

Broadview®
www.broadview.com.cn

以任务驱动方式讲解，用实例引导读者学习
只需21天，便可轻松掌握Oracle数据库

畅销书
新品

21天学通 Oracle

张朝明 等编著

14
小时多媒体
语音视频教学

DVD

本书特色

- 基础知识→核心技术→典型实例→综合练习→项目案例
- 313个典型实例、2个项目案例、94个练习题
- 一线开发人员全程贴心讲解，上手毫不费力

超值DVD

- 14小时多媒体语音视频教学
- 本书源代码 + 本书电子教案 (PPT)
- 1000余页编程参考宝典电子书 (免费赠送)



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

21天学通系列

本书涵盖主题

- Oracle安装配置
- SQL Plus和PL/SQL
- Oracle数据表
- 视图
- 游标
- 序列
- Oracle数据类型
- Oracle中的控制语句
- SQL更新数据
- 并发控制
- Oracle在Java开发中的应用
- Oracle常用工具
- Oracle数据库
- 约束
- 函数与存储过程
- 触发器
- 用户角色与权限控制
- Oracle中的函数与表达式
- SQL查询
- 数据库速度优化与数据完整性
- Oracle中的正则表达式
- Oracle在C#开发中的应用

精彩内容，尽在21天学编程

多媒体
语音视频教学

DVD



责任编辑：高洪霞
责任美编：李玲



本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

上架建议：Oracle

ISBN 978-7-121-10626-2



9 787121 106262 >

定价：49.80元(含DVD光盘1张)

本书特点

千里之行，始于足下！

——老子

为什么要写这样一本书

Oracle 数据库是目前全球功能最强大、应用最广泛的数据库。在一般的 Oracle 图书中，要么只包含针对 Oracle 数据库管理员 DBA 进行管理配置方面的知识，要么只针对软件程序员讲解 SQL 语法方面的知识。这使得很多 Oracle 数据库的管理者和开发者很难对 Oracle 有全面的了解。因此，笔者在本书中讲述了 Oracle 数据库对象的管理和配置、SQL 开发，以及与开发语言相结合三大方面的内容，以使管理者和开发者都能够对 Oracle 数据库有更全面的了解。

为了能让大多数读者都能够快速而轻松地掌握 Oracle 数据库各个方面的知识，笔者尝试从初学者的角度进行思考，并结合实际开发经验，编写了这本 Oracle 快速入门和提高的教程。在本书中，笔者首先介绍 Oracle 中主要的数据库对象，以及如何使用这些对象，并尝试从数据库中搜寻有关这些对象的详细信息，让读者更加清晰地认识 Oracle 数据库，接着针对开发者讲述 SQL 语句的详细用法，并提供了高级内容——索引、锁定等，以使读者在最短的时间内，提升对 Oracle 数据库的认识。本书采用了目前应用最为广泛的 Oracle 10 版本进行讲解。

本书有何特色

1. 细致体贴的讲解

为了让读者更快地上手，本书特别设计了适合初学者的学习方式，用准确的语言总结概念 ➡ 用直观的图示演示过程 ➡ 用详细的注释解释代码 ➡ 用形象的比方帮助记忆。效果如下：

① **知识点介绍** 准确、清晰是其显著特点，一般放在每一节开始位置，让零基础的读者了解相关概念，顺利入门。

② **范例** 书中出现的完整实例，以章节顺序编号，便于检索和循序渐进地学习、实践，放在每节知识点介绍之后。

③ **范例代码** 与范例编号对应，层次清楚、语句简洁、注释丰富，体现了代码优美的原则，有利于读者养成良好的代码编写习惯。对于大段程序，均在每行代码前设定编号，便于学习。

④ **运行结果** 对范例给出运行结果和对应图示，帮助读者更直观地理解范例代码。

⑤ **代码解析** 将范例代码中的关键代码行逐一解释，有助于读者掌握相关概念和知识。

⑥ **综合练习** 为了便于读者巩固所学内容，本书每章中均提供了综合练习，并给出了操作提示和结果，配合读者自己动手实践。

⑦ **习题** 每章最后提供专门的测试习题，供读者检验所学知识是否牢固掌握，题目的提示或答案放在光盘中。



⑧ 贴心的提示 为了便于读者阅读，全书还穿插着一些技巧、提示等小贴士，体例约定如下：

- 提示：通常是一些贴心的提醒，让读者加深印象或提供建议，或者解决问题的方法。
- 注意：提出学习过程中需要特别注意的一些知识点和内容，或者相关信息。
- 警告：对操作不当或理解偏差将会造成的灾难性后果做警示，以加深读者印象。

经作者多年的培训和授课证明，以上讲解方式是最适合初学者学习的方式，读者按照这种方式，会非常轻松、顺利地掌握本书知识。

2. 实用超值的 DVD 光盘

为了帮助读者比较直观地学习，本书附赠 DVD 光盘，内容包括多媒体视频、电子教案（PPT）、编程参考宝典电子书、各章习题答案和实例源代码等。

● 多媒体视频

配有长达 14 小时手把手教学视频，讲解关键知识点界面操作和书中的一些综合练习题。作者亲自配音、演示，手把手教会读者使用。

- 21天学通Oracle光盘
- PHEI Broadview 2008专业书目 第一期
- pheibook
- 编程参考宝典电子书
- 电子教案 (PPT)
- 多媒体视频
- 习题答案
- 源代码



● 电子教案 (PPT)

本书可以作为高校相关课程的教材或课外辅导书, 所以笔者特别为本书制作了电子教案(PPT), 以方便老师教学使用。

● 编程参考宝典电子书

为方便广大读者学习, 特别制作了编程开发参考电子书, 供读者查阅和参考。



3. 提供完善的技术支持

本书提供了论坛: <http://www.rzchina.net>, 读者可以在上面提问交流。另外, 论坛上还有一些小的教程、视频动画和各种技术文章, 可帮助读者提高开发水平。

4. 丰富的额外素材下载

相关的开发素材文件在 www.broadview.com.cn 提供下载。

推荐的学习计划

本书从初学者角度出发, 结合对学习阶段的认识, 归纳了最适合初学者的学习模式, 并为读者总结了合理的学习时间分配方式, 列表如下:

推荐时间安排	自学目标 (框内打钩表示已掌握)	难度指数
第1周	了解 Oracle 数据库	<input type="checkbox"/>
	安装 Oracle 数据库	<input type="checkbox"/>
	尝试使用 Oracle 数据库的各种客户端工具	<input type="checkbox"/>
	尝试使用 PL/SQL Developer 或者 TOAD	<input type="checkbox"/>
	利用 SQL Plus 编写简单的代码块	<input type="checkbox"/>
	了解 Oracle 数据库、表空间、数据表之间的关系	<input type="checkbox"/>
	掌握创建 Oracle 数据库	<input type="checkbox"/>
	掌握备份/恢复 Oracle 数据库	<input type="checkbox"/>
	掌握创建表空间	<input type="checkbox"/>
第2天	掌握创建、删除数据表	<input type="checkbox"/>
	掌握修改数据表结构	<input type="checkbox"/>
	掌握特殊的数据表 dual 和临时表的使用	<input type="checkbox"/>
第3天	了解和掌握主键约束的使用	<input type="checkbox"/>
	了解和掌握外键约束的使用	<input type="checkbox"/>

续表

推荐时间安排		自学目标（框内打钩表示已掌握）	难度指数
第1周	第3天	了解和掌握惟一性约束的使用 <input type="checkbox"/> 了解和掌握检查约束的使用 <input type="checkbox"/> 了解和掌握默认值约束的使用 <input type="checkbox"/>	★★★
	第4天	了解和掌握关系视图的使用 <input type="checkbox"/> 了解和掌握内嵌视图的使用 <input type="checkbox"/> 了解和掌握对象视图的使用 <input type="checkbox"/> 了解和掌握物化视图的使用 <input type="checkbox"/>	★★★
	第5天	了解和掌握函数的编写和使用 <input type="checkbox"/> 了解和掌握存储的编写和使用 <input type="checkbox"/> 了解和掌握程序包的编写使用 <input type="checkbox"/>	★★★★
	第6天	了解游标的基本概念 <input type="checkbox"/> 掌握静态（显式和隐式）游标的使用 <input type="checkbox"/> 掌握动态游标的使用 <input type="checkbox"/>	★★★★
	第7天	了解触发器的概念 <input type="checkbox"/> 掌握如何创建和使用语句触发器 <input type="checkbox"/> 掌握如何创建和使用行触发器 <input type="checkbox"/> 掌握如何创建和使用 instead of 触发器 <input type="checkbox"/> 掌握如何创建和使用系统触发器 <input type="checkbox"/> 理解各种触发器之间的区别 <input type="checkbox"/>	★★★★
第2周	第8天	了解序列的基本概念 <input type="checkbox"/> 掌握序列各属性的修改，以及各属性的用途 <input type="checkbox"/> 了解序列在开发中的实际应用场景 <input type="checkbox"/>	★★
	第9天	了解用户、权限与角色的概念 <input type="checkbox"/> 掌握如何创建和使用用户、权限与角色 <input type="checkbox"/> 掌握如何高效地为用户分配权限 <input type="checkbox"/>	★★★
	第10天	了解 Oracle 中基本的数据类型 <input type="checkbox"/> 掌握开发中常用的数据类型 <input type="checkbox"/> 掌握 Oracle 中特殊的数据类型 <input type="checkbox"/>	★★
	第11天	掌握 Oracle 中的各种内置函数 <input type="checkbox"/> 掌握 Oracle 中的基本运算 <input type="checkbox"/> 掌握 Oracle 中的特殊判式 <input type="checkbox"/>	★★★
	第12天	了解分析函数与窗口函数的提出意义 <input type="checkbox"/> 了解分析函数与窗口函数的工作原理 <input type="checkbox"/> 掌握常用分析函数 <input type="checkbox"/> 掌握窗口函数中各种子句的使用 <input type="checkbox"/>	★★★★
	第13天	了解 PL/SQL 控制语句的基本种类 <input type="checkbox"/> 掌握条件控制语句 <input type="checkbox"/> 掌握各种循环语句 <input type="checkbox"/>	★★

续表

推荐时间安排		自学目标（框内打钩表示已掌握）	难度指数
第2周	第14天	掌握 Oracle 的基本查询语法 <input type="checkbox"/>	★★★
		掌握子查询的使用 <input type="checkbox"/>	
		掌握联合查询的使用 <input type="checkbox"/>	
		掌握联接语句的使用 <input type="checkbox"/>	
第3周	第15天	理解 Oracle 的层次化查询的意义 <input type="checkbox"/>	★★★★
		掌握层次化查询的使用 <input type="checkbox"/>	
		掌握层次化查询的相关函数的使用 <input type="checkbox"/>	
	第16天	掌握插入数据的 SQL 语句的使用 <input type="checkbox"/>	★★
		掌握更新数据的 SQL 语句的使用 <input type="checkbox"/>	
		掌握删除数据的 SQL 语句的使用 <input type="checkbox"/>	
		掌握提交和回滚的基本动作 <input type="checkbox"/>	
	第17天	了解索引提高数据库性能的原理 <input type="checkbox"/>	★★★★
		掌握索引的使用 <input type="checkbox"/>	
		了解数据完整性的重要意义 <input type="checkbox"/>	
		掌握利用各种约束保持数据完整性 <input type="checkbox"/>	
	第18天	了解数据一致性的重要意义 <input type="checkbox"/>	★★★★
		掌握利用事务保持数据一致性 <input type="checkbox"/>	
		掌握 Oracle 中事务的属性和隔离级别 <input type="checkbox"/>	
		掌握事务的处理原则 <input type="checkbox"/>	
	第19天	了解并发与锁定的意义 <input type="checkbox"/>	★★★★
		了解 Oracle 中锁定转换的意义 <input type="checkbox"/>	
		掌握悲观锁与乐观锁的使用 <input type="checkbox"/>	
		了解处理数据库并发的其他方法 <input type="checkbox"/>	
	第20天	了解正则表达式的意义 <input type="checkbox"/>	★★★
		掌握正则表达式的一般语法 <input type="checkbox"/>	
		掌握 Oracle 中正则表达式的特殊用法 <input type="checkbox"/>	
		掌握 Oracle 中正则表达式的相关函数 <input type="checkbox"/>	
	第21天	了解 Java 中使用 Oracle 数据库的一般模式 <input type="checkbox"/>	★★★
		掌握如何利用 JDBC 和 Hibernate 操作 Oracle 数据库 <input type="checkbox"/>	
		了解 C#中使用 Oracle 数据库的一般模式 <input type="checkbox"/>	
		掌握如何在 C#中操作 Oracle 数据库 <input type="checkbox"/>	

本书适合哪些读者阅读

本书非常适合以下人员阅读：

- Oracle 数据库开发人员；
- Oracle 数据库管理员；

- 基于 Oracle 数据库的软件程序员；
- 大专院校学生；
- 其他编程爱好者。




本书作者

本书主要由张朝明编写，其他参与编写和资料整理的人员有昊燃、方振宇、陈冠佐、傅奎、陈勤、梁洋洋、毕梦飞、陈庆、柴相花、陈非凡、陈华、陈嵩、承卓、陈先在。





目 录


第一篇 Oracle 管理配置



第 1 章 Oracle 安装配置 ( 教学视频: 10 分钟)	23
1.1 Oracle 简介	23
1.1.1 数据库术语	23
1.1.2 主流数据库简介	24
1.1.3 Oracle 数据库的特点	24
1.2 安装 Oracle 数据库	25
1.2.1 Oracle 数据库的版本变迁及安装环境	25
1.2.2 安装过程	26
1.2.3 安装中需要注意的问题	27
1.3 本章小结	28
1.4 习题	28
第 2 章 Oracle 常用工具 ( 教学视频: 7 分钟)	29
2.1 Net Configuration Assistant (网络配置助手)	29
2.1.1 监听程序配置	29
2.1.2 命名方法配置	31
2.1.3 本地 Net 服务名配置	32
2.2 Net Manager (网络管理员)	34
2.3 本章实例	36
2.4 本章小结	38
2.5 习题	38
第 3 章 SQL Plus 和 PL/SQL ( 教学视频: 11 分钟)	39
3.1 SQL Plus 与 PL/SQL 简介	39
3.2 使用 SQL Plus	40
3.2.1 登录 SQL Plus	40
3.2.2 SQL Plus 输出结果的格式化	41
3.2.3 SQL Plus 小结	46
3.3 PL/SQL	46
3.3.1 PL/SQL 常用开发工具	46
3.3.2 开发一个简单的 PL/SQL 程序	48
3.4 本章实例	49


3.5 本章小结.....	50
3.6 习题	50

第二篇 Oracle 数据库对象

第 4 章 Oracle 数据库 ( 教学视频: 15 分钟)	51
4.1 创建 Oracle 数据库	51
4.2 Oracle 数据库的相关术语	52
4.2.1 数据库	53
4.2.2 数据库实例和 SID	53
4.2.3 ORACLE_SID	54
4.3 Oracle 数据库的备份与恢复	55
4.3.1 逻辑备份/恢复 (导出/导入)	55
4.3.2 物理备份/恢复	56
4.3.3 利用 PL/SQL Developer 备份数据库	60
4.4 本章实例	61
4.5 本章小结	61
4.6 习题	62
第 5 章 Oracle 数据表对象 ( 教学视频: 42 分钟)	63
5.1 Oracle 表空间	63
5.1.1 Oracle 表空间简介	63
5.1.2 创建 Oracle 表空间	64
5.1.3 查看表空间	66
5.1.4 修改数据库默认表空间	67
5.1.5 修改表空间名称	68
5.1.6 删除表空间	69
5.2 创建 Oracle 数据表	70
5.2.1 利用工具创建数据表	70
5.2.2 利用工具查看数据表	71
5.2.3 利用命令创建数据表	72
5.2.4 利用命令查看表结构	72
5.3 修改 Oracle 数据表结构	73
5.3.1 利用工具修改数据表结构	73
5.3.2 利用命令修改数据表结构	74
5.4 删除数据表	75
5.4.1 利用工具删除数据表	76
5.4.2 利用 SQL 语句删除数据表	76
5.5 备份/恢复数据表	76

5.5.1	利用工具备份/恢复数据表	77
5.5.2	利用命令备份/恢复数据表	82
5.6	临时表	83
5.6.1	临时表简介	83
5.6.2	会话级临时表	84
5.6.3	事务级临时表	85
5.6.4	查看临时表在数据库中的信息	86
5.6.5	临时表的应用场景	86
5.7	特殊的表 dual	87
5.7.1	分析 dual 表	87
5.7.2	dual 表的应用场景	87
5.7.3	修改 dual 表对查询结果的影响	88
5.8	本章实例	89
5.9	本章小结	90
5.10	习题	90
第 6 章	约束 ( 教学视频: 43 分钟)	91
6.1	主键约束	91
6.1.1	主键简介	91
6.1.2	创建主键约束	92
6.1.3	修改表的主键约束	94
6.1.4	主键应用场景	96
6.2	外键约束	97
6.2.1	外键简介	97
6.2.2	创建外键约束	97
6.2.3	级联更新与级联删除	100
6.2.4	修改外键属性	102
6.2.5	外键使用	104
6.3	唯一性约束	105
6.3.1	唯一性约束简介	105
6.3.2	创建唯一性约束	105
6.3.3	修改唯一性约束	107
6.3.4	唯一性约束的使用	108
6.4	检查约束	108
6.4.1	检查约束简介	108
6.4.2	创建检查约束	108
6.4.3	修改检查约束	110
6.4.4	检查约束的使用	111

