

锋利的SQL

```
CREATE TABLE Waiters
(start_time datetime,end_time datetime)
MAX(W1.room_id)
FROM Waiters AS W1INNER JOIN Waiters AS W2
waiter_nameAND W2
W1.start_time GROUP BY W1.waiter_name,
W1.room id
```

FROM Freights AS F2WHEF

人民邮电出版社
POSTS & TELECOM PRESS

每多学一点知识，就少写一行代码。

锋利的SQL

```
CREATE TABLE Waiters
(
    room_id int, waiter_name char(20),
    start_time datetime, end_time datetime
)
SELECT W1.waiter_name, W1.start_time, W1.end_time,
MAX(W1.room_id) AS tally
FROM Waiters AS W1
GROUP BY W1.waiter_name, W1.start_time, W1.end_time
ORDER BY W1.waiter_name, W1.start_time, W1.end_time
TH T1 (waiter_name, start_time, end_time, tally)

-- 统计每个服务员在一天中工作的总时长
SELECT W1.waiter_name, W1.start_time, W1.end_time,
SUM(DATEDIFF(MINUTE, W1.start_time, W1.end_time)) AS tally
FROM Waiters AS W1
GROUP BY W1.waiter_name, W1.start_time, W1.end_time
ORDER BY W1.waiter_name, W1.start_time, W1.end_time
TH T1 (waiter_name, start_time, end_time, tally)
```

爱技术、爱分享，爱各种数据库，
更爱自己的著作堆积成山，不同凡响。
我不是什么技术砖家，不是什么业界大腕，

我是张洪举。

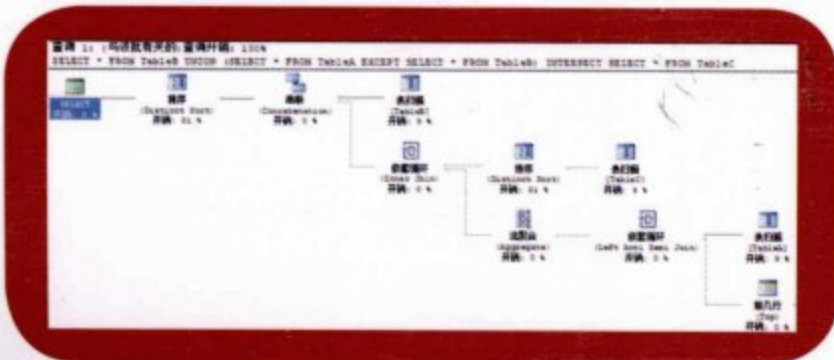
我没什么特别，我很特别，
我和别人不一样，我和你一样，
我是一名疯狂的

SQL爱好者。

● 由浅入深的体系结构



● 配合内容讲解的大量图表



● 稳定高效、易于理解的代码，并提供详细的注释

```
USE AdventureWorks;
GO
-- 定义并设置变量，指定尝试提交更新的次数
DECLARE @retry INT;
SET @retry = 5;
-- 如果被作为死锁牺牲品，保持尝试更新
WHILE (@retry > 0)
BEGIN
```

● 可操作性很强的实例，帮助读者举一反三



分类建议：计算机 / 数据库

人民邮电出版社网址：www.ptpress.com.cn

封面设计：任文杰



ISBN 978-7-115-23502-2



9 787115 235022 >

ISBN 978-7-115-23502-2

定价：49.00 元

锋利的SQL

张洪举 编著

```
CREATE TABLE Waiters
(start_time datetime,end_time datetime)
MAX(W1.room_id)
FROM Waiters AS W1INNER JOIN Waiters AS W2
waiter_nameAND W2.
W1.start_time GROUP BY W1.waiter_name,
W1.room_id
```

W1.
start_
AS
(SELECT

W1.

W1.end_time,

FROM

INNER JOIN

ON W1.waiter_

AND W2.start_

AND W2.end_

waiter_name,

waiter_na

GROUP BY

waiter_

MAX

waiter_name,

COUNT(*)

INNER

waiter

W1.start_

AS

L2.loan

L1.loan_

AS naext

bal FROM

SELE CT

loan_ date,

(DAY,loan_date,

(loan_date) FR

WHERE L2.l oan

AND L2.loan_date > L

balFROM Loans A S L1;

(SELECT MIN(F2.Nu mb)

WHERE F2.Numb > F1. Numb)

CREATE TABLE Freights(Num

INSERT INTO Freights VALU

SELECT n1 + 1 AS start_id, n2

(SELECT MIN(F2.Numb) FROM Fre

FROM Freights AS F1) AS

AS(SELECT F1.Numb AS n1, (S

(room_id int,waiter_name char(20),
SELECT W1.waiter_name, W1.start_time,W1.end_time,

COUNT(*) AS tally

ON W1.waiter_name = W2.

start_time <= W1.start_timeAND W2.end_time >

W1.start_time, W1.end_time,

ORDER BY W1.waiter_name,

WI TH T1 (waiter_name,

tally) — 定义CTE表达式的名称和列

W1. waiter_na me,

rt_time,

M T1

NT(*) AS tal

AS W1

AS W2

.waiter_name

start_time

GROUP BY W1

)SELECT

tally

SELECT T1.

name,

FROM (SELECT W1.

W1.end_time,

.waiters AS W1

ON W1.waiter_name = W2.

time >

time)

date >

date)

date,

loan_id,

DATEDIFF

(SELECT MIN

OM Loans AS L2

id = L1.loan_id

(12,2)); 1,loan_date)) AS diff_days,

SELECT F1.Numb AS n1,

FROM Freights AS F2

INTO AS n2FROM Freights AS F1;

nt NOT NULL;

1),(2),(3),(4),(87),(89),(99),(100);

nd_idFROM (SELECT F1.Numb AS n1,

ERE F2.Numb > F1.Numb)AS n2

n2 - n1 > 1;WITH F3 (n1, n2)

mb) FROM Freights AS F2WHER

JOIN Waiters AS W2ON W1.waiter_name = W2.
nameAND W2.start_time <= W1.start_timeAND W2.end_
timeGROUP BY W1.waiter_name, W1.start_time, W1.end_

T1GROUP BY T1.waiter_name;

CREATE TABLE Loans (loan_id int,

loan_date date, summary char(10),

dr_amt decimal

(12,2),cr_amt

decimal(12,2),

bal decimal

(12,2));

INSERT

INTO

人民邮电出版社

北京

PDG

图书在版编目 (C I P) 数据

锋利的SQL / 张洪举编著. — 北京 : 人民邮电出版社, 2010. 11
ISBN 978-7-115-23502-2

I. ①锋… II. ①张… III. ①关系数据库—数据库管理系统, SQL Server IV. ①TP311.138

中国版本图书馆CIP数据核字(2010)第160625号

内 容 提 要

本书从基础、开发、性能调整和实战4个方面介绍了SQL技术及其应用,包括数据库管理、表管理、索引管理、基本查询、子查询、联接和APPLY运算符、操作结果集、窗口计算和表旋转、数据修改、视图、游标、存储过程、触发器、用户自定义函数、事务处理、并发访问控制、查询的优化与执行等内容。

本书既覆盖了改善效率和性能的普通SQL技术,也深入探讨了SQL新技术,更包含一些实用的查询解决方案,希望本书能够成为引领读者进入SQL查询殿堂的捷径。

锋利的SQL

- ◆ 编 著 张洪举
责任编辑 杜 洁
- ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京艺辉印刷有限公司印刷
- ◆ 开本: 800×1000 1/16
印张: 24.5
字数: 595千字
印数: 1—4000册
- 2010年11月第1版
2010年11月北京第1次印刷

ISBN 978-7-115-23502-2

定价: 49.00元

读者服务热线: (010)67132692 印装质量热线: (010)67129223
反盗版热线: (010)67171154

前言

从 2009 年年初开始动笔到最终定稿，本书的编写花费了我一年多的时间，大大超出了先前预期的计划。当时，书刚刚开了个头，我便被安排参加了单位的一个流程优化项目，其中包含 5 个新系统的开发和 1 个旧系统的改造。同时，我自己还主持着一个管理信息系统的开发工作。项目目标定义、业务需求分析、系统架构设计，环环相扣，整天忙得不可开交，写作成了真正忙里偷闲的事。

当系统一个个瓜熟蒂落，能够自己支配的时间才稍微充裕了一些，这本书的写作才算步入正轨。好在写作只是我的一种业余爱好，没有其他方面的压力，唯一的目的是希望能够创作一些精品图书。

之所以要写作这本图书，主要出于两方面的原因：一是伴随着各种数据库技术日新月异的发展，无论是哪种数据库产品，想用有限的篇幅去描述它的全貌，几乎是不可能完成的任务。所以我就在考虑能否抽取出各种数据库产品中一些大家共同关心的内容，进行深入细致的挖掘，而 SQL 无疑是这方面的首选。二是在与一些开发公司的合作中，发现公司间的 SQL 应用情况也差异很大，一些公司出于产品的可移植性考虑，拒绝使用一些新的 SQL 技术，甚至尽量避免在服务器上部署存储过程。所以，我希望在深入地讨论一些常用 SQL 技术的同时，也尽可能多地介绍一些 SQL 的新技术，从而解除大家对新技术的恐惧感，对新技术的推广能够起到一定的推波助澜作用。

本书特点

本书既覆盖了改善效率和性能的普通 SQL 技术，也深入探讨了 SQL 新技术，更包含一些实用的查询解决方案，希望本书能够成为引领读者进入 SQL 查询殿堂的捷径。

本书的内容是基于 SQL Server 数据库产品进行讨论的。不过，无论是哪种数据库产品的 SQL，由于大家都在遵循 ANSI-SQL 标准，所以差别并不大。数据库开发人员在跨越不同的数据库产品时，一般不会遇到什么障碍。

本书在介绍各种查询语法时，更注重对查询逻辑思维方式的引导和介绍。只有这样，才可以帮助读者在阅读之后举一反三，提升自己动手解决实际问题的能力。

本书适用读者

本书是按照由浅入深、循序渐进方式对 SQL 进行介绍的，既包含了入门知识，也包含了深层次技术的讨论。即使是最基本的查询语句，我们也会尽力为读者提供解决深层次问题的能力。也就是说，同样一个问题，开发人员可以写几十甚至上百行的 SQL 语句来解决问题，也有可能仅通过一条 SELECT 语句就可以解决问题。作为 SQL 而言，虽然代码最简化并不一定是性能最优化，但至少是对思维能力的一种提升。

从这个角度而言，本书可以作为 SQL 入门书籍，也可以作为 SQL 程序员、DBA 的参考书籍。

本书内容与结构

本书共 19 章，可大致分为基础篇、开发篇、性能调整篇和实战篇，共 4 部分。

基础篇包括第 1 章至第 10 章。其中，第 1 章是对查询工具、书写规范等基本内容的介绍，第 2 章至第 5 章是对数据库、表和索引的介绍，第 5 章至第 10 章则介绍了使用 SELECT 进行数据查询和使用 INSERT、UPDATE、DELETE 进行数据修改的各个方面。

开发篇包括第 11 章至第 15 章。如果将 SELECT、INSERT、UPDATE 和 DELETE 作为基本查询语句，则 IF...ELSE、WHILE 和 TRY...CATCH 构造等则可以看作是 SQL 编程语句。在存储过程、触发器等对象中可以通过这些语句实现一些复杂的逻辑处理。如果你曾经是一位使用 VB 或 VC 程序员，在学习 C/S 或 B/S 编程时，你应当掌握这种服务器端的编程工具，从而将业务逻辑计算合理地分布到服务器和客户端。

性能调整篇包括第 16 章至第 18 章。第 16 章和第 17 章介绍的是事务处理机制和并发访问控制。其实，无论是微软还是 Oracle、IBM，其数据库产品的核心功能都是一样的，即在保证数据完整性的前提下提供最大的并发支持。数据库系统是通过“锁机制”来实现的，数据库引擎都提供有多种粒度的锁定模式，从而允许用户可以根据需要将资源锁定在适当的级别，尽量减少锁定开销。第 18 章则是讨论了查询优化器的工作原理，重用查询计划，可以减少额外的编译开销，提高服务器性能。

实战篇仅包含第 19 章的内容，提供了同一时间范围内并发数统计、时间段天数统计、数字范围统计、地域范围内最大数统计等较为常用问题的解决方案。

本书声明

本书实例中使用的操作系统是 Windows 7，使用的数据库是 SQL Server 2008 开发者版本，开发工具是 Visual Studio 2008。

感谢

在本书的完成过程中，得到了诸多 SQL Server 技术专家和爱好者的支持和帮助。他们无私和热情的参与，使本书的内容更加实用和更具指导性，在此一并表示感谢。他们是王向东、秦广、魏兰花、凌亚东、王亚羽、陈雨薇、王光辉、高存亨、桑晓红、王新河、张宪国、李联国、韩燕军。

由于时间仓促，加之水平有限，书中不足之处在所难免，敬请读者批评指正。

编 者
2010 年 9 月

目 录

基 础 篇

第 1 章 SQL 简介2	
1.1 SQL 的历史起源.....2	
1.1.1 CODASYL.....2	
1.1.2 IMS.....3	
1.1.3 RDBMS 和 SQL.....3	
1.1.4 ANSI 和 SQL 方言.....4	
1.2 Transact-SQL 语言的类型.....5	
1.2.1 DDL 语句.....5	
1.2.2 DML 语句.....6	
1.2.3 编程和流控制语句.....7	
1.2.4 SQL 语句的批处理.....9	
1.3 Transact-SQL 语法.....10	
1.3.1 使用标识符进行对象引用.....10	
1.3.2 设置对象的数据类型.....11	
1.3.3 函数.....14	
1.3.4 使用表达式求值.....15	
1.3.5 使用运算符进行计算操作.....15	
1.3.6 使用注释符添加 SQL 语句说明.....15	
1.3.7 保留关键字.....16	
1.4 常量和变量.....16	
1.4.1 常量的类型.....16	
1.4.2 变量的类型.....18	
1.5 运算符.....21	
1.5.1 使用算术运算符执行数学运算.....21	
1.5.2 使用赋值运算符为变量赋值.....22	
1.5.3 使用位运算符执行按位运算.....22	
1.5.4 使用比较运算符进行大小比较.....23	
1.5.5 使用逻辑运算符进行条件测试.....24	
1.5.6 字符串串联运算符.....25	
1.5.7 一元运算符.....25	
1.6 常用函数.....25	
1.6.1 聚合函数.....25	
1.6.2 配置函数.....27	
1.6.3 游标函数.....28	
1.6.4 日期和时间函数.....29	
1.6.5 数学函数.....30	
1.6.6 数据类型转换函数.....32	
1.6.7 字符串函数.....35	
1.6.8 文本和图像函数.....38	
1.7 查询工具.....38	
1.7.1 使用 Management Studio 进行 Windows 方式查询.....39	
1.7.2 使用 sqlcmd 进行 MS-DOS 方式查询.....39	
1.8 SQL 书写规范.....40	
1.8.1 使用大小写规范提高词义 识别能力.....40	
1.8.2 使用空格提供更好的语言 标记区分.....42	
1.8.3 使用缩进提高语句的逻辑层次 表达能力.....42	
1.8.4 使用垂直空白道提高关键词 与参数的区分能力.....43	
1.8.5 使用分组进行语句的段落划分.....43	
第 2 章 数据库管理44	
2.1 创建数据库.....44	
2.1.1 数据库文件和文件组.....45	
2.1.2 CREATE DATABASE 语句的 语法格式.....46	
2.1.3 创建数据库示例.....48	
2.1.4 判断数据库是否已经存在.....50	
2.2 修改数据库.....51	
2.2.1 扩展数据库和文件.....51	
2.2.2 向数据库中添加、删除和 修改文件组.....52	
2.2.3 收缩数据库和文件.....53	
2.2.4 设置数据库选项.....55	
2.2.5 重命名数据库.....58	

2.3	删除数据库	58
第 3 章	表管理	59
3.1	表的物理存储方式	59
3.1.1	最基本的数据存储单位： 数据页	59
3.1.2	最基本的管理空间单位：区	60
3.2	创建表	61
3.2.1	创建基本表	61
3.2.2	使用允许和禁止空值设置 进行值约束	61
3.2.3	使用标识符和全球唯一标识符 创建唯一值	62
3.2.4	为列指定默认值	63
3.3	修改表	63
3.3.1	为表添加新列	64
3.3.2	修改表中的列	64
3.3.3	删除表中的列	65
3.4	重命名和删除表	66
3.5	临时表	67
3.5.1	创建本地表和全局表	67
3.5.2	使用表变量代替临时表	68
第 4 章	索引管理	69
4.1	索引的基础知识	69
4.1.1	索引的类型	69
4.1.2	索引的特征	73
4.1.3	常规索引设计规则	73
4.2	创建索引	75
4.2.1	最大索引限制	75
4.2.2	限制索引参与的数据类型	75
4.2.3	创建聚集索引	76
4.2.4	创建非聚集索引	77
4.2.5	创建具有包含性列的索引	78
4.2.6	为计算列创建索引	79
4.3	修改索引	81
4.3.1	禁用索引	81
4.3.2	重新组织和重新生成索引	82
4.3.3	设置索引选项	84
4.3.4	重命名索引	85
4.4	删除索引	85

第 5 章	基本查询	86
5.1	基本的 SELECT 语句	86
5.1.1	SELECT 语句的结构	86
5.1.2	数据库对象的引用规则	88
5.2	使用选择列表和表别名	89
5.2.1	选择所有列	89
5.2.2	选择特定列	89
5.2.3	在选择列表中使用常量、函数 和表达式	90
5.2.4	使用表别名简化表引用	93
5.3	使用 WHERE 子句筛选行	94
5.3.1	使用比较搜索条件	94
5.3.2	使用范围搜索条件	95
5.3.3	使用列表搜索条件	95
5.3.4	使用模式匹配搜索条件	97
5.3.5	使用 NULL 比较搜索条件	99
5.4	使用 GROUP BY 子句和聚合函数 进行分组计算	100
5.5	使用 HAVING 子句从分组后结果 中筛选行	101
5.6	使用 ORDER BY 子句进行排序	102
5.6.1	指定排序列	102
5.6.2	指定排序顺序	103
5.6.3	指定排序规则	104
5.7	使用 TOP 子句和 SET ROWCOUNT 限制结果集	107
5.7.1	使用 TOP 子句返回前几行	108
5.7.2	使用 SET ROWCOUNT 语句 限制结果集大小	109
5.8	使用 DISTINCT 消除重复行	109
5.9	查询的逻辑处理	110
5.9.1	逻辑处理过程简介	111
5.9.2	步骤 1：使用 FROM 确定输入表	115
5.9.3	步骤 2：使用 WHERE 筛选数据	119
5.9.4	步骤 3：进行数据分组	120
5.9.5	步骤 4：使用 HAVING 筛选数据	122

5.9.6	步骤 5: 通过 SELECT 列表 确定返回列	122	7.4.3	使用右外部联接保留右表 全部行	156
5.9.7	步骤 6: 使用 ORDER BY 子句 排序查询结果	123	7.4.4	使用完全外部联接保留两侧表 全部行	157
第 6 章	子查询	125	7.5	自联接	158
6.1	在选择列表中使用子查询	125	7.5.1	使用不同列实现自联接	158
6.1.1	子查询示例	125	7.5.2	使用同一列实现自联接	159
6.1.2	子查询与联接的关系	126	7.6	多表联接	160
6.2	含有 IN 和 EXISTS 的子查询	127	7.6.1	顺序联接	160
6.2.1	使用含有 IN 的子查询进行 单列匹配	127	7.6.2	嵌套联接	161
6.2.2	使用含有 EXISTS 的子查询 进行整行匹配	129	7.6.3	指定联接的物理顺序	163
6.2.3	含有 NOT IN 和 NOT EXISTS 的子查询	130	7.6.4	多表联接示例	165
6.3	使用含有比较运算符的子查询	132	7.7	联接算法	167
6.4	使用 ANY、SOME 或 ALL 关键字 进行批量比较	134	7.7.1	嵌套循环联接	167
6.5	使用多层嵌套子查询	136	7.7.2	合并联接	168
6.6	子查询应遵循的规则	136	7.7.3	哈希联接	169
第 7 章	联接和 APPLY 运算符	138	7.7.4	使用联接提示强制联接策略	171
7.1	联接的基本知识	138	7.8	使用 APPLY 运算符	173
7.1.1	联接的语法格式	138	第 8 章	操作结果集	176
7.1.2	联接所使用的逻辑处理阶段	139	8.1	合并结果集	176
7.1.3	列名限定和选择列表的使用	140	8.1.1	使用 UNION 与 UNION ALL 进行结果集合并	177
7.1.4	联接条件设定	141	8.1.2	使用 ORDER BY 子句对合并 结果集排序	178
7.2	交叉联接	141	8.1.3	结果集的合并顺序	178
7.2.1	交叉联接的语法格式	142	8.2	查询结果集的差异行	179
7.2.2	使用交叉联接查询全部数据	142	8.2.1	使用 EXCEPT 运算符	179
7.2.3	使用交叉联接优化查询性能	145	8.2.2	查询全部差异行	181
7.2.4	为交叉联接添加 WHERE 子句	146	8.3	查询结果集的相同行	182
7.3	内部联接	147	8.3.1	使用 INTERSECT 运算符	183
7.3.1	内部联接的语法格式	147	8.3.2	查询全部相同行	183
7.3.2	使用等值进行内部联接	148	8.4	UNION、EXCEPT 和 INTERSECT 的 执行顺序	184
7.3.3	使用不等值进行内部联接	150	8.5	在其他语句中使用 UNION、EXCEPT 和 INTERSECT	186
7.4	外部联接	152	8.5.1	使用 INTO 子句指定结果 存储位置	186
7.4.1	外部联接的语法格式	152	8.5.2	突破结果集操作的限制	186
7.4.2	使用左外部联接保留左表 全部行	153			

8.6 使用公用表表达式	188	10.1 插入数据	219
8.6.1 CTE 的语法结构	188	10.1.1 使用 INSERT 和 VALUES 插入行	219
8.6.2 多 CTE 定义和 CTE 的多次引用	190	10.1.2 使用 INSERT 和 SELECT 子查询插入行	221
8.6.3 CTE 的间接嵌套	192	10.1.3 使用 INSERT 和 EXEC 插入行	221
8.6.4 使用递归 CTE 返回分层数据	193	10.1.4 使用 SELECT INTO 插入行	222
8.7 汇总数据	200	10.2 更新数据	223
8.7.1 使用 CUBE 汇总数据	200	10.2.1 使用 SET 和 WHERE 子句 更新数据	223
8.7.2 使用 ROLLUP 汇总数据	201	10.2.2 使用 FROM 子句更新数据	224
8.7.3 区分空值和汇总值	202	10.2.3 使用 CTE 和视图更新数据	226
8.7.4 返回指定维度的汇总	203	10.3 删除数据	226
第 9 章 窗口计算和表旋转	205	10.3.1 使用 DELETE 删除行	227
9.1 窗口和开窗函数	205	10.3.2 使用 TRUNCATE TABLE 删除所有行	228
9.2 基于窗口的排名计算	206	10.4 使用 TOP 限制数据修改	228
9.2.1 使用 ROW_NUMBER() 实现 分区编号	206	10.4.1 使用 TOP 限制插入数据	229
9.2.2 使用 RANK() 和 DENSE_RANK() 函数实现分区排名	208	10.4.2 使用 TOP 限制更新数据	229
9.2.3 使用 NTILE() 函数实现 数据分组	209	10.4.3 使用 TOP 限制删除数据	230
9.3 基于窗口的聚合计算	210	10.5 使用 OUTPUT 输出受影响行的信息	230
9.3.1 分区聚合计算与联接的比较	211	10.5.1 在 INSERT 中使用 OUTPUT 子句	230
9.3.2 对不同类型分区的聚合计算	212	10.5.2 在 DELETE 中使用 OUTPUT 子句	231
9.4 表旋转	213	10.5.3 在 UPDATE 中使用 OUTPUT 子句	233
9.4.1 使用 PIVOT 运算符将表的 行转换为列	213		
9.4.2 使用 UNPIVOT 运算符将表的 列转换为行	217		
第 10 章 数据修改	219		

开 发 篇

第 11 章 视图	235	11.4 删除和重命名视图	242
11.1 创建视图	235	第 12 章 游标	243
11.1.1 创建简单视图	235	12.1 创建游标的步骤	243
11.1.2 创建索引视图	236	12.2 快速只进游标和可滚动游标	245
11.1.3 创建分区视图	237	12.3 静态游标、动态游标和由键集驱动 的游标	248
11.2 修改视图	237		
11.3 更新视图中的数据	239		

12.4 使用可更新游标进行数据更新	249	14.1.5 嵌套和递归触发器	285
第 13 章 存储过程	250	14.1.6 INSTEAD OF 触发器	288
13.1 存储过程的类型	250	14.2 使用 DDL 触发器	292
13.1.1 用户定义的存储过程	250	14.2.1 激发 DDL 触发器的 DDL 事件 和事件组	292
13.1.2 扩展存储过程	251	14.2.2 创建 DDL 触发器	295
13.1.3 系统存储过程	251	14.3 CLR 触发器	298
13.2 SQL 存储过程	251	14.3.1 SqlTriggerContext 类	298
13.2.1 创建存储过程	251	14.3.2 创建 CLR DML 触发器的步骤	301
13.2.2 修改存储过程	256	14.3.3 创建 CLR DDL 触发器的步骤	305
13.2.3 存储过程的重新编译	256	14.4 修改、删除和禁用触发器	307
13.2.4 存储过程的错误处理	258	14.4.1 DML 触发器	307
13.3 CLR 存储过程	266	14.4.2 DDL 触发器	308
13.3.1 创建一个具有输出参数的 CLR 存储过程	266	14.4.3 CLR 触发器	309
13.3.2 创建返回行集和信息的 CLR 存储过程	271	第 15 章 用户自定义函数	310
13.3.3 删除 CLR 存储过程和程序集	273	15.1 标量 UDF	310
13.3.4 CLR 与 SQL 存储过程的择取 建议	274	15.2 表值 UDF	312
13.4 嵌套存储过程	275	15.2.1 使用内联式表值 UDF 实现 参数化视图功能	312
第 14 章 触发器	277	15.2.2 使用多语句式表值 UDF 进行 复杂计算	313
14.1 DML 触发器	277	15.3 CLR UDF	315
14.1.1 AFTER 触发器	277	15.3.1 标量 UDF	315
14.1.2 进行事务提交和回滚操作	279	15.3.2 表值 UDF	318
14.1.3 检测对指定列的 UPDATE 或 INSERT 操作	281	15.3.3 聚合 UDF	320
14.1.4 指定 First 和 Last 触发器	284	15.4 修改和删除 UDF	324

性能调整篇

第 16 章 事务处理	326	17.1.1 并发影响	334
16.1 自动事务处理	326	17.1.2 并发控制	335
16.2 显式事务处理	327	17.2 锁管理器的数据锁定	335
16.3 隐式事务处理	328	17.2.1 锁的粒度和层次结构	336
16.4 使用嵌套事务	329	17.2.2 锁的模式	337
16.5 使用事务保存点	332	17.2.3 锁的兼容性	339
第 17 章 并发访问控制	334	17.2.4 锁升级	339
17.1 并发影响和并发控制类型	334	17.3 自定义锁定	341

17.3.1	自定义锁的超时时间	341	18.1.1	查询计划定义的内容	353
17.3.2	使用表级锁提示	342	18.1.2	生成查询计划	355
17.4	使用事务隔离级别	343	18.2	执行计划的缓存与执行	357
17.5	使用行版本的事务隔离级别	345	18.2.1	执行计划的副本和执行上下文	357
17.5.1	快照隔离和行版本控制的 工作原理	345	18.2.2	执行计划的开销管理	358
17.5.2	使用基于行版本控制的隔离 级别	346	18.3	执行计划的重用	359
17.6	处理死锁	349	18.3.1	通过简单参数化提高计划 重用率	359
17.6.1	防止死锁的方法	350	18.3.2	通过强制参数化提高计划 重用率	359
17.6.2	使用 TRY...CATCH 处理死锁	350	18.3.3	使用显式参数化提高计划 重用率	361
第 18 章	查询的优化与执行	353	18.4	执行计划的重新编译	361
18.1	查询的优化	353			

实 战 篇

第 19 章	SQL 查询演练	365	19.5	从分组中取前几行数据	374
19.1	同一时间范围内并发数统计	365	19.5.1	使用联接获取前几行	375
19.1.1	使用子查询	366	19.5.2	使用窗口排名函数获取前几行	376
19.1.2	使用 CTE	367	19.6	取出多列中的非空值	377
19.2	时间段天数统计	368	19.6.1	姓名问题处理	377
19.3	数字范围统计	369	19.6.2	工资问题处理	379
19.3.1	查找剩余空位区间和剩余 空位编号	370	19.7	将数据由行转换为列	380
19.3.2	查找已用货位区间	372			
19.4	地域范围内最大数统计	373			

1

第 1 部分 基础篇

-
- 第 1 章 SQL 简介
 - 第 2 章 数据库管理
 - 第 3 章 表管理
 - 第 4 章 索引管理
 - 第 5 章 基本查询
 - 第 6 章 子查询
 - 第 7 章 联接和 APPLY 运算符
 - 第 8 章 操作结果集
 - 第 9 章 窗口计算和表旋转
 - 第 10 章 数据修改



第 1 章 SQL 简介

SQL 的全称是结构化查询语言 (Structured Query Language)，这是一种非常易读的语言，只要稍微有一点英语基础，一些简单的数据查询、操作语句几乎都可以理解。但是，要想精通 SQL，也并不是一件很容易的事情，因为在数据处理方面有许许多多的个案。要处理好这些个案，除了具有缜密的逻辑思维，还需要多练习和实践，从而增强记忆。从本章开始，打好坚实的基础，并在此基础上不断提升自己的理论知识体系，当感受某个成功喜悦的时候，或许就会发现自己已经站在了 SQL 的巅峰。

在本章我们将讲述一些最基本的 SQL 内容，如 SQL 的历史起源、ANSI 是什么，以及 SQL 的语法元素和执行 SQL 的工具等。对于基本知识，学习起来可能比较枯燥。但是，如果你是初学 SQL，这些基本知识对于学习好本书的后续内容，将起着至关重要的作用。

1.1 SQL 的历史起源

在 20 世纪 60 年代，网状数据库系统（如 CODASYL）和分层数据库系统（如 IMSTM）是用于自动化银行业务、记账和订单处理系统的一流技术，这些系统是由于商业大型计算机的引入才启用的。而 SQL 是在 70 年代创建的一种基于关系数据库管理系统（Relational Database Management System, RDBMS）模型的数据查询、操作语言。

1.1.1 CODASYL

CODASYL 是美国数据系统语言协会（Conference on Data System Language）的英文缩写，该协会成立于 1957 年，主要目的是为了开发一种用于创建商业应用的通用语言。1959 年 5 月 28 日该协会召开了首次会议，就语言开发进行讨论。这个语言实际上就是 Cobol 语言。

1963 年 6 月 10 日，加利福尼亚州的系统开发公司（System Development Corporation）举办了一个题为“基于计算机的数据库开发和管理”（Development and Management of a Computer-centered Data Base）的研讨会，首次提出并定义了数据库（Database）术语，即：一组文件（表）的集合，其中文件是数据项（行）的有序集合，而每个数据项由数据以及一个或多个键组成。

1965 年，CODASYL 成立了“列表处理任务组”（List Processing Task Force），后更名为“数据

库任务组”(Data Base Task Group)。1971 年 4 月任务组发布了一份重要的报告, 报告概述了网状数据模型, 被称为 CODASYL 或 DBTG (即 Data Base Task Group 的缩写) 数据模型。这个模型定义了数据库的几个关键概念, 包括定义模式的语法、定义子模式的语法和数据操作语言。

1.1.2 IMS

IMS 是信息管理系统 (Information Management System) 的英文缩写, 是 IBM 公司的产品, 这是一款分层数据库管理和事务处理系统。

IMS 最初的开发目的是为了支持美国的阿波罗太空计划。1966 年, IBM 公司的 12 名成员、美国洛克维尔公司 (American Rockwell) 的 10 名成员和卡特彼勒公司 (Caterpillar Tractor) 的 3 名成员被集合起来, 开始开发信息控制系统 (Information Control System, ICS) 和数据语言/接口 (Data Language/Interface, DL/I), 用于辅助跟踪建造太空船所需要的材料。其中, ICS 部分是用于存储和获取数据的数据库部分, 而 DL/I 部分则是用来与之交互的查询语言。

在开发过程中, IBM 小组转移到加利福尼亚州南部的洛杉矶, 并增加至 21 名开发人员。1967 年, IBM 团队完成了 ICS 的第一个版本。1968 年 4 月开始进行安装测试。1968 年 8 月 14 日, 第 1 个安装成功信息显示在美国国家航空航天局洛克维尔部门的 IBM 2740 打印机终端上。

1969 年, ICS 被更名为信息管理系统, 即 IMS。自第一个版本问世以来, IBM 一直在不断开发和完善 IMS 的功能。2007 年, IBM 推出了 IMS 10 版本。IMS 10 具备了增强的 XML 和网络服务功能, 并且也是第一个将标准 XML 查询语言应用于层次结构化数据的数据库系统。

1.1.3 RDBMS 和 SQL

无论 CODASYL, 还是 IMS, 虽然这些系统为早期系统提供了良好的基础, 但它们的基本体系结构将数据的物理操作与逻辑操作混合在一起。当数据的物理位置改变时, 也必须更新应用程序来引用新的位置, 给数据查询带来了不便。

SQL 是一种关系型数据库查询语言, 要介绍 SQL 的起源, 就不得不介绍 IBM 公司的两个重量级人物——E.F. Codd 博士和 Don Chamberlin 博士。E. F. Codd 博士最早提出了关系数据库管理系统 (Relational Database Management System, RDBMS) 模型, 而 Don Chamberlin 博士则是 SQL 和 XQuery 语言的主要创造者之一。他们对数据库的变革起到了革命性的作用。

Don Chamberlin 最初在 IBM 的 T.J.Watson 研究中心 (<http://www.watson.ibm.com/>) 工作, 当时该中心的主要研究方向是操作系统。Chamberlin 一开始从事的项目是 System A, 但项目很快便以失败而告终。当时担任项目经理的 Leonard Liu (现为 Augmentum 公司 CEO, <http://www.augmentum.com>) 很有远见地预见到数据库的美好前景, 他转变了整个小组的研究方向。Chamberlin 因此在数据库软件和查询语言方面进行了大量研究, 并成为了小组中最好的网状数据库

CODASYL 专家。

此时的 E. F. Codd 博士在 IBM 的 San Jose 研究中心（即现在的 Almaden 研究中心，<http://www.almaden.ibm.com/>）工作，1970 年 6 月，他发表了里程碑性的论文《大型共享数据库数据的关系模型》，确立了关系数据库的概念。但是，由于 IBM 正在从事 IMS 的开发，这种思想对 IBM 本身产品造成了威胁，所以公司内部最初持压制态度。当然这也与 Codd 采用了大量的数学方法，不容易理解有关。

1973 年，IBM 在外部竞争压力下，开始加强在关系数据库方面的投入。Chamberlin 被调到 San Jose 研究中心，加入新成立的项目 System R。System R 基于 Codd 提出的关系数据库管理系统模型。

System R 项目包括研究高层的关系数据系统（Relational Data System, RDS）和研究底层的存储系统（Research Storage System, RSS）两个小组，Chamberlin 担任 RDS 组的经理。RDS 实际上就是一个数据库语言编译器，由于 Codd 提出的关系代数和关系演算过于数学化，影响了易用性。于是 Chamberlin 选择了自然语言作为研究方向，其结果就是诞生了结构化英语查询语言（Structured English Query Language, SEQUEL）。后来，由于商标之争，SEQUEL 更名为 SQL。

System R 是一个具有开创性意义的项目。它第一次实现了结构化查询语言，并具以成为标准的关系数据查询语言。同时，它也是第一个证明了关系数据库管理系统可以提供良好事务处理性能的系统。System R 系统中的设计决策，以及一些基本算法选择（如查询优化中的动态编程算法）对以后的关系系统都产生了积极影响。

System R 本身作为原型虽然并未问世，但鉴于其影响，计算机协会（Association for Computing Machinery, ACM, <http://www.acm.org/>）还是把 1988 年的“软件系统奖”授予了 System R 开发小组。

1.1.4 ANSI 和 SQL 方言

ANSI 是美国国家标准学会（American National Standards Institute）的英文简称，成立于 1918 年。当时，美国的许多企业和专业技术团体，已开始了标准化工作，但因彼此间没有协调，存在不少矛盾和问题。为了进一步提高效率，数百个科技学会、协会组织和团体，均认为有必要成立一个专门的标准化机构，并制定统一的通用标准。1918 年，美国材料试验协会（ASTM）、美国机械工程师协会（ASME）、美国矿业与冶金工程师协会（ASMME）、美国土木工程师协会（ASCE）和美国电气工程师协会（AIEE）等组织，共同成立了美国工程标准委员会（AESC）。美国政府的商务部、陆军部和海军部也参与了该委员会的筹备工作。1928 年，美国工程标准委员会改组为美国标准协会（ASA）。为致力于国际标准化事业和消费品方面的标准化，1966 年 8 月，又改组为美利坚合众国标准学会（USASI）。1969 年 10 月 6 日改成现名：美国国家标准学会（ANSI）。

虽然 IBM 首创了关系数据库理论，但 Oracle 却是第一家在市场上推出了这套技术的公司。随着时间的推移，SQL 的简洁、直观，在市场上获得了不错的反响，从而引起了 ANSI 的关注，分别在 1986 年、1989 年、1992 年、1999 年及 2003 年发布了 SQL 标准。SQL Server 2000 遵循 ANSI

SQL:1992 标准, 而 SQL Server 2005 和 2008 还实现了 ANSI SQL:1999 和 ANSI SQL:2003 中的一些重要特性。

数据库生产商在遵循 ANSI 标准的同时, 也会根据自己产品的特点对 SQL 进行了一些改进和增强, 于是也就有了 SQL Server 的 Transact-SQL、Oracle 的 PL/SQL 等语言, 我们称之为 SQL 方言。在本书中, 我们将以 Transact-SQL 为基础进行 SQL 语言的介绍。实际上, 在学习过程中, 大家也没有必要刻意关心哪些语句或关键字是 SQL 标准, 哪些是 Transact-SQL 的扩展。其实常见的数据库操作, 在绝大多数支持 SQL 语言的数据库中差别并不大, 所以数据库开发人员在跨越不同的数据库产品时, 一般不会遇到什么障碍。但是对于数据库管理员来说, 则需要面对很多挑战, 不同数据库产品在管理、维护和性能调整方面区别很大。

1.2 Transact-SQL 语言的类型

在介绍了 SQL 的起源后, 来看一下 Transact-SQL 包括哪些语言类型。首先, 为了遵循 ANSI SQL 标准, Transact-SQL 提供了数据定义语言(Data Definition language, DDL)语句和数据操作语言(Data Manipulation Language, DML)语句; 其次, 为了增强灵活性, Transact-SQL 还提供了用于编程的流控制语句和其他语句。

对于语言类型, 读者仅作大致了解就可以。在实际应用中, 就像没必要区分哪些是 SQL 标准, 哪些是 SQL 扩展一样, 也没有必要区分 DDL 和 DML, 它们是一个协同工作的整体。

1.2.1 DDL 语句

DDL 语句用于创建数据库对象, 如表、视图、索引等, 表 1-1 中列出了一些常用的 DDL 语句。

表 1-1 DDL 常用语句

语 句	作 用
ALTER DATABASE	修改一个数据库或与该数据库关联的文件和文件组
ALTER FUNCTION	更改用户定义函数
ALTER LOGIN	更改 Microsoft Windows 或数据库服务器登录账户的属性
ALTER PROCEDURE	修改存储过程
ALTER SCHEMA	更改架构所有者
ALTER TABLE	通过修改、添加或删除列和约束来修改表定义
ALTER TRIGGER	更改 DML 或 DDL 触发器的定义
ALTER VIEW	修改先前创建的视图
CREATE DATABASE	创建新的数据库和用来存储数据库的文件
CREATE FUNCTION	创建用户定义函数
CREATE INDEX	为指定表或视图创建关系索引, 或为指定表创建 XML 索引

续表

语 句	作 用
CREATE LOGIN	创建新的 Microsoft Windows 或数据库服务器登录账户
CREATE PROCEDURE	创建存储过程
CREATE RULE	创建规则对象
CREATE SCHEMA	在当前数据库中创建架构
CREATE TABLE	创建新表
CREATE TRIGGER	创建 DML 或 DDL 触发器
CREATE VIEW	创建一个虚拟表（视图），该表以一种备用方式提供一个或多个表中的数据
DROP DATABASE	从数据库服务器实例中删除一个或多个数据库或数据库快照
DROP FUNCTION	从当前数据库中删除一个或多个用户定义函数
DROP INDEX	从当前数据库中删除索引
DROP LOGIN	删除 Microsoft Windows 或数据库服务器登录账户
DROP RULE	从当前数据库中删除一个或多个用户定义的规则
DROP SCHEMA	从数据库中删除架构
DROP TABLE	删除表的表定义和所有数据、索引、约束和权限规范
DROP TRIGGER	从当前数据库中删除一个或多个 DML 或 DDL 触发器
DROP VIEW	从当前数据库中删除一个或多个视图

1.2.2 DML 语句

DML 语句用来检索和修改数据库的内容，表 1-2 中列出了一些常用的 DML 语句。

表 1-2 DML 常用语句

语 句	作 用
BEGIN TRANSACTION	显式开始一个本地事务
CLOSE	释放当前结果集，然后解除定位游标的行上的游标锁定，从而关闭一个开放的游标
COMMIT	结束事务成功并提交
DELETE	从表或视图中删除行
INSERT	添加新行到表或视图中
READTEXT	从 text、ntext 或 image 列读取 text、ntext 或 image 值，从指定的偏移量开始读取指定的字节数
ROLLBACK	回滚事务
SAVE TRANSACTION	在事务内设置保存点
SELECT	从数据库中检索行，并允许从一个或多个表中选择一个或多个行或列

续表

语 句	作 用
TRUNCATE TABLE	删除表中的所有行，而不记录单个行删除操作
UPDATE	更改表或视图中的现有数据
UPDATETEXT	更新现有的 text、ntext 或 image 字段。使用 UPDATETEXT 可以只更改 text、ntext 或 image 列的一部分

1.2.3 编程和流控制语句

许多人在学习编程时，经常询问编程工具的好坏，那种语言好学，或是在开发上更具备优势。实际上，编程工具可能还有优劣之分，但是编程语言之间差别并不大。尤其是像 SQL 这样的数据处理语言，只要具备了顺序语句、判断语句和循环语句，就可以处理所有业务逻辑问题。表 1-3 中列出了一些常用的编程和流控制语句。

表 1-3

DDL 常用语句

语 句	作 用
BEGIN...END	中间可以包括一系列 SQL 语句，从而提供执行一组 SQL 语句的能力
BREAK	退出 WHILE 或 IF...ELSE 语句中最里面的循环。执行出现在 END 关键字后面的任何语句，END 关键字为循环结束标记
EXECUTE	执行批中的命令字符串、字符串，以及执行系统存储过程、用户定义存储过程、标量值用户定义函数或扩展存储过程
FETCH	通过服务器游标检索特定行
GOTO	将执行流更改到标签处
CONTINUE	在 CONTINUE 关键字之后的任何语句都将被忽略，并继续执行 WHILE 循环
DEALLOCATE	删除游标引用
DECLARE @local_variable	声明变量。变量名称必须使用@前缀
DECLARE CURSOR	定义服务器游标的属性，如游标的滚动行为和用于生成游标所操作的结果集的查询
IF...ELSE	指定语句的执行条件
OPEN	打开服务器游标，然后通过执行在 DECLARE CURSOR 或 SET cursor_variable 语句中指定的 SQL 语句填充游标
PRINT	向客户端返回用户定义消息
RAISERROR	生成错误消息并启动会话的错误处理
RETURN	从查询或过程中无条件退出
SET @local_variable	将先前使用 DECLARE @local_variable 语句创建的局部变量设置为指定值
SELECT @local_variable	将先前使用 DECLARE @local_variable 创建的局部变量设置为指定表达式
WAITFOR	在达到指定时间或时间间隔之前，阻止执行批处理、存储过程或事务
WHILE	设置重复执行 SQL 语句或语句块的条件。只要指定的条件为真，就重复执行语句。可以使用 BREAK 和 CONTINUE 关键字在循环内部控制 WHILE 循环中语句的执行

下面将对一些常用的编程语句进行介绍。

1. IF...ELSE 语句

当条件为“真”时，将执行 IF 关键字后面的语句，当条件为“假”时，将执行 ELSE 后面的语句。例如，下面示例的含义是：如果变量@i > 0，设置@MyVar1 = 100，否则设置@MyVar1 = 10，最后使用 PRINT 语句输出@MyVar1 的值。其中的“--”是注释字符。

```
DECLARE @i AS int, @MyVar1 AS int -- 声明变量，详细信息可以参考 3.3 节的介绍
SET @MyVar1 = 0 -- 设置变量值
SET @i = 1

IF @i > 0
    SET @MyVar1 = 100
ELSE
    SET @MyVar1 = 10
PRINT CONVERT(char(12),@MyVar1) --输出@MyVar1 变量的值
```

2. BEGIN...END

BEGIN...END 提供了执行一组 SQL 语句的方法，此语句对编写 IF...ELSE 和 WHILE 循环非常有用。现在将上面示例中 IF...ELSE 语句改写为下列形式：

```
IF @i > 0
    SET @MyVar1 = 100
ELSE
    BEGIN
        SET @MyVar1 = 10
        PRINT CONVERT(char(12),@MyVar1) --输出@MyVar1 变量的值
    END
```

此时 IF...ELSE 语句的含义是：如果@i > 0，设置@MyVar1 = 100，否则设置@MyVar1 = 10，并使用 PRINT 语句输出@MyVar1 的值。也就是说，SET @MyVar1 = 10 和 PRINT 语句此时都属于 ELSE 部分。

3. WHILE 循环

在 WHILE 关键字后面，可以编写一个控制循环执行的条件。在 WHILE 语句循环体的每一次执行前，都要测试条件。如果条件为真，则执行循环体；否则，将执行循环体后面的语句。

例如，下面的示例中使用了循环和判断语句。首先声明 3 个变量，@MyVar1 用于累加 1~100 之间的奇数值，@MyVar2 用于累加 1~100 之间的偶数值。

```
DECLARE @i AS int, @MyVar1 AS int, @MyVar2 AS int -- 声明变量
SET @MyVar1 = 0 -- 设置变量值
SET @MyVar2 = 0
SET @i = 1

WHILE @i < 100 -- 开始循环
BEGIN -- 指定包含在循环中的语句
    IF @i % 2 <> 0 -- 判断是否为奇数
        SET @MyVar1 = @MyVar1 + @i -- 累加奇数
```

```

ELSE
    SET @MyVar2 = @MyVar2 + @i -- 累加偶数
SET @i = @i + 1 -- 递增@i
END -- 结束循环

SELECT @MyVar1, @MyVar2, @i -- 显示变量值

```

4. GOTO 语句和标签

GOTO 语句用于将执行流更改到标签处，虽然 Transact-SQL 和 PL/SQL 都提供了该语句，但是作为编程而言，我们不推荐使用此编程技术。要编写一个标签，应当在标识符后面加一个冒号。例如，下面的示例使用 GOTO 语句代替 WHILE 循环，与上面的 WHILE 循环示例功能相同。

```

DECLARE @i AS int, @MyVar1 AS int, @MyVar2 AS int -- 声明变量
SET @MyVar1 = 0 -- 设置变量值
SET @MyVar2 = 0
SET @i = 1

table_loop: -- 指定标签
    IF @i % 2 <> 0 -- 判断是否为奇数
        SET @MyVar1 = @MyVar1 + @i -- 累加奇数
    ELSE
        SET @MyVar2 = @MyVar2 + @i -- 累加偶数
    SET @i = @i + 1 -- 递增@i
    IF (@i < 100) GOTO table_loop -- 跳转到标签处

SELECT @MyVar1, @MyVar2, @i -- 显示变量值

```

5. 使用 WAITFOR 语句

WAITFOR 语句用于延迟后面语句的执行，可以指定延迟的时间长度或是具体的时间。参考下面的语句：

```

WAITFOR DELAY '00:1:15'
PRINT N'到时间了'

-- 另一种形式
WAITFOR TIME '10:02:10'
PRINT N'到时间了'

```

第 1 个 WAITFOR 语句使用 DELAY 关键字指定在 1 分 15 秒后执行后面的 PRINT 语句，最长延迟时间为 24 小时。第 2 个 WAITFOR 语句使用了 TIME 关键字，指定在 10 点 2 分 10 秒的时候执行后面的 PRINT 语句。

此语句通常用于并发测试，实际应用中并不多见。例如，可以指定在同一个时间执行多个相同的 SQL 语句。

1.2.4 SQL 语句的批处理

应用程序可以将多个 SQL 语句作为一个批发送到服务器，然后服务器将该批中的语句编译成一个执行计划。在 SQL Server 的 Management Studio 工具中，可以使用 GO 作为批发送的分隔符号。

例如，在图 1-1 所示的语句中共包含 3 个批，选中后单击“执行”按钮一起执行。但是由于包含 3 个 GO，所以会被分批发送，其中第 1 和第 2 个批都能正常执行，第 3 个批中由于最后一行 INSERT 语句的 VALUSE 关键字错了（正确的应当为 VALUES），这个批在将这 3 个 INSERT 语句编译成一个执行计划时，将发生错误。因此，这 3 条 INSERT 语句都不会被执行，而不是仅仅发生错误的 INSERT 语句。从这个示例可以看出，理解批处理 Transact-SQL 语句的执行方式很重要。

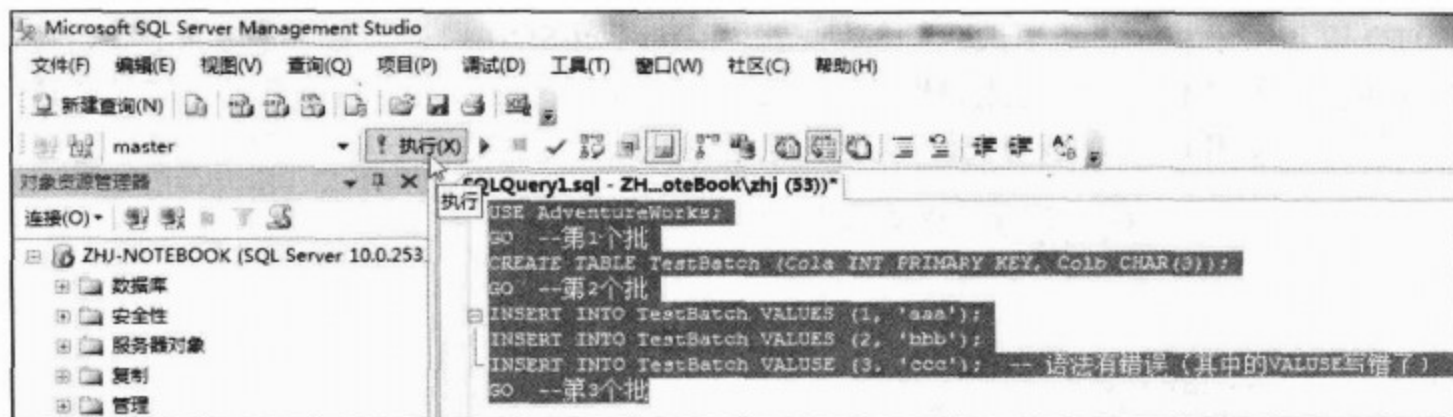


图 1-1 分批执行 SQL 语句

1.3 Transact-SQL 语法

Transact-SQL 具有一些大多数语句都使用或受之影响的元素，包括标识符、数据类型、函数、表达式、运算符和保留关键字等。

1.3.1 使用标识符进行对象引用

数据库对象的名称即为其标识符，如服务器、数据库和数据库对象（如表、视图、列、索引、触发器、过程、约束及规则等）都可以有标识符。对象标识符是在定义对象时创建的，创建完成后便可以使用标识符引用该对象。例如，下列语句创建一个名为 TableX 的表，其中包含 KeyCol 和 Description 列，则 TableX、KeyCol 和 Description 都是标识符。

```
CREATE TABLE TableX (KeyCol INT PRIMARY KEY, Description nvarchar(80))
```

无论是 TableX、还是 KeyCol 和 Description，这些中间无空格的字符，都称为常规标识符。常规标识符格式规则取决于数据库兼容级别（可以使用 sp_dbcmtlevel 存储过程设置该级别）。当兼容级别为 90（SQL Server 2005）或 100（SQL Server 2008）时，常规标识符使用下列规则：

第 1 个字符必须是下列字符之一。

- Unicode 标准 3.2 所定义的字母。Unicode 中定义的字母包括拉丁字母 a~z 和 A~Z，以及来自其他语言的字母字符。
- 下划线（_）、at 符号（@）或者数字符号（#）。
- 在 SQL Server 中，某些位于标识符开头位置的符号具有特殊意义。以 at 符号开头的标识

符表示局部变量或参数。以一个数字符号开头的标识符表示临时表或过程。以两个数字符号 (##) 开头的标识符表示全局临时对象。

● 某些 SQL 函数的名称以两个 at 符号 (@@) 开头。为了避免与这些函数混淆, 不应使用以 “@@” 开头的名称。

后续字符可以包括:

- 如 Unicode 标准 3.2 中所定义的字母。
- 基本拉丁字符或其他国家/地区字符中的十进制数字。
- at 符号、美元符号 (\$)、数字符号或下划线。

此外, 常规标识符不能是 SQL 保留字, 不允许嵌入空格或其他特殊字符, 不允许使用 Unicode 标准之外的增补字符。

如果标识符中必须使用空格 (如 My Table) 或其他不符合常规标识符规则的字符, 则必须包含在双引号 (") 或者方括号 ([]) 内, 否则无法正确识别它们。双引号和方括号被称为分隔标识符。例如, 下面语句中的 My Table 和 order 必须包含在分隔标识符内, 因为 My Table 中间有空格, order 是 SQL Server 用于 ORDER BY 子句的保留字。

```
SELECT * FROM [My Table] WHERE [order] = 10
```

常规标识符和分隔标识符包含的字符数必须在 1~128 之间。对于本地临时表, 标识符最多可以有 116 个字符。

在使用双引号作为分隔符时, 服务器遵从的规则受 SET QUOTED_IDENTIFIER 设置影响。设置为 ON (默认值) 时, 双引号只能分隔标识符, 文字必须由单引号分隔; 设置为 OFF 时, 标识符不能加引号, 且必须符合所有常规标识符规则。

如果字符串中已经包含有单引号, 则应在该单引号前再添加一个单引号。例如, 下面的语句用于从 My Table 表中查找 Last Name 为 O' Brien 的行。

```
SELECT * FROM "My Table"
WHERE "Last Name" = 'O''Brien'
```

1.3.2 设置对象的数据类型

大多数 SQL 语句并不显式引用数据类型, 但是, 由于语句中所引用对象的数据类型间的交互作用, 语句的返回结果会受到影响。下列对象具有数据类型:

- 表和视图中的列。
- 存储过程中的参数。
- 变量。
- 返回一个或多个特定数据类型数据值的 Transact-SQL 函数。

- 具有返回代码（始终为 integer 数据类型）的存储过程。

为对象分配数据类型时可以为对象定义 4 个属性：

- 对象包含的数据种类。
- 所存储值的长度或大小。
- 数值的精度（仅适用于数字数据类型）。
- 数值的小数位数（仅适用于数字数据类型）。

1. 基本类型

数据类型大体可分为精确数字、近似数字、日期和时间、字符串、Unicode 字符串、二进制字符串和其他数据类型 7 种类别，详细信息如表 1-4 所示。

表 1-4 数据类型

类 别	数 据 类 型	范 围	存 储
精确数字	bigint	-2^{63} ($-9\ 223\ 372\ 036\ 854\ 775\ 808$) 到 $2^{63}-1$ ($9\ 223\ 372\ 036\ 854\ 775\ 807$)	8 字节
	int	-2^{31} ($-2\ 147\ 483\ 648$) 到 $2^{31}-1$ ($2\ 147\ 483\ 647$)	4 字节
	smallint	-2^{15} ($-32\ 768$) 到 $2^{15}-1$ ($32\ 767$)	2 字节
	tinyint	0~255	1 字节
	bit	如果表中的列为 8 bit 或更少，则这些列作为 1 个字节存储。如果列为 9~16 bit，则这些列作为 2 个字节存储，依次类推	
	decimal[(p[,s])] 和 numeric[(p[,s])]	带固定精度和小数位数的数值数据类型。使用最大精度时，有效值从 $-10^{38}+1$ 到 $10^{38}-1$ 。numeric 在功能上等价于 decimal p (精度)。最多可以存储的十进制数字的总位数，包括小数点左边和右边的位数。必须是从 1~38 之间的值，默认精度为 18	2~17 个字节（取决于精度。每增加 2 位小数，需要增加
	money	-922 337 203 685 477.5808 到 922 337 203 685 477.5807	8 字节
	smallmoney	-214 748.3648 到 214 748.3647	4 字节
近似数字	float[(n)]	$-1.79E+308$ 至 $-2.23E-308$ 、0 以及 $2.23E-308$ 至 $1.79E+308$ 。其中 n 为用于存储 float 数值尾数的位数，以科学记数法表示，因此可以确定精度和存储大小。如果指定了 n，则它必须是介于 1 和 53 之间的某个值。n 的默认值为 53	取决于 n 的值
	real	$-3.40E+38$ 至 $-1.18E-38$ 、0 以及 $1.18E-38$ 至 $3.40E+38$ ，精度为小数点后 1~7 位	4 字节
日期和时间	datetime	1753 年 1 月 1 日到 9999 年 12 月 31 日，精确度 3.33 毫秒	8 字节
	smalldatetime	1900 年 1 月 1 日到 2079 年 6 月 6 日，精确度 1 分钟	4 字节
	date	0001-01-01 到 9999-12-31，精确度 1 天	3 字节
	time	00:00:00.0000000 到 23:59:59.9999999，精确度 100 纳秒	5 字节
	datetime2	日期范围 0001-01-01 到 9999-12-31，时间范围 00:00:00 到 23:59:59.9999999，精确度 100 纳秒	6~8 字节

续表

类 别	数 据 类 型	范 围	存 储
日期和时间	datetimeoffset	定义一个与采用 24 小时制并可识别时区的一日内时间相组合的日期, 日期范围 0001-01-01 到 9999-12-31, 时间范围 00:00:00 到 23:59:59.9999999, 精确度 100 纳秒	10 字节
字符串	char[(n)]	固定长度, 非 Unicode 字符数据, 长度为 n 个字节。n 的取值范围为 1~8000	n 个字节
	varchar[(n max)]	可变长度, 非 Unicode 字符数据。n 的取值范围为 1~8000。max 指示最大存储大小是 $2^{31}-1$ 个字节。存储大小是输入数据的实际长度加 2 个字节。所输入数据的长度可以是 0 个字符	实际长度加 2 个字节
	text	服务器代码页中长度可变的非 Unicode 数据, 最大长度为 $2^{31}-1$ (2 147 483 647) 个字符	16 字节+二进制数据所需存储空间大小
Unicode 字符串	nchar[(n)]	n 个字符的固定长度的 Unicode 字符数据。n 值必须在 1 到 4000 之间 (含)	2n 个字节
	nvarchar[(n max)]	可变长度 Unicode 字符数据。n 值在 1~4000 之间 (含)。max 指示最大存储大小为 $2^{31}-1$ 字节。所输入数据的长度可以是 0 个字符	所输入字符个数的两倍+2 个字节
	ntext	长度可变的 Unicode 数据, 最大长度为 $2^{30}-1$ (1 073 741 823) 个字符	所输入字符个数的两倍
二进制字符串	binary[(n)]	长度为 n 字节的固定长度二进制数据, 其中 n 是从 1 到 8000 的值。存储大小为	n 字节
	varbinary[(n max)]	可变长度二进制数据。n 可以取从 1 到 8000 的值。max 指示最大的存储大小为 $2^{31}-1$ 字节。所输入数据的长度可以是 0 字节	所输入数据的实际长度+2 个字节
	image	长度可变的二进制数据, 从 0 到 $2^{31}-1$ (2 147 483 647) 个字节	16 字节+二进制数据所需存储空间大小
其他数据类型	cursor	用于变量或存储过程 OUTPUT 参数的一种数据类型, 该参数包含对一个游标的引用。使用 cursor 数据类型创建的变量可以为空	N/A (仅用于变量)
	sql_variant	用于存储各种数据类型 (不包括 text、ntext、image、timestamp 和 sql_variant) 的值	最大 8016 个字节
	table	用于存储结果集以进行后续处理的特定数据类型。table 主要用于临时存储一组行, 这些行是作为表值函数的结果集返回的	N/A (仅用于变量)
	timestamp	数据库中自动生成的唯一二进制序号	8 字节
	uniqueidentifier	全局唯一标识符 (GUID)	16 字节
	xml	存储 XML 数据的数据类型	最大 2GB

2. 用户自定义数据类型

除了上面介绍的数据类型, 还可以在创建 3 种用户自定义数据类型。

一种是从基本数据类型创建的别名数据类型, 这样做的目的是为了更清楚地说明对象中值的类

型。例如，下面的语句创建了一个基于 `datetime` 的 `birthday` 数据类型，用于在 `employee` 的 `emp_birthday` 列中存储生日数据。

```
-- 创建一个允许 null 的 birthday 数据类型
CREATE TYPE birthday
FROM datetime NULL
GO
-- 创建一个使用新数据类型的表
CREATE TABLE employee
(emp_id char(5), emp_first_name char(30), emp_last_name char(40), emp_birthday birthday)
```

另一种是 CLR 用户定义数据类型，它是在 Microsoft .NET Framework 公共语言运行时 (CLR) 中使用编程方法创建的，这是从 SQL Server 2005 开始提供的一种新功能。此外，包括触发器、存储过程、函数、聚合函数，都可以利用 CLR 提供的丰富的编程模型来扩展服务器的功能。

最后一种是用用户定义表数据类型，也就是说用户可以定义一个表示表结构的数据类型。这是从 SQL Server 2008 开始提供的一种新功能。下面的语句首先创建一个名为 `LocationTableType` 的表数据类型，然后创建一个基于该类型的变量，并向其中插入数据和查询数据。

```
-- 创建一个表数据类型 LocationTableType
CREATE TYPE LocationTableType AS TABLE
(
    FirstName VARCHAR(50),
    LastName VARCHAR(50)
);
GO

-- 创建一个基于 LocationTableType 的变量
DECLARE @MyTable AS LocationTableType;
-- 向变量中插入数据行
INSERT INTO @MyTable VALUES('Ken','Levy');
INSERT INTO @MyTable VALUES('Sara','Ford');
-- 查询数据
SELECT * FROM @MyTable;
GO
```

3. 数据类型的隐式转换

当两个不同数据类型的表达式用运算符组合后，数据类型优先级规则指定将优先级较低的数据类型转换为优先级较高的数据类型。如果此转换不是所支持的隐式转换，则返回错误。当两个操作数表达式具有相同的数据类型时，运算的结果便为该数据类型。优先级顺序是：用户定义数据类型（最高）、`sql_variant`、`xml`、`datetime`、`smalldatetime`、`float`、`real`、`decimal`、`money`、`smallmoney`、`bigint`、`int`、`smallint`、`tinyint`、`bit`、`ntext`、`text`、`image`、`timestamp`、`uniqueidentifier`、`nvarchar`、`nchar`、`varchar`、`char`、`varbinary`、`binary`（最低）。

1.3.3 函数

与其他程序设计语言中的函数相似，SQL Server 的函数可以有零个、一个或多个参数，并返回一个标量值或表格形式的值的集合。

1.3.4 使用表达式求值

表达式是标识符、值和运算符的组合，可以对其求值以获取结果。访问或更改数据时，可在多个不同的位置使用数据。例如，可以将表达式用作要在查询中检索的数据的一部分，也可以用作查找满足一组条件的数据时的搜索条件。

表达式可以是下列任何一种形式：

- 常量；
- 函数；
- 列名；
- 变量；
- 子查询；
- CASE、NULLIF 或 COALESCE。

1.3.5 使用运算符进行计算操作

运算符是表达式的组成部分之一，可以使用运算符执行算术、比较、串联或赋值操作。例如，表达式 `PriceColumn * 1.1` 中的乘号 (*) 使价格提高 10%。

1.3.6 使用注释符添加 SQL 语句说明

注释是程序代码中不执行的文本字符串，也称为备注。注释可用于对代码进行说明或暂时禁用正在进行诊断的部分、SQL 语句和批（批是指一起发送到服务器的多条 SQL 语句，SQL Server 是按批进行优化的）。使用注释对代码进行说明，便于将来对程序代码进行维护。

● --（双连字符）。该注释字符可与要执行的代码处在同一行，也可另起一行。从双连字符开始到行尾的内容均为注释。对于多行注释，必须在每个注释行的前面使用双连字符。

● /* ... */（正斜杠-星号字符对）。这些注释字符可与要执行的代码处在同一行，也可另起一行，甚至可以在可执行代码内部。开始注释对 (/*) 与结束注释对 (*/) 之间的所有内容均视为注释。对于多行注释，必须使用开始注释字符对 (/*) 来开始注释，并使用结束注释字符对 (*/) 来结束注释。批中的注释没有最大长度限制。

下面是一些有效注释的示例。

```
USE AdventureWorks;  
GO  
-- 这是我的单行注释  
  
SELECT EmployeeID, Title
```

```

FROM HumanResources.Employee
GO

/* 这是多行注释的第 1 行，
   这是多行注释的第 2 行。*/
SELECT Name, ProductNumber, Color
FROM Production.Product
GO

-- 在一个 SQL 语句中添加注释
SELECT ContactID, /* FirstName, */ LastName
FROM Person.Contact

-- 在代码行后使用注释
USE AdventureWorks;
GO
UPDATE Production.Product
SET ListPrice = ListPrice * .9 -- 我在代码行使用注释
GO

```

1.3.7 保留关键字

可以说，任何数据库产品，都会保留某些关键字供自己专用。例如，在 OSQL 或 SQL Server 代码编辑器会话中使用 DUMP 关键字或 BACKUP 关键字，即表示让 SQL Server 备份全部或部分数据库或者备份日志。

除数据库服务器所限定的位置以外使用保留关键字，均为非法。数据库中对象的名称不能与保留关键字相同。如果有这样的名称，则必须始终使用带分隔符的标识符来引用这个对象。尽管这个方法允许存在名称为保留关键字的对象，还是建议不要用与保留关键字相同的名称命名任何数据库对象。

1.4 常量和变量

常量和变量是程序设计过程中必不可少的元素，在前面的章节中也涉及了变量的介绍。

1.4.1 常量的类型

常量，也称为文字值或标量值，是表示一个特定数据值的符号。常量的格式取决于它所表示的值的类型。

1. 字符串常量

字符串常量包含在单引号内，可以由字母数字字符（a~z、A~Z 和 0~9）以及特殊字符（如!、@和#）组成。例如：

```
! '这是我的字符串常量'
```


如果单引号中的字符串包含一个嵌入的引号，可以使用两个单引号表示嵌入的单引号。对于嵌入在双引号中的字符串则没有必要这样做。例如，下面是“I'm Tom”常量的正确书写方式。

```
| 'I' 'm Tom'
```

空字符串用中间没有任何字符的两个单引号表示。

2. Unicode 字符串常量

Unicode 字符串的格式与普通字符串相似，但它前面有一个 N 标识符（N 代表 SQL-92 标准中的区域语言）。N 前缀必须是大写字母。例如，'Michel'是字符串常量而 N'Michel'则是 Unicode 常量。

Unicode 常量被解释为 Unicode 格式数据，并且不使用代码页进行计算。对于字符数据，存储 Unicode 数据时每个字符使用 2 个字节，而不是每个字符 1 个字节。

3. 二进制常量

二进制常量具有前缀 0x，并且是十六进制数字字符串。这些常量不使用引号括起。例如，下面是二进制字符串的示例：

```
| 0xAE  
| 0x12Ef  
| 0x69048AEFDD010E  
| 0x （空二进制常量）
```

4. bit 常量

bit 常量使用数字 0 或 1 表示，并且不括在引号中。如果使用一个大于 1 的数字，则该数字将转换为 1。

5. datetime 常量

datetime 常量使用特定格式的字符日期值来表示，并被单引号括起来。

- 字母日期，如'April 15, 1998'。
- 数值日期格式，如'4/15/1998'。
- 未分隔的字符串格式，如'19981207'指 1998 年 12 月 7 日。

下面分别是使用 24 小时制和 12 小时制方式表示的时间常量：

```
| '14:30:24'  
| '04:24 PM'
```

被支持的所有时间格式，可以参考 1.6.6 小节的表 1-18。

6. integer 常量

integer 常量以没有用引号括起来并且不包含小数点的数字字符串来表示。integer 常量必须全

部为数字，并且不能包含小数。如：

```
1894
2
```

7. decimal 常量

decimal 常量由没有用引号括起来并且包含小数点的数字字符串来表示。如：

```
1894.1204
2.0
```

8. float 和 real 常量

float 和 **real** 常量使用科学记数法来表示。如：

```
101.5E5
0.5E-2
```

9. money 常量

money 常量是以可选的货币符号作为前缀的一串数字。**money** 常量可以包含小数点，但是不能使用引号括起来。如：

```
$12
$542023.14
```

10. uniqueidentifier 常量

uniqueidentifier 常量是表示 GUID 的字符串。可以使用字符或二进制字符串格式指定。例如，下面的示例指定的是相同的 GUID：

```
'6F9619FF-8B86-D011-B42D-00C04FC964FF'
0xff19966f868b11d0b42d00c04fc964ff
```

11. 指定负数和正数

可以在数字前面添加“+”和“-”一元运算符来表示正数和负数。如果没有添加一元运算符，则默认为正数。如：

```
-- integer 表达式
-2147483648
-- decimal 表达式
+145345234.2234
-2147483648.10
-- float 表达式
-12E5
-- money 表达式
-$45.56
```

1.4.2 变量的类型

变量对应内存中的一个存储空间。与常量不同，变量的值在运行过程中可以随时改变。

1. 局部变量和全局变量

根据变量的作用范围不同，可以分为局部变量和全局变量。

局部变量是用户在程序中定义的变量，它仅在定义的程序范围内有效。局部变量可以用来保存从表中读取的数据，也可以作为临时变量保存计算的中间结果。在批处理和脚本中的变量通常用于：

- 作为计数器计算循环执行的次数或控制循环执行的次数。
- 保存数据值以供控制流语句测试。
- 保存存储过程返回代码要返回的数据值或函数返回值。

局部变量名称的第1个字符必须为一个@。

在 Microsoft SQL Server 的早期版本中，如果变量以@@开头，则被称为**全局变量**。这些变量实际上是 SQL Server 的系统函数，它们的语法遵循函数的规则。用户可以在程序中使用这些函数测试系统特性和 SQL 命令的执行情况。

2. 声明变量

变量只有在声明后才能使用，可以使用 DECLARE 语句来声明变量。在声明变量时可以指定变量的数据类型和长度。例如，下面的 DECLARE 语句使用 int 数据类型创建名为@mycounter 的局部变量。

```
DECLARE @MyCounter int
```

如果要声明多个局部变量，需要在定义的局部变量后使用一个逗号，然后指定下一个局部变量名称和数据类型。例如，下面的 DECLARE 语句创建 3 个局部变量，名称分别为@last_name、@fname 和@salary。

```
DECLARE @LastName nvarchar(30), @FirstName nvarchar(20), @Salary decimal(7,2)
```

在使用变量时，需要注意变量的作用域。变量具有局部作用域，只在定义它们的批处理或过程中可见。作用域范围从声明变量的地方开始到声明变量的批处理或存储过程的结尾。例如，下面的脚本存在语法错误，因为在一个批处理中引用了在另一个批处理中声明的变量。

```
USE AdventureWorks -- 指定使用的数据库
GO
DECLARE @MyVariable int
SET @MyVariable = 1
GO -- 该语句将结束批
-- @MyVariable 已经超出了范围并不再存在

-- 下面的 SELECT 语句会发生语法错误，因为它引用了一个不存在的变量
SELECT *
FROM HumanResources.Employee
WHERE EmployeeID = @MyVariable
```


3. 为变量设置值

在声明变量后，变量值被默认设置为 NULL。要为变量赋值，可以使用 SET 或 SELECT 语句。其中，SET 是为变量赋值的首选方法。

仍旧使用上面的示例，将声明的 MyVariable 变量赋值为 1，并在 SELECT 语句的 WHERE 子句中使用该变量。语句如下：

```
USE AdventureWorks -- 指定使用的数据库
GO
DECLARE @MyVariable int
SET @MyVariable = 1 -- 设置变量值为 1

SELECT * FROM HumanResources.Employee
WHERE EmployeeID = @MyVariable
GO
```

变量也可以通过选择列表中当前所引用的值赋值。如果在选择列表中引用变量，则它应当被赋以标量值或者 SELECT 语句应仅返回一行。例如：

```
USE AdventureWorks -- 指定使用的数据库
GO
DECLARE @EmpIDVariable int

SELECT @EmpIDVariable = MAX(EmployeeID)
FROM HumanResources.Employee

SELECT @EmpIDVariable --显示@EmpIDVariable 的值
GO
```

注意：如果在单个 SELECT 语句中有多个赋值子句，服务器并不保证表达式求值的顺序。只有当赋值之间有引用时才能看到影响。

如果 SELECT 语句返回多行而且变量引用一个非标量表达式，则变量被设置为结果集最后一行中表达式的返回值。例如，下面的语句将 @EmpIDVariable 设置为所返回的最后一行的 EmployeeID 字段的值。

```
USE AdventureWorks
GO
DECLARE @EmpIDVariable int

SELECT @EmpIDVariable = EmployeeID
FROM HumanResources.Employee
ORDER BY EmployeeID DESC

SELECT @EmpIDVariable --显示@EmpIDVariable 的值
GO
```

1.5 运算符

运算符是一种符号，用来指定要在一个或多个表达式中执行的操作。运算符可以分为算术运算符、逻辑运算符、赋值运算符、字符串串联运算符、按位运算符、一元运算符和比较运算符，共 6 个类别。

1.5.1 使用算术运算符执行数学运算

算术运算符可以对两个表达式执行数学运算，这两个表达式可以是数值数据类型类别的任何数据类型。可用的算术运算符如表 1-5 所示。

表 1-5 算术运算符

运 算 符	说 明
+	加法
-	减法
*	乘法
/	除法
%	返回一个除法运算的整数余数。例如， $13 \% 5 = 3$
+=	将原始值加上一定的量，并将原始值设置为结果（仅限于 SQL Server 2008）
-=	将原始值减去一定的量，并将原始值设置为结果（仅限于 SQL Server 2008）
*=	将原始值乘上一定的量，并将原始值设置为结果（仅限于 SQL Server 2008）
/=	将原始值除以一定的量，并将原始值设置为结果（仅限于 SQL Server 2008）
%=	将原始值除以一定的量，并将原始值设置为余数（仅限于 SQL Server 2008）

加 (+) 和减 (-) 运算符也可用于对 `datetime` 和 `smalldatetime` 型值执行算术运算。

在进行算术运算时，需要注意计算结果的精度、小数位数和数据类型长度的变化。所谓精度，是指数字中的数字个数。小数位数是数中小数点右边的数字个数。例如，数 123.45 的精度是 5，小数位数是 2。

`numeric` 和 `decimal` 数据类型的默认最大精度为 38，而早期的版本中，默认的最大精度为 28。

数字数据类型的长度是指存储此数所占用的字节数。字符串或 `Unicode` 数据类型的长度是字符个数。`binary`、`varbinary` 和 `image` 数据类型的长度是字节数。例如，`int` 数据类型可以有 10 位数，用 4 个字节存储，不接受小数点。`int` 数据类型的精度是 10，长度是 4，小数位数是 0。

除了 `decimal` 类型之外，数字数据类型的精度和小数位数是固定的。如果算术运算符有两个相同类型的表达式，结果就为该数据类型，并且具有对此类型定义的精度和小数位数。如果运算符有

两个不同数字数据类型的表达式，则由数据类型优先级决定结果的数据类型。

表 1-6 列出了当运算结果是 decimal 类型时，如何计算结果的精度和小数位数。只有在下列任一条件成立时，结果才为 decimal 数据类型。

- 两个表达式都是 decimal 类型。
- 一个表达式是 decimal 类型，而另一个表达式是比 decimal 优先级低的数据类型。

操作数表达式由表达式 e1（精度为 p1，小数位数为 s1）和表达式 e2（精度为 p2，小数位数为 s2）来表示。非 decimal 类型的任何表达式的精度和小数位数，采用该表达式的当前设置。

表 1-6 decimal 类型数据的精度和小数位数的计算方式

运 算	结 果 精 度	结果小数位数
e1 + e2	$\max(s1, s2) + \max(p1-s1, p2-s2) + 1$	$\max(s1, s2)$
e1 - e2	$\max(s1, s2) + \max(p1-s1, p2-s2) + 1$	$\max(s1, s2)$
e1 * e2	$p1 + p2 + 1$	$s1 + s2$
e1 / e2	$p1 - s1 + s2 + \max(6, s1 + p2 + 1)$	$\max(6, s1 + p2 + 1)$

注意：结果精度和小数位数的绝对最大值为 38。当结果精度大于 38 时，相应的小数位数会减少，以避免结果的整数部分被截断。

1.5.2 使用赋值运算符为变量赋值

等号(=)是唯一的赋值运算符。它通常与 SET 语句一起使用，为变量赋值。例如，下面将创建一个 @MyCounter 变量，然后使用赋值运算符为其赋值。

```
DECLARE @MyCounter INT
SET @MyCounter = 1
```

也可以使用赋值运算符在列标题和定义列值的表达式之间建立关系。下面的语句将显示列标题 FirstColumnHeading 和 SecondColumnHeading。在所有行的 FirstColumnHeading 列中均显示字符串“xyz”。然后，在 SecondColumnHeading 列中列出来自 Product 表的每个产品 ID。

```
USE AdventureWorks
GO
SELECT FirstColumnHeading = 'xyz',
       SecondColumnHeading = ProductID
FROM Production.Product;
GO
```

1.5.3 使用位运算符执行按位运算

位运算符在两个表达式之间执行按位运算，这两个表达式可以是整数数据类型类别中的任何数

据类型。可用的位运算符如表 1-7 所示。

表 1-7 位运算符	
运 算 符	说 明
& (位与)	按位进行与运算 (两个操作数)
(位或)	按位进行或运算 (两个操作数)
^ (位异或)	按位进行异或运算 (两个操作数)
&=	对原始值执行位与运算, 并将原始值设置为结果 (仅限于 SQL Server 2008)
^=	对原始值执行位异或运算, 并将原始值设置为结果 (仅限于 SQL Server 2008)
=	对原始值执行位或运算, 并将原始值设置为结果 (仅限于 SQL Server 2008)

位运算符的操作数可以是整数或二进制字符串数据类型类别中的任何数据类型 (image 数据类型除外), 但两个操作数不能同时是二进制字符串数据类型类别中的某种数据类型。表 1-8 列出了所支持的操作数数据类型。

表 1-8 支持位运算的数据类型	
左 操 作 数	右 操 作 数
binary	int、smallint 或 tinyint
bit	int、smallint、tinyint 或 bit
int	int、smallint、tinyint、binary 或 varbinary
smallint	int、smallint、tinyint、binary 或 varbinary
tinyint	int、smallint、tinyint、binary 或 varbinary
varbinary	int、smallint 或 tinyint

对于按位进行与运算、或运算和异或运算的计算规则如表 1-9 所示。

表 1-9 按位进行与运算、或运算和异或运算的计算规则				
位 1	位 2	&运算	运算	^运算
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

1.5.4 使用比较运算符进行大小比较

比较运算符测试两个表达式是否相同。除了 text、ntext 或 image 数据类型的表达式外, 比较运算符可以用于所有的表达式。表 1-10 列出了可用的比较运算符。

表 1-10

比较运算符

运 算 符	说 明
=	等于
>	大于
<	小于
>=	大于或等于
<=	小于或等于
<>	不等于
!=	不等于 (非 SQL-92 标准)
!<	不小于 (非 SQL-92 标准)
!>	不大于 (非 SQL-92 标准)

使用比较运算符的表达式的计算结果为布尔数据类型，它有 3 种值：TRUE、FALSE 和 UNKNOWN。与其他数据类型不同，布尔数据类型不能被指定为表列或变量的数据类型，也不能在结果集中返回。

1.5.5 使用逻辑运算符进行条件测试

逻辑运算符对某些条件进行测试，以获得其真实情况。逻辑运算符和比较运算符一样，返回带有 TRUE 或 FALSE 值的布尔数据类型。表 1-11 列出了可用的逻辑运算符。

表 1-11

逻辑运算符

运 算 符	说 明
ALL	如果一组的比较都为 TRUE，那么就为 TRUE。例如，5>ALL(3, 2, 1)的结果是 TRUE；而 5>ALL(3, 7, 2)的结果是 FALSE，因为 5 小于 7
AND	如果两个布尔表达式都为 TRUE，那么就为 TRUE
ANY	如果一组的比较中任何一个为 TRUE，那么就为 TRUE。例如，5>ANY(3, 6, 9)的结果是 TRUE，因为 3、6 和 9 中有一个小于 5 的数
BETWEEN	如果操作数在某个范围之内，那么就为 TRUE。BETWEEN 通常与 AND 一起使用。例如，5 BETWEEN 3 AND 6 的结果为 TRUE
EXISTS	如果子查询包含一些行，那么就为 TRUE
IN	如果操作数等于表达式列表中的一个，那么就为 TRUE。例如，5 IN (1, 2, 4)的结果是 FALSE，因为 5 没有出现在列表中
LIKE	如果操作数与一种模式相匹配，那么就为 TRUE。LIKE 子句中会使用“%”和“_”等通配符。“_”表示一个字符，“%”表示一个字符串。例如，“_oon”模式表示在“oon”前面有一个字符的字符串。对于表达式 Varx LIKE _oon 来说，当变量 Varx 的值（如 moon、noon）符合“_oon”模式时，则返回 TRUE
NOT	对任何其他布尔运算符的值取反
OR	如果两个布尔表达式中的一个为 TRUE，那么就为 TRUE
SOME	如果在一组比较中，有些为 TRUE，那么就为 TRUE。SOME 与 ANY 的功能相同

1.5.6 字符串串联运算符

加号(+)是字符串串联运算符,可以用它将字符串串联起来。其他所有字符串操作都使用字符串函数(如 SUBSTRING)进行处理。

例如,下面使用加号将“abc”和“efg”串联起来,得到的结果是“abcefg”。

■ 'abc'+'efg'

在进行字符串串联时,需要注意字符串长度的变化:

- 在将两个 char、varchar、binary 或 varbinary 表达式串联时,所生成表达式的长度是两个源表达式长度之和,或是 8000 字符,以二者中少者计算。
- 在将两个 nchar 或 nvarchar 表达式串联时,所生成表达式的长度是两个源表达式长度之和,或是 4000 字符,以二者中少者计算。

1.5.7 一元运算符

一元运算符只对一个表达式执行操作,该表达式可以是数字数据类型类别中的任何一种数据类型。表 1-12 列出了可用的一元运算符。

表 1-12 一元运算符

运 算 符	说 明
+(正)	数值为正
-(负)	数值为负
~(位非)	返回数字的非。~运算符对表达式逐位执行逻辑位非运算,也就是说,如果位是 1,则改成 0; 0 则改成 1

+和-运算符可以用于数字数据类型类别中任一数据类型的任意表达式。~运算符只能用于整数数据类型类别中任一数据类型的表达式。

1.6 常用函数

为满足通常的程序设计需要,Transact-SQL 语言提供了非常丰富的函数,包括聚合函数、配置函数、游标函数、日期和时间函数、数学函数等。本节将介绍一些常用的函数类型。

1.6.1 聚合函数

聚合函数对一组值执行计算并返回单个值,如表 1-13 所示。除了 COUNT 以外,聚合函数都

会忽略空值。聚合函数经常与 SELECT 语句的 GROUP BY 子句一起使用。

表 1-13

聚合函数及其功能

函 数	功 能
AVG([ALL DISTINCT] expression)	返回组中各值的平均值。空值将被忽略，ALL（默认值）指定对所有的值进行聚合函数运算，DISTINCT 指定 AVG 只在每个值的唯一实例上执行，而不管该值出现了多少次，expression 是精确数值或近似数值数据类别（bit 数据类型除外）的表达式
CHECKSUM(* expression [,...n])	返回按照表的某一行或一组表达式计算出来的校验和值，可用于对列进行等价搜索。CHECKSUM 用于生成哈希索引。 *指定对表的所有列进行计算。如果有任一列是非可比数据类型，则 CHECKSUM 返回错误。非可比数据类型为 text、ntext、image 和 cursor。expression 是除非可比数据类型之外的任何类型的表达式
CHECKSUM_AGG ([ALL DISTINCT] expression)	返回组中各值的校验和，可用于检测表中的更改。ALL（默认值）指定对所有的值进行聚合函数运算，DISTINCT 指定 CHECKSUM_AGG 返回唯一校验值，expression 是常量、列或函数以及数字、位运算和字符串运算符的任意组合。expression 是 int 数据类型的表达式
COUNT ({ [[ALL DISTINCT] expression] * })	返回组中的项数，包括 NULL 值和重复项。ALL（默认值）指定对所有的值进行聚合函数运算，DISTINCT 指定 COUNT 返回唯一非空值的数量，expression 是除 text、image 或 ntext 以外任何类型的表达式
COUNT_BIG ({ [ALL DISTINCT] expression } *)	COUNT_BIG 的用法与 COUNT 函数类似。两个函数唯一的差别是 COUNT_BIG 始终返回 bigint 数据类型值，而 COUNT 始终返回 int 数据类型值
GROUPING (column_name)	产生一个附加的列。当行由 CUBE 或 ROLLUP 运算符添加时，该函数将导致附加列的输出值为 1；当行不由 CUBE 或 ROLLUP 运算符添加时，该函数将导致附加列的输出值为 0。column_name 是 GROUP BY 子句中的列，用于测试 CUBE 或 ROLLUP 空值
MAX ([ALL DISTINCT] expression)	返回表达式的最大值。ALL（默认值）指定对所有的值应用此聚合函数，DISTINCT 对于 MAX 无意义，仅仅是为了符合 SQL-92 标准，expression 是常量、列名、函数以及算术运算符、位运算符和字符串运算符的任意组合。MAX 不能用于 bit 列
MIN ([ALL DISTINCT] expression)	返回表达式中的最小值。参数说明请参考 MAX 函数
SUM ([ALL DISTINCT] expression)	返回表达式中所有值的和或仅非重复值的和。SUM 只能用于数字列（bit 数据类型除外）。ALL（默认值）指定对所有的值应用此聚合函数，DISTINCT 指定 SUM 返回唯一值的和，expression 是常量、列或函数与算术、位和字符串运算符的任意组合
STDEV ([ALL DISTINCT] expression)	返回指定表达式中所有值的标准偏差。ALL（默认值）指定对所有值应用该函数，DISTINCT 指定仅考虑每个唯一值，expression 是一个数值表达式（bit 数据类型除外）
STDEVP ([ALL DISTINCT] expression)	返回指定表达式中所有值的总体标准偏差。参数说明请参考 STDEV 函数
VAR ([ALL DISTINCT] expression)	返回指定表达式中所有值的方差。参数说明请参考 STDEV 函数
VARP ([ALL DISTINCT] expression)	返回指定表达式中所有值的总体方差。参数说明请参考 STDEV 函数

例如，下面的语句使用 COUNT 函数来获取 Employee 表中的员工数量。

```
USE AdventureWorks -- 指定使用的数据库
SELECT COUNT(EmployeeID)
FROM HumanResources.Employee
GO
```

又如，下面的语句使用 SUM 函数计算 SalesOrderDetail 表中的销售总额。

```
USE AdventureWorks
SELECT SUM(LineTotal)
FROM Sales.SalesOrderDetail
GO
```

1.6.2 配置函数

配置函数用于返回当前配置选项的设置信息，如表 1-14 所示。

表 1-14 配置函数及其功能

函 数	功 能
@@DATEFIRST	针对会话返回 SET DATEFIRST 的当前值。SET DATEFIRST 表示指定的每周的第 1 天。美国英语中默认 7 对应星期日
@@DBTS	返回当前数据库最后使用的时间戳值。在插入或更新包含 timestamp 列的行时，将产生一个新的时间戳值
@@LANGID	返回当前使用的语言的本地语言标识符 (ID)。如果要查看有关语言设置的全部信息，可执行不带参数的 sp_helplanguage (EXECUTE sp_helplanguage)
@@LANGUAGE	返回当前所用语言的名称。如果要查看语言设置的全部信息，可执行不带参数的 sp_helplanguage
@@LOCK_TIMEOUT	返回当前会话的锁定超时设置 (毫秒)。可以使用 SET LOCK_TIMEOUT 设置语句等待阻塞资源的最长时间。当一条语句等待的时间长度超过 LOCK_TIMEOUT 所设置的时间长度时，被锁住的语句将自动取消，并给应用程序返回一条错误消息
@@MAX_CONNECTIONS	返回服务器实例允许同时进行的最大用户连接数，实际允许的用户连接数还依赖于所安装的 SQL Server 的版本以及应用程序和硬件的限制
@@MAX_PRECISION	按照服务器中的当前设置，返回 decimal 和 numeric 数据类型所用的精度级别。默认情况下，最大精度为 38
@@NESTLEVEL	返回对本地服务器上执行的当前存储过程的嵌套级别 (初始值为 0)。每次一个存储过程通过引用公共语言运行时 (CLR) 例程、类型或聚合来调用另一个存储过程或执行托管代码时，嵌套级别都会增加。超过最大级数 32 时，事务即被终止
@@OPTIONS	返回有关当前 SET 选项的信息。例如，NOCOUNT 选项的值为 512，下面的示例测试客户端是否启用了 NOCOUNT 选项： SET NOCOUNT ON IF @@OPTIONS & 512 > 0 RAISERROR ('当前用户已设置 NOCOUNT 为 ON', 1, 1)
@@REMSERVER	返回远程数据库服务器在登录记录中显示的名称
@@SERVERNAME	返回本地服务器的名称。在安装了多个服务器实例时，如果本地服务器名称自安装后未发生更改，则 @@SERVERNAME 返回以下本地服务器名称信息： 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'

续表

函 数	功 能
@@SERVICENAME	返回服务器正在其下运行的注册表项的名称。如果当前实例为默认实例，则 @@SERVICENAME 返回 MSSQLSERVER；如果当前实例是命名实例，则该函数返回该实例名称
@@SPID	返回当前用户进程的会话 ID
@@TEXTSIZE	返回 SET 语句中的 TEXTSIZE 选项的当前值。该值指定 SELECT 语句返回的 varchar(max)、nvarchar(max)、varbinary(max)、text 或 image 数据的最大长度
@@VERSION	返回当前的数据库服务器版本、处理器体系结构、生成日期和操作系统

例如，下面的语句使用 @@LANGUAGE 函数返回当前会话的语言名称。

```
SELECT @@LANGUAGE AS 语言名称
```

1.6.3 游标函数

游标函数用于返回有关游标的信息，如表 1-15 所示。

表 1-15 游标函数及其功能

函 数	功 能
@@CURSOR_ROWS	<p>返回最近打开游标的结果集内的行数。为了提高性能，服务器支持异步填充大型键集和静态游标。在异步填充的情况下，@@CURSOR_ROWS 返回的数字是负数。以下是该函数返回值的含义。</p> <ul style="list-style-type: none"> • -m 游标被异步填充。返回值 (-m) 是键集中当前的行数。 • -1 游标为动态游标。因为动态游标可反映所有更改，所以游标符合条件的行数不断变化。因此，永远不能确定已检索到所有符合条件的行。 • 0 没有已打开的游标，对于上一个打开的游标没有符合条件的行，或上一个打开的游标已被关闭或被释放。 • n 游标已完全填充。返回值 (n) 是游标中的总行数
@@FETCH_STATUS	<p>返回最近 FETCH 语句的状态，可以通过函数的返回值判断数据是否读取完毕。以下是函数返回值的含义。</p> <ul style="list-style-type: none"> • 0 FETCH 语句成功。 • -1 FETCH 语句失败或行不在结果集中。 • -2 提取的行不存在
CURSOR_STATUS ({ 'local', 'cursor_name' } { 'global', 'cursor_name' } { 'variable', 'cursor_variable' })	<p>用于确定存储过程是否已为给定的参数返回了游标和结果集。'local' 是一个常量，指示游标的源是一个本地游标名。'cursor_name' 是游标的名称。'global' 是一个常量，指示游标的源是一个全局游标名。'variable' 是一个常量，该常量指示游标的源是一个本地变量。'cursor_variable' 是游标变量的名称，必须使用 cursor 数据类型定义游标变量</p>

下面的示例使用 @@FETCH_STATUS 函数和 WHILE 循环，逐步检索游标中的行，直至检索完毕。


```

DECLARE Employee_Cursor CURSOR FOR
SELECT EmployeeID, Title FROM AdventureWorks.HumanResources.Employee -- 建立游标
OPEN Employee_Cursor -- 打开游标
FETCH NEXT FROM Employee_Cursor -- 开始检索
WHILE @@FETCH_STATUS = 0
BEGIN
    FETCH NEXT FROM Employee_Cursor -- 检索下一行
END
CLOSE Employee_Cursor -- 关闭游标
DEALLOCATE Employee_Cursor -- 删除游标引用
GO

```

1.6.4 日期和时间函数

日期和时间函数用于对日期和时间输入值执行操作，并返回一个字符串、数字值或日期和时间值，如表 1-16 所示。

表 1-16 日期和时间函数及其功能

函 数	功 能																								
DATEADD (datepart , number, date)	<p>返回给指定日期加上一个时间间隔后的新 datetime 值。</p> <p>datepart 指定要为日期的哪部分增加数值。下面列出了可识别的日期部分及其缩写。</p> <table><tr><td>日期部分</td><td>缩写</td><td>可用值的范围</td></tr><tr><td>year</td><td>yy, yyyy</td><td>年 (1753~9999)</td></tr><tr><td>quarter</td><td>qq, q</td><td>季度 (1~4)</td></tr><tr><td>month</td><td>mm, m</td><td>月份 (1~12)</td></tr><tr><td>dayofyear</td><td>dy, y</td><td>一年中的日期 (1~366)</td></tr><tr><td>day</td><td>dd, d</td><td>一月中的日期 (1~31)</td></tr><tr><td>week</td><td>wk, ww</td><td>一年中的星期 (1~53)</td></tr><tr><td>weekday</td><td>dw, w</td><td>一周中的日期 (1~7, Sun~Sat)</td></tr></table>	日期部分	缩写	可用值的范围	year	yy, yyyy	年 (1753~9999)	quarter	qq, q	季度 (1~4)	month	mm, m	月份 (1~12)	dayofyear	dy, y	一年中的日期 (1~366)	day	dd, d	一月中的日期 (1~31)	week	wk, ww	一年中的星期 (1~53)	weekday	dw, w	一周中的日期 (1~7, Sun~Sat)
日期部分	缩写	可用值的范围																							
year	yy, yyyy	年 (1753~9999)																							
quarter	qq, q	季度 (1~4)																							
month	mm, m	月份 (1~12)																							
dayofyear	dy, y	一年中的日期 (1~366)																							
day	dd, d	一月中的日期 (1~31)																							
week	wk, ww	一年中的星期 (1~53)																							
weekday	dw, w	一周中的日期 (1~7, Sun~Sat)																							
DATEADD (datepart , number, date)	<table><tr><td>hour</td><td>hh</td><td>小时 (0~23)</td></tr><tr><td>minute</td><td>mi, n</td><td>分钟 (0~59)</td></tr><tr><td>second</td><td>ss, s</td><td>秒 (0~59)</td></tr><tr><td>millisecond</td><td>ms</td><td>毫秒 (0~999)</td></tr></table> <p>number 指定要相加的值。如果指定了非整数值，则将舍弃该值的小数部分。</p> <p>date 是一个日期表达式，该表达式将与 number 相加。</p> <p>例如，下面的语句在 2005 年 12 月 1 日的基础上增加 8 个月，返回值为 2006 年 8 月 1 日。</p> <pre>DECLARE @curdate as datetime SET @curdate = '20051201' SELECT DATEADD(month, 8, @curdate)</pre>	hour	hh	小时 (0~23)	minute	mi, n	分钟 (0~59)	second	ss, s	秒 (0~59)	millisecond	ms	毫秒 (0~999)												
hour	hh	小时 (0~23)																							
minute	mi, n	分钟 (0~59)																							
second	ss, s	秒 (0~59)																							
millisecond	ms	毫秒 (0~999)																							
DATEDIFF (datepart , startdate , enddate)	<p>返回跨两个指定日期的日期边界数和时间边界数。datepart 请参考 DATEADD 函数中的介绍。startdate 指定计算的开始日期。enddate 指定计算的结束日期。如果 startdate 晚于 enddate，则返回负值</p>																								

续表

函 数	功 能
DATENAME (datepart ,date)	返回 date 指定日期中的指定部分。datepart 请参考 DATEADD 函数中的介绍。date 是一个日期表达式。 例如, 执行下面的语句, 将返回“星期二”。 SELECT DATENAME(weekday,'20060418')
DATEPART (datepart , date)	返回 date 指定日期中指定部分的整数。参数说明见 DATENAME 函数。例如, 下面的语句将返回 3。 SELECT DATEPART(weekday,'20060418')
DAY (date)	返回指定日期的“天”部分。等价于 DATEPART(dd, date)
GETDATE ()	返回当前的系统日期和时间
GETUTCDATE()	返回表示当前的 UTC 时间(通用协调时间或格林尼治标准时间)的 datetime 值。当前的 UTC 时间得自当前的本地时间和运行服务器实例的计算机操作系统中的时区设置
MONTH (date)	返回指定日期的“月”部分的整数。等价于 DATEPART(mm, date)
YEAR (date)	返回表示指定日期的年份的整数。等价于 DATEPART(yy, date)

1.6.5 数学函数

数学函数根据提供的输入值执行计算后返回一个数值, 如表 1-17 所示。其中, 算术函数(如 ABS、CEILING、DEGREES、FLOOR、POWER、RADIANS 和 SIGN) 返回与输入值相同数据类型的值。三角函数和其他函数(包括 EXP、LOG、LOG10、SQUARE 和 SQRT) 将输入值转换为 float 并返回 float 值。

表 1-17 数学函数及其功能

函 数	功 能
ABS (numeric_expression)	返回指定数值的绝对值。numeric_expression 是精确数字或近似数字数据类型类别 (bit 数据类型除外) 的表达式
ACOS (float_expression)	以弧度表示 float_expression 的反余弦值。float_expression 是 float 型或可以隐式转换为 float 型的表达式, 取值范围从-1 到 1
ASIN (float_expression)	以弧度表示 float_expression 的正弦值。float_expression 是 float 型或可以隐式转换为 float 型的表达式, 取值范围从-1 到 1
ATAN (float_expression)	以弧度表示 float_expression 的反正切值。float_expression 是 float 型或可以隐式转换为 float 型的表达式
ATN2 (float_expression1, float_expression 2)	以弧度表示 float_expression1/float_expression 2 的反正切值。float_expression1 和 float_expression 2 是数据类型为 float 的表达式
CEILING (numeric_expression)	返回大于或等于指定数值表达式的最小整数。numeric_expression 是精确数字或近似数字数据类型 (bit 数据类型除外) 的表达式
COS (float_expression)	以弧度表示 float_expression 的余弦值。float_expression 是 float 型的表达式

续表

函 数	功 能
COT (float_expression)	以弧度表示 float_expression 的余切值。float_expression 是 float 型或能够隐式转换为 float 型的表达式
DEGREES (numeric_expression)	返回由弧度 numeric_expression 指定的角度值。numeric_expression 是精确数字或近似数字数据类型 (bit 数据类型除外) 的表达式
EXP (float_expression)	取幂, 由 float_expression 指定的自然对数的幂 (基数为 e)。float_expression 是 float 型或能够隐式转换为 float 型的表达式
FLOOR (numeric_expression)	返回小于或等于 numeric_expression 的最大整数。numeric_expression 是精确数字或近似数字数据类型 (bit 数据类型除外) 的表达式
LOG (float_expression)	返回 float_expression 的自然对数。float_expression 是属于 float 型或能够隐式转换为 float 型的表达式
LOG10 (float_expression)	返回 float_expression 的常用对数。float_expression 是属于 float 型或能够隐式转换为 float 型的表达式
PI ()	返回常数 3.14159265358979
POWER (numeric_expression , y)	返回 numeric_expression 的 y 次幂的值。numeric_expression 和 y 是精确数值或近似数值数据类型 (bit 数据类型除外) 的表达式
RADIANS (numeric_expression)	返回由角度 numeric_expression 表示的弧度值。numeric_expression 是精确数字或近似数字数据类型 (bit 数据类型除外) 的表达式
RAND ([seed])	返回从 0 到 1 之间的随机 float 值。seed 是提供种子值的整数表达式 (tinyint、smallint 或 int)。如果未指定 seed, 则随机分配种子值。对于指定的种子值, 返回的结果始终相同
ROUND (numeric_expression , length [,function])	将 numeric_expression 四舍五入到指定的长度或精度。numeric_expression 是精确数值或近似数值数据 (bit 数据类型除外) 的表达式。 length 指定 numeric_expression 的舍入精度。length 必须是 tinyint、smallint 或 int。如果 length 为正数, 则将 numeric_expression 舍入到 length 指定的小数位数。如果 length 为负数, 则将 numeric_expression 小数点左边部分舍入到 length 指定的长度。function 指定要执行的操作的类型。function 必须为 tinyint、smallint 或 int。如果省略 function 或其值为 0 (默认值), 则将舍入 numeric_expression; 如果指定了 0 以外的值, 则将截断 numeric_expression
SIGN (numeric_expression)	当 numeric_expression 为负数时, 返回 -1; 当 numeric_expression 为 0 时, 返回 0; 当 numeric_expression 为正数时, 返回 1
SIN (float_expression)	以弧度表示 float_expression 的正弦值。float_expression 是 float 型或能够隐式转换为 float 型的表达式
SQRT (float_expression)	返回 float_expression 的平方根。float_expression 是 float 型或可以隐式转换为 float 型的表达式
SQUARE (float_expression)	返回 float_expression 的平方。float_expression 是 float 型或可以隐式转换为 float 型的表达式
TAN (float_expression)	以弧度表示 float_expression 的正切值。float_expression 是 float 型或可以隐式转换为 float 型的表达式

1.6.6 数据类型转换函数

数据类型转换函数仅包括 CAST 和 CONVERT 函数，二者的功能类似。使用这两个函数可以显式地将一种数据类型的表达式转换为另一种数据类型的表达式。

下面分别是 CAST 和 CONVERT 函数的语法格式：

```
CAST ( expression AS data_type [ ( length ) ] )
CONVERT ( data_type [ ( length ) ] , expression [ , style ] )
```

expression 是任何有效的表达式。data_type 是目标数据类型（仅限于系统数据类型），包括 xml、bigint 和 sql_variant。length 是 nchar、nvarchar、char、varchar、binary 或 varbinary 数据类型的可选参数，对于 CONVERT 函数，如果未指定 length，则默认为 30 个字符。

style 用于以下 3 个方面。

- 指定在将 datetime 或 smalldatetime 数据转换为字符数据（nchar、nvarchar、char、varchar、nchar 或 nvarchar 数据类型）时的日期格式的样式。
- 用于将 float、real、money 或 smallmoney 数据转换为字符数据时的字符串格式的样式。
- 用于指定二进制和字符型十六进制值之间数据转换时的格式（仅限于 SQL Server 2008）。

表 1-18 列出了将 datetime 或 smalldatetime 数据转换为字符数据时的可用值。左侧的两列是 style 的值，将 style 值加 100，将返回包括世纪数的 4 位年份格式。

表 1-18 转换 datetime 或 smalldatetime 为字符数据时的 style 可用值

不带世纪数位	带世纪数位	标 准	日期/时间格式
-	0 或 100	默认设置	mon dd yyyy hh:miAM（或 PM）
1	101	美国	mm/dd/yyyy
2	102	ANSI	yy.mm.dd
3	103	英国/法国	dd/mm/yy
4	104	德国	dd.mm.yy
5	105	意大利	dd-mm-yy
6	106	-	dd mon yy
7	107	-	mon dd, yy
8	108	-	hh:mm:ss
-	9 或 109	默认设置+毫秒	mon dd yyyy hh:mi:ss:mmmAM（或 PM）
10	110	美国	mm-dd-yy
11	111	日本	yy/mm/dd
12	112	ISO	yymmdd

续表

不带世纪数位	带世纪数位	标 准	日期/时间格式
-	13 或 113	欧洲默认设置+毫秒	dd mon yyyy hh:mm:ss:mmm (24h)
14	114	-	hh:mi:ss:mmm (24h)
-	20 或 120	ODBC 规范	yyyy-mm-dd hh:mi:ss (24h)
-	21 或 121	ODBC 规范 (带毫秒)	yyyy-mm-dd hh:mi:ss.mmm (24h)
-	126	ISO8601, 为用于 XML 而设计	yyyy-mm-ddThh:mm:ss.mmm (无 空格)
-	127	带时区 Z 的 ISO8601。仅支持从字符数据转换为 datetime 或 smalldatetime。时区指示符 Z 为可选项。仅表示日期或时间成分的字符数据将转换为 datetime 数据类型。未指定的时间成分设置为 00:00:00.000, 未指定的日期成分设置为 1900-01-01	yyyy-mm-ddThh:mm:ss.mmmZ (无空格)
-	130	回历。这是一种有多种变体的日历系统, SQL Server 使用科威特算法。默认情况下, SQL Server 基于截止年份 2049 年来解释两位数的年份。换言之, 就是将两位数的年份 49 解释为 2049, 将两位数的年份 50 解释为 1950。为对日期进行一致性处理, 建议使用四位数年份	dd mon yyyy hh:mi:ss:mmmAM
-	131	回历	dd/mm/yy hh:mi:ss:mmmAM

表 1-19 列出了在将 float 或 real 转换为字符数据时的 style 可用值。

表 1-19 转换 float 或 real 为字符数据时的 style 可用值

style 值	输 出
0 (默认值)	最多包含 6 位。根据需要使用科学记数法
1	始终为 8 位值。始终使用科学记数法
2	始终为 16 位值。始终使用科学记数法

表 1-20 列出了在将 money 或 smallmoney 转换为字符数据时的 style 可用值。

表 1-20 转换 money 或 smallmoney 为字符数据时的 style 可用值

style 值	输 出
0 (默认值)	小数点左侧每 3 位数字之间不以逗号分隔, 小数点右侧取 2 位数, 例如 4235.98
1	小数点左侧每 3 位数字之间以逗号分隔, 小数点右侧取 2 位数, 例如 3,510.92
2	小数点左侧每 3 位数字之间不以逗号分隔, 小数点右侧取 4 位数, 例如 4235.9819

在将 numeric 或 decimal 数据转换为字符数据时, 如果要删除结果集尾随的零, 可以使用 128 作为 style 的值。

表 1-21 列出了在将字符串转换为 xml 数据时的 style 可用值。

表 1-21

转换字符串为 xml 数据时的 style 可用值

style 值	输 出
0 (默认值)	使用默认的分析行为, 即放弃无用的空格, 且不允许使用内部 DTD 子集
1	保留无用空格。此样式设置将默认的 xml:space 处理方式设置为与指定了 xml:space="preserve" 的行为相同
2	启用有限的内部 DTD 子集处理, 并忽略外部 DTD 子集。此外, 不评估 XML 声明来查看 standalone 属性是设置为 yes 还是 no, 而是将 XML 实例当成一个独立文档进行分析; 如果启用了此设置, 服务器将使用内部 DTD 子集提供的以下信息来执行非验证分析操作: <ul style="list-style-type: none"> • 应用属性的默认值; • 解析并扩展内部实体引用; • 检查 DTD 内容模型以实现语法的正确性
3	保留无用空格, 并启用有限的内部 DTD 子集处理

下面的语句使用 CAST 函数将 Production.Product 表中的 ListPrice 列由 money 转换为 varchar 数据类型。

```
USE AdventureWorks
SELECT 'The list price is ' + CAST(ListPrice AS varchar(12)) AS ListPrice
FROM Production.Product
GO
```

返回的结果类似下列形式:

```
ListPrice
-----
The list price is 8.09
The list price is 108.00
The list price is 7.16
使用 CONVERT 函数同样可以实现上面的功能, 参考下面的语句。
USE AdventureWorks
SELECT 'The list price is ' + CONVERT(varchar(12), ListPrice) AS ListPrice
FROM Production.Product
GO
```

表 1-22 列出了二进制和字符型十六进制值转换时的 style 可用值。

表 1-22

二进制和字符型十六进制值转换时的 style 可用值

值	输 出
0 (默认值)	将 ASCII 字符转换为二进制字节, 或者将二进制字节转换为 ASCII 字符。每个字符或字节按照 1:1 进行转换。如果 data_type 为二进制类型, 则会在结果左侧添加字符 0x
1, 2	对于 style 1, 将在转换后的结果左侧添加字符 0x。作为要转换的二进制表达式, 字符 0x 必须为表达式中的前两个字符。 在 style 为 2 的情况下, 生成的二进制值不会包含字符 0x。作为要转换的二进制表达式, 也不需要再在字符前面包含字符 0x。 如果 data_type 为二进制类型, 则表达式必须为字符表达式; 如果转换后的表达式长度大于 data_type 长度, 则会在右侧截断结果; 如果固定长度 data_types 大于转换后的结果, 则会在结果右侧添加零。 如果 data_type 为字符类型, 则表达式必须为二进制表达式。每个二进制字符均转换为两个十六进制字符。如果转换后的表达式长度大于 data_type 长度, 则会在右侧截断结果。 如果 data_type 为固定大小的字符类型, 并且转换后的结果长度小于其 data_type 长度, 则会在转换后的表达式右侧添加空格, 以使十六进制数字的个数保持为偶数

参考下面的示例代码：

```
-- 转换二进制值 0x4E616D65 到一个字符值
SELECT CONVERT(char(8), 0x4E616D65, 0) AS 'Style 0. 二进制到字符'
-- 下面的示例演示了 Style 为 1 的情况下, 如何强行截断结果值
-- 产生的结果值由于包含字符 0x , 所以被截断
SELECT CONVERT(char(8), 0x4E616D65, 1) AS 'Style 1. 二进制到字符'
-- 下面的示例演示了 Style 为 2 的情况下, 没有截断结果值
-- 这是因为 0x 字符未包含在结果中
SELECT CONVERT(char(8), 0x4E616D65, 2) AS 'Style 2. 二进制到字符'
-- 转换字符值 Name 到一个二进制值
SELECT CONVERT(binary(8), 'Name', 0) AS 'Style 0. 字符到二进制'
SELECT CONVERT(binary(4), '0x4E616D65', 1) AS 'Style 1. 字符到二进制'
SELECT CONVERT(binary(4), '4E616D65', 2) AS 'Style 2. 字符到二进制'
```

运行结果如图 1-2 所示。

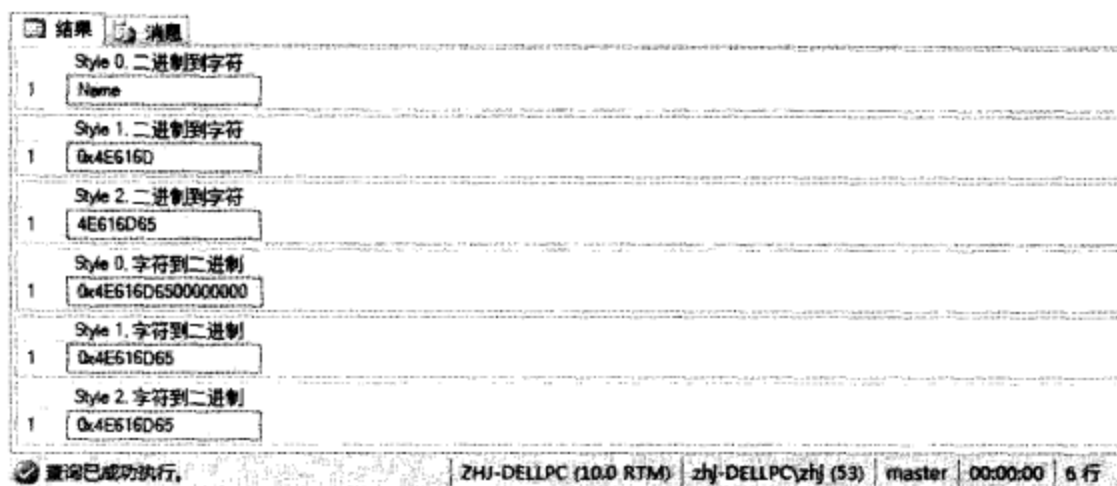


图 1-2 转换结果

1.6.7 字符串函数

使用字符串函数可以对字符串输入值进行剪裁、替换等操作，并返回字符串或数值。字符串函数及其功能如表 1-23 所示。

表 1-23

字符串函数及其功能

函 数	功 能
ASCII (character_expression)	<p>返回字符表达式中最左侧的字符的 ASCII 代码值。character_expression 是 char 或 varchar 类型的表达式。</p> <p>例如，执行下面的语句将返回字母 A 的 ASCII 代码值 65：</p> <pre>SELECT ASCII('A')</pre>
CHAR (integer_expression)	<p>将 int 类型 ASCII 代码值转换为字符。使用此函数可以将控制字符插入字符串中，如制表符 (CHAR(9))、回车符 (CHAR(13))。integer_expression 是介于 0~255 之间的整数。</p> <p>例如，执行下面的语句将返回字符“A”：</p> <pre>SELECT CHAR(65)</pre>

续表

函 数	功 能
CHARINDEX (expression1 ,expression2 [, start_location])	<p>返回字符串中指定表达式的开始位置。</p> <p>expression1 是要查找的字符串。expression2 是包含要查找字符串的表达式。start_location 指定在 expression2 中搜索 expression1 时的开始位置，如果未指定该参数，或是指定了一个负数或零，则将从 expression2 的开头开始搜索。</p> <p>如果在 expression2 内找到了 expression1，则返回字符串的所在位置；否则，CHARINDEX 返回 0。</p> <p>例如，下面的语句在指定字符串查找“山”，将返回 7： SELECT CHARINDEX(N'山',N'春天的山已经绿了',1)</p>
DIFFERENCE (character_expression , character_expression)	<p>返回一个 0~4 之间的整数值，表示两个字符表达式 SOUNDEX 值之间的差异。0 表示几乎不同或完全不同，4 表示几乎相同或完全相同。character_expression 为 char 或 varchar 类型的表达式。也可以是 text 类型，但只有前 8000 个字节有效。</p> <p>例如，下面的语句返回 3，表示两个表达式大体相同： SELECT DIFFERENCE('Gree','Greene')</p>
LEFT (character_expression , integer_expression)	<p>返回字符串中从左边开始指定个数的字符。</p> <p>character_expression 是字符或二进制数据表达式。character_expression 可以是任何能够隐式转换为 varchar 或 nvarchar 的数据类型，否则，应当使用 CAST 函数进行显式转换。integer_expression 指定从 character_expression 返回的字符数。</p> <p>例如，下面的语句将返回字符串的前两个字符“这是”： SELECT LEFT(N'这是我的字符串',2)</p>
LEN (string_expression)	<p>返回指定字符串表达式的字符（而不是字节）数，其中不包含尾随空格。string_expression 是要计算的字符串。</p> <p>例如，下面的语句返回“7” SELECT LEN('这是我的字符串');</p>
LOWER (character_expression)	<p>将大写字符数据转换为小写字符数据后返回字符表达式。</p> <p>character_expression 是字符或二进制数据的表达式。character_expression 必须是可隐式转换为 varchar 的数据类型，否则，需要使用 CAST 显式转换</p>
LTRIM (character_expression)	<p>返回删除了前导空格之后的字符表达式。character_expression 参考 LOWER 函数中的解释</p>
NCHAR (integer_expression)	<p>根据 Unicode 标准的定义，返回指定整数代码的 Unicode 字符。</p> <p>integer_expression 是介于 0~65535 之间的正整数</p>
PATINDEX ('%pattern%', expression)	<p>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。</p> <p>pattern 是一个文字字符串。可以使用通配符，但 pattern 之前和之后必须有“%”字符（搜索第 1 个或最后 1 个字符时除外）。</p> <p>expression 是要在其中查找 pattern 模式的字符串数据类型表达式。</p> <p>例如，下面的语句用于在字符串中查找与“%my%”模式匹配的字符出现的位置： SELECT PATINDEX('%my%', 'This is my strings')</p>

