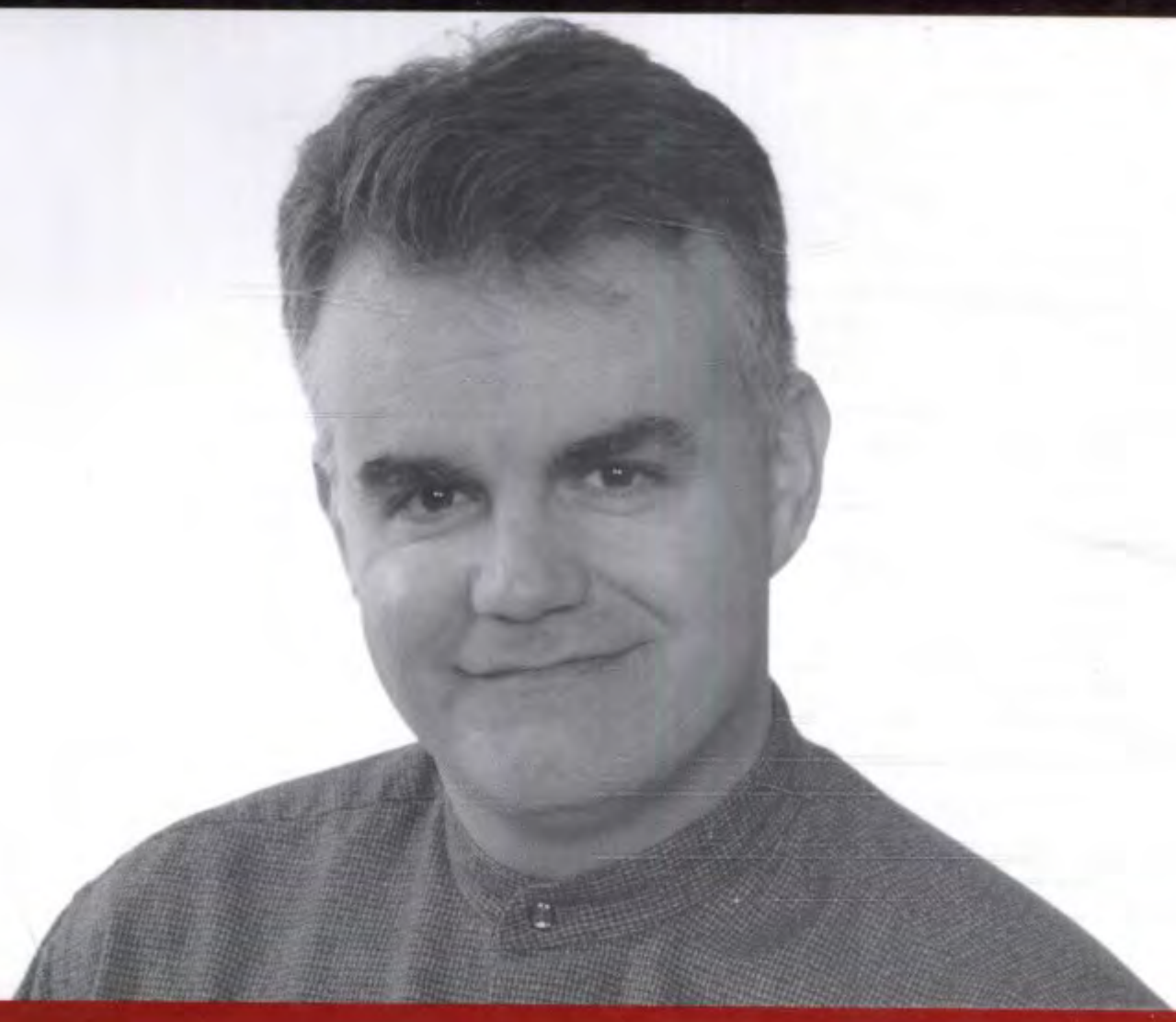
PROGRAMMER TO PROGRAMMER™



Beginning Microsoft SQL Server 2008 Programming

SQL Server 2008 编程入门经典

（第3版）

(美) Robert Vieria 著
马 煜
孙 皓 译
杨大川 审校

wrox™

WILEY

清华大学出版社

SQL Server 2008
编程入门经典

(第 3 版)

(美) Robert Vieira 著

马 煜
孙 皓 译

杨大川 审校

清华大学出版社

北 京

www.TopSage.com

Robert Vieira
Beginning Microsoft SQL Server 2008 Programming
EISBN：978-0-470-25701-2
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：01-2009-1222

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 编程入门经典(第 3 版)/(美)韦拉(Viera, R)著；马煜，孙皓 译；杨大川 审校.
—北京：清华大学出版社，2010.1
书名原文：Beginning Microsoft SQL Server 2008 Programming
ISBN 978-7-302-21432-8

Ⅰ.S…　Ⅱ. ①韦…　②马…③孙…④杨…　Ⅲ. ①关系数据库—数据库管理系统，SQL Server 2008
Ⅳ. ①TP311.138

中国版本图书馆 CIP 数据核字(2009)第 204196 号

责任编辑：王　军　谢晓芳
装帧设计：孔祥丰
责任校对：成凤进
责任印制：孟凡玉
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：三河市新茂装订有限公司
经　销：全国新华书店
开　本：185×260　**印　张**：36　**字　数**：968 千字
版　次：2010 年 1 月第 1 版　　**印　次**：2010 年 1 月第1 次印刷
印　数：1～4000
定　价：69.80 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：(010)62770177 转 3103　　产品编号：029395-01

www.TopSage.com

审 校 者 序

数据库技术自诞生以来，关系数据库技术一直是最主流的技术，它至今依然是绝大部分技术人员研究和学习的重点。

微软的 SQL Server 产品，从最初的 SQL Server NT 版，到后来的 SQL Server 2000 和 SQL Server 2005，再到现在最新的 SQL Server 2008，一直倍受大量用户的青睐，与此同时企业 IT 部门也急需能够精通这门技术的数据库程序员。

我觉得，精通 SQL 开发需要经历一个循序渐进的过程，最后才能真正变成一个顶级高手。

第一步：了解关系型数据库和 SQL 语言，可以熟练编写 Select/Insert 等语句。此刻，您已经可以使用很多软件了，例如 Office(数据连接)，也可以在程序中嵌入一些简单的数据库访问操作。其实，您能够进行很多基本的数据库操作了。但我可以告诉您，至少 90%以上的程序员都能达到这一步，您离高手还远。

第二步：掌握 T-SQL 语言、存储过程等。您会发现，原来开发 SQL 时，不仅可以传递变量和使用分支语句，还可以以调试模式单步测试。并且很多熟悉的编程语法也可以在数据库里直接使用。此时，您已经可以进行比较复杂的数据库开发了。比起来第一阶段的同行，您的薪酬可能会翻倍。

第三步：如果您运气够好，那么可以接触到 TB 级的海量数据了，甚至是每天 TB 级别的增量数据。您必须面对数据库访问超时的僵局。经历了这个阶段，您应该学会如何优化海量数据库，如何针对海量数据库的特殊要求来开发 SQL 和优化性能。现在的您，已经是市场上不可多得的人才了。

第四步：如果您了解一些 BI 知识的话，就应该知道在微软产品线中，SSIS 做为 ETL 工具的重要性。您也会知道另两个产品：SSAS 和 SSRS。如果您还不知道如何编写 MDX，那就赶紧去学吧：多维数据库已经越来越热门。还有 XML/A 技术、LINQ 技术、等等。术语很多，难度并非想象的那么大。如果您已经有了前面的基础话，那么这些都是完全可以学好和学精的。现在的您，我个人建议您应该脱离程序员的范畴，往资深顾问的路线发展，因为您的视野开阔了(如果熟悉一些其他数据库和 BI 产品就更好，但是切忌多而不精)。

第五步：您或许已经开始思考：关系数据库、多维数据库、面向对象数据库，以及未来的数据库会怎样？从 SQL 到存储过程，从 ODBC 到 LINQ，下一个数据库程序开发的热点又在哪里？当您苦苦思索这些问题的时候，您的水平已经远远超出我能够预测的了。

本书涵盖的范围，大致上是前 3 个阶段，除了学习本书之外，我还鼓励大家学习与之相关的其他知识，如 Reporting Service, Integration Service，当然还有.NET 程序语言等。

我在国外学习、工作了多年之后，深感数据库技术、商业智能已经成为目前 IT 领域最热门的话题，精通这些技术的人才是每个企业都不可或缺的。因此从 2003 开始我创建了北京迈思奇科技有限公司，致力于将国外的先进数据库技术、商业智能技术和工具引进国内，帮助国

www.TopSage.com

内的企业提高数据分析效率、增强竞争实力。公司成立 6 年来，与微软密切合作，通过近百次讲座和培训，为企业培养数据库和商业智能专业人员；同时，在承担相关项目实施的过程中，公司也积累了优秀的团队和丰富的项目案例，创立了国内一流的品牌。

感谢清华大学出版社，及时引进了这本快速入门的 SQL Server 2008 的优秀参考书；感谢本书的译者，准确而清晰地传达了原著的精华；也感谢迈思奇公司参与本书审校工作的各位咨询顾问。

北京迈思奇科技有限公司　　杨大川

www.TopSage.com

前　言

正如我曾说过的，写书的经历奇特而漫长。早在 1999 年，当我首次编写 *Professional SQL Server 7.0 Programming* 一书时，无论是书籍还是程序开发环境都与今天的情况截然不同。那时，.NET 尚未出现，Visual Studio 98 是当时最流行的开发环境，Java 开发环境正被广泛使用，而像 Delphi 这样可供选择的开发工具也比现在更具竞争力。现在正快速进入所谓的.com 时代，数据库管理系统(DBMS，如 SQL Server)的使用也在呈指数增长。

然而，却存在一个问题。虽然我们可以找到一些有关 SQL Server 的书籍，但这些书籍都面向数据库管理员。这些书籍的作者将大量的时间和精力花费在了普通开发人员几乎不关注的内容上。事实上，对于一些无能为力的事情，我们必须放弃。就像迫切需要解决的全球饥饿或核扩散问题，因为我们根本无力解决，但我们完全能够编写一些面向开发人员的有关 SQL 的新书籍。

当时我写了 *Professional SQL Server 7.0 Programming* 一书，内容可谓包罗万象，从基础内容到高级内容，适合于各类读者。但其造就的结果就是书特别的厚。

第一本书 SQL Server 2005 发行后，我们计划重新编写 SQL Server 的第二次修订版本，但再次碰到了一个问题：有关高级编程的内容实在是太多了，以至于无法用一本书来涵盖全部内容。SQL Server 2005 的新增内容也很多，要全放在一本书中介绍实属不易(几乎走出了一本书的范围)，因此我们决定将原来的“高级编程”系列丛书分成“入门经典”和“高级编程”篇。尽管我们花了大量精力去进行拆分，但仍有不少内容重叠。SQL Server 2008 又增加了一些新功能，不过这样一来，我们可以使得“高级编程”篇面向高级读者，而“入门经典”篇内容也更加丰富。

本书尽可能覆盖了 SQL Server 的主要内容，我们也希望它能与先前编写的 *Professional SQL Server 7.0 Programming* 一书一样成功。在学习完“入门经典”篇后，如果您希望自己成为一名高级的 SQL Server 2005 编程人员，那么可以继续学习本套书的“高级编程”篇。

0.1　本书读者对象

本书书名中的“入门”一词容易引起误解。本书的确非常适合初学者，但它也适合您入门之后的进一步学习。其内容是初学者所必须了解的，但由于其内容繁多，读者不可能全部记牢，因此您可在需要时回顾并参考相关内容，这对于中、高级用户来说也同样适用。

初学者一般希望从头开始学习。而本书主要内容都是按“需要掌握”哪类知识来编排的。除了第 16 章、第 17 章和第 18 章之外，本书的每一章都是易于理解的基本内容，这样读者在遇到 SQL Server 问题时，可很容易地找到相关内容来学习。

www.TopSage.com

对于中级用户，可以跳过本书的前 7 章或前 8 章。但我们仍建议浏览这几章，以弥补这方面知识的漏洞。也可以先浏览一下这几章，然后快速跳过熟悉的内容，去阅读不了解的内容。

高级用户除了可利用本书作为很好的参考资料之外，可能会希望学习第 12 章以及后面的章节。事实上，第 12 章以后的内容更加新颖(如新的调试、事务、XML、Reporting Services、Integration Services 内容等)。

0.2 本书内容简介

从本书书名可以看出，本书涵盖的 SQL Server 2008 的内容是明确地面向开发人员的。

SQL Server 2008 是发展近 20 年的数据库管理系统的最新版本。它是在对 SQL Server 7.0 版本重新设计的基础上建立的——提供了单独的日期和时间数据类型，以及地理空间支持。本书着重解决各层次开发人员的核心编程需求。另外，本书主要围绕 SQL Server 2008 版本讲解，但也会涉及向后兼容性的问题，因为这可能影响设计或编码选择。

0.3 本书内容安排

本书内容按照从基本到高级的顺序安排，同时假定您已经是富有经验的开发人员(除了数据库方面)。要学习本书，您必须具有编程的基本知识，如变量、数据类型以及过程化编程。当然，您不必学过查询。

本书尤其适合于开发人员使用。为了简洁起见，本书会掩盖或完全忽略那些面向数据库管理员而非开发人员的内容。当然本书也涉及数据库管理的问题，因为这些问题在开发过程中需要考虑到，本书将在第 19 章对几个数据库管理相关的问题作简单的介绍。

在客户端开发方面，本书尽量做到与语言无关。通常忽略所使用的语言，如 VB、C#、C++、Java，以及其他语言(本书侧重于服务器端的开发)，而在涉及时也对不同语言一视同仁。

在学习顺序方面，首先学习 SQL 基本对象，然后是基本的查询和连接知识。接着学习数据库的其他对象，并讨论与数据库物理设计有关的内容。再然后学习如何编写更健壮的 SQL Server 脚本代码、存储过程、用户自定义函数与触发器。再进一步学习一些 SQL Server 的外围功能。本书最后要学习的是数据库管理方面的重要内容，有助于维护用户开发的数据库。

0.4 使用本书的条件

使用本书需要安装 SQL Server。由于本书大量使用各种 SQL Server 2008 管理工具，因此我推荐安装包含所有模块功能的 SQL Server 版本，而不是 SQL Server Express。虽说如此，本书还是侧重于介绍脚本编写，因此即使是 SQL Server Express 用户，也能学习本书的大部分内容。还需要 AdventureWorks2008 样本数据库，以及安装一些自定义数据库。访问这些数据库的指令参见本书配套网站(www.wrox.com)上的 ReadMe 文件。

www.TopSage.com

虽然 Visual Studio 对于学习本书也很有用，但是大部分所需的 Visual Studio 特性都包括在 SQL Server 产品附带的 Business Intelligence Studio 中。

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，就可以获得所有的源代码。

注释：

由于许多图书的标题都很类似，因此按 ISBN 搜索是最简单的，本书英文版的 ISBN 是 978-0-470-25701-2。

在下载了代码后，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，如果您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，可以让其他读者避免受挫，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，我们就会检查您的反馈信息，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，便于您张贴与 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，它可以给您传送感兴趣的论题。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

www.TopSage.com

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 按扭。

(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 按扭。

(4) 您会收到一封电子邮件，其中的信息描述了如何验证账户，完成加入过程。

注释：

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。

www.TopSage.com

目　录

第 1 章　RDBMS 基础：SQL Server 数据库的构成……1
1.1　数据库对象概述……1
1.1.1　数据库对象……2
1.1.2　事务日志……6
1.1.3　最基本的数据库对象：表……6
1.1.4　文件组……8
1.1.5　数据库关系图……8
1.1.6　视图……9
1.1.7　存储过程……10
1.1.8　用户自定义函数……10
1.1.9　用户和角色……10
1.1.10　规则……10
1.1.11　默认值……11
1.1.12　用户自定义数据类型……11
1.1.13　全文目录……11
1.2　SQL Server 数据类型……11
1.3　SQL Server 对象标识符……16
1.3.1　需要命名的对象……16
1.3.2　命名规则……17
1.4　本章小结……17

第 2 章　SQL Server 管理工具……18
2.1　联机丛书……18
2.2　SQL Server 配置管理器……19
2.2.1　服务管理……20
2.2.2　网络配置……20
2.2.3　协议……21
2.2.4　客户端……23
2.3　SQL Server Management Studio……25
2.3.1　启动 Management Studio……25
2.3.2　“查询”窗口……28
2.4　SQL Server 集成服务(SSIS)……32
2.5　Bulk Copy Program(bcp)……33
2.6　SQL Server Profiler……33
2.7　sqlcmd……34
2.8　PowerShell……34
2.9　本章小结……34

第 3 章　T-SQL 基本语句……35
3.1　基本 SELECT 语句……36
3.1.1　SELECT 语句与 FROM 子句……36
3.1.2　WHERE 子句……39
3.1.3　ORDER BY 子句……42
3.1.4　使用 GROUP BY 子句聚合数据……44
3.1.5　使用 HAVING 子句给分组设置条件……52
3.1.6　使用 FOR XML 子句输出 XML……53
3.1.7　通过 OPTION 子句利用提示……54
3.1.8　DISTINCT 和 ALL 谓词……54
3.2　使用 INSERT 语句添加数据……56
3.2.1　多行插入……60
3.2.2　INSERT INTO…SELECT 语句……61
3.3　用 UPDATE 语句更改数据……63
3.4　DELETE 语句……65
3.5　本章小结……66
3.6　练习……66

第 4 章　连接……67
4.1　连接……67
4.2　内部连接……69
4.3　外部连接……75

www.TopSage.com

4.3.1 简单的外部连接 ……76
4.3.2 处理更复杂的外部连接 ……81
4.4 完全连接 ……85
4.5 交叉连接 ……86
4.6 JOIN 语句的早期语法结构 ……87
4.6.1 内部连接的早期语法结构 ……88
4.6.2 外部连接的早期语法结构 ……88
4.6.3 交叉连接的早期语法结构 ……89
4.7 联合 ……90
4.8 本章小结 ……94
4.9 练习 ……94

第 5 章 创建和修改数据表 ……95
5.1 SQL Server 中的对象名 ……95
5.1.1 模式名 ……95
5.1.2 数据库名 ……98
5.1.3 通过服务器命名 ……98
5.1.4 回顾默认值 ……98
5.2 CREATE 语句 ……99
5.2.1 CREATE DATABASE ……99
5.2.2 CREATE TABLE ……104
5.3 ALTER 语句 ……115
5.3.1 ALTER DATABASE ……115
5.3.2 ALTER TABLE ……119
5.4 DROP 语句 ……122
5.5 使用 GUI 工具 ……123
5.5.1 使用 Management Studio 创建数据库 ……123
5.5.2 回到代码：使用 Management Studio 创建脚本的基础知识 ……127
5.6 本章小结 ……128
5.7 练习 ……128

第 6 章 约束 ……129
6.1 约束的类型 ……130
6.1.1 域约束 ……130
6.1.2 实体约束 ……130
6.1.3 参照完整性约束 ……131
6.2 约束命名 ……131
6.3 键约束 ……132
6.3.1 主键约束 ……132
6.3.2 外键约束 ……134
6.3.3 唯一约束 ……144
6.4 CHECK 约束 ……145
6.5 DEFAULT 约束 ……146
6.5.1 在 CREATE TABLE 语句中定义 DEFAULT 约束 ……147
6.5.2 在已存在的表中添加 DEFAULT 约束 ……148
6.6 禁用约束 ……148
6.6.1 在创建约束时忽略无效的数据 ……148
6.6.2 临时禁用已存在的约束 ……150
6.7 规则和默认值 ……152
6.7.1 规则 ……152
6.7.2 默认值 ……154
6.7.3 确定哪个表和数据类型使用给定的规则或默认值 ……154
6.8 用于实现数据完整性的触发器 ……155
6.9 如何选择 ……155
6.10 本章小结 ……156

第 7 章 更复杂的查询 ……157
7.1 子查询的概念 ……158
7.2 关联子查询 ……161
7.2.1 关联子查询的工作原理 ……162
7.2.2 在 WHERE 子句中的关联子查询 ……162
7.2.3 处理 NULL 数据——ISNULL 函数 ……165
7.3 派生表 ……166
7.4 EXISTS 运算符 ……168
7.5 混合数据类型：CAST 和 CONVERT ……171
7.6 MERGE 命令 ……174
7.6.1 BY TARGET 和 BY SOURCE ……177
7.6.2 OUTPUT 子句 ……177
7.7 性能方面的考虑 ……180

www.TopSage.com

7.8 本章小结 ……181
7.9 练习 ……181

第 8 章 规范化和其他基本设计问题 ……182
8.1 表 ……182
8.2 保持数据“规范” ……183
8.2.1 准备工作 ……184
8.2.2 第一范式 ……185
8.2.3 第二范式 ……188
8.2.4 第三范式 ……189
8.2.5 其他范式 ……191
8.3 关系 ……192
8.3.1 一对一关系 ……192
8.3.2 一对一或一对多关系 ……193
8.3.3 多对多关系 ……195
8.4 数据库关系图 ……198
8.4.1 表 ……200
8.4.2 添加和删除表 ……200
8.4.3 关系 ……205
8.5 反规范化 ……207
8.6 超出规范化的一些规则 ……208
8.6.1 保持简单 ……208
8.6.2 选择数据类型 ……208
8.6.3 尽量进行存储 ……209
8.7 创建一个快速示例 ……209
8.7.1 创建数据库 ……209
8.7.2 添加数据库关系图和初始表 ……210
8.7.3 添加关系 ……213
8.7.4 添加一些约束 ……215
8.8 本章小结 ……216
8.9 练习 ……216

第 9 章 SQL Server 存储和索引结构 ……217
9.1 SQL Server 存储机制 ……217
9.1.1 数据库 ……217
9.1.2 区段 ……218
9.1.3 页 ……218
9.1.4 行 ……219
9.1.5 稀疏列 ……219
9.2 理解索引 ……219
9.2.1 平衡树(B-树) ……220
9.2.2 SQL Server 中访问数据的方式 ……223
9.3 创建、修改和删除索引 ……230
9.3.1 CREATE INDEX 语句 ……230
9.3.2 创建 XML 索引 ……234
9.3.3 随约束创建的隐含索引 ……235
9.3.4 在稀疏列和地理空间列上创建索引 ……236
9.4 明智地选择——在何时何地使用何种索引 ……236
9.4.1 选择性 ……236
9.4.2 注意成本 ……236
9.4.3 选择群集索引 ……237
9.4.4 列顺序问题 ……239
9.4.5 修改索引 ……239
9.4.6 删除索引 ……241
9.4.7 从查询计划中获取提示 ……242
9.4.8 使用数据库引擎调整向导 ……242
9.5 维护索引 ……242
9.5.1 碎片 ……242
9.5.2 确定碎片和页拆分的可能性 ……243
9.6 本章小结 ……246
9.7 练习 ……247

第 10 章 视图 ……248
10.1 简单的视图 ……248
10.2 更加复杂的视图 ……253
10.3 使用 T-SQL 编辑视图 ……259
10.4 删除视图 ……260
10.5 在 Management Studio 中创建和编辑视图 ……260
10.6 审查：显示现有的代码 ……263
10.7 保护代码：加密视图 ……264
10.8 关于模式绑定 ……265
10.9 使用 VIEW_METADATA 使视图看起来像表一样 ……266

www.TopSage.com

10.10 索引(物化)视图……266
10.11 本章小结……268
10.12 练习……269

第 11 章 编写脚本和批处理……270
11.1 脚本的基础知识……270
11.1.1 USE 语句……271
11.1.2 声明变量……271
11.1.3 使用@@IDENTITY……275
11.1.4 使用@@ROWCOUNT……278
11.2 批处理……279
11.2.1 批处理中的错误……281
11.2.2 何时使用批处理……281
11.3 sqlcmd……284
11.4 动态 SQL：用 EXEC 命令动态生成代码……288
11.5 控制流语句……293
11.5.1 IF…ELSE 语句……293
11.5.2 CASE 语句……297
11.5.3 用 WHILE 语句进行循环……302
11.5.4 WAITOR 语句……303
11.5.5 TRY/CATCH 块……303
11.6 本章小结……306
11.7 练习……306

第 12 章 存储过程……307
12.1 创建存储过程：基本语法……307
12.2 使用 ALTER 修改存储过程……309
12.3 删除存储过程……309
12.4 参数化……310
12.5 通过返回值确认成功或失败……315
12.6 错误处理……317
12.6.1 以前的方式……318
12.6.2 在错误发生前处理错误……323
12.6.3 手动引发错误……328
12.6.4 添加自定义的错误消息……331
12.7 存储过程的优点……333
12.7.1 创建可调用的进程……333
12.7.2 为了安全性使用存储过程……333
12.7.3 存储过程和性能……334
12.8 扩展存储过程(XP)……336
12.9 递归简介……336
12.10 调试……338
12.10.1 启动调试器……339
12.10.2 调试器的组成……339
12.10.3 使用调试器……342
12.11 .NET 程序集……343
12.12 本章小结……344

第 13 章 用户自定义函数……345
13.1 UDF 的定义……345
13.2 返回标量值的 UDF……346
13.3 返回表的 UDF……350
13.4 调试用户自定义函数……358
13.5 数据库中的.NET……358
13.6 本章小结……358
13.7 练习……358

第 14 章 事务和锁……359
14.1 事务……359
14.1.1 BEGIN TRAN……360
14.1.2 COMMIT TRAN……360
14.1.3 ROLLBACK TRAN……361
14.1.4 SAVE TRAN……361
14.2 SQL Server 日志的工作方式……361
14.2.1 失败和恢复……362
14.2.2 隐式事务……364
14.3 锁和并发……364
14.3.1 通过锁可以防止的问题……365
14.3.2 可以锁定的资源……368
14.3.3 锁升级和锁对性能的影响……368
14.3.4 锁定模式……369
14.3.5 锁的兼容性……370
14.3.6 指定一种特有的锁定类型——优化器提示……371
14.4 设置隔离级别……372
14.5 处理死锁(1205 错误)……375

www.TopSage.com

14.5.1 SQL Server 判断死锁的方式……375
14.5.2 选择死锁牺牲者的方式……375
14.5.3 避免死锁……375
14.6 本章小结……377

第 15 章 触发器……379
15.1 触发器的概念……380
15.1.1 ON 子句……381
15.1.2 WITH ENCRYPTION 选项……381
15.1.3 FOR | AFTER 子句与 INSTEAD OF 子句……381
15.1.4 WITH APPEND 选项……384
15.1.5 NOT FOR REPLICATION 选项……384
15.1.6 AS 子句……384
15.2 使用触发器实施数据完整性规则……384
15.2.1 处理来自于其他表的需求……385
15.2.2 使用触发器来检查更新的变化……386
15.2.3 将触发器用于自定义错误消息……388
15.3 触发器的其他常见用途……388
15.4 触发器的其他问题……388
15.4.1 触发器可以嵌套……389
15.4.2 触发器可以递归……389
15.4.3 触发器不能防止体系结构的修改……389
15.4.4 可以在不删除的情况下关闭触发器……390
15.4.5 触发器的激活顺序……390
15.5 INSTEAD OF 触发器……392
15.6 性能考虑……392
15.6.1 触发器的被动性……392
15.6.2 触发器与激活的进程之间不存在并发问题……392
15.6.3 使用 IF UPDATE()和 COLUMNS_UPDATED()……393
15.6.4 保持触发器短小精悍……395
15.6.5 选择索引时不要忘记触发器……395
15.6.6 不要在触发器中进行回滚……395
15.7 删除触发器……395
15.8 调试触发器……396
15.9 本章小结……397

第 16 章 初识 XML……398
16.1 XML 基础……399
16.1.1 XML 文档的各个组成部分……400
16.1.2 名称空间……407
16.1.3 元素内容……408
16.1.4 有效与格式良好——架构和 DTD……409
16.2 SQL Server 提供的 XML 支持……410
16.2.1 将列定义为 XML 类型……410
16.2.2 XML 架构集合……411
16.2.3 创建、修改和删除 XML 架构集合……413
16.2.4 XML 数据类型方法……415
16.2.5 强制执行架构集合之外的约束……421
16.2.6 用 XML 格式检索关系数据……421
16.2.7 RAW 选项……422
16.2.8 AUTO 选项……424
16.2.9 EXPLICIT 选项……426
16.2.10 PATH 选项……427
16.2.11 OPENXML 函数……432
16.3 XSLT 简介……436
16.4 本章小结……438

第 17 章 Reporting Services 概述……439
17.1 Reporting Services 101……439
17.2 创建简单的报表模型……440

www.TopSage.com

17.2.1 数据源视图 ……443
17.2.2 建立数据模型 ……447
17.2.3 创建报表 ……450
17.3 报表服务器项目……454
17.4 本章小结……459

第 18 章 使用 Integration Services 进行集成 ……460
18.1 理解问题……460
18.2 使用导入/导出向导生成基本的程序包 ……461
18.3 程序包基础知识……466
18.3.1 “连接管理器”面板 ……467
18.3.2 程序包编辑器面板 ……468
18.4 执行程序包……472
18.4.1 在 Development Studio 中运行程序包……472
18.4.2 使用执行包实用工具 ……472
18.4.3 使用 SQL Server Agent 执行程序包……475
18.4.4 从程序中执行程序包 ……475
18.5 关于程序包的补充说明……475
18.6 本章小结……475

第 19 章 扮演管理员的角色……476
19.1 调度作业……477
19.1.1 创建操作员 ……477
19.1.2 创建作业和任务 ……479
19.2 备份和恢复……487
19.2.1 创建备份——转储 ……488
19.2.2 恢复模式 ……491
19.2.3 还原 ……491
19.3 索引维护……493
19.4 数据归档……495
19.5 基于策略的管理……496
19.6 本章小结……496
19.7 练习……497

附录 A 系统函数……498

附录 B 非常简单的连接示例……550

附录 C 学习资源……558

www.TopSage.com

第 1 章

RDBMS 基础：SQL Server 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是没有什么作用的)。但实际上关系数据库管理系统(Relational Database Management System，RDBMS)不仅仅是数据。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 不仅可以保存数据本身，还可以定义数据类型和数据业务规则。

不要将“数据业务规则”与更通用的业务规则两者相混淆，后者可以应用于整个系统(例如，用户在登录之前不能看到任何数据，或在每月的第一天自动调整财务系统中的本期)。实际上，系统的任意层都可以实施这类规则(目前，在 n 层系统的中间层或客户层已经很常用)。但这里所涉及的业务规则特指与数据本身有关的规则。例如，不存在数额为负的销售订单。使用 RDBMS，可将这些业务规则与数据库本身的完整性集成。

本书主要讲述了保存数据的数据库的概念以及向数据库输入和从中提取数据的最好方法，这是本书全部内容的基础。本章概述了本书其余的内容，所介绍的内容将涵盖本书后面的章节，为学习后面的内容提供一个指导。本章内容包括：

- 数据库对象
- 数据类型
- 保证数据完整性的其他数据库概念

1.1 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)的事物是否满足对象的标准定义，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

www.TopSage.com

- 事务日志
- 索引
- 文件组
- 数据库关系图
- 视图
- 存储过程
- 用户自定义函数
- 用户
- 角色
- 程序集
- 表
- 报表
- 全文目录
- 用户自定义数据类型

1.1.1 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，因此本书不作深究)。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：

如果熟悉 SQL Server，您就可能要问：“登录名或 SQL Agent 任务发生了什么变化？”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 程序包之外，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多作考虑。它们通过 SQL 管理对象(SQL Management Object，SMO)进行编程，不过这一内容超出了本书范围。尽管也有例外之处，但作者通常将它们视为高级内容，将留到高级版中作介绍。

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。存储过程和视图是对保存在数据库表中的数据的特定分组。

在一个数据库中可以存储什么类型的表，而在另一数据库中又可以存储什么类型的表呢？这个问题将在本书的后面章节详细讨论，现在可以简单地认为：属于同一系统或明显相关的数据要存储在同一数据库中。RDBMS(如 SQL Server)在一个服务器上可能有多个用户数据库，当然也可能只有一个用户数据库。单个 SQL Server 上可容纳多少数据库的决定因素有容量(CPU 能力、磁盘 I/O 限制、内存等)、自主权(您希望某个人具有系统所在的服务器的管理权，而其他人也具有另一不同服务器的管理权)或者仅仅是您公司或客户具有多少数据库。有些服务器仅有一个产品数据库，而其他服务器则有多个数据库。另外记住：目前仍在生产的任何版本的 SQL Server (SQL Server 2000 被取代之前已使用了 5 年之久，因此假设大部分软件店有该版本或更高版本的 SQL Server 销售)，都可以在同一个物理服务器上有 SQL Server 的多个实例，并有单独的登录和管理权限。

www.TopSage.com

提示：

有人会问："能否在同一服务器上安装不同版本的 SQL Server，如 SQL Server 2005 和 SQL Server 2008？"答案是肯定的。可以将 SQL Server 2005 和 SQL Server 2008 混合安装。本书不推荐采用这种配置，但是如果有需要的话，可以这样安装。

安装好的 SQL Server 第一次启动时至少包括 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，如果缺少其中任意一个数据库，服务器都无法运行)。根据安装选项的不同，服务器也会不同，可能包括以下样本数据库：

- ReportServer(为 Reporting Server 配置和模型存储需求提供服务的数据库)
- ReportServerTempDB(用于 Reporting Server 的工作数据库)
- AdventureWorks2008 (样本数据库)
- AdventureWorksLT2008 (样本数据库的轻量级新版本)
- AdventureWorksDW 2008(用于 Analysis Services 的样本)

除系统安装的样本数据库外，还可参考一些老版本的样本数据库(通过网络搜索或是其他教程)。

- pubs
- Northwind

提示：

在本书前面的版本中，作者大量使用了老版本的样本数据库。遗憾的是，由于无法保证这些样本数据库一直可供下载，因此作者选择采用新样本数据库。新的 AdventureWorks2008 样本数据库当然更加健壮，并提供了可在 SQL Server 2008 轻松利用的任何示例，但问题在于复杂性。AdventureWorks2008 数据库对于数据库培训来说是极其复杂的。其功能可能只在个案中使用，并且是用作主要特性。因此，作者认为 AdventureWorks2008 并不一定可用作其他类似应用程序中的模板。

1. master 数据库

任意 SQL Server 都有 master 数据库，而不管其是哪一版本或是自定义修改的。master 数据库保存一组特殊的表(系统表)用于系统的总体控制。例如在服务器上新建一数据库，则在 master 数据库中的 sysdatabases 表中将加入该项。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使用于哪一数据库。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。

过去，系统表(包括 master 数据库中的)总在必要时提供系统配置信息，如某些对象在对其执行操作之前是否存在。Microsoft 多年来一直告诫我们不要直接使用系统表，但由于几乎没有其他选择，大部分开发人员忽视了这一劝告。但所幸的是，Microsoft 通过系统和信息模式视图的形式提供了其他选择；现在可在需要时直接利用这些视图获取系统元数据。例如，如果试图

www.TopSage.com

在特定数据库中创建已经存在的对象，系统会提示出错。如果想要解决该问题，则可以测试 sys.objects 表，确认是否存在对应该数据库的表对象的记录项。如果有，则在重新创建之前要删除它。

注意：

如果您很豪爽，就可能会对自己说："我迫不及待就想去弄乱系统表！"不要尝试那样做。实际上以任何形式使用系统表都非常危险。Microsoft 不能绝对保证不同版本的 master 数据库的兼容性，而事实上，master 数据库几乎都被更改。不过有几个对象(如系统函数、系统存储过程和信息模式视图)可用于检索存储在系统表中的大部分元数据。

虽说如此，有时还是必须使用系统表，但通常还是最好还是敬而远之，避免去碰它。

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 数据库构成任何新建数据库的模板。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 数据库作为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：第一，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，强烈建议不要修改这一项。

3. msdb 数据库

msdb 数据库是 SQL Agent 进程存储任意系统任务的地方。如果计划对一数据库每夜进行备份，则在 msdb 数据库中有一个记录项。如果每次执行任务时调度存储过程，则在 msdb 数据库中会有一个记录项。SQL Server 中的其他大部分子系统以类似的方式使用 msdb。SSIS 程序包和基于策略的管理的定义就是使用 msdb 的进程的示例。

4. tempdb 数据库

tempdb 数据库是服务器的主要工作区域之一。在执行一个复杂或者大型的查询操作时，SQL Server 可能需要创建一些中间表来完成，而这是在 tempdb 数据库中进行的。只要创建临时表，这些表就会创建在 tempdb 数据库中，即使您是在当前数据库中创建这些表的(别名是在本地数据库中创建以供引用的，而物理表在 tempdb 中创建)。只要需要临时存储数据，则很可能是将数据存储在 tempdb 数据库中。

tempdb 数据库与其他任意数据库不同。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，tempdb 数据库是系统中唯一完全重建的数据库。

注意：

从技术上来说，可以在 tempdb 数据库中创建自己的对象，但本人强烈反对进行该操作。可以在系统中任意有权访问的数据库中创建临时对象，该对象也会存储在 tempdb 数据库中。

www.TopSage.com

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。这也是不要尝试做的一件事情。

5. ReportServer 数据库

这个数据库只有在安装了 ReportServer 的情况下才存在(它不必与数据库引擎相同，但注意，如果是不同的服务器，它要求单独的许可)。ReportServer 数据库存储 Reporting Server 实例的任何持久化元数据。注意，它是只可用于给定 Reporting Server 实例的操作性数据库，只能通过 Reporting Server 修改或访问。

6. ReportServerTempDB 数据库

除了存储非持久化数据(如有关正在运行的报表的数据)外，该数据库的基本作用与 ReportServer 数据库相同。同样，它也是个纯粹操作性的数据库，只能通过 Reporting Server 访问或修改。

7. AdventureWorks2008 数据库

在有 AdventureWorks2008 数据库之前，SQL Server 就包括了其他样本数据库。老样本数据库有不足之处，如它们包含了一些不好的设计习惯。此外，老的例子过于简单，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。这里不再探讨 AdventureWorks2008 数据库是否存在相同问题。AdventureWorks2008 数据库确实试图解决这些问题。

在 SQL Server 2005 发展之初，Microsoft 就想要一个更加健壮的样本数据库，使其尽可能作为其产品的一个示例。AdventureWorks2008 数据库就是这一想法的结果。尽管该数据库对于初学者过于复杂，但是其作为数据库示例确实是一杰作。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特征的部件。从这一方面看，该数据库是非常不错的。

AdventureWorks2008 是中心数据库，本书将广泛使用该数据库。

8. AdventureWorksLT2008 数据库

这里的 LT 表示轻量级(lite)。它只是 AdventureWorks2008 数据库完整版的极小一部分。该思想提供了更简化的样本集，便于理解基本概念和完成简单的练习。尽管本人并不知晓推出这一新样本集的真正原因，但我猜测是为了取代较老的 Northwind 和 Pubs 样本集(相较于 AdventureWorks2008 样本集，许多培训者更喜欢采用 Northwind 和 Pubs 样本集，因为 Adventures2008 数据库对于初级培训来说通常过于复杂和笨重)。

9. AdventureWorksDW2008 数据库

AdventureWorksDW2008 数据库是 Analysis Services(分析服务)样本数据库(DW 表示数据仓库(Data Warehouse)，大部分 Analysis Services 项目是建立在数据仓库的基础上)。Microsoft 预见

www.TopSage.com

性地将分析样本数据库与事务样本数据库联系在一起，以提供展示两者协同运行的完整样本数据库。

因为决策支持数据库远远超出本书的范畴，所以您不必使用决策支持数据库，只是在启动 Analysis Services 时接触该数据库。比较一下两个数据库之间的区别。虽然它们服务于同一个虚构的公司，但是它们具有不同的目的。

10. pubs 数据库

pubs 数据库是我们所熟悉的。它是原来的样本数据库之一，在 SQL Server 2005 之前，作为安装数据库的一部分随 SQL Server 一起提供。pubs 数据库是目前仅能从 Microsoft 网站上单独下载安装的样本数据库。当前的许多培训资料和书籍仍涉及该数据库，但 Microsoft 并未对其还可用的时间作出承诺。pubs 数据库与 SQL Server 操作绝对无关，它仅仅是为培训或试验提供一个一致的环境。虽然读者并不需要用 pubs 数据库来操作本书中的示例，但可能会希望下载并安装它来练习网上的其他一些示例和教程。

11. Northwind 数据库

如果您有 Access 或者 Visual Basic 方面的编程经验，就很可能已经对 Northwind 数据库不陌生了。Northwind 数据库是从 SQL Server 7.0 版本开始被添加进来的，但是在 SQL Server 2005 的基本安装中被删除了。与 pubs 数据库类似，Northwind 数据库的下载可与 SQL Server 的基本安装分开进行(其下载和安装与 pubs 样本数据库的下载和安装一并进行)。和 pubs 数据库一样，Northwind 数据库也不用于练习本书中的示例，但可很方便地用于练习网上的其他一些示例和教程。

1.1.2 事务日志

不管您相信与否，数据库文件本身并不是“事情发生最多的场所”。数据当然要从数据库文件读取，但实际上，任意数据库的更改起初不进入数据库本身，而是不断地被写入到事务日志。在后面的某个时间点上，数据库发出检查点，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。

数据库为随机访问配置，但是日志是自然连续的。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。日志不断积累已经确认了的更改，并且一次性地将几次更改写入到物理数据库文件。

第 14 章将深入讨论更改如何写入到日志，现在要记住的是：日志是数据进入磁盘上的最先位置，且稍后会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。

1.1.3 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。表类似会计账簿或 Excel 表格。表由称为域的数据(列)和实体数据(行)构成。数据库中实际的数据都存储在表中。

www.TopSage.com

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。

以 AdventrueWorks2008 数据库中的 Production.Location 表为例(如图 1-1 所示的视图来自 SQL Server Management Studio。SQL Server Management Studio 是 SQL Server 的基本工具，本书第 2 章将学习该工具的使用)。

	LocationID	Name	CostRate	Availability	ModifiedDate
1	1	Tool Crib	0.00	0.00	1998-06-01 00:00:00.000
2	2	Sheet Metal Racks	0.00	0.00	1998-06-01 00:00:00.000
3	3	Paint Shop	0.00	0.00	1998-06-01 00:00:00.000
4	4	Paint Storage	0.00	0.00	1998-06-01 00:00:00.000
5	5	Metal Storage	0.00	0.00	1998-06-01 00:00:00.000
6	6	Miscellaneous Storage	0.00	0.00	1998-06-01 00:00:00.000
7	7	Finished Goods Storage	0.00	0.00	1998-06-01 00:00:00.000
8	10	Frame Forming	22.50	96.00	1998-06-01 00:00:00.000
9	20	Frame Welding	25.00	108.00	1998-06-01 00:00:00.000
10	30	Debur and Polish	14.50	120.00	1998-06-01 00:00:00.000
11	40	Paint	15.75	120.00	1998-06-01 00:00:00.000
12	45	Specialized Paint	18.00	80.00	1998-06-01 00:00:00.000
13	50	Subassembly	12.25	120.00	1998-06-01 00:00:00.000
14	60	Final Assembly	12.25	120.00	1998-06-01 00:00:00.000

图　1-1

如图 1-1 所示的表由 5 列数据构成。与表中的数据的多少(即使是 0)无关，列的数目是不变的。当前图 1-1 所示的表有 14 条记录。在增删数据时，表中记录的数目会随之增减，但是每一记录(或行)的数据类型由列的数据类型定义。

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。索引分为如下两类。

- 群集索引——每个表只能有一个群集索引。如果是群集索引，其含义为：群集索引对应的表按照其索引进行物理排序。如果为一百科全书做索引，则群集索引是书的页码；按页码顺序保存百科全书中的信息。
- 非群集索引——每个表可以有多个非群集索引。非群集索引的含义与您在平时听到的“索引”的含义更接近。非群集索引是让您找到数据的一些其他值。对于百科全书，非群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)必须在有非群集索引之前至少有一个群集索引。

www.TopSage.com

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，但主要用于在输入时复制数据或更新时检查数据，确保数据满足相应标准。

3. 约束

约束是仅限于表中存在的另一对象。顾名思义，约束就是限制表中数据满足某种条件。约束在某种方式上类似触发器，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。

1.1.4 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，并不一定要保持这样的配置。

SQL Server 允许定义约 32 000 个辅助文件(如果需要更多的辅助文件，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。仅能有一个主要文件组，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。

1.1.5 数据库关系图

在讲述规范化和数据库设计时会详细讲述数据库关系图，现在只要清楚数据库关系图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(比如“供应”和“购买”)。

提示：

遗憾的是，SQL Server 2008 内含的数据库设计工具保留的很少。实际上工具使用的数据库关系图方法并未遵循 ER 数据库关系图中公认的标准。但这些数据库关系图工具事实上提供了所有的“必须”内容，因此也可从它们入手。

如图 1-2 所示的数据库关系图表示 AdventureWorks2008 数据库中的各表之间的关系。该数据库关系图也描述了数据库的许多其他特征(如果您是第一次接触它，可能会注意不到)。注意标识键的小图标和无穷大符号。这些符号描述了两张表之间的本质关系。本书第 7、8 章会详细讲述关系，而且在本书的后面部分会深入讲述数据库关系图。

www.TopSage.com

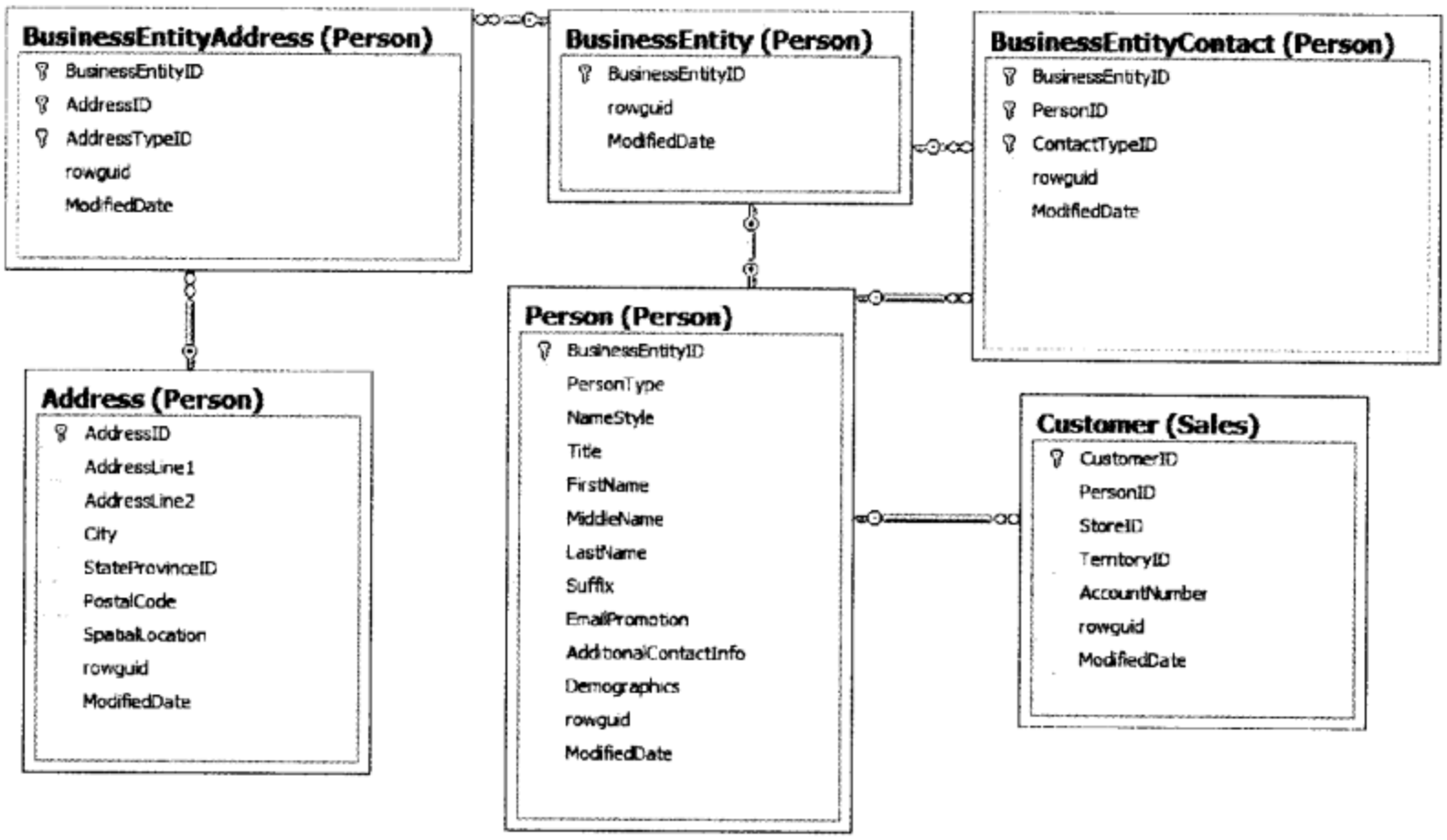


图　1-2

1.1.6　视图

视图是一种虚拟表。除了视图不包含本身的任意数据外，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的一个预先计划的映射和表示。视图以查询的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。

SQL Server 2000 版本之后，应用视图的主要目的是控制用户所要显示的数据。这有两方面的作用：安全和易于使用。使用视图可以控制用户所见，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)，则可以创建允许任何用户访问的只包含这两列数据的视图。此外，可以剪裁视图，使得用户不必搜索任何不必要的信息。

除了这个最基本的用途之外，还能够创建所谓的索引视图(indexed view)。索引视图除了可以创建索引之外，其他都与视图相同。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，索引视图可读性更好，因为它预先构造了表之间的连接。
- 由于预先计算好视图中执行的聚合(aggregation)，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取。
- 由于视图上的索引必须及时更新，因此插入和删除索引需要更多的系统开销；如果更新影响到索引的键列或群集键，则更新也需要更多的系统开销。

本书第 10 章将深入探讨索引视图对系统性能的影响。

www.TopSage.com

1.1.7 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 Microsoft SQL Server 数据库查询的语言)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。与向服务器发送单条语句相比，使用存储过程有以下几个优点。

- 不使用长字符串文本而使用短存储过程名，可减少存储过程中的运行代码所要的网络信息流量。
- 预先优化和预编译，节省存储过程每次运行的时间。
- 通常考虑安全原因，或仅仅是隐藏数据库的复杂性，将过程封装。
- 可从其他存储过程调用，使得它们在某些限定意义上重用。

尽管存储过程是 SQL Server 编程功能的核心，但也要小心使用它。通常情况下，它们是解决方案，但通常也不是唯一的解决方案。在选择存储过程作为解决方案前，要确保它是正确的选择。

1.1.8 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。不包括的返回值类型有：text、ntext、image、cursor 和 timestamp。
- 没有“副作用”。基本上，用户自定义函数不能越俎代庖，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，并且有一个值输出。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。UDF 输入值的传递与变量按引用传递或是指针的传递不同。但是 UDF 可以返回一种特殊的数据类型——表。第 13 章将深入讲述该部分内容。

1.1.9 用户和角色

用户和角色关系密切。用户(user)等价于登录名。简言之，该对象表示登录 SQL Server 的标识符。登录 SQL Server 的任何人都映射(直接或间接映射，取决于使用的安全模型)到一个用户。用户依次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。

1.1.10 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了规则，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条件。与约束不同，规则不限于特定表。它们是独立对象，可绑定到多个表，或者甚至绑定到特定数据类型(从而间接用于表中)。

目前，Microsoft 对于一些版本已考虑弃用规则。规则应该考虑向下兼容，而在新的开发中应该避免使用。

www.TopSage.com

提示：

考虑到 Microsoft 已在 SQL Server 2008 中引入一些新的弃用处理功能，我怀疑在一些版本中弃用的功能(如规则)将最终在 SQL Server 下一版中删除。因此，我要强调一下，在新开发中不要使用规则。事实上，很早前就开始移除它们了。

1.1.11　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。第 6 章将深入研究两种默认值的类型。

1.1.12　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或.NET 程序集中方法定义的复杂数据类型的扩展。用户自定义数据类型几乎可定义任意数据。虽然 SQL Server 2000 及其更早版本也有用户自定义数据类型，但是它们被限制在现有的数据类型的筛选中。自 SQL Server 2005 发行后，可以将.NET 程序集绑定到您自己的数据类型上，即能够定义一种数据类型，用于存储(在合理范围之内).NET 对象可以存储的任意对象。事实上，SQL Server 2008 中新添加的空间数据类型(Geographic 和 Geometric)就是使用基于.NET 程序集的用户自定义类型实现的。Wrox 出版社出版的由 Robert Vieira 编写的 *Professional SQL Server 2008 Programming* 一书详细介绍了.NET 程序集。

提示：

要谨慎使用用户自定义数据类型！你所使用的数据类型对于数据和数据存储来说非常重要。虽然能够定义任意对象的功能很优秀，但需要大量性能开销。所以要考虑清楚，确信是需要自定义数据类型，然后再进行多次测试！

1.1.13　全文目录

全文目录是数据映射，用于加速列中特定文本块的搜索，该列需要启动全文搜索功能。在 SQL Server 2008 之前，全文目录存储在数据库外部(因此导致一些重大的备份和恢复问题)。从 SQL Server 2008 开始，全文目录已集成到主数据库引擎和存储机制中。全文目录的内容超出了本书的范围，但在 *Professional SQL Server 2008 Programming* 一书中有详细介绍。

1.2　SQL Server 数据类型

在熟悉了 SQL Server 数据库中的基本对象之后，下面介绍 SQL Server 中的数据类型。注意：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，所以这里假设您已经清楚数据类型的工作原理，只介绍特定的 SQL Server 数据类型。

SQL Server 2008 所有自带的数据类型如表 1-1 所示。

www.TopSage.com

表 1-1

数据类型名称	类　别	长度(以字节为单位)	数据特点
Bit	整型	1	表中的第 1 个 Bit 数据类型占 1 个字节；其余 7 个位也用作 Bit 数据类型。允许空格使其占用一个额外字节
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}~$2^{63}-1$
Int	整型	4	取值范围为 -2 147 483 648 ~ 2 147 483 647
SmallInt	整型	2	取值范围为 -32 768 ~ 32 767
TinyInt	整型	1	取值范围为 0~ 255
Decimal/Numeric	数字型	可变	固定精度，取值范围为 $-10^{38}-1$ ~ $10^{38}-1$。两者含义相同
Money	货币	8	货币单位，取值范围为 -2^{63} ~ 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	4	货币单位，取值范围为 -214 748.3648 ~ +214 748.3647
Float (Real)	近似小数	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 ~ 1.79E + 308
DateTime	日期/时间	8	日期与时间，取值范围为 1753 年 1 月 1 日~ 9999 年 12 月 31 日，精确到 0.03 秒
DateTime2	日期/时间	可变(6~8)	新扩展的 DateTime 典型数据类型。支持更大的日期范围和更高的时间部分精度(精确到 100 纳秒)。和 DataTime 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	4	日期与时间，取值范围为 1900 年 1 月 1 日~2079 年 6 月 6 日，精确到分钟
DateTimeOffset	日期/时间	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	3	只存储 Gregorian 日历定义的 0001 年 1 月 1 日~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

www.TopSage.com

(续表)

数据类型名称	类	长度(以字节为单位)	数据特点
Time	日期/时间	可变(3~5)	只存储用户可选的精度为纳秒(默认)的时间数据
Cursor	特殊小数	1	指向游标的指针，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	特殊小数(二进制)	8	给定数据库的唯一特定值。即使 UPDATE 语句没有 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设定(不允许直接更新 timestamp 字段)
UniqueIdentifier	特殊小数(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时间内的唯一
Char	字符	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	可变	长度可变的字符数据。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，使用 max 关键字表示其长度可足够大(数据长度可达 2^{31} 字节)
Text	字符	可变	从 SQL Server 2005 开始支持向后兼容。可使用 varchar(max)代替
NChar	Unicode	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	可变	可变长度的 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，使用 max 关键字表示其长度可足够大(数据长度可达 2^{31} 字节)
Ntext	Unicode	可变	可变长度的 Unicode 字符数据。类似 Text 数据类型，仅用作向后兼容。可使用 nvarchar(max)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	可变长度二进制数，最大特定长度为 8 000 字节，可使用 max 关键字使其作为 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 开始支持向后兼容。可使用 varbinary(max)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)的结果输出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

www.TopSage.com

(续表)

数据类型名称	类	长度(以字节为单位)	数据特点
HierarchyID	其他	特殊	维护层次结构位置信息的特殊数据类型。提供特定于层次结构需要的特殊功能。允许作深度、父/子关系和索引比较。实际尺寸随层次结构中的节点数和平均深度而变
Sql_variant	其他	特殊	与 VB 和 C++中的变量基本无关。其实质是用于保存大多数其他 SQL Server 数据类型的容器。当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 不同的是，使用这种数据类型要将其显式转换为更具体的数据类型
XML	字符	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
CLR	其他	可变	随 CLR 对象的特性而变，CLR 对象支持基于自定义数据类型的 CLR

表 1-1 中的数据类型与其他编程语言中的数据类型相同。例如 SQL Server 中的整型(int)等价于 Visual Basic 中的长整型(long)，对于 C++中的大部分系统和编译器组合来说，它等价于有符号整型。

注意：

SQL Server 中没有无符号数值数据类型。

通常 SQL Server 数据类型与绝大多数其他现代编程语言的使用类似。数的相加为和，而字符串的相加为字符串的连接。如果混合了不同数据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 1-3 包含了所有可能的不同转换。

为什么要进行数据类型转换？下面举一个简单的示例。如果要输出“Today's date is ##/##/####”的短语，其中##/##/####表示当前日期，则可能进行如下编程：

```
SELECT 'Today''s date is ' + GETDATE()
```

提示：

本书后面还将更加详细地讲解 Transact-SQL 语句，但是上述示例的输出结果显而易见。

问题是该语句生成如下结果：

www.TopSage.com

	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 CAST，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 1-3

```
消息 241，级别 16，状态 1，第 1 行
从字符串转换日期/或时间时，转换失败。
```

这并不是真正想要的结果。下面使用 CONVERT()函数：

```
SELECT 'Today''s date is ' + CONVERT(varchar(12), GETDATE(),101)
```

该语句输出的结果为：

www.TopSage.com

```
-----------------------------------
Today's date is 01/01/2008
(1 行受影响)
```

像 GETDATE()函数输出的日期与时间(date/time)数据类型，不能自动转换成字符串数据类型，如“Today's date is ”，但是经过转换规则可以进行这些转换。CAST 和 CONVERT()函数可将 SQL Server 的许多数据类型相互进行转换。本书后面的章节会讲述 CAST 和 CONVERT()函数。

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。数据类型可以避免编程的错误，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)，而且确保可执行期望的操作。

NULL 数据

如果表中有一行的某一列没有任何数据，您是否知道该列的值？比如，让一记录保存某一年的公司业绩信息，假设字段之一为公司业绩比前一年的增长百分比，但是在数据库中没有上一年的业绩信息记录，则可能在 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%。而实际上是不知道上一年的业绩信息。

不确定的值可设置为 NULL。几乎在我每一次上编程课时，都至少有一个学生问我怎么定义 NULL 值。这是一个很好的问题，因为根据定义，NULL 值也就是不确定的值。它可能是 1，也可能是 347，还可能是 - 249 等。简而言之，NULL 的含义是未定义，或者是不能应用的值。

1.3 SQL Server 对象标识符

到这里为止，本书介绍了与 SQL Server 中对象相关的内容。下面介绍 SQL Server 中对象的命名。

1.3.1 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。

- Stored procedures(存储过程)
- Tables(表)
- Columns(列)
- Views(视图)
- Rules(规则)
- Constraints(约束)
- Defaults(默认值)
- Indexes(索引)
- Filegroups(文件组)
- Triggers(触发器)
- Databases(数据库)
- Servers(服务器)
- User Defined Functions (用户自定义函数)
- Logins(登录名)
- Roles(角色)
- Full-text catalogs(全文目录)
- Files(文件)
- User Defined Types (用户自定义类型)

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。命名的技巧是使名称有用且实际。

www.TopSage.com

1.3.2　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所示。

- 对象的名称必须以 Unicode 3.2 规范定义的任意字母开头。包括大部分西文字母 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。名称中除了开头字母之外的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 SET QUOTED_IDENTIFIER 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。

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，但如果是 SQL Server 的本地化版本(适用于特定语言、方言或地区)，则会稍有不同。对于特定对象类型存在其他规则。

说明：

这里值得注意的是对象的命名。SQL Server 可以在命名中嵌入空格，甚至有时可以使用关键字来命名。注意，要避免这种命名方式！在使用 SELECT 语句时，嵌入空格的列名可能使得标题更好看，但还有其他方法同样能达到这种效果。在列名中使用内嵌空格或关键字容易引起故障、混淆，甚至其他严重后果。后面会解释为什么 Microsoft 允许这种命名，这里只需要记住在命名中使用内嵌空格或关键字等于自找麻烦(本书会多次提到这一点)。

1.4　本章小结

许多小的数据库对象造就了 RDBMS 的强大功能。几乎所有的人(只要是看到本书)都能想到使用列和行的方式来存储数据，即使他们还不清楚这些列和行的组合应该称之为表。但是少量表却不能构成真正的数据库。目前的 RDBMS 之所以如此强大，是因为数据库对象使您能将与数据相关的功能和业务规则放到带有数据的数据库中。

数据库数据与其他大部分编程环境一样，也具有数据类型。在 SQL Server 的编程中要对数据类型作必要的考虑。复习一下可用的数据库数据类型，思考该数据类型如何变换到您熟悉的任何编程环境中相对应的数据类型。

www.TopSage.com

第 2 章

SQL Server 管理工具

第 1 章介绍了 SQL Server 中各种类型的对象，本章介绍如何找到这些对象，以及如何监控系统。

本章将探究为 SQL Server 的基本引擎(即关系数据库引擎)提供服务的工具，它们用于管理其他章节中将详细介绍的插件服务。其中部分工具只能完成非常专业的一小部分功能；另外一部分工具可以完成许多不同的功能。大部分管理工具已经在 SQL Server 中以一种或另一种形式使用了很长时间。

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 集成服务(SQL Server Integration Services，SSIS)，包括输入/输出向导
- Reporting Services 配置管理器
- Bulk Copy Program(bcp)
- Profiler
- Sqlcmd
- PowerShell

2.1 联机丛书

联机丛书也被视为工具吗？确实，本书中是这样认为的。无论您读过多少遍本书或者其他有关 SQL Server 的书，您都不可能记住 SQL Server 中所有需要了解的内容。虽然 SQL Server 是我经常使用的软件产品之一，但我也无法记住所有的内容。联机丛书是 SQL Server 中最重要的工具之一。

提示：

我非常讨厌与编程有关的书或其他参考资料。大约 1980 年左右我就开始编程，而当时可能记住绝大部分内容(但不是所有的内容)。但今天已经不可能记住那么多内容。如果有一点变化(某些内容本身当时就很难)，就会有太多的内容需要记住，而那些不是每天使用的内容肯定会逐渐淡忘。

www.TopSage.com

这里有条建议：不需要记住所有的内容。只要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)，并且在收藏夹里保存一系列优秀的 SQL Server 网址来补充不经常使用的内容和某些细节。

如图 2-1 所示，SQL Server 中的联机丛书使用最新的.NET 联机帮助界面，代替了以前在 Microsoft 技术产品系列(Back Office、MSDN 和 Visual Studio)中使用的联机帮助界面。

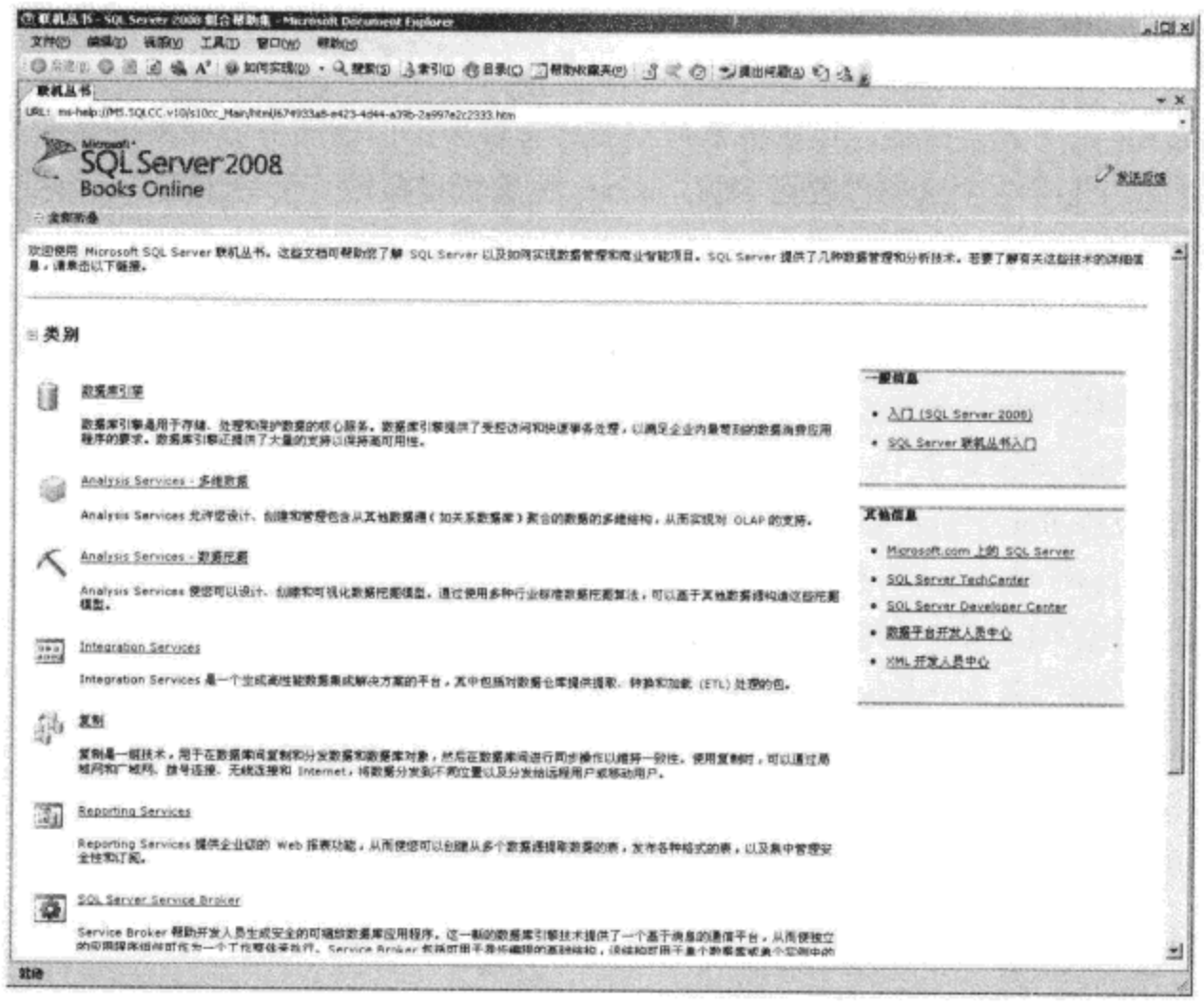


图　2-1

帮助系统的使用都是为我们所熟悉的，因此这里不作详细介绍。无论使用什么计算机，SQL Server 联机丛书都随时可提供快速的参考帮助。

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兆字节的价格，BOL 其实不会占用太多的空间，并且无论何时运行 SQL Server，都可以快速找到参考资料，从而能节省大量时间(在计算机上安装联机丛书约占用 110MB 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的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成为一个设置工具。配置管理器的管理项目分为两部分：

www.TopSage.com

- 服务管理
- 网络配置

提示：

尽管我在这里只列出了两个主要部分，但实际配置管理器将网络配置端的内容又分为多个节点。

2.2.1 服务管理

SQL Server 是一大型产品，在服务器后台要运行许多不同的服务。完整安装的 SQL Server 包括 9 个服务，其中 7 个服务可使用 SQL Server 配置管理器这一部分来管理(另两个是作为后台支持的服务)。

可管理的这些服务包括：

- 集成服务(Integration Services)——支持第 18 章将介绍的 Integration Services 引擎。
- 分析服务(Analysis Services)——支持 Analysis Services 引擎。
- 全文(Full Text)目录——支持文本搜索功能。
- 报表服务(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，可以按照不同调度安排作业。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例程输入与输出任务。
- 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 Browser——支持通告服务器，通过浏览局域网可以确认系统是否安装了 SQL Server。

2.2.2 网络配置

很多时候，发现的网络连接问题是因客户机网络配置不合理，或者是客户机网络配置与服务器配置不匹配引起的。

SQL Server 提供了几种网络库(Net-Librariy，NetLib)。它们是 SQL Server 用来与某些网络协议通信的动态链接库(Dynamic-Link Library，DLL)。NetLib 作为客户应用程序与网络协议之间的“绝缘体”，而网络协议实质是用于在网卡之间相互通信的语言。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)
- TCP/IP(默认协议)
- 共享内存(Shared Memory)
- VIA(硬件存储器供应商可能支持的特殊虚拟接口)

提示：

VIA 是专门用于特殊的(很贵的)硬件的特殊网络库。如果运行在 VIA 环境，则要清楚与其相关的特殊需要。对于不运行在该环境的系统，VIA 为服务器之间的高速通信提供快速但昂贵的通信方案。它通常不用于普通的客户机。

在客户机与服务器计算机上，相同的 NetLib 必须都可用，这样它们可以通过网络协议彼

www.TopSage.com

此进行通信。选择在服务器上不支持的客户机 NetLib 会导致通信连接失败，返回“Specified SQL Server Not Found”消息。

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)，通常是驱动程序与 NetLib 通信。通信过程原理如图 2-2 所示。步骤如下：

(1) 客户应用程序与驱动程序通信(SQL Native Client、ODBC)。

(2) 驱动程序调用客户机 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 将客户机的请求传送到 SQL Server。

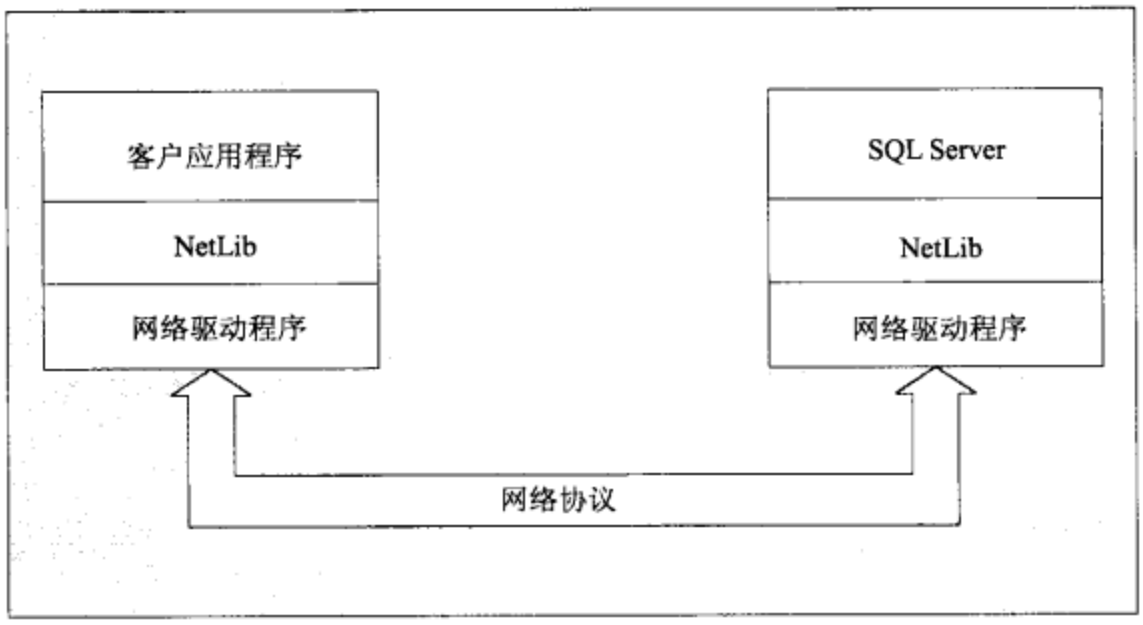


图　2-2

注意：

如果您熟悉 TCP/IP，就知道 IP NetLib 将侦听的默认端口是 1433。端口可看作为无线信号的频道——信号在不同频率上跳跃，但只有在“听到”需要的频道时对您才有用。注意，这是默认的，因此无法保证您所尝试连接的特定服务器也侦听这个特定端口——事实上，大部分安全专家建议将它改为非标准的端口。

从 SQL Server 到客户机的回复按照同样的顺序进行，只是通信方向相反。

2.2.3　协议

首先看一下可用的选项。如果运行 Configuration Management 实用程序，打开 SQL Server 网络配置树，将显示如图 2-3 所示的窗口。

提示：

这里显示多少个节点取决于安装程序。在图 2-3 中，网络配置和客户机配置都有重复节点，分别针对 32 位和 64 位库。如果运行 32 位安装，则每种配置只有一个节点。

注意：

出于安全的考虑，在安装时只启用 Shared Memory。

您可能希望在本地访问机器时启用 Shared Memory(这只在客户机与 SQL Server 安装位于同一物理服务器上时起作用)。但如果想要远程连接 SQL Server(如从 Web 服务器或网络中不同的

www.TopSage.com

客户机)，则需要至少启用另一个 NetLib。

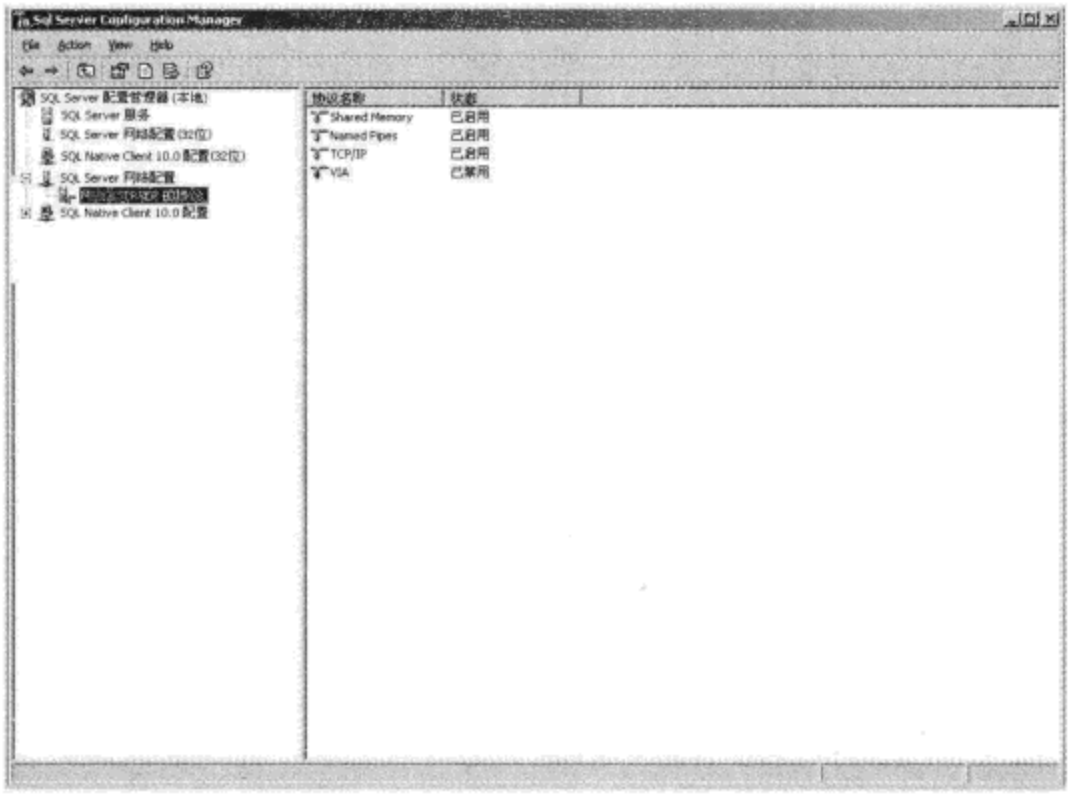


图 2-3

记住，为了让客户机与服务器进行连接，服务器必须侦听协议，而客户机使用该协议试图与服务器通信，如果是 TCP/IP 协议，则要在相同端口上。

提示：

此时您可能要问："为什么不启用所有的 NetLib？然后就可以高枕无忧了。"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)，而且降低系统安全性。

接下来看一下支持的协议以及选择特定协议的原因。

1. 命名管道

在 TCP/IP 不可用或者没有域名服务(Domain Name Service，DNS)服务器用于对 TCP/IP 下的服务器命名时，命名管道非常有用。

注意：

从技术上来说，可以使用其 IP 地址连接到运行 TCP/IP 的 SQL Server，而不使用服务器名。即使没有 DNS 服务，只要有从客户机到服务器的路由，该方法就始终有效(如果它有 IP 地址，则不需要名称)。记住，如果 IP 地址因为某种原因改变了，则需要更改访问的 IP 地址(如果有大量配置文件需要更改是很令人头疼的)。

2. TCP/IP

TCP/IP 协议已经成为事实上的标准网络协议。如果想要通过 Internet 直接连接到 SQL Server，则 TCP/IP 协议是唯一的选项。

www.TopSage.com

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)，则要特别注意数据服务器的安全防范。

3. 共享内存

共享内存减少了进程间编组的需要。编组是将信息传送到进程边界之前将信息打包的方法，如果运行在相同的区域，则进程间的边界是指服务器和客户机之间的边界。客户机可直接访问服务器存储数据的同一内存映射文件。这样可以大量减少系统开销，而且访问速度很快。共享内存只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。

2.2.4　客户端

以上介绍了所有可能的协议以及如何选择各协议。只要知道服务器提供的内容，就可以配置客户机。多数情况下，默认的设置就能正常工作，这里查看如何配置。展开“客户端网络配置”树，然后选择“客户端协议”节点，如图 2-4 所示。

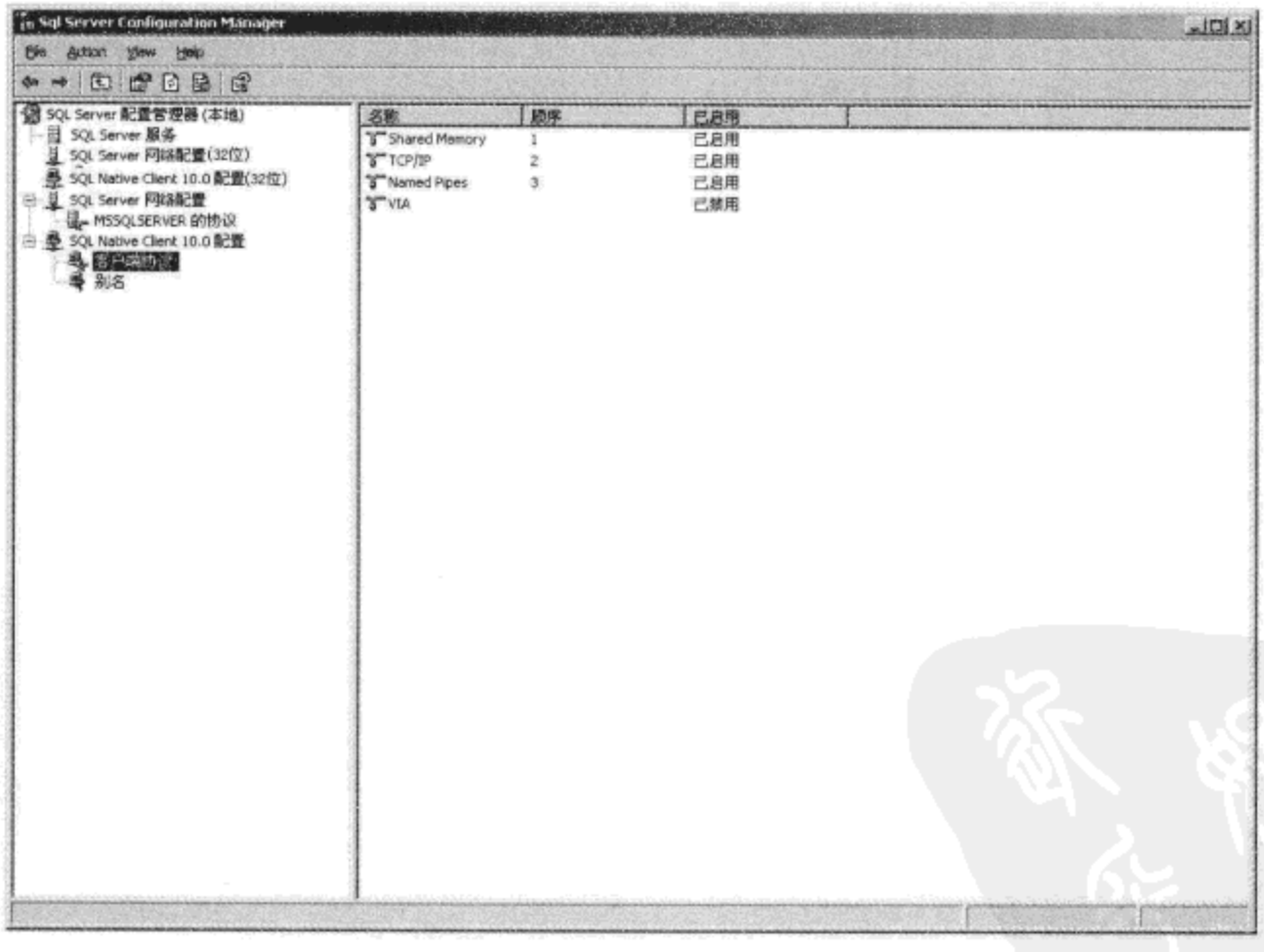


图　2-4

SQL Server 提供了一项功能：即客户机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

www.TopSage.com

来更改使用协议的优先级)，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 Netlib(“客户端网络配置”的下一节点的别名列表)，然后是 TCP/IP 协议等。

注意：

如果网络支持 TCP/IP 协议，则对于任何远程访问，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。但该协议对本地服务器(即该服务器与客户机在同一物理系统中)没有意义，如果不需要通过网络栈来查看本地 SQL Server，则使用 Shared Memory NetLib 速度更快。

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。可以使用 IP 来连接服务器，也可以使用 Named Piped 来连接服务器，而不管是需要连接哪一个服务器。图 2-5 显示了一台客户机被配置为使用 Named Pipes NetLib 来连接名为 HOBBES 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。

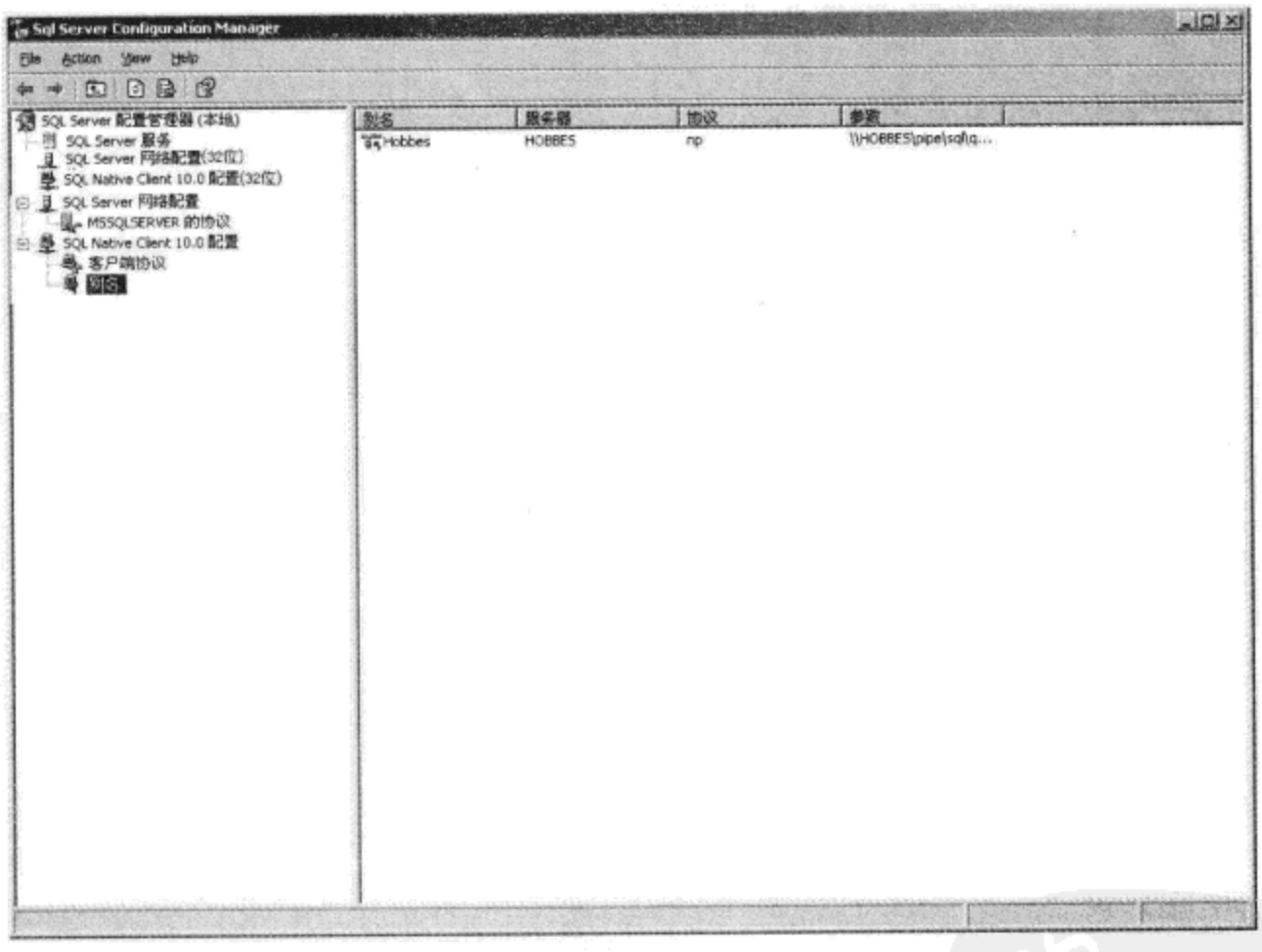


图 2-5

记住，设置在网络机器上的 Client Network Configuration 必须有一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。

提示：

如果通过 Internet 连接到 SQL Server(从安全的角度看，这不是个好主意，但还是有人这样做)，则您想要用的是服务器的实际 IP 地址，而不是服务器名。在遇到 SQL Server 与 Internet 时会涉及到域名解析的问题。记住，如果服务器有新的 IP 地址，则要手动更改 IP 地址，DNS 不会自动更改 IP 地址。

www.TopSage.com

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。该工具提供了通过相对易于使用的图形用户界面来管理服务器的各种功能。作为 Visual Studio IDE 环境的一个子集，该工具集成了在独立工具中使用的多个功能。

由于本书的目的是面向开发人员，因此不涵盖 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 程序包)
- 显示当前活动(如登录用户、锁定的对象，以及在哪个客户端运行)
- 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
- 建立并管理全文搜索目录
- 管理服务器的配置设置
- 启动一个新的 PowerShell 控制台实例
- 建立并管理数据库副本的发布与预订

由于本书会大量使用 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

第一次启动 Management Studio 时，会出现类似图 2-6 所示的“连接”对话框。

图　2-6

您所看到的登录界面可能与图 2-6 所示的界面有所不同，这取决于之前是否已经登录，登录了哪台机器，以及登录所使用的登录名。登录界面中的大部分选项都可顾名思义，这里深入介绍其中两个选项。

1. 服务器类型(Server Type)

该选项关系到要登录的 SQL Server 各个子系统(标准的数据库服务器、分析服务、报表服务或集成服务)。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(Server Name)

您可能猜到，该选项表示要登录的那一个 SQL Server。如图 2-6 所示，我们选择了“.”。该选择不表示服务器名为“.”，而指要登录的是在本机上的默认 SQL Server 实例，并不管本机如何命名。选择“.”(local)不仅自动确定了您想要使用的服务器(与实例)，而且确定了登录的方式。也可以使用“(local)”代替“.”。

www.TopSage.com

注意：

SQL Server 允许同时运行多个 SQL Server 实例。不过这些实例单独加载到彼此独立运行的 SQL Server 引擎的内存中。

注意，服务器默认的实例与网络上的机器名相同。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，重则服务器没有反应。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 HOBBES 或 KIERKEGAARD)，然后在其后加一美元符号($)，如实例名为 SIDDARTHA$SHRAMANA。

如果选择了“.”或(local)，则系统使用共享内存 NetLib，而不管您连接其他服务器时选择哪一个 NetLib。这样做有利有弊。不利的是放弃了一小部分控制权(SQL Server 将总是使用共享内存来连接服务器，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获得高性能选项。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，这与您在与其他系统通信时是一样的，而不管是否在本机。

图 2-7

如果记不住服务器的名称，该如何操作？只要单击服务器列表框右边的下拉箭头，可显示最近连接的服务器名列表。向下滚动，可以看到“浏览更多”选项，单击该选项，SQL Server 将获得在网络上“登记”的任意服务器。如图 2-7 所示，出现两个选项卡：一个用于显示本地服务器(正在运行的相同系统上的 SQL Server 实例)，而另一选项卡显示在网络上运行的其他 SQL Server。

选择其中的任意服务器，然后单击“确定”按钮。

注意：

在使用 Server selection 对话框时要小心。虽然通常非常可靠，但是有多种使得 SQL Server 并不在网络上广播的配置方法。当服务器按照这种方法配置时，服务器名将不会显示在列表中。而且，仅仅侦听 TCP/IP NetLib 且没有 DNS 入口的服务器也不会显示在列表中。这种情况下您必须知道 IP 地址，再使用 IP 地址来找到服务器。

3. 身份验证

可以选择 Windows 身份验证或 SQL Server 身份验证。不管如何配置服务器，即使配置为 SQL Server 身份验证，Windows 身份验证也通常有效。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)登录名仅在打开 SQL Server 身份验证时有效。

1) Windows 身份验证

Windows 身份验证针对 Windows 用户和组。这些 Windows 用户在其 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，他们在整个 Windows 域内有效，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。

www.TopSage.com

该模型的最佳之处在于仅有一个密码(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，仅仅从您当前登录 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中的管理用户。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。

2) SQL Server 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 SQL Server 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。

试一试　连接服务器

现在进行登录。

(1) 为 SQL Server 选择(local)选项。

(2) 选择 SQL Server 身份验证。

(3) 选择登录名为 sa，记住 sa 代表系统管理员。当然也可以以不同用户登录，只要该用户具有系统管理员权限。

(4) 输入安装 SQL Server 时设定的 sa 密码。对区分大小写的服务器，登录名也区分大小写，因此确认以小写字母输入。

如果连接到由其他人安装的服务器，或者更改了默认用户信息，则必须提供匹配更改的登录信息。单击“确定”按钮后，可以看到如图 2-8 所示的“对象资源管理器”窗口。

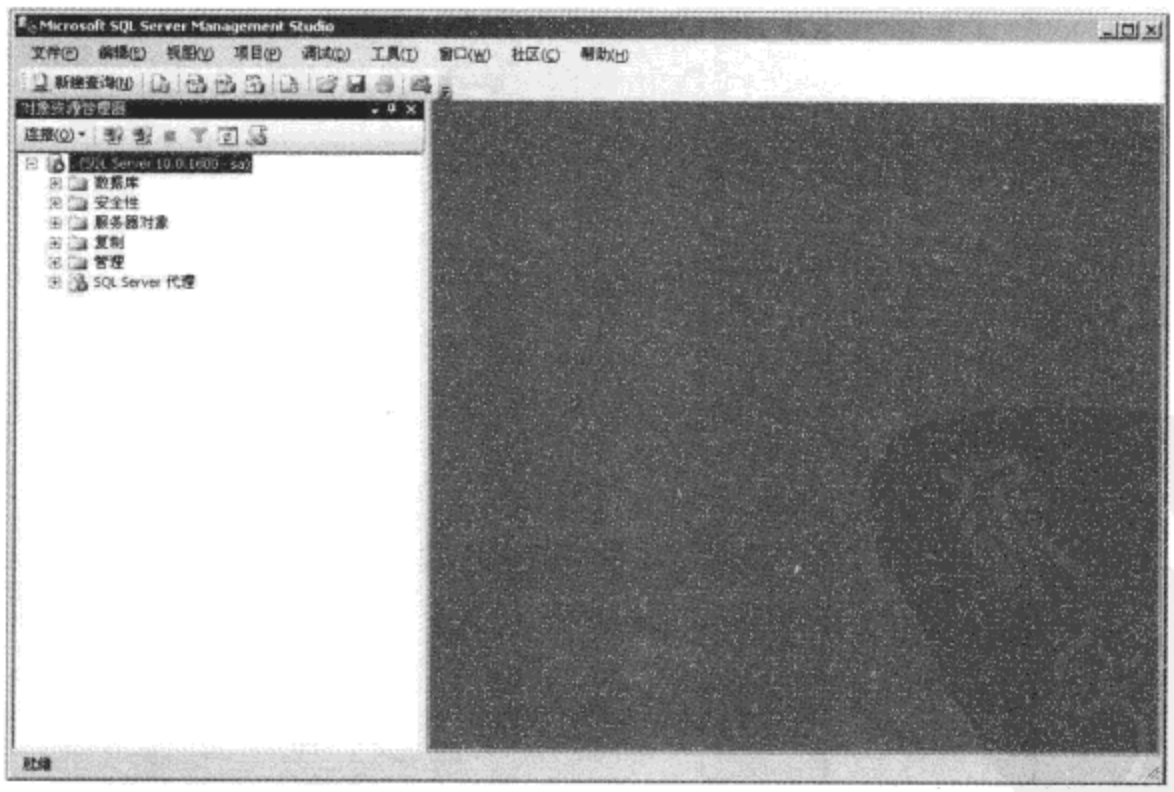


图　2-8

说明：

注意 sa 用户的密码。系统管理员的 sa 用户或其他用户为超级用户，可访问任意对象。

www.TopSage.com

示例说明

“登录”对话框收集建立连接所需要的所有信息。收集好信息后，再将这些连接信息组装成一个连接字符串，并将该连接字符串发送给服务器。该连接或者被接受或者被拒绝，如果被接受，则“查询”窗口得到连接句柄，连接可由多次查询重复使用，只要不断开连接。

同样，虽然其中的许多菜单(如“新建”、“打开”、“保存”、“剪切”、“粘贴”等)类似其他 Windows 应用程序，而您也应该很熟悉，但其中也有许多是 SQL Server 特有的。注意 Management Studio 窗口中的菜单是区分上下文的，即窗口中的菜单会根据活动窗口的内容不同而改变。当您浏览 Management Studio 不同的部分时，注意观察得到的不同的上下文菜单。

2.3.2 “查询”窗口

“查询”窗口取代了以前版本的一个工具——查询分析器(Query Analyzer)。“查询”窗口是给定 SQL Server 的交互式会话工具，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 的本地语言，也是结构化查询语言(SQL)的方言，在很大程度上遵循现代 ANSI/ISO SQL 标准。其实绝大部分 RDBMS 产品都支持基本的 ANSI/ISO SQL 兼容性。

由于在本书中要经常使用“查询”窗口，因此这里深入介绍该工具，并帮助您熟悉其使用方法。

1. 开始操作

前面花费了大量篇幅来介绍理论知识，现在开始一些操作。为此，单击 Management Studio 窗口左上部分的“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”窗口。打开“查询”窗口后，其菜单与早期作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，但首先来进行第一个查询操作。

首先在 SQL 编辑器工具栏的数据库下拉列表框中选择 AdventureWorks2008，然后在“查询”窗口的主窗口中输入以下代码：

```
SELECT * FROM Person.Address;
```

输入时注意一些变化情况。

- 词和短语的颜色在输入时改变。
- 在输入时，Management Studio 会猜测用户要输入的内容(如图 2-9 所示)。和 Visual Studio 类似，使用 IntelliSense，SQL Server 将提示接下来可能要输入的代码内容。

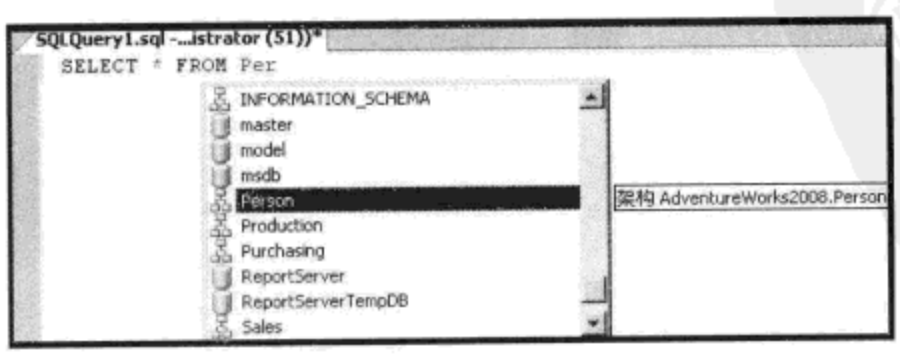


图 2-9

www.TopSage.com

语句关键字应为蓝色。无法确定的项，如列名和表名(每个服务器上的每个数据库的表名都各不相同)为黑色。语句参数和连接器为红色。注意颜色的变化并记住这些。它们可以帮助您在运行语句之前(与在看到出错结果之后)发现许多故障。

提示：

IntelliSense 是 SQL Server 2008 的新功能。尽管其很强大，但与其他语言相比，并不是没有 SQL 所带来的怪异问题。在添加要选择的列时，它特别有用。在后续章节将看到相关的语法，但 SQL 语法要求列名位于列所属的表名之前。这就会给 IntelliSense 带来问题，如在输入列名时，该工具不知道要从哪个表来获得这些列(因此不会给出适当提示)。如果结果令人失望，可以忽略它，直接添加表名，然后回来填充列名。

SQL 编辑器工具栏上的 √ 按钮(语法分析)为另一简单调试工具，该工具可在实际运行语句之前快速分析查询语句。如果有任何语法错误，则在运行之前可找到这些语法错误。还有另一种发现错误的调试器，该调试器在第 12 章中会深入介绍。

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”窗口变为如图 2-10 所示。

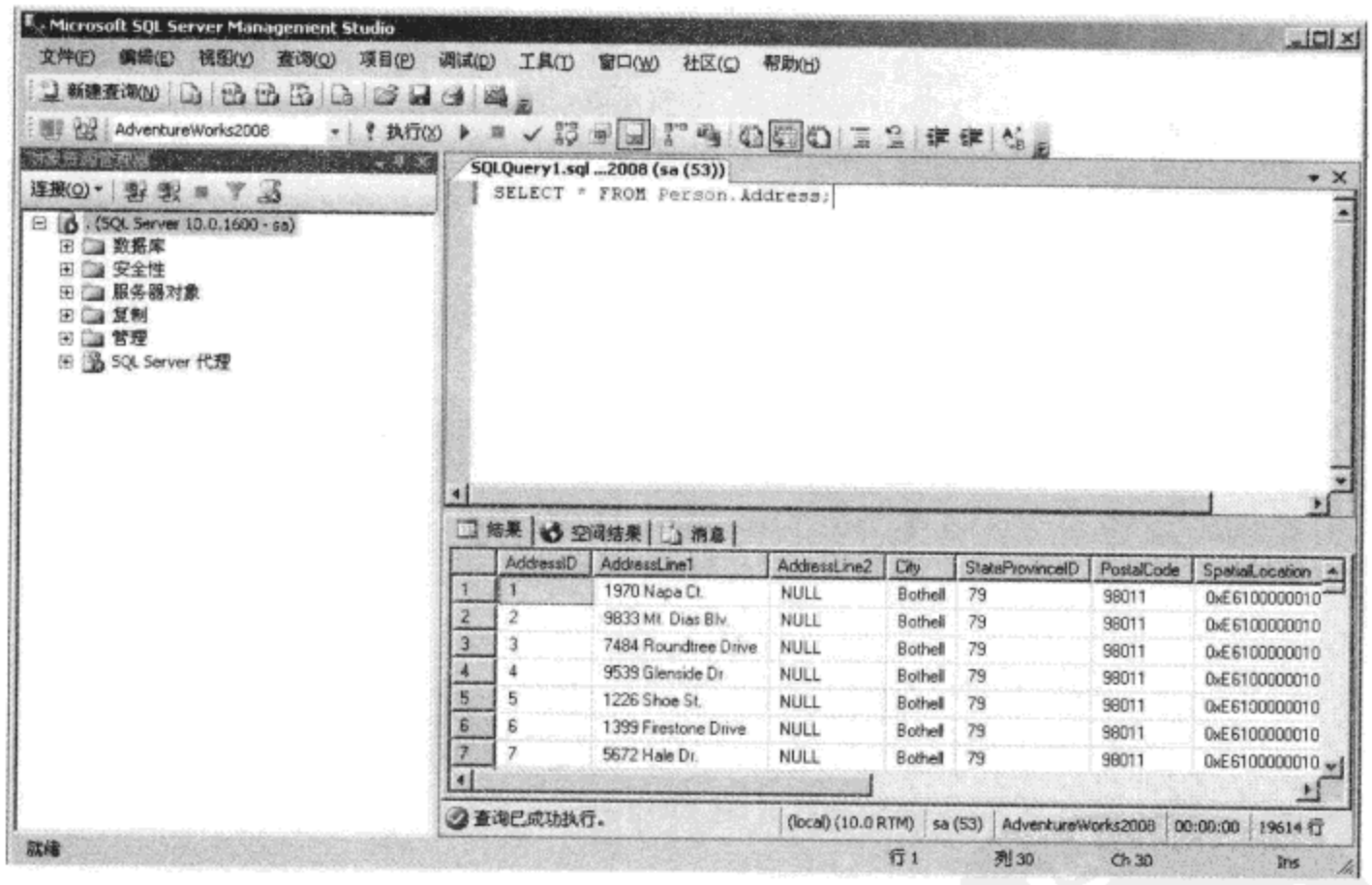


图　2-10

注意，主窗口已经自动划分为两个面板。上面的面板是初始查询文本；下面的面板称为结果面板。此外，注意结果面板的顶部有一个选项卡。在运行返回多个数据集的查询后，将会看到在每个单独的选项卡上获得每个结果；这可能相当便利，因为通常不知道每个数据集或结果集的长度。

注意：

术语结果集和记录集频繁用于表示一组数据，这些数据作为正在运行的一些命令的结果返回。可以将这些单词看作是可交换的。

www.TopSage.com

现在，改变一处或两处设置，查看具体的变化情况。查看“查询”窗口上方的工具栏，并且检验一组的 3 个图标，图 2-11 中突出显示了这 3 个图标。

图 2-11

这些图标可以控制输出结果的方法。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)。

进一步讨论之前，先使用该选项来再次运行上一查询，并查看返回的结果。选择“以文本格式显示结果”选项，并单击“执行”按钮再次运行到先前的查询，如图 2-12 所示。

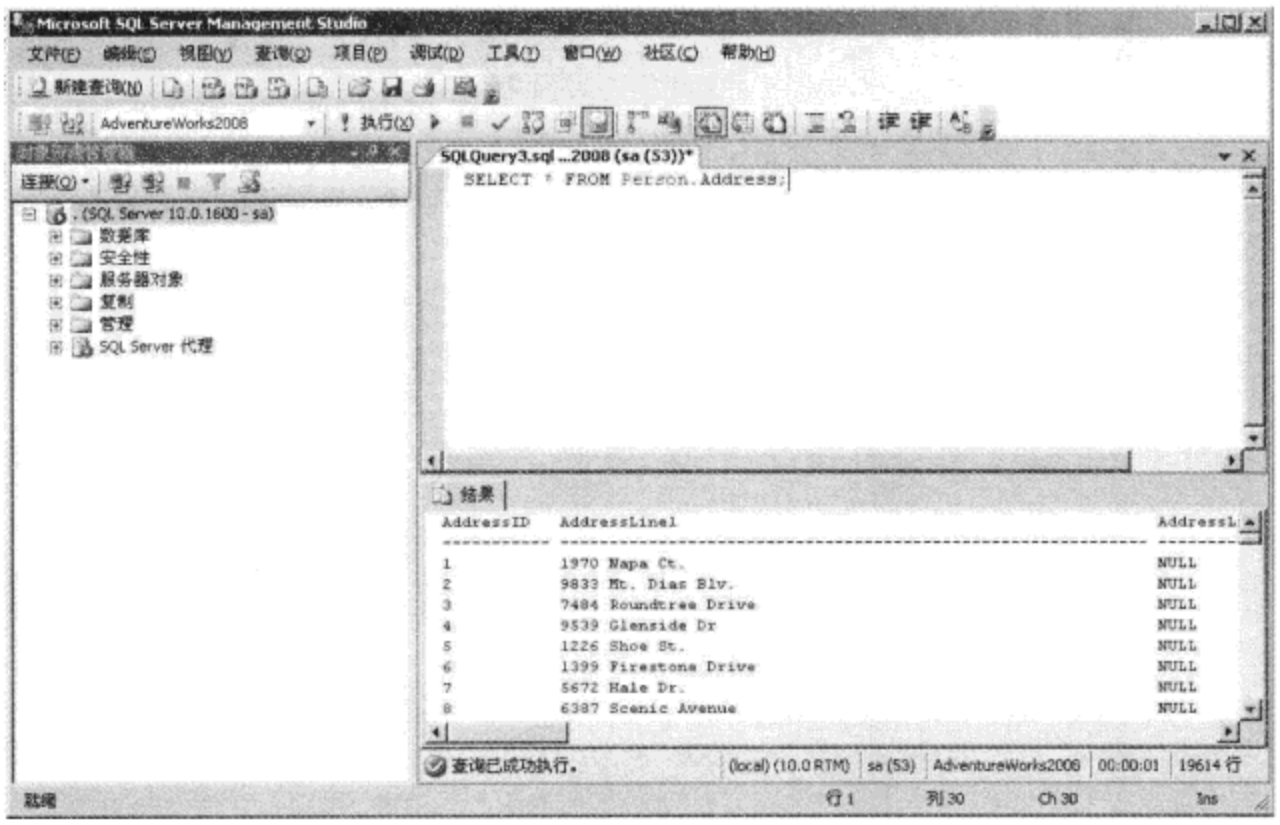


图 2-12

返回的数据与刚才得到的结果完全相同。只是与刚才的格式不同。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，且该结果只有很窄的几列
- 想要以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。

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。该选项使得返回结果具有(“以文本格式

www.TopSage.com

显示结果”选项不具有)以下几个特点。

- 可以更改列的大小，只要将鼠标指针悬停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自动调整大小。
- 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)，则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，剪切的数据会全部粘贴到一个单元格中)。
- 可以从多行只选择一列或两列(在“以文本格式显示结果”选项下，如果选择几行，则所有行包括的列都被选中；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

该选项与“以文本格式显示结果”类似，但它是直接将结果输出到文件而不是屏幕。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箱发送的文件。

5. sqlcmd 模式

稍后将更详细地讨论 sqlcmd。目前只要知道，它是帮助我们从 Windows 命令行运行查询的工具。它有一些主要针对命令行脚本的特殊功能。默认情况下，这些特殊的脚本命令在“查询”窗口中不可用。而启用 sqlcmd 模式就可在“查询”窗口中激活特殊的 sqlcmd 脚本选项。

注意：

要知道“查询”窗口总是使用 SQLNativeClient 连接方法(即使在 sqlcmd 模式下运行)，而实际的 sqlcmd 实用程序将使用 OLE DB 连接。结果将看到使用 sqlcmd 运行脚本的行为与从“查询”窗口使用 sqlcmd 有所不同。区别很小，几乎不可见，通常也是无影响的。

6. 显示执行计划

每次运行查询时，SQL Server 要将查询语句分解成各个部分，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，它能使得查询运行时返回结果的速度最快，而对其他用户影响最小。使用“显示估计的执行计划”选项，会返回关于 SQL Server 运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。绝大部分情况下，实际执行与估计的执行计划相同，但偶尔也有不同，这是因为当运行查询语句时优化器决定条件的更改，以及运行查询语句时的执行计划的实际开销发生变化。

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”项，再执行查询，如图 2-13 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，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，估计了所有的开销信息，但查询语句实际并未运行。在“包括实际的执行计划”选项下，物理上执行了查询语句，且得到的是实际开销信息，而不是预计的开销信息。

www.TopSage.com

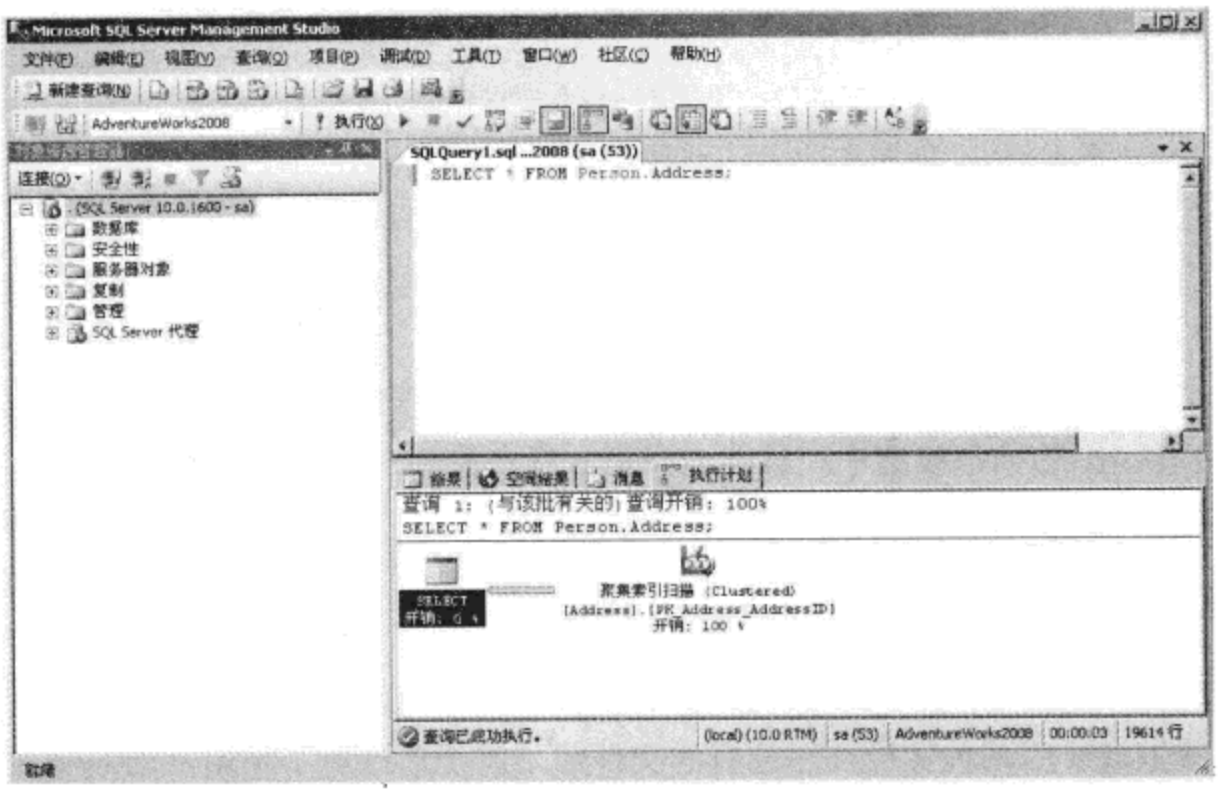


图 2-13

提示：

图 2-13 中的计划是非常简单的执行计划。更复杂的查询可能会显示多种分支和并行操作。

7. Available Databases 组合框

最后介绍一下 Available Databases 组合框。简单地说，您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(前面已将 AdventureWorks2008 改为默认数据库)。最初，“查询”窗口中是用户登录时默认的数据库(对 sa 用户，默认的数据库为 master 数据库，除非更改了系统)。然后可以变换到其他任何数据库，只要允许当前登录用户访问该数据库。因为这里使用 sa 用户名，所以在 Available Databases 组合框中应包括当前服务器上的所有数据库的选项。

8. 对象浏览器

这种实用的小工具可用来导航数据库，查找对象名，甚至执行脚本并查看底层数据。

在图 2-14 的示例中，可展开数据库节点，一直展开到 AdventureWorks2008 数据库中所有表的列表。可进一步打开每一列，包括列的数据类型和类似属性。对象浏览器对于数据库的浏览来说非常方便。

图 2-14

2.4 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。为了让您有直观的体会，在过去多年作者做了几个支持决策系统(Decision Support Systems，DSS)项目(这些系统通常没有在线数据输入/输出，而是将所有的数据集中到一起，以帮助管理制定决策)。

www.TopSage.com

DSS 项目从许多数据源中收集数据，并将数据集中到一个数据库，用于集中报表。

这些项目花费很高，而且要求迅速，因为项目试图解决如下实际问题：不是每个系统中相同名称的变量意味着相同的数据。要解决的问题可能有无限种。可能包括数据完整性规则(如何处理不允许 NULL 值的字段为 NULL 值)或者业务规则的不同(某个系统可以允许为负数的银行存款，而另一系统不允许，则要有一组表来处理银行存款)。这个问题列表还会不断扩展，因此花费很高。

使用 SSIS，可以避免(至少可以简化)用于处理这些问题的冗长的编码工作(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，并将数据输入到 SQL Server 数据表。

注意：

要知道有一种用于 ODBC 的特殊的 OLE DB 提供程序。该提供程序使您可以将 OLE DB 访问直接映射到 ODBC 驱动程序。即 ODBC 可以访问的任何数据也可以被 OLE DB 访问(因此 SSIS 也可访问)。

应当指出：SSIS 虽然只是 SQL Server 的一部分，但对任意 OLE DB 源和任意 OLE DB 目标起作用。即 SQL Server 不参与数据处理过程，而只提供数据泵(data pump)。例如，可以从 Oracle 中提取数据到 Excel 中，或者甚至从 DB/2 提取数据到 MySQL。

在传送数据时，可以对该数据应用相应的转换。数据转换本质上是指按照某种逻辑规则处理数据。数据的处理可简单，如更改列名一样；也可复杂，如数据完整性分析，必要时可应用完整性规则对数据更改。考虑前面介绍的示例，其功能是从一个可为 NULL 值的字段中取得数据并将数据传送到一个不允许 NULL 值的表中。使用 SSIS，可在数据传送过程中自动将 NULL 值改成选择的其他值(对于数字，可能为 0；对于字符，可能为未知的内容)。

2.5　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，其主要目的是将导入导出 SQL Server 的数据全部格式化。bcp 在 SSIS 很早之前就有，而 SSIS 在多数导入/导出功能上取代了 bcp，但 bcp 对于那些习惯命令行实用程序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 工具来快速转移数据。

2.6　SQL Server Profiler

很多次在我无计可施的时候，该工具向我指出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，但当您无计可施的时候，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

www.TopSage.com

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，我确信您能够想象在 1~2 分钟内打印这么多人执行的语句要花费大量的纸张。幸运的是，Profiler 工具有很强的过滤功能，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语法。如果存储过程有一些条件语句，会在不同环境下运行不同的语句，Profiler 就很有用。

2.7 sqlcmd

正如在讨论管理控制台时提到的，为了在命令行批处理文件中包括 SQL 命令和管理任务时，SQL Server 提供了一个工具——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实用程序的存在。因为 sqlcmd 是控制台应用程序，而不是 Windows 程序。

在 7.0 版本和 DTS(现在是 SSIS)出现之前，sqlcmd 通常与 Bulk Copy Program(bcp)一起使用来管理外部系统的数据导入。现在这种形式的使用在逐步减少，因为数据库管理员与开发人员都会学习功能强大而简单的 SSIS。但即使这样，在有些情况下，您仍然希望将一些任务脚本化为更大的命令行过程，而 sqlcmd 工具具有这种功能。

sqlcmd 使用起来很方便，尤其是要使用包含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 sqlcmd 工具的功能，而且具有和其他 SQL Server 工具一致的用户界面。在 *Professional SQL Server 2008 Programming* 一书中有 sqlcmd 的全面介绍。

提示：

为了能够理解 SQL Server 不同的“方言”，要知道 sqlcmd 是该工具的众多名称中的又一个新名称。它最早称为 ISQL，在 SQL Server 2000 和 7.0 版本中称为 osql。

2.8 PowerShell

PowerShell 是 SQL Server 2008 的新功能。它是一个非常健壮的脚本和服务器导航引擎。使用 PowerShell，用户可以导航服务器上所有对象，就好像它们是文件系统中目录结构的一部分一样(甚至可使用命令窗口中使用的 dir 和 cd 类型的命令)。

2.9 本章小结

本章提供的大部分工具并不是您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，知道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再次使用到这些工具。

注意在“开始”菜单里面没有某些其他工具的快捷方式(如连接工具、服务器诊断与维护工具)，这些工具通常与管理员相关。

www.TopSage.com

第 3 章

T-SQL 基本语句

到目前为止，我们已经介绍了 SQL Server 2008 中最繁琐的内容。相比于基本对象和工具，T-SQL 语言的学习要简单得多。但在建房子之前必须打好地基，而前两章内容的学习就是在打“地基”。在使用数据库前，必须清楚如何访问数据库的数据，然后才能学习如何以最佳方式来保存数据。

本章介绍最基本的 T-SQL 语句。T-SQL 是 SQL Server 的结构化查询语言(Structured Query Language，SQL)的“方言”。本章要学习的 T-SQL 语句如下所示：

- SELECT
- INSERT
- UPDATE
- DELETE

以上语句是 T-SQL 的基础内容。后面几章还会继续介绍许多其他的 T-SQL 语句，但这 4 个语句构成了 T-SQL 最基本的数据操作语言(Data Manipulation Language，DML)，因为在进行数据操作时，通常这 4 个语句(如读取和修改)比其他命令(如给用户授权或创建表)的使用频率要高得多。

此外，SQL 提供了许多的运算符和关键字，用于优化数据查询。本章将介绍一些最基本的运算符和关键字。

提示：

尽管 T-SQL 语言是 SQL Server 独有的，但通常使用的语句并不是 T-SQL 语言。T-SQL 语言在很大程度上还是遵循 ANSI/ISO 标准的(该标准最初由 ANSI 制定，之后被 ISO 接管。由于长时间都是由 ANSI 领导，因此人们还都称其为 ANSI 标准)，即 T-SQL 语言遵循广泛采用的开放式业界标准。对于开发人员来说，本书要学习的大部分 SQL 语言可直接转换至其他基于 SQL 的数据库服务器，如 Sybase(很早就与 SQL Server 共享相同的代码基)、Oracle、DB2 和 MySQL。但要知道，每个 RDBMS 会将标准范围扩大并改进，因而使用的标准往往超过 ANSI/ISO 标准。作者也将注意区分 ANSI 标准与非 ANSI 标准的适用范围。有时要在性能与其他 RDBMS 系统的可移植性之间作出权衡。但在大部分情况下，使用 ANSI 标准的方法与其他方法同样快速。

www.TopSage.com

因此通常应选择遵循 ANSI 标准的 SQL 语言。

3.1 基本 SELECT 语句

如果您在此之前未使用过 SQL 语言，或者还未真正理解，那么要注意本章内容的学习！SELECT 语句及其语法结构是 SQL Server 执行全部命令的基础。SELECT 语句的基本语法规则如下：

```
SELECT [ALL|DISTINCT] [TOP (<expression>) [PERCENT] [WITH TIES]] <column list>
[FROM <source table(s)/view(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 <column list>]
[[FOR XML {RAW|AUTO|EXPLICIT|PATH [(<element>)]}[, XMLDATA][, ELEMENTS]
[, BINARY base 64]]
[OPTION (<query hint>, [, ...n])]
```

上述语法结构很复杂，以下将依次介绍。

提示：

从技术上讲，TOP 表达式小括号内的内容是可选的。而 Microsoft 要求它们是“强制的”，并指出省略小括号实际也是支持的，但只是为了向后兼容。这意味着，Microsoft 在后面的版本中仍会有这一要求，因此如果不需要支持 SQL Server 以前版本，本人强烈建议使用小括号作为查询中的 TOP 表达式的定界符。

3.1.1 SELECT 语句与 FROM 子句

动词 SELECT 为整个语句的谓语部分，即告诉 SQL Server 执行什么操作。SELECT 表示仅仅读取数据信息，而不修改数据信息。所选择的内容由 SELECT 后面的表达式或列列表确定。稍后将介绍相关示例。

接着，要加入更具体的限制，如从哪里读取数据。FROM 语句指定要读取的数据所在的一个表或几个表的名称。了解了这些概念，就可以建立一个基本的 SELECT 语句。启动 SQL Server Management Studio，看一下下面这个简单的 SELECT 语句。

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

该语句的功能是显示所选择的信息。其中符号“*”看上去很奇怪，它是一个通配符，实际功能与其他地方使用的含义相同。使用“SELECT *”表示选择表中所有的列。然后，FROM 子句表示要输出信息的来源，这里是“INFORMATION_SCHEMA.TABLES”。

注意：

INFORMATION_SCHEMA 为特定的访问路径，用于显示系统的数据库及其内容的相关元数据。INFORMATION_SCHEMA 有几个可在句号后指定的部分，如 INFORMATION_

www.TopSage.com

SCHEMA.SCHEMATA 或 INFORMATION_SCHEMA.VIEWS。由于系统的元数据的特定访问路径都保存于此，因此不必使用系统表。

试一试　使用 SELECT 语句

将 AdventureWorks2008 数据库设置为当前数据库。回顾第 2 章所学的内容，只要在 Management Studio 的“查询”窗口顶部工具栏的下拉列表框中选择 AdventureWorks2008 选项，如图 3-1 所示。

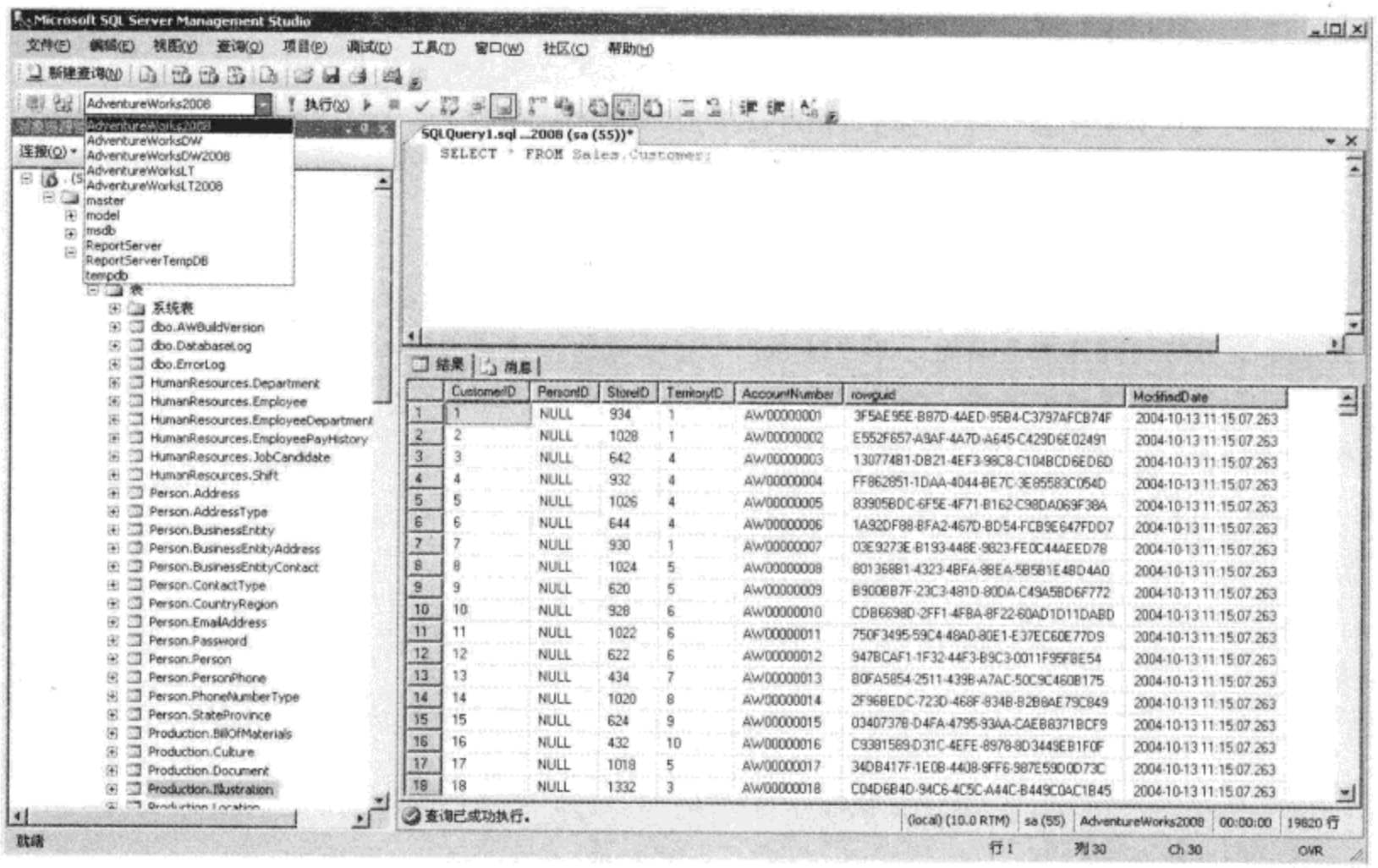


图　3-1

提示：

如果找不到列出各种数据库的下拉列表框，则单击一次“查询”窗口。SQL Server Management Studio 工具栏是上下文敏感的，即工具栏根据“查询”窗口当前的操作而改变。如果“查询”窗口当前不是活动窗口，可能有另一组工具栏显示(这组工具更适合于其他任务)。只要“查询”窗口为活动窗口，就会切换到适合查询需求的一组工具栏。

选择 AdventureWorks 数据库之后，先查看该数据库的一些数据。使用以下语句查询：

```
SELECT * FROM Sales.Customer;
```

在“查询”窗口输入以上语句后，单击工具栏上的“执行”按钮，查看 SQL Server 返回的结果。该查询列出当前数据库(AdventureWorks2008 数据库)中 Sales.Customer 表中所有列的每一行数据。如果在运行该查询语句之前没有更改系统设置或 AdventureWorks2008 数据库中的数据，则在单击“消息”选项卡后，将显示如下的消息。

www.TopSage.com

```
(19820 行受影响)
```

对于 SELECT 语句，显示的数字表示查询返回行的数目。在状态栏(结果面板之下)的右侧可看到相同的信息，同时还有其他一些有用的信息，如用户登录时使用的登录名、最后一次查询运行时使用的当前数据库(即使改变了数据库下拉框中的数据库，当前数据库仍将保持不变直到在这个查询窗口运行下一个查询)，以及该查询执行使用的时间。

示例说明

注意，SELECT 语句中的 SELECT 和 FROM 都是大写，而 SQL Server 并不要求大写，可以写成 SeLeCt 和 frOM，而该语句功能不变。通常习惯采用大写，以增加程序的可读性。许多 SQL 编码器对所有的命令和关键字采用约定的大写形式，而表名、列名和非常数变量名使用大小写混合形式。读者所选择或施加的标准可能不同，但至少要遵守保持一致的规则。

提示：

要读取代码或记住拼写不一致的表名是很痛苦的事情。当某人要查看您的代码，更重要的或者要使用您的列名或表名时，不应该占用他很长时间，让他凭经验来猜测您的代码。保持拼写一致是最简单不过的事情，但我曾使用的几乎所有数据库中都会做到这一点。切记拼写要保持一致！

SELECT 语句告诉“查询”窗口要完成什么功能，而符号“*”表示所要的信息(“*”表示所有的列)。然后是 FROM 语句。

FROM 子句定义数据的来源。FROM 之后接一个或多个表名。在上面的查询中，所有的数据来自 Customer 表。

下面的语句表示选择更具体的信息，想要按照姓显示所有客户列表。

```
SELECT LastName FROM Person.Person;
```

返回结果如下所示：

```
Achong
Abel
Abercrombie
...
He
Zheng
Hu
```

注意，为了简化起见，中间使用了省略号，实际会返回 19 972 行。因为想要显示每个客户的姓名，所以选择了所有的姓名。

提示：

许多 SQL 编码人员养成了简化查询语句的习惯，通常在选择语句中使用符号“”来选择所有的列。这一习惯要改掉！输入符号“*”比输入想要的列名是要省事，但检索的数据也比实际需要的多。此外，SQL Server 必须计算“*”表示多少列以及哪些列，这会降低应用程*

www.TopSage.com

序和网络的使用性能。简而言之，良好的规则就是选择您想要的列，不多不少。

试试另一个简单的查询。代码如下所示：

```
SELECT Name FROM Production.Product ;
```

同样，假设没有修改样本数据库中的数据，SQL Server 应返回 AdventureWorks 数据库中可用的 504 种不同产品的列表。

```
Name
----------------------------------------
Adjustable Race
Bearing Ball
BB Ball Bearing
…
…
Road-750 Black, 44
Road-750 Black, 48
Road-750 Black, 52
```

您选择的列正好是 SELECT 语句后的 SELECT 列表。简而言之，SELECT 列表由查询中要求输出的列名组成。

说明：

您选择的列正好是 SELECT 语句后的 SELECT 列表。

3.1.2　WHERE 子句

WHERE 子句用来设定返回记录的限制条件。上述的实例中返回的记录包括表中的每一行，即所要的信息没有任何限制条件。不加限制条件的查询在填充列表框和组合框时非常有用，并且在其他想要提供域列表(domain listing)的情况下也非常有用。

注意：

不要将域与 Windows 域混淆。域列表是要排除一些选择的列表。比如，如果想要某人提供给您美国一个州的信息，则您可能要提供给他们一个列表，限制其选择范围在 50 个州。这样才能保证选项的信息有用。在讨论数据库设计以及实体约束和域约束时，还会进一步涉及到域的概念。

下面考虑更具体的信息。假设只要列出指定产品名称的信息，而不是所有产品名的列表。比如使用查询语句返回 ProductID 为 356 的产品的名称、数量和订购点。

分解并构建该查询语句的过程如下：先考虑要返回的信息，因此使用 SELECT 语句。有关所要信息的查询语句是期望返回产品的名称、数量和订购点，因此要清楚这些信息对应的列名。另外还要清楚可以在哪个/哪些表中检索这些信息。

下面考虑要用到的表。之前已经使用过 Production.Product 表，因而清楚所要的信息就在那个表中。Production.Product 表包括几列。可从 Management Studio 的 Production.Product 表的“对

www.TopSage.com

象资源浏览器”树中快速查看所要列选项的列表。要在 Management Studio 中打开显示列表的窗口，可单击 AdventureWorks2008 数据库下面的“表”成员。然后将 Production.Product 表和 Columns 节点展开。如图 3-2 所示，显示了每一列以及其数据类型和是否可为 NULL 值。此外，本章后面的内容还会介绍查找这些信息的一些其他方法。

图 3-2

没有名为 product name 的列，但确实有可能是我们所要查询的 Name 列。其他两列很容易确认，列名只是将其中的空格省去。

这样所要的信息可以通过 FROM 从 Products 表查询，而 Name 列、ProductNumber 列和 ReorderPoint 列用于从指定列获取信息。

```
SELECT Name, ProductNumber, ReorderPoint
FROM Production.Product
```

但以上查询语句仍不能返回所要的结果；它仍返回了太多的信息。运行上面的查询语句，其返回了表中的每一个记录，而不是想要的那一行。

如果该表仅仅有少许的几个记录，则通过快速浏览返回的结果就能解决问题。毕竟浏览的只是一个短的列表。但是，如果表包括的记录非常多，则不可行。在任何有意义的系统中，几乎不存在只有少量几行记录的表。您不希望被迫去浏览 10 000 个记录。如果表有 100 000 或 1 000 000 个记录，那么要浏览所有的信息几乎是不可能，而且返回结果要花费很长时间。下面介绍如何在应用程序中设计这种查询以及快速而直接地找到所要的信息。

条件语句可以用来限制查询产品编号为 356 的产品，即 WHERE 子句。WHERE 子句在 FROM 子句之后，它定义要显示的记录必须满足的条件。本例查询的条件是 ProductID 等于 356，因此查询语句为：

```
SELECT Name, ProductNumber, ReorderPoint
FROM Production.Product
WHERE ProductID = 356
```

在 AdventureWorks2008 数据库运行该查询语句，则返回结果如下：

```
Name                          ProductNumber          ReorderPoint
----------------------------  -----------------      ------------
LL Grip Tape                   GT-0820                   600

(1 行受影响)
```

上面返回的结果正好是所要的信息，不多也不少。此外，该查询比第一个查询运行起来更快。

www.TopSage.com

表 3-1 列出了 WHERE 子句中可以使用的所有运算符。

表　3-1

运　算　符	示　　例	功　　能
=, >, <, >=, <=, <>, !=, !>, !<	<Column Name> = <Other Column Name> <Column Name> = 'Bob'	标准的比较运算符，与其他编程语言中的运算符功能基本相同，但要注意以下两点： (1) “大于”、“小于”和“等于”可能因情况不同而改变。如不区分大小写情况下，“ROMEY” = “romey”，但在区分大小写时，“ROMEY” <> “romey”。 (2) != 和 <> 都表示“不相等”。而 !< 和 !> 分别表示“不小于”和“不大于”
AND, OR, NOT	<Column1> = <Column2> AND <Column3> >= <Column 4> <Column1> != "MyLiteral" OR <Column2> = "MyOtherLiteral"	标准逻辑运算符。在 WHERE 子句中可以使用多个条件。它们在语句中运算的先后顺序为 NOT 、AND、OR。如果需要改变运算顺序可以使用小括号。注意这里不支持 XOR
BETWEEN	<Column1> BETWEEN 1 AND 5	第一个值在第二个值与第三个值之间时其值为 TRUE，其等价于 A≥B 且 A≤C。指定的值可以为列名、变量或字面量
LIKE	<Column1> LIKE "ROM%"	可使用%和_作为通配符。%表示可以代替任意长度的字符串。_表示可以代替任意的单个字符。 [ ]指定一个字符、字符串或范围，要求所匹配对象为它们中的任一个(如[a-c]表示 a、b 和 c。而 [ab]表示 a 或 b)。表示下一个字符是要被排除的，用于 NOT 运算
IN	<Column1> IN (List of Numbers) <Column1> IN ("A", "b", "345")	关键字 IN 左边的表达式与 IN 右边的任意值匹配时返回 TRUE。IN 常用于子查询。第 16 章将进一步介绍该运算符
ALL, ANY, SOME	<column|expression> 比较运算符 <ANY|SOME> (子查询)	子查询中的条件全部值/任意值满足比较运算符(如<、>、 =、>=)时返回 TRUE。ALL 指表达式要匹配结果集中的所有值。ANY 和 SOME 功能相似，在表达式匹配结果集中的任意值时返回 TRUE
EXISTS	EXISTS (子查询)	子查询返回至少一行记录时为 TRUE。第 16 章将进一步介绍该运算符

www.TopSage.com

提示：

这些不是 SQL Server 中唯一的运算符。它们只是适用于 WHERE 子句的运算符。有一些只用于赋值(而不是比较)的运算符，它们不适用于 WHERE 子句。

3.1.3 ORDER BY 子句

在前面运行的几个查询示例中，多数查询结果按照字母顺序输出。输出结果的顺序是否纯属偶然呢？答案会让您大吃一惊，确实纯属偶然。如果在查询中未指定查询结果的排列顺序，则由 SQL Server 决定数据的输出顺序。输出结果的顺序通常使得 SQL Server 以最低系统开销收集数据。通常的输出顺序按照表的物理存储顺序，或者按照 SQL Server 用于查找数据的索引的顺序。

提示：

Microsoft 的示例有一个令人讨厌的创建习惯，就是数据按照字母顺序出现(有关其原因说来话长，但作者希望不要这样)，但大部分数据不是那样。简单地说，其原因与 Microsoft 插入数据的顺序有关。由于通常是按字母排列顺序插入数据，ID 列也是按字母顺序(纯属偶然)。而大部分表在物理上也是按 ID 顺序排列(第 9 章将更详细地介绍数据的物理排序)，数据最终按字母顺序呈现。如果 Microsoft 以更随机的名称顺序插入——现实情况很可能是这样——如果未指定按名称排序，名称就会混乱排列。

ORDER BY 子句用来定义返回数据的排列顺序。在 ORDER BY 子句可以使用任意的组合列，只要按照 FROM 子句就可以在表中找到这些列或它们的派生列。

考虑以下查询语句：

```
SELECT Name, ProductNumber, ReorderPoint
FROM Production.Product;
```

运行后产生以下结果：

```
Name                     ProductNumber        ReorderPoint
---------------------  ----------------     ------------
Adjustable Race         AR-5381                   750
Bearing Ball            BA-8327                   750
...
...
Road-750 Black, 48      BK-R19B-48                75
Road-750 Black, 52      BK-R19B-52                75

(504 行受影响)
```

果然，查询结果集按照 ProductID 次序进行排序。这是由于 SQL Server 认为使用索引来查看数据最合理，而索引是按照 ProductID 来对数据记录进行排序。这样能使得查询的系统开销(就 CPU 和 I/O 而言)最低。当表中的记录变得比这大得多时，SQL Server 可能会选择完全不同的执行方案，而数据的排序也完全不同。可将查询语句作如下更改，但排序结果的顺序不变：

```
SELECT Name, ProductNumber, ReorderPoint
```


www.TopSage.com

```
FROM Production.Product
ORDER BY Name;
```

注意以上查询语句中没有使用 WHERE 子句，是否需要使用 WHERE 子句由查询功能确定。不过记住一点：如果确实用到了 WHERE 子句，则 WHERE 子句要放在 ORDER BY 子句之前。

遗憾的是，以上查询语句是否有 ORDER BY 子句对输出结果没有影响。但下面的查询语句将根据 ProductNumber 的顺序来更改输出结果的排列顺序。

```
SELECT Name, ProductNumber, ReorderPoint
FROM Production.Product
ORDER BY ProductNumber;
```

这次输出的结果有很大的不同。输出的数据基本重新排列：

```
Name                          ProductNumber  ReorderPoint
----------------------------  -------------  ------------
Adjustable Race               AR-5381        750
Bearing Ball                  BA-8327        750
LL Bottom Bracket             BB-7421        375
ML Bottom Bracket             BB-8107        375
...
...
Classic Vest, L               VE-C304-L      3
Classic Vest, M               VE-C304-M      3
Classic Vest, S               VE-C304-S      3
Water Bottle - 30 oz.         WB-H098        3

(504 行受影响)
```

SQL Server 还是选择系统开销最小的方法来输出查询的结果，但是由于查询语句的更改，实际需要执行的特定任务也随之变动。

也可以使用数据类型为 numeric 的字段来对输出结果进行排序(注意，这次是查询一个新的表)。

```
SELECT Name, SalesPersonID
FROM Sales.Store
WHERE Name BETWEEN 'g' AND 'j'
   AND SalesPersonID > 283
ORDER BY SalesPersonID, Name DESC;
```

查询结果如下：

```
Name                                SalesPersonID
----------------------------------  -------------
Inexpensive Parts Shop              286
Ideal Components                    286
Helpful Sales and Repair Service    286
Helmets and Cycles                  286
Global Sports Outlet                286
Gears and Parts Company             286
Irregulars Outlet                   288
```


www.TopSage.com

```
Hometown Riding Supplies          288
Good Bicycle Store                288
Global Bike Retailers             288
Instruments and Parts Company     289
Instant Cycle Store               290
Impervious Paint Company          290
Hiatus Bike Tours                 290
Getaway Inn                       290

(15 行受影响)
```

对于上述查询语句要注意几点。其中包括了几个知识点，混合使用了多个 WHERE 条件子句和一个 ORDER BY 子句；另外，在 ORDER BY 子句中作了一些调整。首先，ORDER BY 子句按多列排序。实现的方法是用逗号分隔要排序的列。这里是先按 SalesPersonID 排序，然后再按 Name 排序。其次，加入了新的关键字 DESC。DESC 关键字使 SQL Server 的输出结果按 Name 降序排列，而不是按照默认的升序排列(如果想要输出结果为升序排列，则可使用 ASC 关键字)。

注意：

通常基于返回的一列来对结果进行排序，但值得注意的是，ORDER BY 子句可基于查询使用的任何表的任何列来排序，而不管其是否在 SELECT 列表中。

3.1.4 使用 GROUP BY 子句聚合数据

把 ORDER BY 子句与本章前面的 SELECT 语句的输出结果相比，我们能够改变查询结果的顺序。

先回顾一下 SELECT 语句的整个语法结构：

```
SELECT [TOP (<expression>) [PERCENT] [WITH TIES]] <column list>
[FROM <source table(s)/view(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 <column list>]
[[FOR XML {RAW|AUTO|EXPLICIT|PATH [(<element>)]}[, XMLDATA][, ELEMENTS]
[, BINARY base 64]]
[OPTION (<query hint>, [, ...n])]
```

为何本书先介绍位于 GROUP BY 子句之后的 ORDER BY 子句？原因有以下两点。

- ORDER BY 子句比 GROUP BY 子句更常用。
- 要理解，只要按照 SQL Server 规定的顺序(按照语法结构定义)，就可灵活应用 FROM 子句之后的所有子句。

GROUP BY 子句用于聚合信息。下面介绍一个没有使用 GROUP BY 子句的简单查询。该示例想要知道给定订单中的订购数量。

```
SELECT SalesOrderID, OrderQty
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672);
```


www.TopSage.com

产生的结果集如下所示：

```
SalesOrderID  OrderQty
------------  --------
43660          1
43660          1
43670          1
43670          2
43670          2
43670          1
43672          6
43672          2
43672          1

(9 行受影响)
```

尽管只查询了 3 个订单，但可得到所有订单的每条订购信息。这时可使用计算器，也可使用 GROUP BY 子句和聚合函数。这里将使用 SUM()函数。

```
SELECT SalesOrderID, SUM(OrderQty)
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```

这次返回了预期结果：

```
SalesOrderID(无列名)
------------ -----------
43660        2
43670        6
43672        9

(3 行受影响)
```

按照期望，SUM 函数返回和，但是是返回什么数之和呢？可为结果提供一个别名。修改该查询，为输出提供一个列名：

```
SELECT SalesOrderID, SUM(OrderQty) AS TotalOrderQty
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```

得到的是相同的基本输出，但为分组的列提供了一个列名。

```
SalesOrderID TotalOrderQty
------------ -------------
43660         2
43670         6
43672         9

(3 行受影响)
```


www.TopSage.com

提示：

如果您只是想快速得到结果，就不需要像这里所做的那样使用别名，但您所编写的许多查询可能要为其他大型程序的元素提供信息。利用这些查询的代码将需要某种引用分组列的方法；在这种情况下为列取个有用的别名是很关键的。稍后将讨论别名。

如果不使用 GROUP BY 子句，那么 SUM 函数将返回指定列的所有行之和。而这里使用 GROUP BY 子句，因此 SUM 函数的结果是每一组之和。

注意：

在使用 GROUP BY 子句时，SELECT 列表中的所有列必须为聚合列(SUM、MIN/MAX、AVG 等)或是 GROUP BY 子句中包括的列。同样，如果在 SELECT 列表中使用聚合列，SELECT 列表必须只包括聚合列，否则必须有一个 GROUP BY 子句。

也可以基于多列进行分组，这时只要在多列列名之间加逗号。例如，查看每个销售人员与前 10 名客户签订的订单数量。这时可以在 GROUP BY 子句中使用 SalesPersonID 和 CustomerID 两列(稍后将介绍 COUNT()函数的使用)。

```
SELECT CustomerID, SalesPersonID, COUNT(*)
FROM Sales.SalesOrderHeader
WHERE CustomerID <= 11010
GROUP BY CustomerID, SalesPersonID
ORDER BY CustomerID, SalesPersonID;
```

COUNT 函数用于计数，这里计算了给定销售人员与给定客户签订的订单数量。

```
CustomerID SalesPersonID(无列名)
----------- ------------- -----------
11000              NULL      3
11001              NULL      3
11002              NULL      3
11003              NULL     3
11004              NULL      3
11005              NULL      3
11006              NULL      3
11007              NULL      3
11008              NULL      3
11009              NULL      3
11010              NULL      3

(11 行受影响)
```

1. 聚合函数

聚合函数常用于 GROUP BY 子句，用于聚合分组的数据。如在上述的查询示例中，返回了 OrderQty 列之和。对每一分组上选择的列进行计算并返回和，而分组由 GROUP BY 子句定义。在 SUM 示例中，只有 SalesOrderID 一个分组。聚合函数可在更大范围内使用，这里只给出了最常见的用法。

www.TopSage.com

注意：

聚合函数在和 GROUP BY 子句一起使用时显示出其强大功能，但聚合函数的使用不限于分组查询；如果查询语句中使用了聚合函数，而没使用 GROUP BY 子句，则聚合函数是用于聚合整个结果集(匹配 WHERE 子句的所有行)。当不使用 GROUP BY 子句时，在 SELECT 列表中某些聚合函数只能与其他的聚合函数一起使用，即聚合函数必须使用 GROUP BY 子句才能在 SELECT 列表中与列名配对。例如，不使用 GROUP BY 子句，SELECT 列表中 AVG 只能和 SUM 对应，但不能对应特定列。

2. AVG

AVG 函数用于计算平均值。试着运行之前返回订单数量的那个查询，不过这次改为返回每个订单的平均数，而不是每个订单的数量之和：

```
SELECT SalesOrderID, AVG(OrderQty)
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```

注意，返回结果完全改变：

```
SalesOrderID  (无列名)
------------ -------------
43660          1
43670          1
43672          3

(3 行受影响)
```

可对订单号为 43672 的 3 项记录之和求平均值进行验算：9/3=3。

3. MIN/MAX

顾名思义，MIN/MAX 用于计算所选择列分组的最小值与最大值。同样，对上一查询示例采用 MIN 函数进行修改：

```
SELECT SalesOrderID, MIN(OrderQty)
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```

返回的结果如下所示：

```
SalesOrderID (无列名)
------------ -----------
43660        1
43670        1
43672        1

(3 行受影响)
```


www.TopSage.com

再将查询示例改为采用 MAX 函数：

```
SELECT SalesOrderID, MAX(OrderQty)
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```

返回的结果如下所示：

```
SalesOrderID  (无列名)
------------ -----------
43660         1
43670         2
43672         6

(3 行受影响)
```

如果要在查询语句中同时使用 MIN 和 MAX 函数，也很简单，如下所示：

```
SELECT SalesOrderID, MIN(OrderQty), MAX(OrderQty)
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```

现在输出的结果多出了一列，产生了一点小问题：

```
SalesOrderID  (无列名)        (无列名)
------------ -----------  -----------
43660         1              1
43670         1              2
43672         1              6

(3 行受影响)
```

问题在哪儿呢？虽然返回了所有需要的列，但现在不仅仅包括 1 列聚合信息，如何确定哪一列数据对应哪一聚合数。当然，在这个示例中，可以确定两列分别是由 MAX 函数产生的最大值和由 MIN 函数产生的最小值，但结果对应于哪一列通常并不会这么简单。这时可以使用别名(alias)，它更改返回的结果集中的列名。可以使用 AS 关键字来创建它。

```
SELECT SalesOrderID, MIN(OrderQty) AS MinOrderQty, MAX(OrderQty) AS MaxOrderQty
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```

这样下面所示的返回结果的含义更明显：

```
SalesOrderID MinOrderQty MaxOrderQty
------------ ----------- -----------
43660         1           1
43670         1           2
43672         1           6
```


www.TopSage.com

```
(3 行受影响)
```

注意，AS 关键字是可选的。它曾经(SQL Server 6.5 版本之前)为非法关键字。删除两个关键字 AS 对查询的执行结果没有任何影响。同时注意还可以对任意列(甚至是表名，第 4 章会演示)取别名，而不仅仅是聚合函数。

再次运行最后一个查询语句，这次在某处不使用关键字 AS 为每一列取别名：

```
SELECT SalesOrderID AS "Order Number", MIN(OrderQty) MinOrderQty, MAX(OrderQty)
MaxOrderQty
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```

尽管有些别名没有使用 AS 关键字，但输出的每一列的列名都改为别名：

```
Order Number MinOrderQty MaxOrderQty
------------ ----------- -----------
43660        1           1
43670        1           2
43672        1           6

(3 行受影响)
```

提示：

必须承认，我通常在使用别名时也不使用 AS 关键字，而我也承认这是不好的习惯。在 AS 关键字可用之前，我已经开始使用 SQL Server，遗憾的是，旧习难改(因而很容易忘记用 AS 关键字)。但是我强烈建议使用 AS 关键字。这样一来使程序可读性更好，二是编写的代码符合 ANSI/ISO 标准。

对此进行说明的原因在于，作者希望读者能按标准使用 AS 关键字，但同时也想让读者了解一些其他使用方式，这样读者在看到一些稍有不同的用法时，就不会感到迷糊。

4. COUNT(表达式|*)

COUNT(*)函数用于计算查询中返回的行数。先看一个最常见的查询示例：

```
SELECT COUNT(*)
FROM HumanResources.Employee
WHERE HumanResources.Employee.BusinessEntityID = 5;
```

返回的结果与之前示例返回的结果有所不同：

```
(无列名)
-----------
1

(1 行受影响)
```

观察与以前返回查询结果的不同。第一，返回的所有列变为一个函数调用结果，没有默认

www.TopSage.com

列名。如果需要列名，则必须提供别名。第二，没有返回任何行记录。返回的结果表示查询中满足 WHERE 条件的行数，查询的表由 FROM 子句确定。

注意：

可以使用这个基本的查询语句来验证表中匹配 WHERE 条件的实际行数。

试一试不使用 WHERE 子句的查询：

```
SELECT COUNT(*)
FROM HumanResources.Employee;
```

如果没有对 Employee 表进行任何删除或插入操作，则应该返回如下结果：

```
(无列名)
-----------
290

(1 行受影响)
```

数字 290 表示 Employee 表中的总行数。记住这点，后面会用到。

可采用两种方法使用 COUNT 函数。前面已经讨论了使用“*”选项的 COUNT 函数，下面介绍使用表达式(通常为列名)的 COUNT 函数。

先用老方法运行 COUNT 函数，但采用的是一个新表：

```
SELECT COUNT(*)
FROM Person.Person;
```

该表数据量较大，因此返回的 COUNT 数也较大：

```
(无列名)
-----------
19972

(1 行受影响)
```

修改查询语句，选择指定列进行计数：

```
SELECT COUNT(AdditionalContactInfo)
FROM Person.Person;
```

返回的结果与上一个示例结果有所不同：

```
(无列名)
-----
10
警告：聚合或其他 SET 操作消除了 Null 值

(1 行受影响)
```

新的结果带来了一个问题：既然每一行都有 AdditionalContactInfo 列，那么与通常的行计数相比为什么两次查询得到的 AdditionalContactInfo 列的数目不相同？答案很明显，不是每一

www.TopSage.com

行的 AddistionalContactInfo 列都有值。简而言之，除了 COUNT(*)函数，以其他任意形式使用的 COUNT 函数都忽略 NULL 值。下面验证正是 NULL 值导致了差异：

```
SELECT COUNT(*)
FROM Person.Person
WHERE AdditionalContactInfo IS NULL;
```

产生的结果如下所示：

```
(无列名)
-----------
19962

(1 行受影响)
```

下面做个简单的算术：

```
10 + 19 962 = 19 972
```

AdditionalContactInfo 字段中定义了值的 10 条记录，加上 AdditionalContactInfo 字段中值为 NULL 的 19962 行，正好是总行数 19972。

注意：

除了 COUNT(*)函数外，所有的聚合函数都忽略 NULL 值。要考虑到这一点对聚合结果的重要影响。许多用户在求平均值时，把数值类型字段中的 NULL 值看作为 0，但实际上 NULL 值与 0 并不等同，并不能这样使用。如果对有 NULL 值的列执行 AVG 函数或其他聚合函数，NULL 值将不会计入聚合值中，除非使用如 COALESCE()或 ISNULL()等函数，将 NULL 值转换成非 NULL 值。在第 7 章将继续深入介绍这一点，但在 T-SQL 编程或者数据库设计时要注意该问题。

为什么说这在数据库设计中很重要？因为它会关系到是否允许在字段中使用 NULL 值，要考虑到可能对数据库执行的查询方法和如何进行聚合操作。

COUNT 函数可与 GROUP BY 子句联合使用。

注意：

对于下一示例而言，需要加载和执行和可下载源代码一起打包的 BuildAndPopulate Employee2.sql 文件(可从 wrox.com 或 professionalsql.com 网站获取)。

下列示例中对 Employee 表的所有引用应指向新的 Employee2 表，而不是 Employee 表。

例如，老板要求获取每位经理管理的职员人数。之前使用的语句可以是把表中所有的行累加(使用 COUNT(*))，也可以是把表中除了列值为 NULL 的所有的行求和(使用 COUNT(列名))。当加入 GROUP BY 子句时，聚合函数执行的结果不变，只是聚合结果是针对每一组，而不是整个表。这样，可以使用下列查询语句获得这一职员人数：

```
SELECT ManagerID, COUNT(*)
FROM HumanResources.Employee2
GROUP BY ManagerID;
```


www.TopSage.com

注意只使用了 ManagerID 来进行分组，COUNT()函数是聚合函数，因此不必包括在 GROUP BY 子句中。

```
ManagerID    (无列名)
----------- -----------
NULL            1
1               3
4               3
5               4

(4 行受影响)
```

上述结果表明 ManagerID 为 1 的经理手下有 3 名职员，ManagerID 为 4 的经理有 3 名职员，ManagerID 为 5 的经理有 4 名职员。此外返回结果表明有一个 Employee 记录的 ManagerID 字段为 NULL 值，即该职员不受任何人管理(估计是公司总裁)。

从技术上来说，可以使用不带任何聚合函数的 GROUP BY 子句，但这样的语句可能毫无意义。要将所有行分组，SQL Server 必须要对所有的行操作，但从功能上说，可以使用 DISTINCT 选项来达到完全相同的效果(这点稍后会介绍)，而且运行速度更快！

以上介绍了如何分组操作，下面介绍一个易于混淆的概念。当然，学完 3.1.5 节之后，该概念对您可能只是小菜一碟。

3.1.5 使用 HAVING 子句给分组设置条件

以上介绍的查询条件都是针对指定行进行的。如果一行中的给定列没有给定值，或者该列值不在给定值的范围，则整行都被忽略。在使用分组之前确实存在这种情况。

如果要将查询条件放到分组之后，则可以使用 HAVING 子句。

HAVING 子句仅用于带有 GROUP BY 子句的查询语句中。WHERE 子句应用于每一行(在变成一组的某一部分之前)，而 HAVING 子句应用于分组的聚合值。

下面将 3.1.4 节的 GROUP BY 查询稍做修改，查询要求返回分配给每位经理管理的 EmployeeID 的职员数目：

```
SELECT ManagerID AS Manager, COUNT(*) AS Reports
FROM HumanResources.Employee2
GROUP BY ManagerID;
```

第 4 章会介绍如何把 Manager 列中的姓名输出到 EmployeeID 列。该公司显然有 3 位经理。除了一个不属经理支配的人之外，所有的人都要向这 3 位经理汇报，这个人可能是公司的总裁(可以使用查询语句来验证，但这里暂且相信假设成立)。

由于查询中没有使用 WHERE 子句，因此 GROUP BY 子句对表中的每一行都进行操作，且每一行都包括在一个分组中。下面将 WHERE 子句添加到该查询语句，看看查询结果会发生什么变化。

```
SELECT ManagerID AS Manager, COUNT(*) AS Reports
FROM HumanResources.Employee2
WHERE EmployeeID != 5
GROUP BY ManagerID;
```


www.TopSage.com

但返回的查询结果与预期的结果有点不同：

```
Manager      Reports
----------- -----------
NULL         1
1             3
4             2
5             4

(4 行受影响)
```

没有从结果集中排除任何行，但 ManagerID 4 相应的结果少 1(这和 ManagerID 5 有关系)。WHERE 子句过滤掉 EmployeeID 为 5 的一行。果然，EmployeeID 5 由 ManagerID 4 管理，因此 ManagerID 4 的总人数少 1(EmployeeID 5 不再被计数)。ManagerID 5 不受影响，因为我们是用 EmployeeID 排除而不是 ManagerID。关键要知道是在应用 GROUP BY 子句之前排除了 EmployeeID 5。

换个角度想想，看看如何解答下面这个问题：哪些经理要管理 4 个以上的职员？可以根据不使用 WHERE 子句的 COUNT 查询结果来解决该问题，但如何编程实现？即如何使用查询语句来返回那些要管理 4 个以上的职员的经理？如果试图用 WHERE 子句来解决该问题，你会发现根据聚合函数返回的结果不能解决该问题——因为在聚合函数执行之前，WHERE 子句已经由系统执行完成。下面使用 HAVING 子句来查询：

```
SELECT ManagerID AS Manager, COUNT(*) AS Reports
FROM HumanResources.Employee2
WHERE EmployeeID != 5
GROUP BY ManagerID
HAVING COUNT(*) > 3;
```

试一试，下面的结果与想要的结果更加相近：

```
Manager       Reports
-----------  -----------
5              4

(1 行受影响)
```

只有 1 个经理要管理 4 个以上的职员。

3.1.6　使用 FOR XML 子句输出 XML

SQL Server 自身有大量支持 XML 的功能。它在满足 XML 数据存储和操作需求方面提供了相当健壮的支持，从有效索引 XML 数据到根据模式文档验证 XML。

其中最早就有的一项功能是可与 SELECT 语句一起使用的 FOR XML 子句。使用它，可使查询结果以 XML 格式输出，另外可用许多选项精确地指定 XML 输出样式。因为 XML 的讨论要涉及到 XML 本身，所以这里暂且不详细介绍 XML 子句，而在本书第 16 章再详细介绍。目前还是先学习基本知识。

www.TopSage.com

3.1.7 通过 OPTION 子句利用提示

OPTION 子句是一种用来最优化运行 SQL 查询的方法，它重写了 SQL Server 的某种思想。因为 SQL Server 通常能做到最优化查询，所以使用 OPTION 子句反而可能画蛇添足。不过，还是有必要对它有所了解。

本书后面的章节涉及锁定时，会大量介绍查询提示，这里您只要理解 OPTION 子句影响提示信息，掌握 OPTION 子句几乎不需要什么基础，所以这里不做介绍。

3.1.8 DISTINCT 和 ALL 谓词

DISTINCT 和 ALL 谓词也是行为语句中的重要概念，与重复数据的处理有关。

例如，想要显示在一次销售中订单数量大于 30 的所有产品的 ID 列表。可以很容易使用以下查询语句来从 SalesOrderDetail 表中获得信息：

```
SELECT ProductID
FROM Sales.SalesOrderDetail
WHERE OrderQty > 30;
```

结果就是从 SalesOrderDetails 表中返回的订单数量大于 30 的每一行的匹配 ProductID：

```
ProductID
-----------
709
863
863
863
863
863
715
863
...
...
869
869
867

(31 行受影响)
```

从技术上角度说，结果满足要求，而实际返回的结果不符合要求。返回结果中有很多重复行。尽管我们可以查看哪些产品是在一次销售中订单数量大于 30 的，但返回的行数和重复行的数目实在太多。同前面讨论的问题一样，我们对此有解决办法，即采用 SELECT 语句中的 DISTINCT 谓词。

对上述查询语句稍做修改后再运行一次：

```
SELECT DISTINCT ProductID
FROM Sales.SalesOrderDetail
WHERE OrderQty > 30;
```


www.TopSage.com

这次返回的结果才是真正的一次订单数量超过 30 的 ProductID：

```
ProductID
-----------
709
715
863
864
867
869

(6 行受影响)
```

此时返回结果的数量大大减小，而且返回的内容也更合理。该查询的另一优点是比第一个查询语句实际执行更快。本书后面讨论性能问题时会深入探讨其中的原因，这里只要知道，因为不需要返回所有的行，所以对于满足查询的要求来说，也减小了 SQL Server 的工作量。

到此还没有介绍完 DISTINCT 谓词。DISTINCT 谓词不仅能用于 SELECT 语句，还能用于聚合函数的表达式中。注意比较以下 3 个查询语句。

第一个查询语句是获取 AdventureWorks 数据库中 SalesOrderDetail 表的行数：

```
SELECT COUNT(*)
FROM Sales.SalesOrderDetail;
```

如果 SalesOrderDetail 表未做修改，那么此时返回的结果应为 121317 行。

下面运行相同的查询，但是是对指定列进行计数：

```
SELECT COUNT(SalesOrderID)
FROM Sales.SalesOrderDetail;
```

由于 SalesOrderID 列为该表的键的一部分，因此其值不能为 NULL(有关索引的内容本章作了更多介绍)。所以本查询返回的结果与使用 COUNT(*)查询返回的结果相同，都为 121317 行。

注意：

键(key)是数据库的一个术语，用于标识表中每一行。实际上有几种不同的键(本书第 6、8、9 章将详细介绍)，通常“key”是指表的主键。主键是能唯一标识行的一列或列组。当使用主键来标识行时，可以肯定返回的结果只有 1 行记录，因为在同一表中不存在两行有相同的主键。

下面的示例对查询语句做如下更改：

```
SELECT COUNT(DISTINCT SalesOrderID)
FROM Sales.SalesOrderDetail;
```

这次返回完全不同的结果：

```
(无列名)
-----------
31465

(1 行受影响)
```


www.TopSage.com

因为聚合之前将所有的重复行先排除，所以返回结果的行更少。

提示：

可以在任意聚合函数中使用 DISTINCT 谓词，即使聚合函数中使用 DISTINCT 谓词没有实际意义。例如，在 AVG 函数中使用 DISTINCT 没有任何意义。

这里顺便介绍 ALL 谓词。很少在查询语句中使用 ALL 谓词。毫无疑问，ALL 与 DISTINCT 谓词含义相反。DISTINCT 谓词用于过滤掉重复的行，而 ALL 指包括所有的行。ALL 是任意 SELECT 语句的默认值，但有 UNION 的 SELECT 语句除外。第 4 章会详细介绍 ALL 谓词对 UNION 用法的影响，这里只要记住 ALL 表示除了 DISTINCT 谓词之外的任意情况。

3.2 使用 INSERT 语句添加数据

现在，读者应该对基本 SELECT 语句有了很好的掌握。现在开始讨论 SQL Server 的另一主要问题。如果不能先把数据输入到数据库，则无法得到数据库中大量的数据。可以使用 INSERT 语句来解决这个问题。

INSERT 语句的整体语法结构如下所示：

```
INSERT [TOP ( <expression> ) [PERCENT] ] [INTO] <tabular object>
  [(<column list>)]
  [ OUTPUT <output clause> ]
{ VALUES (<data values>) [,(<data values>)] [, …n]
    | <table source>
    | EXEC <prodecure>
    | DEFAULT VALUES
```

这个结构看起来有些繁杂，因此需要稍作简化。INSERT 语法更基本的语法结构如下所示：

```
INSERT [INTO] <table>
   [(<column list>)]
VALUES (<data values>) [,(<data values>)] [, …n]
```

下面介绍各部分的功能。

在该结构中，INSERT 为该语句的实际操作，告诉 SQL Server 使用该语句执行什么功能，而 INSERT 后面部分仅用于说明插入数据的详细信息。

INTO 关键字无真正含义，唯一的目的是增强整个语句的可读性。INTO 关键字完全可选，这里强烈建议语句中加入关键字 INTO，因为可增强语句的可读性。在学完本节之后，试试在该语句中加入 INTO 或不加入 INTO 关键字。不加入 INTO 关键字的确省事，但读起来有点奇怪——用不用 INFO 关键字由您决定。

语法结构中接下来是要插入数据的表(从技术来讲，可能会是表、视图、或常见表的表达式，不过这里只讨论向表中插入数据的情况)。

到目前为止，语法结构的含义还是相当简单的。其接下来的部分更难：列列表(column list)。显式列列表为可选项(在此明确规定用于接收数据值的列)，但在不提供列列表时要特别谨慎。

www.TopSage.com

如果不显式提供列列表，则要假定 INSERT 语句中的每一个值与表中的列一一对应(第一个值对应第一列，第二个值对应第二列，依此类推)。而且必须根据每一列的顺序提供插入值，直到最后一列不允许 NULL 值也没有默认值(稍后读者会明白其含义)。例外的是 IDENTITY 列，在提供值时应忽略它(SQL Server 将自动填充)。总之，列列表是一列或多列的列表，而 INSERT 语句要插入的数据安排在语句的下一部分。

最后要提供要插入的值。有两种方式提供插入值，这里只介绍显式提供数据来插入一行。要提供插入值，要使用 VALUES 关键字，然后是要插入值的列表，小括号里的各插入值之间由逗号分开。插入值列表中的项数必须与列列表中的列数完全匹配。插入值的数据类型必须能匹配或隐式转换成与列对应的数据类型。

提示：

是否要对所有列专门声明一个值？我强烈建议每一次都要给每一列命名，即使只是使用 DEFAULT 关键字或显式给定 NULL 值。DEFAULT 告诉 SQL Server 为该列使用默认值(如果没有默认值，则会出错)。

这样做的好处是增强了代码的可读性，可以很清楚知道哪一列要插入哪一值。此外，显式给定每一列的值可以减少错误。

下面来做一些练习。

要练习 INSERT、UPDATE 和 DELETE 操作，首先要创建一些表(AdventureWorks 有些庞大)。要尝试下面几个示例，需要运行一些还未讨论的语句。读者暂且不用为其内容而担心，因为第 5 章将详细讨论它们。

可以自己输入代码并执行，或是使用与本书可下载文件一起打包的 Chap3CreateExampleTables.sql 文件。

注意：

下面这个代码块被称为脚本。这个脚本由一个批处理组成。第 11 章将详细介绍批处理。

```
/* This script creates a couple of tables for use with
** several examples in Chapter 3 of Beginning SQL Server
** 2008 Programming
*/

CREATE TABLE Stores
(
 StoreCode      char(4)       NOT NULL PRIMARY KEY,
 Name           varchar(40)   NOT NULL,
 Address        varchar(40)   NULL,
 City           varchar(20)   NOT NULL,
 State          char(2)       NOT NULL,
 Zip            char(5)       NOT NULL
);

CREATE TABLE Sales
```


www.TopSage.com

```
(
 OrderNumber    varchar(20)    NOT NULL PRIMARY KEY,
 StoreCode       char(4)       NOT  NULL
   FOREIGN KEY REFERENCES Stores(StoreCode),
 OrderDate       date          NOT NULL,
 Quantity        int           NOT NULL,
 Terms           varchar(12)   NOT NULL,
 TitleID         int           NOT NULL
);
```

下面的插入练习主要集中于 Stores 表，先回顾一下 Stores 表的属性。在 Management Studio 的“对象资源浏览器”窗口中，在运行上面的脚本时展开的当前数据库(可能是 AdventureWorks 给出的我们在运行的其他示例的脚本)的“表”节点，然后展开“列”节点，如图 3-3 所示。

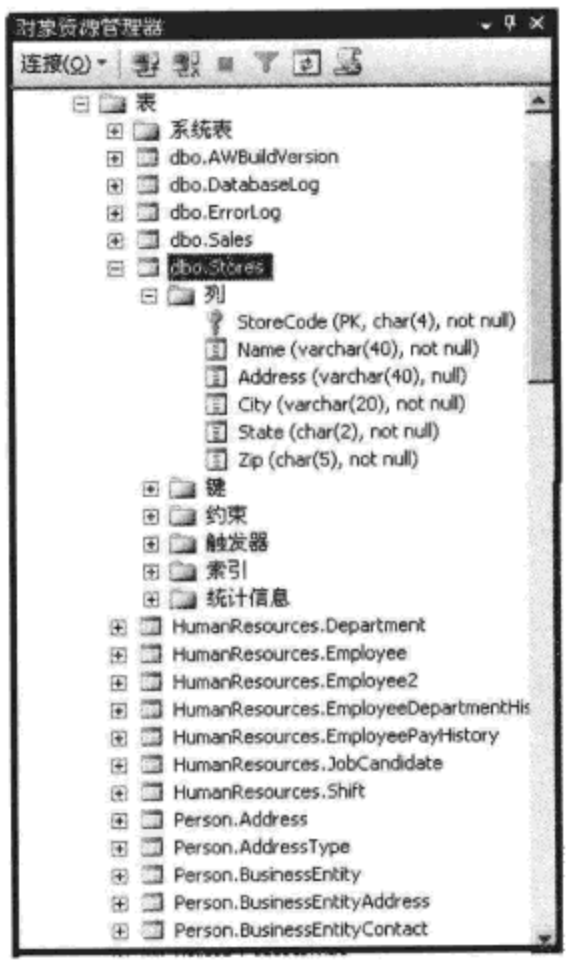


图 3-3

该表中所有列的数据类型为 char 或 varchar。

SQL Server 假设要插入的数据与表的每列一一对应，插入语句忽略列列表可选项。第一个示例如下所示：

```
INSERT INTO Stores
VALUES
   ('TEST', 'Test Store', '1234 Anywhere Street',
    'Here', 'NY', '00319');
```

如前所述，除非给定不同的列列表(稍后介绍如何给定列列表)，否则给定的所有要插入的值必须与表中定义的列一一对应。执行该查询语句后，返回提示信息：本查询插入了 1 行数据。然后再运行一次本查询，此时得到下列出错提示信息：

```
消息 2627，级别 14，状态 1，第 1 行
违反了 PRIMARY KEY 约束'PK__Stores__02A384F9175827B'不能在对象'dbo.Stores'中插入重复键
语句已终止。
```

为何该语句第一次可以执行，而第二次执行出错？其原因是本表的 StoreCode 字段为主键，该字段值不允许重复。只要更改该主键值，而剩下的其他列值不变，就可以将其插入到新的一行。在有关数据库设计与约束的章节中会更详细地介绍主键。

可使用以下语句检查刚插入的数据：

```
SELECT *
FROM Stores
WHERE StoreCode = 'TEST';
```

返回的结果正是插入的数据：

www.TopSage.com

```
StoreCode  Name        Address                City    State   Zip
---------  ----------  ---------------------- -----   -----   ----
TEST       Test Store  1234 Anywhere Street   Here    NY      00319

(1 行受影响)
```

注意，作者对每一列列名后面的空格作了调整，使得整个输出结果更加整齐，但实际插入的数据正是要插入的数据。

再对插入给定列语句稍作修改：

```
INSERT INTO Stores
   (StoreCode, Name, City, State, Zip)
VALUES
   ('TST2', 'Test Store', 'Here', 'NY', '00319');
```

注意，对以上语句插入数据值的那行做了两处改动。第一，更改了要插入的主键值，这样插入数据时不会出错。第二，省略了与 Address 列对应的值，这是因为在列列表中已经忽略了该列。有几种情况可以忽略列列表中的列，而且不用在 INSERT 语句中提供该列对应的插入数据。目前，仅利用了 Address 列不是必须给定插入值的列，因为如果没有提供该列的插入值，且该列没有默认值的话，则该列接受 NULL 值(稍后会详细介绍默认值)。当执行插入语句时，该列被设置为 NULL 值。下面对 SELECT 语句稍做修改，再次运行并验证插入值。

```
SELECT *
FROM Stores
WHERE StoreCode = 'TST2';
```

返回的结果有一点不同：

```
StoreCode  Name        Address  City     State    Zip
---------  ----------- -------  -------  -------- ------
TST2       Test Store  NULL     Here     NY       00319

(1 行受影响)
```

可注意到，为省略的列插入了 NULL 值。

注意，忽略的列必须允许设置为 NULL 值。顾名思义，它意味着该列允许有 NULL 值。本书会以大篇幅来介绍列是否可赋 NULL 值，现在只要记住，有些列可以设置 NULL 值，有些列不能设置为 NULL 值。在插入数据时，对于允许设置为 NULL 值的列，可以省略插入数据。

如果列不能设置NULL值，则必须为以下三种情况之一，否则系统会提示出错信息，INSERT 命令被拒绝执行。

- 列定义为默认值。默认值是在未提供插入值时自动插入的一个常数。第 7 章会介绍如何定义默认值。
- 列定义为接受某种形式的系统生成值。最常用的系统生成值为 IDENTITY 值(在数据库设计的章节将详细介绍)，即系统通常从第 1 个记录开始计数，在第 2 个记录时增加到 2，依次类推。这些值并不是真正的“行编号”，因为行以后可能会被删除，而这样的行编号在某种情况下会有跳跃，但是这些值还是可以作为每一行的标识。

www.TopSage.com

- 插入数据时已经提供了该列的值。

再举一个 INSERT 语句的示例。这次要把一个新的销售记录插入 Sales 表中。要查看 Sales 表的属性，可以像对 Stores 表那样打开其“属性”对话框，或者也可以运行系统存储过程 sp_help。存储过程 sp_help 的功能是给出任意数据库对象、用户自定义数据类型或 SQL Server 数据类型的信息。执行存储过程 sp_help 的语法结构如下：

```
EXEC sp_help <name>
```

要查看 Sales 表的属性，只要在“查询”窗口中输入以下命令：

```
EXEC sp_help sales
```

返回如下信息：

```
Column_name       Type       Computed  Length    Prec   Scale Nullable
---------------   --------   --------  --------  ----- ----- --------
OrderNumber       varchar    no         20                      no
StoreCode         char       no         4                       no
OrderDate         date       no         3          10     0     no
Quantity          int        no         4          10     0     no
Terms             varchar    no         12                      no
TitleID           int        no         4          10     0     no
```

Sales 表有 6 列，特别注意 Quantity、OrderDate 和 TitleID 列。它们是到现在为止的 INSERT 语句中还未使用过的数据类型。

本查询语句中需要注意的是插入数据的格式问题。插入数值不需要使用引号，而插入字符数据时要使用引号。但插入 date 数据类型时需要使用引号(实际上，其作为字符串输入，然后再转换成日期)。

```
INSERT INTO Sales
   (StoreCode, OrderNumber, OrderDate, Quantity, Terms, TitleID)
VALUES
   ('TEST', 'TESTORDER', '01/01/1999', 10, 'NET 30', 1234567);
```

该语句返回信息为：1 行受影响。可注意到，在插入操作中移动了列顺序。虽然插入语句的 VALUES 部分的数据需要与列列表匹配，但列列表可以为任意顺序；它不必按照物理表中列出的列顺序。

注意：

这里使用的 date 数据格式为 MM/DD/YYYY，该格式在美国很常用，可以使用其他广泛使用的格式(如国际上更为常用的格式 YYYY-MM-DD)。服务器上默认使用的格式会因您购买的是 SQL Server 本地化版本或者服务器上的设置被更改而异。

3.2.1 多行插入

一次插入多行是 SQL Server 2008 的新功能。要实现这一点，只需添加额外的用逗号分隔的插入值，如下所示：

www.TopSage.com

```
INSERT INTO Sales
   (StoreCode, OrderNumber, OrderDate, Quantity, Terms, TitleID)
VALUES
   ('TST2', 'TESTORDER2', '01/01/1999', 10, 'NET 30', 1234567),
   ('TST2', 'TESTORDER3', '02/01/1999', 10, 'NET 30', 1234567);
```

该插入语句使用一条语句插入了两组值。要检查结果，可查询 Sales 表。

```
SELECT *
FROM Sales;
```

显然，得到了之前插入的一行及刚插入的两行。

```
OrderNumber  StoreCode    OrderDate  Quantity   Terms      TitleID
------------ ---------    ---------- --------   ---------  -----------
TESTORDER     TEST        1999-01-01 10         NET 30     1234567
TESTORDER2    TST2        1999-01-01 10         NET 30     1234567
TESTORDER3    TST2        1999-02-01 10         NET 30     1234567

(3 行受影响)
```

提示：

执行多个插入操作可以潜在提升性能。以前，客户端应用程序必须为要插入的每条数据行单独发出插入命令(也有其他一些方法，不过需要开发人员作些思考和努力)。而使用这一方法可以大大减少往返于服务器的次数；记住，这也意味着应用程序将不向后兼容以前的 SQL Server 版本。

3.2.2　INSERT INTO…SELECT 语句

如何一次插入一块数据？前面讲到，可以显式执行多行插入，但如果要插入的数据块是从其他数据源选择获得的，又该怎么办？其他数据源包括：

- 数据库中的另一表
- 同一服务器上一个完全不同的数据库
- 来自另一 SQL Server 的异类查询或其他数据
- 同一表(通常在这种情况下，是在做数学计算或者是 SELECT 语句中其他的调整)。

INSERT INTO…SELECT 语句可完成一次插入一个数据块的功能。其语法结构为 INSERT 语句与 SELECT 语句语法结构的组合，如下所示：

```
INSERT INTO <table name>
[<column list>]
<SELECT statement>
```

由 SELECT 语句产生的结果集为 INSERT 语句中要插入的数据。

下面举例验证该语句的功能，当您成为编程高手时，就会发现需要经常选择一些数据保存到某些形式的临时表中。本示例要声明一个 table 类型的变量，用来保存 Orders 表中的各行数据。

www.TopSage.com

注意：

和之前创建数据库的示例一样，下列代码块也是所谓的脚本。本书第 11 章将详细介绍批处理。

```
/* This next statement is going to use code to change the "current" database
** to AdventureWorks2008. This makes certain, right in the code that we are
** going to the correct database.
*/

USE AdventureWorks2008;

/* This next statement declares our working table.
** This particular table is actually a variable we are declaring on the fly.
*/

DECLARE @MyTable Table
(
   SalesOrderID int,
   CustomerID char(5)
);

/* 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 AdventureWorks2008.Sales.SalesOrderHeader
   WHERE SalesOrderID BETWEEN 44000 AND 44010;

-- Finally, let's make sure that the data was inserted like we think
SELECT *
FROM @MyTable;
```

运行后输出的结果如下所示：

```
SalesOrderID CustomerID
------------ ----------
44000         27918
44001         28044
44002         14572
44003         19325
44004         28061
44005         26629
44006         16744
44007         25555
44008         27678
44009         27759
44010         13680

(11 行受影响)
```