6.5 默认值约束	111
6.5.1 默认值约束简介	112
6.5.2 创建默认值约束	112
6.5.3 修改默认值约束	113
6.6 本章实例	115
6.7 本章小结	116
6.8 习题	116
第7章 视图 ( 教学视频: 50 分钟)	117
7.1 关系视图	117
7.1.1 建立关系视图	117
7.1.2 修改/删除视图	118
7.1.3 联接视图	120
7.1.4 编译视图	122
7.1.5 使用 force 选项强制创建视图	124
7.1.6 利用视图更新数据表	125
7.1.7 with check option 选项	126
7.1.8 关系视图小结	128
7.2 内嵌视图	128
7.2.1 内嵌视图简介	128
7.2.2 内嵌视图的使用	128
7.2.3 内嵌视图小结	130
7.3 对象视图	131
7.3.1 对象视图简介	131
7.3.2 对象视图简介	131
7.4 物化视图	133
7.4.1 物化视图简介	133
7.4.2 物化视图的使用	133
7.4.3 物化视图的数据加载	135
7.4.4 物化视图的数据更新	135
7.4.5 查询重写	136
7.5 本章小结	136
7.6 本章实例	137
7.7 习题	137
第8章 函数与存储过程 ( 教学视频: 48 分钟)	138
8.1 函数	138
8.1.1 函数简介	138

8.1.2	创建函数	139
8.1.3	函数中的括号	140
8.1.4	函数的参数	141
8.1.5	函数的确定性	142
8.1.6	典型函数举例	143
8.2	存储过程	144
8.2.1	存储过程简介	144
8.2.2	创建存储过程	144
8.2.3	存储过程的参数——IN 参数	146
8.2.4	存储过程的参数——OUT 参数	147
8.2.5	存储过程的参数——IN OUT 参数	149
8.2.6	存储过程的参数——参数顺序	149
8.2.7	存储过程的参数——参数的默认值	152
8.2.8	存储过程的参数——参数顺序总结	153
8.3	程序包	153
8.3.1	规范	153
8.3.2	主体	155
8.3.3	调用程序包中的函数/存储过程	157
8.3.4	程序包中的变量	158
8.4	本章实例	159
8.5	本章小结	161
8.6	习题	161
第9章	游标 ( 教学视频: 36 分钟)	162
9.1	游标简介	162
9.2	显式游标	162
9.2.1	声明游标	162
9.2.2	使用游标	164
9.3	隐式游标	166
9.3.1	sql 隐式游标	166
9.3.2	cursor for 游标	168
9.3.3	隐式游标和显式游标	169
9.4	游标属性	169
9.5	动态游标	171
9.5.1	强类型动态游标	171
9.5.2	弱类型动态游标	173
9.5.3	比较两种动态游标	176
9.6	本章实例	176


9.7 本章小结	176
9.8 习题	177
第 10 章 触发器 (教学视频: 58 分钟)	178
10.1 触发器简介	178
10.2 创建和使用触发器	178
10.2.1 创建触发器	179
10.2.2 触发器的作用级别	180
10.2.3 在多个事件上定义触发器	181
10.2.4 为同一事件定义多个触发器	182
10.2.5 触发器限制	183
10.3 语句触发器	184
10.3.1 创建语句触发器	184
10.3.2 触发器谓词	186
10.3.3 触发时机	187
10.3.4 触发器级联	188
10.4 行触发器	189
10.4.1 行触发器与引用	189
10.4.2 触发时机与引用	191
10.4.3 触发时机与瞬态	193
10.5 instead of 触发器	195
10.5.1 创建和使用 instead of 触发器	195
10.5.2 instead of 触发器与引用	198
10.6 系统事件与用户事件触发器	198
10.6.1 系统事件触发器	198
10.6.2 用户事件触发器	200
10.7 启用和禁用触发器	201
10.7.1 启用和禁用触发器的场景	201
10.7.2 禁用触发器	202
10.7.3 启用触发器	202
10.7.4 触发器信息	203
10.8 本章实例	203
10.9 本章小结	205
10.10 习题	205
第 11 章 序列 (教学视频: 28 分钟)	206
11.1 创建和使用序列	206
11.1.1 创建序列	206

11.1.2	使用序列	207
11.1.3	序列初始值 start with	208
11.2	修改序列属性	209
11.2.1	修改 minvalue 和 maxvalue	209
11.2.2	修改 increment by	210
11.2.3	修改 cycle	211
11.2.4	修改 cache	212
11.3	本章实例	213
11.4	本章小结	214
11.5	习题	214
第 12 章	用户角色与权限控制 (教学视频: 45 分钟)	215
12.1	用户	215
12.1.1	Oracle 中的用户	215
12.1.2	创建新用户	216
12.1.3	用户与模式 (Schema)	216
12.1.4	系统用户 sys 和 system	217
12.2	权限	218
12.2.1	系统权限	218
12.2.2	对象权限	221
12.3	角色	225
12.3.1	创建和使用角色	225
12.3.2	继承角色	226
12.3.3	禁用和启用角色	228
12.4	本章实例	229
12.5	本章小结	230
12.6	习题	230

第三篇 Oracle 中的 SQL

第 13 章	Oracle 数据类型 (教学视频: 21 分钟)	231
13.1	Oracle 中的数据类型	231
13.1.1	字符型	231
13.1.2	数值型	232
13.1.3	日期时间型	232
13.1.4	lob 类型	233
13.2	Oracle 中的特殊数据	233
13.2.1	rowid	233
13.2.2	null 与空字符串	234


13.2.3 单引号与双引号	235
13.3 本章实例	237
13.4 本章小结	238
13.5 习题	239
第 14 章 Oracle 中的函数与表达式 (教学视频: 111 分钟)	240
14.1 Oracle 中的字符串函数	240
14.1.1 lpad()函数	240
14.1.2 rpad()函数	241
14.1.3 lower()函数——返回小写字符串	241
14.1.4 upper()函数——返回大写字符串	241
14.1.5 initcap()函数——单词首字母大写	242
14.1.6 length()函数——返回字符串长度	242
14.1.7 substr()函数——截取字符串	243
14.1.8 instr()函数——获得字符串出现的位置	243
14.1.9 ltrim()函数——删除字符串首部空格	244
14.1.10 rtrim()函数——删除字符串尾部空格	244
14.1.11 trim()函数——删除字符串首尾空格	245
14.1.12 to_char()函数——将其他类型转换为字符类型	245
14.1.13 chr()函数——将 ascii 码转换为字符串	247
14.1.14 translate()函数——替换字符	247
14.2 Oracle 中的数学函数	248
14.2.1 abs()函数——返回数字的绝对值	248
14.2.2 round()函数——返回数字的“四舍五入”值	248
14.2.3 ceil()函数——向上取整	249
14.2.4 floor()函数——向下取整	249
14.2.5 mod()函数——取模操作	250
14.2.6 sign()函数——返回数字的正负性	250
14.2.7 sqrt()函数——返回数字的平方根	251
14.2.8 power()函数——乘方运算	251
14.2.9 trunc()函数——截取数字	251
14.2.10 vsize()函数——返回数据的存储空间	252
14.2.11 to_number()函数——将字符串转换为数值类型	252
14.3 Oracle 中的日期函数	252
14.3.1 to_date()函数——将字符串转换为日期型	252
14.3.2 add_months()函数——为日期加上特定月份	253
14.3.3 last_day()函数——返回特定日期所在月的最后一天	253
14.3.4 months_between()函数——返回两个日期所差的月数	253




14.3.5	current_date()函数——返回当前会话时区的当前日期	254
14.3.6	current_timestamp()函数——返回当前会话时区的当前时间戳	254
14.3.7	extract()函数——返回日期的某个域	254
14.4	Oracle 中的聚合函数	255
14.4.1	max()函数——求最大值	256
14.4.2	min()函数——求最小值	257
14.4.3	avg()函数——求平均值	257
14.4.4	sum()函数——求和	258
14.4.5	count()函数——获得记录数	258
14.5	Oracle 中的其他函数	259
14.5.1	decode()函数——多值判断	259
14.5.2	nvl()函数——处理空值	260
14.5.3	cast()函数——强制转换数据类型	261
14.6	Oracle 中的运算表达式	263
14.6.1	数学运算	263
14.6.2	逻辑运算	264
14.6.3	位运算	265
14.7	Oracle 中的特殊判式	266
14.7.1	between——范围测试	266
14.7.2	in——集合成员测试	267
14.7.3	like——模式匹配	267
14.7.4	is null——空值判断	268
14.7.5	exists——存在性判断	268
14.7.6	all, some, any——数量判断	269
14.8	Oracle 高级函数——分析函数与窗口函数	270
14.8.1	排名	270
14.8.2	分区窗口	272
14.8.3	窗口子句	273
14.8.4	主要的分析函数	277
14.9	本章实例	280
14.10	本章小结	281
14.11	习题	281
第 15 章	Oracle 中的控制语句 ( 教学视频: 16 分钟)	282
15.1	Oracle 中的条件语句	282
15.1.1	利用 if else 进行条件判断	282
15.1.2	利用 case when 进行分支判断	283
15.2	Oracle 中的循环语句	285

15.2.1	无条件循环	285
15.2.2	while 循环	286
15.2.3	for 循环	287
15.3	本章实例	288
15.4	本章小结	289
15.5	习题	289
第 16 章 SQL 查询 (教学视频: 55 分钟)		290
16.1	基本查询	290
16.1.1	select 语句查询执行步骤	290
16.1.2	where 子句	292
16.1.3	利用 distinct 获得唯一性记录	293
16.1.4	order by 子句	293
16.1.5	group by 子句	294
16.1.6	having 子句	297
16.2	子查询	298
16.2.1	理解子查询	298
16.2.2	子查询使用实例	298
16.3	联合语句	300
16.3.1	union 查询	300
16.3.2	union all 查询	301
16.3.3	intersect 查询	303
16.3.4	minus 查询	303
16.4	联接	304
16.4.1	自然联接	304
16.4.2	内联接	305
16.4.3	外联接——左联接	306
16.4.4	外联接——右联接	308
16.4.5	外联接——完全联接	310
16.5	层次化查询	312
16.5.1	利用 connect by 进行层次化查询	313
16.5.2	connect by 的使用场景	315
16.5.3	sys_connect_by_path()函数的使用	316
16.6	本章实例	316
16.7	本章小结	318
16.8	习题	318
第 17 章 SQL 更新数据 (教学视频: 34 分钟)		319
17.1	插入数据	319

17.1.1	insert 语句向表中插入数据	319
17.1.2	利用子查询批量插入数据	320
17.1.3	insert 语句与默认值	321
17.1.4	insert 语句与唯一性约束	321
17.1.5	insert 语句与外键约束	321
17.2	修改数据	322
17.2.1	利用 update 修改单列的值	322
17.2.2	利用 update 修改多列的值	323
17.2.3	利用 where 子句限制修改范围	323
17.3	删除数据	324
17.3.1	用 delete 命令删除数据	324
17.3.2	用 truncate 命令删除数据	325
17.4	数据提交与回滚	325
17.4.1	回滚动作	325
17.4.2	提交动作	326
17.4.3	PL/SQL Developer 中的回滚与提交	327
17.5	本章实例	329
17.6	本章小结	331
17.7	习题	331

第四篇 Oracle 编程高级应用

第 18 章	数据库速度优化与数据完整性 ( 教学视频: 32 分钟)	332
18.1	利用索引加快数据引用	332
18.1.1	索引的原理	332
18.1.2	利用索引提高数据库性能	333
18.1.3	索引对 DML 的影响	335
18.1.4	索引的使用时机	336
18.2	利用约束保持数据完整性	337
18.2.1	数据库完整性的重要性	337
18.2.2	保持数据库完整性的重要方面	337
18.2.3	利用约束保持数据完整性	337
18.3	本章实例	338
18.3.1	使用比较运算符不当	339
18.3.2	函数的使用	339
18.3.3	联合索引	339
18.4	本章小结	340
18.5	习题	340

第 19 章 数据一致性与事务管理 ( 教学视频: 46 分钟)	341
19.1 什么是数据一致性和事务	341
19.1.1 数据一致性	341
19.1.2 事务	341
19.2 Oracle 中的事务处理	342
19.2.1 commit 命令	342
19.2.2 roll back 命令	343
19.2.3 savepoint 和 roll back to savepoint 命令	343
19.2.4 事务的属性和隔离级别	345
19.3 事务处理原则	349
19.3.1 原子性	349
19.3.2 一致性	350
19.3.3 隔离性	352
19.3.4 持久性	353
19.4 本章实例	353
19.5 本章小结	355
19.6 习题	355
第 20 章 并发控制 ( 教学视频: 35 分钟)	356
20.1 并发与锁定	356
20.2 数据锁定	359
20.2.1 悲观锁定	359
20.2.2 乐观锁定	361
20.2.3 悲观锁定与乐观锁定的比较	362
20.2.4 锁定转换	363
20.3 并发控制的其他方法	363
20.4 本章实例	364
20.5 本章小结	368
20.6 习题	368
第 21 章 Oracle 中的正则表达式 ( 教学视频: 29 分钟)	369
21.1 正则表达式简介	369
21.1.1 正则表达式与通配符	369
21.1.2 正则表达式与编程语言	369
21.2 正则表达式基础知识	370
21.2.1 元字符和普通字符	370
21.2.2 量词	370
21.2.3 字符转义与字符类	370

21.2.4	字符组的使用	371
21.2.5	正则表达式分支	371
21.2.6	Oracle 中正则表达式的特殊性	371
21.3	正则表达式在 Oracle 中的应用	372
21.3.1	regexp_like() 的使用	372
21.3.2	regexp_instr() 的使用	372
21.3.3	regexp_substr() 的使用	373
21.3.4	regexp_replace() 的使用	373
21.4	本章实例	374
21.5	本章小结	374
21.6	习题	375

第五篇 Oracle 与编程语言综合使用实例

第 22 章	Oracle 在 Java 开发中的应用 (教学视频: 38 分钟)	376
22.1	通过 JDBC 使用 Oracle	376
22.1.1	JDBC 简介	376
22.1.2	准备工作	376
22.1.3	JDBC 连接 Oracle	377
22.1.4	利用 JDBC 查询数据	379
22.1.5	利用 JDBC 更新数据	380
22.1.6	总结 JDBC 操作数据库	381
22.2	通过 Hibernate 操作 Oracle 数据库	382
22.2.1	准备工作	382
22.2.2	配置 Hibernate	382
22.2.3	利用 Hibernate 查询数据	386
22.2.4	利用 Hibernate 更新数据	387
22.2.5	利用 Hibernate 插入数据	388
22.3	本章小结	390
22.4	习题	390
第 23 章	Oracle 在 C# 开发中的应用 (教学视频: 12 分钟)	391
23.1	在 C# 中连接 Oracle 数据库	391
23.2	在 C# 中操作 Oracle 数据库	392
23.3	在 C# 中使用 Oracle 数据库事务	394
23.4	本章小结	397
23.5	习题	397

第一篇 Oracle 管理配置

第 1 章 Oracle 安装配置

Oracle 是目前全球最流行、最强大的数据库系统。Oracle 数据库具有完备的数据管理功能，能完美地刻画数据关系，并实现了完善的分布式处理功能。由于 Oracle 强大的功能，针对其安装配置也具有一定的复杂性。本章着重讲述 Oracle 数据库的安装及配置。本章的内容包括：

- 数据库术语；
- 主流数据库简介；
- Oracle 数据库的安装环境；
- 安装 Oracle 数据库服务器。

通过本章的学习，读者能够掌握基本的数据库术语，并能了解当前主流数据库的特点。对于 Oracle 数据库服务器端，则可以独立完成在 Windows 环境中的安装。



1.1 Oracle 简介

数据库 (database) 是数据存储仓库的简称。数据库是一个经久不衰的话题，本节将首先介绍数据库的基本术语，接着介绍当前主流数据库，最后介绍 Oracle 相对其他数据库的特点。

1.1.1 数据库术语

在介绍数据库的配置和开发之前，了解数据库的基本术语是必要的。这些术语并非仅仅用于 Oracle 或其他特定数据库，而是作为一种标准称谓在各数据库中共享使用。

1. 数据

数据是数据库的最基本的存储对象。文本、图像、声音、视频等媒体格式在存储于数据库时，都被称为数据。数据是数据库建立的根本目的。

2. 数据库及数据库管理系统

数据库是数据存储的仓库。数据库都是建立在计算机设备上的，最常见的设备为计算机硬盘。数据库以文件的形式存在，而文件的具体格式则由各数据库厂商进行定义。

数据库管理系统是用于管理数据库的工具。因为所有的数据都是以某种格式存储在文件中的，用户不可能直接操作文件来实现对数据库的操作。这样非但具有相当大的安全隐患，而且不具有可行性。因此，各数据库厂商都会提供本身的工具（一般为图形界面软件）作为用户接口。数据库用户通过这些工具进行各种数据库操作。常见的数据库管理系统如 Oracle 的 OEM (Oracle Enterprise Manager)、SQL Server 的企业管理等。

3. 关系型数据库

关系型数据库实际指代了一种数据库模型。将某些相关数据存储于同一个表，表与表之间利用相互关系进行关联。例如，表示员工信息的员工工号、员工姓名、员工年龄等信息存储在员工表中，而表示员工的工资、奖金等存储在工资表中。二者往往利用员工工号作为联络的组



