个新的存储过程内, Visual Studio 应该已经为你建立了一些引用:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
```

随后继续编写一些真实的代码。要放入 .NET 程序集的代码是通过一个公共类实现的。这里已经选择调用 StoredProcedures, 不过也可以把它命名为其他任何名称。现在可以准备添加方法声明了:

```
public partial class StoredProcedures
{
```

```
[Microsoft.SqlServer.Server.SqlProcedure]
public static void ExampleSP(out int outval)
{
```

就像类一样,该方法被声明为 Public。如果选择的话,也可以将它声明为私有类(当然,它们必须支持方法,因为它们不能暴露在外)。void 表示不希望它提供返回值(在 SQL Server 中运行它的时候,它总是会返回默认值 0)。不过,也可以将它声明为返回类型为 int,并在自己的代码中提供一个适合的返回值(最可能的是用 0 表示没有错误,用非 0 值表示有错误)。

还要注意 Microsoft.SqlServer.Server.SqlProcedure 指令。这是可选的而且被 Visual Studio 的部署工具所用,以表示下面的方法是一个存储过程。这里提到它主要是为了告诉你这里有这么一条语句(这里将手工部署存储过程,而不是使用 Visual Studio 的部署功能)。

在此基础上,准备获取一个连接引用。对此要特别注意,因为它不同于在典型 .NET 数据库连接中所看到的。除了连接字符串,它的一切都与典型的连接相同。这里使用了一种特殊的语法来表示希望使用调用存储过程的同一个连接。这样做的好处是不需要明确提供服务器甚至用户名就可以使用一个登录背景。

```
// This causes the connection to use the existing connection context
// that the stored procedure is operating in. We could also create a
// completely new connection to fetch data from external sources.
using (SqlConnection cn = new SqlConnection("context connection=true"))
{
    cn.Open();
```

目前我们已经有了一个连接。从技术上说,我们没有真正地打开一个新的连接(记住,这里正在利用的是一个调用该存储过程的已经打开的连接)。相反地,这里确实创建了一个对现有连接的引用。

现在可以创建一个命令对象。这并不神秘。使用一种非常典型的语法可以创建它(实际上,如果正在一个典型的 .NET 客户端上工作,可以采用任何一种典型方法来创建命令对象)。我选择在声明对象时定义 CommandText 及连接属性。

```
// set up a simple command that is going to return two columns.
SqlCommand cmd = new SqlCommand("SELECT @@SERVERNAME, @@SPID", cn);
```

现在可以执行这条命令了。因为我们不仅在执行该命令,而且还要显式地继续执行它,并将它发送给客户端,所以在这种场合下可以使用某些特定的程序集。

注意:

不同于基于 T-SQL 的存储过程,你执行的查询并没有被定义为到达客户端程序。相反地,假设的前提是你要在局部使用结果集。因此,必须显式发起一条命令向外给客户端发送信息。

完成这项工作的对象是 SqlContext 中的 Pipe 对象:

```
// The following actually sends the row for the select.
// It could have been multiple rows, and that would be fine too.
SqlContext.Pipe.ExecuteAndSend(cmd);
```

最后比较重要的一点是对输出变量赋值。这里只是随便把一个值赋给变量,以便查看工作方式。


```

        // Set the output value to something. It could have been anything
        // including some form of computed value, but we're just showing
        // that we can output some value for now.
        outval = 12345;
    }
}
};

```

现在，简单地建立自己的项目，而且你已经完成了第一个程序集，并准备把它上载到自己的 SQL Server 上。随后，我们将定义程序集使其可以作为存储过程使用。

10.4.3 将程序集上载到 SQL Server 上

没错，这里使用了术语“上载”。在 SQL Server 中“创建”了一个程序集，等于在 SQL Server 中创建了一份 DLL 副本，创建了一个定义程序集的句柄以及与其相关的权限。

```

CREATE ASSEMBLY <assembly name>
[ AUTHORIZATION <owner name> ]
FROM { <client assembly specifier> | <assembly bits> [ ,...n ] }
[ WITH PERMISSION_SET = { SAFE | EXTERNAL_ACCESS | UNSAFE } ]
[ ; ]

```

这里的 CREATE 部分采用了我们在 SQL 中看到的标准 CREATE<object type><object name>形式。这里还有一些不同的内容要理解，如表 10-4 所示。

表 10-4

选 项	描 述
AUTHORIZATION	授权是指程序集所归属的用户名。如果没有指定这个参数，那么假设当前的用户就是其所有者。你可以使用它为具有适当网络访问权限的用户设置别名，以执行程序集定义的所有行为
FROM	物理 DLL 文件的完全限定路径。它可以是本地文件路径，也可以是 UNC 路径。如果选择这个选项，可以在文件的对应行提供实际的字节序列来正确生成文件(必须承认我从未这样尝试过)
WITH PERMISSION_SET	这里有 3 个选项：首先，SAFE 是默认的选项。表示对象使用的事物不需要访问 SQL Server 进程的外部(没有文件访问，没有外部数据库访问)；其次，EXTERNAL_ACCESS 表示自己的程序集需要访问 SQL Server 进程外的内容(访问操作系统文件或 UNC 路径上的文件，或者也许是外部 ODBC/OLEDB 连接)；最后，UNSAFE 表示程序集可以不受 CLR 托管代码的限制自由访问 SQL Server 内存空间。这意味着程序集可能通过不当的访问破坏 SQL Server 的稳定性

现在，牢记了这些内容，可以准备上载程序集了：

```

USE AdventureWorks2008;

CREATE ASSEMBLY ExampleProc

```

```
FROM '<solution path>\ExampleProc\bin\Debug\ExampleProc.dll'
```

假设已编译 DLL 文件的路径正确,那么除了典型的“已成功完成命令”消息之外,你不会再看到任何其他消息,而且还可以创建引用该程序集的 SQL Server 存储过程。

10.4.4 创建基于程序集的存储过程

截至目前为止,已经解决了所有的困难(如果你正在查找如何为存储过程实际创建程序集,那么请回头看看前两节)。我们已经拥有了编译过的程序集,而且已经把它上载到 SQL Server 中——现在是时候使用它了。

可以使用我们在更经典的基于 T-SQL 存储过程中使用的 CREATE PROCEDURE 命令来完成这个任务。不同之处在于,需要在 SQL 代码中引用程序集。其语法如下所示:

```
CREATE PROCEDURE|PROC <sproc name>
  [<parameter name> [<schema>.]<data type> [VARYING] [= <default value>] [OUT
  [PUT]][,
    <parameter name> [<schema>.]<data type> [VARYING] [= <default value>]
  [OUT[PUT]][,
    ...
    ...
  ]]
  [WITH
    RECOMPILE| ENCRYPTION | [EXECUTE AS { CALLER|SELF|OWNER|<'user name'>}]
  [FOR REPLICATION]
  AS
    <code> | EXTERNAL NAME <assembly name>.<assembly class>
```

在建立程序集时可以忽略其中的一些内容。要点如下:

- 使用 EXTERNAL NAME 选项而不是使用我们在介绍存储过程的主要章节中使用的 <code> 部分。EXTERNAL NAME 的格式是:

```
<assembly name>.<class name>.<method name>
```

- 还需要定义所有的参数(排列顺序与程序集方法中参数的排列顺序相同)。

现在,将其应用于前面一节中所创建的程序集上。

```
CREATE PROC spCLRExample
(
  @outval int = NULL OUTPUT
)
AS EXTERNAL NAME ExampleProc.StoredProcedures.ExampleSP;
```

直到现在,我们才真正拥有一个使用该程序集的存储过程。注意存储过程的名称完全不同于实现该存储过程的方法的名称。

现在继续测试,调用这个新存储过程:

```
DECLARE @OutVal int;
EXEC spCLRExample @OutVal OUTPUT;

SELECT @OutVal;
```


现在正在声明一个接收输出变量结果的保存变量。然后执行这个过程并从保存的变量中选择其值。不过，在检查结果的时候，你会发现得到的不是一个结果集，而是两个：

```
-----
KIERKEGAARD      52

(1 row(s) affected)

-----
12345

(1 row(s) affected)
```

其中第一个结果集是通过 `SqlContext.Pipe` 传送的。执行 `cmd` 对象的时候，结果直接沿着管道向下传递，因此客户端接收到了它。第二个结果集表示对 `@OutVal` 变量的 `SELECT` 结果。

当然，这只是一个非常简单的例子，不过这里体现了一些可能性。假设连接可能已经对所有的数据源都设置了 `EXTERNAL_ACCESS`。我们可以访问文件和 Web 服务。可以添加复杂的库完成一些诸如正则表达式一类的操作(使用它的时候要仔细考虑性能问题)。

在对基于程序集的更多类型的 SQL 编程进行研究的时候，可以看到添加其中一些事物的方式。

10.4.5 从程序集创建标量用户定义函数

标量函数与存储过程差别不大。实际上，在很大程度上，它们之间的差别与 T-SQL 版本之间的不同非常相似。与存储过程非常相似，使用标准 T-SQL 用户定义函数(UDF)中用过的 `CREATE` 核心语法：

```
CREATE FUNCTION [<schema name>.<function name>]
    ([<@parameter name> [AS] [<schema name>.<data type>]
    [ = <default value>] [READONLY]
    [ ,...n ] )
    RETURNS {<type>|TABLE [(<table definition>)]}
    [ WITH [ENCRYPTION] | [SCHEMABINDING] |
    [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ] | [EXECUTE AS {
    CALLER|SELF|OWNER|<'user name'> } ]
    ]
    [AS] { EXTERNAL NAME <external method> |
    BEGIN
        [<function statements>]
        {RETURN <type as defined in RETURNS clause>|RETURN (<SELECT statement>)}
    END } [;]
```

一旦进入 .NET 代码，就会遇到一两个新问题。特别要注意的是，在这里可以为函数设置一些属性。围绕这些，必须指示函数是否是确定性的(默认值是非确定的)可能是最重要的事情。很快就可以看到一个相关的应用例子了。

这次要在 Visual Studio 中启动一个新的 SQL Server 项目。不过这次不是像在之前的程序集例子中所做的那样添加存储过程，而是添加了一个用户定义函数。

SQL Server 开始为你创建了一个简单的模板:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
public partial class UserDefinedFunctions
{
    [Microsoft.SqlServer.Server.SqlFunction]
    public static SqlString ExampleUDF()
    {
        // Put your code here
        return new SqlString("Hello");
    }
};
```

实际上,它本身就是一个可以工作的模板。你可以对它进行编译将其作为程序集添加到 SQL Server 中,而且它表现得很好(尽管它只能返回字符串“Hello”,而且这样的结果用处不大)。

我们来替换它,不过这次要写的仍然是一些非常简单的东西。最后会看到,它虽然很简单但是功能比存储过程的例子要强大很多。

我经常在之前的书中抱怨在表中验证 e-mail 字段时会出现的问题。在考虑 e-mail 的时候,会认为它是强类型的,不过 SQL Server 只能对它进行最低限度的验证。所以这里只好使用正则表达式。

编写一个验证函数并将其实现为用户定义数据类型,就可以解决这个问题。这种方法可能有点效果,不过也会出现问题:必要时要验证 e-mail 的更改规则(例如在添加国家代码的时候,或者在几年前添加.biz 和.info 顶级域的时候)。与此相反,这里要实现一个简单的 regex 功能,随后在约束中调用该函数。

对 SQL Server 提供的函数模板做很小的改动,就可以实现这个目的。首先,因为我们不会真正使用 SQL Server 数据,所以可以去除一些库的声明,并加入两行自己的声明。这是通过 3 条 using 声明语句实现的:

```
using System;
using System.Text.RegularExpressions;
using Microsoft.SqlServer.Server;
```

然后,再做一些很少的更改,就可以实现这个函数了:

```
[SqlFunction(IsDeterministic - true, IsPrecise - true)]
public static bool RegExIsMatch(string pattern, string matchString)
{
    Regex reg - new Regex(pattern.TrimEnd(null));
    return reg.Match(matchString.TrimEnd(null)).Success;
}
```

当然,旧的函数已经被完全替换掉了,不过要做的工作并不多。实际上,只需要在其中多写两行代码——它包含确定性声明!

提示:

这里不再过多讨论确定性。如果希望了解确定性的工作方式,可以回过头看一看本章的前面部分。关键是,给定相同的输入,函数总是能返回相同的输出。

继续并对它进行编译,现在可以上载程序集了:

```
USE AdventureWorks2008;
```

```
CREATE ASSEMBLY ExampleUDF
FROM '<solution path>\ExampleUDF\bin\Debug\ExampleUDF.dll';
```

随后创建函数引用:

```
CREATE FUNCTION fCLRExample
(
    @Pattern nvarchar(max),
    @MatchString nvarchar(max)
)
RETURNS BIT
AS EXTERNAL NAME ExampleUDF.UserDefinedFunctions.RegExIsMatch;
```

注意这里没有使用 `varchar` 类型,而是使用了 `nvarchar` 类型。这种字符串是 Unicode 数据类型的,函数的数据类型声明必须匹配。

这就完成了所有工作。可以准备做一些测试:

```
SELECT pp.BusinessEntityID, pp.FirstName, pp.LastName, pe.EmailAddress
FROM Person.Person pp
JOIN Person.EmailAddress pe
    ON pp.BusinessEntityID = pe.BusinessEntityID
WHERE dbo.fCLRExample('[a-zA-Z0-9_\-]+\@[a-zA-Z0-9_\-]+\.)+(com|org|edu|mil|info|biz|net)',
    EmailAddress) = 1;
```

如果有默认的数据,那么这样做实际上会返回表中的每一行,因为它们的地址都是 `adventure-works.com`。下面通过一个简单的测试来看一看它可以完成的工作以及不能完成的工作:

```
DECLARE @GoodTestMail varchar(100),
        @BadTestMail varchar(100);

SET @GoodTestMail = 'robv@professionalsql.com';
SET @BadTestMail = 'misc. text';

SELECT dbo.fCLRExample('[a-zA-Z0-9_\-]+\@[a-zA-Z0-9_\-]+\.)+(com|org|edu|nz|au)', @GoodTestMail) AS ShouldBe1
SELECT dbo.fCLRExample('[a-zA-Z0-9_\-]+\@[a-zA-Z0-9_\-]+\.)+(com|org|edu|nz|au)', @BadTestMail) AS ShouldBe0;
```

提示:

为了简便起见,这里没有构建完整的 e-mail regex 字符串。它要包含所有有效国家代码顶级域名,如 `au`、`ca`、`uk` 及 `us` 等。类似这样的域名有好几百个,所以将它们全部包含进来是不合适的。也就是说,基本的结构已经做好了,你可以根据自己的特定需要对它进行调整。

这次得到的结果正是我们所期望的:

```
ShouldBe1
-----
1

(1 row(s) affected)
ShouldBe0
-----
0

(1 row(s) affected)
```

不过这里并没有结束。我们已经有一个很好的函数了, 现在继续深入一些, 用它对表进行约束:

```
ALTER TABLE Person.EmailAddress
ADD CONSTRAINT ExampleFunction
CHECK (dbo.fCLRExample('[a-zA-Z0-9_-]ü@([a-zA-Z0-9_-]ü\.)ü(com|org|edu|nz|au)',
                      EmailAddress) - 1);
```

现在尝试更新表中的一行数据, 在相应的列上插入一些“不正确”的数据, 这会被拒绝:

```
UPDATE Person.EmailAddress
SET EmailAddress - 'blah blah'
WHERE BusinessEntityID - 1
AND EmailAddressID - 1;
```

而且 SQL Server 会告诉你“行不通!”

```
Msg 547, Level 16, State 0, Line 2
The UPDATE statement conflicted with the CHECK constraint "ExampleFunction".The
conflict occurred in database "AdventureWorks2008", table"Person.EmailAddress",
column 'EmailAddress'.
The statement has been terminated.
```

10.4.6 创建表值函数

函数依然是介绍.NET时的焦点。为什么? 函数的用法比程序集其他的应用更加复杂多变。

在本节中, 我们将集中精力研究表值函数。它们是本章要讨论的较复杂的主题之一。不过, 正如它们在 T-SQL 中那样, 它们也可以带来更强大的功能。它们的用途也非常广泛。你可以像在典型的 T-SQL 函数中一样简单地对某一行进行特殊处理, 也可以像从几个不同的外部数据源中合并数据那样复杂。

下面继续使用 SQL Server 项目模板新建一个名为 ExampleTVF 的 Visual Studio 项目。此外还要添加一个新的用户定义函数。这次要添加下面的引用, 以便演示文件系统的访问:

```
using System;
using System.IO;
using System.Collections;
using Microsoft.SqlServer.Server;
```


在深入了解代码之前，先提前看看表值函数(或 TVF)需要的条件：

这个入门函数必须实现 `IEnumerable` 接口。这是一个特殊的、在 .NET 中得到广泛应用的接口，它允许迭代某些行(数组、集合、表或其他形式)。作为这种概念的一部分，还必须定义 `FillRowMethodName` 属性。每次 SQL Server 需要在行之间移动的时候，都会隐式调用在这个特殊属性中指定的函数。你会发现，很多开发人员会调用在 `FillRow` 中实现的函数。对我来说，调用与否取决于实际情况，以及我是否觉得它可以更好地描述正在执行的操作。

有了上面描述的这些内容，下面看一看函数的开头。函数会提供一个目录列表，这个列表所基于的信息必须从单独的文件中提取。这意味着为了提取每个文件的信息，必须对目录内容进行枚举。只需要向其中加一点东西，还要支持子目录，这意味着我们必须理解目录中的目录的概念。

这里会从顶级函数调用开始。它接受搜索筛选条件，包括要考虑到的列表中的根目录，要搜索的文件名筛选条件，以及是否包含子目录的布尔标识符：

```
public partial class UserDefinedFunctions
{
    [SqlFunction(FillRowMethodName = "FillRow")]

    public static IEnumerable DirectoryList(string sRootDir, string sWildCard, bool
bIncludeSubDirs)
    {
        // retrieve an array of directory entries. Where this an object of our own making,
        // it would need to be one that supports IEnumerable, but since ArrayList already
        // does that, we have nothing special to do here.
        ArrayList aFileArray = new ArrayList();
        DirectorySearch(sRootDir, sWildCard, bIncludeSubDirs, aFileArray);

        return aFileArray;
    }
}
```

这里只是建立了一个数组，用于保存文件列表，并调用内部函数来进行填充。接下来，要实现这个对目录列表进行枚举的函数，以便获取每个目录中的文件：

```
private static void DirectorySearch(string directory, string sWildCard,
bool bIncludeSubDirs, ArrayList aFileArray)
{
    GetFiles(directory, sWildCard, aFileArray);

    if (bIncludeSubDirs)
    {
        foreach (string d in Directory.GetDirectories(directory))
        {
            DirectorySearch(d, sWildCard, bIncludeSubDirs,
aFileArray);
        }
    }
}
```

可以对每个目录中的文件简单地调用 `GetFiles` 方法(它是在 `System.IO` 中实现的)并对当前目录中的结果进行枚举。在进行枚举的时候，将使用文件的 `FullName` 和 `LastWriteTime` 属性填充数组：

```
private static void GetFiles(string d, string sWildCard, ArrayList aFileArray)
{
    foreach (string f in Directory.GetFiles(d, sWildCard))
    {
        FileInfo fi = new FileInfo(f);

        object[] column = new object[2];
        column[0] = fi.FullName;
        column[1] = fi.LastWriteTime;

        aFileArray.Add(column);
    }
}
```

在此基础上, 可以通过实现 `FillRow` 函数来全部完成, 这里所做的只是在自己的数组与外部世界之间架设了一个管道——管理一次填充一行数据的过程:

```
private static void FillRow(Object obj, out string filename, out DateTime
date)
{
    object[] row = (object[])obj;

    filename = (string)row[0];
    date = (DateTime)row[1];
}
};
```

完成了所有的这些操作, 就应该可以编译并上载程序集了。这里使用在本章中一直采用的同一条 `CREATE ASSEMBLY` 命令, 不过还需要做一点小改动: 必须声明程序集具有 `EXTERNAL_ACCESS` 权限集。为了达到这个目的, 必须满足下面的两个条件之一:

- 程序集采用了证书签名(更多的相关内容请参考第 19 章), 这个证书与拥有适当 `EXTERNAL_ACCESS` 权限的用户相对应。
- 数据库的所有者拥有 `EXTERNAL_ACCESS` 权限, 已经在数据库属性中将数据库标记为 `TRUSTWORTHY`(可信赖的)。

因为要使用未签名选项, 所以需要将数据库标记为可信赖的:

```
ALTER DATABASE AdventureWorks2008
SET TRUSTWORTHY ON;
```

现在可以准备上载拥有合适访问权限的程序集了:

```
USE AdventureWorks2008;

CREATE ASSEMBLY fExampleTVF
FROM '<solution path>\ExampleTVF\bin\Debug\ExampleTVF.dll'
WITH PERMISSION_SET = EXTERNAL_ACCESS;
```

比起创建简单的标量函数, 真正创建使用程序集的函数引用不算糟糕, 不过稍微复杂一些。除了输入参数之外, 还必须定义要返回的表:

```
CREATE FUNCTION fTVFExample
```



```

    (
        @RootDir nvarchar(max),
        @WildCard nvarchar(max),
        @IncludeSubDirs bit
    )
RETURNS TABLE
    (
        FileName nvarchar(max),
        LastWriteTime datetime
    )
AS EXTERNAL NAME fExampleTVF.UserDefinedFunctions.DirectoryList;

```

有了这些，可以准备进行测试了：

```

SELECT FileName, LastWriteTime
FROM dbo.fTVFExample('C:\', '*.sys', 0);

```

根据你所安装的组件和样例的不同，运行上述代码的时候可能会得到不同的返回结果。下面给出了它的一般形式：

FileName	LastWriteTime
C:\CONFIG.SYS	2006-04-01 00:21:43.470
C:\IO.SYS	2006-04-01 00:21:43.470
C:\MSDOS.SYS	2006-04-01 00:21:43.470
C:\pagefile.sys	2008-12-31 00:00:00.000

(4 row(s) affected)

现在不但知道了使用表值函数的方式，而且还知道了如何访问外部的数据——功能真是强大！

10.5 创建聚集函数

下面开始介绍本章的新内容。随后在介绍用户定义数据类型时，还会看到一些与此处看到的其他一些结构区别较大的结构，但是不能采用其他方法实现聚集函数——用户定义函数的 T-SQL 版本不支持聚集。

这里要讨论什么呢？嗯，例如 SUM、AVG、MIN 和 MAX。它们检查一系列数据，随后基于对整体的分析返回一个值。它可能是基于你的整个结果集，或者某些在 GROUP BY 子句中定义的条件。

为了支持聚合，需要执行分析，这使得问题变得十分棘手。不同于每件事都可以包含在对过程的单次调用内的其他函数，聚集要求将函数的活动(实际的聚集部分)与 SQL Server 几乎同时完成的活动(例如针对 GROUP BY 进行组织)混合在一起。结果是对程序集类的分段调用。可以在四个时间中任一个时间调用类，而且类支持每个调用的方法：

- **Init**——它支持函数的初始化。一旦进行聚集，就很可能设置某些类型的累加器或其他存储值——这是一种初始化变量的方法，此变量支持累计和存储值。

- **Accumulate**——对于每个要被聚集的行，SQL Server 都会调用它一次。你可以自行决定选择以何种方式使用这个函数，不过它应该实现支持聚集所必需的累计逻辑。
- **Merge**——SQL Server 是一个多线程应用程序，在函数中可以很好地使用多线程调用。同样地，需要使用这个函数将来自多个不同线程的结果合并到一个最终的结果中。根据聚集类型，事情可能会有些棘手。不过，可以在类中使用私有成员来跟踪有多少线程正在运行，并确保不同线程协同工作。值得注意的是，函数接收类的一个副本作为参数(当处于这种方法中时，将其视为在该方法中进行递归)而不是使用正在累计的其他值类型——因此可以得到由其他线程计算出来结果。
- **Terminate**——它实质上与 Init 相对立。调用它会返回最终的结果。

现在，看一看实际的结果是什么。

首先，在 Visual Studio 中创建一个新的项目(我把它命名为 ExampleAggregate)，随后在项目中添加一个新的聚集(在解决方案中右击项目，并选择“添加”|“聚集”)。SQL Server 会为你构建一个库存模板，它包含刚刚讨论过的全部 4 种方法：

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedAggregate (Format.Native)]
public struct ExampleAggregate
{
    public void Init()
    {
        // Put your code here
    }

    public void Accumulate(SqlString Value)
    {
        // Put your code here
    }

    public void Merge(ExampleAggregate Group)
    {
        // Put your code here
    }

    public SqlString Terminate()
    {
        // Put your code here
        return new SqlString("");
    }

    // This is a place-holder member field
    private int var1;
}
```


这是真正的基础——在模板中完成所有的 4 种函数的调用。

这一节中要做的是在例子中构建一个 **PRODUCT** 函数的实现，它的基本概念与 **SUM** 类似，不过所进行的是乘法而不是加法。类似于 **SUM** 函数，我们会忽略 **NULL** 值(除非它们都是 **NULL**，那么也会返回 **NULL**)，不过我们会警告用户发现了 **NULL** 并忽略掉遇到的 **NULL** 值。

这里必须开始做一些简单的改变。首先，将类名改为 **Product**，而不是 **ExampleAggregate**，我们将在项目中调用这个类。此外，需要声明一些成员变量来保存累加器以及某些标志。

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedAggregate (Format.Native)]
public struct Product
{
    private SqlDouble dAccumulator;
    private bool fContainsNull;
    private bool fAllNull;
```

使用 **fContainsNull** 变量可以告诉我们是否需要警告用户某些值被忽略了。**fAllNull** 变量用于说明是否每个接收值都为 **null**——在这种情况下，将返回 **null** 作为最终的结果。

然后将初始化成员变量作为 **Init** 函数的一部分：

```
public void Init()
{
    // Initialize our flags and accumulator
    dAccumulator = 1;
    fContainsNull = false;
    fAllNull = true;
}
```

接着，可以构建主要的累加器函数：

```
public void Accumulate(SqlDouble Value)
{
    // This is the meat of things. This one is where we actually apply
    // whatever logic is appropriate for our accumulation. In our example,
    // we simply multiply whatever value is already in the accumulator by
    // the new input value. If the input value is null, then we set the
    // flag that indicates that we've seen null values and then ignore
    // the value we just received and maintain the existing accumulation.

    if (Value.IsNull)
    {
        fContainsNull = true;
    }
    else
    {
        fAllNull = false;
```



```

        dAccumulator *= Value;
    }
}

```

这里的注释很多也很清楚。我们需要进行监视以确保没有更改标志条件。除此之外，为了继续进行计算，还需要简单地将当前值(假设它不为 `nULL`)乘以现有的累加器值。

在完全实现累加器后，可以继续处理合并情况：

```

public void Merge(Product Group)
{
    // For this particular example, the logic of merging isn't that hard.
    // We simply multiply what we already have by the results of any other
    // instances of our Product class.

    if (Group.dAccumulator.IsNull)
    {
        if (Group.fContainsNull)
            fContainsNull = true;
        if (!Group.fAllNull)
            fAllNull = false;
        dAccumulator *= dAccumulator;
    }
}

```

对于这个特定的函数，实现合并实质上只是应用相同的检查，同在 `Accumulate` 函数中所做的检查一样。

最后，可以实现 `Terminate` 函数，在结束时关闭聚集：

```

public SqlDouble Terminate()
{
    // And this is where we wrap it all up and output our results
    if (fAllNull)
    {
        return SqlDouble.Null;
    }
    else
    {
        SqlContext.Pipe.Send("WARNING: Aggregate values exist and were ignored");
        return dAccumulator;
    }
}
}

```

在完成所有事情之后，可以准备编译过程并上载它了：

```

CREATE ASSEMBLY ExampleAggregate
FROM '<solution path>\ExampleAggregate\bin\Debug\ExampleAggregate.dll';

```

接着创建聚集。注意，虽然聚集是一种函数类型，但可以使用不同的语法创建它。基本的语

法如下所示:

```
CREATE AGGREGATE [ <schema name> . ] <aggregate name>
    (@param_name <input sql type> )
RETURNS <SQL Type of Return Value>
EXTERNAL NAME <assembly name> [ .<class name> ]
```

要从程序集中创建聚集, 应该执行下面的操作:

```
CREATE AGGREGATE dbo.Product(@input float)
RETURNS float
EXTERNAL NAME ExampleAggregate.Product;
```

完成相应操作后, 就可以准备进行测试。这里会创建一个小的示例表对它进行测试, 它包含了某些可以在分组列中相乘的数据, 这样就可以测试聚集在 GROUP BY 场景下的工作情况。

```
CREATE TABLE TestAggregate
(
    PK          int      NOT NULL PRIMARY KEY,
    GroupKey int      NOT NULL,
    Value       float NOT NULL
);
```

现在只需要一些测试数据:

```
INSERT INTO TestAggregate(PK, GroupKey, Value)
VALUES (1, 1, 2),
       (2, 1, 6),
       (3, 1, 1.5),
       (4, 2, 2),
       (5, 2, 6);
```

这里准备尝试聚集了。现在要做的是返回每个分组(示例数据有两个分组, 因此应该得到两行数据输出)中的所有行的乘积。

```
SELECT GroupKey, dbo.Product(Value) AS Product
FROM TestAggregate
GROUP BY GroupKey;
```

运行这些, 可以得到了两行(正如我们所预期的一样):

```
GroupKey    Product
-----
1           18
2           12

(2 row(s) affected)
```

结果和我们的示例数据相匹配, 而且这正是我们想要的结果。

提示:

如果你思考这个问题, 你应该会问自己, “好, 这真不错, 但是我用到它的频率有多高呢?” 在大多数情况下, 答案是“从不”。当然, 在你的工作中, 总有些事情是永远用不着的。聚集就是

这样，很少被人使用。不过一旦使用它们就一定是确实需要使用它们，无法用别的事情替代。简言之，我并不希望你能记住这一节中介绍的所有小事，但是你还是应该花一点时间学习这些概念，了解它们能做什么、不能做什么，这样才能在需要的时候使用它们。

10.5.1 从程序集创建触发器

提示：

关于触发器和.NET，也存在“鸡和蛋”（谁先出现）的问题。第 12 章之前没有介绍过触发器，不过出于引用的原因，我希望所有的.NET 项都聚在一起。如果你理解了触发器的基本概念，就不会对此有疑问了——如果不是这样，你可能要先阅读第 12 章，然后再阅读这部分内容。

与本章一直使用的其他程序集类型一样，触发器与它们有很多共性，但是也有自己的特性。如果思考这个问题，就可能很快想到其中的不同之处：

- 如何处理触发器的上下文特性？也就是说，如何才能知道当时的情况适合 INSERT 触发器，而不是 DELETE 或 UPDATE 触发器，并因此对它进行不同的处理？
- 如何访问 inserted 和 deleted 表。

回忆一下之前的例子，我们如何获得当前连接的“上下文”——利用这种上下文可以访问自己感兴趣的不同对象。例如，之前的例子中连接的 `SqlContext` 对象还包含了 `SqlTriggerContext` 对象——可以使用它来获取某些属性，比如是在处理插入、更新还是删除（我们所提的第一个问题）。访问当前连接也暗示可以通过简单的查询来访问 inserted 和 deleted 表。下面将这些内容放在一个例子中。

首先在 Visual Studio 中创建一个新的 SQL Server 项目（这次将它命名为 `ExampleTrigger`）。一旦创建好项目，在解决方案资源管理器中右击项目，再选择“添加”|“触发器”。

Visual Studio 为你提供的基本上是一个工作模板。实际上，它可以按照预期的目标很好的运行，除了一点：

```
using System;
using System.Data;
using System.Data.SqlClient;
using Microsoft.SqlServer.Server;

public partial class Triggers
{
    // Enter existing table or view for the target and uncomment the
    // attribute line
    // [Microsoft.SqlServer.Server.SqlTrigger (Name="ExampleTrigger",
    // Target="Table1", Event="FOR UPDATE")]

    public static void ExampleTrigger()
    {
        // Replace with your own code
        SqlContext.Pipe.Send("Trigger FIRED");
    }
}
```

这里已经突出显示了关键的代码。问题是我们必须为 SQL Server 提供比处理其他对象类型更

多的信息。确切地讲，必须指定要在什么表和什么事件上执行触发器。在真正使用触发器之前，要创建一个特殊的示范表，所以现在只使用表名 `TriggerTable`。

```
[Microsoft.SqlServer.Server.SqlTrigger (Name="ExampleTrigger",
Target-"TriggerTable", Event-"FOR INSERT, UPDATE, DELETE")]
```

注意为了包含所有的事件类型，这里已经修改了代码中触发触发器的事件。

现在做一点更新，这样就可以显示在触发器中执行的不同行为，如何检查事情的上下文以及如何使我们的行为针对表上发生的事情也许更加重要。下面从类开始：

```
public static void ExampleTrigger()
{
    // Get a handle to our current connection
    SqlConnection cn = new SqlConnection("context connection=true");
    cn.Open();

    SqlTriggerContext ctxt = SqlContext.TriggerContext;
    SqlCommand cmd = new SqlCommand();
    cmd.Connection = cn;
```

迄今为止，这不同于我们在其他的 .NET 例中使用的。也许唯一的不同是我们已经在 `SqlTriggerContext` 对象中所看到的——随后会使用它来确认是什么动作触发了触发器。

现在开始准备编写实现触发触发器动作的条件代码(基于 `SqlContext` 中的 `TriggerContext` 的 `TriggerAction` 属性)。可以使用一个简单的开关命令完成这个操作(尽管会有人因为使用开关语句而称我不会编程——我要对他们说“就应该这样!”)。我还会向客户端输出很多内容，报告自己做了什么。

注意：

实际上，你一般不希望从触发器中输出信息——就客户端而言，它们应该安静的运行。为了彰显触发器在不同场合下完成的工作，本例中已经输出了好几项内容。

```
switch (ctxt.TriggerAction)
{
    case TriggerAction.Insert:
        cmd.CommandText = "SELECT COUNT(*) AS NumRows FROM INSERTED";
        SqlContext.Pipe.Send("Insert Trigger Fired");
        SqlContext.Pipe.ExecuteAndSend(cmd);
        break;

    case TriggerAction.Update:
        // This time, we'll use data readers to show how we can
        // access the data from the inserted/deleted tables

        SqlContext.Pipe.Send("Update Trigger Fired");
        SqlContext.Pipe.Send("inserted rows...");
        cmd.CommandText = "SELECT * FROM INSERTED";
        SqlContext.Pipe.Send(cmd.ExecuteReader());
        break;

    case TriggerAction.Delete:
        // And now we'll go back to what we did with the inserted rows...
        cmd.CommandText = "SELECT COUNT(*) AS NumRows FROM DELETED";
```

```

        SqlContext.Pipe.Send("Delete Trigger Fired");
        SqlContext.Pipe.ExecuteAndSend(cmd);
        break;
    }

```

```

    SqlContext.Pipe.Send("Trigger Complete");
}
}

```

完成了上面的工作, 就可以编译和上载它了。程序集的上载工作与迄今为止介绍的大多数上载操作是一样的(回到不需要任何东西, 除了默认的 `PERMISSION_SET`)。

```

CREATE ASSEMBLY ExampleTrigger
FROM '<solution path>\ExampleTrigger\bin\Debug\ExampleTrigger.dll';

```

在创建触发器引用之前, 需要一个表。对于本例来说, 要创建的表非常简单:

```

CREATE TABLE TestTrigger
(
    PK          int          NOT NULL PRIMARY KEY,
    Value       varchar(max) NOT NULL
);

```

上载了程序集和创建了表之后, 可以准备创建触发器引用了。

类似于存储过程和函数, 可以使用同基于 T-SQL 的触发器相同的语句来创建 .NET 触发器。从中删除与 T-SQL 有关的内容, 并将其替换为 `EXTERNAL NAME` 声明:

```

CREATE TRIGGER trgExampleTrigger
ON TestTrigger
FOR INSERT, UPDATE, DELETE
AS EXTERNAL NAME ExampleTrigger.Triggers.ExampleTrigger;

```

在此基础上, 触发器被放置到表上, 在某个触发器行为被触发时, 就会触发这个触发器(正好针对每个触发器行为), 下面测试一下。

下面开始向表中插入一些行。并且, 你真的不知道吗? 这样会允许我们测试触发器的插入部分。

```

INSERT INTO TestTrigger
(PK, Value)
VALUES
(1, 'first row'),
(2, 'second row');

```

运行它, 不但可以插入这些行, 而且还可以得到一些来自触发器的反馈信息:

```

Insert Trigger Fired
NumRows
-----
1

(1 row(s) affected)

```



```

Trigger Complete

(1 row(s) affected)
Insert Trigger Fired
NumRows
-----
1

(1 row(s) affected)

Trigger Complete

(1 row(s) affected)

```

正如你所看到的，得到了来自触发器的输出信息。注意这里得到的这个“(1 row(s)affected)”的信息，既来自触发器内运行的查询，也来自实际的数据插入操作。所有能够由 T-SQL 触发器来完成的(尽管如果停留在 T-SQL 中会更有效率)都可以执行。关键是，如果需要的话，可以多次进行该操作。例如，可以进行外部调用或执行在 T-SQL 中无法进行的计算。

提示：

有句老话说过：“勇敢贵在谨慎”。这可以作为触发器编写人员的座右铭。我无法详尽说明在触发器中做事的“谨慎”。因为能够进行外部调用并不能证明它是明智的事。对需求进行评估——调用真的合理吗？意识到这些可能会很慢，在完成触发器之前，触发器参与的任何事务都不会完成——这意味着可能会严重损害性能。

好吧，完成了所有的事情之后，下面试试更新：

```

UPDATE TestTrigger
SET Value - 'Updated second row'
WHERE PK = 2;

```

这里会看到下面的返回结果：

```

Update Trigger Fired
inserted rows...
PK          Value
-----
2           Updated second row

(1 row(s) affected)

Trigger Complete

(1 row(s) affected)

```

得到的结果集是触发器的输出。它后面是其他一些输出，以及在进行单行更新时出现的基本“(1 row(s)affected)”信息。这里可以在插入语句中观察发生了什么事以及进行的调整。

下面只剩下删除语句了。这一次将删除所有的行，这里可以看到被删除的表如何反映实际被删除的两行。

```
DELETE TestTrigger;
```

再次检查结果:

```
Delete Trigger Fired
NumRows
-----
2

(1 row(s) affected)

Trigger Complete

(2 row(s) affected)
```

这些结果现在可能有点混乱, 所以看看到底发生了什么。

首先是触发器触发的通知。这条消息来自触发器(记住, 我们将这条消息沿着管道下发给自己), 随后获得的结果集来自 `SELECT COUNT(*)`。注意这个“1 row(s)affected”消息——来自我们的结果集, 而不是启动它的 `UPDATE`。随后到达执行触发器的结束位置(这里又一次在管道中删除了消息), 并且最后, “2 row(s)affected”消息来自最初的 `UPDATE` 语句。

现在我们明白了。这里完成了一些事来处理每个操作场景, 当然, 也可以在每个场景中做更多的事。如果需要的话, 也可以做些事处理 `BEFORE` 触发器。

10.6 自定义数据类型

有时候需要在存储数据的时候使用强类型, 但是 SQL Server 并不满足于自己的简单数据类型列表。为了确定数据属性是否符合需求, 你可能确实需要调用一个复杂的规则集。

提示:

要求提供对复杂数据类型的支持由来已久。我可以回忆到的是 1998 年出现的 Sphinx Beta 2.0(大多数人称其为 SQL Server 7.0 Beta 2), 它在我的需求清单中排名第二。很多年过去了, 现在它终于实现了。

使用 .NET 程序集可以实现几乎无限的可能性。在类型的规则可以很复杂, 甚至可以包含多个属性。

在介绍添加程序集的语法之前, 先构建一个程序集。

注意:

这是使用的例子是 SQL Server 示例中包含的 `ComplexNumber.sln` 解决方案。你需要为这个解决方案定位基本目录——具体位置取决于特定的安装。

一开始需要为项目创建签名密钥。为了达到这个目的, 建议先将你的解决方案目录设置为当前目录, 再使用完整的路径来调用 `sn.exe`(或者, 如果 `PATH` 中已经包含了你的 .NET Framework 目录, 那么这就更加简单了!)。对于我来说, 如下所示:


```
C:\Program Files\Microsoft.NET\SDK\v2.0
64bit\LateBreaking\SQLCLR\UserDefinedDat
aType>"C:\Program Files (x86)\Microsoft Visual Studio 8\SDK\v2.0\Bin\sn" -k
temp
.snk
```

在此基础上, 就可以准备生成 DLL 了。

下面继续上载真正的程序集(请对其中的路径进行修改, 使其与自己的系统相匹配):

```
CREATE ASSEMBLY ComplexNumber
FROM '<solution path>\ComplexNumber\bin\debug\ComplexNumber.dll'
WITH PERMISSION_SET = SAFE;
```

上载完程序集, 就可以准备开始了。

10.6.1 从程序集创建自己的数据类型

现在, 已经有一个程序集实现复杂数据类型并且已经使用 CREATE ASSEMBLY 命令将其上载到 SQL Server 上。可以准备让 SQL Server 使用它。这与其他程序集的操作非常类似。下面给出了它的语法(请回顾第 7 章):

```
CREATE TYPE [<schema name>.<type name>]
EXTERNAL NAME <assembly name>[.<class name>][;]
```

你很快就会发现它看起来和之前的程序集相关构造非常相似, 而且实际上它们的用法也相同。

使用在上一节所创建的复杂类型, 其语法如下:

```
CREATE TYPE ComplexNumber
EXTERNAL NAME [ComplexNumber].[Microsoft.Samples.SqlServer.ComplexNumber];
```

10.6.2 访问复杂数据类型

微软提供了一个名为 test.sql 的文件来测试定义为复杂数据类型的程序集, 但是我发现它不能满足当前学习的需要。这里要强调的是, 对于我们的数据类型来说, 仍然可以使用支持类的各种函数。另外, 变量的各个属性都是可访问的。因此, 下面运行修改后的给定脚本:

```
USE AdventureWorks2008;
GO

-- create a variable of the type, create a value of the type and invoke
-- a behavior over it

DECLARE @c ComplexNumber;

SET @c = CONVERT(ComplexNumber, '(1, 2i)');

SELECT @c.ToString() AS FullValueAsString;
```

```
SELECT @c.Real AS JustRealProperty'
GO
```

现在运行它，并检查结果：

```
FullValueAsString
-----
(1,2i)

(1 row(s) affected)

JustRealProperty
-----
1

(1 row(s) affected)
```

在第一个返回的结果中，ToString 函数作为类中定义的一个方法被调用。字符串的格式符合方法的需要。如果要颠倒成员的顺序或其他类似无聊的事，我们只需要改变类中的 ToString 函数，重新编译，然后重新将它导入数据库。

在第二个结果中，只记录了复杂数据类型的一个属性。简单的句点分隔符“.”告诉 SQL Server 我们正在查找属性——就像在 C#或 VB.NET 中一样。

10.6.3 删除数据类型

正如你所期待的，删除用户定义数据类型的语法与其他的删除语句一样：

```
DROP TYPE [<schema name>.<type name>[:;]]
```

这样就行了——也许吧。

为什么这次是“也许”呢？嗯，如果这里有很多对象引用该数据类型，那么该 DROP 操作就不被允许，而且会失败。因此，如果表中存在使用了该类型的列，那么试图删除它就会导致失败。同样地，如果存在使用了这种类型的模式绑定视图、存储过程、触发器或函数，那么删除操作也会失败。

提示：

注意，SQL Server 的其他地方也会出现此类限制(比如删除作为外键引用目标的表)，但是在那些情况下的受限程度比这个要小(实际上在数据库中使用它都会阻止删除操作)，因此，我觉得没有必要再指出这一点了(它们是不言而喻的)。

10.7 小结

嗯，在阅读本章时如果你没有一路感叹“哇，这真是强大啊”，那么我只能猜测你根本没有阅读本章的具体内容，而是直接跳到“小结”这一节来了。这就是本章要介绍的内容，赋予你能力(在某些情况下，这只是非常简单的任务，但无法用前面的方法来完成)完成非常复杂的操作。

本章中有很多问题需要思考。表值参数大大减少了客户端的循环次数，并允许你在单个父存储过程中捆绑更多的逻辑。

使用程序集的时候要非常小心。思考自己要做的事情，在真正执行之前，先彻底分析程序集要做的每一步操作。如果要创建一些运行时间很长的进程，要考虑因此增加的等待时间。如果要进行外部调用，就需要考虑外部的依赖性——这些外部进程有多可靠？你必须知道这一点，因为你的系统的可靠程度取决于所调用的外部系统。

像平常一样，考虑自己需要什么，并且不让自己的解决方案比自己需要的更复杂。不过要记住，一开始看上去非常复杂的解决方案有可能最后会非常简单。我曾经见过存储过程解决了一些看上去无法解决的 T-SQL 问题。保持自己的系统远离程序集似乎可以简化问题，不过什么才是最好的呢：一个 300 行的复杂的 T-SQL 存储过程，还是一个仅有 25 行，非常简练的包含声明语句的程序集？

请做出明智的选择吧。

第 11 章

事务和锁

“怎么办？”这是我在构思本章时想到的问题。鉴于我经常向所谓的“新手”讲授这个主题（见清华大学出版社引进并出版的《SQL Server 2008 编程入门经典》），所以我犹豫是否要将其从“高级篇”中拿掉。不过问题是虽然它在本质上是基础内容，但是事务和锁却是许多高级用户都没有完全“掌握”的基本知识。尽管本章没有特别难的内容，但是事务和锁似乎是数据库领域中最容易引起误解的两个领域。

读过本章再回去工作会有一种很充实的满足感。掌握了这种看似“初级”（至少我认为这是基本概念）的概念会使你看起来更像一个真正的高级程序员。

本章将涉及：

- 研究事务
- 讨论 SQL Server 日志及“检查点”的工作方式
- 解开对锁的困惑

现在，你可能认为我将你视为新手，事实确实如此。多个位置介绍的内容都比面向初学者的内容更深刻。

11.1 事务

事务是原子性的。原子性是必须作为一个完整的单元执行某些操作。从数据库的观点来看，在执行一条或多条语句的组合时，要么它们全部被执行，要么全部不执行。

在处理数据的时候，常常希望一件事情发生的同时也发生另一件事情，或者两者都不发生。实际上有时候需要 20 件（或者更多）事情同时发生，或一件事情都不发生。下面看一个典型的例子。

假设你是一个银行家，莎丽进来了希望从支票帐户向储蓄帐户转帐 1 000 美元。当然，你很乐意为她服务，所以就遵照她的要求执行了相关的操作。

此时在后台出现了这样的情况：

```
UPDATE checking
SET Balance = Balance - 1000
WHERE Account = 'Sally'
```



```
UPDATE savings
SET Balance = Balance + 1000
WHERE Account = 'Sally'
```

它简单的描述了将要发生的事情，不过它抓住了事情的要点：必须发出两个不同的语句——每条语句针对不同的帐户。

现在，如果执行了第一条语句，但是没有执行第二条语句，那么会出现什么情况呢？Sally 可能会损失一千美元！也许短时间内你还感觉不错(刚刚赚了一千美元！)，但是这种感觉的持续时间不会很长。在这个下午，你的客户一定会对你大发雷霆，之后离开你的银行。没有储户的话，银行业务是很难维持的。

你需要一种方式确保若执行第一条语句，第二条语句也会被执行。首先，似乎不可以确定是否出现了那样的错误。所有的事情都可能出错，包括硬件故障和违反数据完整性规则这样的小事。不过幸运的是，有一种针对需要作为同一个整体而实现的方法。让我们可以忘记第一条语句曾经发生。至少可以作出这样的强制：如果一件事没有发生，那么所有的事都不会发生——至少在事务的作用域中都是如此。

不过为了掌握事务的概念，必须定义非常明确的边界。一个事务必须拥有确定的起点和终点。在 SQL Server 中所执行的每个 SELECT、INSERT、UPDATE 和 DELETE 语句实际都是隐式事务的一部分。甚至即使只执行一条语句，也会将这条语句视为一个事务进行处理。与这条语句有关的所有事都会被执行，或者都不被执行。毫无疑问，默认情况下，一个事务的长度就是一条语句。

注意：

再次强调：在 SQL Server 中执行的每条 SELECT、INSERT、UPDATE 和 DELETE 语句都是隐式事务的一部分。即使只执行一条语句，也将其看作一个事务。有关该语句的所有事情要么被一起执行，要么都不执行。

不过如果希望同时执行多条语句或令多条语句同时不被执行——例如前面提到的银行的例子，那么应该怎么做呢？这种情况下，需要一种方法标记事务的开始和结束位置，以及事务的成功或失败。要实现这个目的，可以使用几条 T-SQL 语句在事务中“标记”这些点。我们可以：

- 开始(BEGIN)一个事务——设置起始点；
- 提交(COMMIT)一个事务——让事务成为数据库中永久、不可逆转的部分；
- 回滚(ROLLBACK)一个事务——本质上说就是希望忘记已经发生的事；
- 保存(SAVE)一个事务——创建一个特定的标记，允许用户只做局部的回滚。

在将它们组合应用到我们自己的第一个事务之前，先看看它们各自的含义。

11.1.1 BEGIN TRAN

事务的开始可能是事务过程中最容易理解的概念。它在整个生命期中的唯一目的就是指定单元的开始点。如果基于某种原因，我们不能或不想提交事务，那么这个点就是所有数据库活动被回滚到的那个位置。这就是说，数据库会忽略此点之后的每一件没有被真正提交的事情。

下面给出了它的语法：

```
BEGIN TRAN[SACTION] [<transaction name>|<@transaction variable>]
[WITH MARK [<description>']] [;]
```

WITH MARK 部分可有可无, 而且在实践中很少会用到它。不过不要认为它无关紧要——事实正好相反。

如果要标记事务, 必须给出事务名(注意这是个名称, 不是对需要的描述。如果不标记事务, 那么这个名称是可有可无的)。提供的描述最多只能有 255 个字符(它可以更长, 不过如果是这样的话, 描述内容会被截短, 保留 255 个字符)。

关于标记事务

从 SQL Server 2005 开始, 我们就有能力在从备份和日志中将数据库还原到某个特定的时间点上。可以指定备份要前向回滚的某个特定时间(利用日志), 而且 SQL Server 会恢复那个点之后发生的所有一切, 之前的一切不做改动。标记事务通过在事务日志中创建一个特殊的标识使自己具有了这种能力。在执行时间点恢复的时候, 可以指定某个标记的事务作为要恢复的点, 不用指定时间, 只需要指定标记的描述。下面列出了可以使用的事务:

- 在发生重要操作的时候做一个标记, 以便于在需要的时候可以恢复到这个点。
- 标记两个数据库的行为, 以便于数据库可以还原到同步的数据点。

标记自己的时间点这个概念使用起来很方便。虽然这是一种极端的使用方法, 但是你会发现, 在需要与外部系统(甚至不必是一个 SQL Server 系统)在备份上完成同步的时候, 它还是非常有效的。

11.1.2 COMMIT TRAN

完成事务的终点是提交事务。在此处发出 COMMIT TRAN, 事务就会被认为是持久的。也就是说, 事务的结果会被永久保存, 即使出现系统错误也可以恢复(只要拥有备份, 或者数据库文件未受到物理损坏)。“撤消”已完成事务的唯一方法是执行一个新的事务, 从功能上说, 它的作用就是第一个事务的逆转。

COMMIT 的语法同 BEGIN 非常类似:

```
COMMIT [TRAN[SACTION] [<transaction name>|<@transaction variable>]] [;]
```

提示:

与将 EXECUTE 截短为 EXEC 的方法类似, TRANSACTION 可以被截短为 TRAN。虽然 TRANSACTION 的形式更加完整和清楚, 但是你会发现, 绝大多数的开发人员在实践中都使用简短的 TRAN(我能说什么呢? 很显然, 我们都很懒惰)。

SQL Server 也会以下面的形式支持一种更加符合 ANSI 标准的语法形式:

```
COMMIT [WORK] [;]
```

这个语法不支持事务名绰号, 而且虽然它更符合服从 ANSI 标准, 但是处于某种原因(可能是因为它最近才被加入到产品中), 在实际的应用场合中, 它在 SQL Server 中应用很少。

11.1.3 ROLLBACK TRAN

只要我想起 ROLLBACK 这个词, 就会想起一部名为《公主新娘》的老电影。如果你看过这个电影(如果你没看过, 我强烈推荐你看一看)就会知道其中 Vizzini(电影中的一个巨才)这个角色总

是在说,“如果任何事出错就回到起点”。

这是一些非常好的建议。一条 ROLLBACK 可以返回到起点。也就是说,令自己的事务返回起点。在相关联的 BEGIN 语句后发生的所有事情都会被忘掉。在回到起点的时候,唯一的例外是使用所谓的保存点的时候,这里会简单介绍这个概念的。

除了允许设置保存点之外,ROLLBACK 的语法看起来也来非常类似:

```
ROLLBACK TRAN[SACTION] [<transaction name>|<save point name>|
    <@transaction variable>|<@savepoint variable>][;]
```

还可以使用类似于 COMMIT 的 ANSI 语法:

```
ROLLBACK [WORK][;]
```

11.1.4 SAVE TRAN

保存事务,在本质上就是创建书签。可以为书签命名(书签可以有多个)。“书签”创建好之后,就可以在回滚操作中引用这个书签。这样做的好处是只需命名要回滚到的保存点,就可以回滚到你所希望的代码中任意一个准确位置。

注意:

保存点的命名必须符合第 1 章中讨论过的标识符命名规则。不过,这里有一点不同,保存点命名的长度被限制为 32 个字符。

其语法非常简单:

```
SAVE TRAN[SACTION] [<save point name>| <@savepoint variable>][;]
```

关于保存点,要记住的是它们是在 ROLLBACK 上被清除——即,即使已经保存了 5 个保存点,一旦执行了一次 ROLLBACK 操作,那么这些保存点就都消失了。可以再次设置新的保存点,并回滚到这些保存点上,不过一旦再次执行 ROLLBACK 操作,这些保存点就会再次消失。

提示:

在我第一次遇到保存点的时候,它们是我的最困惑的问题。联机丛书指出,在回滚到一个保存点之后,必须使事务有一个合理的结论(这在技术上是正确的)。困惑的原因是在联机丛书中提到的一处暗示,似乎是说在执行 ROLLBACK 或 COMMIT 的时候不能使用更多的保存点。事实并非如此。你只是不能使用 ROLLBACK 之前声明的保存点,之后声明的保存点还是可以使用的。

下面用一些代码对它进行测试,查看将不同类型的 TRAN 命令混合使用时,会出现什么后果。输入下面的代码,然后运行对这段程序的注解:

```
USE AdventureWorks2008; -- We're making our own table - what DB doesn't matter

-- Create table to work with
CREATE TABLE MyTranTest
(
    OrderID INT PRIMARY KEY IDENTITY
);
-- Start the transaction
BEGIN TRAN TranStart;
```

```
-- Insert our first piece of data using default values.
-- Consider this record No1. It is also the 1st record that stays
-- after all the rollbacks are done.
INSERT INTO MyTranTest
    DEFAULT VALUES;

-- Create a "Bookmark" to come back to later if need be
SAVE TRAN FirstPoint;

-- Insert some more default data (this one will disappear
-- after the rollback).
-- Consider this record No2.
INSERT INTO MyTranTest
    DEFAULT VALUES;

-- Roll back to the first savepoint. Anything up to that
-- point will still be part of the transaction. Anything
-- beyond is now toast.
ROLLBACK TRAN FirstPoint;

-- Insert some more default data.
-- Consider this record No3 It is the 2nd record that stays
-- after all the rollbacks are done.

INSERT INTO MyTranTest
    DEFAULT VALUES;

-- Create another point to roll back to.
SAVE TRAN SecondPoint;

-- Yet more data. This one will also disappear,
-- only after the second rollback this time.
-- Consider this record No4.
INSERT INTO MyTranTest
    DEFAULT VALUES;

-- Go back to second savepoint
ROLLBACK TRAN SecondPoint;

-- Insert a little more data to show that things
-- are still happening.
-- Consider this record No5. It is the 3rd record that stays
-- after all the rollbacks are done.
INSERT INTO MyTranTest
    DEFAULT VALUES;

-- Commit the transaction
COMMIT TRAN TranStart;

-- See what records were finally committed.
SELECT TOP 3 OrderID
FROM MyTranTest
ORDER BY OrderID DESC;

-- Clean up after ourselves
DROP TABLE MyTranTest;
```

首先, 创建一个测试表:

```
-- Create table to work with
```


SQL Server 2008 高级程序设计

```
CREATE TABLE MyTranTest
(
    OrderID    INT    PRIMARY KEY    IDENTITY
);
```

既然已经创建了表供本例子使用，那么这里使用的数据库就不会真正影响本次演示了。

现在可以开始事务了。这开始了“要么全部执行，要么全部都不执行”的语句组合。随后插入一行。在这个关键位置，只插入了一行：

```
-- Start the transaction
BEGIN TRAN TranStart;

-- Insert our first piece of data using default values.
-- Consider this record No1. It is also the 1st record that stays
-- after all the rollbacks are done.
INSERT INTO MyTranTest
    DEFAULT VALUES;
```

接下来，创建一个名为 **FirstPoint** 的保存点并插入另一行。此时，已经插入了两行，不过要记住，它们尚未被提交，因此数据库并不将其视为自己的一部分：

```
-- Create a "Bookmark" to come back to later if need be
SAVE TRAN FirstPoint;

-- Insert some more default data (this one will disappear
-- after the rollback).
-- Consider this record No2.
INSERT INTO MyTranTest
    DEFAULT VALUES;
```

随后进行 **ROLLBACK** 操作——说得更明白一点，并不是回滚到起始点，而是只回滚到 **FirstPoint** 点。在 **ROLLBACK** 的作用下，**ROLLBACK** 与 **FirstPoint** 保存点之间的所有操作都会被撤消。鉴于在 **ROLLBACK** 和 **SAVE** 之间有一条 **INSERT** 语句，因此该 **INSERT** 语句就被回滚了。此时，我们就退回到只插入一行的状态。任何对保存点的引用企图都会失败，因为在 **ROLLBACK** 之后所有的保存点都被复位了：

```
-- Roll back to the first savepoint. Anything up to that
-- point will still be part of the transaction. Anything
-- beyond is now toast.
ROLLBACK TRAN FirstPoint;
```

插入另一行，就会回到总共插入两行的状态上。这里还创建了一个新的保存点。这样做非常有效，因为这个保存点是在 **ROLLBACK** 之后创建的，所以现在可以引用它了，：

```
-- Insert some more default data.
-- Consider this record No3. It is the 2nd record that stays
-- after all the rollbacks are done.

INSERT INTO MyTranTest
    DEFAULT VALUES;

-- Create another point to roll back to.
SAVE TRAN SecondPoint;
```

现在可以插入另一行，这样当前依然有效的插入数变为 3：

```
-- Yet more data. This one will also disappear,  
-- only after the second rollback this time.  
-- Consider this record No4.  
INSERT INTO MyTranTest  
    DEFAULT VALUES;
```

现在执行另一个 ROLLBACK 操作，这次引用新的保存点(因为执行过第一次 ROLLBACK 操作后 FirstPoint 就被重新复位了，所以这恰好是目前唯一有效的保存点)。这次在它及其引用的保存点之间的所有操作都会被撤消——在这种情况下就是一条 INSERT 语句。这样就会回到 2 条 INSERT 语句仍然有效的情况：

```
-- Go back to second savepoint  
ROLLBACK TRAN SecondPoint;
```

然后再调用另一条 INSERT 语句，将作为事务一部分的 INSERT 语句的总数增加至 3：

```
-- Insert a little more data to show that things  
-- are still happening.  
-- Consider this record No5. It is the 3rd record that stays  
-- after all the rollbacks are done.  
INSERT INTO MyTranTest  
    DEFAULT VALUES;
```

最后(对这里的事务来说)也是非常重要的一点，这里发出 COMMIT TRAN 语句锁定自己的事务，并将其它永久保存在数据库历史中：

```
-- Commit the transaction  
COMMIT TRAN TranStart  
  
-- See what records were finally committed.  
SELECT TOP 3 OrderID  
FROM MyTranTest  
ORDER BY OrderID DESC;
```

注意：

注意，如果任一 ROLLBACK 语句不包含保存点的名字，或者包含了一个用 BEGIN 语句设置的名字，那么整个事务都必须被回滚，此时可以考虑关闭该事务。

在完成这个事务的时候，可以调用少量语句来显示这 3 行。观察结果可以发现在事务中插入并删除行的方法：

```
OrderID  
-----  
5  
3  
1  
  
(3 row(s) affected)
```

这样说明已经很充分了，进行了隔行插入。

最后，自己完成清理工作，此时不需要再对事务进行其他操作了。


```
DROP TABLE MyTranTest;
```

11.2 SQL Server 日志的工作方式

在了解 SQL Server 如何追踪数据库中的行为之前，必须充分理解事务的概念。要知道，你所认识的数据库很少是所有数据的完整版本。除了正好已将所有数据都写入了磁盘的极少数情形外，数据库中的数据是由物理数据库文件中的数据和自从上一次检查点以来的已被提交到日志中的所有事务组成的。

在常规数据库操作中，绝大多数的操作都被“记录”到事务日志中，而不是被直接写入数据库。检查点是一个周期性的操作，它强制将当前使用的数据库中所有的脏(dirty)页写入磁盘。脏页是在被读入到缓存之后又被修改的日志或数据页，不过这个修改尚未被写入磁盘。没有检查点，日志会填满并(或)使用所有可用的磁盘空间。图 11-1 中的框图显示了整个工作流程。

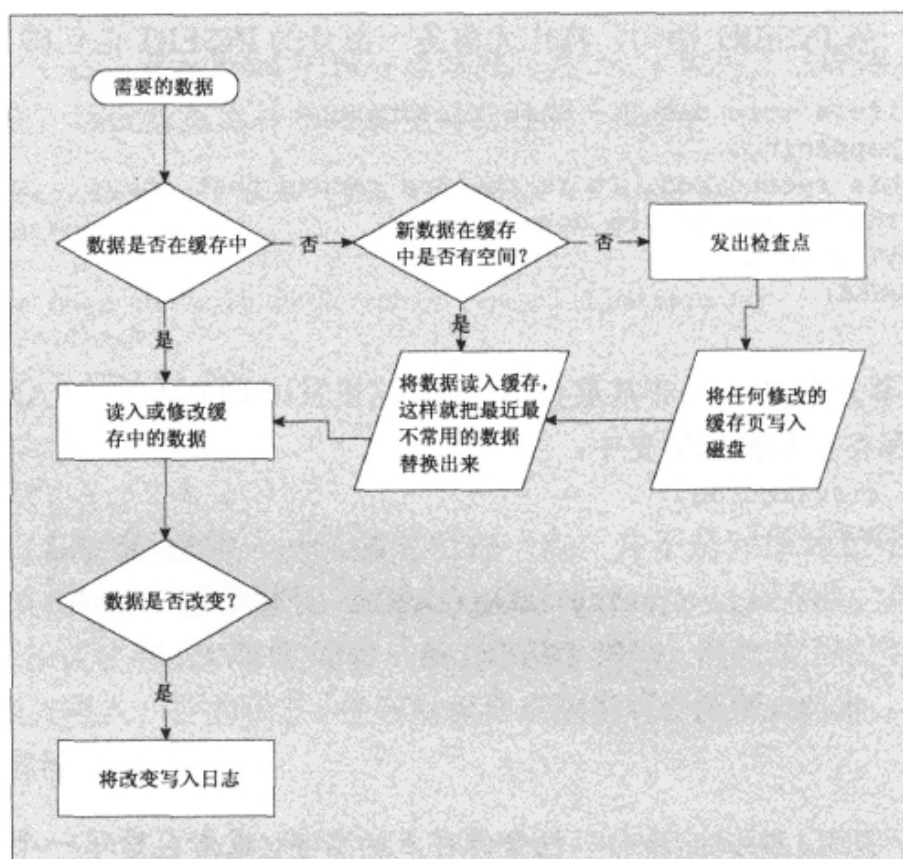


图 11-1

提示:

不要误以为所有的这一切意味着必须执行一些特殊的操作才能从缓存中取出数据。SQL Server 会处理一切。为了方便你理解日志的工作方式，以及处理事务所需的步骤，这里给出了相应的信息。缓存中是否有数据对性能的影响很大，所以在寻求最佳性能时，了解何时记录日志以及何时将数据存入和取出缓存是至关重要的。

注意，发布检查点的原因并不仅仅是需要将数据读入已经存满了的缓存中。在下面的情况下也可以发出检查点：

- 通过一条手动的语句——使用 CHECKPOINT 命令；
- 在常规的关闭服务器过程中(除非使用了 WITH NOWAIT 选项)

- 在修改数据库选项时(例如, 只允许单用户, 只允许 dbo 等);
- 使用“简单恢复”选项, 同时已经占用了 70% 的日志空间;
- 当从上一检查点以来的日志数据(通常称作活动日志部分)超过了在恢复间隔选项中设置的时间内服务器可以恢复的数据大小。

下面更加仔细地查看这些选项。

11.2.1 使用 CHECKPOINT 命令

存在一种方法可以在数据库中手动发布检查点(不过可能是最不常用的方法)。可以在任何时候使用下面的指令完成这个操作:

```
CHECKPOINT
```

就是这么简单。

SQL Server 在检查点方面表现很好。因此很少有机会需要手工发布检查点。

提示:

在一种场合下必须这么做: 在开发周期内, 为数据库打开了简单恢复模型(你一定不希望在生产数据库中激活这个选项)。这种情况在数据库的开发过程中非常普遍, 数据库执行的操作运行时间很长, 而且会很快填满日志。虽然可以自己执行合适的命令来截短日志, 不过 CHECKPOINT 是一种比较简单、快速的方法, 而且在使用简单恢复模型时, 具有相同的效果。

11.2.2 在服务器正常关机时执行

有时候 SQL Server 关机特别慢, 你是否对此产生过好奇? 在系统关机的过程中, 除了释放内存单元、运行其他销毁例程以便卸载系统, SQL Server 必须在关机过程开始之前发出一个检查点。这意味着在继续关机操作之前, 必须取出那些已经被提交到日志中的数据并将其写入物理数据库中。使用下面的方法终止服务器时, 会生成检查点:

- 使用 Management Studio;
- 在提示命令窗口(DOS 窗口可能会调用它)中使用 NET STOP MSSQLSERVER 指令;
- 使用 Windows 控制面板中的“服务”图标, 选择 MSSQLSERVER 服务, 并单击“停止”按钮。

注意:

不同于 Checkpoint on Recovery, 这是我喜欢的方法。我喜欢的事实是: 所有提交的事务都在物理数据库中(不被分割成日志和数据库), 我觉得这更加简洁, 降低了破坏数据的可能性。

如果愿意的话, 也可以绕过关机延迟。要实现这个目的, 必须在 T-SQL 中使用 SHUTDOWN 命令。只需要在关机语句中添加 WITH NO WAIT 关键字, 就可以消除与检查点有关的和检查点本身的延迟,:

```
SHUTDOWN [WITH NO WAIT]
```

注意, 除非在编程方面对服务器关闭有特殊的需求, 否则我强烈建议不要这样做。因为这样

会导致后续重启时间延长,以便在服务器上恢复数据库,而且这样表示关机不彻底(某些数据只存在于日志中,而并不是所有的数据都存放在数据库文件中)。

11.2.3 在更改数据库时执行

不管数据库选项如何改变(例如,使用 `sp_dboption` 或者 `ALTER DATABASE`),只要改变数据库选项,就会发布一个检查点。在对数据库做真正更改之前发生检查点。

11.2.4 在启用 Truncate on Checkpoint 选项时执行

如果已经启用了 `Truncate on Checkpoint` 数据库选项(通常会在开发数据库阶段这么做),那么一旦日志占用的大小超过总日志空间的 70%,SQL Server 就会自动发出一个检查点。

11.2.5 在恢复时间超过设置的恢复间隔时执行

正如不久前看到的(随后还会看到更深入的细节),每次启动 SQL Server 的时候,SQL Server 会执行一个所谓的恢复过程。只要估计的恢复时间超过在数据库恢复间隔时间选项中设置的时间,SQL Server 就会发出一个检查点。默认情况下,恢复间隔时间被设置为 0——这表明 SQL Server 会自己决定这个时间(在实践中,这个时间大约是 1 分钟)。

11.2.6 失败与恢复

每次启动 SQL Server 都会发生一次恢复过程。SQL Server 获取数据库文件,并在其上应用自上个检查点以来向日志提交的所有修改(将它们写入物理数据库文件)。所有尚未被提交的日志中的更改都会被回滚——即,它们确实被遗忘了。

它的工作方式取决于数据库中的事务的发生方式。假设有 5 个事务跨越了日志,如图 11-2 所示。

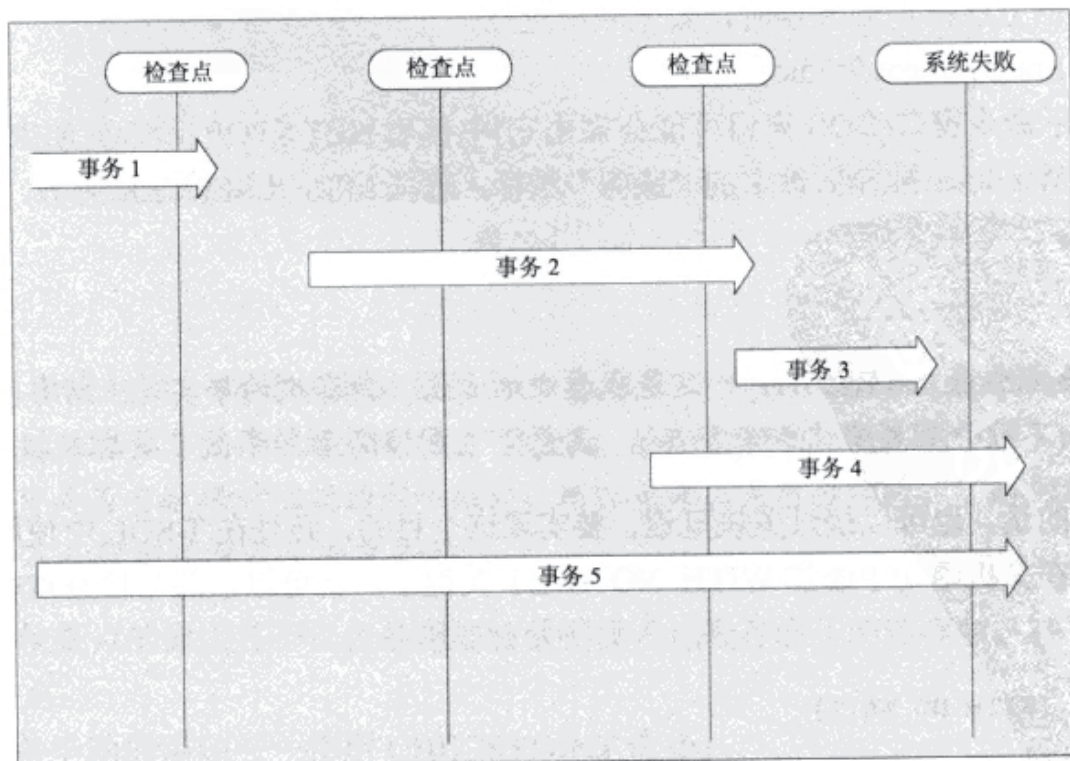


图 11-2

下面看看这些事务是如何依次发生的。

1. 事务 1

绝对不会发生任何事。这个事务已经通过了检查点，并被完整提交到数据库中。不需要再为恢复做任何事，因为数据缓存已经读入的所有数据已经可以反映提交的事务了。

2. 事务 2

尽管这个事务在发布检查点的时候已经存在，但是它没有被提交(事务仍在继续)。没有被提交的事务不能真正参与到检查点中。所以这个事务会被“向前滚动”。这可以很好的方式，缓存需要重新读取所有相关的页面，并使用日志中的信息重新运行在该事务中所运行过的所有语句。完成了上述操作，事务看起来和系统故障之前的状态一样。

3. 事务 3

虽然看上去不同，不过如果站在希望完成的工作角度来看，这个事务同事务 2 确实一摸一样。再次声明，因为事务 3 在最后一个检查点的时刻尚未完成，所以它和事务 2 一样不能参与到检查点中。唯一的不同是，当时并不存在事务 3。不过从恢复的角度来看，这没有什么不同——只有在事务提交被执行的时候，结果才会有所不同。

4. 事务 4

在系统出现故障的时候，还没有完成这个事务，因此这个事务也必须被回滚。实际上，按照行数据的观点来看，这个事务根本没有发生过。用户必须重新输入数据，所有的操作都必须从起点重新开始。

5. 事务 5

这个事务和事务 4 差别不大。因为这个事务运行的时间比较长，所以它看上去有些不同，不过本质上并没有什么差别。在系统出现故障的时刻，不会提交事务，因此该事务也一定会被回滚。

11.2.7 隐式事务

主要是为了同其他主流的关系数据库管理系统(如 Oracle 或 DB2)相兼容，SQL Server 支持所谓的隐式事务(在默认情况下是关闭的，但是如果需要的话可以开启它)。隐式事务不需要 BEGIN TRAN 语句——相反，它们自动从第一条语句处开始。然后继续执行，直至你发出 COMMIT TRAN 或 ROLLBACK TRAN 语句为止。下一个事务则继续从下一条语句开始。

从理论上来说，这样做是为了确保每条语句都能作为事务的一部分。SQL Server 也希望每条语句都能作为事务的一部分，但是，在默认情况下，采用了不同的方法——如果没有 BEGIN TRAN 语句，那么 SQL Server 就假设你拥有的事务只包含一条语句，自动为你开始和结束这个事务。而在其他的一些系统中，会看到隐式事务的方法。这些系统会假设每条语句都是事务的开始，所以要求使用 COMMIT 或 ROLLBACK 来显式的结束每个事务。

默认情况下，IMPLICIT_TRANSACTIONS 选项是关闭的(连接处于自动提交事务的模式)。可以执行下面的命令来启用这个选项：


```
SET IMPLICIT_TRANSACTIONS ON;
```

此后，下面的任何一条语句都会启动一个事务：

```
CREATE  
ALTER TABLE  
GRANT  
REVOKE  
SELECT  
UPDATE  
DELETE  
INSERT  
TRUNCATE TABLE  
DROP  
OPEN  
FETCH
```

除非发出 COMMIT 或 ROLLBACK，否则事务会继续。注意隐式事务选项只会影响当前的连接——这个选项对其他用户仍然是关闭的，除非他们也执行了 SET 语句。

注意：

隐式事务选项是非常危险的，我强烈建议不要使用这个选项，除非有充分的理由启用它(比如要与另一个系统中编写的代码兼容)。

这里有一种常见的场景：一个用户打电话来说“我在过去的半个小时内一直在插入数据，但是这种修改却没有得到显示。”所以你运行 DBCC OPENTRAN，发现那里已经有一个事务存在很久了——你可以猜测一下发生了什么事情。用户打开了一个事务，不过在提交事务之前，不会显示他或她所做的任何修改。用户可能已经使用一条显式的 BEGIN TRANS 语句完成这个操作，不过他或她也可能已经执行了一些代码，启动了隐式事务而且后面未关闭它。这样做的结果将是一塌糊涂。

11.3 锁和并发

并发是所有数据库系统的主要问题。它描述了两个或多个用户尝试同时处理一个对象的概念。交互操作的本质对每个用户来说都是不同的(更新、删除、读取和插入)，处理这种对象更改控制所发生的冲突的理想方法，取决于这些用户正在做的工作内容，以及这些工作的重要性。用户越多(更确切地讲，事务越多)，即在同一时刻可以成功完成的事情越多，并发性也就越高。

在联机事务处理(Online Transaction Processing, OLTP)环境下，在数据中首先要处理的事情通常就是并发，它是本书推出的大多数数据库概念中的焦点。联机分析处理(Online Analytical Processing, OLAP)则通常作为事后的内容；它并不需要在那里出现，但是的确出现了。处理并发问题对系统的性能至关重要。在数据库中，处理并发的基本方法是一个称之为锁定的过程。

锁是一种机制，用来防止进程对对象进行操作的时候，与已经在该对象上执行的某些操作发生冲突。也就是说，如果某人之前已经在对象上进行过操作，你就不能再对这个对象进行操作了。能否在对象上执行操作，取决于其他的用户正在执行什么操作。因为它也是对已经完成事物的一种描述，所以系统会知道第二个操作过程是否同第一个过程兼容。例如，有 1、2、10、100、1000 或任意数量的用户连接到系统中，只要他们希望以只读的方式获取记录，就可以同时共享相同的

数据片段。把它想象成一个水晶商店：只要不移动商品，购买商品，或者改动商品，那么很多人都是可以观看商品(甚至是相同的商品)。如果多个人同时移动、购买或改动商品，你就可能要担心水晶是否会损坏。正是由于这个原因，店主一般会密切注意商品，而且通常他们会决定由谁最先拿起它。

SQL Server 的锁管理器就是这个店主。当你进入 SQL Server “商店”，锁管理器会询问你的意图——也就是你要做什么。如果你说“只是随便看看”，而且这里所有的人都“只是随便看看”，那么锁管理器就会让你进入。如果你希望“购买”(更新或删除)某些东西，那么锁管理器就会检查，看看别人是否已经在这儿了。如果是这样，你就必须等待，任何在你之后来到的人也必须等待。当允许你进入“购买”商品的时候，别人也就不能进入了，直至你完成了操作为止。

通过这种操作机制，SQL Server 能够帮助我们避免那些因为并发问题而生成的许多不同错误。我们会检查发生并发错误的可能性，并了解应该如何设置事务隔离级别才能防止这些错误。不过现在先看看什么能锁，什么不能锁，以及有哪些种类的锁可供使用。

11.3.1 通过锁可以防止的问题

锁可以解决 4 个主要的问题：

- 脏读(Dirty read)；
- 不可重复读(Non-repeatable read)；
- 幻影(Phantoms)；
- 丢失更新(Lost updates)。

每项都会各自产生一系列的问题，可以混合使用那些设置了合理的事务隔离级别的解决方法来解决这些问题。为了让你回过头再看这一章的时候能够发现这对工作是非常有帮助的，接下来会给出一些每种情况下哪种事务隔离级别合适的信息。很快就会看到完整的隔离级别，不过现在，首先要确定自己已经理解了这些问题。

1. 脏读

当事务读取一条记录，而该记录是另一尚未完成的事务的一部分时，就会发生脏读。如果第一个事务正常完成，那么这样做似乎不会出现问题。但是如果这个事务被回滚情况又会怎么样呢？你会从事务中得到信息，从数据库的角度来说，并不存在这样的信息！

让我们在表 11-1 中看看其连续的步骤：

表 11-1

事务 1 命令	事务 2 命令	逻辑数据库值	未提交的数据库值	事务 2 显示内容
BEGIN TRAN		3		
UPDATE col=5	BEGIN TRAN	3	5	
SELECT anything	SELECT @var=col	3	5	5
ROLLBACK	UPDATE anything	3		5
	SET whatever=@var			

这里出现问题了!!!

事务 2 现在正在使用一个已经失效的值！如果试图返回审核，会发现根本无法追踪要该数字的来源，这件事很令人头痛。

幸运的是，如果使用 SQL Server 默认的事务隔离级别(称作 READ COMMITTED，后面“设置隔离级别”一节中会介绍它)，就不会发生这种事。

2. 不可重复读

很容易把它同脏读混淆。不过不用担心——这只是术语而已。概念才是重要的。

在一个事务中两次读取记录，并且在两次读取之间，只有一个事务修改了这个数据，那么就会导致不可重复读。要说明这种情况，再回顾一下银行的例子。记住，这里不希望帐户上的值被降到 0 美元以下，如表 11-2 所示：

表 11-2

事务 1	事务 2	@Var	事务 1 认为表中的值 是什么	表中的值
BEGIN TRAN		NULL		125
SELECT @Var= Value FROM table	BEGIN TRAN	125	125	125
	UPDATE value , SET Value=value-50			75
IF @Var>=100	END TRAN	125	125	75
UPDATE value, SET value= Value-100 (Finish, wait For lock to Clear, then Continue)		125	125(等待清除锁)	75
		125	75	-25(如果没有 CHECK 约束 强制大于 0)或错误 547(如果有 CHECK 约束)

这里又出错了。事务 1 已经预先扫描(在某些情况下这是很好的行为)以确认值是有效的，而且事务可以继续下去(在帐户中有足够的钱)。问题在于，由于进行了 UPDATE 操作，事务 2 让事务 1 做出另外的决定性举动。如果表上没有任何 CHECK 约束可以避免负数值，那么实际上这个值会被设置为-25——尽管在逻辑上看来，这里已经通过 IF 语句来防止此事的发生。

只有两种方法可以防止出现这种问题：

- 创建 CHECK 约束并监视 547 号错误；
- 设置隔离级别(ISOLATION LEVEL)为 REPEATABLE READ 或 SERIALIZABLE。

CHECK 约束的作用似乎非常明显。它在这里所体现的是被动地做某些事，而不是主动地使用这个方法。尽管如此，在大多数情况下，我们可能会遇到不可重复读，所以在大多数环境下这

是更好的选择。

下面很快就会完整地介绍隔离级别，不过在目前阶段可以肯定地说，将它设置为 REPEATABLE READ 或 SERIALIZABLE，你带来麻烦可能和它所能解决的事情一样多(也许还要更多)。毕竟，它只是一个选项。

3. 幻影

不(这里不是要讨论歌剧)，这里要讨论的是看上去不可思议的记录，在执行了 UPDATE 或 DELETE 语句后，这些记录似乎并没有受到任何影响。在系统正常运行的情况下，这种问题很可能会以合理的面目出现，而且不需要任何提示警告。这里用一个典型的例子说明它是如何发生的。

假设你正在开一个快餐店。如果开的是那种典型的公司，就可能会有很多雇员拿着政府所规定的“最低工资”。政府刚刚决定将最低工资标准从每小时 6.55 美元提高到每小时 7.25 美元，你希望通过更新名为 Employees 的表，将那些收入少于每小时 7.25 美元的工资水平提高到新的最低工资水平。你会说这很简单，同时执行一个非常简单的语句：

```
UPDATE Employees
SET HourlyRate = 7.25
WHERE HourlyRate < 7.25;

ALTER TABLE Employees
ADD CONSTRAINT ckWage CHECK (HourlyRate >= 7.25);
GO
```

这是小事一桩，对吗？错了！为了说明这个问题，这里说你肯定会得到一条错误消息：

```
Msg 547, Level 16, State 1, Line 1
ALTER TABLE statement conflicted with COLUMN CHECK constraint 'ckWage'. The
conflict occurred in database 'FastFood', table 'Employees', column 'HourlyRate'.
```

因此，你会快速运行一条 SELECT 语句，检查少于 7.25 美元的值，肯定会找到一条记录。问题来得真快，“你怎么会在这里！我刚刚才更新了应该修改的内容啊！”你的确运行了这条语句，而且它运行得很好——不过这里只是遇到了幻影。

这种幻影读取的实例实属罕见，只有在特定的环境中才会发生。简言之，正在运行 UPDATE 更新的时候，如果另外有人恰好在此时执行了 INSERT 语句，就会出现这种情况。因为它是一个完整的新行，它上面没有锁，所以 INSERT 语句也执行得很好。

解决这个问题的唯一办法就是将事务的隔离级别设置为 SERIALIZABLE，此时对表的所有更新都不能在 WHERE 子句内部，否则它们就会被锁定。

4. 丢失更新

当成功地将一个更新写入数据库中，但是又被另一个事务意外地覆盖时，就会发生丢失更新的现象。我似乎能听到你的惊叹，“呀！这怎么可能发生呢？”

丢失更新可以发生在两个事务对一个完整的记录进行读取的时刻，一个事务将更新信息写入记录，而另一个事务也将更新信息写入记录。下面看一个例子。

假设你是公司的信用分析师。你接到一个电话，说你的客户 X 达到了他(或她)的信用额度，并希望提高它，因此你要提取这个客户的信息以查看详情。你看到客户的信用额度是 5000 美元，而且似乎总能按时还款。

在你浏览的时候,你所在信用部门的另一个人 Sally,正在提取客户 X 的记录,并更新了地址。在她提取用户记录的时候,记录中的信用额度是 5000 美元。

此时,你决定将客户 X 的信息额度提升到 7500 美元,并按下了回车键。数据库现在会显示对于客户 X,其信用额度已经是 7500 美元了。

现在 Sally 完成了自己对地址信息的更新,但是她使用的是与你相同的编辑屏幕——也就是说,她更新的是整个记录。还记得在她的屏幕上,信用额度显示的是多少吗? 5000 美元。数据库现在又再次把客户 X 的信用额度设置为 5000 美元。你的更新丢失了!

要解决这个问题的方案取决于自己的代码,在读取数据和对数据进行更新的这两个时间间隔之内,如何识别另一个连接对记录的更新操作。使用不同的访问方式,识别机制也会发生很多变化。

11.3.2 可锁的资源

在 SQL Server 中,有 6 种资源是可锁的,它们形成分级层次。锁的级别越高,其粒度就越小(也就是说,如果在某些层叠的操作中,选择的锁定级别越高,被锁定的对象数量也就越多,因为包含它们的对象已经被锁定)。在这里按粒度的升序列出了它们:

- **数据库:** 整个数据库都被锁定。这通常只在更改数据库模式的时候才会发生。
- **表:** 整个表被锁定。它包含了同该表相关联的所有数据对象,包括实际的数据行(其中的每项内容)以及同表相关联的所有索引上的所有键。
- **区段:** 锁定整个区段。记住一个区段由 8 页组成,所以区段锁定意味着锁控制了整个区段,在该区段上的 8 个数据或索引页,以及在这 8 页上的所有数据行。
- **页:** 锁定该页上的所有数据或索引键。
- **键:** 这是在索引上的个别键或一系列键上的锁。在同一个索引页上的其他键不受影响。
- **行或行标识(RID):** 尽管锁从技术上说是放置在行的标识符(一个内部的 SQL Server 构件)上,但是实际上它可以锁定整个行。

11.3.3 锁升级以及锁对性能的影响

升级(Escalation)是指认识到,当锁定的项目数量较少时,维持一个较精细的粒度级别(譬如用行锁来代替页锁)更加合理。不过,越来越多的项目被锁定,维护这些锁所带来的开销会对性能产生影响。它可能会导致锁在一个地方存留的时间过长,并因此带来争用的问题。锁在某个位置上存留的时间越长,有人希望得到这个特定记录的可能性越大。稍微思考一下这种情况,你就会明白这可能是在某个位置上对行为的平衡化,而且这的确是一个锁管理器要利用升级完成的事情。

当要维护的锁的数量达到某个阈值的时候,锁被升级到下一个更高的级别,因为不用再对较低级别的锁进行紧密的管理(释放资源并提高速度,但不考虑争用)。

注意,升级是基于锁的数量,而不是用户的数量。重要性在于可以通过执行大规模更新来单独锁定一个表。行锁定可以升级为页锁定,然后再升级为表锁定。这意味着可以将其他所有的用户锁定在表之外。如果查询使用了多个表,就可以将每个用户都锁定在这些表之外。

提示:

虽然你不希望将其他所有的用户锁定在对象之外,但是有时候,还是需要令所执行的更新生效。关于升级你可以做的事情很少,除了让自己的查询尽可能的有目的性。既然已经认识到将会

发生升级，所以要确保你已经了解了升级在查询中可能带来的后果。

11.3.4 锁模式

不仅要考虑正在锁定的资源的级别，而且还应该考虑在查询中需要的锁定模式。就像有很多资源可以被锁定，也存在很多种锁模式。

一些模式之间是相互排斥的(这意味着它们不能在一起工作)。另一些模式则只是修改其他模式。模式之间是否能够在一起工作，要取决于它们是否兼容。本章后面会更加深入地说明锁之间的兼容性。

1. 共享锁

这是最基本的锁类型。只有读取数据的时候才会用到共享锁——也就是说不需要修改任何东西。共享锁希望成为你的朋友，因为它同其他的共享锁之间是兼容的。这并不意味着它不会给你带来麻烦——虽然共享锁不介意其他类型的锁，但是还有一些其他的锁不喜欢共享锁。

共享锁会告诉其他的锁你现在在这里。这是那种“看我！我是不是特别呀？”的老掉牙的事。它们并没有什么目的性，不过却不容忽略。只是，拥有共享锁的事物可以防止用户在其上发生脏读之类的操作。

2. 排他锁

排他锁名副其实。它与其他类型的锁都不兼容。如果已经存在其他的锁，就无法使用排他锁。当排他锁活动的时候，在资源上也不能创建任何类型的新锁。这样可以防止两个人在同一时刻进行更新、删除或进行其他的相关操作。

3. 更新锁

更新锁(Update Locks)是共享锁和排他锁之间的某种混合体。更新锁是一种特殊类型的占位符。请想象一下——为了进行 UPDATE，需要验证 WHERE 子句(假设存在一个 WHERE 子句)，以推算要更新的行。这意味着在真正进行物理更新之前只需要一个共享锁。在进行物理更新的时候，才需要排他锁。

更新锁意味着在完成对数据的初始扫描，从而确认需要更新的准确内容之后，共享锁变成排他锁。这表明在更新过程中要经历两个不同的阶段：

- 首先，第一个阶段是确认满足 WHERE 子句中的条件的内容(了解要更新的内容)。这是拥有更新锁的更新查询的一部分。
- 其次，在这个阶段，如果确实决定执行更新，就将锁转换为排他锁。否则，转换为共享锁。

这样做的好处是形成了一道屏障，以防止一种死锁变体的发生。死锁本身并不是一种锁类型，而是一种已经形成的矛盾的状态。如果因为另一个锁持有了相应的资源，一个锁就不能完成自己要做的事情从而释放，就会形成死锁。问题是，相对的资源在等待第一个事务上的锁释放。

如果没有更新锁，就会经常突然出现死锁。在共享方式下运行两个更新查询。查询 A 完成了自己的查询，并等待进行物理更新。它希望升级为排他锁，但是却不可以，因为查询 B 正在完成自己的查询。查询 B 完成了查询之后，需要进行物理更新。为此，查询 B 希望将自己升级为排他锁，但是不可以，因为查询 A 还在等待。这就陷入了僵局。

与此相反,更新锁从创建伊始就禁止生成其他的更新锁。第二个事务试图获得更新锁的时候,这个新的事务会被置为等待状态,等待锁超时;在这段时限内,锁是不会被允许的。如果在该超时期限到期之前第一个锁释放了,那么该锁就会被授予新的请求者,随后可以继续这个过程。如果没有,就会产生一个错误。

更新锁只同共享锁和意向共享锁兼容。

4. 意向锁

意向锁(intent lock)是一个真正的占位符,用户处理对象层次问题。想象一下这样的一种情况,假设在行上建立了一个锁,但是另一些人要在页上、或者区上建立锁,或者对一个表进行修改。你肯定不希望有另外的事务以更高的级别在自己的事务周围运行,是吧?

如果没有意向锁,这个更高级别的对象甚至不会知道你已经在一个较低的级别上持有锁。意向锁可以提高性能,因为 SQL Server 不需要检测表中的每行或页上的锁,只需要在表的级别上检测意向锁就可以确认一个事务是否可以安全地锁住整张表。意向锁有三种不同的形式:

- **意向共享锁:** 已经或者将要在一个较低级别的位置上创建共享锁。例如,在页上可以创建页级的共享锁。这种类型的锁只应用在表和页上。
- **意向排他锁:** 它与意向共享锁类似,只是在较低级别的项目上要放置的排他锁。
- **意向排他锁共享:** 共享锁已经或将要位于对象层次结构的较底层,但是意向是要修改数据,所以它在某个时刻会变成意向排他锁。

5. 模式锁

它们有两种类型:

- **模式修改锁(Sch-M):** 对模式的改变被应用到对象上。在 Sch-M 锁定期间,不能针对对象运行任何查询或其他 CREATE、ALTER 或 DROP 语句。
- **模式稳定锁(Sch-S):** 它非常类似于共享锁。这种锁的唯一目标是为了防止在对象上为其他查询(或 CREATE、ALTER 和 DROP 语句)而生成的锁在活动时,出现 Sch-M 锁。它与其他所有类型的锁都兼容。

6. 批量更新锁

批量更新锁(BU)只是表锁的一种变种,其中只有一点(但是很重要)不同。批量更新锁允许并行载入数据——也就是说,表被锁定以防止其他任何“正常”(T-SQL 语句)的活动,不过可以同时执行多个 BULK INSERT 或 bcp 操作。

7. 范围键锁

范围键锁(Ranged Keylocks)只是 SQL Server 更加有效地控制内部独立锁的一种方法。它自己不是锁,只是跟踪哪些锁已经被持有的一种方法。SQL Server 不会在访问的某个范围内持有每行的独立锁,它还有一个锁处理整个范围(因此节省了内存和锁操作)。

11.3.5 锁的兼容性

表 11-3 显示了资源锁定模式的兼容性(列表按锁的强度正序排列)。现有的锁是按列显示的,

请求的锁是按行显示的:

表 11-3

	IS	S	U	IX	SIX	X
意向共享(IS)	是	是	是	是	是	否
共享(S)	是	是	是	否	否	否
更新(U)	是	是	否	否	否	否
意向排他(IX)	是	否	否	是	否	否
意向排他共享(SIX)	是	否	否	否	否	否
排他(X)	否	否	否	否	否	否

还有:

- Sch—S 同所有的锁模式兼容, 除了 Sch—M;
- Sch—M 与所有的锁模式都不兼容;
- BU 只同模式稳定锁和其他批量更新锁兼容。
- 范围 S-S、范围 S-U、范围 I-N 和范围 X-X 都是范围锁, 它们与相应的 S、U 和 X 锁类型匹配, 而且在范围 I-N(N 代表空)中, 锁定一系列可能的行以阻止幻影。

11.3.6 指定特定的锁类型——优化器提示

有时候可能希望在查询中或在整个事务中更好地控制锁行为。这可以通过使用所谓的优化器提示(optimizer hints)来实现。

优化器提示是一种明确告诉 SQL Server 将锁升级到指定级别的方法。它们位于要操作的表的名称后面(在 SQL 语句中), 并按照表 11-4 的方式被指定:

表 11-4

提 示	描 述
SERIALIZABLE/HOLDLOCK	一旦通过语句在事务中建立了锁, 在事务结束(通过 ROLLBACK 或 COMMIT)之后才会释放这个锁。在执行插入操作时, 如果要插入的记录同建立该锁的查询中的 WHERE 子句中的条件相匹配(不是幻影), 那么插入操作也被阻止。这是最高的隔离级别, 并且绝对保证数据的一致性
READUNCOMMITTED/NOLOCK	不获取锁(甚至不使用共享锁)也不支持其他的锁。这是一个非常快速的选项, 可能产生脏读, 也可能产生其他问题
READCOMMITTED	这是默认选项。支持所有的锁, 但是处理获得的锁的方式取决于数据库的 READ_COMMITTED_SNAPSHOT 选项。如果打开这个选项, 那么 READCOMMITTED 不会获取锁, 作为替代方案, 它会通过使用一个行版本模式来确认是否发生了冲突。在实际情况下, 这样做的效果不错, 只有当需要提供向后兼容性, 而且需要较好的性能时, 才应该考虑使用 READCOMMITTED 方法

(续表)

提 示	描 述
READCOMMITTEDLOCK	它在这里是令人讨厌的。在绝大多数情况下它同 READCOMMITTED 非常相似(实际上, 它的功能同以前版本的 SQL Server 中的 READCOMMITTED 完全一样)。它支持所有的锁, 一旦不再需要相关的对象, 就释放所持有的锁。执行的操作同 READ COMMITTED 隔离级别类似
REPEATABLEREAD	一旦在事务中通过语句建立了锁, 直到事务结束才会释放这个锁(通过 ROLLBACK 或 COMMIT)。不过, 可以插入新的数据
READPAST	不是等待锁被释放, 而是跳过所有被锁定的行。这种跳过的行为仅限于行锁(仍然会等待页、区段和表上的锁), 并且只能用在 SELECT 语句上
NOWAIT	立刻导致查询失败, 而不是等待(如果检测到任何锁)
ROWLOCK	即使优化器选择其他较小的粒度锁定策略, 它也会强行将锁的初始级别设置为行级。如果锁的数量达到系统的锁阈值, 它不会阻止锁升级到更小的粒度级别
PAGLOCK	使用页级锁, 不理睬优化器所做出的其他选择。在两种方式中它比较有用。有时候你知道对于资源保存来说, 页锁比行锁更加合适。另一些时候, 优化器会选择表锁, 而你希望将争用最小化
TABLOCK	强制使用完整的表锁, 不理睬锁管理器使用了什么。事实上这样可以加快对已知表的扫描, 但是如果其他用户希望修改表中的数据, 也会导致比较大的争用问题
TABLOCKX	同 TABLOCK 相似, 不过要创建排他锁——根据 TRANSACTION ISOLATION LEVEL 的设置方式, 在语句或事务期间, 将其他所有的用户锁定在表外
UPDLOCK	使用更新锁来替代共享锁。在同死锁的斗争中, 并没有充分发挥这个工具的功能。因为它仍然允许其他的用户使用共享锁, 不过会确保不会进行数据修改(其他更新锁), 直至语句或事务终止(大概在继续运行并更新行之后)
XLOCK	虽然植根于 TABLOCKX, 这是它第一次出现在 SQL Server 2000 中。这样做的好处是你不用理会自己已经选中(或没有选中)的锁粒度, 可以指定一个排他锁

绝大多数的这些选项都有自己的特定适用场合, 不过在使用它们之前, 确保自己已经理解了本章后面会介绍的隔离级别的概念。

使用它们的语法相当简单——只需要在表名后加上它, 或是在所使用的别名后加上它:

```
....
FROM <table name> [AS <alias>] [[WITH] (<hint>)]
```

这里在几个例子中都用到了它们, 所有的用法都是合法的, 都会在 SalesOrderHeader 表上强行应用表锁(而不是使用键或行锁):

```
SELECT * FROM Sales.SalesOrderHeader AS ord WITH (TABLOCKX)
```

```
SELECT * FROM Sales.SalesOrderHeader AS ord (TABLOCKX)
```

```
SELECT * FROM Sales.SalesOrderHeader WITH (TABLOCKX)
```

```
SELECT * FROM Sales.SalesOrderHeader (TABLOCKX)
```

这里从多表的角度观察这个问题。下面的查询完成的操作与之前锁定方面的操作一样。它们会在 SalesOrderHeader 表上强加一个排他的表锁。不过值得注意的是, 它们没有在 SalesOrderDetail

表上放置任何的特殊锁。SQL Server 锁管理器仍然可以完全控制这个表。

```

SELECT *
FROM Sales.SalesOrderHeader AS ord WITH (TABLOCKX)
JOIN Sales.SalesOrderDetail AS od
    ON ord.SalesOrderID = od.SalesOrderID;

SELECT *
FROM Sales.SalesOrderHeader AS ord (TABLOCKX)
JOIN Sales.SalesOrderDetail AS od
    ON ord.SalesOrderID = od.SalesOrderID;

SELECT *
FROM Sales.SalesOrderHeader WITH (TABLOCKX);
JOIN Sales.SalesOrderDetail AS od
    ON Sales.SalesOrderHeader.SalesOrderID = od.SalesOrderID;

SELECT *
FROM Sales.SalesOrderHeader (TABLOCKX)
JOIN Sales.SalesOrderDetail AS od
    ON Sales.SalesOrderHeader.SalesOrderID = od.SalesOrderID;

```

也可以在这里完成某些完全不同的工作，并在 salesOrderDetail 表上放置一个完全不同的提示——这一切都由你来决定。

1. 使用 Management Studio 来确定锁

使用 Management Studio 也许是观察锁的最好方法。Management Studio 通过使用活动监视器 (Activity Monitor)，采用两种不同的排序方式(基于进程 ID 或基于对象)显示锁。

要使用 Management Studio 显示锁，需要先浏览到服务器上，然后右击选中“活动监视器”。这样做会打开一个新窗口，如图 11-3 所示(这里已经扩展了进程框架)。

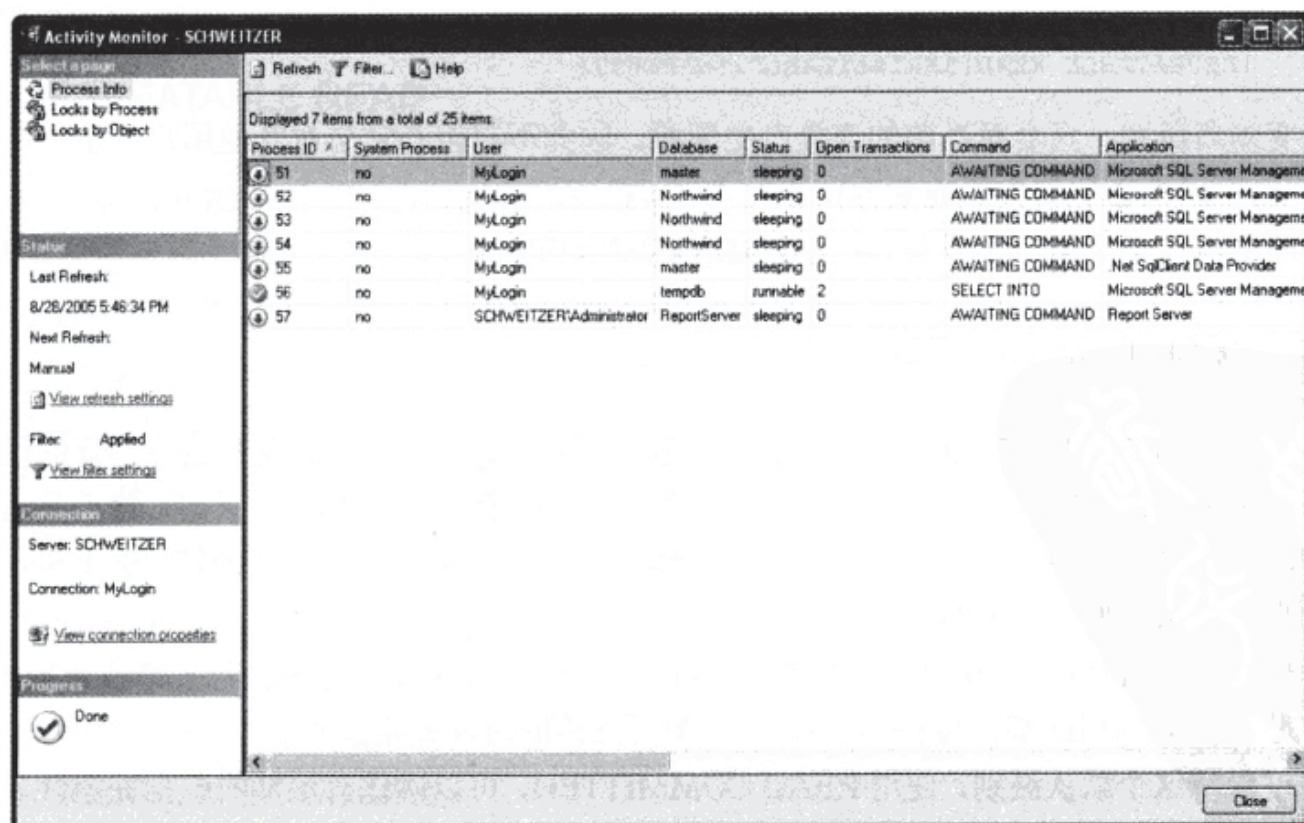


图 11-3

只需要展开你感兴趣的节点(或者是进程 ID, 或对象), 就可以看到各种各样的锁。

提示:

在右边窗口中双击相应的锁, 也许会显示 Management Studio 中最酷的功能。这时会弹出一个对话框, 告诉你该进程 ID 运行的上一条语句。当希望对死锁状态进行排错时, 这是非常便利的。

11.4 设置隔离级别

我们已经看到, 通过使用不同的锁策略, 可以防止发生几种不同的问题。我们还看到了有哪些类型的锁可供使用, 以及这些锁影响资源的可用性的方式。现在是时候深入了解一下进程管理块是如何协同工作以确保全面的数据完整性, 并且确保你能够得到所期待的结果。

关于事务和锁之间的关系, 首先要知道的是在它们之间的关系是盘根错节密不可分的。在默认情况下, 一旦创建了任何同数据修改有关的锁, 这个锁将在整个事务的持续期内被持有。如果事务持续的时间很长, 就意味着这个锁会在这段很长的时间内一直防止其他进程访问对象。很难说这不是个问题。

不过, 这只是默认情况。事实上, 可以设置 5 种不同类型的隔离级别:

- READ COMMITTED(默认值);
- READ UNCOMMITTED;
- REPEATABLE READ;
- SERIALIZABLE;
- SNAPSHOT

在它们之间进行切换的语法是很直观的:

```
SET TRANSACTION ISOLATION LEVEL <READ COMMITTED|READ UNCOMMITTED  
|REPEATABLE READ|SERIALIZABLE|SNAPSHOT>
```

改变隔离级别, 只会对当前的连接产生影响。因此不用担心它会对其他用户产生不良的影响(同样, 也不用担心他们会影响到你)。

下面更深入地了解一下默认情况(READ COMMITTED)。

11.4.1 READ COMMITTED

有了 READ COMMITTED, 只要执行完创建共享锁的语句, 就会自动释放你创建的任何共享锁。也就是说, 如果启动了一个事务, 运行几条语句, 接着运行 SELECT 语句, 然后又运行更多的语句, 一旦完成了 SELECT 语句, 就会立刻释放与 SELECT 语句相关联的锁。SQL Server 不会等待到事务结束的时候再释放这些锁。

操作查询(UPDATE、DELETE 和 INSERT)稍有不同。如果在事务中执行了修改数据的查询, 那么在整个事务的期间都会保持那些锁(以便需要回滚的时候使用)。

通过保持这个默认级别, 使用 READ COMMITTED, 可以确保有足够的数据库完整性以防止脏读。不过仍然可能会出现不可重复读和幻影。

11.4.2 READ UNCOMMITTED

在所有的隔离级别中, READ UNCOMMITTED 是最危险的, 但从速度的角度来说它也是性能最好的。

将隔离级别设置为 READ UNCOMMITTED 时, SQL Server 不设置任何锁, 也不使用任何锁。对于这种隔离级别, 可能会经历我们在本章前面讨论过的各种各样的问题(特别是脏读)。

为什么会冒遇到脏读的风险呢? 当我在 Usenet 上浏览新闻组的时候, 我看到这是常出现的问题。虽然它让很多人感到惊讶, 但使用这种隔离级别的理由的确很充分, 几乎总是在报表中使用它。

在一个 OLTP 环境中, 锁既是你的保护, 也是你的敌人。它们防止数据出现完整性问题, 但是也经常防止或阻止你得到自己需要的数据。最常见的情况是管理层希望定期运行报表, 但是因为管理员的报表持有锁, 数据输入人员输入数据的操作却经常被阻止或延时。

使用 READ UNCOMMITTED, 经常会解决这种问题——至少对于数字精度要求不高的报表而言, 是这样的。例如, 假设销售经理希望知道到今天为止的销售额。实际上, 他确实很麻烦而且一天问了好几次同样的问题(以重复运行报表的形式)。

如果运行报表要花费较长的时间, 那么在运行过程中因为锁的问题, 很有可能会破坏其他用户的生产率。这样对报表有什么好处呢? 这只是一个近似的报表。实际的数据可能根本没有意义。经理所看到的只是一个近似数字。

通过使用 READ UNCOMMITTED 隔离级别, 我们不设置任何锁, 这样也就不会锁定任何其他的事务。这里的数字可能不太准确(因为有出现脏读的风险), 但是这里根本不需要特别精确的数字, 只要知道这些数字逼近真实值, 脏读被回滚的机会很小。

可以通过在查询中添加 NOLOCK 优化器提示, 获得同 READ UNCOMMITTED 一样的效果。设置该隔离级别的好处在于, 不需要在查询的每个表上使用提示, 也不需要多个查询中使用它。使用 NOLOCK 优化器提示的好处在于, 不需要记住(使用 READ UNCOMMITTED)将隔离级别设置为连接的默认值。

11.4.3 REPEATABLE READ

REPEATABLE READ 可以略微提升隔离级别, 并提供双重保护的额外级别, 不但可以防止脏读(默认情况下已经做到了), 而且还可以防止不可重复读。

防止不可重复读是最重要的, 但是持有的哪怕是共享锁, 也会在事务结束之前阻止用户对对象的访问, 因此会损害生产率。就个人而言, 我宁愿不使用这个选项而使用其他的数据完整性选项(例如设置 CHECK 约束和错误处理), 不过要知道它还是一个有用的选项。

同 REPEATABLE READ 隔离级别等价的优化器提示是 REPEATABLE READ(它们是相同的, 只是没有空格)。

11.4.4 SERIALIZABLE

SERIALIZABLE 是隔离级别的某种堡垒。它可以防止任何形式的并发问题。除了不能防止丢失更新, 即使是幻影也可以被防止。

当隔离级别被设置为 SERIALIZABLE 时, 应用到事务所使用的表上的任何 UPDATE、DELETE 或 INSERT, 都不会满足在那个事务的任何语句中的 WHERE 子句。在本质上, 用户必

须等到事务结束之后，才可以在事务关注的记录上执行一些操作。

在查询中使用 `SERIALIZABLE` 或 `HOLDLOCK` 优化器提示，也可以模拟 `SERIALIZABLE` 隔离级别。同样，与 `READ UNCOMMITTED` 和 `NOLOCK` 之间的争论理由非常相似，使用这个隔离级别就不用在代码中每次使用相应的优化器提示，而使用优化器提示就不用每次都记住将隔离级别设置回默认值。

提示：

表面上，`SERIALIZABLE` 隔离级别好像能帮助你完成所有的事情。实际上，它为数据库提供了最高级别的所谓一致性——这就是说，更新过程适用于多个用户，就好像所有的用户每次只执行一个事务(操作过程是连续的)。

但是，就像人生中的大多数事情一样，这样做也是有代价的。在特定的场合下，一致性和并发性会成为完全对立的两极。令事务 `SERIALIZABLE` 可以防止其他用户获取所需要的对象；这等同于降低了并发性。反过来也是这样：增加并发性(例如使用 `REPEATABLE READ`)会降低数据库的一致性。

对此我个人建议除非有特别的理由，否则请保持默认设置(`READ COMMITTED`)。

11.4.5 SNAPSHOT

注意：

注意在默认情况下，不可以使用 `SNAPSHOT` 事务隔离级别。要使用它，必须利用 `ALTER DATABASE` 命令为自己的数据库启用 `ALLOW_SNAPSHOT_ISOLATION` 选项。

这是新加入 SQL Server 2005 的选项，而且还没有被完全公开(而且如果你问我的话，我会说它们还没有得到很好的记录!)。`SNAPSHOT` 利用了所谓的“行版本”。被阻塞在某个特定记录外的那些事务可以读取相应记录(处于最近一次已知的正确状态)，即记录处于阻塞事务开始修改行之前的状态。

`SNAPSHOT` 的好坏参半。一方面因为当时允许读取事务继续，而且它的值当时为正确的技术值(至少说数据已经实际提交)，并发性会增加。不过它也有负面影响，也就是允许事务继续处理那些很有可能很快会变得不准确的数据。

你会用哪个呢？正如你想象得到的，我的答案可能是“要视情况而定”。更安全的答案是坚持 `READ COMMITTED` 的默认值。不过有时候，不需要安全性，所以好的并发性应该是个很好的选择。

注意：

使用 `ALTER DATABASE` 命令启用 `READ_COMMITTED_SNAPSHOT` 数据库选项，可以将 `READ COMMITTED` 默认隔离级别切换到使用行版本的另一个版本上，它和 `SNAPSHOT` 一样有效。不过要确保在修改之前，自己充分理解了两种 `READ COMMITTED` 实现之间的区别。

11.5 处理死锁(也称作“A 1205”)

好。现在你已经看过锁，也看过事务了。既然对两者都已经有所了解，下面可以开始讨论这

个比较棘手的问题了，就是处理死锁。

正如已经提到过的，死锁本身并不是一种锁类型，而是一种由其他锁形成的自相矛盾的状态。无论你是否愿意，你都会经常遇到这个问题(特别是当你只是一个新手的时候)，得到编号为 1205 的错误。这个错误是非常典型的，所以你会听到很多数据库开发人员在报告错误的时候只是简单的说出这个号码。

当一个锁由于第二个锁持有资源，而无法完成操作以便释放(反之亦然)时，就会导致死锁。发生死锁的时候，某些人会从中获胜，因而 SQL Server 会选择死锁的牺牲品。死锁牺牲品的事务被回滚，并被告知出现了 1205 号错误。另一事务则正常继续(事实上它根本不会觉察到这里存在错误，只是觉得执行的时间增加了)。

11.5.1 SQL Server 判断死锁的方式

每隔 5 秒钟，SQL Server 都会检查所有的当前事务，查看有哪些锁正在等待而尚未被允许。在它进行检查的时候，实际上会记录现有的请求。然后它会再次重新检查所有的开锁请求状态，如果之前的某个请求仍然没有被许可，它就会递归检查所有打开的事务，查看是否存在锁请求的循环链。如果找到了这样的循环链，就会选择一个或多个死锁的牺牲品。

11.5.2 如何选择死锁牺牲品

默认情况下，死锁牺牲品是基于有关事务的“成本”进行选择的。回滚成本最少的事务会被选中(换句话说，SQL Server 只需要完成很少的工作就可以撤消它)。可以通过使用 SQL Server 中的 DEADLOCK_PRIORITY SET 选项在一定程度上覆盖此设置。不过，通常不应该这么做，这也超出了本书的范围(我认为这是管理员应该掌握的内容，而不是开发人员应该掌握的)。

11.5.3 避免死锁

在一个复杂系统中，不可能百分之百避免死锁，不过通过遵循一些特定的准则，几乎可以消灭死锁——也就是说，令系统中出现死锁的几率变得很少。

要减少或消灭死锁，请遵循下面的简单(好吧，通常是比较简单的)准则：

- 按照同样的顺序使用对象；
- 使事务尽可能的短，并在一个批次中处理；
- 使用必要的最低事务隔离级别；
- 在同一个事务中，不允许无限制地中断(用户的交互，批分离)；
- 在可控制的环境中，使用绑定连接。

几乎每次在我遇到死锁问题的时候，都至少违背了这其中的一个(通常更多)规则。下面逐个介绍这些准则。

1. 按照同样的顺序使用对象

我所认为这是一种最常见的最基本规则。使用这个规则最大的好处是几乎不需要花费任何代价；它更多的是思考的方法。你可以在设计过程中早做决定，确定访问数据库对象的方式，包括顺序…让它成为你为项目编写每一个查询、过程或触发器的习惯。

花点时间考虑一下——如果这里的问题是在两个连接中，每个连接都想要其他连接所要的东西，那就意味着在这场游戏中再做处理已经太迟了。下面看一个简单的例子。

假设有两个表：Suppliers 和 Products。现在假设有两个进程都使用了这两个表。进程 1 接受库存输入，用手头上的新产品数量来更新 Products，然后再用已购买的产品数量来更新 Suppliers 表。进程 2 记录销售量，它在 Suppliers 表中更新销售的产品总数，然后在 Products 表中减少库存量。

如果同时运行这两个过程，就会遇到麻烦。进程 1 会在 Products 表上持有排他锁。进程 2 则在 Suppliers 表上持有排他锁。进程 1 试图在 Suppliers 表上拥有锁，但是被强制等待进程 2 释放其现有的锁。与此同时，进程 2 试图在 Products 表上创建锁，但是必须等待进程 1 释放其现有的锁。现在就出现了自相矛盾的情况：两个进程都在等待对方回应。SQL Server 不得不选择一个死锁的牺牲品。

现在，重组一下这个场景。进程 2 被改变为首先在 Products 表中减少库存数量，然后在 Suppliers 表中更新销售的产品总数。它在功能上等价于第一种方式描述的过程，采用这种方式不会多花费任何开销。而且效果也非常好，它不会出现死锁(至少在这两个进程之间不会出现死锁)！下面看一看到底发生了什么。

同时运行这两个进程的时候，进程 1 会在 Products 表中持有一个排他锁(到目前为止，一切都相同)。随后进程 2 也试图在 Products 表上持有锁，但是被强制等待进程 1 完成操作(注意这时还没有对 Supplier 做任何操作)。进程 1 完成了对 Products 表的操作，但是因为还没有完成事务，所以它并不会释放锁。进程 2 仍然在等待进程 1 释放在 Products 表上的锁。进程 1 现在移动到 Suppliers 表上，并在其上持有锁。进程 2 继续等待进程 1 释放在 Products 表上的锁。进程 1 根据需要，完成并提交或回滚事务，释放所有的锁。现在进程 2 可以在 Products 表上持有锁，并完成剩余的事务，而没有更多的事了。

只需要交换两个查询的运行顺序，就可以杜绝潜在的死锁问题。在任何可能的情况下，都应该尽可能保持相同的顺序，这样才可以减少遇到死锁的机会。

2. 令事务尽可能的短

这是另一个基本准则。同样，这应该成为一种本能——一种你不需要真正思考，但确实要这么做的事情。

它也不会让你花费任何代价。只将自己需要的内容放入事务中，而将其他不需要的东西放在事务之外。就是这么简单。为什么这样做会有效果呢？这不像火箭科学那样复杂。事务打开的时间越长，它接触到的东西就越多(在事务中)，在其他过程中需要正被使用的对象的可能性就越高(减少并发)。如果要保持事务的短小性，可以减少可能潜在的导致死锁的对象数量，减少对被锁定对象的占用时间，就是这么简单。

将事务保持在一个批内，可以在事务进行过程中减少网络流量，减少在完成事务和释放锁时可能导致的延迟。

3. 尽可能使用最低的事务隔离级别

这个规则不是最基本的规则，要求认真考虑。如果经常忽略它，也不用感觉惊讶。想一下 ROB 定理：要想的通常不是被想的。这里应该有所不同：多考虑一下。

可以使用几种不同的事务隔离级别。默认的是 READ COMMITTED。与使用较高的隔离级别

相比,使用较低的隔离级别会让持有共享锁的时间更短,因此也减少了锁的争用。

4. 不允许无限制的事务

这也许是所有推荐内容中最常见的,不过根据过去的实践,这也是最常被违背的。

在操作完成之前,获取锁并持有它是一种用来防止丢失更新(这是大型机的时代,伙计!)的方法。这里不能告诉你这里会有多少问题(你会说这种废话吗!)

想象一下这个真实的例子:在服务部门中,有些人喜欢使用更新(排他锁)屏幕而不是显示(共享锁)屏幕来查看数据。他说“毕竟,如果要看到要修改一些东西的话,我就可以用这种方式去编辑”。在他查看一个工作订单的时候,他的伙伴向他打招呼,问他是否准备去吃午饭。他回答“好的”,这个服务职员就会停止工作,去吃一顿耗时很长的午餐(1~2 个小时)。在这个职员吃午餐的整个时间段内,对这些记录感兴趣的所有人都被锁定在记录之外。

等一会儿情况会变得更糟。在使用大型机的年代,你会经常看到排队的概念(它实际上可能是相当有效的)。现在某人针对这项工作订单提交了一个打印任务(它会排队)。它会停留在队列中,等待释放记录锁。因为这是一个队列环境,在公司中每个针对工作订单的打印任务都会堆积在第一个打印任务(在它被清除前,会一直等待这个职员吃完午餐)之后。

这是一个非常极端的例子,不过也是一个我经常看到的真实场景。我希望这个例子可以清楚地说明这个问题。在开始某种类型的无限制过程时,不要创建一直被打开的锁。通常这是指用户的交互性(就像这里的午餐爱好者一样),不过也可能是任何无限制的等待过程。

5. 使用绑定连接

嗯,我不得不讨论是否要包含该项,因为它是某种装满了小虫子的盒子。一旦打开了它,就不能再将它们放回去。我只能说它很少被用到,而且只是为有勇气的人准备的。

但是这并不是说它没有用,只是它会使事情迅速变得十分复杂,所以需要好好管理它。对我个人来说,通常会有更好的解决方法。

这样做会带来一个问题,什么是绑定连接。绑定连接是一种已经关联的连接并允许共享其相同的锁集合。这意味着两个事务可以协同操作,而不用担心死锁或彼此锁定。反过来说,需要自己处理大多数并发问题。锁不能再保持安全了。

在 99.9%的情况下,我都不会这样尝试。这里会忘掉它是一个存在的选项。如果你还是坚持要使用它,请记住你必须处理非常复杂的连接之间的关系,如果要维持系统中的数据完整性,就必须深入管理连接活动。

11.6 小结

事务和锁是 SQL Server 工作的两块基石,了解它们可以大大增强我们在 SQL Server 中开发解决方案的能力。

使用事务可以确保所需要的每件事都作为一个整体而发生,或者都不发生。SQL Server 使用锁最大可能地避开并发陷阱(你不能完全避开它们,但是通过一些——事实上是很多——计划可以接近这个目标)。通过这两者的协同工作,就可以通过数据库行业的所谓 ACID 测试。如果一个事务是 ACID 的,那么它拥有:

- **原子性**: 要么完成事务中的全部内容, 要么什么都不完成;
- **一致性**: 已经符合所有的约束以及其他的数据完整性规则, 所有的相关对象(数据页、索引页)已经被完整更新;
- **分离性**: 每个事务都与其他事务独立。一个事务中的行为, 不会被另外事务中的行为所干扰;
- **持久性**: 在完成事务之后, 它对系统中相应位置的影响是永久的。数据是“安全的”, 即使是一些譬如断电或其他的非磁盘系统故障之类的环境下, 也不会导致数据只被写入一半。

简言之, 使用事务和锁可以减少死锁, 保证数据完整性, 提高系统的整体效率。

下一章中会看到触发器。实际上, 可以看到在多种可能的触发器使用中, 事务和锁的概念将会在触发器中占据中心位置。

第12章

触 发 器

我常常问自己“是否应该使用触发器？”在 SQL 中，答案是“要视情况而定”。SQL Server 中很少有完全对立的事物，触发器实际上就是这样的一个灰色地带。

在使用触发器前要明白自己正在做什么——这对自己数据库的健康和性能非常重要。值得庆幸的是我们正要学习这部分内容。

像本书中涉及的大多数核心主题一样(那些至关重要以至于不能一笔带过的内容除外)，这里假设读者已经了解了基本概念，因此对这部分内容的介绍会比较简短。不过，即使你已经变成一个相对高级的 SQL Server 用户，也可能从未触碰过这类课题。即，初学者在安装某些产品时可能需要触发器，但是“专业人员”却从未触碰过它们(SQL 就是这样...)。如果你阅读过了我的《SQL Server 2008 编程入门经典》(由清华大学出版社引进并出版)一书，那么你一定注意到两者之间存在某些重叠(不过你会发现这里涉及的内容更有深度)。如果你已经读过那本书的话，那么请直接跳到 INSTEAD OF 触发器小节。

本章会尝试观察触发器的方方面面——从黑的一面一直讲到白的一面，并兼顾介于黑白之间的许多内容。本章将处理的主要问题包括：

- 什么是触发器？(快速简短地介绍)
- 为了更灵活的引用完整性而使用触发器；
- 使用触发器创建灵活的数据完整性规则。
- 使用 INSTEAD OF 触发器创建更灵活的可更新视图
- 触发器的其他常见用途
- 控制触发器的激活顺序
- 性能考虑

学过本章之后，你应当认识到决定在什么时候和什么地方不使用触发器是非常复杂的。而且你也会大概认识到触发器有多么的灵活以及功能是何等的强大。

最重要的是，如果这里讲得清楚的话，你不会对触发器持一种极端的态度(很多我遇到的 SQL Server 用户都是这样的)，将其曲解为有害的而且是绝对不能使用的。你也不会陷入另一个极端：触发器可以解决一切问题。这个问题的正确答案应该是触发器可以为你做很多事情，但是也会带来很多问题。关键是只在适合使用触发器的时候才使用它们。

下面列出了触发器的常见用途：

- 强制引用完整性：虽然我建议尽可能使用声明性引用完整性(DRI)，但是有许多事情 DRI 是无法完成的(例如，跨数据库或者甚至跨服务器的引用完整性、许多复杂的关系类型等)。虽然使用触发器实现引用完整性只是一个非常特殊的例子，但是这也不失是一种方法。
- 创建审核跟踪，就是说不但记录跟踪最近的数据，而且还要跟踪每一条记录实际的变更历史。
- 功能类似于 CHECK 约束，但是可以跨表、数据库或者服务器工作。
- 在用户的操作语句处，替换你自己的语句(通常用于在复杂视图中启用插入)。

此外，还有一种新的、或许应用场合极少的(正如我所说的，它们是新加入的，因此肯定只有时间才能证明)触发器用来监视对表结构的修改——DDL 触发器。

这些只是其中的一部分。闲话少说，下面来看看究竟什么是触发器？

12.1 触发器的含义

触发器是一种响应特定事件的特殊类型的存储过程。触发器有两种：数据定义语言(DDL)触发器和数据操作语言(DML)触发器。

DDL 触发器会在人们以某种方式(CREATE、ALTER、DROP 及类似的语句)对数据库结构进行修改的时候作出响应而激活。DDL 触发器是 SQL Server 2005 中新增的功能，而且对于一些(极高安全性的安装)安装系统非常重要，但是它们的使用范围十分有限。一般来说，只有在迫切需要对数据库结构的修改(历史)进行审核时，才要使用 DDL 触发器。这部分内容将留到最后讲述。

DML 触发器是一些附加在特定的表或视图上的代码。触发器与存储过程不同，存储过程中的代码必须被明确地调用；而对于触发器而言，只要在表中发生了触发器附加到的事件，就会自动运行触发器中的代码。实际上，不能显示调用触发器——调用触发器的唯一方法是在触发器被指派到的表上执行必需的操作。

提示：

除了不能显式调用触发器之外，参数和返回代码也是存储过程特有的而触发器不具有的能力。

虽然触发器不接收参数，但的确有一种机制可以确定它们将作用在哪些记录上(本章随后会研究这方面的内容)。而且，虽然可以使用关键字 RETURN，但不能返回具体的返回代码(由于没有显式调用触发器，你要将代码返回给谁呢？)。

触发器可以附加到哪些事件呢？答案是在 SQL 中使用的 3 种“操作”查询类型。因此，最后会得到基于插入、更新和(或)删除的触发器(可以混合搭配触发器可以附加到的事件)。

提示：

值得注意的是，有时触发器是不会被触发的——即使正在执行的操作看起来似乎属于前面的范畴。问题是正在执行的操作是否被记入活动日志。例如，DELETE 语句是标准的、可以触发任何 delete 触发器的被记录活动，但是，具有删除行效果的 TRUNCATE TABLE 只是释放了表使用的空间。不记录任何的单独删除，因此也不会触发触发器。

除了必须附加在一个类似于索引的表上,用于创建触发器的语法与其他所有的 CREATE 语法非常相似,触发器不能建立在触发器上。

下面看一下语法:

```
CREATE TRIGGER <trigger name>
  ON [<schema name>.<table or view name>]
  [WITH ENCRYPTION | EXECUTE AS <CALLER | SELF | <user> >]
  {{{FOR|AFTER} <[DELETE] [,] [INSERT] [,] [UPDATE]>} | INSTEAD OF}
  [WITH APPEND]
  [NOT FOR REPLICATION]
AS
  <sql statements> | EXTERNAL NAME <assembly method specifier>
```

可以看出,这里依然有我们非常熟悉的 CREATE<object type><object name>,它同样存在于许多其他对象的执行过程中——只不过这里添加了 ON 子句,表示将要附加触发器的表,以及触发器的触发时间和条件。

12.1.1 ON

这一部分只是用来说明要基于什么对象创建触发器。请记住,如果触发器的类型是 AFTER(使用 FOR 或 AFTER 声明的触发器),则 ON 子句的对象必须是表——视图不支持 AFTER 触发器。

12.1.2 WITH ENCRYPTION

这个选项在触发器中的功能与其在视图和存储过程中的功能一样。如果添加了这个选项,就能确定没有人可以查看你的代码(甚至你自己都不行!)。如果将要开发商业销售软件,或者如果非常关注安全性并且不希望用户看到你正在修改或访问的数据,那么这个选项会有很大的帮助。显然,你应该在其他地方备份创建触发器所需的代码,以便于下次重新创建触发器的时候使用。

与视图和存储过程一样,在使用 WITH ENCRYPTION 选项时要记住,每次修改触发器的时候,必须重新应用这个选项。如果使用了 ALTER TRIGGER 语句,并且语句中没有包含 WITH ENCRYPTION 选项,那么触发器将不再是被加密的。

12.1.3 FOR|AFTER 与 INSTEAD OF 子句

除了要决定触发触发器的查询类型(INSERT、UPDATE 和/或 DELETE)之外,还必须选择触发触发器的时间。虽然 FOR(此外,也可以选择使用关键字 AFTER)触发器已经出现很久而且大家通常都会考虑使用 FOR 触发器,不过也可以运行 INSTEAD OF 触发器。选择其中的一种触发器会影响到是在修改数据之前进入触发器,还是在数据修改之后进入。无论是哪种情况,都会在真正将修改提交给数据库之前进入触发器。

或许已经糊涂了吧?下面换一种方式,用图表显示触发每个选项的位置(如图 12-1 所示)。

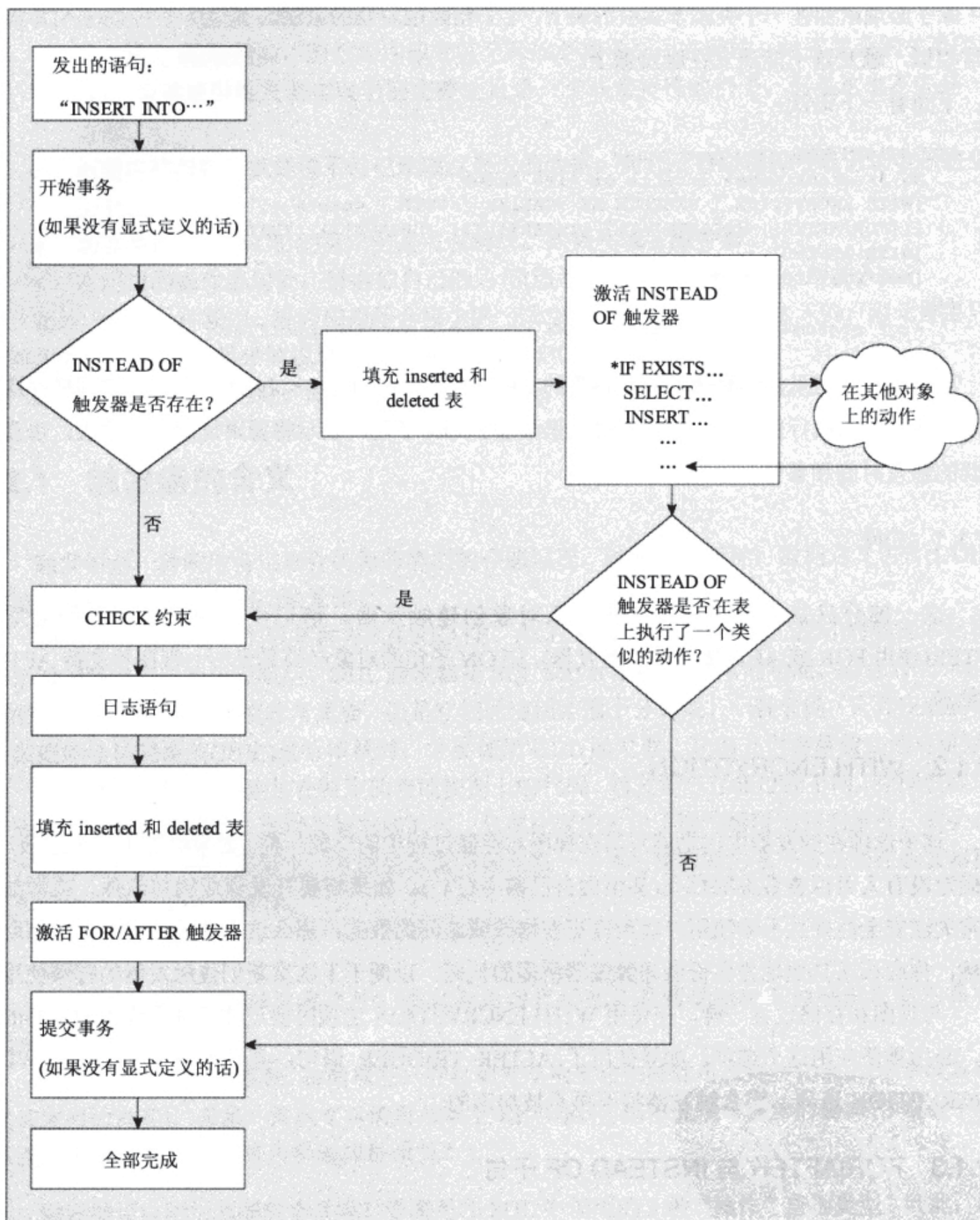


图 12-1

这里要注意的是，不论选择了哪个选项，SQL Server 会将两个工作表放在一起——一个用来保存插入记录的副本(称之为 INSERTED)，一个用来保存所有删除记录的副本(称为 DELETED)。稍后将更详细地了解这两个工作表的使用方法。目前要知道的是，使用 INSTEAD OF 时，会在检查约束之前创建这些工作表，而使用 FOR 的时候，会在约束检查完毕后创建这些表。

实际上，INSTEAD OF 触发器的关键是：可以在用户请求的任何地方运行自己的代码。这意

味着可以处理掉视图中不明确的插入问题(回忆第8章中在存在 JOIN 时,在视图中的插入问题)。这还意味着我们甚至可以在检查约束之前就采取行动清理违反约束的情况。

使用 FOR 和 ALTER 声明的触发器作用基本相同。它们与 INSTEAD OF 触发器之间最大的不同是,在检查过所有的约束之后才创建工作表。

AFTER(或者 FOR)子句指出了触发触发器的操作类型。可以在出现 INSERT、UPDATE、DELETE 或三者的任意组合时触发触发器。例如,下面给出了可能使用的 FOR 子句:

```
AFTER INSERT, DELETE
```

或:

```
AFTER UPDATE, INSERT
```

或:

```
AFTER DELETE
```

正如在 ON 子句那一节所说过的,使用 AFTER 或 FOR 子句声明的触发器只能附加在表上——不允许附加在视图上(请参考 INSTEAD OF 触发器)。

提示:

值得注意的是,不同于本书之前的版本,实际上我在 AFTER 和 FOR 之间的确作出了某种选择。虽然二者都是可用的,而且也没有迹象表明会丢弃任何一方,但是, AFTER 子句是“标准”的方法,因而更可能得到其他数据库厂商支持。

1. INSERT 触发器

只要有人向表中插入新行,都会执行所有标记为 FOR INSERT 的触发器中的代码。对此,SQL Server 会为每一个新插入的行创建一个副本,并将其插入到一个只存在于触发器域内的特殊表中。这个特殊的表被称为 INSERTED。在学习本章过程中,我们会看到很多与它有关的内容。这里有一点非常重要的内容要理解,即只有在存在触发器的时候,INSERTED 表才存在。在触发器开始前或触发器结束后,认为这个表都是不存在的。

2. DELETE 触发器

除了 INSERTED 表将为空之外,DELETE 触发器与 INSERT 触发器大致相同(毕竟,这里是要删除而非插入,因此,INSERTED 表中没有记录)。这里会把删除的每一条记录的副本插入另一个名为 DELETED 的表中。DELETED 表与 INSERTED 表一样,只存在于触发器的生命周期内。

3. UPDATE 触发器

UPDATE 触发器与前面的触发器几乎相同,只有一点除外。一旦表中现有的记录发生变化,就会触发在触发器中被声明为 FOR UPDATE 的代码。这里变化的是,不存在名为 UPDATED 的表。SQL Server 会将每一行视为:已经删除了现有的记录,插入一个全新的记录。也许你可以由此猜想,声明为 FOR UPDATE 的触发器并非只包含一个表,而包含两个特殊的表,INSERTED 和 DELETED。当然,这两个表的行数完全相同。

12.1.4 WITH APPEND

老实说, WITH APPEND 确实有些奇怪, 所以你可能不会用到它。尽管如此, 为了“以防万一”, 这里还是包含了这部分内容。WITH APPEND 仅仅适用于 6.5 兼容模式(可以使用 sp-dbcmtlevel 设置)。

SQL Server 6.5 以及之前的版本不允许单个表上有多个类型相同的触发器。例如, 如果已经声明了一个名为 trgCheck 的触发器, 在更新和插入时候保证数据的完整性, 这样就不能为级联更新创建单独的触发器了。一旦创建了一个更新(插入或删除)触发器, 就只能使用它了——不能为同类的操作创建另外的触发器。

这样做真是麻烦。它意味着必须把逻辑上不同的活动合并在一个触发器中。在大多数情况下, 尝试让两个完全不同的进程在一起工作是非常困难的。除此之外, 这样做也会使代码更难阅读。

SQL Server 7.0 问世之后, 规则发生了重大的改变。我们不必担心同一种类型的查询操作上有多少个触发器——只要愿意, 可以有多个这样的触发器。但是数据库运行在 6.5 兼容模式下就会遇到问题: 此时, 数据库工作的环境仍然规定某个表上只能有一个给定类型的触发器。

即使表上已经存在这种类型的触发器, WITH APPEND 通过明确告知 SQL Server 我们要加入一个新的这种类型的触发器就可以回避这个问题; 出现特定的触发操作(INSERT、UPDATE 或 DELETE)时, 两个触发器都会被触发。这是一种“一箭双雕”的方法。

只要正在运行的不是“时光倒流版”的 SQL Server(即 6.5 兼容模式), 就不需要这个选项。除非你有特殊的原因, 否则不建议你使用它。

注意:

此时, 运行 6.5 兼容版本就意味着要求 SQL Server 按照十几年前的方式运行, 而且其版本的兼容等级也已经落后了四代。如果代码非常重要, 以至于十几年过去之后还要运行它, 那么似乎更应该将其升级至最新支持版本上。

12.1.5 NOT FOR REPLICATION

添加该选项会略微改变何时激发触发器的规则。若使用这个选项, 就不会在执行与复制有关的任务修改表时触发触发器。通常, 在修改原始表的时候, 会触发触发器(进行一些常规的事务或级联操作等), 但再次完成这种操作是没有意义的。

12.1.6 AS

与存储过程中的 AS 作用完全一样, 这是问题的实质内容。AS 关键字告知 SQL Server 接下来将要启动你的代码。从这里开始, 进入触发器的脚本部分。

12.2 为数据完整性规则使用触发器

虽然触发器不应该是你的首选方案, 但是触发器也可以完成与 CHECK 约束或 DEFAULT 一样的功能。“我应该使用触发器还是使用 CHECK 约束?” 这个问题的答案十分明确: “视情况而

定。”如果 CHECK 能够完成任务，那么最好选择它。但是，有时 CHECK 约束无法完成工作，或者，有时 CHECK 方法固有的一些东西使其效果不如触发器理想。下面给出了应该使用触发器而不应该使用 CHECK 的情况：

- 业务规则需要引用不同表中的数据；
- 业务规则需要检查更新前后的差异；
- 需要自己定义错误消息。

这其实只是一些皮毛。由于触发器非常灵活，实际上，何时应该使用它们这个问题归纳起来就是：在需要完成一些特别的事情的时候使用触发器。为了给出一些指导性建议，这里给出了我过去所著书中的一个对比表，如表 12-1 所示：

表 12-1

范 围	过 程	约 束
约束	迅速	必须针对每个表重新定义
	可以引用其他列	不能引用其他表
	符合 ANSI 标准	不能同数据类型绑定
触发器	极其灵活	命令出现后发生
	可以引用其他的列和其他表	花费巨大
	甚至可以使用.NET 引用 SQLServer 外部的信息	

注意这些描述都不是非常详细。因为每种情况都会有所不同，所以这里提供的只是选项何时成功何时失败的一组指南。

提示：

可能有人已经注意到对之前的表的讲述没有包含之前版本中介绍过的规则和默认值选项。为什么这里不介绍这些内容呢？因为现在有若干版本已经开始反对规则和默认值(是默认对象，不是默认约束)了，所以我也在逐步强化自己的这种思想：它们只是起到后向兼容的作用。

12.2.1 处理源自其他表的要求

CHECK 约束表现不错(既迅速又有效)，不过它们并不能完成你希望的所有事情。或许，它们最大的缺点会在你必须跨表验证数据时显现出来。

下面看一看 AdventureWorks2008 中的 Products 和 SalesOrderDetail 表以及相关的 SpecialOfferProduct 表。它们之间的关系如图 12-2 所示。



图 12-2

在标准的声明性引用完整性(DRI)的约束之下,可以确定,除非 **Products** 表中有一个匹配的 **productID**(经由 **SpecialOfferProduct** 表中的链完成),否则不能向 **SalesOrderDetail** 表添加任何订单项。

库存部门已经开始抱怨客户支持人员保存的产品订单不连续。应该在这些订单进入系统之前就取消它们。

不能使用 **CHECK** 约束处理这种情况,因为可以了解不连续状态的地方(**Products** 表)与放置约束的地方(**SalesOrderdetail** 表)并不在同一个表中。不过也不用因此汗流浹背,你可以对库存部门说,“没问题!”只需要使用触发器就能解决:

```
USE AdventureWorks2008;
GO

CREATE TRIGGER OrderDetailNotDiscontinued
ON Sales.SalesOrderDetail
AFTER INSERT, UPDATE
AS
IF EXISTS
(
    SELECT 'True'
    FROM Inserted i
    JOIN Production.Product p
      ON i.ProductID = p.ProductID
    WHERE p.DiscontinuedDate IS NOT NULL
)
BEGIN
    RAISERROR('Order Item is discontinued. Transaction Failed.',16,1);
    ROLLBACK TRAN;
END
```

接下来检验它的效果。首先，至少要有一条记录在遇到触发器的时候会失效。这就是说，Products 表中必须有一个不连续项；不过问题是，目前没有这样的记录。

```
SELECT ProductID, Name
FROM Production.Product
WHERE DiscontinuedDate IS NOT NULL;
```

```
ProductID    Name
-----
```

```
(0 row(s) affected)
```

因此，这里选出一条记录并对它进行修改，以满足测试的目的：

```
UPDATE Production.Product
SET DiscontinuedDate = GETDATE()
WHERE ProductID = 680;
```

完成这些改动后，就可以查看触发器是否有效了。接下来添加一个违反约束的单项商品。这里会利用一个已经存在的 SalesOrderHeader，所以不必再费心建立完整的订单：

```
INSERT Sales.SalesOrderDetail
(
    SalesOrderID,
    OrderQty,
    ProductID,
    SpecialOfferID,
    UnitPrice,
    UnitPriceDiscount
)
VALUES
(
    43660,
    5,
    680,
    1,
    1431,
    0
);
```

这里会得到预期的拒绝效果：

```
Msg 50000, Level 16, State 1, Procedure OrderDetailNotDiscontinued, Line 14
Order Item is discontinued. Transaction Failed.
Msg 3609, Level 16, State 1, Line 1
The transaction ended in the trigger. The batch has been aborted.
```

记得前面说过，如果需要的话，也可以创建自定义的错误消息以供发出，而不是发出与 RAISERROR 命令一起使用的即席消息。

12.2.2 使用触发器检查更新的差异

有时候，比起对过去的值或现在的值究竟是多少，你可能对这些值究竟变化了多少更感兴趣。虽然这些信息虽然不能从任何的列或表中获得，不过可以使用触发器中的 Inserted 和 Deleted 表将其计算出来。

出于安全的原因写入记录就是这样的一个例子。例如。为了审核，需要跟踪所有的库存调整，而不管是谁发起了它(例如，不通过订单商品调整库存，可以直接在库存表上调整库存)。

为了实现这个目的，这里需要一张审核表，以便使用 Inserted 和 Deleted 表：

```
USE AdventureWorks2008;

CREATE TABLE Production.InventoryAudit
(
    TransactionID int IDENTITY PRIMARY KEY,
    ProductID int NOT NULL
        REFERENCES Production.Product(ProductID),
    NetAdjustment smallint NOT NULL,
    ModifiedDate datetime DEFAULT(CURRENT_TIMESTAMP)
);

GO

CREATE TRIGGER ProductAudit
    ON Production.ProductInventory
    FOR INSERT, UPDATE, DELETE
AS
INSERT INTO Production.InventoryAudit
(ProductID, NetAdjustment)
    SELECT COALESCE(i.ProductID, d.ProductID),
           ISNULL(i.Quantity, 0) - ISNULL(d.Quantity, 0) AS NetAdjustment
FROM Inserted i
FULL JOIN Deleted d
    ON i.ProductID = d.ProductID
    AND i.LocationID = d.LocationID
WHERE ISNULL(i.Quantity, 0) - ISNULL(d.Quantity, 0) != 0;
```

在对上述代码进行测试之前，先分析一下这里完成了哪些工作。首先添加一个审核表，接收基表的改动信息。然后创建一个触发器，对表进行任何的修改都会激活该触发器，并将下一步的改动写入新的审核表中。

下面通过运行来检验测试脚本：

```
PRINT 'The values before the change are:'
SELECT ProductID, LocationID, Quantity
FROM Production.ProductInventory
WHERE ProductID = 1
    AND LocationID = 50;

PRINT 'Now making the change'
UPDATE Production.ProductInventory
SET Quantity = Quantity + 7
WHERE ProductID = 1
    AND LocationID = 50;

UPDATE Production.ProductInventory
SET Quantity = Quantity - 7
WHERE ProductID = 1
    AND LocationID = 50;

PRINT 'The values after the change are:'
SELECT ProductID, LocationID, Quantity
FROM Production.ProductInventory
WHERE ProductID = 1
```

```
AND LocationID = 50;
```

```
SELECT * FROM Production.InventoryAudit;
```

可以利用更改前后的输出来确定是否正确写入审核记录:

The values before the change are:

ProductID	LocationID	Quantity
1	50	353

```
-----
1          50          353
```

(1 row(s) affected)

Now making the change

(1 row(s) affected)

(1 row(s) affected)

(1 row(s) affected)

(1 row(s) affected)

The values after the change are:

ProductID	LocationID	Quantity
1	50	353

```
-----
1          50          353
```

(1 row(s) affected)

TransactionID	ProductID	NetAdjustment	ModifiedDate
1	1	7	2008-12-15 22:29:11.900
2	1	-7	2008-12-15 22:29:11.900

```
-----
1          1          7          2008-12-15 22:29:11.900
2          1         -7          2008-12-15 22:29:11.900
```

(2 row(s) affected)

12.2.3 使用触发器实现自定义错误消息

在其他的一些例子中已经涉及到了这部分内容。不过要记住,触发器可以非常方便地控制传给用户或客户端应用程序的错误消息或编号。

例如,使用 CHECK 约束,只会得到标准的错误 547 以及对这个错误的一个描述得不够详细的说明。它通常不能帮助用户找到实际出错的地方——实际上,客户端应用程序常常不能得到足够的信息,以便替用户做出智能、有效的响应。

简言之,有时候你创建触发器是因为,虽然已经存在某种事物可以实现自己想要的数据完整性,但是它不能提供足够的处理信息。

12.3 触发器的其他常见用途

除了可以用来保持数据完整性之外,触发器还有许多其他的用途。触发器的用途实际上是无限多的,这里只列举几种常见的用途:

- 更新摘要信息;
- 向非规范化的表输入数据以用于报告;
- 设置条件标记。

12.3.1 更新摘要信息

有时, 为了便于报告或者加快检查条件的执行性能, 我们希望保留聚集信息。

以客户的信用额度和当前余额为例。额度是比较静态的事情, 可以方便地将其与客户的其它信息存储在一起。当前余额则完全不同。虽然, 总是可以运行查询得到这个客户所有订单的全部未付差额, 以计算出当前余额。但是, 我们来思考一下, 假设你为 Sears 工作, 更确切地说每年要进行数以百万计的交易。现在, 想一下如何对表中数百万条记录进行查询并进行整理, 同时, 为了运行查询, 你将要和许多其他交易竞争。如果恰好可以从某个地方获取总计的差额, 情况就会好很多——不过, 如何维护这些信息呢?