第一部分信息(11 行受影响)是 INSERT…SELECT 语句运行的结果——SELECT 语句返回

www.TopSage.com

11 行记录，因此这 11 行记录被插入到临时表中。然后直接用 SELECT 语句来验证插入的数据。

注意：

如果您对@MyTable 表运行 SELECT 命令(也就是在脚本外)，则系统报错。@MyTable 是声明的一个变量，它只会在批处理运行时存在。运行后会自动销毁。

本示例中还要值得注意的是：使用了所谓的临时表，其与表变量实质上类似，但使用的方式完全不同。第 11 ~13 章还将介绍临时表和表变量。

3.3 用 UPDATE 语句更改数据

和其他 SQL 语句一样，UPDATE 语句表示更新，即 UPDATE 语句用来更新已有的数据。尽管有一定的相似性，但其语法结构与 SELECT 语句的语法结构有一点不同。和 INSERT 语句一样，它有着相当复杂的选项，但有一个能满足大多数需求的更基本的形式。

其完整语法也支持 SELECT 和 INSERT 中所支持的 TOP 及类似的谓词：

```
UPDATE [TOP ( <expression> ) [PERCENT] ] <tabular object>
 SET <column> = <value>[.WRITE(<expression>, <offset>, <length>)]
   [,<column> = <value>[.WRITE(<expression>, <offset>, <length>)]]
 [ OUTPUT <output clause> ]
[FROM <source table(s)>]
[WHERE <restrictive condition>]
```

UPDATA 语法更基本的结构如下所示：

```
UPDATE <table name>
SET <column> = <value> [,<column> = <value>]
[FROM <source table(s)>]
[WHERE <restrictive condition>]
```

可在多个表上创建 UPDATE 语句，但仅更新一个表的数据。即可以在多个不同表中构建条件或检索数据，但在同一时刻只有一个更新的主体。不要太担心这个——连接多个表还未介绍(第 4 章将介绍)，因此这里不会涉及到复杂的 UPDATE 语句。以下举一个简单的 UPDATA 语句示例。

要对 3.2 节中 INSERT 语句插入的数据进行更新。再次运行查询查看插入的 1 行数据(别忘了返回 pubs 数据库)：

```
SELECT *
FROM Stores
WHERE StoreCode = 'TEST';
```

返回结果如下所示：

```
StoreCode    Name          Address                City    State   Zip
----------  ---------- ---------------------     -----  -------  ----
TEST         Test Store 1234 Anywhere Street      Here    NY      00319
```


www.TopSage.com

更新 City 列的值：

```
UPDATE Stores
SET City = 'There'
WHERE StoreCode = 'TEST';
```

与 INSERT 语句的运行相似，SQL Server 没有返回更多的信息：

```
(1 行受影响)
```

但当再次运行 SELECT 语句时，发现 City 列的值确实已更新：

```
StoreCode  Name       Address                City     State    Zip
---------  ---------- -------------------- -------  -------  ----
TEST       Test Store 1234 Anywhere Street  There    NY       00319
```

注意，还可通过加逗号和列表达式来改变更多列的值。例如，下列语句将用来更新两列：

```
UPDATE Stores
SET city = 'There', state = 'NY'
WHERE StoreCode = 'TEST';
```

如果可以选择，则 SET 子句还可以使用表达式，而不使用之前示例中采用的确定值。例如，通过将现有商店名与一个后缀相串连，可为所有商店名添加后缀：

```
UPDATE Stores
SET Name = Name + ' - ' + StoreCode;
```

执行完上述更新语句后，对 Stores 表运行 SELECT 语句：

```
SELECT *
FROM Stores
```

这时将会发现 Name 列都添加了相应 Storecode 后缀：

```
StoreCode  Name               Address                City    State   Zip
---------  ---------------    --------------------   ------ ------ ----
TEST       Test Store - TEST  1234 Anywhere Street   There   NY     00319
TST2       Test Store - TST2  NULL                   Here    NY     00319
```

返回的结果表明，一条简单的 UPDATE 语句功能的相当强大，而这仅是个开始。后面的章节将对 UPDATE 作进一步的探讨。

注意：

SQL Server 功能非常强大，使得可以更新任一列数据(但有一些列不能更新，如 timestamp)，但在更新主键时应非常谨慎。更新主键会“孤立”其他数据(这些数据与更改的主键相关)。

例如，Stores 表中的 StoreCode 字段是主键。如要将 Stores 表中 StoreCode 字段的数据 10 改为 35，且如果与 Stores 表中 StoreCode 记录相关的所有 StoreCode 值没有由 10 更新为 35，则与 Stores 表相关的 Sales 表中的数据可能被“孤立”或“丢失”。当要执行该更新操作时，由于存在引用 Stores 表的约束，SQL Server 在这种情况下将禁止这种“孤立”操作的执行(第 7 章将

www.TopSage.com

详细介绍约束)，但最好不要更新主键。

3.4 DELETE 语句

DELETE 语句是本章介绍的所有语句中一个最为简单的语句。即使是最复杂的语句形式，该语句中也没有列列表，通常只有表名和 WHERE 子句。DELETE 语句的完整语法结构如下所示：

```
DELETE [TOP ( <expression> ) [PERCENT] ] [FROM] <tabular object>
 [ OUTPUT <output clause> ]
[FROM <table or join condition>]
[WHERE <search condition> | CURRENT OF [GLOBAL] <cursor name>]
```

其基本语法非常简单。

```
DELETE <table name>
[WHERE <condition>]
```

WHERE 子句和其他 WHERE 子句一样。由于要删除整行(而不是半行)，因此不需要提供列列表。

由于 WHERE 语句的使用较简单，因此本节只对 3.2 节的插入数据作快速删除。首先，运行 SELECT 语句，确保插入的第一行还在表中：

```
SELECT *
FROM Stores
WHERE StoreCode = 'TEST';
```

如果还未删除，那么应能找到之前 INSERT 语句插入的一行数据。可执行如下语句进行删除：

```
DELETE Stores
WHERE StoreCode = 'TEST';
```

可注意到，SQL Sever 拒绝删除该行，因为其违反了参照完整性：

```
消息 547，级别 16，状态 0，第 1 行
DELETE 语句与 REFERENCE 约束"FK__Sales__Stores__1D114BD1 冲突。
该冲突发生于数据库"AdventureWorks2008"，表"dbo.Sales"，column 'StoreCode'。
语句已终止。
```

SQL Server 不允许删除作为外键约束被引用的行。第 7 章会详细介绍外键，现在只要记住，如果一行使用外键引用另一行(无论是否在同一个表)，则要先删除被引用行后才能删除引用行。上一个 INSERT 语句在 Sales 表中插入了 StoreCode 为 TEST 的一行记录，这行记录正在引用想要删除的记录。

在删除 Stores 表中的该行记录之前，必须先删除在 Sales 表中引用的该行记录：

```
DELETE Sales
WHERE (StoreCode) 'TEST';
```


www.TopSage.com

再一次运行第一个 DELETE 语句可成功删除该行：

```
DELETE Stores
WHERE (Store Code) = 'TEST';
```

有两种方法来快速验证数据是否真正被删除。一种方法是在执行 DELETE 语句时，会自动弹出消息说明已经删除一行。另外一种方法是再次运行 SELECT 语句时，没有返回任何行。

再对 DELETE 语句稍作修改，就可以删除第二行记录：

```
DELETE Sales
WHERE StoreCode = 'TST2';
```

删除语句非常简单！像本章其他语句一样，在介绍更复杂的搜索条件时，本书会再次介绍 DELETE 语句的使用。

3.5 本章小结

T-SQL 是 SQL Server 的 SQL“方言”。它在很大程度上遵循 ANSI/ISO 标准，但同时又有很多其自身的扩展，后面几章会对此做更多的介绍。

为了实现向后兼容，SQL Server 语句有多种不同的语法结构，但功能相同，这时应尽可能使用 ANSI 标准格式。当有不同的语法结构时，本书通常都会一一介绍，但读者还是应尽可能使用 ANSI 标准格式。当后台或数据库服务器可能要更改时，这就显得特别重要。ANSI 代码在新的数据库服务器上非常有可能运行，而只是 T-SQL 的代码则肯定不能运行。

在本章中，读者已掌握了 T-SQL 中的操作单个表的语句，但实际上通常要从多个表来获得信息。第 4 章将学习如何使用 JOIN 语句来操作多个表。

3.6 练习

1. 编写输出 pubs 数据库的 Product 表(Production 模式中)中所有行与所有列的数据的查询语句。
2. 修改练习 1 的查询语句，仅搜索无 ProductSubCategoryID 的产品(提示：有 209 个，需要搜索 NULL 值)。
3. 在 AdventureWorks 数据库的 ProductLocation 表(Production 模式中)中添加一行。
4. 删除练习 3 中添加的行。

www.TopSage.com

第 4 章

连　　接

第 3 章介绍了最基本的 SQL 语句，但是它们仅仅是编程所要使用的 SQL 语句中的一小部分。简而言之，需要经常处理的并不仅仅是只有一个表——尤其是在高规范化的数据库中。

数据库规范化是指将数据从更大的表分割成几个较小的表，目的是消除重复的数据、节省空间、提高数据库性能，并增加数据的完整性。对于关系数据库，数据库规范化非常重要，当然这也意味着需要从各个地方获得数据。

提示：

本书第 8 章会详细介绍数据库规范化的概念。这里只要记住数据库越规范，就越可能要将多个表连接在一起，以便获得所需要的所有数据。

本章将介绍使用下列几种形式的 JOIN 子句，将多个表组合并形成一个结果集。

- INNER JOIN
- OUTER JOIN(LEFT 和 RIGHT)
- FULL JOIN
- CROSS JOIN

本章会介绍各种连接语句的语法结构，以及如何正确选择各语法结构。另外本章还要介绍 UNION 运算符，其作用是把两个查询的结果合并。

4.1　连接

在规范化环境中操作数据库时，经常会遇到这样的情况：不是所有想要的数据都在一个表中。在其他情况下，想要返回的全部数据都在一个表中，而想要设置条件的数据却在另一个表中。解决这些问题需要引入 JOIN 子句。

JOIN 含义为连接——把来自两个表的信息放置在一个结果集中。可以把一个结果集想象成为一个“虚拟”的表，它既有列又有行，并且列还有数据类型。本书第 7 章将介绍如何把结果集看成是一个表来处理，以及如何将结果集用于其他查询。

www.TopSage.com

JOIN 语句是如何正确地把来自两个表中的信息放置于一个结果集中的呢？其实，那取决于您是如何告诉它将数据放在一起的——这就是为什么有 4 种不同的 JOIN。所有 JOIN 的共同点是：将一个记录与另外一个或多个记录相匹配，从而生成一个新的记录，这个记录是由两个记录的合并列所产生的一个超集。

例如，从如表 4-1 所示的 Films 表中提取一个记录。

表 4-1

FilmID	FilmName	YearMade
1	My Fair Lady	1964

然后从表 4-2 所示的 Actors 表中取得一条记录。

表 4-2

FilmID	FirstName	LastName
1	Rex	Harrison

通过 JOIN，能够将完全不同的表中的两个记录合并成一个记录，如表 4-3 所示。

表 4-3

FilmID	FilmName	YearMade	FirstName	LastName
1	My Fair Lady	1964	Rex	Harrison

这个 JOIN(十分明显)按照一对一的关系连接记录。有一个 Films 记录与一个 Actors 记录相连接。

稍微扩展一下，看看结果有何不同。将另外一个记录加入到 Actors 表中，如表 4-4 所示。

表 4-4

FilmID	FirstName	LastName
1	Rex	Harrison
1	Audrey	Hepburn

现在把它与相同的 Films 表(只有一个记录)连接，看看连接后会产生什么结果，如表 4-5 所示。

表 4-5

FilmID	FilmName	YearMade	FirstName	LastName
1	My Fair Lady	1964	Rex	Harrison
1	My Fair Lady	1964	Audrey	Hepburn

正如所见，结果有所变化——不再是一对一，而是一对二，或者更恰当地称为一对多，可

www.TopSage.com

以多次使用 Films 表中的单个记录，以便于得到 Actors 表中的匹配记录的完整(连接的)信息。

您注意到它们是如何匹配的吗？没错！是通过两个表中的 FilmID 字段匹配来从两个记录生成一个记录。

由于只有有限的几个数据集，因此无论使用哪种类型的 JOIN，JOIN 都会产生相同的结果。下面详细介绍各种不同的 JOIN 类型。

4.2 内部连接

内部连接是连接类型中最普通的一种。与大多数连接一样，内部连接根据一个或几个相同的字段将记录匹配在一起，但是内部连接仅仅返回那些存在字段匹配的记录。在前一示例中，每一个记录都至少有一次被包含在结果集中，但是这种情况在现实情况中并不常见。

下面修改 Films 表，查看使用内部连接返回的结果。Films 表如表 4-6 所示。

表 4-6

FilmID	FilmName	YearMade
1	My Fair Lady	1964
2	Unforgiven	1992

Actors 表如表 4-7 所示。

表 4-7

FilmID	FirstName	LastName
1	Rex	Harrison
1	Audrey	Hepburn
2	Clint	Eastwood
5	Humphrey	Bogart

使用内部连接，返回的结果集如表 4-8 所示。

表 4-8

FilmID	FilmName	YearMade	FirstName	LastName
1	My Fair Lady	1964	Rex	Harrison
1	My Fair Lady	1964	Audrey	Hepburn
2	Unforgiven	1992	Clint	Eastwood

注意，Bogey 并不在返回的结果集中。那是因为在 Films 表中没有与之匹配的记录。如果在两个表中都没有匹配项，则不会返回该记录。现在有了足够的理论知识——通过编码实践一下。

对于内部连接，其常用语法结构如下：

www.TopSage.com

```
SELECT <select list>
FROM <first_table>
<join_type> <second_table>
       [ON <join_condition>]
```

这是一个ANSI语法，与使用SQL Server 6.5版本之前所需的专用语法相比，在非SQL Server数据库系统使用上述语法，您会感觉很幸运(6.5版本至今仍然被许多的开发人员所使用)。后面几章将介绍其他的语法。

提示：

值得注意的是“ANSI 语法”这个术语，因为其最初是在 20 世纪 80 年代中期作为 ANSI 标准的一部分而创建的。这一标准后被 ISO 接管，因此您可能会看到它基于这两个标准组织中的任意一个。

启动 Management Studio，在 AdventureWorks2008 数据库上测试内部连接的功能，使用的代码如下所示：

```
SELECT *
FROM Person.Person
INNER JOIN HumanResources.Employee
   ON Person.Person.BusinessEntityID =
     HumanResources.Employee.BusinessEntityID
```

因为查询返回的结果太多而无法在本书中显示出来，但如果运行该代码，那么将返回 290 行排序的结果。返回的结果中有些信息值得注意。

- 虽然 BusinessEntityID 列出现了两次，但是不清楚 BusinessEntityID 列来自哪一个表。
- 所有列都从两个表中返回。
- 列出的第一列来自列出的第一个表。

通过查看选择的第一个表，并把该表字段和返回的第一个 BusinessEntityID 列相匹配，可以确定 BusinessEntityID 列对应哪一个表，但是过程不但繁琐，而且容易出错。这也是为什么在连接中不建议使用“*”运算符的众多原因之一。然而在内部连接的示例中，这并不是很大的问题，因为我们知道两个 BusinessEntityID，虽然它们来自不同的表，但彼此完全相同。我们是怎样知道的呢？想想看——既然在两列中执行内部连接，它们就必须匹配，否则记录不能被返回！但不要养成这个习惯。在介绍其他的连接类型时，记录的匹配并不依据连接值相等。

正如希望的那样，返回的所有列都来自两个表。如以前学过的一样，使用“*”运算符将返回所有的列。先前提到过，在连接语句中使用“*”运算符是个不好的习惯。编写代码可能快速简单，但是易于出错，并降低系统的健壮性。

提示：

如第 3 章所提到的，早期采用的原则是：选择您需要的列，并返回您选择的列。在此的经验是：每一个返回的额外的记录或列都要占用额外的网络带宽，并且额外的查询要在 SQL Server 上运行。选择不必要的信息不仅仅降低当前用户的性能，还影响系统的每一个用户以及 SQL Server 所在的网络用户。

仅选择要使用的列，并使用 WHERE 子句来尽可能约束返回的列。

www.TopSage.com

如果要坚持使用“*”运算符，那么只能应用于需要从其中返回所有列的表，且“*”运算符只用于单个表。比如，假设需要所有联系人的基本信息，但只需要通过 Employee 表获取他们的 JobTitle，则应当将查询改成如下形式：

```
SELECT Person.BusinessEntity.*, JobTitle
FROM Person.BusinessEntity
INNER JOIN HumanResources.Employee
 ON Person.BusinessEntity.BusinessEntityID =
HumanResources.Employee.BusinessEntityID
```

本查询返回的结果并不包括大部分雇员的信息。实际上返回的结果只包括一个 BusinessEntityID 列的实例。返回的结果集包括 BusinessEntity 表中的所有列(由于使用了“*”运算符，该表的一个 BusinessEntityID 实例来自 SELECT 列表的一部分)以及唯一的 JobTitle 列(来自于 Employee 表)。现在再试一次，这次代码只有一处改动如下：

```
SELECT Person.BusinessEntity.*, BusinessEntityID
FROM Person.BusinessEntity
INNER JOIN HumanResources.Employee
   ON Person.BusinessEntity.BusinessEntityID =
HumanResources.Employee.BusinessEntityID
```

运行出了问题，返回以下错误信息：

```
消息 209，级别 16，状态 1，第 1 行
列名'BusinessEntityID'不明确。
```

为什么可以选择 JobTitle 列而不能选择 BusinessEntityID 列？SQL Server 返回信息已指出原因——选择列的名称存在多义性。JobTitle 仅存在于 Employee 表，而 BusinessEntityID 存在于两个表。SQL Server 无法知道想要哪一个。以上所有实例都可以确定要返回的 BusinessEntityID 字段：即 SQL Server 能够确定哪个表对应哪一列的字段。在第一个查询中(使用了“*”运算符)要求 SQL Server 返回所有字段——包含两个 BusinessEntityID 列，因此不需要解析名称。在第二个示例中(只对 BusinessEnitity 表使用了“*”)，不需要明确指定所选择的 BusinessEntityID 列——即返回了 Contact 表的所有字段，而 BusinessEntityID 列正好也包括在返回的列中。SQL Server 可以确定选择的 JobTitle 列，因为两个表中只有一个 JobTitle 列。

当返回列的名称在连接结果中多次出现时，必须完全限定列的名称。可以通过以下两种方法中的任一种来确定列名。

- 提供选择列所在表的表名，接着是一个点号(.)和列名(即表.列名)。
- 为表提供一个别名，接着是一个点号(.)和列名(即别名.列名)。

如果已经清楚知道“*”限定运算符的使用，则提供表名的方法非常简单。试一试使用限定列名的方法来返回 BusinessEntityID 列的查询。

```
SELECT Person. BusinessEntity.*, HumanResources.Employee. BusinessEntityID
FROM Person. BusinessEntity
INNER JOIN HumanResources.Employee
   ON Person. BusinessEntity. BusinessEntityID
      = HumanResources.Employee. BusinessEntityID
```


www.TopSage.com

这次语句运行成功，Employee 表中的 BusinessEntityID 列加到了结果集的最右边。

给表取别名的方法更麻烦一点，但这能够降低代码的冗余性并提高查询语句的可读性。和第 3 章在简单的 SELECT 语句中给列取别名相同，只要简单地将要用的别名放在表名后。注意，和列的别名类似，可以使用 AS 关键字(但是由于一些莫名奇妙的原因，实际中很少使用)。

```
SELECT pbe.*, hre.BusinessEntityID
FROM Person.BusinessEntity pbe
INNER JOIN HumanResources.Employee hre
   ON pbe.BusinessEntityID = hre.BusinessEntityID
```

运行以上代码，返回的结果与上一查询的返回的结果相同。

建议在语句中要么全部使用别名，要么从不使用别名。如果决定给表取别名，则必须在查询的每一个部分都使用别名。这是表与表之间关系的基础。试一试，运行以下混合使用别名的代码：

```
SELECT pbe.*, HumanResources.Employee.BusinessEntityID
FROM Person.BusinessEntity pbe
INNER JOIN HumanResources.Employee hre
   ON pbe.BusinessEntityID = hre.BusinessEntityID
```

代码看上去运行良好，但运行时返回如下错误信息：

```
消息 4104，级别 16，状态 1，第 1 行
无法绑定由多个部分组成的标识符 "HumanResources.Employee.BusinessEntityID" 。
```

可以选择哪些表使用别名，哪些表不使用别名，代码中别名和表可以混合使用，但是只要确定了使用表的别名，则必须一直使用别名。

回顾前几个示例返回的结果，注意到连接语句中的第一个表的列正是第一次返回的列。考虑一下其中的原因，以及如何控制返回的列。

SQL Server 通常按照连接语句中列的顺序返回，即按照您想要的列的返回顺序。由于第一个查询使用了“*”运算符，因此 SQL Server 按照物理表中实际列的顺序来返回结果。可以很容易将返回列的顺序重排，其中最简单的方法就是改变返回表的顺序，但通过在连接语句中明确地指定想要的列(甚至是所有的列)以及其顺序的方法，可以将返回列的顺序进行任意混合和匹配。

试一试　简单的连接语句

运行一个简单的查询语句来验证这一点：

```
SELECT pbe.BusinessEntityID, hre.JobTitle, pp.FirstName, pp.LastName
FROM Person.BusinessEntity pbe
INNER JOIN HumanResources.Employee hre
   ON pbe.BusinessEntityID = hre.BusinessEntityID
INNER JOIN Person.Person pp
   ON pbe.BusinessEntityID = pp.BusinessEntityID
WHERE hre.BusinessEntityID < 4
```


www.TopSage.com

返回的结果集非常简单：

```
BusinessEntityID    JobTitle                          FirstName    LastName
----------------    ----------------------------      --------------------
1                   Chief Executive Officer           Ken          Sánchez
2                   Vice President of Engineering     Terri        Duffy
3                   Engineering Manager               Roberto      Tamburello

(3 行受影响)
```

示例说明

和没有明确指定返回列(当仅仅使用*运算符)时不同，示例中指定了所要返回的列。因此SQL Server 清楚需要返回给的列——这些列都按照在 SELECT 列表中规定的顺序排列。事实上，即使添加一个附加表，也能够按期望的顺序混合排列所有表的列。

INNER JOIN(内部连接)——类似 WHERE 子句

以上所有的内部连接示例中，只介绍了任何一个连接类型都共有的概念——列的排序和取别名，它们对于任意一个连接都相同。内部连接与其他连接的不同之处在于，内部连接是一个排他的连接——即排除在两表中没有匹配的所有记录(第一个指定的表(即左表)和第二个指定的表(即右表))。

第一个示例中构造的 Films 和 Actors 表说明了内部连接这一特点。返回结果不包括 Bogey，是因为它在 Films 表中没有一个与之相匹配的电影。再看一两个真实的示例。

尽管您可能在前面的示例中还未意识到，但实际上我们已经利用了内部连接的这一排他特性。BusinessEntity 表使用的数据远远要超过 290 行。事实上，BusinessEntity 表包含了有关企业涉及的任何人的信息。他们可能是与特定客户或供应商相关联的个人，或者是雇员。可以用下列语句查看 BusinessEntity 表中有多少行：

```
SELECT COUNT(*)
FROM Person.BusinessEntity
```

运行它并假定没有对其作添加或删除操作，结果有将近 20 777 行；这个结果远远大于与 Employee 相关的查询所返回的 290 行。

另外 20 487 行去哪里了？正如所预期的，由于在 Employee 表中没有与之匹配的记录，它们被排除在结果集之外。正是由于这个原因，内部连接与 WHERE 子句相似。内部连接返回在其他的表中存在匹配的记录，类似的，WHERE 子句也只返回匹配规定标准的记录。

为了进一步证明和实践，这次查询要求返回至少与一个客户相关联的人的姓名列表和被关联的客户的账号。考虑表 4-9 所示的表。

表 4-9

Person.Person	Sales.Customer
BusinessEntityID	CustomerID
PersonType	PersonID
NameStyle	StoreID

www.TopSage.com

(续表)

Person.Person	Sales.Customer
Title	TerritoryID
FirstName	AccountNumber
MiddleName	rowguid
LastName	ModifiedDate
Suffix	
EmailPromotion	
AdditionalContactInfo	
Demographics	
rowguid	
ModifiedDate	

试试自己花几分钟时间完成这个查询语句，然后我们将把该问题分解。

首先要清楚需要返回的数据。问题要求返回两类不同的信息：联系人姓名和相关联的客户的账号。联系人的姓名(部分)在 Person.Person 表中。客户账号在 Sales.Customer 表中，因此可以写出 SELECT 语句的第一部分。为了简单起见，考虑连接联系人的姓和名：

```
SELECT LastName + ', ' + FirstName AS Name, AccountNumber
```

提示：

与其他语言相同，“+”运算符用于字符串的连接以及数字的加法。本示例用于将姓与名连接起来，姓名之间用逗号隔开。

接下来要把两个表连接起来，这时碰到的第一个问题：在使用连接时，表似乎没有连接的共同字段。

如果进一步看一下 Sales.Customer 表的定义，就可看到 Customer.PersonID 列有一个外键(表明指定列依赖于另一列的信息)。事实上，PersonID 与 Person.Person 表中的 BusinessEntityID 相关联。

事实上，我个人并不乐意使用 AdventrueWorks2008 数据库。我甚至为采用它而感到歉意，但出版商却强烈要求采用该数据库作为本书的样本数据库。这些列的命名很奇怪，而且在 AdventrueWorks2008 数据库中比比皆是，但还是要采用它们。

不过庆幸的是不要求连接的列用同样的名称，因此可通过为每个表指定适当的名称来连接列。

试一试　更复杂的连接

在 JOIN 语句中使用这个不匹配的名称对非常简单——只要采用 FROM 子句和 JOIN 关键字(不要忘了将当前数据库换成 pubs 数据库)：

www.TopSage.com

```
SELECT CAST(LastName + ', ' + FirstName AS varchar(35)) AS Name, AccountNumber
FROM Person.Person pp
JOIN Sales.Customer sc
 ON pp.BusinessEntityID = sc.PersonID
```

这样，SELECT 语句就完成了。如果执行它，则结果如下所示：

```
Name                      AccountNumber
------------------------  -------------
Robinett, David           AW00011377
Robinson, Rebecca         AW00011913
Robinson, Dorothy         AW00011952
Rockne, Carol Ann         AW00020164
Rodgers, Scott            AW00020211
Rodman, Jim               AW00020562
...
...
He, Crystal               AW00024634
Zheng, Crystal            AW00021127
Hu, Crystal               AW00027980

(19119 行受影响)
```

提示：

读者所返回结果的排列顺序以及结果顶部和底部的实际名称可能与这里的结果不同。SQL Server 不保证返回结果的顺序，除非使用 ORDER BY 子句，因为本语句没有使用 ORDER BY 子句，所以实际返回的结果有所不同。

示例说明

如果对 Person 表使用简单的 SELECT * 语句，则 Person 表中的几个联系人信息会被遗弃，因为他们很显然不是客户(可能是雇员、供应商联系人或潜在客户)，虽然已经将他们的信息输入到 Person 表中。内部连接(INNER JOIN)的关键就是排他性。

注意：

在上一查询语句中没有使用 INNER 关键字，这是因为内部连接(INNER JOIN)为默认连接类型。虽然有不同观点，但认为不使用 INNER 关键字的这种编码方法更为常见，因此要使用 INNER 关键字反而令人混淆，所以本书中不再使用 INNER 关键字。

4.3 外部连接

外部连接(OUTER JOIN)不像是规则，而更像是例外。这么说并不是因为它没有用途，而是因为以下几个原因。

- 经常使用的是内部连接，想利用其排他性。
- 许多 SQL 开发人员只是学习内部连接，而不会进一步学习外部连接，甚至不理解外部连接。

www.TopSage.com

- 通常有其他的方法完成外部连接的功能。
- 通常容易忘记该选项——外部连接。

内部连接本质上是排他的，而外部连接以及本章后面要讲到的完全连接是包含的。大家应该知道怎么使用外部连接，因为外部连接似乎将复杂的事情简单化。外部连接用于替代嵌套子查询时通常还能加快系统性能(本书第 7 章详细介绍)。

前面提到了连接有左侧和右侧的概念。第一个指定的表被看作是在左侧，而第二个指定的表被看作是在右侧。使用内部连接，因为两侧要求相等，所以很容易理解这个概念。但使用外部连接时，左侧与右侧的理解非常关键。表面上看非常简单，但通常由于不能区分左侧和右侧，而在使用外部连接时会出错。

为正确使用外部连接，这里介绍两种语法结构。第一种语法结构处理两个表的外部连接。第二种语法结构可处理多个表的外部连接及其他类型的连接。

4.3.1 简单的外部连接

第一种语法很简单，大多数人很容易理解。

```
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>
```

提示：

在示例中，本人习惯使用完整的语法——即使用 OUTER 关键字(例如 LEFT OUTER JOIN)。注意 OUTER 关键字为可选项，只需要使用 LEFT 或 RIGHT 关键字(如 LEFT JOIN)。而在实际中，OUTER 关键字很少使用。

这里可理解为：JOIN 关键字之前的表可看作为 LEFT 表，而 JOIN 关键字之后的表为 RIGHT 表。

外部连接本质上是包含的。明确包含的记录取决于使用的连接的哪一侧。LEFT OUTER JOIN 包含的信息来自左侧的表，而 RIGHT OUTER JOIN 包含的信息来自右侧的表。下面举一个简单的查询来说明。

假设要知道具体折扣信息、每一折扣的数量和哪些商品打折。查看 AdventureWorks2008 数据库，可发现 Sales 模式中有一个 SpecialOffer 表。还有一个名为 SpecialOfferProduct 的关联表，从中可知道哪些商品有哪些折扣信息。如表 4-10 所示。

表 4-10

Sales.SpecialOffer	Sales.SpecialOfferProduct
SpecialOfferID	SpecialOfferID
Description	ProductID
DiscountPct	rowguid
Type	ModifiedDate
Category	

www.TopSage.com

(续表)

Sales.SpecialOffer	Sales.SpecialOfferProduct
StartDate	
EndDate	
MinQty	
MaxQty	
rowguid	
ModifiedDate	

如果使用内部连接，则可以直接依据 SpecialOfferID 字段来连接两个表，查询语句如下所示：

```
SELECT sso.SpecialOfferID, Description, DiscountPct, ProductID
FROM Sales.SpecialOffer sso
JOIN Sales.SpecialOfferProduct ssop
ON sso.SpecialOfferID = ssop.SpecialOfferID
WHERE sso.SpecialOfferID != 1
```

注意，这里特意删除了无折扣的行(即 SpecialOfferID 为 1 的行)。该查询生成了 243 行——每行都有一个相关的 ProductID：

```
SpecialOfferID   Description                    DiscountPct   ProductID
--------------   ----------------------------   -----------   ----------
2                Volume Discount 11 to 14       0.02          707
2                Volume Discount 11 to 14       0.02          708
2                Volume Discount 11 to 14       0.02          709
2                Volume Discount 11 to 14       0.02          711
...
...

...
...
16               Mountain-500 Silver Clearance  0.40          986
16               Mountain-500 Silver Clearance  0.40          987
16               Mountain-500 Silver Clearance  0.40          988

(243 行受影响)
```

对此作一下思考。我们希望返回所有的折扣信息，而且折扣不必一定是实际使用的。以上查询返回实际使用了的折扣信息，这并不是想要的答案。

我们想要返回的是所有的折扣信息以及相应的 ProductID。

试一试 外部连接

为了返回所有的折扣信息和 ProductID，只需改变查询中的连接类型：

www.TopSage.com

```
SELECT sso.SpecialOfferID, Description, DiscountPct, ProductID
FROM Sales.SpecialOffer sso
LEFT OUTER JOIN Sales.SpecialOfferProduct ssop
 ON sso.SpecialOfferID = ssop.SpecialOfferID
WHERE sso.SpecialOfferID != 1
```

返回了类似的结果，但有一个重要的不同：

```
SpecialOfferID  Description                    DiscountPct    ProductID
--------------  ----------------------------   -------------  -----------
2               Volume Discount 11 to 14       0.02            707
2               Volume Discount 11 to 14       0.02            708
2               Volume Discount 11 to 14       0.02            709
2               Volume Discount 11 to 14       0.02            711
...
...
6               Volume Discount over 60        0.20            NULL
...
...
16              Mountain-500 Silver Clearance  0.40            986
16              Mountain-500 Silver Clearance  0.40            987
16              Mountain-500 Silver Clearance  0.40            988

(244 行受影响)
```

如果对 SpecialOffer 表执行 SELECT * 语句，那么可以很快发现返回了除 SpecialOfferID 为 1 的行(已经从结果中显式排除了)外的所有折扣信息。

示例说明

本例使用了 LEFT JOIN，而 SpecialOffer 表在 JOIN 关键字的左侧。如果使用连接，但在 SpecialOfferProduct 表中没有与之匹配的记录，则如何处理？因为不是在连接的包含侧(本示例为 LEFT 侧)，如果在右侧表中不存在与左侧表匹配的字段时，SQL Server 会为其他任意值填充 NULL 值。本示例中，除一行记录外的所有行都有 ProductID。可以看出，除 SpecialOfferID 6 外，所有的 SpecialOffr 都至少与一个产品关联。

这样就解决了上述问题：共有 16 种折扣类型，只有一种未使用(即 Volume Discount over 60)。

在本例中，改变成 RIGHT JOIN 会生成和内部连接相同的结果，因为 SpecialOfferProduct 表只包含了折扣和产品间有活动链接的行。不过，概念还是一样的。例如，可以改变引用表的顺序，然后使用 RIGHT JOIN。

试一试　RIGHT OUTER JOIN

将连接改变为 RIGHT OUTER JOIN 来查看运行结果：

```
SELECT sso.SpecialOfferID, Description, DiscountPct, ProductID
FROM Sales.SpecialOfferProduct ssop
```


www.TopSage.com

```
RIGHT OUTER JOIN Sales.SpecialOffer sso
 ON ssop.SpecialOfferID = sso.SpecialOfferID
WHERE sso.SpecialOfferID != 1
```

示例说明

如果尝试用 RIGHT JOIN 作同样的查询，那么将得到 243 行，与使用内部连接一样。运行上述代码，将得到未使用的折扣信息。

找出孤立或不匹配的记录

实际上可以使用 OUTER JOIN 的包含性来找到排除表的不匹配的记录。下面举例说明这点。

改变上面讨论的折扣问题，考虑要返回与任何产品不关联的折扣名称。想想如何使用查询语句来完成该问题。实际上，上一查询示例已经给出了 90%的解决方法。因为考虑到 OUTER JOIN 在没有匹配的折扣信息的 ProductID 列时返回 NULL 值，所要的结果与上一查询返回的结果集大致相同，所以只要过滤出任何有 ProductID 的记录，并且只返回折扣名称。因此只要更改 SELECT 列表并加入 WHERE 子句的限制条件：

```
SELECT Description
FROM Sales.SpecialOfferProduct ssop
RIGHT OUTER JOIN Sales.SpecialOffer sso
 ON ssop.SpecialOfferID = sso.SpecialOfferID
WHERE sso.SpecialOfferID != 1
AND ssop.SpecialOfferID IS NULL
```

返回了 ProductID 为 NULL 值的折扣产品的信息：

```
Description
------------------------------
Volume Discount over 60

(1 行受影响)
```

此刻您可能有一问题：折扣记录是否真的有 NULL 值？那是可以在连接中使用相同字段的 WHERE 子句的原因。如果基于两表的 SpecialOfferID 列进行连接，则只能存在以下三种情况。

- 如果 SpecialOfferProduct.SpecialOfferID 列有非 NULL 值，则根据 JOIN 子句的 ON 运算符；若折扣记录存在，则 SpecialOffer.SpecialOfferID 必须有与 SpecialOffer Product. SpecialOfferID 相同的值(ON ssop.SpecialOfferID = sso.SpecialOfferID)。
- 如果 SpecialOfferProduct.SpecialOfferID 列有非 NULL 值，则根据 JOIN 子句的 ON 运算符；若折扣记录不存在，则 SpecialOffer.SpecialOfferID 将返回为 NULL。
- 如果 SpecialOfferProduct.SpecialOfferID 正好有一个 NULL 值，且 SpecialOffer. SpecialOfferID 也有 NULL 值，则没有连接(两个 NULL 并不相等)，且 SpecialOffer. SpecialOfferID 将返回 NULL，因为没有与之匹配的记录。

NULL 值无法连接 NULL 值。为什么？考虑 NULL 值的比较——NULL 值不等于 NULL 值。在编码时要注意这一点。另一问题是为何可以在 NULL 之间使用等号，但实际上他们并不相等。

www.TopSage.com

要验证这一点，可以试试执行以下简单的代码：

```
IF (NULL=NULL)
   PRINT 'It Does'
ELSE
   PRINT 'It Doesn''t'
```

运行该语句，可以确定 SQL Server 是否认为 NULL 值与 NULL 值相等，返回信息是："It Doesn't"。

使用这一概念，能够确定不匹配的记录，用来找到先前内部连接示例中丢弃的记录。记得以下两个对 AdventureWorks2008 数据库运行的查询吗？

```
SELECT pbe.BusinessEntityID, hre.JobTitle, pp.FirstName, pp.LastName
FROM Person.BusinessEntity pbe
INNER JOIN HumanResources.Employee hre
   ON pbe.BusinessEntityID = hre.BusinessEntityID
INNER JOIN Person.Person pp
 ON pbe.BusinessEntityID = pp.BusinessEntityID
WHERE hre.BusinessEntityID < 4
```

与

```
SELECT COUNT(*)
FROM Person.BusinessEntity
```

第一个查询语句用来阐明内部连接功能。而第二个查询语句用来确定第一个查询语句排除的记录。现在使用外部连接来确定第一个查询语句中排除的记录。

从第二个查询语句SELECT COUNT(*)中可以确认第一个查询语句从BusinessEntity表中丢弃了一些记录(也可能从 Employee 表中丢弃一些记录，这里不讨论)。其含义是 BusinessEntity 表中的某些记录与 Employee 表中的记录不相匹配。当然，这是有可能的，因为有些人是客户或供应商联系人。尽管经理的第一个问题是所有雇员的联系信息，当然通常反过来也可能会问："哪些人不是雇员？"问题等价于"Person 表中的哪些记录与 Employee 表中的记录不相匹配？"解决办法与查询与任何商品不关联的折扣信息所使用的结构相同：

```
SELECT pp.BusinessEntityID, pp.FirstName, pp.LastName
FROM Person.Person pp
LEFT OUTER JOIN HumanResources.Employee hre
 ON pp.BusinessEntityID = hre.BusinessEntityID
WHERE hre.BusinessEntityID IS NULL
```

这样很快能得到只包含客户的联系人列表：

```
BusinessEntityID    FirstName                LastName
----------------    ---------------------    ----------------------------
293                 Catherine                Abel
295                 Kim                      Abercrombie
2170                Kim                      Abercrombie
...
```


www.TopSage.com

```
...
2088                    Judy                    Zugelder
12079                   Jake                    Zukowski
2089                    Michael                 Zwilling

(19682 行受影响)
```

注意：

LEFT 或 RIGHT JOIN 都可使用，只要正确的表或表组位于 JOIN 的相应侧。例如，可以在上一查询语句中使用 RIGTH JOIN，只要将 JOIN 两侧的 Person 与 Employee 表位置交换。例如，下列这样将产生相同的结果：

```
SELECT pp.BusinessEntityID, pp.FirstName, pp.LastName
FROM HumanResources.Employee hre
RIGHT OUTER JOIN Person.Person pp
 ON pp.BusinessEntityID = hre.BusinessEntityID
WHERE hre.BusinessEntityID IS NULL
```

在解决更复杂的查询时，存在更为常用的方法，用来找到一个表中的哪些记录与另一个表中的记录不相匹配。如果考虑使用性能，那么通常使用 JOIN 是最好的方法。当然也有例外，在后面就会碰到，但是通常使用 JOIN 是解决多表连接的最好方法。

4.3.2 处理更复杂的外部连接

下面介绍第二个示例，并讲述如何使用该示例。示例要处理外部连接与其他连接的混合使用(不论是哪种类型)。

当外部连接与其他连接混合使用时，对侧这个概念的理解非常关键。要理解：所有查询语句中 JOIN 左侧或之前的所有表只作为一个表，不论查询语句是要包含或排除表中的数据。同样，所有查询语句中 JOIN 右侧或之后的所有表只作为一个表。这里常见的错误是在查询语句中先执行了 LEFT OUTER JOIN，然后再使用 INNER JOIN。查询语句中的 OUT JOIN 包括了表中所有的记录，而 INNER JOIN 会丢弃表中的某些记录。大部分人起初对此很难理解，因此这里举例进行说明。因为 SQL Server 自带的数据库都不适合作为该点内容的阐述，这里要创建一个数据库和样本数据。

从下载的源代码中找到Chapter4DB.sql，运行该代码可创建名为Chapter4DB的示例数据库。在 Management Studio 查询窗口中打开该文件并执行它。

注意：

要利用下面几个示例，必须运行本书下载的源代码中包括的 Chapter4DB.sql 脚本。

下面将逐步构建查询语句并查看返回结果。查询语句要求返回供应商的名字和地址。因为示例数据库只有很少的几行记录，所以先查找所有供应商的名字。可以一直使用别名，查询语句如下所示：

```
USE Chapter4DB

SELECT v.VendorName
```


www.TopSage.com

```
FROM Vendors v
```

结果返回很少的 3 行记录：

```
VendorName
--------------------------------------
Don's Database Design Shop
Dave's Data
The SQL Sequel

(3 行受影响)
```

这次返回的结果是所有供应商的名字。现在要加入地址信息，但是这里有两个问题。第一，由于无论如何要返回所有的供应商，因此使用外部连接。第二，由于一个供应商可能有多个地址，反之亦然，一个地址也可能对应多个供应商，因此数据库的设计要使用关联表。即不能直接将 Vendors 表与 Address 表连接，而必须将这两个表与链接表(即 VendorAddress 表)连接。逻辑上先编写如下查询代码：

```
SELECT v.VendorName
FROM Vendors v
LEFT OUTER JOIN VendorAddress va
            ON v.VendorID = va.VendorID
```

因为 VendorAddress 表本身没有地址信息，所以在 SELECT 列表中没有从该表中返回的任何列。VendorAddress 表的唯一目的是一直用作为多对多关系的连接点(一个供应商可有多个地址，且一个地址可有多个供应商)。运行该语句，返回如下预期的结果：

```
VendorName
--------------------------------------
Don's Database Design Shop
Dave's Data
The SQL Sequel

(3 行受影响)
```

在这先停下来查看一下连接的表的数据，从 VendorAddress 表中选择所有的数据：

```
SELECT *
FROM VendorAddress
```

只返回了两行记录：

```
VendorID  AddressID
--------  -------------
1          1
2          3

(2 行受影响)
```

因此可知外部连接起作用了。因为在 VendorAddress 表中只有两行记录，而返回了 3 个供

www.TopSage.com

应商，所以必须从 Vendors 表中返回至少一行的记录，该行记录与 VendorAddress 表中的记录不相匹配。在这只要在供应商查询语句中简单添加一列就可以验证这一点：

```
SELECT v.VendorName, va.VendorID
FROM Vendors v
LEFT OUTER JOIN VendorAddress va
            ON v.VendorID = va.VendorID
```

VendorAddress 表中的 VendorID 列果真存在一 NULL 值：

```
VendorName                     VendorID
--------------------           --------------
Don's Database Design Shop     1
Dave's Data                    2
The SQL Sequel                 NULL

(3 行受影响)
```

若使用内部连接或右连接，则不会返回名为“The SQL Sequel”的供应商。而使用左连接确保查询结果返回所有的供应商。

现在验证了一部分信息，回到开始的查询语句，再添加第二个连接，用来获取实际的地址信息。由于不介意是否返回所有的地址信息，因此采用内部连接：

```
SELECT v.VendorName, a.Address
FROM Vendors v
LEFT OUTER JOIN VendorAddress va
            ON v.VendorID = va.VendorID
JOIN Address a
   ON va.AddressID = a.AddressID
```

结果返回了地址信息，但存在一个问题：

```
VendorName                     Address
---------------------------    ---------------
Don's Database Design Shop     1234 Anywhere
Dave's Data                     567 Main St.

(2 行受影响)
```

不知何故，返回的结果丢失了一个供应商信息。这里因为 SQL Server 按声明的顺序应用了规则。先对 Vendors 表和 VendorAddress 表使用外部连接。SQL Server 会按照查询要求返回所有的供应商。但接下来的一段指令出现了问题。返回的结果集包括所有的供应商，但是将结果集作为内部连接的一部分应用。因为内部连接对连接的两侧的表是排除的，因此只能返回结果集与 Address 表中匹配的记录。因为结果集只有两个记录与 Address 表中记录匹配，所以最终返回的结果集只有两个记录。可以用两种方法来解决该问题。

- 添加另一个外部连接
- 改变连接的顺序

www.TopSage.com

可以尝试下这两种方法。首先添加另一个外部连接：

```
SELECT v.VendorName, a.Address
FROM Vendors v
LEFT OUTER JOIN VendorAddress va
            ON v.VendorID = va.VendorID
LEFT OUTER JOIN Address a
            ON va.AddressID = a.AddressID
```

现在返回了所要的结果：

```
VendorName                    Address
----------------------------  ------------
Don's Database Design Shop    1234 Anywhere
Dave's Data                   567 Main St.
The SQL Sequel                NULL

(3 行受影响)
```

然后对开始的查询语句重新排序：

```
SELECT v.VendorName, a.Address
FROM VendorAddress va
JOIN Address a
   ON va.AddressID = a.AddressID
RIGHT OUTER JOIN Vendors v
            ON v.VendorID = va.VendorID
```

也返回了所要的结果：

```
VendorName                    Address
----------------------------  --------------
Don's Database Design Shop    1234 Anywhere
Dave's Data                   567 Main St.
The SQL Sequel                NULL

(3 行受影响)
```

您可能要问哪种方法更好？在 SQL 中，通常有几种方法来执行相同的查询，各种方法之间没有明显的优点。这两种方法也是一样。

本人建议您使用第二种解决方案。

注意：

通常，应首先使用内部连接，然后尽量少地使用外部连接，这样可减少代码中的错误。

原因是在查询时必须尽可能快地浏览表中的数据。如果要添加外部连接，不是因为要添加当前连接的表，而且因为试图要利用上一连接返回的结果集，那么这时返回的可能不是所要的结果，或者您可能会犯大的逻辑错误。第二种方法是在必要时使用一次外部连接来解决该问题。并非总能创造条件来灵活地连接，但通常是可以的。

www.TopSage.com

提示：

本人经常看到连接顺序的错误。这通常会让开发人员感到吃惊。本人多次被找去检查一些查询语句，而这些查询语句的各部分都由某些人花了几个小时验证过；至少一半情况下我都会被问到这是不是 SQL Server 的 bug。实际上这不是 SQL Server 中的 bug，而是开发人员代码中的 bug。当没有返回所要的结果时，作者建议先检查连接顺序的错误。

4.4 完全连接

像 SQL 中的其他用法一样，顾名思义，完全连接(FULL JOIN，或称为 FULL OUTER JOIN)用来将 JOIN 两侧的数据全部匹配，并返回所有的记录，不论记录在 JOIN 的哪一侧的表中。

在学习完全连接时，通常会感觉很酷，但几乎不会使用它。完全连接很少使用。完全连接的目的是返回数据之间的所有关系，不需给定连接的两侧表的优先顺序，返回所有记录，而不丢弃任何记录。

如果您会使用 LEFT JOIN 和 RIGHT JOIN，则 FULL JOIN 返回的结果最容易理解。根据完全连接的 JOIN 字段，可以返回所有匹配的记录。还可以返回所有左侧表中的任何记录，而在右侧表中没有匹配的记录时右侧列返回 NULL 值。最后(注意，这里所说的“最后”并不是指它们在查询中处于最后)还可以返回所有右侧表中的记录，而在左侧表中没有匹配的记录时左侧列返回 NULL 值。返回结果的顺序完全取决于 SQL Server 所认定的以最小的系统开销检索记录的方法(除非使用了 ORDER BY 子句)。

试一试 完全连接

使用之前在 4.3 节中演示的一个查询语句：

```
SELECT v.VendorName, a.Address
FROM VendorAddress va
JOIN Address a
   ON va.AddressID = a.AddressID
RIGHT OUTER JOIN Vendors v
              ON v.VendorID = va.VendorID
```

这里再次摘取一部分代码并在 SELECT 列表中添加字段来查看运行结果。首先对两个表使用完全连接：

```
SELECT a.Address, va.AddressID
FROM VendorAddress va
FULL JOIN Address a
      ON va.AddressID = a.AddressID
```

这里使用完全连接与右外部连接的结果相同：

```
Address          AddressID
--------------   -------------
1234 Anywhere    1
```


www.TopSage.com

```
567 Main St.      3
999 1st St.       NULL
1212 Smith Ave    NULL
364 Westin NULL

(5 行受影响)
```

再加入另一个连接：

```
SELECT a.Address, va.AddressID, v.VendorID, v.VendorName
FROM VendorAddress va
FULL JOIN Address a
      ON va.AddressID = a.AddressID
FULL JOIN Vendors v
      ON va.VendorID = v.VendorID
```

这样返回了所有的信息：

```
Address          AddressID    VendorID     VendorName
--------------   ----------   --------     ------------------------------
1234 Anywhere    1            1            Don's Database Design Shop
567 Main St.     3            2            Dave's Data
999 1st St.      NULL         NULL         NULL
1212 Smith Ave   NULL         NULL         NULL
364 Westin       NULL         NULL         NULL
NULL             NULL         3            The SQL Sequel

(6 行受影响)
```

示例说明

如返回结果所示，前两行和使用内部连接语句返回的结果相同。接下来是 3 个 Address 记录，它们与另一表的记录也不相匹配。最后一个记录来自 Vendors 表，同样没有与之匹配的记录。

再强调一次，在需要返回 JOIN 两侧表的所有记录时使用完全连接，不管记录是否匹配，但即使不匹配的记录也包括在返回结果中。

4.5 交叉连接

交叉连接的使用确实很怪异。交叉连接与其他连接不同，它不使用 ON 运算符，而将 JOIN 左侧的所有记录与另一侧的所有记录连接。简言之，返回的是 JOIN 两侧表记录的笛卡儿积。交叉连接与任何其他连接语法类似，不过它使用 CROSS 关键字(而不是 INNER、OUTER 或 FULL 关键字)，且不使用 ON 运算符。交叉连接的示例如下：

```
SELECT v.VendorName, a.Address
FROM Vendors v
CROSS JOIN Address a
```

回想一下，在 Vendors 表中有 3 个记录，而在 Address 表中有 5 个记录。如果将 Vendors 表中的所有记录与 Address 表中的所有记录匹配，则使用交叉连接可以返回 15 个记录：

www.TopSage.com

```
VendorName                    Address
----------------------------  --------------------
Don's Database Design Shop    1234 Anywhere
Don's Database Design Shop     567 Main St.
Don's Database Design Shop     999 1st St.
Don's Database Design Shop    1212 Smith Ave
Don's Database Design Shop     364 Westin
Dave's Data                   1234 Anywhere
Dave's Data                    567 Main St.
Dave's Data                    999 1st St.
Dave's Data                   1212 Smith Ave
Dave's Data                    364 Westin
The SQL Sequel                1234 Anywhere
The SQL Sequel                 567 Main St.
The SQL Sequel                 999 1st St.
The SQL Sequel                1212 Smith Ave
The SQL Sequel                 364 Westin

(15 行受影响)
```

这正是所要的结果。

每次教 SQL 课程时，关于交叉连接，总有人提问："为什么要使用交叉连接？" 本人所知的是交叉连接的科学用途，知道高等数学里面有使用许多笛卡儿积的函数。假设您读过大量的表结构示例，然后执行交叉连接，那么可以建立示例数据的笛卡儿积。但是通常使用交叉连接来建立测试数据。

当建立数据库时，数据库通常是大规模系统的一小部分，而系统需要大量的测试。测试大规模系统的一个常见问题是要创建大量的测试数据。使用交叉连接，只要将少量的数据项输出到两个或多个表中，然后对这些表执行交叉连接就可产生大量的测试数据。在上一查询示例中，如果将一组地址与一组供应商匹配，则简单的查询语句从 8 个记录生成 15 个记录。当然使用交叉可生成非常大的数据量。例如，如果建立了一个有 50 个名的表，再建立有 250 个姓的表，则使用交叉连接可建立 12 500 个不同姓名组合的表。这意味着通过输入 300 个姓和名，就可以立即得到一组 12 500 个名字的测试数据。

4.6 JOIN 语句的早期语法结构

本节要介绍的是目前大部分人仍视为"常用"的编写连接代码的方法。在 SQL Server 6.5 版本之前，本节将介绍的这些语法结构正是 SQL Server 使用的唯一语法结构，那时还不存在我们现在所说的"标准"的编写连接代码的方法。

现在的 SQL 语句使用 ANSI/ISO 语法结构。本人强烈建议使用 ANSI/ISO 标准，因为使用该标准编写的代码大大增强了系统之间的可移植性，以及代码的可读性。值得注意的是实际上早期的语法结构应当在当前的跨平台得到很好的支持。

提示：

这里介绍早期语法结构的原因是，毫无疑问迟早您会碰到早期的代码，而不想使您在碰到

www.TopSage.com

这类代码时说："这是什么鬼玩意？"

这里反复强调推荐您尽可能使用 ANSI 语法结构。此外，使用 ANSI 语法结构可增加代码的可读性，而且 Microsoft 不可能无限期地支持早期的语法结构。很难相信将大量的早期代码放到现在的代码中，Microsoft 迟早会清除这些早期的语法。

推荐使用 ANSI 语法结构的最大原因是 ANSI 语法功能更强大。在早期的语法下，可能会产生多义性的查询逻辑，即可能有多个方法来解释查询语句。新的语法排除了这一个问题。

本章前面介绍了 JOIN 子句与 WHERE 子句的区别，这是因为早期连接语句的语法结构要采用 WHERE 子句来表示。

除了完全连接外，早期的语法支持所有使用 ANSI 的连接。如果要执行完全连接，则只能采用 ANSI 版本。

4.6.1 内部连接的早期语法结构

回到本章第一个内部连接的示例：

```
USE AdventureWorks2008

SELECT *
FROM Person.Person
INNER JOIN HumanResources.Employee
   ON Person.Person.BusinessEntityID =
     HumanResources.Employee.BusinessEntityID
```

这次返回 290 行记录。除了使用连接，还可使用 WHERE 子句重写该连接语句。实际上很简单，只要去掉上一语句中的 INNER JOIN，并添加一个逗号，使用 WHERE 子句来代替 ON 运算符。

```
SELECT *
FROM Person.Person, HumanResources.Employee
WHERE Person.Person.BusinessEntityID =
   HumanResources.Employee.BusinessEntityID
```

非常简单！而且返回了与采用其他语法相同的 290 行记录。

提示：

实际上，当今世界主流的 SQL 系统(Oracle、DB2、MySQL 等)都支持该语法结构，但在查询处理中应用限制条件时会造成一些多义性。虽说遇到这种多义性的可能性很小，但还是存在的。因此建议对新查询使用 JOIN 语法并尽可能将旧的查询也修改为新的语法。

4.6.2 外部连接的早期语法结构

注意：

只有 SQL Server 在 SQL Server 2000 兼容模式下(通过 ALTER DATABASE 命令将兼容级别设置为 80)运行，才可用外部连接的早期语法。

早期外部连接的语法结构与内部连接的语法结构类似，但由于内部连接没有 LEFT 或

www.TopSage.com

RIGHT 关键字(当然也没有 OUTER 或 JOIN 关键字)，早期外部连接语句需要借助表 4-11 所示的几个特殊运算符。

表 4-11

运 算 符	ANSI
*=	LEFT JOIN
=*	RIGHT JOIN

回到本章外部连接的第一个示例。该示例使用 pubs 数据库，语句如下所示：

```
SELECT sso.SpecialOfferID, Description, DiscountPct, ProductID
FROM Sales.SpecialOffer sso
JOIN Sales.SpecialOfferProduct ssop
ON sso.SpecialOfferID = ssop.SpecialOfferID
WHERE sso.SpecialOfferID != 1
```

同样，只要去掉 LEFT OUTER JOIN 关键字，并使用 WHERE 子句代替 ON 运算符；或者，在这种情况下，在已有的 WHERE 子句上添加 ON 条件：

```
SELECT sso.SpecialOfferID, Description, DiscountPct, ProductID
FROM Sales.SpecialOffer sso,
   Sales.SpecialOfferProduct ssop
WHERE sso.SpecialOfferID *= ssop.SpecialOfferID
AND sso.SpecialOfferID != 1
```

如果运行它(建议改变需要的兼容级别，本例只是想告诉您这一类型的代码已经过时了)，那么返回的结果不变。RIGHT JOIN 早期语法结构的功能基本相同。

4.6.3 交叉连接的早期语法结构

早期的交叉连接无疑是最简单的。要使用早期语法来建立交叉连接，并不需要做很大改动，不需要在 WHERE 子句的 FROM：TableA. ColumnA = TableB. ColumnA 中加入任何元素。

看一下本章交叉连接中使用的第一个示例，ANSI 语法的语句如下：

```
USE Chapter4DB

SELECT v.VendorName, a.Address
FROM Vendors v
CROSS JOIN Address a
```

要将其转换为早期语法结构，只要去掉 CROSS JOIN 关键字并添加一个逗号：

```
USE Chapter4DB

SELECT v.VendorName, a.Address
FROM Vendors v, Address a
```

返回的结果与使用 ANSI 语法的交叉连接返回的结果相同：

```
VendorName                          Address
---------------------------         --------------------
```


www.TopSage.com

```
Don's Database Design Shop        1234 Anywhere
Don's Database Design Shop         567 Main St.
Don's Database Design Shop         999 1st St.
Don's Database Design Shop        1212 Smith Ave
Don's Database Design Shop         364 Westin
Dave's Data                       1234 Anywhere
Dave's Data                        567 Main St.
Dave's Data                        999 1st St.
Dave's Data                       1212 Smith Ave
Dave's Data                        364 Westin
The SQL Sequel                    1234 Anywhere
The SQL Sequel                     567 Main St.
The SQL Sequel                     999 1st St.
The SQL Sequel                    1212 Smith Ave
The SQL Sequel                     364 Westin

(15 行受影响)
```

所有版本的 SQL Server 和大部分数据库管理系统都支持早期的语法结构。

4.7 联合

对比了新语法和旧语法的不同，下面介绍在不同语法中用法相同的操作符——UNION 操作符。UNION 是一个特殊的运算符，用于使两个或两个以上的查询产生一个结果集。

联合不是真正的连接，作用更像将一个查询返回的数据附加到另一个查询结果的末尾(这样描述功能上可能不大恰当，但更容易让人理解联合的概念)。JOIN 将信息水平连接(添加更多列)，而 UNION 将数据垂直连接(添加更多行)，联合的功能如图 4-1 所示。

当使用 UNION 处理查询时，要注意以下几个关键点。

- 所有 UNION 的查询必须在 SELECT 列表中有相同的列数。即如果在 SELECT 列表中第一个查询选择了 3 列，则第二个查询(和 UNION 后面的任意查询)也要选择 3 列。如果第一个查询选择了 5 列，则第二个查询也要选择 5 列。不管第一个查询选择多少列，随后的查询必须选择相同的列。
- UNION 返回的结果集的标题仅从第一个查询获得。若第一个查询的 SELECT 列表为“SELECT Col1, Col2 AS Second, Col3 FROM...”，则不管随后的查询的列如何命名或取别名，使用 UNION 返回的列的标题分别为 Col1、Second 和 Col3。
- 查询中的对应列的数据类型必须隐式一致。注意这里没有要求相同的数据类型，只要数据类型可隐式转换(第 1 章的图 1-3 列出了可隐式与显式转换的数据类型的转换表)。如果第一个查询的第二列的数据类型为 char(20)，则第二个查询的第二列的数据类型可为 varchar(50)。但是，因为联合是基于第一个查询，所以第二个结果集中记录长度超过 20 个字符的部分将会被忽略。
- 与其他非 UNION 查询不同，UNION 查询的默认返回选项为 DISTINCT，而不是 ALL。这可能确实让人迷惑。在其他查询中，不管记录是否相同，都返回所有的行，但 UNION 查询不同。除非在 UNION 查询中使用 ALL 关键字，才能返回重复的行。

www.TopSage.com

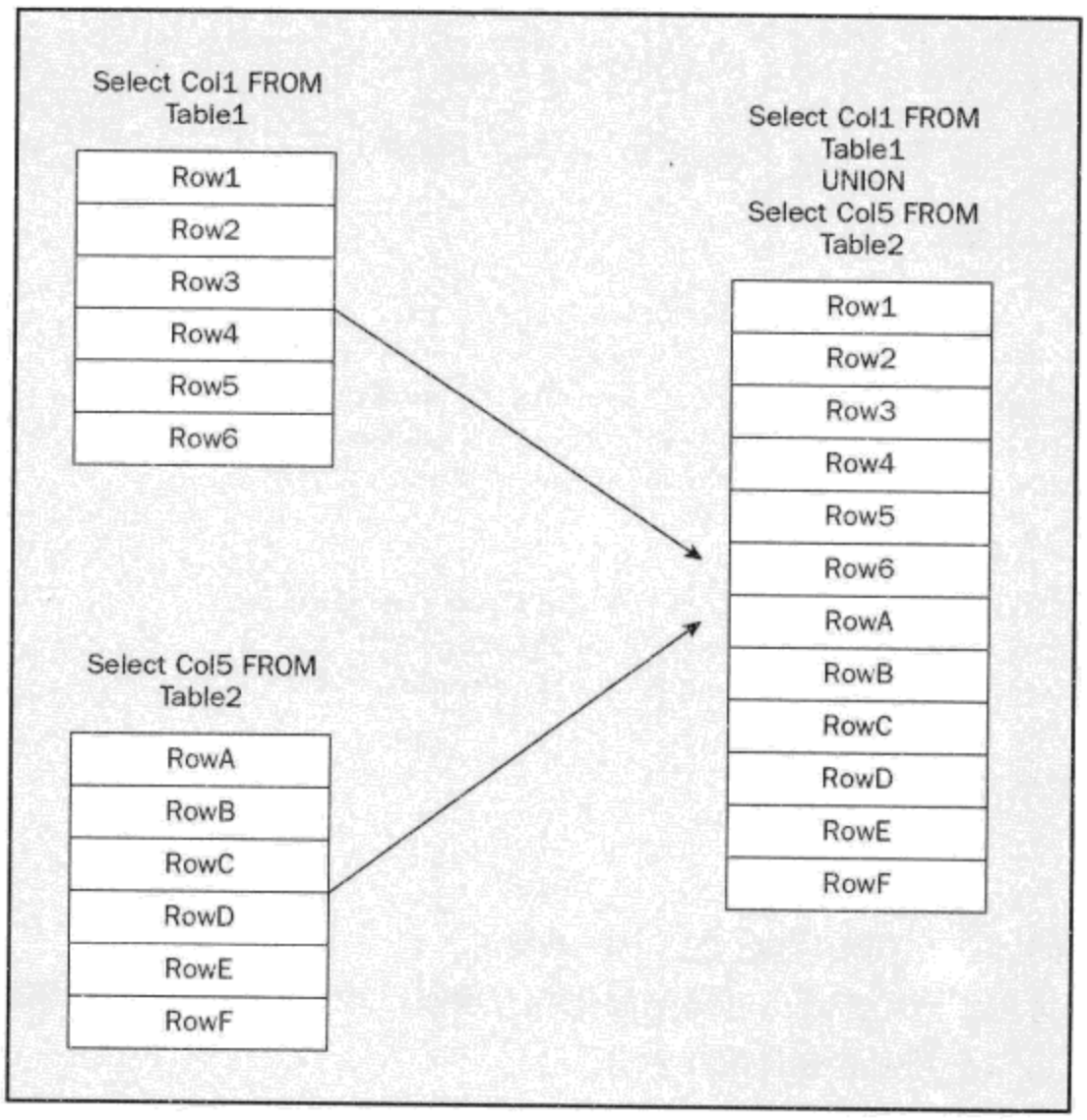


图 4-1

同样，下面使用示例来介绍 UNOIN。

试一试 联合

先看一个实际使用的 UNION 语句(实际工作中碰巧有，但不是很常见)。对于本示例，假设将 AdventureWorks 数据库移到了一台新设备，因此要向所有客户和供应商发邮件进行通知。需要返回所有人的姓名和 email 地址列表，以便更新地址。可以使用以下的查询语句：

```
USE AdventureWorks2008

SELECT FirstName + ' ' + LastName AS Name, pe.EmailAddress
       EmailAddress
FROM Person.Person pp
JOIN Person.EmailAddress pe
   ON pp.BusinessEntityID = pe.BusinessEntityID
JOIN Sales.Customer sc
   ON pp.BusinessEntityID = sc.CustomerID

UNION

SELECT FirstName + ' ' + LastName AS Name, pe.EmailAddress
       EmailAddress
FROM Person.Person pp
```


www.TopSage.com

```
JOIN Person.EmailAddress pe
   ON pp.BusinessEntityID = pe.BusinessEntityID
JOIN Purchasing.Vendor pv
   ON pp.BusinessEntityID = pv.BusinessEntityID
```

只返回一个结果集。

```
Name                      EmailAddress
--------------------      ------------------------------------
A. Scott Wright           ascott0@adventure-works.com
Aaron Adams               aaron48@adventure-works.com
Aaron Allen               aaron55@adventure-works.com
...
...
Zachary Wilson            zachary36@adventure-works.com
Zainal Arifin             zainal0@adventure-works.com
Zheng Mu                  zheng0@adventure-works.com

(10274 行受影响)
```

示例说明

UNION 语句从两个查询中返回了 1 个结果集。

SQL Server 运行了两个查询，并将结果集相互叠加，形成一个联合的结果集。注意，返回列的标题都来自第一个查询的 SELECT 列表。

下面介绍第二个示例，用于说明 UNION 语句如何处理重复行——实际上 UNION 与一般查询相反，默认为丢弃重复记录(在先前的查询中，除非使用了 DISTINCT 关键字，默认为保留所有的重复记录)。该示例没有任何现实意义，但运行快，可说明上述问题。

本示例从选择结果创建两个表。然后向每个表插入 3 行数据，并使两个表之间的 1 行数据相同。如果执行使用 ALL 关键字的查询，则返回每一行(其中的 6 行)。如果执行使用 DISTINCT 关键字的查询，则仅返回 5 行记录(丢弃一重复行)。

```
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')
```


www.TopSage.com

```
INSERT INTO UnionTest1
VALUES
   ('CCC')

INSERT INTO UnionTest2
VALUES
   ('CCC')

INSERT INTO UnionTest2
VALUES
   ('DDD')

INSERT INTO UnionTest2
VALUES
   ('EEE')

SELECT col2
FROM UnionTest1

UNION

SELECT col4
FROM UnionTest2

PRINT 'Divider Line--------------------------'

SELECT col2
FROM UnionTest1

UNION ALL

SELECT col4
FROM UnionTest2

DROP TABLE UnionTest1
DROP TABLE UnionTest2
```

返回的主要信息如下所示(会发现某一行在这里受到影响——忽略它直到得到可视化的查询结果):

```
col2
----
AAA
BBB
CCC
DDD
EEE

(5 行受影响)

Divider Line--------------------------
col2
----
AAA
```


www.TopSage.com

```
BBB
CCC
CCC
DDD
EEE

(6 行受影响)
```

第一个结果集是由不使用任何附加参数的简单 UNION 语句返回。可以看到有 1 行被丢弃。即使在两个表中都插入了“CCC”记录，由于 UNION 默认删除重复记录，因此只显示了其中 1 行。

对第二个 UNION 语句稍做改动。这次使用 UNION ALL，而 ALL 关键字可保证返回所有的行。所以丢弃的行在最后的查询中又突然两次出现。

4.8 本章小结

在 RDBMS 中，想要的数据通常分布在多个表中。连接可以通过多种方法将多个表中的数据组合起来。

- 要排除不匹配的字段时，使用内部连接。
- 要检索尽可能匹配的数据，但又要包含 JOIN 一侧的表的所有数据时使用外部连接。
- 要检索尽可能匹配的数据，但又要包含 JOIN 两侧的表的所有数据时使用完全连接。
- 要基于两个表的记录建立笛卡儿积时使用交叉连接。交叉连接通常用于科学领域和生成测试数据。
- 要将第二个查询结果附加到第一个查询结果时使用联合。

对于内部连接和外部连接来说，有两种不同的语法结构。这里提供了早期的语法结构有助于处理早期的代码，但本书提供的较新的 ANSI 语法结构更为常用，且可读性更好，不易造成早期的语法结构的多义性，而且将来的 SQL Server 版本将无限期支持 ANSI 语法结构。

本书接下来的几章要学习如何构建表，以及如何将其相互关联。这样要连接的列的概念会更加清楚。

4.9 练习

1. 针对 AdventrueWorks2008 数据库编写一个查询语句，要求返回 Name 列并包含 NationalIDNumber 为 112457891 的雇员的姓。

www.TopSage.com

第 5 章

创建和修改数据表

每次讲授创建数据库、表、键以及约束的 T-SQL 代码时，总有人问我同样的问题："难道就不能在 GUI 工具中来做这些工作吗？"答案无疑是肯定的。接着，他们通常又会问："那为什么花时间学习从来不用的东西呢？"答案同样是明确的——即将在类似正则的基础上使用正则语法。现实中您可能很少从头开始写代码，但是由于在您所从事的所有大型数据库项目中大部分都需要验证以及编辑代码，因此最好理解代码的工作原理。

本章将介绍创建自己的数据表的语法。还将学习如何使用 SQL Management Studio 来帮助完成这个任务(在明白如何使用之后)。

然而，在深入学习创建表和其他对象的实际语句之前，需要首先了解完全限定对象名的约定，并稍微了解一下对象所有者的约定。

5.1 SQL Server 中的对象名

到目前为止，本书所执行的所有查询都采用了简单的命名。在运行任何查询之前，本书都要求在 Query Analyzer 中切换活动数据库，这样有助于查询正常执行。为什么要这样呢？其实 SQL Server 在识别和定位在查询以及其他语句中命名的对象时，只查看很小的范围。例如，只提供表名，而没有其他任何附加信息，但是实际上 SQL Server 表(和其他类似的 SQL Server 对象)有 4 层命名约定。完全限定的命名如下所示：

```
[ServerName.[DatabaseName.[SchemaName.]]]ObjectName
```

在对某个对象执行操作时，必须提供对象名，但是上面代码中对象名左边的几个部分都是可选的。实际上在很多情况下，这些都是不需要的，因此省略掉了。然而，在创建对象之前，熟练掌握命名的各部分还是应该的。所以本章将从对象名左边开始按顺序介绍。

5.1.1 模式名

如果使用模式(大部分老的数据库没有使用模式，但是模式在以后似乎会变得越来越重要)，那么需要指出对象是在哪种模式下的。完全有可能存在两个同名的对象，但是在不同的模式下。如果想访问不在默认模式(基于登录名设置的)下的对象，那么需要特别指明对象的模式名称(即

www.TopSage.com

所有权)。例如，AdventureWorks2008 数据库是我所见过的最糟糕的模式使用方法之一，再看一下下面这个要获取雇员和雇员所居住城市的列表的查询：

```
SELECT
 e.NationalIDNumber, p.FirstName,p.LastName, City
FROM
 HumanResources.Employee e
INNER JOIN
 Person.Person p on p.BusinessEntityID = e.BusinessEntityID
INNER JOIN
 Person.BusinessEntityAddress a on p.BusinessEntityID = a.BusinessEntityID
INNER JOIN
 Person.Address pa on pa.AddressID = a.AddressID
```

这个示例使用了分布在两个模式下的 4 个表。如果两个相关模式(HumanResources 模式和 Person 模式)中的某一个正好是默认模式，那么在命名该模式下的表时，可以省略模式名称。在这里，为了安全起见，命名了所有模式。

提示：

这是我又一次讨论一致性问题。如果打算继续使用模式，那么我强烈推荐在所有查询中都使用两部分命名(模式和表名称)。改变用户默认模式实在是太简单了，也很容易给模式取其他的别名，以至于造成当前默认模式的失效。如果在数据库设计时根本没有采用不同的模式，那么最好完全省略模式(这样设计过程中代码可读性更强)，但是要注意，在将来开始使用模式时，这可能要付出代价。

1. 关于模式的进一步讨论

SQL 的 ANSI/ISO 标准在很早以前就有所谓模式的概念。虽然在 SQL Server 中很早就存在同样的概念，但是指代的对象有些不同(实际上，即使以同样的方式使用，还是有不同的内涵)。因此，在 SQL Server 2008 以及其他如 Oracle 这样的数据库中所指的“模式”，通常是 SQL Server 2000 及其以前版本中所指的“所有者”。

在以前，模式的概念一直都不是很清晰的。然而由于现在这个概念仍很重要，因此 Microsoft 添加了与模式相关的几个新概念，来简化模式的问题。然而，如果需要处理与以前 SQL Server 版本的向后兼容性问题，则要么避免使用这个新特征，要么使用与之相关的技巧——这意味着“所有权”(在以前版本中是广为人知的)这个概念依然存在较大的争议。

总有一些人喜欢在以前的设计中使用所有权，但是我肯定不是他们当中的一个。现在，关键是要知道，“模式”在概念上与从前的“所有权”有重叠。这两个术语都在被使用。模式在处理 SQL Server 的其他方面也变得很重要，如通知服务(Notification Services)。

下面来关注什么是模式，模式是如何工作的。

在以前发布的版本中，所有权(在当时就是这么命名的)实际上就是通过完全限定的名称来识别是谁“拥有”这个对象。通常，所有者是创建该对象的用户，或是数据库的所有者(通常指 dbo——稍后将讲述 dbo)。在 SQL Server 2008 中，工作原理是一样的，但是对象被赋予模式，而不是所有者。一个所有者与特定的登录名相关，而模式可以为多个登录名所共享，一个登录名也可以拥有多个模式。

www.TopSage.com

在默认情况下，只有当用户是 sysadmin 系统角色，或者 db_owner 或 db_ddladmin 数据库角色时，才能在数据库中创建对象。

提示：

这里提到的角色只是 SQL Server 2008 中众多可用的系统与数据库角色中的几个。根据角色可能被使用的方式赋予角色一个逻辑权限集合。当将一个特定的角色赋予某个人时，可以给予他该角色具有的所有权限。

个体用户可以被赋予一定权限来创建特定类型的数据库及系统对象。如果这些个体用户确实已经创建了一个对象，那么这个对象被赋予那个登录名列出的任意默认模式。

注意：

并不是说存在一个特征，就应使用这个特征。将 CREATE 权限授予个体用户也可能出现麻烦的事情。要想跟踪谁创建了什么、在什么时候、为什么创建，几乎是不可能的。简单地说，要将 CREATE 权限限于 sysadmins 或 db_owner 安全角色的成员。

2. 默认模式：dbo

无论谁创建了数据库，它都会被认为是“数据库所有者”，即 dbo。dbo 在数据库里面创建的任何对象都应该带有 dbo 模式，而不是个体的用户名。

例如，假设我是某个数据库的日常用户，我的登录名是 MySchema，并且我被赋予了这个数据库的 CREATE TABLE 权限。如果我创建了一个名为 MyTable 的表，那么所有者限定的对象名是 MySchema.MyTable。注意，由于这个表有特定的所有者，除了我(这里“我”指的是 MySchema)以外的所有 MySchema.MyTable 的用户需要提供由两部分组成(模式–限定)的名称，这样 SQL Server 才能解析这个表的名称。

现在，假设还有一个用户，登录名为 Fred。Fred 是数据库的所有者(仅与任何 db_owner 的成员相对)。如果 Fred 使用与 MySchema 所使用的同样的 CREATE 语句来创建一个名为 MyTable 的表，那么模式限定的表名称是 dbo.MyTable。另外，因为 dbo 正好是默认的所有者，所以任何用户都可以用 MyTable 来引用该表。

还需要指出的是，sysadmin 角色的成员(包括 sa 登录名)总是 dbo 的别名。即无论谁实际上拥有数据库，sysadmin 角色的成员总拥有完全的权限，就好像是 dbo 一样。而且，如果未进行显式定义，sysadmin 角色的成员创建的任何对象都显示所有权为 dbo。与此不同的是，由 db_owner 数据库角色的成员创建的对象的默认模式不是 dbo，这些对象将被赋予特定用户所设定的默认模式(可以是任何东西)。虽然有点奇怪，但事实就是这样。

提示：

我从和一些在 Microsoft 的老朋友谈话中了解到，他们似乎觉得模式这个名称不错，而且很乐意这个名称的改变。我也很乐意这个改变，但主要是因为这些改变可以进一步简化模式的使用，而并不是说这是一个新的特征，大家就要抢着使用。

不管使用模式做什么，添加模式都将增加数据库的复杂性。虽然模式能解决设计中的组织问题，但是这些问题通常可以采用其他方法解决，从而产生更加用户友好的数据库。另外，模

www.TopSage.com

式虽然是 ANSI/ISO 兼容的概念，但并没有得到其他主要的 RDBMS 产品同样的支持。这意味着使用模式将影响支持多平台的编码。

5.1.2 数据库名

完全限定的命名约定中的下一个条目是数据库名。有时需要从默认数据库或者当前数据库以外的数据库检索数据。可能实际上想连接(JOIN)几个数据库的数据。数据库限定的命名提供了这个功能。例如，假设登录时，AdventureWorks2008 是当前数据库，而又希望引用 Accounting 数据库(本章后面将创建)中的 Orders 表，那么可以通过 Accounting.dbo. Orders 来引用该表。因为 dbo 是默认的模式，所以也可以通过 Accounting..Orders 来引用。如果在 MyDatabase 数据库中，名为 MySchema 的模式拥有一个名为 MyTable 的表，那么可以使用 MyDatabase.MySchema. MyTable 来引用该表。记住，当前数据库(由 USE 命令决定；如果使用 SQL Server Management Studio，那么从下拉框中选择)总是默认的数据库。所以，如果只需要当前数据库中的数据，那么不需要在完全限定的名称中包括数据库名。

5.1.3 通过服务器命名

除了命名所连接的服务器上的其他数据库以外，还可以“链接”到另外一个服务器。链接的服务器允许执行跨多个服务器的连接(JOIN)，甚至是不同类型的服务器(SQL Server、Oracle、DB2、Access——只要带有 OLE DB 提供程序就行)。在本书后面将有更多关于链接的服务器的内容，但是这里只需要知道：在命名层次结构中还有一层，即允许访问不同的服务器，而且其工作方式与数据库和所有者非常类似。

现在在前面的示例中增加了一些内容。假设已经创建了与名为 MyServer 的服务器的链接，现在希望从该服务器上的 MyDatabase 数据库中的 MySchema 所拥有的 MyTable 表中检索信息，那么完全限定的名称为 MyServer.MyDatabase.MySchema.MyTable。

5.1.4 回顾默认值

最后回顾一下命名层次结构中从右到左的各个层的默认值是怎样的。

- **对象名**：没有默认值——必须提供一个对象名。
- **所有权**：可以省略该值。在省略情况下，首先用当前用户名来解析，如果这个对象名的所有者不是当前用户，那么将使用 dbo 作为所有者。
- **数据库名**：除非提供了服务器名，这个也可以省略——在这种情况下，必须为 SQL Server 提供数据库名(其他服务器类型根据特定的服务器有所变化)。
- **服务器名**：可以提供链接的服务器名，但是在大多数情况下，省略该名称，这样 SQL Server 就是登录的默认服务器。

如果希望省略模式名，但仍然提供数据库或者服务器的信息，那么仍必须在所有者原本应该在的位置保留相应的“.”符号。例如，如果登录本地服务器，使用 AdventureWorks2008 数据库，但是希望引用在一个名为 MyOtherServer 的链接服务器上 AdventureWorks2008 数据库中的 Sales.Customer 表，那么可以通过 MyOtherServer.AdventureWorks2008..Customer 来引用该表。因为没有提供具体的模式名，所以将认为登录到该链接服务器的用户 ID 的默认模式或 dbo 就是所需要的对象(这里是 Customer)的模式。因为 Customer 表不是 dbo 模式的一部分，用户需要

www.TopSage.com

有一个默认的销售模式，否则在没有找到对象时会出现错误。通常，建议对想要引用的对象采用显示命名模式。

5.2 CREATE 语句

我们需要提供良好定义的语法来在数据库中创建对象。为此，使用 CREATE 语句。

现在看一下 CREATE 语句的完整结构，首先是最通用的形式。可以发现，CREATE 语句的开头都是一样的，然后是特定的细节。CREATE 的第一部分总是如下所示：

```
CREATE <object type> <object name>
```

下面介绍根据所创建对象的本质而不同的细节内容。

5.2.1 CREATE DATABASE

在这里，需要创建一个数据库，名为 Accounting，在以后开始创建表时还需要使用这个数据库。CREATE DATABASE 语句最基本的语法如上面示例所示。

```
CREATE DATABASE <database name>
```

注意：

需要指出的是，在创建一个新的对象时，除了创建者、系统管理员、数据库所有者(如果创建的对象是数据库，那么数据库所有者就是创建者)以外，其他人都无法访问。这允许您在创建对象后，进行所需要的任何修改，之后才明确允许其他人访问您创建的对象。

还需要注意的是：只能使用 CREATE 语句在本地服务器上创建对象(添加特定的服务器名称是不起作用的)。

这将得到一个看起来与模型数据库一样的数据库(在第 1 章中讨论过模型数据库)。不过因为用户的需求几乎总是不同的，所以看一下更完整的语法列表：

```
CREATE DATABASE <database name>
[ON [PRIMARY]
 ([NAME = <'logical file name'>,]
  FILENAME = <'file name'>
  [, SIZE = <size in kilobytes, megabytes, gigabytes, or terabytes>]
  [, MAXSIZE = size in kilobytes, megabytes, gigabytes, or terabytes>]
  [, FILEGROWTH = <kilobytes, megabytes, gigabytes, or terabytes|percentage>])]
  [LOG ON
 ([NAME = <'logical file name'>,]
FILENAME = <'file name'>
  [, SIZE = <size in kilobytes, megabytes, gigabytes, or terabytes>]
  [, MAXSIZE = size in kilobytes, megabytes, gigabytes, or terabytes>]
  [, FILEGROWTH = <kilobytes, megabytes, gigabytes, or terabytes|percentage>])]
   [ COLLATE <collation name> ]
  [ FOR ATTACH [WITH <service broker>]| FOR ATTACH_REBUILD_LOG| WITH DB_CHAINING
ON|OFF | TRUSTWORTHY ON|OFF]
[AS SNAPSHOT OF <source database name>]
[;]
```


www.TopSage.com

要注意上面的一些选项是互斥的(例如，如果创建了用于附加的数据库，那么除了文件位置以外，其他选项都是无效的)。有很多这样的情况，所以分别讲述。

1. ON

ON 用于两个地方：一是定义存储数据的文件的位置，二是定义存储日志的文件的位置。还可以看到 PRIMARY 关键字——这意味着随后是物理上存储数据的主文件组。也可以将数据存储在所谓的附属文件组里——这个用法超出了本书范围。现在只需要认定一个默认的概念：希望将所有的东西放在一个文件里。

注意：

SQL Server 允许将数据库存储在多个文件中，而且允许将这些文件放在一个逻辑组里面，称为文件组。文件组的使用是比较高级的概念，超出了本书的范围。

2. NAME

虽然这个选项用于指定定义的文件的名称，但只是一个逻辑名称——即 SQL Server 在内部使用该名称引用该文件。当需要修改(扩充或缩小)数据库和/或文件的大小时，需要使用这个名称。

3. FILENAME

顾名思义，这个选项——实际操作系统文件在磁盘上的物理名称，在该文件中将存储数据或日志(取决于定义哪个部分)。这里的默认值(假设您使用了一开始看到的简单语法)取决于是处理数据库本身还是日志。在默认情况下，文件位于\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL 主目录(或者您在安装时修改的 SQL Server 主目录)的\Data 子目录下。如果是处理物理数据库文件，那么名称将和数据库名称一样，但是带有.mdf 扩展名。如果是处理日志文件，那么名称和数据库文件的名称一样，但是有后缀_Log 以及扩展名.ldf。如果显式命名这些文件，那么可以指定其他扩展名，但是本书强烈建议您使用默认的 mdf(数据库)和 ldf(日志文件)扩展名。顺便说一下，附属文件的默认扩展名为.ndf。

要记住，虽然 FILENAME 是个可选的参数，但是只在开始介绍的极其简单的语法(基于模型数据库创建新的数据库)的情况下是可选的。如果提供任何附加的信息，那么必须包含显式定义的文件名——还要确保提供完整的路径。

4. SIZE

这里没有什么神秘的。正如字面意思一样——即数据库的大小。在默认情况下，大小的单位是 megabyte(兆字节)，但是可以通过在数字后面使用 KB 而不是 MB 来指定 kilobyte(千字节)单位，或者指定更大的单位 GB(gigabyte，千兆字节)或者甚至 TB (terabyte，兆兆字节)。要记住，这个值至少与模型数据库一样大，而且必须是整数(而不是带小数的)，否则将出错。如果没有为 SIZE 提供一个值，那么数据库的初始大小值与模型数据库一样。

www.TopSage.com

5. MAXSIZE

这个选项和字面意思也差不多，与 SIZE 参数相比只有一点区别。SQL Server 有一个机制允许数据库在必要的时候自动分配附加的磁盘空间(增加大小)。MAXSIZE 是数据库可以增加的最大大小。同样，在默认情况下，大小的单位是 megabyte，但是同 SIZE 一样，可以使用 KB、GB 或 TB 来指定不同的单位。稍微不同的是，这个选项没有固定的默认值。如果没有提供这个参数的数值，那么认为没有最大值——实际的大小可以大到磁盘空间用完为止。

如果数据库的大小达到了 MAXSIZE 参数指定的值，那么将开始出错，例如插入操作将不能执行。如果日志达到了最大值，那么在数据库中将无法执行任何日志活动(日志活动是数据库的主要活动)。我个人推荐设置警报(alert)。可以使用警报来预报某个条件的存在(如数据库或者日志几乎满了)。在第 19 章讲述如何创建警报。

注意：

本书推荐总是指定一个 MAXSIZE 的值，可以指定这个值使得至少留几兆字节以免用完整个磁盘空间。这样建议是因为磁盘空间完全用完时会出现一些问题，那样就无法永久存储任何信息。如果日志试图扩展，那结果可能是灾难性的。另外，在操作系统完全用完磁盘空间时偶尔也可能出现问题。

还有，如果在这个问题上打算采纳我的建议，需要注意可能存在同一个系统中有多个数据库的情况。如果指定每个数据库都可以占用整个磁盘空间直到预留的几兆字节，那么还有可能使用完整个磁盘空间的(如果每个数据库都扩充的话)。因此针对这样的情形，不存在绝对的答案。只是需要根据可能的使用率对空间作优先考虑，更严密地监控数据库大小并在 Windows Server 中设置警报来通知磁盘空间较低的情况。

6. FILEGROWTH

SIZE 设定了数据库的初始大小，MAXSIZE 决定数据库文件大小的最大值，FILEGROWTH 主要用于确定数据库达到这个最大值的速度。提供一个值来说明文件每次增加多少字节(以 KB、MB、GB 或 TB 为单位)。或者，也可以选择指定数据库文件每次增长的百分比。如果选择百分比，那么数据库文件大小将按照当前数据文件的大小以指定的百分比增长。因此，如果指定一个初始大小为 1GB 的数据库文件以 20%的速度增长，那么第一次扩展为 1.2GB，第二次为 1.44GB，以此类推。

7. LOG ON

LOG ON 选项允许指定哪些文件需要日志，以及这些文件位于什么位置。如果没有指定该选项，那么 SQL Server 将在一个单独文件中创建日志，日志文件默认的大小是数据文件大小的 25%。在大多数其他方面，日志文件与主数据库文件具有同样的文件规范参数。

提示：

强烈推荐将日志文件存储在与主数据文件不同的磁盘分区上。这样除了提供针对某个磁盘驱动器失效的额外安全措施外，还可以避免主数据文件与日志文件竞争磁盘文件的输入输出口。

www.TopSage.com

8. COLLATE

该选项处理排序、字母大小写以及是否对重音敏感问题。在安装 SQL Server 时，提供了默认的排序规则，但是可以在数据库层(以后在列层还可以看到)重写该默认值。

9. FOR ATTACH

可以使用该选项将已存在的一些数据库文件附加到当前的服务器上。这里的文件必须是数据库的一部分，并且在某个时候已经使用 sp_detach_db 恰当地与数据库分离了。这样贬低了 sp_detach_db 的功能，但是带 FOR ATTACH 的 CREATE DATABASE 命令具有可以访问 32 000 多个文件的优点，而 sp_attach_db 只限于 16 个文件。

如果使用 FOR ATTACH，那么必须完成文件位置信息中的 ON PRIMARY 部分的设置。CREATE DATABASE 参数列表的其他部分可以省略，只要把数据库的文件路径附加到与原来数据库分离时相同的文件路径中。

10. WITH DB CHAINING ON|OFF

如何以初学者能理解的方式讲述这个选项是个难题。确实要讲清楚该选项是非常困难的，也没有办法以“初学”的概念来描述。清楚了这一点，下面就讲述相关的缩版内容。

前面已经提到过，在 SQL Server 的以前版本中并不存在真正的“模式”的概念。而是存在“所有权”的概念。所有权可能引起的比较麻烦的问题是所谓的“所有权链”。即这样一种情况：如果 A 是一个对象的所有者，而 B 是另外一个对象的所有者，但是这个对象依赖于 A 所有的那个对象。这样可能存在一个又一个的人创建的对象依赖其他人创建的对象，这样，基于这些对象的所有权交织成了个很复杂的问题。

这个选项就是针对这样的跨数据库的所有权链的(A 的对象在 DB1 中， B 的对象在 DB2 中)。如果打开该选项，那么跨数据库的所有权链有效，而关闭该选项，则所有权链无效。应避免使用这样的所有权链，因为它们就像是瘟疫——数据库里面的瘟疫。

11. TRUSTWORTHY

这个选项是新添加的选项，为访问在 SQL Server 环境以外的系统资源和文件添加额外的安全层。例如，可能运行.NET 程序集访问网络上的文件，如果这样的话，那么必须检查数据库确保程序集是可以信赖的。

由于安全原因，这个选项在默认情况下是关闭的——在确实明白您将要做的事情和原因之后，再打开这个选项。

12. 构建数据库

现在准备构建数据库。下面是创建数据库的语句，但是要明白，在创建完全实用的数据库的过程中，数据库本身只是创建的众多对象中的一个。

```
CREATE DATABASE Accounting
ON
 (NAME = 'Accounting',
 FILENAME = 'C:\Program Files\Microsoft SQL
```


www.TopSage.com

```
Server\MSSQL10.MSSQLSERVER\MSSQL\DATA\AccountingData.mdf',
 SIZE = 10,
 MAXSIZE = 50,
 FILEGROWTH = 5)
LOG ON
 (NAME = 'AccountingLog',
 FILENAME = 'C:\Program Files\Microsoft SQL
Server\MSSQL10.MSSQLSERVER\MSSQL\DATA\AccountingLog.ldf',
 SIZE = 5MB,
 MAXSIZE = 25MB,
 FILEGROWTH = 5MB);
GO
```

现在是学习 SQL Server 中一些有用的信息工具的好机会。在第 4 章介绍了 sp_help，但是在这里，试着运行称为 sp_helpdb 的命令。该命令是特别为数据库结构信息定制的，如果对数据库本身而不是数据库包含的对象感兴趣的话，这个命令经常能提供更好的信息。sp_helpdb 带有一个参数——数据库名。

```
EXEC sp_helpdb 'Accounting'
```

执行上面的代码，实际上得到两个分开的结果集。第一个是基于数据库的(数据与日志)组合信息，如表 5-1 所示。

表　5-1

name	db_size	owner	dbid	created	status	compatibility_level
Accounting	15.00MB	sa	9	05 28 2008	Status=ONLINE, Updateability =READ_WRITE, UserAccess= MULTI_USER, Recovery=FULL, Version=598, Collation= SQL_Latin1_ General_CP1_ CI_AS, SQLSort- Order=52, IsAutoCreate- Statistics, IsAutoUpdate- Statistics, IsFullText- Enabled	90

www.TopSage.com

提示：

您所获得的每个字段的实际值可能与本例不同。例如，DBID 值会因您创建数据库的数目和顺序的不同而不同。各种状态信息会因您创建数据库时的服务器选项设置以及随后的一些选项改动的不同而不同。

注意 db_size 属性是数据库大小与日志大小的总和。

第二个表提供了构成数据库的不同文件的信息，包括文件当前的大小和增长设置，如表 5-2 所示。

表 5-2

name	fileid	filename	file-group	size	maxsize	growth	usage
Accounting	1	C:\Program Files\ Microsoft SQL Server\mssql1\ data\ AccountingData .mdf	PRIMARY	10240KB	51200KB	5120KB	data only
AccountingLog	2	C:\Program Files\ Microsoft SQL Server\mssql1\ data\ AccountingLog .ldf	NULL	5120KB	25600KB	5120KB	log only

在创建表并插入数据之后，数据库大小将根据需要自动增长。

5.2.2 CREATE TABLE

创建表的第一部分与创建任何对象的完全一样——还记得那一行代码吗？下面再次列了出来：

```
CREATE <object type> <object name>
```

因为要创建的是表，所以可以更具体些：

```
CREATE TABLE Customers
```

使用 CREATE DATABASE 时，只需给出前面 3 个关键词，就可以根据建立模型数据库的指导原则创建一个数据库。但是在创建表时，没有模型，因此需要提供一些更具体的形式，如列、数据类型以及特殊的运算符。

www.TopSage.com

下面看看具有更多选项的语法：

```
CREATE TABLE [database_name.[owner].]table_name
(<column name> <data type>
{[DEFAULT <constant expression>]
  |[IDENTITY [(seed, increment) [NOT FOR REPLICATION]]]
  [ROWGUIDCOL]
  [COLLATE <collation name>]
  [NULL|NOT NULL]
  [<column constraints>]
  |[column_name AS computed_column_expression]
  |[<table_constraint>]
  [,...n]
)
[ON {<filegroup>|DEFAULT}]
[TEXTIMAGE_ON {<filegroup>|DEFAULT}]
```

其选项并不很多，但是为了简单明了，还是省略一些部分。同样，分别看一下每个选项，从第二行开始(第一行已经讲过)。

1. 表名和列名

名称有什么意义？坦率地说，很多。您可能还记得我在第 2 章中对名称的说明。我保证那不是您最后一次听见我提到这个主题，这里也不是最后一次。

表和列的命名规则通常与数据库对象的命名规则相同。SQL Server 文档将这些归为标识符的规则，与第 1 章最后看到的规则一样。这些规则其实很简单；这里是要介绍如何命名对象的一些概念——不是指定 SQL Server 接受什么命名的规则，而是如何命名表和列，使得这些命名有效并且有意义。

命名数据库对象的“标准”多如牛毛——特别是表和列。本人的规则是很简单的。

- 对于名称的每个单词，要求首字母大写，而其他字母小写。
- 名称尽量短，但是要具有描述性。
- 限制使用缩写。只有在大家都能理解时才能采用缩写。本人使用的缩写示例包括“ID”表示标识(identification)、“No”表示数字(number)、“Org”表示组织(organization)。要使名称长度合适，有时需要放宽缩写标准，但是记住，首先和最重要的是，名称的意思要清楚明了。
- 当基于其他表来构建表时(通常称为链接或关联表)，需要在新的表名中包含所有父表的名称。例如，假设有一个电影数据库，很多明星出现在很多电影中。如果有一个 Movies 表和一个 Stars 表，那可能使用一个名为 MovieStars 的表将这两个表关联起来。
- 当名称中有两个单词时，不要用任何分隔符(单词直接连接起来)——通过将每个单词的首字母大写来区分单词。

我在命名问题上与其他数据库人员有过不少争论。例如，您会发现很多人认为应该采用下划线符号(_)来分隔单词。为什么我不采用这个方法呢？其实这是个简单的问题。下划线会带来下面一些不同的问题。

- 首先，很多人在输入下划线符号会感到比较麻烦，因为需要手离开固定的键盘位置——这样会导致大量打字错误。

www.TopSage.com

- 其次，在文档中，表和列名出现下划线是很正常的。根据字体不同，当文本带下划线时，将不能与下划线符号区别显示——这将带来混乱以及错误。
- 最后，这需要多输入一个符号(当然，这有点挑刺的意味)。

提示：

采用空格分隔单词也是一种选择。如果您回顾第 1 章的内容，您就会明白这并不是个真正意义上的选项——这是个很坏的做法，会引起无数的错误。这个方法用来帮助 Access 升级，但我仍要批评那些还提这个用法的人——我确信它们有意义，但是它们现在是造成数据库世界很多悲剧的根源之一。

当然，上面这些规则不是不变的，这只是我用于命名表的规则的一部分。我发现这些规则可以避免很多麻烦。我希望对您来说也是这样。

注意：

一致性，一致性，还是一致性。每次我讲课时，我总是告诫我的学生：这是我会不断重复的词，而且它在命名方面的重要性比在任何其他方面更明显。如果您必须遵循一个规则，那么就采用这个规则，不管您的标准是什么。如果出于某种原因需要缩写，那么在每个地方都缩写(并且都是同样的缩写)。在命名时不管做什么，确保将它一致地应用于整个数据库——考虑有一个标准文档或者风格指导来确保其他开发人员使用与您一样的规则。这样能避免大量的错误，而且能节省用户了解数据库的时间。

2. 数据类型

这里内容不多——数据类型就是我在第 2 章讲述的。只需紧接着列名之后提供数据类型即可，没有默认的数据类型。

3. DEFAULT

对此，本书将在关于约束的一章中作详细讨论。但是现在只是说一下，默认值是在插入任何记录时用户没有为某列提供值时采用的值。如果要使用默认值，就必须紧跟在数据类型之后给定这个值。

4. IDENTITY

在数据库设计中，标识符值的概念是很重要的。本书在关于设计的章节中将比较详细地讨论使用标识列的方法。什么是标识列？在设定一个列为标识列时，SQL Server 自动分配一个顺序号给插入的每个行。SQL Server 开始计数的数字称为种子值，而这个值随着每行增加或减少的数量值称为增量。默认情况下是种子为 1，增量为 1，很多设计要求不要改动这个默认设置。不过，也可以设定种子为 3，增量为 5。在这种情况下，将从 3 开始计数，然后每次加 5，得到 8、13、18、23 等。

标识列必须是数值类型，在实际中，通常使用整型或大整型实现。

用法是很简单的，只要在列的数据类型之后紧跟上 IDENTITY 关键字。标识符选项不能和默认约束联合使用。如果想想会觉得这点是有道理的：如果每次增加或减少某个列值，怎么可

www.TopSage.com

能会有一个固定的默认值呢？

提示：

值得注意的是，标识列是按顺序工作的。即一旦设置了种子(起点)和增量，值只会增长(如果将增量设置为负数的话就减少)。没有自动机制能返回去填充已经删除的任何行。如果希望填充这样的空白行，那么需要使用 SET IDENTITY_INSERT ON，这个选项允许关闭(是的，设置为 on 意味着关闭自动标识列选项，即打开了插入您自己的值的功能，这样就关闭了自动功能)自动标识的过程来从当前连接插入。然而这样能产生麻烦，如果您不注意，或者其他人还像您这样使用系统的话，就会出现错误，所以使用时必须小心。

标识列最常见的用法是生成一个新值作为每行的标识符，即标识列通常用于创建表的主键。然而，要记住，IDENTITY 列和 PRIMARY KEY 是完全不同的概念，即不会因为有一个 IDENTITY 列就说明这个值是唯一的(例如，可以重新设置种子值，使用前面用过的值)。IDENTITY 值通常用于 PRIMARY KEY 列，但并不是必须这样使用。

注意：

如果您以前使用过 Access，就会发现 IDENTITY 列很像 AutoNumber 列。主要区别是在 SQL Server 中对该列有更多的控制。

5. NOT FOR REPLICATION

这个选项是很难对付的。因此在这里至少会跳过一部分内容，在关于复制的章节里再讲述。

注意：

简单地讲，复制就是自动处理的过程，广义地讲，即复制数据库的一部分或全部信息到其他数据库中。其他数据库可以和原数据库位于同一个物理机器上，也可以位于远程机器上。

NOT FOR REPLICATION 参数决定了当列(通过复制)发布到另外一个数据库时，是为新的数据库分配一个新的标识值，还是保留已有的值。本书后面会讲述更多关于这个问题的内容。

6. ROWGUIDCOL

这个选项也与复制有关，在很多情况下，和标识列有相同的目的。我们已经知道如何使用标识列以一种简单的方式来确保每行有一个唯一的值，从而可以用于标识每行。然而，这个方法在处理复制以及其他分布式环境中很容易产生错误。

想想看，既然标识列可以从一个设定值开始计数，怎样使这个值在不同数据库中不重复？现在考虑复制这个值，将原来在不同数据库中的行合并在同一个数据库中。这样列中就有复制的值，而这个值被认为是每行的唯一标识。

很多年来，解决这个问题的一般方法是在需要复制的不同数据库中使用不同的种子值。例如，数据库 A 以 1 开始计数，数据库 B 以 10 000 开始计数，而数据库 C 以 20 000 开始计数。现在可以安全地将这些数据库公布到同一个数据库中——不过这只是暂时的。只要数据库 A 插入的记录数多于 9 999，那么就会有麻烦了。

您可能会问："那为什么不用 100 000 或者 500 000 来分割值呢？"但是即便如此，当

www.TopSage.com

某个表具有大量活动时，早晚还是会有麻烦的——这就是需要使用 ROWGUIDCOL 的原因。

什么是 ROWGUIDCOL？其实这个很像标识列，即用于唯一标识表的每行。不同的是在哪种程度上确保该值是唯一的。SQL Server 没有采用数字计数，而是采用全局唯一标识符(Globally Unique Identifier)。标识值通常(除非修改了某些设置)是在时间上唯一的，而不是在空间上唯一的。因此，可以有两个表在运行，为两个表的不同行都分配相同的标识值。而刚开始一切都好，但是当想将两个表的行放到一起作为一个复制的表就会有很大的问题。唯一标识符有时仍被称为 GUID，在空间和时间上都是唯一的。

提示：

GUID(或者现在更通用的 UUID——看上去相同，但在执行相同任务时更有帮助)如今在计算中逐步得到广泛应用。例如，如果检查注册表，可以发现大量的 GUID。一个 GUID 是一个 128 位的值——从算术上讲，就是十进制的 38 位。如果每秒钟生成一个 GUID，从理论上讲，需要几百万年才会生成同样大小的值。

GUID 是通过信息组合生成的——每个 GUID 在空间或者时间上都是唯一的。如果将它们组合起来，从统计理论上讲，得到的值在空间和时间上也将是唯一的。

在通常的编程中，有一个 Windows API 调用用于生成 GUID。但是，除了对一个列设置 ROWGUIDCOL 选项外，SQL 还有一个特殊的函数返回一个 GUID——这个函数称为 NEWID() 函数，可以在任何时候调用。

7. COLLATE

这个选项与在 CREATE DATABASE 命令中的作用基本一样，主要的区别是作用范围(这里是在列的层次而不是数据库的层次定义的)。

8. NULL/NOT NULL

这个选项非常简单——说明指定的列是否接受 NULL 值。在第一次安装 SQL Server 时，默认的列值是 NOT NULL，除非指定允许为空。然而，有很多不同的设置可以改变这个设置，从而影响这个默认值。例如，使用 sp_dbcmptlevel 存储过程设定一个值或者设置 ANSI-兼容选项可以改变这个选项的设置。

注意：

作者强烈建议在构建的每个表的每个列都显式说明 NULL 选项。为什么呢？正如前文所述，有大量的不同设置可以影响系统使用列值是否为空的默认设置。如果依赖这些默认值，那么可能在以后发现脚本并没有正确地工作(因为您或者其他人改变了相关的设置而没有意识到这个改变的所有影响)。

9. 列约束

本书有一整章的内容讲述约束，因此这里就不多讲了。不过现在概述一下列约束问题——简单地讲，列约束就是对单个列设置的关于该列可插入数据的限制和规则。

例如，如果有一个列要存储年中的月份，可能定义列为 tinyint 类型——但是这个不能阻止

www.TopSage.com

某个人在该列插入数 54。因为 54 是不正确的数据(不能代表某个月份)，所以需要提供一个约束，说明列中的数据必须在 1~12 之间。在第 6 章可以看到如何设置约束。

10. 计算列

也可以创建一个本身没有任何数据的列，但列值是由表中其他列动态生成的。如果想想，可以会觉得有点奇怪，因为可以在查询时指定列值。但是，这确实能给很多应用带来方便。

例如，假设在做一个货品计价系统。希望存储已销售的商品的数量信息以及价格信息。过去通常的做法是添加列来存储的这个信息，再添加另外一列存储扩充的信息(价格乘以数量)。然而，这样会导致不必要的磁盘空间浪费以及在总计值和基值不同步时的维护问题。可以通过采用计算列来解决这个问题，即定义计算列的值为价格乘以数量。

下面是具体的语法：

```
<column name> AS <computed column expression>
```

第一项有点不同——这里提供与值相对应的列名。这个只是用于引用根据 AS 关键字后面的表达式计算出的值的别名。

接着是计算列的表达式。该表达式可以是任何的常规表达式，使用字面值或者同一个表的列值。因此，在价格和数量的示例中，可以将列定义为：

```
ExtendedPrice AS Price * Quantity
```

对于使用字面值的示例，假设总是赚取比商品成本高 20%的固定利润。可用一列跟踪成本信息，而 ListPrice 列使用计算列：

```
ListPrice AS Cost * 1.2
```

这很简单，不过还有一些警告和限制条件。

- 不能使用子查询，而且值不能来自其他不同的表。
- 在 SQL Server 2000 与之前的版本中，不能使用计算列作为键(主键、外键、唯一键)的任何部分，也不能和默认约束一起使用。在 SQL Server 2005 中，可以在约束中使用计算列(然而在使用时，必须将计算列标记为持久的)。
- 可以在计算列上创建索引，但是必须采用特定的步骤。第 9 章将探讨利用索引时要采用的这些步骤。

本章稍后将给出具体示例说明如何使用计算列。

提示：

实际上，很少听见关于使用计算列的争论，我对此感到很吃惊。标准化数据的规则认为在表中不应有一个可从其他列派生的列——而计算列正是这种性质的！

传统的标准化的追求者并没有攻击该问题，可以把计算列视为一个折衷的问题。在默认情况下，没有存储数据两次，也没有派生值与基值不一致的问题，因为派生值是根据基值动态计算的。但最终仍得到了所希望的结果。注意，如果索引计算列，那么实际上存储了数据(这是必须为索引做的)。不过，这能提高读性能。

这一方法不能解决所有派生数据问题，但是在很多情况下确实是很好的助手。

www.TopSage.com

11. 表约束

表约束和列约束很相似，都是对可以插入表中的数据进行限制。它们之间有一点不同的是：表约束可以基于多个列。

同样，有关约束的章节将讲述这个内容，但是表级别的约束包括 PRIMARY 和 FOREIGN KEY 约束以及 CHECK 约束。

注意：

那为什么 CHECK 约束是表约束呢？既然它会影响给定列的设置的约束，为什么它不是列约束呢？其实，它即是表约束，也是列约束。如果它只基于单个的列，那么满足列约束的规则。如果(CHECK 约束可以)依赖于多个列，那么就是表约束。

12. ON

在处理数据库创建时，提到过可以创建不同的文件组。表定义中的 ON 子句就是用于指定希望表位于哪个文件组的一种方法。可以将给定的表放在一个特定的物理设备上，或者，就按大多数时候做的那样，省略 ON 子句，那样表将位于默认文件组(除非设置成其他的，否则就是 PRIMARY)中。本书关于性能调整一章将具体讲述该选项的用法。

13. TEXTIMAGE_ON

这个选项和前面刚讲过的 ON 子句基本上相同，不同的是将该选择将表的特定部分移动到不同的文件组中。这个子句只有在表的定义中有 text、ntext 或 image 列时才有效。当使用 TEXTIMAGE_ON 子句时，只是将 BLOB 信息移动到分离的文件组中——表的其他部分还在默认文件组或者 ON 子句选择的文件组中。

注意：

将数据库分开到多个文件，然后将这些文件存储在不同的物理磁盘中，这可能会引起严重的性能问题。当这么做时，意味着从不同驱动器中获得 I/O。有关这个问题的讨论超出了本书的范围，但是要记住：在需要收集更多的信息时，需要考虑 I/O 性能问题。

14. 创建一个表

前面已经讲了很多内容了，现在开始构建几个表。

在本节开始，首先看看标准的 CREATE 语法：

```
CREATE <object type> <object name>
```

然后就开始具体创建 Customers 表(实际上，语句的第一行将创建表)：

```
CREATE TABLE Customers
```

Customers 表是数据库中的第一个表，把数据库的表放在一起来跟踪公司的账目。本书将用几章的内容来介绍如何设计数据库，但是现在通过构建几个表来学习 CREATE TABLE 语句。本节将讲述构建表的大部分概念，但是还会保留一些到本书后面部分再讲。下面先开始构建几个表。

www.TopSage.com

在 CREATE 代码前面添加 USE <database name>行。这样能确保当运行脚本时，是在所指定的数据库中创建表。然后在已经看到的第一行后面跟上几列。

提示：

为了在特定数据库中正常使用而创建的任何脚本都应该使用带有该数据库名称的 USE 命令。这样确保您的确在所指定的数据库中创建、修改和删除对象。不止一次我都因为自己的愚蠢而成为受害者，当我盲目地打开脚本执行之后才发现当前的数据库不对，这样有相同名称的任何表都被删除了(从而丢失了所有数据)，而且更换了新的布局。还可以通过看 master 数据库分辨其他人何时进行了该操作——您经常会在这个数据库中发现几个外来的表，那些表是由于其他人运行了本来应该在其他数据库中运行的 CREATE 脚本生成的。

```
USE Accounting
CREATE TABLE Customers
(
   CustomerNo      int             IDENTITY NOT NULL,
   CustomerName    varchar(30)              NOT NULL,
   Address1        varchar(30)              NOT NULL,
   Address2        varchar(30)              NOT NULL,
   City            varchar(20)              NOT NULL,
   State           char(2)                  NOT NULL,
   Zip             varchar(10)              NOT NULL,
   Contact         varchar(25)              NOT NULL,
   Phone           char(15)                 NOT NULL,
   FedIDNo         varchar(9)               NOT NULL,
   DateInSystem    smalldatetime            NOT NULL
)
```

这个表和在现实生活中可能用的表相比有些简化，但是在以后有很多机会来修改它(也将做这些修改)。

一旦构建了表，就希望验证一下这个表的确已经创建了，而且具有所要求的所有列和类型。可以利用几个命令来做这些验证，但是最好的一个就是我们所熟悉的的 sp_help。语法是很简单的：

```
EXEC sp_help <object name>
```

要指定刚才创建的表对象，试着执行下面的代码：

```
EXEC sp_help Customers
```

EXEC 命令有两种使用方法。在这个例子中用于执行一个存储过程——一个系统存储过程。在后面讲到高级查询话题和存储过程时将看到第二种用法。

注意：

从技术上说，可以通过简单地调用来执行存储过程(不需要使用 EXEC 关键字)。问题是这只有在被调用的存储过程是批处理中的第一条语句时才能工作。在上面的代码中，使用语句 sp_help Customers 即可，但是如果试图在这之前运行 SELECT 语句——这将带来麻烦。不使用 EXEC 将导致不可预料的行为，所以应该避免。

www.TopSage.com

试着执行命令，就会发现一个接着一个地得到几个结果集。检索到的信息包括不同的结果集：

- 表名、模式、表类型(系统的或者用户的)、创建日期
- 列名、数据类型、是否可为空、大小、排序规则
- 标识列(如果存在这样的列的话)，包括初始种子和增量值
- ROWGUIDCOL(如果有的话)
- 文件组信息
- 索引名(如果存在的话)、类型和包括的列
- 约束名(如果有的话)、类型和包括的列
- 外键(如果有的话)名称和列
- 任何模式绑定的视图(第 10 章有更多相关内容)的名称，这些视图依赖于表

既然确认表已经创建了，接下来再创建一个表——Employees 表。这次，先看看在表里需要哪些内容，然后看看如何自己编写 CREATE 脚本。

Employees 表是另一个比较简单的表。包含下列信息：

- 雇员 ID——这个需要系统自动生成
- 名
- 中间名(可选)
- 姓
- 职位
- 社会保障号
- 薪水
- 以前的薪水
- 上次加薪数量
- 雇用时间
- 终止时间(如果有的话)
- 雇员的经理
- 部门

首先可以试着自己给出布局。

在开始处理这些之前，先不要太担心您的布局没有和我的完全一样。有多少数据库设计者，就有多少种数据库设计——所有一切都从表设计开始。对同一个问题可以有不同的解决方法。要考虑的是您是否具有需要说明的所有概念。先来看一种构建表的方法。

在这里有一个特别的列。EmployeeID 是由系统生成的，因此标识列或者 ROWGUIDCOL 是很好的候选。您可能会有一些理由希望采用这两种方法当中的某一种，但是我们将采用标识列，理由如下。

- 普通人都会去使用这个(难道您希望去记住一个 GUID 吗)
- 其开销更低

现在准备构建脚本：

```
CREATE TABLE Employees
(
```


www.TopSage.com

```
EmployeeID    int        IDENTITY NOT NULL,
```

对于这一列，我们最终选择了 NOT NULL 选项，因为使用了 IDENTITY 列。在 IDENTITY 列中不允许有 NULL 值。注意，根据服务器的设置，很可能仍需要包含 NOT NULL 选项(如果采用默认值，那可能就会出错，这取决于默认值是否允许 NULL)。

下一步，需要添加姓名列。通常允许大概 25 字符的姓名。大部分姓名都比这个短得多，但是还是有可能会遇到特别长的姓名(特别是现在流行带连字符的名称)，因此预留了额外的空间。另外，我使用变长数据类型，有如下两个原因。

- 可回收定义的列长度比实际数据长度大的那部分空间(检索空白区)
- 简化 WHERE 子句中的搜索——固定长度的列用空格填充，当执行这种类型字段的比较时需要作额外的规划

例外的是中间名。因为对此只需要允许一个字符，回收空间不再是问题。而变长数据类型在这种情况下将使用更多的空间，因为 varchar 不仅需要存储数据的空间，还需要一小部分系统开销来跟踪数据的长度。另外，易搜索性也不是个问题，因为如果在字段里有任何值的话，没有多余的空间放置填充的空格。

注意：

对于那些直接用 T-SQL 写的代码，SQL Server 会自动调整填充空格问题——(如果比较的话)放在 char(5)中的“xx”将等同于放在 varchar(5)中的“xx”——然而，这在像 SqlNativeClient 和 ADO.NET 这样的客户端 API 中是不相等的。如果在 ADO.NET 中，放在 char(5)中的“xx”将等同于 xx 后面带 3 个空格——如果和“xx”比较，将被认为是不相等的。而放在 varchar(5)中的“xx”将自动去除尾部的空格，在 ADO.NET 中与“xx”比较则是相等的。

由于雇员的姓名是个重要的条目，所以在名和姓列中不允许任何 NULL 值。中间名并不是这么重要(的确，一些美国人根本就没有中间名，而编辑告诉我，在英国有几个中间名也是很常见的)，因此只允许这个字段为 NULL。

```
FirstName       varchar(25)   NOT NULL,
MiddleInitial   char(1)       NULL,
LastName        varchar(25)   NOT NULL,
```

接下来是雇员职位。如果需要支付他们的薪水的话，必须知道他们是做什么的，所以也把这个作为必须的字段：

```
Title    varchar(25)    NOT NULL,
```

对于雇员薪水来说，我们必须知道他们的社会保障号(或者美国以外类似的身份证号)以便缴税。在这里，使用 varchar，允许最多 11 个字符，因为不同国家的身份证号码位数不同。如果知道应用程序只需要美国的 SSN 号，那么可以使用 char(11)：

```
SSN      varchar(11)    NOT NULL,
```

必须知道给雇员付多少薪水——这个似乎挺简单的——但是接下来的情况有点不一样。当添加以前薪水和上次加薪数时，就碰到了可以使用计算列的情况。新的薪水是以前的薪水与上

www.TopSage.com

次加薪数之和。薪水(Salary)数是经常需要用到的——可能希望在它上面建立索引，以进行限定范围的查询，但是由于各种原因，这里并没有这样做(第 9 章将讲述在计算列上创建索引)，而是使用 LastRaise 作为计算列。

```
Salary            money            NOT NULL,
PriorSalary       money            NOT NULL,
LastRaise AS Salary - PriorSalary,
```

如果雇用他们，那么肯定知道雇用的日期——因此雇用日期也是需要的：

```
HireDate   date    NOT NULL,
```

可注意到，这里选择了使用 date 数据类型而不是标准的 datetime，这是为了节省空间。datetime 数据类型将存储精确到秒部分的日期和时间信息。因为主要对雇用的日期而不是时间感兴趣，所以 date 可以满足需求，而且只占用一半的空间。

注意：

要知道，和较老的 datetime 和 smalldatetime 数据类型不同，date、time 和 datetime2 是 SQL Server 2008 提供的新数据类型。如果仍需要向后兼容以前的版本，则需要采用 datetime 和 smalldatetime 数据类型。

终止日期可能是不确定的(我们喜欢认为一些雇员会一直为我们工作)，因此需要将它设置为可以为空：

```
TerminationDate    date      NULL,
```

我们希望知道经理是谁(肯定有人雇佣了他们)和他们在哪个部门工作：

```
   ManagerEmpID      int                 NOT NULL,
   Department        varchar(25)       NOT NULL
)
```

为了清楚起见，看一下创建该表的整个脚本：

```
USE Accounting

CREATE TABLE Employees
(
   EmployeeID        int IDENTITY    NOT NULL,
   FirstName         varchar(25)     NOT NULL,
   MiddleInitial     char(1)         NULL,
   LastName          varchar(25)     NOT NULL,
   Title             varchar(25)     NOT NULL,
   SSN               varchar(11)     NOT NULL,
   Salary            money           NOT NULL,
   PriorSalary       money           NOT NULL,
   LastRaise AS Salary - PriorSalary,
   HireDate          date            NOT NULL,
   TerminationDate   date            NULL,
   ManagerEmpID      int             NOT NULL,
   Department        varchar(25)     NOT NULL
)
```


www.TopSage.com

我再次推荐对该表执行 sp_help 以确认表是按照所期望的方式创建的。

5.3 ALTER 语句

现在已经有了一个数据库和一些不错的表。但有时要求修改表，而不是重新创建。同样地，有时可能需要修改数据库大小、文件位置或其他特性。这时就需要使用 ALTER 语句。

和 CREATE 语句很相似，ALTER 语句的开头总有一样的：

```
ALTER <object type> <object name>
```

到目前为止是完全枯燥的，不过不会一直这样。将很快看到与该语句相关的一些入门问题。在第 6 章更进一步地使用这个语句时(在处理约束时)，事情将变得很有趣(读起来费劲和费解)。

5.3.1 ALTER DATABASE

先介绍修改数据库。但实际上需要做若干个修改，看看各有哪些不同效果，以便了解语法的变化。

也许使用 ALTER 语句时最大的技巧是记住您已经拥有了哪些东西。因此，再次检查一下：

```
EXEC sp_helpdb Accounting
```

结果与创建的一样，如表 5-3 和表 5-4 所示。

表　5-3

name	db_size	owner	dbid	created	status	compatibility_level
Accounting	15.00MB	sa	9	05 28 2000	Status=ONLINE, Updateability= READ_WRITE, UserAccess= MULTI_USER, Recovery=FULL, Version=598, Collation=SQL_ Latin1_General_ CP1_CI_AS, SQLSortOrder=52, IsAutoCreate- Statistics, IsAutoUpdate- Statistics, IsFullTextEnabled	90

www.TopSage.com

可注意到，这次没有像上次使用这个存储过程那样使用引号。那是因为这个是系统过程，和很多其他系统过程一样，可以接受称为 sysname 的特殊数据类型。只要传递的是系统中有效的对象的名称，对于这种数据类型，引号是可选的。

表 5-4

name	fileid	filename	file-group	size	maxsize	growth	usage
Accounting	1	c:\Program Files\ Microsoft SQL Server\ MSSQL.1\mssql\ data\ AccountingData .mdf	PRIMARY	10 240 KB	51 200 KB	5 120 KB	data only
AccountingLog	2	c:\Program Files\ Microsoft SQL Server\ MSSQL.1\ mssql\data\ AccountingLog .ldf	NULL	5 120 KB	25 600 KB	5 120 KB	log only

假设需要作些小的改动。例如，假设向数据库导入大量数据。目前，数据库大小只有 15MB。因为打开了 Autogrow 选项，直接就可以导入，SQL Server 会自动以每次 5MB 的增长速度扩大数据库。然而要记住，实际上重新分配数据库的大小需要很多工作。如果插入 100MB 大小的数据，那么服务器至少要重新分配 16 次(在 20MB、25MB、30MB……的时候)。如果知道将增加到 100MB 的数据，为什么不一次增加呢？要这么做的话，需要使用 ALTER DATABASE 命令。

通常的语法类似于下面这样：

```
ALTER DATABASE <database name>
   ADD FILE
       ([NAME = <'logical file name'>,]
        FILENAME = <'file name'>
        [, SIZE = <size in KB, MB, GB or TB>]
        [, MAXSIZE = < size in KB, MB, GB or TB >]
        [, FILEGROWTH = <No of KB, MB, GB or TB |percentage>]) [,...n]
             [ TO FILEGROUP filegroup_name]
   [, OFFLINE ]
   |ADD LOG FILE
       ([NAME = <'logical file name'>,]
       FILENAME = <'file name'>
       [, SIZE = < size in KB, MB, GB or TB >]
       [, MAXSIZE = < size in KB, MB, GB or TB >]
```


www.TopSage.com

```
    [, FILEGROWTH = <No KB, MB, GB or TB |percentage>])
  |REMOVE FILE <logical file name> [WITH DELETE]
  |ADD FILEGROUP <filegroup name>
  |REMOVE FILEGROUP <filegroup name>
  |MODIFY FILE <filespec>
  |MODIFY NAME = <new dbname>
  |MODIFY FILEGROUP <filegroup name> {<filegroup property>|NAME =
        <new filegroup name>}
  |SET <optionspec> [,...n ][WITH <termination>]
  |COLLATE <collation name>
```

实际上是您很少会使用所有的选项——有时我想，Microsoft 搞这么多选项只是为了把大家弄糊涂(只是开个玩笑)。

因此，在看过所有这些长篇大论后，现在只关注如何将数据库扩充至 100MB：

```
ALTER DATABASE Accounting
 MODIFY FILE
 (NAME = Accounting,
 SIZE = 100MB)
```

可注意到，与创建数据库时不同，这里没有得到任何空间分配的信息——而是得到很不详细的信息：

```
命令已成功完成。
```

由于信息量太少，所以最好自己来检验一下。

```
EXEC sp_helpdb Accounting
```

结果如表 5-5 和表 5-6 所示。

表　5-5

name	db_size	owner	dbid	created	status	compatibility_level
Accounting	105.00 MB	sa	9	05 28 2008	Status=ONLINE, Updateability= READ_WRITE, UserAccess=MULTI_ USER, Recovery= FULL, Version=598, Collation=SQL_ Latin1_General_ CP1_CI_AS, SQLSortOrder=52, IsAutoCreate- Statistics, IsAutoUpdate- Statistics, IsFullTextEnabled	90

www.TopSage.com

表 5-6

name	fildid	filename	file-group	size	maxsize	growth	usage
Accounting	1	c:\Program Files\ Microsoft SQL Server\ MSSQL10. MSSQLSERVER \MSSQL\ DATA\ AccountingData .mdf	PRIMARY	102 400KB	102 400KB	5 120KB	data only
AccountingLog	2	c:\Program Files\ Microsoft SQL Server\ MSSQL10. MSSQLSERVER \MSSQL\ DATA\ AccountingLog .ldf	NULL	5 120KB	25 600KB	5 120KB	log only

其他更一般的数据库层次的修改与此完全类似。然而修改是无止境的。更复杂的文件组修改等内容超出了本书的范围，但是如果您需要有关的更多信息，我推荐找一本面向管理员的书(这样的书多如牛毛)。

选项和终止规范

SQL Server 有一些选项可以通过 ALTER DATABASE 语句来设置。在这当中有大部分 SET 选项可用的特定数据库的默认值(例如 ANSI_PADDING、ARITHABORT——在使用索引或分区视图时是很方便的)、状态选项(例如，单用户模式或只读模式)和恢复选项。不同 SET 选项的作用在本书各个相关内容中讨论。这个新的 ALTER 功能只是提供另一种方法来修改任何特定数据库的默认值。

SQL Server 还能控制一些对数据库所作修改的实现。很多修改需要具有数据库的独占控制权限——如果系统中已经有其他用户，这就很难处理。SQL Server 允许适当地迫使其他用户退出数据库，这样可以完成数据库的修改。这些动作的强度由低到高，从在踢出其他用户之前等待几秒钟(由用户来确定多久)到立即终止任何选项事务(自动回滚)。相对失控(从客户端看来)的

www.TopSage.com

事务终止不是可以轻易进行的。这样的动作一般在数据库管理员的权限范围之内。因此，更进一步的讨论已经超出了本书的范围。

5.3.2　ALTER TABLE

更为常见的情景是需要修改表的结构，其所包括的内容范围很广，从添加新列到修改数据类型等复杂问题。

首先看一下修改表的基本语法：

```
ALTER TABLE table_name
   {[ALTER COLUMN <column_name>
      { [<schema of new data type>].<new_data_type> [(precision [, scale])] max |
<xml schema collection>
      [COLLATE <collation_name>]
      [NULL|NOT NULL]
      |[{ADD|DROP} ROWGUIDCOL] | PERSISTED}]
   |ADD
      <column name> <data_type>
      [[DEFAULT <constant_expression>]
      |[IDENTITY [(<seed>, <increment>) [NOT FOR REPLICATION]]]]
      [ROWGUIDCOL]
      [COLLATE <collation_name>]
         [NULL|NOT NULL]
      [<column_constraints>]
      |[<column_name> AS <computed_column_expression>]
   |ADD
      [CONSTRAINT <constraint_name>]
      {[{PRIMARY KEY|UNIQUE}
         [CLUSTERED|NONCLUSTERED]
         {(<column_name>[ ,...n ])}
         [WITH FILLFACTOR = <fillfactor>]
         [ON {<filegroup> | DEFAULT}]
         ]
         |FOREIGN KEY
            [(<column_name>[ ,...n])]
            REFERENCES <referenced_table> [(<referenced_column>[ ,...n])]
            [ON DELETE {CASCADE|NO ACTION}]
            [ON UPDATE {CASCADE|NO ACTION}]
            [NOT FOR REPLICATION]
         |DEFAULT <constant_expression>
            [FOR <column_name>]
         |CHECK [NOT FOR REPLICATION]
            (<search_conditions>)
      [,...n][ ,...n]
         |[WITH CHECK|WITH NOCHECK]
    | { ENABLE | DISABLE } TRIGGER
      { ALL | <trigger name> [ ,...n ] }
|DROP
   {[CONSTRAINT] <constraint_name>
     |COLUMN <column_name>}[ ,...n]
   |{CHECK|NOCHECK} CONSTRAINT
```


www.TopSage.com

```
        {ALL|<constraint_name>[ ,...n]}
    |{ENABLE|DISABLE} TRIGGER
        {ALL|<trigger_name>[ ,...n]}
  | SWITCH [ PARTITION <source partition number expression> ]
      TO [ schema_name. ] target_table
      [ PARTITION <target partition number expression> ]
}
```

和 CREATE TABLE 命令一样，有很多问题需要处理。

接下来要用示例来说明，首先查看一下 Accounting 数据库中的 Employees 表：

```
EXEC sp_help Employees
```

为了节省一些篇幅，对表 5-7 所示的结果进行编辑，只给出那些关注的数据——实际看到的结果要更多。

表 5-7

列 名 称	类 型	是否是计算列	长 度	精 度	小数位数	是否可为空
EmployeeID	int	no	4	10	0	no
FirstName	varchar	no	25			no
MiddleInitial	char	no	1			yes
LastName	varchar	no	25			no
Title	varchar	no	25			no
SSN	varchar	no	11			no
Salary	money	no	8	19	4	no
PriorSalary	money	no	8	19	4	no
LastRaise	money	yes	8	19	4	yes
HireDate	date	no	4			no
TerminationDate	date	no	4			yes
ManagerEmpID	int	no	4	10	0	no
Department	varchar	no	25			no

假设决定保留雇员以前的雇主的信息。这需要增加另外一列，实际上不是很复杂。语法看起来很像 CREATE TABLE 语句，不同的是这里有明显的变更：

```
ALTER TABLE Employees
   ADD
      PreviousEmployer varchar(30) NULL
```

如果希望的话，也可以一次添加几列。它将看起来像下面这样：

```
ALTER TABLE Employees
   ADD
      DateOfBirth      datetime  NULL,
      LastRaiseDate    datetime  NOT NULL
         DEFAULT '2008-01-01'
```


www.TopSage.com

提示：

注意这里列出的 DEFAULT。实际上还没有讲到这一内容(将在第 6 章介绍)，但在这里使用它是为了指出一种特例。

如果想在事后添加 NOT NULL 列，那么必须考虑如何处理已经有 NULL 值的行。在这里，解决方案是提供一个默认值。该默认值用于为表中已有的任何行填充新列。

在讨论其他话题之前，先看看添加了哪些数据，如表 5-8 所示。

```
EXEC sp_help Employees
```

表　5-8

列　名　称	类　　型	是否是计算列	长　　度	精　度	小数位数	是否可为空
EmployeeID	int	no	4	10	0	no
FirstName	varchar	no	25			no
MiddleInitial	char	no	1			yes
LastName	varchar	no	25			no
Title	varchar	no	25			no
SSN	varchar	no	11			no
Salary	money	no	8	19	4	no
PriorSalary	money	no	8	19	4	no
LastRaise	money	yes	8	19	4	yes
HireDate	date	no	4			no
TerminationDate	date	no	4			yes
ManagerEmpID	int	no	4	10	0	no
Department	varchar	no	25			no
PreviousEmployer	varchar	no	30			yes
DateOfBirth	datetime	no	8			yes
LastRaiseDate	datetime	no	8			no

可以看到，所有的列都添加了。然而，需要注意的是这些列都在列列表的末尾。在 SQL Server 中，没办法将列添加到特定的位置。如果想将一个列移动到中间，那么需要创建一个全新的表(采用一个不同的名称)，将数据复制到新表中，删除已有的表，然后将新表重命名。

提示：

这个列的移动问题的确非常麻烦。甚至一些支持这个过程自动化的工具经常也有问题。为什么呢？因为任何引用本表的外键约束必须在允许删除当前版本的表之前删除掉。这意味着必须删除所有的外键，执行修改，然后将所有的外键添加回去。然而问题还不止如此，在原来的表上定义的任何索引在删除现有表时会自动删除——这意味着必须记得把重新创建索引作为创

www.TopSage.com

建新表的构建脚本的一部分。

但是还有一个问题。因为还没有考虑视图，所以有必要说明一下在添加一列时视图所发生的改变。应该明白，即使视图是使用 SELECT*作为基本语句来构建的，在重新构建视图前，新列不会在视图中出现的。由于性能方面的原因，视图中的列名称是在视图创建时解析的。这意味着在添加列时已创建的视图已经使用原来的列列表解析了——必须要么先删除视图然后再重新创建视图，要么使用 ALTER VIEW 语句重新构造视图。

5.4 DROP 语句

执行 DROP 语句与删除在 DROP 语句中引用的任何对象一样。这一操作快而简单，其语法对于所有主要的 SQL Server 对象(表、视图、存储过程及触发器等)来说是相同的，如下所示：

```
DROP <object type> <object name> [, ...n]
```

实际上，这和 SQL 语句一样简单。如果需要，可以同时删除两个表：

```
USE Accounting

DROP TABLE Customers, Employees
```

这样就将两个表都删除了。

注意：

使用这个命令时要非常小心。该命令没有像“您确定吗？”这样的提示问题——它只假定您知道您所做的事情，删除要删除的对象。

删除整个数据库的语法与此很类似。下面删除 Accounting 数据库：

```
USE master

DROP DATABASE Accounting
```

可在“消息”窗格中看到如下内容：

```
命令已成功完成。
```

可能会碰到这样的情况：返回一个错误，指出因为数据库正被使用，所以不能删除。如果是这样的话，要检查下面两个问题。

- 确保在 Management Studio 中的当前数据库不是要删除的数据库(即在想删除数据库时，确保没有正在使用该数据库)。
- 确保没有(使用 Management Studio 或者 sp_who)打开其他显示要删除的数据库为当前数据库的连接。

通常如代码示例中的方法解决第一个问题——切换到 master 数据库。第二个问题需要手动检查——我通常是采用完全关闭其他会话的方法。

www.TopSage.com

5.5　使用 GUI 工具

在前面花了大量时间来讲述创建数据库和一些表的完整语法——就目前而言，这些足够了。下面来看一下 Management Studio 中用于构建和关联表的图形工具。从现在开始，将不仅仅涉及代码，还要学习使用能生成大量代码的图形工具。

5.5.1　使用 Management Studio 创建数据库

如果运行 SQL Server Management Studio，然后展开“数据库”节点，则可以看到如图 5-1 所示的内容。

图　5-1

提示：

如果仔细看一下该图，会发现仍显示了 Accounting 数据库，尽管在前面一个示例中已删除了它。这个数据库是否可见，取决于 Management Studio 是在通过“查询”窗口删除数据库时已经打开还是在删除数据库之后打开的。

为什么会有这样的区别呢？在 SQL Server 的早期版本中，相当于现在的 Management Studio 的工具会定期更新可用数据库这类信息。而现在只是在有必要时才做更新(例如使用 Management Studio Object Explorer 而不是“查询”窗口删除了某个内容，或者明确选择刷新)。这一改变是为了提高性能。原来的 Enterprise Manager 6.5 的性能较差，因为它总是不断地来回“轮询”服务器。而新的方法在性能上有很大改进，但不一定保证是最新的信息。

总的来说，如果在 Management Studio 窗口中所看到的内容不是所预期的，可试着按下 F5 键(刷新)，这样就会进行更新。

现在右击“数据库”节点，然后选择“新建数据库”选项。

这样会打开“新建数据库”对话框，可以填入希望数据库如何创建的信息。使用和在本章开始创建 Accounting 数据库时相同的选择。首先是基本的名称和大小信息，如图 5-2 所示。

先是名称，这是非常基本的信息。在前面，称之为 Accounting，而且因为已经删除了原来创建的，所以还是这样命名。

然后是文件名、大小和增长信息。

提示：

作者已经手动展开了对话框，这样可以看到所有内容。您所能看到的可能会比图中显示的要少，因为默认大小的对话框不足以显示所有信息——可以拖住对话框的一角，展开到能看其余信息。

接下来是“选项”选项卡，其中包含了额外设置，如图 5-3 所示。

www.TopSage.com

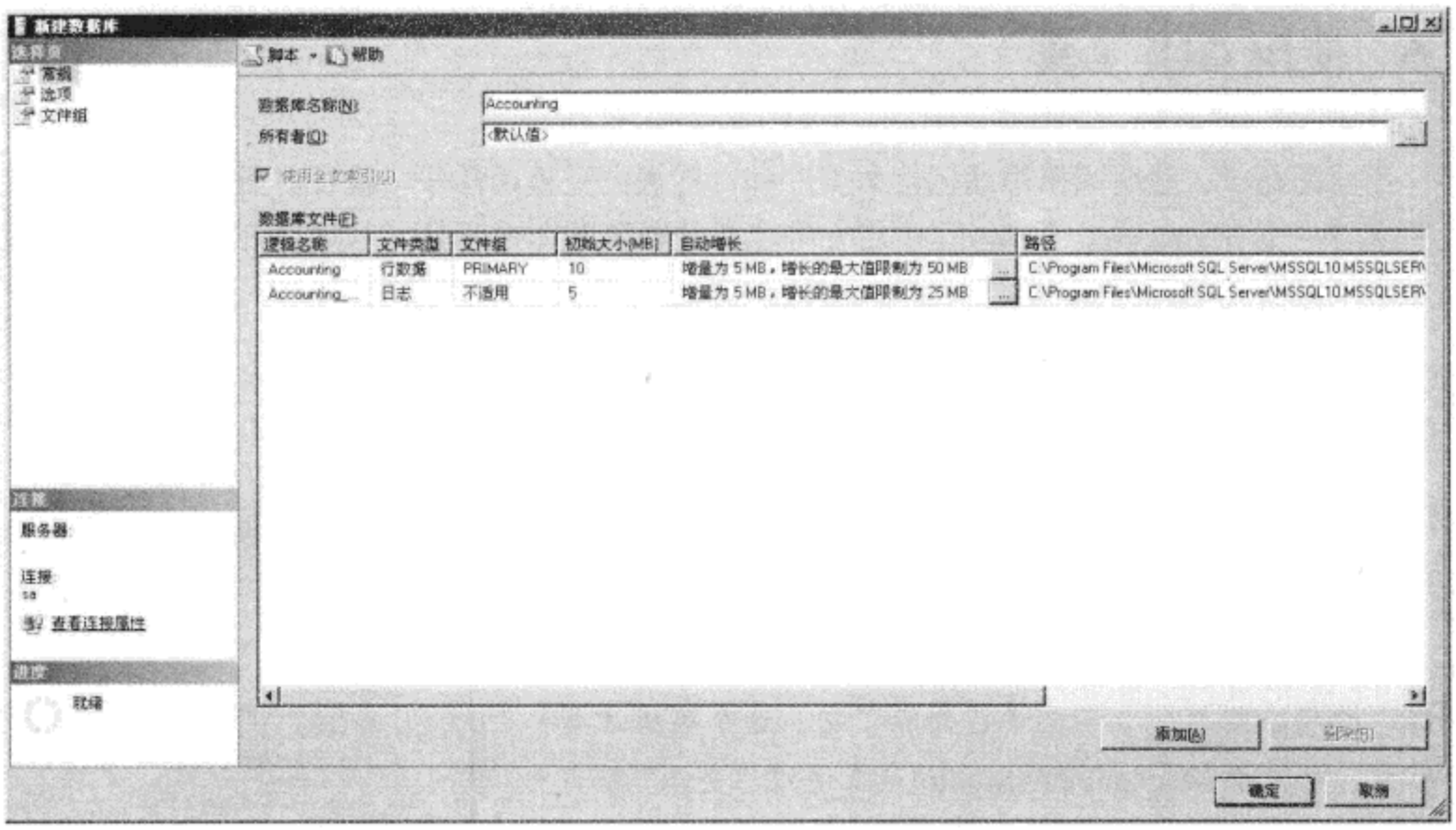


图 5-2

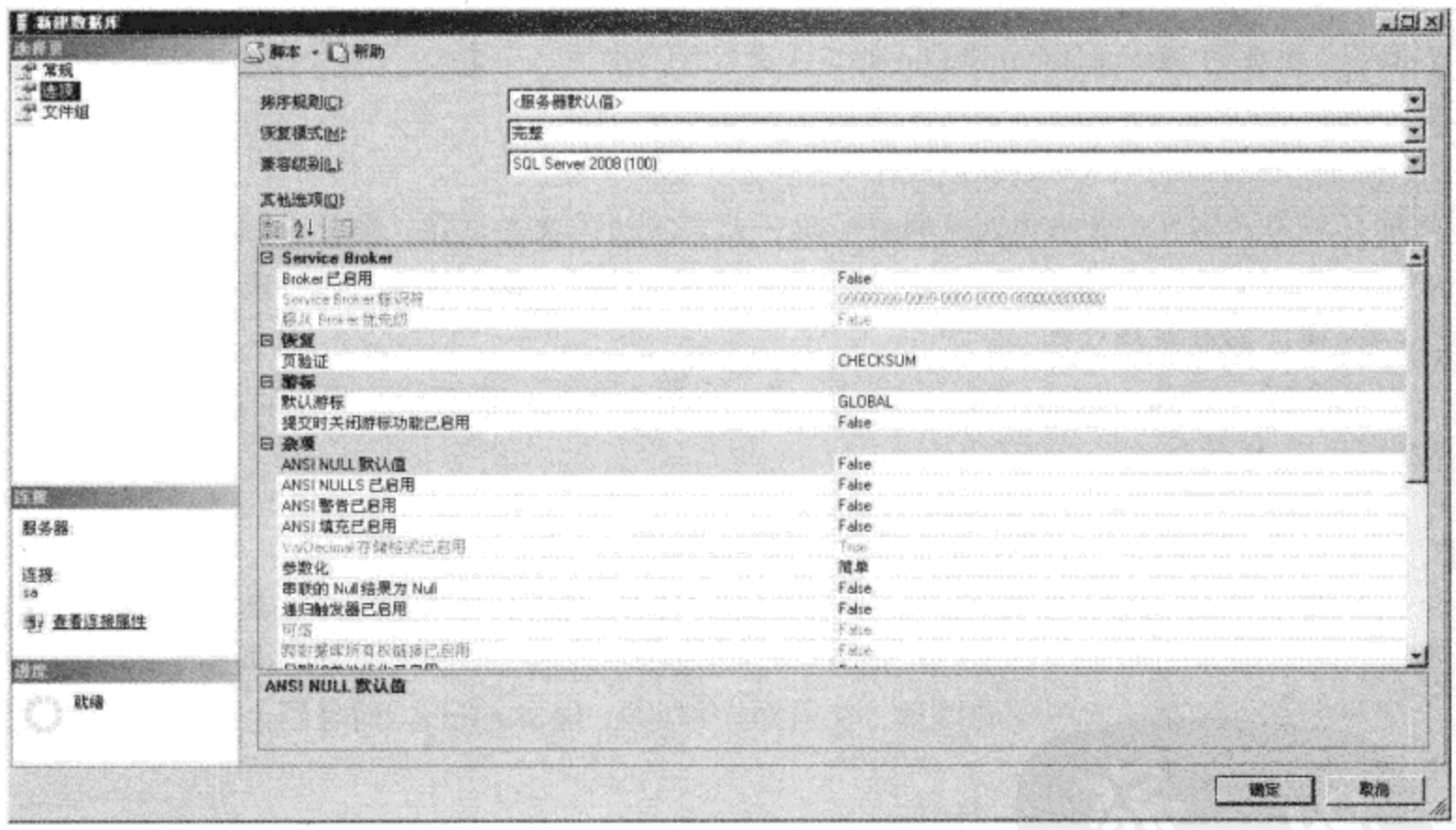


图 5-3

这里最有趣的要数排序规则名称。可以让每个数据库(甚至是每个列)有自己的排序规则。对于大部分的安装来说，还是要遵循服务器安装时的默认设置(也许，有人已经仔细考虑过这个问题了)。然而，可以为当前数据库修改这些默认设置。

提示：

您可能会问："为什么需要不同的排序规则呢？"因为像在讲英语的国家中，对于特定排序规则通常的要求是：一些应用程序不区分"a"和"A"，而其他一些应用程序区分大小写

www.TopSage.com

(即“a”和“A”是不一样的)。在以前，必须采用单独的服务器来处理这个问题。另外，在不讲英语的许多国家中，有着不同的方言——即使他们说同样的通用语言。

接下来是兼容级别。这将控制是否支持一些特定的 SQL Server 2008 语法和关键字。顾名思义，该设置的目的是：如果特定的应用程序需要的话，允许回滚到与旧版本更匹配的关键字和功能行为。例如，如果回退至之前的版本，新版本中作为关键字的词恢复为非关键字对待，在新版本中更改了默认值的特定行为将恢复为以前的默认值。

其余属性随安装不同而不同，但是按照本章前面讲述的那样工作。

假设其他设置与在本章前面看到的完全一样，继续操作。单击“确定“按钮，在短暂等待后(这实际上是在创建数据库)，将看到数据库添加到节点树中了。

现在展开节点树，看看在 Accounting 节点下面的各个项，然后选择“数据库关系图”节点。右击它，将出现一个对话框，提示数据库缺少一些支持数据库关系图的对象，如图 5-4 所示，单击“是”按钮。

提示：

您只会在第一次为数据库创建数据库关系图时看到它。SQL Server 在特殊表中跟踪数据库关系图，它只在实际要创建一个将使用这些表的数据库关系图时才在数据库中进行创建。

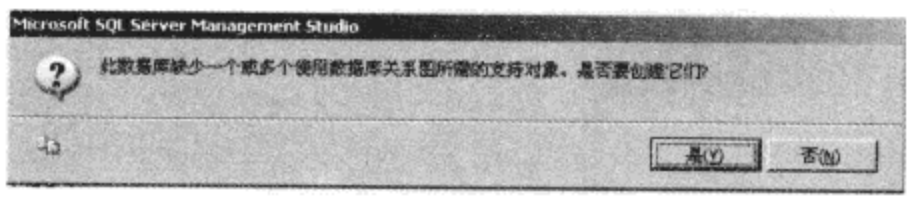


图　5-4

之后，会看到一个“添加表”对话框，如图 5-5 所示。这用来决定在数据库关系图中需要包含哪些表——只要愿意，可以创建多个数据库关系图，每个数据库关系图都会包括整个数据库模式的某个子模型(submodel)。在这里，只有一个空列表，因为只创建了数据库。

现在单击“关闭”按钮(如果没表供添加就不能添加表)，会看到一个空的数据库关系图屏幕。这时，可以右击并选择合适的选项，或是单击工具栏的“新建表”图标来添加表。选择“新建表”命令后，SQL Server 会要求设定新表的名称。然后会出现一个较有用的对话框，可以一行行地填充表——完成需要填写的标签，如图 5-6 所示。

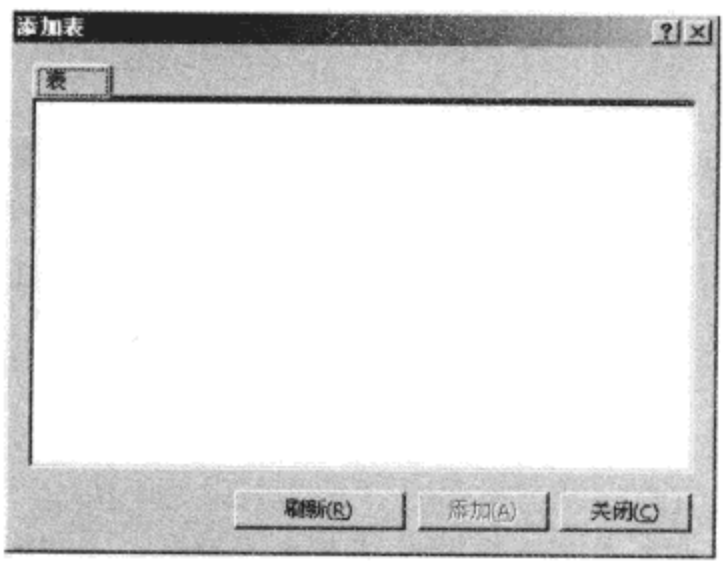


图　5-5

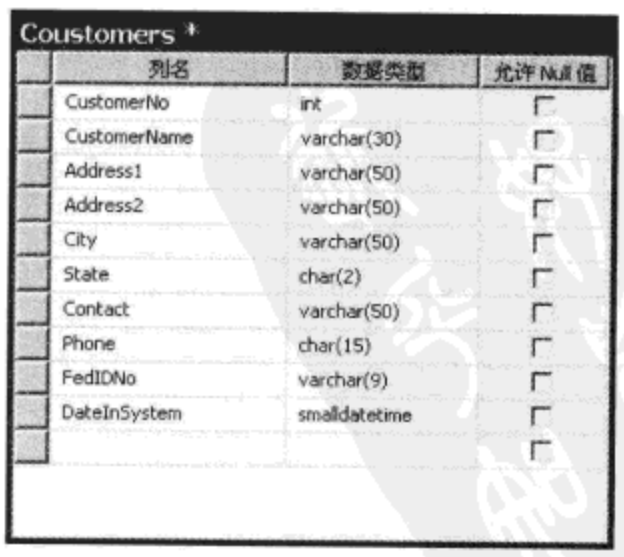


图　5-6

www.TopSage.com

像原来的 Customers 表中的设置那样来填写列，但是还需要定义第一列作为标识列。不过，这里没有办法使用默认网格来定义标识列。要修改为表定义的条目，需要在编辑对话框中右击，选择“表视图”|“修改自定义”。

然后得到一个可从中进行选择的条目列表，如图 5-7 所示。现在，只选择那些需要的条目——Identity 及相关的元素种子和增量。

现在回到编辑对话框，选择“表视图”|“自定义”来查看标识列(如图 5-8 所示)，然后准备填充表定义。

提示：

SQL Server 在这一点上可能会有些不稳定。如果没有选择复选框使之为默认值，那么 SQL Server 将改变“定制”视图的外观，但是不会使“定制”视图成为活动视图——结果是在退出对话框时看不到所作的改变。所以，要在修改了视图后，再次右击并选择“表视图”|“修改自定义”，如图 5-8 所示。

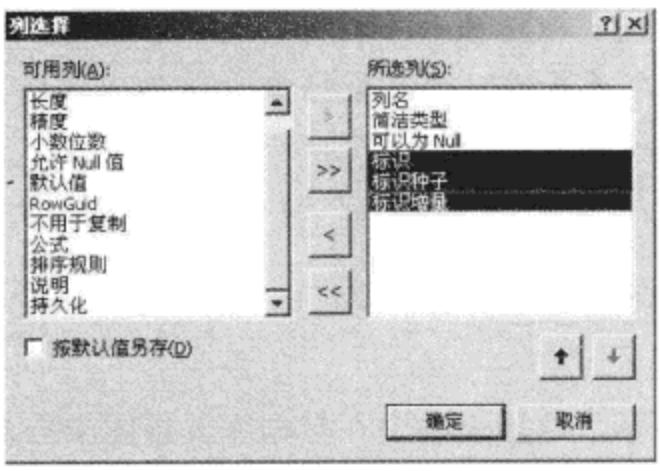


图 5-7

图 5-8

一旦填充了表，可以保存修改，这样将创建表。

提示：

这纯粹是个人喜好问题，但是我喜欢将视图设置在列名下面。可以单击工具栏中的“显示”图标来设置，或者像我喜欢的那样右击表并选择“表视图”|“修改列名”。我发现这样能省出很多屏幕空间给其他表。

现在，按本章前面定义的那样来添加 Employees 表。步骤和创建 Customers 表的步骤基本相同，只有一个地方不同——有一个计算列。要处理计算列，再次选择“修改自定义”命令(从右击菜单中选择)，添加“公式”列。然后，只要添加合适的公式(在这里是 Salary-PriorSalary)。当输入了所有的列后，保存新的表(接受确认对话框)，结果如图 5-9 所示，这个数据库关系图里面应该有两个表。

提示：

需要了解的是，SQL Server 里面包含的数据库关系图工具并非对每个人都是面面俱到的。当您阅读本书的这个部分时，也意味着您刚刚开始您的数据库之旅——这个工具暂时对您

www.TopSage.com

来说是足够的。最后，您可能还希望使用更高级的(代价更昂贵的)工具来帮助进行数据库设计。

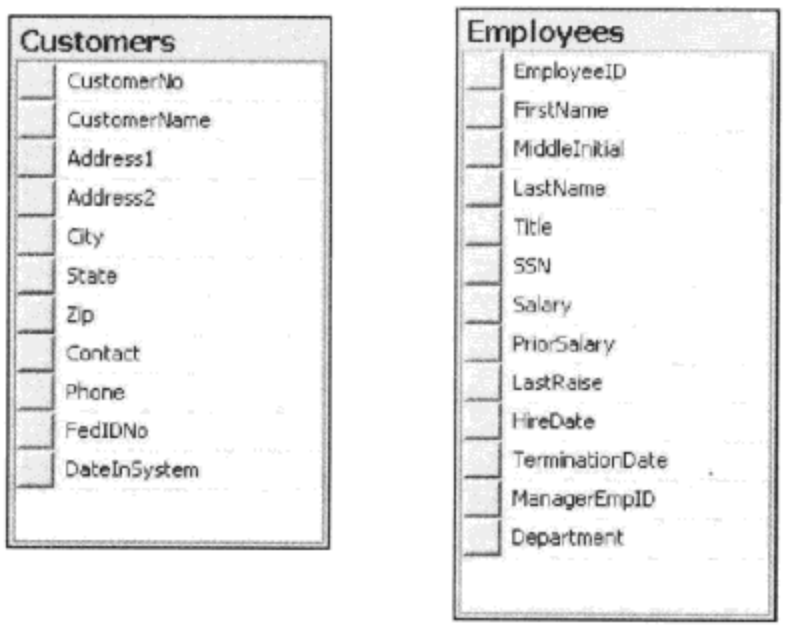


图　5-9

5.5.2　回到代码：使用 Management Studio 创建脚本的基础知识

本章最后简要介绍一个内容——希望让读者了解用 Management Studio 编写脚本的基础知识。不过，只一个简要的入门介绍。

要生成脚本，可在 Management Studio 窗口中右击希望为其生成脚本的数据库(在这里，将在 Accounting 数据库上生成脚本)。在弹出菜单中，选择“编写数据库脚本为”|“CREATE 到”|“新查询编辑器窗口”命令，如果 5-10 所示。

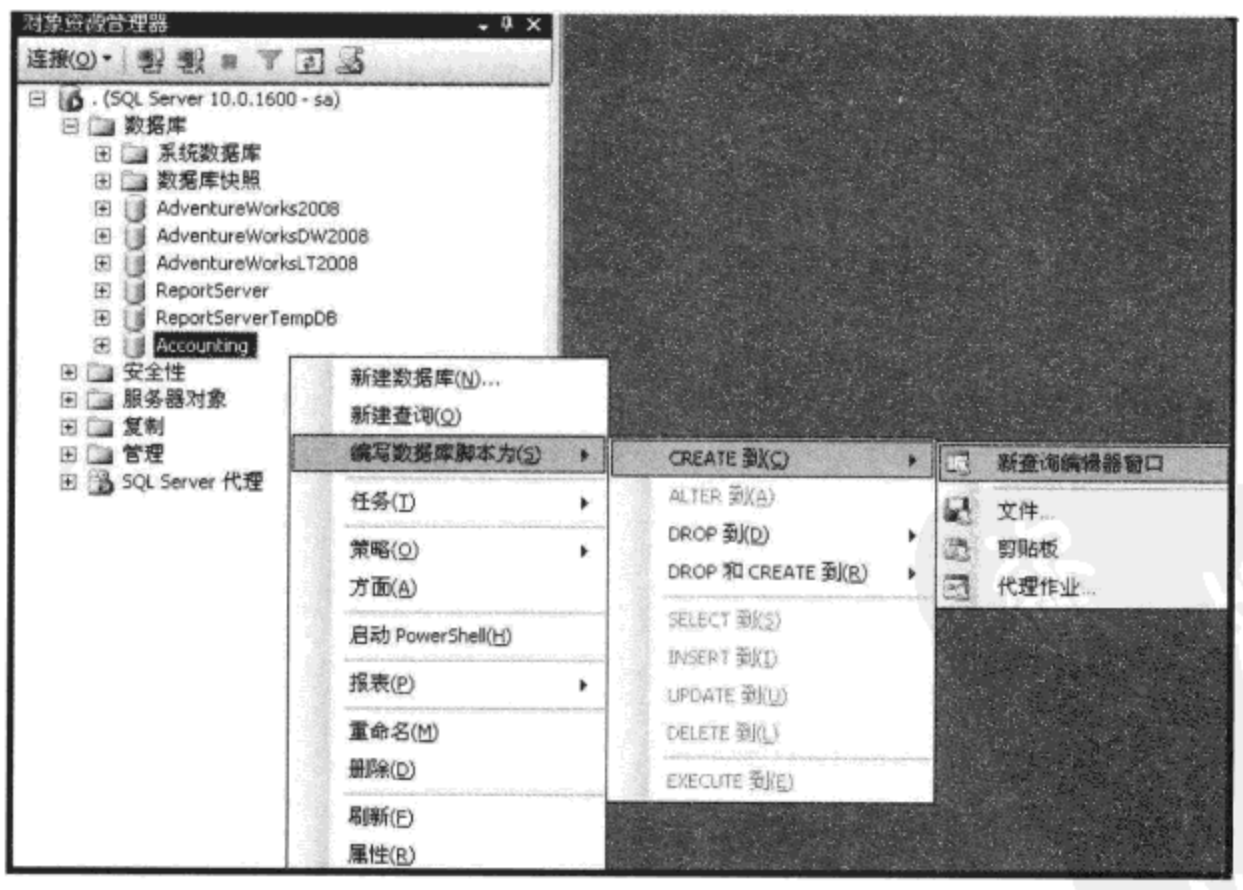


图　5-10

SQL Server 生成了比开始创建数据库时多得多的代码。然而，不用惊慌，这里所做的只是

www.TopSage.com

显式编写主要数据库设置的脚本，而不是像我们自己编写脚本时那样依赖默认值。

注意，并不限于为数据库编写脚本——如果希望为数据库中的其他对象编写脚本，可以像右击 Accounting 数据库一样，定位和右击相应对象进行编写。现在，您得到了一个 SQL 脚本。

可以看到，编写脚本很容易。当将一个复杂的数据库内容放在一起时，它可能不会像这个特殊示例那样简单，但是比全部手动编写简单得多。一旦学会了编写脚本的选项，就很简单了，本书后面将介绍更多的内容。

5.6 本章小结

本章涵盖了有关创建数据库和数据表的 CREATE、ALTER、DROP 语句的基础知识。当然，在本书中还会介绍更多相关内容。本章还介绍了用于在数据库和表中完全控制数据的各种不同的选项。最后，学习了在 Management Studio 中可以使用的很多功能，这简化了我们的工作，使得设计和编写脚本更简单。

后面准备学习如何布局表的一些核心内容，讨论规范化的概念和更一般的数据库设计。然而，在这之前，还要用一章来介绍约束和键，然后再涉及设计问题。

5.7 练习

1. 使用 Management Studio 的脚本生成器，为 Customers 表和 Employees 表生成 SQL 脚本。
2. 不使用 Management Studio，编写脚本创建名为 MyDB 的数据库，初始数据库大小为 17MB，初始日志大小为 5MB，将数据库和日志设置为以 5MB 的增量增长。
3. 创建一个名为 Foo 的表，该表只有一个变长字符类型的字段，字段名为 Col1，限制 Col1 的大小为 50 个字符。

www.TopSage.com

第 6 章

约　　束

本书前面已经几次提到过约束的概念，现在将作详细讲述。考虑到读者可能直接跳读到本章，所以我们简要回顾一下前面的内容。

注意：
约束是一种限制。通过在列级或表级设置约束，确保数据符合某种数据完整性规则。

回顾本书第 1 章和第 2 章讲过的一些概念，确保数据完整性不是使用数据库的程序的责任，而是数据库本身的责任。一般是从不同的源向数据库中插入、更新和删除数据。即使在单独的应用程序中(只有一个程序访问数据库的情况)，都可能出现程序的不同地方访问同一个表的情况。当然远不止如此。数据库管理员(也可能是具有双重角色的用户)可能偶尔修改数据来处理发生的问题。在更复杂的情景下，可能碰到这样的情况：改变一条数据就几乎存在几百条不同的访问路径，更不用说整个数据库。

将维护数据完整性的责任移到数据库本身是数据库管理的一次革命。当试图将数据插入数据库中时还可能产生很多错误，但数据库现在是主动而不是被动地应对问题。另外，关于允许什么数据进入数据库的很多问题可以在开发过程中更早地被发现，因为尽管客户机程序允许数据通过，但数据库知道如何拒绝它。如何做到这点的呢？主要是使用约束(以及数据类型和触发器)。

本章将从较高的层次介绍 3 种不同类型的约束：

- 实体约束
- 域约束
- 参照完整性约束

而在更具体的层次，将介绍实现这几种约束的方法，包括：

- 主键约束
- 外键约束
- 唯一约束(也称为替换键)
- CHECK 约束
- DEFAULT 约束

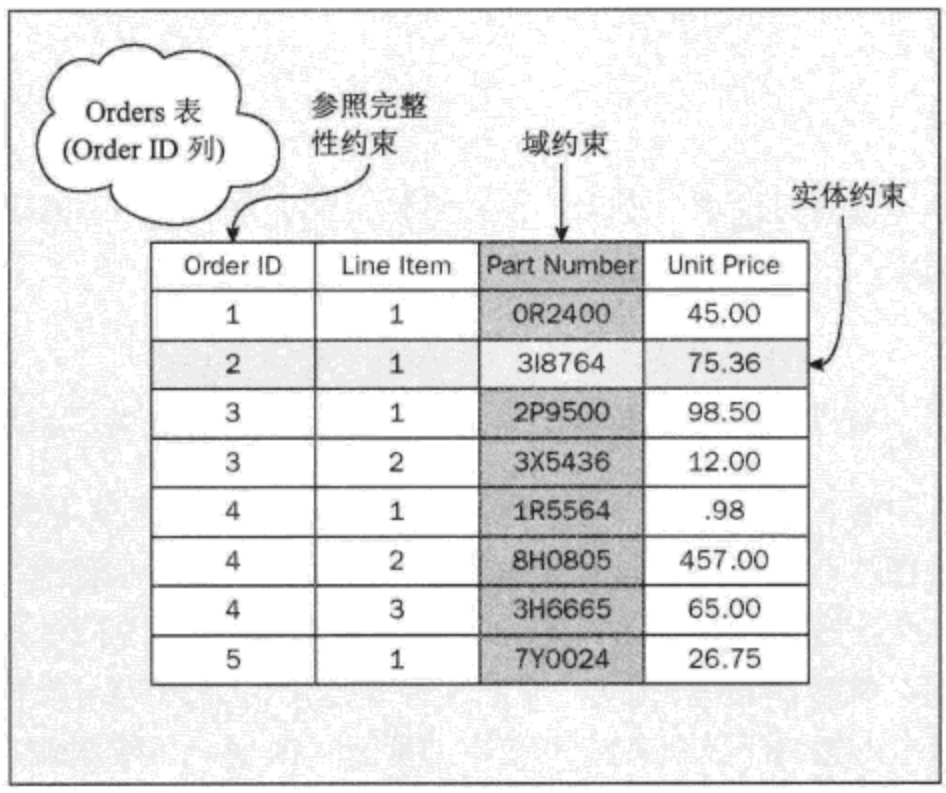
www.TopSage.com

- 规则
- 默认值(和 DEFAULT 约束类似)

本章还将粗略讲一下触发器和存储过程(本书后面将详细讲述)这两种实现数据完整性规则的方法。

6.1 约束的类型

有很多不同的方法来实现约束，但是这些方法可以分为 3 类：实体、域和参照完整性约束，如图 6-1 所示。



Order ID	Line Item	Part Number	Unit Price
1	1	0R2400	45.00
2	1	3I8764	75.36
3	1	2P9500	98.50
3	2	3X5436	12.00
4	1	1R5564	.98
4	2	8H0805	457.00
4	3	3H6665	65.00
5	1	7Y0024	26.75

图 6-1

6.1.1 域约束

域约束处理一个或多个列。这里只讲如何确保一个特定列或一组特定列满足特定的标准。在插入或更新一行时，约束在应用时不考虑表的其他任何行——因为只是对列数据感兴趣。

例如，如果想限定 UnitPrice 列的值大于等于 0，那就是域约束。虽然任何包含 UnitPrice 不符合约束的行会被拒绝，但是实际上会强制完整性确保整个列(不管多少行)满足约束。域是列，这儿的约束是域约束。

在处理 CHECK 约束、规则、默认值和 DEFAULT 约束时会看到这种约束。

6.1.2 实体约束

实体约束都是关于每一行的。这种形式的约束并不关心整体的列，只对特定的行感兴趣。最好的示例是这样的一个约束：要求对于某一列或列的组合，每一行都有一个唯一的值。

您可能会问："唯一列？那不是意味着域约束吗？"不是的。这并不是说每一列必须满足特定的格式，或者那个值必须大于或小于某个值，而是对这一行来说，同样的值不能存在于其

www.TopSage.com

他行中。

本章将在讲述主键约束和唯一约束时学习这类约束。

6.1.3 参照完整性约束

如果某列的值必须与其他列(该列可能在同一个表中，或者更通常的是在不同的表中)的值匹配，那就意味着创建了参照完整性约束。

假设要接受产品订购并接受用信用卡付款。为了接受信用卡公司的付款，就需要与该公司有某种形式的商业协议。而且不想让雇员接受那些不准备支付的公司的信用卡。这样就需要参照完整性约束——它允许构建域或查找表。所谓域表只是提供可接受值的限制列表(通常称为域列表)的表。在这里，可以构建如表 6-1 所示的表。

表 6-1

CreditCardID	CreditCard
1	VISA
2	MasterCard
3	Discover Card
4	American Express

然后可以构建一个或多个引用域表的 CreditCardID 列的表。通过参照完整性，定义为引用 CreditCard 表的任何表(例如 Orders 表)将需要有一列与 CreditCard 表的 CreditCardID 列匹配。对于插入引用表的每行，必须要有一个值存在于域列表中(在 CreditCard 表中必须有相应的行)。

在本章后面学习外键约束时将看到更多内容。

6.2 约束命名

在了解约束的本质之前，先讲述一下约束命名的问题。

注意：

对于本章中要讲述的每种不同类型的约束，可以选择不提供名称——即可以让 SQL Server 提供名称。不过，尽量不要这么做。因为您很快会发现让 SQL Server 创建其自己的名称并不是特别有用。

由系统生成的名称可能是这样的：PK_Employees_145C0A3F。这是 SQL Server 为 Accounting 数据库的 Employees 表的主键生成的名称，本章后面将进行创建。PK 代表主键(primary key，这部分是最有用的)，Employees 说明是在 Employees 表中，剩下的部分为确保唯一性而随机生成的值。只有通过脚本创建主键，才会得到这种类型的命名。如果通过 Management Studio 创建表，那么得到的名称是 PK_Employees。

这个名称还不算糟糕，但是对于其他约束来说，这种名称就没有什么帮助了。例如，本章后面使用的一个 CHECK 约束可能生成这样的名称：CK_Customers_22AA2996。从中可以看出

www.TopSage.com

这是一个 CHECK 约束，但是不知道这个 CHECK 约束具体是指什么。

由于在一个表中可以有多个 CHECK 约束，因此约束的命名可能如下所示：

- CK__Customers__22AA2996
- CK__Customers__25869641
- CK__Customers__267ABA7A

如果需要编辑这些约束当中的一个，那么就很难分辨这些约束。

就个人而言，我采用约束类型与一个说明约束用途的短语相组合的方式，或者采用约束类型与约束影响到的列名相组合的方式。例如，如果有一个约束来确保我的用户不能亏本销售产品，那么我可能使用 CKPriceExceedsCost 来命名；而对于要确保某一列电话号码格式正确的约束，我可能使用 CKCustomerPhoneNo 来命名。

和本书中要使用的其他命名一样，名称本身并不是特别重要。重要的是要做到以下几点：

- 一致性
- 通俗易懂
- 在满足上述条件的情况下简化名称

6.3 键约束

在您学习数据库时，可能听说过 4 种不同类型的常用键，它们是主键、外键、替换键和倒置键。本章只讲述前面 3 种键，因为这些键提供对数据库的约束。

提示：

倒置键基本上是一种索引(本书第 9 章将讲述索引)，该索引对表不应用某种形式的约束(主键、外键、唯一)。倒置键并没有强制数据完整性，只是一种排列数据的可选方法。

键是数据库设计和管理的基础概念之一，因此要打起精神来学习它。这将是本书中讲述的最重要的概念之一，而且在第 8 章讲述规范化时，这个概念将是非常关键的。

6.3.1 主键约束

在定义什么是主键之前，先简单讨论一下关系数据库。关系数据库是在“关联”数据的概念上构建的。因此，对于关系数据库的大部分表(有例外，但是非常罕见)，对于每一行采用唯一标识符变得非常重要。唯一标识符允许准确地引用数据库中其他表中的记录——这样在这两个表之间形成一种关系。

提示：

这是与 20 世纪 80 年代和 90 年代早期的大型机环境或者 ISAM 数据库(dBase、FoxPro 等)中完全不同的概念。在那些早期环境下，每次处理一条记录——通常打开整个表，每次检索一条记录直到找到要查找的为止。如果需要另一个表的数据，则要单独地打开那个表，提取表中的数据，然后自己通过编程的方式来组合数据。

主键是每行的唯一标识符，必须包含唯一的值(因此不能为 NULL)。由于主键在关系数据

www.TopSage.com

库中的重要性，因此它是所有键和约束中最重要的。

注意：

不要混淆主键和 GUID 的概念。主键唯一标识表中的每一行，而 GUID 是更一般的工具，常用于在空间和时间上标识某种事物(不仅仅是行)。虽然 GUID 可以用作主键，但是它会带来一些系统开销，在只处理表内容时通常是不需要的。实际上，只有在处理重复的和其他分布式数据的数据库环境下将 GUID 作为主键才是非常有用的。

一个表最多可以有一个主键。如前面所提到的，很少有不需要主键的表。

提示：

本书说“很少”时，意思是非常少见。没有主键的表严重违反了关系数据的概念——那样意味着不能保证可以关联一个特定的记录。表中的数据不再提供任何的区别特性。

具有多个行逻辑上相同的情况并不罕见，但是这不意味着不需要主键。在这些实例中，您需要虚构某种键——这种方法通常使用标识列来实现，不过在某些情况下使用 GUID 更合理。

主键确保声明作为主键的那些列的唯一性，这个唯一值用作表中每行的标识符。如何创建主键呢？实际上有两种方法。可以用 CREATE TABLE 命令或者 ALTER TABLE 命令创建主键。

提示：

本章后面的大部分示例都使用第 5 章创建的 Accounting 数据库。假定自从在第 5 章最后使用 Management Studio 在数据库中创建表后，未对该数据库作任何改动。

1. 在创建表的时候创建主键约束

回顾一下第 5 章讲述的一个 CREATE TABLE 语句：

```
CREATE TABLE Customers
(
   CustomerNo        int IDENTITY       NOT NULL,
   CustomerName      varchar(30)        NOT NULL,
   Address1          varchar(30)        NOT NULL,
   Address2          varchar(30)        NOT NULL,
   City              varchar(20)        NOT NULL,
   State             char(2)            NOT NULL,
   Zip               varchar(10)        NOT NULL,
   Contact           varchar(25)        NOT NULL,
   Phone             char(15)           NOT NULL,
   FedIDNo           varchar(9)         NOT NULL,
   DateInSystem      smalldatetime      NOT NULL
)
```

这个 CREATE 语句现在看起来比较古老了，但是缺少了一个很重要的方面——主键约束。最好使用 CustomerNo 作为主键。为什么呢？第 7 章将介绍如何选择一个合适的主键，但是现在只要简单考虑一下——希望有两个客户有相同的 CustomerNo 吗？当然不希望——所以使用 CustomerNo 作为客户的标识符是非常合理的。的确，因为这样的系统已经使用了好几年了，所以这里也没有必要搞什么新花样了。

www.TopSage.com

要修改 CREATE TABLE 语句来包含主键约束，只需在希望作为主键的列后面直接添加约束信息。在这里，作如下修改：

```
CREATE TABLE Customers
(
   CustomerNo        int IDENTITY    NOT NULL
      PRIMARY KEY,
   CustomerName      varchar(30)     NOT NULL,
   Address1          varchar(30)     NOT NULL,
   Address2          varchar(30)     NOT NULL,
   City              varchar(20)     NOT NULL,
   State             char(2)         NOT  NULL,
   Zip               varchar(10)     NOT NULL,
   Contact           varchar(25)     NOT NULL,
   Phone             char(15)        NOT NULL,
   FedIDNo           varchar(9)      NOT NULL,
   DateInSystem      smalldatetime   NOT NULL
);
```

由于 Accounting 数据库中已有该表(在第 5 章中创建的)，所以在运行这个 CREATE TABLE 脚本前，需要首先使用 DROP TABLE Customers 命令来删除已存在的表。在这里，修改了一行(所做的就是删除了逗号)，然后在那一列后面添加了一行代码。总而言之，很容易！另外，这里只添加了一个简单的关键字(虽然是两个词，但是作为一个整体起作用)，这样就有了一个主键。

2. 在已存在的表上创建主键约束

现在，假设已经有一个表，那又如何设置主键呢？其实这个也很容易——如给 Employees 表设置主键：

```
USE Accounting

ALTER TABLE Employees
   ADD CONSTRAINT PK_EmployeeID
   PRIMARY KEY (EmployeeID);
```

ALTER 命令告诉 SQL Server 下列信息。

- 添加了一些内容到表中(也可以删除表中的某些内容)
- 添加了什么内容(一个约束)
- 对约束的命名(允许以后直接访问约束)
- 约束的类型(主键约束)
- 约束应用于哪个列

6.3.2 外键约束

外键既能确保数据完整性，也能表现表之间的关系。如果添加外键到一个表，就在您定义外键的表(引用表)和外键引用的表(被引用表)之间创建依赖关系。添加外键之后，插入引用表的记录要么必须与被引用表中被引用列的某条记录匹配，要么外键列的值必须设置为 NULL。这可能不好理解，因此下面用示例来说明。

www.TopSage.com

提示：

当本书表示一个值必须“设置为 NULL”时，指的是 INSERT 语句实际看起来的样子。正如马上就会讲到的，数据进入表以后，根据 FOREIGN KEY 声明中设置的选项的不同，实际上数据看起来会有些不同。

在 Accounting 数据库中创建另外一个名为 Orders 的表。在这个 CREATE 脚本中您会注意到一点：既使用主键又使用外键。在设计过程中可以知道，主键是表的关键部分。添加外键到脚本中的方式几乎和添加主键的方式一模一样，不同的是在引用方面。设置一列或几列外键约束的语法如下所示：

```
<column name> <data type> <nullability>
FOREIGN KEY REFERENCES <table name>(<column name>)
   [ON DELETE {CASCADE|NO ACTION|SET NULL|SET DEFAULT}]
   [ON UPDATE {CASCADE|NO ACTION|SET NULL|SET DEFAULT}]
```

试一试　创建一个带有外键的表

暂时忽略 ON 子句。对于 Orders 表，其脚本如下所示：

```
USE Accounting

CREATE TABLE Orders
(
   OrderID        int    IDENTITY      NOT NULL
      PRIMARY KEY,
   CustomerNo     int                  NOT NULL
      FOREIGN KEY REFERENCES Customers(CustomerNo),
   OrderDate      date                 NOT NULL,
   EmployeeID     int                  NOT NULL
);
```

注意，实际被引用的列必须定义一个主键或者唯一约束(本章后面将讲述唯一约束)。

提示：

还需要指出的是：主键和外键可以存在于同一列中。可以在 AdventureWorks2008 数据库的 SalesOrderDetail 表中看到这样的示例。主键由 SalesOrderID 列和 SalesOrder DetailID 列组成——前者也是外键，还引用 SalesOrderHeader 表。将在本章后面创建一个表，其中有一个同时是主键和外键的列。

示例说明

一旦成功运行了上面的代码，运行 sp_help，会在 sp_help 信息的约束部分看到关于新约束的报告。如果希望得到关于约束的更详细的信息，可以运行 sp_helpconstraint，语法很简单：

```
EXEC sp_helpconstraint <table name>
```

在新建的 Orders 表上运行 sp_helpconstraint，可以得到表中全部的约束名、标准以及状态信息。在这里，Orders 表有一个外键约束和一个主键约束。

www.TopSage.com

注意：

当在这个表上运行 sp_helpconstraint 时，“群集的”(clustered)这个词将会在 PRIMARY KEY 报告之后出现——这意味着有一个群集索引。本书将在第 9 章进一步探讨该含义。

由于新的外键在表的物理定义中被引用，因此它现在是表的一部分。如第 1 章所述，数据库负责其自身的完整性——外键对数据施加一个约束，确保数据库完整性。

和主键不同，每个表中的外键数目不限制唯一性。在每个表中，可以有 0~253 个外键。唯一的限制是一个给定的列只能引用一个外键。然而，一个外键可以涉及多列。一个给定的被外键引用的列也可以被很多表引用。

TopSage.com

1. 在已存在的表中添加一个外键

像主键或者任何约束一样，会出现需要在已存在表中添加外键的情况。这个过程和创建主键的过程很相似。

试一试 在已存在的表中添加一个外键

在 Orders 表中添加另外一个外键来限制 EmployeeID 字段(该字段保存输入订单的雇员的 ID)为 Employees 表中定义的有效雇员。为此，需要在被引用表中唯一识别目标记录。如前所述，可以通过引用一个主键或者引用一个具有唯一约束的列来实现。在这里，使用本章前面在 Employees 表中已存在的主键。

```
ALTER TABLE Orders
   ADD CONSTRAINT FK_EmployeeCreatesOrder
   FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID)
```

现在对 Orders 表再次执行 sp_helpconstraint，就会看到已添加了新的约束。

示例说明

刚添加的约束和之前添加的约束一样起作用——物理表定义能识别对其包含的数据设置的规则。就像不允许将字符串数据添加到数值列一样，如果某行中引用的负责订单的雇员不具备有效的 EmployeeID，就不允许把该行插入 Orders 表中。如果有人试图添加和雇员记录不匹配的行，那么为了保持数据库完整性，将拒绝插入 Orders 表中。