续表

函 数	功 能
QUOTENAME ('character_string' [, 'quote_character'])	<p>返回带有分隔符的 Unicode 字符串。</p> <p>'character_string'是由 Unicode 字符数据构成的字符串。'quote_character'是用作分隔符的单字符字符串。可以是单引号 (')、左方括号或右方括号 ([、]) 或者英文双引号 (")。如果未指定此参数, 则使用方括号。</p> <p>例如, 下面的语句接收字符串 "abc[]def" 并使用 "[" 和 "]" 字符来创建有效的分隔标识符。返回 "[abc[]def]" :</p> <p>SELECT QUOTENAME('abc[]def')</p>
REPLACE ('string_expression1' , 'string_expression2' , 'string_expression3')	<p>将 string_expression1 中的所有 string_expression2 字符串替换为 'string_expression3'。</p> <p>例如, 下面的语句使用 "xxx" 替换 "abcdefghicde" 中的所有 "cde" 字符串, 得到 "abxxxfgihxxx" 结果:</p> <p>SELECT REPLACE('abcdefghicde','cde','xxx')</p>
REPLICATE (character_expression , integer_expression)	<p>以指定的次数重复字符表达式。character_expression 是要重复的字符。integer_expression 指定重复次数。</p> <p>例如, 下面的语句返回 "abcabcabc" :</p> <p>SELECT REPLICATE('abc',3)</p>
REVERSE (character_expression)	<p>返回字符表达式的逆向表达式。</p> <p>character_expression 是要进行逆向的字符数据表达式。</p> <p>例如, 下面的语句返回 "cba" :</p> <p>SELECT REVERSE('abc')</p>
RIGHT (character_expression , integer_expression)	<p>返回字符串中从右边开始指定个数的字符。参数说明参考 LEFT 函数</p>
RTRIM (character_expression)	<p>截断 character_expression 字符型数据表达式的所有尾随空格后并返回一个字符串</p>
SOUNDEX (character_expression)	<p>返回一个由 4 个字符组成的代码 (SOUNDEX), 用于评估两个字符串的相似性。character_expression 是一个字符数据的字母数字表达式。</p> <p>参考下面的语句:</p> <p>SELECT SOUNDEX ('Smith'), SOUNDEX ('Smythe')</p> <p>返回结果如下:</p> <p>-----</p> <p>S252 S200</p>
SPACE (integer_expression)	<p>返回由重复的空格组成的字符串。integer_expression 指定空格的数量</p>
STR (float_expression [, length [, decimal]])	<p>将数字数据转换为字符数据。</p> <p>float_expression 是带小数点的近似数字 (float) 数据类型的表达式</p> <p>length 指定要生成字符数据的总长度, 包括小数点、符号、数字以及空格。</p> <p>decimal 指定小数点后的位数, 必须小于或等于 16</p>
STUFF (character_expression , start , length , character_expression)	<p>删除指定长度的字符, 并在指定的起点处插入另一组字符。</p> <p>character_expression 是一个字符数据表达式。start 是一个整数, 指定删除和插入的开始位置。length 是一个整数, 指定要删除的字符数。</p> <p>下面的语句从第 1 个字符串 "abcdef" 的第 2 个位置 (字符 b) 开始, 删除 3 个字符, 然后在删除的起始位置插入第 2 个字符串, 得到 "aijklmnef" :</p> <p>SELECT STUFF('abcdef', 2, 3, 'ijklmn')</p>

续表

函 数	功 能
SUBSTRING (expression ,start , length)	截取字符表达式、二进制表达式、文本表达式或图像表达式的一部分。 expression 是字符串、二进制字符串、文本、图像、列或包含列的表达式，不能使用包含聚合函数的表达式。start 是指定开始截取的位置的整数。 length 指定要截取的长度。 例如，下面的语句从第 3 个字符位置开始截取 2 个字符，返回值为“cd”： SELECT SUBSTRING('abcdefg', 3,2)
UNICODE ('ncharacter_expression')	按照 Unicode 标准的定义，返回输入表达式中第 1 个字符的整数值。该函数与 NCHAR 函数的功能相反。'ncharacter_expression'是一个 nchar 或 nvarchar 表达式
UPPER (character_expression)	将小写字符数据转换为大写字符后返回表达式。character_expression 参考 LOWER 函数中的介绍

1.6.8 文本和图像函数

使用文本和图像函数可以对文本或图像输入值或列执行操作。文本和图像函数及其功能如表 1-24 所示。

表 1-24

文本和图像函数及其功能

函 数	功 能
TEXTPTR (column)	返回指向 text、ntext 或 image 列的 varbinary 格式文本指针值。检索到的文本指针值可用于 READTEXT、WRITETEXT 和 UPDATETEXT 语句。 需要注意的是，在后续版本中将删除该功能，没有可用的替代功能 column 是要使用的 text、ntext 或 image 列
TEXTVALID ('table.column' ,text_ptr)	检查特定文本指针是否有效的 text、ntext 或 image 函数。如果有效返回 1；否则返回 0。 需要注意的是，在后续版本中将删除该功能，没有可用的替代功能 table 指定要使用的表的名称。column 指定要使用的列的名称。text_ptr 是要检查的文本指针

1.7 查询工具

在本章的前面介绍了 SQL 的语法规则、常量和变量、运算符、函数等基本知识，到底通过什么工具来执行 SQL 语句呢？对于程序开发人员而言，可以从客户端应用程序发送 SQL 语句到服务器端执行。此外，出于服务器管理和 SQL 测试等需要，SQL Server 也提供了 Management Studio、sqlcmd、bcp 和 sqlps 工具，可以通过它们执行 SQL 语句。

其中，bcp 用于将大量行插入表，但该工具不需要具有 SQL 知识。sqlps 是一个 Microsoft C#命令提示实用工具，用于以交互方式即席运行 PowerShell 命令或是运行 PowerShell 脚本文件。由于这两个工具的功能超出了本书的范围，我们仅对 SQL Server Management Studio 和 sqlcmd 做一下介绍。

1.7.1 使用 Management Studio 进行 Windows 方式查询

在 Windows 中依次选择“开始”→“程序”→Microsoft SQL Server 2008，单击 SQL Server Management Studio（在 Windows Vista 操作系统下要以管理员身份运行），将打开登录窗口，选择身份验证方式后，将打开 Management Studio。

单击工具栏中的“新建查询”按钮可以打开一个查询窗口，如图 1-3 所示。可以在查询窗口中输入 SQL 语句后，单击“执行”按钮执行查询。如果希望仅执行其中的部分语句，可以选定要执行的语句，然后再单击“执行”按钮。



图 1-3 SQL Server Management Studio

1.7.2 使用 sqlcmd 进行 MS-DOS 方式查询

在 Windows 中依次选择“开始”→“程序”→“附件”→“命令提示符”（在 Vista 中要以管理员身份运行），将打开“命令提示符”窗口。

要连接到 SQL Server 服务器，必须指定服务器名称。安装在命名实例中的，还必须指定实例名。默认情况下，sqlcmd 使用 Windows 身份验证。如果要使用 SQL Server 身份验证连接到 SQL Server 的，则还必须提供连接用户名和密码。例如，如果要连接到名为 server1 的服务器，则需要使用下列参数：

```
sqlcmd -S server1 -U SqlUserAccount -P SqlPassword
```


如果是受信任的 Windows 用户，则可以省略-U 和-P 参数。例如，图 1-4 所示的 sqlcmd 窗口中使用 sqlcmd -S (local) 命令连接到服务器，并从 HumanResources.Employee 表中查找 EmployeeID 为 1 的雇员。注意其中的 GO 命令，该命令用于执行所输入的 SQL 语句。

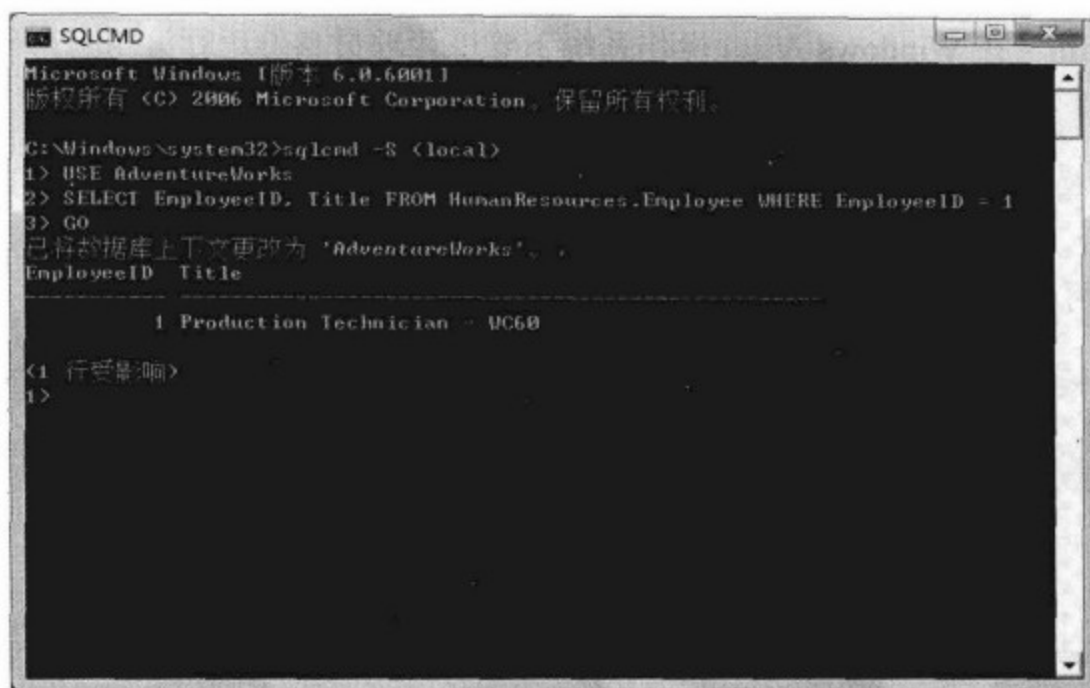


图 1-4 sqlcmd 窗口

要退出 sqlcmd，可以执行 exit 或 quit 命令。

1.8 SQL 书写规范

书写规范与语法规则是两个完全不同的概念，违反语法规则会致程序执行错误，而违反书写规范虽然不会导致错误，但是会导致阅读困难和代码的通用性。这些书写规范是根据大多数人阅读代码时的习惯而提出的，并不是必须完全遵守的。

1.8.1 使用大小写规范提高词义识别能力

1. 在名称中仅使用字母、数字和下划线

之所以要在名称中仅使用字母、数字和下划线，因为这些字符可以被移植到任何其他编程语言中。在应用程序的数据库和宿主语言中能够使用相同的名称，会非常方便。

但是，也存在一些特殊情况。例如，在 SQL Server 中临时表名称需要以“#”开头，而它在其他编程语言中具有特殊含义。如果必须使用临时表，则只能使用“#”。此外，参数名称也存在这种情况，它需要以“@”开头。但是，无论怎样，在名称中尽量避免使用特殊符号是一个非常正确的选择。

不要将下划线作为名称的第一个或最后一个字母，因为这看上去像少了一部分一样。

2. 列名、参数和变量等标量小写

通常情况下，小写单词比大写单词容易阅读。曾经做过测试，阅读小写文本的速度比大写的速度快 5%~10%。当名称由两个单词组合而成时，为便于阅读，应当采用大小写混合的写法。例如，下面按由易至难的方式列出了存放修改日期列的 3 种书写方法：

```
ModifiedDate    -- 比较容易阅读
modifieddate    -- 阅读难度增加
MODIFIEDDATE    -- 阅读最困难
```

但是，也有一种观点认为大小写混合的写法阅读起来比全部小写要难一些，原因是在全部小写的情况下，会把 `modifieddate` 看作是一个单词，而 `ModifiedDate` 这种形式会被看作两个单词，分散注意力。总之，在列名、参数和变量中全部使用大写字母是一个非常糟糕的选择。

3. 模式对象名首字母大写

模式对象包括表、视图和存储过程等，在创建这些名称时，应当将首字母大写，表示为专有名词。

4. 保留关键字大写

保留关键字是 SQL 语言语法的一部分，用于定义、操作和访问数据库。将保留关键字大写后，会起到一种突出效果，使整个语句重点突出、结构清晰。看一下下面的语句：

```
select a, b, c from MyTable where id = 1;
```

对比一下：

```
SELECT a, b, c FROM MyTable WHERE id = 1;
```

阅读上面的两个语句，看一下能否快速找出每个子句，而下面的书写格式则阅读起来会更清晰。

```
SELECT a, b, c
FROM MyTable
WHERE id = 1;
```

下面列出了 SQL Server 的保留关键字：

```
ADD | ALL | ALTER | AND | ANY | AS | ASC | AUTHORIZATION
BACKUP | BEGIN | BETWEEN | BREAK | BROWSE | BULK | BY
CASCADE | CASE | CHECK | CHECKPOINT | CLOSE | CLUSTERED
COALESCE | COLLATE | COLUMN | COMMIT | COMPUTE | CONSTRAINT
CONTAINS | CONTAINSTABLE | CONTINUE | CONVERT | CREATE | CROSS
CURRENT | CURRENT_DATE | CURRENT_TIME | CURRENT_TIMESTAMP
CURRENT_USER | CURSOR
DATABASE | DBCC | DEALLOCATE | DECLARE | DEFAULT | DELETE
DENY | DESC | DISK | DISTINCT | DISTRIBUTED | DOUBLE | DROP | DUMP
ELSE | END | ERRLVL | ESCAPE | EXCEPT | EXEC | EXECUTE | EXISTS
EXIT | EXTERNAL
FETCH | FILE | FILLFACTOR | FOR | FOREIGN | FREETEXT
FREETEXTTABLE | FROM | FULL | FUNCTION
GOTO | GRANT | GROUP
HAVING | HOLDLOCK | IDENTITY | IDENTITY_INSERT | IDENTITYCOL
IF | IN | INDEX | INNER | INSERT | INTERSECT | INTO | IS
```

```

| JOIN
| KEY | KILL | LEFT | LIKE | LINENO | LOAD
| MERGE
| NATIONAL | NOCHECK | NONCLUSTERED | NOT | NULL | NULLIF
| OF | OFF | OFFSETS | ON | OPEN | OPENDATASOURCE | OPENQUERY
| OPENROWSET | OPENXML | OPTION | OR | ORDER | OUTER | OVER
| PERCENT | PIVOT | PLAN | PRECISION | PRIMARY | PRINT
| PROC | PROCEDURE | PUBLIC
| RAISERROR | READ | READTEXT | RECONFIGURE | REFERENCES
| REPLICATION | RESTORE | RESTRICT | RETURN | REVERT | REVOKE
| RIGHT | ROLLBACK | ROWCOUNT | ROWGUIDCOL | RULE
| SAVE | SCHEMA | SECURITYAUDIT | SELECT | SESSION USER | SET
| SETUSER | SHUTDOWN | SOME | STATISTICS | SYSTEM USER
| TABLE | TABLESAMPLE | TEXTSIZE | THEN | TO | TOP | TRAN
| TRANSACTION | TRIGGER | TRUNCATE | TSEQUAL
| UNION | UNIQUE | UNPIVOT | UPDATE | UPDATETEXT | USE | USER
| VALUES | VARYING | VIEW
| WAITFOR | WHEN | WHERE | WHILE | WITH | WRITETEXT

```

1.8.2 使用空格提供更好的语言标记区分

在语言标记之间放置一个空格，尽量地符合英语书写习惯，可以增强语句的可阅读性。

1. 等号两边使用空格

在书写赋值语句时，应当在等号两边使用空格分隔，如 `SET @i = 1` 比 `SET @i=1` 更容易阅读。

2. 逗号后面使用空格

应当遵循在逗号后面使用空格的原则，因为英语中逗号和句号很容易混淆，如：

```

SELECT MyTable.a,MyTable1.b,MyTable2.c
FROM MyTable,MyTable1,MyTable2

```

下面的形式会更容易阅读一些：

```

SELECT MyTable.a, MyTable1.b, MyTable2.c
FROM MyTable, MyTable1, MyTable2

```

当表或列名称比较长时，下面的形式则更好一些。

```

SELECT EmployeeID,
       Title,
       BirthDate,
       MaritalStatus
FROM HumanResources.Employee

```

1.8.3 使用缩进提高语句的逻辑层次表达能力

必要的缩进会使语句的层次和逻辑关系更加清晰，通常是缩进 2 个空格。例如，在下面的语句中，`AND` 关键词连接了两个筛选条件，缩进后会更加突出 `WHERE` 子句。

```
SELECT *
FROM HumanResources.Employee
WHERE ManagerID = 16
      AND EmployeeID > 100
```

下面是一个左外连接的语句，首先将 `HumanResources.Employee` 和 `Person.Contact` 表中列分别放在了单独的行中，以便进行区分；然后 `LEFT` 缩进后表示与 `FROM` 后面的表进行连接，`ON` 再次缩进表示是 `LEFT` 的连接条件。

```
SELECT E.EmployeeID, E.Title,
       P.FirstName, P.LastName, P.EmailAddress
FROM HumanResources.Employee AS E
LEFT OUTER JOIN Person.Contact AS P
  ON E.EmployeeID = P.ContactID
WHERE E.ManagerID = 16
      AND E.EmployeeID > 100
```

1.8.4 使用垂直空白道提高关键词与参数的区分能力

还有一种观点认为，在关键词与参数之间应当使用垂直的空白道方式进行分隔，会增强可阅读性，如：

```
SELECT E.EmployeeID, E.Title,
       P.FirstName, P.LastName, P.EmailAddress
FROM HumanResources.Employee AS E
LEFT OUTER JOIN Person.Contact AS P
  ON E.EmployeeID = P.ContactID
WHERE E.ManagerID = 16
      AND E.EmployeeID > 100
```

又如，下面的语句使用了垂直空白道分隔，并对子查询使用了缩进。

```
SELECT DISTINCT CustName
FROM Customers AS C
WHERE NOT EXISTS
  (SELECT *
   FROM OrderHeader
   WHERE CustID = Customers.CustID);
```

1.8.5 使用分组进行语句的段落划分

存在多行 `SQL` 的情况下，相关语句之间可以直接换行书写，而对于两个步骤之间的语句应当间隔一个空行。如果需要的话，也可以加入一些适当地注释语句。例如：

```
USE AdventureWorks;
GO

-- 读取 Employee 表的数据
SELECT *
FROM HumanResources.Employee;
GO
```



第2章 数据库管理

从本章开始,以及后面的几章,我们将简要讲述一下创建数据库、表和索引等方面的知识。实际上,如果仅仅是单纯的 SELECT 语句查询,你可能感觉不到数据库存在的价值和意义。在一些小型数据库系统中,对于自由表同样支持符合 ANSI 标准的 SELECT 查询,并不需要创建数据库。但是,当需要保持表之间数据的一致性时,你可能需要使用到触发器这样的工具,例如,当删除一个表中的某行时,触发器会自动删除另一个表中的相关行,这时候就需要使用到数据库。或是当你在开发客户/服务器程序时,如果将所有数据处理都下载到客户端去执行,那将是非常耗时的工作,如果把在服务器端可以完成的逻辑计算分离出来,单独交给服务器来执行,就可以显著提高执行的效率,这时候你可能需要使用到存储过程或函数,这些对象也是存储在数据库中的。从这方面讲,数据库是一个存储表、索引,以及表之间逻辑运算关系的容器。

在 SQL Server 中,可以通过 SQL 语句或 Management Studio 来进行数据库管理,包括创建、修改和删除数据库操作。

2.1 创建数据库

在创建数据库之前,必须先确定数据库的名称、所有者、大小以及存储该数据库的文件和文件组。所谓所有者,即是创建数据库的用户。一般情况下,大多数产品对象由数据库所有者拥有。

在创建数据库之前,应注意下列事项。

- 要创建数据库,必须至少拥有 CREATE DATABASE、CREATE ANY DATABASE 或 ALTER ANY DATABASE 权限。
- 创建数据库的用户将成为该数据库的所有者。
- 对于一个服务器实例,最多可以创建 32 767 个数据库。
- 数据库名称必须遵循为标识符指定的规则。
- 在创建新数据库时,model 数据库中的所有用户定义对象都将复制到所有新创建的数据库中。因此,可以向 model 数据库中添加任何对象(如表、视图、存储过程和数据类型),以便将这些对象包含到所有新创建的数据库中。

2.1.1 数据库文件和文件组

数据库是作为一组操作系统文件的形式出现的。数据和日志信息绝不混合在同一个文件中，而且一个文件只能由一个数据库使用。文件组是文件的命名集合，用于简化数据存放和管理任务（如备份和还原操作）。

1. 数据库文件

可以使用3种文件类型来存储数据库，包括主文件、次要文件和事务日志。

在主文件中包含着数据库的启动信息。此外，主文件还用于存储数据。每个数据库都有一个主文件。主文件的建议文件扩展名为.mdf。

次要文件包含不能放置在主数据文件中的所有数据。如果主文件足够大，能够包含数据库中的所有数据，则该数据库不需要次要数据文件。有些数据库可能非常大，因此需要多个次要数据文件，也可能在独立的磁盘驱动器上使用次要文件以将数据分散到多个磁盘上。次要文件的建议文件扩展名为.ndf。

事务日志文件包含用于恢复数据库的日志信息，每个数据库必须至少有一个事务日志文件。日志文件最小为512KB。事务日志的建议文件扩展名为.ldf。

在创建数据库时，应当根据数据库中预期的最大数据量，创建尽可能大的数据文件。

2. 文件和文件组的填充策略

每个数据库有一个PRIMARY文件组。此文件组包含主文件和未放入其他文件组的所有次要文件。可以创建用户定义的文件组，用于将数据文件集合起来，以便于管理、数据分配和放置。

文件组对组内的所有文件都使用按比例填充策略。将数据写入文件组时，数据库引擎会根据文件中的可用空间量将一定比例的数据写入文件组中的每个文件，而不是将所有数据先写满第1个文件，然后再写入下一个文件。例如，如果文件f1有100MB可用空间，文件f2有200MB可用空间，则从文件f1中分配一个区，从文件f2中分配两个区，依次类推。这样，两个文件几乎同时填满。

文件组中的所有文件一满，数据库引擎就自动按照循环方式一次扩展一个文件，以容纳更多数据（假定数据库设置为自动增长）。例如，某个文件组由3个文件组成，它们都设置为自动增长。当文件组中所有文件的空间都已用完时，只扩展第1个文件。当第1个文件已满，无法再向文件组中写入更多数据时，将扩展第2个文件。当第2个文件已满，无法再向文件组中写入更多数据时，将扩展第3个文件。当第3个文件已满，无法再向文件组中写入更多数据时，将再次扩展第1个文件，依次类推。

使用文件和文件组可以改善数据库的性能，因为这样允许跨多个磁盘、多个磁盘控制器或RAID

(独立磁盘冗余阵列) 系统创建数据库。例如, 如果计算机上有 4 个磁盘, 那么可以创建一个由 3 个数据文件和 1 个日志文件组成的数据库, 每个磁盘上放置 1 个文件。在对数据进行访问时, 4 个读/写磁头可以同时并行地访问数据。这样可以加快数据库操作的速度。

另外, 文件和文件组还允许数据布局, 因为可以在特定的文件组中创建表。这样可以改善性能, 因为可以将特定表的所有 I/O 都定向到一个特定的磁盘。例如, 可以将最常用的表放在一个文件组的一个文件中, 该文件组位于一个磁盘上; 而将数据库中其他不常访问的表放在另一个文件组的其他文件中, 该文件组位于第 2 个磁盘上。

3. 文件和文件组的设计规则

下列规则适用于文件和文件组:

- 一个文件或文件组不能由多个数据库使用;
- 一个文件只能是一个文件组的成员;
- 数据和事务日志信息不能属于同一个文件或文件组;
- 事务日志文件不能属于任何文件组。

下面是使用文件和文件组时的一些建议。

- 大多数数据库在只有单个数据文件和单个事务日志文件的情况下性能良好。
- 如果使用多个文件, 应当为附加文件创建第 2 个文件组, 并将其设置为默认文件组。这样, 主文件将只包含系统表和对象。
- 要使性能最大化, 应当在尽可能多的不同的可用本地物理磁盘上创建文件或文件组。应当将争夺空间最激烈的对象置于不同的文件组中。
- 使用文件组将对象放置在特定的物理磁盘上。
- 将在同一联接查询中使用的不同表置于不同的文件组中。由于采用并行磁盘 I/O 对联接数据进行搜索, 所以性能将得以改善。
- 将最常访问的表和属于这些表的非聚集索引置于不同的文件组中。如果文件位于不同的物理磁盘上, 由于采用并行 I/O, 所以性能将得以改善。
- 不要将事务日志文件置于其中已有其他文件和文件组的物理磁盘上。

2.1.2 CREATE DATABASE 语句的语法格式

可以使用 CREATE DATABASE 语句创建数据库, 其语法格式如下:

```
CREATE DATABASE database_name
    [ ON
        [ PRIMARY ] [ <filespec> [ ....n ]
        [ , <filegroup> [ ....n ] ]
    [ LOG ON { <filespec> [ ....n ] } ]
    ]
    [ COLLATE collation_name ]
```

1 [[:]]

- **database_name**: 要创建的新数据库的名称。
- **ON**: 指定以显式定义方式指定存储数据库数据部分的磁盘文件（数据文件）。
- **PRIMARY**: 指定<filespec>列表中的主文件。在<filespec>项中的第 1 个文件将成为主文件。如果没有指定 PRIMARY, 则 CREATE DATABASE 语句中列出的第 1 个文件将成为主文件。
- **LOG ON**: 指定存储数据库日志的磁盘文件（日志文件）。LOG ON 后跟着以逗号分隔的用于定义日志文件的<filespec>项列表。如果没有指定 LOG ON, 将自动创建一个日志文件, 其大小为该数据库的所有数据文件大小总和的 25%或 512KB, 取两者之中的较大者。
- **COLLATE collation_name**: 指定数据库的默认排序规则。排序规则名称既可以是 Windows 排序规则名称, 也可以是 SQL 排序规则名称。如果没有指定排序规则, 则将服务器实例的默认排序规则分配为数据库的排序规则。排序规则一般用于 SELECT 查询的 ORDER BY 子句, 详细信息参考 5.6 节的介绍。

CREATE DATABASE 语句中的<filespec>部分用于控制文件属性, 其语法格式如下:

```
(
    NAME = logical_file_name ,
    FILENAME = 'os_file_name'
    [ , SIZE = size [ KB | MB | GB | TB ] ]
    [ , MAXSIZE = { max_size [ KB | MB | GB | TB ] | UNLIMITED } ]
    [ , FILEGROWTH = growth_increment [ KB | MB | GB | TB | % ] ]
) [ ....n ]
```

- **NAME logical_file_name**: 指定文件的逻辑名称。logical_file_name 必须在数据库中唯一, 必须符合标识符规则。
- **FILENAME 'os_file_name'**: 指定操作系统（物理）文件名称。执行 CREATE DATABASE 语句前, 指定路径必须存在。如果指定了 UNC（通用命名约定）路径, 则无法设置 SIZE、MAXSIZE 和 FILEGROWTH 参数。
- **SIZE size**: 指定文件的初始大小。如果没有为主文件指定 size, 则数据库引擎将使用 model 数据库中的主文件的大小。如果指定了辅助数据文件或日志文件, 但未指定该文件的 size, 则数据库引擎将以 1MB 作为该文件的大小。

可以使用千字节（KB）、兆字节（MB）、千兆字节（GB）或兆兆字节（TB）后缀, 默认为 MB。

- **MAXSIZE max_size**: 指定文件可增大到的最大大小, 可以使用 KB、MB、GB 和 TB 后缀, 默认为 MB。
- **UNLIMITED**: 指定文件将增长到磁盘已满。指定为 unlimited 增长的日志文件的最大大小为 2TB, 而数据文件的最大大小为 16TB。
- **FILEGROWTH growth_increment**: 指定每次需要新空间时为文件添加的空间量。growth_increment 值不能超过 MAXSIZE 设置值。该值可以使用 MB、KB、GB、TB 或百分比（%）为单位指定。默认值为 MB。growth_increment 值为 0 时表明自动增长被关闭, 不允许增加空间。

如果未指定 FILEGROWTH, 则数据文件的默认值为 1MB, 日志文件的默认增长比例为 10%, 并且最小值为 64 KB。

CREATE DATABASE 语句中的 <filegroup> 部分用于控制文件组属性, 其语法格式如下:
 FILEGROUP filegroup_name [DEFAULT]
 <filespec> [....n]

- FILEGROUP filegroup_name: 文件组的逻辑名称。filegroup_name 必须在数据库中唯一, 不能是系统提供的名称 PRIMARY 和 PRIMARY_LOG。
- DEFAULT: 指定文件组为数据库中的默认文件组。

2.1.3 创建数据库示例

1. 创建未指定文件的数据库

下面的语句将创建名为 mydata 的数据库, 并创建相应的主文件和事务日志文件。因为语句没有 <filespec> 项, 所以主数据库文件的大小为 model 数据库主文件的大小。事务日志将设置为下列值中的较大者: 512 KB 或主数据文件大小的 25%。因为没有指定 MAXSIZE, 文件可以增大到填满所有可用的磁盘空间为止。

```
CREATE DATABASE mydata
```

2. 创建指定数据和事务日志文件的数据库

下面的语句将创建数据库 Sales。因为没有使用关键字 PRIMARY, 第 1 个文件 (Sales_dat) 将成为主文件。因为在 Sales_dat 文件的 SIZE 参数中没有指定 MB 或 KB, 将默认按 MB 分配。Sales_log 文件以 MB 为单位进行分配, 因为 SIZE 参数中显式声明了 MB 后缀。

```
CREATE DATABASE Sales
ON
( NAME = Sales_dat,
  FILENAME = 'c:\saledat.mdf',
  SIZE = 10,
  MAXSIZE = 50,
  FILEGROWTH = 5 )
LOG ON
( NAME = Sales_log,
  FILENAME = 'c:\salelog.ldf',
  SIZE = 5MB,
  MAXSIZE = 25MB,
  FILEGROWTH = 5MB );
```

3. 通过指定多个数据和事务日志文件创建数据库

下面的语句将创建数据库 Archive, 该数据库具有 3 个 100MB 的数据文件和 2 个 100MB 事务日志文件。主文件是列表中的第 1 个文件, 并使用 PRIMARY 关键字显式指定。事务日志文件在 LOG ON 关键字后指定。

```
CREATE DATABASE Archive
```



```

ON
PRIMARY
    (NAME = Arch1,
    FILENAME = 'c:\archdat1.mdf',
    SIZE = 100MB,
    MAXSIZE = 200,
    FILEGROWTH = 20),
    ( NAME = Arch2,
    FILENAME = 'c:\archdat2.ndf',
    SIZE = 100MB,
    MAXSIZE = 200,
    FILEGROWTH = 20),
    ( NAME = Arch3,
    FILENAME = 'c:\archdat3.ndf',
    SIZE = 100MB,
    MAXSIZE = 200,
    FILEGROWTH = 20)
LOG ON
    (NAME = Archlog1,
    FILENAME = 'c:\archlog1.ldf',
    SIZE = 100MB,
    MAXSIZE = 200,
    FILEGROWTH = 20),
    (NAME = Archlog2,
    FILENAME = 'c:\archlog2.ldf',
    SIZE = 100MB,
    MAXSIZE = 200,
    FILEGROWTH = 20);

```

4. 创建具有文件组的数据库

下面的语句将创建数据库 Sales，该数据库具有以下文件组：

- 包含文件 Spri1_dat 和 Spri2_dat 的 PRIMARY 文件组。将这些文件的 FILEGROWTH 增量指定为 15%；
- 名为 SalesGroup1 的文件组，其中包含文件 SGrp1Fi1 和 SGrp1Fi2；
- 名为 SalesGroup2 的文件组，其中包含文件 SGrp2Fi1 和 SGrp2Fi2。

```

CREATE DATABASE Sales
ON PRIMARY
    ( NAME = SPri1_dat,
    FILENAME = 'c:\SPri1dat.mdf',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 15% ),
    ( NAME = SPri2_dat,
    FILENAME = 'c:\SPri2dt.ndf',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 15% ),
FILEGROUP SalesGroup1
    ( NAME = SGrp1Fi1_dat,
    FILENAME = 'c:\SG1Fi1dt.ndf',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 5 ),
    ( NAME = SGrp1Fi2_dat,
    FILENAME = 'c:\SG1Fi2dt.ndf',

```

```

        SIZE = 10,
        MAXSIZE = 50,
        FILEGROWTH = 5 ),
    FILEGROUP SalesGroup2
    ( NAME = SGrp2Fi1_dat,
      FILENAME = 'c:\SG2Fi1dt.ndf',
      SIZE = 10,
      MAXSIZE = 50,
      FILEGROWTH = 5 ),
    ( NAME = SGrp2Fi2_dat,
      FILENAME = 'c:\SG2Fi2dt.ndf',
      SIZE = 10,
      MAXSIZE = 50,
      FILEGROWTH = 5 )
LOG ON
( NAME = Sales_log,
  FILENAME = 'c:\salelog.ldf',
  SIZE = 5MB,
  MAXSIZE = 25MB,
  FILEGROWTH = 5MB );

```

5. 创建数据库并指定排序规则名称

下面的示例将创建数据库 `MyOptionsTest`，并将排序规则指定为 `French_CI_AI`。

```

CREATE DATABASE MyOptionsTest
COLLATE French_CI_AI;

```

可以使用以下语句验证数据库的选项设置：

```

SELECT name, collation_name
FROM sys.databases
WHERE name = N'MyOptionsTest';

```

在创建数据库后，应当备份 `master` 数据库。因为创建数据库将更新 `master` 中的系统表。如果 `master` 需要还原，则从上次备份 `master` 之后新建的所有数据库都将仍然在系统表中有引用，因而可能导致出现错误信息。

2.1.4 判断数据库是否已经存在

可以使用 `DB_ID` 函数判断数据库是否已经存在，该函数用于返回数据库的标识号，如果标识号不为空，则表示数据库已经存在。例如，下面的语句返回 `AdventureWorks` 数据库的标识号。

```

SELECT DB_ID(N'AdventureWorks') AS [Database ID];
GO

```

如果未指定数据库名称参数，则返回当前数据库的标识号。例如，下面的语句返回当前数据库 `master` 的标识号。

```

USE master; -- 切换到 master 数据库
GO
SELECT DB_ID() AS [Database ID]; -- 返回当前数据库的标识号
GO

```

实际上, 在创建示例数据库时经常用到该函数。例如, 下面的语句首先判断要创建的 **mytest** 数据库是否已经存在, 如果存在则先删除掉, 然后再新建数据库。

```
USE master;
GO
IF DB_ID (N'mytest') IS NOT NULL
DROP DATABASE mytest;
GO
CREATE DATABASE mytest;
GO
```

2.2 修改数据库

创建数据库后, 可以对其原始定义进行更改, 如扩展或收缩数据库、设置数据库选项等。要修改数据库。修改数据库可以使用 **ALTER DATABASE** 等语句。

2.2.1 扩展数据库和文件

默认情况下, 服务器可根据创建数据库时定义的增长参数自动扩展数据库。也可以通过为现有数据库文件分配更多空间, 或者创建新文件来手动扩展数据库。如果未将数据库设置为自动增长或硬盘上没有足够的磁盘空间的情况下, 数据库已经用完分配给它的空间且不能自动增长, 会出现 1105 错误。

扩展数据库时, 必须使数据库的大小至少增加 1 MB。如果扩展了数据库, 则根据被扩展的文件, 数据文件或事务日志文件将可以立即使用新空间。扩展数据库时, 应指定允许文件增长到的最大大小。这样可防止文件无限制地增大, 以致于用尽整个磁盘空间。

可以使用 **ALTER DATABASE** 语句设置数据库大小或向数据库添加文件, 其语法格式如下:

```
ALTER DATABASE database_name
ADD FILE <filespec> [ , ...n ]
    [ TO FILEGROUP { filegroup_name | DEFAULT } ]
| ADD LOG FILE <filespec> [ , ...n ]
| REMOVE FILE logical_file_name
| MODIFY FILE <filespec>
```

其中<filespec>部分用于设置文件组的属性, 语法格式如下:

```
(
    NAME = logical_file_name
    [ , NEWNAME = new_logical_name ]
    [ , FILENAME = 'os_file_name' ]
    [ , SIZE = size [ KB | MB | GB | TB ] ]
    [ , MAXSIZE = { max_size [ KB | MB | GB | TB ] | UNLIMITED } ]
    [ , FILEGROWTH = growth_increment [ KB | MB | GB | TB | % ] ]
    [ , OFFLINE ]
)
```

其中的 **OFFLINE** 选项用于将文件设置为脱机并使文件组中的所有对象都不可访问, 仅在文

件已损坏但可以还原时，才能使用该选项。其他参数选项请参考前面 CREATE DATABASE 语句中说明。

例如，下面的语句用于将 Sales 中的 SPri1_dat 文件扩展到 15MB，并将最大值设置为 25MB。

```
ALTER DATABASE Sales
MODIFY FILE
(
    NAME = 'SPri1_dat',
    SIZE = 15MB,
    MAXSIZE = 25MB
)
```

下面的语句向 SalesGroup1 文件组中添加一个 SGrp1Fi3_dat 文件。

```
ALTER DATABASE Sales
ADD FILE
(
    NAME = SGrp1Fi3_dat,
    FILENAME = 'c:\SG1Fi3dt.ndf',
    SIZE = 5MB,
    MAXSIZE = 10MB,
    FILEGROWTH = 5MB
)
TO FILEGROUP SalesGroup1 ;
```

执行下面的语句则可以删除上面添加的 SGrp1Fi3_dat 文件。

```
ALTER DATABASE Sales
REMOVE FILE SGrp1Fi3_dat ;
```

2.2.2 向数据库中添加、删除和修改文件组

下面是使用 ALTER DATABASE 语句向数据库中添加、删除和修改文件组时的语法格式：

```
ALTER DATABASE database_name
    ADD FILEGROUP filegroup_name
    | REMOVE FILEGROUP filegroup_name
    | MODIFY FILEGROUP filegroup_name
        { <filegroup_updatability_option>
        | DEFAULT
        | NAME = new_filegroup_name
        }
<filegroup_updatability_option>部分的语法格式如下：
{ READONLY | READWRITE } | { READ_ONLY | READ_WRITE }
```

例如，下面的语句用于向 Sales 数据库中添加一个名为 SalesGroup3 的文件组。

```
ALTER DATABASE Sales
ADD FILEGROUP SalesGroup3 ;
```

下面的语句重命名文件组 SalesGroup3 为 SalesGroup4。

```
ALTER DATABASE Sales
MODIFY FILEGROUP SalesGroup3
NAME = SalesGroup4 ;
```


2.2.3 收缩数据库和文件

可以使用 `DBCC SHRINKDATABASE` 语句或 `DBCC SHRINKFILE` 语句来手动收缩数据库或数据库中的文件。数据库中的每个文件都可以通过删除未使用的页的方法来减小。尽管数据库引擎会有效地重新使用空间，但某个文件多次出现无需原来大小的情况后，收缩文件就变得很有必要了。可以成组或单独地手动收缩数据库文件，也可以设置数据库的 `AUTO_SHRINK` 选项为 `ON` 来指定按间隔自动收缩。

文件始终从末尾开始收缩。例如，如果有一个 5GB 的文件，并且在 `DBCC SHRINKFILE` 语句中指定为 4GB，则数据库引擎将从文件的最后一个 1GB 开始释放尽可能多的空间。如果文件中被释放的部分包含使用过的页，则数据库引擎先将这些页重新放置到文件的保留部分。只能将数据库收缩到没有剩余的可用空间为止。例如，如果某个 5GB 的数据库有 4GB 的数据，并且在 `DBCC SHRINKFILE` 语句中指定为 3GB，则只能释放 1GB。

在使用 `DBCC SHRINKDATABASE` 语句时，无法将整个数据库收缩得比其初始大小更小。例如，如果数据库创建时的大小为 10MB，后来增长到 100MB，则该数据库最小只能收缩到 10MB，即使已经删除数据库的所有数据也是如此。

但是，使用 `DBCC SHRINKFILE` 语句时，可以将各个数据库文件收缩得比其初始大小更小。必须对每个文件分别进行收缩，而不能尝试收缩整个数据库。

1. 手动收缩数据库

下面是 `DBCC SHRINKDATABASE` 语句的语法格式：

```
DBCC SHRINKDATABASE
( 'database_name' | database_id | 0
  [ ,target_percent ]
  [ , { NOTRUNCATE | TRUNCATEONLY } ]
)
[ WITH NO_INFOMSGS ]
```

● `'database_name' | database_id | 0`：要收缩的数据库的名称或 ID。如果指定 0，则使用当前数据库。

● `target_percent`：数据库收缩后的数据库文件中所需的剩余可用空间百分比，等于图 4-4 中的“收缩后文件中的最大可用空间”值。

● `NOTRUNCATE`：指定在数据库文件中保留所释放的文件空间。如果未指定，将所释放的文件空间释放给操作系统。

● `TRUNCATEONLY`：将数据文件中任何未使用空间释放给操作系统，并将文件收缩到最后分配的区，从而无需移动任何数据即可减小文件大小。使用 `TRUNCATEONLY` 时，将忽略 `target_percent` 设置。

- **WITH NO_INFOMSGS**: 取消严重级别从 0~10 的所有信息性消息。

下面的语句使 Sales 数据库中文件有 10% 的可用空间。

```
DBCC SHRINKDATABASE ('Sales', 10)
```

2. 使用 ALTER DATABASE 设置自动收缩数据库

将数据库的 **AUTO_SHRINK** 选项设置为 ON 后, 数据库引擎将自动收缩有可用空间的数据库。有关使用 Management Studio 设置数据库选项的方法参考 4.1.1 小节介绍。下面是使用 **ALTER DATABASE** 语句将 Sales 数据库的 **AUTO_SHRINK** 选项设置为 ON 的方法。

```
ALTER DATABASE Sales
SET AUTO_SHRINK ON ;
```

3. 收缩文件

下面是 **DBCC SHRINKFILE** 语句的语法格式:

```
DBCC SHRINKFILE
(
    { 'file_name' | file_id }
    { [ , EMPTYFILE ]
    | [ [ , target_size ] [ , { NOTRUNCATE | TRUNCATEONLY } ] ] }
)
[ WITH NO_INFOMSGS ]
```

- **'file_name'**: 要收缩的文件的逻辑名称。
- **file_id**: 要收缩的文件的标识 (ID) 号。可以使用 **FILE_ID** 函数获取文件的 ID。
- **target_size**: 用兆字节表示的文件大小。如果未指定, 则 **DBCC SHRINKFILE** 将文件大小减少到默认文件大小。
- **EMPTYFILE**: 将指定文件中的所有数据迁移到同一文件组中的其他文件。
- **NOTRUNCATE**: 将释放的文件空间保留在文件中。当与 **target_size** 一起指定 **NOTRUNCATE** 时, 释放的空间不会释放给操作系统。唯一影响是将已使用的页从 **target_size** 行前面重新定位到文件的前面。
- **TRUNCATEONLY**: 将文件中的任何未使用空间释放给操作系统, 并将文件收缩到最后一次分配的区, 从而减小了文件大小, 但是没有移动任何数据。不会尝试将行重新定位到未分配的页。使用 **TRUNCATEONLY** 时, 将忽略 **target_size**。
- **WITH NO_INFOMSGS**: 禁止显示所有信息性消息。

例如, 下面的语句将 Sales 数据库中的 **SPri1_dat** 文件的大小收缩到 8MB。

```
USE Sales ;
GO
DBCC SHRINKFILE (SPri1_dat, 8) ;
```

下示例演示了清空文件以便从数据库中将其删除的步骤。针对此示例, 首先创建一个数据文件,

并假设该文件包含数据。

```
USE AdventureWorks;
GO
-- 创建一个数据文件并假设其包含数据
ALTER DATABASE AdventureWorks
ADD FILE (
    NAME = Test1data,
    FILENAME = 'C:\t1data.ndf',
    SIZE = 5MB
);
GO
-- 清空数据文件
DBCC SHRINKFILE (Test1data, EMPTYFILE);
GO
-- 从数据库中移除数据文件
ALTER DATABASE AdventureWorks
REMOVE FILE Test1data;
GO
```

2.2.4 设置数据库选项

可以在新建数据库时或对现有数据库通过“数据库属性”窗口进行部分数据库选项设置，而使用 ALTER DATABASE 的 SET 子句则可以进行更加全面的选项设置。可用的设置选项如表 2-1 所示。

表 2-1

可用的数据库设置选项

选 项	说 明
自动选项	
AUTO_CLOSE	设置为 ON 时，数据库将在最后一个用户退出后完全关闭，它占用的资源也将释放。当用户尝试再次使用该数据库时，该数据库将自动重新打开。设置为 OFF 时，最后一个用户退出后数据库仍保持打开。对于 SQL Server Desktop Engine 或 SQL Server Express 的数据库，默认设置为 ON，其他版本默认为 OFF
AUTO_CREATE_STATISTICS	设置为 ON 时，将自动创建谓词所使用的列的统计信息；设置为 OFF 时，需要手动创建统计信息。默认值为 ON
AUTO_UPDATE_STATISTICS	设置为 ON 时，优化查询所需的任何缺少的统计信息将在查询优化过程中自动生成；设置为 OFF 时，统计信息必须手动创建。默认值为 ON
AUTO_SHRINK	设置为 ON 时，数据库文件可作为定期收缩的对象；设置为 OFF 时，在定期检查未使用空间的过程中，数据库文件不自动收缩。默认值为 OFF
游标选项	
CURSOR_CLOSE_ON_COMMIT	设置为 ON 时，所有打开的游标都将在提交或回滚事务时关闭；设置为 OFF 时，打开的游标将在提交事务时仍保持打开，回滚事务将关闭所有游标，但定义为 INSENSITIVE 或 STATIC 的游标除外。默认值为 OFF
CURSOR_DEFAULT	如果指定了 LOCAL，并且创建游标时没有将其定义为 GLOBAL，则游标的作用域将局限于创建游标时所在的批处理、存储过程或触发器。游标名仅在该作用域内有效。 如果指定了 GLOBAL，并且创建游标时没有将其定义为 LOCAL，则游标的作用域将是相应连接的全局范围。在由连接执行的任何存储过程或批处理中，都可以引用该游标名称。默认值为 GLOBAL

续表

选 项	说 明
数据库可用性选项	
OFFLINE ONLINE EMERGENCY	指定为 OFFLINE 时，数据库将完全关闭和退出，并标记为脱机；指定为 ONLINE 时，数据库处于打开状态并且可供使用；指定为 EMERGENCY 时，数据库将标记为 READ_ONLY，日志记录将被禁用，并且只有 sysadmin 固定服务器角色的成员才能进行访问。默认值为 ONLINE
READ_ONLY READ_WRITE	指定为 READ_ONLY 时，用户可以从数据库中读取数据，但不能修改它；指定为 READ_WRITE 时，可对数据库进行读写操作。默认值为 READ_WRITE
SINGLE_USER RESTRICTED_USER MULTI_USER	指定为 SINGLE_USER 时，一次只允许一个用户连接到数据库；指定为 RESTRICTED_USER 时，只允许 db_owner 固定数据库角色的成员以及 dbcreator 和 sysadmin 固定服务器角色的成员连接到数据库，不过对连接数没有限制；指定为 MULTI_USER 时，允许所有具有相应权限的用户连接到数据库。默认值为 MULTI_USER
日期相关性优化选项	
DATE_CORRELATION_OPTIMIZATION	指定为 ON 时，服务器将维护数据库中所有由 FOREIGN KEY 约束链接的包含 datetime 列的两个表中的相关统计信息；指定为 OFF 时，不会维护相关统计信息。默认值为 OFF
外部访问选项	
DB_CHAINING	指定为 ON 时，数据库可以是跨数据库所有权链的源或目标；指定为 OFF 时，数据库不能参与跨数据库的所有权链接。默认值为 OFF
TRUSTWORTHY	设置为 ON 时，使用了模拟上下文的数据库模块（例如，用户定义函数或存储过程）可以访问数据库以外的资源；指定为 OFF 时，在模拟上下文中无法访问数据库以外的资源。默认值为 OFF
参数化选项	
PARAMETERIZATION	指定为 SIMPLE 时，将根据数据库的默认行为参数化查询；指定为 FORCED 时，将参数化数据库中所有的查询。默认值为 SIMPLE
恢复选项	
RECOVERY	指定为 FULL 时，将使用事务日志备份在发生媒体故障后进行完全恢复。如果数据文件损坏，媒体恢复可以还原所有已提交的事务。指定为 BULK_LOGGED 时，将综合某些大规模或大容量操作的最佳性能和日志空间的最少占用量，在发生媒体故障后进行恢复。指定为 SIMPLE 时，将提供占用最小日志空间的简单备份策略。默认值为 FULL
PAGE_VERIFY	指定为 CHECKSUM 时，数据库引擎将在页写入磁盘时计算整个页的内容的校验和并存储页头中的值。从磁盘中读取页时，将重新计算校验和，并与存储在页头中的校验和值进行比较。指定为 TORN_PAGE_DETECTION 时，在将 8KB 的数据库页写入磁盘时，该页的每个 512 字节的扇区都有一个特定的位保存并存储在数据库的页头中。从磁盘中读取页时，页头中存储的残缺位将与实际的页扇区信息进行比较。指定为 NONE 时，数据库页写入将不生成 CHECKSUM 或 TORN_PAGE_DETECTION 值。默认值为 CHECKSUM
Service Broker 选项	
ENABLE_BROKER DISABLE_BROKER NEW_BROKER ERROR_BROKER_CONVERSATIONS	指定为 ENABLE_BROKER 时，将为指定数据库启动 Service Broker；指定为 DISABLE_BROKER 时，将对指定的数据库禁用 Service Broker；指定为 NEW_BROKER 时，数据库将收到新的代理标识符；指定为 ERROR_BROKER_CONVERSATIONS 时，数据库中的会话将在附加数据库时收到一个错误消息。默认值为 DISABLE_BROKER

续表

选 项	说 明
快照隔离选项	
ALLOW_SNAPSHOT_ISOLATION	指定为 ON 时, 事务可以指定 SNAPSHOT 事务隔离级别。当事务在 SNAPSHOT 隔离级别运行时, 所有的语句都将数据快照视为位于事务的开头。指定为 OFF 时, 事务无法指定 SNAPSHOT 事务隔离级别。默认值为 OFF
READ_COMMITTED_SNAPSHOT	指定为 ON 时, 指定 READ_COMMITTED 隔离级别的事务将使用行版本控制而不是锁定。当事务在 READ_COMMITTED 隔离级别运行时, 所有的语句都将数据快照视为位于语句的开头。指定为 OFF 时, 指定 READ_COMMITTED 隔离级别的事务将使用锁定。默认值为 OFF
SQL 选项	
ANSI_NULL_DEFAULT	指定在 CREATE TABLE 或 ALTER TABLE 语句中未显式定义为空性的别名数据类型或 CLR 用户自定义类型列的默认值 (NULL 或 NOT NULL)。当指定为 ON 时, 默认值为 NULL; 当指定为 OFF 时, 默认值为 NOT NULL。默认值为 OFF
ANSI_NULLS	指定为 ON 时, 所有与空值的比较运算计算结果为 UNKNOWN; 指定为 OFF 时, 非 UNICODE 值与空值的比较运算在两者均为 NULL 时结果为 TRUE。默认值为 OFF
ANSI_PADDING	指定为 ON 时, 在出现如除以零或聚合函数中出现空值这类情形时, 将发出错误或警告; 指定为 OFF 时, 在出现如除以零这类情形时, 不会发出警告, 并返回空值。默认值为 OFF
ARITHABORT	指定为 ON 时, 在执行查询期间发生溢出或除以零的错误时, 该查询将结束; 指定为 OFF 时, 出现其中一个错误时将显示警告信息, 而查询、批处理或事务将继续处理, 就像没有出现错误一样。默认值为 OFF
CONCAT_NULL_YIELDS_NULL	指定为 ON 时, 如果串联操作的两个操作数中任意一个为 NULL, 则结果也为 NULL; 当指定为 OFF 时, 空值将按空字符串对待。默认值为 OFF
QUOTED_IDENTIFIER	指定为 ON 时, 双引号可用来将分隔标识符括起来; 指定为 OFF 时, 标识符不能用引号括起来, 而且必须遵循所有用于标识符的 Transact-SQL 规则。默认值为 OFF
NUMERIC_ROUNDABORT	指定为 ON 时, 表达式中出现失去精度时将产生错误; 指定为 OFF 时, 失去精度不生成错误信息, 并且将结果舍入到存储结果的列或变量的精度。默认值为 OFF
RECURSIVE_TRIGGERS	指定为 ON 时, 允许递归激发 AFTER 触发器; 指定为 OFF 时, 仅不允许直接递归激发 AFTER 触发器。默认值为 OFF
补充日志记录	
SUPPLEMENTAL_LOGGING	指定为 ON 时, 会将详细信息添加到第三方产品的日志中; 指定为 OFF 时, 则不将详细信息添加到日志中 默认值为 OFF

例如, 下面的语句设置 Sales 数据库的 ANSI_NULLS 和 ANSI_NULL_DEFAULT 选项为 ON。

```
ALTER DATABASE Sales
SET ANSI_NULLS ON,ANSI_NULL_DEFAULT ON ;
```

2.2.5 重命名数据库

在重命名数据库之前，应该确保没有人使用该数据库，而且该数据库设置为单用户模式。

下面是使用 ALTER DATABASE 语句重命名数据库时的语法格式：

```
ALTER DATABASE database_name
MODIFY NAME = new_database_name ;
```

例如，下面语句将 Sales 数据库重命名为 Sales1。

```
ALTER DATABASE Sales
SET SINGLE_USER; --设置为单用户
GO

ALTER DATABASE Sales
MODIFY NAME = Sales1; --重命名为 Sales1
GO

ALTER DATABASE Sales1 --重新设置为多用户
SET MULTI_USER;
```

2.3 删除数据库

在数据库删除之后，文件及其数据都将从服务器磁盘中删除。在删除数据库时，可以不用管数据库所处的状态（包括脱机、只读和可疑），但是应当满足下列前提条件。

- 如果数据库涉及日志传送操作，在删除数据库之前应当取消日志传送操作。
- 要删除为事务复制发布的数据库，或删除为合并复制发布或订阅的数据库，应当首先从数据库中删除复制。如果数据库已损坏，不能首先删除复制，则通常仍然可以通过首先使用 ALTER DATABASE 将数据库设置为脱机然后再删除的方法来删除数据库。
- 必须首先删除数据库上存在的数据库快照。

在删除数据库后，应备份 master 数据库，因为删除数据库将更新 master 数据库中的信息。

可以使用 DROP DATABASE 语句删除数据库，其语法格式如下：

```
DROP DATABASE { database_name | database_snapshot_name } [ ...,n ]
```

- database_name: 指定要删除的数据库的名称。
- database_snapshot_name: 指定要删除的数据库快照的名称。

例如，下面的语句将删除 Sales 数据库。

```
DROP DATABASE Sales ;
```



第3章 表管理

我们的业务数据被存储在不同功能的表中，然后查询围绕着这些表而展开。表设计（或者说是业务数据分布）是否合理，对于后期的查询设计起着关键作用。抛开表设计而单纯讨论查询优化，某些时候会有点舍本求末的味道。例如，如果一个表的数据量非常大，你应当将数据分割成多个表存储，或是使用分区表技术，不要以为完全依靠升级硬件或创建索引能够解决多大的问题。从一个上百 GB 的表中检索数据，远不如将其分割成 1GB 的表检索数据快得多。

逻辑上，表由行和列组成，并且每列具有一个系统数据类型或用户定义数据类型。物理上，表具有两种数据存储单位：数据页和区。数据页是基本的数据存储单位，而区则是由八个物理上连续的数据页组成的集合，可以用来有效地管理数据页。本章将介绍表的物理存储方式以及表的创建、修改和删除方法。

3.1 表的物理存储方式

作为一本以讲解 SQL 查询为主的书籍，之所以要介绍表的物理存储方式，这是因为在了解了该原理之后，对于理解查询优化知识可以起到一些帮助作用。

3.1.1 最基本的数据存储单位：数据页

除大型对象数据类型之外的所有数据类型，表数据都存储在数据页中。在 SQL Server 中，页的大小为 8KB（8192 字节），这意味着每 MB 将有 128 个页。每个数据页中包含有页标头、数据行和用于包含行偏移量的可用空间，数据行紧接着标头按顺序放置，页的末尾是行偏移表。

所谓偏移量，也就是用于记录数据行的第 1 个字节与页标头的距离值，其作用是区分页中的各个数据行。所以对于页中的每一行，偏移表中都包含一个与之对应的偏移量条目。在检索特定数据行时，会直接根据偏移量指定的字节位置开始读取。行偏移表中的条目的顺序与页中行的顺序相反，如图 3-1 所示。

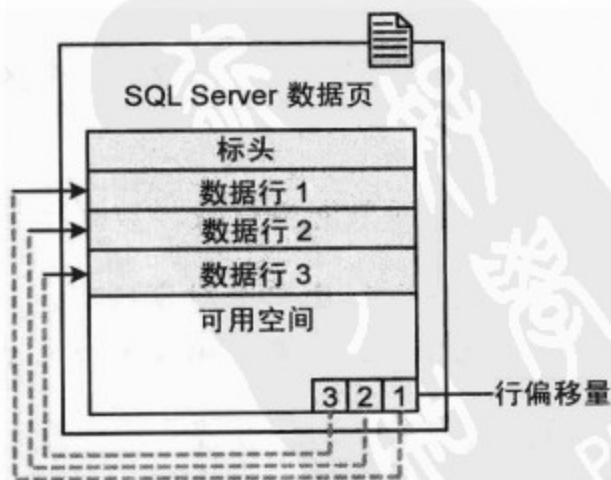


图 3-1 数据页的结构

页标头使用每页开头的 96 个字节，用于存储有关页的系统信息（包括页码、页类型、页的可用空间以及拥有该页的对象的分配单元 ID）。剩余的 8096 字节用于存储数据行和行偏移。

一个表中的每一行最多可以包含 8060 字节。如果表中包含了 text 或 image 数据，则文本和图像很可能不能够被存储在一行中。由于数据引擎保持每页 8KB 的限制，因此，当列超过此限制时，数据库引擎将把页中最大宽度的记录列移动到另一页上，而在原始页上保留一个 24 字节指针，指向实际的数据存储位置。如果更新操作使记录变短，记录可能会移回到原始页中。

SQL Server 支持 6 种类型的数据页，如表 3-1 所示。

表 3-1 数据页的类型

页 类 型	内 容
Data	包含除 nvarchar(max)、varchar(max)、varbinary(max)和 xml 数据之外的，以及当 text in row 设置为 ON 时的 text、ntext、image 数据之外的所有数据的数据行
Index	包含索引条目
Text/Image	大型对象数据类型。包含 text、ntext、image、nvarchar(max)、varchar(max)、varbinary(max)和 xml 数据，以及数据行超过 8KB 时的可变长度列（数据类型包括 varchar、nvarchar、varbinary 和 sql_variant）
Global Allocation Map、Shared Global Allocation Map	包含区是否已被分配的信息
Page Free Space	包含页中可用空间的信息
Index Allocation Map	包含每个分配单元中表或索引所使用的区的信息
Bulk Changed Map	包含每个分配单元中自最后一条 BACKUP LOG 语句之后的大容量操作所修改的区的信息
Differential Changed Map	包含每个分配单元中自最后一条 BACKUP DATABASE 语句之后更改的区的信息

注意：日志文件不包含页，而是包含一系列日志记录。

3.1.2 最基本的管理空间单位：区

区是管理空间的基本单位。一个区是 8 个物理上连续的页（即 64KB）。这意味着数据库中每 MB 有 16 个区。

为了使空间分配更有效，数据库引擎不会将所有区分配给包含少量数据的表。区有两种类型：

- 统一区。由单个对象所有，区中的所有 8 页只能由所属对象使用。
- 混合区。最多可由 8 个对象共享，区中的每页可由不同的对象所有。

通常从混合区向新表或索引分配页。当表或索引增长到 8 页时，将变成使用统一区进行后续分配。如果对现有表创建索引，并且该表包含的行足以在索引中生成 8 页，则对该索引的所有分

配都使用统一区进行。

3.2 创建表

在一个数据库中，可以创建多达 20 亿个表。表名称最大可以达到 128 个字符长度。表名称在数据库架构中必须是唯一的，但是在不同的架构之间，表名称可以相同。一个表中最多可有 1024 个列。

可以使用 CREATE TABLE 语句创建表。由于该语句的语法格式比较复杂，其中涉及表所属的架构、键约束、表分区等，本节我们将通过示例的方式介绍一些常用的表创建方法。表架构属于服务器安全方面的内容，在本书中不作讲述。

3.2.1 创建基本表

下面的语句将创建一个名为 Orders 的表，其中包含 3 列，分别是 int 类型的 OrderID 列、datetime 类型的 OrderDate 列和 varchar 类型的 SendTo 列。

```
CREATE TABLE Orders
(
    OrderID int,
    OrderDate datetime,
    SendTo varchar(40)
);
```

3.2.2 使用允许和禁止空值设置进行值约束

在表中创建列时，可以指定是否允许为空值。空值表示未向该列的当前行中输入内容，它不同于 0 或空字符串。以主键约束或标识定义的列不允许空值。

如果列允许空值，向表中添加了一个行，但是未给列指定值，数据库引擎将自动为列插入空值，除非为该列指定了默认值设置。当为列设置了默认值并插入一个空值时，会自动使用默认值替换空值。此外，如果列允许为空值，可以显式地使用 NULL（不要使用引号）关键字设置一个列为空值。

例如，下面创建的 Orders 表中所包含的 3 列均不允许有空值。

```
CREATE TABLE Orders
(
    OrderID int NOT NULL,
    OrderDate datetime NOT NULL,
    SendTo varchar(40) NOT NULL
);
```

3.2.3 使用标识符和全球唯一标识符创建唯一值

在设计表时，通常需要考虑使用唯一标识符来作为主键，或是用来确保被添加的数据不会与现存数据存在冲突。对于主键唯一标识符，可以包含客户账号或社会保障号，但是，如果某个唯一标识符不可用，你可能希望能够使用“标识”属性为表中的每行生成一个唯一序列号。

例如，下面创建的 **Orders** 表包含一个标识符列 **OrderID**，种子值为 1，增量值为 1。当向表中插入数据时，**OrderID** 列的数据会自动插入并递增。同时，为表创建了一个基于 **OrderID** 列的 **PRIMARY KEY** 约束。

```
CREATE TABLE Orders
(
    OrderID int IDENTITY(1,1) PRIMARY KEY,
    OrderDate datetime,
    Sendto varchar(40)
);
```

向表中插入两行数据，由于为 **OrderID** 指定了 **IDENTITY** 属性，因此在插入时可以不指定值。插入后的结果如表 3-2 所示。

```
INSERT INTO Orders (OrderDate, Sendto)
VALUES ('2008-1-1', 'Jinan City, China'),
('2008-2-19', 'Beijing City, China');
```

表 3-2 Orders 表中的内容

OrderID	OrderDate	Sendto
1	2008-01-01 00:00:00.000	Jinan City, China
2	2008-02-19 00:00:00.000	Beijing City, China

上面通过 **IDENTITY** 属性为 **OrderID** 列实现了自动编号，但这是一种局部解决方案，各自具有标识符列的各个表会生成相同的值，这是因为 **IDENTITY** 属性仅在使用它的表上保证是唯一的。如果应用程序生成一个标识符列，并且该列在整个数据库或全球联网的所有计算机上的所有数据库中必须是唯一的，则需要为列指定 **uniqueidentifier** 数据类型，并使用 **NEWID()** 或 **NEWSEQUENTIALID()** 函数指定默认值。下面创建的 **Cutomers** 表中的 **CustID** 列是一个全球唯一标识符（Globally Unique Identifier, GUID）列，默认值通过 Transact-SQL 的 **NEWID()** 函数获得。

```
CREATE TABLE Customers
(
    LastName varchar(40),
    FirstName varchar(20),
    Phone char(12),
    CustID uniqueidentifier DEFAULT NEWID()
);
```

向表中插入两行数据，由于为 **CustID** 列设置了默认值，因此在插入时可以不指定该列的值，而是由 **NEWID()** 函数自动生成。插入后的结果如表 3-3 所示。

```
INSERT INTO Customers (LastName, FirstName, Phone)
VALUES ('Zhang', 'Hongju', '021-58968888'),
('Shi', 'Sharelly', '010-89963419');
```

表 3-3 Customers 表中的内容

LastName	FirstName	Phone	CustID
Zhang	Hongju	021-58968888	17626825-4C3E-4879-9D58-8F6169C897BB
Shi	Sharelly	010-89963419	A35CB093-F909-429D-8849-42C51C4E912B

标识和全球唯一标识符并不是相互排斥的，每个表中可以同时具有标识列和全球唯一标识符列，这些值通常一起使用。例如，下面的 MyTable 表包含了这两种类型的标示列。

```
CREATE TABLE dbo.MyTable(
    MyIdentity int IDENTITY(1,1) NOT NULL, --标识列，标识种子和增量都为 1
    MyGUI uniqueidentifier ROWGUIDCOL NOT NULL DEFAULT NEWID() --全球唯一标识符列
)
```

3.2.4 为列指定默认值

在不知道值或在值丢失的情况下，允许空值是非常使用的。但是，某些时候空值是容易引起争议的，一种更好的办法就是使用默认值。默认值用于在向表中插入行并且未给列指定值时。例如，可以在一个基于字符的列设置默认值为 N/A，而不是在允许为空的情况下自动插入 NULL。

表 3-4 总结出了默认值和为空性相结合时的不同处理方法。最需要注意的是：如果为列设置了默认值，在未为列指定值的情况下，将使用默认值。

表 3-4 默认值和为空性相结合的处理方法

列 定 义	未输入值，无默认值定义	未输入值，有默认值定义	输入了一个空值
允许空值	设置为 NULL	设置为默认值	设置为 NULL
禁止空值	发生错误	设置为默认值	发生错误

下面的创建的 Customers 表中，为 Phone 列指定了默认值“未输入”。当未为该列指定插入值时，将使用默认值。

```
CREATE TABLE Customers
(
    FirstName varchar(20) NOT NULL,
    LastName varchar(40) NOT NULL,
    Phone char(12) DEFAULT '未输入',
);
```

3.3 修改表

当表被创建后，在使用过程中可能会出现一些新的需求，这时候可能需要修改表的结构。如果表中已经填充了数据，重新建表会造成现有数据的丢失，为此，可以使用 ALTER TABLE 语句对表

结构进行修改。

3.3.1 为表添加新列

向表中添加列的前提是所添加列允许使用 NULL 值或者对该列使用 DEFAULT 约束指定了默认值。向一个表添加新列时，数据库引擎会在该列中为表中的每个现有数据行插入一个值。因此，在向表中添加列时为列指定 DEFAULT 定义会很有用。如果新列没有 DEFAULT 定义，则必须指定该列允许 NULL 值，数据库引擎将 NULL 值插入该列。如果新列不允许 NULL 值，则返回错误。

例如，下面的语句向 Customers 表添加一个名为 Email 的列，列的数据类型为 VARCHAR，并允许 NULL 值。

```
ALTER TABLE Customers
ADD Email varchar(50) NULL;
```

如果需要添加一个非空约束，则必须为列指定一个默认值，参考下面的语句：

```
ALTER TABLE Customers
ADD Email varchar(50) NOT NULL DEFAULT '未知邮箱';
```

当向表中添加多个列时，列之间可以使用逗号分隔。例如，下面的语句向 Customers 表中添加了 Email 和 Phone 列。

```
ALTER TABLE Customers
ADD Email varchar(50) NOT NULL DEFAULT '未知邮箱',
    Phone char(20);
```

3.3.2 修改表中的列

对于表中的现有列，可以更改列的名称、约束条件和数据类型等。

1. 修改列的名称

更改列的名称应当使用系统存储过程 sp_rename，而不是 ALTER TABLE 语句。sp_rename 用于修改当前数据库中用户所创建对象的名称，包括表、索引、列、别名数据类型或 Microsoft .NET Framework 公共语言运行 (CLR) 时用户定义数据类型。

sp_rename 的语法格式如下：

```
sp_rename [ @objname = ] 'object_name' , [ @newname = ] 'new_name' [ , [ @objtype = ] 'object_type' ]
```

● [@objname =] 'object_name' 用户对象或数据类型的当前限定或非限定名称。如果要重命名的对象是表中的列，则 object_name 的格式必须是 table.column。

● [@newname =] 'new_name' 指定对象的新名称。

● [@objtype =] 'object_type' 要重命名的对象的类型，可用值如表 3-5 所示。

表 3-5 被重命名对象的可用类型

值	说 明
COLUMN	要重命名的是列
DATABASE	要重命名的是用户定义数据库
INDEX	要重命名的是用户定义索引
OBJECT	要重命名的是约束、用户表和规则等
USERDATATYPE	要重命名的是别名数据类型或 CLR 用户定义数据类型

例如，下面的语句将 Customers 表的 Email 列重命名为 EmailNew。

```
EXEC sp_rename 'Customers.Email', 'EmailNew', 'COLUMN';
```

2. 修改列的数据类型

在更改列的数据类型时，以前的数据类型必须可以隐式转换为新数据类型。一般情况下，不能修改 text、ntext、image 或 GUID 列，不能修改计算或复制的列，不能修改引用计算列或者约束列。

例如，下面的语句将表中 column_a 列的数据类型由 INT 更改为 DECIMAL。

```
CREATE TABLE doc_exa ( column_a INT );
ALTER TABLE doc_exa ALTER COLUMN column_a decimal(5, 2);
```

3. 修改列的大小和为空性

ALTER TABLE 语句还允许改变列的大小和为空性。

在更改列的大小时，通常是由小向大改变。在由大向小改变时，在列中存在数据的情况下，应当设置一个适当的合理值，否则会造成数据被截断，丢失数据。

在修改列的为空性时，应当考虑各种约束条件。例如，假设当前是一个标示符列，则不能修改为允许空值。

下面的语句将 Customers 表中的 Email 列修改为 30 个字符长度。

```
ALTER TABLE Customers
ALTER COLUMN Email varchar(30);
```

3.3.3 删除表中的列

在删除一列之前，必须先删除任何引用该列的约束、默认值表达式、计算列表表达式或索引。当为列设置有默认值时，SQL Server 会为自动列创建约束，这些约束以 DF 开头，并且含有表的名称和列的名称，如 DF_Customers_Email_267ABA7A。参考下面的语句：

```
ALTER TABLE Customers
DROP CONSTRAINT DF_Customers_Email_267ABA7A; --删除约束
ALTER TABLE Customers
DROP COLUMN Email; --删除列
```

执行 `sp_help` 系统存储过程，可以列出表中的约束。参考下面的语句：

```
EXEC sp_help Customers
```

图 3-2 是执行 `sp_help` 后返回的结果信息。

Name	Owner	Type	Created_datetime
1 Customers	dbo	user table	2008-11-20 20:06:59.460

Column_name	Type	Computed	Length	Prec	Scale	Nullable	TrimTrailingBlanks	FixedLenNullInSource	Collation
1 CustID	char	no	5			no	no	no	Chinese_PRC_CI_AS
2 City	char	no	20			no	no	no	Chinese_PRC_CI_AS
3 Phone	char	no	20			yes	no	yes	Chinese_PRC_CI_AS

Identity	Seed	Increment	Not For Replication
1 No identity column defined.	NULL	NULL	NULL

RowGuidCol
1 No rowguidcol column defined.

Data_located_on_filegroup
1 PRIMARY

index_name	index_description	index_keys
1 PK_Customer_049E3A89145C0A3F	clustered, unique, primary key located on PRIMARY	CustID

constraint_type	constraint_name	delete_action	update_action	status_enabled	status_for_replication	constraint_keys
1 PRIMARY KEY (clustered)	PK_Customer_049E3A89145C0A3F	(n/a)	(n/a)	(n/a)	(n/a)	CustID

图 3-2 执行 `sp_help` 后返回的结果信息

3.4 重命名和删除表

对表进行重命名，可以使用在 3.3.2 小节中介绍的系统存储过程 `sp_rename`。例如，下面的语句将 `MyTable` 重命名为 `MyNewTable`。

```
EXEC sp_rename 'dbo.MyTable', 'MyNewTable'
```

可以使用 `DROP TABLE` 语句删除一个或多个表定义，以及这些表的所有数据、索引、触发器、约束和指定的权限。

对于被 `FOREIGN KEY` 约束引用的表，必须先删除引用 `FOREIGN KEY` 约束或引用表。如果要在同一个 `DROP TABLE` 语句中删除引用表以及包含主键的表，则必须先列出引用表。

删除表时，表的规则或默认值将被解除绑定，与该表关联的任何约束或触发器将被自动删除。如果要重新创建表，则必须重新绑定相应的规则和默认值，重新创建触发器，并添加所有必需的约束。

如果表包含带有 `FILESTREAM` 属性的 `VARBINARY (MAX)`列，即在表之外的文件系统中存

储文本文档、图像和视频这样的非结构化数据，在删除表时，不会删除在文件系统中存储的任何数据。

下面的语句将从数据库中删除 MyNewTable 表。

```
❏ DROP TABLE dbo.MyNewTable
```

如果仅仅是希望从表中删除所有行，并保留当前表的完整结构，可以使用 DELETE 语句。参考下面的语句：

```
❏ DELETE dbo.MyNewTable
```

3.5 临时表

在前面介绍的表都是永久表，实际上，临时表与永久表很相似，创建方法也相同。永久表存储在它所创建的数据库中，而临时表存储在 tempdb 数据库中。

3.5.1 创建本地表和全局表

临时表有两种类型：本地表和全局表。它们在名称、可见性以及可用性上有区别。本地临时表的名称以单个数字符号（#）打头；它们仅对当前的用户连接是可见的；当用户从 SQL Server 实例断开连接时被删除。全局临时表的名称以两个数字符号（##）打头，创建后对任何用户都是可见的，当所有引用该表的用户与服务器断开连接时被删除。

例如，如果创建了 Employees 表，则任何在数据库中有使用该表的安全权限的用户都可以使用该表，除非已将其删除。如果数据库会话创建了本地临时表 #Employees，则仅会话可以使用该表，会话断开连接后就将该表删除。如果创建了 ##Employees 全局临时表，则数据库中的任何用户均可使用该表。如果该表在创建后没有其他用户使用，则当断开连接时该表删除。如果用户创建该表后另一个用户在使用该表，则数据库引擎将在用户断开连接并且所有其他会话不再使用该表时将其删除。

例如，下面的语句创建了一个临时表，并向其中插入一行数据。

```
❏ CREATE TABLE #MyTempTable (cola int PRIMARY KEY);  
❏ INSERT INTO #MyTempTable VALUES (1);
```

如果在单个存储过程或批处理中创建了多个临时表，由于它们都存储在 tempdb 数据库中，所以它们必须有不同的名称。

当创建本地或全局临时表时，CREATE TABLE 语法支持除 FOREIGN KEY 约束以外的其他所有约束定义。如果临时表中指定了 FOREIGN KEY 约束，则该语句将返回一条表明已跳过此约束的警告消息。此表仍将创建，但不使用 FOREIGN KEY 约束。

3.5.2 使用表变量代替临时表

表变量实际上也是一种临时表，但是它们之间也有一定的区别。当必须对临时表显式地创建索引时，或多个存储过程或函数必须使用表值时，临时表很有用。但是，表变量通常可提供更有效的查询处理。

表变量的行为类似于局部变量，有明确定义的作用域，也就是声明该变量的函数、存储过程或批处理。表变量可应用于 SELECT、INSERT、UPDATE 和 DELETE 语句中用到表或表的表达式的任何地方。

下面的语句首先声明一个表变量@TableVar，然后插入了两行值，并使用 SELECT 语句从变量中检索数据。返回结果如表 3-6 所示。

```
DECLARE @TableVar TABLE (Cola int PRIMARY KEY, Colb char(3))
INSERT INTO @TableVar VALUES (1, 'abc')
INSERT INTO @TableVar VALUES (2, 'def')
SELECT * FROM @TableVar
```

表 3-6

由表变量返回的结果集

Cola	Colb
1	abc
2	def



第4章 索引管理

如果你的表数据日常更新比较少,应当使用索引技术,因为通过索引可以减少为返回查询结果集而必须读取的数据量。如果你的表经常需要执行插入、删除、修改操作,过多的索引则会导致数据维护性能的减弱,因为在维护这些数据的同时,数据库引擎还要维护这些索引。

可以在表、视图和列上创建索引。除了提高查询性能,索引还可以强制表中的行具有唯一性,从而确保表数据的完整性。

4.1 索引的基础知识

索引是一个单独的、物理数据存储结构,其中包含由表或视图中的一列或多列生成的键。这些键就像书籍的目录,可以通过检索键值来查找与之对应的数据表中的内容。仅检索键值比检索整个表可以有效地减少磁盘 I/O 操作,从而提高查询性能。

4.1.1 索引的类型

索引可以大致分为聚集索引、非聚集索引、全文索引、XML 索引和空间索引 5 种主要类型,此外还包括唯一索引、包含列索引、索引视图和筛选索引。

1. 聚集索引

聚集索引根据数据行的键值在表或视图中按排序顺序存储这些数据行,即表中数据行的物理顺序与索引顺序一致。由于数据行本身只能按一个顺序排序,所以每个表只能有一个聚集索引。如果表具有聚集索引,则该表称为聚集表。如果表没有聚集索引,则其数据行存储在一个称为堆的无序结构中。

每个表几乎都对列定义聚集索引来实现下列功能:

- 可用于经常使用的查询;
- 提供高度唯一性;
- 可用于范围查询。

2. 非聚集索引

非聚集索引具有独立于数据行的结构。非聚集索引包含的键值，并且每个键值项都有指向包含该键值的数据行的指针。从非聚集索引中的索引行指向数据行的指针称为行定位器。行定位器的结构取决于数据页是存储在堆中还是聚集表中。对于堆，行定位器是指向行的指针。对于聚集表，行定位器是聚集索引键。

通常情况下，设计非聚集索引是为改善经常使用的、没有建立聚集索引的查询的性能。

与使用书籍目录的方式相似，查询优化器在搜索数据值时，先搜索非聚集索引以找到数据值在表中的位置，然后直接从该位置检索数据。这使非聚集索引成为完全匹配查询的最佳选择，因为索引包含说明查询所搜索的数据值在表中的精确位置的项。例如，为了从 `HumanResources.Employee` 表中查询向特定经理负责的所有雇员，查询优化器可能使用非聚集索引 `IX_Employee_ManagerID`，该索引以 `ManagerID` 作为其键列。查询优化器能快速找出索引中与指定 `ManagerID` 匹配的所有项。每个索引项都指向表或聚集索引中准确的页和行，其中可以找到相应的数据。在查询优化器在索引中找到所有项之后，它可以直接转到准确的页和行进行数据检索。

3. 全文索引

全文索引是一种特殊类型的基于标记的功能性索引，它是由全文引擎生成和维护的。生成全文索引的过程不同于生成其他类型的索引。全文引擎并非基于特定行中存储的值来构造 B 树结构，而是基于要编制索引的文本中的各个标记来生成倒排、堆积且压缩的索引结构。

可以对包含 `char`、`varchar` 和 `nvarchar` 数据的列生成全文索引，也可以对包含格式化二进制数据（如存储在 `varbinary(max)` 或 `image` 列中的 Microsoft Word 文档）的列创建全文索引。

4. XML 索引

可以对 `xml` 数据类型列创建 XML 索引。它们对列中 XML 实例的所有标记、值和路径进行索引，从而提高查询性能。存在下列情况下时，可以考虑为 XML 创建索引。

- 对 XML 列的查询操作在工作负荷中很常见，可以考虑创建 XML 索引。但是，在提升查询性能的同时，也必须考虑数据修改时造成的 XML 索引维护开销。

- XML 值相对较大，而检索的部分相对较小，可以考虑创建 XML 索引。这样可以避免在运行时分析所有数据，并能实现高效的查询处理。

XML 索引可以是非聚集索引（默认值），也可以是聚集索引。每个表中最多可以包含 249 个 XML 索引。

5. 空间索引

SQL Server 2008 及更高版本支持空间数据。这包括对平面空间数据类型 `geometry` 的支持，该

数据类型支持欧几里得坐标系统中的几何数据（点、线和多边形）。geography 数据类型表示地球表面某区域上的地理对象，如一片陆地。geography 列的空间索引会将地理数据映射到二维非欧几里得空间。

空间索引是对包含空间数据的表列（“空间列”）定义的，每个空间索引指向一个有限空间。例如，geometry 列的索引指向平面上用户指定的矩形区域。

6. 唯一索引

唯一索引能够保证索引键中不包含重复的值，从而使表中的每一行从某种方式上具有唯一性。只有当唯一性是数据本身的特征时，指定唯一索引才有意义。如果用户尝试向索引列中输入相同值时，将显示错误消息并且不能输入重复的值。

使用多列唯一索引，索引能够保证索引键中值的每个组合都是唯一的。例如，如果为 LastName、FirstName 和 MiddleName 列的组合创建了唯一索引，则表中的任意两行都不会有这些列值的相同组合。

聚集索引和非聚集索引都可以是唯一的。只要列中的数据是唯一的，就可以为同一个表创建一个唯一聚集索引和多个唯一非聚集索引。

7. 包含列索引

一种非聚集索引，它扩展后不仅包含键列，还包含非键列。包含列索引是从 SQL Server 2005 开始提供的。通过包含非键列，可以创建覆盖更多查询的非聚集索引。当查询中的所有列都作为键列或非键列包含在索引中时，带有包含性非键列的索引可以显著提高查询性能。因为查询优化器可以在索引中找到所有列值，不需要访问表或聚集索引数据，从而减少磁盘 I/O 操作。

此外，通过包含非键列，可以突破索引的一些规则限制：

- 可以将不允许作为索引键列的数据类型包含在索引中。
- 在计算索引键列数或索引键大小时，数据库引擎不考虑非键列，这可以突破索引大小的限制（最大键列数为 16，总索引键大小为 900 字节）。

例如，假设要为 AdventureWorks 示例数据库的 Document 表中的以下列建立索引：

```
Title nvarchar(50)
Revision nchar(5)
FileName nvarchar(400)
```

因为 nchar 和 nvarchar 数据类型的每个字符需要 2 个字节，所以包含这 3 列的索引将超出 900 字节的大小限制（455×2）。使用 CREATE INDEX 语句的 INCLUDE 子句，可以将索引键定义为 (Title, Revision)，将 FileName 定义为非键列。这样，索引键大小将为 110 个字节（55×2），并且索引仍将包含所需的所有列。

创建包含列索引必须遵循下列限制：

- 必须至少定义一个键列。最大非键列数为 1023 列。也就是最大的表列数减 1。
- 所有非键列的总大小只受 INCLUDE 子句中所指定列的大小限制，如 varchar(max) 列限制为 2GB。

8. 索引视图

视图也称为虚拟表，因为视图所返回的结果集的一般格式与表相同，都是由列和行组成，而且在 SQL 语句中引用视图的方式也与引用表的方式相同。标准视图的结果集不是永久地存储在数据库中，每次查询引用标准视图时，数据库引擎都会在内部将视图的定义替换为该查询。

对于标准视图而言，为每个引用视图的查询动态生成结果集的开销很大，特别是对于那些涉及对大量行进行复杂处理（如聚合大量数据或联接许多行）的视图。如果在查询中频繁地引用这类视图，可通过对视图创建唯一聚集索引来提高性能。对视图创建唯一聚集索引后，结果集将存储在数据库中，就像带有聚集索引的表一样。

对视图创建索引的另一个好处是：优化器可以在未直接在 FROM 子句中指定某一视图的查询中使用该视图的索引。这样一来，可从索引视图检索数据而无需重新编码，由此带来的高效率也使现有查询获益。

对基表中的数据进行更改时，数据更改将反映在索引视图中存储的数据中。视图的聚集索引必须唯一，这一要求提高了在索引中查找受任何数据更改影响的行的效率。

如果很少更新基础数据，则索引视图的效果最佳。如果经常更新基础数据，则维护索引视图数据的成本可能超过使用索引视图所带来的性能收益。如果基础数据以批处理的形式定期更新，但在更新之间主要作为只读数据进行处理，则可以在更新前删除所有索引视图，然后再重新生成。这样做可以提高更新的性能。

9. 筛选索引

筛选索引是一种经过优化的非聚集索引，尤其适用于涵盖从定义完善的数据子集中选择数据的查询。筛选索引使用筛选谓词对表中的部分行进行索引。与全表索引相比，设计良好的筛选索引可以提高查询性能、减少索引维护开销并可降低索引存储开销。

筛选索引与全表索引相比具有以下优点。

- 提高了查询性能和计划质量。筛选索引比全表非聚集索引小并且具有经过筛选的统计信息，这些统计信息更加准确，因为它们只涵盖筛选索引中的行。
- 减少了索引维护开销。仅在数据操作语言（DML）语句对索引中的数据产生影响时，才对索引进行维护，减少了索引维护开销。筛选索引的数量可以非常多，特别是在其中包含很少受影响的数据时。同样，如果筛选索引只包含频繁受影响的数据，则索引大小较小时可以减少更新统计



涵盖索引可以提高查询性能，因为符合查询要求的全部数据都存在于索引本身中。也就是说，只需要索引页，而不需要表的数据页或聚集索引来检索所需数据，因此，减少了总体磁盘 I/O。例如，假设对某一表的列 a、列 b 和列 c 创建了组合索引，当对列 a 和列 b 的查询，仅仅从该索引本身就可以检索指定数据。

所谓的涵盖索引，就是传统方式在多个列上创建的索引。涵盖索引与包含列索引的区别在于，涵盖索引的所有列都是键列，而包含列索引可以包含非键列。涵盖索引可以是聚集索引，也可以是非聚集索引。

将插入或修改尽可能多的行的查询写入单个语句内，而不要使用多个查询更新相同的行。仅使用一个语句，就可以利用优化的索引维护。

此外，创建索引前应当评估查询类型以及列在查询中的使用方式。例如，在完全匹配查询类型中使用的列就适合用于非聚集索引或聚集索引。

3. 列注意事项

对于聚集索引，应当保持较短的索引键长度。另外，对唯一列或非空列创建聚集索引可以使聚集索引获益。不应在频繁更改的列上创建聚集索引（如 ID 号列）。由于数据库引擎必须按物理顺序保留行中的数据值，当在索引列上更改数据时，将导致表中数据的整行移动。对于频繁更改的列使用聚集索引，将影响表的更新性能。

不能将 `ntext`、`text`、`image`、`varchar(max)`、`nvarchar(max)` 和 `varbinary(max)` 数据类型的列指定为索引键列。不过，`varchar(max)`、`nvarchar(max)`、`varbinary(max)` 和 `xml` 数据类型的列可以作为非键索引列被包含在包含列索引中。

`xml` 数据类型的列只能在 XML 索引中用作键列。有关详细信息，请参阅 XML 数据类型列的索引。

同一个列组合的唯一索引比非唯一索引会提供更有用的查询优化器的附加信息。

应当分析列中数据的分布情况。通常情况下，为包含很少唯一值的列创建索引，或是在这样的列上执行联接，将会导致查询时间较长。例如，如果物理电话簿按姓的字母顺序排序，而城市里所有人的姓都是 Smith 或 Jones，则无法快速找到某个人。

考虑对具有定义完善的子集的列（例如，稀疏列、大部分值为 NULL 的列、含各类值的列以及含不同范围的值的列）使用筛选索引。设计良好的筛选索引可以提高查询性能，降低索引维护成本和存储成本。

如果索引包含多个列，则应考虑列的顺序。用于等于 (=)、大于 (>)、小于 (<) 或 BETWEEN 搜索条件的 WHERE 子句或者参与联接的列应该放在最前面。其他列应该基于其非重复级别进行排序，就是说，从最不重复的列到最重复的列。

必要的时候,也可以考虑对计算列进行索引。

4.2 创建索引

在创建索引前,需要确定要对哪些列进行索引,要使用的索引类型(如聚集或非聚集)和索引选项,以及确定文件组或分区方案布置。

创建索引时需要考虑的另一个重要因素是对空表还是对包含有数据的表创建索引。对空表创建索引时,不会对性能产生任何影响,只有向表中添加数据时,才会对性能产生影响。对大型表创建索引时需要仔细计划,防止影响数据库性能。对大型表创建索引的首选方法是先创建聚集索引,然后创建任何非聚集索引。在对现有表创建索引时,可以将表的 ONLINE 选项设置为 ON。在设置为 ON 时,将不持有长期表锁,可以继续对基础表进行查询或更新。

4.2.1 最大索引限制

表 4-1 列出了应用于聚集索引、非聚集索引、XML 索引和空间索引的最大值。

表 4-1 最大索引限制

最大索引限制	值	说 明
每个表的聚集索引数	1	
每个表的非聚集索引数	249	包括使用 PRIMARY KEY 或 UNIQUE 约束创建的非聚集索引,但不包括 XML 索引
每个表的 XML 索引数	249	包括 XML 数据类型列的主 XML 索引和辅助 XML 索引
每个表的空间索引	249	
每个索引的键列数	16	如果表中还包含主 XML 索引,则聚集索引限制为 15 列
最大索引键记录大小	900 字节	不适用于 XML 索引或空间索引。为了使表支持空间索引,最大索引键记录大小应当为 895 个字节

4.2.2 限制索引参与的数据类型

通常可以对表或视图中的任何列创建索引。表 4-2 列出了限制索引参与的数据类型。

表 4-2 限制索引参与的数据类型

数 据 类 型	索 引 参 与
CLR 用户定义类型	如果类型支持二进制顺序,则可以进行索引
大型对象 (LOB) 数据类型: image、ntext、text、varchar(max)、nvarchar(max)、varbinary(max) 和 xml	<p>不能作为索引键列。但是, xml 列可以作为表中的主 XML 索引或辅助 XML 索引的键列。</p> <p>可以作为非键 (包含性) 列参与非聚集索引, image、ntext 和 text 除外。如果是计算列表达式的一部分,则可以参与</p>

续表

数据类型	索引参与
计算列	可以进行索引。这包括定义为 CLR 用户定义类型列的方法调用的计算列，但方法应当被标记为确定性
推送到行外的 Varchar 列	一个表中的每一行最多可以包含 8060 字节，当合并 varchar、nvarchar、varbinary、sql_variant 或 CLR 用户定义类型的列超过此限制时，数据库引擎将把最大宽度的记录列推送到 ROW_OVERFLOW_DATA 分配单元的另一页上，而在原始页上保留一个 24 字节指针。聚集索引的索引键不能包含在 ROW_OVERFLOW_DATA 分配单元中具有数据的 varchar 列。如果对 varchar 列创建了聚集索引，并且在 IN_ROW_DATA 分配单元中存在现有数据，则对该列执行的将数据推送到行外的后续插入或更新操作将会失败

4.2.3 创建聚集索引

在表中创建 PRIMARY KEY 约束时，如果不存在该表的聚集索引且未指定唯一非聚集索引，则将自动对一列或多列创建唯一聚集索引。主键列不允许空值。例如，下面的语句在创建 Customers 表时并创建一个聚集索引。

```
CREATE TABLE Customers
(
    CustID int PRIMARY KEY,
    CustName char(20)
);
```

在创建 UNIQUE 约束时，可以包含 CLUSTERED 关键字指定创建聚集索引。否则，将默认创建唯一非聚集索引。下面的语句将创建一个名为 IX_Customers 的索引，其中仅包含有 CustID 列。

```
CREATE TABLE Customers -- 创建表
(
    CustID int NOT NULL, -- 注意这里应当建立一个非空列
    CustName char(20)
);
GO
CREATE UNIQUE CLUSTERED INDEX IX_Customers -- 创建索引
ON Customers(CustID)
```

然后向表中插入几行数据，注意其中是按 CustID 列降序方式插入的。

```
INSERT INTO Customers
VALUES(3, 'Ken levy'),
(2, 'Jay Kinker'),
(1, 'Grace Zhang');
```

执行下面的语句，查询结果如表 4-3 所示。由于我们创建的是聚集索引，表中的数据是按索引顺序存储的，所以会看到查询结果是按 CustID 列升序排列的。

```
SELECT * FROM Customers
```


表 4-3

Customers 的查询结果

CustID	CustName
1	GraceZhang
2	JayKinker
3	Kenlevy

如果不希望创建唯一聚集索引，可以省略 UNIQUE 关键字，参考下面的语句：

```
CREATE CLUSTERED INDEX IX_Customers
ON Customers(CustID)
```

4.2.4 创建非聚集索引

可以使用 NONCLUSTERED 关键字创建非聚集索引。例如，下面的语句为 Customers 表创建了一个名为 IX_Customers 的非聚集唯一索引，其中仅包含有 CustID 列。

```
CREATE TABLE Customers -- 创建表
(
    CustID int NOT NULL, -- 注意这里应当建立一个非空列
    CustName char(20)
);
GO
CREATE UNIQUE NONCLUSTERED INDEX IX_Customers
ON Customers(CustID)
```

使用下面的语句，按 CustID 列降序方式向表中插入几行数据。

```
INSERT INTO Customers
VALUES(3, 'Ken levy'),
(2, 'Jay Kinker'),
(1, 'Grace Zhang');
```

索引创建后，进行查询时是否使用索引，完全由查询优化器来决定。在前面已经讲过，对于小表进行索引可能不会产生优化效果，因为查询优化器在遍历用于搜索数据的索引时，花费的时间可能比执行简单的表扫描还长。但是，如果用户确认使用索引可以提高性能，也可以使用 WITH 子句强制指定要使用的索引。

下面的两行语句都是要查询表中 CustID 大于 1 的行，但是第 2 行强制使用了上面创建的 IX_Customers 索引。

```
SELECT * FROM Customers
WHERE CustID > 1;
SELECT * FROM Customers WITH (INDEX (IX_Customers))
WHERE CustID > 1;
```

得到的查询结果分别如表 4-4 和表 4-5 所示。

表 4-4

未强制使用索引得到的查询结果

CustID	CustName
3	Kenlevy
2	JayKinker

表 4-5

强制使用索引后得到的查询结果

CustID	CustName
2	JayKinker
3	Kenlevy

由上面两个表可以看出，表 4-5 是按索引顺序排列的，而表 4-4 则不是。这说明在未强制使用索引的情况下，查询优化器认为数据表太小，直接扫描表会比使用索引更有效率，在查询时未使用索引。通过查看生成的实际执行计划，也印证了这种判断。通过图 4-1 可以看到，直接使用表扫描开销仅为 33%，而在使用索引的情况下，开销达到了 67%。

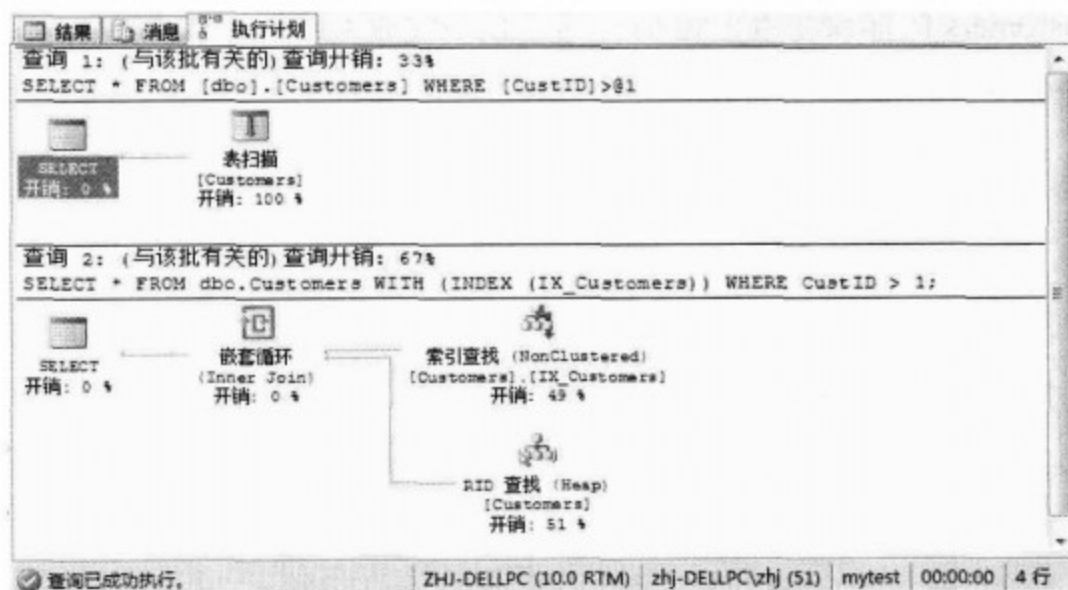


图 4-1 查询语句的实际执行计划

如果语句中既未指定 NONCLUSTERED，也未指定 CLUSTERED，则默认创建非聚集索引。如：

```
CREATE UNIQUE INDEX IX_Customers
ON Customers(CustID)
```

4.2.5 创建具有包含性列的索引

通过包含非键列，可以创建覆盖更多查询的非聚集索引。当查询中的所有列都作为键列或非键列包含在索引中时，带有包含性非键列的索引可以显著提高查询性能。

创建带有包含性非键列的索引，可以实现将不允许作为索引键列的数据类型的列包含在索引中。并且，在计算索引键列数或索引键大小时，数据库引擎不将非键列计算在内，从而不受索引大小限制。

例如，假设要为 AdventureWorks 示例数据库的 Document 表中的以下列建立索引：

```
Title nvarchar(50)
Revision nchar(5)
FileName nvarchar(400)
```

因为 nvarchar 数据类型要求每个字符 2 个字节，所以包含这 3 列的索引将超过 900 字节的大小限制。使用 CREATE INDEX 语句的 INCLUDE 子句，可以将索引键定义为 (Title, Revision)，将 FileName 定义为非键列。这样，索引键大小将为 110 个字节 (55×2)，并且索引仍将包含所需的所有列。参考下面的语句：

```
USE AdventureWorks;
GO
CREATE INDEX IX_Document_Title
ON Production.Document (Title, Revision)
INCLUDE (FileName);
```

当查询中的所有列都作为键列或非键列包含在索引中时，带有包含性非键列的索引可以显著提高查询性能。因为查询优化器可以在索引中找到所有列值，不需要访问表或聚集索引数据，从而减少磁盘 I/O 操作。当索引包含查询引用的所有列时，称为“覆盖查询”。

例如，假设要设计覆盖下列查询的索引：

```
USE AdventureWorks;
GO
SELECT AddressLine1, AddressLine2, City, StateProvinceID, PostalCode
FROM Person.Address
WHERE PostalCode BETWEEN N'98000' and N'99999';
```

如果要使用覆盖查询，必须在索引中定义每列。尽管可以将所有列定义为键列，但这样键的合计大小为 334 字节。由于实际上用作搜索条件的列只有 PostalCode 列（长度为 30 字节），所以更好的索引设计应该将 PostalCode 定义为键列，其他所有列作为非键列。参考下面的语句：

```
USE AdventureWorks;
GO
CREATE INDEX IX_Address_PostalCode
ON Person.Address (PostalCode)
INCLUDE (AddressLine1, AddressLine2, City, StateProvinceID);
```

4.2.6 为计算列创建索引

基表中计算列的值是从基表中非计算列派生而来，可以为基表中的计算列指定索引。例如，下面的语句定义了一个表，其中的 C 列是计算列。

```
CREATE TABLE T
( A int,
  B int,
  C AS A+B )
```

如果从表中检索数据，数据库引擎会为每条记录计算 C 列的值。但是如果使用下面的语句为

计算列创建一个索引，则在索引中把 C 列的值存储为键值。这样就允许基于 C 列进行快速查询，同时在检索时也减少了计算 C 列的时间。

```
CREATE INDEX X  
ON T ( C )
```

要为计算列定义索引，还有许多的严格要求，包括所有权要求、确定性要求、精度要求、数据类型要求和 SET 选项要求。

1. 所有权要求

计算列中的所有函数引用必须与表具有相同的所有者。

2. 确定性要求

计算列必须具有确定性。如果对于一组指定的输入表达式始终返回相同的结果，则说明表达式具有确定性。通过 COLUMNPROPERTY 函数的 IsDeterministic 属性可以确定计算列表式是否具有确定性。

如果下列一项或多项为真，则计算列表式具有确定性：

- 表达式引用的所有函数都具有确定性，并且是精确的。这些函数包括用户定义函数和内置函数。
- 表达式引用的所有列都来自包含计算列的表。
- 没有列引用从多行中请求数据。如 SUM 或 AVG 等聚合函数依靠来自多行的数据，这会使计算列表式具有不确定性。
- 没有系统数据访问或用户数据访问。

3. 精度要求

计算列表式必须精确。如果下列一项或多项为真，则表达式是精确的。

- 表达式的数据类型不是 float 或 real。
- 表达式定义中没有使用 float 或 real 数据类型。例如，在下列语句中，列 y 为 int 且具有确定性，但不精确。

```
CREATE TABLE t2 (a int, b int, c int, x float,  
y AS CASE x  
WHEN 0 THEN a  
WHEN 1 THEN b  
ELSE c  
END);
```

4. 数据类型要求

为计算列定义的表达式的值不能为 text、ntext 或 image 数据类型。

只要计算列的数据类型可以作为索引键列，从 image、ntext、text、varchar(max)、nvarchar(max)、

varbinary(max)和 xml 数据类型派生的计算列上就可以创建索引。

只要计算列的数据类型可以作为非键索引列,从 image、ntext 和 text 数据类型派生的计算列就可以作为非聚集索引中的非键(包含性)列。

5. SET 选项要求

执行定义计算列的 CREATE TABLE 或 ALTER TABLE 语句时,必须将 ANSI_NULLS 连接级选项设置为 ON。

对于在其中创建索引的连接和所有尝试执行 INSERT、UPDATE 或 DELETE 语句(将更改索引中的值)的连接,必须将 ANSI_NULLS、ANSI_PADDING、ANSI_WARNINGS、ARITHABORT、CONCAT_NULL_YIELDS_NULL 和 QUOTED_IDENTIFIER 选项设置为 ON,将 NUMERIC_ROUNDABORT 选项设置为 OFF。如果不具有上述选项设置的连接执行了任何 SELECT 语句,优化器将忽略计算列的索引。

4.3 修改索引

在创建索引后,可能需要修改索引的属性设置、重命名索引、禁用索引等。

4.3.1 禁用索引

禁用索引可防止用户访问被屏蔽的索引,对于聚集索引,还可防止用户访问基础表数据。索引被禁用后,索引定义和非聚集索引的索引统计信息会被保留。但是,对视图禁用非聚集索引或聚集索引会以物理方式删除索引数据。禁用聚集索引可以防止对数据的访问,但是数据仍保留在表中,但在删除或重新生成索引之前,无法对这些数据执行 DML 操作。

在数据库引擎升级期间会自动禁用索引,也可以使用 ALTER INDEX 手动禁用索引。

例如,下面的语句通过禁用 PRIMARY KEY 索引来禁用 PRIMARY KEY 约束,对基础表的 FOREIGN KEY 约束自动被禁用,并显示警告消息。

```
USE Adventureworks;
GO
ALTER INDEX PK_Department_DepartmentID ON HumanResources.Department
DISABLE ;
```

返回此警告消息如下。

```
警告:由于禁用了索引'PK_Department_DepartmentID',导致引用表'Department'的表'EmployeeDepartmentHistory'
上的外键'FK_EmployeeDepartmentHistory_Department_DepartmentID'也被禁用。
警告:由于禁用了表的聚集索引,表'Department'的索引'AK_Department_Name'已禁用。
```

下面的语句通过重新生成 PRIMARY KEY 索引,将启用上面被禁用的 PRIMARY KEY 约束。

```
ALTER INDEX PK_Department_DepartmentID ON HumanResources.Department
REBUILD;
```

下面的语句将启用 FOREIGN KEY 约束。

```
ALTER TABLE HumanResources.EmployeeDepartmentHistory
CHECK CONSTRAINT FK_EmployeeDepartmentHistory_Department_DepartmentID;
```

4.3.2 重新组织和重新生成索引

对基础数据执行插入、更新或删除操作时，数据库引擎会自动维护索引。随着时间的推移，这些修改可能会导致索引中的信息分散存储在数据库中，形成索引碎片。碎片非常多的索引可能会降低查询性能，导致应用程序响应缓慢。

可以通过重新组织索引或重新生成索引来修复索引碎片。对于基于分区方案生成的已分区索引，可以在完整索引或索引的单个分区上使用下列方法之一。

1. 检测碎片

要使用哪种碎片整理方法，首先应当使用系统函数 sys.dm_db_index_physical_stats 分析一下索引的碎片程度。对于已分区索引，sys.dm_db_index_physical_stats 还可以提供每个分区的碎片信息。

表 4-6 所示的列包含在 sys.dm_db_index_physical_stats 函数返回结果集中，可以通过这些列确定索引的碎片程度。

表 4-6 确定碎片程度的列

列	说 明
avg_fragmentation_in_percent	逻辑碎片（索引中的无序页）的百分比
fragment_count	索引中的碎片（物理上连续的叶页）数量
avg_fragment_size_in_pages	索引中一个碎片的平均页数

当 avg_fragmentation_in_percent 值大于 5%并且小于等于 30%时，可以使用 ALTER INDEX REORGANIZE 重新组织索引；当 avg_fragmentation_in_percent 值大于> 30%时，应当使用 ALTER INDEX REBUILD 语句重新生成索引。

sys.dm_db_index_physical_stats 函数的语法格式如下：

```
sys.dm_db_index_physical_stats (
    { database_id | NULL | 0 | DEFAULT }
    , { object_id | NULL | 0 | DEFAULT }
    , { index_id | NULL | 0 | -1 | DEFAULT }
    , { partition_number | NULL | 0 | DEFAULT }
    , { mode | NULL | DEFAULT }
)
```

● database_id | NULL | 0 | DEFAULT

数据库的 ID。可以使用 DB_ID() 函数来获取数据库的 ID，例如 DB_ID(N'mytest') 表示要获取 mytest 数据库的 ID。如果未指定数据库名称，则获取当前数据库的 ID。指定 NULL 可返回服务器实例中所有数据库的信息，并且，object_id、index_id 和 partition_number 应同时指定为 NULL。

● object_id | NULL | 0 | DEFAULT

该索引所基于的表或视图的对象 ID。可以使用 OBJECT_ID() 函数来获取该 ID，例如 OBJECT_ID(N'mytable') 表示要获取 mytable 表的 ID。指定 NULL 可返回指定数据库中的所有表和视图的信息，并且，index_id 和 partition_number 应同时指定为 NULL。

● index_id | 0 | NULL | -1 | DEFAULT

索引的 ID。

● partition_number | NULL | 0 | DEFAULT

对象中的分区号。

● mode | NULL | DEFAULT

模式的名称。用于指定获取统计信息的扫描级别，有效输入为 DEFAULT、NULL、LIMITED、SAMPLED 或 DETAILED。默认值 (NULL) 为 LIMITED。

例如，下面的示例语句将返回 Production.Product 表的所有索引的平均碎片。

```
USE AdventureWorks;
GO
SELECT a.index_id, b.name, avg_fragmentation_in_percent
FROM sys.dm_db_index_physical_stats (DB_ID(), OBJECT_ID(N'Production.Product'),
    NULL, NULL, NULL) AS a
JOIN sys.indexes AS b
ON a.object_id = b.object_id AND a.index_id = b.index_id;
```

得到的查询结果如表 4-7 所示。可以看出，对于 PK_Product_ProductID 索引应当重新组织，而其他则需要重新生成。

表 4-7 返回的索引碎片信息

index_id	name	avg_fragmentation_in_percent
1	PK_Product_ProductID	23.0769230769231
2	AK_Product_ProductNumber	50
3	AK_Product_Name	66.6666666666667
4	AK_Product_rowguid	50

2. 重新组织索引

使用 ALTER INDEX 语句的 REORGANIZE 子句可以重新组织一个或多个索引。如果要重新组织已分区索引的单个分区，可以使用 ALTER INDEX 的 PARTITION 子句。

重新组织索引是通过对叶页进行物理重新排序，使其与叶节点的逻辑顺序（从左到右）相匹配，从而对表或视图的聚集索引和非聚集索引的叶级别进行碎片整理。索引在分配给它的现有页内重新组织，而不会分配新页。如果索引跨多个文件，将一次重新组织一个文件，不会在文件之间迁移页。

重新组织还会压缩索引页。如果还有可用的磁盘空间，将删除此压缩过程中生成的所有空页。

重新组织进程使用最少的系统资源。而且，重新组织是自动联机执行的。该进程不持有长期阻塞锁，所以不会阻止运行查询或更新。

例如，下面的语句重新组织 PK_Product_ProductID 索引。

```
ALTER INDEX PK_Product_ProductID ON Production.Product  
REORGANIZE;
```

3. 重新生成索引

重新生成索引将删除该索引并创建一个新索引。此过程中将删除碎片，通过使用指定的或现有的填充因子设置压缩页来回回收磁盘空间，并在连续页中对索引行重新排序。

例如，下面的语句将重新生成 AK_Product_Name 索引。

```
ALTER INDEX AK_Product_Name ON Production.Product  
REBUILD;
```

4.3.3 设置索引选项

使用 ALTER INDEX 语句的 SET 子句可以随时设置 ALLOW_PAGE_LOCKS、ALLOW_ROW_LOCKS、IGNORE_DUP_KEY 或 STATISTICS_NORECOMPUTE 选项，设置时会立即应用于索引，而不需要重新生成。

例如，下面的语句为索引 PK_MyTable__08EA5793 设置了几个选项。

```
ALTER INDEX PK_MyTable__08EA5793 ON MyTable  
SET (  
    ALLOW_ROW_LOCKS = OFF,  
    ALLOW_PAGE_LOCKS = OFF  
);
```

对于其他一些选项，可以使用 REBUILD WITH 子句进行设置。例如，下面的语句指定了 ALL 关键字，将重新生成与表相关联的所有索引，其中指定了 4 个选项。

```
ALTER INDEX ALL ON MyTable
```



```
REBUILD WITH (PAD_INDEX = ON, FILLFACTOR = 80,
              SORT_IN_TEMPDB = ON, STATISTICS_NORECOMPUTE = ON);
```

4.3.4 重命名索引

可以使用 `sp_rename` 存储过程重命名索引。有关 `sp_rename` 的使用说明，可参考 3.3.3 小节的介绍。例如，下面的语句将 `MyTable.PK__MyTable__08EA5793` 重命名为 `PK__MyTable__Main`。

```
EXEC sp_rename 'MyTable.PK__MyTable__08EA5793', 'PK__MyTable__Main', 'INDEX'
```

4.4 删除索引

当一个索引不再需要时，可以将其从数据库中删除，以回收它当前使用的磁盘空间。必须先删除 `PRIMARY KEY` 或 `UNIQUE` 约束，才能删除约束使用的索引。在删除视图或表时，将自动删除为永久性和临时性视图或表创建的索引。

要删除 `PRIMARY KEY` 或 `UNIQUE` 约束，可使用 `ALTER TABLE DROP CONSTRAINT` 语句。例如，下面的语句删除了 `Production.ProductCostHistory` 表中具有 `PRIMARY KEY` 约束的聚集索引，如果 `ProductCostHistory` 表具有 `FOREIGN KEY` 约束，则应当首先删除 `FOREIGN KEY` 约束。

```
USE AdventureWorks;
GO
-- 在除 Enterprise Edition 之外的其他版本中应当设置 ONLINE = OFF
ALTER TABLE Production.ProductCostHistory
  DROP CONSTRAINT PK_ProductCostHistory_ProductID_StartDate
  WITH (ONLINE = ON);
```

在删除 `PRIMARY KEY` 或 `UNIQUE` 约束后，可以使用 `DROP INDEX` 语句删除其他索引。例如，下面的语句首先判断 `IX_SalesPerson_SalesQuota_SalesYTD` 索引是否存在，存在的话，则删除该索引。

```
IF EXISTS (SELECT name FROM sys.indexes
           WHERE name = N'IX_SalesPerson_SalesQuota_SalesYTD'
           ) --判断索引是否已经存在
  DROP INDEX IX_SalesPerson_SalesQuota_SalesYTD ON Sales.SalesPerson ; --删除索引
```



第5章 基本查询

从本章开始，我们将对数据查询进行介绍。进行数据查询需要使用 **SELECT** 语句，该语句可以从一个或多个表或视图中检索行，然后以一个或多个结果集的形式将其返回给用户。结果集的列将使用以前定义的列名、数据类型和精度，用户也可以根据需求重新指定这些元素。

本章内容将是后面各种深入查询的基础，尤其是理解好查询的逻辑处理步骤，哪个步骤在先，哪个步骤在后，以及每个步骤的作用，这对于能够快速编写出正确的查询语句，具有很重要的意义。

5.1 基本的 **SELECT** 语句

SELECT 语句的完整语法是比较复杂的，尤其是在包含子表引用、多表联接时，无论是语法，还是逻辑关系，则更为复杂。我们将从最基本的 **SELECT** 语句开始，逐步深入探讨这些方面的内容。

5.1.1 **SELECT** 语句的结构

基本的 **SELECT** 语句通常包含如下 4 部分内容：

- (1) **SELECT** select_list
- (2) **FROM** table_list
- (3) **WHERE** search_conditions
 GROUP BY group_by_list
 HAVING search_conditions
- (4) **ORDER BY** order_list

第 (1) 部分用于指定结果集中的列的名称、数据类型和精度，列名称之间使用逗号分隔。通常情况下，这些列是对源表或视图中的列的引用，但也可以是其他表达式（如常量或 Transact-SQL 函数）的引用。“*” 表达式指定返回源表的所有列。

第 (2) 部分指定从中检索结果集的表或视图的名称。

第 (3) 部分的 **WHERE**、**GROUP BY** 和 **HAVING** 用于指定检索结果的条件，源表中的行只有达到这个条件才会被放置到结果集中，否则将被过滤掉。

WHERE 子句首先进行第一次筛选，**GROUP BY** 子句根据 **group_by_list** 指定的列对筛选结果进行分组，**HAVING** 子句则是从分组结果中再过滤出符合指定条件的记录。

第(4)部分指定结果集中行的排序顺序，**order_list** 是使用逗号分隔排序列表。可以使用关键字 **ASC** 和 **DESC** 指定是按升序还是降序排序。

SELECT 语句中的子句必须以适当顺序指定，在后面的介绍中会逐步说明。

下面的示例首先创建一个名为 **MyTest** 的数据库，并在其中创建一个 **Orders** 表。

```
USE master;
GO
IF DB_ID (N'mytest') IS NOT NULL
    DROP DATABASE mytest;
GO
CREATE DATABASE mytest;
GO

USE mytest;
GO
IF OBJECT_ID(N'dbo.Orders') IS NOT NULL
    DROP TABLE dbo.Orders;
GO
CREATE TABLE dbo.Orders
(
    ProductID INT NOT NULL,
    MadeFrom CHAR(20),
    Sales MONEY NOT NULL
);
```

执行下面的 **INSERT** 语句，向 **Orders** 表插入示例数据。表 5-1 显示了表中的内容。

```
INSERT INTO dbo.Orders
VALUES (1,'China',100.00),
      (1,'China',100.00),
      (2,'China',80.00),
      (2,'China',80.00),
      (3,'China',90.00),
      (4,'USA',200.00);
```

表 5-1 Orders 表中的内容

ProductID	MadeFrom	Sales
1	China	100.00
1	China	100.00
2	China	80.00
2	China	80.00
3	China	90.00
4	USA	200.00

下面是一个典型的 SELECT 语句，首先取出 Orders 表中 MadeFrom 为 China 的行，然后根据 ProductID 分组对 Sales 求和，再使用 HAVING 筛选出求和结果大于 150 的行，最后对结果集按 ProductID 排序输出。查询结果如表 5-2 所示。

```
SELECT ProductID, SUM(Sales) AS TotalSales
FROM dbo.Orders
WHERE MadeFrom = N'China'
GROUP BY ProductID
HAVING SUM(Sales) > 150
ORDER BY ProductID;
```

表 5-2

查询结果

ProductID	TotalSales
1	200.00
2	160.00

如果对上面的示例理解起来有一些难度，可以先继续阅读后面的内容，该示例仅旨在说明一下 SELECT 语句中各个部分的具体应用，其中如 GROUP 子句等相关内容在后面还有更详细的介绍。

5.1.2 数据库对象的引用规则

对数据库对象的引用方面应当遵循下面的规则，避免引起歧义。

如果在一个数据库中有多个对象具有相同的名称，则应使用架构名称来限定表名称。例如，假设 Schema1 和 Schema2 架构都含有一个名为 TableX 的表，而下面的语句明确指定要引用 Schema1 中的 TableX 表：

```
SELECT *
FROM Schema1.TableX;
```

如果所引用的对象不是在当前数据库中，则应当以数据库和架构来限定对象名称。例如，下面的语句明确指定要引用 AdventureWorks 数据库中的 Purchasing.ShipMethod 表：

```
SELECT *
FROM AdventureWorks.Purchasing.ShipMethod;
```

也可以使用 USE 语句将指定数据库切换为当前数据库。例如，上面的语句可以改写为下列格式：

```
USE AdventureWorks;  -- 选择 AdventureWorks 数据库
GO
SELECT *
FROM Purchasing.ShipMethod;  -- 从当前数据库的 Purchasing.ShipMethod 表检索数据
```

如果 FROM 子句中所指定的表和视图存在相同的列名，则应当使用表或视图名称来限定列名。例如，下面语句所生成的结果集合由 Customer 表的 CustomerID 列和 Store 表的 Name 列组成，在指定连接条件时，为 CustomerID 分别使用了 Sales.Customer 和 Sales.Store 限定。


```
SELECT Sales.Customer.CustomerID, Sales.Store.Name
FROM Sales.Customer
JOIN Sales.Store
ON Sales.Customer.CustomerID = Sales.Store.CustomerID
WHERE Sales.Customer.TerritoryID = 1;
```

5.2 使用选择列表和表别名

选择列表用于定义 `SELECT` 语句的结果集中的列。选择列表是一系列以逗号分隔的表达式，每个表达式定义结果集中的一列。结果集中列的排列顺序与选择列表中表达式的排列顺序相同。

5.2.1 选择所有列

在选择列表中，可以使用 “*” 关键字指定返回表中的所有列。如果 “*” 前面没有使用表或视图名称限定符，则被解析为对 `FROM` 子句中指定的所有表或视图中的所有列的引用。例如，下列的语句将检索 `Product` 表中的所有行：

```
SELECT *
FROM Production.Product;
```

如果在 “*” 前面使用了表或视图名称进行限定，则被解析为对限定表或视图中的所有列的引用。例如，下面的语句将引用 `Product` 表中的所有列：

```
SELECT Product.*
FROM Production.Product;
```

当使用 “*” 时，结果集中的列的顺序与 `CREATE TABLE`、`ALTER TABLE` 或 `CREATE VIEW` 语句中所指定的顺序相同。

5.2.2 选择特定列

如果在与结果集中的列数具有逻辑相关性的应用程序或脚本中使用 `SELECT` 时，最好指定具体的列名，而不是使用 “*” 来检索所有列。由于每次执行 `SELECT *` 语句时，表结构的更改都会自动反映出来，所以，在添加或删除列后，很可能影响应用程序或脚本的逻辑。尤其是对视图目录、动态管理视图和系统表值函数，更应当避免使用 “*”，因为在将来的升级和版本可能会添加列，并更改这些视图和函数中的列的顺序，这些更改可能会中断需要特定列顺序和列数的应用程序。

下面的语句指定返回 `Sales.Customer` 表中的 `CustomerID` 和 `TerritoryID` 等列，结果集中列的顺序将按照 `SELECT` 定义中列出的顺序排列。

```
SELECT CustomerID,
TerritoryID,
AccountNumber,
CustomerType,
rowguid,
```

```

ModifiedDate
FROM Sales.Customer

```

5.2.3 在选择列表中使用常量、函数和表达式

在 SELECT 的选择列表中可以包含常量、函数和使用列名、常量和函数表达式。所有的数据库产品都支持数值的标准四则运算（+、-、*、/）、取模或取余（%）运算。进行四则运算的算术运算符可在如 int、smallint、tinyint 和 decimal 等任何数值列或表达式中使用，模运算符只能在 int、smallint 或 tinyint 列或表达式中使用。

也可使用日期函数或常规加或减算术运算符对 datetime 和 smalldatetime 列进行算术运算。

1. 使用常量

例如，下面的语句使用了常量和算术表达式，检索到的结果集如表 5-3 所示。

```

USE AdventureWorks;
GO
SELECT N'产品编号' AS 我的常量列,
       ProductID,
       StandardPrice*1.1 AS 调整后价格
FROM Purchasing.ProductVendor;

```

表 5-3 使用常量和算术表达式得到的查询结果

我的常量列	ProductID	调整后价格
产品编号	1	52.65700
产品编号	2	43.91200
产品编号	4	59.74100
产品编号	317	30.98700
产品编号	317	28.34700
产品编号	318	37.81800
产品编号	318	35.17800
...