我们只能确定, 自己总是使用存储过程来添加和支付订单记录, 随后让存储过程更新客户的当前余额。但是, 这意味着必须确保可能影响客户余额的所有存储过程都包含更新代码。如果某个存储过程遗漏了更新代码, 那么麻烦就大了; 并且即便是在最理想的情况下, 要找出违规的那个存储过程也是相当麻烦的, 在不理想的情况下甚至出现问题。不过使用触发器, 对客户余额的更新就会变得非常简单。

事实上, 我们可以维护所有要跟踪的聚集信息。不过必须记住, 每一个加入的触发器都会增加完成交易所必须的工作量。这就意味着给系统增添了额外的负担, 而且加大了陷入死锁的可能性。

12.3.2 向反规范化的表输入数据以用于报告

首先要说明, 在大多数情况下不应该使用这种方式。通常应该将这类数据传输视为在夜间或者系统非高峰时段运行的一部分批处理过程来操作——根据要传输事物的性质, 最佳答案也可能是复制。第 17 章会详细讨论复制。

之前已经说过了, 有时需要报告表中的数据正好是最新的。解决这个问题的唯一办法是: 要么修改所有的存储过程和系统的其他访问点, 以便于在更新联机事务处理(OLTP)表的同时更新报告表(这很让人讨厌!), 要么使用触发器传递记录的所有更新。

使用这种方法传递数据的好处是, 总是能确定 OLTP 表中最新发生了什么变化。话虽如此, 它却破坏了保存独立报告表的大部分意图。尽管用反规范化的格式保存数据可以大大提升查询性能, 不过在大多数安装中, 这样做主要是为了减少重要的 OLTP 数据库中的报告并尽可能减少并发问题。如果所有的 OLTP 更新还必须更新报表中的信息, 那么, 已经完成的避免数据库中发生死锁或其他并发问题。从 OLTP 的观点来看, 增加的工作不会带来任何好处。

这里必须要权衡: 是否值得牺牲 OLTP 系统的性能来提高报告的性能。

12.3.3 设置条件标志

使用这种情况类似于聚集——在变化发生时维护一个标志而不必在整张表上查找特定的条件。虽然查找标志会破坏规范化规则(不应该存储可以从其他地方导出的数据), 不过它们的确可以大大提升系统性能。

这里给出了这个问题的一个实例,假设要维护出售产品的各种信息。材料安全数据表(MSDS)是关于供应商的信息——假想这里可以有无限多的不同文档,所有的文档都提供产品的信息。现在,进一步设想我们的产品数量比 AdventureWorks2008 数据库中 504 种产品多(在商业贸易中,编目里有 50 000 或更多不同的产品是很平常的)。可能的信息记录数量可以变化很大。

这里希望可以在客户支持窗口中放置标志,告诉接收订单的人是否有更多的额外产品信息。如果完全按照规范化数据库的规则,必须查看 ProductDocument 表,查看这个表中是否存在与 ProductID 匹配的记录。

不需要完成那些查找,只要向 Products 表加入一个标识“是”或“否”的位字段,指示是否存在其他可用信息。随后可以在 ProductInformation 表上放置一个更新 Products 表中位字段的触发器。如果记录被插入 ProductInformation 表,那么相应产品的位标志会被设置为 TRUE。删除 ProductInformation 表中的记录时,会查看它是否是最后一条记录,如果答案是肯定的,那么会将 Products 表中的位标志设置为 FALSE。

下面将给出一个简短的例子。首先要创建位标记字段和 ProductDocument 表:

```
ALTER TABLE Production.Product
  ADD InformationFlag bit NOT NULL
  CONSTRAINT InformationFlagDefault
  DEFAULT 0 WITH VALUES;
```

为了顾及已有的文档,必须对表中的数据进行修改:

```
UPDATE p
SET p.InformationFlag = 1
FROM Production.Product p
WHERE EXISTS
  (
    SELECT 1
    FROM Production.ProductDocument pd
    WHERE pd.ProductID = p.ProductID
  );
```

随后可以添加触发器:

```
CREATE TRIGGER DocumentBelongsToProduct
  ON Production.ProductDocument
  FOR INSERT, DELETE
AS
  DECLARE @Count int;

  SELECT @Count = COUNT(*) FROM Inserted;

  IF @Count > 0
  BEGIN
    UPDATE p
    SET p.InformationFlag = 1
    FROM Inserted i
    JOIN Production.Product p
      ON i.ProductID = p.ProductID;
  END

  IF @@ERROR != 0
    ROLLBACK TRAN;

  SELECT @Count = COUNT(*) FROM Deleted
```



```

IF @Count > 0
BEGIN
    UPDATE p
        SET p.InformationFlag = 0
        FROM Inserted i
        RIGHT JOIN Production.Product p
            ON i.ProductID = p.ProductID
        WHERE i.ProductID IS NULL
END

IF @@ERROR != 0
    ROLLBACK TRAN;

```

准备好进行测试:

```

SELECT ProductID, InformationFlag
FROM Production.Product p
WHERE p.ProductID = 1;
INSERT INTO Production.ProductDocument
    (ProductID, DocumentNode)
VALUES
    (1, 0x);
SELECT ProductID, InformationFlag
FROM Production.Product p
WHERE p.ProductID = 1;

```

结果表明, 这样做产生了正确的更新:

ProductID	InformationFlag
1	0

(1 row(s) affected)

(1 row(s) affected)

(1 row(s) affected)

ProductID	InformationFlag
1	1

(1 row(s) affected)

执行删除操作:

```

DELETE Production.ProductDocument
WHERE ProductID = 1
    AND DocumentNode = 0x;

```

```

SELECT ProductID, InformationFlag
FROM Production.Product p
WHERE p.ProductID = 1;

```

再次返回了正确的更新:

ProductID	InformationFlag
1	0

(1 row(s) affected)

现在,可以在用于获取产品基本信息的查询中查看是否存在产品文档。除非确实要检索出某些内容,否则不用付出开销去查询 ProductDocument 表。

12.4 其他触发器问题

现在,我们已经了解了很多有关触发器的问题,但是,如果你认为已经完成了对触发器的学习,那么请三思。正如本章之前说过的,触发器制造了许多要思考的问题。下面的几节将尝试指出一些需要重点考虑问题,并提供其他有关触发器功能和潜力的信息。

12.4.1 嵌套触发器

你发出语句不会直接触发嵌套的触发器,嵌套的触发器是由另一个触发器发出的语句触发的。

这样做实际上可以触发一连串的事——一个触发器引起另一个触发器的触发,而这个触发器接着引起另一个触发器触发,依此类推。触发器能引起多少相关的触发器触发,这取决于以下几个方面:

- 是否为系统启动嵌套触发器(这是系统级选项,而非数据库级选项。它通过 ManagementStudio 或 sp_configure 设置,其默认值为启用)。
- 是否限制最多嵌套 32 级。
- 是否已经触发触发器。在默认情况下,一个触发器只能在每个触发器事务中被触发一次。一旦触发器被触发,它会忽略掉同一触发器操作中的活动引发的其他任何调用。一旦继续进入到一个全新的语句(甚至就在同一个完整的事务中),这个过程又重新开始了。

大多数情况下,确实需要触发器嵌套(因此在默认情况下是允许嵌套的),但是,必须考虑进入触发器触发触发器的循环中可能会发生的情况。如果两次都返回到同一个表中,那么触发器就不会被第二次触发,因此不会发生你可能认为重要的事情。例如,可能会出现违反数据完整性的情况。还要注意的,如果在嵌套链的任何地方执行了 ROLLBACK,那么整个链都将被回滚。换句话说,把整个嵌套的触发器链当作一个事务来对待。

12.4.2 递归触发器

什么是递归触发器?当触发器的某些操作最终会触发同一个触发器时,该触发器被认为是递归的。递归可以是直接递归(通过在设置触发器的表上进行操作查询)也可以是间接递归(通过嵌套过程)。

递归触发器比较少见。在默认情况下,递归触发器实际上是禁用的。尽管如此,这种方法可以处理之前描述过的情况:在进行触发器嵌套时,希望能发生第二次更新。不同于嵌套,递归是数据库级选项,可以使用 sp_dboption 系统存储过程来设置。

使用递归触发器可能引发的危险是:会进入某种非预期的循环。因此,必须确保在适当的地方进行某种形式的递归检查,以便于在必要时终止这个过程。

12.4.3 触发器调试

调试触发器非常棘手。因为存在某种程度的间接性(编写语句触发触发器,而不是自己明确地触发它),所以似乎必须在事后猜测发生了什么情况。

可以利用第 10 章中用过的相同的调试器——只不过使用它需要技巧。要使用什么技巧呢?这里要创建一段代码触发触发器(存储过程或批处理),然后进入并单步执行那段代码。这样就能直接进入触发器并单步执行。

如果使用内置工具调试,则需要耐心,可以使用 PRINT 和 SELECT 语句输出触发器中的值。它们除了可以告知我们变量在整个过程中的变化情况,还能透露递归和某些情况下的嵌套问题。

提示:

嵌套问题可能是触发器设计中最大的问题。你会发现,在执行命令时,由于不了解会依次触发多少个其他的触发器,导致最终得到的值并非预期结果,这样的情况屡见不鲜。更有甚者,如果嵌套的触发器对发起表执行更新,那么这个触发器将不能被第二次触发——这样做会在某些地方造成数据完整性问题,而这些地方正是你确信触发器可以避免数据完整性问题的地方。第一次触发时,触发器的代码或许是正确的,但是在嵌套的情况下,它甚至不会运行第二次。

也可以使用 SELECT @@NESTLEVEL 显示自己进入嵌套级别的深度。

不过要记住,PRINT 和结果集(生成 SELECT 语句)只会将数据发送至屏幕(在 Management Studio 中)或将其作为信息性消息发送(数据访问模型)。通常这样做是最令人不解的。因此,这里强烈推荐在完成调试之后,进入到生产版本前,先删除这些语句。

12.4.4 触发器不妨碍架构的修改

这是典型的有利有弊的情况。

使用触发器可以方便地修改架构。事实上,在开发周期的早期(此时很可能要对数据库的设计进行大量的修改),我常常使用触发器保持引用完整性,随后在即将投入生产的时候,改为使用 DRI。

如果要使用 DRI 删除并重新创建表,必须在删除表之前先删除所有的约束。删除多个约束、完成自己的修改、然后再添加约束,这个过程可能非常复杂。为了运行修改过的脚本尝试删除所有必需删除的东西会导致严重的混乱。然后,确保所有必需的东西都已经被添加同样令人感到迷惑。触发器可以处理上述所有的情况,因为只有在实际运行的时候才会关注变化。

可是,难点出现在——它们运行的时候。可以看出,这意味着你在没有意识到自己做了什么之前,修改了架构并破坏了若干个触发器。只有在那些触发器首次试图处理讨论中的对象时,你才会发现错误。此时你会发现,很难将已经做过的准确地拼凑在一起,也无法解释原因。

两边都各有自己的麻烦,无论采用哪种方法,都要记住它们的麻烦之处。

12.4.5 不必删除就可以禁用触发器

就像 CHECK 约束一样,有时候你想要禁用完整性功能,这样一来,就可以完成那些将会违反约束但依然有理由发生的事情(其中最常见的是数据导入)。

这样做的另一个理由是,在进行某种大容量插入时(再次涉及导入),已经百分之百确定数据是有效的。在这种情况下,或许想要禁用触发器,以消除它们的开销并加快插入的速度。

可以使用 ALTER TABLE 语句来禁用和启用触发器。其语法如下所示：

```
ALTER TABLE <table name>  
<ENABLE|DISABLE> TRIGGER <ALL|<trigger name>>
```

你可能已经料想到了，这里最大的告诫是，“不要忘记再次启用触发器！”

还有最后一件事情要说明。如果为了进行某种大规模的数据导入而禁用触发器，我强烈建议注销所有的用户，然后进入 RESTRICTED_USER 模式。这样做可以确保没有人可以在禁用触发器时偷偷潜入。

注意：

在消除安全问题的时候，一定要考虑是否可以禁用触发器。如果希望触发器代替你完成审核工作，不过却允许禁用触发器(诚然，它们可能已经具有了某种安全等级，不过还是必须全面考虑各种可能性)，那么这样一来，审核过程中一定有漏洞。

12.4.6 触发器的触发顺序

在很早的 SQL Server 版本(7.0 或之前的版本)中，触发顺序是不能控制的。实际上你可能会提醒我之前讨论过在 7.0 版之前，任何特定类型的触发器(INSERT、UPDATE、DELETE)都只能有一个，因此触发顺序没有什么实际意义。之后的 SQL Server 版本中，对于以何种顺序触发某种触发器的问题有了一定的控制。对于任何给定的表(并非视图，因为只能为 AFTER 触发器指定触发顺序，而视图只接受 INSTEAD OF 触发器)，你可以选定一个(而且只能是一个)最先被触发的触发器。同样，也可以选出一个(而且只能是一个)触发器最后被触发。其他所有的触发器都被认为没有先后触发顺序——也就是说，对于触发顺序为“未定义”的触发器来说，只能保证它们在 FIRST 触发器(如果有的话)完成之后，在 LAST 触发器完成之前(如果有的话)被触发，而不能保证它们具体的触发顺序(如图 12-3 所示)。

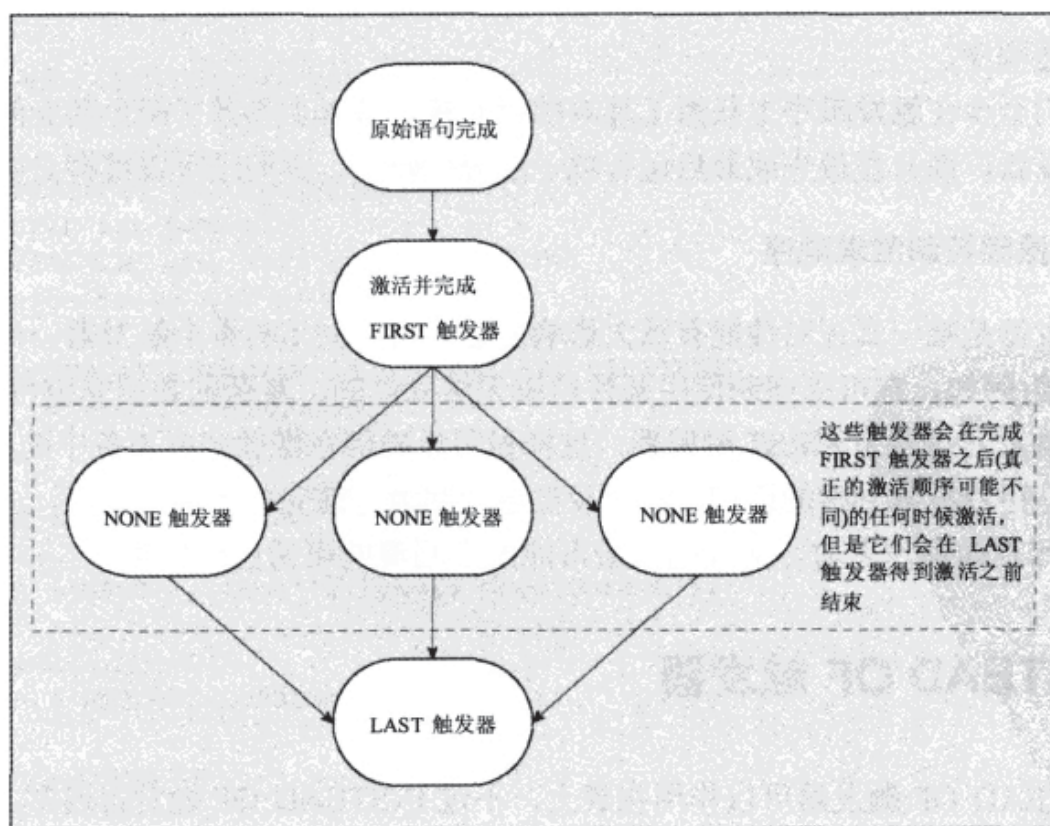


图 12-3

创建第一个被触发的触发器和创建最后一个被触发的触发器完全一样。在已经创建了触发器之后，可以通过使用特殊的系统存储过程 `sp_settriggerorder` 指定触发的优先权。

`sp_settriggerorder` 的语法如下所示：

```
sp_settriggerorder[@triggername =] '<trigger name>',  
    [@order =] '{FIRST|LAST|NONE}',  
    [@stmttype =] '{INSERT|UPDATE|DELETE}'
```

对于任何特定的操作(INSERT、UPDATE 或 DELETE)，只能有一个“第一个被触发的”触发器。同样，对于任何特定的操作只能有一个“最后一个被触发的”触发器。被认为是“未定义”的触发器可以有很多——即，没有特定触发顺序的触发器数量是不受限制的。

此时问题应该是，“为什么要在意它们的触发顺序呢？”好吧，我们通常不关心触发顺序。不过有时候，触发顺序可能在逻辑上十分重要，或者是提高性能的好方法。接下来，仔细考虑一下其中的细节。

1. 为逻辑原因控制触发顺序

为什么要在其他触发器被触发之前触发某个触发器？最常见的原因可能是，第一个触发器可以为后面将要发生的事情奠定某种基础，或者使后面的事情生效。在 SQL Server 6.5 或更早的版本中，不必过多考虑这类事情——给定表上任何特定的操作类型(UPDATE、DELETE 或 INSERT)都只能有一个触发器。这意味着令一件事情在另一件事情之前发生不是真正的问题。因为将所有的逻辑组合放在一个触发器中，只需要把第一件要发生的事情放在代码的最前面，把最后要发生的事情放在代码的最后即可(这完全不是什么高深的学问)。

7.0 版本出现之后，事情开始变得比以前更好，不过也比以前更坏了。你不必把所有的逻辑塞入一个触发器内。这实在很不错，因为为了方便管理代码，可以在物理上隔离不同逻辑的触发器各部分的代码，并禁用某部分代码(还记得前几节关于 NO CHECK 的讲述吗？)，同时允许继续运行其他部分代码。这样做的弊端是，如果以这种方式隔离代码，就会失去代码在一个触发器的时候逻辑上的步进顺序。

眼下，我们至少在触发顺序上获得了基本的控制权，基本上具备了两全其美的效果：可以在逻辑上隔离触发器，而且在最先或最后运行哪一部分代码上，我们还可以维持必要的优先顺序。

2. 为性能原因控制触发顺序

FIRST 触发器是唯一真正对性能有重大影响的触发器。如果有多个触发器，但只有一个触发器可以产生回滚(例如，它可以强制使用某种约束无法处理的、复杂的数据完整性规则)，你或许要考虑将这个触发器设置为 FIRST 触发器。这样做可以确保在继续完成事务中的其他操作之前，已经完成了那些可能导致回滚的语句。在发现要进行回滚之前完成的事情越多，必须回滚的事情也越多。在进行其他操作之前，必须确保最可能发生回滚的事情已经发生。

12.5 INSTEAD OF 触发器

尽管 INSTEAD OF 触发器可以作用在表上，不过 INSTEAD OF 触发器通常主要用来对视图进行更新(此前，这是无法实现的)。

INSTEAD OF 触发器实质上是一段代码,这些代码可以拦截任何人对表或视图所完成的操作。我们可以选择继续完成用户的要求,也可以选择执行完全不同的操作。

与 FOR(AFTER)触发器一样,INSTEAD OF 触发器有三种不同的类型——INSERT、UPDATE 和 DELETE。不同于 FOR(AFTER)触发器,每一个表或视图上的每种不同操作类型(INSERT、UPDATE 和 DELETE)只能有一个 INSTEAD OF 触发器。

如果要研究这些问题,需要有一些恰当的示例表。为此,下面给出了四个表(如果愿意的话,也可以将脚本改为使用现有的数据库)

```
CREATE DATABASE OurInsteadOfTest;
GO

USE OurInsteadOfTest;

CREATE TABLE dbo.Customers
(
    CustomerID varchar(5) NOT NULL PRIMARY KEY ,
    Name varchar(40) NOT NULL
);

CREATE TABLE dbo.Orders
(
    OrderID int IDENTITY NOT NULL PRIMARY KEY,
    CustomerID varchar(5) NOT NULL
        REFERENCES Customers(CustomerID),
    OrderDate datetime NOT NULL
);

CREATE TABLE dbo.Products
(
    ProductID int IDENTITY NOT NULL PRIMARY KEY,
    Name varchar(40) NOT NULL,
    UnitPrice money NOT NULL
);

CREATE TABLE dbo.OrderItems
(
    OrderID int NOT NULL
        REFERENCES dbo.Orders(OrderID),
    ProductID int NOT NULL
        REFERENCES dbo.Products(ProductID),
    UnitPrice money NOT NULL,
    Quantity int NOT NULL
        CONSTRAINT PKOrderItem PRIMARY KEY CLUSTERED
            (OrderID, ProductID)
);

-- INSERT sample records
INSERT dbo.Customers
VALUES ('ABCDE', 'Bob''s Pretty Good Garage');

INSERT dbo.Orders
VALUES ('ABCDE', CURRENT_TIMESTAMP);

INSERT dbo.Products
VALUES ('Widget', 5.55),
      ('Thingamajig', 8.88)
```



```
INSERT dbo.OrderItems
VALUES (1, 1, 5.55, 3);
```

在接下来的三个 INSTEAD OF 触发器实例中会用到这些表。

12.5.1 INSTEAD OF INSERT 触发器

INSTEAD OF INSERT 触发器允许检查即将插入表或视图中的数据，并在物理插入前决定如何处理它。通常，该触发器的典型应用是在视图上——在尝试物理插入数据前对数据进行操纵，这意味着插入成功或失效的差异。

下面创建一个可更新视图的实例——确切地讲，是接受 INSERT 操作的视图。不过在使用 INSTEAD OF INSERT 触发器之前，无法执行这样的操作。

为了说明更新问题，这里会创建一个视图，然后探讨如何解决这个问题。下面以显示一些订单商品为例，不过，还要显示更多有关客户和产品的完整信息(确保目前使用的数据库是在其中创建示例表的数据库)：

```
USE OurInsteadOfTest;
GO

CREATE VIEW CustomerOrders_vw
WITH SCHEMABINDING
AS
SELECT o.OrderID,
       o.OrderDate,
       od.ProductID,
       p.Name,
       od.Quantity,
       od.UnitPrice
FROM dbo.Orders AS o
JOIN dbo.OrderItems AS od
  ON o.OrderID = od.OrderID
JOIN dbo.Products AS p
  ON od.ProductID = p.ProductID;
```

```
CREATE VIEW CustomerOrders_vw
WITH SCHEMABINDING
AS
SELECT o.SalesOrderID,
       o.OrderDate,
       od.ProductID,
       p.Name,
       od.OrderQty,
       od.UnitPrice,
       od.LineTotal
FROM Sales.SalesOrderHeader AS o
JOIN Sales.SalesOrderDetail AS od
  ON o.SalesOrderID = od.SalesOrderID
JOIN Production.Product AS p
  ON od.ProductID = p.ProductID
```

目前这个视图并不是完全可更新的。SQL Server 怎么知道哪些数据应该被放入哪些表中呢？当然，可能有理由使用直接更新的语句，不过此处并没有每个表的主键。更糟糕的是，如果要进行插入又该怎么办呢？

SQL Server 不能给出这个问题的答案——你必须提供更多的指令，告知在这种复杂的情况下

应该做些什么。这就是 INSTEAD OF 触发器大显身手的地方。

下面看一看作为示例的订单：

```
SELECT *
FROM CustomerOrders_vw
WHERE OrderID = 1;
```

这返回为这个示例准备的一行：

```
Bob's Pretty Good Garage...1...2006-04-13 05:14:22.780...1...Widget...3...5.55
```

现在尝试插入一个新的订单项以证明无法进行插入操作：

```
INSERT INTO CustomerOrders_vw
(
    OrderID,
    OrderDate,
    ProductID,
    Quantity,
    UnitPrice
)
VALUES
(
    1,
    '1998-04-06',
    2,
    10,
    6.00
)
```

正如我们所料想的，这条语句无法正常工作：

```
Server: Msg 4405, Level 16, State 1, Line 2
View or function 'CustomerOrders_vw' is not updatable because the modification affects
multiple base tables.
```

现在可以用 INSTEAD OF 触发器处理这个问题了。这里要做的就是必须提前决定要处理什么样的情况以及如何处理这种情况(此时，只需插入新的 OrderItem 记录)。

接下来要尝试将 INSERT 视为添加一个新的订单项。本例假定已经存在客户，而且 OrderID 是可用的(如果要更加复杂的话，可以进一步分解问题)。此时的触发器如下所示：

```
CREATE TRIGGER trCustomerOrderInsert ON CustomerOrders_vw
INSTEAD OF INSERT
AS
BEGIN
    -- Check to see whether the INSERT actually tried to feed us any rows.
    -- (A WHERE clause might have filtered everything out)
    IF (SELECT COUNT(*) FROM Inserted) > 0
    BEGIN
        INSERT INTO dbo.OrderItems
        SELECT i.OrderID,
               i.ProductID,
               i.UnitPrice,
               i.Quantity
        FROM Inserted AS i
        JOIN Orders AS o
            ON i.OrderID = o.OrderID;
        -- If we have records in Inserted, but no records could join to
```



```
-- the orders table, then there must not be a matching order
IF @@ROWCOUNT = 0
    RAISERROR('No matching Orders. Cannot perform insert',10,1);
END
END
```

下面再次尝试插入:

```
INSERT INTO CustomerOrders_vw
(
    OrderID,
    OrderDate,
    ProductID,
    Quantity,
    UnitPrice
)
VALUES
(
    1,
    '1998-04-06',
    2,
    10,
    6.00
)
```

这里已经明确说明了要插入到什么表中,因此 SQL Server 将欣然从命。如果需要的话,可以方便地对其进行扩展,寻访那些视图中没有的、不允许为空的列(客户不能向这些列提供数值,因为客户正在使用的视图中没有这些列)

12.5.2 INSTEAD OF UPDATE 触发器

现在,已经知道了在视图上进行插入操作的 INSERT 语句可能导致情况不明,以及如何使用 INSTEAD OF INSERT 触发器解决这个问题——但是,如果进行的是更新,情况又将如何呢?

即便是更新数据,语句也可能会模棱两可;如果对 CustomerOrders_vw 表中的 ProductName 进行更新,这是否意味着要改变产品的实际名称,抑或是要改变该项目正在销售的产品?当然,答案要随情况的变化而变化。对于一个系统,可能要改变 ProductName,而对于另一个系统来说,或许要改变售出的产品。

与 INSTEAD OF INSERT 触发器非常相似,INSTEAD OF UPDATE 触发器能够捕获将会出现什么情况并明确地处理它。在 ProductName 的例子中,可以选择以任何一种方式来进行处理。默认情况下,SQL Server 可以更新 Products 表中的名字。不过如果用户需要,我们可以使用 INSTEAD OF UPDATE 触发器来捕获它,并查找 ProductName 以便于找到 ProductID。此后,如果提供的 ProductID 与对应的名字不匹配,就可能会生成错误。

12.5.3 INSTEAD OF DELETE 触发器

关于 INSTEAD OF 触发器,最后要讲述的是 INSTEAD OF DELETE 触发器,这可能也是最不常用的。与另外两种 INSTEAD OF 触发器一样,这类触发器几乎专门用于允许视图在一个或多个底层表中删除数据。

下面继续前面的 CustomerOrders_vw 实例,这里将加入删除功能。不过,这次复杂度要稍微

高一些。我们要删除给定订单中所有的行，不过如果删除这些行意味着订单中再没有任何详细的订单项，那么还要删除订单标题。

从上一节可以知道(假设你一直在阅读本书)，订单 1 中有两行(其中一个是在创建表时加入的，另一个是在 `INSTEAD OF INSERT` 例子中插入的)。不过，在开始删除之前，要先创建触发器：

```
CREATE TRIGGER trCustomerOrderDelete ON CustomerOrders_vw
INSTEAD OF DELETE
AS
BEGIN

    -- Check to see whether the DELETE actually tried to feed us any rows
    -- (A WHERE clause might have filtered everything out)
    IF (SELECT COUNT(*) FROM Deleted) > 0
    BEGIN
        DELETE oi
            FROM dbo.OrderItems AS oi
            JOIN Deleted AS d
                ON d.OrderID = oi.OrderID
                AND d.ProductID = oi.ProductID;

        DELETE Orders
            FROM Orders AS o
            JOIN Deleted AS d
                ON o.OrderID = d.OrderID
            LEFT JOIN OrderItems AS oi
                ON oi.OrderID = d.OrderID
                AND oi.ProductID = d.ProductID
            WHERE oi.OrderID IS NULL;
    END
END
```

现在可以测试了。一开始我们会从 `CustomerOrders_vw` 视图中删除一行：

```
DELETE CustomerOrders_vw
WHERE OrderID = 1
AND ProductID = 2;
```

可以再次运行选择语句：

```
SELECT ProductID, UnitPrice, Quantity
FROM CustomerOrders_vw
WHERE OrderID = 1;
```

毫无疑问，最初在 `INSTEAD OF INSERT` 小节中插入的那一行被删掉了：

ProductID	UnitPrice	Quantity
1	5.55	3

(1 row(s) affected)

如此看来，删除单独明细行的操作进行得非常顺利。接下来慢慢删除整个订单：

```
DELETE CustomerOrders_vw
WHERE OrderID = 1
```

其实，需要直接使用 `Orders` 表就可以检查上面的语句是否成功：


```
SELECT * FROM Orders WHERE OrderID = 1;
```

当然，订单已被删除了。

注意：

虽然使用 INSTEAD OF DELETE 触发器时不用考虑独立的列(因为是按行删除而不是按列删除)，不过这里意识到定义了 INSTEAD OF DELETE 触发器的表(视图)上存在哪些引用完整性操作。就像 INSTEAD OF UPDATE 触发器一样，不允许在拥有引用完整性操作的表上定义 INSTEAD OF DELETE 触发器。

12.6 IF UPDATE()和 COLUMNS_UPDATED()

在 UPDATE 触发器中，通过检查是否改变了我们感兴趣的列，通常可以限制触发器中实际执行的代码数量。为此，需要使用 UPDATE()或 COLUMNS_UPDATED()函数。下面将一一讲述这两个函数。

12.6.1 UPDATE()函数

UPDATE()函数只与触发器内部的作用域有关。它唯一的目的是，返回一个布尔值(真或假)标识是否更新了某个特定列，可以使用这个函数来确定是否需要运行特定的一段代码——例如，只有在更新特定列时，代码才是相关的。

接下来修改之前的一个触发器，运行一个简短的例子：

```
USE AdventureWorks2008;
GO

ALTER TRIGGER Production.ProductAudit
ON Production.ProductInventory
FOR INSERT, UPDATE, DELETE
AS
IF UPDATE(Quantity)
BEGIN
    INSERT INTO Production.InventoryAudit
    (ProductID, NetAdjustment)
    SELECT COALESCE(i.ProductID, d.ProductID),
           ISNULL(i.Quantity, 0) - ISNULL(d.Quantity, 0) AS NetAdjustment
    FROM Inserted i
    FULL JOIN Deleted d
    ON i.ProductID = d.ProductID
    AND i.LocationID = d.LocationID
    WHERE ISNULL(i.Quantity, 0) - ISNULL(d.Quantity, 0) != 0;
END
```

完成这些改动后，现在限制只有在改变 Quantity 列(我们关注该列)的时候才运行其他代码。用户可以修改任何其他列的值，不过我们并不在乎。这意味着要执行更少的代码行，因此该触发器的性能会比之前的触发器性能更好一些。

12.6.2 COLUMNS_UPDATED()函数

这个函数的工作方式不同于 UPDATE()，但是二者的用途相同。COLUMNS_UPDATED()使我

们能够一次检查多列。为了实现这个目标，函数使用位掩码将一个或多个字节的 varbinary 数据中单独的位与表中单独的列联系起来。最终结果如图 12-4 所示。

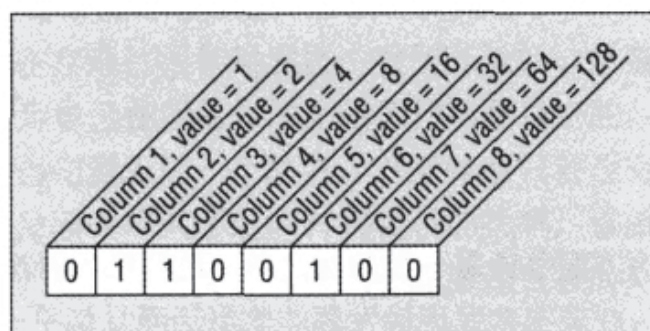


图 12-4

此时，图中的单字节数据表明更新了第 2 列、第 3 列和第 6 列——没有更新其余的列。如果列数多于八个，那么 SQL Server 只会向右边再添加一个字节，并继续计数(如图 12-5 所示)。

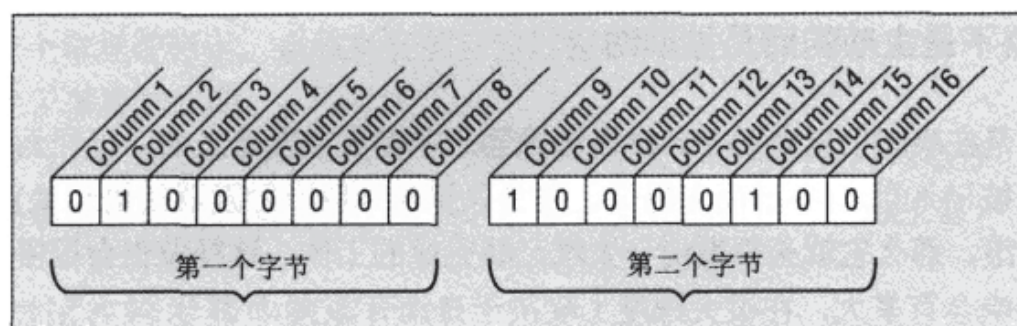


图 12-5

这一次更新了第 2 列、第 9 列和第 14 列。

你可能会说：“噢，这样很好呀——不过我怎么利用它呢？”要回答这个问题，必须进入布尔代数领域。

利用这一信息就意味着需要将所有字节的布尔值相加，而且要考虑到最左边的数字是最不重要的。这样，如果比较要考虑到 2、5 和 7，就必须将每个比特的二进制值相加： $2+16+64$ 。随后你需要使用位运算符将所有列的二进制码与比特掩码相比：

- | 表示位运算符 OR；
- & 表示位运算符 AND；
- ^ 表示位操作运算符 Exclusive OR。

回头阅读刚才写下的内容时，我意识到这是正确的，不过它非常含混不清，因此要用几个例子来让你更进一步了解我的意思。

假设更新了一个包含 5 个列的表。如果更新的是第 1 列、第 3 列和第 5 列，那么 COLUMNS_UPDATED 使用的位掩码包含 10101000，由 $1+4+16=21$ 得到。这里可以使用：

- `COLUMNS_UPDATED() > 0` 以找出是否更新了任何列；
- `COLUMNS_UPDATED() ^ 21 = 0` 以找出是否更新了所有指定的列(这里是 1、3 和 5)，而且没有更新其他列；
- `COLUMNS_UPDATED() & 21 = 21` 以找出是否更新了所有指定的列，不过并不关注其他列的状态是否被更新；
- `COLUMNS_UPDATED() | 21 != 21` 以找出除了感兴趣的列之外是否有其他的列被更新。

提示:

要知道这样做很难——对于大多数人来说，布尔数学是很难掌握的概念，因此需要仔细检查并反复地测试，测试，测试！

12.7 性能考虑

关于触发器利弊的正反两方面争论是一种门派之争。最糟糕的看法来自纯化主义者——那些酷爱理论而且只处理理论的人，或者那些发现触发器非常灵活而且似乎一切都要用触发器解决的人。

本章前面已经讲过，在我看来，应该在适当的时候使用触发器。这种说法是否有些模糊不清——一点没错！在编程中很少是黑白分明的，数据库几乎从不会那样。不过这里将指出一些事实以供思考。

12.7.1 触发器不是主动的而是被动的

这里要说的是在事实出现之后才会触发触发器。在触发触发器之前，已经运行了完整的查询而且事务也已经被记入日志(不过尚未提交，并且只记录到触发触发器的语句处)。这意味着，如果触发器必须回滚，那么它就必须撤消一大堆已经完成的工作。这样做也会很慢！要掌握这种平衡知识。总的影响会有多大，在很大程度上取决于查询有多大。

你会说，“那又怎样呢？”比较触发器和约束，约束是主动的——即，在执行你的语句之前它们就已经发生了。这意味着，在大部分工作完成之前，它们避免了发生那些最终会失败的事情。这通常说明使用约束至少会比触发器稍快一些——查询越复杂，速度越快。注意，实际上只有在发生回滚的时候才能体现出额外的速度。

那么最终的分析结果是什么呢？如果处理回滚的机会很少，而且(或者)受影响的语句运行时间和复杂性很低，那么触发器与约束之间的差别就不大。确实有差别，只不过并不显著。不过，如果不能预知回滚的数量，或者你已经知道即将发生较多数量的回滚，那么就应该尽可能使用约束(坦白地说，除非有非常特殊的原因，否则建议使用约束)

12.7.2 触发进程与触发器之间不存在并发问题

在本章中你可能已经注意到了，即使没有发出 `BEGIN TRAN` 语句，也常常会用到 `ROLLBACK` 语句。这是因为触发器总会隐性地成为事务的一部分(触发触发器的那条语句也属于此事务)。

如果触发触发器的语句不是显式事务的一部分(具有 `BEGIN TRAN` 的事务)，那么它将成为自身单语句事务的一部分。无论情况如何，在触发器内部发出的 `ROLLBACK TRAN` 都将回滚整个事务。

这种“触发器与触发它的语句属于同一个事务”的情况会导致另一个结果：触发器继承了它们所在的事务上已经打开的锁。这意味着不必完成什么特别的操作就可以保证不会撞上事务中其他语句创建的锁。我们可以在事务范围内自由访问，而且我们看到的数据库是基于事务内之前的语句所修改的。

12.7.3 简洁明了

我觉得这里声明的内容是很显而易见的，不过理由充分。

我简直说不清有多少次在存储过程和触发器中看到冗长的、愚蠢的代码。不知道是编写代码的人太过匆忙，还是他们认为自己使用的方法无论如何都是很快的，因此不要把它们放在心上。

记住触发器和调用它的语句属于同一个事务。这意味着只要没有完成触发器，语句就没有结束。思考一下——如果在触发器中写入了运行时间很长的代码，那就意味着要长时间运行自己创建的触发器的所有代码。要弄清楚为什么代码会花费这么长时间是件非常痛苦的事情。你编写的存储过程看似非常高效，不过表现得却很糟糕。你也许会花上几周的时间却全然不知存储过程没有问题——只是触发了一个性能不佳的触发器而已。

12.7.4 选择索引时不要忘记触发器

这里有另一个常见的错误。查找所有的存储过程和视图，找出最佳索引——不过完全不记得在触发器中运行了重要的代码。

这与 12.7.3 节中的观念一致——长时间运行查询导致了语句的长时间运行，长时间运行语句继而导致所有的事情都会长时间运行。所以，不要在优化的时候忘记触发器！

12.7.5 不要尝试在触发器中回滚

因为回滚在通过触发器实现的各种任务中通常是非常重要的部分，所以要做到这一点非常困难。

只要记住，AFTER 触发器(这肯定是最常用的触发器类型)只有在完成大部分工作之后才会被触发——这就意味着回滚代价不菲。在这里，DRI 几乎彰显了所有的性能优势。如果在触发器中使用大量的 ROLLBACK TRAN 语句，那么，请务必保证在执行会触发触发器的语句之前，预先处理了查找错误。即，因为 SQL Server 在这种情况下不是主动的，就由你来为它进行预处理。应该预先查找手头上的错误而不是消极地等待回滚。

12.8 删除触发器

删除触发器可能是迄今为止所遇到的最简单的操作：

```
DROP TRIGGER <trigger name>
```

执行上面的语句后，触发器就被删掉了。

12.9 小结

触发器是一种功能强大的工具，它能够为数据完整性和系统的总体操作增加极大的灵活性。虽然这样，但是也不能对触发器掉以轻心。如果触发器用的恰到好处，可以大大提高系统的性能，不过它们也可能成为毒药。触发器很难调试(即使有调试器)，而且编写得很差的触发器不仅会影响触发器自身，而且会影响到触发这个触发器的任何语句。



第13章

SQL 游 标

到现在为止，本书中所处理的都是行集中的数据。这与过程驱动的语言处理事情的方式是相反的。事实上，当数据到达客户端时，几乎总是要获取行集，然后逐行地处理。这里用到了游标。即使在传统的 SQL Server 工具中，如果在使用新的基于 CLR 的语言支持的脚本中利用了非面向 SQL 的语言，就会最终成为某种游标模式。

本章将讨论以下内容：

- 什么是游标
- 游标的生命期
- 游标类型(敏感性和可滚动性)
- 游标的使用

我们将发现，在创建游标时有很多要考虑的内容。

注意：

在创建游标时，可能最需要考虑的事情是，“是否有办法避免使用游标？”如果每次准备创建游标时都自问这个问题，那么将可能得到性能更好的系统。虽说如此，有时我们别无其他选择。

13.1 游标的含义

所谓游标，是指取用一组数据并能够一次与一个单独的记录进行交互的方法。它发生的频率并不像我们所想象的那样频繁，不过有时，确实不能通过在整个行集中修改或者甚至选取数据来获得我们所要的结果。行集是由所有行共有的一些东西生成的(通过 SELECT 语句定义)，但是随后需要逐一处理这些行。

放入游标中的结果集有几个显著的特征，这就将其与标准的 SELECT 语句区分开了：

- 声明游标与实际执行游标的内容是分开的。
- 游标以及结果集在声明中命名，然后通过名称来引用它。
- 游标中的结果集一旦打开，就一直保持打开状态，直至你关闭它。

- 游标有一组专门用来导航记录集的命令。

虽然 SQL Server 自带的引擎处理游标,但有一些不同的对象库也能在 SQL Server 中创建游标。

- SQL Native Client(由 ADO.NET 使用)
- OLE DB(由 ADO 使用)
- ODBC(由 RDO、DAO、OLE DB/ADO(有时)使用)
- JDBC(由 Java 使用)
- DB-Lib(是个久远的遗留产品,但在一些较老的应用程序中仍有使用)。

客户端应用程序一般使用这些库来访问单独的记录。每一种都有其特有的导航记录集的或者管理游标的语法。不过,它们有共同的一组基本概念,因此,一旦获得了游标的一个对象模型,对于它们来说,就完成了一大半的工作。

提示:

所有的数据访问 API(ADO.NET、ADO、ODBC、OLE DB、JDBC 等)都在游标中向客户端应用程序或组件返回数据。这是非 SQL 编程语言当前处理事情的唯一方式。这也是这种类型的游标与 SQL Server 游标之间存在较大差异的原因。如果使用 SQL Server 游标,通常可以选择像行集操作一样执行,这是 SQL Server 被设计来做的事情。如果使用基于 API 的游标,所拥有的一切就是游标,因此,不会在服务器端的活动中出现游标与非游标的争论。

数据处理中客户端的部分将用游标来完成。那只是个假设,因此不用担心。不过,要担心的是使服务器端的数据访问尽可能高效;这意味着如果可能,在服务器端尽量不使用游标。

13.2 游标的生命期

游标有很多较小的组成部分,但是,我们最好先讨论一下游标最基本的形式,然后再开始逐步建立游标。

在了解实际的语法之前必须知道,使用游标时不止用一个语句——实际要用到几个语句。其主要部分包括:

- 声明
- 打开
- 使用或导航
- 关闭
- 释放

虽说如此,声明游标的基本语法如下所示:

```
DECLARE <cursor name> CURSOR  
FOR <select statement>
```

记住,这是特别简单的表达。要尽可能使用默认值创建游标。在本章后面,我们将看到更高级的游标。

除了不需要“@”前缀外,游标名与任何其他变量名一样,必须遵循 SQL Server 命名规则。SELECT 语句可以是任何返回结果集的有效 SELECT 语句。不过要注意,一些结果集是不可更新的(例如,如果使用了 GROUP BY,那么更新组中的哪一部分呢?同理,计算的字段也是不可更

新的)。

接下来开始创建一个相当简单的例子。现在我们还不会大量使用它，但在后面将会看到，以它开头的代码是管理索引的一种相当便利的工具(如果支持旧版本的 SQL Server，也是很好用的)：

```
USE AdventureWorks2008;

DECLARE @SchemaName varchar(255);
DECLARE @TableName varchar(255);
DECLARE @IndexName varchar(255);
DECLARE @Fragmentation float;
DECLARE TableCursor CURSOR FOR
    SELECT SCHEMA_NAME(CAST(OBJECTPROPERTYEX(i.object_id, 'SchemaId') AS int)),
           OBJECT_NAME(i.object_id),
           i.name,
           ps.avg_fragmentation_in_percent
    FROM sys.dm_db_index_physical_stats (DB_ID(), NULL, NULL, NULL, NULL) AS ps
    JOIN sys.indexes AS i
        ON ps.object_id = i.object_id
        AND ps.index_id = i.index_id
    WHERE avg_fragmentation_in_percent > 30;
```

提示：

这只是一个开头。对于游标首先要注意的是，与常见的 SELECT 语句相比，它们需要多得多的代码。

我们声明了一个名为 TableCursor 的游标，它基于一个将选取数据库中所有表的 SELECT 语句。我们还声明了一个变量，在使用游标时，它将包含当前行的值。

只声明游标是不够的，我们需要打开它：

```
OPEN TableCursor;
```

实际上，这将执行作为 FOR 子句的主题的查询，但是我们仍未获得可使用的东西。因此，需要做以下两件事情：

- 获取(或 FETCH)第一条记录
- 必要的话进行循环，获取其余的记录

我们发出第一个 FETCH 语句。这是一个表明要检索特定记录的命令。同时，还必须说明想要将值放置在哪个变量中。

```
FETCH NEXT FROM TableCursor INTO @TableName, @IndexName, @Fragmentation
```

现在，我们有了第一条记录，接下来就可以对游标集执行操作：

```
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT @SchemaName + '.' + @TableName + '.' + @IndexName + ' is '
        + CAST(@Fragmentation AS varchar) + '% Fragmented';
    FETCH NEXT FROM TableCursor INTO @SchemaName, @TableName, @IndexName,
        @Fragmentation;
END
```

每次提取一行时，@@FETCH_STATUS 就会更新，告知我们提取进行情况。其可能的值是：

- 0 Fetch succeeded——一切正常；

- **-1 Fetch failed**——找不到记录(还没有到达末尾,但自打开游标以后,记录已经被删除)。本章后面会对此作进一步的讨论;
- **-2 Fetch failed**——这一次是由于已经超出了游标中的最后一条记录(或者到达第一条之前)。同样,本章后面将对此作更详细的讨论。

对我们这里的目的而言,一旦退出循环,就说明我们已经完成了对游标的使用,因此关闭游标:

```
CLOSE TableCursor ;
```

不过,关闭游标并没有释放与游标相关联的内存。关闭游标只是释放了与游标相关联的锁。为了确保完全释放游标使用的资源,必须取消分配:

```
DEALLOCATE TableCursor ;
```

为了清楚起见,把这些代码集合在一起:

```
DECLARE @SchemaName varchar(255)
DECLARE @TableName varchar(255)
DECLARE @IndexName varchar(255)
DECLARE @Fragmentation float
DECLARE TableCursor CURSOR FOR
    SELECT SCHEMA_NAME(CAST(OBJECTPROPERTYEX(i.object_id, 'SchemaId') AS int)),
           OBJECT_NAME(i.object_id),
           i.name,
           ps.avg_fragmentation_in_percent
    FROM sys.dm_db_index_physical_stats (DB_ID(), NULL, NULL, NULL, NULL) AS ps
    JOIN sys.indexes AS i
        ON ps.object_id = i.object_id
        AND ps.index_id = i.index_id
    WHERE avg_fragmentation_in_percent > 30
OPEN TableCursor
FETCH NEXT FROM TableCursor INTO @SchemaName, @TableName, @IndexName,
    @Fragmentation

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT @SchemaName + '.' + @TableName + '.' + @IndexName + ' is ' + CAST(@Fragmentation
AS varchar) + '% Fragmentented'
    FETCH NEXT FROM TableCursor INTO @SchemaName, @TableName, @IndexName,
        @Fragmentation
END
CLOSE TableCursor
DEALLOCATE TableCursor
```

现在已可以运行了,但是此时已经完成对它的创建,其实只不过是运行了 SELECT 语句本身而已(从技术上讲,事实并非如此,因为不能“PRINT”出 SELECT 语句,但效果实际上是一样的)。

```
Production.ProductInventory.PK_ProductInventory_ProductID_LocationID is 42 . . .
Production.ProductListPriceHistory.PK_ProductListPriceHistory_ProductID_St . . .
Sales.SpecialOfferProduct.PK_SpecialOfferProduct_SpecialOfferID_ProductID . . .
Sales.SpecialOfferProduct.AK_SpecialOfferProduct_rowguid is 50% Fragmenten . . .
. . .
. . .
. . .
Production.ProductCostHistory.PK_ProductCostHistory_ProductID_StartDate is . . .
Production.ProductDescription.AK_ProductDescription_rowguid is 66.6667% Fr . . .
```

```
dbo.DatabaseLog.PK_DatabaseLog_DatabaseLogID is 33.3333% Fragmentented
```

不同之处在于, 如果选择这样做, 那么可以针对每一行做几乎所有的事情。接下来完成我们的小工具以进行说明。

在过去, 没有任何单独的语句可以重建整个数据库中的所有索引(幸运的是, 现在 DBCC INDEXDEFRAG 中有一个选项可以针对整个数据库来实现)。不过, 对索引进行碎片整理是系统管理的核心部分。这里要使用的游标示例是对完成这种索引碎片整理的常见方式的某种演变。然而, 在这个较新的版本中, 我们使用了具体的碎片信息, 并且可使用 ALTER INDEX(在如何确切地进行碎片整理方面提供了更多的选项), 而非 DBCC INDEXDEFRAG。

这样, 我们就有了一些不同的方法来重建或重新组织索引, 而不必完全删除并重新创建索引。其中, ALTER INDEX 是最灵活的, 它可让你选择不同的整理碎片的底层方法(联机或脱机, 完全重建或仅对部分进行重新组织, 等等), 因此, 我们将运用这种方法。ALTER INDEX 的语法的最简单的版本如下所示:

```
ALTER INDEX <index name> | ALL
ON <object>
{[REBUILD] | [REORGANIZE]}
```

这是 ALTER INDEX 最简单的形式。第6章讲述过 ALTER INDEX 的许多其他的开关和选项。

尝试用这里的语句来重建所有表上的所有索引的问题在于, 该语句被设计为一次针对一个表。如果想要创建一个表上的所有索引, 可使用 ALL 选项来替代索引的名称, 但却不能省略表名为所有的表创建所有索引。事实上, 即使已经使用了像 DBCC INDEXDEFRAG 这样的工具——该工具能针对整个数据库来工作, 但却无法拥有很多选项——仍然执行“全有”或“全无”的操作。也就是, 我们不能告诉它只针对高于某个碎片级别的表, 或者排除某些你希望包含碎片的表。

提示:

有时碎片也是个好东西。特别是当要在一个表上进行大量的随机插入时, 碎片可帮助减少页拆分的次数。

通过使用游标动态构建 DBCC 命令, 就可以解决这一问题:

```
USE AdventureWorks2008;

DECLARE @SchemaName varchar(255);
DECLARE @TableName varchar(255);
DECLARE @IndexName varchar(255);
DECLARE @Fragmentation float;
DECLARE @Command varchar(max);
DECLARE TableCursor CURSOR FOR
SELECT SCHEMA_NAME(CAST(OBJECTPROPERTYEX(i.object_id, 'SchemaId') AS int)),
       OBJECT_NAME(i.object_id),
       i.name,
       ps.avg_fragmentation_in_percent
FROM sys.dm_db_index_physical_stats (DB_ID(), NULL, NULL, NULL, NULL) AS ps
JOIN sys.indexes AS i
  ON ps.object_id = i.object_id
 AND ps.index_id = i.index_id
WHERE avg_fragmentation_in_percent > 30;
OPEN TableCursor;
FETCH NEXT FROM TableCursor INTO @SchemaName, @TableName, @IndexName,
                                  @Fragmentation;
```



```

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT 'Reindexing ' + ISNULL(@SchemaName, 'dbo') + '.' + @TableName +
        '.' + @IndexName;
    SET @Command = 'ALTER INDEX [' + @IndexName + '] ON [' + ISNULL(@SchemaName,
        'dbo') + '.' + @TableName + '] REBUILD';
    EXEC (@Command);
    FETCH NEXT FROM TableCursor
        INTO @SchemaName, @TableName, @IndexName, @Fragmentation;
END
CLOSE TableCursor;
DEALLOCATE TableCursor;

```

现在，我们实现了仅仅使用基于行集命令无法完成的事情。ALTER INDEX 命令需要一个参数。向其提供记录集不奏效。通过把集合操作(构成游标的基础的 SELECT)的概念与单一数据点操作(游标中的数据)的概念结合在一起，可以解决这一问题。

为了混合这些基于集合的操作与单一数据点的操作，我们必须采取一系列的步骤。首先，声明游标和所有必需的存放数据的变量。然后“打开”游标，直到这时才真正从数据库中检索出数据。接下来，通过导航游标来使用游标。在这里，我们只向前导航，不过在后面将看到，也可以创建可向前滚动和向后滚动的游标。再接着，我们关闭游标(如果游标仍然有一些打开的锁，则此刻将解开这些锁)，但是，为游标分配的内存依然被占用。最后，取消游标的分配。这时，游标使用的所有资源被释放，这些资源可以被系统中其他对象使用。

就在这短短的时间内，我们就有了自己的第一个游标。不过，这只是开始。除了本例中呈现的这些内容外，还有更多关于游标的内容。接下来，我们将继续深入了解赋予游标更多灵活性的强大功能。

13.3 游标的类型和扩展的声明语法

游标有多种不同的类型(我们将在后面全部接触到)。默认的游标类型是只进的(在记录中只能前进，而不能后退)和只读的，但游标也可是可滚动的和可更新的。另外，对于其他进程对底层数据的修改，它们有不同的敏感性级别。

提示：

在 SQL Server 本身的游标引擎以及我所遇到过的几乎所有的游标模型中，默认的游标类型都是只进、只读游标。与其他游标类型相比，这种游标的开销非常小，而且，由于它们枚举数据所能达到的速度，我们通常把这种游标称为“流水”游标。和消防水龙带模式一样，它知道如何只以一个方向来堆放数据(你无法使水龙带里的水倒流，不是吗？)。在多数情况下，流水游标只是比其他基于游标的选项要快，但不要误认为它的性能超过了集合操作。相比而言，即使流水游标也比大多数对等的集合操作要慢。

现在，让我们先来看一下关于游标的扩展语法，然后分别讨论所有这些选项：

```

DECLARE <cursor name> CURSOR
[LOCAL|GLOBAL]
[FORWARD_ONLY|SCROLL]
[STATIC|KEYSET|DYNAMIC|FAST_FORWARD]

```

```
[READ_ONLY|SCROLL_LOCKS|OPTIMISTIC]
[TYPE_WARNING]
FOR <SELECT statement>
[FOR UPDATE [OF <column name >[,...n]]][;]
```

另外, 还有一个能更好地支持 ANSI/ISO 的语法:

```
DECLARE <cursor name> [INSENSITIVE|SCROLL] CURSOR
FOR <SELECT statement>
[FOR [READ ONLY|UPDATE [OF <column name >[,...n]]][;]
```

提示:

ANSI/ISO 版本的语法是在游标添加到产品之后很久才添加的(游标首次在 20 世纪 90 年代中期发布的 SQL Server 6.0 中出现)。可能出于这个原因(以及前一版本所提供的更多功能),我在 SQL Server 中所看到的所有游标都运用了我所列出的第一个版本,而不是更具可移植性的 ANSI/ISO 版本。

乍一看,内容并不多,但在声明游标时,要考虑许多事情(正如前面讲过,要考虑的最重要的事情可能是这样的,“确实需要在游标中完成这一功能吗?”)。好的一面是,其中一些选项互相暗示的,因此,一旦做出了一个决定,其他选项也很快能够确定。

现在,我们将一步步地应用具体的语法,从而了解与之相关的重要概念。

13.3.1 作用域

LOCAL 与 GLOBAL 选项决定着游标的作用域,也就是哪些连接和进程可以“看到”游标。对于大多数有作用域的项目来说,其作用域默认为更保守的方式,即只有最小作用域(这种情况下将是 LOCAL)。不过,SQL Server 游标算是个例外——它的默认作用域是 GLOBAL。在进一步讨论 LOCAL 与 GLOBAL 作用域方面的问题之前,最好先暂时撇开这一话题,了解一下这里所说的局部和全局的含义。

我们已经讨论了游标的例外之处,即游标的默认作用域被设置为全局范围而非更保守的局部范围。但例外并不仅限于此。在 SQL Server 中,所谓全局的或局部的通常是指,它可被所有的连接看见而非仅限于当前连接。不过,对于游标声明的目的而言,它是指当前连接中所有的进程(批处理、触发器、存储过程)是否可看见它,或者只是当前进程可看见它。

图 13-1 说明了游标的作用域。

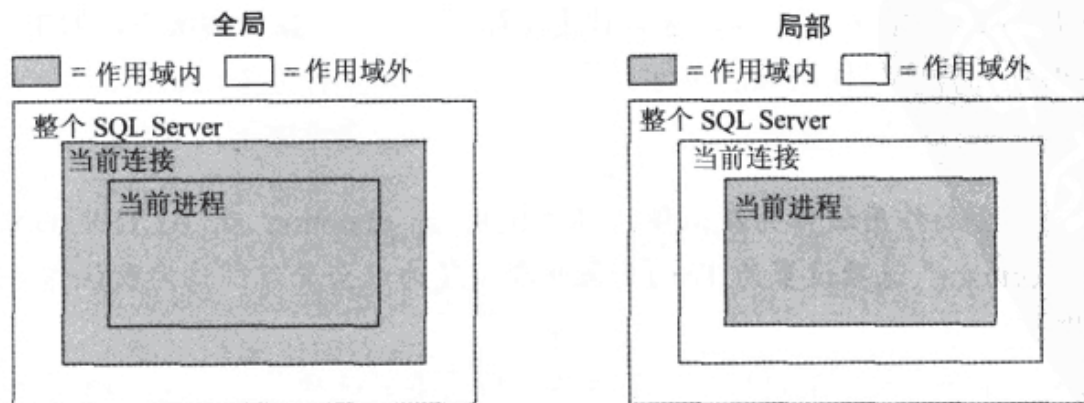


图 13-1

现在，考虑一下作用域的含义并进行测试。

如你所料，默认作用域为全局范围既有好的一面，也有不好的一面。作用域为全局意味着可以在一个存储过程中创建游标，然后从另一个不同的存储过程中引用它——不必传递对它的引用。但全局作用域不好的方面是，如果试图另一个同名的游标，则将得到一个错误。

我们可用一个简短的例子来试验。这里要做的是创建一个能为我们生成游标的存储过程：

```
USE AdventureWorks2008;
GO

CREATE PROCEDURE spCursorScope
AS

DECLARE @Counter      int,
        @OrderID      int,
        @CustomerID   int

DECLARE CursorTest CURSOR
GLOBAL
FOR
    SELECT SalesOrderID, CustomerID
    FROM Sales.SalesOrderHeader;

SELECT @Counter = 1;
OPEN CursorTest;
FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID;
PRINT 'Row ' + CAST(@Counter AS varchar) + ' has a SalesOrderID of ' +
    CONVERT(varchar,@OrderID) + ' and a CustomerID of ' + CAST(@CustomerID AS
varchar);

WHILE (@Counter<=5) AND (@@FETCH_STATUS=0)
BEGIN
    SELECT @Counter = @Counter + 1;
    FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID;
    PRINT 'Row ' + CAST(@Counter AS varchar) + ' has a SalesOrderID of ' +
        CONVERT(varchar,@OrderID) + ' and a CustomerID of ' + CAST(@CustomerID
AS varchar);
END
```

要注意该存储过程中的几件事情。首先，声明了用于存储的变量，以便为我们做一些事情。第一个变量@Counter 将只用来跟踪事情的进展，这样我们就只需在几条记录间移动，而不用在整个记录集上移动。当在结果集中逐行检索时，第二个和第三个变量@OrderID 和@CustomerID 分别用于保存从查询中检索到的值。

接着，声明了实际的游标。注意，这里显式设置了作用域。默认情况下，如果省略 GLOBAL 关键字，仍会得到拥有全局作用域的游标。

注意：

不一定非要以全局作用域作为默认值。可以使用 sp_dboption 或 ALTER DATABASE，将“Default to local cursor”选项设置为 True(如果想要恢复为以全局范围作为默认作用域，将该选项设置为 False 即可)。

这正好又是一个很好的、能说明总是显式声明所需要的选项是很有意义的例子。不要过分依赖默认值。设想一下，如果你依赖于默认的 GLOBAL，但后来某人在系统中更改了该选项！可能

很多人会说,“噢,没人会修改那个选项。”错!人们常常为了解决别处的某些问题作这种“很小的改动”。由于使用游标时的不明确性,你可能会在数周后遇到麻烦——到那时你已经完全忘记作的修改。

我们接下来打开游标,在数条记录中一步步地浏览。不过要注意,我们没有关闭或释放游标。在退出存储过程时,我们任由游标打开着并保持可用。

提示:

这我不禁想到老电影《迷失太空》,机器人不停地喊着“危险,Will Robinson!危险!”让游标胡乱地打开着将给你带来痛苦、挫折和极度的沮丧。

这里这样做是为了充分说明作用域的概念,但是对于这样的用法一定要非常小心。在调用存储过程时,如果没有意识到存储过程结束后不会将其自身作妥善处理,那将是有危险的。如果没有在存储过程外部把游标清理干净(关闭并释放),那么这种被丢弃的但仍然处于活动状态的游标会造成资源泄漏。另外,如果再次调用同一个存储过程(它会尝试再次声明并打开游标,但这个游标已经存在了),那将可能产生错误。

当我们声明游标用于输出时,如果想要允许外部与游标进行交互,将会发现大量更显而易见且更好的选择。

现在,我们已经枚举了数条记录并证实了存储过程正在运行,接着要退出该存储过程(记住,我们没有关闭或释放游标)。然后,从存储过程外部引用游标:

```
EXEC spCursorScope;

DECLARE @Counter      int,
        @OrderID      int,
        @CustomerID   int;

SET @Counter=6;

WHILE (@Counter<=10) AND (@@FETCH_STATUS=0)
BEGIN
    PRINT 'Row ' + CAST(@Counter AS varchar) + ' has a SalesOrderID of ' + CAST(@OrderID
    AS varchar) + ' and a CustomerID of ' + CAST(@CustomerID AS varchar);
    SELECT @Counter = @Counter + 1;
    FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID;
END

CLOSE CursorTest;
DEALLOCATE CursorTest;
```

让我们看一下这里都发生了什么。

首先,执行存储过程。正如我们已经看到的,该存储过程创建了游标,随后枚举了数行。然后退出存储过程,并任由游标保持打开。

接着声明变量,这些变量和存储过程中所声明的变量完全一样。为什么必须再次声明变量,但却不声明游标?因为只有游标被默认为是全局的。也就是说,当存储过程离开作用域时,变量就消失了。我们不能再继续引用这些变量,否则将得到变量未定义的错误。因此必须重新声明它们。

接下来的代码结构看起来与存储过程中的几乎相同。我们再次循环浏览以枚举出数条记录。

最后，在证实了游标在存储过程外部依然是活动的之后，我们准备关闭并释放游标。只有在关闭了游标后，才释放游标中使用的结果集所占用的内存或 tempdb 空间，并且，只有在释放了游标之后，才释放了游标变量及其查询定义所占用的内存空间。

现在，在系统中创建存储过程(如果还未创建)并执行脚本。得到的结果如下所示：

```
Row 1 has a SalesOrderID of 43659 and a CustomerID of 29825
Row 2 has a SalesOrderID of 43660 and a CustomerID of 29672
Row 3 has a SalesOrderID of 43661 and a CustomerID of 29734
Row 4 has a SalesOrderID of 43662 and a CustomerID of 29994
Row 5 has a SalesOrderID of 43663 and a CustomerID of 29565
Row 6 has a SalesOrderID of 43664 and a CustomerID of 29898
```

```
Row 7 has a SalesOrderID of 43665 and a CustomerID of 29580
Row 8 has a SalesOrderID of 43666 and a CustomerID of 30052
Row 9 has a SalesOrderID of 43667 and a CustomerID of 29974
Row 10 has a SalesOrderID of 43668 and a CustomerID of 29614
```

从中可以看出游标依然是打开的，存储过程外部的循环可以在存储过程内部的代码已经停止的地方继续拣选。

现在看看如果把存储过程中的游标的作用域声明为局部的，将发生什么情况：

```
USE AdventureWorks2008;
GO
```

```
ALTER PROCEDURE spCursorScope
```

```
AS
```

```
DECLARE @Counter      int,
        @OrderID      int,
        @CustomerID int;
```

```
DECLARE CursorTest CURSOR
```

```
LOCAL
```

```
FOR
```

```
    SELECT SalesOrderID, CustomerID
    FROM Sales.SalesOrderHeader;
```

```
SELECT @Counter - 1;
```

```
OPEN CursorTest;
```

```
FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID;
```

```
PRINT 'Row ' + CAST(@Counter AS varchar) + ' has a SalesOrderID of ' + CAST(@OrderID AS
    varchar) + ' and a CustomerID of ' + CAST(@CustomerID AS varchar);
```

```
WHILE (@Counter < 5) AND (@@FETCH_STATUS = 0)
```

```
BEGIN
```

```
    SELECT @Counter = @Counter + 1;
```

```
    FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID;
```

```
    PRINT 'Row ' + CAST(@Counter AS varchar) + ' has a SalesOrderID of ' + CAST(@OrderID
        AS varchar) + ' and a CustomerID of ' + CAST(@CustomerID
        AS varchar);
```

```
END
```

看起来似乎上面作的只是很小的改动，但是，当我们再次执行脚本时，会发现结果有很大不同：

```
Row 1 has a SalesOrderID of 43659 and a CustomerID of 29825
Row 2 has a SalesOrderID of 43660 and a CustomerID of 29672
Row 3 has a SalesOrderID of 43661 and a CustomerID of 29734
Row 4 has a SalesOrderID of 43662 and a CustomerID of 29994
Row 5 has a SalesOrderID of 43663 and a CustomerID of 29565
Row 6 has a SalesOrderID of 43664 and a CustomerID of 29898
```

```
Msg 16916, Level 16, State 1, Line 14
A cursor with the name 'CursorTest' does not exist.
Msg 16916, Level 16, State 1, Line 18
A cursor with the name 'CursorTest' does not exist.
Msg 16916, Level 16, State 1, Line 19
A cursor with the name 'CursorTest' does not exist.
```

在位于存储过程内时，一切都与以前一样。但当位于存储过程外部时，此时游标不再在其作用域内，因此无法再引用它，脚本运行中出现了数个错误。本章后面将介绍如何从创建有着局部作用域的游标的存储过程外部访问该游标。

从本节中应当明白的首要事情是，必须考虑游标的作用域。和其他使用 DECLARE 语句声明的事情相比，游标的作用方式不太一样。

13.3.2 可滚动性

和本章中讨论的大多数概念一样，可滚动性适用于几乎所有的游标模型。其概念实际非常简单：能够在任何方向导航还是限制为只能向前移动？默认为不可滚动的，只能向前移动。

1. 只进

“只进”游标表示只能向前移动。由于只进是默认方式，因此，在知道它是我们到现在为止唯一使用过的游标类型时，你可能也不会感到惊讶。在使用只进游标时，唯一有效的导航选项是 FETCH NEXT。在前往下一条记录之前，必须确保已经利用完每一条记录，因为一旦移至下一记录，则无法再回到之前访问过的记录上，除非关闭并重新打开游标。

2. 可滚动

“可滚动”游标的概念也是可从其字面上进行理解的。可在必要时向前和向后“滚动”游标。如果你使用某种 API(ODBC、OLE DB 和 JDBC 等)，那么根据当前正处理的对象模型，通常可以直接导航到指定的记录。事实上，通过 ADO、ADO.NET 和 LINQ，甚至可以对数据进行重新排序和添加额外的筛选器。

滚动的基础是 FETCH 关键字。可使用 FETCH 在游标中向前和向后滚动，也可以移动到指定的位置。FETCH 的主要参数如下：

- NEXT——移至下一条记录
- PRIOR——移至上一条记录
- FIRST——移至第一条记录
- LAST——移至最后一条记录

本章后面将更深入地探讨 FETCH，不过目前要认识到 FETCH 的存在并知道它是用来控制如何在游标集中进行导航的事物。

下面用一个简短的例子来说明可滚动游标的概念。实际上，这里使用的只是本章前面创建的

SQL Server 2008 高级程序设计

存储过程的一个变体。

```
USE AdventureWorks2008;
GO

CREATE PROCEDURE spCursorScroll
AS

    DECLARE @Counter      int,
            @OrderID      int,
            @CustomerID   int

    DECLARE CursorTest CURSOR
    LOCAL
    SCROLL
    FOR
        SELECT SalesOrderID, CustomerID
        FROM Sales.SalesOrderHeader;

    SELECT @Counter = 1;
    OPEN CursorTest;
    FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID;
    PRINT 'Row ' + CAST(@Counter AS varchar) + ' has a SalesOrderID of ' + CAST(@OrderID
        AS varchar) + ' and a CustomerID of ' + CAST(@CustomerID AS
        varchar);

    WHILE (@Counter<=5) AND (@@FETCH_STATUS=0)
    BEGIN
        SELECT @Counter = @Counter + 1;
        FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID;
        PRINT 'Row ' + CAST(@Counter AS varchar) + ' has a SalesOrderID of ' + CAST(@OrderID
            AS varchar) + ' and a CustomerID of ' + CAST(@CustomerID
            AS varchar);
    END

    WHILE (@Counter > 1) AND (@@FETCH_STATUS = 0)
    BEGIN
        SELECT @Counter = @Counter - 1;
        FETCH PRIOR FROM CursorTest INTO @OrderID, @CustomerID;
        PRINT 'Row ' + CONVERT(varchar,@Counter) + ' has an SalesOrderID of ' + CAST(@OrderID
        AS varchar) + ' and a CustomerID of ' + CAST(@CustomerID
        AS varchar);
    END

    CLOSE CursorTest;
    DEALLOCATE CursorTest;
```

明显的不同之处是：

- 游标是用 **SCROLL** 选项声明的；
- 在原先使用 **NEXT** 的地方增加了一个新的导航关键字——**PRIOR**；
- 这里在存储过程中关闭并释放了游标，而不是使用外部的过程。

结果中出现了有趣的事情。这里不需要特定的测试脚本。直接执行它：

```
EXEC spCursorScroll ;
```

将看到订单值前后滚动：

```
Row 1 has a SalesOrderID of 43659 and a CustomerID of 29825
```

```
Row 2 has a SalesOrderID of 43660 and a CustomerID of 29672
Row 3 has a SalesOrderID of 43661 and a CustomerID of 29734
Row 4 has a SalesOrderID of 43662 and a CustomerID of 29994
Row 5 has a SalesOrderID of 43663 and a CustomerID of 29565
Row 6 has a SalesOrderID of 43664 and a CustomerID of 29898

Row 5 has an SalesOrderID of 43663 and a CustomerID of 29565
Row 4 has an SalesOrderID of 43662 and a CustomerID of 29994
Row 3 has an SalesOrderID of 43661 and a CustomerID of 29734
Row 2 has an SalesOrderID of 43660 and a CustomerID of 29672
Row 1 has an SalesOrderID of 43659 and a CustomerID of 29825
```

正如你所看到的，我们不仅能像前面那样成功地向前导航游标，而且也可以向后导航。

只进游标显然是这两种选项中效率更高的那种。考虑一下开销：如果游标是只进的，则 SQL Server 只需要跟踪下一条记录——按照链表的方式。如果需要以其他方式重新部置游标，为了正确地找出需要的行，必须存储额外的信息。具体实现取决于所选择的游标类型。

一些游标类型是隐式可滚动的，而另一些则不是。一些类型的游标对于数据的修改是敏感的，而另一些则不是。下一节将对这些问题中的一部分进行讨论。

13.3.3 游标类型

通常，各种 API 将游标分为 4 种类型：

- 静态游标
- 键集驱动的游标
- 动态游标
- 只进游标

这 4 种类型如何实现(以及如何称谓)有时会随 API 和对象模型的不同而有略微变化，但它们的本质几乎是一样的。

这些游标类型不同之处在于，它们是否可滚动，以及它们对于游标生命周期中发生在数据库上的更改是否是敏感的。我们已经知道了可滚动的概念，但“敏感”一词听上去更像是畅销书《男人来自火星，女人来自金星》中的内容，而非有关编程的书中的词语。尽管如此，在选择游标类型时，敏感性还是要考虑的一个相当重要的概念。

游标是否敏感是指在游标被打开后，它是否能感知到发生在数据库中的修改。另外，这也定义了发现发生了改变后应该如何做。让我们看两个有关这方面的终极版本——静态游标与动态游标。静态游标一旦创建，就绝对不会注意到对数据库的任何更改。而动态游标只要处于打开状态，就能意识到对数据库所做的每一个修改(插入记录、删除、更新等)。我们将在讨论每一种游标类型时探讨有关敏感性的问题。

1. 静态游标

静态游标表示了数据的一个“快照”。实际上，至少有一个数据访问对象模型把它称为快照记录集，而非静态记录集。

在创建静态游标时，整个记录集在 tempdb 的临时表中创建。在创建了静态游标后，任何人或任何事都不能够修改静态游标。也就是说，它是固定不变的。一些对象模型允许你更新静态游标中的信息，而另一些则不允许，但有一点是相同的：不能通过静态游标将更新写入到数据库中。

在更深入地研究这种游标类型前，我要说明一下，在服务器端需要使用静态游标的情形并不多见。我并不是说这种情况不存在。确实有，只是这样的情形非常少见。

如果打算在服务器端使用静态游标，那可在之前问自己如下问题：

- 可以用临时表来实现吗？
- 可以完全在客户端实现吗？

记住，静态游标被 SQL Server 保存在 tempdb 里的私有表中。如果 SQL Server 无论如何都要使用临时表，那为什么不只是使用临时表呢？有时，临时表不能满足需要(记录操作而非行集操作)。不过，如果只想要数据的一个快照，而并不想实现基于记录的操作，那可以使用 SELECT INTO 或 CREATE TABLE 和 INSERT INTO 创建你自己的临时表，这可为你(和 SQL Server)节省大量开销。

如果采用的是客户端-服务器架构，通常在客户端处理静态游标会更好些。通过把整个操作移到客户端，可极大地减少与服务器通信的网络往返的次数。由于已经知道对数据库的修改不会影响到游标(毕竟，这就是首先选择静态游标的原因)，因此，在游标已创建好之后，没有理由再让它与服务器进行联系。

再来看一个静态游标的例子。本例将涉及创建静态游标的概念，然后进行修改并查看会发生什么情况。在本章这一部分剩余内容的讲述过程中，当讨论到每一种游标类型时，都会用到该示例的变种。

首先创建一个用来测试的表，然后创建游标并操纵它以了解其中有什么。

```
USE AdventureWorks2008;
/* Build the table that we'll be playing with this time */
SELECT SalesOrderID, CustomerID
INTO CursorTable
FROM Sales.SalesOrderHeader
WHERE SalesOrderID BETWEEN 43661 AND 43665;

-- Declare our cursor
DECLARE CursorTest CURSOR
GLOBAL          -- So we can manipulate it outside the batch
SCROLL         -- So we can scroll back and see the changes
STATIC         -- This is what we're testing this time
FOR
SELECT SalesOrderID, CustomerID
FROM CursorTable;

-- Declare our two holding variables
DECLARE @SalesOrderID int;
DECLARE @CustomerID varchar(5);

-- Get the cursor open and the first record fetched
OPEN CursorTest;
FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID;

-- Now loop through them all
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT CAST(@SalesOrderID AS varchar) + ' ' + @CustomerID;
    FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID;
END

-- Make a change. We'll see in a bit that this won't affect the cursor.
```

```

UPDATE CursorTable
  SET CustomerID = -111
  WHERE SalesOrderID = 43663;

-- Now look at the table to show that the update is really there.
SELECT SalesOrderID, CustomerID
FROM CursorTable;

-- Now go back to the top. We can do this since we have a scrollable cursor
FETCH FIRST FROM CursorTest INTO @SalesOrderID, @CustomerID;

-- And loop through again.
WHILE @@FETCH_STATUS=0
BEGIN
  PRINT CONVERT(varchar(5),@SalesOrderID) + ' ' + @CustomerID;
  FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID;
END

-- Now it's time to clean up after ourselves
CLOSE CursorTest;

DEALLOCATE CursorTest;

DROP TABLE CursorTable;

```

让我们看一下从以上代码得到了什么(注意, 为了能更方便地将结果集与 PRINT 消息显示在一起, 我把显示结果的方式切换到了“以文本格式显示结果”选项):

```

(5 row(s) affected)
43661  29734
43662  29994
43663  29565
43664  29898
43665  29580

(1 row(s) affected)
SalesOrderID CustomerID
-----
43661        29734
43662        29994
43663        -111
43664        29898
43665        29580

(5 row(s) affected)

43661 29734
43662 29994
43663 29565
43664 29898
43665 29580

```

对于运行脚本的过程中发生的情况, 需要注意以下几件事情:

- 第一, 尽管对于表来说有结果集打开着, 我们仍能够执行更新。在这种情况下, 是因为我们使用的是静态游标。一旦创建了游标, 它就与实际记录相分离并不再维持任何锁。

- 第二, 虽然可看到确实在实际的表上发生了更新, 但更新不会影响游标中的数据。同样, 这是因为一旦创建了游标, 游标就是独立的, 不再以任何方式与原始数据相关联。
- 第三, 你可能已经注意到, 我们对 FETCH 关键字使用了一个新的参数。这次通过使用 FETCH FIRST, 我们返回到了结果集的顶端。

2. 键集驱动的游标

当和游标一起谈论键集时, 我们并不是在讨论你们当地的锁匠。事实上, 我们讨论的是对一组数据的维护, 而这组数据在数据库中唯一标识一整行。

键集驱动的游标有下列一些要点:

- 它们要求表上存在唯一索引;
- 只有键集存储在 tempdb 中, 而非整个数据集;
- 它们能感知到对已是键集一部分的行所做的修改, 包括这些行已被删除的可能性;
- 它们不能感知到在创建了游标之后添加的新行;
- 键集游标可作为将对数据执行更新的游标的基础。

现在, 我们已经知道了其名称为“键集”, 并且我也已说过, 键集唯一标识每一行, 那么, 那么你可能应该猜到: 必须要有某种形式的唯一索引(通常是主键, 但也可以是任何显式定义为唯一的索引)来创建键集。

键都存储在 tempdb 的私有表中。如果想要在游标中寻找特定行, SQL Server 使用键作为找回数据的方法。要注意的一点是, 在发出 FETCH 时, 是基于键提取了实际的数据。这样做的优点是, 当提取特定行时, 该行中的数据是最新的。键集驱动的游标的缺点(或优点, 这取决于想要用游标来做什么)是, 它使用已创建的键集进行查找。这意味着一旦创建了键集, 所有的行都包含在了游标中。在创建了游标之后添加的任何行都不会被游标看见(即使这些添加的行满足 SELECT 语句中 WHERE 子句的条件, 也是如此)。根据所选择的游标选项, 可以通过游标操作更新已经是游标一部分的行。

我们将修改一下前面的脚本, 以说明在使用键集驱动的游标时的敏感性问题:

```
USE AdventureWorks2008;
/* Build the table that we'll be playing with this time */
SELECT SalesOrderID, CustomerID
INTO CursorTable
FROM Sales.SalesOrderHeader
WHERE SalesOrderID BETWEEN 43661 AND 43665;

-- Now create a unique index on it in the form of a primary key
ALTER TABLE CursorTable
    ADD CONSTRAINT PKCursor
    PRIMARY KEY (SalesOrderID);

/* The IDENTITY property was automatically brought over when
** we did our SELECT INTO, but I want to use my own SalesOrderID
** value, so I'm going to turn IDENTITY_INSERT on so that I
** can override the identity value.
*/
SET IDENTITY_INSERT CursorTable ON;

-- Declare our cursor
DECLARE CursorTest CURSOR
```



```

GLOBAL          -- So we can manipulate it outside the batch
SCROLL          -- So we can scroll back and see the changes
KEYSET          -- This is what we're testing this time
FOR
SELECT SalesOrderID, CustomerID
FROM CursorTable;

-- Declare our two holding variables
DECLARE @SalesOrderID int;
DECLARE @CustomerID varchar(5);

-- Get the cursor open and the first record fetched
OPEN CursorTest;
FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID;

-- Now loop through them all
WHILE @@FETCH_STATUS < 0
BEGIN
    PRINT CAST(@SalesOrderID AS varchar) + ' ' + @CustomerID;
    FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID;
END

-- Make a change. We'll see that does affect the cursor this time.
UPDATE CursorTable
    SET CustomerID = -111
    WHERE SalesOrderID = 43663;

-- Now we'll delete a record so we can see how to deal with that
DELETE CursorTable
    WHERE SalesOrderID = 43664;

-- Now Insert a record. We'll see that the cursor is oblivious to it.
INSERT INTO CursorTable
    (SalesOrderID, CustomerID)
VALUES
    (-99999, -99999);

-- Now look at the table to show that the changes are really there.
SELECT SalesOrderID, CustomerID
FROM CursorTable;

-- Now go back to the top. We can do this since we have a scrollable cursor
FETCH FIRST FROM CursorTest INTO @SalesOrderID, @CustomerID;

/* And loop through again.
** This time, notice that we changed what we're testing for.
** Since we have the possibility of rows being missing (deleted)
** before we get to the end of the actual cursor, we need to do
** a little bit more refined testing of the status of the cursor.
*/
WHILE @@FETCH_STATUS < -1
BEGIN
    IF @@FETCH_STATUS = -2
    BEGIN
        PRINT ' MISSING! It probably was deleted.';
    END
    ELSE
    BEGIN
        PRINT CAST(@SalesOrderID AS varchar) + ' ' + CAST(@CustomerID AS varchar);
    END
END

```



```

    FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID;
END

```

```

-- Now it's time to clean up after ourselves
CLOSE CursorTest;

```

```

DEALLOCATE CursorTest;

```

```

DROP TABLE CursorTable;

```

这里作的修改并不十分明显。我们添加了必需的唯一索引。这里选择了主键，因为对于我们要提取信息的表来说，它最为合适，但也可以使用并非主键的唯一索引。另外，还添加了一些内容来插入数据行，这样就可以清楚地观察到键集看不见该数据行。

在所做的修改中，可能最重要的事情是改变了最后在游标上运行的 WHILE 循环的条件。从技术上讲，应该对两个循环都作这样的修改，但对于本例中第一次进行的循环来说，不存在删除记录的风险，而且我也希望在同一个脚本中显示出一些差异。

进行这样的修改是为了处理新加入的一些事情——可能会发现一条记录没有了。这很可能是有人删除了它。

运行该脚本后得到的结果如下所示：

```

(5 row(s) affected)
43661 29734
43662 29994
43663 29565
43664 29898
43665 29580

(1 row(s) affected)

(1 row(s) affected)

(1 row(s) affected)
SalesOrderID CustomerID
-----
-99999      -99999
43661       29734
43662       29994
43663       -111
43665       29580

(5 row(s) affected)

43661 29734
43662 29994
43663 -111
MISSING! It probably was deleted.
43665 29580

```

让我们看一下这里的一些重点。

刚开始，一切都与以前几乎相同。和上一次一样，在第一个结果集中有同样的 5 行。然后，看到了几条额外的“affected by”(受影响)消息。这是因为我们添加了 INSERT、UPDATE 和 DELETE 语句。接着是第二个结果集。从这里开始，事情变得有些有趣了。

在接下来的结果集中，我们看到了 UPDATE、INSERT 和 DELETE 语句的实际结果。正如我

们认为已经完成的那样, SalesOrderID 43664 已经被删除, 并且插入了 SalesOrderID 为-99999 的新记录。这正是表中的数据, 但在游标中, 事情并没有这么轻松。

接下来的(和最后的)结果集表明了呈现在游标中的事情与重新运行查询的一些不同之处。这里碰巧得到了 5 条记录——就像我们一开始运行 SELECT 语句显示出的实际表中的记录一样。但是, 那完全是巧合。

事实上, 游标显示的记录和表显示的记录之间有几个关键的不同点。第一点很明显。结果集实际知道有一条记录不见了。我们可以看到, 游标继续显示键集中键的位置, 但当它去查找数据时, 数据已经不在那里了。@@FETCH_STATUS 被设置为-2, 我们可以测试它并进行报告。SELECT 语句显示实际在那里的数据, 对于曾经存在的数据没有任何印象。另一方面, INSERT 对于游标来讲是未知的。在创建游标时记录不在那里, 因此游标不知道它的存在。它不会显示在我们的结果集中。

如果需要在一定程度上感知到对数据的修改, 但不必了解最新发生的所有插入, 可以使用键集游标。根据要找的结果集的性质以及键集的性质, 键集游标可以大大减少必须复制和存储到 tempdb 中的数据量。这可对整个服务器的性能产生有利的影响。

注意:

警告!!! 如果是在没有唯一索引的表上定义类型为 KEYSET 的游标, 那么 SQL Server 将把游标隐式地转换为 STATIC 类型。行为上发生改变这一事实可能会让你有些不安, 但是情况还不仅限于此。它并不会告知你发生的事情。在默认情况下, 对于这一转换, 你不会得到任何警告。幸运的是, 通过在游标中使用 TYPE_WARNING 选项, 可以注意到这类事情。本章后面将对这一选项作简单讲述。

3. 动态游标

动态游标有什么特别之处? 你可能会说“它们是动态的”。你应该很希望自己在智力竞赛节目中让人们回答这样一个问题“动态游标有什么特别之处”吧? 然后, 可以料想的是抢着回答这一问题的人很少, 但参与的人会立刻说“它是动态的, 对吗?”。

这一答案基本正确。因为动态游标不会主动地告知你对于底层数据的修改, 所以它们还称不上是动态的。但它们对于底层数据上的所有修改是敏感的, 所以它们非常接近动态。当然, 和生活中的大多数事情一样, 有得就必有失。

如果希望把插入的记录添加到游标中, 这没有问题。如果希望更新的记录在游标中也能够完全更新, 这也没有问题。如果希望删除的记录从游标的数据集中删除, 这仍没有问题(但由于不会像在键集游标类型中那样看到缺少的记录, 因此实际无法说出删除了什么东西)。不过, 如果想要拥有并发性, 就会遇到问题(因为让数据集更长时间地打开着, 所以可能与其他用户发生冲突)。如果想要拥有较低的开销, 这会有很大的问题(实际上, 每个 FETCH 都会重新进行查询)。是的, 动态游标会严重损害性能, 但生活就是这样的。

总之, 通常应避免使用动态游标。

那为什么还要这样作大肆宣传? 为了理解动态游标可能产生的影响, 必须对其工作方式有一些了解。可以看到, 使用动态游标时, 每发出一次 FETCH, 都要重建游标。是的, 构成查询基础的 SELECT 语句以及与之相关联的 WHERE 子句会再次运行。考虑一下处理大型数据集的情况。在此情况下, 你只会想到一个词——糟糕。确实非常糟糕。

提示:

从接触到 RDBMS 开始, 我就知道动态游标是非常耗性能的——但我发现并不总是这样, 特别是当底层表不是很大时更是如此。如果稍加思考, 你可能就会明白为什么动态游标在原始速度上实际可以稍快一些。

有关于其原因, 我猜是因为为键集游标使用了 tempdb。尽管为了处理动态游标, 对于每个 FETCH 都要做更多的工作, 但用于再次查询的数据常常是完全位于缓存中(这取决于系统的大小和加载情况)。这意味着动态游标的大部分工作来自于 RAM。另一方面, 键集游标存储在 tempdb 中, 对于大多数系统来说, tempdb 位于硬盘上(也就是说, 速度会慢很多)。

随着表的大小变大, 服务器上会有更多不同的流量, 分配给 SQL Server 的内存会变小, 而键集驱动的游标就将比动态游标越有优势。另外, 原始速度并非一切, 实际还必须考虑并发性问题(本章后面将更详细地研究用于并发性的选项), 而并发性在动态游标中很成问题。如果处理的是具有较小数据集的服务器端游标, 那在考虑动态游标时也不要只考虑速度。

接下来, 对最后一个脚本只作一个修改——即把 KEYSET 改为 DYNAMIC, 然后重新运行它:

```
USE AdventureWorks2008;
/* Build the table that we'll be playing with this time */
SELECT SalesOrderID, CustomerID
INTO CursorTable
FROM Sales.SalesOrderHeader
WHERE SalesOrderID BETWEEN 43661 AND 43665;

-- Now create a unique index on it in the form of a primary key
ALTER TABLE CursorTable
    ADD CONSTRAINT PKCursor
    PRIMARY KEY (SalesOrderID);

/* The IDENTITY property was automatically brought over when
** we did our SELECT INTO, but I want to use my own SalesOrderID
** value, so I'm going to turn IDENTITY_INSERT on so that I
** can override the identity value.
*/
SET IDENTITY_INSERT CursorTable ON;

-- Declare our cursor
DECLARE CursorTest CURSOR
GLOBAL          -- So we can manipulate it outside the batch
SCROLL          -- So we can scroll back and see the changes
DYNAMIC         -- This is what we're testing this time
FOR
SELECT SalesOrderID, CustomerID
FROM CursorTable;

-- Declare our two holding variables
DECLARE @SalesOrderID int;
DECLARE @CustomerID varchar(5);

-- Get the cursor open and the first record fetched
OPEN CursorTest;
FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID;

-- Now loop through them all
WHILE @@FETCH_STATUS = 0
BEGIN
```

```
PRINT CAST(@SalesOrderID AS varchar) ' ' @CustomerID;
FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID;
END

-- Make a change. We'll see that does affect the cursor this time.
UPDATE CursorTable
    SET CustomerID --111
    WHERE SalesOrderID = 43663;

-- Now we'll delete a record so we can see how to deal with that
DELETE CursorTable
    WHERE SalesOrderID = 43664;

-- Now Insert a record. We'll see that the cursor is oblivious to it.
INSERT INTO CursorTable
    (SalesOrderID, CustomerID)
VALUES
    (-99999, -99999);

-- Now look at the table to show that the changes are really there.
SELECT SalesOrderID, CustomerID
FROM CursorTable;

-- Now go back to the top. We can do this since we have a scrollable cursor
FETCH FIRST FROM CursorTest INTO @SalesOrderID, @CustomerID;

/* And loop through again.
** This time, notice that we changed what we're testing for.
** Since we have the possibility of rows being missing (deleted)
** before we get to the end of the actual cursor, we need to do
** a little bit more refined testing of the status of the cursor.
*/
WHILE @@FETCH_STATUS != -1
BEGIN
    IF @@FETCH_STATUS = -2
    BEGIN
        PRINT ' MISSING! It probably was deleted.';
    END
    ELSE
    BEGIN
        PRINT CAST(@SalesOrderID AS varchar) ' ' CAST(@CustomerID AS varchar);
    END
    FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID;
END

-- Now it's time to clean up after ourselves
CLOSE CursorTest;

DEALLOCATE CursorTest;

DROP TABLE CursorTable;
```

结果如下所示:

```
(5 row(s) affected)
43661 29734
43662 29994
43663 29565
43664 29898
43665 29580
```



```

(1 row(s) affected)

(1 row(s) affected)

(1 row(s) affected)
SalesOrderID CustomerID
-----
-99999          -99999
43661           29734
43662           29994
43663           -111
43665           29580

(5 row(s) affected)
-99999 *
43661   29734
43662   29994
43663   -111
43665   29580

```

前两个记录集和上一次得到的完全一样。而当查看第三个(也是最后一个)结果集时,就有了改变:

- 尽管我们删除了一条记录,但是对于失败的提取没有任何指示(没有通知);
- 被更新的记录显示出了更新(就和使用键集时一样);
- 插入的记录现在显示在游标集中。

动态游标是所有游标中最敏感的。对底层数据做的所有事情都会影响到动态游标。动态游标的缺点是它们会带来额外的并发性问题,并且在处理较大的数据集时会给系统带来沉重的负担。

提示:

从技术上讲,与键集游标不同的是,动态游标可以运行于没有唯一索引的表中。要不惜一切代价避免这种情况(依我看来,应该防止这样做并抛出错误)。在特定情况下,由于动态游标不能追踪它在游标集中的位置,它很可能导致死循环。避免这种情况的最万无一失的方法是,不使用动态游标或只在真正有唯一索引的表上使用动态游标。

4. 快进(FAST_FORWARD)游标

快进游标中的“快”(从游标的观点来看——查询使所有游标像蜗牛一样慢)是最重要的一个词。快进游标是通常对只进游标使用的“流水游标”的典型。我总是用这个比拟来暗示数据向前涌出的方式。即一旦取出,就无法再将它放回。简而言之,数据像潮水一样涌出。使用FAST_FORWARD游标时,只是打开游标,然后处理数据而不做其他任何事情,接着向前移动,最后释放游标(注意,我没有说关闭游标)。

现在可以说,把这种类型称为一种游标“类型”有用词不当之嫌。虽然在几种不同的情况下,这种游标类型会自动转换成其他的游标类型,但是由于这种游标中的成员是固定的,因此我认为它们更像是一种键集驱动的游标。一旦确定了游标的成员,就不会添加新记录。被删除的行将显示为缺失的记录(@@FETCH_STATUS为-2)。要记住,尽管如此,如果游标被转换为其他的类型(通过自动转换),它将表现出新游标类型的行为。

注意：

这里比较讨厌的地方是，如果没有在游标的定义中添加 `TYPE_WARNING` 选项，SQL Server 不会告知你发生了转换。

正如前面所说，在许多情况下，`FAST_FORWARD` 游标会隐式转换为其他游标类型。表 13-1 中列出了这些转换。

表 13-1

条 件	转 换 为
底层查询要求创建临时表	静态游标
底层查询本质上是分布式的	键集游标
游标被声明为 <code>FOR UPDATE</code>	动态游标
存在一个将转换为键集驱动的条件，但是至少一个底层表上没有唯一索引	静态游标

我听说在其他一些情况下也会发生游标转换，但是我没有见到过任何相关文档，并且我自己没有遇到过这种情况。

注意：

如果你发现遇到了计算机相关领域中的最恐怖的情形(不明确的结果)，那么可以使用 `sp_describe_cursor`(一个系统存储过程)来列出游标当前的所有激活选项。

值得注意的是，所有 `FAST_FORWARD` 游标本质上是只读的。你可以显式地将这种游标设置为具有 `FOR UPDATE` 选项，但是正如前面的隐式转换表中所示，这种游标将被隐式地转换为动态的。

那么，与声明为 `FORWARD_ONLY` 的游标相比，`FAST_FORWARD` 游标到底有什么不同之处呢？`FAST_FORWARD` 游标至少有以下两个绝招中的一个：

- 第一个是预提取数据。也就是说，在打开游标的同时，将自动提取第一行。这意味着如果是在使用 ODBC 的客户端-服务器环境中，这样可省掉一次到服务器的往返。遗憾的是，这只在 ODBC 下可用。
- 第二个是确信无疑的——自动关闭游标。由于运行的是只进游标，因此，一旦到达记录集的末尾，SQL Server 会假定你想要关闭游标。同样，这节省了一次往返并获得了一点额外的性能。

在构建游标时，选择游标类型是最至关重要的决策之一。在游标任务的实际输出中出现的一点点差别的选择会导致性能大相径庭。其他影响涉及对修改的敏感性、并发性问题以及可更新性。

13.3.4 并发性选项

在有关事务和锁的一章中，我们初次接触了并发性问题。我们知道，当两个或更多的进程试图同时访问同样的数据时，要处理并发性问题。不过，在处理游标时，问题就变得更加麻烦。

问题是多方面的：

- 操作往往持续更长时间(有更多的时间发生并发性问题);
- 提取数据时会读取每一行, 但有人试图在你进行更新前编辑它;
- 你可以在结果集中向前或向后滚动, 而对于这样的操作没有时间限制(我希望你不要这样做, 但这样做是可以的)。

和所有的并发性问题一样, 这种问题往往更容易出现在事务环境中, 而非运行单条语句时。事务的持续时间越长, 就越容易出现并发性问题。

SQL Server 提供了 3 种处理这一问题的选项:

- READ_ONLY;
- SCROLL_LOCKS(在大多数术语中, 等同于 Pessimistic);
- OPTIMISTIC。

每种选项都有其特别之处, 因此下面对它们逐一进行讨论。

1. READ_ONLY

在只读的情况下, 不用担心游标是否将试图获取任何类型的更新锁或排他锁。同样, 在你忙于修改自己的数据时, 也不必担心是否有其他人编辑了数据。这两者使得事情变得相当简单。

READ_ONLY 选项义如其名。如果选择该选项, 不能更新任何数据, 但是也完全跳过了大多数(但不是全部)并发性问题。

2. SCROLL_LOCKS

在各种 API 和对象模型中, 滚动锁通常地等同于悲观锁定。其最简单的形式意味着, 只要你在编辑记录, 就不允许任何人对其进行编辑。这期间具体的实现细节会因下面两个方面而不同:

- 是否位于事务中;
- 设置的事务隔离级别是什么。

注意, 这与我们在锁和事务一章中看到的有所不同。

使用更新锁, 可以防止其他用户更新数据。在事务的持续期间都会持有该锁。如果事务是一个单语句事务, 那么只有在所有受更新影响的行都完成后, 才会释放锁。

滚动锁和更新锁几乎一样, 只有一个明显的不同之处——持有锁的持续时间不同。使用滚动锁时, 根据游标是否参与多语句事务, 情况会有很大的不同。假定不在事务中, 则只在游标中当前的记录上持有锁——也就是说, 从首次提取该记录起直至提取下一条记录(或者到达结果集的末尾)为止。一旦移至下一条记录, 前一条记录上的锁将被移除。

接下来通过在本章中大部分地方使用的脚本的精简版本予以演示:

```
USE AdventureWorks2008;
/* Build the table that we'll be playing with this time */
SELECT SalesOrderID, CustomerID
INTO CursorTable
FROM Sales.SalesOrderHeader
WHERE SalesOrderID BETWEEN 43661 AND 43665;

-- Now create a unique index on it in the form of a primary key
ALTER TABLE CursorTable
ADD CONSTRAINT PKCursor
PRIMARY KEY (SalesOrderID);

/* The IDENTITY property was automatically brought over when
```

```

** we did our SELECT INTO, but I want to use my own SalesOrderID
** value, so I'm going to turn IDENTITY_INSERT on so that I
** can override the identity value.
*/
SET IDENTITY_INSERT CursorTable ON;

-- Declare our cursor
DECLARE CursorTest CURSOR
GLOBAL          -- So we can manipulate it outside the batch
SCROLL         -- So we can scroll back and see the changes
DYNAMIC        -- This is what we're testing this time
SCROLL_LOCKS
FOR
SELECT SalesOrderID, CustomerID
FROM CursorTable;

-- Declare our two holding variables
DECLARE @SalesOrderID int;
DECLARE @CustomerID varchar(5);

-- Get the cursor open and the first record fetched
OPEN CursorTest;
FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID;

```

由于在前面的代码块中只添加了一行，因此不会看到很多灰色(以显示在该行上作了修改)。其余的改动是对代码行作了删除，因此，对于这些删除的代码，我无法把它们显示为灰色作为提醒。如果你尝试运行这些代码，只需确保作了适当的修改即可。

这里所作的修改是去掉了大部分要发生的事情，而再次把注意力集中在游标上。但可能最需要注意的是那些被我们故意忽略掉的几件事情，如果在没有它们的情况下进行操作，常会带来问题：

- 在游标上没有 CLOSE，这时也没有予以释放；
- 在提取了第一行之后，甚至没有再进一步进行滚动。

之所以将游标打开着是为了创造这样一个情形：游标保持打开的时间足够长，从而可以探讨有关锁的问题。另外，只提取第一行是为了确保当前有一个活动的行(这样能保证，在开始运行其他可能冲突的语句之前，不会到达数据集的末尾)。

我们想要做的是执行前面的代码，然后在 AdventureWorks 2008 活动的情况下打开一个完全不同的连接窗口。接着在新的连接窗口运行一个简单的测试：

```
SELECT * FROM CursorTable;
```

如果你还没有理解本节讲述的内容，可能会对下面的结果有些惊奇：

```

SalesOrderID CustomerID
-----
43661         29734
43662         29994
43663         29565
43664         29898
43665         29580

(5 row(s) affected)

```


根据已经了解的有关锁的知识(第 11 章),你可能会认为前面的 SELECT 语句将被当前记录上的锁阻塞。但是,使用滚动锁时不会这样。锁只存在于当前在游标中的记录上,并且,或许更重要的是,锁只会阻止对记录的更新。任何 SELECT 语句(如上面的语句)都可以毫无问题地看到游标中的内容。

现在,我们已经看到滚动锁是如何工作的,接下来回到原来的窗口并运行代码进行清理。这就回到了本章中大部分地方使用的相同的代码上:

```
-- Now it's time to clean up after ourselves
CLOSE CursorTest;

DEALLOCATE CursorTest;

DROP TABLE CursorTable;
```

提示:

不要忘记运行上面的清理代码!如果忘记了,那么在终止连接前,在系统中将会有有一个打开的事务。当连接断开时,SQL Server 会清理任何打开的事务(通过回滚这些事务),但是,我曾看到过这样的情形:在运行数据库一致性检查器(DBCC)时,发现有一些非常旧的事务。SQL Server 遗漏了这些事务的清理。

3. OPTIMISTIC

乐观锁定所创造的情形是这样的:在游标上没有设置任何类型的滚动锁。这里假设的是,在进行更新时,你仍然希望人们能够获取数据。之所以说这种方式是乐观的,是因为从你把数据提取到游标中到应用你的更新时,你认为(或许说“希望”)在这之间没有任何人会编辑你的数据。

乐观不一定就不行。如果你有大量的记录且没有很多用户,那么,两个人试图同时编辑同样的记录的几率会非常小(取决于业务流程的类型)。不过,如果选择了乐观,也必须为你错误的判断做好准备——也就是说,在你执行提取和对数据库进行实际的更新之间,可能有人修改了数据。

如果遇到了这样的问题,SQL Server 将发出一个错误,其中@@ERROR 的值为 16394。当发生这种情况时,必须完全从游标中重新提取数据(这样才能知道作了什么更改),并且回滚事务或是尝试再次进行更新。

13.3.5 检测游标类型转换: TYPE_WARNING

这非常简单。如果在游标中添加了该选项,那当在游标上发生了任何隐式转换时,你将会得到通知。如果不使用该语句,则对于转换不作任何通知。如果转换非预期的行为,那么很可能会看到在计算机领域中最令人担心的事情(不可预知的结果)。

或许通过一个例子可以更好地帮助理解,因此,我们对在本章中大部分地方使用的游标再作些修改并运行它。

在这个例子中,我们将去掉用来为表创建键的代码。前面讲过,在没有唯一索引的表上,键集将被隐式地转换为静态游标:

```
USE AdventureWorks2008;

/* Build the table that we'll be playing with this time */
SELECT SalesOrderID, CustomerID
```

```
INTO CursorTable
FROM Sales.SalesOrderHeader
WHERE SalesOrderID BETWEEN 43661 AND 43665;

-- Declare our cursor
DECLARE CursorTest CURSOR
GLOBAL                -- So we can manipulate it outside the batch
SCROLL                -- So we can scroll back and see the changes
KEYSET
TYPE_WARNING
FOR
SELECT SalesOrderID, CustomerID
FROM CursorTable;

-- Declare our two holding variables
DECLARE @SalesOrderID int;
DECLARE @CustomerID varchar(5);

-- Get the cursor open and the first record fetched
OPEN CursorTest;
FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID;

-- Now loop through them all
WHILE @@FETCH_STATUS=0
BEGIN
    PRINT CAST(@SalesOrderID AS varchar) + ' ' + CAST(@CustomerID AS varchar);
    FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID;
END

-- Now it's time to clean up after ourselves
CLOSE CursorTest;

DEALLOCATE CursorTest;

DROP TABLE CursorTable;
```

这里并没有非常特殊之处。我认为这完全是重写，因为我们从原先的代码中删除了许多东西。表和游标的创建与在本章很早之前讲述键集游标时所做的事情几乎一样。主要的修改是删除了本例中不需要的代码，并在游标的声明中添加了 `TYPE_WARNING` 选项。

现在，我们得到了一些有趣的结果：

```
(5 row(s) affected)
The created cursor is not of the requested type.
43661    29734
43662    29994
43663    29565
43664    29898
43665    29580
```

所有一切都正常运行。我们只看到了一个完全作为警告的语句。如果转换了游标类型，那么结果可能并非你所希望的。

提示：

这里的缺点是，只发送了一个消息，而没有任何错误。从编程上讲，实际上没有办法告知你收到了该消息——这使得该选项在生产环境中用处不大。不过，当试图调试游标以确定为什么它

没有按照希望的方式运作时，该选项还是非常便利的。

13.3.6 FOR <SELECT>

游标声明中的这部分是最核心的部分。即使在最基本的游标语法中，这部分也是必需的，因为它是确定应把什么数据放入游标中的唯一子句。

几乎所有的 SELECT 语句都是有效的——即使是包含 ORDER BY 子句的 SELECT 语句也是如此。只要 SELECT 语句提供单一的结果集，就没什么问题。可能出现问题的选项的例子是任何汇总选项，如 CUBE 或 ROLLUP 等。

13.3.7 FOR UPDATE

默认情况下，任何可更新的游标是完全可更新的——也就是如果能编辑一个列，那么所有的列都可被编辑。

FOR UPDATE <column list> 选项允许指定游标中只有特定的列是可被编辑的。如果包含了该选项，则只允许更新在列列表中的列。任何没有明确提及的列都被认为是只读的。

13.4 在游标中导航：FETCH 语句

我猜想，第一个创建 SQL 游标语法的人肯定非常喜欢狗。他们可能把要寻找的数据想成是骨头，而 SQL Server 是忠实的警犬。我猜也是从这里开始，就有了 FETCH 关键字。

如果思考一下，你就会觉得这是很贴切的术语。简单地说，它告诉 SQL Server “去取过来！”在收到这一指令后，忠实的狗(以 SQL Server 的形式)就会去找到我们要取得的特定的骨头(行)。在本章前面介绍的一些游标中，我们已经对 FETCH 语句有了一点接触，不过，现在要更深入地研究这一非常重要的语句。

实际上，FETCH 的选项比我们至今所看到的要多得多。到目前为止，我们见过 3 种不同的 FETCH 选项(NEXT、PREVIOUS 和 FIRST)。这还算是个不错的开始。的确，我们只需要向最基本的游标导航命令组中再加入一个命令即可，并且之后，再加入几个命令就拥有了完整的命令集。

表 13-2 中列出了每一个游标导航命令及其作用：

表 13-2

FETCH 选项	描 述
NEXT	该选项使得在结果集中向前移动一行，它是主要的游标选项。90%或更多的游标有此选项就够了。在决定是否声明为 FORWARD_ONLY 时，要记得这个。当尝试进行 FETCH NEXT，并且这导致超出了最后一条记录时，@@FETCH_STATUS 将会为-1
PRIOR	正如你可能已经猜到的，该选项的功能与 NEXT 相反。该选项向后移动一行。当位于结果集中的第一行时执行 FETCH PRIOR，@@FETCH_STATUS 将为-1，就好像在使用 FETCH NEXT 时超出了文件末尾一样
FIRST	和大多数游标选项一样，该选项的名称很清楚地表明了它的作用。如果执行 FETCH FIRST，则你将处于记录集中的第一行。使用该选项时，如果结果集为空，@@FETCH_STATUS 将为-1

(续表)

FETCH 选项	描 述
LAST	该选项与 FIRST 的功能相反, FETCH LAST 将使你移动到结果集中的最后一行。同样, 使用该选项时, 唯一会使 @@FETCH_STATUS 为-1 的情况是结果集为空
ABSOLUTE	使用该选项时, 需要提供一个整数值, 该值表明距离游标开头有多少行。如果提供的值为负, 则表明距离游标末尾有多少行。注意, 动态游标不支持该选项(由于动态游标中的成员在每次提取时重新生成, 你可以“真正知道你在哪里”)。在一些客户访问对象模型中, 这大致等同于导航到某个特定的“绝对位置”
RELATIVE	这种导航方式是指相对于当前行向前或向后移动指定数目的行

在前面介绍游标时, 我们已经对这些选项作了相当多的讨论。其他导航选项的工作方式几乎一样。

13.5 在游标中修改数据

一直到现在, 我们都没有真正介绍直接在游标中修改数据的概念。现在是时候了解一下在游标中对记录进行更新和删除了。

由于这里是在处理特定的行而非行集数据, 因此, 需要用某种专门的语法来告诉 SQL Server 我们想要进行更新。幸运的是, 在知道了如何执行 UPDATE 或 DELETE 的情况下, 该语法实际上将是很简单的。

我们将在游标底层的表中更新或删除数据。其操作与运行我们已熟悉的 UPDATE 和 DELETE 语句一样简单, 不过需要用一个匹配游标行的 WHERE 子句来进行限制。在 DELETE 或 UPDATE 语句中加入一行语句即可:

```
WHERE CURRENT OF <cursor name>
```

该语法没有什么需要特别之处。不过, 为了完美起见, 我们准备使用这一语法来实现一个游标。

```
USE AdventureWorks2008;
/* Build the table that we'll be playing with this time */
SELECT SalesOrderID, CustomerID
INTO CursorTable
FROM Sales.SalesOrderHeader
WHERE SalesOrderID BETWEEN 43661 AND 43665;
```

```
-- Now create a unique index on it in the form of a primary key
ALTER TABLE CursorTable
    ADD CONSTRAINT PKCursor
    PRIMARY KEY (SalesOrderID);
```

```
/* The IDENTITY property was automatically brought over when
** we did our SELECT INTO, but I want to use my own OrderID
** value, so I'm going to turn IDENTITY_INSERT on so that I
** can override the identity value.
*/
SET IDENTITY_INSERT CursorTable ON;
```


SQL Server 2008 高级程序设计

```

-- Declare our cursor
DECLARE CursorTest CURSOR
SCROLL          -- So we can scroll back and see if the changes are there
KEYSET
FOR
SELECT SalesOrderID, CustomerID
FROM CursorTable;

-- Declare our two holding variables
DECLARE @SalesOrderID int;
DECLARE @CustomerID varchar(5);

-- Get the cursor open and the first record fetched
OPEN CursorTest;
FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID;

-- Now loop through them all
WHILE @@FETCH_STATUS=0
BEGIN
    IF (@SalesOrderID % 2 = 0) -- Even number, so we'll update it
    BEGIN
        -- Make a change. This time though, we'll do it using cursor syntax
        UPDATE CursorTable
            SET CustomerID = -99999
            WHERE CURRENT OF CursorTest;
    END
    ELSE -- Must be odd, so we'll delete it.
    BEGIN
        -- Now we'll delete a record so we can see how to deal with that
        DELETE CursorTable
            WHERE CURRENT OF CursorTest;
    END
    FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID;
END

-- Now go back to the top. We can do this since we have a scrollable cursor
FETCH FIRST FROM CursorTest INTO @SalesOrderID, @CustomerID;

-- And loop through again.
WHILE @@FETCH_STATUS != -1
BEGIN
    IF @@FETCH_STATUS = -2
    BEGIN
        PRINT ' MISSING! It probably was deleted.';
    END
    ELSE
    BEGIN
        PRINT CAST(@SalesOrderID AS varchar) + ' ' + CAST(@CustomerID AS varchar);
    END
    FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID;
END

-- Now it's time to clean up after ourselves
CLOSE CursorTest;

DEALLOCATE CursorTest;

DROP TABLE CursorTable;

```

同样,把这个游标当作一个全新的游标来对待。我们已经做了很多的删除、添加和更新工作,我想你会发现,与必须通过逐行浏览来找出可能缺失的内容相比,第二次只是键入内容要更容易些。

我们再次使用了在本书前面用过的取模运算符(`%`)。我们知道,该运算符只返回余数。因此,如果任何数除以 2 的余数为零,那我们就知道该数是偶数。

这里剩下的部分没有任何难处可言,而且我们很快就得到了一些结果:

```
(5 row(s) affected)

(1 row(s) affected)

(1 row(s) affected)

(1 row(s) affected)

(1 row(s) affected)

(1 row(s) affected)
MISSING! It probably was deleted.
43662 *
MISSING! It probably was deleted.
43664 *
MISSING! It probably was deleted.
```

可以看到多个“1 row affected”的消息,这是针对所有受 UPDATE 和 DELETE 语句影响的行所返回的消息。当我们进行最后一个结果集的枚举时,很快就能知道删除了所有的奇数数值(这是我们让代码去做的事情),并且,用新的 CustomerID 更新了偶数数值的行。

没有任何难点——只是一个使用了 WHERE CURRENT 参数的 WHERE 子句而已。

13.6 小结

游标使我们想起了过去能够进行逐行处理的时光。听上去很浪漫。但这种想法是不正确的!无论何时如果能够避开它,我都会坚持使用行集操作的。

事实是行集操作无法完成所有的事情。任何时候必须基于每一行来解决问题时,游标都是解决的方式。注意,我使用了“必须”一词,并且那也是你应考虑的方式。在处理一些无法通过任何其他方法解决的问题方面,游标还是相当不错的。

话虽如此,还是记得尽可能地避免使用游标。游标非常消耗资源,而且几乎总是会对性能造成 100 倍甚至更大的负面影响。如果只是想着那种逐行的方法,那肯定会觉得其非常诱人——尤其是如果你所熟悉的是大型机世界或者 dBase 后台。千万不要落入那个圈套!应只在别无其他选择时使用游标。



第14章

Reporting Services

我所写的书中有一些章节重叠了入门篇和高级篇的部分内容。虽然初级和高级内容应该分得很清，但那其实仅限于理想情况：即人人都以同种方式、按同样顺序获得经验，并且对初级和高级的定义有统一的认识。

本章中就有一些重叠的内容，已经阅读过入门篇的读者会注意到这一点。对于 Reporting Services 而言，这样做的原因是多方面的，主要的两个原因是：

- 有些人只是出于控制个人报表的目的而进行数据库开发(在这种情况下，他们几乎都是直接从 Reporting Services 开始学起，然后学习用于支持报表中的数据的查询)。而还有一些人是数据库“专家”，他们的目的在于使用 SQL Server 提供的“额外”功能。
- 这是一项较新的功能(相对于 SQL Server 这个具有长久生命力的产品而言)，所以它对于许多专业级别的人士来说也是很生疏的。

即使阅读过入门篇中的“报告”一章，对本章内容也不要匆匆一瞥而过。因为尽管这里重复了一些关键内容，但是会稍作深入讨论，并更着重于介绍真正面向开发人员的内容(而较少关注模型驱动方面的内容)。不过，直接跳至有关数据源和数据源视图的章节也是可以的，我们将从更为“专业”的角度研究其中的内容，包括参数化、钻取以及图表制作。

14.1 报表服务概述

在完成了所有查询语句的编写和存储过程的运行之后，为使得数据有用，还有一件相当重要的事情要做——使数据对终端用户可用。

报表是一种看上去极为简单，实则相当棘手的事物。你不能只是简单地把数字堆在人们面前。这些数字必须是有意义的，可能的话，还要能够抓住报表接受者的注意力。要生成实际可用的、有用的报表，需记住以下几点：

- **数据的使用量要恰当：**一份报表中所列出的数据不能太多，也不能太少。一份充斥了大量数据的报表会令阅读者迅速失去兴趣，并且仅在最初的几个项生成之后就被弃而不用。同样，一份言之无物的报表也仅能得到匆匆一瞥，随即便被毫不犹豫地丢弃。所以，应在恰当的数据和恰当的数据量之间找到一种平衡。

- **报表要美观大方：**不得不说的是，报表的另一个重要要素就是所谓的“美观”，也就是说，报表应该做得赏心悦目。难看的报表是没有人会看的。

在这一章中，我们将介绍 Reporting Services(通常也称作 SSRS)的几个关键概念，然后再进一步介绍一些高级内容。虽然略过了一些“基础”内容，但为了使更高级的内容明白易懂，我们也涉及到了一些必要的基本知识，但接着很快就介绍 Report Designer 的内容，其涉及 Reporting Services 提供的最为高级的报表选项。

提示：

出于行文简洁的考虑(也是为了使重复的内容最少化)，本书对于报表模型仅介绍其作用，而不附具体示例，以此区分初级和高级内容。不过，即使你在阅读本章之前还不太理解报表模型，也可以通过学习一些核心内容，如数据源和报表设计器，来了解如何使用报表模型器和报表模型设计器。实际报表的构建也是一样简单。

14.2 Reporting Services 入门

很有可能，你已经生成过一些报表。它们可能是通过打印机打印出来的纸质报表(采用像 Access 所提供的那样的初级报表功能，实际上这是我最喜欢的 Access 的功能之一)。或者你使用了某种相当强大的报表引擎，例如 Crystal Reports。即便没有使用过这些工具，你应该也有过从存储过程生成简单(虽然不太美观)的报表来提交给老板的经历。我认为这种可能性很大。

但实际情况是，如今的经理和合作者们的期望越来越高。Reporting Services 才会应运而生，它实际上有两种不同的操作：

- **报表模型(Report Model)：**它利用一个相对简单、Web 驱动接口让终端用户自行创建简单报表。
- **在 Business Intelligence Development Studio 中生成报表：**这并不意味着一定要编写代码(实际上通过拖放功能就可以创建一个相当棒的报表)，而是可根据所需创建更复杂的报表。

注意，虽然用户最终是可以通过相同的 Reporting Services Web 主机访问这些报表，但这些报表是基于不同的体系结构(并且以不同的样式创建)。

需要补充的是，Reporting Services 提供预生成报表的功能(如果构成报表基础的查询需要一些时间运行，那么这很有用)，同样，Reporting Services 也提供通过电子邮件分发报表的功能。报表的输出格式可以是 PDF、Excel 或 Word。

14.2.1 管理 Reporting Services 的工具

Reporting Services 提供了多种工具帮助创建、使用及管理报表。它们包括：

- **Reporting Services 配置管理器：**在 SQL Server 主文件夹下的“Configuration Tools”子文件夹中可以找到该工具。使用这一工具可以对下列内容进行配置，如 Reporting Services 的运行帐户、支持的 Web 服务器所响应的 IP 地址和端口、用于 Reporting Services 的虚拟目录名、使用的电子邮件帐户、跟踪 Reporting Services 信息的数据库，以及加密密钥和可伸缩性配置信息。

- **Business Intelligence Development Studio(BIDS):** 这实质上就是安装了一套针对 Reporting Services、Analytics Integration Services 和 Data Mining 的模板的 Visual Studio。如果已经安装了 Visual Studio 2008, 那么 BIDS 也只是添加了一些模板和访问 Visual Studio 的快捷方式而已。本书将在后面的多个章节中广泛使用 Development Studio(有时它是以 SQL Server 的基本安装形式存在, 有时又作为 Visual Studio 完整安装的一部分)。
- **SQL Server Management Studio:** 在 Management Studio 中, 可以连接到所有和 SQL Server 相关的服务, 以此来管理某个特定的服务。尽管 Management Studio 中只打包了我认为是功能完全的基本数据引擎, 但它仍是执行安全性需求最高的任务和进行作业调度的场所。
- **Report Server Web 站点:** 在这里实际运行将在 Reporting Services 中执行的大多数报表, 但却是通过“站点设置”链接来进行的(位于浏览器的右上方)。这里也可以管理一些服务器的要素(例如缓存、角色分配以及计划)。

遗憾的是, 还没有单个的工具可以囊括 Reporting Services 中的所有内容。事实上, 上述工具距离这一目标都还相差甚远(就像 Management Studio 之于数据库引擎)。但是, 通过组合使用各项工具, 可以管理 Reporting Services 的方方面面。

14.2.2 访问 Reporting Services 的其他方法

Reporting Services 同样也支持一个相当健壮的 Web 服务模型。通过提供一套库来支持 .NET 项目访问 Reporting Services Web Services API。本章末尾将对其基本内容作一个介绍。

14.3 报表服务器项目

报表模型(在入门篇中是作为 Reporting Services 的主要内容)仅算是触及到一些皮毛。Reporting Services 的灵活性远远不止于此(事实上, 有很多专门关于 Reporting Services 的书籍; 其内容相当丰富)。除了报表模型之外, Business Intelligence Development Studio 也允许创建报表服务器项目。

前文提到, 关于这一主题可以写一整本书, 所以这里将采取的方法是通过一个简单的示例开始试着了解。然后再稍微做一点扩展。

提示:

在此次发行版本中, 报表服务项目的观感都发生了很大变化。Microsoft 购买了 Dundas(一家组件开发公司)大量 Reporting Services 组件的许可。Reporting Services 的组件有了重大升级。

在了解报表服务器项目的过程中, 将从对报表模型器和报表服务器项目而言都比较常见的几处关键内容开始。如果你已经熟悉了数据源和数据源视图, 那么可以浏览下面的两节, 选取与项目示例相关的内容即可, 或者也可以直接跳至关于实际报表布局的那一节。

让我们从一个报表服务器项目开始。首先打开 Business Intelligence Development Studio, 打开

一个新项目。通过 Business Intelligence Development Studio 中的“报表服务器项目”模板使用商业智能，如图 14-1 所示。



图 14-1

注意：

请注意这一对话框的确切外观将因 Visual Studio 安装与否以及具体安装的语言和模板而有所不同。该图是 Visual Studio 的一个完整版本，为的是满足本书中更高级内容的需求。

这一项目将作为本章中大多数操作要进行的对象。在该项目创建之后，便可以深入到报表的一些关键概念。对于已经阅读过初级内容的人而言，其中的某些概念可以说是一次回顾，但是你也一定想将这这份报表的内容整合起来，使其对随后出现的更为健壮的示例而言也是可用的。

14.3.1 数据源

数据源和数据源视图(这将在后文提到)可能是 Reporting Services 中最为重要的内容了。无论建立的报表是何种类型，也无论报表是使用报表模型器还是报表项目，这两项内容都以某种方式发挥着作用。虽然它们的名称相似，但是在将数据安排进入报表的体系结构中，它们各自服务的层次略有不同。

数据源从本质上讲就是对到从中获取数据的位置的连接的定义。它可以是与一个 SQL Server 数据源的连接，也可以是与任意 OLE DB 或 ODBC 数据源的连接。如果稍稍思考一下这种可能性，那么很快就能得出结论，那就是，虽然 Reporting Services 与 SQL Server 相关联，但是却可以在报表中使用广泛的非 SQL Server 数据源。这确实是一个非常强大的概念。

数据源有两种类型：

- **嵌入式：**这种类型的数据源储存在定义报表的同一文件内。在本章后面的内容中将会介绍到 XML(称作报表定义语言或 RDL, Report Definition Language)，只需说明所有有关数

据源的信息都储存在报表定义文件下的 XML 块中就足够了。对这种数据源定义的访问受限于其所嵌入的报表。

- **共享式：**这种类型的数据源和嵌入式大体相同，区别在于这种数据源的定义存储在其自身文件中(扩展名通常为.ds)。

本章将在后面的内容中使用到共享式数据源。

无论是何种类型，数据源都存储了多条必须的信息，并且有选择地存储了附加选项，以应对可能出现的安全问题。

1. 创建一个数据源

接下来要创建一个数据源，并在本章其后的内容中一直使用它。

如果你的 Visual Studio 仍是默认设置，那么可以看一下右上方的“解决方案资源管理器”。右击“共享数据源”，选择“添加新数据源”，如图 14-2 所示。

然后进入“共享数据源属性”对话框(如图 14-3 所示)。

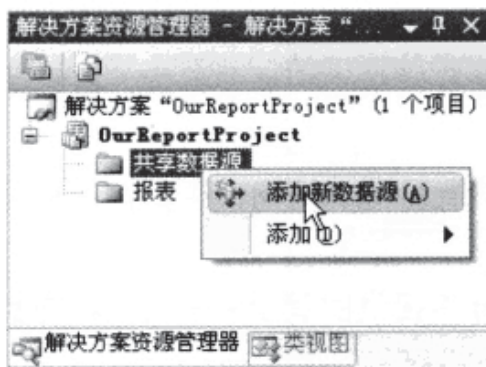


图 14-2

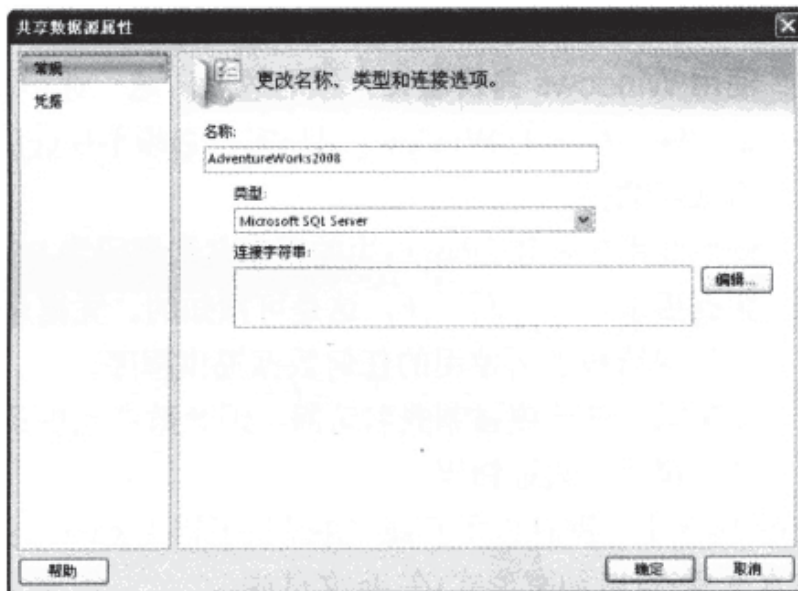


图 14-3

这个对话框有两个主要部分，第一部分需要定义名称(本例中将其命名为将要连接的数据库名)和数据源的连接字符串(对不熟悉连接字符串的人而言，它的作用是为所有连接到数据源的对象指出路径和登录方法)。对连接字符串可以直接编辑，也可以单击“编辑”按钮，弹出如图 14-4 所示的“连接属性”对话框。

提示：

当我第一次看到这个对话框时，我微微有些吃惊，因为它与我们在 Management Studio 中多次使用的连接对话框是不同的；但是，它包含了相同的基本要素，只是在视觉上略有不同(简而言之，不用为了外观不同而担心)。

在本例中，选择了本地服务器、系统管理员帐户(sa)以及我们非常熟悉的 AdventureWorks2008 数据库。

继续单击“确定”，然后单击“共享数据源属性”对话框中的“凭据”选项，得到数据源安全选项(见图 14-5)。

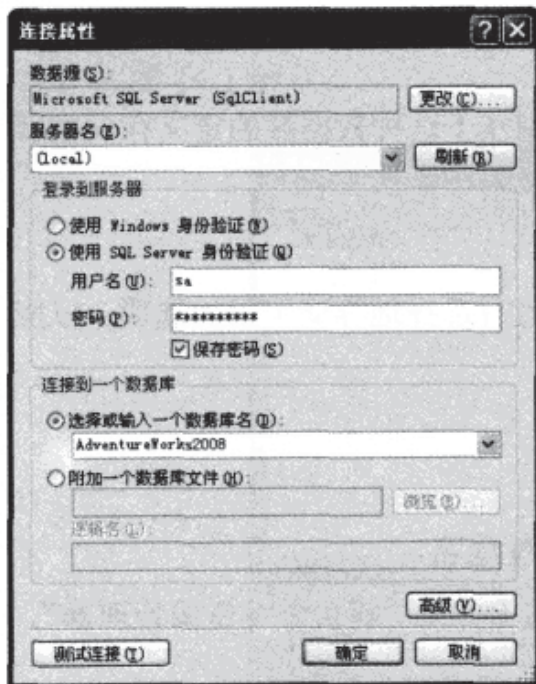


图 14-4

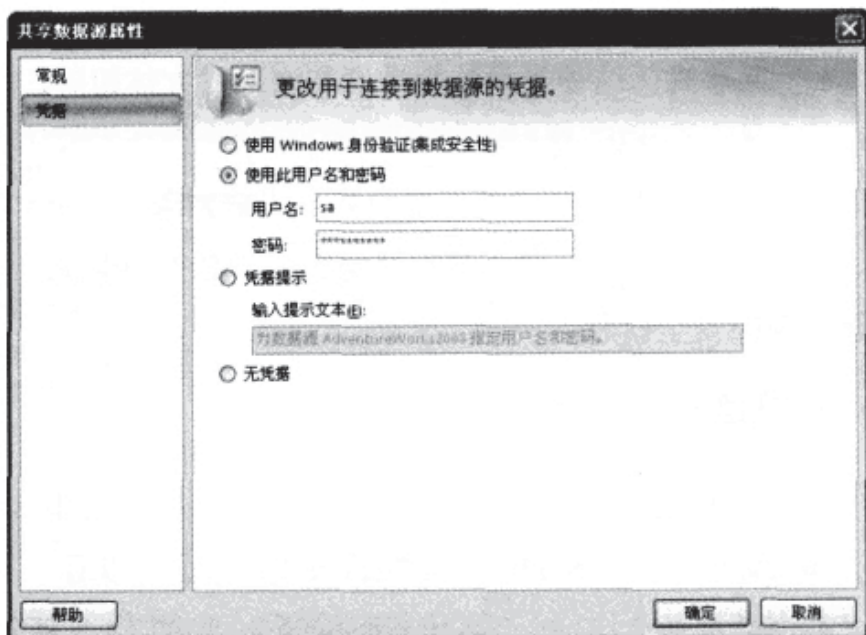


图 14-5

这里有一些选项颇为值得研究——它们包括：

- **使用 Windows 身份验证：**顾名思义，这一选项是基于执行报表的用户来进行身份验证。这意味着相关的 Windows 用户帐户必须不仅能访问报表，而且有权访问和报表相关的所有底层数据。
- **使用此用户名和密码：**引用的用户名和密码为 SQL Server 登录信息(而不是 Windows 的)。
- **凭据提示：**与上面一样，这是可预知的。凭据是在运行时从用户处获得的。提供的凭据将传递给报表所使用的任何数据提供程序。
- **无凭据：**该选项强制匿名访问，因此数据提供程序应支持这类访问，否则在运行报表时会出现身份验证错误。

在图 14-5 中，我们选择了 sa，并提供了相关密码。这意味着提供的登录名和密码将与数据源一起被永久保存(以加密形式)在 ds 文件中。

在该对话框中单击“确定”后，回到较普通的 Visual Studio 项目，但此时已有了新数据源，开始准备为报表创建更多必须的内容。

14.3.2 使用报表向导

虽然在创建项目时，没有选择报表向导项目类型，但是报表向导中的各部分对报表的创建而言也是可用的。事实上，默认请求一个新报表的简单方式就是进入报表向导。除非你取消了所有向导创建了一张空白报表，否则，Visual Studio 也会尝试使用该向导来进行一些操作。

继续看已经建立的示例，这里添加了一个报表以完成报表向导的过程演示。在本例中，假设经理需要一份摘要报表，要求显示 David Campbell 在 2003 年 7 月完成的所有销售订单的销售总量(按类别分类)。并且经理已事先提醒过她可能晚些时候还要查看其他销售人员不同时期业绩的报表，但是现在她只需要 Campbell 先生在 2003 年 7 月的相关信息。

首先右击“解决方案资源管理器”中的“报表”节点，选择“添加新报表”，如图 14-6 所示，进入“报表向导”对话框。

单击“下一步”进入“选择数据源”对话框，如图 14-7 所示。请注意，虽然这里选择使用的

是刚才创建的共享数据源，但也可以在这个对话框中创建一个新数据源(新数据源可以是嵌入式的，但是经过选择之后，也可以在后面转换为共享式)。



图 14-6

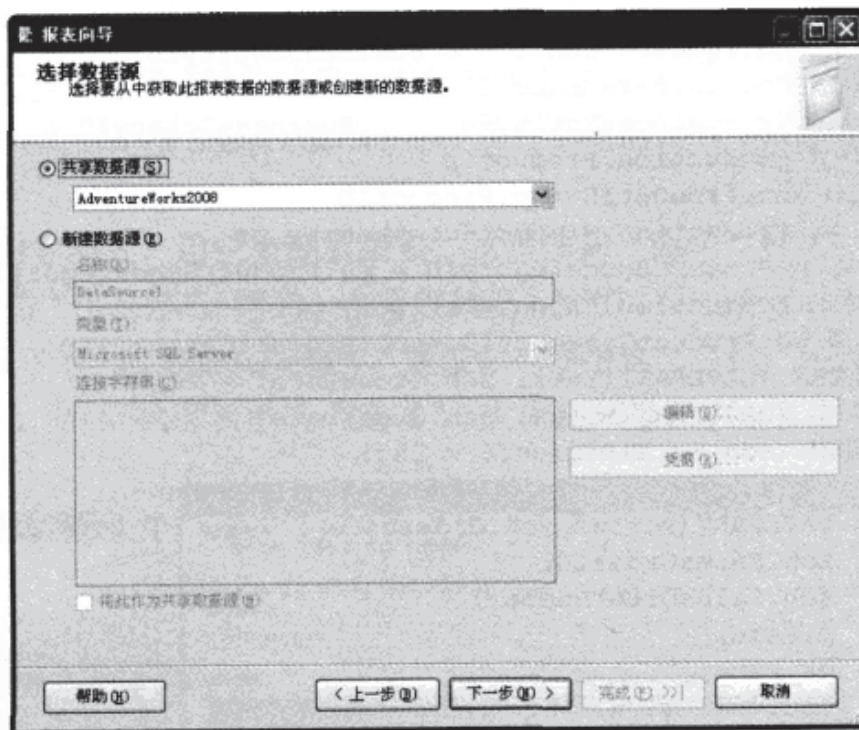


图 14-7

再次单击“下一步”进入“查询生成器”对话框，如图 14-8 所示。这里已经创建了一个查询，语句如下：

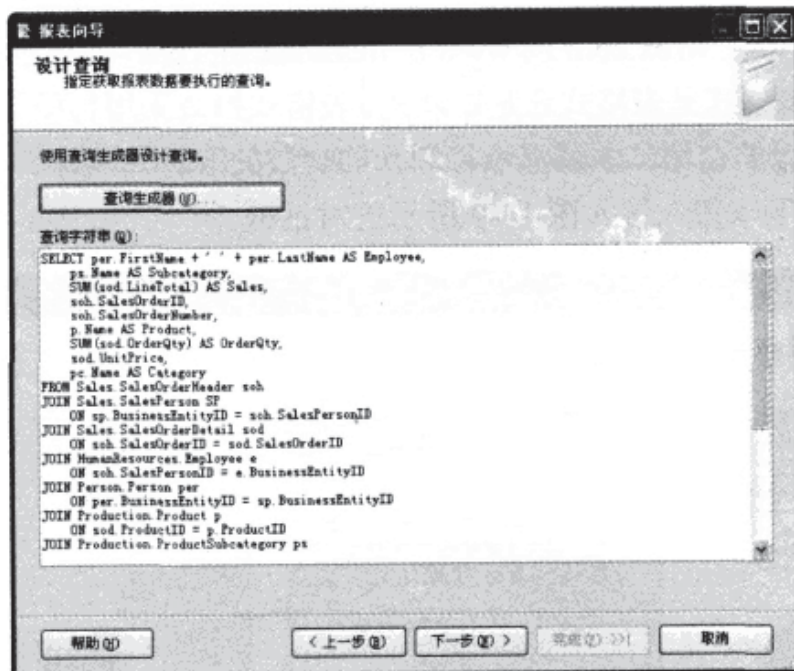


图 14-8

```
SELECT per.FirstName + ' ' + per.LastName AS Employee,
       ps.Name AS Subcategory,
       SUM(sod.LineTotal) AS Sales,
       soh.SalesOrderID,
       soh.SalesOrderNumber,
       p.Name AS Product,
       SUM(sod.OrderQty) AS OrderQty,
       sod.UnitPrice,
       pc.Name AS Category
FROM Sales.SalesOrderHeader soh
JOIN Sales.SalesPerson SP
```



```

ON sp.BusinessEntityID = soh.SalesPersonID
JOIN Sales.SalesOrderDetail sod
ON soh.SalesOrderID = sod.SalesOrderID
JOIN HumanResources.Employee e
ON soh.SalesPersonID = e.BusinessEntityID
JOIN Person.Person per
ON per.BusinessEntityID = sp.BusinessEntityID
JOIN Production.Product p
ON sod.ProductID = p.ProductID
JOIN Production.ProductSubcategory ps
ON p.ProductSubcategoryID = ps.ProductSubcategoryID
JOIN Production.ProductCategory pc
ON ps.ProductCategoryID = pc.ProductCategoryID
WHERE (DATEPART(Year, soh.OrderDate) = 2003)
AND (DATEPART(Month, soh.OrderDate) = 7)
AND (soh.SalesPersonID = 283)
GROUP BY per.FirstName + ' ' + per.LastName,
DATEPART(Month, soh.OrderDate),
soh.SalesOrderID,
soh.SalesOrderNumber,
p.Name,
ps.Name,
sod.UnitPrice,
pc.Name

```

这个查询中并没有什么深奥的内容,它只是把 ID 号为 283 的销售员(也就是 David Campbell)在 2003 年 7 月的总销售额收集起来。本章稍后将演示如何使其变得可选,但是现在采用的简单的、硬编码查询。

粘贴这段查询代码(可在 wrox.com 或 www.professionalsql.com 网站上可获得这一示例代码),单击“下一步”,选择报表格式是表格式还是矩阵式。表格式报表采用传统的逐行显示数据的布局。矩阵式报表查找的是数据的交集,主要显示行和列的交集的汇总。具体到本例中的报表,这里选择的是表格式,单击“下一步”进入图 14-9 所示的对话框。

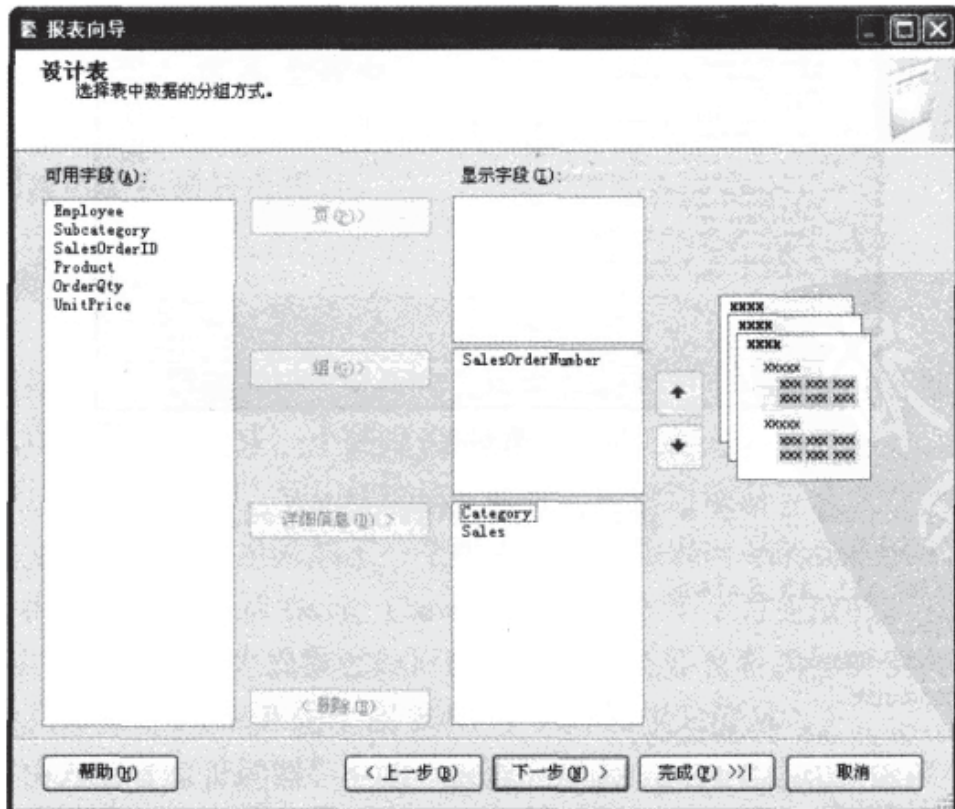


图 14-9

生成的销售报表将显示出 Campbell 先生在 2003 年 7 月所发出的所有销售订单的总和。现在所做的选择是要让向导创建我们所需要的格式。选择 SalesOrderNumber 作为“组”，选择 Category 和 Sales 字段作为“详细信息”项，单击“下一步”。在下一个对话框中(如图 14-10 所示)，我选择了“块”格式，其中并没藏着什么玄机，选择这种格式只是因为我认为它与具体数据最为相配。我还勾选了“包括小计”复选框。因为我们是按 SalesOrderNumber 进行分组的，所以将会得到一个所有 SalesOrderNumber 值的总和。

再次单击“下一步”，选择一种向导样式，用它来配置报表。这里选择“海洋”样式，不过所有样式都能正常工作。最后一次单击“下一步”，得到一个报表摘要，如图 14-11 所示，内容包括向导的任务以及为报表名称(我选择了 SalesOrderSummary，建议你也使用这一名称，因为在本章的学习过程中，将对报表作出改动)。然后就可以单击“完成”，生成实际报表。

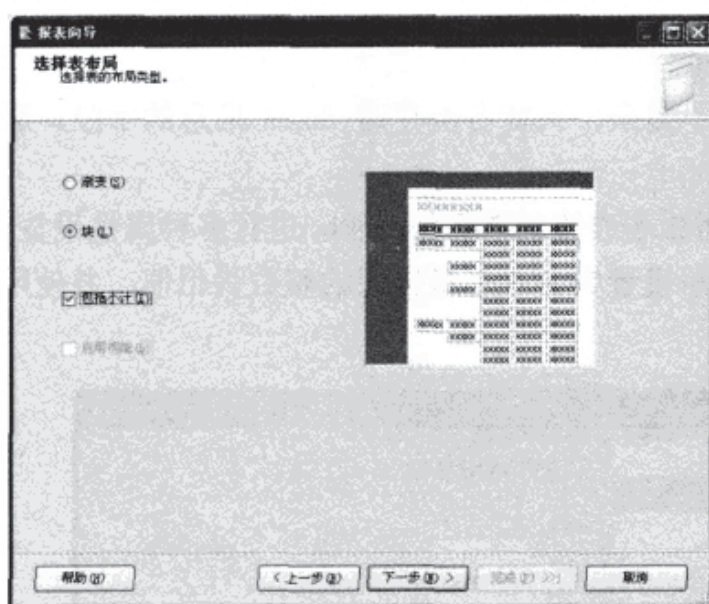


图 14-10

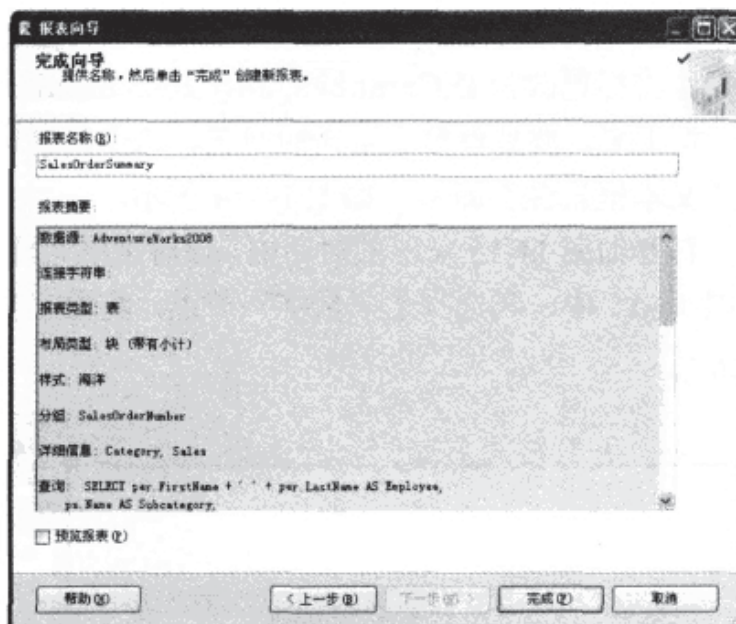


图 14-11

第一次出现的报表(如图 14-12 所示)看上去并不复杂。

接下来，选择“预览”选项卡，预览包含真实数据的报表外观(如图 14-13 所示)。

Sales Order Number	Category	Sales
5051089	Clothing	29.388000
	Components	3299.040000
	Components	1637.400000
	Components	144.324000
	Components	46.594000
	Components	126.336000
	Components	749.370000
	Components	599.496000
	Components	599.496000
	Components	792.150000
	Components	950.580000
	Components	475.290000
	Components	475.290000
	Components	26.724000
	Components	72.682000
	Components	65.088000
	Clothing	149.970000
	Components	1310.724000
	Components	1092.270000
	Components	655.362000
	Components	37.254000
	Components	140.904000
	Bikes	6894.970000
	Bikes	9638.958000
Total	[Sum(Sales)]	

图 14-12

Sales Order Number	Category	Sales
5051089	Clothing	29.388000
	Components	3299.040000
	Components	1637.400000
	Components	144.324000
	Components	46.594000
	Components	126.336000
	Components	749.370000
	Components	599.496000
	Components	599.496000
	Components	792.150000
	Components	950.580000
	Components	475.290000
	Components	475.290000
	Components	26.724000
	Components	72.682000
	Components	65.088000
	Clothing	149.970000
	Components	1310.724000
	Components	1092.270000
	Components	655.362000
	Components	37.254000
	Components	140.904000
	Bikes	6894.970000
	Bikes	9638.958000

图 14-13

这是一个好的开始，但还有一些重大的缺陷，所以接下来要学习对报表进行编辑。

14.3.3 编辑报表

要编辑报表，需先回到 Visual Studio 报表中的“设计”选项卡。继续使用已有的示例，为了使报表看上去清爽整洁，需要注意以下几个问题：

- 标题格式尽量恰当得体。
- 数字值尽量接近货币值。
- 我们要查看每个销售类别的信息，而不是总和。

下面将逐条讨论这些问题。

首先，要更改标题。这是最简单的一项改动，只要在标题区域内单击一次选中标题，然后再单击一次激活光标，就可以像对其他选项卡对象一样对标题进行编辑。直接双击也是可以的。按此操作将标题改为 **D.Campbell, July 2003 Summary**。

接下来，要处理数字格式的问题。这个问题也不复杂。只要右击存储 Sales 信息的字段，选择“文本框属性”即可，如图 14-14 所示。

打开如图 14-15 所示的对话框，该对话框允许对报表表格的单元格(tablix)进行多种属性设置。在图 14-15 中，我选择了“数字”节点，将数字显示设置为舍入至最接近的整数货币值，并使用千分符。

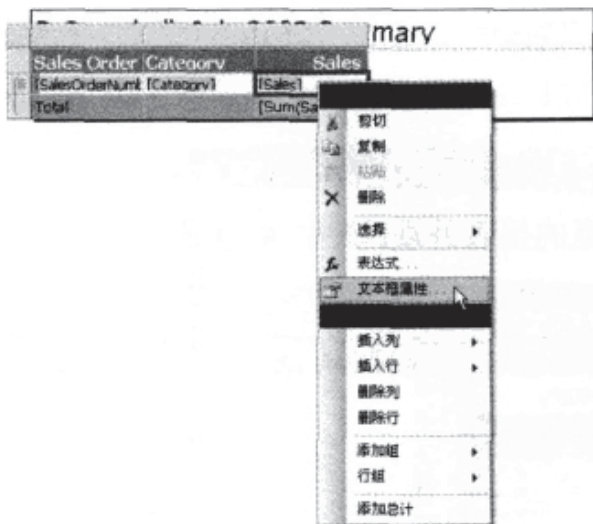


图 14-14

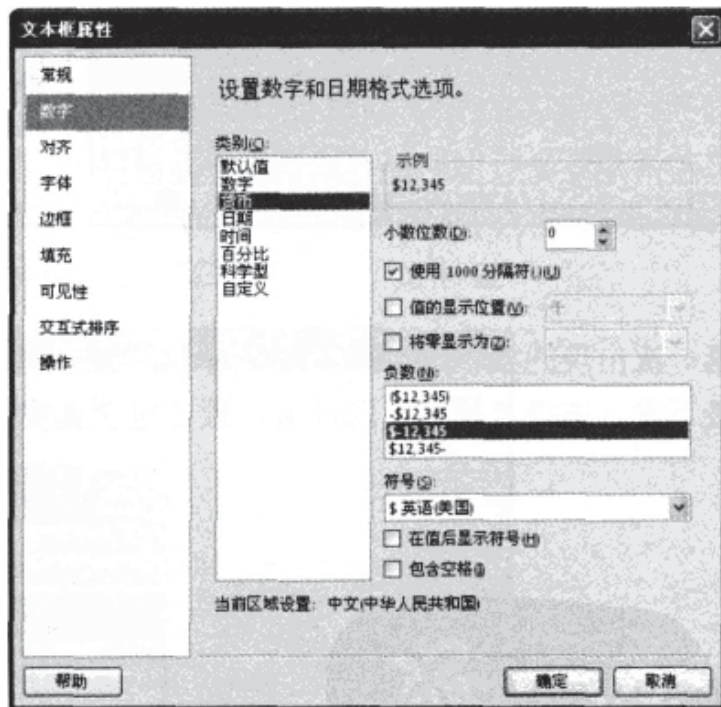


图 14-15

提示：

这项操作并没有询问使用何种符号作为千分符，也没有将它假设为逗号。千分符的使用将随报表服务器的区域配置不同而不同，可针对每一份报表进行覆盖。

接下来要进入最后也是最难的一项改变：将每个销售订单的所有类别进行汇总。首先右击含有 [Sales] 值的单元格，如图 14-16 所示。选择“行组”，使用图 14-17 所示的对话框修改其属性。这限制了在更大的 SalesOrderNumber 组中，每一类别只返回一行(注意 tablix 最左边的括号，回忆一下，我们曾在报表向导中选择添加它)。到这里工作还没有完成。我们一直把重点放在类别上，

但是还应该对类别进行排序使其更具有可读性。要实现这一点,可以在当前对话框中选择“排序”节点,如图 14-18 所示。

当以上操作都完成后,可以再次预览报表,发现虽然许多问题都有了显著改善,但还是存在几个问题(如图 14-19 所示)。

虽然报表逐渐开始变得“美观”,但在数字方面仍有一些问题。如果将数字与先前返回的值(回到图 14-13)比较一下,很快可以发现数字没有进行相加。没错,报表不会显示出每一类的总和,而是对每一类别返回第一行。这样是不行的。



图 14-16

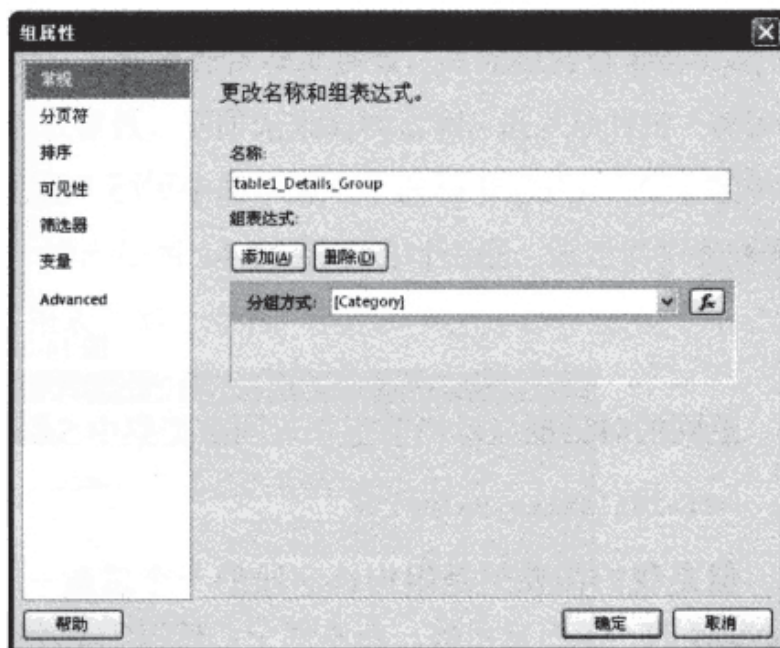


图 14-17

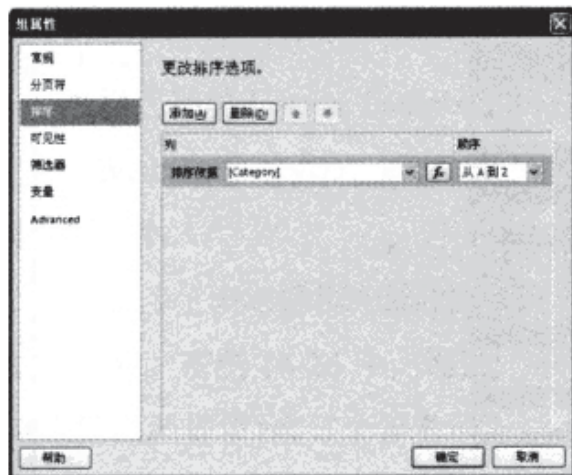


图 14-18

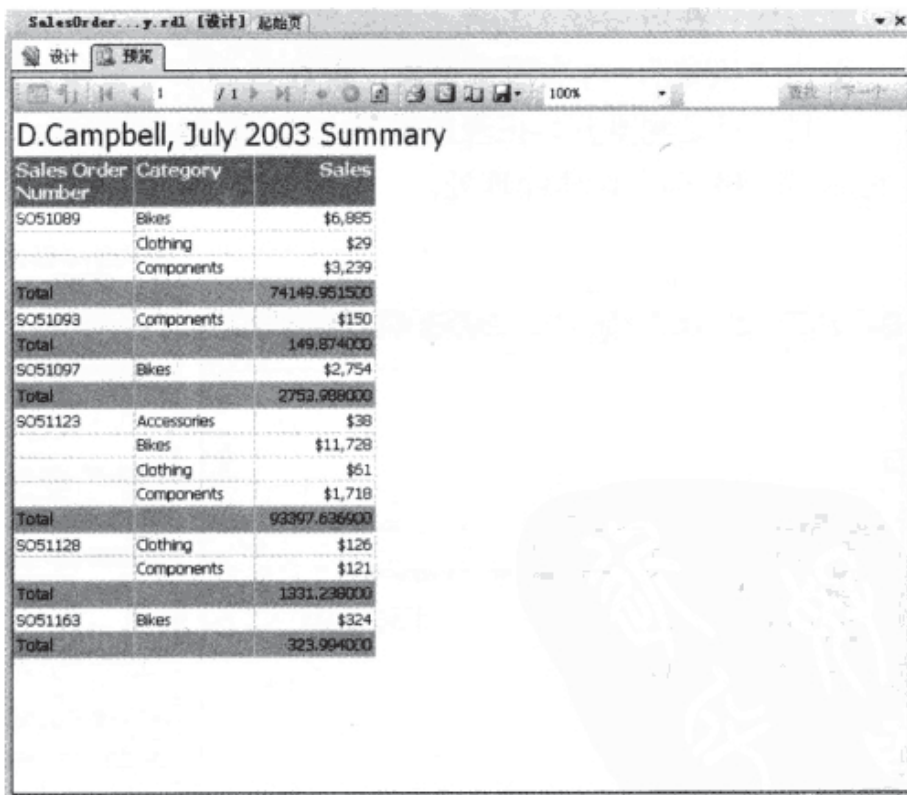


图 14-19

为解决这个问题,需要显式指出对每个单元格所要进行的操作是什么。再次右击[Sales]单元格,这一次要单击“表达式”,如图 14-20 所示。



图 14-20

返回的对话框显示当前正在返回数据集中 **Sales** 字段的确切值:

```
=Fields!Sales.Value
```

但是我们需要的是组中该字段的一个总数——或总和。要实现这一点, 可使用 Reporting Services 的一个内置函数。在本例中, 使用 **Sum** 函数:

```
=Sum(Fields!Sales.Value)
```