提示：

注意，虽然添加了两个外键，但是在 sp_helpconstraint 结果的最后(如果选择了“以网格显示结果”，则在在“消息”选项卡下)会显示这样一行内容——“没有外键引用该表”，这告诉我们，虽然在这个表中有外键引用其他的表，但是还没有其他表引用这个表。如果想看看区别在哪里，可以在 Customers 表或者 Employees 表上运行 sp_helpconstraint，将会看到这两个表都被新表 Orders 引用。

2. 使一个表自引用

如果希望引用的列不在另外一个表中，而是在希望构建引用的表中，该怎么做？一个表可

www.TopSage.com

以同时是引用表和被引用表吗？虽然这种情况不是很常见，但实际上是可以的。

提示：

在实际创建这种引用基于标识列的非空字段的自引用约束之前，很关键的一点是在添加外键之前表中至少有一行。为什么呢？问题在于，实际上是在检查并强制外键操作之后才选择并填充标识值的。那样意味着当检查发生时，还没有值供第一行引用。在这里，另外一种办法就是照常创建外键，但是在添加第一行的时候禁用它。本章稍后将学习如何禁用约束。

因为这个表需要引用基于标识列的列，所以需要在添加约束之前在表中添加一行：

```
INSERT INTO Employees
   (
   FirstName,
   LastName,
   Title,
   SSN,
   Salary,
   PriorSalary,
   HireDate,
   ManagerEmpID,
   Department
   )
VALUES
   (
   'Billy Bob',
   'Boson',
   'Head Cook & Bottle Washer',
   '123-45-6789',
   100000,
   80000,
   '1990-01-01',
   1,
   'Cooking and Bottling'
   );
```

有了初始行后，就可以添加外键了。在 ALTER 语句中，这和任何其他外键定义一样。现在可以试试下列代码：

```
ALTER TABLE Employees
   ADD CONSTRAINT FK_EmployeeHasManager
   FOREIGN KEY (ManagerEmpID) REFERENCES Employees(EmployeeID)
```

这与 CREATE 语句有一个区别。可以(但不是必须)省略 FOREIGN KEY，只使用 REFERENCES 子句。此时已经建立了 Employees 表，但是如果要从头开始创建的话，可以使用下面的代码(特别注意在 ManagerEmpID 列上的外键)：

```
CREATE TABLE Employees (
   EmployeeID          int IDENTITY      NOT NULL
      PRIMARY KEY,
   FirstName           varchar (25)      NOT NULL,
   MiddleInitial       char (1)          NULL,
```


www.TopSage.com

```
    LastName            varchar (25)      NOT NULL,
    Title               varchar (25)      NOT NULL,
    SSN                 varchar (11)      NOT NULL,
    Salary              money             NOT NULL,
    PriorSalary         money             NOT NULL,
    LastRaise AS Salary — PriorSalary,
    HireDate            smalldatetime     NOT NULL,
    TerminationDate     smalldatetime     NULL,
    ManagerEmpID        int               NOT NULL
       REFERENCES Employees(EmployeeID),
    Department          varchar (25)      NOT NULL
);
```

注意：

值得注意的是，如果这时想删除 Employees 表(来运行第二个示例)，那么会得到一个错误。为什么呢？当在 Orders 表中建立到 Employees 表的引用时，这两个表就“模式绑定”了——即现在 Employees 表知道在其上有所谓的依赖了。SQL Server 不允许删除一个被其他表引用的表。必须先在 Orders 表中删除外键，然后 SQL Server 才允许删除 Employees 表(或者 Customers 表)。

另外，该约束中的自引用外键不允许涉及初始行，因此只有当外键约束的列允许 NULL 值时，才可以这么做——这样可以让初始行的相应列为 NULL，就不必预先输入初始行了。

3. 级联动作

外键和其他类型键的一个重要区别是：外键是双向的，即不仅是限制子表的值必须存在于父表中，还在每次对父表操作后检查子行(这样避免了孤行)。SQL Server 的默认行为是在子行存在时“限制”父行被删除。然而，有时会自动删除任何依赖的记录，而不是防止删除被引用的记录。同样，在更新记录时，可能希望依赖的记录自动引用刚刚更新的记录。比较少见的情况是，您可能希望将引用行改变为某个已知的状态。为此，可以选择将依赖行的值设置为 NULL 或者那个列的默认值。

这种进行自动删除和更新的过程称为级联。这个过程，特别是删除过程，可以经过几层的依赖关系(一条记录依赖另一条记录，而这另一条记录又依赖其他记录，如此等等)。那么，如何在 SQL Server 中实现级联动作？需要做的就是修改声明外键的语法——只需要添加在前面章节中省略的 ON 子句。

在 Accounting 数据库中添加一个新表来检验这点。创建一个表，存储订单中的各个相关条目信息，称之为 OrderDetails：

```
CREATE TABLE OrderDetails
(
   OrderID          int             NOT NULL,
   PartNo           varchar(10)     NOT NULL,
   Description      varchar(25)     NOT NULL,
   UnitPrice        money           NOT NULL,
   Qty              int             NOT NULL,
   CONSTRAINT       PKOrderDetails
      PRIMARY KEY   (OrderID, PartNo),
   CONSTRAINT     FKOrderContainsDetails
      FOREIGN KEY   (OrderID)
```


www.TopSage.com

```
        REFERENCES Orders(OrderID)
        ON UPDATE  NO ACTION
        ON DELETE  CASCADE
);
```

因为这次将所有的代码一起给出了，所以要逐条地来解释。

注意：

在深入了解外键方面的问题之前——注意一下这里是如何处理主键的。这里不是在键的后面直接声明，而是将它声明为单独的约束项。这样既方便处理多列的主键(这样就不能声明为列约束)，又使得整个 CREATE TABLE 语句整洁。同样的，本应该在列的后面直接声明外键，但这里声明为单独的约束项。本章稍后还将讨论这个问题。

首先，可注意到外键也是主键的一部分——这在子表里面并非少见，而实际上在关联表(第7章详细讲述)中常常出现。要记住，每个约束是独立的——分别地添加、改变或删除每个约束。

下面来看看外键声明：

```
FOREIGN KEY (OrderID)
REFERENCES Orders(OrderID)
```

将 OrderID 声明为依赖于“外部”列。在这里，依赖的是在另外一个表(Orders 表)里的一个列(也称为 OrderID)，但是，在本章前面看到，这个也可以很容易地在同一个表中实现。

注意：

当创建引用本表上定义的外键时，需要注意一些问题。这种外键不允许有声明性的 CASCADE 动作。这种限制是为了防止循环更新或删除——即防止出现第一个更新影响第二个更新，而第二个更新又试图去更新第一个的情况。结果就是无限循环下去。

接下来就是级联问题的关键所在。然而，先来看看 ON 子句：

```
ON UPDATE NO ACTION
ON DELETE CASCADE
```

至此已经定义了两种不同的参照完整性动作。正如您所猜测的，参照完整性动作就是当所定义的参照完整性规则被调用时希望执行的任何动作。在父记录(在 Orders 表中)被更新的情况下，不希望将级联更新到子表(OrderDetalis 表)中。然而，为了便于说明问题，本书选择了 CASCADE 用于删除。

提示：

注意，由于 NO ACTION 是默认值，因此指定代码中这个值是可选的。尽管一般的编码方法中省略了 NO ACTION，但为了使使明确目的，我还是建议显式包含 NO ACTION。

下面试着将一条记录插入到 OrderDetails 表：

```
INSERT INTO OrderDetails
VALUES
   (1, '4X4525', 'This is a part', 25.00, 2);
```


www.TopSage.com

如果未对数据作过修改，那么将会产生一个错误：

```
消息 547，级别 16，状态 0，第 1 行
INSERT 语句与 FOREIGN KEY 约束"FKOrderContainsDetails"冲突。该冲突发生于数据库
"Accounting"，表"dbo.Orders"，column 'OrderID'。
语句已终止。
```

为什么呢？由于还没有向 Orders 表中插入任何数据，因此没法在 Oders 表中还没任何内容的时候来引用 Oders 表中的记录。

提示：

这里将讲述关系数据库工作中的一个问题——依赖链。如果一些数据依赖于另一些数据，而另一些数据又依赖于别的数据，如此等等，那就存在一种依赖链。对此，您实际做不了什么——这是伴随数据库的工作而出现的。必须从依赖链的顶端开始，沿着该链，直到完成需要插入的记录。幸运的是，需要的记录通常已经存在，这样节省了一两层依赖。

因此，为了在 OrderDetails 表中插入一行，在 Orders 表中必须已有一条记录。但是，要在 Orders 表中插入一行，必须先要求在 Customers 表(还记得在 Orders 表中创建的外键吗？)中存在一行。因此，要一步步地进行：

```
INSERT INTO Customers -- Our Customer.
                      -- Remember that CustomerNo is
                      -- an Identity column
VALUES
      ('Billy Bob''s Shoes',
      '123 Main St.',
      ' ',
      'Vancouver',
      'WA',
      '98685',
      'Billy Bob',
      '(360) 555-1234',
      '931234567',
      GETDATE()
      );
```

现在已经有了一个客户记录，可以查看它以进行确认，结果如表 6-2 所示。

表 6-2

Customer No	1
Customer Name	Billy Bob's Shoes
Address 1	123 Main Street
Address 2	
City	Vancouver
State	WA
Zip	98685

www.TopSage.com

(续表)

Contact	Billy Bob
Phone	(360) 555-1234
FedIDNo	931234567
DateInSystem	2000-07-1021:17:00

这样，有了一个值为 1 的 CustomerID(数字可能随试验的次数不同而不同)。将在下面的 INSERT 操作(最终插入到 Orders 中)时使用该数字。为 CustomerID 为 1 的客户插入一个订单：

```
INSERT INTO Orders
   (CustomerNo, OrderDate, EmployeeID)
VALUES
   (1, GETDATE(), 1)
```

这次，一切应该工作正常。

注意：

这里没出现另外一个错误的原因是已经在 Employees 表中插入了初始行；否则需要在该表中插入一行，然后 SQL Server 才允许插入到 Orders 表中(还记得 Orders 表中的外键吗？)。

至此，插入 OrderDetails 表的准备工作已经做好。为了有助于演示后面的 CASCADE 示例，这里实际不是插入一行，而是两行：

```
INSERT INTO OrderDetails
VALUES
   (1, '4X4525', 'This is a part', 25.00, 2)

INSERT INTO OrderDetails
VALUES
   (1, '0R2400', 'This is another part', 50.00, 2);
```

通过运行 SELECT 来验证一下：

```
SELECT OrderID, PartNo FROM OrderDetails
```

返回了所期望的两行：

```
OrderID   PartNo
--------- ---------
1          0R2400
1          4X4525

(2 行受影响)
```

既然已经添加好了数据，来看看 CASCADE 对数据有什么影响。将从 Orders 表中删除一行，然后看看 OrderDetails 表中会发生什么情况：

www.TopSage.com

```
USE Accounting

-- First, let's look at the rows in both tables
SELECT *
FROM Orders

SELECT *
FROM OrderDetails

-- Now, let's delete the Order record
DELETE Orders
WHERE OrderID = 1

-- Finally, look at both sets of data again
-- and see the CASCADE effect
SELECT *
FROM Orders

SELECT *
FROM OrderDetails;
```

这样产生了一些有趣的结果：

```
OrderID    CustomerNo   OrderDate              EmployeeID
---------  ------------ --------------------- --------------
1          1            2000-07-13 22:18:00   1

(1 行受影响)

OrderID    PartNo  Description            UnitPrice     Qty
--------- ------- ---------------------  ------------  ----
1          0R2400  This is another part   50.0000       2
1          4X4525  This is a part         25.0000       2

(2 行受影响)


(1 行受影响)

OrderID    CustomerNo   OrderDate      EmployeeID
--------   -----------  ------------   -----------------------

(0 行受影响)
OrderID    PartNo       Description    UnitPrice     Qty
---------- -----------  -------------- ------------ --------

(0 行受影响)
```

注意，即使只是在 Orders 表中进行了删除操作，这个删除操作还是级联到了 OrderDetails 表中匹配的记录。两个表中的记录都被删除了。如果在表中定义了 CASCADE 更新，并且更新一个相关的记录，那么然后这个操作也会传播到子表中。

提示：

值得注意的是：CASCADE 动作所能影响的深度并未受限制。例如，如果有一个

www.TopSage.com

ShipmentDetails 表通过 CASCADE 动作引用了 OrderDetails 表中的行，那么在对 Orders 表进行删除时，那些 ShipmentDetails 表中的相关记录也会被删除。

这实际上是级联动作的一个危险之处——很容易实现不了 DELETE 或者 UPDATE 语句在数据库中进行的所有操作。因为诸多原因，我不太喜欢使用级联动作——这些动作让人们变得懒惰，而且以那种方式删除数据通常不是个好方法。

1) 其他级联动作

以上是那些级联更新和删除的示例，但是其他两种提到的级联动作是怎样的呢？即 SET NULL 和 SET DEFAULT 是怎样的呢？

因为这些是 SQL Server 2005 中新引入的，所以如果想与 SQL Server 2000 向后兼容的话，应避免使用这两个动作，但是它们的操作是非常简单的：如果执行更新而改变了一个父行的值，那么子行的值将被设置为 NULL，或者设置为该列的默认值(不管选择 SET NULL 或者 SET DEFAULT)，就这么简单。

4. 关于外键的其他方面考虑

在使用外键之前，还需要考虑其他一些问题。本书将多次讲述这一主题，但是现在只讲两个细节问题。

- 如何使外键中的值为必须的或可选的
- 外键实现双向的方式

1) 如何使外键中的值为必须的和可选的

关于如何对定义了外键的列进行填充，外键提供了两个可能的选择。

- 在列中填充与被引用表中的相应列相匹配的值
- 不填充任何值，而使该值为 NULL

通过定义引用列为 NOT NULL，可以使外键完全是必须的(即让用户只能选择前面列表中的第一个)。因为 NULL 值是无效的，而外键需要非 NULL 值与被引用的表匹配，这样可以确保每行在被引用表中都有一个匹配的行。换句话说，引用是必须的。

允许引用列有 NULL 值将可以实现同样的需求，不同的是用户可以选择不提供值——即使在被引用表中没有与 NULL 值匹配的行，还是允许插入。

2) 外键实现双向的方式

在讨论 CASCADE 动作时，曾提到过这一内容，但是当定义外键时，并没有强调这将对两个表都有限制。目前，已讲述了引用表相关的一些内容；然而，一旦定义了外键，被引用表就也应该符合一种规则。

注意：

在默认情况下，如果某记录被依赖表引用，就不能从被引用表中删除该记录或者更新被引用的列。如果希望能够删除或更新这样的记录，那么需要为删除和/或更新设置 CASCADE 动作。

www.TopSage.com

下面介绍如何理解“不能删除或更新一个被引用的记录”这一思想。

本书为 Orders 表定义了一些外键。其中有一个引用了 Employees 表的 EmployeeID 列。例如，假设有一个 EmployeeID 为 10 的雇员，该雇员在一两年内为我们拿到了很多订单，然后决定辞职去干其他工作。很可能删除 Employees 表中该雇员的记录，但是这将带来一个很大的问题——这将在 Orders 表中产生所谓的孤立的记录。Orders 表可能有很多记录仍然具有值为 10 的 EmployeeID。如果允许从 Employees 表中删除值为 10 的 EmployeeID，那么无法知道哪个雇员输入了这些订单的——Orders 表中 EmployeeID 列的值就没有意义了。

现在进一步看看这个示例。现在假设这个雇员没有辞职，而是由于某种原因，需要改变该雇员的 ID 号。如果对 Employees 表进行了这个改变(通过 UPDATE 语句)，但是没有对 Orders 表进行相应的更新，那么还是出现了孤立的记录——在 Orders 表中 EmployeeID 列值为 10 的记录没有匹配的雇员。

下面更进一步讨论这个问题。假设有人又插入了 EmployeeID 值为 10 的新记录——现在在 Orders 表中有很多记录被关联到没有拿到这些订单的雇员上。这就存在错误的数据了。

为了避免出现孤立的记录，SQL Server 在默认情况下限制删除或更新被引用表(在这里是 Employees 表)中的记录，除非任何依赖记录已经从引用表(这里是 Orders 表)中被删除或更新了。

注意：

对于在哪些情况下适合(或不适合)采用 CASCADE 动作作进一步讨论还是有意义的。为了数据完整性，如果在 Orders 表中有依赖行的话，可能就不希望允许删除某个雇员。无法跟踪雇员将降低数据的价值。另一方面，(由于某些奇怪的原因)改变雇员的 ID 是完全有效的。可以级联 Orders 表的更新而没有副作用。另外，不必对 UPDATE 和 DELETE 作同样的 CASCADE 决策，可以将它们分开考虑，而且要仔细考虑。

正如所看到的，虽然在一个表中定义了外键，但它实际上对两个表有限制(如果外键是自引用的，那么两个限制都是针对同一个表的)。

6.3.3 唯一约束

这些相对更容易。唯一约束与主键比较相似，共同点在于它们都要求表中指定的列(或者列组合)上有一个唯一值。您可能会常听到，将唯一约束称为替换键。主要的区别是唯一约束没有被看作表中记录的唯一标识符(即使您可能按这样方式使用也有效)，而且可以有多个唯一约束(而在每个表中只能有一个主键)。

一旦建立了唯一约束，那么指定列中的每个值必须是唯一的。如果更新或者插入一行在带唯一约束的列上有已经存在的值的记录，SQL Server 将抛出错误，拒绝这个记录。

注意：

和主键不同，唯一约束不会自动防止设置一个 NULL 值。是否允许 NULL 取决于表中相应列的 NULL 选项的设置。然而要记住，如果确实允许 NULL，那么只能插入一个 NULL(虽然这些 NULL 值并不相等，但在唯一约束看来，它们是相等的)。

因为这里没有什么新的内容(本章在讲述主键时已经讲得够多了)，下面直接看看代码。下面的代码将在 Accounting 数据库中创建另外一个表——Shippers 表：

www.TopSage.com

```
CREATE TABLE Shippers
(
   ShipperID        int   IDENTITY   NOT NULL
      PRIMARY KEY,
   ShipperName      varchar(30)      NOT NULL,
   Address          varchar(30)      NOT NULL,
   City             varchar(25)      NOT NULL,
   State            char(2) NOT      NULL,
   Zip              varchar(10)      NOT NULL,
   PhoneNo          varchar(14)      NOT NULL
      UNIQUE
);
```

对 Shippers 表运行 sp_helpconstraint，确认 Shippers 表已经创建了合适的约束。

在已存在的表中创建唯一约束

同样，创建方法和前面创建主键与外键很相似。下面在 Employees 表中创建唯一约束：

```
ALTER TABLE Employees
   ADD CONSTRAINT AK_EmployeeSSN
   UNIQUE (SSN);
```

快速运行 sp_helpconstraint 确认已经正确创建约束，并查看约束是针对哪一列。

提示：

这里在约束名称中使用的 AK 代表替换键(Alternate Key)—— 与PK 代表主键、FK 代表外键一样。您还经常会看到前缀 UQ 或者 U，这代表唯一约束名称。

6.4 CHECK 约束

CHECK 约束较好的方面是它不限于一个特定的列。CHECK 约束可以和一个列关联，但是也可以和表关联，因为它们可以检查一个列的值相对于另外一个列的值，只要这些列都在同一个表中以及值是在更新或者插入的同一行中。CHECK 约束还可以用于检查列值组合是否满足一个标准。

可以使用与 WHERE 子句一样的规则来定义 CHECK 约束。CHECK 约束条件的示例如表 6-3 所示。

表 6-3

目 标	SQL
限制 Month 列为合适的数字	BETWEEN 1 AND 12
正确的 SSN 格式	LIKE '[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]'
限制为一个 Shippers 的特定列表	IN ('UPS', 'Fed Ex', 'USPS')
价格必须为正数	UnitPrice >= 0
引用同一行中的另外一列	ShipDate >= OrderDate

www.TopSage.com

上面列出的只是一小部分，而条件实际上是无限多的。几乎所有可放置到 WHERE 子句的条件都可以放到该约束中。而且，和其他选择(规则和触发器)相比，CHECK 约束执行速度更快。

还是在 Accounting 数据库上进行操作，现在对 Customers 表修改，检查 DateInSytem 字段的日期的有效性(在系统中不可能有一个未来的日期)：

```
ALTER TABLE Customers
   ADD CONSTRAINT CN_CustomerDateInSystem
   CHECK
   (DateInSystem <= GETDATE ());
```

现在试着插入一个违反 CHECK 约束的记录，将得到一个错误：

```
INSERT INTO Customers
   (CustomerName, Address1, Address2, City, State, Zip, Contact,
   Phone, FedIDNo, DateInSystem)
VALUES
   ('Customer1', 'Address1', 'Add2', 'MyCity', 'NY', '55555',
   'No Contact', '553-1212', '930984954', '12-31-2049')
```

```
消息 547，级别 16，状态 0，第 1 行
INSERT 语句与 CHECK 约束"CN_CustomerDateInSystem"冲突。该冲突发生于数据库"Accounting"，表
"dbo.Customers"，column 'DateInSystem'。
语句已终止。
```

现在如果改变一下，使用一个满足 CHECK 约束条件的 DateInSystem 数据(今天或者以前的任何日期)，INSERT 工作将正常进行。

6.5 DEFAULT 约束

这是与“默认值”有关的两种不同类型的数据完整性工具中的第一个。但是，这非常容易让人混淆，本节将尽量解释清楚。

提示：

在本章后面讲述规则和默认值时，将看到另一种与“默认值”有关的工具。

和所有约束一样，DEFAULT 约束也是表定义的一个组成部分。它定义了当插入的新行对于定义了默认约束的列未提供相应数据时该怎么办。可以定义它为一个字面值(例如，设置默认薪水为 0，或者设置字符串列为“UNKNOWN”)，或者某个系统值(如 GETDATE())。

对于 DEFAULT 约束，要了解以下几个主要内容。

- 默认值只在 INSERT 语句中使用——在 UPDATE 语句和 DELETE 语句中被忽略。
- 如果在 INSERT 语句中提供了任意值，那么就不使用默认值。
- 如果没有提供值，那么总是使用默认值。

默认值只在 INSERT 语句中使用。这是个很多 SQL Server 初学者经常混淆的问题。可以这样考虑——当第一次插入记录时，SQL Server 没有为列提供任何值，除非您自己提供，或者这个列有默认值。如果不是那两种情况，那么 SQL Server 将插入一个 NULL 值(实际意思是“我

www.TopSage.com

不知道”)，或者如果列定义为 NOT NULL，那么 SQL Server 将拒绝该记录。在第一次插入之后，SQL Server 在该列中已经有了一个值。如果要更新该列，那么就有一个新的值。如果该列不在 UPDATE 语句中，那么 SQL Server 就保留列中原来的值。

如果为列提供了一个值，就没有理由使用默认值了——使用提供的值。

如果没有给定任何值，那么总是使用默认值。现在看来问题似乎很简单，不过当您实际上想插入记录中指定列的是一个 NULL 值时，事情就复杂了。如果您没有为某个定义了默认值的列提供一个值，那么将使用默认值。但如果实际上希望这个值为 NULL，那该怎么办呢？比如可以用 INSERT 语句插入 NULL。

注意：

另外值得注意的是，有关不使用默认值的 UPDATE 命令的规则有一个例外，如果显式说明希望使用默认值就属于例外。可以通过使用关键字 DEFAULT 作为希望列更新的值来进行设置。

6.5.1 在 CREATE TABLE 语句中定义 DEFAULT 约束

同样，定义方式和本书前面讲述的其他列约束类似。只是在列定义后面添加约束。

为了举例说明，先删除本章前面创建的 Shippers 表。然后创建该表的一个简化版本，包括默认值：

```
CREATE TABLE Shippers
(
   ShipperID int IDENTITY NOT NULL
      PRIMARY KEY,
   ShipperName varchar(30) NOT NULL,
   DateInSystem smalldatetime NOT NULL
      DEFAULT GETDATE ()
);
```

运行 CREATE 脚本后，可再次使用 sp_helpconstraint 来进行查看。然后可以插入一个新记录来测试一下默认值是如何工作的：

```
INSERT INTO Shippers
   (ShipperName)
VALUES
   ('United Parcel Service');
```

然后在 Shippers 表上运行 SELECT 语句：

```
SELECT * FROM Shippers
```

因为没有提供任何值，所以为 DateInSystem 列生成默认值：

```
ShipperID        ShipperName            DateInSystem
---------------------------------------- ---------------------------
1                 United Parcel Service  2000-07-13 23:26:00

(1 行受影响)
```


www.TopSage.com

6.5.2 在已存在的表中添加 DEFAULT 约束

虽然这与前面也是很相似的，但是有个小变化。这里使用 ALTER 语句，像以前一样添加约束，但是将添加一个 FOR 运算符，告诉 SQL Server 哪一列是针对于 DEFAULT 约束的：

```
ALTER TABLE Customers
   ADD CONSTRAINT CN_CustomerDefaultDateInSystem
      DEFAULT GETDATE() FOR DateInSystem
```

另外一个示例：

```
ALTER TABLE Customers
   ADD CONSTRAINT CN_CustomerAddress
      DEFAULT 'UNKNOWN' FOR Address1
```

和除主键约束外的其他所有约束一样，在每个表中可以添加多个 DEFAULT 约束。

注意：

可以混合使用所有这些约束——不过要小心不要创建有互斥条件的约束。例如，不要出现一个约束要求 col1>col2，而另外一个约束要求 col2>col1。SQL Server 允许这么做，但只有到运行时才能发现问题。

6.6 禁用约束

有时会希望暂时或永久地消除约束。您可能想当然地认为 SQL Server 肯定会提供删除约束的方法，但是 SQL Server 只允许禁用外键约束或者 CHECK 约束，而同时保持约束的完整性。

取消数据完整性规则的概念初看起来可能比较荒唐。这里的意思是为什么要取消可以确保没有无效数据的东西呢？首先，要认识到取消数据完整性规则只是暂时性的(否则可以完全舍弃它)。禁用一个数据完整性规则通常是因为已经有无效数据了。这样的数据通常分为以下两类。

- 在创建约束时已经在数据库中的数据
- 在约束创建以后希望添加的数据

SQL Server 允许禁用完整性检查一段时间来对例外的无效数据作处理，然后再重新启用完整性——不是物理删除数据完整性约束。

注意：

不能禁用主键约束或者唯一约束。

6.6.1 在创建约束时忽略无效的数据

在同时创建表和约束的情况下，所有这些语法都正常工作。然而，更常见的情况是：数据规则是事后加入的。比如，在设计数据库时忽略了一些问题，现在在 Invoicing 表中有一些记录显示有一些负的发票金额。您可能希望添加一个规则，防止另外的负的发票金额添加到数据库中，但同时，希望按原样保留已存在的记录。

要添加一个约束，但是又不应用到已存在的数据中，可以在执行 ALTER TABLE 语句添加

www.TopSage.com

约束时使用 WITH NOCHECK 选项。按照惯例，先看个示例。

在 Accounting 数据库中创建的 Customers 表中有一个名为 Phone 的字段。Phone 字段在创建时的数据类型是 char，因为希望所有电话号码都有一样的长度。本书将长度设置为 15，确保有足够的空间给所有格式的字符。然而，并没有做任何其他事情来确保插入到数据库的记录的确和期望的格式条件相匹配。为了进行检验，下面将插入一个不是所希望格式的记录，但是可能是某人输入号码时很容易发生的错误：

```
INSERT INTO Customers
   (CustomerName,
   Address1,
   Address2,
   City,
   State,
   Zip,
   Contact,
   Phone,
   FedIDNo,
   DateInSystem)
VALUES
   ('MyCust',
   '123 Anywhere',
   '',
   'Reno',
   'NV',
   80808,
   'Joe Bob',
   '555-1212',
   '931234567',
   GETDATE ())
```

下面添加一个约束来控制 Phone 字段的格式：

```
ALTER TABLE Customers
   ADD CONSTRAINT CN_CustomerPhoneNo
   CHECK
   (Phone LIKE '([0-9][0-9][0-9]) [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]')
```

当运行上述代码时，出现一个问题：

```
消息 547，级别 16，状态 0，第 1 行
ALTER TABLE 语句与 CHECK 约束 "CN_CustomerPhoneNo"冲突。
该冲突发生于数据库"Accounting"，表"db0.Customers"，column "Phone"。
```

除非已存在的数据满足约束条件，否则 SQL Server 将不会创建约束。要解决这个问题，需要修改已存在的数据，或者必须在 ALTER 语句中使用 WITH NOCHECK 选项。如下代码所示，在语句中添加 WITH NOCHECK：

```
ALTER TABLE Customers
   WITH NOCHECK
   ADD CONSTRAINT CN_CustomerPhoneNo
   CHECK
```


www.TopSage.com

```
(Phone LIKE '([0-9][0-9][0-9]) [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]')
```

现在如果再次运行同样的 INSERT 语句(这次插入时没有出现上次那样的问题)，则约束起作用了，数据被拒绝：

```
消息 547，级别 16，状态 0，第 1 行
INSERT 语句与 CHECK 约束"CN_CustomerPhoneNo" 冲突。
该冲突发生于数据库 "Accounting"，表 "dbo.Customers"，column 'Phone'。
```

然而，如果修改 INSERT 语句以符合约束条件，然后重新执行，那么数据行正常插入：

```
INSERT INTO Customers
   (CustomerName,
   Address1,
   Address2,
   City,
   State,
   Zip,
   Contact,
   Phone,
   FedIDNo,
   DateInSystem)
VALUES
   ('MyCust',
   '123 Anywhere',
   '',
   'Reno',
   'NV',
   80808,
   'Joe Bob',
   '(800)555-1212',
   '931234567',
   GETDATE ())
```

这时，试着在 Customers 表上运行一个 SELECT 语句。将会看到符合 CHECK 约束条件的数据以及不符合条件的数据并存：

```
SELECT CustomerNo, CustomerName, Phone FROM Customers

CustomerNo     CustomerName            Phone
------------   ---------------------   --------------
1              Billy Bob's Shoes      (360) 555-1234
2              Customer1                    553-1212
3              MyCust                       555-1212
5              MyCust                 (800) 555-1212

(2 行受影响)
```

保持原来的数据以备向后引用，但是新的数据必须符合新的条件。

6.6.2 临时禁用已存在的约束

现在，您已经明白了为什么需要添加新的约束而不对原来的数据作检查了。但是为什么希望临时禁用已存在的约束呢？为什么要将已知的无效数据添加到数据库中呢？实际上，最普通

www.TopSage.com

的原因和使用 WITH NOCHECK 选项的原因一样——为了原来的数据。

原来的数据不只是以添加到数据库中的形式存在，也可能是从遗留的数据库或者其他系统中导入的数据。不管什么原因，存在同样的问题——有一些已存在的数据和规则不匹配，而您需要将数据导入到表中。

当然一种方法是删除约束，添加需要的数据，然后使用 WITH NOCHECK 将约束添加回去。但是，这太麻烦。幸运的是，不需要这么做。相反，可以采用名为 NOCHECK 的选项来运行 ALTER 语句，这样取消需要处理的约束。下面的代码禁用了 6.6.1 节添加的 CHECK 约束：

```
ALTER TABLE Customers
   NOCHECK
   CONSTRAINT CN_CustomerPhoneNo
```

现在可以再次运行 INSERT 语句——已经证实，如果约束有效，这个语句就不能正常工作：

```
INSERT INTO Customers
   (CustomerName,
   Address1,
   Address2,
   City,
   State,
   Zip,
   Contact,
   Phone,
   FedIDNo,
   DateInSystem)
VALUES
   ('MyCust',
   '123 Anywhere',
   '',
   'Reno',
   'NV',
   80808,
   'Joe Bob',
   '555-1212',
   '931234567',
   GETDATE())
```

这样又可以向表中插入格式不匹配的数据。

现在，您可能会问：怎么知道约束是否是启用的呢？如果需要创建一些测试数据插入来验证约束是否是启用的，那么这是非常麻烦的。像很多(并非全部)这类的两难问题，SQL Server 提供一个过程来显示约束的状态，它就是已介绍过的 sp_helpconstraint 约束。要在 Customers 表中执行这个约束是很简单的：

```
EXEC sp_helpconstraint Customers
```

这个结果很长，不适合在本书页面中显示。不过，这个过程生成的第二个结果集包含一个称为 state_enabled 的列。这个列指示了约束的状态——在这里，当前状态应该是 Disabled。

当准备再次启用约束时，只需要简单地以 CHECK 代替 NO CHECK 来执行同样的命令：

```
ALTER TABLE Customers
```


www.TopSage.com

```
    CHECK
    CONSTRAINT CN_CustomerPhoneNo
```

如果运行 INSERT 语句来验证约束重新起作用，就会看到一个熟悉的错误信息：

```
消息 547，级别 16，状态 0，第 1 行
INSERT 语句与 CHECK 约束 "CN_CustomerPhoneNo"冲突。
该冲突发生于数据库"Accounting"，表 "dbo.Customers"，column 'Phone'。
语句已终止。
```

另一种选择当然是再次运行 sp_helpconstraint，检查 status_enabled 列。如果显示 Enabled，那么约束一定是重新起作用了。

6.7 规则和默认值

规则和默认值的应用要早于 CHECK 和 DEFAULT 约束。它们是较老的 SQL Server 备用约束的一部分，当然也不是没有优点。

因此，建议您了解一下它们的向后兼容性和遗留代码，这里不再详细解释。规则和默认值不遵循 ANSI 标准(这产生了移植问题)，而且可执行性能不如约束好。自 7.0 版本之后(即 4 个版本，约 10 年时间)，Microsoft 列出规则和默认值只是为了向后兼容，而不准备在以后继续支持这个特性。在此我不是建议您开始更改，将所有旧代码替换掉，但是在生成新代码时应该使用约束。

将规则和默认值与约束区分开的主要原因在于它们有本质的不同：约束是一个表的特征，本身没有存在形式，而规则和默认值是表或自身的实际的对象，本身存在。约束是在表定义中定义的，而规则和默认值是单独定义，然后“绑定”到表上的。

规则和默认值的独立对象特性使得它们可以在重用时不用重新定义。实际上，规则和默认值不限于被绑定到表上，它们也可以绑定到数据类型上——这有助于创建高度功能化的用户自定义数据类型。

6.7.1 规则

规则和 CHECK 约束非常相似。它们之间的唯一区别是规则每次只能作用于一个列。可以将同一个规则分别绑定到一个表中的多个列，但是规则分别作用于每个列，根本不会意识到其他列的存在。像 QtyShipped <= QtyOrdered 这样的约束定义不适用于规则(它引用了多个列)，而 LIKE ([0-9][0-9][0-9])这样的定义适用于规则(它只作用于规则所绑定的列)。

下面定义一个规则，这样就可以首先看到区别所在：

```
CREATE RULE SalaryRule
  AS @Salary > 0
```

注意，这里比较的是一个变量——不管被检查的列是什么值，这个值将用于替换@Salary。因此，在这个示例中，规则所绑定的任何列的值都必须大于 0。

如果希望返回看看规则看起来是什么样子的，那么可以使用 sp_helptext：

www.TopSage.com

```
EXEC sp_helptext SalaryRule
```

结果将显示精确的规则定义：

```
Text
------------------------------------
CREATE RULE SalaryRule
    AS @Salary > 0
```

现在已经有一个规则了，但是这没起任何作用。如果试图在 Employees 表中插入一个记录，则可以插入任何值，除了数据类型外没有其他任何限制。

为了激活这个规则，需要利用一个称为 sp_bindrule 的特殊存储过程。要将 SalaryRule 绑定到 Employees 表中的 Salary 列。语法如下所示：

```
sp_bindrule <'rule'>, <'object_name'>, [<'futureonly_flag'>]
```

rule 部分很简单——就是希望绑定的规则。object_name 也很简单——就是希望规则绑定到的对象(列或者用户自定义数据类型)。唯一比较奇怪的参数是 futureonly_flag，它只在规则绑定到用户自定义数据类型时应用。默认关闭该选项。然而，如果将这个参数设置为 True 或者传递一个 1，那么绑定规则只应用于用户自定义数据类型所绑定的新的列。任何采用原来的数据类型的列仍然使用原来的形式。

因为将规则绑定到一列，所以语法中只需要前面两个参数：

```
EXEC sp_bindrule 'SalaryRule', 'Employees.Salary';
```

再仔细看看 object_name 参数——这里使用“.”来分隔 Employees 和 Salary，为什么要这样呢？因为在将规则绑定到任何特定的表上之前，规则不需要与任何特定的表和列关联，所以需要说明规则要绑定的表和列。如果不使用 tablename.column 命名格式，那么 SQL Server 将假设所命名的是用户自定义数据类型——如果没有发现这种类型，那么会得到一个错误消息；如果本不想将规则绑定到一个数据类型，那么会出现混乱：

```
消息 15148，级别 16，状态 1，过程 sp_bindrule，第 190 行数据类型或表列 'Salary' 不存在，或你没有所需的权限。
```

在本例中，试着插入或者更新一个带负值的 Employees 记录，这样违反了规则，会产生一个错误。

如果希望在这个列中删除规则，那么可以使用 sp_unbindrule：

```
EXEC sp_unbindrule 'Employees.Salary'
```

这里的 futureonly_flag 参数通常是可选的，但是不要在这个特定的示例中使用。如果使用 sp_unbindrule 时打开 futureonly_flag 选项，并且只用于用户自定义数据类型(而不是特定的列)，那么解除绑定只适用于以后使用该数据类型——使用该数据类型的已存在的列仍然使用这个规则。

删除规则

如果希望将规则从数据库中彻底删除，那么可以在表中使用非常熟悉的 DROP 语法。

www.TopSage.com

```
DROP RULE <rule name>
```

6.7.2 默认值

与规则和 CHECK 约束的类似程度相比，默认值更类似于 DEFAULT 约束。实际上，它们的工作原理一样，区别在于它们被追加到表中的方式和对用户自定义数据类型的默认值(是对象，而不是约束)支持。

注意：

很多人都难以掌握默认值和 DEFAULT 约束的概念。毕竟，它们具有几乎相同的名称。如果说“默认值”，那么可能是指基于对象的默认值(在本节中所讲的)，或者是指实际的默认值(如果不提供一个显式值，就提供默认值)的简写。如果说“DEFAULT 约束”，那么指的是不基于对象的解决方案——这个解决方案是表定义的一个组成部分。

定义默认值的语法和定义规则的语法类似：

```
CREATE DEFAULT <default name>
AS <default value>
```

因此，假设要为 Salary 定义一个值为 0 的默认值：

```
CREATE DEFAULT SalaryDefault
   AS 0
```

同样，如果不绑定到一个对象上，则默认值是不起作用的。要绑定的话，使用 sp_bindefault。除了过程名称不同外，这个过程和 sp_bindrule 过程的语法完全一样：

```
EXEC sp_bindefault 'SalaryDefault', 'Employees.Salary'
```

要从表中解除默认值的绑定，使用 sp_unbindefault：

```
EXEC sp_unbindefault 'Employees.Salary'
```

要记住，futureonly_flag 也适用于这个存储过程，只不过这里没有使用而已。

删除默认值

如果要从数据库中彻底删除一个默认值，则可以使用 DROP 语法。这一语法和在表和规则中使用的语法相同：

```
DROP DEFAULT <default name>
```

6.7.3 确定哪个表和数据类型使用给定的规则或默认值

如果希望删除或者修改规则或默认值，那么您可能想先看看哪些表和数据类型在使用它们。SQL Server 还是采用系统存储过程来解决这个问题。这个过程是 sp_depends。其语法如下所示：

```
EXEC sp_depends <object name>
```

sp_depends 提供了依赖于您所查询对象的所有对象列表。

www.TopSage.com

注意：

遗憾的是，sp_depends 不一定能提供依赖于一个父对象的每个对象。SQL Server 支持一种称为“延迟名称解析”的机制。延迟名称解析指的是可以创建依赖于另一个对象的对象(主存储过程)，甚至是可以在另一对象(依赖的目标)创建之前创建。例如，SQL Server 现在允许在某表创建之前创建一个引用该表的存储过程。在这种情况下，SQL Server 无法列出该表上的依赖关系。即使在添加此表后，如果使用 sp_depends，也不会提供依赖列表。

6.8 用于实现数据完整性的触发器

尽管本书将用一整章来讲述触发器，但是在讨论约束、规则和默认值时，还是要提及一下它。

触发器最常见的用途是实现数据完整性规则。因为后面将用一章的内容来介绍触发器，所以这里不深入讲述，而只是要说明触发器有很多方法来实现数据完整性，而这些都是约束或者规则无法做到的。不足之处是触发器需要额外的系统开销，因此几乎在任何情况下都会慢很多。它们本质上就是过程(这也是它们功能强大的原因)，但是它们在所有其他事情完成之后发生，应当作为最后采用的方法。

6.9 如何选择

既然现在有这么多选择，那么如何挑选合适的约束呢？有些约束是比较独立的(主键约束、外键约束和唯一约束)——可以使用任何一个，或者什么都不用。其他的约束则互相交错，在决定如何选择时会让人有些迷惑。从本章中可以知道关于各个选项的优点和缺点的一些提示，但是可从总体来作一下比较，这样更能作出适当的决定，如表 6-4 所示。

表 6-4

限 制	优 点	缺 点
约束	快速 可以引用其他列 在命令执行前发生 遵循 ANSI 标准	必须对每个表重新定义 不能引用其他表 不能绑定到数据类型
规则、默认值	独立的对象 可重用 可以绑定到数据类型 命令执行前发生	稍慢 不能跨列引用 不能引用其他表 实际上只用于向后兼容
触发器	非常灵活 可以引用其他列或者其他表 可以通过.NET 引用 SQL Server 之外的其他信息	在命令执行之后发生 系统开销很大

www.TopSage.com

如果要实现更健壮的逻辑模型以及广泛使用用户自定义数据类型，则一般使用规则和默认值。在这种情况下，规则和默认值可以提供很多功能，容易管理，而不用太多的编程开销——您只需要明白在以后可能不再采用它们。可能不是很快，但在未来的某一天。

只有在不能选择约束时使用触发器。和约束一样，它们被附加到表中，而且必须对创建的每个表重新定义。好的方面是，触发器几乎可以做数据完整性方面的任何操作。实际上，它们过去是强制外键(在外键约束出现之前)的常用方法。本书后面将详细讲述这些内容。

而在其他情况下，应将约束作为数据完整性解决方案的选择。它们执行速度快，而且不难创建。它们的缺点是功能有限(除了外键约束，都不能引用其他表)，而且对于通用约束逻辑来说，需要一次次地重新定义。

注意：

不管采用哪种完整性机制(键、触发器、约束、规则、默认值)，都要进行权衡。

添加到数据库中的每项内容都会增加额外的系统开销，因此在添加任何内容之前，需要确认所添加的内容都有意义。要避免出现像多余完整性实现(例如，我经常碰到某个数据库既定义了外键来实现参照完整性，又使用触发器做同样的事情)这样的情况。在添加下一个约束之前，确保清楚自己已经有了哪些约束，而且确保确切知道要使用它实现什么。

6.10 本章小结

本章讲述的不同类型的数据完整性机制是实现健壮数据库的支柱的一部分。也许 RDBMS 最强大的地方就是现在数据库可以负责数据完整性，而不用依赖应用程序。这意味着即使是即席查询也要遵循数据规则，在数据完整性问题上，对多个应用程序的要求是一样的。

接下来的章节将讲述一些形式的约束和索引之间的关系，而且还将讲述可用触发器实现的高级数据完整性规则。此外，还要学习选用这些不同机制对设计决策的影响。

www.TopSage.com

第 7 章

更复杂的查询

在多年前我第一次编写有关 SQL Server 的内容时，碰到的一个问题是在知识体系的哪一部分介绍更复杂的查询——本书再次碰到了这个问题。这是个类似“先有鸡还是先有蛋”的问题——即是先讲述脚本、变量之类的问题呢，还是先讲述在初学者编写服务器端的脚本之前就可能使用到的内容呢？这次就先讲述更多查询内容。

本章的一些概念将启发一种新的思维方式。您已经了解一些处理连接的方法，但是还没有达到本章的那个深度。即使您没有很多过程式编程方面的经验，也会自然地倾向将复杂问题分解为更小的部分(子过程、逻辑步骤)，而不是整体(以“集合”或者 SQL 的方式)一起解决。

虽然 SQL Server 2008 可以比以往更好地支持过程式语言，但是这里的一个挑战是开始时尽量将问题作为整体来考虑。确定不能从单个的查询中解决整个问题。即使不能整体考虑，通常可以将问题分解为几个小的查询，然后将这些查询组合成一个大的查询，一次完成整个任务。尽量从整体看待问题，如果不能做到这一点，那么将问题分解，然后再次将问题组合成一个尽可能大的有意义的问题。

这个“新思维方式”的核心在于——整体看待问题而不是分步骤地看待。在使用大多数的语言编程时，通常以一种线性的思维方式工作。然而，使用 SQL 时，需要更多地考虑集合理论。可以将这些与数学概念来类比，即 A 并 B 或者 A 交 B 的概念。更少地考虑分步骤处理数据，而更多地考虑将数据“配合”在一起。

本章将使用数据配合的概念，要求在一个查询中解决多个问题。本质上，本章将学习把看似多个查询的问题集合起来，放置在一个完整的单元中执行。另外，还将了解查询的性能，以及如何尽最大可能地利用查询。

本章的主题包括：

- 嵌套子查询
- 关联子查询
- 派生表
- 使用 EXISTS 运算符
- MERGE
- 优化查询性能

www.TopSage.com

本章将学习如何使用子查询，将那些看似不可能的查询变得完全有可能，了解一些奇怪的设置如何使得查询性能大不相同。

7.1 子查询的概念

子查询是嵌套在另一个查询中的普通 T-SQL 查询。在有一个 SELECT 语句作为另一个查询的部分数据或条件的基础时，通过使用小括号创建子查询。

子查询通常用于满足下列某个需求。

- 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
- 为父查询中的每个记录提供一个查询表

有些子查询很容易想到和构建，但是有一些则非常复杂——这通常取决于内部(子)查询和外部(顶层)查询之间关系的复杂度。

还值得注意的是，大部分(当然不是全部)子查询可以使用连接来编写。在可以使用连接的地方，出于很多原因连接是首选的方式，本书剩余部分将继续探讨这个问题。

提示：

我曾经和一个同事就连接与子查询的问题进行了很长的争论(大概来回有 20 或者 30 封 e-mail，使用了示例，推理等方式，经过好几天的争论)。

传统的逻辑总是使用连接，而这也是我所推崇的(由于经验而不是由于传统逻辑——您已经在本书几个地方看到我指出传统逻辑可能是假设的)。我同事则推崇子查询实际上系统开销更少的理念——我决定验证一下。

最后我发现(正如您所期望的)在特定的条件下，这些说法都是对的。在您具有一定背景知识以后，本章结尾将充分探讨这些条件。

读者已经从理论上知道了子查询是什么。现在来看看子查询的一些具体类型和示例。

构建嵌套子查询

嵌套子查询只在一个方向嵌套——返回在外部查询中使用的单个值，或者与 IN 运算符一起使用的一个完整的值列表。在需要使用一个显式的“=”运算符时，将使用一个返回单个值的查询——这就是一行中的一列。如果希望返回一个列表，那么需要在外部查询中使用 IN 运算符。

从不严格意义上说，查询语法看起来像下面的两个语法模板：

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

或者：

www.TopSage.com

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

显然，确切的语法会有所变化。不仅因为需要替换选择列表和确切的表名称，而且因为在内部查询和/或外部查询中有一个多表连接。

1. 使用返回单个值的 SELECT 语句的嵌套查询

下面通过一个显式的示例来了解问题的本质。例如，假设希望知道第一天通过系统销售的产品的每个条目的 ProductID。

如果已经知道第一天将订单放到系统中的的日期，那么就没有问题了，查询代码如下所示：

```
SELECT DISTINCT sod.ProductID
FROM Sales.SalesOrderHeader soh
JOIN Sales.SalesOrderDetail sod
 ON soh.SalesOrderID = sod.SalesOrderID
WHERE OrderDate = '07/01/2001'; --This is first OrderDate in the system
```

这产生了正确的结果：

```
ProductID
-----------
707
708
709
...
...
776
777
778

(47 行受影响)
```

但是，例如定期清理系统中的数据，不过还是希望将同样的问题作为自动报表的一部分。

既然要自动生成，就不能运行一个查询来找到系统中的第一天，然后手动添加到查询中——但或许也可以这样？实际上，回答是肯定的。就是将所有的内容放到一个语句中：

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

一切就这么快速而又简单。内部查询(SELECT MIN…)检索单个的值用于外部查询。因为使用一个“等于”符号，所以内部查询必须只能返回一行中的一列，否则会得到一个运行时错误。

www.TopSage.com

提示：

在这个新查询中添加了订单日期。尽管这对于查询报告适当的 ProductID 来说不是必要的，但它确实说明了这些 ProductID 相对应的日期。虽然在第一个查询中知道日期，因为显式指定了；但由于在这个新查询中，这个日期是数据驱动的，因此通常有必要作为结果的一部分提供它。

2. 使用返回多个值的子查询的嵌套查询

也许，最常见的一种子查询是检索某些形式的域列表，然后用作另一个查询的条件。

对此，下面将回顾第 4 章介绍外部连接时讨论的一个问题。我们需要得到具有折扣信息的所有产品的列表。

可以这样编写代码：

```
SELECT ProductID, Name
FROM Production.Product
WHERE ProductID IN (
 SELECT ProductID FROM Sales.SpecialOfferProduct);
```

结果返回 295 行：

```
ProductID     Name
-----------   ---------------------------------------------------
680           HL Road Frame - Black, 58
706           HL Road Frame - Red, 58
707           Sport-100 Helmet, Red
...
...
997           Road-750 Black, 44
998           Road-750 Black, 48
999           Road-750 Black, 52

(295 行受影响)
```

尽管这种工作方式很好，但更好的方式是使用内部连接实现，而不是使用嵌套的 SELECT。例如，可以通过运行简单的连接来得到同样的结果：

```
SELECT DISTINCT pp.ProductID, Name
FROM Production.Product pp
JOIN Sales.SpecialOfferProduct ssop
 ON pp.ProductID = ssop.ProductID;
```

出于性能方面的考虑，如果没有特别的理由要使用嵌套的 SELECT 的话，则还应使用连接方法作为默认解决方案——在本章结束前会进一步讨论这个问题。

提示：

SQL Server 可以灵活处理这类问题。在大部分情况下，SQL Server 实际上会将嵌套子查询方法解析为和使用连接一样的查询计划——事实上，如果检查嵌套子查询和前面的连接的查询计划，就会发现它们是完全相同的计划。因此，在大多数情况下，这两种方法是没多大区别的。

www.TopSage.com

当然，问题是这里所说的是“大多数情况”。当查询计划不同时，连接通常是更好的选择，因此推荐在默认情况下使用连接的语法。

3. 使用嵌套的 SELECT 发现孤立的记录

这种嵌套的 SELECT 和前面的示例几乎相同，除了添加了 NOT 运算符。这使得在转换连接语法时设置为等同于外部连接而不是内部连接。

在使用嵌套的 SELECT 语法前，先来看看第 4 章的另外一个外部连接示例。在这个查询中，需要标识所有没有匹配商品的折扣记录：

```
SELECT Description
FROM Sales.SpecialOfferProduct ssop
RIGHT OUTER JOIN Sales.SpecialOffer sso
 ON ssop.SpecialOfferID = sso.SpecialOfferID
WHERE sso.SpecialOfferID != 1
   AND ssop.SpecialOfferID IS NULL;
```

结果返回 1 行：

```
Description
---------------------
Volume Discount over 60

(1 行受影响)
```

按理来说这是应该使用的方法(或者使用 LETT JOIN)。然而，不能说通常是这样做的。连接通常考虑更多的问题，所以一般使用嵌套的 SELECT。

试试是否可以自己编写这个嵌套 SELECT。在写完之后，回头再来看看下面的代码。

代码如下所示：

```
SELECT Description
FROM Sales.SpecialOffer sso
WHERE sso.SpecialOfferID != 1
 AND sso.SpecialOfferID NOT IN
   (SELECT SpecialOfferID FROM Sales.SpecialOfferProduct);
```

结果得到的记录是一样的。

7.2　关联子查询

对于“关联子查询”这个概念要引起注意。如果确实理解它，那么可以使您变得与众不同。不过，所谓“了解”不仅是指理解其工作原理，还要知道其重要性。

关联子查询是那些使不可能的事情变成可能的事物之一。而且，经常能使几行代码变成一行，提高相应的性能。存在的问题是需要不同寻常的思维。关联子查询可能是 SQL 中一个最容易学习并理解的概念，但是又很容易被人忘记，因为其采用了与常人不同的思维方式。如果您能想到使用它，那么就能解决常人无法解决的问题，也能够获得更好的查询性能。

www.TopSage.com

7.2.1 关联子查询的工作原理

关联子查询和刚学过的嵌套子查询的不同之处在于信息传递是双向的，而不是单向的。在嵌套子查询中，内部查询只处理了一次，然后将信息传递到外部查询，而外部查询也只执行一次——提供和您自己可能输入的完全相同的值或列表。

而在关联子查询中，内部查询用外部查询提供的信息运行，反之亦然。这似乎令人有些困惑，但是这是个一分 3 个步骤进行的处理过程。

(1) 外部查询获得一个记录，然后将该记录传递到内部查询。

(2) 内部查询根据传递的值执行。

(3) 然后内部查询将结果值传回外部查询，而外部查询利用这些值完成处理过程。

7.2.2 在 WHERE 子句中的关联子查询

考虑到读者对此可能有些疑惑，所以先来看一个示例。

回到 AdventureWorks2008 数据库，仍然讨论那个查看第一天放置到系统中的订单的查询。然而，这次希望添加一个新的要求：需要知道系统中每位客户第一个订单的 Sales OrderID 和 OrderDate。即需要知道客户第一天下订单的日期和这些订单的 ID。下面逐条来分析。

首先，希望知道每个结果的 OrderDate、SalesOrderID 和 CustomerID。所有这些信息都可以在 SalesOrderHeader 表中找到，因此查询将基于该表，至少是部分基于该表。

然后，需要知道系统中每个客户第一天下订单的日期。这里就需要一点技巧。当使用嵌套子查询时，只在整个文件中查找第一天——现在需要查找每个客户第一天下订单的日期。

如果用两个单独的查询来完成，那也没有太多困难——只需要创建一个临时表，然后与之连接。

提示：

顾名思义，临时表就是为临时使用而创建的表，在处理完成之后将删除。临时表存在的确切时间是变化的，这一内容超出了本书的范围。不过本书后面会再讲一些临时表的内容。

使用临时表的解决方案有点类似如下所示：

```
USE AdventureWorks2008;

-- Get a list of customers and the date of their first order
SELECT soh.CustomerID, MIN(soh.OrderDate) AS OrderDate
INTO #MinOrderDates
FROM Sales.SalesOrderHeader soh
GROUP BY soh.CustomerID;

-- Do something additional with that information
SELECT soh.CustomerID, soh.SalesOrderID, soh.OrderDate
FROM Sales.SalesOrderHeader soh
JOIN #MinOrderDates t
   ON soh.CustomerID = t.CustomerID
   AND soh.OrderDate = t.OrderDate
ORDER BY soh.CustomerID;

DROP TABLE #MinOrderDates;
```


www.TopSage.com

结果得到 19000 多行：

```
(19119 行受影响)
CustomerID      SalesOrderID  OrderDate
-----------     ------------  -----------------------
11000           43793         2001-07-22 00:00:00.000
11001           43767         2001-07-18 00:00:00.000
11002           43736         2001-07-10 00:00:00.000
...
...
30116           51106         2003-07-01 00:00:00.000
30117           43865         2001-08-01 00:00:00.000
30118           47378         2002-09-01 00:00:00.000
(19134 行受影响)
```

提示：

正如前面所述，不要担心您的结果和上面显示的稍微有点不同—— 这只是说明您对 AdventureWorks2008 数据库进行的处理和这里或多或少有些不同。

事实上，将看到结果中有两条不同的“受影响的行”的信息，这说明了构建了两个完全不同的结果集。这通常会影响到性能。这将在探讨其他一些选择后再进一步说明。

有时，在不使用游标的情况下，使用多查询的方法是唯一的方法——不过这里并不是这样的。

如果希望在一个查询中运行，那么需要找到一种方法来查看每个客户。可以让内部查询基于当前外部查询的 CustomerID 来执行查找。然后需要将返回值传递给外部查询，这样可基于最早的订单日期进行匹配。

代码如下所示：

```
SELECT soh1.CustomerID, soh1.SalesOrderID, soh1.OrderDate
FROM Sales.SalesOrderHeader soh1
WHERE soh1.OrderDate = (SELECT Min(soh2.OrderDate)
                        FROM Sales.SalesOrderHeader soh2
                        WHERE soh2.CustomerID = soh1.CustomerID)
ORDER BY CustomerID;
```

这样得到同样的 19134 行：

```
CustomerID      SalesOrderID  OrderDate
-----------     ------------  -----------------------
11000           43793         2001-07-22 00:00:00.000
11001           43767         2001-07-18 00:00:00.000
11002           43736         2001-07-10 00:00:00.000
...
...
30116           51106         2003-07-01 00:00:00.000
30117           43865         2001-08-01 00:00:00.000
30118           47378         2002-09-01 00:00:00.000

(19134 行受影响)
```

在这个查询中要注意以下内容。

www.TopSage.com

- 只看到一条“受影响的行”的信息——暗示只执行了一个查询计划。
- 外部查询(在这个示例中)看起来很像一个嵌套子查询。然而内部查询显式引用外部查询(注意使用了“o1”别名)。
- 在两个查询中都使用了别名——虽然外部查询看起来不需要使用别名——因为只有显式引用另外查询的一个列(内部引用外部的列，反之亦然)时才需要。

注意：

最后那个有关需要别名的问题有很大迷惑性。事实上有时需要，而有时又不需要。在本章前面的嵌套子查询中，一般都不使用别名，但在处理关联子查询时都使用别名。

最严格的规则是必须给需要被其他查询引用的表(以及相应的列)取别名。问题是这很容易令人迷惑。安全的方法是一律都取别名——这样可以明确知道是在哪个查询中的哪个表提取信息。

只有一次看到 19 134 行受影响这一信息。这是因为只一次影响到 19 134 行。通过观察，可推测这个版本可能比两个查询的版本运行得更快，而实际上是这样的。同样，稍后会作进一步了解。

在这个特殊查询中，外部查询只在 WHERE 子句中引用内部查询——也可以在选择列表中包含来自内部查询的数据。

通常，由自己决定是否使用别名，但是对于关联子查询来说，通常是需要的。这个特殊的查询非常适合用来说明原因，因为内部查询和外部查询基于同一个表。既然两个查询都需要从对方获取信息，如果没有别名，就无法知道需要的数据来自哪个表实例。

在 SELECT 列表中的关联子查询

子查询也可以用于在选择结果中提供不同的答案。当所要查找的信息与查询中的其他信息完全不同时，经常出现这种情况(例如，需要一个字段的聚合结果，但是又不希望所有这些数据影响其他返回的字段)。

要进行验证，可运行 7.1 节使用的查询的修改版本。这里需要查找的是客户的账号和在哪天开始订购某物。

这与初看起来有较明显的改变。现在要求查找客户的账号，这意味着需要牵涉到 Customers 表。另外，不再需要加入任何条件——而需要查找所有客户(没有限制)，并要知道客户开始订购的日期。

这个查询实际上比上一个查询更简单些，代码如下所示：

```
SELECT sc.AccountNumber,
   (SELECT Min(OrderDate)
           FROM Sales.SalesOrderHeader soh
           WHERE soh.CustomerID = sc.CustomerID)
           AS OrderDate
FROM Sales.Customer sc;
```

返回的数据如下所示：

```
AccountNumber OrderDate
------------- -----------------------
AW00000001    NULL
```


www.TopSage.com

```
AW00000002     NULL
AW00000007     NULL
...
...
AW00030102     2002-12-01 00:00:00.000
AW00030104     2002-07-01 00:00:00.000
AW00030113     2002-09-01 00:00:00.000

(19820 行受影响)
```

可注意到，有几行的 Order Date 列有 NULL 值。为什么会这样？原因是在 Orders 表中没有记录与 Customers 表(外部查询)的当前记录匹配。

所以需要先来了解一个在这种情况下非常有用的函数——ISNULL()。

7.2.3　处理 NULL 数据——ISNULL 函数

实际上有好几个函数可以处理 NULL 数据。而在这里特别有用的是 ISNULL()。ISNULL()接受一个变量(第 11 章将讲述)或者表达式来验证其是否是一个 NULL 值。如果确实是 NULL 值，那么函数返回其他预先指定的值。如果原值不是 NULL，那么返回原值。这个语法非常简单：

```
ISNULL(<expression to test>, <replacement value if null>)
```

示例如表 7-1 所示。

表　7-1

ISNULL 表达式	返　回　值
ISNULL(NULL, 5)	5
ISNULL(5, 15)	5
ISNULL(MyColumnName, 0) where MyColumnName IS NULL	0
ISNULL(MyColumnName, 0) where MyColumnName = 3	3
ISNULL(MyColumnName, 0) where MyColumnName ='Fred Farmer'	Fred Farmer

现在看看这个函数在查询中起什么作用：

```
SELECT sc.AccountNumber,
 ISNULL(CAST((SELECT Min(OrderDate)
          FROM Sales.SalesOrderHeader soh
          WHERE soh.CustomerID=sc.CustomerID) AS varchar),'NEVER ORDERED')
          AS OrderDate
FROM Sales.Customer sc;
```

现在，有问题的几行示例信息如下所示：

```
...
...
AW00000696 NULL
AW00000697 NULL
AW00000698 NULL
AW00011012 2003-09-17 00:00:00.000
```


www.TopSage.com

```
AW00011013 2003-10-15 00:00:00.000
AW00011014 2003-09-24 00:00:00.000
...
...
```

接下来是更有用的信息：

```
...
...
AW00000696 NEVER ORDERED
AW00000697 NEVER ORDERED
AW00000698 NEVER ORDERED
AW00011012 Sep 17 2003 12:00AM
AW00011013 Oct 15 2003 12:00AM
AW00011014 Sep 24 2003 12:00AM
...
...
```

提示：

可注意到还必须使用 CAST()函数来完成这个工作。原因是要处理强制转换和隐式转换。因为 Order Date 列是 DateTime 类型的，NEVER ORDERED 不能转换为 DateTime 数据类型，所以就产生错误。记住，CAST()能帮助解决类似这样的小问题。本章后面将进一步作介绍。

到现在为止，介绍了关联子查询为 WHERE 子句和选择列表提供的信息。可以在同一个查询中混合使用这两种方式。

7.3 派生表

有时，会碰到不能采用前面所讲的子查询的方式来处理查询结果的情况。例如，对于给定表的每个行，可能在子查询中有多个结果，但是需要比 IN 运算符提供更复杂的操作。实际上，这种情况下需要在子查询中使用 JOIN 运算符。

这时需要了解 SQL 中鲜为人知的结构——派生表。既然派生表由一个查询结果集的列和行构成(既然它们和正常的表一样具有列、行，以及数据类型等，那么为什么不能像正常的表一样使用呢？)。

假设现在希望得到订购某个特定产品(如 HL Mountain Reel Wheel)的所有客户的账号及相应地区列表。那该查询可能如下所示：

```
SELECT sc.AccountNumber, sst.Name
FROM Sales.SalesOrderHeader soh
JOIN Sales.SalesOrderDetail sod
 ON soh.SalesOrderID = sod.SalesOrderID
JOIN Production.Product pp
 ON sod.ProductID = pp.ProductID
JOIN Sales.Customer sc
 ON sc.CustomerID = soh.CustomerID
JOIN Sales.SalesTerritory sst
 ON sst.TerritoryID = sc.TerritoryID
WHERE pp.Name = 'HL Mountain Rear Wheel';
```


www.TopSage.com

这很容易。现在要给您出个难题——假设现在需要知道所有不仅订购了 HL Mountain Rear Wheel 还订购了 HL Mountain Front Wheel 的客户。注意，这里说的是两者都订购的客户——问题出现了。首先，您可能编写下面的代码：

```
WHERE pp.Name = 'HL Mountain Rear Wheel'
 AND pp.Name = 'HL Mountain Front Wheel'
```

但是这样根本没用——每行只对应一个产品，怎么可能同时针对 HL Mountain Rear Wheel 和 HL Mountain Front Wheel 两种产品呢？不对，这样是不起作用的(实际上，虽然这些代码能运行，但是根本没有返回任何行)。

这里真正需要做的是：分别查询订购了 HL Mountain Rear Wheel 的客户和订购了 HL Mountain Front Wheel 的客户，然后将得到的两个查询结果连接起来。但是，怎样连接这两个结果呢？您可能从本节标题已经猜到，就是使用派生表。

要创建派生表，需要做以下两件事情。

- 将生成结果集的查询用小括号括起来
- 给查询结果取别名，这样它可以作为表被引用

因此，语法如下所示：

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

现在应用该语法。因为需要知道既订购了 HL Mountain Rear Wheel 又订购了 HL Mountain Front Wheel 的所有公司的账号和相应地区。查询代码如下所示：

```
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;
```

结果得到 58 个账号：

www.TopSage.com

```
AccountNumber Name
------------- ----------------------------
AW00029484    Southeast
AW00029490    Northwest
AW00029499    Canada
...
...
AW00030108    Canada
AW00030113    United Kingdom
AW00030118    Central

(58 行受影响)
```

如果需要检验这个结果，则可以分别运行得到两个派生表的查询，然后比较结果。

提示：

对于这个特定的查询，需要使用 DISTINCT 关键字。如果不采用的话，那么对于每个客户而言，可能会得到多个行——例如，AW00029771 订购了 HL Mountain Rear Wheel 两次，这样会得到两次订购的记录。但事实上只要求知道哪些客户同时订购了这两种产品，而不需要知道每个客户订购了多少次。

这样，可以使那些看似不可能的查询变得可能，而且可以很好地执行。

记住，派生表不是万能的。例如，如果结果集很大，并且得到大量连接记录，那么可能要使用临时表并在上面建立索引(派生表没有索引)。每种情况都是不同的，但重要的是现在又多了一个工具。

7.4 EXISTS 运算符

这里将 EXISTS 称为运算符，但是联机丛书中称之为关键字。这可能只是描述方式不同。EXISTS 运算符与 IN 关键字很相似，但是在有的方面有些不同。

使用 EXISTS 时，并不真正返回数据，而是根据是否存在数据满足查询中 EXISTS 语句所建立的条件标准，返回简单的 TRUE 或者 FALSE 结果。

先看一个示例，这样就能明白如何应用这个运算符。这里要查询的是拥有雇员的人的列表：

```
SELECT BusinessEntityID, LastName + ', ' + FirstName AS Name
FROM Person.Person pp
WHERE EXISTS
   (SELECT BusinessEntityID
         FROM HumanResources.Employee hre
         WHERE hre.BusinessEntityID = pp.BusinessEntityID);
```

正如所预期的，得到了 Person 表的一个较小子集——290 行。

```
BusinessEntityID    Name
----------------    ----------------------------------------------------
263                 Trenary, Jean
```


www.TopSage.com

```
78                  D'sa, Reuben
242                 Poe, Deborah
...
...
95                  Scardelis, Jim
215                 Harrington, Mark
112                 Evans, John

(290 行受影响)
```

也可以很容易地使用连接完成同样的查询：

```
SELECT pp.BusinessEntityID, LastName + ', ' + FirstName AS Name
FROM Person.Person pp
JOIN HumanResources.Employee hre
   ON pp.BusinessEntityID = hre.BusinessEntityID;
```

这个基于连接的语法也得到相同的结果(只是可能排列顺序有些不同)。那么，为什么要使用新的语法呢？简单地说，是为了性能。

当使用 EXISTS 关键字时，SQL Server 不需要执行一行一行的连接，而是寻找记录，直到找到第一个匹配的记录，然后停止在那里。只要有一个匹配，EXISTS 就为 TRUE，不需要继续查找。

下面简单看一下事情的另一方面——如果需要查询没有雇员的人，那么该怎么做？使用第 4 章的连接方法，必须做一些大改动来获得需要的结果。首先，需要使用一个外部连接。然后执行一个比较，看是否有 Employee 记录为 NULL。

代码如下所示：

```
SELECT pp.BusinessEntityID, LastName + ', ' + FirstName AS Name
FROM Person.Person pp
LEFT JOIN HumanResources.Employee hre
   ON pp.BusinessEntityID = hre.BusinessEntityID
WHERE hre.BusinessEntityID IS NULL;
```

执行后返回 19682 行。

```
BusinessEntity    ID Name
--------------    ------------------------------------------------------
293               Abel, Catherine
295               Abercrombie, Kim
2170              Abercrombie, Kim
...
...
2088              Zugelder, Judy
12079             Zukowski, Jake
2089              Zwilling, Michael

(19682 行受影响)
```

如果对此问题改用 EXISTS，只需要添加一个字——NOT：

```
SELECT BusinessEntityID, LastName + ', ' + FirstName AS Name
FROM Person.Person pp
```


www.TopSage.com

```
WHERE NOT EXISTS
   (SELECT BusinessEntityID
           FROM HumanResources.Employee hre
           WHERE hre.BusinessEntityID = pp.BusinessEntityID);
```

这次也返回 19682 行。

在大部分情况下，这里的性能差别比使用内部连接要更加显著。SQL Server 只是对于 EXISTS 语句应用了反向逻辑。在使用 NOT 的情况下，SQL 还是能在发现一个匹配的记录时停止查找——不同的是在查找时知道返回 FALSE 而不是 TRUE。从性能上看，查询的其他方面都是一样的。

以其他方式使用 EXISTS

EXISTS 的一个通常的用法是在运行 CREATE 语句之前检验表是否存在。有时可能希望删除一个已存在的表，如果表已经存在的话，可能使用 ALTER 语句或者其他语句修改已存在的表。一种最常用的使用方法如下所示：

```
IF EXISTS
 (SELECT *
 FROM sys.objects
 WHERE OBJECT_NAME(object_id) = 'foo'
   AND SCHEMA_NAME(schema_id) = 'dbo'
   AND OBJECTPROPERTY(object_id, 'IsUserTable') = 1)
BEGIN
 DROP TABLE dbo.foo;
 PRINT 'Table foo has been dropped';
END
GO

CREATE TABLE dbo.foo
(
 Column1 int IDENTITY(1,1) NOT NULL,
 Column2 varchar(50)NULL
);
```

因为 EXISTS 只返回 TRUE 或者 FALSE，这意味着它像一个优秀的条件表达式一样工作。上面示例仅当表存在时才运行 DROP TABLE 代码；否则，跳过这个部分，直接进入 CREATE 语句。当运行以上脚本时，避免了出现两个错误。首先，当对象存在时，不能运行 CREATE 语句(如果在其他依赖于该操作先执行的表的脚本中运行它，这可能引起其他问题)。其次，当表不存在时，不能删除表(如果能删除，那可能产生一条让安装产品的客户疑惑不解的消息)。这里避免了这两个错误。

作为示例，下面编写创建数据库(这在自动过程中常被跳过)的脚本。但是数据库的创建经常采用含义模糊的方式，如“创建名为‘xxxx’的数据库”。有趣的是，实际安装的用户(通常不知道他们正在做哪些工作)会从包括引号开始，或者创建的数据库太小，或者可能犯很多其他小错误。这时需要有好的技术支持。

下面将构建一个小脚本来创建数据库对象，该对象和 AdventureWorks2008 不冲突。为了安全起见，将它称为 AdventureWorksCreate。同时还使得代码尽量少，因为感兴趣的是 EXISTS

www.TopSage.com

而不是 CREATE 命令：

```
USE MASTER;
GO

IF NOT EXISTS
 (SELECT 'True'
 FROM sys.databases
 WHERE name = 'AdventureWorksCreate')
BEGIN
   CREATE DATABASE AdventureWorksCreate;
END
ELSE
BEGIN
   PRINT 'Database already exists. Skipping CREATE DATABASE Statement';
END
GO
```

第一次运行上面脚本时，不存在名为 AdventureWorksCreate 的数据库(除非您碰巧在之前这样命名了某个对象)，因此得到下面的返回消息：

```
命令已成功完成。
```

这个消息不能准确说明您到底做了什么，但是至少可以知道是否完成了所想要做的。

现在再次运行这个脚本，可以看到下面的返回消息：

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

这样，添加了一个小脚本，它有助于产品的安装用户更有效地工作。安装者可能是购买现有产品的最终用户，或者可能是您自己——无论怎样，都比完全脚本化要好。

总而言之，EXISTS 确实是个很有用的关键字。它可以使得一些查询运行得更快，还可以简化一些查询和脚本。

提示：

这里又是一个容易陷入“传统思维”的地方。虽然在大部分的可采用 EXISTS 有效结构的查询中，EXISTS 要优于其他选项，但是并不总是这样。例如，使用派生表的示例也可以写成使用几个 EXISTS 运算符(每个产品对应一个)的方式，但是派生表的运行速度快一倍。这只是个例外，而不是规则——通常 EXISTS 的性能比派生表的性能更好些。不过要记住这个规则有时有例外情况。

7.5　混合数据类型：CAST 和 CONVERT

CAST 和 CONVERT 都经常被使用。实际上，本章已简单介绍过这两个函数。考虑到要经常使用这两个函数，因此进一步了解一下这两者的功能。

CAST 和 CONVERT 都可以执行数据类型转换。在大部分情况下，两者执行同样的功能，不同的是 CONVERT 还提供一些日期格式转换，而 CAST 没有这个功能。

www.TopSage.com

提示：

您很可能会有疑问："如果 CONVERT 包括了 CAST 的所有功能，而且 CONVERT 还能进行日期转换，那么为什么需要使用 CAST 呢？"简单地说，是为了 ANSI/ISO 兼容。CAST 是 ANSI 兼容的，而 CONVERT 不兼容。

下面看看各自的语法。

```
CAST (expression AS data_type)

CONVERT(data_type, expression[, style])
```

除了一些顺序问题，以及 CONVERT 具有额外的格式选项外(使用 style 参数)，两者的语法基本相同。

CAST 和 CONVERT 可以进行很多数据类型转换。在 SQL Server 不进行隐式转换时，需要这些转换。例如，通常需要将数字转换为字符串。如下所示：

```
SELECT 'The Customer has an Order numbered ' + SalesOrderID
FROM Sales.SalesOrderHeader
WHERE CustomerID = 29825;
```

这将产生错误：

```
消息 245，级别 16，状态 1，第 1 行
在将 varchar 值 'The Customer has an Order numbered' 转换成数据类型 int 时失败。
```

但是如果改变代码，先对数字进行转换：

```
SELECT 'The Customer has an Order numbered ' + CAST(SalesOrderID AS varchar)
FROM Sales.SalesOrderHeader
WHERE CustomerID = 29825;
```

这样将得到相差很大的结果：

```
------------------------------------------------------------------
The Customer has an Order numbered 43659
The Customer has an Order numbered 44305
The Customer has an Order numbered 45061
The Customer has an Order numbered 45779
The Customer has an Order numbered 46604
The Customer has an Order numbered 47693
The Customer has an Order numbered 48730
The Customer has an Order numbered 49822
The Customer has an Order numbered 51081
The Customer has an Order numbered 55234
The Customer has an Order numbered 61173
The Customer has an Order numbered 67260

(12 行受影响)
```

这个转换可能也不是很直观。再例如，需要将 timestamp 列转换为正常的数字。Timestamp 类型的值是个二进制数字，因此转换也不是很麻烦：

www.TopSage.com

```
CREATE TABLE ConvertTest
(
   ColID int IDENTITY,
   ColTS timestamp
);
GO

INSERT INTO ConvertTest
   DEFAULT VALUES;

SELECT ColTS AS Uncoverted, CAST(ColTS AS int) AS Converted
FROM ConvertTest;
```

这将得到下面的结果(您所得到的确切数字可能与此不同)：

```
(1 行受影响)
Uncoverted          Converted
-----------------   -------------
0x00000000000000C9  201

(1 行受影响)
```

还可以转换日期：

```
SELECT OrderDate, CAST(OrderDate AS varchar) AS Converted
FROM Sales.SalesOrderHeader
WHERE SalesOrderID = 43663;
```

这将得到类似下面的结果(根据系统日期配置，您所得到的确切格式可能不同)：

```
OrderDate                   Converted
----------------------- ------------------------------
2001-07-01 00:00:00.000 07 1 2001 12:00AM

(1 行受影响)
```

注意 CAST 也能进行日期转换；不过不能像 CONVERT 那样控制日期格式。例如：

```
SELECT OrderDate, CONVERT(varchar(12), OrderDate, 111) AS Converted
FROM Sales.SalesOrderHeader
WHERE SalesOrderID = 43663;
```

结果如下所示：

```
OrderDate                   Converted
----------------------- ------------
2001-07-01 00:00:00.000 2001/07/01

(1 行受影响)
```

这和 CAST 的结果有很大不同。实际上，可以转换成 34 种两位数字或 4 位数字年份格式的任何一种：

www.TopSage.com

```
SELECT OrderDate, CONVERT(varchar(12), OrderDate, 5) AS Converted
FROM Sales.SalesOrderHeader
WHERE SalesOrderID = 43663;
```

这将得到：

```
OrderDate                  Converted
---------------------- ------------
2001-07-01 00:00:00.000 01-07-01

(1 行受影响)
```

全部需要做的是在 CONVERT 函数后面提供一个代码(前面示例中的 111 给出日本标准，带 4 位数字的年份；5 是意大利标准，带两位数字的年份)说明需要的格式。任何以超过 100 的代码表示的是 4 位数字的年份；而小于 100 的表示两位数字的年份，不过也有一些这样的例外。可用的格式可以在联机丛书的 CONVERT 或者 CASE 主题中找到。

提示：

可以设置一个分隔点让 SQL Server 来识别两位的年份前应该添加 20 还是应该添加 19。默认的分隔点是 49/50—— 两位的年份如果是 49 或者更小时，在前面添加 20。而更大的年份前面将添加 19。这可以在数据库服务器配置中进行更改(第 19 章将讨论管理问题)。

7.6 MERGE 命令

MERGE 命令是 SQL Server 2008 中的新功能，它提供了一种不同的方式来使用 DML 语句。通过 MERGE 命令，可以将多个 DML 动作语句(INSERT、UPDATE、DELETE)组合成一个整体动作，从而提高性能(它们可共享许多相同的物理操作)和简化事务。MERGE 利用一个特殊的 USING 子句，该子句有点类似于 CTE。然后，USING 子句中的结果集可按条件应用于 INSETT、UPDATE 和 DELETE 语句。其基本语法如下所示：

```
MERGE [ TOP ( <expression> ) [ PERCENT ] ]
  [ INTO ] < target table> [ WITH ( <hint> ) ] [ [ AS ] < alias> ]
  USING < source query>
    ON < condition for join with target>
  [ WHEN MATCHED [ AND < clause search condition> ]
      THEN < merge matched> ]
  [ WHEN NOT MATCHED [ BY TARGET ] [ AND < clause search condition> ]
      THEN < merge not matched> ]
  [ WHEN NOT MATCHED BY SOURCE [ AND < clause search condition> ]
      THEN < merge matched> ]
  [ < output clause> ]
  [ OPTION ( < query hint> [ ,...n ] ) ];
```

下面采用一个收货库存的示例来说明。假定为了做报表，创建了一个特殊的销售汇总表。希望每日运行查询，将新的销售记录添加到每月汇总表中。在某月的第一天晚上，几乎没有什么工作可做，因为该月没有其他汇总记录，该天的销售记录也已汇总并插入了。但在第二天晚

www.TopSage.com

上，情况不同了：需要像第一天那样汇总并插入新记录，但(对于该月已销售的产品)只需要更新已有记录。

下面看一下 MERGE 命令如何在一个步骤中同时管理两个动作。在这之前，需要创建汇总表：

```
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)
);
```

这是个非常简单的每月汇总表的示例，可以很容易获得指定年月的某个产品的销售总数。然而，为了利用它，需要定期用细节表中的汇总数据填充它。因此，将使用 MERGE。

首先，需要建立结果集，指明需要从哪些行获取填充到汇总表中的数据。本例中，将选用 2003 年 8 月份，从该月的第一天开始查询：

```
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;
```

这样就得到指定时间内(这里的时间范围恰好仅限于一天)每天的总销售数据(按 ProductID 分类)。

提示：

这里有个小问题。使用 GROUP BY 对 OrderDate 进行分组，但 OrderDate 是 datetime 数据类型的，而不是 date 数据类型的。如果订单带有实际时间，那将与按日期分组订单的假定不符。如果是在生产环境中，那可能希望将 OrderDate 转换为 date 数据类型，或使用 DATEPART 确保按天而不是按时间分组。

现在，准备构建 MERGE 语句：

```
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
```


www.TopSage.com

```
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 语句最后的分号是必须的。尽管大部分 SQL 语句中保留分号是为了向后兼容，但在越来越多的语句中，这变成了强制要求。对于像 MERGE 这样的多部分语句来说尤其如此。

如果运行这一代码，结果是 192 行受影响(假定未对 AdventureWorks2008 中的数据作过修改)。现在，由于 Sales.MonthlyRollup 表是空的，没有任何匹配，因此插入所有行。可以查询 Sales.MonthlyRollup 表进行验证。

```
SELECT *
FROM Sales.MonthlyRollup;
```

结果为 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 行受影响)
```

支持 MERGE 的基本的 SELECT 语句中的每行记录都被插入表中。继续看该月第二天的结果：

```
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
```


www.TopSage.com

```
WHEN NOT MATCHED THEN
 INSERT (Year, Month, ProductID, QtySold)
 VALUES (DATEPART(yy, s.OrderDate),
         DATEPART(m, s.OrderDate),
         s.ProductID,
         s.QtySold);
```

更新日期，继续运行(仿真在该月的第 2 天运行它)，运行结果为 38 行：

```
(38 行受影响)
```

但这次情况有些不同——表中已有一些行是与新一批的销售记录相匹配的。虽然知道影响了 38 行，但如何影响的呢？再次对表运行 SELECT 查询：

```
SELECT *
FROM Sales.MonthlyRollup;
```

结果只得到 194 行，而不是 230 行(即 192 行加 38 行)。实际上，38 行中有 36 行是重复的销售记录，因此被看作是更新，而不是插入。另外两行(ProductID 为 882 和 928)是该月中之前还未销售的产品的销售信息，因此需要作为新行插入——一条语句忽略该数据，但对应的另一条语句会运行。

可以执行类似的操作，根据匹配或不匹配的条件删除行。

7.6.1　BY TARGET 和 BY SOURCE

在上面的示例中，忽略了在决定执行的动作时有关表匹配的问题。由于默认使用 BY TARGET，因此所有的示例(没有使用 BY 关键字)都是根据目标表(在 MERGE 关键字后指定的表)中是否有匹配来进行分析。从匹配的角度看，比较这个概念类似于外部连接。通过对连接的分析可知，可以在源方、目标方或两者间有匹配。如果指定了 BY TARTGET(或不使用 BY 关键字，因为按目标匹配是默认的)，那只有在连接的目标方有匹配时，才应用(插入、更新或删除)动作。同样，如果指定了 BY SOURCE，那么只有连接的源方有匹配时，才应用合并动作。

通常都可以将一个特定合并操作映射到一个特定的匹配场景。

- NOT MATCHED　[BY TARGET]：这通常对应根据源表中的数据将数据插入到表中的场景。
- MATCHED　[BY TARGET]：这暗示了行已经存在目标表上，因此很可能对目标表的行执行更新操作。
- NOT MATCHED　BY SOURCE：这通常用于处理源表中缺少(可能是删除的)的行，在这种场景下，通常将删除目标表中的行(尽管更新行的操作可能只是设置非有效标志或类似标记)。

还有其他可能的组合，但这些基本上概括了大部分 SQL 开发人员遇到的情况。

7.6.2　OUTPUT 子句

MERGE 命令还提供了一个将 MERGE 实际执行的动作的细节数据输出到 SELECT 语句中的选项。OUTPUT 关键字实际上可替代 SELECT，但同时它还有一些特殊运算符来匹配合并的数据，如下所示。

www.TopSage.com

- $action：返回 INSERTED、UPDATED 或 DELETED，表明对特定行执行的动作。
- inserted：对内部工作表的引用，该工作表包含了为给定行插入的数据的引用。注意，这包括了已更新数据的当前值。
- deleted：对内部工作表的引用，该工作表包含了从给定行中删除的数据的引用。注意，这包括了已更新数据之前的值。

提示：

第 15 章讨论触发器时将更详细地介绍 inserted 和 deleted 表。

为了作测试，可重新设置 MonthlyRollup 表，执行带有 OUTPUT 子句的 MERGE 语句。首先截断 MonthlyRollup 表以清除前面的工作：

```
USE AdventureWorks2008
TRUNCATE TABLE Sales.MonthlyRollup;
```

这就清空了表中的所有数据，将表重置为刚使用 CREATE 命令创建完后的状态。现在准备再次执行第一个 MERGE 语句，但这次要包括 OUTPUT 子句。

```
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)
 OUTPUT $action,
         inserted.Year,
         inserted.Month,
         inserted.ProductID,
         inserted.QtySold,
         deleted.Year,
         deleted.Month,
         deleted.ProductID,
         deleted.QtySold;
```

当然，这次会执行和之前同样的动作(插入 192 行)，但得到的结果集提供了有关所执行动作的信息：

www.TopSage.com

```
$action Year Month  ProductID QtySold Year Month ProductID  QtySold
------  ----        --------- -------- ---- ----- --------- ---------
INSERT  2003 8      707       242      NULL NULL  NULL      NULL
INSERT  2003 8      708       281      NULL NULL  NULL      NULL
INSERT  2003 8      711       302      NULL NULL  NULL      NULL
...
...
INSERT  2003 8      997       43       NULL NULL  NULL      NULL
INSERT  2003 8      998       138      NULL NULL  NULL      NULL
INSERT  2003 8      999       103      NULL NULL  NULL      NULL

(192 行受影响)
```

注意，由于在这个特殊的查询中只插入行，因此 deleted 表中的所有数据都为 NULL。当运行第二个 MERGE 语句(同样添加 OUTPUT 子句)时，情况很快发生改变。

```
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)
OUTPUT $action,
       inserted.Year,
       inserted.Month,
       inserted.ProductID,
       inserted.QtySold,
       deleted.Year,
       deleted.Month,
       deleted.ProductID,
       deleted.QtySold;
```

这次可看到多个操作。在 UPDATED 结果中，包括了 inserted(新值)表和 deleted(旧值)表的数据：

```
$action   Year Month ProductID QtySold   Year  Month   ProductID   QtySold
-------   ---- ----- --------- -------   ----- -----   ---------   -------
INSERT    2003  8     928        2       NULL  NULL    NULL        NULL
INSERT    2003 8      882        1       NULL  NULL    NULL        NULL
UPDATE    2003 8      707        249     2003  8       707         242
```


www.TopSage.com

```
...
...
UPDATE    2003 8     963        32     2003   8      963        31
UPDATE    2003 8     970        54     2003   8      970        53
UPDATE    2003 8     998        139    2003   8      998        138
(38 行受影响)
```

7.7 性能方面的考虑

本章已经从宏观上介绍了“怎么做才是最好的选择”。但是，实际操作起来却并不容易。而本节只是想提供一个有关查询性能问题的快速参考，以指导您作出正确的选择。

注意：

本节将讲述的是通常采用的方式。“通常”一词在这里的运用很恰当。在 SQL 中很少有绝对的规则，例外是在所难免的，对于 SQL Server 中的性能问题来说更是如此。

简单来说，您需要评估一个给定查询的性能的重要性。如果性能要求是关键的，那么不要严格遵循这些规则，而是作为一个起点来使用，然后不断测试！

JOIN 与子查询的对比

如本章前面提到的，关于 JOIN 和子查询的问题，作者和同事曾展开过激烈的讨论并且，当两个人都如此坚持自己的观点，从某种观点来说二者都是对的(同样也可以从某种观点说二者都是错的)。

长期以来的传统观点是：如果可能，尽量使用连接替代子查询。这有时是完全正确的。但在现实中，要取决于大量的因素。表 7-2 列举了一些与性能平衡相关的因素，以及建议优先采用的方法。

表 7-2

情　况	优先采用的方法
对于外部查询的每行，子查询返回的值都一样	预查询。声明一个变量，然后选择需要的值赋给这个变量，这样子查询只执行一次，而不是对外部表的每个记录都执行一次
两个表都相对较小(比如不超过 10000 个记录)	子查询。不知道确切的原因，但是运行了几个测试示例，每次性能都很好。因此怀疑是因为查找比连接的系统开销更少
根据所有条件，匹配只会返回一个值	子查询。同样地，和连接整个表相比，子查询只需要找到一个记录然后替换，系统开销更小
根据所有条件，匹配只返回很少的值，在查找列上没有索引	子查询。单个查找或者少量查找的系统开销通常比散列连接的系统开销少
查找表相对很小，但是基表很大	如果可应用的话，就采用嵌套子查询；其次是连接，最后是关联子查询。子查询只查找一次，系统开销相对较少。而采用关联子查询时，需要循环查找很多次——这种情况下，连接是更好的选择

www.TopSage.com

(续表)

情　况	优先采用的方法
关联子查询和连接相比	连接。关联子查询将在内部创建嵌套循环。这可能造成很大的系统开销。在大部分情况下，关联子查询比游标更快，但是比其他可选的方法都慢
派生表和其他任何可选方法	派生表通常需要很大的系统开销，因此要慎用。需要记住的是，派生表运行一次，然后驻留在内存，因此大部分系统开销是在初始创建和(在较大结果集中)缺少索引时。它可能很快，也可能很慢，要视情况而定。所以使用前要慎重考虑
EXISTS 和其他任何可选方法	EXISTS。对于同样的匹配，EXISTS 不需要多次查找，一旦为特定行找到一个匹配，就转向下一个查找，这能大量节省系统开销

这些只是比较常见的，而实际可能有各种各样的情况。

注意：

当性能问题很关键时，一定要对问题的各种解决方案作测试。在很多情况下，采用通常的规则即可，但是不总是如此。通过进行适当测试，可以确保作出正确的选择。

7.8　本章小结

在第 3 章和第 4 章中学习的查询选项涉及了大概 80%的查询情况，但是另外的 20%还是可能会难住您。有时问题出在能否找到您所需要的查询，而有时则是某个特定的查询的性能极差。不管怎样，都会碰到简单查询和连接无法解决的情况。而本章介绍的内容将有助于处理这些复杂情况。

7.9　练习

1. 编写一个查询，以 MM/DD/YY 的格式返回 AdventureWorks2008 中所有雇员的就职日期。
2. 分别使用 JOIN、子查询和 EXISTS 编写查询，列出 AdventureWorks2008 中没有下任何订单的所有客户。
3. 编写查询显示 AdventureWorks2008 中花费超过 70 000 美元的账号所对应的最近 5 个订单。

www.TopSage.com

第 8 章

规范化和其他基本设计问题

到目前为止，您可能对一些表的构建方式及构建原因仍有点困惑。除了一两章外，本书中的示例都采用联机事务处理(OLTP)。同时，本书还将不断地指出 OLTP 和更偏重面向分析的联机分析处理(Online Analysis Processing，OLAP)之间的区别。在大部分示例中，都可以看到一种为最常见的数据库优化的表设计——OLTP。因此，表的示例所具有的数据库布局通常都被规范化为所谓的第三范式。

那么什么是“范式”呢？本章将作详细讲述，但简单地说就是将数据转变成一个有逻辑的、无重复的格式，可以很容易将此格式再重新组合成一个整体。除了规范化(即将数据库转换成范式的过程)之外，本章还将探讨 OLTP 和 OLAP 数据库的特征。另外，本章将通过很多示例来了解约束是如何在整体解决方案中实现的。

提示：

本章也许是本书中最难理解的一章，因为存在一个先学什么的问题。本章中的一些概念将在本书后面解释——如触发器和存储过程。另一方面，在没有理解这些主题在数据库设计中所扮演的角色之前，是难以将这些主题联系起来的。

作者强烈推荐先通读本章，在学习了本书后面几章以后再来回顾本章内容。

8.1 表

虽然不能将表的概念归为是基本的，但是还是要简单了解一下其准确定义。显然，这里说的表(table)不是指厨房里的桌子(table)，而是指数据库的一个主要对象。

表是具有相同常规属性(attribute)的数据实例的集合。这些数据实例组合成数据行和列。表应该表示“现实中”的数据集合(通常称为实体)，而且和其他表之间存在信息关系。关于不同实体(表)和关系(实体之间如何一起工作)的图通常称为实体-关系数据库关系图，即 ER 图。有时“ER 图”还简写为 ERD。

通过不同关系将两个或更多的表相连接，可以根据需要从两个表的组合数据临时创建其他表(可从第 4 章和第 5 章中看到一些)。然后，相关实体的集合组成数据库。

www.TopSage.com

8.2　保持数据“规范”

规范化的概念是现代 OLTP 数据库设计的基石。规范化和关系数据库的概念是同时出现的。两者都来源于 E. F. Codd(IBM)在 1969 年发表的著作。Codd 提出这样的概念：数据库由一系列无序的表组成，可以对这些表进行非过程操作来返回表。

这个概念的几个关键部分如下所示。

- 顺序必须是不重要的。
- 表可以以非过程的方式彼此“关联”(实际上，Codd 称表为“关系”)。
- 通过关联这些基本表，可以创建虚拟表满足新的需求。

规范化是“关系”数据库设计的分支。

提示：

规范化的概念是编程中最常被引用，但是也最容易被误解的概念。人人都以为自己理解规范化，事实上，很多人至少在理论上是理解的。但是，规范化很容易成为很多数据库设计者配戴的十字架——就是表明他们是“真正的”数据库设计师的象征。而实际上是，这只表明他们知道什么是范式——仅此而已。规范化实际上只是数据库设计的一小部分。有时需要规范化数据，有时又需要有意地反规范化数据。即使在规范化过程中，也有很多方式去实现技术上所认定的规范化数据库。

这里需要提醒的是：规范化只是一个理论。一旦选择了是否实现规范化策略，最终拥有的是一个数据库——可能是您所能设计的最好的数据库。不要限于书本上(包括本书)让您做的事情，而是要做适合自己情况的事情。作为本书的作者，我所能做的是给出概念——而不能为您实现，其他作者也不能(至少不能通过文字来)。您需要在这些概念之间选择最合适的、最好的解决方案。现在，开始学习范式，了解范式可以做什么。

首先需要说明的是，总共有 6 种范式。对于那些以前接触过数据库和规范化的人来说，这个数字可能会有些让人吃惊。您很可能听说完全规范化的数据库就是规范化为第三范式的数据库——那不是说只有 3 种范式吗？对于那些认为只有 3 种范式的人来说，如果只讲述他们所听说的 3 种范式，可能会让他们觉得更舒服些，因为只有这 3 种范式在现实中经常使用。不过，本章也会非常简要地讲述一下其他 3 种范式。

前面已经讲述过如何创建主键，以及在表中使用主键的原因——如果希望能只针对某一行操作，那么需要唯一标识该行。规范化的概念在很大程度上依赖于关于主键的定义和什么列依赖主键的问题。在规范化中常听到的一句话是：

规范化就是键，整个键，只是键。

也有人开玩笑地说：

我向 Codd 发誓，规范化就是键，整个键，只是键。

这是对规范化中第一范式到第三范式最简练的概括。如果所有的列都只依赖于整个键，那就正好说明其处于第三范式。

下面讲述各种范式以及各自的作用。

www.TopSage.com

8.2.1 准备工作

在将数据变成第一范式之前，需要一些先决条件。在将表视为关系数据库意义上的真正实体之前，需要确保下列几个事项。

- 表只应描述一个实体(不要简化或者组合)。
- 所有行必须是唯一的，而且必须有一个主键。
- 列和行的顺序必须是无关的。

首先是要确定有合适的实体。有些实体是很明显的，而有些则不那么明显。在规范化过程中，会提供很多实体并细化实体。至少需要标识所有明显的实体。

提示：

如果熟悉面向对象编程，那么可以将大部分顶层的逻辑实体看作对象模型中的对象。

考虑一个超级简单的模型——销售模型。暂时不需要担心可能的不同变化，或者需要什么列——而只需要标识出系统中的基本实体。

首先考虑最基本的过程。需要做的是为每个原子单元(采用原子单元在该过程中维护数据)创建一个实体。该过程是这样的：一位客户打电话或者直接过来和接受订单的员工交谈。

这个过程可能首先需要一个实体：订单。

提示：

随着对规范化逐步熟悉，类似这样的情况，可能使得您在一开始时就能得到更多的实体。但从现在开始，只考虑这个实体，了解规范化过程如何显示其他需要的内容。

假设在概念上知道了实体将会是怎样的，下一步是指出开始的几个列，从中指出主键。记住，主键为每行提供一个唯一的标识符。

可以仔细考察列的列表，确定候选键。候选键的列表应包括可能用于唯一标识实体中每行的任何列。不过没有严格而快速的方法决定哪个列作为主键(这也是如何设计包含相同基本信息的数据库时很多不同的原因之一)。在有的情况下，甚至可能找不到一个候选键，而需要创建一个(回想一下 Identity 列和 rowguid()列)。

在第 7 章已经创建了 Orders 表，但是作为示例，看看在以前的平面文件设计中 Orders 表的一种很普通的实现，如表 8-1 所示。

表 8-1

Orders
OrderNo
CustomerNo
CustomerName
CustomerAddress
CustomerCity
CustomerState
CustomerZip

www.TopSage.com

(续表)

Orders
OrderDate
ItemsOrdered
Total

因为这是一个 Orders 表，从逻辑上来说，一个订单对应一个订单号，所以选择 OrderNo 作为主键。

这样就有了一个基本实体。这个实体不关注列的顺序(按惯例，表通常将主键作为第一列，但是从技术上来说，这不是必需的)。表的基本组成部分也不关注行的顺序。表至少在表面上只描述一个实体。简单地说，准备开始规范化过程(实际上，已经做到一些规范化了)。

8.2.2　第一范式

第一范式(1NF)全部是关于消除重复数据组和保证原子性(数据是自包含和独立的)的规范化信息。在较高的层次，这指的是创建主键(已经创建了主键)，然后将任何重复的数据组移动到新的表中，为这些表创建新键，如此进行下去。另外，将任何组合数据的列按每部分数据分成不同的行。

在更传统的平面文件设计中，重复数据是很常见的——即在一列中有多条信息——这在很多方面都有问题。

- 在以前，磁盘存储空间非常昂贵。多次存储数据意味着浪费空间。随着数据存储器越来越便宜，这使得重复存储问题不那么显著了。
- 重复数据意味着更多的数据移动，更多的 I/O 数量。这意味着影响了性能，因为这需要经过数据总线或者网络传输大量的数据。即使现在有更快的传输技术，还是对性能有很大负面影响。
- 在行与行之间的重复数据经常出现不一致，产生自相矛盾的数据，而且通常缺乏数据完整性。
- 如果需要查询有组合数据的列的信息，那么必须采用一种方法解析该列的数据(这个速度非常慢)。

现在，表中有很多列，而且很容易添加更多的列。不过好的方面是可以查询到关于 Orders 表的每一个数据。

然而，要更进一步了解问题，需要看看这个表中的一些数据是怎样的，如表 8-2 所示。虽然这里只选择了几列，以便可以在一页内显示，但是这足以说明问题。

如果要规范化这个表，那么需要处理很多问题。虽然这里有一个功能性的主键(早在关系系统出现之前就有了)，但还是碰到了第一范式的两个主要问题。

- 有重复数据组(客户信息)。需要将重复数据组分割到一个不同的表中。
- ItemsOrdered 列包含的不是原子性的数据。

www.TopSage.com

表 8-2

Order No	Order Date	Customer No	Customer Name	Customer Address	Items Ordered
100	1/1/99	54545	ACME Co	1234 lst St.	1A4536, Flange, 7lbs, $75; 4-OR2400, Injector, .5lbs, $108; 4-OR2403, Injector, .5lbs, $116; 1-4I5436, Head, 63lbs, $750
101	1/1/99	12000	Sneed Corp.	555 Main Ave.	1-3X9567, Pump, 5lbs, $62.50
102	1/1/99	66651	ZZZ & Co.	4242 SW 2nd	7-8G9200; Fan, 3lbs, $84; 1-8G5437, Fan, 3lbs, $15; 1-3H6250, Control, 5lbs, $32
103	1/2/99	54545	ACME Co.	1234 lst st.	40-8G9200, Fan, 3lbs, $480; 1-2P5523, Housing, 1lbs, $165; 1-3X9567, Pump, 5lbs, $42

首先将几列从表中移出，如表 8-3 所示。

表 8-3

OrderNo(PK)	OrderDate	CustomerNo	ItemsOrdered
100	1/1/1999	54545	1A4536, Flange, 7lbs, $75; 4-OR2400, Injector, .5lbs, $108; 4-OR2403, Injector, .5lbs, $116; 1-4I5436, Head, 63lbs, $750
101	1/1/1999	12000	1-3X9567, Pump, 5lbs, $62.50
102	1/1/1999	66651	7-8G9200, Fan, 3lbs, $84; 1-8G5437, Fan, 3lbs, $15; 1-3H6250, Control, 5lbs, $32
103	1/2/1999	54545	40-8G9200, Fan, 3lbs, $480; 1-2P5523, Housing, 1lbs, $165; 1-3X9567, Pump, 5lbs, $42

将它们放置到各自的表中，如表 8-4 所示。

表 8-4

CustomerNo(PK)	CustomerName	CustomerAddress
54545	ACME Co	1234 lst St.
12000	Sneed Corp.	555 Main Ave.
66651	ZZZ & Co.	4242 SW 2nd

www.TopSage.com

对于旧表和新表，需要注意以下几个问题。

- 在新表中必须要有主键以确保每行是唯一的。对于 Customers 表来说，有两个候选键——CustomerNo 和 CustomerName。实际上就是为了这个目的才创建 CustomerNo，这从逻辑上看起来是合适的选择——毕竟，很有可能有几个客户同名(例如，在美国有几百个 ACME 公司)。
- 虽然已经将数据移出 Orders 表，但还需要保持对新的 Customers 表中数据的引用。因此还能在 Orders 表中看到 CustomerNo 列(主键)。以后，在创建引用时，需要创建外键约束来强制所有订单具有有效的客户号。
- 可以删除 ACME Co 的一个信息实例，其目的就是移动出现在重复组中的数据——只存储一次。这不仅节省空间，还能防止出现冲突的值。
- 这里只移动重复的数据组。还能看到几次同样的订单日期，但是这实际上不适于放在一个组中——这只是相对随机的数据，在该表以外没有任何关联。

这样，解决了重复数据的问题；下面来解决违反第一范式的另一个问题——原子性。看看 ItemsOrdered 列，可以发现那里实际上有几个不同的数据。

- 一个或多个零件号
- 各个零件的重量信息

零件号、重量和价格各自本身分别是原子数据，但是组合在一起，就不具有原子性。

提示：

不管您相信与否，事情有时就是这样的。初看起来，这似乎很容易做到——纸质的订单上通常只有一大块空白区域供客户填写需要的信息，而基于计算机的系统通常只是模仿使用纸的方式。

继续考虑将数据分开。在分开时，添加一个以单价形式表示的新信息，如图 8-1 所示。问题是，一旦分开了这个信息，主键不再是行的唯一标识符了——行还是唯一的，但是主键不能唯一地标识行了。

Order No (PK)	Order Date	Customer No	Part No	Description	Qty	Unit Price	Total Price	Wt.
100	1/1/1999	54545	1A4536	Flange	5	15	75	6
100	1/1/1999	54545	0R2400	Injector	4	27	108	.5
100	1/1/1999	54545	0R2403	Injector	4	29	116	.5
100	1/1/1999	54545	4I5436	Head	1	750	750	3
101	1/1/1999	12000	3X9567	Pump	1	62.50	62.50	5
102	1/1/1999	66651	8G9200	Fan	7	12	84	3
102	1/1/1999	66651	8G5437	Fan	1	15	15	3
102	1/1/1999	66651	3H6250	Control	1	32	32	5
103	1/2/1999	54545	8G9200	Fan	40	12	480	3
103	1/2/1999	54545	2P5523	Housing	1	165	165	1
103	1/2/1999	54545	3X9567	Pump	1	42	42	5

图　8-1

现在，通过在表中添加行项号来解决这个问题，如图 8-2 所示，这样，就又能唯一地标识每个行了。

www.TopSage.com

Order No (PK)	Line Item (PK)	Order Date	Customer No	Part No	Description	Qty	Unit Price	Total Price	Wt.
100	1	1/1/1999	54545	1A4536	Flange	5	15	75	6
100	2	1/1/1999	54545	0R2400	Injector	4	27	108	.5
100	3	1/1/1999	54545	0R2403	Injector	4	29	116	.5
100	4	1/1/1999	54545	4I5436	Head	1	750	750	3
101	1	1/1/1999	12000	3X9567	Pump	1	62.50	62.50	5
102	1	1/1/1999	66651	8G9200	Fan	7	12	84	3
102	2	1/1/1999	66651	8G5437	Fan	1	15	15	3
102	3	1/1/1999	66651	3H6250	Control	1	32	32	5
103	1	1/2/1999	54545	8G9200	Fan	40	12	480	3
103	2	1/2/1999	54545	2P5523	Housing	1	165	165	1
103	3	1/2/1999	54545	3X9567	Pump	1	42	42	5

图 8-2

注意：

如果不想创建另外一列，那么可以采用的方法是将 PartNo 作为主键的一部分。这样做不好的地方是在一个订单中同样的零件号不能出现两次。第 9 章将简单地讨论基于多列的键——或者称为组合键。

这样，就符合了第一范式的标准。没有重复数据组，所有列是原子的。不过在某些列中还有数据重复的问题(因为相对于主键来说，其所有行的数据都是一样的)，稍后将讲述这个问题。

8.2.3 第二范式

规范化的第二步是达到第二范式(2NF)。第二范式进一步减少重复数据的出现(不一定是数据组)。

第二范式有以下两个规则。

- 表必须符合第一范式的规则(规范化是一个类似堆积木块的过程——如果没有前面两块积木，就不能堆第三块)。
- 每列必须依赖于整个键。

上面的示例在这方面有一个问题——实际上，有很多问题。再次看看第一范式版本的 Orders 表(如图 8-2 所示)。每列都依赖于整个键吗？是否有某个列只依赖于部分的键？

第一个问题的答案是否定的，第二个问题的答案是肯定的。有两列只依赖于 OrderNo 列——而不依赖于 LineItem 列。这两列是 OrderDate 列和 CustomerNo 列；两者对于整个订单来说都是一样的，不管有多少行项。为了处理这个问题，引入另外一个表。这里，第一次接触到了表头和细节表的概念。

有时在实际中，一个实体还需要分割成两个表，这样就有两个实体。表头是两个有关系的表中的父表的一部分。表头包含的信息只需要存储一次，而细节表存储可能有多个实例的信息。表头通常保持原表的名称，细节表的名称通常以表头名称开头，添加一些表明细节表的信息(例如，OrderDetails)。对每个表头记录，通常至少有一个细节记录，也可以有多个。这是某种关系(一对多关系)的一个示例，将在 8.3 节中讲述。

下面再次将表分割，如表 8-5 所示。先从细节表开始，因为细节表包含了大部分列。从这里开始，称该表为 OrderDetails。

www.TopSage.com

表　8-5

OrderNo (PK)	LineItem (PK)	PartNo	Description	Qty	Unit Price	Total Price	Wt
100	1	1A4536	Flange	5	15	75	6
100	2	OR2400	Injector	4	27	108	.5
100	3	OR2403	Injector	4	29	116	.5
100	4	4I5436	Head	1	750	750	3
101	1	3X9567	Pump	1	62.50	62.50	5
102	1	8G9200	Fan	7	12	84	3
102	2	8G5437	Fan	1	15	15	3
102	3	3H6250	Control	1	32	32	5
103	1	8G9200	Fan	40	12	480	3
103	2	2P5523	Housing	1	165	165	1
103	3	3X9567	Pump	1	42	42	5

然后看看表头(尽管您可能认为该表两个表中的是新表)，如表 8-6 所示，这里仍用 Orders 作表名。

表　8-6

OrderNo(PK)	OrderDate	CustomerNo
100	1/1/1999	54545
101	1/1/1999	12000
102	1/1/1999	66651
103	1/2/1999	54545

因此，现在就有了第二范式。所有列都依赖于整个键。正如您所猜测的，还存在一两个问题，这将在 8.2.4 节讨论。

8.2.4　第三范式

这只是规范化方面的一个相对终点。从技术上讲，也有超过这个层次的规范化技术层次，但是只是在学术圈外得到比较多的关注。8.2.5 节将简要介绍，不过首先要解决当前的问题。

在讨论完第二范式之后，提到还存在一些问题——即还没有达到第三范式(3NF)。第三范式处理的问题是使得表中所有列不仅仅是依赖于某个事物——而是要依赖于正确的事物。第三范式有以下三个规则。

- 表必须是符合 2NF(前面提到过，这个类似堆积木)。
- 任何列都不能依赖于非键列。
- 不可以有派生的数据。

www.TopSage.com

现在已经符合第二范式，下面来看看后面两个规则。

首先，看看是否有某些列依赖于主键以外的列？是的，实际上，表中有很多列依赖于 PartNo，即使没有依赖于主键的列那么多，也相差不多。Weight 和 Description 是完全依赖于 PartNo 列的——再次需要分割出另外的一个表。

提示：

您很可能会将 UnitPrice 归为这一类，但这并不完全正确。将创建的 Products 表可以并且应该有一个 UnitPrice 列——但是在本质上有些不同。实际上，可能命名为 ListPrice 更合适，因为这通常设置为那个产品的价钱。OrderDetails 表中 UnitPrice 的不同有两重意思。首先，可能提供折扣，这样在某次销售时改变制定的价格。这意味着在 OrderDetails 记录中的价格可能和在 Products 表中制定的价格不同。其次，制定的价格将随时间不同而改变，如通货膨胀，但是未来价格的改变不会改变过去接受的实际订单的价格。换句话说，价格所指实际上有两种可能——一个依赖于 PartNo，一个依赖于 OrderDetails 表的主键(即 OrderID 和 LineItem)。

首先，需要创建一个新的表(称为 Products)来容纳部分信息。新表包含 OrderDetails 中更依赖于 PartNo 而不依赖于 OrderID 或者 LineItem 的信息，如表 8-7 所示。

表 8-7

PartNo(PK)	Description	Wt
1A4536	Flange	6
OR2400	Injector	.5
OR2403	Injector	.5
4I5436	Head	3
3X9567	Pump	5
8G9200	Fan	3
8G5437	Fan	3
3H6250	Control	5
8G9200	Fan	3
2P5523	Housing	1
3X9567	Pump	5

然后将除外键外的产品数据从 OrderDetails 表中分出去，如表 8-8 所示。

表 8-8

OrderNo (PK)	LineItem (PK)	PartNo	Qty	Unit Price	Total Price
100	1	1A4536	5	15	75
100	2	OR2400	4	27	108
100	3	OR2403	4	29	116

www.TopSage.com

(续表)

OrderNo (PK)	LineItem (PK)	PartNo	Qty	Unit Price	Total Price
100	4	4I5436	1	750	750
101	1	3X9567	1	62.50	62.50
102	1	8G9200	7	12	84
102	2	8G5437	1	15	15
102	3	3H6250	1	32	32
103	1	8G9200	40	12	480
103	2	2P5523	1	165	165
103	3	3X9567	1	42	42

这处理了第 1 个问题(交叉列依赖)，但是没有处理派生数据的问题。这里有一个名为 TotalPrice 的列，包含可以直接通过 Qty 和 UnitPrice 相乘得到的数据。这在规范化中是不允许的。

注意：

我经常会对派生数据进行“反规范化”。为什么呢？为了速度！读入 WHERE TotalPrice>$100 的查询比读入 WHERE Qty*UnitPrice>50 的查询快得多，特别是在可以对计算得到的 TotalPrice 进行索引时更是如此。

另一方面，更经常采用混合方法，即通过使用计算列和让 SQL Server 计算其他两列和的方法(可以查看第 5 章在 Accounting 数据库的 Employees 表中使用计算列求 PreviousSalary 的示例)。如果从性能上看这是个非常重要的列(可能很多列需要基于该列的值进行过滤)，那么可能需要添加索引到计算列。这样，索引就“物化”了计算的数据。这是什么意思呢？这意味着 SQL Server 不必动态地对计算列进行计算——而是只在行存储到索引中时计算一次，之后，使用预先计算得到的列。这实际上速度非常快，本书将在第 9 章进一步讲述。不过，那也意味着要付出一定代价——即空间。因为这是在存储本不需要存储的数据，如果对与派生数据相关的所有数据都这样处理，那占用的空间会陡然增加。更多的空间意味着要读取更多数据，也意味着处理速度变慢。重点还是要权衡自己的选择。

因此，要达到第三范式，只需要删除 TotalPrice 列，在需要的时候进行计算。

8.2.5　其他范式

规范化模型中还有其他一些范式(至少在学术界是这么认为的)。它们包括以下范式。

- Boyce-Codd(这实际上是第三范式的变体)：这个范式试图解决有多个重叠候选键的情况。这只可能在下列条件下发生：
  (a) 所有候选键是组合键(即键由多个列组成)。
  (b) 不止一个候选键。
  (c) 每个候选键至少有一个列和另外一个候选键的列相同。

这种情况可以有任意多的解决方案，而且几乎从来没有在学术界以外得到逻辑上的考虑。

www.TopSage.com

- **第四范式**：这个范式试图解决多值依赖问题。具体情况是：在单个行中，没有列依赖于主键以外的列，而且只依赖于整个主键(符合第三范式)。然而，还有更奇怪的情况：主键中的一列可能分别依赖于主键中的其他列。这种情况非常罕见，通常不会引起实际的问题。因此，在数据库领域中基本被忽略了，这里也不讨论这个问题。
- **第五范式**：处理无损和有损分解。实际存在着这样的特定的情况：虽然可以分解一个关系，但不能从逻辑上重新构成到原来的形式。这也是非常罕见的，很大程度上是学术问题，这里不再讨论。

当然，这里只是作一个简单了解——也是作者的本意。主要原因是考虑到读者在实际中需要了解这些，或者是为了给您的朋友留下好的印象(证明您“知道所有范式”)，或者为了不在数据库大师讨论这个问题时显得很无知。

8.3 关系

有专家说过，建立成功关系的关键是知道双方的角色，而且人人知道各自所处关系的边界和规则。这里讲的数据库关系与人之间的关系类似。

有以下 3 种主要的关系。

- 一对一
- 一对多
- 多对多

每种关系都有一些变体，这取决于关系的一方是否可以为空。例如，可能有零或者一对一关系而不只是一对一关系。

8.3.1 一对一关系

正如名称所示，一对一关系指如果在一个表中有一个记录，那在另一表中也会有一个与之匹配的记录。

要说明一对一关系，看一下前面的一个示例，不过会有些微小变化。假设您有一些客户——和前面示例一样。不过这次假设是一个大公司的一个子公司。父公司希望跟踪所有客户，要求知道每个客户的总订单——不管客户是从哪个子公司购买的订单。

即使所有子公司都运行总部的一台服务器，不同子公司还是很有可能运行自身的数据库。一种跟踪所有客户信息的方法是创建一个父公司所有的主客户数据库，将各个子公司的信息组合起来。每个子公司维护自身的客户表，但是和父公司的客户表是一对一关系。在父公司创建的任何一个客户记录也需要在子公司有一个对应的记录。任何一个子公司创建的客户记录也需要在父公司里创建一个备份。

另外一个示例是——在 SQL Server 7.0 版本以前很常用的——在一行中放不下很多信息的情况。要知道 SQL Server 中，对于非 BLOB 数据，允许每行最多 8060 字节。这与 6.5 版本只允许 1962 字节相比好多了。但是很可能碰到这样的情况：要存储大量的列或者虽然列比较少，但是每列长度很大。解决这个问题的一个方法是创建两个不同的表，将记录分开在两个表中存储。这样就有了一对一关系。两个表中匹配行的组合就可以满足较大的行的存储需求。

www.TopSage.com

提示：

SQL Server 本身没有方法来强制真正的一对一关系。可能表 A 需要在表 B 中有一个匹配的记录，但是表 B 需要在表 A 中有一个匹配的记录，这样就出现了矛盾——哪个表首先获得记录呢？如果需要在 SQL Server 中强制这种关系，最好的办法是所有的插入都通过一个存储过程来完成。存储过程具有这样的逻辑：要么在两个表中都插入数据，要么两个表都不插入数据。外键约束和触发器都不能处理这个循环关系。

零或一对一关系

SQL Server 可以处理零或一对一关系的实例。这在本质上和一对一相同，不同的是关系的一方可以选择有一个记录或者没有记录。

回到父公司和子公司的示例中，您可能更愿意创建这样一个关系，即父公司需要有与每个子公司中的记录相匹配的记录，但是子公司不需要父公司的信息。例如，子公司可以有不同的客户(例如铁路和建筑公司)。父公司需要知道所有客户，而不管其来自哪个子公司，但是建筑公司可能并不关心铁路客户。在这种情况下，需要零或一个建筑客户对应一个父公司的客户记录。

在 SQL Server 中，可以通过下列一些方式强制零或一对一关系。

- 唯一键约束或者主键约束和外键约束组合。外键约束可以实现在“一”(示例中的父公司)表中必须至少存在一个记录，但是不能确保只存在一个(可能有多个记录)。使用主键约束或者唯一键约束能确保只有一个记录。
- 触发器。注意在两个表中都需要触发器。

注意：

SQL Server 可以处理零或一对一关系，而不能处理一对一关系的原因是难以解决“哪个记录先出现”的问题。在真正的一对一关系中，不能插入任何一个另外一个表中尚不存在的记录——这是个矛盾。然而，对于零或一对一关系，可以先在必需的表(即其中的“一”)中插入数据，如果需要的话，然后在可选的表(即其中的“零或一”)中插入。这个问题同样存在于“一对一或者一对多”和“一对零或者一对一或者一对多”关系中。

8.3.2　一对一或一对多关系

这是一种常见的外键关系，通常以表头/细节表关系形式出现。一个很好的示例是 Orders 表的情况，如图 8-3 所示。OrderDetails(关系的一方或者多方)在不属于 Orders 表头时就没有意义了(如果有一个零件的订单不知道是谁订购的，就没有什么意义了)。同样地，如果没有实际订购的物品，那么订单也是没有意义的(例如，“看看，ACME 公司昨天什么都没有订购”)。

然而这碰到与一对一关系同样的基本问题。还是那个先有鸡还是先有蛋的问题——哪个记录先出现呢？在 SQL Server 中，解决这个问题的唯一方法是通过存储过程限制所有数据的插入或删除。

www.TopSage.com

Order No(PK)	Order Date	Customer No
100	1/1/1999	54545
101	1/1/1999	12000
102	1/1/1999	66651
103	1/1/1999	54545

Order No(PK)	Line Item(PK)	Part No	Qty	Unit Price	Total Price
100	1	1A4536	5	15	75
100	2	0R2400	4	27	108
100	3	0R2403	4	29	116
100	4	4I5436	1	750	750
101	1	3X9567	1	62.50	62.50
102	1	8G9200	7	12	84
102	2	8G5437	1	15	15
102	3	3H6250	1	32	32
103	1	8G9200	40	12	480
103	2	2P5523	1	165	165
103	3	3X9567	1	42	42

图 8-3

一对零、一对一或一对多关系

这是另外一个在日常生活中更常见的外键关系。在实现中唯一的区别在于引用字段(在有外键约束的表中)允许为空；即实际上在“一”表中有一个记录，而这在引用表中未必有任何匹配的记录。

假设有一个这样的场景：要跟踪已发出的订单的发货人，同时还有一个客户自行取货的通话记录。如果有发货人，那么希望将它限定在所批准的发货人列表内，但也可能根本没有发货人，如图 8-4 所示。

Shipper ID	Company Name	Phone
1	Speedy Express	(503) 5559831
2	United Package	(503)5553199
3	Federal Shipping	(503) 5559931

Order ID	Order Date	Customer ID	Shipper ID	Ship To Address
10500	1997-04-09	LAMAI	1	1 rue Alsace-Lorraine
10501	1997-04-09	BLAUS	3	Forsterstr 57
10502	1997-04-10	PERIC	1	Calle Dr. Jorge...

图 8-4

www.TopSage.com

在 AdventureWorks2008 数据库的 Purchasing.PurchaseOrderHeader 和 Purchasing.ShipMethod 之间的关系中可以发现一个几乎相同的示例，唯一的区别是公司列表列出的是发货给我们的公司而不是从我们这边发货的公司。

注意：

这种关系通常构成所谓的域关系。一个域是一个受限制的值列表，依赖表必须从中选择——域列表以外的值都认为是无效的选项。包含构成域列表的行的表通常称为域表或者查找表。几乎所有创建的数据库都至少有一个域表，也可能有多个域表。这里的 Shippers 表是一个域表——需要这个表的目的不仅是存储发货人的姓名和电话号码信息，还限制了 Orders 表中可能的发货人列表。

在 SQL Server 中，可以通过下面两个方法强制关系。

- 外键约束：只需要在作为关系中“多”方的表上声明外键约束，然后引用作为关系中“一”方的表和列(可以确保在被引用表中只有一方，因为被外键引用的列必须要有主键约束或者唯一约束)。
- 触发器：实际上，在所有早期 SQL Server 版本中，这个是真正实现参照完整性的唯一选择。实际需要添加两个触发器——关系的每方各一个。在作为关系中“多”方的表中添加触发器，用于检查在该表中任何插入或者修改的行在被依赖表(关系中的“一”方)中有一个匹配的行。然后可以在另外一个表中添加一个删除触发器或更新触发器——这个触发器检查从被引用表中删除(或更改)的记录，以确保不出现孤立记录(也就是使得表没有引用)。

前面在第 6 章中已经讨论过针对性能的选择问题。使用外键约束通常更快些——特别是在有违反范式的情况下。那就是说，在需要触发器执行时(或者需要其他特殊约束时)，触发器还可能是更好的选择。

8.3.3　多对多关系

在这种关系中，关系的两方都可以有多个记录(不只是一个)匹配。例如产品和订单的关系。一个特定的订单可能有很多不同的产品。同样地，任何一个特定的产品可以订购很多次。然而，可能还是希望关联这两个表——例如，来确保订单是订购已知的产品(在 Products 表中)。

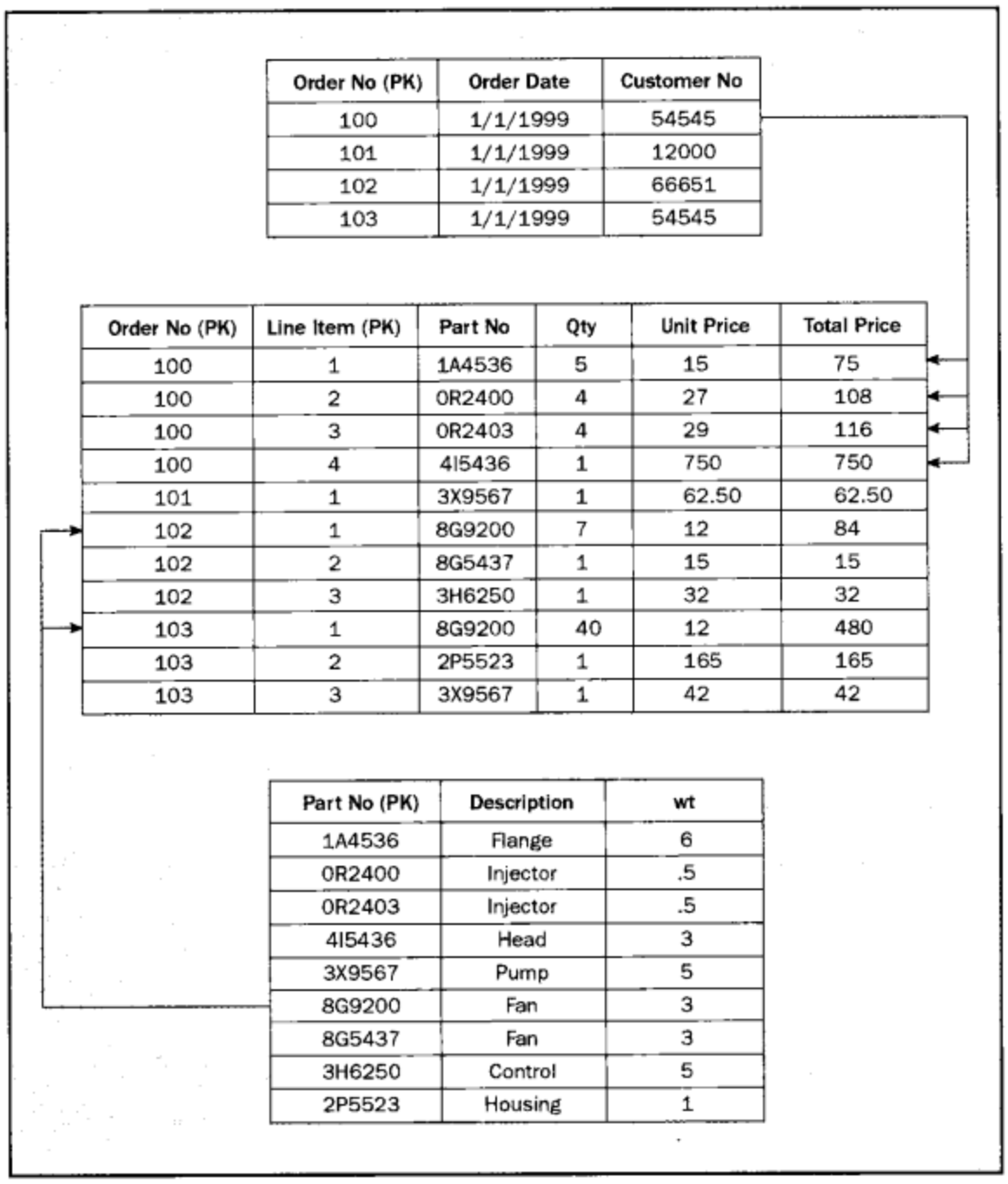
由于 SQL Server 没有办法在物理上建立直接的多对多关系，因此通过中间表来组织关系。一些表是在正常的规范化过程中偶然地创建了多对多关系——其他一些则是从零开始创建，完全只是为了建立这种关系。后者中的“中间表”通常称为链接表或者关联表，有时也称为合并表。

首先介绍正常的规范化过程中创建的多对多关系。在 Accounting 数据库(在第 5 章和第 6 章创建的)的 OrderDetails 表中可以看到一个示例，该示例创建了 Orders 表和 Products 表之间的多对多关系，如图 8-5 所示。

通过使用第 4 章讲述的连接语法，可以将一个产品关联到多个订单，该产品是这些订单中的一种产品；也可以通过另外的方式，将一个订单关联到订单中的所有产品。

继续来看第二个示例——从头开始创建关联表来获得多对多关系。下面采用一个用户与一组用户对系统的权限的示例来说明。

www.TopSage.com



Order No (PK)	Order Date	Customer No
100	1/1/1999	54545
101	1/1/1999	12000
102	1/1/1999	66651
103	1/1/1999	54545

Order No (PK)	Line Item (PK)	Part No	Qty	Unit Price	Total Price
100	1	1A4536	5	15	75
100	2	0R2400	4	27	108
100	3	0R2403	4	29	116
100	4	4I5436	1	750	750
101	1	3X9567	1	62.50	62.50
102	1	8G9200	7	12	84
102	2	8G5437	1	15	15
102	3	3H6250	1	32	32
103	1	8G9200	40	12	480
103	2	2P5523	1	165	165
103	3	3X9567	1	42	42

Part No (PK)	Description	wt
1A4536	Flange	6
0R2400	Injector	.5
0R2403	Injector	.5
4I5436	Head	3
3X9567	Pump	5
8G9200	Fan	3
8G5437	Fan	3
3H6250	Control	5
2P5523	Housing	1

图 8-5

首先看一下 Permissions 表，如表 8-9 所示。

表 8-9

PermissionID	Description
1	Read
2	Insert
3	Update
4	Delete

然后添加一个 Users 表，如表 8-10 所示。

表 8-10

UserID	Full Name	Password	Active
JohnD	John Doe	Jfz9..nm3	1
SamS	Sam Spade	klk93)md	1

www.TopSage.com

现在问题出现了——如何定义哪些用户具有哪些权限呢？可能首先倾向于添加名为 Permissions 的列到 Users 表中，如表 8-11 所示。

表　8-11

UserID	Full Name	Password	Permissions	Active
JohnD	John Doe	Jfz9..nm3	1	1
SamS	Sam Spade	klk93)md	3	1

这初看起来是可以的，不过需要解决一个问题——当用户有多个权限，该怎么办呢？

在以前的平面文件中，可能将所有权限组合到一个单元，如表 8-12 所示。

表　8-12

UserID	Full Name	Password	Permissions	Active
JohnD	John Doe	Jfz9..nm3	1,2,3	1
SamS	Sam Spade	klk93)md	1,2,3,43	1

这违反了第一范式，第一范式规定任何列的值必须具有原子性。另外，这可能非常慢，因为需要从程序上解析单元的各个值。

Users 表和 Permissions 表之间实际上是多对多的关系——需要一种方法在数据库中建立这种关系。可以通过添加一个关联表的方法来解决，如图 8-6 所示。在很多情况下，该表不添加任何新的数据到数据库中，只是将其他两个表的记录关联起来。

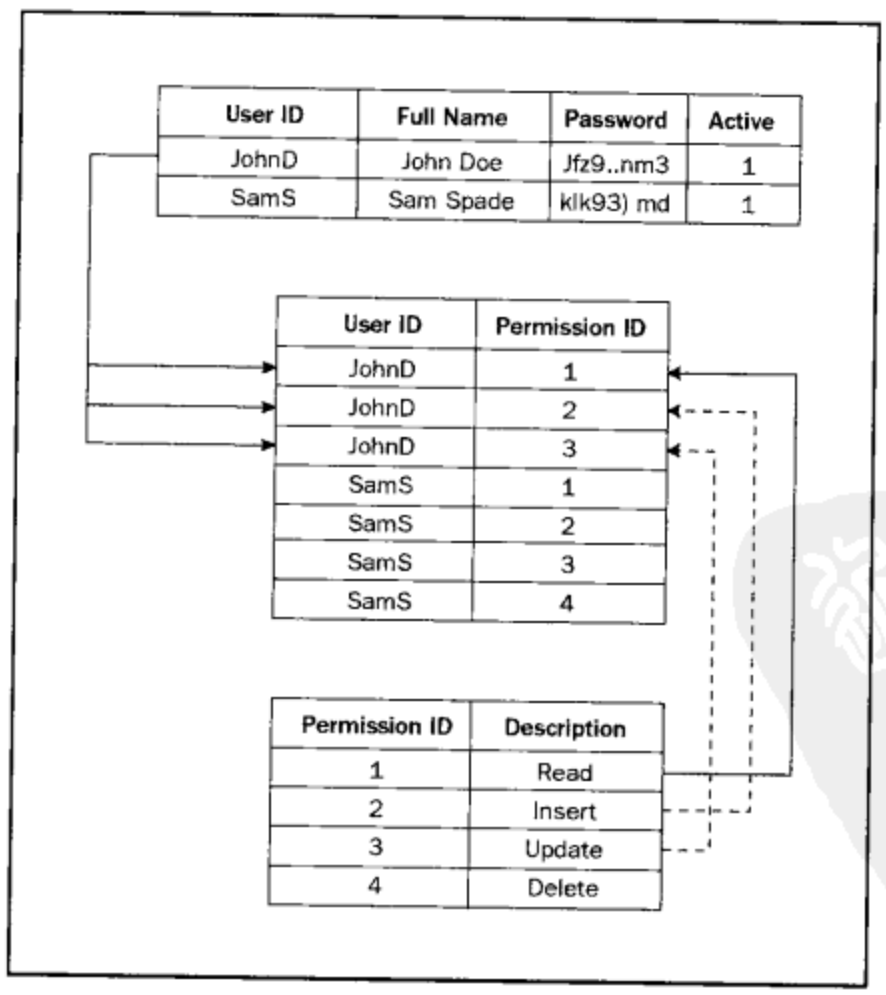


图　8-6

www.TopSage.com

通过添加新表(这里称为 UserPermissions)，可以混合和匹配权限和用户。

注意到，在这两个示例中，参照完整性的实现是一样的——每个基表(包含基本的数据，有多对多关系)和关联表有一对多的关系。这可以通过触发器或者外键约束来实现。

8.4 数据库关系图

实体关系数据库关系图(ERD)是实现良好的数据库设计的重要工具。较小的数据库通常可以很容易直接通过一些脚本创建和实现，根本不需要画什么关系数据库关系图。然而，当数据库越来越大时，把“内容放在脑子里”的问题会越来越多。ERD 解决了不少问题，因为通过 ERD 可以快速可视化和理解实体还有实体之间的关系。

幸运的是，SQL Server 提供一个很基本的数据库关系图工具，可以用于开始构建基本的 ERD。

提示：

在作者首次撰写这个主题时，花了很长时间来讨论如何处理这个问题。一方面，严格的 ER 图通常是通过专门的 ER 图工具实现的。这些工具几乎总是至少支持多种工业标准数据库关系图方法中的一种。一些更大众化的数据库关系图工具(如 Visio)也支持很多 ERD 方法。SQL Server 内置了 ERD 工具，这就是问题所在。SQL Server 2008 包含的工具是 Microsoft 多年来使用的工具集和数据库关系图方法的变体。问题是这些工具达不到在任何其他地方所看到的 ERD 标准。每次写到这个主题时，都采用您所拥有的内置工具进行讲解。不过，也鼓励您了解商业的 ERD 工具，看看它们所提供的简化数据库设计的丰富内容。

要打开 SQL Server 的内置工具，可以找到希望为其构建数据库关系图的数据库的数据库关系图节点(先展开服务器，然后展开数据库)。下面看到的一些内容是很熟悉的——一些对话框和在第 5 章创建表时看到的一样。

SQL Server 数据库关系图工具没有那么多选项，所以可以很快熟悉。实际上，如果熟悉 Access 中的关系编辑器，那么对于很多 SQL Server 的工具来会陌生。

试一试　画数据库关系图

开始创建第一个数据库关系图。要创建新的数据库关系图，可以右击 AdventureWorks2008 数据库下的“数据库关系图”节点，选择“新建数据库关系图”选项。

提示：

回顾第 5 章，(如果是第一次创建数据库关系图的话)可以看到一个对话框警告说此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。是否要创建它们——选择“是”按钮。

和第 5 章看到的一样，SQL Server 首先弹出“添加表”对话框(如图 8-7 所示)——只是列出的表不同。

选择所有的表(记住按住 Ctrl 键选择多个表)，如图 8-8 所示。

www.TopSage.com

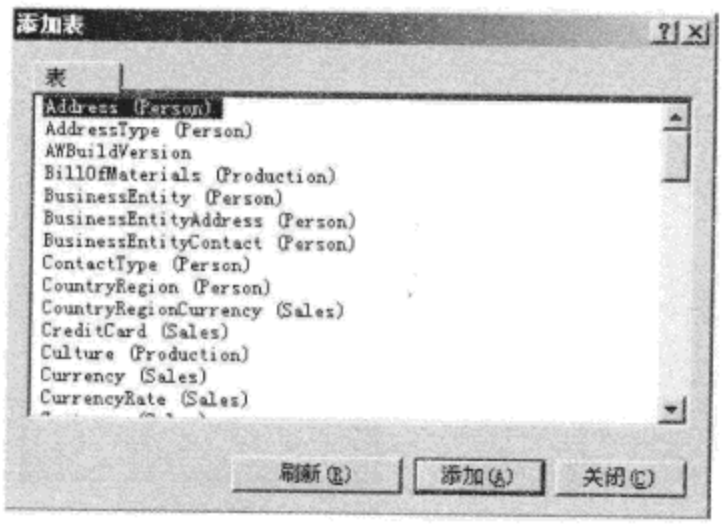


图　8-7

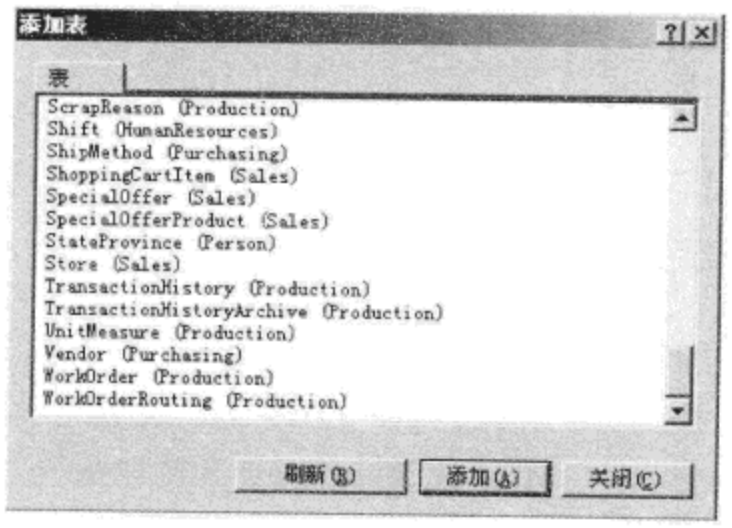


图　8-8

然后单击“添加”按钮，稍后，SQL Server 画出选择的所有表，然后单击“关闭”按钮。SQL Server 已经将表添加到数据库关系图中，但是根据屏幕分辨率不同，由于数据库关系图的缩放尺寸不同，可能看不到所有表。要看到更多的表，可以改变工具栏上的缩放比例设置。如何在一次能看到尽量多的表和每个表都能看清楚之间找到平衡点是个问题，但是可以按您的偏好作调整——现在将其设置为 75%，这样可以一次看到所有的表(通常实际的数据库中表的尺寸不是正好这么大)，如图 8-9 所示。

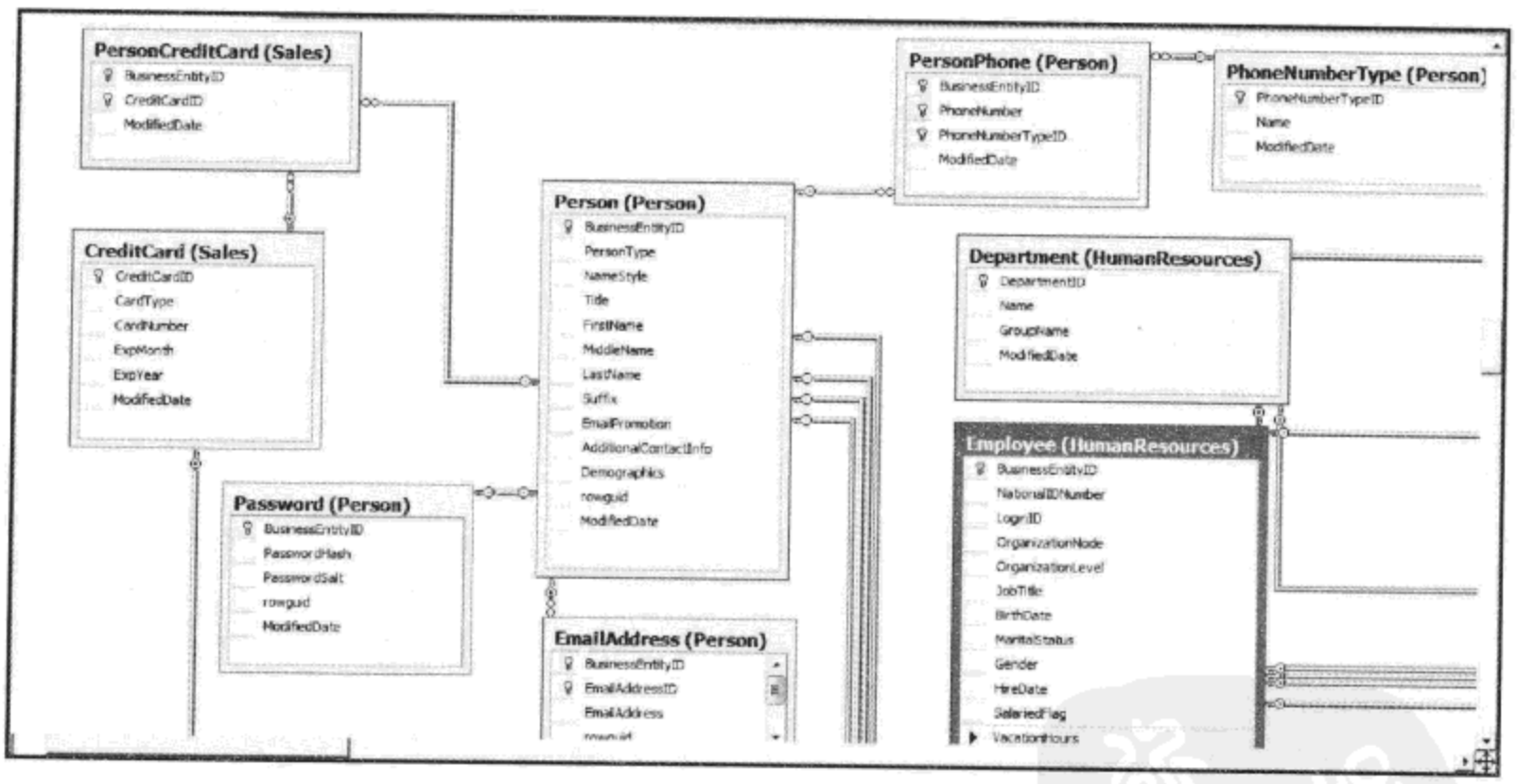


图　8-9

示例说明

您可以很快发现，这里的内容比在第 5 章的数据库关系图工具中看到的丰富很多。SQL Server 列举了希望添加和分析的每个表，以此确定与这些表相关的其他对象。除了表自身之外的其他各项是对表约束的一些对象——即主键和外键。

因此，现在使用这个数据库关系图作为一个起点，来解释数据库关系图工具是如何工作的，然后不时地构建一些表。

www.TopSage.com

8.4.1 表

每个表有自身的窗口，可以用鼠标移动。主键在名称左边的列中使用一个小钥匙的符号来显示，就像图 8-10 中 CustomerID 旁边的那个钥匙符号。

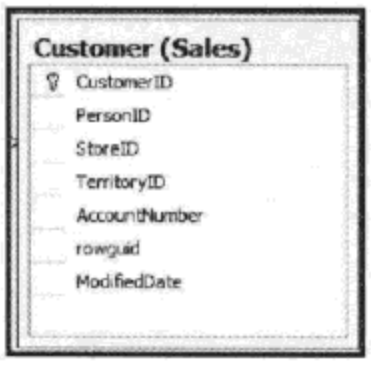


图 8-10

和第 5 章一样，这只是表的默认视图——可以选择其他几种视图来编辑表的各个组成部分。要检查表的视图选项，右击需要查看的表。默认情况下只有列名，但是可以像在第 5 章那样选择 Custom；Custom 或 Standard 选项是在数据库关系图中编辑表时需要使用的。

8.4.2 添加和删除表

可以有一两种方法来添加新的表到数据库关系图中。

如果在数据库中已经存在一个表(但是不在数据库关系图中)，但是现在又想将该表添加到数据库关系图中，则只需要单击在数据库关系图窗口中工具栏的“添加表”按钮。在数据库中会显示所有表的列表——直接选择需要添加的表，这样就会在数据库关系图中显示该表，以及该表和其他表的关系。

如果需要添加一个全新的表，单击数据库关系图窗口工具栏的“新建表”按钮，或者右击数据库关系图，然后选择“新建表”，系统就会提示输入新表的名称，之后表会添加到“列属性”视图的数据库关系图中。直接编辑需要的列名称、数据类型等属性，这样在数据库中就有了新表。

提示：

在这个过程中需要注意一些问题。

首先，别忘在表中添加主键。SQL Server 不会自动添加主键，也不会进行提示(Access 会提示)。要添加主键，必须选择键中需要包含的列。然后右击并从快捷菜单中选择“设置主键”命令。

其次，要明白只有在选择保存后，新表才真正被添加到数据库中——在编辑表时也是如此。

试一试　在数据库关系图中添加表

接下来在数据库中添加一个表以了解其工作原理。

在“数据库关系图”窗口的工具栏中单击“新建表”按钮。在提示输入名称之后，输入名称 CustomerNotes(如图 8-11 所示)。使用标准视图就能看到一个新的表窗口。

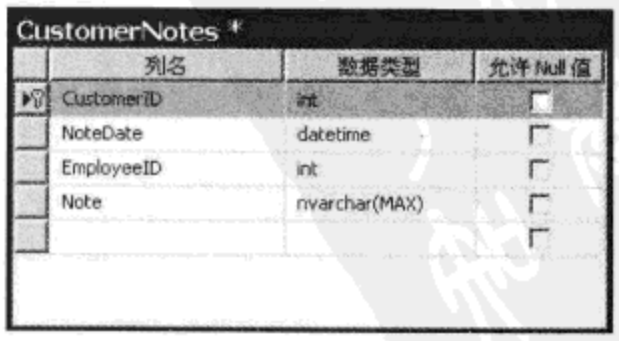


图 8-11

可注意到，这里已经添加了几列和一个主键(记住，选择希望作为主键的列，然后右击并从快捷菜单中选择“设置主键”)。在单击进行保存之前，再试一

www.TopSage.com

试一些功能——打开 Menagement Studio，对新表运行一个查询：

```
USE AdventureWorks 2008;

SELECT * FROM CustomerNotes;
```

返回一个错误消息：

```
消息 208，级别 16，状态 1，第 1 行
无效对象名 'CustomerNotes'。
```

这是因为表只是以数据库关系图中一个编辑的形式存在——只有在保存改动之后才真正添加。

注意：

如果这时查看“数据库关系图”窗口中的 CustomerNotes 表，可以看到名称的右边有一个“”——这说明该表中有未保存的改动。*

现在切换回到 Management Studio。有以下两个保存选项。

- “保存”：这个选项保存对数据库关系图和数据库的改动(这是工具栏上一个小的磁盘图标)。
- “保存更改脚本”：这个选项保存脚本的改动，可以在以后运行保存的脚本(该选项在“表设计器”菜单中，而在“表设计器”工具栏上以图标显示)。

提示：

“表设计器”工具栏包含了一些智能图标，可用于快速访问一些常用的表设计功能。遗憾的是，在 SQL Server 2008 的 RTM 版本中，它未默认出现在设计器中。可以右击工具栏上的空白区域并选择“表设计器”工具栏添加它。

选择“保存”选项，系统弹出确认信息(毕竟，将要改变数据库——对此没有“取消”的方法)。

确认所作的改动，试着重新对 CustomerNotes 表运行查询。这次不会接收错误消息，因为表已经创建了(不会返回任何行，但是查询运行正常)。

示例说明

在创建一个数据库关系图时，SQL Server 在后台创建脚本，基本上和第 6 章脚本化更改时所做的一样。不过，这些脚本直到选择保存数据库关系图时才会真正生成和运行。

这样在数据库中添加了 CustomerNotes 表，但是要注意一个问题——声明主键的方式，一位客户只能有一个备注。更可能的情况是，随着时间推移，需要给客户添加越来越多的备注。这意味着需要改变主键，而根据需求可以有以下几种选择。

- **将日期作为主键的一部分**：这在两个方面存在问题。首先，如果要跟踪哪个雇员作的备注——那么要是不同的雇员同时想添加备注怎么办呢？当然可以添加 EmployeeID 到主键。其次，如果同一个雇员不希望在同一天输入完全分开的备注，那该怎么办(因为

www.TopSage.com

这是 datetime 字段，只要不在同一微秒中插入两行就可以分别输入——不过这里该怎么处理)？这样即使在主键中包含 EmployeeID 也没有用。

- **添加另外一列作为键的组成结构**：可以为每个客户每个备注添加计数列。另外一个方法是添加一个标识列来保证唯一性——这意味着主键和任何列无关，但也并不麻烦(虽然意味着需要维护另外一个索引)，并且允许给每个客户作相对无限的备注。

这里采用向表中添加 Sequence 列的方法。按惯例(不是要求人人都这么做)，主键通常是表中的第一列。如果需要，可以使用脚本来做，可能采用 ALTER TABLE 语句和 ADD 列——这样将新列放置到列表的最后。如果需要改正，那么必须将所有数据复制到一个表中，删除原先表中的任何关系，删除原来的表，创建一个具有所需的列和列顺序的新表，重新建立关系，将数据复制回来(这是冗长而繁琐的过程)。然而，使用数据库关系图工具，SQL Server 可处理所有这些问题。

要在表中间插入新的行，右击需要插入位置的后一行并选择“插入列”命令。这个工具将提供插入位置，如图 8-12 所示。

可以插入新的列，重新设置主键，如图 8-13 所示(选择两行，右击并从快捷菜单中选择“设置主键”)命令。

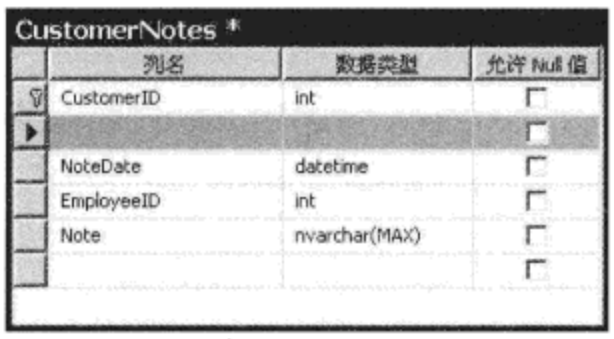


图 8-12

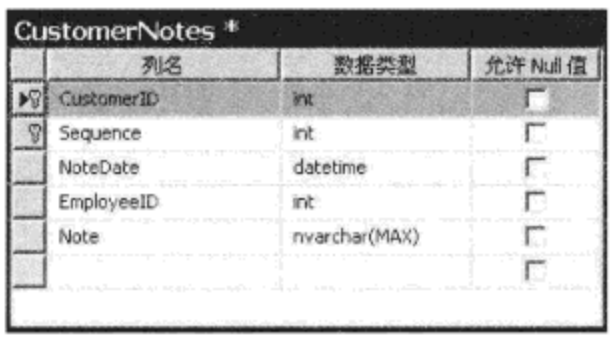


图 8-13

只要一保存，就得到期望的列顺序的表了。要验证这一点，可以使用 sp_help：

```
EXEC sp_help CustomerNotes
```

可以看到列顺序的确是所希望的那样：

```
....
CustomerID
Sequence
NoteDate
EmployeeID
Note
....
```

提示：

根据特定的选项设置，在保存 CustomerNotes 表时可能遇到一个错误。问题出在 SQL Server 2008 新提供的安全功能上，它能够防止在设计器中意外地覆盖表。如果错误信息中包含“阻止保存要求重新创建表的更改”，那可以在“工具”菜单的“选项”对话框中找到它——需要导航至该对话框中的“表设计器”|“数据库设计器”选项，然后取消选择“阻止保存要求重新创建表的更改”选项。之后表就可以顺利保存。

www.TopSage.com

注意：

像作列顺序调整这类的改动正好是内置工具所擅长的领域。作者使用过很多其他的 ERD 工具，它们也都能在数据库和数据库关系图之间同步列顺序的改变——但其成功纯属偶然(换句话说，它们根据活动数据做了大量细致工作)。

还有另外一个问题——如果是像操作活动数据一样进行改动，那么使用脚本选项而不是活动连接到数据库来进行改动。这样可以在使用实际数据之前，先充分测试脚本而不是测试数据库。在作这些改动之前，确保完全备份数据库。

1. 编辑表属性和表中的对象

除了前述的基本属性之外，还可以编辑表的其他很多方面。有两种不同的方法来编辑或添加它们。

- **属性**：可在一个弹出窗口中编辑，该窗口默认停靠在“数据库关系图”窗口中 Management Studio 的右边。要打开“属性”窗口，单击 Management Studio 工具栏的“属性窗口”图标。
- **属于表的对象，如索引、约束和关系**：可在它们各自的对话框中编辑。可以右击数据库关系图中的表，选择需要设置的条目，打开相应的对话框。

基于数据库关系图的编辑有很多重要的方面，下面来看看主要的方面。

2. “属性”窗口

图 8-14 显示了 CustomerNotes 表的“属性”窗口。

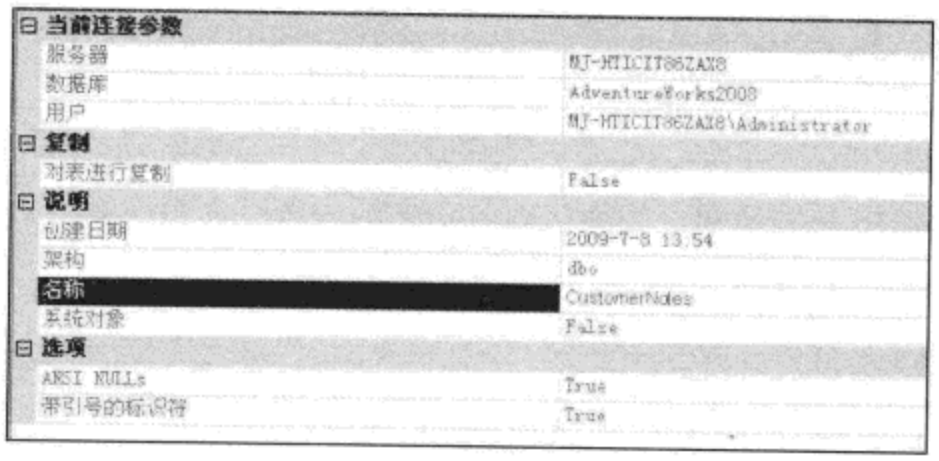


图　8-14

可以使用这个属性窗口设置几个关键的表属性——最明显的是表所属的模式，以及表是否有标识列。

3. 关系

可以右击表头并选择“关系”或是选择“表设计器”工具栏中的“关系”图标打开该对话框。顾名思义，这个对话框可用于编辑表之间的关系。从图 8-15 可以看到，CustomerNotes 表的关系中还没有任何内容。

现在，要认识到在这里可以编辑有关关系的任何内容。例如，可以创建与另外一个表的关系，只需要单击“添加”按钮，填写弹出的各个输入框。这将在稍后再讲述。

www.TopSage.com

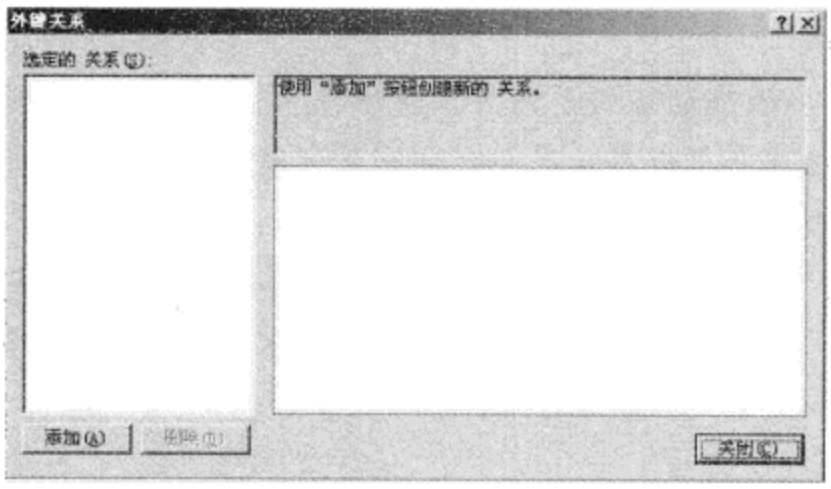


图 8-15

4. 索引/键

这里的很多内容对您来说还是很神秘的——因为还没有学习有关索引的章节，所以有些术语有点陌生。还是来看看图 8-16 中有哪些内容(同样，打开的方法是可以右击表头并选择“索引/键”或是选择“表设计器”工具栏中的“管理索引和键”图标)。

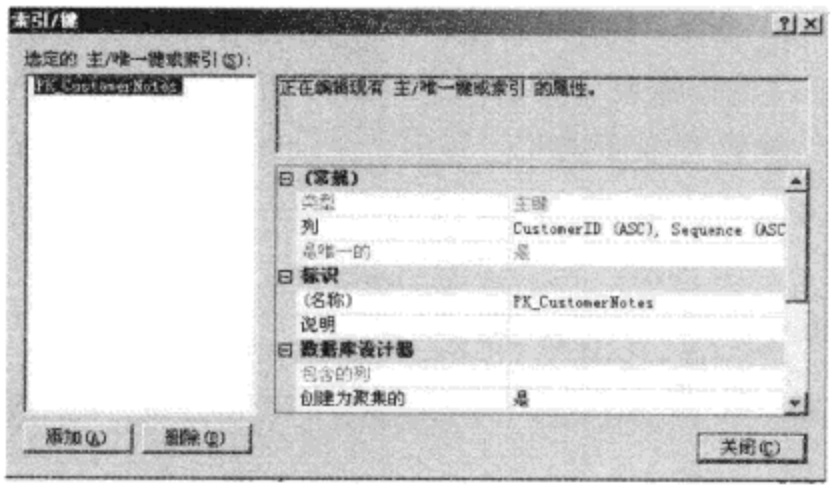


图 8-16

从这里开始，您可以自己设想、创建、编辑和删除索引。还可以建立存储索引的文件组(在很多情况下，是不作改动)。将在第 9 章详细讲述索引。

5. CHECK 约束

接着看下一个菜单选项(或使用工具栏定位到对话框的相应图标)。可以在如图 8-17 所示的对话框中管理 CHECK 约束。

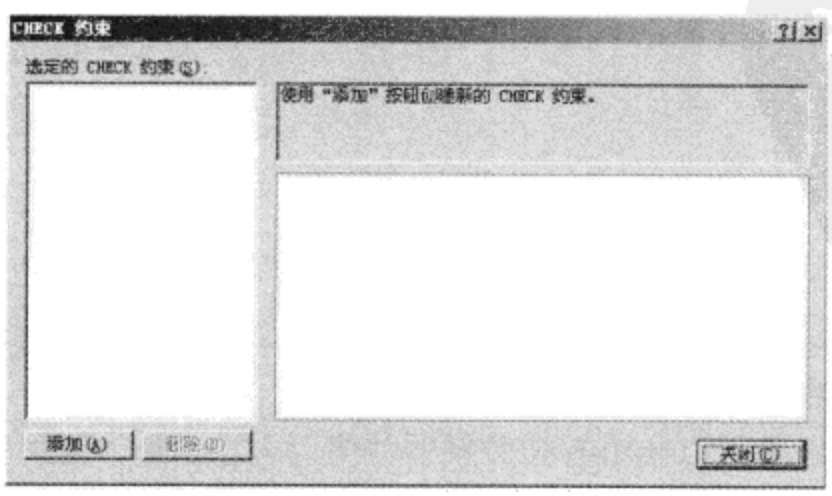


图 8-17

www.TopSage.com

这个选项卡是灰色的，为什么呢？在 CustomerNotes 表中，还没有定义主键以外的任何约束，而主键约束在“索引/键”选项卡中。这个对话框只针对 CHECK 约束——如果希望看到这个选项卡起作用，需要单击“添加”按钮来添加约束。

8.4.3　关系

本章前面介绍了数据库关系图工具提供的与表有关的方面，下面需要了解的是关系线(和基本的关系细节)。

在关系线中，带有钥匙的一端是关系的“一”方。带有无穷符号的一端表示“多”方。这个工具没有专门提供关系线表示可能为零的关系(仍使用同样的线表示)。另外，数据库关系图中实际显示的唯一关系是使用外键约束声明的。任何使用触发器强制的关系——不管哪种关系——不会出现关系线。

再次看看 AdventureWorks2008 数据库关系图，右击 SalesOrderHeader 表，选择“关系”。这将弹出在 8.4.2 节看到的“关系”对话框——SalesOrderHeader 表的“关系”对话框如图 8-18 所示。

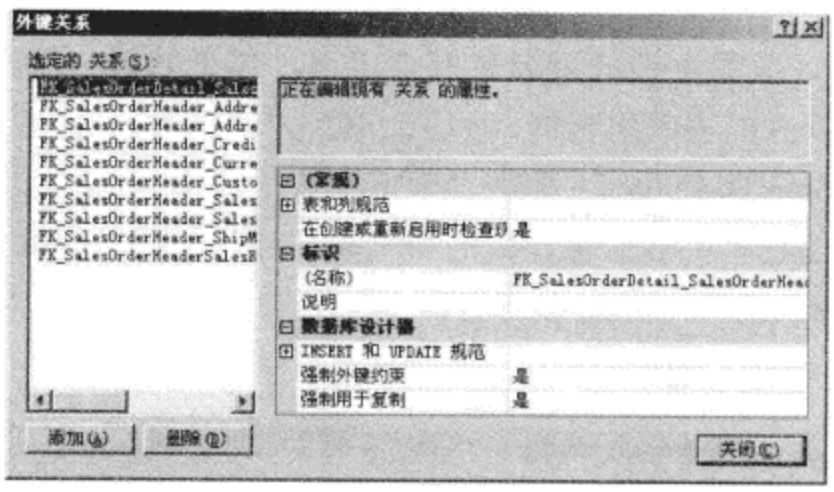


图　8-18

从这里可以编辑关系，包括级联动作、是否启用外键(例如，希望特意添加违反关系的数据)，甚至关系的名称。

数据库设计者对关系的命名各不相同。有的不关心关系的名称，但是作者喜欢采用一个动词短语来描述关系——例如，在 Customers/Orders 关系中，可能命名为 CustomerHas Orders 或者这类的名称。这是无关紧要的——很多情况下都用不到——但发现在查看较长的对象列表或是一个特别复杂的 ER 图(关系线可能跨页越过几个无关的实体)时，上述名称非常有用。

在数据库关系图工具中添加关系

只需要拖放就行了——就这么简单。唯一的技巧是确保在正确的位置拖放。如果有疑问的话，在拖放之前选择需要的列。

试一试　添加关系

在新的 CustomerNotes 表(8.4.2 创建的)和 Customers 表之间添加关系——毕竟，如果是作客

www.TopSage.com

户备注，可能希望确保对一个有效的客户做备注。要添加关系，可以单击然后按住 Customers 表的 CustomerID 列左边的灰色区域，然后拖动鼠标直到指向 CustomerNotes 表的 CustomerID 列。这时会弹出一个对话框来确认相关表的列映射(如图 8-19 所示)。

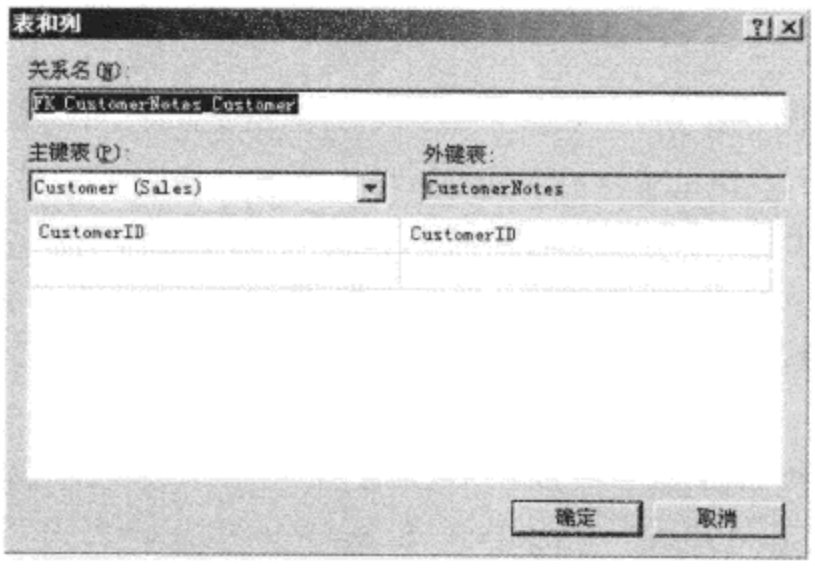


图 8-19

如果拖放正确的话，关系两边的列名称都会出现，但是如果出现不希望的名称，不要担心——只要单击需要改变列的表的组合框，然后选择新列。将默认的关系名 FK_CustomerNotes_Customers 改为 FKCustomerHasNotes。单击“确定”按钮后，弹出更标准的关系对话框，这样可以设置任何需要调整的选择，然后保存表的新关系。接下去将“删除规则”和“更新规则”改为“级联”——确保相关的客户记录得到更新或删除，客户的备注也将在需要时更新或删除，如图 8-20 所示。

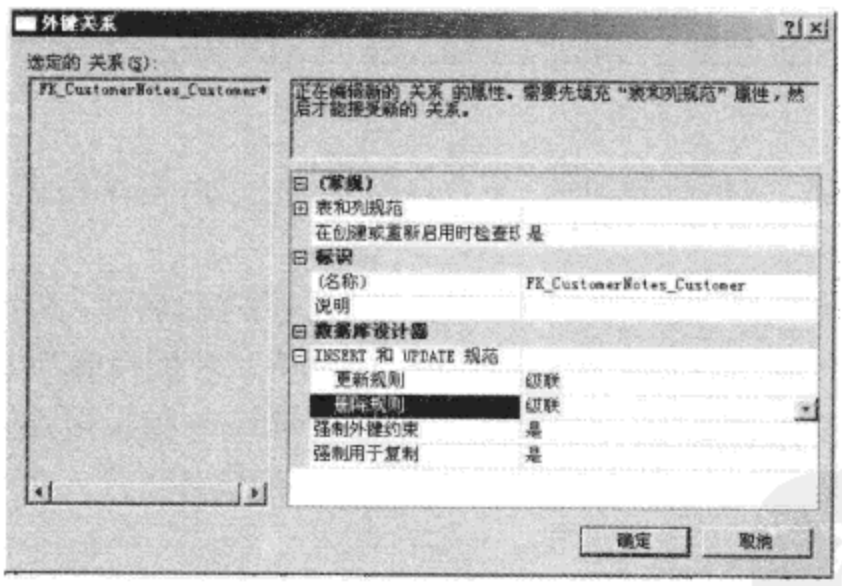


图 8-20

之后，单击“确定”按钮，看看数据库关系图中新的关系线。

示例说明

和 8.4.2 节添加表一样，SQL Server 在后台构建 SQL，作需要的改变。到目前为止，只在数据库关系图中添加关系——将鼠标指针悬停在关系上，可以看到一个带有关系名称和性质的工具提示框，如图 8-21 所示。

www.TopSage.com

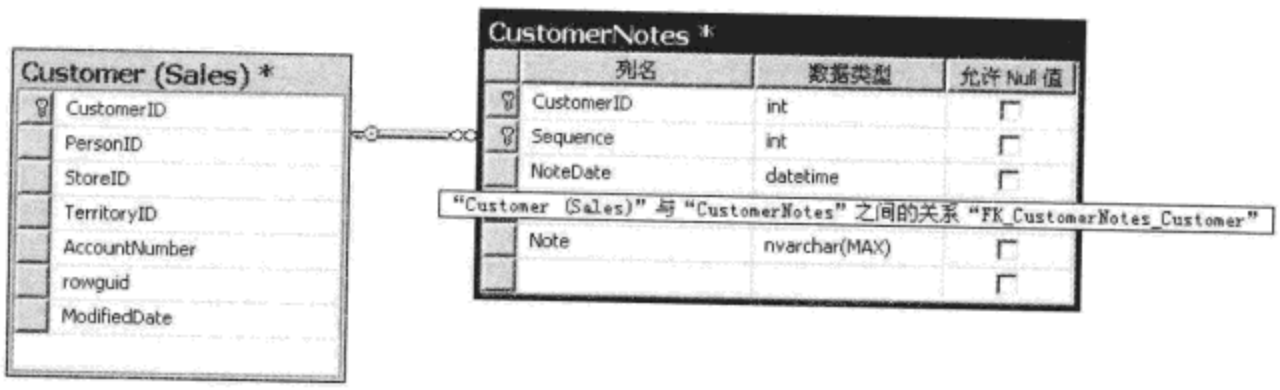


图　8-21

提示：

注意两个表中的星号。所做的改动只添加到数据库关系图的改动列表中——直到选择保存数据库关系图后，才真正添加到物理数据库中。

当“禁用”外键时，关系线的显式方式将改变。可以在第 7 章看到如何禁用约束，也可以在“关系”对话框中通过将“强制外键约束”下拉框设置为“否”来禁用约束。这么做时，关系线将作改变来表明约束已经禁用了。结果如图 8-22 所示。

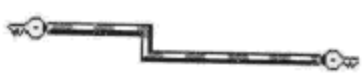


图　8-22

如果看到这种情况，您可能首先会问“为什么外键被禁用了？”也许是有意的，不过需要确认。

8.5　反规范化

本节将简单讲述反规范化的内容，因为这可能涉及到很高级的概念，但是记住不要被数据的规范化所迷惑。

本章前面已经提到过，规范化有时会成为数据库设计者佩戴的十字架，会变成他们的信仰，他们是为了规范化而规范化数据而不是为了数据库的适用性。下面是对这个问题的一些思考。

- 如果声明计算列或者存储一些派生数据有助于更有效地运行报表，那么不管怎样也要采用。只需要考虑优点和可能的风险(例如，如果“汇总”数据与所派生的数据不同步该怎么办？如何确定不同步的情况，以及如果不同步又该如何处理？)
- 有时，在表中包括一个(或者多个)反规范化的列，可以消除或者显著减少检索信息所需的连接数量。要注意这些情况——这些会经常出现。我曾碰到过这种情况，通过添加一列到常用的基表，将表连接从 9 个降低到 3 个，这个过程中查询时间减少了 90%。
- 如果要保持历史数据(大部分不改动的，而且只用于报表的数据)，那么完整性问题是相对次要的问题。一旦数据写到只读区域并经过验证，那么可以确保不会出现“不同步”的问题(不同步问题是数据规范化主要处理的问题)。这时，将数据以平面形式(反规范化)存储到一些表中则更好(也更快)。

www.TopSage.com

- 需要连接的表越少，用户在制作报表的时候就更开心。这个用户群会越来越熟悉他们使用的工具。用户逐渐变成他们自己的 DBA，要直接访问数据库来制作他们自己的自定义报表。对这些用户来说，高度规范化的数据库可能看起来像迷宫，几乎变得没用。反规范化数据可以使这些用户操作起来更容易。

总之，如果有疑问，就规范化数据。通常这样设计关系系统是有原因的。因为越规范化，就越能获得更好的数据完整性，在事务环境下也会获得更好的性能。

8.6 超出规范化的一些规则

本节将讲述设计中“超出规范化”的一些基本规则。这些几乎都不是严格的规则——只是要考虑的问题。这里需要理解的重点是：虽然规范化在数据库设计中很重要，但它不是唯一重要的事情。

8.6.1 保持简单

一些人确实能想出聪明的办法来解决不同的问题。有的想法特别棒并且非常有用。有的办法虽然很好，但是不很实用。往往有的办法——可能很新颖，但是并不好。

在过多谈及创新思维之前，还是要阐明一下这里想传达的思想——不要接受那种“因循守旧”的方法，而是要认识到经过检验的好方法仍需针对具体情况再作检验——这通常很有效。

SQL Server 2008 采用了所有新的方法，使得问题过于复杂。现在通过对产品附加强大而灵活的新功能(代码驱动函数和数据类型)，就可以获得复杂数据规则和复杂数据类型。通常要避免在数据库中引入的复杂性超出真正所需的。最低要求的方法通常(不总如此)产生的结果不仅更容易编辑，而且运行得更快。

8.6.2 选择数据类型

为了贯彻最低要求的思想，就要选择需要的，并且只选择需要的。

例如，如果需要存储月份(数字 1～12)——可以使用单字节数据类型 tinyint。那为什么经常碰到数据库中存储月份的字段使用 int 数据类型(这个是 4 字节的)呢？如果不需要 Unicode 处理任何数据类型，就不要使用 nchar 或者 nvarchar——和相应的非 Unicode 相比，这些数据类型每个都需要两个字节。

注意：

很可能有人认为这是出于对空间的考虑。当提到这个问题时，有人这样评论：“啊，磁盘空间现在很便宜呀！”事实上，一个品牌 SCSI 硬盘的价钱还不至于到了可以随意扔掉的程度，另外还有网络带宽的问题。假设对于每行需要通过网络额外传输 100 字节，而且要传递 100 个记录，这就意味着需要额外传输 10KB 阻塞网络的数据。还觉得不够有说服力？那么假设有 100 个用户每小时执行 50 个事务——那样每小时浪费了 50MB 以上的带宽。

重点是：数据库中出现的很多事情都会重复——这意味着小的错误会像滚雪球一样变得非常大。

www.TopSage.com

8.6.3　尽量进行存储

每次构建数据库时，都会碰到这样一个问题："以后需要这些信息吗？"我的建议是，如果有疑问，就保存信息。要知道，很多时候无法找回失去的数据。

可以确信至少有一次(可能很多次，远不止一次)会发生这样的情况——一位客户(记住，客户就是任何需要从您这里获取信息的人——有可能是内部客户，而不仅仅是"应收账款"账户)找到您并说："可以给我一份去年关于非公司企业的支付情况报表吗？"

那么您存储了有关供应商是否是公司的信息了吗？如果要受美国税法(1099 报告)约束的话，那么最好存储。这样您可以回答说，可以处理这个问题，然后客户回答道："很好！那么可以将他们去年年底时的地址一起打印出来吗？"

可以猜想您没有存储过去的地址，或者至少没有存储地址改动的日期。简单地说，您无法知道系统的用户将会要求什么——尽量确保您拥有用户所要求的所有信息。只是记住不要在网络上用电线传输不必要的数据(参见关于选择数据类型的讨论)。如果只是存储数据以备后用，那么确保在不需要的情况下就不要将它们放到应用程序的 SELECT 语句中(实际上，不管存储什么数据，都应采用这一策略)。

注意：

如果您认为两种处理方法(保留或者删除)可能牵扯到法律问题，那么请咨询您的律师。有时，在法律上要求一些数据必须要保留一段时间；有时又要求尽可能快地合法删除数据。

8.7　创建一个快速示例

下面快速看一下设计"货品计价"数据库(在开始介绍规范化时就使用了该数据库)的过程。对于其中大部分操作，只是将数据库关系图工具应用于已经设计的内容，不过还是会抛出一些新问题，来揭示它们如何影响设计。

8.7.1　创建数据库

和很多第三方数据库关系图工具不同，SQL Server 数据库关系图工具不会创建数据库——必须先创建数据库，这样数据库关系图才能进行工作。

本节并不准备利用任何数据进行讲解，而只是创建一个小的数据库，名为 Invoice。接着使用 Management Studio 中的对话框来设计这个示例。

右击服务器的"数据库"节点后，从快捷菜单中选择"新建数据库"，输入名为 Invoice 数据库的相关信息，其大小为 3MB。

因为已经有第 5 章关于创建数据库的内容(也为了简洁起见)，这里接受所有其他的默认选项，如图 8-23 所示。

www.TopSage.com

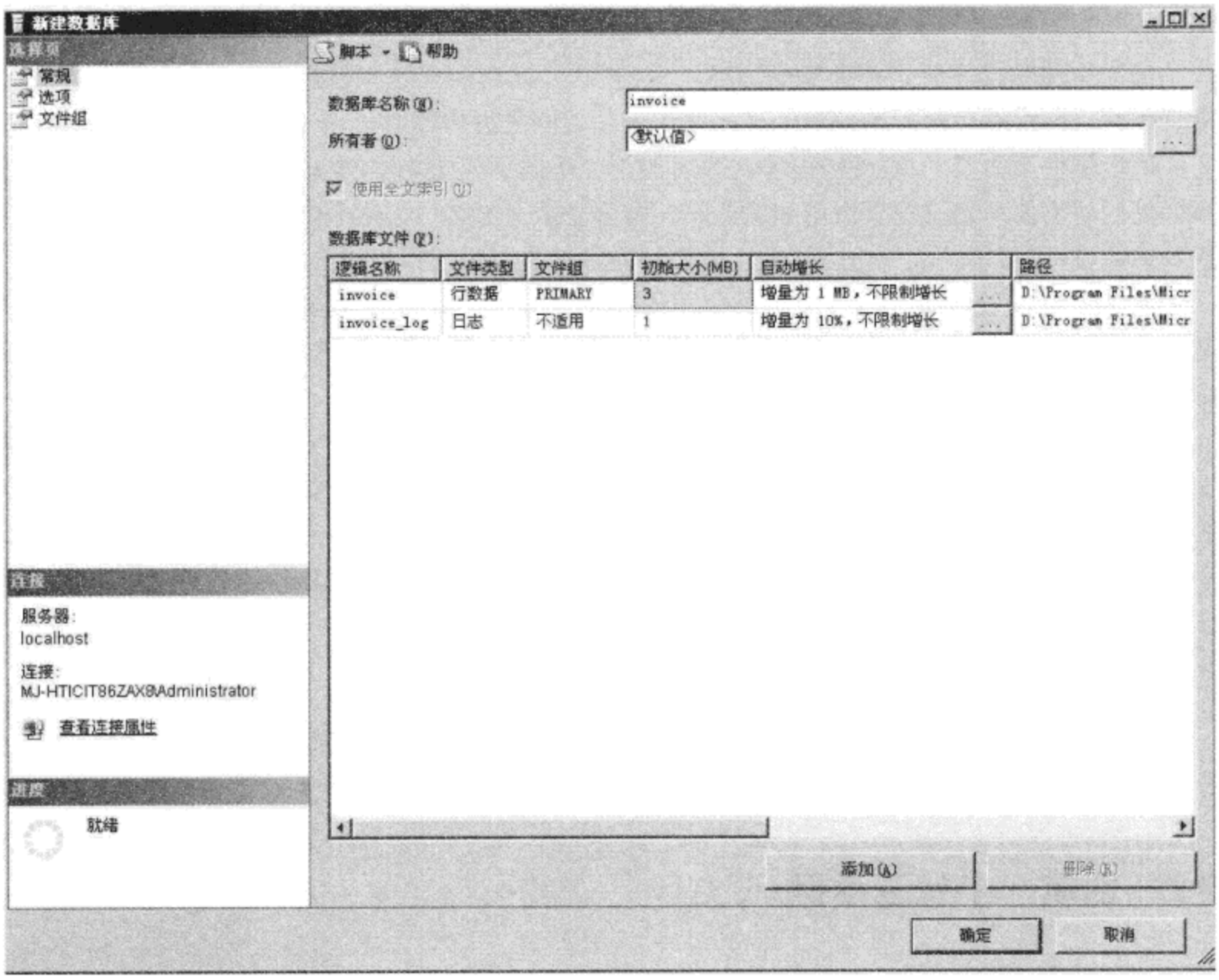


图 8-23

8.7.2 添加数据库关系图和初始表

和创建 AdventrueWorks2008 数据库关系图时一样，展开数据库的节点(应是被添加在“数据库”节点下面，但可能需要右击“数据库”节点并选择“刷新”)，并接受对话框中有关“是否创建数据库关系图所需的对象”的询问。然后右击“数据库关系图”节点并选择“新建数据库关系图”。系统弹出“添加表”对话框，但是由于数据库中没有用户表，单击“关闭”按钮，这样得到一个空白的表格。