带。关系型数据库使用简单、各表中的数据相互独立，而又可以进行联系，是目前主流的关系模型。

4. 常见的数据库对象

数据库对象是数据库中用于划分各种数据和实现各种功能的单元。数据库用户往往利用数据库对象来实现对数据库的操作。

- 用户：用户是创建在数据库中的账号。通过这些账号来登录数据库，并实现对不同使用者权限的控制。
- 表：表是最常见的数据库对象。与现实世界中的表具有相同的结构——每个表都由行组成，各行由列组成。例如，在员工表中，每位员工的信息均可看做行，而员工的姓名、年龄则作为列。
- 索引：索引是根据指定的数据库表列建立起来的顺序，对于每一行数据都会建立快速访问的路径，因此，可以大大提高数据访问的效率。
- 视图：视图可以看做虚拟的表。视图并不存储数据，而是作为数据的镜像。
- 函数：数据库中的函数与其他编程语言中的函数类似，都是用来按照规则提供返回值的流程代码。
- 存储过程：数据库中的存储过程类似于其他编程语言中的过程。不过，存储过程还具有自身的特点，例如，具有输入参数和输出参数等。
- 触发器：触发器的作用类似于监视器。触发器的本质也是执行特定任务的代码块。当数据库监控到某个事件时，会激活建立在该事件上的触发器，并执行触发器代码。

1.1.2 主流数据库简介

当前数据库市场，主流的数据库包括：Oracle、Sybase、DB2、SQL Server、My SQL。

- Oracle：开发商为美国的甲骨文公司（Oracle）。Oracle 数据库是以高级结构化查询语言（SQL）为基础的大型关系数据库，是目前最流行、应用最广泛的客户端/服务器（Client/Server）体系结构的数据库。
- Sybase：开发商为 Sybase 公司。Sybase 数据库性能较高，安全性极高，可运行于 UNIX、Windows 及 Novell Netware 环境。该数据库不但具有优越的性能，而且具备跨平台能力。
- DB2：开发商为 IBM。DB2 数据库支持各种机型及操作系统环境。支持面向对象编程，并有强大的开发和管理工具。
- SQL Server：开发商为微软公司。SQL Server 在性能及安全性上不及以上三种数据库，但是其占用系统资源较少，操作简单、灵活。
- My SQL：由原 My SQL 公司开发。My SQL 数据库使用简单、操作方便，性能也较高。难能可贵的是，My SQL 是开源数据库，而且完全免费，这也成为其迅速崛起的主要原因之一。

1.1.3 Oracle 数据库的特点

相较于其他数据库，Oracle 具有以下特点。

- 毫无疑问，优越的性能是 Oracle 战胜其他数据库的首要法宝。Oracle 优越的性能使得其成为大型应用和超大型系统的首选数据库，而且甲骨文公司从未停止过在这方面的进步。



- 提供了基于角色的权限管理模式。通过角色管理，大大加强了数据库的安全性，同时，也为 DBA 提供了更加方便、快捷的管理用户和权限的途径。
- 可良好地支持大数据存储格式，如图形、音频、视频、动画等媒体格式。
- 提供了良好的分布式管理功能，用户可以很轻松地实现多数据库的协调工作。
- 提出了独创性的表空间理念。在数据模型方面，Oracle 有着区别于其他数据库的表空间概念。使数据在逻辑上划分得更加清晰，而且具有更大的灵活性。



1.2 安装 Oracle 数据库

在简单了解了数据库的基本知识之后，本节将讲述 Oracle 数据库的安装过程。目前，Oracle 广泛使用的版本为 Oracle 10g，本书的所有内容均以 Oracle 10g 数据库作为基础展开。

1.2.1 Oracle 数据库的版本变迁及安装环境

Oracle 数据库自发布至今，也经历了一个从不稳定到稳定，从功能简单至强大的过程。从第二版开始，Oracle 的每一次版本变迁，都具有里程碑意义。

- 1979 年的夏季，RSI（Oracle 公司的前身，Relational Software, Inc）发布了 Oracle 第二版。
- 1983 年 3 月，RSI 发布了 Oracle 第三版。从现在起 Oracle 产品有了一个关键的特性——可移植性。
- 1984 年 10 月，Oracle（RSI 更名为 Oracle）发布了第 4 版产品。这一版增加了读一致性这个重要特性。
- 1985 年，Oracle 发布了 5.0 版。这个版本是 Oracle 数据库较为稳定的版本。并实现了 C/S 模式工作。
- 1986 年，Oracle 发布了 5.1 版。该版本开始支持分布式查询。
- 1988 年，Oracle 发布了第 6 版。该版本中引入了行级锁特性，同时还引入了联机热备份功能。
- 1992 年 6 月，Oracle 发布了第 7 版。该版本增加了包括分布式事务处理功能、用于应用程序开发的新工具及安全性方法等功能。
- 1997 年 6 月，Oracle 第 8 版发布。Oracle8 支持面向对象的开发及新的多媒体应用。
- 1998 年 9 月，Oracle 公司正式发布 Oracle 8i。正是因为该版本对 Internet 的支持，所以，在版本号之后，添加了 i 标识。
- 2001 年 6 月，Oracle 发布了 Oracle 9i。
- 2003 年 9 月，Oracle 发布了 Oracle 10g。这一版的最大特性就是加入了网格计算的功能，因此版本号之后的标识使用了字母 g，代表 Grid——网格。
- 2007 年 7 月 11 日，Oracle 发布了 Oracle 11g。Oracle 11g 实现了信息生命周期管（Information Lifecycle Management）等多项创新。

Oracle 的最新版本为 Oracle 11g，但是，目前应用最广泛的版本为 Oracle 9i 和 Oracle 10g。本书选取使用的版本为 Oracle 10g。

Oracle 具有强大的功能，因此，对于硬件要求也较高。Oracle 10g 安装的硬件要求如下：

- 1024MB 以上的物理内存。
- 1.5~3.5 GB 磁盘空间，具体大小由安装类型决定。

软件环境如下：Windows XP、Windows 2003 或者 Linux Red Hat 5.0 以上版本。需要注意



的是, Windows Vista 与 Oracle 10g 的兼容性较差, 不推荐使用。

1.2.2 安装过程

本节所讲述的安装过程是在 Windows XP 下实现的。

1. 查看安装文件的目录结构

Oracle10g 的安装文件夹目录结构如图 1-1 所示。

其中, setup.exe 文件即为安装文件。

2. 数据库的安装方法

单击 setup.exe 文件后, 将出现【Oracle Database 10g 安装】对话框, 如图 1-2 所示。

在安装对话框中, 安装方法指的是基本安装和高级安装。高级安装允许用户针对数据字符集等进行配置, 对于初学者, 选择基本安装即可。【创建启动数据库】选项用于对是否创建默认数据库进行选择, 如果不选择, 则不会创建该数据库; 如果选择了该选项, 那么就需要为数据库的 4 个默认用户 SYS、SYSTEM、SYSMAN 和 DBSNMP 设置密码。在本例中设置为 abc123, 然后单击【下一步】按钮, 将进入数据库安装概要信息页面。

3. 数据库安装概要

数据库安装概要中, 详细列举了将要安装的 Oracle 产品的安装目录、组件信息等。在【Oracle Universal Installer: 概要】对话框中, 可以通过单击【已安装产品】按钮来查看本机已安装的 Oracle 组件, 如图 1-3 所示。

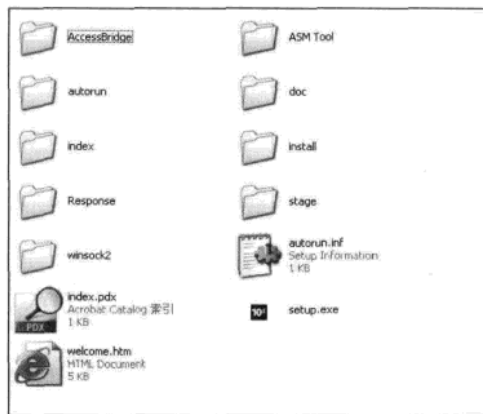


图 1-1 Oracle 10g 安装文件夹的目录结构

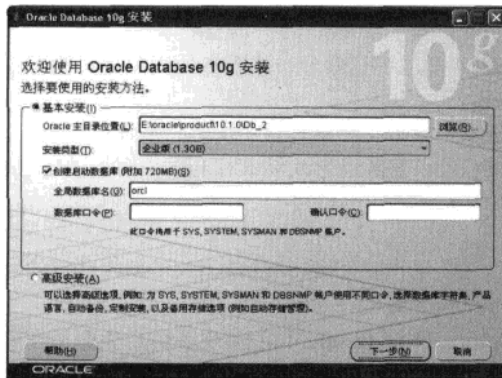


图 1-2 Oracle Database 10g 安装对话框

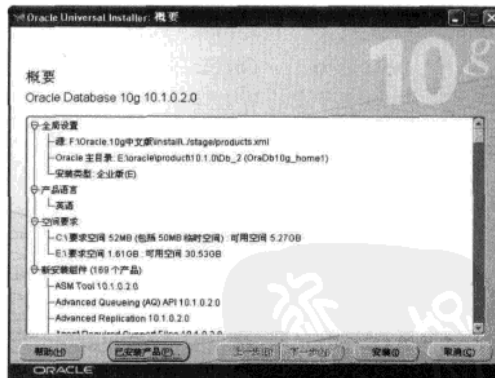


图 1-3 Oracle Universal Installer: 概要

在了解了数据库安装的概要状况之后, 单击【安装】按钮, 开始安装数据库, 如图 1-4 所示。

4. Oracle 安装之后的环境变量

Oracle 在安装过程中会修改系统的环境变量。数据库安装完成之后, 可以查看环境变量的变化。

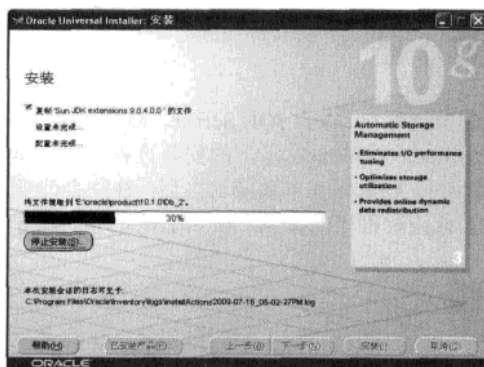


图 1-4 数据库安装进度窗口

- (1) 右键单击桌面上【我的电脑】图标，在弹出的快捷菜单中选择【属性】命令，弹出【系统属性】对话框。
- (2) 在【属性】对话框中选择【高级】选项卡。
- (3) 单击【高级】选项卡中的【环境变量】按钮。
- (4) 可以在【系统变量】列表中找到 Path 变量的值，如图 1-5 所示。

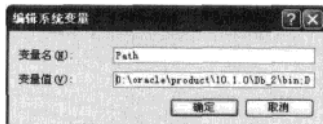


图 1-5 安装 Oracle 之后对 Path 变量的影响

在 Windows 的命令行下直接输入命令时，如果所输入的命令不是内部命令，那么 Windows 将在 Path 环境变量所指定的路径中搜索与输入命令同名的 EXE 文件进行执行。此时的环境变量 Path 的值已经发生了变化，其开头添加了如下路径。

```
D:\oracle\product\10.1.0\Db_2\bin;D:\oracle\product\10.1.0\Db_2\jre\1.4.2\bin\
client;D:\oracle\product\10.1.0\Db_2\jre\1.4.2\bin
```

需要注意的是，Oracle 提供了 JRE 的 bin 目录，并且该版本为 1.4 版本。如果在命令行中键入 java 或者 javac 等命令来执行或编译 java 文件，实际使用的是 JDK1.4，即使以前安装了其他版本的 JDK。



注意：Oracle 安装之后，针对环境变量的修改有可能影响其他应用程序的使用。

1.2.3 安装中需要注意的问题

Windows 下安装 Oracle 数据库简单易学，但应当注意以下几点。

- 注意硬件配置：Oracle 对硬件要求较高，因此，安装时尤其需要注意服务器的配置是否达到了要求。
- 注意空间分配：Oracle 作为数据库的本质，决定了其使用过程中是一个体积不断增大的过程，因此尤其需要注意对硬盘空间的分配。
- 安装路径的要求：Oracle 的安装路径不要包含中文字符。含有中文字符的路径在 Oracle 下并不能被很好地支持。



1.3 本章小结

本章介绍了当前主流的数据库 Oracle、SQL Server、My SQL，并突出了 Oracle 的特点——强大的功能、优越的性能。Oracle 在 Windows 下的安装过程比较简单，对于初学者，只需掌握在 Windows 下的安装过程即可。Oracle 在安装成功之后，往往会改变系统环境变量。由于系统环境变量的改变引起的对其他应用程序的影响，应当引起足够的注意。



1.4 习题

1. Oracle 安装的硬件环境是什么？
2. Oracle 当前常用的版本有哪些？
3. Oracle 9i 中的 i 和 Oracle 10g 中的 g 各代表什么意思？
4. Oracle 安装时需要注意的主要问题有哪些？



第 2 章 Oracle 常用工具

Oracle 的强大功能带来了一定的复杂性，相应地，甲骨文公司提供了很多配置管理工具，以方便用户的使用。Oracle 常用的配置管理工具包括：

- Oracle 企业管理器；
- Net Configuration Assistant（网络配置助手）；
- Oracle Net Manager（网络管理器）；
- Database Configuration Assistant（数据库配置助手）。

本章将着重讲述配置工具 Net Configuration Assistant 和 Oracle Net Manager 用法。通过本章的学习，读者可以利用这两种工具建立监听和 Net 服务名。



2.1 Net Configuration Assistant（网络配置助手）

网络配置助手，主要为用户提供 Oracle 数据库的监听程序、命名方法、本地 NET 服务名和目录配置。网络配置助手以向导的形式出现，使配置过程更加简单。

2.1.1 监听程序配置

监听器是 Oracle 基于服务器端的一种网络服务。监听器创建在数据库服务器端，主要作用是监视客户端的连接请求，并将请求转发给服务器。Oracle 监听器总是存在于数据库服务器端，因此在客户端创建监听器毫无意义。Oracle 监听器是基于端口的，也就是说，每个监听器会占用一个端口。配置监听程序的步骤如下。

① 在 Windows 任务栏中依次选择【开始】|【程序】|【Oracle 10g Home】|【Configuration and Migration Tools】|【Net Configuration Assistant】命令，将出现网络配置助手的欢迎界面，如图 2-1 所示。

② 选择【监听程序配置】单选按钮，并单击【下一步】按钮，将进入监听程序配置界面，如图 2-2 所示。



图 2-1 网络配置助手的欢迎界面

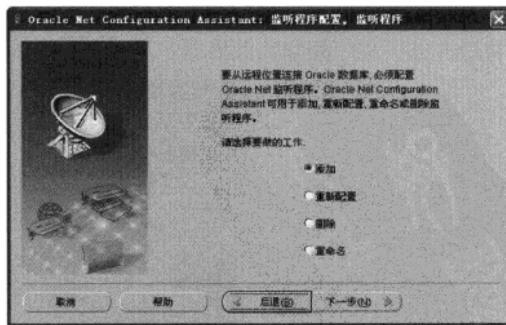


图 2-2 选择监听配置工作



③ 在工作选择界面中，选择【添加】单选按钮，并单击【下一步】按钮，将进入监听程序名配置界面，如图 2-3 所示。

④ 为监听程序输入名称，例如“LISTENER”。单击【下一步】按钮，将进入协议选择界面，如图 2-4 所示。

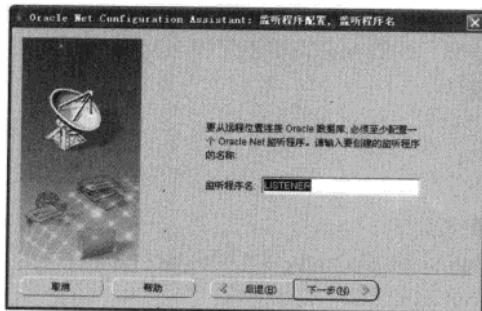


图 2-3 配置监听程序名

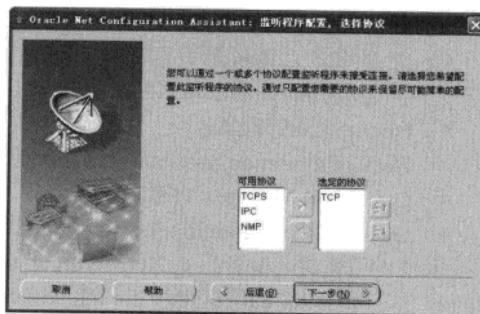


图 2-4 选择协议

⑤ 在协议选择界面中，保持默认的 TCP 协议即可。单击【下一步】按钮，将进入端口选择界面，如图 2-5 所示。

⑥ 在端口选择界面中，使用默认的 1521 端口。单击【下一步】按钮，将进入更多监听程序的选择界面，如图 2-6 所示。

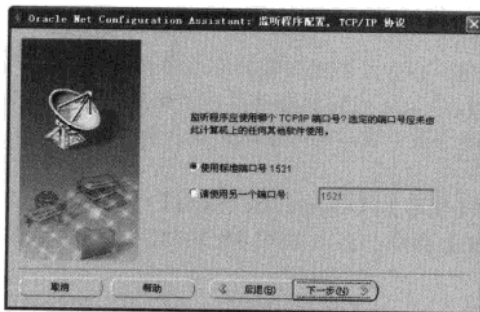


图 2-5 选择端口

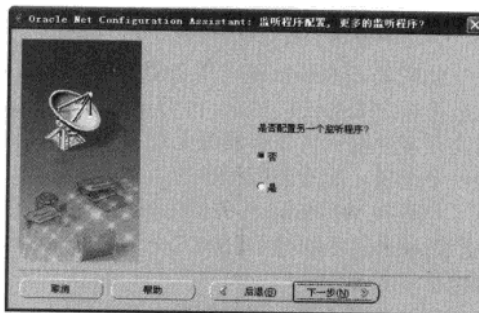


图 2-6 更多监听程序配置

⑦ 在【是否配置另一个监听程序】选项中，选择【否】单选按钮。单击【下一步】按钮，将进入监听程序配置完成界面，如图 2-7 所示。



图 2-7 监听程序配置成功

⑧ 在监听程序配置成功之后，需要关注的是操作系统中服务与 Oracle 安装目录下文件的变化。在操作系统的服务中，将会看到有关于新建监听的服务自动启动，如图 2-8 所示。