要看到它在对话框中的样子, 参见图 14-21。

单击“确定”, 再次预览报表, 现在得到了一份格式基本良好的报表(不要在格式上过度要求——目前只是刚刚开始报表的学习), 如图 14-22 所示, 并且已经可以准备运行, 打印输出(或输出为另一格式)并送达经理处。

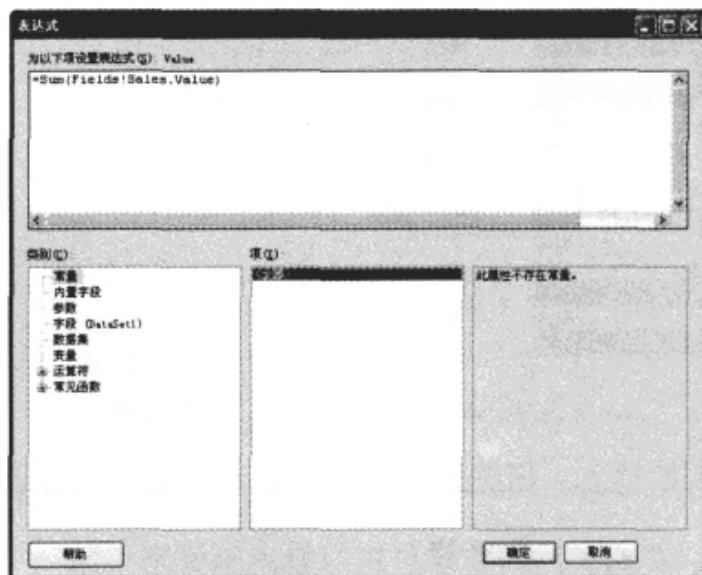


图 14-21

Sales Order Number	Category	Sales
9051089	Bikes	\$59,609
	Clothing	\$1,303
	Components	\$13,239
Total		\$74,150
9051093	Components	\$150
Total		\$150
9051097	Bikes	\$2,754
Total		\$2,754
9051123	Accessories	\$1,151
	Bikes	\$64,442
	Clothing	\$2,963
	Components	\$24,840
Total		\$93,396
9051128	Clothing	\$336
	Components	\$995
Total		\$1,331
9051163	Bikes	\$324
Total		\$324

图 14-22

14.3.4 参数化报表

关于 David Campbell 的这份报表做得很好,但是有很大局限性。回想之前,经理曾经提醒过她将会在晚些时候查看其他销售人员在其他时间内的相关信息。这便要求报表实现这一功能。

对于大多数报表项目来说,参数化都是至关重要的一个部分。所幸让 SQL Server 对参数化的报表进行识别还相对容易些。一旦报表被参数化,SQL Server 就会以某种方式提示用户提供一个参数值。本节将介绍许多使用户进行参数选择变得更为简单的选项。

第一步,要将最基本的参数化添加至报表中。要使报表依赖于参数,首先要将查询语句改为需要参数。然后只需通知报表在执行之前请求获取参数。下面先来编辑查询。进入项目中“视图”菜单下的“报表数据”项(在“解决方案资源管理器”面板中显示为一个选项卡形式)。“报表数据”选项卡如图 14-23 所示。双击报表中的数据集,弹出如图 14-24 所示的对话框,该对话框允许对查询进行编辑(有些报表有多个数据集。这里的报表只有一个)。

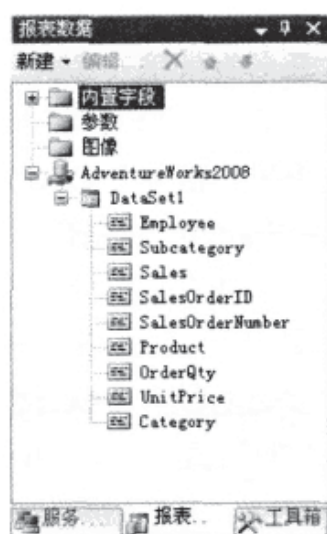


图 14-23

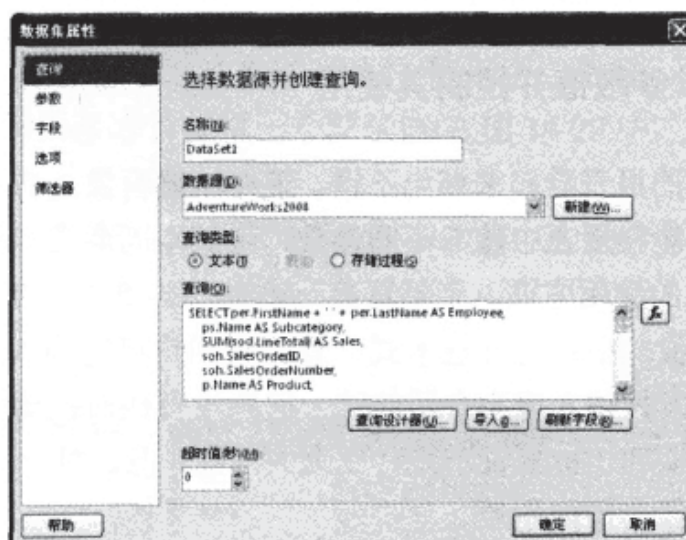


图 14-24

提示:

也可以右击数据集,选择“查询”,在一个单独的查询编辑器窗口中编辑查询。

这里已经将 Darren Campbell 的 BusinessEntityID、7 月和 2003 年这些硬编码值改为参数值(分别为@BusinessEntityID、@Month 和@Year)。完成之后,可以进入显示参数选项的对话框,如图 14-25 所示。

由于已在该对话框中添加了每一个参数,所以这里可以直接单击“确定”,准备预览(或直接运行)报表。在图 14-26 中,通过“预览”选项卡运行了报表。请注意在面板顶部它是如何要求提供(这里已经提供)这三个参数的。

在浏览报表的过程中,可以发现虽然它是以和原始报表中完全一样的值而告终,但是现在却能够针对某一时间段或某一销售代表来运行报表。报表开始逐渐更趋灵活。

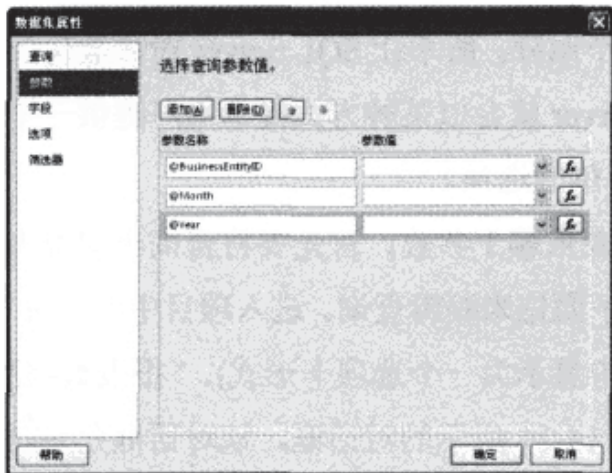


图 14-25

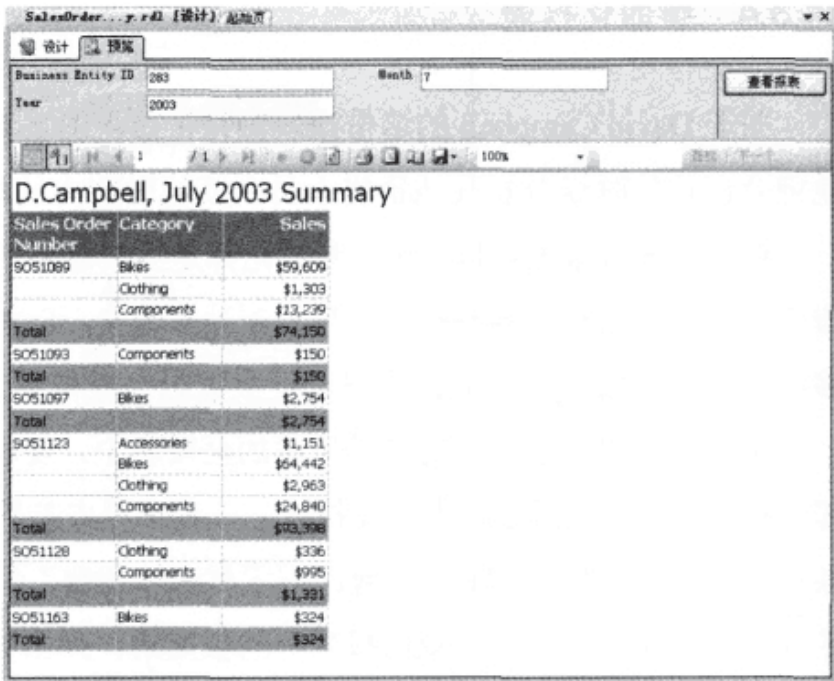


图 14-26

14.3.5 提供参数值并控制其使用

目前生成的报表看起来相当不错。现在不能再像之前那样仅提供一份关于 David Campbell 的报表了，而是能在报表中输入不同参数，包括不同雇员的 BusinessEntityID 和完全不同的时间段。但是，在报表的使用方面，仍然还有一些问题，其中包括：

- 输入值的格式为任意形式，意味着用户可能输入非法值。
- 对于何种输入值为合适的这一问题没有任何提示，用户要么知道，要么凭借猜测。这对于日期和年份来说不是什么太大问题，但是对于获得正确的销售人员的 BusinessEntityID 来说就会造成困难。
- 无论输入哪一位销售人员，报表头都被硬编码为 David Campbell。月份和年份也存在类似的问题。

下面看一下如何解决这些问题。

1. 创建预设的参数表

Reporting Services 拥有为参数创建预定义值列表的能力。该功能运用已定义的参数，并对其添加附加属性。

要将固定列表添加至 @Month 和 @Year 参数中，需导航至“报表数据”选项卡下的“参数”节点，展开列表，双击需要赋值的参数(也可以右击参数，选择“参数属性”)。对 @Month 参数进行上述操作，弹出如图 14-27 所示的对话框。

注意这里可以为参数设置一个自定义的提示(不一定为参数名)。还可以控制参数的初始可见性(有可能某一参数只有当另一参数设为一个特定值时才为有效)以及是否允许空值或空白值。

这里保留了大部分的默认设置，但是将数据类型改成了整型(记住，我们将月份的数字作为了一项参数)。下面就可以进入如图 14-28 所示的显示“可用值”节点的对话框。

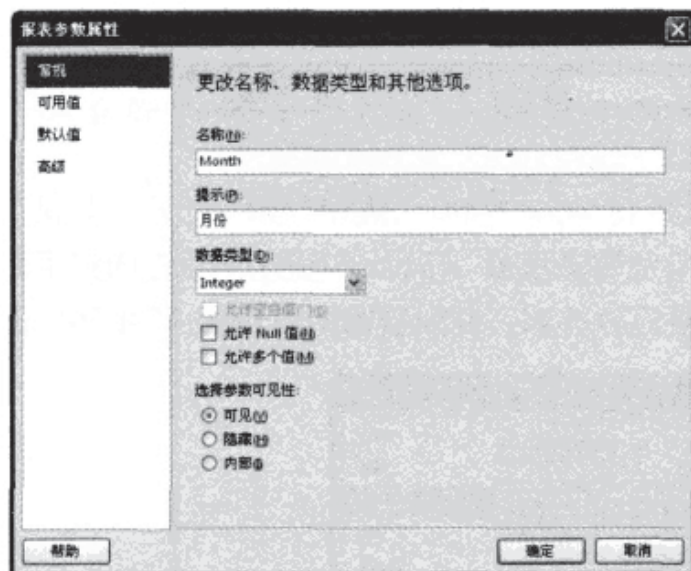


图 14-27

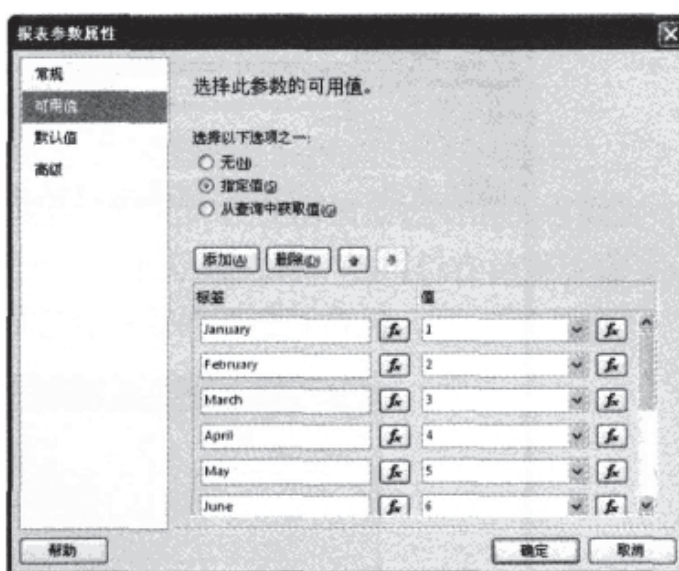


图 14-28

在这个对话框中，我作了许多修改——最明显的是提供了单独的标签和值。标签为用户显示出可供选择的项，值则是指报表在执行时将要传递给参数的内容。通过选择“指定值”单选按钮，可以创建这一列表。但是请注意，这一列表也可以是查询驱动的(稍后将作介绍)。

继续转到“默认值”节点，可以发现这里允许提供一个默认值(在图 14-29 中，选择的值是一直在使用的 7)。

最后，切换到“高级”节点(如图 14-30 所示)，如果用户改变参数值，这里给出了报表数据将选择在何时作出相应改变的选项。我们可以强制始终刷新，可以要求用户明确提出刷新，也可以让 SQL Server 来决定恰当的刷新时机。

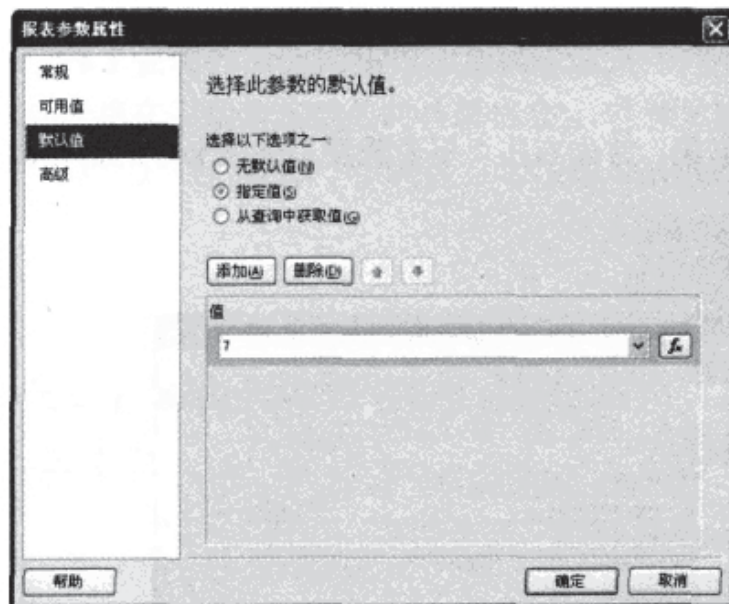


图 14-29

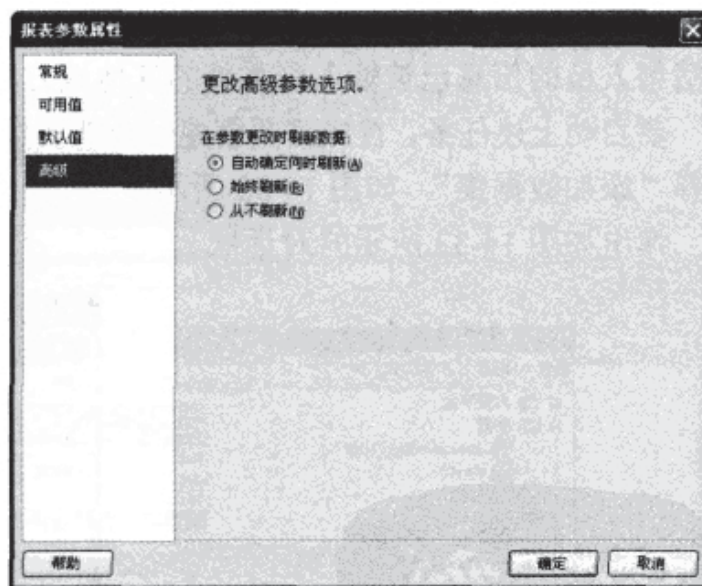


图 14-30

接下来你可以自己尝试一下将参数@Year 的数据类型设置为整数型，默认值为 2003。然后就可以再次预览或运行报表，查看所做改动的效果，如图 14-31 所示。

虽然在 BusinessEntityID 和 Year 这两个参数中没有发现显著的不同，但可以很快注意到现在的 Month 已变成一张下拉列表，即使月份的参数实际使用的是整数值，该列表也可以为每个月份提供名称。还可以测试一下在 year 字段中输入文本。SQL Server 会相对得体地指出类型不匹配的问题(虽然并不是以一种最完美的方式，但总比爆满屏幕的错误要好得多)。

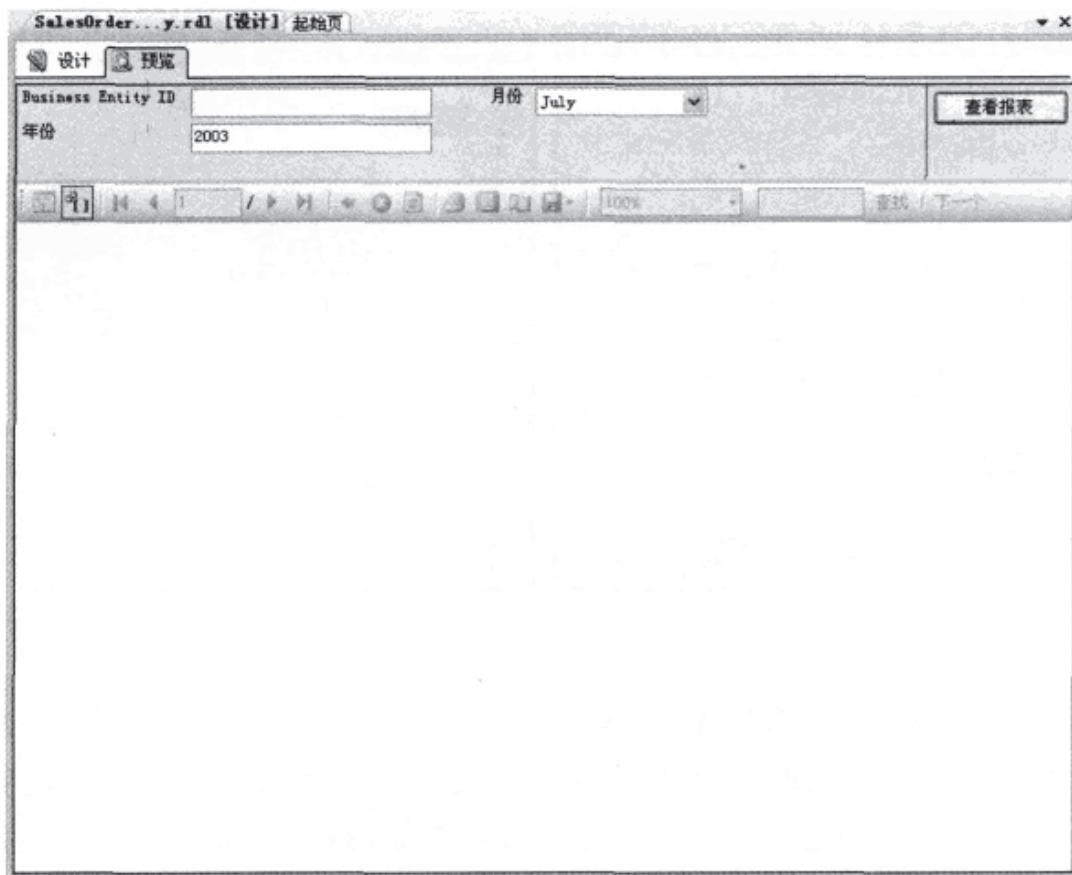


图 14-31

2. 通过查询创建参数表

为参数@BusinessEntityID 提供一个预先填充的列表要比其他两个参数难一些。虽然可以像@Month 一样创建一个固定列表，但那意味着只要销售人员的列表发生变动，就必须随时编辑报表。月份比较稳定，但是销售人员的变动频率却很高。每时每刻地编辑报表是不现实的，尤其是当销售人员的信息已被输入到系统的其他地方时。

要启动上述任务，首先需要创建一个新的数据集。首先右击“报表数据”选项卡中的数据源，选择“添加数据集”，如图 14-32 所示。

弹出如图 14-33 所示的对话框。



图 14-32

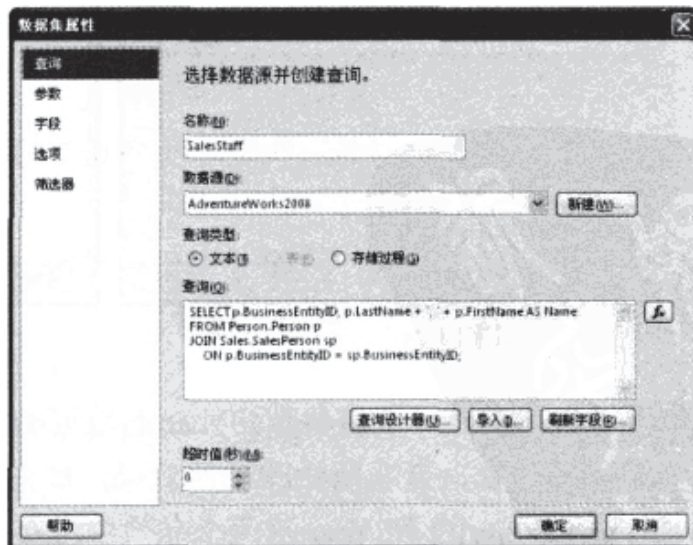


图 14-33

这里已经给出了一个列出所有销售人员的查询。它在对话框中是完全可见的，为使它更清楚，整理成如下所示：

```
SELECT p.BusinessEntityID, p.LastName + ', ' + p.FirstName
FROM Person.Person p
JOIN Sales.SalesPerson sp
ON p.BusinessEntityID = sp.BusinessEntityID;
```

继续进入“字段”节点,如图 14-34 所示。该对话框允许在使用该数据集的所有报表中选择返回字段的名称(以便进行访问)。这里选择默认名称,即使已经选择,也可以在结果中改变名称。单击“确定”,数据集便创建成功。现在可以使用它对参数列表进行布局。

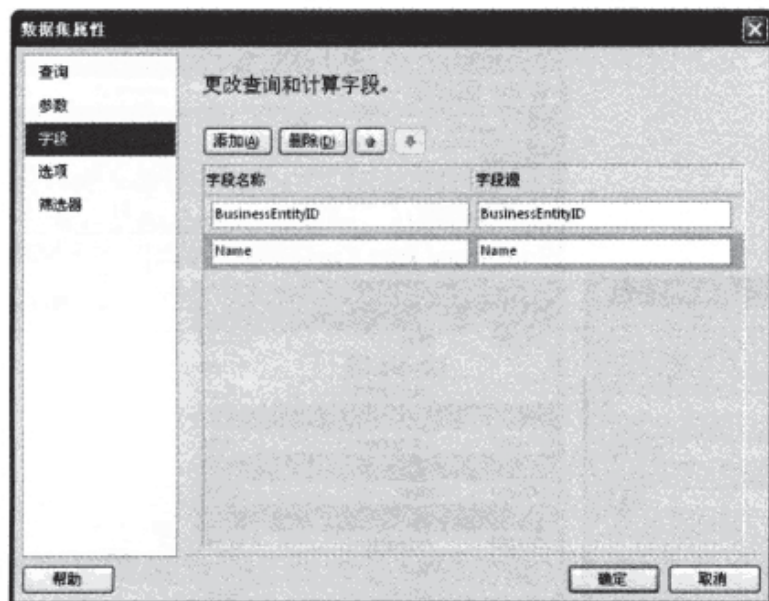


图 14-34

再次双击参数 BusinessEntityID, 将其打开以备编辑, 然后转到“可用值”节点, 如图 14-35 所示。这里又一次预先填入了合适的值。如你所想, 这里选择了“从查询中获取值”选项。同样还选择了使用什么数据集作为数据源以及哪些数据集的字段与值和标签字段相关(值和标签字段的功能将在手动为其赋值时体现出来)。在预览或执行报表之前(如图 14-36 所示), 还要进入“默认节点”节点, 设置一个默认值 283(即 David Campbell)。

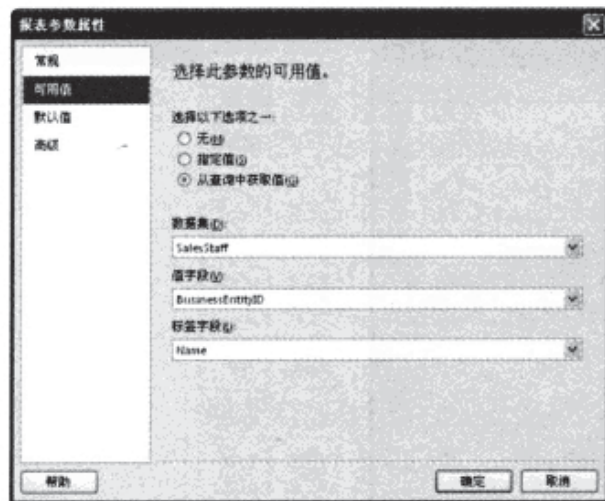


图 14-35

Sales Order Number	Category	Sales
9051089	Bikes	\$59,609
	Clothing	\$1,303
	Components	\$13,239
Total		\$74,150
9051093	Components	\$150
Total		\$150
9051097	Bikes	\$2,754
Total		\$2,754
9051123	Accessories	\$1,151
	Bikes	\$64,442
	Clothing	\$2,963
	Components	\$24,840
Total		\$93,396
9051128	Clothing	\$336
	Components	\$995
Total		\$1,331
9051163	Bikes	\$324
Total		\$324

图 14-36

这样一来, 参数默认值和数据类型便被迅速设置恰当。剩下来要做的就是处理固定表头的问题。

3. 通过参数获取表头和其他字段

通过编辑文本框来使用参数值相对简单。首先选中拥有现有固定值的文本框,进入编辑模式。要使其变为动态的,需要结合多个项。首先,要为动态值提供一个前缀,这里使用“Summary for:”。然后再次右击,选择“创建占位符”,进入如图 14-37 所示的对话框。占位符使得 Reporting Services 可以对文字文本和功能代码进行区分。注意,“值”字段有一个下拉框,将其展开可以在大量的动态值间进行选择。本例中为用户选择的某一个参数提供了一个参考值。继续并单击“确定”,可预览或运行报表查看效果(如图 14-38 所示)。

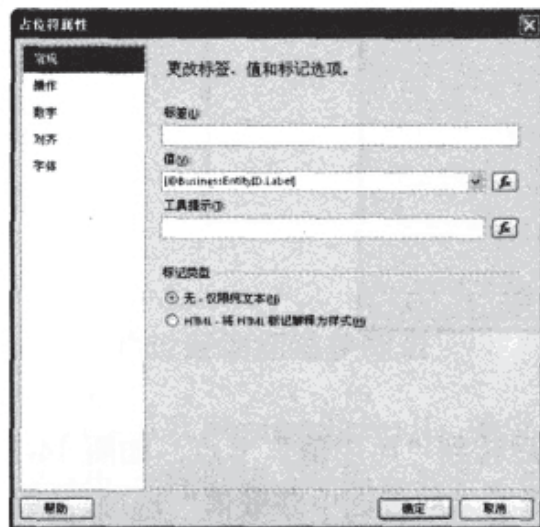


图 14-37

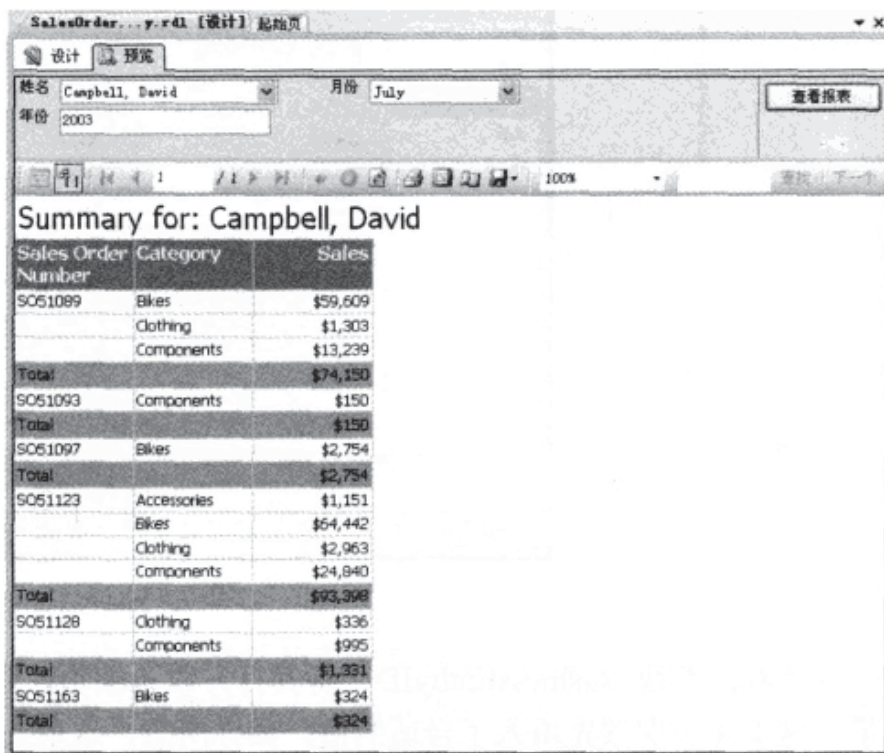


图 14-38

要完成这一小节的内容，还需再添加一两个占位符，但这一次，使用的是表达式编辑器。在刚刚创建的占位符后面添加一个逗号和一个空格，然后右击再次选择“添加占位符”。但是这一次，单击“值”字段右边的 Fx 按钮，进入如图 14-39 所示的对话框。在这幅图中，我正在为用户运行报表时所选择的 Month 参数添加一个参考值，但请注意在编辑时，Visual Studio 提供了 IntelliSense。继续为 Month 和 Year 都添加占位符，报表外观将如图 14-40 所示。

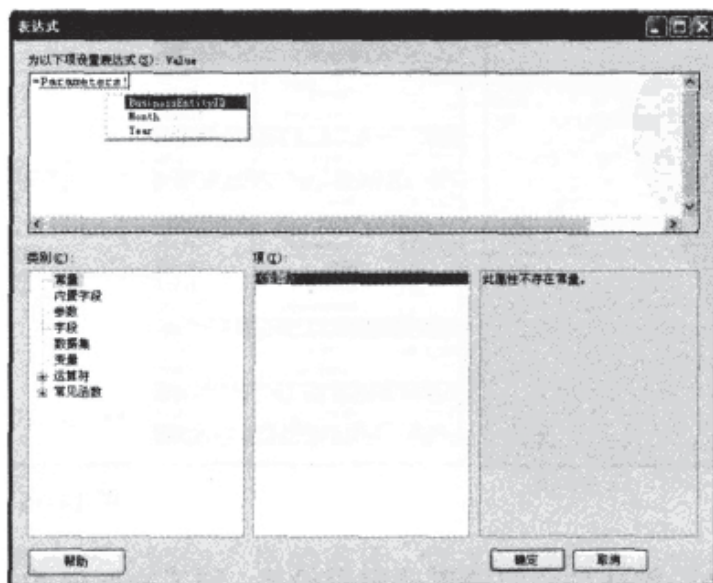


图 14-39

Summary for: Campbell, David, July, 2003

Sales Order Number	Category	Sales
S051089	Bikes	\$59,609
	Clothing	\$1,303
	Components	\$13,239
Total		\$74,150
S051093	Components	\$150
	Total	\$150
S051097	Bikes	\$2,754
	Total	\$2,754
S051123	Accessories	\$1,151
	Bikes	\$64,442
	Clothing	\$2,963
	Components	\$24,840
Total		\$93,398
S051128	Clothing	\$336
	Components	\$995
Total		\$1,331
S051163	Bikes	\$324
	Total	\$324

图 14-40

14.3.6 添加图表

Reporting Services 还支持图表对象。图表是一种功能较为强大的事物，因为它在很大程度上使得报表不仅仅是一份报告来源，而且还能够进行更真实的分析。下面要为报表添加一张图表，为本月各类产品的销售情况提供一个可视化表示形式。

首先打开 Visual Studio 的工具箱，将 Chart 对象拖入报表中(这里将它放在 tablix 的右边)。这将进入如图 14-41 所示的对话框，并允许在大量的图表类型中进行选择。

考虑到我们并没有太多类别可供选择，因此决定使用 3D 形式的饼型图表(如图 14-41 所示)。可将“报表数据”选项卡中的数据集中的字段直接拖入图表中的特殊接收区域(如图 14-42 所示)。我将 Sales 字段由 Dataset1 拖至“将数据字段拖至此处”区域，又将 Categories 字段拖入“将类别字段拖至此处”区域。



图 14-41

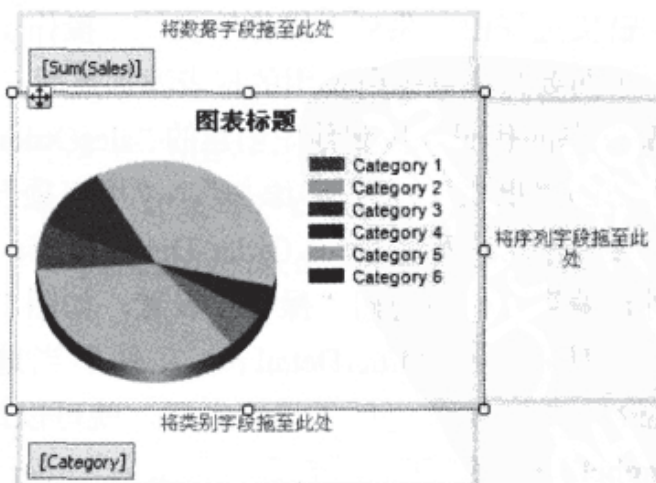


图 14-42

另外在图表属性中将标题区域改为 Sales by Category, 然后便可以再次运行或预览报表, 如图 14-43 所示。

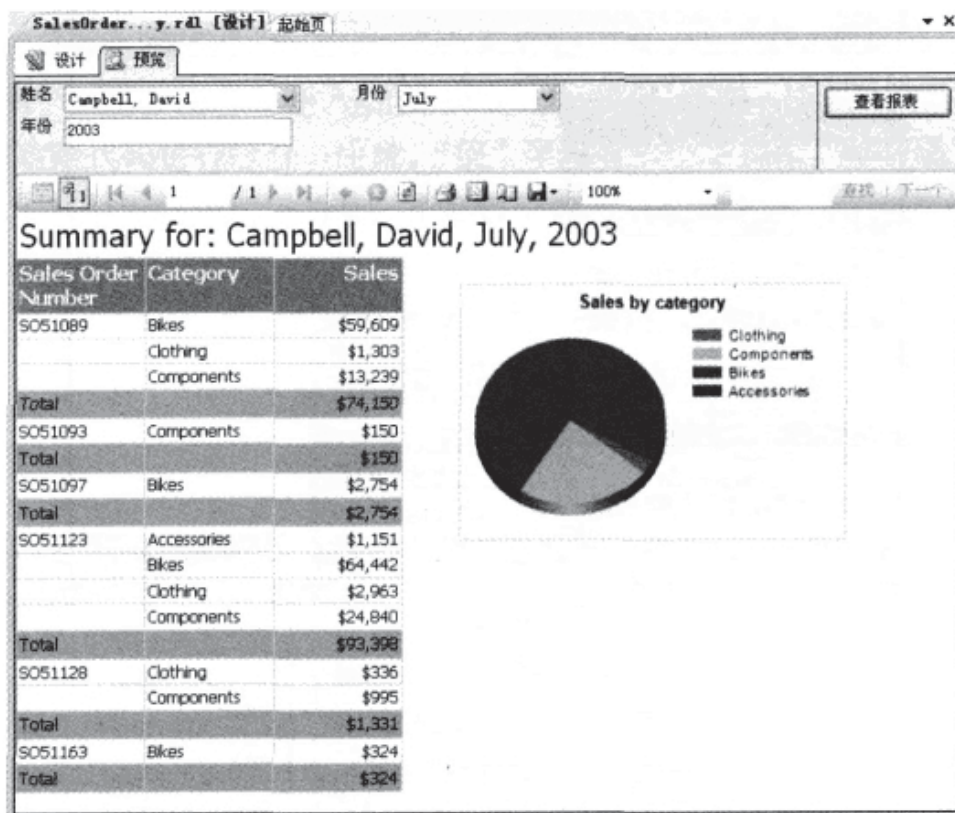


图 14-43

这样, 我们便迅速拥有了一张提供 tablix 数字的可视化表示的基本图表。

提示:

两个对象间不存在相互关联。它们只是碰巧使用了相同的数据集, 而并不是被要求这样做的。事实上, 在报表建立的大部分过程中, 都没有使用图表, 如果愿意, 也可以删除 tablix, 而只保留图表。

14.3.7 链接报表

Reporting Services 允许对多个报表进行链接, 既可以向下钻取到更深层的细节, 也可以钻取到完全不同的报表中。

链接过程由一系列“操作”支持。操作支持内部的(其他报表)和外部的(如一个网站)链接。

下面要为本章中所使用的报表添加最后一个要素。要利用这个链接, 需要下载(你可能还没有下载)本书的代码, 找到预先创建的 SalesOrderDetail.rdl 文件。可以通过右击“解决方案资源管理器”中的“报表”, 选择“添加”>“现有项”, 将该文件添加到项目当中。

要使用这一新的 Sales Order Detail 报表, 需要对报表中含有 Sales Order Number 的文本框属性进行编辑, 然后访问“操作”设置, 如图 14-44 所示。

一旦将 SalesOrderDetail.rdl 文件适当地添加到了项目中, 并对图 14-44 所示的 Sales OrderNumber 操作进行了设置之后, 就可以最后一次运行或预览摘要报表了。现在单击 David Campbell 在 2003 年 7 月的第一个 Sales Order Number, 就可以得到如图 14-45 所示的 Sales Order Detail 报表。

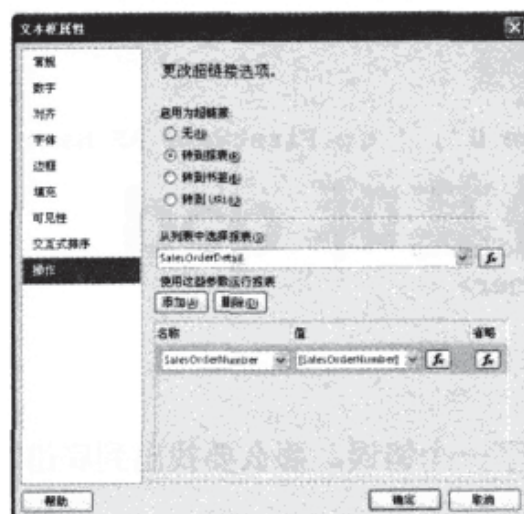


图 14-44

Qty	Product No.	Name	Shipping No.	Unit Price	Discount	Line Total
4	FR-M94B-42	HL Mountain Frame - Black, 42	A635-4BAF-BC	\$809.76	0.00	\$3,239.04
6	SE-M798	ML Mountain Seat/Saddle	A635-4BAF-BC	\$23.40	0.00	\$140.90
5	LJ-0192-X	Long-Sleeve Logo Jersey, XL	A635-4BAF-BC	\$29.99	0.00	\$149.97
2	BK-M188-44	Mountain-500 Black, 44	A635-4BAF-BC	\$323.99	0.00	\$647.99
5	BK-M395-38	Mountain-400-W Silver, 38	A635-4BAF-BC	\$461.69	0.00	\$2,308.47
1	PD-M562	HL Mountain Pedal	A635-4BAF-BC	\$48.59	0.00	\$48.59
4	SE-M236	LL Mountain Seat/Saddle	A635-4BAF-BC	\$16.27	0.00	\$65.09
2	SH-W890-M	Women's Mountain Shorts, M	A635-4BAF-BC	\$41.99	0.00	\$83.99
2	FR-M94S-38	HL Mountain Frame - Silver, 38	A635-4BAF-BC	\$818.70	0.00	\$1,637.40
4	FR-M21B-44	LL Mountain Frame - Black, 44	A635-4BAF-BC	\$149.87	0.00	\$599.50
2	GL-H102-S	Half-Finger Gloves, S	A635-4BAF-BC	\$14.69	0.00	\$29.39
7	BK-M585-46	Mountain-200 Silver, 46	A635-4BAF-BC	\$1,391.99	0.00	\$9,743.96
3	PD-M282	LL Mountain Pedal	A635-4BAF-BC	\$24.29	0.00	\$72.88
3	FR-M635-46	ML Mountain Frame-W - Silver, 46	A635-4BAF-BC	\$218.45	0.00	\$655.36
4	SE-M940	HL Mountain Seat/Saddle	A635-4BAF-BC	\$31.58	0.00	\$126.34
5	BK-M588-38	Mountain-200 Black, 38	A635-4BAF-BC	\$1,376.99	0.00	\$6,884.97
1	BK-M185-40	Mountain-500 Silver, 40	A635-4BAF-BC	\$338.99	0.00	\$338.99
2	BK-M395-40	Mountain-400-W Silver, 40	A635-4BAF-BC	\$461.69	0.00	\$923.39
6	BK-M585-42	Mountain-200 Silver, 42	A635-4BAF-BC	\$1,391.99	0.00	\$8,351.96
4	FR-M21B-48	LL Mountain Frame - Black, 48	A635-4BAF-BC	\$149.87	0.00	\$599.50
7	BK-M588-42	Mountain-200 Black, 42	A635-4BAF-BC	\$1,376.99	0.00	\$9,638.96
1	HB-M243	LL Mountain Handlebars	A635-4BAF-BC	\$26.72	0.00	\$26.72
5	BK-M585-38	Mountain-200 Silver, 38	A635-4BAF-BC	\$1,391.99	0.00	\$6,959.97
2	BK-M188-42	Mountain-500 Black, 42	A635-4BAF-BC	\$323.99	0.00	\$647.99

图 14-45

14.3.8 部署报表

剩下来要做的事就是对报表进行部署。在“解决方案资源管理器”中右击报表，选择“部署”。但是这里还有个小问题要注意——需要在项目定义中对部署的目标进行定义。

(1) 右击报表服务器项目，选择“属性”。

(2) 在 TargetReportFolder 字段中，设置登录到报表管理器时想驻留报表的文件夹。

(3) 在 TargetServerURL 字段中，输入报表服务器的 URL。在本例中，该 URL 很简单，为 <http://localhost/ReportServer>，但服务器名可以是有权限对其进行部署的任一服务器(如果在安装时作了定义，那除 Report Server 外，还可用 Virtual Directory)。

部署完成之后(通过右击项目选择“部署”)，便可以查看报表。导航到报表服务器(如果是在本地主机上并使用默认目录，则地址为 <http://localhost/Reports>)。单击报表文件夹，选择 SalesOrderSummary 报表。

首次加载报表可能需要一些时间，你所见到的报表和我们在项目中所定义的一样(如果回头再次导航一下，报表定义将被缓存，加载速度就会变得相当之快)。

14.4 有关 RDL 的简注

RDL 全称为 Report Definition Language——是一种基于 XML 的报表定义语言。在本章中我们对报表所做的所有改动都经由 Visual Studio 转换成了 RDL。如果想查看报表项目的 RDL，可右击报表选择“查看代码”。下面是我为本章所创建的一个示例报表的一段摘录。它定义了为销售人员的参数提供值的数据集。

```
<DataSet Name="SalesStaff">
  <Fields>
    <Field Name="BusinessEntityID">
      <DataField>BusinessEntityID</DataField>
      <rd:TypeName>System.Int32</rd:TypeName>
    </Field>
```



```

    <Field Name="Name">
      <DataField>Name</DataField>
      <rd:TypeName>System.String</rd:TypeName>
    </Field>
  </Fields>
</Query>
<DataSourceName>AdventureWorks2008</DataSourceName>
  <CommandText>SELECT p.BusinessEntityID, p.LastName ü', ' üp.FirstName AS Name
FROM Person.Person p
JOIN Sales.SalesPerson sp
ON p.BusinessEntityID = sp.BusinessEntityID;</CommandText>
  <rd:UseGenericDesigner>true</rd:UseGenericDesigner>
</Query>
</DataSet>

```

可以直接修改 RDL (但是要小心。如果在直接编辑时引入了一个错误, 那么要找出到底错在哪里将会十分困难)。

14.5 小结

Reporting Services 对许多 SQL Server 的安装而言都有重要影响。对许多公司而言, 在中央数据仓库中内置一个相对强大的报表服务器是一种解放, 使得向数据使用者散布信息变得更为简单。对另一些组织而言, Reporting Services 提供的解决方案足以替代长期以来所使用的报表包, 如 Crystal Reports。SQL Server 2008 添加了一些新的功能和控件, 使得报表更为精致和强大, 引擎也经过了重新设计, 可伸缩性更高。

尽管本章中所使用的报表较为可靠, 但也只是我们初尝到的一点甜头。报表可以被参数化, 可以嵌入图表, 可以与其他产品整合(如 Microsoft Sharepoint Services 或 Microsoft Office Sharepoint Services)、可以从一个报表钻取到另一个报表, 甚至可以在报表中再嵌入其他报表。

要了解报表的更多内容, 建议阅读有关 Reporting Services 的专门书籍。

第15章

bcp 和其他基本的大容量操作

如果你的系统将运行在某种美轮美奂的梦幻环境中，那或许可以跳过本章。遗憾的是，现实世界不会是这样的，因此，可能还是要看一下本章内容。

对于大部分系统来说，很多时候需要移动大块的数据。有时，你需要引入格式不正确的数据或者位于另一个应用程序的数据文件中的数据。有时需要直接从另一系统中提取数据。好在 SQL Server 提供了两个能帮助快速移动数据的工具——大容量复制程序(Bulk Copy Program, bcp)和 SQL Server Integration Services(SSIS)。本章将主要讨论第一个。另外，还将讨论与 bcp 类似的 BULK INSERT 命令和 OPENROWSET (BULK)。

提示：

下一章将研究 SSIS。

bcp 是我们所熟悉的，不过现在已很少见。很长一段时间以来，我们都用它作为移动大块数据的方法，并且其速度惊人(就目前来看，也仍是这样)。不过，它缺乏吸引力——坦白来说，从 SQL Server 7.0 版开始，它在大部分领域就没什么吸引力了。

那为什么我还要用一章来讲述它呢？这是由于 bcp 仍然有其用处。它的优点包括：

- 非常简洁
- 可以非常快速地移动大量数据
- 它是遗留下来的事物——也就是可能有已经运行着的代码有效地利用了它，因此没有理由改变它
- 使用隐秘但仍很传统的脚本风格(这对于一些人来说可能有吸引力)
- 一致性程度非常高

bcp 用于将文本和 SQL Server 本机格式数据传入和传出 SQL Server 表。在最近的几个版本中，bcp 几乎没什么改变，并且尽管其他大容量功能在不断地削弱 bcp 的地位，但它仍在发挥作用。可以把 bcp 看作是数据泵，除了尽可能高效地将数据从一个地方移至另一个地方外，它几乎没有其他功能。本章中将谈到的其他几种大容量操作通常更易于使用，但是常常也是以降低灵活性为代价的。

本章将详细讨论 bcp 的一些优缺点，然后以我们学到的有关 bcp 的知识为基础，来学习其他有着类似目的的多种功能——从而尽可能快地将数据导入导出系统。

15.1 bcp 实用工具

bcp 通过在操作系统命令行提示符下运行来导入或导出本机数据(特定于 SQL Server)、ASCII 文本或 Unicode 文本。这意味着可以从操作系统批文件、用户定义的存储过程以及其他地方来执行 bcp。bcp 还可以作为计划作业的一部分来运行,或者通过使用 shell 命令从 .NET 对象中执行。

和大部分命令行实用工具一样,可以使用连字号(-)或斜杠(/)来指定选项。不过,和大多数 DOS 或 Windows 家族的工具不同,选项开关是区分大小写的。

15.1.1 bcp 语法

```
bcp {[[<database name>.]<owner>.]<table name>|<view name>|"<query>" }
    {in | out | queryout | format} <data file>
    [-m <maximum no. of errors>] [-f <format file>] [-x] [-e <error file>]
    [-F <first row>] [-L <last row>] [-b <batch size>]
    [-n] [-c] [-w] [-N] [-V (60 | 65 | 70 | 80 | 90)] [-6]
    [-q] [-C <code page>] [-t <field term>] [-r <row term>]
    [-i <input file>] [-o <output file>] [-a <packet size>]
    [-S <server name>[\<instance name>]] [-U <login id>] [-P <password>]
    [-T] [-v] [-R] [-k] [-E] [-h "<hint> [,...n]"]
```

要理解的语法很多,因此我们将在表 15-1 中逐个讲述这些开关(不过好在这里的大多数开关都是可选的,因此通常只需包含进一小部分)。

注意:

用于 bcp 实用工具的许多开关都是区分大小写——通常,对于一个给定的字母,其大小写有着完全不同的含义。

表 15-1

参 数	说 明
<i>Database name</i>	和其名称的意思完全一样。这是四部分命名方案的标准部分。如果不进行指定,则使用用户的默认数据库
<i>owner</i>	四部分命名方案的一部分。同样,和其名称的意思一样
<i>Table 或 View name</i> "query"	只能是其中一个——表、视图或查询。 这是输入的目标表或视图,或者输出的源表或视图。 SQL Server 查询只能用作 bcp 输出的目标,并且只能在指定了 queryout 时使用。如果查询返回多个结果集,则 bcp 只使用第一个结果集
<i>in data file</i> <i>out data file</i> <i>queryout data file</i> <i>format data file</i>	同样,只能是其中一个。如果使用了其中的任何一个,还必须提供源或目标文件 确定 bcp 操作的方向。in 表示将数据从源文件导入到表或视图中;out 表示将数据从表或视图导出到目标文件中;只有在使用查询作为源并把数据输出到目标文件中时,才使用 queryout;使用 format 根据选定的格式选项创建格式化文件。还必须指定 -f, 以及格式选项(-n、-c、-w、-6、-C 或 -N),或者对 bcp 交互的提示作答 源或目标的路径和文件名以<data file>格式来指定,并且包含的字符不能超过 255 个

(续表)

参 数	说 明
-m <maximum errors>	可以指定在 SQL Server 取消大容量复制操作之前, 允许的错误的最大数目, 默认值是 10 个错误。bcp 无法复制的每一行都计为一个错误
-f <format file>	格式化文件中包含的是存储的响应, 这些响应来自于之前对同一个表或视图作的 bcp 操作。该参数应包含格式化文件的完整路径和文件名。这一选项主要与 in 和 format 选项一起使用, 以便在使用或创建格式化文件时指定路径和文件名
-x	生成基于 XML 的格式化文件, 而不是默认的文本形式的文件(虽然非 XML 形式的文件是遗留的支持, 但是现在仍是默认的)。该选项必须与 format 和-f 选项一起使用
-e <error file>	为错误文件指定完整路径和文件名, 以便存储 bcp 无法传输的所有行。否则, 不会创建错误文件。任何错误消息将显示在客户工作站上
-F first row	如果想要指定大容量复制操作将复制的第一行, 可以使用该选项。如果不指定该选项, 则 bcp 默认为 1, 并从源数据文件中的第一行开始复制。如果想要大块地进行加载, 使用该选项是非常便利的, 并且可以从之前的停止加载的地方开始加载
-L last row	该选项是对-F 的补充。它用来确定作为 bcp 执行的一部分, 想要加载的最后一行。如果不指定, 则 bcp 默认为 0, 即源文件中的最后一行。当与-F 一起使用时, 该选项允许一次加载一大块数据——小块地加载数据, 然后在上一次加载停止的地方继续
-b batch size	可以指定每批复制的行数。每批作为一个单独的事务进行复制。与所有事务一样, 每批中的行以“要么全部成功, 要么全部失败”的方式提交——即要么所有的行都被提交, 要么事务被回滚, 就好像整个批处理未发生过一样。-h(提示)开关有类似的选项(ROWS_PER_BATCH), 该选项与-b 互斥(两者都不使用或使用其中一个, 但不会一起使用二者)
-n	使用本机数据类型(SQL Server 数据类型)进行复制操作。使用该选项可以避开关于传输中要用到的数据类型的问题(它只是使用本机类型)
-c	该选项指定对所有字段使用字符数据(文本)进行操作, 因此不需要每个字段是单独的数据类型。如果没有使用-t 选项, 则使用制表符作为默认的字段分隔符, 并且如果没有使用-r 来指定行终止符, 则使用换行符作为行分隔符
-w	-w 选项和-c 选项类似, 但它为所有字段使用 Unicode 数据类型而不是 ASCII。同样, 如果没有用-t 和-r 选项予以覆盖, 则分别以制表符和换行符作为字段分隔符和行分隔符。该选项不能在 SQL Server 6.5 或更早的版本中使用
-N	该选项与-w 基本相同, 对于字符数据使用 Unicode 字符, 而对于非字符数据使用本机数据类型(数据库数据类型)。当将数据从一个 SQL Server 传送到另一个 SQL Server 时, 该选项能提供更高的性能。和-w 选项一样, 该选项不能在 SQL Server 6.5 或更早的版本中使用
-V(60/65/70/80)	让 bcp 使用 SQL Server 早期版本中可用的数据类型格式: 60 使用 6.0 数据类型、65 使用 6.5 数据类型、70 使用 7.0 数据类型、80 使用 2000 数据类型、90 使用 2005 数据类型。该选项替代了-6 选项
-6	该选项强迫 bcp 使用 SQL Server 6.0 或 6.5 数据类型。该选项与-c 或-n 一起使用只是为了向后兼容。对于 SQL Server 7.0 或更高版本, 应当使用-V

(续表)

参 数	说 明
-q	使用-q 指定表或视图名称包含非 ANSI 字符。该选项实际上为 bcp 使用的连接执行 SET QUOTED_IDENTIFIER ON 语句。完全限定名称、数据库、所有者和表或视图必须用双引号括起来, 就像这样的形式 "database name.owner.table "
-C <code page>	该选项用来指定数据文件中数据的代码页。该选项仅在 char、varchar 或 text 数据包含小于 32 或大于 127 的 ASCII 字符值时才适用。值为 ACP 的代码页指定 ANSI/Microsoft Windows(ISO 1252)。OEM 指定客户端使用的默认代码页。如果指定 RAW, 则不会发生代码页转换。另外, 还可以选择提供特定的代码页值。应尽可能避免使用该选项, 而在格式化文件中或被 bcp 问及时, 使用指定的排序规则
-t <field terminator>	该选项允许覆盖默认的字段终止符。默认的终止符是制表符。可以指定的终止符包括: 制表符(\t)、换行符(\n)、回车符(r)、反斜杠(\\)、空终止符(0)、任何可打印的字符或最长可达 10 个可打印字符的字符串。例如, 对逗号分隔的文本文件使用 "-t," 选项
-r <row terminator>	该选项与-t 选项类似, 不过它允许覆盖默认行终止符(而非字段终止符)。默认的终止符是换行符\n, 而规则在其他方面与-t 相同
-i <input file>	可以用 input file 来指定响应文件, 该文件包含在交互模式下执行 bcp 操作时要使用的响应(这样就不必回答大量的问题)
-o <output file>	可以把 bcp 输出从命令提示行重定向到输出文件。当从无人参与的批处理或存储过程执行 bcp 时, 该选项可用于捕获命令输出和结果
-a <packet size>	可以指定在网络上传输的数据包的默认大小。如果有良好的网络质量(几乎没有 CRC 错误), 较大的数据包会带来较好的性能。指定的值必须位于 4096~65535 之间(包括 4096 和 65535), 并且该值将覆盖为服务器设置的默认值。安装时, 默认的包大小为 4096 字节。可以使用 SQL Server Management Studio 或 sp_configure 系统存储过程来覆盖该值
-S <server name>	如果从服务器上运行 bcp, 默认的服务器是本地 SQL Server。使用该选项可以指定一个不同的服务器, 并且如果从网络上的远程系统运行 bcp, 必须使用该选项
-U <login name>	如果没有通过可信连接与 SQL Server 进行连接, 则必须提供用于登录的有效用户名
-P password	在提供用户名时, 还必须提供密码。否则, bcp 命令将提示输入密码。如果-P 是最后一个选项, 并且没有提供密码, 则意味着指定密码为 null
-T	可以通过可信连接使用网络用户凭据连接到服务器。如果指定了可信连接, 则不需要为连接提供登录名或密码
-v	如果使用了该选项, bcp 将返回版本号和版权信息
-R	使用该选项指定在复制货币、日期和时间数据时, 使用客户端区域设置中定义的区域格式。默认忽略区域设置
-k	在大容量复制过程中, 使用该选项覆盖列的默认值, 忽略任何默认约束。空列将保留空值, 而不是列的默认值
-E	当导入的源文件包含标识列值时会在导入期间使用该选项, 该选项本质上等同于 SET IDENTITY_INSERT ON。如果没有指定, 则 SQL Server 将忽略源文件中提供的值并自动生成标识列值。如果是从不包含标识值的源文件中导入数据, 可以使用格式化文件来忽略标识列, 并让 SQL Server 产生标识值

(续表)

参 数	说 明
-h "hint[,...]"	通过此选项可以指定大容量复制操作使用的一个或多个提示。SQL Server 6.5 或更早的版本不支持-h 选项
ORDER <i>column</i> [ASC DESC]	如果源数据文件的排序次序与目标表中的聚集索引匹配,可以使用该提示提高性能。如果目标表没有聚集索引,或者数据按不同的次序排序,则忽略 ORDER 提示
ROWS_PER_BATCH= <i>nn</i>	该提示可以代替-b 选项,指定每批中传送的行数。不要将该提示与-b 选项一起使用
KILOBYTES_PER_BATCH= <i>nn</i>	对于每批中要传送的数据,可以选择以千字节计的近似值来指定其大小
TABLOCK	该提示将导致操作期间获得表级别的锁。默认锁定行为由表选项 table lock on bulk load 设置
CHECK_CONSTRAINTS	默认情况下,在导入操作期间将忽略 CHECK 约束。使用该提示强制在导入期间检查 CHECK 约束
FIRE_TRIGGERS	和 CHECK_CONSTRAINTS 类似,该选项使得目标表上的所有触发器在事务执行期间激发。默认情况下,触发器在大容量操作期间不会激发。SQL Server 2000 之前的版本不支持该选项

bcp 在交互模式下运行,执行命令时如果没有指定-f、-c、-n、-w、-6 或-N,则会提示格式信息。当在交互模式下运行时,bcp 在收到格式信息后,还会提示创建格式化文件。

15.1.2 bcp 导入

到目前为止,我们一直在讲预备知识。接下来将开始具体讨论 bcp。

bcp 最常见的用途可能就是把大量数据导入到已有的 SQL Server 表和视图中。要导入数据,必须拥有对服务器的访问权限(通过登录 ID 或者通过可信连接获得),并且必须在目标表或视图上拥有 INSERT 和 SELECT 权限。

源文件可包含本机代码、ASCII 字符、Unicode 或混合的本机和 Unicode 数据。要记住使用合适的选项描述源数据。另外,为了让数据文件可用,必须说明字段终止符和行终止符(使用-t 和-r),或者分别用默认的制表符和换行符来终止字段和行。

在开始前必须了解目的地。bcp 有一些会影响数据导入的怪异行为。为时间戳列或计算列提供的值将被忽略。如果源文件中有这些列的值,它们将被忽略。如果源数据文件不包含这些列的值,则需要使用格式化文件(我们将在本章后面看到),这样会忽略这些列。

提示:

在使用的软件中,这是你不时都会碰到的怪异行为之一。这种情况下,如果目标表中包含这些列,那么即使 SQL Server 将忽略这些数据,也还必须包含代表时间戳或计算数据的列。这好像很傻。但可以使用格式化文件明确地表明要忽略这些列,以此来解决该问题。

对于 bcp 操作来说,规则是被忽略的。如果没有指定 FIRE_TRIGGERS 和/或 CHECK_CONSTRAINTS 提示,所有的触发器和约束都将被忽略。唯一约束、索引以及主键/外键约束是强制的。如果不指定-k 选项,将强制使用默认约束。

1. 数据导入的示例

了解 bcp 导入如何工作的最简单的方法是通过例子。首先是一个简单的例子，是一个以制表符分隔的文件，该文件包含 AdventureWorks2008 数据库的部门信息。数据如下所示：

```
1 Smart    Guys    Research and Development 2006-04-01 00:00:00.000
2 Product Test Research and Development 2006-04-01 00:00:00.000
```

要在本地服务器上使用可信连接把这些数据导入到 Department 表中，可运行下列命令：

```
BCP AdventureWorks2008.HumanResources.Department in c:\DepartmentIn.txt -c -T
```

提示：

这里有两个重要的事情：第一，到目前为止，我们运行的所有事情都是在 Management Studio 中进行的。不过，对于 bcp 来说，是在命令提示符框中输入命令。第二，需要修改前面的命令行，以便与你下载的本书的示例文件(或数据)相匹配。

由于 Department 表中的第一列是标识列，并且没有指定-E 选项，因此 SQL Server 将忽略文件中的标识值并生产新的值。-c 选项将源数据指定为字符数据，而-T 选项指定使用可信连接。

提示：

如果没有使用 Windows 身份验证，并且没有在 SQL Server 中建立有适当权限的网络登录名，那么可能需对上例进行修改以使用-S 和-P 选项。

在执行时，SQL Server 很快告知了我们关于大容量操作进行情况的一些基本信息：

```
2 rows copied.
Network packet size (bytes)    : 4096
Clock Time (ms.) Total        : 109 Average : (18.35 rows per sec.)
```

我们可以回到 Management Studio，并确认数据像预期的那样进入了 Department 表中：

```
USE AdventureWorks2008;

SELECT * FROM HumanResources.Department;
```

结果就是返回了几行——最重要的是要看到我们预期通过 bcp 操作导入的那两行：

DepartmentID	Name	GroupName	ModifiedDate
1	Engineering	Research and Development	1998-06-01...
2	Tool Design	Research and Development	1998-06-01...
...			
...			
16	Executive	Executive General and Administration	1998-06-01...
17	Smart Guys	Research and Development	2006-04-01...
18	Product	Test Research and Development	2006-04-01...

提示：

和往常一样，除了我们刚导入的两行之外，你的数据看起来会有些不同，这取决于你在本书

的哪些部分运行过示例，哪些部分没有运行过示例，以及你又自行进行过什么操作。对于本例而言，我们只是想看到 Smart Guys 和 Product Test 以适当的信息复制到了表中。标识值将根据特定服务器上的下一个值来重新分配。

接下来看一个更复杂的示例。假设有一个名为 CustomerList 的表。用于创建表 CustomerList 的 CREATE 语句如下所示：

```
CREATE TABLE dbo.CustomerList
(
    CustomerID nchar(5) NOT NULL
        PRIMARY KEY,
    CompanyName nvarchar(40) NOT NULL,
    ContactName nvarchar(30) NULL,
    ContactTitle nvarchar(30) NULL,
    Address nvarchar(60) NULL,
    City nvarchar(15) NULL,
    Region nvarchar(15) NULL,
    PostalCode nvarchar(10) NULL,
    Country nvarchar(15) NULL,
    Phone nvarchar(24) NULL,
    Fax nvarchar(24) NULL
);
```

我们有一个以逗号分隔的文件(其格式和.csv 文件一样)，该文件中有新顾客的信息。这次，文件看上去像下面这样：

```
XWALL,Wally's World,Wally Smith,Owner,,,,,(503)555-8448,,
XGENE,Generic Sales and Services,Al Smith,,,,,(503)555-9339,,
XMORE,More for You,Paul Johnston,President,,,,,(573)555-3227,,
```

注意：

源文件中的这些逗号是干什么用的？这些逗号是 CustomerList 表中列的占位符。源文件没有为所有的列提供值，因此使用逗号来忽略这些列。对于源文件没有为所有的列提供值的问题，处理方法并非只有这一种。你可以使用格式化文件，将源文件与目的位置进行匹配。本章稍后将讨论格式化文件。

假设运行 bcp 把数据导入到远程系统中。命令如下所示：

```
BCP AdventureWorks2008.dbo.CustomerList in c:\newcust.txt -c -t, -r\n -Ssocrates -
Usa -Pbubbagump
```

提示：

这里显示的换行是为了让命令行更易于阅读。如果你自己试验该示例时，不要按回车进行换行。将这些命令作为单独一行那样输入，让其在命令提示符中自行换行即可。

再次将数据指定为字符数据。-t 选项将文件指定为以逗号分隔(终止)的数据，-r\n 选项把换行符指定为行终止符。另外，这次提供的服务器连接信息也稍有些变化，以 sa 作为登录名，以 bubbagump 为密码。

bcp 同样确认了传输，并给出基本信息：

Starting copy...

3 rows copied.

Network packet size (bytes): 4096

Clock Time (ms.) Total : 15 Average : (200.00 rows per sec.)

我们将再次确认数据已经像预期的那样到达目的地:

```
USE AdventureWorks2008;
```

```
SELECT CustomerID, CompanyName, ContactName
```

```
FROM dbo.CustomerList
```

```
WHERE CustomerID LIKE 'X%';
```

结果当然是所有数据都在那里。

CustomerID	CompanyName	ContactName
XGENE	Generic Sales and Services	Al Smith
XMORE	More for You	Paul Johnston
XWALL	Wally's World	Wally Smith

2. 记录日志与不记录日志

bcp 既可以以快速模式(不记录日志)运行,也可以以缓慢模式(记录日志)运行。每种方式都有其优点。快速模式可提供最好的性能,但慢速模式可提供最大限度的可恢复性。由于慢速模式要记录日志,因此可以在导入之后立即进行一次快速事务日志备份,这样就能在出现故障时恢复数据库。

如果需要传输大量数据,快速模式常常是最好的选择。快速模式不仅传输速度更快,而且由于操作不记录日志,因此不用担心事务日志耗尽空间。如果要以不记录日志的方式运行 bcp,必须满足以下几个条件:

- 不能复制目标表;
- 如果目标表有索引,则该表当前不得包含任何行;
- 如果目标表中已经有行,则该表上不得包含索引;
- 指定了 TABLOCK 提示;
- 目标表上必须没有触发器;
- 对于 SQL Server 2000 之前的版本,必须把 select into/bulkcopy 选项设置为 true。

显然,如果要在拥有数据的索引表中进行快速模式复制,必须做到:

- 删除索引;
- 删除任何触发器;
- 运行 bcp;
- 对目标表重建索引;
- 重新创建所有的触发器。

完成了不记录日志的 bcp 操作后,必须立即备份目标数据库。

如果目标表不满足进行快速 bcp 的条件,则操作将被记录。这意味着在传输大量数据时,可

能要面临事务日志被填满的风险。可以使用 WITH TRUNCATE_ONLY 选项运行 BACKUP LOG 来清除事务日志。TRUNCATE_ONLY 选项只截断日志中不活动的部分，而不备份任何数据。

提示：

要强调的是，bcp 操作对于事务日志的大小可能是致命的。如果不能实现以最小化方式记录的操作，那需要考虑把批的大小调整得小一些，并且在操作期间打开 TRUNCATE ON CHECKPOINT。另一个解决方案是使用 -F 和 -L 选项，以便每次把事情作为一个块来处理，并截断每块数据之间的日志。不过要知道，为了实现正确还原数据库，备份策略的一个重要部分即事务日志现在是缺失的。因此一旦大容量加载活动完成，就要创建最新备份。

15.1.3 bcp 导出

如果你准备通过大容量操作接受数据，那么你可能就想到要把数据提取出来。

bcp 允许从表、视图或查询中导出数据。你必须指定一个目标文件名。如果文件已经存在，则它将被覆盖。与导入操作不同，在导出时不允许忽略列。时间戳、rowguid 和计算列导出的方式与任何其他 SQL Server 列一样(就好像它们是“真正的”数据一样)。要导出，必须在源表上拥有适当的 SELECT 权限。

让我们看两个使用 AdventureWorks2008 数据库中的 HumanResources.Department 表的简单示例。

要使用默认格式将数据导出到数据文件，可以运行下列命令：

```
BCP AdventureWorks2008.HumanResources.Department out c:\somedir\
DepartmentOut.txt -c -T
```

提示：

如果运行的是 Vista 版本或更高版本的 Windows 操作系统(包括 Windows Server 2008)，新的安全控制可能会阻止对根目录(在大部分系统中是 C:\)进行大容量操作——因此我在前面的代码中使用了 somedir。

这将创建如下的文件：

```
1 Engineering Research and Development 1998-06-01 00:00:00.000
2 Tool Design Research and Development 1998-06-01 00:00:00.000
...
...
17 Smart Guys Research and Development 2006-04-01 00:00:00.000
18 Product Test Research and Development 2006-04-01 00:00:00.000
```

提示：

在这里，我们不必使用格式化文件，系统也不会显示提示信息要求你提供字段长度等——选项 -c 表明只希望所有内容(无论什么类型)都以默认的格式按基本的 ASCII 文本那样导出。默认以制表符作为字段分隔符，以换行符作为行分隔符。

记住，如果目的文件已经存在，它将被覆盖。这将在没有任何类型的提示或警告的情况下发生。

如果要将分隔符改为某种自定义字符，可以运行如下代码：

```
BCP AdventureWorks2008.HumanResources.Department out DepartmentOut.txt -c -T -t,
```

提示：

最后的逗号不是输入错误。t后面的字符是字段分隔符——这里是逗号。

运行结果如下所示：

```
1,Engineering,Research and Development,1998-06-01 00:00:00.000
2,Tool Design,Research and Development,1998-06-01 00:00:00.000
...
...
17,Smart Guys,Research and Development,2006-04-01 00:00:00.000
18,Product Test,Research and Development,2006-04-01 00:00:00.000
```

我们使用逗号分隔符代替制表符，这样实际就得到了一个.csv文件。

15.2 格式化文件

如果你之前曾接到一些文件并被要求将其加载到数据库中，那在看过我之前给出的导入示例后可能会说“希望数据是实际完全格式化的”等等。确实，数据形式很少会像我们所期望的那样，这就有了格式化文件的概念。

上一节中提到了格式化文件。我们可以把格式化文件视为某种导入模板，在下列情况下，除了其他因素外，它使得更容易进行重复的导入操作：

- 源文件与目标表的结构或顺序不匹配；
- 想要跳过目标表中的列；
- 文件中包含的数据使默认数据输入和排序规则难以实现或不可行。

格式化文件有两种类型：非 XML 和 XML。首先讨论“老”方法(非 XML 版本)，然后再讨论新的 XML 格式化文件。

为了更好地了解每种格式化文件是如何工作的，我们来看几个特定的例子。首先看一下当源和目标匹配时如何构建文件。接下来可以把这种情况与下列情况进行比较：源文件中的字段数与表列数不匹配，或者源文件中的字段的排列顺序与表列不同。

你可以创建一个默认的格式化文件(为了向后兼容，该文件是非 XML 的)，当在交互模式下运行 BCP 时，可将此文件用作源。在提示输入列值信息后，可以选择保存该文件。其默认的文件名是 bcp.fmt，但是也可以用任何有效的文件名来命名它。

如果要像这样为 AdventureWorks2008 数据库的 HumanResources.Department 表创建默认的格式化文件，可以运行下列命令：

```
BCP AdventureWorks2008.HumanResources.Department out c:\somedir\ department.txt -T
```

这是一种快速创建格式化文件的便利方式，之后你可以在需要的时候对其进行编辑。由于可以对任何表这样做，因此，可以先使用 bcp 创建格式化文件。

对每一个文件都采用默认的前缀和数据长度信息，在这里，以逗号作为字段分隔符。SQL

Server 将在你输入了所有的格式信息后，提示你保存格式化文件；这里，我将把它保存为 Department.fmt。然后，可以使用任何文本编辑器(例如 Windows Notepad)编辑这个格式化文件，使其满足特定需求。

我们来看一下刚才生成的格式化文件：

```
10.0
4
1      SQLSMALLINT 0 2      ","      1 DepartmentID      ""
2      SQLNCHAR    2 100    ","      2 Name
SQL_Latin1_General_CP1_CI_AS
3      SQLNCHAR    2 100    ","      3 GroupName
SQL_Latin1_General_CP1_CI_AS
4      SQLDATETIME 0 8      ","      4 ModifiedDate      ""
```

文件中的前两行说明了 bcp 的版本号(10.0 表示 SQL Server 2008, 9.0 表示 SQL Server 2005 等)和宿主文件中字段的数目。剩下的行描述宿主数据文件，以及字段如何与目标列和排序规则相匹配。

第一列是宿主文件字段号。从 1 开始编号，一直到字段的总数目。接着是宿主文件数据类型。此示例文件中混合了几种数据类型。所有的文本都使用 Unicode 格式，因此所有字段的数据类型都是 SQLNCHAR。如果此数据中这里没有特殊的字符，那我们可以只使用 SQLCHAR(ASCII) 格式。

接下来的两列说明了数据字段的前缀和数据长度。前缀是字段中前缀字符的数目。前缀描述了实际 bcp 文件中数据的长度，并允许将数据文件的大小进行压缩。数据字段是存储在字段中数据的最大长度。接着是字段终止符(分隔符)。在本例中，逗号用作字段终止符，并且用换行符作为行终止符。再往后的两个列通过提供服务器列顺序和服务列名来描述目标表列。本例中，由于服务器列和宿主字段之间是直接匹配的，列和字段号是相同的，因此不必像这样来指明。最后但并非最不重要的是每列的排序规则(记住，在 SQL Server 2000 和更新的版本中，表中的每一列可以有不同的排序规则)。

现在，我们来讨论 XML 版本。要创建 XML 版本的格式化文件，可以使用几乎一样的命令，但是增加了 -x 开关：

```
BCP AdventureWorks2008.HumanResources.Department out c:\somedir\department.txt -T-x
```

得到的格式化文件看上去完全不同：

```
<?xml version="1.0"?>
<BCPFORMAT
xmlns="http://schemas.microsoft.com/sqlserver/2004/bulkload/format"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <RECORD>
    <FIELD ID="1" xsi:type="NativeFixed" LENGTH="2"/>
    <FIELD ID="2" xsi:type="NcharPrefix" PREFIX_LENGTH="2" MAX_LENGTH="100"
COLLATION="SQL_Latin1_General_CP1_CI_AS"/>
    <FIELD ID="3" xsi:type="NcharPrefix" PREFIX_LENGTH="2" MAX_LENGTH="100"
COLLATION="SQL_Latin1_General_CP1_CI_AS"/>
    <FIELD ID="4" xsi:type="NativeFixed" LENGTH="8"/>
  </RECORD>
</ROW>
```



```

<COLUMN SOURCE="1" NAME="DepartmentID" xsi:type="SQLSMALLINT"/>
<COLUMN SOURCE="2" NAME="Name" xsi:type="SQLNVARCHAR"/>
<COLUMN SOURCE="3" NAME="GroupName" xsi:type="SQLNVARCHAR"/>
<COLUMN SOURCE="4" NAME="ModifiedDate" xsi:type="SQLDATETIME"/>
</ROW>
</BCPFORMAT>

```

可注意到, 在这里所有的内容都被明确地标示出来。另外, 有一个与 XML 格式化文件相关的 XML 模式文档, 这意味着可以在你选择的 XML 编辑器中验证 XML。

提示:

我并打算进行辩解, 我就是钟爱新的 XML 版本的格式化文件。对我而言, 如果不必担心向后兼容 SQL Server 2005 之前的版本, 使用它是毫无疑问的。

尽管旧式的格式化文件也能起作用, 但是, 每当我大量使用它们时, 我都要考虑如何缓解痛苦。因为必须做默认之外的所有事情, 那是很令人头痛的。与它们有关的所有事情都必须是“井井有条”, 在较大的表中, 由于没有清晰地分隔开字段, 很容易遗漏打字错误。XML 标记可解决所有这些问题, 并能明确每一个小条目的作用——调试要容易得多。

15.2.1 如果列不匹配

如果这个世界是完美无暇的, 而且我们接收到的数据文件总是正好和表类似, 那该有多好。

但是还是不要太理想化。我对于眼下的世界还算满意, 但它并不完美, 需要进行大容量操作的数据文件类型很少与目标表类似。那么, 如果源数据文件与目标表未按照我们所希望的方式匹配, 该怎么做呢? 或者反过来将表转换为预期的与之完全不同的数据文件格式, 又该怎么做呢?

幸运的是, 格式化文件可帮助处理源与目标数据之间可能存在的几种不同的差异。接下来就来看看具体情况。

1. 文件中的列比表中的列少

首先讨论数据文件中的字段比目标表中的列少的情况。对于这种情况, 需要修改已经在使用的格式化文件, 以标识哪些列在数据文件中不存在, 相应地要忽略表中的哪些列。实现这一目的方法是把每个缺少字段的前缀长度和数据长度设置为 0, 并把要跳过的列的表列顺序号设置为 0。

例如, 假定数据文件中只有 DepartmentID、Name 和 GroupName, 则要把该文件修改为:

```

10.0
4
1      SQLSMALLINT    0  2      "," 1 DepartmentID      ""
2      SQLNCHAR       2 100    "," 2 Name
SQL_Latin1_General_CP1_CI_AS
3      SQLNCHAR       2 100    "," 3 GroupName
SQL_Latin1_General_CP1_CI_AS
4      SQLDATETIME    0  0      "," 0 ModifiedDate      ""

```

正如你所见, ModifiedDate 字段和列都以 0 表示。由于没有提供 ModifiedDate, 并且该列有默认值(Getdate()), 因此, 对插入的行将使用默认值。

XML 版本看上去没有太大不同, 但对于定义的元素不是用 0 表示, 只是未作定义:

```
<?xml version="1.0"?>
<BCPFORMAT xmlns="http://schemas.microsoft.com/sqlserver/2004/bulkload/
format"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <RECORD>
    <FIELD ID="1" xsi:type="NativeFixed" LENGTH="2"/>
    <FIELD ID="2" xsi:type="NCharPrefix" PREFIX_LENGTH="2" MAX_LENGTH="100"
COLLATION="SQL_Latin1_General_CP1_CI_AS"/>
    <FIELD ID="3" xsi:type="NCharPrefix" PREFIX_LENGTH="2" MAX_LENGTH="100"
COLLATION="SQL_Latin1_General_CP1_CI_AS"/>
  </RECORD>
  <ROW>
    <COLUMN SOURCE="1" NAME="DepartmentID" xsi:type="SQLSMALLINT"/>
    <COLUMN SOURCE="2" NAME="Name" xsi:type="SQLNVARCHAR"/>
    <COLUMN SOURCE="3" NAME="GroupName" xsi:type="SQLNVARCHAR"/>
  </ROW>
</BCPFORMAT>
```

文件中没有列要定义，因此就没有定义。在 **ModifiedDate** 列中不粘贴任何东西，因此我们跳过该列(依赖该列的默认值)。

2. 文件中的列比表中的列多

“数据文件中的列比表中的列多”这种情况实际上与前面讨论过的“数据文件中的列较少的情况”很相似。这里的唯一诀窍是，必须为额外的字段添加列信息，但前缀长度、数据长度以及列顺序号都设置为 0：

```
10.0
4
1      SQLSMALLINT 0    2    "," 1 DepartmentID      ""
2      SQLNCHAR    2    100 "," 2 Name
SQL_Latin1_General_CP1_CI_AS
3      SQLNCHAR    2    100 "," 3 GroupName
SQL_Latin1_General_CP1_CI_AS
4      SQLDATETIME 0    8    "," 4 ModifiedDate      ""
5      SQLDATETIME 0    0    "," 0 CreatededDate      ""
```

这一次，宿主文件中包括了部门的创建日期字段。目标表中没有接收这一信息的列。这一字段被添加到最初的格式化文件中，并加入两个列顺序号为 0 的伪列。这将强制 **bcp** 忽略这些字段。

对于这种情况，XML 版本必须处理文件中有需要说明的列的事实。不过，对于目标表而言，可以继续忽略该字段：

```
<?xml version="1.0"?>
<BCPFORMAT xmlns="http://schemas.microsoft.com/sqlserver/2004/bulkload/
format"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <RECORD>
    <FIELD ID="1" xsi:type="NativeFixed" LENGTH="2"/>
    <FIELD ID="2" xsi:type="NCharPrefix" PREFIX_LENGTH="2" MAX_LENGTH="100"
COLLATION="SQL_Latin1_General_CP1_CI_AS"/>
    <FIELD ID="3" xsi:type="NCharPrefix" PREFIX_LENGTH="2" MAX_LENGTH="100"
```



```

COLLATION-"SQL_Latin1_General_CP1_CI_AS"/>
<FIELD ID-"4" xsi:type-"NativeFixed" LENGTH-"8"/>
<FIELD ID-"5" xsi:type-"NativeFixed" LENGTH-"8"/>

</RECORD>
<ROW>
  <COLUMN SOURCE-"1" NAME-"DepartmentID" xsi:type-"SQLSMALLINT"/>
  <COLUMN SOURCE-"2" NAME-"Name" xsi:type-"SQLNVARCHAR"/>
  <COLUMN SOURCE-"3" NAME-"GroupName" xsi:type-"SQLNVARCHAR"/>
  <COLUMN SOURCE-"4" NAME-"ModifiedDate" xsi:type-"SQLDATETIME"/>
</ROW>
</BCPFORMAT>

```

3. 字段顺序不匹配

另一种可能的情况是：宿主数据文件与目标表有相同的字段，但是字段顺序不匹配。在这种情况下，可以修改服务器列顺序，使之与宿主数据文件中的字段顺序相匹配：

```

10.0
4
1          SQLSMALLINT 0      2      "," 1 DepartmentID      ""
2          SQLNCHAR    2      100    "," 3 GroupName
SQL_Latin1_General_CP1_CI_AS
3          SQLNCHAR    2      100    "," 2 Name
SQL_Latin1_General_CP1_CI_AS

4          SQLDATETIME 0      8      "," 4 ModifiedDate      ""

```

在这里，源数据文件中组名在部门名之前列出。我们对服务器列顺序进行了修改，以反映这一事实。注意，服务器列的列出顺序并没有改变，只是交换了列的顺序号。

对于 XML 版本来说，只需对最初的 XML 文件改动一两个字段即可：

```

<?xml version-"1.0"?>
<BCPFORMAT xmlns-"http://schemas.microsoft.com/sqlserver/2004/bulkload/
format"
xmlns:xsi-"http://www.w3.org/2001/XMLSchema-instance">
  <RECORD>
    <FIELD ID-"1" xsi:type-"NativeFixed" LENGTH-"2"/>
    <FIELD ID-"2" xsi:type-"NCharPrefix" PREFIX_LENGTH-"2" MAX_LENGTH-"100"
COLLATION-"SQL_Latin1_General_CP1_CI_AS"/>
    <FIELD ID-"3" xsi:type-"NCharPrefix" PREFIX_LENGTH-"2" MAX_LENGTH-"100"
COLLATION-"SQL_Latin1_General_CP1_CI_AS"/>
    <FIELD ID-"4" xsi:type-"NativeFixed" LENGTH-"8"/>
  </RECORD>
  <ROW>
    <COLUMN SOURCE-"1" NAME-"DepartmentID" xsi:type-"SQLSMALLINT"/>
    <COLUMN SOURCE-"3" NAME-"Name" xsi:type-"SQLNVARCHAR"/>
    <COLUMN SOURCE-"2" NAME-"GroupName" xsi:type-"SQLNVARCHAR"/>

    <COLUMN SOURCE-"4" NAME-"ModifiedDate" xsi:type-"SQLDATETIME"/>
  </ROW>

```

</BCPFORMAT>

15.2.2 使用格式化文件

让我们来看一个使用格式化文件进行导入的例子。该命令将基于名为 `shortdept.txt` 的文件，把记录复制到 `Department` 表中。我们将使用 `ShortDept.fmt` 作为非 XML 格式化文件的例子，使用 `ShortDeptX.fmt` 作为 XML 格式化文件。

```
BCP AdventureWorks2008.HumanResources.Department in c:\shortdept.txt -  
fc:\shortdept.fmt -Usa -Pbubbagump
```

提示：

为了做一点变化，上面的示例命令行使用了 SQL Server 身份验证，而不是 Windows 身份验证。如果你想使用 Windows 身份验证，只需用前面常使用的 `-T` 参数替换上面的 `-U` 和 `-P` 参数即可。

本例中使用的示例文件 `ShortDept.txt`、`ShortDept.fmt` 和 `ShortDeptx.fmt` 可从 Wrox 网站或 ProfessionalSQL.com 下载。

15.2.3 尽量提高导入性能

最大化 bcp 性能的一个显而易见的方法是，确保目标表满足以不记录日志操作的方式执行 bcp 的所有要求。这意味着必须：

- 删除目标表上任何已有的索引。尽管这仅当你想要实际以最小化方式记录日志的操作时才是必需的，但在大容量操作期间停用索引而不管日志的状态是可以极大地提高性能的。不过，要确保在完成了大容量操作后重建索引。
- 尽管以与聚集索引(如果有的话)相同的顺序来创建源数据文件。在重建索引时，就可使用 `SORTED_DATA_REORG` 选项，该选项可大大加快索引的创建速度(从而缩短 bcp 操作的总时间)。甚至如果必须保留聚集索引，在对排好序的数据执行 BCP 时可使用 `ORDER` 列选项(在 `-HINT` 选项中)。
- 确保维护属性设置为简单或不记录日志。如果设置为完全恢复模式，则将不允许 bcp 以最小方式记录日志操作。

如果想在向表中导入数据时获得额外的性能提升，可以从多个客户端执行数据并行加载。要实现这一点，必须：

- 使用 `TABLOCK` 提示。
- 删除所有索引(可以在操作完成后重建索引)。
- 将服务器恢复选项设置为 `Bulk-Logged`。

这是如何实现的呢？这不是导入一个很大的文件，而是将其分解为多个较小的文件。然后，从多个客户系统启动 bcp，每个客户系统导入较小文件中的其中一个。显然，只有在这样的导入带来的性能提升所节省下来的时间多于准备源数据文件并将它们复制到客户端所花费的时间时，你才会考虑这样做。

SQL Server 6.5 或更早的版本不支持并行加载。

注意:

不管使用这些操作中的哪一种, 在操作完成之后, 都必须在目标表上重建所有索引。要在重建任何非聚集索引之前重新创建目标表的聚集索引(如果有的话)。

通过让 SQL Server 忽略 CHECK 约束和触发器(默认选项), 可以获得额外的性能提升。要记住, 这样做可能会导致加载违反表的 CHECK 约束和任何由触发器强制的数据完整性规则的数据。

15.3 BULK INSERT

本章开始提到过一个与 BCP 类似的大容量操作命令, 那就是 BULK INSERT 命令。要使用该命令, 必须是 sysadmin 或 bulkadminserver 角色的成员。

BULK INSERT 操作在本质上类似于可以直接在 T-SQL 中使用的限制版 bcp。其语法如下所示:

```
BULK INSERT [['<database name>'.][ '<schema name>'.] '<table name>'] FROM '<data file>'
    [WITH
    (
        [BATCHSIZE [ = <batch size>]]
        [, CHECK_CONSTRAINTS]
        [, CODEPAGE [= {'ACP'|'OEM'|'RAW'|'<code page>'}]]
        [, DATAFILETYPE [= {'char'|'native'|'widechar'|'widenative'}]]
        [, FIELDTERMINATOR [= '<field terminator>']]
        [, FIRSTROW [= <first row>]]
        [, FIRE_TRIGGERS]
        [, FORMATFILE = '<format file path>']
        [, KEEPIDENTITY]
        [, KEEPNULLS]
        [, KILOBYTES_PER_BATCH [= <no. of kilobytes>]]
        [, LASTROW [ = <last row no.>]]
        [, MAXERRORS [ = <max errors>]]
        [, ORDER ({column [ASC|DESC]} [ ,...n ] )]
        [, ROWS_PER_BATCH [= <rows per batch>]]
        [, ROWTERMINATOR [ = '<row terminator>']]
        [, TABLOCK]
        [, ERRORFILE = '<file name>']
    )
    ]
```

现在, 如果你感到似曾相识, 那么说明你确实熟练掌握了其要点。几乎所有这些开关都在本章开始讲述的基本 bcp 导入语法中有对等物。

BULK INSERT 的特殊权限要求有些麻烦(不是所有的人都属于 sysadmin 或 bulkinsert), 但是, 它也的确带来了几个显著优点:

- 它可作为使用 BEGIN TRAN 和相关语句的用户定义事务的一部分执行;
- 它在 SQL Server 进程内运行, 由于避免了编组, 因此性能得到了一些提高;
- 它比 bcp 使用的命令行语法略易理解些。

有关 BULK INSERT 的最大问题就是它是大容量插入。BULK INSERT 将不会帮助你创建格

式化文件。它不会帮助导出数据。它只是获得 bcp 功能从而在 SQL Server 中把数据移入数据库的一种简单可行的方法。

15.4 OPENROWSET (BULK)

还有另一个与 bcp 类似的大容量操作,不过该操作与 bcp 关系较远。OPENROWSET (BULK) 将大容量行集提供程序与 OPENROWSET 的能力结合在一起,当用于查询中时可快速并且相对灵活地访问外部文件,而不必把它们加载到中间表中。

bcp 更常见的用途之一是加载外部数据文件供周期性的进程使用。例如,你可能接收到这样的文件,这些文件中包含信用报告、供应商目录以及其他由供应商以一般格式表示的数据。这些是对于你而言至关重要的信息,但与真正导入这些数据相比,你更感兴趣的可能是与这些数据作一次性的交互。OPENROWSET (BULK)允许将那样的文件(或仅仅是文件的一部分)当作一个表来处理。而且,与简单的链接表所能提供的转换相比,它可以利用格式化文件为文件布局提供更好的转换。其语法如下所示:

```
OPENROWSET
( BULK '<data file>' ,
  [ [ FORMATFILE = '<format file>' ] [
    [, CODEPAGE [= ('ACP'|'OEM'|'RAW') '<code page>']]
    [, FIRSTROW [= <first row>]]
    [, LASTROW [ = <last row no.>]]
    [, MAXERRORS [ = <max errors>]]
    [, ROWS_PER_BATCH [= <rows per batch>]]
    [, ERRORFILE = '<file name>']
  ]
  | SINGLE_BLOB | SINGLE_CLOB | SINGLE_NCLOB )
)
```

要记住, OPENROWSET 更像是一种大容量访问的方法,而非插入方法。当数据源是 OPENROWSET 时,肯定可以进行 INSERT INTO 操作(实际也常常是这样使用它的),但是, OPENROWSET 还可以更灵活。在了解了这些后,接着来讨论有关 OPENROWSET 的两个重要的大容量选项方面的问题。

15.4.1 ROWS_PER_BATCH

这一选项易让人误解。要记住的最重要的事情是,如果使用该选项,那么实际上是在为查询优化器提供一个提示。SQL Server 将总是处理整个文件,但是如果为该选项指定了值,那该值将成为提供给优化器的提示,说明文件中有多少行。尽量使值准确或者干脆忽略该值。

15.4.2 SINGLE_BLOB、SINGLE_CLOB 或 SINGLE_NCLOB

这些选项说明要把整个文件当作一个事物来对待——只有单个列的单个行。这种类型就像是 varbinary(max)。如果使用 SINGLE_BLOB, 将应用 Windows 编码约定。SINGLE_CLOB 将假定数据是 ASCII 数据,而 SINGLE_NCLOB 将假定数据是 Unicode 数据。

15.5 小结

本章中，我们讲述了两个重要的数据导入/导出实用工具中的其中一个——bcp。bcp 主要用来把存储为文本文件的数据导入和导出 SQL Server。另外，我们还讲述了两个与 bcp 类似的大容量操作方法。

作为一种遗留工具，大多数使用过的 SQL Server 的人对 bcp 还是熟悉的。由于微软公司还在继续增强 bcp 背后的核心技术，所以我敢肯定 bcp 还将继续保留。

虽说如此，bcp 常常并不是最佳的选择。一定要用 BULK INSERT(和在 SQL Server 进程内运行所带来的好处)以及 OPENROWSET (BULK)来确定你的选择。

在接下来的一章中，我们将讨论 bcp 的主要竞争对手——SQL Server Integration Services(SSIS)。虽然 SSIS 拥有 bcp 所缺乏的魅力和光环，但是它也有其自身的怪异之处，这有时就使得 bcp 的简单性看起来更有吸引力。

第16章

开始集成

SQL Server Integration Services——或 SSIS——是从另一个叫做数据转换服务——或 DTS 的工具演化而来的。记住 DTS 之所以如此重要，一个特别的原因是在它被发布的那个时代(在 1999 年早期作为 SQL Server 7.0 的一部分)，它是一个具有革命性的工具。这个工具能够移动和转换大型数据块，在这之前，主流关系数据库管理系统(RDBMSs)都不曾提供类似的工具。各种各样原本非常困难或者需要十分昂贵的第三方工具才能实现的任务顿时变得相对容易起来。随着 SQL Server 2008 时代的快速到来，现在的 SSIS(在 SQL Server 2005 中，名称已经发生了改变，同时也彻底重写了这个服务)相对来说仍然比较单一，其目的是为了使用户能够更容易地使用这样一个重要的工具。

本章将讲述如何执行基本的数据导入和导出，并且将讨论如何使用诸如 Integration Services 之类的工具来完成其他一些任务。本章将把主要精力放在对 SSIS 包的基础知识的介绍上，更高级的关于 SSIS 编程的内容将在第 25 章(这是一个在 Web 上发布的章节)中给出。

16.1 理解问题

大多数系统中都存在 Integration Services 所解决的问题——如何从外部数据源中获取数据或者如何把系统数据填充到外部数据源中。这可能会是把旧系统中的数据导入到新系统中，或者从一个厂商或其他来源的可用项列表中导入数据。然而，所有这些情形的共同点是要把在不匹配的表中定义的数据加入到表中。

这里用户所需的是一个能够提取、转换并把数据加载到数据库中的工具——通常把这样的工具简称为“ETL”。不同工具能够处理的问题的复杂程度是不一样的，但是 SQL Server Integration Services——或 SSIS——几乎能够处理用户可能遇到的所有情形。

提示：

这也许会引出这样一个问题，“好吧，既然这个工具是内置的，那为什么不是所有人都使用它呢？”在跨平台的环境中，这个问题的答案就显得非常直观了。因为有一些第三方软件包无缝融合性更好，并且具有更友好的用户界面。实际上，这些工具使得无经验的用户能够相对更加容易地移动数据——同样，它们也是非常昂贵的。在旧的 DTS 产品流行的年代，有些 Oracle 或其他

DBMS 的用户仅仅为了使用 DTS 而购买了完整的 SQL Server 许可。虽然竞争产品的价格在下降同时 SQL Server 许可的价格在上涨,但是笔者确信仍然会有很多用户会购买 SQL Server 的许可,因为他们需要使用 SSIS。

16.2 包的综述

SSIS 使用了“包”的概念,一个包包含了一组要做的动作,每个动作叫做一个“任务”。用户可以把一系列的任务集中起来,甚至可以使用控制流选择在不同的条件下按照指定的顺序运行不同的任务(例如,如果一个任务失败,则运行一个不同的任务)。用户可以用编程的方式创建包(使用一个相当健壮的对象模型,第 25 章将介绍这个内容),但是大多数包的初始设计都是在 SQL Server 提供的设计器中完成的。

为了熟悉一下环境,接下来读者将会创建一个简单的包。为了使用 SSIS,需要启动 SQL Server Business Intelligence Development Studio,可以单击“所有程序”|“Microsoft SQL Server 2008”——然后选择“Integration Services 项目”作为项目类型,如图 16-1 所示。

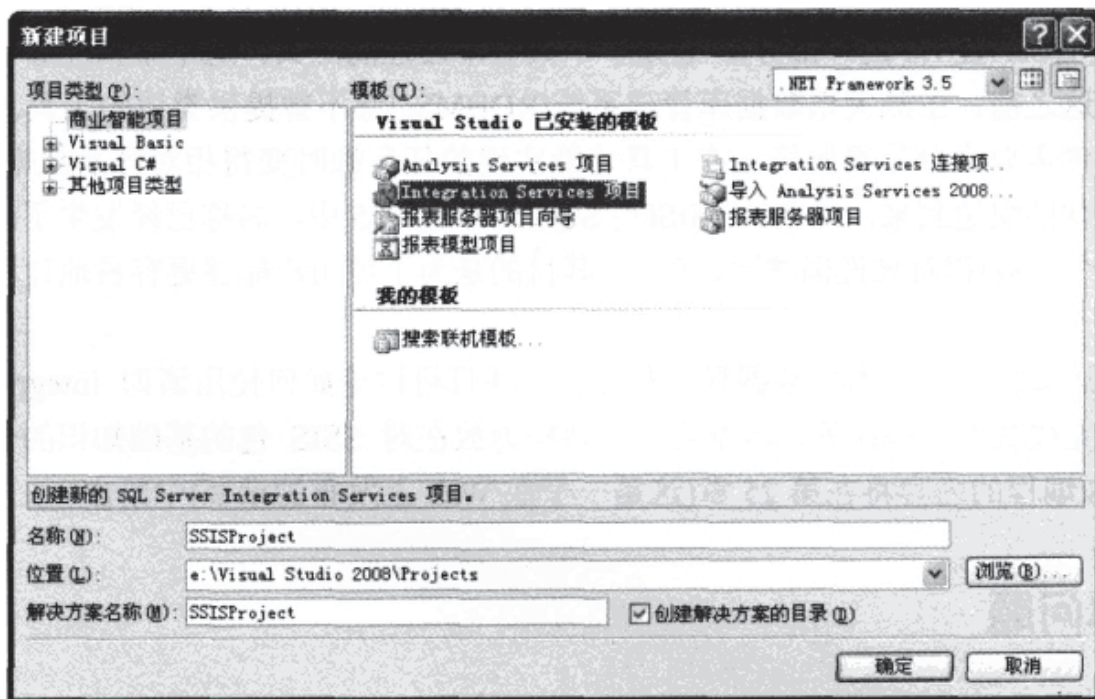


图 16-1

提示:

老实说,笔者认为把 Integration Services 建模器从 Management Studio 中移到 Intelligence Studio 中实在很愚蠢。虽然如此,Microsoft 并没有在移动他们的工具之前咨询用户的习惯,因此用户必须得忍受他们的行为。

要重申的是 SSIS 工具位于 Business Intelligence Studio 中(与报表服务相关的项目很像),而不是在 Management Studio 中,而之前讨论过的大多数项目都是在 Management Studio 中的。

注意:

读者看见的对话框可能会和图 16-1 中显示的对话框有些出入,这取决于你是否安装了 Visual Studio,并且即使你安装了 Visual Studio,也要取决于你安装了哪些组件。

本例中项目的名称“SSISProject”是非常具有描述性的——然后只要单击“确定”，接着 SQL Server 会创建这个项目，并且会为 SSIS 相关的项目打开默认的项目窗口，如图 16-2 所示。

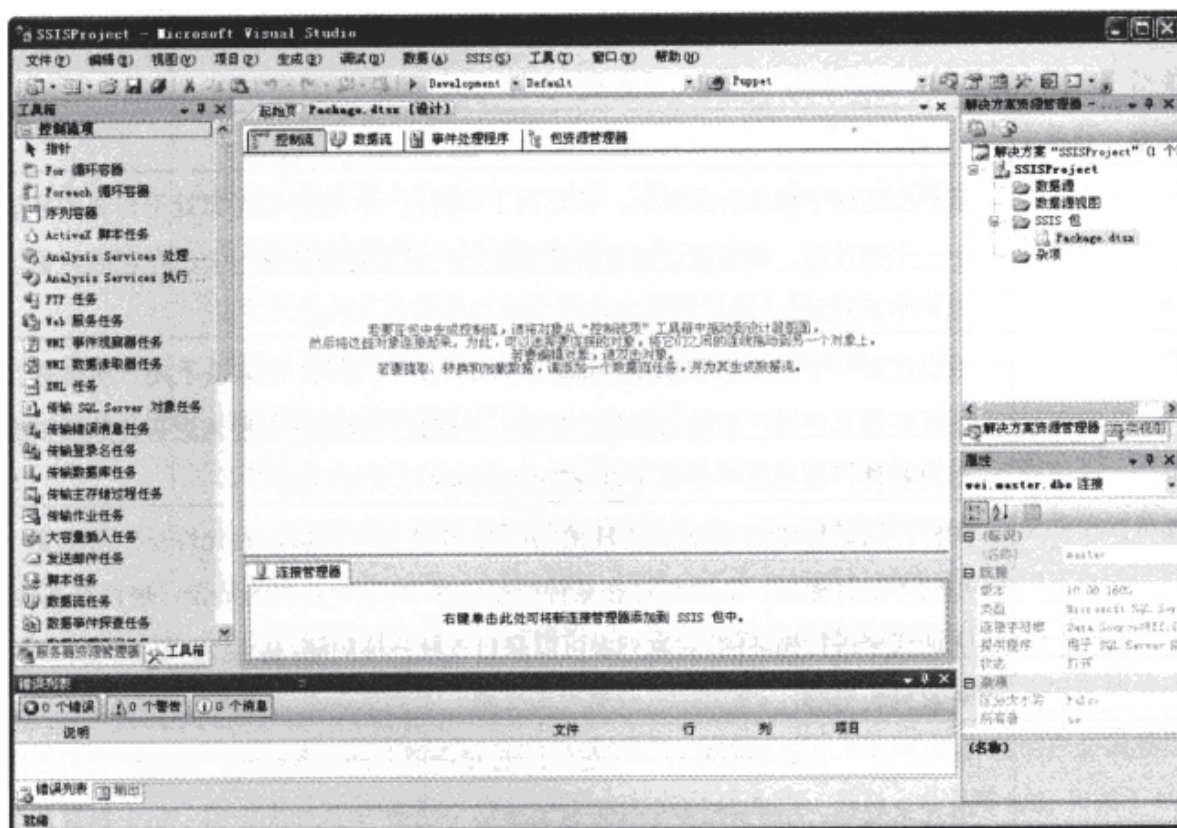


图 16-2

熟悉 Visual Studio 环境的读者相对来说会对这个环境熟悉一点，这个环境与大多数 Dev Studio 项目仅有的最大的差别在于当用户构建项目时，“设计”选项卡会以图形化的方式显示，而不会以代码的方式显示出来。

在这个项目中有 4 个重要的窗口，接下来将从介绍这些窗口开始，接着在本章后面会介绍一个实例。

16.2.1 任务

项目的左边(这取决于用户的设置，用户可能需要单击这个选项卡来展开它)是工具箱窗口。工具箱的最顶端是“控制流项”列表，是首先可以看到的，向下滚动滚动条后应该可以看到“维护计划”任务(管理员可能会更多地关注这些任务，但是读者需要注意它们——它们强调了前面提到过的 Integration Services 不仅仅关注 ETL 活动，同时还提供了很多其他的动作，包括很多读者可能认为在 Management Studio 中能够找到的动作)。注意很多“控制流项”条目被标记为了“任务”。

顾名思义，一个任务通常是指用户需要执行的一个动作。包括迁移任务(如在服务器之间移动对象)、数据迁移和转换、以及管理其他程序或包的执行等。尽管大多数动作都叫做任务，但是还存在一些容器对象，它们可以帮助用户组织或封装包中的其他对象。

提示：

需要注意的是用户可以重新组织任务。例如，用户可以拖放任务列表中任务来对它们重新排序(可能会把最常用的任务移到最顶端，这样就能更容易地找到这些任务)，同时用户也可以创建自己的选项卡以包含那些最常用的任务。此外，就像用户可以向其他 Dev Studio 项目中添加新控

件一样，用户也可以向列表中添加新任务。简而言之，开发环境是相当容易定制的。

任务的数量是相当庞大的，表 16-1 将快速介绍一下各个基本任务的功能。

表 16-1

任 务	说 明
指针	虽然描述这个概念有些多余，但是为了以防万一：指针让对象处于普通的拖放模式。当选 中一个指针后，单击设计器窗格意味着用户只是想要选择一个已存在的对象，而不是要添 加一个新对象
For 循环容器	这仅仅是一个美化的 FOR 语句(或者 FOR/NEXT 语句，这取决于用户所选择的语言)。FOR 循环容器允许用户初始化控制计数器，并根据所设置的条件来更新计数器以及设置循环的 退出条件。使用这个任务可以受控地重复执行其他任务
Foreach 容器	同样，这是普通的 FOR/EACH 语句。与 FOR 循环类似，它允许受控的重复，但是这里使 用的不是计数器，而是通过在某种类型的集合(也许是表的集合，也许是其他对象的集合) 上迭代来进行循环的。对象列表可以来自各种各样的源，从 ADO 和 ADO.NET 行集到 SMO 对象列表等等
序列容器	可以把序列容器看成某种“子包”，它允许用户把任务分组，然后把它们当成一个独立的 单元来处理。当需要把多个任务包装进单个事务时(从而允许整个包中包含几个不同的事务 ——每个事务可能需要执行多个任务)，序列容器是非常有用的 可以根据特定的条件激活或禁用单个序列容器，因此用户可以通过禁用某个序列容器来关 闭整个任务集(用户甚至可以根据前面任务返回的结果以编程的方式来激活或禁用某个序 列容器)
脚本任务	顾名思义，脚本任务使用户可以使用任何 ActiveX 脚本语言(JavaScript 或 VBScript)或基 于 .NET 的语言来运行自定义的代码。对于 ActiveX 语言则可以使用 ActiveX 脚本任务，而 对于 .NET 代码则可以使用脚本任务
Analysis Services 任务	这个任务允许用户构建或修改 Analysis Services 对象以及执行它们
大容量插入任务	正如读者所猜测的那样，这个任务允许用户大容量导入数据。它所使用的大容量插入工具 与读者在 bcp 章节中接触到的工具是一样的，但是这里允许大容量插入成为更大的控制流 的一部分。对于 SSIS 包而言，要把数据导入到系统，大容量插入任务无疑是最快速的方法。 但是需要注意的是，只有当登录的用户属于系统管理员服务器角色的成员时，才能运 行包含大容量插入任务的包
数据流任务	数据流任务封装了数据源之间的连接和在数据源之间移动数据时所需进行的转换。数据流 任务是 SSIS 包中最复杂的任务之一，因为它同时作为任务和容器来操作。数据流任务把 一个给定数据流的几个部分与它联系起来，从这种意义上来说数据流任务是一个容器。此 外，数据流任务还定义了数据的源和目的地以及在源和目的地之间发生的转换。编辑数据 流任务将自动打开主编辑窗口中一个单独的选项卡
数据挖掘查询任务	这个任务要求用户已经在 Analysis Services 中定义了数据挖掘模型。用户可以使用这个任 务运行预测性查询并且把其结果输出到表中(接着用户可以定义其他使用这些表的任务)
执行任务	这些任务特定于用户所需执行的任务，它们可以是运行其他包(运行旧的 DTS 包和运行新 的 SSIS 包是不同的任务)，也可以是运行外部程序，或者还可以运行 SQL 脚本

(续表)

任 务	说 明
文件系统任务	这些任务允许用户创建、移动和删除文件以及目录。在各种 SSIS 环境中, 传送文件的能力对于包的性能和执行是至关重要的。例如, 出于性能上的考虑, 用户可能需要把一个文件从远程某个位置复制到本地存储上, 因为用户需要在这个文件上执行一些操作。同样, 用户的网络接入可能只允许读取或创建文件, 但是不允许修改文件——文件系统任务可以使用户能够顺利完成所需完成的任务
FTP 任务	这个任务与文件系统任务在概念上稍微有些不同。它允许用户使用 FTP 协议来检索文件(当需要传输文件到厂商、客户或其他合作伙伴, 或者要从厂商、客户或其他合作伙伴传输文件时该任务是非常方便的)
消息队列任务	这个任务允许用户通过 Microsoft 消息队列来发送和接收消息。它确实是一个非常强大的工具, 允许用户在远程主机当前未联机的情况下递送或接收文件以及其他消息。此时, 用户可以把文件加入到一个队列中, 这样当下次主机联机时就可以通知它有可用的文件。同样也可以把留在队列中, 这样当用户执行包时相应的进程就可以收取文件了
发送邮件	是的, 这又是一个“像听上去那样”的任务。它允许用户指定包含附件的邮件, 这个附件可能是在之前执行包的过程中创建的。该任务唯一麻烦的地方是用户必须要指定用来发送邮件的 SMTP 连接(一般来说是出站邮件服务器)。此外, 也支持 SSL 和基于 Windows 的身份验证
传输任务	这些任务可以是诸如传送登录帐户、错误消息以及主数据库的存储过程之类的服务器迁移任务, 也可以是更简单的诸如传送一张表之类的传送任务
Web 服务任务	这个任务允许用户执行 Web 服务方法并把结果保存到一个变量中。然后用户可以在包中剩余的任务中使用那个结果
WMI 任务	Windows Management Instrumentation(WMI)是允许用户对系统进行监控和控制的一组 API。它是基于 Web 的企业级管理(WBEM)在 Windows 平台上的特定实现, 而 WBEM 则是用于访问系统信息的行业标准。SSIS 中包含了监控 WMI 事件(这样用户可以知道何时在系统上发生了特定的事件)的任务和以 WMI 查询的方式向 WMI 请求数据的任务。例如, 用户可以询问 WMI 服务器上总的系统内存是多少
XML 任务	XML 任务允许各种各样的 XML 操作。用户可以应用 XSLT 转换、合并文档、使用 XPath 筛选 XML 文档和列表等等
维护任务	很多与这类任务有关的内容都超出了本书的范围, 但是这类任务允许用户在服务器上执行各种各样的维护任务。从开发者的角度来看, 此处的关键用途是在进行重大的导入或其他类似的活动之前进行一次备份, 其中这些活动属于包的一部分。同样地, 对于需要在特定表上进行重大操作的任务, 用户可能会想要在任务执行完成之后重建索引或进行其他的维护工作

16.2.2 主窗口

在 Dev Studio 中, 该窗口处于默认的 SSIS 包窗口布局的中心。需要注意的是在这个窗口中有四个可用的选项卡, 每一个选项卡都有各自的用途——下面逐一介绍它们。

提示:

需要注意的是,用户可以把默认的选项卡风格界面改为基于窗口的界面(在 Visual Studio 的选项菜单中)。

1. 控制流

实际上,这是包的中心内容的聚集地。虽然一个包不会仅仅由流构成,但是这里是用户一开始拖进所有任务并建立任务的执行顺序的地方。

2. 数据流

当用户把数据流对象放入控制流窗格时,就可以在数据流窗格中进一步定义它们了。数据流任务需要额外的对象以定义数据连接、数据源和数据目的地以及实际转换等事项。

3. 事件处理程序

SSIS 包在执行的时候会产生大量的事件,这个选项卡允许用户捕捉特定的事件并对这些事件做出响应。表 16-2 列出了一些值得捕捉的重要事件:

表 16-2

事 件	说 明
OnError	这是 SSIS 的一个激动人心的新特性。尽管 DTS 有一个类似的错误处理程序,但是功能比较弱,而现在的处理程序则要健壮很多
OnExecStatusChanged	每当任务进入不同的状态时都会引发这个事件。可能的状态有空闲、正在执行、异常终止、完成、挂起和验证。用户可以为每个条件设置捕获并运行相应的代码
OnPostExecute	当一个任务的执行结束后将会立即引发这个事件。从理论上讲,这个事件与引发 OnExecStatusChanged 事件是一样的,只不过其状态是变为完成,但是需要指出的是笔者对此所做的测试不够充分,因此无法断言
OnProgress	当包的执行取得任何适当的可度量的进展时就会有规律地引发这个事件。当用户以程序的方式控制包时可能比使用其他某种执行方法更有用,但是从为最终用户提供进度条(如果需要的话)的观点来看,其他执行方法也是非常有用的

还存在其他一些可用的事件方法,但是上面的介绍已经使读者有了某种程度的认识。

4. 包资源管理器

笔者认为包资源管理器的放置位置有些奇怪。简单地说,包资源管理器给出了包的树形结构和所有的事件处理程序、连接以及可执行文件(包含了在包中定义的所有任务)。之所以说包资源管理器的有些奇怪是因为笔者原本期望它会是解决方案资源管理器的一部分,或者至少与解决方案资源管理器类似。虽然如此,它的确给用户提供了一种在整个包的级别查看项目的方式。

16.2.3 解决方案资源管理器

这个窗口与 Dev Studio 中的其他资源管理器是相当类似的, 用户可以获取所有属于该解决方案的文件列表, 同时该窗口会根据这些文件的属性(如包和数据源视图)对它们进行分组。

16.2.4 属性窗口

这个窗口与读者在 SQL Server 和 Dev Studio 中看到的其他属性窗口几乎是一样的。这里唯一需要注意的是需要弄清楚究竟选择了哪个对象, 这样才能知道现在正在为哪个对象设置属性。如果选择了包中的某个对象, 那么出现的属性窗口应该是属于那个特定任务或事件对象的属性窗口。如果没有选中任何对象, 那么应该出现整个包的属性窗口。

16.3 创建简单的包

现在是时候通过一些实例来深入理解前面讲述的概念了, 虽然这里给出的实例有些简单, 但是本章后面会介绍 SSIS 的几个关键功能。

首先要做一些准备工作, 本例将使用一个 vbScript 文件来生成一些数据, 后面会导入这些数据。读者可以把这些脚本看成对任何预处理脚本的一种模拟, 在进行重大的导入和导出之前都需要运行预处理脚本。

使用下面的代码创建一个名为 CreateImportText.vbs 文件:

```
Dim iCounter
Dim oFS
Dim oMyTextFile

Set oFS = CreateObject("Scripting.FileSystemObject")
Set oMyTextFile = oFS.CreateTextFile("C:\TextImport.txt", True)

For iCounter = 1 to 10
    oMyTextFile.WriteLine(Chr(iCounter) & vbTab & """"TestCol" &
    Chr(iCounter) & """"")
Next

oMyTextFile.Close
```

当执行这个脚本时将会创建一个新的文本文件(如果该文件存在, 那么将会替换现有的文件)。它将向文件中添加 10 行文本, 每行文本中包含了两个以制表符分隔的列以及一个作为行终止符的换行符。本例将通过此文件以及其他一些任务在 SQL Server 中创建和填充表。

注意:

CreateImportText.vbs 文件默认会在 C 盘的根目录下创建一个文本文件。如果读者的系统上启用了用户访问控制, 那么系统可能会阻止运行这个脚本。如果出现这种情况, 那么只需要把这个文件移到根目录之下的某个目录中就可以了, 注意对本例后面给出的路径需要做相应的调整。

提示:

这是一个相当简单的实例, 但是请耐心看完。最后希望读者能够了解这样一个概念: 先运行

某种预处理(在本例中是一个用来生成文件的脚本,但是它可以是任何类型的脚本或外部进程),然后运行 SQL Server 脚本和数据泵活动。

创建完示例 vbScript 之后就可以开始构建包了。

这里从前面一节中创建的 SSISProject 项目文件开始。现在,控制流应该是空的。为了让整个过程能够继续下去,需要调用 vbScript 文件来生成一个将要导入的文本文件。从“工具箱”中把“执行进程任务”拖到主“控制流”窗口中,这样 SSIS 将会向“控制流”窗口中添加“执行进程任务”,如图 16-3 所示。

为了进一步配置这个新任务,需要双击这个任务以打开“执行进程任务编辑器”,如图 16-4 所示。

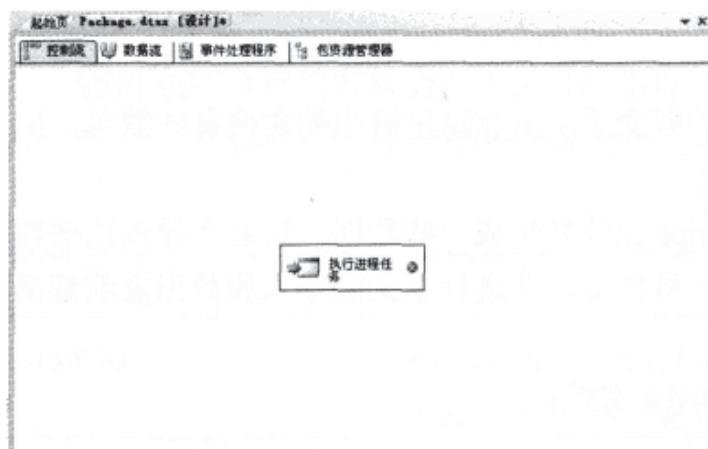


图 16-3

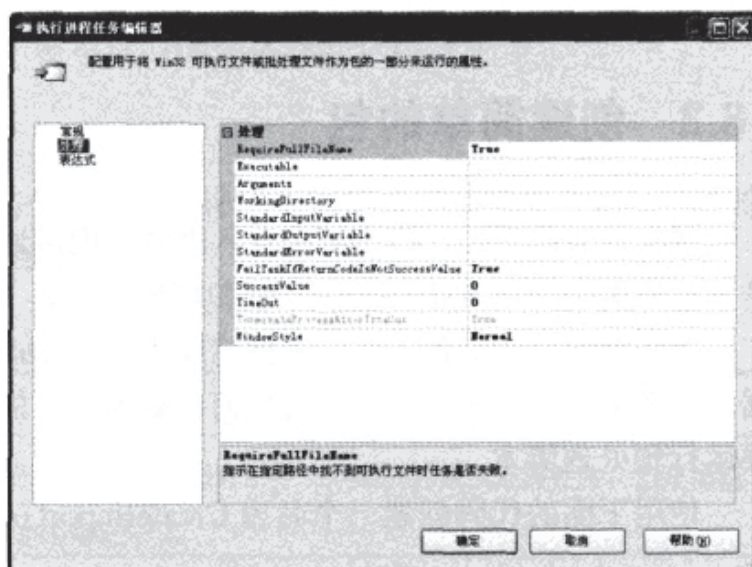


图 16-4

注意这里已经切换到了“处理”选项上,因为与“常规”选项相比,该页面上的内容在屏幕截图上显示的内容更丰富。表 16-3 简单介绍一下本例需要用到的设置:

表 16-3

选 项	设 置
常规 Name	GenerateImportFile
常规 Description	生成用于导入的文本文件
处理 Executable	CreateImportText.vbs(前面加上脚本文件的完整路径)
处理 Working Directory	C:\(或者读者选择的目录——只要确保一致就可以了)

当完成了这些修改之后单击“确定”,此时“控制流”窗口中除了任务的名称变为 GenerateImportFile 之外没有发生任何变化。

接着把“执行 SQL 任务”对象拖动到“控制流”窗口中。现在选中 GenerateImportFile 任务后应该会有一个箭头悬挂在任务方框的底部,如图 16-5 所示。

接下来是比较麻烦的部分了:单击 GenerateImportFile 任务的“输出”——也就是单击小箭头的末端。把箭头拖动到“执行 SQL 任务”的顶端,接着生成器将连接两个任务(如图 16-6 所示)——注意箭头是如何指示控制流的。

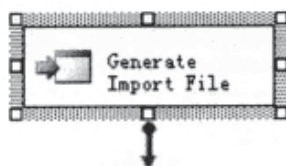


图 16-5

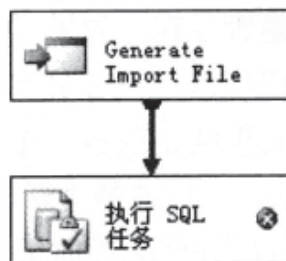


图 16-6

现在来看看这个箭头代表什么。双击这个箭头将会打开“优先约束编辑器”，如图 16-7 所示。

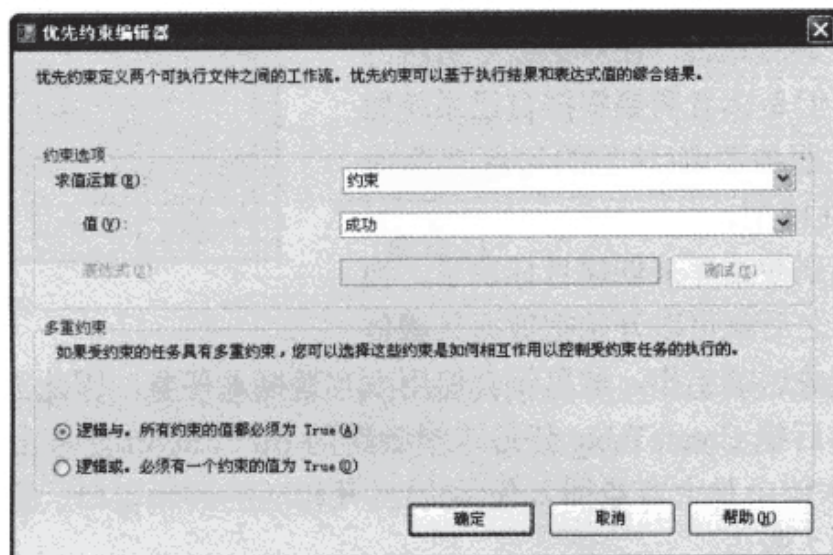


图 16-7

注意它是如何定义流在什么条件下允许执行的。在本例中只有当 GenerateImportFile 任务成功完成后才能转到“执行 SQL 任务”。这里也可以定义其他的流来处理诸如任务失败之类的情況或允许在第一个任务完成(任何类型的完成，不管是否成功)时，不管该任务是成功还是失败都运行第二个任务。

单击“取消”退出对话框并返回主窗口，然后双击“执行 SQL 任务”以打开“执行 SQL 任务编辑器”，如图 16-8 所示。

同样，这里需要编辑一下名称。接着单击 SQLStatement 选项以打开“输入 SQL 查询”对话框，如图 16-9 所示。

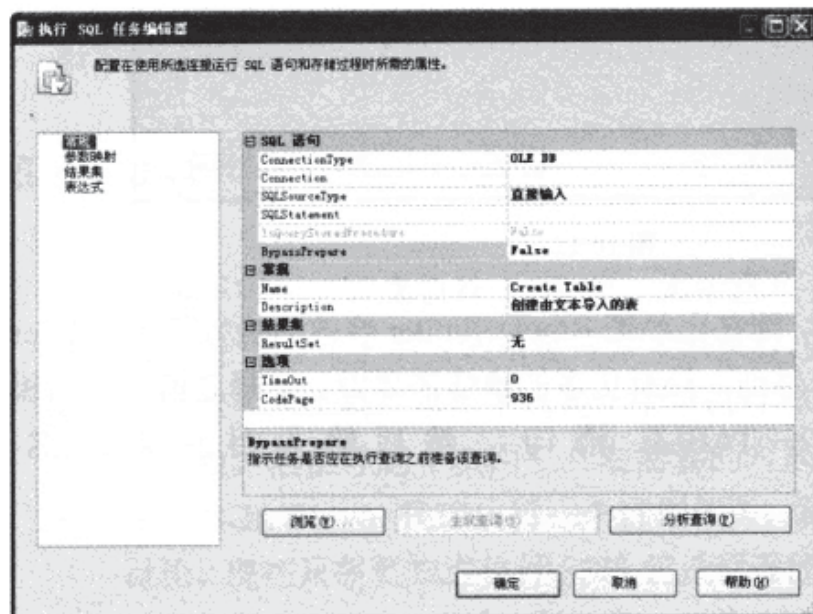


图 16-8



图 16-9

首先检查一下表是否存在, 如果已经存在, 那么就删除它。然后, 在确信该表不存在后(如果该表存在, 那么刚才已经删除它了)创建目标表。

最后一件需要做的事情是获取一个可以工作的连接。单击“连接”选项, 然后选择“新建连接...”, 接着会弹出连接管理器对话框(读者如果足够细心的话将会注意到连接管理器窗格处于包的主窗格的下面——本质上这是相同的功能区域)。现在, 在本例中创建的包还没有任何连接, 因此需要再次单击“新建”来打开 OLE DB 连接管理器对话框。在对话框中所填的内容如图 16-10 所示, 但是读者需要根据自己系统做一些调整以匹配数据库服务器名(这里的句点“.”表示本地服务器)和安全性模型。



图 16-10

现在已经可以连接到数据库来创建目标表了。当创建了源数据和目标表之后就可以开始把数据从源传送到目的地了。为了完成这项工作, 这里将会使用大容量插入任务, 因此需要把满足要求的一个任务拖动到模型中, 然后把 CreateTable 任务连接到新的 BulkImport 任务上, 如图 16-11 所示。

同样, 双击任务(本例中是大容量插入任务)以打开相应的编辑窗口。这里需要关注的是“连接”选项卡, 如图 16-12 所示。

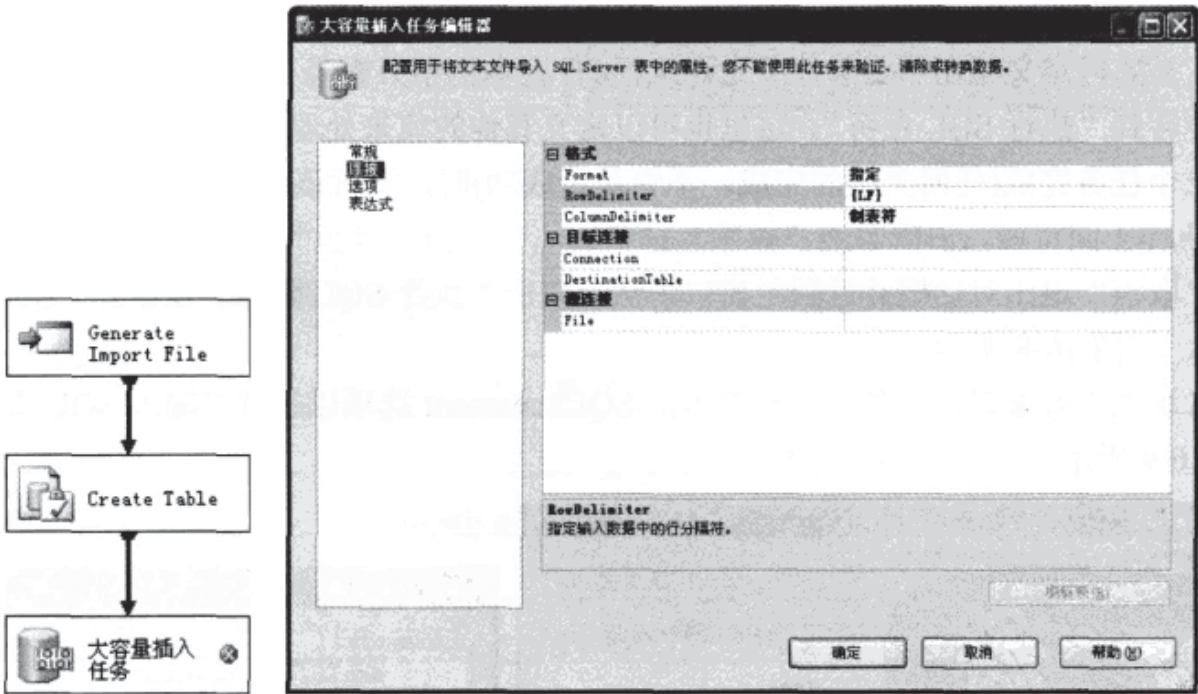


图 16-11

图 16-12

在这个选项卡上需要做几处修改。例如, 现在已经把 RowDelimiter 修改为换行符了, vbScript 中的 WriteLine 命令将会写入这个换行符。然而, 还有更多的事情需要做。选择之前创建的那个连接以便在该连接上运行 CREATE TABLE 语句, 然后输入目标表的名称 ([AdventureWorks].[dbo].[TextImportTable])。

注意:

注意在对话框中引用的表必须存在。为了准备好数据库, 笔者手动运行了一次 CREATE 语句

以确保任何在编译时需要引用该表的任务都能正常工作。由于这个过程每次都会删除原来的表并创建一张新表,因此这样做不会带来任何问题。

最后单击“文件”连接框,然后选择“新建连接”。这将为文本文件打开“文件连接管理编辑器”,如图16-13所示。

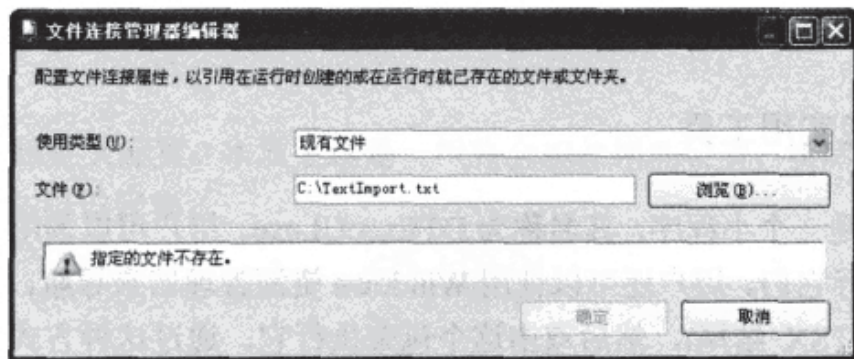


图 16-13

注意:

注意文件不存在的错误,这与在创建 TextImportTable 表时碰到的问题是一样的。可以创建一个空的伪文件或者运行 CreateImportText.vbs 文件一次来生成初始文件,然后刷新,这样错误解决了。

单击“确定”退回到“控制流”主窗口,现在可以执行包了。

如果要立即执行包,可以单击运行图标(工具栏上绿色的箭头)。当任务运行时注意观察 Dev Studio 是如何通过改变不同任务的颜色来显示进度的。

提示:

还有几点需要注意: SSIS 能够一次运行多个任务。例如,本例中之所以把项目做成完全线性(一次一块)是由于直到最后(当确信有新的数据可用时)才会删除目标数据。但是,就像把 CREATE TABLE 依赖关系直接链接到导入上一样,本例也可以把文件生成的任务直接链接到构造导入的任务上。如果这样做了,那么 SQL Server 将会同时运行表的 DROP/CREATE 和文件的生成,只是在允许执行构造导入之前“二者”必须都完成。

由于下一节将使用本例中的包,因此请继续构建这个包(在“生成”菜单中选择“生成”选项)。

16.4 执行包

存在几种不同的方式来执行 SSIS 包,本章将在 Dev Studio 以某种测试模式使用其中一种方式,但是读者在平时几乎不会采用这种方式来执行包。更常见的执行包的方式包括:

- **执行包实用工具:** 本质上这是一个可执行程序,用户可以指定需要执行的包,设置所需的参数,然后让这个实用工具在需要的时候运行包。
- **使用 SQL Server 代理执行一个计划任务:** 第 22 章将会对 SQL Server 代理进行更深入的讨论,现在只需要知道执行 SSIS 包是代理能够完成的作业类型中的一种。用户可以指定包的名称和运行的时间以及频率,接着 SQL Server 代理会处理包的执行。

- **从程序内部执行：**SQL Server 提供了一整套的对象模型来支持用户在程序中实例化 SSIS 对象，设置包的属性并且执行包。这涉及到很多内容——内容是如此之多以至于 Wrox 专门为该主题出版的一本书：*Professional SQL Server 2008 Integrations Services*，该书由 Knight 等人完成 (Wiley 2009)。本书的第 25 章(可以从 p2p.wrox.com 或者 professionalsql.com 上下载)将会简要介绍这个主题，但是如果读者需要进行 SSIS 编程，那么笔者建议看看 Brian 所做的工作。

16.4.1 使用执行包实用工具

执行包实用工具是一个小程序，其名称为 DTEExecUI.exe。用户可以调用它来为现有的包指定设置和参数，然后执行它们。用户还可以使用 Windows 资源管理器来导航，在文件系统中找到需要执行的包(它们以 .DTSX 结尾)，然后双击这个包来执行它。通过这种方式执行本章中的文本导入包将得到图 16-14 中显示的执行对话框。

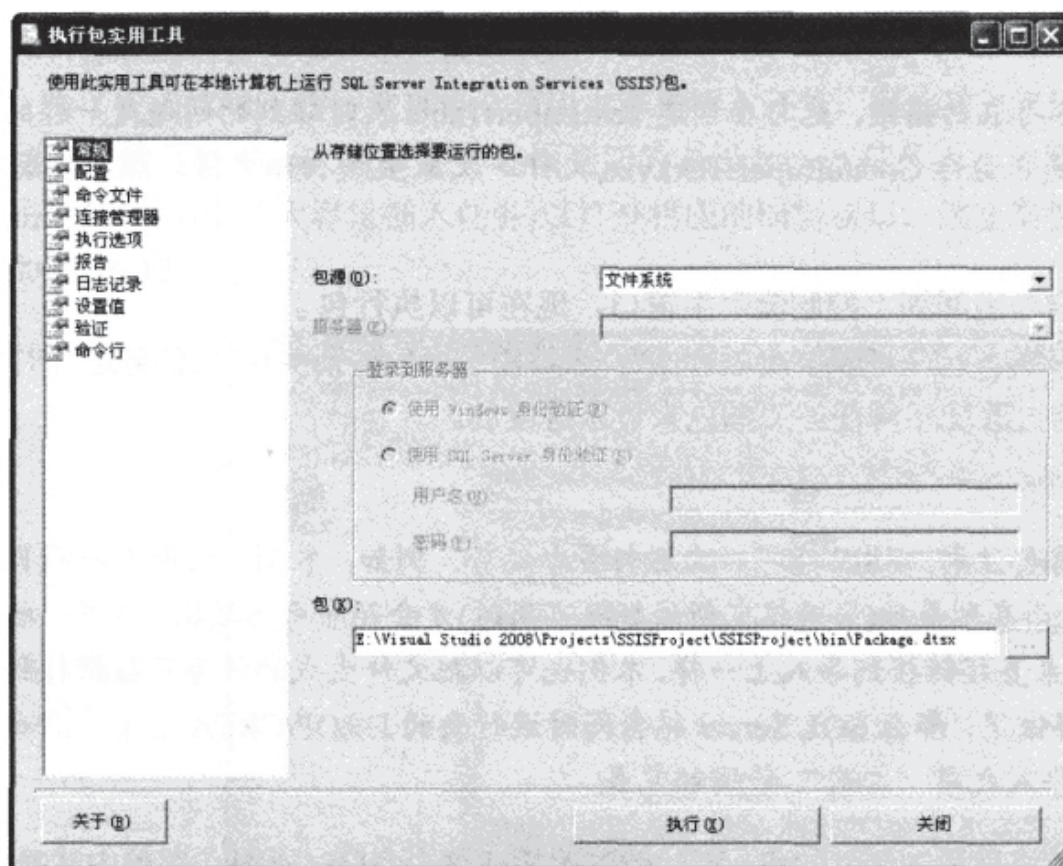


图 16-14

正如读者所看到的那样，单击对话框左边的不同选项能够得到不同的对话框。要完整地讨论这些内容需要整整一本书才能完成，下面只介绍这个实用工具中一些关键对话框中的重要部分。

1. 常规

这个对话框中的很多区域都是不言自明的，这里需要特别注意的是“包源”区域。用户可以把 SSIS 包存储在以下三个位置之一：

- **文件系统：**导入/导出向导包就是存储在文件系统上的。这个选项对于移动性来讲是非常不错的——用户可以很容易地把包保存起来，然后把它移动到另一个系统上。

- **SQL Server:** 该选项将把包存储到 SQL Server 中。如果采用了这种方法,那么每当用户备份 MSDB 数据库(这是一个系统数据库,所有 SQL Server 安装中都包含这个数据库)时,其中存储的包也会被备份。
- **SSIS 包存储区:** 这个存储模型提供了一种有组织的“文件夹”集的概念,用户可以把包与其他一般类型或用途相同的包存储在一起,而文件夹则可以存储在 MSDB 或文件系统中。

2. 配置

SSIS 允许用户为包定义配置。本质上这是一组将会被使用的设置,实际上用户可以把多个设置结合在一起,形成一组设置集。

3. 命令文件

这些是用户想要作为包的一部分运行的批处理文件。用户可以使用它们做一些系统级别的工作,如把文件复制到需要的地方(它们将在运行 Integration Services 服务的帐户之下运行,因此必须要授予那个帐户所有所需的网络访问权限)。

4. 连接管理器

这里有点命名不当——与其说这是连接管理器列表,不如说是连接列表。只要看一下“说明”列,读者就可以看到包所使用的每个连接的许多重要属性。注意在本例的包中有两个连接,如果读者仔细观察的话,那么可以看出其中一个是与文件信息相关联的(对于正在使用的平面文件的连接而言),而另外一个专门与 SQL Server(导出源连接)相关联的。

5. 执行选项

不要低估这个选项的重要性。它不仅允许用户在较高层次上指定在出现问题(如果这里有错误)时所采取的动作,而且允许用户建立检查点跟踪——这样可以容易地看出用户的包是在何时以及何处进入到不同的执行点的。这在性能调优和调试中是至关重要的。

6. 报表

这个选项完全是为了告知用户系统发生了什么事情。用户可以设置这个选项来建立反馈机制:具体有多少反馈取决于想要跟踪的事件以及所确定的信息级别。

7. 日志

要设置这个选项并使它工作起来是相当复杂的,但是它能够为用户提供非常灵活的框架来跟踪甚至最复杂的包,因此有很高的“平稳”因素。

用户可以使用这个区域配置包,以便把日志信息写入许多预先配置好的“提供程序”中(本质上是存储日志数据的易于理解的目标)。除了预先安装好的诸如文本文件、甚至是 SQL Server 表之类的提供程序之外,用户甚至可以创建自定义的提供程序(不适合胆小的人)。用户可以在包级别上记录日志,也可以以非常精细的粒度记录日志并能够把包中不同任务的日志写入到不同的位置。

8. 设置值

通过这个选项,可以设置包的所有运行时属性的初始值(本例中的简单包中没有这种属性)。

9. 验证

完全不同的包可以有相同的文件名(例如,只要位于文件系统中不同的地方即可)。此外,包能够在相同的文件或包存储区域内保留它们自身的不同版本。“验证”对话框完全是用来筛选和验证用户想要执行的包/版本的。

10. 命令行

用户可以从命令行中执行 SSIS 包(有时候这是非常方便的,例如用户想要在一个批处理文件之外运行 DTS 包)。SSIS 包执行实用程序中的这个选项用来指定当用户在命令行中运行这个包时使用的参数。

这个实用工具将会为用户确定大多数的参数——这里该选项仅仅是让用户在执行该实用工具时能够覆盖选项。

11. 执行包

当用户单击包执行实用工具中的“执行”时,包将会开始运行。运行完毕之后将会在用户指定的位置生成一个文本文件——打开这个文本文件,查看其内容,验证一下其内容是否如期望的那样。

16.4.2 在 Management Studio 中执行

虽然 Management Studio 不提供包编辑器,但是用户可以使用它来运行包。

在 Management Studio 中的“对象资源管理器”窗格中,单击“连接”图标并选择 Integration Services,在连接对话框中填写相关信息。这将会创建一个到该服务器上的 Integration Services 的连接并在“对象资源管理器”中添加一个 Integration Services 节点。

当使用这种方式(使用 Management Studio)执行包时,包必须处于服务器本地(不在文件系统中)。幸运的是,用户可以通过右击“已存储的包”下的“文件系统”节点来使用 SQL Server 提供的导入包的功能。只要在文件系统中导航到之前创建的包,然后给它一个在包存储区域中使用的名称,然后就可以导入它了。接着用户随时都可以右击这个包并执行它(这将会打开前面一节中讨论的执行实用工具,读者应该对这部分内容比较熟悉了)。

16.5 小结

SQL Server Integration Services 是一个可靠的提取、转换以及加载工具。用户可以使用 Integration Services 来一次性或重复地把数据导入到数据库或者从数据库中导出数据——使用它能够混合各种各样的数据源。

实际上本章讨论的内容已经有些稍微超出基础知识的范围了——涉及到了外部访问和多级控制流。要了解 Integration Services 提供的所有功能并成为专家肯定是一件非常困难的事情,而掌握基本的导入和导出以及包的运行则是一件相对简单的事情。笔者建议读者从简单的事情开始,根据自身的需要学习新的内容。当需要更深入地了解 SSIS 的功能时可以参考一下专门介绍 SSIS 功能的书籍。

第 17 章

复 制

SQL Server 2005 对复制功能做了较大的改动,而在 SQL Server 2008 中,复制功能则是几个改动较少的功能之一。实际上,未与非复制功能直接相关联的部分几乎没有发生任何变化(需要允许复制新的数据类型,对吧?)。

通常,复制功能是大家都会忽视的功能之一——除非需要使用它。因此,对很多人来说,快速学习并实施(遗憾的是并不一定以这种顺序进行)它是一种突发性的历险。

那么,复制究竟是什么呢?这里笔者将抛弃 Webster 的定义而使用自己的定义:

提示:

复制是一个获取一个或多个数据库的过程,它系统地提供了一种基于规则的复制机制来把数据复制到一个不同的数据库中或从一个不同的数据库中复制数据。

复制常常是一种拓扑和管理问题。正因为如此,许多开发人员习惯于忽视它——这可不是一个好主意。对于软件架构师来说,复制是非常重要的,因为使用它可以解决很多复杂的加载和数据分布问题,如:

- 为那些通常不会连接在主网络上的客户提供数据
- 平衡与繁重的报表需求相关的负载
- 为那些需要访问地理上分散的数据库的应用程序解决延迟问题
- 支持地理上的冗余

这些只是它所提供的众多功能中的一小部分而已。

所以,请记住这些,接下来将要全面介绍复制。需要预先警告读者的是这里不会像之前那样介绍很多实例,但是请耐心一点——这是有原因的。简单地说,读者一旦已经构建了一种或两种类型的复制就已经学习到了大部分的“构建”方式。此外,构建复制实例的任务实际上更多的是管理员的任务。因此,本章将主要介绍其内部发生的事情,从而可以为本章节省下大量的篇幅以用来介绍不同的复制方法是如何创建和解决问题的以及如何使用不同的复制模型来解决不同的问题。

本章将介绍下列内容:

- 复制的一般概念
- 可用的复制模型(本章将会介绍一到两个例子)
- 安全性问题
- 复制管理对象(RMO)——以程序的方式管理复制

最后,虽然本章无法保证使读者成为复制方面的专家(老实说,笔者也不是专家),但是通过本章,读者能够深入地理解基础知识并能合理地处理各种问题。

17.1 复制的基础知识

复制就像是一个大的拼图——通过某种方式把多个小块组成一个完整的部分。其内容包括拓扑结构考虑事项(发布服务器、订阅服务器和分发服务器)和发布模型(合并、事务或快照)。用户在决定采用何种拓扑结构和发布模型时需要考虑以下一些因素。

17.1.1 计划复制时需要考虑的事项

在考虑可用的拓扑结构和复制方法时需要考虑很多因素。这些因素应该是用户在设计时所做的评估的一部分,用户通过做评估来决定应用程序该采用何种形式的复制。其中需要考虑的因素包括:

- 自治
- 延迟
- 数据一致性

下面简要介绍一下这些因素。

1. 自治

自治是指一个复制实例在多大程度上能够独立运行。哪些数据需要被复制以及复制的频率是多少呢?例如,读者可能在为一个销售应用程序提供支持,在这个销售程序中,每个站点保留着各自的客户记录。用户可能想要把这些记录复制到一个中心数据库中以供报表使用,此外,其他诸如自动库存统计之类的任务可能需要使用这些记录。每个站点都是高度自治的(实际上它们并不关心中心数据库有没有获取它们的数据,它们仍然可以根据本站点上的数据继续销售)。实际上,尽管有些应用会依赖于中心数据库,但是即使中心数据库上缺失了某个站点上一天的数据也不会造成什么灾难性的后果(取决于用户如何使用中心数据库生成的报表或者在重新进货之前有多少可用的存货)。

2. 延迟

延迟是指在相邻更新之间的时间间隔,换句话说,它是指在发布服务器上发生变化之后到订阅服务器得知该变化所花费的时间。站点之间的自治性越高,相邻更新之间的延迟可能就越大。

确定一个可接受的延迟是一件非常困难的事情,并且可能会牵涉到上述的自治性问题。如果把一个站点上的信息传输到中心服务器上仅仅是为了周期性累加报表,那么或许可以每天(甚至更长时间)更新一次。但是如果站点为了进行某些类型的销售需要从中心数据库中获取出货能力的数据,那么就需要及时更新中心数据库了,这样才能保证产品不会过度销售(两个站点试图销售库存中同一件产品)。

3. 数据一致性

显然,数据一致性是任何分布式系统都要考虑的一个关键问题。当然,这里需要确保的是各

个复制实例一直包含了相同的数据，而这可以通过两种方式来实现：

- **数据集中：**最终所有的站点将包含相同的数据；但是如果所有的变化都是发生在一台服务器上的话，那么这些值就不需要跟它们本应该具有的值相同。一个可能的例子是过度销售情况。如果两个销售发生在同一台服务器上，那么第二次销售就能够检测出过度销售的情况，这样第二次销售就不会进行。而如果不是发生在同一台服务器上，那么每个数据库都认为还有一件产品可用，因此用户可能最终会得到一个负的库存级别，当然这还要取决于库存调整的处理方式。同样，用户的数据最终可能会得到同样的值，但是它们得到该值的中间步骤可能是不同的(更新的顺序可能是不同的，这取决于参与更新的复制客户数量以及它们进行同步的时间)。
- **事务一致性：**任何服务器上的结果都是一样的，就像所有事务是在一台服务器上执行的一样。这是通过某种机制——正如其名称所暗示的那样——事务来实现的。读者只要稍微思考一下就能意识到这种机制所带来的延迟的影响——在事务结束之前所有参与这个复制集的服务器上的事务都要结束。

4. 模式一致性

对于那些习惯于在非复制环境下工作的开发人员来讲，只要经过授权就能够很容易地改变数据库的模式。是否需要添加或删除一个新列？没有问题。需要添加一张新表吗？这没什么大不了的。除了在任何数据库环境下都能够从容应对的基本问题之外，读者很快就会发现复制环境中事情变得有点复杂了。

提示：

不管是否进行复制，需要记住的是当用户修改了表的模式后，其本质上是修改了整个系统的基础(或者至少是被修改的模式对象所服务的那部分)。模式变更一般都被看做是非常重大的变更，需要仔细考虑并进行系统地计划。一些变更(特别是添加)通常只会产生相对较小的间接影响。而那些修改或删除现有对象的变更则通常比较麻烦，特别是当需要处理向后兼容的问题时往往会产生致命的问题。此外，还需要记住的是其他人可能已经对现有系统进行了“扩展”，他们的系统依赖于现有的模式，而当修改现有模式时通常很难预计这种变更会对那些系统产生什么影响。

一个好消息是 SQL Server 继续为在复制期间进行模式变更提供支持。在发布服务器上添加或删除的字段将会在后面的复制操作阶段被传播到所有的订阅服务器上。而坏消息则是用户需要遵循更严格的变更过程。这里的底线是如果用户需要经常变更模式，那么在实施复制之前需要对变更策略做出完整的计划。

提示：

当复制模式变更的概念首次被加进 SQL Server 的时候是通过使用名为 `sp_repladdcolumn` 和 `sp_repldropcolumn` 的特殊存储过程来实现的，而不是通过更常见的 `ALTER TABLE` 命令来实现的。而在 SQL Server 2005 中，这种实现方式被改回来了，现在已经不建议使用 `sp_repladdcolumn` 和 `sp_repldropcolumn` 了(避免使用它们)。

5. 其他需要考虑的因素

其他一些需要考虑的因素包括：

- 服务器之间的连接有多可靠？如果是一个本地连接，那么用户可以完全信任这个连接，但是如果连接的是不同的地理位置呢？如果是不同的国家呢？
- 连接的延迟属于哪一类？这在某种程度上又回到了可靠性的问题，但是实际上它是一个独立的问题。如果等待一个简单的 ping 的返回结果(假设现在是一个数据块)都需要花上一到两秒钟，那么还真的想要强行使用事务复制吗？
- 与连接的延迟一样，可供使用的带宽是多少？有多少流量需要经过这条线路以及其他进程将会如何使用同一条线路？需要压缩复制相关的数据吗？
- 复制方法是通过有线连接的吗？也就是说如果根本就没有到复制的目标服务器上的连接该怎么办？尽管 SQL Server 支持非连接模型，但是在相邻两次所需时间较长的更新的间隔期间，这又意味着什么呢？

17.1.2 复制角色

复制的过程包括三个角色：发布服务器、分发服务器以及订阅服务器。任何一台服务器都可以担当其中一个角色(或几个角色)。这种结构是非常灵活的，如图 17-1 所示。

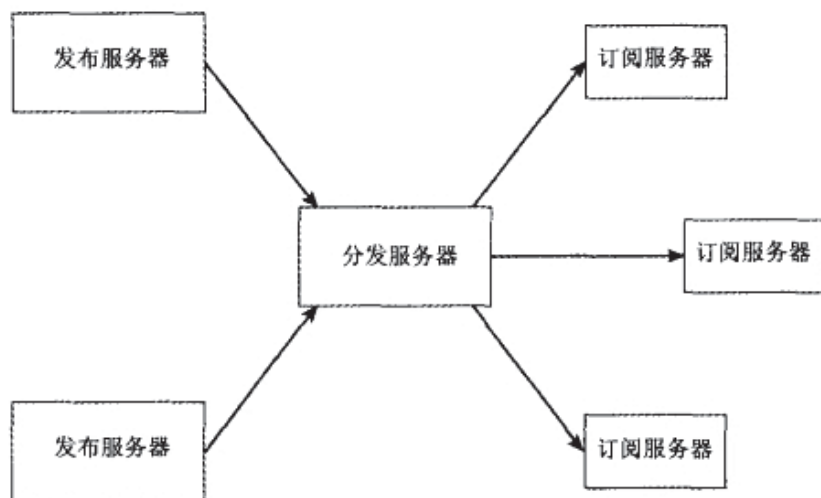


图 17-1

正如图中所显示的那样，多个发布服务器可以共用同一台分发服务器，同时任何一个发布可以拥有多个订阅服务器。下面将详细介绍这些角色。

1. 发布服务器

可以把发布服务器看成源数据库。即使是在发布服务器和它的订阅服务器平等地共享数据的情况下，还是会有一个数据库是作为控制数据库而存在的。

2. 分发服务器

分发服务器就像管理变更的票据交换所。它上面有一个特殊的分发数据库，用来跟踪变更以及记录哪些订阅服务器已经收到了那些变更。此外，它还会跟踪所有同步进程的运行结果并且能够了解到冲突是如何发生的，而这些冲突是需要解决的(本章后面将会用更多的篇幅来介绍冲突)。

3. 订阅服务器

所有参与复制发布过程但不是真正的发布服务器的数据库都可以看成是订阅服务器。但是这

并不意味着订阅服务器只是接收数据的——实际上, 订阅服务器可能同时接收和分发数据, 当然, 这要取决于用户所选择的具体模型(同样, 后面将会用更多的篇幅来介绍这一概念)。

17.1.3 订阅

订阅服务器接收到的订阅(subscription)叫做发布(publication)。一个发布中将会包含一个或多个项目(article)。通常, 一个项目是一张表或者表中数据的某个子集, 但是它也可以是一个或一组存储过程。通过订阅一个发布, 订阅服务器将会订阅发布的所有项目, 而不能订阅单个项目。

可以把订阅设置成推订阅或拉订阅:

- 如果是推订阅, 那么发布服务器将会决定何时更新订阅服务器。当用户想要使延迟最小化(由于发布服务器通常是唯一接收变更的数据库副本, 因此它会在变更发生时及时得知并做出相应的动作)或者因为其他一些原因而要使发布服务器拥有完全的控制权, 那么用的最多的就是这种模型。
- 如果是拉订阅, 那么订阅服务器将会请求更新。这种方式允许更高级别的自治, 因为订阅服务器将自己决定何时进行更新。

一个发布可以同时支持推和拉订阅, 但是所有订阅服务器则只能采用推或者拉订阅——它不能对同一个发布同时采用推和拉订阅。

17.1.4 订阅服务器的类型

SQL Server 支持三种类型的订阅服务器:

- 默认是本地订阅服务器。发布服务器是唯一知道订阅服务器的服务器。本地订阅服务器通常作为一种安全机制来使用, 或者当用户想要最大化服务器之间的自治性时会使用本地订阅服务器。
- 当所有参与发布的服务器(成为发布服务器或订阅服务器)都知道其他订阅服务器时, 通常采用全局订阅服务器。全局订阅服务器通常用于多服务器环境, 在这种环境中, 用户通常希望能够把来自不同发布服务器的数据合并到一个订阅服务器上。
- 匿名订阅服务器仅对已经连接了订阅服务器的发布服务器可见。在设置基于 Internet 的应用程序时, 这种方式是非常有用的。

17.1.5 筛选数据

SQL Server 提出了水平或垂直筛选表的概念。水平筛选(有时也称为水平分区)在表中标示出要发布的行(通过 WHERE 语句)。例如, 用户可以根据仓库来划分库存信息以管理各个仓库的库存总数。垂直筛选(也叫做垂直分区)标示出要被复制的列。例如, 用户可能想要发布库存表中的手头拥有的存货数量信息, 而不是订单上的数量信息。

17.2 复制模型

在复制中有三种可用的模型。它们都综合考虑了延迟、自治性以及本章中之前讨论的其他需

要考虑的因素以便在这些因素之间取得某种均衡。要确定使用哪种模型则需要在以下几个方面予以权衡：

- **自治程度**：服务器之间是否有持续可用的连接？如果有，那么连接的带宽是多少？需要复制的事务有多少？
- **冲突管理**：同时或者在复制被更新期间对多个地方的同一数据进行修改将会带来怎样的风险？当一个或多个服务服务器上的数据不一致时，对数据不一致性的容忍度是多少？

一些复制场景不允许有连接性，除了一些零星的情况之外——其他一些场景可能不允许有任何连接性(除非使用那种具有讽刺意味的叫做“sneaker net”的方式来传输数据，用户可以通过人力、邮寄或空运的方式，将磁盘或其他可移动存储介质从一个站点送到另一个站点处，从而实现运行、邮件、迁移之类的任务)。其他一些复制场景则绝对需要在站点之间保持完美的持续的连接，同时不允许有任何的数据丢失。

按自治性从高到低排列，这三种模型分别为：

- 快照复制
- 合并复制
- 事务复制

下面将介绍每种复制模型的优缺点，列出每种复制模型的适用场合以及保证数据完整性需要考虑的因素。

需要指出的是，用户可以根据需要混合多种复制模型以满足特定的实施需求。在一些发布中，用户可能想要在站点之间实现更高的自治性，而在另一些发布中，把延迟最小化则是一个关键问题。

提示：

这里需要指出的是一个发布仅仅是——一个发布。不需要人为地将一个发布同一个数据库等同起来。一个发布中包含的项目可能只是用户所订阅的数据库的一部分。而订阅数据库中的其他对象可能由另一个不同的发布提供——可能来自一个完全不同的发布服务器。

17.2.1 快照复制

快照复制将会在所有被复制的数据的源上拍摄一张“照片”(如图 17-2 所示)。这些数据将用来替换目标服务器上的数据。

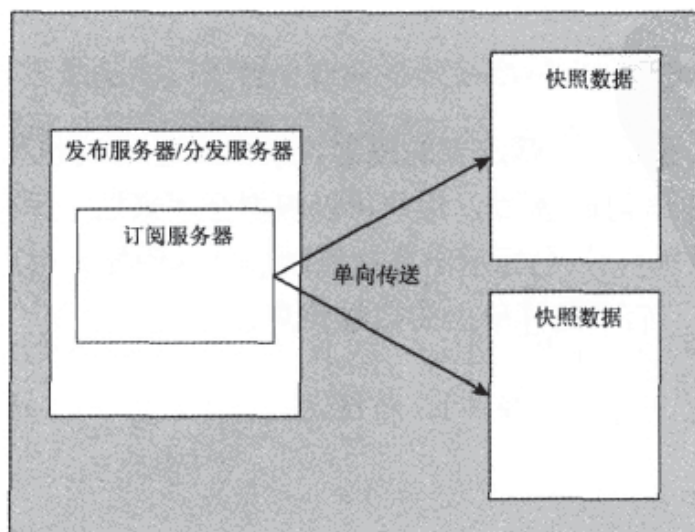


图 17-2

形式最简单的快照复制是最容易设置和管理的复制类型。在复制过程中,整张表或表段(用于分区表)都会被写入订阅服务器。由于仅有周期性的更新,因此复制所需的服务器或网络开销是最小的。

快照复制常常用来更新订阅系统上的只读表。它允许订阅服务器具有高级别的自治性,但是这是以高延迟为代价的。在使用快照复制时,用户能够严格地控制周期性更新发生的时间。这意味着用户可以把更新安排在网络和服务活动较少的时间段进行(用户甚至可以使用磁盘或其他的硬件媒介来把快照复制到其他地方)。这里可能需要考虑的一个问题是周期性更新过程中完成复制需要花费的时间和资源。随着源表的增长,每次更新所需传送的数据也会增长。随着时间的推移,可能需要改变复制类型或者对表进行分区以减少所需复制的数据量,从而把流量保持在一个可管理的级别上。

注意:

快照复制的一个变体是带有立即更新订阅服务器的快照复制。使用这种复制模型就可以更改订阅服务器上的数据了。除非已经部署了立即更新,否则那些变更将会周期性地被发送到发布服务器上,部署了立即更新之后将会实时执行分布式事务。

1. 快照复制如何工作

复制是通过复制代理实现的。每个代理本质上是一个独立的小程序,它负责事务监控以及根据需要为特定类型的代理分发数据。

快照代理

快照代理支持快照复制以及为其他类型的复制(它们的首次同步数据都是通过快照实现的)进行数据表的首次同步。所有类型的复制都要求在开始复制之前源和目标表必须要处于同步状态,这可以通过复制代理或手动同步来实现。不管采用哪种方式,快照代理都承担着同样的职责。它将为要发布的数据拍摄“照片”,然后把这些文件存储到分发服务器上。

分发代理

在进行首次同步和快照复制(后面将会看到事务复制也是这样做的)时,分发代理是用来把数据从发布服务器移动到订阅服务器上的。如果是推订阅,那么分发代理通常运行在分发服务器上。如果是拉订阅,那么分发代理通常运行在订阅服务器上。分发代理的实际位置是通过一个选项来配置的,用户可以通过 Management Studio 或 RMO 配置该选项。

快照复制的过程

快照复制使用了周期性的更新(用户可以根据需要配置其频率,但是,一般来讲,通常会在作业管理器中安排一个作业来定期运行快照复制)。在更新期间,模式和数据文件将会被创建,并会被发送到订阅服务器上。下面是其基本过程(如图 17-3 所示):

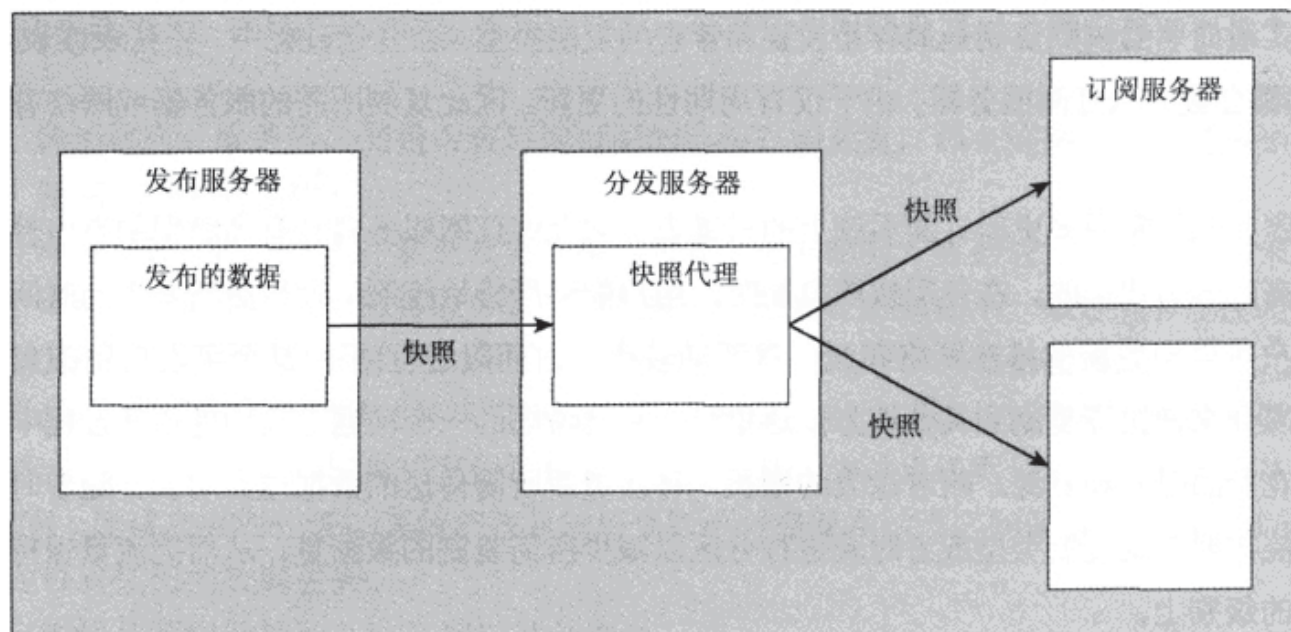


图 17-3

- (1) 快照代理在将要被复制的发布中的所有项目上加一把共享锁以确保数据一致性。
- (2) 每个项目的表模式的副本将会被写入分发服务器上的分发工作文件夹中。
- (3) 表中数据的快照副本将会被写入快照文件夹中。
- (4) 快照代理释放发布所包含的项目上的共享锁。
- (5) 分发代理在订阅服务器上创建目标表 and 数据库对象(如索引)并把快照数据复制进这些对象，如果表已经存在，那么将覆盖现有的表。

如果所有的订阅服务器都是 Microsoft SQL Server，那么快照数据将存储为一个原生的 bcp(第 15 章介绍过这一内容)文件。如果需要支持异类数据源(非 SQL Server)，那么将会创建字符模式文件，而不会创建 SQL Server bcp 文件。

注意：

SQL Server 支持复制异类数据源。目前，针对 Oracle 的所有 O/S 平台以及针对 DB2 的大多数 O/S 平台都支持事务和快照复制。

2. 何时使用快照复制

当需要更新远程服务器上的查询数据或数据的只读副本时可以使用快照复制。如果用户希望(或需要)间歇性地连接到发布服务器上，可以使用快照复制。

举一个例子，考虑花园供应连锁商店是如何管理服务器的。在本例中，商店位于几个城市，一些较大的城市还拥有多家商店。那么哪些数据可以考虑使用快照复制呢？

显然客户记录是一个选择。客户，如庭院美化师，可能会出现在不同的地方。在大多数情况下，客户信息更新的延迟将不会带来什么问题。这也给用户提供了一种方式来确保仅有那些能够访问发布服务器的用户能够修改客户记录。

库存记录可能会有一点问题。库存中保存的项目通常会随着季节的变化而变化。尽管这样，用户可能还是会把该项目保存在文件中，只不过是把可用数量变为 0 而已。现在的问题是用户希望能够在商店之间复制更多的最新库存记录。这使得用户能够搜索手头上没有货的项目，而不用查询所有商店。及时的更新从很大程度上来说就意味着要使用事务复制(稍后将会讨论这个问题)。

3. 特别的计划需求

建立快照复制的一个重要问题是时机。需要确保在快照代理生成快照期间用户不需要对任何发布的表进行写访问(是否还记得会在发布包含的所有项目上设置共享锁吗? 这就是为了在锁期间——也就是说在发布对外分发的期间阻止对数据的插入、更新和删除)。还需要确保的是复制产生的网络流量不会对其他网络操作带来影响。

随着被发布的表的增长, 存储空间可能也会成为一个问题。用户需要确保在目标文件夹(CD-ROM、DVD、移动驱动器或磁带等)上有足够的物理磁盘空间来存放快照文件夹。

17.2.2 合并复制

尽管快照是很好用的, 但是人们并不总是生活在“只读”世界中的。有多种方案可以用来处理发生在多个服务器上的数据变更, 合并复制是其中一种解决方案。发布服务器在接收到来自所有站点的变更之后会合并这些变更(参见图 17-4)。而更新则可以通过周期性的方式(通过计划任务——这是完成这类任务的典型做法)或按需请求的方式来实现。

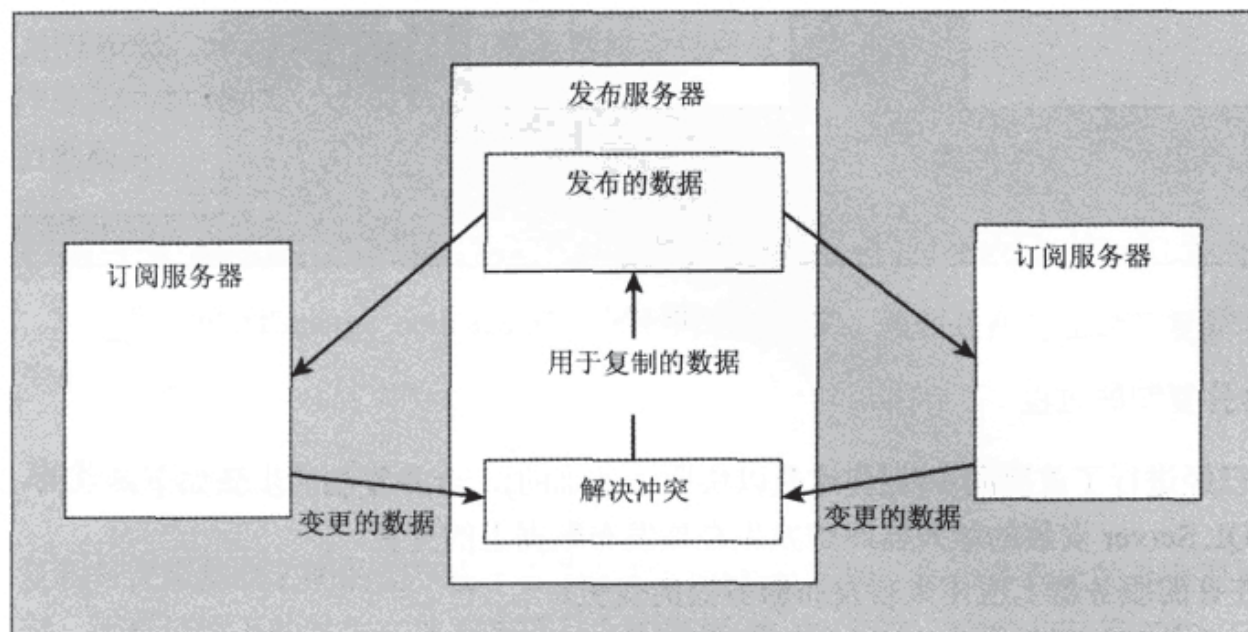


图 17-4

合并复制具有高度的自治性, 但同时也具有较高的延迟以及事务一致性较低的风险。与事务和快照复制能够保证一致性不同, 合并复制并不能保证一致性。在需要决定何时进行合并复制时, 这是其中需要考虑的较重要因素之一——一致性到底有多重要?