现在开始添加新表。单击工具栏的“新建表”或者右击数据库关系图的任何地方，从快捷菜单中选择“新建表”。首先添加 Orders 表，如图 8-24 所示。

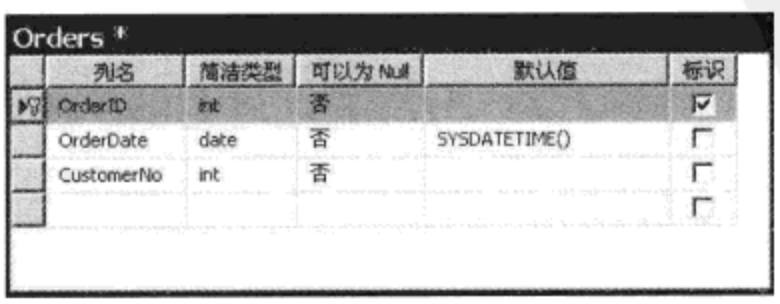


图 8-24

www.TopSage.com

注意：

下面将默认视图(没有将“默认值”列和“标识”列作为视图的一部分)切换到“自定义”视图。还必须选择“修改自定义”，然后选择将默认值和标识列添加到“自定义”视图。

先暂停一下来看看所做的一些决策。虽然解决了规范化问题，但是没有解决其他基本问题。首先是数据类型问题。

因为 OrderID 是该表的主键，所以需要确保有足够的空间使得在插入更多数据时该列的值为唯一的。如果不打算在这个表中经常插入数据，那么可以选择更小的数据类型，但是由于是 Orders 表(希望输入大量订单)，需要增加其存储空间。另外，数值型订单号似乎是合适的(要确保客户喜欢这种编号)，这样可以采用标识列作为自动编号机制。如果需要 20 亿以上的订单编号(在这种情况下，可能是希望拥有公司的一些股票)，那么可以采用更大的 BigInt 数据类型(可以肯定地说，一般不会用到该数据类型，因为不会有这么多订单——只是要记住使用的额外空间，虽然这与在更大的模式下整个数据库使用的空间相比是微不足道的)。

对于 OrderDate，使用新的 Date 数据类型。如果除了希望显示订单日期外，还要显示订单时间，那么需要使用某种 DateTime 数据类型。Date 类型更为紧凑，只能满足简单的“货品计价”数据库的需求。不过要注意 DateTime 数据类型的选择；可以发现有些 DateTime 数据类型在一些语言中更容易起作用。例如，.NET 版本之前的 Visual Basic 在使用 smalldatetime 字段时会不稳定，甚至在后面使用标准的 DateTime 时也还会如此。可以绕过此问题，但很费劲。而对于.NET 版本，一般都使用较新的 DateTime2 数据类型，而不是 DateTime，因为它与.NET 中的 Datetime 数据类型能更直接地对应。

客户已经说过(在前面的样例数据中已看到)，CustomerNo 是 5 位数字的，都是数值型的。您会这样开始问客户：“您确定以后不使用任何字母符号吗？”假如回答是肯定的，那么可以使用整数，原因如下。

- 查找更快。
- 尺寸更小——4 字节可以很容易地包含 5 位数字，但是至少需要 5 字节(如果使用变长字段，需要 6 字节)来处理 5 个字符。

提示：

注意这里的一个技巧——实际上，该表的客户号是由即将与 Customers 表构建的关系来定义的。因为这是本示例中的最后一个表，因此将继续填充该字段。

选择数据类型之后，需要确定列的大小——对于这个特定的表不需要过多考虑，因为所有数据类型都是固定大小的。

下面的问题是，行是否可以为空。在这里，需要确保在输入订单时提供所有可用信息，所以不允许为空。

提示：

以前也接触过这个问题，但是您也可以使得在数据库中允许为空。有时会出现不可避免的情况——“未定义”值是合法的。仍将经常性地在文本字段中输入“未知值”或类似的值。

这样做的原因是可空字段和未声明变量所引发的错误是一样的。每当您在表中遇到空值

www.TopSage.com

时，可能会自问："是我本来就是这样设置的，还是忘记给表中的该行输入值了"——即是否在程序中有错误呢？

接下来的问题是默认值。不能给 OrderID 设定默认值，因为要把 OrderID 作为"标识"列(这两者是互斥的)。不过对于 OrderDate 来说，设定默认值是有意义的。如果没有提供 OrderDate，那么假设订购日期就是今天。最后是 CustomerNo——哪个客户是默认的呢？——当然不能有默认值。

再下来就是"标识"列。OrderID 是个理想的候选"标识"列——该值除了保持行唯一以外，没有其他用途。使用计数器(如标识字段)可以适当而有序地维护唯一值。因为这里没有理由改变标识种子和增量，所以不作改动。让该列从 1 开始计数，每次加 1。

现在准备转向下一个表——OrderDetails 表，如图 8-25 所示。

OrderDetails *

列名	数据类型	允许 Null 值
OrderID	int	☐
LineItem	int	☐
PartNO	char(6)	☐
Qty	int	☐
UnitPrice	money	☐
		☐

图 8-25

由于在这个表中，OrderID 列上有一个外键，因此数据类型是确定的——因为必须采用和引用的字段相同的类型和大小，所以为整型。

每行的 LineItem 会重新计数，因此这里可以使用 tinyint。不过为了安全起见，还是采用 int 类型(以前就碰到有人超越这一限制)。

在这个表中，定义 PartNo 时需要确保它和 Products 表的 PartNo 匹配。因为在那个表中采用 char(6)(后面将很快讲述)，所以就选择这个类型。

Qty 列只能靠猜测。问题是，对于一个行项来说，可能的最大订单数量是多少呢？因为不知道要销售多少，无法猜测最大数量(例如，如果销售桶装石油，可能一次购买几百万桶)。这里也使用 int，但是如果按重量(如加仑数)销售燃料，需要可以接受小数的数据类型，因此需要对需求作仔细的考虑。

UnitPrice 是相对容易的：由于这个字段要存储货币值，其数据类型必须是 money。要小心使用它。实际需要多少小数位呢？美国的开发人员会认为只需要两位(有些货币通常以更多的小数位交易)，但日用品又是怎样的情况呢？例如，汽油在销售时通常有分币的零头。Money 数据类型带 4 个小数位，这满足大部分货币的需求。

接下来又需要考虑所有必需的数据字段。这里不允许有可空的值。

这个表也不适合采用默认值，因此跳过这个部分。

哪个作为标识列呢？您可能希望再次采用 OrderID 作为标识列。不要这么做！记住 OrderID 是需要和另一个表中的某一列匹配的一个值。而那个表已经有一个值了(其恰巧被设置为标识列，但是不一定必须这样做)，因此如果再设置该列作为标识列会产生冲突。插入时会出现警告，因为正试图设置一个标识值。所有其他列或者从另一个表中获得数据，或者需要用户输入数据。IsRowGuid 也不适用。

www.TopSage.com

再接着是 Products 表和 Customers 表，分别如图 8-26 和 8-27 所示。

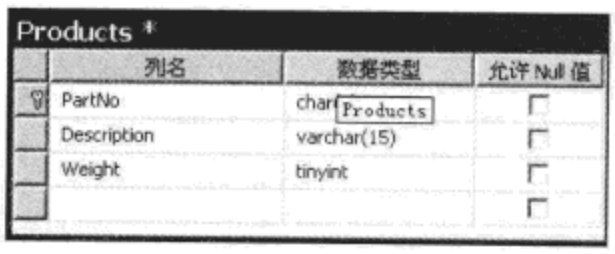


图　8-26

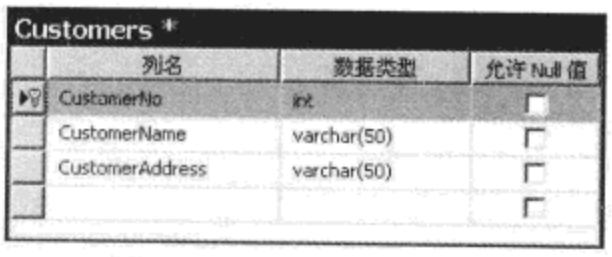


图　8-27

下面将主要讨论一下这里所作的一些选择。

PartNo 已经用在介绍规范化时所看到的数据作了定义。PartNo 列的数据类型是一个数字后面跟着一个字母，然后是 4 个数字。即 6 个字符，看起来像是固定的。希望客户能够确认零件号不会再增大。假设确实是那样，那么将采用 char(6)。因为 char 比 varchar 的系统开销稍微少一些，而长度总是保持不变(这意味着不接受可变长度)。

提示：

重申一下，要确保客户真正考虑了其未来的需求。例如，对 PartNo 列使用简单的 6 字符字段就可能是个值得怀疑的问题。开发人员们总喜欢对零件号采用各种新的编号方法。可能今天的库存管理人员刚发誓说会坚持一直使用某种零件编号方法，但明天又会有新的库存管理人员上任，或是您的公司要与另一家使用 10 位数字的零件号的公司合并。扩展字段不是件坏事，但任何转换都有风险，因此要第一次就做对。

Description 也是不确定的。有时这样的一个字段是由用户界面要求确定的(其宽度不要超出屏幕可显示的范围)，而有时需要估计一个“足够”的空间。这里采用变长 char 而不用定长 char，主要有以下两个理由。

- 节省一点空间
- 不需要处理尾部空格(如果对此有疑问，可以参考第 1 章的 char 和 varchar 数据类型)

这里没有采用 nchar 和 nvarchar，因为这只是美国商业活动中一种较简单的货品计价系统，并不要考虑本地化问题。如果处理多种语言场景，那么可能要对 Unicode 数据类型有更多关注。如果在内部存储国际化信息(如 URL)——其中很可能有日文或类似字符，那需要考虑它们。

Weight 和 Description 有点相似，不容易确定。这里选择 tinyint，是因为产品不会超过 255 磅。可注意到，不允许重量出现小数位(只允许是整数)。正如前面讨论 PartNo 时所说的，要确保仔细考虑了需求——保守很重要，但过分保守可能导致后面更多的工作。

CusotmerNo 字段的内容在创建 Orders 表时讨论过。

CustomerName 和 CustomerAddress 和 Description 的情况基本相同——问题是，多大字符长度才足够呢？但是需要确保不要设置得太大。

和以前一样，所有字段都是必需的(每个表都不允许有可空的字段)，而且不需要默认值。标识列也不合适，因为客户号和零件号都有特定的格式，所以不能采用标识所提供的自动编号系统。

8.7.3　添加关系

为了使数据库关系图不过于复杂，我查看了所有 4 个表并使表的视图只显示列表。要实现

www.TopSage.com

这一目的，方法是右击表，选择“列名”命令。

得到的数据库关系图如图 8-28 所示。

所得到的表位置可能和这里不完全一样，但是内容应该一样的。现在开始准备添加关系，但是可能需要来思考一下需要哪些关系。

所有在 SQL Server 数据库关系图工具中可以用关系线画出来的关系有：一对零、一对一或者一对多关系。SQL Server 不会隐式处理任何其他关系。像本章前面讨论的那样，可以添加唯一约束和触发器之类的事物来支持 SQL Server 对关系的处理，但是这里假设并不这样做，而只限于使用一对零、一对一或者一对多关系。

提示：

不过好的方面是，最常用的就是这些关系。简单地说，不要抱怨 SQL Server 没有包括每种关系。标准的外键约束(实际上就是由引用线表示的)适用于几乎所有需要处理的问题，其他的情况通常可以使用其他方法仿真处理。

从系统的中心表——Orders 表开始。首先，看看该表需要的任何关系。在这里，有一个关系——需要引用 Customers 表。这是一对多关系，Customers 作为父表(“一”方)，而 Orders 作为子表(“多”方)。

要构建该关系(外键约束作为这个关系的基础)，只需在 Customers 表中 CustomerNo 列的最左边(在灰色区域中)单击并按住鼠标不放，然后拖动到 Orders 表中 CustomerNo 列旁的同样位置(灰色区域)，最后释放鼠标按钮。SQL Server 迅速地先后弹出两个对话框来确认关系的配置。第一个对话框如图 8-29 所示，确认实际关联的列。

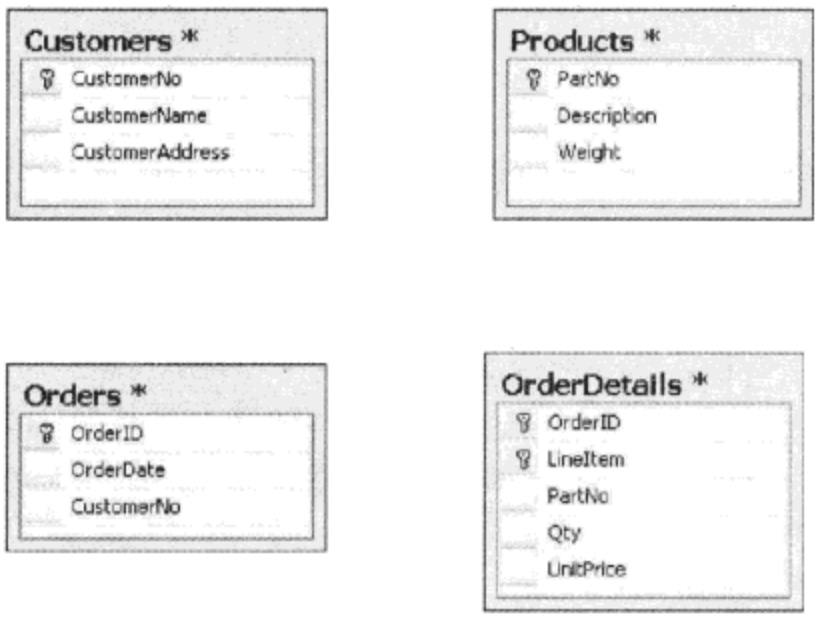


图 8-28

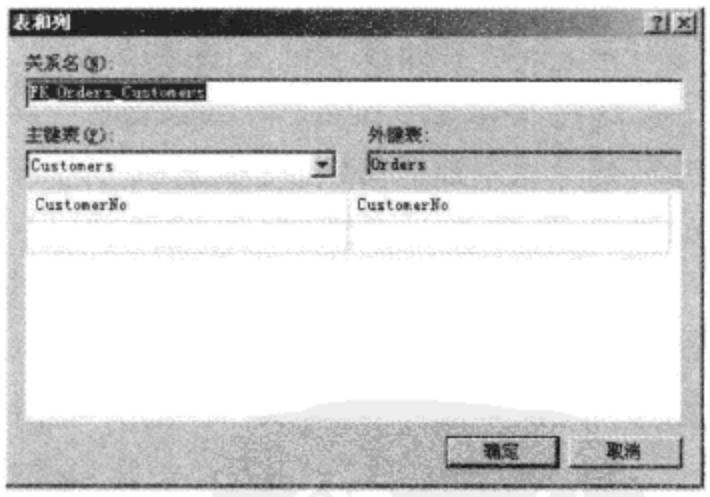


图 8-29

提示：

正如本章前面指出的，不要担心出现的名称和所需的不一致——可以使用组合框来改变为需要的列，让两边都是 CustomerNo。另外，两边的名称不一定要相同——取相同的名称只是为了在真正相同时避免混淆。

单击这个对话框中的“确定”按钮，然后单击“外键关系”对话框中的“确定”按钮，接受默认值。在单击第二个对话框的“确定”按钮之后，就在新数据库中建立了第一个关系，如

www.TopSage.com

图 8-30 所示。

接下来对其他两个关系进行同样的操作。需要基于 OrderID 建立 Orders 到 OrderDetails 的一对多关系(将出现一个订单对应一个或多个订单细节)。还需要基于 ProductID 建立 Products 到 OrderDetails 类似的关系(将出现一个 Products 记录对应多个 OrderDetails 记录)，如图 8-31 所示。

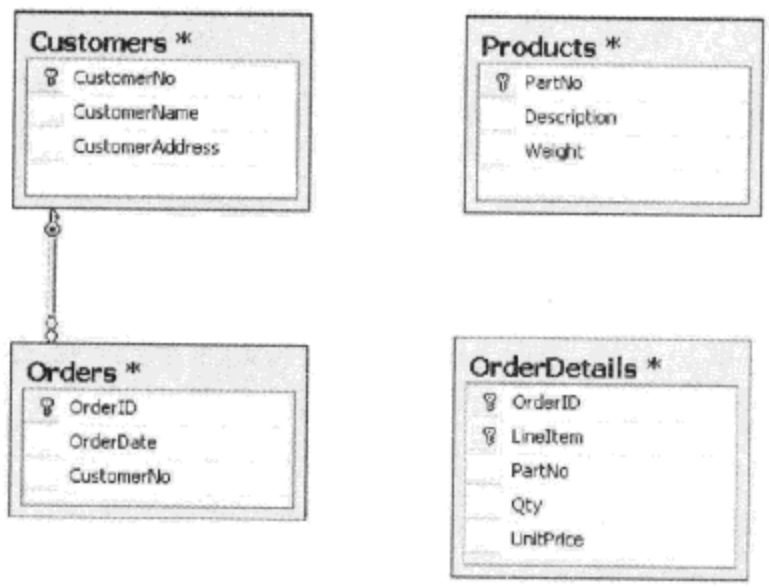


图　8-30

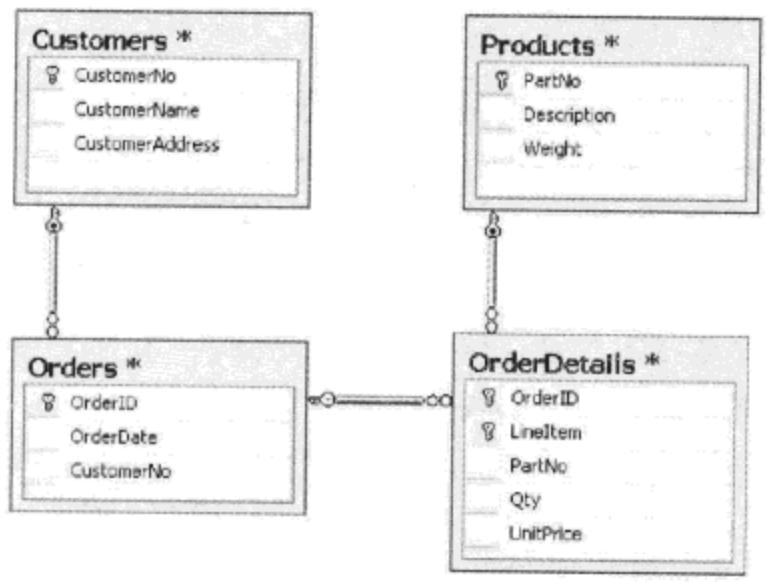


图　8-31

8.7.4　添加一些约束

在构建表和关系的过程中，曾经提到过一个还没有处理的要求。这个要求需要通过约束来强制：零件号格式是 9A9999，这里“9”表示 0～9 的数字，而“A”表示一个字母(非数字)符号。

要添加这个要求，可以右击 Products 表，选择“Check 约束”，弹出的对话框如图 8-32 所示。

在这里单击“添加”按钮来定义约束。为了限制输入的零件编号为定义的格式，可以使用 LIKE 运算符：

```
(PartNo LIKE '[0-9][A-Z][0-9][0-9][0-9][0-9]')
```

这将评估用户输入到表的 PartNo 列的每个字符。第一个字符必须为 0~9 的数字，第二个为 A~Z 的字母，后面 4 个都为数字(还是 0~9)。将这些输入到“表达式”文本框中。另外，将约束的默认名称 CK_Products 改为 CKPartNo，如图 8-33 所示。

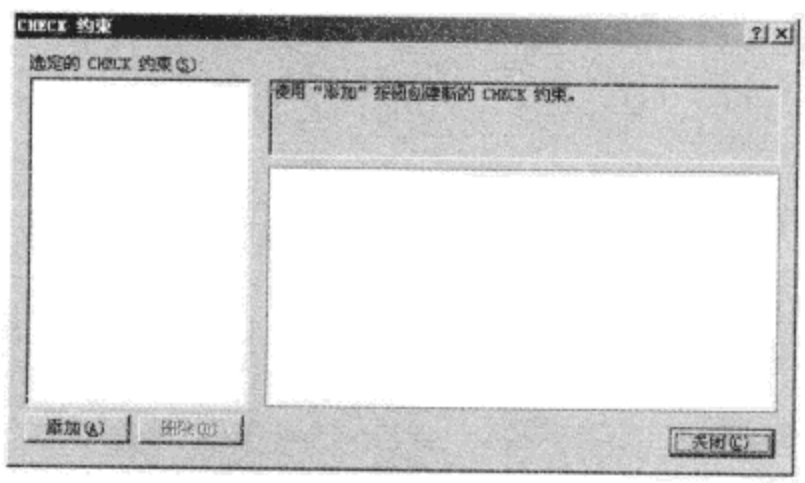


图　8-32

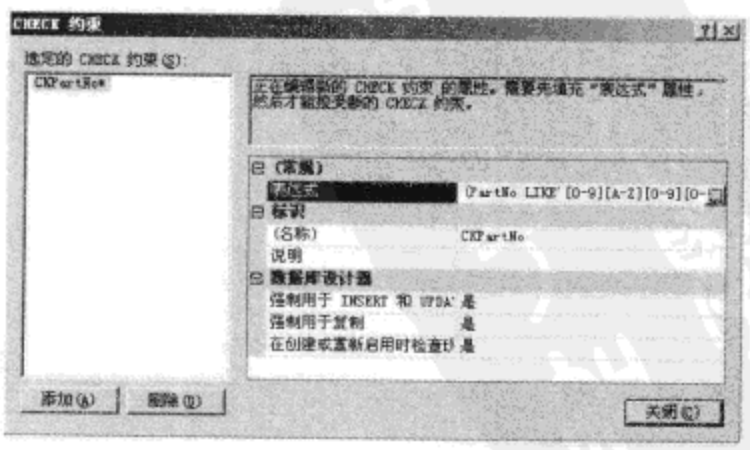


图　8-33

www.TopSage.com

这不需要很长时间——现在就设计好了第一个数据库！

当然，这个是相对简单的模型——但这里已经完成了构建实际数据体系结构的 90%或者更多的工作。

8.8 本章小结

数据库设计是个庞大的概念，有很多优秀的书籍专门讲述这个主题。实际上不太可能在一两章中涵盖所有的数据库设计概念。

本章为读者打下一个坚实的数据库设计的基础。当数据处于第三范式时，就可认为数据已经规范化了。如果到达该层次，则说明消除了重复信息，而数据完全依赖于键——简单说，数据依赖于“键，整个键，只是键。”然而，规范化并不总是适当的选择——有时反规范化数据的策略可以简化数据库，提高报表性能。最后，讲述了数据库设计中一些与反规范化相关的概念，还讲述了如何使用内置数据库关系图工具设计数据库。

第 9 章将更详细讲述 SQL Server 如何存储信息以及如何有效使用索引。

8.9 练习

1. 将表 8-13 中的数据规范化为第三范式。

表 8-13

Patient	SSN	Physician	Hospital	Treatment	AdmitDate	ReleaseDate
Sam Spade	555-55-5555	Albert Schweitzer	Mayo Clinic	Lobotomy	10/01/2005	11/07/2005
Sally Nally	333-33-3333	Albert Schweitzer	NULL	Cortizone Injection	10/10/2005	10/10/2005
Peter Piper	222-22-2222	Mo Betta	Mustard Clinic	Pickle Extraction	11/07/2005	11/07/2005
Nicki Doohickey	123-45-6789	Sheeze Sheila	Mustard Clinic	Cortizone Injection	11/07/2005	11/07/2005

www.TopSage.com

第 9 章

SQL Server 存储和索引结构

索引是数据库规划和系统维护的一个关键部分。它们为 SQL Server(以及任何其他的数据库系统)提供了查找数据和定位到数据物理位置的快捷方式的其他方法。通过添加正确的索引可以大大减少查询执行时间。遗憾的是，许多设计很差的索引实际上会增加查询运行所花费的时间。事实上，索引正逐步成为 SQL Server 中最容易令人误解的对象，因此也是最容易管理不当的对象。

本章将从开发人员和管理者的角度学习索引，但是为了理解索引，也需要理解在 SQL Server 中是如何存储数据的。因此，本章将首先介绍 SQL Server 的数据存储机制。

9.1 SQL Server 存储机制

SQL Server 中的数据可以看成是以某种层次结构形式存在的。这种层次结构相当简单。其中的一些对象是将要被直接处理的，因此很容易理解。少数其他的一些对象存在于内部，虽然在某些情况下可以直接处理它们，但通常是不可以的。下面将逐个介绍。

9.1.1 数据库

数据库很容易理解。常听见人们说："数据库，我知道的。"是的，您可能确实了解它，但这里将它作为单个实体特别列出，是因为(对于给定服务器来说)它是最高级别的存储机制。尽管不能显式创建一个数据库级锁，但这仍是可以建立锁的最高级别。

提示：

锁是系统使用的一种占位标记。当使用 SQL Server(或其他数据库)进行开发时，您将发现理解和管理锁对于系统而言是非常关键的。

第 14 章将详细地介绍锁，但是在介绍存储机制时会顺带讨论一下 SQL Server 中对象的可锁定性。

www.TopSage.com

9.1.2 区段

区段(extent)是用来为表和索引分配空间的基本存储单元。它由 8 个连续的 64K 数据页组成。

基于区段(而不是实际使用空间)分配空间的概念，对于习惯于操作系统存储原理的人而言可能有些难以理解。关于区段的要点包括以下两方面。

- 一旦区段已满，那么下一记录将要占据的空间不是记录的大小，而是整个新区段的大小。许多 SQL Server 的新手都会在空间估计上犯错误，一部分原因是由于一次分配一个区段而不是一个记录。
- 通过预先分配空间，SQL Server 节省了为每个记录分配新空间的时间。

只是因为向当前分配的区段添加了过多的行就要占用整个区段，看起来是一种浪费，但是这种方法所浪费的空间数量一般不是非常多。尽管如此，这些浪费的空间会累加——特别在碎片较多的环境中——因此应当谨记这一点。

占据所有空间的好处是 SQL Server 省去了一些分配时间的系统开销。不需要在每次写入行时都考虑分配问题，SQL Server 仅仅当需要新的区段时才处理额外的空间分配。

提示：

不要将区段占用的空间和数据库占用的空间相互混淆。分配给数据库的那些空间就是磁盘驱动器将要减少的可用空间。区段仅仅是在由数据库保留的整个空间内仿次进行分配的方式。

9.1.3 页

与区段是数据库中的分配单元类似的是，页是特定区段中的分配单元。每个区段包含 8 页。

页是在到达实际数据行之前所能达到的最后一个存储级别。尽管每个区段中的页数是固定的，但是每一页中的行数不是固定的——这完全取决于行的大小，而行的大小是可以变化的。可以把页看作是表行和索引行数据的容器。通常不允许行跨页。

有许多种不同的页类型。本书只涉及以下两种类型。

- **数据**：数据页是不需解释的——它们是表中的实际数据，但任何没有用 text in row 选项、varchar(max)或 varbinary(max)定义的 BLOB 数据除外。
- **索引**：索引页也是相当直观的——它们既包括非群集索引的非叶级页和叶级页(本章后面将研究这些内容)，又包括群集索引的非叶级页。通过本章的后续介绍，读者将更了解这些索引类型。

页拆分

当页满时，它会进行拆分。这意味着多个新页被分配——也意味着现有页上有近半的数据被移到新页上。

在使用群集索引时，会有例外。如果有一个群集索引，并且下一个插入的记录在物理上将作为表中的最后一个记录，那么创建一个新页，然后将该新行添加到这个新页中，而不需要重新定位任何现有数据。在研究索引时将介绍有关页拆分的内容。

www.TopSage.com

9.1.4　行

您可能听说过很多有关“行级锁定”的内容，因此对该术语应该不会感到陌生。行最大可达 8KB。

除了 8 060 字符的限制外，还有最大 1 024 标准列(非稀疏列)的限制。实际上，您会发现在突破 8 060 字符限制前用完列的情况并不多见。1 024 提供了小于 8 字节的平均列宽度。对于大多数使用情况，将会很容易超出这个平均值(因此会在超出 1 024 列之前超出 8 060 字符)。而例外情况往往出现在度量和统计信息中——其中有大量存储了数值样本的不同数据内容。即使在那些应用程序中，也很难突破 1 024 列的限制。如果突破了，则要运用到稀疏列，9.1.5 节将介绍它。

9.1.5　稀疏列

稀疏列(sparse column)是 SQL Server 2008 新引入的一种特殊的数据结构。它们主要用于处理重复的场景，其中的列只是偶尔会用到。也就是，它们大部分时候为空。许多时候，一旦遇到少量这类列，就意味着之后可能会遇到大量这类列。使用稀疏列，可以将单个表中允许的列的总数提高到 30 000。

在内部，标记为稀疏填充的列的数据嵌在单个列中——可以打破之前的 1 024 列的限制，而不用作较大的体系结构更改。

Image、text、ntext、geography、geometry、timestamp 和所有用户自定义数据类型不能被标记为稀疏列。

注意：

尽管较新版本的 SQL Native Client 会自行处理稀疏列，但其他形式的数据访问在访问稀疏列时会有不同的行为。在按名称选择列时，列的稀疏属性是透明的，但如果通过在选择列表中使用“*”进行选择，不同的客户端访问方法可能会以一致的 XML 列形式提供稀疏列，或是根本不显示这些列。您现在可能希望尽可能快地更新客户端库。

提示：

稀疏列属于高级主题范畴，但作者的本意是让读者知道有这样一个概念，并且明白它对于特定场景是个可行的方案。

9.2　理解索引

韦氏字典中将索引定义为“通常按字母顺序排列的一些指定数据(如作者、主题或者关键字)的列表(如目录信息或著作正文的引用)”。

下面将在数据库上下文中采用一种更简单的表述，即索引是一种能快速访问数据的方法。尽管如此，韦伯斯特的定义其实也不错——甚至对数据库而言也适用。

韦伯斯特定义中的关键词是“通常”。而“字母顺序”这一定义会随许多规则而变化。例如，在 SQL Server 中，有许多不同的可用排列规则选项。其中一些选项如下所示。

www.TopSage.com

- **二进制**：按字符的数字表示形式排序(例如，在 ASCII 码中，用数字 32 表示空格，用 68 表示字母“D”，而用 100 表示字母“d”)。因为所有内容都表示为数字，所以处理起来速度最快——遗憾的是，它并不总是如人们所想象，在 WHERE 子句中进行比较时，使用该选项会造成严重的混乱。
- **字典顺序**：这种排序方式与在字典中看到的排序方式一样，但是稍有不同。可以设置大量不同的额外选项来决定是否区分大小写、音调和字符集。

如果告诉 SQL Server 注意区分大小写，那么“A”和“a”将不相等，这一点很容易理解。同样，如果告诉 SQL Server 不区分大小写，那么“A”和“a”将相等。如果添加区分音调的选项，情况就变得有些混乱。即如果 SQL Server 注意区分变音符号，那么“a”和“á”是不同的，同时和“à”也不同。令许多人感到更加困惑的是：排列规则顺序如何既影响数据的相等性，又影响排序顺序(从而影响它们在索引中的存储方式)。

通过示例，看一下表 9-1 中一些排列规则选项的性质，以及它们对排序顺序和相等性的影响。

表 9-1

排列规则顺序	比 较 值	索引存储顺序
字典顺序、不区分大小写、不区分音调(默认)	A = a = à = á = â = Ä = ä = Å = å	a, A, à, â, á, Ä, ä, Å, å
字典顺序、不区分大小写、不区分音调、大写字母优先	A = a = à = á = â = Ä = ä = Å = å	A, a, à, â, á, Ä, ä, Å, å
字典顺序、区分大小写	A≠a, Ä≠ä, Å≠å, a≠à≠á≠â≠ä≠å, A≠Ä≠Å	A, a, à, á, â, Ä, ä, Å, å
字典顺序、区分大小写	A≠a, Ä≠ä, Å≠å, a≠à≠á≠â≠ä≠å, A≠Ä≠Å	A, a, à, á, â, Ä, ä, Å, å

这里的要点在于：索引中的存储顺序取决于为数据建立的排列规则信息。可以在数据库或列级设置排列规则，因此有相当细粒度的控制级别。如果假定服务器是不区分大小写的，那么需要确保系统文档能进行处理，或者有良好的电话技术支持——特别是如果您正在美国之外进行销售。假设您是独立软件供应商(ISV)，并且将产品销售给准备在现有的服务器上进行安装的顾客(对于顾客而言，这似乎是完全合理的事情)，但是现有的服务器碰巧是已设置为区分大小写的老式服务器。那您将会接到不满意的顾客的电话，要求相关的技术支持。

注意：

一旦设置了排列规则顺序，改变它是非常困难的(但也是可能的)，因此在设置之前要确定所需的排列规则顺序。

9.2.1 平衡树(B-树)

平衡树或 B-Tree 的概念当然不是随 SQL Server 引入的。实际上，在数据库系统内外有非常多的索引系统使用平衡树。

平衡树仅是提供了一种以一致且相对低成本的方式查找特定信息的方法。其名称中的“平衡”是自说明的。平衡树是自平衡的(只有很少的例外情况)，这意味着每次树进行分支时都有

www.TopSage.com

接近一半的数据在一边，而另一半数据在另一边。而名称中的“树”的概念也是很清楚的——如果绘制该结构，然后倒置它，会发现该结构具有树的一般形状，因此称为树。

平衡树始于根节点(这是另一个类似于树的方面，但还不止这些)。如果有少量的数据，这个根节点可以直接指向数据的实际位置。在这种情况下，最终获得的结构如图 9-1 所示。

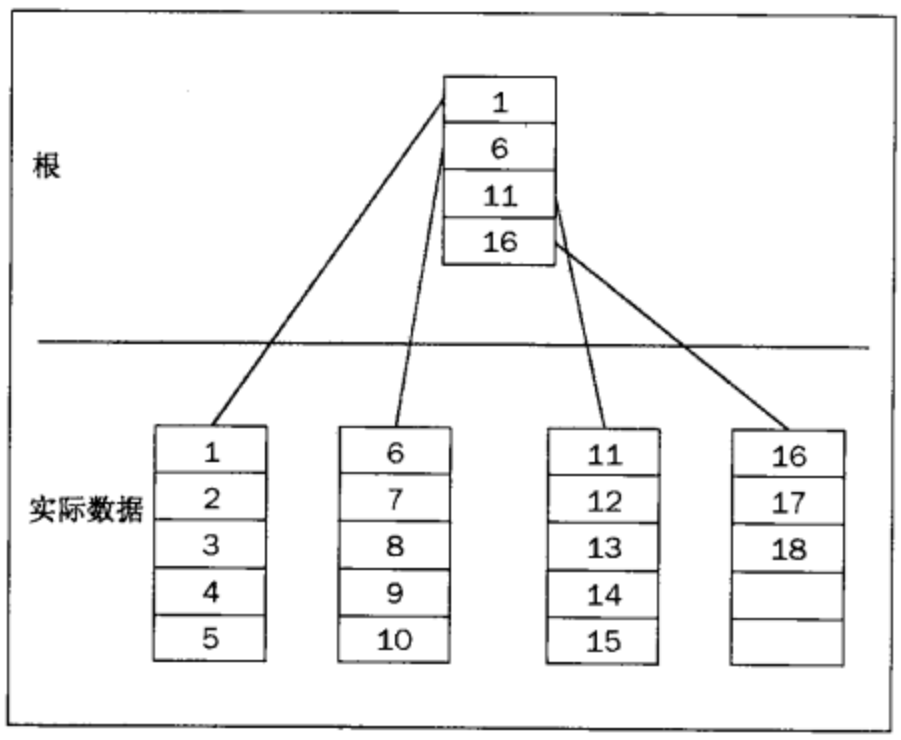


图　9-1

因此，从根节点开始并浏览记录，直到找到以小于查找值的值开始的最后一页。然后获得指向该节点的一个指针并且浏览它，直到找到想要的行。

然而，在大多数情况下，有很多通过根节点引用的数据，因此根节点指向中间的节点(或称为非叶级节点)。非叶级节点是位于根节点和说明数据的物理存储位置的节点之间的节点。非叶级节点可以指向其他非叶级节点或指向叶级节点(最后一次拿树作类比)。叶级节点是从中可获取对实际物理数据的引用的节点。如同叶子是导航树的线路的末端一样，叶级节点是导航索引的线路的末端——从那里，可以直接到达在其上部数据的实际数据节点。

从图 9-2 中可以看出，也是开始于根节点，然后移动到以等于或小于查找值的最高值开始的同时也在下一级中的节点。然后重复这个过程——查找具有等于或者小于查找值的最高起始值的节点。继续沿着树一级级往下，直到到达叶级——从中知道数据的物理位置，并且可以很快地定位数据。

页拆分简介

所有这些页在读取方面工作良好——但在插入时会有点棘手。回顾一下，“B-树”中的“B”表示平衡(Balanced)。而且前面提到每次遇到树中的分支时，因为每一边都大约有一半的数据，所以 B-树是平衡的。另外，由于添加新数据到树上的方法一般可避免出现不平衡，所以 B-树有时被认为是自平衡的。

通过将数据添加到树上，节点最终将变满，并且将需要拆分。因为在 SQL Server 中，一个节点相当于一个页——所以这被称为页拆分，如图 9-3 所示。

www.TopSage.com

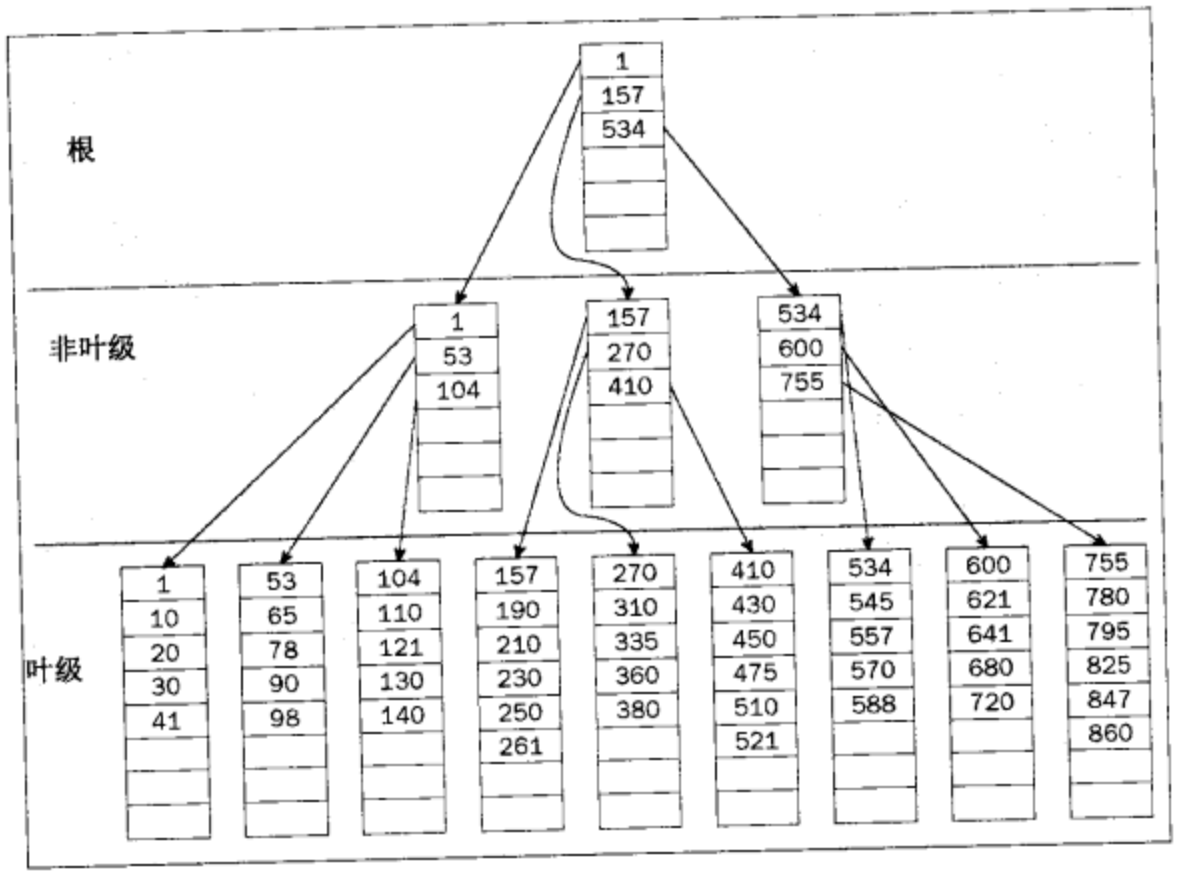


图 9-2

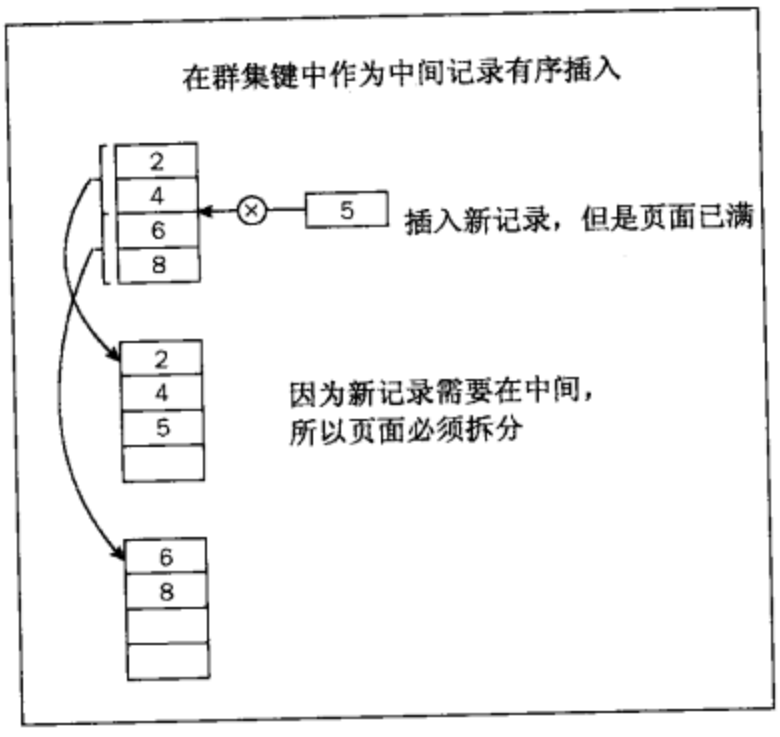


图 9-3

提示：

当发生页拆分时，数据自动地四处移动以保持平衡。数据的前半部分保留在旧页上，而数据的剩余部分添加到新页——这样就形成对半拆分，使得树保持平衡。

如果稍考虑下这个拆分过程，将认识到它在拆分时增加了大量系统开销。不只是插入一页，而是进行下列操作。、

- 创建新页
- 将行从现有页移动到新页上
- 将新行添加到其中一页上

www.TopSage.com

- 在父节点中添加另一个记录项

但是，系统开销远不止这些。因为在进行树的排列，就可能有级联动作。创建新页时(因为拆分的缘故)，需要在父节点中建立另一个记录项。在父节点中的这个记录项在该级别也可能导致页拆分，而且整个过程会重新开始。实际上，这种可能性扩展到所有节点，甚至会影响根节点。

如果根节点拆分，那么实际最终会创建两个额外的页。由于只能有一个根节点，所以之前作为根节点的页被拆分成两个页，而且成为树的新中间级别。然后创建全新的根节点，并且将有两个记录项(一个指向旧的根节点，另一个指向拆分的页)。

显然，页拆分会对系统性能产生非常负面的影响，其表现是在服务器上的处理会暂停几秒(此时页被拆分并且重写)。

接下来将讨论一下如何防止页拆分。

提示：

虽然叶级的页拆分是很常见的，但是在中间节点进行页拆分却很少发生。当表增长时，索引的每一层将进行页拆分，但是，因为中间节点对于下一级节点上的几个记录项只有一个记录项，所以当向树的上层移动时，页拆分的数量变得越来越少。尽管如此，对于发生在叶级以上的拆分来说，在下一级别上一定已经有一个拆分——这意味着沿树而上的页拆分在本质上是累积的(而且严重影响性能)。

虽然 SQL Server 有许多不同类型的索引(稍后将讨论)，但是所有这些索引都以某种方式利用这种平衡树方法。事实上，由于平衡树的灵活特性，所有索引在结构上非常类似。不过，它们实际上还有一些重要的区别，并且这些区别会对系统的性能产生影响。

提示：

对于 SQL Server 索引而言，树的节点以页的形式出现，但是实际上可以将根节点、非叶级节点、叶级节点，以及树结构的概念应用于 SQL Server 或数据库之外的其他系统。

9.2.2　SQL Server 中访问数据的方式

从广义上讲，SQL Server 检索所需数据的方法只有两种。

- 使用表扫描
- 使用索引

SQL Server 使用何种方法来执行特定查询将取决于可用的索引、所需的列、使用的连接以及表的大小。

1. 使用表扫描

表扫描是相当直观的过程。当执行表扫描时，SQL Server 从表的物理起点处开始，浏览表中的每一行。当发现和查询条件匹配的行时，就在结果集中包含它们。

您可能听说过有关表扫描的不好方面，基本上这些情况是属实的。但有时，表扫描实际上是最快的访问方法。一般来说，当从相对较小的表中检索数据时，使用表扫描访问方法就非常好。这时，表的确切大小将随表的宽度和查询的特性而有很大变化。

www.TopSage.com

提示：

您知道在查询的 WHERE 子句中使用 EXISTS(如果确实使用得当的话)可以大大提高性能的原因吗？当使用 EXISTS 运算符时，只要找到和条件匹配的记录，SQL Server 就会立即停止。如果有一个包含一百万条记录的表，并且在第三个记录中找到了匹配的记录，那么使用 EXISTS 选项将避免读取 999 997 条记录！NOT EXISTS 以同样的方式工作。

2. 使用索引

当 SQL Server 采用索引，该过程实际上与表扫描的工作方式相类似，但是有一些捷径。

在查询优化过程中，优化器查看所有可用的索引结构并且选择最好的一个(这主要基于在连接和 WHERE 子句中所指定的信息，以及 SQL Server 在索引结构中保存的统计信息)。一旦选择了索引，SQL Server 将在树结构中导航至与条件匹配的数据位置，并且只提取它所需要的记录。区别在于，因为数据是排序的，所以查询引擎知道它何时到达正在查找的当前范围的下界。然后它可以结束查询，或者根据需要移至下一数据范围。

如果考虑一下迄今已经涉及的查询主题(特别是第 7 章)，就可以注意到 EXISTS 选项的工作方式有一些明显的类似之处。EXISTS 关键字允许查询在找到一个匹配记录时立刻停止运行。使用索引所获得的性能与使用 EXISTS 类似甚至更好，因为查找数据的过程的工作方式是类似的；也就是说，服务器可能使用某种索引知道何时没有相关的内容，并且立即停止。更好的情况是，通过使用索引，不必限于采用布尔逻辑的情况。可以将这种相同的概念应用到范围的开始和末尾。可以收集一定范围内的数据，其优点本质上与使用索引查找数据的优点相同。此外，可以对数据执行非常快速的查找(称为 SEEK)，而不是在整个表中查找。

提示：

不要因为前面对索引与 EXISTS 运算符所进行的比较就认为索引可完全取代 EXISTS 运算符(或反之)。这两者不是相互排斥的；它们可以一起使用，并且可以是经常性的。在这里同时提到它们，仅仅是因为它们有一些相似性，即能够显示其工作何时完成，以及在到达表的物理末尾之前退出。

3. 索引类型和索引导航

尽管表面上在 SQL Server 中有两种索引结构(群集索引和非群集索引)，但是实际上，就内部而言，有 3 种不同的索引类型。

- 群集索引
- 非群集索引，该索引又包括以下两种。
  - 堆上的非群集索引
  - 群集表上的非群集索引

物理数据的存储方式在群集索引和非群集索引中是不同的。而 SQL Server 遍历平衡树以到达末端数据的方式在所有 3 种索引类型中也是不同的。

所有的 SQL Server 索引都有叶级和非叶级页。正如讨论平衡树时所提到的那样，叶级是保存标识记录的“键”的级别，非叶级页是叶级的引导者。

索引在群集表(如果表有群集索引)或者堆(用于没有群集索引的表)上创建。

www.TopSage.com

1) 群集表

群集表是在其上具有群集索引的任意表。后面将详细讨论群集索引，但是它们对于表而言意味着以指定顺序物理存储数据。通过使用群集键唯一地标识独立的行——群集键即定义群集索引的列。

提示：

您可能会有这样一个疑问："如果群集索引不是唯一的，那将怎样？"也就是说，如果索引不是唯一索引，那么群集索引如何用于唯一地标识一行？一切都在内部处理——SQL Server 迫使任何群集索引为唯一的——即使您没有以这样的方式定义。幸运的是，它并没有改变使用索引的方式。如果愿意的话，仍然可以插入重复的行，但是 SQL Server 将在内部添加一个后缀到键上，以保证行具有唯一的标识符。

2) 堆

堆是在其上没有群集索引的任意表。在这种情况下，基于行的区段、页以及行偏移量(偏移页顶部的位置)的组合创建唯一的标识符，或者称为行 ID(RID)。如果没有可用的群集键(没有群集索引)，那么 RID 是唯一必要的内容。

4. 群集索引

群集索引对于任意给定的表而言是唯一的——一个表只能有一个群集索引。不一定非要有群集索引，但是如果查看索引类型，您会发现由于多种很显然的原因，它正成为最常被选用的一种类型。

使群集索引特殊的方面是，群集索引的叶级是实际的数据——也就是说，数据重新排序，按照和索引排序条件声明的相同物理顺序存储。这意味着，一旦到达索引的叶级，就到达了数据。任何新记录都根据其正确的物理顺序插入到群集索引中。创建新页的方式随需要插入记录的位置而变化。

如果新记录需要插入到索引结构中间，就会发生正常的页拆分。来自旧页的后一半记录被移到新页，并且在适当的时候，将新记录插入到新页或旧页。

如果新记录在逻辑上位于索引结构末端，那么创建新页，但是只将新记录添加到新页，如图 9-4 所示。

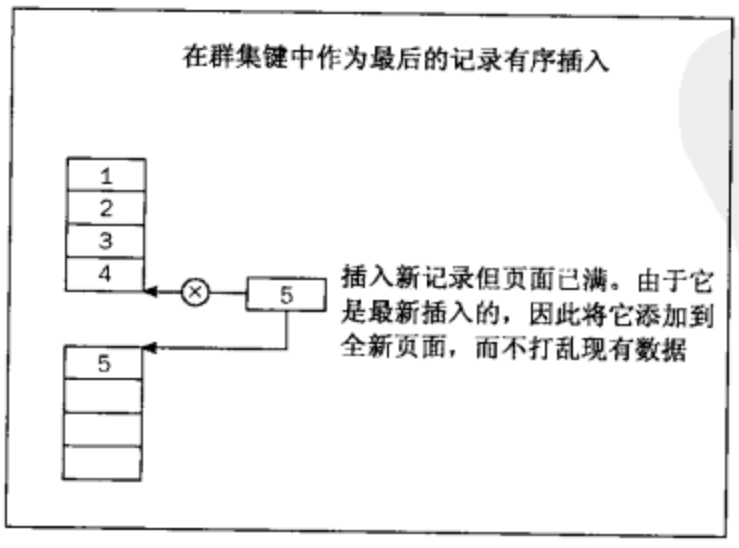


图　9-4

www.TopSage.com

5. 导航树

正如前面所指出的，在 SQL Server 中甚至索引也是存储在平衡树中。在理论上，平衡树在作为树分支的每个可能方向上总是具有一半的剩余信息。群集索引的平衡树形式如图 9-5 所示。

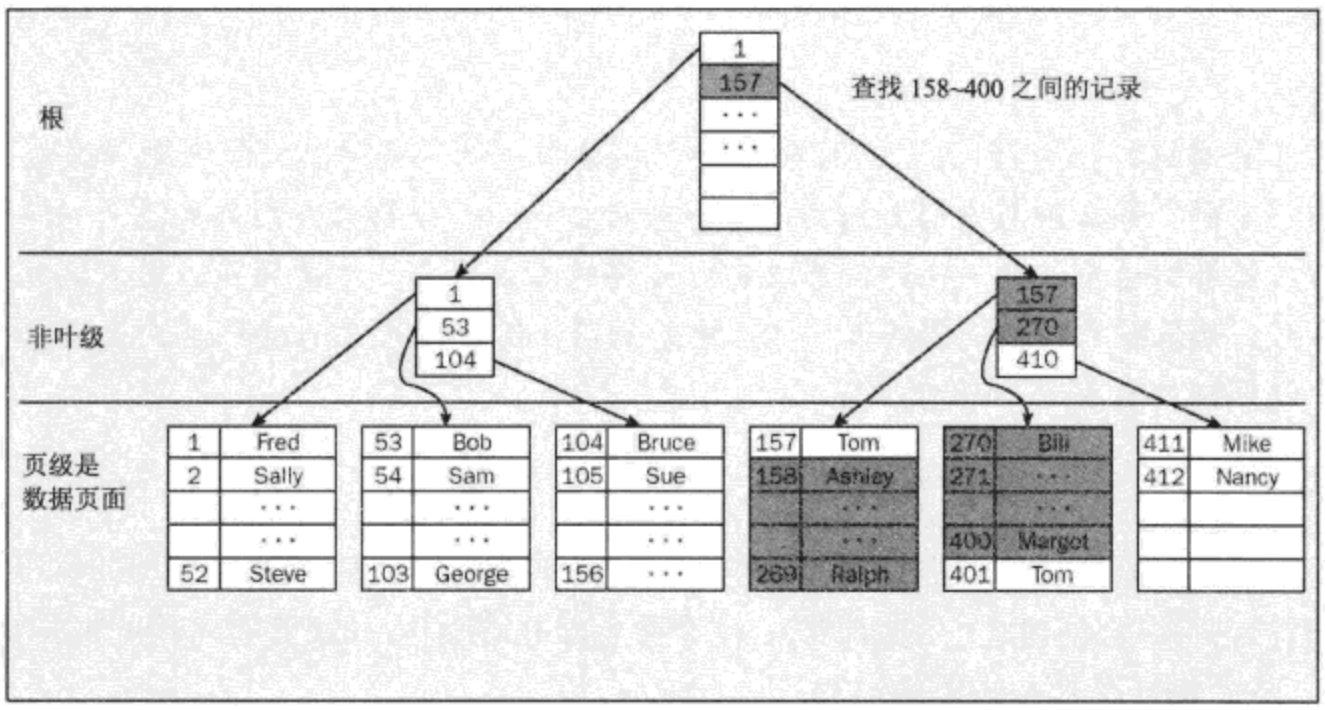


图 9-5

正如所看到的那样，它看起来和本章前面讨论的较为一般的平衡树是基本相同的。在这里，执行对数字 158～400 的范围查询(群集索引特别擅长的事情)。全部需要做的工作是导航到第一个记录，并且包含在该页上的所有剩余的记录。之所以知道需要该页的剩余部分，是因为来自于上一级节点的信息也需要来自于一些其他页的数据。因为这是有序列表，所以可以确信它是连续的——这意味着如果下一页有应当包含的记录，那么这个页的剩余部分必须被包含。无需任何验证就可从那些页上提取数据。

首先导航到根节点。SQL Server 能够基于在 Sys.indexes 系统元数据视图中保存的记录项定位根节点。

通过查找作为根节点的页，可以知道下一个需要检查的页是什么(即这里的第 2 个级别的第 2 个页)。然后继续这个过程。沿着树一步步往下，就到达越来越小的数据子集。最终，将到达索引的叶级。对于群集索引来说，到达索引的叶级意味着也到达了期望的行和期望的数据。

注意：

下列区别非常重要：使用群集索引，当完全导航至索引时，也意味着完全导航至数据。而在使用非群集索引时，其产生的性能差别才真正显示出来——特别是当非群集索引建立在群集索引之上时。

6. 堆上的非群集索引

堆上的非群集索引和群集索引在大多数方面以类似的方式工作。尽管如此，它们具有一些显著的差别。

叶级不是数据——相反，它是一个可从中获得指向该数据的指针的级别。该指针以 RID 的形式出现，如同在本章前面描述的那样，这种 RID 由索引指向的特定行的区段、页以及行偏移

www.TopSage.com

量构成。即使叶级不是实际的数据(相反，它具有 RID)，使用叶级也仅仅比使用群集索引多一个步骤。因为 RID 具有行的位置的全部信息，所以可以直接到达数据。

然而，不要误以为这“多的一个步骤”只有少量的系统开销，并且堆上的非群集索引将和群集索引几乎一样快地运行。使用群集索引，数据在物理上是按照索引的顺序排列的。这意味着，对于一定范围的数据，当找到在其上具有数据范围起点的行时，那么很可能有其他行在同一页上(也就是说，因为它们存储在一起，所以在物理上已几乎到达下一个记录)。使用堆，数据并未通过除索引外的其他方法链接在一起。从物理上看，绝对没有任意种类的排序。这意味着，从物理读取的角度看，系统可能不得不从整个文件中检索记录。实际上，很可能最终多次从同样的页中取出数据。SQL Server 没有方法知道它将需要回到该物理位置，因为在数据之间没有链接。使用群集索引，它知道这是物理上的排序，因此仅仅通过访问页一次就完全获得数据。

提示：

堆上的非群集索引和群集索引一样，通常任何已读取一次的页将仍然在内存缓存中，而且同样将非常快地被检索。此外，它添加了一些额外的逻辑操作来检索数据。

图 9-6 显示了与使用群集索引执行的同样的查询，只是这次使用在堆上的非群集索引的方式。

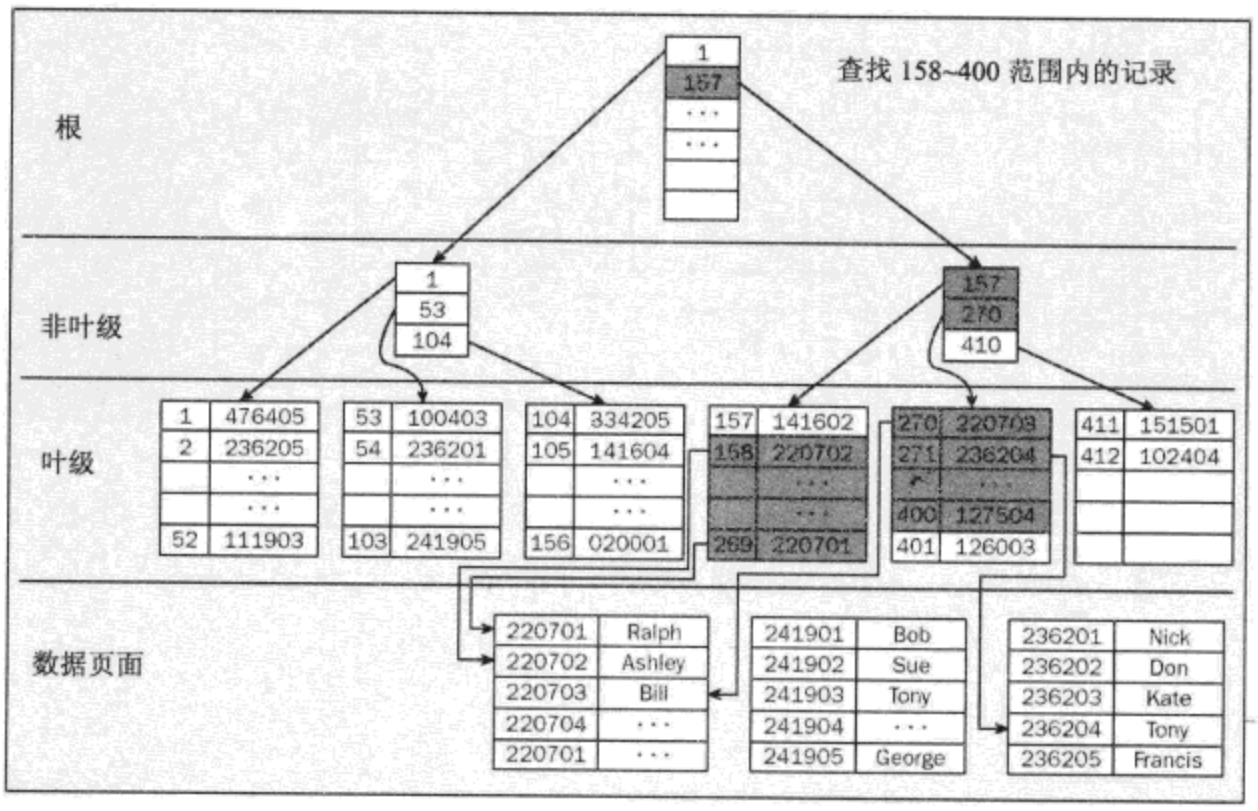


图　9-6

主要通过索引导航，一切都按以前的方式工作。以相同的根节点开始，并且遍历树，处理越来越焦中的页，直到到达索引的叶级。这里就有了区别。以群集索引的方式，能够正好在这里停止，而以非群集索引的方式，则需要做更多的工作。如果非群集索引是在堆上，那么只要再进入一个级别。获得来自叶级页的 RID，并且定位到该 RID——直到这时才可以直接获得实际的数据。

7. 群集表上的非群集索引

使用群集表上的非群集索引时，还有一些类似性——但同样也有区别。和堆上的非群集索引一样，索引的非叶级节点的工作与使用群集索引时相比几乎一样。区别出现在叶级。

www.TopSage.com

在叶级，与使用其他两种索引结构所看到的内容有相当明显的区别。有另外一个索引来查找。使用群集索引，当到达叶级时，可以找到实际的数据。使用堆上的非群集索引，不能找到实际的数据，但是可以找到能够直接获得数据的标识符(仅仅多了一步)。使用群集表上的非群集索引，可以找到群集键。也就是说，找到足够的信息继续并利用群集索引。

最终结果类似于图 9-7 所示。

最终是两种完全不同的查找。

在图 9-7 中，首先是一个范围搜索。在索引中执行一次单独的查找，并且可以浏览非群集索引以找到满足条件(如'T%')的连续数据范围。这种能够直接到达索引中的特定位置的查找被称为 seek。

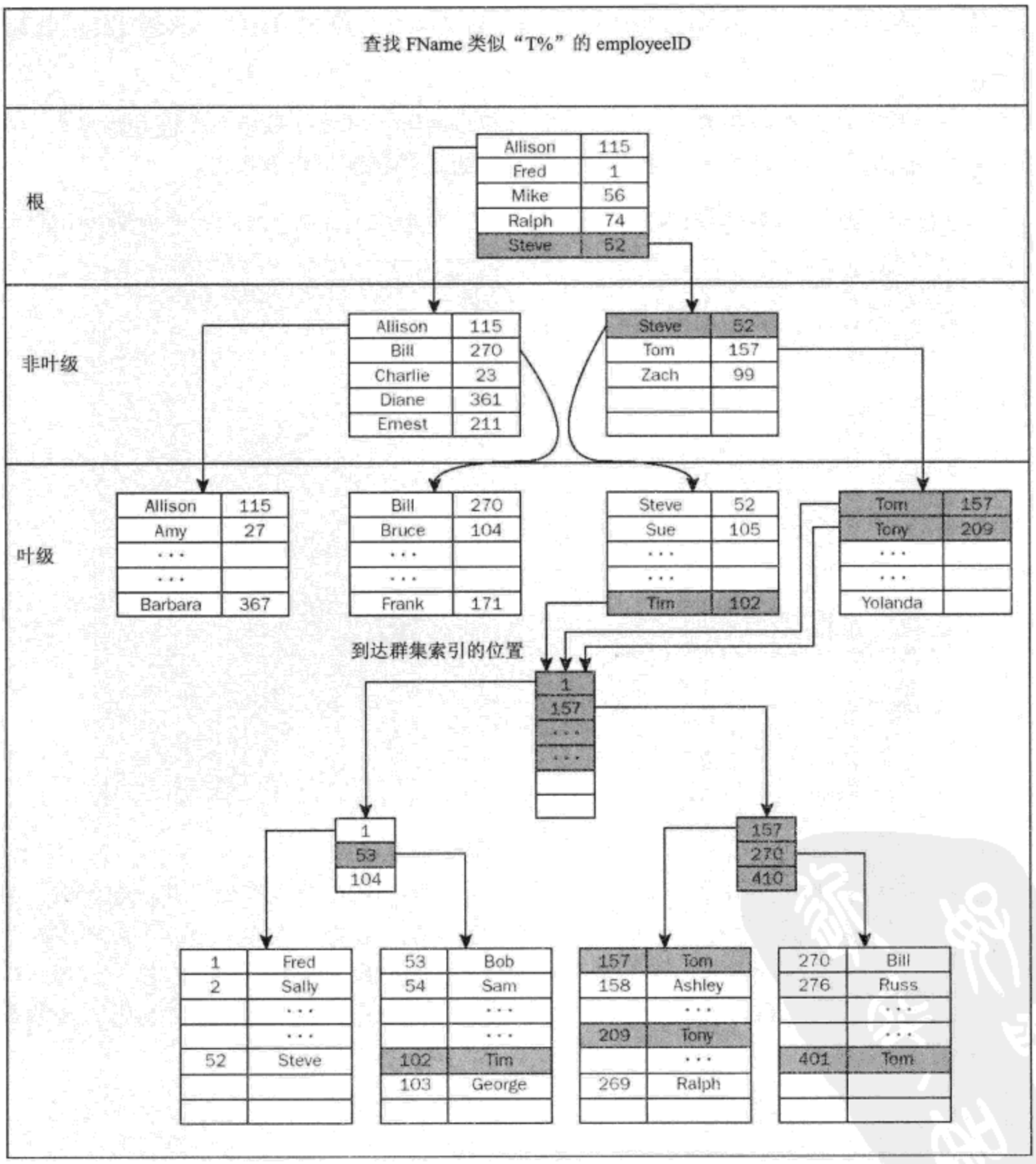


图 9-7

www.TopSage.com

然后开始第二个查找——使用群集索引的查找。第二种查找非常迅速；问题在于它必须执行多次。可以看到，SQL Server 从第一个索引中查找检索列表(所有名字以“T”开始的列表)，但是该列表在逻辑上并没有以任意连续的方式与群集键相匹配——每个记录需要单独地查找，如图 9-8 所示。

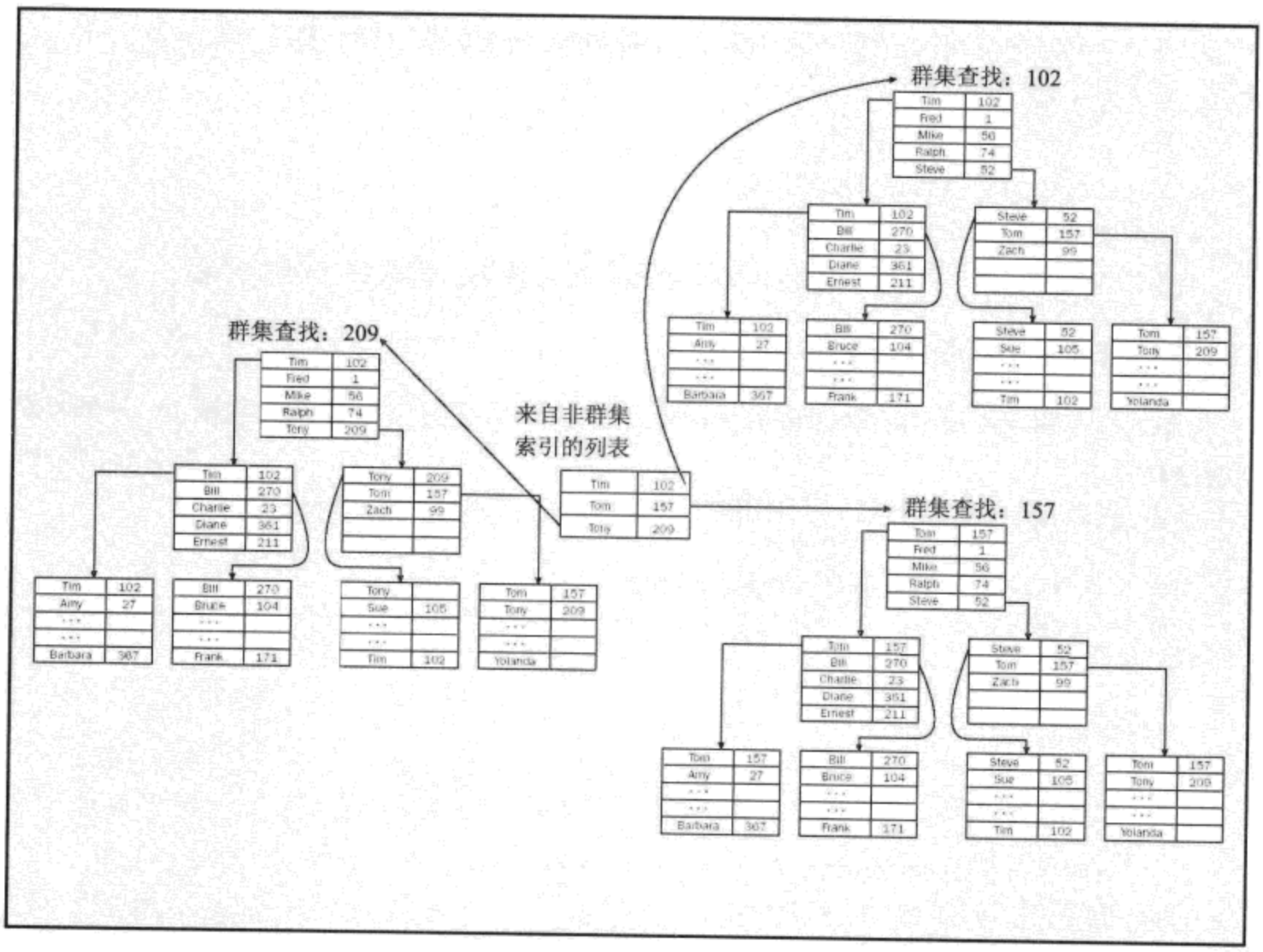


图　9-8

自然，这种多个查找的情况比一开始仅能使用群集索引引入了更多的系统开销。第一个索引查找——通过非群集索引的方法——只需要非常少的逻辑读操作。

例如，如果有一个每行 1 000 字节的表，而且执行了类似的查找(比如将要返回 5 行或 6 行的内容)；它将仅以 8~10 个逻辑读的顺序获取来自非群集索引的信息。然而，这仅使得可以在群集索引中查找行。这些查找每个都将花费 3~4 个逻辑读，或者 15~24 个额外的读取。乍看起来，这可能并不是大问题，但是如果逻辑读从最小值 3 上升到最大值 24——这意味着需要完成的工作量以 800%的速度增长。

如果扩展一下思维——即来自非群集索引的值范围不是 5 或 6 行，而是 5 000 或者 6 000 行，或者 500 000 或 600 000 行——那么将有巨大的影响。

提示：

不要被与群集索引对比的额外系统开销所吓倒——这并不是说不使用索引，而是指出从读操作的角点看，非群集索引通常不如群集索引有效(有时在插入时，它实际上是更好的选择)。索引通常(有一些例外)都是执行查找的最快方法。本章后面将介绍使用何种索引以及

www.TopSage.com

选用的原因。

9.3 创建、修改和删除索引

创建、修改和删除索引的操作和对其他对象(例如表)的操作一样。接下来就来看一下每个操作，首先从 CREATE 开始。

可以以以下两种方式创建索引。

- 通过显式的 CREATE INDEX 命令
- 在创建约束时作为隐含的对象

因为这两种方式都有一些可做什么和不可做什么的特性，所以要分别作讨论。

9.3.1 CREATE INDEX 语句

CREATE INDEX 语句所做的事情与其听上去一样——用于在指定表或视图上基于声明的列创建索引。

创建索引的语法如下所示，并且引入一些到目前为止还没有真正讨论的项：

```
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：

这里许多选项都有遗留语法，因此可以使用这些语法支持以前版本的 SQL Server。但是，并不赞成使用该语法，并且它也将在某个时候被删除——这里强烈推荐尽可能使用较新的语法。

注意：

对于创建 XML 索引而言，有一个类似的但完全不同的语法。本节最后将单独介绍它。

不严格地说，这种语句遵循前面已多次看到的相同的 CREATE <object type> <object name> 语法(而且将要更多地看到这种语法)。主要问题是有一些在其他地方还没有看到过的干扰参数。

正如在第 10 章中将要看到的关于视图的内容，必须在 CREATE 语句中添加额外的子句以处理索引实际上不是孤立的对象的事实。CREATE INDEX 语句必须随表或者视图出现，并且需

www.TopSage.com

要声明列所在(ON)的表。

在 ON <table or view name>(<column name>)子句之后，所有内容都是可选择的。可以混合搭配这些选项。它们中的大多数很少使用，但是一些选项(例如 FILLFACTOR)会对系统性能和行为产生明显的影响，所以逐个讨论它们。

1. ASC/DESC

这两个选项允许为索引选择升序和降序排列顺序。默认选项为 ASC，它是升序。

您可能会产生如下的疑问，为什么升序和降序对索引有关系呢——如果需要相反的排列顺序，SQL Server 只需反向查看索引即可。但是，生活并不总是如此简单。如果只处理一列，或者所有列总以同样的方式排序，那么反序查看索引是可行的——但是如果需要在索引中混合排序顺序，该怎么办呢？也就是说，如果一列需按照升序排序，但是其他列按照降序排列，这该如何处理?因为索引的列存储在一起，所以对一列反向查看索引也将倒转其他列的顺序。如果显式地声明某一列是升序，而另一列是降序，那么将直接在物理数据中倒转第二列——突然间就不必改变访问数据的方式了。

举一个简单的示例。设想一个生成报表的场景，在其中需要将雇员列表按照雇佣日期排序，从最近雇佣的开始(降序)，但是也想按照他们的姓(升序)排列。在早期的版本中，SQL Server 将不得不执行两个操作——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列顺序，在组合列上获得了灵活性。

2. INCLUDE

这是 SQL Server 2005 及后续版本支持的选项。它的目的是为覆盖查询(covered queries)提供更好的支持。

当包含(INCLUDE)列而不是将列放在 ON 列表上时，SQL Server 仅仅在索引的叶级上添加它们。因为在索引叶级上的每一行对应于一个数据行，所以所做的事情在本质上是将更多的原始数据包含在索引叶级上。如果对此作一下考虑，或许会猜测 INCLUDE 实际上仅应用于非群集索引(群集索引已经是叶级的数据，所以这样做无意义)。

这样做有何关系？正如本书后面进一步讨论的，SQL Server 一旦有了它实际需要的内容，就停止工作。所以，如果在遍历索引时没有继续访问实际的数据行就找到需要的所有数据，那么就不必再到达数据行(考虑一下这样做的意义)。通过在索引中包含特定的列，可以在叶级“覆盖”利用该特定索引的查询，从而节省了与使用索引指针到达数据页相关的 I/O。

提示：

注意不要滥用该选项！当包含列时，将增加索引页的叶级的大小。这意味着每页中的行数将更少，因此需要更多的 I/O 来查看相同数量的行。结果可能是，加快一个查询的同时可能减慢了其他的查询。要考虑对系统各个部分的影响，而不是仅仅考虑某个时候正在使用的特定查询。

3. WITH

WITH 非常简单——它只是告诉 SQL Server 将要提供一个或者多个跟在后面的选项。

www.TopSage.com

4. PAD_INDEX

在语法列表中，该选项第一个出现——但是在理解了 PAD_INDEX 所做的工作后，会觉得该选项有点奇怪。简而言之，该选项决定了第一次创建索引时，索引的非叶级页将有多满(用百分比表示)。不用在 PAD_INDEX 中声明百分比，因为将使用后面的 FILLTACTOR 选项中指定的百分比。设置不带有 FILLFACTOR 选项的 PAD_INDEX＝ON 将是没有意义的(这就是先出现该选项会有点奇怪的原因)。

5. FILLFACTOR

当 SQL Server 第一次创建索引时，默认情况下将尽可能地将页填满，仅留两个记录的空间。可以将 FILLTACTOR 设置为在 1 ~ 100 之间的任意值。一旦索引构造完成，这个数字将表示页相对满的程度的百分比。但是，要牢记在进行页拆分时，数据将仍然在两页之间对半分布——除了定期重建索引(这是您应当做的事情)外，不能不断地控制填充百分比。

当需要调整页密度的时候，使用 FILLTACTOR。可以作如下几方面考虑。

- 如果是 OLTP 系统，那么需要较低的 FILLFACTOR。
- 如果是 OLAP 或者其他非常稳定的(就改变而言，指几乎没有添加和删除)系统，那么需要尽可能高的 FILLFACTOR。
- 如果事务比例中等，且有很多基于它的报表类型查询，那么可能需要中等水平的 FILLFACTOR(不太低，也不太高)。

如果没有提供值，那么 SQL Server 将把页填充至差两行满为止，同时保证每页最少有一行(例如，如果行是 8 000 字符宽，那么每页只能放一行，所以无法达到差两行满)。

6. IGNORE_DUP_KEY

IGNORE_DUP_KEY 选项几乎是一种回避系统的方法。简而言之，它使得唯一约束与其应有的操作方式有些不同。

通常，唯一约束(或唯一索引)不允许任何种类的重复——如果事务尝试基于定义为唯一的列创建重复值，那么事务将被回滚并且拒绝。然而，一旦设置了 IGNORE_DUP_KEY 选项，就将得到混合的行为。仍然接收错误消息，但是错误将仅仅是一种警告——记录仍然没有被插入。

从 IGNORE_DUP_KEY 的角度看，上面最后一行提到的“记录仍然没有被插入”是个重要概念。不会对该事务进行回滚(错误是警告错误，而不是关键错误)，但重复的行将被拒绝。

为什么要这样做？这是在不干扰试图插入重复值的事务的情况下存储唯一值的方法。对于任何正在插入将要重复的值的过程来说，重复行完全没有问题(没有来自于它的逻辑错误)。相反，该过程可能的态度是：“只要知道那里有类似的行存在就行了——不关心它是否是尝试插入的特定行”。

7. DROP_EXISTING

如果指定 DROP_EXISTING 选项，那么具有所讨论名称的任何现有索引将在构造新索引之前被删除。当和群集索引一起使用该选项时，这个选项比简单删除并重新创建现有的索引更加有效。如果重新构建与现有索引完全匹配的索引，那么 SQL Server 知道它不需要涉及非群集索

www.TopSage.com

引，然而为了适应不同的行位置，显式删除和创建将导致重新构建所有非群集索引两次。如果使用 DROP_EXISTING 改变索引的结构，那么 NCI 只被重新构建一次，而不是两次。此外，不能简单地删除和重新创建由约束所创建的索引，例如，用于实现特定的 FILLFACTOR。而 DROP_EXISTING 可用于完成这一工作。

8. STATISTICS_NORECOMPUTE

默认情况下，SQL Server 试图自动化在表和索引上更新统计信息的过程。通过选择 STATISTICS_NORECOMPUTE 选项，表示将负责更新统计信息。为了关闭这个选项，需要运行 UPDATESTATISTICS 命令，但不使用 NORECOMPUTE 选项。

这里强烈建议不要使用这个选项。原因何在？查询优化器使用索引上的统计信息来指出索引对于给定查询有多大用处。随着表中数据大量增多或减少，以及列中的特定值改变，索引上的统计信息会不断变化。结合这两个事实，可以知道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。打开自动更新统计信息功能意味着统计信息将周期性地更新(多长时间更新一次取决于对表更新的本质和频繁程度)。相反，关闭自动更新统计信息功能意味着信息会过时，或者需要设定计划以手动运行 UPDATE STATISTICS 命令。

9. SORT_IN_TEMPDB

只有在 tempdb 存储在与包含新索引的数据库物理上分离的驱动器上时，该选项才有意义。该选项在很大程度上是管理功能，因此并不打算过多介绍这个主题，而只是简要概述它是什么以及为什么它只有在 tempdb 位于独立的物理设备上时才有意义。

当 SQL Server 建立索引时，它必须执行多个读操作以处理各种索引构造步骤。

(1) 遍历所有的数据，构建对应于实际数据每一行的叶行。类似于实际数据和最后的索引，这些内容进入用于临时存储的页。这些中间页不是最终的索引页，而是在每次排序缓冲器已满时临时存储的位置。

(2) 通过这些中间页单独运行，以将它们合并到索引的最终叶级页。

(3) 当填充叶级页时，构建非叶级页。

如果没有使用 SORT_IN_TEMPDB 选项，那么中间页将被写入在其中存储数据库的相同物理文件中。这意味着实际数据的读操作必须与构建过程的写操作竞争。这两种情况造成磁头需要移动到一个不同的位置(读与写)。结果是磁头经常地来回移动——这会花费时间。

另一方面，如果使用 SORT_IN_TEMPDB，那么中间页将被写入 empdb 中，而不是数据库自己的文件。如果它们在单独的物理驱动器上，这意味着在索引构建的读和写操作之间没有竞争。但是要牢记，只有在 tempdb 位于与数据库文件分离的独立物理驱动器上，这才会有效；否则，只是名义上发生改变，而 I/O 的竞争仍然是个问题。

提示：

如果要使用 SORT_IN_TEMPDB，那么确保在 tempdb 中有用于支持大文件的足够空间。

10. ONLINE

如果将这个选项设置为 ON，那么它将强制表对于一般的访问保持有效，并且不创建任何

www.TopSage.com

阻止用户使用索引和/或表的锁。默认情况下，全索引操作将获得所需的锁(最终得到表锁)，以便对表进行完全和有效的访问。然而，副作用是这将会阻止用户(这是矛盾的；一方面可能正在建立索引以使数据库更为有用，但是同时又使表变得不可用)。

现在，您可能会想："如果时时这样设置，那么用户就不会受影响，这似乎是个不错的主意。"实际上这是非常糟糕的想法。要记住，任何类似的索引构造都可能是 I/O 密集的操作，因此它会或多或少地影响用户。而且在构建索引时为了不妨碍任何用户，需要许多额外的系统开销。如果在构建索引时，能让 SQL Server 自由控制表，那么将能更快地构建索引，而且由于构建而影响系统的总时间将更少。

注意：

只有 SQL Server 的企业版支持 ONLINE 索引操作。在其他的版本中，也能够执行带有 ONLINE 指令的索引命令，但是它将被忽略。因此在使用较低版本的 SQL Server 时，如果使用 ONLINE 并且发现用户仍然被索引操作阻止在外，不要对此感到惊讶。

11. ALLOW ROW/PAGE LOCKS

这是一个比 ONLINE 更长期的指令，而且也是个非常高级的主题。出于本书的目的以及到目前为止对锁的讨论，这里只作一个相当简单的解释。

到现在为止，本书中不断地使用"锁"这个术语。正如前面所解释的那样，这在某种避免数据完整性冲突的占位符。这里的 ALLOW 设置用于用于确定索引是否允许行锁或页锁。这可以归入极端性能调整的主题。

12. MAXDOP

该选项用于为构建索引覆盖关于最大并行度的系统设置。因为并行不在本书的讨论范围之外，所以这里将简要介绍它。

简而言之，并行度是指将有多少进程用于一个数据库操作(在这种情况下，该操作是索引构造)。有一个称为最大并行度的系统设置，允许限制每个操作中的处理器数。索引创建选项中的 MAXDOP 选项允许将并行度设置为高于或者低于基本系统设置，只要合适就行。

13. ON

SQL Server 允许通过使用 ON 选项将数据和索引单独存放。从以下观点来看，这是很不错的。

- 索引需要的空间可以分散到其他的驱动器中。
- 用于索引操作的 I/O 不会加重物理数据检索的负担。

还有更多与之相关的内容，但是都涉及非常高级的主题。这些内容与数据和使用关系密切，所以在本书的范围内不作讨论。

9.3.2 创建 XML 索引

XML 索引是 SQL Server 2005 中的新增功能。对类似 XML 的非结构化的内容进行索引是许多人尝试解决的问题，但是几乎没有人真正成功过，而如今却有了这样一个优秀的存储 XML 数据的工具，那应该感到庆幸才是。接下来就讲述关于 XML 索引的所有内容。

www.TopSage.com

这又是一个“先有鸡还是先有鸡蛋”的问题，因为到目前为止在本书中还没有真正讨论过 XML。尽管如此，还是更多地将其看作索引主题，而不是 XML 主题。实际上，除了 IGNORE_DUP_KEY 和 ONLINE 外，XML 创建语法支持在前面的 CREATE 语句中所看到的所有相同选项。

与迄今为止所看到的关系型数据不同，XML 往往是相对非结构化的数据。它利用标记来标识数据，并且可以与模式关联，给基于 XML 的数据提供类型和验证信息。XML 的非结构化特性需要“导航”或者“路径”信息的概念，以在 XML 文档中查找数据“节点”。在另一方面，索引尝试提供数据的特定结构和顺序——这在一定程度上有冲突。

在 SQL Server 中，可以在类型为 XML 的列上创建索引。这样做的主要要求如下。

- 在包含需要索引的 XML 的表上必须具有群集索引。
- 在创建“辅助”索引之前(稍后将有更多介绍)，必须先在 XML 数据列上创建“主”XML 索引。
- XML 索引只能在 XML 类型的列上创建(而且 XML 索引是可以在该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——不能在视图上创建索引。

1. 主 XML 索引

在 XML 索引上创建的第一个索引必须声明为“主”索引。当创建主索引时，SQL Server 创建一个新的群集索引，这个群集索引将基表的群集索引和来自任何指定的 XML 节点的数据组合在一起。

2. 辅助 XML 索引

这里没有任何特别之处——非常类似于指向群集索引的群集键的非群集索引，辅助 XML 索引以很相似的方法指向主 XML 索引。一旦创建了主 XML 索引，就能在 XML 列上创建多达 248 个以上的 XML 索引。

9.3.3　随约束创建的隐含索引

这里称这个索引为“意外的索引”。这并不是说索引不应当在那里——如果需要创建索引的约束，那么它肯定会在那里。只是已经多次看见这种糟糕情况：系统中只有索引是以这种方式创建的。通常，这说明了系统管理员和/或设计者几乎忘记了索引的概念。

然而，您还会在这方面看到另一奇怪现象——管理员或设计者知道如何创建索引，但并不真正知道如何说明系统中有什么索引，以及它们的作用是什么。重复的索引就是这种情况的典型代表。只要索引有不同的名称，SQL Server 就会非常乐于为您创建索引。

当向表中添加如下两种约束之一时，就会创建隐含索引。

- 主键约束
- 唯一约束(也称为替换键)

到目前为止，因为已经学习了大量的 CREATE 语法，所以就不再赘述——但要注意，当创建一个索引作为约束的隐含索引时，除{CLUSTERED|NONCLUSTERED}和 FILLFACTOR 外，所有选项都不允许使用。

www.TopSage.com

9.3.4 在稀疏列和地理空间列上创建索引

这些内容肯定是超出入门书籍的范畴了，但事实是可在稀疏列和地理空间类型数据上创建特殊索引。因此，如果要应用这些特殊类型的列，就需要将此谨记在心。

9.4 明智地选择—— 在何时何地使用何种索引

到现在，您可能在心想："我将始终选择创建群集索引！"这样想有很多很好的理由。但要记住，也有一些理由使得不一定这么做。

选择包含何种索引以及不包含何种索引是一个困难的过程，而且，如果那样做还不够，就必须对需要的索引类型作出一些决定。由于只能有一个群集索引，后一决策既较容易又较困难。这意味着必须明智地选择以最有效地使用它。

9.4.1 选择性

索引，特别是非群集索引，主要在索引中有相当高级别的选择性的情况下是有益的。所谓选择性，指的是列中唯一值的百分比。列中唯一值的百分比越高，选择性就越高，从而索引的益处就越大。

如果回顾关于非群集索引的章节——特别是关于非群集索引和群集索引比较的章节——那么您将回忆起在非群集索引中的查询实际上只是开始。要找到真正的数据，仍需要对群集索引再执行一次循环遍历。甚至使用堆上的非群集索引，仍然需要执行多个单独的物理读操作。

如果非群集索引中的一个查找将要在群集索引上产生多个额外的查找，那么进行表扫描可能更好。这里可能产生的影响实际是非常惊人的。如果被索引列中的唯一性达不到 90%～95%，那么考虑由非群集索引创建的循环过程是不值得的。

这对群集索引几乎没有影响，因为一旦处于数据范围的起点处——唯一的或非唯一的——就已经到达目的地。不需要再读额外的索引页。另外，群集索引很可能拥有其他事物，可以有更大用处。

提示：

选择性规则的一个例外与外键有关。如果表中有一列是外键，那么在该列上有一个索引，这很可能是有益的。为什么是外键而不是其他列？外键常常是与它们引用的表连接的目标。不管选择性如何，索引在连接性能方面是非常有帮助的，因为它们允许合并连接。合并连接从每个表中获取一行并比较，查看它们是否和连接条件(连接的依据)匹配。因为两个表中的相关列上存在索引，所以对两个行的查找是非常快的。

这里的要点是选择性不是一切，但它是要考虑的重要问题。如果讨论的列不是外键，那么在必须考虑的问题方面，选择性问题几乎仅次于像"多长时间使用一次？"这样的问题。

9.4.2 注意成本

记住，虽然索引在读数据时可提高性能，但是在修改数据时，它们实际上花费很高。索引没有通过魔法来维护。每次对数据进行修改时，任何与该数据相关的索引也需要更新。

www.TopSage.com

当插入新行时，必须将新的条目放入表上的每一个索引中。另外也要记住，当更新行时，是将该操作作为删除和插入进行处理——索引也必须被更新。但是等等，还有更多内容！当删除记录时——必须再次更新所有索引——而不只是数据。对于每个创建的索引，意味着创建了一组必须更新的条目。

顺便说一句，此处提及的条目使用复数形式——不只是一个。记住，平衡树有多个级别。每次对叶级进行修改时，就可能发生页拆分，而且也必须修改一个或者多个非叶级页，以便引用正确的叶级页。

有时——实际上是经常——要做的事情是不创建额外的索引。有时候，最好是基于对系统而言至关重要的事务选择索引，并且使用讨论中的表。事务的代码中是否有 WHERE 子句?它使用了什么列？需要进行排序吗？

9.4.3　选择群集索引

记住，只可以有一个群集索引，所以需要明智地选择它。

默认情况下，主键是和群集索引一起创建的。这通常是个不错的选择，但并不总是如此(实际上，在有些情况下，这会带来严重的危害)，并且如果这样做，那么将不能在其他任何地方使用群集索引。这里的要点在于不要接受默认方式。在定义主键时要考虑一下——确实想要它作为群集索引吗?

如果确实要作改变——也就是说，不想声明为群集索引，那么在创建表时，只需添加 NONCLUSTERED 关键字。例如：

```
CREATE TABLE MyTableKeyExample
(
 Column1 int IDENTITY
   PRIMARY KEY NONCLUSTERED,
 Column2 int
)
```

一旦创建了索引，改变它的唯一方法是删除和重建它，所以需要一开始就做对。

要记住，如果改变了群集索引所在的列，那么 SQL Server 将需要对整个表完全重新排序(记住，对于群集索引，表的排列顺序和索引顺序是相同的)。现在，考虑一个 5 000 字符宽的表，而且该表有 100 百万行——这是必须重新排序的非常多的数据。根据这种情况，应该会想到如下一些问题。

- 它将要花费多长时间?可能是很长的时间，而且实际上没有好的方法来估计这种时间。
- 是否有足够的空间?为了在群集索引上执行重新排序，额外需要的平均空间量将为表已经占用的空间量的 1.2 倍(工作空间加上新的索引)。如果在处理大型的表，那么这可能变成非常巨大的空间量——确保有足够的空间来操作。顺便说一句，所有这些活动将在数据库中发生——因此这也会受到为数据库设置的最大尺寸和增长选项的影响。
- 应当使用 SORT_IN_TEMPDB 选项吗？如果 tempdb 位于与主数据库不同的物理阵列上，并且它有足够的空间，那么答案很可能是肯定的。

www.TopSage.com

1. 正面观点

如果讨论的列常作为范围查询的对象，那么群集索引对于这类查询是很有用的。这类查询通常使用 BETWEEN 语句或者<or>符号。使用 GROUP BY 以及利用 MAX、MIN 和 COUNT 聚合函数的查询也是使用范围和偏好群集索引的查询的重要示例。群集索引适合用于此处，是因为搜索可以直接到达物理数据中的特定点，可一直读数据，直到到达范围的末端，然后停止。这种方法非常有效。

当想要数据基于群集键排序(使用 ORDER BY)，群集也是极好的方法。

2. 反面观点

但在两种情况下，您可能不想创建群集索引。第一种情况很明显——当有更好的位置来使用它时。听上去这有点重复，但是不要因为列看上去适合作群集索引就将它用作群集索引(主键是常见的罪魁祸首)——要确定没有更适合的其他列。

尽管如此，更加不适合使用群集索引的另一情况是：在将要以非连续的顺序进行大量插入时。还记得页拆分的概念吗？这里会进行页拆分，并且会消耗大量时间。

假设这样的场景：您正在创建一个记账系统。想要在交易文件中将交易号作为主键，但是也希望交易号在一定程度上能够指示交易类型(它实际上有助于会计人员排除故障)。因此，您想出了一种方案——在所有事务上添加前缀，表明它们来自什么子系统。它们类似于如下所示：

```
ARXXXXXX        Accounts Receivable Transactions
GLXXXXXX        General Ledger Transactions
APXXXXXX        Accounts Payable Transactions
```

其中，XXXXXX 是连续的数值。

这看起来像是一个好主意，所以实现它，并且让主键继续作为群集索引。

初看起来，这一设置非常好。您将会有唯一的值，而且会计也因为能够根据交易号来推断交易来自哪里而感到高兴。因为群集索引常用于查询交易 ID 的范围，所以其看起来是有意义的。

如果真是那么简单就好了。稍微考虑一下插入的操作。使用群集索引时，最初有一个好的机制来避免大量页拆分的系统开销。当插入一条新记录时，该记录将跟在表中最后一个记录后面，即使发生页拆分，也只有该记录被移至新页——SQL Server 不必尝试移动任何旧的数据。不过，现在已陷入麻烦中。

总分类账(General Ledger)插入的新记录将接续在文件的末端(按照字母排列顺序，GL 是在后面，而且号码是连续的)。但是 AR 和 AP 交易有大问题——它们将执行非连续的插入。当 AP000025 插入而且在页上没有空间的时候，SQL Server 将在表中看到 AR000001，而且知道它不是连续的插入。在插入 AP000025 之前，来自旧页的一半记录将被复制到新页。

这样做的系统开销可能会令人感到惊讶。要记住正在处理的是群集索引，而群集索引是数据。数据是以索引顺序排列的。这意味着，当把索引移动到新页时，也在移动数据。现在假设正在典型的 OLTP 环境下运行这个记账系统(记账系统最具 OLTP 特征)，其中有许多数据录入人员正尽可能快地输入供应商发票或者客户订单。系统将会经常发生页拆分，而且每次进行页拆分时，该表的用户将在系统移动数据时出现短暂停顿。

www.TopSage.com

幸运的是，有一些方法可避免出现这种情形。

- 选择在插入时是连续的群集键。可以为此创建一个标识列，或者也可以使用另一个列，该列对于任何输入交易来说(不管来自哪个系统)在逻辑上都是连续的。。
- 选择不在这个表上使用群集索引。对于类似这里的情形来说，这通常是最好的选择，因为在堆上非群集索引中的插入一般比在群集键上的插入更快。

提示：

尽管前面提到要倾向于使用群集键以避免页拆分，但也要认识到这会有开销。连续的群集键的一个不利方面是并发性(两个或者更多的人试图同时访问同一对象)。因此，所要做的是权衡想要干什么、正在做什么，以及在其他方面将要付出的代价。

这可能是为什么我如此深入研究事情是如何进行的一个最好例子。在明白是否使用了正确的索引之前，需要考虑事情将如何完成。

9.4.4　列顺序问题

仅仅因为索引中有两个列，这不能说明索引对于任何引用其中一列的查询是有用的。

如果查询中使用了索引中列出的第一个列，那么可考虑使用该索引。好的方面是不必在每一列上一对一匹配——只需第一个列匹配。当然，(按照顺序)匹配的列越多，越好，但只需通过第一个列就可确定“不要使用”某索引。

可以这样考虑一下。假设在使用电话本。所有项都按先姓后名的方式进行索引——如果只知道要通电话的人的名是 Fred，那么这种排列顺序能带来任何好处吗？另一方面，如果只知道他的姓是 Blake，那么索引将可以用来缩小查找范围。

索引结构中较为常见的一种错误是认为一个包含所有列的索引将对任何情况都是有帮助的。实际上，这样做只是将所有数据又存储了一次。如果索引的第一个列没有在查询中的 JOIN、ORDER BY 或者 WHERE 子句中被提及，那么索引将完全被忽略。

9.4.5　修改索引

ALTER INDEX 命令在其用来做什么方面多少有些欺骗性。截至现在，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——该命令与维护有关，而与结构完全不相干。如果需要修改索引的组成，那么只能 DROP(删除)然后 CREATE(创建)索引，或者用 DROP_EXISTING=ON 选项 CREATE(创建)并使用索引。

正如在本章前面所看到的，SQL Server 提供了一个用于控制叶级页可填充到多满的选项，并且如果选择使用，还有另一个选项用来处理非叶级页。遗憾的是，这些选项是主动型的选项。它们只应用一次，然后在需要时，必须通过重建索引并重应用这些选项来再应用它们。

在 9.5 节关于索引维护的内容中，将了解到更多关于在何处以及为什么要使用这一命令的内容，但现在，只需知道将使用像 ALTER INDEX 这样的维护命令作为常规维护程序的一部分。

ALTER INDEX 的语法类似于下面这样：

```
ALTER INDEX { <name of index> | ALL }
   ON <table or view name>
   { REBUILD
```


www.TopSage.com

```
      [ [ 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 } ]
      )
  } [ ; ]
```

其中一些选项与 CREATE INDEX 命令相同，因此这里将略过对这些选项的重新定义。除此之外，相当多的 ALTER 特定选项都是细节性的，且与处理碎片之类的事情有关(稍后将介绍碎片和维护)，或是更为面向数据库管理员的，且通常用以处理非常特殊的问题。不过，这里的核心元素都是常规维护计划的一部分。

下面将从最先的几个参数开始讲述，然后介绍更大的维护计划所需的选项。

1. 索引名

如果想维护一个特定的索引可以指定该索引，或者使用 ALL 表明想要维护与指定的表相关联的所有索引。

2. 表名或视图名

从字面可知，即想要在其上维护的特定对象(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

这是用来修复索引的“工业级强度”的方法。如果使用该选项运行 ALTER INDEX，那么将完全丢弃旧的索引并重新生成新的索引。结果是真正优化的索引，其中所有叶级和非叶级的页都按照定义进行了重新构建(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群集索引，那么也会重新组织物理数据。

默认情况下，页将被重新组织为差两行满。和 CREATE TABLE 语法一样，可以将

www.TopSage.com

FILLFACTOR 设置为 0~100 之间的任何值。该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)。但要记住，在进行页拆分时，数据将对半分布在两个页上——除了定期重建索引外，不能不断地控制填充的百分比。

提示：

要小心使用该选项。一旦开始 REBUILD，在完成索引重建之前，正在使用的索引实际就没有了。依赖该索引的所有查询可能会变得异常的慢(可能会降几个数量级)。对于这类事情，首先需要在离线系统上测试，以了解整个过程将花多少时间。然后，计划在非高峰时段运行(最好有人监控，以确保它在高峰时段来临时已恢复联机状态)。

重建过程可能有许多副作用，因此，重建索引应当由数据库管理员来处理。

4. DISABLE

该选项名副其实，只是方式有点过激。如果该命令的全部作用只是为了让索引离线，直至您决定了进一步要做什么，则它是不错的选择，但它实际上会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)。

您自己极少会使用该选项(您更可能做的是删除索引)——它很可能在 SQL Server 升级过程中或者其他一些古怪的情况下用到。

注意：

如果对表禁用了群集索引，那么也会禁用表。数据仍会保留，但在重建群集索引之前，不能被所有索引(因为它们都依赖于群集索引)访问。

5. REORGANIZE

从开发人员的角度来看，这一选项特别好。如果重新组织索引，就得到了比完全重建索引稍逊一点的完全优化，但这种方法可以联机进行(用户仍能使用索引)。

如果仔细琢磨的话，上面的描述可能会让您想到一个问题——“稍逊一点”到底是指什么。其实是指，REORGANIZE 只在索引的叶级起作用，而不触及非叶级。这意味着未获得完全优化。但是，对于大部分的索引而言，那不是真正产生碎片的地方(尽管可能会发生，并且遇到的情况也不尽相同)。

由于该选项对用户的影响非常小，通常会希望把该工具作为常规维护计划的一部分来使用。后面讨论碎片时将更进一步讨论它。

9.4.6　删除索引

如果在不断地重新分析情况和添加索引，那么也不要忘记删除索引。记住在插入索引上的系统开销。由于在考虑需要的索引时并没对插入的开销太在意，因此也没有考虑过哪些索引是不需要的。总是要自问一下：“可以从中去掉哪些索引？”

删除索引的语法非常类似于删除表的语法。唯一的不同在于需要使用索引所附着的表或视图来限定索引名称：

```
DROP INDEX <table or view name>.<index name>
```


www.TopSage.com

这样就可以删除索引。

9.4.7 从查询计划中获取提示

SQL Server 2008 中提供的一个新功能是查询计划信息中的索引提示，它们将给出查询优化器所认为有用的、但未存在的索引。

9.4.8 使用数据库引擎调整向导

我希望您已经对索引有了足够的了解，而不需要再使用数据库引擎调整向导。不过，它仍是个非常方便的工具。它的工作方式是获取由 SQL Server Profiler(*Professional SQL Server 2008 Programming* 一书中讨论)生成的工作负载文件并对其信息进行考查，确定出对系统最有利的索引和/或分区。

数据库引擎调整向导位于 SQL Server Management Studio 的 Tools 菜单中。也可通过 Windows 的“开始”菜单上的一个单独的程序项来访问它。和大多数其他调整工具一样，不推荐将这个工具作为决定构建何种索引的唯一方法，但在提出一些您可能未考虑到的建议方面，它还是很有用的。

9.5 维护索引

作为开发人员，在产品交付以后，经常会倾向于忘记它。对于许多类型的软件来说，一般都可以很好地逃脱责任——售出产品，然后进行下一个产品或者下一个版本的开发。然而，对于数据库驱动的项目，实际上不可能逃避这种责任，而需要在交付日期后担负起让产品良好工作的职责。

请不要以为这里的意思是您必须为技术支持部门分担工作——本节实际上在讨论更加重要的事情：维护计划。

实际上，就索引的维护而言有以下两个问题需要处理。

- 页拆分
- 碎片

这两个问题都和页密度相关，虽然两者的表现形式在本质上有区别，但是故障排除工具是一样的，因为处理也是相同的。

9.5.1 碎片

前面已经讨论了很多关于页拆分的内容，但是实际上没有涉及到碎片。此处所谈论的不是您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。

当数据库增长、页拆分，然后删除数据时，就会产生碎片。虽然从增长的观点看，平衡树机制在保持平衡方面做得还不错，但在删除数据时，它并没有太多作用。最终，可能出现这样一种情况，其中在这一页上有一个记录，而在那一页上有几个记录——在这种情况中，许多数据页上的数据量只是它们可以保存的总数据量的一小部分。

www.TopSage.com

关于碎片首先会想到的第一个问题是——浪费的空间。前面说过，SQL Serve 每次分配一个区段的空间。如果一个页上只有一条记录，则仍然会分配整个区段。

第二个问题是——散布在各处的记录会造成数据检索时的额外系统开销。为了获取需要的 10 行记录，SQL Server 不是只加载一个页，而是可能必须加载 10 个不同的页来获取相同的信息。并不只是读取行导致了这一结果——SQL Server 必须首先读取该页。更多的页意味着更多的读取工作量。

虽说如此，数据库碎片也有它好的一面——OLTP 系统就喜欢碎片。原因是什么呢？页拆分。没有许多数据的页在插入数据时几乎或完全不用担心页拆分。

所以，大量的碎片意味着较差的读取性能，但是它也意味着极好的插入性能。正如您所预料的，这意味着 OLAP 系统实际不喜欢碎片。

9.5.2　确定碎片和页拆分的可能性

SQL Server 提供了一个特殊的元数据函数 sys.dm_db_index_physical_stats，它有助于确定数据库中的页和区段有多满。然后可以使用该信息作出一些关于如何维护数据库的决策。

sys.dm_db_index_physical_stats 是表值函数(table valued function，第 13 章将作更详细的介绍)。简而言之，这意味着尽管它是个函数，也可像表一样使用它，这表示可在其上附加 WHERE 条件和其他类似的构造。该函数的语法相当简单：

```
sys.dm_db_index_physical_stats (
   { <database id> | NULL | 0 | DEFAULT }
   , { <object id> | NULL | 0 | DEFAULT }
   , { <index id> | NULL | 0 | -1 | DEFAULT }
   , { <partition no> | NULL | 0 | DEFAULT }
   , { <mode> | NULL | DEFAULT }
)
```

注意，它要求用相对非直观的 id 作为输入值，而不是过去常用的逻辑名。幸运的是，SQL Server 提供了大量可返回服务器上或数据库中特定对象的正确 id 的函数。

举例说明一下，假定要从 Sales.SalesOrderDetail 表中获取所有索引信息。由于要求获取表中的所有索引，因此只需要该表的 id——而不是任何单独的索引。由于这是个表值函数，需要编写一个查询从该函数返回结果。例如：

```
DECLARE @db_id SMALLINT;
DECLARE @object_id INT;
SET @db_id = DB_ID(N'AdventureWorks2008');
SET @object_id = OBJECT_ID(N'AdventureWorks2008.Sales.SalesOrderDetail');
SELECT database_id, object_id, index_id, index_depth,
 avg_fragmentation_in_percent,
page_count
FROM sys.dm_db_index_physical_stats(@db_id,@object_id,NULL,NULL,NULL);
```

相比于以前的工具，其输出是更不用加以说明的。

```
database_id object_id  index_id index_depth avg_fragmentation_in_percent page_count
----------  ---------  -------- ----------- ------------------------     ----------
8            610101214 1          3          0.565885206143896            1237
8            610101214 2          3          1.70731707317073             410
```


www.TopSage.com

```
8            610101214  3          2              6.14035087719298              228

(3 行受影响)
```

该函数还提供了大量其他信息，因此可凭兴趣作进一步研究，但这里还是看一下与碎片有关的列。

提示：

有大量的字段说明了有关索引结构的类型(群集索引和非群集索引)、碎片页的数目，以及其他感兴趣的信息。还有一些列的信息是有关高级 SQL Server 内容的，因此如果没有理解该函数提供的所有列，也不必有压力。

表 9-2

统 计 信 息	说　　明
database id	这是对运行函数的数据库的 SQL Sever 内部标识符。NULL 值表示会返回所有数据库的信息。记住，如果选择 NULL 选项，则所有后续参数必须也设置为 NULL。对此参数而言，0 和 DEFAULT 在功能上等同于 NULL
object id	这是对运行函数的表的 SQL Sever 内部标识符。NULL 值表示会返回所有表的信息。记住，如果选择 NULL 选项，则所有后续参数必须也设置为 NULL。对此参数而言，0 和 DEFAULT 在功能上等同于 NULL
index id	这是对运行函数的索引的 SQL Sever 内部标识符。NULL 值表示返回所有索引的信息。注意，如果表是群集表，那么群集索引将总是为 1，如果表是堆，那么堆索引将为 0。非群集索引将为 2 或更大的数字。记住，如果选择 NULL 选项，则 partition_no 必须也设置为 NULL。对此参数而言，－1 和 DEFAULT 在功能上等同于 NULL
parttion no	这个信息有点超出本书的范围，但表和索引可分区到单独的物理存储单元。使用这一参数可标识想要获取特定分区的磁片信息。NULL 值表示将返回所有分区的信息。对此参数而言，1 和 DEFAULT 在功能上等同于 NULL
Mode	这定义了用于生成返回的统计信息的扫描级别。选项包括 DEFAULT、NULL、LIMITED、SAMPLED 或 DETAILED。DEFAULT 和 NULL 在功能上等同于 LIMITED
Logical Scan Fragmentation	通过扫描索引的叶级页检查到的无序页的百分比。它只与群集表上的扫描有关。无序页是指在索引分配映射(index allocation map，IAM)中指示的下一个页不同于叶级页中的下一个页指针指向的页
Extent Scan Fragmentation	该信息表明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。这只意味着索引的叶级页在物理上不是有序的(虽然它们在逻辑上是有序的)，并且表明了这一问题占区段的百分比

现在，问题是一旦得到该信息，该如何使用它呢？答案当然是看情况而定。

www.TopSage.com

使用来自碎片查询的输出，就可正确认识到数据库是否已满、是否有碎片或者介于中间的某个状态(后者更可能是希望看到的情况)。如果运行的是 OLAP 系统，那么会希望看到页已满——而碎片会令人沮丧。对于 OLTP 系统，将希望有相反的结果(尽管是只在这一点上)。

所以，如何处理这个问题？要回答它，需要深入了解索引重建和填充因子的概念。

ALTER INDEX 和 FILLFACTOR

正如在本章前面所看到的，SQL Server 提供了一个控制叶级页有多满的选项，当然，也提供了另一个处理非叶级页的选项。遗憾的是，这些都是主动型的选择——它们只应用一次，然后在需要时，必须通过重建索引并重应用这些选项来再应用它们。

为了重建索引，可以删除索引并再次创建索引(如果要这样做，使用 DROP_EXISTING 选项是一个好主意)，或是运用 ALTER INDEX 命令和 REBUILD 选项。不过要记住前面讨论重建索引时所讲的——除非可使用 ONLINE 选项，否则在重建完成前，重建索引会使该索引(可能是整个表)完全离线。通常，重新组织是更好的选择。

重新组织只会影响索引的叶级(大部分问题很可能都在其中)，并且在此过程中使索引保持联机状态。重建索引可重新构造这些索引中的所有叶级信息，而且重建一个页填充度的基本百分比。如果讨论的索引是群集索引，那么也会重新组织物理数据。遗憾的是，REORGANIZE 并不允许更改一些索引设置，如 FILLFACTOR。

如果索引没有指定 fillfactor，那么页将被重新构造为差两行填充满。和 CREATE TABLE 语法一样，可以将 FILLFACTOR 设置为 0~100 之间的任意值。一旦数据库重新组织完成，这个数字将是页填充的百分比。但要记住，当进行页拆分时，数据将仍然对半分布在两个页之间——除了定期重建索引外，不能不断地控制填充百分比。

当需要调整页密度时，使用 FILLFACTOR。正如前面所讨论过的，较低的页密度(因而具有较低的 FILLFACTOR)对于有许多插入操作的 OLTP 系统而言是理想的——这助于避免页拆分。较高的页密度是 OLAP 系统所期望的(读取较少的页，但是由于很少或几乎没有插入，因此实际没有页拆分的风险)。

如果想重建之前的 Sales.SalesOrderDetail 表中作为主键的填充因子为 65 的索引，则可以采用下列 ALTER INDEX 命令：

```
ALTER INDEX PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID
 ON Sales.SalesOrderDetail
   REBUILD WITH (FILLFACTOR = 100)
```

提示：

现在可以偏离一下主题，讨论一下有关对象名的内容。可注意到，这个名称较长。字符越多，就越可能出现输入错误。对需要的名称作一下权衡是有意义的。对于这个特定的查询，本书的命名约定不包括表名(当引用某个索引时，通常已知道所指的表)，同时省略下划线，使其最终形式为 PKSalesOrderIDSalesOrderDetailID。该名称指出了索引的性质(它支持主键)和涉及的列。不过，要慎用列名。在极少数情况下，如果在索引中出现几个列，这会导致不合规则的索引名。

然后重新运行 sys.dm_db_index_physical_stats 以查看其作用：

www.TopSage.com

```
database_id   object_id   index_id  index_depth   avg_fragmentation_in_percent  page_count
----------- --------- -------  ---------   ----------------------   --------
8             610101214  1        3            0.0810372771474878        1234
8             610101214  2        3            1.70731707317073          410
8             610101214  3        2            6.14035087719298          228
```

(3 行受影响)

这里要注意的最大问题是在 avg_fragmentation_in_percent 中的改变。假定我们正查看高页密度(较强的可读性)，那么希望此数字尽可能地低。该数字实际不会达到 0%，因为 SQL Server 必须处理页和行的大小问题，但会尽量接近该值。

关于 ALTER TABLE REINDEX/REORG 和 FILLFACTOR，需要注意下列几个要点。

- 如果没有提供 FILLFACTOR，那么 ALTER TABLE 将使用以前建立索引时设置的填充因子。如果从未指定过，那么 FILLFACTOR 会使得页填充至差两行满(这对大多数情况而言都太满了)。
- 如果提供了 FILLFACTOR，那么该值成为该索引的默认 FILLFACTOR。
- 虽然 REORGANIZE 可以联机工作，REBUILD 也可以(如果有启用该功能的许可)，但本书强烈建议不要使用它——它会锁定资源并会引起许多问题。顶多在非高峰时执行它。

9.6 本章小结

在 SQL Server 或者任何其他数据库环境中，索引都是一个基本的主题，而且不可轻视。它们有助于获得良好的性能，但也可能导致极差的性能。

关于索引要考虑的几个首要问题如下。

- 群集索引通常比非群集索引快(可以说前者总是比后者快，但是也有一些例外)。
- 仅在将得到高级别选择性的列(也就是说，95%或者更多的行是唯一的)上放置非群集索引。
- 所有的数据操作语言(DML：INSERT、UPDATE、DELETE、SELECT)语句可以通过索引获益，但是插入、删除和更新(记住，它们使用删除和插入方法)会因为索引而变慢。索引有助于查询的查找过程，但是任何修改数据的行为将有额外的工作要做(除了实际数据外，还要维护索引)。
- 索引会占用空间。
- 仅当索引中的第一列和查询相关时才使用索引。
- 索引的负面影响和它的正面影响一样多——了解为什么建立索引，以及为什么只建立需要的索引。
- 索引可为非结构化 XML 数据提供结构化的数据性能，但是要记住，和其他索引一样，这会涉及系统开销。

在考虑索引时，最好问一下自己一些问题，如表 9-3 所示。

www.TopSage.com

表 9-3

问　题	回　答
会对这个表进行大量插入或者修改吗？	如果是，尽量少用索引。这种类型的表通常通过主键的单个记录查找完成修改——这往往是该表上需要的唯一索引。如果插入是非连续的，不考虑使用群集索引
这是一个报表吗?也就是说，这里没有许多插入，但是会以许多不同的方法运行报表吗？	有更多索引是好的。将群集索引确定为频繁使用的、可能会在范围内提取的信息。OLAP 系统中的索引数目通常是 OLTP 环境中所看到的许多倍
在数据上有高级别的选择性吗？	如果是，而且它通常是 WHERE 子句的目标，那么添加索引
已经删除了不再需要的索引吗？	如果没有，为什么不删除？
已建立了维护策略吗？	如果没有，为什么不建立？

9.7 练习

1. 请说出至少两种确定在 AdventureWorks2008 数据库的 HumanResources.Employee 表中有何种索引的方法。
2. 在 AdventureWorks2008 数据库的 Production.ProductModel 表的 ModifiedDate 列上创建一个非群集索引。
3. 删除在练习 2 中创建的索引。

www.TopSage.com

第10章

视　图

到目前为止，所涉及的都是一些基本对象——自身有一定实质性内容的对象。本章将讲述虚拟的对象，了解一下视图的概念。

对视图的使用往往不是过多或者就是不够——很少是正好的。当学完本章以后，应当可以使用视图达到以下目的。

- 为终端用户减少明显的数据库复杂性
- 防止敏感的列被选择，但是仍然提供对其他重要数据的访问
- 给数据库添加额外的索引以提高查询性能——甚至在没有使用索引所基于的视图时也是如此

视图实际上就是一个存储查询。重点是可以混合和匹配来自基本表(或者其他视图)的数据，从而创建在很多方面上像另一个基表那样起作用的对象。可以创建一个简单的查询，仅仅从一个表中选择几列，而忽略其他列；或者也可以创建一个复杂的查询，连接几个表，使得这些表看起来像一个表。

10.1　简单的视图

视图语法的最基本形式是本书前面讲述的几个内容的组合——在第 5 章讲述的基本 CREATE 语句，加上已经使用过多次的 SELECT 语句：

```
CREATE VIEW <view name>
AS
<SELECT statement>
```

当然，上面的语法只是代表了最简语法，但是这对于大多数情况来说已经足够。扩展的语法如下所示：

```
CREATE VIEW [schema_name].<view name> [(<column name list>)]
[WITH [ENCRYPTION] [, SCHEMABINDING] [, VIEW_METADATA]]
AS
```

www.TopSage.com

```
<SELECT statement>
WITH CHECK OPTION
```

下面将分别地讲述上面代码的每一部分，但是，先来看一个极其简单的视图。

试一试 创建一个简单的视图

此处将该视图称为客户电话列表，并且创建其作为第 5 章创建的 Accounting 数据库中的 CustomerPhoneList_vw。

```
USE Accounting
GO

CREATE VIEW CustomerPhoneList_vw
AS
   SELECT CustomerName, Contact, Phone
   FROM Customers
```

注意，在 Management Studio 中执行 CREATE 语句时，它的工作方式类似于前面讲述过的所有其他 CREATE 语句——没有返回任何行。它只是说明了视图已经创建完成：

```
命令已成功完成。
```

现在切换到使用网格视图(如果还未在该视图下的话)，以便于查看多个结果集。然后对视图运行一条 SELECT 语句——像表一样使用它——并且直接对 Customers 表执行另一条 SELECT 语句：

```
SELECT * FROM CustomerPhoneList_vw

SELECT * FROM Customers
```

示例说明

得到的结果集与之前看到的几乎是相同的——实际上，在它们共有的列中，两个结果集是相同的。为了阐明 SQL Server 是如何看待视图上的查询的，下面在逻辑上稍微作一些分解。创建视图时的 SELECT 语句被定义为：

```
SELECT CustomerName, Contact, Phone
FROM Customers
```

所以当运行下列语句时，

```
SELECT * FROM CustomerPhoneList_vw
```

实质上是告诉 SQL Server“把执行 SELECT CustomerName，Contact，Phone FROM Customers 语句时得到的所有的行和列给我”。

这样创建了一个类似“通道”的情形——也就是说，视图没有真正改变任何内容，只是“通过了”所访问的数据的过滤版本。这样做的好处是为终端用户减少了复杂性。在现在这个时代，

www.TopSage.com

有大量工具可使用户工作变得更加简单，这可能看起来也没有什么很了不起的——但是对于用户而言，这的确减少了复杂性。

注意：

要意识到，在默认情况下，对于视图而言并没有特别要做的事情。视图就像在命令行执行的查询那样运行——没有任何的预先优化过程。这意味着在数据的请求和所交付的数据之间增加了额外的一层系统开销，而视图的运行速度总是比直接运行内部的 SELECT 语句要慢。也就是说，视图存在的一个理由——即对于用户而言是安全或者简化的——在需求和系统开销之间寻求平衡以适合特定的情况。

下面用另一个视图说明在隐藏敏感数据方面能做些什么。为此，回到 Accounting 数据库(稍后将创建一个特殊版本的 Accounting 数据库)中的 Employees 表。先看一下如表 10-1 所示的表布局。

表 10-1

Employees
EmployeeID
FirstName
MiddleInitial
LastName
Title
SSN
Salary
HireDate
TerminationDate
ManagerEmpID
Department

美国联邦的法律保护其中的一些信息——所以要只限于“必须知道”的人访问。然而，其他列可供任何人查看。如果想将没有限制的列展示给一些人，但是又不想让他们看见全部表结构或者数据，那该怎么办呢？一种解决方法是用一个单独的表，只包括需要提供的列，如表 10-2 所示。

表 10-2

Employees
EmployeeID
FirstName
MiddleInitial
LastName

www.TopSage.com

(续表)

Employees
Title
HireDate
TerminationDate
ManagerEmpID
Department

虽然从表面看这满足了上述的需求，但是存在很大问题。

- 要使用磁盘空间两次。
- 如果一个表得到更新但是另一个表没有得到更新，那么就存在同步问题。
- 无论何时需要插入、更新或者删除行，都必须进行双重 I/O 操作(必须在两个地方而不是一个地方读和写数据)。

视图为这个问题提供了一个简单和相对适当的解决方案。通过使用视图，数据仅存储一次(在底层表中)——消除了前面的所有问题。不是建立一个完全单独的表，而是建立一个功能几乎一样的视图。

为了确保了解 Accounting 数据库的状态，首先创建一个特殊版本的 Accounting 数据库，以在下一个的示例中使用。为此，将 CChapter10AccountingDBCreate.sql 文件(以源代码的形式提供)加载到 Management Studio 并且运行它。然后添加下面的视图到 Accounting Chapter10 数据库中：

```
USE AccountingChapter10
GO

CREATE VIEW Employees_vw
AS
SELECT   EmployeeID,
         FirstName,
         MiddleInitial,
         LastName,
         Title,
         HireDate,
         TerminationDate,
         ManagerEmpID,
         Department
FROM Employees;
```

现在准备让任何人都能直接或者间接访问 Employees 表中的数据。拥有“必须知道”权限的用户现在可以直接访问 Employees 表，但是仍拒绝其他用户的访问。不过，没有“必要知道”权限的用户可以访问 Employees_vw 视图。如果他们想利用 Employees_vw 视图，那么可以像对待表一样地操作。

```
SELECT *
FROM Employees_vw
```


www.TopSage.com

注意：

这实际就涉及到命名约定的一个棘手问题。因为一直在使用_vw 后缀，所以可以很容易地看出这是视图而不是表。但有时希望稍微作些隐藏，所以可能想故意省去_vw。这样做就意味着必须使用不同的名称(Employees 已是基表的名称)，但是令人惊讶的是，如果这么做的话，很多用户都不知道在视图和表之间是有区别的。

作为过滤器的视图

这可能是本书中最简短的一节。为什么？因为没有比这更简单的内容。

前面已经讲过如何创建一个简单的视图——就是使用一个简单的 SELECT 语句。那么怎样过滤查询的结果呢？答案是使用 WHERE 子句。使用 WHERE 子句对于视图来说并没有什么区别。

考虑上节中的 Employees_vw 视图，并在此基础上创建一个只包括当前雇员的列表的视图。为此，实际上只需要作两个改动。

首先，必须过滤出不再为公司工作的雇员。当前的雇员有终止日期吗？可能没有，所以如果限制结果是 TerminationDate 为 NULL 的行，那么就得到要找的内容。

第二个改动说明了关于视图像查询一样工作的另一个简单要点——包含在 WHERE 子句中的列不需要包含在 SELECT 列表中。在这里，在当前雇员的结果集中包含终止日期是没有任何意义的。

试一试　使用视图来过滤数据

根据以上讲的两点，通过对原来的视图稍作改动来创建一个新的视图：

```
CREATE VIEW CurrentEmployees_vw
AS
SELECT EmployeeID,
       FirstName,
       MiddleInitial,
       LastName,
       Title,
       HireDate,
       ManagerEmpID,
       Department
FROM Employees
WHERE TerminationDate IS NULL
```

除了名称的变化以及添加了 WHERE 子句外，可注意到这里还从 SELECT 列表中删除了 TerminationDate 列。

要检验这是如何工作的，可以直接对 Employees 表运行 SELECT 语句以及限制 SELECT 列表为所需要的内容：

```
SELECT EmployeeID,
       FirstName,
```


www.TopSage.com

```
        LastName,
        TerminationDate
FROM Employees
```

这就返回了来自整个表的所有行的几列：

```
EmployeeID     FirstName        LastName       TerminationDate
------------   -------------    ------------   -----------------------
1              Joe              Dokey          NULL
2              Peter            Principle      NULL
3              Steve            Smith          1997-01-31
4              Howard           Kilroy         NULL
5              Mary             Contrary       1998-06-15
6              Billy            Bob NULL

(6 行受影响)
```

现在检查一下视图：

```
SELECT EmployeeID,
       FirstName,
       LastName
FROM CurrentEmployees_vw
```

结果集稍小一些：

```
EmployeeID     FirstName    LastName
-------------  ----------   ----------------
1              Joe          Dokey
2              Peter        Principle
4              Howard       Kilroy
6              Billy        Bob

(4 行受影响)
```

和第一个选择相比，第二个选择的结果中少了几个人——这正是想要的结果。

示例说明

正如之前讨论的那样，视图实际上只是一个用户不可见的 SELECT 语句，所以用户可以忽略 SELECT 语句所说明的，而只将其产生的结果看作像表一样——可以把这个比作在第 7 章讨论的派生表。因为数据在通过名称引用视图之前被筛选过，所以这里的查询甚至不需要考虑数据问题(视图已经筛选过滤器)。

10.2 更加复杂的视图

尽管在这里使用了“复杂”这个词——不过别让这个词吓到您。从某种程度上讲，视图中最难以理解的内容仍然比 SQL 中大多数的其他内容简单。

这里使用更加复杂的视图所做的实际上只是添加连接、小结，或是列的重命名。

视图最常见的一个用途可能是展开数据——也就是说，本章开头所说的消除复杂性。可以

www.TopSage.com

想象一下为管理提供一个视图，以便更加容易地检查销售信息。虽然本人无意于冒犯正在阅读本书的管理人员，但是事实上自己写复杂查询的管理人员仍然是相当少的——即使在信息时代。

为了举例说明，再次使用 AdventureWorks2008 数据库。管理人员希望执行简单的查询以了解订购了哪些订单，是订购哪些零件的，以及使用哪个账号订购的。所以，下面的示例创建了一个能在其上执行非常简单的查询的视图——要记住是在 AdventureWorks2008 中创建了这个视图：

```
USE AdventureWorks2008
GO

CREATE VIEW CustomerOrders_vw
AS
SELECT sc.AccountNumber,
       soh.SalesOrderID,
       soh.OrderDate,
       sod.ProductID,
       pp.Name,
       sod.OrderQty,
       sod.UnitPrice,
       sod.UnitPriceDiscount*sod.UnitPrice*sod.OrderQty AS TotalDiscount,
       sod.LineTotal
FROM Sales.Customer AS sc
INNER JOIN Sales.SalesOrderHeader AS soh
       ON sc.CustomerID = soh.CustomerID
INNER JOIN Sales.SalesOrderDetail AS sod
       ON soh.SalesOrderID = sod.SalesOrderID
INNER JOIN Production.Product AS pp
       ON sod.ProductID = pp.ProductID;
```

现在执行 SELECT 语句：

```
SELECT *
FROM CustomerOrders_vw;
```

这样得到一些行——超过 121 000 行——但是还得到了对于普通的管理人员来说更容易理解和区分的信息。而且，不需要怎么培训，管理人员(或者任何可能的用户)就可获取他们要查找的数据：

```
SELECT AccountNumber, LineTotal
FROM CustomerOrders_vw
WHERE OrderDate = '01/07/2002';
```

用户不需要知道如何执行一个四个表的连接——那是隐藏于视图中的。他们只需要少量的技巧(以及相关的少量想像)就可完成这个工作。

```
AccountNumber LineTotal
------------- ---------------------------------------
AW00014937    3578.270000
AW00018710    3578.270000
AW00025713    699.098200
AW00020558    699.098200

(4 行受影响)
```


www.TopSage.com

但是，可以使得查询更加有目的性。例如，仅仅希望视图返回昨天的销售情况。只需对查询作很小的改变：

```
USE AdventureWorks2008
GO

CREATE VIEW YesterdaysOrders_vw
AS
SELECT sc.AccountNumber,
       soh.SalesOrderID,
       soh.OrderDate,
       sod.ProductID,
       pp.Name,
       sod.OrderQty,
       sod.UnitPrice,
       sod.UnitPriceDiscount * sod.UnitPrice * sod.OrderQty AS
       TotalDiscount,sod.LineTotal
FROM Sales.Customer AS sc
INNER JOIN Sales.SalesOrderHeader AS soh
       ON sc.CustomerID = soh.CustomerID
INNER JOIN Sales.SalesOrderDetail AS sod
       ON soh.SalesOrderID = sod.SalesOrderID
INNER JOIN Production.Product AS pp
       ON sod.ProductID = pp.ProductID
WHERE CAST(soh.OrderDate AS Date) =
       CAST(DATEADD(day,-1,GETDATE()) AS Date);
```

因为 AdventureWorks2008 数据库中的所有日期都过时了，以至于这个视图不会返回任何数据，所以先修改一些现有行来进行测试。

```
USE AdventureWorks2008

UPDATE Sales.SalesOrderHeader
SET OrderDate = CAST(DATEADD(day,-1,GETDATE()) AS Date),
   DueDate = CAST(DATEADD(day,11,GETDATE()) AS Date),
   ShipDate = CAST(DATEADD(day,6,GETDATE()) AS Date)
WHERE Sales.SalesOrderHeader.SalesOrderID BETWEEN 43659 AND 43662;
```

这是个相当简单的更新语句，它把一些订单的日期重新设置为与昨天(即运行该更新语句的那天的前一天)相对的关系。正如您所猜测的，GETDATE()函数返回当前日期。稍后将对其他一些内容作讨论。不过现在，您只要确信需要运行这些语句，这样在 AdventureWorks2008 中得到一个值，这个值将出现在视图中。可以从 Management Studio 中看到类似下面的一个结果：

```
(31 465 行受影响)
```

提示：

要知道，只有使用的是 Management Studio 的“以网格显示结果”模式，上面列出的消息才出现在“消息”面板中。如果使用的是文本模式，那么将出现在 Results 表中。

OrderID 可能不同，但是其他的内容应当是确定的。

现在对视图运行一个查询来查看得到的结果：

www.TopSage.com

```
SELECT AccountNumber, SalesOrderID, OrderDate FROM YesterdaysOrders_vw;
```

可看见确实显示了 4 个订单，由于每个订单都有一些行项，因此最终将得到 121 317 行。

```
AccountNumber SalesOrderID  OrderDate
------------- ------------  -----------------------
AW00000676    43659         2007-09-30 00:00:00.000
AW00000676    43659         2007-09-30 00:00:00.000
AW00000676    43659         2007-09-30 00:00:00.000
AW00000676                  43659 2007-09-30 00:00:00.000
...
...
AW00000227                  43662 2007-09-30 00:00:00.000
AW00000227                  43662 2007-09-30 00:00:00.000

(51 行受影响)
```

DATEADD 和 CAST 函数

尽管这里的连接比到目前为止前面所做的大部分连接都要大，但是仍然是相当简单的。连续添加表，并将每个新表中的列连接到已命名的表中的匹配列。注意，这些列不必具有相同的名称——它们只是必须有互相关联的数据。

因为这是一个相对复杂的连接，接下来就看看在支持这个视图的查询中做了什么。

最有趣的地方是其中的 WHERE 子句：

```
WHERE CAST(soh.OrderDate AS Date) =
      CAST(DATEADD(day,-1,GETDATE()) AS Date)
```

虽然这是一个简单的比较，但是还是使用了一些函数来得到所需结果。

我们想将 SalesOrderHeader 表中的 OrderDate 与 GETDATE(今天的日期)减一天直接进行比较——其中减操作正是 DATEADD 函数的工作。DATEADD 可以增加(如果使用负数就是减少)任意长的时间。只要告诉它想要在哪个日期上操作，以及想要在它上面添加的时间单位(天、星期、年、秒等)。从表面上看，应当能以 GETDATE()获取今天的日期，然后使用 DATEADD 以减一天。问题是 GETDATE()返回 datetime 数据类型，因此包含了当天的当前时间，所以将只能得到这样的行，其发生在前一天的相同时间，并且具体时间精确到 3.3333 毫秒(datetime 数据类型的精度)——这不是一个可能的匹配。所以需要多做一项操作，在比较之前使用 CAST 函数将等式两边的日期转换为时间精度较低的数据类型。这样，视图将显示前一天的任意时间的销售情况。

注意：

这一代码在 SQL Server 2005 和更早版本中并不能工作，因为 Date 数据类型是 SQL Server 2008 新提供的。在旧版本的 SQL Server 中，使用 CONVERT 函数将日期改为不包括时间的 varchar 类型，然后进行比较。

使用视图更改数据——不使用 INSTEAD OF 触发器

如前所述，从使用的角度看，视图的工作方式与表很相像(显然，创建它们的方式有相当大

www.TopSage.com

的不同)。不过，还是有一些不同之处。

尽管令很多人惊讶，但是确实可以成功地对视图执行 INSERT、UPDATE 以及 DELETE 语句。不过，在通过视图改变数据时，要记住以下一些内容。

- 如果视图包含连接，在大多数情况下，除非使用 INSTEAD OF 触发器，否则不能对数据执行 INSERT 或者 DELETE 操作。有时(只要只更新来自单个表的列)，UPDATE 可以不使用 INSTEAD OF 触发器来工作，但是这需要一些规划，否则很快会遇到问题。
- 如果视图仅仅引用单个表，那么在表中所有必需字段都在视图中或者有默认值的情况下，可以通过使用视图而不用 INSTEAD OF 触发器来对数据执行 INSERT 操作。即使对于单个表的视图来说，如果有一列没有出现在不具有默认值的视图中，那么如果想要允许 INSERT 操作，必须使用 INSTEAD OF 触发器。
- 可在有限范围内限制是否可以在视图中插入或更新的内容。

本书已经多次提到了 INSTEAD OF 触发器。这里的问题是 INSTEAD OF 触发器比较复杂而且还没有比较深入地讨论过触发器。这是 SQL Server 中经常出现的情况，即碰到了“先有鸡还是先有蛋”的老问题。由于 INSTEAD OF 触发器与视图相关，因此需要讨论 INSTEAD OF 触发器，但是在理解可在其上创建 INSTEAD OF 触发器的两个对象(表和视图)之前，本节又并不准备讨论 INSTEAD OF 触发器。

本章的处理方法就是以过去的方式处理视图——假定还没有像 INSTEAD OF 触发器这样的事物。尽管本章中不涉及 INSTEAD OF 触发器的详细内容，但是要知道什么时候必须使用 INSTEAD OF 触发器。然后在第 15 章简要地讲述 INSTEAD OF 触发器时，回过头来更全面地讲述这些问题。

提示：

在此，本书将提供一点背景知识——INSTEAD OF 触发器是一种特殊类型的触发器，它实际上是“代替”任何可能激活触发器的语句运行。结果是它能知道语句所能完成的工作，然后在触发器上对如何解决任何冲突或者可能出现的其他问题作出决策。它是非常强大的，但也是相当复杂的，这是目前推迟讲述的原因。

1. 用连接的数据处理视图的变化

如果视图有多个表，那么在很多情况下使用视图来修改数据是不允许的——除非使用 INSTEAD OF 触发器。因为这在键的排列上造成了一些多义性，所以当有多个表时，Microsoft 会默认锁定。为了解决这个问题，可以使用 INSTEAD OF 触发器来检查改变的数据，并明确地告诉 SQL Server 希望利用它做什么。

2. 必需字段必须在视图中出现或者具有默认值

在默认情况下，如果使用视图来插入数据(内部查询中必定有一个单个表的 SELECT 操作或者至少必须限制插入只影响到一个表并且使所有必需的列出现)，那么必须能为所有的必需字段(不允许为空的字段)提供一些值。注意“提供一些值”并不意味着它们必须出现在 SELECT 列表中——默认值就可很好地提供所需的值。不过要认识到任何没有默认数值以及不接受 NULL 数值的列需要出现在视图中，以通过视图执行 INSERT 操作。解决这个问题唯一的方法

www.TopSage.com

当然是使用 INSTEAD OF 触发器。

3. 通过 WITH CHECK OPTION 限制插入到视图中的内容

WITH CHECK OPTION 是 SQL Server 中鲜为人知的功能之一。规则很简单——为了通过使用视图更新或者插入数据，结果行必须符合要求以显示在视图结果中。重申一下，插入或者更新的行必须满足视图中 SELECT 语句的 WHERE 条件。

试一试　使用 WITH CHECK OPTION

为了对 WITH CHECK OPTION 进行说明，继续以 AdventureWorks2008 数据库为例，创建只显示 Oregon 的 Portland 地区地址的视图。因为在 Address 表中只有有限的字段，所以要利用 PostalCode 指出地区地址具体所在。数据库提供了足够的信息，允许基于 PostalCodes(Portland 地区的邮编以 970、971 和 972 开始，而 Washington 的 Vancouver 地区的邮编是 98660~98699)进行过滤，并在视图中应用。

```
CREATE VIEW PortlandAreaAddresses_vw
AS
SELECT AddressID,
       AddressLine1,
       City,
       StateProvinceID,
       PostalCode,
       ModifiedDate
FROM Person.Address
WHERE PostalCode LIKE '970%'
   OR PostalCode LIKE '971%'
   OR PostalCode LIKE '972%'
   OR PostalCode LIKE '986[6-9]%'
WITH CHECK OPTION;
```

对这个视图运行 SELECT *，并返回约 792 行：

```
AddressID AddressLine1          City         StProvID  PostalCode  ModifiedDate
--------  --------------------  ---------    --------  ---------   ----------
22        636 Vine Hill Way     Portland       58        97205      2001-06-24
312       1286 Cincerto Circle  Lake Oswego    58        97034      2002-01-23
322       1 Mt. Dell Drive      Portland       58        97205      2002-01-24
...
...
29792     5186 Oeffler Ln.      Beaverton      58        97005      2003-03-20
29822     2613 West I St.       Beaverton      58        97005      2003-12-25
29856     1132 Plymouth Dr.     Lake Oswego    58        97034      2004-04-07

(792 行受影响)
```

现在尝试通过使用视图来更新某一行——将 PostalCode 设置为不是以 97 或 98 开头：

```
UPDATE PortlandAreaAddresses_vw
SET PostalCode = '33333' -- it was 97205
WHERE AddressID = 22;
```


www.TopSage.com

SQL Server 很快做了一个结论。

```
消息 550，级别 16，状态 1，第 1 行
试图进行的插入或更新已失败，原因是目标视图或者目标视图所跨越的某一视图指定了 WITH CHECK OPTION，
而该操作的一个或多个结果行又不符合 CHECK OPTION 约束。
语句已终止。
```

示例说明

WHERE 子句过滤视图中的内容，以便只显示以 970、971 或者 972 开头或在 9866~9869 之间的邮编，并且 WITH CHECK OPTION 说明任何 INSERT 或者 UPDATE 语句必须满足 WHERE 子句条件(而(33333)这个邮编不满足这个条件)。

因为更新不能满足 WHERE 子句条件，所以被终止了；但是，如果在基表中更新该行：

```
UPDATE Person.Address
SET PostalCode = '33333' -- it was 97205
WHERE AddressID = 22;
```

那么 SQL Server 会相当友好：

```
(1 行受影响)
```

该限制仅应用于视图——而不应用于底层表。这在极少数情况下是非常方便的。假设这样一种情形：允许一些用户在表中插入或者更新数据，但是要求更新或者插入的数据满足特定的条件。可以通过添加 CHECK 约束到底层表中应对这一限制——但是这可能并不总是一种理想的解决方案。

现在假设增加第二个要求——允许其他一些用户将数据插入到表中，即使不满足条件也可以。这时，CHECK 约束就不能区分这两种用户了。通过使用带 WITH CHECK OPTION 的视图，就可以让受限的用户使用视图，而让未受限的用户使用基表或者没有这种限制的视图。

注意，这对于 INSERT 操作也一样。运行违反 WHERE 子句的 INSERT，将碰到运行 UPDATE 时同样的终止错误。

10.3 使用 T-SQL 编辑视图

在使用 T-SQL 编辑视图时要记住，这是在完全替换现有的视图。使用 ALTER VIEW 语句和 CREATE VIEW 语句的区别有以下几点。

- ALTER VIEW 期望找到一个已有的视图，而 CREATE 则不是。
- ALTER VIEW 保留了视图上已经建立的任何权限。
- ALTER VIEW 保留了任何依赖信息。

第二点是非常重要的。如果先执行 DROP，然后使用 CREATE，那么其效果与使用 ATLTER VIEW 语句几乎是一样的。问题是需要完全重建关于允许或不允许谁使用这个视图的权限。

www.TopSage.com

10.4 删除视图

这是最简单的，通过下列语句：

```
DROP VIEW <view name>, [<view name>,[ ...n]]
```

就完成了视图的删除。

10.5 在 Management Studio 中创建和编辑视图

对于实际不清楚他们在做什么的人们而言，在 Management Studio 中创建和编辑视图的功能是相当酷的。创建视图是很容易的，并且并不需要知道很多关于查询的内容。

要了解其内容，可启动 Management Studio，打开“数据库”节点的 AdventureWorks2008 数据库子节点并且右击“视图”节点。选择“新建视图”命令，弹出如图 10-1 所示的对话框。

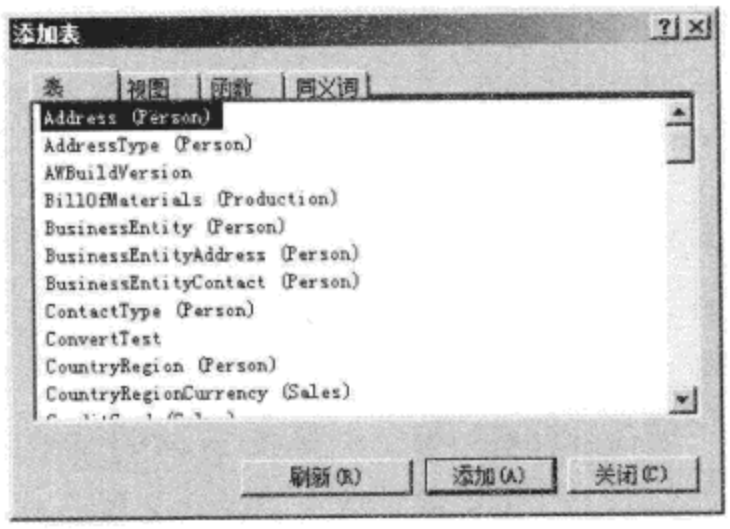


图 10-1

使用这个对话框可以很容易地选择那些希望获取从中数据的表。图 10-1 中选中了 Address 表，但是这里将使用的不仅仅是另外一个表——而是另外 4 个表。

这就出现了这样的问题：“如何选择多个表？”这很简单——只需按住 Ctrl 键来选择所有需要的表。现在，首先单击 Customers 表，然后按住 Ctrl 键，选择 SalesOrderHeader、SalesOrderDetail 和 Product 表。所有选中的表都是突出显示的。现在单击“添加”按钮，SQL Server 将添加一些表到视图上(实际上，在大多数系统上，应当能看见这些表被添加到后面将要看到的视图编辑器上)。

在关闭“添加表”对话框之前，花点时间注意一下该对话框顶部的“视图”、“函数”和“同义词”选项卡。因为视图可以直接引用这些对象，所以该对话框提供了一个直接添加这些对象的方法。

不过现在只需单击“添加”按钮，然后检查弹出的视图编辑器，如图 10-2 所示。

视图生成器(View Builder)有 4 个窗格——每个窗格可以独立打开或者关闭：

- “显示关系图”窗格
- “显示条件”窗格
- “显示 SQL”窗格
- “显示结果”窗格

对于使用过 Access 的人来说，可以看出“显示关系图”窗格的工作方式和它在 Access 查询中很相似。可以添加或者删除表，甚至定义关系。每个添加的表、选择的列以及定义的关系将自动地在“显示 SQL”窗格中以与数据库关系图匹配的 SQL 形式出现。为了识别工具栏上的每个图标，只需将鼠标指针在图标上悬停一会儿，就将得到说明每个按钮的作用的工具提示。

www.TopSage.com

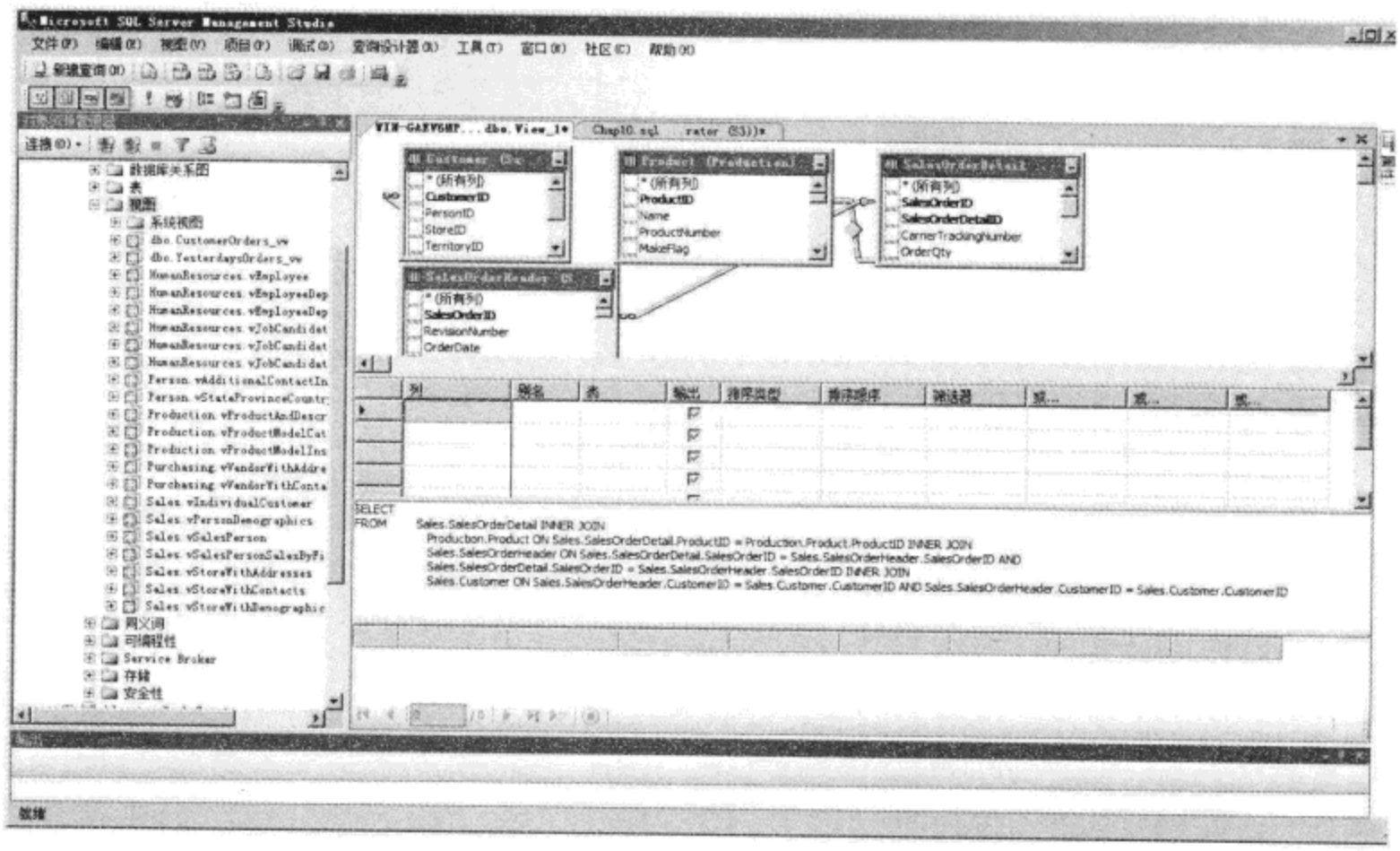


图　10-2

注意：

可以通过右击“显示关系图”窗格(在图 10-2 中的顶部)然后选择“添加表”命令或者通过单击“添加表”工具栏按钮(在图标左上角带有一个指向右边的箭头)来添加表。

现在选择几列，如图 10-3 所示。

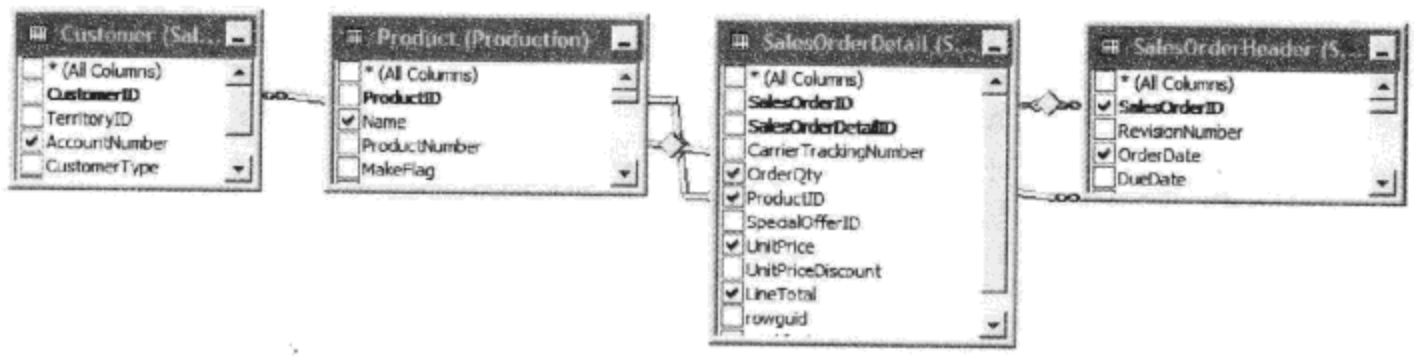


图　10-3

注意这里只显示了“显示关系图”窗格以节省空间。如果“显示条件”窗格是打开的，那么在选择每列时，可以看见这些列出现在“显示条件”窗格内。而如果 SQL 窗格是打开的，那么也可看见这些列出现在 SQL 代码中。

可能您还没有意识到，这里所创建的视图和之前第一个复杂视图(CustomerOrders_vw)是同样的。唯一棘手的事情是计算列(TotalDiscount)。为了处理计算列，必须手动把这个等式输入“显示 SQL”窗格里，或者可以把这个等式连同别名一起输入到“显示条件”窗格的“列”列里(如图 10-4 所示)。

www.TopSage.com

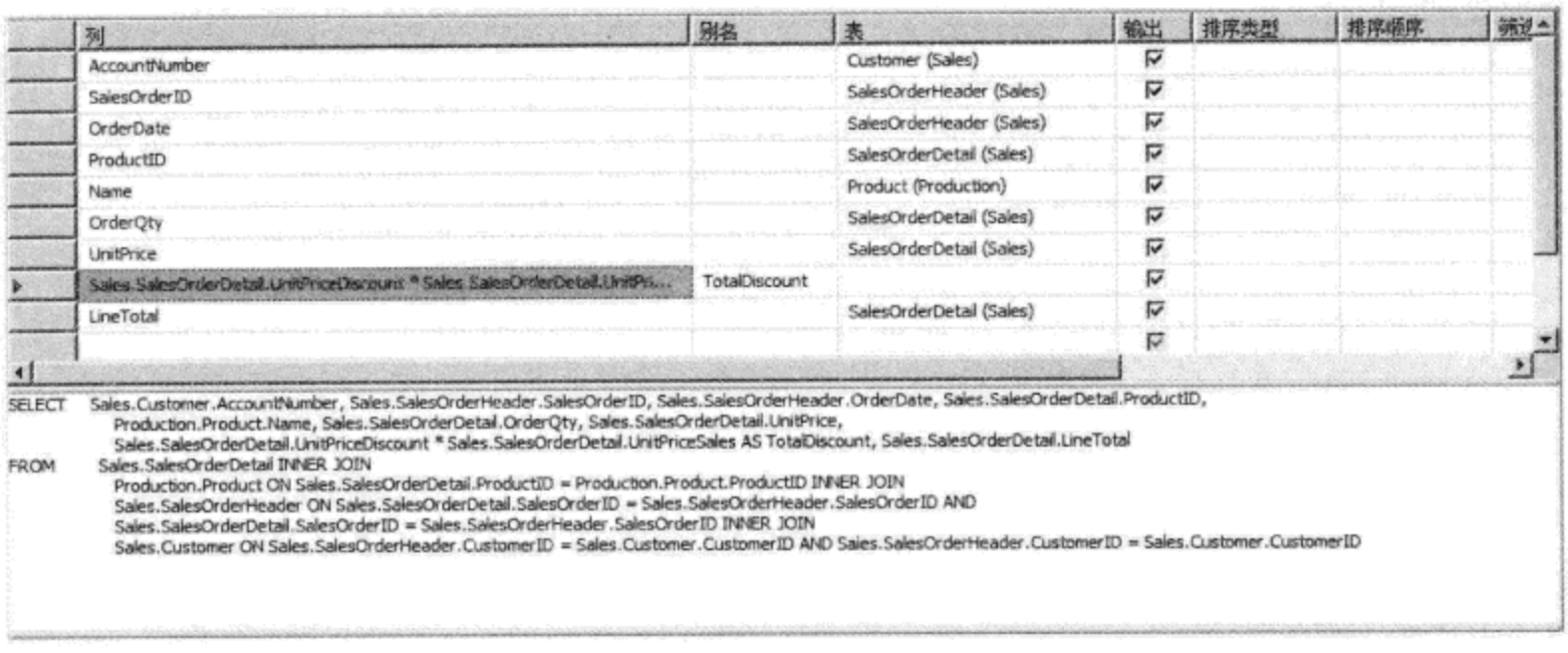


图 10-4

当所有这些完成时，视图生成器提供了下列 SQL 代码：

```
SELECT Sales.Customer.AccountNumber,
       Sales.SalesOrderHeader.SalesOrderID,
       Sales.SalesOrderHeader.OrderDate,
       Sales.SalesOrderDetail.ProductID,
       Production.Product.Name,
       Sales.SalesOrderDetail.OrderQty,
       Sales.SalesOrderDetail.UnitPrice,
       Sales.SalesOrderDetail.UnitPriceDiscount *
        Sales.SalesOrderDetail.UnitPrice *
        Sales.SalesOrderDetail.OrderQty AS TotalDiscount,
      Sales.SalesOrderDetail.LineTotal
FROM   Sales.SalesOrderDetail
INNER JOIN Production.Product
 ON Sales.SalesOrderDetail.ProductID = Production.Product.ProductID
INNER JOIN Sales.SalesOrderHeader
 ON Sales.SalesOrderDetail.SalesOrderID=Sales.SalesOrderHeader.SalesOrderID
 AND Sales.SalesOrderDetail.SalesOrderID=Sales.SalesOrderHeader.SalesOrderID
INNER JOIN Sales.Customer
 ON Sales.SalesOrderHeader.CustomerID = Sales.Customer.CustomerID
 AND Sales.SalesOrderHeader.CustomerID = Sales.Customer.CustomerID
```

虽然格式上并不完全相同，但是如果仔细查看，可以发现这和手写的代码基本上是一样的。

注意：

如果您正在努力学习 T-SQL 查询语法，那么可以使用这个工具来学习查询语法。只需将一些表拖进或者拖出“显示关系图”窗格，从每个表中选择想要的列，在大部分情况下，SQL Server 都会建立一个查询——然后您可以学习视图生成器中使用的语法，下一次就可以自己手动建立查询了。

接下来，将它保存(这里是通过工具栏里的磁盘图标来保存的)为 CustomerOrders2_vw 并且关闭视图生成器。

www.TopSage.com

在 Management Studio 中编辑视图

在 Management Studio 中修改视图和创建视图一样简单。唯一的区别是需要导航到特定的视图并且右击它——然后选择“设计”命令，会弹出和过去创建查询时使用的同样友好的查询设计器。

10.6 审查：显示现有的代码

如果您有一个视图，但是不能确定该视图是做什么的，那该怎么办？对此，第一选择是很简单的——就像将要编辑视图那样进入 Management Studio。找到“视图”子节点，选择想要编辑的视图，右击并选择“设计”命令。将看见视图后面具有彩色编码的全部代码。

遗憾的是，并不总是能通过 Management Studio 提供所需要的信息(可能正在使用某种轻量级工具)。好的方面是有两种得到实际视图定义的方法。

- sp_helptext
- sys.modules 元数据函数

一般优先使用 sp_helptext，因为当新版本出现时，它将根据系统表的变化自动地更新。

提示：

还有第 3 种选择，即直接访问 syscomments 系统表。Microsoft 在一些版本中警告说不要直接使用系统表。在编写本书时，syscomments 系统表仍然存在，但在运行查询时，结果中可能会有一些不真实的信息。它们可能较好地工作，但 Microsoft 一直建议目前暂时不要使用 syscomments，因此采用“被批准”的方法可能是更好的选择。

例如，对 10.2 节中在 AdventureWorks2008 数据库中创建的一个视图(YesterdaysOrders_vw)运行 sp_helptext：

```
EXEC sp_helptext YesterdaysOrders_vw;
```

SQL Server 提供了下列视图代码：

```
Text
----------------------------------------------------------------------
CREATE VIEW YesterdaysOrders_vw
AS
SELECT sc.AccountNumber,
       soh.SalesOrderID,
       soh.OrderDate,
       sod.ProductID,
       pp.Name,
       sod.OrderQty,
       sod.UnitPrice,
       sod.UnitPriceDiscount * sod.UnitPrice * sod.OrderQty AS
       TotalDiscount,sod.LineTotal
FROM Sales.Customer AS sc
INNER JOIN Sales.SalesOrderHeader AS soh
       ON sc.CustomerID = soh.CustomerID
```


www.TopSage.com

```
INNER JOIN Sales.SalesOrderDetail AS sod
      ON soh.SalesOrderID = sod.SalesOrderID
INNER JOIN Production.Product AS pp
      ON sod.ProductID = pp.ProductID
WHERE CAST(soh.OrderDate AS Date) =
      CAST(DATEADD(day,-1,GETDATE()) AS Date)
```

现在尝试另外一种方法——使用 sys.sql_modules 元数据函数。使用这一函数的主要问题在于所有对象都用对象 ID 编码。

注意：

对象 ID 是 SQL Server 跟踪事物的内部方法。它们是整型值而不是用于对象的名称。通常而言，这超出了本书的范围，但是还是要知道它们的存在，因为您可能会发现在从其他人那里复制过来的脚本或者自己编写的 SQL 脚本中使用了对象 ID。

幸运的是，可以通过使用 OBJECT_ID()函数来避开这个问题：

```
SELECT *
FROM sys.sql_modules
WHERE object_id = OBJECT_ID('dbo.YesterdaysOrders_vw');
```

再次得到同样的代码块(实际上，sp_helptext 所做的与这个查询所做的工作一样)。

注意：

默认情况下，“查询”窗口中的“以文本格式显示结果”选项限制了任何单独列的结果为 256 个字字符，因此运行之前的 sys.sql_modules 查询将获取视图文本的截断版本。如果在网格模式下查看它，则限制为 64KB，因此会全部显示。可以通过选择“工具”|“选项”|“查询结果”|“SQL Server”|“将结果保存到文件”命令，并改变适当的设置，可以更改每列的最大字符数。

10.7 保护代码：加密视图

如果您正在构建某种商业软件产品，那么可能会对保护源码有兴趣。首先可对视图进行保护。

加密视图所有必须做的工作是使用 WITH ENCRYPTION 选项。如果您熟悉 WITH CHECK OPTION 子句，那要注意这里的几个技巧。

- WITH ENCRYPTION 跟在视图的名称之后，但在 AS 关键字之前
- WITH ENCRYPTION 不使用 OPTION 关键字

除此之外，记住如果使用 ALTER VIEW 语句，那么意味着除了访问权限外，完全替换了现有的视图。这意味着加密方法也被替换了。如果想对被改变的视图加密，那么必须在 ALTER VIEW 语句中使用 WITH ENCRYPTION 子句。

对在 AdventureWorks2008 中创建的 CustomerOrders_vw 视图执行 ALTER VIEW。如果还没有创建 CustomerOrders_vw 视图，那么只需将 ALTER 改为 CREATE：

www.TopSage.com

```
ALTER VIEW CustomerOrders_vw
WITH ENCRYPTION
AS
SELECT sc.AccountNumber,
       soh.SalesOrderID,
       soh.OrderDate,
       sod.ProductID,
       pp.Name,
       sod.OrderQty,
       sod.UnitPrice,
       sod.UnitPriceDiscount * sod.UnitPrice * sod.OrderQty AS
       TotalDiscount, sod.LineTotal
FROM Sales.Customer AS sc
INNER JOIN Sales.SalesOrderHeader AS soh
    ON sc.CustomerID = soh.CustomerID
INNER JOIN Sales.SalesOrderDetail AS sod
    ON soh.SalesOrderID = sod.SalesOrderID
INNER JOIN Production.Product AS pp
    ON sod.ProductID = pp.ProductID;
```

现在对 CustomerOrders_vw 执行 sp_helptext：

```
EXEC sp_helptext CustomerOrders_vw
```

SQL Server 很快地说明它不能完成所要求的工作：

```
对象 'CustomerOrders_vw' 的文本已加密。
```

您想知道原因，于是又迅速采用 sys.sql_modules 元数据函数。

```
SELECT *
FROM sys.sql_modules
WHERE object_id = OBJECT_ID('dbo.CustomerOrders_vw');
```

但是那也没有提供太多信息——SQL Server 认识到表是加密的，然后返回一个 NULL 结果。

简言之——您的代码是安全的和可靠的。即使采用其他视图查看器(例如 Management Studio，它甚至都不会对加密的表提供“修改”选项)，结果也是类似的。

注意：

确保在使用 WITH ENCRYPTION 选项之前，已在某处存储了源代码。一旦源代码被加密了，就没有办法恢复。如果没有在其他地方存储代码，又需要改变代码，那么就要重新编写。

10.8 关于模式绑定

模式绑定实际上就是将视图所依赖的事物(表或者其他的视图)“绑定”到视图。其重要意义在于除非首先删除模式绑定的视图，否则没有人可以修改那些对象(CREATE、ALTER)。

为什么要这么做呢？有下列一些理由。

www.TopSage.com

- 可以防止修改底层对象时使视图“孤立”。想象一下，可能某个人执行 DROP 或者做了一些其他的改变(甚至删除一列也会给视图带来麻烦)，但是没有考虑视图。如果视图是模式绑定的，那么可以防止这种情况发生。
- 为了允许创建索引视图：如果想要在视图上创建索引，那么必须使用 SCHEMABINDING 选项来创建视图(稍后就会讲述索引视图)。
- 如果要创建一个模式绑定的用户自定义函数(在一些实例中，用户自定义函数必须是模式绑定的)来引用视图，那么视图也必须是模式绑定的。

在创建视图时，要牢记这些要点。

10.9 使用 VIEW_METADATA 使视图看起来像表一样

对于 DB-LIB、ODBC 以及 OLE-DB 客户端来说，这个选项可以使得视图看起来就像像实际的表一样。如果没有这个选项，那么传回到客户端 API 的元数据是视图所依赖的基表的数据。

提供这些元数据信息使得任何客户端游标(客户应用程序管理的游标)可被更新。要注意，如果想要支持这样的游标的话，还需要使用 INSTEAD OF 触发器。

10.10 索引(物化)视图

当一个视图被引用时，组成该视图的查询中的逻辑实际上被合并到调用查询中去。遗憾的是，这意味着调用查询会变得更加复杂。动态计算视图(以及它表示的数据)影响的额外系统开销实际上可能变得非常大。更甚者，需要经常以在视图中连接表的形式添加额外的连接到视图中。而索引视图提供了一种在运行查询前处理这个影响的方法。

索引视图实际上是一种将一组唯一值“物化”为群集索引形式的视图。其优点是它在提取视图背后的信息方面提供了一个非常快的查找方法。在第一个索引(必须是针对一组唯一值的群集索引)之后，通过使用来自第一个索引的群集键作为参考点，SQL Server 还能在视图上建立额外的索引。不过，天下没有免费的午餐——对于何时可以(或不可以)在视图上创建索引有一些限制。

- 视图必须使用 SCHEMABINDING 选项。
- 如果视图引用了任何用户自定义函数(后面将更详细讲述)，那么这些函数也必须是模式绑定的。
- 视图不可以引用任何其他的视图——只能引用表和 UDF。
- 在视图中引用的所有表和 UDF 必须采用两部分的(不允许三部分和四部分的名称)命名约定(例如 dbo.Customers、BillyBob.SomeUDF)，并且也必须具有和视图相同的所有者。
- 视图和视图引用的所有对象必须在相同的数据库里。
- 在创建视图和所有底层表时，必须打开 ANSI_NULLS 以及 QUOTED_IDENTIFIER 选项(使用 SET 命令)。
- 视图引用的任何函数必须是确定的。

为了创建一个索引视图示例，首先对本章之前创建的 CustomerOrders_vw 对象作一些修改：

www.TopSage.com

```
ALTER VIEW CustomerOrders_vw
WITH SCHEMABINDING
AS
SELECT sc.AccountNumber,
       soh.SalesOrderID,
       soh.OrderDate,
       sod.ProductID,
       pp.Name,
       sod.OrderQty,
       sod.UnitPrice,
       sod.UnitPriceDiscount*sod.UnitPrice*sod.OrderQty AS TotalDiscount,
       sod.LineTotal
FROM   Sales.Customer AS sc
INNER JOIN Sales.SalesOrderHeader AS soh
     ON sc.CustomerID = soh.CustomerID
INNER JOIN Sales.SalesOrderDetail AS sod
     ON soh.SalesOrderID = sod.SalesOrderID
INNER JOIN Production.Product AS pp
     ON sod.ProductID = pp.ProductID;
```

这里要注意的重要事情是必须使视图使用 SCHEMABINDING 选项。这实际上只是开始——还没有创建好索引视图。事实上，目前所具备的只是一个可以被索引的视图。在构建索引之前，需要设定一个基准，即当前如何对视图执行查询。对视图执行一个简单的 SELECT 操作：

```
SELECT * FROM CustomerOrders_vw;
```

尽管很简单，但要记住这个视图底层有 4 个表。如果选择显示“估计执行计划”工具按钮(工具栏上的一个图标)，如图 10-5 所示，那么将看到一些执行查询的复杂步骤。

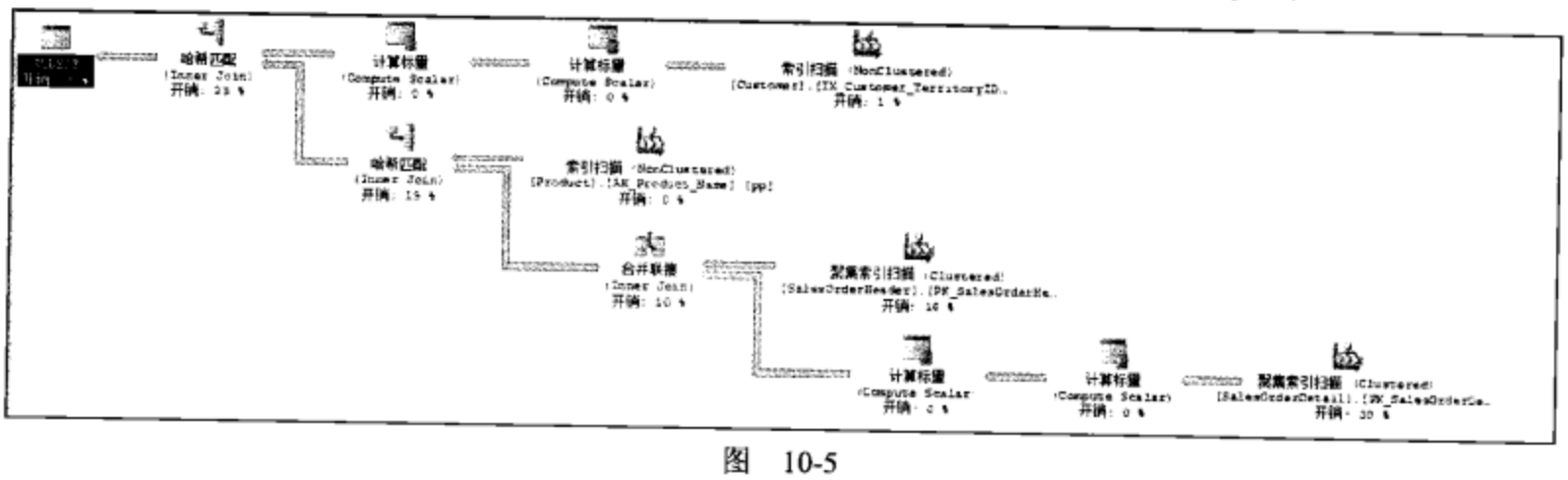


图　10-5

当将鼠标指针悬停在执行计划的左上节点上时，会显示黄色的弹出菜单。这是执行计划的顶级节点，查看该节点上的“估计子树大小”命令，可看到显示了整个查询的估计成本(对本书而言，是 3.49114)。在了解了查询的复杂性和估计的成本后，准备创建索引。

当创建索引时，在视图上创建的第一个索引必须是群集的和唯一的。

```
CREATE UNIQUE CLUSTERED INDEX ivCustomerOrders
ON CustomerOrders_vw(AccountNumber, SalesOrderID, ProductID);
```

一旦执行该命令，就有了视图的群集索引(图 10-6 所示)，再次检查基本的 SELECT 查询的“估计执行计划”命令。

www.TopSage.com

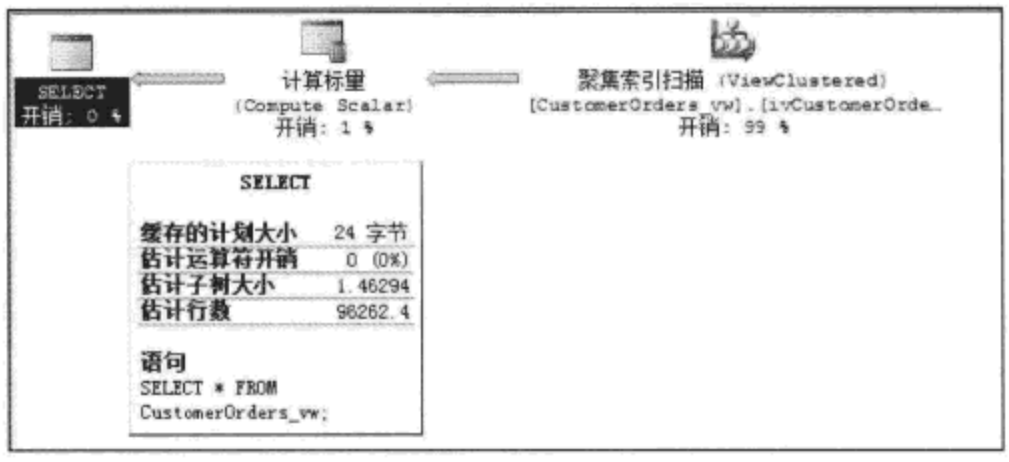


图 10-6

注意，这是个非常简单的查询。执行的估计成本的下降幅度超出了 50%。这意味着索引是好的选择吗？不是的。这只意味着它可能是。和任何索引一样，需要记住索引的维护成本。维护该索引将会使对底层表的 INSERT、UPDATE 和 DELETE 语句的执行速度减慢多少？这是个平衡问题，要视每个表和每个索引而定。尽管如此，索引视图还是一种较强大的工具，因此作仔细地权衡。

10.11 本章小结

在大多数数据库中，视图这一工具的使用往往不是过多就是过少。一些人喜欢使用视图抽象化几乎所有一切内容(当他们这样做时，通常忽略了这意味着增加了一层处理)。另外一些人则好像忘记视图也是一种选择。本书认为，应该像大多数事物那样，在适当的时候使用视图，而不要走极端。有关视图要记住的事项包括以下几个方面。

- 避免基于视图构建视图——而应该将来自第一个视图的合适的查询信息应用到新的视图中。
- 记住使用 WITH CHECK OPTION 的视图提供了一些普通 CHECK 约束所不具备的灵活性。
- 如果不希望其他人能看见您的源代码(出于商业产品或一般的安全考虑)，就加密视图，但要记得备份未加密的代码；因为在加密后就不能再恢复未加密的代码了。
- 除了权限之外，使用 ALTER VIEW 意味着完全替换了现有的视图。这意味着如果要使被修改的视图里的加密和限制仍然是有效的，那么必须在 ALTER 语句里包含 WITH ENCRYPTION 和 WITH CHECK OPTION 子句。
- 使用 sp_helptext 显示视图的支持代码——避免使用系统表。
- 最小化用于生产查询的视图的用户——因为他们增加了额外的系统开销并且危害性能。

视图一般用于以下情况。

- 过滤行
- 保护敏感数据
- 降低数据库复杂性
- 将多个物理数据库抽象为一个逻辑数据库

www.TopSage.com

第 11 章将讲述批处理和脚本。批处理和脚本将引导读者学习存储过程——这是 SQL Server 中最接近于 SQL Server 自身程序的部分。

10.12 练习

1. 在 AdventureWorks2008 数据库中添加一个名为 HumanResources.Managers 的视图，只显示管理其他雇员的雇员。
2. 将刚创建的视图进行加密。
3. 根据 ManagerID 和 EmployeeID 列，对新视图创建索引。

www.TopSage.com

第11章

编写脚本和批处理

不管您是否已经意识到这一点，现在可以编写 SQL 脚本了。您所编写的每一条 CREATE 语句、每一条 ALTER 语句以及每一条 SELECT 语句可能是一个脚本的全部(如果运行单条语句)或者部分(多条语句)。然而，只有一行的脚本意义并不大，脚本必须在上下文中才有意义。

SQL 脚本在很大程度上也是如此。当将一些命令串成一个更长的脚本时，事情会变得更有意思。现在可以想象一下将添加更多的.NET 语言元素。

脚本通常都有一个统一的目标。也就是说，脚本中所有的命令通常是为了达到一个总体的目的。本章的示例包括了构建数据库的脚本(这些脚本可能用于系统安装)，以及用于系统维护的脚本(备份、数据库一致性检查器(DBCC)实用程序、索引碎片整理等)——几个命令通常一起运行且用于任何情况的脚本。

本章除了介绍脚本外，还会介绍一下批处理的概念——批处理控制 SQL Server 组合命令的方式。此外，还将简介一下 sqlcmd——命令行实用程序，以及它与脚本的关系。

提示：

sqlcmd 是 SQL Server 2005 中首次引入的。为了向后兼容，SQL Server 继续支持 osql.exe(以前执行命令行工作的工具)。可能还会看到 isql.exe(不要将这个和 isqlw.exe 混淆)，其在早期版本中提供了相同的功能。不过，SQL Server 2005 不再支持 isql.exe。

11.1 脚本的基础知识

从技术上讲，只有将脚本存储到一个文件中并且可提取和重用时，脚本才成为一个真正的脚本。SQL 脚本以文本文件的形式存储。SQL Server Management Studio 提供了许多辅助编写脚本的工具。基本查询窗口是彩色编码的，这样不仅有助于识别关键字，而且有助于理解它们的性质。除此之外，还有 Intellisense、单步调试器、代码模板、对象浏览器等。

脚本通常被看作一个单元。也就是说，正常情况下，或者执行整个脚本或者什么也不执行。脚本可以使用系统函数和局部变量。例如，看一下在第 5 章创建的 Accounting 数据库中插入订单记录的脚本：

```
USE Accounting;
DECLARE @Ident int;
```

www.TopSage.com

```
INSERT INTO Orders
(CustomerNo,OrderDate, EmployeeID)
VALUES
(1, GETDATE(), 1);
SELECT @Ident = @@IDENTITY;
INSERT INTO OrderDetails
(OrderID, PartNo, Description, UnitPrice, Qty)
VALUES
(@Ident, '2R2416', 'Cylinder Head', 1300, 2);
SELECT 'The OrderID of the INSERTed row is ' + CONVERT(varchar(8),@Ident);
```

注意：

如果没有填充版本的 Accounting 数据库，可从 Wrox 支持网站(wrox.com)或作者的个人网站(www.professionalsql.com)下载一个重新创建 Accounting 数据库的脚本。

这里用到了 6 个截然不同的命令，涵盖了在脚本中可能做的许多不同的事情。使用了系统函数、局部变量、USE 语句、INSERT 语句，以及 SELECT 语句的赋值版本和常规版本。所有这些命令协调一致地工作以完成一个任务——将全部的订单插入到数据库中。

11.1.1　USE 语句

USE 语句用于设置当前数据库。这会影响到对完全限定对象名的数据库部分使用默认值的任何地方。在这个特定示例中，还没有指出 INSERT 或 SELECT 语句中的表来自什么样的数据库，但是，因为在使用 INSERT 和 SELECT 语句之前已经包含了 USE 语句，所以它们将使用指定的数据库(在这里使用 Accounting 数据库)。如果没有 USE 语句，那么就由执行脚本的任何用户来确定执行脚本时的当前数据库是正确的。

注意：

不要认为这里的意思是指应当总是在脚本中包含 USE 语句——这取决于脚本的目的。如果只是一个通用脚本，那么省去 USE 语句实际上可能更有益。

通常，如果在脚本中命名特定于数据库的表(也就是说，非系统表)，那么需要使用 USE 命令。如果脚本用于修改一个特定的数据库，那么可以发现这是非常有帮助的——正如本书之前几章中已经讲过的，本书多次无意间地在 master 数据库中创建了大量的表，其实这些表本来应该用于用户数据库的。

接下来使用 DECLARE 语句声明变量。本书前面的一些脚本中已使用了 DECLARE 语句，现在将作更详细的介绍。

11.1.2　声明变量

DECLARE 语句的语法相当简单：