2. 使用函数

也可以将经函数运算后的列结果放置到结果集中。例如，下面的语句使用 RTRIM 函数删除 FirstName 和 LastName 列的尾部空格，并在二者之间使用空格分割后通过“+”串联在一起。得到的结果集如表 5-4 所示。

```

USE AdventureWorks;
GO
SELECT ContactID,
       RTRIM(FirstName) + ' ' + RTRIM(LastName) AS 姓名,
       EmailAddress AS 邮箱,

```

Phone AS 电话
FROM Person.Contact;

表 5-4 使用函数得到的查询结果

ContactID	姓 名	邮 箱	电 话
1	Gustavo Achong	gustavo0@adventure-works.com	398-555-0132
2	Catherine Abel	catherine0@adventure-works.com	747-555-0171
3	Kim Abercrombie	kim2@adventure-works.com	334-555-0137
4	Humberto Acevedo	humberto0@adventure-works.com	599-555-0127
5	Pilar Ackerman	pilar1@adventure-works.com	1 (11) 500 555-0132
6	Frances Adams	frances0@adventure-works.com	991-555-0183
7	Margaret Smith	margaret0@adventure-works.com	959-555-0151
8	Carla Adams	carla0@adventure-works.com	107-555-0138
9	Jay Adams	jay1@adventure-works.com	158-555-0142
10	Ronald Adina	ronald0@adventure-works.com	453-555-0165
11	Samuel Agcaoili	samuel0@adventure-works.com	554-555-0110
...

3. 使用表达式

例如，通过 CASE 结构，可以根据条件返回相应的值。CASE 结构的语法格式如下：

```
CASE
  WHEN condition1 THEN result-expression1
  ...
  WHEN conditionn THEN result-expressionn
  ELSE result-expressionx
END
```

数据库引擎按照顺序计算每一个条件，如果找到条件为真的语句，就执行 THEN 关键字后面的表达式。如果所有条件均不符合，则执行 ELSE 关键字后面的语句。如果既没有为真的条件，也没有 ELSE 关键字，则 CASE 表达式返回值为空。

例如，下面的语句从 SalesOrderHeader 表中取出每笔订单的金额小计，并在结果集中以“2 万以上订单”等文字说明信息列出，如表 5-5 所示。

```
USE AdventureWorks;
GO
SELECT SalesOrderID, SubTotal,
  CASE
    WHEN SubTotal > 20000 THEN N'2 万以上订单'
    WHEN SubTotal >= 10000 AND SubTotal <= 200000 THEN N'1-2 万订单'
    ELSE N'1 万以下订单'
  END AS 订单说明
FROM Sales.SalesOrderHeader
```

表 5-5 使用表达式的查询结果

SalesOrderID	SubTotal	订 单 说 明
43659	24643.9362	2 万以上订单
43660	1553.1035	1 万以下订单
43661	39422.1198	2 万以上订单
43662	34689.5578	2 万以上订单
43663	503.3507	1 万以下订单
43664	29312.401	2 万以上订单
43665	17199.2839	1~2 万订单
43666	6079.6842	1 万以下订单
...

也可以在表达式中使用算术运算符和函数进行计算。例如，下面的语句首先将 ProductID 列转换为 VARCHAR 类型，然后使用“+”与 Name 列连接在一起。查询结果如表 5-6 所示。

```
USE AdventureWorks;
GO
SELECT ProductID, Name,
       (CAST(ProductID AS VARCHAR(10)) + ': ' + Name ) AS ProductIDName
FROM Production.Product
```

表 5-6 使用算术运算符和函数的查询结果

ProductID	Name	ProductIDName
1	Adjustable Race	1: Adjustable Race
879	All-Purpose Bike Stand	879: All-Purpose Bike Stand
712	AWC Logo Cap	712: AWC Logo Cap
3	BB Ball Bearing	3: BB Ball Bearing
2	Bearing Ball	2: Bearing Ball
877	Bike Wash - Dissolver	877: Bike Wash - Dissolver
316	Blade	316: Blade
...

4. 使用特殊关键字

在 SQL Server 中允许使用 \$IDENTITY 关键字替代表中具有 IDENTITY 属性的列的名称，或者使用 \$ROWGUID 关键字替代表中具有 ROWGUIDCOL 属性的列的名称。

例如，由于为 AdventureWorks 数据库中 SalesOrderHeader 表的 SalesOrderOrderID 列定义了 IDENTITY 属性，因此表达式 SalesOrderHeader.\$IDENTITYCOL 等于 SalesOrderHeader.OrderID。

5. 指定结果集的列名称

在选择列表中，有些列并不是对源表列的简单引用，而是经过计算或其他操作的派生列。在这种情况下，就需要使用 AS 子句为列指定一个名称，否则派生列没有名称。在前面的介绍中，我们实际上也多次使用了 AS 子句的功能。在下面的语句中，如果删除 AS 子句，则使用 DATEDIFF 函数指定的派生列将会没有名称：

```
SELECT SalesOrderID,
       DATEDIFF(dd, ShipDate, GETDATE()) AS DaysSinceShipped
FROM AdventureWorks.Sales.SalesOrderHeader
WHERE ShipDate IS NOT NULL
```

如果结果集的列名称是遵循标识符规则的常规标识符，可以像上面的 DaysSinceShipped 一样直接写入，否则就必须将列名包含在方括号 ([]) 或双引号 (") 内。

注意：每个列名可最多使用 128 个字符。但是，DB-Library 应用程序（例如 isql 工具）在查询输出中最多将结果集列名截取为 30 个字符。SQL Server 6.5 或更早版本的 Microsoft SQL Server ODBC 驱动程序也最多将结果集列名截断为 30 个字符。

例如，下面语句返回的结果集列的名称是 Product Name，而不是默认的名称。

```
SELECT Name AS "Product Name"
FROM Production.Product
```

此外，Transact-SQL 的保留关键字加上引号之后也可以用作列标题。例如，下面的语句使用保留字 SUM 作为列标题：

```
SELECT SUM(TotalDue) AS "sum"
FROM Sales.SalesOrderHeader
```

5.2.4 使用表别名简化表引用

在前面介绍了可以使用 AS 为列指定一个名称，同样，也可以使 AS 关键字为表指定别名，以提高 SELECT 语句的可读性，如：table_name AS table_alias。也可以不使用 AS 关键字，直接以 table_name table_alias 形式列出。

例如，在下面的语句中，将别名 C 分配给了 SalesOrderHeader，将别名 S 分配给 SalesOrderDetail。返回的结果集中包含 SalesOrderHeader 表的 SalesOrderID 列和 SalesOrderDetail 表的 LineTotal 列。

```
USE AdventureWorks;
GO
SELECT C.SalesOrderID, S.LineTotal
FROM Sales.SalesOrderHeader AS C
     JOIN Sales.SalesOrderDetail AS S
       ON C.SalesOrderID = S.SalesOrderID;
```

如果为表指定了别名后，在 Transact-SQL 语句中对该表的所有显式引用都必须使用别名，而不能再使用表名。

5.3 使用 WHERE 子句筛选行

如果不希望在查询时取出表中的所有行，而是仅希望得到满足一定条件的行，则可以使用 WHERE 子句。在 WHERE 子句中可以包含多个筛选条件，各条件之间可以使用逻辑运算符 AND 或 OR 连接。也可以使用 NOT 进行求反。

5.3.1 使用比较搜索条件

可以使用=、>、<等比较运算符指定一个比较搜索条件。例如，下面的语句用于检索出 Product 表中标价（ListPrice）高于\$50 并且产品 ID（ProductID）大于 850 的所有行。

```
USE AdventureWorks;
GO
SELECT *
FROM Production.Product
WHERE ListPrice > $50.00 AND ProductID > 850;
```

也可以使用 NOT 对表达式求反。例如，下面的语句将查找 Product 表中标价（ListPrice）小于或等于\$50 的所有行。

```
USE AdventureWorks;
GO
SELECT *
FROM Production.Product
WHERE NOT ListPrice > $50.00;
```

当一个语句中使用了多个逻辑运算符时，计算顺序依次为：NOT、AND 和 OR。算术运算符和位运算符优先于逻辑运算符处理。例如，下面的语句要搜索 ProductModelID 等于 20 的产品，或是 ProductModelID 等于 21 并且 Color 等于 Red 的产品。也就是说，其中的 Color = 'Red' 条件适用于 Product Model 21，而不是 Product Model 20，因为 AND 优先于 OR。

```
USE AdventureWorks;
GO
SELECT ProductID, ProductModelID
FROM Production.Product
WHERE ProductModelID = 20 OR ProductModelID = 21
    AND Color = 'Red';
```

在比较字符串数据时，比较结果受排序规则所定义的字符顺序控制。对于非 Unicode 数据，在比较时将忽略尾随空格。例如，下列语句是等效的：

```
WHERE LastName = 'White'
WHERE LastName = 'White '
WHERE LastName = 'White' + SPACE(1)
```

5.3.2 使用范围搜索条件

可以使用 **BETWEEN** 关键字检索介于两个指定值之间的所有值。例如，下面的语句返回标价在\$15 和\$25 之间的所有产品：

```
USE AdventureWorks;
GO
SELECT ProductID, Name
FROM Production.Product
WHERE ListPrice BETWEEN $15 AND $25;
```

也可以使用大于和小于运算符 (>和<) 指定一个范围搜索条件。例如，下面的语句返回与上面语句同样的结果集。

```
USE AdventureWorks;
GO
SELECT ProductID, Name
FROM Production.Product
WHERE ListPrice > $15 AND ListPrice < $25;
```

可以使用 **NOT BETWEEN** 查找指定范围之外的所有行。例如，下面的语句用于查找标价在 15～25 范围之外的所有产品：

```
USE AdventureWorks;
GO
SELECT ProductID, Name
FROM Production.Product
WHERE ListPrice NOT BETWEEN $15 AND $25;
```

5.3.3 使用列表搜索条件

可以使用 **IN** 关键字选择与列表中的值相匹配的行，列表中的值的各项必须以逗号隔开。例如，下面的语句用于查找 **ProductSubCategoryID** 等于 12、14、16 的行：

```
USE AdventureWorks;
GO
SELECT ProductID, Name
FROM Production.Product
WHERE ProductSubCategoryID IN (12, 14, 16);
```

如果不使用 **IN** 关键字，要实现上面的查询，则需要通过 **OR** 来连接多个条件。参考下面的语句：

```
USE AdventureWorks;
GO
SELECT ProductID, Name
FROM Production.Product
WHERE ProductSubcategoryID = 12
   OR ProductSubcategoryID = 14
   OR ProductSubcategoryID = 16
```

实际上, IN 关键字最重要的应用是在嵌套查询(也称为子查询)中。使用下面的语句分别创建 `dbo.Orders` 和 `dbo.OrderDetail` 表并插入一些数据, 其中 `Orders` 中保存的是订单的发货日期和目的地, `OrderDetail` 中保存的是每笔订单中产品 ID 和售价。`Orders` 和 `OrderDetail` 表中的内容如表 5-7 和表 5-8 所示。

```
IF OBJECT_ID(N'dbo.Orders') IS NOT NULL
    DROP TABLE dbo.Orders;
IF OBJECT_ID(N'dbo.OrderDetail') IS NOT NULL
    DROP TABLE dbo.OrderDetail;
GO
-- 创建 dbo.Orders 表并插入数据
CREATE TABLE dbo.Orders
(
    OrderID INT NOT NULL,
    ShipDate DATETIME NOT NULL,
    SendTO CHAR(20)
);
INSERT INTO dbo.Orders
VALUES (1, '2008-5-1', 'ShanDong, China'),
      (2, '2008-6-10', 'ShanDong, China'),
      (3, '2008-7-9', 'ShangHai, China')
-- 创建 dbo.OrderDetail 表并插入数据
CREATE TABLE dbo.OrderDetail
(
    OrderID INT NOT NULL,
    ProductID INT NOT NULL,
    Sales MONEY NOT NULL
);
INSERT INTO dbo.OrderDetail
VALUES (1, 210, 90.00),
      (2, 211, 100.00),
      (2, 213, 110.00),
      (3, 211, 100.00);
```

表 5-7

Orders 表的内容

OrderID	ShipDate	SendTO
1	2008-05-01 00:00:00.000	ShanDong, China
2	2008-06-10 00:00:00.000	ShanDong, China
3	2008-07-09 00:00:00.000	ShangHai, China

表 5-8

OrderDetail 表的内容

OrderID	ProductID	Sales
1	210	90.00
2	211	100.00
2	213	110.00
3	211	100.00

下面语句中的子查询首先从 `Orders` 表中读取 `ShipDate` 日期大于 2008-6-1 的订单 ID, 分别是 2 和 3, 然后从 `OrderDetail` 表中读取这些订单中产品的信息。查询结果如表 5-9 所示。

```
SELECT OrderID, ProductID, Sales
```



```

FROM dbo.OrderDetail
WHERE OrderID IN
  (SELECT OrderID
   FROM dbo.Orders
   WHERE ShipDate > CAST('2008-6-1' AS DATETIME));

```

表 5-9

查询结果

OrderID	ProductID	Sales
2	211	100.00
2	213	110.00
3	211	100.00

有关子查询的详细信息，可参考第6章的内容。

5.3.4 使用模式匹配搜索条件

可以使用 LIKE 关键字搜索与指定模式相匹配的字符串、日期或时间值。模式中应当包含要搜索的字符串，以及用于模式匹配的通配符。如表 5-10 所示。

表 5-10

可用于模式匹配的通配符

通 配 符	含 义
%	代表任意字符串
_	代表单个任意字符
[]	指定范围（如[a-f]）或集合（如[abcdef]）内的任意单个字符
[^]	不在指定范围（如[^a-f]）或集合（如[^abcdef]）内的任意单个字符

1. 使用 “%” 通配符

例如，下面的语句用于搜索 Name 列中以 LL 开头、以 le 结尾的所有行：

```

USE AdventureWorks;
GO
SELECT ProductID, Name
FROM Production.Product
WHERE Name Like 'LL%le';

```

也可以使用 LIKE '%en%' 这样的条件搜索在任意位置包含字母 en 的所有字符串（如 Bennet、Green 和 McBadden）。

2. 使用 “_” 通配符

下面的语句将搜索 Name 列中以字母 hain 结尾的所有 6 个字母的名称（如 Chain）：

```

USE AdventureWorks;
GO
SELECT ProductID, Name
FROM Production.Product

```

```
WHERE Name LIKE '_hain';
```

3. 使用 “[]” 通配符

下面的语句将搜索 Name 列中以 inger 结尾、以 M 到 Z 中的任何单个字母开头的行（如 Ringer）：

```
USE AdventureWorks;
GO
SELECT ProductID, Name
FROM Production.Product
WHERE Name LIKE '[M-Z]inger';
```

4. 使用 “[^]” 通配符

下面的语句将搜索 Name 列中以字母 M 开头，并且第 2 个字母不是 c 的所有名称（如 MacFeather）：

```
USE AdventureWorks;
GO
SELECT ProductID, Name
FROM Production.Product
WHERE Name LIKE 'M[^c]%';
```

5. 使用 NOT 进行求反

例如，下面的语句将搜索 Phone 列中不以 415 开头的行：

```
USE AdventureWorks;
GO
SELECT Phone
FROM Person.Contact
WHERE Phone NOT LIKE '415%';
```

也可以写为下列形式：

```
SELECT Phone
FROM Person.Contact
WHERE NOT Phone LIKE '415%';
```

IS NOT NULL 子句可以与通配符和 LIKE 子句结合使用。例如，下面的语句将搜索 Phone 列中以 415 开头且 IS NOT NULL 的所有行：

```
SELECT Phone
FROM Person.Contact
WHERE Phone LIKE '415%'
AND Phone IS NOT NULL;
```

注意：如果模式表达式以通配符开头，则不能使用索引。例如，如果指定了 LIKE '%mith' 这样的搜索条件，索引则不知道应从哪一页开始查找。但是，表达式中间或结尾处的通配符并不妨碍索引的使用，如 LIKE 'Samuel%'。

6. 使用转义符

如果要搜索列中包含的通配符字符，则需要使用转义符，或将要搜索的通配符包含在方括号（[]）内。可以使用 `ESCAPE` 关键字来定义转义符，当把转义符放置在通配符的前面时，则该通配符就被解释为普通字符。例如，要搜索在任意位置包含“5%”的字符串，需要使用下面的子句：

```
WHERE ColumnA LIKE '%5/%' ESCAPE '/'
```

上面的子句中的前导和结尾百分号（%）被解释为通配符，而斜杠（/）之后的百分号被解释为字符“%”。

也可以将要搜索的通配符包含在方括号中。例如，上面的语句可以写成下面的形式：

```
WHERE ColumnA LIKE '5[%]'
```

在使用 `LIKE` 执行字符串比较时，模式串中的所有字符（包括每个前导空格和尾随空格）都有意义。例如，`LIKE 'abc '`（`abc` 后跟一个空格）要求返回带有字符串“abc”的所有行，而不会返回列值为“abc”（注意，`abc` 后没有空格）的行。但是，如果使用了 `LIKE 'abc'`（`abc` 后没有空格），却会返回以 `abc` 开头并且具有 0 个或多个尾随空格的所有行。

5.3.5 使用 NULL 比较搜索条件

`NULL` 值表示数据值未知（没有输入数据）或不可用，`NULL` 值与 0、0 长度的字符串或空白（字符值）的含义不同。

比较空值时的行为取决于 `SET ANSI_NULLS` 选项的设置。当 `SET ANSI_NULLS` 为 `ON` 时，如果比较中有一个或多个表达式为 `NULL`，则返回 `UNKNOWN`。这是因为未知值不能与其他任何值进行逻辑比较。例如，下面的语句将返回 `UNKNOWN`。

```
ytd_sales > NULL
```

当 `ANSI_NULLS` 设置为 `OFF` 时，两个都取空值的表达式比较输出 `TRUE`。将一个值与空值比较则返回 `FALSE`。例如，下面的语句返回 `Customer` 表中 `TerritoryID` 为空值的所有行：

```
USE AdventureWorks;
GO
SELECT CustomerID, AccountNumber, TerritoryID
FROM Sales.Customer
WHERE TerritoryID = NULL;
```

也可以使用 `IS NULL` 或 `IS NOT NULL` 子句测试 `NULL` 值。例如，下面的语句执行与上面语句同样的功能：

```
SELECT CustomerID, AccountNumber, TerritoryID
FROM Sales.Customer
WHERE TerritoryID IS NULL;
```

要搜索列中为非空值的行，可以使用下面的子句：

```
WHERE column_name IS [NOT] NULL
```

5.4 使用 GROUP BY 子句和聚合函数进行分组计算

GROUP BY 子句用于按指定行对表中相同的数据进行分组，并为每组在结果集中生成一行。GROUP BY 需要与聚合函数协同工作。下面的语句根据 SalesOrderID（销售订单 ID）对 LineTotal 列进行分组求和，计算每笔销售订单的总额。

```
USE AdventureWorks;
GO
SELECT SalesOrderID, SUM(LineTotal) AS SubTotal
FROM Sales.SalesOrderDetail
GROUP BY SalesOrderID;
```

GROUP BY 关键字后跟一个列的列表，称为组合列。在使用 GROUP BY 关键字时，SELECT 语句的选择列表中的列必须包含在组合列中，否则，应当将列包含在聚合函数中。

执行下面的语句，创建一个名为 Mytable 的表，并向其中插入如表 5-11 所示的数据。

```
IF OBJECT_ID(N'dbo.Mytable') IS NOT NULL
    DROP TABLE dbo.Mytable;
CREATE TABLE dbo.Mytable
(
    ColA INT,
    ColB CHAR(20),
    ColC INT
);

INSERT INTO dbo.Mytable
VALUES (NULL, NULL, NULL),
       (NULL, NULL, NULL),
       (1, 'abc', 1),
       (1, 'def', 1),
       (1, 'ghi', 1),
       (2, 'jkl', 1),
       (2, 'mno', 1);
```

表 5-11

Mytable 表的内容

ColA	ColB	ColC
NULL	NULL	NULL
NULL	NULL	NULL
1	abc	1
1	def	1
1	ghi	1
2	jkl	1
2	mno	1

执行下面的语句，按 ColA 列进行数据分组。


```
SELECT ColA,
       COUNT(ColB) AS ColBCount,
       SUM(ColC) AS ColCSum
FROM Mytable
GROUP BY ColA
```

由于 ColB 和 ColC 列没有出现在 GROUP BY 的组合列中，所以必须将其包含在聚合函数中。如果分组列包含一个或多个空值，则这些空值将被放入同一个组中。最终的结果集如表 5-12 所示。

表 5-12 根据 ColA 列分组后的结果集

ColA	ColBCount	ColCSum
NULL	0	NULL
1	3	3
2	2	2

同时，需要注意的是，当使用聚合函数对包含 NULL 的列计算时，NULL 不会被计算在内。例如，Mytable 的 ColB 列包含两行 NULL，但是，由上表可以看出，COUNT() 函数的计算结果为 0，而对 ColC 列的 SUM() 求和，NULL 组的计算结果仍旧为 NULL。下面的语句对表中的所有行进行计数和求和，更能说明这个问题：

```
SELECT COUNT(ColB) AS ColBCount,
       SUM(ColC) AS ColCSum
FROM Mytable;
```

查询结果如表 5-13 所示。可以看到对 ColB 列的计数为 5，两行 NULL 值未被计算在内。当对包含 NULL 值的行进行求和时，NULL 值会被忽略，所以查询结果中 ColCSum 列的值为 5。

表 5-13 对包含 NULL 值的列进行计数和求和的结果

ColBCount	ColCSum
5	5

5.5 使用 HAVING 子句从分组后结果中筛选行

HAVING 子句与 WHERE 子句的功能类似，都是对行进行筛选。但是，WHERE 搜索条件是在分组操作之前对记录进行筛选，然后再由 GROUP BY 对筛选后符合条件的行进行分组；而 HAVING 搜索条件则是对分组操作之后得到的行再进行筛选操作。在 HAVING 子句中可以包含聚合函数，而 WHERE 子句不能。

下面是 WHERE、GROUP BY 和 HAVING 子句的正确顺序：

- WHERE 子句用来筛选 FROM 子句中指定的操作所产生的行；
- GROUP BY 子句用来分组 WHERE 子句的输出；
- HAVING 子句用来从分组的结果中筛选行。

在没有包含 GROUP BY 子句的情况下，HAVING 子句与 WHERE 子句功能完全相同，但是使

用 WHERE 子句会更高效。因为 WHERE 子句能够事先把不必要的数据过滤掉，从而减少了在执行 SELECT 时的数据处理量。但是，有些数据事先并不知道是否需要过滤掉，要根据结果才能确定，这时候就必须使用 HAVING 子句解决。

例如，下面的语句首先按 ProductID 对 SalesOrderDetail 表中的行进行分组，然后 HAVING 子句从分组结果中过滤出那些订单合计（SUM(LineTotal)）大于\$1,000,000 的行。

```
USE AdventureWorks;
GO
SELECT ProductID, SUM(LineTotal) AS Total
FROM Sales.SalesOrderDetail
GROUP BY ProductID
HAVING SUM(LineTotal) > $1000000.00;
```

5.6 使用 ORDER BY 子句进行排序

ORDER BY 子句指定对输出的结果集按一列或多列进行排序。ORDER BY 子句中的列的数目没有限制，但是，排序操作所需的中间工作表的行大小限制为 8060 个字节，这限制了 ORDER BY 子句中列的总大小。

5.6.1 指定排序列

1. 按单列排序

指定排序列时，可以是一个列名称或列别名，也可以是列在 SELECT 选择列表中所处位置的一个整数。例如，下面的查询语句指定按 ProductID 列排序。

```
USE AdventureWorks;
GO
SELECT ProductID, ProductLine, ProductModelID
FROM Production.Product
ORDER BY ProductID;
```

而下面的查询则是使用 ProductID 列在 SELECT 选择列表中的位置号代替了具体的列名，查询结果与上面相同。

```
SELECT ProductID, ProductLine, ProductModelID
FROM Production.Product
ORDER BY 1;
```

2. 按多列排序

上面介绍的查询语句，都是按单列进行数据的排序。按多列排序时，列之间需要以逗号进行分隔。例如，下面查询语句首先按 ProductModelID 排序，对于同一 ProductModelID 中的产品则按 ProductID 进行排序。

```
SELECT ProductModelID, ProductID, ProductLine
FROM Production.Product
```

```
ORDER BY ProductModelID, ProductID;
```

3. 使用列别名进行排序

通常是在需要进行表达式计算的列时，为列指定一个别名。如果一个列的名称过长或不容易记忆，也可以为列再重新指定一个名称。例如，下面的查询语句中将 `FirstName` 列和 `LastName` 列联接在了一起，别名指定为 `FullName`，并按 `FullName` 排序。

```
SELECT ContactID,
       RTRIM(FirstName) + ' ' + RTRIM(LastName) AS FullName
FROM Person.Contact
ORDER BY FullName
```

4. 使用选择列之外的列排序

`ORDER BY` 子句可包括未在选择列表中列出的项。例如，下面的查询仅返回 `ProductLine` 和 `ProductModelID` 列的内容，但是仍旧按 `ProductID` 排序。

```
SELECT ProductLine, ProductModelID
FROM Production.Product
ORDER BY ProductID;
```

但是，如果已指定了 `SELECT DISTINCT` 或该语句包含 `GROUP BY` 子句，或是 `SELECT` 语句包含 `UNION` 运算符，则排序列必须包含在选择列表中。例如，下面的语句将产生错误。

```
SELECT DISTINCT ProductLine, ProductModelID
FROM Production.Product
ORDER BY ProductID;
```

注意：对 `ntext`、`text`、`image` 或 `xml` 列不能使用 `ORDER BY` 子句。

5.6.2 指定排序顺序

可以使用 `ASC` 或 `DESC` 关键字指定按升序或降序排序。在未指定的情况下，默认为升序。例如，下面的查询指定按 `ProductID` 降序排序。

```
SELECT ProductID, ProductLine, ProductModelID
FROM Production.Product
ORDER BY ProductID DESC;
```

如果 `ORDER BY` 子句中指定了多个列，则排序是嵌套的。例如，下面的语句首先按产品子类 (`ProductSubcategoryID` 列) 降序排列 `Production.Product` 表中的行，然后在每个产品子类中按 `ListPrice` 升序排序这些行。

```
SELECT ProductID, ProductSubcategoryID, ListPrice
FROM Production.Product
ORDER BY ProductSubcategoryID DESC, ListPrice;
```

在 `ORDER BY` 排序列中不允许使用子查询、聚合和常量表达式，但是可以使用在选择列表中

为聚合或表达式指定的别名。参考下面的语句：

```
SELECT Color, AVG (ListPrice) AS 'average list price'  
FROM Production.Product  
GROUP BY Color  
ORDER BY 'average list price';
```

ORDER BY 只保证查询的最外面的 **SELECT** 语句的排序结果。例如，以下语句定义了一个名为 **TopView** 的视图：

```
CREATE VIEW TopView AS  
SELECT TOP 50 PERCENT * FROM Person.Contact  
ORDER BY LastName
```

然后查询视图：

```
SELECT * FROM TopView
```

尽管视图定义中包含 **ORDER BY** 子句，但是该 **ORDER BY** 子句仅用于确定 **TOP** 子句返回的行数。查询视图自身时，并不能保证对结果进行排序，除非像下面这样明确指定。

```
SELECT * FROM TopView  
ORDER BY LastName
```

5.6.3 指定排序规则

排序规则不同，所生成查询结果的排序也不同。并且排序规则不同的两列，也无法进行比较。可以通过 **COLLATE** 子句指定一个排序规则，而不是按表或视图中所定义的列的排序规则排序。**COLLATE** 仅适用于 **char**、**varchar**、**nchar** 和 **nvarchar** 数据类型的列。

可以在创建数据库、表时指定排序规则，也可以在创建后重新修改或转换排序规则。在创建数据库、表时，如果未指定排序规则，则会使用 服务器实例的默认排序规则。

COLLATE 可以引用数据库服务器或 Windows 排序规则的名称，通过系统函数 **fn_helpcollations** 可以检索到 Windows 和数据库服务器排序规则的所有有效排序规则名称。如：

```
SELECT *  
FROM fn_helpcollations();
```

1. 排序规则的类型

根据表现形式的不同，排序规则可以分为 4 种类型，分别是强制默认、隐式、显式和无排序规则。

(1) 强制默认。

任何字符串变量、参数、文字、目录内置函数的输出，或是没有字符串输入但生成字符串输出的内置函数，会使用数据库默认排序规则。

如果在用户定义函数、存储过程或触发器中声明对象，则为该对象分配创建函数、存储过程或触发器所采用的数据库默认排序规则。如果在批处理中声明对象，则为该对象分配用于连接的当前数据库的默认排序规则。

(2) 隐式。

当在查询中引用列时，查询结果中的相应列会隐式地得到该列的排序规则。

(3) 显式。

使用表达式中的 COLLATE 子句可以将当前排序规则显式转换为特定排序规则。

(4) 无排序规则。

当表达式的值是两个字符串的运算结果，并且这两个字符串具有隐式排序规则冲突，表达式的结果被定义为无排序规则。

2. 排序规则的优先顺序

当仅引用一个字符串对象时，表达式的排序规则是被引用对象的排序规则。当被引用两个操作数表达式的排序规则相同时，被计算表达式的排序规则为操作数表达式的排序规则。

如果被引用两个操作数表达式的排序规则不同，则被计算表达式的排序规则基于下列规则：

- 显式优先于隐式，隐式优先于强制默认，即：显式>隐式>强制默认。
- 组合两个已被分配有不同排序规则的显式表达式将生成错误，即：显式 X+显式 Y=错误。
- 组合两个具有不同排序规则的隐式表达式将生成无排序规则的结果，即：隐式 X+隐式 Y=无排序规则。
- 将无排序规则的表达式与具有显式排序规则的表达式组合，将生成显式的表达式，即：无排序规则+显式 X=显式。
- 将无排序规则的表达式与除显式排序规则之外任何表达式组合，都将生成无排序规则的结果，即：无排序规则+任何内容=无排序规则。

此外，下列规则也适用于排序规则优先顺序：

- 在已经是显式表达式的表达式上不能有多余个 COLLATE 子句。例如，下面的 WHERE 子句无效，因为已经为显式表达式指定了 COLLATE 子句。

```
WHERE ColumnA = ( 'abc' COLLATE French_CI_AS ) COLLATE French_CS_AS
```

- 不允许进行 text 数据类型的代码页转换。如果排序规则的代码页不同，则不能将 text 表达式从一种排序规则转换为另一种排序规则。如果右边文本操作数的排序规则代码页与左边文本操作数的排序规则代码页不同，则不能为赋值运算符赋值。

例如，下面的语句首先创建一个 TestTab 表，并为 CharCol 指定排序规则为 French_CI_AS。

进行查询时，由于表达式 `N'abc'` 的 `Unicode` 数据类型有更高的数据类型优先级，因此生成的表达式将 `Unicode` 数据类型分配给 `N'abc'`。但是，表达式 `CharCol` 具有隐式排序规则，而 `N'abc'` 是 `Transact-SQL` 字符串，属于级别更低的强制默认排序规则。因此，`CharCol` 的最终排序规则是 `French_CI_AS`。

```
CREATE TABLE TestTab (
    id int,
    GreekCol nvarchar(10) collate greek_ci_as,
    LatinCol nvarchar(10) collate latin1_general_cs_as
)
INSERT TestTab VALUES (1, N'A', N'a');
GO

SELECT *
FROM TestTab
WHERE LatinCol LIKE N'a';
```

下面的语句中，由于 `GreekCol` 和 `LatinCol` 具有不同排序规则，因此会产生错误。

```
SELECT *
FROM TestTab
WHERE GreekCol = LatinCol;
```

可以使用下面的方式来得到查询结果。由于右表达式以显式方式指定了排序规则 `Latin1_General_CI_AS`，它的优先级高于左表达式的隐式排序规则。

```
SELECT *
FROM TestTab
WHERE GreekCol = LatinCol COLLATE Latin1_General_CI_AS;
```

在下面的语句中，由于 `GreekCol` 和 `LatinCol` 具有隐式排序规则冲突，因此 `CASE` 表达式具有无排序规则。虽然 `CASE` 不区分排序规则，但是由于计算结果是要放入列中，这会导致列无法确定要使用的排序规则，产生错误。

```
SELECT (CASE WHEN id > 10 THEN GreekCol ELSE LatinCol END)
FROM TestTab;
```

下面是错误提示信息：

```
消息 451，级别 16，状态 1，第 1 行
无法解决列 1(在 SELECT 语句中)的排序规则冲突。
```

下面的方式可以解决隐式排序规则冲突问题，得到正确的查询结果。

```
SELECT (CASE WHEN id > 10 THEN GreekCol ELSE LatinCol END) COLLATE Latin1_General_CI_AS
FROM TestTab
```

3. 区分排序规则与不区分排序规则

运算符和函数可以区分排序规则，也可以不区分排序规则。在区分排序规则的情况下，指定无排序规则操作数会产生一个编译时错误。

(1) 运算符和排序规则。

比较运算符以及 MAX、MIN、BETWEEN、LIKE 和 IN 运算符都区分排序规则。运算符所使用的字符串被赋以具有较高优先顺序的操作数的排序规则。UNION 运算符也区分排序规则，且所有字符串操作数和最终结果被赋以具有最高优先顺序的操作数的排序规则。按列评估 UNION 操作数和结果的排序规则优先顺序。

赋值运算符不区分排序规则，右边的表达式转换到左边的排序规则上。

字符串串联运算符不区分排序规则，两个字符串操作数和结果被赋以具有最高排序规则优先顺序的操作数的排序规则标签。UNION ALL 和 CASE 运算符不区分排序规则，所有的字符串操作数和最终结果都被赋以具有最高优先顺序的操作数的排序规则。按列评估 UNION ALL 操作数和结果的排序规则优先顺序。

(2) 函数和排序规则。

对于 char、varchar 和 text 数据类型，CAST、CONVERT 和 COLLATE 函数区分排序规则。如果 CAST 和 CONVERT 函数的输入和输出是字符串，则输出字符串具有输入字符串的排序规则。如果输入不是字符串，则输出字符串为强制默认并被赋以连接所使用的当前数据库的排序规则，或是包含引用 CAST 或 CONVERT 的用户定义函数、存储过程或触发器的数据库的排序规则。

对于返回字符串但不使用字符串输入的内置函数，结果字符串为强制默认并被赋以当前数据库的排序规则，或是包含引用该函数的用户定义函数、存储过程或触发器的数据库的排序规则。

下面的函数区分排序规则，并且它们的输出字符串具有输入字符串的排序规则标签：

CHARINDEX	LOWER	SOUNDEX
DIFFERENCE	PATINDEX	STUFF
ISNUMERIC	REPLACE	SUBSTRING
LEFT	REVERSE	UPPER
LEN	RIGHT	

5.7 使用 TOP 子句和 SET ROWCOUNT 限制结果集

可以使用 TOP 子句限制结果集中返回的行数。指定返回行数的方法有两种，既可以具体的指定数量，也可以是某一百分比数量的行。TOP 表达式也可用在 INSERT、UPDATE 和 DELETE 语句中，分别用于限制插入、更新和删除的行数。也可以使用 SET ROWCOUNT 语句通过停止查询处理的方式限制返回的结果集。

5.7.1 使用 TOP 子句返回前几行

TOP 子句的语法格式如下：

```
TOP ( expression ) [ PERCENT ] [ WITH TIES ]
```

expression 是指定返回行数的数值表达式，如果指定了 **PERCENT**，则是指返回的结果集行的百分比。

例如，下面的语句指定从 **Sales.SalesOrderHeader** 表返回 10 行数据。

```
SELECT TOP 10 *
FROM Sales.SalesOrderHeader;
```

下面的语句使用 **PERCENT** 指定从 **Sales.SalesOrderHeader** 表返回 15% 的行。

```
SELECT TOP 10 PERCENT *
FROM Sales.SalesOrderHeader;
```

如果在表达式指定返回的行数或百分比，则必须将表达式包含在括号内。参考下面的语句：

```
DECLARE @n AS int;
SET @n = 10
SELECT TOP (@n) *
FROM Sales.SalesOrderHeader ;
```

需要注意的一个问题是，在未指定 **ORDER BY** 子句的情况下，**TOP** 子句返回的数据行，是那些在物理循序上优先访问到的行，而并不一定是逻辑上的前几行。也就是说，这种返回结果具有不确定性。即使指定了 **ORDER BY** 子句，但是所指定的排序列中含有重复值，返回的结果也具有不确定性。不确定的数据，对于数据使用而言，没有多少价值。下面来看一个表 5-14 所示的 **Sales.Orders** 表数据，具体说明一下。

表 5-14

Sales.Orders 表中的内容

OrderID	OrderDate	SubTotal
1	2008-5-1 0:00:00	2190.32
2	2008-5-1 0:00:00	2321.33
3	2008-5-1 0:00:00	1333.01
4	2008-5-2 0:00:00	6513.89
5	2008-5-2 0:00:00	5219.87
6	2008-5-2 0:00:00	11214.52
7	2008-5-3 0:00:00	290.44
8	2008-5-3 0:00:00	2311.67

下面的语句指定按 **OrderID** 列排序后，返回前 2 行数据。由于 **OrderID** 列中的数据是唯一的，因此它会始终返回 **OrderID** 为 1 和 2 的行。


```
SELECT TOP 2 *
FROM Sales.Orders
ORDER BY OrderID;
```

下面的语句指定按 `OrderDate` 列排序，并返回前 2 行数据。从表 5-14 中可以看出，`OrderDate` 为 2008-5-1 的有 3 行，这时返回的查询结果有可能是 `OrderID` 为 1 和 2 行，也可能是 `OrderID` 为 1 和 3 的行，或是 `OrderID` 为 2 和 3 的行。

```
SELECT TOP 2 *
FROM Sales.Orders
ORDER BY OrderDate;
```

要解决列中存在重复值时返回结果的不确定性问题，可以使用 `WITH TIES`。该关键字将指定返回包含 `ORDER BY` 子句返回的最后一个值的所有行，这样将超过 `expression` 指定的数量。下面的语句在返回结果集中前 2 行记录的同时，将返回 `OrderDate` 的最后一个值（即 2008-5-1 0:00:00）的所有记录。也就是说，下面的语句将返回表 5-14 中的 `OrderID` 为 1~3 的行。下面的语句在包含 `WITH TIES` 的情况下，使用 `TOP 1`、`TOP 2`、`TOP 3` 返回的结果集是相同的。

```
SELECT TOP 2 WITH TIES *
FROM Sales.Orders
ORDER BY OrderDate
```

5.7.2 使用 SET ROWCOUNT 语句限制结果集大小

也可以使用 `SET ROWCOUNT n` 语句限制结果集的大小，该语句指定在返回指定的 `n` 行后停止处理查询。`SET ROWCOUNT` 与 `TOP` 的不同之处体现在以下几方面。

- `SET ROWCOUNT` 限制适用于计算 `ORDER BY` 后在结果集中生成行。如果指定了 `ORDER BY`，`SELECT` 语句将从分类排序后的某个值集中选择 `n` 行后结束。

- `TOP` 子句适用于指定了该子句的单个 `SELECT` 语句。直到执行下一个 `SET ROWCOUNT` 语句前，`SET ROWCOUNT` 设置将一直有效。如果执行 `SET ROWCOUNT 0` 将关闭该选项。

需要注意的是，`TOP` 与 `SELECT` 一起使用要优于使用 `SET ROWCOUNT`，应当尽量避免使用 `SET ROWCOUNT`。

下面的 `SELECT` 语句虽然指定了要返回 10 行数据，但是查询了 4 行后，会停止查询。

```
SET ROWCOUNT 4;
SELECT TOP 10 *
FROM Sales.SalesOrderHeader;
```

5.8 使用 DISTINCT 消除重复行

使用 `DISTINCT` 关键字可以从 `SELECT` 语句的结果中消除重复的行。例如，假设有一个 `Customers` 表，表中的内容如表 5-15 所示。

表 5-15

Customers 表

CustID	CustName
1	Grace
2	Grace
3	Ken
3	Ken
4	Holly

下面的语句将消除 CustName 中有重复名称的行，返回的查询结果如表 5-16 所示。

```
SELECT DISTINCT CustName
FROM Customers;
```

表 5-16

消除具有重复名称后的查询结果

CustName
Grace
Holly
Ken

而下面的语句则消除 CustID 和 CustName 都重复的行，两个 Grace 由于 CustID 不同，所以不会被消除。返回的结果如表 5-17 所示。

```
SELECT DISTINCT CustID, CustName
FROM Customers;
```

表 5-17

消除具有重复 ID 和名称后的查询结果

CustID	CustName
1	Grace
2	Grace
3	Ken
4	Holly

需要注意的是，对于 DISTINCT 关键字来说，空值将被认为是相互重复的内容。当 SELECT 语句中包括 DISTINCT 时，不论遇到多少个空值，结果中只返回一个 NULL。

5.9 查询的逻辑处理

在介绍了查询的一些基本知识后，我们来看介绍查询的逻辑处理步骤。虽然本节中的内容会涉

及一些尚未讲述的知识，如表之间的联接等，这看似不太合适。但是，在了解了查询的逻辑处理步骤后，读者能够更好地理解后续章节中介绍的各种复杂查询的原理。

之所以使用“逻辑”处理的提法，因为查询在实际执行前需要经过查询优化器的编译，处于性能考虑，实际的物理处理过程可能与我们介绍的逻辑处理过程有所不同。查询优化器决定了表的访问顺序、访问方法和所使用的索引、联接算法等，优化器会在生成的多个有效执行计划中选择成本最低的执行计划。也就是说，在保证查询结果正确的情况下，优化器会选择一些捷径。也可以说，逻辑处理过程能够帮助我们正确理解查询的逻辑关系，而实际的物理处理过程却能够帮助我们进行查询性能的优化。

5.9.1 逻辑处理过程简介

查询的逻辑处理过程是分阶段完成的，每个阶段都会产生一个虚拟表，该虚拟表会作为下一个阶段的输入。但是，这些过程中间阶段生成的虚拟表对于查询用户是不可用的，只有最后阶段所生成的虚拟表（即查询结果）才返回给查询用户。

最初介绍查询逻辑处理步骤的是 Kalen Delaney (SQL Server MVP, http://sqlblog.com/blogs/kalen_delaney/default.aspx)，在她的新书《Inside Microsoft SQL Server 2008:T-SQL Querying》中将查询分为如下 6 个步骤，而不再是《Inside Microsoft SQL Server 2005:T-SQL Querying》中的 10 个步骤。其中的 3-CR 步骤是我们补充上的。

```
(5)SELECT (5-2)DISTINCT (5-3)<TOP_specification> (5-1)<select_list>
(1)FROM (1-J) <left_table> <join_type> JOIN <right_table> ON <join_condition>
    |(1-A) <left_table> <apply_type> APPLY <right_table_expression> AS <alias>
    |(1-P) <left_table> PIVOT(<pivot_specifications>) AS <alias>
    |(1-U) <left_table> UNPIVOT(<unpivot_specifications>) AS <alias>
(2)WHERE <where_condition>
(3)GROUP BY <group_by_list> (3-CR)WITH {CUBE | ROLLUP}
(4)HAVING <having_condition>
(6)ORDER BY <order_by_list>
```

图 5-1 则更详细地描述了各个处理步骤的流程。

由上面的步骤号可以看出，查询语句的逻辑处理顺序不同于其他编程语言。在大多数编程语言中，代码按编码顺序被处理。但在查询语句中，第 1 个被处理的是 FROM 子句，虽然 SELECT 语句第 1 个出现，但几乎总是在最后被处理。

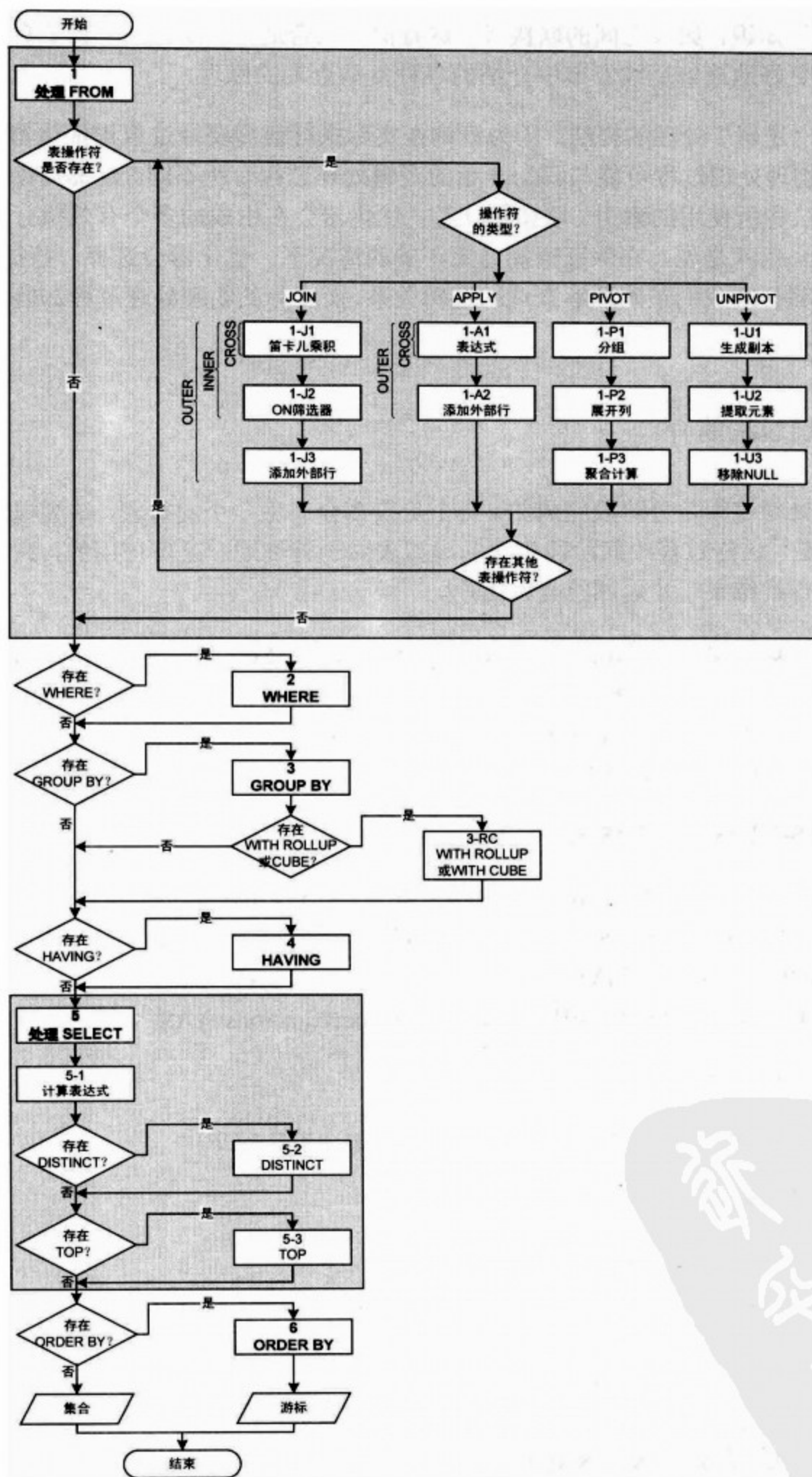


图 5-1 逻辑查询处理步骤流程

注意：步骤 3-CR 中的 WITH ROLLUP 和 WITH CUBE 参数，在 SQL Server 2008 中已经被 GROUP BY 子句的 GROUPING SETS、ROLLUP 和 CUBE 运算符代替，不再推荐使用不符合 ISO 标准的 WITH ROLLUP、WITH CUBE 和 ALL 语法。但是，这并不影响逻辑处理的顺序。

下面是对逻辑处理过程中各个步骤的说明，请注意虚拟表（VTn）的生成步骤。

(1) 步骤 1 (FROM)：该步骤中用于验证查询的源表，并处理表操作符。每个表操作符应用于一系列子步骤。例如，在上面用于联接的 (1-J) 步骤中会涉及如下的子步骤。最终这些子步骤完成后，将生成虚拟表 VT1。

- (1-J1)：执行 left_table 和 right_table 两个表的交叉联接（笛卡尔乘积），生成虚拟表 VT1-J1。
- (1-J2)：对笛卡尔乘积应用 ON 筛选器，生成虚拟表 VT1-J2。
- (1-J3)：如果是外部联接，会在该步骤中将被 ON 筛选掉的外部行添加到 VT1-J2 中，生成 VT1-J3。否则，将跳过该步骤。

(2) 步骤 2 (WHERE)：对 VT1 应用 WHERE 筛选器，将符合筛选条件的行插入到 VT2 中。

(3) 步骤 3 (GROUP BY)：按 GROUP BY 子句中的列表对 VT2 中的行分组，生成 VT3。如果语句中包含 WITH CUBE 或 WITH ROLLUP，则将分组统计结果再次加总后插入 VT3，生成 VT3-RC。

(4) 步骤 4 (HAVING)：对 VT3 应用 HAVING 筛选器，将符合筛选条件的行插入到 VT4。

(5) 步骤 5 (SELECT)：处理 SELECT 子句中的元素，生成 VT5。

- (5-1) 计算表达式：该步骤计算 SELECT 列表中的表达式，生成 VT5-1。
- (5-2) DISTINCT：从 VT5-1 中移除重复行，生成 VT5-2。
- (5-3) TOP：该步骤根据 ORDER BY 子句中指定的排序规则，从 VT5-2 的开始处筛选出指定数量或比例的行。

(6) 步骤 6 (ORDER BY)：该步骤对 VT5-3 中的行按 ORDER BY 子句中的列列表进行排序，生成一个游标 VC6。

为了详细说明各步骤的处理方法，我们创建两个示例表 Customers 和 Orders，并向其中插入数据，内容分别如表 5-18 和表 5-19 所示。其中，Customers 存放的是客户的 ID 和客户的城市信息，Orders 存放的是订单的 ID 和订单所属客户的 ID。

```
USE master;
GO
IF DB_ID (N'mytest') IS NOT NULL
    DROP DATABASE mytest;
GO
CREATE DATABASE mytest;
GO
```

```
USE mytest;
GO

IF OBJECT_ID(N'dbo.Customers') IS NOT NULL
    DROP TABLE dbo.Customers;
IF OBJECT_ID(N'dbo.Orders') IS NOT NULL
    DROP TABLE dbo.Orders;
GO

CREATE TABLE dbo.Customers
(
    CustID CHAR(5) NOT NULL PRIMARY KEY,
    City CHAR(20) NOT NULL
);
CREATE TABLE dbo.Orders
(
    OrderID INT NOT NULL PRIMARY KEY,
    CustID CHAR(5) NULL
);
GO

INSERT INTO dbo.Customers
VALUES('C1', 'Shanghai'),
('C2', 'Beijing'),
('C3', 'Beijing'),
('C4', 'Beijing');
INSERT INTO dbo.Orders
VALUES(1, 'C1'),
(2, 'C2'),
(3, 'C2'),
(4, 'C3'),
(5, 'C3'),
(6, 'C3'),
(7, NULL);
GO
```

表 5-18

Customers 表的内容

CustID	City
C1	Shanghai
C2	Beijing
C3	Beijing
C4	Beijing

表 5-19

Orders 表的内容

OrderID	CustID
1	C1
2	C2
3	C2
4	C3
5	C3
6	C3
7	NULL

下面的是一个具有左外联接的语句，用于查询来自 Beijing 且订单数小于 3 的客户，查询结果按订单数增量排序，如表 5-20 所示。实际上，分析 Orders 表的 CustID 列，也可以直接看出只有 C2 和 C4 客户的订单数小于 3，并且来自 Beijing。在下面的语句中，除了联接未讲述外，其他部分在第 2 章中都有介绍。

```
SELECT C.CustID, COUNT(O.OrderID) AS OrderNum
FROM dbo.Customers AS C
LEFT OUTER JOIN dbo.Orders AS O
ON C.CustID = O.CustID
WHERE C.City = 'Beijing'
GROUP BY C.CustID
HAVING COUNT(O.OrderID) < 3
ORDER BY OrderNum;
```

表 5-20 查询来自 Beijing 且订单数小于 3 的客户

CustID	OrderNum
C4	0
C2	2

5.9.2 步骤 1：使用 FROM 确定输入表

该步骤首先识别被进行查询的表，如果指定了表操作符，则还要从左至右依次处理这些操作符。每个操作符基于一个或两个输入表，并返回一个输出表。该输出表会被作为下一个表操作符的左输入，如果不存在下一个表操作符，则会作为下一逻辑处理步骤的输入。

每个表操作符都具有自己的子处理步骤，像前面图 6-1 中描述的 JOIN 操作符，具有 1-J1、1-J2 和 1-J3 三个步骤。下面我们将以 JOIN 操作符为例，详细讲述一下这 3 个步骤中的数据处理方法。

1. 步骤 1-J1：执行笛卡尔乘积

JOIN 操作处理的第 1 步是联接所涉及的两个表执行交叉联接，也就是笛卡尔乘积，生成虚拟表 VT1-J1。该表中将包含左表和右表中每一行的所有可能选择。例如，如果左表有 n 行，右表有 m 行，则该表将具有 $n \times m$ 行。将 Customers 和 Orders 进行交叉联接，Customers 表中的每一行与 Orders 表的所有行进行联接，因此，所生成虚拟表 VT1-J1 的大小为 $4 \times 7 = 28$ 。

VT1-J1 中列的名称以源表名或别名进行限定。尤其是对于两个表中都存在的列，这种限定则是更为重要，因为所生成的虚拟表中不可能存在两个同名的列。VT1-J1 的内容中如表 5-21 所示。

表 5-21 VT1-J1 的内容

C.CustID	C.City	O.OrderID	O.CustID
C1	Shanghai	1	C1
C1	Shanghai	2	C2
C1	Shanghai	3	C2

续表

C.CustID	C.City	O.OrderID	O.CustID
C1	Shanghai	4	C3
C1	Shanghai	5	C3
C1	Shanghai	6	C3
C1	Shanghai	7	NULL
C2	Beijing	1	C1
C2	Beijing	2	C2
C2	Beijing	3	C2
C2	Beijing	4	C3
C2	Beijing	5	C3
C2	Beijing	6	C3
C2	Beijing	7	NULL
C3	Beijing	1	C1
C3	Beijing	2	C2
C3	Beijing	3	C2
C3	Beijing	4	C3
C3	Beijing	5	C3
C3	Beijing	6	C3
C3	Beijing	7	NULL
C4	Beijing	1	C1
C4	Beijing	2	C2
C4	Beijing	3	C2
C4	Beijing	4	C3
C4	Beijing	5	C3
C4	Beijing	6	C3
C4	Beijing	7	NULL

2. 步骤 1-J2: 使用 ON 筛选器 (联接条件)

在查询的 3 个筛选器 ON、WHERE 和 HAVING 中，ON 是第 1 个被使用的，其次是 WHERE，最后才是 HAVING。根据 ON 指定的联接条件，对 VT1-J1 中的所有行进行筛选，只有计算结果为 TRUE 的行才会被包含在本步骤要生成的虚拟表 VT1-J2 中。

由表 5-21 所示的 VT1-J1 可以看出，在 C.CustID 与 O.CustID 的比较值中，存在与 NULL 值的比较问题。我们知道，NULL 代表空值。但是空值不同于空白或零值，它表示的是一种未知状态。没有两个相等的空值，比较两个空值或将空值与任何其他值相比均返回未知，这是因为每个空值均为未知。也就是说，如果数据中出现 NULL 值，则逻辑运算符和比较运算符有可能返回 TRUE 或 FALSE 以外的第三种结果 UNKNOWN，这被称为“三值逻辑”。

NULL 值通过任何比较运算符与任何值（包括 NULL 值）进行比较时都是 UNKNOWN，返回 TRUE 的唯一搜索条件是 IS NULL 谓词。在 SQL 中，只有筛选条件计算为 TRUE 时才选择行，不选择值为 UNKNOWN 或 FALSE 的行。

假设某个表的 Salary 列为 NULL，表 5-22 列出了所有可能条件的计算结果。

表 5-22 包含 NULL 值的条件计算结果

条 件	值
Salary = NULL	UNKNOWN
Salary <> NULL	UNKNOWN
NOT (Salary = NULL)	UNKNOWN
NOT (Salary <> NULL)	UNKNOWN
Salary = 1000	UNKNOWN
Salary IS NULL	TRUE
Salary IS NOT NULL	FALSE

此外，Transact-SQL 也提供了对 NULL 值处理的扩展功能。如果 ANSI_NULLS 选项设置为 OFF，则空值之间的比较（如 NULL = NULL）返回 TRUE。空值与任何其他数据值之间的比较都返回 FALSE。

根据表 5-22 中列举的条件和计算结果，当使用 ON C.CustID = O.CustID 条件对 VT1-J1 进行计算时，可以很容易得出如表 5-23 所示的结果。

表 5-23 对 VT1-J1 中的行应用 ON C.CustID = O.CustID 条件后计算出的逻辑结果

C.CustID	C.City	O.OrderID	O.CustID	逻辑结果
C1	Shanghai	1	C1	TRUE
C1	Shanghai	2	C2	FALSE
C1	Shanghai	3	C2	FALSE
C1	Shanghai	4	C3	FALSE
C1	Shanghai	5	C3	FALSE
C1	Shanghai	6	C3	FALSE
C1	Shanghai	7	NULL	UNKNOWN
C2	Beijing	1	C1	FALSE
C2	Beijing	2	C2	TRUE
C2	Beijing	3	C2	TRUE
C2	Beijing	4	C3	FALSE
C2	Beijing	5	C3	FALSE
C2	Beijing	6	C3	FALSE
C2	Beijing	7	NULL	UNKNOWN
C3	Beijing	1	C1	FALSE

续表

C.CustID	C.City	O.OrderID	O.CustID	逻辑结果
C3	Beijing	2	C2	FALSE
C3	Beijing	3	C2	FALSE
C3	Beijing	4	C3	TRUE
C3	Beijing	5	C3	TRUE
C3	Beijing	6	C3	TRUE
C3	Beijing	7	NULL	UNKNOWN
C4	Beijing	1	C1	FALSE
C4	Beijing	2	C2	FALSE
C4	Beijing	3	C2	FALSE
C4	Beijing	4	C3	FALSE
C4	Beijing	5	C3	FALSE
C4	Beijing	6	C3	FALSE
C4	Beijing	7	NULL	UNKNOWN

只有逻辑结果为 TRUE 的行才会被插入到虚拟表 VT1-J2 中，如表 5-24 所示。

表 5-24

VT1-J2 的内容

C.CustID	C.City	O.OrderID	O.CustID	逻辑结果
C1	Shanghai	1	C1	TRUE
C2	Beijing	2	C2	TRUE
C2	Beijing	3	C2	TRUE
C3	Beijing	4	C3	TRUE
C3	Beijing	5	C3	TRUE
C3	Beijing	6	C3	TRUE

3. 步骤 1-J3: 添加外部行

这一步仅与在查询中是否指定了外部联接 (OUTER JOIN) 有关。外部联接会返回 FROM 子句中提到的至少一个表或视图中的所有行，然后根据指定的联接条件返回另外一个表中的匹配行。根据联接时获取一个全部行表的位置 (ON 条件的左边表或右边表) 不同，可以将外部联接分为左外部联接、右外部联接和完全外部联接。左外部联接将返回左边表中的全部行和右边表中的匹配行，而右外部联接则是返回右边表中的全部行和左边表中的匹配行，完全外部联接是在左边表与右边表匹配行的基础上，再返回两个表中剩余的全部行。被保留全部行的我们称为“保留表”。

在该示例中使用的左外部联接，左边表为 Customers。通过表 5-24 可以看出，仅有 C4 未找到匹配行，因此 C4 被添加到上面的 VT1-J2 中，O.OrderID 和 O.CustID 为空值，生成虚拟表 VT1-J3，如表 5-25 所示。

表 5-25 VT1-J3 中的内容

C.CustID	C.City	O.OrderID	O.CustID
C1	Shanghai	1	C1
C2	Beijing	2	C2
C2	Beijing	3	C2
C3	Beijing	4	C3
C3	Beijing	5	C3
C3	Beijing	6	C3
C4	Beijing	NULL	NULL

如果联接两个以上的表，则将 VT1-J3 与第 3 个要联接的表重复执行步骤 1-J1、1-J2 和 1-J3，以此类推，直至联接完成。最终的虚拟表被作为下一步骤的输入。

5.9.3 步骤 2：使用 WHERE 筛选数据

在该步骤中，将对 VT1-J3 中的所有行使用 WHERE 筛选器，只有符合 C.City = 'Beijing' 条件的行才会被放入到虚拟表 VT2 中。

前面已经提到了，在 ON、WHERE 和 HAVING 这 3 个筛选器中，ON 是第 1 个被使用的，其次是 WHERE。这里读者可能有一个疑问，能否将两个筛选条件，或是包括 HAVING 在内的 3 个筛选条件，使用 AND 逻辑运算符直接联接在一起，使用一条语句完成，即类似下面的形式：

```
SELECT C.CustID, COUNT(O.OrderID) AS OrderNum
FROM dbo.Customers AS C
LEFT OUTER JOIN dbo.Orders AS O
ON C.CustID = O.CustID AND C.City = 'Beijing'
GROUP BY C.CustID
HAVING COUNT(O.OrderID) < 3
ORDER BY OrderNum;
```

这样做是不行的，因为 ON 是在添加外部行之前被应用，当我们挑选出符合条件的行后，还会把 Customers 中剩余的全部行添加回来。执行上面的语句会得到如表 5-26 所示的错误结果，可以看出 City 为 NULL 的 C4 和 City 为 Shanghai 的行又都被添加了回来。

表 5-26 使用 AND 进行条件归并后得到的错误结果

CustID	OrderNum
C1	0
C4	0
C2	2

自然，这种情况仅发生在查询中包含外部联接的情况下，如果是内部联接，由于不存在步骤 3 的添加外部行动作，无论在哪里指定筛选条件都无所谓。读者可以分析下面的 3 个查询，它们得到的结果是完全一样的，如表 5-27 所示。

```

-- 正常的书写方式
SELECT C.CustID, COUNT(O.OrderID) AS OrderNum
FROM dbo.Customers AS C
JOIN dbo.Orders AS O
  ON C.CustID = O.CustID
WHERE C.City = 'Beijing'
GROUP BY C.CustID
HAVING COUNT(O.OrderID) < 3
ORDER BY OrderNum;

-- 将筛选条件全部写在了 ON 中
SELECT C.CustID, COUNT(O.OrderID) AS OrderNum
FROM dbo.Customers AS C
JOIN dbo.Orders AS O
  ON C.CustID = O.CustID AND C.City = 'Beijing'
GROUP BY C.CustID
HAVING COUNT(O.OrderID) < 3
ORDER BY OrderNum;

-- 将筛选条件全部写在了 WHERE 中
SELECT C.CustID, COUNT(O.OrderID) AS OrderNum
FROM dbo.Customers AS C
CROSS JOIN dbo.Orders AS O -- 注意：这里是交叉联接，交叉的结果会得到步骤 1 的 VT1 表
WHERE C.CustID = O.CustID AND C.City = 'Beijing'
GROUP BY C.CustID
HAVING COUNT(O.OrderID) < 3
ORDER BY OrderNum

```

表 5-27 多种筛选条件书写方式得到的相同结果

CustID	OrderNum
C2	2

此外，WHERE 与 HAVING 是否具有区别？在后面的部分中还会进行介绍，下面还是回到对本节问题的讨论中。

由表 5-25 可以看出，在 VT1-J3 中只有 C1 的 City 为 Shanghai，因此需要移除掉，生成的 VT2 如表 5-28 所示。

表 5-28 VT2 中的内容

C.CustID	C.City	O.OrderID	O.CustID
C2	Beijing	2	C2
C2	Beijing	3	C2
C3	Beijing	4	C3
C3	Beijing	5	C3
C3	Beijing	6	C3
C4	Beijing	NULL	NULL

5.9.4 步骤 3：进行数据分组

在该步骤中，将根据在 GROUP BY 子句中指定的分组条件 C.CustID，对 VT2 进行数据分组。

每一行被分配到一个组，并且仅分配到一个组，得到虚拟表 VT3。VT3 由两部分组成，一部分是组的构成部分，另一部分是由基行组成的原始部分，如表 5-29 所示。

表 5-29 VT3 中的内容

组	原始行			
C.CustID	C.CustID	C.City	O.OrderID	O.CustID
C2	C2	Beijing	2	C2
	C2	Beijing	3	C2
C3	C3	Beijing	4	C3
	C3	Beijing	5	C3
	C3	Beijing	6	C3
C4	C4	Beijing	NULL	NULL

此步骤以后的步骤中，只能引用 GROUP BY 列表中所包含的列，如果要引用原始部分中的列，则必须对它们进行聚合运算。例如，下面的语句会产生错误，因为 SELECT 列表步骤中包含的 OrderID 列既没包含在 GROUP BY 列表中，也没有使用聚合函数。

```
SELECT CustID, OrderID
FROM dbo.Orders
GROUP BY CustID;
```

如果指定了 GROUP BY ALL，在步骤 3 中使用 WHERE 筛选器移除掉的组将被添加到 VT3 中。无论外部联接还是内部联接，都会存在这种情况。例如，执行下面的查询将得到如表 5-30 所示的结果，可以看到被移除的 C1 又被添加了回来。

```
SELECT C.CustID, COUNT(O.OrderID) AS OrderNum
FROM dbo.Customers AS C
LEFT OUTER JOIN dbo.Orders AS O
ON C.CustID = O.CustID
WHERE C.City = 'Beijing'
GROUP BY ALL C.CustID
ORDER BY OrderNum;
```

表 5-30 使用 GROUP BY ALL 得到的查询结果

CustID	OrderNum
C1	0
C4	0
C2	2
C3	3

由此可以看出，GROUP BY ALL 带来了一些与 WHERE 筛选方面的问题。实际上，GROUP BY ALL 以及后面将要讲述的 WITH CUBE 和 WITH ROLLUP 均不符合 ISO 标准，在 SQL Server 2008 中已经被 GROUP BY 子句的 GROUPING SETS、ROLLUP 和 CUBE 运算符代替。

如果在 GROUP BY 子句后面指定了 WITH CUBE 或 WITH ROLLUP，则会按层次结构顺序，

从组内的最低级别到最高级别生成汇总组，并添加到上一步返回的虚拟表 VT3 中，生成虚拟表 VT3-RC。

5.9.5 步骤 4：使用 HAVING 筛选数据

HAVING 可以用于对已分组数据的筛选，这是也是唯一可用于已分组数据的筛选器。它与 WHERE 最大的区别也正是在于此。在 HAVING 中必须有聚合函数，而 WHERE 中则不能出现聚合函数。

在该步骤中，HAVING 的条件是 $\text{COUNT}(\text{O.OrderID}) < 3$ ，因此 C3 组由于包含 3 个订单而被移除。生成的虚拟表 VT4 如表 5-31 所示。

表 5-31

VT4 中的内容

组	原始行			
C.CustID	C.CustID	C.City	O.OrderID	O.CustID
C2	C2	Beijing	2	C2
	C2	Beijing	3	C2
C4	C4	Beijing	NULL	NULL

5.9.6 步骤 5：通过 SELECT 列表确定返回列

SELECT 列表在查询中虽然是最先被指定的，但是由于它是要处理返回给调用者查询结果的列的列表，因此几乎总是被最后被处理。在该阶段将构建最终返回给调用者的表，它包括计算表达式、使用 DISTINCT 和 TOP 选项。

1. 步骤 5-1：计算表达式

在前面讲过，步骤 3 以后的步骤中，只能引用 GROUP BY 列表中所包含的列，如果要引用原始部分中的列，则必须对它们进行聚合运算。在 SELECT 列表步骤中，也有一个需要注意的地方，就是为列指定的别名，不能在前面的步骤中使用。未指定别名的情况下，基列将沿用自己的列名称，而对于基于表达式的列，则应当为其指定一个别名。例如，下面的语句将发生错误，因为 GROUP BY 步骤在 SELECT 列表步骤之前，但是它却引用了 SELECT 列表步骤中为列指定的别名。

```
SELECT CustID AS Cus, COUNT(OrderID)
FROM dbo.Orders
GROUP BY Cus;
```

该查询中的 SELECT 列表如下：

```
SELECT C.CustID, COUNT(O.OrderID) AS OrderNum
```

得到的虚拟表 VT5-1，如表 5-32 所示。

表 5-32

VT5-1 中的内容

C.CustID	OrderNum
C2	2
C4	0

2. 步骤 5-2: 使用 DISTINCT 子句

如果在查询中指定了 DISTINCT 子句, 则将从 VT5-1 中移除重复行, 得到虚拟表 VT5-2。由于我们的示例中未指定该子句, 因此会跳过此步骤。

3. 步骤 5-3: 使用 TOP 子句

可以使用 TOP 子句限制结果集中返回的行数或百分比。通常情况下, 该子句要与 ORDER BY 子句结合在一起使用, 以确保所返回行的确定性。如果未指定 ORDER BY 子句, 会返回那些在物理上首先被访问到的行, 每次查询可能会返回不同的结果。

可以使用如下形式指定 TOP 子句:

```
-- 返回结果集的前 2 行
SELECT TOP(2) *
FROM dbo.Orders
ORDER BY OrderID;

-- 返回结果集的前@n 行
DECLARE @n AS BIGINT;
SET @n = 2;
SELECT TOP(@n) *
FROM dbo.Orders
ORDER BY OrderID;

-- 返回结果集的前 15% 的行
SELECT TOP(15)PERCENT *
FROM dbo.Orders
ORDER BY OrderID;
```

在本节的示例中, 由于未指定 TOP 子句, 故而跳过该步骤。

5.9.7 步骤 6: 使用 ORDER BY 子句排序查询结果

在此步骤中, 将按 ORDER BY 子句中的列列表对上面返回的虚拟表进行排序。需要注意的是, 在该步骤中返回的是一个游标 (VC6), 而不是一个虚拟表。在 SQL 中, 数据是一种基于逻辑关系的集合, 成员的顺序无关紧要。对表进行排序的查询可以返回一个对象, 包含特定物理顺序组织的行, 这个对象就是游标。

数据的无序存放, 可以提高数据的插入速度。例如, 假设有一个具有 10 万行的有序表, 如果希望向中间位置插入一行, 则需要移动插入位置后面的所有行, 性能可想而知。无序的存放, 自然也给排序带来了一定的困难, 需要付出一定的性能成本。因此, 除非特别需要, 不要使用 ORDER

BY 子句。

在该步骤中，可以使用 SELECT 列列表中为列指定的别名。为示例使用的排序语句如下：

ORDER BY OrderNum

得到如表 5-33 所示的 VC6。

表 5-33

VC6 游标中的内容

C.CustID	OrderNum
C4	0
C2	2

至此，查询语句的一些基本处理步骤已经全部介绍完毕。对于其中一些重点的地方，读者应当加以注意，如：ON、WHERE 和 HAVING 的顺序等。有了这个基础之后，就可以更好地理解或书写出一些更加复杂的查询语句。



第 6 章 子查询

子查询是一个嵌套在 SELECT、INSERT、UPDATE 或 DELETE 语句或其他子查询中的查询，它很多时候是在充当中间结果集角色，因为并不是所有的查询都可以一蹴而就。子查询也称为内部查询或内部选择，而包含子查询的语句也称为外部查询或外部选择。根据可用内存和查询中其他表达式的复杂程度的不同，嵌套限制也有所不同，但嵌套到 32 层是可能的。个别查询可能不支持 32 层嵌套。任何可以使用表达式的地方都可以使用子查询，只要它返回的是单个值。

子查询可以按多种方式进行分类：按照子查询返回值的数量，可以分为标量子查询和多值子查询；按照子查询对外部查询的依赖性，可以分为独立子查询和相关子查询；按照所使用比较运算符的不同，可以分为使用 IN、EXISTS、ANY、SOME 和 ALL 等多种形式。

标量子查询和多值子查询可以是独立子查询，也可以是相关子查询。独立子查询可以被独立执行，不受外部查询影响，而相关子查询则需要与外部查询联合来确定最终结果集。

6.1 在选择列表中使用子查询

在选择列表中使用的子查询是作为结果集中的一列出现的，因此要求每次调用子查询返回的是一个结果值，而不是一个值列表。此时，子查询属于标量子查询。

6.1.1 子查询示例

使用下面的语句分别创建一个名为 Orders 和 OrderDetail 的表，并向其中插入一些示例数据。Orders 表中存放的订单编号、日期和货物的目的地，OrderDetail 表中存放的是每笔订单中详细的产品编号、数量、单价和小计。注意下面语句中为 OrderDetail 表的 OrderID 列添加了一个外键约束，防止表中出现 Orders 中不存在的订单编号。同时，LineTotal 是一个计算列。

```
CREATE TABLE Orders
(OrderID int NOT NULL PRIMARY KEY,
 OrderDate datetime NOT NULL,
 ShipTo char(20) NOT NULL);
CREATE TABLE OrderDetail
(OrderID int NOT NULL
 FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),
 ProductID int NOT NULL,
 OrderQty int NOT NULL,
```

```

UnitPrice money NOT NULL,
LineTotal AS ISNULL(UnitPrice * OrderQty, 0.0));

INSERT INTO Orders
VALUES (1, '2009-03-01', 'Shanghai'),
       (2, '2009-03-02', 'Beijing');
INSERT INTO OrderDetail
VALUES (1, 714, 1, 500.00),
       (1, 715, 2, 250.00),
       (2, 716, 1, 300.00);

```

Orders 和 OrderDetail 表的内容分别如表 6-1 和表 6-2 所示。

表 6-1 Orders 表中的内容

OrderID	OrderDate	ShipTo
1	2009-03-01 00:00:00.000	Shanghai
2	2009-03-02 00:00:00.000	Beijing

表 6-2 OrderDetail 表中的内容

OrderID	ProductID	OrderQty	UnitPrice	LineTotal
1	714	1	500.00	500.00
1	715	2	250.00	500.00
2	716	1	300.00	300.00

下面的查询将返回每笔订单的编号、日期和销售小计，如表 6-3 所示。该子查询仅为每行返回一个值，因此它是一个标量子查询。同时，它的 WHERE 子句需要与外部查询相关联，因此它又是一个相关子查询。

```

SELECT OrderID, OrderDate, (SELECT SUM(D.LineTotal)
                             FROM OrderDetail AS D
                             WHERE D.OrderID = O.OrderID) AS OrderTotal
FROM Orders AS O

```

表 6-3 查询结果

OrderID	OrderDate	OrderTotal
1	2009-03-01 00:00:00.000	1000.00
2	2009-03-02 00:00:00.000	300.00

该子查询会为每笔订单执行一次。在该示例中，由于 Orders 表中含有 2 行数据，因此该子查询会被执行 2 次。所以，对于相关子查询而言，查询的性能通常会由于数据量的增加，性能下降较为明显。

6.1.2 子查询与联接的关系

许多包含子查询的 SQL 语句都可以改用联接表示，包含子查询的语句和语义上等效的不包含子查询的语句在性能上通常没有差别。但是，在一些必须检查存在性的情况中，使用联接会产生更

好的性能。上面的子查询，可以转换为下面的联接，二者返回的结果相同。

```
SELECT O.OrderID, MAX(O.OrderDate), SUM(D.LineTotal)
FROM Orders AS O
LEFT OUTER JOIN OrderDetail AS D
ON O.OrderID = D.OrderID
GROUP BY O.OrderID
```

注意上面语句中的 **MAX** 函数并没有实际意义，我们只是希望返回的数据中包含 **OrderDate** 列，但是对于未包含在 **GROUP BY** 列表中的列，在 **SELECT** 列表中必须包含在聚合函数中。

6.2 含有 IN 和 EXISTS 的子查询

通过 **IN** 或 **EXISTS** 引入的子查询在功能方面有一些类似，并且所有使用 **IN** 或由 **ANY**、**ALL** 修改的比较运算符的查询都可以通过 **EXISTS** 表示。但是，它们也存在一定的区别。通过 **IN** 引入的子查询结果是包含 0 个或多个值的列表，它表示的是一种值的“等于”关系；而通过 **EXISTS** 引入的子查询实际上不产生任何数据，它只返回 **TRUE** 或 **FALSE** 值，它表示的是一种“存在”行为。

6.2.1 使用含有 IN 的子查询进行单列匹配

首先来看一下含有 **IN** 的子查询的语法格式：

```
SELECT select_list
FROM table_source
WHERE search_expression [NOT] IN (subquery)
```

语句中的 **search_expression** 可以是一个常量值、列名、表达式或子查询。

使用下面的语句分别创建一个名为 **Customers** 和 **OrderHeader** 的表，并插入一些示例数据。其中，**Customers** 中存放的是客户信息，而 **OrderHeader** 中存放的是客户的订单信息。

```
CREATE TABLE Customers
(CustID int NOT NULL PRIMARY KEY,
CustName char(20) NOT NULL,
City char(20) NOT NULL);
CREATE TABLE OrderHeader
(CustID int NOT NULL
FOREIGN KEY (CustID) REFERENCES Customers(CustID),
OrderID int NOT NULL,
OrderDate datetime NOT NULL,
ShipTo char(20) NOT NULL);

INSERT INTO Customers
VALUES (1, 'Zhang Hongju', 'Beijing'),
(2, 'Li Ming', 'Shanghai'),
(3, 'Sun Lihua', 'Beijing'),
(4, 'Wang Gang', 'Beijing');

INSERT INTO OrderHeader
VALUES (1, 110, '2009-04-01', 'Fangshan, Beijing'),
(1, 111, '2009-04-01', 'Haidian, Beijing'),
(2, 113, '2009-04-02', 'Pudong, Shanghai'),
(3, 114, '2009-04-03', 'Chongwen, Beijing');
```

Customers 表和 OrderHeader 表中的内容分别如表 6-4 和表 6-5 所示。

表 6-4 Customers 表中的内容

CustID	CustName	City
1	Zhang Hongju	Beijing
2	Li Ming	Shanghai
3	Sun Lihua	Beijing
4	Wang Gang	Beijing

表 6-5 OrderHeader 表中的内容

CustID	OrderID	OrderDate	ShipTo
1	110	2009-04-01 00:00:00.000	Fangshan, Beijing
1	111	2009-04-01 00:00:00.000	Haidian, Beijing
2	113	2009-04-02 00:00:00.000	Pudong, Shanghai
3	114	2009-04-03 00:00:00.000	Chongwen, Beijing

现在要查询 OrderHeader 中 Beijing 客户的订单，假设我们已经知道哪些客户来自 Beijing（见表 6-4），完全可以把查询语句写成下面的形式。通过这个语句可以看出，IN 需要的是一个值的列表，表示的是一种与值的“等于”关系（即 CustID = 1 OR CustID = 3 OR CustID = 4）。

```
SELECT OrderID, OrderDate, ShipTo
FROM OrderHeader
WHERE CustID IN (1, 3, 4);
```

为了使用上面的查询语句更具有通用性，则可以把括号内的值列表替换为下面的子查询方式。该语句将分两步完成，首先执行子查询，从 Customers 表中取出符合条件的 CustID 信息；然后从 OrderHeader 表中检索出与子查询中 CustID 相同的订单信息。查询结果如表 6-6 所示。

```
SELECT OrderID, OrderDate, ShipTo
FROM OrderHeader
WHERE CustID IN (SELECT CustID
                  FROM Customers
                  WHERE City = 'Beijing');
```

表 6-6 查询结果

OrderID	OrderDate	ShipTo
110	2009-04-01 00:00:00.000	Fangshan, Beijing
111	2009-04-01 00:00:00.000	Haidian, Beijing
114	2009-04-03 00:00:00.000	Chongwen, Beijing

上面的子查询是一个独立子查询，可以单独执行。

6.2.2 使用含有 EXISTS 的子查询进行整行匹配

含有 EXISTS 的子查询的语法格式如下：

```
SELECT select_list
FROM table_source
WHERE [NOT] EXISTS (subquery)
```

含有 EXISTS 的子查询实际上不产生任何数据，在有返回行的情况下，子查询将返回 TRUE，否则，将返回 FALSE。要使 EXISTS 判断有意义，subquery 应当是一个包含搜索条件的查询。由于仅仅是判断是否有返回行，所以子查询的选择列表通常指定为星号 (*)。

仍旧使用表 6-4 和表 6-5 所示的数据。下面的语句在检索 Customers 中每一行时，通过子查询中的 WHERE 条件，在 OrderHeader 中检索是否存在与 Customers.CustID 相等的行。如果存在，则将 Customers 表的 CustName 和 City 列的内容放入到结果集中。下面的语句将返回 Customers 表在 OrderHeader 表中有 ID 的客户的行，查询结果如表 6-7 所示。

```
SELECT CustName, City
FROM Customers
WHERE EXISTS
  (SELECT *
   FROM OrderHeader
   WHERE CustID = Customers.CustID);
```

表 6-7

查询结果

CustName	City
Zhang Hongju	Beijing
Li Ming	Shanghai
Sun Lihua	Beijing

上面的查询是一个相关子查询，子查询中 WHERE 子句的 CustID 根据外部查询的 Customers.CustID 进行数据查询。通常情况下，含有 EXISTS 的是相关子查询，而 IN 则可以是独立子查询，也可以是相关子查询。

通常情况下，使用 EXISTS 表示的查询和使用 IN 表示的查询可以相互转换。例如，上面的语句也可以使用下面的 IN 形式来实现。

```
SELECT CustName, City
FROM Customers
WHERE CustID IN (SELECT CustID
                 FROM OrderHeader);
```

又如，下面是 6.2.1 小节中包含 IN 的子查询：

```
SELECT OrderID, OrderDate, ShipTo
FROM OrderHeader
WHERE CustID IN (SELECT CustID
```

```
FROM Customers
WHERE City = 'Beijing');
```

也可以转换为以下使用 EXISTS 表示的查询:

```
SELECT OrderID, OrderDate, ShipTo
FROM OrderHeader
WHERE EXISTS (SELECT *
              FROM Customers
              WHERE CustID = OrderHeader.CustID
              AND City = 'Beijing');
```

6.2.3 含有 NOT IN 和 NOT EXISTS 的子查询

NOT IN 和 NOT EXISTS 分别用于对 IN 和 EXISTS 进行求反。上文已经讨论了 IN 和 EXISTS 相互转换的可能性。当子查询结果不存在 NULL 值时, NOT IN 和 NOT EXISTS 得到的查询结果也完全相同, 所生成的查询计划也完全相同。

1. 不含有 NULL 值的子查询

仍旧使用表 6-4 和表 6-5 的数据, 执行下面的语句, 查询 Customers 表在 OrderHeader 中没有订单的客户。分析这两个表, 可以看出只有 CustID 为 4 的 Wang Gang 在 OrderHeader 中没有订单。

```
SELECT CustName, City
FROM Customers
WHERE NOT EXISTS
  (SELECT *
   FROM OrderHeader
   WHERE CustID = Customers.CustID);
```

上面的语句的查询结果如表 6-8 所示。

表 6-8	查询结果
CustName	City
Wang Gang	Beijing

将上面的 NOT EXISTS 可以转换如下形式的语句, 查询结果完全相同。并且二者的执行计划也完全相同, 如图 6-1 所示。

```
SELECT CustName, City
FROM Customers
WHERE CustID NOT IN
  (SELECT CustID
   FROM OrderHeader );
```

2. 含有 NULL 值的子查询

当子查询中包含 NULL 值时, NOT IN 和 NOT EXISTS 并不是等价的。也就是说, 他们的返回结果并不相同。

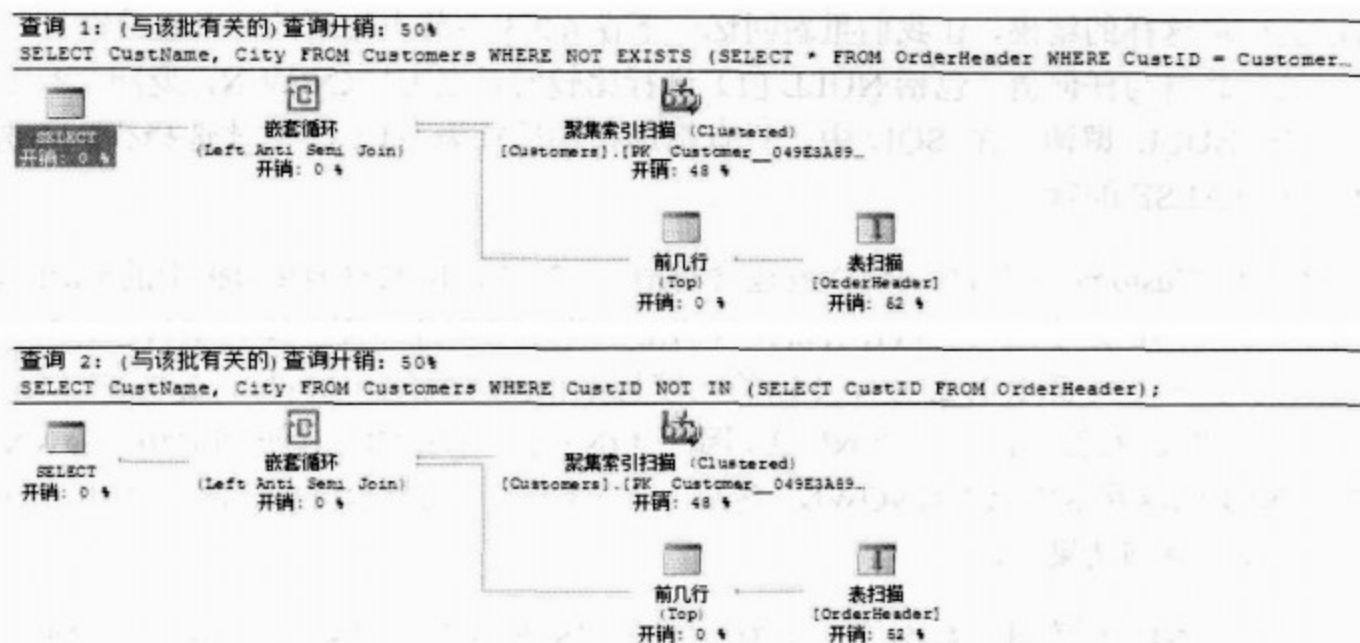


图 6-1 执行计划

执行下面的语句, 重新建立 OrderHeader 表, 允许 CustID 值为 NULL, 并向表中插入包含 NULL 值的行。

```
DROP TABLE OrderHeader;

CREATE TABLE OrderHeader
  (CustID int NULL,
   OrderID int NOT NULL,
   OrderDate datetime NOT NULL,
   ShipTo char(20) NOT NULL);

CREATE INDEX IX_OrderHeader
  ON OrderHeader(CustID);

INSERT INTO OrderHeader
VALUES (1, 110, '2009-04-01', 'Fangshan, Beijing'),
      (1, 111, '2009-04-01', 'Haidian, Beijing'),
      (2, 113, '2009-04-02', 'Pudong, Shanghai'),
      (3, 114, '2009-04-03', 'Chongwen, Beijing'),
      (NULL, 115, '2009-04-03', 'Langfang, Beijing');
```

重新执行下面的两个语句。读者会发现含有 NOT EXISTS 的语句仍旧返回如表 6-8 所示的结果, 而含有 NOT IN 的语句则未返回任何数据。

```
SELECT CustName, City
FROM Customers
WHERE NOT EXISTS
  (SELECT *
   FROM OrderHeader
   WHERE CustID = Customers.CustID);

SELECT CustName, City
FROM Customers
WHERE CustID NOT IN
  (SELECT CustID
   FROM OrderHeader);
```

之所以会出现这样的结果，让我们重新回忆一下在 6.2.3 小节中介绍的“三值逻辑”。NULL 值通过任何比较运算符与任何值（包括 NULL 值）进行比较时都是 UNKNOWN，返回 TRUE 的唯一搜索条件是 IS NULL 谓词。在 SQL 中，只有筛选条件计算为 TRUE 时才选择行，不选择值为 UNKNOWN 或 FALSE 的行。

在该示例中，Customers 中的 CustID 列包含有值 1、2、3、4，OrderHeader 中的 CustID 列包含有值 1、1、2、3、NULL。执行 `1 IN (1, 1, 2, 3, NULL)`，返回 TRUE，而 `1 NOT IN (1, 1, 2, 3, NULL)` 返回 FALSE。2 和 3 与列表(1, 1, 2, 3, NULL)的比较与 1 相同。执行 `4 IN (1, 1, 2, 3, NULL)`，由于 4 不包含在列表中，但是列表中包含一个 NULL，因此 `4 IN (1, 1, 2, 3, NULL)` 返回 UNKNOWN。`4 NOT IN (1, 1, 2, 3, NULL)` 返回 NOT UNKNOWN，等价于 UNKNOWN。所以 Customers 中 CustID 列为 4 的行不会被包含在查询结果中。

同时，当含有 NULL 值时，数据库引擎为 NOT EXISTS 和 NOT IN 生成的执行计划并不相同。NOT EXISTS 的效率会更高一些，如图 6-2 所示。

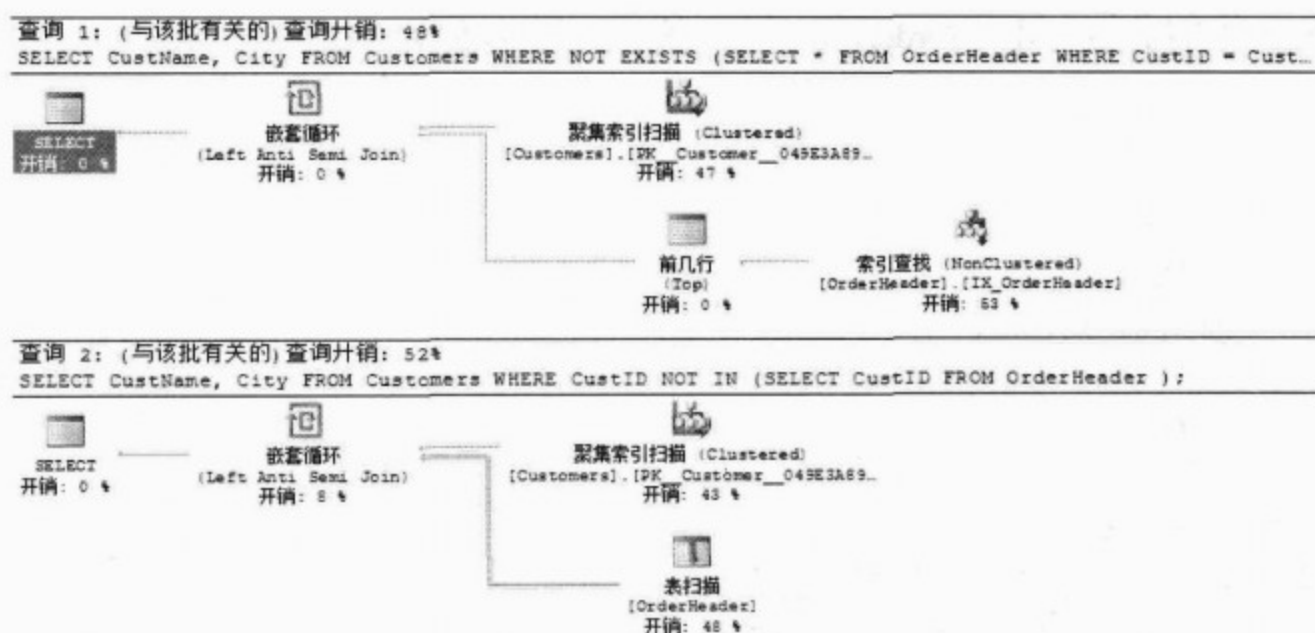


图 6-2 执行计划

为了避免在包含 NOT IN 的子查询中出现 NULL 值，应当使用 IS NOT NULL 进行过滤，参考下面的语句，它可以返回与 NOT EXISTS 相同的查询结果（见表 6-8）。

```
SELECT CustName, City
FROM Customers
WHERE CustID NOT IN
    (SELECT CustID
     FROM OrderHeader
     WHERE CustID IS NOT NULL);
```

6.3 使用含有比较运算符的子查询

子查询可以由一个比较运算符（=、<>、>、>=、<、!>、!< 或 <=）引入。与 IN 不同，使用比较运算符引入的子查询必须返回单个值而不是一个值列表；否则，将显示一条错误信息。

仍旧使用如表 6-4 和表 6-5 所示的两个表。例如，假设要查找 OrderHeader 中 Shanghai 客户的订单，可以使用以下语句：

```
SELECT OrderID, OrderDate, ShipTo
FROM OrderHeader
WHERE CustID = (SELECT CustID
                 FROM Customers
                 WHERE City = 'Shanghai');
```

分析如表 6-4 所示的 Customers 表，可以看出其中只有 1 个客户来自 Shanghai，所示上面这条语句可以被正常执行。如果希望查找来自 Beijing 客户的订单，执行以下语句则会引发错误，因为 Customers 中有 3 个客户来自 Beijing。

```
SELECT OrderID, OrderDate, ShipTo
FROM OrderHeader
WHERE CustID = (SELECT CustID
                 FROM Customers
                 WHERE City = Beijing);
```

错误信息如下：

消息 512，级别 16，状态 1，第 1 行
子查询返回的值不止一个。当子查询跟随在 =、!=、<、<=、>、>= 之后，或子查询用作表达式时，这种情况是不允许的。

对于这种多值情况，仍旧需要使用 IN 或=ANY 来完成，如：

```
-- 使用 IN
SELECT OrderID, OrderDate, ShipTo
FROM OrderHeader
WHERE CustID IN (SELECT CustID
                 FROM Customers
                 WHERE City = 'Beijing');

-- 使用=ANY
SELECT OrderID, OrderDate, ShipTo
FROM OrderHeader
WHERE CustID = ANY(SELECT CustID
                   FROM Customers
                   WHERE City = 'Beijing');
```

由于由比较运算符引入的子查询要返回单个值，因此通常在子查询中包含聚合函数。例如，仍旧使用如表 6-5 所示的 OrderHeader 中的数据，以下语句将找出 OrderHeader 中订单数量大于客户平均订单数量的 CustID。分析 OrderHeader 表可以看出，总订单数量为 4，订单客户为 3，平均订单数量为 1.33。这样只有 CustID 为 1 的客户超出了平均订单数量，查询结果如表 6-9 所示。

```
SELECT CustID, COUNT(CustID) AS OrderCnt
FROM OrderHeader
GROUP BY CustID
HAVING COUNT(CustID) > (SELECT COUNT(CustID)*1.0/COUNT(DISTINCT CustID)
                        FROM OrderHeader);
```

表 6-9

查询结果

CustID	OrderCnt
1	2

6.4 使用 ANY、SOME 或 ALL 关键字进行批量比较

可以使用 ALL 或 ANY 关键字修改引入子查询的比较运算符。SOME 是与 ANY 等效的 ISO 标准。ALL 要求 WHERE 表达式与子查询返回的每个值进行比较时都应满足比较条件，ANY 则要求 WHERE 表达式与子查询返回的值进行比较时至少有一个应满足比较条件。

以 > 比较运算符为例，>ALL 表示大于每一个值，也就是大于最大值。例如，>ALL (1, 2, 3) 等价于 “>1 AND >2 AND >3”，即大于 3。>ANY 表示至少大于一个值，即大于最小值。例如，>ANY (1, 2, 3) 等价于 “>1 OR >2 OR >3”，即大于 1。

假设 WHERE 表达式的值为 2，子查询的返回的值为 2 和 3，2=ALL (2, 3) 等价于 “2=2 AND 2=3”，计算结果为 FALSE，因为子查询的 3 不满足表达式的条件。2<>ALL (2, 3) 的计算结果同样为 FALSE，因为子查询的 2 不满足表达式的条件。2=ANY (2, 3) 等价于 “2=2 OR 2=3”，计算结果为 TRUE，因为其中至少子查询的 2 满足表达式条件。2<>ANY (2, 3) 的计算结果同样为 TRUE，因为其中至少子查询的 3 满足表达式条件。

表 6-10 总结了比较运算符与 ANY 或 ALL 关键字连用时的取值情况。

表 6-10 比较运算符与 ANY 或 ALL 关键字连用时的取值情况

比较运算符	关键字	对子查询结果的取值
>、>=、!<	ALL	最大值
	ANY	最小值
=	ANY	等于其中的任意值，相当于 IN
	ALL	等于所有值
<>	ALL	不等于所有值，相当于 NOT IN
	ANY	不等于其中任意值
<、<=、!>	ALL	最小值
	ANY	最大值

来看一个比较实际的应用。假设一个大学中有 3 门选修课程，学生根据需要至少选修 2 门，使用下面的语句首先创建示例数据，内容如表 6-11 所示。

```
CREATE TABLE Lessons
(StudentName char(20) NOT NULL,
 LessonNbr int NOT NULL,
 LessonStat char(4) NOT NULL
 CHECK (LessonStat IN ('DONE', 'WAIT')));
```

```

INSERT INTO Lessons
VALUES ('Ken', 1, 'DONE'),
      ('Ken', 2, 'WAIT'),
      ('Ken', 3, 'WAIT'),
      ('Nan', 1, 'WAIT'),
      ('Nan', 2, 'WAIT'),
      ('Tom', 1, 'DONE'),
      ('Tom', 2, 'DONE'),
      ('Tom', 3, 'DONE'),
      ('Bob', 1, 'DONE'),
      ('Bob', 2, 'WAIT');

```

表 6-11 Lessons 表中的内容

StudentName	LessonNbr	LessonStat
Ken	1	DONE
Ken	2	WAIT
Ken	3	WAIT
Nan	1	WAIT
Nan	2	WAIT
Tom	1	DONE
Tom	2	DONE
Tom	3	DONE
Bob	1	DONE
Bob	2	WAIT

现在希望找出课程 1 已经修完，但是其他课程没有修完的学生。分析表 6-11 可以看出，只有 Ken 和 Bob 符合这样的条件限定。下面语句中的 WHERE 子句首先判断 LessonNbr 为 1 并且 LessonStat 为 DONE，然后通过子查询根据当前行的学生姓名找出所有 LessonNbr 不为 1 并且 LessonStat 为 WAIT 的行。

```

SELECT StudentName
FROM Lessons AS L1
WHERE LessonNbr = 1
      AND LessonStat = 'DONE'
      AND 'WAIT' = ALL (SELECT LessonStat
                        FROM Lessons AS L2
                        WHERE LessonNbr <> 1
                        AND L1.StudentName = L2.StudentName);

```

假设希望找出课程 1 已经修完，但是其他课程没有修完，并且选修课程数量为 3 的学生。分析表 6-11 可以看出，现在只有 Ken 符合这样的条件限定。下面的语句在以上基础上又增加了一个含有子查询的筛选条件。

```

SELECT StudentName
FROM Lessons AS L1
WHERE LessonNbr = 1
      AND LessonStat = 'DONE'
      AND 'WAIT' = ALL (SELECT LessonStat
                        FROM Lessons AS L2
                        WHERE LessonNbr <> 1

```

```

        AND L1.StudentName = L2.StudentName)
AND 3 = (SELECT COUNT(LessonNbr)
        FROM Lessons AS L3
        WHERE L1.StudentName = L3.StudentName);

```

实际上，上面的语句也可以改写为 EXISTS 形式。请读者自行分析以下语句：

```

SELECT StudentName
FROM Lessons AS L1
WHERE LessonNbr = 1
    AND LessonStat = 'DONE'
    AND EXISTS (SELECT *
                FROM Lessons AS L2
                WHERE LessonNbr <> 1
                    AND L1.StudentName = L2.StudentName
                    AND LessonStat = 'WAIT')
AND EXISTS (SELECT *
            FROM Lessons AS L3
            WHERE L1.StudentName = L3.StudentName
            HAVING COUNT(LessonNbr) = 3);

```

6.5 使用多层嵌套子查询

子查询自身可以包括一个或多个子查询，一个语句中可以嵌套任意数量的子查询。查询的执行顺序是由里至外。

例如，下面的语句将查找作为销售人员的雇员的姓名。其中最里层的查询将返回销售人员的 ID。再上一层查询将用这些销售人员 ID 进行取值，并返回雇员的 ContactID 号。最后，外部查询将使用这些 ContactID 查找雇员的姓名。

```

USE AdventureWorks;
GO
SELECT LastName, FirstName, EmailAddress
FROM Person.Contact
WHERE ContactID IN (SELECT ContactID
                   FROM HumanResources.Employee
                   WHERE EmployeeID IN (SELECT SalesPersonID
                                       FROM Sales.SalesPerson));

```

6.6 子查询应遵循的规则

在前面几节中详细介绍了子查询的使用方法。此外，在编写子查询时还应当注意遵循以下几方面的规则。

通过比较运算符引入的子查询选择列表只能包括一个表达式或列名称。例如，以下语句将找出定价高于平均定价的所有产品的名称。其中的子查询只包含一个聚合函数表达式。

```

USE AdventureWorks;
GO
SELECT Name
FROM Production.Product
WHERE ListPrice > (SELECT AVG (ListPrice)
                  FROM Production.Product);

```


如果外部查询的 **WHERE** 子句包括列名称, 则必须与子查询选择列表中的列是联接兼容的。仍旧以上面的语句为例, **WHERE** 中的 **ListPrice** 与子查询中的 **AVG (ListPrice)** 数据类型一致, 具有可比性。

而 **ntext**、**text** 和 **image** 数据类型不能用在子查询的选择列表中。

由于必须返回单个值, 所以由未修改的比较运算符 (即后面未跟关键字 **ANY** 或 **ALL** 的运算符) 引入的子查询不能包含 **GROUP BY** 和 **HAVING** 子句。例如, 下面语句中的子查询按 **Color** 进行分组, 由于返回了多个值, 所以该查询无法执行。

```
SELECT Name
FROM Production.Product
WHERE ListPrice > (SELECT AVG (ListPrice)
                   FROM Production.Product
                   GROUP BY Color);
```

在比较运算符后加入 **ANY** 或 **ALL** 关键字后, 该查询可以正常执行, 如:

```
SELECT Name
FROM Production.Product
WHERE ListPrice > ALL(SELECT AVG (ListPrice)
                     FROM Production.Product
                     GROUP BY Color);
```

ORDER BY 子句不能用于子查询, 但是, 在指定了 **TOP** 时却可以。例如, 下面的语句由于子查询中包含了 **ORDER BY**, 而不能被执行。

```
SELECT Name
FROM Production.Product
WHERE ListPrice > ALL(SELECT AVG (ListPrice)
                     FROM Production.Product
                     GROUP BY Color
                     ORDER BY Color);
```

此外, 如果创建视图的语句包含子查询, 则无法通过该视图进行数据更新。





第7章 联接和 APPLY 运算符

在上一章中已经介绍了子查询，实际上，大多数子查询问题都可以使用联接来处理。自然，使用联接能够完成的任务，大部分也可以用子查询来完成。将业务数据分别放置在不同表中后，就方便了数据的维护与存储，当我们再希望看到完整的业务数据时，则就经常需要使用联接查询。例如，Customer 表存放着顾客的姓名、顾客编号、邮寄地址、联系电话等信息，Orders 表中存放着订单所对应的顾客编号、商品名称、销售金额等，当将这些商品邮寄给客户时，需要在包裹的封面上打印客户信息和商品信息，这时就需要使用联接查询，将 Customer 表的客户信息与 Orders 表中相应的商品信息组合在一起。

本章将对联接的主要类型、联接算法等进行详细介绍，并提供一些具体的实际应用示例。

7.1 联接的基本知识

通过联接，可以根据表间的逻辑关系，从两个或多个表中检索数据。联接查询是关系型数据库的一个主要特点，同时也是区别于其他类型数据库管理系统的一个主要标志。

将不同类型的数据存放在不同表中，可以防止产生冗余数据。例如，假设部门表中包含部门编号和部门名称两列，而雇员表中包含雇员编号、雇员隶属部门编号和雇员姓名等列。假设要获取部门名称和部门中雇员的姓名，则可以根据两个表中的都有的部门编号列，使用联接的方式检索数据；否则，则只能在雇员表中也同样保存有部门名称列，这样就会导致产生冗余数据。

7.1.1 联接的语法格式

联接条件通常在 FROM 子句中指定，但是像内部联接也可以在 WHERE 子句中指定。此外，在建立联接时，还可以使用 WHERE 和 HAVING 子句指定搜索条件，以进一步筛选根据联接条件选择的行。

1. 使用 FROM 子句联接

首先来看一下使用 FROM 子句联接的基本语法格式，如下所示：

```
FROM first_table join_type second_table [ON (join_condition)]
```

`join_type` 指定要执行的联接类型, 根据查询时数据的检索方式, 可以分为交叉联接、内部联接和外部联接。`join_condition` 用于指定联接条件。

例如, 假设存在 `Departs` 和 `Employees` 两个表, 内容如表 7-1 和表 7-2 所示。下面语句中的 `FROM` 子句指定将 `Departs` 表和 `Employees` 表进行联接, 联接条件为 `Departs.DepID = Employees.DepID`。

```
SELECT Departs.DepName, Employees.EmpID, Employees.EmpName
FROM Departs JOIN Employees
ON (Departs.DepID = Employees.DepID)
```

表 7-1 Departs 表中的内容

DepID	DepName
1	销售部门 1
2	销售部门 2

表 7-2 Employees 表中的内容

DepID	EmpID	EmpName
1	1	张三
1	2	李四

2. 使用 WHERE 子句联接

下列查询包含与上面语句相同的联接条件, 但是, 该联接条件是在 `WHERE` 子句中指定的。

```
SELECT Departs.DepName, Employees.EmpID, Employees.EmpName
FROM Departs, Employees
WHERE Departs.DepID = Employees.DepID
```

上面的语句实际上是一种 ANSI SQL:1989 规定的语法格式。联接表之间使用逗号分隔, 联接条件使用 `WHERE` 子句指定。

ANSI SQL:1989 仅支持交叉和内部联接, 不支持外部联接。从 ANSI SQL:1992 加入了对外部联接的支持, 语法格式也转变为使用 `JOIN` 关键字联接表, 使用 `ON` 关键字指定联接条件。虽然旧标准在 SQL Server 中仍旧被支持, 但是为了语句的规范性, 建议读者使用新标准格式进行语句书写。

7.1.2 联接所使用的逻辑处理阶段

读者对第 5 章介绍的查询的逻辑处理步骤是否还有印象? 这些逻辑步骤对于理解表之间的联接处理和编写正确的联接查询语句, 非常有帮助。交叉联接、内部联接和外部联接所使用的逻辑处理阶段是不同的。交叉联接只使用第 1 个阶段, 即笛卡尔乘积。内部联接使用第 1 个和第 2 个阶段, 即笛卡尔乘积和 `ON` 联接条件筛选器。外部联接应用前 3 个阶段, 即笛卡尔乘积、`ON` 联接条件筛选器和添加外部行。

7.1.1 小节使用的 `FROM` 联接查询是一个内部联接, 它首先执行对 `Departs` 和 `Employees` 执行

笛卡尔乘积，得到如表 7-3 所示的虚拟表内容。然后再对该虚拟表执行 ON 筛选条件 `Departs.DepID = Employees.DepID`，由表 7-3 可以看出，只有第 1 行和第 3 行符合该条件，该步骤得到的虚拟表如表 7-4 所示。如果是外部联接，还会根据外部联接（即左外部联接、右外部联接和完全外部联接）将保留表中的内容添加回虚拟表中。最后一步是根据 SELECT 列表，将指定列的内容返回给查询调用者，如表 7-5 所示。

表 7-3 执行笛卡尔乘积得到的虚拟表

DepID	DepName	DepID	EmpID	EmpName
1	销售部门 1	1	1	张三
2	销售部门 2	1	1	张三
1	销售部门 1	1	2	李四
2	销售部门 2	1	2	李四

表 7-4 执行 ON 筛选条件后得到的虚拟表

DepID	DepName	DepID	EmpID	EmpName
1	销售部门 1	1	1	张三
1	销售部门 1	1	2	李四

表 7-5 查询语句返回的结果集

DepName	EmpID	EmpName
销售部门 1	1	张三
销售部门 1	2	李四

上面我们简单回顾了联接处理的各个阶段，对交叉联接、内部联接和外部联接有了大致了解。除了这种两个表之间的基本联接，还可以在 FROM 子句中包含多个联接。但是，无论 FROM 子句中包含几个联接，每次只会对其中的两个表根据联接类型来选择所需要执行的阶段。联接所产生的虚拟表再与 FROM 子句中的下一个表进行联接，直至处理完所有联接。

7.1.3 列名限定和选择列表的使用

在单个查询中引用多个表时，所有列引用都必须是明确的，重复的列名都必须用表名加以限定。例如，上例中 `Departs` 和 `Employees` 表由于都含 `DepID` 的列，在查询中使用该列时必须加表名进行限定。

即使列名在表之间不存在重复，将所有列都用它们的表名加以限定，会提高查询的可读性。如果使用了表的别名，将会进一步提高可读性，尤其是当表名自身必须用数据库名和所有者名加以限时。例如，下面的语句为表分配了别名，并且用表的别名对列加以限定，从而提高了语句的可读性：

```
SELECT D.DepName, E.EmpID, E.EmpName
FROM Departs AS D JOIN Employees AS E
ON (D.DepID = E.DepID)
```


联接选择列表可以引用联接表中的所有列或任意一部份列,不必包含联接中每个表的列。但是,不要忘记,在进行联接时是对两个表的所有列进行联接,而不仅仅是 SELECT 列表中的列。因此,即使某个列不包含在 SELECT 列表中,仍旧可以引用该列。例如,在下面的语句中虽然 E.EmpID 列没有包含在 SELECT 列表中,但是仍旧通过该列进行数据筛选。

```
SELECT D.DepName, E.EmpName
FROM Departs AS D JOIN Employees AS E
ON (D.DepID = E.DepID)
WHERE E.EmpID = 1
```

7.1.4 联接条件设定

虽然联接条件通常使用相等比较(=),但也可以像指定其他谓词一样指定其他比较运算符或关系运算符。

联接条件中用到的列不必具有相同的名称或相同的数据类型。但如果数据类型不相同,则必须兼容,或者是可隐式转换的类型。如果数据类型不能进行隐式转换,则联接条件必须使用 CAST 函数显式转换数据类型。

7.2 交叉联接

上文已经多次对交叉联接(CROSS JOIN)进行了介绍。交叉联接是联接查询的第1个阶段,它对两个表进行笛卡尔乘积。即将第1个表中的每一行与第2个表的所有行进行联接,因此,生成的结果集的大小等于第1个表的行数乘以第2个表的行数。图7-1演示了T1表与T2表交叉联接的结果。

```
SELECT *
FROM T1 CROSS JOIN T2;
```

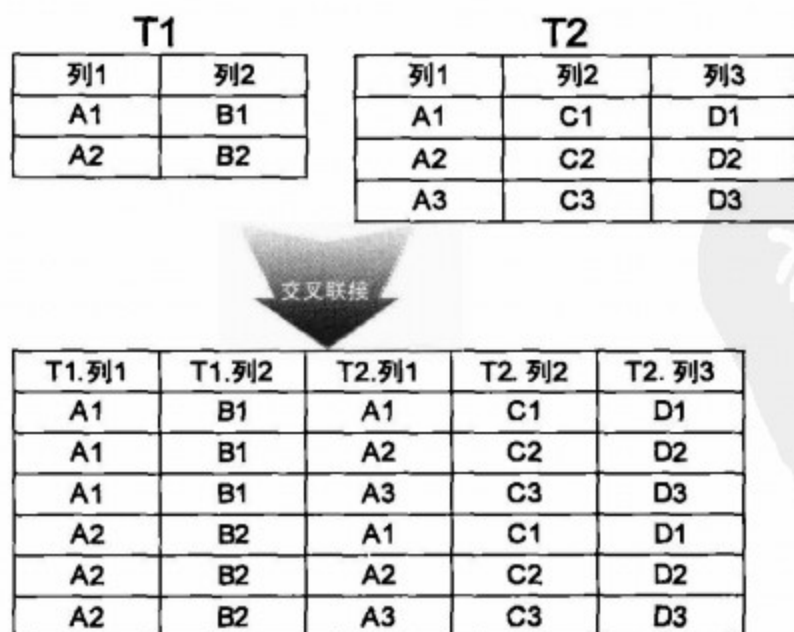


图 7-1 T1 与 T2 的交叉联接结果

7.2.1 交叉联接的语法格式

要实现交叉联接，应当使用 **CROSS JOIN** 关键字联接两个表，也可以按照 ANSI SQL:1989 的规定使用逗号来联接。例如，下面的语句将 **table1** 表和 **table2** 表进行交叉联接。

```
SELECT *  
FROM table1 CROSS JOIN table2;
```