从某种程度上来说, 在合并复制中, 角色已经变得有点模糊不清了。发布服务器是被合并数据的最初来源, 但是变更却可以同时发生在发布服务器和订阅服务器上。可以通过行或列来跟踪变更。这里事务一致性之所以无法保证是因为当不同的系统更新同一行时可能会发生冲突。而数据一致性则是通过根据所确立的标准(用户甚至可以编写自定义的冲突解决算法)进行冲突解决来维持的。用户可以判断出到底是行还是列出现了冲突。

与事务复制一样, 快照代理会为同步准备好最初的快照。但是同步过程是不同的, 这里合并代理将执行同步。此外, 它还会应用自最初的快照被创建以来所发生的变更。

1. 合并代理

就像在快照复制中看见的那样，合并复制也使用一个代理——合并代理。代理将会从所有的订阅服务器上复制变更，然后把这些变更应用到发布服务器上，如图 17-5 所示。接着它将会把发布服务器上的所有变更(包括在冲突解决过程中合并代理自己所做的变更)复制到订阅服务器上。如果是推订阅，那么合并代理通常会运行在分发服务器上，如果是拉订阅，那么合并代理通常会运行在订阅服务器上，但是与快照和事务复制一样，可以把它配置为远程运行。

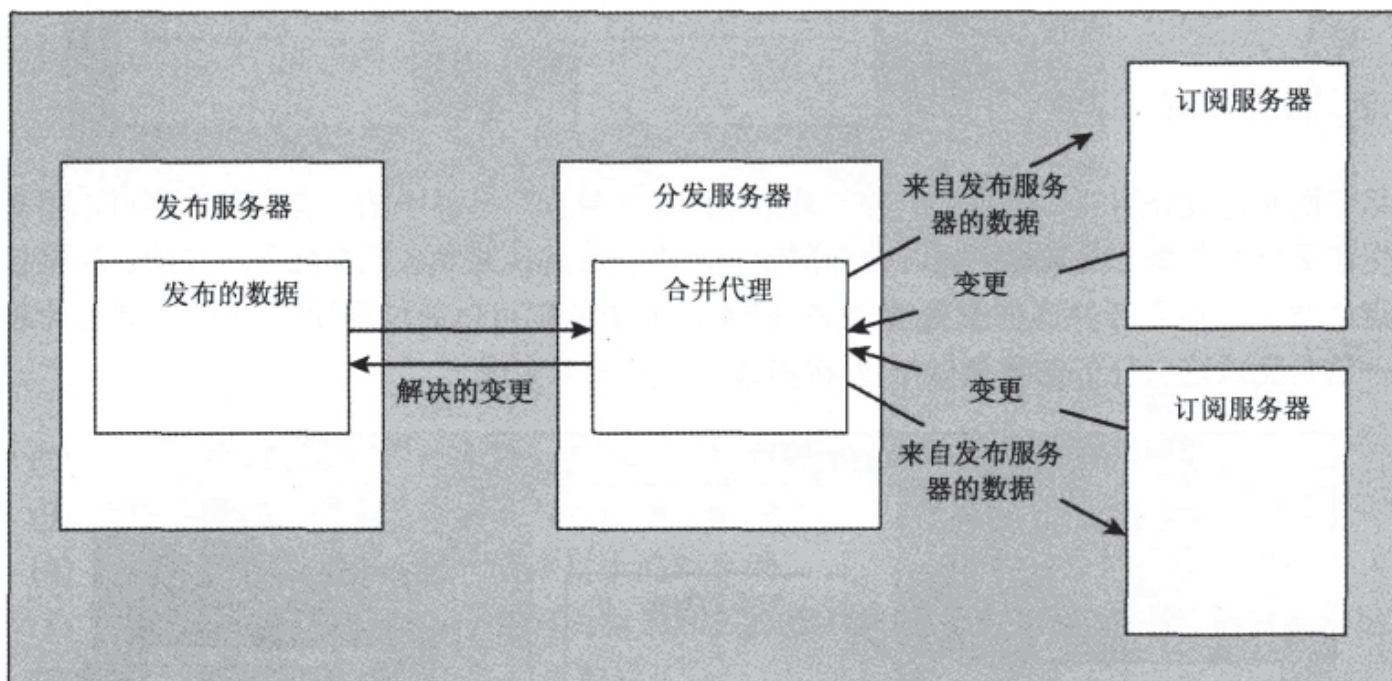


图 17-5

2. 合并复制的过程

假设已经进行了首次同步(记住这是以快照为基础的)，合并复制的步骤如下：

- (1) SQL Server 安装的触发器跟踪发生在被发布数据上的变更。
- (2) 在订阅服务器上应用来自发布服务器的变更。
- (3) 在发布服务器上应用来自订阅服务器的变更，同时解决所有的冲突。

注意：

合并触发器不会影响到用户自定义的触发器的放置或使用。

不管是发生在发布服务器还是订阅服务器上的变更都是由合并代理来应用的。此外，合并代理会自动使用冲突解决器(用户可以选择一个，甚至可以自己构建一个)来解决冲突。合并代理将会在行或列级别上跟踪每一行上的更新以确定是否存在冲突，当然这要取决于所配置的冲突解决方案。此外，用户还需要定义当在新(收到的)数据和当前数据值之间产生冲突时所需采用的优先级计划。

3. 何时使用合并复制

一种使用合并复制的情况是为了支持分区表。回到前面的花园供应业务中，读者可以设置筛选(分区)以使得每家商店都能够查看任何商店的库存信息，但只能直接更新自己的库存信息。而变更将通过合并复制来传播。可以水平或垂直筛选数据，还可以从表中排除要复制的行以及列。

合并复制将会监测被复制行中所有列上的更新。在这个特定的场景中，库存信息发生冲突的风险是很小的，因为每家商店只能更新自己的库存信息，但是如果允许所有商店都能更新客户数据(如为客户添加一个新地址)，那结果会怎样呢？这个问题的答案是视情况而定。这个例子说明不同的需求会对复制设计提出不同的要求。

4. 特别的计划需求

在实施合并复制时需要进行一些检查以确保要复制的数据已经准备好了。在进行合并复制的时候，SQL Server 可能会自动对数据库对象做一些变更。因此在选择要发布的表时要仔细考虑。如果需要在订阅服务器上应用验证(如查询表或其他的外键情形)，那么进行数据验证所需的所有表都应该被包含进发布中。

SQL Server 将会使用一列作为全局唯一的标识符来标识被发表中的每一行。如果表中已经存在了 `uniqueidentifier` 列，那么 SQL Server 将会自动使用那一列。否则，它会在表中添加一个 `rowguid` 列(当这种情况发生时，它也叫做 `rowguid`)并根据这一列来建立索引。

在发布服务器和订阅服务器上被发表的表中将会创建触发器。这些触发器用来跟踪行或列上的数据变更以供合并代理使用。

为了进行跟踪，还会添加几张表。服务器将会使用这些表来管理：

- 冲突检测和解决
- 数据跟踪
- 同步
- 报表

例如，冲突是通过 `MSmerge_contents` 表中的一列来检测的，该表是在建立合并复制时所创建的众多表中的一张。

17.2.3 事务复制

事务复制与快照复制之间的差别在于在事务复制中被复制到订阅服务器上的是变更增量，而不是整张表。所有记录下的发生在被发布项目上的变更(如 `INSERT`、`UPDATE` 和 `DELETE` 语句)都会被跟踪并被复制到订阅服务器上。在事务复制中，只有发生变化的表才会被分发，同时会维持相同的事务序列。换句话说，应用到发布服务器上的事务序列与应用到订阅服务器上的事务序列是完全一样的。

注意：

注意只有记录到日志中的动作才会被完全复制。未记录到日志中的大容量操作(如执行 `bcp` 操作时禁用了日志)以及不产生完整的日志项的二进制大对象(BLOB)操作是不能被正确复制的。

在最简单的形式中，变更只能发生在订阅服务器上，如图 17-6 所示。变更可以以预先设定的时间间隔或接近实时更新的方式被复制到订阅服务器上。虽然在复制进行的过程中无法进行更多的控制，但是每次复制所移动的数据通常都会变少。更新的发生将会更加频繁，延迟将会保持在最小。可靠的以及持续的接近实时(立即事务一致性)的订阅服务器更新要求在发布服务器和订阅服务器之间有一个可靠的网络连接(如果更新的频率非常高，且/或更新量非常大，那么请确保在发布服务器和订阅服务器之间的连接有足够的带宽来应付它们之间的对话)。

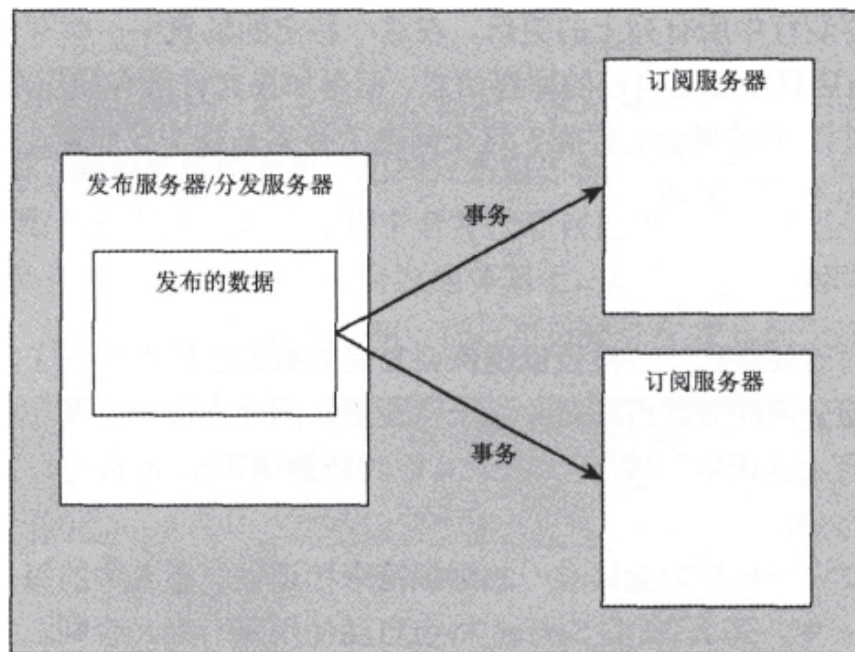


图 17-6

与合并复制一样,在进行事务复制之前,在发布服务器和订阅服务器之间必须要对被发布的项目进行首次同步。通常,这是通过自动同步来管理的,即使用快照复制。在自动同步不可行或效率较低的情形中,可以使用手动同步来让订阅服务器做好准备。这是一个相对简单的过程:

- (1) 运行 `BACKUP DATABASE` 来备份发布服务器的数据库。
- (2) 把磁带备份递送到订阅服务器系统。
- (3) 运行 `RESTORE DATABASE` 来创建数据库和数据库对象,并加载数据。

发布服务器和订阅服务器同步到备份运行的那个时间点。

还可以使用事务复制来复制存储过程。在最简单的实现中,变更只能发生在发布服务器上。这意味着用户不必担心会有冲突发生。

注意:

也可以把事务复制部署为带立即更新订阅服务器的事务复制。这意味着变更既可以发生在发布服务器上,也可以发生在订阅服务器上。发生在订阅服务器上的事务将被当做分布式事务来处理。Microsoft 分布式事务协调器(MS DTC)用来保证本地数据和发布服务器上的数据将会被同时更新以避免更新冲突。排队更新——更新将会被放置在一个有序的“待完成”列表中——被用来作为一种保障机制以防出现网络连接性问题,如连接断开或网络物理链路发生故障。

另一种方式是直接部署分布式事务,而不使用事务复制。这种方式比事务复制产生的延迟要小,但是在把变更从发布服务器传播到所有订阅服务器的过程中还是会存在分发延迟。假设服务器之间的连接非常稳固,当任何服务器上的数据发生变更时,分布式事务能够为所有的服务器提供几乎实时的更新。然而,取决于服务器之间的连接速度和可靠性,这可能会导致性能问题,包括锁的冲突。

1. 日志读取代理

在事务复制中会使用日志读取代理(Log Reader Agent)。在一个数据库被设置为使用事务复制之后,日志读取代理将会监视相关的事务日志以了解在被发布表上所发生的变更。然后代理将负责把那些标记为要复制的事务从发布服务器复制到订阅服务器,如图 17-7 所示。在事务复制中还会用到分发代理,它负责将事务从分发服务器移动到订阅服务器上。

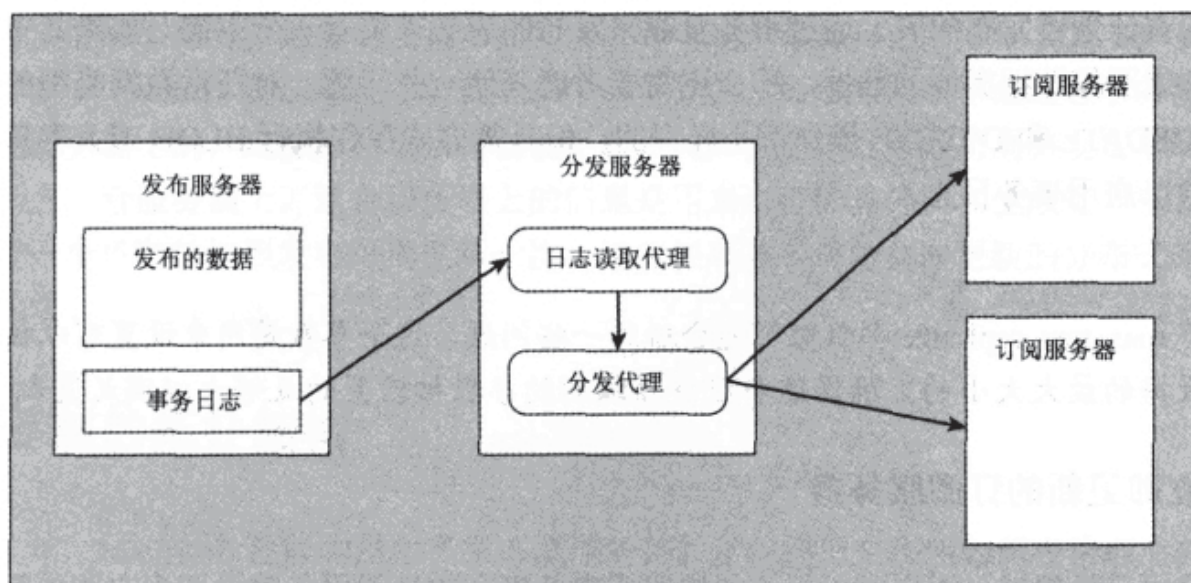


图 17-7

2. 事务复制过程

假设已经进行了首次同步，事务复制遵循下列基本步骤：

- (1) 更改被提交到发布数据库并被记录在相关的事务日志中。
- (2) 日志读取代理读取事务日志并确定标记为待复制的变更。
- (3) 从事务日志中确定的变更将会被写入到分发服务器上的分发数据库中。
- (4) 分发代理把变更应用于相应的数据库表中。

可以把日志读取代理设置为一直读取事务日志或者按照指定的工作计划读取事务日志。与之前一样，如果是推订阅，那么分发代理通常运行在发布服务器上，如果是拉订阅，那么分发代理通常运行在订阅服务器上，但是可以在 Management Studio 或 RMO 中把它配置成远程运行。

3. 何时使用事务复制

当需要或者仅仅想要减少延迟以及给订阅服务器提供相对及时的信息时可以使用事务复制。几乎实时的更新通常要求局域网的连接，但是计划复制通常可以通过计划更新来管理。如果选择使用计划更新，那么延迟会增加，但是在复制过程中也可以获取更多的控制。

现在回到前面讨论的花园供应商店和库存问题上。用户希望每家商店能够具有最新或至少相对新的库存信息。因此可能会使用计划复制来把数据传送到订阅服务器上。

现在看看能否使事情变得更复杂一点。假设现在不仅有连锁商店，还有旅行销售人员，他们将访问最大的客户并获取订单。他们需要至少相对新的库存信息，但是不能花几天时间坐等来自发布服务器的更新。这种类型的系统可以采用拉订阅，让销售人员决定何时连接到服务器下载最近的事务。

对于这两种场景，读者可能已经注意到了一个问题。远程服务器可以接收数据，但是它们却无法修改数据。本章后面将会讨论这个问题。当事务复制以这种方式来实现时通常用以支持订阅服务器系统上数据的只读副本。

4. 特别的计划需求

在为事务复制制作计划时，空间是一个需要考虑的重要问题。需要确保在发布服务器上有足够的空间来存储事务日志，而在分发服务器上需要有足够的空间来存储分发数据库。

检查所有计划要发布的表。通过事务复制来发布的表必须要有一个主键。如果其中有些表支持存放文本或图像数据类型的数据,那么还需要考虑其他一些问题。对任意数据类型的数据进行 INSERT、UPDATE 和 DELETE 操作都是允许的,但是需要确保在执行 BLOB 或大容量操作时使用一个选项以启用事务日志。

注意:

在使用 max text repl size 参数时可能会碰到一些问题,这个参数是用来设置可以被复制的文本或图像数据的最大大小的。确保这个服务器级别的参数被设置的足够大以满足复制的需要。

17.2.4 立即更新的订阅服务器

本章前面已经提过,用户可以把订阅快照或事务发布的订阅服务器设置为立即更新的订阅服务器。立即更新的订阅服务器能够更新被订阅的数据,同时更新将会立即反映到发布服务器上。这是通过 MS DTC 管理的两阶段提交协议来实现的。这是一种有效的在更新发布服务器时不会带来延迟的方法。而订阅服务器的更新则会正常进行(就好像变更最初发生在发布服务器上一样),因此对其他订阅服务器的更新延迟则取决于那些订阅服务器的更新速率。

当需要提交发生在一个或多个订阅服务器上的对被复制数据的变更并且需要以几乎实时的方式传播这些变更时可以考虑采用立即更新的订阅服务器。用户可能正在使用多台服务器来支持联机事务处理(OLTP)应用程序以提高性能并提供几乎实时的冗余。当任意一台服务器上发生了一个事务时,该事务会被发送到发布服务器上,然后通过这台发布服务器来把事务传播到剩余的服务器上。

注意:

与任意形式的合并复制有很大的相似之处,当使用立即更新的订阅服务器时可能会产生冲突。为了辅助冲突识别和管理,一个名为 uniqueidentifier 的列将会被添加到所有被发布但还不存在该列的表中(如果表中已经存在该列,那么该列将会有有一个列级别的属性 IsRowGUID,同时这个属性的值为 true——每一张表中只能有一个 RowGUID 列)。

除非使用排队更新,否则在发布服务器和任何立即更新的订阅服务器之间需要有一个高速并可靠的连接,如局域网连接。如果配置了排队更新,那么复制进程就可以容忍不可靠的连接,当连接恢复时它可以处理所有排队的事务。

注意:

记住排队更新增加了发生冲突的可能性。因为订阅服务器可能正在修改数据,而发布服务器并不知道,因此发布服务器与订阅服务器修改同一行的可能性就增大了。如果它们确实修改了同一行数据,那么在复制时冲突解决器将会识别出冲突的存在并根据已确立的规则来解决冲突。

17.2.5 混合复制类型

用户可以根据需要混合搭配复制类型。实际上,不仅可以在同一台服务器上配置不同的复制类型,还可以为同一张表配置不同的复制类型。

那为什么要这么做呢?考虑这样一个例子,假设一个大型设备仓库希望拥有最新的库存信

息,同时希望能够引用各个地方的发货单的副本。每个地方都有自己的本地 SQL Server。发货单是通过一个基于 Internet 的应用程序被提交到中心位置的。而这些发货单是通过事务复制被复制到所有本地服务器上的,这样库存记录将会被更新。此外,用户还希望每周将发货单和库存信息复制更新到另一台服务器上。这台服务器上的信息是用来进行业务分析和生成每周报表的。这台服务器通过一个单独的快照发布每周更新一次,这些快照发布将与立即更新的分布式库存服务器引用同样的表。

17.3 复制的拓扑结构

最近几年,Microsoft 已经勾画出了很多复制的拓扑结构模型来说明如何在物理上部署复制。下面将介绍其中一些模型以了解如何部署它们。需要注意的是,可以混合和修改这些模型,实际上这种做法是相当常见的。

注意:

决定采用何种复制类型与决定采用何种复制模型拓扑在某种程度上是相互独立的。也就是说,物理拓扑的限制(如传输带宽)可能会影响用户所做的选择。

17.3.1 简单模型

本节从简单的模型开始。一旦读者有了基本的概念后就可以介绍模型的一些变体以及如何混合这些模型。

1. 中心发布服务器/分发服务器

这是默认的 SQL Server 模型。在这个模型中有一个系统作为发布服务器和它自己的分发服务器,如图 17-8 所示。发布服务器/分发服务器支持任意数量的订阅服务器。发布服务器拥有所有的被复制数据,并且是复制的唯一数据源。这个最基本的模型假设所有发布到订阅服务器上的数据是只读数据。通过只把被复制表上的 SELECT 权限赋给用户可以在订阅服务器上强制只读访问。

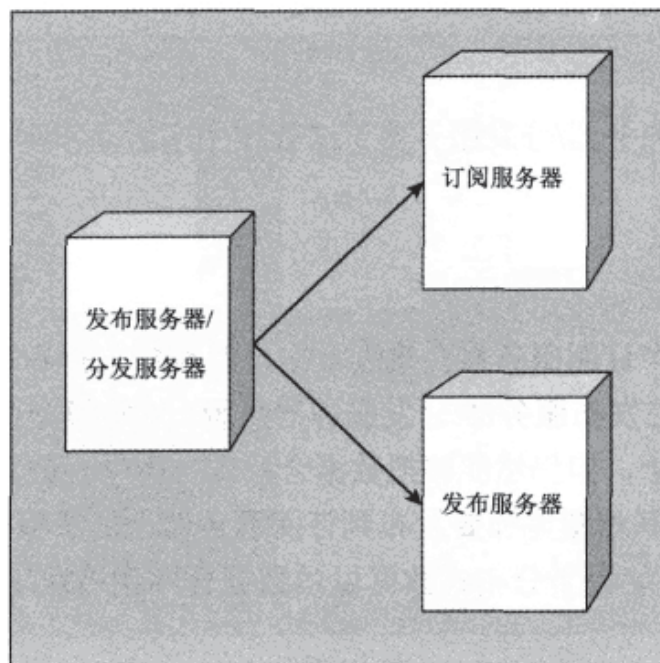


图 17-8

由于这是最容易设置和管理的模型，因此任何合适的场合都应该考虑使用这个模型。如果只有一个发布服务器，有一个或多个订阅服务器，同时对订阅服务器上的数据是只读访问的，那么这个模型是最好的选择。

2. 中心发布服务器/远程分发服务器

被复制的数据量和/或发布服务器上的活动量可能会要求发布服务器和分发服务器分别部署在不同的系统上。从操作的角度来看，这实际上与发布服务器/分发服务器模型是一样，如图 17-9 所示。发布服务器仍然是被复制数据的所有者，同时也是被复制数据的唯一的源。同样，这个简单的模型假设订阅服务器上的数据是只读的。

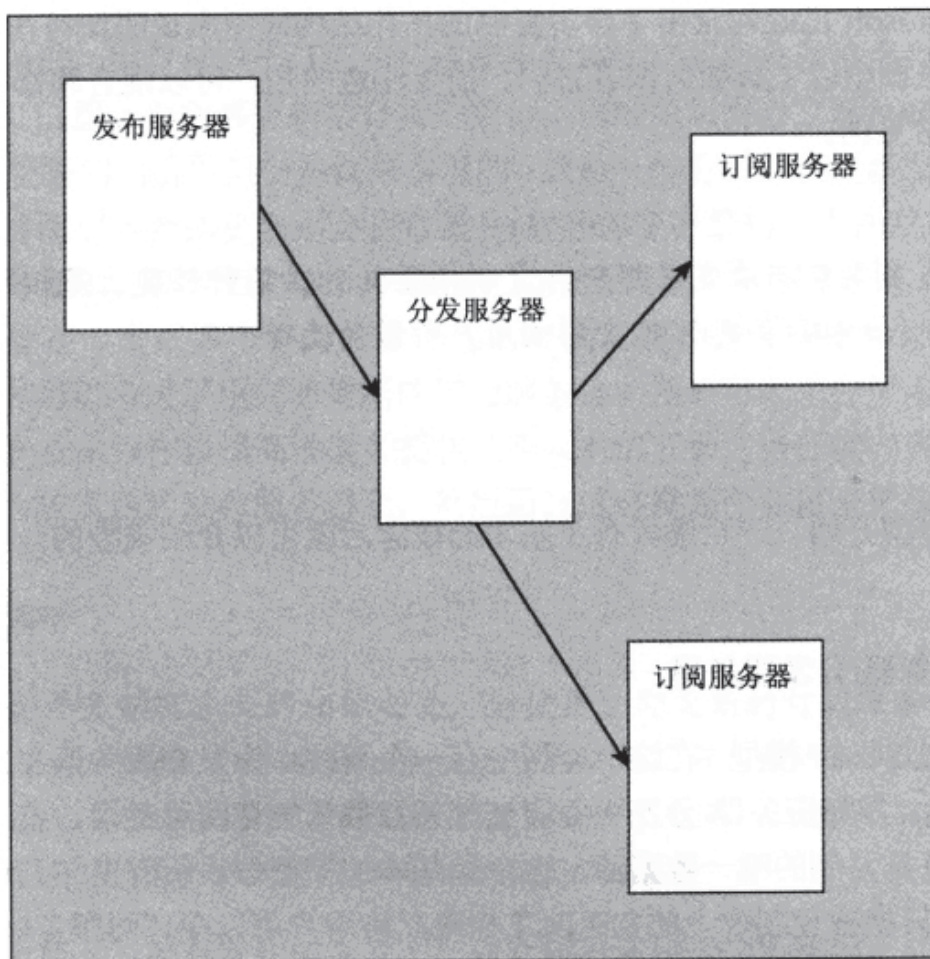


图 17-9

显然，只有当单个发布服务器/分发服务器无法处理生产活动和到订阅服务器的复制活动时，才会考虑使用这个模型。

3. 中心订阅服务器

在这个模型中，只有一个订阅服务器会接收数据，但是发布服务器却有多，如图 17-10 所示。可以把发布服务器配置成发布服务器/分发服务器系统。这种模型提供了一种方式使得可以把本地数据保留在本地服务器上，但仍然能够把数据合并到一个中心位置上。此外，可能需要进行水平筛选以防止发布服务器互相覆盖掉要发布到订阅服务器上的数据。

当有数据合并的需求时(如收集分布式数据以供数据仓库使用)可以考虑使用这个模型。

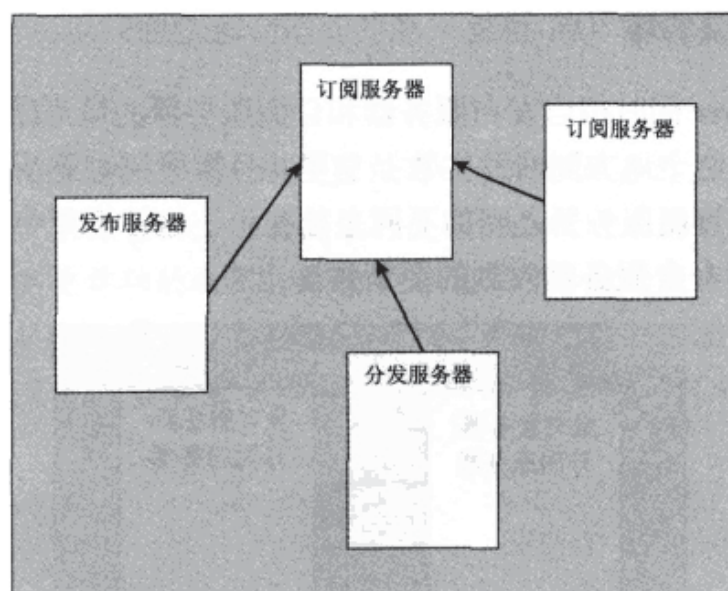


图 17-10

17.3.2 混合模型

基于经常需要混合搭配模型的想法，本节将会介绍几个模型的变体。可以把它们看成仅仅是对各种可能性的一种尝试——“只是一个开始”。组合的可能性几乎是无穷无尽的。

1. 发布订阅服务器

注意：

发布订阅服务器(即把订阅服务器也配置成发布服务器)可以被添加到任意的基本模型中。这个模型有两个发布服务器，它们发布同样的数据。原始的发布服务器把数据复制到它的订阅服务器上，其中一个发布订阅服务器。接着发布订阅服务器可以把相同的数据传递给它的订阅服务器。

当有许多服务器或者服务器之间的链路特别慢或昂贵时，这种模型是非常有用的，如图 17-11 所示。另一种可能的情况是原始发布服务器与所有潜在的订阅服务器之间并没有直接的连接。发布服务器只需要把数据传递到链路远端的一个系统上，然后发布订阅服务器会把数据传递给其他订阅服务器。

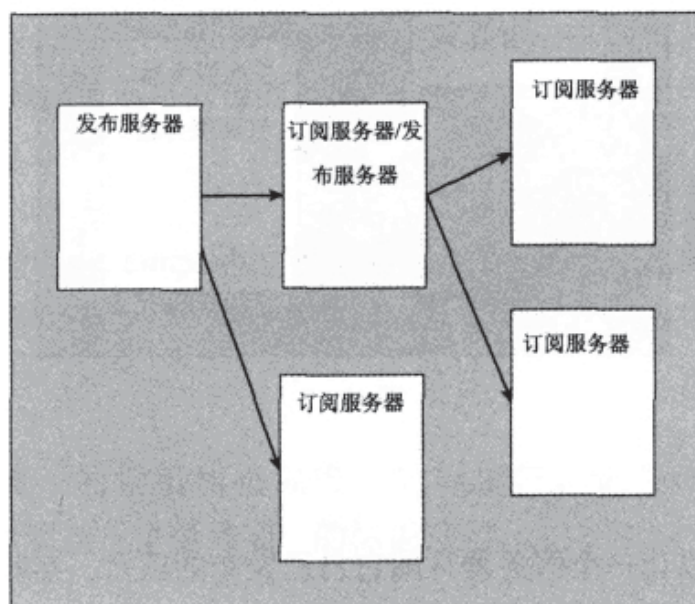


图 17-11

2. 发布服务器/订阅服务器

这是另一种 SQL Server 同时担当发布服务器和订阅服务器的情形(图 17-12)。每个服务器都有自己负责的数据集。当在两个地方同时发生数据变更并且需要同时更新两个服务器时可以使用这种模型。这种模型与发布订阅服务器之间的不同之处在于在这种模型中, 每个服务器将会产生自己的数据, 而不是把从另一台服务器收到的更新传递出去。

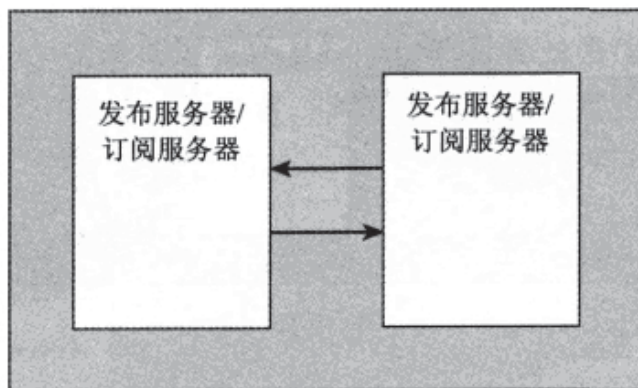


图 17-12

3. 多个订阅服务器/多个发布服务器

图 17-13 显示了其中一个更复杂一点的场景。在这个场景中有多个发布服务器和多个订阅服务器。系统可能作为(也可能不作为)发布服务器/订阅服务器或者发布订阅服务器。使用该模型要求进行非常仔细的规划以提供最优的通信以及确保数据一致性。

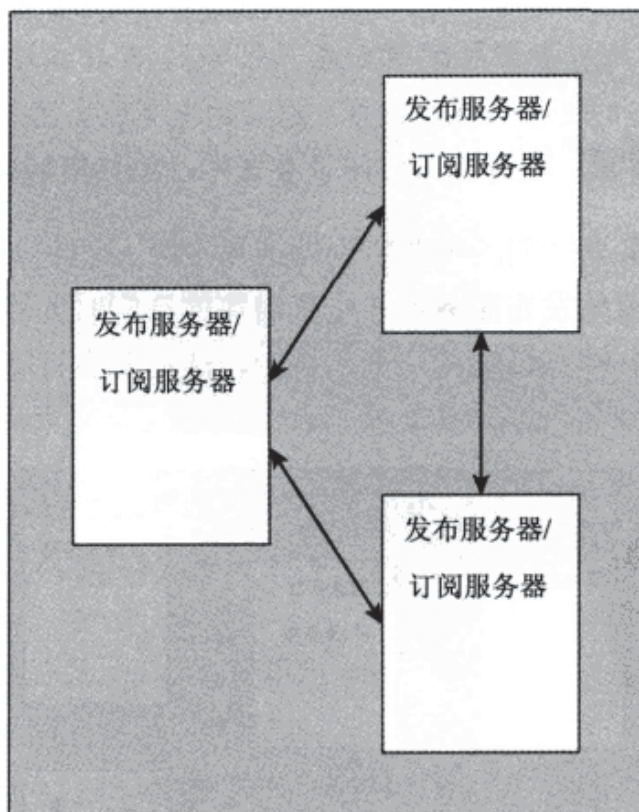


图 17-13

4. 自发布

特别值得一提的是可以让一个服务器订阅它自己发布的项目。实际上, 这在小型安装中是非常常见的, 因为虽然小型安装中的会有各种各样的需求, 但是其负载一般不会超过一台服务器的承受能力。例如, 用户可能想要把用于联机事务处理和用于决策的数据隔离。这可以通过使用复

制来创建相互隔离的数据(可以按照任意合适的计划来更新)的只读副本以供参考使用。

提示:

是否把其他数据库——如数据仓库——放置在同一台物理服务器上作为核心系统要根据个人喜好和特定的场景而定。当事务量较少但需要进行复杂的分析时采用这种方式是非常合适的。根据笔者的经验,那些非常需要单独的数据仓库的公司通常对单独的服务器也有物理的或操作上的需要,但这离“总是”这样的场景还差得很远。因此需要对特定情况进行考虑——服务器上是否有足够的空间来共享这些负载?当发生灾难事件时两个数据库同时脱机的风险能够承受吗?

17.4 制定复制计划

复制是那些可以很容易进行“只是把一些东西混合到一起”的事情之一。但是如果用户不仔细规划,那么它也是那些很容易造成巨大混乱的事情之一。记住实施复制时 SQL Server 可能会自动对模式进行一些修改——在未经全盘考虑的情况下是否真的想要 SQL Server 向数据库中添加列和对象呢?当然不是。

任何值得做的复制安装都值得花一些时间来好好规划。规划时需要考虑的问题包括:

- 待复制的数据是什么
- 复制类型
- 复制模型

除了这些因素之外还有其他一些因素会影响决策,如当前的网络拓扑、当前的服务器配置、潜在的服务器增长和活动级别等等。每种复制方法都有其优点和缺点,没有哪种方法能够满足所有数据复制的需求。例如,如果网络连接较慢或者连接不可靠,那么可能就无法采用事务复制。相反,如果复制是在计划连接时间期间进行的,那么用户可能会倾向于使用合并复制。此外,正如本章中反复指出的那样,用户还需要权衡一致性的需求。

17.4.1 涉及的数据

首先需要考虑的是要发生什么以及发布给谁。用户需要识别出项目(待发布的表和特定的列)以及计划如何把它们组织进发布中。此外,还需要注意其他一些数据问题。其中一些问题已经提及过,但是花一些时间回顾一下还是值得的。

1. timestamp

在事务复制中包含一个 `timestamp` 列。它提供了一种方式以便在更新时检测冲突。如果已经有了一个 `timestamp` 列,那么就已经满足增加立即更新订阅服务器的一部分要求了。

2. uniqueidentifier

合并复制需要一个唯一的索引和全局唯一的标识符。记住,如果被发布的表中没有 `uniqueidentifier` 列,那么会添加一个全局唯一的标识符列。

3. 用户自定义的数据类型

除非用户自定义的数据类型在订阅服务器的目标数据库中存在, 否则用户自定义的数据类型是不被支持的。作为一种替代方法, 用户可以在同步期间把用户自定义的数据类型转换为基本数据类型。

4. NOT FOR REPLICATION

NOT FOR REPLICATION 子句使得用户可以禁用订阅服务器上的表的动作。用户可以禁用:

- IDENTITY 属性
- CHECK 约束
- 触发器

当且仅当复制进程修改订阅服务器上的数据时才会忽略这些动作。任何其他进程还是像平常一样使用它们。因此, 向原始的接收数据库中插入数据将会指派一个标识值, 但是当该行随后被发布(以 INSERT 的形式)到订阅服务器上时将会使用现有的标识值, 而不会生成一个新值。

17.4.2 移动设备

SQL Server 还存在“移动的”版本。这个版本的 SQL Server 占用的空间极少, 是为 Windows Mobile Edition 而设计的。从订阅服务器的角度来看, 移动版本是支持复制的。支持快照和合并复制——不支持事务复制。

很多针对移动设备需要考虑的问题只是之前在复制中讨论过的某些主题的变体——如带宽和空间。记住, 与完整的服务器类系统相比(甚至是销售人员的笔记本电脑), 移动设备的限制就要大得多了。

17.5 在 Management Studio 中设置复制

设置复制通常需要几个步骤。特别是需要:

- 配置发布服务器和分发服务器以准备执行这些任务
- 配置真正的发布
- 配置订阅服务器

下面介绍如何在 Management Studio 中完成这些步骤。

17.5.1 为复制配置服务器

在服务器上配置发布或分发之前首先需要针对复制对服务器进行配置。

为了在 Management Studio 中完成这个任务, 导航到“复制”节点, 右击并选择“配置分发”。

注意:

注意, 为了配置复制, 用户必须要使用服务器的真实名称(不支持 local、句点(.)、localhost 以及 IP 地址)来连接到“对象资源管理器”上。如果不使用服务器的 DNS 名连接服务器, 那么将会得到一个错误信息并且需要重新连接。

SQL Server 会使用一个标准的启动窗口欢迎用户，读者在其他向导中已经见过这个窗口，接着将会进入到简介对话框——在本例子中，它将列出读者将会在本向导中碰到的一些选项。单击“下一步”，接着将会进入另一个对话框(如图 17-14 所示)，在这个对话框中需要决定是否把这个发布服务器配置成它自己的分发服务器或者使用一个现有的分发服务器。

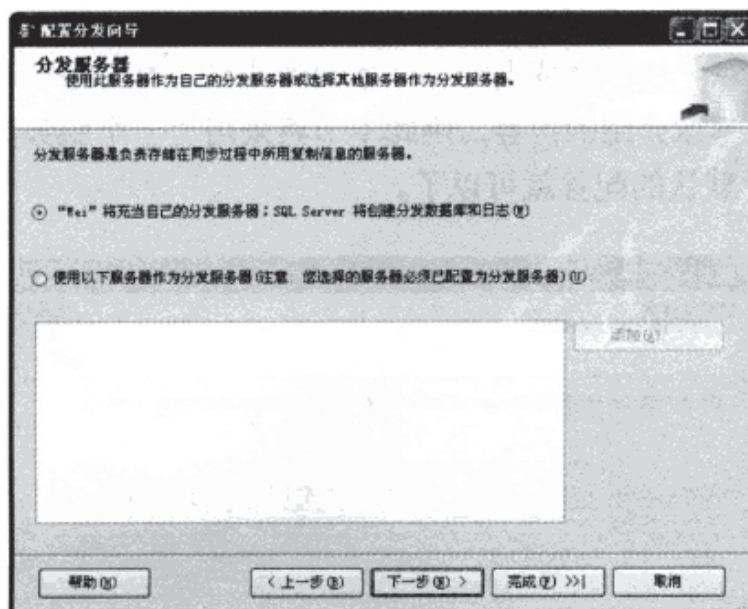


图 17-14

如果选择使用另一个服务器作为分发服务器并选择“添加”，那么会弹出一个标准的连接对话框(请求输入分发服务器的登录帐户安全信息)。在本例中将保留默认的选项(即这个服务器将担当它自己的分发服务器)，单击“下一步”。

注意：

在分发服务器对话框之后到底出现哪个对话框要取决于有没有把 SQL Server 代理配置成随系统启动而自动启动。

如果没有把 SQL Server 代理配置成自动启动(尽管在一个生产服务器上几乎是肯定希望它自动启动的)，那么 SQL Server 将会弹出一个对话框询问相关信息(如果已经把代理配置成随系统自动而自动启动，那么这个对话框将会被跳过)，如图 17-15 所示。

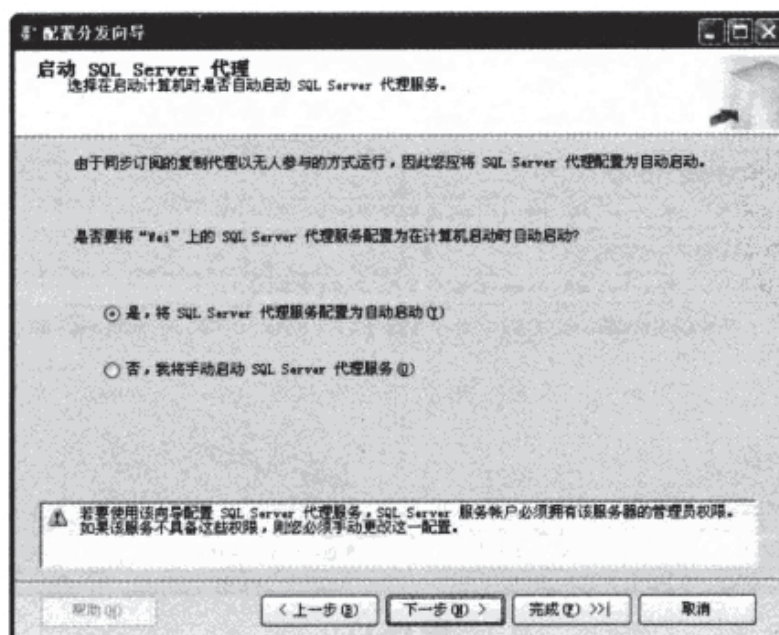


图 17-15

如果系统上已经配置了 SQL Server 代理,那么可以保留原先的配置(但是 SQL Server 会在对话框中默认把 SQL Server 代理服务修改为自动启动),但是需要记住的是一些类型的复制需要代理运行才能工作。

单击“下一步”。接着需要配置一个快照文件夹,如图 17-16 所示。这个文件夹默认将会被配置成主 SQL Server 文件夹下的一个目录,对于很多安装来说,这个目录可能没有足够的空间来存放大型数据库的快照。可以把快照文件夹配置成本地卷或者 UNC 路径。由于读者可能并没有一个完整的服务器场来试验本章介绍的内容,因此本节将采用“一个服务器做所有事情”的方式来完成这个例子,那么接受默认的配置就可以了。

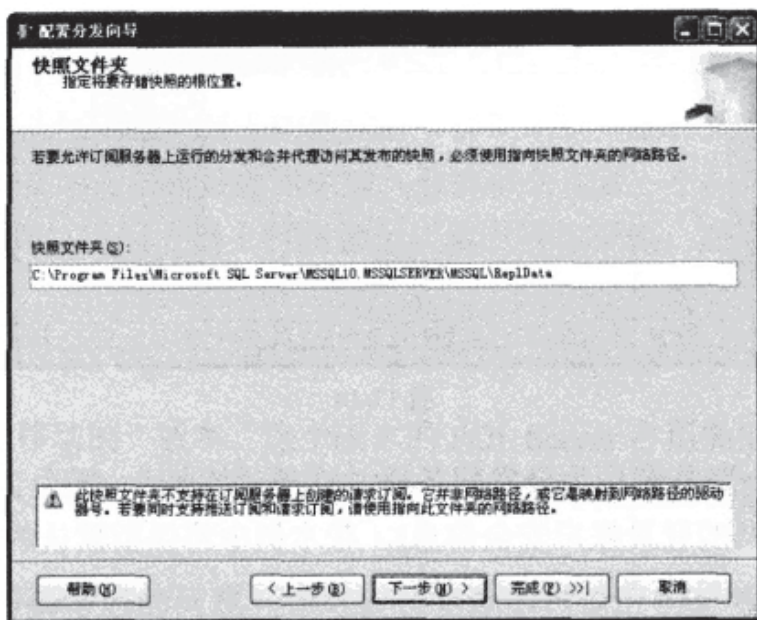


图 17-16

现在要开始配置真正的分发数据库了。SQL Server 将会通过一个对话框来获取一些典型的数据库创建信息(数据库的名称是什么以及把数据库存储在什么地方),如图 17-17 所示。

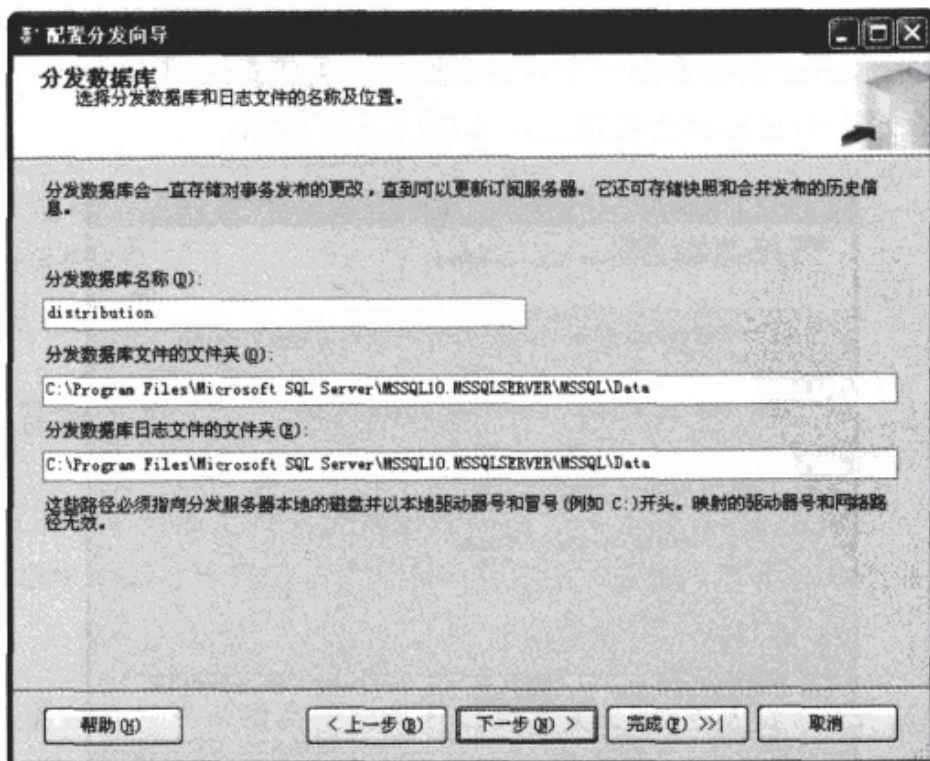


图 17-17

接着将会进入一个一开始看起来相当乏味的对话框(如图 17-18 所示),它看上去没有什么新东西。

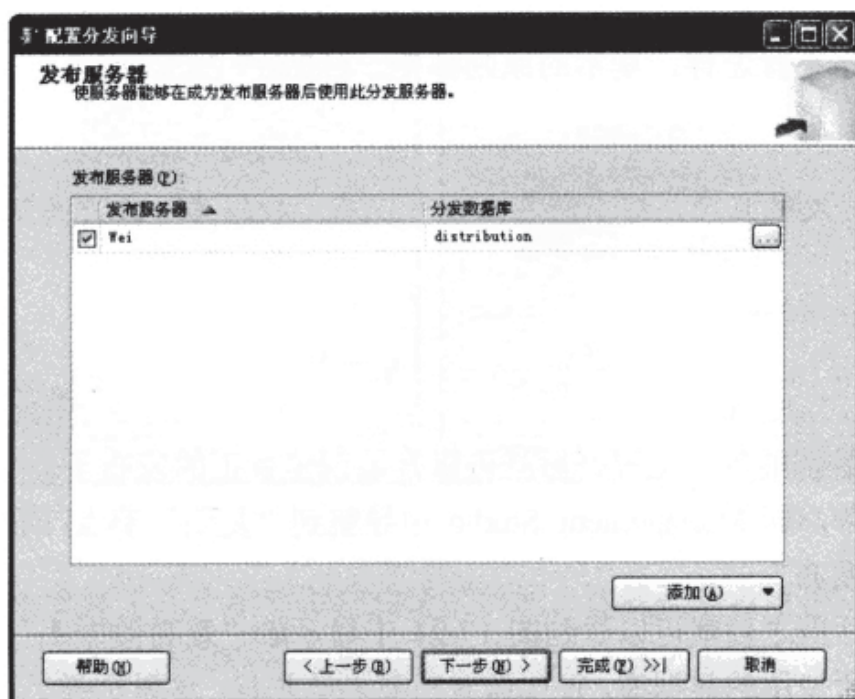


图 17-18

但是,表面现象有时候会欺骗人的。如果单击右边的省略号(...),那么会出现另一个对话框(如图 17-19 所示)——这个对话框中包含了需要注意的关键项目。

正如图 17-19 中所示的那样,用户可以专门配置代理到发布服务器的连接模式。在大多数情况下,默认的模拟代理进程是能够工作的,但是需要记住的是,可以根据需要使用专门的 SQL Server 安全凭据。

取消这个属性对话框,然后单击“下一步”回到发布服务器对话框(图 17-18 所显示的那个)。在向导的最后会显示确认对话框,在对话框中将会列出所需做的配置任务,如图 17-20 所示。注意它所提供的不仅仅是立即配置分发的选项,并且还提供了为配置生成脚本以供后面或远程使用的选项。

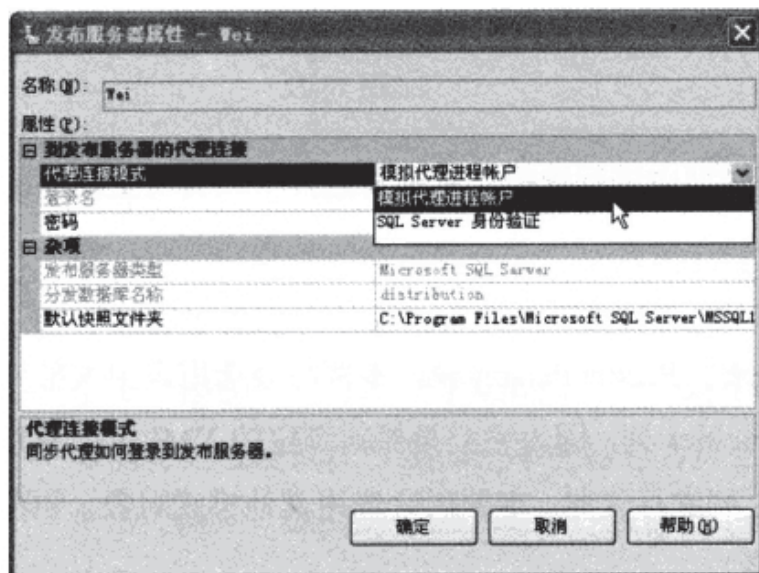


图 17-19

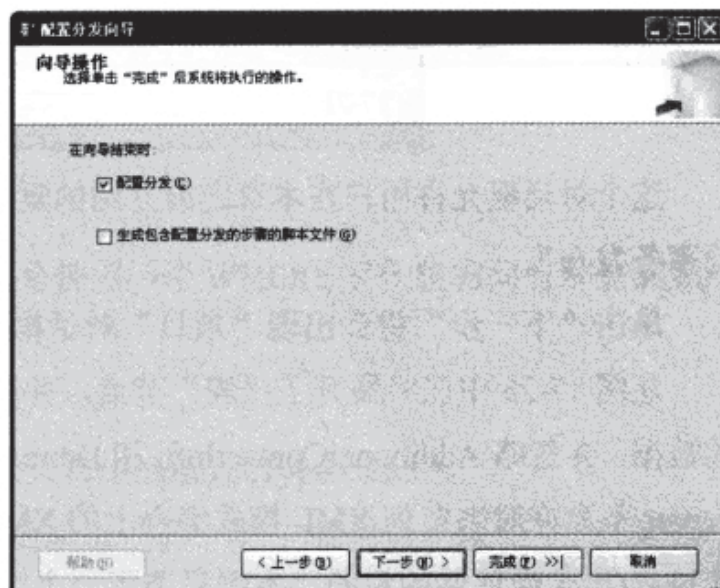


图 17-20

继续向前并单击“完成”(下一个对话框只是一个小结,因此不需要在上面花费时间)。接着

SQL Server 将会开始处理配置请求。当处理完成之后就继续向前并关闭这个对话框。

就是这么快，现在已经发布和分发被复制数据的服务器配置好了。显然，如果这个是一个生产环境，那么还需要做一些其他的选择，如特定的位置或是否想要把发布服务器和分发服务器配置在同一系统上，但是不管怎样，基本的原则都是一样的。

提示：

如果读者想知道分发数据库，那么现在应该可以在“数据库”文件夹下的“系统数据库”子文件夹下找到它。

17.5.2 配置发布

当把所有的服务器都准备并配置好后就可以开始创建真正的发布了。

为了完成这个任务，在 Management Studio 中导航到“复制”节点，右击“本地发布”子节点，然后选择“新建发布”。

在常规的简介对话框之后就可以看到图 17-21 中显示的“发布数据库”对话框了。这个对话框允许用户选择发布所使用的数据库。正如读者所看到的那样，这里选择了 AdventureWorks2008。

单击“下一步”，接着将会出现图 17-22 中显示的“发布类型”对话框。

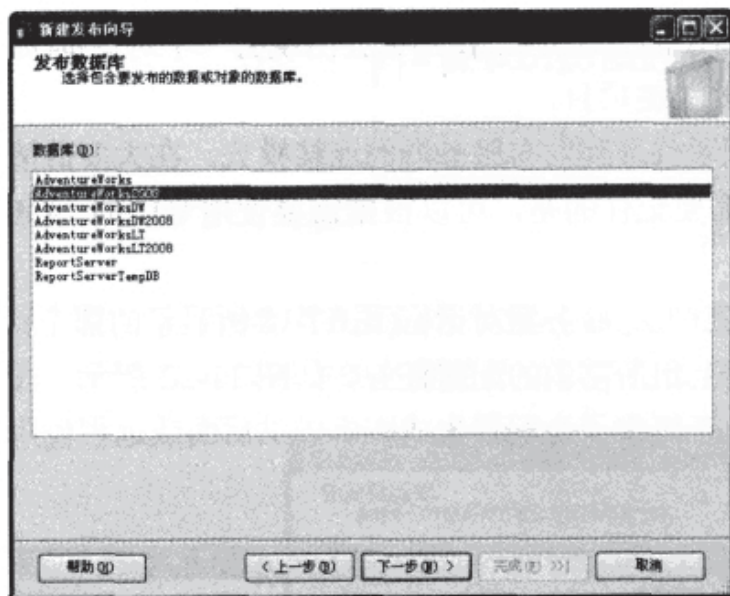


图 17-21

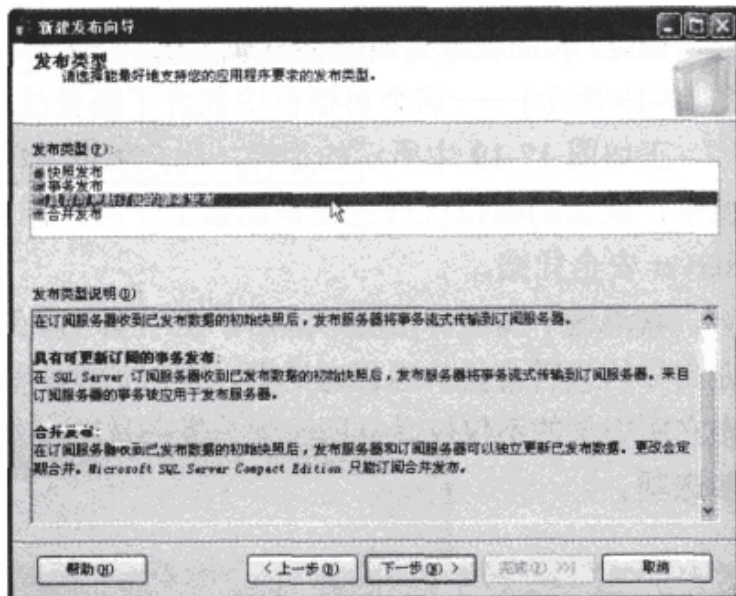


图 17-22

这个对话框允许用户在本章之前介绍的复制类型中进行选择。本例将选择“具有可更新订阅的事务发布”。

单击“下一步”将会出现“项目”对话框。

在图 17-23 中已经展开了“表”节点，并选择了 Person.Person 表。本例将会选用表中大部分的数据，并忽略 AdditionalContactInfo 和 Demographics 列，因为它们是模式绑定的 XML 列，SQL Server 不允许对绑定到 XML 模式集合上的 XML 列进行复制。本例还会选用其他模式对象，如存储过程(出于简单性的目的，本例只选了一个对象)。

单击“下一步”，接着会出现“项目问题”对话框，如图 17-24 所示。

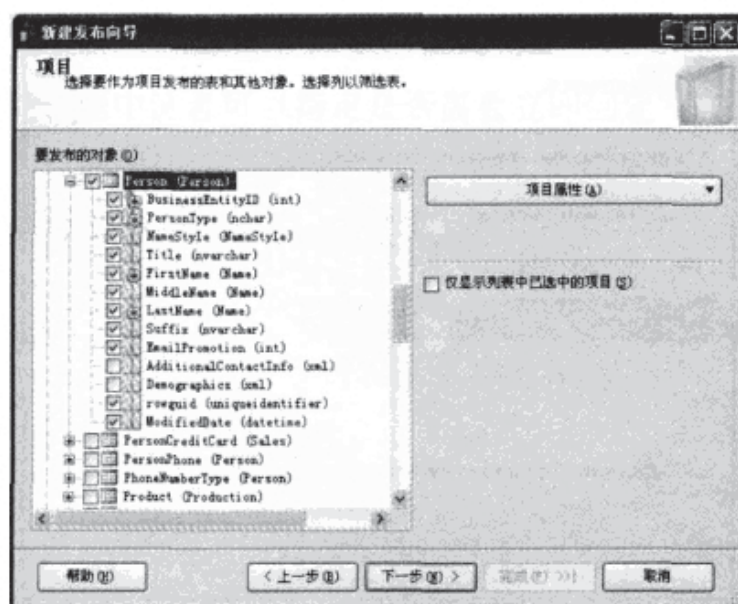


图 17-23

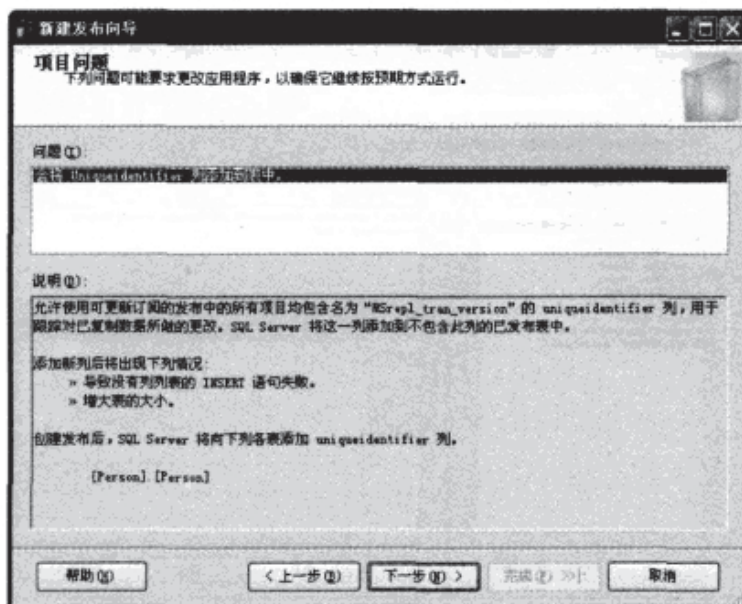


图 17-24

注意 SQL Server 检测到了几个希望用户知道的问题。这就是笔者所说的“SQL Server 团队的辉煌成就”之一，他们让一些基础的事情成为问题之前就让用户知晓。

单击“下一步”，接着会出现“筛选表行”对话框，如图 17-25 所示。

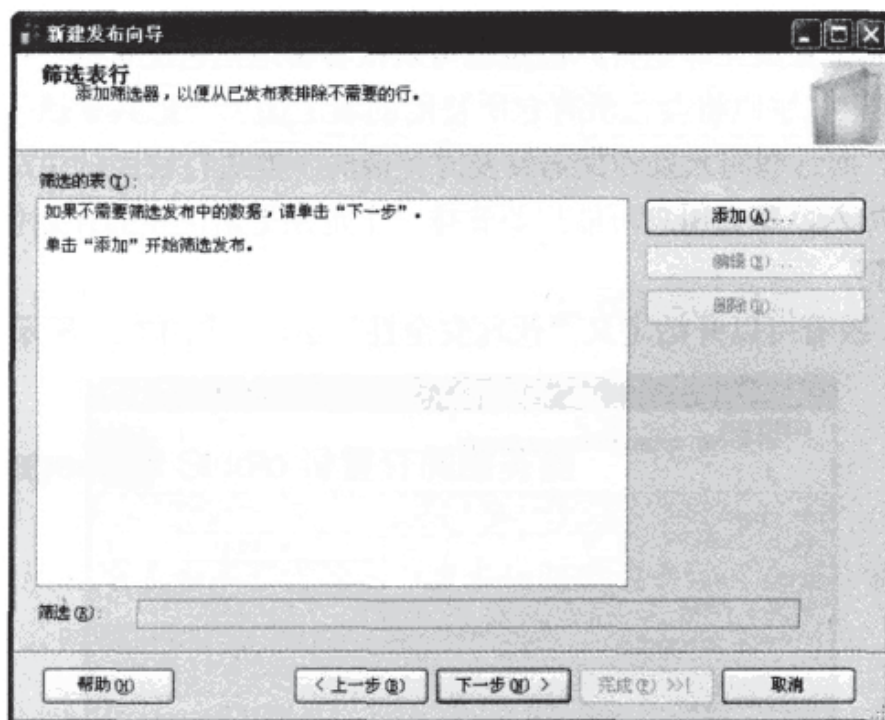


图 17-25

这个对话框允许进行水平分区——本质上仅仅是使用一个 WHERE 字句使得只有那些满足特定条件的行才会被加到发布中。

单击“添加”会出现图 17-26 所显示的对话框。

本例将把被复制的行限制在那些标记为员工的人员(EmployeeType='EM')。

单击“确定”返回到“筛选表行”对话框，然后单击“下一步”，接着会出现图 17-27 所显示的“快照代理”对话框。

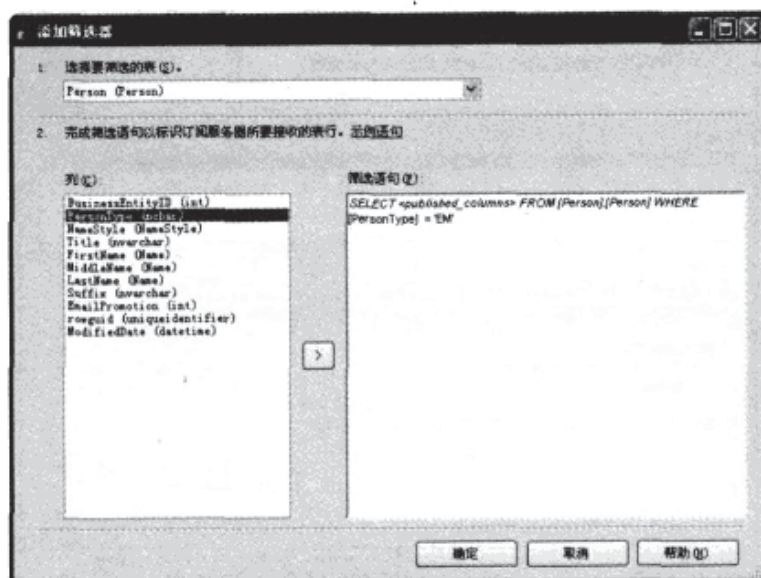


图 17-26

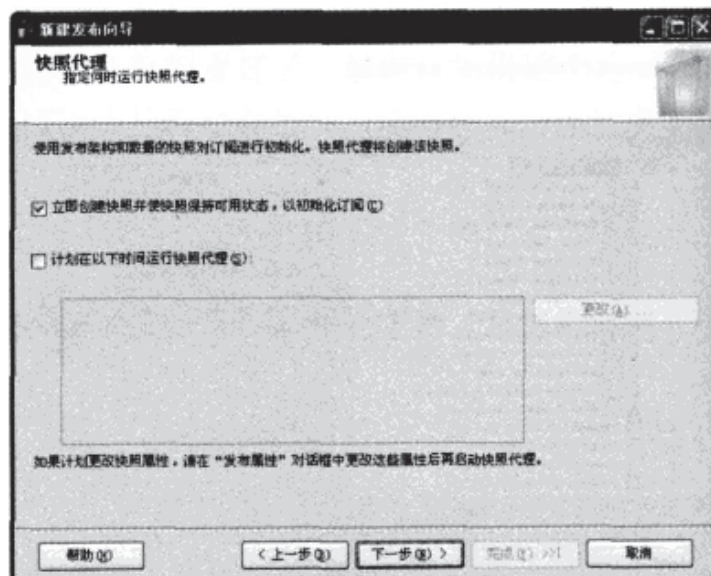


图 17-27

注意:

记住所有的订阅都必须以一个快照为基础开始同步, 不管它订阅的是快照、合并还是事务复制模型。后续的变更都是相对于这个快照开始的。

本例将把这个快照配置成立即运行, 但是也可以很容易地把它配置成一个计划任务以在之后的某个时间生成快照(记住快照将会在所有它所使用的表上加上一把共享锁——当这种锁问题会阻塞对数据库的写入, 而这些写入操作又需要及时完成时不要进行快照的生成)。例如, 如果经常会有新的订阅服务器加入, 那么用户可能想要安排一个定期更新快照的计划使得订阅服务器能够同步到尽可能新的数据。

单击“下一步”, 接着可以开始定义“代理安全性”了, 如图 17-28 所示。

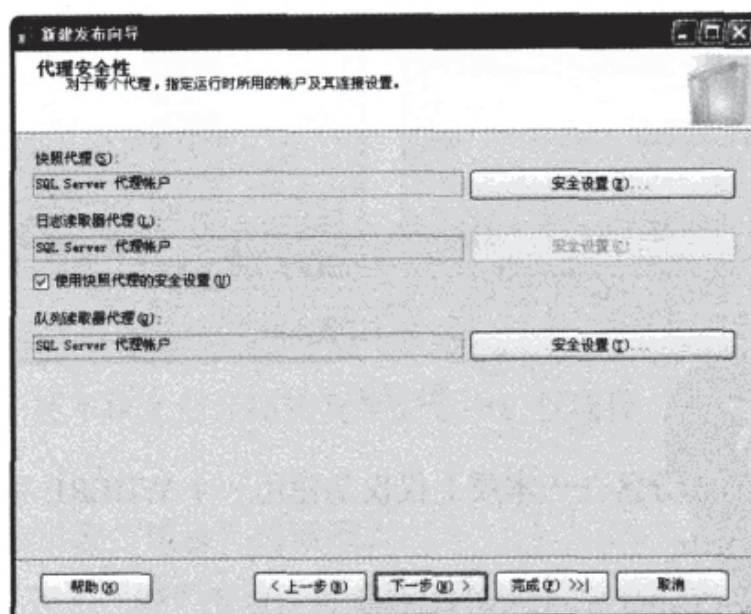


图 17-28

提示:

本例将使用“安全设置”对话框把代理设置成使用 SQL Server 代理帐户。但是, 在生产环境中, 出于安全性的原因, 这并不是一个好的做法。给代理它们自己的帐户, 这样既可以限制代理的访问, 也可以增加审计的能力。

单击“下一步”，接着会出现“向导操作”对话框(类似于图 17-20 中显示的对话框)，在这个对话框中读者可以指定是否需要立即创建发布或者作为计划任务稍后执行。

再次单击“下一步”按钮，接着会出现小结信息，现在可以为发布定义一个名称了，如图 17-29 所示(本例中选择 Employees)。

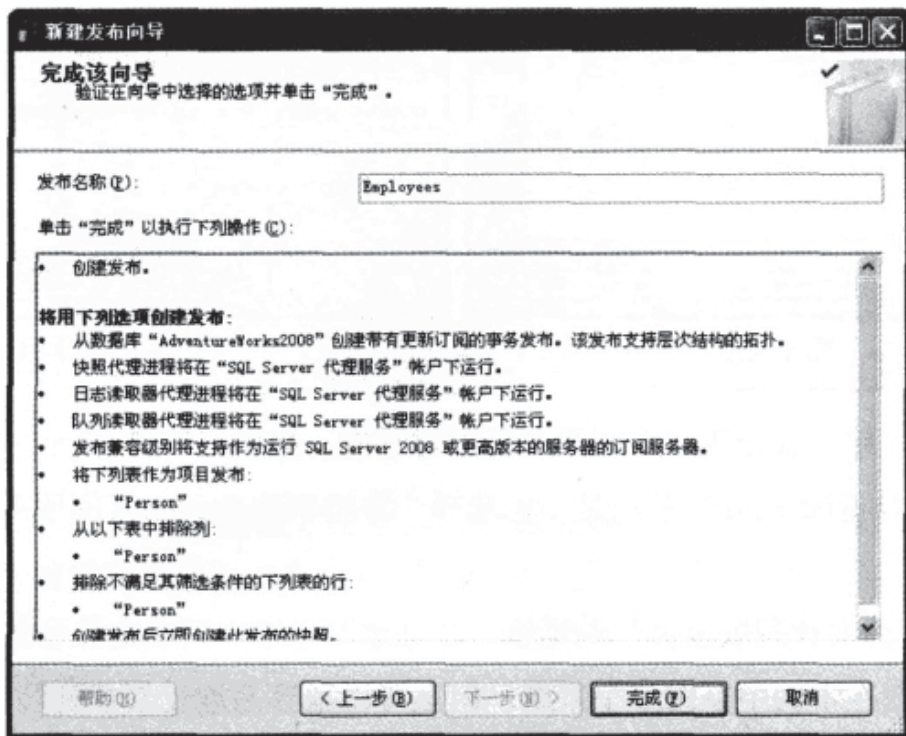


图 17-29

继续向前并单击“完成”按钮以创建发布，与这个过程类似，现在已经可以开始创建订阅服务器了。

17.5.3 通过 Management Studio 设置订阅服务器

设置订阅服务器所遵循的基本理念与在设置发布时所遵循的理念是一样的。在开始本例之前，首先设置一个伪数据库来担当订阅服务器的角色：

```
CREATE DATABASE AWSubscriber;
```

当把这个数据库创建好之后就可以开始订阅一些数据了。

在 Management Studio 中右击“复制”节点下的“本地订阅”子节点，然后选择“新建订阅”。在常规的简介对话框之后将会开始识别发布，如图 17-30 所示。由于现在只有一个发布，因此没有什么可供选择的，但是这个列表很容易就会成为包含很多很多发布的列表。

单击“下一步”，接着会出现“分发代理位置”对话框，如图 17-31 所示。记住既可以在订阅服务器上运行复制代理，也可以在分发服务器上运行复制代理。在本例中不管复制代理运行在哪个服务器上都没有关系，因为它们是同一台服务器，但是根据服务器的负载，用户可能会做出不同的选择。

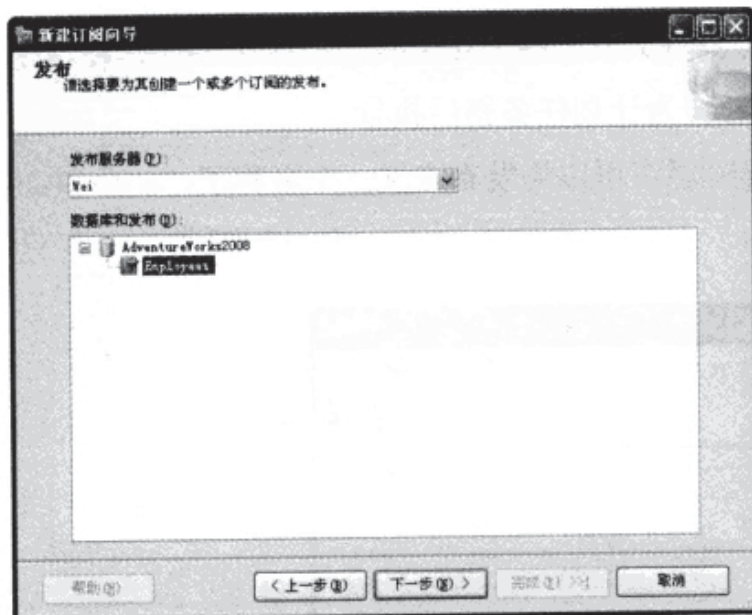


图 17-30

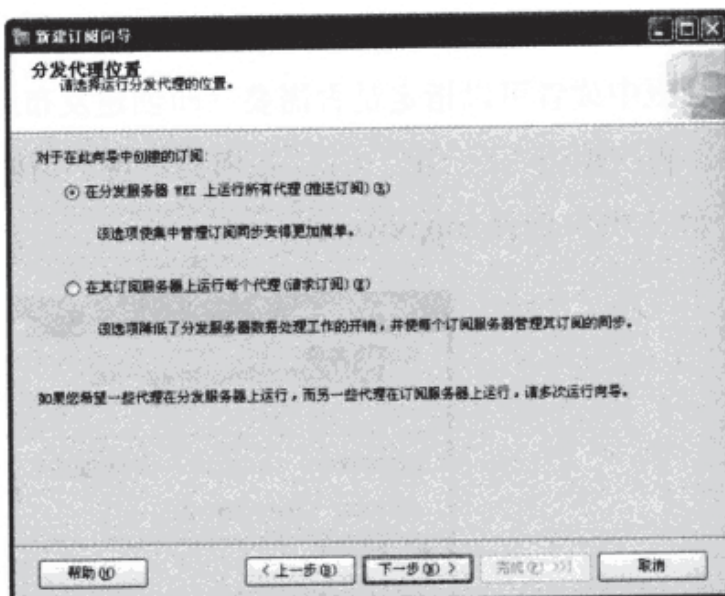


图 17-31

单击“下一步”，接着会出现“订阅服务器”对话框，如图 17-32 所示。本例中已经选择了 AWSubscriber 数据库，但是需要注意的是如何选择“添加 SQL Server 订阅服务器”来一次性配置多个订阅服务器。

接着将会出现“分发代理安全性”对话框。在这个对话框中可以定义分发服务器和订阅服务器所运行的安全环境(在本例中是同一个系统，但是可以很容易地配置成远程系统)。本例选择了模拟 SQL Server 代理安全环境，如图 17-33 所示。但是，在一个生产服务器上，出于安全性的原因，用户通常可能想要为复制代理指定一个更特定的安全环境。

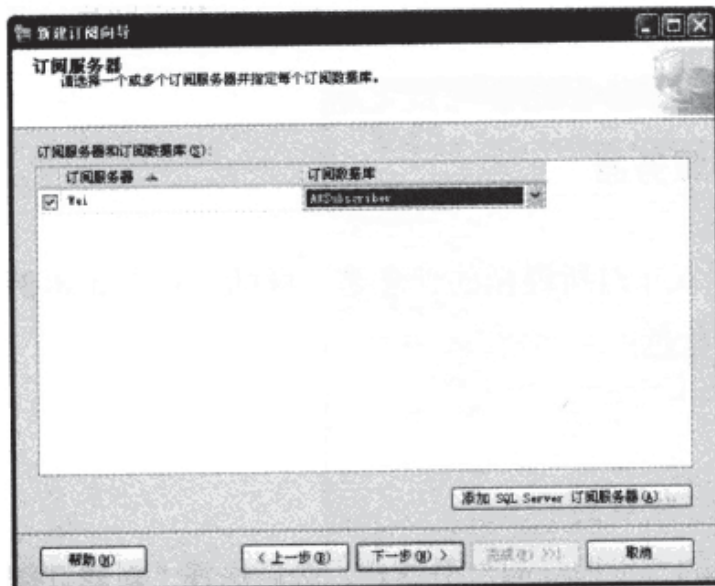


图 17-32

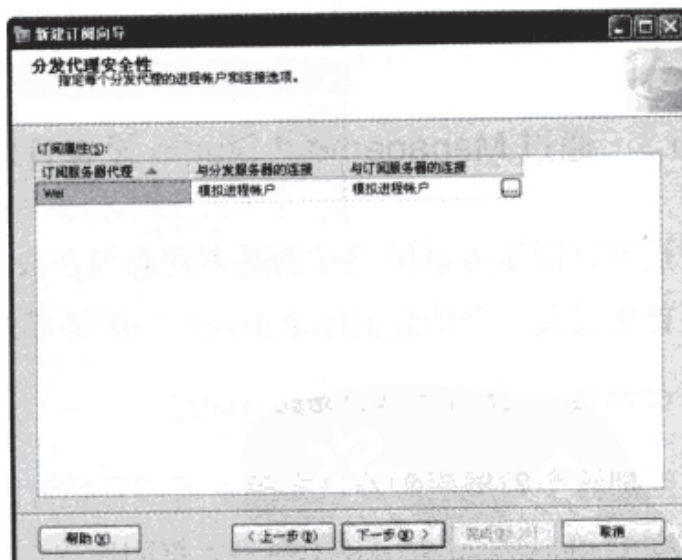


图 17-33

接着可以快速略过剩余的对话框了，只需要把代理设置成“持续运行”，并且把默认的“在发布服务器上提交”配置设置成“同时提交更改”就可以了。接着将会出现“用于可更新订阅的登录名”对话框，如图 17-34 所示。

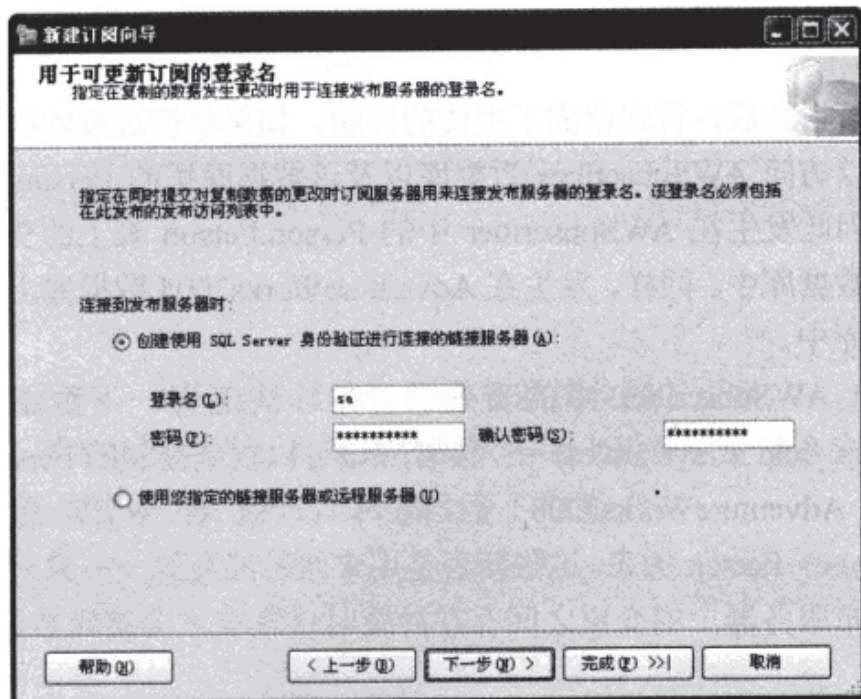


图 17-34

由于这些动作(分发和订阅)都是发生在同一台服务器上的,因此这台服务器是一台链接服务器(一台服务器作为链接服务器总是对自己可用)。当使用一台远程分发服务器时,既可以使用一个普通的 SQL Server 登录帐户,也可以像这里一样使用一台链接服务器(尽管在后一种情况中需要单独配置链接服务器)。

提示:

一台链接服务器是另一个 SQL Server 或 ODBC 数据源,它在服务器上建有一个别名。当用户通过名称引用一台链接服务器时,实际上是在获取一个指向到该链接服务器的连接信息的引用。

图 17-35 中的对话框允许用户选择何时初始化订阅(本例中选择了默认的立即初始化)。初始化过程包括从分发服务器上获取快照并应用该快照。接着将会以该快照为基础进行同步以应用变更。

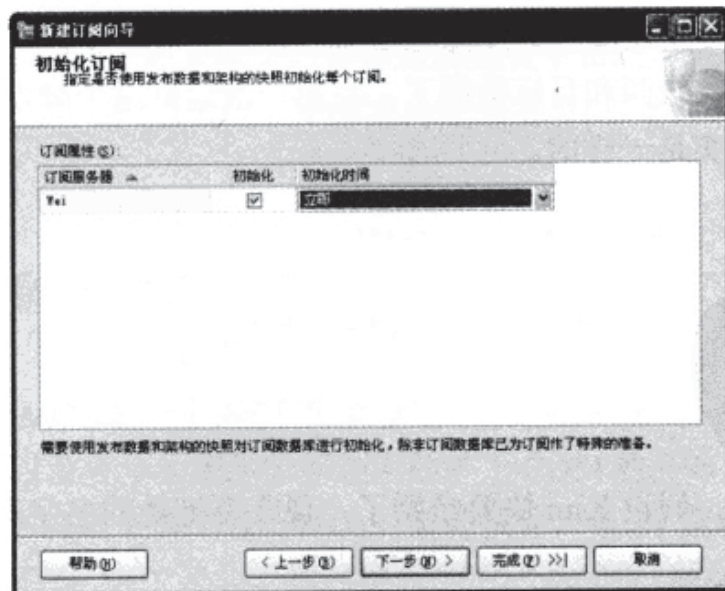


图 17-35

单击“下一步”后会出现完成对话框,这个对话框与之前例子中见过的结束对话框是一样的(何时运行任务以及一个摘要页面),接着单击“完成”。

17.5.4 使用复制数据库

一旦复制数据库可用之后，管理就成了主要的问题。如果事情进展的很顺利。那么需要做的事情就很少。用户可以访问 AWSSubscriber 数据库以及该数据库里的 Person.Person 表。由于配置了更新订阅服务器，因此发生在 AWSSubscriber 中的 Person.Person 表上的变更会被立即反映到源 AdventureWorks2008 数据库中。同样，发生在 AdventureWorks2008 数据库上的变更也会被反映到订阅服务器上的数据库中。

读者可以从查看 AWSSubscriber 中的表列表开始，快速看一下数据库中的表列表(使用 Management Studio、sp_help 或 sys.tables)——读者应该可以找到复制的 Person.Person 表。接着继续向前看一下 AdventureWorks2008 数据库。读者应该会找到一张叫做 Person.conflict_Employees_Person 的表。这张新表是用来跟踪冲突的——只有当发生在订阅服务器上的变更与发生在发布服务器上的变更之间存在冲突时这张表才会接收数据。

注意：

当发生冲突时，默认的发布代理将会在发布服务器的数据与客户的数据之间优先选择发布服务器的数据。用户可以修改这个默认的设置使它优先选择最近的变更或者根据其他现有的标准做出选择。用户还可以编写自定义的解决算法以使用与众不同的规则来解决冲突。

现在对数据进行一些修改以测试一下基于事务的复制。我们首先查看要修改的行的起始值：

```
SELECT aw.FirstName AS PubFirst,
       aw.LastName AS PubLast,
       aws.FirstName AS SubFirst,
       aws.LastName AS SubLast
FROM AdventureWorks2008.Person.Person aw
JOIN AWSSubscriber.Person.Person aws
  ON aw.BusinessEntityID = aws.BusinessEntityID
WHERE aw.BusinessEntityID = 38;
```

这里对数据库进行了交叉连接，这样就可以同时看到发布服务器和订阅服务器了。采用这种方式就可以在一个查询中比较源和目标数据了。在第一次运行这个脚本时(在做出任何变更之前)可以看到起始值，它们确实是一样的。

```
PubFirst  PubLast  SubFirst  SubLast
-----
Kim       Abercrombie Kim       Abercrombie

(1 row(s) affected)
```

好，现在做一些变更。假设 Kim 快要结婚了，现在要把她的名字改为 Abercrombie-Smith。

```
USE AdventureWorks2008;

UPDATE Person.Person
SET LastName = 'Abercrombie-Smith'
WHERE BusinessEntityID = 38;
```

现在再次运行原先的 SELECT 语句来检查一下结果:

PubFirst	PubLast	SubFirst	SubLast
Kim	Abercrombie-Smith	Kim	Abercrombie-Smith

(1 row(s) affected)

正如读者所看到的那样,发布服务器和订阅服务器接收到了更新。

现在稍微修改一下脚本,然后在订阅服务器的数据库上运行这个脚本来看看在发布服务器上发生了什么。这次把 Kim 的名字改回来(她可能改变了主意...):

```
USE AWSubscriber;
```

```
UPDATE Person.Person
```

```
SET LastName = 'Abercrombie'
```

```
WHERE BusinessEntityID = 38;
```

现在已经可以再次运行原先的 select 语句了:

PubFirst	PubLast	SubFirst	SubLast
Kim	Abercrombie	Kim	Abercrombie

(1 row(s) affected)

同样,在两个数据库中都看到了所做的变更。

注意:

更改在两个方向上都可以被看到,并且是立即被复制的,因为这里使用了带立即更新订阅服务器的事务复制。其他可供选择的复制方法将会导致在变更中出现延迟或者可能在没有人工干预的情况下根本就不复制变更。请确保复习一下所有的复制类型(本章前面所讨论的)以理解每种复制类型的行为。

17.6 复制管理对象(RMO)

复制管理对象,或者 RMO,是一个.NET 对象模型,它是在 SQL Server 2005 中第一次引入的,旨在替代 SQL Server 2000 以及更早的版本中所使用的基于 COM 的分布式管理对象(DMO)对象模型。读者可以把 RMO 看成 SQL 管理对象(SMO)的某种伙伴,第 23 章将广泛讨论 SMO。

RMO 让用户能够使用任何.NET 语言,通过编程的方式访问复制创建和配置中的任何部分。使用 RMO 的例子是自动化某些操作,如:

- **创建和配置一个发布:** 用户可以使用 ReplicationDatabase 和 TransPublication 或者 MergePublication 对象来定义发布。

- 添加和删除项目: TransArticle 对象支持在发布中添加和删除项目。此外, 用户还可以添加列筛选器或添加一个 FilterClause 属性来限制被复制的行。
- 重新发布快照。

这些例子仅仅是每天都可能要做的事情。此外, RMO 还能够创建、修改或删除复制过程的任意部分。

可以在 Visual Studio 中使用 RMO, 只要添加一个 Microsoft.SqlServer.Replication .NET Programming Interface 库的引用就可以了。接着用户可以指定 include、import 或者 using 指令引用 Microsoft.SqlServer.RMO。与所有支持 SQL Server 的管理库一样, 用户还需要引用 Microsoft.SqlServer.ConnectionInfo 库。

一个使用 RMO 的样例程序可以从 Wrox 网站(wrox.com)或者 professionalsql.com 上下载, 它创建与本章前面使用 GUI 所创建的发布相同的发布。

17.7 小结

虽然本章介绍了很多内容, 但是这仅仅是对复制做了一次介绍。本章罗列了很多架构师需要考虑的问题, 但是复制的领域是很广的, 其内容足够用一本书来讲述。实际上, 要为复杂场景构建恰当的模型需要考虑很多问题。好消息是, 如果读者已经真正掌握了本章的内容, 那么大概就已经能够处理所遇到的问题中 90% 的问题了。时间和艰苦的磨练会教给读者其余的部分。

如果读者已经从本章学到了什么, 那么希望你能够理解复制能够解决的一些常规问题的, 以及在根据拓扑规划和应用程序的一般架构(确保它考虑了复制的特殊需求)来制定计划时能够知道复制如何工作得最好。

下一章将介绍 SQL Server 的另一个“扩展”领域——全文索引。

第18章

全文搜索

在 SQL Server 2008 中,全文搜索是在架构上有较大改动的功能之一。虽然其核心用途和功能并没有发生大的变化,但是在这个发行版中全文功能已经被集成进了 SQL Server 的内核。如果读者认为自己已经很熟悉全文搜索并且准备跳过本章,那么笔者建议你至少浏览一下架构上的变化并思考一下它们在某些功能上有何不同,如备份、恢复以及扩展查询结果支持。

如果使用普通的 T-SQL(没有全文功能),那么对文本信息查询的处理手段是有限的。实际上,只有几个方法可用:

- 使用 LIKE 子句。这种方法的效率通常是非常低的。除非搜索模式以一个显式值开头,否则就无法使用任何类型的索引结构。如果搜索以一个通配符(如%或_)开头,那么 SQL Server 将无法知道从哪个索引的哪个位置开始——任何索引都变得没有价值了。
- 使用其他形式的模式匹配,如 PATINDEX 或 CHARINDEX。通常,这些方法更加低效,但是它们可以做到 LIKE 无法做到的事情。

有了全文搜索之后就可以为文本内容创建索引了——本质上是保存一个单词列表,使用户知道能够找到哪些单词以及这些单词在哪一行。此外,搜索将不再局限于模式匹配算法,还可以搜索单词的多种形式。例如,用户可能会使用单词 university,但是 SQL Server 仍然可以找到单词 universities,或者甚至更好,当要查找的单词为 drink 时,SQL Server 能够找到像 drunk 这样的单词。搜索的精确性是由用户决定的,但是即使要查找的单词位于一大块文本深处,SQL Server 仍然能够快速找出包含这个单词的行。

全文搜索(或者 FTS)支持任意类型的文档,但是这个文档类型需要向系统注册一个筛选器,并且该筛选器需要支持 iFilter 接口。这意味着用户可以把 Word、Excel、Acrobat 和其他受支持的文件存储到图形数据类型中,但仍然可以在这些数据上进行全文搜索。实际上,用户可以根据需要编写自己的扩展来支持其他文档类型。

提示:

笔者个人认为最后一项功能是相当棒的。实现 iFilter 接口允许用户把文本信息和格式信息分离开来,因此用户可以编写自定义的 iFilter 来把 XML 文件中 XML 标记字符串化以允许在自定义的 XML 文档类型上进行全文搜索。

本章将介绍这些全文搜索功能以及其他一些内容。

通过本章读者将可以了解到以下内容:

- 全文搜索体系结构

- 设置全文索引和目录
- 全文查询语法
- 全文杂项(quirk)
- 干扰词(Noise word)

此外,本章还会介绍如何使用目前的两种方法来完成大多数的全文搜索相关的操作。在介绍完这部分内容之后,读者应该能够应付 FTS 的挑战了,但是读者也需要准备好使用那些奇妙的功能。

18.1 全文搜索的体系结构

这个发行版对 FTS 的体系结构进行了重大改动。虽然一些基本的概念(如单词分解器、筛选器和索引)仍然是适用的,但是这些项目的使用方式已经发生了一些变化。图 18-1 给出的新的体系结构图(相当复杂)。

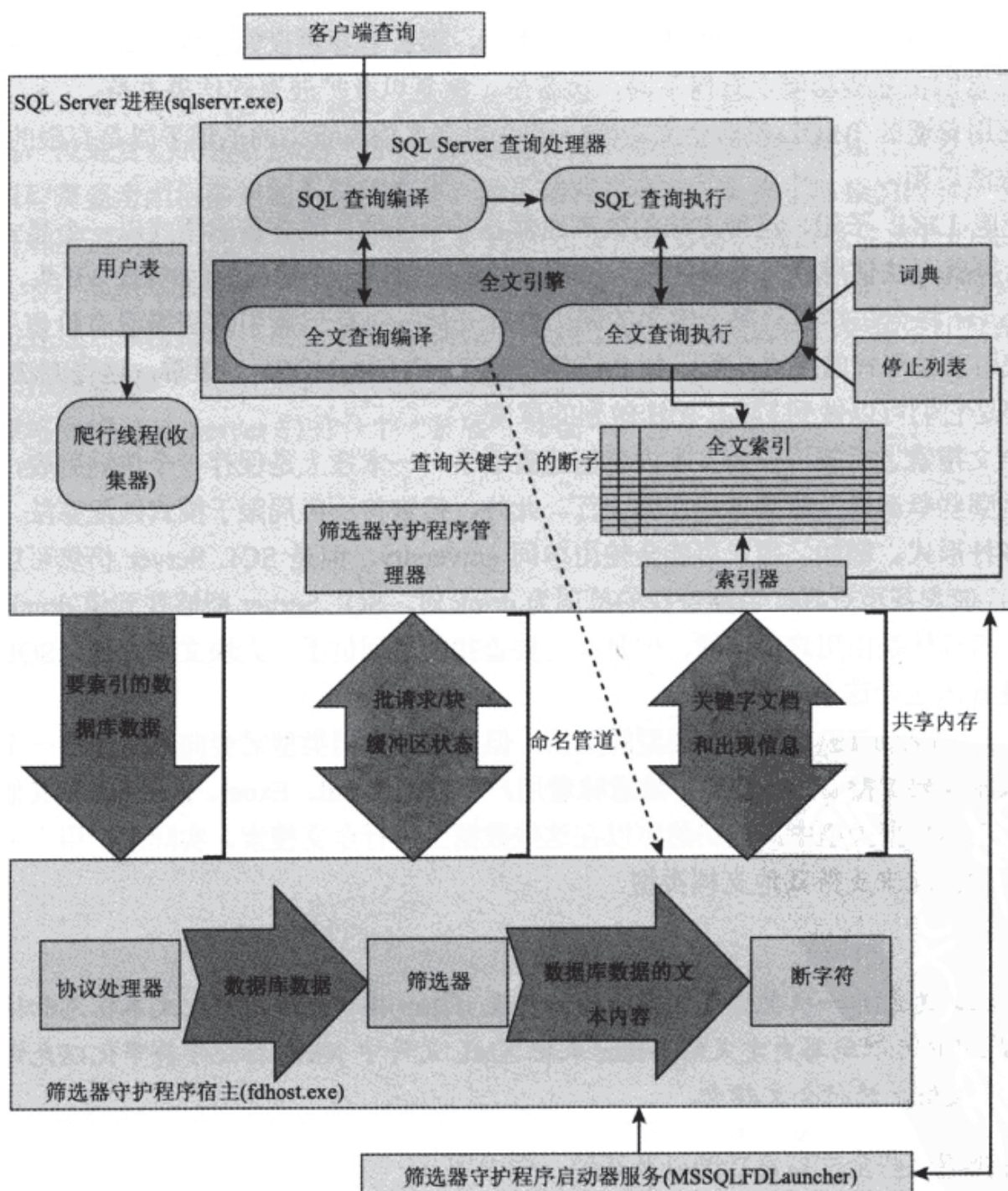


图 18-1

在 SQL Server 之前的版本中,全文搜索的核心实际上并不是 SQL Server 的一部分。它是一项共享技术,最初来自 Microsoft Index Server。用户将会看到有一个独立的进程与 SQL Server 安装在一起,表示该进程的服务名为 MSFTESQL。在 SQL Server 2008 中,全文现在已经是主 SQL Server 进程的一个基本部分了。全文引擎非常擅长检查原始文本数据和汇聚单词列表。它会维持单个单词和短语与 FTS 找到它们的位置之间的关联关系。

注意:

全文现在是 SQL Server 核心进程的一部分了。但是出于安全的原因,各个筛选器仍然由它们自己的进程进行实例化。

要在任意 SQL Server 表上进行全文查询,用户必须要为该表构建一个全文索引。全文索引的构建和维护——或填充索引——是通过 SQL Server 的一个进程来完成的,该进程会实例化一个筛选器后台实例,然后把文本流传给这个实例,接着该实例将会把流中的单词记录在目录中,并在目录项和包含该单词的行之间建立一个关联关系。

在默认情况下表根本就没有全文功能。一张表以及表中包含文本数据类型的数据的事实并不能保证在这张表上有一个全文索引。如果想要一个全文索引,那么需要创建该索引。即使创建了一个全文索引,这个索引中也没有任何内容。要使一个索引完全起作用就需要填充这个索引。

填充进程将检查索引指定的列并构建将会用到的单词列表。与 SQL Server 中的标准索引类似,只有那些被指定为需要包含进索引的列才会成为索引的一部分。但是与 SQL Server 中的普通索引不同的是一张表上只允许有一个全文索引——因此参与全文查询的所有列都需要成为索引的一部分。

差异还不止这些,实际上存在很多不同之处。主要的差异包括:

- **内部结构:** 通常,SQL Server 的索引是以平衡树的结构存储的。但是,全文索引使用的是一个基于令牌的结构,该结构是一个反转结构(本质上是向后存储内容)并且会进行压缩。
- **创建方法:** 在 T-SQL、SQL Management Objects(SMO)或 Windows Management Instrumentation (用户可以使用 Management Studio,但是它也是使用 SMO 的)中是使用 CREATE INDEX 命令来创建 SQL Server 索引的。而全文索引则是通过使用特殊的系统存储过程或使用 CREATE FULLTEXT INDEX 命令创建的。
- **更新方法:** 当底层的 SQL Server 数据发生正常变化时,SQL Server 索引会自动被更新。而全文索引则可以按需进行填充或通过一种带按需清理的“变更跟踪”机制来更新。

这里快速介绍了一下全文索引的体系结构。随着对本章剩余内容的学习,全文索引同 SQL Server 中以“常规”方式所实现的索引之间的差异所带来的影响将会越来越清晰。

18.2 设置全文索引和目录

正如在前面一节中看到的那样,SQL Server 数据库中的每张表都只能有零个或一个全文索引。在 SQL Server 2008 中,这些全文索引与数据库的其余部分是存储在一起的(想要把全文项目存放在一个独立的存储区域中则可以通过指定一个特定的文件组存放全文索引来实现)。一个目录可以存放多个全文索引。但是这些索引必须来自同一个数据库,用户还可以把来自同一个数据库的索

引存放在多个目录中，这样就可以使用不同的计划来管理这些索引的填充，或者把它们存储在不同的文件组中。

18.2.1 为数据库启用全文功能

在 SQL Server 2008 之前的版本中存在为数据库“启用”全文功能的概念。但是在 SQL Server 2008 中，所有数据库上的全文功能总是处于启用状态。

18.2.2 创建、修改、删除和操作全文目录

全文索引的 CREATE 语法与其他的 CREATE 语法十分相似，但是还是存在一些细微的差异：

```
CREATE FULLTEXT CATALOG <catalog name>
    [ON FILEGROUP <filegroup> ]
    [IN PATH <'root path'>]
    [WITH ACCENT_SENSITIVITY = {ON|OFF}]
    [AS DEFAULT]
    [AUTHORIZATION <owner name> ]
```

其中大多数条目都是不言自明的，但是表 18-1 还是简单介绍一下：

表 18-1

ON FILEGROUP	这个条目存在的原因仅仅是为了与 SQL Server 2005 保持向后兼容(在 SQL Server 2000 中不存在 CREATE FULLTEXT CATALOG 这个命令)。在 SQL Server 2008 中这个条目不起作用
IN PATH	同样，这个条目也是为了向后兼容而存在的。在之前的发行版中，真正的全文目录不是创建在数据库中的，而是作为一个独立的文件存放在磁盘上的。这个选项告诉 SQL Server 在哪个路径上创建该文件。在 SQL Server 2008 中，这个选项不起作用
WITH ACCENT_SENSITIVITY	这个选项的名称很好地说明了它的作用。它决定搜索是否考虑重音(如“e”是否与“é”相同)。记住如果在创建目录之后改变这个设置，那么整个目录都需要重新填充。全文目录将会根据数据库的设置来使用各种重音敏感性
AS DEFAULT	这是另一个顾名思义的选项，它将正在创建的全文目录设置为所创建的任何全文索引的默认目录
AUTHORIZATION	这个选项稍微有点复杂。正如读者可能想到的那样，这个选项是关于安全和权限的。它将全文目录的所有者改为指定用户或角色，而不再使用默认的配置(默认为真正创建该目录的用户)。SQL Server 把所有者变为模式之后该选项就变得有点混乱了。所有者在很大程度上体现为模式，不过该特殊设置的本质则更加接近早前版本中所有者的概念。这里需要意识到的一个关键概念是角色可以是全文目录的拥有者——不仅仅是用户。如果用户把所有者更改为一个特定的角色，那么创建该全文目录的用户在创建该目录时必须要是该角色的一个成员

现在为 AdventureWorks2008 创建全文目录。只要调用：

```
USE AdventureWorks2008;  
  
CREATE FULLTEXT CATALOG MainCatalog;
```

提示：

这是众多执行后不会得到很多反馈的命令之一，只要读者没有看到错误，那么目录应该已经创建成功了。

就是这么快，现在 AdventureWorks2008 已经有了一个可用的全文目录了。这里没有把这个全文目录指定为默认目录，因此任何想要使用这个目录的全文索引都需要显式地把该目录声明为它的目标。

1. 修改全文目录

修改全文目录的方式与创建它们的方式十分相似，但是可以修改的部分是有限的。其语法如下：

```
ALTER FULLTEXT CATALOG <catalog name>  
{ REBUILD [WITH ACCENT_SENSITIVITY = {ON|OFF} ]  
  | REORGANIZE  
  | AS DEFAULT  
}
```

在这个 ALTER 语句中有三个顶级选项可以设置。下面简要介绍一下这些选项。

REBUILD

顾名思义，这个选项会完全重新构建全文目录。在默认情况下将以与该目录之前被创建时所具有的设置完全一样的设置(所有者以及是否为默认目录)来创建它。

提示：

记住在进行重新构建的过程中，全文目录以及该目录中包含的所有索引都将会变成脱机。

除了简单的重新构建之外，用户通常做的就是压缩文件(删除行等)，用户还可以通过重新构建来修改重音敏感性。如果想要重新设置重音敏感性，那么在发出 REBUILD 命令时只需要指定是否将该选项关闭还是开启就可以了。

任何重新构建都包含了重新填充该目录下的所有索引。

REORGANIZE

这个选项与 REBUILD 类似，也有其优缺点。

REORGANIZE 将会清理目录，但是它是在联机的状态下进行的。其结果更像是进行重新安排，而不是将事物移走后再重头开始。它看起来相当不错，但是可能没有从头开始那样好。

用户可以把 REORGANIZE 看成一个碎片整理的过程。它把目录内部的几个不同的索引结构合并起来(出于在对全文进行分析时的性能方面的原因，一些项目可能会被留在它们自己的索引结构中，而不会被合并到目录的主索引中)。这个命令试图对目录进行矫正。与 REBUILD 不同，REORGANIZE 还会为全文目录重新组织内部结构(存储元数据的结构)。

AS DEFAULT

这个选项的工作方式与它在 CREATE 中的工作方式是一样的。它把这个特定的目录设置为默认的全文目录，为该数据库所创建的新的全文索引将会被存放到这个目录中。

2. 删除全文目录

读者终于看到这个操作了——毕竟，它与我们经常使用的核心 DROP 语法是一样的：

```
DROP FULLTEXT CATALOG <catalog name>
```

当然，这个目录会被删除。

18.2.3 创建、修改、删除和操作全文索引

好，现在已经有有了一个全文目录，它在很大程度上仅仅是一个容器。全文目录本身什么也没有——把它想象成一个没有气的气罐。我们所需要的是真正的全文索引。全文目录是存储全文索引的地方，而索引本身会提供真正的引用信息以允许全文查询快速和高效地运行。

1. 创建全文索引

在创建全文索引时，命令的核心项目与常规的索引创建命令没有根本的不同之处，与常规索引拥有属性一样(如是否为聚集索引或非聚集索引)，全文索引也有自己的属性。

创建全文索引的语法如下所示：

```
CREATE FULLTEXT INDEX ON <table name>
    [( <column name> [TYPE COLUMN <type column name> ]
      [LANGUAGE <language term>] [,...n])]
KEY INDEX <index name>
    [ON <fulltext catalog name> ]
[WITH
    { CHANGE_TRACKING [=] { MANUAL | AUTO | OFF }
    [, NO POPULATION] }
] | [STOPLIST [=] {OFF | SYSTEM | <stop list name>}]
```

注意，这里的选项有点不够典型。在大多数情况下，必需的项目首先会被列出，但是该语法的古怪之处是在必需的参数(关键索引)之前需要给出可选的参数(一个列表)。下面先给出一个简单的例子，然后介绍其中各个部分：

```
CREATE FULLTEXT INDEX ON Production.ProductModel
    ( Name LANGUAGE English)
KEY INDEX PK_ProductModel_ProductModelID
    ON MainCatalog
WITH CHANGE_TRACKING OFF, NO POPULATION;
```

这里为 Production.ProductModel 表创建了一个全文索引。这里已经把列中所使用的语言显式地指定为 U.S.English 了。如果需要，可以添加一个逗号，后跟另外一个列名以及一个可选的 TYPE COLUMN 或另一个 LANGUAGE 标识符。本例中在语言选项后面特意指定了存储该索引的全文目录并且指定关闭变更跟踪，同时不对索引进行初始化填充。

这里需要考虑的问题很多,因此下面稍微深入介绍一下各个选项。

注意:

本例没有为全文索引设置一个名称。由于任意给定的表最多只能有一个全文索引,因此没有必要命名这个索引(本质上是通过它构建时所依赖的那张表来识别的)。确保已经包含了所有希望在其之上进行全文搜索的列。

列列表

这个选项可能是所有选项中最富有技巧性的一个了。尽管前面的语法中写“column name”,但是实际上使用的是列列表。这里的问题是在能够移到下一列之前,对于列出的每个列都需要包含所有与该列有关的内容。也就是说用户需要在命名下一列之前对当前列进行 TYPE COLUMN 和 LANGUAGE 参数的设置(如果需要的话)。

例如,如果还希望包含目录描述,那么可以通过在第一列定义之后加上这个描述来实现:

```
CREATE FULLTEXT INDEX ON Production.ProductModel
```

```
( Name LANGUAGE English,  
  CatalogDescription)
```

```
KEY INDEX PK_ProductModel_ProductModelID  
ON MainCatalog  
WITH CHANGE_TRACKING OFF, NO POPULATION;
```

提示:

这个例子仅供参考。它将会运行出错,因为前面已经在 Production.ProductModel 表上创建了一个全文索引。

LANGUAGE

这个选项指定了刚才识别的列所采用的语言。这对干扰词(那些经常出现,但是对搜索没什么价值的单词,请参见本章后面的介绍)判断以及排序之类的操作是非常重要的。任何 SQL Server 支持的区域化语言都是有效的(在本书撰写时有 33 个区域)。要获得可用的区域别名列表,可以查询 master 数据库中的 sys.syslanguages 元数据视图:

```
SELECT name, alias FROM master.sys.syslanguages;
```

TYPE COLUMN

当用户需要在类型为 image 或 varbinary 的列上创建全文索引时需要使用这个选项。在 AdventureWorks2008 数据库上建立的全文索引就使用了这个选项(在 Production.Documents 表上)。本章后面将会介绍这个例子。现在假设需要使用 SQL Server 进行文档管理(对 SQL Server 来说,这根本就是很常见的用法)。如果要存储的文档是用一个或多个应用程序混合编写的,如 Microsoft Word(.doc)、Acrobat(.PDF)、Excel(.XLS)或者文本编辑器(.TXT),然后全文搜索需要知道它所分析的每一行中存储的是什么类型的文档,从而也就知道该使用哪个分析插件了。

在这种情况下,用户需要向表中添加另一列(除 image 或 varbinary 列之外),其中包含存储在二进制列中的文档的扩展名(.DOC、.PDF 等等)。这个列将会成为 CREATE FULLTEXT INDEX 命

令中 TYPE COLUMN 属性的参数值。

KEY INDEX

与 CREATE FULLTEXT INDEX 命令中的其他选项不同，这个选项是必需的。

所有具有全文索引的表必须要有一列能够唯一地标识每一行。这可以是一个主键或一个 unique 约束。这里需要记住的是需要提供与 unique 标识符相关联的索引的名称，而不是列或约束的名称。由于在全文索引中将会重复使用该列来关联数据，因此笔者建议读者使用最小的可用主键或 unique 索引。

ON

这是存储索引的全文目录的名称。如果数据库有一个默认的全文目录，那么这个选项是可选的，但是如果没有建立默认的目录，那么必须要指定该选项。

WITH

这个选项提供了一组指令来指示如何使用数据填充索引以及当索引创建完毕之后如何应对表上发生的变更。

CHANGE_TRACKING

变更跟踪所涉及的是全文索引如何应对发生在底层表上的变更。

这里存在的一个难题是如何在全文搜索的准确性与保持一个负载较重的系统处于最新状态所需的开销之间进行权衡(对比于维护标准 B 树索引)。

变更跟踪对变更提供了三种级别的支持(见表 18-2):

表 18-2

OFF	<p>只有在对索引进行完全填充时才会更新全文索引。本质上，用户每次进行填充时都需要从头开始重新构建。这意味着持续的维护开销将不复存在，但这同时也意味着表中某些行将不会在全文查询中被返回，或者更糟的是，某些行可能会因为包含了用户感兴趣的单词而被返回，但是由于这些行发生了变更，从而导致它们不再包含这些单词了。当数据移动比较缓慢(不经常发生变更)并且/或者不需要非常精确的结果时，这个选项是很好用的。作为放弃数据精确性的回报，这意味着用户不需要持续地维护索引，同时该索引也总是尽可能地保持紧凑，因为它们不存在碎片的问题。但是，这也意味着在进行重新填充时需要有一段宕机时间，从而导致整个过程所需的时间变长了</p>
AUTO	<p>在这种模式下，SQL Server 会持续为发生在表上的变更更新索引。虽然在变更的发生与变更被反映到全文索引中之间存在一些延迟，但是这个延迟已经被最小化了，索引的更新已经接近实时了。当数据移动非常快或者对结果的精确性要求非常高时可以使用这个选项。由于 SQL Server 会使用很小的中间结构来跟踪变更，因此需要承受开销较高的问题。随着时间的流逝，这些过程可能会变得非常低效，并且可能会影响查询的性能。不过在短期看来，这不是一个大问题。如果使用了这个选项，那么仍然需要考虑定期对索引进行重新组织或完全重新填充</p>
MANUAL	<p>这是一种折衷的方式。它确实进行了跟踪以标识出变更，但是它不会更新全文索引，直到有人显式地告诉它去更新全文索引。因此用户可以手动执行更新来把发生的变更应用到现有的索引上，而不用进行完全重新填充了</p>

NO POPULATION

只有当把变更跟踪选择设置为 OFF 时才能使用这个选项。

在默认情况下, 当创建全文索引时, SQL Server 会启动一个后台进程来填充这个索引。如果把变更跟踪关闭并且指定了 NO POPULATION 选项, 那么将只会定义这个全文索引, 而不会真正的往里面填充任何数据以使得它可用。用户然后可以安排自己的索引填充作业在后面某个时间运行(大概在一天中负载较轻的时间段)。

STOPLIST

停止列表替代了之前版本中的干扰词列表。干扰词现在叫做停止词(stop word)。它们是那些被显式地排除在索引之外的单词。一般来讲, 这些单词相当常见(在英语中, 这些单词可能包括“the”、“and”、“or”以及其他一些出现频率异常高的单词), 但是很少为内容检索增加什么实际价值。在之前的发行版中, 干扰词是存放在一个单独的文件中的, 但是 SQL Server 2008 把停止词存储在了停止列表中。用户能够为全文检索定义的每种语言都有一个相关联的系统停止列表, 但是用户还可以创建自定义的停止列表。如果想要包含所有的单词, 那么用户还可以关闭停止列表的使用。

2. 修改全文索引

好, 现在已经有有了一个索引, 但是需要修改这个索引。正如读者可能所期待的那样, 新的全文索引语法会支持 ALTER 语句。它的形式如下:

```
ALTER FULLTEXT INDEX ON <table name>
{
    ENABLE
    | DISABLE
    | SET CHANGE_TRACKING { MANUAL | AUTO | OFF }
    | ADD (<column name>
    [TYPE COLUMN <type column name> ]
    [LANGUAGE <language alias>] [,...n] )
    | DROP (<column name> [,...n] )
    | START { FULL | INCREMENTAL | UPDATE } POPULATION
    | {STOP | PAUSE | RESUME} POPULATION
    | SET STOPLIST { OFF | SYSTEM | <stoplist name> }
    [WITH NO POPULATION]
}
```

提示:

这个 ALTER 语句与之前遇到的 ALTER 语句有本质上的区别。看到这里的 START 和 STOP 动词了吗? 这个 ALTER 语句不仅可以改变全文索引的定义, 而且可以对索引进行一定的管理。记住这些差别, 因为与 SQL Server 中其他的 ALTER 语句相比, 它们之间的差别不是特别直观。

这个语句中的一些元素的工作方式与它们在 CREATE 语句中的工作方式是完全一样的。这里只不过把一个选项从一个命令移到了另一个命令中。但是, 其中一些元素是全新的。这里从更传统的 ALTER 语句所包含的项目开始介绍, 然后介绍语句中那些面向管理的部分。

ENABLE/DISABLE

顾名思义, 如果禁用了全文索引, 那么索引就会被保留在那边, 其中所有的数据都不会发生

改变。变化之处在于全文查询已经不能使用索引了，同时索引的数据也不会进行更新(在发出 **DISABLE** 命令时正在进行的更新将会立即被停止)。

当设置 **ENABLE** 之后，这个索引会从它停止的地方重新开始(它可能需要赶工，但是已经存在的数据将不会发生变化，同时用户也不需要进行完全填充)。

ADD

该选项的工作方式与对列进行初始定义一样。例如，如果想要在 **Production.ProductModel** 表的全文索引上添加 **Instructions** 列，那么可以使用下面的代码：

```
ALTER FULLTEXT INDEX ON Production.ProductModel  
ADD ( Instructions )
```

LANGUAGE 和 **TYPE COLUMN** 属性的工作方式与之前它们在 **CREATE** 语法中的工作方式一样。

DROP

同样，这个选项的工作方式正如读者所期待的那样。如果想要删除刚才添加的 **Instructions** 列，那么可以使用下面的代码：

```
ALTER FULLTEXT INDEX ON Production.ProductModel  
DROP ( Instructions )
```

START...POPULATION

START 提供了三个选项来指定使用哪种类型的填充。

FULL

好用并且简单的选项——把它想象成“重新开始！”命令。每一行都会被重新检查，同时会从头开始重建索引。

INCREMENTAL

只有当表中包含一个类型为 **timestamp** 的列时这个选项才会有效(否则会默认回到 **FULL** 填充)，它将会把从上次对表的索引进行填充之后发生在行上变更填充到索引中。可以把这种填充方式想象成“请赶上你的工作！”。增量填充不要求启用变更跟踪。

UPDATE

当关闭了索引的 **AUTO** 填充，但又想要所有的更新、插入或删除操作能够反映到索引中时该选项就派上用场了。它不要求启用变更跟踪。

STOP、PAUSE 和 RESUME

它们将对全文索引上正在进行的任何填充操作执行特定的动作。**STOP** 选项不会停止自动变更跟踪——只会停止完全或增量更新。**PAUSE** 和 **RESUME** 就像读者预期的那样运作。

3. 删除全文索引

笔者确信读者自己能够推测出相应的语法了，但是出于完整性的考虑，下面还是介绍一下其

语法:

```
DROP FULLTEXT INDEX ON <table name>
```

因此, 如果运行下面的命令(不要运行这个命令, 因为下一个例子中会使用这个索引):

```
DROP FULLTEXT INDEX ON Production.ProductModel
```

那么这个全文索引会被删除。

18.2.4 针对旧语法的说明

在 SQL Server 2005 之前, 用户使用名为 `sp_fulltext_catalog` 的特殊系统存储过程。通常, 还会使用其他系统存储过程来完成全文功能。

在最近的两个发行版中已经不推荐使用这些存储过程了, 同时下一个全文体系结构中将不会涉及它们了。这里将不会深入介绍这些存储过程, 但是希望读者能够了解它们以防在生产配置中碰到它们。如果真的碰到它们了, 那么笔者建议根据实际情况尽快将它们迁移到新的语法上(基本上不再需要支持 SQL Server 2000 了)。

18.3 更多有关索引填充的内容

与“常规的”SQL Server 索引不同(SQL Server 的性质以及索引的存储方式将很自然地会使索引保持最新), 全文索引运作在不同的存储结构上并且填充索引需要更多的开销。正因为如此, 在索引更新到最新以表示真正的数据之前需要对它进行一定程度的干预。

填充有三种方式(好吧, 其实更像是两种半)。下面分别介绍每种方式。

- **完整:** 顾名思义, 对于这种填充方式, SQL Server 基本上会忘记以前知道的任何数据并从头开始填充。每一行将会被重新扫描, 索引将会从头开始被重新构建。
- **增量:** 在这个选项中, SQL Server 将会使用一个类型为 `timestamp` 列来跟踪从上次填充以来哪些列发生了变更。在这个场景中, SQL Server 只需要记录发生在行上的变更。这个选项要求表中有一个 `timestamp` 列。任何不引起 `timestamp`(不存在记录日志的操作——通常是 BLOB 活动)发生变化的更新将无法被检测到, 除非同一行中的其他内容发生了变化。
- **变更跟踪:** 跟踪上次填充以来发生的变更。这个选项可以使全文索引几乎实时地保持更新, 但是记住填充全文索引需要花费很多的 CPU 时间和内存, 从而可能会影响服务器的性能。用户需要对立即更新和保留更新并在服务器不忙时一下子更新两种方案进行比较并做出适当的选择。

除非用户使用变更跟踪, 否则全文索引只会在用户指定开始时才会填充, 或者根据用户创建的填充计划任务来填充。

显然, 每当用户首次创建全文索引或者改变参与索引的列列表时都需要完全重新填充这个索引(发生在之前为空的索引上的增量变更意味着每一行都需要被重新扫描——对吧?)。现在 SQL Server 会自动完成这个任务, 除非用户显式地指定它不要这么做。用户可以在目录级别或表级别执行手动填充。通常会为新增加的或发生变化的索引进行表级别的重新填充, 而在进行例行维护时会执行目录级别上的重新填充。

有了这些概念之后，现在可以开始填充先前在 `Production.ProductModel` 表上创建的全文索引了。如果前面没有专门指定 `NO POPULATION` 选项，那么 SQL Server 会自动填充这个索引。但是由于前面已经指定它不要进行填充，因此现在需要下达填充的命令。由于这是首次填充，因此需要进行一次完整填充(坦白讲，进行增量填充的结果是一样的，因此使用哪种方式并没有什么关系，但是这里采用这种从逻辑上更讲得通一点)。使用新语法的代码如下所示：

```
ALTER FULLTEXT INDEX ON Production.ProductModel
  START FULL POPULATION;
```

注意：

全文索引的填充是作为一个后台进程运行的。正因为如此，读者的命令将会在填充作业启动后返回一个“成功完成”消息。但是这个消息并不意味着索引的填充过程已经结束，如果是在一张大型的表上创建索引的话，那么填充过程可能需要几个小时才能完成。

如果想要知道全文填充过程的状态，那么可以右击数据库的“存储”|“全文目录”节点下的全文索引的名称，然后检查一个叫做“填充状态”的属性。

由于这张表是相对较小的，因此读者不需要等待很长的时间就可以在索引上运行查询并获取结果了：

```
SELECT ProductModelID, Name
FROM Production.ProductModel
WHERE CONTAINS(Name, 'Frame');
```

这个查询将按顺序返回 10 行数据：

ProductModelID	Name
5	HL Mountain Frame
6	HL Road Frame
7	HL Touring Frame
8	LL Mountain Frame
9	LL Road Frame
10	LL Touring Frame
14	ML Mountain Frame
15	ML Mountain Frame-W
16	ML Road Frame
17	ML Road Frame-W (10 row(s) affected)

现在已经有了全文索引！是开始介绍刚才运行的那个查询能够做些什么以及还有哪些可用的选项的时候了。

18.4 全文查询语法

全文搜索有自己的查询语法。它添加了特殊的命令来扩展 T-SQL，同时这也表明支持查询的是全文引擎，而不是常规的 SQL Server 引擎。

幸运的是，基本的全文查询真的是非常基本。全文引擎只支持四种基本的语句，实际上它们被分为了两个相互重叠的类别，每个类别包含两个语句，如表 18-3 所示：

表 18-3

	精确或特定的术语	含 义
条件	CONTAINS	FREETEXT
分级别的表	CONTAINSTABLE	FREETEXTTABLE

两个条件谓词的工作方式与 EXISTS 运算符的工作方式非常相似。本质上，它们会对每一行给出一个 yes 或 no 的结果，判断该行是否满足提供的搜索条件。用户在查询的 WHERE 子句中会使用到它们。另一方面，两个分级别的查询则不会提供条件。相反，它们将以表格形式返回一个结果集，其中包含了所有找到匹配(这就是用户所关联的)的行的键值以及表明匹配程度的分级信息。

下面详细介绍这四个关键字。

18.4.1 CONTAINS

这个术语将会根据特定的单词或短语查找相匹配的内容。在默认情况下，它查找的是精确的匹配(也就是说 swim 必须要是 swim——而不是 swam)，不过也可以通过使用修饰符来查找所有的特定匹配(拥有相同字根的单词——如 swim 和 swam)。CONTAINS 能够识别特定的关键字。

现在，这里只会介绍简单形式的 CONTAINS。在介绍完这四个语句的基础知识后将会介绍它们的高级特性(由于它们共享某些修饰符，因此可以一次性介绍它们)。

基本的语法如下所示：

```
CONTAINS(({<column>|*} , '<search condition>')
```

用户可以指定检查特定的列，或者使用*，在这种情况下将会对所有具有索引的列进行比对。在最简单的形式中，搜索条件应该只包含一个单词或短语。

提示：

这里有两点值得提一下。第一点是需要记住，只会返回那些全文索引中包含的列的结果。对于在 ProductModel 表中最后创建的索引来说，这意味着搜索将只包含 Name 和 CatalogDescription 列。而其他像 Introduction 这样的列就不会被包含到搜索中，因为它们不在索引里(读者可能会记起在测试 ALTER 语法的时候删除了该列)。第二点是搜索条件可以比本例中所使用的简单条件复杂很多，在读者有了基础知识之后会介绍复杂的条件。

例如，现在回到之前用过的那个查询来证明本章的填充练习能够正常工作：

```
SELECT ProductModelID, Name
FROM Production.ProductModel
WHERE CONTAINS(Name, 'Frame');
```

这里的意思是希望找到那些在索引中的 Name 列中包含单词 Frame 的所有行，并返回这些行中的 ProductModelID 和 Name 列。

如果读者检查一下结果中的 Name 列，那么会发现所有行都是精确匹配的。

下面浏览一下另一个例子。这次运行的基本上的同样的查询，但是要查找的单词是 Sport：

```
SELECT ProductModelID, Name
```



```
FROM Production.ProductModel
```

```
WHERE CONTAINS (Name, 'Sport');
```

这次只返回了一行：

```
ProductModelID Name
```

```
-----
```

33	Sport-100
----	-----------

```
(1 row(s) affected)
```

同样，这次返回了所有 **Name** 列与单词 **Sport** 精确匹配的行。但是，如果读者看一下表中其他的行，将会发现其中存在单词 **Sport** 的其他变体(本例中是复数形式)，但是这些行没有被返回。

同样——**CONTAINS** 的默认行为是精确匹配。

18.4.2 FREETEXT

FREETEXT 与 **CONTAINS** 难以置信的相似。实际上，它们的语法几乎是一样的：

```
FREETEXT ({<column>|*} , '<search condition>') [;]
```

它们之间的唯一区别就是返回结果。读者可以看到，**FREETEXT** 对于结果的精确度要求更为宽松。与一个一个的精确的字母拼写相比，它对单词的意思更感兴趣。

为了快速说明问题，下面看一下前面一节中的 **Sport** 查询，但是稍微修改一下，用 **FREETEXT** 替代 **CONTAINS**：

```
SELECT ProductModelID, Name
FROM Production.ProductModel
```

```
WHERE FREETEXT (Name, 'Sport');
```

执行这个查询返回的结果与使用 **CONTAINS** 返回的结果稍微有点不同：

```
ProductModelID Name
```

```
-----
```

13	Men's Sports Shorts
33	Sport-100

```
(2 row(s) affected)
```

本例中的差别是由对单词复数形式的不同解释造成的——**FREETEXT** 查询返回了包含单词 **Sports** 的行——不只是返回那些包含单词 **Sport** 的行。**FREETEXT** 还能够处理像 **swim** 和 **swamp** 以及其他单词变体的情况。

18.4.3 CONTAINSTABLE

从找出匹配的行的角度来看，**CONTAINSTABLE** 与 **CONTAINS** 的工作方式是完全一样的。

它们之间的差别在于如何处理结果。

CONTAINSTABLE 的语法与之前的语法是类似的,但是需要指定它所操作的表以及使用一个可选的参数来限制返回的行的数量。

```
CONTAINSTABLE (<table>, {<column>|*}, '<contains search condition>' [, <top 'n'>])
```

CONTAINS 返回的是一个简单的布尔结果,适合用在 WHERE 子句中,而 CONTAINSTABLE 则会返回一张表——返回的结果中包含级别信息来说明搜索短语与返回的行的匹配程度。

下面通过运行原先的查询来说明问题,这里将使用 CONTAINSTABLE:

```
SELECT *
FROM CONTAINSTABLE(Production.ProductModel,Name, 'Sport');
```

这个查询将返回一行——与 CONTAINS 一样——但是返回值中提供的信息则稍微有些不同:

```
KEY          RANK
-----
33           128

(1 row(s) affected)
```

返回结果中有两列:

- **KEY:** 还记得前面讲过的全文索引必须要能够与被索引的表中的一个关键列关联起来吗?好,CONTAINSTABLE 返回的 KEY 列将会与那个键列关联起来。即在输出中 KEY 列的值将会匹配索引表中唯一的一行,该行是通过这个键来识别的。
- **RANK:** 一个从 0 到 1000 的值,表明返回的行在搜索结果中的匹配程度——值越高,匹配程度越高。

为了使用 CONTAINSTABLE,只需要把原先的表与 CONTAINSTABLE 返回的结果连接起来就可以了。例如:

```
SELECT Rank, ProductModelID, Name
FROM Production.ProductModel p
JOIN CONTAINSTABLE(Production.ProductModel,Name, 'Sport') ct
ON p.ProductModelID = ct.[KEY];
```

提示:

注意在 KEY 列名上使用了方括号。这里使用方括号的原因是 KEY 也是一个关键字。记住这里的命名规则,如果列或者表名使用了一个关键字(用户不应该这样做),那么需要把它放在方括号中。

这样将会返回原先的行,但是这次返回了底层表上的额外信息:

```
Rank          ProductModelID Name
-----
128           33           Sport-100

(1 row(s) affected)
```


在本例中, Rank 中的值是一样的, 但是如果有多个相异的值的话, 那么可以对它们采取下列措施:

- 根据任意的 Rank 值进行筛选。例如, 用户可能只想要根据分数返回那些最匹配的行。
- 根据等级进行排序(对等级进行排序——最有可能的是从最高排到最低)。

18.4.4 FREETEXTTABLE

就像 FREETEXT 是 CONTAINS 的近亲一样, FREETEXTTABLE 是 CONTAINSTABLE 的近亲。FREETEXTTABLE 只是简单地将 FREETEXT 对单词的不精确匹配同 CONTAINSTABLE 返回的表形式结合了起来。

现在可以把前面的某些的例子结合起来看看 FREETEXTTABLE 是如何改变事情的:

```
SELECT Rank, ProductModelID, Name
FROM Production.ProductModel p
JOIN FREETEXTTABLE(Production.ProductModel, Name, 'Sport') ct
ON p.ProductModelID = ct.[KEY];
```

上面的代码与原先的 FREETEXT 查询一样返回了两行, 但是其中包含了 CONTAINSTABLE 中所包含的排名信息:

Rank	ProductModelID	Name
102	13	Men's Sports Shorts
102	33	Sport-100 (2 row(s) affected)

提示:

对全文索引进行试验, 读者将会看到这些排名信息是如何帮助获取更好的结果的。

18.4.5 处理短语

各种全文关键字都能够处理短语。但是, 它们解析和处理短语的方式则稍微有些不同。

本节从最简单的例子开始——一个简单的包含两个单词的短语。这次所要查找的短语是 damaged seats。为了使事情更复杂一点, 这里将不再限制从哪个列中进行查询(只要该列是全文索引的一部分即可)。

```
SELECT DocumentNode, DocumentSummary, Document
FROM Production.Document
WHERE CONTAINS(*, '"damaged seats"');
```

注意短语被双引号括起来了。当需要把一组词当成一个单元来处理时都需要这么做。然而, 这里的查询只返回了一行结果。把该结果列出来则需要占用较大的篇幅(这是由 Document 和 DocumentSummary 列的规模决定的), 其相关部分如下:

DocumentNode	DocumentSummary	Document

```
0x7C20      Worn or damaged se . . . 0xD0CF11E0A1B11AE100000000000...
```

```
(1 row(s) affected)
```

主要把短语用双引号括起来, CONTAINS 就会对精确匹配该短语的行进行检查(搜索短语需要放在单引号中)。FREETEXT 也采用同样的工作方式。

18.4.6 布尔操作

SQL Server 还支持在搜索中使用布尔操作。布尔操作的关键字有:

- ADD
- OR
- AND NOT

这实在不像火箭技术那么复杂, 因此本节将直接从一个简单的例子开始并指出其中一些注意点。下面的代码对之前用过的一个例子稍微做了一些修改:

```
SELECT DocumentNode, DocumentSummary, Document
FROM Production.Document
```

```
WHERE CONTAINS(*, '"damaged" OR "seats"');
```

这里所做的是把搜索精确的短语 **damaged seats** 改为搜索两个单词中的一个, 而不需要关心它们是否被一起使用了。执行这段代码将会返回两行结果, 而不是一行。

之前提过, 这里需要注意的地方是 NOT 不能单独使用。只有与 AND 结合使用时, NOT 才能用在全文搜索中。

18.4.7 邻近词

全文索引还允许用户使用邻近词(proximity term)。目前, 被支持的邻近词列表只有一个术语——NEAR。NEAR 的作用就像它的名称一样, 它表明在 NEAR 关键字两边的词语必须要足够的接近。Microsoft 没有具体说明单词之间接近到何种程度才能算得上 NEAR, 但是它指出在大多数情况下, 在 8 到 10 个单词之内就算邻近了。

提示:

从技术上讲, 在邻近关键字列表中有不止一个“单词”, 但是它根本就不是一个“单词”——而是一个符号。如果需要, 读者可以选择使用波浪号(~)来替代 NEAR 关键字。它会以同样的方式工作。就个人而言, 笔者不推荐这样做, 因为这样做会影响代码的可读性。阅读这些代码的人中没有多少人能够了解~的含义, 但是他们中的大多数在碰到 NEAR 时至少会做一下猜测。

对于描述 NEAR 的工作方式的例子, 这里将会使用介绍 CONTAINSTABLE 时所使用的例子。在其他全文查询运算符中, NEAR 的工作方式几乎是一样的, 因此这里将会把注意力集中在在 NEAR 查询中排名是如何发生的, 同时有哪些东西被包含进了查询, 哪些东西没有被包含进查询。

例如, 下面将查找单词 **repair** 和 **instructions**:

```
SELECT Rank, DocumentNode, DocumentSummary
```



```
FROM Production.Document pd
JOIN CONTAINSTABLE(Production.Document, *, 'repair near instructions') ct
ON pd.DocumentNode = ct.[KEY];
```

为了不占用大量的篇幅，这里只列出了前两列，但是注意到返回的两行具有不同排名。

Rank	DocumentNode
3	0x5B40
2	0x7B40

(2 row(s) affected)

如果读者仔细观察结果中的 **DocumentSummary** 列，那么就会注意到两行确实都包含了这两个单词(同样，为了不占用大量的篇幅，这里没有列出所有的列)，但是 **repair** 单词在 **DocumentNode** 0x5B40 行中出现了两次，因此它的排名更高。

提示：

当发现更接近搜索条件的记录的排名比不那么接近搜索条件的记录的排名更低时，不要感到惊讶。记住，即使使用了 **NEAR** 关键字，接近程度只是 SQL Server 用来对行进行排名的几个标准中的一个。其他诸如匹配单词的百分比、大小写以及其他很多因素都会对排名产生影响。

18.4.8 权重

这些排名很酷并且不可琢磨，但是如果在搜索条件中一个单词比其他单词更重要，那么该怎么办呢？

为了处理给一个或多个单词设置优先级的情形，全文索引提供了 **ISABOUT()** 函数和 **WEIGHT** 关键字。其语法如下所示：

```
ISABOUT(<weighted term> WEIGHT (<weight value>), <weighted term> WEIGHT (<weighted term>), ...,n)
```

假设用户想要允许客户在几种不同型号的自行车之间进行选择，但是还需要能够提供“首选”选项。在本例中，假设客户对山地车最感兴趣，但是对旅行车和公路自行车也感兴趣——就按这样的顺序。读者可是使用下面的代码来获取一个排名列表：

```
SELECT Rank, ProductModelID, Name
FROM Production.ProductModel pm
JOIN CONTAINSTABLE(
    Production.ProductModel,
    Name,
    'ISABOUT (Road WEIGHT (.2), Touring WEIGHT (.4), Mountain WEIGHT (.8) )'
) ct
ON pm.ProductModelID = ct.[KEY]
ORDER BY Rank DESC;
```

现在看一下结果：

Rank	ProductModelID	Name
------	----------------	------

```

-----
31          5          HL Mountain Frame
31          7          HL Touring Frame
31          8          LL Mountain Frame
...
...
...
31         123         LL Mountain Rear Wheel
31         124         ML Mountain Rear Wheel
31         125         HL Mountain Rear Wheel
7          126         LL Road Rear Wheel
7          113         Road Bottle Cage
7          93          Road Tire Tube
...
...
...
7          16          ML Road Frame
7          17          ML Road Frame-W
7          9           LL Road Frame
7          6           HL Road Frame

```

(89 row(s) affected)

注意在这个世间上不是所有的事情都能够做到完美的——一些旅行车排到了权重更高的山地车前面，不过如果读者看完这个列表，那么将会发现结果具有非常大的偏好性，在排名中绝大多数的山地车处于前列。

18.4.9 屈折性

实际上该特性不适用于 FREETEXT，因为 FREETEXT 天生就具有屈折性。那么 INFLECTIONAL 是什么呢？好吧，它主要是告诉 SQL Server 不同形式的单词具有同样的含义。其语法如下所示：

```
FORMSOF(INFLECTIONAL, <term>[, <term>[, ...n]] )
```

单词的屈折形式与单词本身具有通常的常规含义。例如，swam 仅仅是 swim 的过去式，它们底层的含义是相同的。

18.5 停止词

正如之前讨论的那样，各种语言中使用大量的单词(全文功能不止支持美国英语)。大多数语言中都有一些经常出现的词，但是这些词并没有多少实质性的含义。例如，在英语中，前置词(you、she、he 等等)、冠词(the、a、an)、以及连接词(and、but、or)就是这样的例子，它们经常出现在句子中，但是在句子中通常又没有什么实际意义。如果 SQL Server 需要关注这些词，同时用户根据它们进行搜索，那么用户会对 SQL Server 所返回的结果感到迷茫。在通常情况下，几乎所有的行都会被返回。这个问题的解决方案就是停止列表(在之前的发行版中被称为干扰词列表)。它是一个包含一系列单词的列表(单个单词叫做停止词)，SQL Server 在进行匹配的时候会忽略它们。

SQL Server 为它支持的每种语言创建了一个默认的停止列表。用户可以使用这个系统提供的停止列表(如果用户需要在命令中显式地引用它,那么通常称之为 SYSTEM),也可以使用 CREATE FULLTEXT STOPLIST 命令来创建自己的停止列表。其语法如下所示:

```
CREATE FULLTEXT STOPLIST <stoplist name>
    [FROM { [<database name>.] <source stoplist name> } | SYSTEM STOPLIST ]
    [AUTHORIZATION <owner name> ]
[;]
```

一般来讲,用户会需要一个填充好的停止列表,因此需要从一些现有的停止列表中来预先填充自己的停止列表。例如,下面会为 AdventureWorks2008 创建一个停止列表,它以 SYSTEM 停止列表中所包含的停止词开始:

```
CREATE FULLTEXT STOPLIST ADStopList
    FROM SYSTEM STOPLIST;
```

注意:

用户创建的停止列表将不会自动与任何全文索引关联起来——用户需要通过 ALTER FULLTEXT INDEX 命令来手工把新的停止列表附到全文索引上。

用户可以根据应用的特定需要向列表中添加单词或从列表中删除单词。例如,如果用户是销售拖拉机拖车装置的,那么可能需要把诸如 hauling 之类的单词添加到干扰词列表中。更有可能的是,客户中有很大大一部分人的名字中包含这个单词,因此这些单词对于搜索来说用处不大。为了在停止列表中添加和删除单词,需要使用 ALTER FULLTEXT STOPLIST 命令,完整的语法如下所示:

```
ALTER FULLTEXT STOPLIST stoplist_name
{
    ADD '<stop word>' LANGUAGE <language number or moniker>
    | DROP
    {
        '<stop word>' LANGUAGE <language number or moniker>
        | ALL LANGUAGE <language number or moniker>
        | ALL }
}
[;]
```

下面的代码在刚才创建的 AWStopList 中添加了一个停止词:

```
ALTER FULLTEXT STOPLIST ADStopList
    ADD 'bicycle' LANGUAGE 1033;
```

如果重新对全文索引进行填充,那么单词 bicycle(在所有文档都包含 bicycles 的业务中,把该单词作为搜索条件将没有任何价值)将会被忽略。

提示:

在停止列表中添加和删除单词有时候是一把双刃剑。当把一个单词添加到列表中后就意味着包含该单词的搜索将不再返回某些结果,而用户可能更加期望返回这些结果。同样,这也意味着,根据单词被使用的频率,这种机制可以大大减少处理时间和目录的大小。

18.6 小结

现在全文搜索是 SQL Server 引擎的核心(在之前的发行版中,它是一个独立的服务),但是每次执行搜索的时候全文守护进程管理器都会孵化一个独立的进程来进行搜索。

在实施全文索引的时候还需要考虑填充进程对服务器所加的负载,并将其与变更被反映到搜索结果所需的时间之间取得某种均衡。如果可能,那么就把全文索引的重新填充推迟到系统处于非高峰期时再进行。

全文索引是一种强大且快速的引用大多数基于字符的列所包含的内容的方法。它本质上比 LIKE 子句更加高效和强大,但是需要额外的空间和处理时间。





第19章

安 全 性

或许，有多少程序员就有多少关于安全的想法。它是一件不一定有必然正确的做法，但是必定有大量错误的做法的事情。

关于安全性首先需要理解的一点是没有绝对安全的应用程序。如果能够使应用程序安全，那么有人肯定能够在系统的其他部分挫败用户的努力并“侵入”系统中。即使有了这样的意识，但是目标仍然是需要把讨厌的入侵者阻挡在系统之外。关于安全性的好消息是在大多数的情况下，用户可以很容易地让事情变得十分复杂，以至于 99.999% 的入侵者不想为此劳神。至于那其他的 0.001%，笔者只能鼓励用户去相信所有的员工都热爱生活，因此他们都面对的是 99.999% 的那部分人。而这 0.001% 的人有可能会去入侵别的系统。

SQL Server 2005 标志着 Microsoft 开始花精力来提高 SQL Server 的安全性了。那些使用 SQL Server 已经有一段时间的用户可能还记得在 SQL Server 2000 的生命期内出现的围绕“Slammer”病毒的吵闹声。Microsoft 在 Slammer 病毒引起恐慌之后大幅修改了 SQL Server 的安全设置，而 SQL Server 2005 则是 Slammer 病毒出现之后的首个完整的发行版，它提供了一系列的功能来阻止黑客的入侵以保护 SQL Server 中的数据的安全性和私密性。在 SQL Server 2005 中新增了大量新功能，而在 SQL Server 2008 中也新增了一些功能，下一版本的 SQL Server 将会新增更多的功能。因此，在安全性这一领域有很多内容需要介绍。

本章将会介绍：

- 安全性基础
- SQL Server 安全性选项
- 数据库和服务角色
- 应用程序角色
- 凭据
- 证书
- 模式管理
- XML 集成安全性问题
- 更高级的安全性

通过本章读者将会发现有很多不同的方式来解决安全性问题。安全性不仅仅是给某人一个用户 ID 和一个密码——读者将会看到很多需要考虑的问题。

在开始介绍本章的例子之前，读者需要加载并执行名为 NorthwindSecure.sql 的脚本。这个脚

本将会创建一个特殊的数据库，本章将会用到这个数据库。读者可以从本书的网站 www.wrox.com 或 www.professionalsql.com 上下载学习所需的资料。

提示：

是的，为了能够运行本章的例子，读者需要创建一个能够工作的数据库——很抱歉。这里使用的是旧的 Northwind 数据库，但是要删除所有对权限的修改。本章所使用的 NorthwindSecure 数据库是一种更典型的场景——即除了创建表和对象等天生所具有的权限之外，绝对不会添加任何权限(这意味着 NONE)。随着本章的深入，读者将会学习到如何处理这种情况以及如何显式地添加所需的权限。

19.1 安全性基础知识

笔者确信本节中所介绍的内容有相当一部分看起来似乎是极为愚蠢的——我的意思是，这些内容难道不是尽人皆知的吗？然而，即使是这些规则中最简单的一个，违背它的情形也屡屡发生，因此笔者可以说：“不，显然不是”。笔者所能做的是请读者能够耐心一点，不要跳过本节。虽然某些内容看起来是显而易见的，但是令人惊讶的是，它们常常被忘记或忽略。

这里要介绍的基础知识包括：

- 一个人，一个登录 ID，一个密码
- 密码过期
- 密码长度和组成
- 尝试登录的次数
- 用户 ID 和密码信息的存储

19.1.1 一个人，一个登录名，一个密码

不断让笔者感到震惊的是，无论我走到哪里，几乎总能发现一个公司至少有一个“全局”用户——几乎一个部门甚至整个公司中所有人都知道网络或特定应用程序的一些登录帐户。通常，这个“全局”用户拥有全权的(也就是说完全的)访问权。对于 SQL Server 来说，一个常见的情形就是安装甚至还嫌麻烦把 sa 的密码设置为非空密码。这确实是一个非常糟糕的现象。

提示：

在 SQL Server 2000 之前，sa 帐户的默认密码是空的——即它没有密码。谢天谢地，现在 SQL Server 不仅改变了这个默认行为，而且还不允许使用简单的密码(取决于用户的 Windows 策略设置)。假设用户的 Windows 策略设置允许空白密码，同时用户坚持使用一个空白密码，那么 SQL Server 将会主动告知用户你是一个不折不扣的白痴。这里需要注意的事情是，在进行开发的时候把密码设置成“简单”的密码确实是非常常见的，但是用户仍然需要记住在把系统部署到生产环境中之前需要修改密码，或者如果开发服务器将直接暴露在 Internet 上或其他一些不可信的访问区域上，那么从一开始就要把密码设置得复杂一点。

即使是现在，当大多数的安装都不使用空密码时，有很多人知道密码是什么的现象也极为普遍。

那么第一个基础知识是如果所有人都能够访问一个本质上是匿名(如果所有人都知道这个帐户,那么任何人都可能使用过这个帐户)并且能够访问所有区域的用户 ID,那么用户已经完全击败了自己的安全模型。同样,如果给每个用户一个拥有完全访问权限的帐户,那么用户再次严重地破坏了自己的安全前景。剩下的唯一真正的好处是能够了解在任意时刻什么人连接到了系统上(假设他们确实使用了自己的登录帐户,而不是那个全局登录帐户)。

应该把拥有全权访问的用户限制在一到两个人之内。在理想情况下,如果用户需要这种全权访问的密码,那么可以创建不同的登录帐户,同时它们各自拥有访问权限,但是每个登录帐户只有一个人知道其密码。

提示:

读者会发现用户常常会与其他人共享他们的密码以便某人能够临时获得某种级别的访问(通常是因为该登录 ID 的所有者不在办公室或者当时没有时间来完成此事)。如果可能的话,应当严禁这样做。

由密码共享所引起的问题是多方面的。首先,一些用户获得了对某些区域的访问权,而这种访问权最初是决定不给予他们的(否则,为什么他们自己没有所需的权限呢?)。如果以前不希望他们有这种权限,那么为什么现在就希望他们有这种权限了呢? 其次,不具有访问权的用户现在可能有了半永久的访问权。由于用户几乎从来不修改他们的密码(除非强制他们去修改密码),被给予密码的人多半能不定期地使用那个登录 ID,并且,笔者向你保证,他们一定会使用这个登录 ID 的! 再次,再次丢失了审计。你可能使用了某种机制根据登录 ID 来跟踪用户的行为。如果拥有该登录 ID 的人员超过了一个,那么如何来确定在某一时刻登录到系统的人员呢?