```
DECLARE @<variable name> <variable type>[= <value>][,
  @<variable name> <variable type>[= <value>][,
  @<variable name> <variable type>[= <value>]]]
```

可以一次仅仅声明一个变量，也可以一次声明几个变量。人们常常以一次声明一个变量的

www.TopSage.com

方式重用 DECLARE 语句，而不是使用逗号分隔的方法一次声明多个变量。选择哪种方法取决于您自己，但是不论选择何种方法，都必须初始化变量(使用“=”语法)，否则变量值为 NULL，直到显式地将其设置为一些其他值。

在这个示例中，将一个名为@ident 的局部变量声明为整型。这里没有选择初始化该局部变量，因为这个变量用于接受另一个数据源的值。从技术上讲，可以不声明这个变量——而是可以选择直接使用@@IDENTITY。@@IDENTITY 是一个系统函数。它总是可用的，并提供在当前连接中分配的最近标识值。和大多数系统函数一样，应当形成一个习惯，即显式地将@@IDENTITY 中的值移动到一个局部变量中。这样，可以确保该值不会被无意间修改。在这里没有那种危险，但是，为了保持一致性，还是要这么做。

注意：

本书喜欢将从系统函数中取出的值移动到用户自己的变量中。那样，就能安全地使用这个值并且知道只有当用户修改这个值时，它才会改变。而对于系统函数本身，因为大多数系统函数不是由您设置的而是由系统设置的，所以有时不能确定函数值何时变化。这会造成一种情况，即不期望系统值改变时，偏偏系统改变了某个数值，这样就会产生不可预知的可怕结果。

1. 设置变量中的值

知道了如何声明变量后，接下来的问题是，“如何修改它们的值？”有两种设置变量值的方法。可以使用 SELECT 语句或者 SET 语句。从功能上看，它们的作用几乎是相同的，不同的是 SELECT 语句允许源数据值来自 SELECT 语句中的某一列。

那么为什么有两种方法来做这件事情呢？我并不知道原因。在我出版的 4 本书中都提到了这个问题，我指望能有人给我发邮件并给出一个好的答案——但是没有。有充分的理由认为，SET 现在是 ANSI/ISO 标准的一部分，这是保留 SET 的理由。然而，我找不到 SELECT 中的相同功能方面有任何错误，甚至 ANSI/ISO 似乎也认为它是正确的。我确信这种冗余有其意义，但是我说不清具体原因是什么。虽说如此，但两种方法在实际使用中仍有一些区别。

1) 使用 SET 设置变量

SET 通常以在更加过程化的语言中常见的方式来设置变量。典型的使用示例是：

```
SET @TotalCost = 10
SET @TotalCost = @UnitCost * 1.1
```

可注意到，这些都是用显式的值或者其他变量来直接赋值的。使用 SET，不能将查询得到的值赋给变量——必须将查询和 SET 分开。例如下列语句：

```
USE AdventureWorks2008;

DECLARE @Test money;
SET @Test = MAX(UnitPrice) FROM [Order Details];
SELECT @Test;
```

就会产生错误，但是下列语句：

```
USE AdventureWorks2008;
```


www.TopSage.com

```
DECLARE @Test money;
SET @Test = (SELECT MAX(UnitPrice) FROM Sales.SalesOrderDetail);
SELECT @Test;
```

就没有问题。

提示：

尽管后一种语法可以起作用，但是习惯上，从不采用这种方法实现代码。我也不能确定“不采用这种方法实现”的原因，但是我猜想和可读性有关——您可能希望 SELECT 语句和检索表的数据相关，而 SET 用于简单的变量赋值。

2) 使用 SELECT 设置变量

当变量中存储的信息来源于查询时，经常用 SELECT 给变量赋值。例如，对于上一个示例，通常使用 SELECT 完成：

```
USE AdventureWorks2008;

DECLARE @Test money;
SELECT @Test = MAX(UnitPrice) FROM Sales.SalesOrderDetail;
SELECT @Test;
```

可注意到，这样会更清楚一点(花费了更少的步骤来做同样的事情)。

再次强调，关于何时使用哪种方法的约定如下。

- 当执行简单的变量赋值时，使用 SET——这时已知值是一个显式值或者其他变量。
- 当基于查询进行变量赋值时，使用 SELECT。

提示：

这里可能会有人表示怀疑，因为事实上可以看见在本书中的许多地方都违反了最后一个约定。第一次将 SET 用于变量赋值是在 7.0 的版本中，我必须承认，甚至在其发行后的 10 年多时间里，我仍然没有完全适应。虽然如此，这看起来实际上是 Microsoft 和 SQL Server 社团所推出的东西，所以我强烈建议作出正确的选择，遵守这个约定。

2. 系统函数回顾

可用的不带参数的系统函数有 30 多个。较老的系统函数都以@@符号打头——以前称为“全局变量”。不过所幸的是现在它们都更准确地改称为系统函数，并且大多数系统函数都没有@@前缀。其中最值得关心的是在表 11-1 列出的一些系统函数。

表　11-1

变　量	用　途	注　释
@@DATEFIRST	返回当前设置的一周的第一天(比如，星期日或星期一)	这是一个系统范围的设置——如果有人改变了这个设置，就得不到所期望的结果
@@ERROR	返回当前连接中最后一条执行的 T-SQL 语句的错误号。如果没有错误，返回 0	在每个新的语句中重新设置。如果需要保存这个值，可以在执行完需要保存其错误代码的语句之后，立刻把这个值移到一个局部变量中

www.TopSage.com

(续表)

变　量	用　途	注　释
@@IDENTITY	返回作为最后一条 INSERT 或者 SELECT INTO 语句的结果插入的标识值	如果没有标识值产生，那么设置其为 NULL。即使缺少标识值是由于语句未成功运行，也是如此。如果是通过一条语句执行多个插入，那么只返回最后的标识值
IDENT_CURRENT()	返回对特定表插入的最后标识值，而不管是哪个会话或作用域	如果插入多个表，它就不会被重写；但如果有其他连接插入到特定表，则它所提供的值就不是所预期的
@@OPTIONS	返回通过使用 SET 命令设置的选项的信息	尽管只得到一个值，但可以设置许多选项，SQL Server 使用二进制标记说明设置什么值。为了检验是否设置了自己感兴趣的选项，必须将这个选项值和一个位运算符一起使用
@@REMSERVER	仅在存储过程中使用。返回调用存储过程的服务器的值	如果希望存储过程根据调用它的远程服务器(通常指地理位置)的不同表现出不同的行为，这个选项是很方便的。在.NET 时代，我仍要提出这样一个问题：任何需要这个变量的操作是否可以通过使用.NET 中的其他更好的功能来编写
@@ROWCOUNT	一个常用的系统函数。返回最后一条语句所影响的行数	一般在非运行时错误检查时使用。例如，如果尝试通过使用一个 WHERE 子句删除一行，并且没有行被影响，那么这意味着一些不期望的事情发生了。那么可以手动引发错误
SCOPE_IDENTITY	类似于@@IDENTITY，但返回在当前会话和作用域中插入的最后标识	对于避免触发器或嵌套的存储过程执行重写预期的标识值的额外插入很有用
@@SERVERNAME	返回正在运行脚本的本地服务器的名称	可以通过 sp_addserver 然后重启 SQL Server 来改变，但是很少需要改变
@@TRANCOUNT	返回当前连接的活动事务数——实质是事务嵌套级别	除非使用了保存点，否则 ROLLBACK TRAN 语句会将@@TRANCOUNT 减少到 0。BEGIN TRAN 增量可使@@TRANCOUNT 递增 1， COMMIT TRAN 减量使@@TRANCOUNT 递减 1
@@VERSION	返回当前的 SQL Server 安装版本以及日期、处理器和 O/S 体系结构	遗憾的是，这不能以任何结构化的字段排列形式返回信息，所以如果需要对特定信息进行测试的话，必须对它进行分析。还要确保使用了 xp_msver 扩展存储过程

www.TopSage.com

如果不理解其中的一些术语也不要担心。随着不断的学习，会逐步了解它们的，并且在以后需要的时候可以回过头来查看这个表或附录 A。只是要记得可从这些地方找到关于系统和行为的当前状态的完整信息。

11.1.3　使用@@IDENTITY

@@IDENTITY 是所有系统函数中最重要的一个。标识列是这样的一种列，即不对其提供一个值，而是由 SQL Server 自动地插入一个已编号的值。

在示例中，在对 Orders 表执行一个插入操作之后，就得到了@@IDENTITY 的值。问题是没有提供这个表的键值——当执行插入操作时，它是自动创建键值的。现在想要将一个记录插入到 OrderDetails 表中，但是需要知道 Orders 表中关联记录的主键的值(记住，在引用 Orders 表的 OrderDetails 表上有一个外键约束)。因为 SQL Server 生成了该值，而不是由用户提供的，所以需要有一种方法检索到该值，以便后面在脚本上进行相关的插入。@@IDENTITY 给出了那个自动产生的值，因为那是最后运行的语句。

在这个示例中，也可以不将@@IDENTITY 移动到一个局部变量中——可以在下一个 INSERT 查询中显式地引用它。但是，本书总是习惯于将它移动到一个局部变量中，以避免在确实需要其副本时可能出现的错误。例如，假定有另一个 INSERT 操作依赖于前面 INSERT 到 Orders 表操作的标识值。如果没有将这个值移动到一个局部变量中，那么在执行这下一个 INSERT 操作时，这个值就会丢失，因为它被来自于 OrderDetails 表的值重写，由于 OrderDetails 表没有标识列，因此这意味着@@IDENTITY 将被设置为 NULL。将@@IDENTITY 的值移动到一个局部变量中也使得用户保存了这个值，可以保存在输出语句附近以备后用。

下面创建一些表进行试验：

```
CREATE TABLE TestIdent
(
 IDCol int IDENTITY
 PRIMARY KEY
);

CREATE TABLE TestChild1
(
 IDcol int
 PRIMARY KEY
 FOREIGN KEY
 REFERENCES TestIdent(IDCol)
);

CREATE TABLE TestChild2
(
 IDcol int
 PRIMARY KEY
 FOREIGN KEY
 REFERENCES TestIdent(IDCol)
);
```

这里得到一个父表——它有一个用作主键的标识列(这是该表具有的唯一一列)。还有两个子表。它们都是用于标识关系的对象——也就是说，两个子表都通过在另一个表(父表)上设置

www.TopSage.com

外键以获得主键的一部分或者全部(这里是主键的全部)。所以出现了这样一种情况，即两个子表需要从父表得到它们的键。因此，需要首先将一个记录插入到父表中，然后检索生成的标识值以在其他的表中使用。

试一试　使用@@IDENTITY

既然已经有一些可以操作的表，那么准备尝试一些测试脚本：

```
/*****************************************
** This script illustrates how the identity
** value gets lost as soon as another INSERT
** happens
****************************************** */

DECLARE @Ident int; -- This will be a holding variable
/* We'll use it to show how we can
** move values from system functions
** into a safe place.
*/
INSERT INTO TestIdent
 DEFAULT VALUES;
SET @Ident = @@IDENTITY;
PRINT 'The value we got originally from @@IDENTITY was ' +
CONVERT(varchar(2),@Ident);
PRINT 'The value currently in @@IDENTITY is ' + CONVERT(varchar(2),@@IDENTITY);

/* On this first INSERT using @@IDENTITY, we're going to get lucky.
** We'll get a proper value because there is nothing between our
** original INSERT and this one. You'll see that on the INSERT that
** will follow after this one, we won't be so lucky anymore. */
INSERT INTO TestChild1
VALUES
 (@@IDENTITY);

PRINT 'The value we got originally from @@IDENTITY was ' +
CONVERT(varchar(2),@Ident);
IF (SELECT @@IDENTITY) IS NULL
 PRINT 'The value currently in @@IDENTITY is NULL';
ELSE
 PRINT 'The value currently in @@IDENTITY is ' + CONVERT(varchar(2),@@IDENTITY);
-- The next line is just a spacer for our print out
PRINT '';
/* The next line is going to blow up because the one column in
** the table is the primary key, and primary keys can't be set
** to NULL. @@IDENTITY will be NULL because we just issued an
** INSERT statement a few lines ago, and the table we did the
** INSERT into doesn't have an identity field. Perhaps the biggest
** thing to note here is when @@IDENTITY changed - right after
** the next INSERT statement. */
INSERT INTO TestChild2
VALUES
 (@@IDENTITY);
```


www.TopSage.com

示例说明

在这个脚本中所做的是：如果直接依赖@@IDENTITY 而不是将该值移动到一个安全的地方，那么将会发生什么？当执行前面的脚本时，一切将工作得很好，只有最后的 INSERT 有问题。最后的语句是尝试直接使用@@IDENTITY，但是前面的 INSERT 语句已经修改了@@IDENTITY 中的值。因为该语句是针对没有标识列的表，所以@@IDENTITY 中的值被设置为 NULL。因为在主键中不能有 NULL 值，所以最后的 INSERT 失败了：

```
(1 行受影响)
The value we got originally from @@IDENTITY was 1
The value currently in @@IDENTITY is 1

(1 行受影响)
The value we got originally from @@IDENTITY was 1
The value currently in @@IDENTITY is NULL

消息 515，级别 16，状态 2，第 37 行
不能将值 NULL 插入列'IDcol'，表'Accounting.dbo.TestChild2'；列不允许有 NULL 值。
INSERT 失败。
语句已终止。
```

如果只是稍作改变(即保存原始的@@IDENTITY 值)：

```
/*****************************************
** This script illustrates how the identity
** value gets lost as soon as another INSERT
** happens
***************************************** */

DECLARE @Ident int; -- This will be a holding variable

/* We'll use it to show how we can
** move values from system functions
** into a safe place.
*/
INSERT INTO TestIdent
 DEFAULT VALUES;
SET @Ident = @@IDENTITY;
PRINT 'The value we got originally from @@IDENTITY was ' +
CONVERT(varchar(2),@Ident);
PRINT 'The value currently in @@IDENTITY is ' + CONVERT(varchar(2),@@IDENTITY);

/* On this first INSERT using @@IDENTITY, we're going to get lucky.
** We'll get a proper value because there is nothing between our
** original INSERT and this one. You'll see that on the INSERT that
** will follow after this one, we won't be so lucky anymore. */
INSERT INTO TestChild1
VALUES
 (@@IDENTITY);

PRINT 'The value we got originally from @@IDENTITY was ' +
CONVERT(varchar(2),@Ident);
IF (SELECT @@IDENTITY) IS NULL
```


www.TopSage.com

```
 PRINT 'The value currently in @@IDENTITY is NULL';
ELSE
 PRINT 'The value currently in @@IDENTITY is ' + CONVERT(varchar(2),@@IDENTITY);
-- The next line is just a spacer for our print out
PRINT '';
/* This time all will go fine because we are using the value that
** we have placed in safekeeping instead of @@IDENTITY directly.*/
INSERT INTO TestChild2
VALUES
 (@Ident);
```

这次一切都运行得非常好：

```
(1 行受影响)
The value we got originally from @@IDENTITY was 1
The value currently in @@IDENTITY is 1
(1 行受影响)
The value we got originally from @@IDENTITY was 1
The value currently in @@IDENTITY is NULL
(1 行受影响)
```

注意：

在这个示例中，很明显存在问题，因为尝试将一个 NULL 值插入到主键中。现在，假设一个较不好的情况——第二个表有一个标识列。您可能很容易就会将虚假的数据插入到表中，甚至都没有发觉——至少直到出现一个非常严重的数据完整性问题时才会发觉！

11.1.4 使用@@ROWCOUNT

在到目前为止所执行的许多查询中，可以很容易地知道一条语句影响了多少行——“查询”窗口会告诉我们。例如，如果运行：

```
USE AdventureWorks2008
SELECT * FROM Person.Person;
```

这样可以看见 Person 中的所有行，但是也可以看见查询所影响的行数(本例中是表中的所有行)：

```
(19972 行受影响)
```

但是如果需要通过编程来知道影响了多少行，那么应该怎么做呢？与@@IDENITY 类似，@@ROWCOUNT 这个工具知道在脚本运行时正在发生什么——不过，这次得到的值是影响的行数而不是标识值。

用一个示例进一步检验这点：

```
USE AdventureWorks2008;
GO

DECLARE @RowCount int; --Notice the single @ sign
SELECT * FROM Person.Person;
SELECT @RowCount = @@ROWCOUNT;
print@RowCount
```


www.TopSage.com

这一次同样显示了所有的行，但是注意得到的新结果：

```
The value of @@ROWCOUNT was 19972
```

在本书后面介绍存储过程时，将可以看到这在哪些情形下可能是有用的方法。目前，只要认识到@@ROWCOUNT 系统函数提供了一种了解语句所做事情的方法，并且这也并不局限于 SELECT 语句——UPDATE、INSERT 和 DELETE 也可设置这个值。

注意：

如果浏览这个示例，那么您可能注意到，和使用@@IDENTITY 时所做的一样，这里选择将这个值移动到一个存放的变量中。@@ROWCOUNT 将会被接下来的语句重置为一个新的值，所以，如果将使用@@ROWCOUNT 值执行多个任务，那么应当将它移动到一个安全的存放位置中。

11.2　批处理

批处理是作为一个逻辑单元的一组 T-SQL 语句。一个批处理中的所有语句被组合成一个执行计划，因此对所有语句一起进行语法分析，并且必须通过语法验证，否则将不执行任何一条语句。注意，尽管如此，这并不能防止运行时错误的发生。如果发生运行时错误，那么任何在发生运行时错误之前执行的语句将仍然是有效的。简言之，如果一条语句不能通过语法分析，那么不会运行任何语句。如果一条语句在运行时失败，那么在产生错误的语句之前的所有语句都已经运行了。

到目前为止已经运行的所有脚本都是由一个批处理组成的。甚至迄今为止在本章中已经分析过的脚本都只是一个批处理。为了将一个脚本分为多个批处理，可使用 GO 语句。

- GO 语句必须自成一行(并且只有注释可以在同一行上)；也有一个例外，稍后将讨论，但是这里先将 GO 语句看作需要自成一行。
- 它使得自脚本的开始部分或者最近一个 GO 语句(任何一个更接近的)以后的所有语句编译成一个执行计划并发送到服务器，与任何其他批处理无关。
- GO 语句不是 T-SQL 命令，而是由各种 SQL Server 命令实用程序(sqlcmd 和 Management Studio 中的“查询”窗口)识别的命令。

1. 自成一行

GO 命令应当自成一行。在技术上，可以在 GO 命令之后的同一行上开始一个新的批处理，但是这会严重影响可读性。T-SQL 语句不能放在 GO 语句之前，否则 GO 语句经常会被错误地理解，从而造成语法分析错误或产生一些其他不可预料的后果。例如，如果在 WHERE 子句之后使用一个 GO 语句：

```
SELECT * FROM Person.Person WHERE ContactID = 1 GO
```

分析器就不知如何处理：

```
消息 102，级别 15，状态 1，第 1 行
```


www.TopSage.com

```
'GO'附近有语法错误。
```

2. 每个批处理单独发送到服务器

因为每个批处理被独立地处理，所以一个批处理中的错误不会阻止另一个批处理运行。要说明这点，看一下下面的代码：

```
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相互依赖性，那么或者每个批处理都失败——或者，至少在错误发生之后的每个批处理都失败——但是它们之间没有依赖性。如果执行以上的脚本，那么结果如下所示：

```
Done with first Batch...
消息 137，级别 15，状态 2，第 2 行
必须声明标量变量 "@MyVarchar"。
Done with third batch
```

重申一下，每个批处理在运行时是完全自治的。尽管如此，要记住，可以构建下面这种意义上的相互依赖关系：即后一个批处理试图执行的工作依赖于前一个批处理已完成的工作——在 11.2.2 节讨论什么能否跨越批处理时，会介绍这方面的内容。

3. GO 不是 T-SQL 命令

一个常见的错误是认为 GO 是 T-SQL 命令。其实 GO 是一个只能被编辑工具(Management Studio、sqlcmd)识别的命令。如果您使用了第三方工具，那么它可能支持也可能不支持 GO 命令，但是大多数声称支持 SQL Server 的工具是支持 GO 命令的。

当编辑工具遇到 GO 语句时，会将 GO 语句看作一个终止批处理的标记，将其打包，并且作为一个独立的单元发送到服务器——不包括 GO。这是正确的，因为服务器本身根本不知道 GO 应该是什么意思。

如果在一个 pass-through 查询中用 ODBC、OLE DB、 ADO、 ADO.NET 、SqlNativeClient，

www.TopSage.com

或者任何其他访问方法，那么会得到来自服务器的一个错误消息。GO 只是一个指示器，指明什么时候结束当前的批处理，以及什么时候适合开始一个新的批处理。

11.2.1　批处理中的错误

批处理中的错误分为以下两类。

- 语法错误
- 运行时错误

如果查询分析器发现一个语法错误，那么批处理的处理过程会立即被取消。因为语法检查发生在批处理编译或者执行之前，所以在语法检查期间的一个失败意味着还没有批处理被执行——不管语法错误发生在批处理中的什么位置。

运行时错误的工作方式有很大不同。因为任何在遇到运行时错误之前执行的语句已经完成了，所以除非是未提交的事务的一部分，否则这些语句所做的任何事情将不受影响(第 14 章将介绍事务，但是这里相关的是事务表示一个全部完成的或者什么都没有发生的情况)。运行时错误之外所发生的事情取决于错误的性质。一般而言，运行时错误将终止从错误发生的地方到批处理末端的批处理的执行。一些运行时错误(例如违反参照完整性)只是阻止违反参照完整性的语句运行——仍然会执行批处理中的所有其他语句。后一种情况说明了为什么错误检查如此重要——本书第 12 章将详细介绍错误检查。

11.2.2　何时使用批处理

使用批处理有若干目的，但是所有的批处理具有一个共同点——当脚本中的一些事情必须发生在另外一件事情之前或者分开发生时，需要使用批处理。

1. 要求有自己的批处理的语句

有一些命令必须有它们自己的批处理。这些命令包括：

- CREATE DEFAULT
- CREATE PROCEDURE
- CREATE RULE
- CREATE TRIGGER
- CREATE VIEW

如果想在单个脚本中将这些语句中的任意一个和其他的语句进行组合，那么需要通过使用 GO 语句将它们分散到各自的批处理中。

注意：

注意，如果 DROP 一个对象，那么可能想要将 DROP 语句放在它自己的批处理中或者至少和其他 DROP 语句在一个批处理中。为什么呢？因为如果将在以后创建一个具有相同名称的对象，那么除非 DROP 已经发生了，否则 CREATE 不能通过批处理语法分析。这意味着需要在前面一个单独的批处理中运行 DROP，以使得当具有 CREATE 语句的批处理执行时 DROP 操作已经完成。

www.TopSage.com

2. 使用批处理建立优先权

当需要建立优先权时，就很可能用到批处理——也就是说，在下一个任务开始之前，需要全部完成上一个任务。在大多数时候，SQL Server 可以很好地处理这种情况——脚本中的第一条语句是首先执行的，并且脚本中的第二条语句可以依赖第二条语句运行时服务器所处的适当状态。尽管如此，SQL Server 有时还是不能解决这种问题。

下面先看一个创建一个带有表的数据库的示例：

```
USE master;

CREATE DATABASE Test;

CREATE TABLE TestTable
(
 col1 int,
 col2 int
);
```

执行上面的脚本，初看起来，每条语句都运行得很好：

```
命令已成功完成。
```

不过，事情并不是它们所看起来的那样——在 Test 数据库中检验 INFORMATION_SCHEMA。

```
USE Test;

SELECT TABLE_CATALOG
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_NAME = 'TestTable';
```

那么可以注意到缺少了一些东西：

```
TABLE_CATALOG
--------------------
AdventureWorks2008
(1 行受影响)
```

为什么表被创建在错误的数据库中？答案取决于当运行 CREATE TABLE 语句时，当前的数据库是什么。在这个例子中，它碰巧是 master 数据库，所以表就创建在该数据库中。

提示：

注意，因为在运行这个脚本时，当前数据库可能不是 master 数据库，所以可能得到一个不同的结果。不过，当前数据库几乎可能是任何数据库。这就是要注意使用 USE 语句的原因。

这看似是一个很容易修正的问题——只需使用 USE 语句，但是在测试新的理论之前，必须删除旧的(较旧的)数据库。

```
USE MASTER;
DROP DATABASE Test;
```


www.TopSage.com

然后可以运行新修改的脚本：

```
CREATE DATABASE Test;

USE Test;

CREATE TABLE TestTable
(
 col1 int,
 col2 int
);
```

遗憾的是，这样也有它自身的问题：

```
消息 911，级别 16，状态 1，第 3 行
数据库'Test' 不存在。 请确保正确地输入了该名称。
```

分析器尝试验证该代码并且发现其正在通过 USE 命令引用一个不存在的数据库。这时批处理语句不可或缺，因此在尝试使用新的数据库之前，需要先完成 CREATE DATABASE 语句：

```
CREATE DATABASE Test;
GO
USE Test;
CREATE TABLE TestTable
(
 col1 int,
 col2 int
);
```

现在事情变得好多了。结果看起来是相同的：

```
命令已成功完成
```

但是当运行 INFORMATION_SCHEMA 查询时，事情得到证实：

```
TABLE_CATALOG
--------------------
Test
(1 行受影响)
```

下面继续来看看另一个例子，在这个示例中更需要显式建立优先权。

当使用 ALTER TABLE 语句修改一个列的类型或者添加列时，只有执行修改任务的批处理完成时，才能利用这些更改。

如果添加一列到 Test 数据库里的 TestTable 表中，然后尝试在第一个批处理没有结束时引用添加的列：

```
USE Test;

ALTER TABLE TestTable
 ADD col3 int;
INSERT INTO TestTable
```


www.TopSage.com

```
 (col1, col2, col3)
VALUES
 (1,1,1);
```

得到了一个错误的消息——SQL Server 不能解析新的列名称，于是产生错误：

```
消息 207，级别 16，状态 1，第 6 行
列名'col3'无效。
```

不过只需在 ADD col3 int 之后添加一个简单的 GO 语句，一切就会正常运行：

```
(1 行受影响)
```

11.3 sqlcmd

sqlcmd 是一个允许通过 Windows 命令提示符来运行脚本的实用程序。这对于执行转换或者维护脚本是非常好的，也是一种获取文本文件的简易方式。

sqlcmd 代替了旧的 osql。Osql 仍包括在 SQL Server 中，不过只是为了向后兼容。更老的命令行实用程序——isql，现在已不再被支持。

通过命令行运行 sqlcmd 的语法包含大量不同的开关，语法如下所示：

```
sqlcmd
[ { { -U <login id> [ -P <password> ] } | -E } ]
[ -z <new password> ] [ -Z <new password> and exit]
[-S <server name> [ \<instance name> ] ] [ -H <workstation name> ] [ -d <db name> ]
[ -l <login time out> ] [ -t <query time out> ] [ -h <header spacing> ]
[ -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:<input codepage> [ <, o: <output codepage> ]
[ -k [ 1 | 2 ] ]
[ -y <display width> ] [-Y <display width> ]
[ -p [ 1 ] ] [ -R ] [ -b ] [ -v var="<value>" ] [ -A ] [ -X [ 1 ] ] [ -x ]
[ -? ]
]
```

对于这些标记，需要记住的最重要的事情是这些标记中大部分(奇怪的是，不是全部)是区分大小写的。例如，“-Q”和“-q”都将执行查询，但是当查询完成时，前者将退出 sqlcmd，而后者不会退出。

所以，下面尝试直接从命令行执行快速查询。要记住，这意味着在 Windows 命令提示符下运行(不使用 Management Console)：

```
SQLCMD -Usa -Pmypass -Q "SELECT * FROM
 AdventureWorks2008.Production.Location"
```

提示：

-P 是指示口令的标记。如果服务器配置成需要其他口令而不是空口令(应该这么设置！)，那

www.TopSage.com

么需要紧跟在-P 之后提供口令，中间没有空格。如果使用 Windows 身份验证而不是 SQL Sever 身份验证，那么在-U 和-P 参数的位置用-E 参数替换(删除这两个参数，而只替换成一个-E 参数)。

如果在命令提示符下运行该命令，那么应当得到这样的结果：

```
C:\>SQLCMD -E -Q "SELECT * FROM AdventureWorks2008.Production.Location"
LocationID Name                                          CostRate       Avail
ability ModifiedDate
---------- ---------------------------------------- ------------ -----

         1 Tool Crib 0.0000
    .00 1998-06-01 00:00:00.000
         2 Sheet Metal Racks 0.0000
    .00 1998-06-01 00:00:00.000
         3 Paint Shop 0.0000
    .00 1998-06-01 00:00:00.000
         4 Paint Storage 0.0000
    .00 1998-06-01 00:00:00.000
         5 Metal Storage 0.0000
    .00 1998-06-01 00:00:00.000
         6 Miscellaneous Storage 0.0000
    .00 1998-06-01 00:00:00.000
         7 Finished Goods Storage 0.0000
    .00 1998-06-01 00:00:00.000
        10 Frame Forming 22.5000
  96.00 1998-06-01 00:00:00.000
        20 Frame Welding 25.0000
 108.00 1998-06-01 00:00:00.000
        30 Debur and Polish 14.5000
 120.00 1998-06-01 00:00:00.000
        40 Paint 15.7500
 120.00 1998-06-01 00:00:00.000
        45 Specialized Paint 18.0000
  80.00 1998-06-01 00:00:00.000
        50 Subassembly 12.2500
 120.00 1998-06-01 00:00:00.000
        60 Final Assembly 12.2500
 120.00 1998-06-01 00:00:00.000

(14 行受影响)

C:\>
```

现在，快速创建一个文本文件以了解 sqlcmd 语法包括一个文件时的工作方式。在命令提示符下输入下列代码：

```
C:\>copy con testsql.sql
```

这样将转入一个空行(没有任何种类的提示符)，在该空行处里可以输入下列命令：

```
Select * from AdventureWorks 2008.Production.Location
```

然后按 F6 键和回车键(结束文本文件的创建)。这时应得到这样的一条消息：

www.TopSage.com

```
已复制 1 个文件。
```

这次使用一个脚本文件重新尝试一下先前的查询。提示符处的命令行仅仅有一个细微的变化：

```
C:\>sqlcmd -Usa -Pmypass -i testsql.sql
```

所得到的结果与使用-Q 运行查询时所得到的一样。当然，主要的区别在于这次是从一个文件获取命令。这个文件中可以有几百个(甚至几千个)不同的命令。

试一试　使用 sqlcmd 生成一个文本文件

作为说明sqlcmd的最后一个示例，下面将利用它生成一个文本文件，可能将这个文本文件导入到另外一个应用程序(例如Excel)中进行分析。

在第 10 章，创建了一个显示前一天订单的视图。因此，首先将视图的核心查询复制到一个文本文件中。

```
C:\copy con YesterdaysOrders.sql
```

这将再次转一个空行(没有任何种类的提示符)，在那里可以输入以下代码：

```
USE AdventureWorks2008

SELECT sc.AccountNumber,
       soh.SalesOrderID,
       soh.OrderDate,
       sod.ProductID,
       pp.Name,
       sod.OrderQty,
       sod.UnitPrice,
       sod.UnitPriceDiscount * sod.UnitPrice * sod.OrderQty AS
       TotalDiscount,sod.LineTotal
FROM Sales.Customer AS sc
INNER JOIN Sales.SalesOrderHeader AS soh
     ON sc.CustomerID = soh.CustomerID
INNER JOIN Sales.SalesOrderDetail AS sod
     ON soh.SalesOrderID = sod.SalesOrderID
INNER JOIN Production.Product AS pp
     ON sod.ProductID = pp.ProductID
WHERE CAST(soh.OrderDate AS Date) =
     CAST(DATEADD(day,-1,GETDATE()) AS Date)
```

再次按下 F6 和 Enter 键，让 Windows 保存文件。

现在有了用于查询的文本文件源码，并且基本上准备好了让sqlcmd帮助产生输出。尽管如此，首先是需要有一些前一天的数据(除非运行了第 10 章使用的UPDATE语句脚本改变了前一天订单的日期，否则没有样本数据会有前一天的数据)。这里所指的是更新语句，是在“视图”一章中出现的那个。所以，明白了这点，再次运行该语句(如果喜欢的话，可以通过“查询”窗口来运行)：

www.TopSage.com

```
USE AdventureWorks2008

UPDATE Sales.SalesOrderHeader
SET OrderDate = CAST(DATEADD(day,-1,GETDATE()) AS Date),
   DueDate = CAST(DATEADD(day,11,GETDATE()) AS Date),
   ShipDate = CAST(DATEADD(day,6,GETDATE()) AS Date)
WHERE SalesOrderID < 43663;
```

所以现在至少有了一行数据，那是前一天的订单。这样就基本准备好了；然而，因为前面已经说过想要将结果放到一个文本文件中，所以这次将需要添加一些额外的参数到 sqlcmd 命令行，告诉 SQL Server 输出到何处：

```
C:\>sqlcmd -UMyLogin -PMyPass -iYesterdaysOrders.sql
 -oYesterdaysOrders.txt
```

当通过 sqlcmd 运行上述代码时，没有任何特别之处——只是再次得到 Windows 提示符(最可能是 C:\)，但是现在来检查一下 YesterdaysOrders.txt 文件中有什么内容：

```
C:\TYPE YesterdaysOrders.txt
```

这样得到一行：

```
C:\>TYPE YesterdaysOrders.txt
Changed database context to 'AdventureWorks2008'.
AccountNumber SalesOrderID OrderDate        ProductID     Name
                        OrderQty UnitPrice                    TotalDiscount
      LineTotal
------------- ----------- ---------------------- ---------- ---------------
------------------------------ -------- ------------------- ------------
------ ----------------------------------------
AW00000676    43659 2007-10-06 00:00:00.000      776 Mountain-100 Blac
k, 42                                   1          2024.9940
0.0000                            2024.994000
AW00000676    43659 2007-10-06 00:00:00.000      777 Mountain-100 Blac
k, 44                                   3          2024.9940
0.0000                            6074.982000
AW00000676    43659 2007-10-06 00:00:00.000      778 Mountain-100 Blac
k, 48                                   1          2024.9940
0.0000                            2024.994000
...
...
AW00000227    43662 2007-10-06 00:00:00.000      738 LL Road Frame - B
lack, 52                                1          178.5808
0.0000                             178.580800
AW00000227    43662 2007-10-06 00:00:00.000      766 Road-650 Black, 6
0                                       3          419.4589
0.0000                            1258.376700
AW00000227    43662 2007-10-06 00:00:00.000      755 Road-450 Red, 60
                                        1          874.7940
0.0000                             874.794000

(51 行受影响)
C:\>
```


www.TopSage.com

示例说明

这里开始将必需的 SQL 命令打包成单独一个脚本——首先是 USE 命令，然后是实际的 SELECT 语句。

然后使用 sqlcmd 执行语句。正如本章前文所述，–U 和–P 命令提供了登录用户名和口令信息。–i 参数告诉 sqlcmd 有一个输入文件，并且紧跟在–i 参数后面包含那个文件的名称。最后，包含了– o 参数以告诉 sqlcmd 希望将输出写到一个文件中(当然，然后提供了一个文件的名称——YesterdaysOrders.txt)。不要被这两个名称都是 YesterdaysOrders 的文件所迷惑——它们分别是带有.sql 和 .txt 后缀的单独文件，后缀用来区分它们的特定用途。

有很多种不同的参数用于 sqlcmd，但是最重要的参数是登录名、口令和说明您想做什么的参数(直接查询或者输入文件)。可以将这些参数混合和匹配，以便通过这个看似很简单的命令行工具实现相当复杂的操作。

11.4 动态 SQL：用 EXEC 命令动态生成代码

将所有需要的代码保存在脚本中是非常好的，但是如果直到运行时，才知道需要执行什么代码，那该怎么办？

提示：

到目前为止都用 sqlcmd 来运行——下面的示例应用 Management Console 来运行。

SQL Server 允许通过使用字符串操作动态构建 SQL 语句，但是带有一些缺陷。需要这么做的原因是：往往是直到运行时，才能知道一些内容的细节。语法如下所示：

```
EXEC ({<string variable>|'<literal command string>'})
```

或者：

```
EXECUTE ({<string variable>|'<literal command string>'})
```

和执行一个存储过程一样，使用 EXEC 或者 EXECUTE 没有任何区别。

为了举例，下面在 AdventureWorks2008 数据库中创建一个哑表，以从中获得动态信息：

```
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
```


www.TopSage.com

```
/* 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;
```

结果类似如下所示：

```
(73 行受影响)
```

提示：

实际的结果取决于您在本书中采用的示例、没有采用的示例，以及那些主动采用并在完成之后执行 DROP 操作的示例。在任何情况下，不要太担心。

所以现在拥有的是当前数据库中所有表的一个列表。现在假设希望选择某一个表的一些数据，但是希望只在运行时通过使用表的 ID 来识别这个表。例如，取出表中 ID 为 1 的所有数据：

```
DECLARE @SchemaName varchar(128);
DECLARE @TableName varchar(128);

-- Grab the table name that goes with our ID
SELECT @SchemaName = SchemaName, @TableName = TableName
   FROM DynamicSQLExample
   WHERE TableID = 25;

-- Finally, pass that value into the EXEC statement
EXEC ('SELECT * FROM ' + @SchemaName + '.' + @TableName);
```

如果就像这里所做的那样，将表的名称加入到 DynamicSQLExample 表，那么 25 这个 TableID 应该对应 Production.UnitMeasure 表。如果那样的话，那么得到的结果如下所示(为了简短起见，最右边的列已经略去了)。

```
UnitMeasureCode    Name                 ModifiedDate
---------------    -----------------    -----------------------
BOX                Boxes                1998-06-01 00:00:00.000
BTL                Bottle               1998-06-01 00:00:00.000
C                  Celsius              1998-06-01 00:00:00.000
...
...
PC                 Piece                1998-06-01 00:00:00.000
PCT                Percentage           1998-06-01 00:00:00.000
PT                 Pint, US liquid      1998-06-01 00:00:00.000

(38 行受影响)
```

EXEC 的陷阱

和大多数有用的功能一样，使用 EXEC 时也会产生一些问题。EXEC 的陷阱包括以下几点：

- EXEC 和调用它的代码都在单独的作用域下运行——也就是说，调用代码不能引用 EXEC 语句中的变量，并且在调用代码中的变量被解析为用于 EXEC 语句的字符串之

www.TopSage.com

后，EXEC 不能引用这些变量。如果需要在动态 SQL 和调用它的例程间传递值，考虑使用 sp_executesql。

- 默认情况下，EXEC 在当前用户的安全上下文下运行——而不是调用对象(对象经常在对象所有者的上下文下运行，而不是当前用户)的安全上下文下。
- EXEC 与调用对象运行在相同的连接和事务环境下(本书第 14 章将进一步讨论)。
- 对 EXEC 字符串执行的要求函数调用的串联必须先于实际调用的 EXEC 语句——不能在执行 EXEC 调用的相同语句中执行函数的串联。
- EXEC 不能在用户自定义函数内使用。

因为上述的每一点都有些难以理解，所以分别来介绍。

1. EXEC 的作用域

用 EXEC 语句确定变量作用域不是很直观。调用 EXEC 语句的实际语句行与运行 EXEC 语句的批处理或过程的其他部分具有相同的作用域，但是作为 EXEC 语句的结果来执行的代码被认为是处在其自身的批处理中。通常是这样的情形，下面这个示例最能说明问题：

```
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.Contact 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.Contac WHERE ContactID = 1');
-- @OutVar will still be NULL because we haven't been able to put anything
 in it
SELECT @OutVar;
```

现在，看看以上代码的输出：

```
消息 137，级别 15，状态 1，第 1 行
必须声明标量变量 "@OutVar"。
消息 137，级别 15，状态 1，第 1 行
必须声明标量变量 "@OutVar"。

----------------------------------------------------
NULL

(1 行受影响)
```

SQL Server 立即指出用户显然不知道正在发生的事情。为什么在已经声明@OutVar 情况下，还会得到一个“Must Declare”的错误消息呢？因为在外部作用域内声明了该变量——而

www.TopSage.com

不是在 EXEC 内部。

如果运行下面略有区别的代码，那么将会发生什么事情？

```
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);
```

这次返回的结果更接近于所期望的结果：

```
--------------------------------------------------------------
The Value Is Ken
```

提示：

这里使用连在一起的两个引号标志以表明实际上想要一个引号标志而不是终止字符串。

所以，这里所看见的是有两个不同的作用域操作，并且这两个作用域也不能通信。遗憾的是，如果不使用外部机制，如临时表或名为 sp_executesql 的特殊存储过程，就没有办法在内部和外部作用域之间传递信息。如果决定使用一个临时表在作用域之间通信的话，那么要记住任何在 EXEC 语句作用域内创建的临时表的存活期只能与 EXEC 语句一样长。

注意：

临时表的存活期只能和 EXEC 过程一样长，后面将在讲述触发器和存储过程时继续讨论这种行为特性。

该规则的例外情况

在 EXEC 的作用域内有个例外情况，这可在执行 EXEC 之后看到——即系统函数， 所以，像@@ROWCOUNT 这样的事物仍然可以使用。再次，看一个快速示例：

```
USE AdventureWorks2008;

EXEC('SELECT * FROM Production.UnitMeasure');

SELECT 'The Rowcount is ' + CAST(@@ROWCOUNT as varchar);
```

得到的结果为(在结果集之后)：

```
The Rowcount is 38
```


www.TopSage.com

2. 安全上下文和 EXEC

因为还没有讲述存储过程和安全性的问题，所以这时候讲述这一内容是比较困难的。不过 EXEC 命令的讨论还是属于本章范围之内而不属于“存储过程”那一章，所以在这里讨论这个问题。

如果授予某个人运行存储过程的权限，这就暗示着他们再次获得了在存储过程内执行要求的操作的权限。例如，假设有一个存储过程，用于列出在去年雇佣的所有雇员。有权限执行该存储过程的人就可以这么做(并且得到返回的结果)，即使他们没有直接访问 Employees 表的权限。这的确是很方便的，而这样做的原因将在“存储过程”一章探讨。

开发人员通常认为这样的隐含权限对 EXEC 语句也是有效的——但是实际上不是这样的。任何在 EXEC 语句内部所作的引用默认将运行在当前用户的安全上下文下。所以，假设用户有运行名为 spNewEmployees 过程的权限，但是没有访问 Employees 表的权限。如果 spNewEmployees 通过运行一个简单的 SELECT 语句得到这些值，那么一切工作正常。但是，如果 spNewEmployees 使用 EXEC 语句执行这个 SELECT 语句，那么因为用户没有在 Employees 表上运行 SELECT 语句的权限，所以 EXEC 语句将失败。

由于目前还不了解关于存储过程的更多信息，因此现在就略去更进一步的讨论，但是在后面讨论存储过程时，将回过头来讨论这个问题。

3. 函数串联和 EXEC

这实际上是不太方便的做法，因为有一种更简单的方案。简单地说，不能对 EXEC 命令中的 EXEC 字符串运行函数。例如：

```
USE AdventureWorks2008;

-- This won't work
DECLARE @NumberOfLetters int = 15;
EXEC('SELECT LEFT(Name,' + CAST(@NumberOfLetters AS varchar) + ') AS ShortName
FROM Production.Product');
GO
-- But this does
DECLARE @NumberOfLetters AS int = 15;
DECLARE @str AS varchar(255);
SET @str = 'SELECT LEFT(Name,' + CAST(@NumberOfLetters AS varchar) + ') AS ShortName
FROM Production.Product';
EXEC(@str);
```

第一个例子出错，是因为 CAST 函数需要在 EXEC 语句之前被完全地解析：

```
消息 102，级别 15，状态 1，第 9 行
'CAST'附近有语法错误。
```

而第二个例子正常工作，因为 EXEC 的输入已经是一个完整的字符串。

```
ShortName
---------------
Adjustable Race
All-Purpose Bik
```


www.TopSage.com

```
AWC Logo Cap
...
...
Women's Tights,
Women's Tights,
Women's Tights,
```

4. EXEC 和 UDF

因为至今还没有讨论过用户定义函数，所以这是个很难讲述的问题，但是可以肯定的是不允许在 UDF 中使用 EXEC 运行动态 SQL(然而，在有些情况下使用 EXEC 运行存储过程是合法的)。

11.5　控制流语句

控制流语句对于如今的所有编程语言来说是不可或缺的。如果说编写的代码不能根据条件改变运行的命令，这是无法想象的。T-SQL 提供了许多用于控制流的语句，包括：

- IF…ELSE
- GOTO
- WHILE
- WAITFOR
- TYY/CATCH

还有 CASE 语句(也就是其他语言中的 SELECT CASE、DO CASE 和 SWITCH/ BREAK)，但其控制流的功能无法与其他语言中的对应语句相比。

11.5.1　IF…ELSE 语句

IF…ELSE 语句的工作方式和其他语言中的类似，不过本书认为其与 C 语言中的实现方式最接近。基本语法为：

```
IF <Boolean Expression>
  <SQL statement> | BEGIN <code series> END
[ELSE
  <SQL statement> | BEGIN <code series> END]
```

其中的表达式几乎可以是任何布尔表达式。

提示：

这让我们想到了一个大多数 SQL 编程人员都会陷入的陷阱——未正确使用 NULL 值。我之前很多次调试存储过程，都只是为了寻找如下所示的语句：

```
IF @myvar = NULL
```

当然，这对于大多数系统来说都是不正确的，最终忽略了系统所有的 NULL 值。事实上，应写成这样：

www.TopSage.com

```
IF @myvar IS NULL
```

例外情况取决于是否将 ANSI_NULLS 选项设置为 ON 或 OFF。默认值是 ON，这时可看到上述行为。可以通过将 ANSI_NULLS 选项设置为 OFF 来改变这一行为。本书强烈建议不要这样做，因为这违反 ANSI 标准(这也是常见的错误)。

注意，只有紧跟在 IF 关键字之后的语句才被视为有条件性的(例如每个 IF)。使用 BEGIN…END，可以将多条语句作为控制流块的一部分，稍后将作讨论。

为了显示其简单的用法，运行一个常见的构建脚本的示例。假定要创建一个表(如果没有的话)，如果已经存在就不进行任何操作。可以使用 EXISTS 操作符(您可能记得我曾抱怨过，联机丛书中称 EXISTS 为关键字，而我认为它是操作符)。

```
-- We'll run a SELECT looking 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(
        Col1 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';
```

注意中间部分，只有在无匹配表的情况下，才运行 CREATE TABLE 语句。第一次检查(在脚本开始处)发现表不存在，因此可以知道 IF 条件成立，将执行 CREATE TABLE 语句。

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

(0 行受影响)

-----------------------------------------------------------------Found
Table dbo.OurIFTest

(1 行受影响)
```


www.TopSage.com

1. ELSE 语句

有条件地运行语句这一概念是相当不错的，但它并不总是处理我们希望处理的所有情况。通常，在处理 IF 条件时，希望一些特定的语句不只是在条件为真的情况下执行，而希望一些语句在条件为假或 ELSE 条件下运行。

说明：

可能会碰到不能判断布尔值的情况——结果是未知的(例如，如果与 NULL 值比较)。任何返回未知值结果的表达式将被视为 FALSE。

ELSE 语句的工作方式和其他语言中的一样。确切语法可能稍有不同，但具体细节还是一样的；如果 IF 子句中的语句不成立，那么执行 ELSE 子句中的语句。

现在对前面的示例作一下扩展，即如果未创建表，就输出警告消息。

```
-- 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(
      Col1 int PRIMARY KEY
      );
ELSE
   PRINT 'WARNING: Skipping CREATE as table already exists';
```

如果已经运行了前面的示例，那么表将已存在，运行这第二个示例将得到警告消息：

```
WARNING: Skipping CREATE as table already exists
```

2. 将代码组合成块

有时，需要像对一条语句那样对待一组语句(如果执行这块语句，就全部执行它们——否则，一个都不执行)。例如，IF 语句默认只将紧跟在 IF 关键字之后的语句视为条件代码的一部分。如果条件是要运行多条语句，那该怎么办？如果要为条件中要运行的每行代码都创建一个单独的 IF 语句，那会是件很痛苦的事。

幸运的是，和大多数其他语言中的 IF 语句一样，SQL Server 提供了一种将代码组合成块的方法，可以将它们视为一个整体。这种代码块以 BEGIN 语句开始，以 END 语句结束。其工作方式如下所示：

```
IF <Expression>
BEGIN --First block of code starts here -- executes only if
      --expression is TRUE
   Statement that executes if expression is TRUE
   Additional statements
  ...
  ...
```


www.TopSage.com

```
      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
```

注意代码块的嵌套。在每种情况下，内部块都被看成是外部块的一部分。嵌套 BEGIN…END 块的层数一般是无限制的，不过本书还是建议尽量少用一些。如果进行嵌套，则其可读性实际还是有限的——即使特别注意代码的格式问题。

为了运用上述概念，对创建表的示例进行一些修改。这次将提供一个通知消息，而不管是否创建表。

```
-- 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.';
```


www.TopSage.com

```
        PRINT 'CREATING: Table dbo.OurIFTest';
        CREATE TABLE OurIFTest(
            Col1 int PRIMARY KEY
            );
    END
ELSE
        PRINT 'WARNING: Skipping CREATE as table already exists';
```

这里混合了 IF 语句的各种使用形式。有最基本的 IF 语句——没有 BEGIN…END 或 ELSE 语句。而在其他 IF 语句中，IF 部分使用了 BEGIN…END 块，但 ELSE 没有。

提示：

这样做是为了说明如何混合使用它们。不过，这里还是建议您要注意一贯性。如果混合分组的话，就很难分清由哪个 IF…ELSE 条件控制哪条语句。在实践中的做法是，如果在给定的 IF 块中的任一语句使用 BEGIN…END，那就对 IF 语句中的每个代码块都使用它，即使对于特定条件只有一条语句。

11.5.2　CASE 语句

在某些方面，CASE 语句可以与其他一些语言的某个语句相对应。过程式编程语言中与 CASE 语句类似的语句有：

- Switch：C、C#、C++、Java、php、Perl 和 Delphi
- Select Case：Visual Basic
- Do Case：Xbase
- Evaluate：COBOL

我确信还有其他一些编程语言——上述这些只是来自我多年来使用过的语言。T-SQL 中的 CASE 语句的唯一缺点是，它基本上只是个替换操作符而不是控制流语句。

有多种方法编写 CASE 语句——用输入表达式或布尔表达式。第一选择是使用输入表达式，它将与每个 WHEN 子句中使用的值比较。SQL Server 文档将这称为简单 CASE 语句：

```
CASE <input expression>
WHEN <when expression> THEN <result expression>
[...n]
[ELSE <result expression>]
END
```

另一选择是给每个 WHEN 子句提供一个布尔表达式，将评估其值为 TRUE 还是 FALSE。这被称为搜索 CASE 语句：

```
CASE
WHEN <Boolean expression> THEN <result expression>
[...n]
[ELSE <result expression>]
END
```

CASE 语句的最大的好处是它可与 SELECT 语句“内联”使用(也就是作为其不可分割的一部分)。其作用实际上是非常强大的。

www.TopSage.com

1. 简单 CASE 语句

简单 CASE 语句采用一个比较结果为布尔值的表达式。看一下下列示例：

```
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;
```

有些读者可能不清楚，%运算符用于取模。取模运算与除运算类似，不过它只返回余数——因此，16%4=0(16 可被 4 除尽)，但 16%5=1(16 除以 5 还余 1)。在这个示例中，由于除以 10，因此取模运算返回了所计算数字的最后一位。

结果如下所示：

```
SalesOrderID Last Digit  Position
------------ ----------- --------------
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
43701         1             First
(10 行受影响)
```

注意，无论何时列表中有匹配值，都会调用 THEN 子句。由于有一个 ELSE 子句，与前面的值都不匹配的值将被赋予 ELSE 子句中的值。如果省略 ELSE 子句，那么相对应的值将为 NULL。

再看一个详述表达式的用法的示例。这次，将使用查询中的另一列。

```
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'
```


www.TopSage.com

```
    END
FROM Sales.SalesOrderDetail
ORDER BY SalesOrderID DESC;
```

注意，这里的每一步都使用了等式，但它仍然正常工作。

```
OrderLastDigit ProductLastDigit     How Close?
-------------- ----------------     -------------------
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!
0                4                    Exact Match!
(10 行受影响)
```

只要表达式的计算结果为与输入表达式的类型兼容的特定值，就可以分析表达式并应用适当的 THEN 子句。

2. 搜索 CASE 语句

搜索 CASE 语句的工作方式与简单 CASE 语句很接近，不过有如下两处不同。

- 没有输入表达式(即 CASE 关键字和第一个 WHEN 之间的部分)。
- WHEN 表达式必须判断为一个布尔值(而在刚才看到的简单 CASE 示例中，使用了像 1、3 和 ProductID +1 这样的值)。

搜索 CASE 语句最棒的地方是可以完全更改构成表达式基础的内容——根据可能的不同情形，混合搭配列表达式。

同样，理解其工作方式的最好途径是运用示例：

```
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;
```

这与简单 CASE 语句有很大不同，但它仍然可以运行。

```
OrderLastDigit ProductLastDigit    How Close?
-------------- ----------------    ------------------------
3                2                   More Than One Apart
3                9                   More Than One Apart
3                8                   More Than One Apart
```


www.TopSage.com

```
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
0                    4                    Ends With Less Than Three
(10 行受影响)
```

有关 SQL Server 如何作判断有以下一些注意点。

- 即使两个条件被判断为 TRUE，也只使用第一个条件。例如，倒数第二行满足第 1 个条件(最后一位小于 3)和第 3 个条件(最后一位与 ProductID 之差小于等于 1)。对于许多语言来说，包括 Visual Basic，这种语句总是这样工作。然而，如果使用 C 语言(或其他类似语言)，那么在编写代码时要记得这一点；不要求“中断”的语句——总是在一个条件满足后终止。
- 可以在条件表达式中混合搭配使用的字段。在这个示例中，使用了 SalesOrderID 和 ProductID，并对两者作了混合使用。
- 可以执行任何表达式，只要最后它被判断为布尔结果。

现在看一个稍复杂的示例。在这个示例中，并不准备进行混合搭配——而是只用采用一列(可以改变测试的列，但大部分时候并不需要)。下面将处理一个更现实的场景，用于更大的电子商务站点。

这个场景是这样的：营销人员喜欢净价。他们讨厌在成本上加 10%的价，使得价格为$10.13 或$23.19。他们喜欢以 49、75、95 或 99 这样的数字结尾的灵活价格。在这里，假定创建一个新的价格列表用于分析，并希望它满足特定条件。

对于零头小于 50 美分的新价格(例如前面的$10.13)，营销人员希望保持价格的美元数不变，但使零头增加为 49 美分(即$10.49)。将零头为 50~75 美分的价格改为零头为 75 美分，零头大于 75 美分的价格改为零头为 95 美分。具体所希望的情况如表 11-2 所示：

表 11-2

如果新价格为	那么应改为
$10.13	$10.49
$17.57	$17.75
$27.75	$27.75
$79.99	$79.95

从技术上讲，这可以使用嵌套的 IF…ELSE 语句来实现，但它有下列缺点。

- 可读性差——特别是如果规则较复杂的话。
- 必须使用游标实现代码并且一次检查一行。

总之比较麻烦。

而 CASE 语句使这一过程变得非常简单。而且，可以使条件与查询内联，将它作为 SET 操作的一部分——这意味着将获得比使用游标更好的性能。

www.TopSage.com

营销部门希望看到将价格提升 10%后的结果，因此将 10%的加价结合到 CASE 语句中，同时稍作一些分析，就可以得到所要的数字。

```
USE AdventureWorks2008;
GO

/* I'm setting up some holding variables here. This way, if we get asked
** to run the query again with a slightly different value, we'll only have
** to change it in one place.
*/
DECLARE @Markup money;
DECLARE @Multiplier money;

SELECT @Markup = .10;                    -- Change the markup here
SELECT @Multiplier = @Markup + 1; -- We want the end price, not the amount
                                  -- of the increase, so add 1

/* Now execute things for our results. Note that we're limiting things
** to the top 10 items for brevity -- in reality, we either wouldn't do this
** at all, or we would have a more complex WHERE clause to limit the
** increase to a particular set of products
*/
SELECT TOP 10 ProductID, Name, ListPrice,
   ListPrice * @Multiplier AS "Marked Up Price", "New Price" =
   CASE WHEN FLOOR(ListPrice * @Multiplier + .24)
             > FLOOR(ListPrice * @Multiplier)
                      THEN FLOOR(ListPrice * @Multiplier) + .95
        WHEN FLOOR(ListPrice * @Multiplier + .5) >
             FLOOR(ListPrice * @Multiplier)
                      THEN FLOOR(ListPrice * @Multiplier) + .75
        ELSE FLOOR(ListPrice * @Multiplier) + .49
   END
FROM Production.Product
WHERE ProductID % 10 = 0 -- this is just to help the example
ORDER BY ProductID DESC;
```

FLOOR 函数是比较简单的——它获取提供的值并向下舍入为最接近的整数。

当听到某人说出“分析”二字时，我感到非常怀疑，特别是这是从营销或销售人员口中说出的。不要误会，其实这些人和我一样，也是在做本职工作。问题是，他们如果提出了某个要求，下次可能还会提出一个类似的要求。因此，我准备构建一个脚本——这样当他们要求将价格提升 15%时，只需要改变初始值@Markup。看一下提价 10%的结果：

```
ProductID  Name                          ListPrice Marked Up Price  New Price
--------   ------------------------     --------- --------------  -----------
990        Mountain-500 Black, 42         539.99   593.989          593.9500
980        Mountain-400-W Silver,38       769.49   846.439          846.4900
970        Touring-2000 Blue, 46         1214.85  1336.335         1336.4900
960        Touring-3000 Blue, 62          742.35   816.585          816.7500
950        ML Crankset                    256.49   282.139          282.4900
940        HL Road Pedal                   80.99    89.089           89.4900
930        HL Mountain Tire                35.00    38.50            38.7500
920        LL Mountain Frame-Silver,52    264.05   290.455          290.4900
910        HL Mountain Seat/Saddle         52.64    57.904           57.9500
```


www.TopSage.com

```
900        LL Touring Frame - Yellow,50   333.42   366.762          366.9500

(10 行受影响)
```

仔细查看一下，就可发现结果正是所希望的。而且，这里不必构建游标。

11.5.3 用 WHILE 语句进行循环

WHILE 语句的工作方式与其他语言中的类似。基本上就是，每次到达循环顶部时测试一个条件。如果条件仍为 TRUE，那么再次执行循环——否则就退出。

其语法如下所示：

```
WHILE <Boolean expression>
     <sql statement> |
[BEGIN
  <statement block>
  [BREAK]
  <sql statement> | <statement block>
  [CONTINUE]
END]
```

尽管可以只执行一条语句(就像 IF 语句一样)，但几乎所有的 WHILE 关键字之后都会跟有一个 BEGIN…END 语句块。

BREAK 语句用于退出循环，而不需要等待到达循环底部和表达式被重新评估。

提示：

使用 BREAK 语句通常是较差的做法。本书对这一点保持中立态度。不过尽可能避免使用它。大部分情况，可以通过移动一两条语句来避免它，而结果是一样的。其优点通常是获得更具可读性的代码。如果只有一个进入/退出点，那么这更易于处理循环结构(或就此而言的任何结构)。而使用 BREAK 违反了这一理念。

虽说如此，有时重新格式化代码来避免 BREAK 语句，但只会使事情变得更糟。另外，我曾看到有人为了避免使用 BREAK 语句，导致所编写出来的代码运行速度更慢。

CONTINUE 语句与 BREAK 语句完全相反。简言之，它告诉 WHILE 循环回到起点。不管是在循环中的哪个位置，都会立即回到顶部，重新计算表达式(如果表达式不再为 TRUE，就退出)。

下面将用一个简短的示例说明。如前所述，在无游标的情况下，WHILE 循环是很少见的。不过这里的示例只是为了演示。

这里准备使用 WHILE 循环和 WAITFOR 命令创建一个监控进程(11.5.4 节将介绍 WAITFOR 命令)。该进程用于每天一次自动更新统计信息。

```
WHILE 1 = 1
BEGIN
   WAITFOR TIME '01:00';
   EXEC sp_updatestats;
   RAISERROR('Statistics Updated for Database', 1, 1) WITH LOG;
END
```


www.TopSage.com

这将在每天凌晨 1 点更新数据库中的每个数据表的统计信息，并将日志项写入 SQL Server 日志和 Windows 应用程序日志。如果想查看该进程是否工作，则可以使其整夜运行并在早上检查日志。

提示：

注意，这样的无限循环和调度任务不同。如果希望每天运行某项操作，可以使用 Management Studio 建立作业。除了不需要一直打开连接(前面的示例会做这个工作)外，还可以根据脚本成功与否执行动作。另外，还可以使用电子邮件和 netsend 命令发送有关完成状态的消息。

11.5.4　WAITOR 语句

很多时候，您并不希望或无法使某操作正好在某个时点发生，同时也不希望操作被挂起，直到适当的时间执行该操作。

这没问题——可以使用 WAITFOR 语句并使 SQL Server 等待。语法非常简单：

```
WAITFOR
   DELAY <'time'> | TIME <'time'>
```

WAITFOR 语句所做的就是等待对参数所指定的操作发生。可以为某操作的发生指定明确的时间，或是指定等待一段时间后执行该操作。

1. DELAY 参数

DELAY 参数指定了等待的时间段。不能指定天数——只能指定小时数、分钟数和秒数。允许延迟的最长时间为 24 小时。因此，例如下列语句：

```
WAITFOR DELAY '01:00';
```

将运行 WAITOR 语句前的任何代码，然后到达 WAITFOR 语句，停止一个小时，之后，继续执行下一条语句。

1. TIME 参数

TIME 参数指定到达指定时间的等待时间。这也不能指定日期——只能是 24 小时制的某个时间。同样，可延迟的最长时间也是一天。例如下列代码：

```
WAITFOR TIME '01:00';
```

将运行 WAITOR 语句前的任何代码，然后到达 WAITOR 语句，直到凌晨 1 点停止执行，之后执行 WAITFOR 语句后的下一条语句。

11.5.5　TRY/CATCH 块

在以前(指 SQL Server 2005 之前)，错误处理的选项是相当有限的。用户可以检查错误条件，但必须主动去做。有时，可能会有一些导致退出存储过程或脚本的错误，而根本无法捕获错误(现在这仍会发生，但有一些选项去降低错误的机率)。这里将把有关错误处理的更详细的讨论放到第 12 章有关存储过程的讨论中去，而介绍新的 TRY/CATCH 块的基本原理。

www.TopSage.com

在 SQL Server 中 TRY/CATCH 块的工作方式与那些派生自 C 的语言(C、C++、C#、Delphi，及其他语言)中的相似。其语法如下所示：

```
BEGIN TRY
   { <sql statement(s)> }
END TRY
BEGIN CATCH
   { <sql statement(s)> }
END CATCH [ ; ]
```

简而言之，SQL Server 将“尝试”运行 BEGIN TRY…END TRY 块中的任何语句。如当且仅当错误级别为 11~19 的错误条件发生，那 SQL Server 才立即退出 TRY 块并转到 CATCH 块中的第一行。因为错误级别不只是从 11~19，还有更多可能的错误级别，如表 11-3 所示。

表 11-3

错误级别	性质
1~10	只是信息错误。这包括了上下文更改，如调整了设置或在进行聚合运算时发现 NULL 值。这些不会触发 CATCH 块，因此如果需要测试这个级别的错误，需要手动选用@@ERROR 来实现
11~19	相对严重的问题，但可由代码处理(例如违反外键)。有些错误比较严重，而可能不希望继续处理(例如超出内存错误)，但至少可以捕获它们并适当退出
20~25	非常严重。这些通常是系统级错误。服务器端代码从来不知道此类错误发生，因为脚本和连接将立即终止

记住，如果需要处理超出 11~19 级范围的错误，那么需要作其他规划。

现在，为了进行测试，对之前介绍 IF…ELSE 语句时构建的 CREATE 脚本进行修改。您可能记得，最初测试表是否已存在的部分原因是避免创建一个导致脚本失败的错误条件。这种测试是以前的做法(其他选项也没有更多的内容)。通过采用 TRY/CATCH 块，只要尝试 CREATE 操作，如果有错误就进行处理。

```
BEGIN TRY
   -- Try and create our table
   CREATE TABLE OurIFTest(
      Col1 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(),
```


www.TopSage.com

```
        @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，这里使用了一些特殊的函数来检索错误条件，如表 11-4 所示。

注意：

本书将它们移到局部变量中，这样就不会丢失。

表　11-4

函　　数	返　回　值
ERROR_NUMBER()	实际错误号。如果这是个系统错误，那么在 sysmessage 表(使用 sys.messages 查看)中会有一项与该错误匹配，并且包含将从其他错误相关函数中获取的信息
ERROR_SEVERITY()	等同于本书和联机丛书中提到的“错误级别”
ERROR_STATE()	这个函数用作位置标记。对于系统错误来说，其值总为 1。在第 12 章深入讨论错误处理时，将看到如何引发自定义错误。那时，可用状态表明错误发生在存储过程、函数或触发器的哪个位置(这有助于在多个位置的其中一个位置中处理指定的错误)
ERROR_PROCEDURE()	在前面的示例中未使用该函数，因为它只与存储过程、函数和触发器相关。这提供了导致错误的存储过程的名称——如果存储过程是嵌套的，使用它就很方便，因为导致错误的存储过程可能不是实际处理该错误的那个
ERROR_LINE()	错误所在的行号
ERROR_MESSAGE()	消息文本。对于系统消息，这与从 sys.message 函数选择消息时所看到的一样。对于用户自定义错误，这是提供给 RAISERROR 函数的文本

这里的示例利用了一个已知的错误 id，如果试图创建已存在的对象，SQL Server 就会引发该错误。通过从 sys.message 表函数中选择，可以看到所有系统错误消息。

提示：

从 SQL Server 2005 起，sys.messages 输出变得很长，以至于很难通过扫描发现到要找的消息。此处所采用的解决办法不完美但很有效——人为创建要查找的错误，然后查看系统提供的错误号(简单问题简单解决)。

本节只是执行想执行的代码(这里是 CREATE 语句)并处理出现的错误——就是这样。第 12 章将更详细地介绍错误处理。同时，可使用 TRY/CATCH 为脚本提供基本的错误处理。

www.TopSage.com

11.6 本章小结

理解脚本和批处理是理解 SQL Server 编程的基础。而脚本和批处理的概念是实现多种功能的基础，包括编写构建全部数据库的脚本、编写代码存储过程和触发器等。

局部变量的作用域只限于一个批处理。即使已在同一脚本内声明了变量，如果在一个新的批处理中引用它之前，不重新声明该变量(并且赋值)，那么将仍然会得到错误消息。

有许多系统函数可用。这里只是提供了一个最有用的系统函数的列表，但实际还有很多。对于比较晦涩难懂的函数，可以查看联机丛书或本书后的附录 A。系统函数不需要声明，并且总是可用的。有些函数的作用域是整个服务器，而有些只返回当前连接的特定值。

可以使用批处理在脚本的不同部分创建优先权。第一个批处理从脚本的起点处开始，结束于脚本的末端或者第一条 GO 语句——无论哪个批处理先出现。下一个批处理(如果有的话)从第一个批处理结束后的那行开始，执行到脚本的末端或者下一条 GO 语句——同样无论哪个批处理先出现。这个过程继续到脚本的末端。来自脚本顶部的第一个批处理第一个被执行，第二个批处理则第二个被执行，以此类推。每个批处理内的所有命令必须通过查询分析器的验证，否则不执行该批处理中的任何命令；然而，任何其他的批处理将单独地分析并且将仍然被执行(如果它们通过分析器的验证)。

另外，还介绍了如何动态地创建和执行 SQL。这可用于处理不能完全预测的场景，或是构造语句所需的内容实际上是一块数据这样的情况。

最后介绍了 SQL Server 提供的控制流结构。通过混合使用这些结构，可以有条件地执行代码、创建循环或是提供某种字符串替换。

在下面几章中，将深化脚本和批处理的概念，把它们应用到存储过程、用户自定义函数和触发器——即 SQL Server 中最接近于实际程序的内容。

11.7 练习

1. 编写一个简单脚本，创建两个整型变量(一个名为 Var1，另一个名为 Var2)，将它们的值分别设置为 2 和 4，然后输出这两个变量的和。
2. 创建一个名为 MinOrder 的变量并且用 AdventureWorks2008 中 CustomerID 1 的打折后的最小行项总额填充该变量(注意：这里在处理货币，所以不要想当然地使用 int 类型)。输出 MinOrder 的最终值。
3. 使用 sqlcmd 将查询 SELECT COUNT(*)FROM Customers 的结果输出到控制台窗口。

www.TopSage.com

第12章

存储过程

存储过程很有用。如果您一直使用的是过程式编程语言，那么本章的内容可能正是适合您的。现在，看一下 SQL Server 代码的主要种类，不过在深入研究它之前，需要了解的是——其种类数目可能低于也可能高于您所期望的。好在有了 SQL Server 2008，就有了.NET 支持——使得原先很多不可能的事都变得可能。

存储过程(stored procedure)有时也称为 sproc，它是真正的脚本——或者更准确地说，它是批处理(batch)——它存储于数据库中而不是单独的文件中。无论如何，这个比较并不是很确切——存储过程中有输入参数、输出参数以及返回值等，而脚本中不会有这些内容——但也不是偏得太离谱。

目前，SQL Server 中唯一的编程语言依然是 T-SQL，如果要说到真正的编程语言，它离过程语言还是有差距的。但是，如果论及 T-SQL 的作用——数据的定义、操作和访问，T-SQL 要强于 C、C++、Visual Basic、Java、Delphi，以及其他的语言。但是 T-SQL 的强大功能在数据访问和管理上也是有限的。简而言之，它有能力完成大部分简单的任务，但是它并不总是完成这些任务的最佳方式。

就本章而言，不必太在意 T-SQL 的缺点——相反，要将精力集中在如何最大限度地利用 T-SQL，以及.NET 所提供的一些支持。本章将介绍参数、返回值、流控制、循环结构、基础的和高级的错误捕获等。简而言之，本章将介绍很多的主题。所有主要的主题内容都用单独的章节进行介绍，这样可以每次学习一部分的内容，但是首先学习一下创建存储过程的基础知识。

12.1 创建存储过程：基本语法

创建存储过程的方法和创建数据库中任何其他对象一样，除了它使用的 AS 关键字外(在介绍视图时出现过 AS 关键字)。存储过程的基本语法如下：

```
CREATE PROCEDURE|PROC <sproc name>
   [<parameter name> [schema.]<data type> [VARYING] [= <default value>] [OUT [PUT]]
[READONLY]
   [,<parameter name> [schema.]<data type> [VARYING] [= <default value>] [OUT[PUT]]
```

www.TopSage.com

```
[READONLY]
[, ...
   ...
     ]]
[WITH
 RECOMPILE| ENCRYPTION | [EXECUTE AS { CALLER|SELF|OWNER|<'user name'>}]
[FOR REPLICATION]
AS
 <code> | EXTERNAL NAME <assembly name>.<assembly class>.<method>
```

在这里，仍然使用基本的 CREATE <Object Type> <Object Name>语法，它是每个 CREATE 语句的主干。唯一特别的是要选择创建 PROCEDURE 还是 PROC。这两个选项都可行，但也有例外情况。因此，建议对所选择的选项保持一致(就个人而言，我喜欢选用 PROC，因为输入的字母更少)。存储过程的名称必须遵循第 1 章的命名规则。

在对存储过程命名完之后，接着是参数列表。参数是可选的，本章稍后将对其进行讨论。

最后一点也很重要，就是关键字 AS 其后的实际代码。

基本存储过程的示例

要演示基本的存储过程语法，最佳示例要属那种最基本的存储过程，该存储过程返回表中所有数据行的所有列——简而言之，就是针对表中所有数据。

这里，需要您已经知道如何用查询返回表中的所有内容(提示：使用 SELECT * FROM…)。如果不会的话，建议您参考有关基本查询语法的章节。要创建能执行该基本查询功能(这里将对 HumanResources 模式中的 Employee 表执行)的存储过程，只要在存储过程语法的代码区域中加入查询语句：

```
USE AdventureWorks2008

GO
CREATE PROC spEmployee
AS
   SELECT * FROM HumanResources.Employee;
```

这很简单。您可能要问为什么要在 CREATE 语法之前加上 GO 这个关键字(如果只是运行简单的 SELECT 语句，就不需要使用它)，那是因为大多数非表(non-table)的 CREATE 语句不能与其他任何代码共享批处理。事实上，即使是 CREATE TABLE 语句，不使用 GO 也可能有危险。在这里，禁止将 USE 命令与 CREATE PROC 语句一起使用，否则会产生错误。

既然创建了存储过程，那么执行它来看一下结果：

```
EXEC spEmployee;
```

如果运行嵌入在存储过程中的 SELECT 语句，就会得到应该得到的结果：

```
BusinessEntityID NationalIDNumber LoginID
---------------- --------------- ----------------------------------
1                 295847284        adventure-works\ken0      ---
2                 245797967        adventure-works\terri0         ---
```


www.TopSage.com

```
3                    509647174          adventure-works\roberto0        ---
...
288                  954276278          adventure-works\rachel0         ---
289                  668991357          adventure-works\jae0      ---
290                  134219713          adventure-works\ranjit0         ---
(290 行受影响)
```

注意：

此处删除了结果集中右侧的几列(受书页面所限，只能放这么多)——实际将获得更多的列，如 JobTile、BirthDate 等。

这就创建了第一个存储过程。这很简单。坦率地说，对于大部分情况，存储过程的编写并不像数据库工作人员所说的那么难(出于职业保护的目的)。不过，也可能会有比较难的情况，毕竟这里只是初级内容。

12.2 使用 ALTER 修改存储过程

必须承认的是——本书从视图那一章剪切了将在本节和 12.3 节“删除存储过程”中讲述的几乎所有内容。说明这一点，是想让读者知道从 ALTER 语句的使用方式来看，不管是视图还是存储过程，其工作原理都是一样的。

在使用 T-SQL 编辑存储过程时要记住，这是在完全替换现有的存储过程。使用 ALTER PROC 和 CREATE PROC 语句的唯一区别包括以下几点。

- ALTER PROC 期望找到一个已有的存储过程，而 CREATE 则不是。
- ALTER PROC 保留了存储过程上已经建立的任何权限。它在系统对象中保留了相同的对象 ID 并允许保留依赖关系。例如，如果过程 A 调用过程 B，并删除和重新创建了过程 B，那么就不能再看到这两者间的依赖关系。如果使用 ALTER，则依赖关系仍然存在。
- ALTER PROC 在可能调用被修改的存储过程的其他对象上保留了任何依赖信息。

后面这两点非常重要。

注意：

如果先执行 DROP，然后使用 CREATE，那么其效果与使用 ALTER PROC 语句几乎是一样的，除了一个很重要的区别——如果使用 DROP 和 CREATE，则需要完全重建有关允许和不允许谁使用这个存储过程的权限。

12.3 删除存储过程

这是最简单的，通过下列语句：

```
DROP PROC|PROCEDURE <sproc name>[;]
```

就完成了存储过程的删除。

www.TopSage.com

12.4 参数化

存储过程提供了一些(或者对于.NET 来说是很多的)过程式的能力，它也提升了性能(稍后将详细介绍)，但是如果存储过程没有接受一些数据，告诉其要完成的任务，则在大多数情况下存储过程不会有太多的帮助。例如，对于存储过程 spDeteleVendor，如果不指定要删除的供应商，则它不会有太多用处，所以这里使用输入参数。同样地，经常想从存储过程中获取信息——不只是一个或多个表数据的记录集，而是更直接的一些信息。例如，更新了表中的一些记录，并且想知道更新的数量。通常，不太容易以记录集的形式获取该信息，而要使用输出参数。

在存储过程的外部，可以通过位置或者引用传递参数。在存储过程的内部，不用关心参数传递的方式——因为它们使用同样的方式声明。

声明参数

声明参数时需要以下两到四部分的信息：

- 名称
- 数据类型
- 默认值
- 方向

其语法如下所示：

```
@parameter_name [AS] datatype [= default|NULL] [VARYING] [OUTPUT|OUT]
```

对于名称，有一组简单的规则。首先，它必须以@符号(和变量一样)开始。此外，除了不能内嵌空格外，其规则和第 1 章介绍的命名规则是相同的。

数据类型和名称一样，必须像变量那样声明，采用 SQL Server 内置的或用户自定义的数据类型。

提示：

声明数据类型时需要记住的是，当声明 CURSOR 类型参数时，必须也使用 VARYING 和 OUTPUT 选项。一般很少使用这种类型的参数，而且这也超出了本书的讨论范围，但是还是要记住它，以防在联机丛书或其他文档中看到它时产生疑问。

同样要注意的是，OUTPUT 可简写为 OUT。

在默认值方面，参数与变量不同。对于同样的情况，变量一般初始化为 NULL 值，而参数不是。事实上，如果不提供默认值，则会假设参数是必须的，并且当调用存储过程时需要提供一个初始值。

例如，创建一个与前面的存储过程稍有不同的存储过程。这次将从 Person.Person 表提供名字信息并采用一个针对姓的过滤器。

```
USE AdventureWorks2008;

GO
```


www.TopSage.com

```
CREATE PROC spEmployeeByName
   @LastName nvarchar(50)
AS

SELECT p.LastName, p.FirstName, e.JobTitle, e.HireDate
FROM Person.Person p
JOIN HumanResources.Employee e
   ON p.BusinessEntityID = e.BusinessEntityID
WHERE p.LastName LIKE @LastName + '%';
```

首先提供所需的默认值，尝试这个存储过程：

```
EXEC spEmployeeByName 'Dobney';
```

结果是一个非常简短的列表(事实上只是一个雇员)：

```
LastName     FirstName  JobTitle                          HireDate
----------   ---------  ------------------------------   ----------
Dobney       JoLynn     Production Supervisor - WC60     1998-01-26
```

说明：

要谨慎使用通配符匹配，例如前面代码中的 LIKE 语句。这个特别的例子很可能正常工作，因为通配符在最后。记住，在搜索一开始运用通配符可有效避免 SQL Server 使用索引，因为任何开始字符都可能是匹配的。

现在看一下，如果不提供默认值会发生什么情况：

```
EXEC spEmployeeByName;
```

SQL Server 立刻通知出错：

```
消息 201，级别 16，状态 4，过程 spEmployeeByName，第 0 行
过程或函数'spEmployeeByName' 需要参数 '@LastName'，
但未提供该参数。
```

因为没有提供默认值，因此就假定参数是必须的。

1. 提供默认值

为了使参数是可选的，必须提供默认值。方法是在数据类型后在逗号之前添加“=”符号和作为默认值的值。这样做之后，存储过程的用户可决定对此参数数不提供值或是提供他们自己的值。

例如，如果希望前面示例中的参数为可选的，可修改参数声明使之包括默认值：

```
USE AdventureWorks2008;

DROP PROC spEmployeeByName; -- Get rid of the previous version
GO
CREATE PROC spEmployeeByName
@LastName nvarchar(50) = NULL
AS
```


www.TopSage.com

```
IF @LastName IS NOT NULL
   SELECT p.LastName, p.FirstName, e.JobTitle, e.HireDate
   FROM Person.Person p
   JOIN HumanResources.Employee e
      ON p.BusinessEntityID = e.BusinessEntityID
   WHERE p.LastName LIKE @LastName + '%';
ELSE
   SELECT p.LastName, p.FirstName, e.JobTitle, e.HireDate
   FROM Person.Person p
   JOIN HumanResources.Employee e
      ON p.BusinessEntityID = e.BusinessEntityID;
```

注意一下这里是如何在脚本中使用第 11 章介绍的控制流结构来决定运行哪个更好的查询。两者区别很小，只是添加了 WHERE 子句。

提供了新默认值后，现在可在无参数的情况下运行存储过程：

```
EXEC spEmployeeByName;
```

正如所预料的，得到更完整的结果集：

```
LastName       FirstName    JobTitle                          HireDate
------------   ----------   ------------------------------    ----------
Sánchez        Ken          Chief Executive Officer           1999-02-15
Duffy          Terri        Vice President of Engineering     1998-03-03
Tamburello     Roberto      Engineering Manager               1997-12-12
...
...
Valdez         Rachel       Sales Representative              2003-07-01
Pak            Jae          Sales Representative              2002-07-01
Vakey Chudu Ranjit          Sales Representative              2002-07-01

(290 行受影响)
```

如果像前面一样，用同样的参数(Dobney)运行该存储过程，那么结果是一样的——唯一不同是现在允许参数是可选的，可以处理不提供参数的情况。

2. 创建输出参数

有时希望给调用存储过程的对象传递非记录集的信息。此处在上面两个存储过程基础上创建一个修改版本作为示例。

最常见的就是使用存储过程对表执行生成标识值的插入操作。通常，调用存储过程的代码希望知道这一过程完成时的标识值。

为此，将利用 AdventureWorks2008 数据库中的一个存储过程——uspLogError，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
```


www.TopSage.com

```
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.'
            +'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;
```


www.TopSage.com

注意这里突出显示的部分——它们是输出参数的核心所在。第一个部分声明参数为输出参数。第二个部分利用标识值进行插入，最后，SET 语句捕获标识值。如果这一过程存在，@ErrorLogID 中的值就传递给调用脚本。

这里利用第 11 章最后的 TRY/CATCH 示例，不过这次调用 uspLogError：

```
USE AdventureWorks2008;

BEGIN TRY
-- Try and create our table
CREATE TABLE OurIFTest(
   Col1 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
```

如果在没有 OurIFTest 表的数据库中运行它，那么将得到简单的结果：

```
命令已成功完成。
```

但如果在 OurIFTest 已存在的情况下运行上述代码(例如，如果之前未运行过 CREATE 代码，而运行上述代码两次)，那将得到下列错误：

```
WARNING: Skipping CREATE as table already exists
A error was logged. The Log ID for our error was 3
```

注意：

实际错误号取决于在您运行该测试前，ErrorLog 表中是否插入了其他错误。

现在对错误日志表运行简单的 SELECT 查询：

```
SELECT *
FROM ErrorLog
WHERE ErrorLogID = 3; -- change this value to whatever your
                      -- results said it was logged as
```

可看到错误被正确记录了：

```
ErrorLogID UserName       ErrorMessage
---------- -----------    ---------------------------------------------
3           dbo           数据库中已存在名为'OurIFTest'的对象...
```


www.TopSage.com

```
(1 行受影响)
```

对于存储过程本身以及调用脚本对它的使用，需要注意以下几点。

- 对于存储过程声明中的输出参数，需要使用 OUTPUT 关键字。
- 和声明存储过程时一样，调用存储过程时必须使用 OUTPUT 关键字。这样就对 SQL Server 作了提前通知，告诉它参数所需要的特殊处理。但是需要注意的是，如果忘记包含 OUTPUT 关键字，不会产生运行时错误(不会得到关于它的任何消息)，但是输出参数的值不会传入变量中(得到的很有可能是 NULL 值)。这意味着碰到了“非预期的结果”。
- 赋给输出结果的变量不需要和存储过程中的内部参数拥有相同的名称。例如，在前面的存储过程中，内部参数叫做 @ErrorLogID，而被传递值的变量叫做 @MyOutputParameter。
- EXEC(或 EXECUTE)关键字是必须的，因为对存储过程的调用并不是批处理要做的第一件事(如果存储过程的调用是批处理要做的第一件事，则可以不使用 EXEC)，无论如何还是建议您使用它。

12.5　通过返回值确认成功或失败

可以看到返回值用于多种用途。首先就是返回数据，例如标识值或者是存储过程影响的行数。而其实际作用是返回值可用来确定存储过程的执行状态。

提示：

如果说这里看上去像是介绍如何使用返回值，那也不奇怪，因为此处确实有此意。我最初学到的是利用返回值作为一种“技巧”来避免使用输出参数——实际上，是作为一种捷径。和大部分捷径一样，其问题是删减了一些内容，在这里，则是删减了相当重要的内容。

当需要返回真正的错误代码时，使用返回值作为给调用例程返回数据的方法会使返回代码的意思变得模糊。简单地说，不要这样做！

返回值指示了存储过程的成功或者失败，甚至是成功或失败的程度或属性。对于 C 程序员来说，这是个相当简单的策略——使用函数的返回值作为成功代码是很常见的，任何非 0 值都说明某种问题。如果在 SQL Server 中坚持使用默认的返回代码，则会发现规则同样适用。

如何使用 RETURN

事实上，不管是否提供返回值，程序都会收到一个返回值。SQL Server 默认会在完成存储过程时自动返回一个 0 值。

为了从存储过程向调用代码传递返回值，只需要使用 RETURN 语句：

```
RETURN [<integer value to return >]
```


www.TopSage.com

注意：

返回值必须为整数。

关于 RETURN 语句，最重要的是知道它是无条件地从存储过程中退出的。这是指，无论运行到存储过程的哪个位置，在调用 RETURN 语句之后将不会执行任何一行代码。

这里的无条件，并不是说无论执行到代码的何处都将执行 RETURN 语句。相反，可以在存储过程中有多个 RETURN 语句，只有当代码的标准条件结构发出命令的时，才会执行这些 RETURN 语句。一旦发生，就不能再返回了。

此处以一个简单的测试存储过程来说明 RETURN 语句的影响。

```
USE AdventureWorks2008;
GO

CREATE PROC spTestReturns
AS
 DECLARE @MyMessage        varchar(50);
 DECLARE @MyOtherMessage   varchar(50);

 SELECT @MyMessage = 'Hi, it''s that line before the RETURN';
 PRINT @MyMessage;
 RETURN;
 SELECT @MyOtherMessage = 'Sorry, but we won''t get this far';
 PRINT @MyOtherMessage;
RETURN;
```

注意：

该存储过程并未选择在声明中初始化两个消息变量。原因何在？在这里，我认为这样可使代码更具可读性——对于大多数其初始值不只是几个字符的字符串变量来说，同样是如此。

现在已经有了存储过程，但需要一小段脚本作测试。想看到的运行结果是：

- 输出的结果
- RETURN 语句返回的值

为了能捕获 RETURN 语句的值，需要在 EXEC 语句中把值赋给变量。例如，下面的代码将把返回值赋给@ReturnVal 变量：

```
EXEC @ReturnVal =spMySproc
```

现在使用一个更有用的脚本来测试该存储过程：

```
DECLARE @Return int

EXEC @Return =spTestReturns
SELECT @Return
```

虽然简单但是还不错。当运行时，可以看到 RETURN 语句确实在运行其他代码前终止了代码运行：

```
Hi, it's that line before the RETURN

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


www.TopSage.com

```
0
(1 行受影响)
```

同样也得到了该存储过程的返回值，该返回值为 0。注意即使没有指定一个特定的返回值，这个返回值仍为 0——这是因为默认值为 0。

提示：

考虑一下这个问题——如果返回值默认为 0，就意味着默认的返回值实际上“没有错误”。这有些危险。此处的重点是确保总是显式定义返回值，这样可以确保返回期望的值而不是意外的值。

现在修改存储过程来验证可以用返回值形式传递任何整数值：

```
USE AdventureWorks2008;
GO

ALTER PROC spTestReturns
AS
   DECLARE @MyMessage varchar(50);
   DECLARE @MyOtherMessage varchar(50);

   SELECT @MyMessage = 'Hi, it''s that line before the RETURN';
   PRINT @MyMessage
   RETURN 100;
   SELECT @MyOtherMessage = 'Sorry, but we won''t get this far';
   PRINT @MyOtherMessage;
RETURN;
```

再次运行测试脚本，除了返回值改变了之外，会得到相同的结果。

```
Hi, it's that line before the RETURN
-----------
100
(1 行受影响)
```

12.6　错误处理

提示：

的确，可能并不需要这一节的内容。但是代码可能永远没有错误，而且永远都不会碰到问题吗？这大概只是幻想而已，现实情况并非如此。和软件工程中一样，代码总是会出错。幸运的是，可以对它进行处理。遗憾的是，可能现有的工具并不理想。然而又很幸运的是，在 SQL 领域中总有物尽其用的方法，也有许多隐藏错误处理不足之处的方法。

第 11 章在介绍 TRY/CATCH 块时讲到了一些错误处理的内容。实际上，那些只是在只需要支持 SQL Server 2005 或更高版本时处理传统错误的方法。对于 SQL Server 错误和错误处理，考虑的问题要比 TRY/CATCH 块中的多得多。下面将作详细讨论。

SQL Server 中有以下 3 种常见的错误类型。

www.TopSage.com

- 产生运行时错误并终止代码继续运行的错误。
- SQL Server 知道的、但不产生使代码停止运行的运行时错误的错误。这类错误也被称为内联(inline)错误。
- 更具逻辑性但在 SQL Server 中不太引起注意的错误。

现在，问题变得有些棘手，并且版本也变得重要，但还是要继续面对它。

提示：

在写本书的时候，大部分关于 SQL Server 2008 的书还没有出版——但我可以大胆地猜测，大多数的入门书籍不会对以前的版本过多讨论。本书也会尽量地回避这个问题，因为这会增加更多的复杂性。不过本节将涉及以前版本的内容。这是因为大多数的数据库开发人员可能还在使用 SQL Server 2005 以前版本的代码(在 TRY/CATCH 首次被引入时)。在 SQL Server 2000 和更早的版本中并没有正式的错误处理程序。

因此，这里会给出说明以前如何错误处理的一个简要介绍——因为它有助于理解可能会在以前代码中碰到的“为什么会以这种方式处理”这样的问题。如果您确定您将只会面对“SQL Server 2005 或更新版本的代码”的问题，那么可以跳过下面的内容。

旧的错误处理模型和新的错误处理模型之间有一个共同点——更高级别的运行时错误。

可能会产生导致 SQL Server 立即结束脚本的错误。这在 TRY/CATCH 引入之前就已经是事实了，而且在 TRY/CATCH 引入之后也是如此。那些会产生运行时错误的严重性错误从 SQL Server 方面来讲是有问题的。相比于 SQL Server 2005 之前的版本，TRY/CATCH 逻辑在处理某些错误方面有更多的灵活性，但是仍有可能存在存储过程不知道发生错误的情况，因为存储过程立即终止了，未注意到错误(至少存储过程没有)。令人高兴的是，当前所有的数据访问对象模型都通过了这类错误的消息，所以可能在客户应用程序中知道这些错误并对它们进行处理。

12.6.1 以前的方式

在 SQL Server 2000 及更早版本中，没有正式的错误处理程序。无法实现“如果发生错误，就略过这段代码”。而是要在代码中监视错误的条件，然后决定在检测到错误的时(很可能是在发生实际错误以后)要做的处理。

1. 处理内联错误

内联错误是指那些能让 SQL Server 继续运行，但是因为某种原因而不能成功完成指定任务的错误。例如，试着向 Person.BusinessEntityContact 表中插入一条记录，而在 BusinesEntity 表或 Person 表中没有与之对应的记录：

```
USE AdventureWorks2008;
GO

INSERT INTO Person.BusinessEntityContact
            (BusinessEntityID
            ,PersonID
            ,ContactTypeID)
   VALUES
            (0,0,1);
```


www.TopSage.com

SQL Server 不会执行该插入，因为在 BusinessEntityID 和 PersonID 上有外键约束，这些外键约束引用其他表。由于在两表中没有匹配记录，因此试图向 Person.BusinessEntityContact 表插入记录违反了外键约束，并且会被拒绝：

```
消息 547，级别 16，状态 0，第 2 行
INSERT 语句与 FOREIGN KEY 约束"FK_BusinessEntityContact_Person_PersonID"冲突。
该冲突发生于数据库"AdventureWorks2008"，表"Person.Person"，
column "BusinessEntityID"。
语句已终止。
```

注意上面的 547 错误——这是后面可以利用的。

注意：

在 SQL Server 提供的错误中只显示违反了第一个外键。这是因为 SQL Server 到达了该错误点，并且知道继续运行没有任何意义。如果修正了第一个错误，那么将检测到第二个，再次会给出错误消息。

2. 利用@@ERROR

在介绍脚本的时候就已经讨论过这个系统函数了，现在更进一步研究它。

回顾一下，@@ERROR 包含了最后执行的 T-SQL 语句的错误号。如果该值为 0，则没有发生错误。这有点类似于在第 11 章第一次讨论 TRY/CATCH 块时看到的 ERROR_ NUMBER()函数。ERROR_NUMBER()函数只在 CATCH 块中有效(并且不管在 CATCH 块中的哪个位置都有效)，而@@ERROR 根据执行的每条语句接收一个新值。

说明：

有关@@ERROR 的警告是每条新语句都会使之重置。这意味着如果想延迟分析该值，或者想多次使用它的话，则需要把该值放入其他的保存容器中——即为此而声明的一个局部变量。

使用以前的 INSERT 示例作演示：

```
USE AdventureWorks2008;
GO

DECLARE @Error int;

-- Bogus INSERT - there is no PersonID or BusinessEntityID of 0. Either of
-- these could cause the error we see when running this statement.
INSERT INTO Person.BusinessEntityContact
           (BusinessEntityID
           ,PersonID
           ,ContactTypeID)
   VALUES
           (0,0,1);

-- 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;
```


www.TopSage.com

```
-- 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
-- since our last statement (the PRINT) didn't have an error.
PRINT 'The Value of @@ERROR is ' + CONVERT(varchar, @@ERROR);
```

现在执行脚本，检查一下是如何影响@@ERROR 的：

```
消息 547，级别 16，状态 0，第 6 行
INSERT 语句与 FOREIGN KEY 约束"FK_BusinessEntityContact_Person_PersonID"冲突。
该冲突发生于数据库"AdventureWorks2008"，表 "Person.Person"，column
 'BusinessEntityID'。

语句已终止。

The Value of @Error is 547
The Value of @@ERROR is 0
```

这很快地说明了通过@@ERROR 保存值的问题。第一个错误语句实际上只是一条信息。SQL Server 已经抛出了这个错误，但是并没有停止运行代码。实际上，存储过程访问该消息的唯一一个部分就是错误号。这个驻留在@@ERROR 中的错误号只是针对下一条 T-SQL 语句——在这之后就没有该错误号了。

注意：

@Error 和@@ERROR 是两个截然不同的变量，可以分别引用它们。这不仅是由于大小写的区别(根据配置服务器的方法，区分大小写会影响变量的名称)，还因为有作用域的不同。@和@@是名称的一部分，所以前端@符号数目的不同使得这两个变量相互区别。

3. 在存储过程中使用@@ERROR

首先作个假定：如果使用@@ERROR，那么很可能不使用 TRY/CATCH 块。如果因为后向兼容问题而未作这一选择，那么此处将建议您重新考虑一下——TRY/CATCH 其实是更简洁和更通用的方法。

注意：

TRY/CATCH 将处理之前的版本中终止脚本执行的各种错误。

虽说如此，如果需要向后兼容 SQL Server 2000 或更早版本，那么 TRY/CATCH 就不适合，下面简单看一下。

看两个简短的过程。这两者都基于在脚本或之前的存储过程示例所做的操作，但我们希望看一下内联错误检查的工作方式，用于确定内联错误检查什么时候可以运行，什么时候不能运行，以及为什么不能运行(特别是内联错误检查不能运行，而 TRY/CATCH 可以运行的情况)。

首先看一下本章前面的参照完整性示例：

www.TopSage.com

```
USE AdventureWorks2008;
GO

INSERT INTO Person.BusinessEntityContact
          (BusinessEntityID
          ,PersonID
          ,ContactTypeID)
   VALUES(0,0,1);
```

您可能知道这会引发简单的547错误。这是可捕获的错误之一。可以通过一个简单的脚本捕获它，但这里使用存储过程。

```
USE AdventureWorks2008
GO

CREATE PROC spInsertValidatedBusinessEntityContact
   @BusinessEntityID int,
   @PersonID int,
   @ContactTypeID int
AS
BEGIN

   DECLARE @Error int;

   INSERT INTO Person.BusinessEntityContact
      (BusinessEntityID
      ,PersonID
      ,ContactTypeID)
   VALUES
     (@BusinessEntityID, @PersonID, @ContactTypeID);

   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
```

现在试着用适当的值执行上述代码：

```
EXEC spInsertValidatedBusinessEntityContact 1, 1, 11;
```

插入操作正确进行，没有检测到错误条件(因为没有)：

```
(1 行受影响)
New Record Inserted
```


www.TopSage.com

现在试一下下列代码：

```
EXEC spInsertValidatedBusinessEntityContact 0, 1, 11;
```

这样不仅看到实际的 SQL Server 消息，还有错误捕获中的消息(注意，没办法消除 SQL Server 消息)。

```
消息 547,级别 16,状态 0, 过程 spInsertValidatedBusinessEntityContact, 第 11 行
INSERT 语句与 FOREIGN KEY 约束"FK_BusinessEntityContact_Person_PersonID"冲突。 该冲突发生于数据库"AdventureWorks2008", 表"Person.BusinessEntity", column 'BusinessEntityID'。
语句已终止。

At least one provided parameter was not found. Correct and retry
```

正如您所见，可以在不使用 TRY/CATCH 块的情况下检测错误。

现在，通过一个示例说明为什么采用 TRY/CATCH 块会更好的原因——其中 TRY/CATCH 块工作得很好，但内联错误检查会失败。为此，需要使用“脚本”一章的 TRY/CATCH 示例，如下所示：

```
BEGIN TRY
   -- Try and create our table
   CREATE TABLE OurIFTest(
      Col1 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
```

这工作得很好。但如果试图使用内联错误检查，就会有问题：

```
CREATE TABLE OurIFTest(
   Col1 int PRIMARY KEY
   );
IF @@ERROR != 0
```


www.TopSage.com

```
    PRINT 'Problems!';
ELSE
    PRINT 'Everything went OK!';
```

运行上述代码(如果表不存在，需要运行代码两次来产生错误)，很快会发现，没有 TRY 块，SQL Server 在特定的错误处完全中止脚本运行。

```
消息 2714，级别 16，状态 6，第 2 行
数据库中已存在名为'OurIFTest'的对象。
```

可注意到，PRINT 语句没有机会执行——SQL Server 已终止运行。通过 TRY/CATCH 块，可以捕获并处理这一错误；但使用内联错误检查，尝试捕获这样的错误就会失败。

12.6.2 在错误发生前处理错误

有时，有些错误是 SQL Server 没有办法知道的，更别说进行提醒了。而有时，我们想在错误发生前预防错误。这些都需要检测并自行处理。

为了作演示，这里为 AdventrueWorks2008 中现有的名为 HumanResources.uspUpdate EmployeeHireInfo 存储过程创建一个新版本——即 HumanResources.uspUpdateEmployee HireHnfo2。在这里，要采用一些业务规则，它们在本质上是逻辑性的，但在数据库中不一定实现(或在这里，甚至可以用约束处理)。首先看一下现有的存储过程：

```
CREATE PROCEDURE HumanResources.uspUpdateEmployeeHireInfo2
   @BusinessEntityID int,
   @JobTitle nvarchar(50),
   @HireDate datetime,
   @RateChangeDate datetime,
   @Rate money,
   @PayFrequency tinyint,
   @CurrentFlag dbo.Flag
WITH EXECUTE AS CALLER
AS
BEGIN
   SET NOCOUNT ON;

   BEGIN TRY
      BEGIN TRANSACTION;

      UPDATE HumanResources.Employee
      SET JobTitle = @JobTitle,
         HireDate = @HireDate,
         CurrentFlag = @CurrentFlag
      WHERE BusinessEntityID = @BusinessEntityID;

      INSERT INTO HumanResources.EmployeePayHistory
         (BusinessEntityID,
         RateChangeDate,
         Rate,
         PayFrequency)
      VALUES (@BusinessEntityID, @RateChangeDate, @Rate, @PayFrequency);

      COMMIT TRANSACTION;
```


www.TopSage.com

```
    END TRY
    BEGIN CATCH
        -- Rollback any active or uncommittable transactions before
        -- inserting information in the ErrorLog
        IF @@TRANCOUNT > 0
        BEGIN
            ROLLBACK TRANSACTION;
        END

        EXECUTE dbo.uspLogError;
    END CATCH;
END;
```

这个存储过程中的内容是很简单的：有两条语句(一个更新现有雇员记录，另一个处理额外的历史记录)，外加一个普通的错误处理程序。要做的是向该存储过程中添加一些新代码，识别可能发生的错误，提供返回值向客户端通知更具体的错误信息。这里并不准备花时间捕获潜在的所有错误，但会捕获一些基本的以显示其工作原理(读者可自行作进一步研究)。

这个存储过程中的错误处理程序是非常普通的，并不会真正做一些特别针对这个存储过程的工作，因此首先考虑一下这个存储过程中可能发生的错误，例如：

- **其 BusinessEntityID 不存在的雇员**：这个存储过程中的 UPDATE 语句对于没有有效的 BusinessEntityID 的情况会运行良好(没有错误)。它只是会找不到匹配，而不影响任何行；这一错误在本质上是逻辑性的，SQL Server 不会认为有错。应自己检测这一错误，并在存储过程继续运行前在此捕获它(因为 EmployeePayHistory 和 Employee 之间有外键关系，当无法找到匹配 BusinessEntityID 时，SQL Server 将在 INSERT 语句上引发错误)。
- **两个更新操作在同一 RateChangeDate 影响同一 BusinessEntityID**：同样，UPDATE 语句进行这类更新时没什么问题，但 INSERT 语句会出错(EmployeePayHistory 的主键由 BusinessEntityID 和 RateChangeDate 组成)。

现在在新版存储过程中解决这两个问题。

首先，做一些基本工作。尽管 SQL Server 中没有常量的概念，本书还是将把一些变量作为常量来使用。为此，将不准备只是返回错误号，而是返回一个变量名，通过表明返回错误的性质使代码更具可读性：

```
...
...
SET NOCOUNT ON;

-- Set up 'constants" for error codes
DECLARE @BUSINESS_ENTITY_ID_NOT_FOUND int,
        @DUPLICATE_RATE_CHANGE int
SET @BUSINESS_ENTITY_ID_NOT_FOUND = -1000
SET @DUPLICATE_RATE_CHANGE = -2000


BEGIN TRY
...
...
```


www.TopSage.com

提示：

您可能会对这里对错误使用负值感到奇怪。尽管这方面没有标准可言，本书还是习惯于用正值表示信息类型的返回代码(可能有多个可能的成功结果，此处希望表明完成的何种成功操作)，而用负值表示错误。确保遵循一致性原则。该示例还故意没有使用 SQL Server 2008 提供的初始化语法(而将声明改为 DECLARE @BUSINESS_ENTITY_ID_NOT_ FOUND int = -1000)。这纯粹是为了向后兼容，因此作相应调整。

接下来，可以测试对 HumanResources.Employee 的 UPDATE 操作影响了多少行，并利用测试结果检测 BusinessEntityID Not Found 错误：

```
...
...
UPDATE HumanResources.Employee
SET JobTitle = @JobTitle,
   HireDate = @HireDate,
   CurrentFlag = @CurrentFlag
WHERE BusinessEntityID = @BusinessEntityID;

IF @@ROWCOUNT > 0
-- things happened as expected
   INSERT INTO HumanResources.EmployeePayHistory
      (BusinessEntityID,
      RateChangeDate,
      Rate,
      PayFrequency)
   VALUES (@BusinessEntityID, @RateChangeDate, @Rate, @PayFrequency);
ELSE
-- ruh roh, the update didn't happen, so skip the insert,
-- set the return value and exit
BEGIN
   PRINT 'BusinessEntityID Not Found';
   ROLLBACK TRAN;
   RETURN @BUSINESS_ENTITY_ID_NOT_FOUND;
END
 ...
 ...
```

注意：

从 UPDATE 语句中删除 HireDate 列。

正如前面所讨论的，RETURN 将立即退出该存储过程，并给出所提供的返回值(这里是-1000，与变量相匹配的值)。客户编应用程序现在可测试该返回值并将返回值与已知的可能错误列表匹配。

现在来看第二个潜在的错误。有多种方法处理该错误。可以预查询 EmployeePayHistory 表，查看是否已有一个匹配行，然后完全避免 INSERT 操作。或者，可以允许 SQL Server 检测该错误，采用错误处理程序处理这种可能性。在这里将采用后者。通常更好的做法都是依据规则，处理异常。本书将这一特别错误视为会经常发生的，因此下面将大胆推测它不会发生，如果发生就处理它。了解了这一点后，只需要对错误处理程序作些更改。

www.TopSage.com

```
...
...
BEGIN CATCH
   -- Rollback any active or uncommittable transactions before
   -- inserting information in the ErrorLog
   IF @@TRANCOUNT > 0
   BEGIN
      ROLLBACK TRANSACTION;
   END

   EXECUTE dbo.uspLogError;

   IF ERROR_NUMBER() = 2627 -- Primary Key violation
   BEGIN
      PRINT 'Duplicate Rate Change Found';
      RETURN @DUPLICATE_RATE_CHANGE;
   END
END CATCH;
...
...
```

作了这些更改后，看一下这个新的存储过程。尽管这是个新存储过程，还是突出显示了作了更改的代码行：

```
CREATE PROCEDURE HumanResources.uspEmployeeHireInfo2</codeScreen>
   @BusinessEntityID [int],
   @JobTitle [nvarchar](50),
   @HireDate [datetime],
   @RateChangeDate [datetime],
   @Rate [money],
   @PayFrequency [tinyint],
   @CurrentFlag [dbo].[Flag]
WITH EXECUTE AS CALLER
AS
BEGIN
   SET NOCOUNT ON;
   -- Set up "constants" for error codes
   DECLARE @BUSINESS_ENTITY_ID_NOT_FOUND int = -1000,
           @DUPLICATE_RATE_CHANGE int = -2000;

   BEGIN TRY
      BEGIN TRANSACTION;

      UPDATE HumanResources.Employee
      SET JobTitle = @JobTitle,
         HireDate = @HireDate,
         CurrentFlag = @CurrentFlag
      WHERE BusinessEntityID = @BusinessEntityID;

      IF @@ROWCOUNT > 0
      -- things happened as expected
         INSERT INTO HumanResources.EmployeePayHistory
            (BusinessEntityID,
            RateChangeDate,
```


www.TopSage.com

```
            Rate,
            PayFrequency)
        VALUES
            (@BusinessEntityID,
            @RateChangeDate,
            @Rate,
            @PayFrequency);
    ELSE
    -- ruh roh, the update didn't happen, so skip the insert,
    -- set the return value and exit
    BEGIN
        PRINT 'BusinessEntityID Not Found';
        ROLLBACK TRAN;
        RETURN @BUSINESS_ENTITY_ID_NOT_FOUND;
    END

        COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    -- Rollback any active or uncommittable transactions before
    -- inserting information in the ErrorLog
    IF @@TRANCOUNT > 0
    BEGIN
        ROLLBACK TRANSACTION;
    END

    EXECUTE dbo.uspLogError;

    IF ERROR_NUMBER() = 2627 -- Primary Key violation
    BEGIN
        PRINT 'Duplicate Rate Change Found';
        RETURN @DUPLICATE_RATE_CHANGE;
    END
  END CATCH;
END;
```

继续运行下列代码：

```
DECLARE @Return int;

EXEC @Return = HumanResources.uspEmployeeHireInfo2
   @BusinessEntityID = 1,
   @JobTitle = 'His New Title',
   @HireDate = '1996-07-01',
   @RateChangeDate = '2008-07-31',
   @Rate = 15,
   @PayFrequency = 1,
   @CurrentFlag = 1;

SELECT @Return;
```

一切工作正常，但第二次运行上述代码，会得到一些不同的结果：

```
Duplicate Rate Change Found
-----------
```


www.TopSage.com

```
-2000

(1 行受影响)
```

刚才尝试插入有着同样的支付记录的另一行，但 SQL Server 拒绝了。使用 PRINT 提供一些信息性输出，因为有可能不是通过“查询”窗口来执行语句；并且还输出了一个特定的返回值，客户端可将该返回值与资源列表中的值相匹配。

现在，尝试同样的基本测试，但使用一个无效的 BusinessEntityID：

```
DECLARE @Return int;

EXEC @Return = HumanResources.uspEmployeeHireInfo2
   @BusinessEntityID = 99999,
   @JobTitle = 'My Invalid Employee',
   @HireDate = '2008-07-31',
   @RateChangeDate = '2008-07-31',
   @Rate = 15,
   @PayFrequency = 1,
   @CurrentFlag = 1;

SELECT @Return;
```

结果得到一个类似的错误消息和返回代码，但对于检测到的特定错误，会略有不同：

```
BusinessEntityID Not Found

-----------
-1000

(1 行受影响)
```

注意这个不是 SQL Server 的错误——就 SQL Server 而言，没有任何问题。这样的好处是，如果使用客户端应用程序(可以使用 C#、VB.NET、C++或其他语言编写)，则和重复插入支付历史记录项一样，可以通过一个已知常量来跟踪 - 1000 错误并向终端用户发送一个特定的消息。

12.6.3 手动引发错误

有时会出现 SQL Server 不能识别的错误，但是希望 SQL Server 能识别这些错误。例如，在前一个示例中并不想返回 - 1000，而是想在客户端创建一个运行时错误，而客户端可以使用它来调用错误处理程序并采取相应的动作。为此，可使用 T-SQL 中的 RAISERROR 命令。其语法很简单：

```
RAISERROR (<message ID | message string | variable>, <severity>, <state>
[, <argument>
[,<...n>]] )
[WITH option[,...n]]
```

下面介绍 RAISERROR 命令的具体用法。

www.TopSage.com

1. 消息 ID/消息字符串

消息 ID(message ID)或消息字符串(message string)决定了向客户端发送的消息。

使用消息 ID 可以手动引发带有指定的 ID 以及和该 ID 相关消息的错误，错误消息可以在 master 数据库中的 sys.messages 系统视图中找到。

提示：

*如果想要了解 SQL Server 中预定义的消息，那么可以执行 SELECT * FROM master.sys.messages。这包括了使用 sp_addmessage 存储过程或通过 SQL Server Management Studio 手动添加到系统的任何消息。*

可以以特殊文本的形式提供消息字符串，而不用在系统中创建一个更持久的消息：

```
RAISERROR ('Hi there, I''m an error', 1, 1);
```

这会引发一个简单的错误消息：

```
Hi there, I'm an error
消息 50000，级别 1，状态 1
```

注意，默认指定的消息号是 50000，尽管并没有提供该默认值。这是针对任何特别错误的默认错误值。可以使用 WITH SETERROR 选项来重写该默认值。

2. 严重性

在第 11 章介绍 TRY/CATCH 块时提到过此内容。对于熟悉 Windows 服务器的用户来说，对严重性(severity)还是相当熟悉的。严重性表明了错误的严重程度。不过对于 SQL Server 来就，严重性代码的意义有些古怪。它们包括了信息性的(严重性 1～18)、系统级的(19～25)，甚至是灾难性的(20～25)。如果引发的错误的严重性是 19 或更高的话(系统级的)，那么必须要指定 WITH LOG 选项。20 以及更高级别的错误严重性会自动终止用户的连接。

现在回到刚才提到的古怪之处。SQL Server 对其行为划分的门类要多于 Windows，甚至比联机丛书中提到的都要多。其错误被分成 6 类，如表 12-1 所示。

表 12-1

1～10	纯信息错误，但在消息信息中会返回特定的错误代码
11~16	如果没有设置 TRY/CATCH 块，那么这些错误会终止过程执行并在客户端引发错误。状态显示为设置的任何值。如果定义了 TRY/CATCH 块，那么将调用错程处理程序，而不是在客户端引发错误
17	通常，只有 SQL Server 会使用这个错误严重性级别。基本上，它表明 SQL Server 用完了资源(例如 tempdb 已满)而且不能完成请求。同样，TRY/CATCH 块将在客户端之前获取该错误
18～19	这些都是严重的错误，而且暗示系统管理员要注意底层原因。对于错误严重性级别 19 来说，需要使用 WITH LOG 选项，事件将在 Windows Event Log 中显示。这是可用 TRY/CATCH 块捕获的最后一个级别的错误——超过这一级别的错误将直接在客户端引发
20～25	遇到这些错误就没办法了。基本上，它们是致命的错误，会终止连接。和错误级别 19 一样，必须使用 WITH LOG 选项，如果可应用的话，消息将会出现在 Event Log 中

www.TopSage.com

3. 状态

状态(state)是个特殊的值。它会识别在代码中的多个地方发生的相同错误。这就可以发送位置标记表明发生错误的地方。

状态值可以在 1~127 之间。如果通过 Microsoft 技术支持来对错误进行排查，那么他们显然拥有一些只为他们所了解的知识，而不能与我们分享状态值所表示的含义。如果向 Microsoft 寻求技术支持，则他们很可能会询问并利用这个状态信息。

4. 错误参数

一些预定义的错误会接受参数。这就可能通过改变错误特性来使错误更具动态性。也可以格式化错误消息来接受参数。

如果想在静态错误消息中使用动态信息，就需要格式化消息的固定部分，为消息的参数部分留下空间。可以通过使用占位符来完成。如果您是一位 C 或 C++程序员，那么您将识别这些参数占位符——它们和 printf 命令参数是相似的。如果您不是 C 程序员，那么可能会觉得这有点怪异。所有的占位符都以%符号开始，然后是传递的信息类型的编码，如表 12-2 所示。

表 12-2

占位符类型指示符	类 型 值
D	有符号的整数；联机丛书中指出 it 也是个可接受的选项，但本书在使用它时碰到了一些问题
O	无符号的八进制数
P	指针
S	字符串
U	无符号的整数
X 或 x	无符号的十六进制数

另外，还有一些使用一些额外的标记和宽度信息来作为占位符的前缀的选项，如表 12-3 所示。

表 12-3

标 记	作 用
—(短划线或减号)	左对齐；只在提供固定宽度时有所不同
+(加号)	如果参数是有符号的数值类型，则指示参数为正或为负
0	告诉 SQL Server 在数值类型值的左边填充 0，直到满足宽度选项中指定的宽度
#(磅符号)	只应用于八进制和十六进制数。告诉 SQL Server 根据是八进制还是十六进制来使用适当的前缀(0 或 0x)
' '	如果数值为正，则在该值的左边填充空格

www.TopSage.com

最后一点也很重要，可以设定参数的宽度、精度和数据类型的长/短。

- **宽度(Width)**：通过提供整数值来设定想要为参数值所保留的空间。也可以指定“*”，在这种情况下，SQL Server 会自动根据设定的精度值来决定宽度。
- **精度(Precision)**：确定了数值型数据输出的最大数据位数。
- **长/短(long/short)**：当参数类型为整数、八进制数或十六进制数时，使用 h(short)或 l(long)来设定。

在下面这个简单示例进行应用：

```
RAISERROR ('This is a sample parameterized %s, along with a zero
padding and a sign%+010d',1,1, 'string', 12121);
```

如果执行这个代码，就会得到和引号中稍有不同的信息：

```
This is a sample parameterized string, along with a zero
padding and a sign+000012121
消息 50000，级别 1，状态 1
```

提供的额外信息是按顺序插入到占位符中的，最终的值根据指定的形式重新格式化。

5. WITH<option>

目前，在引发错误时可混合搭配以下 3 个选项。

- LOG
- SETERROR
- NOWAIT

1) WITH LOG

这告诉 SQL Server 在 SQL Server 错误日志和 Windows 应用程序日志中记录错误。对于严重性级别大于或等于 19 的，这个选项是必需的。

2) WITH SETERROR

默认情况下，RAISERROR 命令不会将@@ERROR 设置为所生成的错误值—— 而@@ERROR 反映了实际 RAISERROR 命令是成功或失败的。SETERROR 重写了这个值并且把@@ERROR 的值设置为特定的错误 ID。

3) WITH NOWAIT

这将立即向客户端通知错误。

12.6.4 添加自定义的错误消息

可以利用特殊的系统存储过程向系统添加消息。这个存储过程称为 sp_addmessage，语法如下所示：

```
sp_addmessage [@msgnum =] <msg id>,
[@severity =] <severity>,
[@msgtext =] <'msg'>
[, [@lang =] <'language'>]
```


www.TopSage.com

```
[, [@with_log =] [TRUE|FALSE]]
[, [@replace =] 'replace']
```

除了其他的语言和替换参数，以及 WITH LOG 选项的一点区别外，该存储过程所有参数的含义与 RAISERROR 中的是相似的。

1. @lang

这个参数指定了消息应用的语言。可以为 syslanguages 中支持的任何语言指定消息的一个单独版本。

2. @with_log

其工作方式与 RAISERROR 中的一样，如果设为 TRUE，当出现错误的时候，则会向 SQL Server 错误日志和 NT 应用程序日志记录错误消息(只有运行在 NT 环境下才会实现后一操作)。这里唯一特别之处就是通过把该参数置为 TRUE 而不是使用 WITH LOG 选项来把消息写入日志。

提示：

在联机丛书中要小心这个问题。根据理解的不同，您很可能在应该将@with_log 设置为 TRUE 的时候，而认为应把它设置为字符串常量 'WITH_LOG'。可能更让人迷惑的是 REPLACE 选项，因为必须把它设置为字符串常量而不是 TRUE。

3. @replace

如果是编辑现有的消息而非创建一个新消息，那么必须把@replace 参数设置为 'REPLACE'。倘若不这样做，那么如果消息已经存在，就会得到一个错误。

注意：

创建一个额外消息的列表供应用程序使用可以大大提高复用性，但是更为重要的是，可以极大地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自定义错误代码的常量列表的情况。那么就可以很方便地建立一个常量文件(例如资源或包含库)，该文件有一个适当错误的列表；甚至可以创建一个能对部分或所有错误进行通用处理的包含库。简而言之，如果要在同一个环境中创建多个 SQL Server 应用程序，可考虑使用一个对于所有应用程序公用的错误列表。不过，要记住，许多系统管理员并不喜欢特定应用程序的更改影响到 master 数据库(写自定义错误消息的地方)，因此如果不能完全控制自己的服务器，在编写依赖于自定义错误的代码之前，要确保这些自定义错误消息在服务器上是允许的。

4. 使用 sp_addmessage

如前所述，sp_addmessage 创建消息的方式和使用 RAISERROR 创建特殊消息的方式是一样的。

例如，假定 AdventrueWorks2008 正实施一个规则，即如果订单超出 7 天，就不能输入。针对这一规则，可以添加自定义的消息来告诉用户关于订单日期的问题：

```
sp_addmessage
```


www.TopSage.com

```
 @msgnum = 60000,
 @severity = 10,
 @lang='English'
 @msgtext = '%s is not a valid Order date.
Order date must be within 7 days of current date.';
```

执行该存储过程，它确认添加的新消息：

```
命令已成功完成。
```

注意：

当运行 sp_addmessage 时，无论使用何种数据库，实际的消息都被添加到主数据库，任何时候都可通过 sys.messages 系统视图来查看。这意味着，如果把数据库迁移到一台新的服务器上，则需要在新的服务器上重新添加消息(而以前的消息仍旧在旧服务器的 master 数据库中)。同样地，强烈建议您把所有自定义的消息存储在一个脚本中，这样可以很方便地将消息添加到新的系统中。

5. 删除已有的自定义消息

要删除自定义消息，可使用下列语句：

```
sp_dropmessage <message number >
```

12.7　存储过程的优点

在学习创建存储过程的方法后，我们可能会问“为什么使用它们”这样的问题。其中的一些原因是很基本的，而如果您不熟悉 RDBMS，则可能不会立刻想到其原因所在。存储过程的主要优点包括以下几个方面。

- 使得需要过程式动作的进程可调用
- 安全性
- 性能

12.7.1　创建可调用的进程

前面已经提过，存储过程是存储在数据库中的脚本。因为它是数据库对象，好处是可以调用它——不需要在执行前从文件中手动地加载存储过程。

存储过程可以调用其他存储过程(称为嵌套)。对于 SQL Server 2008 来说，可以嵌套 32 层的深度。这就可以像在传统过程式语言中使用子例程一样，重用单独的存储过程。在一个存储过程中调用另一个存储过程的语法和在脚本中调用存储过程是一样的。

12.7.2　为了安全性使用存储过程

很多人并没有意识到要充分使用存储过程，使其作为实现安全性的工具。和视图类似，可以创建一个返回记录集的存储过程而不用赋予用户访问底层数据表的权限。赋予某人执行一个存储过程的权限意味着他们可以在该存储过程中执行任何动作，不过要假设动作是在存储过程

www.TopSage.com

的上下文中执行的。也就是说，如果赋予用户执行返回 Customers 表中所有记录的存储过程的权限，而不是访问实际 Customers 表，当用户执行该存储过程时(而尝试直接访问表是行不通的)，那么仍然可以从 Customers 表中得到数据。

真正方便之处在于可以让用户通过存储过程修改数据，但是对于底层表只能允许用户读访问。如果采用存储过程(可能会强制一些业务规则)，那么可以修改表中的数据。可以通过使用 Excel、Access 或者其他任何的报表来直接与 SQL Server 相关联，建立各自的自定义报告，而没有“意外”修改数据的危险。

注意：

通过 Access 或 Excel 使用户与生产数据库相关联是一件对系统非常有用的但又很愚蠢的事情。当您对用户授权的同时，从用户可以使用的资源以及他们可以执行的长期执行的查询(自然地，他们并不会注意到这对系统的破坏)来看，实际也是在自找麻烦。

如果您确实想让用户直接访问，那么考虑一下使用复制或备份以及还原创建一个完全单独的数据库副本以供他们使用。这有助于保证不会出现记录的锁定、使系统停顿的查询，以及一大堆其他问题。

12.7.3 存储过程和性能

一般来说，存储过程有助于系统性能的提高。但是请记住，和生活中大多数事情一样，这没有绝对保证——事实上，如果设计的存储过程缺乏智能，那么它会使在其创建的进程变得非常缓慢。

性能从何而来？当创建一个存储过程时，其进程就像图 12-1 一样运行。

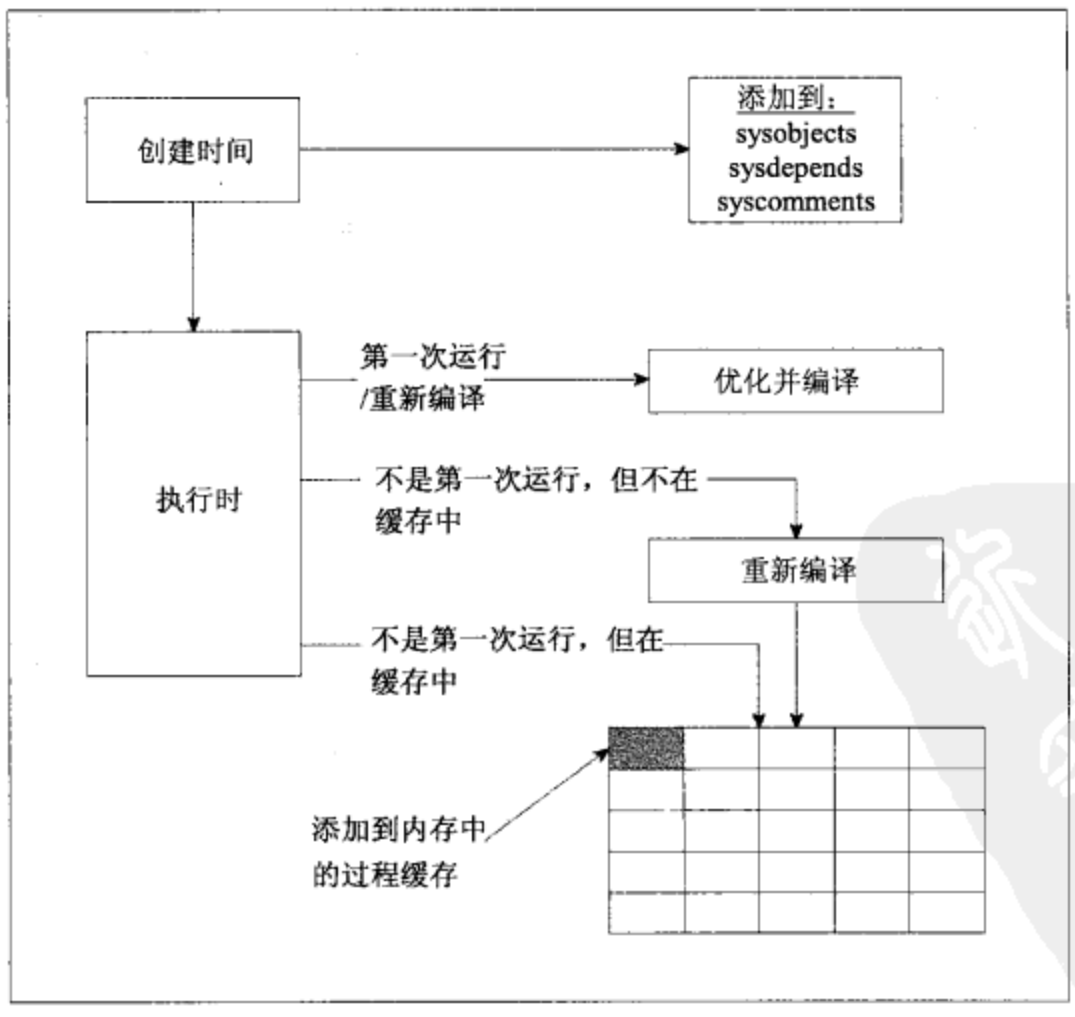


图 12-1

www.TopSage.com

首先运行 CREATE PROC 过程。这会解析查询以确保会实际运行这些代码。它与直接运行脚本的区别在于 CREATE PROC 命令可以利用所谓的延迟名称解析。延迟名称解析可以忽略一些对象还不存在的事实。这样就可以稍后创建这些对象。

在创建了存储过程后，它将等待第一次执行。在那时，存储过程被优化，而查询计划被编译并且缓存到系统上。后续几次运行该存储过程时，除非通过使用 WITH RECOMPILE 选项指定，否则都会使用缓存的查询计划而不是创建一个新的查询计划(也有存储过程被重新编译的情况，不过这超出了本书范围)。这意味着每次使用该存储过程时，存储过程都会跳过很多优化和编译工作。节省的确切时间取决于批处理的复杂性、批处理中表的大小，以及每个表上索引的数量。通常，节省的时间不是很多——对于大多数场景来说可能是 1 秒或更少——但通过百分比可以计算出此区别(1 秒比 2 秒快了 100%)。当需要进行多次调用时或针对循环的情况，这一区别会变得更明显。

存储过程的不利方面

对于存储过程的不利之处要认识到的最重要的一点是，除非手动地干预(使用 WITH RECOMPILE 选项)，否则只会在第一次运行存储过程的时候，或者当查询所涉及的表更新了统计信息时，才对存储过程进行优化。

这种“一次优化，多次使用”的策略节省了存储过程的时间，但是该策略也是一把双刃剑。如果查询是动态的(即是在使用 EXEC 命令时建立的)，那么只会在第一次运行时对存储过程进行优化，但是会发现以后再也不这样了——简而言之，可能会使用错误的计划。

不只是存储过程中的动态查询会导致这样的场景。想象一下 Web 页面允许混合搭配几个搜索条件的情况。例如，如果想要向 AdventureWorks2008 数据库添加一个存储过程，该存储过程支持一个 Web 页面允许用户根据以下内容来搜索订单。

- Customer number
- Order ID
- Product ID
- Order date

用户可以提供任何混合信息，通过提供的每一条新信息缩小搜索范围，理论上也更加快速。

针对这个问题采用的方法是采用多个查询，并且根据用户提供的信息来选择正确的查询。第一次执行存储过程时，存储过程会运行一些 IF…ELSE 语句并且选择正确的查询来运行。但是，它只对于这次运行存储过程来说是正确的(其他时候则不一定)。在存储过程第一次选择一个不同的查询之后的任何时间，它仍会使用存储过程第一次运行时的查询计划。简而言之，查询性能会受影响。

使用 WITH RECOMPILE 选项

可以利用存储过程提供的安全性和代码封装方面的好处，但还是忽略了预编译代码方面的影响。可以回避未使用正确的查询计划的问题，因为可以确保为特定一次运行创建新的计划。方法就是使用 WITH RECOMPILE 选项。使用该选项的方式有两种。

首先，可以在运行时包含 WITH RECOMPILE。只要在执行脚本中包含 WITH RECOMPILE：

```
EXEC spMySproc '1/1/2004'
```


www.TopSage.com

```
WITH RECOMPILE
```

这告诉 SQL Server 抛弃已有的执行计划并且创建一个新的计划——但只是这一次。也就是说，只是这次使用 WITH RECOMPILE 选项来执行存储过程。

也可以通过在存储过程中包含 WITH RECOMPILE 选项来使之变得更持久。如果使用这种方式，则在 CREATE PROC 或 ALTER PROC 语句中的 AS 语句前添加 WITH RECOMPILE 选项即可。

如果通过该选项创建存储过程，那么无论在运行时选择了其他什么选项，每次运行存储过程时都会重新编译它。

12.8 扩展存储过程(XP)

SQL Server 中.NET 的支持大大改变了扩展存储过程所占的比重。过去，扩展存储过程是核心代码中所必不可少的——也就是当 SQL Server 的基本 T-SQL 语法和其他功能无法提供所需的内容的时候，扩展存储过程就可帮助解决问题。

随着.NET 可以处理像 O/S 文件访问和其他外部通信或一些复杂的公式之类的事情，XP 被淘汰的日子不远了。不过，由于下列原因，XP 还是占有一席之地。

- 由于关键性能的原因，因此希望外部代码真正地在 SQL Server 进程中运行(这在.NET 时代实际是一种极端的做法)
- 出于安全的考虑，管理员不允许执行.NET 代码。

在本书中，这里只想说明 SQL Server 允许外部编写的代码作为.DLL 在 SQL Server 进程中运行。XP 是使用 C 或 C++创建的。

12.9 递归简介

编程中并不经常用到递归。然而，当需要用到它时，您会发现使用递归是再好不过的方法。因此，还是简单回顾一下递归的概念。

简单地说，递归是指一块代码调用自身的情况。其危险也是非常明显的——如果它调用了自身一次，那么如何防止它反复地调用它本身？这完全取决于您自己。也就是说，在代码会递归调用的情况下，您要提供递归检查确保在适当的时候跳出。

出于演示的目的，此处使用经典的递归示例，它几乎出现在所有有关递归讨论的教材中。它是一个几乎能为任何人所理解的示例，现在来看一下该示例。

这个经典示例是什么？是阶乘(factorial)。对于学过数学(或递归)的人来说，可知阶乘是指将一个数与比之小 1 的数相乘，再乘以下一个再小 1 的数，这样相乘下去，直至乘以 1 所得到的结果。例如，5 的阶乘是 120——也就是 5×4×3×2×1。

看一下这个递归的存储过程的实现：

```
CREATE PROC spFactorial
@ValueIn int,
@ValueOut int OUTPUT
```


www.TopSage.com

```
AS
DECLARE @InWorking int;
DECLARE @OutWorking int;
IF @ValueIn != 1
BEGIN
   SELECT @InWorking = @ValueIn - 1;

   EXEC spFactorial @InWorking, @OutWorking OUTPUT;

   SELECT @ValueOut = @ValueIn * @OutWorking;
END
ELSE
BEGIN
   SELECT @ValueOut = 1;
END
RETURN;
GO
```

此处所做的是接受一个值(想计算其阶乘的值)并输出一个值(计算出的阶乘值)。令人惊讶的是，这个存储过程并不是一步完成计算阶乘所需的工作。相反，它取一个数的阶乘值并且再次调用它本身。第二次调用第一次的运行结果。可以一直这样下去，直到到达 32 层的递归限制。一旦 SQL Server 进行了 32 次的递归，则它会引发错误并停止处理。

注意：

对.NET 程序集的任何调用会作为额外的一层计入递归计数，而在该程序集中的任何行为都不会计入。尽管 SQL Server 中的.NET 功能超出了本书的范围，仍要记住.NET 是解决嵌套层数问题的方法。

使用一个脚本来测试这个递归的存储过程：

```
DECLARE @WorkingOut int;
DECLARE @WorkingIn int;
SELECT @WorkingIn = 5;
EXEC spFactorial @WorkingIn, @WorkingOut OUTPUT;

PRINT CAST(@WorkingIn AS varchar) +' factorial is ' + CAST(@WorkingOut AS varchar);
```

得到了预期的结果 120：

```
5 factorial is 120
```

可以尝试为@WorkingIn 提供不同的值，除了下面两个重大的问题，一切都运作正常。

- 对于 int(甚至是 bigint)数据类型，当阶乘的值过大时会发生算术溢出
- 32 层的递归限制

可以输入一个大数来测试一下算术溢出——对于这个示例来说，任何比 13 大的数都可以。

测试 32 层的递归限制需要对存储过程稍做修改。这次求的是三角形数。它和阶乘类似，只是这里使用的是加法而不是乘法。因此，5 的三角形数为 15(5+4+3+2+1)。此处创建一个新的存储过程来进行测试——除了一些细小的变化外，三角形数和阶乘的存储过程是一样的。

www.TopSage.com

```
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
```

正如所见，没有作多少更改。同样地，只需要对存储过程的调用和测试脚本的 PRINT 文本作如下改变：

```
DECLARE @WorkingOut int;
DECLARE @WorkingIn int;
SELECT @WorkingIn = 5;

EXEC spTriangular @WorkingIn, @WorkingOut OUTPUT;

PRINT CAST(@WorkingIn AS varchar) + ' Triangular is ' + CAST(@WorkingOut
AS varchar);
```

如果采用@ValueIn 值 5 来运行该存储过程，可得到期望的结果 15：

```
5 Triangular is 15
```

可是，如果使用大于 32 的@ValueIn 值，就会得到以下的错误：

```
消息 217，级别 16，状态 1，过程 spTriangular，第 12 行
超出了存储过程，函数，触发器或视图的最大嵌套层数(最大层数为 32)。
```

也有一些解决此问题的方案。但是，除非可以用某种方法将递归调用进行分段(运行到 32 层的深度，接着从调用栈返回，然后再次往下运行)，否则是没办法的。只要记住，递归层次太深的函数可以用更标准的循环结构来重写，循环结构没有任何硬性的限制。所以，在必须使用递归之前先看一下是否能使用循环。

12.10 调试

以前(SQL Server 2000 时)，Management Studio 有真正的实时调试工具。这些调试工具有些

www.TopSage.com

笨拙，也就是它们只能调试存储过程(没有方法只调试脚本，而调试触发器要求创建一个激活触发器的存储过程)，但是通过一些替代方案，也获得了一个寻找良久的调试器。SQL Server 2005 出现后，从 Management Studio 中删除了所有调试功能(把调试功能放到了产品中，但要获得调试功能，必须使用作为 Business Intelligence Development Studio 一部分的 Visual Studio 安装程序)。不过，高兴的是，调试工具仍在 Management Studio 中，甚至比以前更好了。

12.10.1　启动调试器

和以前版本不一样，SQL Server 2008 中的调试器很容易找到。使用调试器的方式与 VB 或 C#中是一样的——就此而言，可能像大多数现代调试器。只需选择“调试”菜单(当“查询”窗口活动时可用)。然后从选项中选择启动方式：Start Debugging(Alt+F5)或 Step Into(F11)。

现在做一些准备看看调试器在标准脚本和存储过程中如何工作。为此，采用 12.9 节使用的脚本(来测试本章前面创建的 spTriangular 存储过程)。脚本如下所示：

```
DECLARE @WorkingOut int;
DECLARE @WorkingIn int = 5;

EXEC spTriangular @WorkingIn, @WorkingOut OUTPUT;

PRINT CAST(@WorkingIn AS varchar) + ' Triangular is '
   + CAST(@WorkingOut AS varchar);`
```

当该脚本出现在活动查询窗口中时，通过“逐语句”选项开始调试(从“调试”菜单中选择该选项或是按 F11 键)。

12.10.2　调试器的组成

当首次弹出“调试”窗口时，需要注意以下问题。

- 左面的黄色箭头(如图 12-2 所示)指示了当前执行行——如果选择“运行”或是开始单步执行代码，那么这就是下一行将要执行的代码。
- 顶部(如图 12-3 所示)有一些图标来指示不同的选项，包括：
  - **“继续”**：这将运行至存储过程的末尾或下一断点(包括观察条件)。
  - **“逐语句”**：这将运行下一行代码并且在运行接下来的代码行前停止，而不管代码位于哪个过程或函数中。如果执行的当前代码行调用一个存储过程或函数，那么“逐语句”选项会去调用该存储过程或函数，把它添加到调用堆栈中，使本地窗口显示新嵌套的存储过程而不是父存储过程，并且在嵌套的存储过程的第一行代码处停止。
  - **“逐过程”**：这会执行转到调用堆栈中同一层的下一条语句必须的每一行代码。如果没有调用另外一个存储过程或 UDF，那么这个命令和“逐语句”选项一样。如果调用了另一个存储过程或 UDF，那么“逐过程”选项会转到紧接着那个存储过程或 UDF 返回它的值的位置的语句。
  - **“跳出”**：这会执行到调用堆栈中下一个最高点为止的每一行代码。也就是说，会一直运行下去，直到到达了与当前所处的代码调用层次相同的那一层次。

www.TopSage.com

- **“停止调试”**：顾名思义，它的功能是立即停止执行。但是调试窗口仍然是打开的。
- **“切换断点”和“移除所有断点”**：可以通过单击代码窗口的左边空白区域来设置断点。设置断点是用来告诉 SQL Server 当在调试模式下运行代码时在此处停止。如果对于不想处理每一行代码大型存储过程或函数，这就很有用——只是希望它运行到某一点并且每次到达该处时停止。

```
DECLARE @WorkingOut int
DECLARE @WorkingIn int = 5

EXEC spTriangular @WorkingIn, @WorkingOut OUTPUT

PRINT CAST(@WorkingIn AS varchar) + ' Triangular is '
    + CAST(@WorkingOut AS varchar)
```

图 12-2

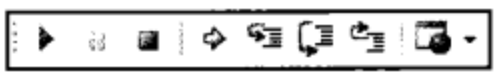


图 12-3

另外，还有一个选项可以打开“断点”窗口，其中列出了当前设置的所有断点(同样在较大块代码中有用)。还有一些“状态”窗口；下面将介绍其中比较重要的几个。

1. “局部变量”窗口

就像本书一开始所说的那样，假设您在过程语言方面已经有了一些的经验。因此，“局部变量”窗口(如图 12-4 所示，它与图 12-2 所示的当前状态相匹配)对您来说也不会陌生。简单地说，它显示了当前作用域下所有变量的当前值。如果单步执行嵌套的存储过程并再次停止，“局部变量”窗口的变量列表会随之改变(值也会改变)。记住——这些只是关于下一条将要运行语句的作用域内的那些变量。

在图 12-4 中，刚开始运行这个存储过程，因此已设置了@ValueIn 参数的值，但其他变量和参数还未设置，因此实际上为 null。

对于每个变量或参数，都提供了以下 3 部分的信息。

- 名称
- 当前值
- 数据类型

不过，“局部变量”窗口最好的一个功能是可以编辑每个变量的值。这意味着可以很方便地动态改变变量的值测试存储过程的一些行为。

2. “监视”窗口

在这里，可以设置要跟踪的变量，而不管当前位于调用堆栈的哪个位置。可以手动输入要观察的变量名，或是可以从代码中选择该变量，右击并选择“添加监视”命令。在图 12-5 中，对@ValueOut 进行观察，但由于在代码中未处理该变量，因此可看到未设置任何值。

www.TopSage.com

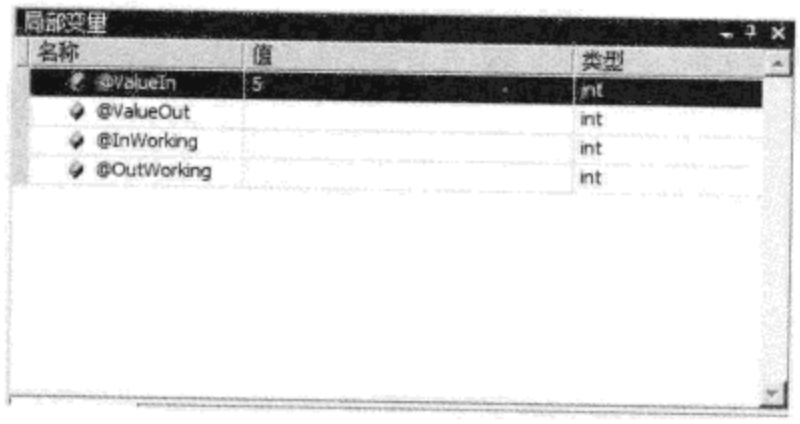


图　12-4

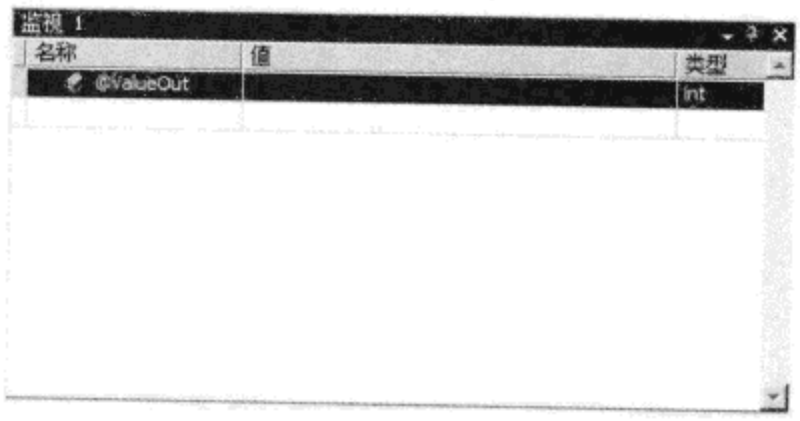


图　12-5

3. “调试堆栈”窗口

“调试堆栈”窗口提供了在当前运行的进程中有效的所有存储过程和函数的列表。其方便之处就是当在嵌套情况下运行时可了解当前嵌套深度，而且可以切换嵌套层数以验证每一层中的当前变量值。

在图 12-6 中，已经单步执行 spTriangular 存储过程的代码，因此可看到它正处理工作值 3。同样，可只观察“局部变量”窗口中的@ValueIn 变量并查看它在单步执行时如何改变。当单步执行时，调用堆栈中有若干 spTriangular 实例运行(一个针对值 5，一个针对值 4，现在针对值 3)，并且指出了当前作用域中下一条语句是什么。

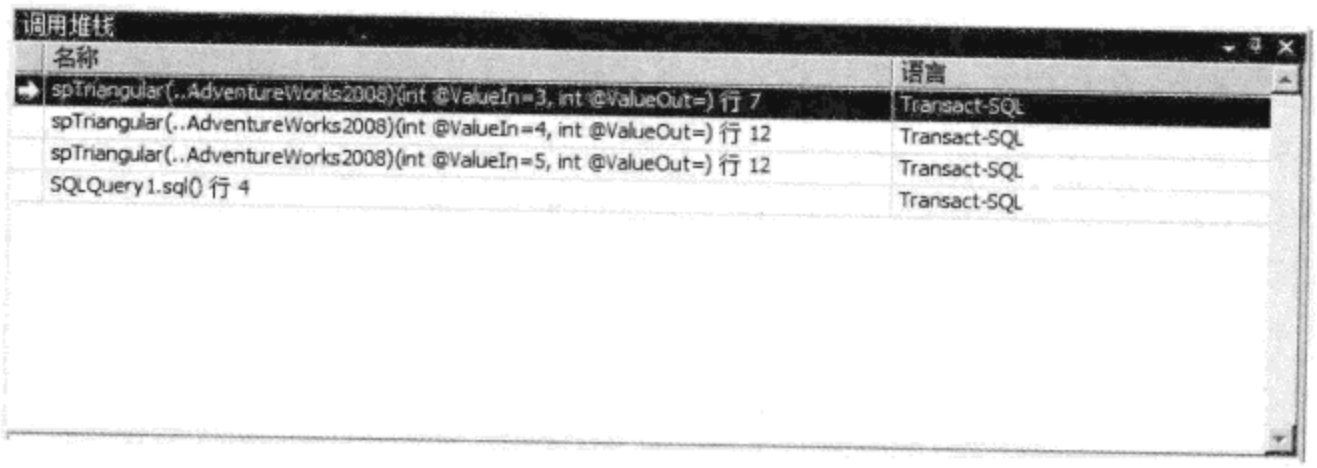


图　12-6

4. “输出”窗口

“输出”窗口是 SQL Server 打印输出的地方。这包括了结果集以及存储过程完成时的返回值，还包括了调试过程的调试信息。图 12-7 显示了调试运行过程中间的一些示例输出。

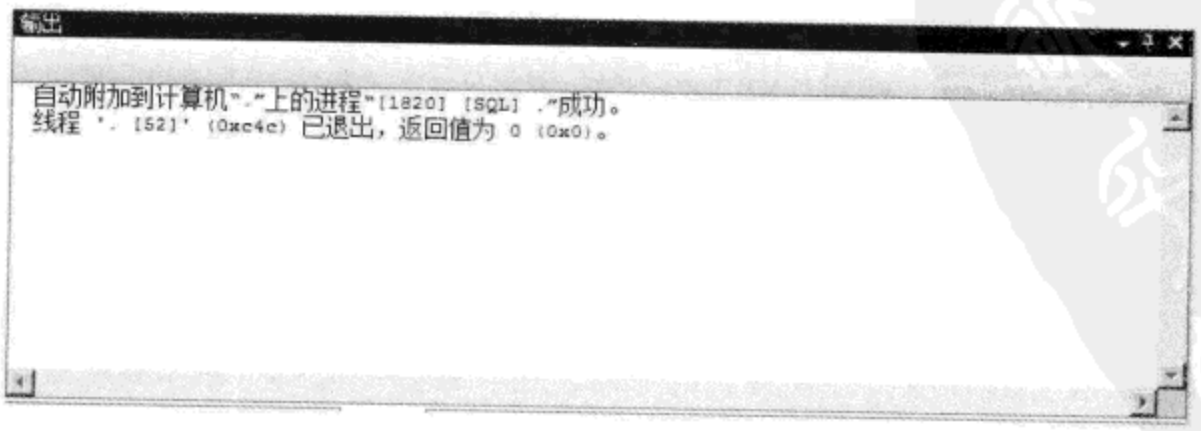


图　12-7

www.TopSage.com

5. “命令”窗口

在 SQL Server 2008 中，“命令”窗口的用处可能要超出其平常所用范围。简言之，它允许用某种命令行模式访问调试器命令和其他对象。不过，这还是加密的，在编写本书时，也没有进行归档。可采用的命令的示例如下所示：

```
>Debug.StepInto
```

有许多支持 Intellisense 功能的命令，但大部分命令在调试时不可用。

12.10.3 使用调试器

既然已了解了一些初步的知识并且打开了调试器窗口，下面就开始调试代码。如果您已经开始了一部分调试，则可以选择关闭调试器并重启它。

这里的存储过程的第一个可执行的代码行具有一些欺骗性——它是@WorkingIn 的声明语句。通常，变量声明不是可执行的，但这里，将初始化变量作为声明的一部分，因此调试器看到了初始化代码。可注意到，现在还未设置任何变量(初始化代码将运行，但还未实际执行)。往前执行(使用菜单选项、工具提示或按 F11 键)，(通过“局部变量”窗口)将看到@WorkingIn 被初始化为值 5——@WorkingOut 未作为声明的一部分初始化。

再次使用“逐语句”图标。开始执行 spTriangular 存储过程并到达存储过程中第一个执行 IF 语句。

因为@ValueIn 的值确实不为 1，所以单步进入 IF 语句指定的 BEGIN…END 块中。特别地，此处转到 SELECT 语句，该语句初始化@InWorking 参数，用于存储过程的特殊执行。稍后将看到，如果@ValueIn 的值确实为 1，那么将立即跳转到 ELSE 语句。

再次通过按 F11 键，或使用“逐语句”图标或菜单选项来向前运行一行代码，直到准备进入下一个 spTriangular 实例之前。

特别注意一下“局部变量”窗口中的@InWorking 的值。可注意到，它改变为 SELECT 语句所设置的正确的值(@ValueIn 当前为 5，所以 5 - 1=4)。同样注意一下，“调用堆栈”窗口中只有当前的存储过程实例(外加当前语句)——因为还没有跳转到嵌套的存储过程中，所以只有一个实例。

现在继续并执行下一条语句。因为这里执行了一个存储过程，所以将在“调试器”窗口中看到一系列的变化。可注意到，指示当前语句的箭头又跳转回 IF 语句。这是为什么呢？因为这是同一个存储过程的一个新实例。从“调用堆栈”窗口中可看到，现在它有两个列出的存储过程实例。顶部(带有黄色箭头)的那个实例是当前实例，带有红色断点的那个实例是等待沿调用堆栈往上的父实例。同样可注意到@ValueIn 参数的值为 2——这是从存储过程的外部实例传入的值。

如果想看一下存储过程的外部实例的作用域内的变量值，则双击“调用堆栈”窗口内的该实例行(带有绿色箭头的行)，就会看到“调试”窗口中的一些变化。

这里需要注意两点。第一，变量的值已经变回存储过程的外部(以及当前选中的)实例作用域内的值了。第二，当前执行行的图标有所不同。这个新的绿色箭头意味着这是该存储过程实例的当前行，但它不是整个调用堆栈中的当前行。

www.TopSage.com

通过单击“调用堆栈”窗口的顶部项可以回到当前实例。接着单步执行 3 次，就会转到该存储过程的第 3 个实例的最顶端的一行(IF 语句)。注意，此时调用堆栈变成了 3 层，而且在“局部变量”窗口中的变量值和参数值再次发生了改变。最后一点也很重要，可注意到，这次@ValueIn 参数值为 3。重复此过程，直到@ValueIn 参数变为 1。

再次单步执行代码，会看到一些改变。这次，由于@ValueIn 中的值为 1，因此转到 ELSE 语句定义的 BEGIN…END 块中。

由于已到达底部，下面准备从调用堆栈返回。使用“逐语句”选项执行过程的最后一行，将看到调用堆栈变成 4 层。同样可注意到，已经适当设置了输出参数(@OutWorking)。

这次，尝试使用“跳出”选项(Shift+F11)。如果不仔细观察的话，那么可能会感觉没有什么改变。

注意：

这种情况下的结果还是比较有欺骗性的。同样，注意一下“调用堆栈”窗口和“局部变量”窗口中值的变化——这里跳出了当前存储过程的实例并在调用堆栈中向上移了一层。如果继续单步执行代码(F11)，那么将完成存储过程的运行并能看到状态窗口的最后形式和它们各自的终值。这里特别需要注意！如果想要看一下真实的终值(例如设置的输出参数)，那么就确保使用“逐语句”选项执行最后一行代码。

如果使用一次执行多行的选项，例如“运行”或“跳出”，则只能得到没有任何最终变量信息的输出窗口。

解决方法是在存储过程最外面的实例中、在希望执行 RETURN 的末尾设置断点。这样的话，可以使用任何调试模式来运行程序，但是仍会在最后中断执行，从而可检查最终的变量。

由此可见，调试器确实是非常便于使用的。

12.11 .NET 程序集

由于程序集是一个很宽泛的问题，程序集会给数据库带来异常的复杂性，所以这里并不过多地去讨论它们，只是说明有这样一个概念。

.NET 程序集可以和系统关联起来，并用来帮助实现真正复杂的操作。例如，可以在一个用户自定义函数中使用.NET 程序集，以从外部数据源(可能是需要动态调用的数据源，如新闻种子或股票报价)提供数据，尽管以前的版本是想在所需的结构和复杂通信中消除这一功能的。

目前不过多地讨论程序集的细节，只是看一下向数据库中添加一个程序集的语法：

```
CREATE ASSEMBLY <assembly name> AUTHORIZATION <owner name> FROM <path to assembly>
WITH PERMISSION_SET = [SAFE | EXTERNAL_ACCESS | UNSAFE]
```

这里的 CREATE ASSEMBLY 和所有 CREATE 语句的作用一样——它指定了创建的对象类型和对象名称。

接下来是 AUTHORIZATION，用于设置运行该程序集的上下文。也就是说，如果它有需要访问的表，那么在 AUTHORIZATION 中如何设置用户或角色名称将决定是否可以访问这些表。

在这之后是 FROM 子句。FROM 子句指定了该程序集的路径以及该程序集的清单。

www.TopSage.com

最后是 WITH PERMISSION_SET。它有以下 3 个选项。

- SAFE：这个选项指的是安全。它限制程序集访问 SQL Server 的外部内容。像文件或网络就不能被程序集访问。
- EXTERNAL_ACCESS：允许外部访问，如访问文件或网络，但是要求程序集仍旧以托管代码形式运行。
- UNSAFE：同样，此选项指的是不安全。它不仅允许程序集访问外部系统对象，也能运行非托管代码。

注意：

如果不在安全模式下运行.NET 程序集，则可能会有危险。即使是在 EXTERNAL_ACCESS 模式下，该模式也允许系统的用户以别名模式访问网络、文件或其他外部资源——也就是说，他们可能得到您并不想让他们得到的东西，而且在进行访问时，他们在网络上采用的是 SQL Server 登录名的别名。所以要十分小心这个问题。

Professional SQL Server 2008 Programming 一书中会对.NET 程序集作更深入的讨论。

12.12 本章小结

本章介绍了很多的内容。从开发人员可在 SQL Server 中实现的功能来看，这一章是本书中最重要的章节。

存储过程是 SQL Server 代码的主要部分。通过存储过程，可以创建可复用的代码，并同时获得更好的性能和灵活性。可以使用您所熟悉的其他编程语言中的多种编程结构，但是存储过程并不意味着一切。

存储过程的优点包括以下几个方面。

- 通常可获得更好的性能
- 通常作为安全隔离层来使用(控制数据库访问和更新的方式)
- 可重用的代码
- 对代码进行区域化(可以封装业务逻辑)
- 根据在运行时建立的动态过程灵活执行

存储过程的缺点包括以下几个方面。

- 不能在平台间移植(例如，Oracle 就有一种完全不同的存储过程实现方法)
- 在某些情况下可能陷入错误的执行计划(实际上会影响性能)

存储过程并不是万能的解决方案，但是它们仍是 SQL Server 编程的基石。第 13 章将介绍与存储过程类似的 UDF。

www.TopSage.com

第13章

用户自定义函数

本章将讲述我个人最喜欢的一个主题。虽然引入用户自定义函数(UDF，User Defined Functions)已有较长时间，但它仍然是 SQL Server 中一个未被充分利用和正确理解的对象。简而言之，当 Microsoft 首先在 SQL Server 2000 中引入用户自定义函数的时候，它令人生畏，而 SQL Server 2005 中.NET 的出现为它增添了一些额外的魅力。假如您是按顺序阅读本书的话，那么意味着您已经了解了编写用户自定义函数所需的大多数知识。用户自定义函数和存储过程非常相似——但它们也有一些其特定的行为和能力，使之与存储过程相区别并在很多情况下解决问题。

本章不仅介绍 UDF 的概念，而且还会说明 UDF 的不同类型、它们与存储过程的区别以及在何种情况下使用 UDF。最后，本书将介绍如何使用.NET 扩展它们的功能。

13.1 UDF 的定义

和存储过程很类似，用户自定义函数一组有序的 T-SQL 语句，UDF 被预先优化和编译并且可以作为一个单元来进行调用。UDF 和存储过程的主要区别在于返回结果的方式。为了能支持多种不同的返回值，UDF 比存储过程有更多的限制。

提示：

上面已经介绍了 UDF 的概念，但这里还是要说明：UDF 绝对不是存储过程的替代品，而是一个不同的选择，可以提供另一种代码灵活性。

使用存储过程时可传入参数，也可以传出参数值。可以返回值，不过该值用于指示成功或失败，而非返回数据。也可以返回结果集，但是在没有将结果集插入到某种表(通常是临时表)中以供后面使用的情况下，不能在查询中真正使用它们。即使使用表值输出参数，在查询中使用结果之前，仍至少需要额外的一个步骤。

然而，使用 UDF 时可传入参数，但不可传出参数。输出参数的概念被更为健壮的返回值取代了。和系统函数一样，可以返回标量(scalar)值，这个值的好处是它并不像在存储过程中那

www.TopSage.com

样只限于整型数据类型，而是可以返回大多数的 SQL Server 数据类型(13.2 节将作更多的介绍)。

就像午夜电视商业广告说的那样："请不要离开，广告以后更精彩！"这里的"更精彩"是指不仅局限于返回标量值，还可以返回表。这是很有用的，将在本章的后面详细讨论。

这里做个小结，UDF 有以下两种类型。

- 返回标量值的 UDF
- 返回表的 UDF

看一下创建 UDF 的基本语法：

```
CREATE FUNCTION [<schema name>.]<function name>
  ( [ <@parameter name> [AS] [<schema name>.]<data type> [ = <default value>
[READONLY]]
    [ ,...n ] ] )
RETURNS {<scalar 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 }[;]
```

这是个很难解释的语法，因为可选的语法部分取决于在 CREATE 语句中所作的选择。这里的问题在于是返回一个标量数据类型还是表，是使用基于 T-SQL 的函数还是使用 CLR 和.NET。下面将分别介绍这些内容。

13.2 返回标量值的 UDF

这种类型的 UDF 可能是您最期待的。它和大多数 SQL Server 内置的函数一样，会向调用脚本或存储过程返回标量值，像 GETDATE()或 USER()函数就会返回标量值。

先前提过，UDF 的好处在于返回值并不限于整数，而是可以返回除了 BLOB、游标(cursor)和时间戳(timestamp)以外的任何有效的 SQL Server 数据类型(包括用户自定义类型)。即使想返回整数，UDF 也有以下两个吸引人的方面。

- 与存储过程不同，用户自定义函数返回值的目的是提供有意义的数据；而对于存储过程来说，返回值只是说明成功或失败，如果失败，则会提供一些关于失败性质的特定信息。
- 可在查询中内联执行函数(例如，作为 SELECT 语句的一部分包含它)，而使用存储过程则不行。

现在创建一个简单的 UDF，看一下它和存储过程不同的使用方法。虽然从代码的角度来说它很简单，但是它还是说明了存储过程与 UDF 的区别。

最常见的一种函数式需求是确定 datetime 字段内的一项是否发生在特定的某一天。通常的问题是这个 datetime 字段含有特定的时间信息，这就不太容易只比较日期。实际上，在第 12

www.TopSage.com

章中我们已经在一些比较中看到了这个问题。

现在回到第 5 章创建的 Accounting 数据库。假设想要了解今天所有订单的情况。首先添加几个带有今天的日期的订单。这里将从已有的表中取出客户和雇员 ID(如果您现在没有任何记录，那么需要插入一些哑(dummy)数据行以供引用。这里将创建一个小型循环来添加一些数据行：

```
USE Accounting;

DECLARE @Counter int = 1;

WHILE @Counter <= 10
BEGIN
   INSERT INTO Orders
      VALUES (1, DATEADD(mi,@Counter,GETDATE()), 1);
   SET @Counter = @Counter + 1;
END
```

这些代码插入了 10 行记录，每一行数据都是以今天的日期插入的，而互相之间都相差 1 分钟的时间。

提示：

如果您正好是在午夜之前执行这段代码，那么其中的一些数据行可能会插入明天的日期，所以这里请当心。对于除夜猫子类型的人之外的所有人来说，这都是可以正常运行的。

现在运行下列这个简单的查询来看一下今天已有的订单情况。

```
SELECT *
FROM Orders
WHERE OrderDate = GETDATE();
```

遗憾的是，这个查询并不能返回任何数据。这是因为 GETDATE()函数获得的是精确到毫秒的时间，而不只是日期。这意味着任何基于 GETDATE()的查询都不能返回任何数据，即使是发生在同一天也不行(对于 smalldatetime 字段来说，可能发生在同一分钟，对于 datetime 字段来说可能发生在同一毫秒，对于 datetime2 字段来说，可能接近 100 毫秒)。

常见的解决方案是把日期转换为字符串并转换回来以截断时间，然后作比较。

代码如下所示：

```
SELECT *
FROM Orders
WHERE CONVERT(varchar(12), OrderDate, 101) = CONVERT(varchar(12),
 GETDATE(), 101)
```

注意：

也可以只是将@Date 的值强制转换为 date 数据类型。这里选择使用 CONVERT，只是为了选择使用一种向后兼容的截断日期的方法(SQL Server 2005 及更早的版本不支持 date 数据类型)。

这次，无论订单的时间如何，都将得到 OrderDate 列中今天的日期的所有数据行。可惜的是，这些代码的可读性并不是最好的。想象一下，如果有一大串的日期需要作这类比较——其

www.TopSage.com

结果肯定是可读性不好。

现在使用一个简单的用户自定义函数来实现这个功能。首先，需要创建实际的函数。这通过新的 CREATE FUNCTION 命令完成，它的格式和存储过程很像。例如，创建这个函数的代码如下所示：

```
CREATE FUNCTION dbo.DayOnly(@Date date)
RETURNS date
AS
BEGIN
   RETURN @Date;
END
```

其中，从 GETDATE()返回的日期将作为参数传入，而且函数体内将包含转换日期的工作，最后将返回截取的日期值。

注意，上述的版本是一个 SQL Server 2008 兼容版本，这依赖于强制转换为参数的 date 数据类型来截断时间。如果想在 SQL Server 2005 进行这样的截断(就像基于查询的示例中所做的那样)，需要像前面一样使用 CONVERT 函数。例如：

```
CREATE FUNCTION dbo.DayOnly(@Date datetime)
RETURNS varchar(12)
AS
BEGIN
   RETURN CONVERT(varchar(12), @Date, 101);
END
```

为了看一下这个函数的效果，这里稍微修改一下查询：

```
SELECT *
FROM Orders
WHERE dbo.DayOnly(OrderDate) = dbo.DayOnly(GETDATE());
```

得到的结果和前面的独立查询一样。即使对于像这样一个简单查询来说，新的代码可读性更好一点。这个调用和在大多数支持函数的语言中一样运行良好。然而，也有一个问题，即需要指定模式(schema)。SQL Server 可能会因为某种原因并不能像对其他对象一样解析标量值函数。

正如您所猜测的，UDF 所提供的远不只有可读性。可以在 UDF 中嵌入查询并将它们用作子查询的封装方法。几乎任何可返回一个离散值的过程式代码都可以封装进 UDF 中并且可以在查询中内联使用。

这里来看一个简单的子查询。该子查询如下所示：

```
USE AdventureWorks2008;

SELECT Name,
      ListPrice,
      (SELECT AVG(ListPrice) FROM Production.Product) AS Average,
       ListPrice - (SELECT AVG(ListPrice) FROM Production.Product)
         AS Difference
FROM Production.Product
WHERE ProductSubCategoryID = 1; -- The Mountain Bikes Sub-cat
```


www.TopSage.com

这将返回一组非常简单的数据：

```
Name                        ListPrice      Average        Difference
--------------------------- -----------    -------------  --------------
Mountain-100 Silver, 38     3399.99        438.6662       2961.3238
Mountain-100 Silver, 42     3399.99        438.6662       2961.3238
Mountain-100 Silver, 44     3399.99        438.6662       2961.3238
Mountain-100 Silver, 48     3399.99        438.6662       2961.3238
Mountain-100 Black, 38      3374.99        438.6662       2936.3238
Mountain-100 Black, 42      3374.99        438.6662       2936.3238
...
...
Mountain-500 Silver, 52      564.99        438.6662        126.3238
Mountain-500 Black, 40       539.99        438.6662        101.3238
Mountain-500 Black, 42       539.99        438.6662        101.3238
Mountain-500 Black, 44       539.99        438.6662        101.3238
Mountain-500 Black, 48       539.99        438.6662        101.3238
Mountain-500 Black, 52       539.99        438.6662        101.3238

(32 行受影响)
```

这里做一次新的尝试，这次将把计算平均值和差值的功能封装到两个函数中。第一个函数封装了计算平均值的任务，而第二个封装了计算差值的任务：

```
CREATE FUNCTION dbo.AveragePrice()
RETURNS money
WITH SCHEMABINDING
AS
BEGIN
RETURN (SELECT AVG(ListPrice) FROM Production.Product);
END
GO
   CREATE FUNCTION dbo.PriceDifference(@Price money)
RETURNS money
AS
BEGIN
   RETURN @Price - dbo.AveragePrice();
END
```

注意，在一个 UDF 中嵌套另一个 UDF 是完全合法的。

提示：

在函数中使用 WITH SCHEMABINDING 选项和在视图中使用该选项是一样的——如果是使用模式绑定(schema-binding)创建函数，则在没有删除该模式绑定函数之前都不能修改或删除任何该函数依赖的对象。在本例中，虽然模式绑定并不是必需的，但这里想说明一下其用法并且用这个示例为下面章节作准备。

现在使用新的函数而不使用原有的子查询模型运行查询：

```
USE AdventureWorks2008

SELECT Name,
```


www.TopSage.com

```
        ListPrice,
        dbo.AveragePrice() AS Average,
        dbo.PriceDifference(ListPrice) AS Difference
FROM Production.Product
WHERE ProductSubCategoryID = 1; -- The Mountain Bikes Sub-cat
```

这将得到和使用子查询所获得的相同的结果。

注意，除了可读性问题以外，这里还得到了代码重用的额外好处。对于这个小示例来说，它可能看上去并没有什么作用，但是随着函数越来越复杂，这会节省很多时间。

13.3 返回表的 UDF

SQL Server 中的用户自定义函数并不只限于返回标量值。它可以返回一些更有意思的内容——表。虽然您对其作用可能有所了解，但是这里将作进一步的介绍，说明返回的表在很大程度上和其他表一样都是很有用的。可以对返回表的 UDF 执行 JOIN，甚至对结果应用 WHERE 条件。这确实非常有用。

改为用表作为返回值并不难。对于 UDF 来讲，表就像任何其他 SQL Server 数据类型一样。为了说明这点，这里创建一个相对简单的函数，如下所示：

```
USE AdventureWorks2008
GO

CREATE FUNCTION dbo.fnContactList()
RETURNS TABLE
AS
RETURN (SELECT BusinessEntityID,
            LastName + ', ' + FirstName AS Name
        FROM Person.Person);
GO
```

这个函数返回一个包含所选择记录的表并且作了一些格式化工作——连接姓和名，并以逗号分隔。

现在，准备像表一样使用函数：

```
SELECT *
FROM dbo.fnContactList();
```

现在添加一些更有意思的内容。到目前为止，对该表的处理就像使用视图一样简单，甚至说更容易。如果想要把视图参数化，该怎么办呢？例如，只想用姓来过滤结果(而不必手动创建 WHERE 子句)。其代码如下所示：

```
--CREATE our view
CREATE VIEW vFullContactName
AS
SELECT p.BusinessEntityID,
          LastName + ', ' + FirstName AS Name,
          ea.EmailAddress
       FROM Person.Person as p
```


www.TopSage.com

```
        LEFT OUTER JOIN Person.EmailAddress ea
          ON ea.BusinessEntityID = p.BusinessEntityID;
GO
```

这会得到想要的结果，但是也有一个小问题。不可以把视图本身进行参数化，所以必须在查询中包含 WHERE 子句。

```
SELECT *
FROM vFullContactName
WHERE Name LIKE 'Ad%';
```

这将得到下列结果：

```
BusinessEntityID    Name                        EmailAddress
----------------    --------------------------  ------------------------
67                  Adams, Jay                  jay0@adventure-works.com
301                 Adams, Frances              frances0@adventure-works.com
305                 Adams, Carla                carla0@adventure-works.com
...
...
16901               Adams, Adam                 adam46@adventure-works.com
16902               Adams, Eric                 eric57@adventure-works.com
16910               Adams, Jackson              jackson47@adventure-works.com

(87 行受影响)
```

为了更简单起见，把所有的代码都封装到一个函数中：

```
USE AdventureWorks2008;
GO

CREATE FUNCTION dbo.fnContactSearch(@LastName nvarchar(50))
RETURNS TABLE
AS
RETURN (SELECT p.BusinessEntityID,
          LastName + ', ' + FirstName AS Name,
          ea.EmailAddress
      FROM Person.Person as p
      LEFT OUTER JOIN Person.EmailAddress ea
        ON ea.BusinessEntityID = p.BusinessEntityID
      WHERE LastName Like @LastName + '%');
GO
```

要执行该函数，只需调用它并提供参数：

```
SELECT *
FROM fnContactSearch('Ad');
```

这样就得到了相同的结果集——没有 WHERE 子句，没有过滤 SELECT 列表；可以反复使用该函数，而不需要进行“剪切和粘贴”。同时需要注意的是，虽然可以使用存储过程和 EXEC 命令得到相似的结果，但是不可能直接地把存储过程的结果和另一个表连接起来。

所有这一切可能很令人兴奋，但有时需要的不只是单个的 SELECT 语句，也不只是一个参数化的视图。事实上，就像在标量函数中看到的那样，可能需要执行多个语句来得到想要的结

www.TopSage.com

果。用户自定义函数正好支持这个概念。确实，它们可以返回使用多条语句创建的表——唯一的区别在于必须对需要返回的元数据(就像临时表一样)进行命名和定义。

为了说明这个示例，将讨论关系数据库中常见的一个问题——分层数据。

假设您在人力资源部门工作。有一个 Employees 表，它有一个一元关系(与同一表中另一列相关联的外键)，该关系通过 ManagerID 列把雇员同他们的雇主相关联——也就是，通过把 ManagerID 列与另一个 EmployeeID 列相关联的方式来知道某人的雇主是谁。这个场景中一个常见的需求是创建一个报告“树”——也就是显示在一个组织机构图中在特定经理下面的所有人员的列表。

过去，关系数据库在处理分层数据方面的能力较弱。大量文章、白皮书和书籍都谈到此问题。幸运的是，SQL Server 2008 引入了一种新的方法来处理分层数据。新引入的功能是 hierarchyID 数据类型和一组内置函数，有助于处理关系数据库中的树型数据结构。这些新功能都较高级，需要花些精力去掌握，因此暂时不深入介绍它们。这里将把它们视为超出本书范围——可查看“高级编程”版本中的“高级数据结构”一章来了解有关新的 HierarchyID 数据类型的更多内容。

注意：

如果您想了解 SQL Server 2008 中新的分层功能的示例，可查看 AdventureWorks2008 中的 HumanResources.Employee 表的 OrganizationNode 和 OrganizationLevel 列。

为了继续分层数据的讨论，将用过去一直使用的方法处理层次结构。由于 AdventureWorks2008 并没有提供用这种旧(仍很流行)方法处理分层数据的好的示例，这里准备自己创建一个新版本的 Employee 表(称之为 Employee2)，它将实现这种老的处理层次结构的方法。如果运行第 3 章中的 BuildAndPopulateEmployee2.sql 文件，那就可以获得新版本的 Employee 表。或者可从 wrox.com 或 professionalsql.com 网址下载并执行它。

该脚本创建的表如图 13-1 所示。

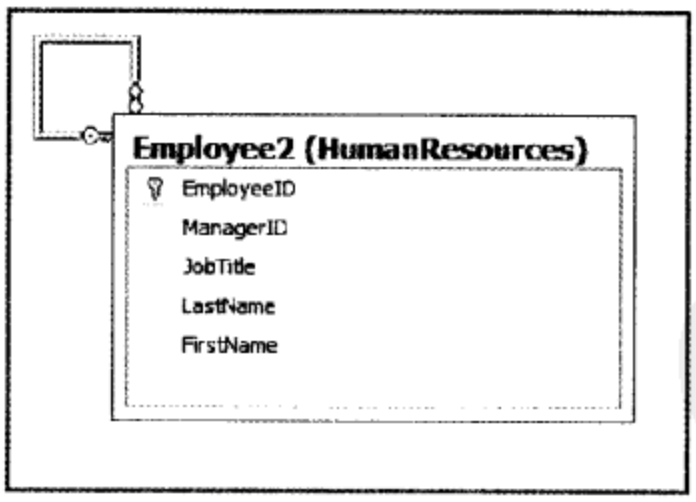


图 13-1

假定已执行了该脚本并有了 Employee2 表，那么看看是否可以检索向 Karla Huntington 报告的人员的列表。

乍一看，这个问题相当简单。如果想要知道所有需要向 Karla 报告的人员，则可能需要编写一个查询，使 Employee2 表和它自己相连接，如下所示：

www.TopSage.com

```
USE AdventureWorks2008;

SELECT
      TheReport.EmployeeID,
      TheReport.JobTitle,
      TheReport.LastName,
      TheReport.FirstName
FROM
      HumanResources.Employee2 as TheBoss
JOIN HumanResources.Employee2 AS TheReport
      ON TheBoss.EmployeeID = TheReport.ManagerID
WHERE TheBoss.LastName = 'Huntington' AND TheBoss.FirstName = 'Karla';
```

这看上去给出了想要的结果：

```
EmployeeID   JobTitle                LastName         FirstName
----------   --------------------    ---------        ---------------
5            VP of Engineering       Olsen            Ken
6            VP of Professional      Services Cross   Gary
7            VP of Security          Lebowski         Jeff
(3 行受影响)
```

但实际上存在一些问题。这一问题是我们想要的是整个 Karla 报告链中的所有人员——并不只是直接向 Karla 报告的人员，而是需要间接向其报告的人员。如果看一下新创建的 Employee2 表中的所有记录，就会发现有许多向 Ken Olsen 报告的人员，但是他们并没有出该查询结果中。

提示：

可能反应快并且更有经验的读者会说："没有关系，只要再一次重新连接 Employee2 表并获取下一级的报告人员就行了！"对于只有少量数据集，或者层次结构的层数固定的情况来说，这样做是可以的。但是如果层次结构的层数不确定呢？如果有向 Robert Cheechov 报告的人员，而且还有向 Robert Cheechov 底下的人报告的人员的话，依次等等，那该怎么办呢？

这里需要的是一个函数，该函数返回所提供的 EmployeeID(或 ManagerID)下面的所有的层次结构。为此，这里有一个经典的递归示例。递归是指一个代码块任何时候都调用它自身。第 12 章中讲述 spTriangular 存储过程有这样的一个示例。现在来考虑一下这个场景。

(1) 需要了解向指定的经理作报告的所有人员。

(2) 对于第(1)步中的每一个人，用户想要知道向他们作报告的人员。

(3) 重复第(2)步，直到再没有下一级人员。

这就是递归。它意味着需要使用一些语句来使函数这样执行：一些语句指出当前的层次，并且至少一条以上的语句来再次调用相同的函数以得到下一个最低的层次。

提示：

记住 UDF 和存储过程有着相同的递归层次限制——也就是说，只能进行 32 层的递归，如果您达到这个限制的话，那就要想办法在代码中避免这个错误。

现在把所有内容都整合起来。注意一下函数中声明的几个变化。这次，需要把名字和返回

www.TopSage.com

值(本例中为@Reports)相关联——这在使用多条语句生成结果时都是必须的。同时还需要定义想要返回的表——这允许 SQL Server 在将表返回调用例程前对将要插入表的对象进行验证。

```
CREATE FUNCTION dbo.fnGetReports
       (@EmployeeID AS int)
       RETURNS @Reports TABLE
       (
       EmployeeID int NOT NULL,
       ManagerID int NULL
       )
AS
BEGIN

/* Since we'll need to call this function recursively - that is once for each
** reporting employee (to make sure that they don't have reports of their
** own), we need a holding variable to keep track of which employee we're
** currently working on. */
DECLARE @Employee AS int;

/* This inserts the current employee into our working table. The significance
** here is that we need the first record as something of a primer due to the
** recursive nature of the function - this is how we get it. */
INSERT INTO @Reports
   SELECT EmployeeID, ManagerID
   FROM HumanResources.Employee2
   WHERE EmployeeID = @EmployeeID;

/* Now we also need a primer for the recursive calls we're getting ready to
** start making to this function. This would probably be better done with a
** cursor, but we haven't gotten to that chapter yet, so.... */
SELECT @Employee = MIN(EmployeeID)
FROM HumanResources.Employee2
WHERE ManagerID = @EmployeeID;

/* This next part would probably be better done with a cursor but we haven't
** gotten to that chapter yet, so we'll fake it. Notice the recursive call
** to our function! */
WHILE @Employee IS NOT NULL
   BEGIN
      INSERT INTO @Reports
         SELECT *
         FROM fnGetReports(@Employee);

         SELECT @Employee = MIN(EmployeeID)
         FROM HumanResources.Employee2
         WHERE EmployeeID > @Employee
            AND ManagerID = @EmployeeID;
   END

 RETURN;

END
GO
```


www.TopSage.com

这个函数只提供关于雇员和他们经理的最少的信息，如果需要取得更多的信息，可以返回连接 Employee2 表。这里作了个随意的选择，向结果添加了选中的经理。这样做主要是为了支持这个递归的场景并且为结果集提供一个基本结果。看一下结果，因为 Karla 是 EmployeeID 4，所以向函数中填入它：

```
SELECT * FROM fnGetReports(4);
```

代码返回的不仅是向 Karla Huntington 作报告的人员，还有那些向 Ken Olsen(他要向 Huntington 女士作报告)作报告的人员以及 Huntington 女士自己(记住，这里加入了 Huntington 女士作为起点)。

```
EmployeeID  ManagerID
----------- -----------
4             1
5             4
8             5
9             5
10            5
11            5
6             4
7             4

(8 行受影响)
```