图 2-8 新建监听的 Windows 服务

在{ORACLE_HOME}\NETWORK\ADMIN 下会自动创建一个名为 listener.ora 的文件，其内容如下：

```
# listener.ora Network Configuration File: D:\oracle\product\10.1.0\Db_2\network\
admin\listener.ora
# Generated by Oracle configuration tools.

SID_LIST_LISTENER =
  (SID_LIST =
    (SID_DESC =
      (SID_NAME = PLSExtProc)
      (ORACLE_HOME = D:\oracle\product\10.1.0\Db_2)
      (PROGRAM = extproc)
    )
  )

LISTENER =
  (DESCRIPTION_LIST =
    (DESCRIPTION =
      (ADDRESS_LIST =
        (ADDRESS = (PROTOCOL = TCP) (HOST = 192.168.1.97) (PORT = 1521))
      )
      (ADDRESS_LIST =
        (ADDRESS = (PROTOCOL = IPC) (KEY = EXTPROC))
      )
    )
  )
```

该文件由网络配置助手自动生成。其中存储了各监听器的配置参数。LISTENER 为监听器名称；PROTOCOL=TCP 指定监听器所使用的协议为 TCP 协议；HOST=192.168.1.97 指定监听器所在的主机 IP，即 Oracle 数据库的安装主机；PORT = 1521 指定监听器的监听端口。

需要注意的是，该文件名 Listener 并非来自监听名 Listener。也就是说，即使新添其他监听程序，其配置也被记录在 Listener.ora 中。当 Oracle 环境中不存在任何监听时，Listener.ora 文件将被自动删除。

2.1.2 命名方法配置

Oracle 客户端在连接数据库服务时，并不会直接使用数据库名等信息，而是使用连接标识符。连接标识符一般存储了连接的详细信息。定义连接标识符的方法一般有 5 种。

- 主机命名 (Host Naming)：客户端利用 TCP/IP 协议、Oracle Net Services 和 TCP/IP 协议适配器，仅凭主机地址，即可建立与数据库的连接。
- 本地命名：使用在每个 Oracle 客户端的 tnsnames.ora 文件中配置和存储的信息来获得数据库的连接描述符，从而实现与数据库的连接。
- 目录命名：将数据库服务或网络服务名解析为连接描述符，该描述符存储在中央目录服务器中。
- Oracle Names：这是由 Oracle Names 服务器系统构成的 Oracle 目录服务，这些服务器可以为网络上的每个服务提供由名称到地址的解析。



- 外部命名：使用受支持的第三方命名服务。

对于一般的小型应用，最常用的命名方法为本地命名方法。配置 Oracle 的命名方法为本地命名的步骤如下。

① 打开【Oracle Net Configuration Assistant】，并选择【命名方法配置】单选按钮，如图 2-9 所示。

② 单击【下一步】按钮，将进入【命名方法配置】界面，如图 2-10 所示。



图 2-9 选择命名方法配置

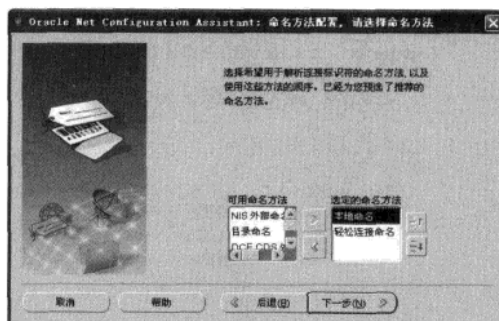


图 2-10 命名方法配置

在右侧的列表中，列出了已选择的命名方法。默认情况下，Oracle 推荐使用本地命名和轻松连接命名，二者的顺序为：首先搜索本地命名，如果不能获得连接描述符，接着搜索轻松连接命名。Oracle 提供了这种允许多种命名方法共存，使用顺序来指定优先级的方式来配置命名方法。在本例中，保持默认值，并单击【下一步】按钮。将进入命名方法配置成功界面，如图 2-11 所示。

在成功配置命名方法之后，可以打开{ORACLE_HOME}\NETWORK\ADMIN\sqlnet.ora 文件，文件内容如下：

```
# sqlnet.ora Network Configuration File: D:\oracle\product\10.1.0\Db_2\ network\
admin\sqlnet.ora
# Generated by Oracle configuration tools.
```

```
SQLNET.AUTHENTICATION_SERVICES= (NTS)
```

```
NAMES.DIRECTORY_PATH= (TNSNAMES, EZCONNECT)
```

其中，NAMES.DIRECTORY_PATH 即为命名方法的搜索路径：首先搜索本地命名，然后搜索轻松连接命名。

2.1.3 本地 Net 服务名配置

本地 Net 服务名配置，即为 2.1.2 节中提到的本地命名。创建一个新的本地 Net 服务名的步骤如下。

① 在 Oracle 网络配置助手的欢迎界面选择【本地 Net 服务名配置】单选按钮，如图 2-12 所示。

② 单击【下一步】按钮，将进入工作选择界面，如图 2-13 所示。

③ 该界面中提供了添加、重新配置、删除、重命名、测试等工作选项。在此，选择【添加】单选按钮，并单击【下一步】按钮，将进入服务名配置界面，如图 2-14 所示。



图 2-11 命名方法配置成功

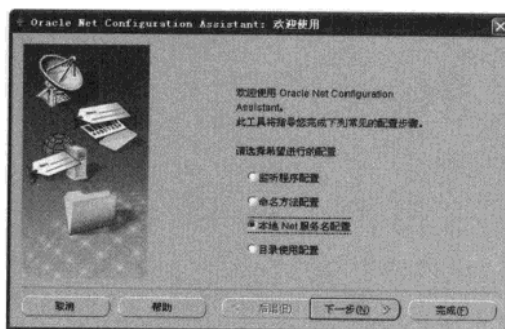


图 2-12 选择本地 Net 服务名配置

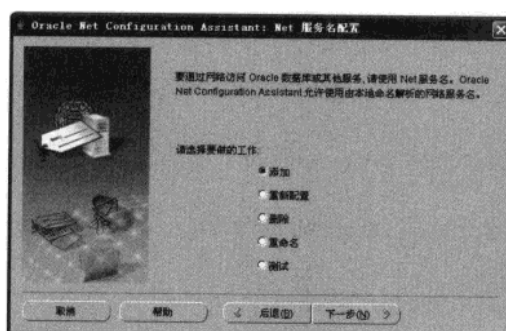


图 2-13 服务名配置中的工作选择

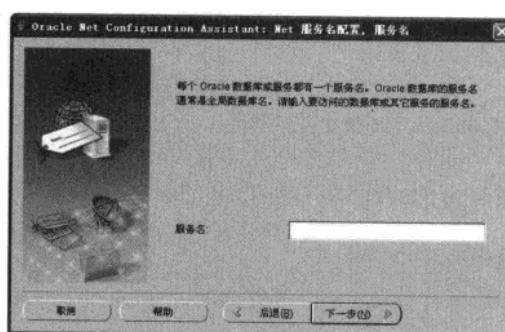


图 2-14 服务名配置界面

④ 在该页面中输入数据库服务名。一般为数据库的全局数据库名，例如默认的 ORCL。单击【下一步】按钮，将进入协议选择界面，如图 2-15 所示。

⑤ 保持协议选择为默认的 TCP 协议，并单击【下一步】按钮，将进入 TCP/IP 协议的详细配置，如图 2-16 所示。

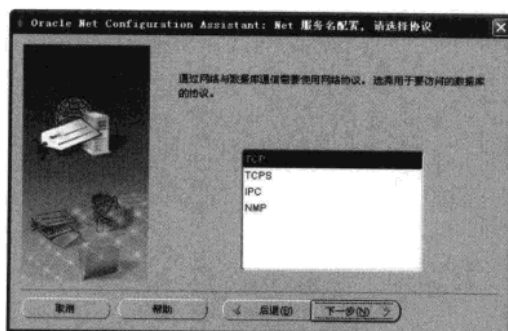


图 2-15 为新建服务名选择协议

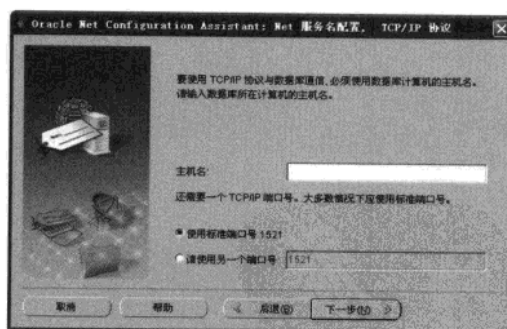


图 2-16 TCP/IP 协议详细设置

⑥ 为主机名输入本机 IP，例如，192.168.1.97，并保持端口号为默认的 1521。需要注意的是，这里的端口号，必须与服务器端的监听器端口号保持一致。单击【下一步】按钮，将进入测试界面，如图 2-17 所示。

⑦ 在测试页面中选择【是，进行测试】单选按钮，并单击【下一步】按钮，开始进行测试。第一次的测试往往不会成功。最常见的原因为用户名和密码错误，如图 2-18 所示。

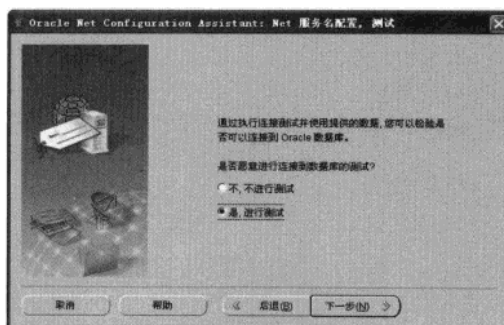


图 2-17 服务名测试页面

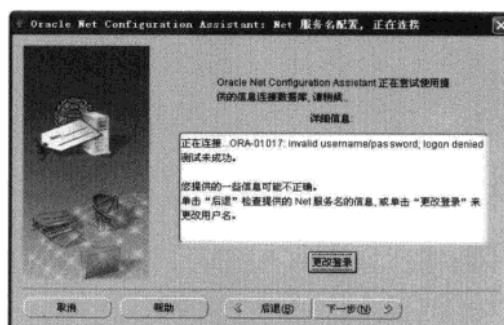


图 2-18 用户名/密码错误导致测试失败

⑧ 单击【更改登录】按钮，可以修改用户名和密码，然后再次进行测试，直至测试成功，如图 2-19 所示。

⑨ 单击【下一步】按钮，进入 NET 服务名页面，并为新建的 NET 服务指定名称，例如 ORACLE，如图 2-20 所示。

此时，打开文件 {ORACLE_HOME}\NETWORK\ADMIN\tnsnames.ora，会发现该文件添加了有关 Net 服务名 ORACLE 的内容，代码如下：

```
# tnsnames.ora Network Configuration File: D:\oracle\product\10.1.0\Db_2\network\
admin\tnsnames.ora
# Generated by Oracle configuration tools.

ORACLE =
(DESCRIPTION =
  (ADDRESS_LIST =
    (ADDRESS = (PROTOCOL = TCP) (HOST = 127.0.0.1) (PORT = 1521))
  )
  (CONNECT_DATA =
    (SERVER = DEDICATED)
    (SERVICE_NAME = ORCL)
  )
)
```

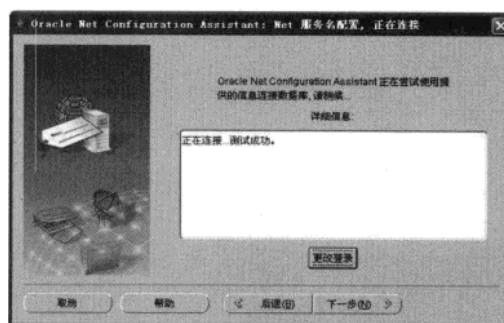


图 2-19 修改用户和密码之后，测试成功



图 2-20 为新建的 Net 服务指定名称



2.2 Net Manager (网络管理员)

Net Manager 和 Net Configuration Assistant 功能相似。Net Configuration Assistant 总是以向导的模式出现，可以引导初学者进行配置；而 Net Manager 则将所有配置步骤集合到同一界面，

更适合熟练者进行操作。

在 Windows 任务栏中依次选择【开始】|【程序】|【Oracle 10g Home】|【Configuration and Migration Tools】|【Net Manager】命令，将出现 Net Manager 的管理页面，如图 2-21 所示。

从图 2-21 可以看出，Net Manager 的主要内容为本地服务命名和监听程序。在 2.1 节中所进行的配置，都会加载到 Net Manager 中。选中服务命名 ORALCE，可以在右侧窗口中查看其详细配置。这其中包括服务名（全局数据库名）、协议 TCP/IP、主机名（192.168.1.97）和端口号 1521。

选中监听程序 LISTENER，可以在右侧窗口中查看其详细信息，如图 2-22 所示。

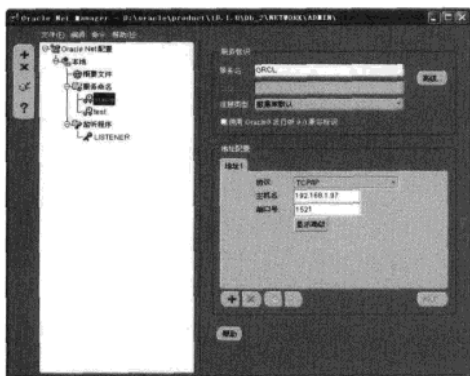


图 2-21 Net Manager 管理界面

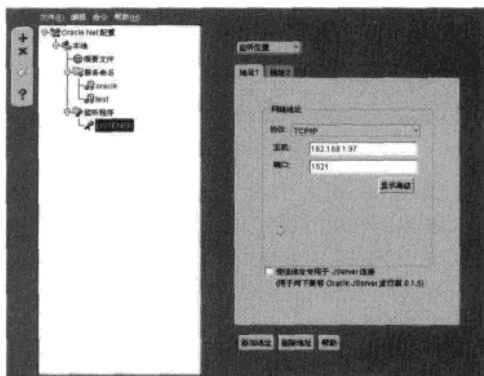


图 2-22 查看 LISTENER 的详细信息

在监听器 LISTENER 中，其使用的协议为 TCP/IP，监听位置为 192.168.1.97 上的 1521 端口。同样，可以利用 Net Manager 来管理监听和本地 Net 服务名。

【范例 2-1】 演示如何利用 Net Manager 来创建 Net 服务名。

- (1) 选中 oracle 服务，并单击左侧的删除按钮，将该 Net 服务名删除，如图 2-23 所示。
- (2) 选中【服务命名】文件夹，并单击左侧的添加按钮，将弹出 Net 服务名向导，如图 2-24 所示。

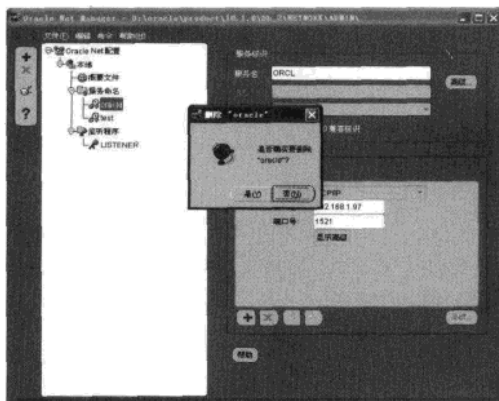


图 2-23 删除 Net 服务名 oracle

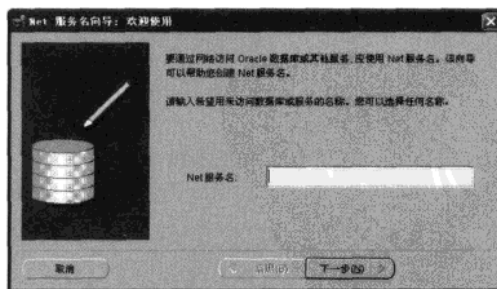


图 2-24 添加 Net 服务名

- (3) 在 Net 服务名中输入名称“ORACLE”，并单击【下一步】按钮，将弹出协议选择界



面，如图 2-25 所示。

(4) 选择默认的 TCP/IP 协议，并单击【下一步】按钮，将进入协议设置界面，如图 2-26 所示。

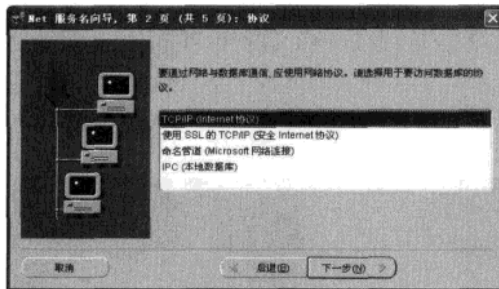


图 2-25 选择协议

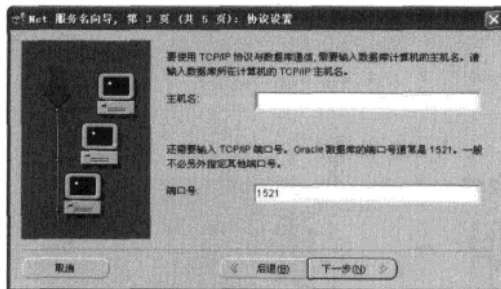


图 2-26 协议设置界面

(5) 为主机名输入 IP: 192.168.1.97，并保持端口号 1521 不变，单击【下一步】按钮，将进入服务名设置界面，如图 2-27 所示。

(6) 将服务名指定为 ORCL，即 Oracle 数据库的全局数据库名，并单击【下一步】按钮，将进入测试页面。在测试成功之后，单击【完成】按钮，以完成 Net 服务名的创建，如图 2-28 所示。