这意味着如果某人将会离开办公室一段时间,可能是他生病了或休假了,同时另外一个人会临时过来做他的工作,那么就应该专门为此接替的人创建一个新的登录 ID 和密码(或者修改他的当前登录 ID 的访问权限),同时一旦原来那个人回来之后就应该把新创建的登录 ID 删除掉。

总之,尽量不要设置全局用户帐户。如果必须要有全局帐户,那么把使用它的人控制在尽可能少的人数之内。通常,人数应该控制在两个人之内(一个是主用户,另一个是第一个人的候补以防第一个人无法完成任务)。如果确实需要有多个人具有重要的访问权限,那么考虑创建多个具有必要访问权限的帐户(一个用户一个帐户)。只要遵循了这些简单的步骤就已经为系统的安全和审计做了很多工作了。

19.1.2 密码过期

密码过期往往会被滥用或者被忽略。这是因为它是一种很好的做法,但是常常会带来不好的影响。

密码过期背后的原则是对系统进行设置使得密码在过了特定的一段时间之后自动过期。在过了这段时间之后,用户必须要修改密码才能继续访问这个帐户。这个概念已经被提出好多年了,如果读者在一家大企业工作,那么来自会计师事务所的审计师很有可能已经坚持要你实现某种形式的密码过期机制(不,不仅仅是 IT 部门控制这个策略——也可能要强制执行审计财务申报的人员给定的策略)。

从 SQL Server 2005 开始,用户可以为 SQL Server 专用密码强制使用 Windows 身份验证权限。或者,用户也可以只使用基于 Windows 的安全性(下一节中将会介绍更多相关内容)。

1. 做出努力的回报是什么？

那么，密码过期能带来什么呢？还记得最后上一节最后部分所讲的一旦密码共享之后，用户将会永远获取访问权吗？好吧，这里是例外情况。如果让密码过期，那么就更新了安全级别——至少暂时如此。密码可能会被再次共享以便该用户重新获得访问权。尽管这远远不算完美(通常登录 ID 的所有者将会更加乐意再次共享它)，但是它确实解决了密码共享只供一次性使用的问题。通常，共享密码的用户甚至不会意识到其他用户在几个月之后仍然拥有这个密码，并且有时候可能会使用该密码来获取他们自身的安全级别本没有的访问权限。

2. 坏消息

一个可能的情形是好事做过了头。笔者之前提到过很多审计公司都希望他们的客户能够实现一种让用户密码定期过期的模型，如 30 天。这确实是一个很糟糕的主意。

笔者见过的所有采用这种模型的安装——无一例外——在实施了 30 天过期的策略之后，安全性变得更糟糕了。正如读者所预期的那样，问题本质上是多方面的。

- 首先，技术支持的需求大大增加了。当用户如此频繁地修改他们的密码时，他们将完全无法记住所有的密码。他们无法记住该使用哪个月的密码，因此他们将不断请求技术支持来重设他们的密码，因为他们忘记了密码是什么。
- 其次，更重要的是用户已经厌倦了想出密码并记住它们。经验表明，在使用了 30 天过期策略的安装中，90%以上的用户会把他们的密码修改为具有难以置信的可预测性(因而是可入侵的)的单词或单词数字组合。实际上，这通常会导致超过 50%的用户具有同样的密码——他们都使用了类似 MMMYY 的东西作为密码，其中 MMM 代表月份，而 YY 代表年份。例如，在 1996 年 1 月的时候，他们可能会把 JAN96 作为密码。不久之后，这里所有人都会这样做。

提示：

笔者曾经见过一些公司尝试使用密码嗅探器来解决这个问题，它将会用户在用户修改密码的时候对密码进行检查。嗅探过程将会查找包含用户的名字或以月份开头的密码。但是这种机制再好，其作用还是很微弱的。

用户通常要远比想象的聪明。大多数的用户只需要花一星期时间就可以绕过第一个密码嗅探器。他们只需要在密码前面加一个“X”为前缀，而其他部分仍然使用之前用过的 MMMYY 格式。简而言之，嗅探器最终所起的作用微乎其微。但是这不会就此停止：他们会与合作者共享新发现的算法，这样其他人也可以绕过这个“问题”了。

这里的底线是不要滥用过期策略。应当使密码过期时间足够短以获取合适的新旧更替间隔并能够解决共享或窃取密码的问题，但是也不要使过期时间设得太短使得用户产生抗拒心理并开始使用简单的密码。从个人角度来讲，笔者建议密码过期时间不要短于 90 天，但是也不要长于 180 天。

19.1.3 密码长度和组成

现在真的应该为 SQL Server 在该领域所做的努力感到欢欣鼓舞。在以前的版本中，如果使用了 SQL Server 的安全性，那么实际上已经对这一方面做不了多少控制了。现在可以让 SQL Server

强制使用 Windows 密码策略(用户可以使用 Windows 中的实用工具来进行调整)。

1. 密码长度

要知道,用户每在密码中包含一个可能的字母数字位,可能的密码数量就会至少增加 36 倍(如果算上特殊字符,那么增长的倍数会更多,但是即使是 36 倍也足以说明问题)。这意味着单个字符的密码仅有 36 种可能,但是两个字符的密码则有 1296 种可能。如果增加到 3 个字符,则可能的密码增加到了 46 656。如果再加上第四个字符,那么可能的密码就会超过一百万个。随着字符越来越多,这些字符的排列方式也会不断增加。但是,字符增加所带来的问题是用户将越来越难以记住他们的密码或想出新密码。实际上,如果处处都要求超过 5 或 6 个字符,那么终端用户很快就会产生极大的反感。

2. 密码组成

是的,笔者已经指出如果要求用户至少使用 4 个字母数字字符,那么可能的密码组合就超过一百万种。但是用户实际上不会使用所有的这些组合,他们将会使用他们熟悉的单词或名字,因此问题就出现了。考虑到一般人经常使用的单词只有 5000 个左右,而对于黑客来说,不需要尝试太多的单词就能达到目的。

如果没有采用默认的 Windows 密码策略,而是采用了其他策略,那么考虑要求密码至少由一个字母(不是数字,只是字母)和一个数字组成。这就排除了容易猜到的简单数字(人们总是喜欢使用自己的社会安全号码、电话号码或生日等等)和所有单词。用户仍然可以创建容易记住的密码——比如“77pizzas”——但是不可能从字典中找出这些密码。为了入侵系统,黑客不得不尝试每一种组合。

19.1.4 尝试登录的次数

无论在物理上是如何存储用户和密码信息的,登录界面都应该与之关联起来,并限制尝试登录的次数。当用户尝试的次数超过限制时,系统做出的响应在程度上可以是不同的,但是需要建立某种机制使得采用编程的方式来尝试所有的密码的方式变得十分困难。

实际上到底应该允许尝试多少次并不是很重要,只要允许尝试的次数是一个合理的较小的数值即可。笔者通常使用三次,但是在其他地方也见过他们使用四次或五次,所有这些都是可以的。

如果使用 Windows 密码策略,那么 SQL Server 将会根据错误密码限制来检查登录的次数并强制执行该策略。

19.1.5 用户和密码信息的存储

显然,只有当使用自己的安全性系统,而不使用内置的 Windows 和/或 SQL Server 安全性系统时才会需要考虑这些问题(很多 Web 应用程序需要考虑这些问题)。在大多数情况下,存储用户配置和密码信息并没有什么高深的学问。但是还是有一些问题需要考虑:

- 由于需要能够在一开始获取信息,因此需要做如下三件事之一:
 - 把密码编译进客户应用程序或组件(然后需要确保在安装应用程序的服务器上创建恰当的登录帐户和密码)。

- 使用 SQL Server 的加密技术对数据库中的数据进行加密和解密。
- 要求使用双密码——一个密码用来获取用户的常规密码信息，另一个密码让用户进入真正的应用程序。强制用户使用两个登录帐户通常会让人无法接受，因此在大多数情况下，还是会回到其他两种选项中的一个。
- 如果采用了双密码，那么可能会把第一个登录帐户的访问权限限制在只执行某个存储过程上。这样就可以在允许第一个登录帐户获取必要的验证而不会把系统暴露给尝试使用 Management Studio 登录的人员。让存储过程(sproc)接受一个用户 ID 和密码，然后只要简单地返回一个布尔值(true 还是 false 以指示用户是否能够登录)或者返回一个记录集列出用户在客户端能够看到的窗口和函数。如果使用原生的 SELECT 语句，那么就无法对用户能够看到的内容进行限制。

笔者曾经为类似的场景实现过一个解决方案，用一个视图把当前的 SQL Server 登录帐户映射到其他登录信息上。这样，应用程序角色将被用来让应用程序获得对系统的所有访问权限。应用程序需要知道用户能够做什么以及不能做什么。所有用户的登录帐户都有权限执行一个存储过程来请求列出它们自己的权限。这个存储过程如下所示(这只是伪码，因此不要尝试运行这段代码)：

```
CREATE PROC GetUserRights
AS

DECLARE @User varchar(128)
SELECT @User = USER_NAME()
SELECT * FROM UserPermissions WHERE LoginID = @User
```

- 如果准备把密码信息存储在系统中——对密码信息进行加密!!! 这一点无论强调多少次也不过分。大多数用户会把密码用在不止一件事情上，因为需要记忆的东西较少时，生活将会更容易一点。在把密码存储到数据库之前对其进行加密，这样可以确保没有人可能知晓用户的密码信息——甚至是无法在偶然间得到。他们可能会看到密码，但是除非他们有解密的密钥，否则他们看到的信息是没有任何用处的。究竟该使用何种加密手段则要取决于用户自己了。用户可以使用内置的加密方法(本章后面将会讨论其中一些方法)，也可以在应用程序层次上实现自己的加密方法。不管是采用单向还是其他方式，都没有理由不好好保护密码信息。

提示：

从个人角度而言，笔者非常相信单向加密。即一旦密码被加密之后就没有任何可能的方法来解密密码。如果用户丢失了他们的密码，那么他们需要走特定形式的重设机制，然后选择一个新密码。笔者之所以推荐这种方式是因为大多数的用户将会在很多应用程序中使用同样的密码，因此他们用来登入到系统中的密码可能与登入到在线银行系统中的密码是同一个密码。创建一个单向加密系统使得系统管理员获取用户密码进行非法活动的风险降到了最小。

19.2 安全性选项

就内置选项来说，在 SQL Server 中进行安全设置有两个选择：

- **Windows 集成安全性:** 用户将会登录到 Windows 上, 而不是 SQL Server 上。身份验证是通过有可信连接的 Windows 来完成的。
- **标准安全性:** 与登录到 Windows 不同, 用户需要专门登录到 SQL Server 上。身份验证是通过 SQL Server 完成的。

下面详细介绍这两种方式。

19.2.1 SQL Server 安全性

本节将从 SQL Server 内置的登录模型开始。在很长一段时间内, 这是一个安全黑洞, 但是在 SQL Server 2005 中, 这方面变得更加可靠了。尽管相对简单的模型仍然可用, 但是现在用户可以做更多的事情来在提高服务器和数据库的安全性。

使用 SQL Server 安全性时, 用户需要创建一个登录 ID, 而这个登录 ID 是与用户的网络登录信息完全分离的。使用 SQL Server 安全性带来的好处包括:

- 用户不必为了获取访问系统的权限而成为域用户。
- 使用程序来控制用户信息变得更加容易。

其中一些不足之处包括:

- 用户可能需要登录两次或更多——一次用于获取网络访问, 另一次用于为各个应用程序创建到 SQL Server 的连接。
- 对于 DBA 来说, 两个登录帐户意味着需要更多的维护工作。
- 如果需要多个密码, 那么它们很容易变得不一致, 同时它会导致大量的登录失败或忘记密码的情形。(这听起来耳熟吗, “现在来看看哪一个密码是用于这个登录的?”)

使用 SQL Server 安全性进行登录的一个例子是使用 sa 帐户, 读者可能在本书中已经多次使用过该帐户了。不管是如何登录进网络中的, 读者都需要使用一个登录 ID sa 以及相应的密码(但愿你设置了十分安全的密码)来登录到 SQL Server 中。

注意:

实际上, 从长远角度来看, 用户不需要天天使用 sa 来登录以完成任务。为什么? 只需要花一两分钟想想就能明白在使用 sa 帐户时可能会完全是因为意外而发生许多可怕的事情(或者是其他任何拥有系统管理员权限的帐户)。使用 sa 帐户意味着用户拥有完全的权限, 这也意味着在处于错误的数据库中时执行 DROP TABLE 语句会根据用户的命令执行相应的动作——删除表!!! 接着就只剩下说“哎呀”了。但是老板说的话可能会完全不同。

即使总是需要完全访问权, 可以使用 sa 帐户把常规的用户帐户加入到 sysadmins 服务器角色中。这样既提供了 sa 帐户的权限, 又获得了单独密码所带来的安全性以及能够对当前谁登录到系统进行审计跟踪(在跟踪工具 Profiler 中或查看系统活动时)。

19.2.2 创建和管理登录

目前在 SQL Server 上创建登录主要有四种方式:

- 使用 CREATE LOGIN
- 使用 Management Studio
- SQL 管理对象(SMO)

- 使用其他几种选项，这些选项的存在只是为了向后兼容

1. CREATE LOGIN

CREATE LOGIN 作为 Microsoft 把创建数据库和服务器的语法标准化的努力的一部分被添加进了 SQL Server 2005 中。它抛弃了旧式的 `sp_addlogin` 方式，在之前的发行版中都是通过使用这个存储过程来创建登录的，现在其语法看起来类似于 SQL 中常见的 `CREATE <object> <object type>`，但是其语法中需要包括一些与存储过程所需的选项类似的选项。

基本的语法是非常直观简单的，但是选项的组合方式则是一个比较难以理解的地方。整个语法如下所示：

```
CREATE LOGIN <login name>
[ { WITH
    PASSWORD = '<password>' [ HASHED ] [ MUST_CHANGE ]
    [, SID = <sid>
        | DEFAULT_DATABASE = <database>
        | DEFAULT_LANGUAGE = <language>
        | CHECK_EXPIRATION = { ON | OFF }
        | CHECK_POLICY = { ON | OFF }
        [ CREDENTIAL = <credential name>
        [, ... <next option>] ]
    } |
    { FROM
        WINDOWS
        [ WITH DEFAULT_DATABASE = <database>
            | DEFAULT_LANGUAGE = <language> ]
        | CERTIFICATE <certificate name>
        | ASYMMETRIC KEY <asymmetric key name>
    }
}
```

这里的关键部分是在登录名之后是选择 FROM 子句还是选择 WITH 子句，下面将介绍 FROM 子句和 WITH 子句及其相关的选项。

CREATE LOGIN ... WITH

WITH 子句将直接定义与 SQL Server 身份验证(而非其他任何身份验证方法)相关的登录选项。只有启用了 SQL Server 安全性(对比于只使用 Windows 验证)时这个选项才可用。这里的选项数量很多，看上去有些令人生畏，因此表 19-1 把它们拆开来讲述。

表 19-1

选 项	说 明
PASSWORD	这个选项的作用就像其名字所说的那样。这里需要留意的地方是密码到底是以明文(在这种情况下，SQL Server 在添加密码时会对密码进行加密)存储还是以散列形式(在这种情况下用户需要提供一个 HASHED 关键字，下面将会介绍这个关键字)存储
HASHED	这个选项跟在密码的后面，只有当提供的密码经过散列(加密)之后才能使用这个选项。这样，SQL Server 在添加密码时将不会再次对其进行加密
MUST_CHANGE	这又是一个顾名思义的选项。简而言之，如果指定了这个选项，那么用户在第一次登录时将会被提示修改密码

(续表)

选 项	说 明
SID	这个选项允许手工指定 SQL Server 使用什么 GUID 来识别该登录。如果没有指定该选项(笔者认为这样做是一种极端的情况), 那么 SQL Server 将会自动生成一个
DEFAULT_DATABASE	每当用户登录时该数据库将会用作当前数据库
DEFAULT_LANGUAGE	这是给用户发送错误和系统消息时所采用的语言
CHECK_EXPIRATION	设置 SQL Server 是否执行密码过期策略。密码默认是不会过期的。把该选项设置为 ON 将会启用该策略
CHECK_POLICY	设置 SQL Server 是否执行密码策略(长度、字符要求等等)。默认情况下, 密码必须要符合 Windows 密码策略的要求。把该选项设置为 OFF 实质上是允许使用任何密码
CREDENTIAL	指定该登录名映射到的凭据(本章后面将会介绍这一概念)。简而言之, 该选项将把登录映射到一组权限上, 从而允许执行 SQL Server 之外的动作(如网络访问等等)

所有选项都可以混合使用, 并且只有在使用 HASHED 和 MUST_CHANGE 选项时这些选项之间的顺序才有关系(如果要全部使用这些选项, 那么它们必须要跟在 PASSWORD 选项的后面)。

CREATE LOGIN ... FROM

FROM 子句表明该登录名不是特定于 SQL Server 的。FROM 子句指定了登录的来源。来源分为几种不同的类别:

- **WINDOWS:** 在这种情况下会映射到一个现有的 Windows 登录名或组上。基本上, 这表明“使用这个现有的 Windows 用户或组, 并赋予他们访问 SQL Server 的权限”。注意这里是说“或组”。用户可以把 SQL Server 登录名映射到一个 Windows 组上, 这表明该组中的任何成员都会被赋予访问 SQL Server 的权限。这对管理整个网络中的用户来说的确是非常方便的。例如, 如果想要会计部门都在 SQL Server 中拥有一组特定的权限, 那么可以创建一个名为 Accounting 的 Windows 组, 然后把它映射到一个 SQL Server 登录上。如果聘用了新人员, 那么只需要把他加入到那个组, 然后他就能够访问 Accounting 组能够访问的所有 Windows 资源, 同时还拥有了 Accounting 组所拥有的 SQL Server 权限。如果使用 Windows 作为 FROM 子句的来源, 那么与基于 SQL Server 的登录名类似, 还可以提供一个 WITH 子句, 但是只能使用默认的数据库和语言。
- **CERTIFICATE:** 这种类型的登录是基于 X.509 证书的, 用户需要使用 CREATE CERTIFICATE 命令把服务器与 X.509 证书关联起来。如何使用证书存在几种不同的方式, 但本质上它们最终都将作为经过验证的安全加密密钥来使用。SQL Server 有自己的“证书颁发机构”, 或者可以导入由其他源产生的证书。本质上, 证书都是作为登录到 SQL Server 的授权来使用的。
- **ASYMMETRIC KEY:** 非对称密钥与证书是同一概念的两种不同的表现方式。本质上, 如果它是 SQL Server 信任的密钥, 那么 SQL Server 就会赋予权限。非对称密钥仅仅是另一种提供安全密钥的方法。

为了准备本章余下部分所使用的例子, 读者需要在 Windows 中创建一个用户, 本章后面将会在例子中赋予该用户权限以及删除该用户的权限。这里把这个测试用户命名为 TestAccount, 读者

可以取一个自认为合适的名字(确保记住把本章的例子中的用户替换为合适的名字)。一旦在 Windows 中有了测试帐户之后就可以把它加入到 SQL Server 中了(同样,读者需要把“HOBBS”修改为自己的系统的名字):

```
CREATE LOGIN [HOBBS\TestAccount] FROM WINDOWS
WITH DEFAULT_DATABASE - NorthwindSecure;
```

现在测试帐户已经有了登录到 SQL Server 的权限了。但是,注意虽然把 NorthwindSecure 数据库设置为 TestAccount 帐户的默认数据库,但是它仍然没有访问这个数据库的权限(稍后将会介绍这部分内容)。

ALTER LOGIN

与大多数在 SQL 中见到的 CREATE 语句一样,CREATE LOGIN 也有一个 ALTER LOGIN 形式的附加语句。与大多数的 ALTER 语句一样,其语法基本上是相应的 CREATE 语句所提供的选项的子集:

```
ALTER LOGIN <login name>
[ { ENABLE | DISABLE } ]
[ { WITH
    PASSWORD = '<password>'
  [ { OLD_PASSWORD = '<old password>'
    | [ UNLOCK ] [ MUST_CHANGE ] }
    | DEFAULT_DATABASE = <database>
    | DEFAULT_LANGUAGE = <language>
    | NAME = <new login name>
    | CHECK_EXPIRATION = { ON | OFF }
    | CHECK_POLICY = { ON | OFF }
    | CREDENTIAL = <credential name>
    | NO CREDENTIAL
```

其中大多数选项与它们在 CREATE 语句中的作用是一样的,其中的一些不同之处如表 19-2 所述:

表 19-2

选 项	说 明
ENABLE DISABLE	启用或禁用该登录。它有点像一个指示器,表明该登录在系统中是否被认为是活跃的,同时不应该把 ENABLE 何 UNLOCK 混淆起来(它们是不同的事情)。禁用一个登录并不会删除该登录,只不过无法使用该登录帐户来登录系统,启用登录将会重新激活该登录
OLD PASSWORD	只有当给定的登录使用 ALTER LOGIN 修改自己的密码时,该选项才适用。拥有修改密码权限的安全管理员不太可能会知道旧密码,相反,他们可以在不知道旧密码的情况下设置一个新密码
UNLOCK	该选项允许用户在登录帐户由于超过密码错误计数而导致被锁定时继续尝试登录
NAME	该选项允许在保留所有的权限和其他登录属性的情况下修改登录名
NO CREDENTIAL	该选项解除登录与它自己之前所映射到的凭据之间的关联关系

DROP LOGIN

它的工作方式与 SQL Server 中其他所有的 DROP 语句的工作方式一样:

DROP LOGIN <login name>

执行该语句后，登录将会被删除。

2. 使用 Management Studio 创建登录

使用 Management Studio 创建一个登录帐户是相当简单的，这与创建 SQL Server 中的大多数对象是一样的。只需要在“对象资源管理器”中导航到恰当的节点(本例中是“安全性”|“登录名”)，然后右击并选择“新建登录名”。接着将会出现在本书中多次见过的 CREATE 对话框，在对话框中把该登录的属性值调整到合适的值(包括本章之前讲述的“CREATE LOGIN”一节中所提到的内容加上接下来要讲述的几个附加部分)，如图 19-1 所示。



图 19-1

只有第一组属性(“常规”属性)可以映射到 CRETE LOGIN 语法。其他选项卡映射到了其他对象上，本章后面将会创建这些对象。

提示：

本章后面将会介绍其他几种以某种方式与登录关联起来的对象。现在只需要注意 Management Studio 中的用户界面是如何让用户一次完成所有的事情即可。随着本章的深入，读者将会发现当使用代码创建这些对象时需要逐个创建，而不能像在 Management Studio 中那样一次性完成(正如读者可能猜想的那样，它实际上只是提前收集了所有必需的信息，然后执行了所有那些单独的程序步骤而已)。

3. SQL 管理对象

这部分内容超出了本章的范围(本书后面将会专门安排一章讲述 SMO)，但是这里需要特别指出的是与使用 CREATE 语句的方式相比，SMO 能够使用简单的对象模型来创建登录。更多信息请参考第 23 章。

4. 遗留选项

当在 SQL Server 之前的发行版中考虑创建登录的方式时，有三种比较重要的方式可以采用：

- **sp_addlogin 以及相关的存储过程：**它是一个存储过程，除了 SQL Server 2005 之前的发行版不支持 CREATE LOGIN 语句中的几个部分之外，本质上对应于 CREATE LOGIN 语句。其基本要素(采用不同于证书或非对称密钥的方式来创建典型的登录帐户)都是一样的。后面将会详细介绍 sp_addlogin。
- **WMI: Windows Management Instrumentation** 是行业标准 Web 管理协议的一个实现。在 SQL Server 2000 首次发行的时候，人们认为基于 WMI 的模型将会成为自动 SQL Server 管理的主要方式。但是，最后没有基于 WMI 的模型能够在 SQL Server 中完成所有的任务，同时现在看来那些努力在很大程度上已经被抛弃了。现在，WMI 已经超出了本书的范围，但是需要知道它的存在，如果需要管理较早版本的 SQL Server 或者熟悉 WMI 的其他作用并且希望将 SQL Server 脚本添加到更大的 WMI 计划中，那么它仍然是一个可用的选项。

sp_addlogin 快速概览

这个存储过程所完成的工作就像其名字所说的那样，它是一种旧式的实现 CREATE LOGIN 功能的方法。虽然在遗留代码中，它被广泛使用着，但是笔者强烈建议在新开发中避免使用 sp_addlogin。它只有一个参数是必需的，但是在大多数情况下需要使用两个或三个。还有很多额外的参数，但是它们极少被用到。其语法如下所示：

```
EXEC sp_addlogin [@loginame -] <'login'>
    [,[@passwd -] <'password'>]
    [,[@defdb -] <'database'>]
    [,[@deflanguage -] <'language'>]
    [,[@sid -] 'sid']
    [,[@encryptopt -] <'encryption_option'>]
```

表 19-3 列出了各个参数。

表 19-3

参 数	说 明
@loginame	顾名思义——这是将会被使用的登录 ID
@passwd	顾名思义——用上述登录 ID 进行登录时所使用的密码
@defdb	默认数据库。该选项定义了用户登录时的第一个“当前”数据库。通常，这应该是应用程序使用的主数据库。如果没有指定，那么 master 数据库将会成为默认数据库(用户通常不想出现这种情况，因此确保提供该参数)
@deflanguage	该用户的默认语言。如果需要支持本地化，那么可以使用该选项来覆盖系统默认的语言
@sid	一个二进制数字，它将成为登录 ID 的安全标识符(SID)。如果没有提供 SID，那么 SQL Server 将会生成一个。由于 SID 必须唯一，因此提供的任何 SID 都必须不存在于系统中。当在一台不同的服务器上还原数据库或者迁移登录信息时使用一个特定的 SID 是非常方便的
@encryptopt	用户的登录 ID 和密码信息是存储在 master 数据库的 sysusers 表中的。@encryptopt 参数指定了是否对存储在 master 数据库中的密码信息进行加密。在默认情况下(或者指定参数值为 NULL)，密码信息是经过加密的。另一个选项是 skip_encryption，顾名思义——密码将不会被加密。还有一个选项是 skip_encryption_old，该选项只是为了向后兼容而存在的，因此不应该使用这个选项

正如读者所看到的那样，大多数项都是能直接映射到 CREATE LOGIN 命令的。如果不是为了向后兼容性而需要使用 sp_addlogin 的话，建议使用 CREATE LOGIN 命令。

sp_password

由于已经介绍了 sp_addlogin，因此接下来将介绍 sp_password。虽然 ALTER LOGIN 能够解决登录的密码维护问题(应该使用该命令来完成任务)，但是 sp_addlogin 却没有这种功能——sp_password 完成这个任务。其语法是相当直观的：

```
sp_password [[@old -] <'old password'>,<br>
    [@new -] <'new password'>
    [,[@loginame -] <'login'>]
```

当然，新密码和旧密码参数的工作方式与读者预期的工作方式完全一样。需用从用户那里接受密码并把它们传递给存储过程。然而，需要注意的是登录名是一个可选的参数。如果没有提供登录名，那么默认将会修改当前连接所使用的登录的密码。注意 `sp_password` 不能作为事务的一部分来执行。

提示：

读者可能在想“大多数的系统不都是要求用户输入两次新密码的吗？”。它们确实的这样做的。因此接下来的问题是“为什么 `sp_password` 不这样做呢？”这个问题的答案是非常简单的——因为 SQL Server 把这个问题留给了开发人员。开发人员需要在客户端应用程序中包含在使用 `sp_password` 之前检查两次输入的新密码的逻辑。`ALTER LOGIN` 也存在同样的问题。

`sp_grantlogin`

这个存储过程模拟 `CREATE LOGIN ... FROM` 的功能把登录与 Windows 登录关联起来(在 SQL Server 2005 之前并不存在源自证书和非对称密钥的映射)。其语法是相当简单的：

```
sp_grantlogin [@loginname -] '<Domain Name>\<Windows User Name>'
```

同样，这个存储过程的存在只是为了向后兼容。对于 2005 和之后的安装，使用 `CREATE LOGIN ... FROM` 语法(这是新代码中所采用的主流方式)。

19.2.3 Windows 身份验证

Windows 身份验证能够把来自信任 Windows 域的登录映射到 SQL Server 中。

它只是一个模型，在这个模型中可以直接向现有的 Windows 域用户帐户或组提供 SQL Server 权限，而不需要强制用户使用独立的密码以及进行单独的登录。

Windows 身份验证允许：

- 只需要在一个地方维护用户的访问。
- 只需要通过把一个用户添加到 Windows 组中就能赋予该用户 SQL Server 权限(这意味着管理员甚至不需要访问 SQL Server 来为用户授予权限)。
- 用户只需要记住一个密码和登录名。

接下来介绍如何把特定的权限赋给特定的用户。

19.3 用户权限

用户权限最简单的定义为“用户能做什么以及不能做什么”。在本例中，这个简单的定义是相当能说明问题的。

用户权限可以分为三个类别：

- 登录的权限
- 访问某个特定数据库的权限
- 在数据库中某个特定对象上执行特定动作的权限

由于已经介绍了如何创建登录，因此这里将集中介绍登录所拥有的具体权限。

19.3.1 授予访问特定数据库的权限

如果想要用户能够访问数据库，那么第一件需要做的事情就是赋予该用户访问数据库的权限。这个任务可以使用 Management Studio 来完成，只要把用户添加到服务器的“数据库”节点下的“用户”成员中即可。如果要使用 T-SQL 来添加用户，那么应该使用 CREATE USER 命令。与 sp_addlogin 类似，为了向后兼容，还可以使用 sp_grantdbaccess 存储过程。

注意：

注意当在数据库中 CREATE 一个用户时，其权限实际上是存储在数据库中并被映射到该用户的服务器标识符上。在还原数据库时可能需要在还原数据库的服务器标识符上重新映射用户权限。

1. CREATE USER

CREATE USER 命令在数据库上添加一个新用户。该用户可以来自现有的一个登录帐户、证书、非对称密钥或只局限于当前数据库的本地帐户。其语法如下所示：

```
CREATE USER <user name>
[ { ( FOR | FROM )
  {
    LOGIN <login name>
    | CERTIFICATE <certificate name>
    | ASYMMETRIC KEY <key name>
  }
  | WITHOUT LOGIN ]
[ WITH DEFAULT_SCHEMA = <schema name> ]
```

表 19-4 快速介绍一些其中某些元素的含义：

表 19-4

选 项	说 明
LOGIN	授予对当前数据库的访问权限的登录名
CERTIFICATE	与该用户相关联的证书的逻辑名。注意必须预先使用 CREATE CERTIFICATE 命令创建该证书
ASYMMETRIC KEY	与该用户相关联的非对称密钥的逻辑名。注意必须预先使用 CREATE ASYMMETRIC KEY 命令创建该密钥
WITHOUT LOGIN	创建一个只能在当前数据库中活动的用户。可以使用这个选项来建立一个特定的安全环境，但是该用户不能被映射到当前数据库之外的登录名，它也无法访问其他任何数据库
WITH DEFAULT_SCHEMA	为当前用户建立一个默认的模式，而不采用默认的“dbo”模式

要授予 TestAccount 帐户访问 NorthwindSecure 数据库的权限，可是使用下面的命令：

```
CREATE USER [HOBBS\TestAccount]
FOR LOGIN [HOBBS\TestAccount]
WITH DEFAULT_SCHEMA - dbo;
```

上面的代码授予了访问指定数据库(本例中是 NorthwindSecure)的权限并把登录的默认模式设置为数据库所有者。

sp_grantdbaccess

这是授予一个登录帐户访问给定数据库的遗留方法。其语法如下所示:

```
sp_grantdbaccess [@loginame =] <'login'>[, [@name_in_db =] <'name in this db'>
```

注意授予的是当前数据库的访问权限——因此在执行命令的时候需要确保想要授予用户访问权限的数据库是当前数据库。这里的登录名是用来登录到 SQL Server 的实际登录 ID。name_in_db 参数允许给该用户取另外的标识别名, 该别名只对该数据库可用——所有其他数据库仍然使用默认的登录 ID 或者在授予用户访问该数据库的权限时所定义的别名。定义别名将会影响标识函数, 如 USER_NAME()。工作在系统级别的函数, 如 SYSTEM_USER 将仍然返回基本的登录 ID。

19.3.2 授予访问数据库中对象的权限

好, 现在用户拥有了登录名并且能够访问数据库了, 那么现在是否就万事大吉了呢? 如果事情真有那么简单就好了! 现在当然还没有一切就绪。

在用户能够访问哪些对象的问题上, SQL Server 提供了级别相当精细的控制。在大多数的时候可能希望用户能够访问一些信息, 但是却不希望用户能够访问数据库中的其他一些信息。例如, 一个客户服务人员可能能够查看和维护订单信息——但是可能不希望他们乱看工资信息。反之亦然——人力资源部门的员工可能能够编辑雇员记录, 但是可能不希望他们能够在一次交易中给某人很大的折扣。

SQL Server 允许给 SQL Server 中的不同对象指派一组不同的权限。可以指派权限的对象包括表、视图以及存储过程。触发器隐含具有创建它的用户所拥有的权限。

对象上的用户权限可以分为六种不同的类型, 如表 19-5 所示:

表 19-5

用 户 权 限	说 明
SELECT	允许用户“查看”数据。如果用户拥有这个权限, 将能够在其被授予权限的表或视图上运行 SELECT 语句
INSERT	允许用户创建新数据。拥有这个权限的用户能够运行 INSERT 语句。注意, 与很多系统不同, 拥有 INSERT 权限并不一定意味着拥有 SELECT 权限
UPDATE	允许用户修改现有的数据。拥有这个权限的用户能够运行 UPDATE 语句。与 INSERT 语句一样, 拥有 UPDATE 权限并不一定意味着拥有 SELECT 权限
DELETE	允许用户删除数据。拥有这个权限的用户能够运行 DELETE 语句。同样, 拥有 DELETE 权限并不一定意味着拥有 SELECT 权限
REFERENCES	假设用户要插入的表中有一个外键约束, 而用户在该外键所引用的表上并没有 SELECT 权限, 拥有这个权限的用户能够向这样表中插入行
EXECUTE	允许用户 EXECUTE 指定的存储过程

在指派权限时可以根据需要在特定表、视图或存储过程上混合搭配这些权限。

可以在 Management Studio 中指派这些权限，只需要导航到服务器的“安全性”节点下的“登录名”选项即可。然后右击该用户并选择“属性”。接着根据当前是处于数据库中还是安全性节点中将会出现不同的对话框，但是不管哪种情况都会出现设置权限的选项。使用 T-SQL 指派权限将会用到三个命令，即使只想使用 Management Studio 来指派权限，了解这三个命令也是非常有益的(其术语是一样的)。

1. GRANT

GRANT 把对象上指定的访问权限指派给指定的用户或角色，对象是 GRANT 语句的主体。GRANT 语句的语法如下所示：

```
GRANT
    ALL [PRIVILEGES] | <permission>[,...n]
    ON
    <table or view name>[(<column name>[,...n])]
    |<stored or extended stored procedure name>
    TO <login or role name>[,...n]
    [WITH GRANT OPTION]
    [AS <role name>]
```

ALL 关键字表明想要授予的是该对象类型适用的所有权限(EXECUTE 永远不适用于表)。如果不使用 ALL 关键字，那么需要为授予权限的对象提供一个或多个特定的权限。

PRIVILEGES 是一个没有实际作用的关键字，它只是提供了 ANSI/ISO 兼容性。

ON 关键字是一个占位符，指示下面是想要授予其权限的对象。注意，如果在表上授予权限，那么可以通过指定受影响的表列列表来指定下至列级别的权限——如果没有提供特定的列，则默认将影响所有的列。

提示：

在对列级别的权限的看法上，Microsoft 做的似乎都是些表面功夫。让用户能够在特定表的特定列上进行 SELECT 看起来是一个非常酷的想法，然而在列级别权限的使用以及 Microsoft 为实现列级别的权限所做的工作看来，它确实让安全性处理太错综复杂了。正因为如此，近几年来关于该主题的文献很少对其进行深入讨论，有时看起来 Microsoft 想要放弃列级别的安全性。他们有时候推荐不要使用这种方式(其他时候则不提供任何建议)——如果需要限制用户只能查看特定列，那么考虑使用视图。

TO 语句所做的事情正如读者所期望的那样：它指定了接受这些权限的对象。它可以是一个登录 ID 或者一个角色名。

WITH GRANT OPTION 允许被授予权限的用户反过来把权限授予给其他用户。

提示：

笔者不推荐使用这个选项，因为很快，跟踪哪些人拥有哪些权限将会成为一件非常痛苦的事情。当然，读者总是可以通过 Management Studio 来查看对象的权限，但是那样将会处于被动状态，而非主动状态——那样将查看当前访问级别的问题在哪里而非提前停止不希望发生的访问。

最后，但并非最不重要的，是 AS 关键字。它处理一个登录属于多个角色的问题。

现在可以介绍一两个例子了。在本章后面读者将会看到之前创建的 `TestAccount` 已经有了一些访问权限，因为它是 `Public` 角色的成员——所有数据库用户都属于这个角色，无法从该角色中删除用户。但是，还有大量项目 `TestAccount` 是没有访问权限的(因为它只属于 `Public` 角色，而 `Public` 并没有这些权限)。

以 `TestAccount` 用户的身份登录。接着在 `Region` 表上尝试运行 `SELECT` 语句：

```
SELECT * FROM Region;
```

SQL Server 将会很快给出警告信息，警告你正在访问不该访问的内容：

```
Server: Msg 229, Level 14, State 5, Line 1
SELECT permission denied on object 'Region', database 'NorthwindSecure', owner 'dbo'.
```

单独以 `sa` 登录——也可以通过选择 `File@@Connect` 菜单项在同一个 QA 实例中登录。然后为这个新连接选择 SQL Server 安全性，并输入恰当的密码以 `sa` 登录。现在执行 `GRANT` 语句：

```
USE NorthwindSecure;

GRANT SELECT ON Region TO [HOBBS\TestAccount];
```

提示：

注意读者需要用自己的计算机或域名替换“HOBBS”。

现在切回 `TestAccount` 连接(记住，以哪个用户进行连接的信息显示在连接窗口的标题栏上)，再次尝试运行 `SELECT` 语句：这次结果好多了：

RegionID	RegionDescription
1	Eastern
2	Western
3	Northern
4	Southern

(4 row(s) affected)

接着尝试运行另外的语句。这次要在 `EmployeeTerritories` 表上运行同样的测试和命令：

```
SELECT * FROM EmployeeTerritories;
```

上面的代码执行失败了——同样，用户没有这个权限，因此需要授予在该表上的权限：

```
USE NorthwindSecure;

GRANT SELECT ON EmployeeTerritories TO [HOBBS\TestAccount];
```

现在，如果重新运行选择语句将能够正常工作：

EmployeeID	TerritoryID
1	06897
1	19713
...	


```
...
...
9          48304
9          55113
9          55439
```

```
(49 row(s) affected)
```

为了增加一点变化, 下面尝试在这张表上运行 INSERT 语句:

```
INSERT INTO EmployeeTerritories
VALUES
    (1, '01581');
```

SQL Server 会立即报告错误。用户没有所需的权限, 因此需要授予用户这些权限(使用 sa 连接):

```
USE NorthwindSecure;

GRANT INSERT ON EmployeeTerritories TO [HOBBS\TestAccount];
```

现在再次尝试 INSERT 语句:

```
INSERT INTO EmployeeTerritories
VALUES
    (1, '01581');
```

一切进展顺利。

2. DENY

DENY 显式地阻止用户访问指定的目标对象。DENY 的关键之处在于它覆盖了任何 GRANT 语句。由于一个用户可以属于多个角色(稍后介绍), 因此用户可能是被授予访问权限的角色的成员, 同时又受 DENY 的影响。如果在用户的个人和角色的权限混合中同时存在 DENY 和 GRANT, 那么 DENY 总是优先的。简而言之, 如果用户或者用户所属的角色中对某一权限有 DENY 语句, 那么用户将无法在那个对象的访问权限。

其语法与 GRANT 语句看起来非常类似:

```
DENY
    [ALL] [PRIVILEGES] | <permission> [, ...n]
ON
    <table or view name> [(column [, ...n])]
    | <stored or extended stored procedure name>
TO <login ID or roll name> [, ...n]
[CASCADE]
```

同样, ALL 关键字表明想要拒绝所有适用于该对象类型的权限(EXECUTE 永远不适用于表)。如果不使用 ALL 关键字, 那么需要为该对象提供一个或多个需要拒绝的特定权限。

注意:

现在 ALL 关键字只为了向后兼容而存在。同时, 弄明白 ALL 关键字实际上已经不再影响“所

有”权限也是非常重要的。虽然它确实影响着主流权限(如 SELECT),但是随着 ALL 关键字变得越来越过时,不受它影响的权限列表一直在增长。

PRIVILEGES 仍然是一个新关键字,它没有实际作用,只是为了提供 ISO 兼容性。

ON 关键字是一个占位符,指示下面将是想要拒绝授予其权限的对象。

到现在为止,所有事情的工作方式与 GRANT 语句的工作方式是相当类似的。CASCADE 关键字对应于 GRANT 语句中的 WITH GRANT OPTION。CASCADE 告诉 SQL Server 拒绝该用户在 WITH GRANT OPTION 规则下授予其他人的访问权限。

为了运行一个关于 DENY 的例子,使用 TestAccount 登录帐户尝试运行一个简单的 SELECT 语句:

```
USE NorthwindSecure;  
  
SELECT * FROM Employees;
```

上面的代码大概返回 9 行结果。这里并没有授予 TestAccount 访问权限,怎么会获取这些结果的呢? TestAccount 属于 Public 角色,而 Public 角色已经被授予了访问 Employees 的权限。

下面假设不希望 TestAccount 拥有这些访问权限。不管是什么原因,TestAccount 是一个例外,这里不希望该用户能够查看那些数据——只需要发出一个 DENY 命令即可(记住使用 sa 登录来发出 DENY 命令):

```
USE NorthwindSecure;  
  
DENY ALL ON Employees TO [HOBBS\TestAccount];
```

当再次使用 TestAccount 运行 SELECT 语句时将会得到一个错误。现在这个帐户不再拥有访问权限了。注意由于这里使用了 ALL 关键字,因此 TestAccount 也会被拒绝授予 Public 角色拥有的 INSERT、DELETE 和 UPDATE 权限。

提示:

同样,注意已经不推荐使用 ALL 了,因此在运行前面的示例代码时收到一个警告信息。这里介绍这个例子的目的是为了让读者明白 ALL 关键字使用的非常广泛,可能会在遗留代码中发现这个关键字的踪迹。

3. REVOKE

REVOKE 删除了之前运行的 GRANT 或 DENY 语句所产生的影响。可以把它想象成某种“撤消”语句。

其语法是 GRANT 和 DENY 语句的混合:

```
REVOKE [GRANT OPTION FOR]  
  [ALL] [PRIVILEGES] | <permission>[,...n]  
ON  
  <table or view name>[(<column name> [,...n])]  
  |<stored or extended stored procedure name>  
TO | FROM <login ID or roll name>[,...n]  
[CASCADE]  
[AS <role name>]
```


这些元素的含义在本质上与它们在 GRANT 和 DENY 语句中的含义是一样的。这里再次列出以便读者能够从书架上拿下本书浏览一下 REVOKE 的语法。

同样, ALL 关键字表明想要撤消所有适用于该对象类型的权限。如果没有使用 ALL 关键字, 那么需要为被撤消权限的对象提供一个或多个特定的被撤消权限。

PRIVILEGES 仍然没有实际的作用, 它的存在只是为了与 ANSI/ISO 兼容。

ON 关键字是一个占位符, 指示下面将是被撤消权限的对象。

CASCADE 关键字对应于 GRANT 语句中的 WITH GRANT OPTION。CASCADE 告诉 SQL Server 撤消该用户在 WITH GRANT OPTION 规则下授予其他人的访问权限。

同样, AS 关键字指定了想要基于什么角色发出这个命令。

使用 sa 连接撤消刚才授予的 NorthwindSecure 数据库中 Region 表上的权限:

```
REVOKE ALL ON Region FROM [HOBBS\TestAccount];
```

执行完上面的代码之后 TestAccount 帐户将无法在 Region 表上运行 SELECT 语句了。

要删除 DENY 也需要发出一个 REVOKE 语句。这次将重新获得访问 Employees 表的权限:

```
USE NorthwindSecure;
```

```
REVOKE ALL ON Employees TO [HOBBS\TestAccount]
```

现在已经介绍了如何使用这些命令对单个用户进行访问控制, 下面将介绍如何通过以组为单位进行管理来极大地简化权限管理问题。

19.3.3 用户权限和语句级别的权限

用户权限不仅仅能够与数据库中的对象关联起来——它还能与还没有被绑定到任何特定对象的语句关联起来。SQL Server 允许对运行不同的语句进行权限控制, 这些语句包括:

- CREATE DATABASE
- CREATE DEFAULT
- CREATE PROCEDURE
- CREATE RULE
- CREATE TABLE
- CREATE VIEW
- BACKUP DATABASE
- BACKUP LOG

到现在为止, 除了两个备份命令(这两个命令的作用是不言自明的, 因此这里将不再花时间介绍它们, 第 22 章将会介绍它们, 现在只需要记住它们能够在语句级别上进行控制)之外, 其他所有命令都已经介绍过了。

好, 那么如何指派这些权限呢? 实际上, 既然读者已经见过如何在对象上使用 GRANT、DENY 和 REVOKE 语句, 那么应该对语句级别的权限控制也相当熟悉了。从语法上来看, 它们与对象级别的权限的工作方式是一样的, 但是它们要更加简单一点(因此不需要填写那么多内容)。其语法如下所示:

```
GRANT {ALL | <statement[,...n]>} TO <login ID>[,...n]
```

很简单吧？为了进行一次快速测试，现在来验证一下测试用户是否已经有了 CREATE 的权限。确保用 TestAccount 登录，然后运行下面的代码。别忘了把下面代码中的 HOBBS 替换为自己的域名：

```
USE NorthwindSecure;

CREATE TABLE TestCreate
(
    Coll int Primary Key
);
```

运行上面的代码时，会看到一个错误消息：

```
Server: Msg 262, Level 14, State 1, Line 2
CREATE TABLE permission denied, database 'NorthwindSecure', owner 'dbo'.
```

现在使用 sa 帐户登录到 SQL Server 上(或者使用另一个在 NorthwindSecure 数据库上拥有 dbo 权限的帐户)，接着运行下面的命令来授予权限：

```
GRANT CREATE TABLE TO [HOBBS\TestAccount];
```

运行上面的命令的结果应该是一个命令运行成功的确认消息。现在尝试一下再次运行 CREATE 语句。记住重新使用 TestAccount 帐户登录：

```
USE NorthwindSecure;

CREATE TABLE TestCreate
(
    Coll int Primary Key
);
```

这次一切都很顺利。

DENY 和 REVOKE 的工作方式与它们在对象级别的权限上的工作方式一样。

19.4 服务器和数据库角色

从最普通的意义上来讲，角色与 Windows 中的组是同一个事物，即它是一组访问权限(或拒绝)，这组权限在指派给角色时自动与用户关联起来。

注意：

角色是一组访问权限，通过简单地把用户分配到那个角色中就可以将这一组访问权限一起指派给该用户。

一个用户至少属于一个角色，也有可能同时属于几个角色。这种组织方式是相当便捷的，因为可以把访问权限组织成更小以及逻辑性更强的组，然后把它们混合搭配为最适合用户的规则。

角色可以分为两类：

- 服务器角色
- 数据库角色

提示:

很快读者就会看到第三种称为角色的事物——尽管笔者希望 Microsoft 选用另外的名字——应用程序角色。这是一种特殊的方式用来把用户化名到不同的权限组中。应用程序角色不是分配给用户的，它是一种让应用程序根据用户不同而拥有不同权限的方法。正因为如此，笔者通常不认为应用程序角色是真正意义上的“角色”。

服务器角色局限于那些在 SQL Server 发行时就已经内置的角色，它们主要用来进行系统维护以及授予进行非特定于数据库的任务(如创建登录帐户和创建链接服务器)的权限。

与服务器角色类似，有很多内置的(或“固定的”)数据库角色，同时可以定义自己的数据库角色来满足特定的需求。数据库角色用来建立和组织一个给定数据库之内的特定用户权限。

下面分别介绍这两种角色。

19.4.1 服务器角色

所有可用的服务器角色都是“固定的”角色，并且从一开始就存在了。自 SQL Server 安装完毕之后，可用的服务器角色就已经存在了，如表 19-6 所示。

表 19-6

角 色	特 性
sysadmin	这个角色可以在 SQL Server 上执行任何活动。任何拥有这个角色的用户本质上是这个服务器的 sa。这个服务器角色的引入使得 Microsoft 能够在某一天删除 sa 登录帐户——实际上，联机丛书本质上已经把 sa 看成遗留物了。值得指出的是 Windows 的 Administrators 组将自动映射为 SQL Server 的 sysadmin 角色。这意味着任何属于服务器的 administrators 组的成员都对 SQL 数据拥有 sa 级别的访问权限。如果需要，可以从 sysadmin 角色中删除 Windows administrators 组以提高安全性，防范漏洞
serveradmin	这个角色可以设置服务器级别的配置选项或关闭服务器。尽管它在范围上相当有限，但是该角色的成员所控制的功能对服务器性能会产生非常重大的影响
setupadmin	这个角色局限于管理链接服务器和启动过程
securityadmin	对于专门用于管理登录帐户、读取错误日志以及 CREATE DATABASE 权限的帐户来说，这个角色是非常便捷的。在很多情况下，这个角色是经典的系统操作员角色——它能够处理大多数的日常事务，但是却没有全能的超级用户所拥有的全局访问权限
processadmin	能够管理运行在 SQL Server 中的进程——这个角色可以根据需要杀死长时间运行的进程
dbcreator	被限制用于创建和修改数据库
diskadmin	管理磁盘文件(数据被指派给哪个文件组、附加和分离数据库等等)
bulkadmin	这个角色有些怪异。它被明确地创建出来用以执行 BULK INSERT 语句，否则只有拥有 sysadmin 权限的用户才能执行这个语句。坦白讲，笔者无法理解为什么不像其他权限一样使用 GRANT 命令来授予权限，但它的确没有。记住，即使某个用户已经被加入到 bulkadmin 组中，他也只有运行该语句的权限，并没有访问需要执行该语句的表的权限。这意味着除了要把用户添加到 bulkadmin 组中之外，还需要授予他们在执行 BULK INSERT 的表上进行 INSERT 的权限。此外，还需要确保他们在 BULK INSERT 语句所引用的表上拥有恰当的 SELECT 访问权限

可以为在服务器上担当管理角色的各个用户混合搭配这些角色。一般来讲,笔者怀疑只有那些最大型的数据库才会用到除 sysadmin 和 securityadmin 之外的角色,但是有其他角色仍然是相当方便的。

提示:

在本章前面,笔者曾就全局用户会带来的问题进行过激烈抨击。因此读者可能不会惊奇我会对新的 sysadmin 角色被加回到 7.0 版本中而感到欣喜若狂。从发展的角度来看,这个角色的存在意味着现在不需要所有人都是用 sa 来登录了。只需要让需要这个级别的访问权限的用户成为 sysadmin 角色的成员即可,这样他们就不再需要使用 sa 来登录了。但是需要小心谨慎,让一个用户已知拥有这个级别的访问权限可能会导致意外事故(从安全性的角度来看,它不会阻止你删除对象或执行类似的任务)。据我了解,很多 IT 部门都给管理员配备多个登录帐户:一个拥有完整的 sysadmin 访问权限,而另一个是进行日常工作的登录帐户,它拥有完成大多数任务所需的权限,但是不具有产生破坏性的风险极高的权限。管理员仍然可以完成他们所需完成的任务,但是他们只有自觉使用特殊的高访问权限的帐户来登录才能执行高风险的活动(这意味着他们在做执行任务的时候会考虑得更多)。

19.4.2 数据库角色

数据库角色的范围仅局限于单个数据库——这是因为一个用户属于一个数据库的 db_datareader 角色并不意味着他就属于另一个数据库的 db_datareader 角色。数据库角色可以分为两类:固定的和用户自定义的。

1. 固定的数据库角色

与存在若干个固定的服务器角色一样,也存在若干个固定的数据库角色。其中一些有其预先定义好的专门用途,使用常规语句是无法复制它们的(即无法创建一个拥有同样功能的用户自定义的数据库角色)。但是,大多数角色都是用来处理更为常见的情形并让事情变得更简单的,如表 19-7 所示。

表 19-7

角 色	特 性
db_owner	这个角色工作起来就好像它是所有其他数据库角色的成员。使用这个角色可以让多个用户完成相同的功能和任务,就好像他们是数据库的所有者一样
db_accessadmin	这个角色拥有服务器角色 securityadmin 的部分功能,但是它局限于被指派的单个数据库和创建用户(不是个人的权限)。它无法创建新的 SQL Server 登录帐户,但是属于这个角色的成员能够把 Windows 用户和组以及现有的 SQL Server 登录帐户添加到数据库中
db_datareader	可以在数据库中所有的用户表上发出 SELECT 语句
db_datawriter	可以在数据库中所有用户表上发出 INSERT、UPDATE 和 DELETE 语句
db_ddladmin	可以添加、修改和删除数据库中的对象
db_securityadmin	Securityadmin 服务器角色在数据库级别的等价物。这个数据库角色无法在数据库中创建用户,但是能够管理数据库中的角色和角色中的成员以及管理语句和对象上的权限

(续表)

角 色	特 性
db_backupoperator	备份数据库(打赌你不会想到那样的一个角色)
db_denydatareader	数据库中所有表和视图上的 DENY SELECT 的等价物
db_denydatawriter	与 db_denydatareader 类似, 但是只影响到 INSERT、UPDATE 和 DELETE 语句

与固定的服务器角色类似, 即使是最大型的数据库也不会用到所有这些角色。其中一些角色是无法用自己的数据库角色替换的, 而其他一些角色则只是为了方便处理常见的简单粗糙的情形而已。

2. 用户自定义的数据库角色

提供固定角色只是为了让读者起步, 真正影响安全性的支柱是创建并指派用户自定义的数据库角色。在这些角色中, 读者可以决定角色需要包含哪些权限。

在用户自定义的角色中, 读者可以像对待单个用户一样采用完全一样的方式执行 GRANT、DENY 和 REVOKE 操作。使用角色的好处是可以根据访问需求对用户进行分类。使用角色可以让发生在一个地方的变更传播到所有类似的用户中(至少是那些指派给这个角色的用户)。

创建用户自定义的角色有两种方式:

- CREATE ROLE(首选方法)
- Sp_addrole(为了向后兼容)

下面分别介绍这两种方法。

使用 CREATE ROLE 创建用户自定义的角色

创建自己的角色的首选方法是使用 CREATE ROLE 命令。与本章中介绍的其他命令类似, 这个命令的功能已经迁移到与 ANSI/ISO 更兼容的语法上了, 在之前的版本中是用一个系统存储过程来支持该功能的——本例中是 sp_addrole 系统存储过程。与其他命令类似, 其语法是相当直观的:

```
CREATE ROLE <role name> [AUTHORIZATION <owner name>][;]
```

role name 是角色的名字。常用的命名模式例子包含部门名(Accounting、Sales、Marketing 等等)或特定的职位名(CustomerService、Salesperson、President 等等)。使用这类角色实际上可以很容易地向系统中添加新用户。如果会计部门招聘了新员工, 那么只需要把她或他添加到 Accounting 角色(或者更加明确一点, 可能会是 AccountsPayable 角色)中即可——不需要研究“这个人该拥有哪些权限?”

AUTHORIZATION 参数是可选的, 它允许覆盖哪个数据库用户或角色将拥有这个新角色(在默认情况下, 这个新角色将会属于运行 CREATE 命令的用户, 通常是 db_owner 角色的一个成员)。

下面的代码将创建一个角色:

```
USE NorthwindSecure;
```

```
CREATE ROLE OurTestRole;
```

执行上面的代码会出现一条友好的消息说新角色已经被添加了。

现在需要做的是向角色中添加一些值, 这些值实际上是要指派给它的一些权限。为了完成这

个任务,只需要像本章前面给实际用户指派权限那样使用 GRANT、DENY 和 REVOKE 语句即可:

```
USE NorthwindSecure;  
  
GRANT SELECT ON Territories TO OurTestRole;
```

现在所有属于这个新角色的用户将会拥有在 Territories 表上进行 SELECT 的权限(除非在他们的安全性信息的其他地方存在 DENY 语句)。

3. 使用 sp_addrole

正如前面所提到的,还有一个旧式的基于系统存储过程的命令,它的存在只是为了向后兼容。同样,其语法是相当简单的:

```
sp_addrole [@rolename =] <'role name'>  
[,[@ownername =] <'owner'>]
```

owner 的含义与系统中所有其他对象是一样的。它的默认值为数据库所有者,笔者强烈建议不要修改它的值(换句话说,忽略这个可选的参数)。如果需要使用旧式的语法来添加特殊的测试角色,那么可以使用下面的代码:

```
USE NorthwindSecure;  
  
EXEC sp_addrole 'OurTestRole';
```

不管使用何种语法,现在应该可以开始添加用户了。

向角色中添加用户

有了所有这些角色固然很好,但是如果不把用户指派给它们,那么它们将没有任何用处。令人感到奇怪的是到现在为止还没有任何新命令来向角色添加用户。因此必须要使用旧式的系统存储过程模型,即调用 sp_addrolemember 系统存储过程并提供数据库名和登录 ID:

```
sp_addrolemember [@rolename =] <role name>,  
[@membername =] <Login ID>[;]
```

上面代码中各个参数的含义都是不言自明的,因此下面将直接介绍例子。这里从验证 TestAccount 没有访问 Territories 表的权限开始:

```
SELECT * FROM Territories;
```

果然被拒绝了(还没有访问权限):

```
Server: Msg 229, Level 14, State 5, Line 1  
SELECT permission denied on object 'Territories', database 'Northwind', owner 'dbo'.
```

下面将把 TestAccount Windows 用户添加到 OurTestRole 角色中:

```
USE NorthwindSecure;  
EXEC sp_addrolemember OurTestRole, [HOBBS\TestAccount];
```

现在是时候再次尝试运行 SELECT 语句了——这次取得了更大的成功(应该返回 53 行结果)。

从角色中删除用户

万物必有起落兴衰，添加到角色中的用户必然也会被删除。

除了所使用的系统存储过程不同之外，从角色中删除用户的工作方式与向角色添加用户几乎一样，这个存储过程叫做 `sp_droprolemember`，其形式如下所示：

```
sp_droprolemember [@rolename =] <role name>,  
  [@membername =] <security account>[;]
```

下面回到例子中，从 `OurTestRole` 数据库角色中删除 `TestAccount`：

```
USE NorthwindSecure;
```

```
EXEC sp_droprolemember OurTestRole, [HOBBS\TestAccount];
```

读者将会收到一个友好的确认消息说一切进展顺利。现在再次运行 `SELECT` 语句：

```
SELECT * FROM Territories;
```

果然再次得到了无法访问的错误。

可以通过这种方式向任何角色或者从任何角色中添加和删除用户。不管这个角色是用户自定义的角色或者固定的角色，还是系统或数据库角色，都没有任何关系。不管是何种情况，其工作方式都是一样的。

注意也可以通过 `Management Studio` 完成这些任务。要修改一个角色的权限，只需要右击“安全性”节点的“角色成员”（在特定的数据库之下），然后使用复选框指派权限。要把一个用户添加到角色中，只需要导航到用户节点（同样，在特定的数据库之下），然后右击并选择“属性”。然后在数据库或服务器角色中勾选该用户所属的所有角色。

删除角色

删除角色与添加角色一样容易。其语法是非常简单的：

```
EXEC sp_droprole <'role name'>[;]
```

执行上面的代码后角色就被删除了。

19.5 应用程序角色

应用程序角色与数据库和服务器角色有些不同。实际上，使用术语“角色”的事实可能会使读者认为它们是紧密相关的。但是它们之间没有关系。

应用程序角色实际上更像是用户的安全性别名。应用程序角色允许定义一个访问列表（由数据库的个人权限或组权限构成）。它们还与用户类似，因为它们有自己的密码。但是，它们又与用户登录帐户不同，因为它们无法“登录”。一个用户帐户必须要首先登录，然后他或她才能激活应用程序角色。

那么应用程序角色用来做什么呢？用于应用程序——还有其他用途吗？读者可能会经常遇到想要根据用户访问数据库时所处的上下文来决定他所拥有的一组权限的情形。有了应用程序角色之后就可以授予用户只读访问数据库（仅 `SELECT` 语句），同时又允许他们在应用程序范围之内修改数据。

其过程如下所示:

- (1) 用户登录(很有可能使用应用程序提供的登录窗口)。
- (2) 验证登录帐户, 接着用户收到他的或她的访问权限。
- (3) 应用程序将会执行一个叫做 `sp_setapprole` 的系统存储过程, 并提供一个角色名和密码。
- (4) 验证应用程序角色, 接着连接被切换到应用程序角色的上下文(该用户拥有的所有权限都会消失——现在他或她将拥有应用程序角色的权限)。
- (5) 用户将在该连接有效期间继续保持应用程序角色的权限, 而不是他或她个人的登录帐户的权限。用户无法回到他或她自己的访问权限。

读者可能只想把应用程序角色作为真正的应用程序场景的一部分来使用, 同时会编写代码来把应用程序角色的权限写入应用程序。此外还会把所需的密码编译进应用程序, 或者把这些信息存储在需要时可以访问的本地文件中。

19.5.1 创建应用程序角色

要创建应用程序角色可以使用 `CREATE ROLE` 的一个变体——`CREATE APPLICATION ROLE`。它又是一个相当容易的命令, 其语法如下所示:

```
CREATE APPLICATION ROLE <role name>  
    WITH PASSWORD = <'password'> [, DEFAULT_SCHEMA = <schema name>][;]
```

与本章中其他 `CREATE` 命令类似, 其参数也是不言自明的。因此下面将直接使用它来创建一个应用程序角色:

```
CREATE APPLICATION ROLE OurAppRole WITH PASSWORD = 'P@ssw0rd';
```

就是那么快, 现在应用程序角色已经创建好了。与大多数的安全性项目类似, 还可以使用一个系统存储过程来实现这个功能, 但是同样, 这个系统存储过程的存在只是为了向后兼容。它与 `CREATE` 语法类似, 如下所示:

```
sp_addapprole [@rolename =] <role name>,  
    [@password =] <'password'>[;]
```

使用系统存储过程创建前面示例中创建的应用程序角色的代码如下所示:

```
EXEC sp_addapprole OurAppRole, 'P@ssw0rd';
```

19.5.2 向应用程序角色添加权限

向应用程序角色添加权限与向其他事物中添加权限一样。只需要把使用登录 ID 或者常规服务器或数据库角色的地方替换为应用程序角色名即可。

同样, 下面直接介绍示例:

```
GRANT SELECT ON Region TO OurAppRole;
```

现在应用程序角色在 `Region` 表上有了 `SELECT` 权限——到现在为止它还没有任何其他权限。

19.5.3 使用应用程序角色

使用应用程序角色是调用一个系统存储过程(`sp_setapprole`)并给它提供应用程序角色名和应

用程序角色的密码的过程。其语法如下所示：

```
sp_setapprole [@rolename =] <role name>,
  [@password =] {Encrypt N<'password'>|<'password'>
  [,[@encrypt =] {'none' | 'odbc'}
  [, [@fCreateCookie = ] {true | false} ]
  [, [@cookie = ] <variable holding cookie> OUTPUT}[;]
```

Role name 是想要激活的应用程序角色的名称。

既可以以明文的方式提供密码，也可以使用 ODBC 的 encrypt 函数来加密密码。如果想要加密密码，那么需要在 Encrypt 关键字之后用引号把密码括起来，并且在密码前面加一个大写的 N——指示 SQL Server 这是一个 Unicode 字符串(如果打算加密密码，那么密码必须为 Unicode)，SQL Server 会做出相应的处理。如果不想加密密码，那么只需要提供密码，同时不要使用 Encrypt 关键字。

提示：

需要指出的是加密只是 ODBC 和 OLE DB 客户端的一个选项。因此无法在 Query 窗口(它使用 SqlClient)中测试这个选项。此外，如果不使用加密，那么需要意识到任何人只要通过嗅探网络中的包就能够查看密码。简而言之，如果不使用 ODBC 加密来发送密码，那么需要在连接上使用 SSL 或 IPsec(两种安全的传输方法)。

下面是语法中的 cookie 部分。设置一个 cookie(并存储 @cookie 输出变量返回的值)为在应用程序角色被激活之前就活跃的权限集提供了某种书签。然后可以使用 sp_unsetapprole 存储过程恢复到原来的安全上下文(由 cookie 指示)。sp_unsetapprole 的语法如下所示：

```
sp_unsetapprole <cookie variable>
```

执行上面的代码后会回到原来的安全上下文。

下面直接介绍一个简单的例子，从验证 TestAccount 用户的一些状态开始。到目前为止(假设读者已经按顺序完成了本章前面介绍的示例)，TestAccount 用户应该无法访问 Region 表，但是能够访问 EmployeeTerritories 表。可以通过运行一系列 SELECT 语句来验证这个情况：

```
SELECT * FROM Region;
```

```
SELECT * FROM EmployeeTerritories;
```

第一个 SELECT 语句应该会返回一个错误，而第二个 SELECT 语句应该返回 50 行左右。现在来激活前面创建的应用程序角色，使用 TestAccount 用户输入下面的代码：

```
EXEC sp_setapprole OurAppRole, 'P@ssw0rd';
```

执行上面的代码应该会返回一个确认消息说应用程序现在应处于“活跃”状态了。

现在再次运行两个 SELECT 语句，这次正常工作和无法正常工作的语句反过来了。即在激活了应用程序角色之后，原来能够访问 EmployeeTerritories 的 TestAccount 无法访问 EmployeeTerritories 了，而原来无法访问 Region 的 TestAccount 可以访问 Region 了，因为应用程序角色提供了访问权限。

由于没有存储 cookie(笔者故意在这里才指出…)，因此没有办法停止当前连接的应用程序角色。在这种情况下只有几个可用的选项，如切换到另一个应用程序角色，但是没有 cookie 就没有

办法回到原来的安全性上下文。

断开 TestAccount 的连接, 然后使用 Windows 验证为 TestAccount 创建一个新连接。接着再次尝试运行两个 SELECT 语句, 这次读者将会发现原来的权限集被还原了。

19.5.4 删除应用程序角色

当不再需要服务器上的应用程序角色时, 可以使用 DROP 命令来删除该角色, 读者应该很熟悉这个命令了:

```
DROP APPLICATION ROLE <role name>
```

当然, 这个命令还有一个版本是系统存储过程(同样, 只是为了向后兼容!)。它的名字是 sp_dropapprole, 其语法如下所示:

```
sp_dropapprole [@rolename =] <role name>
```

使用 DROP 语法从系统中删除应用程序角色只需要发出下面的命令(从 sa 发出):

```
DROP APPLICATION ROLE OurAppRole;
```

19.6 更高级的安全性

本节将介绍一些“另外需要考虑的问题”。所有这些主题都超出了本章一开始定义的基本规则的范畴, 但是它们给出了一些问题的解决思路, 并且也给出了如何关闭系统中的一些常见漏洞方法。

19.6.1 如何处理 guest 帐户

guest 帐户提供了一种拥有默认访问方法。如果 guest 帐户是活跃的, 那么可能会发生一些事情:

- 登录帐户获得了所有数据库上的 guest 级别的访问权限, 但是并没有明确授予它们这些访问权限。
- 外部用户可以通过使用 guest 帐户登录来获得访问权限。虽然这要求他们知道 guest 的密码, 但是他们已经知道了这个用户是存在的(尽管他们还可能知道 sa 帐户也是存在的)。

从个人角度来讲, 笔者对 SQL Server 做的第一件事情就是删除 guest 帐户所有的访问权限(在默认情况下它是没有访问权限的, 因此要做的事情应该很少)。它是一个漏洞, 以一种直觉上并不会想到的方式提供了访问权限(读者可能认为当把权限指派给某人时——他就只拥有那么多权限。当 guest 活跃时事情就不是这样的了)。

然而, guest 帐户的一个用途实际上是相当巧妙的——在应用程序角色中使用它。在这个场景中, 让 guest 帐户拥有访问数据库的权限, 但是除了登录到数据库的权限之外没有任何其他权限——即 guest 帐户只是把登录的数据库变为“当前”数据库。接着可以使用 sp_setapprole 来激活一个应用程序角色, 这样就可以使得匿名用户能够登录到服务器中, 并且拥有适当的权限。但是如果要使用应用程序, 那么就需要执行一个有用的登录。

注意:

这绝对是一个要保护应用程序角色的密码的场景, 就好像你的工作都靠它一样(可能就是这样

的)。可以使用 ODBC 加密选项并且不允许通过 Internet 进行这种类型的访问!

19.6.2 TCP/IP 端口设置

如果使用 TCP/IP, 那么 SQL Server 默认会使用 1433 端口。可以把端口看成类似于无线电信道的东西, 在哪个信道上进行广播并不重要, 因为如果没有人侦听那个信道, 那么将不会带来什么好处。

使用默认的 1433 端口可以使事情变得非常方便。所有客户端都会自动使用 1433 端口, 除非指定其他端口, 这意味着使用默认端口是少了一件需要操心的事情。

但是, 这里存在的问题是任何潜在的 SQL Server 黑客都知道 99% 的 SQL Server 在侦听 1433 端口。如果 SQL Server 直接连接到 Internet 上, 那么笔者强烈建议把端口改为非标准的端口号。可以询问网络管理员, 让其推荐一个可用的端口。记住如果修改了服务器侦听的端口号, 那么还需要修改基于 IP 的客户端所使用的端口号。例如, 如果打算使用 1402 端口, 那么需要进入客户端网络实用工具为服务器建立一个特定的条目并指定使用的 IP 端口为 1402。

也可以告诉客户端动态决定端口号, 只需要选中“动态确定端口”复选框即可。

注意修改端口号对安全性并不会有很大的提高。实际上黑客可以使用一个端口扫描器或者其他工具来判断防火墙上打开的端口, 然后根据收到的响应就能够相当精确地判断出哪个软件使用了哪个端口。也就是说每做一点小事也就使得黑客的入侵变得稍微困难一点。

19.6.3 不要使用 sa 帐户

所有学习 SQL Server 超过 10 分钟的人都知道系统管理员帐户。SQL Server 用固定的服务器角色 sysadmin 来模拟 sa 用户级别的访问权限, 因此强烈建议把登录帐户添加到该角色中, 然后把 sa 的密码修改为很长且很难理解的密码——一个不值得花费时间来破解的密码。如果只需要 Windows 验证, 那么就关闭 SQL Server 安全性, 这样就会一劳永逸地解决了 sa 帐户的问题。

19.6.4 保持 xp_cmdshell 的隐秘性

记住在把使用 xp_cmdshell 的权限授予谁这个问题上一定要小心谨慎。它能够运行所有 Windows 命令行提示符中的命令。用户所拥有的权利取决于 SQL Server 运行在哪个帐户下。如果是一个系统或管理员帐户(大多数情况是这样的), 那么 xp_cmdshell 的用户就会在服务器上拥有非常重要的权限(例如, 它们可以把其他网络位置的文件复制到服务器上, 然后执行那些文件)。把问题说得更严重一点——还有相当部分的服务器是运行在 Windows 域管理员帐户的上下文中的——现在任何使用 xp_cmdshell 的用户对整个网络都拥有了相当开放的访问权限!!!

简而言之, 对于那些不准备授予他们服务器或者可能甚至是域的管理员权限的用户, 不要授予他们访问 xp_cmdshell 的权限。

19.6.5 不要忘记把视图、存储过程和 UDF 作为安全性工具

记住视图、存储过程和 UDF 对于隐藏数据都有很大的帮助。视图通常可以用来替代列级别的安全性。它们可以使用户认为自己正在访问整张表, 但是实际上他们只对所有数据的一个子集

拥有访问权限(还记得前面要筛选掉雇员的敏感信息(如薪水)的例子吗?)。存储过程和 UDF 能够起到同样的作用。可以授予用户执行一个存储过程或 UDF 的权限,但是这并不意味着就能够获取表中所有的数据(他们只能获取存储过程或 UDF 给予他们的数据)——终端用户甚至可能不知道底层是哪个表提供数据的。此外,视图、存储过程和 UDF 有它们自己隐含的权力——即虽然试图和存储过程使用了表,但这并不意味着用户拥有访问该表的权限。

19.7 证书和非对称密钥

到目前为止,本书已经在几个不同的地方(包括本章前面)提到了加密的概念。为不同级别的服务器体系结构定义加密密钥的主要机制是证书和非对称密钥。它们是完成基本任务的两种不同的方法,并且在很多情况中它们是可以互换的。不管是使用证书还是使用非对称密钥都需要记住它们就像是房间的钥匙一样——如果让每个人都拥有密钥,那么它们很快就会失去自己的价值(现在所有人都要钥匙,那么要把谁锁在外面呢?)。

为了满足使用不同加密密钥分离不同的控制权限的需求,SQL Server 支持多个级别的密钥的概念。在每个服务器安装中,SQL Server 会维护一个服务主密钥。它是用 Windows 级别的服务主密钥加密的。同样,每个数据库包含一个数据库主密钥,根据用户的选择,可以使用服务主密钥加密数据库主密钥。在每个数据库中,用户可以定义证书和/或非对称密钥(两种都是某种形式的密钥)。总之,其层次结构如图 19-2 所示。

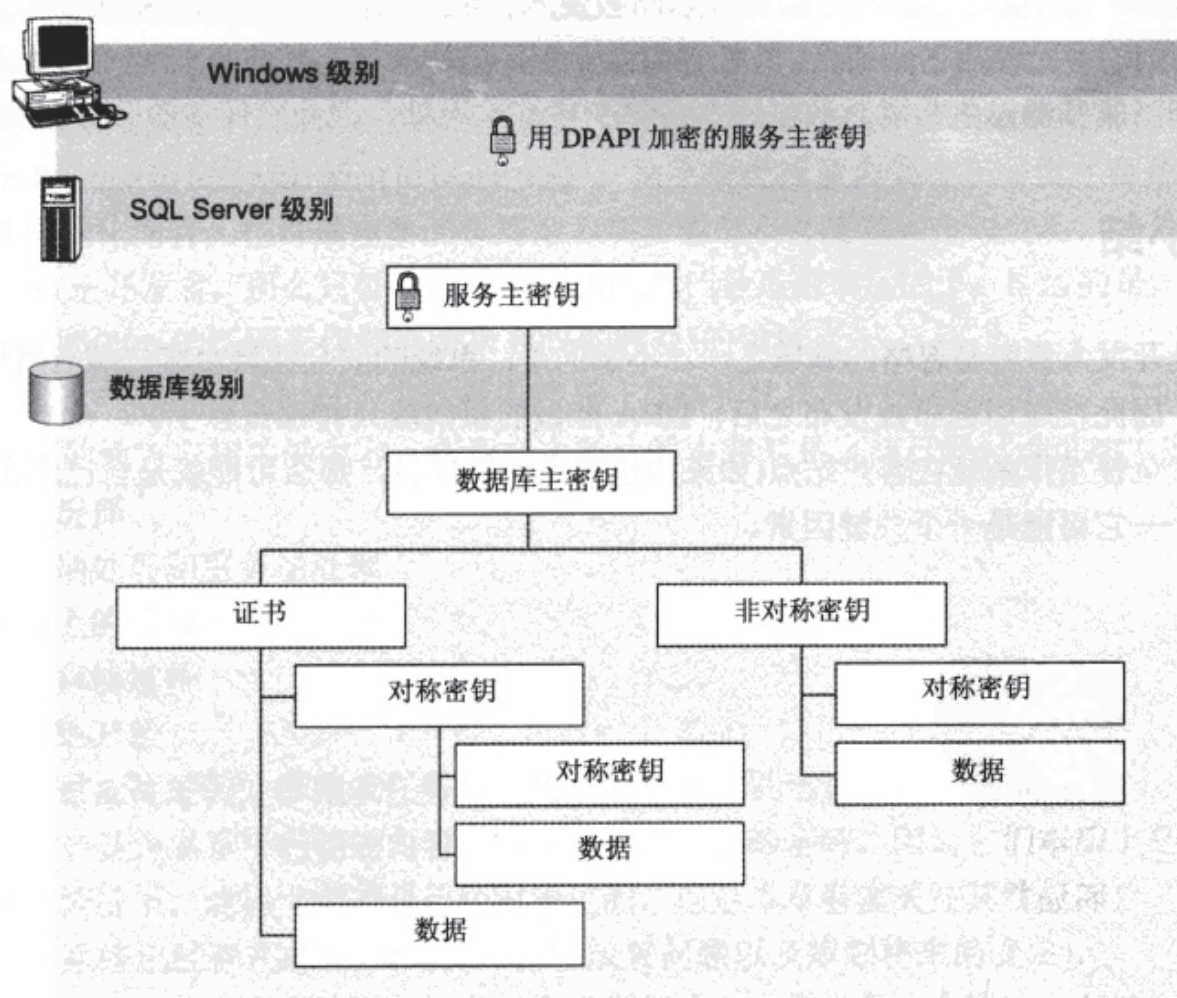


图 19-2

19.7.1 证书

从 SQL Server 2000 开始, SQL Server 有了自己的证书颁发机构, 或称 CA, 同时也支持第三方的 CA。CA 颁发证书, 证书中包含一个加密密钥和一些与证书相关的基本信息, 如证书有效的时间范围(起始日期和到期日期)、证书持有者的名字以及证书颁发机构的信息。使用 CREATE CERTIFICATE 命令可以向服务器中添加一个证书。

19.7.2 非对称密钥

非对称密钥的工作方式与证书大体相同, 但是它是单向指定的, 并且不需任何发行机构验证。与证书一样, 需要指定加密密钥并使用该密钥加密敏感信息。可以使用 CREATE ASYMMETRIC KEY 命令添加非对称密钥。

19.7.3 数据库加密

在 SQL Server 2005 中添加的大多数加密函数都是面向加密特定的数据这个想法的。它们要求使用特殊的函数(至于哪个函数则要取决于所使用的加密类型)来加密数据, 然后使用另一组函数来解密数据。

从 SQL Server 2008 开始又增加了加密整个数据库的概念。注意这里的想法不是用密码保护数据库中的数据, 而是为了保护整个数据库以防被侵入。使用数据库级别的加密实际上是把数据库文件以及它们的备份使用密钥加密, 并把它们保存在数据库所在的服务器上(除非复制了服务器的证书, 否则当服务器发生故障时数据库备份将变得毫无用处, 因此确保备份服务器的证书)。

19.8 小结

安全是开发人员容易忽略的领域之一。不幸的是, 系统的安全性将由客户端应用程序如何行事来决定, 因此在应用程序被发布之后, DBA 能做的事情就只有那么多了。

要把安全性当作系统在客户站点(如果构建的是内部项目, 那么可能就是自己所在的站点)成败的命脉——它可能是一个关键因素。

第20章

设计性能卓越的数据库

从作为作者的角度来讲,本章可能是本书中最难写的一章,不过这不是由一般的原因引起的。通常,这里的问题是如何通过易懂的方式讲述复杂的内容。随着邻近本书的结尾,希望我在这点上是成功的——尽管还有很多值得改进的地方。从以往的经验和本书中已经介绍的主题来看,到目前为止读者应该已经对本章将要讨论的主题有了坚实的基础。这意味着笔者可以相对自由地讲述事情的本质,而不需要太过担心会引起混淆。

那为什么书写本章对我来说是十分困难的呢?因为确定把哪些内容放入本章以及后面的姊妹章节该讲述哪些内容确实是非常困难的。读者可以看到,这并不是一本讲述性能优化的书——性能优化本身就可以写一本书了。这是一本帮助你在使用 SQL Server 的开发中获得成功的书。拥有一个性能卓越的系统对于成功是至关重要的。问题就像 Bob Seger 歌曲中所唱的那样:“要有所为,有所不为(what to leave in, what to leave out)”。这里要把注意力集中在哪里才会最有帮助呢?

对于性能优化而言,也许最重要的事情是必须要明白不可能做到全知全能。如果你是一位普通的 SQL Server 开发者,那么只要知道其中的 20%就已经是很幸运的了。幸运的是,性能调优是适用 80-20 法则(20%的正确工作能够带来 80%的收益)的领域之一。

在本书的这一版中将会对这一主题稍作展开,在结构化决策的基础上添加了“如何发现可以进行性能优化的地方”相关的内容。本章中大部分的内容都是之前已经讨论过的主题,包括:

- 索引选择
- 客户端处理和服务端处理
- 策略上的反规范化
- 组织存储过程
- 使用临时表
- 使用重复性进程分步完成任务与使用长时间运行的进程一次性完成任务

读者可能会认为本章中讨论的内容实际上是属于设计的范畴,因为它们本质上是有点结构性的。在大多数情况下,这些主题都是已经讨论过的,但是本章着重关注其性能部分。下一章将会介绍如何处理系统已经存在的情形(维护、位置放置问题以及规划将来的变更)。

然而,一个共同的观点是读者应该两个章节都阅读:本章只是一个开始。在性能方面最大的事情实际上是停下来并思考一下。由于某些奇怪的原因,在使用 SQL 时有一种倾向,即采用第一次出现在脑海中的方案往往会成功。读者需要使用与完成任何其他开发工作相同的思考方式去处理查询、存储过程以及数据库设计。还需要记住的是 T-SQL 代码只是系统的一部分——硬件、客

户端代码、SQL Server 配置以及网络问题等都是“代码之外”的因素，它们可能对系统产生极大的影响。

注意：

性能对于不同的人来说意味着不同的事情。例如，很多人把它想成简单的响应时间(查询完成得多快)。还有一个概念是对性能感觉上的体验(很多用户关注要多久才能接收到足够的数据以开始工作，而不是完成整个查询需要多久)。另一种看法可能专注于伸缩性(例如，在响应时间遭受影响之前或者知道用户之间开始互相影响之前，到底能在系统上加多大的负载？)。

两个介绍性能的章节中很多示例和建议都是关于原始速度的——返回结果的速度有多快——但是在适当的地方会涉及感知性能和伸缩性问题。确保在设计中已经考虑了性能的所有方面——不仅仅是完成时间。

20.1 优化时机

好吧，这可能看起来有点显而易见，但是在整个开发过程中，对性能的考虑要远远早于编写代码。实际上，对性能的考虑应该从需求收集过程就开始，并且永远不会结束。

在需求收集阶段有关性能调优的最重要的方面是什么呢？现在，这个时候无法物理地对系统进行调优，但是从逻辑上对系统进行调优可以做的事情有很多。例如，客户到底是关注感知性能呢还是更加关注作业真正的完成时间？对于交互式过程来说，如果让他们知道一些事情正在发生(即使只有一个进度条)，那么一般来说他们会更加满意并且认为系统速度较快。此外，只要让系统的“首个响应”——即它什么开始输出内容——更快一点，那么让整个过程完成得稍微慢一点也是值得的。在需求收集阶段应该弄清楚哪些方式是较好的。最后，在需求收集界面应该确定系统的性能需求。

提示：

很多时候，笔者见过很多开发人员认为“足够快”的系统对用户来说其性能是无法接受的。导致这种情况发生的原因有很多，不过最常见的原因肯定是开发人员在逃避现实。

找到用户期待的东西！还要记住在类似真实的硬件上进行实际的负载——不是一两个开发人员坐在他们的开发系统前面进行测试所产生的负载——测试以确定是否达到了预期的性能。

显然，在设计阶段还要考虑性能问题。如果在设计时考虑了性能，那么一般来讲在系统完成之后性能调优所需的工作量将会大大减少。此外，所能达到的“最佳”数字也有可能被极大地增强。

这里需要再强调一下，性能考虑永远不会停止——在真正开始编码，并且代码能够正常工作后便停下来！停下来看看代码。一旦整个系统集成起来后，几乎不用再看实际的代码了，除非：

- 有些地方出问题了(存在 bug)。
- 需要升级系统的一部分。
- 存在明显的性能问题(通常是非常糟糕的问题)。

在前两种情况中，可能不需要关心性能问题，只需要关心如何修正 bug 或者添加额外的功能。这里想要说的是花额外的几分钟时间看一下自己的代码，然后问自己“能够做得更好一点吗？”或者“嗨，我在这里做了什么愚蠢的事情吗？”，然后可以在这里或那里进行一些修改，并且有时候还可以在其他地方进行较多的修改。

提示:

简单地说: 我犯了愚蠢的错误, 你也会犯这些错误。但是, 如果回过头来花一两分钟以挑剔的眼光看一看自己的代码, 并且说“嗨, 真不敢相信我会这样做”, 你会对这种事情发生的频率感到惊讶的。希望那种情形是很少发生的, 但是如果花时间对代码进行审查, 那么将会发现大多数可能会导致系统性能下降的关键问题。至于那些没有发现的问题, 下一章将会解决这类问题!

下一个较大的测试里程碑时间点是在质量保证阶段。在这个时刻应该建立常规的系统基准并将它同在需求阶段建立的性能需求进行比较。

最后, 但并不是最不重要的——永不停止。询问一下最终用户, 从性能角度来看, 他们感到最痛苦的事情是什么。他们说哪些东西比较慢吗? 不要等他们告诉你(通常, 他们会认为“这本来就是这样的”并且不会对你说什么——当然, 除了你老板), 直接去问。

20.2 选择索引

同样, 之前已经深入讨论过这个主题了, 但是该主题仍然值得一提, 因为它对查询性能的作用很重要。

在索引方面, 人们倾向于走极端——笔者建议不要遵循任何规则, 但是需要考虑选择的索引影响到的所有项目。

任何拥有一个主键(除了极少数的情况, 所有表都应该有一个主键)的表至少有一个索引。但是从性能的角度来看, 这并不意味着它是一个有用的索引。应该考虑为那些经常作为 WHERE 或者 JOIN, 或者程度更低一点的 ORDER BY 子句的目标的列创建索引。

但是还需要记住的是索引越多, 插入、更新和删除操作的执行速度将越慢。当修改一个记录时(在执行插入或删除操作时肯定需要修改索引, 在更新任何参与索引的列时需要修改索引)可能需要为那个索引修改一个或多个条目(取决于对 B 树的非叶子节点所做的操作)。这意味着索引越多, SQL Server 在修改语句上所做的工作就越多。在联机事务处理(OLTP)环境中(在这种环境中会存在大量的插入、更新和删除操作), 这可能是致命的。在联机分析处理(OLAP)环境中, 这可能并不是一个大问题, 因为 OLAP 数据通常比较稳定(插入操作较少), 同时插入操作通常是通过高度重复的批处理过程来实现的(不像用户操作那样缺乏可预测性)。

注意:

从技术角度来讲, 更新和删除的问题就小多了。对于更新操作, 只有当被修改的列是索引的键的一部分时才需要更新索引。如果确实想要更新索引, 那么可以把这个操作看成一个删除操作和一个插入操作——这意味着又要面对页面拆分的问题了。

那么删除操作又如何呢? 同样, 当删除一个记录时需要从索引中删除所有的条目, 因此这将带来额外的开销, 但是不需要担心页面拆分问题, 也不需要物理地移动数据。

这里的底线是如果查询操作要大大多于修改操作, 那么创建更多的索引是没有问题的。但是, 如果需要对数据进行很多的修改, 那么只为使用频率较高的列建立索引。

如果读者更多的把本书看成一本参考手册, 而不是一本完整的“学习如何做”的书, 并且还没有花时间阅读介绍索引的章节(第 6 章和第 7 章)——去阅读吧!

检查数据库引擎优化顾问中的索引优化工具

数据库引擎优化顾问是从版本 7.0 中首次出现的 Index Tuning Wizard 演化而来的。虽然数据库引擎优化顾问已经包含了除索引优化之外的更多功能，但是这仍然是它的一个重要功能。

在使用索引自动优化工具时要十分小心谨慎。特别是要注意它要删除的索引。它是根据已经暴露给它的工作负载来给出建议的——那个工作负载可能还没有包含系统中所有的查询。看一下它的建议并问自己为什么那些建议可能是有帮助的。特别是删除索引，问一下自己那个索引是用来做什么的——删除它合理吗？在抓取工作负载文件时是否有哪些运行时间较长并且可能会用到那个索引的报表任务没有被运行呢？

20.3 客户端和服务端处理的对比

决定把工作放在哪一端来处理对整个系统的性能有着非常重要的影响——更好或者更坏。

在客户端/服务端计算首次出现的时候，大家都认为把计算“分布”会变得更快速/更便宜。对于某些任务来讲，情况确实是这样的。但是对于其他任务来讲，投入可能要比产出大很多。

表 20-1 快速回顾一下一些首选项以及它们在客户端和服务端的表现：

表 20-1

静态游标	通常在客户端比较好。由于数据不会变动，因此可以把它们打成包一次性发送给客户端——从而限制了数据往返的次数和对网络的影响。而当生成的游标的唯一目的是用于修改其他记录时则是一个明显的例外。在这种情况下应该把所有的处理放在服务端(很有可能以存储过程的形式完成)——同样消除了往返的次数
只进游标，只读游标	同样放在客户端。ODBC 和其他的库可以以一种特别的方式来使用 FAST_FORWARD 类型的游标以获取最佳性能。只需要让服务器把记录发到客户端游标中，然后它就可以做别的事情了
HOLDLOCK 情形	大多数事务工作在服务端做比在客户端做要好得多
需要工作表的进程	这又是一种在把记录移动到客户端之前需要完成产品的创建工作的情形。如果在数据可用之前把所有的数据都保存在服务端，那么可以最小化服务端与客户端之间的往返次数并提升性能
最小化客户端安装	好吧，这不属于“性能”的范畴，但是它是一个重要的成本因素。如果想要把客户端安装次数最小化，那么尽可能把业务逻辑移出客户端。可以在存储过程中执行业务逻辑或者使用 .NET 提供的基于组件的开发。在理想情况下会把“数据逻辑”(只为解决如何获取最终数据而存在的逻辑)放在存储过程中，把“业务逻辑”放在组件中
重要的筛选和/或重新排序	使用 ADO.NET 或 LINQ。它们有一组工具用于一次性从服务端接收数据(来回次数减少了！)，然后在本地进行筛选和排序。如果想要 SQL Server 以不同的方式对数据进行筛选或排序，那么它将使用新标准运行一个全新的查询。很明显，这里的开销代价是相当昂贵的。ADO.NET 和 LINQ 也内置了一些酷的功能允许在客户端联合不同的数据集(包括异构数据集)。 然而，对于超大型的结果集来讲，客户端计算机可能没有那么多资源来有效地进行筛选和排序——可能不得不回到服务端

实际上这里只是介绍了一些皮毛。需要记住重要的事情是即使在千兆以太网的时代,数据往返仍然是致命的(记住与原始的带宽相比,连接的开销往往更容易成为问题)。这里需要做的是使往返移动的数据量最小——并且只移动一次。通常,这意味着如果可能的话,需要在服务端对数据库做尽可能多的预处理,然后把整个结果发送到客户端。

还需要记住要确保客户端能够处理给它的数据。通常服务端会配备较好的设备以应付大型查询的资源需求。同样,还需要记住服务端可能会处理多个用户的查询请求——这意味着服务端需要具有足够的资源来为这些用户存储所有服务端的活动。如果一个进程太大以至于客户端无法处理它,那么出于资源的原因可以把它移到服务端,但是也需要记住如果多个客户端同时使用那个进程也可能会耗尽服务端的资源。最好是尽可能把结果集和过程的大小保持为最小。

注意:

数据“客户端”有多种可能的含义。从数据连接的角度来讲,客户端可能不是用户所处的位置。如果是一个基于浏览器的应用程序,那么实际处理数据的客户端很有可能是 Web 服务器。尽管 Web 服务器可能会部署在高端硬件上,但是它还是可能会在同一时刻处理多个这样的查询(多个大型数据集),因此需要进行相应的规划。

20.4 策略上的反规范化

这也被称为“死守规则会害死你”。在一个 OLTP 环境中,规范化的数据对数据完整性和性能有益。这里的问题是 OLTP 数据库中的东西并不一定都与事务处理相关。即使是 OLTP 也会做一些报表工作(如对每天输入的事务的小结)。

通常,只需要在表中添加一个额外的列就能够避免大型的连接,或者更糟的涉及几个表的连接的情形。笔者见过这样一个情形,只要添加一个列就能够把 9 张表的连接变为 2 张表的连接。其中的差别是 100 000 条记录和几百万条记录之间的差别。这个更改使得查询的性能有了很大的改变,从几分钟下降到了只有几秒钟。

然而,就像大多数事情一样,这不是应该丢弃的东西。基于某种原因,规范化是实现大多数事情的方式。它对数据完整性有很大的帮助,同时在很多情形中对性能有很大的正面影响。不要为了反规范化而反规范化。要确切地知道需要达到的目标,并且进行测试以确保达到了预期的效果。如果没有达到预期的目标,那么回过头去采用原来的方式完成任务。

20.5 合理组织存储过程

这里将不会从外部讨论(命名约定等等是重要的,但是这里不准备讨论它们),而是站在“它们如何运作”的立场来讨论问题。下面几个小节将会讨论这些主题。

20.5.1 保持事务短小

长的事务不仅可能会导致死锁,也可能会引起基本阻塞(其他人的进程必须要等待你,因为你还没有完成与锁相关的操作)。任何时候,一旦进程被阻塞——即使它最终在阻塞事务结束之后可以继续——还是会产生延迟,从而影响了被阻塞的过程的性能。没有什么比让进程停止并等待更

能直接影响性能了。

20.5.2 尽可能使用限制性最少的事务隔离级别

锁持有得越紧，那么阻塞其他进程的可能性就越大。需要确保那些锁确实是必需的以确保数据完整性——不要尝试持有比这更多的锁。

更多关于隔离级别的信息请参考第 11 章中的事务和锁。

20.5.3 必要时部署多个解决方案

这里的一个例子是一个搜索查询，它接收多个参数，但是并不需要全部这些参数。一个可能的方案是编写存储过程使它只使用一个查询，而不管实际上接收了多少个参数——一种“全能”的方法。从开发的角度来看，这确实能够节省时间，但是从性能的角度来看，这种做法实在是致命的。这很有可能意味着每次运行这个存储过程都需要连接几张并不需要的表！

这里需要做的事情是添加一些 IF...ELSE 语句来检查事物。这更像是一种“三思而后行”的方法。这意味着需要编写多个查询来处理提供的参数的每种可能的组合，但是一旦第一个编写完毕之后，其他的通常可以通过复制第一个并做相应的修改来完成。

提示：

这里编写的代码量大才是真正的问题。开发人员都是善变的群体。他们通常只喜欢做有趣的事情。如果让他们去做前面的例子，那么他们可能很快就会对不断编写那些非常类似的查询以处理参数的细微不同而感到厌烦。

对此我只能说——是的，不是每件事情都很有趣，否则所有人都想成为软件开发了！有时候为了完成产品，只能笑笑并忍受它。

20.5.4 尽可能避免使用游标

如果读者是一个来自 ISAM 或 VSAM 环境(这些是旧式的数据库存储方法)的编程人员，那么在做事情的时候可能会自然而然地使用游标。毕竟，游标过程的工作方式更像读者在那些环境下所熟悉的方式(这样的循环结构在很多非数据库处理结构中也是很常见的)。

不要这样做！

几乎所有的一开始想到要用游标完成的事情实际上都可以使用一组操作来完成。尽管有时候需要相当仔细地思考，但是通常总是能够完成的。

举个例子，几年前笔者被要求将一个多行基于游标的操作尽可能转化为单个语句。现有的过程差不多要花 20 分钟才能运行结束。这个运行时间绝对是有问题的，但是客户并不把这看成是影响性能的原因(他们已经接受了该过程要运行那么长时间)。相反，他们只是尝试简化代码。

他们有一个大型的产品数据库，并想要根据成本自动为可用的产品定价。如果涨价幅度是一个普通的百分数(比如 10%)，那么 UPDATE 语句就会非常简单——像下面这样：

```
UPDATE Products  
SET UnitPrice = UnitCost * 1.1
```

这里的问题是涨价幅度并不是直接已知的——要用一个逻辑模式在求出涨价幅度。这个逻辑

看起来像下面这样：

- 如果在涨价之后，产品价格中的美分值大于或等于 0.50，那么就把价格设置为 0.95。
- 如果美分值小于 0.50，那么就把价格设置为 0.49。

使用游标完成这个逻辑的伪代码如下所述：

```
Declare and open the cursor
Fetch the first record
Begin Loop Until the end of the result set
Multiply cost * 1.1
If result has cents of < .50
    Change cents to .49
Else
    Change cents to .95
Loop
```

当然，这是一个极其简化的版本。实际上完成这个逻辑需要大概 30 到 40 行代码。相反，如果把它修改为围绕单个相关的子查询(内嵌了 CASE 语句)来工作，那么运行时间将会降低到大约 12 秒。

当然，这里的观点是尽可能合理地消除对游标的使用，这样不仅能大大降低复杂性(也是最初的目的)，也有利于性能的提高。

20.6 使用临时表

使用临时表有时候能帮助提高性能——通常可以消除游标或允许根据需要在数据上建立索引。

20.6.1 使用临时表分解复杂问题

正如前面看到的那样，游标的存在通常是一个祸害。有时候使用临时表可以将一个操作转化为两组或多组操作来消除游标的使用。一个初始查询创建了工作数据，然后另一个进程可以使用这些工作数据。

实际上也可以使用上一节中提到的定价示例来说明临时表的概念。尽管这个解决方案并不比相关子查询好，但是它仍然是可以工作的，并且比采用游标的方法要快很多。其步骤如下所示：

```
SELECT ProductID, FLOOR(UnitCost * 1.1) ÷ .49 AS TempUnitPrice
    INTO #WorkingData
    FROM Products
    WHERE (UnitCost * 1.1) - FLOOR(UnitCost * 1.1) < .50
INSERT INTO #WorkingData
SELECT ProductID, FLOOR(UnitCost * 1.1) ÷ .95 AS TempUnitPrice
    FROM Products
    WHERE (UnitCost * 1.1) - FLOOR(UnitCost * 1.1) >= .50
UPDATE p
    SET p.UnitPrice = t.TempUnitPrice
    FROM Product p
    JOIN #WorkingData t
        ON p.ProductID = t.ProductID
```


使用这种方法只需要三步就能解决问题，而不需要三十或四十行代码。尽管这种方式没有使用相关子查询那样快，但是比起使用游标的解决方案，这种方式足以让人惊叹了。

当遇到复杂的问题并且认为需要使用游标时，请记住还有使用临时表将整个过程分为几个小步骤来完成任务的方法。要尽量避免不由自主地使用这种方法——在选择使用这个方法之前找找有没有单个语句查询的方法——但是如果其他方法都失败了，那么与使用游标的方案相比，这种方法确实能够省下不少时间。

20.6.2 使用临时表以允许在工作数据上创建索引

我们常常会碰到一个过程需要很多不同的操作来完成，而这些操作使用的基本上是同样的数据。当运行不同类型的更新(可能是在完全不同的表上进行更新)，但是需要使用同一个源数据来找出哪些需要修改或者需要修改为什么值时便是这样一种情形。笔者已经见过很多同一数据被复用(在同一个过程中)上百次甚至上千次的场景。

在这种“复用”情形中可以考虑只进行一次数据查询，然后把数据放到临时表中。还可以考虑在这些数据上为用到这些数据的查询建立所需的索引。

提示：

即使是只使用两次的的数据，不管是出于什么原因，如果原来的在源数据上的查询不是那么高效，那么使用临时表会带来巨大的差异。有时候这是因为源数据上没有合适的索引，但是在大多数情况下，这是因为需要在大型数据集上进行多表连接。把这些数据放入临时表通常允许在整个过程的早期显式地对大型数据集进行筛选。同样，不要自然而然地使用这种方法，但是记住有这么一种方法。

20.7 及时更新代码

你还仍然支持 SQL Server 2000 吗？7.0 呢？好，现在应该明确不支持 7.0 了，甚至对 2000 的支持也应该丢弃(或者至少也到了丢弃它的最后阶段了)。因此，既然不再支持那些旧版本了，那为什么系统代码和设计看起来仍然在支持它们呢？

好，好，我明白这并没有那么简单，但是在应用程序的每个发布中需要确保分配一段时间(建议 10%-25%)用于提升现有的性能和功能。如果只需要支持 SQL Server 2008，那么寻找一下有没有解决某些问题的特殊代码，SQL Server 2008 可能天生就已经解决了这些问题，如：

- 对于处理到特定表中的 INSERT、UPDATE 和 DELETE 场景的过程或代码流来讲，现在可以使用新的 MERGE 命令，这样只需要遍历一次数据就能够完成这三个修改操作。它还拥有单个语句所具有的优点，这意味着可以不用为三个独立的语句显式地定义事务了。
- 特殊的层次处理：现在 SQL Server 为常见的事物建立了本地结构。其功能不仅仅包括层次特定的功能(如剪枝和嫁接)，还包括垂直和水平索引的功能(非常酷的功能！)。
- 日期和时间数据类型的处理。

20.8 注意细节问题

在所有改善性能的编程中，一个常见的错误是忽略小事。每当尝试挤出大量的性能时，很自然地思路是要调整那些长时间运行的任务。

是的，在长时间运行的进程上确实存在最大的机会获得大的一次性性能收益。但是遗憾的是，它常常会让人们忘记他们真正感兴趣的是节省下来的总时间——即进程真正花费的时间。

虽然在一个查询中的一个变更通常能够使得查询所需的时间从几分钟变成几秒钟，这绝对是真的(笔者曾经确实见过通过对索引和查询进行优化就能够把查询的运行时间从几天减少为只需要几秒钟)，但是对应用程序来说，最大的收益往往来自对那些看起来已经够快的查询做进一步的优化。它们通常是在循环中被反复运行的函数或项目。

稍微思考一下。假设现在一个查询需要花费 3 秒钟，同时每次订单接收器查看可能的销售时都需要用到这个查询——比如每天查询 5000 个项目。现在想象一下如果能节省一秒钟的查询时间情况会怎样呢。那样就能省下 5000 秒，或者超过 1 小时 20 分钟！

20.9 硬件考虑事项

如果觉得这里的内容有点枯燥，请原谅我——我已经尽量让它保持有趣了，如果你是一个普通的开发者，那么可能已经足够了解了这些内容，因此觉得这些内容非常枯燥，但是可能还没有足够了解到使你可以避免一些问题。

在过去的几年里，硬件的价格已经一落千丈了——但是经理或客户可能对硬件购买做出了预算。在为硬件进行预算时，需要记住：

- 一旦已经部署了系统，硬件是用来保持数据安全的——这些数据值多少钱呢？
- 一旦已经部署了系统，可能会有很多用户——如果创建了一个公共网站，那么可能一天 24 小时都有成千上万的用户在系统中活动。如果服务器不可用——或者更糟——丢失了一些数据，那么在生产率损失、错失生意、丢面子以及普通的信用丧失方面将付出什么代价呢？
- 维护系统的花费将很快就会超过系统本身所需的花费。早期在功能强大的主流系统上花费的金钱可能会在长时间运行中省下大量的开支。

在决定向谁购买以及购买哪些特定设备的问题上需要考虑多个方面。这时需要暂时忘掉预算，问自己一些问题，其中一些包括：

- 要购买的计算机是专门用作数据库服务器的吗？
- 系统上的活动是处理器密集型还是 I/O 密集型的呢(对于数据库来讲，几乎总是属于后者，但是也有例外)？
- 是否决定在计算机上运行多个生产数据库呢？如果是，那么其他数据库是否属于不同的类型(OLTP 与 OLAP)？
- 服务器是否位于本地，或者是否需要旅行到另外一个地方进行维护？
- 系统宕机的风险是什么？
- 丢失数据的风险是什么？
- 性能是“一切”吗？

- 当 O/S 和支持系统升级时，能够指望什么类型的长期驱动程序支持？

同样，这里也只是介绍了一点皮毛——但是已经有了一个好的开端。下面分别介绍这些问题的含义。

20.9.1 独占式使用服务器

对于这个问题，笔者认为不需要什么高深的学问就能得出答案。在大多数情况下，让 SQL Server 硬件只专用于 SQL Server 并将其他应用程序部署到完全分离的系统上是最好的方式。然而，事情并不总是这样的。

如果正在运行一个规模相对较小和简单，同时需要与其他子系统(比如 IIS 作为 Web 服务器)协作的应用程序，那么把它们放在一台计算机上比较好，也有益于性能的提升。为什么呢？如果在两个子系统之间需要对大量的数据进行来回传输(SQL Server 中的数据库和 Web 页面或其他独立的进程)，那么比起网络到引起的瓶颈，内存空间之间的通信要快得多——即使是在相对专用的骨干网络环境中。

但是，记住这是一个例外，并不是一个规则。采用这种方式工作得最好的场景通常满足下列条件：

- 系统之间交互程度高。
- 除了相互之间的交互之外，系统要做的事情很少(系统的主要活动就是所有引起交互的活动)。
- 两个进程中只有一个是 CPU 密集型的，而另一个是 I/O 密集型的。

如果存在疑问，那么就考虑使用常规的方式把处理分离到两个或者更多的系统中。

20.9.2 I/O 密集与 CPU 密集的对比

笔者可以听到你们都在喊“两个都要！”如果确实是这种情况，那么希望你有一笔非常大的预算——但是还是要讨论那种情况的。假设你还没有安装系统，这里只是纸上谈兵。如果在 SQL Server 中做的所有事情几乎都是基于数据的，那么肯定需要一定程度的 I/O 密集。而 CPU 的负担则要取决于所运行的查询的类型，其变化非常大，如表 20-2 所示：

表 20-2

低 CPU 负荷	高 CPU 负荷
简单的单表查询和更新	大型连接
在较小的表上进行联合查询	聚合(SUM、AVG 等等)，对大型结果集进行排序

有了这些印象之后，下面更深入介绍每一种情况。

1. I/O 密集

I/O 密集型任务会导致将预算更多地集中在驱动器阵列上，而不是 CPU 上。注意这里是说驱动器“阵列”——并不是说它是一种选择。在这个问题上，恕我直言，如果在数据库存储机制上没有进行某种类型的冗余安排，那么你肯定是没有脑子。任何值得保存的数据都值得对其进行保

护——稍后将会讨论这些选项。

在讨论 I/O 的选项之前先简要看看什么是 I/O 密集型。简而言之，这意味着需要对大量的数据进行检索，但是在系统中运行的处理几乎都是查询(不是复杂的业务过程)和那些不包括需要进行大量计算的更新。记住——从移动数据的角度来看，硬盘驱动器很有可能是系统中最慢的东西(除了 CD-ROM 之外)。

RAID 简介

RAID，它带来了野蛮部落给群众引来恐惧的画面。实际上，大多数的 RAID 级别都是某种故障安全机制，用以防止一种叫做“数据丢失”的野蛮攻击。如果你不是一个 RAID 迷，那么你可能会惊讶并不是所有的 RAID 级别都能够保护数据以防丢失。

RAID 一开始表示由廉价的磁盘组成的冗余阵列。其想法是相当简单的——在那个时候使用很多小磁盘要比使用一个大磁盘便宜。此外，磁盘阵列意味着有多个驱动器磁头在工作，因此也可以构建(如果需要的话)冗余。

由于驱动器价格下降的如此厉害(笔者一直在猜测，但是我敢打赌现在驱动器的价格(每兆字节的美元数)一定远远小于术语 RAID 被创建时的价格的 1%)，笔者已经听到了关于 RAID 的其他含义。最常见的是独立磁盘随机阵列(Random Array of Independent Disks)(就我看来，这个含义有点自相矛盾)和单个磁盘随机阵列(Random Array of Individual Disks)(这个含义不算太坏)。不管它是什么的缩写，需要记住的事情是有两个或多个驱动器在一起工作——通常为了达到在性能和安全性之间取得平衡的目的。

可以从很多地方获得 RAID 有关的信息，表 20-3 介绍一下最常见的三种 RAID 级别(好吧，如果认为一种级别实际上是组合了其他两种的话，那么就四种)：

表 20-3

RAID 级 别	说 明
RAID 0	也叫做无奇偶的磁盘条带。在所介绍的三种 RAID 级别中，这种可能是读者最不可能知道的一种。与 RAID 5 一样，它要求至少三个驱动器才能工作。但是与 RAID 5 不同的是无法从中得到对数据丢失的安全性保证(奇偶是一个特殊的校验和，它允许在某些情况下重新构建丢失的数据——正如这里指出的那样，RAID 0 不带奇偶)。RAID 0 最大的用处是能够获取最大的性能并且不损失任何磁盘空间。在 RAID 0 中，数据分散在存储到阵列中所有的驱动器上(至少有 3 个)。虽然这看起来有些奇怪，但是这意味着总是有三个或多个磁盘驱动器在同时读写数据。在镜像中，所有数据都是存储在一个驱动器上的(副本被存储在一个独立的驱动器上)，这意味着只能等待那一个磁头来完成工作
RAID 1	也叫做镜像。系统中每个活跃的驱动器都有第二个驱动器“镜像”(保留一份完全一样的副本)其信息。通常两个驱动器的大小和类型都是一样的，并且会同时在两个驱动器上存储所有的信息(Windows NT 有一个基于软件的 RAID，它可以镜像任意两个卷，只要它们的大小是一样的)。在写数据时，镜像不会提升性能(仍然需要写两个驱动器)，但是根据控制器的安排，它可以让读取速度提升一倍，因为可以使用两个驱动器读取数据。镜像的优点是如果只有一个驱动器发生故障，那么另一个驱动器仍然可以提供服务，并不会造成数据丢失或性能损失(当然，如果控制器是并行读取数据的，那么读取速度将会变慢)。镜像最大的缺点是为了获取所需的磁盘空间，需要购买两倍的驱动器

(续表)

RAID 级 别	说 明
RAID 5	最常用的 RAID 级别。尽管从技术角度来讲，镜像是一种 RAID(RAID 1)，但是当人们谈起 RAID 时通常是指 RAID 5。RAID 5 的工作方式与 RAID 0 一样，但是有一个非常重要的差别——阵列中所有数据的奇偶信息被保存下来了。例如，假设有一个由 5 个驱动器组成的阵列，则对于任意给定的写入，数据将会被分开存储到所有 5 个驱动器中，但是在每个驱动器上都会留出指定的百分比(它们加起来的总和是一个驱动器的空间)的空间用以存储奇偶信息。同大家的看法相反，没有一个驱动器是奇偶驱动器。相反，一些奇偶信息被写入到所有驱动器中——只要给定字节的奇偶信息不与真正的数据存储在同一个驱动器中即可。如果一个驱动器丢失了，那么可以使用其他驱动器中的奇偶信息来重新构建丢失的数据。RAID 5 的优点是能够获得多驱动器的读取性能，缺点是损失了一个驱动器的空间(如果有一个由三个驱动器组成的阵列，那么只能看到两个驱动器的空间，如果是一个七个驱动器的阵列，那么只能看到六个驱动器的空间)。从每兆字节的价格上来看，它并不比镜像差，但是它仍然能提供非常好的性能
RAID 6	RAID 6 是 RAID 5 的扩展，一般只用在大型阵列中(为提供额外冗余所需的算法开销可以被摊开，因此每个磁盘损失的空间就少了)。与 RAID 5 相比，RAID 6 提供了额外的奇偶编码，这些额外的信息可以在多个磁盘丢失的情况下用以恢复磁盘。RAID 5 一般更加廉价，并且阵列的规模也较小，而 RAID 6 甚至在重建单个发生故障的驱动器时还维持着一定级别的冗余
RAID 10(也叫做 RAID 1+0)或 RAID 0+1	从性能和数据保护的角度来讲，RAID 10 兼具 RAID 0 和 RAID 1 的优点。但是它是这里讨论的 RAID 类型中最昂贵的一种。RAID 10 同时实现了 RAID 1(镜像)和 RAID 0(无奇偶的条带)。其最终结果是可以对条带数据进行镜像。读者可能还听过 RAID 0+1，这些是镜像数据的条带组。它们所需的驱动器数量是相同的，但是 RAID 10 在恢复场景中表现较好，因此通常会采用这种方案
RAID 50	这是通过镜像两个 RAID 5 阵列来实现的。虽然可以证明这种方式是冗余性最强的，但是它仍然存在同一阵列中的两个驱动器同时发生故障的风险。它是这里提供的选项中最昂贵的一个，一般只在最极端的环境中才使用这种方案

总的来说，对数据库安装来说，RAID 5 是事实上的最低配置。尽管如此，如果有一个宽松的预算，那么笔者还是建议将多种方法混合使用。

提示：

在大型安装中 RAID 10 已经成为了标准。但是，对于中等的购买力来说，RAID 5 可能暂时还会占据统治地位——可能当进入服务器驱动器以 TB、PB 甚至是 EB 计量的时代时，情况会有所改变。这一天肯定很快就会到来。

读者至少需要为主数据库配置一个 RAID 5，并且为日志配置一个完全分离的镜像组。采用这种方案的人通常会把 Windows 和日志放到镜像集上，把物理数据库放到 RAID 5 阵列上，但是那些预算更多一点的人通常把 O/S 和日志放在两个不同的镜像集上(数据文件仍然放在 RAID 5 阵列上)。爱追根究底的人一定想要知道为什么要这样做，下面稍微扯开一点，简要介绍一下如何读写日志数据。

与能够并行读取的数据库信息不同(这就是 RAID 5 和 RAID 10 性能更好的原因)，事务日志

是按时间顺序排列的——即需要按照一定的顺序进行读写以确保完整性。这里并不是说在一个连续流中数据的物理顺序一定要正确，而是说流中的每一件事情都需要按照一定的逻辑来完成。正因为如此，如果把日志保存到它自己的驱动器上，那么实际上它会工作的很好，因为驱头将很少从它当前读写的流中离开。采用这种方式的结果就是需要把日志和数据存储到不同的物理设备上，这样读写数据不会扰乱日志的读写。

提示：

如果把多个数据库的日志保存在同一个镜像集上，那么镜像集的顺序读写的性能将会消失(它需要在不同的日志之间来回跳动！)。

然而，日志通常不会像读取数据那样占用大量的空间。对于采用镜像的方式来说，只需要购买两个驱动器就可以提供冗余了。对于采用 RAID 5 的方式来说就需要购买三个了，但是不会看到任何由 RAID 5 的并行读取特性带来的性能收益。当综合考虑这些因素时会发现对日志或 O/S 采用 RAID 5 并没有太大的意义。

注意：

你可以拥有世界上所有的 RAID 阵列，但是从数据的长久安全性的角度来讲，它们仍然比不上一个好的备份。备份易于离线操作，并且不受机械故障影响。RAID 单元虽然冗余并且非常可靠，但是如果两个(而不是只有一个)驱动器同时发生故障，那么它也会变得毫无价值。另一个问题——如果发生了火灾怎么办呢？所有的驱动器可能都会被烧毁——同样，如果没有备份，那么将会陷入麻烦之中。第 22 章将会讨论如何备份数据库。

2. CPU 密集

在 SQL Server 服务器上，你几乎总是希望确保它拥有多个处理器(是的，即使是在多核处理器的今天)，即使是利用率相对较低的计算机。这有助于防止系统中出现那些必定会让用户发狂的短暂的“停顿”，因此将这部分事情当做是一种倾向——特别是在双核处理器的今天。记住 SQL Server 的工作组版只支持最多两个处理器——如果需要使用更多的处理器，那么需要升级到标准版(四个处理器)或者企业版(处理器的数量只受硬件和预算的限制)。

即使是只运行 SQL Server Express——它只支持一个处理器——也要尽可能使用具有双处理器的计算机。记住，除了 SQL Server 之外，系统上还要运行其他程序，因此让另一个可用的处理器执行外部操作可以减少 SQL Server 的延迟。

也许最大的问题是内存。这肯定是一个不希望缺斤短两的区域。此外，记住在多台处理器环境中(而且应该如此)将会有更多的内容同时进入内存。在这个内存廉价的时代，SQL Server 不应该安装在内存小于 512MB 的计算机上——即使是开发环境。生产服务器应该至少配置 2GB 的内存——很可能更多。

在决定使用多少内存时需要考虑的事情包括：

- 同时会有多少个用户连接(每个连接都需要占用空间)？每个连接大概需要占用 24K 的内存(通常会占用更多)。这个问题实际上并不是致命的，因为 1000 个用户只需要占用 24MB 的内存，但是仍然需要考虑这个因素。

- 是否需要进行很多的聚合和/或排序？取决于查询所涉及的数据集的规模，这可能会是致命的。
- 最大的数据库有多大？如果只有一个数据库，并且它的大小只有 1GB(实际上，大多数的数据库比人们想象的要小)，那么根据同时运行的查询数量以及它们所执行的动作，4GB 的内存没有太大的意义。
- SQL Server 2008 的工作组版只支持最多 3GB 的内存寻址。如果需要更多的内存，那么至少需要使用标准版。

此外，一旦处于操作之中——或者在完全填充的测试系统运行时——需要在性能监视器中查询一下缓存的命中率。在第 21 章中会介绍如何计算这个数字。现在只需要知道它是一种测量值，用以说明成功地从内存中(而不是磁盘中)取出数据的频度是什么(内存的速度要比磁盘的速度快很多很多)。低缓存命中率通常表明需要更多的内存。但是记住高缓存命中率并不意味着不应该添加更多的内存。SQL Server 的预先读特性可能会创建人为的高缓存命中率，这会掩盖对额外内存的需求。

20.9.3 OLTP 和 OLAP 的对比

这两个系统之间的需求通常是相互不一致的。第 24 章将会讨论一些设计差异，因此希望读者能够有一个设计考虑差异的概念。

在任何情况下，“从硬件的角度”的建议都将保持简短：

注意：

如果需要运行数据库以支持两种类型的需求，那么把它们运行在不同的服务器上——就是这么简单。

再怎么强调把它们两个分开都不过分。一个大型数据仓库的导入、导出或者甚至是一个大型的报表运行都能够在 OLTP 过程和/或数据缓存中引起重大的翻转，从而对可能拥有很多用户的系统的性能造成极大的影响(因此又是一大笔开销)。

20.9.4 现场和非现场的对比

以前所有基于 SQL Server 的事情都将在现场进行，同时现场有专门的人员负责照看和维护。如果系统宕机了，人们会等在那里进行重新加载并排除故障。

在 Internet 的时代，很多安装都是通过 Internet 服务提供商(ISP)来协同定位的。ISP 会负责确保整个系统已经被备份了——他们甚至可以根据你的指示进行还原——但是他们不负责修改代码。当在系统中碰到灾难性的错误时，这会是一个很困难的问题。虽然总是可以通过远程连接在上面工作，但是可能会碰到一些配置和性能问题，包括：

- **安全性**——对你开放的远程访问意味着从某种程度上它对其他人也是开放的，而你可能不希望那些人能够访问系统。关于这个问题，笔者有两个稍有价值的看法，它们分别是确保有非常紧密的路由和在合适的地方设置端口限制。对于那些不那么精通网络的人(包括笔者)来讲，这意味着需要限制哪些 IP 地址允许被路由到远程服务器、有哪些可用的端口甚至是哪些协议(SSL 和非 SSL)是允许通过的。

- **性能**——读者可能已经习惯了家庭和办公室的 100Mbps 到 1Gbps 的网络速度。现在是通过 Internet 上的虚拟专用网(VPN)或者更糟的拨号来通信的,这让人开始讨厌生活了(太慢了!)。
- **响应**——当运行某些电子商务站点或者其他站点时却又无法让 ISP 的人接听你的电话,或者他们说马上就好,但是几个小时之后系统还没有恢复,这个时候会让人有点心烦。确保仔细调查远程托管公司——不要认为他们在生意做完之后还把你当成重要的客户。
- **硬件维护**——很多共同托管设施不会为你做硬件工作。如果一个硬件故障靠重新加载无法解决,那么可能必须要亲自到那个站点或者叫第三方来进行维护——这意味着应用程序可能要离线几个小时或者可能是几天。

如果是一家小型机构需要在 Internet 上做这样的事情,那么实际上非现场的方式更具可取之处。它虽然比较昂贵,但是通常会获得很多带宽,并且有人会确保确实已经做了备份——确保真正对 ISP 进行了核查。他们中的很多人对 SQL Server 一无所知,因此确保他们具有所需的专业技能。

提示:

最近在主要 ISP 之间形成的一个趋势是把主要的托管设备放在比你一开始预期的还要远很多的地方。通常这样做是为了能够方便地访问水源(用于冷却)、廉价的电力或者两者都有(普遍的做法是靠近水力电气设备)。在很多情况下,这样做并不会有什么问题,但是如果使用了第三方硬件支持公司则需要考虑一下了——这个支持公司是否有合适的人员常驻在托管设备的附近?

如果托管公司位于主要的大城市中,那么就有理由认为该托管公司拥有大量的支持人员,他们能够在 30 到 60 分钟之内赶到 ISP 所在的位置。但是如果 ISP 位于“俄勒冈州的波特兰市的外面”,那么就需要确保这里的“外面”不是 60 或 80 英里之外。如果确实是这样,那么检查一下支持公司到底有多少人员确实在 ISP 所在位置的附近。

20.9.5 宕机的风险

能够承受的宕机时间和频率是多少?这看起来可能像是一个愚蠢的问题。当笔者问这个问题的时候,通常会得到怀疑的目光。对于某些安装来说,其答案是显然的——它们无法承受宕机,哪怕只是一段时间。但是这个数量并不像看上去的那么高。要知道,只有那些急性医疗应用程序或同安全性操作密切相关的程序才是真正性命攸关的应用程序。其他安装损失的则可能只是金钱——它们的宕机甚至可能会导致破产——但是不会影响生命。

这就是说,它实际上不是那种黑白分明的事情。宕机的紧急程度实际上是一个连续渐变的事物。它的范围从前面提到的高端医疗应用程序到低端的在遗留系统上进行数据挖掘操作(通常——对于一些公司来说,这可能是它们所拥有的全部)。几乎所有人都同意的事情是每一个系统的宕机都是极其不受欢迎的。

因此,问题就变成了这种不欢迎程度到底如何?如何去量化它呢?

假设现在有一帮极其善于计算的人(我完全可以这样说,因为我一度就是这样的人)在为你工作,那么要不了多少时间你就会知道有很多指标可以用来度量宕机的代价。例如,假设有一帮员工坐在一起说在系统恢复之前他们无法工作,那么从生产效率的角度来看,受影响的雇员数量乘以他们每个小时的价值(记住,一个雇员的价值比他或她的工资要高)等于系统宕机的代价。但是,

等等，还会更多。如果运行的是联机销售——因为无法及时对客户做出适当的响应会错过多少生意呢？噢——更大的损失。如果与系统一起运行的是一个工厂，那么因为系统宕机会有多少货物无法生产出来——或者，即使仍然能够生产它们，那么是否可能会因为失去质量保证或其他信息而带来损失呢？

现在读者应该能够看到并让你的老板相信宕机时间是非常昂贵的——究竟有多昂贵取决于具体的情况。现在要做的事情是确定究竟愿意花多少钱来确保不会发生这样的事情。

20.9.6 丢失数据

可能无法对此进行度量。在某些情况下，可以用重新构建数据所需的代价来对此进行量化。但是有时候根本就无法重新构建数据，因此可能永远都无法知道丢失数据所带来的损失有多少。

同样，想要花费多少来防止数据丢失会影响到冗余系统的预算和备份磁带驱动器以及非现场的存档服务等。

20.9.7 性能就是全部吗

这个问题的答案多半是否定的。性能很重要，但是到底有多重要则要看它所带来的回报有多少。例如，如果购买额外的 10% 的 CPU 处理能力可以为每个事务节省 2 秒钟——如果有 50 个数据录入员每天竭尽所能地进行录入，那么其回报是相当可观的。在一天的过程中，看起来很小的时间节省是可以累加起来的。如果 50 个录入员每人每天处理 500 个事务，那么每个事务节省 2 秒钟之后总的节省时间累加起来将超过 13 小时(这超过了一个人一整天的工作量！)。节省下这段时间可能允许推迟增加雇员，节省下来的薪水可能轻易地就抵消了额外处理能力上的花费。

然而，隔壁公司可能会以不同的眼光来看待这种问题——他们可能只有一个或两个雇员。此外，他们的工作过程可能需要花费很长时间来填写表——实际存储这些信息的事务并不是一个大问题。在这种情况下，可能不值得花费额外的钱来购买额外的速度。

20.9.8 厂商支持

这里直接从关键的主题开始——笔者无论如何也不会推荐你向某些像“Bob's Pretty Fine Computers”之类的公司购买服务器以节省一点钱(甚至是一大笔钱)。还记得那些风险吗？现在，试试引入硬件和驱动器集合的奇怪混合。现在想象一下当出现问题时——很快就会发现所有这些公司都在相互指责，说“是他们的错！”。你真的希望被夹在中间吗？

你需要的是真正经过考验和测试的知名厂商。服务器——特别是数据服务器——要坚持使用众所周知的、受人信任的品牌。笔者并不是在鼓吹任何人(本书中没有广告！)，而是在讲那些主流的厂商，如 Dell、IBM 和 HP 等等。注意，这里将众所周知和受人信任的品牌是指在服务器领域中众所周知的品牌。某些人一年能卖出十亿台桌上电脑并不意味着他就了解服务器——这几乎就像是苹果和桔子。它们之间有着很大的差别。

使用了知名厂商的设备之后，除了能够确保在发生故障时能够得到恰当的支持之外，还意味着在将来设备更有可能得到持续的升级。O/S 的每个新版本都只明确支持那么多的设备——需要确保你的设备在它的支持之列。

20.9.9 理想的系统

这里首先要说明的是不存在理想的系统。话虽如此，但是就笔者和大多数被称为“专家”的其他人来看，如果预算允许的话，有一个常规配置是几乎可以满足全部需求的。这里所讨论的是驱动器阵列(相对于预算和设置来说，CPU 和内存是无足轻重的)。

这里所需要使用的是混合镜像和 RAID 5 或 10。把 O/S 和日志放在镜像驱动器上(理想情况下是放在不同的镜像集上)。把数据放在 RAID 5/10 阵列上。这样 O/S 和日志——它们都倾向于做很多顺序操作——就都有了属于它们自己的驱动器，从而不会被真正数据的读写操作所干扰。数据拥有多个磁头进行读写以提供最大的性能，同时还维持着一定级别的冗余。

20.10 小结

性能本身是能够(也应该)用一本书来讲述的(实际上 Wrox 出版了很多关于这个主题的书籍)。它所包含的内容有很多，以至于无法在一章或甚至几个章节内介绍所有内容并使读者熟知这些内容。笔者尝试解决这个问题的方式是指出贯穿在本书中的性能问题，这样读者就可以一次阅读其中一部分。本章是同一问题——设计(在性能成为问题之前解决它)——的两个不同方向中的第一个方向。下一个章节将会介绍在系统已经运行之后如何确定并解决性能问题。需要指出的是可以在测试中使用下一章中讨论的技术，这样就可以根据测试的结果来相应地修改设计了。



第21章

性能优化工具

好吧，如果只需要开发软件，接着对它进行收费，然后就可以忘记它…，那这真是太好了。是的，好吧…现在可以不要做梦了——事情不是那样进行的。

从某种角度来讲，通常把软件看成是成功的开发项目的一部分，而任何软件最终都是要在一些用户面前呈现自己的。即使它只是一个原型，也需要分析这个原型是如何满足最初目标的。评价是否达到了目标的一部分是看一下性能并且问自己哪些方面能够做得更好。

在前面一章中曾建议有关性能优化需要明白的最重要的事情是不可能做到全知全能。如果要为“需要明白的最重要的事情”寻找一个对等的看法，那性能优化实际上是永远做不完的。系统中的内容会发生变化，服务器的状态会发生变化，对系统的使用会发生变化。简而言之，整个系统会发生变化，而这种变化会影响性能。这里的难题是要了解哪些部分工作的不是很好，哪些部分工作得很好以及哪些部分已经工作得“足够好了”。

就像前一章中所做的那样，本章将按照已经介绍的主题来介绍性能优化。这里讨论的所有内容都是与性能有关的，但是这次将更加注重找出哪些部分影响了性能。如果读者已经完成了设计和开发的工作，那么应该已经有了一个优秀的设计，但是软件的现实是设计需求很少能够完全匹配一个实际运行中的系统所面对的真实情况。因此，本章将介绍如何了解系统中已经发生的事情并且确定哪些部分能够做得更好。本章涉及的主题包括：

- 例程维护
- 硬件配置问题
- SQL Server Profiler
- 数据收集器

21.1 优化时机(第二部分)

前面一章中也有一节以这个名字命名——何时进行优化。如果读者足够细心，那么应该知道这个过程在本章中讨论的“处于测试或生产”模式之前开始。也就是说本章中该问题的新答案是简单的“定期进行”。不要等到用户向你大喊抱怨——相反，要规划一个定期的优化过程。

很多人都认为发布之后大部分的维护工作是 DBA 的职责，笔者并不打算就此问题而争论，但是基于那样的哲学，请考虑下面几个问题：

- 你正在生产一个供很多人使用的产品(你想要每个客户的 DBA 独自去解决你抛给他们的问题吗?)。
- 如果没有 DBA 怎么办呢? 根据安装情况, 可能没有哪个职员是 DBA, 因此你的系统和/或建议将会采取什么措施来防止最终用户陷入麻烦呢?
- 如果你是 DBA, 那怎么办呢?