现在进行最后一步工作，将它与实际数据连接。将像原来查询向 Karla Huntington 作报告的人员那样操作：

```
DECLARE @EmployeeID int;

SELECT @EmployeeID = EmployeeID
FROM HumanResources.Employee2 e
WHERE LastName = 'Huntington'
AND FirstName = 'Karla';

SELECT e.EmployeeID, e.LastName, e.FirstName, m.LastName AS ReportsTo
FROM HumanResources.Employee2 AS e
JOIN dbo.fnGetReports(@EmployeeID) AS r
   ON e.EmployeeID = r.EmployeeID
JOIN HumanResources.Employee2 AS m
   ON m.EmployeeID = r.ManagerID;
```

这样将得到 Huntington 女士下面所有 7 位员工的信息：

```
EmployeeID  LastName     FirstName     ReportsTo
----------- ------------ --------------- ---------------
4           Huntington   Karla         Smith
5           Olsen        Ken           Huntington
8           Gutierrez    Ron           Olsen
9           Bray         Marky         Olsen
10          Cheechov     Robert        Olsen
11          Gale         Sue           Olsen
```


www.TopSage.com

```
6                Cross           Gary                Huntington
7                Lebowski        Jeff                Huntington

(8 行受影响)
```

正如您所见，实际上可以用非常复杂的代码来建立表结果，但是这只是生成结果表而已，因此，它也能像其他表那样使用。

理解确定性

如果要对 UDF 作全面性的讨论，就必须介绍一下“确定性”这一概念。如果 SQL Server 要建立一个索引，则它必须能确定性地定义(明确定义)对哪个项进行索引。这和函数有什么关系呢？因为有一些函数需要为将建立索引的对象(计算列或索引视图)提供数据。

用户自定义函数可以是确定性的也可以是非确定性的。确定性并不是根据任何参数类型定义的，而是根据函数的功能定义的。如果给定了一组特定的有效输入，每次函数就都能返回相同的结果，那么就说该函数是确定性的。SUM()就是一个确定性的内置函数。3、5、10 的总和永远都是 18——只要每一次都是以这些值作为输入调用函数。但是 GETDATE()的值就是非确定性的，因为每次调用它的时候 GETDATE()都会改变。

为了达到确定性的要求，函数必须满足以下 4 个条件。

- 函数必须是模式绑定的。这意味着函数所依赖的任何对象会有一个依赖记录，并且在没有删除这个依赖的函数之前都不允许改变这些对象。
- 函数引用的所有其他函数，无论其是用户定义的还是系统定义的，都必须是确定性的。
- 不能引用在函数外部定义的表(可以使用表变量和临时表，只要它们是在函数作用域内定义就行)。
- 不能使用扩展存储过程。

确定性的重要性在于它显示了是否要在视图或计算列上建立索引。如果可以可靠地确定视图或计算列的结果，那么才允许在视图或计算列上建立索引。这意味着，如果视图或计算列引用非确定性函数，则在该视图或列上将不允许建立任何索引。这并不是坏事，但是在对使用该函数的视图或列创建索引前，需要想一下该函数是否是确定性的。

所以，这样会产生一个问题：“如何判定函数是否是确定性的？”除了检查上面所描述的规则以外，SQL Server 也能告诉您函数是否是确定性的——这一信息存储在对象的 IsDeterministic 属性中。可以利用 OBJECTPROPERTY 函数检查。例如，可以检查一下本章先前使用过的 DayOnly 函数的确定性：

```
USE Accounting;

SELECT OBJECTPROPERTY(OBJECT_ID('DayOnly'), 'IsDeterministic');
```

您可能会惊讶地发现该函数是非确定性的：

```
-----------
0
(1 行受影响)
```

回头看一下有关确定性函数的要求列表，看看您是否能说出该函数是非确定性的原因。

www.TopSage.com

提示：

当我使用该示例的时候，有人提醒过我这个问题。我确信该函数应是确定性的，但实际上它不是。经过了长时间的冥思苦想，我才发现漏掉了 SCHEMABINDING。

幸运的是，可以改正这个问题。需要做的只是为该函数加上 WITH SCHEMABINDING 选项，可以看到以下结果：

```
ALTER FUNCTION dbo.DayOnly(@Date date)
RETURNS date
WITH SCHEMABINDING
AS
BEGIN
   RETURN @Date;
END
```

下面重新运行一下 OBJECTPROPERTY 查询：

```
-----------
1
(1 行受影响)
```

现在，它成了一个确定性函数！

将该函数与在 AdventureWorks2008 数据库中创建的 AveragePrice 函数相比，AveragePrice 函数如下所示：

```
CREATE FUNCTION dbo.AveragePrice()
RETURNS money
WITH SCHEMABINDING
AS
BEGIN
   RETURN (SELECT AVG(ListPrice) FROM Production.Product);
END
GO

CREATE FUNCTION dbo.PriceDifference(@Price money)
RETURNS money
AS
BEGIN
   RETURN @Price - dbo.AveragePrice();
END
```

在这个函数的一开始就使用了模式绑定，因此看一下 OBJECTPROPERTY：

```
USE AdventureWorks2008;

SELECT OBJECTPROPERTY(OBJECT_ID('AveragePrice'), 'IsDeterministic');
```

尽管它是模式绑定的，但是返回的结果仍然是非确定性的。这是因为该函数引用了一个非本地表(在函数外部创建的一个临时表或表变量)。

同时也值得关注一下与 AveragePrice 一起创建的 PriceDifference 函数，该函数也是非确定

www.TopSage.com

性的。一方面，Pricedifference 函数下是模式绑定的，但是更重要的是它引用了 AveragePrice——如果引用了一个非确定性函数的话，则通过联合创建的函数也是非确定性的。

13.4 调试用户自定义函数

调试用户自定义函数的方式和第 12 章中调试存储过程是一样的。

只要建立一个调用函数的脚本，然后单步执行该脚本(使用工具栏图标或是按 F11 键)。然后就可以单步进入 UDF。

13.5 数据库中的.NET

正如第 12 章所说的那样，在 SQL Server 2005 中，可以在存储过程中或函数中使用.NET 程序集。和存储过程类似，这对于函数来说也很有深意。

考虑到读者大多数都是初学者，所以很难充分说明.NET 对于数据库的影响。事实是您不会经常使用它，但是当您使用它的时候，会发现其效果显著。您需要为特殊函数实现复杂的公式吗？没有问题。您需要访问像信用卡授权公司之类的外部数据源吗？没有问题。您需要访问其他复杂的数据源吗？也没有问题。简而言之，以前必须跳过的操作或者必须通过复杂的开发工作实现的操作一下子变得很简单了。

对于函数来说这意味着什么呢？前面已经给出了一个在函数中实现复杂公式的示例。现在假设一个外部表格数据的情况——要用表格的形式表示.csv 或其他一些数据，那么可在 SQL Server 中通过.NET 程序集创建函数。

然后，SQL Server 中的.NET 程序集仍然是一个高级的概念。将在 SQL Server 2008 的“高级编程”系列中介绍它。虽说如此，还是要清楚可以采用.NET 程序集来解决一些束手无策的问题。

13.6 本章小结

本章所添加的内容在很大程度上来讲并不都是新的内容。事实上，用户自定义函数中的内容和在脚本和存储过程中所看到的语句、变量以及一般编码方式都是一样的。不过，UDF 仍然提供了以前 SQL Server 中无法实现的功能。现在可以封装大量代码，并在查询中内联使用这一封装功能。而且，现在可以提供参数化视图和动态创建表。

在很多方面，用户自定义函数都是 SQL Server 中最有用的新功能。然而，由于本书对 UDF 的使用只是粗浅的。在下一个版本中，可能会有开发人员深入地实现 UDF。

13.7 练习

1. 将第 12 章中的 spTriangular 存储过程重新实现为函数。

www.TopSage.com

第14章

事务和锁

学习本章内容，将对您的工作有很大帮助。本章要介绍的内容并不很难，但事务和锁定一直是数据库中最容易被误解的两个概念。因此，掌握了这一“入门级”(至少本书认为它是基础的)的概念会使您看上去像是一位真正的专家。

本章将要介绍下列内容：

- 事务
- SQL Server 日志和检查点的工作方式
- 锁

本章将介绍这些主题之间的联系为何如此紧密，以及如何使用它们来最小化相互之间问题。

14.1 事务

事务全部是关于原子性(atomicity)的。原子性的概念是指可以把一些事情当作一个单元来看待。从数据库的角度看，它是指应全部执行或全部都不执行的一条或多条语句的最小组合。

当处理数据时，经常要确保如果一件事情发生，另一件事情也随之发生，或者是两件事都不发生。实际上，可能达到的程度是有 20 件事情(或更多)都必须一起发生或都不发生。来看一个经典的示例。

假设您是一位银行家。Sally 想要从支票账户向储蓄账户转账 1000 美元。当然，您乐意做这件事，所以就处理她的请求。

这一场景背后的实际代码如下所示：

```
UPDATE checking
   SET Balance = Balance - 1000
   WHERE Account = 'Sally'
UPDATE savings
   SET Balance = Balance + 1000
   WHERE Account = 'Sally'
```

www.TopSage.com

这是对将要发生的事情的一个高级简化，但是它抓住了重点：需要对每一个账户发出不同的语句指令。

现在，如果第一条语句执行，而第二条没有执行，将是什么结果呢？Sally 会少 1000 美元！从您的角度来说(您刚才赚了 1000 美元)，这在短时间之内还不错，但是这不可能持续很久。因为从这个下午开始，将会有一批忠实的客户离开您的银行——没有存款人，是无法在银行业中待下去的。

您需要一种方法确保如果第一条语句执行，则第二条语句也会执行。实际并不存在这样一种可完全确定的方法——从硬件故障到违反数据完整性规则这样简单的事情，都可能发生错误。然而，幸运的是，有一种可以达到相同目的的方法——基本上可以忘记曾经发生过第一条语句。至少可以强制这个概念：如果某件事情没有发生，那么什么也不会发生——至少在事务的作用范围之内是这样。

然而，为了理解事务的概念，需要能够定义非常明确的边界。事务要有非常明确的开始和结束点。事实上，SQL Server 中的每一条 SELECT、INSERT、UPDATE 和 DELETE 语句都是隐式事务的一部分。即使只发出一条语句，也会把这条语句当作一个事务——要么执行语句中的所有内容，要么什么都不执行。事实上，事务的默认长度为一条语句。

但是，如果需要的不只是一条要么都执行或什么都不执行的语句——就像前面的银行示例那样，该怎么办呢？在这种情况下，就需要有一种方法来标记事务的开始和结束，以及事务的成功或失败。可以使用一些 T-SQL 语句在事务中“标记”这些点，如下所示。

- BEGIN TRAN：设置起始点。
- COMMIT TRAN：使事务成为数据库中永久的、不可逆转的一部分。
- ROLLBACK TRAN：本质上说想要忘记它曾经发生过。
- SAVE TRAN：创建一个特定标记符，只允许作部分回滚。

在将它们组合成一个事务之前，先分别看一下它们各自的内容。

14.1.1 BEGIN TRAN

事务的开始可能是事务过程中最容易理解的概念。它唯一的目的就是表示一个单元的开始。如果由于某种原因，不能或者不想提交事务，那么这就是所有数据库活动将要回滚的起点。也就是说，就数据库而言，它会忽略这个起点之后的最终没有提交的所有语句。

其语法如下所示：

```
BEGIN TRAN[SACTION] [<transaction name>|<@transaction variable>]
[ WITH MARK [ <'description'> ]]
```

这里并不打算详述 WITH MARK 选项，因为这一主题涉及事务工作中的非常高级的知识点，更多的是面向管理员的，超出了本书的范围。

14.1.2 COMMIT TRAN

事务的提交是一个事务的终点。当发出 COMMIT TRAN 命令时，可以认为该事务是持久的。也就是说，事务的影响现在是持久的并会持续，即使发生系统故障也不受影响(只要有备份或者数据库文件没有被物理破坏就行)。撤消已完成事务的唯一方法是发出一个新的事务。从功

www.TopSage.com

能上而言，该事务是对第一个事务的反转。

COMMIT 的语法非常类似于 BEGIN 语法：

```
COMMIT TRAN[SACTION] [<transaction name>|<@transaction variable>]
```

14.1.3　ROLLBACK TRAN

每当考虑 ROLLBACK 时，我就会想到一部老电影《公主新娘》。如果您看过这部电影(如果没有看过，建议您看一下)，就会记得主人公 Vizzini 常说的一句话："如果任何事情出现了问题，那么就回到起点"。

这是个有用的忠告。ROLLBACK 所做的就像 Vizzini 建议的那样——它回到了起点。在这种情况下，是指事务回到了起点。从关联的 BEGIN 语句开始发生的任何事情事实上都会被忘记。不过也有例外，那就是使用保存点(save point)，稍后将会介绍它。

除了允许保存点外，ROLLBACK 的语法看上去和 BEGIN 或 COMMIT 语句一样：

```
ROLLBACK TRAN[SACTION] [<transaction name>|<save point name>|
   <@transaction variable>|<@savepoint variable>]
```

14.1.4　SAVE TRAN

保存事务从本质上说就是创建书签(bookmark)。为书签建立一个名称(可以有多个名称)。在建立了"书签"之后，可以在回滚中引用它。创建书签的好处是可以回滚到代码中的特定点上——只要为想要回滚到的那个保存点命名。

其语法也是相当简单的：

```
SAVE TRAN[SACTION] [<save point name>| <@savepoint variable>]
```

关于保存点需要记住的是，ROLLBACK 时会清除它们——也就是说，即使保存了 5 个保存点，如果执行 ROLLBACK，则所有的保存点都会消失。可以重新设置新的保存点，并且回滚到这些点，但是当发出 ROLLBACK 时，所有已有的保存点都将消失。

SAVE TRAN 可能会让您有点迷惑，本书并不推荐初学者使用，但是请记住它的存在。

14.2　SQL Server 日志的工作方式

在了解 SQL Server 如何跟踪数据库中已有数据之前，您必须明确地知道事务的概念。可以看到，您所了解的数据库只不过是所有数据的一个完整版本而已。除了需要将所有的内容写入磁盘这种很少的情况外，数据库中数据的构成不仅包括物理数据库文件中的数据，而且还包括从最后一次的检查点后提交给日志的事务。

在数据库的正常操作中，大多数执行的活动都是"记录"在事务日志上，而非直接写入数据库中。检查点是指强制地把数据库现在所使用的所有脏(dirty)页写入磁盘的周期性操作。脏页是指一些在读入到缓存后已经被修改，但是所进行的修改还没有写入到磁盘的日志或数据页。如果没有检查点，日志就会填满和/或使用所有可用的磁盘空间。这个过程如图 14-1 所示。

www.TopSage.com

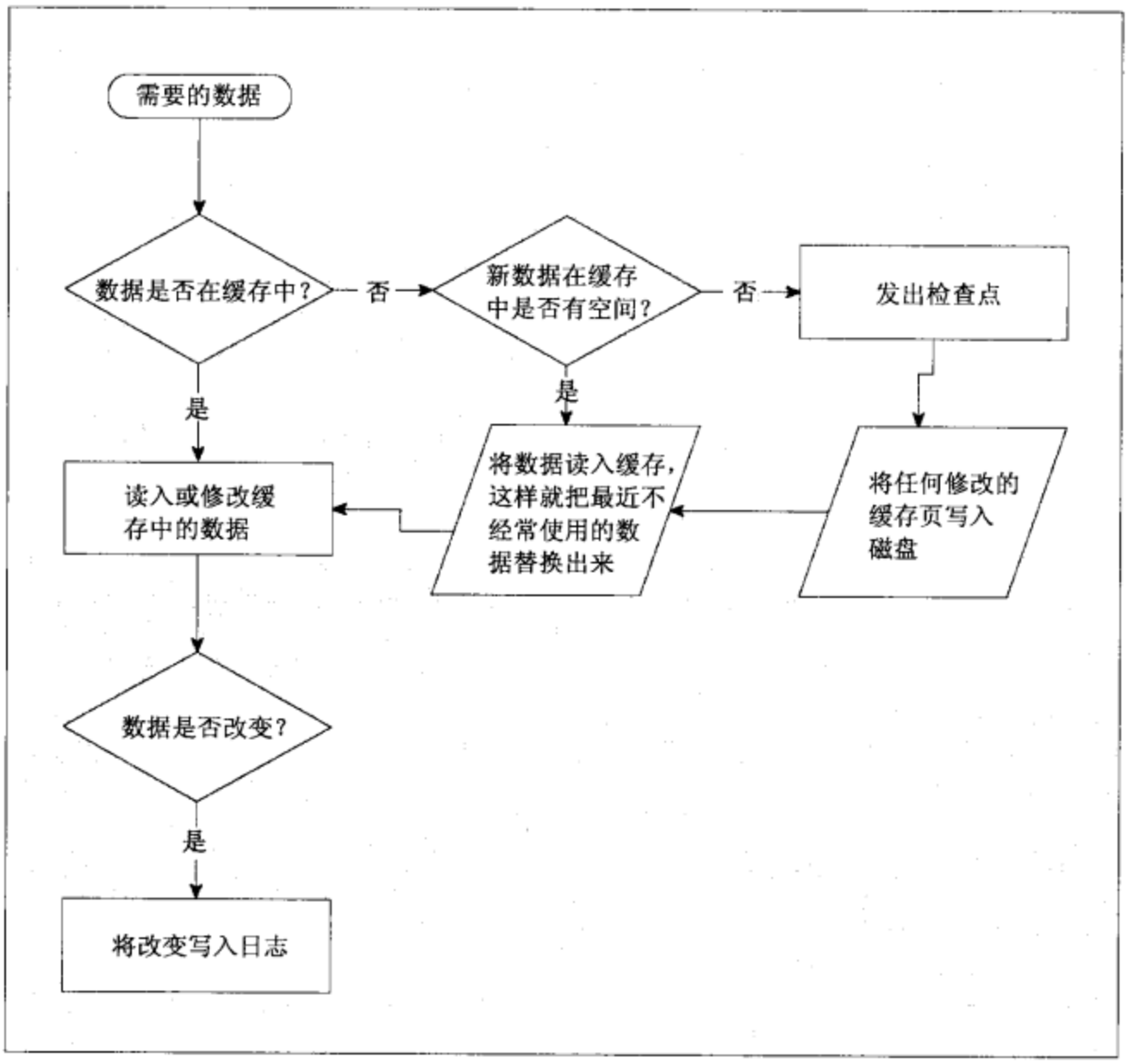


图 14-1

提示：

不要误解这样做的含意是要您必须为获取缓存中的数据做一些特殊的工作。SQL Server 将完成所有这些工作。这些信息只是用来方便您理解日志工作的原理，和处理事务需要的步骤。一些数据是否在缓存中对于性能的影响很大，所以当寻求最佳性能时，理解何时记录日志以及何时进出缓存是很重要的。

注意，需要向已满的缓存中读入数据并不是发出检查点的唯一原因。还可能在以下情况下发出检查点。

- 通过一条手动的语句——使用 CHECKPOINT 命令
- 在正常关闭服务器的时候(除非使用 WITH NOWAIT 选项)
- 当修改任何数据库选项的时候(例如，单用户、只供 dbo 使用等)
- 当使用 Simple Recovery 选项并且日志超过 70%填满的时候
- 当从最后一个检查点开始的日志中的数据总量(通常称为日志的活动部分)超过服务器在 recovery interval 选项指定的时间里可恢复的数据大小时

14.2.1 失败和恢复

恢复(recovery)总是在 SQL Server 每次启动时发生。SQL Server 获取数据库文件，并且应用

www.TopSage.com

自最后的检查点以后日志中任何提交的改变(通过把它们写入到物理数据库文件)。日志中任何没有相对应提交的改变都会回滚——也就是说，会从根本上忘记这些改变。

这里看一下恢复是如何根据数据库中事务发生的方式工作的。假设有 5 个跨日志的事务，如图 14-2 所示。

这里依次地看一下这些事务发生的情况。

1. 事务 1

绝对什么事务都不发生。这个事务已经通过了一个检查点，并且已经完全向数据库提交。没有什么需要恢复，因为已读入到数据缓存的任何数据已经反映提交的事务。

2. 事务 2

即使事务存在于发出检查点的那一刻，也说明仍然没有提交该事务(该事务仍在运行)。没有提交，该事务就不会加入到检查点中。因此，这个事务就会“前滚”。需要将所有相关的页读回到缓存中，然后使用日志中的信息来重新运行在该事务中已经运行过的所有语句。当完成这个操作后，事务看上去就和系统失败前的情况一样。

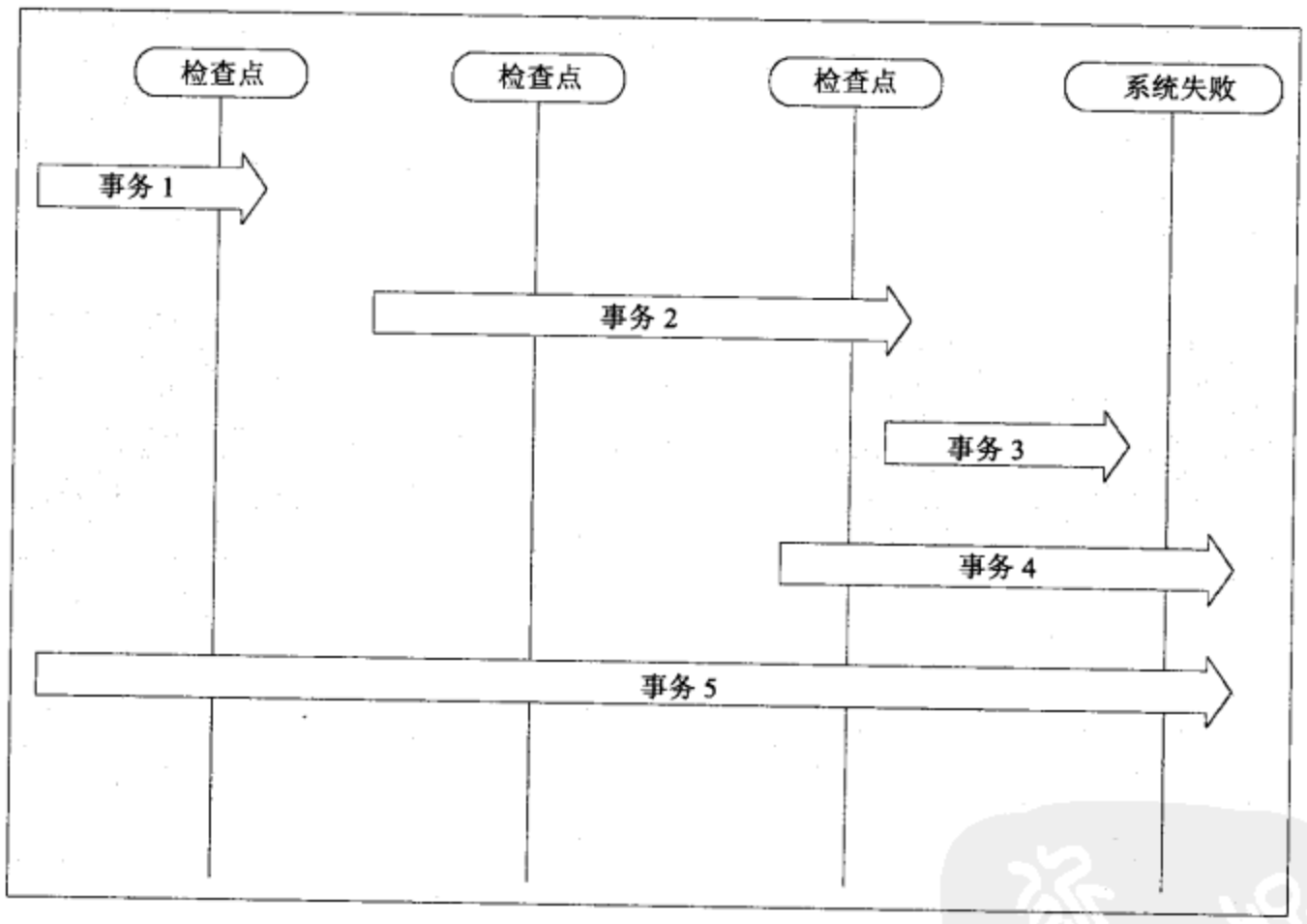


图　14-2

3. 事务 3

不能局部地看待这个事务。从需要完成的操作的角度来说，它和事务 2 是完全相同的。同样，由于事务 3 并不是在发出最后一个检查点时完成，因此它并没有参与到该检查点中，就像事务 2 一样。唯一的区别在于事务 3 甚至在发出最后一个检查点时还不存在，但从恢复的角度来说并没有什么区别——只是发出命令的位置有区别。

www.TopSage.com

4. 事务 4

该事务在系统失败时还没有完成，所以它必须回滚。事实上，从行数据的角度来看，这种情况永远不会发生。用户必须重新输入数据，并且任何过程都需要重新开始。

5. 事务 5

这个事务和事务 4 没有区别。它看上去有点不同，因为这个事务运行的时间很长，但是这没有任何区别。这个事务在系统失败的时候没有提交，所以必须回滚。

14.2.2 隐式事务

为了能和其他主要的 RDBMS 系统(如 Oracle 或 DB2)相兼容，SQL Server 支持(默认情况下是关闭的，但是可以选择将它打开)隐式事务的概念。隐式事务不需要 BEGIN TRAN 语句——而是由第一条语句自动地启动。它们会一直运行，直到发出 COMMIT TRAN 或 ROLLBACK TRAN 语句。下一事务将从下一条语句开始。

提示：

隐式事务是危险的，它不在本书的介绍范围之内。这里强烈建议关掉这个选项，除非有非常特殊的原因需要打开它(例如为了与在其他系统中编写的代码相兼容)。

14.3 锁和并发

并发(concurrency)是任何数据库系统都会遇到的一个主要问题。它指这样一个概念：两个或两个以上的用户都尝试在同一时间与同一个对象交互。对每个用户来说，交互操作的本质都是不同的(更新、删除、读取、插入)，处理控制对象竞争的理想方法根据其中用户的操作以及他们操作的重要性而改变。可同时成功运行的用户(更确切地说是事务)越多，并发程度就越高。

在 OLTP 环境中，并发通常是要在数据处理中的第一件事，并且是本书提出的大多数数据库概念的核心所在(OLAP 通常是后知后觉型的——它本不必采用这样的方式，但是实际采用了)。处理并发问题对于系统性能来说是关键性的。在数据库中处理并发的基础基础就是锁定处理。

锁是一种防止在某对象执行动作的一个进程与已在该对象上执行的其他进程相冲突的机制。也就是说，如果有其他人在操作某个对象，那么您就不能在该对象上进行操作。您能否执行的操作取决于其他用户正在进行的操作。锁也是描述正在执行的操作的一种方式，因此系统可以知道第二个进程动作是否和第一个进程相兼容。例如，系统可处理的 1 个、2 个、10 个、100 个、1 000 个或者任意多个用户连接数，通常可能在同一时间共享同一块数据，只要它们都是只读访问记录。把它想象成一个卖水晶的商店——许多人可以看里面的商品——即使是相同的商品——只要他们不去移动、购买或者改变商品。如果不止一个人在同一时刻采取行动，那么很可能最终导致水晶被打破。这也就是店主通常要细心观察，然后决定谁先处理事务的原因。

SQL Server 的锁管理器(lock manager)就是这样的店主。当您进入 SQL Server“商店”时，

www.TopSage.com

锁管理器将询问您的意图——您准备要做什么。如果您只是说“看看而已”，并且已经有人在商店里但也只是“看看而已”，那么锁管理器就会让您进入商店。如果您想“购买”(更新或删除)，那么锁管理器就会检查是否已经有人在店里面“购买”。如果有的话，那么您必须等待，而且在您之后的任何人也必须等待。当允许您进去“购买”的时候，那么任何人都不允许进去直到您结束“购买”为止。

通过这种方式，SQL Server 可以避免并发问题所引起的各种不同问题。下面将研究可能的并发问题以及设置相互排斥的事务隔离级别的方法。但是，现在先看一下能否锁定的对象以及可用的锁定类型。

14.3.1 通过锁可以防止的问题

锁可以解决以下 4 种主要问题。

- 脏读
- 非重复性读取
- 幻读
- 丢失更新

这其中的每一个都代表了一类不同的问题，可以通过混合的解决方案处理，这些解决方案通常包括了正确设置事务隔离级别。同时，这里还将说明对于其中的每种问题应使用哪种事务隔离级别，以方便读者在后面的学习过程中回过头来查阅。后面将详细介绍一下隔离级别，但是现在先确保理解这些问题的含意。

1. 脏读

如果一个事务读取的记录是另一个未完成事务的一部分，那么这时就发生了脏读。如果第一个事务正常完成，那么就不可能有问题。但是，如果该事务回滚呢？那将从在数据库看来从未发生的事务中获取信息。

这里看一下如表 14-1 所示的示例中的一系列步骤。

表 14-1

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 显示的内容
BEGIN TRAN		3		
UPDATE col = 5	BEGIN TRAN	3	5	
SELECT anything	SELECT @var = col	3	5	5
ROLLBACK	UPDATE anything SET whatever = @var		5	5

这其中有问题！

事务 2 现在使用的是一个非有效的值！如果尝试返回并审核这个值来自哪里，那么可能不

www.TopSage.com

会有任何跟踪结果。

幸运的是，如果使用 SQL Server 默认的事务隔离级别(称为 READ COMMITTED，本节后面的 14.4 节将介绍)，则不会发生这样的情况。

2. 非重复性读取

很容易将非重复性读取和脏读相混淆。不要担心——它只是术语而已。只要知道其概念就行。

如果在一个事务中两次读取记录，而另一个事务在这期间改变了数据，就会发生非重复性读取。对于这个问题，可看一下之前那个银行示例。记住，不希望让账户的金额低于 0 美元，如表 14-2 所示。

表 14-2

事 务 1	事 务 2	@Var	事务 1 所认为的表中的值	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		125	125 (等待锁定清除)	75
(结束，等待锁定清除，然后继续)		125	75	或者是 - 25 (如果不存在 CHECK 约束强制 >0)，或者是 547 错误(如果存在 CHECK 约束)

这里再次碰到了问题。事务 1 进行了预扫描(这在有些实例中是很有用的，请查看第 12 章“在错误发生前处理错误”的内容)来确保该值是有效的，并且事务能进行下去(账户里有充足的金额)。问题在于，在执行 UPDATE 之前，事务 2 先于事务 1 作出了一个决定性的动作。如果在表上没有任何 CHECK 约束来防止负值，则会把它设置为-25——即使逻辑上使用了 IF 语句来防止这个问题的出现。

可以通过下面两种方式来防止这个问题。

www.TopSage.com

- 创建 CHECK 约束并监控 547 错误。
- 将隔离级别设置为 REPEATABLE READ 或 SERIALIZABLE。

CHECK 约束看上去相当直观。要知道的是，这使用的是一种被动的而非主动的方法。然而，在很多情况下可能需要使用非重复性读取，所以这在很多情况下是首选。

稍后将详细介绍隔离级别，现在只要知道把它设置为 REPEATEBLE READ 或 SERIALIZABLE 在解决很多问题的同时，也会带来很多麻烦。它只是一种选择而已。

3. 幻读

这里并不是要谈论“歌剧院”中的“幻影”，而是讨论那些看上去很神秘的记录，这些记录好像不会受到发出的 UPDATE 或 DELETE 语句的影响。这可能会非常自然地发生在对系统进行操作的正常过程中，并且不需要任何精心准备的场景来阐明这个问题。此处有一个经典的示例来说明幻读是如何发生的。

假设您正在经营一家快餐店。如果您实行的是基本编制标准，就可能就会有相当数量的员工拿的都是政府规定的“最低工资”。现在政府决定将最低工资从每小时$6.25 提到$7.50，您要对 Employees 表执行更新操作，使得每一位低于每小时$7.50 的员工工资都涨到新的最低工资标准。您可能会说没有问题，并且采用如下简单的语句：

```
UPDATE HumanResources.EmployeePayHistory
SET HourlyRate = 7.50
WHERE HourlyRate < 7.50;

ALTER TABLE HumanResources.EmployeesPayHistory
   ADD CONSTRATNT ckWage CHECK (HourlyRate >= 7.50);
GO
```

这很简单，是吗？实际上是错误的！只是为了说明，这里给出了错误消息：

```
消息 547，级别 16，状态 0，第 1 行
ALTER TABLE 语句与 CHECK 约束"ckWage"冲突。
该冲突发生于数据库"AdventureWorks2008"，表"HumanResources.EmployeePayHistory",
column 'Rate'。
```

快速运行一条 SELECT 语句来检查一下每小时低于$7.50 的工资，可以肯定的是，您可以找到一条这样的记录。您可能会很快地提出疑问：“怎么会是这样？刚才已经执行了 UPDATE 来解决这个问题。”您的确是运行了这条语句，并且它也运行良好——但是您得到的只是幻读。

幻读的示例是很少的，并且只会在某个凑巧的环境下发生。简而言之，它是指在运行 UPDATE 语句的同时有人执行了 INSERT 语句。因为这是一个全新的数据行，所以没有被锁定，而且运行良好。

解决这个问题的唯一方法是设定事务隔离级别为 SERIALIZABLE，在这种情况下，任何对表的更新都不能放入 WHERE 子句中，否则它们将被锁在外面。

4. 丢失更新

丢失更新发生在一个更新成功写入数据库后，而又意外地被另一个事务重写时。您可能会

www.TopSage.com

问："这是如何发生的？"

如果有两个事务读取整个记录，然后其中一个向记录写入了更新信息，而另一个事务也向该记录写入更新信息，这时就会出现丢失更新。这里看一个示例。

假设您是公司的一位信用分析员。您接到客户 X 打来的电话，说他已到达他的信用额度上限，想申请增加额度，所以您查看了这位客户的信息。您发现他的信用额度是$5 000，并且看似他每次都能按时付款。

当您在查看的时候，信用部门的另一位员工 Sally 也读取了客户 X 的记录，并输入信息改变客户的地址。她读取的记录也显示信用额度为$5 000。

这时，您决定把客户 X 的信用额度提高到$7 500，并且按下 Enter 键。数据库现在显示客户 X 的信用额度为$7 500。

Sally 现在更新了客户地址，但是她仍然使用和您一样的编辑屏幕——也就是说，她更新了整个记录。还记得她的屏幕上显示的信用额度吗？是$5000。数据库现在又一次显示客户 X 的信用额度为$5 000。您的更新丢失了！

解决这个问题的方法取决于在您读取某数据和要更新数据的这段时间内，代码以何种方式识别出另一个连接已经更新了该记录。这个识别的方式取决于您使用的访问方法。

14.3.2 可以锁定的资源

对于 SQL Server 来说，有 6 种可锁定的资源，而且它们形成了一个层次结构。锁的层次越高，它的粒度就越粗(也就是说，选择了以级联的方式锁定更多的对象，因为已经锁定了包含它们的对象)。按粒度由粗到细排列，这些资源包括：

- **数据库**：锁定整个数据库。这通常发生在整个数据库模式改变的时候。
- **表**：锁定整个表。这包含了与该表相关联的所有数据相关的对象，包括实际的数据行(它们中的每一行)以及与该表相关联的所有索引中的键。
- **区段**：锁定整个区段。记住，因为一个区段由 8 页组成，所以区段锁定是指锁定控制了区段、控制了该区段内的 8 个数据或索引页以及在这 8 页中的所有数据行。
- **页**：锁定该页中的所有数据或索引键。
- **键**：在索引中的特定键或一系列键上有锁。相同索引页中的其他键不受影响。
- **行或行标识符(RID)**：虽然从技术上讲，锁是放在行标识符(它是 SQL Server 的内部结构)上的，但是在本质上，它锁定了整个数据行。

14.3.3 锁升级和锁对性能的影响

升级(escalation)是指能够认识到维持一个较细的粒度(例如，行锁而不是页锁)只在被锁定的项数较少时有意义。而随着越来越多的项被锁定，维护这些锁的系统开销实际上会影响性能。这会导致锁持续更长的时间(这样会产生争议问题——锁的时间越长，其他用户想使用这个特定记录的可能也越大)。当考虑这个问题的时候，您会意识到可能会有一种权衡的办法，这就是锁管理器使用升级的用意。

当维持锁的数量达到一定的限度时，则锁升级为下一个更高的层次，并且将不需要再如此紧密地管理低层次的锁(释放资源，而且有助于提高速度)。

注意，升级是基于锁的数量，而不是用户的数量。这里的重点是，可以通过执行大量的更

www.TopSage.com

新来单独地锁定表——行锁可以升级为页锁，页锁可以升级为表锁。这意味着可能将所有其他用户锁在该表之外。如果查询使用了多个表，则它很可能将每个人锁在这些表之外。

提示：

虽然您并不想将其他所有的用户锁在对象之外，但是有时需要执行可能会出现将具有这种效果的更新。对于升级来说，除了使查询尽可能有目的性之外，您几乎别无办法。如果认识到将发生升级，就要确保已经考虑过了查询的可能后果。

14.3.4 锁定模式

除了需要考虑锁定的资源层次以外，还要考虑查询将要获得的锁定模式。就像需要对不同的资源进行锁定一样，也有不同的锁定模式。

一些模式是互相排斥的(指不能一起使用)。一些模式什么都不做，只修改其他的模式。模式是否可以一起使用取决于它们是否是兼容的(本章后面将详细讨论锁之间的兼容性问题)。

和前面介绍可以锁定的资源时一样，下面将逐个介绍每个锁定模式。

1. 共享锁

这是最基本的一种锁。共享锁用于只需要读取数据的时候，也就是说不会改变任何内容。共享锁能和其他共享锁兼容。这不意味着共享锁不会带来麻烦——虽然共享锁并不介意其他锁的存在，但是有些锁并不能和共享锁共存。

共享锁告诉其他锁，某用户已经在那边了。它们并不提供很多的功能，但是不能忽略它们。然而，共享锁能做的是防止用户执行脏读。

2. 排他锁

排他锁可谓是意如其名。排他锁不与其他任何锁兼容。如果有任何其他的锁存在，则不能使用排他锁，而且当排他锁仍然起作用的时候，它们不允许在资源之上创建任何形式的新锁。这可以防止两个人同时更新、删除或执行任何操作。

3. 更新锁

更新锁是共享锁和排他锁的混合。更新锁是一种特殊的占位符。考虑一下，为了能执行 UPDATE，需要验证 WHERE 子句(假设有的话)来指出想要更新的具体的数据行。这意味着只需要一个共享锁，直到真正地进行物理更新。在物理更新期间，需要一个排他锁。

更新锁表明了有一个共享锁，它在您完成初始数据扫描并指出需要更新的数据后变成排他锁。这说明了一个事实：更新操作有以下两个不同的阶段。

- 第一个阶段指出了满足 WHERE 子句条件的内容(需要更新的行)。这是更新查询的一部分，该查询有一个更新锁。
- 第二个阶段是如果决定执行更新，那么锁将升级为排他锁。否则，将把锁转换为共享锁。

这样做的好处是它防止了死锁(deadlock)。死锁本身不是一种锁定类型，而是一种已经形成矛盾的状况。如果一个锁不能完成需要清除资源的工作是因为另一个锁正在占用该资源，则会引起死锁——问题在于这个对应的资源本身也在等待清除第一个事务上的锁。

www.TopSage.com

如果没有更新锁，死锁会一直出现。两个更新查询会在共享模式下运行。Query A 完成了它的查询工作并准备进行物理更新。它想升级为排他锁，但是不可以这么做，因为 Query B 正在完成查询。除非 Query B 需要进行物理更新，否则它会完成查询。为了做到这点，Query B 必须升级为排他锁，但是不能这么做，因为 Query A 正在等待。这样就造成了僵局。

而更新锁阻止建立其他的更新锁。第二个事务只要尝试取得一个更新锁，它们就会进入等待状态，直到锁超时为止——将不会授予这个锁。如果第一个锁在锁超时之前清除的话，则锁定会授予给新的请求者，并且这个处理会继续下去。如果不清除，则会发生错误。

更新锁只与共享锁以及意向共享锁相兼容。

4. 意向锁

意向锁(Intent Lock)是真正的占位符，它用来处理对象层次问题的。假设一下如下情况：已对某一行建立了锁，但是有人想在页上或区段上建立锁，或者是修改表。您肯定不想让另一个事务通过达到更高的层次来妨碍您。

如果没有意向锁，那么较高层次的对象将不会知道在较低层次上有锁。意向锁可改进性能，因为 SQL Server 只需要在表层次上检查意向锁(而不需要检查表上的每个行锁或者页锁)，以此来决定事务是否可以安全地锁定整个表。意向锁分为以下 3 种不同的类型。

- **意向共享锁**：该意向锁指已经或者将要在层次结构的一些较低点处建立共享锁。例如，将要在页上面建立一个页级的共享锁。这种类型的锁只能应用于表和页。
- **意向排他锁**：它与意向共享锁是一样的，但是将会在低层项上设置排他锁。
- **共享意向排他锁**：它指已经或将会在对象层次结构下面建立共享锁，但目的是为了修改数据，所以它会在某个时刻成为意向排他锁。

5. 模式锁

模式锁(Schema Lock)分为以下两种。

- **模式修改锁(Sch-M)**：对对象进行模式改变。在 Sch-M 锁期间，不能对对象进行查询或其他 CREATE、ALTER 或 DROP 语句的操作。
- **模式稳定性锁定(Sch-S)**：它和共享锁很相似；这个锁的唯一目的是防止 Sch-M，因为在该对象上已有其他查询(或 CREATE、ALTER、DROP 语句)的锁。它与其他所有的锁定相兼容。

6. 批量更新锁

批量更新锁(BU)只是一种略有不同(但是意义重大的)的表锁定变体形式。批量更新锁允许并行加载数据——也就是说，对于其他任何普通操作(T-SQL 语句)都会将表锁定，但可以同时执行多个 BULK INSERT 或 bcp 操作。

14.3.5 锁的兼容性

表 14-3 显示了资源锁定模式的兼容性(按照锁定强度列出)。现有的锁以列显示，要兼容的锁以行显示。

www.TopSage.com

表　14-3

锁 的 类 型	IS	S	U	IX	SIX	X
意向共享锁(IS)	YES	YES	YES	YES	YES	NO
共享锁(S)	YES	YES	YES	NO	NO	NO
更新锁(U)	YES	YES	NO	NO	NO	NO
意向排他锁(IX)	YES	NO	NO	YES	NO	NO
共享意向排他锁(SIX)	YES	NO	NO	NO	NO	NO
排他锁(X)	NO	NO	NO	NO	NO	NO

另外：

- Sch-S 与除 Sch-M 以外的所有锁定模式相兼容。
- Sch-M 和所有的锁定模式不兼容。
- BU 只与模式稳定性锁以及其他的批量更新锁相兼容。

14.3.6　指定一种特有的锁定类型——优化器提示

有时想要在查询中或在整个事务中对锁定有更多的控制。可以通过使用优化器提示(optimizer hints)来实现这一点。

优化器提示明确告诉 SQL Server 将一个锁升级为特有的层次。这些提示信息包含在将要影响的表(在 SQL 语句中)的名称之后。

提示：

优化器提示是一个高级主题。有经验的 SQL Server 开发人员会经常使用它，并且他们相当重视它。

可以这样想一下——Microsoft 已经为查询优化器以及清楚在何种情况下使用何种锁定投入了数百万美元。查询提示用于对优化器可能不知道的少许方面进行调整，但是在大多数情况下，您不会比他们的优化器小组知道得更多。暂时不考虑这些内容，可以等到进一步学习 SQL Server 时再作了解(将在“高级编程”系列中介绍它们)。

使用 Management Studio 确定锁

查看锁的最好方式是使用 Management Studio。通过使用 Activity Monitor，Management Studio 会以两种方式显示锁——通过 process ID 或 object。

为了使用 Management Studio 显示锁，只要导航到<Server>的 Activity Monitor 节点，其中的<Server>是要监控其活动的服务器的顶级节点。SQL Server 将显示如图 14-3 所示的新窗口。

展开感兴趣的节点(Overview 部分默认展开——这里手动展开了 Processes 部分)，可以通过滚动条查看大量度量值——包括当前系统中有效的锁。

www.TopSage.com

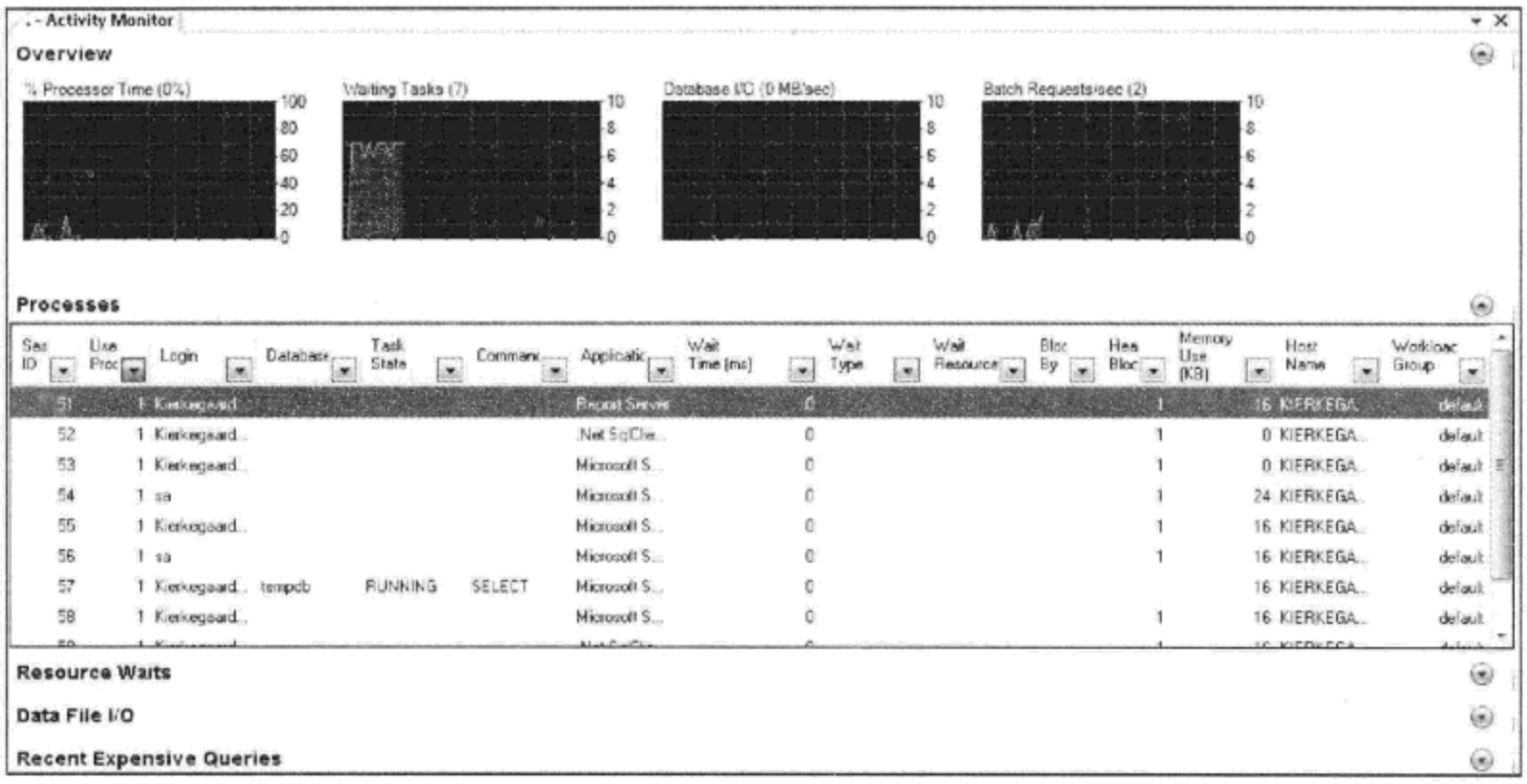


图 14-3

提示：

可能 Activity Monitor 窗口中最棒的功能是当右击 Process 窗口中的特定进程时，可以很容易地针对特定进程启动 SQL Profiler 跟踪。当对多种问题进行故障检修时，这是非常有用的。

14.4 设置隔离级别

前面介绍过，可以使用不同的锁定策略阻止一些不同类型的问题。也介绍过可以使用的锁的类型以及它们如何影响资源的可用性。现在来看一下这些进程管理部分是如何一起工作来确保整个数据的完整性的——确保得到想要的结果。

首先，要理解事务和锁之间的联系是很紧密的。默认情况下，一旦创建了任何与数据修改相关的锁，该锁定就会在整个事务期间存在。如果有一个大型事务，就意味着将在很长一段时间内阻止其他进程访问锁定的对象。这毫无疑问是有问题的。

然而，这只是默认情况。事实上，可以设置以下 5 种不同的隔离级别。

- READ COMMITTED(默认的)
- READ UNCOMMITTED
- REPEATABLE READ
- SERIALIZABLE
- SNAPSHOT

在这些隔离级别之间进行切换的语法也相当直观：

```
SET TRANSACTION ISOLATION LEVEL <READ COMMITTED|READ UNCOMMITTED
   |REPEATABLE READ|SERIALIZABLE|SNAPSHOT>
```

对隔离级别的修改只会影响到当前的连接——所以不必担心会影响到其他的用户(或者是

www.TopSage.com

他们影响您)。

首先进一步介绍默认情况(READ COMMITTED)。

1. READ COMMITTED

通过 READ COMMITTED，任何创建的共享锁将在创建它们的语句完成后自动释放。也就是说，如果启动了一个事务，运行了一些语句，然后运行 SELECT 语句，再运行一些其他的语句，那么当 SELECT 语句完成的时候，与 SELECT 语句相关联的锁就会释放——SQL Server 并不会等到事务结束。

动作查询(UPDATE、DELETE 和 INSERT)有点不同。如果事务执行了修改数据的查询，则这些锁将在事务期间(以防需要回滚)保持有效。

通过设置 READ COMMITTED 这一默认隔离级别，可以确定有足够的数据完整性来防止脏读。然而，仍会发生非重复性读取和幻读。

2. READ UNCOMMITTED

READ UNCOMMITTED 是所有隔离级别选择中最危险的，但是它在速度方面有最好的性能。

设置隔离级别为 READ UNCOMMITTED 将告诉 SQL Server 不要设置任何锁，也不要实现任何锁。通过这个隔离级别，有可能实践本章前面所讨论的任何一种并发问题(特别是指脏读)。

为什么有人要冒脏读的风险呢？当在 Usenet 上查看新闻组的时候，可经常看到此类的问题。对于相当数量的人来说，这是很奇怪的问题，但是也有很好的理由使用这个隔离级别，并且该隔离级别经常和报表有关。

在 OLTP 环境中，锁既是您的保护者也是您的敌人。锁可以防止数据完整性问题，但是锁也经常妨碍或阻止您访问需要的数据。这在如下情况中是很常见的：管理人员需要经常运行报表，但是数据录入人员经常因为管理人员运行报表时设置的锁而无法输入数据或推迟输入。

使用 READ UNCOMMITTED，可以避开这个问题，至少对于数据并不需要很精确的报表来说是如此。例如，假设销售经理想知道今天的销售数据，而且他会在一天内多次提出这个问题(通过重新运行报表的形式)。

如果报表可能的运行时间很长，那么因为锁定的原因，运行这个报表就很可能会影响其他用户的工作效率。但是这个报表的好处在于它是一份真正模糊的报表——其实际数据可能是无意义的。因为这个经理想看的只是些大概的数据。

通过使用 READ UNCOMMITTED 隔离级别，用户不设置任何锁，所以不会阻止任何其他事务的执行。虽然数字可能会让人怀疑(由于脏读的危险)，但是并不需要精确的数字，并且知道这些数字仍然会和精确值非常接近，甚至脏读回滚的可能性极小。

可以通过在查询中添加 NOLOCK 优化器提示达到与 READ UNCOMMITTED 相同的效果。设置隔离级别的好处在于不需要在查询中的每一个表中使用提示，或者是在多个查询中使用提示。使用 NOLOCK 优化器提示的好处在于不需要记住把连接的隔离级别设置为默认值(当隔离级别为 READ UNCOMMITTED 时，要这样做)。

www.TopSage.com

3. REPEATABLE READ

REPEATABLE READ 会稍微地将隔离级别升级，并提供一个额外的并发保护层，这不仅能防止脏读(默认设置已经这样做)，而且能防止非重复性读取。

防止非重复性读取是很大的优势，但是直到事务结束还保持共享锁会阻止用户访问对象，因此会影响效率。就个人而言，本书推荐使用其他的数据完整性选项(例如带有错误处理的 CHECK 约束)，而不是采用这个选择，但是它仍然是可用的选项。

与 REPEATABLE READ 隔离级别等价的优化器提示是 REPEATABLEREAD(两者名称相同，除了有无空格的区别)。

4. SERIALIZABLE

SERIALIZABLE 是堡垒级的隔离级别。除了丢失更新以外，它防止所有形式的并发问题。甚至能防止幻读。

如果设置隔离级别为 SERIALIZABLE，就意味对事务使用的表(或一组表)进行的任何 UPDATE、DELETE、INSERT 操作绝对不满足该事务中任何语句的 WHERE 子句的条件。从本质上说，如果用户想执行一些事务感兴趣的事情，那么必须等到该事务完成的时候。

SERIALIZABLE 隔离级别也可以通过在查询中使用 SERIALIZABLE 或 HOLDLOCK 优化器提示模拟。再次申明，类似于 READ UNCOMMITTED 和 NOLOCK 的讨论，前者不需要每次都设置，而后者需要把隔离级别设置回来。

提示：

从表面上看，使用 SERIALIZABLE 隔离级别可以执行任何操作。而实际上，它给数据库提供了最高层次的隔离性，称为一致性(consistency)——也就是说，如果所有用户在同一时刻运行一个事务(进行连续地处理)，则对于多个用户来说，更新过程是相同的。

但是，和现实生活中的大多数事情一样，这也有一个权衡。从实际意义来说，可以认为一致性和并发性是对立的。使用 SERIALIZABLE 可以阻止其他用户访问需要的对象——这就降低并发性。反之——提高并发性(例如通过 REPEATABLE READ)会降低数据库的一致性。

本书建议您使用默认的隔离级别(READ COMMITTED)，除非您有特殊的原因不想这样做。

5. SNAPSHOT

SNAPSHOT 是最新的一种隔离级别(SQL Server 2005 中引入的)，非常像 READ COMMITTED 和 READ UNCOMMITTED 的组合。要注意的是，SNAPSHOT 默认是不可用的——只有为数据库打开了 ALLOW_SNAPSHOT_ISOLATION 特殊选项时，SNAPSHOT 才可用。

和 READ UNCOMMITED 一样，SNAPSHOT 并不创建任何锁，也不实现任何锁。两者的主要区别是它们识别数据库中不同时段发生的更改。数据库中的更改，不管何时或是否提交，都会被运行 READ UNCOMMITTED 隔离级别的查询看到。而使用 SNAPSHOT，只能看到在 SNAPSHOT 事务开始之前提交的更改。从 SNAPSHOT 事务一开始执行，所有查看到的数据就和在事务开始时提交的一样。

www.TopSage.com

注意：

关于 SNAPSHOT 要注意两件特殊的事。首先，在使用 SNAPSHOT 隔离级别前必须打开特殊数据库选项，并且对于事务中包括的每个数据库都应打开该选项(以防查询跨数据库)。其次，尽管 SNAPSHOT 隔离级别一般并不注意锁或不设置锁，但也有一个特殊情况。如果在 SNAPSHOT 事务进行过程中有数据库恢复回滚，那么 SHANSHOT 事务将适当地设置一个特殊锁作为占位符，然后等待回滚完成。只要回滚一完成，将删除锁，SNAPSHOT 事务将继续正常运行。

14.5 处理死锁(1205 错误)

到现在为止，我们已经了解了锁，也知道了事务。接下来将把注意力转移到处理死锁这一更为麻烦的问题。

前面已经提过，死锁本身不是一种锁的类型，而是由其他锁形成矛盾的情况。不管喜欢与否，您都会经常(特别是刚开始的时候)碰到这个问题，会碰到错误号 1205。这个特殊问题如此常见，以致于许多数据库开发人员经常用 1205 这个错误号来指代死锁。

如果一个锁由于另一个锁占有资源而不能完成应该做的清除资源工作，就会导致死锁；反之亦然，。当发生了死锁时，需要其中的一方赢得这场斗争，所以 SQL Server 选择一个死锁牺牲者。对死锁牺牲者的事务进行回滚，并且通过 1205 错误来通知发生了死锁。另外一个的事务将继续正常地运行(确实，该事务将完全不知道出现了问题，除了会发现执行时间有所增加)。

14.5.1 SQL Server 判断死锁的方式

每隔 5 秒钟，SQL Server 就会检查所有当前的事务，了解他们在等待什么还未被授予的锁。之所以这么做时，本质上是因为存在这样的请求。接下来，它将再一次重新检查所有打开的锁请求的状态，如果先前请求中有一个还没有被授予，则它会递归地检查所有打开的事务，寻找锁定请求的循环链。如果 SQL Server 找到了这样的循环链，则将会选择一个或更多的死锁牺牲者。

14.5.2 选择死锁牺牲者的方式

默认情况下，基于相关事务的“代价”选择死锁牺牲者。SQL Server 将会选择回滚代价最低的事务(换句话说，SQL Server 将做最少的事情来撤消事务)。在一定程度上，可以使用 SQL Server 中的 DEADLOCK_PRIORITY SET 选项来重写它，但是本书并不推荐这样的做法，而且它也超出了本书的范围。

14.5.3 避免死锁

在复杂的系统中不可能 100%地避免死锁，但是从实际的角度出发，几乎总能消除死锁——也就是说，使它们很少发生，从而与系统无关。

为了减少或消除死锁，可遵循下面一些简单的(通常是简单的)规则。

www.TopSage.com

- 按相同的顺序使用对象。
- 使事务尽可能简短并且在一个批处理中。
- 尽可能使用最低的事务隔离级别。
- 在同一事务中不允许无限度的中断(用户交互、批处理分离)。
- 在控制环境中，使用绑定连接。

每次遇到死锁问题，很可能违反了其中至少一条(通常更多)规则。下面分别看一下其中的每一个规则。

1. 按相同的顺序使用对象

这是本书所认为的一些基本的规则中最容易出问题的地方。使用这个规则的好处是，几乎不需要用任何内容去解释它——它更多的是一种思维方式。可以在设计过程之初决定访问数据库对象的方法——包括顺序——而且在为项目编写的每个查询、过程或触发器中也都使用该方法。

可以稍微想一下——如果问题是两个连接互相含有对方想要的内容，那么处理这个问题未免太晚。这里看一个简单的示例。

假定有两个表：Suppliers 和 Products。假设有两个进程将使用这两个表。进程 1 接受库存输入，用手头新的产品总量更新 Products 表，接下来用已经购买的产品总量来更新 Suppliers 表。进程 2 记录销售数据，它在 Suppliers 表中更新销售产品的总量，然后在 Products 中减少库存数量。

如果同时运行这两个进程，那么就有可能碰到麻烦。进程 1 将会在 Products 表上获得一个排他锁。进程 2 将在 Suppliers 表上获取一个排他锁。接下来，进程 1 将试图获取 Suppliers 表上的一个锁，但是进程 1 必须等到进程 2 清除了现有的锁为止。同时，进程 2 将尝试在 Products 表上创建一个锁，但是进程 2 必须等到进程 1 清除了现有的锁为止。现在就有矛盾——两个过程都在等待对方释放锁。SQL Server 必须选择一个死锁牺牲者。

现在重新安排一下这个场景，把进程 2 改变成首先在 Products 中减少库存数量，接着在 Suppliers 表中更新销售产品的总数量。该场景和第一种组织进程的方式在功能上是等价的，而且这种新的方式没有任何代价。但是效果是极好的——不再有死锁(至少死锁不存在于这两个进程之间)！接下来看一下整个过程。

当同时运行这两个进程时，进程 1 将会在 Products 表上获取一个排他锁(到现在为止都是一样的)。接着进程 2 也尝试在 Products 表上操作获取一个锁，但是进程 2 必须等到进程 1 完成为止(注意并没有对 Suppliers 表采取任何操作)。虽然进程 1 完成了更新 Products 表，但是由于事务还没完成，所以进程 1 还不能释放锁。进程 2 仍然在等待清除 Products 表上的锁。现在进程 1 获取了 Suppliers 表的一个锁。而进程 2 继续等待进程 1 清除 Products 表上的锁。进程 1 根据需要完成了事务提交或事务回滚，无论是哪种情况，进程 1 都会释放所有的锁。现在，进程 2 可以在 Products 表上获得锁，而且将没有意外地继续执行剩下来的事务。

只是交换了一下两个查询运行的顺序，就消除了一个潜在的死锁问题。应该尽可能地按相同的顺序使用对象，这样死锁问题会更少。

www.TopSage.com

2. 使事务尽可能简短

这是另一个基本问题。同样，这应成为本能的行为——并没有真正去考虑的，但是就是应这么做的事情。

保持事务简短将不会让您付出任何的代价。在事务中放入想要的内容，把不需要的内容拿出来，就这么简单。它的原理并不复杂——事务打开的时间越长，它触及的内容就越多(在事务中)，那么其他一些进程想要得到正在使用的一个或多个对象的可能性也就越大(减少了并发性)。如果要保持事务简短，那么就将最小化可能引起死锁的对象的数量，还将减少锁定对象的时间。原理就是如此简单。

把事务放在一个批处理中将最小化事务在网络中的往返时间，并且减少了完成事务以及释放锁的可能延时。

3. 尽可能使用最低的事务隔离级别

这个规则算不上基本的规则，它需要您思考一下。但如果不作认真思考也不会有大问题。

有一些不同的事务隔离级别可用。默认的为 READ COMMITTED。使用较低的隔离级别和较高的隔离级别相比，共享锁持续的时间更短，因此会减少锁的竞争。

4. 不要采用允许无限中断的事务

这可能是所有这些建议中最为常识性的规则——但是由于过去的习惯可能会违反这个规则。

用来防止丢失更新的一种方法(大型机时代)是获取锁，并且保持该锁直到完成操作为止。这显然是有问题的。

假设一下这个场景(这是一个现实生活的示例)：在服务部里有人喜欢使用更新模式(排他锁)而非显示模式(共享锁)来查看数据。他在查看一个工作单。这时他的朋友叫他一起吃午饭。他答应了并且去吃了一顿很长时间的(1～2 个小时)午饭。在这个员工吃午饭期间，任何对这一记录感兴趣的人都不能访问它。

情况还可能变得更加糟糕。在大型机时代，经常会用到队列的概念(它是非常有效的)。现在有人提交了这个工作单的打印任务(在队列中)。这个任务将会在队列中等待该记录锁清除。因为这是一个队列环境，公司中所有关于工作单的打印任务都将在第一个打印任务后堆积起来(将一直等待这个员工吃完午饭来清除锁)。

这是一个极端的示例，但是这里希望可以用它来说明问题。当开始执行某种开放式进程时，不要创建将一直占有资源的锁。通常，指的是用户交互，但它也可能是允许无限等待的任何进程。

14.6　本章小结

事务和锁都是 SQL Server 工作原理的基石，它们能使 SQL Server 中解决方案的开发最佳化。

使用事务，可以确保需要发生的事情作为一个单元发生，或者什么也不做。SQL Server 使用锁确保尽可能避免并发问题(不可能完全地避免，但是可以把它最小化)。使用事务和锁定，

www.TopSage.com

可以通过数据库行业中的 ACID 测试。如果事务是 ACID 的，那么它将具有下列特性。

- **原子性(Atomicity)**：事务要么全部执行，要么全部不执行。
- **一致性(Consistency)**：需要遵守所有的约束以及其他的数据完整性规则，并且完全地更新所有相关的对象(数据页、索引页)。
- **隔离性(Isolation)**：每一个事务都与任何其他事务完全地隔离。一个事务的动作不会受到另一个事务动作的干扰。
- **持久性(Durability)**：完成事务后，它的作用结果将永远存在于系统内。数据是“安全的”，这是指在诸如停电或其他非磁盘系统故障的情况下也不会导致发生数据只写入一半的情况。

简而言之，使用事务和锁，可以使死锁数量最小化、确保数据完整性，以及提高系统的整体效率。

第 15 章将讨论触发器。事实上，对于很多可能要用到触发器的情况来说，事务和回滚的概念将是使用的触发器的核心所在。

www.TopSage.com

第15章

触 发 器

触发器不仅很棒而且很灵活，是我们的朋友。但同时，它又是讨厌的和麻烦的，是我们的敌人。简而言之，我经常会被问及“应不应该使用触发器？”这样的问题。其实像 SQL 中的多数事物一样，答案是视情况而定。在奇妙的 SQL Server 的世界里，没有黑白分明的内容——触发器的情况更是极不明朗。

从初学者的角度来说(尽管您到现在为止已经不是一个初学者了)，在开始接触触发器之前要清楚自己在做什么，所以先不要急着行动，而是倾听、学习并判断触发器是否适合您。

本章将讨论触发器的各个方面——包括它坏的方面、好的方面以及不好不坏的方面，将介绍以下主要问题。

- 触发器的概念
- 使用触发器得到更灵活的参照完整性
- 使用触发器创建灵活的数据完整性规则
- 使用 INSTEAD OF 触发器来创建更灵活的可更新视图
- 触发器的其他常见用法
- 控制触发器的激活顺序
- 性能的考虑

学习完这些内容后，您应当对何时何地不能使用触发器这一决策的复杂性有所了解。也会了解触发器的强大性和灵活性。

最为重要的是，在本书内容的影响之下，希望您不会成为一个对触发器有偏见的极端主义者(我遇到的很多 SQL Server 使用者都是这样的人)，认为触发器会带来麻烦，不该使用它。同样也不会一味地站在另外一边，认为触发器可以解决所有问题。关于这个问题的正确答案是，触发器可以做很多事情，但也会带来很多问题。使用它的技巧在于在适当的时候使用，而不要在不适当的时候使用它们。

触发器的一些常见用途如下所示。

- 强制参照完整性：虽然本书建议尽可能地使用声明参照完整性(Declarative Referential Integrity，DRI)，但是 DRI 也有很多不能实现的操作(例如，跨数据库或服务器的参照完整性以及许多复杂的关系类型等)。

www.TopSage.com

- 创建审计跟踪(audit trails)，这意味写出的记录不仅跟踪大多数当前的数据，还包括对每个记录进行实际修改的历史数据。随着 SQL Server 2008 中的更改数据跟踪功能的出现，创建审计跟踪不再那么流行，但以前使用的就是触发器。
- 执行与 CHECK 约束类似的功能，但是跨表、跨数据库甚至是跨服务器使用。
- 用自己的语句代替用户的操作语句(通常用于启动复杂视图中的插入操作)。

此外，还会遇到可能极为少见的新情况(由于是新情况，所以只有经历时间的考验)，例如用来监控表中结构变化的数据定义语言(Data Definition Language，DDL)触发器。

以上列举的只是部分情况。这里不再多作赘述，下面先来看一下触发器的概念。

15.1 触发器的概念

触发器是一种特殊类型的存储过程，对特定事件作出响应。触发器有两种类型：数据定义语言(Data Definition Language，DDL)触发器和数据操纵语言(Data Manipulation Language，DML)触发器。

DDL 触发器在用户以某些方式(CREATE、ALTER、DROP 或相似的语句)对数据库结构进行修改时激活而作出响应。这对于某些安装非常关键(特别是高安全性的安装)，但是它们的使用范围却很狭窄。一般而言，只会在对数据库结构的改变或历史进行极为严格的审计时才会用到 DDL 触发器。有关 DDL 触发器的使用是非常高级的概念，本书在这里只是提醒您“了解它的存在”，下面将学习更为重要的触发器类型。

DML 触发器是一些附加在特定表或视图上的代码片段。与需要显式调用代码的存储过程不同，只要有附加触发器的事件在表中发生，触发器中的代码就会自动运行。实际上也不能显式地调用触发器——唯一的做法是在指定的表中执行所需的操作。

注意：

除了不能够显式地调用触发器，还可在存储过程中发现另外两个触发器所没有的内容：参数和返回码。

尽管触发器不带任何参数，但它们有一种机制来指出在什么记录上触发(本章将在后面作进一步的研究)。虽然可以使用 RETURN 关键字，但是不能返回一个特定的返回码(因为不能显式地调用触发器，返回码没有返回对象)。

可将触发器附加到什么事件呢？因为在 SQL 中可以使用 3 类动作查询，所以就有 3 种类型的触发器，另外加上混合搭配这些事件并对事件定时激活的混合触发器类型。

- INSERT 触发器
- DELETE 触发器
- UPDATE 触发器
- 以上任意类型的混合

注意：

值得注意的是，有时即使执行的动作是前面这些类型中的一种，触发器也不会激活。问题在于进行的操作是否在记录的活动中。例如，DELETE 语句是一个正常的记录活动，它会激活

www.TopSage.com

任何删除触发器，而 TRUNCATE TABLE 也有删除行的作用，但只是把表使用的空间释放而已——没有记录单个行删除操作，所以没有激活任何触发器。批量操作默认情况下不激活触发器(需显式告知批量操作激活触发器)。

创建触发器的语法看上去和所有其他 CREATE 语法示例十分相似， 只是它必须附加在一个表上——因为触发器不能独立存在：

看一下创建触发器的语法：

```
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> >
```

可以看到，其中有非常熟悉的 CREATE <object type> <object name>以及在其他许多对象中可以看到的执行语句——这里只是添加了 ON 子句来指出触发器将要附加的表，以及在何时和何种情况下激活这个触发器。

15.1.1 ON 子句

这部分只是对创建触发器所针对的对象进行命名。记住，如果触发器的类型是 AFTER 触发器(使用 FOR 或 AFTER 来声明触发器)，那么 ON 子句的目标就必须是一个表——AFTER 触发器不支持视图。

15.1.2 WITH ENCRYPTION 选项

该选项和在视图与存储过程中的工作方式一样。如果添加了这个选项，则可以确保没有人能够查看您的代码(甚至是您自己)。如果要构建商业发布软件，或者出于安全性的考虑不想让用户看到所修改或访问的数据，那么这个选项是非常有用的。很明显，如果想在以后重新创建触发器，那么需要保存创建触发器的代码副本，以便在其他地方使用。

和视图与存储过程一样，使用 WITH ENCRYPTION 选项需要记住的是，每次在触发器上使用 ALTER 语句时都必须重新应用该选项。如果使用 ALTER STATEMENT 语句但不包含 WITH ENCRYPTION 选项，那么触发器就不再被加密。

15.1.3 FOR|AFTER 子句与 INSTEAD OF 子句

除了要确定激活触发器(INSERT、UPDATE 和/或 DELETE)的查询类型以外，还要对触发器的激活时间作出选择。虽然人们经常考虑使用 FOR 触发器(也可以选择性地使用关键字 AFTER 来替换)，但是也可以使用 INSTEAD OF 触发器。对这两个触发器的选择将影响到是在修改数据之前还是之后进入触发器。无论是哪种情况，在将任何修改提交给数据库之前，您都将在触发器中。

www.TopSage.com

这可能让人有点困惑。这里换一种不同的方式，使用图来显示每种选择激活的位置(见图 15-1)。

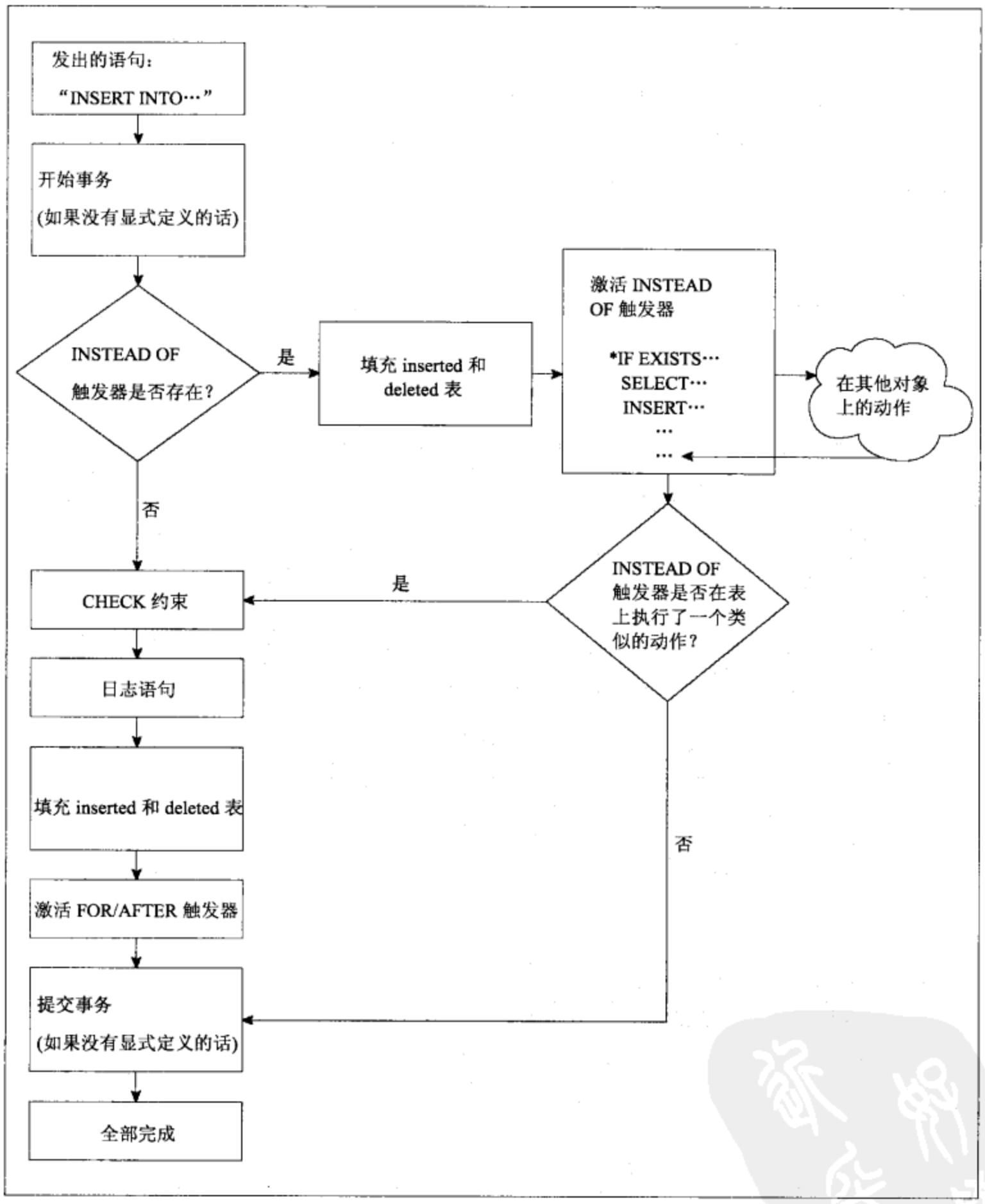


图 15-1

这里需要注意的是，无论进行何种选择，SQL Server 会将两个工作表放在一起——一个表保存插入记录的副本(称作 Inserted 表)，另一个表保存删除记录的副本(称作 Deleted 表)。稍后会看到这两张工作表的使用细节。现在需要认识到的是，如果使用 INSTEAD OF 触发器，工作

www.TopSage.com

表的创建发生在所有约束被检查之前，而如果使用 FOR 触发器，则这些表的创建是发生在约束被检查之后。使用 INSTEAD OF 触发器的关键在于可以在用户请求的任何地方运行您自己的代码。这意味着可以清除视图中任何不确定的插入问题，也意味着在约束被检查之前就可以采取措施清除违反约束的情况。

注意：

这听上去很不错，但是实际上非常复杂。这意味着需要预见每一种可能性。此外，这事实上等于给每一个查询添加了一次预处理(先于主代码运行的进程)，而这些查询可以任意修改表中的数据(这会影响到性能)。因此虽然听上去不错，但是这里把 INSTEAD OF 触发器归为相当高级的类别，它超出了本书的范围。

使用 FOR 和 AFTER 声明的触发器在行为上是一样的。它们与 INSTEAD OF 触发器最大的区别在于它们是在检查完约束之后建立工作表的。

FOR | AFTER

FOR(或者 AFTER)子句表明了期望触发器在何种动作类型下激活。当有 INSERT、UPDATE 或 DELETE 或三者混合操作时，都可以激活触发器。例如，FOR 子句可以如下所示：

```
FOR INSERT, DELETE
```

或者是：

```
FOR UPDATE, INSERT
```

或者是：

```
FOR DELETE
```

就像在 15.1.1 一节中所介绍的那样，使用 FOR 或 AFTER 子句声明的触发器只能附加在表上，而不允许附加在视图上(参见有关 INSTEAD OF 触发器的内容)。

1. INSERT 触发器

当有人向表中插入新的一行时，被标记为 FOR INSERT 的触发器的代码就会执行。对于插入的每一行来说，SQL Server 会创建一个新行的副本并把该副本插入到一个特殊表中，该表只在触发器的作用域内存在。该表被称为 Inserted 表，将在本章的学习过程中了解更多关于 Inserted 表的内容。需要了解的是，Inserted 表只在触发器激活时存在。在触发器开启之前或完成之后，都要认为该表是不存在的。

2. DELETE 触发器

它和 INSERT 触发器的工作方式相同，只是 Inserted 表是空的(毕竟是进行删除而非插入，所以对于 Inserted 表是没有记录的)。相反，每个被删除的记录的副本将插入到另一个表中，该表称为 Deleted 表。和 Inserted 表类似，该表只存在于触发器激活的时间内。

www.TopSage.com

3. UPDATE 触发器

除了有一点改变以外，UPDATE 触发器和前面的触发器是很类似的。当对表中现有的记录进行修改时，都会激活被声明为 FOR UPDATE 的触发器中的代码。唯一的改变是没有 UPDATED 表。SQL Server 认为每一行好像删除了现有记录，并插入了全新的记录。您大概也已经想到了，声明为 FOR UPDATE 的触发器并不是只包含了一个表，而是两个特殊的表，称为 Inserted 表和 Deleted 表。当然，这两个表的行数是完全相同的。

15.1.4 WITH APPEND 选项

WITH APPEND 选项并不常用，老实讲，您用到它的可能性很小；然而，为了以防万一，这里将简单介绍。WITH APPEND 选项只能应用于 6.5 兼容性模式中(可以使用 sp_dbcmptlevel 设置)——这在今天已是非常少见了(淘汰了有十年之久)。

SQL Server 6.5 和先前的版本不允许在单个表上使用同一类型的多个触发器。例如，如果已经声明了一个称为 trgCheck 的触发器在更新和插入时强制执行数据完整性，那么就不能创建另一个触发器来进行级联更新。一旦创建了更新(或插入、删除)触发器，那么就不能创建另一个同一动作类型的触发器。为解决这个问题，WITH APPEND 子句显式地告诉 SQL Server，即使在表上已经有了这种类型的触发器，还可以添加一个新的触发器。当有合适的触发动作(INSERT、UPDATE、DELETE)发生时，会同时激活两个触发器。这是一箭双雕的方法。

15.1.5 NOT FOR REPLICATION 选项

如果添加了该选项，会稍微地改变关于何时激活触发器的规则。在适当的位置使用这个选项，无论与复制相关的任务何时修改表，都不会激活触发器。通常，当修改了原始表并且不会再进行修改的时候会激活触发器(进行内务处理或级联等操作)。

15.1.6 AS 子句

和在存储过程中的使用完全相同，这正是触发器的实质所在。AS 关键字告诉 SQL Server，代码将要启动。从下面开始，将学习触发器的脚本部分的内容。

15.2 使用触发器实施数据完整性规则

虽然这不会成为首要的选择，但是触发器也同样可以执行和 CHECK 约束甚至是 DEFAULT 约束一样的功能。“使用触发器还是 CHECK 约束？”这个问题的答案是相当明显的：“看情况而定”。如果 CHECK 约束可以完成任务，那么可能 CHECK 约束是更受青睐的选择。但是，有时会出现 CHECK 约束不能完成任务的情况，或是 CHECK 过程中的某些固有内容使其显得不如触发器更为可取。想要使用触发器而非 CHECK 约束的例子包括：

- 业务规则需要引用单个表中的数据
- 业务规则需要检查更新的变化(即增量，更新前后的区别)
- 需要一个定制的错误消息

第 6 章的结尾处提供了一个汇总表，该表说明了何时使用何种类型的数据完整性机制。

www.TopSage.com

这些内容很肤浅。由于触发器高度灵活，因此确定何时使用触发器涉及到要权衡特定需求、触发器的系统开销以及根据需要必须采取的一些其他方法。

15.2.1 处理来自于其他表的需求

CHECK 约束快速而且有效，但是它们不是万能的。可能当您需要跨表验证数据时，它最大的缺点就会暴露出来。

为了说明这点，这里看一下 AdventureWorks2008 数据库中的 Sales.SalesOrderDetail 表和 Sales.SpecialOfferProduct 表。它们的关系如图 15-2 所示。

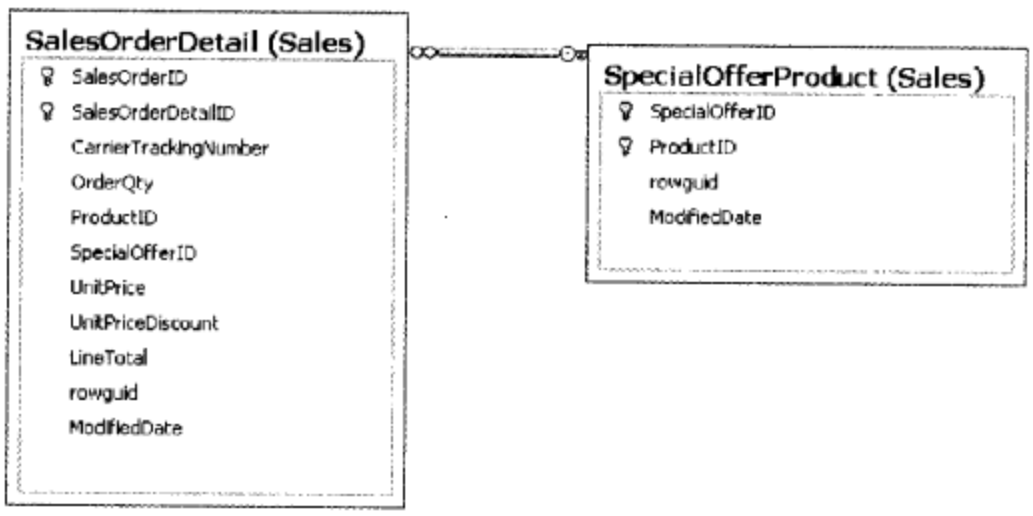


图 15-2

根据 DRI 约束，可以确定的是，除非在 SpecialOfferProduct 表(即使特价优惠记录为“无折扣”)中有匹配的 ProductID，否则将不能向 SalesOrderDetail 表中输入任何 SalesOrderDetail 项。然而，下面将研究的是比正常情况更复杂的情况。

存货部门抱怨说，客户支持人员不断发出已经标出断货的产品的订单。因此他们希望这种订单在进入系统之前就能予以拒绝。

这个问题不能使用 CHECK 约束来处理，因为提供断货状态的表(Product 表)的位置和在其上放置约束的表(SalesOrderDetail 表)的位置是分离的。但也不必为此担心——您完全可以告诉存货部门：“没有问题！”只要使用触发器即可。

```
CREATE TRIGGER Sales.SalesOrderDetailNotDiscontinued
   ON [Sales.SalesOrderDetail]
   FOR INSERT, UPDATE
AS
   IF EXISTS
      (
      SELECT 'True'
      FROM Inserted i
      JOIN Production.Product p
         ON i.ProductID = p.ProductID
      WHERE p.DiscontinuedData IS NOT NULL
      )
   BEGIN
      RAISERROR('Order Item is discontinued. Transaction Failed.',16,1)
      ROLLBACK TRAN
   END
```


www.TopSage.com

下面测试一下代码。首先，需要一两条在激活触发器时会失败的记录。遗憾的是，AdventureWorks2008 以前从未断货过某一产品——因此要做一些改变。

```
UPDATE Production.Product
SET DiscontinuedDate = '01-01-2008'
WHERE ProductID = 680
```

这样一来，便有了一种断货的产品，接下来需添加一个违反约束的订单细节项。注意这里选择了对一个现有的订单(43659)跟踪该细节项：

```
INSERT INTO Sales.SalesOrderDetail
VALUES
(43659, '4911-403C-98', 1, 680, 1, 1431.50,0.00, NEWID(), GETDATE())
```

结果得到了预期的拒绝信息：

```
消息 50000，级别 16，状态 1，过程 SalesOrderDetailNotDiscontinued，第 14 行
Order Item is discontinued. Transaction Failed.
消息 3609，级别 16，状态 1，第 1 行
事务在触发器中结束，批处理已中止。
```

记住，如果需要，还可以引发一个自定义的错误消息，而不是专门用于RAISERROR命令的消息。

注意：

如果进行了以上操作之后，又想尽量保持 AdventureWorks2008 接近其原始状态，那么不要忘记将 ProductID 680 的断货日期恢复为空。

15.2.2 使用触发器来检查更新的变化

有时，您可能并不关心过去的值和现在的值，只是想知道变化的量。虽然没有任何列或表给出这些信息，但是可以在触发器中使用 Inserted 表和 Deleted 表进行计算。

为了验证这点，下面来看一下 Production.ProductInventory 表。ProductInventory 表有一个称为 Quantity 的列。最近，一些产品出现了抢购热潮，AdventureWorks2008 也已经售完了一些产品。由于从长远角度看，AdventureWorks2008 需要与更多的客户保持商业往来，因此它决定对产品建立一个限量供应系统。存货部门要求对任一产品都不能发出订货量超过其一半库存的订单。

为将此付诸实施，可同时使用 Inserted 表和 Deleted 表。

```
CREATE TRIGGER Production.ProductIsRationed
   ON Production.ProductInventory
   FOR UPDATE
AS
IF EXISTS
   (
   SELECT 'True'
   FROM Inserted i
   JOIN Deleted d
```


www.TopSage.com

```
        ON i.ProductID = d.ProductID
      AND i.LocationID = d.LocationID
    WHERE (d.Quantity - i.Quantity) > d.Quantity / 2
      AND d.Quantity - i.Quantity > 0
    )
BEGIN
    RAISERROR('Cannot reduce stock by more than 50%% at once.',16,1)
    ROLLBACK TRAN
END
```

在测试之前，先来分析一下此处进行的工作。

首先，这里使用了本章一直使用的 IF EXISTS。只有在某些情况能够满足繁琐和苛刻的测试条件时，才会进行回滚。

然后把 Inserted 表和 Deleted 表连接起来——这样可以比较两个表。

WHERE 子句可能是会让人产生困惑的地方。它的第一行代码相当直观，是实现业务要求的语句；如果对 Quantity 列的更新操作使得订货量超出先前库存量的一半，那么将满足 EXISTS 标准，并运行条件代码(拒绝该事务)。

但是下一行代码就不是那么直观了。和编程中的所有操作一样，需要考虑语句之外的问题，并考虑到其他后果。该需求实际只表示库存的减少——当然不希望对库存的数量作限制——所以只关心更新后的库存量比更新前的库存量少的情况。条件“> 0”解决了库存中可能仅剩一种产品的情况。在这种情况下，希望将最后一件产品全部售空，但是由于 1 - 0 比 0.5 大，因此如果没有“> 0”这一附加条件，触发器就会拒绝执行。

如果这两个条件都得到满足(超过了 50%并且库存是减少而不是增加)，那么将引发错误。注意在 RAISERROR 中使用了两个%符号，而不是一个。记住，一个%符号是作为参数的占位符，所以%符号本身将不会出现在错误消息中。通过在一行中放置两个%符号，SQL Server 就能知道实际需要输出一个百分号。

现在看一下它的工作原理。这里选择一条记录，尝试做一下更新，将库存量减少 50%以上：

```
UPDATE Production.ProductInventory
SET Quantity = 1 -- Was 408 if you want to set it back
WHERE ProductID = 1
 AND LocationID = 1
```

这里选择了“Adjustable Race”来做试验，也可以选择任何的 ProductID，只要把值设为少于先前值的 50%。这样便会得到一个预期的错误：

```
消息 50000，级别 16，状态 1，过程 ProductIsRationed，第 16 行
Cannot reduce stock by more than 50% at once.
消息 3609，级别 16，状态 1，第 1 行
事务在触发器中结束，批处理已中止。
```

注意本应该对现有的 Quantity 数量引用实际的订单数量，在 SalesOrderDetail 表中实现它，但是会遇到一些问题。

www.TopSage.com

- **更新有所改变**：创建 SalesOrderDetail 记录的过程是在该记录之前还是之后更新 ProductInventory 呢？这会根据如何利用 ProductInventory 表中的 Quantity 值来计算事务的结果而有所区别。
- **对 SalesOrderDetails 表外部库存的更新不会受到影响**：这些更新仍可以将库存量减少过半(这在很多情况下是好事，但是也需要对它进行考虑)。

15.2.3 将触发器用于自定义错误消息

本书已经在一些示例中接触了这个内容，请记住，当想要对传给用户或客户端应用程序的错误消息或错误号进行控制时，使用触发器是很方便的。

例如，如果使用 CHECK 约束，只能得到标准的 547 错误，并且没有详尽的解释。通常，对于想知道具体错误的用户来说，这是无用的信息——确实，客户端应用程序经常因为没有足够的信息而不能代表用户作出智能和有帮助的响应。

简而言之，当已经具备了数据完整性，但是没有足够信息进行处理的时候，可以创建触发器。

注意：

尽管传递自定义错误代码很有用，但 SQL Server 中对自定义错误消息的需求还是相对较少。为什么不传递自定义错误消息呢？原因在于某些用户认为在自定义错误消息之上有一个应用程序层，并且可能需要更多有关错误的上下文信息，因此特定于 SQL-Server 的文本就无法充分发挥作用。而这时如果使用特定的错误代码，对于应用程序则有很大帮助，有助于确定确切发生的事件以及应用正确的客户端错误处理代码。

15.3 触发器的其他常见用途

除了直接的数据完整性用途以外，触发器还有一些其他用途，可以说是相当广泛，这里仅列举以下一些常见用途。

- 更新汇总信息
- 为报表提供非规范化表
- 设置条件标志

可以看到，触发器的用途太过广泛——触发器其实是和具体情况以及特殊的系统需求相关的。

15.4 触发器的其他问题

本章前面已经了解了有关触发器的大部分内容，但是如果您觉得已经完成了触发器的学习，那么请再思考一下。本章前面已经提到过，触发器有许多相关问题需要考虑。下面的章节将指出需要考虑的最重要的问题，还会提供一些关于触发器附加功能和可能性的信息。

www.TopSage.com

15.4.1 触发器可以嵌套

嵌套的触发器是指那些不是由发出的语句直接激活的，而是由另一个触发器发出的语句激活的触发器。

这实际上会引起一连串的事件——一个触发器激活另一个触发器，而另一个触发器又激活其他触发器，如此等等。触发器可以激活的深度取决于以下几个因素。

- 嵌套的触发器是否已在系统中打开(这是系统级的而不是数据库级的选项；可以使用 Management Studio 或 sp_configure 来设置，默认为打开的)。
- 是否有嵌套深度不超过 32 层的限制。
- 触发器是否已经被激活。触发器默认为每个触发器事务只能被激活一次。一旦被激活，则该触发器会忽略其他任何调用，将这些调用作为相同触发器动作的一部分。一旦执行一条全新的语句(即使在同一个完整的事务中)，处理过程就会重新开始。

大多数情况下，您希望嵌套触发器(这是默认的做法)，但是如果进入触发器激活触发器这样的循环，那么就要考虑一下下面将会发生什么。如果两次回到了同一个表，那么触发器就不会二次激活，您认为重要的事情也不会发生；例如，违反数据完整性的操作可能会顺利通过。同样值得注意的是，如果在嵌套链中的任何地方进行 ROLLBACK 操作，那么整条链都将回滚。换句话说，整个嵌套触发器链就像一个事务一样。

15.4.2 触发器可以递归

什么是递归触发器？如果某触发器所做的事情最终激活了自身，那么该触发器就是递归的。可以直接触发(通过在设置了触发器的表上进行动作查询来完成)，也可以间接触发(通过嵌套过程)。

递归触发器比较少见。实际上，默认情况下，递归触发器是关闭的。但是，这是一种处理上述情况(也就是嵌套触发器并且希望发生二次更新)的一种方式。递归和嵌套不同，递归是数据库级的选项，可以使用 sp_dboption 系统存储过程来设置。

递归触发器的风险在于可能会陷入某种非预设的循环之中。这样便需要确保在必要的时候可以通过递归检查的形式来停止这一过程。

15.4.3 触发器不能防止体系结构的修改

这是个典型的好坏参半的情况。

触发器有助于更容易地修改体系结构。事实上，通常在开发周期的早期(此时更有可能对数据库的设计进行修改)使用触发器实施参照完整性，而在后期，也就是要进入生产环境时将其改为 DRI。

在想要删除表并且使用 DRI 重新创建表的时候，必须在删除该表之前先删除所有的约束。这会造成很多麻烦，因为需要删除多个约束、进行修改，并再次添加约束。要确保那些应该删除的内容都已删除为了使修改的脚本可以运行，这要费上一大番功夫。同样，确保一切都回到需要的状态也很费周折。但触发器可以做到所有这一切，因为在实际运行之前，触发器不关心任何改变。但是难也难在这里——实际运行时该怎么办。可以看到，这意味着可能需要修改体系结构甚至是在无意识的情况下破坏一些触发器。直到这些触发器第一次试图解决存在问题的

www.TopSage.com

对象时，您才会发现这种做法的错误之处。到了那时，您才会了解您要做的事情以及这样做的原因。

这两方面各有争议——不管使用的是哪种方法，要记住这些争议。

15.4.4 可以在不删除的情况下关闭触发器

有时，像 CHECK 约束那样，您想要关闭完整性的功能以便于执行一些违反约束但是有效的动作(最常见的可能就是导入数据)。

这么做的另一个常见理由为：要执行某种批量插入(还是导入)，但是已经百分百确认插入的数据有效。在这种情况下，可能要关闭触发器，从而消除系统开销并提高插入数据的速度。

可以使用 ALTER TABLE 语句来关闭触发器，语法如下：

```
ALTER TABLE <table name>
 <ENABLE|DISABLE> TRIGGER <ALL|<trigger name>>
```

您可能会想到，这里最需要注意的是，“不要忘了重新激活触发器”。

最后要做的事情是：如果关闭触发器是为了导入批量数据，那么本书强烈建议踢出所有用户并进入单用户模式、dbo-only 模式，或同时进入这两种模式。这样一来，当关闭触发器时，就能确保万无一失。

15.4.5 触发器的激活顺序

在 SQL Server 以往的版本中(7.0 以及之前的版本)，不能控制激活顺序。实际上，可能您会想到在 7.0 版本之前只有一种特殊类型的触发器(INSERT、UPDATE、DELETE)，所以激活顺序一直是个颇有争议的问题。SQL Server 随后发布的版本为触发器顺序提供了一些有限的控制。对于任何给定的表(不是视图，因为只有 AFTER 触发器才可以指定激活顺序，而视图只接受 INSTEAD OF 触发器)，可以选择某个触发器(并且是唯一的一个)优先激活。同样，可以选择一个触发器(唯一的一个)最后激活。其他所有的触发器之间没有什么优先激活顺序——也就是说，除了能保证触发器是在 FIRST 触发器(如果有的话)完成之后且在 LAST 触发器(如果有的话)开始之前激活以外，不能保证激活顺序为 NONE 的触发器是以何种顺序激活。见图 15-3。

FIRST 和 LAST 触发器的创建和其他任何触发器的创建相同。在已经创建触发器之后使用一个特殊的系统存储过程 sp_settriggerorder 来声明激活顺序的优先选项。

sp_settriggerorder 的语法如下：

```
sp_settriggerorder[@triggername =] '<trigger name>',
   [@order =] '{FIRST|LAST|NONE}',
   [@stmttype =] '{INSERT|UPDATE|DELETE}'
   [, [@namespace =] {'DATABASE' | 'SERVER' | NULL} ]
```

这里，对于任何特殊操作(INSERT、UPDATE、DELETE)来说，只能有唯一的 FIRST 触发器。同样，对于任何特殊操作来说，也只能有唯一的 LAST 触发器。其他触发器的数量可以看作是 NONE——也就是说，没有特殊激活顺序的触发器的数量是没有限制的。

www.TopSage.com

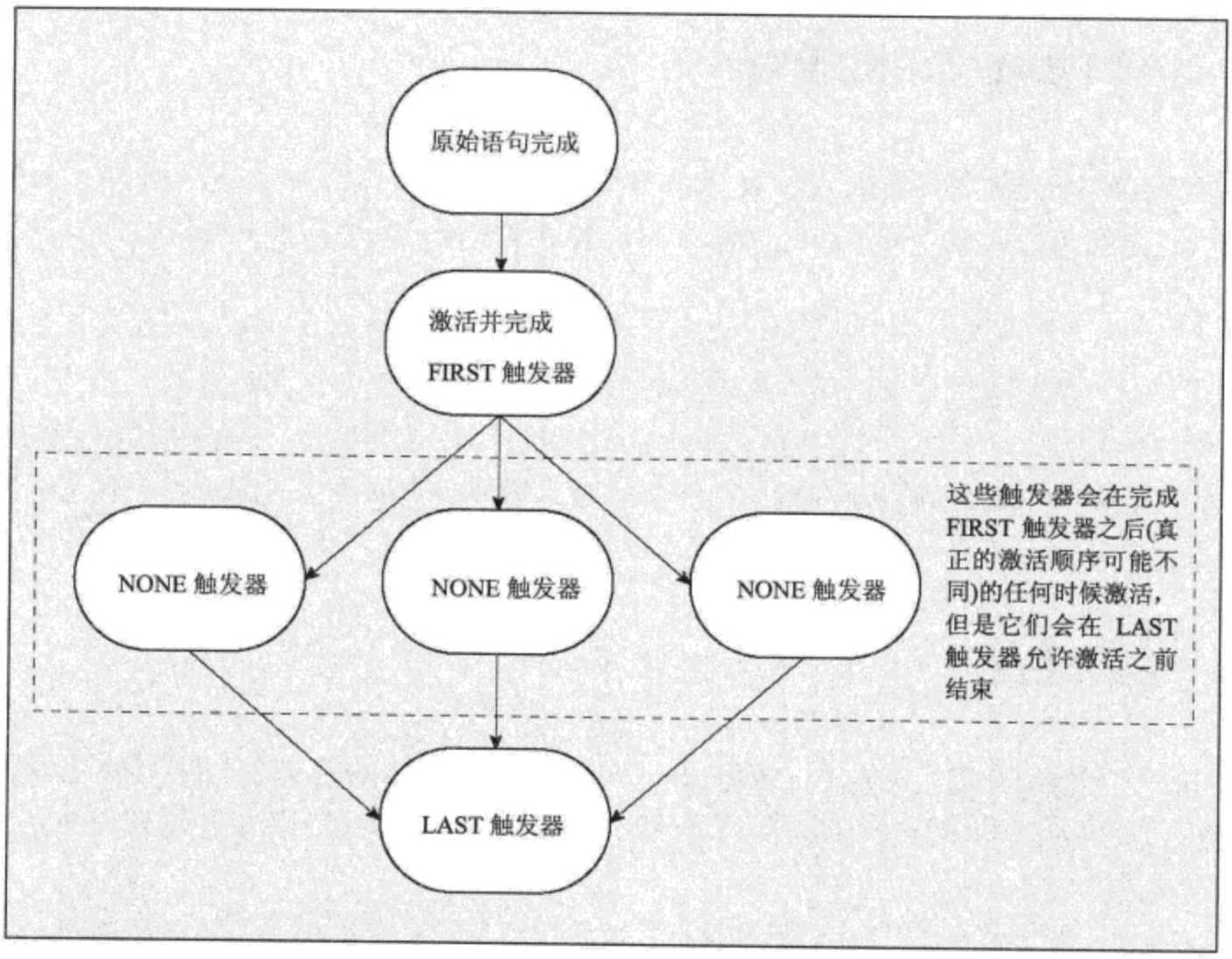


图 15-3

所以这里的问题在于："既然如此，为什么还要注意激活顺序？"通常情况下完全不需要在意这个问题。但有时，这可能是重要的逻辑问题或是仅涉及性能原因。下面看一下有关这一内容的细节。

1. 出于逻辑原因而控制激活顺序

为什么要在激活一个触发器之前去激活另一个触发器？最常见的理由是第一个触发器是后面触发器的基础或是前面的触发器使后面的触发器有效。分离的触发器也允许代码的某个部分失效(请查看前面的 NO CHECK 操作)，而代码的其他部分继续运行。其不利面在于如果将代码分离，放到分离的触发器中，会丢失代码在原本的一个触发器中所拥有的逻辑步进顺序。

通过至少对激活顺序的基础层面的控制，就可以获得两全的结果了——既可以有逻辑地分离触发器，又保证了代码先后运行的必要优先顺序。

2. 出于性能原因而控制激活顺序

在性能方面，FIRST 触发器是唯一起关键作用的触发器。如果有多个触发器，但是其中只有一个触发器可能会产生回滚(例如，它会强制执行约束所不能处理的复杂的数据完整性规则)，那么就需要考虑将这个触发器标记为 FIRST 触发器。这确保在事务中执行更多操作之前已经考虑到最有可能发生回滚的因素。在检测到回滚之前执行的操作越多，需要回滚的内容也越多。因此在执行额外操作前需先确定回滚发生的最大可能性。

www.TopSage.com

15.5 INSTEAD OF 触发器

INSTEAD OF 触发器是 SQL Server 2000 新增的触发器，它也是 SQL Server 中最为复杂的功能之一。虽然这不在初学范围内，但是本书认为初学者还是需要了解其作用，所以这里稍做介绍。

从本质上说，INSTEAD OF 触发器可用作一个对试图对表或视图进行操作的用户进行拦截的代码块。可以选择继续并执行用户请求的动作，也可以执行完全不同的动作。

像常规的触发器一样，INSTEAD OF 触发器也有 3 种：INSERT、UPDATE 和 DELETE。这 3 种触发器最为常见的用途是相同的——在处理基于多个表的视图时解析哪个表实际发生更改的多义性。

15.6 性能考虑

关于触发器的好与坏，人们往往想得出一些绝对的结果。最糟糕的要数那些纯粹论者——他们热爱这个理论，就想用触发器处理所有问题，或者因为发现了触发器的灵活性而想把它们用于所有对象中。

正如本章一开始就提到的，本书的观点是要在恰当的时间使用触发器。如果这听上去有些模棱两可——那就对了！编程问题很少是黑白分明的，数据库更是如此。但是这里仍会列出一些事实供您思考。

15.6.1 触发器的被动性

这里的意思是指触发器发生在事务之后。当激活触发器时，整个查询已经运行并且事务也已经被记录到日志中(但未提交，只是记录到激活触发器的语句点)。这意味着如果触发器需要回滚，那么必须撤销已经做的所有工作。请慎重对待并记住这点。这样带来的影响有多大很大程度上取决于查询的大小。

您可能对此不以为意。这里可以将触发器与约束的概念作一下比较，约束是主动的——也就是约束在语句真正执行前发生。这意味着约束会检测可能失败的操作，并且在进程的前期就予以阻止。所以约束通常会运行得快一些——在更为复杂的查询中速度更快。注意，只有在发生回滚时，这个额外的提速才很明显。

最后的分析结果是，如果正在处理少量回滚，而且/或者受影响语句的复杂性较低，执行时间较短，那么触发器和约束之间没有太大的区别。可能有一些，但不是很多。可是，如果回滚的数量无法预知或者很多的时候，则需要坚持使用约束(本书也建议使用约束，除非有特殊的理由)。

15.6.2 触发器与激活的进程之间不存在并发问题

您可能已经注意到在本章中经常会使用 ROLLBACK 语句，即使没有发出 BEGIN TRAN 命令。这是因为触发器总是隐式地与激活触发器的语句作为一个事务处理。

如果激活语句不是显式事务(含有 BEGIN TRAN 事务)的一部分，那么该语句仍然是其自身

www.TopSage.com

的单语句事务的一部分。无论何种情况，触发器内部发出的 ROLLBACK TRAN 仍然会回滚整个事务。

这种同属一个事务的另一个结果是触发器继承了它们所属事务上已打开的锁。这意味着不需要做任何特殊的处理来避免碰到事务中其他语句创建的锁。在事务的作用域内可以自由访问，并且可以发现数据库基于事务中先前的语句所作的修改。

15.6.3 使用 IF UPDATE()和 COLUMNS_UPDATED()

在 UPDATE 触发器中，可以通过检查感兴趣的列是否已被修改来限制在触发器中执行的代码总量。为了实现这一点，可以使用 UPDATE()或 COLUMNS_UPDATE()函数。下面将分别介绍这两个函数。

1. UPDATE()函数

UPDATE()函数只在触发器的作用域内适用。它唯一的目的是提供一个布尔值(真/假)，来说明某个特殊列是否已经更新。使用这个函数可以决定一个特定的代码块是否需要运行——例如，该代码只在特定列更新时才运行。

这里通过修改一个以前的触发器来运行一个简短的示例：

```
ALTER TRIGGER Production.ProductIsRationed
   ON Production.ProductInventory
   FOR UPDATE
AS
   IF UPDATE(Quantity)
   BEGIN
     IF EXISTS
       (
       SELECT 'True'
       FROM Inserted i
       JOIN Deleted d
         ON i.ProductID = d.ProductID
       AND i.LocationID = d.LocationID
       WHERE (d.Quantity - i.Quantity) > d.Quantity / 2
         AND d.Quantity > 0
       )
     BEGIN
       RAISERROR('Cannot reduce stock by more than 50%% at once.',16,1)
       ROLLBACK TRAN
     END
END
```

通过修改，限制了剩下的代码只在 Quantity 列(我们所关注的列)发生改变时运行。用户可以修改任何其他非关注列的值。这意味着可以执行更少的代码行，而触发器的性能也会比先前的版本好。

2. COLUMNS_UPDATED()函数

这个函数和 UPDATE()的运行方式有点不同，但目的相同。COLUMNS_UPDATED()函数可

www.TopSage.com

用于一次检查多列。为实现这一点，该函数使用了位掩码，位掩码将 varbinary 数据的一个或多个字节中的单个位与表中的单个列相关联。结果如图 15-4 所示。

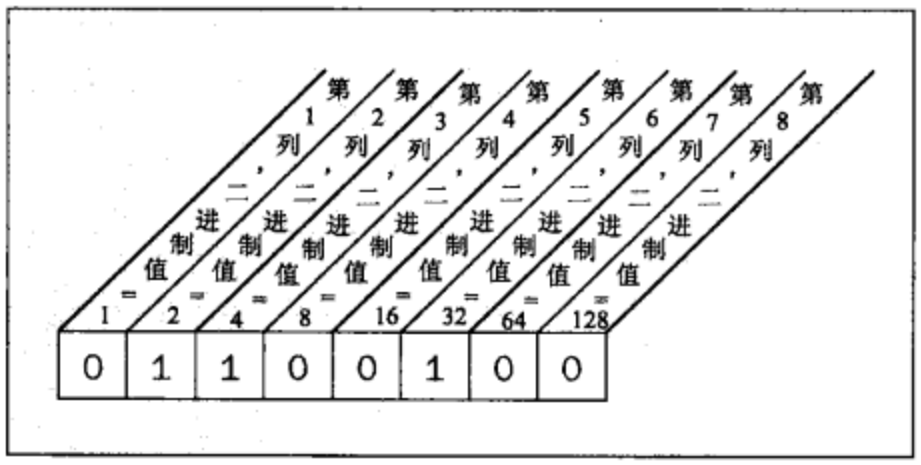


图 15-4

在这种情况下，数据的单个字节说明了第 2、第 3 以及第 6 列已经更新，而其他的列没有更新。

如果超过 8 列，SQL Server 就会在右边添加另一个字节并且继续计数(如图 15-5 所示)。

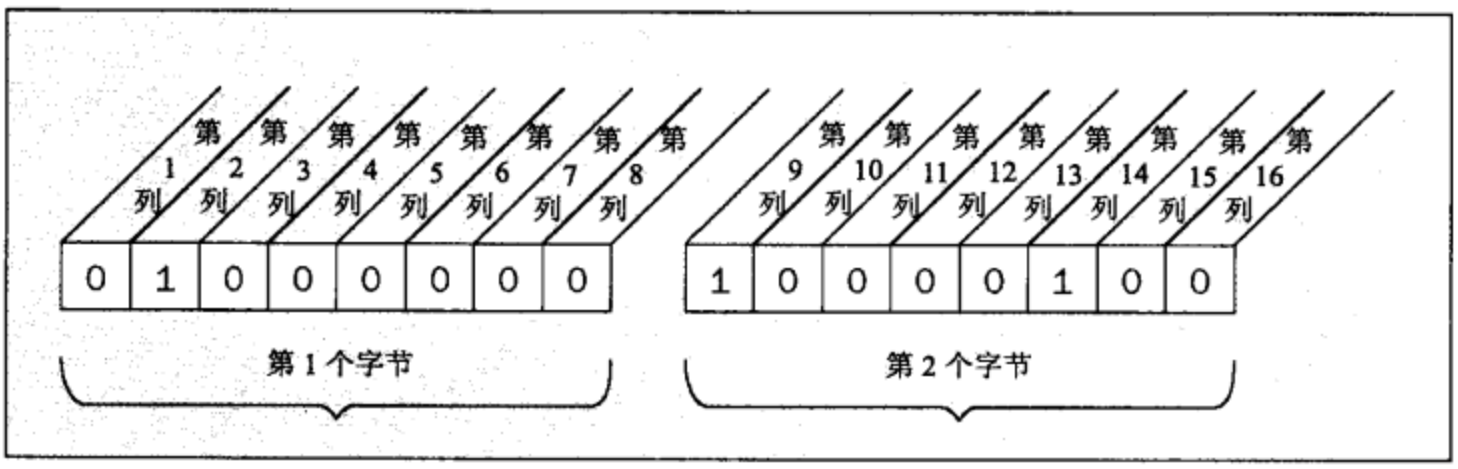


图 15-5

这次是更新第 2、第 9 以及第 14 列。

您可能会说“不错，但是如何使用它们呢？”要回答这个问题，首先需介绍一下布尔代数的内容。

使用这些信息意味着需要将所有字节的二进制值累加起来，并要知道最左面的数字是最不重要的。所以，如果需要考虑比较 2、5、7 列，那么需要把每一位的二进制值累加起来：2+16+64。然后需要使用位操作符对列的二进制值的总和与位掩码进行比较。

- | 表示按位或
- & 表示按位与
- ^ 表示按位异或

虽然这些概念都是正确的，但还是不够清楚。因此这里使用一些示例来说明。

假设对含有 5 个列的表进行更新。如果更新了第 1、第 3 和第 5 列，那么 COLUMNS_UPDATED 使用的位掩码将是 10101000，1+4+16=21。这里可以使用下列表达式。

- 使用 COLUMNS_UPDATED() > 0 来确定是否更新了列。

www.TopSage.com

- 使用 COLUMNS_UPDATED() ^ 21 = 0 来确定是否更新了所有指定的列(本例中为 1、3、5 列)，而没有更新其他列。
- 使用 COLUMNS_UPDATED() & 21 = 21 来确定是否更新了所有指定的列，而其他列的状态不重要。
- 使用 COLUMNS_UPDATED | 21 != 21 来确定是否更新了除感兴趣的列以外的列。

注意：

需要了解的是，布尔代数是比较难的——对很多人来说并不是很容易理解的概念，所以需要仔细地检验并再三测试！

15.6.4 保持触发器短小精悍

这一点很显而易见，但还是要强调一下。

经常会在存储过程和触发器中看到一些冗长而笨拙的代码。不免让人猜测这些代码可能是仓促编写的，或者人们认为他们使用的方法比较迅速，所以代码冗长一点也无大碍。

请记住触发器与激发触发器的语句被视为同一事务处理。这意味着语句直到触发器完成后才算完成。试想一下——如果在触发器中编写运行时间很长的代码，那么这将意味着您所创建的每一条激活触发器的代码都会因此运行很长时间。当试图找出代码运行时间过长的原因时，这就会造成很大的烦恼。您写的存储过程看上去效率很高，但是实际运行起来很糟糕。您用了数周的时间去寻找问题，但永远也想不到问题是出在看上去毫无问题的存储过程上——因为存储过程激活的触发器有问题。

15.6.5 选择索引时不要忘记触发器

这是另一个常见的错误。在检查所有的存储过程和视图以找到最好的混合索引时，完全忘记了已经在触发器中运行了重要的代码。

这里的概念与 15.6.4 节提到的一样——长时间运行的查询导致长时间运行的语句，这样依次导致所有操作都长时间运行。所以进行优化的时候不要忘了触发器。

15.6.6 不要在触发器中进行回滚

这部分内容比较难理解，因为回滚是触发器经常需要用到的一个重要部分。

只要记住，AFTER 触发器(和普通类型的触发器有很大区别)在所有工作已经完成后发生——这意味着回滚的代价是昂贵的。这时，DRI 就获得了其所有的性能优势。如果在触发器中使用很多 ROLLBACK TRAN 语句，那么请确保在执行激活触发器的语句前预先进行错误检查。也就是说，因为 SQL Server 在这种情况下不是主动的，所以您需要主动，事先检查错误，而不是等待回滚。

15.7 删除触发器

删除触发器和到目前为止所做的所有删除操作都略有不同。和表一样，其问题在于触发器的名称被限定在模式级别。这意味着一个触发器可以有两个名称相同的对象，只要放置触发器

www.TopSage.com

的对象与触发器另一个同名的对象位于不同的模式中。重申一次，触发器是以其所处的模式命名的，而不是以触发器所关联的对象命名。删除的语法如下：

```
DROP TRIGGER [<schema>.]<trigger name>
```

除了模式问题，删除触发器就和删除其他对象一样简单。

15.8 调试触发器

调试触发器的方式与其他调试操作基本上是一样的。唯一问题在于不能像调用脚本或存储过程那样随心所欲地调用触发器。

所幸的是，开发人员很聪明，想到了如何对触发器进行调试——只需调试一个激活触发器的语句即可。

注意：

在 SQL Server 2008 发布以前，2005 及其更早版本不具备直接对脚本进行调试的能力。这意味着如果想要调试触发器，就需创建一个使触发器能被调试的存储过程——令人欣慰的是现在已经不需要这么做了。

由于在第 12 章中已经相当透彻地介绍了调试的基本内容，这里就直接进入有关触发器调试的操作。所需做的只是运行一小段代码测试本章中一直使用的触发器。然后可以逐语句运行脚本，观察调试器是如何逐行运行的。

```
BEGIN TRAN
 -- This one should work
 UPDATE Production.ProductInventory
 SET Quantity = 400 -- Was 408 if you want to set it back
 WHERE ProductID = 1
   AND LocationID = 1

 -- This one shouldn't
 UPDATE Production.ProductInventory
 SET Quantity = 1 -- Was 408 if you want to set it back
 WHERE ProductID = 1
   AND LocationID = 1
IF @@TRANCOUNT > 0
 ROLLBACK TRAN
```

当该脚本出现在 Management Studio 的“查询”窗口中时，只要按下 F11 键即可逐语句运行脚本(或者设置一个断点，然后在启动调试器的几个选项中任选一个)。调试当然是从 BEGIN TRAN 语句开始。然后继续逐语句运行(使用 F11 键)，直到执行第一个 UPDATE 语句，注意最后终止于触发器中。图 15-6 显示了首次进入触发器时的调用堆栈。

www.TopSage.com

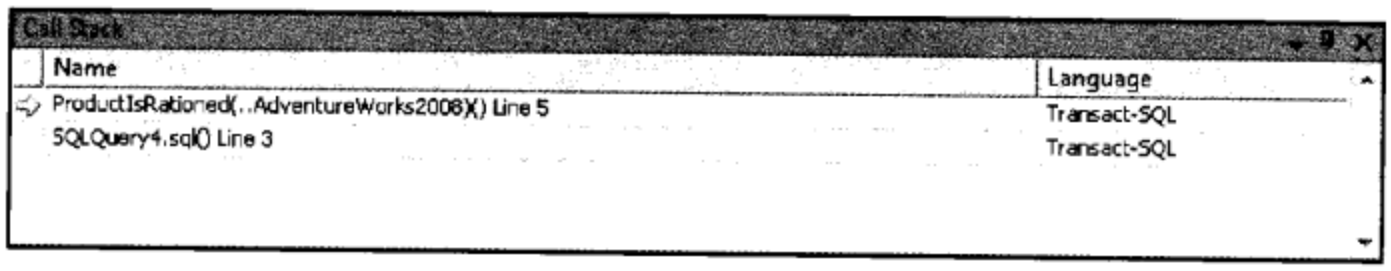


图 15-6

现在跳出第一个 UPDATE，逐语句运行到第二个 UPDATE。由于第一个 UPDATE 语句并不满足触发器中的违反测试条件，所以几乎是立即退出；但对于第二个 UPDATE 语句，会发现激活了取消事务的代码。

简而言之，在本例中，我们已经能够运用一个包含了两个语句的脚本对触发器中两种可能的代码路径进行测试。显然您需要用更为复杂的脚本处理更为复杂的触发器中可能的代码路径，但是除此之外，调试要求与测试 T-SOL 中其他任何代码都一样。

15.9 本章小结

触发器是非常强大的工具，给数据完整性以及系统的整体操作增添了巨大的灵活性。尽管如此，也不要小看触发器。如果使用触发器作适当的数据汇总，那么它将会极大地提高系统的性能，但同时也会成为您的困扰。因为很难对它们进行调试(即使已经有了调试器)，一个编写得很糟糕的触发器不但会影响触发器本身，还会影响激活该触发器的所有语句。

请记住，除了那些可满足大部分触发器需求的触发器模板外，有很多工具可以自动生成满足特定要求的触发器。

www.TopSage.com

第16章

初 识 XML

到目前为止，已经学习了大量有关 SQL Server 结构性的内容，现在将开始外围知识的学习。也就是说，将要开始学习在处理关系数据库系统时常见考虑之外的知识。这里并不是说后面将要介绍关系数据库系统以外的内容，而是说这些知识对于构成一个功能性的 SQL Server 来说不是必须的。毕竟，SQL Server 包含的内容太多，很难把所有的内容都压缩到一本书中进行介绍。

本章将首先介绍一下 XML 的背景——它已逐渐成为 SQL Server 必不可缺的一部分。接着将介绍 SQL Server 中用以支持 XML 的一些功能。这里的问题在于 XML 是完全独立的内容，XML 和到目前为止所接触的关系系统截然不同。那么 SQL Server 为什么还要包含这么多的功能来支持它呢？简单地说，XML 可能是自数据仓库出现以来对数据而言最为重要的事物。

目前 XML 已经出现好几年了，尽管人们经常谈论它，但 XML 的实际应用没有达到应有的程度。自 20 世纪 90 年代末开始，XML 越来越广泛地用于表示数据种子以及适当尺寸的数据文档。XML 提供了数据自描述的方法——也就是说，可以连同 XML 文档一起定义类型和验证信息，这样不论用户是谁(即使他们连如何连接 SQL Server 都不知道)，他们都能够理解该数据的规则。

XML 通常不是存储数据的好方法，但绝对是一种使数据有用的妙招——同样地，利用 XML 的方法也在不断地发展。

综上所述，本章将学习以下的内容。

- XML 的概念
- 其他与 XML 有紧密联系的技术

注意：

前面提到过 XML 通常并不是存储数据的好办法，但也有例外情况。XML 用作数据存储可能是出于归档的目的。XML 压缩的能力非常好，它使用一种非常开放的格式，这种格式至少在未来很多年内也仍然是很容易理解的——并非永远。这里比较一下将 XML 与 SQL Server 2008 的备份功能。十年后，当您需要还原一些旧的数据来回顾归档信息时，很可能没有适当的 SQL Server 安装程序能处理这些旧的备份文件，但可能会有解压缩(假设使用主流的压缩工具，如 ZIP)以及阅读数据的工具。对于这样的“深层”归档来说，XML 很有用。

www.TopSage.com

16.1　XML 基础

有关 XML 的书籍已经很多(例如，清华大学出版社引进并出版的《XML 高级编程》，由 Evjen 等人编写)。考虑到本书的内容已经很多，所以此处不会再添加过多有关 XML 本身的信息，而是假定您已经了解了 XML 的一些内容。但是，也应意识到即使 XML 近些年来成为了主流，也还有很多数据库人员认为 XML“只是一项 Web 技术而已”，因此，他们没有花时间学过这方面的知识——这实在是大错特错。

首要的一点，XML 是一项信息技术。XML 不是 Web 特有的技术。然而，由于以下原因，人们可能会把 XML 误认为是 Web 特有的技术(通常是那些不理解 XML 的人们)。

- XML 是一种标记语言，对于无经验的人而言，XML 看上去和 HTML 很相似。
- XML 通常很容易转换为 HTML。因此人们广泛使用 XML 保存页面的信息部分并根据条件最终转换成 HTML——可以基于条件(例如哪种浏览器正在请求信息)进行单独的转换。
- Microsoft 的 Internet Explorer 是第一个广泛用于的支持 XML 的产品。
- Internet 经常用作信息交换的方式，XML 对此非常适合。

和 HTML 一样，XML 是基于文本的标记语言。它们的确都源于同一种语言，称为 SGML。SGML 的历史比 Internet(指现在的 Internet)更为久远，并且常用于印刷业或政府相关的文档中。SGML 中的 S 并不表示简单(SGML 全称为 standard generalized markup language，标准通用标记语言)——SGML 并不直观，也不易学(我只能读懂所看过的 SGML 文档的 35%。然而阅读时的痛苦却能达到 100%)。因此从另一方面讲，XML 还算是易于理解。

所以，您可能会问：“从哪里能得到 XML 标记的列表呢？”答案是得不到这样的列表——至少和您所想的答案不一样。XML 只有很少的一些标记是其本身语言的一部分。更多是提供对标记进行自定义的方法和使用由其他人(例如本章前面提及的行业组)定义的标记的方法。XML 的主要性能是灵活性——允许使用 XML 架构文档或原有的文档类型定义(Document Type Definition，DTD)来为 XML 设定自己的规则。

XML 文档之所以规则极少，恰恰是因为它是 XML。这里的关键在于 XML 必须是所谓的格式良好的。后面将简要介绍格式良好的概念。仅仅因为 XML 文档满足格式良好的标准并不能说明它有效。有效的 XML 文档不仅是格式良好的，而且不能违反文档引用 XML 架构或 DTD 所设定的任何限制。本章将在后面简单地介绍 DTD 和 XML 架构的内容。

XML 也可以进行转换。这里的意思是指可以相对容易地把 XML 转换成完全不同的 XML 表示形式甚至是非 XML 格式。最为常见的应用是把 XML 转换成为 HTML 以在 Web 上呈现。这种转换需要提供了比较 HTML 和 XML 的机会。简单地说，XML 和信息有关，而 HTML 和显示有关。

XML 中存储的信息通过元素(element)和属性(attribute)来表示。元素通常通过打开和关闭标记(也有例外情况，后面将会作介绍)来创建，并且用一个区分大小写的名称来标识(不允许有空格)。属性是进一步描述元素的项，并且嵌在元素的开始标记内。属性值必须括在匹配的单引号或双引号内。

www.TopSage.com

16.1.1 XML 文档的各个组成部分

前面提及了一些与 XML 相关的名词，但在开始深入介绍之前，还是有必要先学习一些术语，因为在谈及 XML 文档的时候将用到这些术语。

这里提供一个可能会在 XML 文档中碰到的所有重要部分的列表，如图 16-1 所示。文档多数组成部分是可选的，但有少数不是。有时，采用某一项就意味着必须也采用另一项。而有时，文档的各部分是相对独立的。

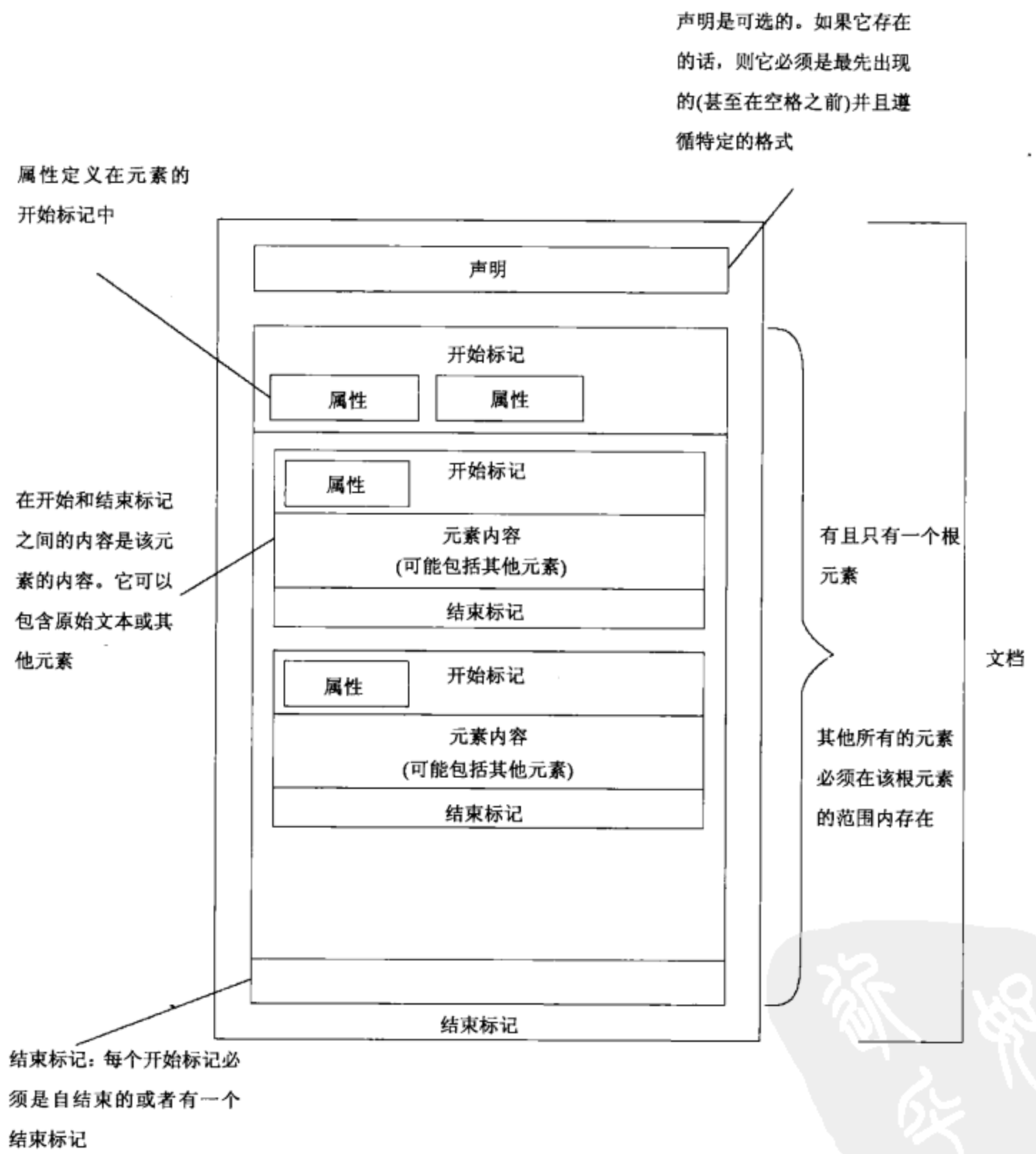


图 16-1

下面将采用层次结构方法(从属内容将列在其所属内容之后)介绍，具体到示例时，则会按各部分在给定的 XML 文档中的顺序说明。

www.TopSage.com

1. 文档(Document)

文档包含从第一个字符到最后一个字符的所有内容。当引用一个 XML 文档时，就意味着既引用了其结构，又引用了该特定 XML 文档中的内容。

2. 声明(Declaration)

从技术上说，声明是可选的，但是实际应用中常常应该包含声明。如果存在声明，那么声明必须在文档中最先出现。在声明之前不能出现其他任何内容，甚至是空白(空格、回车、制表符等)。

声明由一个特殊标记创建，该标记的起始部分是一个问号(表示这个标记是预处理指令)和一个 xml 名称：

```
<?xml version="1.0"?>
```

声明有一个必需的属性(用于进一步描述元素)——version。在前面的示例中，已经声明了这是一个 XML 文档，并且它遵循 XML 规范的 1.0 版本(编写本书时，已升级为 1.1 版本，但多数时候人们还是愿意使用 1.0)。

声明也可以可选地使用其他的属性——这个属性为 encoding，它描述了这个 XML 文档使用的字符集的性质。XML 可以处理一些不同的字符集，最主要的是 UTF-16 和 UTF-8。UTF-16 就是 Unicode 规范，它是一个 16 位的编码规范，适用于当今世界使用的大多数字符。默认的编码方法是 UTF-8，它向后兼容以前的 ASCII 规范。一个完整的声明如下所示：

```
<?xml version='1.0' encoding='UTF-8'?>
```

注意：

规范严禁使用以字母 xml 开始的元素，而是保留该元素用于将来的语言扩充。

3. 元素(Element)

元素的作用是将有关事物的所有描述信息粘合到一起——它可以是任何内容。元素会为描述性信息定义一个清楚的起始点和结束点。通常，元素存在于成对的开始和结束的标记中。然而，开始标记也可以自动结束——实质上可以定义空元素。

XML 元素的结构和 HTML 标记的结构很相似。开始标记以左尖括号(<)开始，包含名称和可能的属性，最后以右尖括号(>)结束：

```
<ATagForANormalElement >
```

这个规则的一个例外是如果元素是自动结束的，那么在这种情况下，结束标记的右尖括号之前有一个正斜杠“/”：

```
<AselfClosingElement/>
```

结束标记和开始标记极为相似(区分大小写)，但要在需要结束的元素名称前加一个正斜杠“/”：

```
<ATagForANormalElement > <== Opening Tag
```


www.TopSage.com

```
Some data or whatever can go in here.
We're still going strong with our data.
</ATagForANormalElement > <== Closing Tag
```

元素也可以包含属性(稍后将作介绍)作为元素的打开标记(但不是关闭标记)的一部分。而且，元素可以包含其他的元素，但是如果包含的话，在关闭外层元素之前必须先关闭内层的元素：

```
<OuterElement>
   <InnerElement>
   </InnerElement>
</OuterElement>
```

在介绍格式良好的概念时，将再次介绍元素相关内容。

4. 节点

如果绘制 XML 文档中所形成的层次结构，就会得到一个熟悉的树结构模型(在任何层次结构关系中都会碰到这种模型)，如图 16-2 所示。树中的每个交叉点称为节点。

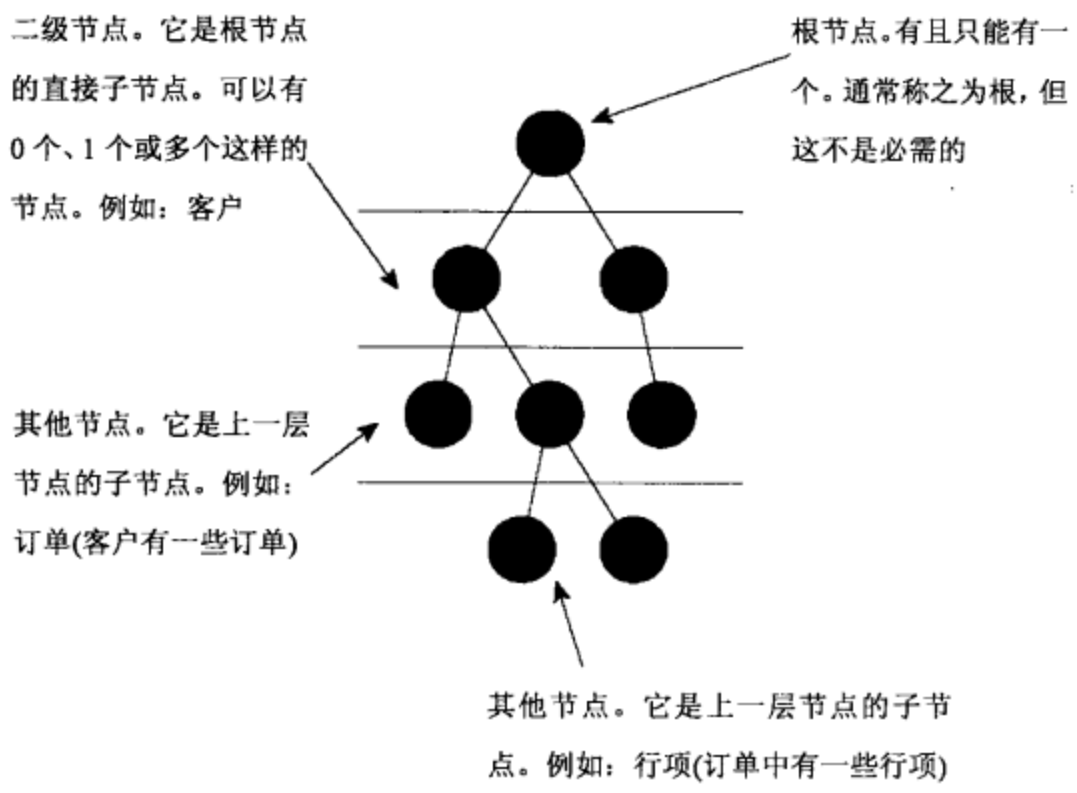


图 16-2

可以基于节点级别以及特定节点的属性值和元素值来导航 XML 文档的内容。

“根”节点

可能 XML 文档中最常见的一个混淆点就是关于“根节点”的。每个 XML 文档都必须有一个(不能多也不能少)根节点。根节点是包含了文档中其他元素(如果有其他元素)的节点。可以把根节点想象成把其下面的所有节点连接在一起并且在任何特定 XML 文档的范围内为子节点提供结构的统一节点。那么，混淆之处在哪里呢？主要有两点：一是人们不知道需要有单独一个根节点(现在您知道了)，二是人们不知道如何对根节点命名(稍后将会介绍)。

www.TopSage.com

因为通常的说法是“必须有一个根节点”，所以人们通常把这句话理解为必须有一个称为 root 的节点。的确，您可能偶尔会发现 XML 文档确实有一个根节点命名为 Root(或 root 或 ROOT)。但是现实的情况是，根节点和其他任何元素一样，都遵循相同的命名架构，除了一点例外——名称在文档中必须唯一。也就是说，整个文档中，其他任何的元素不可以和根节点有相同的名称。

说明：

根节点的存在是 XML 文档和 XML 片段的一个关键区别。通常，当从 SQL Server 中提取某些内容时，只能提取整个 XML 文档中的一小部分，本书称之为 XML 片段。因为 XML 片段并不是整个文档，所以 XML 片段中并不一定有根节点存在。

5. 属性(Attribute)

属性存在于元素的上下文中。属性是对元素进行更进一步描述的方式，位于元素的打开标记的范围内：

```
<SomeElement MyFirstAttribute="Hi There" MySecondAttribute="25">
   Optionally, some other XML
</SomeElement>
```

无论属性值属于何种数据类型的信息，都必须将属性值括在单引号或双引号中。

注意：

默认情况下，XML 文档没有数据类型的概念。本章后面将介绍描述单个文档应用程序规则的方法。并且将看到确保数据类型的一些方法——也就是为其设置规则，而 XML 自身不会设置规则。

6. 格式良好的概念

定义 XML 外观的规则(像使用何种元素、如何定义元素，以及元素有哪些部分)就是用于确定 XML 文档是否是格式良好的。

事实上，所有基于 SGML 的语言都有一个格式良好的概念。甚至 HTML 也有一些格式良好的概念——但大部分已经丧失，因为 HTML 本质上更为宽容，并且浏览器会忽略许多错误。

如果您熟悉 HTML 的话，那么一定了解基于标记的语言是很随意的。而 XML 对于格式有着更为严格的规则。简单地说有如下几点。

- 每个 XML 文档必须有一个唯一的“根”节点。
- 每个标记都必须有匹配的(区分大小写)结束标记，除非开始标记是自动结束的。
- 标记不能跨越其他标记。
- 除了 XML 分析器能够识别的字符外，不能使用限制字符。如果需要表示任何特殊的字符，那么需要使用转义序列(特殊字符会被转换为所需的字符)。

提示：

值得注意的是，现在的 HTML 文档要比以前更为“格式良好”。在 XML 出现时，XHTML 的规范也被开发了出来——也就是，实现也是有效 XML 的 HTML。现在许多开发人员尝试使

www.TopSage.com

他们的 HTML 文档满足 XHTML 标准，至少是获得更加格式良好的 HTML。

下面是一个格式良好的文档的示例：

```
<?xml version="1.0" encoding="UTF-8"?>

<ThisCouldBeCalledAnything>
 <AnElement>
   <AnotherElement AnAttribute="Some Value">
     <AselfClosingElement AnAttributeThatNeedsASpecialCharacter
="Fred&quot;s flicks"/>
   </AnotherElement>
 </AnElement>
</ThisCouldBeCalledAnything>
```

注意：

声明不需要结束标记。这是因为声明是预处理指令，而不是元素。从本质上讲，声明是在开始真正处理 XML 之前通知 XML 分析器需要知道的内容。

尽管这里非常简化地说明有关 XML 文档格式良好的要求，但它很好地概括了本书中要介绍的有限 XML 知识的基本内容。

注意：

了解这些概念对于理解本章后面的内容至关重要。下面介绍的示例可加深您的理解，但如果看过 XML 示例后还是觉得困惑，那么可以再学习一下前面的内容，或者参考清华大学出版社引进并出版的《XML 高级编程》或其他 XML 的书籍。在学习本章最后的样式和架构问题之前，必须理解这些内容。

7. XML 示例

本书会坚持采用示例来解释内容。但前面提到过，本书不是关于 XML 的书籍，所以这里并不准备采用示例作过多说明，而是简单看一下所讨论的主题。

在本章剩下的内容中，您将发现如果您配备了某种 XML 编辑工具(Microsoft 提供了一款免费的编辑工具，即 XML Notepad。我个人喜欢使用 XMLSpy，最早的功能齐全的 XML 编辑器之一)，那么一切都会变得简单很多。因为 XML 是基于文本的，所以可以很容易地在 Notepad 中打开并编辑 XML 文档—— 问题在于不能进行任何错误检查。那么如何知道文档是格式良好的呢？如果文档只有几行，那么可以从头至尾看一下，但如果是完整的文档或是样式表文档(本章后面将讨论转换)，那么查看文档就会变得非常困难。

提示：

可以通过在 Microsoft 的 Internet Explorer 中打开文档来执行一个快速而粗略的检查，以确定 XML 是否格式良好。如果文档不是格式良好，那么 Internet Explore 会报告错误。

在本例中，将看一下 AdventureWorks2008 数据库中数据的 XML 表示。您会看到一些订单信息。先介绍一些相关内容，然后在此基础上进行深入。

www.TopSage.com

首先，对于任何 XML 文档来说都需要一个根节点。可以对根节点任意命名，只要该名称在文档中唯一。一种常见的方法是将该根节点命名为 root。另外一个常见的方法是用能够表述该特定 XML 文档所表示的内容的名称来命名。

下面先从最简单的开始，只是使用 root：

```
<root>
</root>
```

可以迅速地创建第一个格式良好的 XML 文档。注意该 XML 文档没有包含前面见到的<?xml>标记。这里可以把该标记加进去，但它其实是一个可选项。与该标记相关的唯一限制是，如果选择该，则必须把标记放在最前面。为了获得最佳实践效果和清晰度，这里增加了这个选项：

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
</root>
```

事实上，根据 XML 的规则，任何以<?xml>开始的标记都是保留的标记——也就是说，不能命名该标记，因为当 XML 升级为新版本时，这些被保留的标记是供 W3C 现在或将来使用的。

现在已经有了第一个格式良好的 XML 文档。遗憾的是，该文档太过简单——不能说明任何内容。对于本例来说，想要描述订单的信息，所以需要向 XML 文档中加入一些描述订单的信息。首先是 SalesOrderHeader 标记：

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
   <SalesOrderHeader/>
</root>
```

这看上去并没什么变化，不是吗？现在已知 XML 文档中有一个订单，但还是没有关于订单的任何信息。这里通过添加一些属性来扩展订单的信息：

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
 <SalesOrderHeader CustomerID="510" SalesOrderID="43663"
OrderDate="2001-07-01T00:00:00" />
</root>
```

这里无需什么高深的知识，就可以辨别订单的基本内容。

- 客户的 ID 号是 510
- 订单 ID 号是 43663
- 订单日期是 2001 年 7 月 1 日

基本上，该订单和 SQL Server 中 AdventureWorks2008 数据库的 SalesOrderHeader 表中的一行记录等价。如果客户有多个订单，则会有如下结果：

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
 <SalesOrderHeader CustomerID="510" SalesOrderID="43663"
OrderDate="2001-07-01T00:00:00"/>
 <SalesOrderHeader CustomerID="510" SalesOrderID="44281"
```


www.TopSage.com

```
OrderDate="2001-10-01T00:00:00"/>
 <SalesOrderHeader CustomerID="510" SalesOrderID="45040"
OrderDate="2002-01-01T00:00:00"/>
 <SalesOrderHeader CustomerID="510" SalesOrderID="46606"
OrderDate="2002-07-01T00:00:00"/>
 <SalesOrderHeader CustomerID="510" SalesOrderID="47661"
OrderDate="2002-10-01T00:00:00"/>
 <SalesOrderHeader CustomerID="510" SalesOrderID="49824"
OrderDate="2003-04-01T00:00:00"/>
 <SalesOrderHeader CustomerID="510" SalesOrderID="55285"
OrderDate="2003-10-01T00:00:00"/>
 <SalesOrderHeader CustomerID="510" SalesOrderID="61178"
OrderDate="2004-01-01T00:00:00"/>
 </root>
```

虽然这是一个合法和格式良好的 XML 示例，但是它并没有真正表示出所期望的数据层次结构。例如，我们想以略有不同的方式创建该 XML，并且表示这样一种概念：客户通常位于该层次结构链中的较高层(订单的父层)。可以通过修改表示客户的方法来表示这种层次结构。这里客户不再是一个属性，而成为一个元素——可以包含自己的属性——在这个客户元素的内部包含了这个特定客户的订单：

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
<Customer CustomerID="510" AccountNumber="AW00000510">
 <SalesOrderHeader SalesOrderID="43663" OrderDate="2001-07-01T00:00:00"/>
 <SalesOrderHeader SalesOrderID="44281" OrderDate="2001-10-01T00:00:00"/>
 <SalesOrderHeader SalesOrderID="45040" OrderDate="2002-01-01T00:00:00"/>
 <SalesOrderHeader SalesOrderID="46606" OrderDate="2002-07-01T00:00:00"/>
 <SalesOrderHeader SalesOrderID="47661" OrderDate="2002-10-01T00:00:00"/>
 <SalesOrderHeader SalesOrderID="49824" OrderDate="2003-04-01T00:00:00"/>
 <SalesOrderHeader SalesOrderID="55285" OrderDate="2003-10-01T00:00:00"/>
 <SalesOrderHeader SalesOrderID="61178" OrderDate="2004-01-01T00:00:00"/>
</Customer>
</root>
```

如果有多个客户，就不会有问题——只要再添加另一个客户节点即可：

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
<Customer CustomerID="510" AccountNumber="AW00000510">
<SalesOrderHeader SalesOrderID="43663" OrderDate="2001-07-01T00:00:00"/>
 <SalesOrderHeader SalesOrderID="44281" OrderDate="2001-10-01T00:00:00"/>
 <SalesOrderHeader SalesOrderID="45040" OrderDate="2002-01-01T00:00:00"/>
 <SalesOrderHeader SalesOrderID="46606" OrderDate="2002-07-01T00:00:00"/>
 <SalesOrderHeader SalesOrderID="47661" OrderDate="2002-10-01T00:00:00"/>
 <SalesOrderHeader SalesOrderID="49824" OrderDate="2003-04-01T00:00:00"/>
 <SalesOrderHeader SalesOrderID="55285" OrderDate="2003-10-01T00:00:00"/>
 <SalesOrderHeader SalesOrderID="61178" OrderDate="2004-01-01T00:00:00"/>
 </Customer>
<Customer CustomerID="512" AccountNumber="AW00000512">
 <SalesOrderHeader SalesOrderID="46996" OrderDate="2002-08-01T00:00:00"/>
 <SalesOrderHeader SalesOrderID="48018" OrderDate="2002-11-01T00:00:00"/>
 <SalesOrderHeader SalesOrderID="49090" OrderDate="2003-02-01T00:00:00"/>
```


www.TopSage.com

```
 <SalesOrderHeader SalesOrderID="50231" OrderDate="2003-05-01T00:00:00"/>
</Customer>
</root>
```

照这样可以在层次结构中添加无限层(当然也要满足分析器可以处理的前提)。例如，可以为订单中的单个行项添加一层。

8. 确定元素与属性

首先要了解，没有固定的规则可以决定什么内容作为元素以及什么内容作为属性。属性描述了属性所属的元素的特性。某个元素的子元素(或者子节点)也有这样的作用。那么，如何决定哪个元素对应哪个属性呢？为什么属性是必需的？和现实中的大多数事情一样，这需要权衡。

属性在属性值与元素是一对一关系和属性作为元素的固有部分时很有意义。例如，在 AdventureWorks2008 数据库中，每个客户 ID 只有一个客户号——这就适合使用属性。当把关系数据转换为 XML 文档时，表中的列通常会变成元素的属性直接和表中的单个行相关。

元素在元素自身和描述该元素的内容之间存在一对多关系时更有意义。在本章前面的示例中，每一个客户有多个销售订单。从技术上说，可以将每一个订单作为客户元素的一个属性，但那就需要多次地重复许多客户元素信息。同样地，如果 AdventureWorks2008 数据库允许客户有别名(多个账号——类似于有多个联系人名)，那么我们可能希望在客户下面有一个 AccountNumber 元素，用其属性描述单个名称实例。

注意：

不管如何选择，都需要坚持本书中多次强调的规则——前后一致。如果在某处把一个给定的性质定义为属性，那在其他地方也要使之为属性，除非其性质在新位置中使用时有所不同。再提醒一次：前后一致。

16.1.2　名称空间

虽然可以随意地创建自己的标记、混合和匹配其他来源的数据，以及自定义其他操作，但还是会碰到在何处使用何种名称的冲突。例如，在分别应用于图书馆应用程序(对它而言，letter 可能是指一个人写给另一个人的信件)和另一个描述字体(可能描述一个字符集和一系列字形之间的关联)的应用程序时，名为 letter 的元素可能会有完全不同的结构、规则和含义。

进一步地讲，XML 的本质就是世界上的各行业组织都逐步约定了一些命名和结构规则来描述其行业内各种信息。图书馆行业组织可能已经约定了一些元素形式来描述书籍、戏剧、电影、信件、论文等。同时，操作系统和/或制图行业也可能约定了一些元素形式来描述图片、字体和文档布局。

现在，试想一下，这些不幸的开发人员被要求编写一个应用程序，该应用程序要描述图书馆目录。显然，图书馆目录经常会使用到像字体这样的内容——所以，当在 XML 中引用称为 letter 的内容时，是在引用和字体有关的内容还是引用一封某人写给另一个(比如是 Thomas Jefferson 写给 George Washington 的)人的信件呢？这里产生了冲突，需要一种解决这个问题的方法。

这时就需要引入名称空间。名称空间描述了元素和属性的域以及它们的结构。支持图书馆

www.TopSage.com

中字体的结构会在图书馆名称空间中进行描述。同样的，制图行业可能也会有它们自己的名称空间来描述与该行业相关的字体。有关名称空间的信息存储在一个引用文档中，可以使用统一资源标识符(Uniform Resource Identifier，URI)找到它们。URI 是个特殊的名称，和 URL 不同，最终会被解析为引用文档。

当要创建同时引用图书馆和图形结构的 XML 文档时，只需简单地引用相应行业的名称空间。另外，限定元素和属性的性质将由名称空间描述。通过使用名称空间来限定名称，可以确保，即使文档中有结构不同但名称相同的元素，在引用时也不会引用错误的元素或属性类型。

要引用整个文档的名称空间，只要将引用作为根节点的一个特殊属性(称为 xmlns)进行添加。该引用将提供本地名(如何引用名称空间)以及最终解析为引用文档的 URI。如果想在指定了名称空间的节点的范围内应用该特定的名称空间，那么也可以为文档中的其他节点添加名称空间引用(还是使用 xmlns)。

下面是一个 XML 文档的示例(从技术上说，这就是所谓的架构)，本章后面还会用到。请注意其中和名称空间相关的一些内容。

该文档引用了 3 个名称空间——XDR 名称空间(碰巧这是个 XDR 文档)、Microsoft 数据类型名称空间(它会建立一个关于不同数据类型的数目和性质的列表)，以及用于 SQL Server XML 集成的特殊的 SQL 名称空间。

一些属性(包括根节点中的属性)用名称空间的信息加以限定(例如 sql:relation 属性)。

```
<?xml version="1.0" encoding="UTF-8"?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
                xmlns:dt="urn:schemas-microsoft-com:data types"
                xmlns:sql="urn:schemas-microsoft-com:xml-sql"
                sql:xsl='../Customers.xsl'>
<ElementType name="Root" content="empty" />
<ElementType name="Customers" sql:relation="Customers">
      <AttributeType name="CustomerID"/>
      <AttributeType name="CompanyName"/>
      <AttributeType name="Address"/>
      <AttributeType name="City"/>
      <AttributeType name="Region"/>
      <AttributeType name="PostalCode"/>
      <attribute type="CustomerID" sql:field="CustomerID"/>
      <attribute type="CompanyName" sql:field="CompanyName"/>
      <attribute type="Address" sql:field="Address"/>
      <attribute type="City" sql:field="City"/>
      <attribute type="Region" sql:field="Region"/>
      <attribute type="PostalCode" sql:field="PostalCode"/>
 </ElementType>
</Schema>
```

sql 数据类型引用了一些特殊的属性。不需要担心 Microsoft 数据类型名称空间是否也有字段或关系数据类型，因为这里已经完全限定了属性名。即使数据类型名称空间确有一个称为 field 的属性，XML 分析器也会根据 sql 名称空间的规则对这一元素进行处理。

16.1.3 元素内容

另一个关于 XML 和元素值得一提的概念就是元素内容。

www.TopSage.com

元素可以包含属性以外的数据和嵌套元素。虽然嵌套元素是元素内容的一种形式(一个元素包含另一个元素)，但是 XML 也允许在元素中包含行文本信息。例如，可能有如下所示的 XML 文档：

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
 <Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
   <Note Date="1997-08-25T00:00:00">
   The customer called in today and placed another order. Says they really
like our work and would like it if we would consider establishing a location
closer to their base of operations.
   </Note>
   <Note Date="1997-08-26T00:00:00">
   Followed up with the customer on new location. Customer agrees to guarantee
us $5,000 per month in business to help support a new office.
   </Note>
 </Customer>
</root>
```

Note 元素的内容，如“The customer called…”，既不是元素也不是属性，而是有效的 XML 数据。

虽然这类数据存在于 XML 中，但是 SQL Server 并不使用任何自动的样式化方法以这种格式输出数据。RDBMS 系统的行/列方法更适合于元素和属性。要输出这样的注释数据，需要使用支持更显式的 XML 输出格式的方法，或是选择输出数据后执行转换。本章最后将介绍转换。

16.1.4　有效与格式良好——架构和 DTD

仅仅因为 XML 文档是格式良好的并不能说明它是有效的 XML。为避免这种错误认知的影响，这里需要说明的是，除非 XML 能通过一些规范文档形式的验证，否则不能被认为是有效的。目前，只有两种认可的规范文档类型——文档类型定义(DTD)和 XML 架构。

这两种验证文档的基本前提大致相同。XML 作为一种语言，定义了 XML 文档需遵循的基本规则，而 DTD 和 XML 架构探求定义的规则只是针对某一特定类的 XML 文档。这两种方法实现起来略有不同，但都有各自的优点。

- **DTD**：这是一种反复检验和正确的方法。DTD 用于 SGML 中(XML 是一种 SGML 应用程序，可以认为 SGML 是 XML 的超集，但是极难掌握)，DTD 的优点是知名度高且广为接受。现在有很多 DTD 可供使用。

  缺点在于 DTD 毕竟是较“老”的一种方法。这里不是说老的就不好，但是现在 DTD 确实跟不上文档技术发展的脚步了。DTD 并不允许类似限定数据类型这样看似基本的操作。在 XML 中使用 DTD 时会发现，由于 XML 架构更受欢迎，因此在大多数时候都不赞成使用 DTD。
- **XML 架构**：XML 架构有着强类型化的明显优势。XML 架构的优点在于您可以有效地建立自己的复杂数据类型—— 方法是基于结合一个或多个其他数据类型(包括其他复杂的数据类型)，或者需要特殊的架构匹配(例如，社会保障号码只是一个数字，但是通过 XML 架构可以很容易地对它进行强制格式化，使它具有特殊的格式)。XML 架构还有一个优点，顾名思义，XML 架构本身就是一个 XML 文档。这意味着编写 XML 文

www.TopSage.com

档的很多技巧也适用于编写 XML 架构(虽然要学的东西还很多)，而且这些架构本身也可以自描述——可使用另一个架构来对 XML 架构自身进行验证。

16.2 SQL Server 提供的 XML 支持

到目前为止，已经学习了 XML 的基本内容。现在需要了解 SQL Server 中的相关内容。

XML 功能是 SQL Server 后来添加的内容。XML 首次出现时是作为 SQL Server 7.0 的可下载插件。此外，该功能的重要部分最初更多是添加到 Internet Information Server(IIS)中，而非 SQL Server。

SQL Server 2000 把 XML 放入了 Microsoft 所谓的 Web Release 模型中，并对其进行了多次更新。在 SQL Server 2005 中，XML 成为核心的产品。虽然仍旧支持大部分原有的功能，但是 SQL Server 仍在不断增加更多核心特性，使得 XML 成为一项必不可少的内容，而并非只是先前版本中的后续添加功能。

有哪些 XML 功能呢？SQL Server 2008 在以下情况中会使用 XML。

- 支持多种从普通列中选取数据并以 XML 格式接收数据的方法
- 支持在 SQL Server 中使用 XML 数据类型本地存储 XML 数据
- 支持使用 XQuery(一种特殊的 XML 查询语言)和其他方法查询以原 XML 格式保存的数据
- 支持使用 XML 架构对以 XML 格式存储的数据强制数据完整性
- 支持对 XML 数据进行索引
- 支持具有层次结构的数据——为 XML 数据中常见的树状结构提供特别支持

以上这些只是主要应用。

对以上各种应用的支持通常会用到 XML 支持的多个功能领域，下面将逐个介绍这些 XML 支持。

16.2.1 将列定义为 XML 类型

本章前面已经见过 XML 列最基本的定义。例如，如果查看 AdventureWorks2008 数据库中 Production.ProductModel 表最基本的定义，则结果如下所示：

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

通过这两个 XML 列，可得到的以下信息。

(1) 这两个列被定义为 XML，因此将使 XML 数据类型方法可用(后面会详细介绍)。

www.TopSage.com

(2) 允许 NULL，但也可以选择 NOT NULL 作为约束。请注意，NOT NULL 约束强制对应的是行中是否有该列的数据，而不是数据是否有效。

(3) 这里的 XML 被认为是“非类型化的 XML”。也就是说，由于没有任何架构与其相关联，因此 SQL Server 无法获知应该如何使它“有效”。

以上第一条信息暗含的信息在任何一个被定义为 XML 数据类型而不是普通文本的列中。16.2.4 节将介绍更多相关内容。

第二条信息是 SQL Server 的任何数据类型都有的——可以指定该列是否允许使用 NULL 数据。

因此，定义时真正能做的改动是将 XML 列指定为类型化的 XML 还是非类型化的 XML。前面的示例中使用的非类型化的定义说明了 SQL Server 由于不了解存储在列中的任何 XML，因此无法确定其有效性。如果在创建列时将其定为类型化的 XML，那么就可以为 XML 列提供更多有关“有效的”XML 的定义。

AdventureWorks2008 数据库已经有了与期望在这两个 XML 列设置的有效性验证相匹配的架构集合，因此下面就来看一下如何对 CREATE 语句做一些调整，改成类型化的 XML：

```
CREATE TABLE Production.ProductModel
(
   ProductModelID       int IDENTITY(1,1) PRIMARY KEY 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 DF_ProductModel_ModifiedDate DEFAULT (GETDATE())
);
```

这显示了 XML 在实际 AdventureWorks2008 样本数据库中定义的方式。要在 Production.Product Model 表中插入一个记录，必须清空 CatalogDescription 和 Instructions 字段，或提供一个能通过其各自架构测试的有效的 XML。

16.2.2　XML 架构集合

XML 架构集合其实就是数据库中被命名的一个或多个架构文档。名称就相当于架构集合的句柄。通过引用这个集合，表明当与集合中的所有架构进行匹配时，XML 类型的列或变量必须是有效的。

可以使用内置的 XML_SCHEMA_NAMESPACE()函数来查看现有的架构集合。语法如下：

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

上述语法不易理解，因此对这些参数稍作解释：

www.TopSage.com

表 16-1

参　数	说　明
SQL Server schema	这是一个关系数据库架构(不要与 XML 架构混淆)。例如，对表 Production.ProductModel 来说，关系架构是 Production；对表 Sales.SalesOrderHeader 来说，关系架构是 Sales
xml schema collection	这是创建 XML 架构集合后所用的名称。在之前的 CREATE 表的示例中，引用了 ProductDescriptionSchemaCollection 和 ManuInstructionSSchemaCollection 这两个 XML 架构集合
namespace	这是 XML 架构集合中某个特定名称空间的可选名称。请记住 XML 架构集合可以包含多个架构文档——它将返回特定名称空间内的任何内容

因此，要查看 Production.ManuInstructionsSchemaCollection 架构集合，可使用以下查询：

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

得到大量未格式化的 XML：

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:t="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductMode
lManuInstructions"
targetNamespace="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Pro
    ductModelManuInstructions" 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:sequ
    ence></xsd:restriction></xsd:complexContent></xsd:complexType></xsd:element><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:sc
hema>
```

SQL Server 去除了标记间的所有空格，所以如果在创建架构集合时出于可读性的考虑而使用了缩进格式，那么 SQL Server 也会出于高效存储的考虑删除所有空格。

www.TopSage.com

注意：

Management Studio 中对于文本结果返回的默认字符数只能是 256 个字符。如果使用的是文本视图，那么需要选择“工具”|“选项”|“查询结果”|“SQL Server”|“结果显示到文本”命令，然后对可显示的最多字符数进行修改。

16.2.3　创建、修改和删除 XML 架构集合

XML 架构集合的创建、修改和删除(Creat、Alter 和 Drop)操作的方式和 SQL Server 中其他同类语句的工作方式基本一致。这里会一一运行它们，但要特别注意这里的 ALTER 语句，ALTER 语句有一些内容比较怪异，而这些怪异在以前所使用的其他 ALTER 语句中从未见过。

1. CREATE XML SCHEMA COLLECTION

这里的 CREAT 语句仍旧是本书中多次出现的典型的 CREATE <object type> <object name> 语句，并且和存储过程、视图及其他结构性较弱的对象一样，使用了关键字 AS：

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

如果要创建一个和 AdventureWorks2008 数据库中的 Production.ManuInstructions SchemaCollection 集相似的 XML 架构集合，可能要执行到如下语句：

```
CREATE XML SCHEMA COLLECTION ProductDescriptionSchemaCollectionSummaryRequired
 AS
   '<xsd:schema targetNamespace="http://schemas.microsoft.com/sqlserver
   /2004/07/
 adventure-works/ProductModelWarrAndMain"

 xmlns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works
 /ProductModelW arrAndMain"
       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/
adventure-works/ProductModelDescription"
       xmlns="http://schemas.microsoft.com/sqlserver/2004/07/adventure
-works/ProductModelDescription"
       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/adventure
-works/ProductModelWarrAndMain" >
       <xs:import namespace="http://schemas.microsoft.com/sqlserver/2004/07/
```


www.TopSage.com

```
adventure-works/ProductModelWarrAndMain" />
      <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 部分必须在一行中。这里显示为多行是受排版印刷限制。所以请确保整个 URL 位于一行中。

这个架构集合和 Production.ManuInstructionsSchemaCollection 架构集合相似，但这里对架构做了改动，添加了 summary 元素而非使之可选。因为基本结构相同，所以这里使用了相同的名称空间。

2. ALTER XML SCHEMA COLLECTION

这和其他的 ALTER 语句略有不同，因为这里的功能仅限于对架构集合添加新的片段。语法如下：

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

注意：

诚然，在后面的服务包中，其功能会有所提高，但同时要强调的是，这一工具的作用是为架构集合添加内容，而不是修改或删除内容。

3. DROP XML SCHEMA COLLECTION

这是典型的“傻瓜型”操作，工作原理与其他的 DROP 语句相同：

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

因此，如果要删除前面创建的 ProductDescriptionSchemaCollectionSummaryRequired 架构集合，可执行下面的操作：

```
DROP XML SCHEMA COLLECTION ProductDescriptionSchemaCollectionSummaryRequired;
```

这样一来就删除了该架构集合。

www.TopSage.com

16.2.4　XML 数据类型方法

XML 数据类型有若干自带的方法。这些方法是 XML 数据类型所特有的，其他现有的数据类型都没有。这些方法的语法稍有不同，因为它们基于不同的、但多数行业标准的访问 XML 的方法。调用这些方法的基本标准语法如下所示：

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

一共有以下 5 种可用的方法。

- .query：一种行业标准的 XQuery 语言实现方式。该方法允许通过运行 XQuery 格式的查询来访问 XML。XQuery 允许返回多个数据片段而非离散值。
- .value：这种方法允许访问特定元素或属性中的一个离散值。
- .modify：这是 Microsoft 自身对 XQuery 的扩展。XQuery 被限定为只能读数据(没有可用的修改语言)，modify 方法将 XQuery 扩展为可以对数据进行修改。
- .nodes：用于将 XML 数据拆分为单独的、关系型行。
- .exist：该方法与在标准 SQL 中广泛使用的 IF EXIST 子句(用于测试是否存在某种特殊类型的数据)十分相似。exist()将测试是否有某个特定的节点或属性在所测试的 XML 实例中。

1. .query 方法(SQL Server 实现的 XQuery)

.query 是一种行业标准的 XQuery 语言实现方式。其结果类似一个 SQL 查询，除了结果是与 XML 数据节点匹配而不是与关系行或列匹配。

.query 需要一个参数，该参数是对 XML 数据实例运行的一个有效 XQuery。举个例子，如果想跳出 ProductID 66 对应的生产文档，那么可以运行以下语句：

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

注意：

名称空间声明的 URL 部分必须在一行中输入。这里显示为多行是因为排版印刷限制。所以，请确保整个 URL 位于一行中。

这个结果相当冗长，所以这里删除了它的右半部分，以至于仅剩那些位于 XML 层次结构中的步骤层或更低层中的节点：

```
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...
```


www.TopSage.com

```
                    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 行受影响)
```

这里还要指出的是，对于数据库中的每个数据行而言，所有的 XML 只进入每一数据行中的一列。

说明：

这里再次重申，.query 不能修改数据；它是一个只读操作。

另外还请注意，这里需要声明名称空间。由于名称空间是作为引用的架构集合的一部分被声明的，因此名称空间会不断扩展，并且破坏查询的可读性。可使用 WITH XMLNAMESPACES() 声明解决这个问题：

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

SELECT ProductModelID, Instructions.query('/PI:root/PI:Location/PI:step') AS Steps
FROM Production.ProductModel
WHERE ProductModelID = 66;
```

注意：

名称空间声明的 URL 部分必须在一行中输入。这里显示为多行是因为排版印刷限制。所以请确保整个 URL 位于一行中。

这个查询的可读性更好，产生的结果集依然相同。

注意：

您可能发现导航到 ProductManualInstructions 的实际 URL 是件很有意思的事情。在简要介绍 HTML 页面之后，页面会指出查询所用的实际架构文档。

2. .value 方法

.value 方法用于查询离散数据。该方法使用一个称为 XPath 的特殊 XML 路径语言来定位一个特定的节点并提取一个标量值。其语法如下：

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

这里的技巧在于要确保指定的 XPath 确实会返回一个离散值。

例如，如果想知道 ProductModelID 66 的第一个 Location 元素中 LaborHours 属性的值，则可编写如下语句：

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

SELECT ProductModelID,
```


www.TopSage.com

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

注意：

名称空间声明的 URL 部分必须在一行中输入。这里显示为多行是因为排版印刷限制。所以，请确保整个 URL 位于一行中。

检查一下结果：

```
ProductModel          ID Location
--------------        -------------------------------------
66                    1.50

(1 行受影响)
```

注意，SQL Server 仅将指定属性(本例中为 Location 节点的 LaborHours 属性)的值作为一块离散数据进行提取。返回值的数据类型必须强制转换为 SQL Server 的非 XML 类型，而且必须返回一个标量值——也就是说不能有多个行。

3. .modify 方法

这部分内容很有趣。

标准 W3C 形式的 XQuery 是只读的——也就是说，用 XQuery 来选择数据很好，但不能进行 INSERT，UPDATE 和 DELETE 等操作。这真是叫人大失所望。但是 Microsoft 却不会有这个问题，并且已对 XQuery 进行了扩展，使其可以对数据进行操作。这一 XQuery 扩展被称为 XML Data Manipulation Language(XML 数据操纵语言)，或称为 XML DML。XML DML 为 XQuery 添加了以下 3 种新命令。

- insert
- delete
- replace value of

说明：

这些新的命令和所有 XML 关键字一样，都是区分大小写的。

这些命令都名副其实，replace value of 命令取代了 SQL 的 UPDATE 语句。

如果想要在.value 示例中提高原本为 1.5 个小时的劳动时间，可使用如下语句：

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

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


www.TopSage.com

注意：

名称空间声明的 URL 部分必须在一行中输入。这里显示为多行是因为排版印刷限制。所以，请确保整个 URL 位于一行中。

现在再次运行.value 命令：

```
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelMan
uIn
structions' AS PI)

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

注意：

名称空间声明的 URL 部分必须在一行中输入。这里显示为多行是因为排版印刷限制。所以，请确保整个 URL 位于一行中。

得到一个新值：

```
ProductModel       ID Location
--------------     ----------------------------------------
66                 1.75

(1 行受影响)
```

注意：

这其实是在一个 UPDATE 操作内又有一个 UPDATE 操作。因为对 SQL Server 的行作修改，所以必须使用 UPDATE 语句告知 SQL Server，关系数据行(其中正好有 XML)将要被更新。还必须使用关键字 replace value of 来指定要更新的 XML 部分。

4. .nodes 方法

.nodes 方法用于对 XML 架构块进行拆分，如果是存储为关系形式，就拆分成多个数据行。这种将一个 XML 文档拆分为多个单独部分的方法成为“拆解”文档。

实质上，要用.nodes 方法将 XML 数据的实例分散到其各自的表中(有多少满足 XQuery 标准的数据实例，就有多少行)。如您所想的那样，这意味着需将.nodes 结果作为一个表而不是一个列来对待。.nodes 结果和一个典型表的主要区别在于，必须交叉应用.nodes 的结果来返回作为 XML 数据来源的特定表。所以，.nodes 方法涉及更多语法，而不仅仅是“.nodes”——可将其看作连接(join)，只不过用特殊的 CROSS APPLY 关键字取代了 JOIN，用.nodes 取代了 ON 子句。其语法如下所示：

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


www.TopSage.com

这里的内容相当费解，所以回顾一下先前的.value 示例。下面是一个查找特定项的查询，因此得到一个结果：

```
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelMan
uIn
structions' AS PI)

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

注意：

名称空间声明的 URL 部分必须在一行中输入。这里显示为多行是因为排版印刷限制。所以，请确保整个 URL 位于一行中。

.value 方法希望得到一个标量结果，所以需确保 XQuery 对 XML 文档中的每一行只返回单个值。.nodes 方法通知 SQL Server 使用 XQuery 映射到一个特定位置，把在 XQuery 中找到的每个项当作单个行而不是离散值来对待。

```
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/
ProductModelManuInstructions' 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);
```

注意：

名称空间声明的 URL 部分必须在一行中输入。这里显示为多行是因为排版印刷限制。所以，请确保整个 URL 位于一行中。

可注意到，通过.nodes 方法，实质上是将一个表(ProductModel)变成了两个表(数据源表和 ProductModel 表中的 Instructions 列所产生的.nodes 结果)。看一下这一方法的结果：

```
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
```


www.TopSage.com

```
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 行受影响)
```

可以看到，这里对 ProductModel 表中原本的单行返回了多行。例如，ProductModelID 7 有 6 个不同的 Location 元素实例，因此结果得到了 6 行而不是 ProductModel 表中的单行。

虽然这可能是 XML 数据类型方法中最复杂的一种，但它在将 XML 数据转换成关系型数据方面有着强大的功能。

5. .exist 方法

.exist 方法的工作方式和 SQL 中的 EXISTS 语句很相似。它接受一个表达式(在这里是一个 XQuery 表达式而不是 SQL 表达式)，返回一个布尔值，表明该表达式是真还是假(结果也可能为 NULL)。

在.modify 示例中，如果想要显示包含带有规格元素的步骤的行，则可使用.exist 方法：

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

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

注意：

请特别注意应用测试条件的位置。

例如，这段代码显示了至少有一步中含有一个规格元素的行——并不要求每一步都含有规格元素。如果想要对每个元素都进行测试，则需将元素作为单个行取出(使用.nodes)，或者将测试条件放入 XQuery 中。

名称空间声明的 URL 部分必须在一行中输入。这里显示为多行是因为排版印刷限制。所以，请确保整个 URL 位于一行中。

www.TopSage.com

16.2.5　强制执行架构集合之外的约束

现在，对约束的概念已经比较熟悉了，本书也大量使用了约束。如果关系数据库需要约束，其内容和 XML 数据是相同的。事实上，在 XML 中已经通过使用架构集合实现了大量的约束概念。但是如何强制执行基本架构之外的限制条件呢？

令人惊讶的是，在约束声明中不能应用 XML 数据类型方法。那么如何解决这个问题呢？可以将测试打包为一个用户自定义函数(UDF)，然后在约束中使用这个函数。

注意：

可以很惊讶地看到，约束声明中不能使用 XML 数据类型方法，但可以使用函数。这里只能盼望有关人员继续努力，在后续版本中早日解决这个问题。

16.2.6　用 XML 格式检索关系数据

SQL Server 在很早的版本中就提供了检索关系数据的功能。也提供了一些不同的选项，在这些选项内又可以有更多的选项，总之功能非常灵活。下面就介绍一下这些选项。

FOR XML 子句

这个子句是大量可用的不同集成模型的基础。它实际上只是一个添加到现有的 T-SQL SELECT 语句末尾的选项而已，但却把以普通关系格式存储的数据输出作为 XML 的主要的方法。

这里回过头来看一下第 3 章中的 SELECT 语句的语法：

```
SELECT [TOP (<expression>) [PERCENT] [WITH TIES]] <column list>
[FROM <source table(s)/view(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 <column list>]
[[FOR XML {RAW|AUTO|EXPLICIT|PATH [(<element>)]}
 [, XMLDATA | XMLSCHEMA [(<target namespace>)]]
 [, ELEMENTS [XSINIL | ABSENT]][, BINARY base 64][ROOT('<root definition>')]]
[OPTION (<query hint>, [, ...n])]
```

其中大部分的内容现在看来都是微不足道的——毕竟，到目前为止在很多较难的章节中都使用过该语法——这里要介绍的是 FOR XML 语句。

FOR XML 语句提供了 3 种对结果进行 XML 格式化的初始选项。

- RAW：这会将结果集中的每一行数据都作为单个数据元素返回，元素名称为 row，列出来的每一列作为这个 row 元素的属性。即使连接多个表，RAW 选项输出的元素个数也会和标准 SQL 查询中所查询到的行数一样。
- AUTO：这个选项会用表示数据来源的表名或者别名对每个元素进行标记。如果查询中输出数据来源于多个表，那么每个表中的数据都将分割为单独的、嵌套的元素。如果使用了 AUTO 选项，而且想让列数据作为元素而非属性显示，那么也将支持另外一个选项 ELEMENTS。

www.TopSage.com

- EXPLICIT：这个选项是对查询进行格式化的最为复杂的选项，但其最终结果是对 XML 的外观拥有高度的控制能力。有了这个选项，可以为返回的数据定义某种层次结构，然后格式化查询，使其中每一个数据片段都属于一个层次结构的特定层(并相应地被指定一个标记)。EXPLICIT 选项基本上已被 PATH 选项所取代，在这里前者主要是为了向后兼容。
- PATH：该选项用一种更便于使用的格式，具有比 EXPLICIT 更大的灵活性—— 当需要对输出格式进行高度控制时，通常可选择这一选项。

注意：

以上选项都没有提供必要的根元素。如果想使 XML 文档是格式良好的，那么需要将结果放到合适的根元素的开始和结束标记中，或者让 SQL Server 来完成这一工作(后面将介绍 ROOT 选项的使用)。虽然在某种程度还有争议，但根元素确实有好处——这意味着可以把多个 XML 查询串接起来，并且把不同的结果打包成一个 XML 文件，从而建立更为复杂的 XML。

除了以上 4 种主要的格式化选项以外，还有其他可选的参数来进一步修改 SQL Server 在 XML 查询中所给出的输出。

- XMLDATA/XMLSCHEMA：该参数告诉 SQL Server 结果应采用 XML 两种架构中的一种。XMLDATA 是基于原有的 XDR 格式工作的，这在 W3C 最终确定 XML 架构文档的规范之前十分常见。但现在通常会使用 XMLSCHEMA，除非您有非常特殊的理由需要使用原有的 XDR 格式。因为 XMLDATA 选项仅仅是为了向后兼容而提供的，而且不支持 SQL Server 2005 和 2008 中添加的新数据类型。
- ELEMENTS：这个选项只在使用 AUTO 格式化选项时才可用。它告诉 SQL Server 将数据中的列作为嵌套元素而不是作为属性返回。
- BINARY BASE64：该选项告诉 SQL Server 将任何二进制列(binary、varbinary、images)用 base64 格式进行编码。如果也使用了 AUTO 选项，那么默认使用了 BINAR BASE64 选项(即使没有声明，SQL Server 也会使用该选项)。该选项对于使用 EXPLICIT 和 RAW 选项的查询是必须的。
- TYPE：告诉 SQL Server 以 XML 数据类型而非默认的 Unicode 字符类型返回结果。
- ROOT：该选项会使 SQL Server 添加根节点，这样您就不必自己添加。可以为根提供一个名称或者使用默认的名称(root)。

下面将更详细地介绍这些选项。

16.2.7 RAW 选项

这是一个很清晰的选项。只要执行就可以了—— 没有花样，也不需要进行特殊的格式化—— 仅仅是把关系数据行转换为 XML 数据的元素。如果运行一个更为传统的 SELECT 语句，则该元素的名称是 row，Select 列表中的每一列将使用相应的列名把属性添加到该元素中。

注意：

属性命名方式的缺点是，需要确保每一列都有一个名称。通常，SQL Server 在执行聚合或者其他计算列且不提供别名的情况下，将不显示任何列标题——当执行 XML 查询时，所有的

www.TopSage.com

内容都必须有名称，所以不要忘了对计算列起别名。

首先看一下相对简单的示例。假定经理要求我们提供一个列出一些客户订单的查询——如 CustomerID 1 和 CustomerID 2。在浏览了本书前 5 章的内容后，您或许会说“没有问题！”并且给出如下查询：

```
USE AdventureWorks2008;

SELECT Sales.Customer.CustomerID,
 Sales.Customer.AccountNumber,
 Sales.SalesOrderHeader.SalesOrderID,
CAST(Sales.SalesOrderHeader.OrderDate AS date) AS OrderDate
FROM Sales.Customer
JOIN Sales.SalesOrderHeader
 ON Sales.Customer.CustomerID = Sales.SalesOrderHeader.CustomerID
WHERE Sales.Customer.CustomerID=29890 OR Sales.Customer.CustomerID=30067;
```

您将下列结果交给经理：

```
CustomerID  AccountNumber SalesOrderID OrderDate
----------- ------------- ------------ ----------
29890        AW00029890    43671        2008-08-16
29890        AW00029890    45049        2008-08-16
29890        AW00029890    45790        2008-08-16
29890        AW00029890    46619        2008-08-16
29890        AW00029890    47672        2008-08-16
29890        AW00029890    48732        2008-08-16
29890        AW00029890    49866        2008-08-16
29890        AW00029890    61187        2008-08-16
30067        AW00030067    43672        2008-08-16
30067        AW00030067    44294        2008-08-16
30067        AW00030067    45052        2008-08-16
30067        AW00030067    45792        2008-08-16
30067        AW00030067    46622        2008-08-16
30067        AW00030067    47673        2008-08-16
30067        AW00030067    48768        2008-08-16
30067        AW00030067    49860        2008-08-16
30067        AW00030067    51100        2008-08-16
30067        AW00030067    55287        2008-08-16
30067        AW00030067    61222        2008-08-16
30067        AW00030067    67272        2008-08-16
(20 行受影响)
```

很简单，是吧？现在经理又说：“很好——我刚想让 Billy Bob 把这些结果转换成 XML，但他说这需要一天或两天的时间，太糟了。”您便可以不失时机地说：“为什么不早点说呢？”因为只要简单地加入 3 个关键字即可：

```
USE AdventureWorks2008;

SELECT Sales.Customer.CustomerID,
 Sales.Customer.AccountNumber,
 Sales.SalesOrderHeader.SalesOrderID,
```


www.TopSage.com