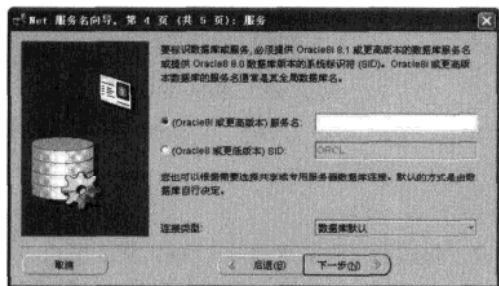


图 2-27 数据库服务名设置

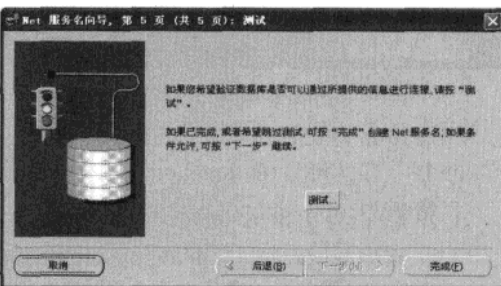


图 2-28 完成 Net 服务名的创建

(7) 界面切换到 Net Manager 的主页面，会发现新的 Net 服务名 ORACLE 已经出现在【服务命名】的管理列表中。通过单击左侧的测试按钮，可以测试 Net 服务名的连接是否成功。

(8) 最后需要注意的是，在完成网络配置的修改之后，需要进行保存。



2.3 本章实例

Net 服务名配置在 Oracle 数据库的客户端，因此，在同一台客户端主机上可以存在多个 Net 服务名，这些 Net 服务名的名称不同，但是却可以使用完全相同的配置，来连接同一个数据库实例。实例 2-2 演示了如何创建与范例 2-1 具有相同配置的 Net 服务名。

【范例 2-2】 演示创建相同配置、但不同名称的 Net 服务名。

(1) 参照路径【开始】|【程序】|【Oracle 10g Home】|【Configuration and Migration Tools】|【Net Manager】来打开 Net Manager，并查看已存在的 Net 服务名，如图 2-29 所示。

(2) 选中【服务命名】文件夹，并单击左侧的添加按钮，将弹出 Net 服务名向导，如图 2-30

所示。

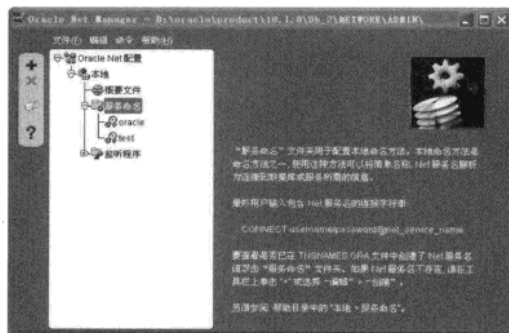


图 2-29 查看已存在的 Net 服务名

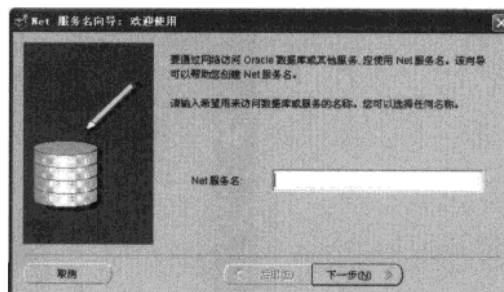


图 2-30 添加 Net 服务名

(3) 在 Net 服务名中输入名称“ORCL”，并单击【下一步】按钮，将弹出协议选择界面，如图 2-31 所示。

(4) 选择默认的 TCP/IP 协议，并单击【下一步】按钮，将进入协议设置界面，如图 2-32 所示。

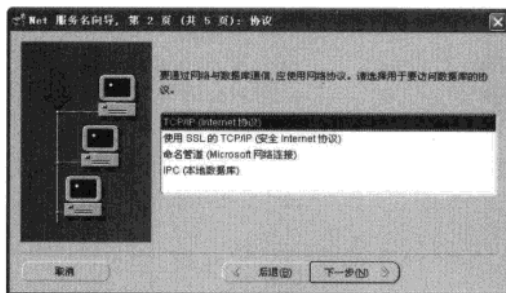


图 2-31 选择协议

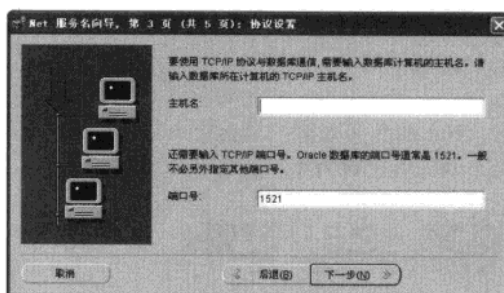


图 2-32 协议设置界面

(5) 为主机名输入 IP: 192.168.1.97，并保持端口号 1521 不变。注意，这里的主机地址和端口号均与已存在的 Net 服务名 ORACLE 中的配置一致。单击【下一步】按钮，将进入服务名设置界面，如图 2-33 所示。

(6) 将服务名指定为 ORCL，即 Oracle 数据库的全局数据库名。注意，这里的 Oracle 数据库的全局数据库名也与已存在的 Net 服务名 ORACLE 中配置的一致。并单击【下一步】按钮，将进入测试页面。在测试成功之后，单击【完成】按钮，以完成 Net 服务名的创建，如图 2-34 所示。

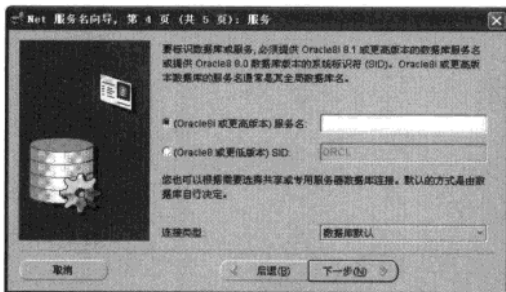


图 2-33 数据库服务名设置

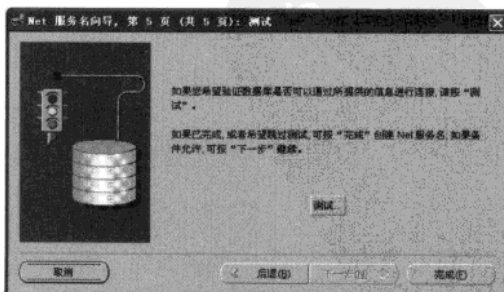


图 2-34 完成 Net 服务名的创建



(7) 界面切换到 Net Manager 的主页面，会发现新的 Net 服务名 ORACLE 已经出现在【服务命名】的管理列表中，如图 2-35 所示。

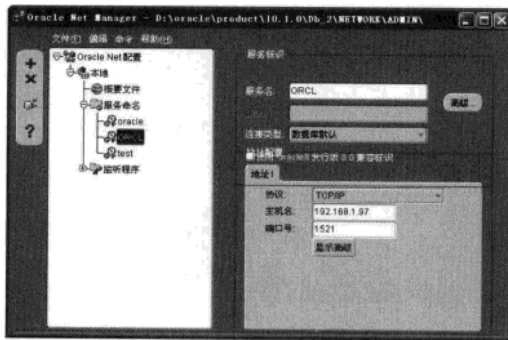


图 2-35 具有不同名称，但是配置完全相同的 Net 服务名



2.4 本章小结

本章着重讲述了两个常用的工具 Net Configuration Assistant 和 Net Manager，并讲述了如何创建监听程序和 Net 服务名。这里需要注意的是，监听程序属于服务器端概念，也就是说，监听永远处于服务器端。它负责将客户端请求转发到相应的数据库实例。而 Net 服务名是客户端概念，Net 服务名是客户端自定义的，只为本机服务。因此，会出现连接同一个数据库实例，但是不同的客户机有不同的 Net 服务名的情形。当然，这些 Net 服务名的连接描述信息是相同的。



2.5 习题

1. 简述 Net Configuration Assistant（网络配置助手）与 Net Manager（网络管理员）这两种工具的关系。
2. 监听程序配置中的主要参数有哪几个？
3. 本地 Net 服务名配置中的主要参数有哪几个？
4. 监听程序与 Net 服务名的关系是怎样的？



第3章 SQL Plus 和 PL/SQL

在 Oracle 中，有两个非常重要但又极易混淆的概念——SQL Plus 和 PL/SQL。本章将讲述二者的区别，并重点讲述 SQL Plus 工具的使用。而 PL/SQL 则在以后的存储过程等过程化编程实例中演示其应用。本章的详细内容包括：

- SQL Plus 与 PL/SQL 概述；
- 如何使用 SQL Plus；
- 在 SQL Plus 中格式化输出；
- PL/SQL 的常用开发工具。

通过本章的学习，读者可以清晰地了解 SQL Plus 与 PL/SQL 的概念，并能够使用 SQL Plus 来开发简单的 PL/SQL 应用。



3.1 SQL Plus 与 PL/SQL 简介

SQL Plus 是 Oracle 提供了一种用户接口。类似于操作系统的命令行，用户可以通过在 SQL Plus 中输入命令来向数据库发送命令，而数据库也将处理结果通过 SQL Plus 呈现给用户。也就是说，SQL Plus 是数据库与用户之间进行交互的工具。

PL/SQL 则是 Oracle 的过程化编程语言。PL/SQL 定义了大量语法，用户可以遵循这些语法来定义程序块，以完成复杂的数据库操作。Oracle 客户端可以解释这些程序块，并将这些命令请求发送到数据库，进行相应的数据库操作。而且，这些代码块可以作为数据库对象进行存储，这有利于实现代码复用。PL/SQL 程序块的开发工作，照样可以在用户工具 SQL Plus 上完成，如图 3-1 所示。

在图 3-1 中，程序编写窗口即为登录 SQL Plus 之后呈现在用户面前的界面。而其中的代码：

```
SQL> begin
2   for employee in (select * from emp) loop
3       dbms_output.put_line(employee.empno || '号员工是' || employee.ename);
4   end loop;
5 end;
```

则是用户编写的 PL/SQL 程序块。第 6 行命令：

```
6 /
```

则用于执行已编写的 PL/SQL 程序块，向数据库发出请求。当数据库处理完客户端请求之后，将处理结果返回给客户端，并显示在 SQL Plus 中。

```
7369 号员工是 SMITH
7499 号员工是 ALLEN
7521 号员工是 WARD
7566 号员工是 JONES
7654 号员工是 MARTIN
7698 号员工是 BLAKE
7782 号员工是 CLARK
```





```

7788 号员工是 SCOTT
7839 号员工是 KING
7844 号员工是 TURNER
7876 号员工是 ADAMS
7900 号员工是 JAMES
7902 号员工是 FORD
7934 号员工是 MILLER

```

PL/SQL procedure successfully completed.

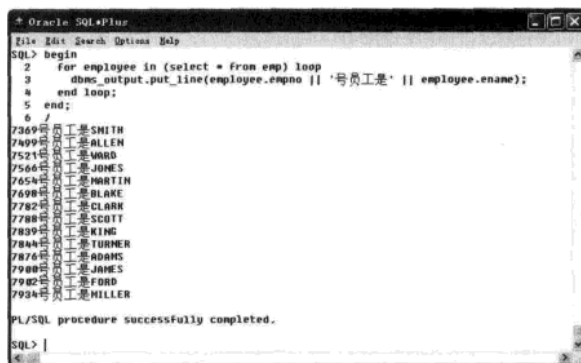


图 3-1 在 SQL Plus 中开发 PL/SQL 程序

以上步骤是一个简单的用户与数据库交互的实例，可以看出 SQL Plus（交互工具）与 PL/SQL（交互语言）的区别，以及二者在整个交互过程中所扮演的角色。



3.2 使用 SQL Plus

SQL Plus 是一个常用工具，其提供者 Oracle。SQL Plus 具有免费、小巧、灵活等优秀的特点。因此，经常用做简单查询、更新数据库对象、更新数据库中数据、调试数据库等的首选工具。本节将详细讲述 SQL Plus 的使用。

3.2.1 登录 SQL Plus

SQL Plus 有两种模式，一种为命令行模式，另一种为 GUI 模式。这两种方式具有相同的功能，但是 GUI 模式的用户界面更加友好。这两种登录方式实际对应了两个可执行文件。在 Windows 下，打开 Oracle 安装目录下的 BIN 文件夹，会获得这两个可执行文件，如图 3-2 所示。



图 3-2 SQL Plus 的两种模式所对应的可执行文件

由于在 Oracle 安装过程中，已经将 BIN 目录置于环境变量中，因此，可以利用 Windows 的【开始】|【运行】命令。在【打开】文本框中输入文件名来打开这两种模式的 SQL Plus。

```
sqlplus scott/abc123@orcl
```

此时，打开的是命令行模式的 SQL Plus，如图 3-3 所示。

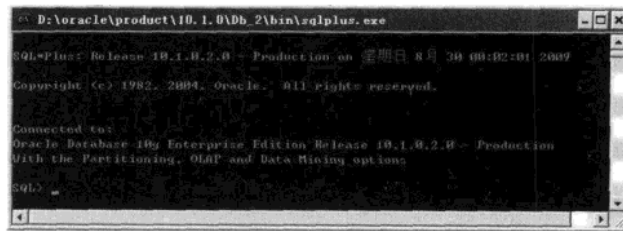


图 3-3 命令行模式下的 SQL Plus

可见，命令行模式与 DOS 界面十分相似，只是其命令提示符为 SQL>。
如果使用以下命令：

```
sqlplusw scott/abc123@orcl
```

打开的则是 GUI 模式的 SQL Plus，如图 3-4 所示。

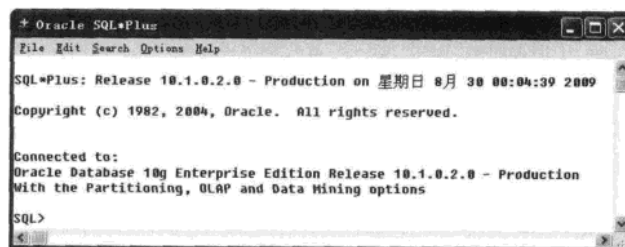


图 3-4 GUI 模式下的 SQL Plus

在两条登录命令中，scott/abc123 为登录的用户名和密码；@orcl 则为在 TNS (tnsnames.ora) 中配置的数据库的 Net 服务名。在本例中，以数据库的默认用户 scott 进行登录。成功登录之后，即可进行与数据库的交互。

3.2.2 SQL Plus 输出结果的格式化

可以在 SQL Plus 中执行数据库操作，就像在其他数据库工具中一样。但是，SQL Plus 有一个缺点，即输出结果的格式化效果并不尽人意。例如，利用 SQL Plus 登录数据库，并查询表 emp 的内容。

```
sqlplusw scott/abc123@orcl
```

表 EMP 虽然只有 14 行，但是查询结果却非常缺乏可读性：

```
SQL> select * from emp;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM
7369	SMITH	CLERK	7902	17-12 月-80	800	
7499	ALLEN	SALESMAN	7698	20-2 月 -81	1600	300
7521	WARD	SALESMAN	7698	22-2 月 -81	1250	500



21 天学通 Oracle

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM

DEPTNO						

7566	JONES	MANAGER	7839	02-4 月 -81	2975	
20						
7654	MARTIN	SALESMAN	7698	28-9 月 -81	1250	1400
30						
7698	BLAKE	MANAGER	7839	01-5 月 -81	2850	
30						
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM

DEPTNO						

7782	CLARK	MANAGER	7839	09-6 月 -81	2450	
10						
7788	SCOTT	ANALYST	7566	19-4 月 -87	3000	
20						
7839	KING	PRESIDENT		17-11 月-81	5000	
10						
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM

DEPTNO						

7844	TURNER	SALESMAN	7698	08-9 月 -81	1500	0
30						
7876	ADAMS	CLERK	7788	23-5 月 -87	1100	
20						
7900	JAMES	CLERK	7698	03-12 月-81	950	
30						
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM

DEPTNO						

7902	FORD	ANALYST	7566	03-12 月-81	3000	
20						
7934	MILLER	CLERK	7782	23-1 月 -82	1300	
10						

14 rows selected.

分析此时的输出结果可知, SQL Plus 每行输出都具有一定长度。如果该行超出了这个长度, 那么将执行换行操作。另外, 当输出的行数超过了一定的数量时, 也将执行分页。

1. 指定行的长度

SQL Plus 输出行的默认长度为 80, 可以通过 SET 命令重新设定行的长度。