这些问题都被简化过了, 但是这里真正的关键点是即使在产品发布并运行之后也需要考虑性能问题。不管是为了下一个版本能够做得更好还是仅仅为了博取客户的欢心, 都应该不断寻找问题(最好是在客户之前了解这些问题)或者简单的方式使得系统工作得更好。

21.2 日常维护

笔者讨厌好的系统变坏, 但是这种情况却经常发生。通常在人们购买或构建系统, 然后让它们运行, 接着就忘记它们时会发生这种情况。

维护对性能方面的关注程度不亚于对系统完整性的关注程度。如查询计划已经过时了, 索引页面满了(因此有很多页面分裂), 存在碎片了, 最佳的索引需要根据使用情况修改, 表中的数据量发生了变化等问题。

只要查看新闻组、与那些有旧系统运行的人交流或访问 Web 上的 SQL Server 支持站点, 读者就会反复听到同样的故事。“我的系统以前运行得很好, 但是现在越来越慢了——我没有改变任何东西, 究竟发生了什么呢?” 随着系统需要搜索的数据量的增长, 变慢是很自然的, 但是, 这个变化应该不会那么明显, 通常也不应该那么明显。相反, 其原因通常是在第一次安装系统时所采用的性能改进方案已经不再适用了。因为用户使用系统的方式和数据量发生了变化, 因此需要重新组合各种选项以获得最佳性能。

下一章将会深入介绍维护, 然而出于两个原因, 这里也会讨论这个主题。第一, 如果读者因某个特定的性能问题查看本章, 那么这里介绍这个主题是有帮助的。第二, 可能更加重要, 因为有一种倾向把维护看成只是防止系统宕机和当最糟糕的情况发生时确保备份可用。事情完全不是这样的。从性能角度来看, 维护也是非常关键的。

21.3 故障排除

SQL Server 提供了很多选项用以帮助防止、检测和度量长时间运行的查询。其中有测量实际性能的被动方法, 也有更加主动的使用查询“调控器”来自动停止那些运行时间超过预先选择的某个数值的方法。这些工具常常被忽视, 或者只被保守地使用——这是一种悲剧——它们能够直接把你引导到有问题的查询上, 甚至直接引导到查询中造成性能问题的特定部分, 从而为排错节省下很多时间。

本节中介绍的工具包括:

- 数据收集器
- SHOWPLAN TEXT|ALL 和图形显示计划
- STATISTICS IO
- 数据库控制台命令(DBCC)

- sys.processes 系统视图
- 活动监视器
- SQL Server Profiler
- 性能监视器

很多人都只用过其中一种工具,但是现实是它们之间只有少许或者没有重叠(取决于拿哪两个工具进行比较)。这意味着只使用其中一种工具的开发人员和 DBA 实际上会错失很多潜在的重要信息。

同样,记住即使使用某种客户端语言并把查询发送到服务器处理(没有存储过程),其中大多数工具在某种形式上仍然是有用的。可以使用 SQL Server Profiler 观察发送到服务器的查询,甚至可以在把它移到客户端代码中之前在 QA 中对它进行测试。

21.3.1 数据收集器

数据收集器是 SQL Server 2008 中新增的工具,它提供了一个框架把系统数据和与活动有关的数据收集到一起并能够对此进行分析、排错(是的,SQL Server 实际上能够使用数据解决它自身的一些问题!)以及持久化这些结果以供进一步地分析和诊断。

数据收集器的组成部分包括:

- 实际的数据收集引擎
- 主动性能监视、排错和优化
- 报表

与之前的发行版中的各种诊断工具相比,这个工具是一个飞跃。数据收集器可以以企业为单位对收集到的数据进行聚合,并且可以跨越多台服务器进行报表和分析工具。

就数据收集器本身而言,其安装和配置需要仔细考虑和分析,其中大部分内容已经超出了本书的范围(很大程度上是管理员的职责),但是一些关键元素包括:

- 设置一个拥有适当权限的登录帐号来收集数据和监视收集到的数据。
- 创建收集集合(一组使用一个或多个收集提供程序来收集数据的对象)。
- 安排数据收集任务。

显然仅凭上面的内容是无法充分理解这个工具的,但是这些内容已经说明配置一个数据收集器不是那么容易的。但是,它能提供有价值的信息,并且当进行伸缩性分析时对测试系统是非常有用的,同时对大型生产环境也非常帮助。

注意:

数据收集器和与之相关联的工具框架是域相关的,可以用来从多台服务器上收集数据,并把这些数据存储在数据仓库以供比较和对整个企业进行分析。在整个企业范围内进行数据收集是 DBA 的职责,它已经超出了本书的范围(但是需要知道有这么一个强大的工具可用!)。

21.3.2 各种显示计划和 STATISTICS

SQL Server 提供了几种不同的选项来显示任意给定的查询所使用的特定计划。根据所选择的选项,其提供的信息可能会有些不同,虽然不同选项提供的信息有相当一部分是重叠的,但是每一个选项肯定会提供其特定的信息。此外,还有很多选项可以用来显示查询统计信息。

下面介绍这些选项以及它们的作用。

1. SHOWPLAN TEXT|ALL

当这两个选项(它们的互斥的)中任何一个被执行时, SQL Server 会修改查询的结果。实际上, NOEXEC 选项(这是说“找出查询计划, 但是不要执行该查询”)将会被设置, 除了 SHOWPLAN 给出的信息之外将不会显示任何结果。

启用和关闭 SHOWPLAN 选项的语法是相当简单的:

```
SET SHOWPLAN TEXT|ALL ON|OFF
```

使用 TEXT 选项会返回查询计划以及运行该计划的预计成本。由于 NOEXEC 将自动附着在 SHOWPLAN 之上, 因此读者将不会看到任何查询结果。

使用 ALL 选项会收到 TEXT 选项返回的所有内容加上大量额外的统计信息, 其中包括:

- 实际计划的物理和逻辑操作
- 估计的行数
- 估计的 CPU 使用率
- 估计的 I/O
- 每行的平均大小
- 查询是否以并行的方式运行

下面使用这两个选项(一次一个)运行一个非常简短的查询:

```
USE AdventureWorks2008;
GO

SET SHOWPLAN_TEXT ON;
GO

SELECT *
FROM Sales.SalesOrderHeader;
GO

SET SHOWPLAN_TEXT OFF;
GO

SET SHOWPLAN_ALL ON;
GO

SELECT *
FROM Sales.SalesOrderHeader;
GO

SET SHOWPLAN_ALL OFF;
GO
```

注意每个语句后面都要跟着一个 GO——这样让它成为自己的批处理的一部分。包含实际查询的批处理可以拥有无限数量的语句, 但是设置了 SHOWPLAN 选项的批处理各自必须要处于一个批处理中。

SHOWPLAN_TEXT 选项的结果如下所示:

StmtText

```

-----

SELECT *
FROM Sales.SalesOrderHeader

(1 row(s) affected)

StmtText
-----
|--Compute Scalar(DEFINE:([AdventureWorks2008]....
  |--Compute Scalar(DEFINE:([AdventureWorks2008]...
    |--Clustered Index Scan(OBJECT:([AdventureWorks2008]...

(3 row(s) affected)

```

遗憾的是，结果太宽了以至于无法在本书的页面中正常排出，但是在生成的内容中有一些关键的东西希望读者要注意：

- 这里显示了多个步骤。
- 在每个步骤中都标出了用到的对象以及提供的操作类型。

如果运行的是一个很大的查询——比如需要进行几次连接——那么甚至会列出更多的子过程，每个子过程都会有缩进以表明层次关系。

这里没有列出 ALL 选项的结果，因为它们不适合本书的格式(它们宽 800 个字符，不适合在书中阅读——即使把它们向一边翻转，也容纳不下)，但是其中包含了一大堆信息。至于到底该使用哪个选项则本质上取决于需要获取多少信息。如果只需要知道基本计划(例如它使用了合并连接还是散列连接)，那么可能只需要使用 TEXT 选项。如果想要知道开销到底在哪里等信息，则需要使用 ALL 选项。

注意：

由于 SHOWPLAN 选项隐含了 NOEXEC 选项，这意味着实际上查询将不会被执行。因此在做其他事情之前需要关闭这个选项，甚至是从一个显示计划切换到另一个显示计划也需要这样做(例如，如果已经运行了 SET SHOWPLAN_TEXT ON 但还没有关闭它，那么运行 SET SHOWPLAN_ALL ON 将不起作用)。

笔者喜欢确保在所有运行的包含 SET SHOWPLAN 语句的每个脚本中，同一脚本都包含一个启用选项和一个关闭选项。这样能够防止忘记已经启用了该选项并且在事情没有像预期那样进行时产生困惑。

2. 图形显示计划

图形显示计划工具对 SHOWPLAN_ALL 选项中的一些片段进行了组合并且把它们封装到单个图形格式中。图形显示计划是一个只在 Management Studio 中可用的工具。可以通过 Management Studio 中的选项来选择它，但无法使用 T-SQL 语法选择它——这意味着只有在使用 Management Studio 时才能使用这个工具。

图形显示计划有两个版本：估计和实际的。估计版本在很大程度上类似于 T-SQL 中的 SHOWPLAN，这意味着只开发查询计划，但不会真正执行这个计划。这实际上会等待，直到查询完成，最后会给出实际完成查询的方式。

为什么它们是不同的呢？好吧，因为 SQL Server 足够聪明，它能够识别出何时根据估计的开销开始运行一个查询计划，然后找出实际情况与估计所依赖的基础之间的差别。SQL Server 使用它自己保存的表和索引上的统计信息来估计开销。而这些统计信息有时候可能会被歪曲或者完全过时。查询优化器在执行时如果发现与预期不相符的地方会即时进行调整。

对于大多数任务而言，估计的执行计划已经很好了。激活图形显示计划选项有三种方法：

- 在“查询”菜单中选择“显示估计的执行计划”选项
- 按下键盘上的 Control+L 键
- 单击工具栏或“查询”菜单上的“显示估计的执行计划”按钮(这个选项仅显示计划，其中 NOEXEC 选项是被激活的)

提示：

就笔者个人而言，我喜欢在正常的查询运行之外使用图形显示计划选项。虽然这意味着不得不在系统中放入实际的查询命中信息，但是它也意味着所得到的数值不再只是一个估计值，而是一个以实际开销为基础的数值。实际上，如果用两种方式运行了显示计划却得到了差别非常大的结果，那么需要看一下最后一次在查询所依赖的表上更新的统计信息。如果需要可以手动它们，然后再次尝试运行该过程。

接着会以图形化的形式显示不同子过程之间的层次关系。要查看任何一个子过程的开销以及其他特定信息，只需要把鼠标悬停在图形显示计划的那部分上，然后就会在一个提示窗口中显示相关信息：

这个布局安排使得从计划中挑选不同部分变得更加容易，如图 21-1 所示。但是其缺点是无法像文本版本那样将结果打印出来用于报表。

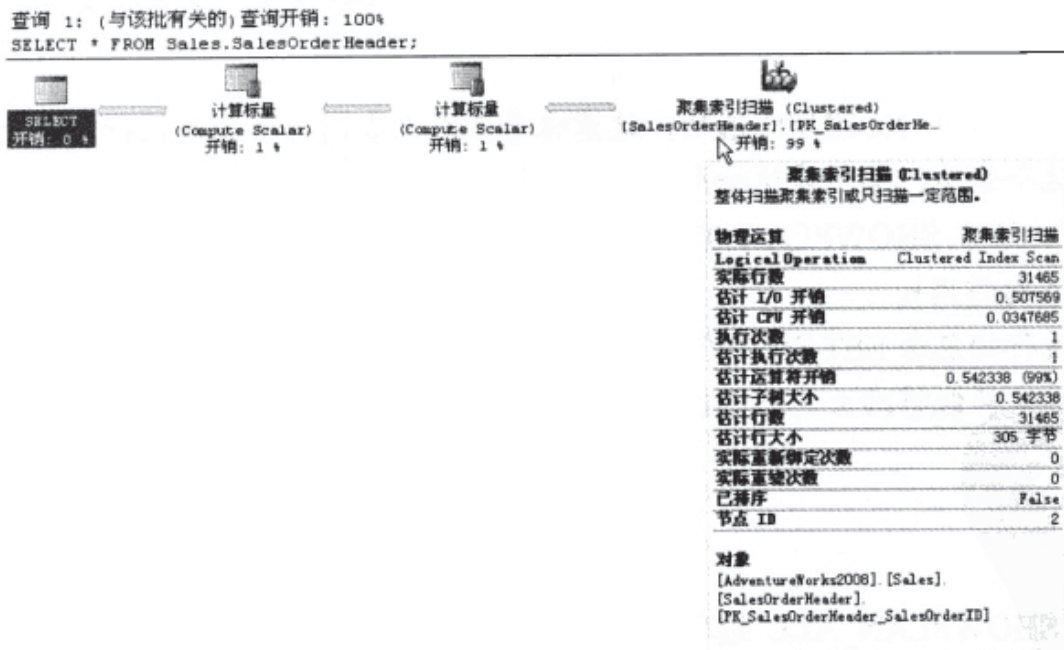


图 21-1

3. 统计信息

除了使用带真正执行查询功能的图形显示计划之外，还有很多选项可以用于在查询的统计信息上提取“真正的”信息：使用 SQL Server Profiler(本章后面将会讨论这个工具)和启用 STATISTICS PROFILER。

实际上,在对查询性能进行排错时, STATISTICS 中的一些选项是非常方便的,包括下面几节中将要讨论的选项。

4. SET STATISTICS IO ON|OFF

这是一个非常常用的工具,用来找出查询在哪里执行以及如何执行。STATISTICS IO 提供了几个与执行查询所必需的实际工作相关的重要信息片段。提供的信息包括:

- **物理读取数:** 它表示从磁盘上读取的实际物理页面数。它永远不会大于,通常会小于逻辑读取的数量。当第二次运行查询时这个数字通常会发生变化(比第一次运行的时候要少),因此这个指标非常容易误导人。对于任何已经在缓存中的页面都将不再进行物理读取,因此如果在相当短的时间内第二次运行查询,那么有关的页面很有可能仍然在缓存中。此外,如果页面已经由 SQL Server 的预先读机制读取过了,那么这个数字将不会增加。这意味着查询可能需要负责物理地把页面加载到缓存中,但是在物理读取数指标中却没有相应地显示出来。
- **逻辑读取数:** 这是页面被实际读取的次数——不管它是从哪里来的。也就是说,任何已经在内存缓存中的页面仍然会产生逻辑读取,只要查询使用到它。注意这里是说页面被读取的次数。这意味着如果一个页面被读取了多次(比如一个嵌套循环需要使用一个页面上的几行数据),那么该页面将会产生多个逻辑读取。
- **预先读取数:** SQL Server 的预先读机制会估计将会被用到的页面并把这些页面读取到缓存中,这个指标表示被预先读取到缓存中的页面数。这些页面可能会被用到——也可能不会被用到。不管有没有被用到,因为是预先读,因此计数器将会增加。预先读取数与物理读取数非常类似,因为它们都表示物理地从磁盘上读取的数据。这里的问题是这个数字是以预先读机制的乐观的性质为基础的,因此并不意味着所有读取的页面都会被用到。
- **扫描次数:** 扫描次数表示一张表被访问的次数。它与逻辑读取数在某种程度上有点不同,逻辑读取数关注的是页面被访问的次数。这是另一个可以用嵌套循环作为例子的情形。构成内部查询的条件基准的外部表的扫描数可能只有 1,但是内部循环表的扫描数每次都会随着循环(即外部表中的每一个记录)而增加。

构成 STATISTICS IO 基础的一些公共信息将会为性能监视器中的缓存命中率提供信息。缓存命中率是用逻辑读取数减去物理读取数,然后再除以总的实际读取数(逻辑读取数)。

在 STATISTICS IO 所给的信息中需要关注的是那些物理或逻辑读取数看起来十分不成比例的表。

如果物理读取数非常高,那么这表明其他进程已经把表中的数据从缓存中挤出了。如果这是一张需要定期访问的表,那么可能需要考虑为系统购买(或者如果读者是一个开发 SQL Server 产品的 ISV,那么建议向客户提出建议)更多的内存了。

如果逻辑读取数非常高,那么该问题可能不仅仅是一个合适的索引能够解决的。下面的例子来自于笔者以前的一个客户。系统在不加载其他部分的时候运行一个查询大概需要 15 秒钟。由于该系统是一个真正的 OLTP 系统,因此让用户花这么长时间来等待输出信息是不可接受的(这个查询实际上是一个相当简单的查找,它碰巧需要对四张表进行连接)。为了找出问题所在,我使用了 STATISTICS IO 工具所返回的信息。虽然使用的是 SQL Server 6.5 中的老式的图形版本,但是数据相差不大。在仅仅把查询运行了一次之后就发现该进程在其中三张表上所需的逻辑读取数少于

20, 但是在第四张表上发生的逻辑读取数却超过了 45 000。只花费了半秒钟时间就可以看到某一张表对应的直方图伸展跨越了整个屏幕, 而其他几张表对应的直方图只占据了几个像素, 这也是我喜欢旧版本的图形工具的原因。从这张图中马上就可以知道问题在哪里——在两分钟之内, 我创建了一个索引以支持外键(记住在默认情况下不会创建它们), 之后响应时间就降到了一秒以内。在这个例子中整个排错过程只花了几分钟, 但是不是所有的性能排错过程都是那么容易的(实际上大多数的都不容易), 但是使用正确的工具可以帮上不少忙。

5. SET STATISTICS TIME ON/OFF

很奇怪, 很少有人知道这个选项。它给出了执行这个查询实际所需的 CPU 时间。就个人而言, 笔者通常在被测试的查询前后使用简单的 `SELECT GETDATE()`——本书中的大部分例子也是这样做的, 但是有这个选项是非常方便的, 因为它把解析和计划查询所需的时间和真正执行查询所需的时间分离开了。它还有一个优点就是不需要自己去计算了(它以毫秒为单位计算时间, 而是用 `GETDATE()`则需要自己进行单位转换)。

包括客户端统计信息

作为查询运行的一部分, 还可以显示连接相关的统计信息。要使用这个功能只需要在“查询”菜单中选择“包括客户端统计信息”。只要设置了这个选项, 每一次执行查询时都会在 Query 窗口的结果窗格中产生一个“客户端统计信息”选项卡, 如图 21-2 所示。

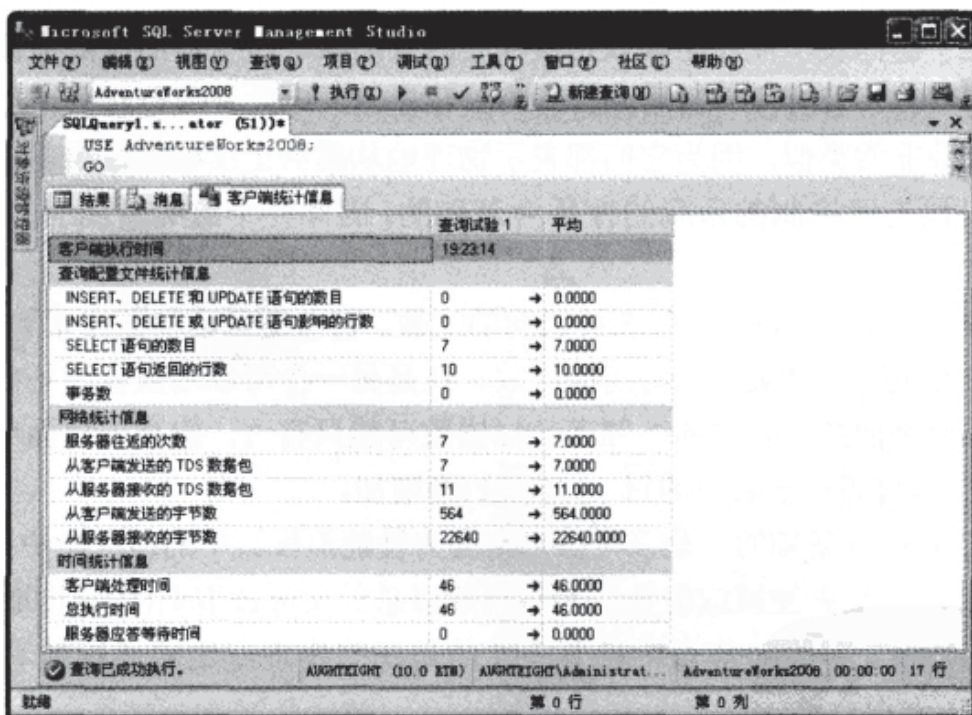


图 21-2

21.3.3 数据库控制台命令(DBCC)

数据库控制台命令(或 DBCC)提供了很多不同的选项允许检查数据库的完整性和结构组成。通常, 这更多的是 DBA 的职责, 而不是开发人员的职责, 因此, 笔者认为 DBCC(至少是大部分的内容)已经超出了本书的范围。

提示:

读者可能还听过 DBCC 是指数据库一致性检查器。DBCC 以前是代表这个。坦白讲, 笔者也

不知道 DBCC 的含义是什么时候被改变的,但是如果你听到了另一个术语,那么现在就知道原因了。

21.3.4 动态管理视图

在 SQL Server 的前一个或前两个版本中,Microsoft 一直在添加名为动态管理视图——或 DMV 的东西。附录 B 提供了这个工具相关的说明和使用信息。它们能够提供广泛的与服务器和/或数据库当前状态相关的信息,并且这些信息的可读性非常好(它们有利于管理任务自动化)。为了能够用例子快速说明这些工具的强大之处,下面快速看一下读者可能感兴趣的一个 DMV。

提示:

需要强调的是本节中介绍的内容实际上只是 SQL Server 提供的各种元数据和动态管理视图所能完成的任务中的一个非常小的部分。读者可以阅读本书中的附录 B 来扎实地学习它们,但是如果需要构建一个可靠的支持工具,那么可能需要查看专门介绍这个在 SQL Server 中不断膨胀的工具集的书籍。

本节将从回顾在第 13 章中首次碰到的一个例子开始,但是会对游标示例中所使用的查询进行一点修改:

```
SELECT SCHEMA_NAME(CAST(OBJECTPROPERTYEX(i.object_id, 'SchemaId') AS int))
      + '.' + OBJECT_NAME(i.object_id)
      + '.' + i.name AS Name,
      ps.avg_fragmentation_in_percent
FROM sys.dm_db_index_physical_stats (DB_ID(), NULL, NULL, NULL, NULL) AS ps
JOIN sys.indexes AS i
  ON ps.object_id = i.object_id
 AND ps.index_id = i.index_id
WHERE SCHEMA_NAME(CAST(OBJECTPROPERTYEX(i.object_id, 'SchemaId') AS int)) =
'Purchasing'
 AND avg_fragmentation_in_percent > 30;
```

上面的代码将返回所有与 Purchasing 模式中的表相关联的索引——不管它们是属于哪张表的,但是更重要的是,索引碎片要超过 30%。

这里的强大之处是可以很容易地根据表的状态编写维护任务的脚本。与之前用来查看碎片的旧式数据库控制台命令选项相比,这是一个重要的改进。

正如笔者之前指出的那样,这是一个相对简单的例子。与 SQL Server 中的很多主题一样,我确信将来会有整本书专门来介绍 SQL Server 提供的动态管理视图的。同样,阅读附录 B 来获取更多信息。

21.3.5 活动监视器

SQL Server 2008 对活动监视器的界面进行了重大改动并添加了一些附加的功能。旧有的进程信息全都保留在那里,但是添加了大量从各种源,如 PerfMon(用于监控系统的一个 Windows 工具)和数据收集器,收集而来的信息。

可以右击 Management Studio 中服务器节点来打开活动监视器。打开之后会发现 5 个主要的主题区域:

- 概览
- 进程
- 资源等待
- 数据文件 I/O
- 最近耗费大量资源的查询

下面快速介绍一下这些主题。

1. 概览

这个部分是最能提醒你这是一个 PerfMon 的地方。它以可调整的时间间隔(默认为 10 秒钟)对系统活动进行采样,并把系统活动用相对直观的图形显示出来(如图 21-3 所示)。注意这里提供的是 SQL Server 所使用的资源相关的信息——而不是整个系统。

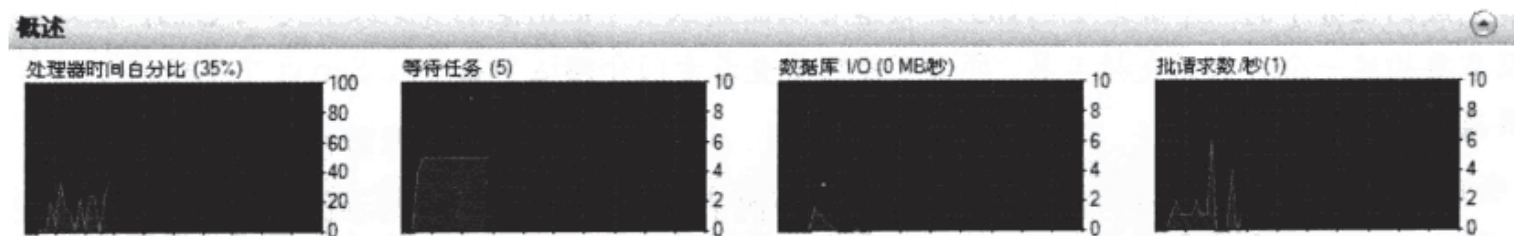


图 21-3

2. 进程

这个部分在很大程度上可以与 SQL Server 2005 中的活动监视器对应起来,如图 21-4 所示。它提供的信息包括哪个进程正在运行、它们当前正在执行的命令和该进程使用的或引起的资源和阻塞方面的度量。

会话 ID	用户进程	登录名	数据库	任务状态	命令	应用程序	等待时间(毫秒)	等待类型	等待资源	阻塞者	阻塞链	内存使用(KB)	主机名	工作组
51	1 Wei\Ad...					Micros...	0					1	16 Wei	default
52	1 NT AUT...					SQLAge...	0					1	16 Wei	default
53	1 NT AUT...					Report...	0					1	16 Wei	default
54	1 Wei\Ad...		tempdb	RUNNING	SELECT	Micros...	0						16 Wei	default
55	1 NT AUT...					Queue ...	0					1	16 Wei	default
56	1 NT AUT...					Queue ...	0					1	16 Wei	default
57	1 NT AUT...		msdb	SUSPENDED	SELECT	SQLAge...	4832	RESOUR...	resour...	54			16 Wei	default

图 21-4

3. 资源等待

在很大程度上与概览类似,这个部分应该能够提醒你这是一个 PerfMon,它使几个不同的计数器来提供有关等待时间方面的度量(如图 21-5 所示)。

资源等待				
等待类别	等待时间 (毫秒/秒)	最近等待时间 (毫秒/秒)	等待应用程序的平均计数	累计等待时间 (秒)
Other	5008	5000	5.0	15008
Network I/O	443	404	0.4	37
Buffer I/O	170	151	0.2	56
CPU	35	51	0.0	10
Latch	0	0	0.0	0
Lock	0	0	0.0	25
Logging	0	0	0.0	3
Memory	0	0	0.0	148
Buffer Latch	0	0	0.0	0

图 21-5

4. 数据文件 I/O

这个部分仍然提供了大量 PerfMon 相关的数字, 主要是 SQL Server 使用的物理文件方面的信息。在这些信息被收集到一个地方之前(如图 21-6 所示), 需要在 PerfMon 中对各个文件进行设置, 然后 SQL Server 会自动收集该种类型的度量。

数据文件 I/O				
数据库	文件名	MB/秒 (读取速率)	MB/秒 (写入速率)	响应时间 (毫秒)
distribution	C:\Program Files\Microsoft SQL Serve...	0.0	0.0	202
tempdb	C:\Program Files\Microsoft SQL Serve...	0.0	0.0	148
AdventureWorks2008	C:\Program Files\Microsoft SQL Serve...	0.7	0.0	67
ReportServer	C:\Program Files\Microsoft SQL Serve...	0.0	0.0	45
tempdb	C:\Program Files\Microsoft SQL Serve...	0.0	0.0	3
AdventureWorks	C:\Program Files\Microsoft SQL Serve...	0.0	0.0	0
AdventureWorks	C:\Program Files\Microsoft SQL Serve...	0.0	0.0	0
AdventureWorks2008	C:\Program Files\Microsoft SQL Serve...	0.0	0.0	0
AdventureWorksDW	C:\Program Files\Microsoft SQL Serve...	0.0	0.0	0

图 21-6

5. 最近耗费大量资源的查询

这个部分提供的信息实际上在 SQL Server 2008 之前是没有的, 如图 21-7 所示。其中一些信息可以使用 SQL Server Profiler(稍后介绍)获得, 但是它提供的信息非常冗长, 并且很有可能会提供大量实际上并不需要的信息(从而掩盖了真正所需的信息)。

最近耗费大量资源的查询							
查询	执行次数/分钟	CPU 时间 (毫秒/秒)	物理读取次数/秒	逻辑写入次数/秒	逻辑读取次数/秒	平均持续时间 (毫秒)	计划计数
SELECT SCHEMA_NAME (sp.schema_id) AS [...	1	5	0	0	0	1646	512
fetch next from hCdatabase into @db...	8	0	0	0	0	0	0
INSERT INTO #an_wait_types VALUES (N...	1445	0	0	0	0	52	0
SELECT SCHEMA_NAME (udf.schema_id) AS ...	1	0	0	0	0	41	1044

图 21-7

提示:

值得明确指出的是, 开销较大的查询的相关信息属于被日志记录到性能数据仓库中的信息的

一部分,这意味着使用数据仓库不仅可以收集前几分钟的度量信息,还可以收集几天甚至几周内的度量信息,当然这要取决于为数据仓库设置的保留规则。

21.3.6 SQL Server Profiler

它是 SQL Server 提供的工具中真正的救护设备,它允许“嗅探”服务器究竟在做什么。

可以在 Windows 的开始菜单菜单中启动 Profiler,还可以通过 Management Studio 中的 Tools 菜单运行它。在首次运行时既可以加载一个现有的 profile 模板,也可以创建一个新的。

下面通过一个简短的示例来介绍主 Profiler 中的一些关键点。

在“文件”菜单中选择“新建”|“跟踪”。登录到正在使用的服务器上,接着会出现图 21-8 中显示的对话框。

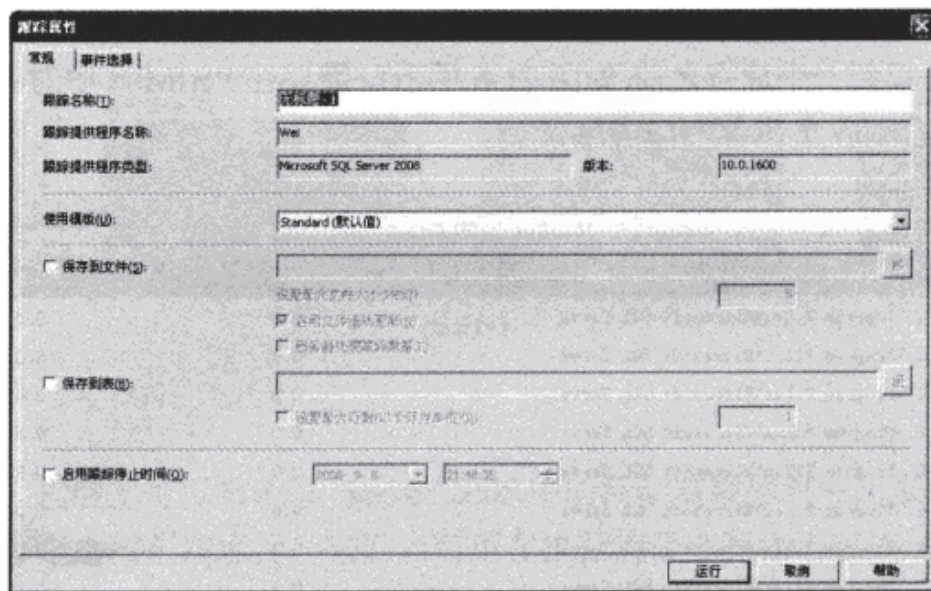


图 21-8

跟踪的名字可能非常明显易懂,但是模板信息可能就不那么明显了。模板是一组预先设置的希望在跟踪中看到的事件、数据列和筛选器,SQL Server 提供的模板是以特定的使用场景命名的,读者可能会想要在这些场景中使用它们。所有存储在默认的 profiler 模板目录下(在 SQL Server 安装目录下的 tools 目录下)的模板都会包含在“使用模板”下拉菜单中。

注意:

特别要注意所选择的模板。它决定了下一个选项卡上究竟会显示多少信息。如果所选择的模板的限制性太强,那么可以选择“显示所有事件”和“显示所有列”展开所有可能的选择。

接着可以选择把跟踪信息存储到磁盘上的某个文件中或者存储到数据库的某张表中。如果存储到某个文件中,那么这个文件只对它所在的系统可用(或者那些能够访问网络共享文件夹的用户,其前提是把该文件存储到了共享文件夹中)。如果存储到某张表中,那么所有能够连接到服务器并拥有适当权限的人都可以检查跟踪。

最后,但不是最不重要,对话框上有一个停止时间功能。它允许让跟踪保持运行(例如,针对工作量文件或其他一些长时间跟踪的需要)并在后面的某个时间自动关闭它。

下一个选项卡将变得更加有趣,如图 21-9 所示。

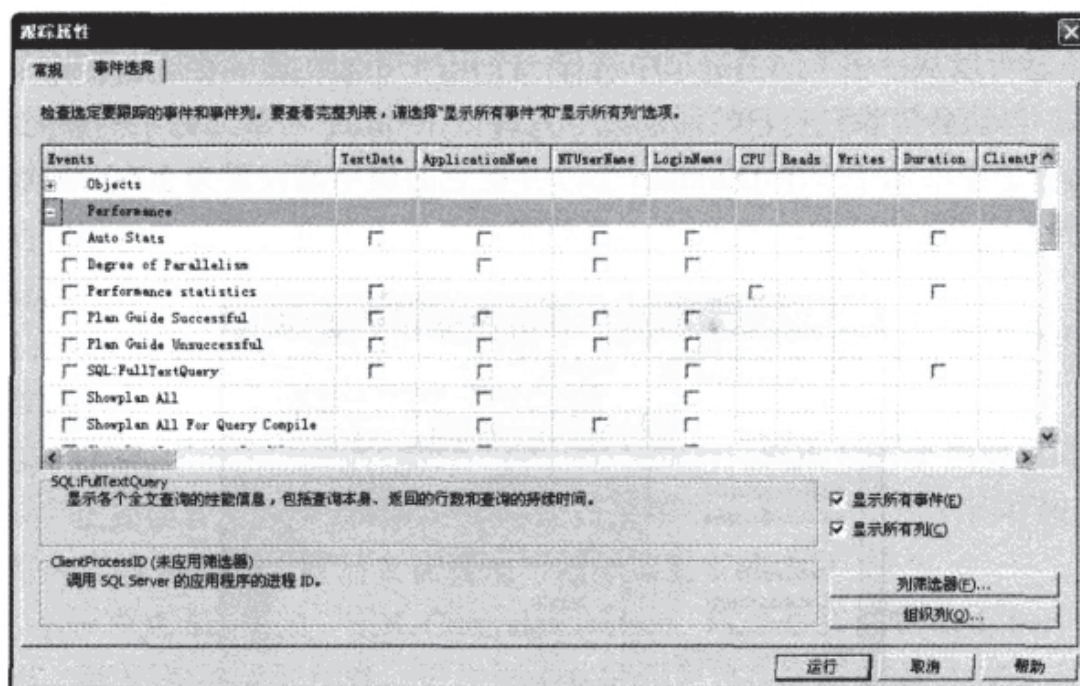


图 21-9

这里选择了“空白”模板，然后滚动到 **Performance** 部分并展开它。该选项卡与将要跟踪的事件相关，正如读者可以看到的那样，其范围是相当大的。例如，如果选择 **Tuning** 跟踪模板，那么初始的设置是针对跟踪数据库引擎优化顾问再加上其他一些内容进行设置的，此外，可以使用该表来选择针对每种事件类型所希望收集的信息。

这里的诱惑是选择能看到的所有项目，这样就能确保拥有所有的信息。但是有两个理由不应该这样做。首先，这意味着会有大量额外的文本通过管道返回到服务器上。记住 **SQL Server Profiler** 必须要在系统中放置一些审计，而这意味着无论何时，只要 **Profiler** 在运行，那么系统就会承担额外的负荷。同时，跟踪越大，负荷也就越大。其次，这通常意味着生产率将会降低，因为在前进的过程中必须要处理一大堆的数据——其中大多数可能并不需要。

在继续讲述之前需要指出一些关键区域：

- **TextData**：这是 **Profiler** 在此刻添加到跟踪中的语句的实际文本。
- **Application Name**：这是另一个没有被充分利用的特性。在为客户端创建连接的时候可以设置应用程序名称。如果使用 **ADO.NET** 或其他数据对象模型和底层连接方法，那么可以在连接字符串中把应用程序名称作为参数进行传递。在 **DBA** 对系统进行排错的时候是该选项是非常方便的。
- **NT User Name**：该选项就像听起来那样，它的好处在于能够提供一定级别的可问责性。
- **Login Name**：与 **NT User Name** 一样，但是只在 **SQL Server** 安全性下使用，而不在 **Windows** 安全性下使用。
- **CPU**：实际使用的 CPU 周期。
- **Duration**：该查询运行的时间长度——包括等待锁之类的资源所需的时间(此时 CPU 可能不在做什么事情，因此不能把这个数据作为负荷的参考)。
- **SPID(SQL 进程 ID)**：如果跟踪揭示了某些事件，而该事件的发生意味着需要终止某个进程，那么该选项是非常好用的。在 **KILL** 语句中需要使用这个数字。

接着继续讲述，下面将介绍我认为的最重要的选项之一——列筛选器。

该选项能够确保在一台生产服务器或负载测试服务器上打开跟踪几分钟之后不会被几千页

的垃圾数据湮没。

利用列筛选器可以从很多不同的选项选择一个用来筛选数据并限制结果集的大小。在默认情况下, Profiler 会自动排除它自己的活动以尽量降低 Profiler 对最终数字的影响。例如, 在图 21-10 中所示的例子中添加了一个 Duration 值, 并把它的最小值设置为 3000 毫秒, 同时没有设置最大值。

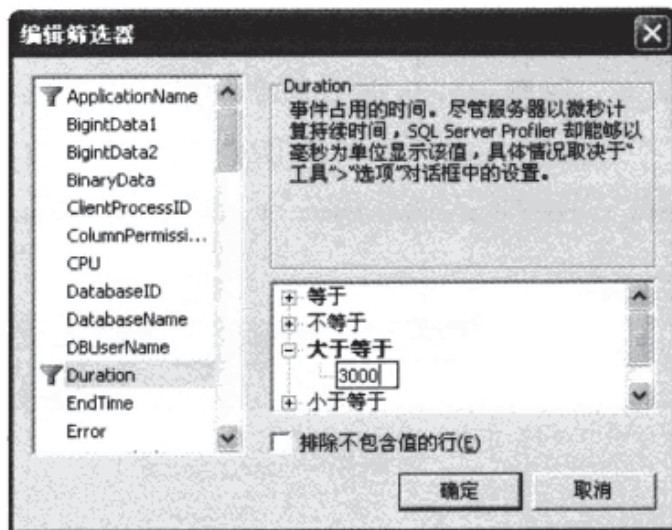


图 21-10

奇怪的是如果在 Sales.SalesOrderHeaders 表上运行查询将不会看到它出现在跟踪中。这是为什么呢? 这可能是因为查询运行得很快, 并不满足包含在跟踪中的条件——这个例子说明了如何设置一个跟踪来抓取在系统上运行的所需时间非常长的查询的查询文本和此人的用户名。现在尝试运行一些时间较长的查询——如一个需要连接很多大型表的查询。现在很有可能就超过了持续时间的阈值, 这样查询就会显示在 Profiler 中(如果没有显示出来, 那么试着减少在 Profiler 中设置的预期的持续时间值)。

在解决性能和其他问题方面再怎么强调这个工具的重要性都不过分。有时候笔者认为存储过程是只沿着一个逻辑路径运行的, 但是没想到却发现一份完全不同的分支被执行了, 这种情况我已经数不清到底发生了多少次了。但是我最初是怎么发现的呢? 是通过在 Profiler 中观察它的执行来发现的。

21.3.7 性能监视器(PerfMon)

在 Windows 上安装完 SQL Server 之后, SQL Server 会向可靠性和性能监视器(有时候叫做 PerfMon, 因为可执行文件的文件名是 perfmon.msc)中添加几个计数器。

在找到问题所在和确定一些问题的性质方面, 它是一个非常优秀的工具。

注意:

在 Windows Vista 和 Windows Server 2008 之前, 可靠性和性能监视器只是简单地叫做性能监视器。

虽然现在很多相关的计数器已经在 Management Studio 的活动监视器中了, 但是仍然可以通过 Windows 中的管理工具菜单来访问可靠性和性能监视器。SQL Server 有很多不同的性能对象, 在每个对象中都有与该对象相关的一组计数器。从历史上看, 其中一些重要的对象包括:

- **SQL Server Cache Manager: Buffer Hit Cache Ratio:** 这是从快速缓存中,而不是从物理磁盘上读取的页面的数量。这里需要注意的是可以不用关注这个数字,当然这要取决于预先读机制的有效程度——所有在查询真正用到之前被预先读机制读取并放入缓存中的页面都算作一次命中——即使对查询来说实际上是一次物理读取。通过这个数字可以知道内存的利用效率如何。实际上这个数字要高一点(大于 90%)以获取最大的性能。一般来讲,低缓存命中率表明需要更多的内存。
- **SQL Server General Statistics: User Connections:** 顾名思义,这是系统中当前活跃的用户连接数。
- **SQL Server Memory Manager: Total Server Memory:** SQL Server 当前总共使用的动态内存数量。正如读者所预期的那样,当这个数字相对于系统中可用的内存较高时(记住还要留一些内存给 O/S!),就需要认真地考虑添加更多的内存了。
- **SQL Server SQL Statistics: SQL Compilations/sec:** 这个数字说明 SQL Server 编译事物(存储过程、触发器)的频繁程度。记住这个数字还会包括重新编译(由发生在索引统计信息上的变更或显式地请求重新编译引起)的次数。在服务器首次启动时,这个数字可能会急速上升,但是当服务器在恒定的一组活动上以恒定的速率运行一段时间后,这个数字将会变得稳定起来。
- **SQL Server Buffer Manager: Page Reads/sec:** 在服务器的磁盘上发生的物理读取次数,这个数字应该要相对小一点。遗憾的是,由于每个系统的需求和活动都是不同的,因此这里无法给出一个基准。
- **SQL Server Buffer Manager: Page Writes/sec:** 在服务器的磁盘上发生的物理写入次数,同样这个数字应该要小一点。

如果想要添加或修改其中任何一个,只需要单击工具栏上面的加号(+)即可。接着会弹出一个对话框要求在不同的对象间进行选择并选择系统上可用计数器(不仅仅是那些与 SQL Server 相关的),如图 21-11 所示。

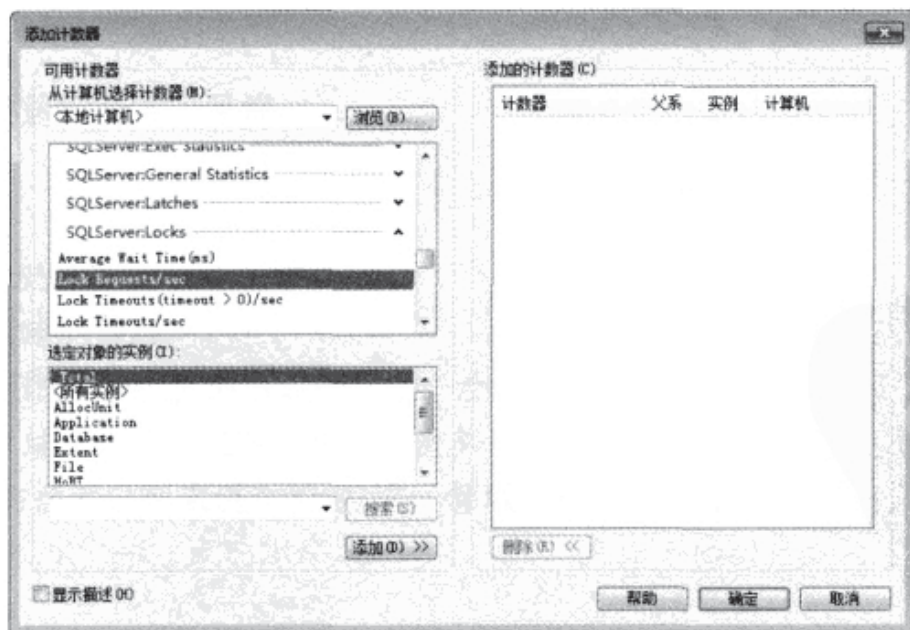


图 21-11

重要的是需要意识到可以混合搭配各种各样的计数器以更好地理解服务器目前的状况,从而能够做出适当的调整。在大多数情况下,这种任务更多的是 DBA 的职责,而不是开发人员的职

责，但是在对应用程序进行负载测试时能够获得这么多的状态信息是很有用的。

21.4 小结

性能本身是能够，也应该用一本书来讲述的。它所包含的内容有很多，以至于无法在一章或甚至几个章节内介绍所有的内容并使读者熟知这些内容。笔者尝试解决这个问题的方式是指出贯穿在本书中的性能问题，这样读者就可以一次阅读其中一部分。

最重要的是要有一个计划——一个性能计划。在项目的启动阶段就要把性能当成一个问题来对待，尽早建立基准，并不断地对照这些基准度量系统以了解哪个部分提高了以及哪些问题可能需要解决。

本章回顾了在本书中提到过的一些性能考虑事项以及需要考虑的几个新增的工具和想法。

下一章将会介绍管理问题。正如读者已经在本书中的一些小节中看到的那样，对性能来说，恰当的管理也是一个关键的因素。

第22章

管 理

好，到现在为止已经介绍了所有核心数据库主题及其相关内容。不过仍然需要用一到两章来扫除开发工作上的障碍，但是大多数内容不都已经介绍过了吗——不，并非如此!!! 对于开发人员来说会认为所有内容都讲完了，但是对于正在构建的应用程序来说，这只是一个开始。因此该讨论一下所开发的数据库的维护和管理了。

如果有什么内容是希望灌输给读者在数据库开发过程中遵循的，那么就是避免“嗨，我已经构建好了——现在是你的问题了”的态度，这种态度在数据库驱动的应用程序开发中是非常常见的。很多开发人员都无法胸有成竹地编写错误相对较少的代码并声称它是能够工作的。由于软件仅仅能够运行并不意味着在最终用户长时间的使用中不会发生错误，因此查看系统是如何被使用的以及弄清楚需要做哪些调整使得软件能够正常工作是非常重要的。

本章将会介绍其中一些必须要做的任务，以确保最终用户不仅能够从问题和灾难中恢复，而且能够执行一些保持系统平稳运行的基本维护工作。

要介绍的主题包括：

- 计划作业
- 备份和恢复
- 基本的碎片整理和索引重建
- 设置警报
- 存档
- 使用 PowerShell
- 考虑基于策略的管理

虽然相对于所有的管理任务而言这些只是一小部分，但是它们确实代表了在应用程序的部署计划中应该涉及的“最低限度”的管理任务。此外，还会进一步介绍如何使用在 SQL Server 2008 中新增的基于策略的管理框架来进行监视(这一领域中的几个项目已经在前面一章的性能调优部分中讨论过了)。

提示：

这是众多我认为必须要与入门书籍的章节所讨论的内容有些重叠的章节之一。现实是大多数的开发人员——甚至是在 SQL Server 领域属于相对专家级别的人员——对于作业计划、索引碎片甚至是备份会恢复都了解得很少。但是如果只是因为看到了入门书籍就认为自己已经了解了本章

中将要介绍的所有内容，那么就要小心了。本章在其中几个主题上增加了更高级的内容，同时还增加了很多管理任务的代码驱动处理方面的内容。

22.1 计划作业

在本章余下的部分将要介绍的很多任务都是可以计划的。计划作业允许在系统处于非高峰时间时在系统上执行任务。它还能确保不会忘记一些事情，从索引重建到备份，读者可能已经一次又一次地听说这样恐怖的故事：公司“忘记”了做某些事情，或者认为已经创建了计划作为，但是从来没有检查过这些作业。

提示：

如果处于 Windows Server 的环境中，并且已经使用过 Windows 的计划管理器服务计划过其他作业，那么可以使用那个计划引擎来支持 SQL Server。虽然把所有的事情都放在 Windows 计划管理器里可以在一个地方管理所有的事情，但是 SQL Server 提供了一些更加可靠的分支选项。

这里有两个基本的术语需要弄清：作业和任务。

- **任务：**它们是将要被执行的单个进程，或将要被运行的一批命令。任务是不能独立存在的——它们只作为作业的成员而存在。
- **作业：**它们是一个由一个或多个应该一起执行的任务组成的组。然而，可以设置依赖关系和根据各个任务的成功或失败来设置分支(例如，如果前一个任务执行成功，那么执行任务 A，而如果前一个任务执行失败，那么执行任务 B)。

可以根据下列方式来计划作业：

- 每天、每周或每月
- 一天中特定的时刻
- 一个特定的频率(比如每十分钟或每小时)
- 当 CPU 空闲一段时间后
- 当 SQL Server 代理启动时
- 对一个警报进行响应

任务是作为作业的一部分并根据为作业定义的分支规则来运行的。一个作业的运行并不意味着该作业中的所有任务都会被运行。其中一些任务可能会被运行，但是其他的则要取决于作业中前一个任务是否成功或失败和已确定的分支规则。SQL Server 不仅允许一个任务在另一个任务结束之后自动运行，还允许在当前任务失败之后做一些完全不同的事情(如运行某种类型的恢复任务)。

除了分支之外，还可以根据发生的情况让 SQL Server：

- 向操作员提供作业执行成功还是失败的通知。允许分别通过网络信息(当用户登录后将会弹出一个用户界面)、寻呼和电子邮件地址向操作员发送各种通知。
- 把信息写入事件日志。
- 自动删除作业(防止以后被执行并进行常规的“清理”)。

下面快速介绍一下如何在 Management Studio 中创建操作员，接着将会介绍如何创建其他计划作业所需的对象。

22.1.1 创建操作员

如果想要使用 SQL 代理的通知功能,那么必须要设置一个操作员来定义对谁进行通知的细节。这种类型的事情——创建操作员——通常不是通过任何类型的自动过程或作为开发代码的一部分来完成的。通常,它们是由 DBA 手工创建的。下面将简要介绍一下如何创建操作员,读者只需要了解一下它是如何同任务的计划管理一同工作的即可。

1. 使用 Management Studio 创建操作员

要使用 Management Studio 创建操作员,需要定位到需要创建操作员的服务器上的 SQL Server 代理节点。展开这个 SQL Server 代理节点并右击“操作员”,并选择“新建操作员”。

提示:

注意,根据特定的安装,SQL Server 代理服务默认可能不会自动启动。读者如果碰到什么问题或者如果 Management Studio 中的 SQL Server 代理图标中有一个小的红色方框,那么很有可能已经把该服务器设置为手动启动或者甚至已经禁用了该服务——读者可能想要把该服务改为自动启动。不管怎样,在练习本章的例子的过程中确保它是运行的。读者可以右击“代理”节点并选择“启动”来启动该服务。

接着将会出现图 22-1 中显示的对话框(我的对话框中已经填写了一部分内容)。

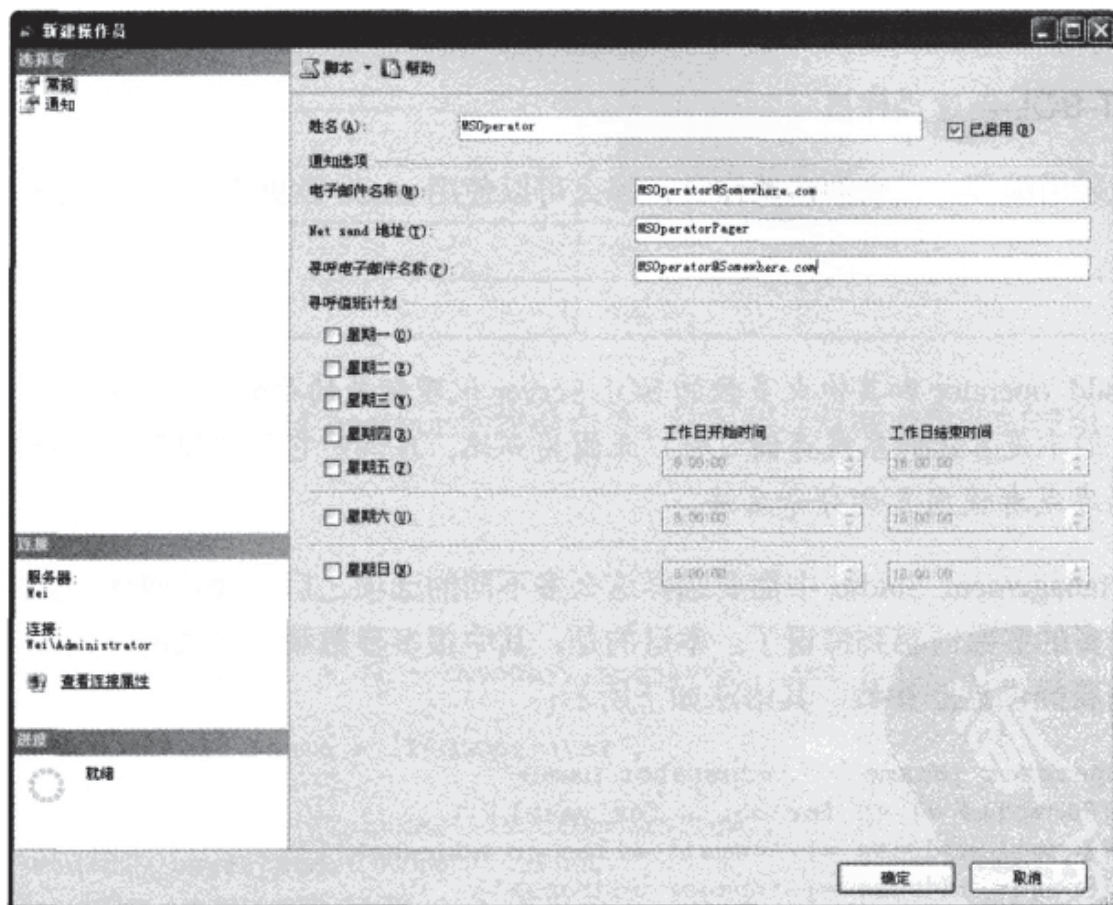


图 22-1

接着可以在这里填写计划来设置该操作员在什么时候会接收到与“通知”选项卡中特定类型的错误相关的电子邮件通知。

说到“通知”选项卡,继续并单击该选项卡,如图 22-2 所示。



图 22-2

除非在系统中有更多的警报(本章后面将会介绍这个主题), 否则这一页没什么太多的意义。在这一页中可以设置当已定义的警报被触发时操作员将会接收到什么样的通知。同样, 在介绍警报之前很难理解这个概念, 但是目前只需要知道当数据库中特定事件发生时就会触发警报即可, 这个页面定义了特定操作员接收哪些警报。

2. 使用 T-SQL 创建操作员

如果决定采用编程方式来创建操作员, 那么可以使用 msdb 数据库中的 sp_add_operator 存储过程。

注意:

注意 sp_add_operator 和其他大多数的 SQL Server 代理相关的存储过程都是通过 msdb 数据库来管理的, 它们不是真正的系统存储过程。正因为如此, 在调用它们的使用需要使得 msdb 数据库为当前数据库或者使用三部分命名法。

看到在 Management Studio 中需要选择这么多不同的选项之后, 读者应该不会对这个存储过程拥有如此众多的参数而感到惊讶了。幸运的是, 其中很多参数都是可选的, 因此只有在用到它们的时候才需要提供这些参数。其语法如下所示:

```
sp_add_operator [@name =] '<operator name>'
[, [@enabled =] <0 for no, 1 for yes>]
[, [@email_address =] '<email alias or address>']
[, [@pager_address =] '<pager address>']
[, [@weekday_pager_start_time =] <weekday pager start time>]
[, [@weekday_pager_end_time =] <weekday pager end time>]
[, [@saturday_pager_start_time =] <Saturday pager start time>]
[, [@saturday_pager_end_time =] <Saturday pager end time>]
[, [@sunday_pager_start_time =] <Sunday pager start time>]
[, [@sunday_pager_end_time =] <Sunday pager end time>]
[, [@pager_days =] <pager days>]
```

```
[, [@netsend_address =] '<netsend address>']
[, [@category_name =] '<category name>']
```

这个存储过程中的大多数参数都是不言自明的，但是有一些参数还是需要深入介绍一下：

- **@enabled**: 这是一个布尔值，其工作方式与通常使用的位标志一样——0 表示禁用这个操作员，而 1 表示启用这个操作员。
- **@email_address**: 这个参数有点儿复杂。为了在 SQL Server 中使用电子邮件，需要使用特定的邮件服务器配置 Database Mail，使它运行起来。这个参数假定所提供的参数值是邮件服务器上的一个别名。如果提供一个更加经典的邮件地址类型 (somebody@SomeDomain.com)，那么需要用方括号把它括起来——就像 [somebody@SomeDomain.com] 这样。注意整个地址——包括方括号——仍然必须要用双引号括起来。
- **@pager_days**: 这是一个数字，表明操作员在哪几天中可以接受寻呼。这可能是所有参数中最难的参数。它使用了单字节位标志的方法，该方法类似于在本书后面介绍系统函数的附录中所介绍的 @@OPTIONS 全局变量所采用的方法。只需要把该操作员活跃的天数的值加起来即可。选项如表 22-1 所示：

表 22-1

星 期	值
星期日	1
星期一	2
星期二	4
星期三	8
星期四	16
星期五	32
星期六	64

好，下面继续并使用 `sp_add_operator` 创建操作员。这里将尽量减少参数的使用，因为很多参数都是多余的：

```
USE msdb;
DECLARE @PageDays int;

SELECT @PageDays = 2 + 8 + 32 -- Monday, Wednesday, and Friday;

EXEC sp_add_operator @name = 'TSQLOperator',
    @enabled = 1,
    @pager_address = 'YourEmail@YourDomain.com',
    @weekday_pager_start_time = 080000,
    @weekday_pager_end_time = 170000,
    @pager_days = @PageDays;
```

如果回到 Management Studio 中并刷新一下操作员列表，那么读者应该会在那里看到新创建的操作员。

如果想要在 T-SQL 中更好地控制操作员，那么还需要使用其他三个存储过程(加上一个提取

信息的过程):

- **Sp_help_operator**: 提供与操作员的当前设置相关的信息。
- **Sp_update_operator**: 接受与 **sp_add_operator** 完全一样的信息; 新信息将会完全替换掉旧信息。
- **Sp_delete_operator**: 从系统中删除指定的操作员。
- **Sp_add_notification**: 接受一个警报名称、一个操作员名称和一个通知方法(电子邮件、寻呼或使用网络发送)。添加通知之后如果警报被触发了, 那么将会通过指定的方式通知指定的操作员。

现在读者已经知道了如何创建操作员, 下面来看一下如何创建实际的作业和任务。

22.1.2 创建作业和任务

正如之前提过的那样, 作业是一个或多个任务的集合。一个任务是工作的一个逻辑单元, 如备份一个数据库或者运行一个 T-SQL 脚本以满足特定的需求, 如重建所有的索引。

尽管一个作业可以包含几个任务, 但是这并不保证作业中的每个任务都会被运行。它们要么被运行, 要么不被运行, 这要取决于作业中的其他任务是执行成功还是失败以及对成功或失败所定义的响应。例如, 可能会在其中一个任务执行失败之后取消作业中余下的任务的执行。

与操作员类似, 既可以在 **Management Studio** 中创建作业, 也可以通过编程方式来创建作业。

1. 使用 Management Studio 创建作业和任务

SQL Server Management Studio 使得创建计划任务变得非常容易。只需要定位到服务器的 SQL Server 代理节点, 然后右击“作业”成员并选择“新建作业”即可。接着将会出现一个多节点的对话框帮助读者一步一步地构建作业, 如图 22-3 所示。

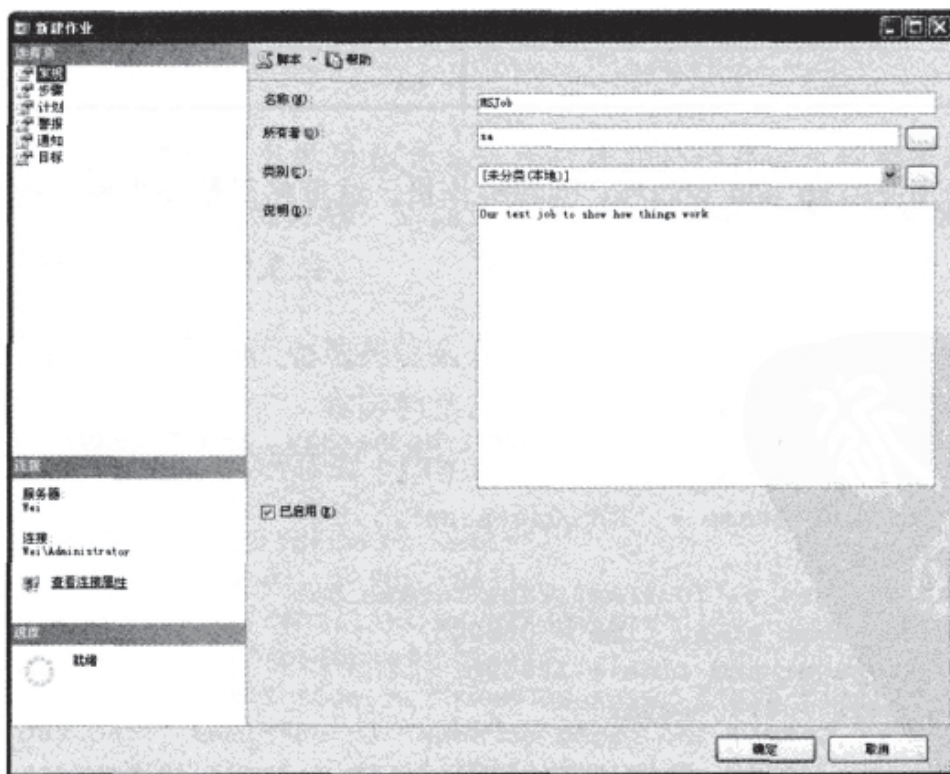


图 22-3

名称可以任意取, 只要符合本书前面介绍的 SQL Server 命名规则即可。

同样，其余信息中的大多数都是不言自明的，除了“类别”之外——它仅仅是把作业组织起来的一种方式。特定于应用程序的很多作业通常是属于未知类别的，但是有时候可能会碰到需要创建 Web 助手、数据库维护、全文或复制作业的情形，在这种情况下可以把它们放到各自的类别中以方便识别。

接着可以移到“步骤”中，如图 22-4 所示。在这里可以告诉 SQL Server 开始创建新任务了，该任务将会作为作业的一部分。

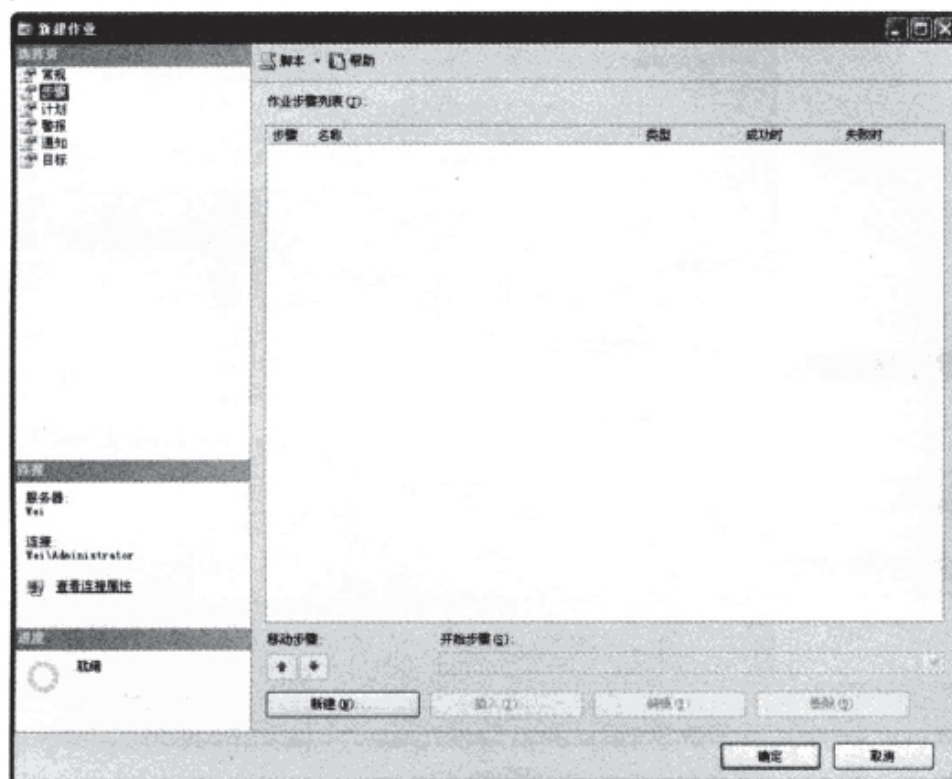


图 22-4

要向作业中添加一个新步骤，只需要单击“新建”按钮并在新对话框中填写相关内容即可，如图 22-5 所示。这里将会使用一个 T-SQL 语句来引发一个伪造的错误，这样在计划该作业的时候就能够知道确实有事情发生了。但是，需要注意的是在命令框的左边有一个“打开”按钮——可以使用这个按钮来导入存储在文件中的 SQL 脚本。

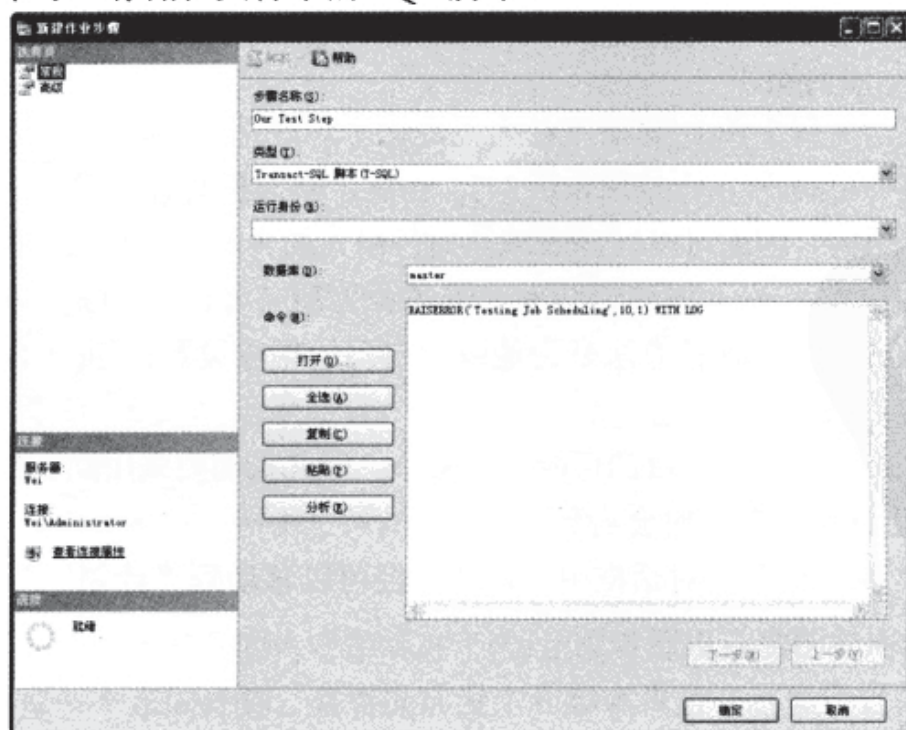


图 22-5

接着继续并移到对话框中的“高级”选项卡上,如图 22-6 所示——在这里才真正开始看到作业计划管理器提供的某些非常棒的功能。

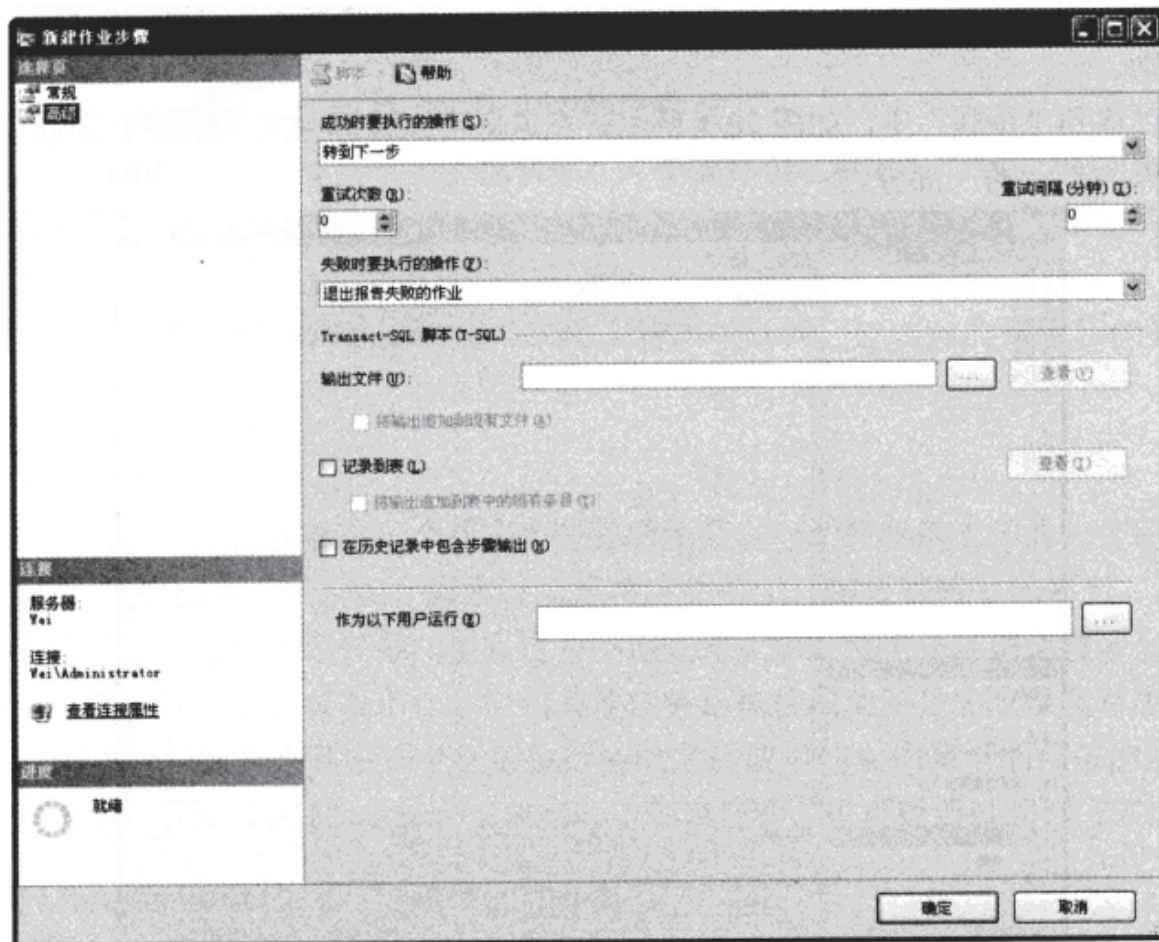


图 22-6

在这个对话框中需要注意几件事情:

- 可以设置作业在任务执行失败后指定的时间间隔自动重新尝试运行。
- 可以选择在作业执行成功或失败之后采取的动作。对于每种结果(成功或失败)可以选择:
 - 退出并报告成功
 - 退出并报告失败
 - 继续下一步的操作
- 可以把结果输出到文件中(对于审计来说,这个功能是非常好的)。
- 可以模拟另一个用户(出于权限的目的)。注意,必须拥有该用户的权限。由于我们是以 sysadmin 登录的,因此可以以 dbo 或其他任何人的身份来运行这个作业。但是普通用户可能最多只能使用 guest 帐号(除非他们是数据库所有者)——但是,嗨,在大多数情况,普通用户不应该使用这种方法来为他或她自己的作业设置计划(让客户端应用程序提供那个功能)。

好, RAISERROR 语句执行失败的可能性非常小,因此只需要采用默认的“退出作业并报告错误”即可(在本章后面介绍备份时会看到其他选择)。

接着会回到主“新建作业”对话框中,现在已经可以移动到“计划”节点上了,如图 22-7 所示。

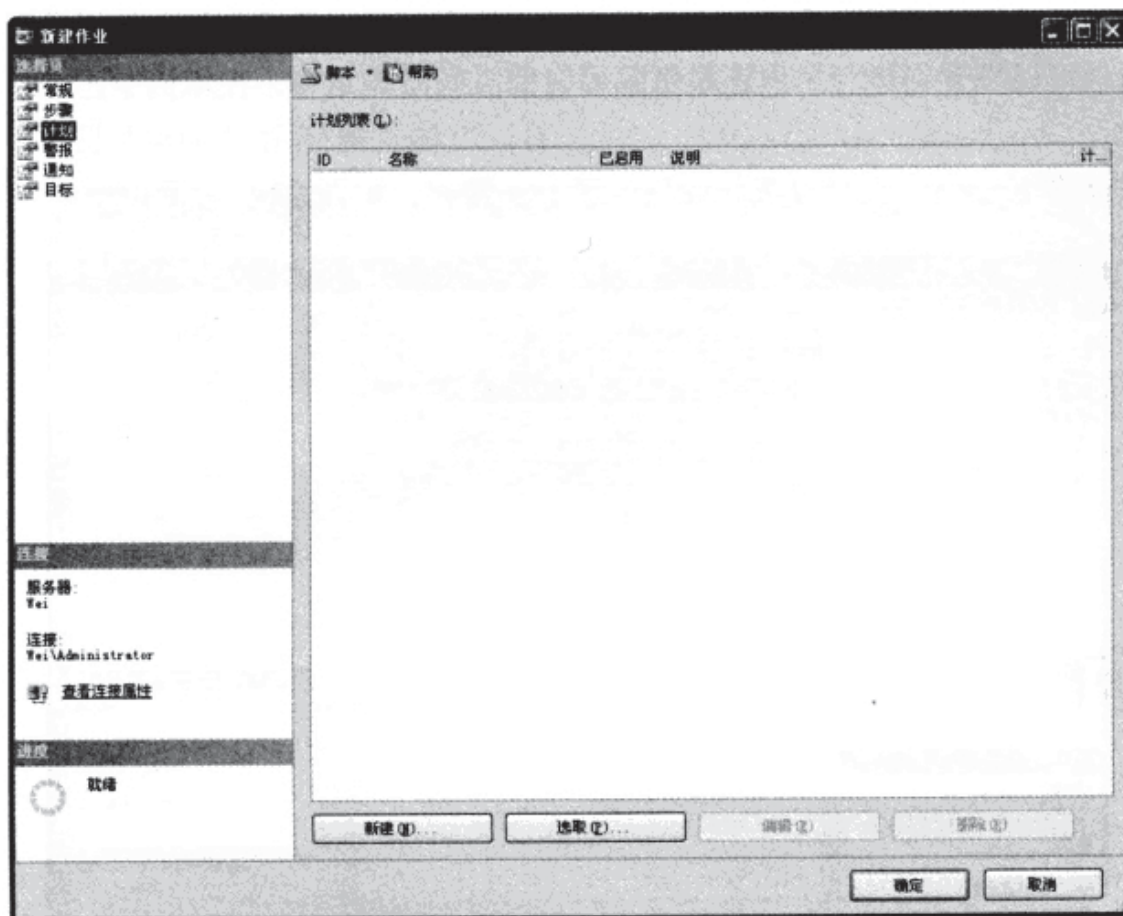


图 22-7

在这个对话框中可以为作业的运行管理一个或多个计划时间。要真正为作业运行创建一个新的计划时间，需要单击“新建”按钮，接着会出现另一个对话框，如图 22-8 所示。

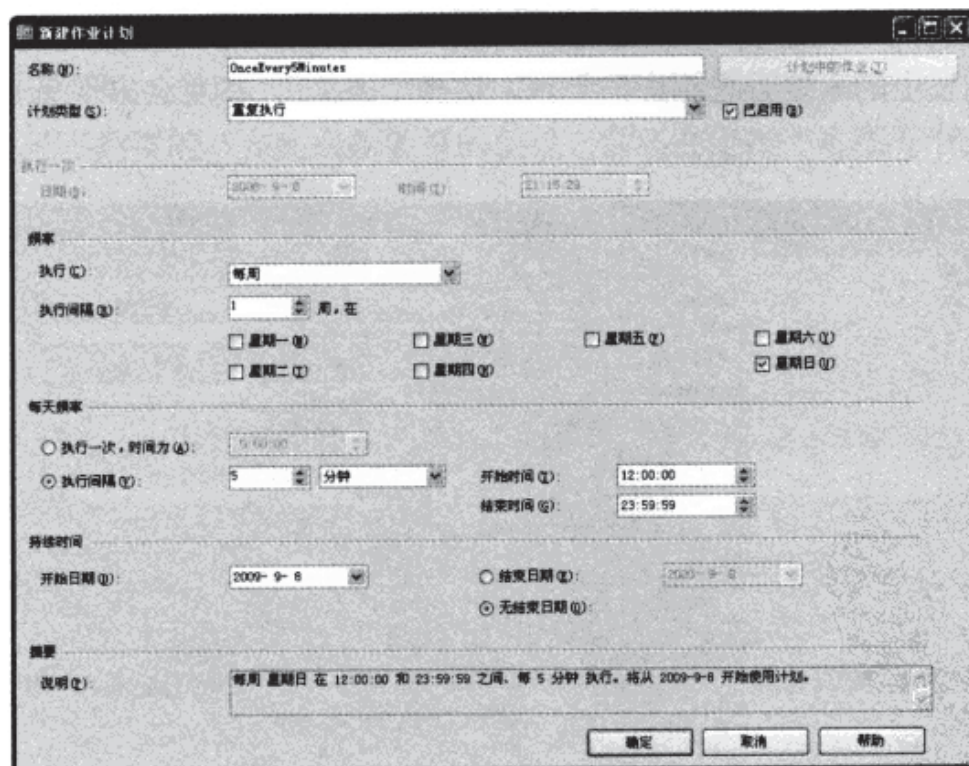


图 22-8

笔者已经把对话框中大部分的内容都填好了(防止读者湮没在截图的海洋中)，在这个对话框中将会为该作业创建一个新的计划，是否循环发生和频率也是在这里设置的。

这里的频率有些令人混淆，因为描述它的言辞有些奇怪。如果想要一个作业每天运行多次，

那么需要把该作业设置为“每天频率”——每一天。这看起来好像这个作业每天只运行一次，但是还有一个选项可以设置它运行一次还是每隔固定的时间间隔运行。在本例中，该作业将会每 5 分钟运行一次。

现在可以移到下一个节点来设置作业属性了——警报，如图 22-9 所示。

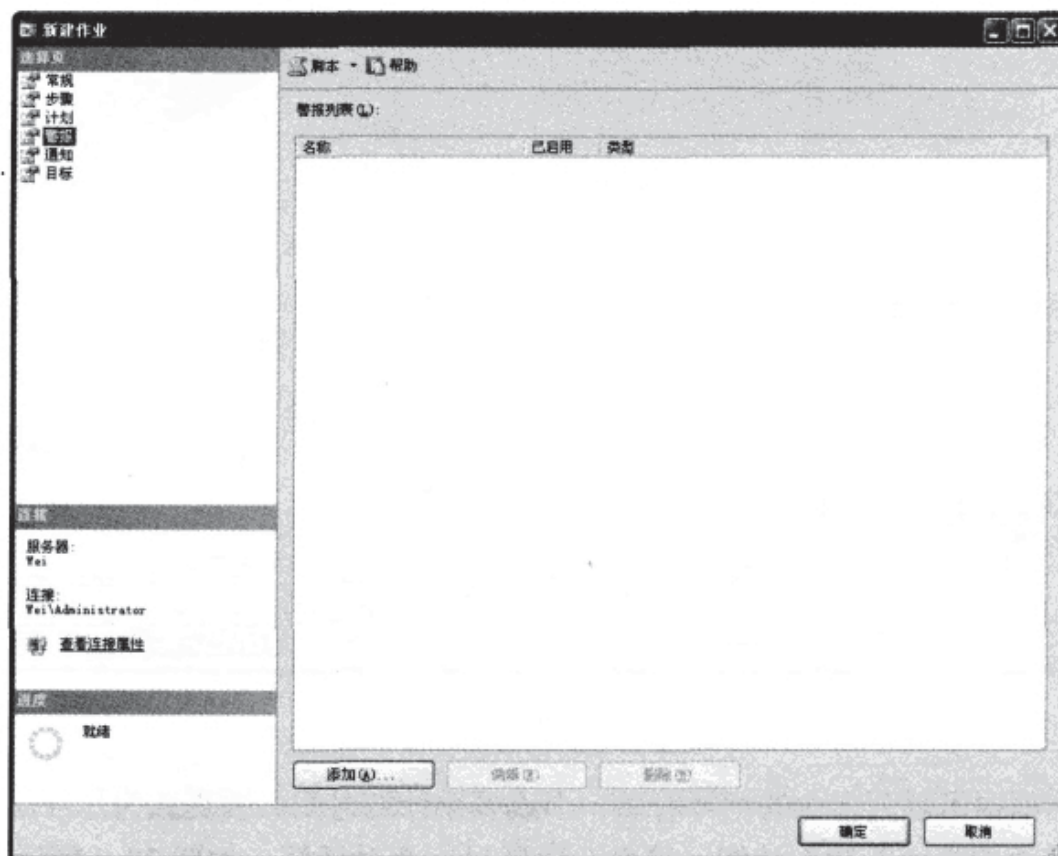


图 22-9

在这可以根据发生的事情设置相应的警报。选择“添加”，接着会出现另一个内容丰富的对话框，如图 22-10 所示。

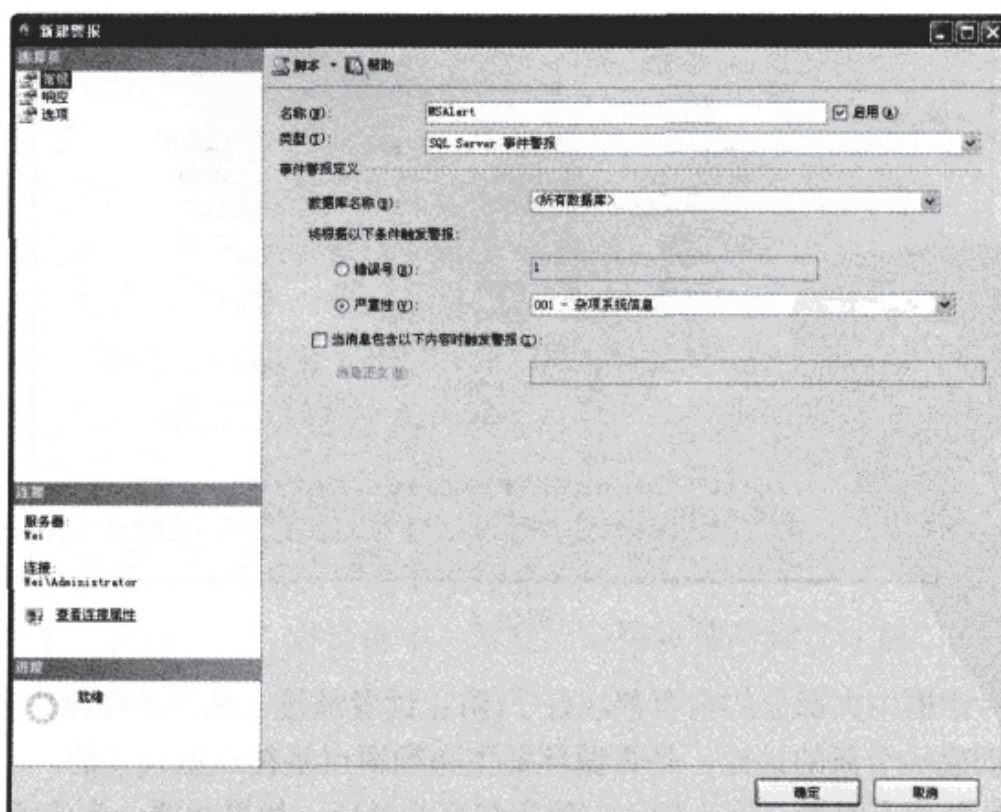


图 22-10

第一个节点——“常规”——允许填写某些基本信息。例如，可以把这个通知限制在一个特定的数据库上。还可以定义在警报被触发之前条件的严重程度(即错误的严重性)。

接着将进入到“响应”节点(参见图 22-11)。

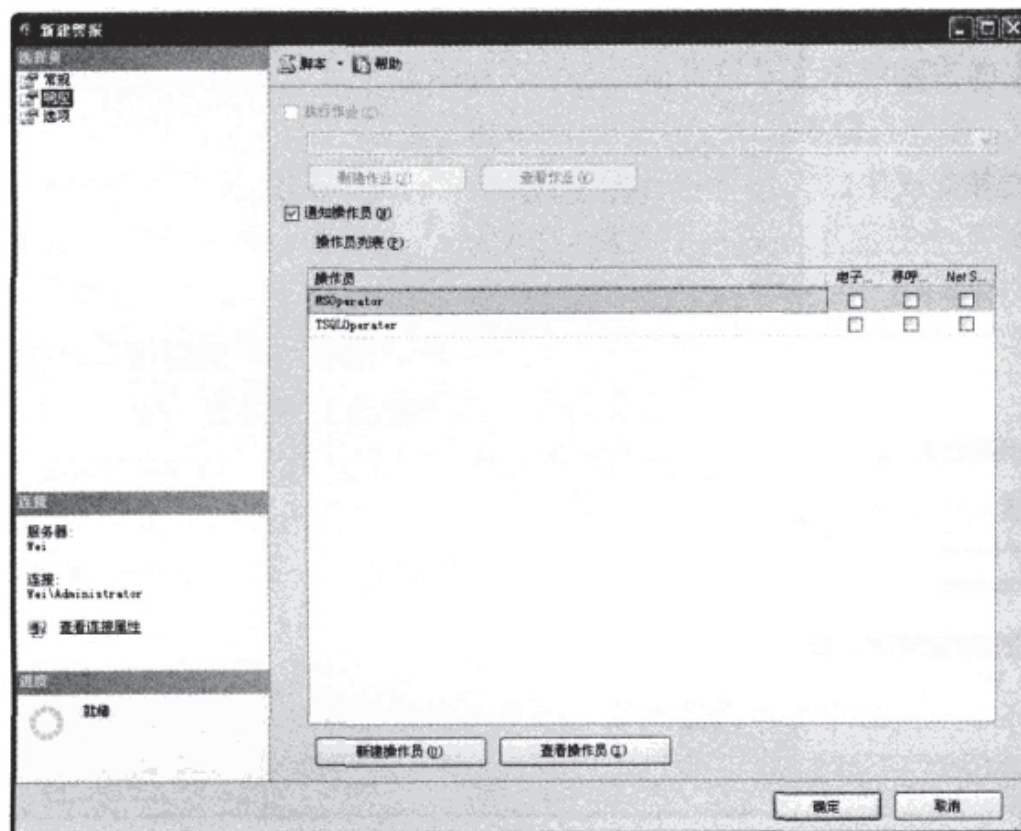


图 22-11

注意这里可以选择本章之前创建的两个操作员之中的任何一个(在本例中将会一直使用通过 Management Studio 创建的操作员)。通过这些操作员的定义, SQL Server 代理将会知道把通知发到哪个电子邮件地址或网络地址。还要注意的是在右边可以控制通知操作员的方式。

最后,但不是最不重要的,在完成新警报的创建之前还会经过“选项”节点(参见图 22-12)。

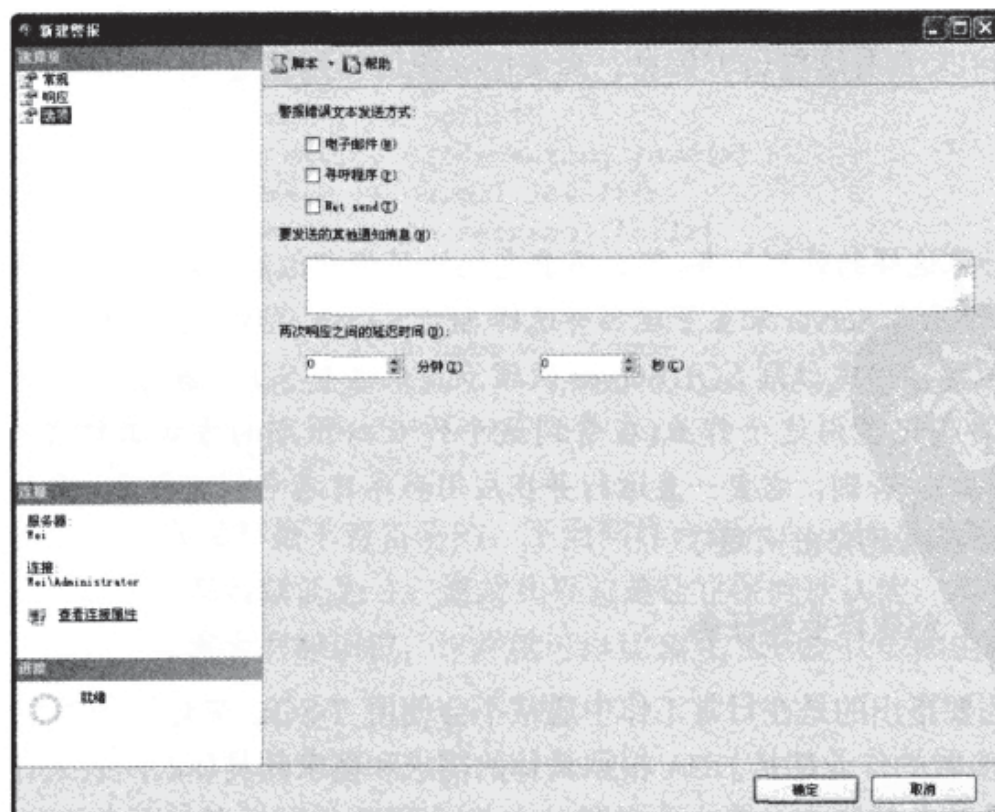


图 22-12

在新警报被创建之后就可以回到主“新建作业”对话框的“通知”节点上了(参见图 22-13)。

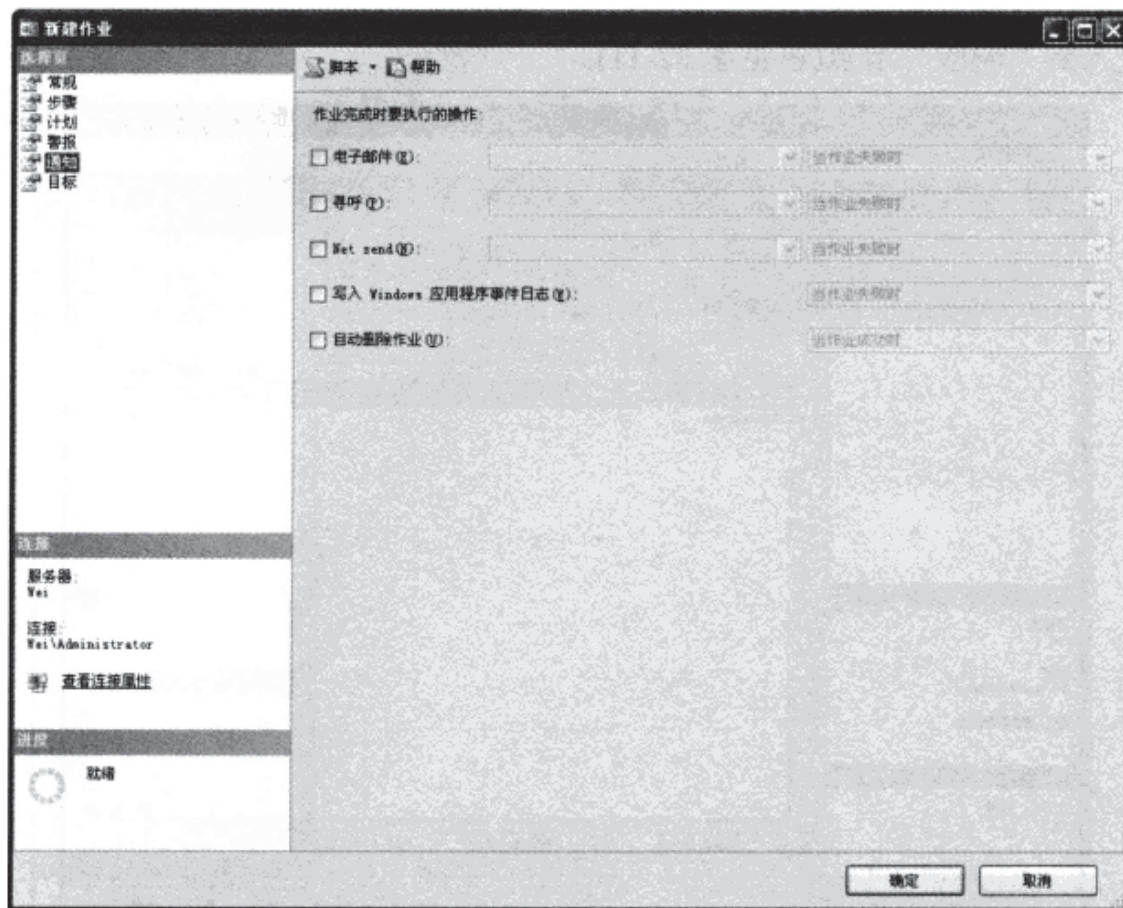


图 22-13

这个窗口允许绕过较老的警报模型来为这个作业定义一个特定的响应——这里将只使用已经设置好的内容，但是读者可以在这个对话框中定义具体的附加通知。

到现在为止可以单击“确定”并退出这个对话框了。在任务被触发之前需要等待几分钟时间，但是在 Windows 事件日志中读者应该可以看到每隔 5 分钟出现的日志条目。读者可以通过系统中的计算机管理工具中的事件查看器(根据读者所运行的 Windows 的版本不同，该工具所在的位置可能也会有些不同)来查看这些日志信息。需要把视图切换到使用应用程序日志(在 Windows 日志下)。

提示：

如果想要运行像这样的计划任务，那么不要忘记为了使它们能够被执行，需要运行 SQL Server 代理。可以通过运行 SQL Server 配置管理器并选择 SQL Server 代理服务或者在 Management Studio 的“对象资源管理器”中定位到 SQL Server 代理节点来检查 SQL Server 代理的状态。

同样，也不要忘记禁用这个作业(在看到这个作业以预期的方式工作之后在 Management Studio 中右击该作业)。否则，它会一直运行并在应用程序日志中创建条目。最终，应用程序日志将会被填满，然后系统就会出问题了。

2. 使用 T-SQL 创建作业和任务

在开始之前需要指出的是在日常工作中通常不会使用 T-SQL 来做这种事情(创建计划作业和任务)。大多数的作业都是 DBA 根据具体的需求和要求的具体时间表来计划的。如果读者并不需要创建任务安装的脚本，那么可以跳过本节了(需要学习很多并不会用到的内容)。也就是

说,只有在最终用户身边没有 DBA 时(例如,小公司通常什么都没有,甚至连稍微像 DBA 一点的人都没有)才会想要利用脚本来执行某些作业以帮助这些一无所知的用户。

在安装过程中常常会忽视把特定作业的创建过程自动化——特别是对于打包软件。如果读者做的是某种形式的顾问工作,或者在私人的 IS 商店环境中工作,那么在安装时很有可能就需要为所有需要的任务设置计划。但是,对于打包软件而言,通常根本无法控制它的安装过程——实际上,你可能离安装有十万八千里远,甚至可能不知道发生了什么事情。

那么如何来确保基本任务(如备份)已经被完成了呢?可以把它们作为安装过程的一部分。

可以使用 T-SQL 把作业添加到 SQL Server 中,这是通过使用下面三个不同的存储过程来实现的:

- `Sp_add_job`: 它创建了真正的作业。
- `Sp_add_job_step`: 它创建了作业中的一个任务。
- `Sp_add_jobschedule`: 它确定了作业什么时候会运行。

与 Management Studio 中的不同选项卡类似,每个存储过程都构建计划任务的整个执行过程的一部分。下一节将分别介绍每一个存储过程。

注意:

所有作业和任务都是存储在 `msdb` 数据库中的。因此在调用其中任何一个存储过程时需要确保 `msdb` 数据库是当前数据库(使用 `USE` 命令)。

`sp_ad_job`

它创建了层次中的最顶层并确定了谁将拥有这个作业以及如何处理通知。它有相当多的参数,但是大多数都是相当容易理解的:

```
sp_add_job [@job_name =] '<job name>'
    [,[@enabled =] <0 for no, 1 for yes>]
    [,[@description =] '<description of the job>']
    [,[@start_step_id =] <ID of the step you want to start at>]
    [,[@category_name =] '<category>']
    [,[@category_id =] <category ID>]
    [,[@owner_login_name =] '<login>']
    [,[@notify_level_eventlog =] <eventlog level>]
    [,[@notify_level_email =] <email level>]
    [,[@notify_level_netsend =] <netsend level>]
    [,[@notify_level_page =] <page level>]
    [,[@notify_email_operator_name =] '<name of operator to email>']
    [,[@notify_netsend_operator_name =] '<name of operator for network message>']
    [,[@notify_page_operator_name =] '<name of operator to page>']
    [,[@delete_level =] <delete level>]
    [,[@job_id =] <job id> OUTPUT]
```

同样,大多数参数的含义都是不言自明的,下面同样介绍一些比较难以理解的参数:

- `@start_step_id`: 该参数默认为 1,通常几乎总是让它保持默认值。本节稍后将会添加步骤,但是那些步骤将会有标识符,这样就可以让 SQL Server 代理知道作业是从哪个步骤开始了。

- **@category_name**: 这个参数与 Management Studio 中看到的类别是直接等同的。它通常为 none(在这种情况下参见 @category_ID), 但是也可以是数据库维护(另一种常见的选择)、全文、Web 助理、复制或使用 sp_add_category 自行添加的类别。
- **@category_id**: 这仅仅是一种不依赖于特定语言来提供类别的方式。如果不希望指派任何特定的类别, 那么建议使用这个选项替代名称, 并指定一个或 0(无类别, 本地运行)或 1(无类别, 多服务器)的值。
- **@notify_level_eventlog**: 对于每种类型的通知, 该参数决定在何种条件下发出通知。然而, 要使用这个存储过程, 需要提供一些常量值来指示何时发生通知。表 22-2 列出了这些常量值:

表 22-2

常 量 值	何时发生通知
0	永不
1	当任务成功时
2	当任务失败时(这是默认值)
3	每当任务运行时

- **@job_id**: 这只是一种找到什么作业 ID 被指派到新创建的作业的方式。当创建作业步骤和作业计划时会需要这个值。有关该参数重要的是:
- 记住把该值提取到一个变量中, 这样就可以重用它们了。
- 该变量的类型为 uniqueidentifier, 而不是读者目前更熟悉的其他类型。

提示:

注意所有非级别的“通知”参数都需要操作员的名称。在运行这个存储过程之前需要先创建操作员。

好, 下面创建一个作业来测试该过程。这里要做的是创建一个与在 Management Studio 中创建的几乎一样的作业。

首先需要创建顶层作业。针对通知所做的是在失败时把消息发送到 Windows 事件日志中。如果读者已经设置了 Database Mail, 那么可以随意把它添加到操作员的通知参数中。

```
USE msdb;

DECLARE @JobID uniqueidentifier;

EXEC sp_add_job
    @job_name - 'TSQLCreatedTestJob',
    @enabled - 1,
    @notify_level_eventlog - 3,
    @job_id - @JobID OUTPUT;

SELECT 'JobID is ' + CONVERT(varchar(128), @JobID);
```

现在执行上面的代码, 其结果如下所示:

```
-----
JobID is 83369994-6C5B-45FA-A702-3511214A2F8A
```

```
(1 row(s) affected)
```

注意读者的特定的 GUID 将和这里显示的不同(注意 GUID 在时间和空间上都必须保证是唯一的)。以后可以使用这个值或者使用作业名称来引用该作业(我发现这种方式要容易得多,但是在多服务器环境中可能会产生问题)。

sp_add_jobserver

这是一个比较简单的存储过程。现在已经有了作业,但是还没有指派该作业在什么服务器上运行。读者将会看到可以在一个服务器创建作业,然后根据所做的选择让它运行在一个完全不同的服务器上。

为了设置一个特定的服务器为目标,需要使用一个名为 `sp_add_jobserver` 的存储过程(仍然在 `msdb` 中)。它的语法是本节中将会介绍的所有语法中最简单的一个,如下所示:

```
sp_add_jobserver [@job_id -] <job id> | [@job_name -] '<job name>',
[@server_name -] '<server>'
```

注意,可以提供作业 ID 或作业名称——但不能同时提供两个。

要为作业指定一个目标服务器,需要运行下面的命令:

```
USE msdb;
```

```
EXEC sp_add_jobserver
    @job_name - 'TSQLCreatedTestJob',
    @server_name - "(local)";
```

注意这里只是指出目标服务器为本地服务器,而不管服务器的名称是什么。也可以指定另一个有效的 SQL Server 使它成为目标服务器。

sp_add_jobstep

过程中的第二步是明确告诉作业要做什么事情。到目前为止,本例中所有的东西只是一个外壳。这个作业没有任何可以执行的任务,这实际上让该作业变得毫无用处。但是,事情还有另外的一面——如果不被指派到某些作业中,步骤甚至无法被创建。

下一步是运行 `sp_add_jobstep`。本质上它会把一个任务添加到作业中。如果希望这个作业执行多个步骤,那么可以多次运行这个特定的存储过程。

其语法如下所示:

```
sp_add_jobstep [@job_id =] <job ID> | [@job_name =] '<job name>']
    [,[@step_id =] <step ID>]
    [,[@step_name =] '<step name>']
    [,[@ subsystem =] '<subsystem>']
    [,[@command =] '<command>']
    [,[@additional_parameters =] '<parameters>']
    [,[@cmdexec_success_code =] <code>]
    [,[@on_success_action =] <success action>]
    [,[@on_success_step_id =] <success step ID>]
    [,[@on_fail_action =] <fail action>]
    [,[@on_fail_step_id =] <fail step ID>]
    [,[@server =] '<server>']
```



```
[,[@database_name =] '<database>']
[,[@database_user_name =] '<user>']
[,[@retry_attempts =] <retry attempts>]
[,[@retry_interval =] <retry interval>]
[,[@os_run_priority =] <run priority>]
[,[@output_file_name =] '<file name>']
[,[@flags =] <flags>]
```

这里其含义不言自明的参数不是太多，因此下面看一下列表中更容易混淆的参数：

- **@job_id** 和 **@job_name**：这实际上是一个相当奇怪的存储过程，因为它要求提供前两个参数中的一个，但不能同时提供两个。可以通过作业的 GUID(在上一个存储过程运行时保存下来的)或作业名称来把该步骤附到作业中。
- **@step_id**：任何作业中的所有步骤都有 ID。SQL Server 会在插入步骤的时候自动为这些步骤指派 ID。既然它是自动指派的，那为什么还需要这个参数呢？那是为了处理要在一个作业的中间插入一个新步骤的情形。如果在作业中已经有了数字 1-5，接着插入一个新步骤并设置步骤 ID 为 3，那么新步骤将会被指派到 3 号位置上，之前的第 3 个步骤将会被移到第 4 号位置上，后续每个步骤都会被加上 1 以腾出空间容纳前一个步骤。
- **@step_name**：顾名思义——该特定任务的名称。注意该参数没有默认值，必须要提供步骤名称。
- **@ subsystem**：它是同作业类别紧密捆绑起来的，并确定 SQL Server 中的哪个子系统(如复制引擎或命令行——命令提示符——或者集成服务)负责执行该脚本。默认是运行一组 T-SQL 语句。可能的子系统如表 22-3 所示：

表 22-3

子 系 统	说 明
ACTIVESCRIPTING	脚本引擎(VB 脚本)。注意这个已经不推荐使用，Microsoft 以后会从产品中删除它
ANALYSISQUERY	分析服务查询(MDX、DMX)
ANALYSISCOMMAND	分析服务命令(XMLA)
CMDEXEC	提供在命令(DOS)提示符中运行已编译的程序或批文件的能力
DISTRIBUTION	复制分发代理
'Dts'	集成服务包执行
LOGREADER	复制日志读取代理
MERGE	复制合并代理
'PowerShell'	PowerShell 脚本
'QueueReader'	复制队列读取代理作业
SNAPSHOT	复制快照代理
TSQL	T-SQL 批处理，这是默认值

- **@command**：这是发送到具体子系统的实际命令。在本例中将是 RAISERROR 命令，就像使用 Management Studio 发出的命令一样，但是它几乎可以是任意 T-SQL 命令。这里真正酷的是可以在命令中使用一些系统提供的值。可以根据需要将它们放置在脚本中，它们在运行时会被替换(本例中将会用到)。可能的系统提供的值如表 22-4 所示：

表 22-4

标 签	说 明
A-DBN	替代数据库名称
A-SVR	使用服务器名称替代标签
A-ERR	错误号
A-SEV	错误严重程度
A-MSG	错误的消息文本
DATE	提供当前日期(采用 YYYYMMDD 格式)
INST	提供当前 SQL Server 实例的名称(默认实例的名称为空白)
JOBID	提供当前作业的 ID
MACH	当前计算机的名称
MSSA	主 SQL Server 代理的名称
OSCMD	运行 CmdExec 步骤的程序
SQLDIR	SQL Server 的 安 装 目 录 (通 常 为 C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL).
STEPCT	该步骤被执行的次数(不包括重试)。可以使用它来记录执行的次数, 并强制停止多步骤的循环
STEPID	步骤 ID
SVR	在其中运行作业的计算机的名称, 如果适用, 还将包括 SQL Server 实例名
TIME	当前时间, 采用 HHMMSS 格式
STRTTM	作业的开始时间, 采用 HHMMSS 格式
STRTDY	作业的开始日期, 采用 YYYYMMDD 格式

提示:

注意所有这些记号都必须要用括号括起来。这与 SQL Server 2005 RTM 的要求(像 SQL Server 2000 那样要求使用方括号)有些不同。从 SQL Server 2005 SP1 开始, 括号取代了之前的方括号的要求, 同时需要一个转义序列(稍后将会介绍)。

从 SQL Server 2005 SP1 开始, 必须要把之前介绍的那些在 @COMMAND 参数中使用的记号封装在一个转义子句中。值转义函数如表 22-5 所示:

表 22-5

\$(ESCAPE_SQUOTE(记号名))	在记号替换字符串中使用两个单引号替换所有的单个单引号
\$(ESCAPE_DQUOTE(记号名))	在记号替换字符串中使用两个双引号替换所有单个双引号的实例
\$(ESCAPE_RBRACKET(记号名))	在记号替换字符串中使用两个右方括号替换所有的单个右方括号的实例
\$(ESCAPE_NONE(记号名))	只为了提供向后兼容性而存在, 它在进行记号替换时不会转义字符串中的任何字符

- @cmdexec_success_code: 该值是命令解释器在作业运行成功时返回的值(只适用于命令提示符子系统)。默认值为 0。
- @on_success_action 和 @on_fail_action: 该参数指定了在步骤成功或失败时真正要做的事情。记住在作业级别可以定义希望发生的通知, 而在步骤级别则可以定义过程如何继续(或者结束)。对于这个参数, 需要提供下列常量值中的一个, 如表 22-6 所示:

表 22-6

值	说 明
1	成功时退出。它是任务执行成功的默认设置
2	失败时退出。它是失败任务的默认设置
3	转到下一步
4	转到 on_success_step_id 或 on_fail_step_id 定义的步骤

- @on_success_step_id 和 @on_fail_step_id: 如果选择了前表中的第 4 个选项, 那么该参数可以指定下一步运行的步骤。
- @server: 运行该任务的服务器(可以在单个主服务器上运行任务, 也可以在多个目标服务器上运行任务)。
- @database_name: 在任务运行时要设置为当前数据库的数据库名称。
- @retry_interval: 该参数以分钟为单位。
- @os_run_priority: 哈, 这是一个未公开的特性。默认值是正常, 但是可以调整 Windows 所认为的 cmdExec(命令行)计划任务的重要程度。可能的取值如表 22-7 所示:

表 22-7

值	优 先 级
-15	只在空闲时运行
-1 到 -14	逐步增长, 但低于正常
0	正常(这是默认值)
1 到 14	逐步增长并高于正常
15	时间关键的

注意:

这里我情不自禁地想起了以前的 TV 秀“迷失太空”, 想要机器人在说“危险, 维尔罗宾逊, 危险!”不要随意搞乱这些值。如果不熟悉 Windows 线程优先级, 那么我建议尽可能远离它们。特别是, 使用的值越高, 对系统的伤害也就越大——包括导致非常严重的不稳定。如果认为这是最重要的事情, 那么记住这样会削弱了某些事情的重要性, 如操作系统功能——这样做不是很聪明。不要使用这个参数, 除非你真正知道自己在做什么。

- @flags: 这个参数与输出文件参数相关, 并指示覆盖现有的文件或者把输出信息附加到现有的文件中。选项如表 22-8 所示:

表 22-8

值	说 明
0	没有指定选项(目前, 这意味着文件每次都会被覆盖)
2	把信息附加到现有的文件中(如果文件存在)
4	明确覆盖文件

好,现在已经介绍过了参数,下面向不久前创建的作业中添加一个步骤:

```
EXEC sp_add_jobstep
    @job_name = 'TSQLCreatedTestJob',
    @step_name = 'This Is The Step',
    @command = 'RAISERROR
        (''RAISERROR ('''TSQL Task is Job ID
$(ESCAPE_SQUOTE(JOBID)).'',10,1) WITH LOG'',10,1)
        WITH LOG',
    @database_name = 'AdventureWorks2008',
    @retry_attempts = 3,
    @retry_interval = 5;
```

注意需要使用转义函数。没有转义函数(在本例中前面四个函数中的任何一个都可以工作),就不会把 JOBID 当成替换记号来处理,而只是把它当成字符串面值“JOBID”。

从技术上来讲,现在作业应该可以运行了。之所以说“从技术上来讲”是因为还没有为作业设置计划,因此唯一运行该作业的方式是手动让作业运行。下面介绍计划问题,在介绍完这个主题之后就完成任务了。

Sp_add_jobschedule

这是拼图游戏中的最后一块了。现在需要告诉作业什么时候运行。为了完成这个任务,需要使用 `sp_add_jobschedule`,它与本节中介绍的其他所有存储过程一样,只能存储在 `msdb` 数据库中。注意这里将会在这个存储过程中多次提交条目,以便为作业创建多个计划。要记住,太多的作业计划可能会引起很大的混淆,因此要明智地为作业设置计划(例如,如果可以把一个作业设置为每天运行,那么就不要把该作业设置为在一周中的每一天都运行)。

它的语法与之前见过的语法有些类似,但是有几部分是新增的:

```
sp_add_jobschedule
    [@job_id =] <job ID>, | [@job_name =] '<job name>', [@name =] '<name>'
    [,[@enabled =] <0 for no, 1 for yes>]
    [,[@freq_type =] <frequency type>]
    [,[@freq_interval =] <frequency interval>]
    [,[@freq_subday_type =] <frequency subday type>]
    [,[@freq_subday_interval =] <frequency subday interval>]
    [,[@freq_relative_interval =] <frequency relative interval>]
    [,[@freq_recurrence_factor =] <frequency recurrence factor>]
    [,[@active_start_date =] <active start date>]
    [,[@active_end_date =] <active end date>]
    [,[@active_start_time =] <active start time>]
    [,[@active_end_time =] <active end time>]
```

同样,下面看一下其中几个参数:

- **@freq_type**: 定义以下参数所设置的时间间隔的性质。这又是一个使用位标记的参数(尽管一次只应该使用一个)。其中一些选择是非常清楚的,但是有些选择则不够清楚,除非已经了解了 `@freq_interval`(稍后介绍)参数。可能的选择如表 22-9 所示:

表 22-9

值	频 率
1	一次
4	每天
8	每周
16	每月(固定的一天)
32	每月(相对于@freq_interval)
64	在 SQL Server 代理启动时运行
128	在 CPU 空闲时运行

- @freq_interval: 确定作业被执行的确切日期,但是这个值的性质要完全依赖于@freq_type 的设置(参见前一个参数的介绍)。这个参数可能有些混淆,但是记住它要与@freq_type 和@frequency_relative_interval 一起使用。其含义如表 22-10 所示:

表 22-10

Freq_type 值	匹配的 freq_interval 值
1(一次)	未使用
4(每天)	每 x 天运行一次, 其中 x 是频率间隔值
8(每周)	频率间隔是下列中的一个或多个: 1(星期天) 2(星期一) 4(星期二) 8(星期三) 16(星期四) 32(星期五) 64(星期六)
16(每月——固定)	每月在频率间隔指定的确切日期运行
32(每月——相对)	运行的日期为下列选项中的一个: 1(星期天) 2(星期一) 3(星期二) 4(星期三) 5(星期四) 6(星期五) 7(星期六) 8(具体某一天) 9(每个工作日) 10(每周末)
64(在代理启动时运行)	未使用
128(在 CPU 空闲时运行)	未使用

- **@freq_subday_type**: 指定**@freq_subday_interval** 的单位。如果设置的是每天运行, 那么把频率设置为在给定的一天内运行。可能值如表 22-11 所示:

表 22-11

值	说 明
1	在指定的时间
4	每 x 分钟, 其中 x 是频率子日期间隔的值
8	每 x 小时, 其中 x 是频率子日期间隔的值

- **@freq_subday_interval**: 这是作业的相邻两次执行之前的时间间隔, 其单位由 **@freq_subday_type** 指定(上表中的 x)。
- **@freq_relative_interval**: 只有当频率类型为每月(相对)(32)时该参数才可用。如果属于这种情况, 那么这个值将决定在哪一周的具体某一天运行作业或者指示作业在每个月的最后一天运行。可能的值如表 22-12 所示:

表 22-12

值	说 明
1	第一周
2	第二周
4	第三周
8	第四周
16	最后一周或最后一天

- **@freq_recurrence_factor**: 在相邻两次执行期间间隔多少周或多少月。究竟该如何对待这个参数依赖于频率类型的设置, 只有在类型为每周或每月(固定或相对)的时候, 该参数才可用。它是一个整型值, 例如, 如果频率类型为 8(每周), 并且频率重复因子为 3, 那么该作业将会在每个第三周的指定日期被运行。

提示:

所有这些参数的默认值为 0。

好, 就像在 Management Studio 中所做的那样, 下面把作业计划设置为每 5 分钟运行一次:

```
EXEC sp_add_jobschedule
    @job_name = 'TSQLCreatedTestJob',
    @name = 'Every 5 Minutes',
    @freq_type = 4,
    @freq_interval = 1,
    @freq_subday_type = 4,
    @freq_subday_interval = 5,
    @active_start_date = 20080731;
```

现在如果在 Management Studio 中看一下该作业, 那么将会发现有一个与直接使用

Management Studio 创建的作业完全一样的作业(除了名字之外)。现在已经通过 T-SQL 完整地实现了该作业。

3. 维护和删除作业和任务

在 Management Studio 中维护作业是相当简单的。只需要双击该作业，然后就像创建一个新作业那样编辑它即可。在 Management Studio 中删除作业和任务则更加简单。只需要选中该作业，然后按下“删除”按钮即可。确认之后，作业就被删除了。

使用 T-SQL 检查作业的状态、编辑作业和删除作业则需要稍微花点功夫了。但是，好消息是维护作业、任务和计划工作与创建它们的方式相当类似，而删除它们则非常简单。

使用 T-SQL 编辑和删除作业

要编辑或删除前面用 T-SQL 创建的四个步骤中的任何一个，只需要使用(有一个是例外)相应的更新存储过程即可——提供给更新存储过程的信息将会完全替代原先添加存储过程(或以前的更新)所接受的参数信息——或者删除存储过程。其参数同添加存储过程是一样的，如表 22-13 所示：

表 22-13

如果添加存储过程是	那么更新存储过程是	而删除存储过程是
sp_add_job	sp_update_job	sp_delete_job
sp_add_jobserver	无(删除并添加)	sp_delete_jobserver
sp_add_jobstep	sp_update_jobstep	sp_delete_jobstep
sp_add_jobschedule	sp_update_jobschedule	sp_delete_jobschedule

22.2 备份和恢复

在没有任何处理备份和恢复的机制之前，任何数据库驱动的应用程序都不应该被部署或卖给客户。正如笔者可能已经告诉人们至少 1000 次了：你一定会对在已经完成的数据库操作中没有任何种类的可靠备份的数据库操作所占的百分比而感到惊讶。

关于备份有一个简单的规则可以遵循——早做并且常做。根据这个规则要做的事情不仅仅是把一个文件备份到同一个磁盘上，然后就忘记它——需要确保把一个副本移到一个完全分开并且安全的地方(理想情况是非现场)。笔者见过服务器发生火灾的事情(散发出了难闻的恶臭味，就像一场噩梦)。你肯定不希望备份同原始数据在烟雾中一起消失吧。

对于由相对缺乏经验的新手开发的应用程序而言，可能会坚持让客户或现场管理员使用 SQL Server 自己的备份和恢复工具，但是即使这样做，在他们使用时应该准备好提供支持。此外，没有任何借口说不了解客户的需求。

22.2.1 创建备份——也叫做“转储”

在 Management Studio 中为给定的数据库创建一个备份文件实际上是相当容易的。只需要在

“对象资源管理器”中定位要感兴趣的数据库，然后右击。

接着选择“任务”和“备份”，如图 22-14 所示。

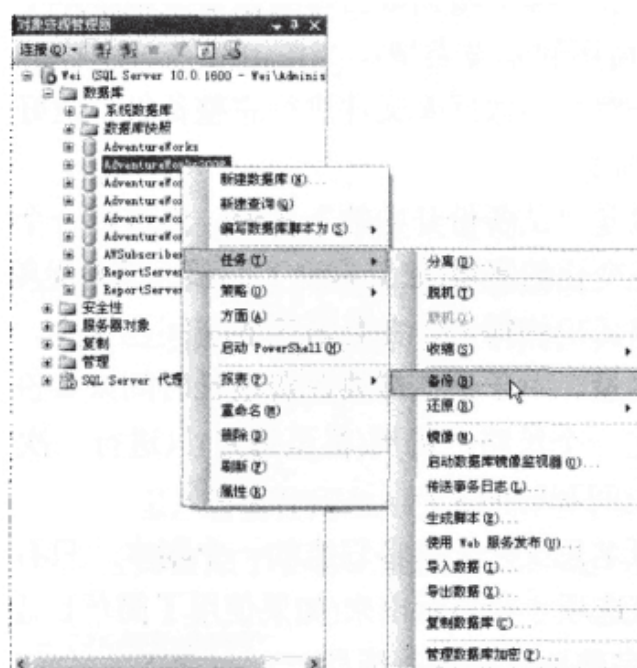


图 22-14

接着将会出现一个对话框，在这个对话框中几乎可以定义备份过程的方方面面，如图 22-15 所示。

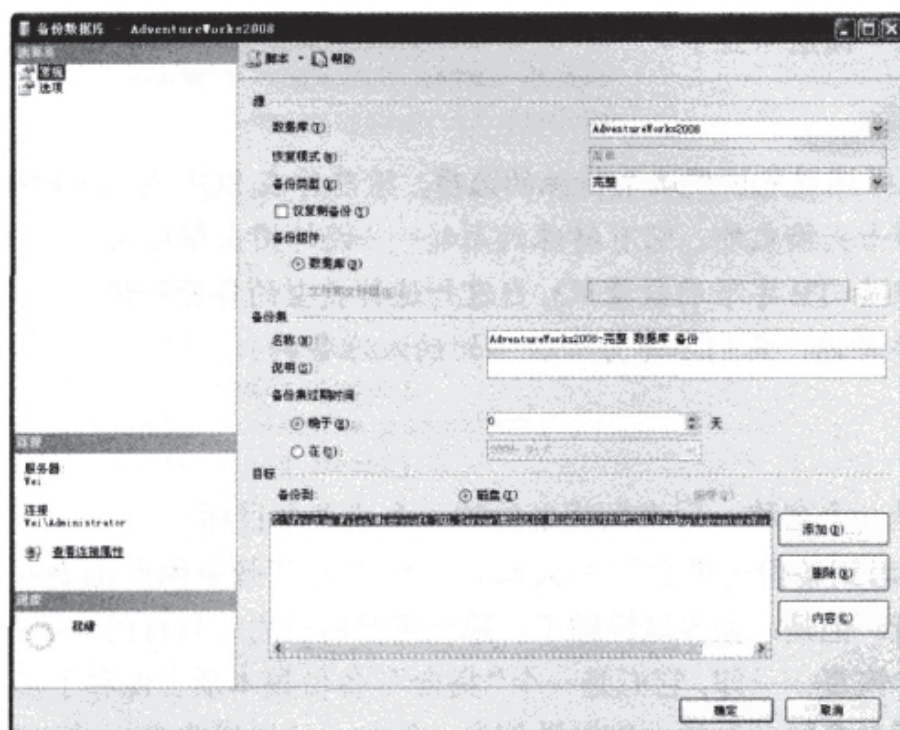


图 22-15

第一个设置的含义是不言自明的。它表明要备份哪个数据库。但是从这里开始，事情就变得有点复杂了。

下面看到的这些项目可能还没有太多的意义，首先是“恢复模式”。这里的“恢复模式”字段只是说明选择备份的数据库的恢复模式是什么。它实际上是一个数据库级别的设置。这里将把对它的讨论稍微推迟一下——当在下一节中介绍备份事务日志的时候将会介绍它。

现在，这些都是简单的部分，下面分析一下剩下的一些可用选项。

1. 备份类型

首先需要做出选择的是备份类型。根据数据库的恢复模式(同样, 请耐心一点, 稍后会介绍这个主题!), 存在两种或三种可用的备份类型:

- **完整:** 顾名思义——对实际数据库文件进行完整备份, 就好像它是在发出备份命令之前提交了最后一个事务时一样。
- **差异:** 这可以被看成是“从备份开始的”备份。当进行一个差异备份时, 它只将自上一次完整备份依赖发生变化的区域(如果忘记了, 那么请参见第 6 章)写入到副本中。通常, 它的运行速度比完整备份快得多, 并且所需的空间也更少。但是少多少呢? 这要依赖于实际发生变化的数据量。对于那些需要花费很长时间来备份的大型数据库来说, 一种非常常见的做法是制定一个策略, 每周(甚至每月)只进行一次完整备份, 然后在此期间进行差异备份以节省空间和时间。
- **事务日志:** 同样, 顾名思义——事务日志的一个副本。只有当数据库被设置为完整或大容量日志模式时, 该选项才会显示出来(如果使用了简单日志模式, 那么该选项将会被隐藏)。通常, 稍后将完整地介绍这些模型。

备份类型的一个子话题是备份组件, 它只适用于完整和差异备份。

就本书而言, 应该把精力主要集中在备份整个数据库上。也就是说, 读者将会注意到另一个选项“文件和文件组”。第 1 章中曾经简要地介绍过用文件组和单个文件存储数据的思想。这个选项允许只选择一个文件或文件组来参与备份。笔者强烈建议避免使用这个选项, 除非你已经从 SQL Server 用户的“专家”课堂毕业了。

提示:

同样, 笔者想要强调避免使用这个特殊的选项, 除非你在 SQL Server 的备份方面已经具备了相当的知识, 除了博士头衔之外。它有特殊的用处——设计用来帮助用于高可用性场景中的非常大的数据库安装(例如 TB 字节的数量级)。当进行这种类型的备份和使用它来进行还原时存在重要的一致性问题的需要考虑, 它们不是为心脏不好的人准备的。

2. 备份集

备份集基本上是一个名称, 用来指代备份的一个或多个目标。

SQL Server 考虑到备份可能会特别大或者可能因为某些原因而需要在多个设备上备份——驱动器或磁带。但是, 如果这样做了, 那么需要所有用作目标的设备都可用才能从它们之中的任意一个设备中恢复——即, 它们是一个“集合”。备份集本质上保存了参与特定备份的目标。此外, 一个备份集还包含了备份的一些属性信息。例如, 可以确定备份的失效日期。创建一个备份集非常容易, 就是在定义备份时指定多个文件或磁带为目标。

3. 目标

这是数据将要被备份到的目的地。在这里可以定义几个目标, 而一个备份集可能会使用这些目标。对于大多数的安装来讲, 这将是一个文件位置, 但是它也可以是任意一个有效的 UNC 路径(它可能不是一个磁盘, SQL Server 并不关心这个, 只要它是一个有效的存储设备即可)。

4. 选项

除了在对话框的“常规”节点中见过的项目之外，还有一个节点可以设置其他杂项。其中大多数选项都是不言自明的。但是需要特别指出的是“事务日志”区域。

5. 计划

在设置了所有选项之后，如果能够创建一个作业来定期运行备份不是很好吗？对话框上面的“计划”按钮使得做这件事情变得非常容易。单击它之后会出现一个在本章前面见过的“作业计划”对话框。接着可以定义一个规律的时间表来运行这个刚刚定义的备份。

6. 使用 T-SQL 备份

要在 T-SQL 中备份数据库或日志，需要使用 BACKUP 命令。BACKUP 的语法的工作方式几乎是一样的，但不是完全一样，这要依赖于备份的是数据库还是日志。其语法如下所示：

```
BACKUP DATABASE|LOG <database name>
    {WITH
        NO_LOG|TRUNCATE_ONLY}
| TO {DISK|TAPE} <backup device(s)> [,...n]
[MIRROR TO <backup device(s)> [, . . . n]]
[WITH
[BLOCKSIZE = <block size>]
[[,] CHECKSUM | NO CHECKSUM ]
[[,] COMPRESSION | NO COMPRESSION]
[[,] STOP_ON_ERROR | CONTINUE_AFTER_ERROR]
[[,] DESCRIPTION = <description of backup>]
[[,] DIFFERENTIAL]
[[,] EXPIREDATE = <expiration date> | RETAINDAYS = <days>]
[[,] PASSWORD = <password>]
[[,] FORMAT|NOFORMAT]
[[,] INIT|NOINIT]
[[,] MEDIADESCRIPTION = <description>]
[[,] MEDIANAME = <media name>]
[[,] MEDIAPASSWORD = <media password>]
[[,] NAME = <backup set name>]
[[,] REWIND|NOREWIND]
[[,] NOSKIP|SKIP]
[[,] NOUNLOAD|UNLOAD]
[[,] RESTART]
[[,] STATS [= <percentage>]]
[[,] COPY_ONLY]
```

下面看一下其中几个参数：

- **<backup device>**：没错，可以备份到多个设备上。它将创建一个所谓的媒介集，当媒介分布在几个磁盘上时能够加快备份的速度，因为它可以并行加载，这样就不会被单个设备的 I/O 限制所约束了。但是，要注意——在还原这种类型的备份时整个媒介集都要可用。

提示：

还需要注意的是 TAPE 选项的存在只是为了提供向后兼容性——对于 SQL Server 来说，现在所有的备份好像都是 DISK(即使实际的设备恰好是磁带)。

- **BLOCKSIZE**: 在使用硬盘驱动器的备份中, 这是自动确定的, 但是使用磁带则需要提供正确的块大小。这个参数需要寻求厂商的帮助。
- **COMPRESSION**: 顾名思义, 指示是否在备份中使用压缩。默认是无压缩, 但是可以在服务器范围级别上修改这个参数。
- **DIFFERENTIAL**: 它用来进行差异备份。差异备份只备份自上一次完整备份以后发生变化的数据。任何日志或其他差异备份都会被忽略。自上一次完整备份以后任何被修改、添加或删除的行/列都会包含在新备份中。差异备份的优点是在创建备份时要比完整备份快得多, 而在还原时要比应用每一个日志快得多。
- **EXPIREDATE/RETAIN DAYS**: 可以让备份媒介在一段特定的时间之后失效。这样做是为了让 SQL Server 知道什么时候可以覆盖较老的媒介了。
- **FORMAT/NOFORMAT**: 确定是否需要重写媒介头(磁带要求重写)。要注意格式化将会影响到整个设备——这意味着为设备上的一个备份进行格式化将会破坏设备上的其他所有备份。
- **INIT/NOINIT**: 重写设备上的数据, 但是不要改变头。
- **MEDIA DESCRIPTION** 和 **MEDIA NAME**: 描述并命名该媒介——描述信息最多可以有 255 个字符, 而名称最多可以有 128 个字符。
- **SKIP/NOSKIP**: 确定是否需要注意磁带上以前的备份的失效信息。如果启用了 **SKIP**, 那么失效信息将会被忽略, 这样就可以覆盖磁带了。
- **UNLOAD/NOUNLOAD**: 只适用于磁带。该选项决定在备份完成之后是倒带并弹出磁带(**UNLOAD**)还是让它保持在当前位置(**NOUNLOAD**)。
- **RESTART**: 在前一个被中断的备份所留下的位置上继续。
- **STATS**: 在备份运行时显示一个进度条以指示进度。
- **COPY_ONLY**: 创建一个备份, 但不会影响其他所有以任何方式创建的备份序列。例如, 采用差异备份的日志将会继续下去, 就好像对备份的复制从来没有发生一样。

下面来尝试创建真正的备份:

```
BACKUP DATABASE AdventureWorks2008
TO DISK - 'C:\Program Files\Microsoft SQL
Server\MSSQL10.MSSQLSERVER\MSSQL\Backup\TSQLDataBackup.bck'

WITH
    DESCRIPTION - 'My what a nice backup!',
    STATS;
```

提示:

高亮显示的代码应该放在一行中。

注意, 读者需要根据自己的特定安装的具体情况修改相应的路径。

现在已经为 AdventureWorks2008 数据库创建了一个备份。

SQL Server 在处理备份的过程中甚至还会友好地提供进度消息:

```
10 percent processed.
20 percent processed.
30 percent processed.
```

```
40 percent processed.
50 percent processed.
60 percent processed.
70 percent processed.
80 percent processed.
90 percent processed.
Processed 25448 pages for database 'AdventureWorks2008', file
'AdventureWorks2008_Data' on file 1.
Processed 36 pages for database 'AdventureWorks2008', file 'FileStreamDocuments'
on file 1.
Processed 1 pages for database 'AdventureWorks2008', file 'AdventureWorks2008_Log'
on file 1.
100 percent processed.
BACKUP DATABASE successfully processed 25484 pages in 10.825 seconds (18.391 MB/sec).
```

就是这么简单，下面进行一个简单的日志备份：

```
BACKUP LOG AdventureWorks2008
TO DISK - 'C:\Program Files\Microsoft SQL
Server\MSSQL10.MSSQLSERVER\MSSQL\Backup\TSQLLogBackup.bck'
```

```
WITH
    DESCRIPTION - 'My what a nice backup of a log!',
    STATS;
```

提示：

高亮显示的代码应该放在一行中。

注意：

值得注意的是当把数据库的恢复模式设置为“简单”时无法对日志进行备份。要把它修改为一个不同的恢复模式，只需要右击 AdventureWorks2008 数据库，然后选择“属性”和“选项”选项卡——在 T-SQL 中可以使用 `sp_dboption` 系统存储过程。读者如果仔细思考一下会发现这是有意义的，因为这样任何已提交事务始终不考虑日志的备份。

还值得注意的是当有用户正在使用数据库时也能够进行备份。SQL Server 能够调节这些变更，因为它可以在日志中了解备份开始运行的确切时间，这样就可以使用这个时间作为参考点来进行余下的备份工作。

22.2.2 恢复模型

好，笔者在上一节中花了好多口舌来保证一定会讨论它们，现在是时候问了：什么是恢复模型？

在第 11 章中讨论过事务日志。除了跟踪事务以处理事务回滚和数据的原子性之外，事务日志对于把数据恢复到系统故障发生的那个时间点来说也是非常至关重要的。

现在想象一下正在运行一家银行。假设现在正在处理最近 6 个小时中发生的存取款业务——这个时间是从上一次进行完整备份到现在的时间。现在如果系统宕机了，笔者猜测你肯定不想使用昨晚的备份来恢复并失去对在这段时间内发生的钱款出入的跟踪。看看已经到达哪里了？你需要每一个时刻的数据，它们都是有价值的。

保存事务日志能够“向前回滚”自上一次完整或差异备份以来发生的所有事务。假设数据备

份和事务日志都可用，那么就能够恢复到故障发生的那个时间点。

恢复模型决定了什么类型日志记录将被保存以及保存多长时间。存在三个选项：

- **完整：**顾名思义，所有内容都会被记录到日志中。在这种模型下，如果数据的备份可用并且拥有自备份以来所有的事务日志，那么当系统发生故障时将不会造成数据丢失。如果丢失了一个日志，或者有一个日志被损坏了，那么就只能把数据恢复到最近一个完整无缺的可用日志了。但是，记住在接收很多变更或新数据的系统上，这种保存所有内容的方式会占用相当多的空间。
- **大容量日志：**这像是一个“轻量级的完整恢复”。在这种选项中，常规的事务将会被日志记录下来，就好像它们是在完整恢复方式中一样，但是大容量操作不会被日志记录下来。其结果是当系统发生故障时，还原备份会包含所有发生在没有参与大容量操作(如大容量数据的导入或者创建索引)的数据页面上的变更，而所有的大容量操作则必须要重做。关于这个问题的好消息时大容量操作的性能已经比以前好很多了，但是在性能提升的同时也带来了风险，因此路途将是变化多端的…。
- **简单：**在这种模型下，事务日志的存在本质上是为了在事务发生时对它们提供支持。事务日志会有规律地被截断，任何已完成或回滚的事务本质上会从日志中删除(虽然没有这么简单，但是其效果是这样的)。这样日志就会更加紧凑，其规模也较小并且性能也要好一点，但是当需要从系统故障中恢复时，这种日志将没有任何用处。

对于大多数的安装来说，完整恢复是希望为生产级别的数据库设置的模型——故事结束。

22.2.3 恢复

这是备份的一个逆向过程。你已经虔诚地进行了备份，现在想要还原一个——要么是出于恢复的目的，要么仅仅是想要在其他地方创建数据库的一个副本。

一旦有了数据库的备份之后，把它还原到原先的位置就相当容易了。要开始操作——它的工作方式与备份非常相似：定位到想要还原的数据库并右击——然后选择“任务”|“还原”，接着会出现“还原”对话框，如图 22-16 所示。

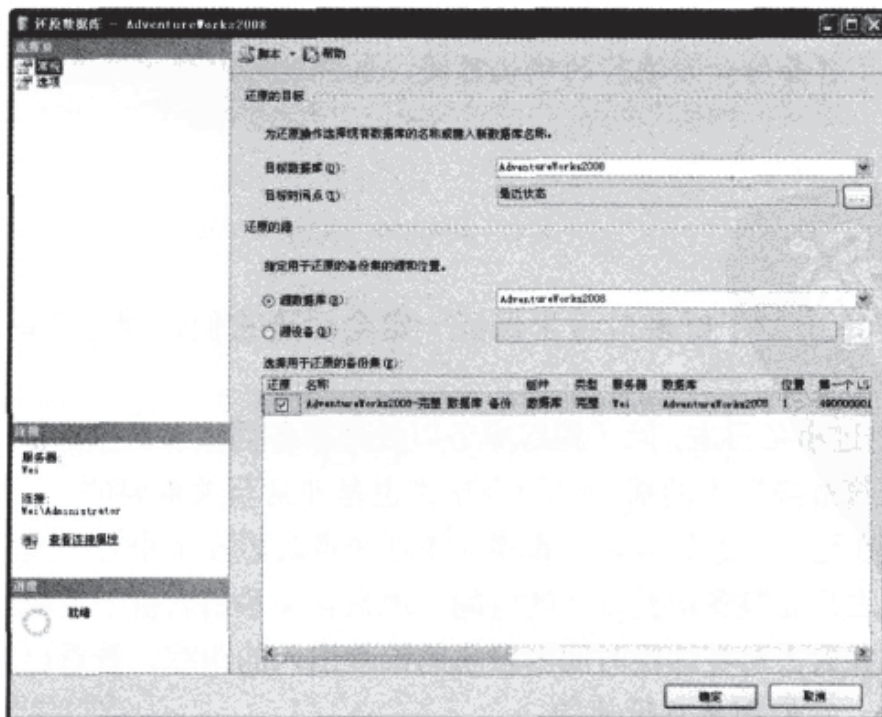


图 22-16

需要做的是选择旧备份并把它放到创建该备份的数据库上,这个过程是相当简单的。只要单击“确定”,数据库将会被还原,不需要任何干涉。

1. 还原到不同的位置

当希望改变要还原的位置时,事情就变得有点复杂了。作为备份过程的一部分,备份会知道正在备份的数据库的名称,并且,也可能更重要,它会知道它将要使用的物理文件的路径。

修改目标数据库的名称非常简单——不是什么大问题——问题是修改目标数据库的名称并不会改变试图存储的物理文件(.MDF 文件和.LDF 文件)。为了解决这个问题,需要进入“还原”对话框的“选项”节点。

同样,这里大多数的选项都是不言自明的,但是需要特别注意“还原为”列。在对话框的这部分中可以替换所有原始文件的目标、位置和名称,这为把一个数据库的多个副本还原到同一台服务器上(可能是出于测试的目的)或者把数据库安装到一个新卷甚至新的系统上提供了一种方式。

2. 恢复状态

这仅仅是在完成还原时希望数据库所处的状态。当还原数据库后仍然存在需要被应用到数据库上的日志时,该选项有特殊的作用。

如果选择默认的选项(等价于在 T-SQL 中使用 WITH RECOVERY 选项),那么当还原操作结束之后数据库会立即处于完整联机的状态。但是如果想要在初始还原结束之后还原其他日志,那么需要选择其他两种选项中的一个。这两个选项都会阻止更新的发生,并把数据库保持在一个可以进行更多的恢复操作的状态。它们之间的差别仅仅在于用户是被允许以“只读”模式访问数据库,还是数据库应该仍然保持脱机状态。

提示:

可用性可能比你想象得还要大。笔者确信有一件事情看起来已经成为大问题了,当还原操作突然把数据库标记为可用时,用户找到进入系统的方式的快速程度确实令人吃惊。尽管知道在当前的还原操作结束之后就“完成任务了”,但是在真正让用户使用系统之前最好还是有机会检查一下数据库。如果情况确实如此,那么确保在还原时使用 NO RECOVERY 方法。你随后可以完全为了设置 WITH RECOVERY 选项而运行一个还原,一旦确信所有的事情都像预期那样之后,就可以让数据库回到完整联机状态了。

3. 使用 T-SQL 还原数据

这里将会使用 RESTORE 命令来还原备份中的数据。其语法如下所示(它存在大量的变体,读者如果需要了解它们之间的细微差别,那么建议阅读专门面向管理的书籍,它们会专门腾出一些章节来介绍备份和恢复):

```
RESTORE DATABASE|LOG <database name>
    [FROM <backup_device> [,...n]]
    [WITH
    [DBO_ONLY]
    [[,] FILE = <file number>]
    [[,] MEDIANAME = <media name>]
    [[,] MOVE '<logical file name>' TO '<operating system file name>'][,...n]
```



```

[[,] {NORECOVERY|RECOVERY|STANDBY = <undo file name>}]
[[,] {NOUNLOAD|UNLOAD}]
[[,] REPLACE]
[[,] RESTART]
[[,] STATS [= percentage]]
[[,] { STOPAT = { <date and time> }
    | STOPATMARK = { '<name of mark>' }
      [ AFTER <date and time> ]
    | STOPBEFOREMARK = { '<name of mark>' }
      [ AFTER <date and time> ]

```

下面看一下其中几个选项：

- **DBO_ONLY**：当还原结束之后，数据库的 `dbo_only` 选项将会被设置为启用。这让 `dbo` 能够在允许用户回到系统之前对数据库进行检查和测试。

注意：

这是一个大问题，笔者强烈建议总是使用这个选项。你会对用户能如此快速地回到系统感到吃惊，即使是系统只恢复了一小段时间。当系统宕机时，你会发现用户非常没有工作耐心。他们会一直不断地尝试登录，并且不厌其烦地问系统是否已经恢复了。当系统运行时，他们会认为已经可以进入系统了。

- **FILE**：可以多次备份到同一个媒介中。这个选项允许选择一个具体的版本来还原。如果没有提供这个选项，那么 SQL Server 会认为想要从最近的备份中还原。
- **MOVE**：允许将数据库还原到一个同数据库一开始被备份时所使用的物理文件不同的物理文件中。
- **NORECOVERY/RECOVERY/STANDBY**：**RECOVERY** 和 **NORECOVERY** 是互斥的。**STANDBY** 与 **NORECOVERY** 是一起工作的。它们按表 22-14 的方式工作：

表 22-14

选 项	说 明
NORECOVERY	还原数据库，但是保持将它标记为脱机。未提交的事务将会被完整无缺地保留下来。这个选项允许继续进行恢复过程——例如，可能仍然存在额外的日志需要应用
RECOVERY	一旦还原命令执行成功之后，数据库将会再次被标记为活跃。数据再次可以被修改，任何未提交的事务将会被回滚。如果没有指定任何选项，那么这个选项是默认选项
STANDBY	STANDBY 允许创建一个撤消文件，这样就可以撤消恢复过程的效果了。STANDBY 允许在发出 RECOVERY 命令之前将数据库设置为只读访问(这意味着至少部分数据库已经被还原了，但是你认为还原过程还没有结束)。这允许用户以只读模式使用系统，同时又可以验证还原过程

- **REPLACE**：覆盖防止在现有数据库上还原的安全特性。
- **RESTART**：告诉 SQL Server 继续执行上一个被中断的还原过程。

下面介绍还原 AdventureWorks2008 数据库的一个示例。不要运行这个语句，除非能够绝对确保备份是成功的并且没有发生任何变化。

首先删除现有的 AdventureWorks2008 数据库:

```
USE master;
```

```
DROP DATABASE AdventureWorks2008;
```

一旦完成之后就可以尝试使用 RESTORE 命令来还原这个数据库了:

```
RESTORE DATABASE AdventureWorks2008  
FROM DISK - 'C:\Program Files\Microsoft SQL  
Server\MSSQL10.MSSQLSERVER\MSSQL\Backup\TSQLEnterprise\AdventureWorks2008.bck'
```

```
WITH  
    DBO_ONLY,  
    NORECOVERY,  
    STATS;
```

提示:

高亮显示的代码应该放在一行内。

这里还原的时候设置了 NORECOVERY 选项,因为还需要做其他一些事情。日志中将会包含在备份数据库或日志到该日志被备份期间发生的所有事务。“应用”这部分日志,这样就可以让数据库尽可能地保持最新:

```
RESTORE LOG AdventureWorks2008  
FROM DISK - 'C:\Program Files\Microsoft SQL  
Server\MSSQL10.MSSQLSERVER\MSSQL\Backup\TSQLEnterprise\AdventureWorks2008_Log.bck'  
WITH  
    DBO_ONLY,  
    NORECOVERY,  
    STATS;
```

注意,如果需要从这样一个设备上应用几个日志,那么指定将要被应用的日志的名称,还需要按照备份它们的顺序来应用它们。

现在本来已经可以启用所有东西了,但是在再次让数据库活跃之前还是稍微等待一会。尽管已经没有任何日志需要应用,但是仍然需要重新运行 RESTORE 语句使得数据库再次活跃:

```
RESTORE LOG AdventureWorks2008 WITH RECOVERY;
```

现在应该能够测试数据库了:

```
USE AdventureWorks2008;
```

```
SELECT * FROM Region;
```

的确,返回了预期的结果。实际上,多运行几个 SELECT 语句就会发现数据库已经被恰当地还原了。

做完检查后,记住在所有这些操作中都选择了 DBO_ONLY 选项。如果运行 sp_dboption,将会发现其他人都无法进入系统:

```
EXEC sp_dboption;
```


看看仅供 dbo 使用到内容:

Settable database options:

```
-----
ANSI null default
ANSI nulls
ANSI padding
ANSI warnings
arithabort
auto create statistics
auto update statistics
autoclose
autoshrink
concat null yields null
cursor close on commit
db chaining
dbo use only
default to local cursor
merge publish
numeric roundabort
offline
published
quoted identifier
read only
recursive triggers
select into/bulkcopy
single user
subscribed
torn page detection
trunc. log on chkpt.
```

记住关闭这个选项, 否则用户将无法进入系统:

```
EXEC sp_dboption AdventureWorks2008, 'dbo use only', 'false';
```

现在已经还原了数据库并且数据库已经出于活跃状态了。

22.3 索引维护

在第 6 章中讨论了索引如何产生碎片的问题。随着时间的流逝, 这个问题可能会成为数据库性能的主要障碍, 需要制定一个策略来解决这个问题。幸运的是, SQL Server 提供了命令来重新组织数据和索引使得其结构更加合理。把它同已经学习过的作业计划结合起来, 就能够使碎片整理的例行操作自动化了。

ALTER INDEX 是进行数据库维护的主要工具。比起以前用于维护的主要工具——DBCC——来说, 它更加容易, 却又略显复杂。下面快速介绍一下这个工具, 接着介绍如何为它设置计划。

22.3.1 ALTER INDEX

从 ALTER INDEX 所具有的功能上来看, 其名字是有一定的欺骗性的。到现在为止, ALTER 命令总是用于修改对象的定义。例如, 在对表执行 ALTER 操作的时候会添加或禁用约束和列。

但是 ALTER INDEX 是不同的，它只用于维护，而不会改变结构。如果想要修改索引的构成，那么仍然需要首先执行 DROP 操作，然后再执行 CREATE 操作，或者在执行 CREATE 的时候使用 DROP_EXISTING=ON 选项。

ALTER INDEX 的语法如下所示：

```
ALTER INDEX { <name of index> | ALL }
    ON <table or view name>
    { REBUILD
        [ [ WITH ( <rebuild index option> [ ,...n ] ) ]
          | [ PARTITION = <partition number>
              [ WITH ( <partition rebuild index option>
                        [ ,...n ] ) ] ] ]
    | DISABLE
    | REORGANIZE
        [ PARTITION = <partition number> ]
        [ WITH ( LOB_COMPACTION = { ON | OFF } ) ]
    | SET ( <set_index_option> [ ,...n ] )
    }
[ ; ]
```

这里大部分的内容都属于“高级 DBA 的领域”——通常会以特别的方式使用它来处理非常特殊的问题。但是这里的一些核心元素应该成为常规维护计划的一部分。下面从介绍几个顶级参数开始，接着介绍一些选项，更大型的维护计划需求将会需要这些选项。

22.3.2 索引名

如果想要维护一个具体的索引，那么可以指定那个具体的索引的名称。使用 ALL 表明想要在所有与该命名表相关联的索引上进行维护。

22.3.3 表名或视图名

顾名思义——希望进行维护的具体对象(表或视图)的名称。注意它必须是一张具体的表(不能提供一个列表并说“请处理所有这些表!”)。

22.3.4 REBUILD

这是修复索引的“业内领先”方法。如果使用这个选项运行 ALTER INDEX，那么旧的索引将会被完全丢弃，并且从头开始重新构建索引。其结果是一个真正最优的索引，其中索引的叶子和非叶子级别中的所有页面都会被重新构建，就像它们被定义的那样(要么使用默认的参数，要么使用开关来修改填充因子等)。

提示：

在使用这个选项时要小心，一旦设置了 REBUILD 选项，那么正在使用的索引就会被删除，直到重建完成之后才可用。任何依赖这个索引的查询可能会变得异常缓慢(可能慢几个数量级)。对于这类事情，首先需要在一个脱机系统上进行测试，以了解这些查询大概需要花费多少时间，然后再设置计划以在空闲的时候进行重建(最好是有人能够监控这个过程以确保在高峰来临之前系统会重新联机)。

当它运行时可能会产生极大的副作用，因此以笔者这还算高明的观点来看，它完全属于数据库管理员的职责范围。

22.3.5 DISABLE

这个选项的作用就像其名字所说的那样，但是只在某些非常特殊的情况下才会用到这个选项。如果该命令所做的只是使索引脱机，直到做出决定下一步该做什么为止就好了，但是相反，它本质上会把索引标记为不可用。一旦一个索引被禁用之后，在使它再次活跃之前必须要重新构建它(不是重新组织，而是重新构建)。

读者亲自运行这个命令的可能性非常非常小(你更可能做的只是删除那个索引)。只有在 SQL Server 升级时，或者其他一些古怪的情况下才有可能使用这个选项。

提示：

这个选项还有一个需要小心的地方。如果禁用表的聚集索引，那么其效果是禁用了整张表。数据将会保留在那里，但是所有的索引都已经无法访问这些数据(因为它们都依赖于这个聚集索引)，直到重新构建聚集索引为止。

22.3.6 REORGANIZE

从开发者的角度来看没错!!! 有了 REORGANIZE 之后，开发人员的生活增添了更多的欢乐。与完整的重新构建相比，重新组织索引在优化程度上稍微有点不彻底，但是它是联机发生的(用户仍然可以使用这个索引)。

如果读者足够细心，那么很快就会提出这个问题“稍微有点不彻底到底意味着什么呢？”好吧，REORGANIZE 只工作在索引的叶子级别上——索引的非叶子级别将不会发生变化。这意味着不会得到完整的优化，但是对于索引最主要的部分来说，它们不是碎片真正产生开销的地方(尽管这是可能发生的，并且路途可能是多种多样的)。

因为它对用户产生的影响更小，因此通常希望把这个工具作为常规的维护计划的一部分。下面来看一下运行一个索引重新组织命令。

按照步骤运行它，下面将要在 AdventureWorks2008 数据库中的一张表上进行重新组织。Production.TransactionHistory 表是一个非常优秀的示例表，因为随着时间的推移，可能会有很多行被插入到该表中，然后当事务足够陈旧以至于可以删除时又会有很多行被清除。在本例中将会使用一个简单的命令来重新组织表上的所有索引：

```
USE AdventureWorks2008;  
  
ALTER INDEX ALL  
ON Production.TransactionHistory  
REORGANIZE;
```

ALTER INDEX 看到提供了 ALL 选项，而不是一个具体索引的名称，因此它将会查询 Production.TransactionHistory 表上所有可用的索引(忽略那些被禁用的索引，因为重新组织不会对它们做任何处理)。接着它会遍历所有的索引，并且在每一个索引上都进行重新组织——只重新组织每个索引的叶子级别(包括重新组织实际的数据，因为表上的聚集索引也会被重新组织)。

通常将不会从数据库中得到什么实质性的结果——只有一个简单的“命令执行成功”消息。

22.4 数据存档

哦——这是一个复杂的问题。有多少数据库工程师就有多少种归档数据的方式。如果正在构建一个 OLAP 数据库——例如,与分析服务一起使用——那么存档通常用于满足长期报表的需求。不管如何确保所需的数据长期一直可用,总会有一天,需要处理因数据过多而导致系统性能不佳的问题。

正如笔者之前说过的那样,由于每个数据库都有点不同,从而导致存档的方法真是太多了。这里的重点是在创建数据库的时候要考虑存档的需求。要意识到,当开始删除记录时就会遇到引用完整性约束和/或孤立记录的问题——在存档的时候设置一个逻辑路径来删除或移动记录。下面是在编写存档脚本时需要考虑的一些事情:

- 如果数据已经存在于 OLAP 数据库中了,那么可能就不需要考虑把它保存到其他任何地方了。跟你的老板和律师讨论这个问题。
- 使用数据的频率如何? 保存这些数据是否值得? 人类天生就喜欢收藏小玩意儿。简单地说,我们讨厌放弃东西——包括数据。如果只是担心需求的合法性,那么可以考虑只将永不或者很少用到的数据保存到磁带上(建议为存档数据创建多个备份),并减少联机的数据量——当用户看到性能得到提升时会感激你的。
- 不要留下孤立记录。当开始删除数据时,引用完整性约束应该会防止留下许多孤立记录,但是还是会遇到某些没有应用引用完整性的地方。这种情形可能会导致严重的系统错误。
- 要意识到存档程序可能需要运行很长时间。它运行的时间长度和影响的行数可能会与联机用户试图访问的数据之间存在并发问题——在用户不使用系统的时候运行它。
- 测试! 测试! 测试!

22.5 PowerShell

现在,SQL Server 支持一个名为 PowerShell 的命令环境。对于那些以前没有听说过 PowerShell 的人来说,值得好好花时间看一看这个工具,而这部分内容超过了本节的范围,因此笔者建议在 Web 上好好搜索一下。

PowerShell 是什么? 从最基本的角度来看,PowerShell 是一个经典的命令行环境——并且在本质上与 Windows 命令行窗口有很大的不同。但是,PowerShell 是可以通过与 .NET 集成来扩展的,并且能够由其他应用程序承载(大多数情况下是用在 SQL Server 2008 中)。为承载 PowerShell 而添加特殊功能的应用程序和操作系统包括:

- SQL Server 2008(否则为什么要讨论它呢?)以及更新的版本。
- Exchange 2007 以及更高的版本。
- Microsoft Office Sharepoint Services(MOSS) 2007 以及更高的版本。
- Vista、Windows XP、Windows Server 2003(通过下载功能插件)。
- Windows Server 2008 和之后的版本都自带这个工具或者把它作为一个选项(取决于版本)。

PowerShell 的扩展性是通过所谓的 cmdlet(发音为 commandlets)来实现的。它们是特殊的.NET 程序集,用于为 PowerShell 环境中的给定应用程序实现指定的功能。这里真正强大的地方是:通过组合 PowerShell 中可用的各种不同的 cmdlet 能够创建功能强大的脚本,以组合利用操作系统命令和具体的一个或多个应用程序的功能(例如,在启动另一个环境中承载的某个应用程序之前,等待并确认把脚本加载到数据库中)。

PowerShell 的 cmdlets 有一个标准的命令结构,它是动词和名词的组合,如 Get-Help 或者 Get-Children。它还包括一个可靠的帮助机制,并且这种帮助机制会定期被更新(通过 TechNet)。

22.5.1 尝试 PowerShell

为了了解 PowerShell 是如何工作的,下面将会对它进行一次快速测试。首先打开一个命令行提示符窗口(“开始”|“运行”,然后输入 cmd,然后按回车)。在命令行中输入 PowerShell。

```
C:\Users\Administrator.Kierkegaard>sqlps
```

这样就会离开标准的命令提示符窗口,并进入 PowerShell 的世界,但是这种进入另一个环境的指示是相对较少的。实际上,唯一明显一点的指示(除了 PowerShell 头)是命令行上的 PS 前缀,而其他部分看起来就像是普通的命令行提示窗口:

```
Microsoft SQL Server PowerShell
Version 10.0.1600.22
Microsoft Corp. All rights reserved.
```

```
PS SQLSERVER:\>
```

下面继续并发出第一个 PowerShell 命令。这里仅仅是请求帮助页面:

```
PS SQLSERVER:\> Get-Help
```

运行上面的命令将会输出大概 1 页左右的有用信息:

TOPIC

Get-Help

SHORT DESCRIPTION

Displays help about PowerShell cmdlets and concepts.

LONG DESCRIPTION

SYNTAX

```
get-help {<CmdletName> | <TopicName>}
help {<CmdletName> | <TopicName>}
<CmdletName> -?
```

"Get-help" and "-?" display help on one page.

"Help" displays help on multiple pages.

Examples:

```
get-help get-process : Displays help about the get-process cmdlet.  
get-help about-signing : Displays help about the signing concept.  
help where-object : Displays help about the where-object cmdlet.  
help about_foreach : Displays help about foreach loops in PowerShell.  
match-string -? : Displays help about the match-string cmdlet.
```

You can use wildcard characters in the help commands (not with -?).
If multiple help topics match, PowerShell displays a list of matching topics. If only one help topic matches, PowerShell displays the topic.

Examples:

```
get-help * : Displays all help topics.  
get-help get-* : Displays topics that begin with get-.  
help *object* : Displays topics with "object" in the name.  
get-help about* : Displays all conceptual topics.
```

For information about wildcards, type:

```
get-help about_wildcard
```

REMARKS

To learn about PowerShell, read the following help topics:

```
get-command : Displays a list of cmdlets.  
about_object : Explains the use of objects in PowerShell.  
get-member : Displays the properties of an object.
```

Conceptual help files are named "about_<topic>", such as:

```
about_regular_expression.
```

The help commands also display the aliases on the system.

For information about aliases, type:

```
get-help about_alias
```

```
PS SQLSERVER:\>
```

这些只是关于如何在 PowerShell 中获取帮助的基本信息。这个帮助窗口提供的信息非常少，如果说已经提供了一些信息，那就是表明它是特定于 SQL Server 的。然而，可以获取运行普通的 T-SQL 的 cmdlet 的帮助信息：

```
PS SQLSERVER:\> Get-Help Invoke-Sqlcmd
```

运行上面的命令将会返回 SQL Server 中具体名为 Invoke-Sqlcmd 的 cmdlet 的帮助信息：

NAME

```
Invoke-Sqlcmd
```

SYNOPSIS

```
Runs a script containing statements from the languages  
(Transact-SQL and XQuery) and commands supported by the SQL Server sqlcmd utility.
```

SYNTAX


```
Invoke-Sqlcmd [-ServerInstance <PSObject>] [-Database <String>] [-EncryptCo
nnection] [-Username <String>] [-Password <String>] [[-Query] <String>] [-Q
ueryTimeout <Int32>] [-ConnectionTimeout <Int32>] [-ErrorLevel <Int32>] [-S
everityLevel <Int32>] [-MaxCharLength <Int32>] [-MaxBinaryLength <Int32>] [
-AbortOnError] [-DedicatedAdministratorConnection] [-DisableVariables] [-Di
sableCommands] [-HostName <String>] [-NewPassword <String>] [-Variable <Str
ing[]>] [-InputFile <String>] [-OutputSqlErrors] [-SuppressProviderContextW
arning] [-IgnoreProviderContext] [<CommonParameters>]
```

DETAILED DESCRIPTION

Runs a script containing the languages and commands supported by the SQL Server `sqlcmd` utility. The languages supported are Transact-SQL and the XQuery syntax supported by the Database Engine. `Invoke-Sqlcmd` also accepts many of the commands supported by `sqlcmd`, such as `GO` and `QUIT`. `Invoke-Sqlcmd` accepts the `sqlcmd` scripting variables, such as `SQLCMDUSER`. `Invoke-Sqlcmd` does not set `sqlcmd` scripting variables by default.

`Invoke-Sqlcmd` does not support the `sqlcmd` commands primarily related to interactive script editing. The commands not supported include `!!`, `:connect`, `:error`, `:out`, `:ed`, `:list`, `:listvar`, `:reset`, `:perftrace`, and `:serverlist`.

The first result set the script returns is displayed as a formatted table. Result sets after the first are not displayed if their column list is different from the column list of the first result set. If result sets after the first set have the same column list, their rows are appended to the formatted table that contains the rows that were returned by the first result set.

`Invoke-Sqlcmd` does not return message output, such as the output of `PRINT` statements, unless you use the PowerShell `-Verbose` parameter.

RELATED LINKS

SQL Server Books Online: Transact-SQL Reference
 SQL Server Books Online: `sqlcmd` Utility
 SQL Server Books Online: XQuery Reference

REMARKS

For more information, type: "get-help Invoke-Sqlcmd -detailed".
 For technical information, type: "get-help Invoke-Sqlcmd -full".