下面是使用逗号进行联接。

```
SELECT *  
FROM table1, table2;
```

也可以使用 **WHERE** 子句为交叉联接指定联接条件。这种情况下，交叉联接的作用则等同于内部联接。如：

```
SELECT *  
FROM table1 CROSS JOIN table2  
WHERE table1.column1 = table2.column1;
```

7.2.2 使用交叉联接查询全部数据

作为联接查询的第 1 个阶段，交叉联接似乎在理论上的意义更大一些。出于这方面的原因，交叉联接遭到了很多置疑，许多人认为根本就不会使用到它们。并且，在多数情况下，交叉联接消耗的资源太多，从而无法高效使用。但是，像 SQL 中的其他任何工具一样，如果能够正确使用它们，有时也会带来意想不到的收获。

首先执行下面的语句创建两个示例表：**Employees** 和 **Orders**，表中内容分别如表 7-6 和表 7-7 所示。

```
CREATE TABLE Employees  
(EmpID int NOT NULL,  
 EmpName char(10) NOT NULL);  
CREATE TABLE Orders  
( EmpID int NOT NULL,  
  SeasonNbr char(10) NOT NULL,  
  Sales money DEFAULT 0.00 NOT NULL);  
  
INSERT INTO Employees  
VALUES (1, 'Grace'),  
       (2, 'Ken'),  
       (3, 'Tom');  
  
INSERT INTO Orders  
VALUES (1, 'Season 1', 100.00),  
       (1, 'Season 2', 100.00),  
       (2, 'Season 3', 120.00),  
       (2, 'Season 4', 130.00);
```

表 7-6 Employees 表中的内容

EmpID	EmpName
1	Grace
2	Ken
3	Tom

表 7-7 Orders 表中的内容

EmpID	SeasonNbr	Sales
1	Season 1	100.00
1	Season 2	100.00
2	Season 3	120.00
2	Season 4	130.00

Employees 中存放着雇员信息，Orders 存放着雇员的季度销售数据。现在假设要返回雇员的每季度的销售数据。这个问题看起来似乎非常简单，但是请考虑一下，如果从 Employees 表到 Orders 表进行了仅执行像下面这样的标准内部联接 (INNER JOIN)，则只会获得雇员有销售数据的季度，如表 7-8 所示。

```
SELECT Employees.EmpName, Orders.SeasonNbr, Orders.Sales
FROM Employees INNER JOIN Orders
ON Employees.EmpID = Orders.EmpID;
```

表 7-8 执行内部联接得到的查询结果

EmpName	SeasonNbr	Sales
Grace	Season 1	100.00
Grace	Season 2	100.00
Ken	Season 3	120.00
Ken	Season 4	130.00

对于雇员没有销售数据的季度，并不会得到 0 值，如 Grace 雇员的 Season 3 和 Season 4 由于无销售数据，所以并未包含在查询结果中。如果想为每个雇员都绘制一个图，以显示每个季度和该季度销售额，则可能希望此图包括销售额为 0 的季度，以便直观标识出这些季度。

可以使用交叉联接的方式解决此问题。创建一个名为 Seasons 的辅助表，表中存放着 4 个季度名称。

```
CREATE TABLE Seasons (SeasonNbr char(10));
INSERT INTO Seasons VALUES ('Season 1'), ('Season 2'), ('Season 3'), ('Season 4');
```

对 Employees 表和 Seasons 表进行交叉联接，会得到如表 7-9 所示的查询结果。

```
SELECT *
FROM Employees CROSS JOIN Seasons;
```

表 7-9 Employees 表和 Seasons 表进行交叉联接后的结果

EmpID	EmpName	SeasonNbr
1	Grace	Season 1
1	Grace	Season 2
1	Grace	Season 3
1	Grace	Season 4
2	Ken	Season 1
2	Ken	Season 2
2	Ken	Season 3
2	Ken	Season 4
3	Tom	Season 1
3	Tom	Season 2
3	Tom	Season 3
3	Tom	Season 4

可以看到，交叉联接后的结果中每个雇员都具有了 4 个季度。然后再将该结果与 Orders 表进行左外联接，可以得到最终要求的数据格式。下面是完整查询语句，查询结果如表 7-10 所示。

```

SELECT Employees.EmpName, Seasons.SeasonNbr,
       CASE
         WHEN Orders.Sales IS NULL THEN 0
         ELSE Orders.Sales
       END AS SeasonSales
FROM Employees CROSS JOIN Seasons -- 交叉联接
LEFT OUTER JOIN Orders -- 再将交叉联接结果与 Orders 进行左外联接
ON Employees.EmpID = Orders.EmpID
AND Seasons.SeasonNbr = Orders.SeasonNbr;
    
```

表 7-10 为雇员返回的每季度销售情况

EmpName	SeasonNbr	SeasonSales
Grace	Season 1	100.00
Grace	Season 2	100.00
Grace	Season 3	0.00
Grace	Season 4	0.00
Ken	Season 1	0.00
Ken	Season 2	0.00
Ken	Season 3	120.00
Ken	Season 4	130.00
Tom	Season 1	0.00
Tom	Season 2	0.00
Tom	Season 3	0.00
Tom	Season 4	0.00

7.2.3 使用交叉联接优化查询性能

在子查询一章中，已经讨论了当在 SELECT 列列表中使用子查询时，比较耗费资源，因为对于所引用表中的每一行理论上都要执行一次子查询。使用交叉联接则可以解决这方面的问题。

仍旧以上面创建的 Orders 表（见表 7-7）为例。假设现在要计算雇员每季度销售额占总销售额的比例，以及与平均销售额的差。按照此语义，最直接的方法就是在 SELECT 的列列表中使用子查询来返回总销售额和平均销售额，参考下面的语句：

```
SELECT EmpID, SeasonNbr, Sales,
       CAST(Sales / (SELECT SUM(Sales) FROM Orders) * 100 AS decimal(5,2)) AS Per,
       Sales - (SELECT AVG(Sales) FROM Orders) AS Diff
FROM Orders;
```

返回结果如表 7-11 所示。该查询生成的执行计划如图 7-2 所示。

表 7-11

查询结果

EmpID	SeasonNbr	Sales	Per	Diff
1	Season 1	100.00	22.22	-12.50
1	Season 2	100.00	22.22	-12.50
2	Season 3	120.00	26.66	7.50
2	Season 4	130.00	27.88	17.50

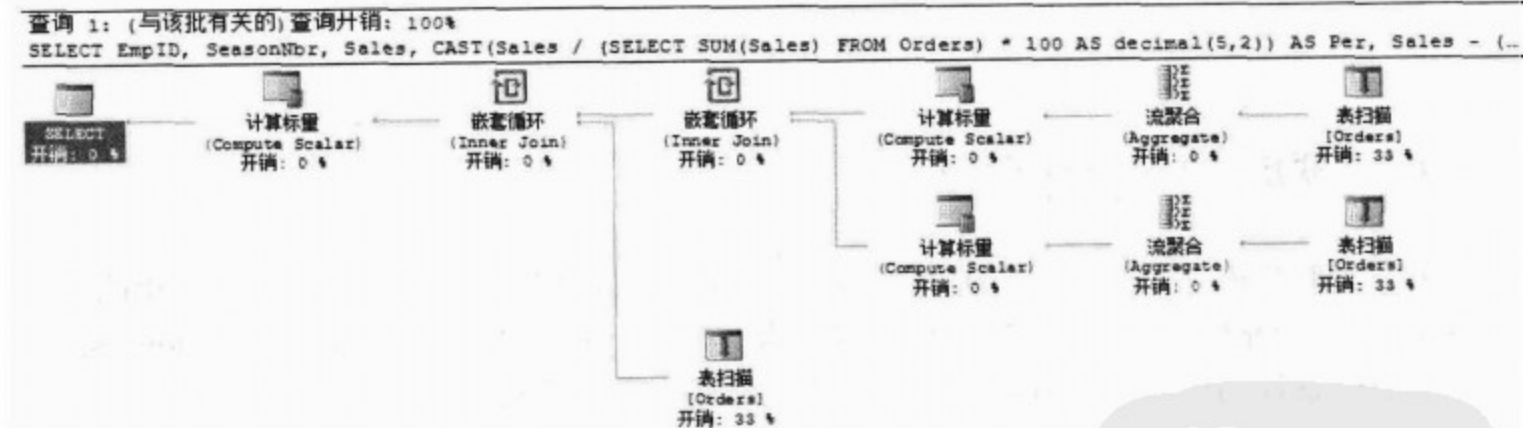


图 7-2 使用子查询时的执行计划

由图 7-2 可以看出，表扫描被执行了 3 次，分别用于计算总销售额、平均销售额和生成结果行。这个查询可以使用交叉联接进行优化。可以在一个查询中计算所有聚合，此时只会进行一次表扫描。然后将该结果与 Orders 表进行交叉联接，就会得到一个既包括基表列又包括聚合列的结果。参考下面的语句，交叉联接后的结果如表 7-12 所示。

```
SELECT *
FROM Orders
CROSS JOIN (SELECT SUM(Sales) AS SumSales,
                  AVG(Sales) AS AvgSales
```

FROM Orders) AS Oth;

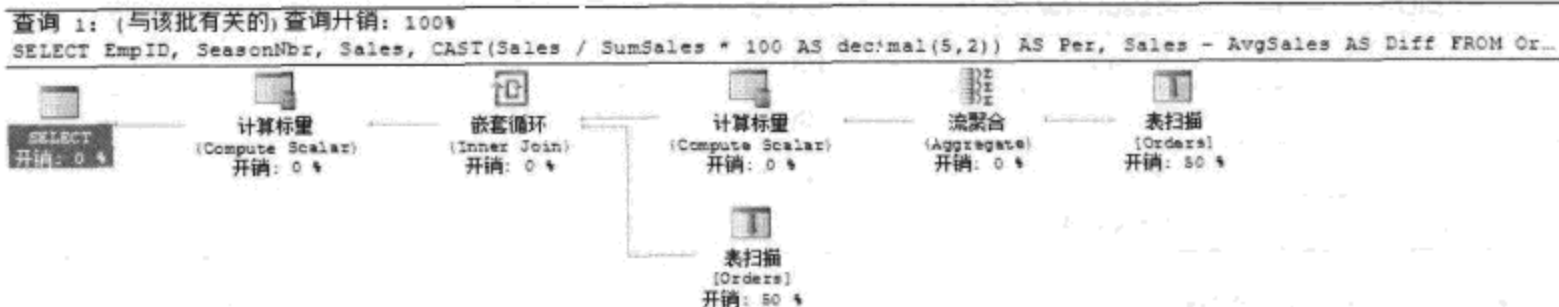
表 7-12

交叉联接后的结果

EmpID	SeasonNbr	Sales	SumSales	AvgSales
1	Season 1	100.00	450.00	112.50
1	Season 2	100.00	450.00	112.50
2	Season 3	120.00	450.00	112.50
2	Season 4	130.00	450.00	112.50

由表 7-12 可以看出, 交叉联接后再编写与总销售额、平均销售额的比较语句是十分简单的。下面是完整的查询语句, 该语句的执行计划如图 7-3 所示。从中可以看出, 表扫描减少为 2 次。

```
SELECT EmpID, SeasonNbr, Sales,
       CAST(Sales / SumSales * 100 AS decimal(5,2)) AS Per,
       Sales - AvgSales AS Diff
FROM Orders
     CROSS JOIN (SELECT SUM(Sales) AS SumSales,
                       AVG(Sales) AS AvgSales
                  FROM Orders) AS Oth;
```



7.2.4 为交叉联接添加 WHERE 子句

如果为交叉联接添加一个 WHERE 子句, 则交叉联接的作用将同内部联接一样。仍旧以前面创建的 Employees 和 Orders 表 (见表 7-6 和表 7-7) 为例, 下面的语句指定仅将符合 Employees.EmpID = Orders.EmpID 条件的行才放入到结果集中。查询结果见前面表 7-8。

```
SELECT Employees.EmpName, Orders.SeasonNbr, Orders.Sales
FROM Employees
     CROSS JOIN Orders
WHERE Employees.EmpID = Orders.EmpID;
```

该语句等同于下面的内部联接, 并且会生成相同的执行计划。

```
SELECT Employees.EmpName, Orders.SeasonNbr, Orders.Sales
FROM Employees
     INNER JOIN Orders
       ON Employees.EmpID = Orders.EmpID;
```

虽然交叉联接支持使用 WHERE 子句筛选行, 由于笛卡尔乘积占用的资源可能会很多, 在并不

是真正需要笛卡尔乘积的情况下，则应当谨慎地使用 CROSS JOIN。使用 INNER JOIN 会获得同样的结果，效率会更高一些。如果需要为所有的可能性都返回数据（就像前面介绍的为雇员生成所有季度数据），则笛卡尔乘积可能会非常有帮助。

7.3 内部联接

内部联接（INNER JOIN）属于典型的联接运算，使用=、<>等比较运算符进行条件联接。内部联接使用联接查询的前两个阶段，即笛卡尔乘积和 ON 筛选器。内部联接仅获取两个表中与联接条件匹配的记录，图 7-4 演示了 T1 表与 T2 表按“T1.列1=T2.列1”条件进行内部联接后的结果。它得到的是两个表的交集，两个表不匹配的行会被消除掉。

```
SELECT *
FROM T1 INNER JOIN T2
ON T1.列1 = T2.列1;
```

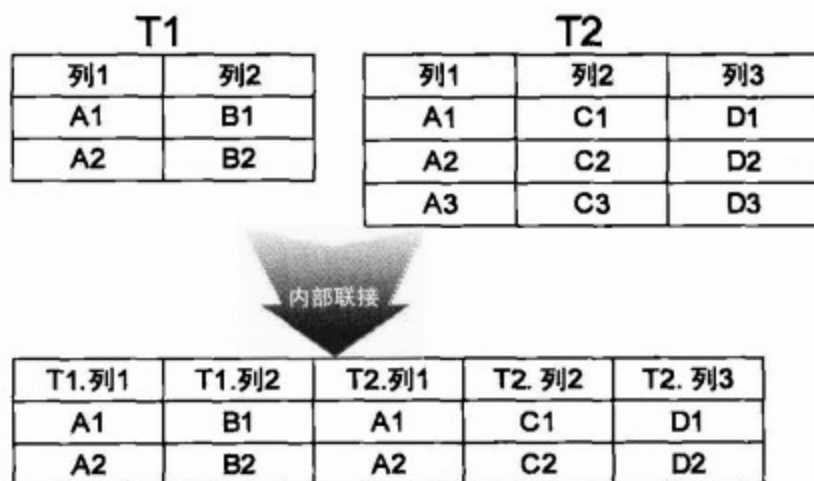


图 7-4 按“T1.列1=T2.列1”条件进行内部联接后的结果

7.3.1 内部联接的语法格式

要实现内部联接，应当使用 INNER JOIN 关键字联接两个表，并使用 ON 关键字指定联接条件。例如，下面的语句将 table1 表和 table2 表进行内部联接，联接条件为：table1.column1 = table2.column1。

```
SELECT *
FROM table1 INNER JOIN table2
ON table1.column1 = table2.column1;
```

INNER 关键字也可以省略，省略后会默认为内部联接。如：

```
SELECT *
FROM table1 JOIN table2
ON table1.column1 = table2.column1;
```

可以为内部联接指定 WHERE 子句。当在内部联接中既包含 ON 子句又包含 WHERE 子句时，

应当将两表之间的联接条件写在 ON 子句中，而对表中数据的筛选写在 WHERE 子句中。例如，下面的语句图 7-4 所示的 T1 表和 T2 表按“T1.列 1 = T2.列 1”条件进行内部联接，并按筛选出 T1 中列 1 为 A1 的行。

```
SELECT *
FROM T1 INNER JOIN T2
  ON T1.列 1 = T2.列 1
WHERE T1.列 1 = 'A1';
```

逻辑上，数据引擎会对笛卡尔乘积首先使用 ON 进行筛选，然后再使用 WHERE 进行筛选。在实际执行时，为提高数据处理速度，在不影响结果正确性的前提下，查询优化器可能会选择 WHERE 先消除一些无效行，然后再进行内部联接。例如，上面的语句可以改写为如下的形式。可以看出，上面 ON 子句和 WHERE 子句的筛选条件现在位于同一水平位置，优化器完全有理由根据性能需要选择优先执行的筛选条件。

```
SELECT *
FROM T1 INNER JOIN T2
  ON T1.列 1 = T2.列 1 AND T1.列 1 = 'A1';
```

而下面则是上面内部联接语句的 ANSI SQL:1989 书写方式，筛选条件同样位于同一水平位置。

```
SELECT *
FROM T1, T2
WHERE T1.c1 = T2.c1 AND T1.c1 = 'A1';
```

可以说，在内部联接中，在 ON 子句或 WHERE 子句中指定逻辑表达式没有任何区别。但是，在外部联结中由于存在添加外部行阶段，二存在一些差别。详细信息参考 6.2.4 和 6.2.5 小节的介绍。

根据联接条件中比较运算符的不同，可以将内部联接区分为等值联接和不等值联接。使用“=”符号进行表联接的内部联接称为等值联接，使用“<”、“>”、“>=”等符号进行表联接的内部联接称为不等值联接。

7.3.2 使用等值进行内部联接

假设某软件公司现在需要开发一个项目，该项目要求开发人员必须同时掌握 SQL Server、C# 和 XML 这 3 项技术。公司的雇员表 SoftEmployees 中存储着每个雇员所掌握的编程技术，SoftSkills 存储着本次要求必须同时掌握的 3 项技术。使用下面的语句创建示例表和所需要的数据，SoftEmployees 和 SoftSkills 的内容分别如表 7-13 和表 7-14 所示。

```
CREATE TABLE SoftEmployees
  (EmpName char(10) NOT NULL,
   SkillName char(20) NOT NULL);
CREATE TABLE SoftSkills
  (SkillName char(20) NOT NULL);

INSERT INTO SoftEmployees
VALUES ('Jones', 'SQL Server'),
      ('Jones', 'C#'),
```



```

('Jones', 'XML'),
('Grace', 'VB'),
('Grace', 'C#'),
('Eddie', 'VB'),
('Eddie', 'J#'),
('Celko', 'SQL Server'),
('Celko', 'C#'),
('Celko', 'XML'),
('Celko', 'J#');
INSERT INTO SoftSkills
VALUES ('SQL Server'),
('C#'),
('XML');

```

表 7-13 SoftEmployees 表中的内容

EmpName	SkillName
Jones	SQL Server
Jones	C#
Jones	XML
Grace	VB
Grace	C#
Eddie	VB
Eddie	J#
Celko	SQL Server
Celko	C#
Celko	XML
Celko	J#

表 7-14 SoftSkills 表中的内容

SkillName
SQL Server
C#
XML

分析 SoftEmployees 表可以看出, 只有 Jones 和 Celko 掌握有所要求的 3 项技术, 并且 Celko 还额外掌握有 J# 技术。我们知道, 在等值内部联接的情况下, 得到的是两个表的交集。也就是说, 将 SoftEmployees 和 SoftSkills 按 “SoftEmployees.SkillName = SoftSkills.SkillName” 条件联接, 会得到 SoftEmployees 中所有掌握 SQL Server、C#、XML 任一技术的雇员清单, 如表 7-15 所示。

```

SELECT *
FROM SoftEmployees AS E1
INNER JOIN SoftSkills AS S1
ON E1.SkillName = S1.SkillName;

```

表 7-15 SoftEmployees 和 SoftSkills 联接后的结果

E1.EmpName	E1.SkillName	S1.SkillName
Jones	SQL Server	SQL Server
Jones	C#	C#
Jones	XML	XML
Grace	C#	C#
Celko	SQL Server	SQL Server
Celko	C#	C#
Celko	XML	XML

从中可以看出, Grace 也被包含在了联接结果中,但是她仅掌握 C# 技术。因此,我们还需要从上面的结果中筛选出能够掌握 3 项技术的雇员。下面是完整的查询语句,它按 EmpName 列分类汇总,计算出同时掌握 3 项技术的雇员。返回结果为 Jones 和 Celko。

```
SELECT EmpName
FROM SoftEmployees AS E1
  INNER JOIN SoftSkills AS S1
    ON E1.SkillName = S1.SkillName
GROUP BY E1.EmpName
HAVING COUNT(E1.EmpName) = 3;
```

为了使语句更加通用一些,可以用子查询代替 3。参考下面的语句:

```
SELECT EmpName
FROM SoftEmployees AS E1
  INNER JOIN SoftSkills AS S1
    ON E1.SkillName = S1.SkillName
GROUP BY E1.EmpName
HAVING COUNT(E1.EmpName) = (SELECT COUNT(*) FROM SoftSkills);
```

7.3.3 使用不等值进行内部联接

在上面将 SoftEmployees 和 SoftSkills 按 “E1.SkillName = S1.SkillName” 条件进行等值联接后,得到了掌握 SQL Server、C#、XML 任一技术的雇员清单(见表 7-15)。下面是将 SoftEmployees 和 SoftSkills 按不等值条件联接的语句,查询结果如表 7-16 所示。

```
SELECT *
FROM SoftEmployees AS E1
  INNER JOIN SoftSkills AS S1
    ON E1.SkillName <> S1.SkillName
```

表 7-16 SoftEmployees 和 SoftSkills 不等值联接后的查询结果

E1.EmpName	E1.SkillName	S1.SkillName
Jones	C#	SQL Server
Jones	XML	SQL Server
Grace	VB	SQL Server

续表

E1.EmpName	E1.SkillName	S1.SkillName
Grace	C#	SQL Server
Eddie	VB	SQL Server
Eddie	J#	SQL Server
Celko	C#	SQL Server
Celko	XML	SQL Server
Celko	J#	SQL Server
Jones	SQLServer	C#
Jones	XML	C#
Grace	VB	C#
Eddie	VB	C#
Eddie	J#	C#
Celko	SQLServer	C#
Celko	XML	C#
Celko	J#	C#
Jones	SQLServer	XML
Jones	C#	XML
Grace	VB	XML
Grace	C#	XML
Eddie	VB	XML
Eddie	J#	XML
Celko	SQLServer	XML
Celko	C#	XML
Celko	J#	XML

分析以上数据,它们是没有掌握 SQL Server 的雇员? 是没有掌握 C#的雇员? 还是没有掌握 XML 的雇员? 都不是。这里面既包含了像 Eddie 这样对 SQL Server、C#和 XML 都没有掌握的雇员,又包含了像 Jones 和 Celko 这样全部掌握这 3 项技术的雇员。可以看出,这组数据没有任何实际意义。

实际上,非等值联接查询通常是需要同其他联接查询结合使用,尤其是同等值联接查询结合。例如,假设我们已经知道 Jones 掌握了 SQL Server、C#和 XML 这 3 项技术,现在需要从 SoftEmployees 找出 Jones 之外能够掌握这 3 项技术的雇员,下面的语句就是一个通过 AND 关键字将等值和不等值条件组合在一起的查询。它的返回结果为 Celko。

```

SELECT EmpName
FROM SoftEmployees AS E1
  INNER JOIN SoftSkills AS S1
    ON E1.SkillName = S1.SkillName AND E1.EmpName <> 'Jones'
GROUP BY E1.EmpName
HAVING COUNT(E1.EmpName) = (SELECT COUNT(*) FROM SoftSkills);

```

也可以将上面的不等值条件放置在 WHERE 子句中,对于内部联接而言,这对执行结果没有任何影响。我们已经多次提到,这是因为内部联接不包含添加外部阶段。参考下面的语句:

```
SELECT EmpName
FROM SoftEmployees AS E1
  INNER JOIN SoftSkills AS S1
    ON E1.SkillName = S1.SkillName
WHERE E1.EmpName <> 'Jones'
GROUP BY E1.EmpName
HAVING COUNT(E1.EmpName) = (SELECT COUNT(*) FROM SoftSkills);
```

不等值联接也可以用于其他联接类型中,一个比较有实用价值的示例请参考 7.5.2 小节的介绍。

7.4 外部联接

只有在两个表中都至少有一个行符合联接条件时,内部联接才返回行。内部联接消除了与另一个表中的行不匹配的行。而外部联接会返回 FROM 子句中提到的至少一个表或视图中的全部行,然后根据指定的联接条件返回另外一个表中的匹配行。被保留全部行的表,称之为保留表。

外部联接包含联接查询的 3 个阶段,即执行笛卡尔乘积、ON 联接条件筛选器和添加外部行。添加外部行阶段添加的是保留表中与联接条件不匹配的所有行。

根据保留表位于联接关键字的位置(左边表或右边表)不同,可以将外部联接分为左外部联接、右外部联接和完全外部联接。左外部联接将返回联接关键字左边表的全部行,并返回右边表中与联接条件匹配的行;右外联接则反之;完全外部联接则包括联接关键字两侧表中所有与联接条件匹配的行和不匹配行。

7.4.1 外部联接的语法格式

由于外部联接具有 3 种不同的联接方式,它们所使用的联接关键字也不相同。下面是 ANSI:SQL-92 中规定外部联接使用的关键字:

- 左外部联接,使用 LEFT OUTER JOIN 或 LEFT JOIN。
- 右外部联,使用 RIGHT OUTER JOIN 或 RIGHT JOIN。
- 完全外部联接,使用 FULL OUTER JOIN 或 FULL JOIN。

除了使用上述关键字外,还应当使用 ON 关键字指定联接条件。例如,下面的语句将 table1 表和 table2 表进行左外部联接,联接条件为: table1.column1 = table2.column1。

```
SELECT *
FROM table1 LEFT OUTER JOIN table2
  ON table1.column1 = table2.column1;
```

OUTER 关键字也可以省略,简写为如下形式:


```
SELECT *
FROM table1 LEFT JOIN table2
ON table1.column1 = table2.column1;
```

与内部联接相同，也可以为外部联接指定 WHERE 子句。当在外部联接中既包含 ON 子句又包含 WHERE 子句时，必须将两表之间的联接条件写在 ON 子句中，对表中数据的筛选写在 WHERE 子句中。前面我们介绍了，在内部联接中允许将 WHERE 子句中的筛选条件通过 AND 关键字一并写在 ON 子句中，这并不影响执行结果。但是，在外部联接中不允许这样做，因为外部联接存在添加外部行阶段。ON 子句是联接查询的第 2 个阶段，它在添加外部行阶段之前执行，保留表中被 ON 子句筛选掉的行，在添加外部行时会被添加回来。而 WHERE 子句是在添加外部行阶段之后进行，筛选掉的行不会再被添加回来。

例如，下面是内部联接的两行语句，它们得到的查询结果会完全相同。

```
SELECT *
FROM table1 INNER JOIN table2
ON table1.column1 = table2.column1
WHERE table1.column1 > 3;

SELECT *
FROM table1 INNER JOIN table2
ON table1.column1 = table2.column1 AND table1.column1 > 3;
```

而对于外部联接，虽然也是通过 AND 关键字合并了 ON 和 WHERE 条件，但是下面两行语句的执行结果并不一定相同。有关外部行添加的详细信息，请参考 6.2.4 和 6.2.5 小节的介绍。

```
SELECT *
FROM table1 LEFT OUTER JOIN table2
ON table1.column1 = table2.column1
WHERE table1.column1 > 3;

SELECT *
FROM table1 LEFT OUTER JOIN table2
ON table1.column1 = table2.column1 AND table1.column1 > 3;
```

7.4.2 使用左外部联接保留左表全部行

左外部联接返回联接条件左边表中的全部行，并从联接条件右边的表中检索出与联接条件匹配的记录。图 7-5 演示了 T1 表与 T2 表按“T1.列 1=T2.列 1”条件进行左外部联接后的结果。

```
SELECT *
FROM T1 LEFT OUTER JOIN T2
ON T1.列 1 = T2.列 1;
```

由图 7-5 可以看出，T1 表的全部行被保留。查询结果中的前两行是 T1 和 T2 中符合联接条件的行，第 3 行则是 T1 中不符合联接条件的行，该行是在添加外部行阶段被添加回来的。由于该行在 T2 中没有匹配行，所以行中 T2.列 1、T2.列 2 和 T2.列 3 的值均为 NULL。

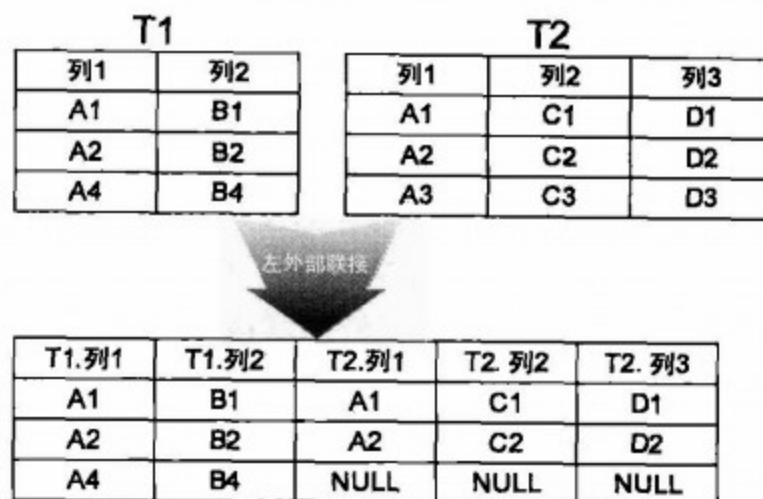


图 7-5 左外部联接后的结果

下面来看一个比较实际的问题。首先执行下面的语句创建两个示例表：Employees 和 Orders，表中内容分别如表 7-17 和表 7-18 所示。

```
CREATE TABLE Employees
(EmpID int NOT NULL,
 EmpName char(10) NOT NULL);
CREATE TABLE Orders
(EmpID int NOT NULL,
 SeasonNbr char(10) NOT NULL,
 Sales money DEFAULT 0.00 NOT NULL);

INSERT INTO Employees
VALUES (1, 'Grace'),
       (2, 'Ken'),
       (3, 'Tom');
INSERT INTO Orders
VALUES (1, 'Season 1', 100.00),
       (1, 'Season 2', 100.00),
       (1, 'Season 3', 120.00),
       (1, 'Season 4', 130.00),
       (2, 'Season 1', 200.00),
       (2, 'Season 2', 300.00),
       (2, 'Season 3', 150.00);
```

表 7-17

Employees 表中的内容

EmpID	EmpName
1	Grace
2	Ken
3	Tom

表 7-18

Orders 表中的内容

EmpID	SeasonNbr	Sales
1	Season 1	100.00
1	Season 2	100.00
1	Season 3	120.00
1	Season 4	130.00

续表

EmpID	SeasonNbr	Sales
2	Season 1	200.00
2	Season 2	300.00
2	Season 3	150.00

Employees 中存放着雇员信息, Orders 存放着雇员的季度销售数据。对于数据存储设计而言, Orders 表的结构设计便于数据存取, 并能够减少无效空间占用。但是, 对于人们的阅读习惯而言, 可能更希望看到如表 7-19 这样的报表格式数据, 因为它更便于进行数据比对。

表 7-19

报表格式数据

EmpName	Season1	Season2	Season3	Season4
Grace	100.00	100.00	120.00	130.00
Ken	200.00	300.00	150.00	NULL
Tom	NULL	NULL	NULL	NULL

现在假设我们就需要一个这样的报表。这实际上是一个表数据的旋转问题, 也就是将表中的行转换为列。下面的语句进行了 4 次左外部联接, 联接顺序自上而下依次进行。第一次是将 Employees 与 Orders (O1) 联接, 得到第一季度的销售数据, 如表 7-20 所示; 第二次是将表 7-20 所示的联接结果再次与 Orders (O2) 联接, 得到第二季度的销售数据, 如表 7-21 所示。依次类推, 执行完 4 次左外部联接后, 就得到了如表 7-19 所示的数据。

```
SELECT E1.EmpName,
       O1.Sales AS Season1,
       O2.Sales AS Season2,
       O3.Sales AS Season3,
       O4.Sales AS Season4
FROM Employees AS E1
LEFT OUTER JOIN Orders AS O1
  ON E1.EmpID = O1.EmpID AND O1.SeasonNbr = 'Season 1'
LEFT OUTER JOIN Orders AS O2
  ON E1.EmpID = O2.EmpID AND O2.SeasonNbr = 'Season 2'
LEFT OUTER JOIN Orders AS O3
  ON E1.EmpID = O3.EmpID AND O3.SeasonNbr = 'Season 3'
LEFT OUTER JOIN Orders AS O4
  ON E1.EmpID = O4.EmpID AND O4.SeasonNbr = 'Season 4';
```

表 7-20

第一次联接后的结果

E1.EmpID	E1.EmpName	O1.EmpID	O1.SeasonNbr	O1.Sales
1	Grace	1	Season 1	100.00
2	Ken	2	Season 1	200.00
3	Tom	NULL	NULL	NULL

表 7-21

第二次联接后的结果

E1.EmpID	E1.EmpName	O1.EmpID	O1.SeasonNbr	O1.Sales	O2.EmpID	O2.SeasonNbr	O2.Sales
1	Grace	1	Season 1	100.00	1	Season 2	100.00
2	Ken	2	Season 1	200.00	2	Season 2	300.00
3	Tom	NULL	NULL	NULL	NULL	NULL	NULL

上面的这个查询语句在执行效率方面似乎有点问题，每次联接只是为了得到一个季度的数据。我们知道，联接是要付出成本的。对于季度而言，这似乎还可以承受。如果要获得的是月度销售数据，难道要联接 12 次吗？下面的语句充分利用了 CASE 函数的，仅使用了一次左外部联接。

```
SELECT E1.EmpID, E1.EmpName,
       MAX(CASE
           WHEN O1.SeasonNbr = 'Season 1' THEN O1.Sales
           ELSE NULL
         END) AS Season1,
       MAX(CASE
           WHEN O1.SeasonNbr = 'Season 2' THEN O1.Sales
           ELSE NULL
         END) AS Season2,
       MAX(CASE
           WHEN O1.SeasonNbr = 'Season 3' THEN O1.Sales
           ELSE NULL
         END) AS Season3,
       MAX(CASE
           WHEN O1.SeasonNbr = 'Season 4' THEN O1.Sales
           ELSE NULL
         END) AS Season4
FROM Employees AS E1
LEFT OUTER JOIN Orders AS O1
ON E1.EmpID = O1.EmpID
GROUP BY E1.EmpID, E1.EmpName;
```

7.4.3 使用右外部联接保留右表全部行

右外部联接返回联接条件右边表中的全部行，并从联接条件左边的表中检索出与联接条件匹配的记录。图 7-6 演示了 T1 表与 T2 表按“T1.列 1=T2.列 1”条件进行右外部联接后的结果。

```
SELECT *
FROM T1 RIGHT OUTER JOIN T2
ON T1.列 1 = T2.列 1;
```

由图 7-6 可以看出，T2 表的全部行被保留。查询结果中的前两行是 T1 和 T2 中符合联接条件的行，第 3 行则是 T2 中不符合联接条件的行，该行是在添加外部行阶段被添加回来的。由于该行在 T1 中没有匹配行，所以行中 T1.列 1 和 T1.列 2 的值均为 NULL。

右外部联接的功能与左外部联接类似，大多数情况下，左外部联接完全可以取代右外部联接。例如，下面的语句只是交换了 7.4.2 小节的左外部联接示例语句中 Employees 和 Orders 表的位置，实现了右外部联接。二者的查询结果完全相同。

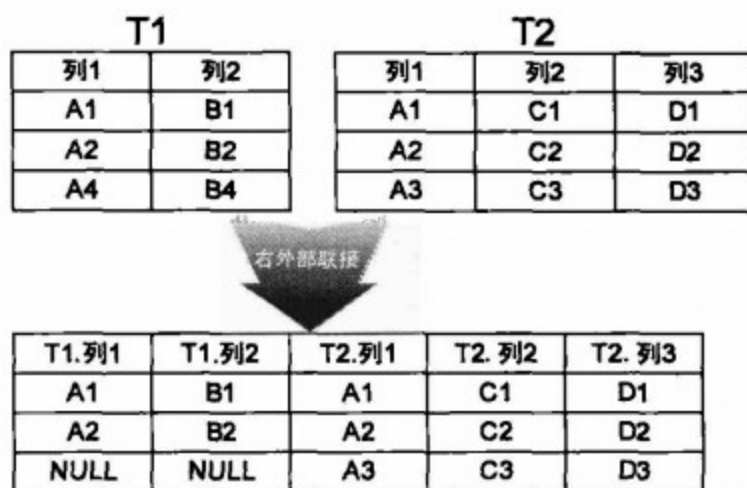


图 7-6 右外部联接后的结果

```

SELECT E1.EmpID, E1.EmpName,
       MAX(CASE
           WHEN O1.SeasonNbr = 'Season 1' THEN O1.Sales
           ELSE NULL
         END) AS Season1,
       MAX(CASE
           WHEN O1.SeasonNbr = 'Season 2' THEN O1.Sales
           ELSE NULL
         END) AS Season2,
       MAX(CASE
           WHEN O1.SeasonNbr = 'Season 3' THEN O1.Sales
           ELSE NULL
         END) AS Season3,
       MAX(CASE
           WHEN O1.SeasonNbr = 'Season 4' THEN O1.Sales
           ELSE NULL
         END) AS Season4
FROM Orders AS O1
RIGHT OUTER JOIN Employees AS E1
ON E1.EmpID = O1.EmpID
GROUP BY E1.EmpID, E1.EmpName;

```

7.4.4 使用完全外部联接保留两侧表全部行

完全外部联接包括联接条件两侧表中所有的匹配和不匹配记录。图 7-7 演示了 T1 表与 T2 表按“T1.列 1=T2.列 1”条件进行完全外部联接后的结果。

```

SELECT *
FROM T1 FULL OUTER JOIN T2
ON T1.列 1 = T2.列 1;

```

由图 7-7 可以看出，T1 和 T2 表的全部行被保留。查询结果中的前两行是 T1 和 T2 中符合联接条件的行，第 3 行则是 T1 中不符合联接条件的行，第 4 行是 T2 中不符合联接条件的行，这两行是在添加外部行阶段被添加回来的。

完全外部查询最直接的用处，就是通过包含 WHERE 子句返回两表之间没有匹配数据的行。例如，下面的语句将检索出 T1 表在 T2 表中没有匹配行的行，以及 T2 表在 T1 表中没有匹配行的行。即如图 7-7 所示的第 3 行和第 4 行。

T1		T2		
列1	列2	列1	列2	列3
A1	B1	A1	C1	D1
A2	B2	A2	C2	D2
A4	B4	A3	C3	D3

完全
外部联接

T1.列1	T1.列2	T2.列1	T2.列2	T2.列3
A1	B1	A1	C1	D1
A1	B1	A2	C2	D2
A4	B4	NULL	NULL	NULL
NULL	NULL	A3	C3	D3

图 7-7 完全外部联接后的结果

```

SELECT *
FROM T1 FULL OUTER JOIN T2
  ON T1.列1 = T2.列1
WHERE T1.列1 IS NULL OR T2.列1 IS NULL;

```

7.5 自联接

在前面介绍的无论是交叉联接，还是内部联接和外部联接，我们介绍的示例都是两个不同表之间的联接。实际上，表可以使用上述任意一种联接方式与自身联接。在自联接的情况下，左边表和右边表通常是以两种角色出现的，并通过为表指定不同的别名来进行区分。

7.5.1 使用不同列实现自联接

使用不同列进行自联接，表的两种角色区分比较明显。下面来看一个比较典型的自联接示例。如表 7-22 所示的 Employees 表中存储的是公司的雇员信息，其中的 MgrID 是雇员所在部门经理的 ID。按照这种对应关系，可以看出 Nancy 的部门经理为 Andrew。

表 7-22

Employees 表的内容

EmpID	EmpName	MgrID
1	Nancy	2
2	Andrew	NULL
3	Janet	2
4	Margaret	2
5	Steven	2
6	Michael	5
7	Robert	5
8	Laura	2
9	Anne	5

现在要找出每位雇员的部门经理的姓名，则可以两次打开 Employees 表，按 EmpID 与 MgrID 的对应关系进行联接。表中的 Andrew 没有部门经理，说明他是公司的高级管理人员，为了在查询结果中保留 Andrew，应当使用左外部联接。下面语句中的 E1 是为了获取 MgrID 信息，E2 是为了获取与 E1 中 MgrID 相匹配的部门经理姓名。查询结果如表 7-23 所示。

```
SELECT E1.EmpID, E1.EmpName, E1.MgrID, E2.EmpName AS MgrName
FROM Employees AS E1
LEFT OUTER JOIN Employees AS E2
ON E1.MgrID = E2.EmpID;
```

表 7-23 Employees 表的内容

EmpID	EmpName	MgrID	MgrName
1	Nancy	2	Andrew
2	Andrew	NULL	NULL
3	Janet	2	Andrew
4	Margaret	2	Andrew
5	Steven	2	Andrew
6	Michael	5	Steven
7	Robert	5	Steven
8	Laura	2	Andrew
9	Anne	5	Steven

7.5.2 使用同一列实现自联接

使用同一列实现表的自联接时，表角色的区分比较模糊。例如，有一个如表 7-24 所示的 Orders 表，存储的是雇员的销售额。

表 7-24 Orders 表中的内容

EmpID	Sales
1	100.00
2	100.00
3	120.00
4	130.00

现在要获取每位雇员的销售额，以及比其销售额高的雇员的平均销售额。例如，比 EmpID 为 1 高的是 3 和 4 雇员，他们两个的平均销售额为 125.00，即 120.00 加上 130.00 除以 2。下面的查询语句两次打开了 Orders 表，其中 O1 是为了获取雇员的销售额，O2 是为了获取大于 O1 中当前雇员的销售额数据。这实际上也是一个不等值联接查询。查询结果如表 7-25 所示。

```
SELECT O1.EmpID,
       MAX(O1.Sales) AS Sales,
       AVG(O2.Sales) AS AvgSales
```

```
FROM Orders AS O1
LEFT JOIN Orders AS O2
ON O1.Sales < O2.Sales
GROUP BY O1.OrderID;
```

表 7-25

查询结果

EmpID	Sales	AvgSales
1	100.00	125.00
2	100.00	125.00
3	120.00	130.00
4	130.00	NULL

7.6 多表联接

虽然每个联接规范只联接两个表，但可以在 FROM 子句中包含多个联接。在前面介绍的示例中，也涉及一些多表联接。在多表联接的情况下，由于每次只联接两个表，然后再将联接结果与下一个表进行联接，这就涉及一个联接的顺序问题。联接顺序不同，产生的结果可能并不相同。根据联接顺序的不同，可以将多表联接分为顺序联接和嵌套联接。

7.6.1 顺序联接

所谓顺序联接，就是按照 FROM 子句中联接的书写顺序进行表之间的联接，在将前两个表联接完成后，再将联接结果与第 3 个表进行联接。

下面的语句将创建 4 个示例表并向其中插入一些数据。其中 Productions 中存储的是不同型号自行车的产品信息，Sales 中存储的是各种自行车的销售数量，Spoliations 中存储的是已销售自行车的损坏数量，Stock 中存储的各种自行车的库存数量。

```
CREATE TABLE Productions
(ProductID int NOT NULL,
ProductName char(15) NOT NULL);
CREATE TABLE Sales
(ProductID int NOT NULL,
ProductCnt int NOT NULL);
CREATE TABLE Spoliations
(ProductID int NOT NULL,
ProductCnt int NOT NULL);
CREATE TABLE Stock
(ProductID int NOT NULL,
ProductCnt int NOT NULL);

INSERT INTO Productions
VALUES (1, 'Bike-51'),
(2, 'Bike-52'),
(3, 'Bike-53'),
(4, 'Bike-54');

INSERT INTO Sales
VALUES (1, 100).
```



```

        (2, 120),
        (3, 130);
INSERT INTO Spoliations
VALUES (1, 10),
        (3, 30);
INSERT INTO Stock
VALUES (2, 20),
        (3, 10),
        (4, 100);

```

现在假设要获取各型号自行车的销售、损坏、库存情况。分析上面的数据，可以看出应当使用左外部联接来实现，因为只有 **Productions** 中的型号信息最全。参考下面的语句：

```

SELECT P.ProductName,
       S.ProductCnt AS SaleCnt,
       Sp.ProductCnt AS SpoliatCnt,
       St.ProductCnt AS StockCnt
FROM Productions AS P
LEFT OUTER JOIN Sales AS S
  ON P.ProductID = S.ProductID
LEFT OUTER JOIN Spoliations AS Sp
  ON P.ProductID = Sp.ProductID
LEFT OUTER JOIN Stock AS St
  ON P.ProductID = St.ProductID;

```

上面的语句中的联接自上而下依次执行，首先将 **Productions** 与 **Sales** 进行联接，联接结果然后与 **Spoliations** 联接，联接结果再与 **Stock** 联接。查询结果如表 7-26 所示。

表 7-26

查询结果

ProductName	SaleCnt	SpoliatCnt	StockCnt
Bike-51	100	10	NULL
Bike-52	120	NULL	20
Bike-53	130	30	10
Bike-54	NULL	NULL	100

7.6.2 嵌套联接

所谓**嵌套联接**，就是在联接中存在着层次关系，最里层的联接优先执行，然后将联接结果再与外层进行联接。与顺序联接最大的区别就是嵌套联接的 **ON** 子句是交错放置的，图 7-8 演示了顺序联接与嵌套连结 **ON** 子句位置的差别和执行顺序的差别。

顺序联接

```

SELECT *
FROM T1
  INNER JOIN T2
    ON T1.ID = T2.ID
  INNER JOIN T3
    ON T2.ID = T3.ID
  INNER JOIN T4
    ON T3.ID = T4.ID

```

嵌套联接

```

SELECT *
FROM T1
  INNER JOIN T2
    INNER JOIN T3
      INNER JOIN T4
        ON T3.ID = T4.ID
      ON T2.ID = T3.ID
    ON T1.ID = T2.ID

```

图 7-8 顺序联接与嵌套联接时 ON 子句的位置和执行顺序差别

仍旧使用上面创建的示例表和数据,现在假设要获取已销售自行车中存在损坏自行车的型号的损坏率。首先使用顺序语句的方式来解决这个问题。下面的查询首先将 Sales 与 Spoliations 进行内部联接,获得了损坏比率,查询结果如表 7-27 所示。

```
SELECT S.ProductID,
       S.ProductCnt AS SaleCnt,
       Sp.ProductCnt AS SpCnt,
       CAST(1. * Sp.ProductCnt / S.ProductCnt AS decimal(5,2)) AS Per
FROM Sales AS S
     INNER JOIN Spoliations AS Sp
       ON S.ProductID = Sp.ProductID;
```

表 7-27 损坏比率

ProductID	SaleCnt	SpCnt	Per
1	100	10	0.10
3	130	30	0.23

下一步应当将上面的结果与 Productions 进行外部联接,获得自行车名称信息。下面是按照顺序执行方式编写的查询语句。注意其中使用了右外部联接。

```
SELECT P.ProductName, S.ProductID,
       S.ProductCnt AS SaleCnt,
       Sp.ProductCnt AS SpCnt,
       CAST(1. * Sp.ProductCnt / S.ProductCnt AS decimal(5,2)) AS Per
FROM Sales AS S
     INNER JOIN Spoliations AS Sp
       ON S.ProductID = Sp.ProductID
     RIGHT OUTER JOIN Productions AS P
       ON S.ProductID = P.ProductID;
```

得到的查询结果如表 7-28 所示。

表 7-28 查询结果

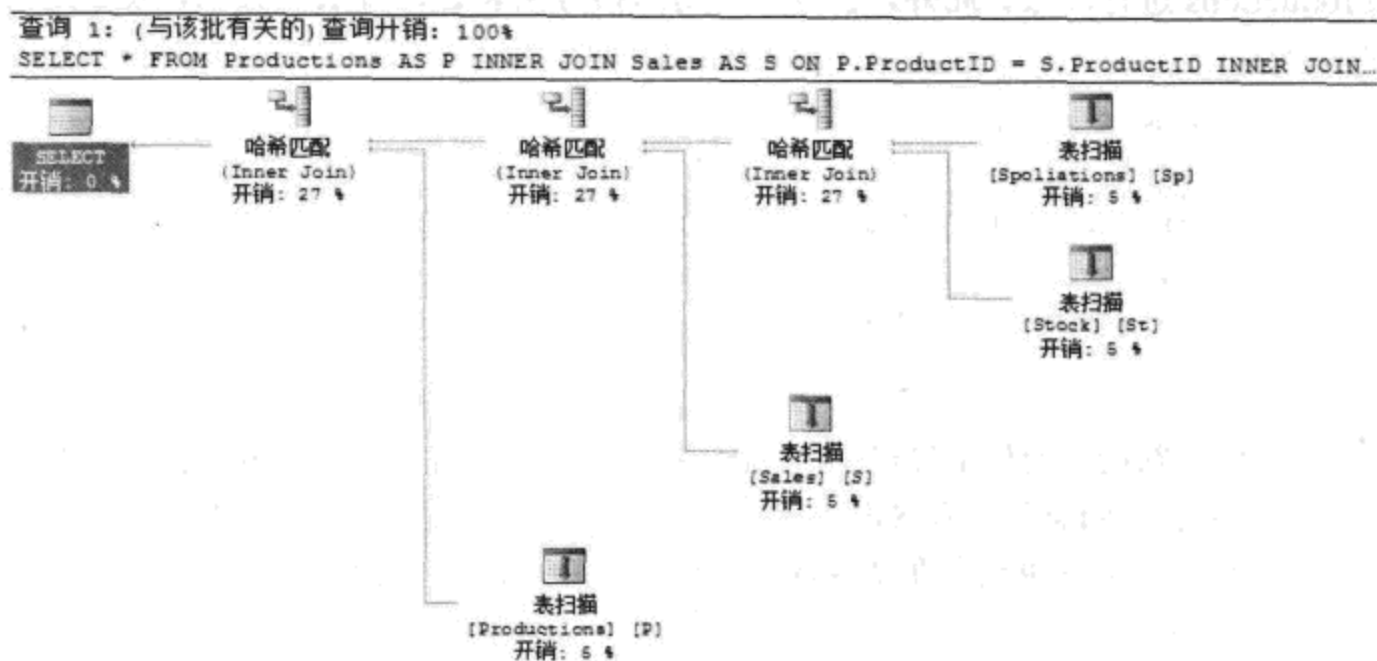
ProductName	ProductID	SaleCnt	SpCnt	Per
Bike-51	1	100	10	0.10
Bike-52	NULL	NULL	NULL	NULL
Bike-53	3	130	30	0.23
Bike-54	NULL	NULL	NULL	NULL

如果我们希望将 Productions 表放在 FROM 子句的第一位,使用左外部联接的方式实现,则应当使用嵌套联接。下面的语句首先将里层的 Sales 与 Spoliations 进行联接,然后将联接结果再与外层的 Productions 执行左外部联接。查询结果与表 7-28 所示的内容完全相同。

```
SELECT P.ProductName, S.ProductID,
       S.ProductCnt AS SaleCnt,
       Sp.ProductCnt AS SpCnt,
       CAST(1. * Sp.ProductCnt / S.ProductCnt AS decimal(5,2)) AS Per
FROM Productions AS P
     LEFT OUTER JOIN (
         SELECT S.ProductID,
                S.ProductCnt AS SaleCnt,
                Sp.ProductCnt AS SpCnt,
                CAST(1. * Sp.ProductCnt / S.ProductCnt AS decimal(5,2)) AS Per
         FROM Sales AS S
              INNER JOIN Spoliations AS Sp
                ON S.ProductID = Sp.ProductID
       ) AS S
       ON S.ProductID = P.ProductID;
```



行计划都具有相关成本。查询优化器必须分析可能的计划并选择一个预计成本最低的计划。有些复杂的 SELECT 语句有成千上万个可能的执行计划。在这些情况下，查询优化器不会分析所有的可能组合，而是使用复杂的算法查找一个执行计划：其成本合理地接近最低可能成本。



同时，查询优化器不只选择资源成本最低的执行计划，还选择能将结果最快地返回给用户且资源成本合理的计划。例如，通常并行处理查询使用的资源比串行处理要多，但完成查询的速度更快。因此如果不对服务器的负荷产生负面影响，优化器将使用并行执行计划返回结果。

因此，如果认为优化器生成的执行计划不是最优方案，可以使用 **FORCE ORDER** 提示强制优化器按语句的逻辑顺序处理联接。参考以下语句：

```
SELECT *
FROM Productions AS P
INNER JOIN Sales AS S
ON P.ProductID = S.ProductID
INNER JOIN Spoiliations AS Sp
ON P.ProductID = Sp.ProductID
INNER JOIN Stock AS St
ON P.ProductID = St.ProductID
OPTION (FORCE ORDER);
```

上面语句的执行计划如图 7-10 所示，可以看出它与语句的逻辑执行顺序相同。

另一个强制按逻辑顺序处理联接的方法是使用 **SET FORCEPLAN** 选项，该选项将影响会话中的所有查询。虽然查询优化器得到的并不一定是最优的执行计划，但是随着表中数据量、索引等的变化，当前的最优执行计划并不一定完全适应将来。所以，应当尽量避免使用这些提示，如果必须使用，则应当定期检查受提示影响的查询语句，是否还适应当前的数据。

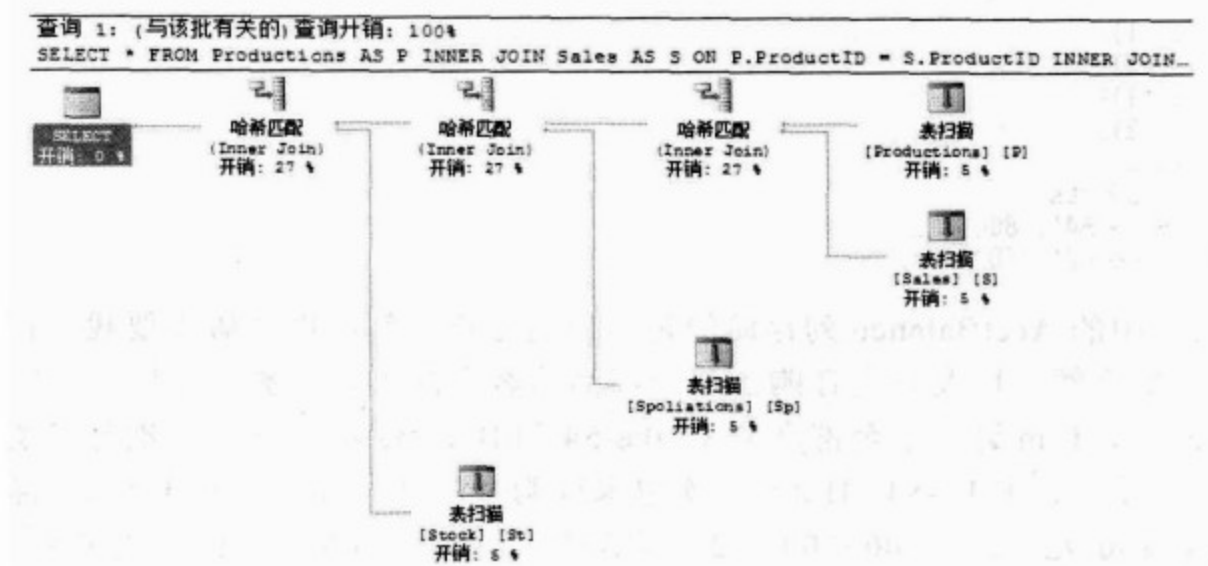


图 7-10 强制按逻辑顺序生成执行计划

7.6.4 多表联接示例

下面来看一个多表联接的示例。在该示例中既包含顺序联接，又包含嵌套联接。首先使用下面的语句创建示例表和所需要的数据，这是一个典型模式的公司表示例，包括客户表、订单表、订单清单表和产品表。

```
CREATE TABLE Customers
  (CustID int NOT NULL PRIMARY KEY,
   CustName char(12) NOT NULL,
   AcctBalance money NOT NULL);
CREATE TABLE Orders
  (CustID int NOT NULL,
   OrderID int NOT NULL PRIMARY KEY,
   ShipDate datetime);
CREATE TABLE OrderDetails
  (OrderID int NOT NULL,
   ItemID int NOT NULL,
   ItemQty int NOT NULL);
CREATE TABLE Products
  (ItemID int NOT NULL PRIMARY KEY,
   ProductName char(15) NOT NULL,
   UnitPrice money NOT NULL);

INSERT INTO Customers
VALUES (1, 'Grace', 200.00),
      (2, 'Tom', 100.00),
      (3, 'Ken', 1000.00),
      (4, 'Eddie', 2000.00),
      (5, 'Holly', 1000.00);

INSERT INTO Orders
VALUES (1, 1, '2009-3-1'),
      (2, 2, '2009-3-2'),
      (3, 3, '2009-3-3'),
      (3, 4, '2009-3-6'),
      (4, 5, '2009-3-7');

INSERT INTO OrderDetails
VALUES (1, 1, 2),
      (1, 2, 1),
```

```
(2, 1, 1),
(2, 2, 1),
(3, 1, 1),
(4, 1, 2),
(5, 1, 1);
INSERT INTO Products
VALUES (1, 'Bike-54', 800.00),
(2, 'Bike-52', 600.00);
```

Customers 中的 **AcctBalance** 列存储的是客户的账户余额，现在假设要找出订购了所有产品的客户的平均余额，以及只是订购了部分产品的客户的平均余额。分析上面的数据可以看出，只有 **Grace** 和 **Tom** 订购了全部产品 (**Bike-54** 和 **Bike-52**)，**Ken** 是两次都订购了 **Bike-54**，**Eddie** 是只订购了一次 **Bike-54**，**Holly** 一次也未订购过产品。因此，订购全部产品的客户的平均余额应当是 150 元，即 $(200+100)/2$ 。只订购了部分产品的客户的平均余额为 1500 元，即 $(1000+2000)/2$ 。

下面的查询语句中，首先执行的是 **Orders** 与 **OrderDetails** 的联接，得到每个客户的订单数量，如表 7-29 所示。然后将联接结果 **Customers** 联接，得到客户的账户余额，如表 7-30 所示。再将联接结果与 **Products** 中的产品数量表 **AllProducts** 进行交叉联接，将产品数量分配到每个客户行中，如表 7-31 所示。最后进行的数据分组，计算平均值，得到如表 7-32 所示的最终结果。

```
SELECT AVG(AcctBalance) AS AvgBal, OrderedDesc
FROM (SELECT Customers.CustID, Customers.AcctBalance,
CASE WHEN OrderedProductCnt = AllProductCnt
THEN 'Ordered all'
ELSE 'Not ordered all'
END AS OrderedDesc
FROM Customers
INNER JOIN
(SELECT CustID, COUNT(DISTINCT ItemID) AS OrderedProductCnt
FROM Orders
INNER JOIN OrderDetails
ON Orders.OrderID = OrderDetails.OrderID
GROUP BY CustID
) AS OrderedProducts
ON Customers.CustID = OrderedProducts.CustID
CROSS JOIN
(SELECT COUNT(DISTINCT ItemID) AS AllProductCnt
FROM Products) AS AllProducts
) AS T
GROUP BY OrderedDesc
```

表 7-29

Orders 与 OrderDetails 联接的结果

CustID	OrderedProductCnt
1	2
2	2
3	1
4	1

表 7-30 与 Customers 联接的结果

CustID	OrderedProductCnt	AcctBalance
1	2	200.00
2	2	100.00
3	1	1000.00
4	1	2000.00


表 7-31 与 AllProducts 联接的结果

CustID	OrderedProductCnt	AcctBalance	AllProductCnt	OrderedDesc
1	2	200.00	2	Ordered all
2	2	100.00	2	Ordered all
3	1	1000.00	2	Not ordered all
4	1	2000.00	2	Not ordered all

表 7-32 最终查询结果

AvgBal	OrderedDesc
1500.00	Not ordered all
150.00	Ordered all

7.7 联接算法

在 Microsoft SQL Server Management Studio 中执行查询时，如果选定工具栏中的  按钮，可以看到为查询生成的执行计划。执行计划以图形方式显示了查询优化器选择的数据检索方法，如表扫描、排序、哈希匹配等。对于联接查询，优化器会根据联接表之间的数据、索引等情况，选择使用嵌套循环联接、合并联接或哈希联接。

7.7.1 嵌套循环联接

嵌套循环联接也称为“嵌套迭代”，它将一个联接输入用作外部输入表（显示为图形执行计划中的顶端输入），将另一个联接输入用作内部（底端）输入表。外部循环逐行处理外部输入表。内部循环会针对每个外部行执行，在内部输入表中搜索匹配行。简单地讲，就是扫描其中的一个联接表，并为该表中的每一行在另一个联接表中搜索匹配行。

如果外部输入较小（不到 10 行）而内部输入较大且预先创建了索引，则嵌套循环联接尤其有效。在许多小事务中（如那些只影响较小的一组行的事务），索引嵌套循环联接优于合并联接和哈希联接。但在大型查询中，嵌套循环联接通常不是最佳选择。

例如，下面的查询由于 Sales.Customer 表行数只有 1 行，而 Sales.SalesOrderHeader 数据量较大，因此将使用嵌套循环联接，生成的执行计划如图 7-11 所示。

```

USE AdventureWorks;
GO
SELECT *
FROM Sales.Customer
    INNER JOIN Sales.SalesOrderHeader
        ON Customer.CustomerID = SalesOrderHeader.CustomerID
WHERE Customer.CustomerID = 1;

```

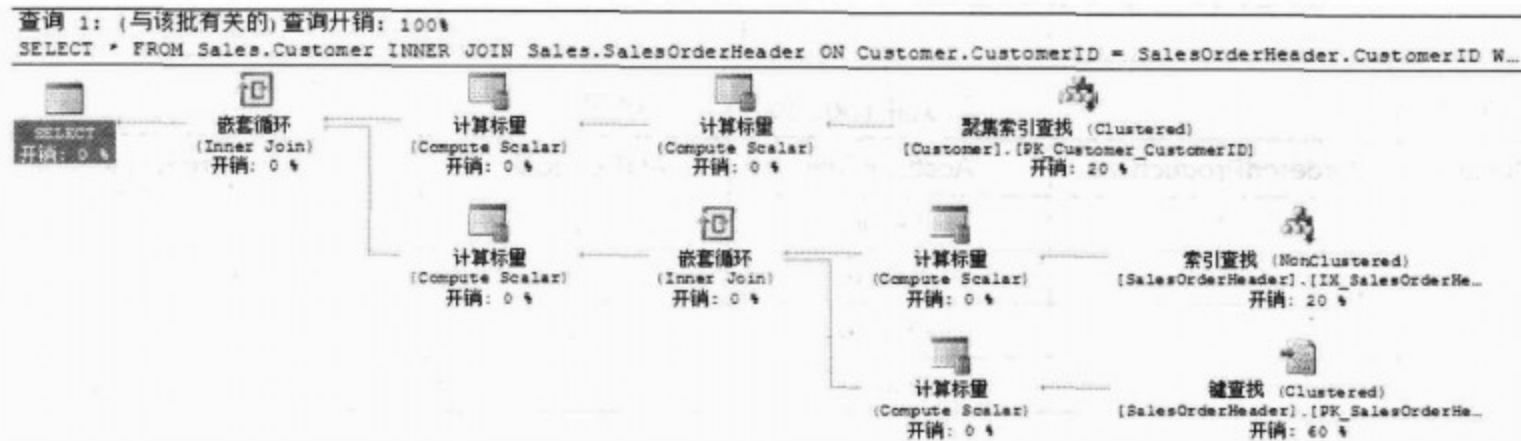


图 7-11 使用嵌套循环的执行计划

在该计划中存在两个嵌套循环，其中只有左边的嵌套循环符用于 `Sales.Customer` 与 `Sales.SalesOrderHeader` 的联接，而右边的嵌套循环是用于 `Sales.SalesOrderHeader` 的索引查找与物理行定位（键查找）之间的联接。执行计划右上角的 `Sales.Customer` 表被作为外部输入，在聚集索引中查找客户。对于每个客户，嵌套循环运算将对 `SalesOrderHeader.CustomerID` 列上的 `IX_SalesOrderHeader_CustomerID` 索引执行一次查找，然后再跟一个键查找来定位要访问的数据行。

7.7.2 合并联接

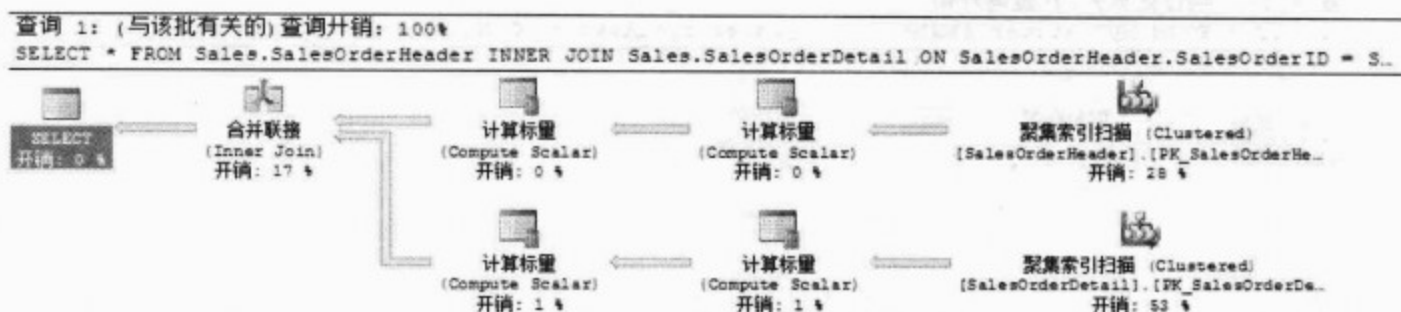
合并联接要求两个输入都在合并列上排序，合并列由联接谓词的等效（ON）子句定义。由于每个输入都已排序，因此合并联接将从每个输入获取一行并将其进行比较。例如，对于内联接操作，如果行相等则返回。如果行不相等，则废弃值较小的行并从该输入获得另一行。这一过程将重复进行，直到处理完所有的行为止。

合并联接操作可以是常规操作，也可以是多对多操作。多对多合并联接使用临时表存储行。如果每个输入中有重复值，则在处理其中一个输入中的每个重复项时，另一个输入必须重绕到重复项的开始位置。

合并联接本身的速度很快，但是如果合并列上未建立索引，选择合并联接有可能会非常费时，因为它首先要对列进行排序操作。然而，如果数据量很大且能够从索引中获得预排序的所需数据，则合并联接通常是最快的可用联接算法。

例如，下面的查询语句将获取订单的详细信息，由于 `SalesOrderHeader` 和 `SalesOrderDetail` 在合并列 `SalesOrderID` 上都具有聚集索引，已经将列进行了排序，所以查询优化器会选择合并联接。如图 7-12 所示。


```
USE AdventureWorks;
GO
SELECT *
FROM Sales.SalesOrderHeader
INNER JOIN Sales.SalesOrderDetail
ON SalesOrderHeader.SalesOrderID = SalesOrderDetail.SalesOrderID;
```



7.7.3 哈希联接

哈希联接可以有效处理未排序的大型非索引输入。因此，它对处理复杂查询的中间结果很有用。查询的中间结果是未经索引的，而且通常不会为查询计划中的下一个操作进行适当的排序。并且，查询优化器只估计中间结果的大小。而对于复杂查询，估计可能有很大的误差，因此如果中间结果比预期的大得多，则处理中间结果的算法不仅必须有效而且必须适度弱化。再像合并联接那样严格要求具备排序列，对于中间结果而言是不现实的，排序成本的付出可能要远远大于数据的直接检索成本。

选择哈希联接的两种情况：一是没有为联接创建合适的索引，二是中间结果比较大。

哈希联接有两种输入：生成输入和探测输入。查询优化器会选择二者中较小的那个作为生成输入，对联接列值应用哈希函数，将生成输入中的行分配到哈希桶中。哈希桶是一种存放所访问数据位置的结构，有了它，进行数据检索时，可以避免不必要的表扫描。

为了验证无索引情况下的哈希联接使用，首先使用下面的语句创建 Sales.Customer 和 Sales.SalesOrderHeader 表的副本。

```
USE AdventureWorks;
GO
SELECT TOP 10 *
INTO MyCustomer
FROM Sales.Customer
ORDER BY CustomerID;

SELECT TOP 100 *
INTO MySalesOrderHeader
FROM Sales.SalesOrderHeader
ORDER BY CustomerID;
```

执行下面的查询，可以看到如图 7-13 所示的执行计划。

```
SELECT *
FROM MyCustomer
  INNER JOIN MySalesOrderHeader
    ON MyCustomer.CustomerID = MySalesOrderHeader.CustomerID;
```

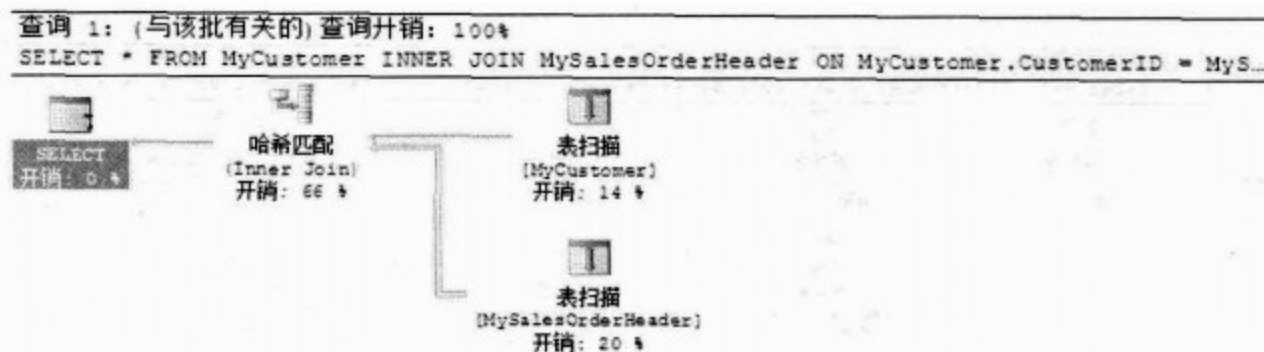


图 7-13 使用哈希联接的执行计划

下面再来看一个比较有趣的示例。下面的查询语句中仅选择了 Sales.Customer 中 CustomerID = 1 的行与 Sales.SalesOrderHeader 进行联接, 由于联接行数很小, 所产生中间结果的数据量也比较小, 因此, 可以看到查询优化器为语句使用了嵌套循环联接。如图 7-14 所示。

```
USE Adventureworks;
GO
SELECT *
FROM Sales.Customer
  INNER JOIN Sales.SalesOrderHeader
    ON Customer.CustomerID = SalesOrderHeader.CustomerID
WHERE Customer.CustomerID = 1;
```

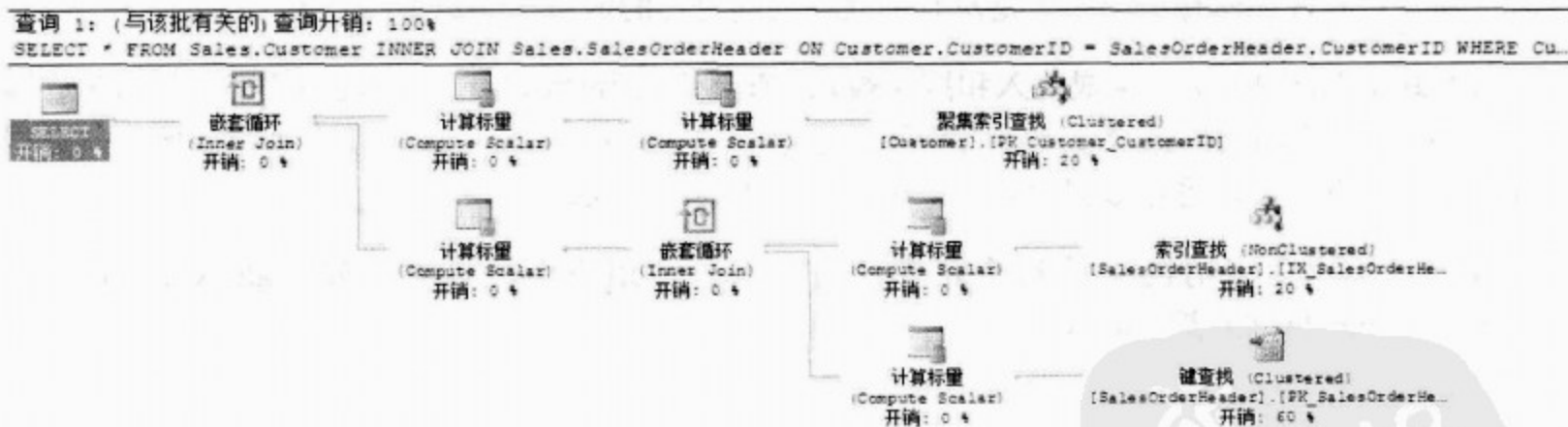


图 7-14 数据量较小时使用嵌套循环联接

同样是上面的联接, 去除掉 WHERE 筛选条件后数据量明显增大, 执行该语句会发现查询优化器使用了哈希联接方式。如图 7-15 所示。

```
SELECT *
FROM Sales.Customer
  INNER JOIN Sales.SalesOrderHeader
    ON Customer.CustomerID = SalesOrderHeader.CustomerID;
```



图 7-15 数据量较大时使用哈希联接

7.7.4 使用联接提示强制联接策略

联接提示用于指定查询优化器在两个表之间强制执行联接策略，提示符包括 **LOOP JOIN**、**MERGE JOIN** 和 **HASH JOIN**，分别用于嵌套循环、哈希和合并联接。如果指定了多个联接提示，则优化器从允许的联接策略中选择开销最少的联接策略。此外，也可以使用 **OPTION** 子句指定联接策略。但是这种方式会影响查询中的所有联接，通常用于旧式联接语法。

1. 为每个联接指定单独的联接策略

可以在 **FROM** 子句中使用 **LOOP JOIN**、**MERGE JOIN** 和 **HASH JOIN** 提示符为每个联接单独指定联接策略。例如，下面的查询语句指定使用嵌套循环联接。

```
USE AdventureWorks;
GO
SELECT *
FROM Sales.Customer
    INNER LOOP JOIN Sales.SalesOrderHeader
        ON Customer.CustomerID = SalesOrderHeader.CustomerID;
```

又如，下面的查询语句指定使用合并联接。

```
USE AdventureWorks;
GO
SELECT *
FROM Sales.Customer
    INNER MERGE JOIN Sales.SalesOrderHeader
        ON Customer.CustomerID = SalesOrderHeader.CustomerID;
```

在多表联接中使用联接提示时，会影响联接的执行顺序。前面已经介绍了，在不影响返回结果正确的情况下，查询优化器会按照效率优先的原则，选择首先执行的联接。例如，下面语句的执行计划如图 7-16 所示，可以看到首先执行的是 **Sales.SalesOrderHeader** 与 **Sales.SalesOrderDetail** 的联接，然后将联接结果再与 **Sales.Customer** 进行联接。

```
USE AdventureWorks;
GO
SELECT *
FROM Sales.Customer
    INNER JOIN Sales.SalesOrderHeader
        ON Customer.CustomerID = SalesOrderHeader.CustomerID
    INNER JOIN Sales.SalesOrderDetail
        ON SalesOrderHeader.SalesOrderID = SalesOrderDetail.SalesOrderID;
```

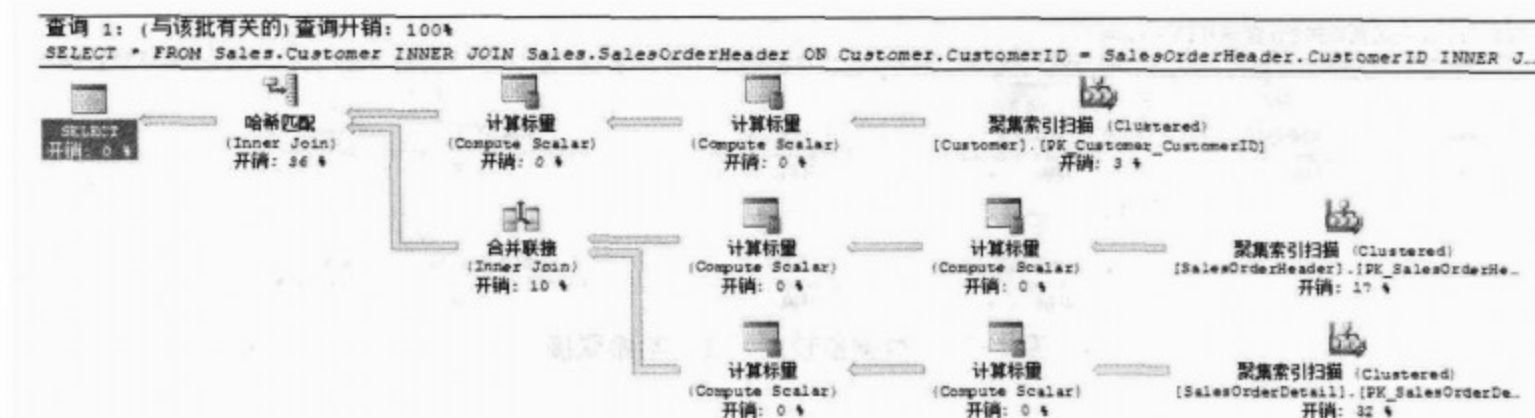


图 7-16 未使用联接提示的执行计划

下面的语句为 Sales.Customer 和 Sales.SalesOrderHeader 指定了合并联接提示, 并且这个提示仅对这两个表起作用, 与 Sales.SalesOrderDetail 的联接策略仍旧由查询优化器决定。由于明确指定了 Sales.Customer 与 Sales.SalesOrderHeader 使用合并联接, 优化器会先执行该联接, 而不是先执行 Sales.SalesOrderHeader 与 Sales.SalesOrderDetail 的联接。否则, 就会造成 Sales.Customer 与 Sales.SalesOrderHeader 和 Sales.SalesOrderDetail 的联接结果再执行合并联接。图 7-17 是该语句的执行计划。

```
SELECT *
FROM Sales.Customer
    INNER MERGE JOIN Sales.SalesOrderHeader
        ON Customer.CustomerID = SalesOrderHeader.CustomerID
    INNER JOIN Sales.SalesOrderDetail
        ON SalesOrderHeader.SalesOrderID = SalesOrderDetail.SalesOrderID;
```

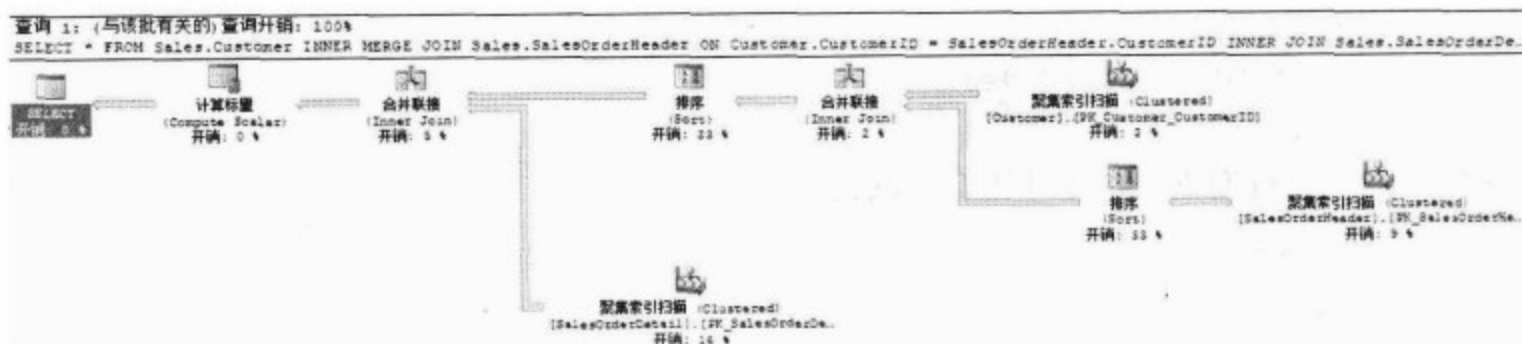


图 7-17 使用联接提示后的执行计划

如果希望 Sales.Customer 与 Sales.SalesOrderHeader 和 Sales.SalesOrderDetail 的联接结果执行合并联接, 则应当使用嵌套联接的方式实现, 参考下面的语句:

```
SELECT *
FROM Sales.Customer
    INNER MERGE JOIN (Sales.SalesOrderHeader
        INNER JOIN Sales.SalesOrderDetail
            ON SalesOrderHeader.SalesOrderID = SalesOrderDetail.SalesOrderID)
        ON Customer.CustomerID = SalesOrderHeader.CustomerID;
```

2. 为全部联接指定统一的联接策略

当使用旧式联接语法时, 应当使用 OPTION 子句指定联接策略, 但是, 这种策略会影响语句

中的全部联接，无法为每个联接单独指定不同的联接策略，如：

```
SELECT *
FROM Sales.Customer, Sales.SalesOrderHeader, Sales.SalesOrderDetail
WHERE Customer.CustomerID = SalesOrderHeader.CustomerID
AND SalesOrderHeader.SalesOrderID = SalesOrderDetail.SalesOrderID
OPTION (MERGE JOIN);
```

该语句的执行计划如图 7-18 所示，可以看到 3 个表之间全部使用了合并联接策略。

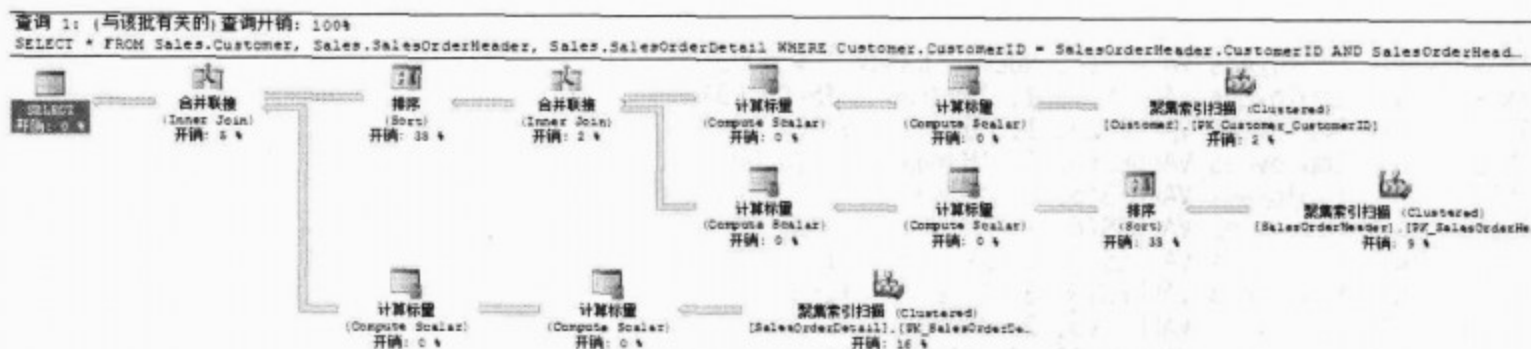


图 7-18 为全部联接使用统一联接策略的执行计划

在 ANSI SQL:1992 规范中，也可以使用 `OPTION` 子句，它同样也是影响语句中的全部联接，如：

```
SELECT *
FROM Sales.Customer
INNER JOIN Sales.SalesOrderHeader
ON Customer.CustomerID = SalesOrderHeader.CustomerID
INNER JOIN Sales.SalesOrderDetail
ON SalesOrderHeader.SalesOrderID = SalesOrderDetail.SalesOrderID
OPTION (MERGE JOIN);
```

7.8 使用 APPLY 运算符

从 SQL Server 2005 开始，提供了 `APPLY` 运算符，因此要使用该运算符，数据库兼容级别必须设置为 90。`APPLY` 运算符的功能与联接十分类似，包含 `APPLY` 的查询也是分为左右两部分，对左表表达式返回的每一行都要调用一次右表表达式，进行计算。也就是说，右表表达式是根据左表表达式提供的值进行计算，来获取结果集的。然后将左输入和右输入组合起来作为最终输出。右表表达式可以是一个相关子查询，也可以是一个表值函数。

`APPLY` 运算符与联接不同的地方是，联接首先执行的是将左表与右表进行笛卡尔乘积，先计算哪个表表达式都可以，而 `APPLY` 必须先计算左表表达式，计算结果作为右表表达式检索数据的参数值。因此，包含 `APPLY` 运算符生成的列的列表首先是左表表达式中的列集，然后是右表表达式的列集。

`APPLY` 有两种形式：`CROSS APPLY` 和 `OUTER APPLY`。`CROSS APPLY` 类似于内部联接，它返回右表表达式能够生成结果集的行。`OUTER APPLY` 类似于左外部联接，它既返回生成结果集的行，也返回不生成结果集的行。

下面通过示例来说明 **APPLY** 运算符的使用方法。首先使用下面的代码创建两个表：**Employees** 和 **Departments**。

```
CREATE TABLE Employees
(
    empid int NOT NULL,
    deptid int NULL,
    empname varchar(25) NOT NULL,
    salary money NOT NULL CONSTRAINT PK_Employees PRIMARY KEY(empid)
);
GO
INSERT INTO Employees VALUES(1, NULL, 'Nancy', $10000.00);
INSERT INTO Employees VALUES(2, 1, 'Andrew', $5000.00);
INSERT INTO Employees VALUES(3, 1, 'Janet', $5000.00);
INSERT INTO Employees VALUES(4, 1, 'Margaret', $5000.00);
INSERT INTO Employees VALUES(5, 2, 'Steven', $2500.00);
INSERT INTO Employees VALUES(6, 2, 'Michael', $2500.00);
INSERT INTO Employees VALUES(7, 3, 'Robert', $2500.00);
INSERT INTO Employees VALUES(8, 3, 'Laura', $2500.00);
INSERT INTO Employees VALUES(9, 3, 'Ann', $2500.00);
INSERT INTO Employees VALUES(10, 4, 'Ina', $2500.00);
INSERT INTO Employees VALUES(11, 7, 'David', $2000.00);
INSERT INTO Employees VALUES(12, 7, 'Ron', $2000.00);
INSERT INTO Employees VALUES(13, 7, 'Dan', $2000.00);
INSERT INTO Employees VALUES(14, 11, 'James', $1500.00);
GO

CREATE TABLE Departments
(
    deptid int NOT NULL PRIMARY KEY,
    deptname varchar(25) NOT NULL,
);
GO
INSERT INTO Departments VALUES(1, 'HR');
INSERT INTO Departments VALUES(2, 'Marketing');
INSERT INTO Departments VALUES(3, 'Finance');
INSERT INTO Departments VALUES(4, 'R&D');
INSERT INTO Departments VALUES(5, 'Training');
INSERT INTO Departments VALUES(6, 'Gardening');
```

Departments 表中的多数部门都具有一个经理 ID，这些 ID 与 **Employees** 表中的雇员相对应。下面的表值函数接受部门 ID 作为参数，并返回该部门的雇员。

```
CREATE FUNCTION dbo.fn_getDeptEmployees(@deptid AS int)
RETURNS TABLE
AS
RETURN
SELECT empid, empname, salary
FROM Employees
WHERE deptid = @deptid;
GO
```

要返回每个部门的雇员，可以使用下面的语句。查询结果如表 7-33 所示。

```
SELECT D.deptname, E.empid, E.empname, E.salary
FROM Departments AS D
CROSS APPLY fn_getDeptEmployees(D.deptid) AS E;
```

表 7-33 CROSS APPLY 查询结果

deptname	empid	empname	salary
HR	2	Andrew	5000.00
HR	3	Janet	5000.00
HR	4	Margaret	5000.00
Marketing	5	Steven	2500.00
Marketing	6	Michael	2500.00
Finance	7	Robert	2500.00
Finance	8	Laura	2500.00
Finance	9	Ann	2500.00
R&D	10	Ina	2500.00

由于 Training 和 Gardening 部门在 Employees 表中没有雇员, 所以这两个部门没有显示查询结果中。与表值函数相结合进行数据检索是 APPLY 的典型应用, 实际上以上语句也可以改写成下面子查询的方式。

```
SELECT D.deptname, E.empid, E.empname, E.salary
FROM Departments AS D
CROSS APPLY (SELECT empid, empname, salary
              FROM Employees
              WHERE deptid = D.deptid) AS E;
```

上面的查询也可以使用内部联接的方式实现, 如:

```
SELECT D.deptname, E.empid, E.empname, E.salary
FROM Departments AS D
INNER JOIN Employees AS E
ON D.deptid = E.deptid;
```

APPLY 的另一种形式是 OUTER APPLY, 在这种情况下, 所有部门名称都将显示在结果集中, 由于 Training 和 Gardening 部门在 Employees 表中没有雇员, 所以显示为空值。下面语句的查询结果如表 7-34 所示。

```
SELECT D.deptname, E.empid, E.empname, E.salary
FROM Departments AS D
OUTER APPLY fn_getDeptEmployees(D.deptid) AS E;
```

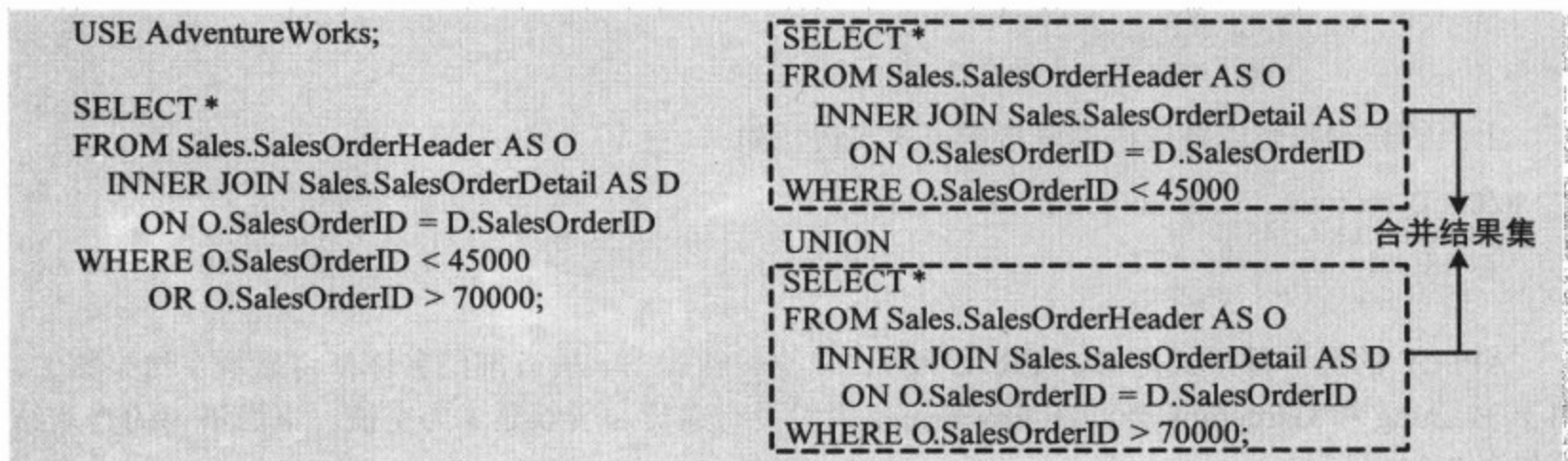
表 7-34 OUTER APPLY 查询结果

deptname	empid	empname	salary
HR	2	Andrew	5000.00
HR	3	Janet	5000.00
HR	4	Margaret	5000.00
Marketing	5	Steven	2500.00
Marketing	6	Michael	2500.00
Finance	7	Robert	2500.00
Finance	8	Laura	2500.00
Finance	9	Ann	2500.00
R&D	10	Ina	2500.00
Training	NULL	NULL	NULL
Gardening	NULL	NULL	NULL



第8章 操作结果集

在上一章中已经介绍了联接操作，而本章将重点介绍对查询结果集合的操作。联接操作无论多复杂，通常都被看作是一条查询语句（即使里面可能包含有子查询），而结果集操作则是很明显对两条查询语句结果的再次整合处理。例如，下面左边的语句是一个标准的内部联接语句，查询结果的数据范围为 SalesOrderHeader.SalesOrderID 小于 45000 或是大于 70000。右边的语句则是将这两个条件分别写在两个联接语句中，然后将两个结果集使用 UNION 运算符合并在一起。



联接与结果集操作的主要区别是：联接是根据联接条件对两个表的指定列进行比较，并将两个表的指定列联结在一起；而结果集操作是把两个查询得到的结果集追加在一起，它不会引起列的变化。由于是追加操作，所以要求两个结果集的列数应当相同，并且相应列的数据类型也应当相同，或是能够隐式转换。如果需要比较，是对两个结果集中的全部列进行比较。可以说，联接操作是两个表横向的结合，而结果集操作是两个结果集纵向的结合。

8.1 合并结果集

UNION 运算符可以将两个或多个 SELECT 语句的结果组合成一个结果集，其语法格式如下：

| select_statement UNION [ALL] select_statement

select_statement 是要组合的 SELECT 语句，各语句中对应结果集列的顺序必须相同，因为 UNION 运算符按照各个查询中给定的顺序一对一地组合各列。一个 SQL 语句中可以出现任意数目的 UNION 运算符。

8.1.1 使用 UNION 与 UNION ALL 进行结果集合并

默认情况下，UNION 运算符将从结果集中删除重复的行。如果使用了 ALL 关键字，则结果中将包含所有的行。使用下面的语句创建 Table1 和 Table2 两个表，并向其中插入部分示例数据，内容分别如表 8-1 和表 8-2 所示。

```
CREATE TABLE Table1
(A int, B char(4), C char(4));
CREATE TABLE Table2
(A char(4), B decimal(5,4));
INSERT INTO Table1
VALUES (1, 'ABC', 'JKL'),
      (2, 'DEF', 'MNO'),
      (3, 'GHI', 'PQR');
INSERT INTO Table2
VALUES ('JKL', 1.0000),
      ('DEF', 2.0000),
      ('MNO', 5.0000);
```

表 8-1 Table1 表中的内容

A	B	C
1	ABC	JKL
2	DEF	MNO
3	GHI	PQR

表 8-2 Table2 表中的内容

A	B
JKL	1.0000
DEF	2.0000
MNO	5.0000

以下两个要组合的 SELECT 语句考虑了对应列的情况，Table2 表的 B 列与 Table1 表的 A 列的数据类型是兼容的，可以进行隐式转换。根据数据类型的优先级规则，结果集中将使用 decimal 数据类型。由于 Table1 中的 (2, DEF) 行与 Table2 中存在重复行，因此会被删除掉。组合后的结果集如表 8-3 所示。

```
SELECT a, b FROM table1
UNION
SELECT b, a FROM table2;
```

表 8-3 Table1 表和 Table2 表使用 UNION 组合后的结果集

A	B
1.0000	ABC
1.0000	JKL
2.0000	DEF
3.0000	GHI
5.0000	MNO

下面的语句则将保留 Table1 和 Table2 中的所有行，组合后的结果集如表 8-4 所示。

```
SELECT a, b FROM table1
UNION ALL
SELECT b, a FROM table2;
```

表 8-4 Table1 表和 Table2 表使用 UNION ALL 组合后的结果集

A	B
1.0000	ABC
2.0000	DEF
3.0000	GHI
1.0000	JKL
2.0000	DEF
5.0000	MNO

8.1.2 使用 ORDER BY 子句对合并结果集排序

UNION 的结果集列名与 UNION 运算符中第 1 个 SELECT 语句的结果集中的列名相同，则另一个 SELECT 语句的结果集列名将被忽略。因此，在执行 ORDER BY 等操作时必须使用第 1 个结果集中的列名称。如：

```
SELECT a, b FROM table1
UNION
SELECT b, a FROM table2
ORDER BY a;
```

在使用 UNION 运算符的情况下，只能在最后一个 SELECT 语句的后面使用一个 ORDER BY 或 COMPUTE 子句，这个 ORDER BY 子句实际上是应用于最终结果集的。例如，上面的 ORDER BY 子句是对 table1 和 table2 合并后结果集进行的排序，而不是对 table2 进行的排序。

由于 ORDER BY 子句是集合操作的最终结果所支持的唯一逻辑处理阶段，因此要应用其他逻辑处理阶段，只能在 ORDER BY 子句之前进行。这些其他逻辑处理阶段被默认为应用于各自的 SELECT 语句，而不是应用于合并后的结果集。例如，下面语句中的第 1 个 WHERE 应用于 table1，第 2 个 WHERE 应用于 table2，而 ORDER BY 则是应用于合并后的结果集。

```
SELECT a, b FROM table1
WHERE b = 'ABC'
UNION
SELECT b, a FROM table2
WHERE a = 'DEF'
ORDER BY a;
```

8.1.3 结果集的合并顺序

默认情况下是从左向右对包含 UNION 运算符的语句求值。例如，下面的语句首先对

table1 和 table2 进行完全合并,然后将得到的结果集再与 table3 进行合并,并删除它们之间的重复行。

```
SELECT * FROM table1
UNION ALL
SELECT * FROM table2
UNION
SELECT * FROM table3
```

如果希望改变结果集的合并顺序,一种方法是改变 SELECT 语句的书写顺序,另一种方法则是使用圆括号指定。例如,下面两个语句是等价的。第一条语句使用圆括号指定先将 table2 和 table3 合并,然后再将得到的结果集与 table1 合并;第二条语句则是改变了 SELECT 语句的书写顺序。

```
SELECT * FROM table1
UNION ALL
(SELECT * FROM table2
UNION
SELECT * FROM table3);

SELECT * FROM table2
UNION
SELECT * FROM table3
UNION ALL
SELECT * FROM table1;
```

8.2 查询结果集的差异行

从 SQL Server 2005 开始支持的 EXCEPT 运算符,可以返回由运算符左侧的查询返回,而又不包含在右侧查询中的行,并剔除其中的重复行。但是,某些时候这种剔除重复行操作也并不是一种好的方法,因为它隐藏了两个结果集的差异程度。不幸的是,并没有像 EXCEPT ALL 这样的关键词来保留全部差异行。

8.2.1 使用 EXCEPT 运算符

与 UNION 运算符一样,EXCEPT 运算符也要求比较的结果集必须具有相同的结构,并且相应的结果集列的数据类型必须兼容。

首先使用下面的语句创建 TableA、TableB 和 TableC 三个结构相同的示例表,表的内容如表 8-5 所示。

```
CREATE TABLE TableA (col1 int);
CREATE TABLE TableB (col1 int);
CREATE TABLE TableC (col1 int);

INSERT INTO TableA VALUES (NULL),(NULL),(NULL),(1),(2),(2),(2),(3),(4),(4);
INSERT INTO TableB VALUES (NULL),(1),(3),(4),(4),(5);
INSERT INTO TableC VALUES (2),(2),(4),(4);
```

表 8-5 TableA、TableB 和 TableC 表的内容

TableA (col1 列, int)	TableB (col1 列, int)	TableC (col1 列, int)
NULL	NULL	2
NULL	1	2
NULL	3	4
1	4	4
2	4	
2	5	
2		
3		
4		
4		

下面的语句将返回位于 TableA 但是不位于 TableB 中的行，分析这两个表可以看出，TableA 中只有 coll 为 2 的行在 TableB 中不存在，返回结果如表 8-6 所示。

```
SELECT * FROM TableA
EXCEPT
SELECT * FROM TableB;
```

表 8-6 返回结果

coll
2

在这里需要注意一下 NULL 值的问题，在 EXCEPT 运算符中，两个 NULL 被认为是相等的。在 6.2.3 小节中介绍了“三值逻辑”，NULL 值通过任何比较运算符与任何值（包括 NULL 值）进行比较时都是 UNKNOWN，返回 TRUE 的唯一搜索条件是 IS NULL 谓词。在 SQL 中，只有筛选条件计算为 TRUE 时才选择行，不选择值为 UNKNOWN 或 FALSE 的行。从理论上讲，下面的 NOT EXISTS 查询应当等价于上面的 EXCEPT 操作。但是由于 NULL 值的比较问题，在 NOT EXISTS 中认为两个 NULL 是不相等的，所以二者的返回结果并不相同。下面语句的返回结果如表 8-7 所示。

```
SELECT DISTINCT *
FROM TableA
WHERE NOT EXISTS
  (SELECT *
   FROM TableB
   WHERE coll = TableA.coll);
```

表 8-7 NOT EXISTS 的返回结果

coll
NULL
2

8.2.2 查询全部差异行

EXCEPT 虽然可以返回左侧查询与右侧查询的差异行,但是它无法反映这种差异程度。例如,下面的两个语句的返回结果相同(见表 8-6),无论是 TableA 与 TableB 比较,还是 TableC 与 TableB 比较,剔除重复行后,只返回 1 行 coll 为 2 的行。实际上,TableA 与 TableB 相比含有 3 行 coll 为 2 的行,还有 2 行 coll 为 NULL 的行,而 TableC 与 TableB 相比仅含有 2 行 coll 为 2 的行。

```
SELECT * FROM TableA
EXCEPT
SELECT * FROM TableB;

SELECT * FROM TableC
EXCEPT
SELECT * FROM TableB;
```

也就是说,全部差异行不仅需要考虑是否存在的问题,还应当考虑左侧查询和右侧查询中重复行的次数问题。例如,像 TableA 中 3 行 coll 为 2 的行,这些行在 TableB 中不存在,应当包含在差异行中;像 coll 为 NULL 的行,在 TableA 中出现 3 次,在 TableB 中出现 1 次,则 TableA 与 TableB 差异的是 2 行 coll 为 NULL 的行。TableA 与 TableB 的全部差异行应当如表 8-8 所示。

表 8-8

TableA 与 TableB 的全部差异行

coll
NULL
NULL
2
2
2

要返回这样的结果,可以利用从 SQL Server 2005 开始支持的排名函数 ROW_NUMBER() 来解决这个问题。例如,下面的两行语句分别对 TableA 和 TableB 中的行进行排名编号,排名依据是 coll 列,编号方式是根据 coll 进行分区编号,如第 1 个 NULL 编号为 1,第 2 个 NULL 编号为 2,依次类推,当遇到第 1 个不为 NULL 的值时,编号重新从 1 开始。下面两行查询语句的结果分别如表 8-9 和表 8-10 所示。

```
SELECT ROW_NUMBER() OVER(PARTITION BY coll ORDER BY coll) AS rn,
       coll
FROM TableA;

SELECT ROW_NUMBER() OVER(PARTITION BY coll ORDER BY coll) AS rn,
       coll
FROM TableB;
```

表 8-9

对 TableA 进行编号的结果

m	coll
1	NULL
2	NULL
3	NULL
1	1
1	2
2	2
3	2
1	3
1	4
2	4

表 8-10

对 TableB 进行编号的结果

m	coll
1	NULL
1	1
1	3
1	4
2	4
1	5

分析表 8-9 和表 8-10 可以看出，要得到如表 8-8 所示的数据，直接对这两个查询结果执行 EXCEPT 就可以。表 8-9 中的第 1 行由于和表 8-10 中的第 1 行相同，会被剔除掉，而表 8-9 中的第 2 行和第 3 行由于在表 8-10 中没有重复行，就会被保留。依次类推，进行逐行比较。以下是完整的查询语句：

```
SELECT coll
FROM (SELECT ROW_NUMBER() OVER(PARTITION BY coll ORDER BY coll) AS rn, coll
      FROM TableA
      EXCEPT
      SELECT ROW_NUMBER() OVER(PARTITION BY coll ORDER BY coll) AS rn, coll
      FROM TableB
      ) AS tmpTable;
```

8.3

查询结果集的相同行

从 SQL Server 2005 开始支持的 INTERSECT 运算符可以返回运算符左侧查询和右侧查询中都存在的行，并剔除其中的重复行。与 EXCEPT 运算符相同，也没有像 INTERSECT ALL 这样的关键词来保留全部相同行。本节仍然使用 8.2 节所创建 TableA、TableB 和 TableC 表进行示例演示。

8.3.1 使用 INTERSECT 运算符

INTERSECT 运算符也要求比较的结果集必须具有相同的结构，并且相应的结果集列的数据类型必须兼容。下面的语句将返回 TableA 和 TableB 中都存在的行，并剔除重复行，返回结果如表 8-11 所示。

```
SELECT * FROM TableA
INTERSECT
SELECT * FROM TableB;
```

表 8-11

返回结果

coll
NULL
1
3
4

在 INTERSECT 运算符中，两个 NULL 也被认为是相等的。由于 TableA 和 TableB 中都存在 NULL 行，所以被包含在了如表 8-11 所示的返回结果中。

不存在 NULL 值比较的情况下，下面的查询语句等价于上面的 INTERSECT 运算符操作。由于在比较运算符中认为 NULL 值的比较是 UNKNOWN，因此下面查询语句的返回结果不包含表 8-11 中所示的 NULL 行。

```
SELECT DISTINCT *
FROM TableA
WHERE EXISTS
  (SELECT *
   FROM TableB
   WHERE coll = TableA.coll);
```

8.3.2 查询全部相同行

与 EXCEPT 相同，也没有像 INTERSECT ALL 这样的关键词来保留全部相同行，要实现这样的功能，我们只能另辟蹊径。

假设现在要获取 TableA 与 TableB 的全部相同行，分析这两个表可以看出，查询结果应当如表 8-12 所示。coll 为 NULL 的行在 TableA 中虽然有 3 行，但是 TableB 中只有 1 行，所以它们之间相同的只有 1 行。也就是说，如果一个行在左侧中出现了 i 次，在右侧中出现了 j 次，则它应当在结果中出现 $\text{MIN}(i, j)$ 次。

表 8-12

TableA 与 TableB 的全部相同行

coll
NULL
1
3
4
4

要返回上面的结果，我们仍旧需要使用从 SQL Server 2005 开始支持的排名函数 ROW_NUMBER()。例如，下面的查询语句与 8.2.2 小节查询全部差异行的语句基本相同，只是将其中的 EXCEPT 操作符替换成了 INTERSECT。它也是分别对 TableA 和 TableB 中的行进行排名编号，排名依据是 coll 列，编号方式是根据 coll 进行分区编号，编号结果见前面的表 8-9 和表 8-10。然后再对这两个表的行进行 INTERSECT 操作。

```
SELECT coll
FROM (SELECT ROW_NUMBER() OVER(PARTITION BY coll ORDER BY coll) AS rn, coll
      FROM TableA
      INTERSECT
      SELECT ROW_NUMBER() OVER(PARTITION BY coll ORDER BY coll) AS rn, coll
      FROM TableB
      )AS tmpTable;
```

8.4 UNION、EXCEPT 和 INTERSECT 的执行顺序

在 8.1.3 小节中介绍了结果集的合并顺序，默认情况下是从左向右对包含 UNION 运算符的语句求值，但是可以使用圆括号改变执行顺序。在同时包含 UNION、EXCEPT 和 INTERSECT 的语句中，INTERSECT 运算符的优先级要高于 UNION 和 EXCEPT，UNION 和 EXCEPT 级别相同。也就是说，在同时包含这 3 个运算符的语句中，最先执行的是 INTERSECT，然后再按自左至右的顺序执行其他运算符。自然，如果语句中包含圆括号，则先执行圆括号内的语句。

例如，下面的查询使用了 3 个运算符：

```
SELECT * FROM TableB
UNION
SELECT * FROM TableA
EXCEPT
SELECT * FROM TableB
INTERSECT
SELECT * FROM TableC;
```

上面的语句首先对 TableB 和 TableC 执行 INTERSECT，得到这两个表交集，并剔除重复行，结果如表 8-13 所示。

表 8-13

TableB 与 TableC 执行 INTERSECT 后的内容

coll
4

然后再按自左至右的顺序将 TableB 与 TableA 执行 UNION，得到这两个表的合并结果并剔除重复行，如表 8-14 所示。

表 8-14

TableB 与 TableA 执行 UNION 后的内容

coll
NULL
1
2
3
4
5

最后将表 8-13 与表 8-14 所示的内容执行 EXCEPT，得到如表 8-15 所示的最终查询结果。

表 8-15

最终查询结果

coll
NULL
1
2
3
5

也可以使用圆括号来改变这种计算顺序。例如，下面的语句首先将 TableA 与 TableB 执行 EXCEPT，得到中间查询结果后，由于 INTERSECT 的优先级高于 UNION，则先将中间查询结果与 TableC 执行 INTERSECT，最后再与 TableB 执行 UNION。从语句的执行计划，也可以看出这种执行顺序，如图 8-1 所示。

```
SELECT * FROM TableB
UNION
(SELECT * FROM TableA
EXCEPT
SELECT * FROM TableB)
INTERSECT
SELECT * FROM TableC;
```

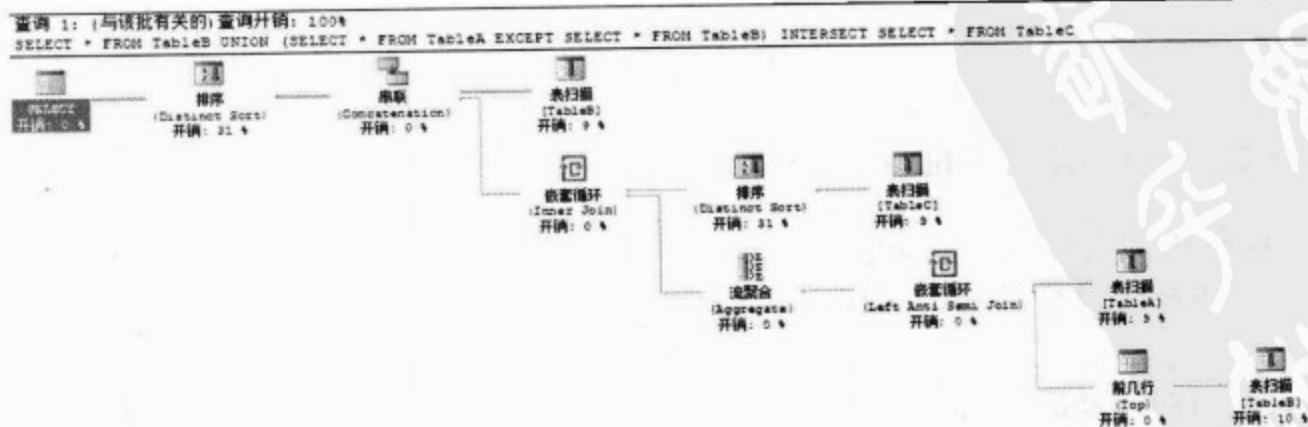


图 8-1 语句的执行计划

8.5 在其他语句中使用 UNION、EXCEPT 和 INTERSECT

8.1.2 小节讨论了在包含 UNION 运算符的查询语句中使用 ORDER BY 子句的限制，即只能在最后一个 SELECT 语句的后面使用一个 ORDER BY 或 COMPUTE 子句，这个 ORDER BY 子句实际上是应用于最终结果集的，用户不能为要合并的结果集单独指定 ORDER BY 子句。此外，当与其他 SQL 语句一起使用 UNION、EXCEPT 和 INTERSECT 时，还有其他一些要遵循的原则，包括：

- 第 1 个查询可以包含一个 INTO 子句，用来创建容纳最终结果集的表。如果在其他查询中出现 INTO 子句，将显示错误消息；
- GROUP BY 和 HAVING 子句只能在各个查询中使用，它们不能用于影响最终结果集；
- FOR BROWSE 子句不能在包含 UNION、EXCEPT 和 INTERSECT 运算符的语句中使用。

8.5.1 使用 INTO 子句指定结果存储位置

要想在 SELECT INTO 语句中使用结果集操作得到的数据，应当在第 1 个查询的 FROM 子句之前指定 INTO 子句。例如，下面的语句将结果集操作的结果存储到临时表 #T1 中，并读取 #T1 中的数据。

```
SELECT * INTO #T1 FROM TableA
EXCEPT
SELECT * FROM TableB
INTERSECT
SELECT * FROM TableC;

SELECT * FROM #T1;
```

8.5.2 突破结果集操作的限制

前面介绍的结果集操作应遵循的各个原则，给结果集数据的使用带来了很多的限制。许多时候，又确实存在着对结果集操作中的单个表进行排序、对结果集进行分组计算等方面的需求。要实现这样的功能，可以通过临时表、派生表、公用表表达式（CTE）过渡一下，从而摆脱这些限制。由于 CTE 还没有讲述，这里尽量使用派生表的方式进行举例说明，在性能方面二者没有差异。示例表仍旧使用在 8.2.1 小节创建的 TableA、TableB 和 TableC 表。

例如，下面的语句通过派生表 tmpTable 解决了不能对最终结果集使用 GROUP BY 子句的限制。将 TableA 与 TableB 完全合并后，计算 coll 中每项的数目，返回结果如表 8-16 所示。

```
SELECT coll, COUNT(*) AS cn
FROM (SELECT * FROM TableA
      UNION ALL
      SELECT * FROM TableB) AS tmpTable
```

GROUP BY coll;

表 8-16 对最终结果集进行分组统计得到的结果

coll	cn
NULL	4
1	2
2	3
3	2
4	4
5	1

如果不使用派生表, 类似下面的查询语句只能对 TableB 进行分组统计, 然后再将得到的结果与 TableA 进行合并。它并不是对 TableA 与 TableB 合并后的结果进行分组统计。

```
SELECT coll, 1 AS cn FROM TableA
UNION ALL
SELECT coll, COUNT(*) AS cn FROM TableB
GROUP BY coll;
```

又如, 下面的语句通过派生表和添加一个辅助列 OrderCol, 实现了对 TableA 按升序排序、对 TableB 按降序排序, 查询结果如表 8-17 所示。

```
SELECT *
FROM (SELECT 1 AS OrderCol, coll
      FROM TableA
      UNION
      SELECT 2 AS OrderCol, coll
      FROM TableB) AS tmpTable
ORDER BY OrderCol,
CASE WHEN OrderCol = 1 THEN coll END,
CASE WHEN OrderCol = 2 THEN coll END DESC;
```

表 8-17 对 TableA 按升序排序、对 TableB 按降序排序后的合并结果

OrderCol	coll
1	NULL
1	1
1	2
1	3
1	4
2	5
2	4
2	3
2	1
2	NULL

8.6 使用公用表表达式

公用表表达式 (Common Table Expression, CTE) 是从 SQL Server 2005 开始支持的一种表达式, 它是一种临时结果集, 与派生表类似, 仅在查询期间有效。与派生表不同的是, CTE 可以调用自身, 从而实现递归。此外, 还可在同一查询中引用多次。

8.6.1 CTE 的语法结构

CTE 由表示 CTE 的名称、可选列列表和定义 CTE 的查询组成。定义 CTE 后, 可以在 SELECT、INSERT、UPDATE 或 DELETE 语句中对其进行引用, 就像引用表或视图一样。CTE 也可用于 CREATE VIEW 语句, 作为定义 SELECT 语句的一部分。

1. CTE 的结构定义

CTE 的基本语法格式如下:

```
WITH cte_name [ ( column_name [...n] ) ]
AS
( CTE_query_definition )
```

cte_name 指定要定义的 CTE 表达式的名称。**column_name** 是 CTE 结果集中列的名称。**CTE_query_definition** 是创建临时结果集的查询语句。

下面的示例显示了 CTE 结构的组件: CTE 的名称 (SalesCTE)、列列表和查询。SalesCTE 包含由 OrderYear 和 SaleTotal 两个列。返回的数据如表 8-18 所示。

```
USE AdventureWorks;
GO

WITH SalesCTE (OrderYear, SaleTotal) -- 定义 CTE 表达式的名称和列
AS
(
    SELECT YEAR(OrderDate), SUM(SubTotal) -- 定义 CTE 查询语句
    FROM Sales.SalesOrderHeader
    GROUP BY YEAR(OrderDate)
)
SELECT * FROM SalesCTE
ORDER BY OrderYear; -- 引用 CTE
```

表 8-18

查询结果

OrderYear	SaleTotal
2001	12966110.5617
2002	36086429.8349
2003	49147162.705
2004	29137477.011

CTE 开头部分对列名称的定义也可以在 CTE 查询语句中使用列别名的方式定义，如：

```
WITH SalesCTE
AS
(
    SELECT YEAR(OrderDate) AS OrderYear, -- 使用别名方式定义 CTE 的列名称
           SUM(SubTotal) AS SaleTotal
    FROM Sales.SalesOrderHeader
    GROUP BY YEAR(OrderDate)
)
SELECT * FROM SalesCTE
ORDER BY OrderYear;
```