```
SQL> show linesize;
linesize 80
SQL> set linesize 100;
```

此时，再次执行相同的查询语句，查询结果如图 3-5 所示。

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMN	DEPTNO
7369	SMITH	CLERK	7902	17-12月-80	800		20
7499	ALLEN	SALESMAN	7698	20-2月-81	1600	300	30
7521	WARD	SALESMAN	7698	22-2月-81	1250	500	30
7566	JONES	MANAGER	7839	02-4月-81	2975		20
7654	MARTIN	SALESMAN	7698	28-9月-81	1250	1400	30
7698	BLAKE	MANAGER	7839	01-5月-81	2850		30
7782	CLARK	MANAGER	7839	09-6月-81	2450		10
7788	SCOTT	ANALYST	7566	19-4月-82	3000		20
7839	KING	PRESIDENT		17-11月-81	5000		10
7844	TURNER	SALESMAN	7698	08-9月-81	1500	0	30
7876	ADAMS	CLERK	7788	23-5月-82	1100		20
14 rows selected.							
7900	JAMES	CLERK	7698	03-12月-81	950		30
7902	FORD	ANALYST	7566	03-12月-81	3000		20
7934	MILLER	CLERK	7782	23-1月-82	1300		10

图 3-5 设定 SQL Plus 行的长度

2. 指定分页的尺寸

当重新设定 SQL Plus 行的长度之后，已经具有了一定的可读性。但是，14 条记录被分割为两页。这与 SQL Plus 的另一属性 `pagesize` 有关。

```
SQL> show pagesize;
pagesize 14
```

这里需要注意的是，`pagesize` 是从输出的第一行开始计数的。SQL Plus 每页的开头都会有一个空行，并且将查询列名及列名与实际数据之间的分隔行都计算在内。因此，第一页的内容如图 3-6 所示。

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMN	DEPTNO
7369	SMITH	CLERK	7902	17-12月-80	800		20
7499	ALLEN	SALESMAN	7698	20-2月-81	1600	300	30
7521	WARD	SALESMAN	7698	22-2月-81	1250	500	30
7566	JONES	MANAGER	7839	02-4月-81	2975		20
7654	MARTIN	SALESMAN	7698	28-9月-81	1250	1400	30
7698	BLAKE	MANAGER	7839	01-5月-81	2850		30
7782	CLARK	MANAGER	7839	09-6月-81	2450		10
7788	SCOTT	ANALYST	7566	19-4月-82	3000		20
7839	KING	PRESIDENT		17-11月-81	5000		10
7844	TURNER	SALESMAN	7698	08-9月-81	1500	0	30
7876	ADAMS	CLERK	7788	23-5月-82	1100		20

图 3-6 默认分页的第一页

而第二页的内容如图 3-7 所示。

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMN	DEPTNO
7900	JAMES	CLERK	7698	03-12月-81	950		30
7902	FORD	ANALYST	7566	03-12月-81	3000		20
7934	MILLER	CLERK	7782	23-1月-82	1300		10

图 3-7 默认分页第二页

可以利用 `set pagesize` 命令重设分页尺寸，从而实现输出效果的格式化。

```
SQL> set pagesize 20;
```



此时的查询效果如图 3-8 所示。

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-12月-80	800		20
7499	ALLEN	SALESMAN	7698	20-2月-81	1600	300	30
7521	WARD	SALESMAN	7698	22-2月-81	1250	500	30
7566	JONES	MANAGER	7839	02-4月-81	2975		20
7654	MARTIN	SALESMAN	7698	28-9月-81	1250	1400	30
7698	BLAKE	MANAGER	7839	01-5月-81	2850		30
7782	CLARK	MANAGER	7839	09-6月-81	2450		10
7788	SCOTT	ANALYST	7566	19-4月-87	3000		20
7839	KING	PRESIDENT		17-11月-81	5000		10
7844	TURNER	SALESMAN	7698	08-9月-81	1500		30
7876	ADAMS	CLERK	7788	23-5月-87	1100		20
7900	JAMES	CLERK	7698	03-12月-81	950		30
7902	FORD	ANALYST	7566	03-12月-81	3000		20
7934	MILLER	CLERK	7782	23-1月-82	1300		10

14 rows selected.

图 3-8 重设分页尺寸之后的输出效果

3. 输出暂停

尽管通过指定 `pagesize` 可以自定义每页显示的行数，但是当查询结果集合非常大时，页面会快速滚动，就像 DOS 或者 UNIX 环境中的屏幕快速滚动，这将严重影响用户体验。在 DOS 和 UNIX 环境中，可以利用 `more` 命令来实现用户手动滚屏。而在 SQL Plus 中则可以使用暂停属性来实现用户手动滚屏。

```
SQL> set pause on;
```

使用账户 `system` 登录 SQL Plus，并设置暂停属性为 `on`，接着查询视图 `user_objects` 中的内容。需要注意的是，SQL Plus 返回第一页之前会暂停，如图 3-9 所示。

OBJECT_NAME	OBJECT_TYPE	CREATED	LAST_DDL_TIME	TimestamP	STATUS
-------------	-------------	---------	---------------	-----------	--------

图 3-9 SQL Plus 在第一页之前暂停

此时，按下 `Enter` 键，SQL Plus 将自动显示下一屏，如图 3-10 所示。

OBJECT_NAME	OBJECT_TYPE	CREATED	LAST_DDL_TIME	TimestamP	STATUS
SYSCATALOG	SYNDHVN	09-03月-04	09-03月-04	2004-03-09:23:59:59	VALID N N N
CATALOG	SYNDHVN	09-03月-04	09-03月-04	2004-03-09:23:59:59	VALID N N N
TAB	SYNDHVN	09-03月-04	09-03月-04	2004-03-09:23:59:59	VALID N N N

图 3-10 按下 `Enter` 键，SQL Plus 自动滚屏

在滚屏过程中，如欲退出至 SQL 命令提示符，则可以按下 Ctrl+C 组合键。

如欲撤销暂停功能，可以使用如下命令。

```
SQL> set pause off;
```

4. feedback

在 SQL Plus 的查询结果的尾行，往往会出现结果集中记录条数的提示信息。该信息即为 feedback。可以通过如下语句查看 SQL Plus 中的 feedback 设置。

```
SQL> show feedback;
FEEDBACK ON for 6 or more rows
```

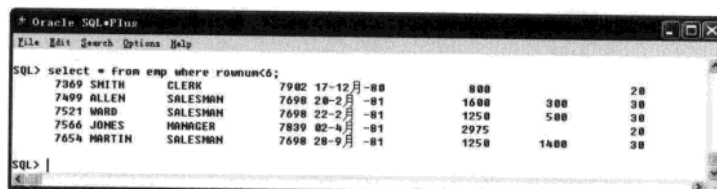
FEEDBACK ON 表示提示信息处于打开状态；for 6 or more rows 表示，仅仅当查询结果集中包含 6 条或 6 条以上记录时，该提示信息才会出现。范例 3-1 演示了针对该临界条件的测试。

【范例 3-1】测试 SQL Plus 的 feedback 属性。

利用如下 SQL 语句，获得表 emp 中的前 5 条记录。

```
SQL> select * from emp where rownum<6;
```

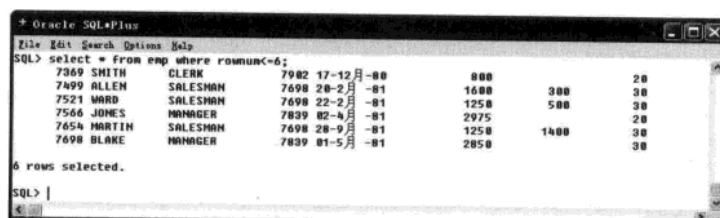
其查询结果如图 3-11 所示。



EMPNO	ENAME	JOB	START_DATE	MONTHS	SAL	COM
7369	SMITH	CLERK	17-12月-80	800		20
7499	ALLEN	SALESMAN	19-8月-81	1600	300	30
7521	WARD	SALESMAN	22-2月-81	1250	500	30
7566	JONES	MANAGER	02-4月-81	2975		20
7654	MARTIN	SALESMAN	28-9月-81	1250	1400	30

图 3-11 查询表 EMP 中的前 5 条记录

当查询结果集中的记录为 5 条时，feedback 的信息被忽略。但是，当查询结果集中的记录条数达到 6 时，feedback 的信息会被反馈给客户，如图 3-12 所示。



EMPNO	ENAME	JOB	START_DATE	MONTHS	SAL	COM
7369	SMITH	CLERK	17-12月-80	800		20
7499	ALLEN	SALESMAN	19-8月-81	1600	300	30
7521	WARD	SALESMAN	22-2月-81	1250	500	30
7566	JONES	MANAGER	02-4月-81	2975		20
7654	MARTIN	SALESMAN	28-9月-81	1250	1400	30
7698	BLAKE	MANAGER	01-5月-81	2850		30

6 rows selected.

图 3-12 查询表 EMP 中的前 6 条记录

可以为 feedback 自定义数值，来指定输出 feedback 信息的最小记录条数，代码如下：

```
SQL> set feedback 5;
```

一旦将 feedback 的数值设为 5，那么图 3-11 中的查询语句将能够生成 feedback 信息。如图 3-13 所示。

当然，也可以完全屏蔽 feedback 的信息，相应代码如下：

```
SQL> set feedback off;
```

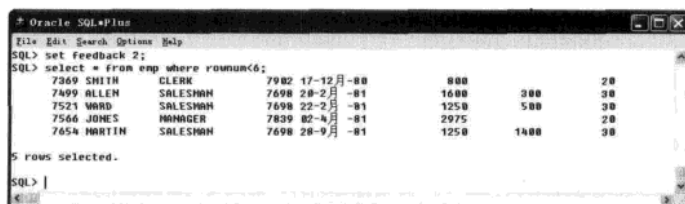


图 3-13 查询表 EMP 中的前 5 条记录

此时，即使结果集的记录数大于 feedback 的设定值，feedback 信息仍然不会被输出，如图 3-14 所示。

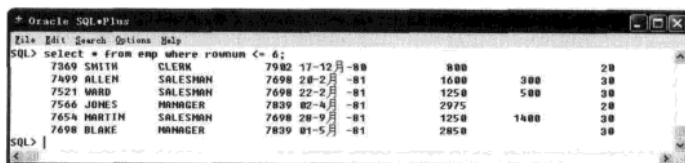


图 3-14 屏蔽 feedback 信息

3.2.3 SQL Plus 小结

SQL Plus 是一款简单易用的 Oracle 客户端工具。除了在本节介绍的格式化属性之外，还可以通过指定其他属性来更好地格式化输出结果。例如，使用 column 命令来设定特定列的输出格式。值得注意的是，在 SQL Plus 中自定义的属性，当 SQL Plus 会话关闭时将失效。

除此之外，还可以通过 host 关键字，后跟系统命令的方式来调用 DOS 命令（Windows）；或者使用“!”后跟系统命令来执行实际的系统命令（Linux）。

总之，SQL Plus 有着丰富的内容，读者可以参考 Oracle 文档进行进一步了解。



3.3 PL/SQL

PL/SQL 是 Procedural Language/SQL 的缩写形式，意为过程化编程语言。PL/SQL 是 Oracle 公司开发的，并且只能在 Oracle 数据库中运行。与其他面向过程的编程语言类似，PL/SQL 的语句对大小写并不敏感。本节将首先介绍 PL/SQL 的开发工具，并开发一个简单的 PL/SQL 程序，以使读者对其有一个大体的了解。

3.3.1 PL/SQL 常用开发工具

PL/SQL 的常用开发工具有 PL/SQL Developer 和 TOAD。其中，PL/SQL Developer 是 Oracle 公司开发的、专门针对 Oracle 的客户端开发工具；而 TOAD 由 Quest Software 开发，是当今世界上排名第一的针对 Oracle 进行开发和管理的工具。

1. PL/SQL Developer

目前，PL/SQL Developer 的最成熟版本为 PL/SQL 7。在成功安装该软件之后，可以在桌面上单击其图标。该工具的第一个窗口为登录窗口，要求用户输入用户名和密码。为其输入用户名和密码（以系统用户 SYSTEM 为例），如图 3-15 所示。

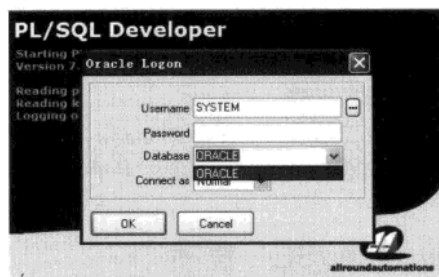


图 3-15 PL/SQL 的登录窗口

在 Database 下拉框中，已经存在了若干可选项。这些可选项来自于 Oracle TNS 的设置。例如，在图 3-15 中，含有选项 ORACLE。文件 {ORACLE_HOME}\NETWORK\ADMIN\tnsnames.ora 中的内容如下：

```
# tnsnames.ora Network Configuration File: D:\oracle\product\10.1.0\Db_2\
network\admin\tnsnames.ora
# Generated by Oracle configuration tools.
```

#对应于 PL/SQL Developer 登录窗口中的数据库选项

```
ORACLE =
(DESCRIPTION =
  (ADDRESS_LIST =
    (ADDRESS = (PROTOCOL = TCP) (HOST = 127.0.0.1) (PORT = 1521))
  )
  (CONNECT_DATA =
    (SERVER = DEDICATED)
    (SERVICE_NAME = orcl)
  )
)
```

PL/SQL Developer 在登录时，首先根据用户所选择的 Database 选项来获得对应的数据库连接。然后，根据数据库中所对应的用户名和密码信息进行校验。

选择 Oracle 服务名，并为 Username 输入“system”，为 password 输入数据库安装时的密码，如“abc123”。然后单击【OK】按钮即可登录 PL/SQL Developer。

2. TOAD

TOAD 的功能要强于 PL/SQL Developer。同样，单击 TOAD 的运行图标，首先弹出的也为登录窗口，如图 3-16 所示。

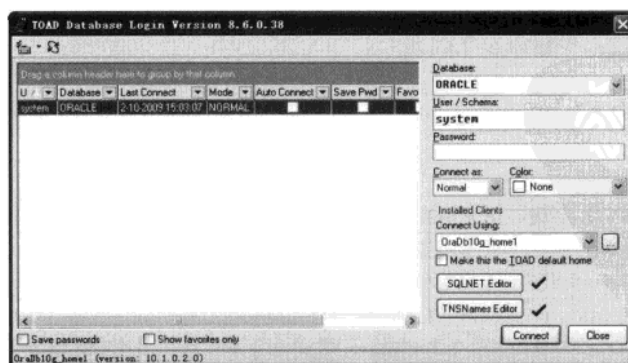


图 3-16 TOAD 的登录窗口



TOAD 的登录窗口中需要填写的内容和 PL/SQL Developer 的填写内容基本一致。Database 下拉框同样从 TNS 的配置中获取数据库服务名。User/Schema 用于输入登录的用户名，而 Password 则用于输入该用户所对应的密码。

需要注意的是窗口左侧的列表中的记录。当成功登录某个数据库之后，TOAD 会将用户名、密码和数据库信息一起存放于左侧列表中。在下次打开 TOAD 时，直接双击左侧列表中的选项即可完成登录。

3.3.2 开发一个简单的 PL/SQL 程序

打开 PL/SQL Developer，并利用用户 scott 登录数据库 ORCL。单击【新建】按钮，并选择【Command Window】菜单项，将弹出新的【Command Window】窗口，如图 3-17 所示。

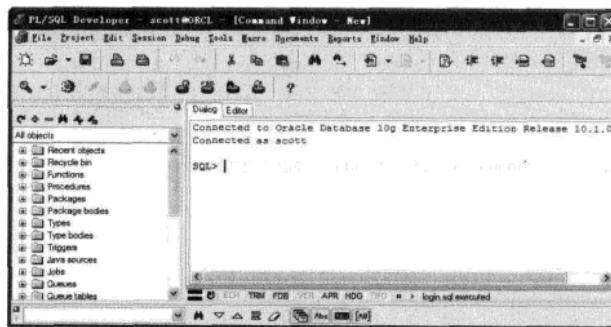


图 3-17 PL/SQL Developer 的【Command Window】窗口

此处的【Command Window】窗口，相当于 SQL Plus 中的命令行。在命令行中键入 edit 命令，打开新的代码编辑窗口，如图 3-18 所示。

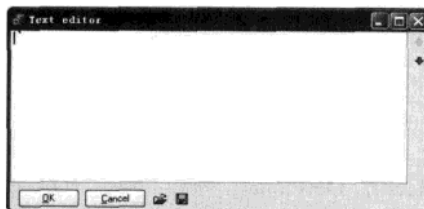


图 3-18 新的代码编辑窗口

可以在该编辑窗口中输入以下代码。

```
begin
  for employee in (select * from emp) loop
    dbms_output.put_line(employee.empno || '号员工是' || employee.ename);
  end loop;
end;
```

同时按下 Alt+O 组合键，关闭编辑窗口，回到【Command Window】窗口。

在【Command Window】窗口中键入如下命令。