PS SQLSERVER:\>

下面快速看一看如何使用它来查询一个相对简单的系统存储过程(`sp_help`):

PS SQLSERVER:\> Invoke-Sqlcmd -Query "EXEC sp_helpdb"

`sp_helpdb` 返回系统中所有数据库的列表。通常将会看到面向列的结果集,但是 PowerShell 对其输出做了调整使其更适合在一行中字符数受限的命令窗口中输出:

```
name           : AdventureWorks2008
db_size        : 245.81 MB
owner          : sa
dbid           : 7
created        : Dec 6 2008
status         : Status=ONLINE, Updateability=READ_WRITE, UserAccess=MULTI
               _USER, Recovery=SIMPLE, Version=655,
Collation=SQL_Latin1
               _General_CP1_CI_AS, SQLSortOrder=52, IsAnsiNullsEnabled,
```

```

        IsAnsiPaddingEnabled, IsAnsiWarningsEnabled, IsArithmetic
        AbortEnabled, IsAutoCreateStatistics,
IsAutoUpdateStatist
        ics, IsFullTextEnabled, IsNullConcat,
IsQuotedIdentifiers
        Enabled, IsPublished
compatibility_level : 100

```

```

name          : AdventureWorksDW2008
db_size       : 71.06 MB
owner         : sa
dbid          : 8
created       : Dec 6 2008
status        : Status=ONLINE, Updateability=READ_WRITE,
UserAccess=MULTI

```

```

        _USER, Recovery=SIMPLE, Version=655,
Collation=SQL_Latin1
        _General_CP1_CI_AS, SQLSortOrder=52, IsAnsiNullsEnabled,
        IsAnsiPaddingEnabled, IsAnsiWarningsEnabled,
IsArithmetic
        AbortEnabled, IsAutoCreateStatistics,
IsAutoUpdateStatist
        ics, IsFullTextEnabled, IsNullConcat,
IsQuotedIdentifiers
        Enabled
compatibility_level : 100

```

```

name          : AdventureWorksLT2008
db_size       : 7.13 MB
owner         : sa
dbid          : 9
created       : Dec 6 2008
status        : Status=ONLINE, Updateability=READ_WRITE,
UserAccess=MULTI

```

```

        _USER, Recovery=SIMPLE, Version=655,
Collation=SQL_Latin1
        _General_CP1_CI_AS, SQLSortOrder=52, IsAnsiNullsEnabled,
        IsAnsiPaddingEnabled, IsAnsiWarningsEnabled, IsArithmetic
        AbortEnabled, IsAutoCreateStatistics,
IsAutoUpdateStatist
        ics, IsFullTextEnabled, IsNullConcat,
IsQuotedIdentifiers
        Enabled
compatibility_level : 100

```

```

name          : tempdb
db_size       : 8.75 MB
owner         : sa
dbid          : 2
created       : Dec 31 2008
status        : Status=ONLINE, Updateability=READ_WRITE,
UserAccess=MULTI

```

```

        _USER, Recovery=SIMPLE, Version=655,
Collation=SQL_Latin1
        _General_CP1_CI_AS, SQLSortOrder=52,
IsAutoCreateStatisti
        cs, IsAutoUpdateStatistics
compatibility_level : 100

```



```
PS SQLSERVER:\>
```

由于篇幅的限制，这里只从结果集中删去了一部分数据库，但是读者可能看到，实际上可以在 PowerShell 中执行任何命令。随着时间的推移，很多命令将会有特定的 cmdlet 来支持它们——支持更强的类型和参数化。到现在为止，大多数已实现的 cmdlet 支持四种主要的对象模型：

- **数据库引擎**：这个模型允许定位到一个给定的服务器。
- **基于策略的管理**：这是一个在 SQL Server 2008 中新增的基于规则的管理工具(在下一节中将会对其做简要介绍)。
- **数据库集合**：它包含了操作一个给定数据库或一组给定数据库的主要工具。
- **服务器注册**：它用于识别服务器，并且在本地注册它们使得访问它们变得更容易。

通过使用这些对象模型，PowerShell 能够使用脚本完成几乎所有的管理任务。在 SQL Server 2008 的生命周期中要关注将会以下载形式加入到这个工具中的更加具体的支持和帮助。

22.5.2 在 PowerShell 中导航

PowerShell 还支持以类似于目录的方式进行导航，这与之前在 SQL Server 中所采用的方式是不同的。实际上，可以把 SQL Server 世界想象成一个大型的层次结构(非常类似于域/目录结构)。可以在一组注册服务器中导航并定位到具体的服务器上，然后在服务器中可以定位到角色和用户，或者可以定位到数据库和数据库中的对象。

下面通过发出一个 `dir` 命令来快速检查一下，其工作方式就像是在操作系统的命令窗口中发出该命令：

```
PS SQLSERVER:\> dir
```

读者可能会感到奇怪，它返回了上一节最后提到的四种对象模型列表：

Name	Root	Description
----	----	-----
SQL	SQLSERVER:\SQL	SQL Server Database Engine
SQLPolicy	SQLSERVER:\SQLPolicy	SQL Server Policy Management
SQLRegistration	SQLSERVER:\SQLRegistration	SQL Server Registrations
DataCollection	SQLSERVER:\DataCollection	SQL Server Data Collection

```
PS SQLSERVER:\>
```

实际上可以像在 Windows 的目录结构那样进行导航——例如：

```
PS SQLSERVER:\> cd SQL
```

读者应该很快就会注意到进入了目录结构的下一层：

```
PS SQLSERVER:\SQL>
```

下面稍微往前进一点，即定位到树的更深处。例如，这里将需要定位到具体的服务器上(笔者的是 KIERKEGAARD，读者应该用自己的 SQL Server 系统的名称替换它)实例(笔者的是 DEFAUCT)以及 DATABASES 节点(也可以使用其他的服务器级别的对象，如 LOGINS)：

```
PS SQLSERVER:\SQL> cd KIERKEGAARD\DEFAULT\DATABASES
```

这里直接定位到了层次结构中的数据库节点上，就像是在目录结构中导航一样：

```
PS SQLSERVER:\SQL\KIERKEGAARD\DEFAULT\DATABASES>
```

还有更好的呢。可以发出一个目录列表命令(以 `dir` 命令的形式)来获取数据库列表，就像本章前面使用 `sp_help` 创建的那个命令一样(虽然结果没有那么详细)：

```
PS SQLSERVER:\SQL\KIERKEGAARD\DEFAULT\DATABASES> dir
```

上面的命令将会输出：

WARNING: column "Owner" does not fit into the display and was removed.

Name	Status	Recovery Model	CompatLvl	Collation
AdventureWorks2008	Normal	Simple	100	SQL_Latin1_Genera l_CP1_CI_AS
AdventureWorksDW2008	Normal	Simple	100	SQL_Latin1_Genera l_CP1_CI_AS
AdventureWorksLT2008	Normal	Simple	100	SQL_Latin1_Genera l_CP1_CI_AS
AWSubscriber	Normal	Full	100	SQL_Latin1_Genera l_CP1_CI_AS
OurInsteadOfTest	Normal	Full	100	SQL_Latin1_Genera l_CP1_CI_AS
ReportServer	Normal	Full	100	Latin1_General_CI AS_KS_WS
ReportServerTempDB	Normal	Simple	100	Latin1_General_CI AS_KS_WS
Test	Normal	Full	100	SQL_Latin1_Genera l_CP1_CI_AS

```
PS SQLSERVER:\SQL\KIERKEGAARD\DEFAULT\DATABASES>
```

当然，这是一个相当简单的例子，但是还可以更进一步扩展这个例子。例如，PowerShell 允许列举诸如刚才创建的目录列表等之类的列表，然后可以根据列表的内容为不同的行为创建脚本。

22.5.3 关于 PowerShell 的最后一点说明

在笔者正在写本书的时候，从 SQL Server 的角度来看，PowerShell 的开发才刚刚开始。可用的有关 cmdlet 的文档仍然相当稀少，但是新项目会定期发布，同时 PowerShell 模型的性质决定了能够持续扩展 PowerShell 的功能，甚至在 Kilimanjaro(SQL Server 的下一个发行版的代码名称)发布之前就可以对其进行扩展。

笔者强烈建议在 Internet(或者经常在 Google 上搜索 SQL Server PowerShell)上关注 SQL Server 社区以了解新增了哪些功能以及这个新脚本工具的发展方向。笔者可以说它将很快成为我首选的创建安装和升级脚本环境。

22.6 基于策略的管理

基于策略的管理——在 SQL Server 2008 的 beta 阶段作为分布式管理框架而被大家所熟知——是一个基于规则的管理基础设施，其主要目的在于管理大型企业的 SQL Server 农场。它的想法是相当简单的，现在存在很多 SQL Server 由很多不同的人员来管理(通常是完全不同的 IT 部门或 DBA，他们甚至根本就不了解其他服务器和 DBA)，那为什么不让所有的 SQL Server 根据一组“策略”来管理自己呢。策略强加的约束可以是多种多样的，从简单的对象命名指南到阻止修改具体的服务器设置等等。管理引擎能够检测出违反策略的行为(简单地报告这个情况)，然后它可以阻止这个变更或者反转这个变更。

基于策略的管理对开发者社区的影响还有待观察。笔者猜测将会有一些优秀的脚本编写应用程序对它提供支持，但是，在很多公司中，基于策略的管理到底将如何开展以及如何实现策略的强制执行还处于研究之中。现在，笔者所能说的是，基于策略的管理功能是通过 SMO(在 `Microsoft.SqlServer.Management.Dmf` 库中)和 PowerShell 来实现的。当然，还有很多期待的东西，如在 Management Studio 之外的对象模型的文档，但是在 Books Online 中已经提供了其中几个函数的文档，因此笔者猜测在 SQL Server 2008 的生命周期内对联机丛书的更新将会加入基于策略的管理的对象模型这部分内容。

22.7 小结

好，现在读者已经有很多东西需要思考了。作为开发人员去考虑很多管理任务和建立类似于《银河系漫游指南》三部曲中被称为“SEP”场的东西是非常容易的事情。那是能够使像管理之类的事情变得看上去不存在的东西，因为它是“其他人的问题。”不要这样做！

提示：

笔者几年前所熟悉的一个项目是一个说明要对可能发生的事情承担责任的极好的例子。在这个例子中有一个公司为美国西北部的一个非盈利组织开发了一套优秀的系统。在运行了 8 个月之后，开发这个软件的公司接到了一个紧急求助电话(这是很常见的)。经过一番讨论之后，大家一致认为数据库莫名其妙地崩溃了，然后他们建议客户从备份中还原数据库。响应是“备份”？开发公司忘记了一些非常重要的事情——他们知道他们的客户不是很有经验，并且将不会配备管理人员，但是如果开发公司不告诉客户要进行备份并帮助他们进行相应的设置，那么谁来帮客户做这些事情呢？我很高兴地说，开发公司从这个经验中学到了很多——你也会学到很多。

在进行设计，特别是在规划部署的时候考虑一下管理问题。如果事先进行规划以简化系统的管理，那么该系统将会更加成功——那通常会转化为对开发者(也就是你！)的奖励。

第23章

SMO: SQL 管理对象

这是一条漫长的路，现在已经越来越接近 SQL Server 之旅的终点了。当然，通常的做法都是把讲述如何以编程方式管理 SQL Server 的章节放在最后，因为在理解 SMO 对象模型和讨论为什么要使用 SMO 之前，需要对所管理的对象以及有哪些管理需求有一个扎实的理解。

那么，SMO 到底是什么呢？好吧，正如本章的标题所蕴含的那样，SMO 是一个管理 SQL Server 的对象模型。像 ADO 和 LINQ 这样的连接模型是针对如何访问数据的，而 SMO 则是针对如何访问系统的结构和状态的。

本章将会介绍：

- SQL Server 管理对象模型的曲折历史。
- SQL SMO 对象模型的基础知识。
- 一个简单的 SMO 示例项目。

提示：

与本书中介绍的很多 SQL Server 主题一样，SQL SMO 也是需要用一本书来讲述的，因此读者不要期望在读完本章之后就成为专家。也就是说，读者在读完本章之后至少能够了解到使用 SMO 能够做什么，以及完成这些任务需要做哪些工作等基础知识。在这个基础上，读者就可以根据需要查询更多的信息来源了。

23.1 SQL Server 管理对象模型的发展历程

对我来说——甚至对那些真正喜爱这个产品的人来说——SMO 不是 SQL Server 的一个亮点。这并不是说 SMO 是一个糟糕的东西，而是说 SQL Server 管理对象模型的历史确实相当坎坷。开发团队在最初启动和后续的开发中都经历过非常困难的时刻。

在笔者编写本书的时候，我使用 SQL Server 已经将近 15 年了。在那段时间里，管理 SQL Server 的方法已经改变了好几次。“一个新的发行版？一个新的管理方法！”简直成了 SQL Server 的座右铭了。

好消息是，至少到现在为止，SMO 看起来比较稳定。作为管理 SQL Server 的主要对象模型，它现在是第二个版本(我知道需要经过两个版本才完成确认让人感到难过)。而其他模型仍然保留在那里，因此下面介绍最近几个发行版中比较突出的地方。它们是一些不同的模型和技术，读者

在接触遗留代码时可能会碰到它们。

23.1.1 SQL 分布式管理对象

分布式管理对象，或者 DMO，是一个相对较老的管理模型。SQL Server 2000 以及之前的版本中 Enterprise Manager 的大多数底层功能最终都是转化为 DMO 调用来实现的。DMO 支持 COM，并且能够执行与管理相关的所有基本任务，如：

- 开始备份
- 从备份中还原
- 创建数据库
- 创建作业和其他代理相关的任务
- 通过逆向工程将表转化为 SQL 代码

这个列表还在增长。

那么 DMO 到底有什么问题呢？好吧，这个对象模型一直被认为是非常笨拙的。实际上，DMO 的各个部分无法很好地一起协同工作，同时脚本引擎也具有非常多的 Bug。简而言之，笔者认识的大多数开发人员都只会在接收电击治疗法以降低对痛苦的敏感度之后才会使用 DMO(好吧，它没有那么糟糕，但是也差不多了)。

23.1.2 SQL 名称空间

SQL 名称空间(SQL NS)实际上主要用于提供 UI 级别的功能。SQL NS 封装了所有在旧的 Enterprise Manager 中能够找到的功能——包括所有 UI 元素。UI 对象被实例化之后会自动在底层使用 SQL DMO，因此不需要再进行编程了。简而言之，如果需要构建一个已经拥有 UI 来执行管理任务的工具，那么应该选择 SQL NS。但问题是什么呢？好吧，换一种方式说——EM 如何？他们准备替换掉它。那 DMO 呢？他们也准备替换掉它。正如读者猜测的那样，显然即使是 Microsoft 也不是那么满意。

提示：

现在，为了避免我听起来像是一个 Microsoft 的抨击者，或者让人误以为我觉得 EM 是一个糟糕的产品，我将会用这种方式说：EM 是一个相当好的“让人一见钟情的”产品。当它第一次出现时，没有哪个 RDBMS 所拥有的远程工具能够像 EM 那样强大和有用——为什么大家认为 SQL Server 要比其他数据库，比如 Oracle(尽管 Oracle 在管理领域的确取得了很大的进步)好用得多，这是一个重要的原因。这种易用性，再加上它曾经非常便宜的价格对 Microsoft 在 SQL Server 上取得成功起了很大的作用。

但是，EM 确实存在一些缺点。随着 Windows 时代教会了我们一个 Windows 应用程序应该具有什么样的外观和功能，EM 的缺点也就越来越明显了。

23.1.3 Windows Management Instrumentation

与到现在为止介绍的其他管理对象相比，Windows Management Instrumentation(WMI)有很大的不同，因为它不是 SQL Server 特有的，而是一个管理脚本模型的实现，并且它已经被用于管理

跨越 Windows 以及其他一些领域的服务器。

WMI 是行业开放标准 WBEM(基于 Web 的企业管理, Web-Based Enterprise Management)的一个实现。WBEM 并不仅仅局限于 Microsoft 的产品, 其理念是让服务器管理员只学习一种核心脚本模型就能够管理所有的服务器。Exchange、SQL Server、Windows O/S 特性等等——它们将都通过 WMI 来管理(实际上, 其中大多数都是通过 WMI 来管理的)。

进入 SQL Server 2000 之后, 其方向变得非常清楚了: WMI 是未来。许多 SQL Server 的坚守者(像我)被一次又一次地告知——DMO 将会被舍弃(结果确实如此), 我们应该在 WMI 上进行新的管理(结果并非如此)。

现实是从来没有在 SQL Server 上完整地实现过 WMI, 但是短时间内它也不会被舍弃。正如笔者所说的那样, WMI 是一个行业标准, 很多其他的 Windows 服务器将会使用 WMI 来进行配置管理。使用 WMI 来完成一些基本的配置任务是很有意义的, 因此, 从这个方面来看, 它可能还将有一席之地(笔者对此没有怨言)。

提示:

值得指出的是, WMI 现在实现为 SMO 之上的一个层——想想吧。

23.1.4 SMO

笔者不清楚 Microsoft 是什么时候决定转向 SMO 的, 但是我能说的是他们知道自问的问题: DMO 显然已经到了生命的尽头, 并且已经计划在 SQL Server 2005 中对 Enterprise Manager 进行完全重写。同时, WMI 显然无法处理所有需要完成的任务(WMI 是面向配置的, 但是 SQL Server 所需的管理任务无论如何也要比 WMI 能够提供得多)。

因此, 随着 SQL Server 2000 进入市场, .NET 明显已经开始兴起。而 Visual Studio 2005 也处于紧张的设计当中, C# 已经作为未来的程序设计语言在兜售了。显然, Microsoft 决定使用 Visual Studio 插件来作为管理中心(实际上, 读者仍然可以看到它明显也用于报表和集成, 并且在一定程度上也用于分析服务)。

最终, SMO 将会包含一组非常有用的 .NET 程序集, Management Studio 将回到原来的角色(与 Visual Studio 捆绑得太紧显然不能很好地完成工作, 我喜欢这个把它们分开的决策), 但是它是基于 Visual Studio 为基础的, 并且利用了 Visual Studio 的某些元素, 如 IntelliSense 现在已经成为 SQL Server 2008 产品的一部分了。要求使用设计器的服务将会使用 Business Intelligence Development Studio, 而它仍然是 Visual Studio 中的一组项目、空间和模板(实际上, 在启动的时候会显示 Visual Studio)。

我的猜测? 好吧, 依赖于 SQL Server 推出新版本之前需要多长时间, 我认为, 至少在最近的一个两个发行版中使用 SMO 作为对象模型都是安全的。目前还没有出现 SMO 要被替代的苗头, 它看起来非常可行(在可预见的将来, 没有理由替换它)。简而言之, 至少在 5 到 10 年内是可以依靠它的, 而这在软件业中大概也是最长的时间了。

注意:

尽管可以假设在接下去的几个发行版中都会支持 SMO, 但是需要指出的是, 在各个发行版之间 SMO 可能并不会 100% 代码兼容。例如, 作为核心的 Microsoft.SqlServer.Smo.dll 一部分的某些类可能会被移到名为 Microsoft.SqlServer.SmoExtended.dll 的新文件中。如果不在项目中添加这个

新的引用文件，那么使用 SQL Server 2008 库进行编译将会失败。

23.2 SMO 对象模型

服务器管理对象(Server Management Objects, SMO)取代了 DMO。这就是说, SMO 在 DMO 所能完成的事情上都要做得更好。除了基本的配置和语句执行之外, SMO 具有如下一些真正高级的功能:

- 事件处理: SMO 支持捕获发生在服务器上的事件并注入代码来处理该事件的理念。
- 能够把服务器中的对象类型当成集合来处理(使得列举它们变得非常容易, 并且提供了对该类型的对象的一致和完整的处理)。
- 以一种相对一致的方式来处理各种作为 SQL Server 一部分的服务器对象。

与所有对象模型一样, SMO 在对象之间建立起了一个层次结构。由于 SQL Server 是一个非常复杂的产品, 因此需要考虑很多很多对象。图 23-1 给出了一个层次结构的例子, 笔者认为图中的对象是 SQL Server 中的“核心”对象。

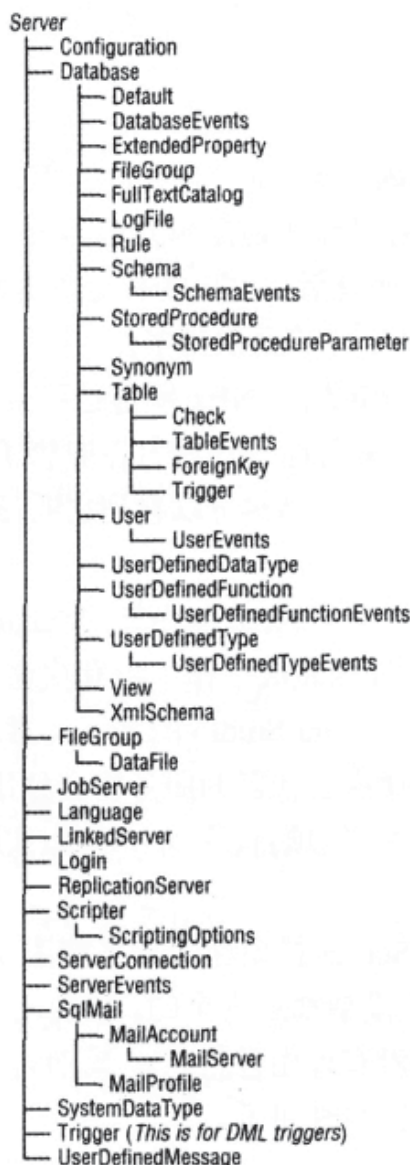


图 23-1

注意这根本谈不上是一张完整的列表! 如果想要一幅包含所有内容的图, 那么可以参考联机丛书(它里面有一张不错的图, 虽然也不是很好——至少它是完整的)。这里我只是试图给读者提

供一些可读性更好的内容，其中包含了所有的核心对象，还有其他一些内容。

23.3 实例演练

这可能是整本书中最杂乱的一节了，因为笔者所要“谈论”的内容包含了大量关于 Visual Studio 的内容，而这些内容没有内建到基本的 SQL Server Business Intelligence Studio 中。

提示：

读者只有安装某个版本的 Visual Studio，才能自己构建这些例子。然而，如果没有的话也不用害怕——这里将会把所有代码都展示出来，这样你至少可以自己看看它们。

还有，虽然下面的例子是使用 C# 完成的，但是基本的对象引用和方法调用是一样的——把它们转换成 VB 或 C++ 对于熟悉这些语言的人来说是非常简单的。

本节将要做的是构建一个小型应用程序来完成几个不同的读者可能感兴趣的“基本”功能。在所有这些各种各样的动作中至少发生一次的事情包括：

- 创建一个到具体服务器的引用，包括到服务器的连接，该连接将使用受信连接来实现。
- 新建一个数据库。
- 在数据库中创建表。
- 为那些表创建主键约束。
- 在一张表中创建外键来引用另一张表。
- 删除数据库。
- 备份数据库。
- 为数据库对象生成脚本。

这里每一个动作都将使用其超级简化的版本。记住，这里引用的每个对象都有很多属性和方法可供设置。例如，在生成脚本的例子中，可以设置脚本选项，来决定哪些属性命令会出现在脚本中以及哪些属性命令不会出现在脚本中。

23.3.1 开始

首先在 Visual Studio 中创建一个新的 Windows 应用程序项目。本例中把它命名为 SQLSMOExample。为了使用 SMO 程序集，需要在项目中至少设置对下面 5 个程序集的引用：

- Microsoft.SqlServer.ConnectionInfo
- Microsoft.SqlServer.Management.Sdk.Sfc
- Microsoft.SqlServer.Smo
- Microsoft.SqlServer.SmoExtended
- Microsoft.SqlServer.SqlEnum

设置一个引用非常简单，只要在“解决方案资源管理器”（或者在“项目”菜单中）中右击“引用”，然后选择“添加引用”。选择前面列表中的 5 个程序集，然后单击“确定”。

在这个例子中，出于简单的目的，所有的代码都放在了一个名为 frmMain 的窗体中。在大多数情况下，应该把方法放到独立的组件文件中，然后根据需要在窗体中调用它们。

1. 声明

为了在代码中以更简洁的方式使用那些对象，需要为几个管理库添加声明：

```
using Microsoft.SqlServer.Management.Smo;
using Microsoft.SqlServer.Management.Common;
using Microsoft.SqlServer.Management.Smo.SqlEnum;
```

这将允许在不使用完全限定名的情况下引用这些库中的对象。

2. 基本连接和服务引用

在本章中创建的所有方法都会重复使用一块代码，这段代码的目的是建立一个连接和一个服务器引用——所有任务都会用到它们。

在实际情况下，可能会想要建立一个或多个与应用程序的全局连接，而不是为某个具体方法建立连接，但是，同样，这里试图让代码块保持一定程度的独立性，这样读者就可以单独分析它们了。

连接和服务引用的代码如下所示：

```
// Create the server and connect to it.
ServerConnection cn = new ServerConnection();
cn.LoginSecure = true;

Server svr = new Server(cn);
svr.ConnectionContext.Connect();
```

23.3.2 创建数据库

创建数据库是相当简单的。在下面的实现中创建了一个 `Database` 对象，并立即使用到 `Server` 对象 `svr` 的引用来对其进行初始化。但是，注意这里只是创建了一个数据库定义对象。在调用数据库对象的 `Create()` 方法之前不会在服务器上真正创建数据库。因此，简而言之，这里定义了一个对象，接着修改对象中的各个属性，然后调用 `Create()` 方法来真正在被 `Server` 对象引用的服务器上创建数据库。

拖一个按钮到主窗体中——这里把它命名为 `btnCreateDB`——现在可以添加一些代码了。一个用于创建数据库的简单方法可能包括：

```
private void btnCreateDB_Click(object sender, EventArgs e)
{
    // Create the server and connect to it.
    ServerConnection cn = new ServerConnection();
    cn.LoginSecure = true;

    Server svr = new Server(cn);
    svr.ConnectionContext.Connect();
    Database db = new Database();

    db.Parent = svr;
    db.Name = "SMODatabase";
    db.Create();
}
```

```
txtResult.Text = "Database Created";

cn.Disconnect();
}
```

这里建立了一个通用数据库对象，然后把它与一个具体的服务器关联起来，并指定了数据库的逻辑名称，然后创建它。

其结果同连接到数据库并发出这样的命令并没有什么不同：

```
CREATE DATABASE SMODatabase
```

最终，一个空的数据库将会被创建，其设置全部采用默认值。但是，也可以设置一些属性，如物理文件位置(包括使用多个文件组来创建它)、默认的排序规则、增长和大小属性——基本上所能想到的数据库的任何属性。然而，更重要的是，这个创建过程是在原生的.NET 环境中完成的，而在客户端语言中可以很容易地处理任何错误、成功消息或者其他通知。

23.3.3 创建表

在本例中将会在空的 SMODatabase 中添加一对表。一张是 ParentTable，另一张是 ChildTable，其中 ChildTable 将会有有一个外键引用 ParentTable，两张表都有主键。

首先需要设置一个引用以指定在哪个数据库中创建表：

```
private void btnCreateTables_Click(object sender, EventArgs e)
{

    // Create the server and connect to it.
    ServerConnection cn = new ServerConnection();
    cn.LoginSecure = true;

    Server svr = new Server(cn);
    svr.ConnectionContext.Connect();

    // Get a reference to our test SMO Database
    Database db = svr.Databases["SMODatabase"];
```

注意，这次没有用“new”来创建一个新的 Database 对象，而是把它与一个来自所引用的 Server 对象中的既有数据库对象关联起来了。

这样就可以创建新的表对象了。与创建 Database 对象一样，这里所做的是在应用程序中创建一个对象定义。在完整地定义 Table 对象并调用 Create()方法之前将不会在数据库中创建任何表。

```
// Create Table object, and begin defining said table
Table ParentTable = new Table(db, "ParentTable");
```

现在已经可以向表的定义中添加一些内容了。不像数据库，数据库拥有足够多的默认值以至于只要指定一个名称就能够创建它了(剩余的部分将会从 model 数据库中复制过来)，而表则需要指定更多的东西——特别是它至少需要一个列。

下面添加一个列，它最后会成为主键：

```
// Build up the table definition
```



```
Column ParentKey = new Column(ParentTable, "ParentKey");
ParentKey.DataType = DataType.Int;
ParentKey.Nullable = false;
ParentKey.Identity = true;
```

现在已经创建了一个新的列对象。它从 `ParentTable` 中被模板化, 并被命名为 `ParentKey`。这里把它的数据类型指定为 `int`, 并把它设置为不能取 `NULL` 值, 同时把它定义为 `IDENTITY` 列。

注意:

尽管从 `ParentTable` 中模板化了改列, 但是它还没有与该表直接关联起来! 模板引用只是帮助为该列建立初始属性值(如排序规则)。

现在添加另一个名为 `ParentDescription` 的列:

```
Column ParentDescription = new Column(ParentTable, "ParentDescription");
ParentDescription.DataType = DataType.NVarCharMax;
ParentDescription.Nullable = false;
```

同样, 这个列被创建了, 但是还没有直接与 `Table` 对象关联起来——下面解决这个问题:

```
// Now actually add them to the table definition
ParentTable.Columns.Add(ParentKey);
ParentTable.Columns.Add(ParentDescription);
```

直到把它们添加到 `Table` 对象的 `Columns` 集合中之后, 它们才会与该表直接关联起来。现在已经定义了一个表对象, 同时还有两个列与之相关联。现在所需的是一个主键:

```
// Add a Primary Key
Index PKParentKey = new Index(ParentTable, "PKParentKey");
PKParentKey.IndexKeyType = IndexKeyType.DriPrimaryKey;

PKParentKey.IndexedColumns.Add(new IndexedColumn(PKParentKey,
"ParentKey"));

ParentTable.Indexes.Add(PKParentKey);
```

注意这里把主键定义为一个索引, 而不是那些明确称为约束的东西。相反, 这里定义了一个索引, 然后告诉索引它是一个主键(通过 `IndexKeyType`)。当索引被创建的时候约束定义也会被添加。

注意:

`Primary` 和 `Unique` 约束不是明确地作为约束来添加的。相反, 它们作为索引来添加的, 同时 `IndexKeyType` 蕴含着它们将会作为一个约束(而不是一个原生索引)被添加。

与列一样, 主键将不会直接与该表关联起来, 除非显式地把它添加到表的 `Indexes` 集合中。所有这些都做完之后就可以创建表了:

```
ParentTable.Create();
```

现在表在物理上已经被创建进了数据库。

好, 现在父表已经创建好了, 可以开始添加子表了。代码中一直到创建主键为止的代码看起来同创建 `ParentTable` 对象的代码是一样的:

```

// Create Table object for child, and begin defining said table
Table ChildTable = new Table(db, "ChildTable");

// Build up the Child table definition
Column ChildParentKey = new Column(ChildTable, "ParentKey");
ChildParentKey.DataType = DataType.Int;
ChildParentKey.Nullable = false;

Column ChildKey = new Column(ChildTable, "ChildKey");
ChildKey.DataType = DataType.Int;
ChildKey.Nullable = false;

Column ChildDescription = new Column(ChildTable, "ChildDescription");
ChildDescription.DataType = DataType.NVarCharMax;
ChildDescription.Nullable = false;

// Now actually add them to the table definition
ChildTable.Columns.Add(ChildParentKey);
ChildTable.Columns.Add(ChildKey);
ChildTable.Columns.Add(ChildDescription);

// Add a Primary Key that is a composite key
Index PKChildKey = new Index(ChildTable, "PKChildKey");
PKChildKey.IndexKeyType = IndexKeyType.DriPrimaryKey;

PKChildKey.IndexedColumns.Add(new IndexedColumn(PKChildKey,
"ParentKey"));
PKChildKey.IndexedColumns.Add(new IndexedColumn(PKChildKey,
"ChildKey"));

ChildTable.Indexes.Add(PKChildKey);

```

但是, 对于 ChildTable, 这里将会以外键的形式向其添加一点花样。为了完成这个任务, 需要创建一个 ForeignKey 对象:

```

// Add a Foreign Key
ForeignKey FKParent = new ForeignKey(ChildTable, "FKParent");

```

接着创建 ForeignKeyColumn 对象, 并把它们添加到 ForeignKey 对象中:

```

// The first "Parent Key" in the definition below is the name in the
// current table
// The second is the name (of just the column) in the referenced table.
ForeignKeyColumn FKParentParentKey = new ForeignKeyColumn(FKParent,
// "ParentKey", "ParentKey");

FKParent.Columns.Add(FKParentParentKey);

```

接着设置一个指向具体表的引用:

```

FKParent.ReferencedTable = "ParentTable";

// I could have also set a specific schema, but since the table was created
// using just a
// default schema, I'm leaving the table reference to it default also. They
// would be

```



```
// created using whatever the user's default schema is

/*
** Note that there are several other properties we could define here
** such as CASCADE actions. We're going to keep it simple for now.
*/
```

接着把外键添加到表中并创建该表：

```
ChildTable.ForeignKeys.Add(FKParent);

ChildTable.Create();

cn.Disconnect();

txtResult.Text = "Tables Created";

}
```

与仅仅连接并发出一个 CREATE TABLE 语句相比，这看起来可能有点麻烦，但是这种方法有几个优点：

- 如果需要动态构建一张表，那么可以把表结构的各个部分封装起来，这比操作字符串要简单得多。
- 与构建字符串相比，修改各种对象的属性受具体执行顺序的影响少得多。
- 所有属性保持离散，因此很容易找到它们并编辑它们，而不需要太多的字符串操作。
- 这就是 SMO 的工作方式——如果需要执行的其他动作已经存在于 SMO 中了，那么与把基于字符串的命令和 SMO 命令混合起来使用相比，在 SMO 中以一种一致的方式做事情可能会减少混淆。

23.4 删除数据库

与大多数删除操作一样，这是非常简单的。本节从已经熟知的服务器和连接信息开始，然后设置一个指向感兴趣的数据库的引用：

```
private void btnDropDB_Click(object sender, EventArgs e)
{
    // Create the server and connect to it.
    ServerConnection cn = new ServerConnection();
    cn.LoginSecure = true;

    Server svr = new Server(cn);
    svr.ConnectionContext.Connect();

    Database db = svr.Databases["SMODatabase"];
```

接着只需要调用 Drop() 方法就完成任务了：

```
db.Drop();
```

```
txtResult.Text = "Database Dropped";

cn.Disconnect();

}
```

注意:

注意这里并没有添加任何捕捉错误的代码(实际上,它们与所使用的语言中的其他错误捕捉问题并没有什么不同)。如果在这个应用程序或其他应用程序(如 Management Studio)中仍然有打开的到数据库的连接,那么删除数据库可能会出现一些问题。笔者建议读者对此进行测试,并决定在错误处理函数中到底该采取什么动作(记住,大多数的.NET 语言都有可靠的错误处理机制),如识别出所有在待删除的数据库上拥有锁的连接并断开这些连接。

23.5 备份数据库

在这个例子中将会切换到 AdventureWorks 数据库,这仅仅是为了备份操作更具实战性。

读者可能已经从迄今为止见过的这么多不同的对象中猜测出来了,在进行备份时将会使用 Backup 对象。它是 Server 对象的一个子对象,但是拥有自己的一组属性和方法。

要创建一个备份需要使用同样的服务器连接代码,这段代码已经见过好多次了:

```
private void btnBackupDB_Click(object sender, EventArgs e)
{

    // Create the server and connect to it.
    ServerConnection cn = new ServerConnection();
    cn.LoginSecure = true;

    Server svr = new Server(cn);
    svr.ConnectionContext.Connect();
```

现在可以创建新的 Backup 对象了。注意,与 Database 对象不同,Database 对象在一开始就需要和某个服务器关联起来,而在真正需要执行备份之前,Backup 对象不需要引用某个具体的服务器。

```
// Create and define backup object
Backup bkp = new Backup();
bkp.Action = BackupActionType.Database;
bkp.Database = "AdventureWorks2008";
bkp.Devices.AddDevice(@"c:\SMOSMOSample.bak", DeviceType.File);
```

现在已经创建了一个 Backup 对象,并且已经告诉它要执行什么类型的备份(一个数据库备份,与其他备份不同,如日志备份)。这里还指定了要进行备份的数据库并且定义了要使用的设备。

提示:

注意,这里直接定义了一个文件设备和路径,也可以连接到服务器并查询服务器上已经定义的设备,然后为备份从中选择一个设备。同样,设备也可以是不同的类型——如磁带。

现在已经可以执行这个备份了。执行这个备份有两种不同的方法:

- **SqlBackup**: 这是同步备份——在备份结束或出现错误之前, 代码将不会再次获得控制权。
- **SqlBackupAsync**: 这告诉服务器开始备份, 然后一旦服务器接受这个备份请求并认为它是合法的之后, 就会把控制权返回给应用程序(接着备份会在后台运行)。需要指出的是, 能够在备份到达结束点(可以定义结束点的粒度)之后接收通知。

本例选择了异步备份方法:

```
// Actually start the backup. Note that I've said to do this Asynchronously
// I could easily have make it synchronous by choosing SqlBackup instead
// Also note that I'm telling it to initialize (overwrite the old if it's
// there).
// Without the initialize, it would append onto the existing file if found.
bkp.Initialize = true;
bkp.SqlBackupAsync(svr);
cn.Disconnect();

}
```

读者在运行这段代码之后可以看一下 C:盘根目录下的 SQLSMOSample.bak 文件, 应该有这么一个文件! 另外尝试多次运行这个备份, 它应该每次都会覆盖文件。如果删除 `bkp.Initialize` 命令, 那么每个新备份应该会附在现有文件的后面。

23.6 生成脚本

对真正的开发人员来说, SMO 提供的功能当中最吸引人的功能也许是能够为已经存在于数据库中的对象生成脚本。实际上, SMO 能够生成的脚本包括备份和逆向工程表, 甚至包括记录被发送到服务器的语句。

在本例中将会为 AdventureWorks 数据库中的 HumanResources.Employee 表进行逆向工程的脚本。读者将会看到可以很容易地为一张相对复杂的表定义生成脚本以用作它用。

这里从同样的服务器、连接和数据库应用代码开始, 这段代码在本章中已经被用过好多次了:

```
private void btnScript_Click(object sender, EventArgs e)
{
    // Create the server and connect to it.
    ServerConnection cn = new ServerConnection();
    cn.LoginSecure = true;

    Server svr = new Server(cn);
    svr.ConnectionContext.Connect();

    // Now define the database we want to reference the table from.
    Database db = svr.Databases["AdventureWorks2008"];
}
```

接着设置一个指向待生成脚本的表的引用——也可以很容易地为不同类型的 SQL Server 对象生成脚本, 如存储过程、视图甚至是数据库。实际上, 它甚至可以是一个服务器级别的对象, 如设备或登录帐户。

```
// Get a reference to the table. Notice that schema is actually the *2nd*
// parameter
```

```
// not the first.
Table Employee = db.Tables["Employee", "HumanResources"];
```

现在可以调用 `Script()` 方法了。这里真正困难的地方是要意识到它返回的不仅仅是一个字符串，而是一个字符串集合。为了接收返回结果，需要创建一个恰当的类型为 `StringCollection` 的变量，而所有的 `using` 声明都没有予以定义，因此需要在变量声明中使用完全限定名。

```
// Call the Script method. The issue with this is that it returns a string
// *collection* rather than a string. We'll enumerate it into a string
// shortly.
System.Collections.Specialized.StringCollection script = Employee.Script();
```

好，现在已经接收到了脚本，但是现在想看一看里面的内容。这里定义了一个存储变量，然后把所有单独的字符串复制到一个在 `MessageBox` 中使用的字符串中：

```
string MyScript = "";

foreach (string s in script)
{
    MyScript = MyScript + s + "\r\n";
}

// Now show what we got out of it - very cool stuff.
MessageBox.Show(MyScript);

cn.Disconnect();
}
```

执行上面的代码后将会得到一个非常有用的脚本，如图 23-2 所示。

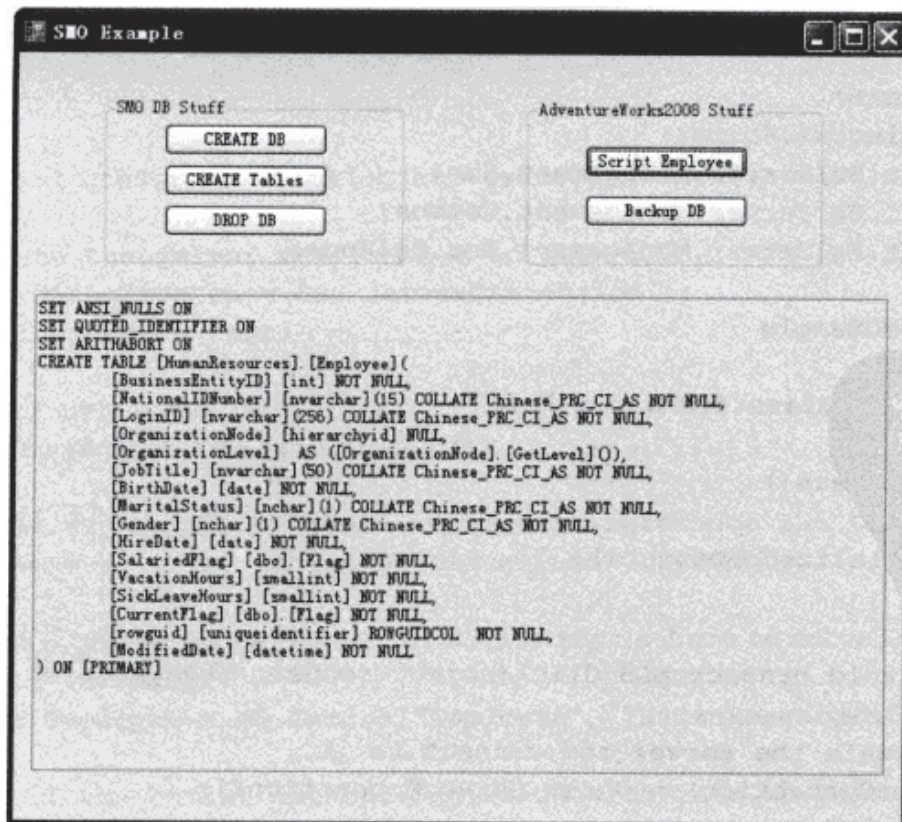


图 23-2

23.7 完整的代码

好，现在已经分段讲述了代码，但是笔者想要提供一个参考小节来展示把各段代码组装起来后是什么样子的。读者怎么设计窗体是你自己的事情，但是本例中的窗体看起来像图 23-3 中显示的那样。根据代码中各个按钮的名称应该就能够知道每个按钮的作用是什么，并把它与界面上的按钮对应起来，最下面是一个文本框，它在代码中的名称是 txtReturn。

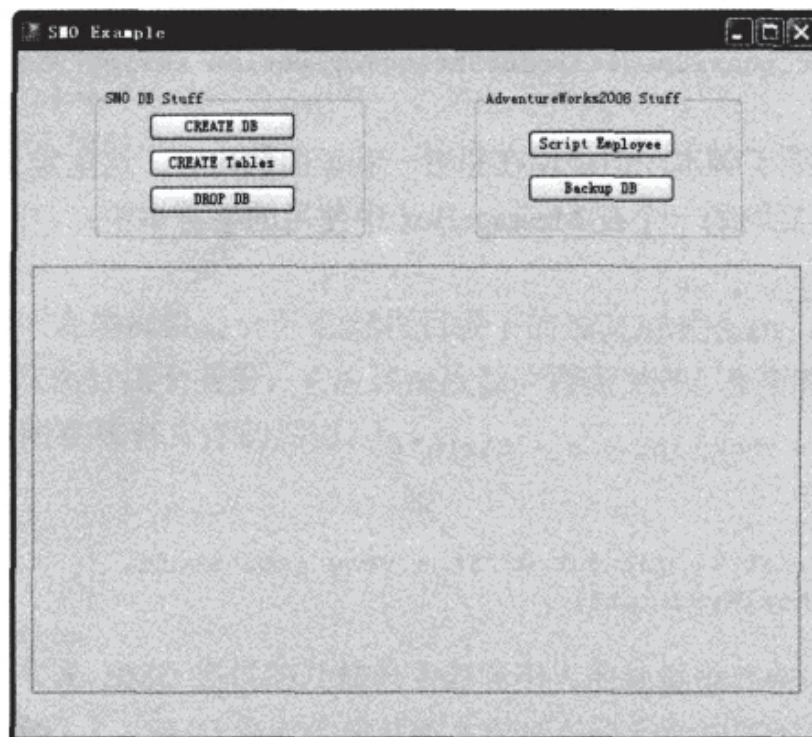


图 23-3

下面是完整的窗体代码：

```
using System;
using System.Text;
using System.Windows.Forms;
using Microsoft.SqlServer.Management.Smo;
using Microsoft.SqlServer.Management.Common;
using Microsoft.SqlServer.Management.Smo.SqlEnum;

namespace SQLSMOSample
{
    public partial class frmMain : Form
    {
        public frmMain()
        {
            InitializeComponent();
        }

        private void btnBackupDB_Click(object sender, EventArgs e)
        {
            // Create the server and connect to it.
            ServerConnection cn = new ServerConnection();
            cn.LoginSecure = true;
            Server svr = new Server(cn);
            svr.ConnectionContext.Connect();
        }
    }
}
```

```

        // Create and define backup object
        Backup bkp = new Backup();
        bkp.Action = BackupActionType.Database;
        bkp.Database = "AdventureWorks2008";
        bkp.Devices.AddDevice(@"c:\SMOSample.bak", DeviceType.File);

        // Actually start the backup. Note that I've said to do this Asynchronously
        // I could easily have make it synchronous by choosing SqlBackup instead
        // Also note that I'm telling it to initialize (overwrite the old if it's there).
        // Without the initialize, it would append onto the existing file if found.
        bkp.Initialize = true;
        bkp.SqlBackupAsync(svr);
        cn.Disconnect();
    }

private void btnCreateDB_Click(object sender, EventArgs e)
{
    // Create the server and connect to it.
    ServerConnection cn = new ServerConnection();
    cn.LoginSecure = true;

    Server svr = new Server(cn);
    svr.ConnectionContext.Connect();

    Database db = new Database();

    db.Parent = svr;
    db.Name = "SMODatabase";
    db.Create();

    txtResult.Text = "Database Created";

    cn.Disconnect();
}

private void btnScript_Click(object sender, EventArgs e)
{
    // Create the server and connect to it.
    ServerConnection cn = new ServerConnection();
    cn.LoginSecure = true;

    Server svr = new Server(cn);
    svr.ConnectionContext.Connect();

    // Now define the database we want to reference the table from.
    Database db = svr.Databases["AdventureWorks2008"];

    // Get a reference to the table. Notice that schema is actually the *2nd* parameter
    // not the first.
    Table Employee = db.Tables["Employee", "HumanResources"];

    // Call the Script method. The issue with this is that it returns a string
    // *collection* rather than a string. We'll enumerate it into a string shortly.
    System.Collections.Specialized.StringCollection script = Employee.Script();
    string MyScript = "";

```



```
        foreach (string s in script)
        {
            MyScript = MyScript + s + "\r\n";
        }

        // Now show what we got out of it - very cool stuff.
        //MessageBox.Show(MyScript);
        this.txtResult.Text = MyScript;

        cn.Disconnect();
    }

    private void btnDropDB_Click(object sender, EventArgs e)
    {
        // Create the server and connect to it.
        ServerConnection cn = new ServerConnection();
        cn.LoginSecure = true;

        Server svr = new Server(cn);
        svr.ConnectionContext.Connect();

        Database db = svr.Databases["SMODatabase"];

        db.Drop();

        txtResult.Text = "Database Dropped";

        cn.Disconnect();
    }

    private void btnCreateTables_Click(object sender, EventArgs e)
    {
        // Create the server and connect to it.
        ServerConnection cn = new ServerConnection();
        cn.LoginSecure = true;

        Server svr = new Server(cn);
        svr.ConnectionContext.Connect();

        // Get a reference to our test SMO Database
        Database db = svr.Databases["SMODatabase"];

        // Create Table object, and begin defining said table
        Table ParentTable = new Table(db, "ParentTable");

        // Build up the table definition
        Column ParentKey = new Column(ParentTable, "ParentKey");
        ParentKey.DataType = DataType.Int;
        ParentKey.Nullable = false;
        ParentKey.Identity = true;

        Column ParentDescription = new Column(ParentTable, "ParentDescription");
        ParentDescription.DataType = DataType.NVarCharMax;
        ParentDescription.Nullable = false;
```

```

// Now actually add them to the table definition
ParentTable.Columns.Add(ParentKey);
ParentTable.Columns.Add(ParentDescription);

// Add a Primary Key
Index PKParentKey = new Index(ParentTable, "PKParentKey");
PKParentKey.IndexKeyType = IndexKeyType.DriPrimaryKey;

PKParentKey.IndexedColumns.Add(new IndexedColumn(PKParentKey, "ParentKey"));

ParentTable.Indexes.Add(PKParentKey);

ParentTable.Create();

// Create Table object for child, and begin defining said table
Table ChildTable = new Table(db, "ChildTable");

// Build up the Child table definition
Column ChildParentKey = new Column(ChildTable, "ParentKey");
ChildParentKey.DataType = DataType.Int;
ChildParentKey.Nullable = false;

Column ChildKey = new Column(ChildTable, "ChildKey");
ChildKey.DataType = DataType.Int;
ChildKey.Nullable = false;

Column ChildDescription = new Column(ChildTable, "ChildDescription");
ChildDescription.DataType = DataType.NVarCharMax;
ChildDescription.Nullable = false;

// Now actually add them to the table definition
ChildTable.Columns.Add(ChildParentKey);
ChildTable.Columns.Add(ChildKey);
ChildTable.Columns.Add(ChildDescription);

// Add a Primary Key that is a composite key
Index PKChildKey = new Index(ChildTable, "PKChildKey");
PKChildKey.IndexKeyType = IndexKeyType.DriPrimaryKey;

PKChildKey.IndexedColumns.Add(new IndexedColumn(PKChildKey, "ParentKey"));
PKChildKey.IndexedColumns.Add(new IndexedColumn(PKChildKey, "ChildKey"));

ChildTable.Indexes.Add(PKChildKey);

// Add a Foreign Key
ForeignKey FKParent = new ForeignKey(ChildTable, "FKParent");

// The first "Parent Key" in the definition below is the name in the current table
// The second is the name (of just the column) in the referenced table.
ForeignKeyColumn FKParentParentKey = new ForeignKeyColumn(FKParent,
    ParentKey, "ParentKey");

FKParent.Columns.Add(FKParentParentKey);

FKParent.ReferencedTable = "ParentTable";

// I could have also set a specific schema, but since the table was created
// using just a

```



```

// default schema, I'm leaving the table reference to it default also. They would be
// created using whatever the user's default schema is

/*
** Note that there are several other properties we could define here
** such as CASCADE actions. We're going to keep it simple for now.
*/

ChildTable.ForeignKeys.Add(FKParent);

ChildTable.Create();

cn.Disconnect();

txtResult.Text = "Tables Created";

}
private void frmMain_Load(object sender, EventArgs e)
{
}
}
}

```

23.8 小结

好吧，我所能说的是“太棒了！”尽管从某种程度上来说，这并不是什么新东西——毕竟，DMO 过去也能完成很多这种类型的任务(实际上，几乎包括了上面用实际代码完成的所有任务)。但是，SMO 使事情变得更加简单了。至于那句“太棒了！”则可以考虑下列几种可能性：

- 想象一下异步地发出一个命令。
- 想象一下随着进度的往前推进，接收事件以监控那些命令的进度。
- 想象一下生成脚本代码以支持几乎任何工作。
- 想象一下在 SQL Server 上注册事件处理函数，然后当服务器上发生自定义的事件时接收通知。

这个列表还在不断地增长。

本章中大多数的概念都不是新的。我们已经介绍了如何创建表以及创建、备份和删除数据库。但是，其强大的地方是使用 SMO 可以分别管理那些任务。现在有望拥有非常可靠的事件和错误处理机制，因此与试图从系统存储过程中解析出属性值相比，以单独属性的形式接收已经存在于服务器的对象的配置信息要容易得多。

对于使用 SMO 所能完成的任务来说，本章确实只介绍了一点皮毛。如果本章激发了你的兴趣，那么笔者建议你考虑在设计工作中使用 SMO，当然如果需要(你很有可能会需要!!!)，还应该阅读一些专门介绍 SMO 的书籍。

第24章

数 据 仓 库

好吧,看起来现在已经介绍了 SQL Server 的各个方面。到现在为止,使用类型最常见的并且大多数数据库开发人员最熟悉的数据库并不会有什么问题:联机事务处理——或者 OLTP——数据库。

然而,本章将会把事情稍微颠倒一下(相对于传统的决定如何做事情的“规则”)。例如,在本书前面或者入门书籍中讨论设计的时候,大多数内容是根据规范化的数据库来讨论的。在本章中将会抛弃这些概念。与到目前为止主要关注面向事务的数据库不同,本章将会把精力集中在面向数据分析的数据库和模型上。目前将主要关注数据仓库和与它的存储衍生物有关的特殊需求以及数据仓库环境中的报表问题。这里将会介绍很多读者以前可能没有听过的术语——数据仓库和分析方面的术语——商业智能(通常简单地使用 BI 来表示)的语言。此外,本章还会通过快速介绍一下 SQL Server 中的另一个服务——分析服务——来探索多维建模领域。

本章将会:

- 讨论事务处理和分析处理的需求之间的差异。
- 讨论这些差异如何导致完全不同的解决方案。
- 探讨把 OLTP 解决方案作为 OLAP 解决方案存在的问题。
- 定义数据立方体的概念,并指出它们如何帮助为一个分析环境的特殊需求提供解决方案。
- 介绍 SQL Server 2008 中分析服务的其他一些方面。

24.1 考虑不同的需求

随着企业构建的应用程序的复杂度的增长以及需要把日常数据存储到数据库中以支持那些应用程序,数据库的规模也在不断膨胀。随着数据库规模的增长,它们通常会对应用程序所使用的系统的性能产生负面的影响。如果不进行检查,那么数据库的规模可能会增长到严重影响响应时间、竞争增加(试图获取同一数据的用户之间的冲突)或者甚至导致整个系统宕机的地步。

最终用户相互之间使用数据源的方式可能会不同。从“他们如何使用数据源”的角度来看,可以把用户分成四个主要的类别:

- 每天都需要访问数据源的人,如检索特定的记录、添加新记录、更新记录或者删除现有的记录。

- 想要弄清楚数据库中庞大的数据的含义的人,如生成报表以帮助为企业做出正确的决策,从而为企业提供竞争优势以在市场上获得成功。
- 想要在从分析或事务系统中获取的知识的基础上进一步预测业务表现和分析将来趋势的人。
- 想要使用高度集中的“快速查询”信息以快速定位应该把时间花在哪里的人。

把 OLTP 和 OLAP 系统分开能够满足前两种用户的不同需求。数据挖掘和立方体分析(通过数据透视表和其他的“*What if?*”分析)能够满足第三种用户的需求。而列表中的最后一项往往是通过目标屏幕——或者“仪表盘”——来解决的,当用户首次登录到系统中的时候通常会出现这个屏幕。下面几个小节将介绍这些系统的特点和技术,以及如何和何时使用它们。

24.1.1 联机事务处理(OLTP)

正如之前所提到的那样,到目前为止一直关注的 OLTP 系统是设计用来允许高并发性的,这样很多用户就能够访问同一个数据源并进行所需的处理。它们往往也是大多数数据的“权威系统”,因此它们非常看重数据完整性。此外,它们往往会在一个详细的级别上存储数据,因此需要制定一些策略使得存储这些数据所需的空间最小。

正如“事务处理”这个名字所蕴含的那样,OLTP 系统是面向在数据库上进行事务处理的理念的。而事务则进一步蕴含着发生在表中数据上的受控的变更,这些变更包括在商务运作过程中发生的插入、更新和删除操作。通常,一个 OLTP 系统将会有大量的客户端应用程序通过各种各样的方式(插入、更新、删除——实际上可以是任何操作)访问数据库以查询一小块信息。

OLTP 系统的实例包括数据输入程序,如银行处理、订票、联机销售和库存管理系统(如 AdventureWorks2008),但是不管是何种应用程序,OLTP 系统的构建通常要实现下列目标:

- 处理由事务产生的数据。
- 通过移除数据冗余来维持高度的精确性。
- 确保数据和信息完整性。
- 产生及时的(更一般一点,“实时”)文档和报表,如收据和发票。
- 提高工作效率。

由于关注的是这些特定的目标,因此数据库的设计通常要遵循第 5 章中讨论的第三范式,以移除冗余性和最大化表之间关系的威力。

24.1.2 联机分析处理(OLAP)

联机分析处理(或 OLAP)是一种广义上的决策支持系统(DSS),或者最近越来越流行的商业智能(BI)。BI 系统的目标是分析海量数据,然后以很多不同的方式(包括每天、每周、每季和年度报告)生成小结和总结以把精力高度集中在记分卡和仪表盘上,它们通常用于帮助那些准备好根据这些数据采取一定的措施的特定用户来获取竞争优势。

在 OLAP 和 BI 中通常会忘记保持数据的规范化。实际上,有时候会特意在一定程度上去除数据库的规范化(或者使它变平)以允许一定的冗余,从而能够避免连接并集中提高数据检索的性能,而不是数据修改的性能。为什么这种方式在数据仓库中是可行的呢?好吧,一旦数据进入数据仓库之后就很少会发生变化。数据被保存在那里用于查询和生成报表,以便帮助决策者规划企

业的未来。但是由于当数据进入数据仓库之后通常就会被视作历史，因此它不需要关心插入、更新和删除操作。因此与高度规范的事务数据库不同，在这种情况下通常会使用所谓的维度数据库(dimensional database)，它将遵循特定的结构或模式。维度数据库可以用来构建数据立方体，数据立方体是数据的多维表示，用来方便联机业务分析和提高查询性能。立方体中的每一维都表示业务数据中的一个不同的分析类别。典型的立方体中的维几乎总是包含时间信息，而且通常还会包含地理位置信息和类似于产品线的信息。这种可能的组合是无穷无尽的，具体需要根据组织的特点。

注意：

不要因为它叫做“立方体”就陷入它仅仅局限于三维的思维陷阱。立方体允许 n 维的查询。“立方体”表示仅仅意味着已经超过了 OLTP 系统中经常见到的典型的表结构表示。

24.1.3 数据挖掘简介

传统的查询技术(如本书中花大部分章节所讲述的查询)和对数据仓库的查询能够帮助从数据中找到有用的信息，而这些数据是以可能已经知道的关系为基础的(它们可能在事务系统中被声明)。例如，可以使用查询或者甚至是立方体来找到每个州或城市中在特定的一段时间内购买特定商品的顾客的数量。这里所探寻的信息已经存在于数据库中了，同时这些查询通常是根据一些直观的问题来检索信息的。

另一方面，数据挖掘的强大之处在于它能够帮助发现数据中隐藏的关系。可以使用它来发现新的趋势、推测特定事件的原因或者甚至预测数据的特定方面的性能或发展方向。例如，数据挖掘可能会帮助发现为什么在特定的区域某种特定的产品比另一种产品卖得好。数据挖掘所使用的算法能够揭露非直观的关系。例如，数据挖掘在很多年前发现购买啤酒的人会更倾向于同时购买奶酪。当零售商发现了这个关系之后，一种常见的做法是把奶酪放在啤酒的旁边以方便和促进那些产品一起销售，而不是每次只销售一种产品。

SQL Server 2008 继续为数据挖掘提供了强有力的支持。然而，数据挖掘是一门很复杂的学科，它已经超出了本书的范围。笔者只是希望读者能够知道有这样的工具，这样就可以使用这些工具来帮助分析以进行数据挖掘。

24.1.4 OLTP 与 OLAP

现在读者已经了解了 OLTP 和 OLAP 两种系统背后的基本思想了，下面考虑一个银行业务。在银行的工作时间内，银行出纳员帮助客户执行事务，如把钱存到他们的帐户中、在帐户之间进行转帐和从这些帐户中取款。客户可能也会使用 ATM(自动取款机)或者使用基于电话和/或计算机的银行服务来执行他们自己的事务。换句话说，这种事务的发生不局限于一天中的特定时间，它们随时可以发生。所有这些操作都会导致存储在数据库中的数据发生变更。这些变更可能是插入新记录、更新记录或者删除现有的记录。

OLTP 允许大量用户并行地访问数据库来执行这些事务。服务于 OLTP 系统的数据库通常是高度规范的关系数据库，同时需要小心地针对正确的字段为其中的表选择索引。OLTP 数据库应该要平衡从报表到高频率的事务操作之间的性能。在 OLTP 系统中执行的查询将会包含大量的插入、更新、删除和选择操作以及这些操作的组合。

下面来看看银行业务例子中的几个不同的场景。假设银行经理正在为将来做规划，他们需要查看银行数据的当前成绩和历史成绩。如果他们查询用于 OLTP 系统的数据库，那么他们很有可能就会与那些进行银行的日常业务的员工产生竞争。银行管理层将要查看的各种各样的报表和分析通常需要运行很长时间，同时这些任务可能会给事务系统带来巨大的负载，因为它需要把很多表连接起来以把各种各样的信息关联起来并通过某种格式来对数据进行总结和聚合。例如，他们可能想要知道某个特定区域中所有客户产生的事务量。这种查询需要对大量的数据进行筛选，而这些数据是散落在很多参与连接的表中的。例如，一个会计总帐事务可能会被存储在一打不同的表中。而查询则需要从这些被连接的表中抽取字段来构建管理层所需的视图，并对这些数据进行分组和聚合。现在想象一下需要一次又一次地重复执行这个过程，因为多个经理问了同样的问题并需要查看同样的数据。

为了应对这些挑战，需要把那些使用现有的银行数据对未来进行规划和展望的经理分离开来，让他们使用一个不同的系统，该系统是根据 OLAP 原则来构建的。这意味着要创建两种不同的系统：一个是用于事务处理的 OLTP 系统，由银行职员和客户使用；另一个是 OLAP 系统，用来帮助决策。

现在有了两种不同的系统，那么它们应该使用同一个数据库但每个系统使用不同的表呢？还是应该使用两个完全不同的数据库？这个问题的答案依赖于其中一个系统对另一个系统的性能的影响有多少以及这些系统的管理工作是如何开展的。如果两个系统很有可能会被同时使用，那么即使表是分开的也会造成性能问题。这是因为两个系统仍然共享了数据库服务器上的很多资源，而这些资源可能很快就会被两个系统耗尽。还有两个系统的优化方式通常是不同的。如果对 OLAP 进行优化，那么可能会对 OLTP 系统的性能造成负面影响，反之亦然。还有，两个系统的管理方式可能也是不同的，它们有不同的用户帐户、不同的备份和维护策略等等。因此，尽管在理论上可以让它们使用同一个数据库，但是更好的做法是为两个系统分别建立数据库并把它们部署在不同的数据库服务器上。采用这种方案之后每个系统就会有自己的资源，同时进行优化也不会影响到另一个系统。

24.2 维度数据库

在 OLTP 系统中进行复杂查询存在一些固有的问题，对这些问题的解决方案是构建一个单独的数据库来更简洁地表示业务事实(fact)。这个数据库的结构不是关系型的，相反，它是维度化的。

24.2.1 事实表

维度数据库的中心表叫做事实表。它的行表示事实，事实表的中心内容是对一个活动或事件的不同实例的测量。

例如，AdventureWorksDW2008 数据仓库示例中包含了一张名为 FactInternetSales 的表以及几张关联表(如图 24-1 所示)。它关注的是单个销售，但是在项目级别则主要关注每个销售的关键指标。它保存着一组测量(通常是数值)——在本例中是 Order Quantity、Unit Price、Extended Amount 还有其他一些测量——并且把它们与恰当地维关联起来(在本例中将会根据产品信息、相关日期、客户信息以及其他一些维来进行分析)。

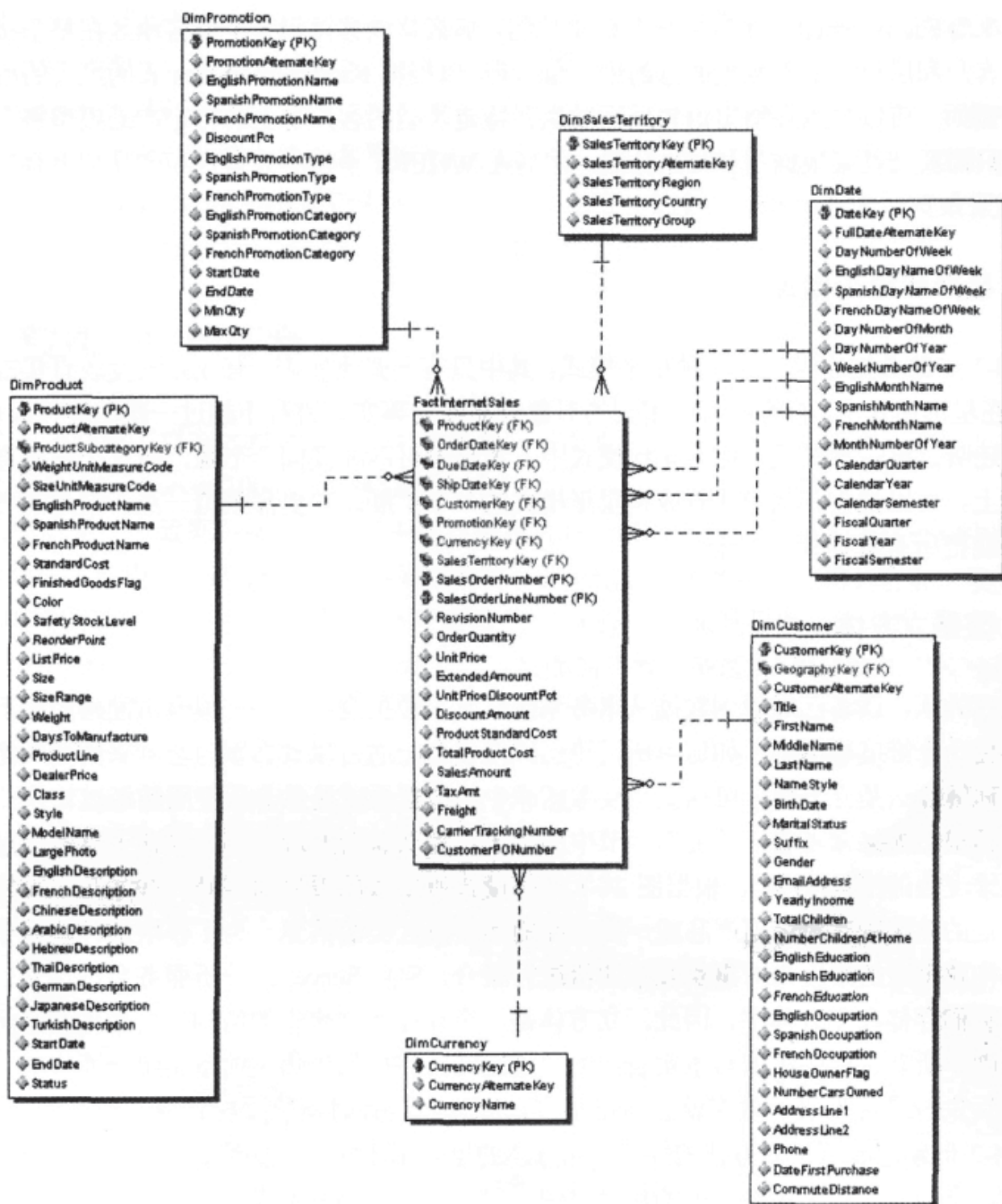


图 24-1

24.2.2 维度表

维度把事实放到上下文中，它表示诸如时间、产品、客户和位置之类的事物。维描述了事实表中的数据。在本节的 AdventureWorksDW2008 示例中，一个恰当的做法是选择日期、客户和产品作为维度。

事实表 FactInternetSales 抓取与每天发生的事务有关联的所有客户和产品。由于每一个明细行有一行相应的数据，因此这张表可能会变得非常大(或者至少我们希望如此，因为这意味着做了很多生意!)。由于为每一单生意都存储客户数据将会占用大量的空间，并且通常无法提供那么多的空间，因此需要把那些不是每个测量实例都发生变更的项目分离出来。这些链接到事实表上的表叫做维度表。它们被用来创建分组，而根据这些分组可以确定事实表中的聚合级别。例如，如果查询 FactInternetSales 表并按照月来分组，那么就可以找到所有销售地区中所有产品每月总的销售

量。如果查询 FactInternetSales 表并按照州来分组,那么就能够找到某个销售地区在整个统计时间内在所有客户和所有产品上发生的总的销售量。还可以根据 FactInternetSales 表的维度的组合来进行聚合。例如,可以找到在特定销售地区发生在特定类型的客户身上的特定产品模型每个月总的销售量,只需要把结果根据州和月来分组,然后在 WHERE 子句中为客户和产品加上合适的标准即可。

24.2.3 星形和雪花模式

图 24-1 中的数据库模式是一种星形模式,其中只有一张事实表,有几张维度表直接链接到事实表上。在星形模式中,查询中可能用到的对象只需要与事实表进行不超过一次的连接即可获得。读者可能还听过雪花状模式。在雪花状模式中,多张表可能涉及同一个维,而这个维将直接链接到事实表上。可以把雪花状模式看成是星形模式的一个扩展,它更加规范一点,但是需要连接额外的表才能把所有数据关联起来。

24.2.4 数据立方体

到现在为止,读者已经看到数据从事务系统移到了数据仓库中——很有可能采用星形或雪花状模式。在一个维度模型中,如这里所讨论的维度模型,通常以数据库为基础来构建所谓的立方体。为了理解什么是立方体,可以把维度数据库中的数据看成是供分析使用的经过转换的原始数据。换句话说,如果读者看一看前面一节中的例子,那么就会发现事实表包含了事务信息和指向待分析的维度表的指针(外键)。根据图 24-1 中的模式所生成的报表通常是一些如在一段特定的时间内在特定的地区顾客对特定产品或一类产品的购买量之类的信息。为了获取这样的结果,需要根据用来构建报表的维度来对事实表中的值进行聚合。SQL Server 的分析服务允许预先计算这种结果并把它们存储在立方体中。因此,立方体是一个存储数据聚合的结构,而数据聚合是通过把维度数据库中所有可能的维值与事实表中的通过因特网完成的销售量的事实组合而来的。采用这种方式之后生成最终报表的效率就会高出很多,因为在运行时不需要执行复杂的查询。

图 24-2 形象地展示了立方体的样子。立方体的维表示事实表中的维。立方体的每个单元表示一个事实,它与立方体中不同维度的粒度对应。尽管立方体的图形表示只能显示三维,但是在使用分析服务时数据立方体可以拥有更多的维。下图显示了 FactInternetSales 表的数据立方体表示,其中显示了地区、产品类别和时间维度。

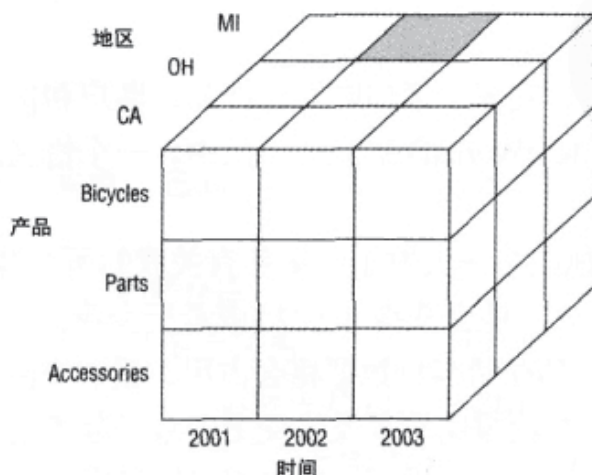


图 24-2

如果想要使用这个立方体找到密歇根州在 2002 年自行车的销售量,那么只需要查看图中用阴影显示的单元即可,取三个维度的交集即可得到这个单元。

分析服务允许从任何拥有 OLE DB 提供者的数据源中构建立方体。它可以是任何拥有 ODBC 驱动程序的数据库管理系统中的关系数据库(如 Oracle、DB2 甚至是 MySQL)或者是本地的 OLE DB 提供者(如 SQL Server、Oracle 或者 MS Access)。立方体的数据源还可以是维度数据库、文本文件甚至可以是一个轻量级目录访问协议(LDAP)数据源。

24.3 数据仓库的概念

现在已经介绍了立方体和维度数据库的概念,下面来定义一个更大的概念,即什么是数据仓库以及如何在 SQL Server 2008 中构建数据仓库。

数据仓库是一个数据存储中心,其中保存着在很长一段时间内公司业务运作过程中收集的数据。数据仓库可能由一个或多个数据市场(规模较小的总结或维度数据的集合,它们一般关注整个数据仓库中数据的一个子集)组成。数据仓库通常使用 OLTP 系统来从日常活动和事务中收集数据。数据仓库的概念中还包含着提取、清洗(参见本章后面的“数据清洗”)和转换数据的过程,通过这个过程可以为数据仓库把数据准备好。最后,它还包括业务分析人员呈现和使用数据所需的工具。这些工具包括 BI 工具(如 Excel 中的数据透视表或者 Performance Point Server)以及数据挖掘和报表工具。图 24-3 显示了数据仓库解决方案概念上的结构及其组成部分。

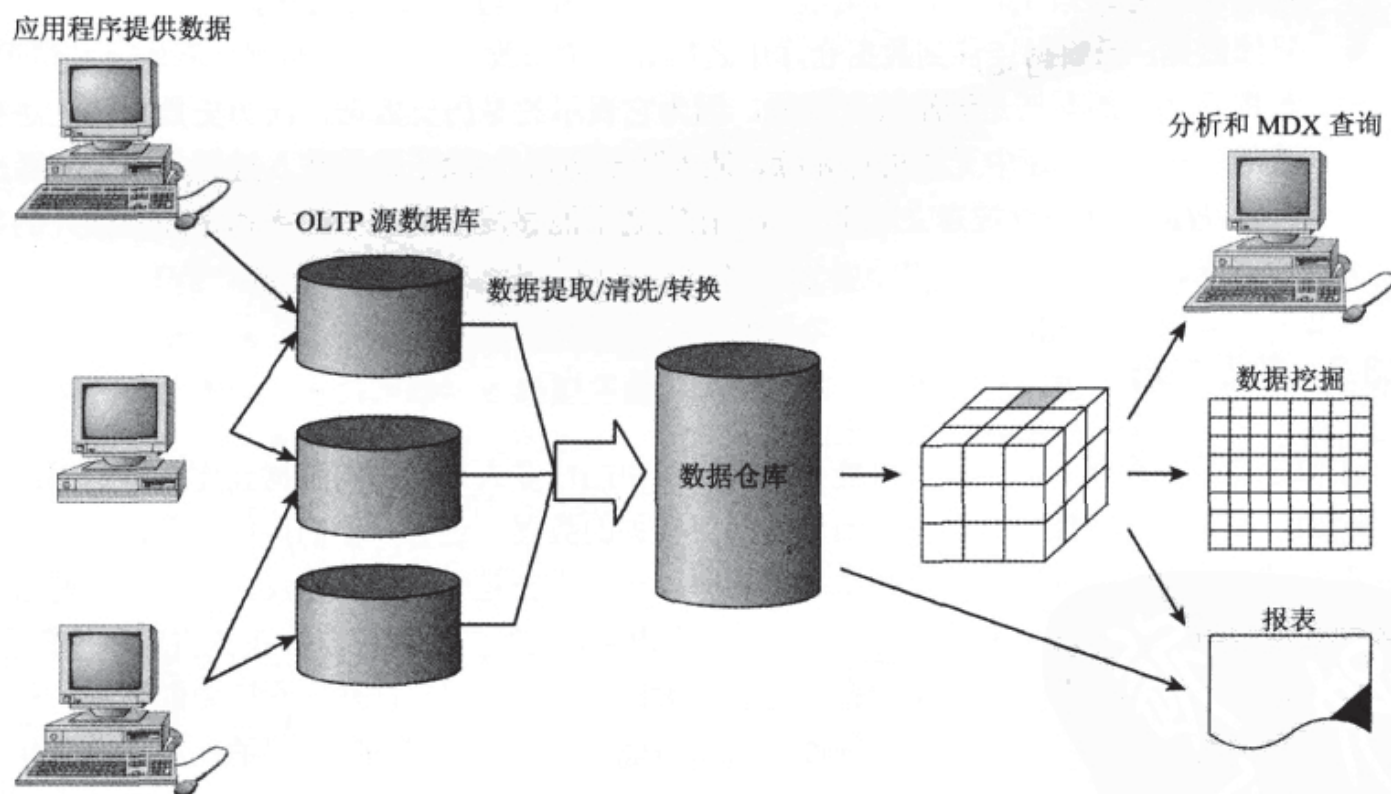


图 24-3

24.3.1 数据仓库的特点

数据仓库通常用来支持决策制定和分析,因为它拥有下列独特的特点:

- **条理清晰并且一致的数据：**在数据仓库中，数据是从不同的源收集而来的，之后还要通过多种方式对其进行整理并且使其一致，所采用的方式包括使用命名约定、度量、物理属性和语义。这样做是非常重要的，因为访问数据并使用其中的数据进行决策的业务分析人员需要使用一致的标准。例如，日期格式可能会遵循一种标准，如显示天、月、季度和年。数据应该以单一的、可接受的格式存储在数据仓库中。这样就可以引用、整理数据以及交叉引用来自多种异类源的数据，如大型机上的遗留数据、电子数据表中的数据甚至是因特网上的数据，进而使得分析人员能够更好地了解业务。

提示：

再怎么强调需要一致地对待数据也不过分，包括引用这些数据的名称。确保在数据库中没有使用同样的名字引用了不同的东西。例如，如果需要引用的销售额类型多于一个，那么销售的每个实例都需要使用限定名——如在“总销售额”中为“自行车销售额”和“服装销售额”使用不同的名称。笔者强烈推荐使用一个数据“字典”来定义所使用的每个名称的含义以及该数据的源。

- **面向主题的数据：**数据仓库将会把来自 OLTP 源的关键业务信息组织起来，这样在进行业务分析时就可以使用这些信息了。在这个过程中，数据仓库将会把源数据存储中不相关的数据清除掉，同时根据数据的主题来组织数据，即把客户信息和产品信息分离开来，而在源数据存储中，这些信息可能是混合在一起的。
- **历史数据：**与 OLTP 系统不同，数据仓库表示的是历史数据。换句话说，在查询数据仓库的时候所使用的数据是以前用 OLTP 系统收集的。与 OLTP 系统包含能够精确地描述系统的当前数据比起来，历史数据覆盖的是很长一段时间内的数据。
- **只读数据：**当数据被移到数据仓库中之后就无法修改它了，除非数据一开始就是错误的。数据仓库中的数据是无法被更新的，因为它表示的是历史数据，而历史数据是无法被修改的。在数据仓库中无法进行删除、插入和更新操作(除了数据载入过程中发生的操作之外)。数据仓库一旦被建立起来之后，在它之上能够发生的唯一的操作是加载额外的数据和查询。

24.3.2 数据市场

在构建完数据仓库之后，读者可能会发现组织中的很多人只会访问数据仓库中特定部分的数据。例如，销售经理可能只会访问与他们的部门相关的数据。还有，他们可能只会访问去年的数据。在这种情况下，让这些人查询整个数据仓库以获取报表是低效的。相反，一个明智的做法是把数据仓库划分为较小的单元，这些单元叫做数据市场，它们是根据各自的业务需求而被创建的。

此外，组织中的一些人可能希望在远离公司大楼的地方能够访问数据仓库中的数据。例如，一个销售经理可能希望在谈生意的时候能够访问与他或她负责的市场区域中的产品和销售相关的数据。这类人员将会从数据市场中受益，因为他们能够使用他们的笔记本电脑来携带数据仓库中的一部分数据，从而使得他们能够在任何需要的时候访问这些数据。

注意：

实际上，这个过程往往是以相反的顺序进行的，即一个较小的数据市场将会作为一个较大的数据仓库的前身。实际上，目前很多企业使用的数据仓库都是通过这样一个过程创建的，即在提

供额外的数据聚合和对各个数据市场中的数据进行汇总之前，在一个数据字典和一致的定义下统一多个相异的数据市场。

当然，数据市场中的数据应该时刻与数据仓库中的数据保持同步。这可以通过多种方式来实现，如使用 SQL Server 的集成服务、脚本(T-SQL 或其他语言)或者成熟的数据管理技术。

24.4 SQL Server 的集成服务

在第 16 章中已经详细介绍了集成服务，但是为了说明它能够以一致的方式用在数据仓库中，在这里重提一下是值得的。

很多组织需要把数据集中起来以提高决策的正确性。被集中的数据通常会以各种各样的格式存储并且来自很多不同的源。在很多情况下，在把这些数据存储到数据仓库中之前，需要对这些数据源中的行数据进行调整和转换。SSIS 是执行这类任务的绝佳工具，它提供了一种方式来把数据从源移到目标数据仓库中，并且在这个过程中会根据需要对数据进行验证、清洗、整理和转换。

24.4.1 数据验证

在数据被传送到目标数据仓库之前对数据进行验证是及其重要的。如果数据是无效的，那么在此数据上进行的业务分析的完整性是令人怀疑的。例如，如果其中一个字段是货币，同时在世界上的多个国家都存在 OLTP 数据源，那么就必须要把这个货币字段中的数据传送到目标数据仓库的货币字段中，同时必须要根据事务发生时的汇率对这个值作相应调整(不仅仅是兑换时那些当前值)。

24.4.2 数据清洗

通常，“清洗”的程度或性质决定了它无法在转换过程中执行进行。例如，可能需要在把数据填入同一个数据仓库的多个源之间调和数据。这个对多个数据源之间的数据进行调和并且在数据上应用其他一致性规则的过程叫做数据清洗。例如，如果一种自行车在一个源中被分为山地车的类别，而在另一个源中被分为娱乐的类别，那么在数据仓库中包含这一汇总将会出现错误的结果，除非在数据转换过程中对这两个数据源进行调和。

可以通过多种不同方式来实现数据清洗。对这些方法的详细介绍已经超出了本书的范围，这里只是简单提一下：

- 在把数据从源复制到目标数据存储时使用 SSIS 修改数据。
- 在临时“清洗”数据库或一组表上应用 T-SQL 脚本。

24.5 创建分析服务解决方案

本节将会快速看一看立方体能用来做什么以及如何创建立方体。接着将会通过一个例子来介绍一下如何使用立方体。这个例子是非常简单的，只是为了让读者感觉一下使用立方体能够做什么。如果读者在做完这个分析服务的例子之后想要了解更多内容，那么笔者建议你阅读专门介绍

分析服务的书籍和介绍数据仓库、商业智能以及维度建模的书籍。

提示:

意识到一个卓越的数据库开发者并不会自动成为一个伟大的数据仓库或商业智能系统的开发者实在是非常重要的。创建一个优秀的决策支持系统所需要考虑的因素与创建一个优秀的事务处理系统那个所需要考虑的因素有很大的不同。历史上出现过失败的项目,即使是一个看起来很有经验的数据库开发者向管理层保证他或她已经知道了数据仓库和分析的方方面面也无法保证项目一定会成功。因此在做出承诺之前确保已经了解了需要做什么。

注意:

在本章余下部分中介绍的例子需要使用 AdventureWorksDW2008 数据库。

启动 Business Intelligence Development Studio(或 BIDS)。虽然在前面使用 BIDS 的章节中已经讨论过,但是这里还是要重提一下,它仅仅是包含在 SQL Server 中的一个特殊版本的 Visual Studio 2008。接着选择“新建项目”。接着出现的画面可能会稍微有些不同,这取决于 Visual Studio 2008 是否与 SQL Server 是分开安装的以及分开安装时所采用的 Visual Studio 的版本。

不管是何种情况,最终都会出现一个类似于图 24-4 显示的对话框。确切的项目类型集合可能会稍微有些不同(同样,这取决于 Visual Studio 的版本)。这里选择了“商业智能项目”里的 Analysis Services 模板。

在为项目选择一个名称(本例选择了极具描述性的名称“AnalysisServicesProject”)之后单击“确定”创建这个项目。Visual Studio 将会创建一个空的分析服务项目,同时还会创建几个文件夹。本书中的例子将不会用到所有的文件夹,但是这些文件夹确实给人一种感觉,即通过分析服务项目可以完成很多任务。

下面继续向前来创建一个新的数据源。要完成这个任务,只需要右击“数据源”文件夹并选择“新建数据源”即可,如图 24-5 所示。

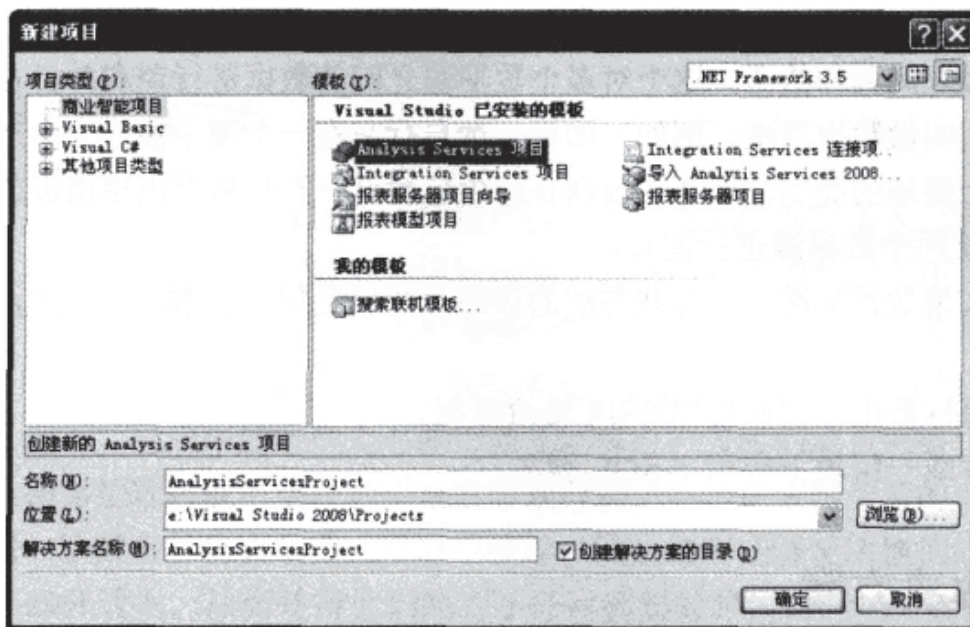


图 24-4



图 24-5

接着应该会出现一个欢迎对话框(除非这个对话框以前出现过,并且选中了“不要再显示本页”选项)。单击“下一步”进入下一个对话框,这个对话框允许选择一种定义数据源的方法。这里保

留默认的选项“根据一个现有或新连接创建数据源”，接着单击“新建”进入另一个对话框，如图 24-6 所示。

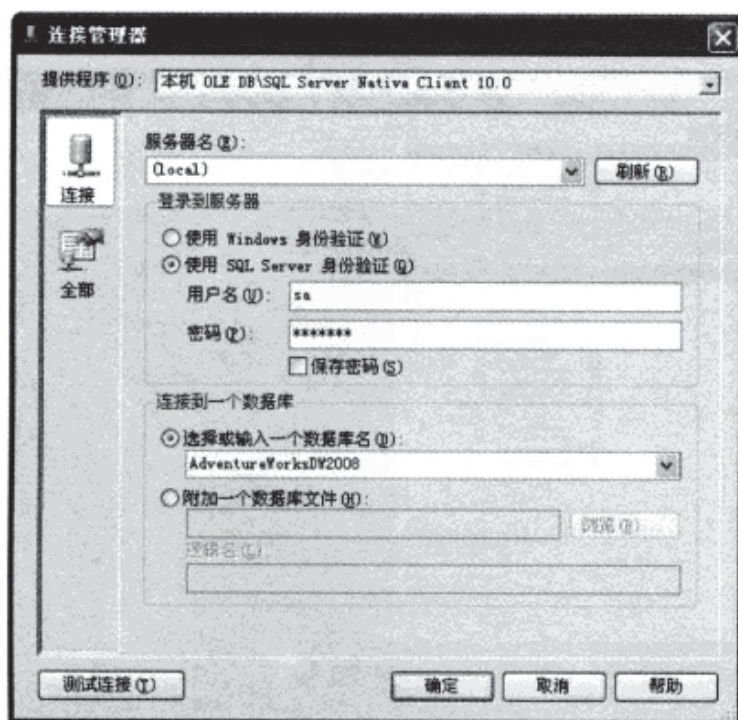


图 24-6

笔者已经填写了几个关键字段以符合特定的需求(读者可能希望选择一个远程服务器或者使用 Windows 身份验证)。单击“确定”创建该数据源并回到原来的对话框中(现在新数据源应该会显示在“数据连接”列表中)。单击“下一步”进入“模拟信息”对话框，如图 24-7 所示。这里可以使用四个安全选项中的一个来确定分析服务在需要连接到刚才定义的数据源上时传入的凭据。本例中选择使用服务帐户，它等同于运行分析服务的 Windows 帐户(因此，如果使用这个选项，那么需要确保帐户拥有访问源数据的权限)。

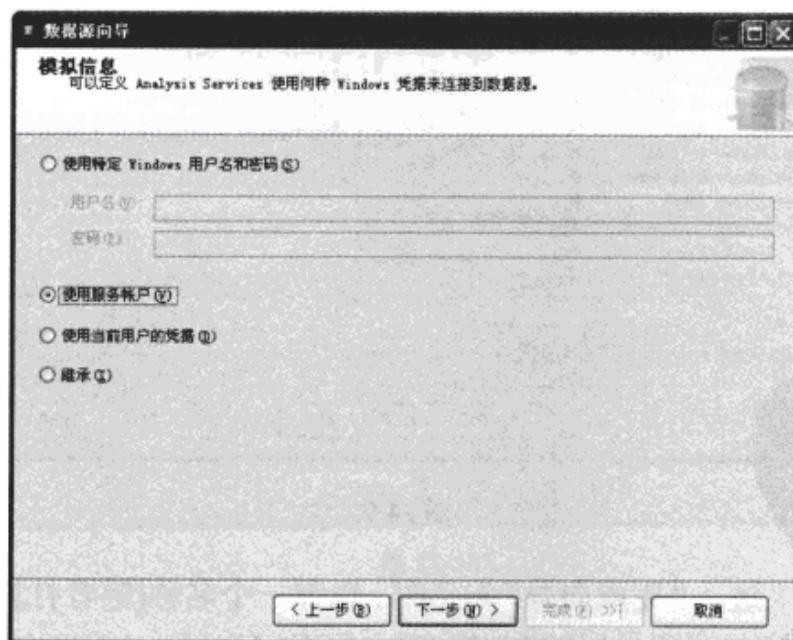


图 24-7

单击“下一步”将会进入“完成向导”对话框，在这个对话框中可以命名数据源并单击“完成”按钮。

接着右击“数据源视图”文件夹并选择“新建数据源视图”。接着会弹出一个对话框，如图

24-8 所示。读者可以看到，之前创建的数据源被列出来了并且默认是被选中的(它还提供了一个创建新数据源的快捷方式)。单击“下一步”选择所需的表和视图。这里选择了在本章前面的星形模式示例中见到的所有的表，如图 24-9 所示。

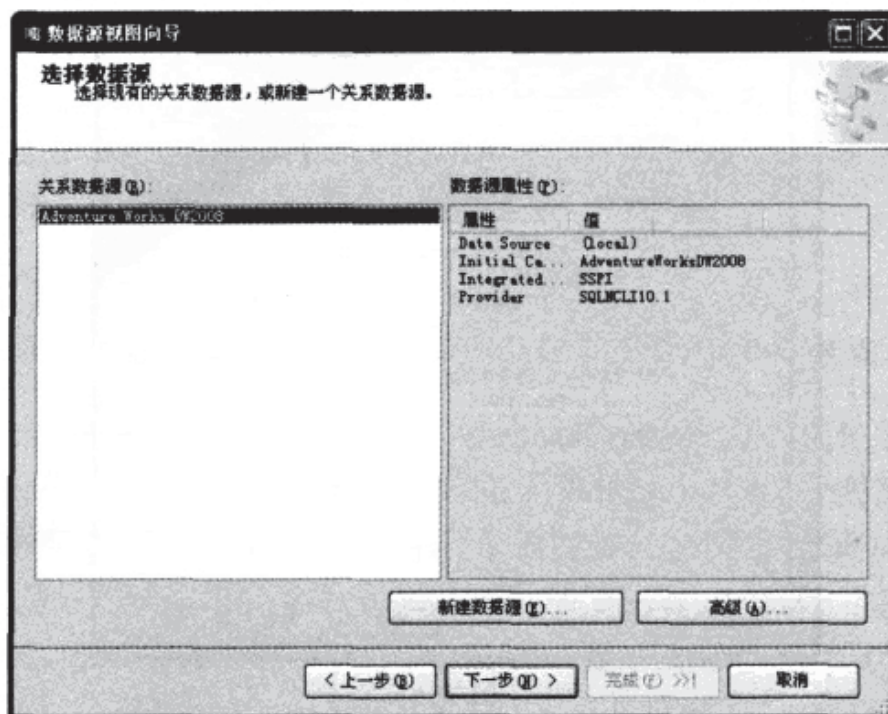


图 24-8

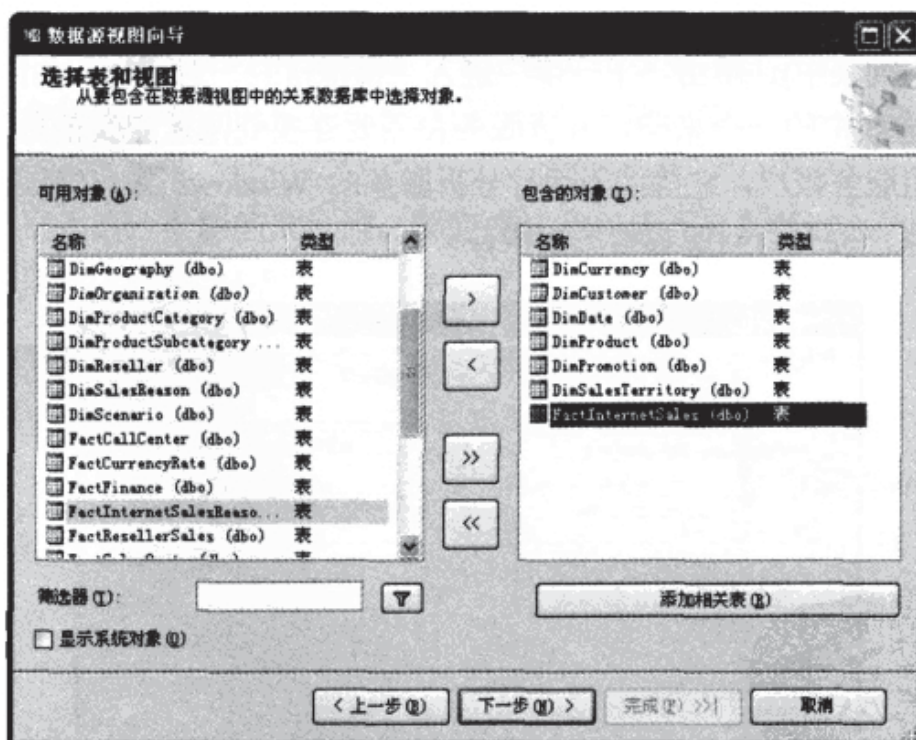


图 24-9

同样，单击“下一步”进入“完成向导”对话框。选择一个名称(笔者打算使用默认的 Adventure Works 2008)，然后单击“完成”。这次得到了更加戏剧性的结果，因为在主项目窗口(如图 24-10 所示)为新数据源视图打开了一个视图设计器。

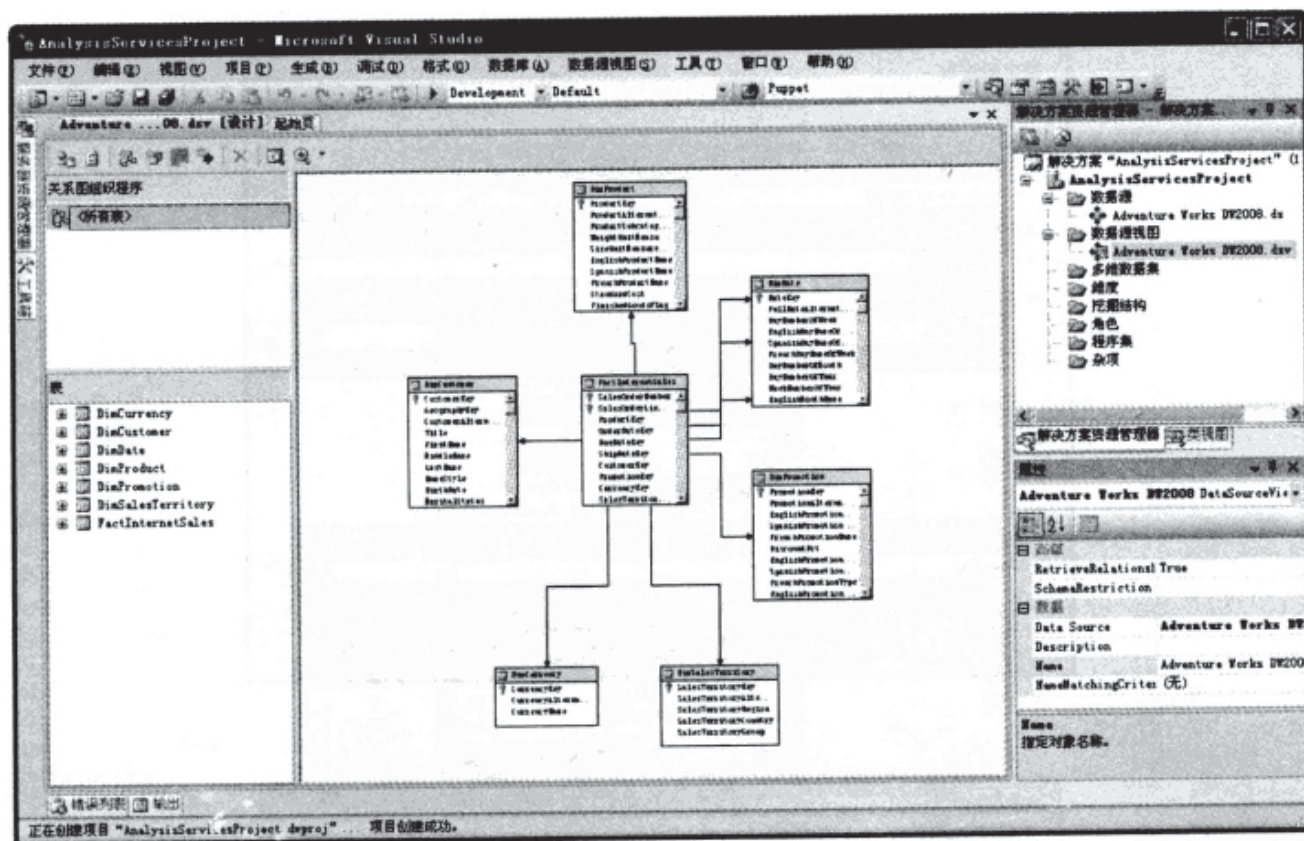


图 24-10

注意它已经知道表之间是相互关联的，甚至已经把图形映射成了美观的“星形”表示。

下面简要介绍一下“维度”文件夹。同样，右击并选择“新建维度”。单击“下一步”跳过欢迎对话框并进入“选择创建方法”对话框，如图 24-11 所示。

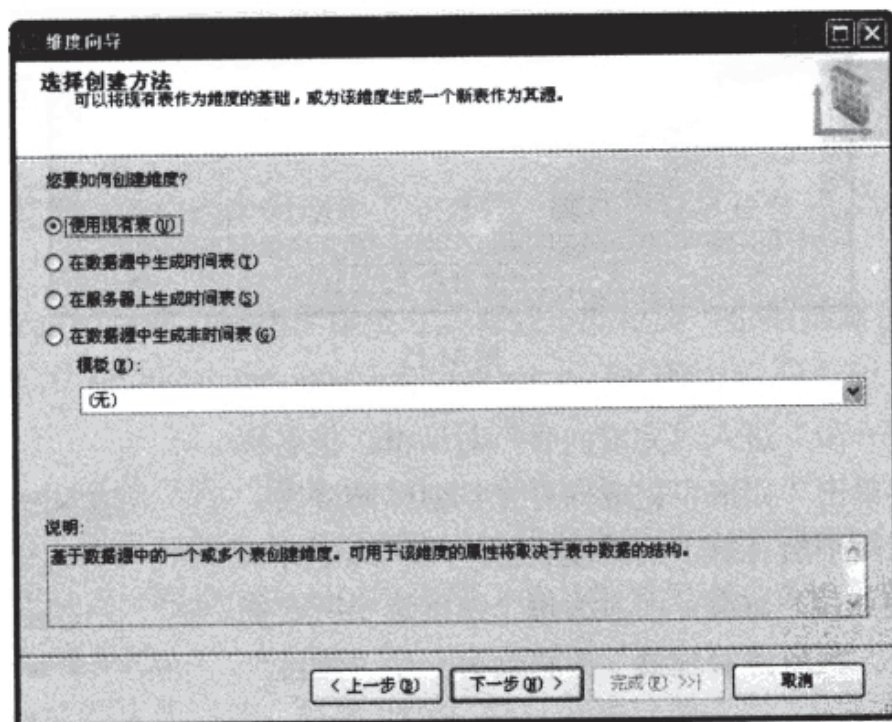


图 24-11

注意可以根据需要使用这里的实用工具来创建一个时间维度表(AdventureWorksDW2008 已经有了这样一张维度表)。保持默认的选项并单击“下一步”进入“指定源信息”对话框，如图 24-12 所示。这里选择了默认的 DimCurrency 表(这是根据字母顺序选择的)。它已经选择了正确的列作为关键字，因此只需要再次单击“下一步”进入“维度属性”对话框，如图 24-13 所示。注意这里添加了 Currency Name 作为一个属性。

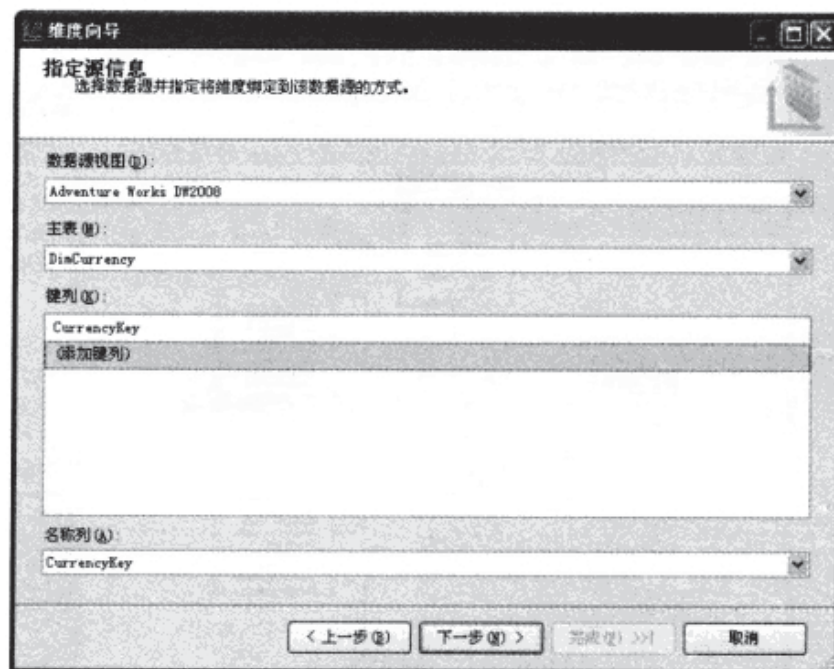


图 24-12

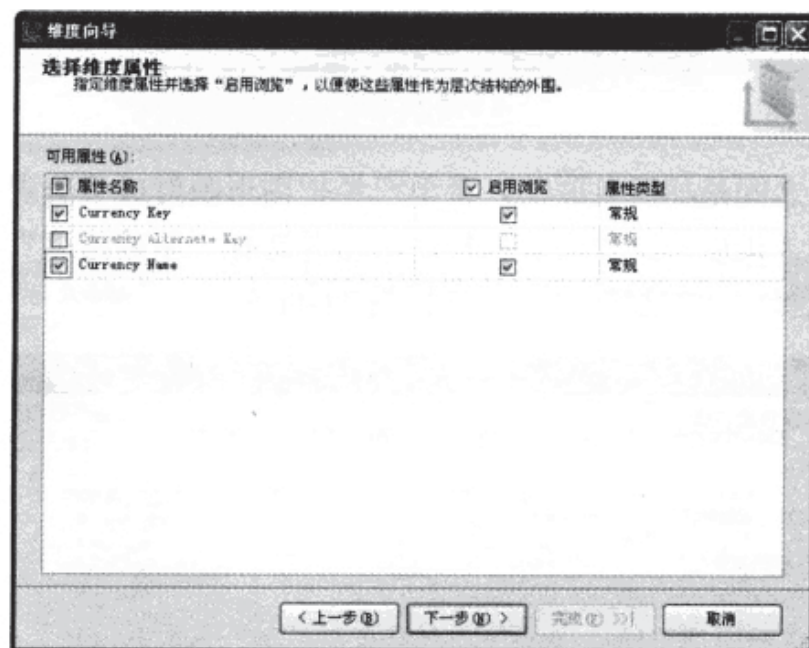


图 24-13

同样，单击“下一步”进入“完成向导”对话框。把名称改为 Currency，然后单击“完成”完成向导来创建该维度表。

现在为数据源视图中所有剩下的维度表(所有以 Dim 开头的表)重复执行“新建维度”过程，同时为每个维度选中所有的属性。最终在“解决方案资源管理器”中将会有有一个“维度”节点，如图 24-14 所示。

好，现在已经创建了维度，但尚未构建立方体。首先需要解决的问题是创建一个时间维度。“但是等等！”刚才说“我们已经有了一个时间维度。”如果读者这么认为，那么你是正确的。但是还有一个小问题。SQL Server 不知道它是一个时间维度。为了解决这个问题，选择“维度”下的 Date.dim 条目(如果在创建的时候没有重命名它，那么它仍然叫做 DimDate.dim)，然后查看左边的“属性”列表，如图 24-15 所示。

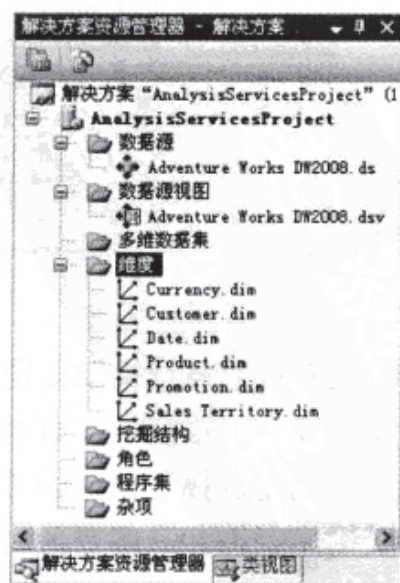


图 24-14

右击 Date 节点并选择“属性”。在“属性”窗格中往下滚动到“基本”部分，并注意到有一个 Type 条目。需要把这个条目修改为 Time，如图 24-16 所示。

当所有这些都创建好之后就可以构建立方体了。只要右击该项目并选择“部署”(也可以限制要构建的项)即可，如图 24-17 所示。

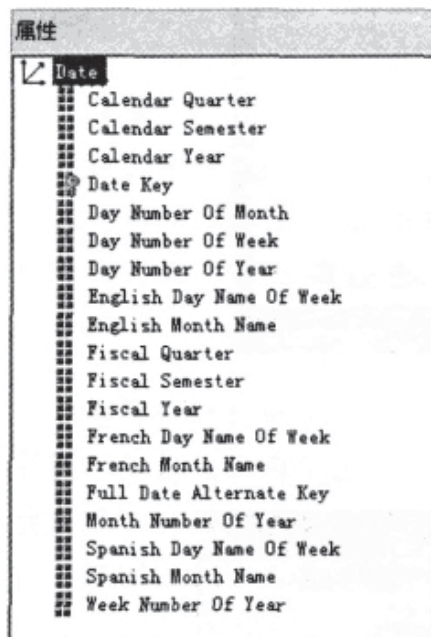


图 24-15

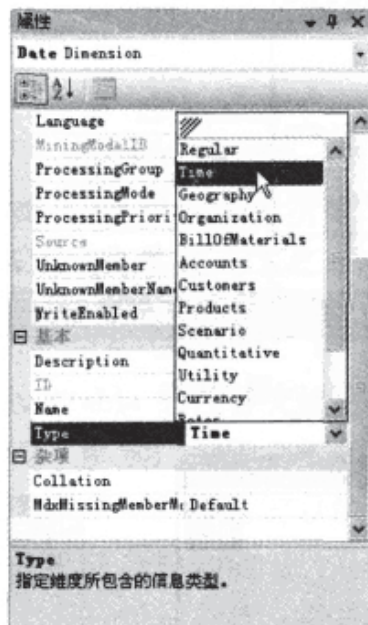


图 24-16

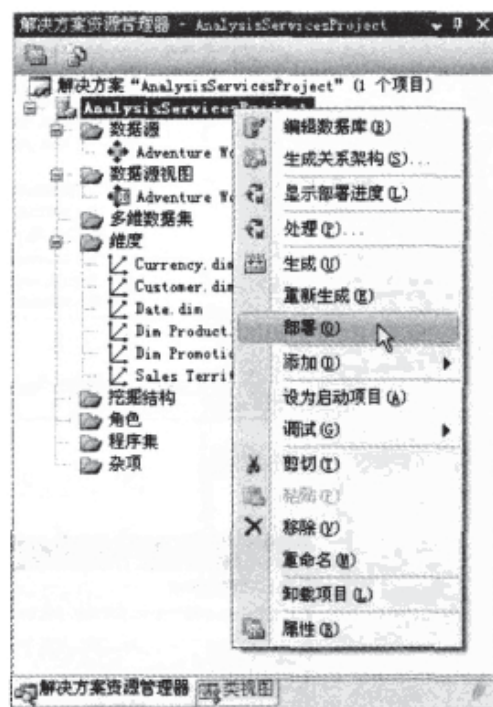


图 24-17

这样立方体就被完全实现了，现在要做的是快速看一看立方体到底能给我们带来什么。

24.6 访问立方体

假设已经遵循了前面示例中的步骤创建了立方体，现在可以真正使用立方体了。使用立方体存在几种方式：

- Microsoft Excel(稍后就会看到，连接之后将会自动得到一个数据透视表)。
- 直接连接并使用 Multi-Dimensions Expressions 或 MDX(分析服务在 T-SQL 中的等价物)来查询。
- 以分析为中心的其他工具，如 Performance Point Server。

作为一个速成例子，这里将使用 Excel 2007 中的数据透视表来连接刚刚创建的立方体。Excel 提供了大量的功能用于询问“假设(What if)?”问题，并且把数据透视表和数据透视图与分析服务立方体集成起来是相当容易的。

下面检验一下 Excel 的这些功能，启动 Excel 2007 并定位到数据带上，如图 24-18 所示。注意这里单击了“来自其他源”选项卡并选择了“来自分析服务”选项(这是内置的，不需要任何特殊的配置!)。接着会弹出一个“数据连接向导”对话框，它与读者在本书中见过的很多其他连接对话框非常相似。输入服务器的名称(如果立方体与运行 Excel 的计算机是同一台计算机，那么只需要输入(local)即可)，并单击“下一步”进入“选择数据库和表”对话框，如图 24-19 所示。现在继续并单击“完成”按钮。这样就会弹出“导入数据”对话框，如图 24-20 所示。这个对话框允许指定数据在电子表格中的存放位置并确认对数据所采取的操作(本例中将会创建一张数据透视表)。继续向前并单击“确定”以接受默认设置。

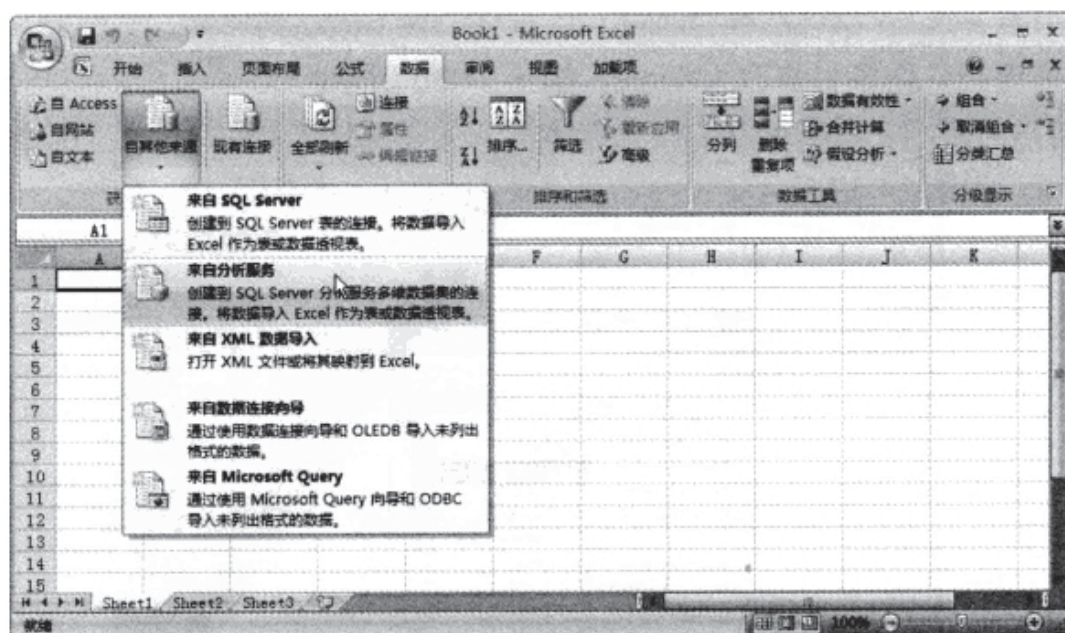


图 24-18

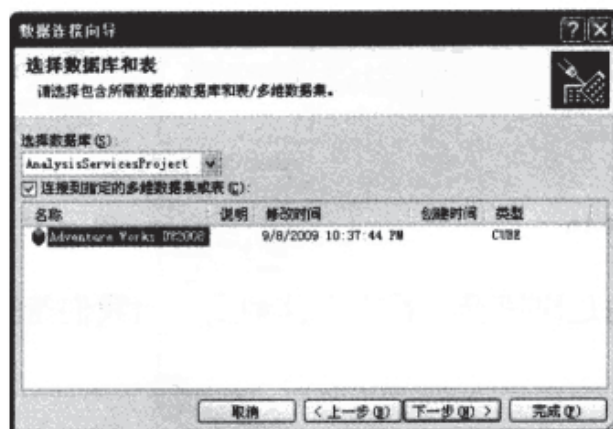


图 24-19

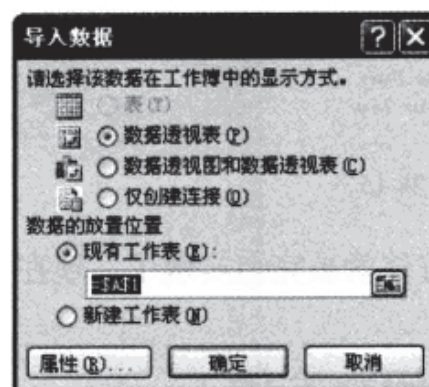


图 24-20

如果读者以前没有接触过数据透视表，那么可能会认为这张电子表格看起来有点虎头蛇尾。毕竟，里面没有数字，也没有真正的报表。但是，表面现象是具有欺骗性的。在工作簿右边的窗格中可以发现 Excel 中数据透视表的功能的秘密，如图 24-21 所示。

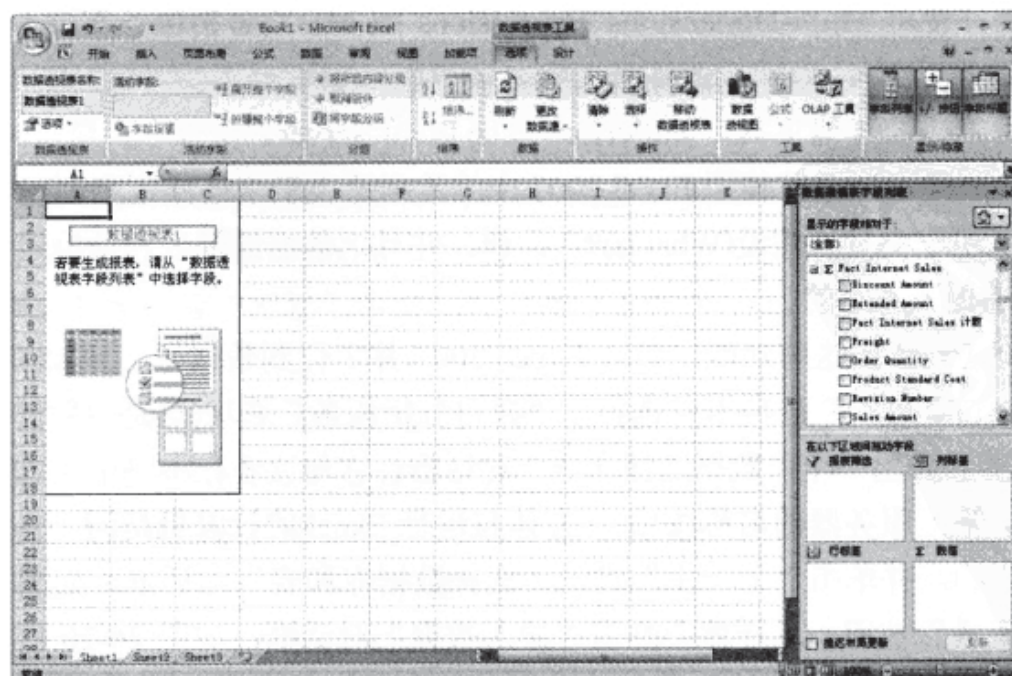


图 24-21

一开始它看起来似乎不包含报表,但是模板使得操作希望放在报表中的信息变得容易了,更重要的是可以使用它来研究结果。为了检验一下这个功能,下面稍微对数据进行一些操作。读者可以把感兴趣的字段从“数据透视表字段列表”中拖动到下面列出的区域中(如图 24-22 所示)。在做这个操作的时候,注意在每个框中删除一个字段对主 PivotTable 区域造成的影响。

注意:

在主电子表格区域中进行单击的时候要小心一点。如果在 PivotTable 区域之外单击了,那么所有的 PivotTable 字段都会消失。如果发生了这种事情,那么只需要在 PivotTable 区域内单击一下之后那些字段将会重新出现。

笔者打算把详细介绍什么是 PivotTable 的任务交给专门介绍 Excel 的书籍,但是读者有望通过这里的介绍了解到,使用分析服务立方体来研究数据是很容易的。记住本例中介绍的只是一种比较容易的连接到数据的方式。还可以使用 MDX 来向立方体发出复杂的查询。这种查询可以对多个维度进行比较并提供了特殊的功能,如对每年的结果进行逐年比较。还有,来自立方体的数据是经过高度优化以专门用于此类比较的。

24.7 小结

实际上,本章中介绍的内容并不能让读者成为数据仓库、分析服务或商业智能方面的专家。相反,这里只是希望读者对创建立方体需要做哪些工作有一个概念,并且大概了解一下使用立方体能做什么。需要强调的是本章,只介绍了该产品的一些表面知识。分析服务本身就需要用一本书来讲述。希望本章的内容能够让读者对分析服务有足够的了解,以至于能够判断出是否需要深入学习这方面的知识。

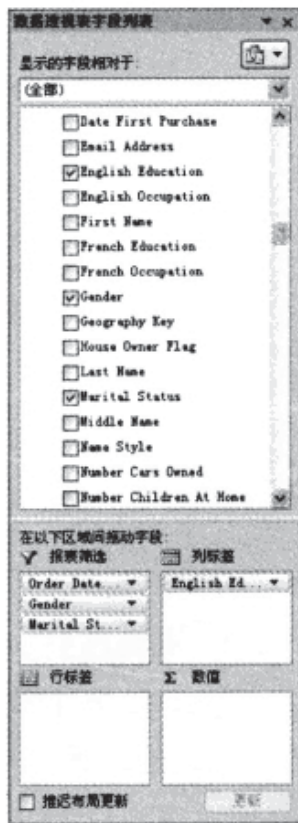


图 24-22



第25章

保证良好的连接性

如果 SQL Server 不允许程序与其进行连接,那么这跟没有 SQL Server 几乎没什么两样。当然,可以登录到 Management Studio 中,然后编写查询,但是现实是大多数用户实际上并不会直接看到数据库,他们只会使用已编写好的某些系统中的输入和报表屏幕(好,现在是一个多人联机世界,并且还有很多其他大型应用程序,而它们也可能被部署在一些高度可伸缩的系统上,但是大多数读者都不会碰到这些情况)。

有了这些概念之后,可能就需要确定应用程序如何与数据库交互了。有大量的书籍是直接介绍这个主题的(并且,除了基本的连接之外,实际上它本身也是一个巨大的主题),因此笔者不打算讨论每种语言中每个访问模型的所有优点。相反,这里将会介绍一些基本概念以及与性能、内存使用有关的一些基本问题和一般的最佳实践。与本书中介绍的其他一些大型主题一样,这里的想法是让读者能够快速掌握一些基础知识,并且了解完成任务需要做哪些事情以及应该问哪几类问题。

好,开玩笑的说,现在是时候讲讲坏消息了(但是这不是坏消息,稍后将解释其原因)。这个特殊的章节是“只通过 Web 发布的”章节,也就是说“你需要从 Web 上下载这个章节”。可以从 p2p.wrox.com 支持站点或者笔者的个人网站 www.professionalsql.com 上下载。为什么要这么做呢?好吧,这是由多个方面决定的。其中一个原因是笔者要接受本书时间上的约束。但是还有另一个原因——及时性。在过去十年或者稍微更长的一段时间里,连接是数据库中变化最大的领域之一。在本章的下载版中将会对此做一些讨论,在其整个历史发展中,各种各样的访问模型出现了,但后来又消失了(还有相当一部分仍然在使用)。在笔者编写本书的时候,.NET 世界大都使用 ADO.NET 和 LINQ。但是,即将要推出的是 Microsoft 的整个实体框架——谁知道在 Kilimanjaro(SQL Server 下一个版本的代码的名称)发布之前会向其中加入什么内容,因此需要出版另一本书来讲述这个主题。通过 Web 来发布使得当变更足够多以至于无法保证本章内容的正确性时,可以按更贴近实际的方式及时更新本章的内容(尽管仍然需要进行编辑,但是已经不需要进行排版或处理页码了)。

下载完本章之后读者将会发现如下信息:

- 过去和现在的各种数据访问对象模型(一点历史)。
- 数据访问的一些基本的最佳实践。
- 一些简短的使用 .NET 连接数据库的例子。



系统函数

SQL Server 包含了许多系统函数以及较典型的函数。其中的一些经常被使用，而且对于初学者来说，其用法相当直观。其他的函数则很少使用，用户对其用法也不是很清楚。

本附录将对这些函数的用法作简要的介绍。

提示：

在先前的版本中，经常会把许多系统函数称为“全局变量”。这是一个误称，Microsoft 在后面几个版本中试图对它进行纠正——修改文档，使用更恰当的名称“系统函数”来指代它。这里只要记住原来那个术语，以防有人提及全局变量时而不知所措。

SQL Server 2008 中可用的 T-SQL 函数分为以下 14 种：

- 遗留的系统函数
- 聚合函数
- 配置函数
- 加密函数
- 游标函数
- 日期和时间函数
- 数学函数
- 元数据函数
- 排名函数
- 行集函数
- 安全函数
- 字符串函数
- 系统函数
- 文本和图像函数

提示：

另外，还有 OVER 运算符，尽管是作为排名工具，也可用于其他 T-SQL 函数(特别是聚合函数)。虽然我只准备将其作为排名函数的一部分讨论，但在本附录中的其他地方也可看到相关内容。

A.1 遗留的系统函数(即全局变量)

A.1.1 @@CONNECTIONS

它返回自上次启动 SQL Server 以来尝试的连接数。

这是自最近一次 SQL Server 启动以来试图连接的次数总和。需要记住的是，这里谈到的是尝试的连接，而不是真正的连接，而且这里谈论的是连接，而不是用户。

无论连接成功与否，每次尝试连接都会增加这个计数器的值。这里要理解的是，尝试连接必须连接到服务器。如果连接因为 NetLib 不同或一些其他网络问题而失败，那么 SQL Server 不会知道增加计数器的值。它只会在服务器知道有尝试连接时才会计数。而尝试成功与否并没有关系。

同样需要理解这里谈论的是连接，而不是尝试登录。根据应用程序的情况，可能会创建到服务器的多个连接，但是可能只会向用户询问一次信息。的确，甚至是查询分析器(Query Analyzer)也会这么做。当单击并出现一个新窗口时，它会自动根据相同的登录信息创建另一个连接。

提示：

与其他许多系统函数一样，系统存储过程 `sp_monitor` 通常可以为其较好地提供服务。通过一个命令，该存储过程提供了包括连接数、CPU 忙闲程度以及 SQL Server 的写入总次数等信息。如果要获取基本信息，那么 `sp_monitor` 可能更合适——如果需要的是可操纵的离散数据，那么最好使用 `@@CONNECTIONS`。

A.1.2 @@CPU_BUSY

返回自上次启动 SQL Server 以来 CPU 的活跃工作时间，该时间以毫秒计数。这个数字基于系统计时器进行分析，这可能会有所不同，所以会在精度上有区别。

这是另一个以“自服务器启动以来”为标准的函数。这意味着不能总是指望该数字在应用程序运行时增长。根据这个数字了解 SQL Server 的 CPU 占用率是可能的。但是实际上，如果要知道该数据，更应该使用性能监视器。如果从扩大知识面的角度讲，这的确有些作用。在大多数的应用程序中，这个函数却并不实用。

A.1.3 @@IDLE

返回 SQL Server 自最后一次启动之后闲置的时间(基于系统计时器进行分析)，该时间以毫秒计数。

可以把它看作 `@@CPU_BUSY` 的相反情况。从本质上讲，它说明了 SQL Server 闲置的总时间。该值没有实际编程价值。

A.1.4 @@IO_BUSY

返回 SQL Server 自上次启动后用于执行输入和输出操作的时间(基于系统计时器进行分析)，该时间以毫秒计数。每次 SQL Server 启动时，都会重置该值。

该值并不是很难去理解，但没有实际编程价值。

A.1.5 @@PACK_RECEIVED 和 @@PACK_SENT

它们各自返回 SQL Server 自上次启动后由 SQL Server 从网络中读取或写入网络的输入数据包数目。

它们主要用作网络故障排除工具。

A.1.6 @@PACKET_ERRORS

返回自 SQL Server 上次启动后，在 SQL Server 连接上发生的网络数据包出错的错误数。它主要用作网络故障排除工具。

A.1.9 @@TIMETICKS

返回一刻度的微秒数。这个值因计算机而有所不同，而且也是没有实际编程价值。

A.1.10 @@TOTAL_ERRORS

返回自 SQL Server 最后一次启动后所碰到的磁盘读/写错误数。

不要将它与运行时错误或@@ERROR 相混淆。这是关于物理 I/O 问题的。它没有实际编程价值。其主要用途类似于系统诊断脚本。一般来说，应该使用 Windows 可靠性和性能监视器来取代它。

A.1.11 @@TOTAL_READ 和 @@TOTAL_WRITE

它们分别返回 SQL Server 启动后磁盘读取/写入的总量。

这两个名称有些误导，因为这里没有包含对缓存的任何读取；它们只是物理 I/O。

A.1.12 @@TRANCOUNT

返回活动的事务数——本质上是当前连接的事务嵌套级别。

当处理事务时，这是一个大问题。我们可能并不喜欢嵌套的事务，但是也会有难以避免的情况。所以，知道现在正处于事务嵌套的哪个位置会很重要(例如，也许会有这样的逻辑，只有未启动事务时，才开始处理一个事务。)

如果不在事务中，那么@@TRANCOUNT 为 0。这里看一个简单的示例：

```
SELECT @@TRANCOUNT As TransactionNestLevel      --This will be zero at this
point

BEGIN TRAN
SELECT @@TRANCOUNT As TransactionNestLevel      --This will be one at this
point
  BEGIN TRAN
    SELECT @@TRANCOUNT As TransactionNestLevel  --This will be two at this
point
  COMMIT TRAN
SELECT @@TRANCOUNT As TransactionNestLevel      --This will be back to one
--at this point

ROLLBACK TRAN
SELECT @@TRANCOUNT As TransactionNestLevel      --This will be back to zero
--at this point
```

在本例中，请注意如果有一个 COMMIT 作为最后的语句，那么最后的@@TRANCOUNT 将变为 0。

A.2 聚合函数

聚合函数应用于记录集而非单个记录。多个记录中的信息将使用一种特殊的方法处理，并在单个记录结果中显示。聚合函数经常与 GROUP BY 子句同时使用。

聚合函数包括如下：

- AVG
- CHECKSUM
- CHECKSUM_AGG
- COUNT
- COUNT_BIG
- GROUPING
- MAX
- MIN
- STDEV
- STDEVP
- SUM
- VAR
- VARP

在大多数聚合函数中，可以使用 ALL 或 DISTINCT 关键字。ALL 参数是默认值，而且会将函数应用于 expression 中的所有值，即使值会多次出现，也同样如此。DISTINCT 参数意味着一个值只能包含在函数中一次，即使它多次出现在该函数中，也同样如此。

聚合函数不能嵌套。expression 不能是子查询。

A.2.1 AVG

AVG 返回 expression 中值的平均值。其语法如下：

```
AVG([ALL | DISTINCT] <expression>)
```

expression 必须包含数值型值。它会忽略 NULL 值。该函数支持本附录“排名函数”一节中介绍的 OVER 运算符。

A.2.2 CHECKSUM

这是通常用于检测数据中的更改或一致性的基本哈希函数。这个特殊的函数接受表达式或*号(这表示要包括所有连接表中的所有列)作为参数。其基本语法如下：

```
CHECKSUM(<expression>, [ . . . n ] | * )
```

注意，表达式的顺序(或使用*号时的连接顺序)将影响检验和值，例如下面这个语句：

```
CHECKSUM(SalesOrderID, OrderDate)
```

和下列语句的结果不同：

```
CHECKSUM(OrderDate, SalesOrderID )
```

该函数不与 OVER 运算符兼容。

A.2.3 CHECKSUM_AGG

和 CHECKSUM 一样，这也是通常用于检测数据中的更改或一致性的基本哈希函数。两者的

主要区别是 CHECKSUM 面向行，而 CHECKSUM_AGG 面向列。其基本语法如下：

```
CHECKSUM_AGG( [ALL | DISTINCT] <expression>)
```

表达式值可以是任何值，包括列的串联(只是记得在必要时进行强制转换)；不过，记住表达式顺序会对结果有影响，因此串联 Col1+Col2 与串联 Col2+Col1 是不一样的。

A.2.4 COUNT

COUNT 返回 expression 中的项的数量。返回的数据类型是 int 类型。其语法如下：

```
COUNT  
(  
    [ALL | DISTINCT] <expression> | *  
)
```

expression 不能是 uniqueidentifier、text、image 或 ntext 数据类型。* 参数返回表中行的数量；它不会消除重复值或者 NULL 值。

该函数支持本附录“排名函数”一节中介绍的 OVER 运算符。

A.2.5 COUNT_BIG

COUNT_BIG 返回组中的项的数量。这和前面描述的 COUNT 函数很相似，不同的是它总是返回 bigint 数据类型的值。其语法如下：

```
COUNT_BIG  
(  
    [ALL | DISTINCT] <expression> | *  
)
```

和 COUNT 一样，该函数支持本附录“排名函数”一节中介绍的 OVER 运算符。

A.2.6 GROUPING

GROUPING 给 SELECT 语句的输出添加一个额外的列。GROUPING 函数与 CUBE 或 ROLLUP 结合使用来区分普通的 NULL 值以及那些作为 CUBE 和 ROLLUP 操作的结果而添加的值。其语法如下：

```
GROUPING (<column_name>)
```

GROUPING 只用在 SELECT 列表中。它的参数是用在 GROUP BY 子句中的列，而且将对这个参数检查 NULL 值。

该函数支持本附录“排名函数”一节中介绍的 OVER 运算符。

A.2.7 MAX

MAX 函数返回 expression 中的最大值。其语法如下：

```
MAX([ALL | DISTINCT] <expression>)
```


MAX 忽略任何 NULL 值。

该函数支持本附录“排名函数”一节中介绍的 OVER 运算符。

A.2.8 MIN

MIN 函数返回 *expression* 中的最小值。其语法如下：

```
MIN([ALL | DISTINCT] <expression>)
```

MIN 忽略任何 NULL 值。

该函数支持本附录“排名函数”一节中介绍的 OVER 运算符。

A.2.9 STDEV

STDEV 函数返回 *expression* 中所有值的标准偏差(deviation)。其语法如下：

```
STDEV(<expression>)
```

STDEV 忽略所有 NULL 值。

该函数支持本附录“排名函数”一节中介绍的 OVER 运算符。

A.2.10 STDEVP

STDEVP 函数返回 *expression* 中所有填充值的标准偏差。其语法如下：

```
STDEVP(<expression>)
```

STDEVP 忽略 NULL 值，并支持本附录“排名函数”一节中介绍的 OVER 运算符。

A.2.11 SUM

SUM 函数返回 *expression* 中的所有值的总和。其语法如下：

```
SUM([ALL | DISTINCT] <expression>)
```

SUM 忽略 NULL 值。该函数支持本附录“排名函数”一节中介绍的 OVER 运算符。

A.2.12 VAR

VAR 函数返回 *expression* 中所有值的方差。其语法如下：

```
VAR(<expression>)
```

VAR 忽略 NULL 值。该函数支持本附录“排名函数”一节中介绍的 OVER 运算符。

A.2.13 VARP

VARP 函数返回 *expression* 中所有填充值的方差。其语法如下：

VARP(<expression>)

VARP 忽略 NULL 值。该函数支持本附录“排名函数”一节中介绍的 OVER 运算符。

A.3 配置函数

配置函数说明了为当前服务器或数据库设置的选项(在适当的时候)。

A.3.1 @@DATEFIRST

返回系统所认为的每周第一天所对应的数字值。

在美国, 这个默认值为 7, 即 Sunday。值的对应关系如下:

- 1——Monday(对于世界上大多数国家来说是第一天)
- 2——Tuesday
- 3——Wednesday
- 4——Thursday
- 5——Friday
- 6——Saturday
- 7——Sunday

当处理本地化问题时, 这很有用, 这样可以恰当地设计任何日历或者有关星期几的信息。

提示:

可以使用 SET DATEFIRST 函数来更改这个设置。

A.3.2 @@DBTS

返回当前数据库最后使用的时间戳。

这个函数初看上去和 @@IDENTITY 的功能类似, 因为它提供了由系统设置的最终值(这次是最后的时间戳而不是最后的标识值)。这里需要注意的内容包括:

- 这个值会根据数据库的任何修改而变化, 不只是根据使用的表
- 任何对数据库中的时间戳进行的修改都会反映出来, 不只是针对当前连接

因为不能指望该值是使用的最终值(因为有些人会修改该值), 所以这个值的实用价值不高。

A.3.3 @@LANGID 和 @@LANGUAGE

它们分别返回当前所使用语言的 ID 和名称。

这些值可以很方便地指出产品是否本地化安装, 如果是这样, 这个语言就是默认值。

要列出 SQL Server 当前支持的所有语言, 可以使用系统存储过程 sp_helplanguage。

A.3.4 @@LOCK_TIMEOUT

返回当前在系统超时前等待阻塞资源的时间, 该时间以毫秒计数。

如果资源(页、行、表或任何内容)被阻塞,那么当前进程将会停下来并等待该阻塞清除。这个值决定了在该语句被取消前当前进程需要等待的时间。

默认的等待时间为 0(这意味着不等待),除非有人在系统级别修改了该值(使用 `sp_configure`)。无论系统默认值如何设置,该全局变量的值都为-1,除非使用 `SET LOCK_TIMEOUT` 为当前连接手动设置该值。

A.3.5 @@MAX_CONNECTIONS

返回 SQL Server 上允许的并发用户连接的最大数量。

不要误认为这个和 Management Console 中的 Maximum Connection 属性一样。`@@MAX_CONNECTIONS` 是基于许可的,如果选择 per seat 许可,那么它的值会很大。

提示:

实际允许的用户连接数也取决于使用的 SQL Server 版本以及应用程序和硬件的限制。

A.3.6 @@MAX_PRECISION

返回当前为 decimal 和 numeric 数据类型所设置的精度级别。

默认的是 38 位,当启动 SQL Server 时也可以使用 `/p` 选项来修改这个值。可以在从命令行启动 SQL Server 添加 `/p`,或者在 Windows Services applet 中,将它添加到 MSSQL Server 服务的 Startup 参数中。

A.3.7 @@NESTLEVEL

返回存储过程的当前嵌套级别。

第一个运行的存储过程的 `nTLEVEL` 是 0。如果这个存储过程调用另一个存储过程,那么第二个存储过程就是嵌套在第一个里面的(`nTLEVEL` 就增长为 1)。同样的,第二个存储过程调用第三个存储过程,直到最大的 32 层深度。如果超过了 32 层的深度,不仅事务会终止,而且你还应该重新审视一下该应用程序的设计。

A.3.8 @@OPTIONS

返回使用 SET 命令所应用的选项的信息。

尽管只返回了一个值,但是可以设置很多选项,SQL Server 使用二进制标志来表示设置了哪个值。为了测试是否设置了你所感兴趣的选项,必须把该选项值与按位运算符一起使用。例如:

```
IF (@@OPTIONS & 2)
```

如果该值为 True,那么就知道已经为当前连接启用了 `IMPLICIT_TRANSACTIONS`。这些值如表 A-1 所示。

表 A-1

位	SET 选项	描 述
1	DISABLE_DEF_CNST_CHK	间歇与延迟约束检测
2	IMPLICIT_TRANSACTIONS	当执行一条语句时隐式地启动一个事务
4	CUSOR_CLOSE_ON_COMMIT	当执行完 COMMIT 操作后控制游标的行为
8	ANSI_WARNINGS	警告聚合中的截断和 NULL
16	ANSI_PADDING	控制定长变量的填充
32	ANSI_NULLS	确定使用等号运算符时 NULL 的处理
64	ARITHABORT	当在查询执行期间发生溢出或除零错误时终止查询
128	ARITHIGNORE	当在查询时发生溢出或除零错误时返回 NULL
256	QUOTED_IDENTIFIER	在计算表达式时区分单引号和双引号
512	NOCOUNT	关闭在每条语句执行后报告有多少行被影响的提示信息
1024	ANSI_NULL_DFLT_ON	将会话行为修改为使用 ANSI 兼容的可空性。对于使用新表创建的列或添加到旧表中的列，如果没有显式设置空值选项，都定义为允许使用空值。它与 ANSI_NULL_DFLT_OFF 互斥
2048	ANSI_NULL_DFLT_OFF	将会话行为修改为不使用 ANSI 兼容的可空性。没有显式地指明可空的新列定义为不允许使用空值。它与 ANSI_NULL_DFLT_ON 互斥
4096	CONCAT_NULL_YIELDS_NULL	当串联 NULL 和字符串时返回 NULL
8192	NUMERIC_ROUNDABORT	当在表达式中发生精度丢失时产生错误

A.3.9 @@REMSERVER

返回调用存储过程的服务器的值(和登录记录中的一样)。

只在存储过程中使用。如果想让存储过程根据调用它的远程服务器(通常指地理位置)而表现不同时，它很有用。

A.3.10 @@SERVERNAME

返回运行脚本的本地服务器的名称。

如果安装了多个 SQL Server 实例(例如 Web 主机服务，它为每个客户端使用单独的 SQL Server 安装)，那么如果本地服务器的名称自安装后没有改变过，则 @@SERVERNAME 将会返回表 A-2 所示的本地服务器的名称信息。

表 A-2

实 例	服务器信息
默认实例	<servername>
命名实例	<servername\instancename>
虚拟服务器-默认实例	<virtualservername>
虚拟服务器-命名实例	<virtualservername\instancename>

A.3.11 @@SERVICENAME

返回 SQL Server 正在其下运行的注册表键的名称。

只在 Windows 2000/2003/XP 下返回名称，并且在这些系统下总是返回 MSSQLService，除非正在该注册表中进行操作。

A.3.12 @@SPID

返回当前用户进程的服务器进程 ID(SPID)。

这和运行 sp_who 所看到的进程 ID 一样。这样做的好处是，可以知道当前连接的 SPID，DBA 可以使用它来监控，并在必要时终止该任务。

A.3.13 @@TEXTSIZE

返回 SET 语句的 TEXTSIZE 选项的当前值，它指定 SELECT 语句在处理文本或图像数据时返回的最大长度，以字节为单位。

默认值是 4096 字节(4KB)。可以使用 SET TEXTSIZE 语句来修改这个值。

A.3.14 @@VERSION

返回 SQL Server 的当前版本、处理器类型和 OS 体系结构。

例如，如果在旧式的 SQL Server 2005 上运行以下语句：

```
SELECT @@VERSION
```

得到如下结果：

```
-----  
Microsoft SQL Server 2008 (RTM) - 10.0.1600.22 (X64)  
Jul 9 2008 14:17:44  
Copyright (c) 1988 - 2008 Microsoft Corporation  
Developer Edition (64-bit) on Windows NT 6.0 <X64> (Build 6001: Service Pack 1)  
  
(1 row(s) affected)
```

遗憾的是，这不会以任何结构化的字段排列形式返回信息，所以如果想要使用它来测试特定信息，那么必须对它进行分析。

考虑使用 xp_msver 系统存储过程来取代它，通过该存储过程可以更容易地从结果中检索特定的信息。

A.4 加密函数

加密函数用于支持加密、解密、数字签名和数字签名验证。其中一些是 SQL Server 2008 新提供的，还有一些是随 SQL Server 2005 提供的。可注意到，从一般用途来看，大多数函数都有对应的副本，但它们的区别在于一个支持对称密钥，而副本(通常名称中带有“Asym”)支持非对称

密钥。

你可能会问“为什么需要这些函数”。答案是多样化的。简单地说，如果想在传输过程中保护发送或接收的数据，需要使用它们。例如，由于 SQL Server 支持 HTTP 端点，并在那里驻留自己的 Web 服务，你可能希望用 Web 服务的客户端接收或返回加密信息。更可能的是你选择在数据库中加密数据，现在为了使用它需要解密。

A.4.1 AsymKey_ID

给定非对称密钥的名称，该函数将从数据库返回与相应 ID 对应的 int 值。其语法很简单：

```
AsymKey_ID('<Asymmetric Key Name>')
```

要使用该函数，必须有对该密钥的权限。

A.4.2 Cert_ID

类似于 AsymKey_ID，它返回证书名称的 ID。其语法很简单：

```
Cert_ID('<Certificate Name>')
```

要使用该函数，必须有对该证书的权限。

A.4.3 CertProperty

返回给定证书(用证书的 ID 指定)的不同属性。有效属性包括开始生效日期、失效日期、证书颁发者的名称、序列号、安全 ID(SID，也可以按字符串形式返回)、证书的主题(证明谁或什么)。其语法如下：

```
CertProperty ( Cert_ID ,  
    'Expiry_Date'|'Start_Date'|'Issuer_Name'|'Cert_Serial_Number'|'Subject'  
    '|SID'|'String_SID' )
```

返回的数据类型随指定的属性而异(可能是 datetime、nvarchar 或 varbinary)。

A.4.4 DecryptByAsmKey

从其名称中可推测出，它利用非对称密钥解密数据。它要求输入密钥(通过 ID)、加密的数据(字符串字面量或字符串可压缩变量)、用于加密数据库中的非对称密钥的密码。其语法很简单：

```
DecryptByAsmKey(<Asymmetric Key ID>, {'<encrypted string>'|<string variable>}  
    [, '<password>'])
```

A.4.5 DecryptByCert

这个函数基本上与 DecryptByAsmKey 相同，除了它接受的是证书而不是非对称密钥。与 DecryptByAsmKey 一样，它用密钥解密一组数据。它要求输入证书(通过 ID)、加密的数据(字符串

字面量或字符串可压缩变量)、用于加密证书私钥的密码(如果使用了密码)。其语法与 `DecryptByAsymKey` 的类似:

```
DecryptByCert(<Certificate ID>, {'<encrypted string>'|<string variable>}
[, '<password>'])
```

同样, 加密证书私钥时使用的密码在解密时也是需要的。

A.4.6 DecryptByKey

和前两个函数一样, 这个函数使用密钥解密数据。不同之处是它不仅要使用对称密钥(而不是其他类型的密钥), 而要求密钥是“打开的”(使用 `OPEN SYMMETRIC KEY` 命令)。除此之外, 其使用方式是类似的, 要求用加密的数据(字符串字面量或字符串可压缩变量)作为参数, 在这里, 可以选择散列密钥作为验证器。

```
DecryptByKey({'<encrypted string>'|<string variable>},
[<add authenticator value>, '<authentication hash>'|<string variable>])
```

注意, 如果提供加验证器的值(用 `int` 形式), 那该值必须与在加密字符串时提供的值一致, 另外还必须提供一个与加密时提供的散列值匹配的值。

A.4.7 DecryptByPassPhrase

正如其名称所示, 这个函数对用通行短语而非正式密钥加密的数据解密。除了接受通行短语参数而非打开的密钥外, 该函数的工作方式与 `DecryptByKey` 十分类似:

```
DecryptByPassPhrase({'<passphrase>'|<string variable>},
{'<encrypted string>'|<string variable>},
[<add authenticator value>, '<authentication hash>'|<string variable>])
```

和 `DecryptByKey` 一样, 如果提供加验证器的值(用 `int` 形式), 那该值必须与在加密字符串时提供的值一致, 另外, 还必须提供一个与加密时提供的散列值匹配的值。

A.4.8 EncryptByAsmKey

使用非对称密钥加密数据。它要求输入密钥(通过 ID)和要加密的数据(字符串字面量或字符串变量)。其语法很简单:

```
EncryptByAsymKey(<Asymmetric Key ID>, {'<string to encrypt>'|<string variable>})
```

A.4.9 EncryptByCert

这个函数与 `EncryptByAsmKey` 基本上相同。不过, 它接受证书而不是非对称密钥。和 `EncryptByAsmKey` 一样, 这个函数使用提供的密钥加密一组数据。它要求输入证书(通过 ID)和要加密的数据(字符串字面量或字符串可压缩变量)。其语法与 `EncryptByAsymKey` 类似:

```
EncryptByCert(<Certificate ID>, {'<string to be encrypted>'|<string
variable>})
```

A.4.10 EncryptByKey

这个函数不仅要求接受对称密钥(而不是非对称密钥),还要求密钥是“打开的(使用 OPEN SYMMETRIC KEY 命令)”和可用于引用该密钥的 GUID。除此之外,其使用方式是类似的,用要加密数据(字符串字面量或字符串可压缩变量)作为参数,在这里,可以选择散列密钥作为验证器。

```
EncryptByKey({<Key GUID>, '<string to be encrypted>'|<string variable>},
[<add authenticator value>, '<authentication hash>'|<string variable>])
```

注意,如果提供加验证器的值(用 int 形式),那么在解密此字符串时必须提供此值,还必须提供散列值。

A.4.11 EncryptByPassPhrase

这个函数使用通行短语(而非密钥)加密数据。除了接受通行短语(而非打开的密钥)作为参数外,EncryptByPassPhrase 的工作方式与 EncryptByKey 类似:

```
EncryptByPassPhrase({'<passphrase>'|<string variable>},
{'<string to be encrypted>'|<string variable>},
[<add authenticator value>, '<authentication hash>'|<string variable>])
```

和 EncryptByKey 一样,如果提供加验证器的值(用 int 形式),那么在解密此字符串时必须提供此值,还必须提供散列值。

A.4.12 Key_GUID

返回当前数据库中指定对称密钥的 GUID。

```
Key_GUID('<Key Name>')
```

A.4.13 Key_ID

返回当前数据库中指定对称密钥的 GUID。

```
Key_ID('<Key Name>')
```

A.4.14 SignByAsymKey

为指定的纯文本值添加非对称密钥签名。

```
SignByAsymKey(<Asymmetric Key ID>, <string variable> [, '<password>'])
```

A.4.15 SignByCert

提供指定证书和纯文本值,返回包含结果签名的 varbinary 值(最大 8000 字节)。

```
SignByCert(<Certificate ID>, <string variable> [, '<password>'])
```


A.4.16 VerifySignedByAsymKey

针对指定的非对称密钥和纯文本值，返回表明签名验证成功与否的 int 值：

```
VerifySignedByAsymKey(<Asymmetric Key ID>, <plain text> , <signature>)
```

A.4.17 VerifySignedByCert

针对指定的证书和纯文本，返回表明签名验证成功与否的 int 值：

```
VerifySignedByCert(<Certificate ID>, <signed plain text> , <signature>)
```

A.5 游标函数

这些函数提供了有关游标状态或性质的各种信息。

A.5.1 @@CURSOR_ROWS

返回当前连接上打开的最后一个游标集中当前行的数目。注意，这针对的是游标，而不是临时表。

记住，每次打开一个新游标时，将会重置该值。如果需要一次打开多个游标，并且想要知道第一个游标中的行数，则需要在打开后续的游标前，将该值移到一个临时保存变量中。

当处理游标时，可以使用该值建立一个计数器来控制 WHILE 循环，但是我强烈建议不要这么做；@@CURSOR_ROWS 中包含的值会随游标类型和 SQL Server 是否异步填充该游标而变化。使用 @@FETCH_STATUS 更可靠，至少易于使用。

如果返回的值是比-1 大的负数，则必须使用一个异步的游标，并且该负值就是到现在为止在游标中创建的记录数。如果该值为-1，那么该游标是动态的游标，因为行的数目在不断地变化。返回值为 0 则表明没有打开任何游标，或者最后一个打开的游标关闭。任何正数值表示了该游标中的行数。

提示：

要创建异步游标，可设置 sp_configure cursor threshold 为比 0 大的值。这样当游标超过这个设置的值时，会返回该游标，而剩下来的记录会异步地放入该游标中。

A.5.2 @@FETCH_STATUS

返回最后一个游标 FETCH 操作的状态的指示器。

如果使用游标，那么就会使用 @@FETCH_STATUS。通过这个值可知尝试导航至游标中的一个记录成功还是失败。它会根据 SQL Server 是否成功执行最后一个 FETCH 操作而返回一个常量，并且如果失败，还会说明具体原因。这些常量是：

- 0——成功
- -1——失败，通常是因为超过了游标集的开始和结束位置。

- -2——失败，提取的行没有找到，通常是因为在创建游标集并导航到当前行期间，该行已被删除。这只会出现在可滚动的、非动态的游标中出现。

为了增强可读性，我在使用@@FETCH_STATUS 之前通常会设置一些常量。

例如：

```
DECLARE @NOTFOUND int
DECLARE @BEGINEND int

SELECT @NOTFOUND - 2
SELECT @BEGINEND - 1
```

然后可以在游标循环的 WHILE 语句的条件中使用这些常量，而不只是使用行的整数值。这会使代码的可读性更好。

A.5.3 CURSOR_STATUS

CURSOR_STATUS 函数允许存储过程的调用者确定该过程是否返回了游标和结果集。其语法如下：

```
CURSOR_STATUS
(
    {'<local>', '<cursor name>'}
    | {'<global>', '<cursor name>'}
    | {'<variable>', '<cursor variable>'}
)
```

local、global 和 variable 都指定了常量，这些常量表明了游标的源。Local 表示本地游标名，global 表示全局游标名，variable 表示本地变量。

如果使用 cursor name 形式，那么就有 4 种可能的返回值：

- 1——游标是打开的。如果游标是动态的，那么结果集有 0 或多行。如果游标不是动态的，那么它有 1 或多行。
- 0——游标的结果集是空的。
- -1——游标是关闭的。
- -3——cursor name 的游标不存在。