```
 CAST(Sales.SalesOrderHeader.OrderDate AS date) AS OrderDate
FROM Sales.Customer
JOIN Sales.SalesOrderHeader
 ON Sales.Customer.CustomerID = Sales.SalesOrderHeader.CustomerID
WHERE Sales.Customer.CustomerID = 29890 OR Sales.Customer.CustomerID = 30067
FOR XML RAW;
```

这样会让经理很开心。输出的结果集是和运行标准 SQL 查询所得到的结果集一对一匹配的：

```
<row CustomerID="1" AccountNumber="AW00000001" SalesOrderID="43860"
OrderDate="Aug 1 2001" />
<row CustomerID="1" AccountNumber="AW00000001" SalesOrderID="44501"
OrderDate="Nov 1 2001" />
<row CustomerID="1" AccountNumber="AW00000001" SalesOrderID="45283"
OrderDate="Feb 1 2002" />
<row CustomerID="1" AccountNumber="AW00000001" SalesOrderID="46042"
OrderDate="May 1 2002" />
<row CustomerID="2" AccountNumber="AW00000002" SalesOrderID="46976"
OrderDate="Aug 1 2002" />
<row CustomerID="2" AccountNumber="AW00000002" SalesOrderID="47997"
OrderDate="Nov 1 2002" />
<row CustomerID="2" AccountNumber="AW00000002" SalesOrderID="49054"
OrderDate="Feb 1 2003" />
<row CustomerID="2" AccountNumber="AW00000002" SalesOrderID="50216"
OrderDate="May 1 2003" />
<row CustomerID="2" AccountNumber="AW00000002" SalesOrderID="51728"
OrderDate="Aug 1 2003" />
<row CustomerID="2" AccountNumber="AW00000002" SalesOrderID="57044"
OrderDate="Nov 1 2003" />
<row CustomerID="2" AccountNumber="AW00000002" SalesOrderID="63198"
OrderDate="Feb 1 2004" />
<row CustomerID="2" AccountNumber="AW00000002" SalesOrderID="69488"
OrderDate="May 1 2004" />
```

提示：

要知道 Management Studio 会截断长度超过“将结果保存到文件”选项卡的“选项”菜单中所设置的值(最大为 8192)的任何列。如果直接输出到文件，则会在结果窗口(网格的或文本的)中出现这一问题。这是工具的问题，而非 SQL Server 本身的问题。如果使用另一种方法来检索结果(例如使用 ADO)，那么就不会遇到这个问题。

对于查询产生的每一个数据行，在 XML 中都有一个元素与之对应。不管是以哪个表作为数据源，所有的列信息都将作为 row 元素的属性来表示。这种做法的缺点是并没有表示出真正的数据层次结构的本质——只能由客户下订单。但是这样做的好处是，如果使用 XML DOM 模型，那么层次不会那么深入，从而只会占用很小的内存空间，并且其执行能力会根据正在进行的操作而表现得更好。

16.2.8 AUTO 选项

AUTO 选项采取的是与 RAW 不同的处理数据的方法。AUTO 选项尝试更好地进行格式

www.TopSage.com

化—— 基于表(或者表的别名)来命名元素。此外，AUTO 选项认识到数据可能会有一些潜在的层次关系方面的概念，这些应该在 XML 中表现出来。

现在回到 16.2.7 节介绍的客户订单的示例。这次将使用 AUTO 选项，所以可以看一下与使用 RAW 选项所得到相对普通的输出结果不同的地方。也可以广泛使用别名来使元素具有更具现实意义的名称：

```
USE AdventureWorks2008;
```

```
SELECT Customer.CustomerID,
 Customer.AccountNumber,
 [Order].SalesOrderID,
 CAST([Order].OrderDate AS date) AS OrderDate
FROM Sales.Customer Customer
JOIN Sales.SalesOrderHeader [Order]
 ON Customer.CustomerID = [Order].CustomerID
WHERE Customer.CustomerID = 29890 OR Customer.CustomerID = 30067
FOR XML AUTO;
```

首先，最明显的区别在于元素名称变成了作为数据源的表的名称或是该表的别名。还可以注意到能够通过方括号限定的方式输出包含了 SQL Server 关键字 Order 的 XML。如果更彻底地观察该 XML，还会发现另一个更显著的区别(为了清晰起见这里再次整理了一下输出)：

```
<Customer CustomerID="1" AccountNumber="AW00000001">
 <Order SalesOrderID="43860" OrderDate="Aug 1 2001" />
 <Order SalesOrderID="44501" OrderDate="Nov 1 2001" />
 <Order SalesOrderID="45283" OrderDate="Feb 1 2002" />
 <Order SalesOrderID="46042" OrderDate="May 1 2002" />
</Customer>
<Customer CustomerID="2" AccountNumber="AW00000002">
 <Order SalesOrderID="46976" OrderDate="Aug 1 2002" />
 <Order SalesOrderID="47997" OrderDate="Nov 1 2002" />
 <Order SalesOrderID="49054" OrderDate="Feb 1 2003" />
 <Order SalesOrderID="50216" OrderDate="May 1 2003" />
 <Order SalesOrderID="51728" OrderDate="Aug 1 2003" />
 <Order SalesOrderID="57044" OrderDate="Nov 1 2003" />
 <Order SalesOrderID="63198" OrderDate="Feb 1 2004" />
 <Order SalesOrderID="69488" OrderDate="May 1 2004" />
</Customer>
```

源自第二个表的数据(由 SELECT 列表决定)嵌套在来自于第一个表的数据中。在本例中，Order 元素嵌套在 Customer 元素中。如果 Order 表中的列位于选择列表中的第一个位置，Customer 表就会嵌套在 Order 表中。

注意：

注意 SELECT 列表的次序！想清楚 XML 查询所要解决的主要问题。排列一下 SELECT 列表使得它能产生适当的 XML 结果。您当然可以把它变成不同的样式——但是如果 SQL Server 一开始就为您生成了合适的结果，又何必要改变呢？

使用 AUTO 的缺点在于 XML 数据模型的结果会稍微复杂一些。优点在于数据更为清楚地

www.TopSage.com

分为层次结构模型。这就使得当元素成为更重要的断点时——比如有一个双排序报表(如在 Customer 表中对 Order 表进行排序)，事情变得较易处理。

16.2.9 EXPLICIT 选项

EXPLICIT 选项很有意思——在尝试创建查询时，该选项会大致描述可能使用的语言。您需要花更多的精力去准备 EXPLICIT 选项，但是也非常值得，因为该选项会对什么是元素和属性，以及什么样的元素可以嵌套在其他的元素中提供非常细粒度的控制。EXPLICIT 选项可以定义层次结构的每个层次和样式。要定义层次结构，需要建立所谓的通用表(universal table)。在许多方面，通用表和 SQL Server 中产生的结果集都很相似。通常可通过每次使用 UNION 语句结合一个层次来产生结果，但是也可以在 UDF 中建立很多数据，然后对数据进行 SELECT 操作来产生最终的 XML。通用表和传统的结果集之间最大的区别在于必须在结果集中提供足够的元数据，这样 SQL Server 才能根据所期望的架构将结果集转换为 XML 文档。

足够的元数据是指什么呢？为了说明这些数据的复杂程度，下面给出了一个真正的通用表，如表 16-2 所示。

表 16-2

Tag	Parent	Customer!1!CustomerID	Customer!1!CompanyName	Order!2!OrderID	Order!2!OrderDate
1	NULL	ALFKI	Alfreds Futterkiste	NULL	NULL
2	1	ALFKI	Alfreds Futterkiste	10643	1997-08-25 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	10692	1997-10-03 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	10702	1997-10-13 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	10835	1998-01-15 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	10952	1998-03-16 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	11011	1998-04-09 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	11078	1999-05-01 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	11079	NULL
2	1	ALFKI	Alfreds Futterkiste	11080	2000-07-22 16:48:00.000

www.TopSage.com

(续表)

Tag	Parent	Customer!1!CustomerID	Customer!1!CompanyName	Order!2!OrderID	Order!2!OrderDate
2	1	ALFKI	Alfreds Futterkiste	11081	2000-07-22 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	11087	2000-08-05 17:37:52.520
1	NULL	ANTON	Antonio Moreno Taquería	NULL	NULL
2	1	ANTON	Antonio Moreno Taquería	10365	1996-11-27 00:00:00.000
2	1	ANTON	Antonio Moreno Taquería	10507	1997-04-15 00:00:00.000
2	1	ANTON	Antonio Moreno Taquería	10535	1997-05-13 00:00:00.000
2	1	ANTON	Antonio Moreno Taquería	10573	1997-06-19 00:00:00.000
2	1	ANTON	Antonio Moreno Taquería	10677	1997-09-22 00:00:00.000
2	1	ANTON	Antonio Moreno Taquería	10682	1997-09-25 00:00:00.000
2	1	ANTON	Antonio Moreno Taquería	10856	1998-01-28 00:00:00.000

EXPLICIT 选项仅用于极为复杂的情况中。在许多情况下，使用 PATH 选项会比 EXPLICIT 选项执行起来容易得多。总的来说，在所有选项中，EXPLICIT 选项应最后考虑—— 该选项非常高级，难于理解，有关 EXPLICIT 选项更深入的讨论超出了本书范围。

16.2.10　PATH 选项

这里将要介绍一种更为实际的获取数据的 XML 方法。

虽然 EXPLICIT 选项还未废弃，但是和 EXPLICIT 选项相比，PATH 选项才是用以解决以往只能用 EXPLICIT 选项来解决的问题的更好方法。PATH 选项在很多方面都很有意义，在大多数情况下，建议您采用这种方法处理复杂的 XML 输出。

注意：

这个建议比看上去更为复杂。Microsoft 的思路是：PATH 选项更加简单。是的，PATH 选项在多数时候确实更为简单，但很快就会发现该选项也有例外情况，往往使人们绞尽脑汁来确

www.TopSage.com

定到底该怎么办。简而言之，在某些情况下，如果不知道 XPath，那么 EXPLICIT 选项其实更为简单。问题是，如果处理的是 XML，就应该学习 XPath，所以，当您逐渐了解它时，就会发现基于 XPath 的方法更有用。

但是，如果需要与 SQL Server 2000 保持向后兼容，那么应该使用 EXPLICIT 选项。

坦率地说，PATH 选项不算很差。所以，这里将首先关注一些 PATH 选项的基本使用方法，接下来介绍一些略为复杂的内容和 PATH 选项需要提供的内容。

1. PATH 101

使用 PATH 选项，就会得到一个按现有标准定位数据的模型——XPath。Xpath 有一个公认的标准，并提供指向 XML 架构中特定点的方法。对于 PATH 选项来说，只是使用了很多相同的规则和想法来说明在 XML 中应如何对待数据。

PATH 选项如何对待引用的数据取决于许多的规则，其中包括是否对列进行了命名(和 EXPLICIT 选项相同，如果使用了别名，那么别名就是列名)。如果列确实有名称，那么会有大量适当的额外规则作用于其上。

下面看一下一些可能的情况。

注意：

XPath 本身内容也很多，有很多书用了整本书的篇幅来介绍它。PATH 利用了 XPath 中多种可用的功能，所以用一章内容无法作全面介绍。这里将只介绍基础内容，在下一节中将讨论一些更为高级的内容。然后，您可自行决定是否想学习更多关于 XPath 和 PATH 的内容。有关这一高级内容的更多介绍可参见本系列的下一本书：*Professional SQL Server 2008 Programming*。

(1) 未命名列

在行元素中会把未命名列的数据作为原始文本来对待。为了说明这点，这里对介绍 XML RAW 时所使用的示例进行一些修改，把感兴趣的两个客户和他们的订单数量列出来：

```
SELECT CustomerID, COUNT(*)
FROM Sales.SalesOrderHeader Orders
WHERE CustomerID = 29890 OR CustomerID = 30067
GROUP BY CustomerID
FOR XML PATH;
```

检查一下输出：

```
<row><CustomerID>29890</CustomerID>8</row>
<row><CustomerID>30067</CustomerID>12</row>
```

这为查询中的每一行创建了一个行元素——和在 RAW 中是相同的——但是请注意这里对待列数据的方式是不同的。

因为对 CustomerID 列进行了命名，所以该列将作为自身的一个元素(在下一小节中将更深入地介绍)——但是，请注意结果中的数字 8。这只是行元素中松散嵌入的文本——因为该文本是在 CustomerID 元素的外部，所以该文本和 CustomerID 甚至没有直接的关联。

www.TopSage.com

注意：

需要多次申明的是，返回的实际计数(在本例中是 8 和 12)取决于对 SalesOrderHeader 表中的数据所影响的程度。关键是要看这个计数如何和 CustomerID 列的不相关，而不只是要查看和行相关的原始文本。

个人感觉，将数据作为顶级元素层的松散文本，其适用范围是相当有限的。规则的确允许这么做，但它未命名列适合于不明确的数据。不过，这就是其工作方式——所以当能满足特定的系统需求时，可以使用未命名列。

(2) 已命名列

这里问题将变得更为复杂。在最简形式下，已命名列和未命名列一样简单——在前面的示例中已经了解了这种情况。如果某列是一个简单的命名列，那么在使用 PATH 选项时，该命名列只是作为行的额外元素被添加。

```
<CustomerID>30067</CustomerID>12</row>
```

CustomerID 列就是一个简单的命名列。

但是可以向列名称中添加特殊的字符，来表明想要让该列有特殊的行为。下面介绍其中最重要的一些内容。

1) @符号

这并不是一个印刷错误——@符号是本节的标题。如果对列名称添加一个@符号，那么 SQL Server 会将该列作为前面列的一个属性来对待。注意需要将别名限定在单引号中以隐藏@符号(它通常是变量的指示符)。这里移动 CustomerID 使其成为该行的顶层元素的一个属性：

```
SELECT CustomerID AS '@CustomerID', COUNT(*)
FROM Sales.SalesOrderHeader Orders
WHERE CustomerID = 29890 OR CustomerID = 30067
GROUP BY CustomerID
FOR XML PATH;
```

这将产生如下结果：

```
<row CustomerID="29890">8</row>
<row CustomerID="30067">12</row>
```

可注意到，订单计数仍然是该行的一个文本元素——这里只把标识为属性的列移入顶层元素中。下一步将通过对计数进行命名并添加前缀来使 OrderCount 成为属性：

```
SELECT CustomerID AS '@CustomerID',
       COUNT(*) AS '@OrderCount'
FROM Sales.SalesOrderHeader Orders
WHERE CustomerID = 29890 OR CustomerID = 30067
GROUP BY CustomerID
FOR XML PATH;
```

这样就不再会有元素的松散文本：

www.TopSage.com

```
<row CustomerID="29890" OrderCount="8"/>
<row CustomerID="30067" OrderCount="12"/>
```

同样应该注意到，SQL Server 是足够智能的，它知道所有的内容都包含在属性中——没有更低层的元素或简单的文本，SQL Server 选择使其变成一个自动结束的标记(可以看到元素的最后有一个“/”)。

为什么说这里面很有技巧呢？因为这里有很多不同的限制性规则。为了演示这一点，可对原来的查询做一些简单的修改。这个示例看上去能够运行，但是如果运行的话，SQL Server 会报错：

```
SELECT CustomerID,
       COUNT(*) AS '@OrderCount'
FROM Sales.SalesOrderHeader Orders
WHERE CustomerID = 29890 OR CustomerID = 30067
GROUP BY CustomerID
FOR XML PATH;
```

这里所做的是让 CustomerID 返回为自身的一个元素。一开始，您可能以为能得到一个 CustomerID 元素以及一个 OrderCount 作为属性，但实际运行情况却不是这样：

```
消息 6852，级别 16，状态 1，第 1 行
在 FOR XML PATH 的 XML 层次结构中，以属性为中心的列'@OrderCount'不能出现在不是以属性为中心的同级成员之后。
```

这里的问题在于不知道要使 OrderCount 成为谁的属性。它是行的属性，还是 CustomerID 的属性呢？

2) /符号

是的，这是一个正斜杠。

和@很相似，这个特殊字符指出了想完成的特殊任务。本质上说，它用于定义路径——将元素和属于该元素的内容进行关联的层次结构。除了作为第一个字符以外，该符号可以出现在列名称中的任何位置。为了说明这点，这里使用上一个(失败的)示例，用该符号来修正错误。

首先，需要修改 OrderID 来取得它所属元素的信息：

```
SELECT CustomerID,
       COUNT(*) AS 'CustomerID/OrderCount'
FROM Sales.SalesOrderHeader Orders
WHERE CustomerID = 29890 OR CustomerID = 30067
GROUP BY CustomerID
FOR XML PATH;
```

通过添加“/”，然后在斜杠之前放入 CustomerID，就告诉了 SQL Server 在层次结构中 OrderCount 是在 CustomerID 下面。现在可以用很多种方法构造 XML 层次结构，这里来看一下 SQL Server 的做法：

```
<row><CustomerID>29890<OrderCount>8</OrderCount></CustomerID></row>
<row><CustomerID>30067<OrderCount>12</OrderCount></CustomerID></row>
```


www.TopSage.com

现在，回想一下，目的是使 OrderCount 成为 CustomerID 的属性，虽然在层次结构中，OrderCount 在 CustomerID 下面，但这仍然不是想要的结果。为实现这一点，可以将/和@结合起来，但是需要定义整个层次结构。由于考虑到这会有点让人困惑，所以这里分两步进行——首先，可以尝试如下操作，但是这会产生和前面的示例相似的错误。

```
SELECT CustomerID,
       COUNT(*) AS 'CustomerID/@OrderCount'
FROM Sales.SalesOrderHeader Orders
WHERE CustomerID = 29890 OR CustomerID = 30067
GROUP BY CustomerID
FOR XML PATH;
```

错误如下所示：

```
消息 6852，级别 16，状态 1，第 1 行
在 FOR XML PATH 的 XML 层次结构中，以属性为中心的列'CustomerID/@OrderCount' 不能出现在不是以属性为中心的同级成员之后。
```

为了解决这个问题，需要在建立 XML 标记时了解事物的结构是怎样的。关键在于这些标记实际上是按照列表的顺序建立的。所以，如果想要为元素添加属性，需要记住属性是元素标记的一部分——这意味着需要在定义该元素(子元素或者是原始文本)的任何内容之前先对所有的属性进行定义。

在本例中，CustomerID 将作为原始文本，但是 OrderCount 将作为属性(现实中的情况可能正好相反，但这里只是举例)。这意味着以相反的方式通知 SQL Server。当 SQL Server 发现 OrderCount 信息时，OrderCount 已经成为了 CustomerID 的属性，且不能返回。

所以，为了解决这个问题，只要在向 SQL Server 通知任何更多元素或原始文本之前先通知属性即可：

```
SELECT COUNT(*) AS 'CustomerID/@OrderCount',
CustomerID
FROM Sales.SalesOrderHeader Orders
WHERE CustomerID = 29890 OR CustomerID = 30067
GROUP BY CustomerID
FOR XML PATH;
```

这看上去可能有点不对劲，但是请再次想一下写入的顺序。首先写的是属性，而且只有在此之后才能写入 CustomerID 元素中的低层信息。运行之后，将得到想要的结果：

```
<row><CustomerID OrderCount="8">29890</CustomerID></row>
<row><CustomerID OrderCount="12">30067</CustomerID></row>
```

现在 OrderCount 已经移入到期望的属性位置，而实际的 CustomerID 仍然是嵌套在元素内的原始文本。

注意：

请遵循所请求的顺序逻辑，因为该顺序逻辑在大多数情况下都可行。如果想要 CustomerID 也成为属性而非原始文本，又不想让 CustomerID 位于 OrderCount 之后，那么可以这样操作——

www.TopSage.com

只要确保 CustomerID 在 OrderCount 的定义之后即可。

3) 补充内容

如前所述，XPath 本身很复杂而且值得用一本书来作介绍，但是这里还想再补充些内容。

@和/符号提供了很多的灵活性，从而有助于按自己的想法构建 XML 的输出，并满足大多数简单应用程序的需要。但是，如果需要更多功能，那么有很多选择。例如：

- 可以使用通配符，它和文本数据一起运行，但不会被当作是单独的列。
- 可在 XML 数据类型列中嵌入本地 XML 数据。
- 使用 XPath 节点测试——这些是能够改变数据行为的特殊 XPath 指令。
- 使用 data()指令以允许多个值作为 XML 中的一个数据点同时运行。
- 使用名称空间。

16.2.11 OPENXML 函数

到目前为止，本章中所介绍的许多概念都属于 SQL Server 的高级内容。OPENXML 函数就更为高级了，因此这里不会深入介绍。但本书仍想确保您了解 OPENXML 函数的作用和使用场合。请记住，许多需要创建 OPENXML 函数来完成的操作现在都可以用一种更本地化的方法进行处理，只要将 XML 放入本地 XML 数据类型中并使用本章前面讨论的 XML 类型方法即可。

当最初的 XML 功能集首次引入到 SQL Server 2000 中时，还不存在本地的 XML 数据类型。FOR XML 能将关系数据转换为 XML，但还需要将 XML 转换为关系形式——这就要用到 OPENXML 函数。

OPENXML 是一个行集函数，它会像其他行集函数(如 OPENQUERY 和 OPENROWSET)那样打开字符串。这意味着可以连接一个 XML 文档，甚至是通过 INSERT…SELECT 或 SELECT INTO 把 XML 文档作为输入数据的源。主要的区别在于 OPENXML 函数需要用到几个系统存储过程以准备文档并且在用完之后释放内存。

为了创建文档，可以使用 sp_xml_preparedocument 存储过程。这会将字符串放入内存中并且对文档进行预分析处理以获得最优的查询性能。XML 文档会一直存在于内存中，直到显式地删除 XML 文档或者终止调用 sp_xml_preparedocument 的连接。语法也很简单：

```
sp_xml_preparedocument @hdoc = <integer variable> OUTPUT,
[, @xmltext = <character data>]
[, @xpath_namespaces = <url to a namespace>]
```

说明：

如果要提供名称空间的 URL，则需要把它包含在“<”和“>”符号中(例如，<root xmlns:sql =run: schemas-microsoft-com:xml-sql>)。

这个存储过程的参数是如下自描述的。

- @hdoc：如果以前使用过 Windows API(或者别的方法，但 API 是比较常用的)进行编程，那么您可能已经接触过这个“h”——这是匈牙利命名法对句柄(handle)的表示。句柄是一个指向内存块的指针，内存中驻留着一些内容(可以是任何内容)。在本例中，这是请求 SQL Server 分析和保持的 XML 文档的句柄。这是一个输出变量——这个引用的变

www.TopSage.com

量会在存储过程返回之后包含 XML 的句柄——请确保已经存储了该句柄，当使用 OPENXML 函数时，需要用到该句柄。

- @xmltext：顾名思义——想要对其进行分析和处理的真正的 XML。
- @xpath_namespaces：正确处理 XML 所需的任意名称空间的引用。

在调用完这个存储过程并保存文档句柄之后，就可以使用 OPENXML 函数。它的语法稍微有些复杂：

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

前面已经讨论过句柄——这是一个整数值，该值将作为存储过程 sp_xml_preparedocument 调用所返回的输出参数。

当调用 OPENXML 函数时，必须为将作为所有查询起始点的节点提供路径。架构声明通过相对于这里设置的基节点导航来引用 XML 文档的所有部分。

接下来介绍映射标志。这些标志将有助于决定在 OPENXML 结果中究竟是使用元素还是属性。这些选项包括以下内容，如表 16-3 所示。

表　16-3

字 节 值	说　　明
0	除了不能把它和 2 或 8 相结合(2+0 还是 2)外，它和 1 相同。这是默认选项
1	除非与下面的 2 结合，否则只使用属性。如果没有指定名称的属性，则返回 NULL。可以将该选项与 2 或 8(或者两者)相结合来组合行为，但是该选项比选项 2 优先。如果 XPath 找到了名称相同的属性和元素，那么将选择属性
2	除非与上面的 1 相结合，否则只使用元素。如果没有指定名称的元素，则返回 NULL。可以将该选项与 1 或 8(或者两者)相结合来组合行为。如果和 1 相结合，那么如果存在属性，则会映射属性。如果没有属性，那么将使用元素。如果没有元素存在，则返回 NULL
8	可以和上面的 1 或 2 相结合。使用的数据不能复制到溢出属性@mp:xmltext 中(必须使用 MetaProperty 架构项来检索)。您可能不准备使用 MetaProperties——大多数时候也不会这么做——但这里还是建议使用这个选项。它占用很少(非常之少)的操作系统开销

最后要介绍的是架构和表。如果在定义架构而又对 XPath 不熟悉，那么这部分的内容会比较棘手。幸运的是，这个特殊的 Xpath 的使用不是很复杂(原理和 Windows 中的目录很相似)。

架构会因声明的方式不同而有所不同。其定义如下所示：

```
WITH (
<Column Name> <data type> [{<Column XPath>|<MetaProperty>}]
[,<Column Name> <data type> [{<Column XPath>|<MetaProperty>}]
...
```

- 列名(column name)就是检索的属性或元素的名称。当建立 SELECT 列表、执行 JOIN 等操作时，列表将作为引用的名称。

www.TopSage.com

- 数据类型(data type)可以是任何有效的 SQL Server 数据类型。因为 XML 可以有与 SQL Server 不等价的数据类型，所以如果有必要，则会发生自动的强制转换，但这通常是可预见的。
- XPath 列是将节点映射到列的 XPath 架构(相对于作为 OPENXML 函数起始点建立的节点)——使用元素还是属性取决于表 16-3 中说明的标志参数。如果不使用它，那么 SQL Server 就会假设您想要将当前节点定义为 OPENXML 语句中的起始点。
- MetaProperties 是一组特殊变量，可以在 OPENXML 查询中引用这些变量。这些变量描述了感兴趣的 XML DOM 的各个方面。要使用这些变量，只要用单引号把它们括起来并放入 XPath 列中。可用的 MetaProperties 变量包括以下几个。
  - @mp:id——不要将该选项与 EXPLICIT 选项中的 XML id 混淆。虽然这个属性提供了相似的功能，但是它是 DOM 节点中的唯一标识符(在文档的作用域内)。区别在于这个值由系统生成——同样地，可以确保该值的存在。该变量保证了只要文档仍然在内存中，就可以引用相同的 XML 节点。如果 id 是 0，那么根节点就是根节点(它的@mp:parentid 属性为 NULL，下面将介绍该属性)。
  - @mp:parentid——除了针对父节点以外，该变量和@mp:id 功能相同。
  - @mp:localname——提供了非完全限定的节点名称。该变量带有前缀和名称空间 URI(Uniform Resource Identifier，统一资源标识符——通常会看到它以 URN 打头)一起使用来对元素或属性节点命名。
  - @mp:parentlocalname——除了针对父节点以外，该变量和@mp:localname 功能相同。
  - @mp:namespaceuri——提供了当前元素的名称空间 URI。如果这个属性的值为 NULL，那么就没有任何名称空间。
  - @mp:parentnamespaceuri——除了针对父节点以外，该变量和@mp:namespaceuri 功能相同。
  - @mp:prefix——存储了当前元素名称的名称空间前缀。
  - @mp:prev——存储相对于节点的前一个兄弟节点的 mp:id。可以使用该变量来说明当前层次结构中的元素的次序。例如，如果@mp:prev 的值为 NULL，那么位于该层的第一个节点。
  - @mp:xmltext——这个 MetaProperty 用于处理目的，并且包含了当前元素的实际 XML 内容。

当然，也可以省略以上所有参数从而节省大量工作。如果有一个表和在 XML 中指定的 XPath 起始点直接相关(名称或数据类型)，那么就可以省略这些参数。如果真有这样一张表，那么可以对表命名，SQL Server 会完成转换的任务。

需要处理的内容很多，现在还没有结束。可以看到，当用完 XML 文档时，需要调用存储过程 sp_xml_removedocument 来清除用于存储该 XML 文档的内存。语法非常简单：

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

注意：

这里需要强调的是，保持良好的清除习惯非常重要。SQL Server 会帮您完成清除工作，但

www.TopSage.com

是不能每次都指望它。只有当您终止连接时，SQL Server 才会清除，但是如果使用的是连接池，那该怎么办呢？如果系统处于低负载，一些连接可能会永远存在。这是个很容易执行的存储过程，所以请务必每次都使用它。

您可能已经迫不及待地想知道该存储过程的用法——现在就介绍一个非常重要的示例。

假设您的公司与另一家公司合并，现在需要把对方的一些数据导入到您的系统中。在这个示例中，假设要导入一些对方有的、而我方没有的 shippers 表的数据。将行记录导入到在第 5 章中创建的 Accounting 数据库中(要使其正常运行，shippers 表是必须的——在第 5 章的代码中可找到其创建语法)。从 XML 文档中导入这些数据的脚本如下所示：

```
USE Accounting;

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

-- define the XML document
SET @xmldoc = '
<ROOT>
<Shipper ShipperID="100" CompanyName="Billy Bob&apos;s Great Shipping"/>
<Shipper ShipperID="101" CompanyName="Fred&apos;s Freight"/>
</ROOT>
';

--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 Shippers;

-- ShipperID is an IDENTITY column, so we need to allow direct updates
SET IDENTITY_INSERT Shippers ON

--See our XML data in a tabular format
SELECT * FROM OPENXML (@idoc, '/ROOT/Shipper', 0) WITH (
   ShipperID     int,
   CompanyName   nvarchar(40));

--Perform and insert based on that data
INSERT INTO Shippers
(ShipperID, ShipperName)
SELECT * FROM OPENXML (@idoc, '/ROOT/Shipper', 0) WITH (
   ShipperID              int,
   CompanyName            nvarchar(40));

--Set things back to normal
SET IDENTITY_INSERT Shippers OFF;

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

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


www.TopSage.com

最终的结果集看上去和期望的结果很相似：

```
ShipperID       ShipperName                    DateInSystem
-----------      ------------------------       ---------------------
1               United Parcel Service          2008-11-05 18:51:42.673

(1 行受影响)

ShipperID       CompanyName
-----------      ----------------------------------------
100             Billy Bob's Great Shipping
101             Fred's Freight

(2 行受影响)

(2 行受影响)

ShipperID       ShipperName                    DateInSystem
-----------     ---------------------------    ----------------------
1               United Parcel Service          2008-11-05 18:51:42.673
100             Billy Bob's Great Shipping     2008-11-09 20:56:29.177
101             Fred's Freight                 2008-11-09 20:56:29.177

(3 行受影响)
```

现在有了另一种获得 XML 查询结果的方法，和一种将 XML 转为关系数据的方法。请记住，OPENXML 函数还有许多其他参数和功能可供您在 SQL Server 的学习过程中进行研究。

16.3 XSLT 简介

这里将开始介绍本章最后的、但并非最复杂的内容。可扩展样式表语言转换(Extensible Stylesheet Language Transformations，XSLT)有点像是 XML 中的附加功能，XSLT 大大提升了 XML 的功能。使用 XSLT，可以把 XML 文档转换成其他的形式。

为了能够快速了解这方面的内容，先来看一下从 Microsoft 中原有的 Northwind 数据库生成的一个 XML 文档。

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
 <Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
         <Products ProductID="28" ProductName="Rössle Sauerkraut"/>
         <Products ProductID="39" ProductName="Chartreuse verte"/>
         <Products ProductID="46" ProductName="Spegesild"/>
 </Customer>
 <Customer CustomerID="BLONP" CompanyName="Blondesddsl père et fils">
         <Products ProductID="28" ProductName="Rössle Sauerkraut"/>
         <Products ProductID="29" ProductName="Thüringer Rostbratwurst"/>
         <Products ProductID="31" ProductName="Gorgonzola Telino"/>
         <Products ProductID="38" ProductName="Côte de Blaye"/>
         <Products ProductID="39" ProductName="Chartreuse verte"/>
         <Products ProductID="41" ProductName="Jack&apos;s New England
```


www.TopSage.com

```
            Clam Chowder"/>
          <Products ProductID="46" ProductName="Spegesild"/>
          <Products ProductID="49" ProductName="Maxilaku"/>
  </Customer>
</root>
```

这里得到的是 XML 的层次结构。在本例中，会碰到客户订购不同产品的情况。XML 文档告诉我们客户购买了哪些产品。这个问题看上去还算简单。

注意：

这是一个纯粹虚拟的示例——您并不需要真的在这里加载一个 Northwind 示例。

现在需要加一点难度——如果问题是“订购每件产品的客户分别是谁”，那该怎么办呢？我们的视角将大为改变。这时，更好的层次结构是产品在外层，而客户在内层。在这种情况下，客户(而不是产品)会多次出现(一次对应一个产品)，但会更接近问题的本质。

将 XML 与 XSL 转换相结合是很简单的。可以看到，无需对数据进行修改——只要转换 XML 文档，就可以以另一种不同的方式观察数据。

注意：

不要以为上面说的“看数据的方式”是指视觉方式——这其实是指如何去理解该数据。这部分是指如何准备将以特殊的方式使用的数据。当客户位于层次结构的顶层时，数据似乎并没有为解决以产品为中心的问题做好准备。这里需要将数据转换为以产品为中心，就像前面的那个问题一样。

下面看一下同样的数据被转换为另一种样式的结果：

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
<Products ProductID="28" ProductName="Rössle Sauerkraut">
      <Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste"/>
      <Customer CustomerID="BLONP" CompanyName="Blondesddsl père et fils"/>
 </Products>
 <Products ProductID="29" ProductName="Thüringer Rostbratwurst">
      <Customer CustomerID="BLONP" CompanyName="Blondesddsl père et fils"/>
 </Products>
 <Products ProductID="31" ProductName="Gorgonzola Telino">
      <Customer CustomerID="BLONP" CompanyName="Blondesddsl père et fils"/>
 </Products>
 <Products ProductID="38" ProductName="Côte de Blaye">
      <Customer CustomerID="BLONP" CompanyName="Blondesddsl père et fils"/>
 </Products>
 <Products ProductID="39" ProductName="Chartreuse verte">
      <Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste"/>
      <Customer CustomerID="BLONP" CompanyName="Blondesddsl père et fils"/>
 </Products>
 <Products ProductID="41" ProductName="Jack&apos;s New England Clam Chowder">
      <Customer CustomerID="BLONP" CompanyName="Blondesddsl père et fils"/>
 </Products>
 <Products ProductID="46" ProductName="Spegesild">
      <Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste"/>
```


www.TopSage.com

```
        <Customer CustomerID="BLONP" CompanyName="Blondesddsl père et fils"/>
    </Products>
    <Products ProductID="49" ProductName="Maxilaku">
        <Customer CustomerID="BLONP" CompanyName="Blondesddsl père et fils"/>
    </Products>
</root>
```

同样，这是相同的数据——只是视角不同而已。

但是为什么要局限于 XML 呢？尽管对每一种转换类型都作介绍超出了本书的范围，重要的要理解 XSL 转换可以在多种格式之间转换。可以转换为其他 XML 布局，也可以完全转换成其他的格式，如 CSV 文件甚至是 Word 文档。

说明：

这里只是将 XSLT 作为一个概念进行介绍。SQL Server 几乎不会用到 XSLT—— 更关注于数据本身而不是数据的表示方式。这里的关键在于理解本质相同的数据可以格式化为多种不同的 XML 形式，而 XSLT 提供了将 XML 转换为其他 XML 布局和转换为非 XML 格式的方法。

16.4 本章小结

本章概述了 XML、DTD、架构以及 XPath 的内容。只用一章的内容来解释 XML 及其相关技术这样宽泛的话题是不够的，但是希望通过本章的学习，您对 XML 及其用法有一个最基本的认识。

XML 是近 10 年来对业界影响最大的技术之一。XML 提供了一种灵活的、可传输的描述数据的方法。描述方法的简化不仅便于进行网站开发，而且有助于实现业内渴望已久的后台信息交换。

www.TopSage.com

第17章

Reporting Services 概述

在编写完所有查询语句和运行完所有存储过程后，要使数据有用——使数据对终端用户可用，仍有一件重要的事要做。

报表是一件看似简单实则不易处理的事情。不能只是简单地堆砌数字，数字必须具有意义，可能的话，还需要能吸引听报告人的注意力。为了生成切实可用的报表，需要记住以下几点。

- **只使用适量的数据**。不要在一份报表中反映过多内容，但也不能过少。仅有一团混乱数字的报表会让读者很快失去兴趣，并且在生成后使用不了几次便会废弃。同样地，对一份空洞乏味的报表，人们只会看上一眼，然后想也不想地扔掉。因此，要在适量数据和正确数据之间找到平衡。
- **使其能引人注意**。报表中另一个重要元素就是所谓的“美观”——也就是说，报表要赏心悦目。一个难看的报表没有任何意义。

本章将介绍 Reporting Services 工具，该工具最先是在 SQL Sever 2000 中作为一种可下载的 Web 插件出现，后成为 SQL Sever 2005 产品的核心部分。和本书中介绍的所有 SQL Sever“插件”功能一样，由于篇幅有限，这里将对该工具作简要的介绍。如果有兴趣的话，那么可以考虑深入阅读更高层次的书籍，如 *Professional SQL Serve 2008 Programming*；或者阅读关于 Reporting Services 方面的专业书籍，如 *Professional SQL Server Reporting Services*。

17.1 Reporting Services 101

您可能已经生成过报表。它们可能是打印机打出来的纸质报表(可能是基本的 Access 报表——实际上这正是我最喜欢 Access 的地方之一)。它们也可能是由相当健壮的报表引擎生成，如 Crystal Report。即使没有使用过这种奇妙的工具，人们也知道用一个存储过程就可以打印输出非常简单的(外观不太好看)报表，本书当然同意这一观点。

但事实是，如今的经理和同事们总是期待更多。这时候就应引入 Reporting Services。Reporting Services 实际上有两种不同的操作类型。

www.TopSage.com

- **报表模型**：这一操作使用了相对简单的、Web 驱动的接口，允许终端用户自行创建简单报表。
- **在 Visual Studio 中生成报表**：这并不意味着一定要编写代码(实际上可以使用拖放功能来创建简单的报表——本章将给出示例)，但这样也能设计出相当健壮的报表。

请注意，虽然用户最终可以从相同的 Reporting Services Web 主机上访问这些报表，但是这些报表却是基于不同的体系结构(可以看出报表是以很多不同样式创建的)。

此外，Reporting Services 提供了预生成报表的功能(如果构成报表基础的查询需要一些时间运行，那么这是很有用的)，以及通过电子邮件分发报表的功能。

17.2 创建简单的报表模型

要创建简单的模型，首先打开 Business Intelligence Studio。

注意：

这和到目前为止一直使用的 SQL Server Management Studio 完全不同。Business Intelligence Studio 是面向高级开发人员(而不是管理员)的工作平台；它是一种只提供针对 SQL Server 提供的“额外”服务的项目模板的 Visual Studio 形式。除了本章中介绍到的 Business Intelligence Studio 的功能外，在第 18 章中使用 Integration Services 时也将用到该工具。

选择“文件”菜单中的“新建项目”命令，出现如图 17-1 所示的对话框。

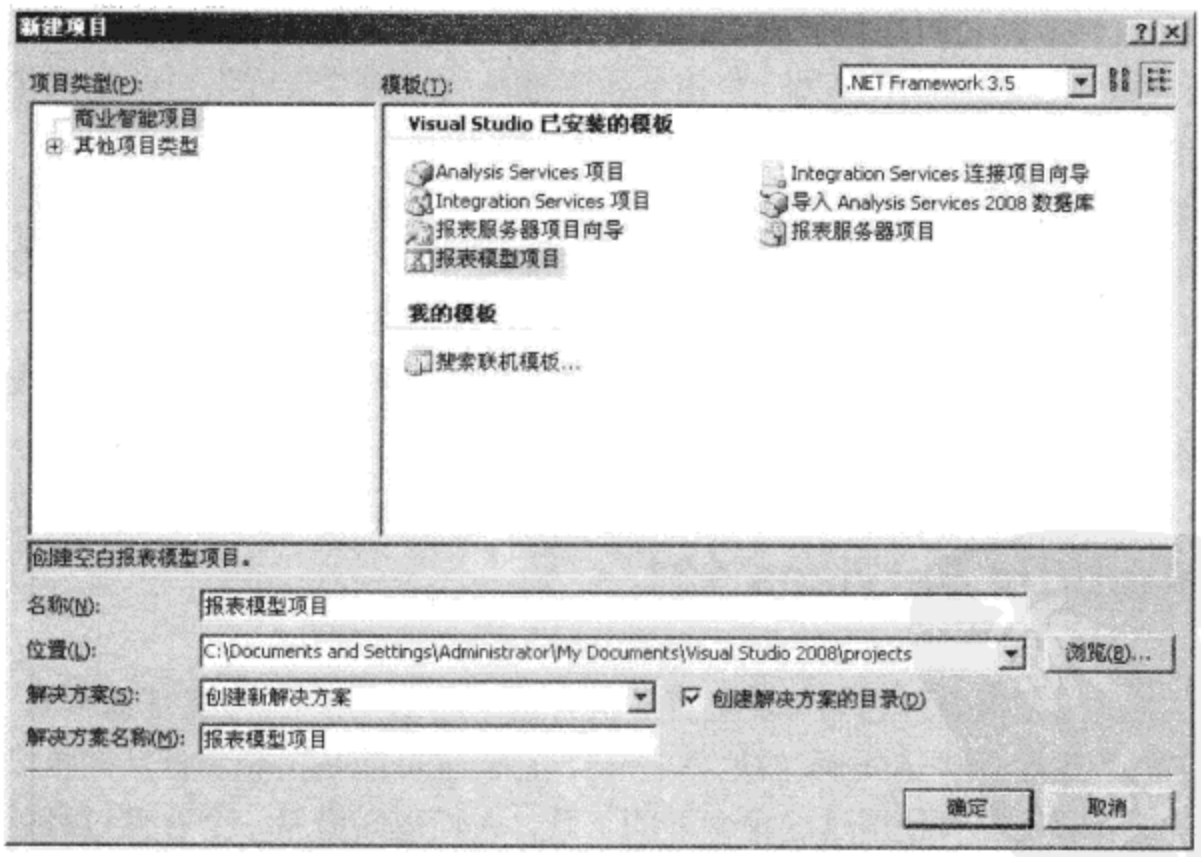


图 17-1

选择适当的名称(如 ReportModelProject)，单击“确定”按钮。这时进入我们非常熟悉的类似于 Visual Studio 开发环境的界面。

www.TopSage.com

说明：

图 17-1 所示的对话框外观可能会因 Visual Studio 安装与否有所不同，即使安装了 Visual Studio，也会随安装的特定语言和模板而发生变化。而该图只是反映最一般情况下的情形。

如果 Visual Studio 环境是默认配置，那么可以看到右上方的“解决方案管理器”面板。右击“数据源”节点并选择“添加新数据源”命令，如图 17-2 所示。

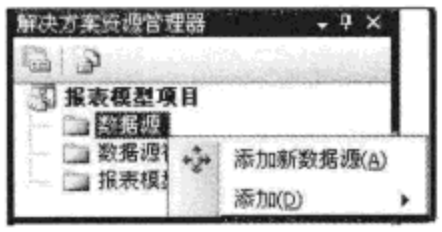


图　17-2

这时出现“数据源向导”的欢迎对话框(如若不要显示该界面，则应先对其设置，选中“不再显示此页”复选框)。单击“下一步”按钮进入实质性的部分——“数据源向导”对话框的数据源选择页面。然而这里有一个问题——还没有任何数据源(如图 17-3 所示)。

显然，没有任何现有的数据连接供选择，因此单击“新建”按钮。

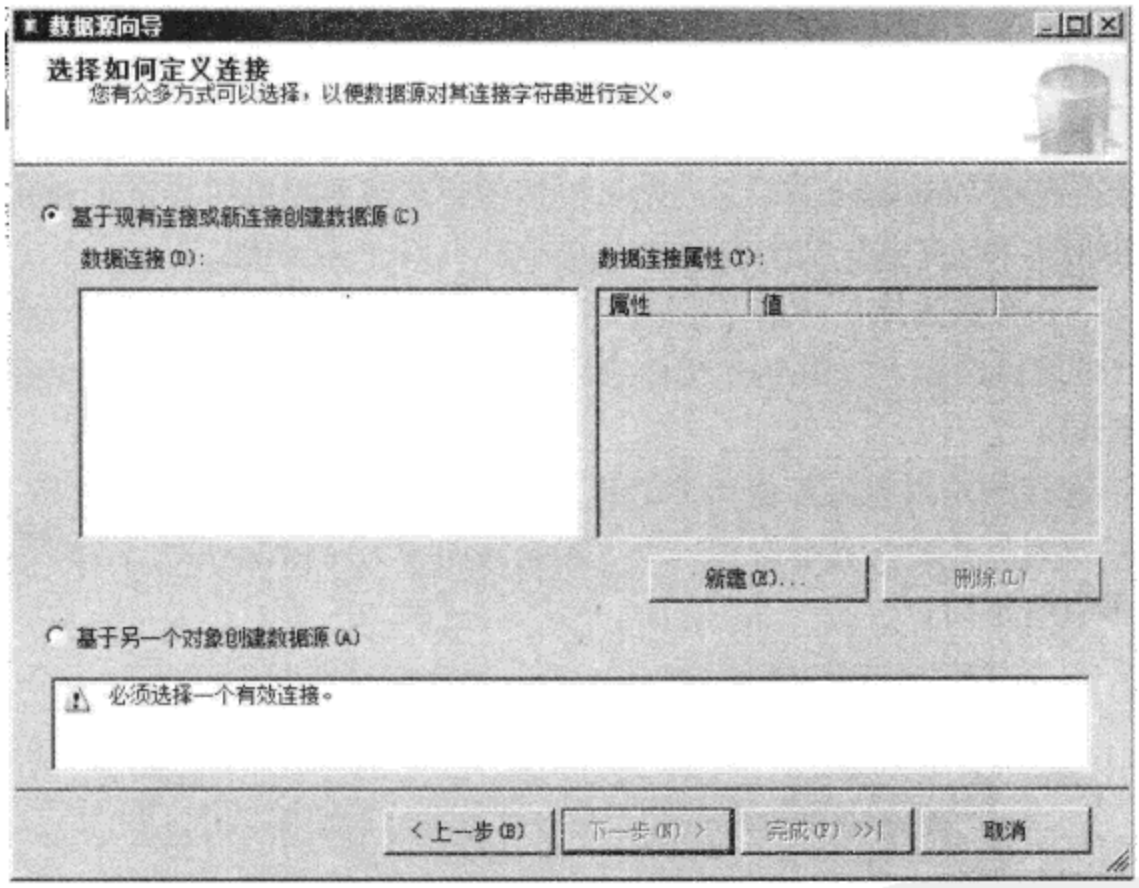


图　17-3

注意：

第一次看到下一个对话框时，您可能会惊讶地发现该对话框和 Management Studio 中反复使用的新建连接对话框有所不同；然而，它包含了相同的基本元素，只是外观上略有区别而已(简而言之，虽然外观有点不同，但是不用担心)。

进入图 17-4 所示的“连接管理器”对话框。

虽然这里的概念和本书其他地方提到的概念相同，但仍有几个新的概念，所以下面将介绍这个对话框中的几个重要的元素。

www.TopSage.com

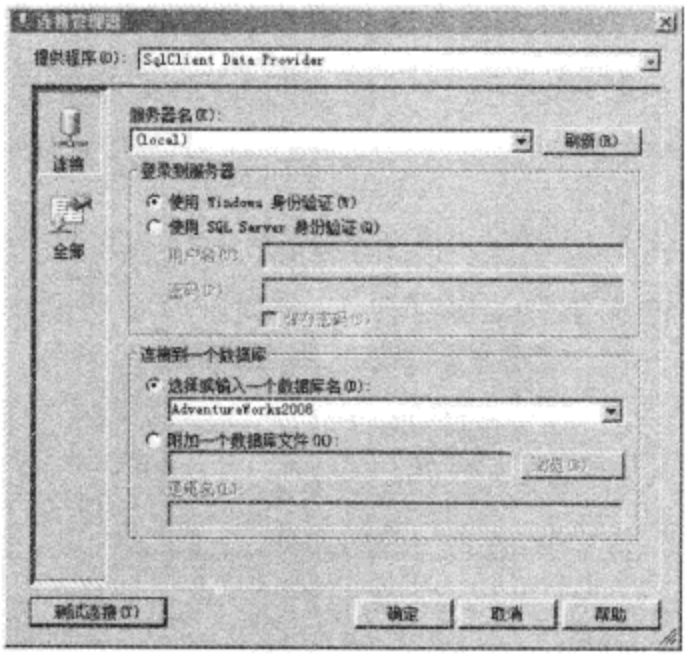


图 17-4

- **服务器名**：这和前面介绍的“服务器名”相同。对想要连接的服务器命名，或者如果是要连接本地服务器默认的 SQL Server 实例，则也可以使用别名如“local”或者“.”(简单的句点)。
- **使用 Windows/SQL Server 身份验证**：选择连接到服务器的身份验证类型。Microsoft(本书也一样)更希望使用“Windows 身份验证”，但是如果服务器不在域内或者是管理员未直接授权给 Windows 登录名，则可以使用管理员所提供的特定于 SQL Server 的用户名和密码(本书在某些章节中使用过 MyLogin 和 MyPassword)。
- **连接到下一个数据库**：事情变得越加有趣。这里既可以继续在选取的服务器上选择数据库，也可以选择直接与 mdf 文件相连接(这样一来，SQL Server Express 引擎会完成转换工作)。

在本例中，选择的是本地服务器和常用的 AdventureWorks2008 数据库。

继续单击“确定”按钮，将会看到一个“数据源向导”对话框(如图 17-5 所示)，这和第一次看到的对话框有些不同。

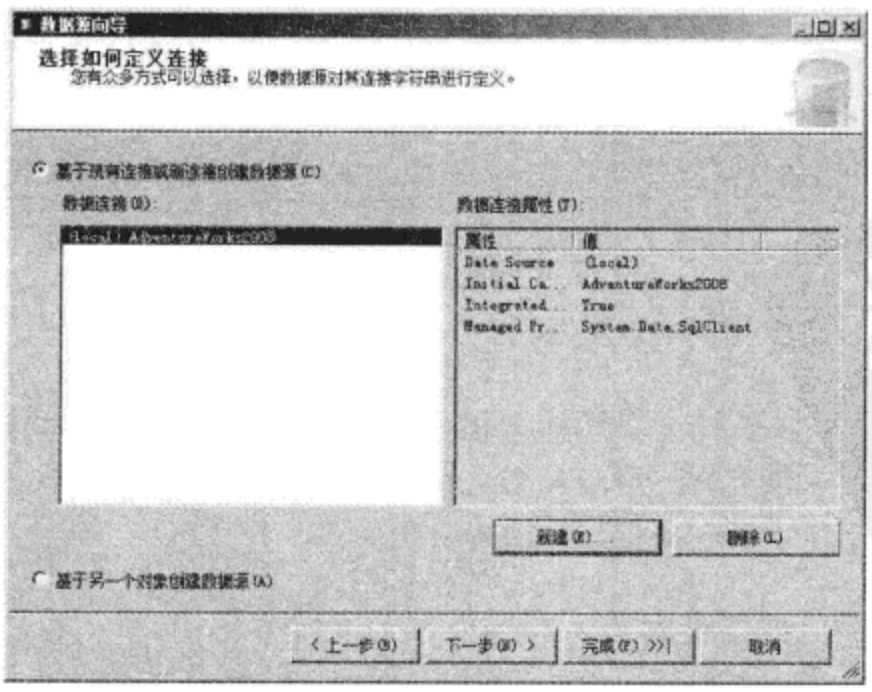


图 17-5

www.TopSage.com

本示例只有刚创建的连接，但是只要愿意，实际上是可以创建多个连接的，并可在其中进行选择。同时请注意在对话框的右边有数据连接属性的摘要信息。

单击“下一步”按钮，会出现图 17-6 所示的该向导的最终对话框。

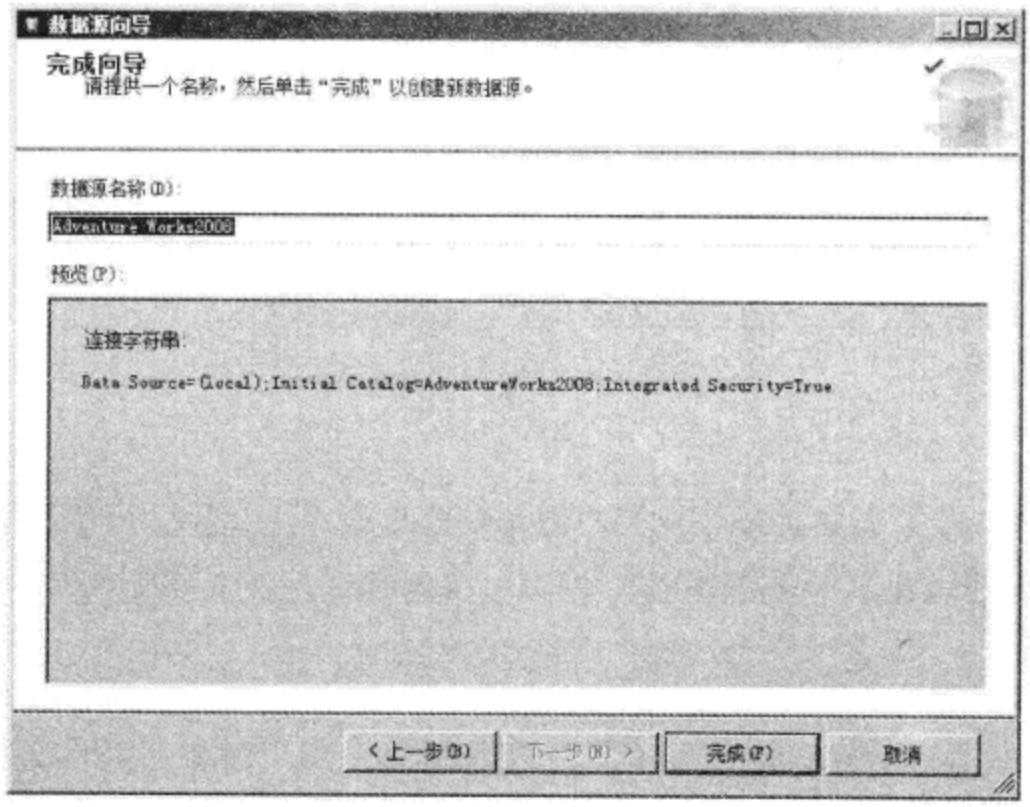


图　17-6

注意默认的名称碰巧是选择的数据库，但这其实只是数据源的一个名称而已——可以根据自己的需要任意命名这个数据源，它仍然连接着本地服务器上的 AdventureWorks2008 数据库。

注意：

还要注意它是如何构建连接字符串的。连接字符串是所有现代数据库连接中最基本的概念——基本上每个连接模型(.NET 托管提供程序，如 OLE DB、ODBC)都会在不同程度上使用连接字符串。如果选择了一些不同的选项——例如，使用 SQL Server 用户名和密码而不是 Windows 安全验证，那么连接字符串会使用一些不同的参数，而且会传递不同的值。

单击“完成”按钮，这时就有了本项目的数据源，如图 17-7 所示。

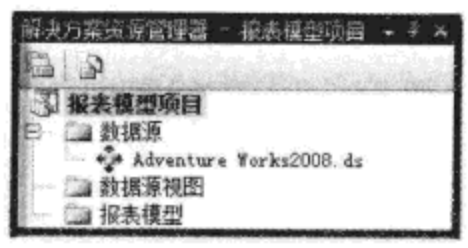


图　17-7

如果稍微思考一下图 17-7，您会注意到这里“报表项目模型”项目中的内容不只是数据源；事实上，现在便需要进入下一步工作，创建数据源视图。

17.2.1　数据源视图

数据源视图和第 10 章中学习的视图有点相似。特别是，数据源视图能够给用户提供访问

www.TopSage.com

数据的能力，能够对用户实际看到的数据进行过滤。如果让用户访问整个数据源，那么他们就能看到该数据源所有可用的数据。数据源视图可以将数据源过滤为可用对象原始列表的一个子集。

要添加数据源视图，只要右击“数据源视图”节点并选择“添加新数据源视图”命令，如图 17-8 所示。

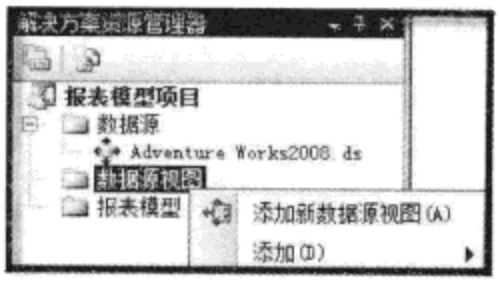


图 17-8

这时又出现了一个“数据源视图向导”对话框，单击“下一步”按钮进入以前看到过的对话框——它基本上和创建数据源的对话框相同(只是在外观上有一两处细微的差别)，如图 17-5 所示。

接受默认选项，然后单击“下一步”按钮，进入“选择表和视图”对话框，如图 17-9 所示。顾名思义，这个对话框允许为正在创建的报表模型的终端用户指定可用的表和视图。

通过选择 SalesOrderDetail 并使用“>”按钮来把可用对象移入到“包含的对象”列中。接下来单击“添加相关表”按钮并查看结果。SQL Server 会根据两个表之间的关系(基于定义的外键)来判断哪个表与选择的一个或多个表相关。在本例中，它会选择 SalesOrderHeader (SalesOrderDetail 表的父表)和 SpecialOfferProduct(SalesOrderDetail 也有 SalesOfferProduct 表的一个外键)。对于正在创建的报表而言，并不需要 SpecialOfferProduct 表，因此选中该表并通过“<”按钮将其移除。最后，添加 Customer 表、Person 表和 Product 表，使得结果与图 17-9 中一样。

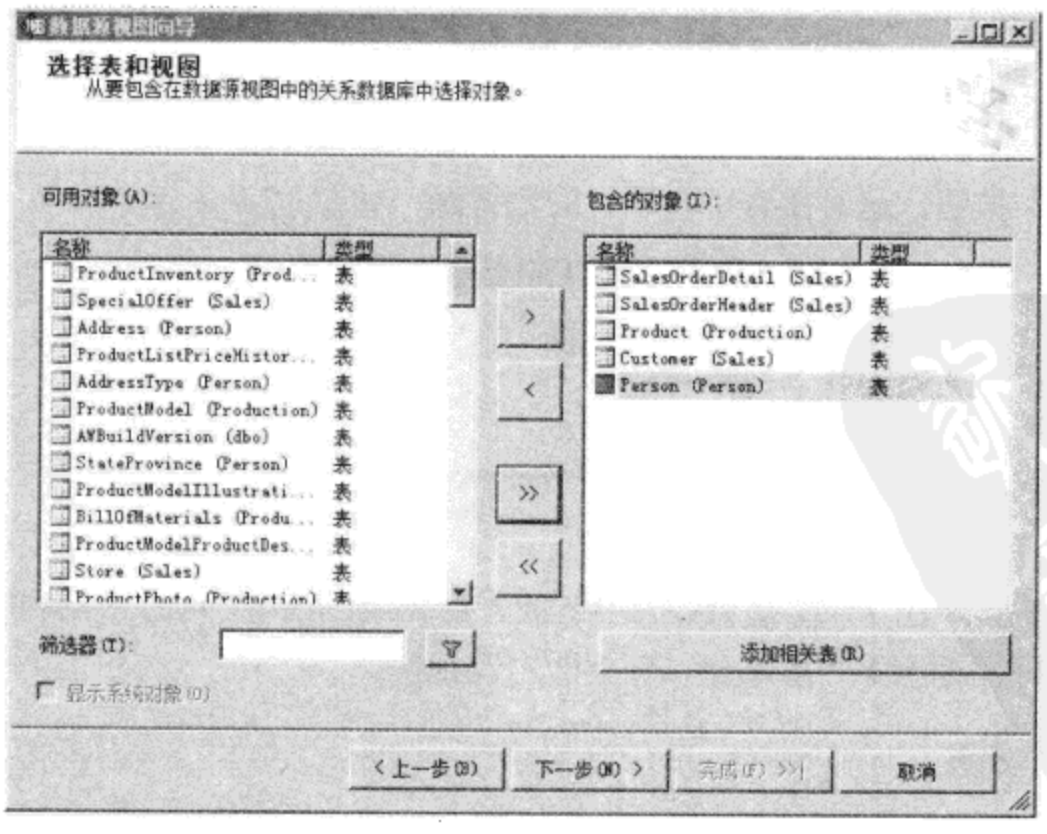


图 17-9

www.TopSage.com

注意：

不用担心该对话框中的“筛选器”或“显示系统对象”选项；二者都没有什么坏处。简单说明一下，“筛选器”选项仅对对象列表进行过滤，从而使数据库的处理变得简单，因为数据库往往需要显示大量的对象。“显示系统对象”选项允许在报表模型中包含系统对象(这一做法对大量的外部应用程序而言是不明智的)。

单击“下一步”按钮，得到如图 17-10 所示的“完成向导”对话框。该对话框极好地概括了最终数据源视图中将包含的选项。

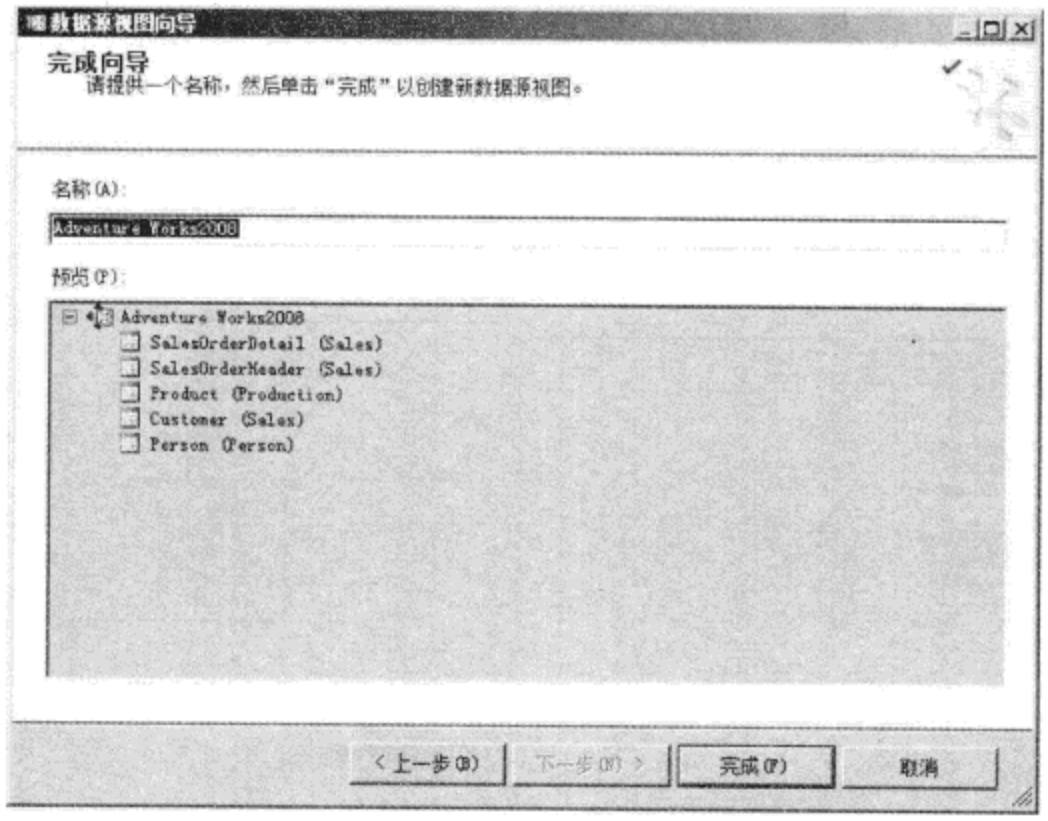


图　17-10

单击“完成”按钮，至此创建了数据源视图并将数据源视图添加到“报表模型项目”中。虽然有了一个可用的视图，但在数据模型使用视图之前，须先了解该视图的有关事项；在数据源视图中，Product 表和其余表之间没有正式的关系。所以下面先来解决这个问题。

1. 处理数据源视图中各对象间的关系

尽管向导试图将有关数据源视图的信息集合到一起，但这些信息是建立在表中的列名、类型以及外键引用的基础之上。虽然外键为两表间的关系给出了最为准确的描述，但有时也会由于某些原因出现在关系上没有定义正式外键的情况。在刚创建的数据源视图中，Products 表和 SalesOrderHeader 表(ProductSpecialOffer 表与二者都相连接，并且这是这两个表相互关联的唯一正式途径)之间就没有正式外键。不过，ProductID 却可作为两表间的连接项，所以应在数据源视图中添加这一引用。

首先在“解决方案资源管理器”窗口中右击新建的“数据源视图”节点，选择“视图设计器”命令，如图 17-11 所示。

Visual Studio 为数据源视图中包含的表提供了可视形式，但是，如图 17-12 所示，Visual Studio 不知道 SalesOrderDetail 表和 Product 表间的关系。

www.TopSage.com

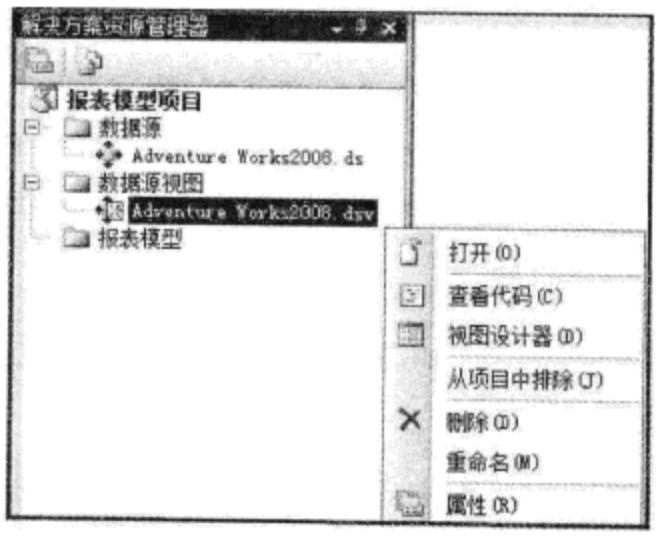


图 17-11

单击 SalesOrderDetail 表中的 ProductID 并将其拖至 Product 表的 ProductID 中，SQL Server 弹出一个关系确认对话框。单击“确定”按钮，将显示关系(如图 17-13 所示)。

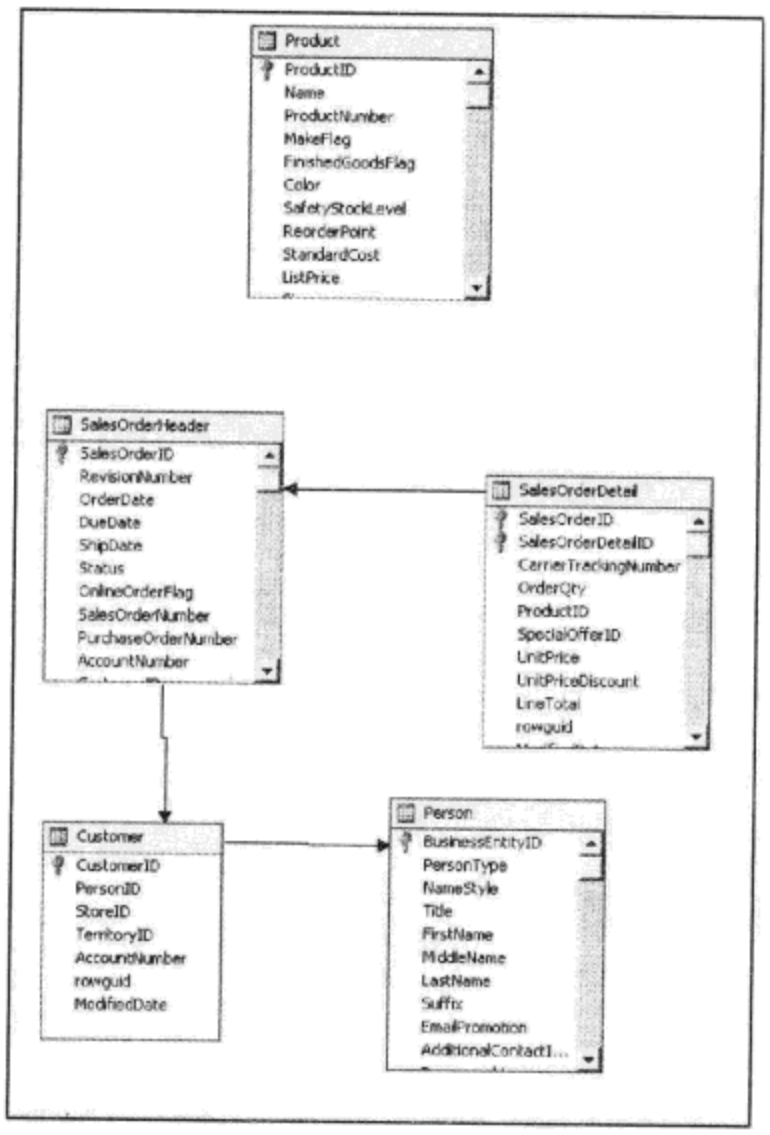


图 17-12

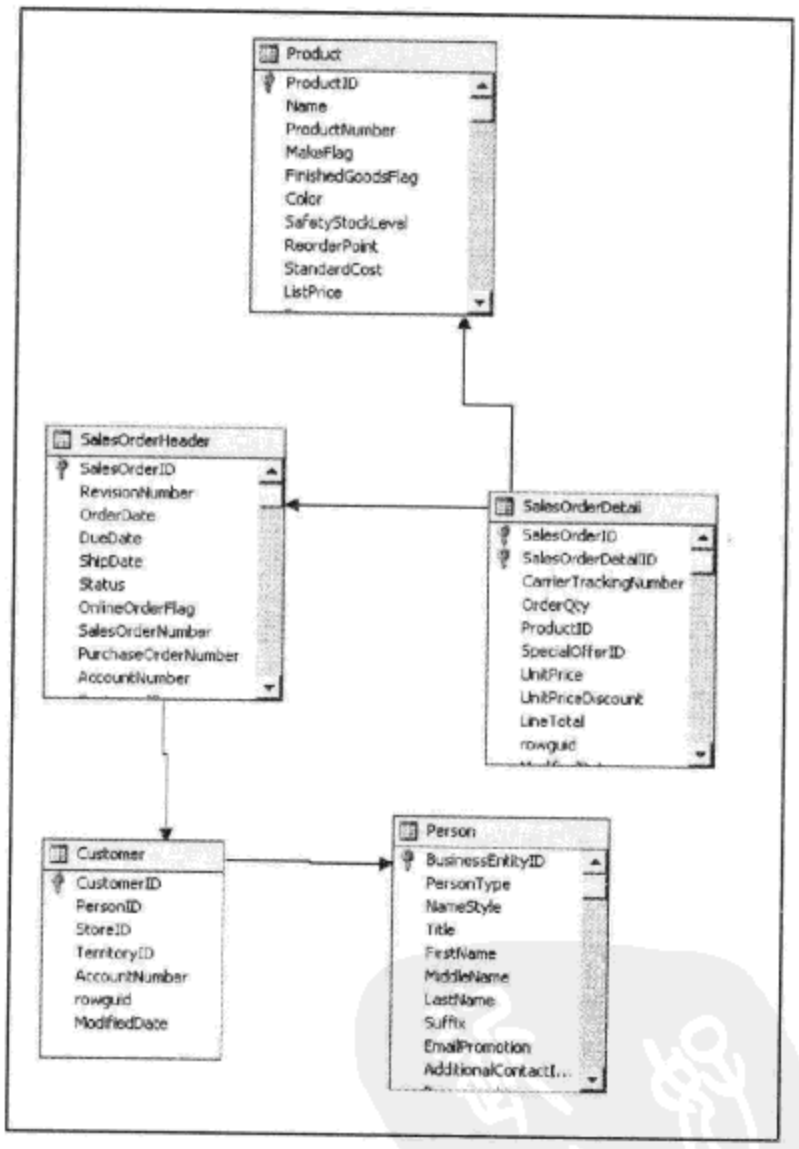


图 17-13

注意：

关联表的点击拖动顺序与在大多数数据库关系图工具中建立外键时所采用的方式相反。在大多数数据库关系图环境下，是从父表向子表拖动；而在数据源视图中拖动的方向是相反的。

www.TopSage.com

17.2.2　建立数据模型

现在就可以创建报表模型了。和创建数据源和数据源视图一样，只需右击“解决方案资源管理器”树中的“报表类型”节点，选择“添加新报表模型”命令，如图 17-14 所示。

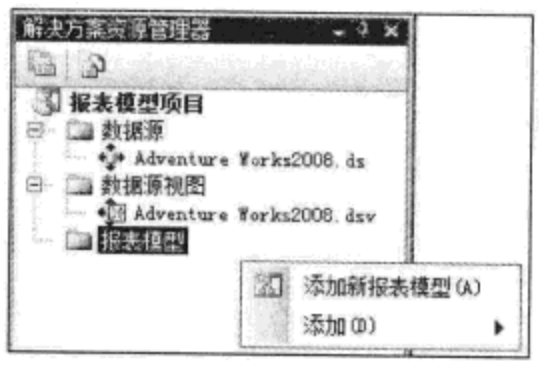


图　17-14

在开始定义报表模型时，出现“报表模型向导”对话框。通过单击打开内容相对丰富的“选择数据源视图”对话框，如图 17-15 所示。不要惊讶，唯一显示的选项就是刚才创建的数据源视图(仍然称作 AdventureWorks2008)。

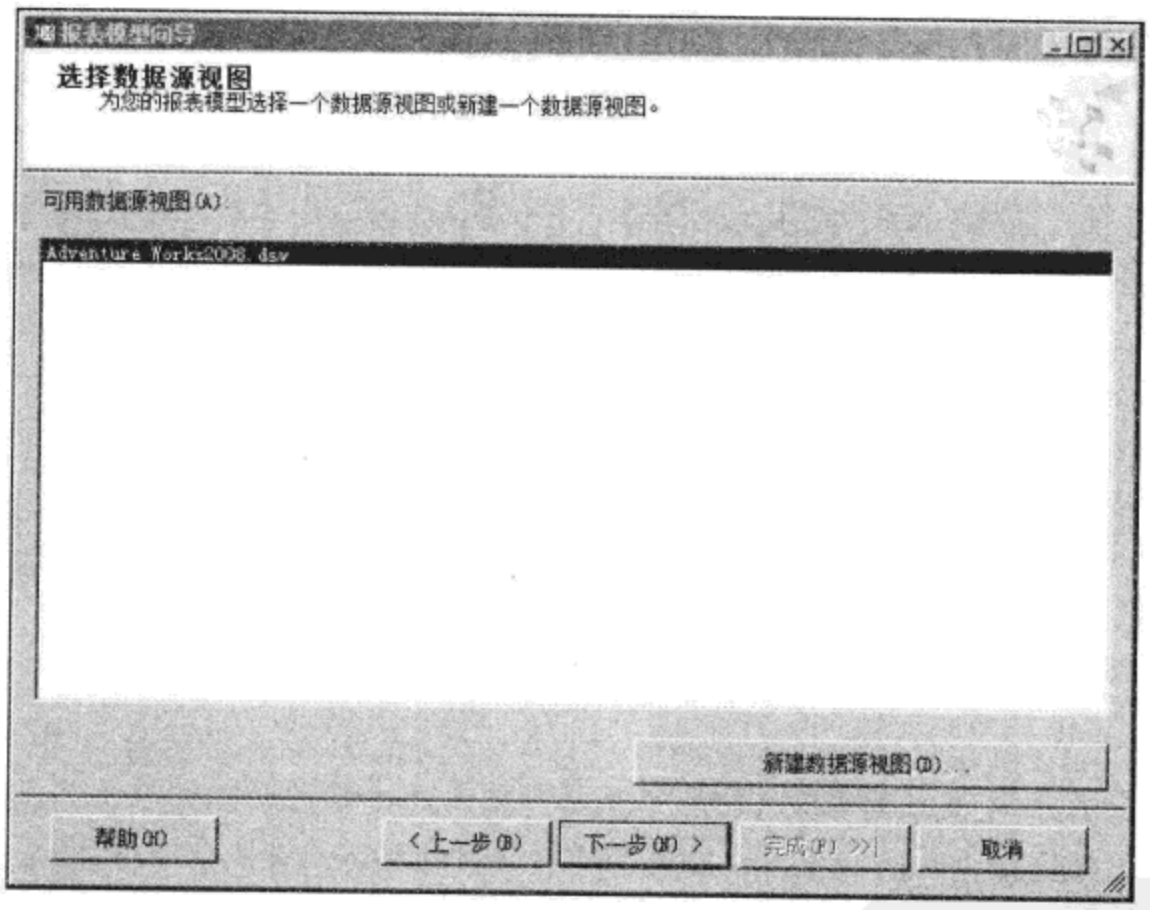


图　17-15

接着选择一个数据源视图并单击“下一步”按钮得到下一个对话框(如图 17-16 所示)，其中显示的是报表模型生成规则。

报表模型生成规则决定了报表模型中可用的汇总方式以及其他用以辅助生成终端用户报表的指导意见。同时之外要注意到可对报表模型默认语言进行控制(这对于跨国企业或者需要用服务器基本安装之外的语言生成报表的公司很方便)。

这里通过单击“下一步”按钮来接受默认选项，然后进入“收集模型统计信息”对话框。该对话框给出了以下两个选项。

www.TopSage.com

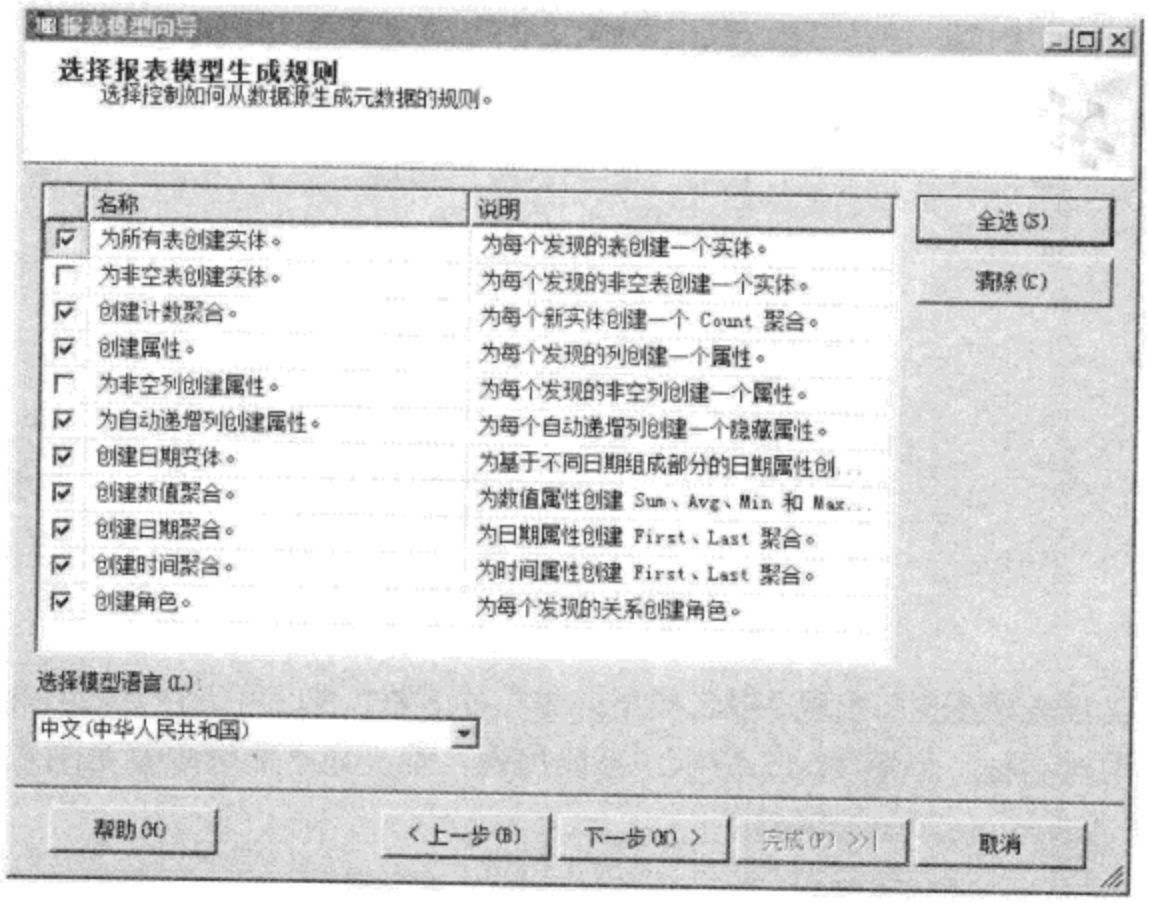


图 17-16

- **在生成前更新模型统计信息**：这一选项会在实际建立报表模型之前更新数据源视图中的所有统计数据。每个数据源视图都保存一组底层数据的统计信息(如列的宽度、使用的数据类型等)，这些统计信息用于确定如何最佳地构建报表模型。如果是初次使用数据源视图(由于刚刚创建视图，因此也可以算作初次)或自上次使用数据源视图以来已发生了重大变化，可选择该选项。
- **在使用数据源视图中存储的当前模型统计信息**：这一选项是使用已有统计信息。根据涉及的表和索引的多少，该选项能够节省大量生成报表所需的时间(因为不需要等待所有的统计信息都完成更新)，但是也有风险，报表模型会假定数据的组成部分作出已经过期。

注意：

是依赖现有统计信息还是继续更新统计信息，这是一个相对高级的概念。这里的建议是使用默认选项，并且持续更新统计信息，如果不这样做，您必须十分清楚这样做的原因及其后果。对于大多数的安装过程来说，在这一场景中不更新统计信息不是什么大问题，但也有可能埋下隐患，所以最好谨慎一些，以避免出现问题。

下一个(也是最后一个)对话框是“完成向导”对话框。对报表模型进行命名(这里选择的是AdventureWorks2008Report Model)并单击“运行”按钮，就会生成报表模型，如图 17-17 所示。

请注意，除了在报表生成时闪现的一些消息，在这个对话框中几乎没有数据显示，除非出现问题(在这种情况下，可看到对话框中出现相关消息)。单击“完成”按钮，可以看到创建的结果，如图 17-18 所示。

www.TopSage.com

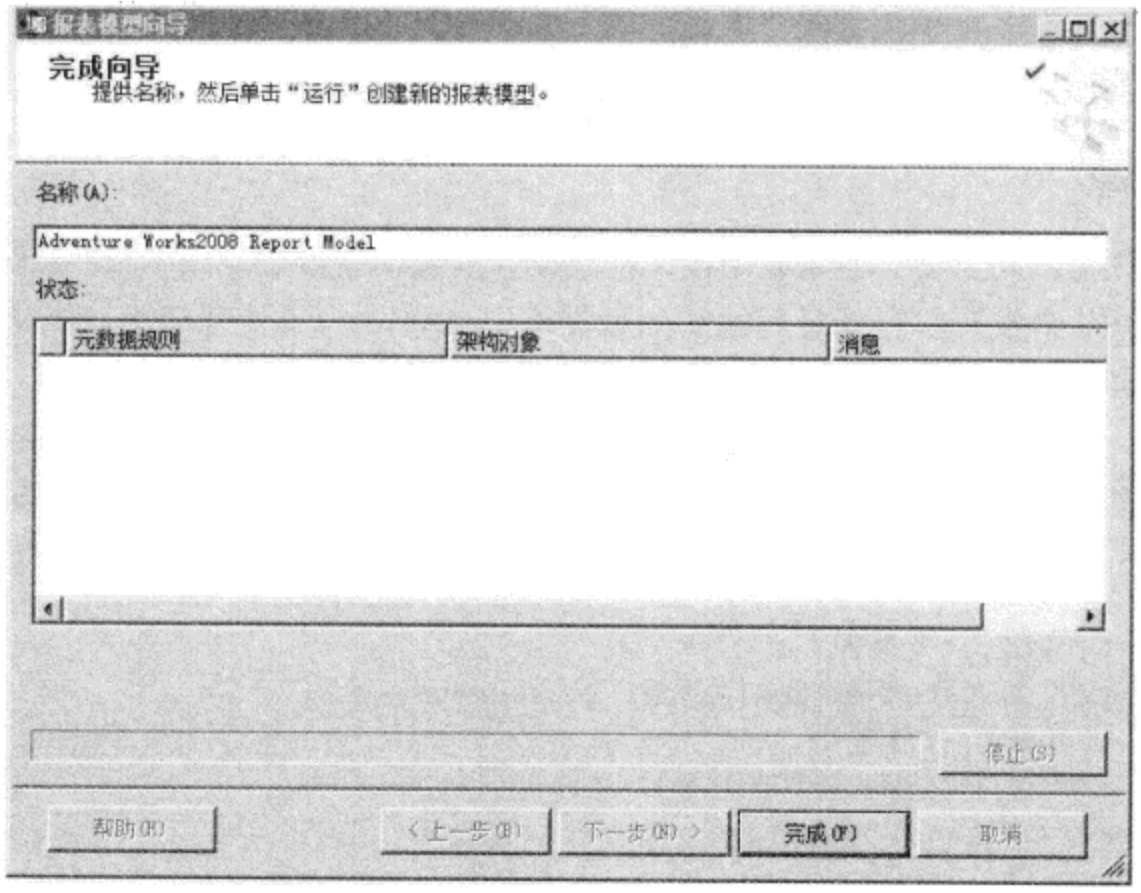


图　17-17

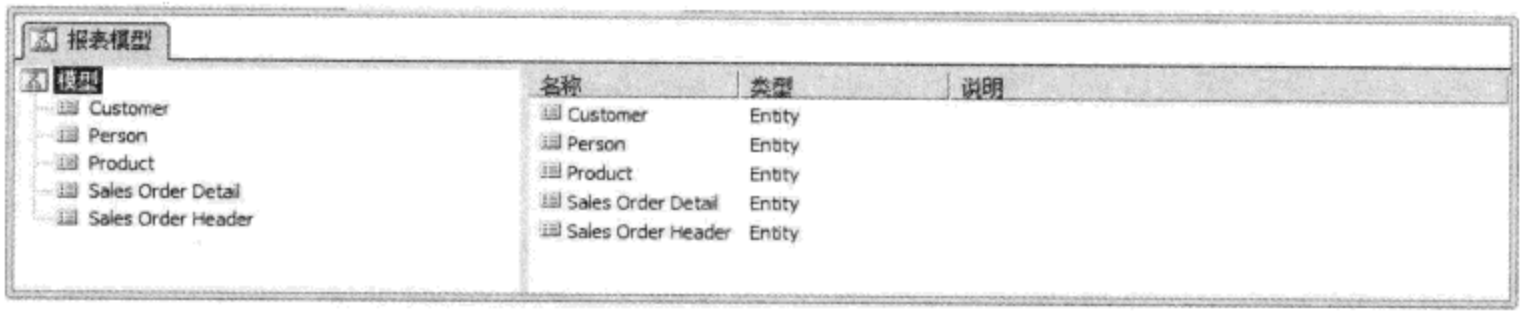


图　17-18

一定要花一些时间探究该报表模型。SQL Server 会对想要显示的以及不想显示的内容作出假设，但对于特定的数据库使用情况而言，SQL Server 并不总是能够作出正确的选择。看一下报表模型所包括的内容、添加的内容，以及用派生特性(汇总等)生成的内容。

下面简单地看一个示例，当向导完成了模型的创建后，某一个值可能被默认为隐藏，而有时这个值对报表又是十分有用的。首先导航至 SalesOrderHeader 表，可注意 Sales Order ID 列变为灰色状态的。如果单击这一列并查看图 17-19 所示的属性窗口，就会发现这一可能十分重要的字段已经定义为隐藏。

图　17-19

需要用一整章介绍模型中每个实体的细微差别和其中的所有属性。请特别注意 SQL Server 是如何根据基本数据类型的常见用法自动地将一些容易理解的属性(如日期)分解成较小的属性(如月、日、年)。

可能确实是想要隐藏这一字段，但有时也会碰到这样的情况，如 Sales Order ID，尽管是由系统生成的，但却会作为订单

www.TopSage.com

号出现在发票上。许多标识值都是“人造”的，因此 SQL Server 会自动认为您并不想要在报表中显示这些内容。

1. 部署模型

建立报表模型当然很好，但是如果没人能使用，报表模型就不能发挥作用(这很像设计一个存储过程，但是从来没有将存储过程添加到数据库中一样)。为了使终端用户能够使用报表，必须对报表进行部署。

所幸的是，部署模型并非难事。事实上，难点就是要了解要做什么并找到完成的命令。要进行部署，只要在“解决方案资源管理器”(或实际报表)的空白区域中右击，或右击项目名称并选择“部署”命令即可。可以在 Visual Studio 的输出窗口中看到部署过程进度。

```
生成完成 --0 个错误，0 个警告
------ 已启动生成:项目:报表模型项目,配置: Production------
生成完成 -- 0 个错误，0 个警告
------ 已启动部署:项目:报表模型项目,配置: Production------
部署到 http://localhost/ReportServer。
部署数据源"http://localhost/ReportServer/数据源/Adventure Works2008"。
部署模型"http://localhost/ReportServer/模型/Adventure Works2008 Report Model.smdl"。
部署完成 - 0 个错误，0 个警告
========== 生成: 成功 1 个，失败 0 个，最新 0 个，跳过 0 个 ==========
========== 部署: 成功 1 个，失败 0 个，跳过 0 个 ==========
```

17.2.3 创建报表

报表模型本身不是报表，它仅仅是有助于生成报表而已(考虑到有很多企业人员理解报表但不理解数据库，所以简化报表成为很重要的一件事)。当对模型进行部署后，就可以从这一个模型生成多个报表。

为了生成并查看报表，需要离开 Business Intelligence Studio 并访问报表用户界面——它本质上是一个 Web 站点。导航到 http://<your reporting server host>/reports，在本例中，因为报表已存在于本地系统中，所以可以导航到 http://localhost/reports，如图 17-20 所示。

这实际上是用户报表等的主页。这时可看到数据源和模型边上的“！新”图标(暗示部署已经生效)。然后单击“报表生成器”图标。

说明：

报表生成器依赖于一个 applet 小程序，当第一次导航到报表生成器时，该小程序会试图在系统上安装。如果想要使用报表生成器，就必须安装这个 applet 程序。

在这个 applet 程序的右边会出现提示框，要求选择报表的数据源。选择刚创建的模型，并单击“确定”按钮，获得默认的报表模板。这是一个相当健壮的拖放设计环境，可以在报表模型中找到可用的表并选择需要的列，将其拖放到报表中。同时请注意报表布局面板的右侧。这里可以为报表提供一些常用的布局选择。

www.TopSage.com

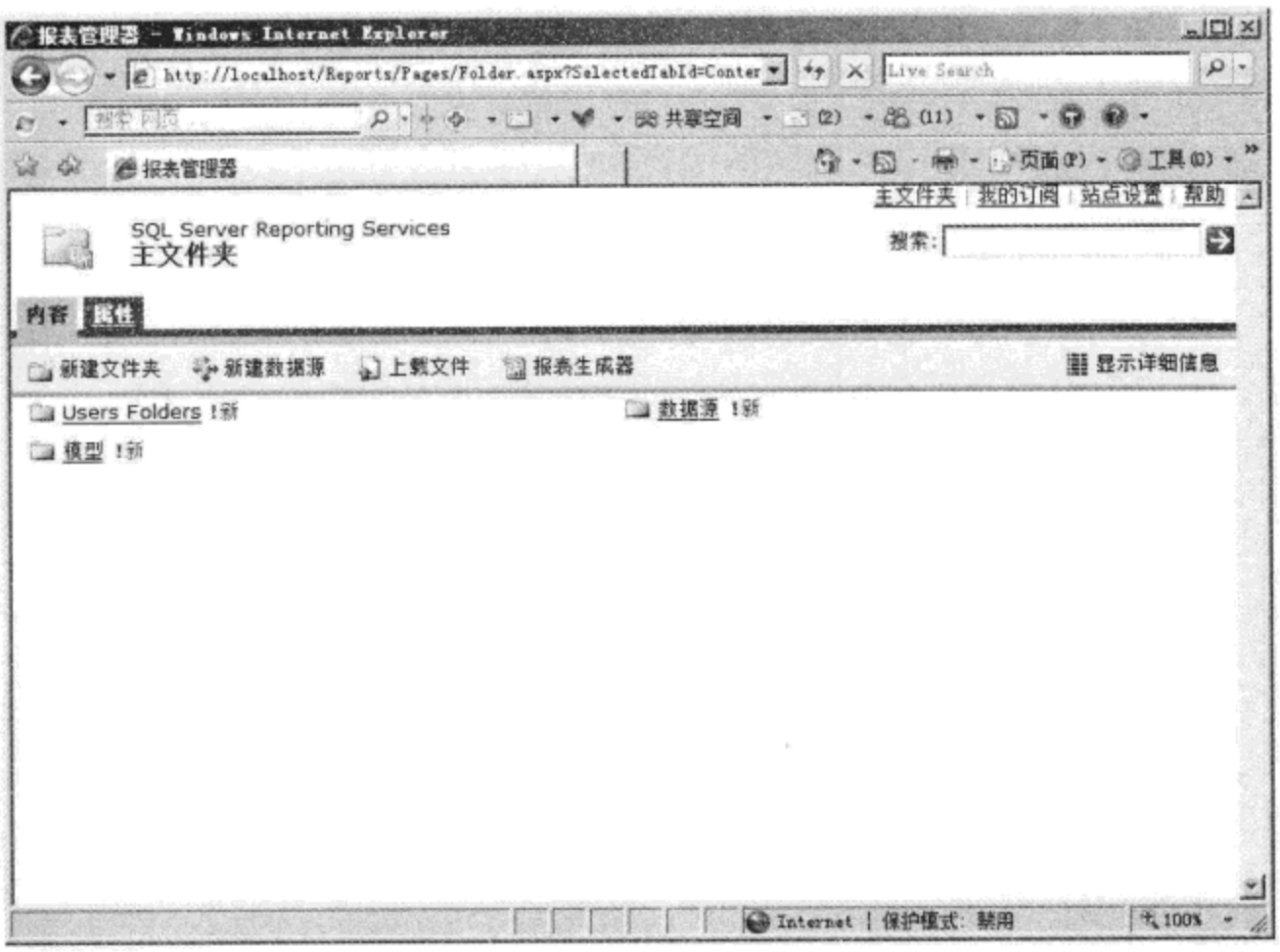


图　17-20

下面将继续创建表形式的报表(默认选择)，建立一个相对简单的报表，列出所有用除信用卡外其他方式支付的订单。

导航到每一个表并将以下相关的列拖进表中。

- SalesOrderHeader.SalesOrderNumber
- Person.LastName
- Person.FirstName
- Product.ProductNumber
- Product.Name
- SalesOrderHeader.OrderDate
- SalesOrderHeader.DueDate

单击标题区域并添加“Non Credit Card Orders”标题。完成后，会得到如图 17-21 所示的结果(注意，这里对列作了调整以适合页面)。同时要注意“资源管理器”窗口下的“字段”面板中所使用的字段是如何加粗显示的。

说明：

现在就可以运行报表了，不过这要取决于报表服务器的配置。但是对于大多数系统而言，都会有一个限制，即在运行报表之前需要对结果进行筛选(以避免过载)。

要创建筛选器，可单击“报表生成器”窗口顶部的“筛选”图标，进入“筛选数据”对话框。将 Credit Card ID 字段拖入对话框的主面板中，这就表示使用 Credit Card ID 作为筛选器，但仍需要设置确切的筛选条件。默认比较结果是“等于”，这不能满足需求，因此单击“等于”这个词，选择“为空”选项，如图 17-22 所示，单击“确定”按钮。

www.TopSage.com

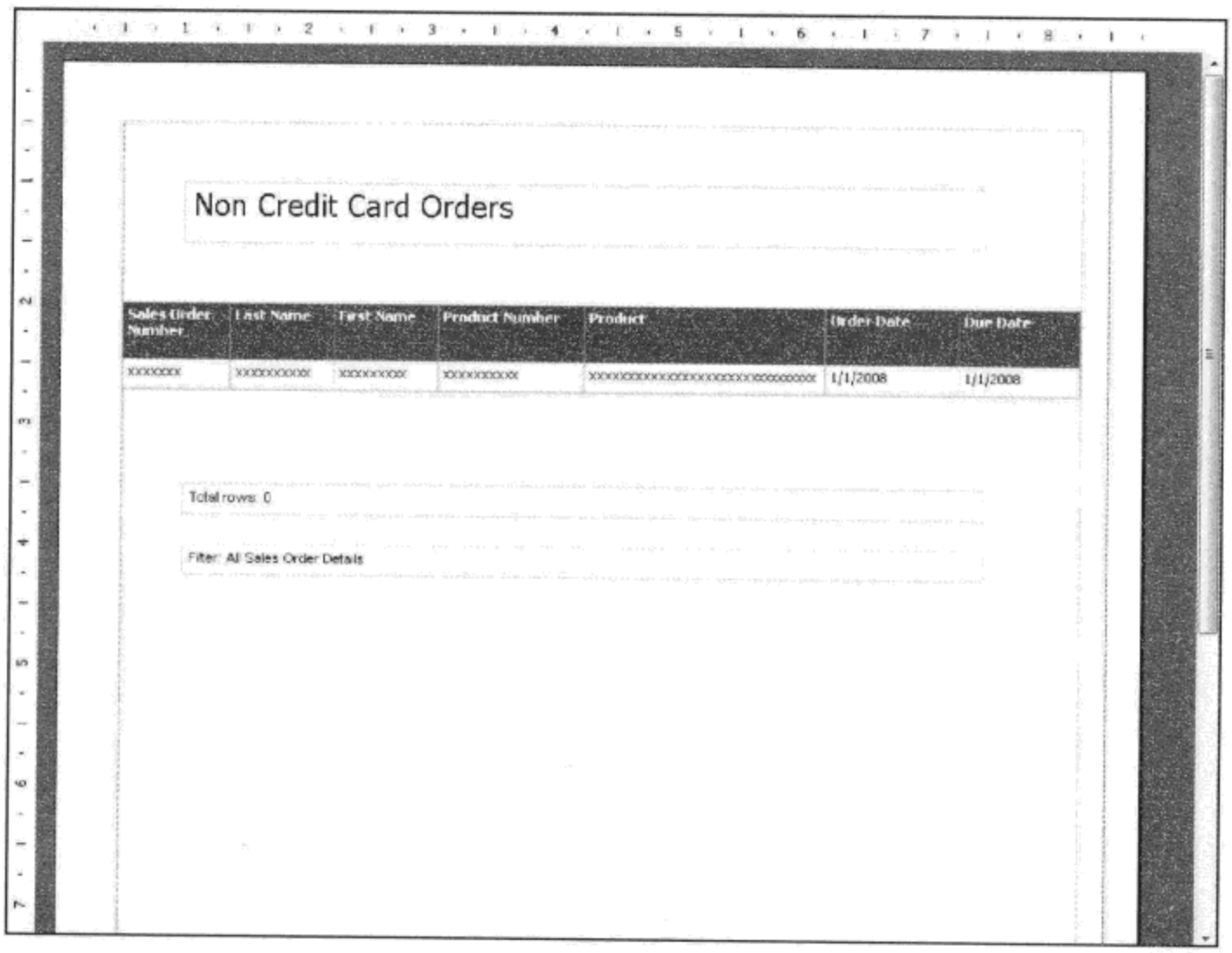


图 17-21

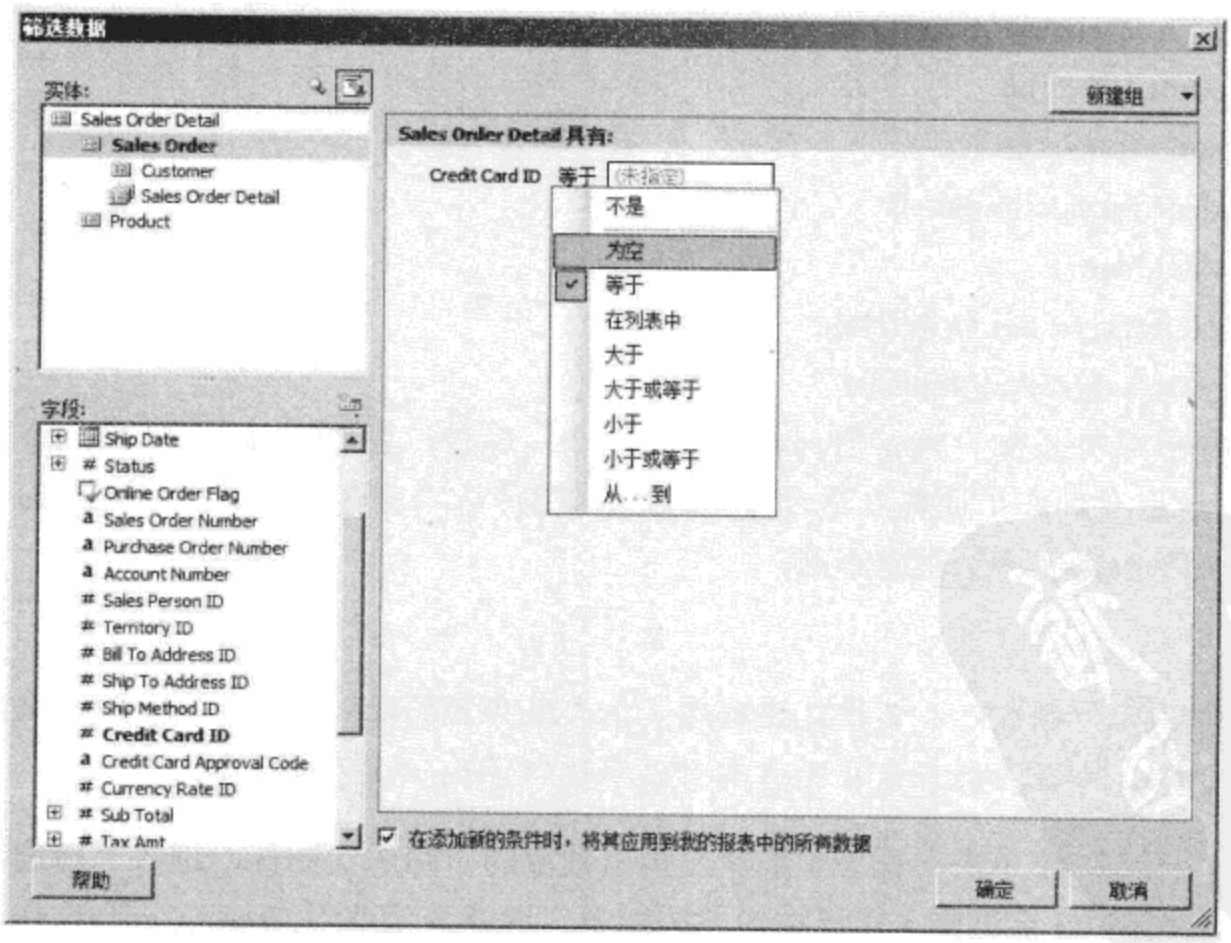
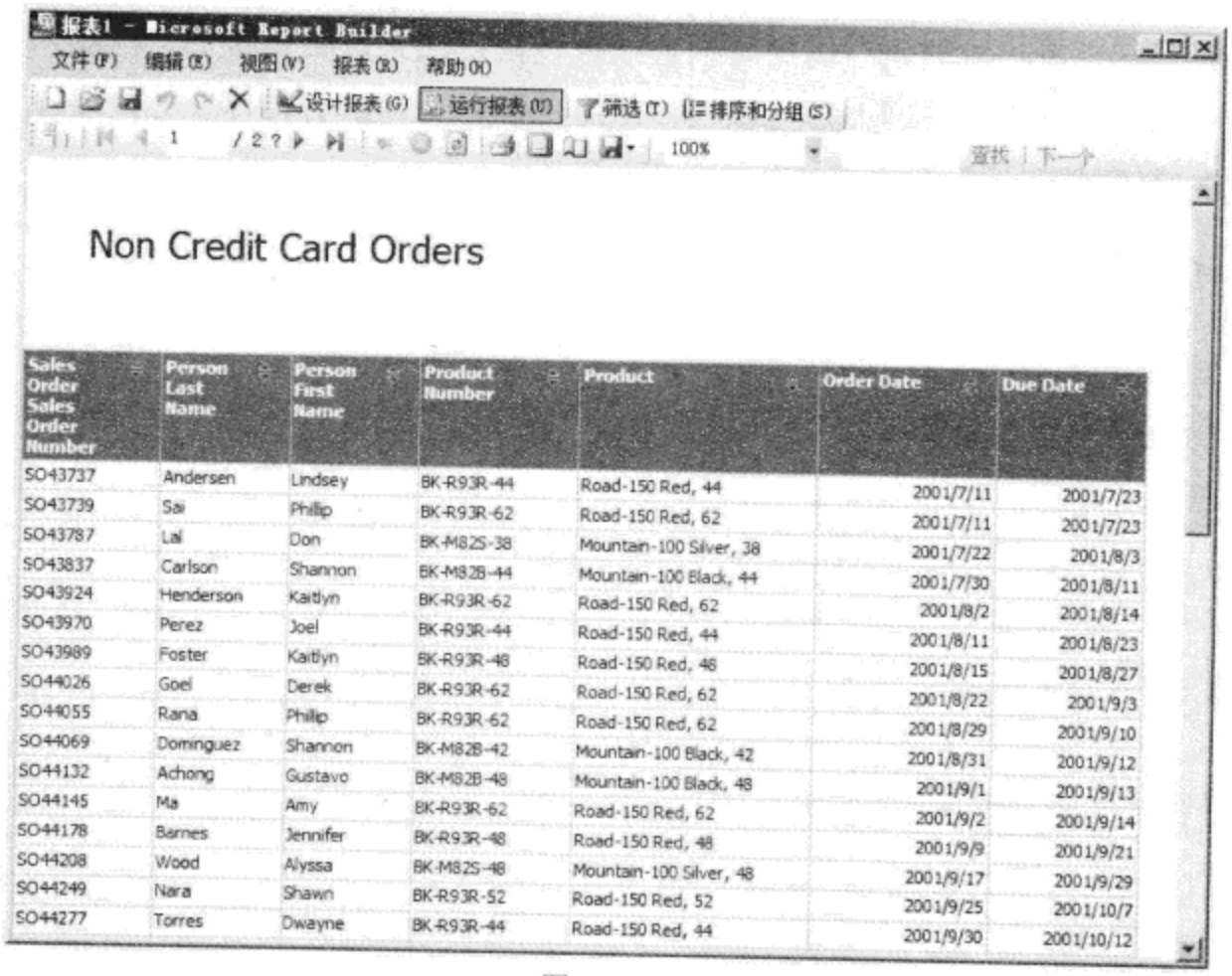


图 17-22

完成这些工作后，单击“运行报表”图标，即可看到努力的成果，如图 17-23 所示。

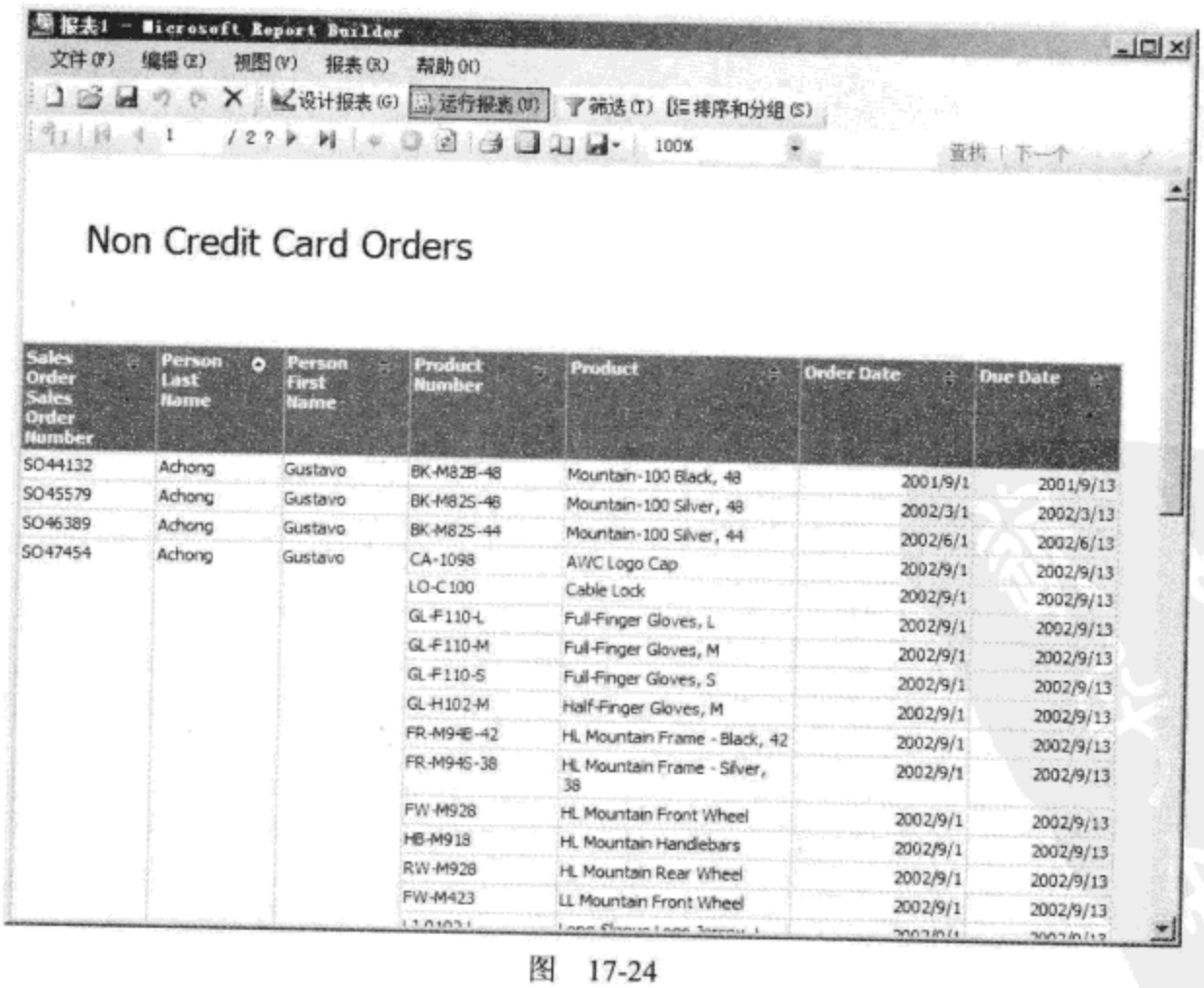
www.TopSage.com



Sales Order Sales Order Number	Person Last Name	Person First Name	Product Number	Product	Order Date	Due Date
SO43737	Andersen	Lindsey	BK-R93R-44	Road-150 Red, 44	2001/7/11	2001/7/23
SO43739	Sai	Phillip	BK-R93R-62	Road-150 Red, 62	2001/7/11	2001/7/23
SO43787	Lal	Don	BK-M82S-38	Mountain-100 Silver, 38	2001/7/22	2001/8/3
SO43837	Carlson	Shannon	BK-M82B-44	Mountain-100 Black, 44	2001/7/30	2001/8/11
SO43924	Henderson	Kaitlyn	BK-R93R-62	Road-150 Red, 62	2001/8/2	2001/8/14
SO43970	Perez	Joel	BK-R93R-44	Road-150 Red, 44	2001/8/11	2001/8/23
SO43989	Foster	Kaitlyn	BK-R93R-48	Road-150 Red, 48	2001/8/15	2001/8/27
SO44026	Goel	Derek	BK-R93R-62	Road-150 Red, 62	2001/8/22	2001/9/3
SO44055	Rana	Phillip	BK-R93R-62	Road-150 Red, 62	2001/8/29	2001/9/10
SO44069	Dominguez	Shannon	BK-M82B-42	Mountain-100 Black, 42	2001/8/31	2001/9/12
SO44132	Achong	Gustavo	BK-M82B-48	Mountain-100 Black, 48	2001/9/1	2001/9/13
SO44145	Ma	Amy	BK-R93R-62	Road-150 Red, 62	2001/9/2	2001/9/14
SO44178	Barnes	Jennifer	BK-R93R-48	Road-150 Red, 48	2001/9/9	2001/9/21
SO44208	Wood	Alyssa	BK-M82S-48	Mountain-100 Silver, 48	2001/9/17	2001/9/29
SO44249	Nara	Shawn	BK-R93R-52	Road-150 Red, 52	2001/9/25	2001/10/7
SO44277	Torres	Dwayne	BK-R93R-44	Road-150 Red, 44	2001/9/30	2001/10/12

图　17-23

但是这里也有一个小问题——报表有若干页(具体有多少页要导航至最后才能确定)。当前的信息排序方式没有多少用处，但是不要紧，报表生成器可以轻松解决这一问题，只需单击报表每一列旁边的上/下箭头即可。在本例中，根据人员的姓排序，如图 17-24 所示。注意报表是如何合理自动地进行分组的(避免了重复值)。



Sales Order Sales Order Number	Person Last Name	Person First Name	Product Number	Product	Order Date	Due Date
SO44132	Achong	Gustavo	BK-M82B-48	Mountain-100 Black, 48	2001/9/1	2001/9/13
SO45579	Achong	Gustavo	BK-M82S-48	Mountain-100 Silver, 48	2002/3/1	2002/3/13
SO46389	Achong	Gustavo	BK-M82S-44	Mountain-100 Silver, 44	2002/6/1	2002/6/13
SO47454	Achong	Gustavo	CA-1098	AWC Logo Cap	2002/9/1	2002/9/13
			LO-C100	Cable Lock	2002/9/1	2002/9/13
			GL-F110-L	Full-Finger Gloves, L	2002/9/1	2002/9/13
			GL-F110-M	Full-Finger Gloves, M	2002/9/1	2002/9/13
			GL-F110-S	Full-Finger Gloves, S	2002/9/1	2002/9/13
			GL-H102-M	Half-Finger Gloves, M	2002/9/1	2002/9/13
			FR-M94B-42	HL Mountain Frame - Black, 42	2002/9/1	2002/9/13
			FR-M94S-38	HL Mountain Frame - Silver, 38	2002/9/1	2002/9/13
			FW-M928	HL Mountain Front Wheel	2002/9/1	2002/9/13
			HB-M918	HL Mountain Handlebars	2002/9/1	2002/9/13
			RW-M928	HL Mountain Rear Wheel	2002/9/1	2002/9/13
			FW-M423	LL Mountain Front Wheel	2002/9/1	2002/9/13

图　17-24

www.TopSage.com

简单操作一下报表，发现还可以用其他方法方便地对结果进行再排序。注意，本来也可以定义一个默认的排序次序——例如，可以根据最近的应付日期进行排序——只要在“排序和分组”对话框中设置(在工具栏的“筛选”图标旁)即可。

1. 关于报表模型的补充说明

当然，报表模型并不是报表的全部。但报表模型是 SQL Server 的一个新功能，即以某种可控制的方式向终端用户提供数据(用户只能看到放入数据源的数据，只能访问安全的数据源，这样便可以掌握哪些用户查看哪些数据)，同时仍允许用户创建自己特有的报表。报表模型为创建“快速而随性”的报表提供很好的选项。

最后，请记住，这里只使用了最简单的布局生成了一个极为简单的报表。报表模型也允许使用基本的图形及矩阵报表。

17.3 报表服务器项目

可以把报表模型看作入门——Reporting Services 比报表模型更为灵活(事实上，需要用整本的书来介绍 Reporting Services，而且也已经有很多关于这方面的书籍)。除了现有的功能之外，还可以使用 Business Intelligence Development Studio 来创建报表服务器项目。

前面提到过，要用整本书来介绍这个主题，所以这里将通过另一个简单示例简单介绍一下这一主题(事实上，是使用项目的方法完成同样的示例)。

这里将要用的概念都是容易理解的，因此不会像前文那样图文并茂，而只是列出所需完成的基本操作来说明。

(1) 在 Business Intelligence Development Studio 中使用“报表服务器”项目模板来打开一个新项目(请注意这与之前使用的报表模型项目不同)。

(2) 为 AdventureWorks2008 数据库创建新的数据源(右击“共享数据源”文件夹并填写对话框——如果想让对话框创建连接字符串，可使用“编辑”按钮，或者也可以将本章前面的连接字符串复制过来)。要部署新数据源，应注意使用的名称与本章之前使用的不要相同(假设完成了前面的示例)。

(3) 右击“报表”文件夹并选择“添加新报表”命令，进入“报表向导”对话框——单击“下一步”按钮进入“选择数据源”对话框，选择刚刚创建的数据源，单击“下一步”按钮。

将得到如图 17-25 所示的查询对话框。

当然，这里已经添加过查询；这一查询会大致复制在 17.2 节中建立的报表，该报表包括对未发送订单的筛选器。也可以单击“查询生成器”按钮，运行查询以验证是否得到预期数据。注意这里还可以运行一个存储过程或查询一个具体的表。

单击“下一步”按钮并接受“表格”报表类型，会得到如图 17-26 所示的报表向导。

由于查询已将数据缩减为需要的几列(虽然各列已按适当次序排列，但是如果需要，也可以重新排序)，所以这里选择了所有的内容并将它们移入到“详细信息”列表框中。

www.TopSage.com

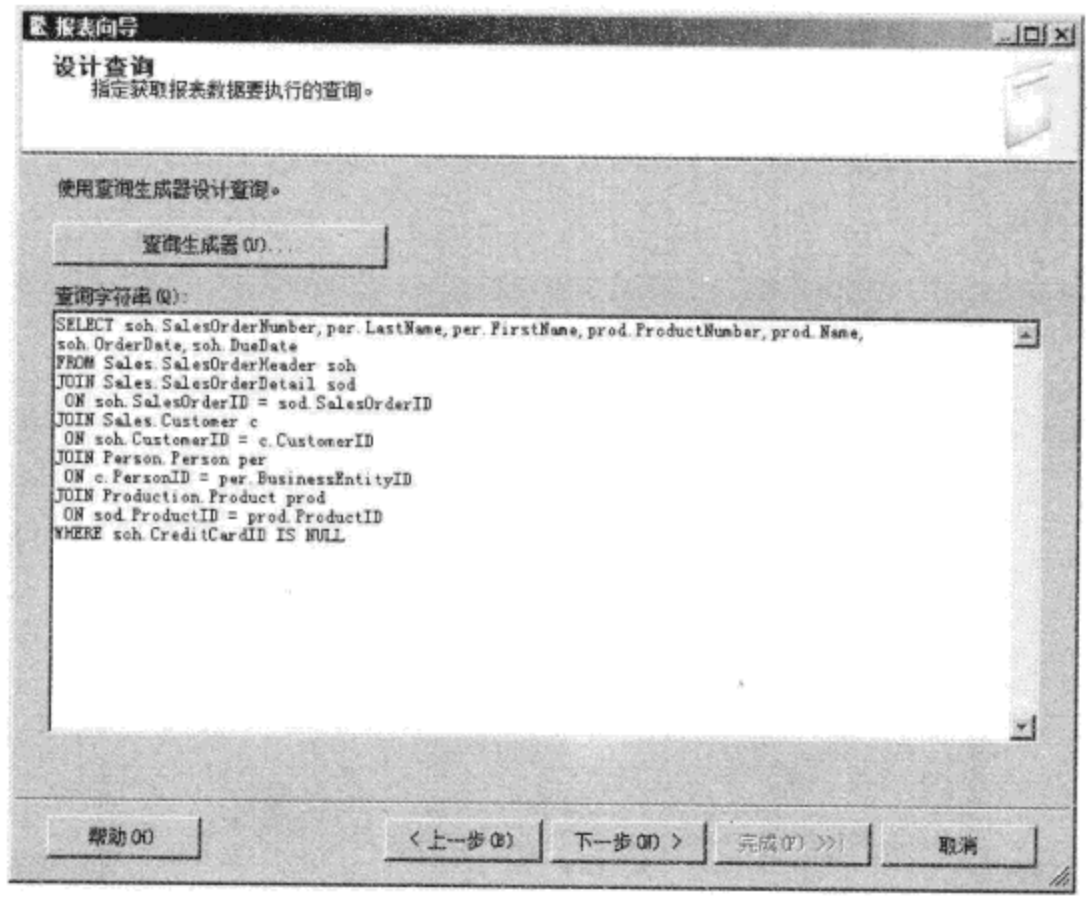


图　17-25

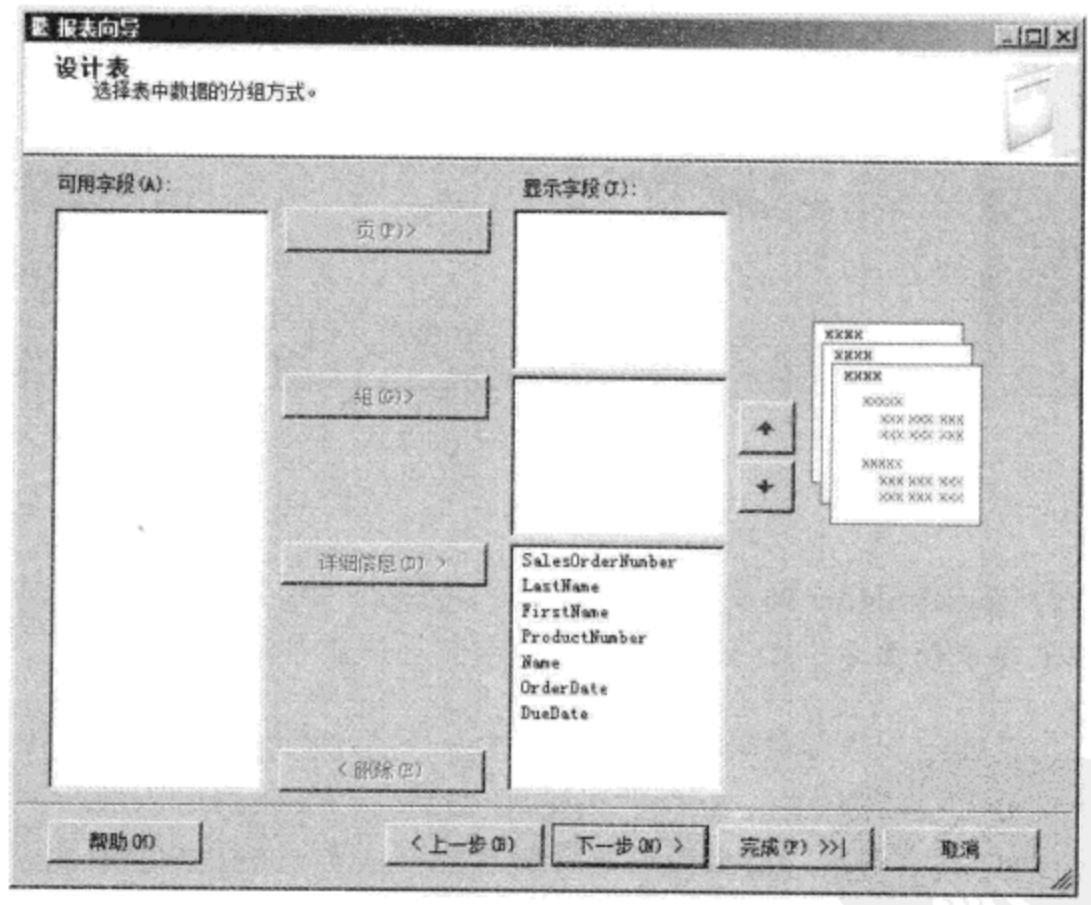


图　17-26

注意：

这里的“页”和“组”列表框允许建立排序的层次结构。例如，如果想要基于每个客户独立分页(例如销售人员想要了解某个特定客户的状态)，可以将 Company Name 移到“页”列表框中。同样地，也可以根据产品名称进行分组(而不是页)使得从库存中提取订单的人员一次就能够获得全部所需产品来交付所有未偿付的订单。

www.TopSage.com

再次单击“下一步”按钮，进入“选择表样式”对话框。选择喜欢的样式(这里使用石板样式)并单击“下一步”按钮，将出现有关报表选择的概要说明。将名称改为“Non Credit Card Orders”，单击“完成”按钮以完成对报表的定义，如图 17-27 所示(如果选择不同的样式，显示结果可能略有不同)。

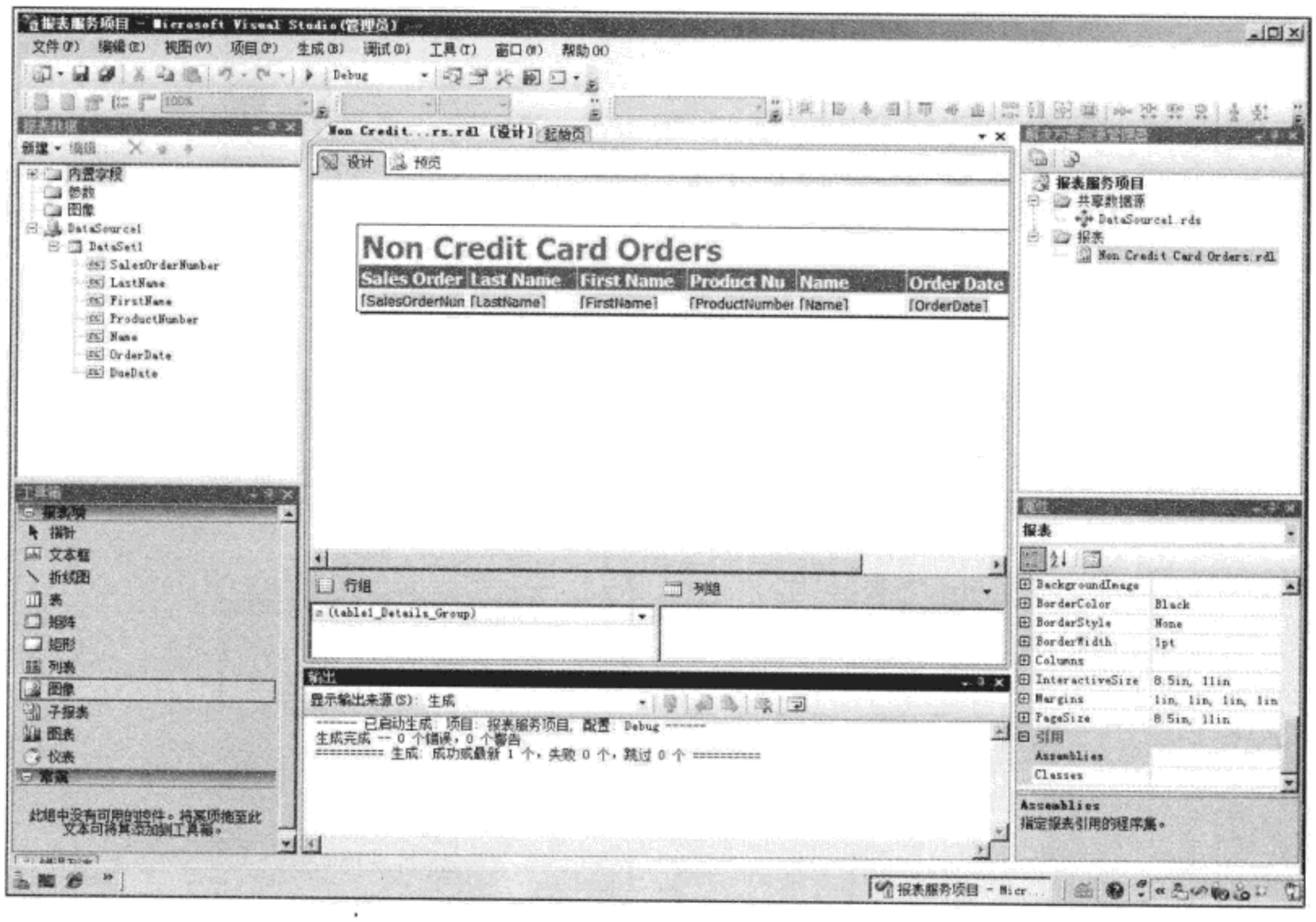


图 17-27

注意：

由于每个人的 Visual Studio 配置不同，因此“工具箱”和“报表数据”面板的位置也会有所不同。这里为了演示的需要，两个面板都被设为可见并且可停靠的。

如果您很熟悉其他的报表设计器，那么可能对这个设计器也不陌生，因为对于报表设计器来说，这里的 WYSIWYG 编辑器是标准的编辑器(当然有些编辑器更为健壮，但是基本显示形式是相当典型的)。但请注意，这里为报表选择的名称已成为该报表的默认表头。

如果预览报表，会发现它和使用报表模型生成的报表很相似。但是报表模型默认的一些格式化区域要比现在这个报表好一些——尤其是在现有报表中，如果时间总为午夜，那么将报表时间作为日期的一部分的做法就没有意义。考虑到这点，下面将对向导生成的报表做一些修改。

试一试　修改报表项目

使用向导很快就生成了报表。但是生成的报表并不完美。我们希望能够更好地格式化日期。

www.TopSage.com

(1) 右击第一个日期字段(Order Date)并选择“文本框属性”命令，进入图 17-28 所示的对话框。

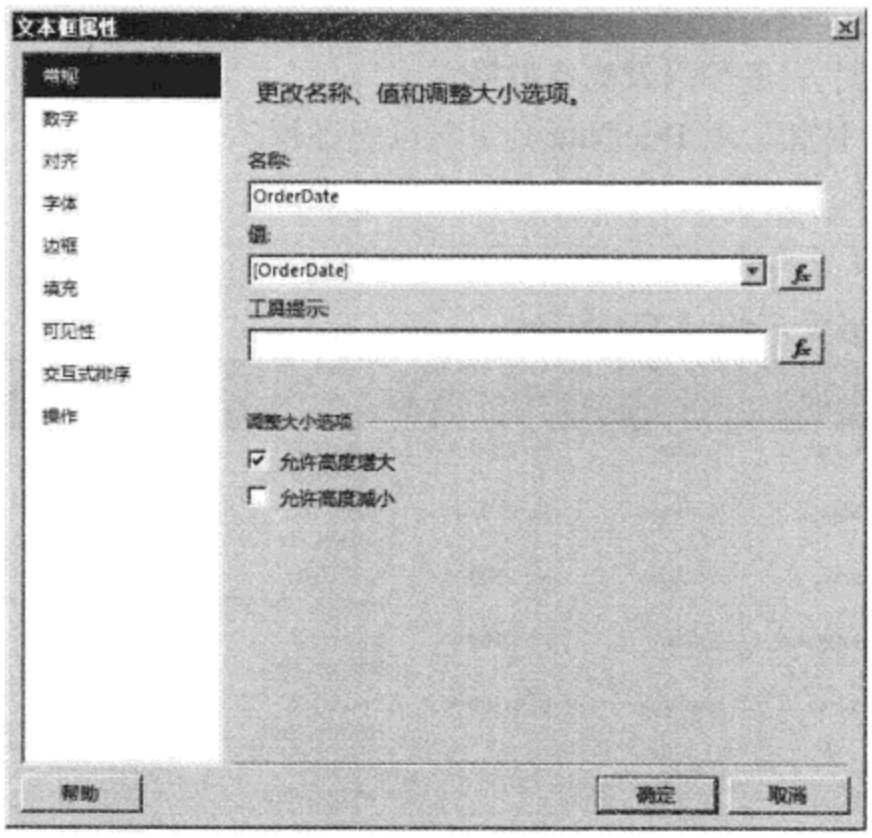


图　17-28

(2) 单击 f_x 按钮，指明要修改函数输出的结果，出现函数对话框，如图 17-29 所示。

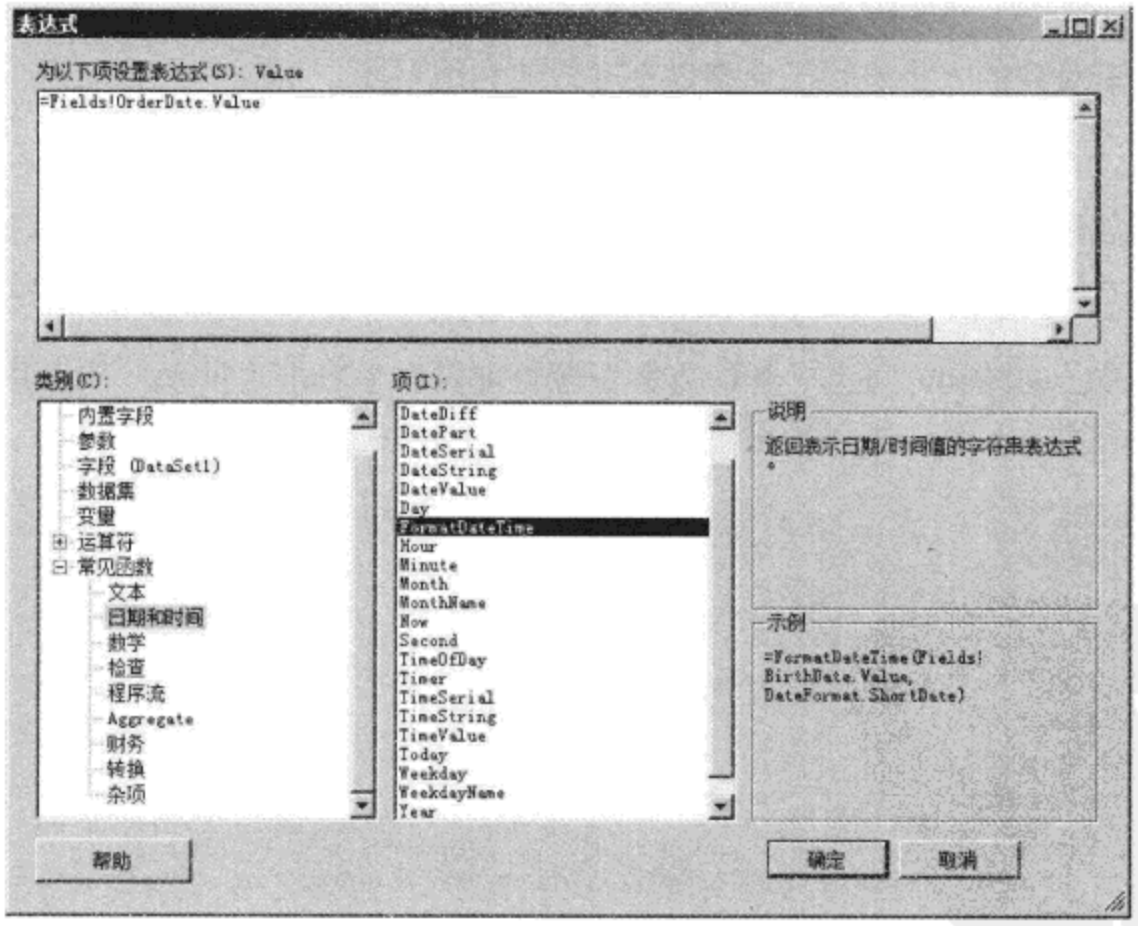


图　17-29

(3) 添加 FormatDateTime 函数。注意这里已经展开了下面的函数插入帮助工具。可以双击该函数把基本函数名插入上面的窗口中。和 Visual Studio 的其他部分一样，在填入函数时，它

www.TopSage.com

会提供上下文敏感提示信息。注意，这个特殊的函数也有可选的参数，允许指定特殊的日期显示风格(譬如欧式或日式的)，但是要根据服务器的区域设置来进行格式化。

(4) 单击“确定”按钮，移至“对齐”选项卡。将“水平”选项修改为“右框线”(从右到左)，这样会使字段右对齐(多数报表的典型样式)。

(5) 单击“确定”按钮，对 DueDate 列重复以上过程。

(6) 再次预览报表，结果应如图 17-30 所示。

Non Credit Card Orders

Sales Order Number	Last Name	First Name	Product Number	Name	Order Date	Due Date
SO47454	Achong	Gustavo	HL-U509-R	Sport-100 Helmet, Red	2002/9/1 0:00:00	2002/9/13 0:00:00
SO48395	Achong	Gustavo	HL-U509-R	Sport-100 Helmet, Red	2002/12/1 0:00:00	2002/12/13 0:00:00
SO50756	Achong	Gustavo	HL-U509-R	Sport-100 Helmet, Red	2003/6/1 0:00:00	2003/6/13 0:00:00
SO51181	Henderson	Kaitlyn	HL-U509-R	Sport-100 Helmet, Red	2003/7/1 0:00:00	2003/7/13 0:00:00
SO51232	Navarro	Dwayne	HL-U509-R	Sport-100 Helmet, Red	2003/7/4 0:00:00	2003/7/16 0:00:00
SO51234	Xu	Peter	HL-U509-R	Sport-100 Helmet, Red	2003/7/4 0:00:00	2003/7/16 0:00:00
SO51374	Raji	Lindsey	HL-U509-R	Sport-100 Helmet, Red	2003/7/13 0:00:00	2003/7/25 0:00:00
SO51407	Suri	Ronald	HL-U509-R	Sport-100 Helmet, Red	2003/7/15 0:00:00	2003/7/27 0:00:00

图 17-30

注意，这里对列的大小进行了调整以更有效地利用页面。

示例说明

在报表背后涉及到报表定义语言(Report Definition Language，RDL)。RDL 实际上是用 XML 实现的。当用户进行修改时，RDL 在后台也进行着修改，这样报表生成器才知道该做什么。

要查看 RDL 的结构，可在“解决方案资源管理器”中右击“报表”节点并选择“查看代码”命令。代码内容很长，下面给出了刚才在报表中编辑的一个字段的摘录：

```
<TablixCell>
 <CellContents>
   <Textbox Name="OrderDate">
     <CanGrow>true</CanGrow>
     <KeepTogether>true</KeepTogether>
     <Paragraphs>
       <Paragraph>
         <TextRuns>
           <TextRun>
             <Value>=FormatDateTime(Fields!OrderDate.Value)</Value>
             <Style>
               <FontFamily>Tahoma</FontFamily>
             </Style>
           </TextRun>
         </TextRuns>
         <Style>
           <TextAlign>Right</TextAlign>
```


www.TopSage.com

```
          </Style>
        </Paragraph>
      </Paragraphs>
      <rd:DefaultName>OrderDate</rd:DefaultName>
      <Style>
        <Border>
          <Color>LightGrey</Color>
          <Style>Solid</Style>
        </Border>
        <PaddingLeft>2pt</PaddingLeft>
        <PaddingRight>2pt</PaddingRight>
        <PaddingTop>2pt</PaddingTop>
        <PaddingBottom>2pt</PaddingBottom>
      </Style>
    </Textbox>
  </CellContents>
</TablixCell>
```

如果想成为专家，可以尝试直接编辑 RDL。

部署报表

剩下来要做的就是部署报表。和采用报表模型的方法一样，可以在“解决方案资源管理器”中右击报表并选择“部署”命令。但是有一点需要注意，就是要在项目定义中定义部署的目标。

(1) 右击“报表服务项目”并选择“属性”命令。

(2) 在 TargetServerURL 字段中，输入导航到 ReportServer 的 URL。在本例中为 http://localhost/ReportServer，但是其服务器名可以是有权对其进行部署的任何一台服务器的名称(如果在安装时这样定义，那么虚拟目录也不会是 ReportServer)。

部署完成后，您可能想要查看报表。如果导航到报表服务器(如果是在本地主机上并使用默认目录，那么应为 http://localhost/Reports，和前面的报表模型示例一样)。单击创建的报表项目，选择 Non Credit Card Orders 报表。该报表在第一次加载时出现的速度较慢(如果以后再次访问该报表，由于报表定义被缓存，因此加载速度非常快)，然后就可以看到在项目中定义的报表。

17.4　本章小结

Reporting Services 对许多 SQL Server 安装都有重要作用。对许多企业而言，在中央数据库建立一个相对健壮的报表服务器，会使得向数据用户传播信息变得更为自由和简单。而对另一些企业而言，Reporting Services 提供了替代长久以来一直采用的报表程序包(如 Crystal Reports)的解决方案。SQL Server 2008 增加了一些新的功能和控件，有助于生成更精美和更强大的报表，并且重新设计了引擎，提供了更大的可伸缩性。

本章的内容较为肤浅。报表可以参数化，可以嵌入数据库关系图，可以与其他产品整合(如 Microsoft SharePoint Services 或 Microsoft Office SharePoint Services)，从一个报表可以关联另一报表，甚至可以在报表中嵌入报表。

要了解更多有关报表的知识，建议阅读专门介绍 Reporting Services 的书籍。

www.TopSage.com

第18章

使用 Integration Services 进行集成

说明：

如果运行环境是带有 SQL Server 2005 的混合环境或者是从该版本迁移过来的，请不用担心。SQL Server 2008 Integration Services 会使用安装 SQL Server 2008 时安装向导中提供的 Legacy Services 安装运行旧的 DTS 程序包。

使用 SSIS Package Migration Wizard 有助于升级原有 DTS 程序包。

第 17 章介绍了 Business Intelligence Development Studio(有时也称作 BIDS)。重申一下，BIDS 是 Visual Studio 2008 开发环境，该开发环境带有一些特殊的面向 SQL Server 的项目模板和工具。以不同的方式利用该环境来开发 SQL Server，揭示了 Visual Studio 开发环境的灵活性——对比将会发现，较之于 Reporting Services，Integration Services 会带来全新的截然不同的体验。

本章将介绍如何使用 SQL Server Integration Services(通常简称 SSIS)执行基本的数据导入和导出操作，还将简要介绍利用 Intergration Services 工具可执行的其他一些可能操作。

18.1 理解问题

Integration Services 所要解决的问题都会以某种形式存在于大部分系统中——即如何从系统或外部数据源中导入或导出数据。可能是将旧系统中的数据导入到新系统中，或者从供应商(或其他人)那里导出供货列表。但是，所有这些事情共同的问题是需要将与表并不一定相符合的数据导入到表中。

这里需要的是可以把数据提取(Extract)、转换(Transform)和加载(Load)到数据库的工具——通常把这种工具称作 ETL 工具。这一工具能够处理的复杂问题是多样性的，但是 Integration Services 几乎可以处理任何情况。

注意：

人们可能会产生疑问，“既然 Integration Services 是内置的，为什么不是人人都使用它呢？”这取决于 Integration Services 在跨平台环境中的本能。有一些第三方程序包，这些程序包更加严格而且有更好的 UI 环境，确实可以让不熟练的用户相对简单地移动数据——但是移动数据的系统开销也非常昂贵。在使用原来的 DTS 产品时，我也遇到过面向 Oracle 或其他 DBMS 的

www.TopSage.com

客户，他们仅仅为了使用 DTS(SQL Server 的 ETL，在 SQL Server 2005 出现之前的叫法)而购买了 SQL Server 的完全授权许可。

18.2　使用导入/导出向导生成基本的程序包

SSIS 程序包实质上就是程序。该程序包捆绑了一套指令(可能包含了有限的条件分支逻辑)，以便进行移动、复制、编辑等处理。导入/导出向导是能够自动建立这样的程序包的工具，为的是实现相对简单的导入或导出。

要使用导入/导出向导，需要启用系统“程序”菜单中的 SQL Server Business Intelligence Development Studio 工具，并新建一个 Integration Services 项目(如图 18-1 所示)。

在最右边(假设没有移动过任何可停靠的窗口)，可以在“解决方案资源管理器”中找到一个命名为“SSIS 包”的节点。右击该节点并选择“SSIS 导入和导出向导”命令，如图 18-2 所示。

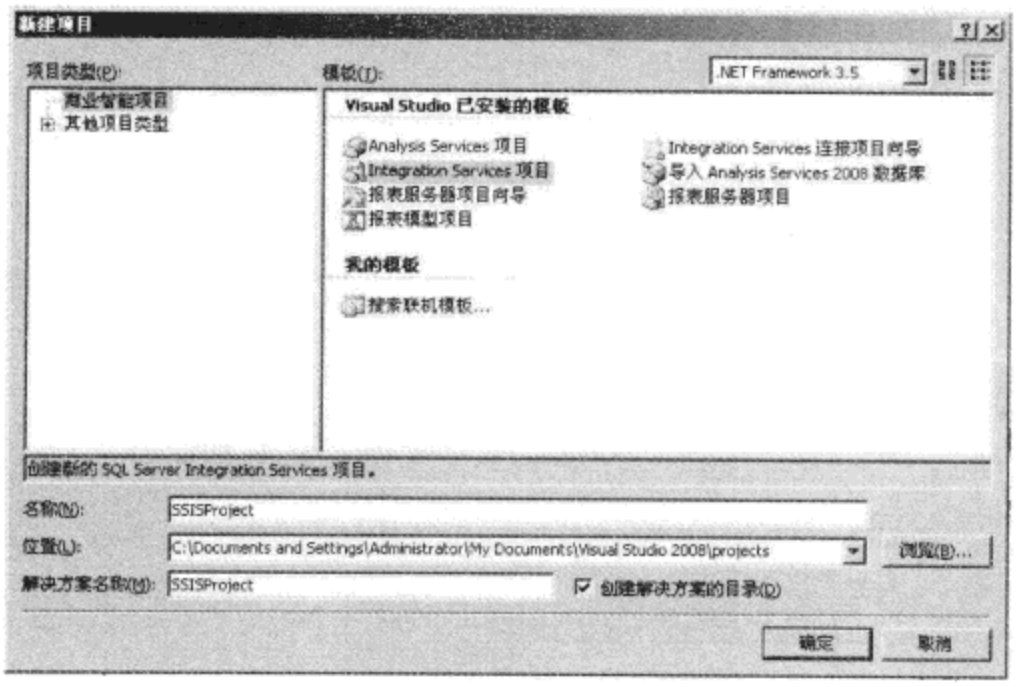


图　18-1

图　18-2

将弹出一个无用的说明性窗口(在第 17 章中常见的欢迎对话框相同)，单击“下一步”按钮，会弹出一个较重要的对话框，有助于建立与数据源的连接(如图 18-3 所示)。

注意：

在这一对话框中，Microsoft 又犯了用两种不同方式使用同一术语——数据源——的错误。第一种方式体现了数据源的全局性概念——包括了生成连接字符串的所有信息(服务器、身份认证、数据库等)。而第二种方式将数据源视为一个 OLEDB 驱动程序类型。

现在，从“数据源”下拉列表框中选择 SQL Server Native Client 10.0——这是访问 SQL Server 的首选方法。在该对话框中设置身份验证的信息(在前面的示例中使用了 Windows 身份验证)并选择 AdventureWorks2008 作为数据库。

现在单击“下一步”按钮，会看到一个基本上相同的对话框。然而，这一次需要做一点改动。将“目标”字段改为“平面文件目标”，如图 18-4 所示。其余选项变得更适合于基于文件

www.TopSage.com

系统的目标而不是 SQL Server 中的表。

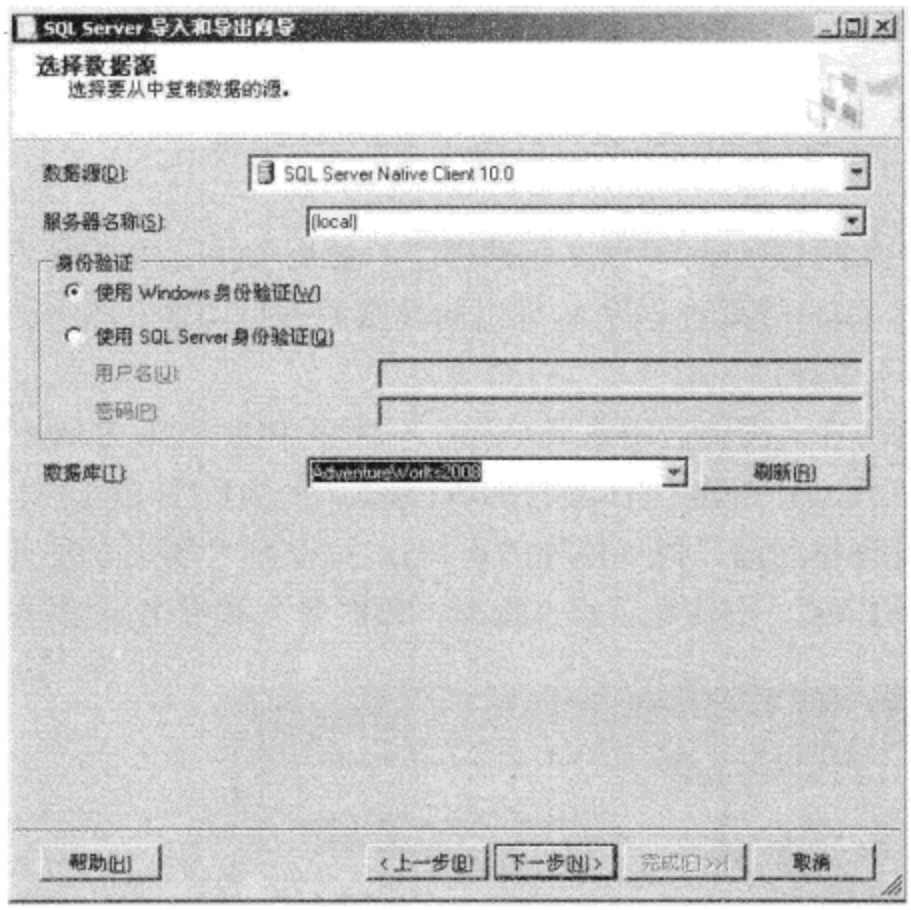


图 18-3

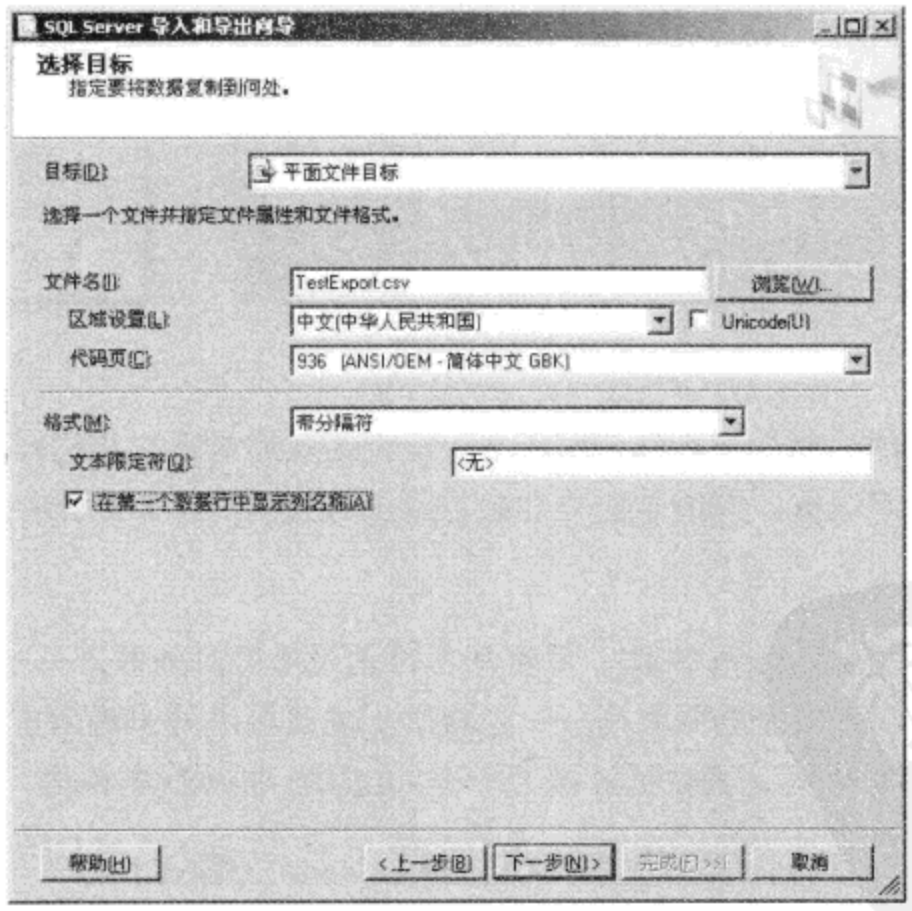


图 18-4

如图所示，这里将输出文件命名为 TextExport.csv，并按照符合普通逗号分隔文件的形式填写其他信息。

www.TopSage.com

注意：

这里需要说几句题外话，要务必小心使用逗号分隔的文件。这是一种“经过实践检验”的导入和导出数据的方法，也应仔细考虑所用的分隔符；数据中是否隐含了逗号？对导出的(或导入的)数据有怎样的影响？在决定格式时请不要忽视这些问题(但大多数时候，您没法选择)。

这里并不准备对此作详细的介绍，因为可用选项会根据所选的 OLE DB 数据源不同而发生很大改变。事实上，这里的关键在于 SSIS 会变更对话框以提供和所用的 OLE DB 数据源相关联的选项。在前一个示例中，选择了“在第一个数据行中显示列名称”选项，这时再次单击“下一步”按钮，系统便会提供从源数据库中选取数据的方式。这里有以下两种选择。

- **复制**：直接复制表或视图。
- **查询**：能够选择多种操作产生结果集。

先来看一下“查询”选项。如果选择了该选项并单击“下一步”按钮，那么会得到另一个对话框；这个对话框比较枯燥，因为只能输入查询文本，如图 18-5 所示。

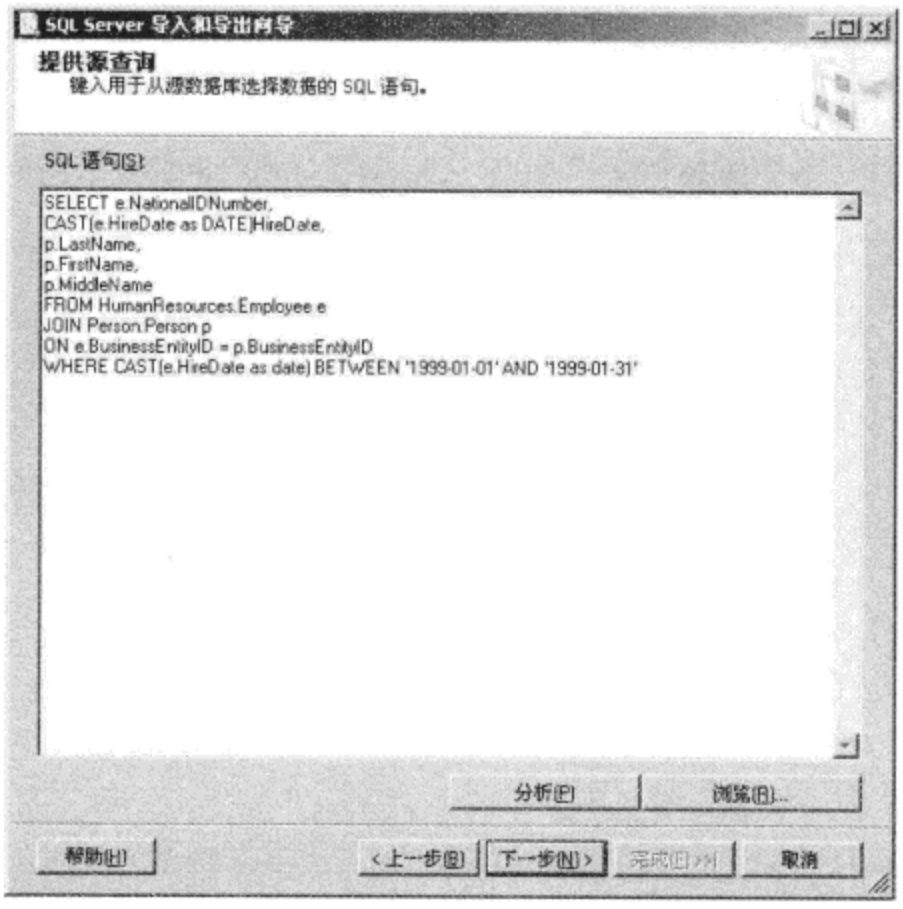


图　18-5

这里选取的例子是要导出公司第一个月雇佣员工的名单。如果想要测试，可以输入该查询，使用“分析”按钮确保待运行的查询的语法是正确的，或将其复制到 Management Studio 中的查询窗口内进行测试。然后单击“下一步”按钮得到如图 18-6 所示的对话框。

对于初学者来说，这一设置将决定最终平面文件的外观。可以很容易地改变行分隔符(例如，如果导出后将用于基于 Linux 或 UNIX 的系统，那可能要选择换行选项，而不是选择换行/回车选项)。可以单击“预览”按钮来查看该查询是否能够返回预期的数据(见图 18-7)。

单击“确定”按钮，回到“配置平面文件目标”对话框。

www.TopSage.com

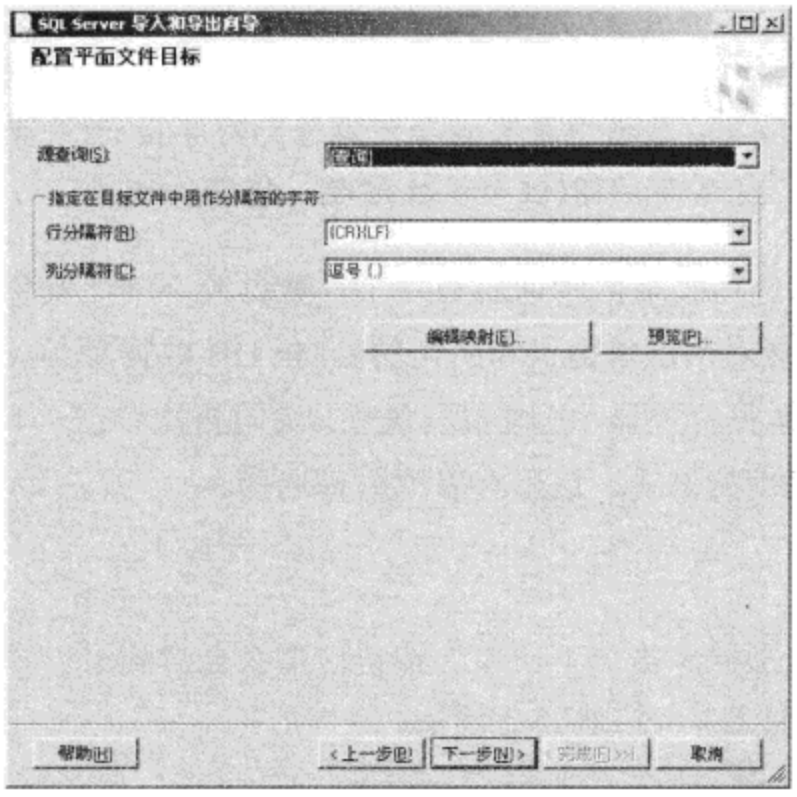


图 18-6

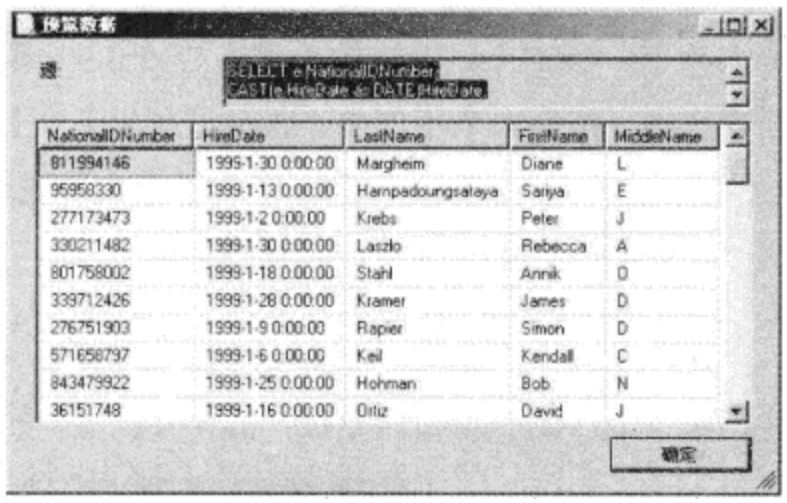


图 18-7

接下来，尝试单击一下“编辑映射”按钮，将出现如图 18-8 所示的对话框。

图 18-8

起初，这些内容可能显得较为平淡，但有趣的东西在后面。

试着单击一下“目标”列中的任意值。将会看到一个下拉选项，它包含了忽略该列的选项——实际上就是说，在输出文件时，从目标中删除“源”列。

注意：

有人可能会问：“如果准备在输出中省略该列，为什么还要将该列放入原始查询中呢？”原因是多方面的。首先，可能主要为了预览的目的而使用该列，为了验证得到的是否为预期数据——例如，当只想输出 EmployeeID 时，可包含人名来验证该人与其 ID 是否相符。另外，也

www.TopSage.com

有可能使用的是从一些其他来源处复制的查询，而自己又不想编辑这个查询(例如查询非常复杂)。

另外有一点是：这个对话框提供了直接复制表的选择(应该记得在选择“查询”时，还有“复制”选项)，所以这个选项在直接对表进行复制时更适用，因为在表中可能有许多您不希望出现在最终输出中的列。

然后，单击“类型”列的值，会看到很多的选择。

注意：

这里强烈建议调整对话框中列的大小，以方便查看“类型”列的内容。就像使用 Excel 时一样，将鼠标指针悬停在列标题的右边，然后单击并向右拖动列分隔线，以此来加大列的宽度。

大多数时候，要基于源数据类型使用默认转换，但是有时也可能需要以某种方式改变输出类型，例如，将整数数据当作字符串来满足对最终目标数据的预期(不是所有的系统都用同一种方式处理相同的数据)。

最后，您可能会对数据的“可以为空”和“小数位数”/“精度”选项感到困惑。用到这些选项的可能性很小，因为这些都是高级的概念，但如果想使用更为高级的转换(而不是使用默认的导入/导出向导)，那么这就是一个很有意思的选项，便于对数据进行错误捕获，使错误不会进入目标系统。然而在多数情况下，可以用原始查询中的 WHERE 子句来完成这个操作。

现在，单击“取消”按钮回到“配置平面文件目标”对话框，然后单击“下一步”按钮进入下一个对话框。

这时，将出现一个简单的确认对话框。该对话框将对向导在要求下执行的所有操作作概述，在本例中包括以下几点。

- 将行从查询复制到 C:\Documents and Settings\xxx\My Documents\TestExport.txt 中。
- 将要创建的新目标表。
- 将程序包保存到程序包文件“C:\Documents and Settings\xxx\My Documents\Visual Studio 2005\Projects\Integration Service Project1\Integration Services Project1\Package1.dtsx”中。
- 不要立即运行程序包。

其中多数内容都易于理解，但是这里需要以下强调几点。

首先，该对话框会以指定名称创建文件；当设计程序包时，如果检测到文件已经存在，那么就会有一些选项来指定是否覆盖现有的文件或者只是在文件末尾添加数据。

更为重要的可能在于这个程序包是专门为您创建的。该程序包相当于一个 SSIS 程序。可以使 SSIS 反复运行这个程序包(包括调度)，该程序包每次都能执行定义的导出工作。

最后，注意“不要立即运行该程序包”；这要求您手动执行或者调度执行。

单击“下一步”按钮，这样就创建了程序包。

注意：

再强调一次，这只是准备步骤；还并没有运行程序包，也没有导出数据。仍然需要执行或者调度这个程序包来获得实际的输出文件。虽然执行这个程序包和执行所有 SSIS 程序包一样，但这不是导入/导出向导特有的，所以请先记住这点，下一节会对此进行讨论。

www.TopSage.com

18.3 程序包基础知识

如果要全面讨论 SSIS 程序包的所有部分及其使用，那么恐怕要用上整本书的篇幅(事实上，已有了许多这方面的专门书籍)。但了解其基本内容对实现 Integration Services 最为基础的操作至关重要，所以下面将研究导入/导出向导刚刚创建的程序包。

假设还没有退出 Development Studio(也就是还打开着刚刚创建的程序包)，可以看到一个相当标准的 Visual Studio 窗口，与图 18-9 十分相似，只是在主面板上有一些与 Integration Services 相关的内容(主面板下面的“连接管理器”面板中也有)。

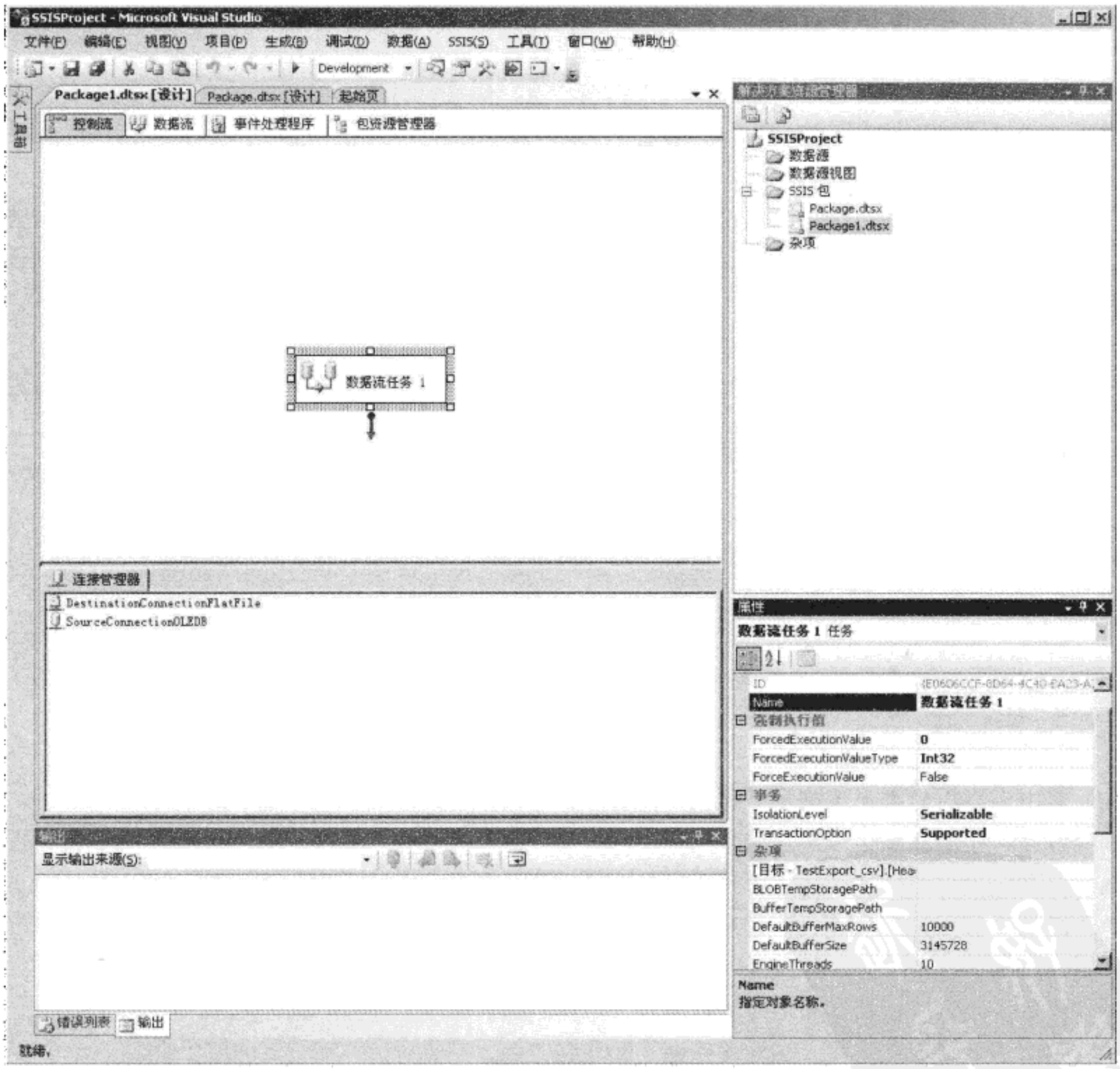


图 18-9

在研究 Integration Services 面板之前，先来看一下标准的可视化“解决方案管理器”面板(默认为在右上方，刚才设置导入/导出向导的地方)。其中有两个程序包。第一个程序包 Package.dtsx

www.TopSage.com

是在首次创建新程序包时选择了 Integration Services 模板而默认创建的程序包。该程序包是空的，可以删除。为与本节后面要讨论的内容吻合，这里选择 Package1.dtsx(选择后也可重命名)。

先来看几个重要的面板。

18.3.1　“连接管理器”面板

注意：

尽管您想要从头至尾了解它，但是我们只对“连接管理器”面板作简单介绍。因为它很简单；下面要重点介绍主面板。

您可能已经认识到“连接管理器”面板(见图 18-10)的作用，“连接管理器”面板有助于迅速访问程序包中用到的所有连接。

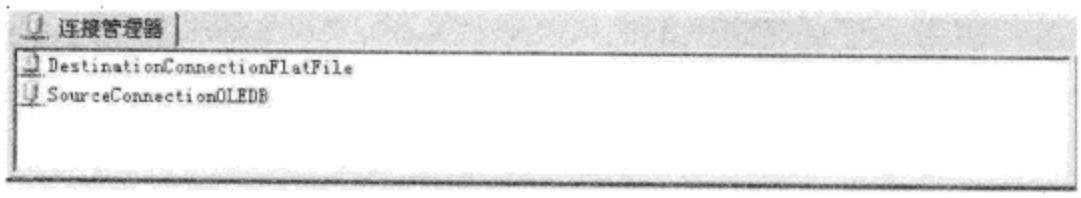


图　18-10

双击这两个连接管理器中的任何一个，会出现一个对话框，与 18.2 节中用到的对话框极为相似(参见图 18-3 和图 18-4)。但请注意，这两个对话框并不完全相同。实际上，“平面文件连接管理器”对话框含有许多相同的信息，但在编辑格式上有所不同。例如，图 18-11 显示了对话框中对输出的数据类型进行编辑的内容。

图　18-11

连接管理器是在没有向导的情况下编辑连接信息(输入和输出)的途径。

www.TopSage.com

18.3.2 程序包编辑器面板

这才是有关程序包的最重要的内容。18.2 节提到没有数据源连接就无法提取大量数据，但对于每个程序包而言，连接部分差异并不大。实际上，变化更大的是数据流处理方式不同。

如图 18-12 顶部所示，程序包编辑器含有 4 个表，下面一一进行分析。

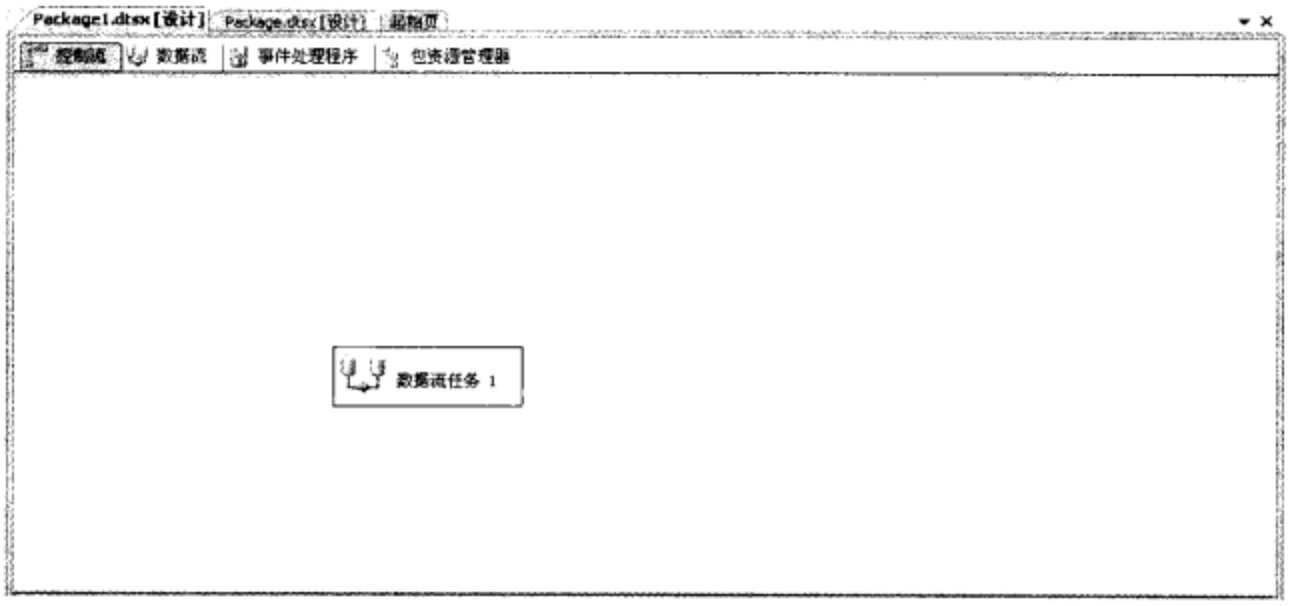


图 18-12

1. 控制流

顾名思义，控制流是指控制程序包中不同逻辑结构或“任务”间的数据流。在本例中，只定义了一个任务，见图 18-12 中的“数据流任务”，但如果选择了该任务(见图 18-13)，则会显示一个向外的箭头。

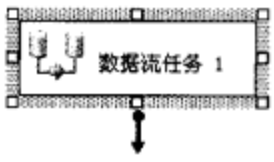


图 18-13

这个箭头代表了可能的流。比如，可以将一个 FTP 或外部处理任务添加到程序包中，以处理 ETL(提取、转换、加载)过程中在单个数据流任务中无法处理的元素。控制流在下列方面对程序包具有关键作用。

- 在这部分程序包中列出的流具有优先权。试想一下，在创建数据后，如果要用 FTP 将数据导出到外部系统中，那么需确保在程序包的导出任务完成之前，FTP 过程不能激活；通过对流进行控制可以实现这一点。
- 如果有多个任务要执行，而又不具有比前面的任务优先执行的特权(或是依赖于同一个任务)，则程序包允许将任务进行分支和同步执行。
- 根据上一个任务的具体返回情况不同(例如，当 FTP 成功时检索一个文件，当 FTP 失败时则通知操作员)，任务可以进一步分支。

根据任务的具体类型不同，可能编辑的内容会有很大不同，包括编辑复杂的连接和登录信息(像在之前提到的 FTP 任务中就有这样的需要)，或者无编辑内容(的确如此，当编辑数据流任务时，会直接进入“数据流”选项卡，与控制流完全无关)。

www.TopSage.com

2. 数据流

由于所给示例是基于一个数据流任务，因此要介绍一下这部分的相关内容。可以通过双击“控制流”选项卡中的数据流任务对数据流进行编辑，也可以选择“数据流”选项卡，从该面板顶部的下拉菜单中选择感兴趣的数据流进行编辑(如图 18-14 所示)。

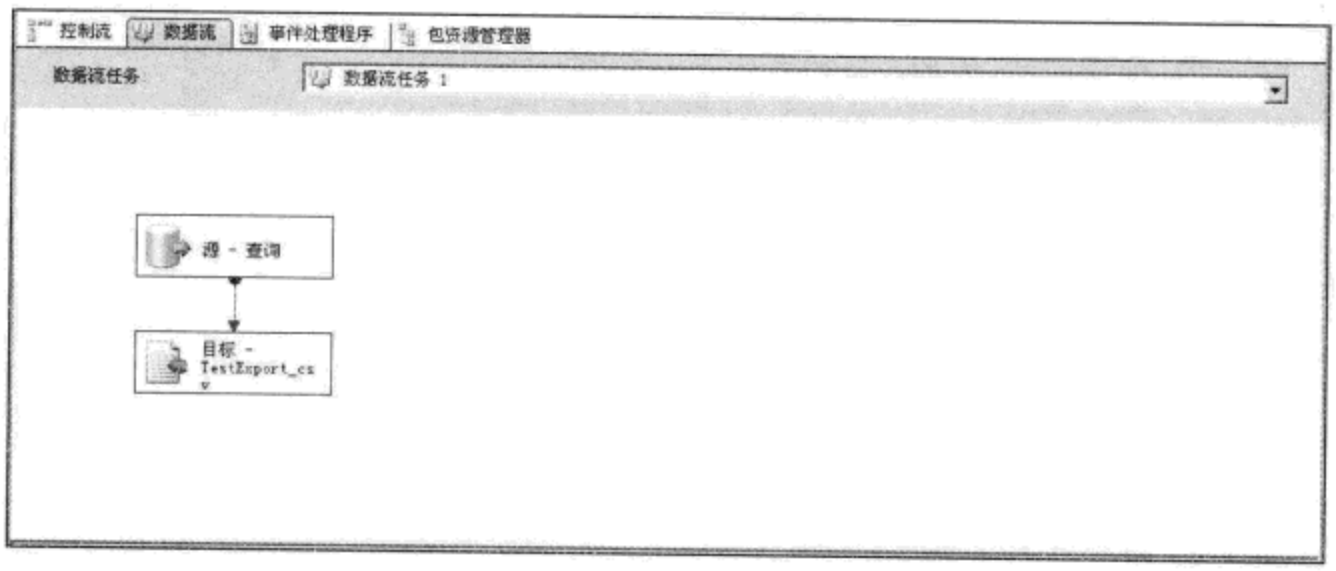


图　18-14

向导已经给出了一些数据流项，这里快速了解一下，以便将各个内容联系起来；双击“源-查询”流项，或右击后选择“编辑”命令。

进入如图 18-15 所示的“OLE DB 源编辑器”对话框。默认编辑选项为“连接管理器”，但请注意，虽然该对话框与之前的对话框有相同之处，但也有许多不同的地方。如果使用参数化查询，则将在此窗口中对参数属性进行编辑。