对于非计算列，列名称的定义可以完全省略，参考下面的语句。但是对于计算列，如果既未在 CTE 开头部分指定列名称，也未在 CTE 查询部分为计算列指定别名，将引发错误。

```
WITH SalesCTE
AS
(
    SELECT *
    FROM Sales.SalesOrderHeader
)
SELECT * FROM SalesCTE;
```

2. 创建和使用公用表表达式的准则

CTE 之后必须跟随引用部分或单条 SELECT、INSERT、UPDATE 或 DELETE 语句，CTE 仅对该语句可见，并可以引用多次，但是对同一批中的其他语句不可见。例如，下面的第二行查询语句会引发错误。

```
WITH SalesCTE
AS
(
    SELECT YEAR(OrderDate) AS OrderYear,
           SUM(SubTotal) AS SaleTotal
    FROM Sales.SalesOrderHeader
    GROUP BY YEAR(OrderDate)
)
SELECT * FROM SalesCTE -- 自身外部查询
ORDER BY OrderYear;

SELECT * FROM SalesCTE -- SalesCTE 对于该语句不可见，该语句无法执行
WHERE OrderYear = 2003;
```

可以在非递归 CTE 中定义多个查询语句，但是，这些查询定义必须用一个集合运算符 UNION ALL、UNION、EXCEPT 或 INTERSECT 联接起来。

不允许在一个 CTE 中再嵌套定义 CTE。例如，像下面这样在 CTE_query_definition 中包含一个子查询，而该子查询又定义了一个 CTE，是不允许的。

```
WITH CTE1
AS
(
    SELECT *
    FROM (
        WITH CTE2
```

```

        AS
        (
            SELECT *
            FROM Sales.SalesOrderHeader
        )
        SELECT * FROM CTE2
    )
SELECT * FROM CTE1;

```

不能在 CTE_query_definition 中使用以下子句:

- COMPUTE 或 COMPUTE BY;
- ORDER BY (除非指定了 TOP 子句);
- INTO;
- 带有查询提示的 OPTION 子句;
- FOR XML;
- FOR BROWSE。

例如, 下面的 CTE 定义中由于包含了 ORDER BY 子句, 将引发错误。

```

WITH SalesCTE
AS
(
    SELECT YEAR(OrderDate) AS OrderYear,
           SUM(SubTotal) AS SaleTotal
    FROM Sales.SalesOrderHeader
    GROUP BY YEAR(OrderDate)
    ORDER BY OrderYear -- 在 CTE 定义中指定了 ORDER BY
)
SELECT * FROM SalesCTE;

```

而下面的 CTE 定义中由于同时包含了 TOP 子句, 则可以正常执行。

```

WITH SalesCTE
AS
(
    SELECT TOP 2
           YEAR(OrderDate) AS OrderYear,
           SUM(SubTotal) AS SaleTotal
    FROM Sales.SalesOrderHeader
    GROUP BY YEAR(OrderDate)
    ORDER BY OrderYear
)
SELECT * FROM SalesCTE; -- 引用 CTE

```

8.6.2 多 CTE 定义和 CTE 的多次引用

1. 多 CTE 定义

CTE 虽然不能嵌套, 但是可以在同一 WITH 子句中同时定义多个 CTE。CTE 定义之间使用逗号分隔。在 CTE 中可以引用预先定义的 CTE, 但不允许前向引用。

例如, 下面的语句定义了两个 CTE, 分别是 SalesCTE1 和 SalesCTE2。SalesCTE1 返回客户 ID、订单的年份和金额小计。在 SalesCTE2 中引用了 SalesCTE1, 根据客户 ID 和订单年份分组汇总计算销售额。得到的查询结果如表 8-19 所示。

```
WITH SalesCTE1 AS
(
    SELECT CustomerID,
           YEAR(OrderDate) AS OrderYear,
           SubTotal
    FROM Sales.SalesOrderHeader
),
SalesCTE2 AS
(
    SELECT CustomerID, OrderYear,
           SUM(SubTotal) AS SaleTotal
    FROM SalesCTE1
    GROUP BY CustomerID, OrderYear
)
SELECT *
FROM SalesCTE2
ORDER BY CustomerID, OrderYear;
```

表 8-19 通过两个 CTE 得到的分组汇总结果

CustomerID	OrderYear	SaleTotal
1	2001	36862.0876
1	2002	65489.709
2	2002	14166.2185
2	2003	10966.5406
2	2004	4490.7426
3	2001	35975.4227
3	2002	152392.4906
3	2003	198583.1915
3	2004	46991.2714

2. CTE 的多次引用

CTE 定义后, 可以在其后面跟随的查询语句中多次引用, 这是与派生表最大的区别。例如, 下面的语句使用派生表的方式, 计算每年销售额与上一年的差别。其中需要两次打开 Sales.SalesOrderHeader 表, 分别作为派生表 Cur 和 Prv。查询结果如表 8-20 所示。

```
USE AdventureWorks;
GO

SELECT Cur.OrderYear AS CurYear,
       Prv.OrderYear AS PrvYear,
       Cur.SaleTotal AS CurSaleTotal,
```

```

        Cur.SaleTotal - Prv.SaleTotal AS DiffSaleTotal
FROM (SELECT YEAR(OrderDate) AS OrderYear,
        SUM(SubTotal) AS SaleTotal
      FROM Sales.SalesOrderHeader
      GROUP BY YEAR(OrderDate)) AS Cur
LEFT JOIN (SELECT YEAR(OrderDate) AS OrderYear,
        SUM(SubTotal) AS SaleTotal
      FROM Sales.SalesOrderHeader
      GROUP BY YEAR(OrderDate)) AS Prv
  ON Cur.OrderYear = Prv.OrderYear + 1
ORDER BY CurYear;

```

表 8-20 计算与上一年的销售额差异

CurYear	PrvYear	CurSaleTotal	DiffSaleTotal
2001	NULL	12966110.5617	NULL
2002	2001	36086429.8349	23120319.2732
2003	2002	49147162.705	13060732.8701
2004	2003	29137477.011	-20009685.694

而下面的语句则是使用 CTE 的方式,同样可以获得表 8-20 所示的数据。SalesCTE 被引用了两次,分别作为 Cur 和 Prv。

```

WITH SalesCTE AS
(
    SELECT YEAR(OrderDate) AS OrderYear,
           SUM(SubTotal) AS SaleTotal
    FROM Sales.SalesOrderHeader
    GROUP BY YEAR(OrderDate)
)
SELECT Cur.OrderYear AS CurYear,
       Prv.OrderYear AS PrvYear,
       Cur.SaleTotal AS CurSaleTotal,
       Cur.SaleTotal - Prv.SaleTotal AS DiffSaleTotal
FROM SalesCTE AS Cur
LEFT JOIN SalesCTE AS Prv
  ON Cur.OrderYear = Prv.OrderYear + 1
ORDER BY CurYear;

```

8.6.3 CTE 的间接嵌套

由于 CTE 不允许嵌套,因此可以通过在视图、用户定义函数(UDF)中定义 CTE 的方式实现间接嵌套。

下面的示例将通过视图的方式介绍间接嵌套的方法,该示例仍旧是计算每年销售额与上一年的差别。首先创建一个名为 `dbo.vSalesYear` 的视图,其中包含了一个 CTE 定义,用于分组统计每年的销售额。

```

USE AdventureWorks;
GO

```



```
CREATE VIEW dbo.vSalesYear
AS
WITH SalesCTE AS
(
    SELECT YEAR(OrderDate) AS OrderYear,
           SUM(SubTotal) AS SaleTotal
    FROM Sales.SalesOrderHeader
    GROUP BY YEAR(OrderDate)
)
SELECT * FROM SalesCTE;
```

而下面的语句则是先创建一个 CTE，用于引用上面所创建的视图 `dbo.vSalesYear` 中的数据，这实际上是实现了 CTE 的间接嵌套（即 `vSalesYearCTE` 嵌套 `SalesCTE`）。CTE 定义后面的查询语句用于计算每年销售额与上一年的差别，查询结果见前面的表 8-20。

```
WITH vSalesYearCTE AS
(
    SELECT OrderYear, SaleTotal
    FROM dbo.vSalesYear
)
SELECT Cur.OrderYear AS CurYear,
       Prv.OrderYear AS PrvYear,
       Cur.SaleTotal AS CurSaleTotal,
       Cur.SaleTotal - Prv.SaleTotal AS DiffSaleTotal
FROM vSalesYearCTE AS Cur
     LEFT JOIN vSalesYearCTE AS Prv
       ON Cur.OrderYear = Prv.OrderYear + 1
ORDER BY CurYear;
```

使用下面的语句删除掉演示用的视图 `dbo.vSalesYear`。

```
DROP VIEW dbo.vSalesYear;
```

8.6.4 使用递归 CTE 返回分层数据

公用表表达式重要的优点就是能够引用自身，从而创建递归 CTE。递归 CTE 是一个重复执行初始 CTE 以返回数据子集，直到获取完整结果集的公用表表达式。

当某个查询引用递归 CTE 时，它即被称为递归查询。递归查询通常用于返回分层数据，例如，显示某个组织图中的雇员或物料清单方案（其中父级产品有一个或多个组件，而那些组件可能还有子组件）中的数据。

递归 CTE 可以极大地简化了在 `SELECT`、`INSERT`、`UPDATE`、`DELETE` 或 `CREATE VIEW` 语句中运行递归查询所需的代码。在 `SQL Server` 的早期版本中，递归查询通常需要使用临时表、游标和逻辑来控制递归步骤流。

递归 CTE 的结构与其他编程语言中的递归例程相似，其他语言中的递归例程返回的是标量值，但递归 CTE 可以返回多行。

1. 递归 CTE 的组成元素

递归 CTE 定义至少必须包含两个 CTE 查询定义，一个定位点成员和一个递归成员。可以定义多个定位点成员和递归成员，但必须将所有定位点成员查询定义置于第 1 个递归成员定义之前。所有 CTE 查询定义都是定位点成员，但它们引用 CTE 本身时除外。

(1) 定位点成员。

递归 CTE 的第 1 个查询可以由一个或多个 UNION ALL、UNION、EXCEPT 或 INTERSECT 运算符联接的查询语句。由于该查询定义形成了 CTE 结构的基准结果集，所以被称为“定位点成员”。

在最后一个定位点成员和第 1 个递归成员之间，以及组合多个递归成员时，只能使用 UNION ALL 集合运算符联接。

(2) 递归成员。

递归 CTE 的第 2 个查询需要引用 CTE 本身，因此被成为“递归成员”。只能引用一次 CTE 本身。递归成员通过 UNION ALL 与定位点成员联接。

由于是结果集操作，所以定位点成员和递归成员中的列数必须一致，并且递归成员中列的数据类型必须与定位点成员中相应列的数据类型一致。

在递归成员的查询定义中不允许出现下列项：

- SELECT DISTINCT;
- GROUP BY;
- HAVING;
- 标量聚合;
- TOP;
- LEFT、RIGHT、OUTER JOIN (允许出现 INNER JOIN);
- 子查询;
- 应用于对 CTE 的递归引用的提示。

(3) 终止检查。

递归调用通常都应当具有一个终止条件。在递归 CTE 调用中，终止检查是隐式的，当上一个调用中未返回行时，递归将停止。

2. 递归 CTE 示例

假设有一个如图 8-2 所示的公司组织结构，其中雇员 A1 是 B1 和 B2 的主管，而 B1 又是 C1 和 C2 的主管，B2 是 C3 的主管。这些数据存储在一个如表 8-21 所示的 Employees 表中，其中 Salary

列存放的是雇员的工资。

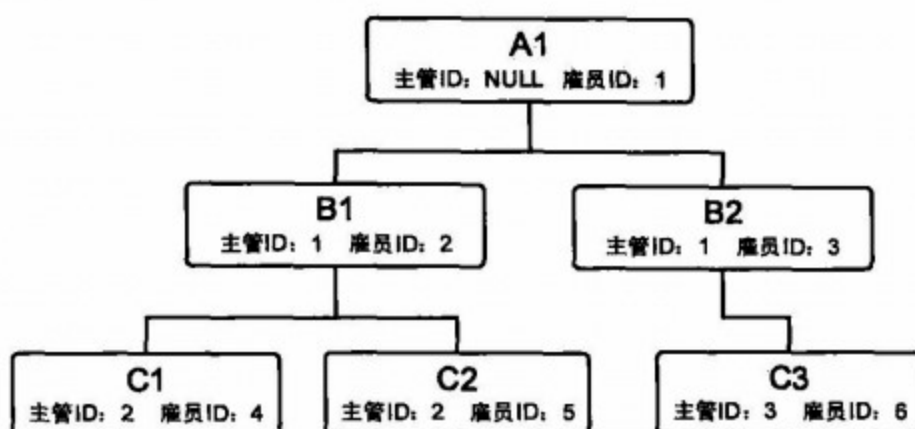


图 8-2 公司雇员的组织结构

表 8-21

Employees 表

EmployeeID	ManagerID	EmployeeName	Salary
1	NULL	A1	10000.00
2	1	B1	9000.00
3	1	B2	9000.00
4	2	C1	8000.00
5	2	C2	8000.00
6	3	C3	8000.00

(1) 显示递归的级别。

下面的示例显示经理以及向经理报告的雇员的层次列表，其中 CTE 定义中包含了一个定位点成员和一个递归成员。

```

WITH DirectReports AS
(
  -- 定位点成员定义
  SELECT ManagerID, EmployeeID, EmployeeName, 0 AS Level
  FROM Employees
  WHERE ManagerID IS NULL
  UNION ALL
  -- 递归成员定义
  SELECT e.ManagerID, e.EmployeeID, e.EmployeeName, Level + 1
  FROM Employees AS e
  INNER JOIN DirectReports AS d
    ON e.ManagerID = d.EmployeeID
)
-- 执行 CTE 的语句
SELECT * FROM DirectReports;
  
```

在执行递归 CTE 时，将按照下面的步骤进行。

- 将 CTE 表达式拆分为定位点成员和递归成员。
- 运行定位点成员，将创建如表 8-22 所示基准结果集 (T0)。ManagerID 为空，表示这是公

司的最高级雇员。

表 8-22

基准结果集

ManagerID	EmployeeID	EmployeeName	Level
NULL	1	A1	0

● 递归成员返回定位点成员结果集中的雇员的直接下属。这是通过在 Employees 表和 DirectReports 之间执行联接操作获得的，正是此次对 CTE 自身的引用建立了递归调用。利用 DirectReports 中的雇员作为输入(T_i)，通过联接($Employees.ManagerID = DirectReports.EmployeeID$)条件，返回主管为 T_i 的雇员作为输出(T_{i+1})。这样，递归成员的第 1 次迭代返回了如表 8-23 所示的结果集。

表 8-23

第 1 次迭代返回的结果集

ManagerID	EmployeeID	EmployeeName	Level
1	2	B1	1
1	3	B2	1

● 重复激活递归成员。递归成员的第 2 次迭代使用步骤 c 中的结果集作为输入值，并返回如表 8-24 所示的结果集。

表 8-24

第 2 次迭代返回的结果集

ManagerID	EmployeeID	EmployeeName	Level
3	6	C3	2
2	4	C1	2
2	5	C2	2

重复此过程，直到递归成员返回一个空结果集。实际上，在执行第 3 次迭代时，使用步骤 d 中的结果集作为输入值，返回的已经是空结果集。

● 返回的最终结果集是定位点成员和递归成员生成的所有结果集的并集。表 8-25 是该示例返回的完整结果集。

表 8-25

完整结果集

ManagerID	EmployeeID	EmployeeName	Level
NULL	1	A1	0
1	2	B1	1
1	3	B2	1
3	6	C3	2
2	4	C1	2
2	5	C2	2

可以通过在 CTE 执行语句中添加 WHERE 子句来限制返回的级别数目，如下面的语句将只返

回如表 8-25 所示的前 3 行。

```
WITH DirectReports AS
(
    SELECT ManagerID, EmployeeID, EmployeeName, 0 AS Level
    FROM Employees
    WHERE ManagerID IS NULL
    UNION ALL
    SELECT e.ManagerID, e.EmployeeID, e.EmployeeName, Level + 1
    FROM Employees AS e
    INNER JOIN DirectReports AS d
        ON e.ManagerID = d.EmployeeID
)
SELECT * FROM DirectReports
WHERE Level < 2 ;
```

也可以在 CTE 的定位点成员定义中使用 **WHERE** 子句，返回指定雇员的下属。例如，下面的语句指定返回 **EmployeeID = 2** 雇员的下属。

```
WITH DirectReports AS
(
    SELECT ManagerID, EmployeeID, EmployeeName, 0 AS Level
    FROM Employees
    WHERE EmployeeID = 2
    UNION ALL
    SELECT e.ManagerID, e.EmployeeID, e.EmployeeName, Level + 1
    FROM Employees AS e
    INNER JOIN DirectReports AS d
        ON e.ManagerID = d.EmployeeID
)
SELECT * FROM DirectReports;
```

(2) 显示雇员的层次列表。

下面的示例通过缩进各个级别，突出显示经理和雇员的层次结构。查询结果如表 8-26 所示。

```
WITH DirectReports (ManagerID, EmployeeID, EmployeeName, Level, Sort)
AS
(
    SELECT ManagerID, EmployeeID,
        CONVERT(varchar(255), EmployeeName),
        0,
        CONVERT(varchar(255), EmployeeName)
    FROM Employees
    WHERE ManagerID IS NULL
    UNION ALL
    SELECT e.ManagerID, e.EmployeeID,
        CONVERT(varchar(255), REPLICATE('|', Level + 1) + e.EmployeeName),
        Level + 1,
        CONVERT(varchar(255), RTRIM(Sort) + '|' + e.EmployeeName)
    FROM Employees AS e
    INNER JOIN DirectReports AS d
        ON e.ManagerID = d.EmployeeID
)
SELECT * FROM DirectReports
ORDER BY Sort;
```

表 8-26

雇员的层次列表

ManagerID	EmployeeID	EmployeeName	Level	Sort
NULL	1	A1	0	A1
1	2	B1	1	A1 B1
2	4	C1	2	A1 B1 C1
2	5	C2	2	A1 B1 C2
1	3	B2	1	A1 B2
3	6	C3	2	A1 B2 C3

(3) 使用 MAXRECURSION 提示限制递归调用次数。

可以使用 MAXRECURSION 来限制递归调用的次数，这样做的目的是防止出现不合理的递归 CTE 进入无限循环。MAXRECURSION 的默认设置为 100，如果想移除该限制，可以设置 MAXRECURSION 为 0。

例如，下面的语句限制递归调用为 1 次，返回结果为如表 8-25 所示的前 3 行。

```
WITH DirectReports AS
(
    SELECT ManagerID, EmployeeID, EmployeeName, 0 AS Level
    FROM Employees
    WHERE ManagerID IS NULL
    UNION ALL
    SELECT e.ManagerID, e.EmployeeID, e.EmployeeName, Level + 1
    FROM Employees AS e
    INNER JOIN DirectReports AS d
        ON e.ManagerID = d.EmployeeID
)
SELECT * FROM DirectReports
OPTION (MAXRECURSION 1);
```

(4) 在 UPDATE 语句中使用递归 CTE。

由于 CTE 是一种临时结果集，不能通过更新 CTE 的方式来更新其引用的表，CTE 只能作为一种匹配条件的筛选工具。例如，下面的语句首先通过 CTE 检索出 EmployeeID = 2 雇员的下属，然后使用 UPDATE 语句将这些员工的工资增加 50%。

```
WITH DirectReports AS
(
    SELECT ManagerID, EmployeeID, EmployeeName, 0 AS Level
    FROM Employees
    WHERE EmployeeID = 2
    UNION ALL
    SELECT e.ManagerID, e.EmployeeID, e.EmployeeName, Level + 1
    FROM Employees AS e
    INNER JOIN DirectReports AS d
        ON e.ManagerID = d.EmployeeID
)
UPDATE Employees
SET Salary = Salary * 1.5
WHERE EmployeeID IN (SELECT EmployeeID FROM DirectReports);
```

(5) 使用多个定位点和递归成员。

具有多个定位点的 CTE, 定位点之间应当使用 UNION ALL、UNION、EXCEPT 或 INTERSECT 运算符联接。

首先使用下面的语句创建一个示例表 **Person**, 其中存放着每个人的父母的姓名。表的内容如表 8-27 所示。

```
CREATE TABLE Person(ID int, Name varchar(30), Mother int, Father int);
GO
INSERT Person
VALUES(1, 'Sue', NULL, NULL),
      (2, 'Ed', NULL, NULL),
      (3, 'Emma', 1, 2),
      (4, 'Jack', 1, 2),
      (5, 'Jane', NULL, NULL),
      (6, 'Bonnie', 5, 4),
      (7, 'Bill', 5, 4);
```

表 8-27

Person 表中的内容

ID	Name	Mother	Father
1	Sue	NULL	NULL
2	Ed	NULL	NULL
3	Emma	1	2
4	Jack	1	2
5	Jane	NULL	NULL
6	Bonnie	5	4
7	Bill	5	4

下面的语句将使用多个定位点和递归成员来返回指定的人的所有祖先, 查询结果如表 8-28 所示。

```
WITH Generation (ID)
AS
(
  -- 第 1 个定位点成员返回 Bonnie 的母亲
  SELECT Mother
  FROM Person
  WHERE Name = 'Bonnie'
  UNION
  -- 第 2 个定位点成员返回 Bonnie 的母亲
  SELECT Father
  FROM Person
  WHERE Name = 'Bonnie'
  UNION ALL
  -- 第 1 个递归成员返回前面生成的人员的男性祖先
  SELECT Person.Father
  FROM Generation, Person
  WHERE Generation.ID=Person.ID
  UNION ALL
  -- 第 2 个递归成员返回前面生成的人员的女性祖先
  SELECT Person.Mother
  FROM Generation, Person
  WHERE Generation.ID=Person.ID
```

```

)
SELECT Person.ID, Person.Name, Person.Mother, Person.Father
FROM Generation, Person
WHERE Generation.ID = Person.ID;

```

表 8-28

Bonnie 的所有祖先

ID	Name	Mother	Father
1	Sue	NULL	NULL
2	Ed	NULL	NULL
4	Jack	1	2
5	Jane	NULL	NULL

8.7 汇总数据

虽然进行分组统计汇总可以通过 GROUP BY 子句结合聚合函数实现，但是，当需要统计分析的列非常多时，单纯使用聚合函数可能会非常繁琐。例如，假设有一个如表 8-29 所示的 Inventory 表，需要统计桌子的数量、椅子的数量、蓝色的数量、红色的数量、蓝色桌子的数量、红色桌子的数量、蓝色椅子的数量、红色椅子的数量、总的货物数量，这需要进行 9 次聚合计算。

多次聚合计算不仅增加了往返服务器的次数，同时也增加了服务器的负担和网络流量。为此，GROUP BY 子句增加了 WITH CUBE 和 WITH ROLLUP 参数，通过提交一条查询语句，便可以获得所有可能的分组统计结果。而在 SQL Server 2008 中，GROUPING SETS、ROLLUP 和 CUBE 运算符已被添加到 GROUP BY 子句中，不再推荐使用不符合 ISO 规范的 WITH ROLLUP、WITH CUBE 语法。

表 8-29

Inventory 表

Item	Color	Quantity
桌子	蓝色	2
桌子	蓝色	1
桌子	红色	3
椅子	红色	1
椅子	蓝色	6

8.7.1 使用 CUBE 汇总数据

CUBE 运算符生成的结果集是基于用户要分析的列建立的，这些列称为维度。因此，这个结果集也称为多维数据集，其中包含各维度的所有可能组合的交叉表格。

CUBE 运算符和维度列在 SELECT 语句的 GROUP BY 子句中指定。结果集中包含维度列中各值的所有可能组合，以及与这些维度值组合相匹配的基础行中的聚合值。

例如，下面的语句将返回一个结果集，其中包含 Item 和 Color 的所有可能组合的 Quantity 列之和，如表 8-30 所示。为便于说明，在表格中加入了一个行号列。

```
SELECT Item, Color, SUM(Quantity) AS QtySum
FROM Inventory
GROUP BY Item, Color WITH CUBE;
```

表 8-30 CUBE 汇总结果

行号	Item	Color	QtySum
1	椅子	红色	1
2	桌子	红色	3
3	NULL	红色	4
4	椅子	蓝色	6
5	桌子	蓝色	3
6	NULL	蓝色	9
7	NULL	NULL	13
8	椅子	NULL	7
9	桌子	NULL	6

下面分析一下结果集中几个特殊行的含义：

- 第 3 行和第 6 行报告了 Color 维度的小计。两行中的 Item 维度值都是 NULL，表示聚合数据来自 Item 维度为任意值的行。
- 第 7 行报告了多维数据集的总计。Item 和 Color 维度都包含 NULL 值，表示此行中汇总了这两个维度的所有值。
- 第 8 行报告的是 Item 维度中包含“椅子”值的所有行的小计。对 Color 维度返回了 NULL 值，表示该行报告的聚合包括 Color 维度为任意值的行。
- 第 9 行报告的是 Item 维度中包含“桌子”值的所有行的小计。
- 其他各行返回的是基于 Item+Color 维度的小计，如第 4 行是表 8-29 中前两行的加总。

在 SQL Server 2008 中，应当使用下面符合 ISO 规范的语法格式。

```
SELECT Item, Color, SUM(Quantity) AS QtySum
FROM Inventory
GROUP BY CUBE(Item, Color);
```

8.7.2 使用 ROLLUP 汇总数据

ROLLUP 运算符生成的结果集与 CUBE 运算符生成的结果集类似。二者之间的区别是：CUBE 生成的结果集是所选列中值的所有组合的聚合，而 ROLLUP 一般用于层次结构中，它生成的结果集是所选列中值的当前粒度级别及其以下级别的聚合。例如，仍旧以表 8-29 所示的 Inventory 表为

例，下面的查询将返回总计、基于 Item 维度的小计，以及其以下级别 Item+Color 维度的小计。但是它不会像 CUBE 那样返回 Color 维度的小计。查询结果如表 8-31 所示。

```
SELECT Item, Color, SUM(Quantity) AS QtySum
FROM Inventory
GROUP BY Item, Color WITH ROLLUP;
```

表 8-31 ROLLUP 汇总结果

Item	Color	QtySum
椅子	红色	1
椅子	蓝色	6
椅子	NULL	7
桌子	红色	3
桌子	蓝色	3
桌子	NULL	6
NULL	NULL	13

由表 8-31 可以看出，ROLLUP 生成的汇总结果缺少了表 8-30 中的第 3 行和第 6 行。

在 SQL Server 2008 中，上面的语句应当写成如下符合 ISO 规范的格式。

```
SELECT Item, Color, SUM(Quantity) AS QtySum
FROM Inventory
GROUP BY ROLLUP(Item, Color);
```

8.7.3 区分空值和汇总值

在前面，无论是 CUBE 操作还是 ROLLUP 操作，我们都是以列中所包含的 NULL 值来判断所生成是否是汇总数据，已经所汇总的层次。如果实际数据中包含 NULL 值，我们就无法判断它是由 CUBE 操作，或是 ROLLUP 操作生成的 NULL 值，还是实际数据中返回的 NULL 值，也就无法区分出哪一行是汇总行。

可以使用 GROUPING 函数来解决这个问题，如果列值来自实际数据，GROUPING 函数将返回 0；如果列值是由 CUBE 或 ROLLUP 操作生成的 NULL，则返回 1。例如，下面的语句将由 CUBE 生成 NULL 替换成了字符串“汇总”，事实数据中的 NULL 将被替换为字符串“未知”。生成的结果集如表 8-32 所示。

```
SELECT CASE WHEN (GROUPING(Item) = 1) THEN '汇总'
            ELSE ISNULL(Item, '未知')
        END AS Item,
        CASE WHEN (GROUPING(Color) = 1) THEN '汇总'
            ELSE ISNULL(Color, '未知')
        END AS Color,
        SUM(Quantity) AS QtySum
FROM Inventory
GROUP BY Item, Color WITH CUBE;
```

表 8-32 结合 GROUPING 函数得到的 CUBE 汇总结果

Item	Color	QtySum
椅子	红色	1
桌子	红色	3
汇总	红色	4
椅子	蓝色	6
桌子	蓝色	3
汇总	蓝色	9
汇总	汇总	13
椅子	汇总	7
桌子	汇总	6

8.7.4 返回指定维度的汇总

CUBE 和 ROLLUP 简化了多维数据汇总的问题，但是这两个运算符返回的汇总数据是非常多的，很多时候可能并不需要某些汇总结果。要返回指定维度的汇总，一种方法是使用派生表结合 WHERE 子句的方式进行筛选，另一种方式是使用 GROUPING SETS 运算符，这是 SQL Server 2008 提供的新功能。

1. 使用派生表

在只有一个维度的情况下，会只生成汇总数据，并且这时候 CUBE 和 ROLLUP 运算符的返回结果是相同的。例如，下面的两个查询语句的返回结果相同，既包括了 Item 中每个值的小计，也显示了 Item 中所有值的总计，如表 8-33 所示。

```

SELECT CASE WHEN (GROUPING(Item) = 1) THEN '汇总'
           ELSE ISNULL(Item, '未知')
           END AS Item,
       SUM(Quantity) AS QtySum
FROM Inventory
GROUP BY Item WITH CUBE;

SELECT CASE WHEN (GROUPING(Item) = 1) THEN '汇总'
           ELSE ISNULL(Item, '未知')
           END AS Item,
       SUM(Quantity) AS QtySum
FROM Inventory
GROUP BY Item WITH ROLLUP;

```

表 8-33 一个维度下生成的汇总结果

Item	QtySum
椅子	7
桌子	6
汇总	13

在多个维度的情况下，可以使用派生表结合 WHERE 子句的方式获取指定的汇总结果。例如，下面的语句指定仅返回椅子的汇总结果，如表 8-34 所示。

```
SELECT *
FROM (SELECT Item, Color, SUM(Quantity) AS QtySum
      FROM Inventory
      GROUP BY Item, Color WITH CUBE) AS tmpCUBE
WHERE Item='椅子' AND Color IS NULL;
```

表 8-34 获取指定的汇总结果

Item	Color	QtySum
椅子	NULL	7

2. 使用 GROUPING SETS

如果不需要获得由 ROLLUP 或 CUBE 运算符生成的全部分组，则可以使用 GROUPING SETS 指定仅获取所需的分组。例如，下面的语句将得到分别按 Item 和 Color 分组汇总结果集的并集，如表 8-35 所示。

```
SELECT Item, Color, SUM(Quantity) AS QtySum
FROM Inventory
GROUP BY GROUPING SETS(Item, Color);
```

表 8-35 使用 GROUPING SETS 得到的汇总结果

Item	Color	QtySum
NULL	红色	4
NULL	蓝色	9
椅子	NULL	7
桌子	NULL	6

上面的语句等同于下面的 UNION ALL 语句。

```
SELECT Item, NULL AS Color, SUM(Quantity) AS QtySum
FROM Inventory
GROUP BY Item
UNION ALL
SELECT NULL AS Item, Color, SUM(Quantity) AS QtySum
FROM Inventory
GROUP BY Color;
```





第9章 窗口计算和表旋转

可以说,许多新技术都是围绕着实际业务应用而诞生的,窗口计算和表旋转就是这样的技术。在设计表结构时,很多情况下都是以方便数据存储为目的,这导致了从业务应用角度直接阅读这些表数据比较困难,而这两项技术则有效解决了表的数据存储和实际应用之间的矛盾问题。前几章已经介绍了 SQL 查询,数据的检索范围都是针对一个表或是几个表进行的,而窗口计算却是对表内每组数据的检索和计算。表旋转则是将表行转换为列,或是将列转换为行。

9.1 窗口和开窗函数

窗口是指为用户指定的一组行。在如表 9-1 所示的 Students 表中,包含有 3 个班级的学生成绩,则其中的每一个班级都可以被看作是一个数据窗口,或是分区。

表 9-1 Students 表中的窗口

ClassID	StudentName	Achievement	
1	Grace	99.00	} 窗口
1	Andrew	99.00	
1	Janet	75.00	
1	Margaret	89.00	
2	Steven	86.00	
2	Michael	72.00	
2	Robert	91.00	
3	Laura	75.00	
3	Ann	94.00	
3	Ina	80.00	
3	Ken	92.00	

之所以要提出窗口这个概念,因为这种基于窗口或分区的重新计算在实际工作应用范围比较广泛。例如,假设我们要对每个班级中的学生按成绩进行排序,在对第 1 个班级排序完成后,对第 2 个班级进行排序时编号需要重新从 1 开始。在 SQL Server 2005 之前,像这种排序方式实现起来是比较烦琐的。可以说,对新窗口重新启动计算是窗口计算的重要特点。

为支持窗口计算,SQL Server 提供了 OVER 子句和窗口函数。窗口函数在 MSDN Library 中被

翻译为开窗函数。虽然“开窗函数”理解起来并不如“窗口函数”容易，但是它描述了数据窗口变化后重新启动计算这样一个动作，所以我们尊重 MSDN Library 中的翻译，在后续的介绍中将使用“开窗函数”这一名词。

窗口计算的两个主要应用就是对每组内的数据进行排序和聚合计算。因此，开窗函数也被分为排名开窗函数和聚合开窗函数。排名开窗函数如 ROW_NUMBER()、RANK()，聚合开窗函数如 AVG()、SUM 等。

9.2 基于窗口的排名计算

进行排名计算时，OVER 子句的语法格式如下：

```
OVER ( [ PARTITION BY value_expression , ... [ n ] ]
      <ORDER BY Clause> )
PARTITION BY value_expression
```

指定对相应 FROM 子句生成的行集进行分区所依据的列。开窗函数分别应用于每个分区，并为每个分区重新启动计算。value_expression 只能引用通过 FROM 子句可用的列，不能引用选择列表中的表达式或别名。value_expression 可以是列表达式、标量子查询、标量函数或用户定义的变量。

┃ <ORDER BY 子句>

指定应用排名开窗函数的排序顺序。只能引用通过 FROM 子句可用的列，但是不同通过指定整数来表示选择列表中列名称或列别名的位置。

下面将以表 9-1 所示的 Students 表为例，进行介绍。像 Students 表这样的数据结构设计，相对于数据库存储而言是比较合理的，因为我们不可能为每个班级创建一个表，但确实又存在像为每个班级中的学生成绩进行排序或为学生编号这样的实际需求，窗口计算技术就有效解决了二者之间的矛盾。

排名函数包括 ROW_NUMBER()、RANK()、DENSE_RANK() 和 NTILE()，它们可以为分区中的每一行返回一个排名值。ROW_NUMBER() 用于按行进行编号，RANK() 和 DENSE_RANK() 用于按指定顺序排名，NTILE() 用于对数据进行分区。

9.2.1 使用 ROW_NUMBER() 实现分区编号

ROW_NUMBER() 返回分区内行的序列号，每个分区的第一行从 1 开始。例如，下面的语句指定按 ClassID 进行分区，并按 StudentName 进行排序编号。查询结果如表 9-2 所示。

```
SELECT ClassID, StudentName, Achievement,
       ROW_NUMBER() OVER(PARTITION BY ClassID ORDER BY StudentName) AS RowNumber
FROM Students;
```


表 9-2 按班级分区、按学生姓名进行编号

ClassID	StudentName	Achievement	RowNumber
1	Andrew	99.00	1
1	Grace	99.00	2
1	Janet	75.00	3
1	Margaret	89.00	4
2	Michael	72.00	1
2	Robert	91.00	2
2	Steven	86.00	3
3	Ann	94.00	1
3	Ina	80.00	2
3	Ken	92.00	3
3	Laura	75.00	4

为了理解排名函数的工作原理，来看一下查询优化器为查询生成的执行计划，如图 9-1 所示。

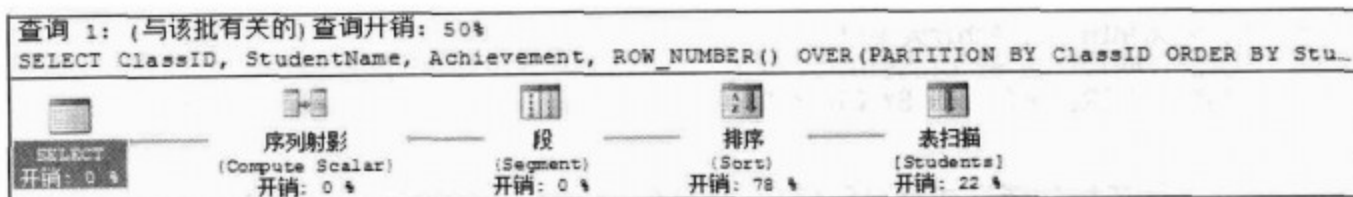


图 9-1 为 ROW_NUMBER() 生成的执行计划

由图 9-1 可以看出，为了计算排名，优化器首先按分区列排序，然后再对分区内行按 ORDER BY 子句指定的列排序。如果事先为表创建了符合该排序条件的索引，则会直接扫描该索引文件，不再进行排序。

“序列射影”运算符的工作是负责计算排名，“段”运算符用于确定分组边界。二者相互协调工作，来确定每一行的排名值。

“段”运算符在内存中会保留一行，用来与下一行的 PARTITION BY 列值进行比较。对于表中的第一行，“段”运算符自然会发送 true 信号。对于后面的行，直到 PARTITION BY 列值有变化之前，会一直发送 false 信号。如果 PARTITION BY 列值发生了变化，说明已经到了下一个分区，“段”运算符会再次发送 true 信号。“序列射影”运算符在接收到 true 信号后，会重置排名值。

如果“序列射影”运算符接收到的是 false 信号，它会确认当前输入行的排序值是否不同于上一行，如果不同，则按排名函数所指示的递增排名值。自然，在该示例中，由于 ROW_NUMBER() 函数需要为每一行递增值。因此，这个排序值比较步骤在该示例中是不存在的。但是，对于像 RANK() 和 DENSE_RANK() 函数，在执行计划中还会有另外一个“段”运算符，用于比较排序值是否有变化，以确定是否递增排名值。此问题在后面的章节中还会详细讨论。

9.2.2 使用 RANK() 和 DENSE_RANK() 函数实现分区排名

ROW_NUMBER() 函数用于编号，它与排名具有不同的概念。例如，由表 9-1 可以看出，班级 1 中的 Grace 和 Andrew 的成绩相同，都是 99 分。如果使用 ROW_NUMBER() 函数编号，有两种编号方案可供选择：一种是 Grace 第 1、Andrew 第 2，另一种是 Andrew 第 1、Grace 第 2。这虽然都是正确的，但它具有不确定性。

而排名则不同了，它具有确定性，相同的排序值总是被分配相同的排名值。Grace 和 Andrew 在排名的情况下都应当是第 1，也就是我们常说的并列第 1。那他们两人之后的名次是什么呢？是第 2 还是第 3 呢？从两人并列第 1 的角度讲，他们两人之后的名次应当是第 2，这也是 DENSE_RANK() 函数的排名方式；前面已经有 2 个人 99 分了，他们后面的人应当是第 3 个高分者，从这个角度理解，后面的名次应当是第 3，这也是 RANK() 的排名方式。DENSE_RANK() 函数的排名方式称之为密集排名，因为它的名次之间没有间隔。

下面的语句演示了 RANK() 和 DENSE_RANK() 的排名方式，查询结果如表 9-3 所示。

```
SELECT ClassID, StudentName, Achievement,
       RANK() OVER(PARTITION BY ClassID ORDER BY Achievement DESC) AS SortRank,
       DENSE_RANK() OVER(PARTITION BY ClassID ORDER BY Achievement DESC) AS SortDense
FROM Students;
```

表 9-3 按班级和考试成绩分别使用 RANK() 和 DENSE_RANK() 排名

ClassID	StudentName	Achievement	SortRank	SortDense
1	Grace	99.00	1	1
1	Andrew	99.00	1	1
1	Margaret	89.00	3	2
1	Janet	75.00	4	3
2	Robert	91.00	1	1
2	Steven	86.00	2	2
2	Michael	72.00	3	3
3	Ann	94.00	1	1
3	Ken	92.00	2	2
3	Ina	80.00	3	3
3	Laura	75.00	4	4

下面是为语句生成的执行计划，与 ROW_NUMBER() 相比，执行计划中多出了一个“段”运算符。右边段的分组依据是 ClassID，左边段的分组依据是 ClassID 和 Achievement，这是多出的“段”。右边的“段”用于分区操作，在到达下一个分区时发送 true 信号，“序列射影”运算符会重置排名值。而左边的“段”用于比较排序值是否有变化，如果有变化，则通知“序列射影”运算符递增排名值，递增方式则按 RANK() 和 DENSE_RANK() 函数的规则进行。

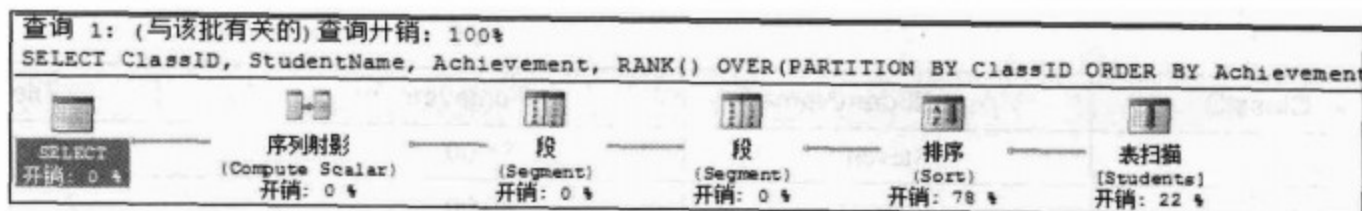


图 9-2 为 RANK() 和 DENSE_RANK() 生成的执行计划

在 SQL Server 2005 之前, 也可以使用子查询的方式实现排名计算。语句的原理就是查询出比当前成绩高的个数, 再加上 1, 就是该成绩的排名。例如, 在第 1 个班级中, 比 99 分高的成绩为 0, 加上 1 后, 该成绩就是第 1 名。下面语句的执行结果与表 9-3 相同, 但是由于对于每个成绩都要执行两次子查询, 在性能方面与 RANK() 和 DENSE_RANK() 函数相差很远。

```
SELECT ClassID, StudentName, Achievement,
       (SELECT COUNT(*) FROM Students AS S2
        WHERE S2.ClassID = S1.ClassID AND S2.Achievement > S1.Achievement)+1 AS SortRank,
       (SELECT COUNT(DISTINCT achievement) FROM Students AS S2
        WHERE S2.ClassID = S1.ClassID AND S2.Achievement > S1.Achievement)+1 AS SortDense
FROM Students AS S1
ORDER BY ClassID, Achievement DESC;
```

9.2.3 使用 NTILE() 函数实现数据分组

NTILE() 函数用于把行分发到指定数目的组中。各个组有编号, 编号从 1 开始。对于每一个行, NTILE 将返回此行所属的组的编号。

NTILE() 函数可以接受一个代表组数量的参数, 分组的方式“均分”原则。例如, 假设一个表有 10 行, 需要分成 2 组, 则每个组都会有 5 行。如果表有 11 行, 需要分成 3 个组, 这时候是无法均分的。它分配方法是先得到一个能够整除的基组大小 ($11/3=3$), 每组应当分配 3 行, 剩余的 2 行 ($11-9$) 会被再次均分到前面的 2 组中。

例如, 下面的语句指定将 Students 表按学生成绩划分为 3 个组, 并且 Students 表恰好也是 11 行, 分组结果如表 9-4 所示。

```
SELECT ClassID, StudentName, Achievement,
       NTILE(3) OVER(ORDER BY Achievement DESC) AS Tile
FROM Students;
```

表 9-4 分组结果

ClassID	StudentName	Achievement	Tile
1	Grace	99.00	1
1	Andrew	99.00	1
3	Ann	94.00	1
3	Ken	92.00	1
2	Robert	91.00	2
1	Margaret	89.00	2

续表

ClassID	StudentName	Achievement	Tile
2	Steven	86.00	2
3	Ina	80.00	2
3	Laura	75.00	3
1	Janet	75.00	3
2	Michael	72.00	3

也可以先分区,再分组。例如,以下语句将每个班级的成绩划分为高、低两组,查询结果如表 9-5 所示。可以看出,包含 4 名学生的班级,每组是 2 人;包含 3 名学生的班级,第 1 组是 2 人,第 2 组是 1 人。

```
SELECT ClassID, StudentName, Achievement,
       CASE NTILE(2) OVER(PARTITION BY ClassID ORDER BY Achievement DESC)
         WHEN 1 THEN '高'
         WHEN 2 THEN '低'
       END AS Tile
FROM Students;
```

表 9-5

按班级分区再按成绩分组结果

ClassID	StudentName	Achievement	Tile
1	Grace	99.00	高
1	Andrew	99.00	高
1	Margaret	89.00	低
1	Janet	75.00	低
2	Robert	91.00	高
2	Steven	86.00	高
2	Michael	72.00	低
3	Ann	94.00	高
3	Ken	92.00	高
3	Ina	80.00	低
3	Laura	75.00	低

9.3 基于窗口的聚合计算

在进行聚合计算时, OVER 子句中不再需要 ORDER BY 子句。因此,其语法简化成如下格式:

```
OVER ( [ PARTITION BY value_expression , ... [ n ] ]
```


9.3.1 分区聚合计算与联接的比较

通过 OVER 子句，可以对每个分区内的数据进行聚合计算。仍旧使用如表 9-1 所示的 Students 表的数据，现假设需要计算每名学生成绩与本班级平均成绩的差异。在之前，我们需要先计算每个班级的平均成绩，然后通过联接的方式将平均成绩关联到相应的学生成绩行，再计算差异。例如：

```
SELECT S1.ClassID,
       S1.StudentName,
       S1.Achievement,
       S2.AvgAch,
       S1.Achievement - S2.AvgAch AS Diff
FROM Students AS S1
LEFT OUTER JOIN (SELECT ClassID, AVG(Achievement) AS AvgAch
                  FROM Students
                  GROUP BY ClassID) AS S2  --计算每个班级的平均成绩
ON S1.ClassID = S2.ClassID;
```

查询结果如表 9-6 所示。

表 9-6 查询每名学生成绩与本班级平均成绩的差异

ClassID	StudentName	Achievement	AvgAch	Diff
1	Grace	99.00	90.500000	8.500000
1	Andrew	99.00	90.500000	8.500000
1	Janet	75.00	90.500000	-15.500000
1	Margaret	89.00	90.500000	-1.500000
2	Steven	86.00	83.000000	3.000000
2	Michael	72.00	83.000000	-11.000000
2	Robert	91.00	83.000000	8.000000
3	Laura	75.00	85.250000	-10.250000
3	Ann	94.00	85.250000	8.750000
3	Ina	80.00	85.250000	-5.250000
3	Ken	92.00	85.250000	6.750000

在使用 OVER 子句的情况下，查询语句会简洁许多，下面语句的查询结果与表 9-6 相同。

```
SELECT ClassID,
       StudentName,
       Achievement,
       AVG(Achievement) OVER(PARTITION BY ClassID) AS AvgAch,
       Achievement - AVG(Achievement) OVER(PARTITION BY ClassID) AS Diff
FROM Students;
```

虽然语句有所简洁，但是在性能方面该语句不如上面的联接方式。查询优化器为该语句生成的查询计划比较复杂，与联接语句在同一个批中执行时，含有 OVER 子句的查询开销占了 66%，如图 9-3 所示。

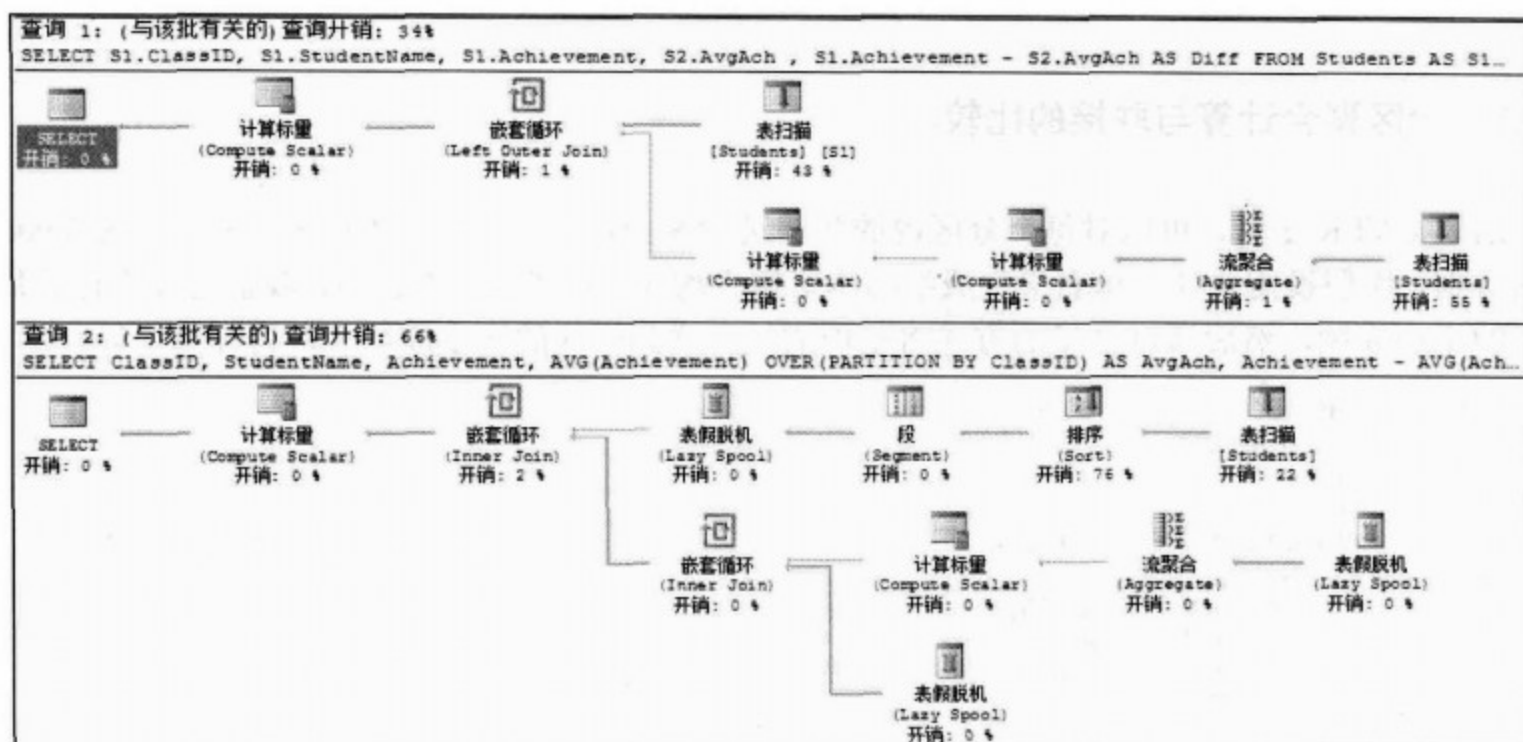


图 9-3 联接方式与 OVER 子句的性能比较

9.3.2 对不同类型分区的聚合计算

当在一个语句中需要计算多个不同类型的分区聚合时，**OVER** 子句有着更明显的优势。例如，假设我们既要计算与本班级平均成绩的差异，又要计算与全部学生平均成绩的差异，含有 **OVER** 子句的查询变化不大，而使用联接方式则需要增加一个联接。参考下面的语句：

```
SELECT ClassID,
       StudentName,
       Achievement,
       AVG(Achievement) OVER(PARTITION BY ClassID) AS AvgAch,
       Achievement - AVG(Achievement) OVER(PARTITION BY ClassID) AS Diff,
       AVG(Achievement) OVER() AS AvgAllAch,      -- 所有学生的平均成绩
       Achievement - AVG(Achievement) OVER() AS DiffAll
FROM Students;

SELECT S1.ClassID,
       S1.StudentName,
       S1.Achievement,
       S2.AvgAch,
       S1.Achievement - S2.AvgAch AS Diff,
       S3.AvgAllAch,
       S1.Achievement - S3.AvgAllAch AS DiffAll
FROM Students AS S1
  LEFT OUTER JOIN (SELECT ClassID, AVG(Achievement) AS AvgAch
                  FROM Students
                  GROUP BY ClassID) AS S2
    ON S1.ClassID = S2.ClassID
  CROSS JOIN (SELECT AVG(Achievement) AS AvgAllAch
              FROM Students) AS S3:      -- 增加了一个联接
```

对于语句中所包含的 **OVER** 子句数量，对于查询的影响不大。例如，下面的第一条语句仅含

有1个 OVER 子句，而第二条语句则含有4个 OVER 子句，但是优化器为它们生成的执行计划完全相同。

```
SELECT ClassID,
       StudentName,
       Achievement,
       AVG(Achievement) OVER(PARTITION BY ClassID) AS AvgAch
FROM Students;

SELECT ClassID,
       StudentName,
       Achievement,
       AVG(Achievement) OVER(PARTITION BY ClassID) AS AvgAch,
       SUM(Achievement) OVER(PARTITION BY ClassID) AS SumAch,
       MIN(Achievement) OVER(PARTITION BY ClassID) AS MinAch,
       MAX(Achievement) OVER(PARTITION BY ClassID) AS MaxAch
FROM Students;
```

9.4 表旋转

所谓表旋转，就是将表的行转换为列，或是将表的列转换为行，这是从 SQL Server 2005 开始提供的新技术。因此，如果希望使用此功能，需要将数据库的兼容级别设置为 90。表旋转在某些方面也是解决了表的数据存储和实际需要之间的矛盾。例如，图 9-4 是一个典型的产品销售统计表，这种格式虽然便于阅读，但是在进行数据表存储时却并不容易管理，产品销售数据表通常需要设计成如图 9-5 所示的结构。这样就带来一个问题，用户既希望数据容易管理，又希望能够生成一种能够容易阅读的表格数据，这时就可以使用表旋转技术。

销量	1月	2月	3月	4月	...
产品					
产品1					
产品2					
产品3					
...					

图 9-4 产品销售表

产品	销售日期	销售额
产品1		
产品2		
产品3		
...		

图 9-5 数据表结构

9.4.1 使用 PIVOT 运算符将表的行转换为列

PIVOT 运算符用于将表的行转换为列，并能同时对行执行聚合运算，其语法格式如下：

```
SELECT <非旋转列>,
       [第 1 个旋转列] AS <列名>,
       [第 2 个旋转列] AS <列名>,
       ...
       [最后的旋转列] AS <列名>
FROM
    (<SELECT 生成数据的查询>) AS <为源查询结果指定的别名>

PIVOT
(
    <聚合函数>(<被聚合的列>)
```

```

FOR
[<包含将被转换为列标头的值的列>]
  IN ( [第 1 个旋转后的列], [第 2 个旋转后的列],
    ... [最后一个旋转后的列] )
) AS <为旋转表指定的别名>

<可选的 ORDER BY 子句>;

```

为了实现行的旋转，源查询获得的结果应当具备 3 列，才能够实现旋转。第 1 列是不进行旋转的列，属于标志列；第 2 列是属性列，也称为透视列，其中的值会被旋转列名；第 3 列是属性值列，这些值将作为新列的值。使用下面的语句创建一个示例表 Orders，内容如表 9-7 所示。

```

CREATE TABLE Orders
(
    ProductID int NOT NULL,
    OrderDate datetime NOT NULL,
    ShipTo char(20) NOT NULL,
    SubTotal money NOT NULL
);
INSERT INTO Orders
VALUES (1, CAST('20090102' AS datetime), 'Shanghai', 100.00),
(1, CAST('20090105' AS datetime), 'Shanghai', 100.00),
(1, CAST('20090123' AS datetime), 'Jinan', 100.00),
(2, CAST('20090125' AS datetime), 'Shanghai', 100.00),
(1, CAST('20090205' AS datetime), 'Jinan', 100.00),
(3, CAST('20090213' AS datetime), 'Shanghai', 100.00),
(3, CAST('20090219' AS datetime), 'Shanghai', 100.00),
(4, CAST('20090309' AS datetime), 'Beijing', 100.00),
(1, CAST('20090311' AS datetime), 'Dalian', 100.00),
(2, CAST('20090324' AS datetime), 'Shanghai', 100.00),
(3, CAST('20090326' AS datetime), 'Wuhan', 100.00);

```

表 9-7

Orders 表的内容

ProductID	OrderDate	ShipTo	SubTotal
1	2009-01-02 00:00:00.000	Shanghai	100.00
1	2009-01-05 00:00:00.000	Shanghai	100.00
1	2009-01-23 00:00:00.000	Jinan	100.00
2	2009-01-25 00:00:00.000	Shanghai	100.00
1	2009-02-05 00:00:00.000	Jinan	100.00
3	2009-02-13 00:00:00.000	Shanghai	100.00
3	2009-02-19 00:00:00.000	Shanghai	100.00
4	2009-03-09 00:00:00.000	Beijing	100.00
1	2009-03-11 00:00:00.000	Dalian	100.00
2	2009-03-24 00:00:00.000	Shanghai	100.00
3	2009-03-26 00:00:00.000	Wuhan	100.00

Orders 表中包含了 3 个月的产品销售数据，现在假设要获得如图 9-4 所示的销售表，则对源表的查询首先需要获得上面讲的 3 列，参考以下语句：

```

SELECT ProductID,
    MONTH(OrderDate) AS OrderMonth,

```

```
SubTotal
FROM Orders;
```

查询结果如表 9-8 所示。其中 ProductID 为标志列，OrderMonth 为属性列，其中的月份要转变为列的名称，SubTotal 为属性值列，这些值将成为新列的值。

表 9-8 获取到的 3 列内容

ProductID	OrderMonth	SubTotal
1	1	100.00
1	1	100.00
1	1	100.00
2	1	100.00
1	2	100.00
3	2	100.00
3	2	100.00
4	3	100.00
1	3	100.00
2	3	100.00
3	3	100.00

完整的旋转查询语句如下，查询结果如表 9-9 所示。

```
SELECT ProductID,
       [1] AS Jan,
       [2] AS Feb,
       [3] AS Mar
FROM (SELECT ProductID, MONTH(OrderDate) AS OrderMonth, SubTotal
      FROM Orders) AS O1
PIVOT
(
    SUM(SubTotal)
    FOR OrderMonth IN ([1], [2], [3])
) AS Pvt
ORDER BY ProductID;
```

表 9-9 旋转后输出的内容

ProductID	Jan	Feb	Mar
1	300.00	100.00	100.00
2	100.00	NULL	100.00
3	NULL	200.00	100.00
4	NULL	NULL	100.00

上面的查询语句将按以下步骤来获取如表 9-9 所示的结果集。

● PIVOT 首先按属性值列之外的列（ProductID 和 OrderMonth）对输入表 Sales.Orders 进行分组汇总，类似执行下面的语句，得到一个如表 9-10 所示的中间结果集。


```
SELECT ProductID,
       OrderMonth,
       SUM(SubTotal) AS SubTotal
FROM (SELECT ProductID, MONTH(OrderDate) AS OrderMonth, SubTotal
      FROM Orders) AS O1
GROUP BY ProductID, OrderMonth;
```

表 9-10 Orders 经分组汇总后的结果

ProductID	OrderMonth	SubTotal
1	1	300.00
1	2	100.00
1	3	100.00
2	1	100.00
2	3	100.00
3	2	200.00
3	3	100.00
4	3	100.00

● PIVOT 根据 FOR OrderMonth IN 指定的值 1、2、3，首先在结果集中建立名为 1、2、3 的列，然后从如表 9-10 所示的 SubTotal 列中取出相符合的值，分别放置到 1、2、3 列中。此时得到的结果集的别名为 pvt（见语句中 AS pvt 的指定）。结果集的内容如表 9-11 所示。

表 9-11 使用 FOR OrderMonth IN ([1], [2], [3])后得到的结果集

ProductID	1	2	3
1	300.00	100.00	100.00
2	100.00	NULL	100.00
3	NULL	200.00	100.00
4	NULL	NULL	100.00

● 最后根据 SELECT ProductID, [1] AS Jan, [2] AS Feb, [3] AS Mar FROM 的指定，从别名 pvt 结果集中检索数据，并分别将名为 1、2、3 的列在最终结果集中重新命名为 Jan、Feb、Mar，得到如表 9-9 所示的结果集。这里需要注意的是 FROM 的含义，其表示从经 PIVOT 关系运算符得到的 pvt 结果集中检索数据，而不是从 Orders 或派生表 O1 中检索数据。

在 SQL Server 2005 之前，要进行行列转换比较繁琐，用户需要考虑源表中行与结果集中行的关系，属性列中的每个唯一值在结果集中都需要一个列。像表 9-7 中的 Orders 表由于包含 3 个月份的数据，因此在 SELECT 列表中需要包含 3 个表达式，分别用于提取 3 个月份中的数据。下面语句的查询结果与表 9-9 相同，请读者自行分析以下语句。

```
SELECT ProductID,
       SUM(CASE WHEN OrderMonth = 1 THEN SubTotal END) AS Jan,
       SUM(CASE WHEN OrderMonth = 2 THEN SubTotal END) AS Feb,
       SUM(CASE WHEN OrderMonth = 3 THEN SubTotal END) AS Mar
FROM (SELECT ProductID,
       MONTH(OrderDate) AS OrderMonth,
```



```

        SubTotal AS SubTotal
    FROM Orders) AS O1
GROUP BY ProductID;

```

9.4.2 使用 UNPIVOT 运算符将表的列转换为行

UNPIVOT 与 PIVOT 执行几乎完全相反的操作，将列转换为行。但是，UNPIVOT 并不完全是 PIVOT 的逆操作，由于在执行 PIVOT 过程中，数据已经被进行了分组汇总，所以使用 UNPIVOT 并不会重现原始表值表达式的结果。假设如表 9-9 所示的结果集存储在一个名为 MyPvt 的表中，现在需要将列 Jan、Feb 和 Mar 转换到对应于相应产品 ID 的行值（即返回到如表 9-10 所示的格式）。这意味着必须另外标识两个列，一个用于存储月份，另一个用于存储销售额。为了便于理解，仍旧分别将这两个列命名为 OrderMonth 和 SubTotal。

下面的语句首先创建 MyPvt 表，然后将查询数据插入到表中。

```

CREATE TABLE MyPvt
(
    ProductID int NOT NULL,
    Jan money,
    Feb money,
    Mar money
);

INSERT INTO MyPvt (ProductID, Jan, Feb, Mar)
SELECT ProductID,
       [1] AS Jan,
       [2] AS Feb,
       [3] AS Mar
FROM (SELECT ProductID, MONTH(OrderDate) AS OrderMonth, SubTotal
      FROM Orders) AS O1
PIVOT
(
    SUM(SubTotal)
    FOR OrderMonth IN ([1], [2], [3])
) AS Pvt
ORDER BY ProductID;

```

下面的语句执行 UNPIVOT，将得到如表 9-12 所示的查询结果。

```

SELECT ProductID, OrderMonth, SubTotal
FROM MyPvt
UNPIVOT
(
    SubTotal FOR OrderMonth IN (Jan, Feb, Mar)
) AS UnPvt;

```

表 9-12

UNPIVOT 得到的查询结果

ProductID	OrderMonth	SubTotal
1	Jan	300.00
1	Feb	100.00
1	Mar	100.00

续表

ProductID	OrderMonth	SubTotal
2	Jan	100.00
2	Mar	100.00
3	Feb	200.00
3	Mar	100.00
4	Mar	100.00

以上语句将按下面的步骤获得输出结果集：

- 首先建立一个临时结果集的结构，该结构中包含 MyPvt 表中除 IN (Jan, Feb, Mar) 之外的列，以及 SubTotal FOR OrderMonth 中指定的属性值列 (SubTotal) 和属性列 (OrderMonth)；
- 然后将在 MyPvt 中逐行检索数据，将表的列名称放入 OrderMonth 列中，将相应的值放入到 SubTotal 列中。

由于在 PIVOT 时为列指定了别名，所以在 UNPIVOT 后，OrderMonth 列中的月份使用的是英文简称，而不是如表 9-10 所示的格式。要得到如表 9-10 所示的格式，可以在查询语句中使用 CASE 表达式来解决这个问题，参考以下语句：

```
SELECT ProductID,
       CAST(CASE
            WHEN OrderMonth = 'Jan' THEN '1'
            WHEN OrderMonth = 'Feb' THEN '2'
            WHEN OrderMonth = 'Mar' THEN '3'
            END AS int) AS OrderMonth,
       SubTotal
FROM MyPvt
UNPIVOT
(
    SubTotal FOR OrderMonth IN (Jan, Feb, Mar)
) AS UnPvt;
```

在 SQL Server 2005 之前，则应当使用以下语句：

```
SELECT * FROM
(
    SELECT ProductID, 1 AS OrderMonth, Jan AS SubTotal
    FROM MyPvt
    UNION ALL
    SELECT ProductID, 2 AS OrderMonth, Feb
    FROM MyPvt
    UNION ALL
    SELECT ProductID, 3 AS OrderMonth, Mar
    FROM MyPvt) AS O
WHERE SubTotal IS NOT NULL;
```



第 10 章 数据修改

数据修改语句包括：INSERT、UPDATE 和 DELETE，用于修改表中的一行或者是一个行集，但不能在一条语句中修改多个表。数据查询的目的通常包括：一是根据检索条件查找特定数据，二是对符合根据检索条件的相应数据进行修改。所以可以为修改语句指定检索条件，检索条件可以是一个 WHERE 子句，也可以是一个比较复杂的联接条件。

10.1 插入数据

为了向表中添加一个或多个新行，可以使用 INSERT 语句。INSERT 语句可以单独使用，也可以与 SELECT 语句结合使用。

10.1.1 使用 INSERT 和 VALUES 插入行

可以使用 INSERT 语句向表中添加数据行，VALUES 关键字为表的某一行或多个行指定值。其语法格式如下：

```
INSERT INTO table_or_view [(column_list)]  
VALUES (value_list1),  
       (value_list2),  
       ...  
       (value_listN)
```

table_or_view 是要添加数据的表或视图的名称。column_list 是要插入列的列表，列名称之间使用逗号分隔。value_list1 是与列对应的值的列表，值之间使用逗号分隔。column_list 可以省略，在省略的情况下，则表示要向整行插入数据，列的顺序是在 CREATE TABLE、CREATE VIEW 或 ALTER VIEW 等 DDL 语句中定义的顺序。

在插入数据时，应当遵循如下的原则：

- 所插入数据的数据类型必须与列的数据类型一致，或是能够转换为列的数据类型。
- 数据的大小应当在数据类型规定的范围之内。
- 在 VALUES 的值列表中，值的位置应当与列的排列位置相同。即第 1 个数据应当对应第一列，第 2 个数据对应第二列。

INSERT 语句不指定下列类型列的值，因为数据库引擎将为这些列生成值：

- 具有 IDENTITY 属性的列，此属性为该列生成值。
- 具有默认值的列，此默认值用 NEWID 函数生成唯一的 GUID 值。
- 计算列。

下面将通过示例的方式来说明 INSERT 语句的使用方法，首先使用下面的语句创建一个名为 Orders 表，该表包含 4 列，每列都具有不同的数据类型。

```
CREATE TABLE Orders
(
    OrderID int NOT NULL PRIMARY KEY,
    OrderDate datetime,
    ShipTo char(20),
    SubTotal money
);
```

下面的语句用于向表中添加一行数据，值的排列顺序要与列的顺序一致，并且数据类型要与列定义的一致。对于像 int、money 这样的数值型列，插入值可以直接放置在 VALUES 列表中，而对于像 datetime 和 char 这样的日期、字符型列，插入值应当放置在单引号之内。

```
INSERT INTO Orders (OrderID, OrderDate, ShipTo, SubTotal)
VALUES(1, '2009-03-09', 'Shanghai', 200.00);
```

上面的语句由于为每列都指定了值，因此也可以省略列列表。如：

```
INSERT INTO Orders
VALUES(1, '2009-03-09', 'Shanghai', 200.00);
```

指定的列列表中，如果没有包含表定义的全部列，在插入数据时，未包含列的值会被赋予 NULL。例如，下面的语句中由于未给 ShipTo 列指定值，因此该列的值为 NULL。

```
INSERT INTO Orders (OrderID, OrderDate, SubTotal)
VALUES(1, '2009-03-09', 200.00);
```

如果需要插入多行数据，在 SQL Server 2008 之前，需要书写多个 INSERT 语句。例如，下面的语句向表中插入了 2 行数据。

```
INSERT INTO Orders
VALUES(1, '2009-03-09', 'Shanghai', 200.00);
INSERT INTO Orders
VALUES(2, '2009-03-09', 'Beijing', 200.00);
```

从 SQL Server 2008 开始，则可以在一个 INSERT 语句中包含多行值列表，每个列表之间使用逗号分隔，一个 INSERT 语句最多可以包含 1000 行。例如，下面的语句向表中插入了 4 行数据。

```
INSERT INTO Orders
VALUES(1, '2009-03-09', 'Shanghai', 200.00),
(2, '2009-03-09', 'Beijing', 200.00),
(3, '2009-03-10', 'Beijing', 100.00),
(4, '2009-03-11', 'Guangzhou', 300.00);
```

10.1.2 使用 INSERT 和 SELECT 子查询插入行

使用 INSERT 语句和 SELECT 子查询可以将查询结果插入到另一个表中，子查询的选择列表必须与 INSERT 语句的列列表匹配。如果没有指定列列表，选择列表必须与正在其中执行插入操作的表或视图的列匹配。

例如，以下语句将 Sales.SalesReason 表中 ReasonType 为 Marketing 的行插入到 MySalesReason 表中。

```
USE AdventureWorks;
GO
CREATE TABLE MySalesReason (
    SalesReasonID int NOT NULL,
    Name nvarchar(50),
    ModifiedDate datetime);
GO
INSERT INTO MySalesReason
    SELECT SalesReasonID, Name, ModifiedDate*
    FROM AdventureWorks.Sales.SalesReason
    WHERE ReasonType = N'Marketing';
GO
SELECT SalesReasonID, Name, ModifiedDate
FROM MySalesReason;
GO
```

10.1.3 使用 INSERT 和 EXEC 插入行

将 INSERT 与 EXEC 语句结合，可以将存储过程或动态 SELECT 语句返回的结果集插入到现有表中。例如，以下语句创建了一个名为 dbo.GetTop 的存储过程，用于根据指定的行数 (@n) 从 Person.Contact 中返回结果集。有可能指定的行数会超出表中的实际行数，因此增加了一个输出参数 @rcc，用于获取所返回结果集的实际行数。

```
CREATE PROC dbo.GetTop
    @n AS INT,
    @rcc AS INT OUTPUT
AS
    SELECT TOP(@n) ContactID, Title, FirstName, LastName, EmailAddress
    FROM Person.Contact;
    SET @rcc = @@ROWCOUNT;
```

以下语句首先创建一个临时表 #T，并声明了一个变量 @rcc，用于接收输出参数的返回值。由存储过程返回的结果集将被插入到 #T 表中，然后显示结果集的实际行数和 #T 中的内容。在该示例中，我们希望返回 20000 行数据，但是 Person.Contact 中只有 19972 行。下面语句的执行结果如图 10-1 所示。

```
IF OBJECT_ID('tempdb..#T') IS NOT NULL
    DROP TABLE #T;
GO
```



```

CREATE TABLE #T
(
    ContactID int,
    Title nvarchar(8),
    FirstName nvarchar(50),
    LastName nvarchar(50),
    EmailAddress nvarchar(50)
);

DECLARE @rcc AS INT;
INSERT INTO #T
EXEC dbo.GetTop @n = 20000, @rcc = @rcc OUTPUT;  -- 将存储过程返回的结果集插入到#T 中

SELECT @rcc;  -- 显示实际返回的行数
SELECT * FROM #T;  -- 显示#T 中的内容

```



	ContactID	Title	FirstName	LastName	EmailAddress
1	1	Mr.	Gustavo	Achong	gustavo0@adventure-works.com
2	2	Ms.	Catherine	Abel	catherine0@adventure-works.com
3	3	Ms.	Kim	Abercrombie	kim2@adventure-works.com
4	4	Sr.	Humberto	Acevedo	humberto0@adventure-works.com
5	5	Sra.	Pilar	Ackerman	pilar1@adventure-works.com
6	6	Ms.	Frances	Adams	frances0@adventure-works.com
7	7	Ms.	Margaret	Smith	margaret0@adventure-works.com
8	8	Ms.	Carla	Adams	carla0@adventure-works.com
9	9	Mr.	Jay	Adams	jay1@adventure-works.com
10	10				

图 10-1 由存储过程返回的结果集

10.1.4 使用 SELECT INTO 插入行

SELECT INTO 语句并不返回结果集，而是创建一个新表，并用 SELECT 语句的结果集填充该表。它可以将几个表或视图中的数据组合成一个表，也可用于创建一个包含选自链接服务器的数据的新表。

新表的结构由选择列表中表达式的属性定义。通常情况下，在进行单表查询时，新表的列将复制源表中列的数据类型、为空性和 IDENTITY 属性。但是，并不会复制源表的约束、索引和触发器。例如，以下语句从 Person.Contact 表中复制 5 列来创建新表 dbo.MyContact。在 Person.Contact 中 ContactID 具有 IDENTITY 属性，因此 dbo.MyContact 中的该列也具有此属性。

```

USE AdventureWorks;
GO
SELECT ContactID, Title, FirstName, LastName, EmailAddress
INTO dbo.MyContact
FROM Person.Contact;

```

当从多个表中选择列来创建表时，IDENTITY 属性不会被复制。例如，下面的语句从多个雇员和与地址相关的表中选择 8 列来创建表 dbo.MyAddresses，但是 ContactID 列并不具有 IDENTITY 属性。

```

USE AdventureWorks;
GO
SELECT c.ContactID, c.FirstName, c.LastName,
       e.Title, a.AddressLine1, a.City,
       sp.Name AS [State/Province], a.PostalCode
INTO dbo.MyAddresses
FROM Person.Contact AS c
JOIN HumanResources.Employee AS e
  ON e.ContactID = c.ContactID
JOIN HumanResources.EmployeeAddress AS ea
  ON ea.EmployeeID = e.EmployeeID
JOIN Person.Address AS a
  ON a.AddressID = ea.AddressID
JOIN Person.StateProvince AS sp
  ON sp.StateProvinceID = a.StateProvinceID;

```

查询为单表查询时，如果不希望向新表列中复制 **IDENTITY** 属性，可以对具有该属性的列使用计算表达式的方法转换一下即可。例如，以下语句使用了 **ContactID+0 AS ContactID** 方法。

```

SELECT ContactID + 0 AS ContactID, Title, FirstName, LastName, EmailAddress
INTO dbo.MyContact
FROM Person.Contact;

```

如果仅希望复制表结构，可以使用下面的方法。

```

SELECT *
INTO dbo.MyContact
FROM Person.Contact
WHERE 1 = 2;

```

不能使用 **SELECT INTO** 创建已分区表，即使源表已进行分区，**SELECT INTO** 也不会使用源表的分区方案，新表是在默认文件组中创建的。

10.2 更新数据

可以使用 **UPDATE** 语句为表或视图中某个或某些行的一列或多列赋予新值，也可以使用该语句更新远程服务器上的行（使用链接服务器名称或 **OPENROWSET**、**OPENDATASOURCE** 和 **OPENQUERY** 函数），前提是用来访问远程服务器的 OLE DB 访问接口支持更新操作。**UPDATE** 语句主要包括 **SET**、**FROM** 和 **WHERE** 子句。

10.2.1 使用 SET 和 WHERE 子句更新数据

SET 子句指定要更改的列和这些列的新值，对所有符合 **WHERE** 子句搜索条件的行，将使用 **SET** 子句中指定的值更新指定列中的值。

下面是一个最基本的更新语句，用于将 **Person.Address** 中城市名称为 **Bothell** 的行的邮政编码更改为 **98000**。

```

USE AdventureWorks;
GO

```

```
UPDATE Person.Address
SET PostalCode = '98000'
WHERE City = 'Bothell';
```

如果没有指定 **WHERE** 子句，则更新表中的所有行。例如，以下语句对 **Sales.SalesPerson** 表中的所有行更新 **Bonus**、**CommissionPct** 和 **SalesQuota** 列中的值。

```
USE AdventureWorks;
GO
UPDATE Sales.SalesPerson
SET Bonus = 6000, CommissionPct = .10, SalesQuota = NULL;
```

计算列的值可在更新操作中计算和使用。下面的语句将所有产品型号为 37 的产品价格提高 10%。

```
USE AdventureWorks;
GO
UPDATE AdventureWorks.Production.Product
SET ListPrice = ListPrice * 1.1
WHERE ProductModelID = 37;
```

此外，**SET** 子句中使用的表达式还可以是只返回一个值的子查询。例如，下面的语句将通过 **Sales.SalesOrderDetail** 中的数据来更新 **SalesOrderHeader** 中每笔订单的销售合计和销售日期。

```
USE AdventureWorks;
GO
UPDATE Sales.SalesOrderHeader
SET SubTotal = (SELECT SUM(LineTotal)
                FROM Sales.SalesOrderDetail
                WHERE SalesOrderID = Sales.SalesOrderHeader.SalesOrderID),
    OrderDate = (SELECT MAX(ModifiedDate)
                FROM Sales.SalesOrderDetail
                WHERE SalesOrderID = Sales.SalesOrderHeader.SalesOrderID);
```

10.2.2 使用 FROM 子句更新数据

上文介绍了在 **SET** 子句中使用子查询的方式来更新列的值。但是，这种语句的效率十分低，因为对于 **SalesOrderHeader** 表中的每行都要调用子查询。而通过 **FROM** 子句的方式，可以将数据从一个、多个表或视图拉入到要更新的表中。以下语句将先计算每笔订单的销售合计和最新日期，然后与 **SalesOrderHeader** 进行联接，并使用联接结果中的 **SumLineTotal** 和 **MaxModifiedDate** 列的值分别更新 **SubTotal** 和 **OrderDate** 列的值。

```
UPDATE Sales.SalesOrderHeader
SET SubTotal = D.SumLineTotal,
    OrderDate = D.MaxModifiedDate
FROM Sales.SalesOrderHeader AS O
    INNER JOIN (SELECT SalesOrderID,
                      SUM(LineTotal) AS SumLineTotal,
                      MAX(ModifiedDate) AS MaxModifiedDate
                FROM Sales.SalesOrderDetail
                GROUP BY SalesOrderID) AS D
ON O.SalesOrderID = D.SalesOrderID;
```

如果联接是一对多关系，虽然语句能够正常执行，但是更新操作会具有不确定性。例如，假设 SalesOrderHeader 和 SalesOrderDetail 存在如表 10-1 和表 10-2 所示的数据，并且这两个表在 SalesOrderID 列上存在一对多关系。

表 10-1 SalesOrderHeader 表

SalesOrderID	SubTotal
43659	
43660	

表 10-2 SalesOrderDetail 表

SalesOrderID	LineTotal
43659	10
43659	20
43660	30
43660	40

执行下面的语句，由于 UPDATE 对同一行只会执行一次更新，因此 SubTotal 的结果会具有不确定性。可能是如表 10-3 所示的更新结果，也可能是如表 10-4 所示或其他的更新结果。

```
UPDATE Sales.SalesOrderHeader
SET SubTotal = D.LineTotal
FROM Sales.SalesOrderHeader AS O
INNER JOIN Sales.SalesOrderDetail AS D
ON O.SalesOrderID = D.SalesOrderID;
```

表 10-3 存在一对多关系下的不确定更新结果

SalesOrderID	SubTotal
43659	10.00
43660	30.00

表 10-4 存在一对多关系下的另一种不确定更新结果

SalesOrderID	SubTotal
43659	20.00
43660	40.00

为了防止出现这种不确定结果，还是应当像上面的语句那样，在更新前将 SalesOrderDetail 中的行小计按 SalesOrderID 列合计在一起。参考下面的语句：

```
UPDATE Sales.SalesOrderHeader
SET SubTotal = D.SumLineTotal
FROM Sales.SalesOrderHeader AS O
INNER JOIN (SELECT SalesOrderID,
SUM(LineTotal) AS SumLineTotal
FROM Sales.SalesOrderDetail
GROUP BY SalesOrderID) AS D
ON O.SalesOrderID = D.SalesOrderID;
```

10.2.3 使用 CTE 和视图更新数据

当使用 FROM 子句进行更新时，可能会觉得语句比较复杂。因为首先要进行的是 FROM 子句后表的联接，然后才执行 UPDATE 操作。如果使用上文介绍的公用表表达式（CTE）或视图技术，则语句的可读性会更高一些。

1. 使用 CTE 更新数据

在 CTE 定义中，是我们熟悉的 SELECT 查询语句，将所需要的数据联接在一起。然后对 CTE 执行 UPDATE，所做修改会更新到 CTE 所引用的表中。参考下面的语句：

```
WITH Sales_CTE AS
(
    SELECT O.SalesOrderID, O.SubTotal, D.SumLineTotal
    FROM Sales.SalesOrderHeader AS O
    INNER JOIN (SELECT SalesOrderID,
                     SUM(LineTotal) AS SumLineTotal
                FROM Sales.SalesOrderDetail
                GROUP BY SalesOrderID) AS D
    ON O.SalesOrderID = D.SalesOrderID
)
UPDATE Sales_CTE
SET SubTotal = SumLineTotal;
```

2. 使用视图更新数据

视图与 CTE 的更新方式基本相同，也是先定义视图，将所需要的数据联接在一起，然后对视图执行 UPDATE。参考下面的语句：

```
CREATE VIEW Sales_view
AS
SELECT O.SalesOrderID, O.SubTotal, D.SumLineTotal
FROM Sales.SalesOrderHeader AS O
INNER JOIN (SELECT SalesOrderID,
                     SUM(LineTotal) AS SumLineTotal
                FROM Sales.SalesOrderDetail
                GROUP BY SalesOrderID) AS D
    ON O.SalesOrderID = D.SalesOrderID;

UPDATE Sales_view
SET SubTotal = SumLineTotal;
```

10.3 删除数据

要删除表或视图中的数据，可以使用 DELETE 或 TRUNCATE TABLE 语句。这两个语句在功能上基本相同，但是在删除表中全部行时，TRUNCATE TABLE 速度更快，并且使用更少的系统资源和事务日志资源。

10.3.1 使用 DELETE 删除行

可以使用 DELETE 语句可删除表或视图中的一行或多行。语法的简化形式为：

```
DELETE [table_or_view]
FROM table_sources
WHERE search_condition
```

table_or_view 指定要从中删除行的表或视图。**table_or_view** 中所有符合 WHERE 搜索条件的行都将被删除。如果没有指定 WHERE 子句，将删除 **table_or_view** 中的所有行。FROM 子句指定可由 WHERE 子句搜索条件中的谓词使用的其他表或视图及联接条件，以限定要从 **table_or_view** 中删除的行。不会从 FROM 子句指定的表中删除行，只从 **table_or_view** 指定的表中删除行。

例如，下面的语句从 Sales.SalesOrderDetail 表中删除 SalesOrderID 为 43659 的行。

```
USE AdventureWorks;
GO
DELETE Sales.SalesOrderDetail
FROM Sales.SalesOrderDetail
WHERE SalesOrderID = 43659;
```

也可以省略 DELETE 后面的表，如果省略 **table_or_view**，则默认要删除行的表与 FROM 子句中所指定的表相同。例如，下面语句的功能与上面相同。

```
DELETE FROM Sales.SalesOrderDetail
WHERE SalesOrderID = 43659;
```

而当语句中包含 OUTPUT 子句时，**table_or_view** 却不能省略，详细信息可参考 10.4 节的内容。

除了使用 WHERE 子句限定删除条件外，也可以使用子查询、联接的方式进行限定。其中子查询方式是 ANSI SQL 标准规定的解决方案，而联接方式则是 Transact-SQL 的扩展功能。下面的两个语句都将是 SalesPersonQuotaHistory 表中删除行，该表基于 SalesPerson 表中所存储的本年度迄今为止的销售业绩。

```
-- ANSI SQL: 2003 标准
USE AdventureWorks;
GO
DELETE FROM Sales.SalesPersonQuotaHistory
WHERE SalesPersonID IN
    (SELECT SalesPersonID
     FROM Sales.SalesPerson
     WHERE SalesYTD > 2500000.00);

-- Transact-SQL 扩展
USE AdventureWorks;
GO
DELETE FROM Sales.SalesPersonQuotaHistory
FROM Sales.SalesPersonQuotaHistory AS spqh
    INNER JOIN Sales.SalesPerson AS sp
    ON spqh.SalesPersonID = sp.SalesPersonID
WHERE sp.SalesYTD > 2500000.00;
```

10.3.2 使用 TRUNCATE TABLE 删除所有行

如果要删除表中的所有行，应当使用 TRUNCATE TABLE，而不是未包含 WHERE 子句的 DELETE 语句。与 DELETE 语句相比，TRUNCATE TABLE 具有以下几个优点。

- DELETE 事务总是会被完整地记录到日志中，每次删除一行，就会在事务日志中为所删除的每行记录一个项。而 TRUNCATE TABLE 通过释放用于存储表数据的数据页来删除数据，并且在事务日志中只记录页释放。因此，TRUNCATE TABLE 所用的事务日志空间较少。

- 当使用行锁执行 DELETE 语句时，将锁定表中各行以便删除。而 TRUNCATE TABLE 则始终锁定表和页，而不是锁定各行。因此，TRUNCATE TABLE 使用的锁较少。

- 执行 DELETE 语句后，表仍会包含空页。例如，必须至少使用一个排他 (LCK_M_X) 表锁，才能释放堆中的空表。如果执行删除操作时没有使用表锁，表（堆）中将包含许多空页。对于索引，删除操作会留下一些空页，尽管这些页会通过后台清除进程迅速释放。而 TRUNCATE TABLE 在没有例外的情况下，在表中不会留有任何页。

无论是 DELETE，还是 TRUNCATE TABLE，在清空表后，但是表的定义、与表相关的索引和其他关联对象仍旧会保留在数据库中。但是，如果表中包含标识列，在使用 TRUNCATE TABLE 清空表后，该列的计数器将重置为最初的种子值。而使用 DELETE 却不会存在这种重置行为。

下面的语句将删除 JobCandidate 表中的所有数据。在 TRUNCATE TABLE 语句之前和之后使用 SELECT 语句来比较结果。

```
USE AdventureWorks;
GO
SELECT COUNT(*) AS BeforeTruncateCount
FROM HumanResources.JobCandidate;
GO
TRUNCATE TABLE HumanResources.JobCandidate;
GO
SELECT COUNT(*) AS AfterTruncateCount
FROM HumanResources.JobCandidate;
GO
```

注意：由于 TRUNCATE TABLE 语句不把删除的行放在事务日志中，因此该操作不能回滚。并且 TRUNCATE TABLE 语句也不能激活表的 DELETE 触发器。不能在外键约束中作为父表引用的表上使用 TRUNCATE TABLE 语句。

10.4 使用 TOP 限制数据修改

TOP 子句除了可以使用在 SELECT 语句中，也可以在 INSERT、UPDATE 和 DELETE 语句中使用。但是，在已分区视图中，不能将 TOP 与 UPDATE 和 DELETE 语句一起使用。

虽然允许更新使用 TOP 子句创建的视图。但是，由于 TOP 子句包含在视图定义中，所以更新

后结果有可能不再符合 TOP 表达式的要求，导致某些行从视图中消失。

10.4.1 使用 TOP 限制插入数据

与 INSERT、UPDATE 或 DELETE 一起使用的 TOP 子句，被引用行不会按任何顺序排列。也就是说，TOP 子句返回的是随机 n 行。例如，虽然下面的 INSERT 语句包含了 ORDER BY 子句，但是插入 Table2 的是 Table1 中随机的两行，而不是按 ColumnA 列排序后返回的前两行。

```
INSERT TOP (2) INTO Table2 (ColumnB)
  SELECT ColumnA FROM Table1
  ORDER BY ColumnA;
```

如果要确保插入 Table1 的是 SELECT 子查询返回的前两行，需要按如下方式书写查询语句。

```
INSERT INTO Table2 (ColumnB)
  SELECT TOP (2) ColumnA FROM Table1
  ORDER BY ColumnA;
```

10.4.2 使用 TOP 限制更新数据

当在 UPDATE 语句中使用 TOP 子句时，也存在与 INSERT 类似的问题。例如，假设要为一位高级销售人员减轻销售负担，将一些客户分配给另一位销售人员。下面的语句将从编号为 275 的销售人员那里随机抽样 10 个客户给编号为 276 的销售人员。

```
USE AdventureWorks;
GO

UPDATE TOP (10) Sales.Store
  SET SalesPersonID = 276
  WHERE SalesPersonID = 275;
```

而下面的语句则是先用子查询从 Sales.Store 中取出了按 CustomerID 排序的 10 个客户，然后使用联接的方式指定了更新条件，将这些客户分配给编号为 276 的销售人员。

```
UPDATE Sales.Store
  SET SalesPersonID = 276
  FROM (SELECT TOP 10 CustomerID, SalesPersonID
        FROM Sales.Store
        WHERE SalesPersonID = 275
        ORDER BY CustomerID DESC) AS S
  WHERE Sales.Store.SalesPersonID = S.SalesPersonID
        AND Sales.Store.CustomerID = S.CustomerID;
```

上面的语句也可以改写为下面使用 IN 关键字和子查询的方式指定更新条件，语句会更加简洁一些，如：

```
UPDATE Sales.Store
  SET SalesPersonID = 276
  WHERE CustomerID IN (SELECT TOP 10 CustomerID
                      FROM Sales.Store
```

```
WHERE SalesPersonID = 275  
ORDER BY CustomerID DESC);
```

10.4.3 使用 TOP 限制删除数据

在 DELETE 语句中直接使用 TOP 语句时，也是随机选择行。例如，下面的语句将从 Sales.Store 中随机删除 SalesPersonID 为 275 的 10 行数据。

```
USE AdventureWorks;  
GO  
  
DELETE TOP (10)  
FROM Sales.Store  
WHERE SalesPersonID = 275;
```

如果需要按指定顺序删除行，则应当使用包含 ORDER BY 子句的子查询筛选出要删除的行。例如，下面的语句将从 Sales.Store 中删除按 CustomerID 排序的 10 行。

```
DELETE FROM Sales.Store  
WHERE CustomerID IN (SELECT TOP 10 CustomerID  
                     FROM Sales.Store  
                     WHERE SalesPersonID = 275  
                     ORDER BY CustomerID DESC);
```

10.5 使用 OUTPUT 输出受影响行的信息

在 INSERT、UPDATE、DELETE 中使用 OUTPUT 子句可以返回受影响各行中的信息，这些结果可以返回到处理应用程序，以供在确认消息、存档以及其他类似的应用程序要求中使用。也可以将这些结果插入表或表变量。

OUTPUT 子句对于在 INSERT 或 UPDATE 操作之后检索标识列或计算列的值可能非常有用。但需要注意的是，对于具有 OUTPUT 子句的 UPDATE、INSERT 或 DELETE 语句，即使在遇到错误需要回滚时，也会将行返回到客户端。如果在运行语句的过程中出现任何错误，都不应使用该结果。

OUTPUT 子句输出受影响行的原理是：对于被修改行的信息，会被保存在名为 Inserted 或 Deleted 的特殊表中。通过指定这两个表，可以将修改结果返回到客户端。对于 INSERT 语句，可以通过 Inserted 获取新插入行的信息；对于 DELETE 语句，可以通过 Deleted 获取被删除行的信息；对于 UPDATE 语句，实际上是通过删除旧行和插入新行，分两个步骤来完成更新操作的。因此，可以通过 Deleted 查询更新前的信息，通过 Inserted 查询更新后的信息。

10.5.1 在 INSERT 中使用 OUTPUT 子句

在 INSERT 中使用 OUTPUT 子句，通常用于检索标识列或计算列的值。因为这些列的值是由数据库引擎自动生成或计算的，在新行插入前，我们无法获知它们的值。而在应用程序开发过程中，许多时候却需要反馈这样的数据给客户端。例如，假设有一个考试报名程序，在考生填写完必要信



但是，当在 DELETE 中包含 OUTPUT 子句时，一般却不能省略。

```
DELETE [table_or_view]
FROM table_sources
WHERE search_condition
```

例如，下面的语句从 Candidate 中删除 ID 为 2 的行，并返回删除结果给调用方。

```
DELETE Candidate
  OUTPUT deleted.*
FROM Candidate
WHERE ID =2;
```

如果希望省略 table_or_view，上面的语句应当书写为如下的格式：

```
DELETE
FROM Candidate
  OUTPUT deleted.*
WHERE ID =2;
```

下面的语句在 FROM 子句中使用联接来定义搜索条件，删除 ProductProductPhoto 表中的行。在这种情况下，table_or_view 是不能省略的。OUTPUT 子句将所删除行的 deleted.ProductID 和 deleted.ProductPhotoID 列，以及 Product 表中的列输出到表变量 @MyTableVar 中。如图 10-4 所示。

```
USE AdventureWorks;
GO
DECLARE @MyTableVar table
(ProductID int NOT NULL,
 ProductName nvarchar(50)NOT NULL,
 ProductModelID int NOT NULL,
 PhotoID int NOT NULL);

DELETE Production.ProductProductPhoto
  OUTPUT deleted.ProductID,
         p.Name,
         p.ProductModelID,
         deleted.ProductPhotoID
  INTO @MyTableVar
FROM Production.ProductProductPhoto AS ph
  JOIN Production.Product as p
  ON ph.ProductID = p.ProductID
  WHERE p.ProductModelID BETWEEN 120 and 130;

-- 显示表变量的结果
SELECT ProductID, ProductName, ProductModelID, PhotoID
FROM @MyTableVar
ORDER BY ProductModelID;
GO
```

	ProductID	ProductName	ProductModelID	PhotoID
1	842	Touring-Penniers, Large	120	1
2	878	Fender Set - Mountain	121	1
3	879	All-Purpose Bike Stand	122	1
4	823	LL Mountain Rear Wheel	123	160
5	824	HL Mountain Rear Wheel	124	160
6	825	HL Mountain Rear Wheel	125	160
7	826	LL Road Rear Wheel	126	160
8	894	Rear Derailleur	127	154
9	907	Rear Brakes	128	1

图 10-4 输出的删除结果



2

第 2 部分 开发篇

第 11 章 视图

第 12 章 游标

第 13 章 存储过程

第 14 章 触发器

第 15 章 用户自定义函数



第 11 章 视图

视图可以被看成是虚拟表或存储查询，其中包含一个 `SELECT` 语句，可以被当作表直接引用。除非是索引视图，否则视图的数据不会存储在数据库中。当访问一个非索引视图时，实际上访问的是基表。也就是说，当访问视图时，查询处理器会根据视图定义生成访问基表的执行计划。

视图通常用来集中、简化和自定义每个用户对数据库的不同认识。视图可用作安全机制，方法是允许用户通过视图访问数据，而不授予用户直接访问视图基表的权限。视图可用于提供向后兼容接口来模拟曾经存在但其架构已更改的表。此外，通过数据分区可以提高对数据的访问性能，再使用视图来联结各分区数据，可以实现统一的数据访问接口。

11.1 创建视图

视图最多可以包含 1024 列，列名称必须唯一。定义视图可以使用 `CREATE VIEW` 语句，但是 `SELECT` 查询中不能包含 `COMPUTE` 子句、`COMPUTE BY` 子句或 `INTO` 关键字；除非 `SELECT` 中包含 `TOP` 或 `FOR XML` 子句，否则不能使用 `ORDER BY` 子句；定义视图的查询不能包含指定查询提示的 `OPTION` 子句，也不能包含 `TABLESAMPLE` 子句。

11.1.1 创建简单视图

当需要频繁地查询列的某种组合时，简单视图非常有用。下面要创建的视图的数据来自 AdventureWorks 数据库的 `HumanResources.Employee` 和 `Person.Contact` 表。这些数据可提供雇员的姓名和雇用日期信息。

```
USE AdventureWorks ;
GO
CREATE VIEW hiredate_view
AS
SELECT c.FirstName, c.LastName, e.EmployeeID, e.HireDate
FROM HumanResources.Employee e
JOIN Person.Contact c
ON e.ContactID = c.ContactID ;
```

在创建视图的 `SELECT` 选择列表中，尽量不要使用 `*`，因为对源表数据结构的更改会影响到视图的结构。

之所以不能在视图中使用 **ORDER BY** 子句，因为视图实际上是一个逻辑实体，除非你创建索引视图，才在其中存储数据。并且，引用视图与引用表是完全相同的，所以你可以在外部查询中使用 **ORDER BY** 子句进行排序。

11.1.2 创建索引视图

可以为视图创建一个唯一的聚集索引，这样视图称为索引视图。索引视图是被具体化了的视图，即它已经过计算并存储，而不仅仅是一个查询语句。

如果很少更新基础数据，则使用索引视图能够获得更佳效果。但是，如果经常更新基础数据，则维护索引视图数据的成本可能超过使用索引视图所带来的性能收益。如果基础数据以批处理的形式定期更新，但在更新之间主要作为只读数据进行处理，可以考虑在更新前先删除所有索引视图，然后再重新生成。这样做可以提高更新的性能。

索引视图可以提高下列查询类型的性能。

- **处理大量行的联接和聚合。**许多查询经常执行的联接和聚合操作。例如，假设在记录库存的联机事务处理（OLTP）数据库中，许多查询都要联接 **ProductMaster**、**PProductVendor** 和 **VendorMaster** 表。虽然执行此联接的每个查询需要处理的行可能并不多，但若将成百上千个这样的查询联接起来，则总的处理量将非常大。由于这些关系不太可能经常更新，所以可以通过定义一个索引视图存储联接结果，以此提高整个系统的总体性能。

- **决策支持。**分析系统的特点是存储不经常更新的、汇总的聚合数据。而许多决策支持查询的特点是进一步聚合数据并联接大量行。同时，决策支持系统有时包含具有大量列和/或较大列的宽表，如果创建的视图仅包含所需要的部分列，则再对这种窄子集进行的查询可以提高查询性能。创建包含单个表的列的子集的窄索引视图称为“垂直分区”策略，因为它垂直拆分表。

在创建索引视图时，不能引用其他视图，只能引用基表；并且基表必须与要创建的视图位于同一数据库中，二者的所有者也要相同。此外，在创建视图时必须指定 **SCHEMABINDING**，还要求 **ANSI_NULLS** 和 **QUOTED_IDENTIFIER** 连接选项应当为 **ON**。为视图创建的第 1 个索引必须是唯一聚集索引，随后创建的索引可以是非聚集索引，也不要求唯一。

下面是一个创建表和索引视图的语句：

```
CREATE TABLE wide_tbl(a int PRIMARY KEY, b int, ..., z int)
CREATE VIEW v_abc WITH SCHEMABINDING AS --创建视图
SELECT a, b, c
FROM dbo.wide_tbl
WHERE a BETWEEN 0 AND 1000
CREATE UNIQUE CLUSTERED INDEX i_abc ON v_abc(a) --创建唯一聚集索引
```

再对 **v_abc** 进行查询，比直接从基础表进行查询能够提高性能，如：

```
SELECT b, count_big(*), SUM(c)
FROM wide_tbl
```



```
WHERE a BETWEEN 0 AND 1000
GROUP BY b
```

11.1.3 创建分区视图

在视图中，可以使用 UNION 运算符将两个或多个查询的结果组合到一起。这在用户看来是一个单独的表，但实际是作为多个表分别存储在同一个服务器实例中；或存储在称为联合数据库服务器的自主服务器实例组中。这样的视图称为分区视图。例如，如果一个表包含华盛顿的销售数据，另一个表包含加利福尼亚的销售数据，则可以对这两个表使用 UNION 创建一个视图。该视图将包含这两个地区的销售数据。

使用分区视图时，首先要创建几个结构相同的表，并指定一个约束来指定可在各个表中添加的数据范围。然后，使用这些基表创建视图。查询视图时，将自动确定查询所影响的表并仅引用这些表。例如，如果一个查询指定只需要华盛顿州的销售数据，将只读取包含华盛顿销售数据的表，而不访问其他表。

假设 Customers 表的数据分布在 3 个服务器的 3 个成员表中：Server1 上的 Customers_33、Server2 上的 Customers_66 和 Server3 上的 Customers_99。在 Server1 中可以通过以下方式定义分区视图：

```
CREATE VIEW Customers
AS
--从本地成员表选取
SELECT *
FROM CompanyData.dbo.Customers_33
UNION ALL
--从 Server2 上的成员表选取
SELECT *
FROM Server2.CompanyData.dbo.Customers_66
UNION ALL
--从 Server3 上的成员表选取
SELECT *
FROM Server3.CompanyData.dbo.Customers_99
```

11.2 修改视图

视图定义之后，可以使用 ALTER VIEW 语句更改视图名称或视图定义而无需删除并重新创建视图。修改视图并不会影响相关对象（例如，存储过程或触发器），除非对视图定义的更改使得该相关对象不再有效。例如，AdventureWorks 数据库中的 employees_view 视图的定义为：

```
CREATE VIEW employees_view
AS
SELECT EmployeeID FROM HumanResources.Employee
```

假设存储过程 employees_proc 的定义如下：

```
CREATE PROC employees_proc
AS
SELECT EmployeeID from employees_view --从 employees_view 视图读取数据
```

这时，如果对视图 `employees_view` 修改为检索 `LastName` 列而不是 `EmployeeID`，此时执行 `employees_proc` 将失败，因为该视图中已不存在 `EmployeeID` 列。

```
ALTER VIEW employees_view
AS
SELECT LastName FROM Person.Contact c
JOIN HumanResources.Employee e ON c.ContactID = e.ContactID
```

当基表结构更改后，应当通过 `sp_refreshview` 刷新视图的元数据信息，否则视图仍旧是创建时所保存的元数据信息。例如，下面的语句创建了一个基于 `Table1` 表的 `View1` 的视图。注意，这里仅是为了演示需要，在实际工作中应避免使用 `SELECT *`。

```
CREATE TABLE Table1(col1 int, col2 int);
INSERT INTO Table1 VALUES(1, 2);
GO

CREATE VIEW View1
AS
SELECT * FROM Table1;
```

执行下面的语句，可得到如图 11-1 所示的结果。

```
SELECT * FROM View1;
```



	col1	col2
1	1	2

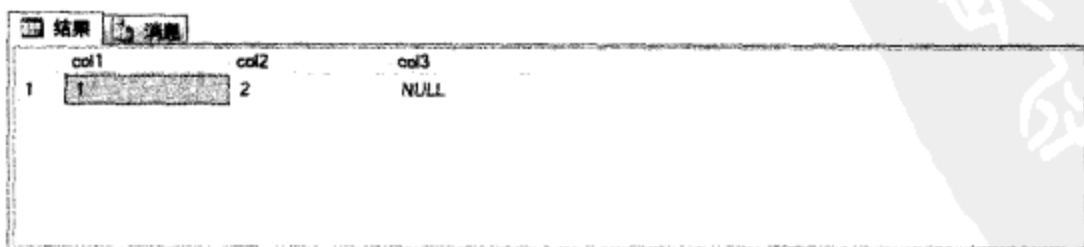
图 11-1 视图 View1 查询结果

执行下面的语句，向 `Table1` 中添加一列。再次执行上面的 `SELECT` 查询，就会发现返回结果与图 11-1 完全相同。

```
ALTER TABLE Table1 ADD col3 int;
```

执行下面的语句，刷新视图信息后，再次执行 `SELECT` 查询，将会返回如图 11-2 所示的正确结果。

```
EXEC sp_refreshview View1;
```



	col1	col2	col3
1	1	2	NULL

图 11-2 视图 View1 查询结果

为防止对基表的修改影响到视图的结构，可以使用 `SCHEMABINDING` 子句将视图绑定到基础

表的架构。指定了 SCHEMABINDING 后, 要对影响视图结构的基表列或其他对象进行修改时, 必须首先修改或删除视图定义本身, 解除这种依赖关系。使用 SCHEMABINDING 时, 查询语句必须使用两部分名称 (即 schema.object 格式), 所有引用的对象必须在同一个数据库内, 并且不允许在 SELECT 列表中使用*。

下面使用 SCHEMABINDING 子句重新创建了视图 View1。

```
CREATE VIEW View1 WITH SCHEMABINDING
AS
SELECT col1, col2 FROM dbo.Table1;
```

执行下面的语句, 删除被视图引用的 col2 列。

```
ALTER TABLE Table1 DROP COLUMN col2;
```

将返回如下的错误信息:

```
消息 5074, 级别 16, 状态 1, 第 1 行
对象 'View1' 依赖于 列 'col2'。
消息 4922, 级别 16, 状态 9, 第 1 行
```

由于一个或多个对象访问此列, ALTER TABLE DROP COLUMN col2 失败。

如果视图包含别名数据类型列, 则无法指定 SCHEMABINDING。

11.3 更新视图中的数据

视图不仅可以作为 SELECT 查询的目标, 也可以通过视图修改基表的数据, 其方式与修改表中数据一样。当修改视图中数据时, 视图将自动修改相应的基表。但是, 在创建可更新视图时, 应注意以下几个限制。

- 视图中要修改的列必须是引用的基表列。并且在向视图插入数据时, 未指定值的其他基表列必须可以隐式获取值, 才能够插入成功。
- 不能修改使用聚合函数 (如 AVG、COUNT、SUM、MIN、MAX、GROUPING 等) 的计算列。
- 不能修改使用集合运算符 (UNION、UNION ALL、CROSSJOIN、EXCEPT 和 INTERSECT) 形成的列得出的计算结果。
- 被修改的列不受 GROUP BY、HAVING 或 DISTINCT 子句的影响。
- 如果视图中使用了联接查询, 则 UPDATE 或 INSERT 只能影响联接的一端。也就是说, INSERT 或 UPDATE 中指定的列只能是一端的, 并且不能从视图中删除数据。
- 如果在视图定义中使用了 WITH CHECK OPTION 子句, 则所有在视图上执行的数据修改语句都必须符合定义视图的 SELECT 语句中所设置的条件, 防止数据在修改完成后从视图中消失。任何可能导致行消失的修改都会被取消, 并显示错误。
- 不能对视图中的 text、ntext 或 image 列使用 READTEXT 语句和 WRITETEXT 语句。

但是，如果在视图上定义了 **INSTEAD OF** 触发器，则可以在触发器中使用自己的代码来代替视图的默认修改行为，故而可以不遵循上述限制。

例如，在下面的视图中，不能接受对 **TotalSalesContacts** 的 **LastName** 列进行修改，因为该列受到 **GROUP BY** 子句的影响。如果同一个姓氏有多个实例，则数据库引擎将无法知道该更新、插入或删除哪个实例。同样，尝试对 **TotalSalesContacts** 的 **TotalSales** 列进行修改将返回错误，因为该列是从聚合函数派生的列。数据库引擎不能直接跟踪该列到其基表 **SalesOrderHeader**。

```
CREATE VIEW TotalSalesContacts
AS
SELECT C.LastName,
SUM(O.TotalDue) AS TotalSales
FROM Sales.SalesOrderHeader O, Person.Contact C
WHERE C.ContactID = O.ContactID
GROUP BY LastName;
```

下面的语句将创建一个基于联接的视图 **VCS**。

```
CREATE TABLE Class
(ClassID int NOT NULL,
ClassName varchar(10) NOT NULL);
INSERT INTO Class VALUES
(1, '班级 1'),
(2, '班级 2');

CREATE TABLE Student
(ClassID int NOT NULL,
StuID int NOT NULL);
INSERT INTO Student VALUES
(1, 10001),
(1, 10002),
(1, 10003),
(2, 20001),
(2, 20002),
(2, 20003);
GO

CREATE VIEW VCS
AS
SELECT C.ClassID, C.ClassName, S.StuID
FROM Class AS C
JOIN Student AS S
ON C.ClassID = S.ClassID;
```

下面的两个修改语句都不能执行。第一条语句是因为违反了不能从基于联接的视图中删除数据的限制；第二条语句是因为 **SET** 子句中既包含 **Class** 表中的列，也包含 **Student** 表中的列，如果违反了则只能影响联接的一端的限制。

```
DELETE FROM VCS WHERE ClassID = 1
UPDATE VCS SET ClassName = '班级 3', StuID = 10004
WHERE ClassID = 1;
```

对基于联接的视图进行更新，应当特别注意，因为修改基表后的结果与用户期望值可能并不一致。首先执行下面的语句，将看到视图中的修改前的数据，如表 11-1 所示。

```
SELECT * FROM VCS;
```

表 11-1

视图数据

ClassID	ClassName	StuID
1	班级 1	10001
1	班级 1	10002
1	班级 1	10003
2	班级 2	20001
2	班级 2	20002
2	班级 2	20003

执行下面的语句，将 StuID 为 10001 的行的 ClassName 修改为“班级 3”，按语句逻辑，应当仅修改表 11-1 中的第 1 行数据。实际上，更新的是基表 Class 中 ClassID 为 1 的 ClassName 的值。

```
UPDATE VCS SET ClassName = '班级 3'
WHERE StuID = 10001;
```

更新完成后，再查询视图中的数据，就会看到前 3 行数据中 ClassName 都被修改了，如表 11-2 所示。

表 11-2

更新后的视图数据

ClassID	ClassName	StuID
1	班级 3	10001
1	班级 3	10002
1	班级 3	10003
2	班级 2	20001
2	班级 2	20002
2	班级 2	20003

尤其是像以下语句，它更新的是联接条件列的值，虽然可以正常更新，但是再次查询视图中的数据时，就会发现只剩下表 11-2 所示的 ClassID 为 2 的行。这是因为 Student 表中没有 ClassID 为 3 的数据，执行 JOIN 联接后，Class 中 ClassID 为 3 的行被丢弃了。

```
UPDATE VCS SET ClassID = 3
WHERE StuID = 10001;
```

为了防止出现丢弃行，确保在数据经修改后仍可通过视图看到数据，应在视图中使用 CHECK OPTION 选项。下面是修改后的创建视图语句，再次执行上面的 UPDATE 语句时，将提示错误信息。

```
CREATE VIEW VCS
AS
SELECT C.ClassID, C.ClassName, S.StuID
FROM Class AS C
JOIN Student AS S
```



```
ON C.ClassID = S.ClassID
WITH CHECK OPTION;
```

11.4 删除和重命名视图

可以使用 **DROP VIEW** 语句删除一个视图。例如，下面的语句删除 **dbo.MyView** 视图：

```
DROP VIEW dbo.MyView;
```

可以使用 **sp_rename** 存储过程对视图进行重命名，语法格式如下：

```
sp_rename [ @objname = ] 'object_name' , [ @newname = ] 'new_name'
[ , [ @objtype = ] 'object_type' ]
```

其中，[@objname =] 'object_name' 用于指定用户对象或数据类型的当前名称。[@newname =] 'new_name' 指定对象的新名称。[@objtype =] 'object_type' 是要重命名的对象的类型，可用值如表 11-3 所示。

表 11-3

object_type 的可用值

值	说 明
COLUMN	重命名列
DATABASE	重命名数据库
INDEX	重命名用户定义索引
OBJECT	在 sys.objects 中跟踪的类型的项。例如，OBJECT 可用于重命名约束（CHECK、FOREIGN KEY、PRIMARY/UNIQUE KEY）、用户表和规则等对象
USERDATATYPE	重命名别名数据类型或 CLR 用户定义类型

例如，下面的语句将 **dbo.MyView** 视图重命名为 **MyNewView**：

```
EXEC sp_rename 'dbo.MyView', 'MyNewView';
```



第 12 章 游标

在关系数据库中，由查询返回的是满足检索条件的所有行。但是，这种行集（结果集）对于应用程序、特别是交互式联机应用程序来讲，并不能将整个行集作为一个单元来有效地处理。这些应用程序需要一种机制以便每次处理一行或一部分行，例如，一个发送邮件的程序，需要向收件人逐个发送，只有当前行处理完成后，才可以进行下一条。游标便提供了这方面的功能，游标是对结果集的一种扩展。游标这种逐行、或是有序访问数据的特性，会带来一定的开销，在运行本章中的示例语句时，读者会发现数据的检索速度明显不如结果集查询快。

12.1 创建游标的步骤

实际上，游标主要应用于存储过程和触发器，在存储过程或触发器中使用游标的典型步骤如下。

- 声明变量，用于包含游标返回的数据。
- 使用 `DECLARE CURSOR` 语句将游标与 `SELECT` 语句相关联。另外，`DECLARE CURSOR` 语句还定义游标的特性，例如，游标名称以及游标是只读还是只进。
- 使用 `OPEN` 语句执行 `SELECT` 语句并填充游标。
- 使用 `FETCH INTO` 语句提取单个行，并将每列中的数据移至指定的变量中。然后，其他语句可以引用变量来访问提取的数据值。
- 使用 `CLOSE` 语句结束游标的使用。关闭游标可以释放某些资源，例如，游标结果集及其对当前行的锁定，但如果重新发出一个 `OPEN` 语句，则该游标结构仍可用于处理。`DEALLOCATE` 语句可以完全释放分配给游标的资源，包括游标名称。释放游标后，必须使用 `DECLARE` 语句来重新生成游标。

下面的语句将创建一个名为 `SalesOrderHeaderProc` 的存储过程，其中包含了一个 `OrderHeaderCursor` 游标，用于列出的 `Sales.SalesOrderHeader` 表中的订单信息。

```
USE AdventureWorks;
GO

CREATE PROCEDURE SalesOrderHeaderProc
AS
DECLARE @SalesOrderID int;
DECLARE @OrderDate datetime;
DECLARE @ShipDate datetime;
DECLARE @AccountNumber nvarchar(15);
DECLARE @SubTotal money;
```

```

-- 声明游标
DECLARE OrderHeaderCursor CURSOR FOR
    SELECT SalesOrderID,
           OrderDate,
           ShipDate,
           AccountNumber,
           SubTotal
    FROM Sales.SalesOrderHeader
    ORDER BY SalesOrderID
    FOR READ ONLY;

-- 打开游标
OPEN OrderHeaderCursor;
WHILE 0 = 0
BEGIN
    FETCH NEXT -- 读取行
    FROM OrderHeaderCursor
    INTO @SalesOrderID,
        @OrderDate,
        @ShipDate,
        @AccountNumber,
        @SubTotal;

    IF @@FETCH_STATUS <> 0 -- 非 0 值表示执行失败
    BEGIN
        BREAK -- 退出循环
    END
    PRINT CAST(@SalesOrderID AS varchar(10)) + ' ' +
          CONVERT(varchar(10), @OrderDate, 102) + ' ' +
          CONVERT(varchar(10), @ShipDate, 102) + ' ' +
          @AccountNumber + ' ' +
          CAST(@SubTotal AS varchar(20)); -- 输出信息
END

CLOSE OrderHeaderCursor; -- 关闭游标
DEALLOCATE OrderHeaderCursor; -- 释放游标资源
GO

EXECUTE SalesOrderHeaderProc -- 执行存储过程

```

游标的名称位于 **DECLARE** 和 **CURSOR** 关键字之间。实际上，游标的核心就是一个 **SELECT** 语句，在语句中不能使用关键字 **COMPUTE**、**COMPUTE BY**、**FOR BROWSE** 和 **INTO**。由 **SELECT** 语句定义的结果集在执行 **DECLARE CURSOR** 语句时并不实际生成，而是当执行 **OPEN** 语句时才生成结果集。

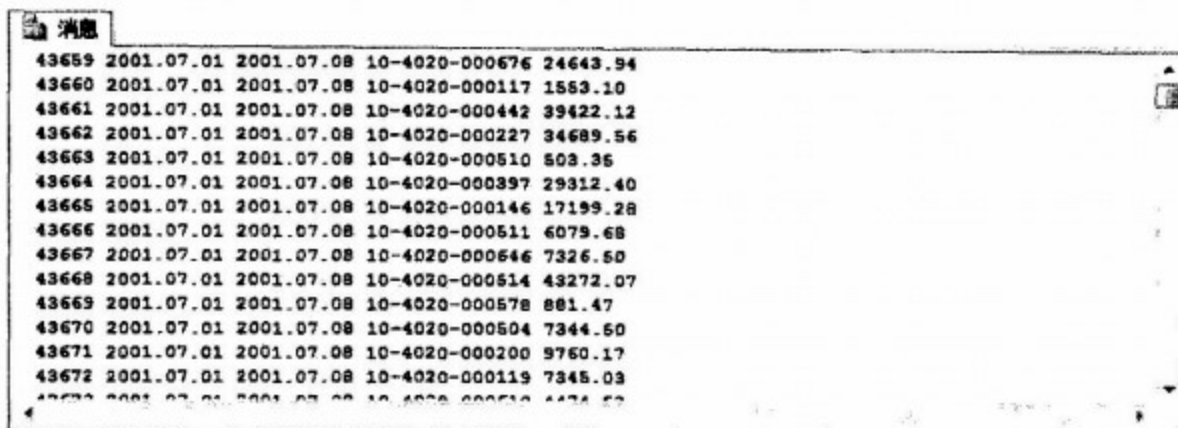
第 1 个 **FETCH NEXT** 语句读取第 1 个选择的行，并把选择列表中的每一个值放在 **INTO** 子句指定的相应变量或参数中。后面的 **FETCH NEXT** 语句继续依次读取下一行，直到 **@@FETCH_STATUS** 函数返回 0 值。