如果使用 cursor variable 形式，那么就会有 5 种可能的返回值：

- 1——游标是打开的。如果游标是动态的，那么结果集有 0 或多行。如果游标不是动态的，那么它有 1 或多行。
- 0——结果集为空。
- -1——游标是关闭的。
- -2——没有分配给 cursor variable 的游标。
- -3——名称为 cursor variable 的变量不存在，或者即使它存在，但没有给它分配游标。

A.6 日期和时间函数

SQL Server 2008 添加了一些新的日期和时间函数。除了处理时间戳数据(这实际上更多是面

向版本控制，而不是时钟或日历)外，日期和时间函数对 SQL Server 支持的任何日期和时间类型值执行操作。

在这些函数中，SQL Server 可以识别 11 种“日期部分”及它们的缩写，如表 A-3 所示。

表 A-3

日期部分	缩写
year	yy, yyyy
quarter	qq, q
month	mm, m
dayofyear	dy, y
day	dd, d
week	wk, ww
weekday	dw
hour	hh
minute	mi, n
second	ss, s
millisecond	ms

A.6.1 CURRENT_TIMESTAMP

CURRENT_TIMESTAMP 函数返回当前的日期和时间，返回值为 datetime 类型。此函数等价于 GETDATE()。其语法如下：

```
CURRENT_TIMESTAMP
```

A.6.2 DATEADD

DATEADD 函数向日期添加一段时间间隔并返回一个新的日期。其语法如下：

```
DATEADD(<datepart>, <number>, <date>)
```

datepart 参数指定了时间间隔的时间刻度(day、week、month 等)，可能是 SQL Server 所能识别的任何“日期部分”。number 参数是要添加到 date 中的“日期部分”数。

A.6.3 DATEDIFF

DATEDIFF 函数返回两个指定日期以指定时间单位(例如：小时、天数、周)计数的差值。其语法如下：

```
DATEDIFF(<datepart>, <startdate>, <enddate>)
```

datepart 参数可以是 SQL Server 能识别的任何日期部分，指定了时间单位。

A.6.4 DATENAME

DATENAME 函数返回一个字符串, 该字符串表示的是指定 date 的指定 datepart(例如, 1999、Thursday、July)的名称。其语法如下:

```
DATENAME(<datepart>, <date>)
```

A.6.5 DATEPART

DATEPART 函数返回一个整数, 该整数表示指定 date 的指定 datepart。其语法如下:

```
DATEPART(<datepart>, <date>)
```

DAY 函数等价于 DATEPART(dd,<date>); MONTH 等价于 DATEPART(mm,<date>); YEAR 等价于 DATEPART(yy,<date>)。

A.6.6 DAY

DAY 函数返回表示指定日期的哪一天的整数。其语法如下:

```
DAY(<date>)
```

DAY 函数等价于 DATEPART(dd,<date>)。

A.6.7 GETDATE

GETDATE 函数返回当前系统的日期和时间。其语法如下:

```
GETDATE()
```

A.6.8 GETUTCDATE

GETUTCDATE 函数返回当前 UTC(Universal Time Coordinate, 世界时间坐标)的时间。换句话说, 这会返回格林威治时间。这个值是通过从服务器获取本地时间、本地时区, 并计算 GMT 得来的。这包括了夏令时。用户自定义函数不能调用 GETUTCDATE。其语法如下:

```
GETUTCDATE()
```

A.6.9 ISDATE

ISDATE 函数用于确定输入表达式是否是有效日期。其语法如下:

```
ISDATE(<expression>)
```

A.6.10 MONTH

MONTH 函数返回表示指定日期的月份的整数。其语法如下:

```
MONTH(<date>)
```


MONTH 函数等价于 DATEPART(mm,<date>)。

A.6.11 SYSDATETIME

与更早出现的 GETDATE 函数一样, SYSDATETIME 返回当前系统日期和时间。不同之处有两个。第一, SYSDATETIME 返回值的精度较高。第二, 它返回更新的 datetime2 数据类型(支持更高的精度——这里精度为 7), 其语法如下所示:

```
SYSDATETIME()
```

A.6.12 SYSDATETIMEOFFSET

类似于 SYSDATETIME, 这个函数返回当前系统日期和时间。不过, 它返回的不是简单的 datetime2 数据类型, 而是新的 datetimeoffset 数据类型(精度为 7)的值, 因此提供了相对于世界时的偏移量信息。其语法如下:

```
SYSDATETIMEOFFSET()
```

A.6.13 SYSUTCDATETIME

与更早出现的 GETUTCDATE 函数类似, SYSDATETIME 返回当前的 UTC 日期和时间。不过, 它返回更新的 datetime2 数据类型(精度为 7)。其语法如下:

```
SYSUTCDATETIME()
```

A.6.14 SWITCHOFFSET

该函数接受两个参数——一个是 datetimeoffset()类型的输入值, 另一个是表示时间偏移量的新 offset 参数。其语法如下所示:

```
SWITCHOFFSET(<datetimeoffset data instance>, <newoffset>)
```

因此, 如果作一个快速测试:

```
CREATE TABLE TimeTest
(
    MyTime datetimeoffset
);

INSERT TimeTest
VALUES ('2008-12-31 6:00:00 -5:00');

SELECT SWITCHOFFSET(MyTime, '-08:00') AS Pacific
FROM TimeTest;

DROP TABLE TimeTest;
```

结果如下所示:

```
(1 row(s) affected)
```

```
Pacific
-----
2008-12-31 03:00:00.0000000 -08:00

(1 row(s) affected)
```

A.6.15 TODATETIMEOFFSET

接受一个指定的日期/时间信息，添加提供的时间偏移量，生成一个 `datetimeoffset` 数据类型的值。其语法如下：

```
TODATETIMEOFFSET(<data that resolves to datetime>, <time zone>)
```

例如：

```
DECLARE @OurDateTimeTest datetime;
SELECT @OurDateTimeTest - '2008-01-01 12:54';
SELECT TODATETIMEOFFSET(@OurDateTimeTest, '-07:00');
```

结果如下：

```
-----
1/1/2008 12:54:00 PM -07:00

(local)(sa): (1 row(s) affected)
```

A.6.16 YEAR

`YEAR` 函数返回表示指定日期中的年份的整数。其语法如下：

```
YEAR(<date>)
```

`YEAR` 函数等价于 `DATEPART(yy,<date>)`。

A.7 数学函数

数学函数执行计算的功能。它们分别是：

- | | |
|-----------|-----------|
| • ABS | • LOG |
| • ACOS | • LOG10 |
| • ASIN | • PI |
| • ATAN | • POWER |
| • ATN2 | • RADIANS |
| • CEILING | • RAND |
| • COS | • ROUND |
| • COT | • SIGN |

- DEGREES
- EXP
- FLOOR
- SIN
- SQRT
- SQUARE
- TAN

A.7.1 ABS

ABS 函数返回数值表达式(numeric expression)的绝对值(正值)。其语法如下:

```
ABS(<numeric expression>)
```

A.7.2 ACOS

ACOS 函数返回以弧度表示的角度值,该角度值的余弦为指定的表达式(换句话说,它返回的是 expression 的反余弦值)。其语法如下:

```
ACOS(<expression>)
```

这个值必须在-1 和 1 之间,必须为 float 数据类型。

A.7.3 ASIN

ASIN 函数返回以弧度表示的角度值,该角度值的正弦为指定的表达式(换句话说,它返回的是 expression 的反正弦值)。其语法如下:

```
ASIN(<expression>)
```

这个值必须在-1 和 1 之间,必须为 float 数据类型。

A.7.4 ATAN

ATAN 函数返回以弧度表示的角度值,该角度值的正切为指定的表达式(换句话说,它返回的是 expression 的反正切值)。其语法如下:

```
ATAN(<expression>)
```

expression 必须为 float 数据类型。

A.7.5 ATN2

ATN2 函数返回以弧度表示的角度值,该角度值的正切介于两个指定的表达式之间(换句话说,它返回的是两个 expression 的反正切值)。其语法如下:

```
ATN2(<expression1>, <expression2>)
```

expression1 和 expression2 都必须是 float 数据类型。

A.7.6 CEILING

CEILING 函数返回大于或等于指定表达式的最小整数。其语法如下：

```
CEILING(<expression>)
```

A.7.7 COS

COS 函数返回指定表达式中指定角度的余弦值。其语法如下：

```
COS(<expression>)
```

角度是以弧度为单位的，**expression** 必须为 float 数据类型。

A.7.8 COT

COT 函数返回指定表达式中指定角度的余切值。其语法如下：

```
COT(<expression>)
```

角度是以弧度为单位的，**expression** 必须为 float 数据类型。

A.7.9 DEGREES

DEGREES 函数接受以弧度为单位的角度(**expression**)，返回相应的以度数为单位的角度。其语法如下：

```
DEGREES(<expression>)
```

A.7.10 EXP

EXP 函数返回在表达式中指定的值的指数值。其语法如下：

```
EXP(<expression>)
```

expression 必须为 float 数据类型。

A.7.11 FLOOR

FLOOR 函数返回小于或等于表达式中指定的值的最大整数。其语法如下：

```
FLOOR(<expression>)
```

A.7.12 LOG

LOG 函数返回表达式中指定的值的自然对数。其语法如下：

```
LOG(<expression>)
```

expression 必须为 float 数据类型。

A.7.13 LOG10

LOG10 函数返回表达式中指定的值的以 10 为底的对数。其语法如下：

```
LOG10(<expression>)
```

expression 必须为 float 数据类型。

A.7.14 PI

PI 函数返回 PI 的常量值。其语法如下：

```
PI()
```

A.7.15 POWER

POWER 函数返回表达式中指定的值的指定幂的值。其语法如下：

```
POWER(<expression>, <power>)
```

A.7.16 RADIANS

RADIANS 函数对于表达式中指定以度数表示的角度返回相对应的弧度值。其语法如下：

```
RADIANS(<expression>)
```

A.7.17 RAND

RAND 函数返回 0~1 之间的随机值。其语法如下：

```
RAND([<seed>])
```

seed 值是个整数表达式，它指定了起始值。一旦指定了种子值，当前连接上对 RAND() 的后续调用都将基于该初始值选择下一个值。如果未指定种子值，SQL Server 将随机值用作初始种子。

提示：

在明确指定种子值时要小心。如果指定一个特定的种子值，SQL Server 将总是返回相同的数字序列，例如，用 RAND(10) 启动与服务器的连接，那每个连接对 RAND() 的前 3 次调用将生成 0.713759689954247、0.182458908613686 和 0.586642279446948——不是很随机。可以使用你自己的随机值，例如是基于当前时间的，但结果仍是一组相对有限的起始值，因此个人建议开始时使用 SQL Server 的随机值。

A.7.18 ROUND

ROUND 函数将表达式中指定的数字舍入为指定的长度：

```
ROUND(<expression>, <length> [, <function>])
```

length 参数指定了 expression 应舍入的精度。length 参数必须是 tinyint、smallint 或 int 数据

类型。可选的 `function` 参数可以用来指定数字是舍入还是截断。如果省略 `function` 值或者其值为 0(默认的), 那么 `expression` 将舍入。如果是 0 以外的值时, 将截断 `expression` 的值。

A.7.19 SIGN

`SIGN` 函数返回表达式的正负号。+1 为正数, 0 为零, -1 为负数。其语法如下:

```
SIGN(<expression>)
```

A.7.20 SIN

`SIN` 函数返回角度的正弦值。其语法如下:

```
SIN(<angle>)
```

`angle` 以弧度为单位, 而且必须为 `float` 数据类型。返回值也必须是 `float` 数据类型。

A.7.21 SQRT

`SQRT` 函数返回在表达式中指定的值的平方根。其语法如下:

```
SQRT(<expression>)
```

`expression` 必须为 `float` 数据类型。

A.7.22 SQUARE

`SQUARE` 函数返回在表达式中指定的值的平方。其语法如下:

```
SQUARE(<expression>)
```

`expression` 必须为 `float` 数据类型。

A.7.23 TAN

`TAN` 函数返回在表达式中指定的值的正切值。其语法如下:

```
TAN(<expression>)
```

`expression` 参数指定了弧度数, 并且必须为 `float` 或 `real` 数据类型。

A.8 基本的元数据函数

元数据函数提供有关数据库和数据库对象的信息。它们分别是:

- `COL_LENGTH`
- `COL_NAME`
- `COLUMNPROPERTY`
- `FULLTEXTSERVICEPROPERTY`
- `INDEX_COL`
- `INDEXKEY_PROPERTY`

- DATABASEPROPERTY
- DATABASEPROPERTYEX
- DB_ID
- DB_NAME
- FILE_ID
- FILE_NAME
- FILEGROUP_ID
- FILEGROUP_NAME
- FILEGROUPPROPERTY
- FILEPROPERTY
- FULLTEXTCATALOGPROPERTY
- INDEXPROPERTY
- OBJECT_ID
- OBJECT_NAME
- OBJECTPROPERTY
- OBJECTPROPERTYEX
- @@POCID
- SCHEMA_ID
- SCHEMA_NAME
- SQL_VARIANT_PROPERTY
- TYPE_ID
- TYPE_NAME
- TYPEPROPERTY

A.8.1 COL_LENGTH

COL_LENGTH 函数返回列的定义长度。其语法如下：

```
COL_LENGTH('<table>', '<column>')
```

column 参数指定要确定其长度的列的名称。table 参数指定了包含该列的表的名称。

A.8.2 COL_NAME

COL_NAME 函数根据指定的表 ID 号和列 ID 号返回数据库列的名称。其语法如下：

```
COL_NAME(<table_id>, <column_id>)
```

column_id 参数指定了列的 ID 号。table_id 参数指定了包含该列的表的 ID 号。

A.8.3 COLUMNPROPERTY

COLUMNPROPERTY 函数返回有关列或过程参数的信息。其语法如下：

```
COLUMNPROPERTY(<id>, <column>, <property>)
```

id 参数指定了表/过程的 ID。column 参数指定了列/参数的名称。property 参数指定了要为列或过程返回的数据。property 参数可以是下面这些值中的一个：

- AllowsNull——允许 NULL 值
- IsComputed——该列为计算列
- IsCursorType——过程参数类型为 CURSOR
- IsFullTextIndexed——已为该列编制了全文索引
- IsIdentity——该列是一个 IDENTITY 列
- IsIdNotForRepl——该列检查 IDENTITY NOT FOR REPLICATION
- IsOutParam——该过程参数为输出参数
- IsRowGuideCol——该列是一个 ROWGUIDCOL 列

- Precision——列或参数的数据类型的精度
- Scale——列或参数的数据类型的小数位数
- UseAnsiTrim——当开始创建表时, ANSI 填充设置为 ON

除了 Precision(返回数据类型的精度)和 Scale(返回小数位数)之外, 这个函数的返回值是, 1 为 True、0 为 False、输入无效时为 NULL。

A.8.4 DATABASEPROPERTY

DATABASEPROPERTY 函数返回指定数据库和属性名的设置。其语法如下:

```
DATABASEPROPERTY('<database>', '<property>')
```

database 参数指定了要返回其命名属性信息的数据库的名称。property 参数包含数据库属性的名称, 可以是下列值中的一个:

- IsAnsiNullDefault——数据库遵循 ANSI-92 标准, 允许 NULL 值。
- IsAnsiNullsEnabled——所有与 NULL 的比较都不能执行。
- IsAnsiWarningsEnabled——当标准错误条件发生时, 发出警告消息。
- IsAutoClose——数据库在最后一位用户退出后释放资源。
- IsAutoShrink——数据库文件可以自动定期收缩。
- IsAutoUpdateStatistics——启用自动更新统计选项。
- IsBulkCopy——数据库允许不记录日志的操作(例如大容量复制程序所执行的操作)。
- IsCloseCursorsOnCommitEnabled——关闭在提交事务时打开的游标。
- IsDboOnly——数据库仅允许 dbo 进行访问。
- IsDetached——分离操作分离了数据库。
- IsEmergencyMode——数据库处于紧急模式。
- IsFulltextEnabled——已为数据库启用了全文功能。
- IsInLoad——正在加载数据库。
- IsInRecovery——正在恢复数据库。
- IsInStandby——数据库处于只读方式, 并允许还原日志。
- IsLocalCursorsDefault——游标声明默认为 LOCAL。
- IsNotRecovered——数据库不能恢复。
- IsNullConcat——与 NULL 串联将产生 NULL。
- IsOffline——数据库脱机。
- IsQuotedIdentifiersEnabled——可对标识符使用双引号。
- IsReadOnly——数据库处于只读访问模式。
- IsRecursiveTriggersEnabled——启用触发器递归触发。
- IsShutDown——数据库启动时遇到问题。
- IsSingleUser——数据库处于单用户访问模式。
- IsSuspect——数据库可疑。
- IsTruncLog——数据库截断其登录检查点。
- Version——创建数据库时所使用的 SQL Server 代码的内部版本号。

除 Version(如果数据库打开, 函数将返回版本号, 如果关闭, 则返回 NULL)外, 这个函数的返回值是, 1 为 True、0 为 False、输入无效时为 NULL。

A.8.5 DATABASEPROPERTYEX

DATABASEPROPERTYEX 函数是 DATABASEPROPERTY 的超集, 返回指定数据库和属性名的当前设置。语法和 DATABASEPROPERTY 很像, 如下所示:

```
DATABASEPROPERTYEX('<database>', '<property>')
```

DATABASEPROPERTYEX 只比 DATABASEPROPERTY 多一些属性, 包括:

- Collation——返回数据库默认的排序规则名称(记住, 也可以在列级别对排序规则进行重写)。
- ComparisonStyle——指出特定排序规则的 Windows 比较样式(例如, 区分大小写)。
- IsAnsiPaddingEnabled——在比较或插入前, 字符串是否填充至相同长度。
- IsArithmeticAbortEnabled——在出现重大算术错误(如数据溢出)时是否终止查询。

database 参数指定了要为其返回命名属性数据的数据库的名称。property 参数包含了数据库属性的名称。

A.8.6 DB_ID

DB_ID 函数返回数据库 ID 号。其语法如下:

```
DB_ID(['<database_name>'])
```

可选的 database_name 参数指定了对应数据库 ID 号的数据库名。如果没有指定 database_name, 则会返回当前数据库的 ID 号。

A.8.7 DB_NAME

DB_NAME 函数返回对应指定 ID 号的数据库的名称。其语法如下:

```
DB_NAME([<database_id>])
```

可选的 database_id 参数指定了要返回的数据库的标识号。如果没有指定 database_id, 则会返回当前数据库的名称。

A.8.8 FILE_ID

FILE_ID 函数返回当前数据库中指定文件名的文件标识(ID)号。其语法如下:

```
FILE_ID('<file_name>')
```

file_name 参数指定了要返回其文件 ID 的文件的名称。

A.8.9 FILE_NAME

FILE_NAME 函数返回指定文件 ID 号的文件名称。其语法如下：

```
FILE_NAME(<file_id>)
```

file_id 参数指定了要返回其对应文件名的文件 ID 号。

A.8.10 FILEGROUP_ID

FILEGROUP_ID 函数返回指定文件组名称的文件组 ID 号。其语法如下：

```
FILEGROUP_ID('<filegroup_name>')
```

filegroup_name 参数指定了要返回其文件组 ID 的文件组的名称。

A.8.11 FILEGROUP_NAME

FILEGROUP_NAME 函数返回指定文件组 ID 号的文件组的名称。其语法如下：

```
FILEGROUP_NAME(<filegroup_id>)
```

filegroup_id 参数指定了要返回文件组名的文件组 ID 号。

A.8.12 FILEGROUPPROPERTY

根据给定的文件组和属性名，FILEGROUPPROPERTY 函数返回指定的文件组属性设值。其语法如下：

```
FILEGROUPPROPERTY(<filegroup_name>, <property>)
```

filegroup_name 参数指定了包含要查询的属性的文件组的名称。property 参数指定了查询的属性，并且可以为以下值之一：

- IsReadOnly——文件组名为只读。
- IsUserDefinedFG——文件组名为用户自定义的文件组。
- IsDefault——文件组名为默认的文件组。

这个函数的返回值是，1 为 True、0 为 False、输入无效时为 NULL。

A.8.13 FILEPROPERTY

根据给定的文件名和属性名，FILEPROPERTY 函数返回指定的文件名属性设值。其语法如下：

```
FILEPROPERTY(<file_name>, <property>)
```

file_name 参数指定了包含要查询的属性的文件的名称。property 参数指定了查询的属性，并且可以是下列值中的一个：

- IsReadOnly——文件为只读。
- IsPrimaryFile——文件为主文件。

- IsLogFile——文件为日志文件。
- SpaceUsed——指定的文件所占用的空间量。

除 SpaceUsed(它会返回文件中分配的页数)外, 这个函数的返回值是: 1 为 True、0 为 False、输入无效时为 NULL。

A.8.14 FULLTEXTCATALOGPROPERTY

FULLTEXTCATALOGPROPERTY 函数返回有关全文目录属性的信息。其语法如下:

```
FULLTEXTCATALOGPROPERTY(<catalog_name>, <property>)
```

catalog_name 参数指定了全文目录的名称。property 参数指定了所查询的属性。可以查询一下这些属性:

- PopulateStatus——可能的返回值是: 0(空闲), 1(正在进行完全填充), 2(已暂停), 3(中止), 4(正在恢复), 5(关机), 6(正在进行增量填充), 7(更新索引)。
- ItemCount——返回全文目录中当前全文索引项的数目。
- IndexSize——返回全文索引的大小(MB)。
- UniqueKeyCount——此目录中组成全文索引的唯一键数。
- LogSize——与全文目录相关联的错误日志组合集的大小, 以字节表示。
- PopulateCompletionAge——上一次全文索引填充的完成时间与 01/01/1990 00:00:00 之间的时间差, 用秒表示。

A.8.15 FULLTEXTSERVICEPROPERTY

FULLTEXTSERVICEPROPERTY 函数返回有关全文服务级属性的信息。其语法如下:

```
FULLTEXTSERVICEPROPERTY(<property>)
```

property 参数指定了待查询的服务级属性的名称。property 参数可以是下列值中的一个:

- ResourceUsage——返回一个从 1(后台)到 5(专用)之间的值。
- ConnectTimeout——返回在超时发生前, 搜索服务(Search Service)等待所有与 SQL Server 数据库服务器的连接完成以便进行全文索引填充所用的时间(以秒为单位)。
- IsFulltextInstalled——如果已在计算机上安装全文服务, 则返回 1, 否则返回 0。

A.8.16 INDEX_COL

INDEX_COL 函数返回索引列名称。其语法如下:

```
INDEX_COL('<table>', <index_id>, <key_id>)
```

table 参数指定了表的名称, index_id 指定了索引的 ID, key_id 指定了键的 ID。

A.8.17 INDEXKEY_PROPERTY

INDEXKEY_PROPERTY 函数返回关于索引键的信息。

```
INDEXKEY_PROPERTY(<table_id>, <index_id>, <key_id>, <property>)
```

`table_id` 参数是要对其检查的表的 ID 号，其数据类型为 `int`。使用 `OBJECT_ID` 找到数值 `table_id`。`index_id` 指定了索引 ID 号，其数据类型是 `int`。`key_id` 指定了键的索引列位置。例如，有一个键有 3 列，设置该值为 2 表示想要检查中间的列。`property` 是想要返回其属性值的两个属性之一的字符串标识符。可能的两个值分别是 `ColumnId`(返回物理列的 `column ID`)和 `IsDescending`(它返回列排序的顺序，1 为降序，0 为升序)。

A.8.18 INDEXPROPERTY

根据指定的表 ID 号、索引名和属性名，`INDEXPROPERTY` 函数返回指定的索引属性值。其语法如下：

```
INDEXPROPERTY(<table_ID>, <index>, <property>)
```

`property` 参数指定了要查询的索引属性。`property` 可以是下面这些值中的一个：

- `IndexDepth`——索引的深度
- `IsAutoStatistic`——索引是由 `sp_dboption` 的自动创建统计信息选项生成的。
- `IsClustered`——索引是群集的。
- `IsStatistics`——索引是由 `CREATE STATISTICS` 语句或由 `sp_dboption` 的自动创建统计信息选项生成。
- `IsUnique`——索引是唯一的。
- `IndexFillFactor`——索引指定其填充因子。
- `IsPadIndex`——索引指定在每个内部节点上将要保持开放的空间。
- `IsFulltextKey`——索引是表的全文键。
- `IsHypothetical`——索引是假设的，不能直接用作数据访问路径。

除 `IndexDepth`(它会返回索引级别数)和 `IndexFillFactor`(它会返回创建或重建索引时所使用的填充因子)外，这个函数的返回值是，1 为 `True`、0 为 `False`、输入无效时为 `NULL`。

A.8.19 OBJECT_ID

`OBJECT_ID` 函数返回指定的数据库对象的 ID 号。其语法如下：

```
OBJECT_ID('<object>')
```

A.8.20 OBJECT_NAME

`OBJECT_NAME` 函数返回指定数据库对象的名称。其语法如下：

```
OBJECT_NAME(<object_id>)
```

A.8.21 OBJECTPROPERTY

`OBJECTPROPERTY` 函数返回当前数据库中对象的相关信息。其语法如下：

```
OBJECTPROPERTY(<id>, <property>)
```


id 参数指定了所需对象的 ID。property 参数指定需要的对象信息。property 可以是下面这些值中的一个：

CnstIsClustKey	ExecIsTriggerDisabled
CnstIsColumn	ExecIsTriggerNotForRepl
CnstIsDeleteCascade	ExecIsUpdateTrigger
CnstIsDisabled	HasAfterTrigger
CnstIsNonclustKey	HasDeleteTrigger
CnstIsNotRepl	HasInsertTrigger
CnstIsNotTrusted	HasInsteadOfTrigger
CnstIsUpdateCascade	HasUpdateTrigger
ExecIsAfterTrigger	IsAnsiNullsOn
ExecIsAnsiNullsOn	IsCheckCnst
ExecIsDeleteTrigger	IsConstraint
ExecIsFirstDeleteTrigger	IsDefault
ExecIsFirstInsertTrigger	IsDefaultCnst
ExecIsFirstUpdateTrigger	IsDeterministic
ExecIsInsertTrigger	IsExecuted
ExecIsInsteadOfTrigger	IsExtendedProc
ExecIsLastDeleteTrigger	IsForeignKey
ExecIsLastInsertTrigger	IsIndexed
ExecIsLastUpdateTrigger	IsIndexable
ExecIsQuotedIdentOn	IsInlineFunction
ExecIsStartup	IsMSShipped
IsPrimaryKey	TableFulltextPopulateStatus
IsProcedure	TableHasActiveFulltextIndex
IsQuotedIdentOn	TableHasCheckCnst
IsQueue	TableHasClustIndex
IsReplProc	TableHasDefaultCnst
IsRule	TableHasDeleteTrigger
IsScalarFunction	TableHasForeignKey
IsSchemaBound	TableHasForeignRef
IsSystemTable	TableHasIdentity
IsTable	TableHasIndex
IsTableFunction	TableHasInsertTrigger
IsTrigger	TableHasNonclustIndex
IsUniqueCnst	TableHasPrimaryKey
IsUserTable	TableHasRowGuidCol
IsView	TableHasTextImage
OwnerId	TableHasTimestamp
TableDeleteTrigger	TableHasUniqueCnst

TableDeleteTriggerCount	TableHasUpdateTrigger
TableFullTextBackgroundUpdateIndexOn	TableInsertTrigger
TableFulltextCatalogId	TableInsertTriggerCount
TableFullTextChangeTrackingOn	TableIsFake
TableFulltextDocsProcessed	TableIsLockedOnBulkLoad
TableFulltextFailCount	TableIsPinned
TableFulltextItemCount	TableTextInRowLimit
TableFulltextKeyColumn	TableUpdateTrigger
TableFulltextPendingChanges	TableUpdateTriggerCount

除了下列情况外，这个函数的返回值是：1 为 True、0 为 False、输入无效时为 NULL。

- **OwnerId**——返回对象的所有者的数据库用户 ID——注意，这与对象的 SchemaID 不同，在 SQL Server 2005 或更高版本中可能不是很有用。
- **TableDeleteTrigger**、**TableInsertTrigger**、**TableUpdateTrigger**——它们都会返回指定类型的第一个触发器的 ID。如果无该类型的触发器存在，则返回 0。
- **TableDeleteTriggerCount**、**TableInsertTriggerCount**、**TableUpdateTriggerCount**——返回特定表中指定类型触发器的数目。
- **TableFulltextCatalogId**——返回全文目录的 ID(如果有的话)，如果表中没有全文目录，则返回 0。
- **TableFulltextKeyColumn**——返回用作全文索引的唯一索引的列的 ColumnID。
- **TableFulltextPendingChanges**——返回自最后一次对表运行全文分析后改变的项数。必须对该函数启用更改跟踪来返回有用的结果。
- **TableFulltextPopulateStatus**——其返回值的可能性有多种：
 - 0——表示全文填充进程目前空闲。
 - 1——表示正在进行完全填充。
 - 2——表示正在进行增量填充。
 - 3——表示正在分析更改并添加到全文目录中。
 - 4——表示正在进行某种后台更新(如自动更改跟踪机制做的工作)。
 - 5——表示全文操作在进行中，但已被中止(允许其他系统请求执行)或暂停。
- 可以利用这一选项的反馈确定其他合适的全文相关选项(检查是否正在进行填充，从而知道其他函数是否有效，如 **TableFulltextDocsProcessed**)。
- **TableFulltextDocsProcessed**——这只在全文索引实际运行时有效，它返回自全文索引处理任务开始以来处理的行数。结果为 0 表示当前没有运行全文索引(结果为 null 表明没有为表配置全文索引)。
- **TableFulltextFailCount**——这只在全文索引实际运行时有效，它返回全文索引因为某种原因跳过(没有指示原因)的行数。和 **TableFulltextDocsProcessed** 一样，结果为 0 表示当前没有对表全文分析，结果为 null 表明没有为表配置全文索引)。
- **TableIsPinned**——这个保留下来只是为了向后兼容，在 SQL Server 2005 和更高版本中将总是返回 0。

A.8.22 OBJECTPROPERTYEX

OBJECTPROPERTYEX 是 OBJECTPROPERTY 函数的扩展版本。

OBJECTPROPERTYEX(<id>, <property>)

和 OBJECTPROPERTY 一样, id 参数指定了某个对象的 ID。property 参数指定需要的对象信息。OBJECTPROPERTYEX 支持 OBJECTPROPERTY 所支持的所有属性值,但同时还支持下列额外的属性值:

- BaseType——返回对象的基本数据类型。
- IsPrecise——表明对象是否包含不精确计算。例如, int 或 decimal 是精确的,而 float 不是。利用非精确数据类型的计算将返回不精确的结果。注意,可以将任何 .NET 程序集明确标记为精确或不精确的。
- IsSystemVerified——表明是否可由 SQL Server 本身(相对于由用户设置而言)验证 IsPrecise 和 IsDeterministic 属性。
- SchemaId——顾名思义,返回给定对象的内部系统 ID。然后可使用 SCHEMA_NAME 为该架构 ID 添加一个更为用户友好的名称。
- SystemDataAccess——表明对象是否依赖于任何系统表数据。
- UserDataAccess——表明对象是否利用任何用户表或系统用户数据。

A.8.23 @@PROCID

返回当前运行过程的存储过程 ID。

当进程正在运行而且用了大量资源时,它是主要的故障排除工具。主要用作 DBA 函数。

A.8.24 SCHEMA_ID

给定一个模式名,返回该模式的内部系统 ID。其语法如下:

SCHEMA_ID(<schema name>)

A.8.25 SCHEMA_NAME

给定一个内部模式系统 ID,返回该模式的用户友好的名称。其语法如下:

SCHEMA_NAME(<schema id>)

A.8.26 SQL_VARIANT_PROPERTY

SQL_VARIANT_PROPERTY 是一个很强大的函数,它会返回有关 sql_variant 的信息。这个信息包括 BaseType、Precision、Scale、TotalBytes、Collation 和 MaxLength 的信息。其语法如下:

SQL_VARIANT_PROPERTY (expression, property)

expression 是 sql_variant 类型的表达式。property 可以下面这些值中的一个,如表 A-4 所示。

表 A-4

值	描 述	返回的 sql_variant 的基本类型
BaseType	数据类型包括: char、int、money、nchar、ntext、numeric、nvarchar、real、smalldatetime、smallint、smallmoney、text、timestamp、tinyint、uniqueidentifier、varbinary、varchar	sysname
Precision	数字基本数据类型的精度: datetime = 23 smalldatetime = 16 float = 53 real = 24 decimal (p,s)和 numeric (p,s) = p money = 19 smallmoney = 10 int = 10 smallint = 5 tinyint = 3 bit = 1 所有其他类型 = 0	int
Scale	int 数字基本数据类型小数点右边的位数: decimal (p,s) 和 numeric (p,s) = s money 和 smallmoney = 4 datetime = 3 所有其他类型 = 0	int
TotalBytes	包含值的元数据和数据所需的字节数。如果该值大于 900, 索引创建将失败	int
Collation	表示特定 sql_variant 值的排序规则	sysname
MaxLength	最大数据类型长度(以字节为单位)	int

A.8.27 TYPEPROPERTY

TYPEPROPERTY 函数返回有关数据类型的信息。其语法如下:

```
TYPEPROPERTY(<type>, <property>)
```

type 参数指定了数据类型的名称。property 参数指定了要查询的数据类型的属性, 它可以是下面这些值中的一个:

- Precision——返回数字位数/字符数
- Scale——返回小数位数
- AllowsNull——返回 1 为 True, 0 为 False
- UsesAnsiTrim——返回 1 为 True, 0 为 False

A.9 行集函数

这些行集函数返回一个可用于在 T-SQL 语句中代替表引用的对象。行集函数包括：

- CHANGETABLE
- CONTAINSTABLE
- FREETEXTTABLE
- OPENDATASOURCE
- OPENQUERY
- OPENROWSET
- OPENXML

A.9.1 CHANGETABLE

```
CHANGETABLE (  
    { CHANGES <table> , <last sync version>  
      | VERSION <table> , <primary key values> } )  
[AS] <table alias> [ ( <column alias> [ ,...n ] )
```

返回自 “last sync version” 参数中指定的时间点后特定表中的所有行。

A.9.2 CONTAINSTABLE

CONTAINSTABLE 函数用于全文查询中。第 18 章提供了一个有关其使用的示例。其语法如下：

```
CONTAINSTABLE (<table>, {<column> | *}, '<contains_search_condition>')
```

A.9.3 FREETEXTTABLE

FREETEXTTABLE 函数用于全文查询中。第 18 章提供了一个有关其使用的示例。其语法如下。

```
FREETEXTTABLE (<table>, {<column> | *}, '<freetext_string>')
```

A.9.4 OPENDATASOURCE

OPENDATASOURCE 函数提供特殊的连接信息。其语法如下：

```
OPENDATASOURCE (<provider_name>, <init_string>)
```

`provider_name` 是注册为用于访问数据源的 OLE DB 提供程序的 ProgID 的名称。`init_string` 对于 VB 编程人员来说并不陌生，它是 OLE DB 提供程序的初始化字符串。例如，`init_string` 可以是如下的形式：

```
"User Id=wonderison;Password=JuniorBlues;DataSource=MyServerName"
```

A.9.5 OPENQUERY

OPENQUERY 函数在指定的 `linked_server` 上执行指定的传递 `query`。其语法如下：

```
OPENQUERY(<linked_server>, '<query>')
```

A.9.6 OPENROWSET

OPENROWSET 函数访问 OLE DB 数据源中的远程数据。其语法如下：

```
OPENROWSET('<provider_name>'
  {
    '<datasource>'; '<user_id>'; '<password>'
    | '<provider_string>'
  },
  {
    [<catalog.>][<schema.>]<object>
    | '<query>'
  })
```

`provider_name` 参数是一个字符串，表示在注册表中指定的 OLE DB 的友好名称。`data_source` 参数是字符串，它对应着所需的 OLE DB 数据源。`user_id` 参数是传递到 OLE DB 提供程序的相关用户名。`password` 参数是与 `user_id` 相关的密码。

`provider_string` 参数是特定于提供程序的连接字符串，用于替代 `datasource`、`user_id` 和 `password` 的组合。

`catalog` 参数是包含所需对象的目录/数据库的名称。`schema` 参数是模式名称或者是所需对象的对象所有者。`object` 参数是对象名。

`query` 参数是提供程序执行的字符串，用于替代 `catalog`、`schema` 和 `object` 的组合。

A.9.7 OPENXML

通过传递 XML 文档作为参数，或者检索 XML 文档以及在一个变量中定义文档，OPENXML 允许我们检查结构并返回数据，就好像 XML 文档是一张表一样。其语法如下：

```
OPENXML(<idoc_int> [in], <rowpattern> nvarchar[in], [<flags> byte[in]])
[WITH (<SchemaDeclaration> | <TableName>)]
```

`idoc_int` 参数是使用 `sp_xml_preparedocument` 系统存储过程定义的变量。`Rowpattern` 是节点的定义。`flags` 参数指定了 XML 文档和 SELECT 语句内返回的行集之间的映射。`SchemaDeclaration` 定义了 XML 文档的 XML 模式；如果在数据库中有遵循 XML 模式的表，那么就可以使用 `TableName`。

在使用 XML 文档之前，必须使用 `sp_xml_preparedocument` 系统过程来对它进行准备。

A.10 安全函数

安全函数返回有关用户和角色的信息。它们分别是：

- HAS_DBACCESS
- IS_MEMBER
- IS_SRVROLEMEMBER
- SUSER_ID
- SUSER_NAME
- SUSER_SID
- USER
- USER_ID
- USER_NAME

A.10.1 HAS_DBACCESS

HAS_DBACCESS 函数是用来确定登录的用户是否可以访问数据库。返回值为 1 意味着用户有访问权，如果为 0 则不能访问该数据库。NULL 值意味着提供的 `database_name` 无效。其语法如下：

```
HAS_DBACCESS ('<database_name>')
```

A.10.2 IS_MEMBER

IS_MEMBER 函数返回当前用户是否是指定 Windows NT 组/SQL Server 角色的成员。其语法如下：

```
IS_MEMBER (('<group>' | '<role>'))
```

`group` 参数指定了 NT 组的名称，并且必须使用 `domain\group` 这种格式。`role` 参数指定了 SQL Server 角色的名称。它可以是数据库固定角色或用户定义的角色，但不能是服务器角色。

如果当前用户是指定组或角色的成员，则该函数返回 1；如果不是指定组或角色的成员，则返回 0；如果指定的组或角色无效，则返回 NULL。

A.10.3 IS_SRVROLEMEMBER

IS_SRVROLEMEMBER 函数的返回值指明用户是否是指定的服务器角色的成员。其语法如下：

```
IS_SRVROLEMEMBER ('<role>' [, '<login>'])
```

可选的 `login` 参数是即将检查的登录帐户的名称；默认是当前用户。`role` 参数指定了服务器角色，并且必须是以下这些值中的一个：

- sysadmin

- dbcreator
- diskadmin
- processadmin
- serveradmin
- setupadmin
- securityadmin

如果指定登录帐户是指定角色的成员，则函数返回 1；如果不是，则返回 0；如果角色或登录名无效，则返回 NULL。

A.10.4 SUSER_ID

SUSER_ID 函数返回指定用户的登录 ID 号。其语法如下：

```
SUSER_ID(['<login>'])
```

login 参数是指定用户的登录 ID 名称。如果没有提供 login 值，则会默认使用当前的用户。SQL Server 2000 包括了 SUSER_ID 系统函数只是为了向后兼容。请改用 SUSER_SID。

A.10.5 SUSER_NAME

SUSER_NAME 函数返回指定用户的登录 ID 名称。其语法如下：

```
SUSER_NAME([<server_user_id>])
```

server_user_id 参数指定了用户的登录 ID 号。如果没有提供 server_user_id 的值，则会默认地使用当前用户的 ID 号。

提示：

SQL Server 2000 包括了 SUSER_NAME 系统函数只是为了向后兼容。请尽可能改用 SUSER_SNAME。

A.10.6 SUSER_SID

SUSER_SID 函数返回指定用户的安全标识号(SID)。其语法如下：

```
SUSER_SID(['<login>'])
```

login 参数是用户的登录名。如果没有提供该值，则会使用当前用户。

A.10.7 SUSER_SNAME

SUSER_SNAME 函数返回与指定安全标识号(SID)关联的登录 ID 名称。其语法如下：

```
SUSER_SNAME([<server_user_sid>])
```

server_user_sid 参数是用户的 SID。如果没有提供 server_user_sid 的值，则会使用当前用户的 SID。

A.10.8 USER

当未提供默认值时，USER 函数允许将系统为当前用户的数据库用户名提供的值插入表内。其语法如下：

USER

A.10.9 USER_ID

USER_ID 函数返回指定用户的数据库 ID 号。其语法如下：

USER_ID(['<user>'])

user 参数是要使用的用户名。如果没有提供 user 的值，则会使用当前用户。

A.10.10 USER_NAME

USER_NAME 函数在功能上与 USER_ID 相反，它根据指定的数据库 ID 号，返回数据库用户名。其语法如下：

USER_NAME([<user id>])

user id 参数指定了与要返回的用户名关联的 ID 号，如果未提供 user id，则假定为当前用户。

A.11 字符串函数

字符串函数对字符串输入值执行操作，返回字符串或数字值。字符串函数包括：

- ASCII
- CHAR
- CHARINDEX
- DIFFERENCE
- LEFT
- LEN
- LOWER
- LTRIM
- NCHAR
- PATINDEX
- QUOTENAME
- REPLACE
- REPLICATE
- REVERSE
- RIGHT
- RTRIM
- SOUNDEX
- SPACE
- STR
- STUFF
- SUBSTRING
- UNICODE
- UPPER

A.11.1 ASCII

ASCII 函数返回字符表达式(character_expression)最左端字符的 ASCII 代码值。其语法如下：

ASCII(<character_expression>)

A.11.2 CHAR

CHAR 函数将 ASCII 代码(expression 中指定的)转换为字符串。其语法如下:

CHAR(<expression>)

expression 可以是 0 到 255 之间的任何整数。

A.11.3 CHARINDEX

CHARINDEX 函数返回 character_string 中指定 expression 的起始位置。其语法如下:

CHARINDEX(<expression>, <character_string> [, <start_location>])

expression 参数是要查找的字符串。character_string 是被搜索的字符串,通常是一列。start_location 是搜索的起始位置。如果它是负数或零,则将从 character_string 的起始处查找。

A.11.4 DIFFERENCE

DIFFERENCE 函数以整数返回两个字符表达式的 SOUNDEX 值之差。其语法如下:

DIFFERENCE(<expression1>, <expression2>)

这个函数返回 0~4 之间的整数值。如果两个表达式相差不大(例如, blue 和 blew),则会返回 4。如果没有任何相似的地方,则返回 0。

A.11.5 LEFT

LEFT 函数返回从表达式的最左边开始指定个数的字符。其语法如下:

LEFT(<expression>, <integer>)

expression 参数指将要提取其最左面部分的字符数据。integer 参数指定了从左面开始的字符数,必须为正数。

A.11.6 LEN

LEN 函数返回指定 expression 中的字符数。其语法如下:

LEN(<expression>)

A.11.7 LOWER

LOWER 函数将 expression 中的任何大写字符转换为小写字符。其语法如下:

LOWER(<expression>)

A.11.8 LTRIM

LTRIM 函数删除 `character_expression` 中的前导空格。其语法如下：

```
LTRIM(<character_expression>)
```

A.11.9 NCHAR

NCHAR 函数返回具有指定 `integer_code` 的 Unicode 字符。其语法如下：

```
NCHAR(<integer_code>)
```

`integer_code` 参数必须是一个 0 到 65 535 的正整数。

A.11.10 PATINDEX

PATINDEX 函数返回指定表达式中某模式第一次出现的起始位置；如果没有找到该模式，则返回 0。其语法如下：

```
PATINDEX('%pattern%', <expression>)
```

`pattern` 参数是要查找的模式字符串。可以使用通配符，但是 `pattern` 之前和之后必须有 % 字符。`expression` 参数是要在其中搜索指定模式的字符数据——通常是一列。

A.11.11 QUOTENAME

QUOTENAME 函数返回带有分隔符的 Unicode 字符串，添加分隔符可使输入的字符串成为有效的 SQL Server 分隔标识符。其语法如下：

```
QUOTENAME(<character_string> [, '<quote_character>'])
```

`character_string` 参数是 Unicode 字符串。`quote_character` 参数是用作分隔符的单字符字符串。`quote_character` 参数可以是单引号(')、左括号或右括号([])或者双引号(")。默认使用括号。

A.11.12 REPLACE

REPLACE 函数用指定的第三个字符串替换第一个字符串中出现的所有第二个指定字符串的实例。其语法如下：

```
REPLACE(<string_expression1>, '<string_expression2>', '<string_expression3>')
```

`string_expression1` 参数是待搜索的表达式。`string_expression2` 参数是 `string_expression1` 中待查找的字符串表达式。`string_expression3` 参数是替换所有 `string_expression2` 的实例的字符串表达式。

A.11.13 REPLICATE

REPLICATE 函数以指定的次数重复 `character_expression`。其语法如下：

```
REPLICATE(<character_expression>, <integer>)
```

A.11.14 REVERSE

REVERSE 函数返回指定 `character_expression` 的反转值。其语法如下：

```
REVERSE(<character_expression>)
```

A.11.15 RIGHT

RIGHT 函数返回指定 `character_expression` 中从右边开始指定个数的字符(`integer` 值)。其语法如下：

```
RIGHT(<character_expression>, <integer>)
```

`integer` 参数必须是一个正整数。

A.11.16 RTRIM

RTRIM 函数从 `character_expression` 中删除所有尾随空格。其语法如下：

```
RTRIM(<character_expression>)
```

A.11.17 SOUNDEX

SOUNDEX 返回一个由 4 个字符组成的代码(SOUNDEX)，用于评估两个字符串的相似性。其语法如下：

```
SOUNDEX(<character_expression>)
```

A.11.18 SPACE

SPACE 函数返回由重复的空格组成的字符串，由 `integer` 表示其长度。其语法如下：

```
SPACE(<integer>)
```

A.11.19 STR

STR 函数把数字数据转换为字符数据。其语法如下：

```
STR(<numeric_expression>[, <length>[, <decimal>]])
```

`numeric_expression` 参数是带小数点的数字表达式。`length` 参数是总长度，包括小数点、符号、数字或空格。`decimal` 参数是小数点右边的位数。

A.11.20 STUFF

STUFF 函数删除指定长度的字符并在该位置处插入另一组字符。其语法如下：


```
STUFF(<expression>, <start>, <length>, <characters>)
```

expression 参数是要对其作删除和增加的字符串。**start** 参数指定了删除和插入字符的开始位置。**length** 参数指定了要删除的字符数。**characters** 参数指定了要插入到 **expression** 中的新的字符集。

A.11.21 SUBSTRING

SUBSTRING 函数返回表达式的一部分。其语法如下：

```
SUBSTRING(<expression>, <start>, <length>)
```

expression 参数指定了将从中取子串的数据，可以是字符串、二进制字符串、文本或者包含表的表达式。**start** 参数指定子串的开始位置，是整数。**length** 参数指定了子串的长度。

A.11.22 UNICODE

UNICODE 函数返回 **character_expression** 中第一个字符的 UNICODE 值。其语法如下：

```
UNICODE('<character_expression>')
```

A.11.23 UPPER

UPPER 函数将 **character_expression** 中的所有小写字符转换为大写的字符。其语法如下：

```
UPPER(<character_expression>)
```

A.12 系统函数

系统函数返回有关 SQL Server 中的值、对象和设置的信息。它们包括：

- APP_NAME
- CASE
- CAST 和 CONVERT
- COALESCE
- COLLATIONPROPERTY
- CURRENT_TIMESTAMP
- CURRENT_USER
- DATALENGTH
- FORMATMESSAGE
- GETANSINULL
- HOST_ID
- HOST_NAME
- IDENT_CURRENT
- IDENT_INCR
- IDENT_SEED
- ISDATE
- ISNULL
- ISNUMERIC
- NEWID
- NULLIF
- PARSENAME
- PERMISSIONS
- ROWCOUNT_BIG
- SCOPE_IDENTITY
- SERVERPROPERTY
- SESSION_USER
- SESSIONPROPERTY
- STATS_DATE
- SYSTEM_USER
- USER_NAME

- IDENTITY

A.12.1 APP_NAME

APP_NAME 函数返回当前会话的应用程序名称(如果应用程序已经予以设置), 返回值为 nvarchar 类型。其语法如下:

```
APP_NAME()
```

A.12.2 CASE

CASE 函数计算条件列表并返回多个可能结果表达式之一。它有两种格式:

- 简单 CASE 函数将某个表达式与一组简单表达式进行比较以确定结果。
- 搜索 CASE 函数计算一组布尔表达式以确定结果。

提示:

这两种格式都支持可选的 ELSE 参数。

1. 简单 CASE 函数:

```
CASE <input_expression>
  WHEN <when_expression> THEN <result_expression>
  ELSE <else_result_expression>
END
```

2. 搜索 CASE 函数:

```
CASE
  WHEN <Boolean_expression> THEN <result_expression>
  ELSE <else_result_expression>
END
```

A.12.3 CAST 和 CONVERT

这两个函数提供了将一种数据类型转换成另一种数据类型值的相似功能。

1. 使用 CAST

```
CAST(<expression> AS <data_type>)
```

2. 使用 CONVERT

```
CONVERT (<data_type>[(<length>)], <expression> [, <style>])
```

当转换为字符数据类型时, 这里的 style 指日期格式的样式。

A.12.4 COALESCE

COALESCE 函数接收数量不定的参数, 返回其参数中第一个非空表达式。其语法如下:

```
COALESCE(<expression> [,...n])
```


如果所有的参数为 NULL，则 COALESCE 返回 NULL。

A.12.5 COLLATIONPROPERTY

COLLATIONPROPERTY 函数返回给定排序规则的属性。其语法如下：

```
COLLATIONPROPERTY(<collation_name>, <property>)
```

collation_name 参数是想要使用的排序规则的名称，property 是想确定的排序规则属性，可以为表 A-5 中 3 个值中的一个。

表 A-5

属 性 名 称	描 述
CodePage	排序规则的非 Unicode 代码页
LCID	排序规则的 Windows LCID。如果是 SQL 排序规则，就返回 NULL
ComparisonStyle	排序规则的 Windows 比较样式。如果是二进制或 SQL 排序规则，就返回 NULL

A.12.6 CURRENT_USER

CURRENT_USER 函数返回当前 sysname 类型的用户。此函数等价于 USER_NAME()。其语法如下：

```
CURRENT_USER
```

A.12.7 DATALENGTH

DATALENGTH 函数返回用于表示 expression 的字节数，该值为整数。DATALENGTH 对 varchar、varbinary、text、image、nvarchar 和 ntext 数据类型特别有用，因为这些数据类型可以存储变长数据。其语法如下：

```
DATALENGTH(<expression>)
```

A.12.8 @@ERROR

返回当前连接上运行的最后一条 T-SQL 语句的错误码。如果没有错误，则该值为 0。如果要编写存储过程或触发器，这是必需的系统函数；不能没有它。

提示：

使用 @@ERROR 需要记住的是，它的生存期只是一条语句。这意味着，如果想要在一条指定语句之后检查错误，那么需要检测下一条语句，或者需要将它移到临时保存变量中。一般来说，我建议在 TRY...CATCH 块中使用 ERROR_NUMBER()，除非需要支持 SQL Server 2005 之前的代码。

使用 master 数据库中的 sys.messages 系统表，可以查看到所有系统错误的列表。要创建自定义的错误，可使用 sp_addmessage。

A.12.9 FORMATMESSAGE

FORMATMESSAGE 函数根据 sysmessages 中现有的消息构造消息。其语法如下：

```
FORMATMESSAGE(<msg_number>, <param_value>[,...n])
```

其中的 msg_number 是 sysmessages 中消息的 ID。

提示：

FORMATMESSAGE 查找用户当前语言的消息。如果消息没有本地化版本，则使用美国英语版本。

A.12.10 GETANSINULL

GETANSINULL 函数返回数据库的默认可空性，返回值为整数。其语法如下：

```
GETANSINULL(['<database>'])
```

database 参数是要返回可空性信息的数据库的名称。

如果指定数据库的可空性允许 NULL 值，并且列或数据类型可空性没有显式定义，则 GETANSINULL 返回 1。这是 ANSI NULL 默认值。

A.12.11 HOST_ID

HOST_ID 函数返回工作站的 ID 号。其语法如下：

```
HOST_ID()
```

A.12.12 HOST_NAME

HOST_NAME 函数返回工作站的名称。其语法如下：

```
HOST_NAME()
```

A.12.13 IDENT_CURRENT

IDENT_CURRENT 函数返回为指定表生成的最后一个标识值，它针对的是任何会话或表作用域。IDENT_CURRENT 类似于 @@IDENTITY 和 SCOPE_IDENTITY。不过，它对返回该值而搜索的作用域没有限制。

其语法如下：

```
IDENT_CURRENT('<table_name>')
```

table_name 是将要返回其标识值的表的名称。

A.12.14 IDENT_INCR

IDENT_INCR 函数返回在带有标识列的表或视图中创建标识列时指定的增量值。其语法

如下:

```
IDENT_INCR('<table_or_view>')
```

`table_or_view` 参数是一个表达式, 用来指定表或视图以检查有效的标识增量值。

A.12.15 IDENT_SEED

`IDENT_SEED` 函数返回在带有标识列的表或视图中创建标识列时指定的种子值。其语法如下:

```
IDENT_SEED('<table_or_view>')
```

`table_or_view` 参数是一个表达式, 用来指定表或视图以检查有效的种子值。

A.12.16 @@IDENTITY

返回当前连接创建的最后一个标识值。

如果使用了标识列, 然后在另一个表中将它们作为外键引用, 那么你会发现一直都在使用这个标识列。可以创建父记录(通常带有你需要检索的标识列), 然后选择`@@IDENTITY` 来了解子记录需要关联的是哪个值。

如果使用标识值来执行对多个表的插入, 那么请记住`@@IDENTITY` 中的值只是最后一个插入的标识值; 任何在其前面插入的值都会丢失, 除非在每次插入后都将该值放入到一个临时保存变量中。同样地, 如果最后插入的列没有标识列, 那么`@@IDENTITY` 将被设置为 `NULL`。

A.12.17 IDENTITY

`IDENTITY` 函数用于将标识列插入到新表中。它只用在带有 `INTO table` 子句的 `SELECT` 语句中。其语法如下:

```
IDENTITY(<data_type>[, <seed>, <increment>]) AS <column_name>
```

其中:

- `data_type` 是标识列的数据类型。
- `seed` 是要指派给表中第一行的值。每一个后续行被指派下一个标识值, 该值等于上一个 `IDENTITY` 值加上 `increment` 值。如果既没有指定 `seed`, 也没有指定 `increment`, 那么它们都默认为 1。
- `increment` 是用来添加到表中后续行的 `seed` 值上的增量。
- `column_name` 是将插入到新表中的列的名称。

A.12.18 ISNULL

`ISNULL` 函数检查表达式的 `NULL` 值, 并使用指定的替换值替换 `NULL`。其语法如下:

```
ISNULL(<check_expression>, <replacement_value>)
```

A.12.19 ISNUMERIC

ISNUMERIC 函数确定表达式是否为一个有效的数字类型。其语法如下：

```
ISNUMERIC(<expression>)
```

A.12.20 NEWID

NEWID 函数创建 `uniqueidentifier` 类型的唯一值。其语法如下：

```
NEWID()
```

A.12.21 NULLIF

NULLIF 函数比较两个表达式并返回空值。其语法如下：

```
NULLIF(<expression1>, <expression2>)
```

A.12.22 PARSENAME

PARSENAME 返回对象名称的指定部分。其语法如下：

```
PARSENAME('<object_name>', <object_piece>)
```

`object_name` 参数指定了要检索其指定部分的对象名。`object_piece` 参数指定了要返回的对象部分。`object_piece` 参数可以为下列这些值之一：

- 1——对象名
- 2——所有者名称
- 3——数据库名称
- 4——服务器名称

A.12.23 PERMISSIONS

PERMISSIONS 函数返回一个包含位图的值，表明当前用户的语句、对象或列权限。其语法如下：

```
PERMISSIONS([<objectid> [, '<column>']])
```

`object_id` 参数指定了对象的 ID。可选的 `column` 参数指定了要返回其权限信息的列的选名。

A.12.24 @@ROWCOUNT

返回上一条语句所影响到的行数。

它是最常用的全局变量之一，经常使用它来检查非运行时错误：也就是说，那些对于程序来说是逻辑错误，但是 SQL Server 不能识别的错误。例如，根据某一条件执行更新，但是发现影响的行数为 0 的情况。这很奇怪，如果客户提交了对特定行的修改操作，那么该行应是匹配指定条件的；而 0 行受到影响表明存在错误。

然而，如果在不返回行的任何语句上测试该系统函数，也会得到返回值 0。

A.12.25 ROWCOUNT_BIG

ROWCOUNT_BIG 函数的功能与 @@ROWCOUNT 类似，返回最后一条语句影响的行数。不过，ROWCOUNT_BIG 的返回类型是 bigint 数据类型。其语法如下：

```
ROWCOUNT_BIG()
```

A.12.26 SCOPE_IDENTITY

SCOPE_IDENTITY 函数返回插入到同一作用域中(也就是说，在相同的存储过程、触发器、函数或批处理中)的标识列内的最后一个值。它和前面讨论过的 IDENT_CURRENT 在功能上相似，不过 IDENT_CURRENT 不局限于在同一个作用域内的标识插入。

这个函数返回 sql_variant 数据类型，其语法如下：

```
SCOPE_IDENTITY()
```

A.12.27 SERVERPROPERTY

SERVERPROPERTY 函数返回有关当前服务器的信息。其语法如下：

```
SERVERPROPERTY('<propertyname>')
```

propertyname 可以是表 A-6 中列出的值中的一个。

表 A-6

属 性 名 称	返 回 的 值
Collation	服务器的默认排序规则名
Edition	安装在服务器上的 SQL Server 实例版本。返回下列 nvarchar 结果中的一个： 'Desktop Engine' 'Developer Edition' 'Enterprise Edition' 'Enterprise Evaluation Edition' 'Personal Edition' 'Standard Edition'
Engine Edition	安装在服务器上的 SQL Server 实例的引擎版本： 1 = Personal 或 Desktop Engine 2 = Standard 3 = Enterprise(对 Enterprise、Enterprise Evaluation 和 Developer 返回该值)
InstanceName	用户连接到的实例的名称
IsClustered	将确定在故障转移群集中是否配置服务器实例： 1 = 群集 0 = 非群集 NULL = 输入无效或发生错误

(续表)

属性名称	返回的值
IsFullTextInstalled	确定在 SQL Server 的当前实例中是否安装全文组件: 1 = 已安装全文组件 0 = 未安装全文组件 NULL = 输入无效或发生错误
IsIntegratedSecurityOnly	确定服务器是否为集成的安全模式: 1 = 集成安全模式 0 = 非集成安全模式 NULL = 输入无效或发生错误
IsSingleUser	确定服务器是否为单用户安装: 1 = 单用户 0 = 非单用户 NULL = 无效输入或错误
IsSyncWithBackup	确定数据库是发布数据库还是分发数据库, 是否可在不中断当前事务复制的情况下还原: 1 = 真 0 = 假
LicenseType	为 SQL Server 实例安装什么类型的许可 PER_SEAT = 每客户模式 PER_PROCESSOR = 每处理器模式 DISABLED = 禁用许可
MachineName	返回运行服务器实例的 Windows NT 计算机名称 对于群集实例(即在 Microsoft Cluster Server 的虚拟服务器上运行的 SQL Server 实例), 返回虚拟服务器的名称
NumLicenses	如果是每客户模式, 则为该 SQL Server 实例的注册客户端许可数 如果是每处理器模式, 则为该 SQL Server 实例的许可处理器数
ProcessID	SQL Server 服务的进程 ID(ProcessID 在标识属于该实例的 sqlservr.exe 方面很有用)
ProductVersion	和 Visual Basic 项目很像, 它返回 SQL Server 实例的版本详细信息, 格式为 “major.minor.build”
ProductLevel	返回当前运行的 SQL Server 实例的版本: “RTM” = 原始发布版 “SPn” = Service Pack 版 “Bn” = 测试版
ServerName	与指定的 SQL Server 实例关联的 Windows NT 服务器和实例信息

提示:

SERVERPROPERTY 函数对于那些其开发人员需要从一台服务器查看信息的、在多点办公的公司来说是很有用的。

A.12.28 SESSION_USER

SESSION_USER 函数允许在未指定默认值时将系统为当前会话的用户名提供的值插入到表中。其语法如下：

```
SESSION_USER
```

A.12.29 SESSIONPROPERTY

SESSIONPROPERTY 函数返回会话的 SET 选项设置。其语法如下：

```
SESSIONPROPERTY (<option>)
```

如果有在特殊场景中修改会话属性的存储过程，这个函数是很有用的。应该少使用这个函数，因为不能在运行时修改过多 SET 选项。

A.12.30 STATS_DATE

STATS_DATE 函数返回最后一次更新指定索引统计信息的日期。其语法如下：

```
STATS_DATE(<table_id>, <index_id>)
```

A.12.31 SYSTEM_USER

当未指定默认值时，SYSTEM_USER 函数允许将系统为当前系统用户名提供的值插入表中。其语法如下：

```
SYSTEM_USER
```

A.12.32 USER_NAME

USER_NAME 返回数据库用户名称。其语法如下：

```
USER_NAME([<id>])
```

id 参数指定了所需用户名的 ID 号，如果省略 id，则假定为当前用户。

A.13 文本和图像函数

文本和图像函数对文本或图像数据执行操作。它们包括：

- PATINDEX(在“字符串函数”一节中已经介绍过)
- TEXTPTR
- TEXTVALID

A.13.1 TEXTPTR

TEXTPTR 函数检查对应于 text、ntext 或 image 列的文本指针值，并返回 Varbinary 值。在运

行 READTEXT、WRITETEXT 和 UPDATE 语句前，应检查文本指针确保其指向了第一个文本页面。其语法如下：

```
TEXTPTR(<column>)
```

A.13.2 TEXTVALID

TEXTVALID 函数用于检查指定的文本指针是否有效。其语法如下：

```
TEXTVALID('<table.column>', <text_ptr>)
```

table.column 参数指定了要使用的表名和列名。**text_ptr** 参数指定了要检查的文本指针。如果指针无效，则返回 0；如果有效，则返回 1。



分析元数据

在过去的几个发行版中，Microsoft 完成了一项令人惊异的工作，即增加了能够在程序中访问的与服务器和数据库有关的数据类型和数据量。其程度已经到了不适当介绍这部分内容就是笔者失职的地步了。

那么笔者正在这里讨论什么东西呢？好吧，SQL Server 提供了一组函数——标量的和表格式的——来返回与服务器或数据库的当前状态有关的目标信息。这些信息可以从简单的服务器上有哪些对象(这个信息总是可用的，但是在很多情况下是“不支持”访问这个信息的)的问题到具体索引的碎片级别。

本附录将会介绍所谓的“元数据函数”的一些基本信息——有时候也叫做 dm 函数(表示 database metadata)以及系统视图。其中大部分内容往往属于数据库管理员的职责范围，但是重要的是要知道在进行系统调优的时候有哪些信息可用以及在系统状态仪表盘上有哪些信息可用，因为可能会希望把它添加到应用程序的管理面板上(如果有的话)。还可以在程序中使用这些信息来处理维护任务计划(如仅当索引的碎片程度超过一定级别时才进行碎片整理，或者周期性地进行检查和警告)。

注意本附录中介绍的大部分内容是表值函数或系统视图(更主流的系统函数已经在附录 A 中介绍过了)。

B.1 系统视图

正如有一句话是这么说的“早在一天”，SQL Server 在获取元数据的“官方”方法方面提供了非常有限的信息。可以了解一个事务是否处于打开状态，但是却无法查询数据库所包含的表列表(甚至是服务器上的数据库列表)，除非直接访问特殊的“系统表”——Microsoft 会告诉你这种操作是不被“支持的”。

人们对这项功能的需求非常多，以至于开发和管理社区基本上会忽略 Microsoft 的“不支持”的注释而仍然使用它们(在没有这些数据的情况下运行一个系统是非常困难的！)。当进入 SQL Server 7.0 时代时，即 Microsoft 首次在产品中引入信息模式视图(一个 ANSI 构造)的时候，对系统表的使用已经泛滥了。那些没有被接受的内容被添加到了新的系统视图中——很多都是直接映射到旧的系统表上(例如，sys.objects 视图逐列映射到了旧的 sysobjects 表上)。但是事情还没有结束。

他们又添加了大量的各种各样的新的可查询的视图，因此下面看一看其中几个关键视图，读者可能会在开发中用到它们。

提示：

笔者再次强调一下这不是一个完整的列表。相反，这里将主要关注那些更有可能在程序中用到的项目。同样，对视图中的列和表值函数的介绍也主要集中在那些很有可能被用到的部分。

其中几个系统视图提供的是服务器级别的信息。虽然它们中的一些可能会指向某个具体数据库上的信息，但是它们是在服务器级别的上下文中的——这意味着它们能够提供与服务器上任何数据库相关的信息。

1. sys.assemblies

这个视图及其相关的视图(sys.assembly_files 和 sys.assembly_references)能够提供大量的与在 SQL Server 环境中安装和注册的程序集有关的信息。

sys.assemblies 担当着头表的角，它将会为每一个安装在系统上(使用 CREATE ASSEMBLY)的程序集返回一行结果。在 sys.assemblies 目录视图所包含的信息中读者可能感兴趣的如表 B-1 所示：

表 B-1

列	类 型	说 明
name	sysname	数据库中程序集的逻辑名称(不是文件或名称空间的名称)。
principal_id	int	程序集所属的模式 ID。
assembly_id	int	数据库中程序集的唯一 ID。
clr_name	nvarchar(4000)	一个包含几种信息的字符串，包括： 简单的名称 版本号 文化 公共密钥 体系结构 它实际上是 CLR 中程序集的 ID。
permission_set	tinyint	程序集的安全性访问权限的 ID/普通文本指示器。有效的值包括：
permission_set_desc	nvarchar(60)	1 (SAFE_ACCESS) 2 (EXTERNAL_ACCESS) 3 (UNSAFE_ACCESS)
create_date	datetime	程序集被创建的日期和时间。
modify_date	datetime	程序集上次被修改的日期和时间。
is_user_defined	bit	True/False 指示器，指示这个程序集是否由用户创建的(False 表明它是一个系统包含的程序集)

2. sys.columns

可以把这个看成是 sys.objects 的子类，但是它只适用于那些以表格的形式提供结果的对象(表、

视图、不管是用户还是系统创建的返回表值的函数)。它将为对象返回的每一列创建一行数据并提供与该列有关的详细信息。该视图中包含的重要项目如表 B-2 所示：

表 B-2

列	类 型	说 明
object_id	int	该列的父对象的 object_id。可以很容易地把这个 object_id 和 sys.objects 连接起来获取与父对象有关的信息(如名称)
name	sysname	列的名称
column_id	int	列的 ID。它在表中是唯一的
system_type_id	tinyint	该列的数据类型的系统标识符。可以把它与 sys.types 目录视图连接起来以解析出数据类型的常用名称
user_type_id	int	列的用户自定义的数据类型——同样，与 sys.types 目录视图进行连接可以获取一些更有用的信息
max_length	smallint	该列允许的最大长度，以字节计数(记住每个 nchar 和 nvarchar 类型的字符需要占用两个字节！)。对于大多数的支持 blob 的数据类型 (varchar(max)、nvarchar(max)、varbinary(max)、xml)来说，这个值将会是-1，但是对于文本数据类型来说，这个值将会是 16(blob 指针的大小)或者是行选项中文本的值(如果已经提供了改值)
precision	tinyint	基于数值的列的精度(0 表示非数值的)
scale	tinyint	基于数值的列的数值范围(0 表示非数值的)
collation_name	sysname	基于字符的列的排序规则名称(NULL 表示非基于字符的数据)
is_nullable is_ansi_padded is_rowguidcol is_identity is_computed is_filestream is_replicated is_non_sql_subscribed is_merge_published is_dts_replicated	bit	在很大程度上可以从这些列的名称上看出它们的含义。简而言之，它们是一组与列的多个属性有关的 true/false 指示器
is_xml_document	bit	同样，其名称在很大程度上已经说明了含义，但是还是有一点细微的差别，因此需要更加具体地说明该列。简而言之，这个属性指示该列是否不仅仅包括 XML，但是只有当该列为一个完全的 XML 文档而不仅仅是一个 XML 文档的片段时，该属性才有效。即 1 表示一个完整的 XML 文档，0 表示 XML 片段或者非 XML 数据
is_sparse is_column_set	bit	这里把这两个属性单独列出来主要是因为它们都与稀疏列有关，并且除非读者熟悉稀疏列，否则 is_column_set 属性可能没有任何意义(请在其他地方查看更多与稀疏列有关的信息，这里只简单地提供与 is_column_set 有关的上下文信息)
xml_collection_id	int	只同具有类型的 XML 列有关，这个属性给出了 XML 模式集合的 ID 来加强 XML 的类型信息

3. sys.databases

该视图映射到旧的 sysdatabases 对象上，提供如表 B-3 所示的信息(这里只列出了关键列，它不是一个完整的列表)：

表 B-3

列	类 型	说 明
name	sysname	数据库的逻辑名。
database_id	int	数据库的 ID。这个 ID 可以用于很多系统函数中，并且可以作为很多其他的系统视图或元数据函数的外键列
create_date	datetime	数据库被创建的日期和时间。注意当重命名数据库时会重设该值
compatibility_level	tinyint	指示该数据库兼容的 SQL Server 版本。有效的值包括： 70 (SQL Server 7.0) 80 (SQL Server 2000) 90 (SQL Server 2005) 100 (SQL Server 2008) NULL (数据库不是联机的)
collation_name	sysname	数据库使用的排序规则(排序顺序以及大小写敏感性、重音、假名和宽度)
state	tinyint	数据库的当前状态。有效值包括： 0 (Online) 1 (Restoring) 2 (Recovering) 3 (Recovery pending) 4 (Suspect) 5 (Emergency) 6 (Offline) 这些状态的含义可以在 state_desc 列中找到
is_in_standby	bit	True/False 指示器，指示是否可以在数据库中应用事务日志或差异备份
is_ansi_nulls_on is_ansi_padding_on is_ansi_warnings_on is_arithabort_on is_quoted_identifier_on is_fulltext_enabled is_trustworthy_on is_encrypted	bit	这些是独立的 true/false 列，指示给定的数据库设置是否活跃。0 表示设置被关闭了，1 表示设置被启用了

4. sys.database_files

这个视图大概地(我是说非常不精确)映射到旧的 sysfiles 系统表上。可以把它看成是一组

sysfiles, 因为它包含的信息要多得多。这里所说的是什么信息呢? 它们是与数据库中每个物理文件有关的信息。对于大多数数据库来说, 这个信息包含两种(主数据文件和日志), 但是如果使用了文件组(包括分区), 那么它们也会被包含在内。

注意:

sys.database_files 主要关注的是某个具体的数据库。还可以使用 sys.master_files 来获取一个与服务上所有数据库有关的相似列表(除了 database_id 之外, 所有相同的关键字段也会出现)。

当希望在应用程序中支持高级的数据库配置时需要关注这个视图。该视图的一些关键字段如表 B-4 所示:

表 B-4

列	类 型	说 明
file_id	int	数据库中文件的内部 ID
file_guid	uniqueidentifier	文件的 GUID。如果是从 SQL Server 的上一个版本升级而来的, 那么这个值可能为 Null
type	tinyint	指示文件的类型。有效的文件类型包括: 0 (存储真正的数据、索引或全文行) 1 (日志数据) 2 (FILESTREAM 信息) 4 (适用于 2008 之前的版本, 存储全文数据) 注意 type_desc 列将会预先解释这些值(但是要记住的是 FULLTEXT 值只适用于 SQL Server 2005 和之前的版本。2008 的全文回到了 ROWS 上)
data_space_id	int	这个特定文件所属的文件组的 ID
name	sysname	数据库的逻辑名
physical_name	nvarchar(260)	物理文件在操作系统级别的名称
state	tinyint	数据库的当前状态。有效的值包括: 0 (Online) 1 (Restoring) 2 (Recovering) 3 (Recovery pending) 4 (Suspect) 5 (Emergency) 6 (Offline) 这些状态的含义可以在 state_desc 列中找到
size	int	文件在 8KB 的数据页面中真正的大小
max_size	int	文件大小允许增长到的最大值(与在 CREATE DATABASE 语句中文件部分所使用的值对应)
growth	int	是否允许这个文件自动增长以及增长多少。如果值为 0, 那么这个文件不允许增长。大于 0 的值表示文件的生长量或者增长的百分比, 这是由 is_percent_growth 列来决定的

5. sys.identity_columns

这是其中一个对笔者来说“非常有用”的视图。虽然从其性质上看，它是一个处于边缘的视图，但是坦白讲，当需要用到它的时候，拥有这个工具还是非常有帮助的。

那么它提供了哪些信息呢？好吧，正如其名称所蕴含的那样，它提供了数据库中所有标识列的一个列表。通常很少会用到这个视图，但是当需要用到这些信息的时候，sys.identity_columns 就像金子一样发出耀眼的光芒(它使事情变得简单的)。

例如，如果想要知道哪些表拥有标识列、哪些是标识列以及它们当前的增量是什么的话，那么就可以使用下面的代码：

```
USE AdventureWorks2008;

SELECT so.name AS TableName,
       sic.name AS ColumnName,
       CAST(last_value AS bigint)
       + CAST(increment_value AS bigint) AS NextValue
FROM sys.identity_columns sic
JOIN sys.objects so
     on sic.object_id = so.object_id
    AND so.type = 'U';
```

注意需要对值进行强制转换，因为它们是以 sql_variant 数据类型存储的，而本例中使用的数学运算需要显式地进行强制转换。还需要注意这里只返回了与真正的用户表有关的结果。采用这种方式可以解决一个问题，即 sys.identity_columns 将会返回所有标记为标识列的列，而不管该对象的性质，这意味着该视图将基于表中的标识列返回列，从而导致系统表也会被返回(如果它们使用了标识列的话。而其中一些系统表确实会使用标识列)。把结果限制在用户表上之后就能够确保不会与系统对象混淆起来，同时只会看到标识列的根源(即表中的标识列——而不是视图中的标识列，视图可能会显示基表中的列)。

其结果是非常简单的：

TableName	ColumnName	NextValue
SpecialOffer	SpecialOfferID	17
Address	AddressID	32522
AddressType	AddressTypeID	7
ProductModel	ProductModelID	129
...		
...		
...		
ShoppingCartItem	ShoppingCartItemID	6
DatabaseLog	DatabaseLogID	1596
ErrorLog	ErrorLogID	NULL

(47 row(s) affected)

注意：

如果在创建标识列之后还没有插入任何行，那么将不会显示任何 last_value，同时它将返回一

个 NULL。

6. sys.indexes

笔者确信读者已经猜测出这个视图是与索引有关的。它是一种主系统目录视图，用来处理索引，并且有几个子或者扩展视图与之相关联。数据库中的每一个索引都会在这张表中有一个条目，不管它是什么类型的索引。它不会提供与具体列相关的信息(可以在 `sys.index_columns` 中查看这些信息)，同时与特殊索引类型(xml、地理空间的)相关的扩展信息将会被存储在一个特殊的扩展了该表的索引类型特定的表中(它们一对一地映射到 `sys.indexes` 中与特定索引类型有关的行上，同时包括 `sys.indexes` 中所有的列以及一些特定于该类型的索引的列)。

`sys.indexes` 中的关键列如表 B-5 所示：

表 B-5

列	类 型	说 明
object_id	int	该索引所依赖的或者所属的对象的 ID。该列与 <code>sys.objects</code> 表中的 <code>object_id</code> 列是捆绑起来的
name	sysname	索引的逻辑名。null 值意味着这个索引是一个堆
index_id	int	索引的 ID。该列的值是可预测的，表上的堆索引的值为 0，聚集索引的值为 1(并且该值只能是其中一个——不能两个都是)。大于或等于 2 的值表示构建在堆或聚集索引上的非聚集索引
type	tinyint	指示索引的类型。有效的索引类型包括： 0 (堆) 1 (聚集) 2 (非聚集) 3 (XML) 4 (空间) 注意 0 和 1 是互斥的(每张表都必须拥有两个值中的一个，并且只能拥有其中之一)
type_desc	nvarchar(60)	类型列的文本版本(文本，而不是数值)
is_unique	bit	True/False 值，指示索引是否必须唯一
data_space_id	int	存储这个索引的文件组或分区方案的标识符
ignore_dup_key	bit	True/False 指示器，指示该索引上忽略重复关键字的选项是否启用
is_primary_key is_unique_constraint	bit	True/False 指示器，指示这个索引是否支持主键或 unique 约束。注意，两个都不是必需的，并且它们是互斥的(一个索引不能同时具有 unique 约束和主键)
fill_factor	tinyint	索引被定义或上次被修改的时候所使用的填充因子。0 表示使用默认的填充因子
is_padded	bit	True/False 指示器，指示索引最后被创建或修改的时候是否使用了 PAD_INDEX 选项
is_disabled	bit	顾名思义——它是一个 True/False 指示器，指示索引是否被禁用了。记住被禁用的索引将完全不可用并且在重新活跃之前需要重建它。禁用聚集索引将会使整张表脱机，并且需要重新聚集索引才能访问表数据

(续表)

列	类 型	说 明
is_hypothetical	bit	True/False 指示器, 指示它实际上仅仅是一个统计信息条目, 而不是一个真正的索引(一个索引的所有统计信息, 但是不包括任何经过排序的数据存储)
allow_row_locks allow_page_locks	bit	读者可能已经从其名字中猜测到了, 它们是 True/False 指示器, 指示在访问索引时是否允许使用行或页锁。其影响只局限于该索引(它不会影响到表), 除非它是一个堆或聚集索引。这些列不是互斥的。把行和页锁都禁用的后果是任何使用这个索引的查询都会产生一个表锁
has_filter	bit	True/False 指示器, 指示这是否是一个经过筛选的索引(绝大多数的索引和 SQL Server 2008 之前的索引都是未经过筛选的索引)
filter_definition	nvarchar(max)	与 WHERE 子句的效果一样, 这个值指出了对参与到索引中的值进行筛选所使用的表达式

读者应该能够从这个冗长的列表中分辨出哪些列是非常有用的, 在确定当前系统状态(用于性能和用途分析)和编写维护脚本方面, 这个系统目录视图是非常有用的。

提示:

考虑把这张表中的值与“缺失索引”相关的一组系统目录视图中的值连接起来。

7. sys.index_columns

sys.indexes 包含的是索引的头部信息, 而这个视图包含的是索引定义中各个列的细节信息。任何参与到索引中的列都会在这个视图中有一个条目与之对应, 而不管它是以何种方式参与到索引中的(有一个指示器会指示一个列是参与到了真正的关键字中, 还是仅仅是一个被包含的列)。一般来讲, 如果不把这个系统目录视图与它的父视图(sys.indexes)连接起来, 那么它实际上没有任何用处。实际上, sys.index_columns 中的所有信息都是与具体列有关的。如表 B-6 所示:

表 B-6

列	类 型	说 明
object_id	int	包含该列所参与的索引的对象的 ID。该列与 sys.objects 表中的 object_id 是捆绑在一起的
index_id	int	与该列相关联的索引的 ID。注意, 当与索引头信息进行连接时需要这个列和 object_id
index_column_id	int	参与这个特定索引的列的 ID
column_id	int	属于某个对象的列的 ID, 包含该列的索引是以这个对象为基础的。如果索引是建立在视图上的, 那么可能会看到重复的 column_id 值, 这是因为列只在某个具体对象的上下文中是唯一的。0 表示该列是一个内部列(显然不会出现在原始对象中), 它保存着堆上非聚集索引的行标识符(RID)
key_ordinal	tinyint	以 1 开始的序数值, 指示该列在索引关键字定义中的位置(不包括被包含的列)。0 值表示该列不属于任何关键字
partition_ordinal	tinyint	在一组分区列中以 1 开始的序数值(一般来讲, 这个值为 0, 表示它不是一个分区列)

(续表)

列	类 型	说 明
is_descending_key	bit	True/False 指示器, 指示是否以降序对待该列
is_included_column	bit	True/False 指示器, 指示该列是否仅仅是一个被包含的列(只存在于索引的叶子级别, 同时在排序的时候不会考虑该列)

8. sys.objects

注意:

可以把 sys.objects 看成是其他几个系统目录视图的父视图。实际上, 在 sys.objects 视图可以确定很多与任何对象的存在有关的信息, 但是需要知道的是, 有一些视图会为几种对象类型(视图、表、约束、过程等等)提供特定于对象类型的信息。

与 sys.database_files 大概地映射到旧的 sysfiles 系统表上类似, sys.objects 映射到旧的 sysobjects 系统表上(这次是相当精确的)。

sys.objects 能够提供大量的信息, 如数据库中存在哪些对象以及那些对象的性质是什么等信息。sys.objects 中的一些关键列如表 B-7 所示:

注意:

sys.objects 局限于存在于数据库中某个模式的那些对象的范围。虽然它包含了数据库中大多数的相关对象, 但是值得指出的是, 这意味着 DDL 触发器将不会出现在 sys.objects 中。

表 B-7

列	类 型	说 明
name	sysname	对象的逻辑名
object_id	int	对象数据库的 ID。这个 ID 在数据库中是唯一的
principal_id	int	如果该对象的拥有者与包含该对象的模式的拥有者不同, 则为该对象的拥有者的 ID(如果对象的拥有者和模式的拥有者是相同的, 那么该列的值为 NULL)。注意, 如果对象的类型是某张表的子对象(任何类型为约束或触发器的对象)或者旧式的 DEFAULT 或 RULE 对象(在产品中支持它们只是为了向后兼容)中的一个, 那么该列将不适用(将会被设置为 NULL)
schema_id	int	包含这个对象的模式的 ID(如果需要模式的名称, 那么与 sys.schemas 进行连接)
parent_object_id	int	当前对象的父对象的 object_id(如某个触发器或约束的父表)
type	char(2)	由 1 个或 2 个字母组成的缩写来表示该对象的类型(一个用普通文本描述对象类型的版本存储在该列的姊妹列中——type_desc)。有效的值包括: AF(CLR 聚合函数) C(Check 约束) D(约束或独立默认值) F(外键约束) FN(SQL 标量函数) FS(CLR 标量函数)

(续表)

列	类 型	说 明
type	char(2)	FT(CLR 表值函数) IF(内联表值函数) IT(内部表) P(存储过程) PC(CLR 存储过程) PG(计划指南) PK(主键) R(规则) RF(复制筛选器) S(系统表) SN(同义字) SQ(服务队列) TA(CLR DML 触发器) TF(表值函数) TR(非 CLR DML 触发器) U(非系统表) UQ(Unique 约束) V(非系统视图) X(扩展存储过程)
type_desc	nvarchar(60)	描述对象类型的一个相对简单明了的版本(比 type 列中的缩写要长一点)
create_date	datetime	对象被创建的日期和时间。注意, 与 sys.databases 视图中的数据库不同, 当重命名对象时不会重设 sys.objects 中显示的创建时间
modify_date	datetime	对象被修改的日期和时间。修改底层的聚集索引(如果这个对象是一张表或者是创建了索引的视图)将会影响这个时间戳, 而修改非聚集索引将不会影响这个时间戳
is_ms_shipped	bit	它是一个标志, 指示它是否是一个系统对象(它是由 SQL Server 提供的对象, 还是由用户或应用程序创建的对象)

9. sys.partitions

正如读者可能猜测的那样, 这个视图是与数据库中定义的分区有关的。不管读者有没有意识到这一点, 每一张表都至少有一个分区。即使没有显式地创建分区方案, 每一张表仍然会有一个分区(它包含了表中的所有内容)。这里关键的事情是使用这个系统视图要能够知道是否已经显式地对表进行了分区以及有多少个分区。从开发的角度来看, 其中一些重要且有用的列如表 B-8 所示:

表 B-8

列	类 型	说 明
partition_id	bigint	数据库中分区的内部 ID(注意这里说的是数据库——不是表)
object_id	int	与该分区关联的具体对象(通过 object_id 来引用该对象)
index_id	int	以 1 开始计数的分区号, 指示该分区在单个表中属于哪个分区
rows	bigint	分区中大概包含的行数。注意这个数值是通过统计信息来维护的, 因此可能并不准确
data_compression	int	笔者不清楚 Microsoft 为什么不使用 tinyint 类型。该列指示在这个分区中如何处理压缩: 0 (无压缩) 1 (行级压缩) 2 (页级压缩) 与在其他系统视图中看到的一些指示器一样, 也有一个用普通文本描述该设置的列(名为 data_compression_desc)

10. sys.partition_functions

一般来讲, 只有在应用程序支持非常高的伸缩性模型时该视图才有意义。通过这个系统视图能够了解到所使用的分区函数的数量和性质。使用这个视图意味着正在做一些与表 B-9 分区有关的紧急事情。

表 B-9

列	类 型	说 明
name	sysname	正如读者可能猜测的那样, 这是分区函数的名称。它在数据库中必须是唯一的
function_id	int	函数的内部标识符。同样, 它在数据库中必须是唯一的
fanout	int	指示该函数在逻辑上创建了多少个分区
boundary_value_on_right	bit	SQL Server 2008 中的表分区延续了只根据范围来分区的思想。该类是一个 True/False 指示器, 指示一个范围的硬边界是否在右边(如果为 false, 那么边界必须在左边)

11. sys.schemas

这个视图简单得难以置信。由于模式在很大程度上就是极其简单的容器, 因此对它不需要了解很多信息。这张表的主要目的是解析拥有某个对象的模式的名称和提供模式拥有者的身份。它只提供了三个列如表 B-10 所示:

表 B-10

列	类 型	说 明
name	sysname	模式的名称(这没什么奇怪的)
schema_id	int	模式的内部标识符, 它在数据库中是唯一的
principal_id	int	拥有这个模式的安全主名(在 sys.user_token 可以找到)的 id

12. sys.servers

本地服务器和每个链接服务器以及注册到这个服务器实例上的远程服务器都会在这个视图中有一行数据与之对应。对于每个服务器, 该视图会提供与通信属性和同服务器交互所需的其他几个属性有关的信息。如表 B-11 所示:

表 B-11

列	类 型	说 明
server_id	int	服务器的内部 ID。本地服务器的 ID 为 0, 而所有其他注册的服务器(链接服务器或远程服务器)的 ID 都将大于 0, 并且它们在这个特定的本地服务器实例中都是唯一的
name	sysname	正如读者可能猜测的那样, 这是分区函数的名称。它在数据库中必须是唯一的
product	sysname	等同于产品属性, 在 OLE DB 连接中将会指定这个属性。如果这是一个本地服务器或另一个 SQL Server, 那么该列的值将为 SQL Server
provider	sysname	OLE DB 提供者的名称——如 SQLNCLI(SQL 本地客户端)、MSDASQL(用于 ODBC 的 OLE DB 提供者)、MSDAORA(Oracle)、Microsoft.Jet.OLEDB.4.0 (Access)
data_source	nvarchar(4000)	OLE DB 使用的数据源。这个值会因为提供者的不同而不同, 对于 SQL Server 来讲, 这个值将会是连接的服务器的名称或 IP 地址
location	nvarchar(4000)	OLE DB 连接中使用的位置。这个值通常为 null(同样, 这取决于提供者)
provider_string	nvarchar(4000)	OLE DB 提供者的字符串属性。虽然在其他设置中通常只需要把连接设置成使用这个属性, 但是在 SQL Server 中只有通过 ALTER 语句才能看到这个属性已经被填充了(必须一开始就使用之前介绍过的离散的属性)
catalog	sysname	OLE DB 目录属性。对于 SQL Server 连接来说, 这等同于为链接服务器所使用的连接设置默认数据库
connect_timeout	int	在连接被自动关闭之前能够处于空闲状态的时间(默认值为 0, 这意味着没有超时时间)
query_timeout	int	查询被中断之前允许运行的时间。同样, 它的默认值为 0, 意味着没有超时时间
is_linked	bit	True/False 指示器, 指示该服务器是否为链接服务器或者其他一些形式的远程服务器连接
is_data_access_enabled	bit	True/False 指示器, 指示在这个连接上是否允许运行分布式查询

(续表)

列	类 型	说 明
is_collation_compatible	bit	True/False 指示器, 指示链接服务器是否使用了一个与本地服务器兼容的数据排序规则。如果它们使用了兼容的排序规则, 那么对分布式查询的性能将会有重大的提升, 因为任何排序规则转换就将被忽略
uses_remote_collation	bit	True/False 指示器, 指示是否使用远程服务器上的排序规则来代替本地的排序规则(假设使用了不兼容的排序规则)
collation_name	sysname	如果没有使用远程排序规则, 那么使用什么排序规则呢? 默认为 NULL, 这意味着希望使用本地的排序规则

13. sys.spatial_indexes

与 sys.xml_indexes 一样, 这个视图是对 sys.indexes 目录视图的一个简单扩展。它包含 sys.indexes(这样就不需要与 sys.indexes 视图进行连接了)中所有的列, 同时还返回三个附加的特定于空间数据的列(由于这三个都是与同一个对象有关的, 因此笔者不清楚你为什么要一起使用它们)并且只筛选出那些空间索引。

附加的三个列如表 B-12 所示:

表 B-12

列	类 型	说 明
spatial_index_type	tinyint	空间索引的类型: 1 (几何的) 2 (地理的)
spatial_index_type	nvarchar(60)	与基本的类型列等同的清晰明了的文本描述。有效的值包括: GEOMETRY(等同于类型 1) GEOGRAPHY(等同于类型 2)
tessellation_scheme	sysname	在数据上创建索引所使用的布局方案的名称。它与所使用的空间索引的类型是相关联的。有效的值包括: GEOMETRY_GRID GEOGRAPHY_GRID

14. sys.synonyms

这个视图是对 sys.objects 目录视图的一个扩展和筛选。它包含 sys.objects 中所有的列, 并且添加了一个名为 base_object_name 的附加列, 该列是作为指针来使用的(使用完全引用名称形式, 最多有 1035 个字符), 指向以这个同义字为别名的基对象。

15. sys.user_token

每一个拥有数据库访问权限的安全主名都会在这个视图中有一行数据与之对应。注意这并不等于每一个拥有访问权限的帐号都会有一个令牌(访问权限可能是通过一个角色来授予的, 而那个角色拥有这个令牌, 因此能够向很多登录帐号授予这种访问权限)。

其包含的列包括表 B-13 所示:

表 B-13

列	类 型	说 明
principal_id	int	数据库中主名的唯一的内部 ID。
sid	varbinary(85)	主名的外部安全标识符。如果是一个 Windows 用户或者组, 那么该值将会是 Windows SID。如果是一个 SQL Server 登录帐号, 那么该值将会是生成登录帐号时由 SQL Server 创建的 SID
name	nvarchar(128)	主名的名称, 在数据库中将会使用这个名称。注意这个值可能与主名的外部名称不同。它在数据库中必须是唯一的
type	nvarchar(128)	一段简单明了的文本, 指示主名所属的类型。有效的值包括: APPLICATION ROLE ASYMMETRIC KEY CERTIFICATE DATABASE ROLE ROLE SQL USER USER MAPPED TO ASYMMETRIC KEY USER MAPPED TO CERTIFICATE WINDOWS LOGIN WINDOWS GROUP
usage	nvarchar(128)	指示针对 GRANT/DENY 权限是检查主名呢还是仅仅把主名当成一个认证者(在加密中会使用它来获取上下文信息)

16. sys.xml_indexes

这个视图是对 sys.indexes 目录视图的一个相对简单的扩展。它包含 sys.indexes(因此不需要与这个视图进行连接)中所有的列, 同时返回三个附加的特定于 XML 索引的列并且只返回那些索引类型为 XML 的行。

三个附加的列如表 B-14 所示:

表 B-14

列	类 型	说 明
using_xml_index_id	int	如果这个值为 NULL, 那么该索引是表的主 XML 索引(记住在表上创建的第一个 XML 索引必须要被标记为主 XML 索引, 并且一张表只能拥有一个主 XML 索引)。任何非 null 值表明这是一个从属索引, 并且提供了在该表的 XML 索引组中唯一的 ID。
secondary_type	char(1)	只与非主索引相关(如果是主索引, 那么该值将会为 NULL), 该值指示了从属索引的性质: P(路径) V(值) R(属性)
secondary_type_desc	nvarchar(60)	secondary_type 列的清晰明了的文本描述

B.2 动态管理视图

这些在 SQL Server 2005 中开始大规模出现的视图确实“非常酷”。与系统视图类似，它们也提供与服务器和/或数据库状态有关的信息。

注意：

与系统视图不同，系统视图在性质上是相对稳定的，但是 Microsoft 并没有承诺动态管理——或 dm——视图也会在不同的发布之间保持稳定。Microsoft 认为这些视图是特定于 SQL Server 每个发行版的实现选择的，因此保留了在不同发布之间修改它们的权利。

dm 视图中大多数的核心列可能相对来说是安全的，但是一个列越具体，那么它在不同的发布之间被修改的可能性就越大。

动态管理视图是如此强大以至于无法忽视它，但是要知道的是使用它们可能会不得不运行一些特定于版本的代码，因此在 SQL Server 的不同版本之间进行迁移时需要做相应的处理。

不同的动态管理视图返回的数据有很大的不同。它们可能会与系统视图一样返回相对静态的数据(那些每天都相当稳定的数据)。它们也可能会返回一直变化的数据(如锁状态、资源管理信息以及其他持续变化的信息)。

提示：

还有一点需要指出的是，所有通常被称为 dm 视图的对象不都是视图，其中几个实际上是表值函数。虽然如此，整个返回动态管理信息的一组对象通常被称为 dm 视图。

B.2.1 索引相关的 dm 视图和函数

与之前所采用的获取索引信息的方式相比，使用 dm 视图来获取索引信息是一个飞跃。在 SQL Server 2000 和之前的发行版中，获取索引信息所采用的方式是使用 DBCC SHOWCONTIG 或者其他数据库一致性检查器命令。虽然这些功能可以提供大多数所需的信息，但是信息的格式是一种报表格式，并且很难在程序中使用这些信息。而 dm 视图返回一个表格式的结果集，并且可以很容易地在结果集上应用 WHERE 或其他 SQL 构造——非常强大。但是 SQL Server 团队并没有就此停滞不前。现在在 SQL Server 2008 中能够获取以前甚至无法获取的索引相关的信息！

正如笔者在第 22 章中提到的那样，现在 SQL Server 能够在进行查询优化的时候分析哪些可用的索引可能是有用的。它会把这些分析信息通过日志记录下来并通过各种元数据源(包括接下来将要介绍的一些与索引相关的 dm 视图)来把这些信息呈现出来。

1. sys.dm_db_index_physical_stats

这是一个获取索引状态的新的信息源。使用这个表值函数能够获取大量的信息，如碎片、索引的深度以及参与索引的记录数等等。

这个函数有一个标准的函数参数格式，其中包括几个必需的参数。其语法如表 B-15 所示：

```
sys.dm_db_index_physical_stats (
    { <database id> | NULL | 0 | DEFAULT },
    { <object id> | NULL | 0 | DEFAULT },
```



```

{ <index id> | NULL | 0 | -1 | DEFAULT },
{ <partition number> | NULL | 0 | DEFAULT },
{ <mode> | NULL | DEFAULT }
)

```

表 B-15

参 数	数 据 类 型	说 明
database id	smallint	索引统计信息所属的数据库的内部标识符。传入 NULL、0 或者 DEFAULT 时函数返回的结果是一样的，即为所有数据库返回数据
object id	int	索引统计信息所属的对象的内部标识符。同样，传入 NULL、0 和 DEFAULT 的效果是一样的(返回所有对象)。使用 OBJECT_ID() 函数来通过对象的名称获取相应的 ID
index id	int	统计信息所属的索引的内部标识符。0 表示只需要堆(基数据页)的统计信息，而不需要表中其他索引的统计信息。同样，任何正数将会与特定表或视图的以 1 开始的索引 ID 进行匹配，而表或视图是由对象 ID 参数来指定的。NULL、-1(注意不是 0)和 DEFAULT 的效果是一样的(返回所有索引)
partition number	int	索引信息所属的内部分区号(以 1 开始)。DEFAULT、NULL 和 0 在功能上是一样的，表明希望获取所有分区的信息。任何大于 0 的整数将会与具体的分区号进行匹配
mode	sysname	指示创建你所收到的统计信息所使用的扫描级别。有效的输入包括 DEFAULT、NULL、LIMITED、SAMPLED 和 DETAILED。其中 DEFAULT、NULL 和 LIMITED 在功能上是一样的

读者在本书中已经见过好几个使用该表值函数的例子了，例如，可以执行：

```

USE AdventureWorks2008;

SELECT OBJECT_NAME(object_id) AS ObjectName,
       index_type_desc,
       avg_fragmentation_in_percent
FROM sys.dm_db_index_physical_stats(
       DB_ID('AdventureWorks2008'),
       DEFAULT,
       DEFAULT,
       DEFAULT,
       DEFAULT)
WHERE avg_fragmentation_in_percent > 50;

```

注意，与所有的表值函数一样，可以指定一个明确的选择列表以只选取那些对需求有意义的列。当然，也可以把返回结果与其他表或视图进行连接以获取额外的信息或隐藏在结果中的信息(例如，可以把结果与 sys.indexes 系统视图连接起来以检索出索引的名称)。

运行上面的代码将会返回索引碎片高于 50% 的表或视图列表：

ObjectName	index_type_desc	avg_fragmentation_in_percent
ProductListPriceHistory	CLUSTERED IND	66.6666666666667
SpecialOfferProduct	CLUSTERED IND	66.6666666666667
ProductReview	NONCLUSTERED	66.6666666666667
Employee	NONCLUSTERED	66.6666666666667

```

Product          NONCLUSTERED      66.6666666666667
ProductCostHistory CLUSTERED IND    66.6666666666667
ProductDescription NONCLUSTERED      66.6666666666667
DatabaseLog      NONCLUSTERED      66.6666666666667

```

(8 row(s) affected)

实际上这个视图中包含的列比结果中显示的列要多得多。其中一些有趣的列如表 B-16 所示：

表 B-16

列 名	数 据 类 型	说 明
database_id	smallint	表或视图的内部标识符。可以使用 DB_NAME 函数来检索数据库的名称
object_id	int	索引所属的表或视图的对象 ID。使用 OBJECT_NAME 函数来返回与 ID 关联的名称
index_id	int	列出的索引的索引标识符。注意这个值只在给定的表或视图中是唯一的。0 表示这是非聚集表的堆，1 表示这是表的聚集索引
partition_number	int	所属对象中以 1 开始的分区号，对象可以是表、视图或索引。1 表示没有分区的索引或堆
index_type_desc	nvarchar(60)	索引类型的说明： HEAP CLUSTERED INDEX NONCLUSTERED INDEX PRIMARY XML INDEX SPATIAL INDEX XML INDEX
alloc_unit_type_desc	nvarchar(60)	分配单元类型的说明： IN_ROW_DATA LOB_DATA ROW_OVERFLOW_DATA LOB_DATA 分配单元包含的是存储在类型为 text、ntext、image、varchar(max)、nvarchar(max)、varbinary(max)和 xml 的列中的数据。 ROW_OVERFLOW_DATA 分配单元包含的是存储在类型为 varchar(n)、nvarchar(n)、varbinary(n)和 sql_variant 的列中并且已经超出行范围的数据
index_depth	tinyint	索引级别数。 1=Heap 或 LOB_DATA 或 ROW_OVERFLOW_DATA 分配单元
index_level	tinyint	索引的当前级别。 0 表示索引页级、堆和 LOB_DATA 或 ROW_OVERFLOW_DATA 分配单元。 大于 0 的值表示非叶级索引。根级索引的 index_level 值是最大的。 只有当 mode=DETAILED 时才会处理非叶级索引。

(续表)

列 名	数 据 类 型	说 明
avg_fragmentation_in_percent	float	索引的逻辑碎片或 IN_ROW_DATA 分配单元中堆的区域碎片。这个值是以百分比计的并且会考虑多个文件。0 用于 LOB_DATA 和 ROW_OVERFLOW_DATA 分配单元。当 mode=SAMPLED 时 NULL 用于堆。
fragment_count	bigint	一个 IN_ROW_DATA 分配单元中叶级的碎片数。NULL 用于非叶级索引和 LOB_DATA 或 ROW_OVERFLOW_DATA 分配单元。当 mode=SAMPLED 时 NULL 用于堆。
avg_fragment_size_in_pages	float	一个 IN_ROW_DATA 分配单元的叶级中一个碎片包含的平均页面数。 NULL 用于非叶级索引和 LOB_DATA 或 ROW_OVERFLOW_DATA 分配单元。当 mode=SAMPLED 时 NULL 用于堆。
page_count	bigint	索引或数据页面的总数。 对于索引来讲, 这个值表示 IN_ROW_DATA 分配单元中当前级别的 B 树所包含的索引页面总数。 对于堆来讲, 这个值表示 IN_ROW_DATA 分配单元中的数据页面总数。 对于 LOB_DATA 或 ROW_OVERFLOW_DATA 分配单元来讲, 这个值表示分配单元中的数据页面数。
avg_page_space_used_in_percent	float	所有页面中可用数据存储空间的平均百分比。 对于索引来讲, 平均适用于 IN_ROW_DATA 分配单元中当前级别的 B 树。 对于堆来讲, 这个值表示 IN_ROW_DATA 分配单元中所有数据页面的平均百分比。 对于 LOB_DATA 或 ROW_OVERFLOW_DATA 分配单元来讲, 这个值表示分配单元中所有页面的平均百分比。 当 mode=LIMITED 时, 其值为 NULL。
record_count	bigint	记录总数。 对于索引来讲, 这个值表示 IN_ROW_DATA 分配单元中当前级别的 B 树所包含的记录总数。 对于堆来讲, 这个值表示 IN_ROW_DATA 分配单元所包含的记录总数。 对于堆来讲, 这个函数返回的记录数可能与在堆上运行 SELECT COUNT(*) 返回的行数不匹配。这是因为一行可能包含多个记录。例如, 在一些更新情形中, 单个堆行可能会因为更新操作而包含一个前向记录和一个后向记录。还有, 大多数大型 LOB 行在 LOB_DATA 存储中会被分割成多个记录。 对于 LOB_DATA 或 ROW_OVERFLOW_DATA 分配单元来说, 这个值表示整个分配单元包含的记录总数。当 mode=LIMITED 时, 其值为 NULL
ghost_record_count	bigint	分配单元中可以被 ghost 清理任务删除的 ghost 记录数。 0 用于 IN_ROW_DATA 分配单元中非叶级索引。 当 mode=LIMITED 时, 其值为 NULL

(续表)

列 名	数 据 类 型	说 明
version_ghost_record_count	bigint	分配单元中由未完成的快照隔离事务保留的 ghost 记录数。 0 用于 IN_ROW_DATA 分配单元中非叶级索引。 当 mode=LIMITED 时, 其值为 NULL
min_record_size_in_bytes	int	记录最小的字节数。 对于索引来讲, 这个值表示 IN_ROW_DATA 分配单元中当前级别的 B 树中记录大小的最小值。 对于堆来讲, 这个值表示 IN_ROW_DATA 分配单元中记录大小的最小值。 对于 LOB_DATA 或 ROW_OVERFLOW_DATA 分配单元来讲, 这个值表示整个分配单元中记录大小的最小值。 当 mode=LIMITED 时, 其值为 NULL
max_record_size_in_bytes	int	记录最大的字节数。 对于索引来讲, 这个值表示 IN_ROW_DATA 分配单元中当前级别的 B 树中记录大小的最大值。 对于堆来讲, 这个值表示 IN_ROW_DATA 分配单元中记录大小的最大值。 对于 LOB_DATA 或 ROW_OVERFLOW_DATA 分配单元来讲, 这个值表示整个分配单元中记录大小的最大值。 当 mode=LIMITED 时, 其值为 NULL
avg_record_size_in_bytes	float	记录大小的平均字节数。 对于索引来讲, 这个值表示 IN_ROW_DATA 分配单元中当前级别的 B 树中记录大小的平均值。 对于堆来讲, 这个值表示 IN_ROW_DATA 分配单元中记录大小的平均值。 对于 LOB_DATA 或 ROW_OVERFLOW_DATA 分配单元来讲, 这个值表示整个分配单元中记录大小的平均值。 当 mode=LIMITED 时, 其值为 NULL
forwarded_record_count	bigint	一个堆中拥有前向指针指向另一个数据位置的记录数。(在更新过程中当原先的位置没有足够的空间来存储新行时会进入这个状态。) NULL 用于除堆中的 IN_ROW_DATA 分配单元之外的任何分配单元。 当 mode=LIMITED 时, NULL 用于堆
compressed_page_count	bigint	经过压缩的页面数。 对于堆来讲, 新分配的页面不是 PAGE 压缩的。堆只有在两种特殊情况下才会进行 PAGE 压缩: 当数据是大容量导入时或者当堆被重新构建时。典型的引起页面分配的 DML 操作不会进行 PAGE 压缩。当 compressed_page_count 的值超过预先设置的阈值时可以重新构建堆。对于拥有聚集索引的表来讲, compressed_page_count 值指示 PAGE 压缩的有效性

2. sys.dm_db_index_usage_stats

这是一个精妙的视图，可能还没有引起世界上的 SQL Server 专家的足够注意。它是什么呢？好吧，这个视图中包含的是在一个索引使用另一个索引(包括一个原始堆)时 SQL Server 维护的一系列计数器的当前状态。从这个视图中可以获取可靠的信息，如在系统的正常运作过程中哪些索引被用到了，哪些索引没有被用到，以及可能同样重要的，系统是如何使用这些索引的。还可以获取特定于操作的信息，如扫描、搜寻和查找以及变更发生的频率是多少。甚至还可以获得索引最近被使用的日期和时间信息(可能在一段时间内会一直使用某个索引，但是新索引或者发生在数据上的变更可能会导致在相当长的一段时间内不会用到这个索引)。这些信息对于弄清楚是否值得创建索引是很关键的。

提示：

注意在这个视图上发生的任何被称为“更新”的操作不仅仅包括 UPDATE 语句——任何修改行的操作(包括插入或删除，不管它们是否由显式的 INSERT 或 DELETE 语句引起还是由 MERGE 语句引起)都是更新操作。

每次重启 SQL Server 服务都会重设 sys.dm_db_index_usage_stats 所使用的行计数器。每次数据库脱机时会重设数据库计数器，而不管造成数据库脱机的原因是什么(分离数据库或者因为其他一些原因需要关闭数据库，如 AUTO_CLOSE，在 SQL Server Express 安装中经常使用这个选项)。

使用这个视图是相当简单的，就把它当成表来使用：

```
SELECT *
FROM sys.dm_db_index_usage_stats;
```

读者可以看到，这里并没有什么高深的学问。这个视图提供了四种使用方式中每种方式被用到的次数和上一次被用到的日期并且把系统和用户访问分别放到了各自的桶中(根据具体应用程序的需求来聚合它们)。同样，每种方式都包含了一个日期并且用户访问和系统访问也被分开了。如表 B-17 所示：

表 B-17

列	类 型	说 明
database_id	smallint	数据库的内部标识符，该数据库包含了用来进行报表的对象。记住可以使用 DB_NAME()函数来把 ID 转换为名称。同样，可以使用 DB_ID()函数来把名称转化为 ID，可以在 WHERE 子句中使用这个 ID
object_id	int	对象的内部标识符，该对象将会被索引用到。同样，使用 OBJECT_ID()函数来把名称转换为 ID 以在 WHERE 子句中使用这个 ID。使用 OBJECT_NAME()函数来把 ID 转换成一个更加用户友好的名称以在选择列表中使用这个名称。还可以把它与 sys.objects 表连接起来以获取与特定对象有关的更多信息
index_id	int	与使用信息有关的索引的内部标识符。如果希望仅仅获取堆(非聚集索引中的基数据页)的统计信息，那么使用 0。如果希望获取聚集表上的聚集索引的统计信息，那么使用 1。 如果希望获取更多与索引的性质有关的信息(哪些列被用到了或被包含了、索引的类型、名称等等)，那么把它与 sys.indexes 表进行连接

(续表)

列	类 型	说 明
user_seeks system_seeks	bigint	索引搜寻的次数。这是 SQL Server 跟随索引到某个具体行的一个实例
user_scans system_scans	bigint	这可以是对索引的完整扫描(与扫描整张表类似, 但是只扫描包含在索引中的数据)或者区域扫描(搜寻到索引中的某个特定的位置, 然后读取之后所有的行, 直到到达某个特定的结束位置)
user_lookups system_lookups	bigint	系统使用一个行标识符(RID)或聚集键来“查找”某个具体的行——把它看成另一种形式的搜寻
user_updates system_updates	bigint	当索引行上发生某种形式的变更时该值将会增加。尽管它叫做“更新”, 但实际上数据上发生的任何变更都会使这个计数器增加, 不管是插入、更新还是删除
last_user_seek last_system_seek	datetime	用户或系统(由名称指示)上次引起在索引上执行一个搜寻操作的日期
last_user_scan last_system_scan	datetime	用户或系统(由名称指示)上次引起在索引上执行一个扫描操作的日期
last_user_lookup last_system_lookup	datetime	用户或系统(由名称指示)上次引起在索引上执行一个查找操作的日期
last_user_update last_system_update	datetime	用户或系统(由名称指示)上次修改与该索引相关联的数据的日期

3. sys.dm_db_missing_index_*视图族

要完整地讨论这一族视图是非常复杂的并且可能需要一个专门的附录来介绍它们。

读者可能已经回忆起笔者在几个不同的章节中提到过, 现在 SQL Server 2008 不仅能够重新组织可用的索引, 而且能够识别出那些目前并不存在但如果存在就能提供有用的帮助的索引。除了那些能够给出很多数据的管理工具之外, 还可以通过使用四个缺失索引视图来在程序中访问缺失索引信息。这些视图是:

- sys.dm_db_missing_index_groups
- sys.dm_db_missing_index_group_stats
- sys.dm_db_missing_index_details
- sys.dm_db_missing_index_columns

理解这些系统视图的关键之处是要知道它们共同提供下列信息:

- 哪些索引是一旦存在就会有帮助的?
- 哪些列应该包含在这些索引中?
- 查询使用这些索引之后会取得多少编译/优化?
- 这个索引将会被如何使用?

存储在这些视图中的信息在本质上是临时的, 并且可以认为它们的工作方式与 tempdb 数据库有很大的相似之处, 即 SQL Server 服务每次重启的时候, 它们将会被完全删除并且所有的信息都会从 0 开始。



基础知识

正如笔者之前在本书的导论部分中所说的那样，我主张分别介绍 SQL Server 的入门级别的知识 and 高级知识。我希望通过折衷方式将笔者前两本书中提及的完整的纲要都介绍完全(以免内容太多以至于无法在一本书中完全介绍完)。也就是说，笔者意识到很多高级开发人员不愿意去购买介绍入门知识的书籍，但可能仍然希望能够方便地查阅、回顾或学习一些在非基本上下文中使用的扩展语法。如果读者是那些人中的一个，那么这个附录就是为你而设计的。其想法是非常简单的：提供语法信息并且在一些情况下(其他情况下则不会)给出使用这些语句的示例。这里将不会长篇累牍地介绍关键概念，并且不会对某些概念进行介绍或者只使用相对较少的篇幅来介绍某些概念。这里将把介绍入门知识的书籍使用几百页来介绍的内容压缩成几十页来介绍。

这里将会对其中几个概念做一点点“额外的”介绍(仍然相当少)。这类介绍一般只局限于那些在最近的发行版中才被加入的命令的入门知识，或者那些已经发生较大变更的内容(这些变更将会导致目前做出的选择与采用以前的知识所做出的选择有所不同)。

C.1 查询的基础知识

本节将会看一看直接与 DML(或“数据操作语言”)相关联的各种主题。本节中介绍的所有内容都会通过某些形式直接捆绑到下列四种语句中某种语句的执行上：

- **SELECT**：读取数据的语句(尽管在一两种罕见的情况下也把它用作数据插入的一部分)。
- **INSERT**：把数据加入数据库的语句。
- **UPDATE**：修改现有数据的值。
- **DELETE**：从系统中删除数据。

虽然本节的讨论主要使用这四种语句，但是要知道它们仅仅是顶级语句。本节还会介绍这四种语句中每种语句所使用的各种谓词、选项以及操作概念。

尽管这里介绍的大多数内容都是比较“陈旧的”——即读者应该已经熟悉这些内容了——但是不要太快地略过这些内容。本节介绍的内容中有几个概念至少的是中级水平的，并且很多关键字的使用相对来说不是那么频繁，因此读者可能没有学习过这些内容或者已经忘记了还有这部分内容。

C.1.1 基本的 SELECT 语句

SELECT 语句以及它所使用的结构形成了能够在 SQL Server 执行的最大一部分命令的基础。一个 SELECT 语句的基本语法规则如下：

```
SELECT <column list>
[FROM <source table(s)> [[AS] <table alias>]
[[{FULL|INNER|LEFT|RIGHT} OUTER|CROSS]] JOIN <next table>
[ON <join condition>] [<additional JOIN clause> ...]]]
[WHERE <restrictive condition>]
[GROUP BY <column name or expression using a column in the SELECT list>]
[HAVING <restrictive condition based on the GROUP BY results>]
[ORDER BY <column list>]
[[FOR XML {RAW|AUTO|EXPLICIT|PATH [(<element>)]}[, XMLDATA][, ELEMENTS][,
BINARY base 64]]
[OPTION (<query hint>, [, ...n])]
[{ UNION [ALL] | EXCEPT | INTERSECT }]
[;]
```

对于那些正在阅读专业级别的书籍的人来说，语法中的大部分都应该是容易理解的。下面通过两个例子来快速看一看语法(一个有 GROUP BY，一个没有)。

这里从一个基本的带 WHERE 子句的多表有序查询开始：

```
SELECT p.Name AS ProductName, soh.AccountNumber, soh.ShipDate
FROM Production.Product p
JOIN Sales.SalesOrderDetail sod
    ON p.ProductID = sod.ProductID
JOIN Sales.SalesOrderHeader soh
    ON soh.SalesOrderID = sod.SalesOrderDetailID
WHERE soh.ShipDate >= '07/01/2001'
    AND soh.ShipDate < '08/01/2001'
ORDER BY AccountNumber, ShipDate;
```

注意，由于省略了可选的 FULL、INNER、OUTER 和 CROSS 关键字，因此将使用默认的连接(INNER)。还可以使用 GROUP BY 子句来编写一个类似的查询(本例中将获取订购了在 7 月份发货的每种产品的不同帐户的帐号，而不考虑订单是何时发出的)：

```
SELECT p.Name AS ProductName,
    COUNT(DISTINCT soh.AccountNumber) AS UniqueAccounts
FROM Production.Product p
JOIN Sales.SalesOrderDetail sod
    ON p.ProductID = sod.ProductID
JOIN Sales.SalesOrderHeader soh
    ON soh.SalesOrderID = sod.SalesOrderDetailID
WHERE soh.ShipDate >= '07/01/2001'
    AND soh.ShipDate < '08/01/2001'
GROUP BY p.Name
ORDER BY p.Name;
```

注意仍然要把 ORDER BY 子句放在最后(GROUP BY 子句插到了 ORDER BY 前面)。

1. WHERE 子句

WHERE 是一个基本的筛选条件，如果某行不满足 WHERE 子句中指定的所有条件，那么该行就不会包含在结果中。下面看一看所有能够在 WHERE 子句中使用的运算符如表 C-1 所示：

表 C-1

运 算 符	使 用 示 例	效 果
=、>、<、>=、<=、 ◇、!=、!>、!<	<列名>=<其他列名> <列名>='Bob'	标准的比较运算符——这些运算符的工作方式与它们在任何程序设计语言中的工作方式几乎是相同的，但是有几点需要注意。第一，构成“大于”、“小于”和“等于”的条件可能会发生变化，这要取决于所选择的排序规则。例如当所选的排序规则大小写不敏感时，“ROMEY”=“romey”。而当排序规则大小写敏感时，“ROMEY”◇“romey”。 第二，!=和◇都表示“不等于”。!<和!>分别表示“不小于”和“不大于”
AND、OR、NOT	<列 1>=<列 2>AND<列 3>>= <列 4> <列 1> != "MyLiteral" OR <列 2>="MyOtherLiteral"	标准的布尔逻辑。可以使用它们把多个条件组合进 WHERE 子句中。NOT 运算符将首先被计算，然后是 AND，接着是 OR。如果希望改变计算顺序，那么就需要使用括号。注意不支持 XOR
BETWEEN	<列 1> BETWEEN 1 AND 5	如果第一个值在第二个和第三个值之间(包括第二个值和第三个值)，那么比较操作就返回 TRUE。它在功能上等于 A>=B AND A <= C。这三个值可以是列名、变量或者字面值
LIKE	<列 1> LIKE "ROM%"	使用%和_字符来通配字符。%表示任何长度的值都可以替换%字符。_字符表示任意一个字符都可以替换_字符。 把字符放在[]符号中表示任何在[]中单个字符都是可以的([a-c]表示 a、b 或 c 都是可以的。[ab]表示 a 或 b 都是可以的)。^的工作方式与 NOT 运算符一样——表示下一个字符就被排除在外
IN	<列 1> IN (一个数字列表) <列 1> IN ("A", "b", "345")	当 IN 关键字的左边的值与 IN 关键字后面的列表中的任何值匹配时，返回 TRUE。经常会在子查询中用到这个运算符，子查询在第三种中介绍
ALL、ANY、SOME	<列 表达式>(比较运算符)<ANY SOME>(子查询)	当子查询中的任意或所有(取决于选择哪个)值满足比较运算符(如<、>、=、>=)的条件时返回 TRUE。ALL 表示该值必须要匹配集合中的所有值。ANY 和 SOME 在功能上是等同的，只要表达式匹配集合中的任意一个值，都会返回 TRUE
EXISTS	EXISTS(子查询)	当子查询至少返回一行结果时返回 TRUE。同样，在第 3 章中已经对此深入介绍过了

2. ORDER BY

顾名思义，指定结果根据哪一列来呈现以及按照何种顺序排序。默认是升序，但是如果希望采用降序，那么也可以提供可选的 DESC 开关。

因此在之前的 SELECT 例子中，只需要简单地加入 DESC 关键字就可以让帐号按照降序显示(注意并没有为 ShipDate 指定任何东西，因此子排序将继续按照升序进行)。

```
ORDER BY AccountNumber DESC, ShipDate;
```

3. GROUP BY 子句

使用了 ORDER BY 之后，本节一开始介绍的 SELECT 语句看起来就稍微有点乱了。GROUP BY 子句用来汇总信息。之前已经介绍过了一个不使用 GROUP BY 的简单查询。本节开头介绍的 GROUP BY 示例将会返回订单的合计，这些订单是根据产品的名称来分组的。使用 GROUP BY 需要注意的关键问题是当使用 GROUP BY 时，在选择列表中指定的任何列都必须是分组所依据的列的一部分(即它们必须出现在 GROUP BY 列表中)，或者必须是聚合函数的目标。

注意：

虽然当把聚合和 GROUP BY 子句一起使用时聚合显示出了它们的威力，但是它们并不局限于分组查询——如果在没有 GROUP BY 的查询中使用聚合，那么它们将会对整个结果集(所有匹配 WHERE 子句的行)进行聚合。这里需要注意的地方是，当不与 GROUP BY 一起使用时，一些聚合只能与其他聚合一起出现在 SELECT 列表中——即它们无法与列名一起出现在 SELECT 列表中，除非有一个 GROUP BY 子句。例如，除非有一个 GROUP BY 子句，否则 AVG 就无法与某个具体的列一起使用，但是可以与 SUM 一起使用。

下面回顾一下其中一些最常用的集合函数(需要知道的是，可以使用基于 CLR 的用户自定义函数来编写自己的聚合函数)。

- AVG：这个函数用于计算平均值。知道这些已经足够了。
- MIN/MAX：正如读者可能猜测的那样，它们用来获取每个分组中所选列的最小值和最大值。
- COUNT(表达式|*)：COUNT(*)函数用来统计查询中的行数。

注意：

除了 COUNT(*)函数之外，所有的聚合函数都将忽略 NULL。这种行为可能会对结果产生重大的影响，因此要小心一点。在计算平均值时很多用户都认为数字字段中的 NULL 值会当成 0 来处理，但是 NULL 并不等于 0，因此不应该这样处理。如果在拥有 NULL 值的列上执行 AVG 或其他聚合函数，那么 NULL 值将不会成为聚合的一部分，除非在函数中把它们变成非 NULL 值(如使用 COALESCE()或 ISNULL())。

4. HAVING 子句

只有当查询中还有一个 GROUP BY 时才会使用 HAVING 子句。即当在某一行甚至还没有机会成为一个组的一部分之前就对其应用 WHERE 子句时，可以把 HAVING 子句应用到该组的聚合值上。

为了说明如何使用 HAVING 子句, 下面使用 GROUP BY 子句示例中所采用的查询例子, 但是添加了一个 HAVING 子句:

```
SELECT p.Name AS ProductName, COUNT(DISTINCT soh.AccountNumber) AS UniqueAccounts
FROM Production.Product p
JOIN Sales.SalesOrderDetail sod
    ON p.ProductID = sod.ProductID
JOIN Sales.SalesOrderHeader soh
    ON soh.SalesOrderID = sod.SalesOrderDetailID
WHERE soh.ShipDate >= '07/01/2001'
    AND soh.ShipDate < '08/01/2001'
GROUP BY p.Name
HAVING COUNT(DISTINCT soh.AccountNumber) > 1

ORDER BY p.Name;
```

C.1.2 超出内部连接的表示能力

也许我们编写的 95% 或者更多的查询都要么不使用连接, 要么使用内部连接。对于绝大多数 SQL Server 开发人员来说, 他们编写的所有查询都将落入之前提及的两个类别中。内部连接是以匹配连接两端的行的思想为基础的——即, 当连接的“左边”和“右边”都满足连接条件时, 就满足了内部连接的要求, 接着会对连接两端的行进行匹配并返回这些行。因此, 内部连接在性质上排外的。如果行不匹配, 那么它们就会被排除在结果之外。但是, 还存在其他连接可用, 因此下面介绍一下外部连接、完整连接和交叉连接。

1. OUTER 连接

内部连接在性质上的排外的, 而外部连接在性质上是包含的。所选择的任何一端(左或右)都会返回所有的行, 而不管它们是否匹配。另外一端中的行需要匹配“外部”一端才能被返回, 否则就会被排除在外。介绍外部连接的简单方式是:

```
SELECT <SELECT list>
FROM <the table you want to be the "LEFT" table>
<LEFT|RIGHT> [OUTER] JOIN <table you want to be the "RIGHT" table>
    ON <join condition>
```

需要选择左端还是右端作为连接中所有行都被包含的一端。在 AdventureWorks2008 数据库中使用外部连接的例子如下所示:

```
SELECT sso.SpecialOfferID, Description, DiscountPct, ProductID
FROM Sales.SpecialOffer sso
LEFT OUTER JOIN Sales.SpecialOfferProduct ssop
    ON sso.SpecialOfferID = ssop.SpecialOfferID
```

上面的代码将会返回 SpecialOffer 表(连接的左端)中的所有行, 不管它们是否满足了连接条件, 但是在 SpecialOfferProduct 表(连接的右端)中只有那些满足连接条件的行才会被包含在返回结果中。如果使用了 RIGHT 关键字来替代 LEFT, 那么两张表的角色就会颠倒过来。

2. FULL 连接

可以把这个看成是同时使用 LEFT 和 RIGHT 连接。使用 FULL 连接就是在告诉 SQL Server 把连接两端的所有行都包含进来。AdventureWorks2008 数据库中并没有什么好的例子来展示如何使用完整连接,但由于一旦理解了外部连接之后概念就变得相当简单了,因此这里将只通过一个相当简单的例子予以说明:

```
CREATE TABLE Film
(FilmID          int          PRIMARY KEY,
FilmName         varchar(20)  NOT NULL,
YearMade        smallint     NOT NULL
);
CREATE TABLE Actors
(FilmID          int          NOT NULL,
FirstName        varchar(15)  NOT NULL,
LastName         varchar(15)  NOT NULL,
CONSTRAINT PKActors PRIMARY KEY(FilmID, FirstName, LastName)
);

INSERT INTO Film
VALUES
(1, 'My Fair Lady', 1964);
INSERT INTO Film
VALUES
(2, 'Unforgiven', 1992);

INSERT INTO Actors
VALUES
(1, 'Rex', 'Harrison');
INSERT INTO Actors
VALUES
(1, 'Audrey', 'Hepburn');
INSERT INTO Actors
VALUES
(3, 'Anthony', 'Hopkins');
```

好,现在运行一下 FULL JOIN 来看看结果是什么:

```
SELECT *
FROM Film f
FULL JOIN Actors a
ON f.FilmID = a.FilmID;
```

读者如果检查结果,那么就会发现数据在匹配的地方被连接了,即如果只有左边的数据(右边的列用 NULL 来填充),那么就只取左边的数据,如果恰好只有右边的数据,那么就只取右边的数据(当然,左边的列用 NULL 来填充)。

FilmID	FilmName	YearMade	FilmID	FirstName	LastName
1	My Fair Lady	1964	1	Audrey	Hepburn
1	My Fair Lady	1964	1	Rex	Harrison
2	Unforgiven	1992	NULL	NULL	NULL
NULL	NULL	NULL	3	Anthony	Hopkins

(4 row(s) affected)

3. CROSS 连接

最后一种类型的连接，即 CROSS JOIN 确实是一个非常奇怪的东西。CROSS JOIN 与其他连接的不同之处在于它没有 ON 运算符以及它会把 JOIN 一端的每个记录与 JOIN 另一端的每个记录连接起来。简而言之，最终会得到一个 JOIN 两端所有记录的笛卡尔乘积。除了使用 CROSS 关键字 (替代 INNER、OUTER 或 FULL) 和没有 ON 运算符之外，它的语法与其他类型的 JOIN 的语法是一样的。

现在使用之前在介绍 FULL 连接时构建的小示例，假设正在玩游戏并且希望把每部电影与每个演员混合起来：

```
SELECT *
FROM Film f
```

```
CROSS JOIN Actors a;
```

上面的代码将会把 Film 表中的每一个记录与 Actors 表中的每一个演员匹配起来：

FilmID	FilmName	YearMade	FilmID	FirstName	LastName
1	My Fair Lady	1964	1	Audrey	Hepburn
1	My Fair Lady	1964	1	Rex	Harrison
1	My Fair Lady	1964	3	Anthony	Hopkins
2	Unforgiven	1992	1	Audrey	Hepburn
2	Unforgiven	1992	1	Rex	Harrison
2	Unforgiven	1992	3	Anthony	Hopkins

(6 row(s) affected)

现在，问题应该出来了“为什么要这么做呢？”问的好。要回答这个问题是非常困难的，因为这是依情况而定的。到目前为止，笔者只见过在两种情况下使用 CROSS JOIN：

- **示例数据：**CROSS JOIN 擅长把较小的数据集放在一起，接着通过所有可能的方式把两组数据混合起来，这样就能够获得一个较大的示例数据集。
- **科学数据：**同样，笔者认为这也与示例有关，但是我知道有很多科学计算会用到笛卡尔乘积。有人告诉我进行 CROSS JOIN 是为进行某些类型的分析“准备”数据的一种方式。我并打算假装已经理解了如何使用它来进行统计分析，但是我知道确实有存在这种方式。

最后需要指出的是——它们极少会被用到，但是需要记住有这些选项以防以后会需要它们！

C.2 INSERT 语句

显然，INSERT 语句时用来把数据放入表中的命令。一个 INSERT 语句的基本语法如下所示：

```
INSERT [INTO] <table> [(<column list>)]
VALUES (<data values>)
[, (<data values>) [, . . . n]]
```

下面分别介绍各个部分：

- INSERT 是动作语句。它告诉 SQL Server 这个语句要做什么，所有跟在这个关键字后面的内容仅仅是为了清楚地说明动作的细节。
- INTO 关键字没什么太大的价值。它存在的唯一目的是为了整个语句更加可读一点。尽管它完全是可选的，但是笔者建议把它加到语句中——它使得语句更加容易阅读。
- 接下来说明要把数据插入到哪张表中。
- 接下来的部分稍微有点困难：列列表。虽然显式的列列表(明确地说明哪些列接收值)是可选的，但是不提供一个列列表意味着需要格外小心。如果没有提供一个显式的列列表，那么 INSERT 语句中的每个值就会按照其出现的顺序与表中的列进行匹配(第一个值匹配到第一列，第二个值匹配到第二列，依此类推)。此外，必须按照顺序为每一个列都提供一个值，直到到达最后不接受 null 值并且没有默认值的列为止。总的来说，这里需要提供一个包含一个或多个列的列表，语句的下一个部分将会为这个列表提供数据。
- 最后提供待插入的值。提供值有两种方式：显式提供的值和从 SELECT 语句中派生的值。
- 为了提供值，需要以 VALUES 关键字开头，后面跟着一个值列表，其中值之间用逗号分隔，并且需要把这个列表用括号括起来。值列表中项目的数量必须要与列列表中列的数量完全匹配。每个值的数据类型必须要匹配或者能够隐式转换到相应的列(按照顺序对应)的类型。如果想要添加多于一行的数据，那么就添加一个逗号，然后再提供一个新的值列表，其中值之间用逗号分隔，并且需要用括号把它们括起来。

注意：

在单个 INSERT 语句中插入多行数据的功能是 SQL Server 2008 新增的。如果需要与 SQL Server 2005 兼容，那么就无法使用这个选项。

因此，INSERT 语句看起来可能会像下面这样：

```
INSERT INTO HumanResources.JobCandidate
VALUES
    (1, NULL, DEFAULT),
    .. (55, NULL, GETDATE());
```

正如之前所说的那样，除非提供了一个不同的列列表(稍后会介绍如何提供一个列列表)，否则所有值的顺序都必须与表中列的定义顺序一样。这个规则的一个例外是如果定义了一个标识列，那么在这种情况下该列将会被跳过。

注意：

读者如果检查一下 HumanResources.JobCandidate 表的定义，那么就会发现它实际上是以一个名为 JobCandidateID 的标识列开头的。由于它是一个标识列，而系统将会自动为该列生成一个值，因此可以跳过该列。

由于读者已经知道了大部分的理论(因为你正在阅读本书，而不是在阅读入门书籍)，而现在仅仅是在回顾这些理论，因此笔者已经在其中介绍了几个概念。

- 完全跳过了标识列(系统将会自动填充该列)。
- 提供了实际值(插入的第一行的 EmployeeID 为 1，插入的第二行的 EmployeeID 为 55，并且显式地声明 NULL 为两行的 Resume 字段的值)。

- 在插入的第一行中使用了 `DEFAULT` 关键字(针对 `ModifiedDate`)告诉服务器为该列使用默认值。
 - 仅用一条语句插入了多行数据(同样, 这是 SQL Server 2008 中新增的功能)。
- 现在再次修改一下, 以便在特定的列中插入数据:

```
INSERT INTO HumanResources.JobCandidate
```

```
(EmployeeID, Resume, ModifiedDate)
```

```
VALUES
```

```
(1, NULL, DEFAULT),
```

```
..(55, NULL, GETDATE());
```

注意这里仍然跳过了标识列。与之前的代码的不同之处仅仅在于这里显式地提供了名称——其他的没有任何变化。

C.2.1 INSERT INTO... SELECT 语句

当然, 能够显式地定义将要被放入表中的数据是很好的。但是如果有一块来自可查询的源的数据并且希望把这块数据插入到表中, 那怎么办呢? 这种情况下可查询的数据源包括:

- 数据库中的另一张表。
- 同一服务器上一个完全不同的数据库。
- 在另一个 SQL Server 或其他数据上的异类查询。
- 同一张表(通常在这种情况下, 会在 `SELECT` 语句中进行一些排序或调整)。

`INSERT INTO ... SELECT` 语句可以完成所有这些任务。这个语句的语法是由读者之前见过的两种语句的语法组合而成的——`INSERT` 语句和 `SELECT` 语句。其语法如下所示:

```
INSERT INTO <table name>
[<column list>]
<SELECT statement>
```

`SELECT` 语句的结果集变成了添加到 `INSERT` 语句中的数据。因此, 脚本示例看起来可能像下面这样:

```
USE AdventureWorks2008;

/* This next statement declares our working table.
** This particular table is table variable we are creating on the fly.
*/

DECLARE @MyTable Table
(
    SalesOrderID    int,
    CustomerID      int
);
/* Now that we have our table variable, we're ready to populate it with data
** from our SELECT statement. Note that we could just as easily insert the
** data into a permanent table (instead of a table variable).
*/
INSERT INTO @MyTable
```



```

SELECT SalesOrderID, CustomerID
FROM AdventureWorks.Sales.SalesOrderHeader
WHERE SalesOrderID BETWEEN 50222 AND 50225;

```

```

-- Finally, let's make sure that the data was inserted like we think
SELECT *
FROM @MyTable;

```

上面的代码返回的结果如下所示:

```

(4 row(s) affected)
SalesOrderID CustomerID
-----
50222          638
50223          677
50224          247
50225          175

(4 row(s) affected)

```

C.3 UPDATE 语句

与大多数的 SQL 语句一样, UPDATE 语句所做的工作与它的名字相当符合——它更新现有的数据。与 SELECT 语句相比,它的结构有一点不同,但是还有一定的相似性的。其语法如下所示:

```

UPDATE <table name>
SET <column> = <value> [, <column> = <value>]
[FROM <source table(s)>]
[WHERE <restrictive condition>]

```

虽然可以从多张表中创建一个 UPDATE 语句,但是它只能影响一张表。举一个 Jo Brown 的例子。Jo 看起来最近已经结婚了,因此需要确保她的数据是准确的。下面运行一个查询来看看其中一行数据:

```

SELECT e.BusinessEntityID,
       e.MaritalStatus,
       p.FirstName,
       p.LastName
FROM HumanResources.Employee e
JOIN Person.Person p
  ON e.BusinessEntityID = p.BusinessEntityID
WHERE p.FirstName = 'Jo'
      AND p.LastName = 'Brown';

```

上面的代码将返回下列结果:

```

EmployeeID  MaritalStatus  FirstName  LastName
-----
16          S              Jo         Brown

```

(1 row(s) affected)

下面把 `MaritalStatus` 的值更新为更恰当的 “M”:

```
UPDATE e
SET MaritalStatus - 'M'
FROM HumanResources.Employee e
JOIN Person.Person pe
  ON e.BusinessEntityID - p.BusinessEntityID
WHERE p.FirstName - 'Jo'
AND p.LastName - 'Brown';
```

注意可以修改多个列，只需要加上一个逗号和额外的列表表达式即可。例如，下面的语句还会对 Jo 进行升职：

```
UPDATE e
SET MaritalStatus - 'M', JobTitle - 'Shift Manager'

FROM HumanResources.Employee e
JOIN Person.Person pe
  ON e.BusinessEntityID - p.BusinessEntityID
WHERE p.FirstName - 'Jo'
AND p.LastName - 'Brown';
```

注意：

虽然 SQL Server 允许更新几乎所有的列(有一些是无法更新的，如时间戳)，但是在更新主键的时候需要格外小心。更新主键具有非常高的风险会“孤立”其他数据(引用正在发生变化的数据的数据)。

C.4 DELETE 语句

与所有其他语句相比，DELETE 语句可能是其中最简单的一个。它没有列列表——只有一个表名并且通常会有一个 WHERE 子句。其语法最为简单：

```
DELETE [TOP (<expression>) [PERCENT]]
[FROM ] <table name>
[FROM ] <table list/JOIN conditions>
[WHERE <search condition>]
```

DELETE 语句的难点是两个 FROM 子句(这不是印刷错误)。第一个子句指定在哪个对象上执行 DELETE 语句，第二个子句则是一个更加传统的 FROM 子句(与 SELECT 中的 FROM 子句一样)，它定义了如何确定在哪些行上执行操作。WHERE 子句的工作方式与到目前为止读者见过的所有 WHERE 子句的工作方式一样。这里不需要提供列列表是因为要删除的是整个行(例如，无法删除半行)。

因此，例如如果希望删除 `Actors` 表(本附录之前的几个例子中所使用的表)中所有在 `Film` 表中没有匹配的行，那么这就要求该查询需要同时知道两张表(因此需要使用 JOIN)。此外，还需要知

道要求的是在连接的一端没有匹配(即 Film 表中没有哪个记录能够与一个特定演员进行匹配)。

读者可能已经回忆起 OUTER 连接在一端没有匹配的时候会返回 NULL。这里打算利用这个性质来实际测试 NULL 值:

```
DELETE FROM Actors
FROM Actors a
LEFT JOIN Film f
  ON a.FilmID = f.FilmID
WHERE f.FilmID IS NULL;
```

如果直接跳到第二个 FROM 子句上,那么可以看到这里使用了 LEFT JOIN。这意味着所有的演员都会被返回。那些有匹配的 FilmID 的电影将会被返回,但是如果不存在匹配,那么电影一端的列就会用 NULL 来填充。在本例的 DELETE 语句中使用了这个知识来测试该字段——如果 FilmID 为 null,那么肯定是没有找到匹配(因此需要删除该演员)。

C.5 探究连接的其他语法

同样,大多数“专业”级别的人应该对此有一定的了解,但是由于这些方式最后肯定会用的越来越少,因此这里将只介绍如何采用那些很多人仍然认为是“正常”的方式来实现连接。

现如今大多数查询使用的是 ANSI/ISO 兼容的 SQL 语法(也应该使用这种语法)。值得指出的是,目前旧语法的跨平台支持实际上做得很好,但是现在所有主流平台也都支持 ANSI/ISO 语法。

提示:

笔者还要在这里介绍旧语法的主要原因在于,有一点是绝对可以肯定的,即读者迟早会碰到遗留代码。我不希望到时候读者盯着代码说“这是什么东西?”也就是说我想要重申我的观点,即我强烈建立读者尽可能地使用 ANSI/ISO 语法。因为除了其他原因之外,它的功能性更加强。使用旧语法实际上有可能会产生具有二义性的查询逻辑——解释查询的方式多于一种。而 ANSI/ISO 语法则消除了这个问题。

C.5.1 内部连接的另一种实现方式

一个像下面这样的内部连接可以使用基于 WHERE 子句的连接语法来重写:

```
SELECT *
FROM HumanResources.Employee e
INNER JOIN HumanResources.Employee m
  ON e.ManagerID = m.EmployeeID;
```

只需要移除单词 INNER JOIN,然后添加一个逗号并使用 WHERE 子句来替代 ON 运算符即可:

```
SELECT *
FROM HumanResources.Employee e, HumanResources.Employee m
WHERE e.ManagerID = m.EmployeeID;
```

提示：
实际上，目前世界上的主流 SQL 系统(Oracle、DB2、MySQL 等等)都支持这种语法。

C.5.2 外部连接的另一种实现方式

在 SQL Server 2005 以及之后的版本中，实际上并没有第二种语法来实现外部连接。实际上，现在这个功能默认是被关闭的——必须要把数据库兼容性级别设置为 80 或者更低的级别(80 是 SQL Server 2000)。可喜的是，现在使用这种语法的代码已经非常少了，并且由于缺少默认的支持，因此笔者猜测到 SQL Server 2008 的生命周期结束的时候，这种类型的代码将会消失。

那就是说笔者在这里只是稍微介绍一下以防你遇到这种代码。它的工作方式与内部连接基本相同，但是由于没有 LEFT 或 RIGHT 关键字(同时也没有 OUTER 或 JOIN 关键字)，因此需要专门为这个任务创建一些特殊的运算符。这些运算符如表 C-2 所示：

表 C-2	
替代方案	ANSI
<code>*=</code>	LEFT JOIN
<code>=*</code>	RIGHT JOIN

因此，像下面这样的外部连接：

```
SELECT e.EmployeeID, m.EmployeeID AS ManagerID
FROM HumanResources.Employee e
LEFT OUTER JOIN HumanResources.Employee m
ON e.ManagerID = m.EmployeeID;
```

可以按以下方式转化为旧式的外部连接语法：

```
SELECT e.EmployeeID, m.EmployeeID AS ManagerID
FROM HumanResources.Employee e, HumanResources.Employee m
WHERE e.ManagerID *= m.EmployeeID;
```

注意：
外部连接的可选语法在默认情况下是不可用的。只有把兼容性模式设置为 80 或者更低才能使用这项功能。

C.5.3 CROSS JOIN 的另一种实现方式

这是到目前为止最简单的一个了。使用旧语法创建交叉连接不需要做任何事情。即不需要在形式为 TableA.ColumnA=TableB.ColumnA 的 WHERE 子句中加入任何东西。

因此像下面这样用 ANSI 语法实现的交叉连接：

```
SELECT *
FROM Film f
CROSS JOIN Actors a;
```


将会变成:

```
SELECT *
```

```
FROM Film f, Actors a;
```

C.6 UNION

同样, 读者应该已经对这部分内容至少有点熟悉了, 但令笔者感到惊奇的是, 我经常发现那些相对来说有经验的 SQL 程序员虽然知道 UNION 的存在, 但是却并不理解 UNION 语句。UNION 是一个特殊的运算符, 使用它可以使得两个或多个查询生成一个结果集。

UNION 把一个查询返回的数据附到另一个查询的结尾(在功能上, 它的工作方式与这里的描述有一点点不同, 但是这是理解这个概念的最简便的方式)。JOIN 是横向地组合信息(添加更多的列), 而 UNION 则是纵向地组合数据(添加更多的行), 如图 C-1 所示。

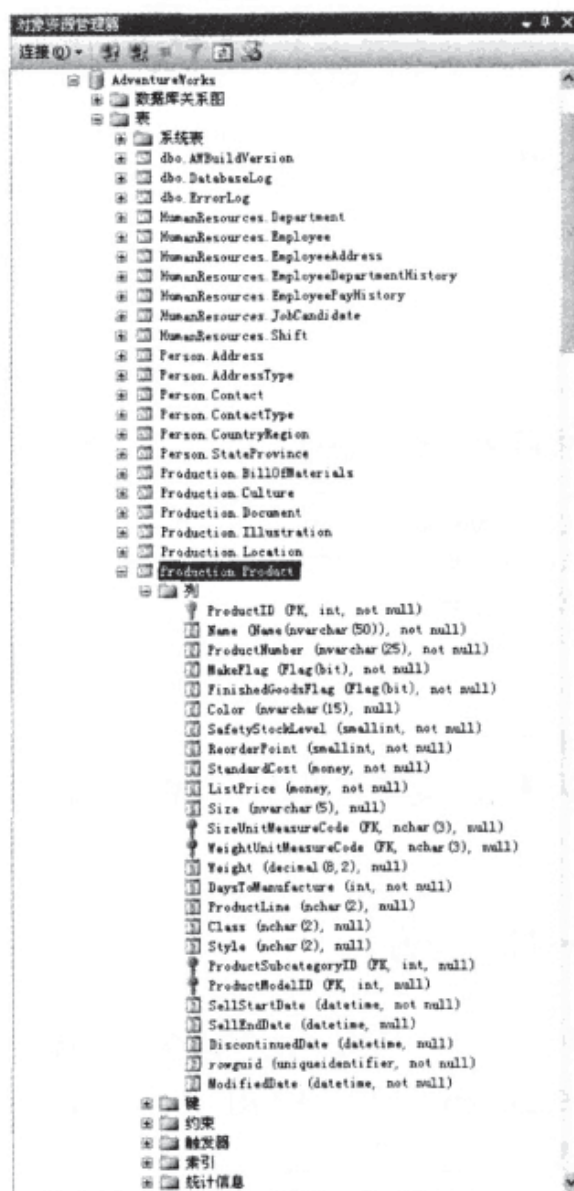


图 C-1

在处理使用 UNION 的查询时有几个关键点需要注意:

- 所有被联合的查询的 SELECT 列表包含的列数必须一样。
- 只从第一个查询中获取返回的合并结果集的表头信息。
- 查询中每个列的数据类型必须要与其他查询中对应列的数据类型隐式兼容。
- 与非联合查询不同，联合的默认返回选项是 DISTINCT，而不是 ALL。除非在查询中使用 ALL 关键字，否则只会返回重复行的一行。

在本例中将会创建两张表并且会在这两张表上执行选择语句。接着会向每张表中插入三行数据，其中有一行数据在两张表中是完全一样的。如果查询使用了 ALL 关键字，那么每一行(6 行)都会显示出来。如果查询使用了 DISTINCT 关键字，那么将只返回 5 行(丢弃了重复行)。

```
CREATE TABLE UnionTest1
(
    idcol  int          IDENTITY,
    col2   char(3),
);

CREATE TABLE UnionTest2
(
    idcol  int          IDENTITY,
    col4   char(3),
);

INSERT INTO UnionTest1
VALUES
    ('AAA');
INSERT INTO UnionTest1
VALUES
    ('BBB');
INSERT INTO UnionTest1
VALUES
    ('CCC');

SELECT *
FROM UnionTest1;

INSERT INTO UnionTest2
VALUES
    ('CCC');
INSERT INTO UnionTest2
VALUES
    ('DDD');
INSERT INTO UnionTest2
VALUES
    ('EEE');

PRINT 'Regular UNION-----'
SELECT col2
FROM UnionTest1
UNION
SELECT col4
FROM UnionTest2;

PRINT 'UNION ALL-----'

SELECT col2
```



```

FROM UnionTest1
  UNION ALL
SELECT col4
FROM UnionTest2;

DROP TABLE UnionTest1;
DROP TABLE UnionTest2;

```

运行上面的代码，关键的结果如下所示：

```

Regular UNION-----
col2
----
AAA
BBB
CCC
DDD
EEE

(5 row(s) affected)

UNION ALL-----
col2
----
AAA
BBB
CCC
CCC
DDD
EEE

(6 row(s) affected)

```

C.6.1 子查询和派生表

子查询代表了利用一个查询提供结果供另一个查询使用的基本概念。它们使用括号把子查询嵌入到顶级查询中，其实现方式存在三种基本形式：

- **嵌套子查询：**一般来讲，“内部”查询将会提供一个简单的查找源来满足外部查询的一部分需要。例如，可以用它来向选择列表提供一个查找源，但这个查找源中的数据与查询中的其他数据不是来自同一个地方，或者也可以用于 **WHERE** 子句以强制查询满足某些查找条件。
- **相关子查询：**它们与普通的嵌套子查询类似，但是在性质上是双向的。即内部查询会接收来自外部查询的信息，然后利用那些信息来确定向外部查询提供哪些信息。
- **派生表(也称为行内视图)：**它们利用了查询返回的结果相当于是一张表的思想，允许像引用表那样引用查询。可以把整张表或视图与某个查询生成的结果进行连接(用于连接的查询称为派生表)。

下面简单地回顾一下每种形式的语法以及使用该语法的示例。

1. 嵌套子查询

嵌套子查询是单向的——返回单个值供外部查询使用或者可能返回整个值列表供 IN 运算符使用。

在最宽泛的意义下，查询语法看起来将会与下列两种语法模板中的一个类似：

```
SELECT <SELECT list>
FROM <SomeTable>
WHERE <SomeColumn> = (
    SELECT <single column>
    FROM <SomeTable>
    WHERE <condition that results in only one row returned>)
```

或者：

```
SELECT <SELECT list>
FROM <SomeTable>
WHERE <SomeColumn> IN (
    SELECT <single column>
    FROM <SomeTable>
    [WHERE <condition>])
```

显然，实际的语法是多种多样的，这不仅仅是因为需要替换选择列表和实际的表名，而且也因为在内部或外部查询中可能会用到多表连接——或者两个查询中都会用到。

嵌套选择的一个简单示例(本例使用第二个模板)如下所示：

```
SELECT ProductID, Name
FROM Production.Product
WHERE ProductID IN (
    SELECT ProductID FROM Sales.SpecialOfferProduct);
```

这里需要掌握的关键点是把另一个查询用括号括起来，而且该查询返回的内容必须要符合要求(如果使用它的查询期望得到一个标量值，那么就返回一个标量值，或者如果列表是合适的，那么就返回一个列表)。

2. 相关子查询

相关子查询与之前介绍的嵌套子查询的不同之处在于信息是双向传递的，而不是单向传递的。

与仅仅向顶层查询提供信息不同，相关子查询是双向的，其工作过程分为三步：

- (1) 外部查询获取一个记录并把这个记录传递给内部查询。
- (2) 根据传入的值执行内部查询。
- (3) 内部查询把结果值传回外部查询，外部查询使用该结果值完成处理。

在选择列表中使用相关子查询的示例如下所示：

```
SELECT sc.AccountNumber,
    (SELECT Min(OrderDate)
     FROM Sales.SalesOrderHeader soh
     WHERE soh.CustomerID = sc.CustomerID)
    AS OrderDate
```



```
FROM Sales.Customer sc
```

这个查询将会查找每个客户在文件中的第一个订单并且返回该订单与帐号。

3. 派生表

这个没有那么众所周知的 SQL 构造由查询返回的结果集中的列和行组成(既然它们像普通的表那样有列、行、数据类型等等,那为什么不把它们当成表来用呢?).与之前在本附录中介绍的子查询一样,只需要把查询嵌入到括号中即可,接着就可以把结果当成表来使用了。可以把它看成是用括号括起来的视图,而不是一个已命名的对象。唯一的附加要求是必需要使用一个别名,这是因为它不像表或视图那样有一个正式的名称。

下面的示例中使用了多张派生表(它使用 AdventureWorks2008 数据库来显示所有买了 HL 后轮山地车和 HL 前轮山地车的客户的帐号和所在地区):

```
SELECT DISTINCT sc.AccountNumber, sst.Name
FROM Sales.Customer AS sc
JOIN Sales.SalesTerritory sst
  ON sc.TerritoryID = sst.TerritoryID
JOIN
  (SELECT CustomerID
   FROM Sales.SalesOrderHeader soh
   JOIN Sales.SalesOrderDetail sod
     ON soh.SalesOrderID = sod.SalesOrderID
   JOIN Production.Product pp
     ON sod.ProductID = pp.ProductID
   WHERE pp.Name = 'HL Mountain Rear Wheel') AS dt1
  ON sc.CustomerID = dt1.CustomerID
JOIN
  (SELECT CustomerID
   FROM Sales.SalesOrderHeader soh
   JOIN Sales.SalesOrderDetail sod
     ON soh.SalesOrderID = sod.SalesOrderID
   JOIN Production.Product pp
     ON sod.ProductID = pp.ProductID
   WHERE Name = 'HL Mountain Front Wheel') AS dt2
  ON sc.CustomerID = dt2.CustomerID
```

特别要注意使用括号把每张派生表括起来以及为每张派生表使用别名(dt1 和 dt2)。

C.7 小结

本附录仅仅是对 SQL Server 中一些极其基本的概念的一个快速回顾。本书假设读者已经对这里介绍的大部分内容有了相当深入的理解。本附录很大程度上是为了回顾,读者如果发现自己正在这里介绍的任何概念上痛苦挣扎,那么应该考虑参阅由清华大学出版社引进并出版的《SQL Server 2008 编程入门经典》一书。