```
SQL> set serverout on;
SQL> /
```

set serverout on，用于打开命令行的输出模式。字符“/”用于执行最近一次编辑缓冲区内

的代码块。输出结果如下：

SQL> /

7369 号员工是 SMITH
7499 号员工是 ALLEN
7521 号员工是 WARD
7566 号员工是 JONES
7654 号员工是 MARTIN
7698 号员工是 BLAKE
7782 号员工是 CLARK
7788 号员工是 SCOTT
7839 号员工是 KING
7844 号员工是 TURNER
7876 号员工是 ADAMS
7900 号员工是 JAMES
7902 号员工是 FORD
7934 号员工是 MILLER

PL/SQL procedure successfully completed



3.4 本章实例

SQL Plus 和 PL/SQL Developer 的【Command Window】窗口都可以实现 PL/SQL 代码块的执行，因此，可以在 SQL Plus 中进行代码的编写和执行。为了使读者熟悉 SQL Plus 中代码块的编写，本实例在 SQL Plus 中实现打印员工姓名的功能。

【范例 3-2】利用 SQL Plus 来执行范例 3-1 中的实例代码。

- (1) 利用用户 system 登录数据库 ORCL，如图 3-19 所示。
- (2) 利用 set serverout on 命令打开命令行的输出模式，如图 3-20 所示。

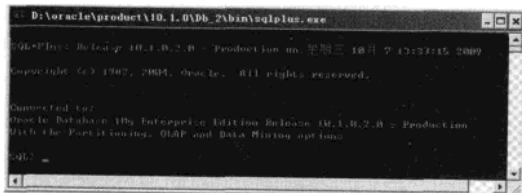


图 3-19 在 SQL Plus 中登录数据库 ORCL

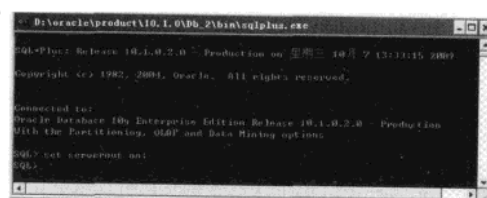


图 3-20 在 SQL Plus 中打开命令行输出

- (3) 键入或直接复制范例 3-2 中的代码至 SQL Plus 命令行，如图 3-21 所示。
- (4) 执行该代码块，将实现与范例 3-1 相同的功能，如图 3-22 所示。

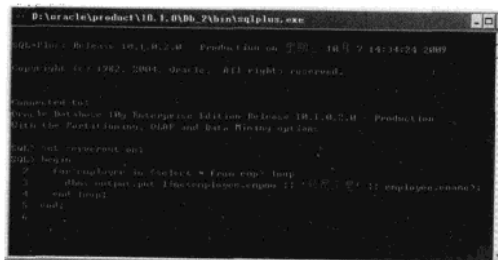


图 3-21 在 SQL Plus 中输入 PL/SQL 代码块



图 3-22 在 SQL Plus 中执行 PL/SQL 代码块



3.5 本章小结

本章介绍了过程化 SQL 语言 PL/SQL 及客户端交互工具 SQL Plus，并简要介绍了如何在 SQL Plus 中格式化输出结果。PL/SQL 是 Oracle 独有的，不能应用于其他数据库。目前最流行的 PL/SQL 的开发工具为 PL/SQL Developer 和 TOAD。本章对这两种工具进行了简要介绍，读者可以根据自己爱好选择使用。



3.6 习题

1. SQL Plus 与 PL/SQL 有何区别？
2. 在 SQL Plus 中，如果在一屏中无法显示查询结果，可以利用哪些手段进行格式化？
3. 简述利用 SQL Plus 登录数据库所使用的命令及命令的各个参数的意义。
4. 试着利用 SQL Plus 来创建一个新表。



第二篇 Oracle 数据库对象

第 4 章 Oracle 数据库

Oracle 数据库是存储数据的仓库,相对于其他数据库,Oracle 数据库具有更加强大的功能。而伴随着强大功能的是更多的复杂性。本章将着重讲述如何利用工具创建 Oracle 数据库,并解释 Oracle 数据库中容易混淆的术语。最后,讲述如何备份和恢复 Oracle 数据库。本章包括以下内容:

- 利用 Database Configuration Assistant 创建 Oracle 数据;
- 术语:数据库、数据库实例、SID 和 ORACLE_SID;
- 数据库的逻辑备份/恢复;
- 数据库的物理备份/恢复;
- 利用 PL/SQL Developer 备份数据库。

通过本章的学习,读者可以了解数据库的基本术语,并掌握最常用的数据库备份/恢复的方法。



4.1 创建 Oracle 数据库

本节主要讲述如何利用工具创建 Oracle 数据库。创建 Oracle 数据库的最常用工具为 Database Configuration Assistant (数据库配置助手)。在安装了 Oracle 服务器端之后,已经默认安装了该工具。可以通过如下步骤打开数据库配置助手。依次选择【开始】|【程序】|【Oracle 10g Home1】|【Configuration and Migration Tools】|【Database Configuration Assistant】命令。在打开数据库配置助手之后,可以按照以下步骤创建数据库。

① 在操作选择页面中选择【创建数据库】单选按钮,并单击【下一步】按钮,如图 4-1 所示。

② 在数据库模板中保持默认选项【一般用途】单选按钮,并单击【下一步】按钮,如图 4-2 所示。

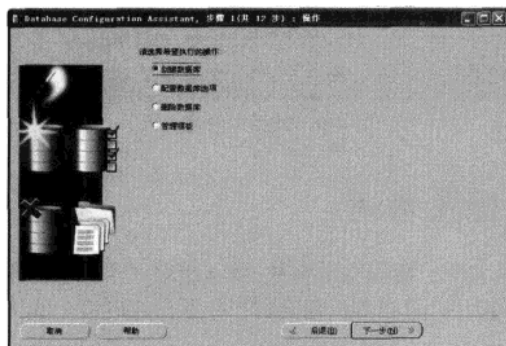


图 4-1 选择创建数据库选项



图 4-2 使用默认的数据库模板



③ 在数据库标识页面中指定 Oracle 全局数据库名称及数据库例程名称 SID。只有当局域网内存在多个同名数据库时，才需要使用域名+数据库名的命名方式来命名全局数据库名。在本例中使用名称为“test”，而 SID 默认情况下，与数据库同名。在指定全局数据库名及数据库例程名称之后，单击【下一步】按钮，如图 4-3 所示。

④ 在管理选项页面中，保持默认配置，并单击【下一步】按钮。

⑤ 在数据库身份证明页面中输入数据库访问密码。这里的数据库密码有两种方式：一为所有账户使用同一口令，二为各账户使用不同口令。在本例中，为所有账户输入统一口令，例如，abc123，并单击【下一步】按钮，如图 4-4 所示。

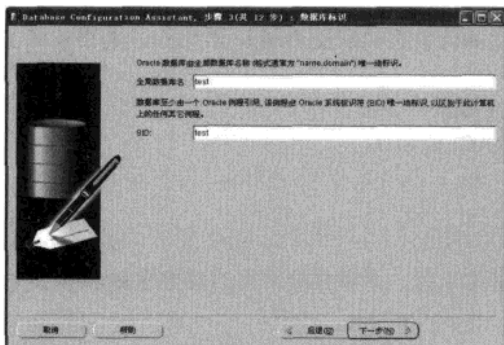


图 4-3 指定数据库的全局数据库名和 SID

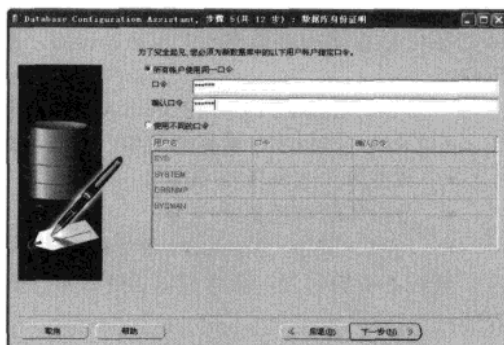


图 4-4 为所有账户输入统一口令

⑥ 其后的配置均可使用默认配置，因此，可以在存储选项页面中，直接单击【完成】按钮，以完成数据库的配置选项，如图 4-5 所示。

⑦ 在所有配置完成并进行确认之后，可以正式开始数据库的创建动作，如图 4-6 所示。

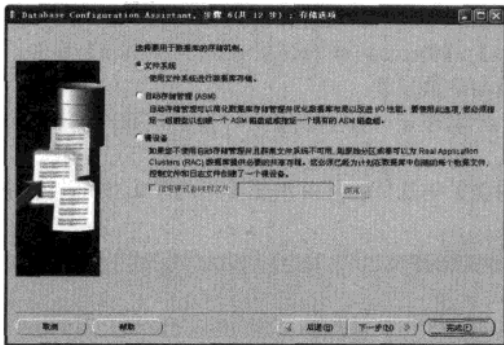


图 4-5 在存储选项页面中单击【完成】按钮

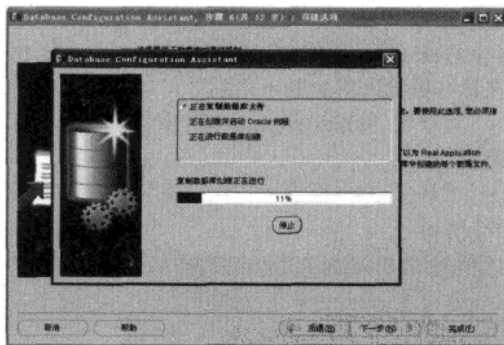


图 4-6 数据库创建页面



4.2 Oracle 数据库的相关术语

Oracle 数据库的相关术语包括以下几个：数据库、数据库实例及 ORACLE_SID。数据库是指真实的数据仓库，它包括了数据文件、控制文件、日志等，是实实在在存储在磁盘上的文件。而数据库实例则是数据库运行时，在内存中的副本。Oracle 数据库与外界环境进行交互，必须通过数据库实例。而 ORACLE_SID 则是一个系统环境变量。本节将详细讲述三者在操作系统中的实际体现。

4.2.1 数据库

数据库是实实在在存在的文件。例如，在创建数据库 test 之后，可以在磁盘中获得该数据库的相关文件。在 Oracle 的安装目录下，存在名为 oradata 的文件夹。该文件夹中，针对每个数据库都会有单独的目录来存储相关文件，如图 4-7 所示。

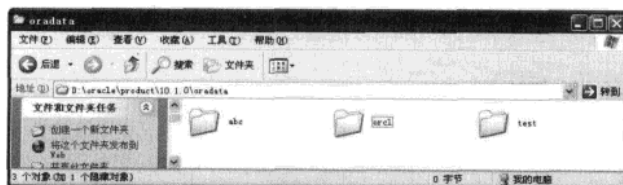


图 4-7 每个数据库相关文件的存储目录

从图 4-7 中可以看出，在整个数据库系统中，存在着三个数据库。数据库名分别是 abc、orcl、test。以数据库 test 为例，打开该目录。该目录下存储了数据库 test 的所有文件，如图 4-8 所示。

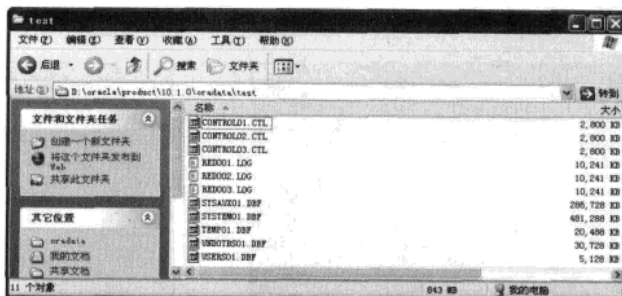


图 4-8 数据库 test 的所有相关文件

从图 4-8 中可以看出，一个数据库文件，大致包含了以下几种：数据文件（DBF）、控制文件（CTL）和日志文件（LOG）。

通过本例分析可知，Oracle 数据库是实实在在存在的物理文件。

4.2.2 数据库实例和 SID

每个 Oracle 数据库都会有一个数据库实例与之对应（非并行服务器结构）。外界环境要通过与数据库实例的交互来实现对数据库的操作。因此，在使用数据库之前，首先要启动数据库对应的实例。在 Windows 下，可以通过【服务】来查看数据库实例。例如，在创建数据库 test 之后，可以在【服务】中获得对应项，如图 4-9 所示。

名称	描述	状态	启动类型	登录为
OracleJobSchedulerTEST		已禁用	手动	本地系统
OracleOraDb10g_home1\SQLPlus	SQL...	已启动	自动	本地系统
OracleOraDb10g_home1\SNMPferEncapsulator			手动	本地系统
OracleOraDb10g_home1\SNMPferMasterAgent			手动	本地系统
OracleOraDb10g_home1\TNSListener		已启动	自动	本地系统
OracleServiceORCL		已启动	自动	本地系统
OracleServiceTEST		已启动	自动	本地系统
Performance Logs and Alerts	收...		手动	网络服务
Plug and Play	使...	已启动	自动	本地系统
Portable Media Serial Number Service	Net...		手动	本地系统
Print Spooler	将...	已启动	自动	本地系统

图 4-9 数据库实例对应于 Windows 服务



从图 4-9 可以看出, 数据库实例对应的 Windows 服务格式为 OracleServiceInstanceName。例如, 对于数据库 TEST, 在创建时, 其实例名为 TEST, 那么所对应的 Windows 服务名为 OracleServiceTEST。当数据库创建完毕之后, 会自动生成对应服务, 并且该服务会自动启动。而服务的启动类型为自动, 以保证在重启操作系统之后, 数据库实例也能自动启动。

数据库实例的唯一标识即 SID (System Identifier)。通常情况下, SID 等于实例名称。数据库实例是一个逻辑概念, 在实际开发中, 总是通过 SID 来引用数据库实例。

4.2.3 ORACLE_SID

ORACLE_SID 是指操作系统环境变量。Oracle 环境的初始化与该变量有关。例如, 当 SQL Plus 尝试登录数据库时, 如果没有显式指定该要连接的 SID(实例名), 那么将使用 ORACLE_SID 所指定的实例进行连接。例如, 利用如下 CMD 语句来连接数据库:

```
sqlplus / as sysdba
```

在登录 Oracle 数据库后, 查询当前数据库名及实例名, 如图 4-10 所示。

可见, 在没有显式指定登录的 SID 时, SQL Plus 将默认登录到数据库 ORCL。即 Oracle 安装时自带的数据库。

在 Windows 环境中, 可以在系统变量中添加新的变量 ORACLE_SID, 如图 4-11 所示。

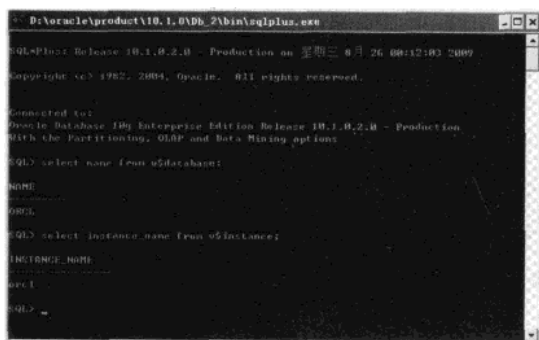


图 4-10 SQL Plus 登录默认数据库

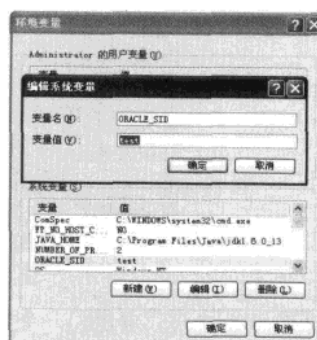


图 4-11 添加或修改系统变量 ORACLE_SID

注销系统, 以使系统变量的更改生效。再次利用 SQL Plus 连接 Oracle 数据库。

```
sqlplus / as sysdba
```

成功登录数据库之后, 可以查看当前数据库与当前实例名称。结果如图 4-12 所示。



图 4-12 SQL Plus 登录的默认数据库与 ORACLE_SID 相关

通过设置系统变量 ORACLE_SID 的值，可以直接影响 SQL Plus 连接的默认数据库实例。当然，ORACLE_SID 的应用并不仅限于此。通过本例，读者应该清楚 ORACLE_SID 是操作系统的环境变量，并且影响 Oracle 的运行环境。



4.3 Oracle 数据库的备份与恢复

在 Oracle 数据库的使用过程中，备份与恢复是经常遇到的操作。Oracle 中的备份分为两大类：逻辑备份和物理备份。其中物理备份又分为两类：冷备份和热备份。本节将简要讲述如何利用各种备份手段进行 Oracle 数据库的备份与恢复。

4.3.1 逻辑备份/恢复（导出/导入）

逻辑备份是指利用 exp 命令进行备份。利用该命令进行备份，简单易行，也不影响正常的数据库操作。因此，经常被作为日常备份的手段。exp 命令可以添加多个参数选项，以实现不同的导出策略。可以通过 exp -? 命令进行查看。其中，常用参数包括：owner、table 和 inctype。

1. 必备参数

对于一个导出命令，可以只使用必备参数，如范例 4-1 所示。

【范例 4-1】 演示导出命令的使用。

```
C:\Documents and Settings\Administrator>exp system/abc123 file=d:/b.dmp
```

【代码说明】 system/abc123 为登录数据库时所使用的用户名和密码；file=d:/b.dmp 指定数据导出所存放的文件完整路径。这里需要注意的是，该命令并未指定登录到哪个数据库实例，因此，将使用系统环境变量 ORACLE_SID 所指定的数据库实例。



注意： 另外一种特殊情况为，在环境变量列表中找到 ORACLE_SID，那么，可以在注册表中搜索 ORACLE_SID 项。Oracle 也会参考注册表中该项的值来设置环境。

2. owner 参数

owner 参数，可以指定一个用户名列表。导出时，将只导出用户名列表中用户所拥有的对象，如范例 4-2 所示。

【范例 4-2】 演示 owner 参数的使用。

```
C:\Documents and Settings\Administrator>exp system/abc123 owner=(test, oracle) file=d:/b.dmp
```

【代码说明】 owner=(test, oracle) 指定 exp 命令仅仅导出 test 和 oracle 两个用户所拥有的对象。如果某个用户不存在，例如，本例中用户 oracle 不存在，那么将给出相应警告，但不会影响对于用户 test 的对象的导出，如图 4-13 所示。