@@FETCH_STATUS 函数返回最近 **FETCH** 语句的状态，可以通过函数的返回值判断数据是否读取完毕。0 表示 **FETCH** 语句读取成功；-1 表示 **FETCH** 语句失败或行不在结果集中；-2 表示提取的行不存在。

在行读取完毕后，应当立即关闭游标。在没有使用 **DEALLOCATE** 语句释放游标资源之前，可以重新使用 **OPEN** 语句打开游标。当游标不再需要时，应当使用 **DEALLOCATE** 语句释放游标所

占用的资源。

执行 SalesOrderHeaderProc 存储过程得到的结果集，如图 12-1 所示。



43659	2001.07.01	2001.07.08	10-4020-000676	24643.94
43660	2001.07.01	2001.07.08	10-4020-000117	1553.10
43661	2001.07.01	2001.07.08	10-4020-000442	39422.12
43662	2001.07.01	2001.07.08	10-4020-000227	34689.56
43663	2001.07.01	2001.07.08	10-4020-000510	503.36
43664	2001.07.01	2001.07.08	10-4020-000397	29312.40
43665	2001.07.01	2001.07.08	10-4020-000146	17199.28
43666	2001.07.01	2001.07.08	10-4020-000511	6079.68
43667	2001.07.01	2001.07.08	10-4020-000646	7326.50
43668	2001.07.01	2001.07.08	10-4020-000514	43272.07
43669	2001.07.01	2001.07.08	10-4020-000578	881.47
43670	2001.07.01	2001.07.08	10-4020-000504	7344.50
43671	2001.07.01	2001.07.08	10-4020-000200	9760.17
43672	2001.07.01	2001.07.08	10-4020-000119	7345.03
43673	2001.07.01	2001.07.08	10-4020-000510	4474.57

图 12-1 执行 SalesOrderHeaderProc 存储过程得到的结果集

也可以通过 SET 语句，将创建的游标赋值给一个变量。但是，该变量必须为 cursor 数据类型。参考下面的语句：

```
DECLARE @OrderHeaderCursor cursor -- 定义变量
SET @OrderHeaderCursor = CURSOR FOR -- 将游标赋值给@OrderHeaderCursor 变量
    SELECT SalesOrderID,
           OrderDate,
           ShipDate,
           AccountNumber,
           SubTotal
    FROM Sales.SalesOrderHeader
    ORDER BY SalesOrderID
    FOR READ ONLY;
```

可以在 CURSOR 关键字的后面使用 LOCAL 和 GLOBAL 关键字来指定游标的作用域。

其中，LOCAL 指定游标的作用域是局部的。对于创建的批处理、存储过程或触发器而言，游标仅在当前过程范围内是可见的，仅能通过 OUTPUT 参数局部游标传递回调用批处理、存储过程或触发器。并且，局部游标将在批处理、存储过程或触发器终止时隐式释放。如果 OUTPUT 参数将游标传递回来，则游标在最后引用它的变量释放或离开作用域时释放。

GLOBAL 指定该游标的作用域相对连接来说是全局的。在由连接执行的任何存储过程或批处理中，都可以引用该游标名称。该游标仅在断开连接时隐式释放。

如果 GLOBAL 和 LOCAL 参数都未指定，则默认值由数据库选项的“默认游标”设置控制。

建议尽量使用局部游标，从而避免在多个存储过程中无意造成的游标名称冲突。

12.2 快速只进游标和可滚动游标

在默认情况下，创建的游标是只进游标，不支持滚动，只能按从头到尾顺序提取行。对于 FETCH 语句来说，也就是只能进行 FETCH NEXT，而不能向前提取。所有由当前用户发出或由其他用户提

交、并影响结果集中的行的 INSERT、UPDATE 和 DELETE 语句，其效果在这些行从游标中提取时是可见的。由于游标无法向后滚动，则在提取行后对行进行的大多数更改通过游标均不可见。当值用于确定所修改的结果集（例如更新聚集索引涵盖的列）中行的位置时，修改后的值通过游标可见。

为了提高只进游标的数据读取性能，可以在 CURSOR 关键字之后添加 FAST_FORWARD 关键字，在打开游标时，查询优化器可以对游标中的 SELECT 语句进行优化。如：

```
DECLARE OrderHeaderCursor CURSOR FAST_FORWARD FOR
    SELECT SalesOrderID,
           OrderDate,
           ShipDate,
           AccountNumber,
           SubTotal
    FROM Sales.SalesOrderHeader
    ORDER BY SalesOrderID
    FOR READ ONLY;
```

但是，如果指定了 SCROLL（创建滚动游标）或 FOR UPDATE（可更新游标），则不能指定 FAST_FORWARD。

要创建可滚动游标，可以在 CURSOR 关键字之前添加 SCROLL 关键字。如：

```
DECLARE OrderHeaderCursor SCROLL CURSOR FOR
    SELECT SalesOrderID,
           OrderDate,
           ShipDate,
           AccountNumber,
           SubTotal
    FROM Sales.SalesOrderHeader
    ORDER BY SalesOrderID
    FOR READ ONLY;
```

可滚动游标支持各种 FETCH 操作，如表 12-1 所示。

表 12-1 FETCH 语句的定位关键字

关 键 字	说 明
NEXT	当前行的下一行
PRIOR	当前行的前一行
FIRST	返回游标中的第一行
LAST	返回游标中的最后一行
ABSOLUTE	n>0, 返回从游标头开始的第 n 行，并将返回行变成新的当前行； n<0, 返回从游标末尾开始的第 n 行，并将返回行变成新的当前行； n=0, 不返回行； 注意：n 必须是整数常量，如果使用变量，则数据类型必须是 smallint、tinyint 或 int
RELATIVE	n>0, 返回从当前行开始的第 n 行，并将返回行变成新的当前行； n<0, 返回当前行之前第 n 行，并将返回行变成新的当前行； n=0, 返回当前行； 注意：在对游标完成第一次提取时，如果在将 n 设置为负数或 0 的情况下指定 FETCH RELATIVE，则不返回行。n 必须是整数常量，如果使用变量，则数据类型必须是 smallint、tinyint 或 int

下面的语句将创建一个可滚动游标，并通过 LAST、PRIOR、RELATIVE 和 ABSOLUTE 选项进行滚动操作。

```
USE AdventureWorks ;
GO
-- 执行下面的 SELECT 语句，显示被游标使用的完整结果集
SELECT SalesOrderID,
       OrderDate,
       SubTotal
FROM Sales.SalesOrderHeader
WHERE SalesOrderID > 75118
ORDER BY SalesOrderID;

-- 声明游标.
DECLARE OrderHeaderCursor SCROLL CURSOR FOR
    SELECT SalesOrderID,
           OrderDate,
           SubTotal
    FROM Sales.SalesOrderHeader
    WHERE SalesOrderID > 75118
    ORDER BY SalesOrderID;

OPEN OrderHeaderCursor;

-- 读取游标中的最后一行
FETCH LAST FROM OrderHeaderCursor ;

-- 读取当前行的前一行
FETCH PRIOR FROM OrderHeaderCursor ;

-- 读取游标中的第 2 行
FETCH ABSOLUTE 2 FROM OrderHeaderCursor ;

-- 读取当前行后的第 3 行
FETCH RELATIVE 3 FROM OrderHeaderCursor ;

-- 读取当前行前面的第 2 行
FETCH RELATIVE -2 FROM OrderHeaderCursor ;

CLOSE OrderHeaderCursor ;
DEALLOCATE OrderHeaderCursor ;
```

得到的结果集如图 12-2 所示。

SalesOrderID	OrderDate	SubTotal	
75119	2004-07-31 00:00:00.000	42.28	
75120	2004-07-31 00:00:00.000	84.96	
75121	2004-07-31 00:00:00.000	74.98	← 完整结果集
75122	2004-07-31 00:00:00.000	30.97	
75123	2004-07-31 00:00:00.000	189.97	
75123	2004-07-31 00:00:00.000	189.97	← 游标中的最后一行
75122	2004-07-31 00:00:00.000	30.97	← 当前行的前一行
75120	2004-07-31 00:00:00.000	84.96	← 游标中的第 2 行
75123	2004-07-31 00:00:00.000	189.97	← 当前行后的第 3 行
75121	2004-07-31 00:00:00.000	74.98	← 当前行前面的第 2 行

图 12-2 使用滚动游标得到的结果集

12.3 静态游标、动态游标和由键集驱动的游标

在默认情况下，创建的游标是动态的。在滚动游标提取行时，游标能够反映结果集中所做的最新更改，结果集中的行数据值、顺序和成员在每次提取时都会改变，所有用户做的全部 UPDATE、INSERT 和 DELETE 语句均通过游标可见。

静态游标（又称不敏感游标）的完整结果集在打开游标时建立在 tempdb 中，静态游标总是按照打开游标时的原样显示结果集。它既不反映在数据库中所做的任何影响结果集成员身份的更改，也不反映对组成结果集的行的列值所做的更改。静态游标始终是只读的。

由于静态游标的结果集存储在 tempdb 的工作表中，因此结果集中的行大小不能超过服务器对表的最大行限制。

要建立一个静态游标，可以在游标名称后面添加 INSENSITIVE 关键字。如：

```
DECLARE OrderHeaderCursor INSENSITIVE CURSOR FOR
    SELECT SalesOrderID,
           OrderDate,
           ShipDate,
           AccountNumber,
           SubTotal
    FROM Sales.SalesOrderHeader
    ORDER BY SalesOrderID
```

即使没有使用 INSENSITIVE 关键字，当某个游标的 SELECT 语句有 DISTINCT、UNION、GROUP BY 或 HAVING 关键字时，处理游标的方式与指定 INSENSITIVE 关键字一样。

打开由键集驱动的游标时，该游标中各行的成员身份和顺序是固定的。由键集驱动的游标由一组唯一标识符（键）控制，这组键称为键集。键集是打开该游标时在 tempdb 中生成的，保存在一个称为 keyset 的表中。

当用户滚动游标时，对非键集列中的数据值所做的更改（由游标所有者做出或由其他用户提交）是可见的。在游标外对数据库所做的插入在游标内不可见，除非关闭并重新打开游标。使用 API 函数（如 ODBC SQLSetPos 函数）通过游标所做的插入在游标的末尾可见。如果试图提取打开游标后已删除的行，@@FETCH_STATUS 将返回“缺少行”状态。对键列进行更新与删除旧键值然后插入新键值作用相同。

如果未通过游标进行更新，则新键值不可见；如果使用 API 函数（如 SQLSetPos）或 SQL 的 WHERE CURRENT OF 子句通过游标进行更新，并且 SELECT 语句的 FROM 子句中不包含 JOIN 条件，则新键值在游标的末尾可见。如果插入时在 FROM 子句中包含远程表，则新键值不可见。尝试检索旧键值将像检索已删除的行时一样获得“缺少行”提取状态。

如果查询引用了至少一个无唯一索引的表，则键集游标将转换为静态游标。

要创建由键集驱动的游标，可以在 CURSOR 关键字后面添加 KEYSET 关键字。如：

```

DECLARE OrderHeaderCursor CURSOR KEYSET FOR
    SELECT SalesOrderID,
           OrderDate,
           ShipDate,
           AccountNumber,
           SubTotal
    FROM Sales.SalesOrderHeader
    ORDER BY SalesOrderID

```

12.4 使用可更新游标进行数据更新

要创建一个可更新游标，可以在游标定义中包含 `FOR UPDATE [OF column_name [,...n]]` 子句，而不是像前面的示例语句那样使用 `FOR READ ONLY` 子句。如果提供了 `OF column_name [,...n]`，则只允许修改列出的列。如果指定了 `UPDATE`，而未指定列的列表，则可以更新所有的列。需要注意的是，不能在静态游标中使用 `FOR UPDATE` 子句。

当定位在可更新游标中的某行上时，可以执行更新或删除操作，这些操作是针对用于在游标中生成当前行的基表行的，称之为定位更新。而不是像 `UPDATE` 语句那样进行搜索更新。

定位更新在打开游标的同一个连接上执行。这就允许数据修改操作共享与游标相同的事务空间，并且使游标持有的锁不会阻止更新。

要在游标中执行定位更新，可以使用 `UPDATE` 或 `DELETE` 语句中的 `WHERE CURRENT OF cursor_name` 子句。`cursor_name` 是 `DECLARE CURSOR` 语句中的游标名称。

参考下面的示例：

```

USE AdventureWorks ;
GO

-- 声明游标.
DECLARE OrderHeaderCursor SCROLL CURSOR FOR
    SELECT SalesOrderID,
           OrderDate,
           SubTotal
    FROM Sales.SalesOrderHeader
    WHERE SalesOrderID > 75118
    ORDER BY SalesOrderID
    FOR UPDATE;

OPEN OrderHeaderCursor;

-- 读取游标中的最后一行
FETCH LAST FROM OrderHeaderCursor ;
UPDATE Sales.SalesOrderHeader
    SET SubTotal = 189.97
    WHERE CURRENT OF OrderHeaderCursor;

-- 读取当前行的前一行.
FETCH PRIOR FROM OrderHeaderCursor ;
DELETE FROM Sales.SalesOrderHeader
    WHERE CURRENT OF OrderHeaderCursor;

CLOSE OrderHeaderCursor
DEALLOCATE OrderHeaderCursor

```



第 13 章 存储过程

在创建 C/S 或 B/S 应用程序时,可以通过两种方式与服务器进行数据交互。一种是将数据检索程序存储在客户端,通过从客户端向服务器发送 SQL 语句的方式返回处理结果;另一种是将多个 SQL 语句组成一个事务处理过程,存储在服务器中,客户端可以像执行一条 SQL 语句那样调用该过程。这个事务处理过程就是存储过程。

存储过程提供了许多标准 SQL 语句所没有的高级特性。其传递参数和执行逻辑表达式的功能,为处理各种复杂任务提供了可能。并且,由于存储过程是经过编译后存储在服务器上的,这减少了执行该过程所需的网络传输带宽和执行时间。如果客户端向服务器直接发送 SQL 语句,这些命令每次都需要进行重新编译,然后才可以被提交执行。同时,存储过程还具有安全特性,可以授予用户具有执行某个存储过程的权限,但是该用户可以不必直接对存储过程中引用的对象具有权限,这就提供了很大的灵活性。例如,假设你希望用户仅能访问某个表的几列,而不是使用 SELECT 的全部列,则可以将需要的那几列放在存储过程中,并拒绝该用户对表的 SELECT 权限。

13.1 存储过程的类型

存储过程大体可分为由用户定义的存储过程、扩展存储过程和系统存储过程。其中,由用户定义的存储过程将是我们重点介绍的内容。

13.1.1 用户定义的存储过程

用户可以创建自己的存储过程,根据所使用的语言,可以分为 SQL 存储过程和 CLR (Common Language Runtime, 公共语言运行时) 存储过程。其中,CLR 存储过程是从 SQL Server 2005 开始提供的功能。

SQL 存储过程是保存在服务器的一个 SQL 语句集合,可以接收和返回用户提供的参数。例如,存储过程中可能包含根据客户端应用程序提供的信息在一个或多个表中插入新行所需的语句。存储过程也可能从数据库向客户端应用程序返回数据。

CLR 存储过程是对 Microsoft .NET Framework 公共语言运行时方法的引用,也可以接收和返回用户提供的参数。CLR 存储过程是一种托管代码,虽然它与扩展存储过程一样是 DLL 文件,但是

它需要 .NET Framework 的支持。

13.1.2 扩展存储过程

扩展存储过程是使用编程语言（例如 C）创建的外部例程（应当是一个 DLL 型文件），然后将其加载到服务器实例中。扩展存储过程的命名使用“xp_”前缀。从 SQL Server 2005 开始，Microsoft .NET Framework 公共语言运行时（CLR）被集成到数据库引擎中，可以使用像 Visual C# 和 Visual Basic 等 .NET 语言编写函数、存储过程和触发器。可以说，这种集成提供了更为可靠和安全的替代方法来编写扩展存储过程。因此，应当使用 CLR 存储过程代替扩展存储过程。并且，在后续版本中将删除扩展存储过程功能，应当避免在新的开发工作中使用该功能。

13.1.3 系统存储过程

在 SQL Server 中，许多管理活动都是通过一种特殊的存储过程执行的，这种存储过程被称为系统存储过程。系统存储过程的命名使用“sp_”前缀，例如，sys.sp_changedbowner 就是一个系统存储过程。从物理意义上讲，系统存储过程存储在源数据库中，并且带有 sp_ 前缀。从逻辑意义上讲，系统存储过程出现在每个系统定义数据库和用户定义数据库的 sys 构架中。也就是说，你可以在自己所创建的数据库中直接使用这些系统存储过程，而不必在架构前面再引用源数据库名称。

13.2 SQL 存储过程

可以在存储过程中包括任意数量和类型的 SQL 语句，但是下面的语句除外。

CREATE AGGREGATE	CREATE RULE
CREATE DEFAULT	CREATE SCHEMA
CREATE 或 ALTER FUNCTION	CREATE 或 ALTER TRIGGER
CREATE 或 ALTER PROCEDURE	CREATE 或 ALTER VIEW
SET PARSEONLY	SET SHOWPLAN_ALL
SET SHOWPLAN_TEXT	SET SHOWPLAN_XML
USE database_name	

存储过程与任何高级语言的过程非常类似，它可以有输入/输出参数、局部变量、数字和字符运算、赋值、数据库操作以及控制流程的逻辑。

13.2.1 创建存储过程

可以使用 CREATE PROCEDURE 语句来创建存储过程，根据可用内存的不同，存储过程最大

可达 128MB。在定义存储过程时，存储过程名称在 CREATE PROCEDURE 关键字之后，AS 关键字表示存储过程主体的开始，存储过程主体由一个或多个 SQL 语句组成。例如，下面的语句创建了一个名为 usp_GetAllEmployees 的存储过程，用于从 vEmployeeDepartment 视图检索数据。

```
USE AdventureWorks;
GO
CREATE PROCEDURE HumanResources.usp_GetAllEmployees
AS
    SELECT LastName, FirstName, JobTitle, Department
    FROM HumanResources.vEmployeeDepartment;
```

要执行一个存储过程，可以使用 EXECUTE 语句，如：

```
EXECUTE HumanResources.usp_GetAllEmployees
```

1. 存储过程的命名建议

在命名存储过程时，建议不要以“sp_”作为名称的前缀。因为“sp_”前缀是 SQL Server 用来指定系统存储过程的。如果指定的名称与系统过程重名，并且没有为存储过程指定架构名称（将默认使用 dbo 架构）或是使用了 dbo 架构，则该存储过程永远不会被执行，取而代之的是始终执行系统存储过程。下面的示例演示了这种行为。sp_who 是一个系统存储过程名称，所创建的 dbo.sp_who 存储过程永远不会被执行。

```
USE AdventureWorks;
GO
CREATE PROCEDURE dbo.sp_who
AS
    SELECT FirstName, LastName FROM Person.Contact;
GO
EXECUTE sp_who;    -- 执行系统存储过程 sys.sp_who
EXECUTE dbo.sp_who; -- 虽然指定了架构，但是仍然执行 sys.sp_who 系统存储过程
```

要解决上述问题，一个是避免使用“sp_”做前缀（如 dbo.usp_who），另一个就是使用显式架构（如 Person.sp_who）。并且使用显式架构限定符能够改善名称解析的性能，因为如果未指定架构名称，则首先在数据库的 sys 架构中寻找该存储过程，找不到的情况下：如果是使用批处理或动态 SQL 调用存储过程的，会继续在调用该存储过程的用户的默认架构中寻找；如果是由另外的存储过程（如 myschma.procl）调用的，则会在所调用存储过程的 myschma 中寻找。如果还找不到，最后才在 dbo 架构中寻找。

2. 存储过程参数

存储过程通过参数与调用程序通信。参数定义应当在存储过程名称的后面，在 AS 关键字的前面。当程序执行存储过程时，可通过存储过程的参数向该存储过程传递值，也可通过 OUTPUT 参数将值返回至调用程序。一个存储过程最多可以有 2100 个参数。每个参数都有名称、数据类型、方向和默认值几个设置选项。

（1）指定参数的名称和数据类型。

参数名称应当以@开始，以后的字符可以是遵守对象标识符规则的任意字符。参数名称不能以@@开始，因为这是 SQL Server 用于标识内置函数的符号。例如，下面创建的 `usp_GetProduct` 存储过程包含有 `@StandardCost` 和 `@ListPrice` 两个参数，参数的数据类型均为 `money`。

```
USE AdventureWorks;
GO

CREATE PROCEDURE Production.usp_GetProduct
    @StandardCost money,
    @ListPrice money
AS
    SELECT [Name],
           StandardCost,
           ListPrice
    FROM Production.Product
    WHERE StandardCost > @StandardCost AND ListPrice > @ListPrice;
```

执行存储过程时，既可以通过显式方式指定参数名称并分配适当的值，也可以直接分配参数值。如果使用了显示方式，则按任意顺序提供参数。如果未指定参数名称，则必须按照参数在存储过程中定义时的顺序（从左至右）来提供参数。

例如，下面的语句使用显式方式为存储过程分配参数值，用于查找 `ListPrice` 大于 100 并且 `StandardCost` 大于 10 的所有行，参数可以按任意顺序放置。

```
EXECUTE Production.usp_GetProduct @ListPrice = 100, @StandardCost = 10
```

而下面的语句则是按存储过程中参数的顺序放置提供参数值。

```
EXECUTE Production.usp_GetProduct 10, 100
```

（2）为参数指定默认值。

在参数定义中可以为可选参数指定一个默认值。执行该存储过程时，如果未指定其他值，则使用默认值。例如，下面的存储过程设置 `@StandardCost` 参数的默认值为 0。

```
USE AdventureWorks;
GO

CREATE PROCEDURE Production.usp_GetProduct
    @StandardCost money = 0,
    @ListPrice money
AS
    SELECT [Name],
           StandardCost,
           ListPrice
    FROM Production.Product
    WHERE StandardCost > @StandardCost AND ListPrice > @ListPrice;
```

要执行该存储过程，可以只为 `@ListPrice` 参数提供值，如：

```
EXECUTE Production.usp_GetProduct @ListPrice = 110
```

由于具有默认值的参数通常是可选参数，所以建议将它们放置在参数列表的末尾，以便于调用。例如，在没有使用显式方式指定参数值的情况下，下面的语句将引发错误。因为 `@StandardCost` 参

数会被默认地赋予 110 值，而 `@ListPrice` 并没有被指定值。

```
EXECUTE Production.usp_GetProduct 110
```

对于字符型的参数，在参数值传递时可以包含通配符。例如，下面的存储过程仅从视图中返回指定的一些雇员（提供名字和姓氏）及其职务和部门名称。该存储过程对传递的参数进行模式匹配。如果没有提供参数，则使用预设的默认值（姓氏以字母 D 开头）。

```
USE AdventureWorks;
GO

CREATE PROCEDURE HumanResources.usp_GetEmployees2
    @lastname varchar(40) = 'D%',
    @firstname varchar(20) = '%'
AS
    SELECT LastName, FirstName, JobTitle, Department
    FROM HumanResources.vEmployeeDepartment
    WHERE FirstName LIKE @firstname
        AND LastName LIKE @lastname;
```

可以按多种组合方式执行 `usp_GetEmployees2` 存储过程，下面只列出了部分组合：

```
-- 使用参数默认值，将搜索姓氏以字母 D 开头的行
EXECUTE HumanResources.usp_GetEmployees2;

-- 搜索姓氏以字母 W 开头的行
EXECUTE HumanResources.usp_GetEmployees2 'W%';

-- 下面的语句将使用@firstname 参数的默认值，搜索姓氏以字母 D 开头的行
EXECUTE HumanResources.usp_GetEmployees2 @firstname = '%';

-- 搜索姓氏以 C 或 K 开头，并且在 n 前是 O 或 E 字母，中间为 ars 的姓氏
EXECUTE HumanResources.usp_GetEmployees2 '[CK]ars[OE]n';

-- 搜索姓氏为 Hesse、名字为 Stefen 的行
EXECUTE HumanResources.usp_GetEmployees2 'Hesse', 'Stefen';

-- 搜索姓氏以字母 H 开头、名字以 S 开头的行
EXECUTE HumanResources.usp_GetEmployees2 'H%', 'S%';
```

（3）指定输出参数。

在默认情况下，所有参数均为输入参数。要指定输出参数，必须在存储过程的参数定义中使用 `OUTPUT` 关键字。当存储过程退出时，它将向调用程序返回输出参数的当前值。例如，下面创建的存储过程定义了一个输出参数 `@ProductCount`，用于返回 `ListPrice` 大于指定值的产品数量。

```
USE AdventureWorks;
GO

CREATE PROCEDURE Production.usp_GetProduct
    @ProductCount int OUTPUT,
    @ListPrice money
AS
    SET @ProductCount =
        (SELECT COUNT(ProductID)
         FROM Production.Product
         WHERE ListPrice > @ListPrice);
```

调用程序也必须使用 OUTPUT 关键字执行该存储过程，才能将该参数值保存到变量中。例如，下面语句将输出的参数值保存到 @MyVar 变量中。

```
DECLARE @MyVar int
EXECUTE Production.usp_GetProduct @MyVar OUTPUT, @ListPrice = 100
SELECT @MyVar
```

(4) 在输出参数中使用 cursor 数据类型。

cursor 数据类型只能用于输出参数，为参数指定 cursor 数据类型后，必须在参数后面指定 VARYING OUTPUT 关键字。所有带 cursor VARYING OUTPUT 参数的存储过程不能通过 OLE DB、ODBC、ADO 和 DB-Library 调用，仅在将 cursor VARYING OUTPUT 参数输出值分配给局部 cursor 变量时，才可以通过批处理、存储过程或触发器调用这些过程。

下面的存储过程中创建了一个 cursor 型输出参数 @List_Cursor，然后将一个可滚动游标赋值给该变量。需要注意的是，为了能够将游标传递给调用存储过程的程序，必须在过程结束的时候使用 OPEN 语句打开游标。

```
USE AdventureWorks;
GO

CREATE PROCEDURE Production.usp_GetProduct
    @StandardCost money,
    @ListPrice money,
    @List_Cursor cursor VARYING OUTPUT
AS
BEGIN
    SET @List_Cursor = CURSOR LOCAL SCROLL FOR --局部可滚动游标
        SELECT [Name],
               StandardCost,
               ListPrice
        FROM Production.Product
        WHERE StandardCost > @StandardCost AND ListPrice > @ListPrice;
    OPEN @list_cursor --打开游标
END
```

为了使用返回的游标，必须声明一个 cursor 类型的变量，并使用 OUTPUT 关键字将它传递给存储过程。参考下面的代码：

```
DECLARE @MyCursor cursor -- 定义变量
EXECUTE Production.usp_GetProduct 100, 10, @MyCursor OUTPUT -- 执行存储过程

IF CURSOR_STATUS('variable', '@MyCursor') <= 0 -- 判断游标是否正常
    PRINT 'N' 游标读取失败'
ELSE
BEGIN
    FETCH NEXT FROM @MyCursor; -- 读取第一行
    WHILE (@@FETCH_STATUS = 0) -- 读取正常的情况下，继续读取，直至完毕
    BEGIN
        FETCH NEXT FROM @MyCursor;
    END
END

CLOSE @MyCursor;
DEALLOCATE @MyCursor;
```

在执行过程时，以下规则适用于 cursor 输出参数：

● 在使用 OPEN 语句打开游标的情况下，游标将获取结果集并进行填充。对于只进游标，在向存储过程调用者返回游标时，仅把游标当前行至游标最后一行范围内的所有行返回给调用者，当前行之前的行不会返回给调用者。如果游标位于第一行的前面，则整个结果集将返回者。在上面这个存储过程中游标，在存储过程关闭前，仅是使用 OPEN 语句打开了游标，没有使用 FETCH 语句进行行移动，因此这个存储过程会返回所有行给调用者。

● 对于可滚动游标，在存储过程执行结束时，结果集中的所有行均会返回给存储过程调用者。返回时，游标当前行的位置停留在存储过程中最后一次执行提取时的位置。

● 对于任意类型的游标，如果游标关闭，则将空值传递回存储过程调用者。如果游标没有打开，也会出现这种情况。注意，关闭状态只有在返回时才有影响。例如，可以在过程中关闭游标，稍后再打开游标，然后将该游标的结果集返回给存储过程调用者。

13.2.2 修改存储过程

如果需要更改存储过程中的语句或参数，可以删除并重新创建该存储过程，也可以通过 ALTER PROCEDURE 语句更改该存储过程。删除并重新创建存储过程时，与该存储过程关联的所有权限都将丢失。更改存储过程时，将更改过程或参数定义，但为该存储过程定义的权限将保留，并且不会影响任何相关的存储过程或触发器。

修改存储过程的方式与创建存储过程完全相同。例如，下面的语句用于修改 Production.usp_GetProduct 存储过程：

```
USE AdventureWorks;
GO

ALTER PROCEDURE Production.usp_GetProduct
    @StandardCost money,
    @ListPrice money
AS
    SELECT [Name],
           StandardCost,
           ListPrice
    FROM Production.Product
    WHERE StandardCost > @StandardCost AND ListPrice > @ListPrice;
```

13.2.3 存储过程的重新编译

在默认情况下，存储过程是经过编译后存储在数据库中的。在执行诸如添加索引或更改索引列中的数据等操作更改了数据库时，应重新编译访问数据库表的原始查询计划，以对其重新优化。在服务器重新启动后，第一次运行存储过程时会自动执行此优化，当存储过程使用的基础表发生变化时，也会执行此优化。但如果添加了存储过程可能从中受益的新索引，将不自动执行优化，则需要一直等到下次重新启动后再运行该存储过程才会进行优化。也可以通过以下几种方式强制重新编译

存储过程。

1. 指定在下次执行时重新编译

可以使用 `sp_recompile` 系统存储过程指定在下次执行存储过程或触发器进行重新编译。例如，以下语句指定将使作用于 `Customer` 表上的存储过程在下次运行时重新编译。

```
EXEC sp_recompile N'Sales.Customer';
```

2. 执行语句级的重新编译

从 SQL Server 2005 开始，引入了对存储过程执行语句级重新编译的功能。也就是说，在重新编译存储过程时，只编译导致重新编译的语句，而不是整个存储过程。

要使用此功能，应当在语句中包含 `RECOMPILE` 查询提示。`RECOMPILE` 指示数据库引擎在执行查询后，丢弃为其生成的查询计划，从而在下次执行时强制重新编译查询计划。如果未指定 `RECOMPILE`，数据库引擎将缓存查询计划并重新使用它们。

例如，下面的存储过程中在 `SELECT` 查询中包含了 `RECOMPILE` 提示，在每次执行存储过程时都会重新编译此 `SELECT` 语句。

```
USE AdventureWorks;
GO

CREATE PROCEDURE Production.usp_GetProduct
    @StandardCost money,
    @ListPrice money
AS
    SELECT [Name],
           StandardCost,
           ListPrice
    FROM Production.Product
    WHERE StandardCost > @StandardCost AND ListPrice > @ListPrice
    OPTION(RECOMPILE);
```

3. 每次执行时重新编译存储过程

可以在创建存储过程时指定 `WITH RECOMPILE` 选项，强制在执行存储过程时对其重新编译。在包含 `WITH RECOMPILE` 选项时，将不为该存储过程缓存执行计划，而在每次执行时都重新编译。

参考下面的示例：

```
USE AdventureWorks;
GO

CREATE PROCEDURE Production.usp_GetProduct
    @StandardCost money,
    @ListPrice money
    WITH RECOMPILE
AS
    SELECT [Name],
           StandardCost,
           ListPrice
```

```
FROM Production.Product
WHERE StandardCost > @StandardCost AND ListPrice > @ListPrice ;
```

需要注意的是，每次执行存储过程时都对其重新编译，这样会导致存储过程的执行变慢。此外，应当尽量避免在存储过程中使用 **SELECT * FROM tablename** 这样的语句，而应当明确指定所需要的列名称。因为在使用*通配符的情况下，在表所包含的列发生变化时，也会发生对存储过程的重编译。

13.2.4 存储过程的错误处理

在某些情况下，由于用户的误操作、数据完整性错误等原因，都可能造成执行存储过程异常终止。这种时候需要向存储过程的调用者返回一个错误信息，通知错误的原因和相关的解决方法。

1. RAISERROR 语句

RAISERROR 语句用于生成一个错误，并将错误信息返回给调用方。此语句生成的错误与数据库引擎代码生成的错误的运行方式完全相同。**RAISERROR** 所需要的错误信息可以由 **ERROR_LINE**、**ERROR_MESSAGE**、**ERROR_NUMBER**、**ERROR_PROCEDURE**、**ERROR_SEVERITY**、**ERROR_STATE** 以及 **@@ERROR** 等系统函数来获得。

(1) 发送特定消息。

下面是一个使用 **RAISERROR** 语句发送特定消息的示例。当为 **@ProductNumber** 参数传递一个“ST-140”或“ST-141”值时，将向调用方发送一个错误消息，并退出存储过程。

```
USE AdventureWorks;
GO

CREATE PROCEDURE dbo.usp_GetProductNumber
    @ProductNumber varchar(20)
AS
    IF (@ProductNumber = 'ST-140') OR (@ProductNumber = 'ST-141')
    BEGIN
        RAISERROR('无效的产品编号: %s.', 12, 1, @ProductNumber)
        RETURN
    END
    SELECT [Name],
           StandardCost,
           ListPrice
    FROM Production.Product
    WHERE ProductNumber = @ProductNumber ;
```

执行下面的语句，向存储过程传递一个“ST-140”值。

```
EXECUTE dbo.usp_GetProductNumber 'ST-140'
```

下面是 **RAISERROR** 语句返回的消息。注意语句中的“%s”参数被替换成了 **@ProductNumber** 的值。

消息 50000, 级别 12, 状态 1, 过程 usp_GetProductNumber, 第 6 行
无效的产品编号: ST-140。

发送特定消息的 RAISERROR 语句的语法格式如下:

```
RAISERROR ( msg_str, severity, state [ , argument [ ....n ] ] )
```

msg_str 是定义的消息, 最长可以有 2047 个字符。如果字符数超过该数值, 会在消息添加一个省略号来表示消息已被截断。

从上面的示例可以看出, 在 msg_str 中可以包含一个类似 “%s” 的参数值, 可以被后面的值替换。下面是使用参数时的语法格式:

```
% [[flag] [width] [.precision] [{h | l}]] type
```

flag 用于指定被替换值与 msg_str 中文本的间距, 以及替换值的对齐方式, 可用的标记如表 13-1 所示。

表 13-1

参数标记

标记	前缀或对齐	说 明
- (减号)	左对齐	在给定字段宽度内左对齐参数值
+ (加号)	符号前缀	如果参数值为有符号类型, 则在参数值的前面加上加号 (+) 或减号 (-)
0 (零)	零填充	在达到最小宽度之前在输出前面加上零。如果出现 0 和减号 (-), 将忽略 0
# (数字)	对 x 或 X 的十六进制类型使用 0x 前缀	当使用 o、x 或 X 格式时, 数字符号 (#) 标志在任何非零值的前面分别加上 0、0x 或 0X。当 d、i 或 u 的前面有数字符号 (#) 标志时, 将忽略该标志
' ' (空格)	空格填充	如果输出值有符号且为正, 则在该值前加空格。如果包含在加号 (+) 标志中, 则忽略该标志

width 指定参数值位置的最小宽度。如果替换值的长度大于 width, 则以替换值的实际长度输出; 如果替换值小于 width, 则填充到 width 指定的宽度。例如, 下面的语句指定 width 的值为 10, 而替换值 “abc” 的宽度为 3, 因此还需要在前面填充 7 个空格, 以达到 width 指定的最小宽度。

```
RAISERROR(N'错误信息: %10s', --msg_str
          10, -- 级别,
          1, -- 状态,
          N'abc'); -- 替换值
-- 输出的消息如下 (注意在 abc 前面添加了 7 个空格):
错误信息:      abc
```

也可以在 width 位置使用星号 (*), 这表示宽度由参数列表中的相关参数指定, 该宽度必须为整数值。参考下面的语句:

```
RAISERROR(N'错误信息: %*s', --msg_str
          10, -- 级别,
          1, -- 状态,
          10, -- 指定第 1 个参数 "*" 的值
          N'abc'); -- 指定第 2 个参数 "s" 的值
```

```
-- 输出的消息如下 (注意在 abc 前面添加了 7 个空格):
错误信息:      abc
```

precision 指定从字符串值的参数值中得到的最大字符数。例如, 下面的语句指定参数的宽度为 10, 从替换值要获取的最大字符数为 1。

```
RAISERROR(N'错误信息: %10.1s', --msg_str
          10, -- 级别,
          1, -- 状态,
          N'abc'); -- 替换值
```

```
-- 输出的消息如下 (注意在 a 前面添加了 9 个空格):
错误信息:      a
```

也可以在此位置使用星号 (*), 则值由参数列表中的相关参数指定。如:

```
RAISERROR(N'错误信息: %*. *s', --msg_str
          10, -- 级别,
          1, -- 状态,
          10, -- 指定第 1 个*的值
          1, -- 指定第 2 个*的值
          N'abc'); -- 替换值
```

{h|l} type 指定消息的格式。如上面示例中一直使用的 **s** 表示消息为字符串。此外, **d** 或 **i** 表示有符号整数, **o** 表示无符号八进制数, **u** 表示无符号整数, **x** 或 **X** 表示无符号十六进制数。

severity 用于定义消息的严重级别。任何用户都可以指定 0~18 之间的严重级别。只有 **sysadmin** 固定服务器角色成员或具有 **ALTER TRACE** 权限的用户才能指定 19~25 之间的严重级别。20~25 之间的严重级别被认为是致命的。如果遇到致命的严重级别, 客户端连接将在收到消息后终止, 并将错误记录到错误日志和应用程序日志。

state 是介于 1~127 之间的任意整数, 用于标识错误的位置, 默认值为 1。在程序的多个位置可能会引发相同的用户定义错误, 可以针对每个位置使用不同的状态号, 这样有助于找到引发错误的代码段。

argument 是在 **msg_str** 中定义的参数。最多可以包含 20 参数个。参考下面的语句:

```
RAISERROR(N'错误信息: %s 用户在%03d 位置%s。', --msg_str
          10, -- 级别,
          1, -- 状态,
          N'MyUser', -- 指定第 1 个参数的替换值
          5, -- 指定第 2 个参数的替换值, %03d 表示使用零填充 (见表 7-2)、宽度为 3、格式为整数
          N'未找到'); -- 指定第 3 个参数的替换值
```

```
-- 输出的消息如下:
错误信息: MyUser 用户在 005 位置未找到。
```

(2) 发送预定义消息。

可以在 **sys.messages** 目录视图中存放一些预先定义的错误消息。然后 **RAISERROR** 语句便可以通过一个存储在视图中消息 ID (50001 和 2147483647 之间的整数) 来发送错误消息。**sys.messages** 中存储的内容如图 13-1 所示。

	message_id	language_id	severity	is_event_logged	text
2	101	1033	15	0	Query not allowed in Waitfor.
3	102	1033	15	0	Incorrect syntax near '%.1s'.
4	103	1033	15	0	The '%S_MSG' that starts with '%.1s' is too long. Max...
5	104	1033	15	0	ORDER BY items must appear in the select list if the...
6	105	1033	15	0	Unclosed quotation mark after the character string '...
7	106	1033	16	0	Too many table names in the query. The maximum a...
8	107	1033	15	0	The column prefix '%.1s' does not match with a tabl...
9	108	1033	15	0	The ORDER BY position number '%d' is out of range...
10	109	1033	15	0	There are more columns in the INSERT statement t...
11	110	1033	15	0	There are fewer columns in the INSERT statement t...
12	111	1033	15	0	'%s' must be the first statement in a query batch.
13	112	1033	15	0	Variables are not allowed in the '%s' statement.

图 13-1 sys.messages 中存储的内容

可以使用 `sp_addmessage` 向 `sys.messages` 中添加用户定义的错误消息。但是只有已经存在美国英语版本消息的情况下，才可以添加另一种语言消息，两种消息版本的严重性必须匹配。例如，下面的语句向视图添加美国英语版本的消息：

```
sp_addmessage 50001, 10, 'The %s user is %s on %03d place.', 'us_English', 'true'
```

其中的第 1 个参数是消息 ID；第 2 个参数是严重级别；第 3 个参数是消息串；第 4 个参数是消息的语言版本，如果省略该参数，则默认使用本地化语言版本，可以从 `sys.syslanguages` 目录视图的 `name` 列获取可用的语言名称；第 5 个参数执行是否将此错误信息记入日志。

下面是使用该预定义消息的示例：

```
RAISERROR(50001,10,1,'MyUser','not found',5)
-- 输出的消息如下：
The MyUser user is not found on 005 place.
```

在添加完美国英语版本消息后，可以添加本地化的语言版本消息。在添加本地话语言消息时，可以省略第 4 和第 5 个参数。由于语言语法不同，本地化消息中的参数可能不会以美国英语消息中相同的顺序出现。所以应当使用顺序号标出与美国英语版本中参数的对应位置。参考下面的语句：

```
sp_addmessage 50001, 10, N'%1!用户在%3!位置%2!。'
```

其中的 %1! 表示对应于美国英语消息中的第 1 个参数，%2! 表示对应于美国英语消息中的第 2 个参数，依次类推。重新使用 `RAISERROR` 语句执行该消息：

```
RAISERROR(50001,10,1,'MyUser','not found',5)
-- 输出的消息如下：
MyUser 用户在 005 位置 not found。
```

要覆盖视图中已有的消息，可以为 `sp_addmessage` 存储过程指定 `replace` 参数。例如，下面的语句将视图中消息 ID 为 50001 的美国英语版本消息替换为下列形式：

```
sp_addmessage 50001, 10, 'The %s user is %s on %03d place. Please restart your application.', 'us_English', 'true', 'replace'
```


要删除 `sys.messages` 中的消息，可以使用 `sp_dropmessage` 存储过程，参考下面的语句：

```
-- 在未指定语言版本的情况下，删除本地语言消息
sp_dropmessage 50001
-- 删除美国英语消息
sp_dropmessage 50001, 'us_english'
```

缺省情况下，`@@ERROR` 内置函数可以返回最近一条 Transact-SQL 导致错误的编号。例如，下面的语句用于判断发生的错误并做响应处理：

```
IF (@@ERROR > 0)
BEGIN
    -- 处理错误的语句
END
```

也可以为 `RAISERROR` 语句添加一个 `WITH SETERROR` 选项来指定使用 `@@ERROR` 返回的错误编号，从而忽略严重级别设置。例如：

```
RAISERROR(50001,16,1,'MyUser','not found',5) WITH SETERROR
```

2. TRY...CATCH 构造

`TRY...CATCH` 构造为处理代码中的错误提供了一种非常好的解决方案，它类似于 Visual C++ 和 Visual C# 语言的异常处理功能。`TRY...CATCH` 构造包括两部分：一个 `TRY` 块和一个 `CATCH` 块。如果 `TRY` 块中的某条 SQL 语句检测到错误条件，将立即把控制传递给 `CATCH` 块，而不再执行 `TRY` 块中所引发错误语句后面的所有语句。可以在 `CATCH` 块中处理错误，如使用 `RAISERROR` 语句向客户端发送一条错误消息。

`CATCH` 块处理该异常错误后，控制将被传递到 `END CATCH` 语句后面的第 1 个 SQL 语句。如果 `END CATCH` 语句是存储过程的最后一条语句，控制将返回到调用该存储过程的代码。

如果 `TRY` 块中没有错误，控制也将传递到关联的 `END CATCH` 语句后面的语句。如果 `END CATCH` 语句是存储过程中的最后一条语句，控制将传递到调用该存储过程的语句。

`TRY` 块以 `BEGIN TRY` 语句开头，以 `END TRY` 语句结尾。在 `BEGIN TRY` 和 `END TRY` 语句之间可以指定一个或多个 SQL 语句。`CATCH` 块必须紧跟 `TRY` 块。`CATCH` 块以 `BEGIN CATCH` 语句开头，以 `END CATCH` 语句结尾。在 SQL 中，每个 `TRY` 块仅与一个 `CATCH` 块相关联。

(1) 使用 `TRY...CATCH` 构造的规则。

● 每个 `TRY...CATCH` 构造都必须包含在一个批处理、存储过程中。例如，下面的示例将 `TRY` 块和 `CATCH` 块分别放置在了不同的批处理中，这将引发一个错误。

```
BEGIN TRY
    SELECT *
      FROM sys.messages
     WHERE message_id = 21;
END TRY
GO
```



生成的错误信息如图 13-2 所示。

ErrorNumber	ErrorSeverity	ErrorState	ErrorProcedure	ErrorLine	ErrorMessage
1	8134	16	1	NULL	3
					遇到以零作除数错误。

图 13-2 返回的错误信息

(3) 在 TRY...CATCH 中使用 RAISERROR 语句。

可以在 TRY 或 CATCH 块内使用 RAISERROR 来影响错误处理行为。在 TRY 块内执行的严重性为 11~19 的错误，RAISERROR 会使控制传递到关联的 CATCH 块。在 CATCH 块内执行的严重性为 11~19 的错误，RAISERROR 将控制返回到调用应用程序或批处理。这样，RAISERROR 可用于返回有关导致 CATCH 块执行的错误信息。

下面的代码示例说明了如何在 CATCH 块内使用 RAISERROR 将原始错误信息返回到调用应用程序或批处理。存储过程 usp_GenerateError 在 TRY 块内执行 DELETE 语句，该语句生成违反约束错误。此错误使执行传递到 usp_GenerateError 内关联的 CATCH 块，存储过程 usp_GenerateError 在此块内使用 RAISERROR 生成错误。RAISERROR 生成的此错误将返回到调用方，并执行调用方中关联的 CATCH 块。

```
USE AdventureWorks;
GO

-- 检验存储过程是否已经存在
IF OBJECT_ID (N'usp_RethrowError', N'P') IS NOT NULL
    DROP PROCEDURE usp_RethrowError;
GO

-- 创建存储过程，使用 RAISERROR 生成一个错误
-- 原始错误信息用于构建 RAISERROR 的 msg_str
CREATE PROCEDURE usp_RethrowError AS
    -- 如果没有检索到错误信息，直接返回
    IF ERROR_NUMBER() IS NULL
        RETURN;

    DECLARE
        @ErrorMessage NVARCHAR(4000),
        @ErrorNumber INT,
        @ErrorSeverity INT,
        @ErrorState INT,
        @ErrorLine INT,
        @ErrorProcedure NVARCHAR(200);

    -- 分配变量到错误处理函数，用于为 RAISERROR 捕获错误信息
    SELECT
        @ErrorNumber = ERROR_NUMBER(),
        @ErrorSeverity = ERROR_SEVERITY(),
        @ErrorState = ERROR_STATE(),
        @ErrorLine = ERROR_LINE(),
        @ErrorProcedure = ISNULL(ERROR_PROCEDURE(), '-');
```

```

-- 构建包含原始错误信息的字符串
SELECT @ErrorMessage =
    N'错误 %d, 级别 %d, 状态 %d, 过程 %s, 行 %d, ' +
    N'信息: ' + ERROR_MESSAGE();

-- 引发一个错误: RAISERROR 的 msg_str 参数将包含原始错误信息
RAISERROR
(
    @ErrorMessage,
    @ErrorSeverity,
    1,
    @ErrorNumber,      -- 参数: 原始错误编号
    @ErrorSeverity,    -- 参数: 原始错误严重级别
    @ErrorState,       -- 参数: 原始错误状态
    @ErrorProcedure,   -- 参数: 原始错误过程名称
    @ErrorLine         -- 参数: 原始错误行号
);

GO

-- 检验存储过程是否存在
IF OBJECT_ID (N'usp_GenerateError', N'P') IS NOT NULL
    DROP PROCEDURE usp_GenerateError;

GO

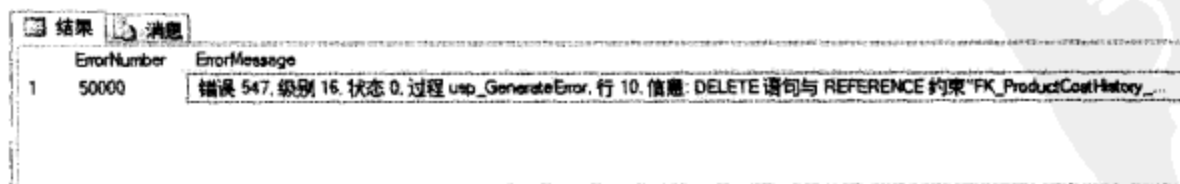
-- 创建存储过程, 用于生成一个违反约束错误。该错误将被相关
-- 联的 CATCH 块捕获, 并且会因为被执行 usp_RethrowError
-- 存储过程而再次引发错误 (注意: RAISERROR 用于生成错误,
-- 而不是生成错误消息)
CREATE PROCEDURE usp_GenerateError
AS
BEGIN TRY
    -- 表中存在一个外键约束, 下面的语句将生成一个违反约束错误
    DELETE FROM Production.Product
        WHERE ProductID = 980;
END TRY
BEGIN CATCH
    -- 调用存储过程来引发原始错误
    EXEC usp_RethrowError;
END CATCH;

GO

-- 在下面的批中, 在 usp_GenerateError 中发生的错误将激活
-- usp_GenerateError 中 CATCH 块, 该 CATCH 块中的 RAISERROR (包含
-- 在 usp_RethrowError 存储过程中) 将生成一个错误, 从而激活
-- 下面调用批中的外部 CATCH 块
BEGIN TRY -- outer TRY
    -- Call the procedure to generate an error.
    EXECUTE usp_GenerateError;
END TRY
BEGIN CATCH -- 外部 CATCH 块
    SELECT
        ERROR_NUMBER() as ErrorNumber,
        ERROR_MESSAGE() as ErrorMessage;
END CATCH;

```

图 13-3 是执行上述代码, 通过调用批中外部 CATCH 块的 SELECT 语句得到的错误信息。



ErrorNumber	ErrorMessage
50000	错误 547, 级别 16, 状态 0, 过程 usp_GenerateError, 行 10, 信息: DELETE 语句与 REFERENCE 约束 'FK_ProductCostHistory...' 冲突。

图 13-3 RAISERROR 语句生成的错误信息

13.3 CLR 存储过程

CLR 存储过程可以返回行集和信息到客户端，可以在 CLR 存储过程中调用数据定义语言 (DDL) 和数据管理语言 (DML)，并可以使用输出参数。

13.3.1 创建一个具有输出参数的 CLR 存储过程

要创建 CLR 存储过程，可以使用 .NET Framework 支持的语言（如 C#）中将存储过程定义为类的静态方法，然后编译该类生成程序集。然后在 Visual Studio 中启动部署，直接将程序集和上载到服务器并自动生成存储过程。也可以使用 CREATE ASSEMBLY 语句手动注册程序集，然后使用 CREATE PROCEDURE 语句创建引用注册程序集的存储过程。

1. 通过 Visual Studio 编写和部署 CLR 存储过程

打开 Visual Studio，从菜单中依次选择“文件”→“新建”→“项目”，打开如图 13-4 所示的“新建项目”对话框。选定 Visual C# 下的“数据库”节点，并在对话框的下方指定项目名称和存储位置后，单击“确定”按钮。

打开如图 13-5 所示的“新建数据库引用”对话框，指定要连接的服务器和数据库名称，并单击“确定”按钮。

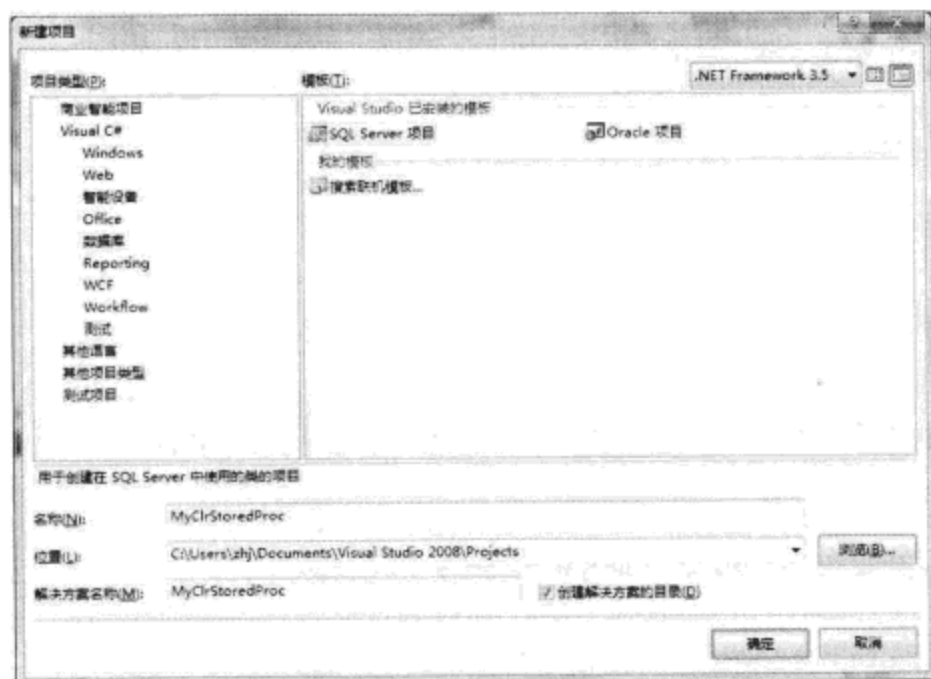


图 13-4 “新建项目”对话框

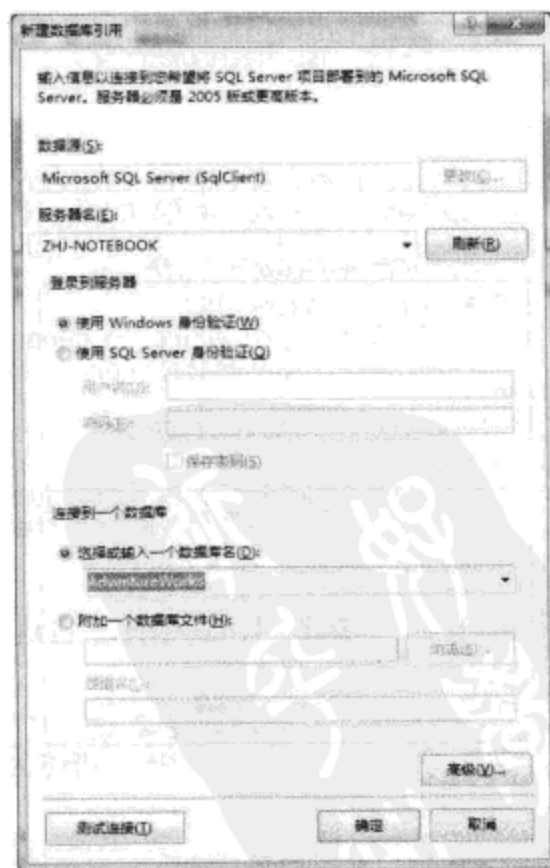


图 13-5 指定要连接的数据

从 Visual Studio 的菜单中依次选择“项目”→“添加存储过程”，打开如图 13-6 所示的“添加新项”对话框，在“名称”文本框中为存储过程指定一个名称，如 OrderQtySum.cs。在 Visual Studio 中，该名称体现为一个静态方法的名称。

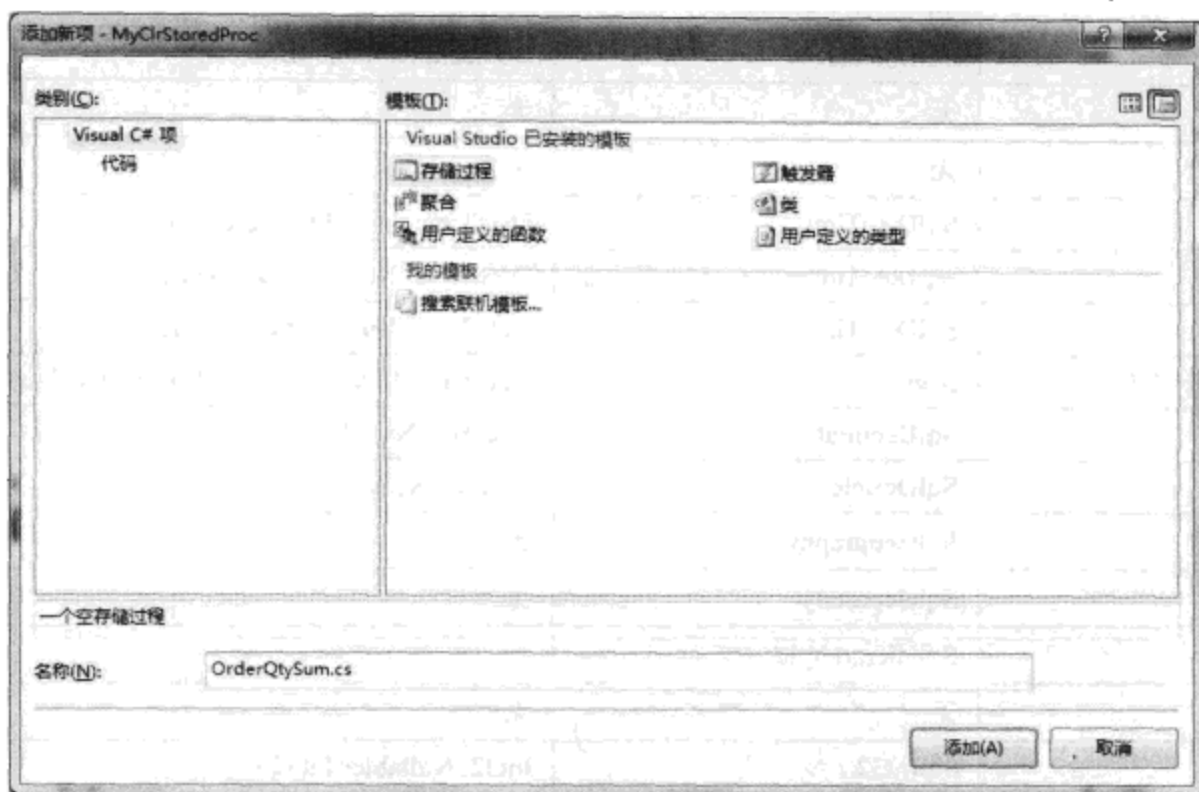


图 13-6 “添加新项”对话框

单击“添加”按钮后，Visual Studio 将自动在存储过程中添加如下代码：

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

public partial class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void OrderQtySum()
    {
        // 在此处放置代码
    }
};
```

这段代码的基本结构是由一系列 using 命令、一个类（StoredProcedures）和类中的方法（OrderQtySum）组成，它是由“存储过程”模板创建的。注意其中的 partial 关键词，表示要使用多个物理文件来存储类定义。如果在日后需要拆分类，这个关键词很有必要。

与 SQL 存储过程相同，可以在参数名称的前面加上 out 关键字，表示这是一个输出参数。在定义参数的数据类型时，应当保持与表中相关列的数据类型一致。SQL Server 与 CLR 数据类型的对应关系如表 13-2 所示。其中的 CLR 数据类型（SQL Server）在 System.Data.SqlTypes 命名空间定义。

表 13-2

SQL Server 与 CLR 数据类型的对应关系

SQL Server 数据类型	CLR 数据类型 (SQL Server)	CLR 数据类型 (.NET Framework)
bigint	SqlInt64	Int64, Nullable<Int64>
binary	SqlBytes, SqlBinary	Byte[]
bit	SqlBoolean	Boolean, Nullable<Boolean>
char	无	无
cursor	无	无
date	SqlDateTime	DateTime, Nullable<DateTime>
datetime	SqlDateTime	DateTime, Nullable<DateTime>
datetime2	SqlDateTime	DateTime, Nullable<DateTime>
DATETIMEOFFSET	None	DateTimeOffset, Nullable<DateTimeOffset>
decimal	SqlDecimal	Decimal, Nullable<Decimal>
float	SqlDouble	Double, Nullable<Double>
geography	SqlGeography	无
geometry	SqlGeometry	无
hierarchyid	SqlHierarchyId	无
image	无	无
int	SqlInt32	Int32, Nullable<Int32>
money	SqlMoney	Decimal, Nullable<Decimal>
nchar	SqlChars, SqlString	String, Char[]
ntext	无	无
numeric	SqlDecimal	Decimal, Nullable<Decimal>
nvarchar	SqlChars, SqlString	String, Char[]
nvarchar(1), nchar(1)	SqlChars, SqlString	Char, String, Char[], Nullable<char>
real	SqlSingle	Single, Nullable<Single>
rowversion	无	Byte[]
smallint	SqlInt16	Int16, Nullable<Int16>
smallmoney	SqlMoney	Decimal, Nullable<Decimal>
sql_variant	无	Object
table	无	无
text	无	无
time	TimeSpan	TimeSpan, Nullable<TimeSpan>
timestamp	无	无
tinyint	SqlByte	Byte, Nullable<Byte>
uniqueidentifier	SqlGuid	Guid, Nullable<Guid>
User-defined type(UDT)	无	绑定到相同程序集或依赖程序集中的用户定义类型的相同类

续表

SQL Server 数据类型	CLR 数据类型 (SQL Server)	CLR 数据类型 (.NET Framework)
varbinary	SqlBytes, SqlBinary	Byte[]
varbinary(1), binary(1)	SqlBytes, SqlBinary	byte, Byte[], Nullable<byte>
varchar	无	无
xml	SqlXml	无

下面是 OrderQtySum 存储过程的代码，用于计算 Sales.SalesOrderDetail 表中前 10 行销售产品的定货数量，并将计算结果返回给调用方。

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

public partial class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void OrderQtySum(out SqlInt16 value) //定义输出参数，类型为 SqlInt16
    {
        //设置连接
        SqlConnection conn = new SqlConnection();
        conn.ConnectionString = "Context Connection=true";
        conn.Open();

        //设置命令
        SqlCommand cmd = new SqlCommand();
        cmd.Connection = conn;
        cmd.CommandText = "SELECT TOP 10 OrderQty FROM Sales.SalesOrderDetail";

        //以只进方式读取 SQL Server 中的数据，并累加
        value = 0;
        SqlDataReader reader = cmd.ExecuteReader();

        while (reader.Read())
        {
            value += reader.GetSqlInt16(0); //获取指定列的值(0表示第1列，依次类推)
        }
    }
}
```

在 Visual Studio 的菜单中依次选择“生成”→“生成 MyClrStoredProc”，对 MyClrStoredProc 项目进行编译。再依次选择“生成”→“部署 MyClrStoredProc”，将把程序集和存储过程部署到 SQL Server 中去。

打开 SQL Server Management Studio，展开 AdventureWorks 数据库中“可编程性”节点下的“存储过程”和“程序集”节点，可以看到已经部署的 MyClrStoredProc 程序集和 dbo.OrderQtySum 存储过程，如图 13-7 所示。

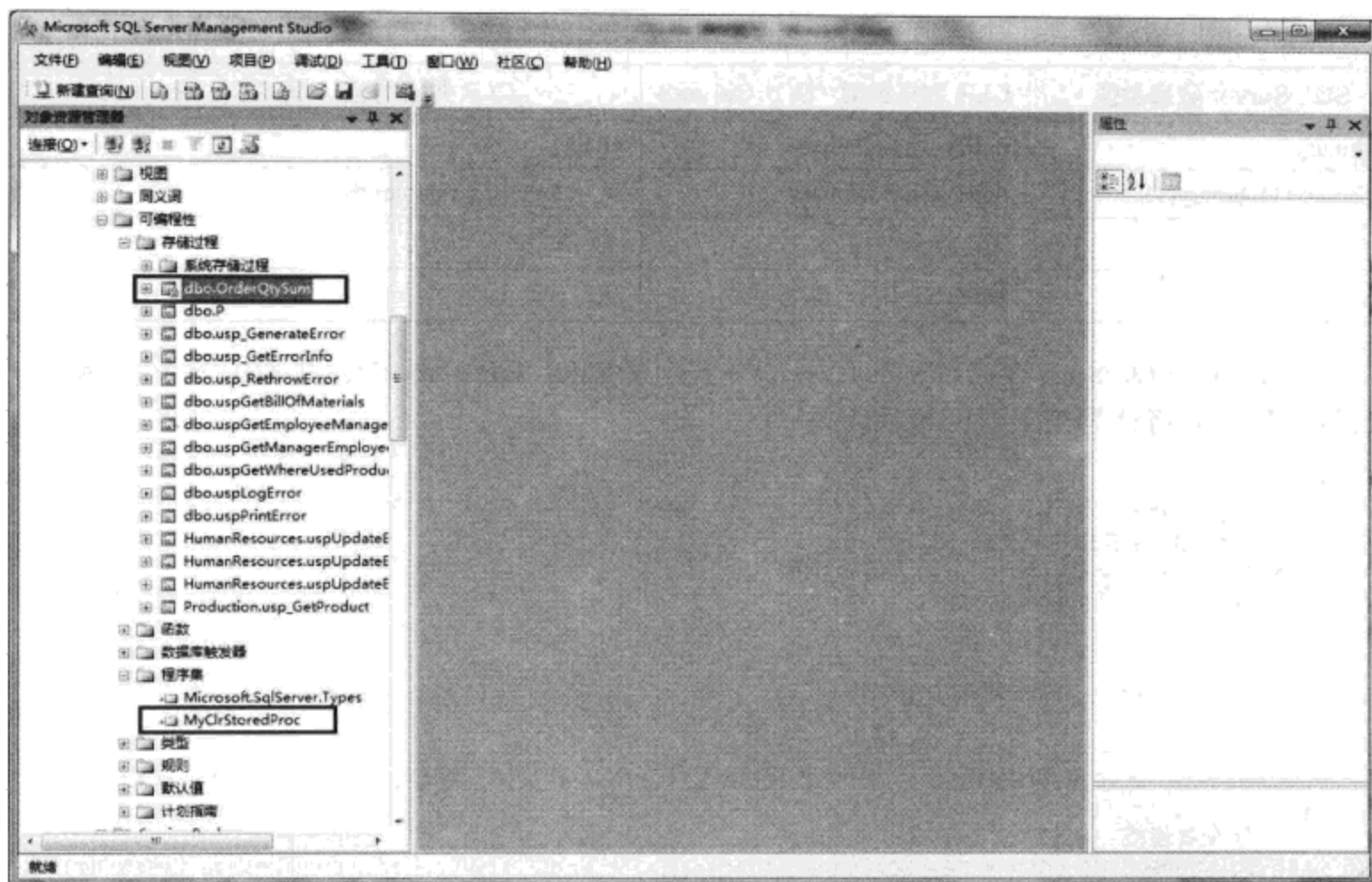


图 13-7 已部署的程序集和存储过程

部署成功后，可以从 `sys.assembly_modules` 视图中检索到已注册组件的信息，如图 13-8 所示。

结果		消息				
object_id	assembly_id	assembly_class	assembly_method	null_on_null_input	execute_as_principal_id	
1	2027154267	65536	StoredProcedures	OrderQtySum	0	NULL

图 13-8 sys.assembly_modules 视图中的组件信息

2. 使用 SQL 添加程序集和创建 CLR 存储过程

应当首先使用 `CREATE ASSEMBLY` 语句将包含托管代码的程序集在 SQL Server 中注册，才可以使使用 `CREATE PROCEDURE` 语句创建引用注册程序集的存储过程。参考下面的语句：

```
USE AdventureWorks;
GO
CREATE ASSEMBLY MyClrStoredProc
FROM 'C:\Users\zhj\Documents\Visual Studio 2008\Projects\MyClrStoredProc\MyClrStoredProc\bin\Debug\
MyClrStoredProc.dll'
WITH PERMISSION_SET = SAFE;
GO

-- 创建基于程序集的存储过程
CREATE PROCEDURE OrderQtySum
```

```

    @value smallint OUTPUT
WITH EXECUTE AS CALLER -- 指定执行权限
AS
    EXTERNAL NAME MyClrStoredProc.StoredProcedures.OrderQtySum; -- 指定要使用的方法名称

```

创建基于程序集的存储过程与普通的 SQL 存储过程有所不同。上面语句中的 **EXECUTE AS CALLER** 指定执行 CLR 模块的用户不仅必须对模块本身拥有适当的权限，还要对模块引用的任何数据库对象拥有适当权限。

EXTERNAL NAME 子句指定要使用 .NET Framework 程序集中的哪个方法，以便 CLR 存储过程引用。方法名称应当按照“程序集.类.方法”这样的对象层次关系进行声明。

此外，还需要注意的是，.NET Framework 中方法的参数数量应当与在 Transact-SQL 存储过程中定义的参数数量和类型要相同。

3. 执行 CLR 存储过程

执行 CLR 存储过程与执行 SQL 存储过程的方法完全相同。但是，在默认情况下，SQL Server 不允许执行 CLR 代码。只有在启用了 **clr enabled** 选项之后，才能执行这些引用。要启用该选项，可以使用 **sp_configure** 系统存储过程，参考下面的语句：

```

sp_configure 'clr enabled',1          -- 1 表示允许执行 CLR
RECONFIGURE WITH OVERRIDE -- 使用已更改的配置值更新当前的运行值，使修改立即生效

```

下面是执行在上面建立的 **OrderQtySum** 存储过程的代码：

```

DECLARE @a smallint -- 定义变量，用于存放输出值
EXECUTE OrderQtySum @a OUTPUT -- 使用 OUTPUT 关键字，表示是一个输出参数
SELECT @a -- 显示变量值

```

13.3.2 创建返回行集和信息的 CLR 存储过程

要返回行集和信息，应当使用 **SqlPipe** 对象来完成。可以通过 **SqlContext** 类的 **Pipe** 属性得到一个 **SqlPipe** 对象，该对象具有一个 **Send** 方法，用于传递数据到调用方。

可以使用 **SqlPipe.Send** 方法发送一个 **SqlDataReader**，以实现行集传递，也可以使用该方法发送一个简单的字符串信息。可以发送的最大字符数是 8000 个字符。

也可以使用 **SqlPipe** 对象 **ExecuteAndSend** 方法发送一个行集到客户端。在传输数据时，数据会被直接传递到网络缓冲区，而不用先复制到可管理内存中，因此这也是一种最直接的方法。

例如，下面的代码创建了一个 **ExecuteToClient** 过程，可以接收一个 **SqlInt32** 型参数，用于检索 **Sales.SalesOrderDetail** 表中指定销售订单中的产品列表。

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;

```



```

using Microsoft.SqlServer.Server;

public partial class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void ExecuteToClient(SqlInt32 OrderID)
    {
        //设置连接
        SqlConnection conn = new SqlConnection();
        conn.ConnectionString = "Context Connection=true";
        conn.Open();


        //设置命令
        SqlCommand cmd = new SqlCommand();
        cmd.Connection = conn;
        cmd.CommandText = "SELECT ProductID, OrderQty, UnitPrice FROM Sales.SalesOrderDetail "+
                           "WHERE SalesOrderID = @OrderID";
        cmd.Parameters.AddWithValue("@OrderID", OrderID); //设置 cmd 所使用的参数的值

        SqlContext.Pipe.ExecuteAndSend(cmd); //传递到客户端
    }
};

```

通过 Visual Studio 进行部署后，可以在 SQL Server Management Studio 中使用以下语句检索订单编号为 43659 的产品列表，得到的结果集合如图 13-9 所示。

```
EXECUTE dbo.ExecuteToClient 43659
```



	ProductID	OrderQty	UnitPrice
1	776	1	2024.994
2	777	3	2024.994
3	778	1	2024.994
4	771	1	2039.994
5	772	1	2039.994
6	773	2	2039.994
7	774	1	2039.994
8	714	3	28.8404
9	716	1	28.8404
10	709	6	5.70
11	712	2	5.1865
12	711	4	20.1865

图 13-9 使用 ExecuteToClient 存储过程得到的订单 43659 的产品列表

使用 `SqlPipe.Send` 方法发送一个 `SqlDataReader` 到客户端，比 `ExecuteAndSend` 方法稍微慢一些，但是，在数据传递给客户端之前它为数据操作提供了更大的灵活性。参考下面的代码：

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

public partial class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void SendReaderToClient(SqlInt32 OrderID)

```

```

{
    //设置连接
    SqlConnection conn = new SqlConnection();
    conn.ConnectionString = "Context Connection=true";
    conn.Open();

    //设置命令
    SqlCommand cmd = new SqlCommand();
    cmd.Connection = conn;
    cmd.CommandText = "SELECT ProductID, OrderQty, UnitPrice FROM Sales.SalesOrderDetail "+
        "WHERE SalesOrderID = @OrderID";
    cmd.Parameters.AddWithValue("@OrderID", OrderID); //设置 cmd 所使用的参数的值

    SqlDataReader r = cmd.ExecuteReader(); //生成 SqlDataReader

    SqlContext.Pipe.Send(r); //传递到客户端
}
};

```

使用 `SqlPipe.Send` 方法发送一个字符串比较简单，参考下面的代码：

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

public partial class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void SayHello()
    {
        SqlContext.Pipe.Send("Hello world! It's now " + System.DateTime.Now.ToString() + "\n");
    }
};

```

13.3.3 删除 CLR 存储过程和程序集

删除一个 CLR 存储过程与 Transact-SQL 存储过程没有任何不同。在删除 CLR 存储过程之后，才可以删除与之相应的程序集。参考下面的语句：

```

USE AdventureWorks;
GO

-- 删除存储过程
DROP PROCEDURE dbo.OrderQtySum;
DROP PROCEDURE dbo.ExecuteToClient;
DROP PROCEDURE dbo.SendReaderToClient;
DROP PROCEDURE dbo.SayHello;
-- 删除程序集
DROP ASSEMBLY MyClrStoredProc;
GO

```

13.3.4 CLR 与 SQL 存储过程的择取建议

在本节中我们介绍了 CLR 存储过程的创建方法。从中可以看出，CLR 存储过程与 SQL 存储过程在功能方面几乎没有差异。好象仅仅是一个使用 .NET 语言编写，而另一个使用 SQL 语言编写。下面将分析一下二者之间的差别，并给出一些合理化的建议。

1. 从语言功能分析

SQL 包含有查询语言（由 SELECT/INSERT/UPDATE/ DELETE 语句组成）和过程控制语言（WHILE、赋值、触发器、光标等），从这一方面说，SQL 在逻辑处理、流程控制方面基本上已经完全够用。CLR 支持只不过是过程语言提供了一种替代方法。

作为数据库应用程序，应该尽可能多地使用查询语言，不应当使用 CLR 来编写可以使用简单 SELECT 语句就可以表示的过程代码。试图使用 CLR 功能的开发人员应该首先确保已经充分地利用了查询语言，只有对于在查询语言中无法以声明方式表示的逻辑，才可以考虑将 CLR 作为有效的替代办法。

2. 从性能分析

在某些使用复杂计算的情况下，CLR 代码的开发人员可以利用 .NET Framework API 丰富的类/函数库，补充 SQL 查询语言的表达能力。此外，CLR 编程语言提供了 SQL 中所没有的丰富构造（例如数组和列表等）。与 SQL 相比，CLR 编程语言之所以具有更好的性能，是因为托管代码是已编译的，而 SQL 是一种解释语言。对于涉及算术计算、字符串处理、条件逻辑等的操作，托管代码的性能可能要优于 SQL 一个数量级。然而，对于数据访问方面，SQL 在性能方面通常会更好。

3. 从数据访问编程中的典型操作分析

（1）将结果发送到客户端。

使用 SQL，只需一个 SELECT 语句，就可以将 SELECT 产生的行发送到客户端。通过托管代码，可以使用 SqlPipe 对象将结果发送到客户端。SQL 和 in-proc ADO.NET 平台在这种情况下作用是一样的。

（2）提交 SQL 语句。

这包括来自过程代码的 SQL 语句的执行往返。在这种情况下，SQL 具有很大的优势（比 in-proc ADO.NET 快两倍多）。之所以在 CLR 中出现性能降低，是因为增加了额外的代码层，包括将来自托管代码的 SQL 语句提交给 SQL Server，并进行语法检验和编译。

（3）只进、只读行导航。

在 SQL 中, 此功能需要通过只进、只读光标实现。在 CLR 中, 这是通过 `SqlDataReader` 实现的。通常, 每一条语句都涉及一些处理。如果忽略了与每行相关联的处理, 则导航行在 CLR 中就比较在 SQL 光标中稍慢。然而, 如果关心为每行执行的处理, 则 CLR 会更有优势, 因为 CLR 在这种处理上比 SQL 做得好。

(4) 带有更新的行导航。

如果需要根据光标中的当前位置更新行, 则没有相关的性能比较, 因为 in-proc ADO.NET 不支持此功能, 而应该通过 SQL 可更新光标来进行此操作。

4. 选择 CLR 和 Transact-SQL 的指导原则

尽可能使用带有 SQL `SELECT`、`INSERT`、`UPDATE` 和 `DELETE` 语句的基于集的处理。只有在无法使用基于集的 DML 语句表示逻辑时, 才应该使用过程和基于行的处理。

如果过程仅仅是一个通过封装基本 `INSERT/UPDATE/DELETE/SELECT` 操作访问基表的包装, 则应该用 SQL 进行编写。

如果过程主要包括结果集中的只进、只读行导航, 以及一些涉及每行的处理, 则用 CLR 编写可能更有效。

如果过程包括大量的数据访问以及计算和逻辑, 则可以考虑将过程代码分隔为 CLR 来调用 SQL 过程, 以进行大部分的数据访问 (反之亦然)。另一个替代方法是, 使用 SQL 批处理, 从而减少从托管代码提交 SQL 语句的往返次数。

13.4 嵌套存储过程

所谓嵌套存储过程, 是指从一个存储过程调用另一个存储过程或执行托管代码。嵌套存储过程和托管代码引用最高可达 32 级。每当调用的存储过程或托管代码引用开始执行, 嵌套级别就增加一级; 执行完成后, 嵌套级别就减少一级。但是, 从托管代码内部调用的方法不根据此限制进行计数。

存储过程可以通过嵌套调用自身, 这种技术称为递归。

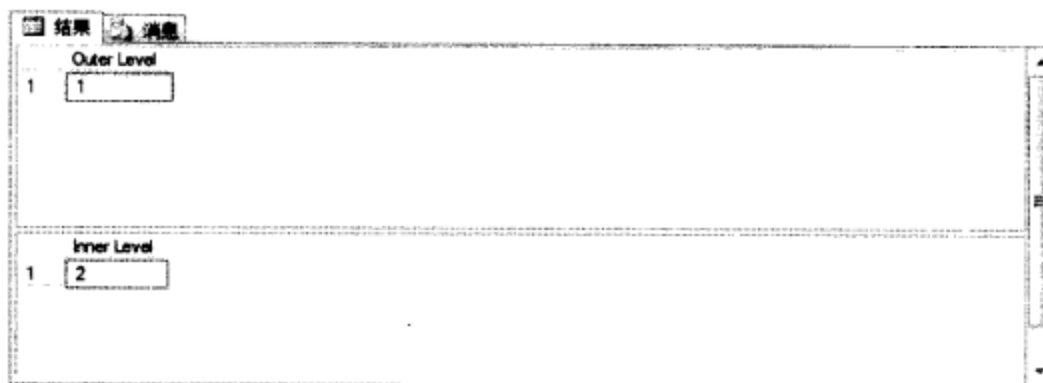
尽管嵌套限制为 32 级, 但是对在给定的存储过程中可以调用的存储过程数量没有限制。因此, 只要从属存储过程不调用其他从属存储过程且不超过最大嵌套级别即可。

可以使用 `@@NESTLEVEL` 函数返回当前的嵌套级别。在 SQL 中使用 `EXEC` 执行 `@@NESTLEVEL` 时, 返回的值为 1 加当前嵌套级别。使用 `sp_executesql` 动态执行 `@@NESTLEVEL` 时, 返回的值为 2 加当前嵌套级别。

下面的示例将创建两个过程, 用于进行嵌套。每个过程都显示当前过程的 `@@NESTLEVEL` 值。

```
USE AdventureWorks;
GO
IF OBJECT_ID (N'usp_OuterProc', N'P') IS NOT NULL
    DROP PROCEDURE usp_OuterProc;
GO
IF OBJECT_ID (N'usp_InnerProc', N'P') IS NOT NULL
    DROP PROCEDURE usp_InnerProc;
GO
CREATE PROCEDURE usp_InnerProc AS
    SELECT @@NESTLEVEL AS 'Inner Level';
GO
CREATE PROCEDURE usp_OuterProc AS
    SELECT @@NESTLEVEL AS 'Outer Level';
    EXEC usp_InnerProc;
GO
EXECUTE usp_OuterProc;
```

执行结果如图 13-10 所示。



Outer Level	
1	1

Inner Level	
1	2

图 13-10 嵌套存储过程执行结果

以下示例则演示了使用 SELECT、EXEC 和 sp_executesql 调用 @@NESTLEVEL 时，它们返回的值的区别。

```
CREATE PROCEDURE usp_NestLevelValues
AS
    SELECT @@NESTLEVEL AS 'Current Nest Level';
    EXEC ('SELECT @@NESTLEVEL AS OneGreater');
    EXEC sp_executesql N'SELECT @@NESTLEVEL as TwoGreater' ;
GO
EXEC usp_NestLevelValues;
```

执行结果如图 13-11 所示。



Current Nest Level	
1	1

OneGreater	
1	2

TwoGreater	
1	3

图 13-11 SELECT、EXEC 和 sp_executesql 的执行结果



第 14 章 触发器

触发器是为执行业务规则和保持数据完整性而提供的一种机制，它可以在执行插入、更新、删除等操作前/后自动触发。触发器与存储过程类似，但是它不能接收输入/输出参数，也不能被显示调用，只能是由服务器事件自动触发，根据引起执行触发器操作的语言不同，可以将其分为 DML 触发器和 DDL 触发器。

14.1 DML 触发器

根据 DML 触发器发生的时间、编写触发器所使用的语言，可以分为 AFTER 触发器、INSTEAD OF 触发器和 CLR 触发器。AFTER 触发器在执行 INSERT、UPDATE 或 DELETE 语句操作之后、INSTEAD OF 触发器和约束之后激发。INSTEAD OF 在处理约束前激发，因此可以在 INSTEAD OF 中使用其他语句来替代激发触发器的 INSERT、UPDATE 等语句。并且，还可为基于一个或多个基表的视图定义 INSTEAD OF 触发器，从而扩展视图可支持的更新类型。CLR 触发器可以是 AFTER 触发器或 INSTEAD OF 触发器，并且也可以是 DDL 触发器。

需要注意的是，在创建 DML 触发器时，不能使用下列语句：

ALTER DATABASE	CREATE DATABASE	DROP DATABASE
LOAD DATABASE	LOAD LOG	RECONFIGURE
RESTORE DATABASE	RESTORE LOG	

14.1.1 AFTER 触发器

一个表中可以具有多个 AFTER 触发器，只要它们的名称不相同即可。每个触发器只能应用于一个表，但是一个触发器可以同时应用于一个表的 3 个用户操作（UPDATE、INSERT 和 DELETE）。

下面的语句创建了一个 PriTrigger 表和一个 DetailTable，其中 PriTrigger 表用于存放销售订单的编号和金额，DetailTable 表用于存放每笔订单中的产品信息。为 PriTrigger 表的 DELETE 操作创建了一个名为 PriTrigger 的触发器，当删除 PriTrigger 表中的订单信息时，该触发器将删除 DetailTable 表中该笔订单的产品信息。

```

USE AdventureWorks;
GO
-- 创建主表, 存放销售订单编号和金额
CREATE TABLE PriTable
(OrderID int IDENTITY(1,1), OrderTotal money);
GO
-- 创建明细表, 存放每笔订单中的产品信息
CREATE TABLE DetailTable
(OrderID int, ProductID int, ProductCount int NOT NULL, Price money);
GO

-- 向主表中插入订单信息
INSERT INTO PriTable VALUES (2100.00);
INSERT INTO PriTable VALUES (1000.00);
-- 向明细表中插入订单的产品信息
INSERT INTO DetailTable VALUES (1,1,10,110.00);
INSERT INTO DetailTable VALUES (1,2,10,100.00);
INSERT INTO DetailTable VALUES (2,2,10,100.00);
GO

-- 为 PriTrigger 表创建触发器
CREATE TRIGGER PriTrigger
ON PriTable
AFTER DELETE
AS
    DELETE FROM DetailTable
    WHERE OrderID IN (SELECT OrderID
                      FROM Deleted);
    PRINT N'已经删除了 DetailTable 表中的相关数据' -- 此句仅为演示需要, 在触发器中不应当使用这样的信息
语句

```

在定义触发器时, 触发器名称在 **CREATE TRIGGER** 关键字之后, **ON** 子句指定要创建触发器的基表。**AFTER** 子句 (也可以使用 **FOR** 来代替 **AFTER** 关键字, 二者功能相同) 指定激活触发器的操作语句, 可以同时指定多个操作语句。例如, “**AFTER DELETE, INSERT**” 表示在对表执行 **DELETE**、**INSERT** 语句时激活触发器。**AS** 关键字后指定触发器执行什么样的操作。

注意 **WHERE** 条件中 **IN** 子句中的 **Deleted** 关键字。当从 **PriTrigger** 表中删除行时, 被删除的行会被复制到一个名为 **Deleted** 的临时内存表中。如果为表指定了一个执行 **INSERT** 语句时的触发器, 则在向表中插入行时, 新行将同时被添加到一个名为 **Inserted** 的临时内存表中。如果为表指定了一个执行 **UPDATE** 语句时的触发器, 由于更新事务类似于在删除操作之后执行插入操作。因此, 旧行被复制到 **Deleted** 表中, 然后, 新行被复制到触发器表和 **Inserted** 表中。

Deleted 表和 **Inserted** 表都是由数据库引擎自动创建和管理的, 这些表的结构与定义触发器的基表的结构相同。

执行下面的语句从 **PriTable** 表中删除 **OrderID** 为 1 的行, 这时触发器会自动删除 **DetailTable** 表中的相关行, 得到的结果和消息如图 14-1 所示。

```

DELETE FROM PriTable WHERE OrderID = 1;
SELECT * FROM PriTable;
SELECT * FROM DetailTable;

```

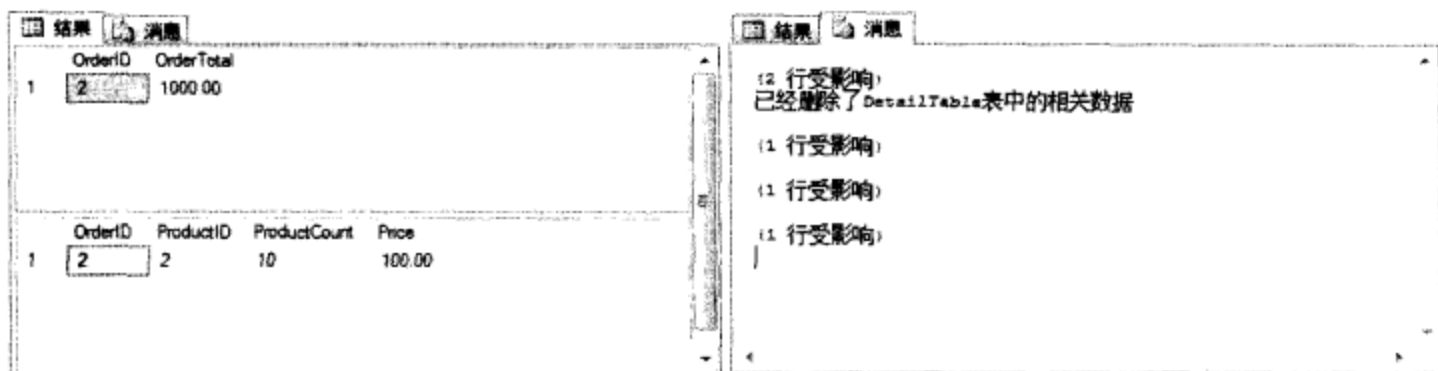


图 14-1 删除 PriTable 表中 OrderID 为 1 的行时得到的结果和消息

如果执行下面的语句，准备从 PriTable 表中删除 OrderID 为 3 的行。由于 PriTable 表中并不存在这样的行，所以并不会删除成功。虽然没有删除成功，但是在消息窗口中仍然可以看到由触发器的 PRINT 语句发回的信息。这说明即使语句没有影响到表中的行，也会激活触发器。在没有删除成功的情况下，Deleted 表是一个空表。

```
DELETE FROM PriTable WHERE OrderID = 3;
```

14.1.2 进行事务提交和回滚操作

当执行能够激发触发器操作的语句时，触发器中的操作也将包含在该语句的事务处理过程中。即使在自动事务处理模式下，也是如此。在自动事务处理模式下，当语句遇到错误时，会有隐含的 BEGIN TRANSACTION 语句来回滚该语句所影响的修改。但是，这个回滚操作对批处理中的其他语句没有影响，不会回滚前面操作正常的语句。因为当语句完成时，该事务要么提交，要么回滚，事务处理已经结束。但是，当该语句激发触发器时，这个隐含事务将一直有效，直至触发器操作完成。因此，这也就给予了在触发器中回滚该语句操作的机会。需要注意的是，这个回滚操作也会终止批中对该语句后面语句的执行。

例如，以下示例为 DetailTable 表的 INSERT 和 UPDATE 操作定义了触发器，当向表中插入新行或更新行时，要求 ProductID、ProductCount 和 Price 列的值不能为 0，否则将执行 ROLLBACK TRANSACTION 进行回滚。

```

USE Adventureworks;
GO

-- 创建表
CREATE TABLE DetailTable
(OrderID int, ProductID int, ProductCount int NOT NULL, Price money);
GO

-- 为 DetailTable 表的 INSERT 和 UPDATE 操作定义触发器
CREATE TRIGGER DetailTrigger
ON DetailTable
AFTER INSERT, UPDATE
AS
    SET NOCOUNT ON
    IF (EXISTS(SELECT *
                FROM Inserted
  
```

```

        WHERE ProductID = 0 OR ProductCount = 0 OR Price = 0))
    BEGIN
        ROLLBACK TRANSACTION ;
    END
GO

INSERT INTO DetailTable VALUES(1,1,10,1000.00); -- 插入一行数据

```

执行下面的语句。在自动事务处理模式下，由于 UPDATE 语句没有违反 ProductID、ProductCount 和 Price 列的值不为 0 的规则，因此这句可以正常提交。下面的 INSERT 语句由于设置 ProductID 列的值为 0，因此会执行触发器中的 ROLLBACK TRANSACTION 语句，被回滚操作。但是，由于 UPDATE 语句已经正常提交，所以对于 INSERT 语句的回滚操作不会涉及到它。由于执行了回滚操作，将终止对批处理中剩余语句的执行，下面的 DELETE 语句不会被执行。

```

UPDATE DetailTable
    SET ProductCount = 100
    WHERE OrderID = 1 AND ProductID = 1;
INSERT INTO DetailTable VALUES(2,0,10,1000.00);
DELETE FROM DetailTable
    WHERE OrderID = 1;

```

对于显式或隐式事务处理，触发器中的 ROLLBACK TRANSACTION 语句将回滚整个事务处理。例如，下面的示例将 UPDATE 和 INSERT 语句包含在了一个显式事务处理中，由 INSERT 语句引发的回滚操作也将回滚 UPDATE 语句所做的修改。

```

BEGIN TRANSACTION
UPDATE DetailTable
    SET ProductCount = 100
    WHERE OrderID = 1 AND ProductID = 1;
INSERT INTO DetailTable VALUES(2,0,10,1000.00);
COMMIT TRANSACTION

```

从前面的分析也可以看出，无论是在自动事务处理模式下，还是在隐式或显式事务处理模式下，只要在触发器中发出 BEGIN TRANSACTION 语句，实际上就开始了一个嵌套事务。当从触发器中使用 ROLLBACK TRANSACTION 语句回滚嵌套事务时，触发器本身发出的所有 BEGIN TRANSACTION 语句都将被忽略，ROLLBACK 将回滚到最外部的 BEGIN TRANSACTION。

如果要在触发器中进行部分回滚，应当使用 SAVE TRANSACTION 语句设置一个事务保存点。例如：

```

CREATE TRIGGER Trigger1
ON MyTable
AFTER UPDATE
AS
    SAVE TRANSACTION MyName
    INSERT INTO MyTable1
        SELECT * FROM inserted;
    IF (@@error <> 0)
    BEGIN
        ROLLBACK TRANSACTION MyName;
    END

```

在触发器中使用 BEGIN TRANSACTION 语句的情况下，其后面的 COMMIT TRANSACTION

语句只应用于该嵌套事务。如果在 COMMIT TRANSACTION 之后执行 ROLLBACK TRANSACTION 语句，则 ROLLBACK 仍旧一直回滚到最外部的 BEGIN TRANSACTION。因此，建议不要在触发器中使用 COMMIT TRANSACTION 语句。

例如，下面的示例为 DetailTable 表创建了一个用于 INSERT 操作的触发器，当向 DetailTable 表中插入行时，该行将被复制到 DetailTable1 表中。虽然在触发器内已经使用 COMMIT TRANSACTION 语句进行了提交，但是后面的 ROLLBACK TRANSACTION 语句仍旧回滚所有嵌套操作。DetailTable 表始终不能插入任何数据。

```
USE AdventureWorks;
GO

-- 创建表
CREATE TABLE DetailTable
(OrderID int, ProductID int, ProductCount int NOT NULL, Price money);
CREATE TABLE DetailTable1
(OrderID int, ProductID int, ProductCount int NOT NULL, Price money);
GO

-- 创建触发器，将插入到 DetailTable 表的行复制到 DetailTable1 中
CREATE TRIGGER TransTrigger
ON DetailTable
AFTER INSERT
AS
    BEGIN TRANSACTION
    INSERT INTO DetailTable1
        SELECT * FROM Inserted;
    COMMIT TRANSACTION
    ROLLBACK TRANSACTION
```

可以使用下面的语句测试插入情况：

```
INSERT INTO DetailTable VALUES(1,1,10,1000.00);
GO
SELECT * FROM DetailTable;
SELECT * FROM DetailTable1;
```

14.1.3 检测对指定列的 UPDATE 或 INSERT 操作

对于 INSERT 或 UPDATE 触发器，可以使用 UPDATE() 或 COLUMNS_UPDATED() 函数来检测对列的修改，从而据以执行相应的操作。其中，UPDATE() 函数可以测试对某个列的 UPDATE 或 INSERT 尝试。COLUMNS_UPDATED() 可以测试对多个列执行的 UPDATE 或 INSERT 操作。

1. 使用 UPDATE() 测试指定列

下面的示例为 DetailTable 表定义了一个用于 UPDATE 操作的触发器。当修改 DetailTable 表中的 ProductID 列或 ProductCount 列时，将把修改前和修改后的数据复制到 DetailTable1 表中。

```
-- 创建表
CREATE TABLE DetailTable
(OrderID int, ProductID int, ProductCount int NOT NULL, Price money);
```



```

CREATE TABLE DetailTable1
(OrderID int, ProductID int, ProductCount int NOT NULL, Price money);
GO

-- 插入行
INSERT INTO DetailTable VALUES(1,1,10,1000.00);
INSERT INTO DetailTable VALUES(2,1,10,1000.00);
GO

-- 创建触发器
CREATE TRIGGER Trigger1
ON DetailTable
AFTER UPDATE
AS
    IF (UPDATE(ProductID) OR UPDATE(ProductCount))
    BEGIN
        INSERT INTO DetailTable1
            SELECT * FROM Deleted; -- 将更新前的数据复制到 DetailTable1 中
        INSERT INTO DetailTable1
            SELECT * FROM Inserted; -- 将更新后的数据复制到 DetailTable1 中
    END

```

下面的 UPDATE 语句将 DetailTable 表中 OrderID 为 1 行中的 ProductID 修改为 2（修改前的值为 1）。从 DetailTable1 表中可以看到修改前和修改后的行都被复制了进来，如图 14-2 所示。

```

UPDATE DetailTable
    SET ProductID = 2
    WHERE OrderID = 1;
SELECT * FROM DetailTable1;

```



	OrderID	ProductID	ProductCount	Price
1	1	1	10	1000.00
2	1	2	10	1000.00

图 14-2 被复制到 DetailTable1 中的修改前和修改后的行

2. 使用 COLUMNS_UPDATED()测试多个列

COLUMNS_UPDATED()可以针对多列执行的 UPDATE 或 INSERT 操作的进行测试。COLUMNS_UPDATED 返回一个或多个从左至右排序的字节。表中的第 1 列由最左侧字节的最右侧位（即最低位）表示；第 2 列由向左的下一位表示，依次类推。如果创建了触发器的表包含 8 列以上，则 COLUMNS_UPDATED()返回多个字节。返回的字节数与表中的列数有关，而与更新的列数无关。

例如，假设 T1 表包含有 10 个列，分别是 C1、C2...C9 和 C10。假设对表中的 C2、C4 和 C9 列进行了更新，由于 T1 表超过了 8 列，所以 COLUMNS_UPDATED()将返回两个字节，以满足表达位值的要求，如图 14-3 所示。

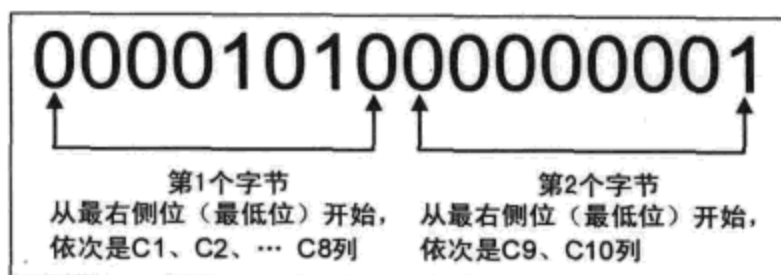


图 14-3 更新 C2、C4、C9 列时返回的位值

以下示例为 T1 表定义了一个用于 UPDATE 的触发器，当更新表中的 C2、C4 和 C6 列时，将返回提示消息。判断修改的方法是通过计算 COLUMNS_UPDATED() 返回的字节值实现的。例如，在 C2、C4 和 C6 列均被修改的情况下，COLUMNS_UPDATED() 返回字节中的二进制位是“00101010”，转换为十进制应当是 $2^5 + 2^3 + 2^1 = 42$ 。

```
USE AdventureWorks;
Go

-- 创建表
CREATE TABLE T1
  (C1 int,
   C2 int,
   C3 int,
   C4 int,
   C5 int,
   C6 int,
   C7 int);

-- 插入行
INSERT INTO T1 VALUES(1, 1, 1, 1, 1, 1, 1);
GO

-- 创建触发器
CREATE TRIGGER Trigger1
ON T1
AFTER UPDATE
AS
  IF (COLUMNS_UPDATED() = POWER(2,(2-1)) + POWER(2,(4-1)) + POWER(2,(6-1)))
    PRINT N'修改了第 2、4、6 列';

GO

-- 测试触发器
UPDATE T1
  SET C2 = 0, C4 = 0, C6 = 0
  WHERE C1 = 1
```

如果表中包含的列数超过了 8 列，COLUMNS_UPDATED 将返回多个字节，这时候应当对字节进行截取，以判断每个字节中二进制位的情况。例如，下面示例中的 T1 表包含有 10 列，所定义的触发器在修改表中 C2、C4 和 C10 列时，将返回提示消息。

```
USE AdventureWorks;
Go

-- 创建表
CREATE TABLE T1
  (C1 int,
```

```

        C2 int,
        C3 int,
        C4 int,
        C5 int,
        C6 int,
        C7 int,
        C8 int,
        C9 int,
        C10 int);

-- 插入行
INSERT INTO T1 VALUES(1, 1, 1, 1, 1, 1, 1, 1, 1, 1);
GO

-- 创建触发器
CREATE TRIGGER Trigger1
ON T1
AFTER UPDATE
AS
    DECLARE @colval binary(2); -- 定义变量存储 COLUMNS_UPDATED() 的返回值
    SET @colval = COLUMNS_UPDATED();
    IF (SUBSTRING(@colval, 1, 1) = POWER(2, (2-1)) + POWER(2, (4-1)) -- 判断存储在第 1 个字节中的表的第 2 列
和第 4 列
        AND SUBSTRING(@colval, 2, 1) = POWER(2, (2-1))) -- 判断存储在第 2 个字节中的表的第 10 列
        PRINT N'修改了第 2、4、10 列';

GO

-- 测试触发器
UPDATE T1
SET C2 = 0, C4 = 0, C10 = 0
WHERE C1 = 1

```

14.1.4 指定 First 和 Last 触发器

可以为一个表定义多个触发器，在某些时候，在执行这些触发器时可能需要一定的顺序。因此，允许为 INSERT、DELETE 或 UPDATE 操作指定激发的第一个和最后一个 AFTER 触发器，分别称之为 First 和 Last 触发器。每个语句类型只能有一个 First 触发器和一个 Last 触发器，并且 First 和 Last 触发器必须是两个不同的触发器。在 First 和 Last 触发器之间的 AFTER 触发器没有固定的执行顺序。

要指定 AFTER 触发器的顺序，应当使用 `sp_settriggerorder` 存储过程。例如，下面的语句设置 MyTrigger 触发器是应用于 UPDATE 语句的第一个触发器：

```
sp_settriggerorder @triggername = 'MyTrigger', @order = 'first', @stmttype = 'UPDATE'
```

由于 INSTEAD OF 触发器一直在对基础表进行更新前激发，因此，不能将 INSTEAD OF 触发器指定为第一个或最后一个触发器。

如果使用 ALTER TRIGGER 语句更改了 First 或 Last 触发器，则会删除它们的顺序值，必须使用 `sp_settriggerorder` 来重新设置。

可以通过 OBJECTPROPERTY()函数的 ExecIsFirstDeleteTrigger、ExecIsFirstInsertTrigger、ExecIsFirstUpdateTrigger 和 ExecIsLastDeleteTrigger、ExecIsLastInsertTrigger、ExecIsLastUpdateTrigger 属性来确定触发器是否为 First 触发器还是 Last 触发器。参考下面的语句：

```
sp_settriggerorder @triggername = 'Trigger1', @order = 'first', @stmttype = 'UPDATE'
SELECT OBJECTPROPERTY(OBJECT_ID(N'Trigger1'), 'ExecIsFirstUpdateTrigger'); -- 返回 1, 表示是用于 UPDATE
的 First 触发器
```

OBJECTPROPERTY()函数的第 1 个参数是数据库中对象的 ID, 可以通过 OBJECT_ID 函数根据指定的对象名称来获得 ID。第 2 个参数是要获取信息的属性名称。如果是, 返回 1, 否则返回 0。如果指定的属性名称不正确, 返回 NULL。

14.1.5 嵌套和递归触发器

1. 嵌套触发器

无论是 DML 触发器还是 DDL 触发器, 如果出现了一个触发器执行启动另一个触发器的操作, 都属于嵌套触发器。DML 触发器和 DDL 触发器最多可以嵌套 32 层, SQL 触发器中对托管代码的任何引用均计为 32 层嵌套限制中的一层, 从托管代码内部调用的方法不根据此限制进行计数。

可以通过 nested triggers 服务器配置选项来控制是否可以嵌套 AFTER 触发器。INSTEAD OF 触发器嵌套不受此选项影响。参考下面的语句：

```
sp_configure 'nested triggers'.1; -- 设置 nested triggers 选项为 1, 允许 AFTER 触发器嵌套
GO
RECONFIGURE; -- 使用新环境值
GO
EXEC sp_configure 'nested triggers'; -- 查看 nested triggers 选项设置
GO
```

如果嵌套触发器中的一个触发器启动了一个无限循环, 则将超出嵌套层限制, 触发器将被终止执行。

在下面的示例中, 当从 PriTable 表中删除订单信息时, PriDelTrigger 触发器将从 DetailTable 表中删除该笔订单的产品信息, 在删除这些信息的同时, DetailDelTrigger 触发器将保存被删除行到 DetailTableBak 表中。这种由一个触发器启动另一个触发器的操作, 就属于嵌套触发器。

```
USE AdventureWorks;
Go
-- 创建主表, 存放销售订单编号和金额
CREATE TABLE PriTable
(OrderID int IDENTITY(1,1), OrderTotal money);

-- 创建明细表, 存放每笔订单中的产品信息
CREATE TABLE DetailTable
(OrderID int, ProductID int, ProductCount int NOT NULL, Price money);

-- 创建备份表, 存储 DetailTable 中被删除的数据
CREATE TABLE DetailTableBak
```

```

(OrderID int, ProductID int, ProductCount int NOT NULL, Price money);
GO

-- 向主表中插入订单信息
INSERT INTO PriTable VALUES (2100.00);
INSERT INTO PriTable VALUES (1000.00);
-- 向明细表中插入订单的产品信息
INSERT INTO DetailTable VALUES (1,1,10,110.00);
INSERT INTO DetailTable VALUES (1,2,10,100.00);
INSERT INTO DetailTable VALUES (2,2,10,100.00);
GO

-- 为 PriTrigger 表创建触发器
CREATE TRIGGER PriDelTrigger
ON PriTable
AFTER DELETE
AS
    DELETE FROM DetailTable
    WHERE OrderID IN (SELECT OrderID
                     FROM Deleted);
GO

-- 为 DetailTable 表创建触发器, 保存被删除的数据到 DetailTableBak 表中
CREATE TRIGGER DetailDelTrigger
ON DetailTable
AFTER DELETE
AS
    INSERT INTO DetailTableBak
    SELECT * FROM Deleted;
GO

-- 测试触发器
DELETE FROM PriTable
WHERE OrderID = 1;
SELECT * FROM DetailTableBak; -- 查看被删除的行

```

2. 递归触发器

(1) 直接递归。

在触发器触发并执行一个导致同一个触发器再次触发的操作时, 将发生此递归。例如, 应用程序更新了表 T3, 从而触发了触发器 Trig3。Trig3 再次更新表 T3, 从而再次触发了触发器 Trig3。

(2) 间接递归。

触发器触发并执行另一个触发器的操作时, 该触发器却再次触发了第 1 个触发器, 这就发生了间接递归。例如, 应用程序更新了表 T1, 从而触发了触发器 Trig1。Trig1 更新了表 T2, 从而触发了触发器 Trig2。Trig2 转而更新了表 T1, 从而再次触发了 Trig1。

只有在设置 RECURSIVE_TRIGGERS 数据库选项为 ON 的情况下, 才允许以递归方式调用 AFTER 触发器。如:

```

ALTER DATABASE Adventureworks
SET RECURSIVE_TRIGGERS ON;

```

在将 RECURSIVE_TRIGGERS 数据库选项设置为 OFF 时, 仅防止直接递归。如果要禁用间接

递归，还应将 `nested triggers` 服务器选项设置为 0。

下面是一个使用递归触发器的示例，在使用个递归触发器时，应当能够有一个明确的条件来结束递归，防止出现死循环。表 `emp_mgr` 中包含 3 列，其中 `emp` 用于存储公司雇员的名称，`mgr` 用于存储每个雇员的经理，`mgcount` 存储每个经理管理的雇员总数。

用于 `UPDATE` 的 `emp_mgrupd` 触发器是一个递归触发器。当改变一个雇员的经理时，将执行触发器中的 `UPDATE` 语句，调整相应经理的管理人数。触发器中的这个 `UPDATE` 语句会再次触发 `emp_mgrupd` 触发器，形成递归。但是，由于触发器中包含有 `IF UPDATE (mgr)` 语句，所以在被递归触发时，不会再执行其中的 `UPDATE` 语句，递归结束。

```
USE AdventureWorks;
GO
-- 允许递归.
ALTER DATABASE AdventureWorks
    SET RECURSIVE_TRIGGERS ON;
GO

-- 创建表
CREATE TABLE emp_mgr (
    emp char(30) PRIMARY KEY,
    mgr char(30) NULL FOREIGN KEY REFERENCES emp_mgr(emp),
    mgcount int DEFAULT 0
);
GO

-- 创建 INSERT 触发器，当插入新行时，增加相应经理的管理人数
CREATE TRIGGER emp_mgrins
ON emp_mgr
AFTER INSERT
AS
    UPDATE emp_mgr
    SET emp_mgr.mgcount = emp_mgr.mgcount + 1
    FROM Inserted
    WHERE emp_mgr.emp = Inserted.mgr;

GO

-- 创建 UPDATE 触发器，当调整雇员的经理时，则相应调整经理的管理人数
-- 该触发器要求每次仅更新一名雇员的经理
CREATE TRIGGER emp_mgrupd
ON emp_mgr
AFTER UPDATE
AS
    IF UPDATE (mgr)
    BEGIN
        UPDATE emp_mgr
        SET emp_mgr.mgcount = emp_mgr.mgcount + 1
        FROM Inserted
        WHERE emp_mgr.emp = Inserted.mgr;

        UPDATE emp_mgr
        SET emp_mgr.mgcount = emp_mgr.mgcount - 1
        FROM Deleted
        WHERE emp_mgr.emp = Deleted.mgr;
    END
GO
```

```
-- 插入测试数据
INSERT emp_mgr(emp, mgr) VALUES ('Harry', NULL);
INSERT emp_mgr(emp, mgr) VALUES ('Alice', 'Harry');
INSERT emp_mgr(emp, mgr) VALUES ('Paul', 'Alice');
INSERT emp_mgr(emp, mgr) VALUES ('Joe', 'Alice');
INSERT emp_mgr(emp, mgr) VALUES ('Dave', 'Joe');
```

执行下面的语句，查看 emp_mgr 表中的数据，以及将 Dave 的经理由 Joe 改变为 Harry 后的数据变化情况，如图 14-4 所示。

```
SELECT * FROM emp_mgr;
GO
-- 将 Dave 的经理由 Joe 改变为 Harry
UPDATE emp_mgr
SET mgr = 'Harry'
WHERE emp = 'Dave';
-- 查看改变经理后的结果
SELECT * FROM emp_mgr;
```



	emp	mgr	mgrcount
1	Alice	Harry	2
2	Dave	Joe	0
3	Harry	NULL	1
4	Joe	Alice	1
5	Paul	Alice	0

	emp	mgr	mgrcount
1	Alice	Harry	2
2	Dave	Harry	0
3	Harry	NULL	2
4	Joe	Alice	0
5	Paul	Alice	0

图 14-4 改变 Dave 的经理前/后的结果集

14.1.6 INSTEAD OF 触发器

在前面讨论的一直都是 AFTER 触发器。INSTEAD OF 在处理约束前激发，可以在 INSTEAD OF 中使用其他语句来替代激发触发器的 INSERT、UPDATE 等语句。

可以为表或视图定义 INSTEAD OF 触发器，但是，INSTEAD OF 触发器的主要优点是可以使不能更新的视图支持更新。基于多个基表的视图必须使用 INSTEAD OF 触发器来对多个表中的数据进行插入、更新和删除操作。INSTEAD OF 触发器的另一个优点是允许在批处理中某些部分成功的同时，能够拒绝批处理中的其他部分。

1. 为表创建 INSTEAD OF 触发器

与 AFTER 触发器相同，INSTEAD OF 触发器也需要指定其适用的操作类型，包括 INSERT、UPDATE 和 DELETE 操作。要创建一个 INSTEAD OF 触发器，必须在 CREATE TRIGGER 语句中使用 INSTEAD OF 关键字，而不是 AFTER 或 FOR 关键字。

例如，下面的示例创建了一个 Employee 表，该表的 EmployeeName 列具有主键约束。创建的 EmpInsteadInsert 触发器用于 INSERT 操作，只有在雇员姓名在表中不存在时，才插入新行。如果

发现要插入的雇员姓名在表中已经存在，仅更新该雇员的地址信息。因此，该表永远也不会违反主键约束。

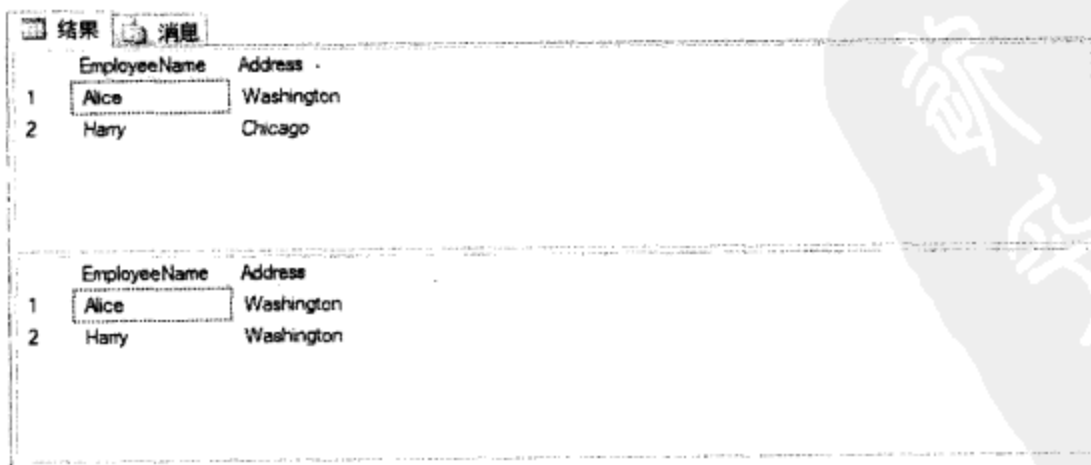
```
USE AdventureWorks;
GO

-- 创建表
CREATE TABLE Employee (
    EmployeeName char(30) PRIMARY KEY,
    Address char(30)
);
GO

-- 创建触发器，当有重复雇员名称时，更新该雇员的地址信息
CREATE TRIGGER EmpInsteadInsert
ON Employee
INSTEAD OF INSERT
AS
    IF EXISTS(
        SELECT *
        FROM Employee E, Inserted I
        WHERE E.EmployeeName = I.EmployeeName)
    BEGIN
        UPDATE Employee
        SET Address = Inserted.Address
        FROM Inserted
        WHERE Employee.EmployeeName = Inserted.EmployeeName;
    END
    ELSE
    BEGIN
        INSERT INTO Employee
        SELECT * FROM Inserted;
    END
END
```

执行下面的测试语句，得到的结果集如图 14-5 所示。

```
-- 下面这两个雇员的姓名在表中不存在，可以正常插入
INSERT INTO Employee VALUES('Harry','Chicago');
INSERT INTO Employee VALUES('Alice','Washington');
SELECT * FROM Employee; -- 查看结果集
-- 由于 Harry 在表中已经存在了，则仅修改 Harry 的地址信息
INSERT INTO Employee VALUES('Harry','Washington');
SELECT * FROM Employee;
```



	EmployeeName	Address
1	Alice	Washington
2	Harry	Chicago

	EmployeeName	Address
1	Alice	Washington
2	Harry	Washington

图 14-5 使用 INSTEAD OF 触发器处理后的结果

2. 为视图创建 INSTEAD OF 触发器

INSTEAD OF 触发器最大的作用是实现视图的更新，尤其是对基于多个表的视图。如果为视图创建了一个用于 INSERT 的 INSTEAD OF 触发器，在使用 INSERT 语句向视图中插入数据时，必须为每个不允许使用空值的视图列提供值。对于引用基表列的视图列，即使基表列指定不能输入值，但是由于不能为空，因此也必须为视图列指定一个值。这样的基表列包括：

- 基表中的计算列；
- 基表的 IDENTITY_INSERT 属性为 OFF 的标识列；
- 具有 timestamp 数据类型的基表列。

虽然视图要求必须为上述列提供值，但是，在 INSTEAD OF 触发器中生成真正要插入基表的 INSERT 语句时，必须忽略掉这些列的值。

例如，下面的语句创建了一个 BaseTable 表，并为其创建了一个包含基表所有列的视图。为视图创建的触发器 InsteadTrigger 在生成 INSERT 语句时，忽略掉了为基表的标识符列 (PrimaryKey) 和计算列 (ComputedCol) 提供的值，因为这些列不能指定值。

```
CREATE TABLE BaseTable
(PrimaryKey      int IDENTITY(1,1),
 Color           nvarchar(10) NOT NULL,
 Material        nvarchar(10) NOT NULL,
 ComputedCol AS (Color + Material));
GO

-- 创建一个包含基表所有列的视图
CREATE VIEW InsteadView
AS
    SELECT PrimaryKey, Color, Material, ComputedCol
    FROM BaseTable;
GO