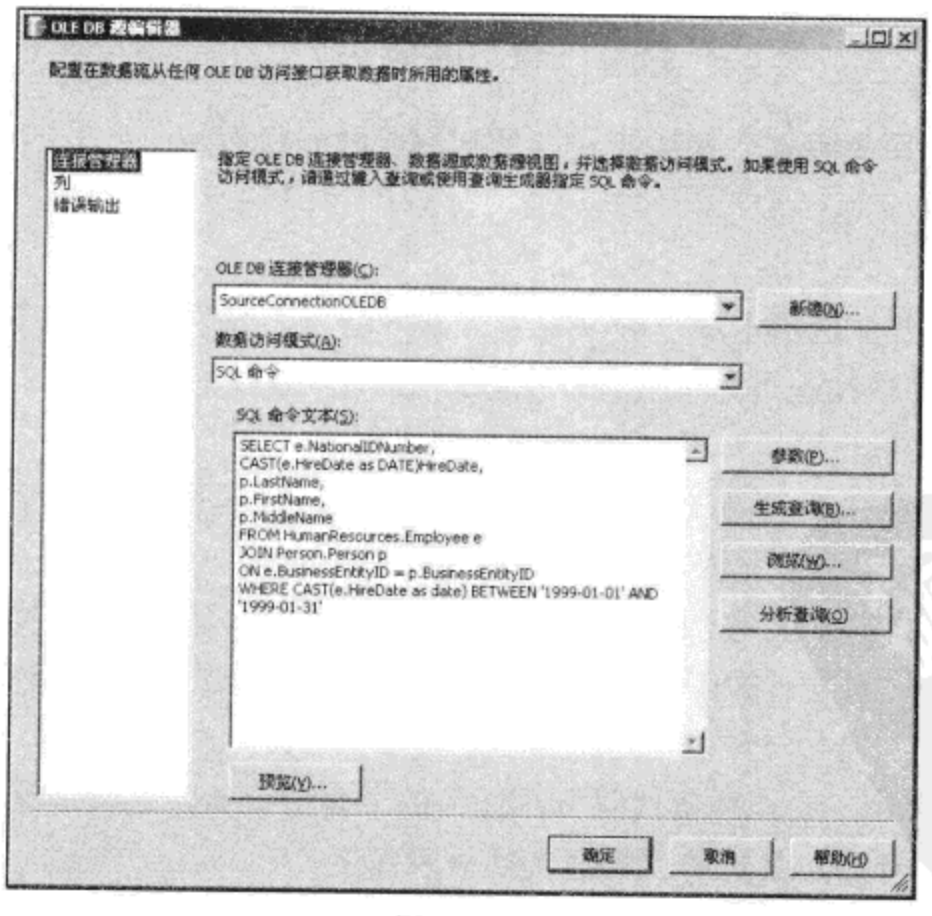


图　18-15

最后，选择对话框列表中的“错误输出”选项，如图 18-16 所示。该对话框最关键的作用

www.TopSage.com

就是为数据流错误的处理方式提供多种选择。在向导生成的这个简单的数据流项中，任何错误对程序包而言都是致命的，程序包将因为返回给流项调用的失败信息而终止(在本例中，控制流区域的数据流任务将依次传递失败，导致程序包终止)。其他选项则包括忽略错误(跳过行)或重定向行(可能是一个临时保存表，在重新尝试导入前，可在该表中可手动或以编程方式修复行)。对于更高级的程序包，可考虑使用这些选项。

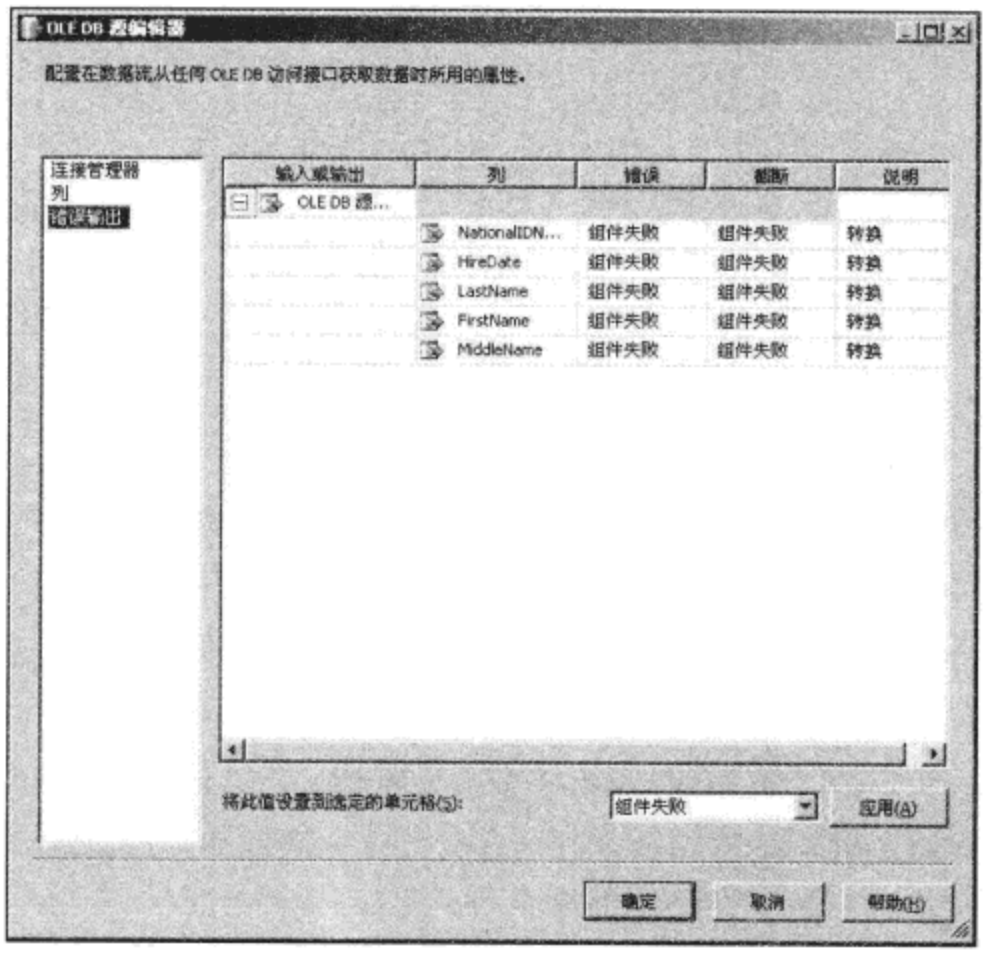


图 18-16

取消对话框中的所有设置，还原为向导最初完成时的状态。

3. 事件处理程序

在“事件处理程序”选项卡中，可以看到许多可以为之提供代码的事件。点击下拉菜单(如图 18-17 所示)，尽管默认选项为 OnError，但是可以选择其他多种选项。下面对其中几个可能用到的事件作一下讨论。

- OnError：这是最明显易懂的一种事件。正如您所想的那样，当检测到事件中有错误时，触发该事件。该事件允许以某种方式解决错误。
- OnWarning：基本上与 OnError 事件相同，但该事件允许检查和解决警告问题。
- OnProgress：对于长时间运行的进程，这可用于更新某种进度指示器(可能在某些地方是状态表)。
- OnPreExecute：允许准备代码等在任务的主要部分执行之前运行。

这里只是作简单介绍，具体使用的事件视需求而变。

www.TopSage.com

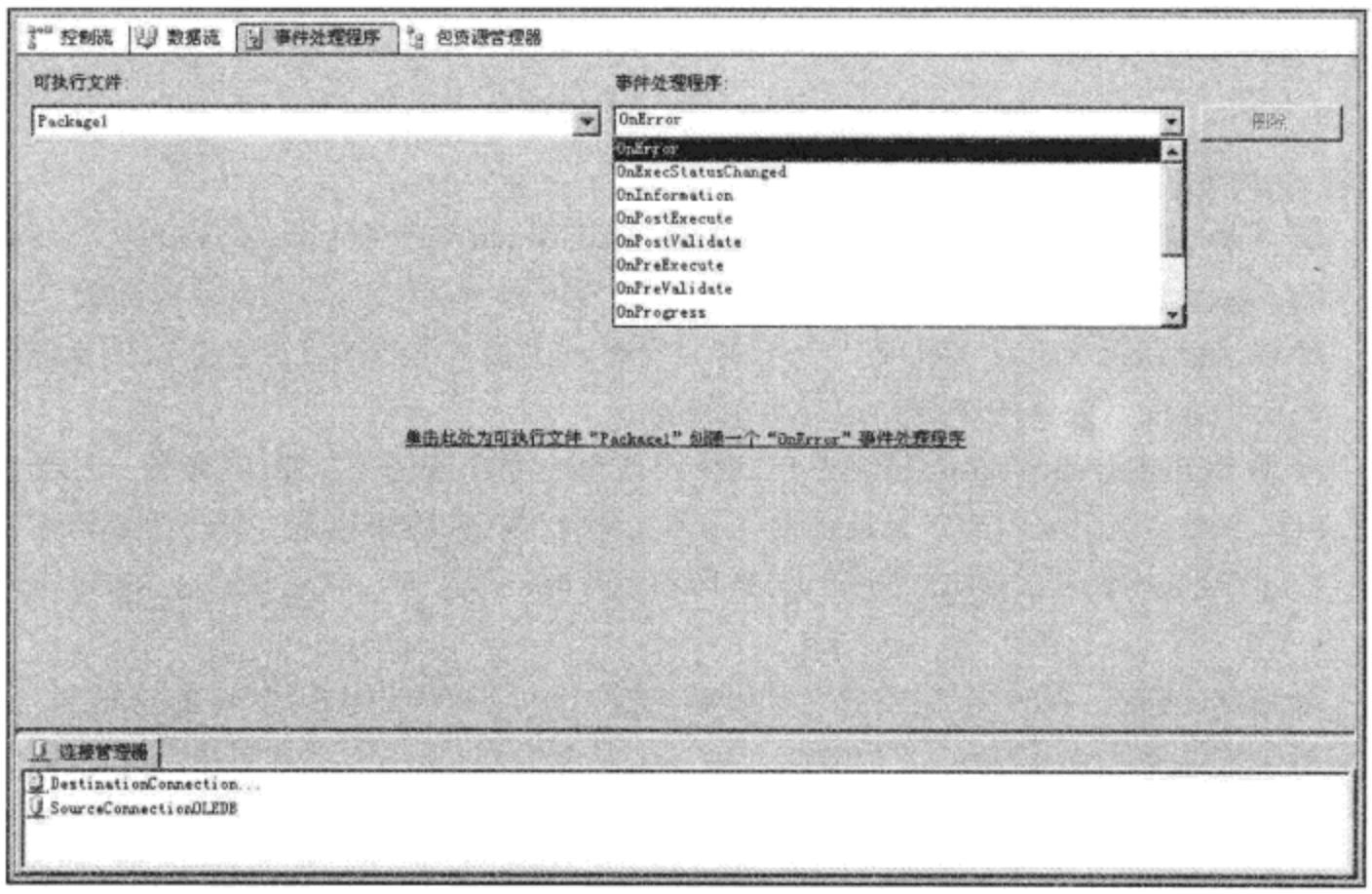


图　18-17

4. 包资源管理器

顾名思义，就像 Windows 资源管理器用于以基于文件夹的形式浏览文件系统一样，包浏览器也用于以类似的形式浏览程序包。图 18-18 展开了几个文件夹，以便更直观地说明问题。

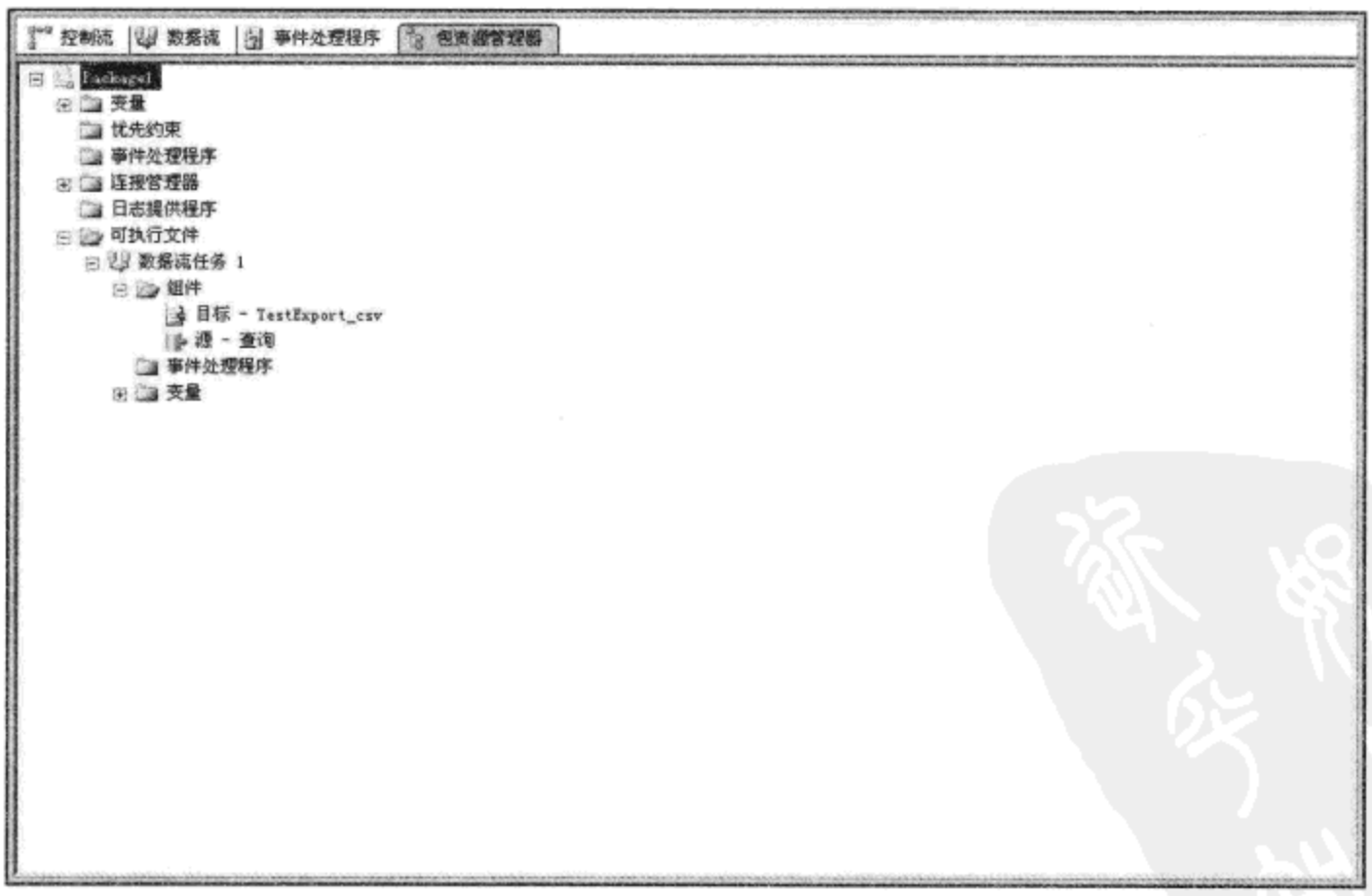


图　18-18

www.TopSage.com

18.4 执行程序包

执行 SSIS 程序包的方法有以下几种：

- **从 Development Studio 中运行**：和 Visual Studio 的任何其他实例一样，Business Intelligence Development Studio 允许在开发环境下运行程序包(但需具备调试能力)。
- **使用执行包工具**：这其实是一个可执行程序，在其中可以指定想要执行的程序包，按需要设置任何必要的参数并运行实用程序。
- **使用 SQL Server Agent 将程序包的执行作为一个调度任务**：第 19 章将介绍更多关于 SQL Server Agent 内容，目前需要认识到，执行 SSIS 程序包是该代理能够理解的各类作业中的一种。可以指定程序包的名称和运行时间及频率，SQL Server Agent 会处理这一切。
- **从程序中运行**：有一个完整的对象模型支持在程序中实例化 SSIS 对象的概念，设置程序包属性并执行程序包。其内容相当多——以至于 Wrox 用一整本书来介绍这个主题(请参考清华大学出版社引进并出版的《SQL Server 2005 Integration Service 专家教程》)。同样，这超出了本书的范围，这里只是让您对该主题有所了解，以便日后进行深入学习。

18.4.1 在 Development Studio 中运行程序包

这就像是在调试 Visual Studio 项目。只要按一下 F5 键，单击工具栏上的绿色箭头，或从“调试”菜单中选择“启动调试”命令。在构建时(假设成功构建)，会看到数据流任务先是黄色，然后变成绿色(表明运行成功)。如果程序包有多个步骤，则会依次触发每个任务，直至完成整个程序包。

继续照此运行程序包，检查生成的文本文件(其确切位置取决于是否在文件定义中指定了完整的路径或 Visual Studio 是如何配置的)。

18.4.2 使用执行包实用工具

执行包实用工具有两种形式：一种是名为 DTExecUI.exe 的 UI 驱动程序，另一种是称为 DTExec.exe 的命令行形式。这里选择 DTExecUI.exe。也可以使用 Windows 资源管理器进行导航，在文件系统中找到程序包(后缀名为.DTSX)，然后双击程序包并执行程序包。对最初的导出程序包采取这一操作，将得到图 18-19 所示的执行对话框。

可以看到，通过单击左边的各种选项可以选择很多不同的对话框。如果要对所有这些选项作介绍，就需要一本书的篇幅，所以这里仅介绍该工具中几个主要对话框上的重要内容。

1. 常规

这个对话框中的多数字段都易于理解，但是请特别注意一下“包源”字段。这里可以将 SSIS 程序包存储在以下 3 个位置。

- **文件系统**：这就是处理导入/导出向导程序包的方法。这个选项有很好的移动性；可以简单地将程序包保存并移动到另一个系统中。

www.TopSage.com

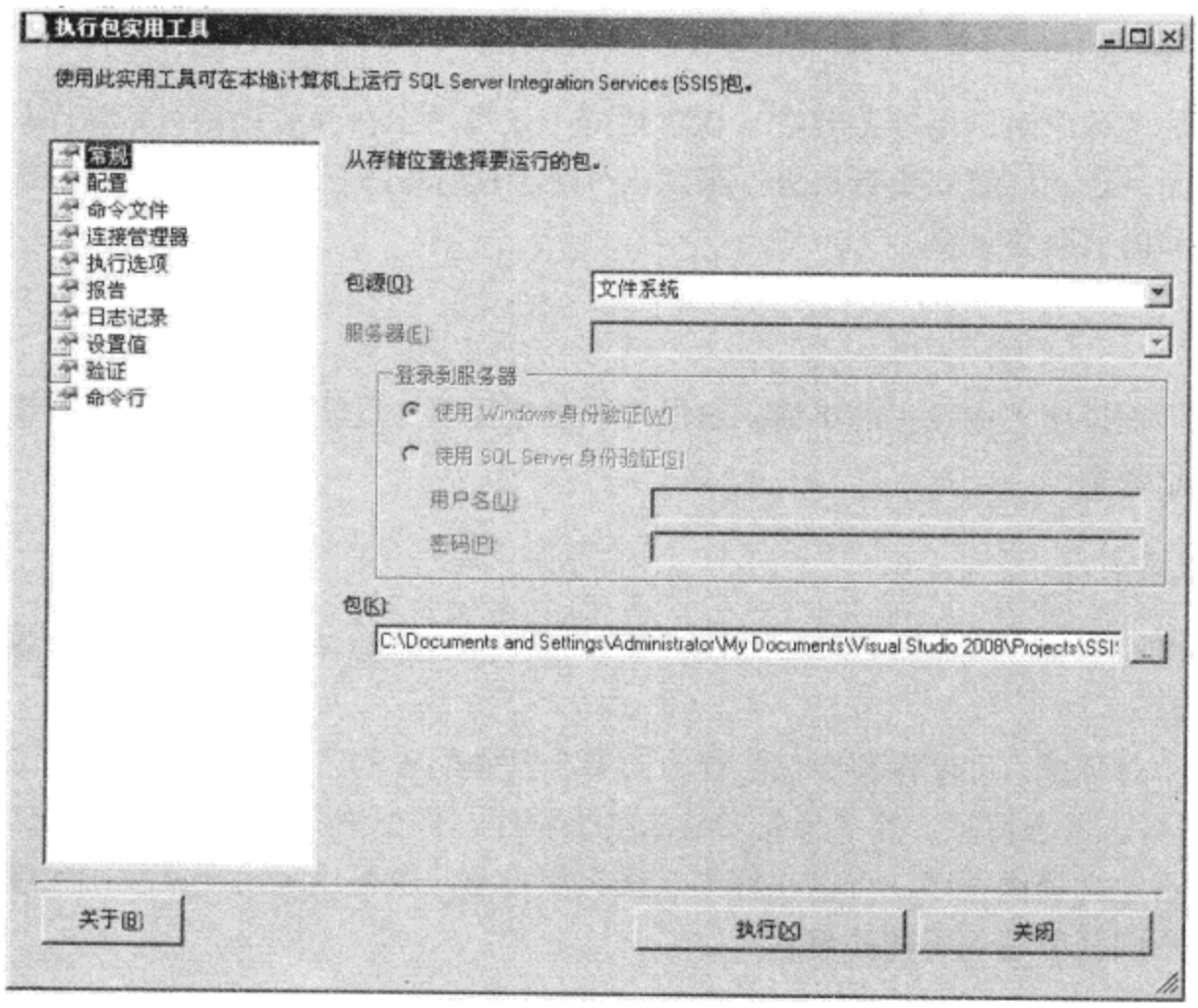


图　18-19

- **SQL Server**：这一模式将程序包存储在 SQL Server 中。使用这种方法，在备份 MSDB 数据库时(MSDB 数据库是每个 SQL Server 安装中的系统数据库)，也会备份这个程序包。
- **SSIS 包存储区**：这个存储模型提供了有组织的文件夹集合，在这个文件夹集合中，可以存储程序包以及其他有着相同类型或用途的程序包。该文件夹可以存储在 MSDB 数据库或文件系统中。

2. 配置

SSIS 允许为程序包定义配置信息。这些信息其实是所用设置的集合，可以将其中的几个设置组合为一个设置套件。

3. 命令文件

命令文件是将要作为程序包的一部分运行的批处理文件。可以使用这些文件来做系统级的操作——例如在需要的地方复制这些文件(命令文件会在 Integration Services 运行的任何账户下运行，所以任何对网络的访问权限都需要赋予这个账户)。

4. 连接管理器

这个名称有点用词不当。因为这并不是连接管理器的列表而是连接的列表。通过“说明”列，可以看到程序包使用的每一个连接的重要属性。本例中的程序包有两个连接，如果仔细观察，将会发现一个连接和文件信息相关(连接到所用的平面文件)，而另一个连接则专门与 SQL Server 相关(导出源的连接)。

www.TopSage.com

5. 执行选项

不要低估了这个对话框的重要性。该对话框不仅可用于在高层指定对错误(如果有错误)的处理方式，而且还允许建立检查点进行跟踪——便于查看程序包在何时到达哪个执行点。这对于性能调整和调试至关重要。

6. 报告

这个对话框说明正在发生的事情。这里可以设置反馈：反馈的多少取决于想要跟踪的事件和建立的信息级别。

7. 日志记录

这个对话框的设置相当复杂，但是能提供非常灵活的跟踪体系结构，即使是最复杂的程序包也没有问题。

使用这个对话框，可以将程序包配置为向多个预先配置的“提供程序”(实质上是明确的日志数据目标)写入日志信息。除了预先安装的提供程序，例如文本文件、甚至 SQL Server 表，还可以创建自定义提供程序。可以在程序包级记录日志，也可以采用非常细的粒度级别，对于程序包中的不同任务写入不同的位置。

8. 设置值

这个对话框为程序包使用的所有运行时属性设置起始值(在简单的程序包中没有这些起始值)。

9. 验证

完全不同的程序包可以有相同的文件名(比如在文件系统中的不同位置中)。另外，程序包可以在相同的文件或程序包存储中保留自己不同的版本。“验证”对话框用于筛选或确认要执行的程序包/版本。

10. 命令行

可以使用命令行来执行 SSIS 程序包(当试图不通过批处理文件运行 DTS 程序包时，命令行很有用)。如果已经从命令行运行了程序包，那么 SSIS 执行包实用工具中的这个选项则用来指定将要使用的参数。

这个实用程序将指定大部分内容；这里的选项只是允许在通知实用程序执行时对使用的选项进行重写。

11. 执行程序包

如果单击“执行包实用工具”对话框中的“执行”按钮，程序包就会运行。运行之后，可以在存储该程序包的位置上发现一个文本文件；可以打开该文本文件，确认其就是期望得到的文件。

www.TopSage.com

18.4.3　使用 SQL Server Agent 执行程序包

到目前为止，本书并没有真正地对 SQL Server Agent 进行讨论，但是，从 SSIS 的角度来看，可以将 SQL Server Agent 看作是一个作业调度器，用于完成和运行 DTExec.exe 一样的操作，但是按特定的时间和频率进行的，并且用非常健壮的作业任务调度系统将程序包集成到一个更大的系统工作集中。

有关代理的更多细节将在第 19 章中谈到。

18.4.4　从程序中执行程序包

SQL Server 提供了一个在任何.NET 语言环境下都可以使用的多功能对象模型。编程对象模型超出了本书的范围，但只要知道，不仅可以通过编程方式执行程序包，还可以在执行之前动态构建程序包以满足特定需求。

18.5　关于程序包的补充说明

再次重申本书仅探讨了浅显的表面内容。如前文所述，有很多大部头的书籍都是专门以 Integration Services 为主题的。

这里所讨论的程序包是由导入/导出向导生成的，但最终产品是一个典型的 SSIS 程序包。不仅可以对其进行编辑，还可以自己从头开始创建。可以轻松地添加其他任务来进行处理工作，如在将数据导入到某种“临时”区域后“净化”数据等，然后执行其他任务。也可以添加复杂的分支来处理错误或并行执行多个任务；等等。

18.6　本章小结

SQL Server Integration Services 是一个健壮的 ETL 工具。可以使用 Integration Services 一次性地或重复性地从数据库中导出数据或者向数据库导入数据——混合各种数据源。

虽然想要成为 Integration Services 方面的专家需要付出巨大的努力，但是实现基本的导入和导出操作相对来说比较简单。本书鼓励您从简单的着手，不断充实内容。在不断深入了解 SSIS 功能的过程中，可参考其他一些关于专门介绍 SSIS 的书籍。

www.TopSage.com

第19章

扮演管理员的角色

至此，本书的内容接近尾声。但是，这也是一个开始。接下来要讨论的是：对开发的数据库进行维护和管理。

作为开发人员，有人会说“这不是数据库管理员的工作吗？”如果您也说过类似的话，就应该受一点教训了。有关数据库开发，本书想要给您灌输的理念是，那就是要避免抱有“事不关己，高高挂起”这样一种普遍存在的态度。

数据库驱动的应用程序与大多数单机应用程序大为不同。大多数单机应用程序或者自行维护，或者处理单个文件，用户可以出于备份的目的将这些文件轻松地复制到某个地方。同样地，单机应用程序通常不存在维护的问题，这一点和数据库也不同。

本章将介绍一些任务，这些任务能够确保终端用户不但能从故障和灾难中恢复，而且还能够完成一些基本的维护工作，帮助系统更顺畅地运行。

本章包括如下内容：

- 调度作业
- 备份和还原
- 执行基本的碎片整理和索引重建
- 使用警报
- 使用 Policy Based Management
- 归档

虽然管理任务远不止这些，但是这些任务确实也反映了对应用程序进行部署时需要解决的一些最基本的问题。

对于本书来说，下面将主要介绍 SQL Server Management Studio 中包含的 GUI 工具(有关 Policy Based Management 这一新内容将作简要介绍)。但是要知道，对于高级用户来说，还有更为强大的选项，这些选项允许使用 SMO(SQL Server Management Object)对象模型以编程的方式控制管理功能。

www.TopSage.com

19.1　调度作业

在接下来的章节中要介绍的许多任务都是可调度的。调度作业允许在非高峰时段运行给系统带来一定负载的任务。调度作业也确保不会遗忘某些需要完成的任务。从索引重建到备份，您可能会听到很多“忘记去做”的故事，或者已经建立了可调度的作业但是从来不对作业进行检查。

注意：

如果后台在 Windows Server 中，而且已经使用了 Windows Scheduler 服务来对其他作业进行调度，那么可以使用该调度引擎支持 SQL Server。全部在 Windows Scheduler 中进行可以把所有事情集中到一个地方，但是 SQL Server 提供了一些更为强大的分支选项。

这里要考虑两个术语：作业(job)和任务(task)。

- **任务**——任务是待执行的单个进程，或是待运行的一个批处理命令。任务是非独立的，任务只作为作业的成员而存在。
- **作业**——作业是一个或多个应该运行的任务的组合。但是，也可以根据单个任务执行成功与否来设置相关性和分支(例如，如果前一个任务成功，则运行任务 A；如果前一个任务失败，则运行任务 B)。

可以根据下面的标准来对作业进行调度。

- 每日、每周或每月
- 一天中的特定时段
- 一个特定的频率(如每 10 分钟或者每小时)
- 当 CPU 空闲一段时间时
- 当 SQL Server Agent 启动时
- 对警报的响应

因为任务是作业的一部分，任务会根据对作业所定义的分支规则来运行。运行一个作业并不意味着该作业的所有任务都将运行；根据作业中上一个任务的成功与否和已经建立的分支规则，一些任务会运行，而另一些任务则不会运行。SQL Server 不但允许在一个任务完成时自动触发另一个任务，而且如果当前任务失败，SQL Server 还允许执行一些完全不同的操作(例如运行某种恢复任务)。

除了可以有分支以外，也可以根据发生的情况要求 SQL Server 执行以下任务。

- 将作业成功或者失败的消息通知给操作员。可以通过网络消息(只要用户登录，就会在用户屏幕上弹出网络消息)、寻呼程序和电子邮件向每位操作员发送通知。
- 将信息写入事件日志。
- 自动删除作业(防止后面继续执行，一般“清除”)。

下面将简要介绍如何在 Management Studio 中创建操作员。

19.1.1　创建操作员

如果想使用 SQL Agent 的通知功能，那么必须设置一个操作员来定义向谁通知具体定义细

www.TopSage.com

节。操作员的创建过程通常并不是通过任何自动过程或是作为开发代码的一部分来完成的，而是一般由 DBA 手动创建。下面将对此作简要的介绍，这里只要理解创建操作员如何与任务调度协同工作。

1. 使用 Management Studio 创建操作员

如果要使用 Management Studio 来创建操作员，就需要导航到想要为其创建操作员的服务器的“SQL Server 代理”节点上。展开“SQL Server 代理”节点，右击“操作员成员”，并选择“新建操作员”命令。

注意：

根据特定的安装，SQL Server Agent Service 可能不是默认自动启动的。如果碰到了任何问题，或者注意到 Management Studio 中的“SQL Server 代理”图标内有一个小红框时，那么可能该服务被设为手动或者甚至禁用；此时可能需要修改该服务，使它自动启动。无论如何，要确保它可用于本章中的示例、试一试和练习。通过右击：“SQL Server 代理”节点并选择“启动”命令，可以启动服务。

这会弹出如图 19-1 所示的对话框(这里已经填写了该对话框的部分内容)。

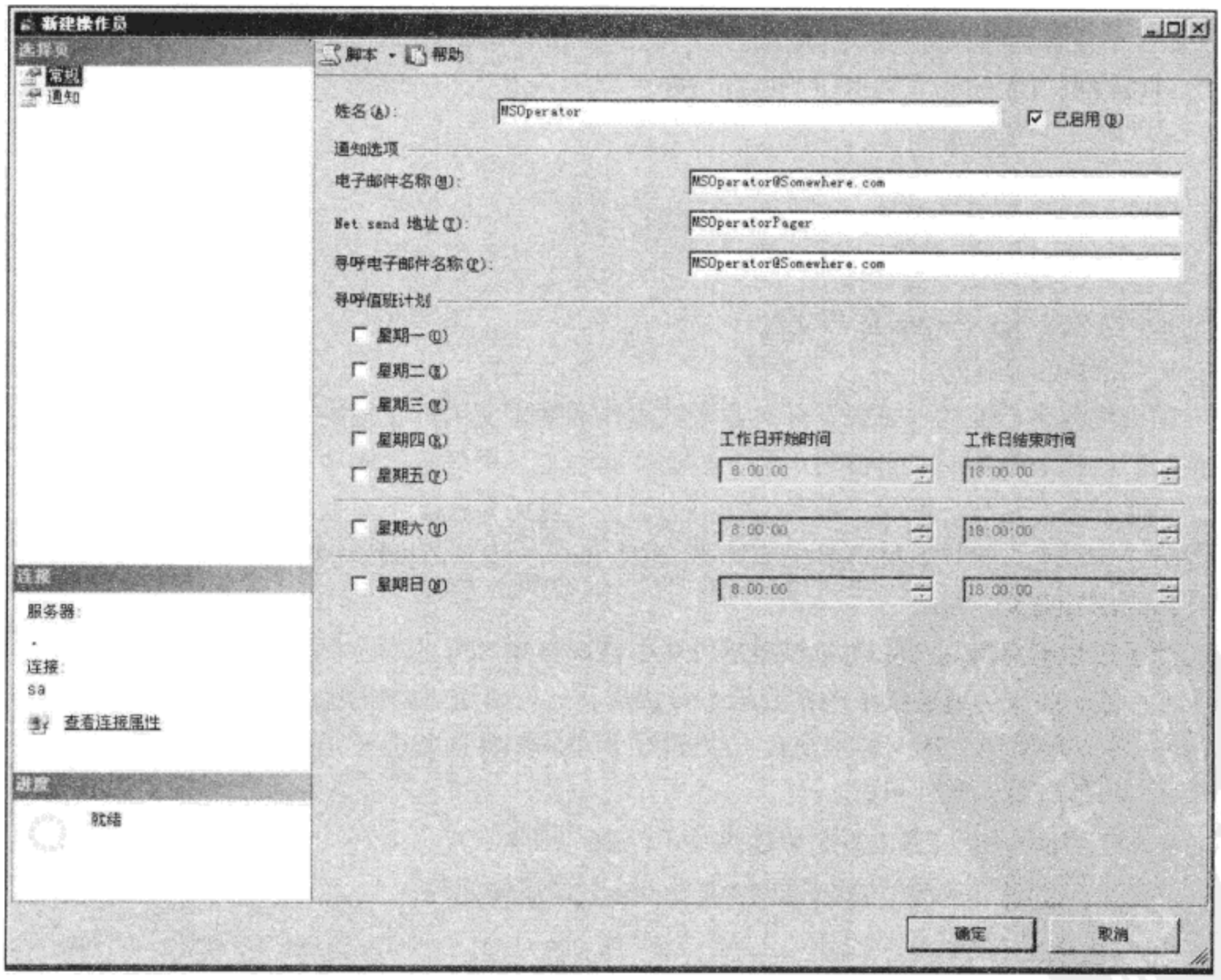


图 19-1

www.TopSage.com

接下来可以填写调度，指定这个操作员何时接收“通知”选项卡中特定类型错误的电子邮件通知。

谈到这个“通知”选项卡，可单击该选项卡，结果如图 19-2 所示。

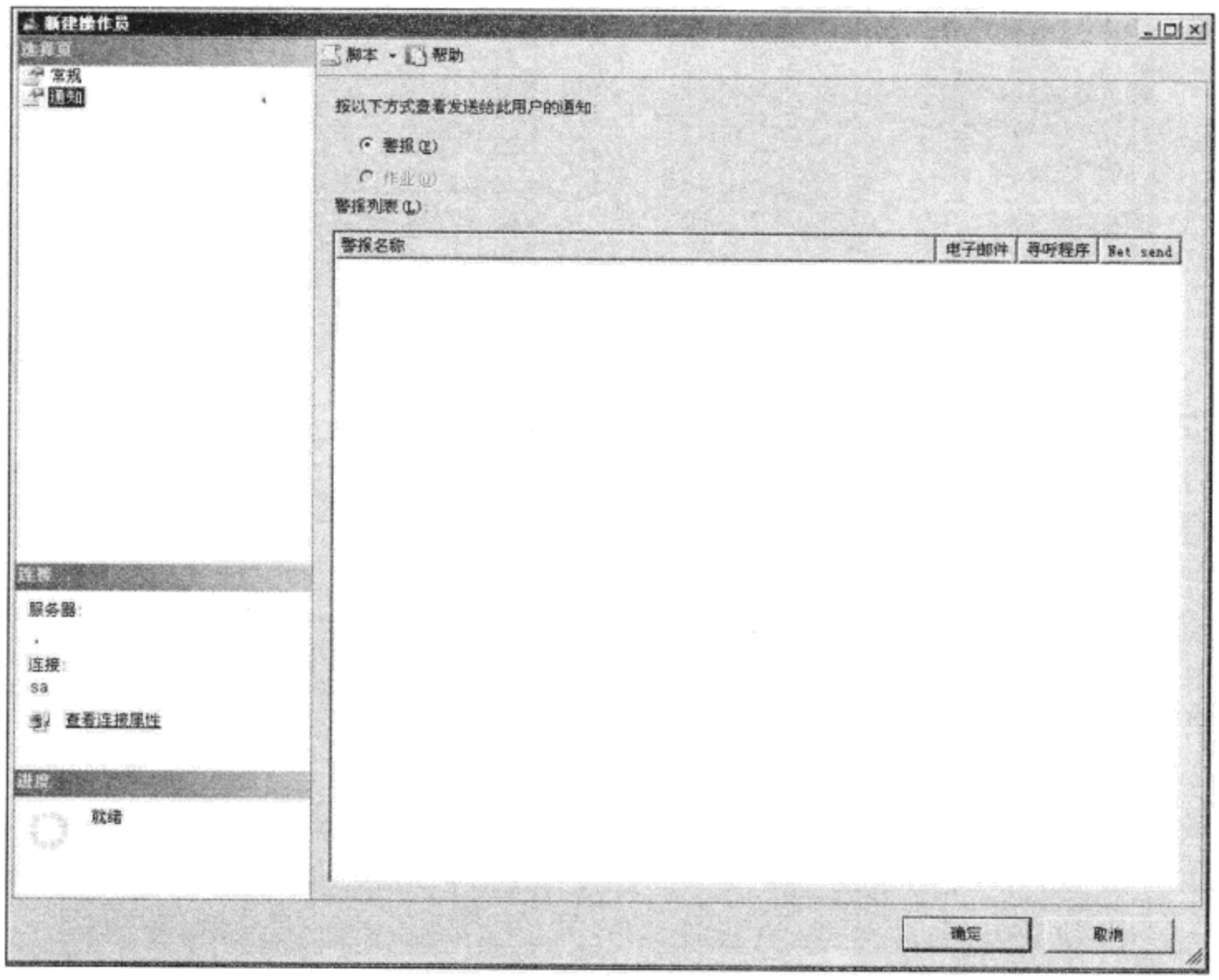


图　19-2

除非在系统中有更多的警报(本章的后面将介绍)，否则这个页面没有太多意义。这个对话框用于根据被触发的定义过的警报为该操作员设置接收何种通知信息。同样，在没有介绍警报前想要理解这个概念是比较困难的，但是要知道，警报会在数据库中发生某些情况时触发，而这个页面定义了这个特定的操作员要接收哪些警报。

19.1.2　创建作业和任务

如前所述，作业是一个或多个任务的集合。任务是一个逻辑工作单元，例如备份一个数据库或者运行 T-SQL 脚本来满足像重建所有索引这样的特定需要。

即使作业可以包含多个任务，也无法保证作业中的每个任务都会运行。它们会根据作业中其他任务的成功与否和对每种成功或失败定义的响应来决定是否运行。例如，如果作业中一个任务失败，那么可以取消剩余的任务。

和操作员一样，作业的创建可以在 Management Studio 中进行，也可以通过编程实现。就本书而言，这里将使用 Management Studio 方法(即使对于高级的 SQL Server 开发人员来说，以编程方式创建作业和任务也很少见)。

www.TopSage.com

1. 使用 Management Studio 创建作业和任务

在 SQL Server Management Studio 中创建可调度的作业很简单。只要导航到服务器的“SQL Server 代理”节点上。然后右击“作业成员”并选择“新建作业”命令。这会得到一个多模式对话框，如图 19-3 所示，它可帮助逐步建立作业。

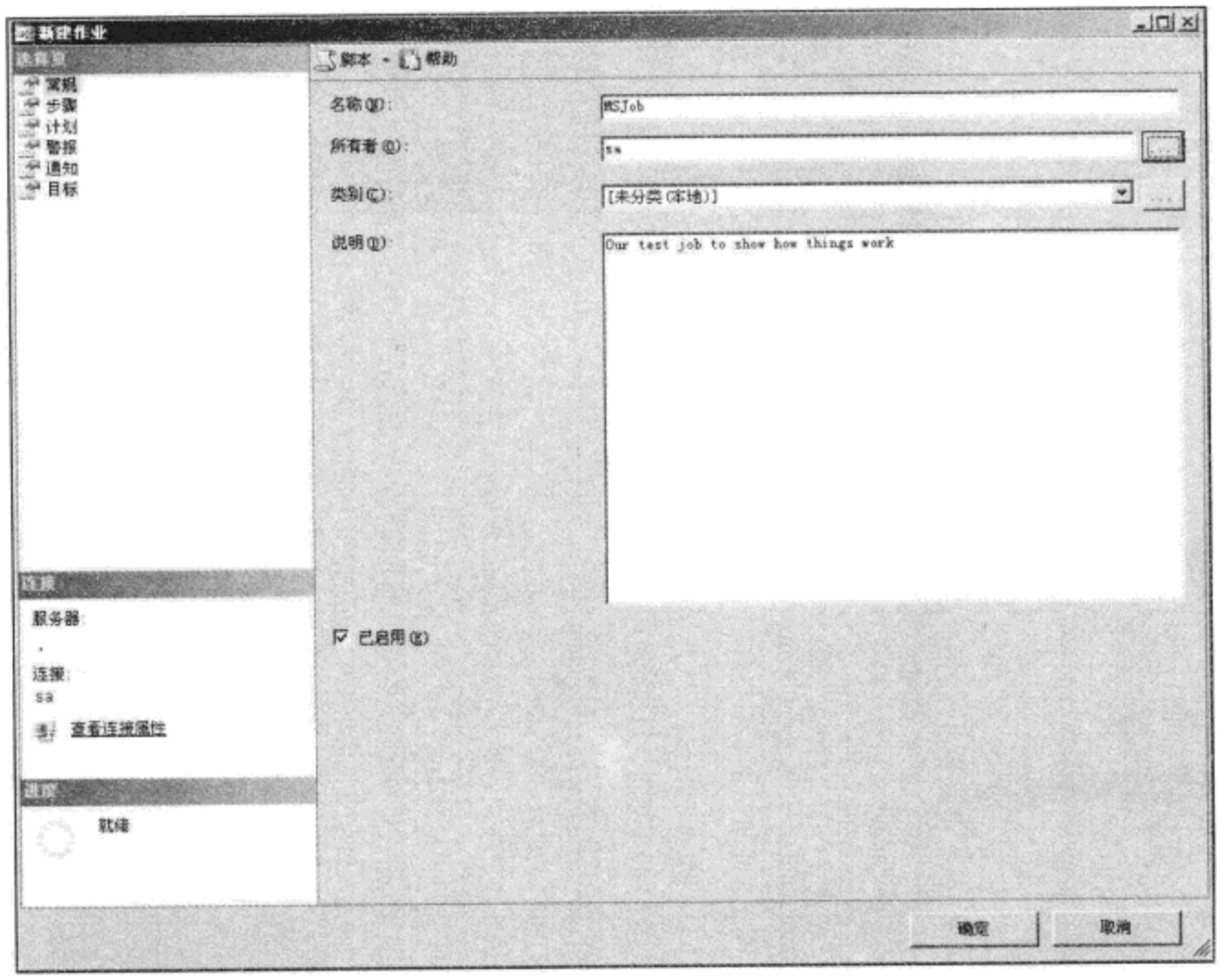


图 19-3

名称可以是任意的，只要名称符合本书前面介绍的 SQL Server 命名规则。

除了“类别”选项之外——这是组合作业的一种方法，剩下来的大部分信息还是比较容易理解。应用程序中许多特定的作业都是未分类的，虽然可能碰到想要创建数据库引擎优化顾问、数据库维护、全文或者 REPL-警报响应作业的情况；为了容易识别起见，任务都各自归类了。

接下来是“步骤”选项，如图 19-4 所示。这里可以告诉 SQL Server 开始创建将作为这个作业一部分的新任务。

为了给作业添加一个新的步骤，只要单击“新建”按钮，并且填写新的对话框，如图 19-5 所示。这里将使用一个 T-SQL 语句来引发一个假的错误，这样就能看到调度作业时发生的操作。但是也请注意，在命令框的左边有一个“打开”按钮；可以使用该按钮导入保存在文件中的 SQL 脚本。

下面将介绍该对话框的“高级”选项卡，如图 19-6 所示；这里可以看到作业调度程序所提供的一些很好的功能。

www.TopSage.com

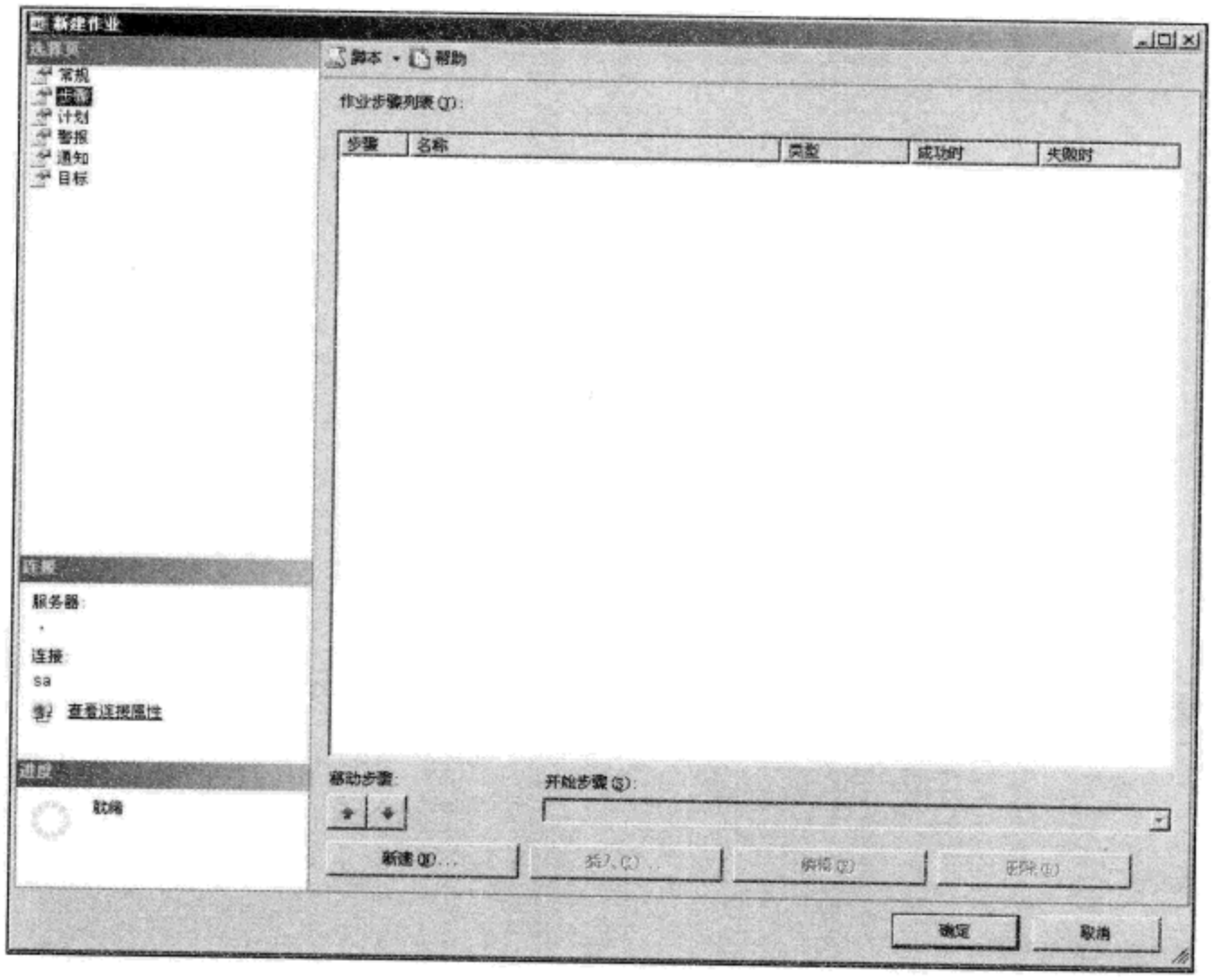


图　19-4

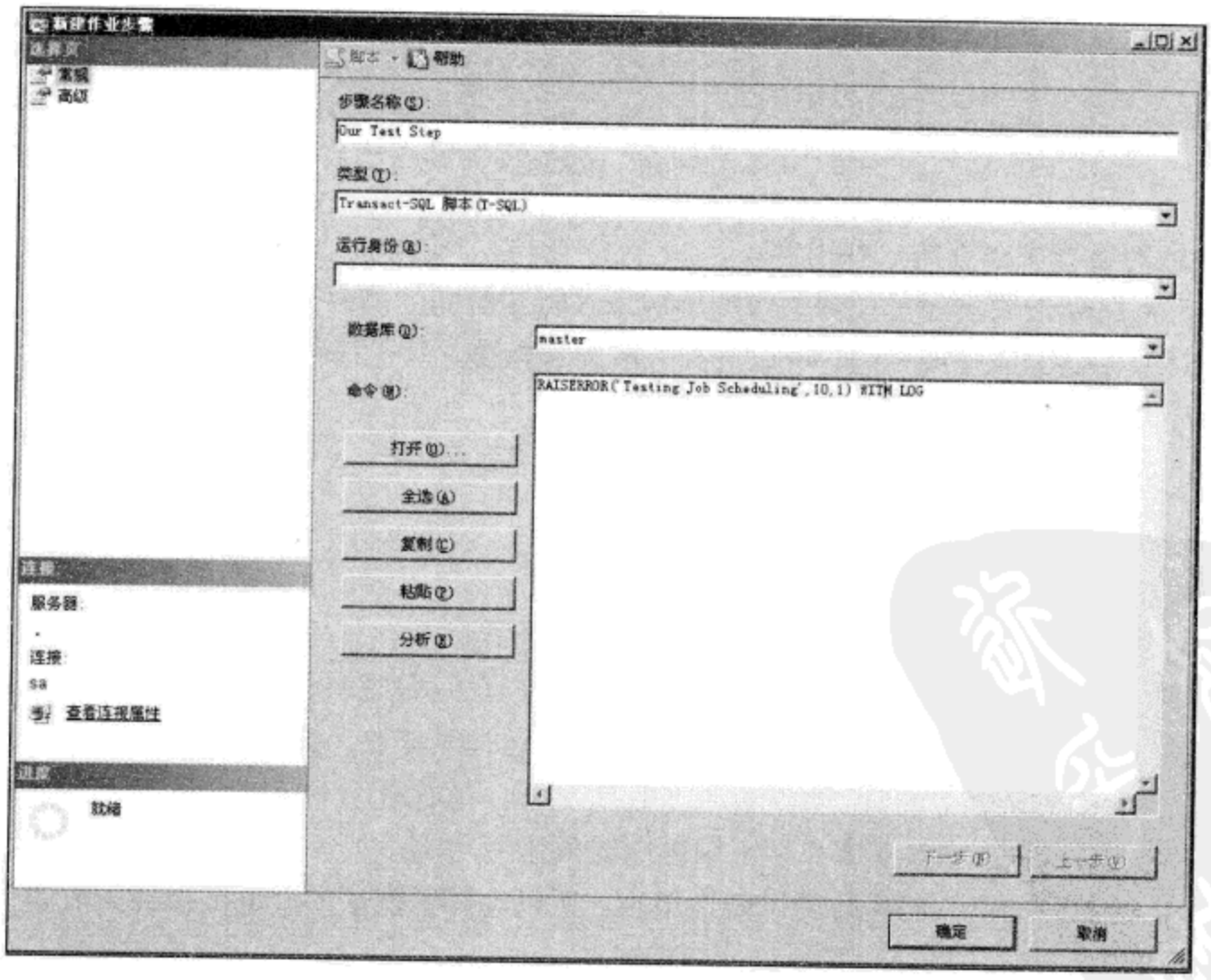


图　19-5

www.TopSage.com

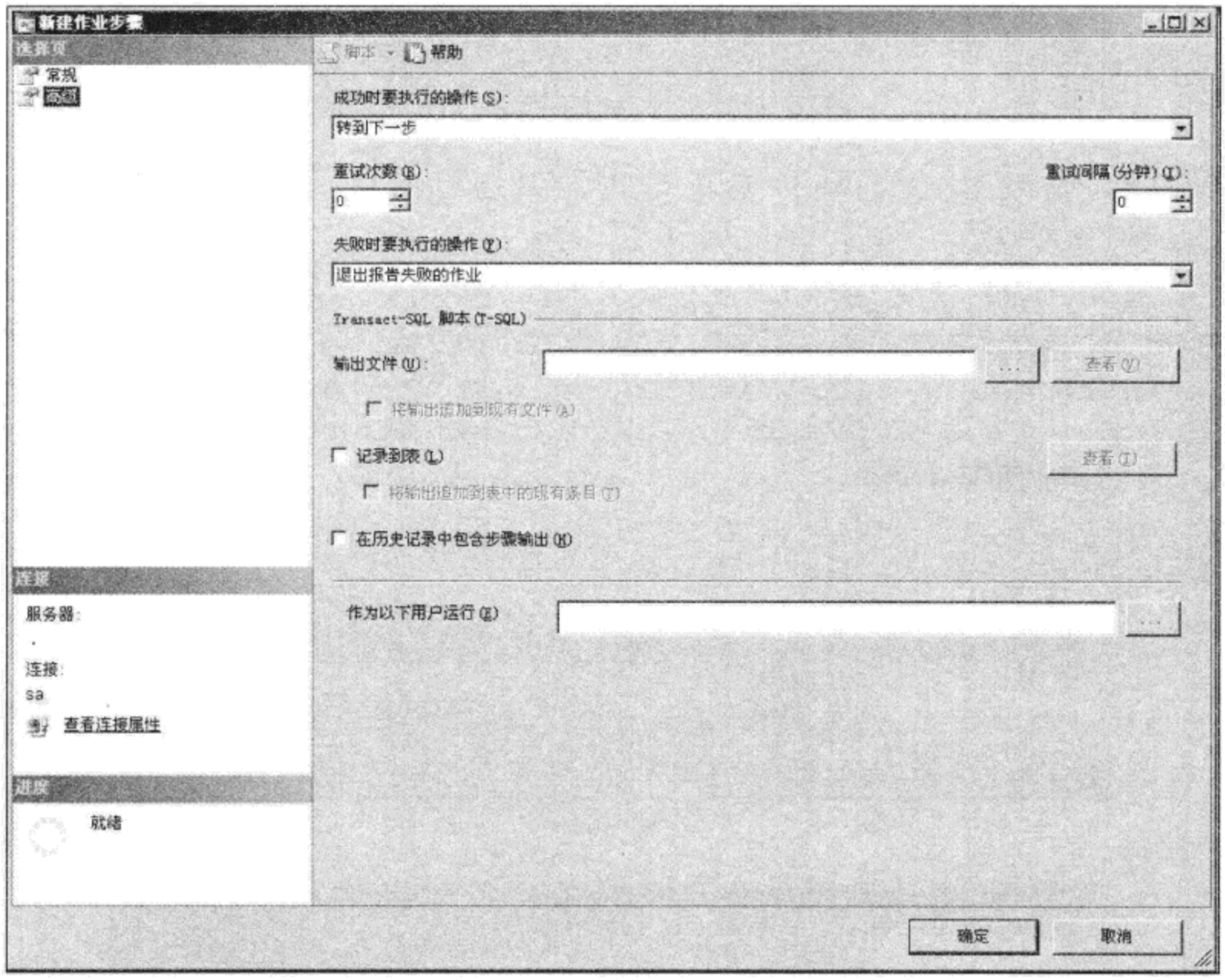


图 19-6

请注意对话框中以下的一些内容。

- 可以自动设置作业在任务失败的情况下在特定时间间隔内重新尝试。
- 对于作业的成功或失败，可以选择采取何种措施。对于每种结果(成功或失败)，可以执行以下操作。
  - 退出并报告成功
  - 退出并报告失败
  - 进入下一个步骤
- 可以将结果输出到文件中(对于审计来说很有用处)
- 可以模拟另一用户(以获取权限)。注意，必须有该用户的权限。因为是以系统管理员身份登录，所以可以作为 dbo 或者其他任何人运行该作业。普通用户可能只有 guest 账户可用(除非他们是 dbo)，但是在大多数情况下，普通用户不应以这种方式对他们自己的作业进行调度(可以让客户端应用程序提供这个功能)。

由于 RAISERROR 语句失败的可能性很小，所以选择了默认的“退出报告失败的作业”选项(在后面谈到备份时还将介绍其他可能的情况)。

www.TopSage.com

这样就回到了“新建作业”主对话框，现在移到“计划”节点上，如图 19-7 所示。

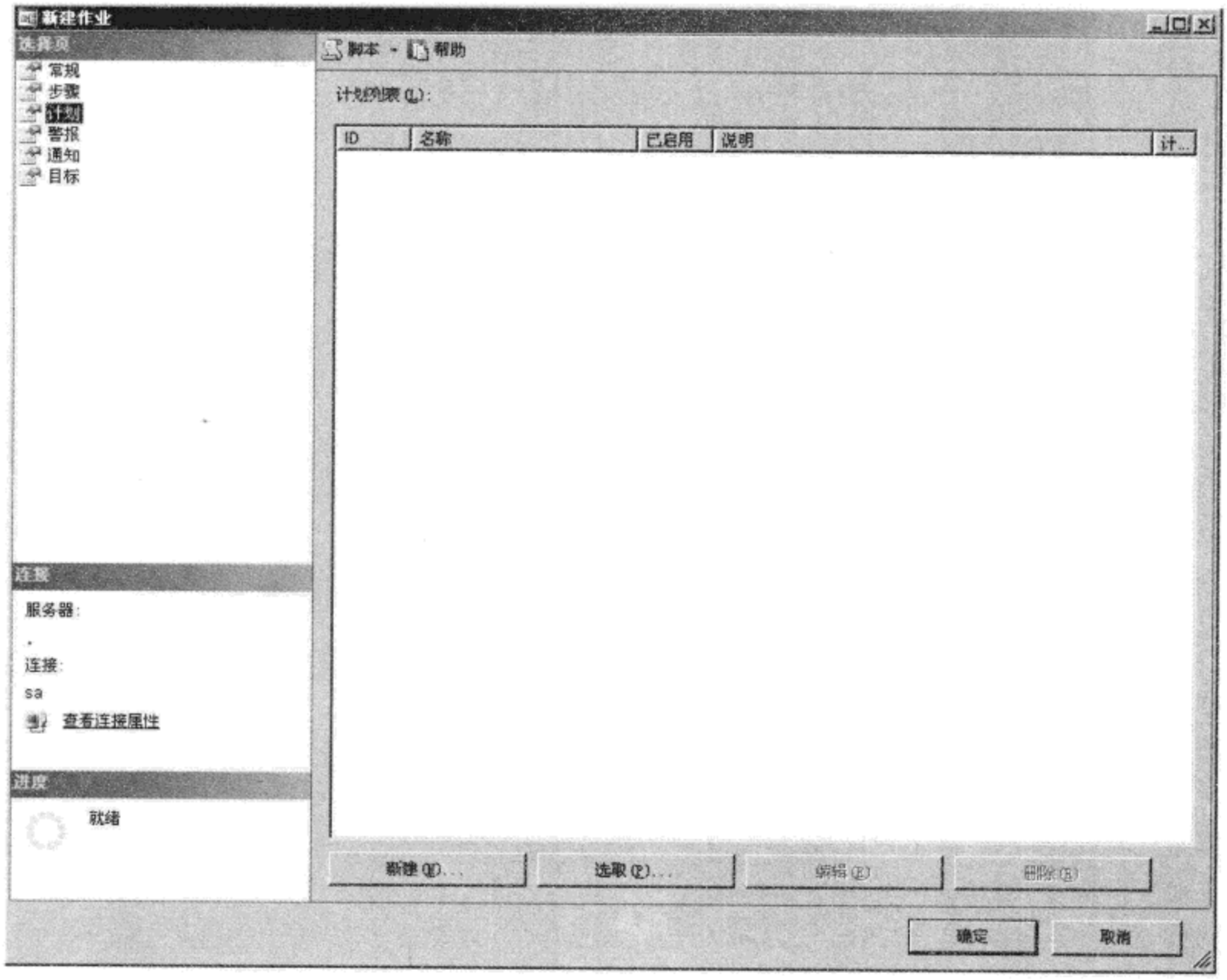


图　19-7

在这个对话框中，可以管理运行这个作业的一个或多个调度时间。为了给将要运行的这个作业创建新的调度时间，需要单击“新建”按钮。出现另一个对话框，如图 19-8 所示。

这里基本上已经填写了该对话框(免得您不知所措)，不过，这个对话框可以为作业创建新的调度。这里设置了重复性和频率。

这里的频率概念因为其奇怪的表达方式而让人有些困惑。如果想要在一天内多次运行某个作业，那么需要将作业设置为“每天执行一次”。这看上去只会每天运行一次，但是仍然有设置选项来选择是运行一次还是按一定时间间隔运行。在本例中，将作业设置为每 5 分钟运行一次。

下面将介绍作业属性中的下一个节点：警报，如图 19-9 所示。

这里，可以根据所发生的操作确定选择何种警报。单击“添加”按钮，出现另一个对话框，如图 19-10 所示。

www.TopSage.com

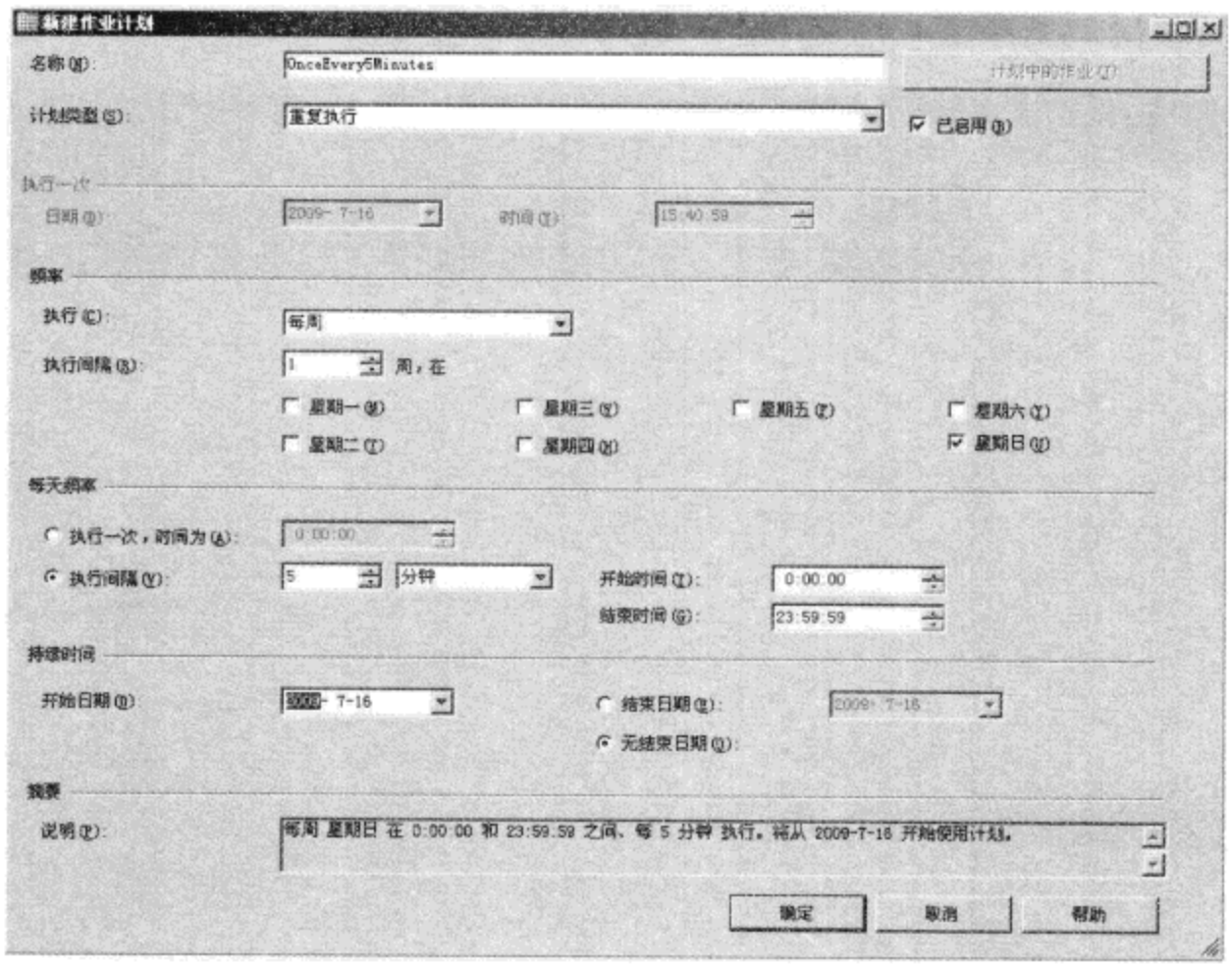


图 19-8

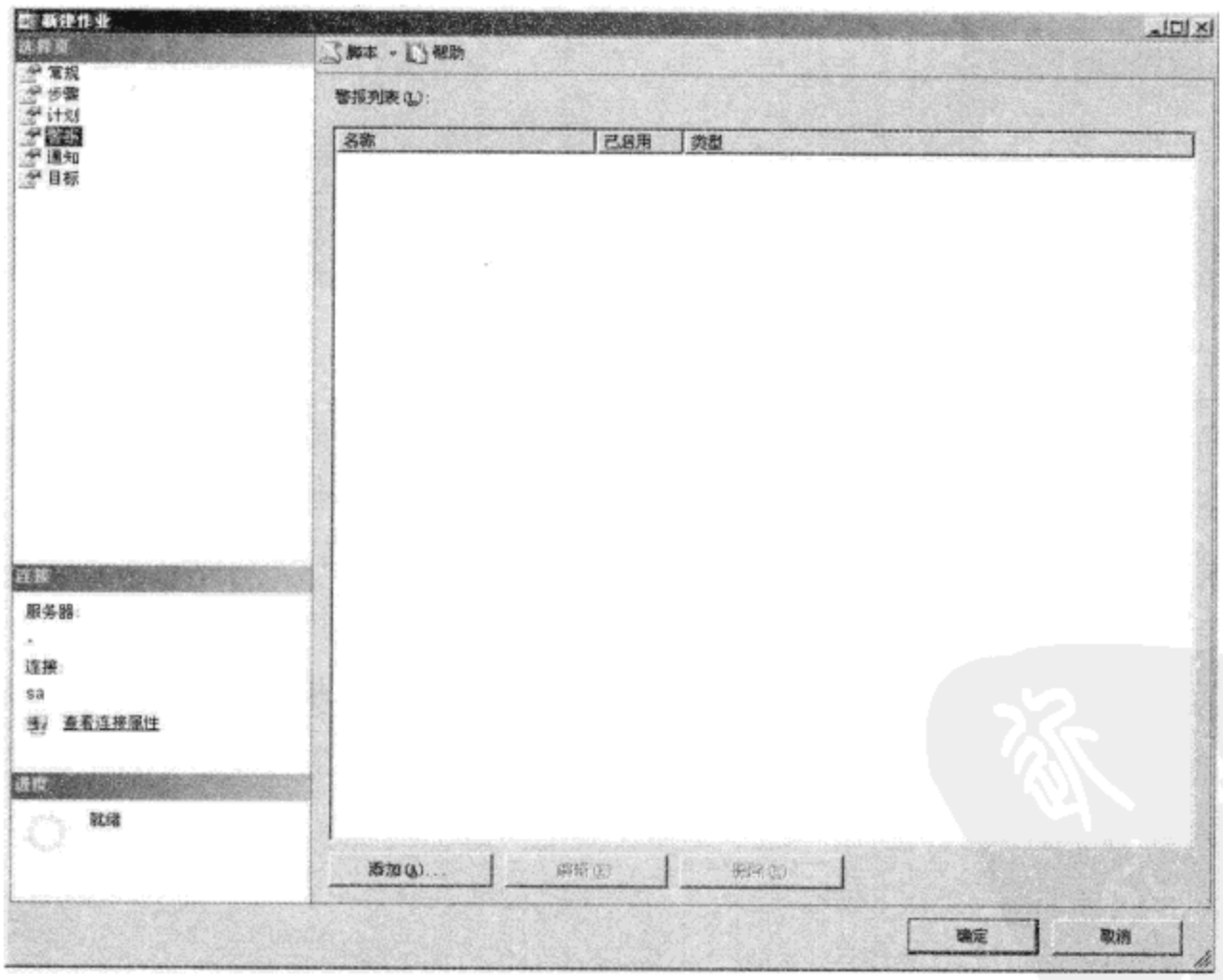


图 19-9

其中第一个节点——“常规”——要求填写一些基本的信息。例如，可以将通知信息限定为某个特定数据库。也可以定义触发这个警报之前所应满足条件的严重性级别(根据错误的严重

www.TopSage.com

性等级)。

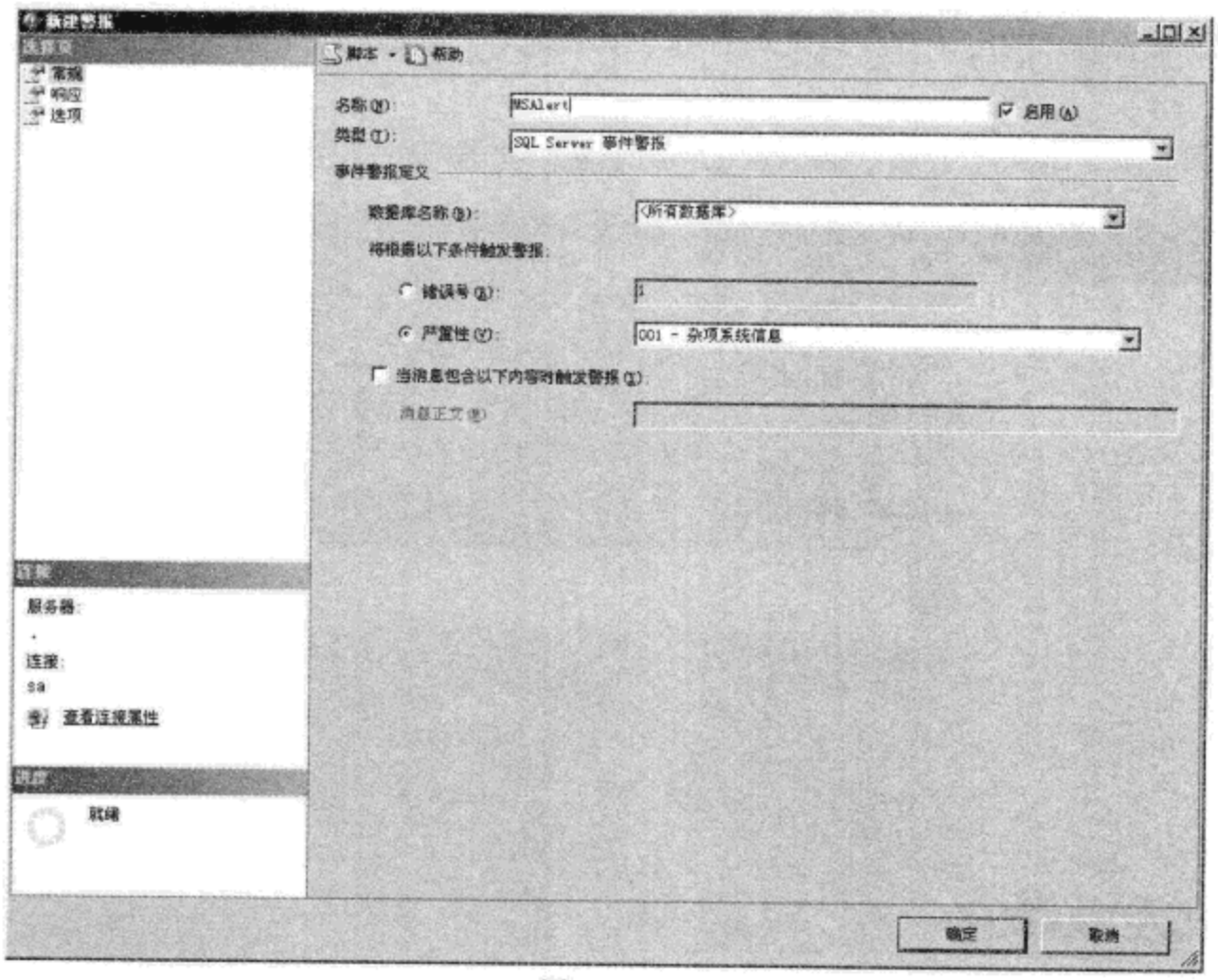


图　19-10

下面将介绍“响应”节点，如图 19-11 所示。

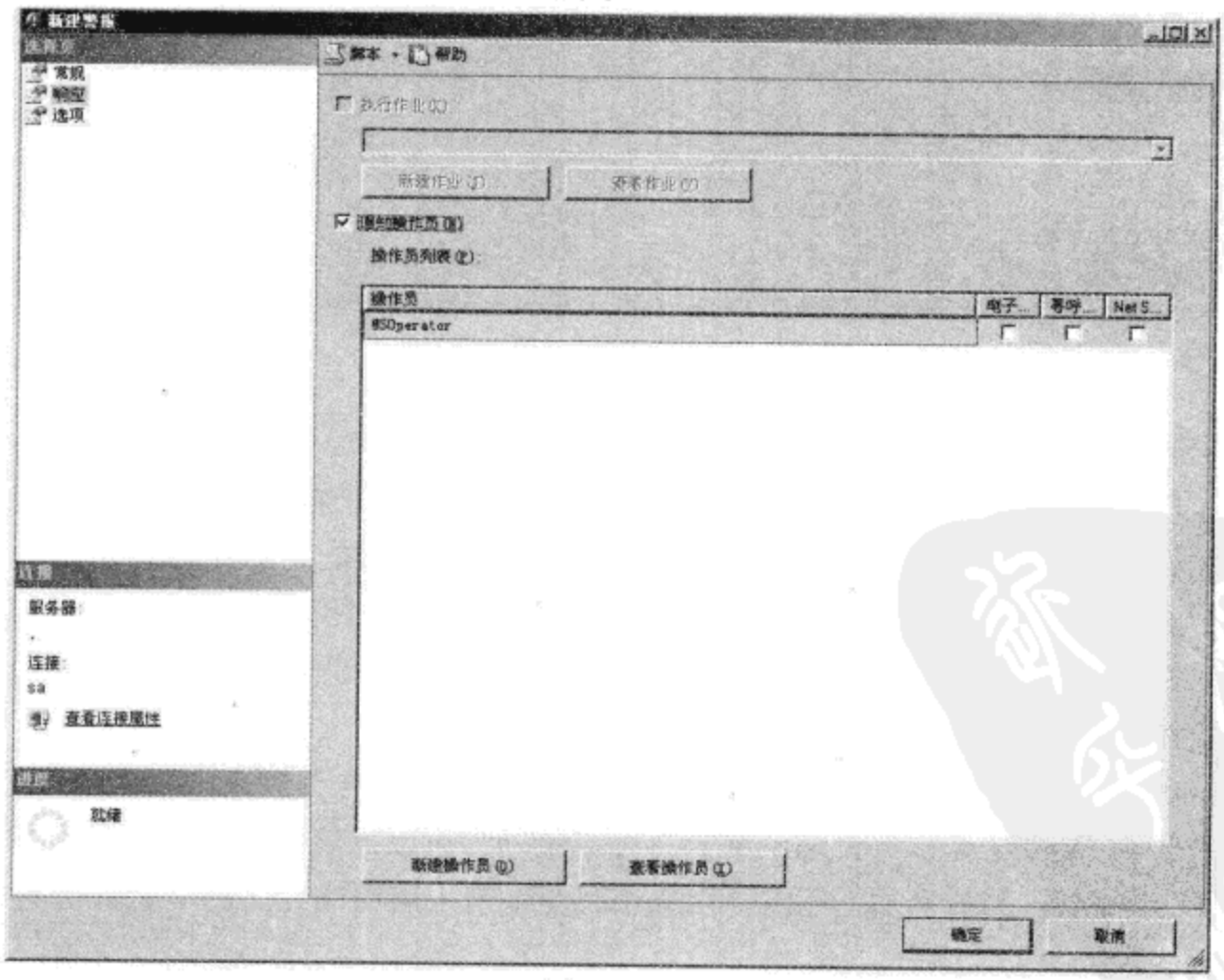


图　19-11

www.TopSage.com

注意，这里可以选择本章前面所创建的操作员。通过对这些操作员的定义，SQL Server 代理就能知道需要通知的电子邮件地址或者 net send 地址。还需注意，在右边还可控制如何通知操作员。

最后将介绍“选项”节点，该节点也十分重要，如图 19-12 所示。

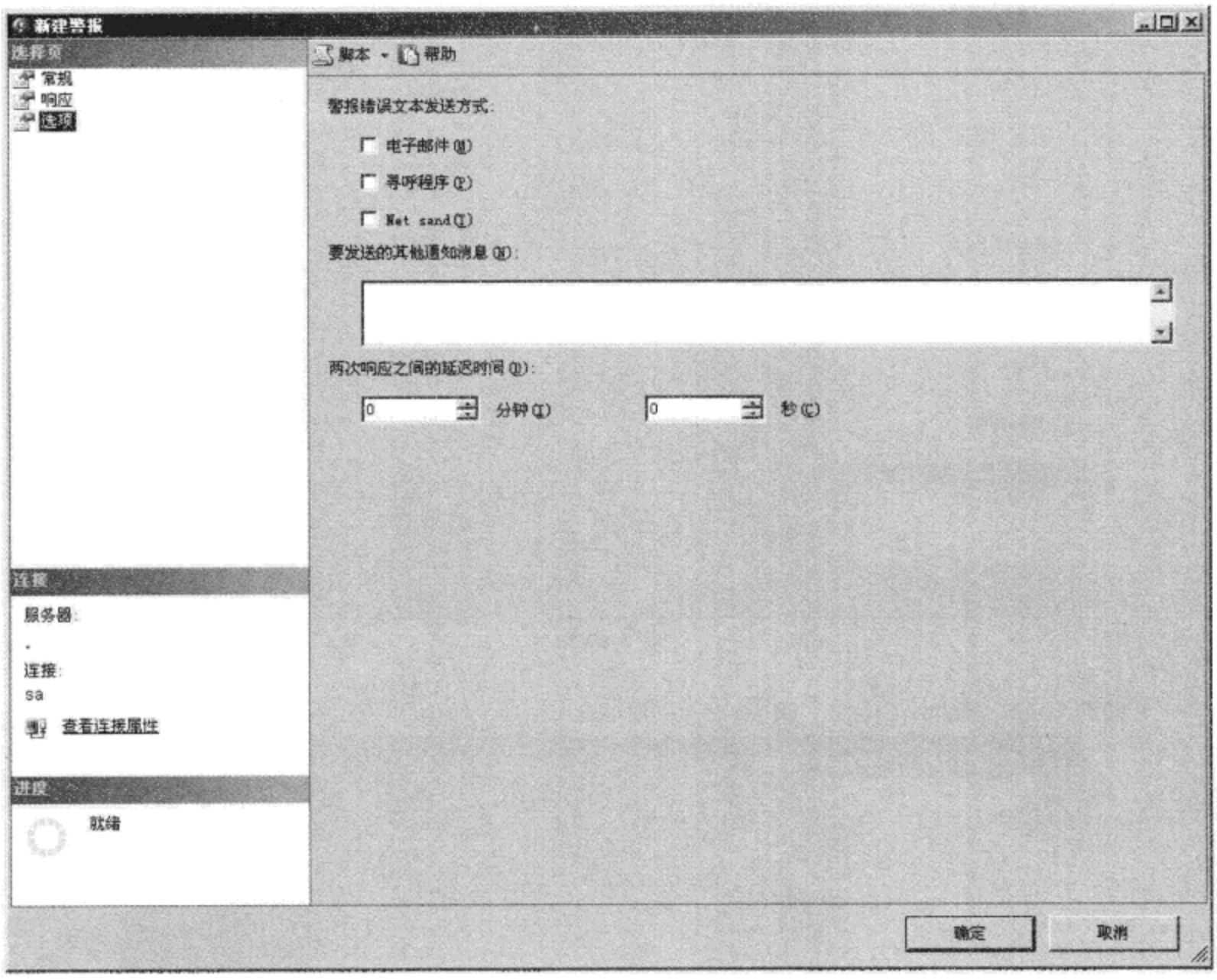


图 19-12

最后，回到“新建作业”主对话框的“通知”节点，如图 19-13 所示。

这个窗口可用于绕过原有的警报模型并对这个作业定义一个特定的响应；不过这里仍然采用原来的选择，但是也可以在这个对话框中定义额外的特定通知。

列表中的最后一个节点为“目标”节点。该节点通常属于管理员和高级开发人员领域，因为它基于的概念是一个 SQL Server 在其他 SQL Server 上调度作业。注意这一节点常用于特殊需要，该主题将在高级篇中进行介绍。

现在，可以单击“确定”按钮并退出该对话框。需要等待几分钟来触发这个任务，但是可以每隔 5 分钟在 Windows 事件日志中看到日志记录项。可以导航到“开始”|“程序”|“管理工具”|“事件查看器”进行查看。还需要切换该视图使用应用程序日志而不是默认的系统日志。

www.TopSage.com

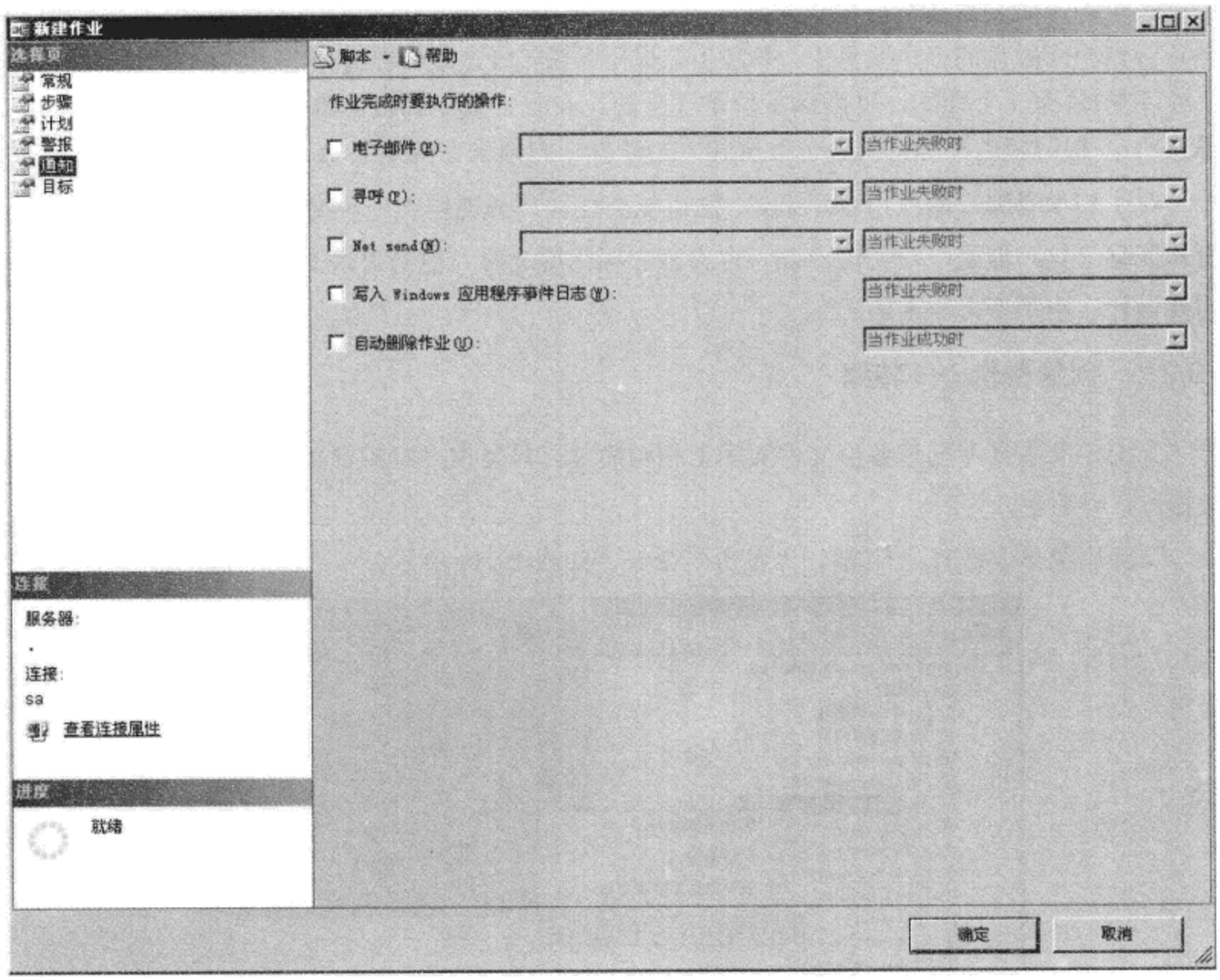


图　19-13

注意：

记住，如果要运行这样的调度任务，需要运行 SQL Server Agent。可以通过运行 SQL Server Configuration Manager 并选择 SQL Server Agent 服务或者导航到 Management Studio 的“对象资源管理器”中的“SQL Server 代理”节点来检查 SQL Server 代理的状态。

同样地，不要忘了禁用该作业(在看到该作业已按照预期的方式运行后，需要在 Management Studio 中右击这个作业来禁止运行。否则，该作业将继续在应用程序日志中创建记录项；最后，应用程序日志会被填满，系统将出现问题)。

19.2　备份和恢复

如果没有处理备份和恢复的机制，则不应部署数据库驱动的应用程序或者把应用程序卖给客户。本书再次重申：没有可靠备份的数据库操作多得令人吃惊。

www.TopSage.com

关于备份需要需要遵循的一个简单规则就是：尽早并且经常备份。另一条规则是，不要只是备份到相同磁盘的一个文件中，并忘了有这个备份；应该确保在完全分离的位置(最好是远离工作现场)还有一个副本，以确保备份是安全的。我曾亲眼见到服务器着火(焦味熏天，员工大惊失色)，相信您也不想看到备份数据和原始数据一同消失在浓烟之中。

对于初学者编写的应用程序来说，您可能会让客户或现场管理员使用 SQL Server 自带的备份和恢复工具，但是，即使这样做，也应该在深化用途时为他们提供支持。另外，不理解客户想要做什么也是不可原谅的。

19.2.1 创建备份——转储

为给定的数据库创建备份文件实际上相当简单。只要在“对象资源管理器”中导航到特定数据库，并右击。

从弹出菜单中选择“任务”|“备份”命令，如图 19-14 所示。

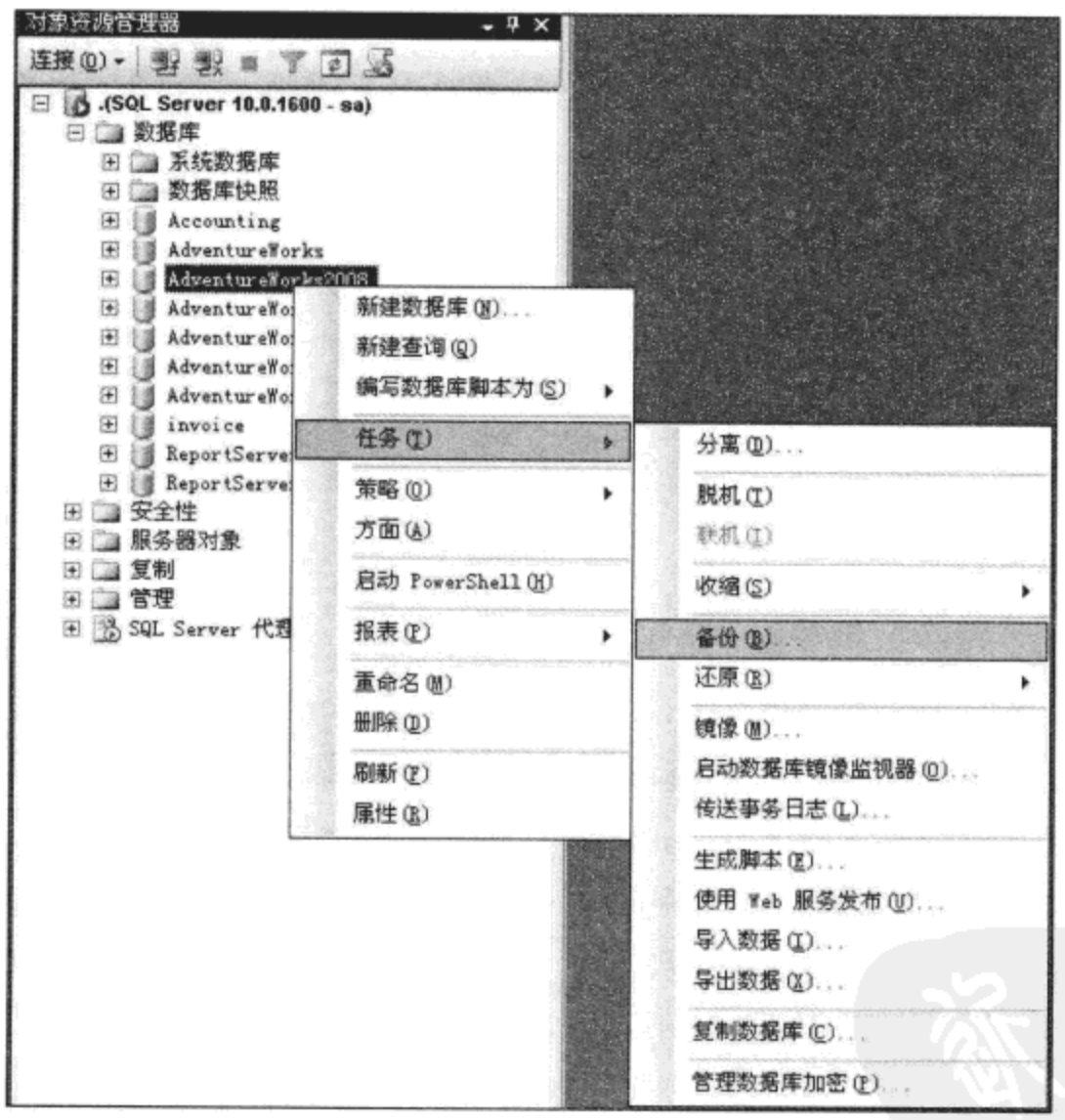


图 19-14

将出现一个对话框，该对话框可定义备份过程中的几乎所有内容，如图 19-15 所示。

www.TopSage.com

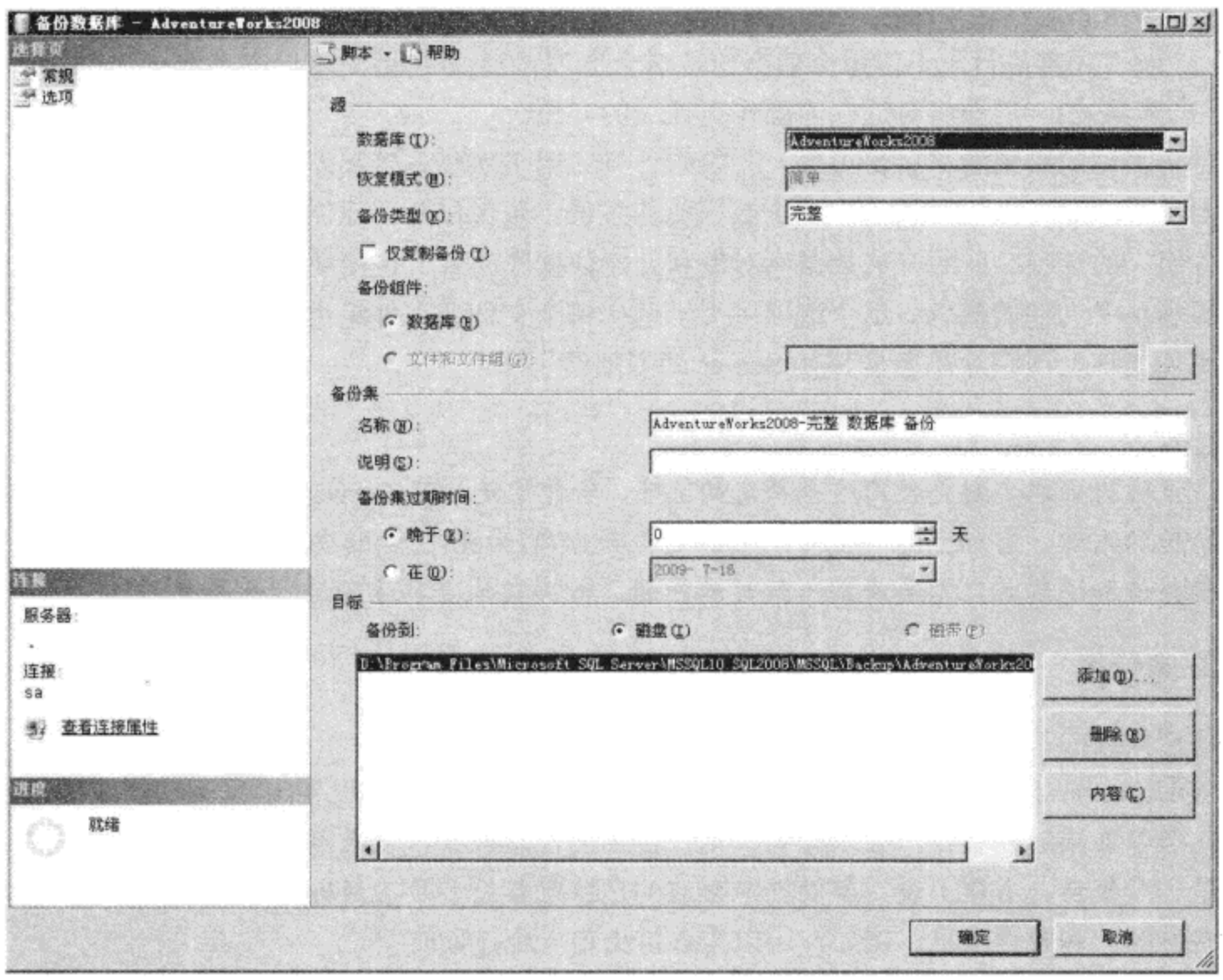


图　19-15

这里的第一个设置很容易理解——指定了想要备份的数据库。但接下来会有一点麻烦。

首先介绍的是恢复模型。这里的“恢复模型”字段只是用来说明为要备份的数据库选择了什么恢复模型；该字段实际上是一个数据库级的设置。这里先不介绍这个字段，而在 19.2.2 节介绍备份事务日志时会介绍它。

以上都是简单的部分，现在将分别介绍剩下来的一些可用选项。

1. 备份类型

首先要选择备份的类型。根据数据库的恢复模型(将在后面的内容中介绍到)，会有以下两种或 3 种可用的备份类型。

- **完整**：顾名思义，这就是对在发出备份命令前，诸如提交的最后一个事务之类的实际数据库文件的完整备份。
- **差异**：这可以称为“备份后”的备份。当采用差异备份时，该备份写入自从上次完整备份后改变的区段的副本(参见第 9 章)。这通常比完整备份速度更快，而且会占用更少的空间。那会少多少的空间呢？这要取决于实际变化数据的大小。对于那些需要花很长时间进行备份的大型数据库来说，比较常见的策略是一周或者一个月进行一次完整备份，而在此期间则采取差异备份，从而节省空间和时间。

www.TopSage.com

- **事务日志**：这个选项也很容易理解，就是事务日志的副本。如果数据库设置为“完整”或“大容量日志”，那么就会出现这个选项(如果只使用简单日志，那么这个选项则是隐藏的)。后面将对这些内容作更详细的介绍。

备份类型的副主题是备份组件，备份组件只应用于完整备份和差异备份。

对于本书而言，这里只关注整个数据库的备份。虽说如此，还需要注意另一个名为“文件和文件组”的选项。前面有关数据库对象和创建数据库的章节也简要地介绍了用于存储数据的文件组和单个文件的概念。这个选项可用于选择单个文件或文件组进行备份；这里强烈建议尽量避免使用该选项，除非您是 SQL Server 的专家级用户。

注意：

这里再次强调要避免使用这个特定的选项，除非您是 SQL Server 备份方面的专家。该选项有着特殊的用途，有助于那些高可用性的大型数据库(千兆字节)的安装。当要对这种类型的备份进行备份和恢复时，需要考虑一致性的问题，能力欠缺者不适合进行此类操作。

2. 备份集

备份集指的是一个用于表示一个或多个备份目标的名称。

SQL Server 允许出现备份文件特别大的情况，也允许跨多个设备(磁盘驱动器或磁带)进行备份。但是如果这样操作，要求能够从用作目标的任意设备恢复数据——也就是说，这些备份设备是一个集合。本质上说，备份集中存放的是特定备份涉及的目标的定义。此外，备份集还包括备份的一些属性信息。例如，可以为备份指定一个过期时间。

3. 目标

这一选项指定了数据的备份位置。在该选项中为一个备份集定义需要使用的一个或多个目标。对于大多数安装程序来说，这是一个文件位置(稍后会将转移到磁带上)，但是也可以定义备份设备，这样可以直接备份到磁带或者类似的备份设备。

4. 选项

除了刚才介绍的对话框中“常规”节点的那些项以外，还有一个节点可以用来设置其他很多的选项。其中的多数选项非常容易理解。但是，要特别注意转换日志区域。

5. 计划

当所有这些选项都设置完后，最好能再设置一个作业定期运行这个备份。这个对话框顶部的“脚本”按钮就用于此目的。单击“脚本”按钮，会得到一个选项列表，选择“将操作脚本保存到作业”，进入在本章之前见到的“作业计划”对话框，只是其中的作业步骤已经完成。可以选择“计划”选项卡并按需要进行填写，定义一个常规计划来运行刚才定义的备份。

说明：

还有一些只能通过发出 T-SQL 命令或者使用.NET 可编程的对象模型 SMO 才能使用的备份相关项。这些都是高级应用，更为详细的介绍请参见 Profession SQL Server 2008 Programming。

www.TopSage.com

19.2.2　恢复模式

什么是恢复模式？

回到介绍事务的章节(参见第 14 章)，其中介绍了事务日志。除了要跟踪事务以处理事务回滚和保证数据原子性以外，事务日志对于将数据恢复至系统故障点也是至关重要的。

假设您正在运营一家银行。已经在 6 个小时内处理了一些存款和取款业务——6 个小时之前刚进行了最后一次完整备份。这时，如果系统出现宕机，则估计您一定不愿意使用昨天晚上的备份，也不愿意失去这 6 小时内所有进出钱款的数据。也这就是说，您需要实时的数据值。

保存事务日志能够自最后一次完整备份或者差异备份完成之后的任何事务进行“前滚”。假设数据备份和事务日志都可用，那么就可以恢复到故障点时的状态。

恢复模式决定了日志记录的保存时间和类型；共有以下 3 种选项。

- **完整**：顾名思义，这个选项将记录所有内容。使用这个模式，假如已经作了数据备份并且拥有备份之后的所有事务日志，那么即使系统故障，也不会丢失任何数据。如果丢失或损坏了一个日志，也可以通过最后记录的完整日志恢复数据。但是请记住，这会占用相当数量的系统空间来接收很多更改或新的数据。
- **大容量日志**：这一选项仅次于完整恢复。这个选项会记录常规的日志，就像使用完整恢复一样，但是对于批量操作不是这样。万一碰到系统故障，还原的备份会包含任何未参与批量操作(例如批量数据的导入或索引创建)的数据页的修改，但是所有批量操作必须重做。这个选项的好处在于批量操作会执行得更好。但是性能伴随着风险，所以要有所取舍。
- **简单**：使用这种模式，事务日志实际上仅仅是为了支持事务的发生而存在的。通常会截断事务日志，任何已经完成的或者是回滚的事务都会从日志中删除(过程可能不会这么简单，但结果却是如此)。这就提供一个更小的和性能更好的紧凑的日志。但是对于系统故障的恢复而言，这种日志没有作用。

对于大多数安装程序来说，都会为生产级数据库选择完整恢复模式。

19.2.3　还原

这是和备份相反的操作。当认真完成备份之后，现在出于恢复或者是仅仅对数据库进行复制的目的，想要还原备份。

一旦有了对数据库的备份，将它还原到原来的位置就很简单。还原的过程和备份的过程很相似：导航到想要还原的数据库并右击，选择“任务”|“还原”，出现“还原数据库”对话框，如图 19-16 所示。

只要要将原备份还原至曾经对其作备份的数据库的顶层，就还是比较简单直接的。单击“确定”按钮，就会进行还原而不会出现任何问题。

1. 还原到不同的位置

如果想要修改还原的位置，那么会有点麻烦。作为备份过程的一部分，备份知道所备份的数据库的名称，而且更重要的是，备份知道要使用的物理文件的路径。

www.TopSage.com

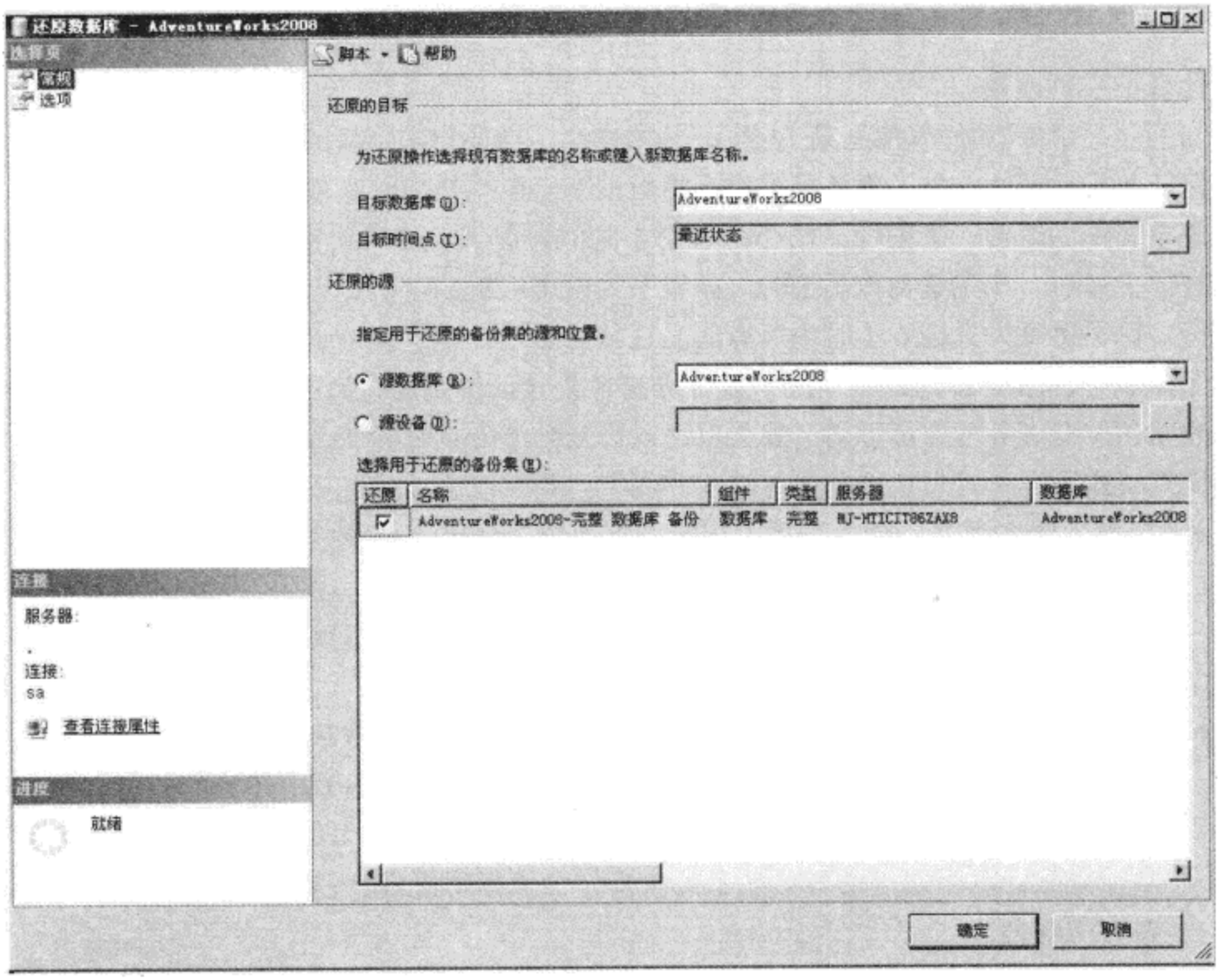


图 19-16

修改目的数据库的名称并不重要，因为修改目的数据库的名称不会改变用于存放数据的物理文件(MDF 和 LDF 文件)。为了处理这个问题，需要选择“还原数据库”对话框的“选项”节点。

再次重申，这里的选项都比较容易理解，但是，请特别注意一下“还原”列。在对话框中，只要有办法将数据库的多个副本还原到相同服务器(可能只是出于测试目的)，或是有办法将数据库安装到新卷或是新系统中，就可以替换每一个原始文件的目标位置和名称。

2. 还原状态

还原状态指的是当完成还原时想要数据库处于的状态。如果您正在还原数据库而且之后仍然有日志应用于该数据库，那么还原状态会表现出特殊的相关性。

如果使用默认的选项(如果使用 T-SQL，就等价于使用 WITH RECOVERY 选项)，那么当还原操作完成时，数据库会立即处于完全的在线状态。如果想要在最初的还原完成之后还原日志，那么需要选择其他两个选项中的一个。这两个选项都用来防止数据库更新，并且使其处于可以完成更多还原的状态；二者的区别仅在于是否允许用户使用只读模式访问数据库，或者是否应将数据库显示为离线状态。

注意：

可用性问题的庞大性可能超乎您的想象。当还原操作骤然将数据库标记为可用时，您会很

www.TopSage.com

惊奇地看到用户如此之快地就找到了进入系统的方法。即使知道在完成现有还原后会发生的事情，您可能还是想在实际用户访问数据库之前有检查数据库的机会。如果是这样，请务必使用 NO RECOVERY 还原方法。在确认进行过检查工作后，可以运行 WITH RECORERY 选项的还原，使数据库完全回到在线的状态。

19.3　索引维护

第 9 章讨论了索引如何产生碎片的问题。这成为影响数据库性能的一个很大的障碍，这一问题需要使用策略来解决。幸运的是，SQL Server 中有命令能够识别数据和索引，并进行清除。同时使用刚学的作业调度，这样就可以自动进行日常碎片整理。

注意：

您可能还会碰到使用旧的 DBCC DBREINDEX 或 DBCC INDEXDEFRAG 命令的维护脚本。但二者的功能都已经被 ALTER INDEX 命令所替代。为了向后兼容，仍支持原有的 DBCC 命令，但 Microsoft 随时可能删除该命令，因此当碰到这样的脚本时，应主动对其进行升级。

ALTER INDEX 是数据库维护的得力工具。第 9 章曾提到这点，现在再次回顾一下，看看如何对它进行调度。

ALTER INDEX

ALTER INDEX 命令功能方面有一些欺骗性。虽然大多数 ALTER 命令是用来改变对象的定义的——例如通过 ALTER 表添加或禁用约束和列——但 ALTER INDEX 则不同：该命令是关于维护的操作，而和结构无关。如果需要对索引的结构进行修改，仍然需要使用 DROP 或 CREATE 命令，或者需要 CREATE 并使用 DROP_EXISTING=ON 选项。

ALTER INDEX 语法的缩略版本如下所示：

```
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 ] )
   }
[ ; ]
```

其中相当一部分内容是高级 DBA 才用到的——通常都用来处理非常特殊的问题，但是这里有一些核心的元素，这些核心元素应该成为日常维护计划的一部分。这里将首先介绍顶部的

www.TopSage.com

几个参数，并介绍大型维护计划需要的一些选项。

1. 索引名

如果想要维护一个特定的索引，那么可以对这个特定索引命名，或者使用 ALL 来表示想要对与这个命名的表相关的每个索引进行维护。

2. 表名或视图名

顾名思义，这是想要对其进行维护的特定对象(表或视图)的名称。注意，必须是一个特定的表(可以提供一个列表，指明所有这些项都执行维护)。

3. REBUILD

这是修复索引的一种强大方法。如果用这个选项运行 ALTER INDEX，那么会完全删除以前的索引并从头建立新的索引。其结果是真正优化的索引，其中所有叶级或非叶级的页都已经按照定义重新构建过，要么是默认的，要么使用开关修改填充因数等。

注意：

要小心使用这个选项。默认情况下，一旦开始 REBUILD，正在使用的索引就不存在了，直到整个重建过程完成。任何依赖于这个索引的查询可能会变得非常慢(可能降几个数量级)。对于这类操作，需要先在离线的系统上测试其运行需要多少时间，然后进行调度，使其在非高峰时间运行，最好有人能够监控它，以保证在高峰时间到来时系统能回到在线状态。

SQL Server 的企业版中包含了这一选项的特殊 ONLINE 版，该选项可使旧索引处于活动状态并构建一个与之平行的新索引，但要知道非企业版本未提供这一选项，即使有，索引的创建对系统而言也会是巨大的负担；在重建过程中要步步小心。通常，应将这一选项视为运行时会产生巨大副作用的选项，并且绝对要在数据库管理员的掌控下进行所有操作。

4. DISABLE

这个选项的作用很明显，只是方式有些激烈。如果命令所做的只是让索引脱机，直到您决定想要进行的操作，那么该选项是不错的选择，但是该选项会将索引标记为不可使用。一旦索引不可用，在使索引再次有效之前，必须重建索引(不是重组，而是重建)。

一般几乎不会用到这个选项(更常见的做法是删除索引)。可能要在 SQL Server 升级或者出现其他一些古怪的情况时才使用这个选项。

注意：

请小心使用这个选项！如果对表禁用了群集索引，那么表也会被禁用。数据仍然存在，但是所有的索引都不能访问这些数据，因为表和数据都依赖于群集索引——直到重建群集索引之前，整个表都为离线状态。

5. REORGANIZE

从开发人员的角度来说，REORGANIZE 选项很不错。如果重新组织索引，就得到了仅次

www.TopSage.com

于完全重建索引的完全优化，但是可以在线进行(用户可以继续使用索引)。

如果注意一下，可能会发现一个问题："仅次于是什么意思？" REORGANIZE 选项只在索引的叶级上起作用；而不会对索引的非叶级起作用。这意味着不会得到完全的优化，但是对于大部分索引来说，非叶级并不是真正产生碎片的地方，虽然可能会产生碎片，但具体情况也不尽相同。

考虑到对用户的影响小得多，REORGANIZE 选项通常是作为日常维护的工具；下面介绍如何运行索引重组的命令。

试一试　索引重组

为了作演示，对 AdventureWorks2008 数据库中的一个表进行索引重组。Production.TransactionHistory 表就是个不错的示例，因为可能会不断地向这个表中插入很多行，当事务的存在时间太长需要删除时，就清除这些行。在这种情况下，可使用一个简单的命令重组表中的所有索引：

```
ALTER INDEX ALL
ON Production.TransactionHistory
REORGANIZE;
```

不能从数据库取回任何数据——只是得到命令成功完成的消息：命令已成功完成。

示例说明

ALTER INDEX 命令根据提供的 ALL 选项(而不是任何特定的索引名)，查找 Production.TransactionHistory 表中可用的索引(忽略被禁用的索引，因为重组不会对被禁用的索引进行任何操作)。然后该命令在后台枚举每个索引并在其上执行重组，而且只对每个索引的叶级进行重组(包括对实际数据进行重组，因为这个表的群集索引也会被重组)。

19.4　数据归档

这是个比较难处理的问题。对数据进行归档的方法有很多种。例如，如果建立一个 OLAP 数据库来使用 Analysis Services，那么通常需要去解决长期报表归档的问题。无论对长期需要的数据可用性有多么确定，总有一天会出现需要处理数据太大而系统不能良好运行的情况。

如上所述，归档的方法有很多种，因为每个数据库之间都有点区别。重点在于需要在创建数据库时就考虑归档的需要。要认识到当开始删除记录时，可能会违反参照完整性约束和/或导致孤立记录；可以设计一个逻辑路径，用来在归档时删除或移动记录。在编写归档脚本时，需要注意以下事项。

- 如果在 OLAP 数据库中已经有了数据，那么可能不需要担心把数据保存到任何其他的位置。与您的老板或律师谈谈看是不是这样。
- 数据使用的频率如何？值得去保存数据吗？人类总是不愿放弃任何事物：当然也包括数据。如果只担心合法性要求，那么就考虑把从来不会用到或很少用到的数据保存到

www.TopSage.com

磁带中(作者建议对归档数据使用多个备份)并且减少在线的数据量；当用户发现性能提高时，他们会很赞成您的做法。

- 不要忘了孤立的数据。当开始删除数据时，参照完整性约束会防止产生很多的孤立数据，但是也可能会有参照完整性应用不到的地方。这种情况会导致严重的系统错误。
- 要认识到归档程序可能会运行很长的时间。运行时间的长短和影响到的行数可能会导致联机用户访问数据的并发问题；最好在不使用系统时运行归档程序。
- 不断地测试！

19.5 基于策略的管理

这是 SQL Server 2008 中相对高级的新功能。其思想十分简单：现代关系系统都是对数据采取主动管理——为什么不主动管理应用于服务器的对象和规则呢？

使用基于策略的管理，可以建立一套规则来管理服务器甚至是域中所有的服务器。策略可针对很多内容实施，例如：

- 对象名：如果想要所有存储过程都以“sp”开头，可以使用基于策略的管理建立这样一条规则。
- 所有数据库都应默认将 ANSI ARITHABORT 选项默认为真。

究竟如何执行是由规则(即执行模式选项)设置的，包括：

- **按需**——只当管理员专门申请对策略进行审核时检查有无违反策略的情况。
- **按计划**——根据调度审核违反策略(将违反结果列成一张表，但不要作任何改动)。
- **更改时：禁止**——这一选项主动拒绝策略违反情况。如前面的示例，任何没有以 sp 开头的存储过程都将在创建过程中失效。
- **更改时：仅记录**——这一选项记下违反情况，但也只是记录而已(便于生成报表)。

大多数安装并不需要基于策略的管理，但是对于大型的多服务器环境而言，基于策略的管理将发挥巨大作用。

19.6 本章小结

本章介绍了一些值得思考的内容。作为开发人员，很容易就会把许多管理任务看作“别人的问题”而视而不见。请不要有这样的想法。

注意：

作者几年前所了解到的一个项目就是一个很好的示例，可以说明问题。某开发公司为美国西北部一个非赢利的组织开发了一个不错的系统。但在运行达 8 个月之后，该公司接到了用户的紧急求助。讨论过后，他们确认数据库已经受到了一些损坏，并建议客户从备份还原数据库。客户的自然反应是：“备份？”这家开发公司忽略了一些非常重要的内容；他们明知道他们的用户没有经验，可能没有数据库管理员——如果该开发公司不做这件事，还会有谁告诉客户需要备份并帮助创建备份呢？该开发公司从这件事中吸取了教训，希望您也如此。

www.TopSage.com

在设计中，尤其在制作开发计划时，请考虑管理问题。如果能简化系统的管理，就会发现这个系统会更加成功；通常开发人员(也就是您自己)也能从中受益。

19.7 练习

1. 调度本章中用来重组 Production.TransactionHistory 表的命令，使之每周日凌晨 2 点运行。
2. 对 AdventureWorks2008 数据库进行完整备份。将备份保存到 C:\MyBackup.bak(如果在该卷上没有足够的磁盘空间，请稍微更改)。
3. 使用一个新的数据库名称 NewAdventures 来对练习 2 中创建的备份进行还原。

www.TopSage.com

附 A 录

系 统 函 数

SQL Server 包含了许多系统函数和大量典型函数。其中的一些函数经常被使用，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。

本附录将简要介绍这些函数的用法。

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 函数分为以下几种。

- 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
- 行集函数
- 安全函数
- 字符串函数
- 系统函数
- 文本和图像函数

A.1 遗留的系统函数(即全局变量)

A.1.1 @@CONNECTIONS

该系统函数返回自上次启动 SQL Server 以来尝试的连接数。

www.TopSage.com

这是自最近一次 SQL Server 启动以来尝试连接的次数总和。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 甚至不知道增加计数器的值。SQL Server 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尝试连接，而不是尝试登录。根据应用程序的情况，可能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：

和许多其他系统函数一样，系统存储过程 sp_monitor 通常可以为该系统函数较好地提供服务。通过一个命令，该存储过程提供了包括连接数、CPU 繁忙程度以及 SQL Server 的写入次数等信息。如果要获取的是基本信息，那么 sp_monitor 存储过程可能更合适——如果需要的是可操纵的离散数据，则最好使用@@CONNECTIONS 系统函数。

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的分辨率，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，如果想要知道该数据，就更应该使用 Performance Monitor。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(基于系统计时器的分辨率)，该时间以毫秒计数。

可以把@@IDLE 看作@@CPU_BUSY 的相反情况。从本质上讲，该系统函数说明了 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(基于系统计时器的分辨率)，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

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的数目。

二者主要用作网络故障检修的工具。

www.TopSage.com

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错误数。

它主要用作网络故障检修的工具。

A.1.9 @@TIMETICKS

返回操作系统时钟节拍的微秒数。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。

不要将该系统函数与运行时错误或@@ERROR 相混淆。这是关于物理 I/O 的问题。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 Performance Monitor 来取代它。

A.1.11 @@TOTAL_READ 和@@TOTAL_WRITE

分别返回自 SQL Server 上次启动后磁盘读取/写入的总量。

这两个名称容易误解，因为二者不包含对缓存的任何读取；它们只是物理 I/O。

A.1.12 @@TRANCOUNT

返回有效的事务数——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。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，如果没有开启事务，则只开始一个事务的处理。)

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。

www.TopSage.com

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：

- 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，即使某个值多次出现。DISTINCT 参数意味着一个值只能在函数中包含一次，即使该值多次出现在函数中。

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。

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。这个特殊的函数视表达式或*号(这表示要包括所有连接表中的所有列)为参数。其基本语法如下：

```
CHECKSUM(<expression>, […n] | * )
```

注意，表达式的顺序——或这里的*号，即连接顺序——将影响检验和的值，例如下面这个语句：

```
CHECKSUM(SalesOrderID, OrderDate)
```

和下列语句的结果不同。

www.TopSage.com

```
CHECKSUM(OrderDate, SalesOrderID )
```

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(只是记得在必要时强制转换类型)；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。

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，不同的是该函数总是返回 bigint 数据类型的值。其语法如下：

```
COUNT_BIG
(
   [ALL | DISTINCT ] <expression> | *
)
```

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。

A.2.7 MAX

MAX 函数返回 expression 中的最大值。其语法如下：

```
MAX([ALL | DISTINCT] <expression>)
```


www.TopSage.com

MAX 忽略任何 NULL 值。

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。

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。

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。

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。

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。

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。

www.TopSage.com

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，对应星期天。值的对应关系如下。

- 1——星期一(对于世界上大多数国家来说是第一天)
- 2——星期二
- 3——星期三
- 4——星期四
- 5——星期五
- 6——星期六
- 7——星期天

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)。这里需要注意以下的内容。

- 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。
- 数据库中任何时间戳的修改都会反映出来，不只是当前连接的时间戳

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

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，可以使用系统的存储过程 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被清除。这种情况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(这意味着无限期的等待)，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_

www.TopSage.com

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。

不要误认为该函数和 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 的版本以及应用程序和硬件的限制。

A.3.6 @@MAX_PRECISION

返回当前为小数和数字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。可以通过在从命令行启动 SQL Server 添加/p，或者在 Windows 2000、2003、XP 或 2008 的 Services applet 中，将/P 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@@NESTLEVEL 是 0。如果这个存储过程调用另一个存储过程，那么就说第二个存储过程嵌套在第一个存储过程里面(@@NESTLEVEL 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，而且您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 使用二进制标记来表示设置了哪个值。为了测试是否设置了您所感兴趣的选项，必须把该选项值与按位运算符一起使用。例如：

```
IF (@@OPTIONS & 2)
```

如果该值为真，那么就知道已经为当前连接打开了 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	临时与延迟约束检验
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

www.TopSage.com

(续表)

位	SET 选 项	描　　述
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以零错误时终止查询
128	ARITHIGNORE	当在查询时发生溢出或除以零错误时返回 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消息
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，就定义为允许使用 NULL 值。它与 ANSI_NULL_DELT_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 NULL 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存储过程的远程服务器(通常指地理位置)而有不同表现时，该函数很有用。

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>

www.TopSage.com

A.3.11 @@SERVICENAME

返回 SQL Server 正在其下运行的注册表键的名称。如果当前实例为 SQL Server 默认实例，则返回 MSSQLSERVER；或者就是具体应用的实例名。

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，可以识别当前连接的 SPID，DBA 可以使用该函数来监控，并在必要时终止该进程。

A.3.13 @@TEXTSIZE

返回 SET 语句的 TEXTSIZE 选项的当前值，该返回值指定 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 语句修改这个值。

A.3.14 @@VERSION

返回 SQL Server 当前安装的版本、处理器类型和 OS 体系结构。

例如：

```
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 行受影响)
```

遗憾的是，这不会以任何结构化的字段排列形式返回信息，所以如果想要使用该函数测试特定信息，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该函数，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，不少函数功能是重复的，但二者的区别在于一个函数支持对称密钥，而另一个函数(通常名称中带有“Asym”)支持非对称密钥。

您可能会问：“为什么需要这些函数呢？”答案是多样化的。简单地说，如果想在传输过

www.TopSage.com

程中保护发送或接收的数据，需要使用这些函数。例如，由于 SQL Server 支持 HTTP 端点，并在那里驻留自己的 Web 服务，您可能希望用 Web 服务的客户端接受或返回加密信息。更可能的是您选择在数据库中加密数据，现在为了使用该加密数据需要解密。

A.4.1 AsymKey_ID

给定非对称密钥的名称，该函数从数据库返回与相关 ID 对应的整数值。其语法很简单：

```
AsymKey_ID('<Asymmetric Key Name>')
```

要使用该函数，必须有对该密钥的访问权限。

A.4.2 Cert_ID

类似于 AsymKey_ID，该函数返回证书名称的 ID。其语法很简单：

```
Cert_ID('<Certificate Name>')
```

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，也可以以字符串形式返回)、证书的主题(证明谁或什么)。其语法如下：

```
CertProperty ( <Cert_ID> ,
   '<Expiry Date>' | '<Start Date>' | '<Issuing Authority>' | '<Certificate
  Serial Number>' | '<Subject>' | '<Security ID>' | '<SID as a String>' )
```

返回的数据类型随指定的属性而不同(可能是 datetime、nvarchar 或 varbinary)。

A.4.4 DecytptByAsmKey

从其名称中可推测出，该函数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变量)，以及用于加密非对称密钥的密码(如果在创建密钥时使用了这一密码)。其语法很简单：

```
DecryptByAsymKey(<Asymmetric Key ID>, {'<encrypted string>'|<string variable>}
[, '<password>'])
```

记住，如果在创建非对称密钥时使用了密码，那么在利用该密钥解密数据时也需要使用这一密码。

A.4.5 DecryptByCert

这个函数基本上与 DecryptByAsmKey 函数相同，除了它接受的是证书而不是非对称密钥。和 DecryptByAsmKey 函数一样，它用密钥解密数据。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变量)，以及用于进一步加密数据的密码(如果使用了密码)。其语法与 DecryptByAsmKey 函数的类似：

```
DecryptByCert(<Certificate ID>, {'<encrypted string>'|<string variable>}
```


www.TopSage.com

```
[, '<password>'])
```

同样，加密数据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)，而且要求密钥是“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，如果提供验证器增加的值(用整数形式)，那么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

顾名思义，这个函数对用密码短语而非密钥加密的数据解密。除了接受密码短语参数而不是假设打开的密钥外，该函数的工作方式与 DecryptByKey 函数类似：

```
DecryptByPassPhrase({'<passphrase>'|<string variable>},
   {'<encrypted string>'|<string variable>},
   [<add authenticator value>, '<authentication hash>'|<string variable>])
```

和 DecryptByKey 函数一样，如果提供验证器增加的值(用整数形式)，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>})
```

记住，如果向数据库中添加非对称密钥时使用了密码，那在使用该密钥解密任何数据时还需要使用这一密码。

A.4.9 EncryptByCert

这个函数与 EncryptByAsmKey 函数基本上相同。不过，它接受证书而不是非对称密钥。和 EncryptByAsmKey 函数一样，这个函数使用提供的密钥加密数据。它要求输入证书(通过 ID)、要加密的数据(字符串字面量或字符串变量)，以及用于进一步加密数据的密码(可选)。其语法与 EncyptByAsymKey 函数类似：

```
EncryptByCert(<Certificate ID>, {'<string to be encrypted>'|<string variable>}
 [, '<password>'])
```

同样，加密数据时使用的密码在解密时也是需要的。

www.TopSage.com

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，用要加密的数据(字符串字面量或字符串变量)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，如果提供验证器增加的值(用整数形式)，那么该值必须与在加密字符串时提供的值一致，另外，还必须提供一个散列值(该散列值同样在解密时也是需要的)。

A.4.11 EncryptByPassPhrase

这个函数使用通行短语而不是密钥加密数据。除了接受通行短语而不是假设打开的密钥作为参数外，EncryptByPassPhrase 函数的工作方式与 EncryptByKey 函数类似：

```
EncryptByPassPhrase({'<passphrase>'|<string variable>},
   {'<string to be encrypted>'|<string variable>},
   [<add authenticator value>, '<authentication hash>'|<string variable>])
```

和 EncryptByKey 函数一样，如果提供验证器增加的值(用整数形式)，那么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 ID。

```
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 8 000 字节)。

```
SignByCert(<Certificate ID>, <string variable> [, '<password>'])
```


www.TopSage.com

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整数值：

```
VerifySignedByAsymKey(<Asymmetric Key ID>, <plain text> , <signature>)
```

A.4.17 VerifySignedByCert

针对指定的证书和纯文本值，返回表明签名验证成功与否的整数值：

```
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的变量中。

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，但是这里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 游标阈值为比 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，就会使用@@FETCH_STATUS 游标函数。通过这个值可知道尝试导航至游标中的一个记录是否成功。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 FETCH 失败，则还会说明具体原因。这些常量包括以下几个。

- 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。这只会在可滚动的、非动态的游标中出现。

www.TopSage.com

为了增强可读性，这里在使用@@FETCH_STATUS 游标函数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决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 cursorname 形式，那么有以下 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，那么结果集有 1 或多行。
- 0——游标的结果集是空的。
- －1——游标是关闭的。
- －3——cursorname 的游标不存在。

如果使用 cursor_variable 形式，那么会有以下 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，那么结果集有 1 或多行。
- 0——结果集为空。
- －1——游标是关闭的。
- －2——没有分配给 cursorvariable 的游标。
- －3——名称为 cursorvariable 的变量不存在，或者即使变量存在，但没有给变量分配游标。

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

www.TopSage.com

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(天、周、月等)，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，指定了要用的时间单位。

www.TopSage.com

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 函数返回一个表示指定日期的那一天的整数。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)的时间。换句话说，这会返回格林尼治时间。这个值是通过从服务器获取本地时间和本地时区，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>)。

www.TopSage.com

A.6.11 SYSDATETIME

和 GETDATE 函数一样，SYSDATETIME 函数返回当前系统日期和时间。不同之处有两个。第一，SYSDATETIME 函数返回值的精度较高。第二，该函数返回较新的 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 函数，这个函数返回当前系统日期和时间。不过，SYSDATETIMEOFFSET 函数返回的不是简单的 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

和 GETUTCDATE 函数类似，SYSDATETIME 函数返回当前的 UTC 日期和时间。不过，SYSDATETIME 函数返回较新的 datetime2 数据类型(精度为 7)。其语法如下：

```
SYSUTCDATETIME()
```

A.6.14 SWITCHOFFSET

返回从存储的时区偏移量变为指定的新时区偏移量时得到的 datetimeoffset 值。

例如，可从下列示例中看到偏移量的应用。

```
 SWITCHOFFSET ( <datetime to offset>, <time zone offset amount> )
CREATE TABLE dbo.test
  (
  ColDatetimeoffset datetimeoffset
  );
GO
INSERT INTO dbo.test
VALUES ('1998-09-20 7:45:50.71345 -5:00');
GO
SELECT SWITCHOFFSET (ColDatetimeoffset, '-08:00')
FROM dbo.test;
GO
--Returns: 1998-09-20 04:45:50.7134500 -08:00
SELECT ColDatetimeoffset
FROM dbo.test;
--Returns: 1998-09-20 07:45:50.7134500 -05:00
```

A.6.15 TODATETIMEOFFSET

接受一个指定的日期/时间信息，添加给定的时间偏移量，生成一个 datetimeoffset 数据类型的值。其语法如下：

www.TopSage.com

```
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 行受影响)
```

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 层次结构函数

正如前面提到的，hierarchyID 数据类型及相关函数的内容已超出了本书范围。要详细了解下列函数的功能，请参阅对本书相对应的“高级编程”系列。这里只是简单地介绍 SQL Sever 2008 提供的一些函数，以帮助处理 hierarcyID 数据类型和相关的树结构。

A.7.1 GetAncestor

GetAncestor 函数返回特定项的第 n 个父级节点的 hierarchyID。其语法如下：

```
GetAncestor(<numeric expression>)
```

传递的参数决定了希望沿层次结构链上升多少级别来检索其子级节点。例如，1 返回特定项的一级子节点，2 返回二级子节点，0 返回特定项本身的 hierarchyID。

A.7.2 GetDescendant

返回父级节点的一个子节点。

A.7.3 GetLevel

返回一个表示节点在树结构中级别的整数。

A.7.4 GetRoot

返回层次结构树的根节点。

www.TopSage.com

A.7.5 IsDescendantOf

如果下一级是指定项的后代，就返回真。

A.7.6 Parse

该函数将 hierarchyID 的规范字符串表示形式转换为 hierarchyID 值。当从字符串类型转换到 hierarchyID 类型时，隐式调用 Parse 函数。

A.7.7 GetReparentedValue

返回一个节点，其相对于根的路径是指向 newRoot 的路径，随后是从 oldRoot 到当前项的路径。

A.7.8 ToString

其功能等价于 CAST 函数，返回应用该函数的层次结构节点的字符串类型值。

A.8 数学函数

数学函数执行计算的功能。它们分别是：

- ABS
- ACOS
- ASIN
- ATAN
- ATN2
- CEILING
- COS
- COT
- DEGREES
- EXP
- FLOOR
- LOG
- LOG10
- PI
- POWER
- RADIANS
- RAND
- ROUND
- SIGN
- SIN
- SQRT

www.TopSage.com

- SQUARE
- TAN

A.8.1 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.8.2 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，它返回的是表达式的反余弦值)。其语法如下：

```
ACOS(<expression>)
```

这个值必须在 - 1~1 之间，必须为浮点数据类型。

A.8.3 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，它返回的是表达式的反正弦值)。其语法如下：

```
ASIN(<expression>)
```

这个值必须在-1~1 之间，必须为浮点数据类型。

A.8.4 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，它返回的是表达式的反正切值)。其语法如下：

```
ATAN(<expression>)
```

表达式必须为浮点数据类型。

A.8.5 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，它返回的是两个表达式的反正切值)。其语法如下：

```
ATN2(<expression1>, <expression2>)
```

expression1 和 expression2 都必须是浮点数据类型。

A.8.6 CEILING

CEILING 函数返回大于或等于指定数值表达式的最小整数。其语法如下：

```
CEILING(<expression>)
```


www.TopSage.com

A.8.7 COS

COS 函数返回指定表达式中指定角度的三角余弦值。其语法如下：

```
COS(<expression>)
```

给定角度应以弧度为单位的，表达式必须为浮点数据类型。

A.8.8 COT

COT 函数返回指定表达式中指定角度的三角余切值。其语法如下：

```
COT(<expression>)
```

给定角度应以弧度为单位的，表达式必须为浮点数据类型。

A.8.9 DEGREES

DEGREES 函数接受以弧度为单位的角度(表达式)，返回相应的以度数为单位的角度。其语法如下：

```
DEGREES(<expression>)
```

A.8.10 EXP

EXP 函数返回指定表达式的指数值。其语法如下：

```
EXP(<expression>)
```

表达式必须为浮点数据类型。

A.8.11 FLOOR

FLOOR 函数返回小于或等于指定表达式的最大整数。其语法如下：

```
FLOOR(<expression>)
```

A.8.12 LOG

LOG 函数返回指定表达式的自然对数。其语法如下：

```
LOG(<expression>)
```

表达式必须为浮点数据类型。

A.8.13 LOG10

LOG10 函数返回指定表达式的以 10 为底的对数。其语法如下：

```
LOG10(<expression>)
```

表达式必须为浮点数据类型。

www.TopSage.com

A.8.14 PI

PI 函数返回 PI 的常量值。其语法如下：

```
PI()
```

A.8.15 POWER

POWER 函数返回指定表达式的指定幂的值。其语法如下：

```
POWER(<expression>, <power>)
```

A.8.16 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.8.17 RAND

RAND 函数返回 0～1 之间的随机值。其语法如下：

```
RAND([<seed>])
```

可选的种子(seed)值是个整数表达式，种子值指定了用于生成随机值的起始值。可以让 SQL Server 在每次调用该函数时生成一个随机种子值，或是您也可以自己指定一个种子值。

注意：

在指定字面量种子值时要小心。指定一个特定的种子值，SQL Server 将总是返回相同的结果(意味着 RAND 不再随机)。如果真的需要生成一个随机数字，就要么允许 SQL Server 选择一个随机种子，要么使用不断变化的种子，例如自一个特定时间点后的纳秒数。

A.8.18 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 参数指定了表达式应舍入的精度。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)，那么表达式将舍入。如果是 0 以外的值时，将截断表达式的值。

A.8.19 SIGN

SIGN 函数返回表达式的符号。+1 表示正数，0 表示零，－1 表示负数。其语法如下：

```
SIGN(<expression>)
```

A.8.20 SIN

SIN 函数返回角度的三角正弦值。其语法如下：

www.TopSage.com

```
SIN(<angle>)
```

angle 应以弧度为单位，而且必须为浮点数据类型。返回值也必须是浮点数据类型。

A.8.21 SQRT

SQRT 函数返回指定表达式的平方根。其语法如下：

```
SQRT(<expression>)
```

表达式必须为浮点数据类型。

A.8.22 SQUARE

SQUARE 函数返回指定表达式的平方。其语法如下：

```
SQUARE(<expression>)
```

表达式必须为浮点数据类型。

A.8.23 TAN

TAN 函数返回指定表达式的正切值。其语法如下：

```
TAN(<expression>)
```

表达式参数指定了弧度，并且必须为浮点或实数数据类型。

A.9 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。基本的元数据包括以下类型。

- COL_LENGTH
- COL_NAME
- COLUMNPROPERTY
- DATABASEPROPERTY
- DATABASEPROPERTYEX
- DB_ID
- DB_NAME
- FILE_ID
- FILE_NAME
- FILEGROUP_ID
- FILEGROUP_NAME
- FILEGROUPPROPERTY
- FILEPROPERTY
- FULLTEXTCATALOGPROPERTY
- FULLTEXTSERVICEPROPERTY

www.TopSage.com

- INDEX_COL
- INDEXKEY_PROPERTY
- INDEXPROPERTY
- OBJECT_ID
- OBJECT_NAME
- OBJECTPROPERTY
- OBJECTPROPERTYEX
- @@POCID
- SCHEMA_ID
- SCHEMA_NAME
- SQL_VARIANT_PROPERTY
- TYPE_ID
- TYPE_NAME
- TYPEPROPERTY

A.9.1 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.9.2 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.9.3 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参数返回的数据。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——该列已经注册为全文索引
- IsIdentity——该列是一个 IDENTITY 列
- IsIdNotForRepl——该列检查 IDENTITY NOT FOR REPLICATION

www.TopSage.com

- 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 表示真、0 表示假、输入无效时为 NULL。

A.9.4 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——启用自动更新统计信息选项。
- IsBulkCopy——数据库允许无日志记录的操作(例如批量复制程序所执行的操作)。
- IsCloseCursorsOnCommitEnabled——关闭在提交事务时打开的任何游标。
- IsDboOnly——数据库只有 dbo 可以访问。
- IsDetached——分离操作分离了数据库。
- IsEmergencyMode——数据库处于紧急模式中。
- IsFulltextEnabled——数据库启用全文功能。
- IsInLoad——正在加载数据库。
- IsInRecovery——正在恢复数据库。
- IsInStandBy——数据库以只读方式联机，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。

www.TopSage.com

- 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 表示真、0 表示假、输入无效时为 NULL。

A.9.5 DATABASEPROPERTYEX

DATABASEPROPERTYEX 函数是 DATABASEPROPERTY 的超集，也返回指定数据库和属性名的当前设置。语法和 DATABASEPROPERTY 函数很像，如下所示：

```
DATABASEPROPERTYEX('<database>', '<property>')
```

DATABASEPROPERTYEX 函数只比 DATABASEPROPERTY 函数多一些属性，包括以下属性。

- Collation——返回数据库默认的排序规则(记住，在列级别可以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——当在查询执行中发生数据溢出或除以零错误时是否终止查询。

database 参数指定了将要返回的命名属性数据的数据库名。property 参数包含了数据库的属性名。

A.9.6 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.9.7 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.9.8 FILE_ID

FILE_ID 函数返回当前数据库中指定文件名的文件 ID 号。其语法如下：

```
FILE_ID('<file_name>')
```


www.TopSage.com

file_name 参数指定对应文件 ID 的文件的名称。

A.9.9 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 参数指定对应文件名的文件 ID 号。

A.9.10 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 参数指定对应文件组 ID 的文件组的名称。

A.9.11 FILEGROUP_NAME

FILEGROUP_NAME 函数返回指定文件组标识 ID 号的文件组的名称。其语法如下：

```
FILEGROUP_NAME(<filegroup_id>)
```

filegroup_id 参数指定对应文件组名的文件组 ID 号。

A.9.12 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值。其语法如下：

```
FILEGROUPPROPERTY(<filegroup_name>, <property>)
```

filegroup_name 参数指定包含正被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 表示真、0 表示假、输入无效时为 NULL。

A.9.13 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值。其语法如下：

```
FILEPROPERTY(<file_name>, <property>)
```

file_name 参数指定包含正被查询的属性的文件的名称。property 参数指定查询的属性，并且可以是下列值中的一个。

- IsReadOnly——文件为只读。

www.TopSage.com

- 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 表示真，0 表示假，输入无效时为 NULL。

A.9.14 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 参数指定了正被查询的属性。可以查询以下这些属性。

- 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——返回此目录中组成全文索引的唯一键数。
- LogSize——返回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——返回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.9.15 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——在超时之前，返回搜索服务等待完成所有与 SQL Server 数据库服务器的连接以便进行全文索引填充所用的时间(以秒为单位)。
- IsFulltextInstalled——如果已安装全文服务组件，则返回 1；否则返回 0。

A.9.16 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 参数指定了索引的 ID，key_id 参数指定了键的 ID。

A.9.17 INDEXKEY_PROPERTY

INDEXKEY_PROPERTY 函数返回关于索引键的信息。

www.TopSage.com

```
INDEXKEY_PROPERTY(<table_id>, <index_id>, <key_id>, <property>)
```

table_id 参数是待检查的表的 ID 号，其数据类型为整数。使用 OBJECT_ID 找到数字 table_id。idex_id 参数指定索引 ID 号，其数据类型是整数。key_id 参数指定键的索引列位置。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 参数是想要返回其设置的属性的字符串标识符。可能的两个值分别是 ColumnId(返回物理位置的列 ID)和 IsDescending(它返回列排序的顺序，1 为降序，0 为升序)。

A.9.18 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 参数指定待查询的索引属性。property 参数可以是以下可能值中的一个。

- IndexDepth——索引的深度
- IsAutoStatistics——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指定的空格。
- 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(它会返回创建或上次重建索引时所使用的填充因子)外，这个函数的返回值是，1 表示真、0 表示假、输入无效时为 NULL。

A.9.19 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.9.20 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.9.21 OBJECTPROPERTY

OBJECTPROPERTY 函数返回当前数据库中对象的有关信息。其语法如下：

```
OBJECTPROPERTY(<id>, <property>)
```


www.TopSage.com

id 参数指定当前数据库中某个对象的 ID。property 参数指定对象需要返回的信息。property 参数可以是下面这些值中的一个。

CnstIsClustKey
CnstIsColumn
CnstIsDeleteCascade
CnstIsDisabled
CnstIsNonclustKey
CnstIsNotRepl
CnstIsNotTrusted
CnstIsUpdateCascade
ExecIsAfterTrigger
ExecIsAnsiNullsOn
ExecIsDeleteTrigger
ExecIsFirstDeleteTrigger
ExecIsFirstInsertTrigger
ExecIsFirstUpdateTrigger
ExecIsInsertTrigger
ExecIsInsteadOfTrigger
ExecIsLastDeleteTrigger
ExecIsLastInsertTrigger
ExecIsLastUpdateTrigger
ExecIsQuotedIdentOn
ExecIsStartup
ExecIsTriggerDisabled
ExecIsTriggerNotForRepl
ExecIsUpdateTrigger
HasAfterTrigger
HasDeleteTrigger
HasInsertTrigger
HasInsteadOfTrigger
HasUpdateTrigger
IsAnsiNullsOn
IsCheckCnst
IsConstraint
IsDefault
IsDefaultCnst
IsDeterministic
IsExecuted
IsExtendedProc
IsForeignKey
IsIndexable
IsIndexed
IsInlineFunction
IsMSShipped
IsPrimaryKey
IsProcedure
IsQueue
IsQuotedIdentOn
IsReplProc
IsRule
IsScalarFunction
IsSchemaBound
IsSystemTable
IsTable
IsTableFunction
IsTrigger
IsUniqueCnst
IsUserTable
IsView
OwnerId
TableDeleteTrigger
TableDeleteTriggerCount
TableFullTextBackgroundUpdateIndexOn
TableFulltextCatalogId
TableFullTextChangeTrackingOn
TableFulltextDocsProcessed
TableFulltextFailCount
TableFulltextItemCount

www.TopSage.com

TableFulltextKeyColumn
TableFulltextPendingChanges
TableFulltextPopulateStatus
TableHasActiveFulltextIndex
TableHasCheckCnst
TableHasClustIndex
TableHasDefaultCnst
TableHasDeleteTrigger
TableHasForeignKey
TableHasForeignRef
TableHasIdentity
TableHasIndex
TableHasInsertTrigger
TableHasNonclustIndex
TableHasPrimaryKey
TableHasRowGuidCol
TableHasTextImage
TableHasTimestamp
TableHasUniqueCnst
TableHasUpdateTrigger
TableInsertTrigger
TableInsertTriggerCount
TableIsFake
TableIsLockedOnBulkLoad
TableIsPinned
TableTextInRowLimit
TableUpdateTrigger
TableUpdateTriggerCount

除了下列情况外，这个函数的返回值是，1 表示真、0 表示假、输入无效时为 NULL。

- OwnerId——返回对象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——返回指定类型的第一个触发器的 ID。如果不存在该类型的触发器，则返回 0。
- TableDeleteTriggerCount、TableInsertTriggerCount、TableUpdateTriggerCount——返回特定表中指定类型的触发器的数目。
- TableFulltextCatalogId——返回全文目录的 ID(如果有的话)，如果表中不存在全文目录，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以下多种。
  - 0——表示全文填充进程目前空闲。
  - 1——表示正在进行完整填充。
  - 2——表示正在进行增量填充。
  - 3——表示更改正在被分析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，但已被中止(允许其他系统必要时请求执行)或暂停。

可以利用这一选项的反馈确定合适的其他全文相关选项(检查是否正在进行填充，从而知道其他函数是否有效，如 TableFulltextDocsProcessed)。

www.TopSage.com

- TableFulltextDocsProcessed——只在全文索引实际运行时有效，返回自全文索引开始处理任务以来处理的行数。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——只在全文索引实际运行时有效，返回全文索引因为某种原因跳过(没有指明原因)的行数。和 TableFulltextDocsProcessed 返回值一样，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——保留该返回值只是为了向后兼容，在 SQL Server 2005 和更高版本中将总是返回 0。

A.9.22 OBJECTPROPERTYEX

OBJECTPROPERTYEX 函数是 OBJECTPROPERTY 函数的扩展版本。

```
OBJECTPROPERTYEX(<id>, <property>)
```

和 OBJECTPROPERTY 函数一样，id 参数指定了某个对象的 ID。property 参数指定对象需要返回的信息。OBJECTPROPERTYEX 函数支持 OBJECTPROPERTY 函数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，整数或小数是精确的，而浮点数不是——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——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.9.23 @@PROCID

返回当前运行过程的存储过程 ID。

当进程正在运行而且使用大量资源时，该函数是主要的故障检修工具。主要用作 DBA 函数。

A.9.24 SCHEMA_ID

给定一个架构名，返回该架构的内部系统 ID。其语法如下：

```
SCHEMA_ID( <schema name> )
```

A.9.25 SCHEMA_NAME

给定一个内部架构系统 ID，返回该架构用户友好的名称。其语法如下：

www.TopSage.com

```
SCHEMA_NAME( <schema id> )
```

A.9.26 SQL_VARIANT_PROPERTY

SQL_VARIANT_PROPERTY 是一个很强大的函数，返回有关 sql_variant 的信息。这个信息包括基本类型、精度、小数位数、全部字节数、排序规则，以及最大长度的信息。其语法如下：

```
SQL_VARIANT_PROPERTY (expression, property)
```

expression 是 sql_variant 类型的表达式。property 参数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
基本类型	数据类型包括：char、int、money、nchar、ntext、numeric、nvarchar、real、smalldatetime、smallint、smallmoney、text、timestamp、tinyint、uniqueidentifier、varbinary、varchar	sysname
精度	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
小数位数	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
全部字节数	保存元数据和数据值所需的字节数。如果该值大于 900，索引创建将失败	int
排序规则	表示特定 sql_variant 值的排序规则	sysname
最大长度	最大数据类型长度(以字节为单位)	int

www.TopSage.com

A.9.27 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属性，该参数值可以是下面这些值中的一个。

- Precision——返回数字位数/字符数
- Scale——返回数据类型的小数位数
- AllowsNull——返回 1 表示真，0 表示假
- UsesAnsiTrim——返回 1 表示真，0 表示假

A.10 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.10.1 CHANGETABLE

返回表的更改跟踪信息。可以使用这一语句返回表的所有更改或特定行的更改跟踪信息。其语法如下：

```
CHANGETABLE (
      { CHANGES table , last_sync_version
      | VERSION table , <primary_key_values> } )
[AS] table_alias [ ( column_alias [ ,...n ] )
```

A.10.2 CONTAINSTABLE

CONTAINSTABLE 函数用于全文查询中。其语法如下：

```
CONTAINSTABLE (<table>, {<column> | *}, '<contains_search_condition>')
```

A.10.3 FREETEXTTABLE

FREETEXTTABLE 函数用于全文查询中。其语法如下：

```
FREETEXTTABLE (<table>, {<column> | *}, '<freetext_string>')
```


www.TopSage.com

A.10.4 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 是用于访问数据源的 OLE DB 提供程序的 ProgID 的注册名称。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.10.5 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.10.6 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 提供程序的名称。data_source 参数是字符串常量，对应着某个特定的 OLE DB 数据源。user_id 参数是传递到指定 OLE DB 提供程序的用户名。password 参数是与 user_id 相关的密码。

provider_string 参数是提供程序特定的连接字符串，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 参数是架构名称或者是所需对象的对象所有者名称。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.10.7 OPENXML

通过传递 XML 文档作为参数，或者检索 XML 文档并且在一个变量中定义文档，OPENXML 函数用于检查结构并返回数据，就好像 XML 文档是一个表一样。其语法如下：

```
OPENXML(<idoc_int> [in],<rowpattern> nvarchar[in],[<flags> byte[in]])
[WITH (<SchemaDeclaration> | <TableName>)]
```


www.TopSage.com

idoc_int 参数是使用 sp_xml_prepareddocument 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 文档的架构；如果在数据库中有遵循 XML 架构的表，那么就可以使用 TableName。

在使用 XML 文档之前，必须使用 sp_xml_preparedocument 系统过程进行准备。

A.11 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.11.1 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，返回值为 0 表示不能访问该数据库。NULL 值意味着提供的 database_name 无效。其语法如下：

```
HAS_DBACCESS ('<database_name>')
```

A.11.2 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，并且必须用 domain\group 这种格式。role 参数指定了 SQL Server 角色的名称。该角色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；如果当前用户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.11.3 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：

www.TopSage.com

```
IS_SRVROLEMEMBER ('<role>' [,'<login>'])
```

可选的 login 参数是待检查的登录账户的名称；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

如果指定登录账户是指定角色的成员，则函数返回 1；如果登录账户不是指定角色的成员，则返回 0；如果角色或登录名无效，则返回 NULL。

A.11.4 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。

提示：

SUSER_ID 系统函数早就被 SUSER_SID 取代，保留该函数只是为了向后兼容。要避免使用 SUSER_ID，因为它返回的内部值可能随服务器改变而改变(考虑到某个数据库可能还原到一台新的服务器，而其中给定的登录名可能有不同的 SUSER_ID，还是使用 SID 更为可靠)。

A.11.5 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 参数指定用户登录的 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 SUSER_NAME 系统函数只是为了向后兼容，因此请尽可能改用 SUSER_SNAME 函数。

A.11.6 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。如果没有提供登录名，则会使用当前用户。

www.TopSage.com

A.11.7 SUSER_SNAME

SUSER_SNAME 函数返回与指定安全标识号(SID)对应的登录名。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.11.8 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.11.9 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.11.10 USER_NAME

USER_NAME 函数在功能上与 USER_ID 相反，并根据指定的数据库 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对应的 ID 号，如果未提供 user id，则假定为当前用户。

A.12 字符串函数

字符串函数对字符串值执行操作，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

www.TopSage.com

- PATINDEX
- QUOTENAME
- REPLACE
- REPLICATE
- REVERSE
- RIGHT
- RTRIM
- SOUNDEX
- SPACE
- STR
- STUFF
- SUBSTRING
- UNICODE
- UPPER

A.12.1 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.12.2 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 0~255 之间的任何整数。

A.12.3 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 是开始搜索的起始字符位置。如果它是负数或零，则将从 character_string 的起始处查找。

A.12.4 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

www.TopSage.com

回 4。如果没有任何相似的地方，则返回 0。

A.12.5 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 参数指将要提取的最左面部分的字符数据。integer 参数指定从左面开始的字符数，必须为正整数。

A.12.6 LEN

LEN 函数返回指定 expression 中的字符数。其语法如下：

```
LEN(<expression>)
```

A.12.7 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.12.8 LTRIM

LTRIM 函数删除 character_expression 中的前导空格。其语法如下：

```
LTRIM(<character_expression>)
```

A.12.9 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 0~65 535 的正整数。

A.12.10 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 参数是要查找的字符串。可以使用通配符，但是 pattern 参数前后必须有%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.12.11 QUOTENAME

QUOTENAME 函数返回带有分隔符的 Unicode 字符串，添加分隔符可使指定入的字符串成为有效的 SQL Server 分隔标识符。其语法如下：

www.TopSage.com

```
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(')、左方括号或方右括号([])或者双引号(")。默认使用方括号。

A.12.12 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第二个指定字符串的所有实例。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.12.13 REPLICATE

REPLICATE 函数以指定的次数重复 character_expression。其语法如下：

```
REPLICATE(<character_expression>, <integer>)
```

A.12.14 REVERSE

REVERSE 函数返回指定 character_expression 的反转值。其语法如下：

```
REVERSE(<character_expression>)
```

A.12.15 RIGHT

RIGHT 函数返回指定 character_expression 中从右边开始指定个数的字符(给定整数值)。其语法如下：

```
RIGHT(<character_expression>, <integer>)
```

integer 参数必须是一个正整数。

A.12.16 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.12.17 SOUNDEX

SOUNDEX 返回一个由 4 个字符组成的代码(SOUNDEX)，用于计算两个字符串的相似性。其语法如下：

```
SOUNDEX(<character_expression>)
```


www.TopSage.com

A.12.18 SPACE

SPACE 函数返回由重复的空格组成的字符串，由整数表示其长度。其语法如下：

```
SPACE(<integer>)
```

A.12.19 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、数字和空格。decimal 参数是小数点右边的位数。

A.12.20 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 参数是将要被删除和添加的字符串。start 参数指定删除和插入字符的开始位置。length 参数指定要删除的字符数。characters 参数指定要插入到 expression 中的新的字符集。

A.12.21 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字符串，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.12.22 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.12.23 UPPER

UPPER 函数将 character_expression 中的所有小写字符转换为大写字符。其语法如下：

```
UPPER(<character_expression>)
```

A.13 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

www.TopSage.com

- CAST 和 CONVERT
- COALESCE
- COLLATIONPROPERTY
- CURRENT_TIMESTAMP
- CURRENT_USER
- DATALENGTH
- FORMATMESSAGE
- GETANSINULL
- HOST_ID
- HOST_NAME
- IDENT_CURRENT
- IDENT_INCR
- IDENT_SEED
- IDENTITY
- ISDATE
- ISNULL
- ISNUMERIC
- NEWID
- NULLIF
- PARSENAME
- PERMISSIONS
- ROWCOUNT_BIG
- SCOPE_IDENTITY
- SERVERPROPERTY
- SESSION_USER
- SESSIONPROPERTY
- STATS_DATE
- SYSTEM_USER
- USER_NAME

A.13.1 APP_NAME

APP_NAME 函数返回当前会话的应用程序名称，返回值为 nvarchar 类型。其语法如下：

```
APP_NAME()
```

A.13.2 CASE

CASE 函数计算条件列表并返回多个可能结果表达式之一。它有以下两种格式。

- 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。

www.TopSage.com

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.13.3 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 参数指定日期格式的样式。

A.13.4 COALESCE

COALESCE 函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.13.5 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 参数是想确定的排序规则属性，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

www.TopSage.com

A.13.6 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.13.7 DATALENGTH

DATALENGTH 函数返回用于表示表达式的字节数，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可变长度数据。其语法如下：

```
DATALENGTH(<expression>)
```

A.13.8 @@ERROR

返回当前连接上运行的最后一条 T-SQL 语句的错误代码。如果没有错误，则该值为 0。

如果要编写存储过程或触发器，那么这是必需的系统函数，不可或缺。

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，如果想要使用该函数在一条指定语句之后检查错误，那么需要检测下一条语句，或者需要将该函数移到临时保存变量中。一般来说，本书建议在 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 sysmessages 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 存储过程。

A.13.9 FORMATMESSAGE

FORMATMESSAGE 函数根据 sysmessages 表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 参数是 sysmessages 表中消息的 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.13.10 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 参数是将要返回可空性信息的数据库的名称。

www.TopSage.com

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 的默认值。

A.13.11 HOST_ID

HOST_ID 函数返回工作站的 ID 号。其语法如下：

```
HOST_ID()
```

A.13.12 HOST_NAME

HOST_NAME 函数返回工作站的名称。其语法如下：

```
HOST_NAME()
```

A.13.13 IDENT_CURRENT

IDENT_CURRENT 函数返回为指定表生成的最后一个标识值，在指定表的任何会话或作用域内。IDENT_CURRENT 函数类似于@@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 是将要返回其当前标识值的表的名称。

A.13.14 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.13.15 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.13.16 @@INDENTITY

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，那么您会发现一直都在使用这个标识列。可以创建父记录(通常带有您需要检索的标识列的记录)，然后选择@@IDENTITY 来识别子记录需要关联的是哪个值。

www.TopSage.com

如果使用标识值执行对多个表的插入，那么请记住@@INDENTITY 中的值只是最后一个被插入的标识值；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保存到一个临时保存的变量中。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.13.17 IDENTITY

IDENTITY 函数用于将标识列插入到新表中。该函数只用在带有 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 是要赋予表中第一行的值。每一个后续行被赋予下一个标识值，该值等于上一个 IDENTITY 值加上 increment 值。如果既没有指定 seed 值，也没有指定 increment 值，那么二者都默认为 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.13.18 ISNULL

ISNULL 函数检查表达式的 NULL 值，并使用指定的值替换 NULL。其语法如下：

```
ISNULL(<check_expression>, <replacement_value>)
```

A.13.19 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.13.20 NEWID

NEWID 函数创建 uniqueidentifier 类型的唯一值。其语法如下：

```
NEWID()
```

A.13.21 NULLIF

NULLIF 函数比较两个表达式并返回 NULL 值。其语法如下：

```
NULLIF(<expression1>, <expression2>)
```

A.13.22 PARSENAME

PARSENAME 返回对象名称的指定部分。其语法如下：

```
PARSENAME('<object_name>', <object_piece>)
```


www.TopSage.com

object_name 参数指定从指定部分检索的对象名。object_piece 参数指定要返回的对象的部分。object_piece 参数可以为下列这些值之一。

- 1——对象名
- 2——所有者名
- 3——数据库名
- 4——服务器名

A.13.23 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 参数指定对象的 ID。可选的 column 参数指定要返回其权限信息的列名。

A.13.24 @@ROWCOUNT

返回上一条语句影响到的行数。

@@ROWCOUNT 函数是最常用的全局变量，经常用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，那么该行应匹配指定条件；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，那么也会得到返回值 0。

A.13.25 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 函数的返回类型是 bigint 数据类型。其语法如下：

```
ROWCOUNT_BIG()
```

A.13.26 SCOPE_IDENTITY

SCOPE_IDENTITY 函数返回插入同一作用域中(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 IDEN_CURRENT 函数在功能上相似，不过 IDENT_CURRENT 函数不局限于在同一个作用域内插入的标识。

这个函数返回 sql_variant 数据类型，其语法如下：

```
SCOPE_IDENTITY()
```

A.13.27 SERVERPROPERTY

SERVERPROPERTY 函数返回有关当前运行的服务器的属性信息。其语法如下：

```
SERVERPROPERTY('<propertyname>')
```

propertyname 参数可以是表 A-6 中列出的值中的一个。

www.TopSage.com

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 = 个人或桌面引擎 2 = 标准 3 = 企业(对企业、企业评估和开发人员返回该值)
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 = 输入无效或错误
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 =输入无效或错误
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

www.TopSage.com

(续表)

属性名称	返回的值
LicenseType	为 SQL Server 实例安装什么类型的许可： PER_SEAT =每个客户模式 PER_PROCESSOR =每个处理器模式 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，该属性返回 SQL Server 实例的版本，格式为 "major.minor.build"
ProductLevel	返回 SQL Server 实例当前运行的版本值： "RTM"= 原始发布版 "SPn"= Service Pack 版 "Bn"= 测试版
ServerName	Windows NT 服务器和与指定的 SQL Server 实例关联的实例信息

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并且在多点办公的公司很有用的。

A.13.28 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.13.29 SESSIONPROPERTY

SESSIONPROPERTY 函数返回会话的 SET 选项。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，那么这个函数是很有用的。应该少用这个函数，因为不能在运行时对 SET 选项作太多的更改。

www.TopSage.com

A.13.30 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.13.31 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.13.32 USER_NAME

USER_NAME 返回数据库用户名称。其语法如下：

```
USER_NAME([<id>])
```

id 参数指定要返回其用户名的 ID 号，如果省略 id，则假定为当前用户。

A.14 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(A.12 节已经介绍过)
- TEXTPTR
- TEXTVALID

A.14.1 TEXTPTR

TEXTPTR 函数检查对应于 text 列、ntext 列或 image 列的文本指针值，并返回一个 varbinary 值。在运行 READTEXT、WRITETEXT 和 UPDATETEXT 语句前，应检查文本指针确保其指向第一个文本页面。其语法如下：

```
TEXTPTR(<column>)
```

A.14.2 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 参数指定要使用的表名和列名。text_ptr 参数指定要检查的文本指针。

如果指针无效，则返回 0；如果指针有效，则返回 1。

www.TopSage.com

附 B 录

非常简单的连接示例

本书围绕连接性的主题编写。推出的前几本书刻意避开了这一主题，因为这是个庞大的主题——不用整本书很难介绍全面。虽说如此，有 SQL Server 而不允许程序与之连接和没有 SQL Server 一样。我们可以登录进入 Management Studio 并直接编写查询，但实际情况是大量用户实际上从未直接看到数据库。

由于本书只是入门级别的，所以将只涉及非常有限的内容，作为其他相关内容的快速参考，或是起到提示作用。如果您想向别人展示 SQL Server 方面或基于连接的数据解决方案的才能，那么这里建议您购买专门介绍数据访问和连接性的书。

本附录汇集了一些非常简单的连接示例，利用了几种客户端语言和两个不同连接模型中的一些功能。本附录将介绍揭示每种语言/模型混合情况的一些基本的连接示例，而将有关连接原因和连接方式的讨论放到专门介绍连接性的书籍中。

提示：

这里要强调的是，这些示例是基础性的。可以对连接性进行很多的选择和优化。每种连接模型都有其自身的特点，不同的语言有时也会显示其独特功能。如果您想了解相关的高级内容，那么可以参阅本书的后续版本 *Professional SQL Server 2008 Programming* 或其他专门介绍连接性的书籍(它们很可能针对的是一种特定语言和连接模型)。

这里涉及的连接模型包括以下两种。

- ADO.NET
- ADO

在 ADO.NET 示例中，将看到两种不同的关于如何利用提供功能的方法。

说明：

这里提供的所有代码示例都是使用 Visual Studio 2008 创建的。其基本原则也适用于其他 Visual Studio 版本和其他语言提供的编辑器。不过，还是会遇到.NET Framework 3.5 的一些特例。

www.TopSage.com

B.1　一些普通的概念

在深入介绍代码之前，有一些重要的概念需要理解。从过去到现在，出现过很多不同的连接模型——您可能听过 OLE-DB、ODBC、ADO 这样的名称。现在用的连接模型是 ADO.NET 或 LINQ。考虑到这些模型的生存期，如果在推出下一版本的 SQL Server 时出现一种不同的模型，那么也不要感到奇怪，这很正常。

尽管出现了很多不同的模型，但是在所有这些对象模型中还是存在着一些共同的概念——这里作一下简要的介绍。

- **连接**：顾名思义，连接对象是用来定义和建立与数据库的实际通信链接的对象。连接对象的参数信息包括用户名、密码、数据库和想要连接的服务器。存在的一些方法包括连接以及断开连接。
- **命令**：这个对象包含关于想要进行什么操作的信息。一些对象模型不包括这个对象，或者没有这样的功能，但是有这样的概念(它有时作为连接对象的方法)。
- **数据集**：如果查询返回数据，这就是查询的结果。对于执行的查询来说，例如，一个简单的 INSERT 语句不会返回结果，但是如果返回结果，则会返回某种数据集(有时称为结果集或记录集)。数据集对象允许对其中的记录进行迭代(通常在方向上都是向前的，但是一般可以设置为更健壮的定位方法)。这同样也允许对数据进行更新、插入和删除。

B.2 连接示例

本节将提供一些在两种常用语言(C#和 VB.NET)中如何连接的示例。对于每种语言，都会显示两个示例——分别针对两种不同的操作(提取一个简单的结果集和执行一个不返回数据集的查询)。

B.2.1　在 C#中连接

C#是一种相当简洁而且易学的语言，和 VB 很相似，但它同时提供了一些基于 C 的概念，并且在语法上也很接近 C(使得在两者之间的转换更为容易)。

1. 返回一个数据集

```
using System;
using System.Data.SqlClient;

class Program
{
   static void Main()
   {
      //Create some base strings so you can look at these
```


www.TopSage.com

```
        // separately from the commands they run in

        // Integrated Security - next line should be uncommented to use
        string strConnect = "Data Source=(local);Initial Catalog=master;
Integrated Security=SSPI";
        // SQL Security - next line should be uncommented to use
        //string strConnect = "Data Source=(local);Initial Catalog
          =master;User Id=sa;Password=MyPass";

        string strCommand = "SELECT Name, database_id as ID FROM
          sys.databases";

        SqlDataReader rsMyRS = null;

        SqlConnection cnMyConn = new SqlConnection(strConnect);

        try
        {
            // "Open" the connection (this is the first time it actually
            // contacts the database server)
            cnMyConn.Open();

            // Create the command object now
            SqlCommand sqlMyCommand = new SqlCommand(strCommand, cnMyConn);

            // Create the result set
            rsMyRS = sqlMyCommand.ExecuteReader();

            //Output what we got
             while (rsMyRS.Read())
            {
                // Write out the first (ordinal numbers here)
                // column. We can also refer to the column by name
                Console.WriteLine(rsMyRS["Name"]);
            }
            Console.WriteLine();
            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }
        finally
        {
            // Clean up
            if (rsMyRS != null)
            {
                rsMyRS.Close();
            }

            if (cnMyConn != null)
            {
                cnMyConn.Close();
```


www.TopSage.com

```
            }
        }
    }
}
```

2. 执行一个不返回数据集的命令

```
using System;
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        //Create some base strings so you can look at these
        // separately from the commands they run in

        // Integrated Security - next line should be uncommented to use
        string strConnect = "Data Source=(local);Initial Catalog=master;
          Integrated Security=SSPI";
        // SQL Security - next line should be uncommented to use
        //string strConnect = "Data Source=(local);Initial Catalog
          =master;User Id=sa;Password=MyPass";

        string strCommand = "CREATE TABLE Foo(Column1 INT NOT NULL PRIMARY
          KEY)";
        string strCommand2 = "DROP TABLE Foo";

        SqlConnection cnMyConn = new SqlConnection(strConnect);

        try
        {
            // "Open" the connection (this is the first time it actually
            // contacts the database server)
            cnMyConn.Open();

            // Create the command object now
            SqlCommand sqlMyCommand = new SqlCommand(strCommand, cnMyConn);

            // Execute the command
            sqlMyCommand.ExecuteNonQuery();

            Console.WriteLine("Table Created");
            Console.WriteLine("Press enter to continue (you can go check
              to make sure that it's there first) ");
            Console.ReadLine();

            // Change the command
            sqlMyCommand.CommandText = strCommand2;

            sqlMyCommand.ExecuteNonQuery();
```


www.TopSage.com

```
            Console.WriteLine("It's gone");

            Console.WriteLine();
            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }
        finally
        {
            // Clean up
            if (cnMyConn != null)
            {
                cnMyConn.Close();
            }
        }
    }
}
```

B.2.2 在 VB.NET 中连接

VB.NET 继续取代 Visual Basic。尽管我个人日益喜欢 C#，但也有许多人选择使用 VB。

1. 返回一个数据集

```
Imports System
Imports System.Data
Imports System.Data.SqlClient

Module Program

    Sub Main()
        'Create some base strings so you can look at these
        'separately from the commands they run in

        ' Integrated Security - next 2 lines should be uncommented to use
        '     Windows authentication
        Dim strConnect As String = _
            "Data Source=(local);Initial Catalog=master;Integrated
             Security=SSPI"
        ' SQL Security - next 2 lines should be uncommented to use
        '     SQL Server authentication
        'Dim strConnect As String = _
        '     "Data Source=(local);Initial Catalog=master;User
          Id=sa;Password=MyPass"
        Dim strCommand As String = _
            "SELECT Name, database_id as ID FROM sys.databases"

         Dim rsMyRS As SqlClient.SqlDataReader
        Dim cnMyConn As New SqlClient.SqlConnection(strConnect)
        ' "Open" the connection (this is the first time it actually
        ' contacts the database server)
```


www.TopSage.com

```
        cnMyConn.Open()

        ' Create the command object now
        Dim sqlMyCommand As New SqlClient.SqlCommand(strCommand, cnMyConn)

        ' Create the result set
        rsMyRS = sqlMyCommand.ExecuteReader()

        'Output what we got
        Do While rsMyRS.Read
            ' Write out the first (ordinal numbers here)
            ' column. We can also refer to the column by name
            Console.WriteLine(rsMyRS("Name"))
        Loop

        Console.WriteLine()
        Console.WriteLine("Press any key to continue...")
        Console.ReadKey()

        ' Clean up
        rsMyRS.Close()
        cnMyConn.Close()

    End Sub

End Module
```

2. 执行一个不返回数据集的命令

```
Imports System
Imports System.Data
Imports System.Data.SqlClient

Module Program

    Sub Main()

        'Create some base strings so you can look at these
        'separately from the commands they run in

        ' Integrated Security - next 2 lines should be uncommented to use
        Dim strConnect As String = _
            "Data Source=(local);Initial Catalog=master;Integrated
              Security=SSPI"
        ' SQL Security - next 2 lines should be uncommented to use
        'Dim strConnect As String = _
        '     "Data Source=(local);Initial Catalog=master;User
          Id=sa;Password=MyPass"

        Dim strCommand As String = "CREATE TABLE Foo(Column1 INT NOT NULL
          PRIMARY KEY)"
```


www.TopSage.com

```
        Dim strCommand2 As String = "DROP TABLE Foo"

        Dim cnMyConn As New SqlClient.SqlConnection(strConnect)

        ' "Open" the connection (this is the first time it actually
        ' contacts the database server)
        cnMyConn.Open()

        ' Create the command object now
        Dim sqlMyCommand As New SqlClient.SqlCommand(strCommand, cnMyConn)

        ' Execute the command
        sqlMyCommand.ExecuteNonQuery()

        Console.WriteLine("Table Created")
        Console.WriteLine("Press enter to continue (you can go check to make
        sure that it's there first)")
        Console.ReadLine()

        ' Change the command
        sqlMyCommand.CommandText = strCommand2

        sqlMyCommand.ExecuteNonQuery()

        Console.WriteLine("It's gone")

        Console.WriteLine()
        Console.WriteLine("Press any key to continue...")
        Console.ReadKey()

        ' Clean up
        cnMyConn.Close()

    End Sub

End Module
```

B.3 LINQ 简介

在我编写本书时，LINQ 可以说是风行一时。对于少数未听说过 LINQ 的人来说，这是一种基于同类数据源的新访问方法(更准确地说，是新的尝试)。LINQ 有其自己的查询语言——该语言类似于 SQL，但还是有区别，以至于在两者间转换需要花一些时间。

尽管 LINQ 有可能成为新的访问数据的方法，但我还是持怀疑态度。不要误会。LINQ 确实提供了一些非常棒的功能，但是由于多年来不少 LINQ 类型的数据访问概念都只是昙花一现，因此我对此类模型的长久性表示怀疑。

LINQ 的全面讨论超出了本书范围，而且其查询语法也是不同的，因此不作介绍也是为了避免给初学者造成混淆。虽说如此，但还是值得简单了解一下 LINQ。

www.TopSage.com

LINQ 使用一种强类型化的模型，允许查询各种类型的数据。尽管访问 SQL Server 的概念与本书最相关，但要注意的是，LINQ 非常擅长于与非表格式的数据源连接。LINQ 不仅可用于访问 XML(XPath 通常是更好的选择，但您可能希望使用 LINQ 与其他数据访问保持一致)，而且可用于访问对象集合之类的组件。简言之，对于返回的数据类型来说，LINQ 是非常灵活的。

除了灵活性外，LINQ 的主要优点在于其是强类型化的。以前的访问模型使用绑定模型，在那种情况下，程序直到运行时才确定返回的数据类型。LINQ 利用了对数据的早期绑定，因此在编码过程中可以很早就捕获大量类型不匹配或类似的错误。当然，这不是绝对可靠的，LINQ 只是在特定领域更有优势。

如果您要对所有的数据访问都使用 LINQ，那么在使用之前，我希望您已经了解了 LINQ 是最佳选择的原因。不要因为 LINQ 有很棒的功能就使用它，而是要确保 LINQ 比其他选择更适合您的需求。

www.TopSage.com

附 C 录

学 习 资 源

案例下载：

- Http://www.minesage.com/html/BookSamples/

在线课程和网络学习：

- SQL Server 2008 BI 解决方案提升 Top10 系列课程：http://www.microsoft.com/china/msdn/events/webcasts/shared/webcast/Series/BI2008.aspx
- SQL Server 2008 开发新特性系列课程：http://www.microsoft.com/china/msdn/events/webcasts/shared/webcast/Series/SQLServer08.aspx
- 技术时文：http://www.minesage.com/msu/minesage_43_371.html
- 网络视频：http://www.minesage.com/msu/minesage_43_372.html

系列培训：

迈思奇商业智能培训中心(MineSage University，MSU)隶属于北京迈思奇科技有限公司，是国内商业智能培训的知名品牌。自 2004 年开始，MSU 不断推出符合市场和企事业需求的商业智能系列培训百余次。

免费讲座：MSU 与微软公司独家合作，开设了 SQL Server BI 系列课程，拥有多个系列 100 多次讲座，以 MSDN 和 TechNet 的网络广播形式广泛传播，并被集成到微软技术支持包中。

定期培训：MSU 定期开设 3~5 天的 BI 定期培训，面向信息技术管理层、开发人员和 BI 技术爱好者系统地传授微软主流商业智能技术和应用。

企业内训：根据各企业的不同需求，MSU 的高级讲师会到企业中去，直接帮助客户搭建符合本企业的 BI 原型，并在此基础上指导客户深入理解和学习 BI 各项技术。

具体的课程培训如表 C-1 所示。

表 C-1

课程编号	课程名称
MSU51008	SQL Server 2008 商业智能核心技术与应用实战—中级 (5 天)
MSU51508	MOSS2007 商业智能集成应用平台—中级(5 天)
MSU51108	MDX 解决方案—高级(3 天)
MSU51208	数据挖掘原理与应用—高级(3 天)

http://www.minesage.com/msu/minesage_40.html

www.TopSage.com

PROGRAMMER TO PROGRAMMER™

SQL Server 2008编程入门经典(第3版)

Beginning Microsoft SQL Server 2008 Programming

本书由浅入深逐步介绍了SQL Server数据库的高级主题，重点讨论了SQL Server 2008的特殊功能以及与其他编程环境的不同之处。作者Robert Vieria是Microsoft SQL Server方面的权威，他采用通俗易懂的方法揭示了SQL Server 2008核心组件的重要更改。

本书首先概述了数据库设计的概念，介绍了如何用SQL Server 2008实现这些基本概念。然后，讲述了RDBMS(关系数据库管理系统)的功能和它在开发系统架构方面的优势。SQL Server 2008的新增内容和更改包括：DATE和TIME数据类型、hierarchyID数据类型、MERGE命令和多行插入，以及递归查询等。本书旨在帮助您快速地掌握Microsoft SQL Server 2008的编程技术。

本书主要内容

- RDBMS如何存储、管理和检索数据
- 创建和修改表的方法
- 数据库范式的多种形式
- 编写脚本和使用存储过程的技巧
- 索引的优缺点
- 锁和死锁对系统性能的各种影响
- 理解触发器及其使用方式

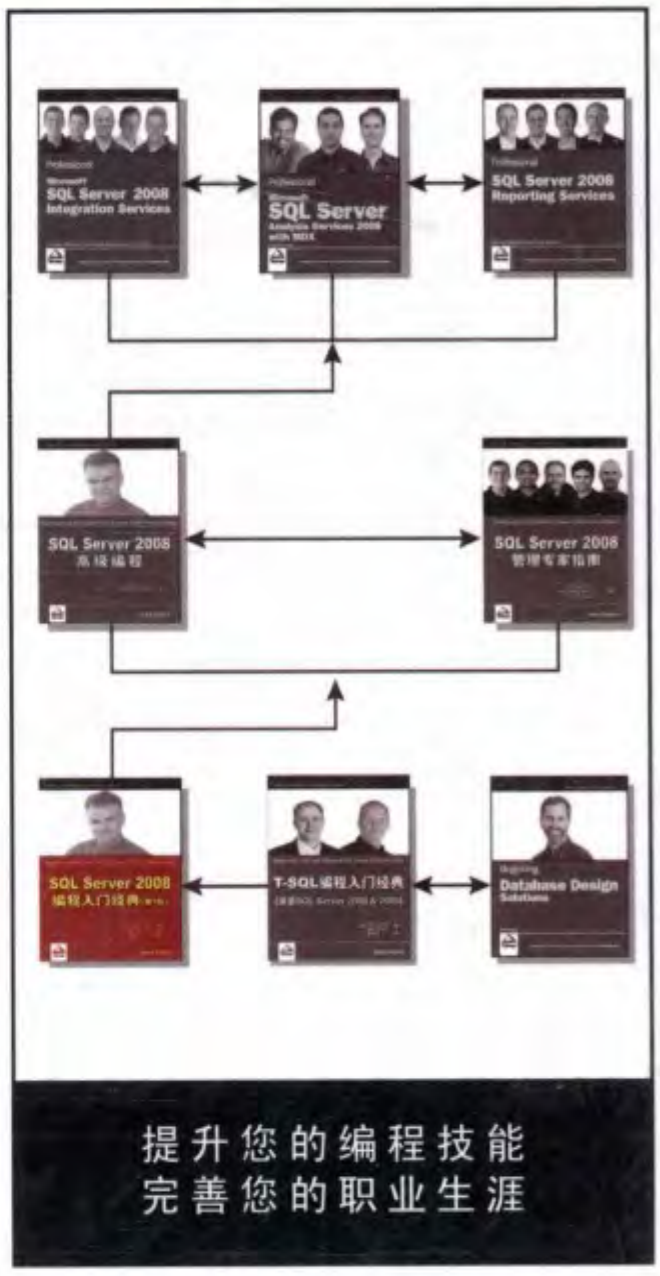
本书读者对象

本书适合于希望全面了解数据库设计概念和学习SQL的开发人员。读者具有数据库方面的基础知识有助于更好地理解本书的内容。

本书源代码下载及技术支持

http://www.wrox.com

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



提升您的编程技能
完善您的职业生涯

Wrox Beginning guides are crafted to make learning programming languages and technologies easier than you think, providing a structured, tutorial format that will guide you through all the techniques involved.

p2p.wrox.com
The programmer's resource center

www.wrox.com

ISBN 978-7-302-21432-8

9 787302 214328 >

定价：69.80元

wrox
An Imprint of WILEY

图书上架分类建议	数据库开发
	SQL Server

读者信箱：wkservice@vip.163.com