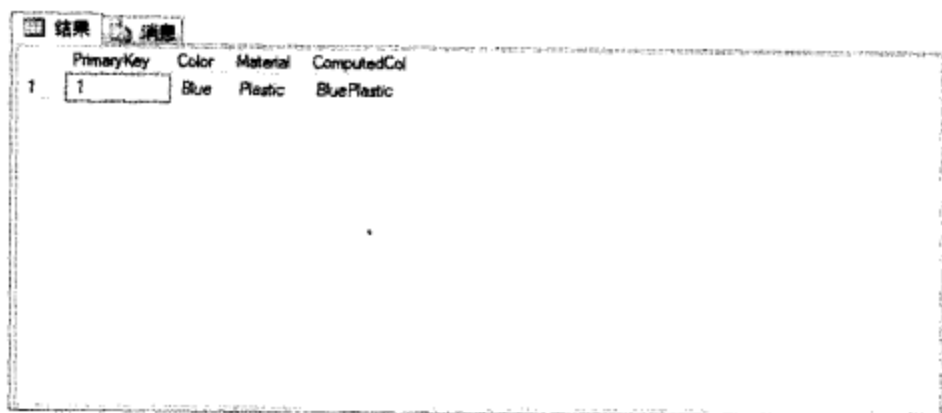
-- 为视图创建一个 INSTEAD OF INSERT 触发器
CREATE TRIGGER InsteadTrigger
ON InsteadView
INSTEAD OF INSERT
AS
BEGIN
    --生成一个 INSERT 语句向基表插入行，注意语句中忽略掉了 Inserted.PrimaryKey 和 Inserted.ComputedCol 列
    INSERT INTO BaseTable
        SELECT Color, Material
        FROM Inserted;
END
```

在通过 InsteadView 视图更新基表 BaseTable 时，引用 InsteadView 的 INSERT 语句必须为 PrimaryKey 列和 ComputedCol 列提供值，因为 PrimaryKey 列不能为空，而 ComputedCol 列是计算列。在下面的语句中，值 999 和 N'XXXXXX'将传递到 InsteadTrigger，但是触发器中的 INSERT 语句未选择 Inserted.PrimaryKey 和 Inserted.ComputedCol。这两个值仅起占位作用，可以是任意数值，只要数据类型正确就可以。

```
INSERT INTO InsteaView (PrimaryKey, Color, Material, ComputedCol)
VALUES (999, N'Blue', N'Plastic', N'XXXXXX')
```

执行下面的语句, 可以看到数据被正确地插入到了基表中, 999 和 N'XXXXXX' 值被忽略掉了, 如图 14-6 所示。

```
SELECT PrimaryKey, Color, Material, ComputedCol
FROM BaseTable;
```



	PrimaryKey	Color	Material	ComputedCol
1	1	Blue	Plastic	BluePlastic

图 14-6 通过视图的 INSTEAD OF INSERT 触发器插入到基表中的数据

应用于 INSTEAD OF INSERT 视图触发器的这些规定, 同样适用于 INSTEAD OF UPDATE 触发器。在执行 UPDATE 语句更新视图时, 必须为不能为空的列指定值, 但是在触发器中需要忽略掉这些值。例如, 下面是为 InsteaView 创建的一个 INSTEAD OF UPDATE 触发器:

```
CREATE TRIGGER InsteaTriggerUpdate
ON InsteaView
INSTEAD OF UPDATE
AS
BEGIN
    UPDATE BaseTable
    SET BaseTable.Color = Inserted.Color,
        BaseTable.Material = Inserted.Material
    FROM Inserted
    WHERE BaseTable.PrimaryKey = Inserted.PrimaryKey
END
GO
```

3. 在 INSTEAD OF 触发器中使用 text、ntext 和 image 数据

数据修改可能会涉及 text、ntext 和 image 列。在基表中, 存储在 text、ntext 或 image 列中的值是文本指针, 它指向保存数据的页。

AFTER 触发器不支持在 Inserted 和 Deleted 表中使用 text、ntext 或 image 数据, 但是 INSTEAD OF 触发器支持。text、ntext 和 image 数据在 Inserted 的 Deleted 表中的存储方式与在基表中的存储方式不同, 它们是作为连续的字符串存储的, 而不是指向数据页的文本指针。因此, 像 TEXTPTR 函数、TEXTVALID 函数、READTEXT 语句、UPDATETEXT 语句等这些需要文本指针的命令和函数无法调用 Inserted 和 Deleted 表中的 text、ntext、image 列。但是, 除此之外的其他文本操作函数 (如 SUBSTRING、PATINDEX 或 CHARINDEX 函数) 等均可以引用它们。在 INSTEAD OF 触发器中, 对 text、ntext 或 image 数据的操作受当前 SET TEXTSIZE 选项的影响, 可以使用 @@TEXTSIZE 函数来确定当前的设置值。

如果 INSERT、UPDATE、或 DELETE 语句修改了含有较大的 text、ntext 或 image 值的很多行，则需要大量内存来备份 Inserted 和 Deleted 表中的 text、ntext 或 image 数据。复制大量的数据还会降低性能。如果 INSERT、UPDATE 和 DELETE 语句引用了具有 INSTEAD OF 触发器的视图或表，应尽可能地每次修改一行，或者一次仅修改几行。

14.2 使用 DDL 触发器

与 DML 触发器不同，DDL 触发器不会为响应针对表或视图的 UPDATE、INSERT 或 DELETE 语句而激发，而是会为响应多种数据定义语言（DDL）语句而激发。这些语句主要是以 CREATE、ALTER 和 DROP 开头的语句。DDL 触发器用于执行管理任务，例如审核和控制数据库操作。并且，DDL 触发器只能在 SQL 语句完成之后才运行，无法作为 INSTEAD OF 触发器使用。

二者也有许多相似之处，例如，DML 和 DDL 触发器都可以运行在 Microsoft .NET Framework 中创建的托管代码；可以为同一个 SQL 语句创建多个 DDL 触发器；DDL 触发器也是与激发它的语句运行在相同的事务中，可从触发器中回滚此事务。此外，也可以对 DDL 触发器实现嵌套。

14.2.1 激发 DDL 触发器的 DDL 事件和事件组

可以指定 DDL 触发器由一个或多个特定的 DDL 语句激发，也可以指定由预定义的一组 DDL 语句激发。这些事件决定了触发器的作用域范围。

数据库范围内的 DDL 触发器作为对象存储在创建它们的数据库中，可以从创建 DDL 触发器的数据库的 sys.triggers 目录视图（也可以使用数据库名称限定要查找的视图，如 AdventureWorks.sys.triggers）中，获取有关这些 DDL 触发器的信息。

服务器范围内的 DDL 触发器作为对象存储在 master 数据库中，可以从任一数据库的 sys.server_triggers 目录视图中获取服务器范围内的 DDL 触发器的信息。

1. 激发 DDL 触发器的 DDL 事件

并不是所有的 DDL 事件都可用于 DDL 触发器中，如 CREATE DATABASE 事件就不能用于 DDL 触发器中。表 14-1 列出了用于数据库作用域的 DDL 事件，每个事件都对应一个 SQL 语句。这些事件名称是通过在 SQL 语句的各关键字之间添加下划线（“_”）的方式来命名的。

表 14-1 数据库作用域的 DDL 事件

用于创建的事件	用于修改的事件	用于删除/禁止等的事件
CREATE_APPLICATION_ROLE	ALTER_APPLICATION_ROLE	DROP_APPLICATION_ROLE
CREATE_ASSEMBLY	ALTER_ASSEMBLY	DROP_ASSEMBLY
	ALTER_AUTHORIZATION_DATABASE	
CREATE_CERTIFICATE	ALTER_CERTIFICATE	DROP_CERTIFICATE

续表

用于创建的事件	用于修改的事件	用于删除/禁止等的事件
CREATE_CONTRACT		DROP_CONTRACT
GRANT_DATABASE		DENY_DATABASE REVOKE_DATABASE
CREATE_EVENT_NOTIFICATION		DROP_EVENT_NOTIFICATION
CREATE_FUNCTION	ALTER_FUNCTION	DROP_FUNCTION
CREATE_INDEX	ALTER_INDEX	DROP_INDEX
CREATE_MESSAGE_TYPE	ALTER_MESSAGE_TYPE	DROP_MESSAGE_TYPE
CREATE_PARTITION_FUNCTION	ALTER_PARTITION_FUNCTION	DROP_PARTITION_FUNCTION
CREATE_PARTITION_SCHEME	ALTER_PARTITION_SCHEME	DROP_PARTITION_SCHEME
CREATE_PROCEDURE	ALTER_PROCEDURE	DROP_PROCEDURE
CREATE_QUEUE	ALTER_QUEUE	DROP_QUEUE
CREATE_REMOTE_SERVICE_BINDING	ALTER_REMOTE_SERVICE_BINDING	DROP_REMOTE_SERVICE_BINDING
CREATE_ROLE	ALTER_ROLE	DROP_ROLE
CREATE_ROUTE	ALTER_ROUTE	DROP_ROUTE
CREATE_SCHEMA	ALTER_SCHEMA	DROP_SCHEMA
CREATE_SERVICE	ALTER_SERVICE	DROP_SERVICE
CREATE_STATISTICS	UPDATE_STATISTICS	DROP_STATISTICS
CREATE_SYNONYM		DROP_SYNONYM
CREATE_TABLE	ALTER_TABLE	DROP_TABLE
CREATE_TRIGGER	ALTER_TRIGGER	DROP_TRIGGER
CREATE_TYPE		DROP_TYPE
CREATE_USER	ALTER_USER	DROP_USER
CREATE_VIEW	ALTER_VIEW	DROP_VIEW
CREATE_XML_SCHEMA_COLLECTION	ALTER_XML_SCHEMA_COLLECTION	DROP_XML_SCHEMA_COLLECTION

表 14-2 列出了用于服务器作用域的 DDL 语句事件，事件的命名方式与用于数据库作用域的 DDL 事件相同。

表 14-2 数据库作用域的 DDL 事件

用于创建的事件	用于修改的事件	用于删除/禁止等的事件
	ALTER_AUTHORIZATION_SERVER	
CREATE_DATABASE	ALTER_DATABASE	DROP_DATABASE
CREATE_ENDPOINT		DROP_ENDPOINT
CREATE_LOGIN	ALTER_LOGIN	DROP_LOGIN
GRANT_SERVER		DENY_SERVER REVOKE_SERVER

2. 激发 DDL 触发器的事件组

图 14-7 列出了可用于激发 DDL 触发器的事件组及其所涵盖的 SQL 语句，以及可以在其中对事件组进行编程的作用域（ON SERVER 或 ON DATABASE）。注意图中由树结构标明的事件组的包含性。例如，指定 FOR DDL_TABLE_EVENTS 的 DDL 触发器将涵盖 CREATE TABLE、ALTER TABLE 和 DROP TABLE 语句，而指定 FOR DDL_TABLE_VIEW_EVENTS 的 DDL 触发器，则将涵盖 DDL_TABLE_EVENTS、DDL_VIEW_EVENTS、DDL_INDEX_EVENTS 和 DDL_STATISTICS_EVENTS 下的所有 Transact-SQL 语句。

事件组及其所涵盖的 Transact-SQL 语句	服务器 范围	数据库 范围
DDL_SERVER_LEVEL_EVENTS (CREATE DATABASE, ALTER DATABASE, DROP DATABASE)	X	
DDL_ENDPOINT_EVENTS (CREATE ENDPOINT, ALTER ENDPOINT, DROP ENDPOINT)	X	
DDL_SERVER_SECURITY_EVENTS	X	
DDL_LOGIN_EVENTS (CREATE LOGIN, ALTER LOGIN, DROP LOGIN)	X	
DDL_GDR_SERVER_EVENTS (GRANT SERVER, DENY SERVER, REVOKE SERVER)	X	
DDL_AUTHORIZATION_SERVER_EVENTS (ALTER AUTHORIZATION SERVER)	X	
DDL_DATABASE_LEVEL_EVENTS		X
DDL_TABLE_VIEW_EVENTS		X
DDL_TABLE_EVENTS (CREATE TABLE, ALTER TABLE, DROP TABLE)		X
DDL_VIEW_EVENTS (CREATE VIEW, ALTER VIEW, DROP VIEW)		X
DDL_INDEX_EVENTS (CREATE INDEX, ALTER INDEX, DROP INDEX, CREATE XML INDEX)		X
DDL_STATISTICS_EVENTS (CREATE STATISTICS, UPDATE STATISTICS, DROP STATISTICS)		X
DDL_SYNONYM_EVENTS (CREATE SYNONYM, DROP SYNONYM)		X
DDL_FUNCTION_EVENTS (CREATE FUNCTION, ALTER FUNCTION, DROP FUNCTION)		X
DDL_PROCEDURE_EVENTS (CREATE PROCEDURE, ALTER PROCEDURE, DROP PROCEDURE)		X
DDL_TRIGGER_EVENTS (CREATE TRIGGER, ALTER TRIGGER, DROP TRIGGER)		X
DDL_EVENT_NOTIFICATION_EVENTS (CREATE EVENT NOTIFICATION, DROP EVENT NOTIFICATION)		X
DDL_ASSEMBLY_EVENTS (CREATE ASSEMBLY, ALTER ASSEMBLY, DROP ASSEMBLY)		X
DDL_TYPE_EVENTS (CREATE TYPE, DROP TYPE)		X
DDL_DATABASE_SECURITY_EVENTS		X
DDL_CERTIFICATE_EVENTS (CREATE CERTIFICATE, ALTER CERTIFICATE, DROP CERTIFICATE)		X
DDL_USER_EVENTS (CREATE USER, ALTER USER, DROP USER)		X
DDL_ROLE_EVENTS (CREATE ROLE, ALTER ROLE, DROP ROLE)		X
DDL_APPLICATION_ROLE_EVENTS (CREATE APPLICATION ROLE, ALTER APPLICATION ROLE, DROP APPLICATION ROLE)		X
DDL_SCHEMA_EVENTS (CREATE SCHEMA, ALTER SCHEMA, DROP SCHEMA)		X
DDL_GDR_DATABASE_EVENTS (GRANT DATABASE, DENY DATABASE, REVOKE DATABASE)		X
DDL_AUTHORIZATION_DATABASE_EVENTS (ALTER AUTHORIZATION DATABASE)		X
DDL_SSB_EVENTS		X
DDL_MESSAGE_TYPE_EVENTS (CREATE MESSAGE TYPE, ALTER MESSAGE TYPE, DROP MESSAGE TYPE)		X
DDL_CONTRACT_EVENTS (CREATE CONTRACT, DROP CONTRACT)		X
DDL_QUEUE_EVENTS (CREATE QUEUE, ALTER QUEUE, DROP QUEUE)		X
DDL_SERVICE_EVENTS (CREATE SERVICE, ALTER SERVICE, DROP SERVICE)		X
DDL_ROUTE_EVENTS (CREATE ROUTE, ALTER ROUTE, DROP ROUTE)		X
DDL_REMOTE_SERVICE_BINDING_EVENTS (CREATE REMOTE SERVICE BINDING, ALTER REMOTE SERVICE BINDING, DROP REMOTE SERVICE BINDING)		X
DDL_XML_SCHEMA_COLLECTION_EVENTS (CREATE XML SCHEMA COLLECTION, ALTER XML SCHEMA COLLECTION, DROP XML SCHEMA COLLECTION)		X
DDL_PARTITION_FUNCTION_EVENTS (CREATE PARTITION FUNCTION, ALTER PARTITION FUNCTION, DROP PARTITION FUNCTION)		X
DDL_PARTITION_SCHEME_EVENTS (CREATE PARTITION SCHEME, ALTER PARTITION SCHEME, DROP PARTITION SCHEME)		X

图 14-7 用于激发 DDL 触发器的事件组及其所涵盖的 SQL 语句

14.2.2 创建 DDL 触发器

与创建 DML 触发器相同，创建 DDL 触发器也需要使用 CREATE TRIGGER 语句。

1. 创建具有数据库作用域的 DDL 触发器

下面的语句创建了一个具有数据库作用域的 DDL 触发器。CREATE TRIGGER 关键字后面是触发器的名称，DATABASE 关键字指定触发器的作用域范围是当前数据库，当发生 FOR 关键字后面指定的 DROP_TABLE 事件时，将激发该触发器。

```
USE AdventureWorks;
GO

CREATE TRIGGER Safety
ON DATABASE
FOR DROP_TABLE
AS
    PRINT N'要删除表，需要禁用或删除 safety 触发器。';
    ROLLBACK;
```

执行下面的测试语句，首先创建一个表 MyTable，然后再删除该表。此时将激发 Safety 触发器，不允许删除表，并返回如图 14-8 所示的消息。

```
CREATE TABLE MyTable (c1 int);
GO
DROP TABLE MyTable;
```



图 14-8 激发 Safety 触发器时返回的消息

如果要为触发器定义多个事件，可以在 FOR 关键字后面使用以逗号分隔的事件列表，也可以使用事件组的方式进行定义。例如，下面的 Safety1 触发器被定义了 DROP_TABLE 和 CREATE_TABLE 两个事件，在当前数据库中创建或删除表时都会激发该触发器。Safety2 触发器则以事件组的方式被指定了多个事件，当执行这些事件组中包含的 Transact-SQL 语句时，将激发 Safety2 触发器。

```
CREATE TRIGGER Safety1
ON DATABASE
FOR DROP_TABLE, CREATE_TABLE
```

```

AS
    PRINT N'要删除或新建表，需要禁用或删除 safety1 触发器。' ;
    ROLLBACK;
GO

-- 以事件组的方式指定事件
CREATE TRIGGER Safety2
ON DATABASE
FOR DDL_VIEW_EVENTS, DDL_INDEX_EVENTS
AS
    PRINT N'要创建、修改和删除视图、索引，需要禁用或删除 safety2 触发器。' ;
    ROLLBACK;

```

2. 创建具有服务器作用域的 DDL 触发器

下面的语句创建了一个具有服务器作用域的 DDL 触发器。其中的 ALL SERVER 指定触发器的作用域是当前服务器实例，当发生 FOR 关键字后面指定的 CREATE_DATABASE 事件时，将激发该触发器。

```

CREATE TRIGGER CreateData
ON ALL SERVER
FOR CREATE_DATABASE
AS
    PRINT N'要创建数据库，需要禁用或删除 CreateData 触发器。' ;
    ROLLBACK;

```

在设置该触发器后，不允许在当前数据库服务器实例上创建数据库。

3. 使用 EVENTDATA() 函数捕获激发 DDL 触发器的事件信息

EVENTDATA() 函数可以返回有关服务器或数据库事件的信息。但是，只有在 DDL 触发器内部被直接引用时，EVENTDATA() 才会返回数据；否则，将返回 NULL。

EVENTDATA() 返回 xml 类型值。其中包含事件发生的时间、执行触发器的连接的系统进程 ID (SPID) 和激发触发器的事件类型等。但是，在隐式或显式事务提交或回滚之后执行 EVENTDATA 函数，所返回的数据将无效。

下面的示例创建的 Safety DDL 触发器，在执行 CREATE TABLE 语句时将被激发。它返回由 EVENTDATA() 函数捕获的事件信息，并回滚 CREATE TABLE 语句操作。

```

USE AdventureWorks;
GO
CREATE TRIGGER Safety
ON DATABASE
FOR CREATE_TABLE
AS
    SELECT EVENTDATA(); -- 获取事件信息
    ROLLBACK;
GO

-- 执行测试语句
CREATE TABLE MyTable (c1 int)

```

在执行测试语句时，Safety 触发器返回如下的 XML 信息：


```

<EVENT_INSTANCE>
  <EventType>CREATE_TABLE</EventType>
  <PostTime>2006-07-17T21:58:09.193</PostTime>
  <SPID>52</SPID>
  <ServerName>CCBSERVER1</ServerName>
  <LoginName>CCBSERVER1\Administrator</LoginName>
  <UserName>dbo</UserName>
  <DatabaseName>AdventureWorks</DatabaseName>
  <SchemaName>dbo</SchemaName>
  <ObjectName>MyTable</ObjectName>
  <ObjectType>TABLE</ObjectType>
  <TSQLCommand>
    <SetOptions ANSI_NULLS="ON" ANSI_NULL_DEFAULT="ON" ANSI_PADDING="ON" QUOTED_IDENTIFIER="ON"
    ENCRYPTED="FALSE" />
    <CommandText>CREATE TABLE MyTable (c1 int)</CommandText>
  </TSQLCommand>
</EVENT_INSTANCE>

```

要从 XML 数据中检索出所需要的内容, 可以使用 XML 数据类型的 `value()` 方法。例如, 现在将 `Safety` 触发器改写为以下形式:

```

USE AdventureWorks;
GO
CREATE TRIGGER Safety
ON DATABASE
FOR CREATE_TABLE
AS
  PRINT N'CREATE TABLE 出错';
  SELECT
  EVENTDATA().value('(/EVENT_INSTANCE/TSQLCommand/CommandText)[1]', 'nvarchar(max)');
  ROLLBACK;

```

其中, `(/EVENT_INSTANCE/TSQLCommand/CommandText)` 指定要返回值的属性名称, `[1]` 指定要返回 XML 中第 1 个 `EVENT_INSTANCE/TSQLCommand/CommandText` 属性的值, 因为一个 XML 中可能有多个同名的属性。 `nvarchar(max)` 指定存放返回值的数据类型。

例如, 下面的示例指定返回 XML 中第 2 个和第 4 个 `MyValue` 属性的值, 并指定值的数据类型为 `int`。

```

DECLARE @myDoc xml
SET @myDoc = '<Root>
  <MyValue>1</MyValue>
  <MyValue>2</MyValue>
  <MyValue>3</MyValue>
  <MyValue>4</MyValue>
  <MyValue>5</MyValue>
</Root>';

SELECT @myDoc.value('(/Root/MyValue)[2]', 'int' );
SELECT @myDoc.value('(/Root/MyValue)[4]', 'int' );

```

当再次执行 “`CREATE TABLE MyTable (c1 int)`” 测试语句时, `Safety` 触发器将返回激发触发器的 SQL 语句。

14.3 CLR 触发器

CLR 扩展了 SQL 触发器的能力，它允许程序员使用自己熟悉的语言（如 C#）来实现一些复杂的功能。使用 CLR 可以创建 DML 触发器和 DDL 触发器。

14.3.1 SqlTriggerContext 类

SqlTriggerContext 类用于提供所激发的触发器的上下文信息，包括导致触发器被激发的操作的类型，以及确定在 UPDATE 操作中哪些列被进行了更新。如果是 DDL 触发器，则还可以得到与 EVENTDATA() 函数同样结构的 XML 类型信息。

SqlTriggerContext 类不能是公共构造，并且只能在一个 CLR 触发器内通过访问 SqlContext.TriggerContext 属性获得。例如：

```
SqlTriggerContext myTriggerContext = SqlContext.TriggerContext;
```

1. 确定激发触发器的操作

可以通过 SqlTriggerContext 类的 TriggerAction 属性获得激发触发器的操作的类型。对于 DML 触发器，TriggerAction 属性可以是 Update、Insert、Delete 值，而对于 DDL 触发器，则可能的 TriggerAction 属性值相当多。表 14-3 列出了 TriggerAction 属性的所有可能值。

表 14-3 TriggerAction 属性的枚举值

成员名称	说 明
AlterAppRole	已执行 ALTER APPLICATION ROLE 语句
AlterAssembly	已执行 ALTER ASSEMBLY 语句
AlterBinding	当事件通知在数据库或服务器实例上创建时，会指定 ALTER_REMOTE_SERVICE_BINDING 事件类型
AlterFunction	已执行 ALTER FUNCTION 语句
AlterIndex	已执行 ALTER INDEX 语句
AlterLogin	已执行 ALTER LOGIN 语句
AlterPartitionFunction	已执行 ALTER PARTITION FUNCTION 语句
AlterPartitionScheme	已执行 ALTER PARTITION SCHEME 语句
AlterProcedure	已执行 ALTER PROCEDURE 语句
AlterQueue	已执行 ALTER QUEUE 语句
AlterRole	已执行 ALTER ROLE 语句
AlterRoute	已执行 ALTER ROUTE 语句
AlterSchema	已执行 ALTER SCHEMA 语句

续表

成员名称	说 明
AlterService	已执行 ALTER SERVICE 语句
AlterTable	已执行 ALTER TABLE 语句
AlterTrigger	已执行 ALTER TRIGGER 语句
AlterUser	已执行 ALTER USER 语句
AlterView	已执行 ALTER VIEW 语句
CreateAppRole	已执行 CREATE APPLICATION ROLE 语句
CreateAssembly	已执行 CREATE ASSEMBLY 语句
CreateBinding	当事件通知在数据库或服务器实例上创建时，会指定 CREATE_REMOTE_SERVICE_BINDING 事件类型
CreateContract	已执行 CREATE CONTRACT 语句
CreateEventNotification	已执行 CREATE EVENT NOTIFICATION 语句
CreateFunction	已执行 CREATE FUNCTION 语句
CreateIndex	已执行 CREATE INDEX 语句
CreateLogin	已执行 CREATE LOGIN 语句
CreateMsgType	已执行 CREATE MESSAGE TYPE 语句
CreatePartitionFunction	已执行 CREATE PARTITION FUNCTION 语句
CreatePartitionScheme	已执行 CREATE PARTITION SCHEME 语句
CreateProcedure	已执行 CREATE PROCEDURE 语句
CreateQueue	已执行 CREATE QUEUE 语句
CreateRole	已执行 CREATE ROLE 语句
CreateRoute	已执行 CREATE ROUTE 语句
CreateSchema	已执行 CREATE SCHEMA 语句
CreateSecurityExpression	
CreateService	已执行 CREATE SERVICE 语句
CreateSynonym	已执行 CREATE SYNONYM 语句
CreateTable	已执行 CREATE TABLE 语句
CreateTrigger	已执行 CREATE TRIGGER 语句
CreateType	已执行 CREATE TYPE 语句
CreateUser	已执行 CREATE USER 语句
CreateView	已执行 CREATE VIEW 语句
Delete	已执行 DELETE 语句
DenyObject	已执行 DENY Object Permissions 语句

续表

成员名称	说 明
DenyStatement	已执行 DENY 语句
DropAppRole	已执行 DROP APPLICATION ROLE 语句
DropAssembly	已执行 DROP ASSEMBLY 语句
DropBinding	当 事 件 通 知 在 数 据 库 或 服 务 器 实 例 上 创 建 时 ， 会 指 定 DROP_REMOTE_SERVICE_BINDING 事件类型
DropContract	已执行 DROP CONTRACT 语句
DropEventNotification	已执行 DROP EVENT NOTIFICATION 语句
DropFunction	已执行 DROP FUNCTION 语句
DropIndex	已执行 DROP INDEX 语句
DropLogin	已执行 DROP LOGIN 语句
DropMsgType	已执行 DROP MESSAGE TYPE 语句
DropPartitionFunction	已执行 DROP PARTITION FUNCTION 语句
DropPartitionScheme	已执行 DROP PARTITION SCHEME 语句
DropProcedure	已执行 DROP PROCEDURE 语句
DropQueue	已执行 DROP QUEUE 语句
DropRole	已执行 DROP ROLE 语句
DropRoute	已执行 DROP ROUTE 语句
DropSchema	已执行 DROP SCHEMA 语句
DropSecurityExpression	
DropService	已执行 DROP SERVICE 语句
DropSynonym	已执行 DROP SYNONYM 语句
DropTable	已执行 DROP TABLE 语句
DropTrigger	已执行 DROP TRIGGER 语句
DropType	已执行 DROP TYPE 语句
DropUser	已执行 DROP USER 语句
DropView	已执行 DROP VIEW 语句
GrantObject	
GrantStatement	
Insert	已执行 INSERT 语句
Invalid	出现一个无效触发操作，该操作不向用户公开
RevokeObject	
RevokeStatement	
Update	已执行 UPDATE 语句

2. 使用 Inserted 和 Deleted 表

通过 `SqlContext` 对象的 `SqlCommand` 对象, CLR 触发器也可以访问 `Inserted` 和 `Deleted` 表, 例如:

```
SqlConnection connection = new SqlConnection ("context connection = true");
connection.Open();
SqlCommand command = connection.CreateCommand();
command.CommandText = "SELECT * from " + "inserted";
```

3. 确定更新列

可以通过 `SqlTriggerContext` 对象的 `ColumnCount` 属性来得到一个 `UPDATE` 操作要修改的列的数目。通过为 `IsUpdatedColumn` 方法提供一个列的顺序位置参数, 可以来确定指定列是否被更新。如果返回 `TRUE`, 表示列已经被更新。

例如, 下面的代码片段将列出所有列的更新情况:

```
reader = command.ExecuteReader();
reader.Read();
for (int columnNumber = 0; columnNumber < triggContext.ColumnCount; columnNumber++)
{
    pipe.Send("Updated column "
        + reader.GetName(columnNumber) + "? "
        + triggContext.IsUpdatedColumn(columnNumber).ToString());
}
reader.Close();
```

4. 在 CLR DDL 触发器中访问 eventdata

激发 DDL 触发器的事件信息可以通过 `SqlTriggerContext` 类的 `EventData` 属性获得, 该属性包含一个 XML 型值。XML 的架构与在 DDL 触发器中使用 `EVENTDATA()` 函数返回的架构相同。

14.3.2 创建 CLR DML 触发器的步骤

创建 CLR 触发器与创建 CLR 存储过程的步骤基本一致: 首先使用 .NET Framework 支持的语言 (如 C#) 将触发器定义为一个类, 并编译该类生成程序集。然后可以在 Visual Studio 中启动部署, 直接将程序集和上载到 SQL Server 中, 也可以使用 `CREATE ASSEMBLY` 语句手动注册程序集。最后使用 `CREATE TRIGGER` 语句创建引用注册程序集的触发器。

1. 通过 Visual Studio 编写和部署 CLR 触发器

打开 Visual Studio, 从菜单中依次选择“文件”→“新建”→“项目”, 打开如图 14-9 所示的“新建项目”对话框。选定 Visual C# 下的“数据库”节点, 并在该对话框的下方指定项目名称和存储位置后, 单击“确定”按钮。

在出现的如图 14-10 所示的“新建数据库引用”对话框中, 指定要连接的服务器和数据库名称,

并单击“确定”按钮。

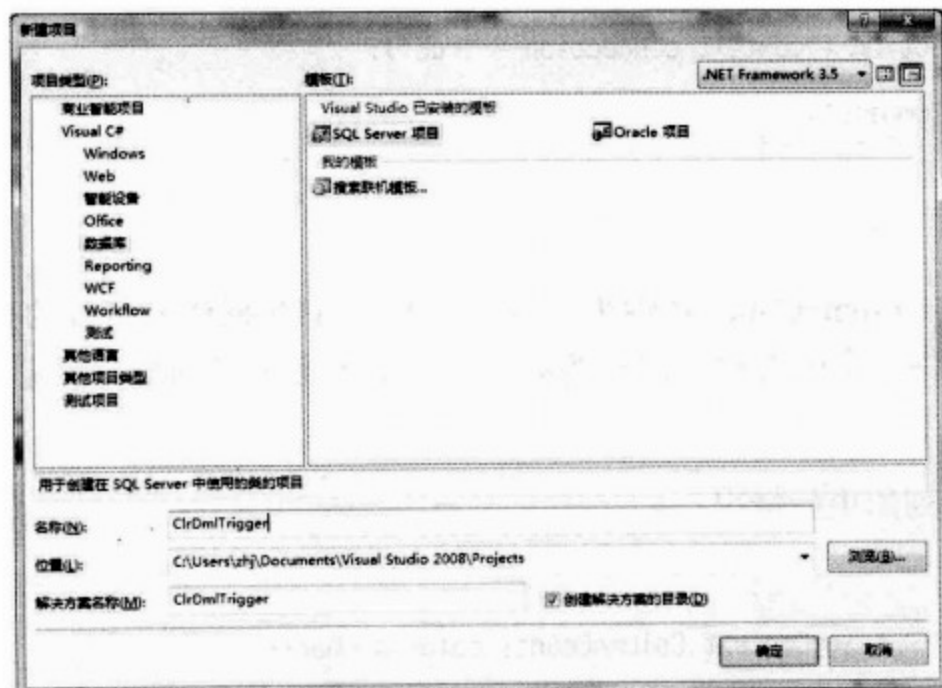


图 14-9 “新建项目”对话框

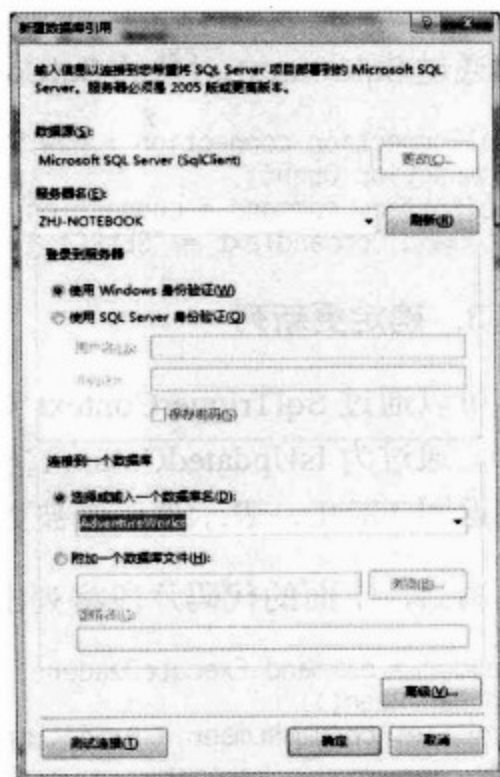


图 14-10 指定要连接的数据

从 Visual Studio 的菜单中依次选择“项目”→“添加触发器”，打开如图 14-11 所示的“添加新项”对话框，在“名称”文本框中为触发器指定一个名称，如 MyTableUpdDelTrigger.cs。

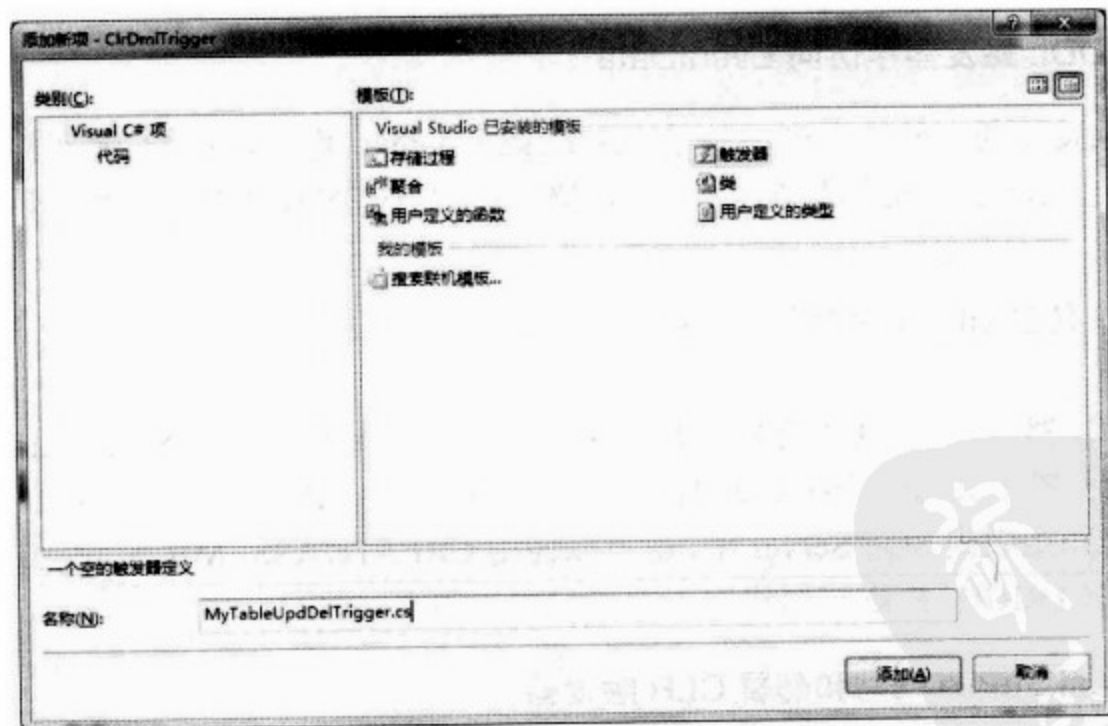


图 14-11 “添加新项”对话框

单击“添加”按钮后，Visual Studio 将自动在触发器中添加如下代码：

```
using System;
using System.Data;
using System.Data.SqlClient;
```

```

using Microsoft.SqlServer.Server;

public partial class Triggers
{
    // 为目标输入现有表或视图并取消对属性行的注释
    // [Microsoft.SqlServer.Server.SqlTrigger (Name="MyTableUpdDelTrigger", Target="Table1", Event="FOR
    UPDATE")]
    public static void MyTableUpdDelTrigger()
    {
        // 用您的代码替换
        SqlContext.Pipe.Send("Trigger FIRED");
    }
}

```

这段代码的基本结构是由一系列 `using` 命令、一个类（`Triggers`）和类中的方法（`MyTableUpdDelTrigger`）组成，它是由“触发器”模板创建的。要编写一个 DML 触发器，应当将 `[Microsoft.SqlServer.Server.SqlTrigger...]` 语句前面的注释去掉。该语句用于将程序集中的方法定义标记为 SQL Server 中的触发器。其中的 `Name` 属性用于指定触发器的名称，`Target` 属性指定将当前触发器应用于哪个表，`Event` 属性指定触发器的类型和可以激发触发器的 DML 操作。

下面是 `MyTableUpdDelTrigger` 触发器的完整代码，该触发器在执行 `UPDATE` 和 `DELETE` 操作时激发。当修改 `MyTable` 表的第 2 个列的值时，将从 `Inserted` 表中检索被修改的行，并返回到客户端。当删除 `MyTable` 表中的行时，将从 `Deleted` 表中检索出被删除的行，并返回到客户端。

```

using System;
using System.Data;
using System.Data.SqlClient;
using Microsoft.SqlServer.Server;

public partial class Triggers //类名称
{
    [Microsoft.SqlServer.Server.SqlTrigger (Name="MyTableUpdDelTrigger", Target="MyTable", Event="FOR
    UPDATE, DELETE")]
    public static void MyTableUpdDelTrigger() //方法名称
    {
        SqlTriggerContext triggContext = SqlContext.TriggerContext;

        //设置连接
        SqlConnection conn = new SqlConnection();
        conn.ConnectionString = "Context Connection=true";
        conn.Open();
        SqlCommand cmd = new SqlCommand();
        cmd.Connection = conn;

        SqlDataReader reader;

        switch (triggContext.TriggerAction)
        {
            case TriggerAction.Update:
                if (triggContext.IsUpdatedColumn(1)) //如果更新了第 2 列（序号从 0 开始）
                {
                    //从 Inserted 表中获取更改后的行
                    cmd.CommandText = "SELECT * FROM INSERTED";
                    reader = cmd.ExecuteReader(); //生成 SqlDataReader

                    //发送数据到客户端
                }
            }
        }
    }
}

```

```

        SqlContext.Pipe.Send(reader);

        //发送消息给客户端
        SqlContext.Pipe.Send("MyTable 表的第 2 列已经被更改");
    }
    break;
case TriggerAction.Delete:
    cmd.CommandText = "SELECT * FROM DELETED";
    reader = cmd.ExecuteReader();
    SqlContext.Pipe.Send(reader);
    SqlContext.Pipe.Send("MyTable 表中被删除的行");
    break;
}
}
}

```

执行下面的语句，在 AdventureWorks 数据库中创建要为之定义触发器的 MyTable 表，并插入测试数据。如果由 Microsoft.SqlServer.Server.SqlTrigger 类的 Target 属性指定的表对象在数据库中不存在，则不能部署成功。

```

USE AdventureWorks;
GO
CREATE TABLE MyTable (c1 int, c2 int);
INSERT INTO MyTable VALUES(1,1);

```

在 Visual Studio 的菜单中依次选择“生成”→“生成 CLR DML Trigger”，对 CLR DML Trigger 项目进行编译。再依次选择“生成”→“部署 CLR DML Trigger”，将把程序集和存储过程部署到 SQL Server 中去。

打开 SQL Server Management Studio，展开 AdventureWorks 数据库中 MyTable 表中的“触发器”节点和“可编程性”中的“程序集”节点，可以看到已经部署的 MyTableUpdDelTrigger 触发器和 CLR DML Trigger 程序集，如图 14-12 所示。

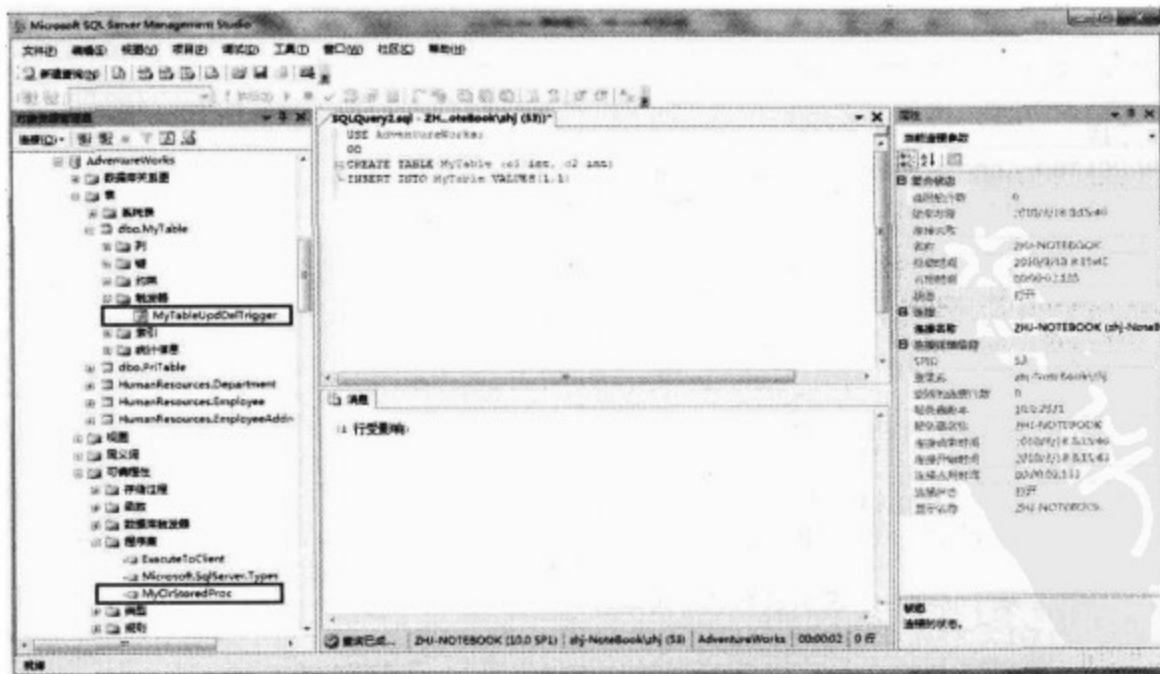


图 14-12 已部署的程序集和存储过程

执行下面的测试语句，更新 MyTable 表的第 c2 列。可以看到被更新后的结果和提示更新消息，如图 14-13 所示。

```
UPDATE MyTable
SET c2 = 2
WHERE c1 = 1;
```

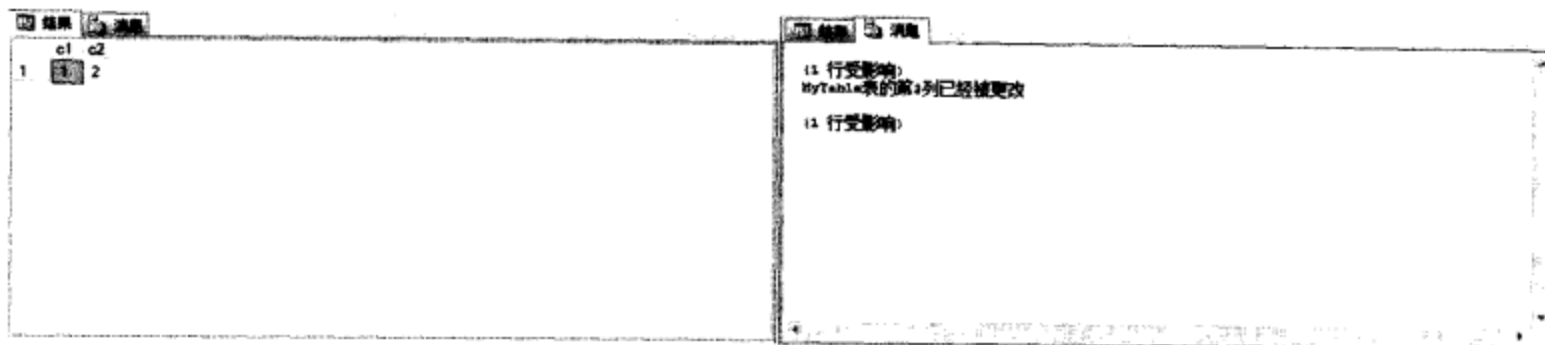


图 14-13 由 CLR 触发器返回的行集和消息

2. 使用 SQL 添加程序集和创建 CLR 触发器

只有首先将包含托管代码的程序集在 SQL Server 中注册，才可以使用 CREATE TRIGGER 语句创建引用注册程序集的触发器。参考下面的语句：

```
USE AdventureWorks;
GO

-- 注册程序集
CREATE ASSEMBLY CLRDDLTrigger
FROM 'C:\MyProject\CLRDDLTrigger\CLRDDLTrigger\obj\Debug\CLRDDLTrigger.dll'
WITH PERMISSION_SET = SAFE;
GO

-- 创建基于程序集的触发器
CREATE TRIGGER MyTableUpdDelTrigger
ON MyTable
AFTER DELETE, UPDATE
AS
    EXTERNAL NAME CLRDDLTrigger.Triggers.MyTableUpdDelTrigger -- 指定要使用的方法名称
GO
```

以上 CREATE TRIGGER 语句中的 EXTERNAL NAME 子句指定触发器要引用的 .NET Framework 程序集中的方法，方法名称应当按照“程序集.类.方法”这样的对象层次关系进行声明。

此外，还需要注意的是，SQL DML 触发器中指定的激发触发器的类型应当与 .NET Framework 方法指定的类型一致。例如，在上面示例中，CLR 和 Transact-SQL DML 中都指定激发触发器的操作为 UPDATE 和 DELETE。

14.3.3 创建 CLR DDL 触发器的步骤

创建 CLR DDL 触发器与创建 CLR DML 触发器的步骤完全相同，只是在定义

Microsoft.SqlServer.Server.SqlTrigger 类的属性时有一些差异。其中的 Target 属性用于指定 DDL 服务器的作用域，只能是 DATABASE（作用域为当前数据库）或 ALL SERVER（作用域为当前服务器实例）。Event 属性指定激发触发器事件名称，可用值应当是在 14.2.1 节中表 14-1 和表 14-2 中列出的事件名称，或是图 14-7 中列出的事件组名称。但是，在触发器方法中，仍旧需要使用 SqlTriggerContext 类的 TriggerAction 属性来判断所发生的事件。例如，如果设置 Event = "FOR ALTER_TABLE"，则应当通过 TriggerAction 属性的 AlterTable 值与之对应。

以下示例创建了一个具有数据库作用域的 DDL 触发器。当在数据库中修改表定义时，将向客户端发送提示消息，并将 SqlTriggerContext 类的 EventData 属性值返回给客户端。当在数据库中删除表时，将从 EventData 属性的 XML 型值中检索出 EventType 元素和 CommandText 元素的值，返回给客户端。为了能够从 XML 型数据中检索数据，应当在项目中包含 System.Xml 命名空间。

```
using System;
using System.Data;
using System.Data.SqlClient;
using Microsoft.SqlServer.Server;
using System.Xml;

public partial class Triggers //类名称
{
    [Microsoft.SqlServer.Server.SqlTrigger(Name = "AlterDropTableTrigger", Target = "DATABASE", Event =
    "FOR ALTER_TABLE, DROP_TABLE")]
    public static void AlterDropTableTrigger() //方法名称
    {
        SqlTriggerContext triggContext = SqlContext.TriggerContext;

        switch (triggContext.TriggerAction)
        {
            case TriggerAction.AlterTable:
                SqlContext.Pipe.Send("触发器检测到从数据库中修改了表");
                SqlContext.Pipe.Send(triggContext.EventData.Value);
                break;

            case TriggerAction.DropTable:
                SqlContext.Pipe.Send("触发器检测到从数据库中删除了表");
                XmlReader reader = triggContext.EventData.CreateReader();
                reader.ReadToFollowing("EventType"); //查找 EventType 元素
                SqlContext.Pipe.Send("事件类型: " + reader.ReadString()); //输出当前元素内容
                reader.ReadToFollowing("CommandText");
                SqlContext.Pipe.Send("命令: " + reader.ReadString());
                break;
        }
    }
}
```

在 Visual Studio 中将上面的触发器部署到 SQL Server 中。执行下面的语句，当删除表时，可以看到如图 14-14 所示的提示消息。

```
CREATE TABLE MyTable (c1 int, c2 int)
DROP TABLE MyTable -- 执行此语句时，将激发 AlterDropTableTrigger 触发器
```

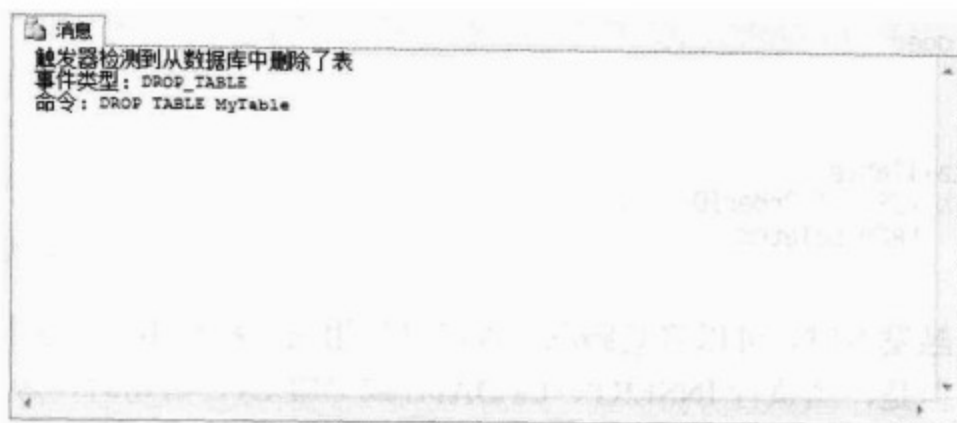



图 14-14 由 CLR DDL 触发器 AlterDropTableTrigger 返回的消息

以下是使用 SQL 语句注册程序集和定义 DDL 触发器的方法：

```
USE AdventureWorks;
GO
-- 注册程序集
CREATE ASSEMBLY CLRDDLTrigger
FROM 'C:\MyProject\CLRDDLTrigger\CLRDDLTrigger\obj\Debug\CLRDDLTrigger.dll'
WITH PERMISSION_SET = SAFE;
GO

-- 创建基于程序集的触发器
CREATE TRIGGER AlterDropTableTrigger
ON DATABASE
WITH EXECUTE AS CALLER
FOR ALTER_TABLE, DROP_TABLE
AS
    EXTERNAL NAME CLRDDLTrigger.Triggers.AlterDropTableTrigger; -- 指定要使用的方法名称
GO
```

在 CREATE TRIGGER 中指定的触发器作用域、激发事件应当与 CLR 中的定义一致。上面语句中的 EXECUTE AS CALLER 指定执行 CLR 模块的用户不仅必须对模块本身拥有适当的权限，还要对模块引用的任何数据库对象拥有适当权限。

14.4 修改、删除和禁用触发器

为了改变一个触发器的定义，可以使用 DROP TRIGGER 语句删除触发器，然后重新创建，也可以使用 ALTER TRIGGER 语句直接重新定义。在删除表或视图的同时，会自动删除与之相关的触发器。

14.4.1 DML 触发器

下面的语句用于删除名为 MyTrigger 的触发器：

```
DROP TRIGGER MyTrigger
```

下面的语句用于修改 MyTrigger 触发器的定义，语法格式与 CREATE TRIGGER 基本相同。

```
ALTER TRIGGER MyTrigger
ON PriTable
AFTER DELETE
AS
    DELETE FROM DetailTable
    WHERE OrderID IN (SELECT OrderID
                      FROM Deleted);
GO
```

当不再需要某个触发器时，可以将其删除，也可以禁用它。被禁用后，触发器仍然作为对象存储于当前数据库中。但是，当执行 INSERT、UPDATE 或 DELETE 语句时，触发器将不再被激发。

要禁用表或视图的触发器，可以使用 `DISABLE TRIGGER` 语句。例如，下面的语句禁用 `dbo.PriTable` 表的 `PriTrigger` 触发器（注意不能对 DDL 触发器包含架构名称）。

```
DISABLE TRIGGER dbo.PriTrigger ON dbo.PriTable
```

也可以使用 `ALTER TABLE` 语句来禁用表的触发器，但是无法使用该语句或 `ALTER VIEW` 语句禁用视图的触发器。参考下面的语句：

```
ALTER TABLE dbo.PriTable
    DISABLE TRIGGER PriTrigger;
```

已禁用的触发器可以被重新启用。启用表的触发器可以使用 `ENABLE TRIGGER` 语句或 `ALTER TABLE` 语句，要启用视图的触发器则必须使用 `ENABLE TRIGGER`。

```
ENABLE TRIGGER dbo.PriTrigger ON dbo.PriTable
-- 或
ALTER TABLE dbo.PriTable
    ENABLE TRIGGER PriTrigger;
```

14.4.2 DDL 触发器

要删除一个 DDL 触发器，需要指定触发器的作用域范围，否则将默认为要删除 DML 触发器。例如，以下语句分别用于删除数据库范围的 DDL 触发器 `MyTrigger` 和服务器范围的 DDL 触发器 `MyOtherTrigger`：

```
DROP TRIGGER MyTrigger
ON DATABASE;
GO

DROP TRIGGER MyOtherTrigger
ON ALL SERVER;
```

要禁用和启用 DDL 触发器，也需要指定触发器的作用域范围。例如，下面的语句分别用于禁用和启用 `Safety` 触发器：

```
DISABLE TRIGGER Safety
ON DATABASE;

ENABLE TRIGGER Safety
ON DATABASE;
```

要修改一个 DDL 触发器，也应当使用 ALTER TRIGGER 语句，如：

```
ALTER TRIGGER Safety
ON DATABASE
FOR CREATE_TABLE
AS
    PRINT N'CREATE TABLE 出错';
    SELECT EVENTDATA().value('(/EVENT_INSTANCE/TSQLCommand/CommandText)[1]','nvarchar(max)');
    ROLLBACK;
```

14.4.3 CLR 触发器

删除一个 CLR 触发器与普通的 DML 和 DDL 触发器没有任何不同。在删除 CLR 触发器后，才可以删除与之相应的程序集。参考下面的语句：

```
USE Adventureworks;
GO

-- 删除 CLR DML 触发器和程序集
DROP TRIGGER MyTableUpdDelTrigger;
DROP ASSEMBLY CLRDDLTrigger;
GO

-- 删除 CLR DDL 触发器和程序集
DROP TRIGGER AlterDropTableTrigger
ON DATABASE;
DROP ASSEMBLY CLRDDLTrigger;
```

要禁用 CLR 触发器，参考本节中对 DML 和 DDL 触发器中介绍。





第 15 章 用户自定义函数

为了一些特定需要，用户可以在数据库服务器上创建自定义函数（User-Defined Functions, UDF），以扩展服务器的功能。用户定义函数可以接受参数、执行操作（例如复杂计算）并将操作结果以值的形式返回，返回值可以是单个标量值或结果集。用户定义函数最多可以有 1024 个输入参数。但是，用户定义函数不支持输出参数。

可分别使用 CREATE FUNCTION、ALTER FUNCTION 和 DROP FUNCTION 语句来分别实现用户定义函数的创建、修改和删除。

在创建用户自定义函数时，允许在函数主体内使用的 SQL 语句包括以下几类。

- DECLARE 语句，该语句可用于定义函数局部的数据变量和游标。
- 为函数局部对象的赋值语句，如使用 SET 为标量和表局部变量赋值。可以使用 INSERT、UPDATE 和 DELETE 语句，用于修改函数内的局部表变量。注意，这些语句仅能对函数内的局部对象（如局部游标或局部变量）进行更改。
- 游标操作，该操作引用在函数中声明、打开、关闭和释放的局部游标。不允许使用 FETCH 语句将数据返回到客户端。仅允许使用 FETCH 语句通过 INTO 子句给局部变量赋值。
- 除 TRY...CATCH 语句之外的控制流语句。
- SELECT 语句，该语句包含具有为函数的局部变量赋值的表达式的选择列表。
- EXECUTE 语句，该语句调用扩展存储过程。

不能在函数中执行的操作包括：对数据库表的修改，对不在函数上的局部游标进行操作，发送电子邮件，尝试修改目录，以及生成返回至用户的结果集。

同时，不确定性函数 NEWID、RAND、NEWSEQUENTIALID 和 TEXTPTR，不能在用户自定义函数中使用。

根据函数的返回值多少，可以将函数分为标量 UDF 和表值 UDF。如果函数仅返回单个值，则称为标量 UDF；如果函数返回一个表，则称为表值 UDF。此外，在 SQL Server 中也允许创建 CLR UDF。

15.1 标量 UDF

可以使用 CREATE FUNCTION 语句创建 UDF，语法格式与定义存储过程、触发器类似，但是

在调用 UDF 时比存储过程要严格，必须为函数限定架构名称，并且不允许忽略可选参数。不过在返回值方面，标量 UDF 有一个很好的性能设置选项：RETURNS NULL ON NULL INPUT，它指示当接收到的任何一个参数为 NULL 时，立即返回 NULL，而不再调用函数体。默认情况下，即使参数为 NULL，也将调用函数体。

下面的示例创建了一个多语句的标量 UDF，用于返回指定客户的所有订单金额的合计。对于多语句标量 UDF，必须将多个 Transact-SQL 语句包含在 BEGIN...END 块中。此函数需要输入一个 int 型的客户 ID 值，返回一个单个数据值。

```
CREATE FUNCTION dbo.GetCustTotalOrder(@CustomerID int)
RETURNS money
AS
-- 返回指定客户的所有订单金额的合计
BEGIN
    DECLARE @TotalOrder money;
    SELECT @TotalOrder = SUM(TotalDue)
    FROM Sales.SalesOrderHeader
    WHERE CustomerID = @CustomerID;

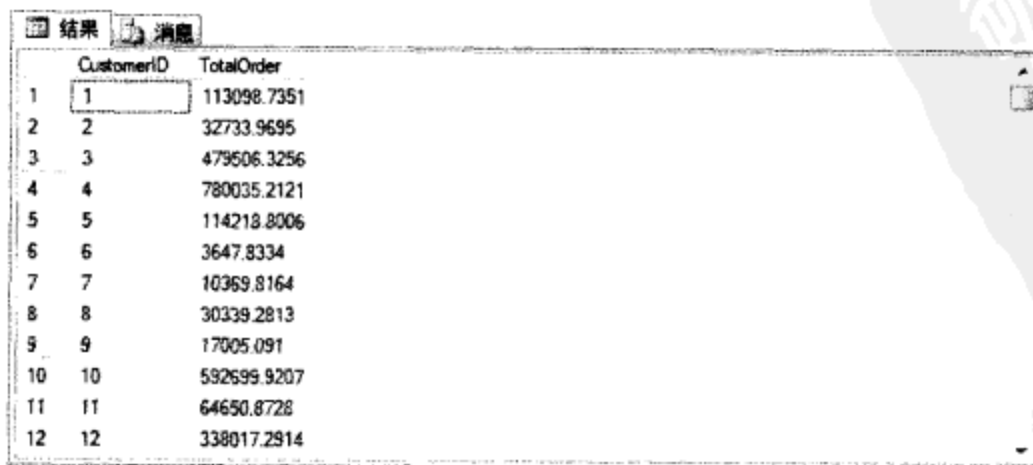
    IF (@TotalOrder IS NULL)
        SET @TotalOrder = 0
    RETURN @TotalOrder
END;
```

其中，GetCustTotalOrder 是函数的名称，@CustomerID 是声明的参数，所有参数应当包含在括号内，参数之间使用逗号分割。参数列表后面的 RETURNS 子句用于指定函数返回值的类型。这可以是除 text、ntext、image、cursor、table 和 timestamp 外的任意数据类型。在函数的最后，需要使用 RETURN 语句返回值，返回值的数据类型应当与 RETURNS 子句中声明的一致。

在任何可以引用服务器内部函数的地方都可以引 UDF。例如，下面是在 SET 语句中使用 UDF：

```
DECLARE @TotalOrder money
SET @TotalOrder = dbo.GetCustTotalOrder(1) -- 将客户 ID 为 1 的客户的所有订单金额合计存储到变量 TotalOrder 中
```

下面是在 SELECT 语句中使用 UDF，表中的 CustomerID 列将作为参数传递给 dbo.GetCustTotalOrder 函数，得到的结果集如图 15-1 所示。



	CustomerID	TotalOrder
1	1	113098.7351
2	2	32733.9695
3	3	479506.3256
4	4	780035.2121
5	5	114218.8006
6	6	3647.8334
7	7	10369.8164
8	8	30339.2813
9	9	17005.091
10	10	592699.9207
11	11	64650.8728
12	12	338017.2914

图 15-1 结果集中的 TotalOrder 列是使用用户自定义函数得到的


```
SELECT DISTINCT CustomerID, dbo.GetCustTotalOrder(CustomerID) AS TotalOrder
FROM Sales.SalesOrderHeader
ORDER BY CustomerID;
```

15.2 表值 UDF

表值 UDF 可以返回一个表，一般用于外部查询的 FROM 子句。创建表值 UDF 有两种方式：内联式和多语句式。

15.2.1 使用内联式表值 UDF 实现参数化视图功能

在内联式表值 UDF 中，仅包含一个用于生成函数返回值的 SELECT 查询语句。RETURNS 子句只包含关键字 table，不必定义返回值的数据格式。由于仅有一行语句，因此函数主体中也不需要开始使用 BEGIN...END 块。

实际上，内联式表值用户自定义函数的一个主要功能就是实现类似参数化视图的行为。在 SQL Server 中，不允许在视图的 WHERE 子句中使用参数作为搜索条件。例如，下面的视图指定返回 Washington 地区的商店名称和城市名称。如果希望返回其他地区的信息，则需要重新定义视图，将 Washington 替换为其他地区名称。

```
USE AdventureWorks;
GO
CREATE VIEW CustomersByRegion
AS
SELECT DISTINCT S.Name AS Store, A.City
FROM Sales.Store AS S
JOIN Sales.CustomerAddress AS CA ON CA.CustomerID = S.CustomerID
JOIN Person.Address AS A ON A.AddressID = CA.AddressID
JOIN Person.StateProvince AS SP ON
    SP.StateProvinceID = A.StateProvinceID
WHERE SP.Name = N'Washington';
```

不能在视图使用参数，限制了视图的灵活性。但是，内联式用户定义函数支持在 WHERE 子句中使用参数。下面的示例创建了一个允许用户在查询中指定区域的内联函数：

```
USE AdventureWorks;
GO
CREATE FUNCTION Sales.ufn_CustomerNamesInRegion( @Region nvarchar(50) )
RETURNS table
AS
RETURN (
    SELECT DISTINCT S.Name AS Store, A.City
    FROM Sales.Store AS S
    JOIN Sales.CustomerAddress AS CA ON CA.CustomerID = S.CustomerID
    JOIN Person.Address AS A ON A.AddressID = CA.AddressID
    JOIN Person.StateProvince AS SP ON
        SP.StateProvinceID = A.StateProvinceID
    WHERE SP.Name = @Region
)
```

执行下面的语句，将返回 Washington 地区的商店名称和城市名称。

```
SELECT *
FROM Sales.ufn_CustomerNamesInRegion(N'Washington');
```

15.2.2 使用多语句式表值 UDF 进行复杂计算

多语句式表值 UDF 的功能更为强大。内联式表值 UDF 和视图只能包含单个 SELECT 语句，而多语句式表值 UDF 可包含更多语句，这些语句的逻辑功能使得可以进行更加复杂的计算。在一定范围内，多语句式 UDF 可以替代返回单个结果集的存储过程，并且用户定义函数返回的 table 可在 FROM 子句中引用，而存储过程返回的结果集不能。

在多语句式表值 UDF 中，RETURNS 子句定义了特殊的 table 数据类型的局部变量。table 变量的声明可以包含列定义、主键约束、唯一性约束和 CHECK。在函数主体内，可以使用 INSERT、UPDATE 和 DELETE 语句对这个 table 变量定义的表进行数据修改。当执行 RETURN 语句时，这个定义的表的当前内容作为函数值被返回。RETURN 语句不能有参数。

例如，下面是 SQL Server 附带的 AdventureWorks 数据库中 ufnGetContactInformation 用户自定义函数的代码，用于返回指定联系人的信息。

```
USE AdventureWorks;
GO
CREATE FUNCTION dbo.ufnGetContactInformation(@ContactID int)
RETURNS @retContactInformation TABLE
(
    -- Columns returned by the function
    ContactID int PRIMARY KEY NOT NULL,
    FirstName nvarchar(50) NULL,
    LastName nvarchar(50) NULL,
    JobTitle nvarchar(50) NULL,
    ContactType nvarchar(50) NULL
)
AS
-- 返回指定联系人姓名、职业称谓和联系人的类型
BEGIN
    DECLARE
        @FirstName nvarchar(50),
        @LastName nvarchar(50),
        @JobTitle nvarchar(50),
        @ContactType nvarchar(50);
    -- Get common contact information
    SELECT
        @ContactID = ContactID,
        @FirstName = FirstName,
        @LastName = LastName
    FROM Person.Contact
    WHERE ContactID = @ContactID;
    SELECT @JobTitle =
        CASE
            -- Check for employee
            WHEN EXISTS(SELECT * FROM HumanResources.Employee e
                WHERE e.ContactID = @ContactID)
```

```

        THEN (SELECT Title
              FROM HumanResources.Employee
              WHERE ContactID = @ContactID)
    -- Check for vendor
    WHEN EXISTS(SELECT * FROM Purchasing.VendorContact vc
              INNER JOIN Person.ContactType ct
              ON vc.ContactTypeID = ct.ContactTypeID
              WHERE vc.ContactID = @ContactID)
    THEN (SELECT ct.Name
          FROM Purchasing.VendorContact vc
          INNER JOIN Person.ContactType ct
          ON vc.ContactTypeID = ct.ContactTypeID
          WHERE vc.ContactID = @ContactID)
    -- Check for store
    WHEN EXISTS(SELECT * FROM Sales.StoreContact sc
              INNER JOIN Person.ContactType ct
              ON sc.ContactTypeID = ct.ContactTypeID
              WHERE sc.ContactID = @ContactID)
    THEN (SELECT ct.Name
          FROM Sales.StoreContact sc
          INNER JOIN Person.ContactType ct
          ON sc.ContactTypeID = ct.ContactTypeID
          WHERE ContactID = @ContactID)
    ELSE NULL
END;
SET @ContactType =
CASE
    -- Check for employee
    WHEN EXISTS(SELECT * FROM HumanResources.Employee e
              WHERE e.ContactID = @ContactID)
    THEN 'Employee'
    -- Check for vendor
    WHEN EXISTS(SELECT * FROM Purchasing.VendorContact vc
              INNER JOIN Person.ContactType ct
              ON vc.ContactTypeID = ct.ContactTypeID
              WHERE vc.ContactID = @ContactID)
    THEN 'Vendor Contact'
    -- Check for store
    WHEN EXISTS(SELECT * FROM Sales.StoreContact sc
              INNER JOIN Person.ContactType ct
              ON sc.ContactTypeID = ct.ContactTypeID
              WHERE sc.ContactID = @ContactID)
    THEN 'Store Contact'
    -- Check for individual consumer
    WHEN EXISTS(SELECT * FROM Sales.Individual i
              WHERE i.ContactID = @ContactID)
    THEN 'Consumer'
END;
-- Return the information to the caller
IF @ContactID IS NOT NULL
BEGIN
    INSERT @retContactInformation -- 将值插入到表中
    SELECT @ContactID, @FirstName, @LastName, @JobTitle, @ContactType;
END;
RETURN; -- 返回表值
END;

```

执行下面的语句，返回 Person.Contact 中 ContactID 为 2200 的联系人的信息，如图 15-2 所示。

```

SELECT ContactID, FirstName, LastName, JobTitle, ContactType
FROM dbo.ufnGetContactInformation(2200);

```

ContactID	FirstName	LastName	JobTitle	ContactType	
1	2200	Amanda	Bailey	NULL	Consumer

图 15-2 由多语句式表值用户自定义函数返回的信息

15.3 CLR UDF

使用 CLR 既可以创建标量 UDF，也可以创建表值 UDF，通过 `SqlUserDefinedAggregate` 属性还可以创建聚合 UDF

15.3.1 标量 UDF

CLR 标量函数在 .NET Framework 程序集中是作为一个类的方法执行的，输入参数和由标量函数返回值可以是除 `rowversion`、`text`、`ntext`、`image`、`timestamp`、`table` 和 `cursor` 之外的任意 SQL Server 支持的类型，必须保证 SQL Server 数据类型与所执行方法返回值的数据类型相匹配。

创建 CLR UDF 的步骤与创建 CLR 存储过程和 CLR 触发器基本一致，下面简单介绍一下该过程。

打开 Visual Studio，从菜单中依次选择“文件”→“新建”→“项目”，打开“新建项目”对话框。选定 Visual C# 下的“数据库”节点，并在对话框的下方指定项目名称（如 `CLRFun`）和存储位置，最后单击“确定”按钮。

在出现的“新建数据库引用”对话框中，指定要连接的服务器和数据库名称，并单击“确定”按钮。

从 Visual Studio 的菜单中依次选择“项目”→“添加用户定义的函数”，打开如图 15-3 所示的“添加新项”对话框，在“名称”文本框中为函数指定一个名称，如 `ReturnOrderCount.cs`。

单击“添加”按钮后，Visual Studio 将自动在函数中添加如下代码：

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

public partial class UserDefinedFunctions
{
    [Microsoft.SqlServer.Server.SqlFunction]
```

```

public static SqlString ReturnOrderCount()
{
    // 在此处放置代码
    return new SqlString("Hello");
}
};

```

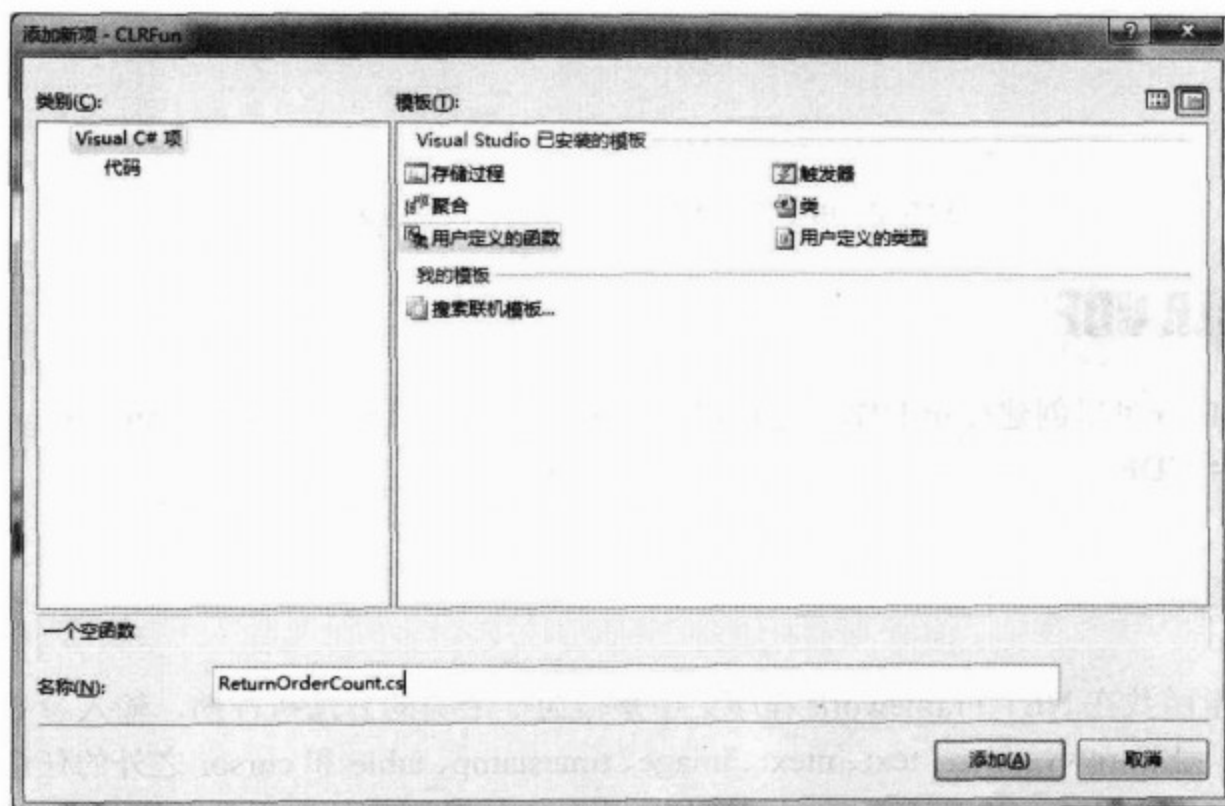


图 15-3 “添加新项”对话框

其中的[Microsoft.SqlServer.Server.SqlFunction]指定将用户定义的方法标记为 SQL Server 函数，UserDefinedFunctions 是类的名称，ReturnOrderCount 是方法的名称。

下面是函数的完整代码，用于根据指定的客户 ID 返回其在 Sales.SalesOrderHeader 表中的订单数量。其中为 DataAccess 属性指定的 DataAccessKind.Read 枚举值，表示要执行的方法需要使用 ADO.NET 并通过“上下文连接”来读取用户数据。

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

public partial class UserDefinedFunctions
{
    [Microsoft.SqlServer.Server.SqlFunction(DataAccess = DataAccessKind.Read)]
    //返回指定客户的订单数量
    public static int ReturnOrderCount(int custID)
    {
        using (SqlConnection conn = new SqlConnection("context connection=true"))
        {
            conn.Open();
            SqlCommand cmd = new SqlCommand();
            cmd.Connection = conn;
            cmd.CommandText = "SELECT COUNT(*) AS 'Order Count' FROM Sales.SalesOrderHeader " +

```



```

        "WHERE CustomerID = @custID";
cmd.Parameters.AddWithValue("@custID", custID); //设置 cmd 所使用的参数的值
return (int)cmd.ExecuteScalar();
    }
};

```

在 Visual Studio 中生成并部署项目，部署成功后，在 SQL Server Management Studio 中依次展开用户数据库下的“可编程性”、“函数”、“标量函数”节点，可以看到 CLR 函数已经被添加到了数据库中，如图 15-4 所示。执行下面的测试语句，将返回 Sales.SalesOrderHeader 表中指定客户的订单数量。

```
SELECT dbo.ReturnOrderCount(1);
```

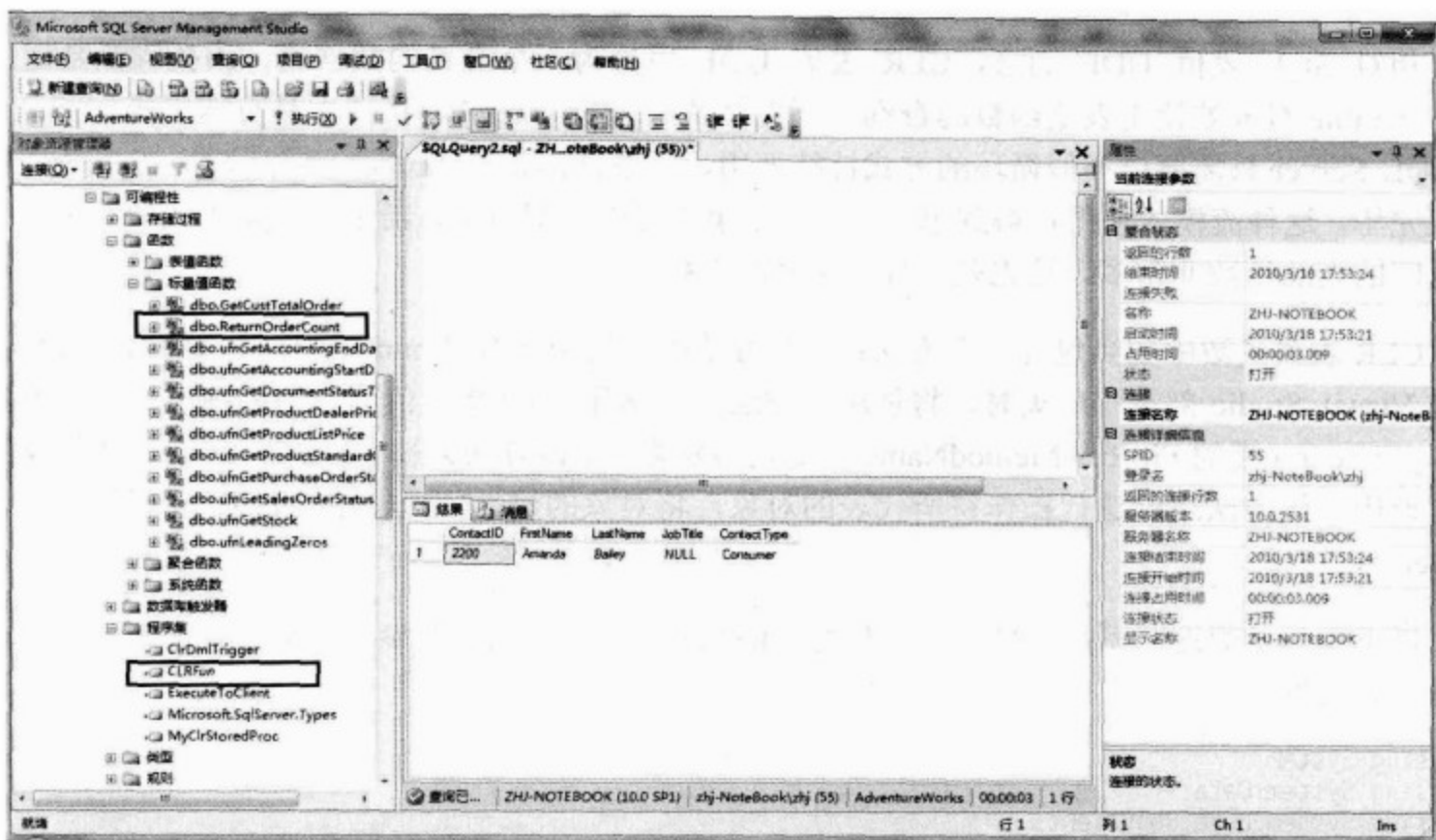


图 15-4 被部署的 CLR 标量函数

下面是使用 SQL 语句注册程序集并创建 CLR 标量 UDF 的方法。

```

USE AdventureWorks
GO

CREATE ASSEMBLY CLRFun
FROM 'C:\MyProject\CLRFun\CLRFun\obj\Debug\CLRFun.dll'; -- 注册程序集
GO
CREATE FUNCTION ReturnOrderCount(@custID int) -- 创建 CLR 标量函数
RETURNS int
WITH EXECUTE AS CALLER
AS
EXTERNAL NAME CLRFun.UserDefinedFunctions.ReturnOrderCount;
GO

```

15.3.2 表值 UDF

对于一个 CLR 表值 UDF，由函数返回的数据应当通过一个 `IEnumerable` 对象或 `IEnumerator` 对象来完成。所返回表中不能包含 `timestamp` 列或非 Unicode 字符串列（如 `char`、`varchar` 和 `text`），并且不支持 `NOT NULL` 约束。

CLR 表值 UDF 也是 .NET Framework 程序集的一个方法，表值函数代码应当包含 `IEnumerable` 接口。`IEnumerable` 接口在 .NET Framework 的 `System.Collections` 命名空间中定义，用来获得 .NET Framework 中的数组和集合。

相对 SQL 表值 UDF 而言，CLR 表值 UDF 表现为一个交替的数据流，由托管函数返回的 `IEnumerable` 对象被使用表值函数的查询计划直接调用，`IEnumerable` 接口中具有迭代操作符，它允许 SQL Server 只需要以类似循环的方式持续调用，逐条读出记录将立即返回到 SQL Server，直至读取完毕。这种流模式使得在得到第一行后，结果集可以立即可用，而不用等待生成整个表。如果要返回的表的行数非常多，这也是一种非常好的选择。

CLR 表值函数中应当包含两个方法：一个方法用于返回含有 `IEnumerable` 接口的对象实例，在通过 Visual Studio 部署程序集时，将使用该方法的名称作为 CLR 表值函数的名称。另一个方法是由 `SqlFunction` 类的 `FillRowMethodName` 指定的方法名称，该方法会被 SQL Server 通过迭代操作符重复调用。该方法通过迭代操作符所代表的对象，将对象的属性值以输出参数的形式返回给 SQL Server。

以下表值函数返回系统事件日志的信息，函数具有一个 `string` 型参数，用于指定一个要读取的事件日志名称。

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
using System.Collections;
using System.Diagnostics;

public partial class UserDefinedFunctions
{
    [SqlFunction(FillRowMethodName = "FillRow",
        TableDefinition = "logTime datetime,Message nvarchar(4000).Category nvarchar(4000).InstanceId
        bigint")]
    public static IEnumerable GetSystemEvent(String logname)
    {
        return new EventLog(logname, Environment.MachineName).Entries;
    }

    public static void FillRow(Object obj, out SqlDateTime timeWritten, out SqlChars message, out SqlChars
        category, out long instanceId)
    {
        EventLogEntry eventLogEntry = (EventLogEntry)obj;
```

```

        timeWritten = new SqlDateTime(eventLogEntry.TimeWritten);
        message = new SqlChars(eventLogEntry.Message);
        category = new SqlChars(eventLogEntry.Category);
        instanceId = eventLogEntry.InstanceId;
    }
}

```

其中的 `GetSystemEvent` 方法通过 `EventLog` 返回 Windows 操作系统的事件记录，由 `Entries` 属性返回的 `System.Diagnostics.EventLogEntryCollection` 对象含有 `GetEnumerator` 方法，支持迭代。在 SQL Server 通过由迭代操作符获得的 `EventLogEntry` 实例后，将其传递给 `FillRow` 方法，并通过输出参数来获取数据。SQL Server 会重复指定该方法，直至读取完毕。

如果要在 Visual Studio 中部署程序集，应当通过 `SqlFunction` 的 `TableDefinition` 属性指定要输出表的结构。由该属性指定的表定义将出现在 `CREATE FUNCTION` 语句的 `table` 变量声明中。

要读取操作系统的事件日志，应当将程序集的注册权限设置为 `UNSAFE`。在 Visual Studio 中，可以打开项目的属性选项卡，在“数据库”页面中设置权限级别为“不安全”，如图 15-5 所示。

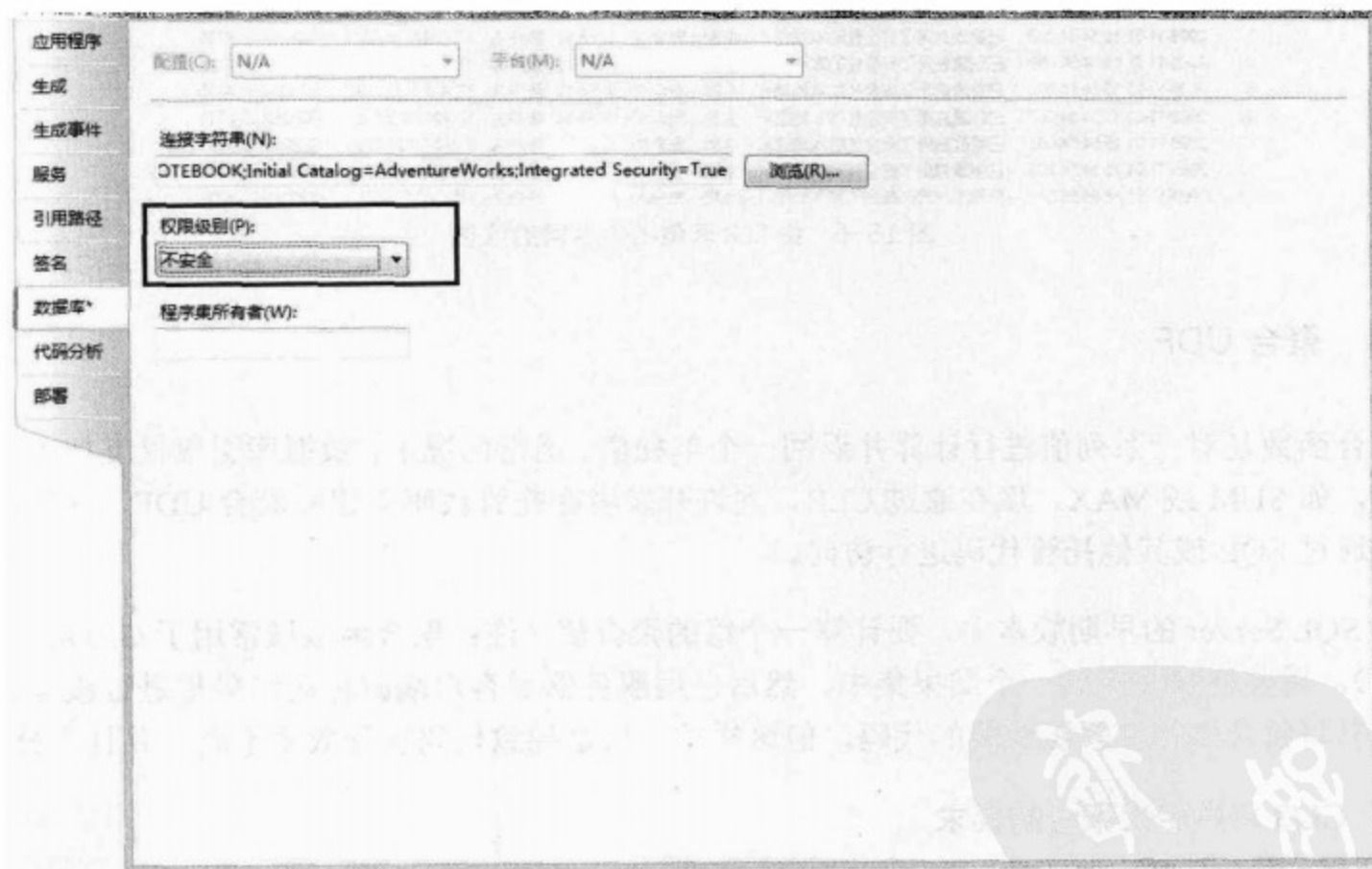


图 15-5 设置程序集的注册权限

要允许将程序集注册权限设置为 `UNSAFE`，应当打开数据库的 `TRUSTWORTHY` 属性设置。然后选择 Visual Studio 的“生成”，并进行部署。

```

ALTER DATABASE AdventureWorks
SET TRUSTWORTHY ON

```

下面是通过 Transact-SQL 语句注册程序集和创建表值函数的方法：

```

CREATE ASSEMBLY CLRFun
FROM 'C:\MyProject\ClrTableFun\ClrTableFun\bin\Debug\CLRTableFun.dll'
WITH PERMISSION_SET = UNSAFE;
GO

CREATE FUNCTION GetSystemEvent(@logname nvarchar(100))
RETURNS TABLE
(logTime datetime, Message nvarchar(4000), Category nvarchar(4000), InstanceId bigint)
AS
EXTERNAL NAME ClrTableFun.UserDefinedFunctions.GetSystemEvent;

```

执行以下测试语句，可以看到如图 15-6 所示的类似信息。

```

SELECT TOP 100 *
FROM dbo.ReadEventLog(N'Security') as T

```

	logTime	Message	Category	InstanceId
1	2009-11-01 15:54:51.000	Windows 正在启动。 当 LSASS.EXE 启动且审核子系统进行初始化时，会记录此事...	(12288)	4608
2	2009-11-01 15:54:51.000	已成功登录帐户。 主题: 安全 ID: S-1-0-0 帐户名: - 帐户域: - 登录 ID: 0x0 登...	(12544)	4624
3	2009-11-01 15:54:54.000	已创建新用户审核策略表。 元素的数量: 0 策略 ID: 0x32b32	(13568)	4902
4	2009-11-01 15:54:54.000	已创建启用了安全性的本地组。 主题: 安全 ID: S-1-5-18 帐户名: 37L4247E29-32\$...	(13826)	4731
5	2009-11-01 15:54:54.000	已更改启用了安全性的本地组。 主题: 安全 ID: S-1-5-18 帐户名: 37L4247E29-32\$...	(13826)	4735
6	2009-11-01 15:54:54.000	已创建启用了安全性的本地组。 主题: 安全 ID: S-1-5-18 帐户名: 37L4247E29-32\$...	(13826)	4731
7	2009-11-01 15:54:54.000	已更改启用了安全性的本地组。 主题: 安全 ID: S-1-5-18 帐户名: 37L4247E29-32\$...	(13826)	4735
8	2009-11-01 15:54:54.000	已创建启用了安全性的本地组。 主题: 安全 ID: S-1-5-18 帐户名: 37L4247E29-32\$...	(13826)	4731
9	2009-11-01 15:54:54.000	已更改启用了安全性的本地组。 主题: 安全 ID: S-1-5-18 帐户名: 37L4247E29-32\$...	(13826)	4735
10	2009-11-01 15:54:54.000	已创建启用了安全性的本地组。 主题: 安全 ID: S-1-5-18 帐户名: 37L4247E29-32\$...	(13826)	4731
11	2009-11-01 15:54:54.000	已更改启用了安全性的本地组。 主题: 安全 ID: S-1-5-18 帐户名: 37L4247E29-32\$...	(13826)	4735
12	2009-11-01 15:54:54.000	已创建启用了安全性的本地组。 主题: 安全 ID: S-1-5-18 帐户名: 37L4247E29-32\$...	(13826)	4731
13	2009-11-01 15:54:54.000	已更改启用了安全性的本地组。 主题: 安全 ID: S-1-5-18 帐户名: 37L4247E29-32\$...	(13826)	4735

图 15-6 由 CLR 表值函数返回的信息

15.3.3 聚合 UDF

聚合函数是对一系列值进行计算并返回一个单独值。通常情况下，数据库引擎仅支持内置的聚合函数，如 SUM 或 MAX。现在通过 CLR，允许开发者在托管代码中建立聚合 UDF，并且这些函数可以通过 SQL 或其他托管代码进行访问。

在 SQL Server 的早期版本中，要计算一个组的聚合值（注：聚合函数通常用于 GROUP BY 分组计算），需要将值检索到一个结果集中，然后使用服务器或客户端游标对结果集进行枚举。这要求必须书写包含迭代和累积逻辑的代码，但这样的结果是导致代码执行效率不高，并且十分复杂。

1. CLR 用户定义聚合的要求

在 CLR 程序集中，只要执行了一些必要聚合协定，该类型就可以被注册为一个用户定义聚合函数。这个协定有 `SqlUserDefinedAggregate` 属性和聚合协定方法组成，聚合协定包含存储聚合中间状态的机制，并且这个机制可以累积新值，而聚合协定方法包括 `Init`、`Accumulate`、`Merge` 和 `Terminate`。

(1) `SqlUserDefinedAggregate`。

每个用户定义聚合函数必须使用 `SqlUserDefinedAggregate` 属性（Attribute）进行声明，这个属性指示 SQL Server 该类型遵从用户定义聚合协定。`SqlUserDefinedAggregate` 必须设置 `Format` 和

MaxByteSize 属性 (Property)，用来控制使用的序列化格式。表 15-1 对这两个属性进行了说明。

表 15-1 Format 和 MaxByteSize 属性

属 性	说 明
Format	类型的序列化格式，可以是 Native 或 UserDefined
MaxByteSize	在计算期间需要存储聚合状态的最大字节数，最大值为 8000。该属性应用于 UserDefined 格式，Native 格式序列化无须指定此属性。对于指定了 UserDefined 序列化的聚合，MaxByteSize 是指序列化的数据的总大小。以一个序列化包含 10 个字符 (Char) 的字符串的聚合为例，当使用 BinaryWriter 序列化该字符串时，序列化后的字符串的总大小为 22 个字节：每个 Unicode UTF-16 字符占据的字节数 2 乘以最大字符数 10，再加上序列化二进制流所引入的开销占用的 2 个控制字节。因此，在确定 MaxByteSize 的值时，必须考虑序列化的数据的总大小：二进制形式的序列化数据的大小加上序列化引入的开销

当查询处理器需要传播聚合的临时结果到工作表时才使用序列。

(2) 聚合方法。

作为用户定义聚合函数注册的类型应当支持下列实例方法，用于计算聚合。

```
public void Init(); /* needed for empty group */
```

查询处理器使用该方法来初始化聚合计算，在进行聚合时该方法每组调用一次。查询处理器允许为计算多个组的聚合值而再次调用相同的聚合类实例。Init 方法将对前一个实例执行一些必需的清理，并允许重新开始一个新的聚合计算。

```
public void Accumulate ( input-type value);
```

input_type 应当是与在 CREATE AGGREGATE 中指定的本机 SQL Server 数据类型匹配的托管 SQL Server 数据类型，可参考 13.3.1 小节中的表 13-2。

对于用户定义数据类型 (UDT)，input-type 与 UDT 类型相同。查询处理器使用该方法累积聚合值，该方法在组中的每个值被聚合时调用一次。查询处理器总是在调用给定聚合类实例的 Init 方法后调用此方法，该方法将更新实例的状态，以反映被传递参数的累积值。

```
public void Merge( udagg_class value);
```

该方法用于将当前实例与另一个实例进行合并，查询处理器使用该方法合并一个聚合的多个部分计算结果。

```
public return_type Terminate();
```

该方法完成聚合计算并返回聚合结果，return_type 应当是与在 CREATE AGGREGATE 中指定的本机 SQL Server 数据类型匹配的托管 SQL Server 数据类型，return_type 也可以是用户自定义数据类型。

2. CLR 用户定义聚合的优化属性

SqlUserDefinedAggregate 还包含一些用于定义聚合算法的属性 (IsInvariantToDuplicates、

IsInvariantToNulls、IsInvariantToOrder 和 IsNullIfEmpty), 查询优化程序可以使用这些属性来查找更有效的查询执行计划。在缺省情况下, 所有这些属性的值为 false。表 15-2 列出了这些属性的功能。

表 15-2 CLR 用户定义聚合的优化属性

属 性	说 明
IsInvariantToDuplicates	指示聚合是否与重复值无关。如果聚合与重复值无关, 则此属性为 true。也就是说, S 和 {X} 的聚合与 S 中已经存在 X 时的 S 的聚合相同
IsInvariantToNulls	指示聚合是否与空值无关。如果聚合与空值无关, 则此属性为 true, 也就是说, S 和 {NULL} 的聚合与 S 的聚合相同
IsInvariantToOrder	指示聚合是否与顺序无关。留供将来使用。查询处理器当前不使用此属性
IsNullIfEmpty	指示在没有对任何值进行累积时聚合是否返回空引用 (在 Visual Basic 中为 Nothing)。为 true 时返回空引用

3. 调用聚合 UDF

在 SELECT 语句中, 在所有能够使用系统聚合函数的地方都可以使用聚合 UDF。调用聚合 UDF, 当前用户必须具有对 UDF 的 EXECUTE 权限。

下面的示例, 用于连接从一个表列中得到的字符串值。

```
using System;
using System.Data;
using Microsoft.SqlServer.Server;
using System.Data.SqlTypes;
using System.IO;
using System.Text;

[Serializable]
[SqlUserDefinedAggregate(
    Format.UserDefined, //使用用户定义格式
    IsInvariantToNulls = true, //优化属性
    IsInvariantToDuplicates = false, //优化属性
    IsInvariantToOrder = false, //优化属性
    MaxByteSize = 8000)
]
public class Concatenate : IBinarySerialize
{
    /// <summary>
    /// 该变量保持连接的中间结果
    /// </summary>
    private StringBuilder intermediateResult;

    /// <summary>
    /// 初始化内部数据结构
    /// </summary>
    public void Init()
    {
        this.intermediateResult = new StringBuilder();
    }

    /// <summary>
    /// 累积下一个值, 只要值不为空
    /// </summary>
    /// <param name="value"></param>
    public void Accumulate(SqlString value)
    {
```

```

        if (value.IsNull)
        {
            return;
        }

        this.intermediateResult.Append(value.Value).Append(',');
    }

    /// <summary>
    /// 将该聚合与部分计算的聚合进行合并
    /// </summary>
    /// <param name="other"></param>
    public void Merge(Concatenate other)
    {
        this.intermediateResult.Append(other.intermediateResult);
    }

    /// <summary>
    /// 在聚合的末尾调用，返回聚合结果
    /// </summary>
    /// <returns></returns>
    public SqlString Terminate()
    {
        string output = string.Empty;
        //删除尾部的逗号
        if (this.intermediateResult != null
            && this.intermediateResult.Length > 0)
        {
            output = this.intermediateResult.ToString(0, this.intermediateResult.Length - 1);
        }

        return new SqlString(output);
    }

    public void Read(BinaryReader r)
    {
        intermediateResult = new StringBuilder(r.ReadString());
    }

    public void Write(BinaryWriter w)
    {
        w.Write(this.intermediateResult.ToString());
    }
}

```

与大多数聚合一样，大部分的逻辑在 **Accumulate** 方法中。由参数传递来的字符串追加到 **StringBuilder** 对象中。假设不是第一次调用 **Accumulate** 方法，会在 **StringBuilder** 中追加一个逗号。在计算任务的结尾调用 **Terminate** 方法，以字符串方式返回 **StringBuilder**。

可以在 **Visual Studio** 中部署该程序集到 **SQL Server** 中，也可以在编译上述代码后，使用以下代码在 **SQL Server** 中注册该程序集并创建聚合函数。

```

CREATE ASSEMBLY MyAgg
FROM 'C:\MyProject\MyAgg\MyAgg\obj\Debug\MyAgg.dll';
GO

CREATE AGGREGATE Concatenate(@input nvarchar(200))
RETURNS nvarchar(max)
EXTERNAL NAME MyAgg.Concatenate;

```

执行下面的语句，创建一个演示该示例的表。

```
CREATE TABLE BookAuthors
(
    BookID int NOT NULL,
    AuthorName nvarchar(200) NOT NULL
);

-- 插入数据
INSERT BookAuthors VALUES(1, 'Johnson');
INSERT BookAuthors VALUES(2, 'Taylor');
INSERT BookAuthors VALUES(3, 'Steven');
INSERT BookAuthors VALUES(2, 'Mayler');
INSERT BookAuthors VALUES(3, 'Roberts');
INSERT BookAuthors VALUES(3, 'Michaels');
```

执行下面的查询，会将同一本书的所有作者放到一列中，结果集如图 15-7 所示。

```
SELECT BookID, dbo.Concatenate(AuthorName)
FROM BookAuthors
GROUP BY BookID;
```



BookID	Concatenate(AuthorName)
1	Johnson
2	Taylor, Mayler
3	Roberts, Michaels, Steven

图 15-7 使用 CLR 自定义聚合函数得到的结果

15.4 修改和删除 UDF

可以使用 ALTER FUNCTION 语句删除使用 SQL 或 CLR 编写的标量 UDF 和表值 UDF，该语句的语法格式与 CREATE FUNCTION 语句基本相同，在此不再赘述。对于 CLR 聚合 UDF，SQL Server 没有提供相应的修改语句，要修改这样的函数，应当将其删除后重建。

要删除一个不再使用的标量或表值用户自定义函数，可以使用 DROP FUNCTION 语句。例如，下面的语句删除了在前面创建的标量 UDF 和表值 UDF，并删除了函数所使用的程序集。

```
DROP FUNCTION dbo.ReturnOrderCount;
DROP FUNCTION dbo.GetSystemEvent;
GO
DROP ASSEMBLY CLRFun; --删除程序集
DROP ASSEMBLY CLRTableFun;
```

要删除一个聚合函数，应当使用 DROP AGGREGATE 语句，参考下面的示例：

```
DROP AGGREGATE dbo.Concatenate;
GO
DROP ASSEMBLY MyAgg;
```

第 3 部分 性能调整篇

第 16 章 事务处理

第 17 章 并发访问控制

第 18 章 查询的优化与执行



第 16 章 事务处理

当对多个表或一个表中的多行进行数据更新时，需要考虑数据的完整性问题。例如，在 `SalesOrderHeader` 表中存储着每笔订单的日期、发货日期和订单金额等信息，而在 `SalesOrderDetail` 中则保存着每笔订单中所有订购的商品信息。假设 `SalesOrderHeader` 在应用程序中被作为检索订单信息的主表，在准备从 `SalesOrderHeader` 中删除一条订单信息时，也需要从 `SalesOrderDetail` 中删除所有属于该订单的商品信息。下面的语句中，如果在执行完第 1 条 `DELETE` 语句后，由于某些原因（如程序被强迫关闭），第 2 条 `DELETE` 语句未能执行，则属于该订单的商品信息则有可能被永远的遗留在 `SalesOrderDetail` 表中。

```
DELETE FROM Sales.SalesOrderHeader  
WHERE SalesOrderID = 43659 ;
```

```
DELETE FROM Sales.SalesOrderDetail  
WHERE SalesOrderID = 43659 ;
```

要解决遗留数据问题，应当将上面的语句包含在一个事务中，在第 2 条 `DELETE` 语句执行失败的情况下，回滚整个事务操作，从而撤消对 `SalesOrderHeader` 表的删除操作。在使用事务处理的情况下，事务内的多行操作要么全部完成，要么回滚所有操作，这为确保表间数据的关联性提供了保障。

事务处理可以分为自动事务处理、显式事务处理和隐式事务处理。

16.1 自动事务处理

自动事务处理是 SQL Server 数据库引擎的默认事务管理模式。当每个 SQL 语句在完成时，不是被提交就是被回滚。如果语句能够成功地完成，则提交该语句；如果遇到错误，则回滚该语句。只要没有指定显式事务或隐性事务处理，与数据库引擎实例的连接都以此默认模式操作。并且当提交或回滚显式事务，或当关闭隐性事务模式时，连接也将重新返回到自动事务处理模式。

在自动事务处理模式下，当遇到的错误是编译错误而非运行时错误时，数据库引擎实例会阻止数据库引擎生成执行计划，所以批处理中的任何语句都不会执行。例如，在下面的语句中，由于第 3 个批中第 3 行 `INSERT` 语句的 `VALUES` 关键字错写成了 `VALUSE`，所以这个批不能生成执行计划，这 3 条 `INSERT` 语句都不会被执行。后面的第 4 个批也会因为前面的错误而终止执行。

```
USE AdventureWorks;  
GO --第 1 个批  
CREATE TABLE TestBatch (Col1a INT PRIMARY KEY, Col1b CHAR(3));
```



```
GO --第 2 个批
INSERT INTO TestBatch VALUES (1, 'aaa');
INSERT INTO TestBatch VALUES (2, 'bbb');
INSERT INTO TestBatch VALUSE (3, 'ccc'); -- 语法有错误
GO --第 3 个批
SELECT * FROM TestBatch; -- Returns no rows.
GO --第 4 个批
```

但是,在能够正常生成执行计划的情况下,在运行时出现的错误,并不会回滚发生错误前被正确执行并提交的操作。例如,在下面的语句中,只是由于第 3 个 INSERT 语句产生了重复键错误。由于前两个 INSERT 语句能够成功地执行并且提交,因此它们并不会被回滚。

```
USE AdventureWorks;
GO
CREATE TABLE TestBatch (Cola INT PRIMARY KEY, Colb CHAR(3));
GO
INSERT INTO TestBatch VALUES (1, 'aaa');
INSERT INTO TestBatch VALUES (2, 'bbb');
INSERT INTO TestBatch VALUES (1, 'ccc'); -- 重复键错误.
GO
SELECT * FROM TestBatch; -- 返回 1 和 2 行
```

在查询优化阶段,并不会解析对象名称,直到执行时,数据库引擎才确定对象名称能否关联到实际的数据库对象。在下面的语句中,虽然 TestBch 表并不存在,但是在编译时刻数据库引擎并不会发现此错误,所以能够生成执行计划。在执行时,前两条 INSERT 语句能够被正确执行并提交,因此它们并不会被回滚。而第 3 条 INSERT 语句由于引用一个不存在的表而产生错误。

```
USE AdventureWorks;
GO
CREATE TABLE TestBatch (Cola INT PRIMARY KEY, Colb CHAR(3));
GO
INSERT INTO TestBatch VALUES (1, 'aaa');
INSERT INTO TestBatch VALUES (2, 'bbb');
INSERT INTO TestBch VALUES (3, 'ccc'); -- 表名称错误
GO
SELECT * FROM TestBatch; -- 返回 1 和 2 行
```

16.2 显式事务处理

显式事务就是可以显式地定义事务开始和结束的事务。它使用 BEGIN TRANSACTION 标记事务的起始点。在没有遇到错误的情况下,使用 COMMIT TRANSACTION 或 COMMIT WORK 来提交修改并结束事务。如果遇到错误,则使用 ROLLBACK TRANSACTION 或 ROLLBACK WORK 回滚操作并结束事务。

显式事务模式持续的时间只限于该事务的持续期。当事务结束时,连接将返回到启动显式事务前所处的事务模式:隐式或自动模式。

要处理语句执行失败或错误等异常情况,应当在事务中使用 TRY...CATCH 构造。例如,在下面的示例中,将 INSERT 语句包含在了显式事务中,如果提交时发生错误或失败,则执行 BEGIN CATCH 语句,返回错误编号并回滚操作。

```

BEGIN TRY
    BEGIN TRANSACTION    -- 开始事务
        INSERT MyTable VALUES(3,'ASDF') ;    -- 插入数据
    COMMIT TRANSACTION    -- 提交插入操作
END TRY

-- 如果上面的 TRY 块中出现错误, 将执行下面 CATCH 块中的语句
BEGIN CATCH
    SELECT ERROR_NUMBER() AS ERRORNUMBER ;    -- ERROR_NUMBER 函数用于返回错误编号
    ROLLBACK TRANSACTION    -- 回滚事务操作
END CATCH

```

在显式事务中可以使用除了以下语句外的所有 SQL 语句（下面的语句不会引发隐式事务）：**ALTER DATABASE**、**BACKUP**、**CREATE DATABASE**、**DROP DATABASE**、**RECONFIGURE**、**RESTORE** 和 **UPDATE STATISTICS**。

另外, 不能使用 **sp_dboption** 存储过程来设置数据库选项, 也不能使用 **master** 数据库的系统过程。

16.3 隐式事务处理

用户可以使用 **SET IMPLICIT_TRANSACTIONS** 语句启动或关闭隐式事务模式。如果设置为 **ON**, 表示启动隐式事务处理; 如果设置为 **OFF**, 则表示关闭隐式事务处理。

当连接以隐性事务模式进行操作时, SQL Server 数据库引擎实例将在提交或回滚当前事务后自动启动新事务。无须描述事务的开始, 只需提交或回滚事务即可。

在为连接打开隐式事务处理后, 当数据库引擎实例首次执行下列任何语句时, 都会自动启动一个事务: **ALTER TABLE**、**CREATE**、**DELETE**、**DROP**、**FETCH**、**GRANT**、**INSERT**、**OPEN**、**REVOKE**、**SELECT**、**TRUNCATE TABLE** 和 **UPDATE**。

在发出 **COMMIT** 或 **ROLLBACK** 语句之前, 该事务将一直保持有效。在第 1 个事务被提交或回滚之后, 下次当连接执行以上任何语句时, 数据库引擎实例都将自动启动一个新事务。该实例将不断地生成隐性事务链, 直到隐性事务模式关闭为止。

下面我们比较一下隐式事务处理和自动事务处理之间的差异, 首先使用下面的语句创建一个示例表 **ImplicitTran**。

```

USE AdventureWorks;
GO
CREATE TABLE ImplicitTran
    (Cola int PRIMARY KEY,
     Colb char(3) NOT NULL);

```

下面的语句中虽然使用了 **COMMIT TRANSACTION** 和 **ROLLBACK TRANSACTION**, 但是由于并未声明隐式事务处理, 所以它们并不起作用, SQL Server 将使用自动事务处理模式。两条 **INSERT** 语句会被分成两个事务进行处理, 由于第 2 条 **INSERT** 语句由于违反了重复键, 所以会被回滚, 但是第 1 条 **INSERT** 语句可以被正常提交, 如图 16-1 所示。

```

BEGIN TRY
    INSERT INTO ImplicitTran VALUES (1, 'aaa');
    INSERT INTO ImplicitTran VALUES (1, 'bbb');
    -- 提交事务
    COMMIT TRANSACTION;
END TRY

-- 如果上面的 TRY 块中出现错误, 将执行下面 CATCH 块中的语句
BEGIN CATCH
    ROLLBACK TRANSACTION; --回滚事务操作
END CATCH
SELECT * FROM ImplicitTran;

```



	Cola	Colb
1	1	aaa

图 16-1 ImplicitTran 表中仅有一条记录

并且由于没有与 ROLLBACK TRANSACTION 匹配的 BEGIN TRANSACTION 语句, SQL Server 会返回如下错误信息。

```

消息 3903, 级别 16, 状态 1, 第 10 行
ROLLBACK TRANSACTION 请求没有对应的 BEGIN TRANSACTION。

```

删除表中刚插入的数据, 然后启动隐式事务处理。在这种模式下, 两条 INSERT 语句会被看成是一个事务, 由于第 2 条 INSERT 造成的错误, 也会回滚第 1 条 INSERT 语句操作。所以 ImplicitTran 表中不会有任何记录。

```

DELETE FROM ImplicitTran;
SET IMPLICIT_TRANSACTIONS ON; -- 启动隐式事务处理
GO

BEGIN TRY
    INSERT INTO ImplicitTran VALUES (1, 'aaa');
    INSERT INTO ImplicitTran VALUES (1, 'bbb');
    -- 提交事务
    COMMIT TRANSACTION;
END TRY

-- 如果上面的 TRY 块中出现错误, 将执行下面 CATCH 块中的语句
BEGIN CATCH
    ROLLBACK TRANSACTION; --回滚事务操作
END CATCH

SET IMPLICIT_TRANSACTIONS OFF; --关闭隐式事务处理
SELECT * FROM ImplicitTran;

```

16.4 使用嵌套事务

显式事务可以嵌套, 这主要是为了支持存储过程中的一些事务。在下面的示例中创建了一个 TransProc 存储过程, 在存储过程内部启用了事务处理机制, 并对操作进行了提交。当在另一个事务中调用 TransProc 时, 用户很可能会忽略 TransProc 中的嵌套事务, 而只是根据外部事务中的操

作采取提交或回滚操作。实际上，SQL Server 数据库引擎也是采用这种机制来处理嵌套事务的。即：数据库引擎只根据最外部事务结束时采取的操作来确定提交或者回滚内部事务。如果提交外部事务，则也将提交内部嵌套事务。如果回滚外部事务，则也将回滚所有内部事务，不管是否单独提交过内部事务。

```
USE AdventureWorks;
GO
CREATE TABLE TestTrans(ColA INT PRIMARY KEY,
    ColB CHAR(3) NOT NULL);
GO

CREATE PROCEDURE TransProc @PriKey INT, @CharCol CHAR(3)
AS
BEGIN
    BEGIN TRANSACTION InProc
    INSERT INTO TestTrans VALUES (@PriKey, @CharCol)
    INSERT INTO TestTrans VALUES (@PriKey + 1, @CharCol)
    COMMIT TRANSACTION InProc;
END
GO

-- 开始一个事务并执行 TransProc 存储过程
BEGIN TRANSACTION OutOfProc;
EXEC TransProc 1, 'aaa';
-- 回滚外部事务，也将回滚 TransProc 中的嵌套事务
ROLLBACK TRANSACTION OutOfProc;
GO

EXECUTE TransProc 3, 'bbb';
GO
-- 下面的 SELECT 语句显示表中仅有 ColA 列为 3 和 4 的行，而没有上面提交 1 和 2 行。
-- 这说明由第 1 个 EXECUTE 语句执行的提交动作被外部的回滚操作覆盖了，并没有
-- 执行。因为对于已实际提交的操作是无法回滚的
SELECT * FROM TestTrans;
```

上面的 SELECT 语句将返回如图 16-2 所示的结果集。



	ColA	ColB
1	3	bbb
2	4	bbb

图 16-2 结果集

在这个示例中，为了标识各个事务，在 BEGIN TRANSACTION 语句的后面都指定了一个事务名称参数。例如，BEGIN TRANSACTION InProc 语句指定事务的名称为 InProc。COMMIT TRANSACTION 语句和 ROLLBACK TRANSACTION 语句也与此类似。

对 COMMIT TRANSACTION 的每个调用都应用于最后执行的 BEGIN TRANSACTION。如果是嵌套 BEGIN TRANSACTION 语句，即使嵌套事务内部的 COMMIT TRANSACTION 语句引用了外部事务名称，该提交也只应用于最后一个嵌套的事务。

在 ROLLBACK TRANSACTION 语句中引用内部事务名称是非法的。如果在一组嵌套事务的任意级别执行使用外部事务名称或是没有指定事务名称的 ROLLBACK TRANSACTION 语句，则回滚所有嵌套事务。

仍旧使用上面的示例，只是对 TransProc 存储过程进行了一下修改，在提交错误时，将回滚 InProc 事务。由于 ROLLBACK TRANSACTION 语句不能引用内部事务名称，在出现错误需要回滚时，SQL Server 将提示无法找到该名称的事务或保存点。

```
USE AdventureWorks;
GO
CREATE TABLE TestTrans(ColA INT PRIMARY KEY,
    ColB CHAR(3) NOT NULL);
GO

CREATE PROCEDURE TransProc @PriKey INT, @CharCol CHAR(3)
AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION InProc --开始 InProc 事务
        INSERT INTO TestTrans VALUES (@PriKey, @CharCol)
        COMMIT TRANSACTION InProc; --提交事务
    END TRY

    -- 在出现提交错误时回滚操作，由于不能引用内部事务名称，在需要回滚时
    -- SQL Server 将提示找不到 InProc 事务名称
    BEGIN CATCH
        ROLLBACK TRANSACTION InProc
    END CATCH
END
GO

-- 插入一行
EXEC TransProc 1, 'aaa';
GO

-- 开始一个事务并执行 TransProc 存储过程
BEGIN TRANSACTION OutOfProc;
GO

-- 再次执行下面的 EXEC 语句会造成 TestTrans 表中出现重复键，因此引发错误，这会
-- 导致执行 TransProc 存储过程中的 CATCH 部分，进行回滚操作。此时，SQL Server
-- 提示无法回滚 InProc，找不到该名称的事务或保存点。全部提示信息如下：
/* (0 行受影响)
消息 6401，级别 16，状态 1，过程 TransProc，第 11 行
无法回滚 InProc。找不到该名称的事务或保存点。
消息 266，级别 16，状态 2，过程 TransProc，第 0 行
EXECUTE 后的事务计数指示缺少了 COMMIT 或 ROLLBACK TRANSACTION 语句。上一计数 = 1，当前计数 = 2。
*/
EXEC TransProc 1, 'aaa';
GO

-- 下面的语句用于回滚所有操作，退出嵌套事务
ROLLBACK TRANSACTION OutOfProc;
```

正确的做法是使用事务保存点，保存点提供了一种回滚部分事务的机制。可以使用 SAVE TRANSACTION savepoint_name 语句创建保存点，然后执行 ROLLBACK TRANSACTION

`savepoint_name` 语句以回滚到保存点，而不是回滚到事务的起点。可以将上面示例中的 `TransProc` 存储过程改写为以下形式，再次出现重复键等错误时，可以正确回滚内部事务中的部分操作，而不是回滚整个内部事务。

```
CREATE PROCEDURE TransProc @PriKey INT, @CharCol CHAR(3)
AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION InProc
        SAVE TRANSACTION ttt -- 设置保存点
        INSERT INTO TestTrans VALUES (@PriKey, @CharCol)
        COMMIT TRANSACTION InProc;
    END TRY

    BEGIN CATCH
        ROLLBACK TRANSACTION ttt -- 回滚到保存点
        -- 由于仅是回滚到保存点，所以仍旧需要使用 COMMIT 语句提交并结束事务
        -- 否则无法退出嵌套。有关嵌套层次的信息，参考后面的@@TRANCOUNT 函数介绍
        COMMIT TRANSACTION InProc
    END CATCH
END
```

使用使用嵌套事务时，需要注意事务的嵌套层次，尤其是在出现错误时，能够正确退出嵌套事务是很重要的。`@@TRANCOUNT` 函数用于记录当前事务的嵌套层次。每个 `BEGIN TRANSACTION` 语句使 `@@TRANCOUNT` 增加 1，每个 `COMMIT TRANSACTION` 语句使 `@@TRANCOUNT` 减去 1。没有指定事务名称或使用一组嵌套事务中最外部事务的事务名称的 `ROLLBACK TRANSACTION` 语句将回滚所有嵌套事务，并使 `@@TRANCOUNT` 减小到 0。在无法确定是否已经在事务中时，可以用 `SELECT @@TRANCOUNT` 确定 `@@TRANCOUNT` 是等于 1 还是大于 1。如果 `@@TRANCOUNT` 等于 0，则表明不在事务中。

16.5 使用事务保存点

在上节中我们使用事务保存点的方法正确处理了嵌套事务中的回滚操作。实际上，保存点主要用在很少出现错误的情况下，来回滚部分事务，因为这种仅在发生错误时才执行的回滚操作比让每个事务在更新之前测试更新的有效性更为有效。更新和回滚操作代价很大，因此只有在遇到错误的可能性很小，而且预先检查更新的有效性的代价相对很高的情况下，使用保存点才会非常有效。

下面的示例说明了保存点在一个订单系统中的使用情况。该系统中存货不足的可能性很小，因为该公司具备有效的供应商和分购点。通常应用程序在尝试更新订单记录时，会先验证目前是否有足够的存货。以下示例假定由于某种原因，验证目前可用存货量代价相对较大（由于连接到一个低速的调制解调器或广域网上）。可将应用程序编写为只进行更新，而且如果收到错误信息表明库存不足时，将回滚该更新。在这种情况下，在插入之后快速检查 `@@ERROR` 要比在更新之前验证库存数量的速度要快得多。

`InvCtrl` 表有一个 `CHECK` 约束，如果 `QtyInStk` 列低于 0，它就会触发 547 号错误。`OrderStock` 过程将创建一个保存点。如果出现 547 错误，它将回滚到该保存点，并将现有商品数返回给调用进

程；然后调用进程可以针对现有的数量重新下订单。如果 OrderStock 返回 0，表示当前有足够的存货来满足订购需要。

```

SET NOCOUNT OFF;
GO
USE AdventureWorks;
GO
CREATE TABLE InvCtrl
    (WhrhousID    int,
     PartNmbr     int,
     QtyInStk     int,
     ReordrPt     int,
     CONSTRAINT InvPK PRIMARY KEY
      (WhrhousID, PartNmbr),
     CONSTRAINT QtyStkCheck CHECK (QtyInStk > 0));
GO
CREATE PROCEDURE OrderStock
    @WhrhousID int,
    @PartNmbr int,
    @OrderQty int
AS
DECLARE @ErrorVar int;
BEGIN TRY
    BEGIN TRANSACTION InProc;
    SAVE TRANSACTION StkOrdTrn;
    UPDATE InvCtrl SET QtyInStk = QtyInStk - @OrderQty
        WHERE WhrhousID = @WhrhousID
          AND PartNmbr = @PartNmbr;
    COMMIT TRANSACTION InProc;
    RETURN 0;
END TRY

BEGIN CATCH
    SELECT @ErrorVar = @@error;
    IF (@ErrorVar = 547)
        BEGIN
            ROLLBACK TRANSACTION StkOrdTrn;
            COMMIT TRANSACTION InProc;
            RETURN (SELECT QtyInStk
                FROM InvCtrl
                WHERE WhrhousID = @WhrhousID
                  AND PartNmbr = @PartNmbr);
        END
END CATCH

```

执行下面的语句用于向 InvCtrl 中插入一条记录，然后执行 OrderStock 存储过程，准备销售 30 个商品，由于大于了库存数量，所以返回当前的库存数量 20。

```

INSERT INTO InvCtrl VALUES (1,2,20,2)

DECLARE @i AS int -- 定义变量保存 OrderStock 的返回值
EXEC @i = OrderStock 1,2,30 -- 准备销售 30，将引发 547 号错误
SELECT @i -- 显示返回值

```



第 17 章 并发访问控制

当多个用户试图访问同一数据时，需要使用一些并发访问控制机制，以防止产生负面影响。例如，如果一个进程正在执行下面的 SELECT 语句：

```
SELECT AVG(Discount)
FROM Customers;
```

而另一个应用程序正在执行下面的语句更新 Discount 列：

```
UPDATE Customers
SET Discount = 0.10
WHERE ShipCity = 'Portland' ;
```

这样，第 1 个进程可能会根据一部分 Portland 客户旧的折扣值和一些新的折扣值来计算平均值。像这种情况，可以通过锁定来防止这两个语句交错地检索和更新行所带来的无效结果。

17.1 并发影响和并发控制类型

在用户可以并发访问数据时，修改数据的用户会影响同时读取或修改相同数据的其他用户。如果数据存储系统没有并发控制，则用户可能会看到丢失更新、脏读等负面影响。所以，当多人试图同时修改数据库中的数据时，必须实施一个控制系统。

17.1.1 并发影响

1. 丢失更新

当两个或多个事务选择同一行，然后基于最初选定的值更新该行时，由于每个事务都不知道其他事务的存在。最后的更新将覆盖由其他事务所做的更新，这将导致其他事务所做修改的数据丢失。例如，两个编辑人员制作了同一文档的电子副本。然后每个编辑人员独立地更改其副本，并保存更改后的副本，这样，最后保存其更改副本的编辑人员覆盖另一个编辑人员所做的更改。如果在一个编辑人员完成并提交事务之前，另一个编辑人员不能访问同一文件，则可避免此问题。

2. 脏读

例如，一个编辑人员正在更改电子文档。在更改过程中，另一个编辑人员复制了该文档（该副

本包含到目前为止所做的全部更改)并将其分发给预期的用户。此后,第 1 个编辑人员认为目前所做的更改是错误的,于是删除了所做的编辑并保存了文档,而已分发给用户的文档仍旧包含已被第 1 个编辑人员取消的内容。这就是数据的脏读现象。如果在第 1 个编辑人员保存最终更改并提交事务之前,任何人都不能读取更改此文档,则可以避免此问题。

3. 不一致的分析(不可重复读)

当一个事务多次访问同一行,而每次会读取不同数据时,就会发生不一致的分析问题。不一致的分析与脏读类似,因为其他事务也是正在更改该事务正在读取的行。但是,在不一致的分析中,该事务读取的数据是由已进行了更改的事务提交的。此外,不一致的分析涉及多次(两次或更多)读取同一行,而且每次信息都被其他事务更改,因此我们称之为“不可重复读”。

4. 幻读

当对某行执行插入或删除操作,而该行属于某个事务正在读取的行的范围时,会发生幻读问题。由于其他事务的删除操作,事务第一次读取的行的范围显示有一行不再存在于第二次或后续读取内容中。同样,由于其他事务的插入操作,事务第二次或后续读取的内容显示有一行并不存在于第一次读取内容中。

17.1.2 并发控制

1. 悲观并发控制

如果用户执行的操作导致应用了某个锁,只有这个锁的所有者释放该锁,其他用户才能执行与该锁冲突的操作。这种方法之所以称为悲观并发控制,是因为它主要用于数据争用激烈的环境中,以及发生并发冲突时用锁保护数据的成本低于回滚事务的成本的环境中。

2. 乐观并发控制

在乐观并发控制中,用户读取数据时不锁定数据。当编辑完发送更新时,系统才检查在该用户读取数据后是否有其他用户更改了该数据。如果其他用户更改了数据,将产生一个错误。一般情况下,收到错误信息的用户将回滚事务,并重新读取新数据。这种方法之所以称为乐观并发控制,是因为它主要用于数据争用较少的环境中,以及回滚事务的成本偶尔高于读取数据时锁定数据的成本的环境中。

17.2

锁管理器的数据锁定

锁定是数据库引擎用来同步多个用户同时对同一个数据块访问的一种机制。在事务获取数据块当前状态的依赖关系(比如通过读取或修改数据)之前,它必须保护自己不受其他事务对同一数据进行修改的影响。事务通过请求锁定数据块来达到此目的。锁有多种模式,如共享或独占。锁模式

定义了事务对数据所拥有的依赖关系级别。如果某个事务已获得特定数据的锁，则其他事务不能获得会与该锁模式发生冲突的锁，直到这个锁被释放为止。

当事务修改某个数据块时，它将持有保护所做修改的锁直到事务结束。事务持有锁的时间长度，取决于事务隔离级别设置。一个事务持有的所有锁都在事务完成（无论是提交还是回滚）时释放。

但是，应用程序一般不直接请求锁。锁由数据库引擎的一个称为“锁管理器”的部件在内部管理。当数据库引擎实例处理 SQL 语句时，数据库引擎查询处理器会决定将要访问哪些资源。查询处理器根据访问类型和事务隔离级别设置来确定保护每一资源所需的锁的类型。然后，查询处理器将向锁管理器请求适当的锁。如果与其他事务所持有的锁不会发生冲突，锁管理器将授予该锁。

17.2.1 锁的粒度和层次结构

数据库引擎具有多粒度锁定，允许一个事务锁定不同类型的资源。为了尽量减少锁定的开销，数据库引擎自动将资源锁定在适合任务的级别。锁定在较小的粒度（例如行）可以提高并发度，但开销较高，因为如果锁定了许多行，则需要持有更多的锁。锁定在较大的粒度（例如表）会降低并发度，因为锁定整个表限制了其他事务对表中任意部分的访问。但其开销较低，因为需要维护的锁较少。

数据库引擎通常必须获取多粒度级别上的锁才能完整地保护资源。这种多粒度级别上的锁称为锁层次结构。表 17-1 列出了数据库引擎可以锁定的资源。

表 17-1 数据库引擎可以锁定的资源

资 源	说 明
RID	用于锁定堆中的单个行的行标识符
KEY	索引中用于保护可序列化事务中的键范围的行锁
PAGE	数据库中的 8 KB 页，例如数据页或索引页
EXTENT	一组连续的 8 页，例如数据页或索引页
HOB	堆或 B 树。保护索引或没有聚集索引的表中数据页堆的锁
TABLE	包括所有数据和索引的整个表
FILE	数据库文件
APPLICATION	应用程序专用的资源
METADATA	元数据锁
ALLOCATION_UNIT	分配单元
DATABASE	整个数据库

17.2.2 锁的模式

数据库引擎使用不同的锁模式锁定资源，这些锁模式确定了并发事务访问资源的方式。

1. 共享锁

共享锁（S 锁）用于不更改或不更新数据的读取操作，如 SELECT 语句。资源上存在共享锁时，任何其他事务都不能修改数据。读取操作一完成，就立即释放资源上的共享锁，除非将事务隔离级别设置为可重复读或更高级别，或者在事务持续时间内用锁定提示需要保留共享锁。

2. 更新锁

更新锁（U 锁）可以防止常见的死锁。在可重复读或可序列化事务中，在读取数据时，所有事务仅获得资源（页或行）的共享锁即可。但是，在修改数据时，要求将共享锁转换为排他锁（X 锁）。假设两个事务都获得了资源上的共享模式锁，然后试图同时更新数据，则一个事务尝试将锁转换为排他锁。共享模式到排他锁的转换必须等待一段时间，因为一个事务的排他锁与其他事务的共享模式锁不兼容。第 2 个事务此时也会试图获取排他锁。由于两个事务都要转换为排他锁，并且每个事务都等待另一个事务释放共享模式锁，因此发生死锁。

要避免这种潜在的死锁问题，可以使用更新锁。更新锁与共享锁兼容，但是与排他锁不兼容，如果进程开始了一个最终要修改数据的搜索操作，它会获取更新锁，直到找到要修改的数据。由于更新锁与共享锁兼容，所以不影响其他事务检索数据。因此，当两个事务同时寻找相同的数据资源时，则第 1 个到达的事务会获取到更新锁，然后第 2 个事务无法取得任何锁定，并且将等待第 1 个事务处理完成。由于第 1 个事务没有被阻塞，它可以将其更新锁转换为排他锁，并完成事务处理后释放锁，然后由第 2 个事务进行其他修改

3. 排他锁

排他锁（X 锁）用于数据修改操作，如 INSERT、UPDATE 或 DELETE。使用排他锁时，任何其他事务都无法修改数据，仅在使用 NOLOCK 提示或未提交读隔离级别时才会进行读取操作。

数据修改语句（如 INSERT、UPDATE 和 DELETE）包含有修改和读取操作。语句在执行修改操作之前，首先需要执行读取操作以获取数据。因此，数据修改语句通常请求共享锁和排他锁。例如，UPDATE 语句可能根据与一个表的联接修改另一个表中的行。在这种情况下，除了请求更新行上的排他锁之外，UPDATE 语句还将请求在联接表中要读取行上的共享锁。

4. 意向锁

数据库引擎使用意向锁来确保共享锁或排他锁能够放置在锁层次结构的低层资源上。之所以称之为意向锁，是因为这些锁是在进行较低级别锁前获得的，然后才会通知意向将锁放置在较低

级别上。

例如，在对某个表的页或行上请求共享锁之前，可以在表级请求共享意向锁。在表级设置意向锁可防止另一个事务随后在该表上获取排他锁。意向锁可以提高性能，因为数据库引擎仅在表级检查意向锁来确定事务是否可以安全地获取该表上的锁，而不需要检查表中的每行或每页上的锁以确定事务是否可以锁定整个表。

表 17-2 列出了意向锁的模式。

表 17-2 意向锁的模式

锁 模 式	说 明
意向共享 (IS)	是在保护对层次结构中某些（而并非所有）低层资源而请求或获取的共享锁
意向排他 (IX)	是在保护对层次结构中某些（而并非所有）低层资源而请求或获取的排他锁。IX 是 IS 的超集，它也保护针对低层级别资源请求的共享锁
意向排他共享 (SIX)	是在保护对层次结构中某些（而并非所有）低层资源而请求或获取的共享锁以及针对某些（而并非所有）低层资源而请求或获取的意向排他锁。顶层资源允许使用并发 IS 锁。例如，获取表上的 SIX 锁也将获取正在修改的页上的意向排他锁以及修改的行上的排他锁。虽然每个资源在一段时间内只能有一个 SIX 锁，以防止其他事务对资源进行更新，但是其他事务可以通过获取表级的 IS 锁来读取层次结构中的低层资源
意向更新 (IU)	是在保护层次结构中所有低层资源而请求或获取的更新锁。仅在页资源上使用 IU 锁。如果进行了更新操作，IU 锁将转换为 IX 锁
共享意向更新 (SIU)	S 锁和 IU 锁的组合，作为分别获取这些锁并且同时持有两种锁的结果。例如，事务执行带有 PAGLOCK 提示的查询，然后执行更新操作。带有 PAGLOCK 提示的查询将获取 S 锁，更新操作将获取 IU 锁
更新意向排他 (UIX)	U 锁和 IX 锁的组合，作为分别获取这些锁并且同时持有两种锁的结果

5. 架构锁

执行表的数据定义语言 (DDL) 操作（例如添加列或删除表）时使用架构修改锁 (Sch-M 锁)。在架构修改锁起作用期间，会防止对表的并发访问。这意味着在释放架构修改锁之前，该锁之外的所有操作都将被阻止。

当编译查询时，使用架构稳定性锁 (Sch-S 锁)。架构稳定性锁不阻塞任何事务锁，包括排他锁。因此在编译查询时，其他事务都能继续运行，但不能在表上执行 DDL 操作。

6. 大容量更新锁

当将数据大容量复制到表，且指定了 TABLOCK 提示或者使用 sp_tableoption 设置了 table lock on bulk 表选项时，将使用大容量更新锁 (BU 锁)。大容量更新锁允许多个线程将数据并发地大容量加载到同一表，同时防止其他不进行大容量加载数据的进程访问该表。

7. 键范围锁

在使用可序列化事务隔离级别时，会在整个事务执行过程中保持关键字范围的锁定，键范围锁

可防止其他事务插入其键值位于可序列化事务读取的键值范围内的新行。例如，可序列化事务发出了一个 SELECT 语句，以读取键值介于'AAA'与'CZZ'之间的所有行。从'AAA'到'CZZ'范围内的键值上的键范围锁可阻止其他事务插入带有该范围内的键值（如'ADG'、'BBD'或'CAL'）的行。键范围锁可防止幻读。通过保护行之间的键范围，它还可以防止对事务访问的记录集进行幻插入。

17.2.3 锁的兼容性

锁的兼容性控制多个事务能否同时获取同一资源上的锁。如果资源已被另一事务锁定，则只有请求锁的模式与现有锁的模式相兼容时，才会授予新的锁请求。如果请求锁的模式与现有锁的模式不兼容，则请求新锁的事务将等待释放现有锁或等待锁超时过期。例如，由于没有与排他锁兼容的锁模式，因此在释放排他锁之前，其他事务均无法获取该资源的任何类型的锁。另一种情况是，如果共享锁已应用到资源，则即使第 1 个事务尚未完成，其他事务也可以获取该项的共享锁或更新锁。但是，在释放共享锁之前，其他事务无法获取排他锁。

表 17-3 列出了最常见的锁模式的兼容性。

表 17-3 常见锁模式的兼容性

请求模式	IS	S	U	IX	SIX	X
意向共享 (IS)	是	是	是	是	是	否
共享 (S)	是	是	是	否	否	否
更新 (U)	是	是	否	否	否	否
意向排他 (IX)	是	否	否	是	否	否
意向排他共享 (SIX)	是	否	否	否	否	否
排他 (X)	否	否	否	否	否	否

17.2.4 锁升级

锁升级是将许多较细粒度的锁转换成数量更少的较粗粒度的锁的过程，这样可以减少系统开销，但却增加了并发争用的可能性。

当数据库引擎获取低级别的锁时，它还将在包含更低级别的对象上放置意向锁。如：当锁定行或索引键范围时，数据库引擎将在包含行或键的页上放置意向锁。当锁定页时，数据库引擎将在包含页的更高级别的对象上放置意向锁。

升级锁时，数据库引擎将尝试将表上的意向锁更改为对应的全锁，例如，将意向排他锁 (IX 锁) 更改为排他锁 (X 锁)，或将意向共享锁 (IS 锁) 更改为共享锁 (S 锁)。如果锁升级尝试成功并获取全表锁，将释放事务在堆或索引上所持有的所有堆或 B 树锁、页锁 (PAGE 锁)、键范围锁 (KEY 锁) 或行级锁 (RID 锁)。如果无法获取全锁，则不会发生锁升级，而数据库引擎将继续获

取行、键或页锁。

数据库引擎不会将行锁或键范围锁升级到页锁，而是将它们直接升级到表锁。同样，页锁始终升级到表锁。

1. 锁升级阈值

当单个 SQL 语句在单个表或索引上获取 5000 个锁时，或数据库引擎实例中的锁的数量超出了内存或配置阈值时，都会启动锁升级。

(1) SQL 语句的升级阈值。

当 SQL 语句在表或索引的单个引用上获取 5000 个锁时，将触发锁升级。该语句必须在同一个堆或索引上获取 5000 个锁才会发生锁升级。例如，如果该语句在一个索引上获取 3000 个锁，在同一表中的另一个索引上获取 3000 个锁，这种情况下不会触发锁升级。

只有触发升级时已经访问的表才会发生锁升级。假定单个 SELECT 语句是一个联接，按照以下顺序访问三个表：TableA、TableB 和 TableC。该语句在 TableA 的聚集索引中获取 3000 个行锁，在 TableB 的聚集索引中获取 5000 个行锁，但尚未访问 TableC。当数据库引擎检测到该语句已经在 TableB 中获取 5000 个行锁时，它将尝试升级会话在 TableB 中持有的所有锁。它还尝试升级会话在 TableA 中持有的所有锁，但因为 TableA 上锁的数量小于 5000，所以升级不会成功。但它不会尝试在 TableC 中进行锁升级，因为发生升级时尚未访问该表。

(2) 数据库引擎实例的升级阈值。

每当锁的数量大于锁升级的内存阈值时，每获取 1250 个新锁，数据库引擎就会定期为锁升级选择语句。

内存阈值取决于 locks 配置选项的设置（可以使用 sp_configure 设置）：

- 如果 locks 选项设置为 0（默认值），当锁对象使用的内存是数据库引擎使用的内存的 40% 时，将达到锁升级阈值。此阈值是动态的，因为数据库引擎动态获取并释放内存以适应各种工作负荷。

- 如果 locks 选项设置为非 0 值，则锁升级阈值是 locks 选项的值的 40%（或者更低，如果存在内存不足的压力）。

2. 减少锁定和升级

在大多数情况下，使用数据库引擎默认的锁定和锁升级设置，性能最佳。如果数据库引擎实例生成大量锁并且频繁进行锁升级，可以通过下列方式减少锁定：

- 对于读取操作，使用不会生成共享锁的隔离级别。包括：当 READ_COMMITTED_SNAPSHOT 数据库选项为 ON 时，使用 READ COMMITTED 隔离级别；使

用 SNAPSHOT 隔离级别；使用 READ UNCOMMITTED 隔离级别。此隔离级别只能用于能对脏读进行操作的系统。

● 使用 PAGLOCK 或 TABLOCK 表提示，使数据库引擎使用页、堆或索引锁而不是行锁。但是，使用此选项增加了用户阻止其他用户尝试访问相同数据的问题，对于并发用户较多的系统，不应使用此选项。

此外，也可以使用跟踪标志 1211 和 1224 来禁用所有或某些锁升级。其中，1211 禁用所有锁升级。在使用此跟踪标志的情况下会生成过多的锁数目，降低数据库引擎的性能，或因为内存不足而导致 1204 错误（无法分配锁资源）。1224 将禁用各个语句的锁升级，与 locks 选项设置为 0 时的行为相同，如果锁对象使用的内存量超出数据库引擎使用的内存的 40%，则数据库引擎只将行锁或页锁升级到表锁。

如果同时设置了跟踪标志 1211 和 1224，则 1224 优先于 1211。

可以使用 DBCC TRACEON 打开指定的跟踪标志，使用 DBCC DBREPAIR 关闭指定的跟踪标志。例如，下面的语句将跟踪标志 1211 设置为开，1224 设置为关闭。

```
DBCC TRACEON (1211)
DBCC DBREPAIR(1224)
```

17.3 自定义锁定

使用数据库引擎的应用程序可以更改默认锁定行为。

17.3.1 自定义锁的超时时间

在默认情况下，没有强制的超时限制。可以使用 LOCK_TIMEOUT 设置指定语句等待阻塞资源的最长时间。如果某个语句等待的时间超过了 LOCK_TIMEOUT 设置，则被阻塞的语句自动取消，并会有错误消息 1222（Lock request time-out period exceeded）返回给应用程序。但是，不会回滚或取消任何包含语句的事务。因此，应用程序必须具有可以捕获错误消息 1222 的错误处理程序。如果应用程序不能捕获错误，则会继续执行事务中后面的语句，由于这些语句可能依赖于前面被阻塞的语句，因此会出现错误。

在错误处理程序捕获到错误消息 1222 时，应用程序应当采取补救措施，如自动重新提交被阻塞的语句或回滚整个事务。

要确定当前 LOCK_TIMEOUT 设置，可以执行 @@LOCK_TIMEOUT 函数：

```
SELECT @@lock_timeout;
```

要指定语句等待锁超时的毫秒数，可以使用 SET LOCK_TIMEOUT 语句。例如，下面的语句将锁超时期限设置为 1800 毫秒：


```
SET LOCK_TIMEOUT 1800
```

在连接开始时，超时设置的默认值为-1。在更改设置后，新设置在其余的连接时间里一直有效。

CREATE DATABASE、ALTER DATABASE 和 DROP DATABASE 语句不使用 SET LOCK_TIMEOUT 设置。

17.3.2 使用表级锁提示

可以在 SELECT、INSERT、UPDATE 及 DELETE 语句中为单个表指定锁定，方法是在表名称后面通过 WITH 关键字加上锁提示。例如，下面的 SELECT 语句不会发生更新操作的冲突：

```
SELECT AVG(Discount)
FROM Customers WITH (TABLOCK)
```

TABLOCK 提示指定在 SELECT 语句执行过程中保持对整个表的共享锁定。如果在 SELECT 语句开始执行时不能获得对 Customers 表的共享锁定，则这条语句就会执行失败。

此外，UPDATE 语句可以使用带有 TABLELOCKX 表提示的 FROM 子句来保持一个独占锁定，这可以防止其他进程对表进行任何类型的访问。如：

```
UPDATE Customers
SET Discount = 0.10
FROM Customers WITH (TABLELOCKX)
WHERE ShipCity = 'Portland'
```

从前面的介绍可以知道，在缺省情况下，数据库引擎使用“锁管理器”来自动为每一个访问确定一个适当的锁定粒度，从而提供最佳的访问效率。当需要对对象所获得锁类型进行更精细控制时，就可以使用表级锁提示。这些锁提示覆盖会话的当前事务隔离级别。表 17-4 列出了常用表提示的锁定粒度。

表 17-4 表提示的锁定粒度

锁 定 粒 度	锁 定 范 围
NOLOCK	不锁定任何资源（仅允许在 SELECT 语句中使用）
ROWLOCK	行
PAGELOCK	页
TABLOCK	表（共享锁定）
TABLOCKX	表（独占锁定）

还有两种类型的表提示适用于行锁定类型。UPDLOCK 表提示指定在读取的每一行上设置一个更新锁定，直到整个语句或事务执行完毕，才释放这个锁定。更新锁定允许其他事务获得同一行的共享锁定，但是不允许它们更新行（或获得更新锁）。XLOCK 指定采用排他锁并保持到事务完成。如果同时指定了 ROWLOCK、PAGELOCK 或 TABLOCK，则排他锁将应用于相应的粒度级别。

当在表上放置了共享锁定或独占锁定后,应当尽量使锁定的时间最短,因为锁定本身可能阻塞其他进程对表的正常访问。缺省情况下,表锁定仅保持在定义该锁定的语句的运行时间段内。如果用户需要一直保持表锁定,直到整个事务执行完毕,则可以添加 HOLDLOCK 关键字:

```
SELECT AVG(Discount)
FROM Customers WITH (TABLOCK, HOLDLOCK)
```

此外,也可以在表名称后面指定隔离级别提示,包括 READUNCOMMITTED、READCOMMITTED、REPEATABLE READ 和 SERIALIZABLE。例如:

```
SELECT AVG(Discount)
FROM Customers WITH (SERIALIZABLE)
```

有关“隔离级别”的详细信息可参考后面的介绍。

17.4 使用事务隔离级别

前面介绍了默认锁定行为和如何使用表提示进行显式锁定的方法,来控制并发访问冲突问题。此外,SQL Server 还提供了一种事务隔离机制来控制并发访问。在第 16 章我们介绍了通过事务保持数据完整性的方法,事务隔离机制也为简化锁定操作、控制事务间的并发冲突提供了解决方法。表 17-5 列出了 SQL-99 标准定义的 4 种隔离级别。其中,READ UNCOMMITTED 是最低的隔离级别,它允许最大范围的并发访问。也就是说,一个事务可以与其他事务最大限度地交互工作。但这也增加了用户可能遇到的并发副作用(例如脏读或丢失更新)的数量。SERIALIZABLE 是最高的隔离级别,它使得在这个级别上运行的事务,在整个执行过程中会保持关键字范围的锁定,事务之间完全隔离,不会出现并发访问问题。

表 17-5 SQL-99 标准定义的隔离级别

隔离级别	作用
READ UNCOMMITTED	隔离事务的最低级别。不进行锁定,也不会对独占锁定特殊处理,只能保证不读取物理上损坏的数据。允许脏读
READ COMMITTED	数据库引擎的默认级别。要求对读取的每一行实行共享锁定。如果行没有被更新,将会释放锁定
REPEATABLE READ	要求对读取的每一行实行共享锁定。并在整个事务执行过程中保持锁定
SERIALIZABLE	隔离事务的最高级别。在整个事务执行过程中保持关键字范围的锁定,事务之间完全隔离

此外,数据库引擎还支持使用行版本控制的两个事务隔离级别,分别是 READ COMMITTED 隔离和 SNAPSHOT ISOLATION (快照隔离) 隔离。READ COMMITTED 是默认的隔离级别。可以使用如下的语句,为一个连接设置不同的隔离级别。

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
```

也可以在特定的 SELECT、UPDATE、INSERT 或 DELETE 语句中的表名称的后面添加一个隔离提示来指定一个隔离级别。例如:

```
UPDATE Customers
SET Discount = 0.10
FROM Customers WITH (REPEATABLE READ)
WHERE ShipCity = 'Portland'
```

隔离提示关键字包括：READ UNCOMMITTED、READ COMMITTED、REPEATABLE READ 和 SERIALIZABLE。

在选择使用隔离级别时要非常谨慎。级别过低，会带来错误，导致不一致的结果。级别过高，则会影响性能。在一般情况下，不要使用 READ UNCOMMITTED，除非用户已经在整个表上放置显式的锁定。使用 READ UNCOMMITTED 时，当前事务可以读取其他事务已经修改，但是没有提交的数据，这也就是所谓的脏读。当一个事务在 READ UNCOMMITTED 隔离级别上检索数据时，其他事务也在并发地修改数据，SQL Server 可能会返回一些不合逻辑的 I/O 错误。

默认的 READ COMMITTED 隔离级别，不允许事务读取没有提交的数据。这样就防止了上面的脏读问题。一个运行在 READ COMMITTED 级别上的 SQL 语句，在读取每一行时，会获取一个共享锁定。如果需要更新行时，这个锁定会转换为一个独占锁定。否则，当读取下一行数据时，这个锁定就会被释放。一旦共享锁定被释放，其他的事务就可以更新这一行数据了。这意味着重复执行同一个 SELECT 语句时，可能会得到不同的结果。例如：

```
-- 在 READ COMMITTED 隔离级别下开始事务处理

SELECT AVG(Discount)
FROM Customers -- 在当前事务中第 1 次执行

-- 假设此时另一个事务修改了一个客户的 Discount 的值

SELECT AVG(Discount)
FROM Customers -- 在当前事务中第 2 次执行

COMMIT -- 提交事务
```

上面的这两个 SELECT 语句可能会返回不同的平均折扣率。在实际情况中，可能很少会需要执行同样的 SELECT 语句两次。但是，复杂的事务可能会在同一事务内重复读取同一行的数据，并且两行的数据必须相同。对于这类型的事务，用户可以使用 REPEATABLE READ 级别。使用 REPEATABLE READ 级别时，共享的行锁定仅在事务执行完毕后才释放。在整个事务的执行过程中一直有效，这样就防止了其他事务更新行数据。

REPEATABLE READ 阻塞了以前读取过行的更新操作，但是不妨碍新行的插入操作。例如：

```
-- 在 REPEATABLE READ 隔离级别下开始事务处理

SELECT AVG(Discount)
FROM Customers -- 在当前事务中第 1 次执行

-- 假设此时另一个事务插入了一个新的客户行

SELECT AVG(Discount)
FROM Customers -- 在当前事务中第 2 次执行

COMMIT -- 提交事务
```

在这个示例中，由于在第 2 次执行 SELECT 语句时，另一个事务插入了一个新行，该行会被包含在第 2 次计算平均折扣率的行集范围内（这也就是幻读现象）。所以这两个 SELECT 语句可能会返回不同的平均折扣率。

为了解决这些问题，可以使用最高级别的 **SERIALIZABLE**。该隔离级别可以防止对事务查询范围内的所有行的更新操作。

以下是选择事务隔离级别时的一些建议。

- 如果用户不在乎结果的精确程度，对于 SELECT 语句和只读游标，**READ UNCOMMITTED** 是一个很好的选择。事实上，在大多数情况下，这都不是一个很好的选择，因为大部分用户都不希望有错误的结果。

- 如果用户在同一事务中不需要重复读取数据时，可以选择 **READ COMMITTED**。

- 当用户需要在同一事务中重复读取数据，且数据需要保持不变时，则可以选择 **REPEATABLE READ**。某些情况下，用户可以将前面读取的数据存储起来，这样就可以不使用 **REPEATABLE READ**。不过，在有些情况下，这个级别还是很有必要的。例如，用户需要在整个事务的执行过程中保持读取过的数据不会发生变化，即使用户不再需要读取这些数据也是如此。这种要求在实际应用中很可能会出现。

- **SERIALIZABLE** 与 **REPEATABLE READ** 所适用的情况相同，只是增加了一个约束条件，即在整个事务执行过程中不会增加新数据。

17.5 使用行版本的事务隔离级别

在前面介绍的由 SQL-99 标准定义的 4 种隔离级别，是基于锁定来解决并发性控制的。设置级别过低，会造成脏读等副作用；设置级别过高，则影响程序的并发性操作。从 SQL Server 2005 开始，提供了依赖于行版本控制的事务隔离级别，它通过将所使用的行复制到 **tempdb** 数据库中的方法，来避免使用锁定，从而提高并发性能。

依赖于行版本控制的增强包括：一是在现有 **READ COMMITTED** 事务隔离级别的基础上提供了一种新的实现方式，用于提供使用行版本控制的语句级快照；二是新增了一个 **SNAPSHOT ISOLATION**（快照隔离）事务隔离级别，用于提供使用行版本控制的事务级快照。

17.5.1 快照隔离和行版本控制的工作原理

启用快照隔离级别时，每次更新行时，数据库引擎在 **tempdb** 数据库中存储原始行的副本，并为该行添加事务序列号。唯一的事务序列号标识每个事务，并且为每个行版本记录这些唯一的编号。事务使用序列号在当前事务序列号之前的最新行版本，并忽略在事务开始之后创建的更新的行版本。

所谓“快照”，是指事务中的所有查询根据事务开始那一刻的数据库状态，能够在当前事务的整个执行过程中看到数据库的相同版本（即无论执行多少次查询）。在快照事务中不会对基础数据行或数据页获取锁，这样就可以执行其他事务，而不会被以前未完成的事务所阻止。修改数据的事务不会阻止读取数据的事务，读取数据的事务不会阻止写入数据的事务，就好像通常情况下使用默认的 READ COMMITTED 隔离级别一样。这种无阻止的行为也大大降低了复杂事务出现死锁的可能性。

快照隔离使用开放式并发控制，不赋予可能阻止其他事务更新行的任何锁。在当前事务使用 tempdb 中的行版本的同时，其他事务仍旧可以对基础表进行检索和更新。但是，如果快照事务尝试提交对事务开始后被其他事务更改过的数据的进行更新，事务将回滚并将引发错误。

下面是发生事务时详细的事件序列过程。

- (1) 新的事务启动，并为该事务分配一个事务序列号（该序列号是依次递增的）。
- (2) 数据库引擎在事务中读取某行，并从 tempdb 中检索与事务序列号最接近并且小于事务序列号的行版本。
- (3) 数据库引擎检查事务编号是否没有在未提交事务的事务编号列表中，这些未提交事务是在快照事务开始时进入活动状态的。
- (4) 事务从 tempdb 中读取自事务开始以来最新的行版本。事务不会看到事务开始后插入的新行，因为这些序列号值将大于事务序列号的值。
- (5) 当前事务将看到事务开始后删除的行，因为 tempdb 中的行版本具有更低的序列号值。

17.5.2 使用基于行版本控制的隔离级别

将 READ_COMMITTED_SNAPSHOT 数据库选项设置为 ON 时，READ COMMITTED 隔离使用行版本控制提供语句级别的读取一致性。将 READ_COMMITTED_SNAPSHOT 数据库选项设置为 OFF（默认设置）时，READ COMMITTED 隔离的行为与在 SQL Server 的早期版本中相同。

SNAPSHOT ISOLATION 隔离级别使用行版本控制来提供事务级别的读取一致性。读取其他事务修改的行时，读取操作将检索启动事务时存在的行的版本。将 ALLOW_SNAPSHOT_ISOLATION 数据库选项设置为 ON 时，将启用 SNAPSHOT ISOLATION。默认情况下，用户数据库的此选项设置为 OFF。

下面将使用两个示例来说明 READ_COMMITTED 语句级快照和 SNAPSHOT ISOLATION 事务级快照在处理并发操作时的行为差异。

1. 使用快照隔离

在下面的示例中，在快照隔离下运行的事务将读取数据，然后由另一事务修改此数据。快照事务不阻塞由其他事务执行的更新操作，它忽略数据的修改继续从版本化的行读取数据。但是，当快照事务尝试修改已由其他事务修改的数据时，快照事务将生成错误并终止。

在会话 1 上执行下面的语句：

```
USE AdventureWorks;
GO

-- 在数据库上允许快照隔离。
ALTER DATABASE AdventureWorks
    SET ALLOW_SNAPSHOT_ISOLATION ON;
GO

-- 开始一个快照事务
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
GO

BEGIN TRANSACTION;
-- 该 SELECT 语句，返回的 VacationHours 列的值为 48
SELECT EmployeeID, VacationHours
    FROM HumanResources.Employee
    WHERE EmployeeID = 4;
```

在会话 2 上执行下面的语句：

```
USE AdventureWorks;
GO

-- 开始一个事务
BEGIN TRANSACTION;
-- 将 EmployeeID 为 4 雇员的 VacationHours 值减去 8，由于在快照隔离下，不需要共享锁定。
UPDATE HumanResources.Employee
    SET VacationHours = VacationHours - 8
    WHERE EmployeeID = 4;

-- 验证 VacationHours 的值，此时应当为 40
SELECT VacationHours
    FROM HumanResources.Employee
    WHERE EmployeeID = 4;
```

在会话 1 上再执行下面的语句。由于使用了快照隔离，下面的 SELECT 语句将从版本行中读取数据，因此 VacationHours 的值仍旧为 48，而不是被会话 2 所更新后的 40。

```
SELECT EmployeeID, VacationHours
    FROM HumanResources.Employee
    WHERE EmployeeID = 4;
```

在会话 2 上执行下面的语句，提交事务，进行数据修改。

```
COMMIT TRANSACTION;
```

在会话 1 上执行下面的语句：

```
-- 即使会话 2 已经提交了修改，但是 VacationHours 的值仍旧为 48。
```

```

SELECT EmployeeID, VacationHours
FROM HumanResources.Employee
WHERE EmployeeID = 4;

-- 由于数据已经被当前快照隔离之外的事务修改，所以下面的更新将失败
-- 并生成 3960 错误，事务将结束
UPDATE HumanResources.Employee
SET SickLeaveHours = SickLeaveHours - 8
WHERE EmployeeID = 4;

-- 撤销会话 1 中的修改，但是，不能撤销会话 2 中所做的修改
ROLLBACK TRANSACTION

```

2. 使用通过行版本控制的 READ COMMITTED

在此示例中，使用行版本控制的 READ COMMITTED 隔离级别事务与其他事务并发运行。READ COMMITTED 事务的行为与快照事务的行为有所不同。与快照事务相同的是，即使其他事务修改了数据，READ COMMITTED 事务也将读取版本化的行。然而，与快照事务不同的是，READ COMMITTED 将执行下列操作：

- 在其他事务提交数据更改后，READ COMMITTED 将读取修改后的数据。
- 能够更新由其他事务修改的数据，而快照事务不能。

例如，在会话 1 上执行下面的语句：

```

USE AdventureWorks;
GO

-- 在数据库上允许 READ_COMMITTED_SNAPSHOT
-- 下面的语句要执行成功，必须仅有该会话连接到 AdventureWorks 数据库
ALTER DATABASE AdventureWorks
SET READ_COMMITTED_SNAPSHOT ON;
GO

-- 开始一个 READ-COMMITTED 事务
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
GO

BEGIN TRANSACTION;
-- 该 SELECT 语句将返回 VacationHours 的值为 48
SELECT EmployeeID, VacationHours
FROM HumanResources.Employee
WHERE EmployeeID = 4;

```

在会话 2 上执行下面的语句：

```

USE AdventureWorks;
GO

-- 开始一个事务
BEGIN TRANSACTION;
-- 更新 VacationHours 的值。由于在 READ-COMMITTED 快照隔离级别下
-- 不要求共享锁，因此更新不会被会话 1 阻塞
UPDATE HumanResources.Employee
SET VacationHours = VacationHours - 8
WHERE EmployeeID = 4;

-- 验证 VacationHours 的值，此时应当为 40。

```

```
SELECT VacationHours
FROM HumanResources.Employee
WHERE EmployeeID = 4;
```

在会话 1 上再执行下面的语句。由于会话 2 还没有将修改进行提交，所以下面的语句仍旧从版本行中读取数据，得到 VacationHours 的值仍旧为 48。

```
SELECT EmployeeID, VacationHours
FROM HumanResources.Employee
WHERE EmployeeID = 4;
```

在会话 2 上执行下面的语句，提交修改：

```
COMMIT TRANSACTION;
```

在会话 1 上执行下面的语句：

```
-- 由于会话 2 已经提交修改，所以下面的 SELECT 语句检索到的 VacationHours 的值
-- 为 40。这是与快照事务所不同的地方，快照事务即使别的事务已经提交了更新，
-- 它仍旧从版本行中读取值
SELECT EmployeeID, VacationHours
FROM HumanResources.Employee
WHERE EmployeeID = 4;

-- 下面的更新语句在快照事务中会发生错误，但是在 READ-COMMITTED 快照中
-- 会更新成功。
UPDATE HumanResources.Employee
SET SickLeaveHours = SickLeaveHours - 8
WHERE EmployeeID = 4;

-- 撤销会话 1 中的修改，但不能撤销会话 2 中的修改
ROLLBACK TRANSACTION;
```

17.6 处理死锁

在两个或多个任务中，如果每个任务锁定了其他任务试图锁定的资源，此时会造成这些任务永久阻塞，从而出现死锁。例如：

- 事务 A 获取了行 1 的共享锁。
- 事务 B 获取了行 2 的共享锁。
- 现在，事务 A 请求行 2 的排他锁，但在事务 B 完成并释放其对行 2 持有的共享锁之前被阻塞。
- 现在，事务 B 请求行 1 的排他锁，但在事务 A 完成并释放其对行 1 持有的共享锁之前被阻塞。

事务 A 必须在事务 B 完成之后才能完成，但事务 B 被事务 A 阻塞。这种情况也称为循环依赖关系：事务 A 依赖于事务 B，而事务 B 又依赖于事务 A，从而形成了一个循环。

除非某个外部进程断开死锁，否则死锁中的两个事务都将无限期等待下去。数据库引擎死锁监视器定期检查陷入死锁的任务。如果检测死锁，将选择其中一个任务作为牺牲品，然后终止其事务。

并提示错误。这样，其他任务就可以完成其事务，从而解除死锁。对于事务以错误终止的应用程序，它还可以重试该事务，但通常要等到与它一起陷入死锁的其他事务完成后执行。

17.6.1 防止死锁的方法

1. 按同一顺序访问对象

如果所有并发事务按同一顺序访问对象，则发生死锁的可能性会降低。例如，如果两个并发事务先获取 `Supplier` 表上的锁，然后获取 `Part` 表上的锁。在其中一个事务完成之前，另一个事务将在 `Supplier` 表上被阻塞。当第 1 个事务提交或回滚之后，第 2 个事务将继续执行，这样就不会发生死锁。如果使用存储过程进行数据修改，则可以使对象的访问顺序标准化。

2. 避免事务中的用户交互

避免编写包含用户交互的事务，因为没有用户干预的批处理的运行速度远快于用户必须手动响应查询时的速度。例如，如果事务正在等待用户输入，而用户去吃午餐了，那么用户就耽误了事务的完成。这将降低系统的吞吐量，因为事务持有的任何锁只有在事务提交或回滚后才会释放。即使不出现死锁的情况，在占用资源的事务完成之前，访问同一资源的其他事务也会被阻塞。

3. 保持事务简短并处于一个批处理中

在同一数据库中，并发执行多个需要长时间运行的事务时通常会发生死锁。事务的运行时间越长，它持有排他锁或更新锁的时间也就越长，从而会阻塞其他活动并可能导致死锁。

4. 使用较低的隔离级别

确定事务是否能在较低的隔离级别上运行。实现已提交读允许事务读取另一个事务已读取（未修改）的数据，而不必等待第 1 个事务完成。使用较低的隔离级别（例如已提交读）比使用较高的隔离级别（例如可序列化）持有共享锁的时间更短。这样就减少了锁争用。

5. 使用基于行版本控制的隔离级别

如果将 `READ_COMMITTED_SNAPSHOT` 数据库选项设置为 `ON`，则在已提交读隔离级别下运行的事务在读操作期间将使用行版本控制而不是共享锁。

17.6.2 使用 TRY...CATCH 处理死锁

在 `TRY...CATCH` 构造的 `CATCH` 块可以捕获 1205 死锁错误，发生错误后，可以通过回滚事务来解除锁定。下面的语句创建了用于说明死锁状态的表和用于打印错误信息的存储过程。

```
USE AdventureWorks;  
GO
```

```

-- 验证表是否已经存在
IF OBJECT_ID (N'my_sales',N'U') IS NOT NULL
    DROP TABLE my_sales;
GO

-- 创建表并插入数据
CREATE TABLE my_sales
(
    Itemid      INT PRIMARY KEY,
    Sales       INT not null
);
GO

INSERT my_sales (itemid, sales) VALUES (1, 1);
INSERT my_sales (itemid, sales) VALUES (2, 1);
GO

-- 验证存储过程是否已经存在
IF OBJECT_ID (N'usp_MyErrorLog',N'P') IS NOT NULL
    DROP PROCEDURE usp_MyErrorLog;
GO

-- 创建存储过程，用于输出错误消息
CREATE PROCEDURE usp_MyErrorLog
AS
    PRINT
        N'错误 ' + CONVERT(VARCHAR(50), ERROR_NUMBER()) +
        N', 严重级别 ' + CONVERT(VARCHAR(5), ERROR_SEVERITY()) +
        N', 状态 ' + CONVERT(VARCHAR(5), ERROR_STATE()) +
        N', 行 ' + CONVERT(VARCHAR(5), ERROR_LINE());
    PRINT
        ERROR_MESSAGE();

```

下面的会话 1 和会话 2 代码脚本在两个单独的 SQL Server Management Studio 连接下同时运行。两个会话都尝试更新表中的相同行。在第一次尝试过程中，其中一个会话将成功完成更新操作，而另一个会话将被选择为死锁牺牲品。死锁牺牲品错误将使执行跳至 CATCH 块，事务将进入无法提交状态。在 CATCH 块中，死锁牺牲品会回滚事务并重试更新此表，直到更新成功或达到了重试限制。

```

会话 1
USE AdventureWorks;
GO

-- 定义并设置变量，指定尝试提交更新的次数
DECLARE @retry INT;
SET @retry = 5;

-- 如果被作为死锁牺牲品，保持尝试更新
WHILE (@retry > 0)
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION;

        UPDATE my_sales
        SET sales = sales + 1
        WHERE itemid = 1;

        -- 延时等待，此时 itemid 为 1 和 2 的行

```

```

会话 2
USE AdventureWorks;
GO

-- 定义并设置变量，指定尝试提交更新的次数
DECLARE @retry INT;
SET @retry = 5;

-- 如果被作为死锁牺牲品，保持尝试更新
WHILE (@retry > 0)
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION;

        UPDATE my_sales
        SET sales = sales + 1
        WHERE itemid = 2;

        -- 延时等待，此时 itemid 为 1 和 2 的行

```



```

-- 在没有提交前，都无法释放锁
WAITFOR DELAY '00:00:13';

UPDATE my_sales
SET sales = sales + 1
WHERE itemid = 2;

SET @retry = 0;

COMMIT TRANSACTION;
END TRY
BEGIN CATCH
-- 检测错误编号，如果是死锁牺牲品，
-- 则减少重新尝试计数。如果是其他
-- 错误，则退出 WHILE 循环
IF (ERROR_NUMBER() = 1205)
    SET @retry = @retry - 1;
ELSE
    SET @retry = -1;

-- 输出错误消息
EXECUTE usp_MyErrorLog;

-- 会话中包含无法提交的事务
-- XACT_STATE 将返回 -1
IF XACT_STATE() <> 0
    ROLLBACK TRANSACTION;
END CATCH;
END; -- 结束 WHILE 循环

```

```

-- 在没有提交前，都无法释放锁
WAITFOR DELAY '00:00:07';

UPDATE my_sales
SET sales = sales + 1
WHERE itemid = 1;

SET @retry = 0;

COMMIT TRANSACTION;
END TRY
BEGIN CATCH
-- 检测错误编号，如果是死锁牺牲品，
-- 则减少重新尝试计数。如果是其他
-- 错误，则退出 WHILE 循环
IF (ERROR_NUMBER() = 1205)
    SET @retry = @retry - 1;
ELSE
    SET @retry = -1;

-- 输出错误消息
EXECUTE usp_MyErrorLog;

-- 会话中包含无法提交的事务
-- XACT_STATE 将返回 -1
IF XACT_STATE() <> 0
    ROLLBACK TRANSACTION;
END CATCH;
END; -- 结束 WHILE 循环

```

下面是会话 1 中返回的消息，表示两行都已经被更新。

(1 行受影响)

(1 行受影响)

下面是会话 2 中返回的消息，会话 2 被作为了死锁牺牲品。

(1 行受影响) -- 由于死锁，会话 2 事务中只成功更新 1 行，这时会发生回滚操作
 错误 1205，严重级别 13，状态 51，行 18 -- 由存储过程 usp_MyErrorLog 输出的错误消息
 事务(进程 ID 52)与另一个进程被死锁在 锁 资源上，并且已被选作死锁牺牲品。请重新运行该事务。 -- SQL Server 的提示

(1 行受影响) -- 本行和下行消息是重新尝试更新后得到的提示消息

(1 行受影响)



第 18 章 查询的优化与执行

我们知道，像 SELECT、INSERT 等这些查询语句都是非程序性的，它不规定数据库服务器检索请求数据的确切步骤。这意味着数据库服务器必须分析语句，以决定提取所请求数据的最有效方法，并根据分析结果生成“查询执行计划”（有时也称为“查询计划”或直接称为“计划”），这个过程就是“查询优化”。也就是说，数据库服务器最终执行的是“查询计划”，而不是直接的查询语句。这样做的目的，就是在考虑服务器负荷等综合性因素的前提下，尽可能地减少查询成本的付出，并尽快地将查询结果返回到客户端。

这里需要注意的是，“优化”和“执行”是查询语句处理的两个不同阶段。对于已经生成的“查询计划”，会在缓存中被保留。如果客户提交的查询符合缓存中的某个“查询计划”，将会跳过“优化”阶段，直接重用该“查询计划”。

18.1 查询的优化

进行查询优化的组件被称为“查询优化器”。优化器根据输入的查询语句、数据库方案（表和索引的定义）以及数据库统计信息生成多个“查询计划”，并通过特定的算法从中选择出最终被数据引擎使用的计划。虽然优化器在分析查询和选择计划时需要耗费一些资源，但当查询优化器选择了有效的计划时，这一开销将节省数倍。尤其是当计划被多次重复执行时，进行查询优化的意义更是不言而喻。

18.1.1 查询计划定义的内容

查询语句的非程序性特点，可以使数据查询设计人员将注意力集中到查询结果的正确性上，而将查询性能的问题尽可能地交给“查询优化器”去处理。例如，一条 SELECT 语句仅定义了如下几方面的内容：

- 结果集的格式。通常在选择列表中指定，像 ORDER BY、GROUP BY 等子句也会影响结果集的最终格式。
- 包含源数据的表。这在 FROM 子句中指定。
- 表之间的逻辑关系。这在联接说明中定义，联接说明可出现在 FROM 子句后的 WHERE 子句或 ON 子句中。例如，下面的两条语句都是按 T1.列 1 = T2.列 1 关系将 T1 和 T2 表进行内部

联接。

```
SELECT *
FROM T1 INNER JOIN T2
ON T1.列1 = T2.列1;
```

-- 下面则是上面语句的 ANSI SQL:1989 书写方式

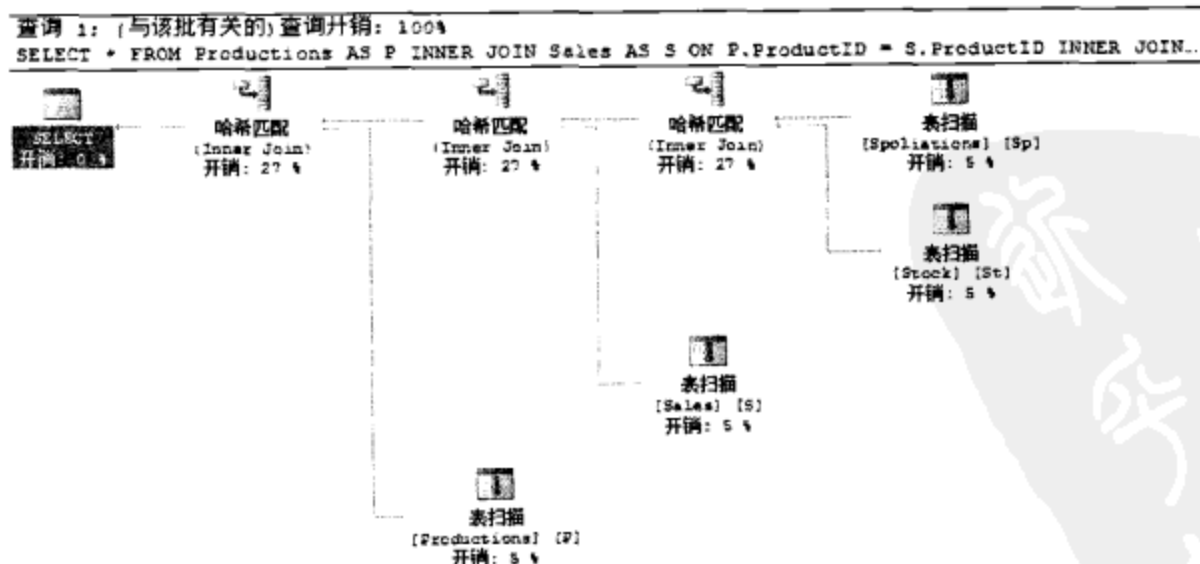
```
SELECT *
FROM T1, T2
WHERE T1.c1 = T2.c1;
```

- 源表中的行所必须达到的条件。这些条件在 WHERE 和 HAVING 子句中指定。

查询优化器通过对上述语句类似内容的分析，按“批”生成一个查询计划。我们知道，在 SQL Server Management Studio 中，GO 命令可以把多条语句作为一个“批”提交给数据库引擎进行处理。但是，查询优化器也不是对“批”中所有语句都进行优化，它仅对涉及表访问并有可能生成多个查询计划的语句进行优化。因此，查询执行计划中定义的内容也就不外乎两个方面：

- 访问源表的顺序。一条语句中如果包含对多个表的访问，按不同的序列访问基表，在某些情况下同样可以返回正确的结果集。具体采用什么样的顺序，主要看析取过程中那种方式能够尽快地排除多余的不匹配行。例如，在多表联接中完全是内部联接的情况下，表的联接顺序不影响查询结果。因此，查询优化器会根据对查询计划的评估，选择优先执行的联接。例如，下面的语句逻辑上应当按书写顺序进行联接，而实际的执行计划却是首先将 Spoliations 与 Stock 进行联接，然后将联接结果与 Saels 联接，联接结果最后再与 Productions 联接，如图 18-1 所示。

```
SELECT *
FROM Productions AS P
INNER JOIN Sales AS S
ON P.ProductID = S.ProductID
INNER JOIN Spoliations AS Sp
ON P.ProductID = Sp.ProductID
INNER JOIN Stock AS St
ON P.ProductID = St.ProductID;
```




- 从每个表析取数据的方法。访问每个表中的数据一般也有不同的方法。如果只需要有特

定键值的几行，数据库服务器可以使用索引。如果需要表中的所有行，数据库服务器则可以忽略索引并执行表扫描。如果需要表中的所有行，而有一个索引的键列在 ORDER BY 中，则执行索引扫描而非表扫描可能会省去对结果集的单独排序。如果表很小，则对该表的几乎所有访问来说，表扫描可能都是最有效的方法。

18.1.2 生成查询计划

当查询语句被送达数据库引擎后，查询优化器首先检查缓存中是否已经存在该语句的执行计划。如果存在，则直接使用该计划，从而节省因优化带来的额外开销。如果不存在，则按照查询语句分析、生成查询树、确定析取数据的方法生成查询计划。最终被优化后用于语句执行的查询计划，称为执行计划。检查现有执行计划，SQL Server 是通过一种很高效的算法来进行的，可以查找任何特定查询语句的现有执行计划。在大多数系统中，通过扫描后重新使用现有计划比直接优化查询语句要节省资源。

在前面我们介绍了查询计划包含的内容，那查询计划究竟是一个什么模样呢？查询计划实际上是一种数据结构，它准确指出了其所包含的每一条命令将影响哪个表，以及从表中析取数据的方法。在 SQL Server Management Studio 中选择  按钮后，可以以图形方式来展示语句的执行计划，如图 18-1 所示就是一个执行计划。为了更清楚地说明查询计划的生成步骤，这里给出了从视图 HumanResources.vEmployee 检索数据时的执行计划，如图 18-2 所示。其中的各个图标被称为物理运算符，与其相对的是逻辑运算符。例如，join 是一个逻辑运算符，而 nested loops joins 则是一个物理运算符。逻辑运算符描述了需要执行哪种关系查询处理操作，而物理运算符使用具体的方法或算法来实施逻辑运算符所定义的操作。图中的这些运算符说明列出了执行查询或数据操作语言（DML）语句的步骤，从这方面看，查询计划是由物理运算符组成的一个树。

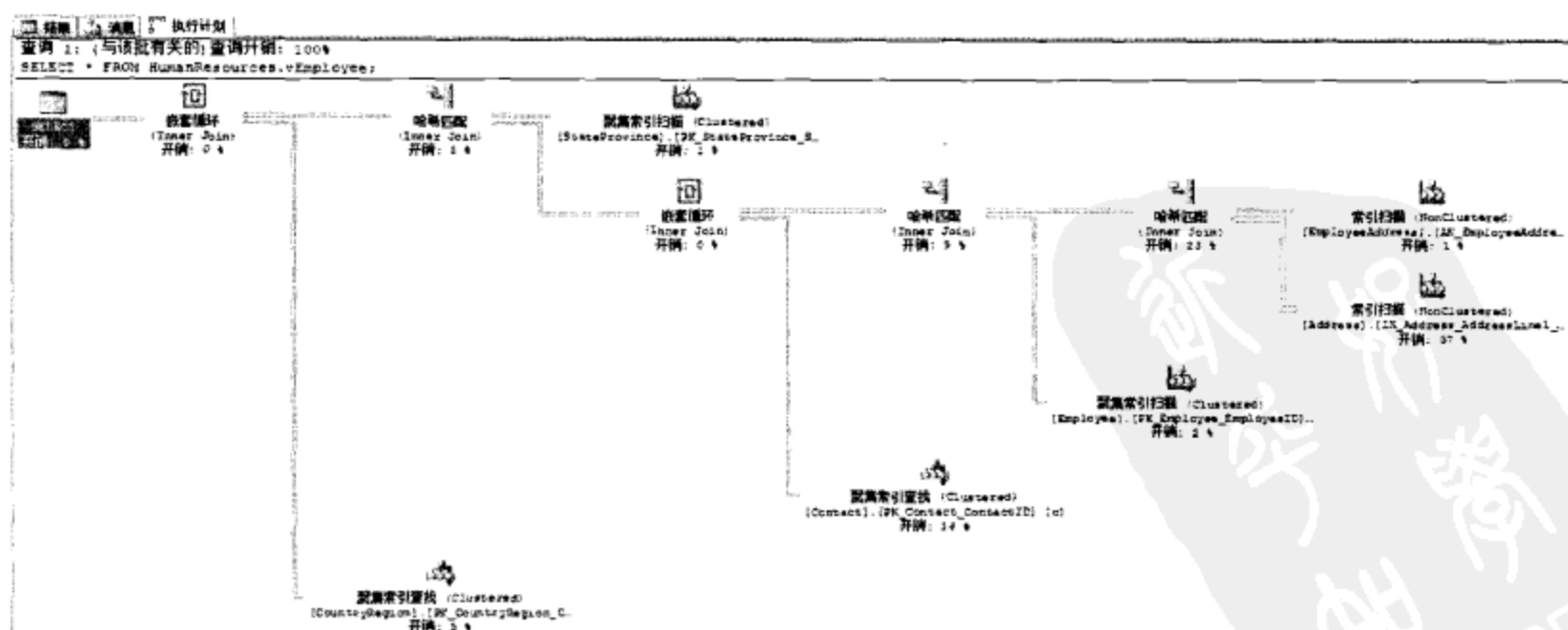


图 18-2 访问 HumanResources.vEmployee 视图时生成的执行计划

1. 查询语句分析

与其他语言编译器相同，在将查询语句生成为查询计划之前，分析器首先扫描语句并将其分成逻辑单元，如关键字、表达式、运算符和标识符。在该过程中，分析器将检查这些逻辑单元中是否存在语法错误，但是，它并不对表或列的有效性进行验证。例如，对于下面的语句，选择列的“[列 1]”中存在空格，所以需要包含在括号内，这符合验证规则，但是至于“[列 1]”是不是位于 Sales.SalesPerson 中，则不是该步骤需要做的事情。

```
SELECT [列 1] FROM Sales.SalesPerson;
```

2. 生成查询树

在语句分析完成后，需要对关键字、表达式、运算符等逻辑单元进行处理，包括将二元运算符转换为 n 元运算符、将表名称或列名称关联到实际的表或列对象、从树的叶级节点自下而上进行数据类型派生等操作，最终生成查询树，为查询优化做准备。也就是说，该步骤虽然描述了一个树，但是节点之间析取数据的方法并没有确定。

在将表名称或列名称关联到实际的表或列对象过程中，如果引用的是视图对象，则会用定义该视图的查询树来替换对视图的引用。如果视图还引用了其他视图，则会递归地进行解析，直到实际的表。例如，图 18-2 中所示的查询虽然访问的是 HumanResources.vEmployee 视图，但是你看到的是对 EmployeeAddress 等的索引扫描。

3. 生成执行计划

查询优化器是一个基于成本的优化器。每个可能的查询计划都会耗费一定的计算资源，查询优化器必须从这些可能的计划中选择一个预计成本最低的计划。但是，有些复杂的查询语句可能会有成千上万个可能的查询计划。在这种情况下，查询优化器不会分析所有的可能组合，而是使用复杂的算法来选择一个能够接近最低成本的查询计划。

当然，查询优化器也不仅仅选择资源成本最低的查询计划，还会选择能将结果最快地返回给用户且资源成本合理的计划。例如，与串行处理查询相比，并行处理查询使用的资源一般更多但完成查询的速度更快。在不对服务器的负荷产生负面影响的情况下，查询优化器将使用并行查询计划返回结果。

查询优化器在估计从表或索引中提取信息的不同方法所需的资源成本时，依赖于分发内容统计信息。这些统计信息是一些对象，其中包含与值在表或索引视图的一列或多列中的分布有关的统计信息。查询优化器使用这些统计信息来估计查询结果中的基数或行数，从而创建高效的查询计划。例如，在一个代表汽车的表中，很多汽车出自同一制造商，但每辆车都有唯一的车牌号。使用车牌号索引比制造商索引更具选择性。

18.2 执行计划的缓存与执行

为了提高执行计划的读取速度，数据库服务器有一个用于存储执行计划和数据缓冲区的内存池。执行计划生成以后，将被放置在该内存池中，池中用于存储执行计划的部分称为过程缓存。池内分配给执行计划或数据缓冲区的百分比并不是固定的，它随系统状态动态波动。

18.2.1 执行计划的副本和执行上下文

执行计划在缓存中主要包含两部分内容：执行计划的副本和执行上下文。

执行计划是一个可以重入的只读数据结构，可由任意数量的用户使用。但是，内存中执行计划的副本永远不超过两个：一个副本用于所有的串行执行，另一个用于所有的并行执行。并行副本覆盖所有的并行执行，与并行执行的并行度无关。

当用户执行查询时，数据库引擎会为用户分配一个包含其执行专用数据（如参数值）的数据结构，这个数据结构就称为执行上下文，如图 18-3 所示。这是真正的执行结构。



图 18-3 执行上下文

执行上下文数据结构可以重新使用，如果用户执行查询而其中的一个结构未使用，将会用新用户的上下文重新初始化该结构。也就是说，虽然可以重新使用，但是在同一时间只能有一条线程可以使用该结构。

由图 18-3 可以看出，在过程缓存中，一个执行计划可以对应多个执行上下文。但是，由于批处理中的执行逻辑原因，继承自同一执行计划的执行上下文的结构可能并不相同。例如，假设批处理中包含 IF 判断语句，在连接 1 中建立的执行上下文判断结果为真，而连接 2 中建立的执行上下文判断结果为假，则会导致这两个上下文中内容并不相同。

18.2.2 执行计划的开销管理

只要过程缓存中有足够的存储空间，执行计划将保留在其中。如果内存不足，数据库引擎将使用基于开销的方法来确定从过程缓存中删除哪些执行计划，以及对每个执行计划增加和降低当前开销变量。

当某个用户进程将执行计划插入缓存中时，该用户进程会将当前开销设置为等于原始查询编译开销。而对于即席执行计划，该用户进程会将当前开销设置为零。但是，当用户进程再次引用执行计划时，都会将当前开销重置为原始编译开销。因此，对于即席执行计划，用户进程数量会增加当前开销。但是，对于所有计划而言，当前开销的最大值就是原始编译开销。

如果遇到内存不足的情况，数据库引擎将会把执行计划从过程缓存中删除以进行响应。数据库引擎通过一种检查机制来确定删除哪些执行计划，在检查时，如果发现没有查询使用当前计划，则数据库引擎将降低该计划的当前开销以将其推向零。但是，开销为零的执行计划不会自动被删除，只有在内存不足的情况下，数据库引擎在检查时才会删除该计划。数据库引擎会重复检查每个执行计划，直至删除了足够多的执行计划，满足内存需求后，数据库引擎不再降低未使用执行计划的当前开销，并且所有执行计划都将保留在过程缓存中，即使其开销为零也是如此。

数据库引擎可以通过资源监视器和用户线程检查并发运行的执行计划，降低每个未使用执行计划的当前开销来响应内存不足问题。如果存在全局内存不足的情况，资源监视器将会从过程缓存中删除执行计划。它释放内存以强制实施系统内存、进程内存、资源池内存和所有缓存最大大小的策略。

当存在单一高速缓存不足的情况时，用户线程将会从过程缓存中删除执行计划。它们强制实施最大单一缓存大小和最大单一缓存条目数的策略。

如果希望从缓存中手动删除执行计划，可以使用 `DBCC FREEPROCCACHE` 命令。在没有为其指定任何参数的情况下，会删除缓存中的所有计划。如果希望删除指定的执行计划，可以为该命令指定一个参数，该参数可以是执行计划的句柄，或是 SQL 语句的句柄，或是资源调控器资源池的名称。执行计划的句柄可以从 `sys.dm_exec_cached_plans`、`sys.dm_exec_requests` 等对象的 `plan_handle` 列获取，SQL 语句句柄可以从 `sys.dm_exec_query_stats`、`sys.dm_exec_requests` 等对象的 `sql_handle` 列获取，资源调控器资源池的名称可以从 `sys.dm_resource_governor_resource_pools` 动态管理视图的 `pool_name` 列获取。例如，可以通过以下语句查询出执行计划的 SQL 语句句柄，然后使用 `DBCC FREEPROCCACHE` 命令删除指定计划。

```
SELECT * FROM sys.dm_exec_requests
WHERE sql_handle IS NOT NULL;
DBCC FREEPROCCACHE (0x02000000664D8E36299BAC80D3960B1017CF8E7C2BF196EE)
```

18.3 执行计划的重用

执行计划被缓存后, 当其所对应的查询语句再次执行时, 会重用该执行计划。现在的问题是, 多条类似语句能否使用相同的执行计划? 这样可以显著提高执行计划的重用率, 减少缓存中的执行计划数目。要解决这个问题, 需要在查询语句中使用参数或参数标记, 这可以提高查询语句与执行计划的匹配能力。

18.3.1 通过简单参数化提高计划重用率

如果执行不带参数的查询语句, 数据库引擎将在内部对该语句进行参数化以增加将其与现有执行计划相匹配的可能性, 这个过程就是简单参数化。来看下面的语句:

```
SELECT * FROM AdventureWorks.Production.Product
WHERE ProductSubcategoryID = 1;
```

数据库引擎将把值 1 指定为参数, 并在此基础上生成执行计划。由于这种简单参数化, 数据库引擎将认为以下语句将生成与上面语句相同的执行计划, 因此会对下面的语句重用上面生成的执行计划:

```
SELECT * FROM AdventureWorks.Production.Product
WHERE ProductSubcategoryID = 4;
```

18.3.2 通过强制参数化提高计划重用率

在简单参数化的默认行为下, 只会对相对较少的一些查询进行参数化。但是, 在 ALTER DATABASE 语句中将 PARAMETERIZATION 选项设置为 FORCED, 可启用对数据库中的所有查询进行参数化, 当然这也可能会受到某些限制。也可以通过“数据库属性”对话框进行强制参数化设置, 选择对话框中的“选项”页, 将“杂项”下的“参数化”选项选择为“强制”, 如图 18-4 所示。对于存在大量并发查询的数据库, 这样做可以减少查询编译的频率, 从而提高数据库的性能。

在强制参数化模式下, SELECT、INSERT、UPDATE 或 DELETE 语句中出现的任何文本值都将在查询编译期间转换为参数。但是, 下列查询构造中出现的文本除外:

- 由于 SQL Server 已对存储过程、触发器、或用户定义函数进行了编译, 所以不会再对其中的语句再进行参数化;
- INSERT...EXECUTE 语句;
- 已在客户端应用程序中参数化的预定义语句;
- 当查询语句中包含用于 xml 数据查询的 XQuery 方法调用时, 如果方法中参数未参数化, 则语句的其他部分将参数化。例如, 为下面语句生成执行计划时, “ProductModelID = 7” 会被参数化 (可以通过 sys.syscacheobjects 动态管理视图的 sql 列来确定查询是否已参数化。如果查询已参数化)

化, 则参数的名称和数据类型将出现在此列中已提交的批的文本前面, 如(@0 int)。

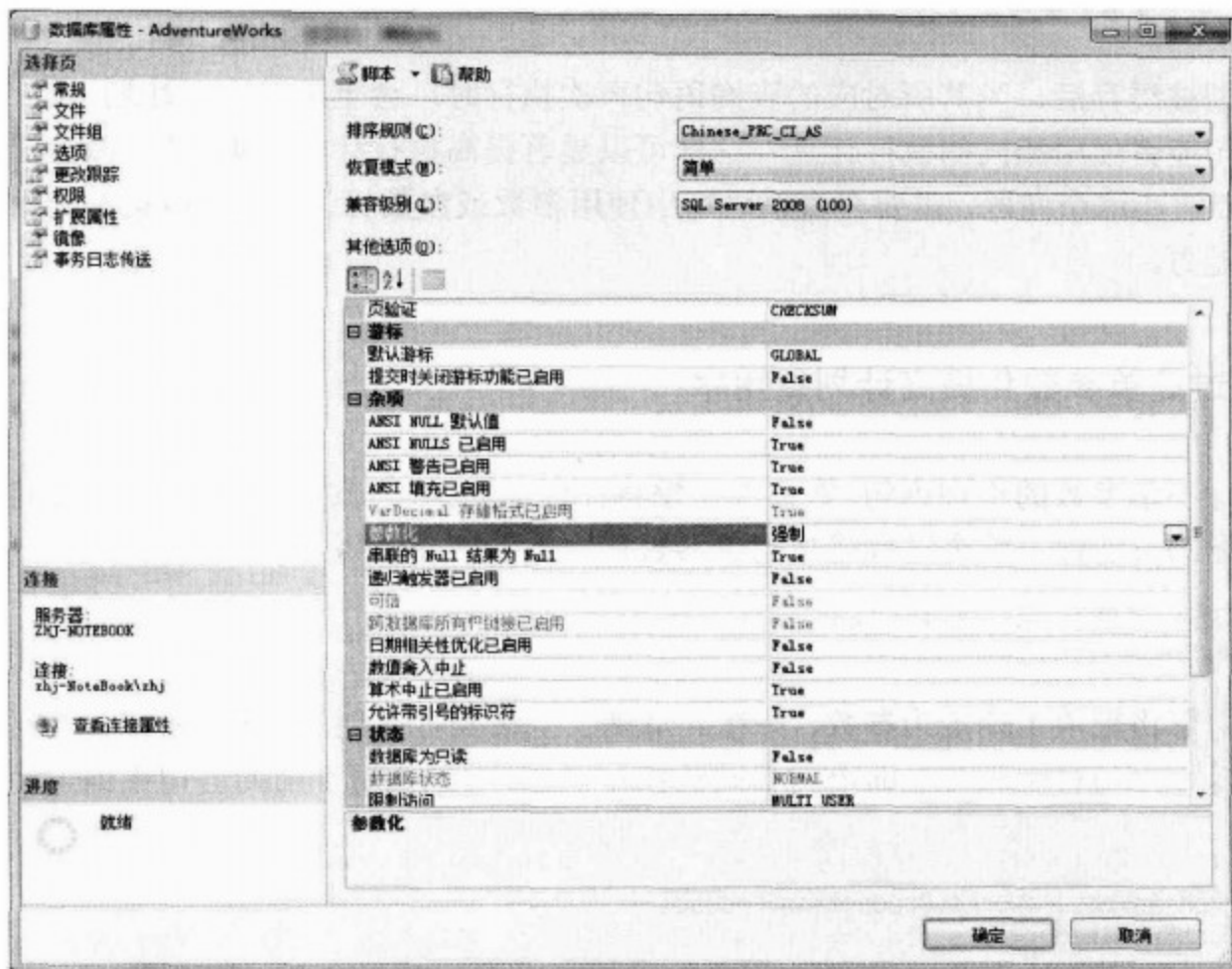


图 18-4 设置强制参数化

```
SELECT Instructions.query('declare namespace AWMI="http://schemas.microsoft.com/sqlserver/2004/07/
adventure-works/ProductModelManuInstructions";
    /AWMI:root/AWMI:Location[@LocationID=10]
') as Result
FROM Production.ProductModel
WHERE ProductModelID = 7;
```

- 游标内的语句;
- 不推荐使用的查询构造;
- 在 ANSI_PADDING 或 ANSI_NULLS 设置为 OFF 的上下文中执行的任何语句;
- 包含 2097 个以上的可参数化文字的语句;
- 引用变量的语句, 例如 “WHERE T.col2 >= @bb”;
- 包含 RECOMPILE 查询提示的语句;
- 包含 COMPUTE 子句的语句;
- 包含 WHERE CURRENT OF 子句的语句。

某些子查询也会存在不会被参数化的情况, 如 SELECT 语句列表中的子查询、IF 语句中出现的子查询 SELECT 语句, 在此不再赘述。但是同一查询中的其他子句有可能会强制参数化。

启用强制参数化后, 仍会发生简单参数化。例如, 根据强制参数化规则, 无法将以下查询参

数化:

```
SELECT * FROM Person.Address
WHERE AddressID = 1 + 2;
```

但是, 根据简单参数化规则, 可以将该查询参数化。尝试强制参数化失败后, 仍将接着尝试简单参数化。

18.3.3 使用显式参数化提高计划重用率

上文已经介绍了简单参数化和强制参数化, 这些都是数据库引擎的自动化行为。为了提高执行计划的重用率, 应当在设计查询时尽量使用参数化设置。

可以通过 `sp_executesql` 进行参数化查询的调用。参考下面的语句:

```
DECLARE @MyIntParm INT
SET @MyIntParm = 1
EXEC sp_executesql
    N'SELECT * FROM AdventureWorks.Production.Product WHERE ProductSubcategoryID = @Parm',
    N'@Parm INT', @MyIntParm
```

建议对动态生成 SQL 语句的触发器、SQL 脚本或存储过程使用此方法。

在 ADO、OLE DB 和 ODBC 中, 可以使用参数标记 (?) 的方式设计查询语句。例如, 下面是在 ODBC 应用程序中进行参数化查询设计的方法。

- 使用 `SQLBindParameter` 将整数变量绑定到 SQL 语句中的第 1 个参数标记。
- 为变量赋整数值。
- 执行语句, 并指定参数标记 (?) :

```
SQLExecDirect(hstmt,
    "SELECT * FROM AdventureWorks.Production.Product WHERE ProductSubcategoryID = ?",
    SQL_NTS);
```

18.4 执行计划的重新编译

对数据库中对象或表中数据的更改, 可能会导致执行计划效率降低或无效。数据库引擎会自动检测到这些导致执行计划无效的更改, 当再次执行查询时, 会重新编译新的计划。

在 SQL Server 2000 中, 只要批处理中的语句导致了重新编译, 就会重新编译整个批处理, 无论此批处理是通过存储过程、触发器、即席批查询, 还是通过预定义的语句进行提交。在 SQL Server 2005 和更高版本中, 只会重新编译批处理中导致重新编译的语句。这种语句级重新编译方式有助于提高性能, 因为在大多数情况下, 只有少数语句导致了重新编译。

如果希望手动编译执行计划, 可以使用 `sp_recompile` 命令。执行该命令后, 存储过程和触发器在下次运行时会被重新编译。例如, 下面的语句可以使作用于 `Customer` 表上的存储过程在下次运

行时重新编译。

```
USE AdventureWorks;
GO
EXEC sp_recompile N'Sales.Customer';
```

对于单个存储过程，可以通过 **WITH RECOMPILE** 选项指定在执行时重新编译。但是，该执行计划仅用于当前查询，不会在过程缓存中存储。如果之前调用该存储过程时未指定 **WITH RECOMPILE** 选项，会在过程缓存中存储该存储过程的执行计划，但是，在执行包含 **WITH RECOMPILE** 选项的该存储过程时，并不会使用该执行计划。参考下面的示例：

```
-- 创建一个名为 p1 的存储过程。
USE AdventureWorks;
GO
CREATE PROC p1
AS
SELECT * FROM Person.Address;
GO

-- 你可以重复执行下面的语句多次
EXEC sp1 WITH RECOMPILE;
GO

-- 然后使用下面的语句查看过程缓存中的内容，会发现并没有 p1 的执行计划
SELECT * FROM sys.syscacheobjects
WHERE sql NOT LIKE '%cache%'
AND sql NOT LIKE '%sys.%';
```

要跟踪执行计划的编译事件，可以使用 SQL Server Profiler 工具。打开 SQL Server Profiler 后，选择主菜单“文件”→“新建跟踪”选项，在打开的“跟踪属性”对话框中单击“使用模板”下拉菜单中的“空白”选项。单击“事件选择”选项卡，选择 **Stored Procedures** 下面的 **SP:Recompile** 选项和 **TSQL** 下面的 **SQL:StmtRecompile** 选项，以及选择 **Performance** 下面的 **Auto Stats** 选项，如图 18-5 所示。

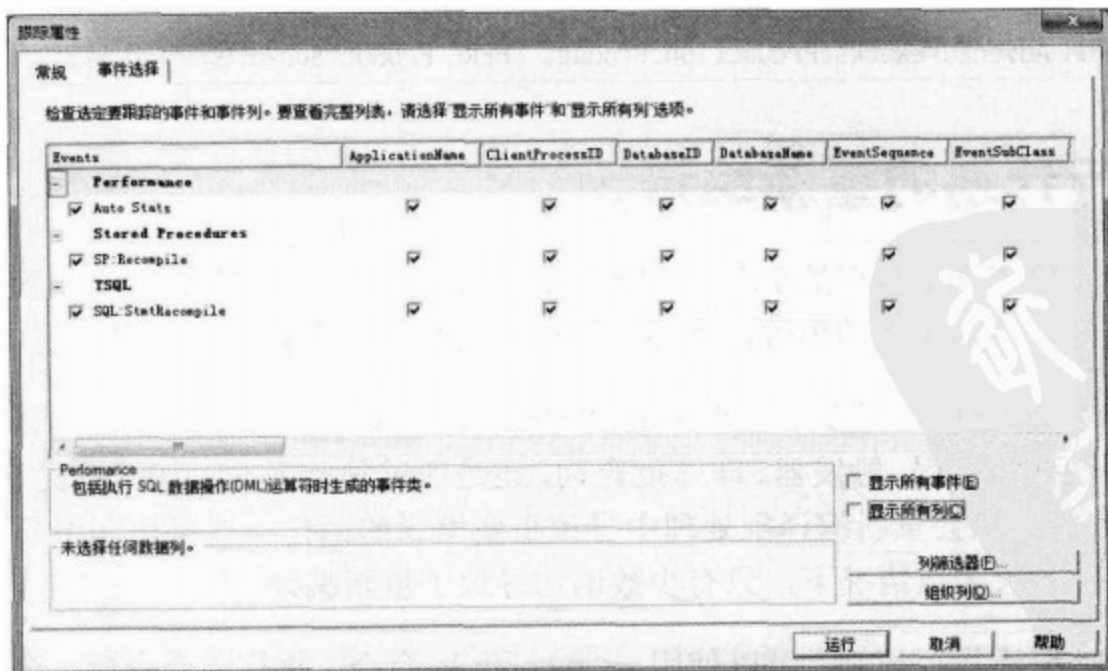


图 18-5 选择要监控的

SP:Recompile 和 SQL:StmtRecompile 都是报告语句级重新编译事件。SP:Recompile 仅针对存储过程和触发器生成,而 SQL:StmtRecompile 则针对存储过程、触发器、即席批查询、使用 sp_executesql 执行的批处理、已准备的查询和动态 SQL 生成。SP:Recompile 和 SQL:StmtRecompile 的 EventSubClass 列都包含一个整数代码,用以指明重新编译的原因,如表 18-1 所示。

表 18-1 EventSubClass 值的含义

值	说 明
1	架构已更改
2	统计信息已更改
3	编译已延迟
4	SET 选项已更改
5	临时表已更改
6	远程行集已更改
7	FOR BROWSE 权限已更改
8	查询通知环境已更改
9	分区视图已更改
10	游标选项已更改
11	已请求 OPTION (RECOMPILE)



4

第 4 部分 实战篇

第 19 章 SQL 查询演练



第 19 章 SQL 查询演练

本章将针对实际生活或工作中遇到的一些问题提供具体的解决方案。对于同一个问题，可能有多个解决方案，如何写出一个快捷高效的查询语句，是检验 SQL 程序员能力高低的主要依据。

19.1 同一时间范围内并发数统计

这是在做一个酒店系统时遇到的问题。通常情况下，一个服务生仅负责一个房间的客人用餐，但是，考虑到效益原因，酒店鼓励服务生同时为多个房间的客人提供服务。这样，酒店每天要对在同一时间段内服务房间最多的一名服务生进行奖励。表 19-1 列出了一天中酒店服务生所服务的房间号和服务时间。

表 19-1

服务清单

room_id	waiter_name	start_time	end_time
1	张岚	2009-02-01 11:30:00.000	2009-02-01 13:30:00.000
2	张岚	2009-02-01 11:40:00.000	2009-02-01 13:15:00.000
3	孙静	2009-02-01 11:45:00.000	2009-02-01 14:20:00.000
4	孙静	2009-02-01 11:39:00.000	2009-02-01 13:20:00.000
5	孙静	2009-02-01 11:49:00.000	2009-02-01 14:16:00.000
6	孙静	2009-02-01 10:37:00.000	2009-02-01 11:00:00.000
3	孙静	2009-02-01 17:45:00.000	2009-02-01 18:20:00.000
4	孙静	2009-02-01 17:39:00.000	2009-02-01 18:25:00.000
5	孙静	2009-02-01 17:49:00.000	2009-02-01 18:36:00.000
6	孙静	2009-02-01 17:37:00.000	2009-02-01 18:40:00.000

下面是建立服务清单数据的 SQL 语句。

```
CREATE TABLE Waiters
(
    room_id int,
    waiter_name char(20),
    start_time datetime,
    end_time datetime
);
INSERT INTO Waiters VALUES
(1, '张岚', '2009-02-01 11:30', '2009-02-01 13:30'),
```

```
(2,'张岚','2009-02-01 11:40','2009-02-01 13:15'),
(3,'孙静','2009-02-01 11:45','2009-02-01 14:20'),
(4,'孙静','2009-02-01 11:39','2009-02-01 13:20'),
(5,'孙静','2009-02-01 11:49','2009-02-01 14:16'),
(6,'孙静','2009-02-01 10:37','2009-02-01 11:00'),
(3,'孙静','2009-02-01 17:45','2009-02-01 18:20'),
(4,'孙静','2009-02-01 17:39','2009-02-01 18:25'),
(5,'孙静','2009-02-01 17:49','2009-02-01 18:36'),
(6,'孙静','2009-02-01 17:37','2009-02-01 18:40');
```

为了更清楚地显示出每名服务生在同一时间内所服务房间的数量,我们将服务清单中的数据以图形的方式表示了出来,如图 19-1 所示。可以看出,虽然孙静在中午和晚上都是服务了 4 个房间,但是只有晚上 4 个房间的时间是重叠的。张岚是 2 个房间的时间重叠在一起。这样不难得出答案,在同一时间内张岚最多的服务房间是 2 个,孙静是 4 个。

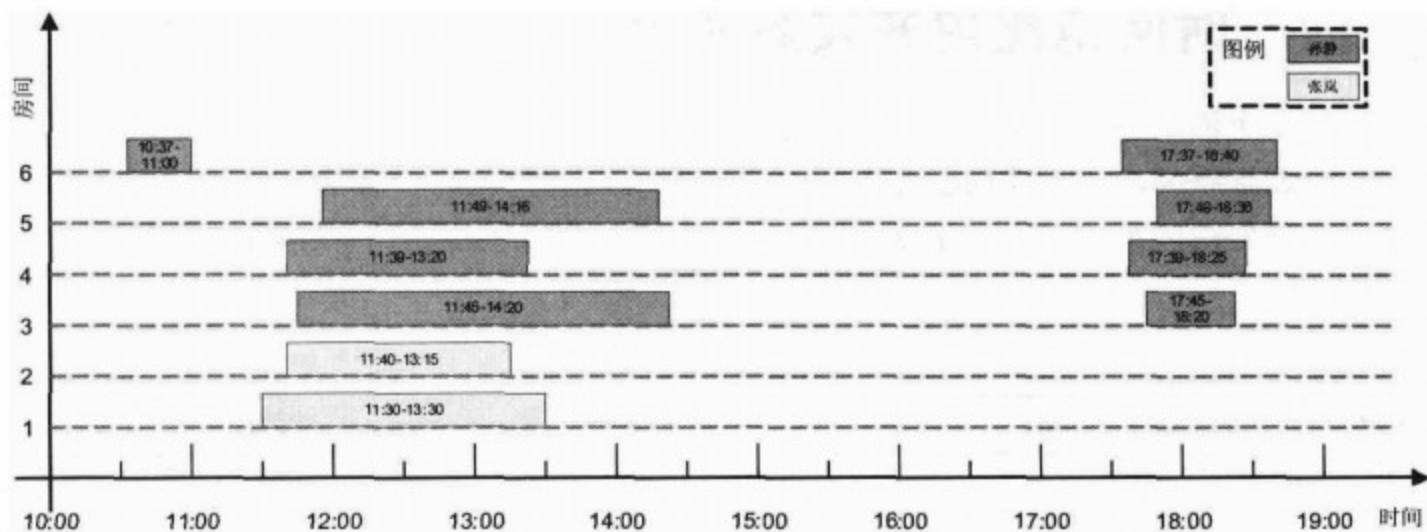


图 19-1 服务清单数据图形展示

19.1.1 使用子查询

要解决这个问题,首先要找出如何判断时间重叠的方法。由图 19-1 可以看出,当一个房间的服务开始之后,只要第 2 个房间的服务开始时间小于或等于第 1 个房间,并且结束时间大于第 1 个房间的结束时间的,就说明这两个房间的服务时间存在重叠部分。例如,中午张岚在 1 号房间的服务开始时间小于 2 号房间的结束时间,并且结束时间大于 2 号房间的结束时间,说明这两个房间的服务时间重叠。而中午孙静在 5 号房间的服务开始之后,6 号、4 号、3 号的服务开始时间都小于 5 号,但是 6 号的服务结束时间小于 5 号的开始时间,说明 6 号与 5 号服务时间并不重叠。

按照上述逻辑,首先给出以下查询语句,以展示每个房间服务开始时所重叠的房间数量,查询结果如表 19-2 所示。表中的黑体部分数据能够更加明显地说明出这种重叠,如 4 号与 6 号重叠,3 号与 4、6 号重叠,5 号则与 3、4、6 号重叠。

```
SELECT W1.waiter_name,
       W1.start_time,
       W1.end_time,
```



```

        MAX(W1.room_id) AS room_id,
        COUNT(*) AS tally
FROM Waiters AS W1
    INNER JOIN Waiters AS W2
        ON W1.waiter_name = W2.waiter_name
        AND W2.start_time <= W1.start_time
        AND W2.end_time > W1.start_time
GROUP BY W1.waiter_name, W1.start_time, W1.end_time, W1.room_id
ORDER BY W1.waiter_name, W1.start_time, room_id;

```

表 19-2

房间重叠数量

waiter_name	start_time	end_time	room_id	tally
孙静	2009-02-01 10:37:00.000	2009-02-01 11:00:00.000	6	1
孙静	2009-02-01 11:39:00.000	2009-02-01 13:20:00.000	4	1
孙静	2009-02-01 11:45:00.000	2009-02-01 14:20:00.000	3	2
孙静	2009-02-01 11:49:00.000	2009-02-01 14:16:00.000	5	3
孙静	2009-02-01 17:37:00.000	2009-02-01 18:40:00.000	6	1
孙静	2009-02-01 17:39:00.000	2009-02-01 18:25:00.000	4	2
孙静	2009-02-01 17:45:00.000	2009-02-01 18:20:00.000	3	3
孙静	2009-02-01 17:49:00.000	2009-02-01 18:36:00.000	5	4
张岚	2009-02-01 11:30:00.000	2009-02-01 13:30:00.000	1	1
张岚	2009-02-01 11:40:00.000	2009-02-01 13:15:00.000	2	2

从上表可以看出，只要按服务生的姓名取出 tally 的最大值，就可以计算出每名服务生同时服务房间的最大数。下面是以子查询方式给出的最终语句，查询结果如表 19-3 所示。

```

SELECT T1.waiter_name, MAX(T1.tally) AS tally
FROM (SELECT W1.waiter_name,
        W1.start_time,
        W1.end_time,
        COUNT(*) AS tally
FROM Waiters AS W1
    INNER JOIN Waiters AS W2
        ON W1.waiter_name = W2.waiter_name
        AND W2.start_time <= W1.start_time
        AND W2.end_time > W1.start_time
GROUP BY W1.waiter_name, W1.start_time, W1.end_time) AS T1
GROUP BY T1.waiter_name;

```

表 19-3

最终查询结果

waiter_name	tally
孙静	4
张岚	2

19.1.2 使用 CTE

上面使用子查询的方式得出了最终结果，这里的子查询实际上起到的是一种临时结果集作用。

所以，这个问题也可以使用 CTE 的方式进行查询。

```
WITH T1 (waiter_name, start_time, end_time, tally) -- 定义 CTE 表达式的名称和列
AS
(
    SELECT W1.waiter_name,
           W1.start_time,
           W1.end_time,
           COUNT(*) AS tally
    FROM Waiters AS W1
    INNER JOIN Waiters AS W2
        ON W1.waiter_name = W2.waiter_name
        AND W2.start_time <= W1.start_time
        AND W2.end_time > W1.start_time
    GROUP BY W1.waiter_name, W1.start_time, W1.end_time
)
SELECT waiter_name, MAX(tally) AS tally
FROM T1
GROUP BY T1.waiter_name;
```

19.2 时间段天数统计

在做利息计算时会经常遇到这个问题。例如，在如表 19-4 所示的表中记录着贷款的发放和分期归还信息，现在要计算该笔贷款的利息，则应当分期计算。从 2009 年 11 月 1 日至 2009 年 12 月 1 日的贷款额是 20000.00 元，资金使用时间是 30 天，这段期间的利息=20000.00×30×日利率。归还 10000.00 元后，从 2009 年 12 月 1 日至 2010 年 1 月 1 日的贷款额变为 10000.00 元，资金使用时间是 31 天，这段期间的利息=10000.00×31×日利率。现在关键的问题是如何计算出两个日期之间的天数。

表 19-4

贷款明细表

loan_id	loan_date	summary	dr_amt	cr_amt	bal
1	2009-11-01	发放贷款	20000.00	NULL	20000.00
1	2009-12-01	归还贷款	NULL	10000.00	10000.00
1	2010-01-01	归还贷款	NULL	5000.00	5000.00
1	2010-02-01	结清	NULL	5000.00	0.00

首先来创建上面的示例表，代码如下。

```
CREATE TABLE Loans
(loan_id int,
 loan_date date,
 summary char(10),
 dr_amt decimal(12,2),
 cr_amt decimal(12,2),
 bal decimal(12,2));

INSERT INTO Loans
VALUES (1, '2009-11-01', '发放贷款', 20000.00, NULL, 20000.00),
       (1, '2009-12-01', '归还贷款', NULL, 10000.00, 10000.00),
       (1, '2010-01-01', '归还贷款', NULL, 5000.00, 5000.00),
       (1, '2010-02-01', '结清', NULL, 5000.00, 0.00);
```

下面的语句将 Loans 表打开 2 次，然后取出大于当前日期的第 1 个日期，保存在 next_date 列中。查询结果如表 19-5 所示。

```
SELECT loan_id, loan_date,
       (SELECT MIN(loan_date)
        FROM Loans AS L2
        WHERE L2.loan_id = L1.loan_id
          AND L2.loan_date > L1.loan_date) AS next_date,
       bal
FROM Loans AS L1
```

表 19-5

查询结果

loan_id	loan_date	next_date	bal
1	2009-11-01	2009-12-01	20000.00
1	2009-12-01	2010-01-01	10000.00
1	2010-01-01	2010-02-01	5000.00
1	2010-02-01	NULL	0.00

由上表可以看出，next_date- loan_date 就可以计算出两次日期之间的天数。下面是完整的语句：

```
SELECT loan_id, loan_date,
       DATEDIFF(DAY, loan_date,
                (SELECT MIN(loan_date)
                 FROM Loans AS L2
                 WHERE L2.loan_id = L1.loan_id
                   AND L2.loan_date > L1.loan_date)) AS diff_days,
       bal
FROM Loans AS L1;
```

19.3 数字范围统计

这是在做一个大型货场租赁系统时遇到的问题，在计算货场剩余存储空间时，不仅仅需要知道哪些货位是空闲的，还要能够判断出哪些货位之间是连续的。因为在新货物入场时，可以判断这些货物是否可以堆放在一起，而不是放在不连续的多个货位上，这样更便于管理，并且在出货时也更加迅速。

假设这个货场共有 100 个货位，现在已存放货物的货位是 1、2、3、4、87、89、99、100，则剩余空位是 5~86、88、90~98。数据库的设计方式一般有两种：一种是在表中为每个货位建一条记录，类似表 19-6 所示的结构设计；另一种设计方式是仅将存放有货物的货位号放在表中，也就是存货情况表中仅有货位编号列，存放 1、2、3、4、87、89、99、100 这几个数值。

表 19-6

存货情况表

货位编号	是否存放有货物 (1-是, 0-否)
1	1
2	1
3	0
...	...

相对于大型数据库而言,第一种架构设计对于查询语句编写方面会更方便一些。如果数据需要驻留在一些手持设备上,多数开发人员会更喜欢第二种架构设计,因为它能够节省宝贵的存储空间。尤其是当表的数据量非常大时,这种设计方式更能显示出它的优势。这里我们将以第二种架构设计方式来演示查询的创建方法,下面是创建示例表的语句。

```
CREATE TABLE Freights(Numb int NOT NULL);
INSERT INTO Freights VALUES
(1),(2),(3),(4),(87),(89),(99),(100);
```

19.3.1 查找剩余空位区间和剩余空位编号

要查找剩余空位区间,就是要找出如表 19-7 所示的数值范围。

表 19-7 剩余空位区间

货位开始编号	货位结束编号
5	86
88	88
90	98

要找出这些区间的开始和结束编号,需要在间断之前的值加 1,在下一组编号开始之间的值减 1。例如,表中的 5~86 是在 4 的基础上加 1、在 87 的基础上减 1 得来的。

首先来看下面的语句,用于获取每个货位号的下一货位号,得到的结果如表 19-8 所示。

```
SELECT F1.Numb AS n1,
       (SELECT MIN(F2.Numb)
        FROM Freights AS F2
        WHERE F2.Numb > F1.Numb) AS n2
FROM Freights AS F1;
```

表 19-8 每一货位号的下一货位号

n1	n2
1	2
2	3
3	4
4	87
87	89
89	99
99	100
100	NULL

可以看出,只要找出 n2-n1 大于 1 的货位组,并在 n1 上加 1,在 n2 上减 1,就可以得到表 19-7 所示的剩余空位区间。参考下面的语句。


```

SELECT n1 + 1 AS start_id, n2 - 1 AS end_id
FROM (SELECT F1.Numb AS n1,
      (SELECT MIN(F2.Numb)
       FROM Freights AS F2
       WHERE F2.Numb > F1.Numb) AS n2
      FROM Freights AS F1) AS F3
WHERE n2 - n1 > 1;

```

上面是使用子查询的方式作为中间结果的存储，也可以使用 CTE 方式，参考下面的语句。

```

WITH F3 (n1, n2)
AS
(
    SELECT F1.Numb AS n1,
           (SELECT MIN(F2.Numb)
            FROM Freights AS F2
            WHERE F2.Numb > F1.Numb) AS n2
    FROM Freights AS F1
)
SELECT n1 + 1 AS start_id, n2 - 1 AS end_id
FROM F3
WHERE n2 - n1 > 1;

```

而下面的语句则是使用内联接和分组计算的方法计算剩余空位区间，与上面的两种方式相比，此方式在 **GROUP BY** 时多出了排序操作，查询开销较大。

```

SELECT (F1.Numb + 1) AS start,
       (MIN(F2.Numb - 1)) AS finish
FROM Freights AS F1
     INNER JOIN Freights AS F2
       ON F2.Numb > F1.Numb
GROUP BY F1.Numb
HAVING (F1.Numb + 1) < MIN(F2.Numb);

```

如果希望返回的不是剩余空位区间，而是剩余空位编号，则需要建立一个全部的货位编号表。下面的语句使用了递归 CTE 循环来建立 1~100 的货位编号。

```

WITH Numbs AS
(
    SELECT 1 AS n
    UNION ALL
    SELECT n + 1 FROM Numbs WHERE n < 100
)
SELECT n FROM Numbs OPTION(MAXRECURSION 0);

```

只要全部货位编号在 **Freights** 表中不存在，则表示该货位号没有使用，参考下面的语句。

```

WITH Numbs AS
(
    SELECT 1 AS n
    UNION ALL
    SELECT n + 1 FROM Numbs WHERE n < 100
)
SELECT n FROM Numbs
WHERE n NOT IN (SELECT Numb FROM Freights)
OPTION(MAXRECURSION 0);

```

如果不需要返回全部的空货位号，而是几个的话，可以使用下面的查询语句。它使用了从 SQL

Server 2005 开始支持的窗口函数进行编号，返回比当前编号小的空货位号。查询结果如表 19-9 所示。对于其中的重复数值，可以使用 DISTINCT 关键字进行过滤。

```
SELECT Numb, rn, (Numb - rn) AS available_Numb
FROM (SELECT Numb, ROW_NUMBER() OVER (ORDER BY Numb)
      FROM Freights) AS F(Numb, rn)
WHERE rn < Numb
```

表 19-9 比当前货位号小的空货位号

Numb	rn	available_Numb
87	5	82
89	6	83
99	7	92
100	8	92

19.3.2 查找已用货位区间

现在已经使用的货位是 1、2、3、4、87、89、99、100，已用货位区间即 1~4、87~87、89~89、99~100。这些区间实际上是一组连续编号中的最小值和最大值，如 1~4 是货位 1、2、3、4 中的最小值和最大值。现在关键的问题是如何判断出这是一组数值，通过表 19-7 读者也许会发现一个有趣的问题，99、100 通过 Numb - rn 后得到的数值是相同的，说明这是一组数值。下面是去掉 WHERE 子句后的查询语句，结果如表 19-10 所示。

```
SELECT Numb, rn, (Numb - rn) AS available_Numb
FROM (SELECT Numb, ROW_NUMBER() OVER (ORDER BY Numb)
      FROM Freights) AS F(Numb, rn)
```

表 19-10 数据分组

Numb	rn	available_Numb
1	1	0
2	2	0
3	3	0
4	4	0
87	5	82
89	6	83
99	7	92
100	8	92

通过上表可以很清晰地看出数据分组情况，因此，我们给出下面的最终查询语句，查询结果如表 19-11 所示。

```
SELECT MIN(Numb) AS start, MAX(Numb) AS finish
FROM (
  SELECT Numb, rn, (Numb - rn) AS available_Numb
```

```

FROM (SELECT Numb, ROW_NUMBER() OVER (ORDER BY Numb)
      FROM Freights) AS F(Numb, rn)
) AS G(Numb, rn, available_Numb)
GROUP BY available_Numb;

```

表 19-11

已用货位区间

start	finish
1	4
87	87
89	89
99	100

19.4 地域范围内最大数统计

这是在做一个客户管理系统时遇到的问题，公司每年需要按销售区域评选出购买量最大的客户进行单独奖励。区域划分使用地域编码起止区间方式，例如，华东地区的起止编码是 10001~10003，在 10001、10002、10003 区域的客户都隶属于华东地区。下面是创建示例的代码，Area 表中存放着区域划分范围，Sales 存放着每个区域中客户的购买信息。

```

CREATE TABLE Areas
(area_name char(25) NOT NULL,
 start_code int NOT NULL,
 end_code int NOT NULL,
 CHECK (start_code <= end_code));
CREATE TABLE Sales
(sale_id int,
 cust_name char(15),
 cust_code int,
 sale_amt decimal (8,2));

INSERT INTO Areas
VALUES ('华东', 10001, 10003),
      ('华南', 10004, 10006),
      ('华北', 10007, 10009);

INSERT INTO Sales
VALUES (1, '张三', 10001, 1000.00),
      (2, '张三', 10002, 1000.00),
      (3, '李四', 10001, 4000.00),
      (4, '王五', 10005, 1000.00),
      (5, '王五', 10006, 2000.00),
      (6, '赵六', 10004, 1500.00);

```

要统计出每个区域中购买量最大的客户，我们首先想到的是使用 GROUP BY 子句进行分类汇总，参考下面的语句。

```

SELECT A.area_name, S.cust_name, SUM(S.sale_amt) AS sale_amt
FROM Areas AS A
JOIN Sales AS S
  ON S.cust_code BETWEEN A.start_code AND A.end_code
GROUP BY A.area_name, S.cust_name
ORDER BY area_name;

```

表 19-12 使用 GROUP BY 按区域、客户分类汇总结果

area_name	cust_name	sale_amt
华东	李四	4000.00
华东	张三	2000.00
华南	王五	3000.00
华南	赵六	1500.00

从表 19-12 可以看出，在华东地区李四的购买量最大，华南地区王五的购买量最大。如果仅统计每地区的最大销售额，不考虑客户名称，完全可以在上表的基础上使用类似以下语句。

```
SELECT area_name, MAX(sale_amt)
FROM [表 19-21]
GROUP BY area_name;
```

下面的语句将 GROUP BY 分组统计封装在 CTE 中，然后将 CTE 打开两次，取出每区域中购买量最大的客户。查询结果如表 19-13 所示。

```
WITH CTE (area_name, cust_name, sale_amt)
AS (SELECT A.area_name, S.cust_name, SUM(S.sale_amt)
    FROM Areas AS A
    JOIN Sales AS S
    ON S.cust_code BETWEEN A.start_code AND A.end_code
    GROUP BY A.area_name, S.cust_name)
SELECT C1.area_name, C1.cust_name, C1.sale_amt
FROM CTE AS C1
WHERE C1.sale_amt = (SELECT MAX(C2.sale_amt)
    FROM CTE AS C2
    WHERE C2.area_name = C1.area_name);
```

表 19-13 查询结果

area_name	cust_name	sale_amt
华南	王五	3000.00
华东	李四	4000.00

19.5 从分组中取前几行数据

这是在做一個考试成绩统计时遇到的问题。假设有如表 19-14 所示的数据，其中包含了 3 个班级的考生成绩，如果是希望获取全部数据的前 2 名，可以使用 TOP 配合 ORDER BY 子句轻易实现，但是如果我们希望取出每个班级中的前 2 名呢？事情就不这么简单了。

```
SELECT TOP(2) *
FROM Students
ORDER BY Achi DESC;
```

表 19-14

考试成绩表

ClassID	StuName	Achi
1	张山	100
1	李明	90
1	王磊	95
2	孙科	100
2	赵强	80
2	王智	90
3	李海	95

下面的语句用于创建示例表。

```
CREATE TABLE Students
(ClassID int,
StuName char(10),
Achi int);
INSERT INTO Students
VALUES(1, '张山', 100),
      (1, '李明', 90),
      (1, '王磊', 95),
      (2, '孙科', 100),
      (2, '赵强', 80),
      (2, '王智', 90),
      (3, '李海', 95);
```

19.5.1 使用联接获取前几行

如果将 **Students** 表打开两次，将一个考生与其大于或等于自己成绩的考生联接，我们看看会得到什么样的结果。参考下面的语句：

```
SELECT S1.*, S2.*
FROM Students AS S1
INNER JOIN Students AS S2
ON S1.ClassID = S2.ClassID
AND S2.Achi >= S1.Achi
ORDER BY S1.ClassID, S1.Achi DESC;
```

结果如表 19-15 所示。

表 19-15

联接结果

S1.ClassID	S1.StuName	S1.Achi	S2.ClassID	S2.StuName	S2.Achi
1	张山	100	1	张山	100
1	王磊	95	1	张山	100
1	王磊	95	1	王磊	95
1	李明	90	1	张山	100
1	李明	90	1	李明	90

续表

S1.ClassID	S1.StuName	S1.Achi	S2.ClassID	S2.StuName	S2.Achi
1	李明	90	1	王磊	95
2	孙科	100	2	孙科	100
2	王智	90	2	孙科	100
2	王智	90	2	王智	90
2	赵强	80	2	孙科	100
2	赵强	80	2	赵强	80
2	赵强	80	2	王智	90
3	李海	95	3	李海	95

从表 19-15 中可以看出, 1 班中的第 1 名张山有 1 条记录, 第 2 名王磊有 2 条记录, 第 3 名有 3 条记录。因此, 可以使用下面的语句来获取每班中前 2 名的考生。查询结果如表 19-16 所示。

```
SELECT S1.ClassID, S1.Achi, MAX(S1.StuName) AS StuName
FROM Students AS S1
  INNER JOIN Students AS S2
    ON S1.ClassID = S2.ClassID
   AND S2.Achi >= S1.Achi
GROUP BY S1.ClassID, S1.Achi
HAVING COUNT(*) <= 2
ORDER BY S1.ClassID, S1.Achi DESC;
```

表 19-16

每班中前 2 名的考生

ClassID	Achi	StuName
1	100	张山
1	95	王磊
2	100	孙科
2	90	王智
3	95	李海

19.5.2 使用窗口排名函数获取前几行

窗口计算是从 SQL Server 2005 开始提供的新技术, 每一组数据被称为一个窗口, RANK() 和 DENSE_RANK() 函数都可以按窗口进行排名计算, 表 19-17 描述了这两种排名方式的差异。

表 19-17

RANK() 和 DENSE_RANK() 函数排名的差异

StuName	Achi	RANK() 排名	DENSE_RANK() 排名
张三	100	1	1
李四	100	1	1
王五	95	3	2
赵六	90	4	3

从表中可以看出,无论是 `RANK()` 还是 `DENSE_RANK()`, 相同的考试成绩排名值是相同的,张三和李四都是第 1, 也就是我们常说的并列第 1。但是, 两人之后的名次 `RANK()` 和 `DENSE_RANK()` 出现了差异, 从两人并列第 1 的角度讲, 他们两人之后的名次应当是第 2, 这是 `DENSE_RANK()` 函数的排名方式, 也称为密集排名, 因为它的名次之间没有间隔; 前面已经有 2 个人 100 分了, 他们后面的人应当是第 3 个高分者, 从这个角度理解, 后面的名次应当是第 3, 这是 `RANK()` 的排名方式。

例如, 下面的语句按班级分组、按成绩降序密集排名, 查询结果如表 19-18 所示。与上面使用联接获取前 2 名的方式相比, 使用排名函数可以正确处理成绩并列现象。同样是获取成绩前 2 名, 存在成绩并列时, 使用排名函数从每个班级中取出的人数有可能超过 2 个。

```
SELECT ClassID, StuName, Achi,
       DENSE_RANK() OVER(PARTITION BY ClassID ORDER BY Achi DESC) AS rank_rn
FROM Students;
```

表 19-18

排序结果

ClassID	StuName	Achi	rank_rn
1	张山	100	1
1	王磊	95	2
1	李明	90	3
2	孙科	100	1
2	王智	90	2
2	赵强	80	3
3	李海	95	1

从表 19-18 可以看出, 只要取出 `rank_rn` 小于或等于 2 的考生即可。以下是完整的查询语句, 在性能方面, 该语句要高于联接方式。

```
WITH StuRank(ClassID, StuName, Achi, rank_rn)
AS
(SELECT ClassID, StuName, Achi,
       DENSE_RANK() OVER(PARTITION BY ClassID ORDER BY Achi DESC)
FROM Students)
SELECT * FROM StuRank WHERE rank_rn <= 2;
```

19.6 取出多列中的非空值

当列中包含空值 (NULL) 时, 由于空值的特殊性, 在进行列计算时会带来一些问题。

19.6.1 姓名问题处理

这是在做一个客户管理系统时遇到的问题。为照顾到一些国外客户, 在设计客户名称列时使用了 3 列, 分别用来存储名字 (First Name)、中间名 (Middle Name) 和姓氏 (Last Name)。将姓氏

和名字分别存储在业务处理上能够更加灵活,例如,在给客户发送电子邮件时可以只获取名字,像 Dear John 这样称呼会使客户感觉更加亲切,而在给客户邮寄东西时,则使用姓名全称能够更加准确地定位客户,这时候则需要将 3 列数据加在一起。现在我们需要讨论的就是如何正确地返回姓名全称。

首先来创建示例表:

```
CREATE TABLE Customers
(
    first_name char(10) NOT NULL,
    middle_name char(10),
    last_name char(10)
);
INSERT INTO Customers
VALUES
    ('Jack', 'A.', 'Tuszynski'),
    ('Sisley', NULL, 'Weilage'),
    ('Eddie', NULL, NULL);
```

如果直接进行简单的相加,就会发现在如表 19-19 所示的查询结果中,只有第 1 行是正确的,第 2 行和第 3 行由于包含了 NULL 值,返回了 NULL。

```
SELECT first_name + ' ' + middle_name + ' ' + last_name AS full_name
FROM Customers;
```

表 19-19

查询结果

full_name		
Jack	A.	Tuszynski
NULL		
NULL		

下面的语句使用 CASE 表达式进行 NULL 值判断,得到了正确的查询结果,如表 19-20 所示。

```
SELECT first_name +
    CASE WHEN (middle_name IS NOT NULL) THEN middle_name
    ELSE ''
    END +
    CASE WHEN (last_name IS NOT NULL) THEN last_name
    ELSE ''
    END AS full_name
FROM Customers;
```

表 19-20

查询结果

full_name		
Jack	A.	Tuszynski
Sisley	Weilage	
Eddie		

还有一种方法就是使用 `COALESCE(expression [,...n])` 函数，它可以返回表达式列表中不为空的第 1 个表达式。下面语句返回与上面相同的结果，语句中的 `COALESCE(middle_name, '')` 包含有 `middle_name` 和空字符串两个表达式，如果 `middle_name` 为空，则返回后面的空字符串，否则直接返回 `middle_name`。

```
SELECT first_name +
       COALESCE(middle_name, '') +
       COALESCE(last_name, '') AS full_name
FROM Customers;
```

下面的语句则是使用 `RTRIM()` 函数去除了每列后面的多余空格。

```
SELECT RTRIM(first_name) + ' ' +
       RTRIM(COALESCE(middle_name, '')) + ' ' +
       RTRIM(COALESCE(last_name, '')) AS full_name
FROM Customers;
```

19.6.2 工资问题处理

这是在做工资计算时遇到的一个问题。在如表 19-21 所示的工资表中，Jack 和 Mary 是按小时计算工资报酬，`hour_sum` 中是累计工作小时，`pay_per_h` 中是每小时的报酬额。而 Joy 和 Nancy 是按天计算工资报酬，`day_sum` 中是累计工作天数，`pay_per_day` 中是每天的报酬额。

表 19-21 工资表 (Salary)

emp_name	hour_sum	pay_per_h	day_sum	pay_per_day
Jack	200	20.00	NULL	NULL
Mary	300	30.00	NULL	NULL
Joy	NULL	NULL	20	120.00
Nancy	NULL	NULL	22	100.00

现在要计算应付给每个人的工资总额，为了能够正确处理 NULL 值，这里仍旧使用 `COALESCE()` 函数，语句如下：

```
SELECT emp_name, COALESCE(hour_sum * pay_per_h, day_sum * pay_per_day) AS pay_all
FROM Salary;
```

返回结果如表 19-22 所示。

表 19-22 工资总额

emp_name	pay_all
Jack	4000.00
Mary	9000.00
Joy	2400.00
Nancy	2200.00

19.7 将数据由行转换为列

9.4 节中已经介绍了使用 PIVOT 运算符进行表行列转换的方法,有效解决了表的数据存储和方便阅读的矛盾问题,本节将提供另一种转换方式。这是在做一个考试系统时遇到的问题,下面是创建示例表的语句。表中存放着学生每次各科的考试成绩。

```
CREATE TABLE exams
(stu_name char(10) NOT NULL,
exam_date date NOT NULL,
exam_sub char(10) NOT NULL,
exam_score int);
INSERT INTO exams
VALUES ('张三', '2009-06-20', '语文', 90),
('张三', '2009-06-20', '数学', 95),
('张三', '2009-06-20', '英语', 100),
('张三', '2009-09-20', '语文', 85),
('张三', '2009-09-20', '数学', 90),
('张三', '2009-09-20', '英语', 98),
('李四', '2009-06-20', '语文', 80),
('李四', '2009-06-20', '数学', 85),
('李四', '2009-06-20', '英语', 90),
('李四', '2009-09-20', '语文', 75),
('李四', '2009-09-20', '数学', 80),
('李四', '2009-09-20', '英语', 88);
```

现在要获得如表 19-23 所示的格式,每个学生按考试日期在表中占一行。

表 19-23

转换后结果

stu_name	exam_date	语 文	数 学	英 语
张三	2009-06-20	90	95	100
张三	2009-09-20	85	90	98
李四	2009-06-20	80	85	90
李四	2009-09-20	75	80	88

在许多情况下,都可以使用 CASE 表达式将表的行转换为列,这是一个非常有用的技巧。参考下面的语句:

```
SELECT stu_name, exam_date,
CASE WHEN exam_sub = '语文' THEN exam_score
ELSE NULL
END AS 语文,
CASE WHEN exam_sub = '数学' THEN exam_score
ELSE NULL
END AS 数学,
CASE WHEN exam_sub = '英语' THEN exam_score
ELSE NULL
END AS 英语
FROM exams;
```

上面语句将得到如表 19-24 所示的结果。

表 19-24

使用 CASE 表达式得到的结果

stu_name	exam_date	语 文	数 学	英 语
张三	2009-06-20	90	NULL	NULL
张三	2009-06-20	NULL	95	NULL
张三	2009-06-20	NULL	NULL	100
张三	2009-09-20	85	NULL	NULL
张三	2009-09-20	NULL	90	NULL
张三	2009-09-20	NULL	NULL	98
李四	2009-06-20	80	NULL	NULL
李四	2009-06-20	NULL	85	NULL
李四	2009-06-20	NULL	NULL	90
李四	2009-09-20	75	NULL	NULL
李四	2009-09-20	NULL	80	NULL
李四	2009-09-20	NULL	NULL	88

由表 19-24 可以看出，只要按 stu_name、exam_date 分组计算最大值，就可以得到如表 19-23 所示的计算结果。参考下面的语句。

```

SELECT stu_name, exam_date,
       MAX(CASE WHEN exam_sub = '语文' THEN exam_score
              ELSE NULL
            END) AS 语文,
       MAX(CASE WHEN exam_sub = '数学' THEN exam_score
              ELSE NULL
            END) AS 数学,
       MAX(CASE WHEN exam_sub = '英语' THEN exam_score
              ELSE NULL
            END) AS 英语
FROM exams
GROUP BY stu_name, exam_date
ORDER BY stu_name, exam_date;

```