```
about to export specified users ...
EXP-00001: ORACLE is not a valid username
Exporting pre-schema procedural objects and actions
Exporting foreign function library names for user TEST
Exporting PUBLIC type definitions
Exporting private type definitions
Exporting object type definitions for user TEST
about to export TEST's objects ...
Exporting database links ...
Exporting sequence numbers ...
```

图 4-13 exp 命令的用户列表中的用户不存在



3. tables 参数

当使用 `exp` 命令时，同样可以指定 `tables` 参数。该参数用于指定导出哪些数据表。范例 4-3 演示了该参数的使用。

【范例 4-3】演示 `tables` 参数的使用。

```
exp system/abc123 tables=(people, employees) file=d:/b.dmp
```

【代码说明】`tables=(people, employees)` 指定了要导出的数据表列表，在 `exp` 命令执行时，将只导出用户 `system` 的 `people` 和 `employees` 表。如果要导出的表不存在，那么将给出警告信息。但不会影响其他表的导出工作。

4. 关于 `inctype` 参数

在 Oracle 9i 之前的版本中，可以利用 `inctype` 参数指定导出的增量类型。但是，在 Oracle 9i 及以后的版本中，该参数不再使用。范例 4-4 演示了在 Oracle 10g 中使用该参数的错误信息。

【范例 4-4】Oracle 10 中参数 `inctype` 已经废除。

```
C:\Documents and Settings\Administrator>exp system/abc123 incyte= complete
file=d:/b.dmp
```

【代码说明】在 `exp` 命令中使用 `inctype` 参数，Oracle 将抛出错误信息，如图 4-14 所示。

图 4-14 Oracle 10g 中不能使用 `inctype` 参数

对于逻辑备份，相应的恢复命令为 `imp`。如果数据库遭到破坏，可以利用如下命令进行恢复：

```
C:\Documents and Settings\Administrator>imp system/abc123 file=d:/b.dmp
```

`imp` 命令同样可以指定参数。例如，使用 `owner` 参数只导入特定用户的对象；使用 `tables` 参数只导入特定表，代码如下：

```
C:\Documents and Settings\Administrator>imp system/abc123 file=d:/b.dmp
tables=(people, employees)
```

该命令将只导入名为 `people` 与 `employees` 的表。

4.3.2 物理备份/恢复

物理备份，是指直接备份数据库的文件。物理备份又分为两种：冷备份和热备份。

1. 冷备份/恢复

冷备份是指在数据库关闭的状态下，备份所有的数据库文件。这些文件包括：所有数据文件、所有控制文件、所有联机 REDO LOG 文件和 `Init.ora` 文件（可选）。

【范例 4-5】演示数据库冷备份。

(1) 首先以管理员身份登录数据库，并将数据库关闭：

```
C:\Documents and Settings\Administrator>sqlplus / as sysdba
```



```
SQL> shutdown normal;
Database closed.
Database dismounted.
ORACLE instance shut down.
```

(2) 在关闭数据库之后,可以对其物理文件进行备份。这些物理文件默认处于{ORACLE_HOME}\product\10.1.0\oradata\test, 其中 test 为数据库名。因此,首先返回到 Windows 命令行下使用复制命令,或者在 SQL 命令行中添加 host 关键字直接使用主机命令:

```
SQL> host copy D:\oracle\product\10.1.0\oradata\test f:\backup\
D:\oracle\product\10.1.0\oradata\test\CONTROL01.CTL
D:\oracle\product\10.1.0\oradata\test\CONTROL02.CTL
D:\oracle\product\10.1.0\oradata\test\CONTROL03.CTL
D:\oracle\product\10.1.0\oradata\test\REDO01.LOG
D:\oracle\product\10.1.0\oradata\test\REDO02.LOG
D:\oracle\product\10.1.0\oradata\test\REDO03.LOG
D:\oracle\product\10.1.0\oradata\test\SYSAUX01.DBF
D:\oracle\product\10.1.0\oradata\test\SYSTEM01.DBF
D:\oracle\product\10.1.0\oradata\test\TEMP01.DBF
D:\oracle\product\10.1.0\oradata\test\UNDOTBS01.DBF
D:\oracle\product\10.1.0\oradata\test\USERS01.DBF
已复制      11 个文件。
```

此时,已经实现了整个数据库的冷备份。而冷备份的恢复则十分方便,只要数据库处于关闭状态,将备份的文件复制到原来的位置即可。

2. 热备份/恢复

数据库的热备份是指对处于启动状态下的数据库进行备份。热备份一个数据库,首先要保证数据库运行于归档模式,然后备份表空间的数据文件,最后备份控制文件。

【范例 4-6】演示数据库热备份。

(1) 在进行日志模式切换之前,必须将运行的数据库正常关闭。因此,首先应该以管理员身份登录数据库,关闭数据库。

```
SQL> shutdown immediate;
Database closed.
Database dismounted.
ORACLE instance shut down.
```

(2) 重新启动数据库实例,但是并不打开数据库。

```
SQL> startup mount;
ORACLE instance started.
```

```
Total System Global Area 171966464 bytes
Fixed Size                  787988 bytes
Variable Size               145488364 bytes
Database Buffers            25165824 bytes
Redo Buffers                 524288 bytes
Database mounted.
```

(3) 利用 alter 命令将数据库切换到归档模式。

```
SQL> alter database archivelog;
```

```
Database altered.
```

(4) 打开数据库,以便对数据库进行操作。

```
SQL> alter database open;
```

```
Database altered.
```





(5) 利用 `archive log list` 命令确认当前数据库处于归档模式。

```
SQL> archive log list;
Database log mode              Archive Mode
Automatic archival             Enabled
Archive destination            USE_DB_RECOVERY_FILE_DEST
Oldest online log sequence     538
Next log sequence to archive   540
Current log sequence           540
```

Archive Mode 表明当前的数据库处于归档模式。Oracle 数据库有联机重做日志，该日志用于记录用户的数据库操作，如插入、删除或更新数据。一般情况下，每个 Oracle 数据库至少含有两个联机重做日志组。当一个联机重做日志组被写满的时候，就会发生日志切换。另一个联机日志组成为当前使用的日志，继续记录用户操作。当前联机日志组写满后，会切换到第一个联机日志组，并覆写其中的数据。

如果数据库处于非归档模式，联机日志在切换时就会丢弃已有信息。而在归档模式下，当发生日志切换时，被切换的日志会首先进行归档，并将信息复制到其他目录。这样，不会造成联机日志信息的丢失。

(6) 将数据库中的表空间 `users` 设置为备份模式，代码如下：

```
SQL> alter tablespace users begin backup;
```

Tablespace altered.

(7) 复制实际的表空间的数据文件到备份目录下，代码如下：

```
SQL> host copy D:\oracle\product\10.1.0\oradata\test\users01.dbf d:\back;
已复制      1 个文件。
```

关闭表空间的备份模式，代码如下：

```
SQL> alter tablespace users end backup;
```

Tablespace altered.

(8) 以同样的方式备份数据库中的其他表空间的数据文件。

(9) 备份控制文件，代码如下：

```
SQL> alter database backup controlfile to 'F:\backup\TEST_BACKUP' reuse;
```

Database altered.

(10) 备份控制文件的创建脚本，代码如下：

```
SQL> alter database backup controlfile to trace;
```

Database altered.

此时会在 `{ORACLE_HOME}\admin\{INSTANCE_NAME}\udump` 目录下生成新的控制文件的跟踪文件。在本例中其路径为 `D:\oracle\product\10.1.0\admin\test\udump`。在目录中获得最新的跟踪文件，该文件记录了数据库控制文件的创建脚本。以下代码为文件片段：

```
-- The following commands will create a new control file and use it
-- to open the database.
-- Data used by Recovery Manager will be lost.
-- The contents of online logs will be lost and all backups will
-- be invalidated. Use this only if online logs are damaged.
-- After mounting the created controlfile, the following SQL
-- statement will place the database in the appropriate
-- protection mode:
-- ALTER DATABASE SET STANDBY DATABASE TO MAXIMIZE PERFORMANCE
STARTUP NOMOUNT
```

```

CREATE CONTROLFILE REUSE DATABASE "TEST" RESETLOGS ARCHIVELOG
    MAXLOGFILES 16
    MAXLOGMEMBERS 3
    MAXDATAFILES 100
    MAXINSTANCES 8
    MAXLOGHISTORY 454
LOGFILE
    GROUP 1 'D:\ORACLE\PRODUCT\10.1.0\ORADATA\TEST\REDO01.LOG' SIZE 10M,
    GROUP 2 'D:\ORACLE\PRODUCT\10.1.0\ORADATA\TEST\REDO02.LOG' SIZE 10M,
    GROUP 3 'D:\ORACLE\PRODUCT\10.1.0\ORADATA\TEST\REDO03.LOG' SIZE 10M
-- STANDBY LOGFILE
DATAFILE
    'D:\ORACLE\PRODUCT\10.1.0\ORADATA\TEST\SYSTEM01.DBF',
    'D:\ORACLE\PRODUCT\10.1.0\ORADATA\TEST\UNDOTBS01.DBF',
    'D:\ORACLE\PRODUCT\10.1.0\ORADATA\TEST\SYSAUX01.DBF',
    'D:\ORACLE\PRODUCT\10.1.0\ORADATA\TEST\USERS01.DBF',
    'F:\DATABASE\TMP\DATA_1.DBF',
    'F:\DATABASE\ORACLE\USER3_DATA.DBF',
    'F:\DATABASE\ORACLE\USER4_DATA.DBF',
    'F:\DATABASE\ORACLE\USER1_DATA.DBF',
    'F:\DATABASE\ORACLE\USER5_DATA.DBF',
    'F:\DATABASE\ORACLE\USER6_DATA.DBF',
    'F:\DATABASE\TMP\DATA_2.DBF'
CHARACTER SET ZHS16GBK
;
-- Commands to re-create incarnation table
-- Below log names MUST be changed to existing filenames on
-- disk. Any one log file from each branch can be used to
-- re-create incarnation records.
-- ALTER DATABASE REGISTER LOGFILE 'D:\ORACLE\PRODUCT\10.1.0\FLASH_
RECOVERY_AREA\TEST\ ARCHIVELOG\2009_08_28\01_MF_1_1_%U_.ARC';
-- ALTER DATABASE REGISTER LOGFILE 'D:\ORACLE\PRODUCT\10.1.0\FLASH_
RECOVERY_AREA\TEST\ ARCHIVELOG\2009_08_28\01_MF_1_1_%U_.ARC';
-- Recovery is required if any of the datafiles are restored backups,
-- or if the last shutdown was not normal or immediate.
RECOVER DATABASE USING BACKUP CONTROLFILE
-- Database can now be opened zeroing the online logs.
ALTER DATABASE OPEN RESETLOGS;
-- Commands to add tempfiles to temporary tablespaces.
-- Online tempfiles have complete space information.
-- Other tempfiles may require adjustment.
ALTER TABLESPACE TEMP ADD TEMPFILE 'D:\ORACLE\PRODUCT\10.1.0\ORADATA\TEST\
TEMP01.DBF'
    SIZE 20971520 REUSE AUTOEXTEND ON NEXT 655360 MAXSIZE 32767M;
-- End of tempfile additions.
--

```

复制该文件，并将该文件复制到备份目录下。至此，热备份的过程结束。

在备份成功之后，一旦出现数据库故障，即可以利用备份文件进行恢复工作。例如，如果数据文件 D:\oracle\product\10.1.0\oradata\test\users01.dbf 损坏，那么在启动数据库时会抛出错误。

```

SQL> startup mount;
ORACLE instance started.

```

```

Total System Global Area 171966464 bytes
Fixed Size                 787988 bytes
Variable Size             145488364 bytes
Database Buffers          25165824 bytes
Redo Buffers               524288 bytes
Database mounted.

```



```
SQL> alter database open;
alter database open
*
ERROR at line 1:
ORA-01157: cannot identify/lock data file 4 - see DBWR trace file
ORA-01110: data file 4: 'D:\ORACLE\PRODUCT\10.1.0\ORADATA\TEST\USERS01.DBF'
```

由于无法找到数据文件 **USERS01.DBF**，数据库启动失败。此时，需要将以前备份的数据文件复制到原目录中。

```
SQL> host copy f:\backup\users01.dbf D:\ORACLE\PRODUCT\10.1.0\ORADATA\
TEST\USERS01.DBF;
已复制      1 个文件。
```

此时，重新启动数据库。

```
SQL> alter database open;
alter database open
*
ERROR at line 1:
ORA-01113: file 4 needs media recovery
ORA-01110: data file 4: 'D:\ORACLE\PRODUCT\10.1.0\ORADATA\TEST\USERS01.DBF'
```

Oracle 在启动时，总是会检查控制文件中的标识（Checkpoint CNT 与 Checkpoint SCN）与数据文件中的标识是否相同。如果不同，则需要重新恢复数据文件，以同步控制文件中的标识与数据文件中的标识。恢复的命令如下：

```
SQL> recover datafile 'D:\ORACLE\PRODUCT\10.1.0\ORADATA\TEST\USERS01.DBF';
```

当然，也可以使用如下语句代替。

```
SQL> recover datafile 4;
```

此时，Oracle 可能会要求用户指定归档日志。归档日志默认存储在 {ORACLE_HOME}\flash_recovery_area\{DATABASE_NAME}\ARCHIVELOG\ 下，为 recover 命令选择最近的归档日志或使用 Oracle 建议的归档日志，如图 4-15 所示。

```
SQL> recover datafile 'D:\ORACLE\PRODUCT\10.1.0\ORADATA\TEST\USERS01.DBF';
ORA-00279: change 7753599 generated at 08/26/2009 23:28:01 needed for thread 1
ORA-00289: suggestion 1
D:\ORACLE\PRODUCT\10.1.0\FLASH_RECOVERY_AREA\TEST\ARCHIVELOGS\2009-08-27-01_RF_1
537.dbf...nnc
ORA-00289: change 7753599 for thread 1 is in sequence 8537
Specify log {<RET>-suggested filename | AUTO | CANCEL}:
```

图 4-15 选择归档日志

在选择了归档日志之后，Oracle 将使用该归档日志进行恢复。恢复成功之后，可以成功打开数据库。

```
SQL> alter database open;
```

```
Database altered.
```

备份控制文件的意义在于，当控制文件丢失时，可以将热备份的控制文件还原。而备份跟踪文件的意义在于，当备份的控制文件无法正常使用，利用跟踪文件重新创建控制文件。

4.3.3 利用 PL/SQL Developer 备份数据库

除了利用各种命令对数据库进行备份之外，还可以利用 PL/SQL Developer 进行备份。利用该工具进行备份，简单易学，而且不易出错。利用 PL/SQL Developer 进行备份的模式包括针对整个数据库、针对某个用户、针对某些特定表等。本节以备份某个用户对象为例，讲述如何

利用 PL/SQL Developer 进行备份。

【范例 4-7】 演示利用 PL/SQL Developer 备份用户 System 所有对象。

- (1) 打开 PL/SQL Developer，并利用 System 用户登录数据库 TEST。
- (2) 在左侧窗口的下拉菜单中选择 **【My objects】** 从而保证所有的操作都是针对当前用户的对象的。
- (3) 选择菜单栏中的 **【Tools】** 菜单下的 **【Export User Objects】** 菜单项，将弹出导出窗口，如图 4-16 所示。

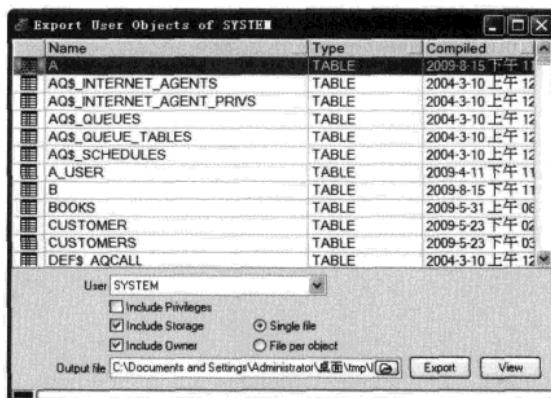


图 4-16 利用 PL/SQL Developer 备份用户的所有对象

(4) 选择列表中所有对象，并为 **【Output File】** 指定导出文件的路径。导出的文件实际上是一个 SQL 脚本文件。其中记录了当前用户的所有对象的创建脚本。一旦数据库出现故障，可以重建数据库，并在其中创建相应用户（System 用户无须创建，为数据库默认用户），然后利用该用户登录数据库，并执行该 SQL 脚本，即可实现数据库对象的重新创建。

(5) 除了备份数据库对象之外，还需要备份数据表中的数据。备份表中数据，需要使用 **【Tools】** 菜单下的 **【Export Tables】** 功能。具体的备份/恢复步骤，在 5.5 节将有详细描述。



4.4 本章实例

对于 exp 命令来说，不仅能够备份 ORACLE_ID 所指定的默认数据库，而且能够备份特定的数据库。

【范例 4-8】 演示利用 exp 命令备份特定数据库。

```
exp system/abc123@ORCL owner=(scott) file=d:/scott.dmp;
```

【代码说明】 system/abc123@ORCL 指定了欲进行备份的数据库 SID 及登录用户名、密码；owner=(scott) 指定了备份用户 scott 的所有对象；file=d:/scott.dmp 指定了备份的目标文件。



4.5 本章小结

本章介绍了如何利用工具创建 Oracle 数据库，并利用实例讲述了 Oracle 数据库中容易混淆的术语。在备份和恢复 Oracle 数据库的各种方法中，利用命令进行备份步骤烦琐，而且容易



出错。对于非专业的 DBA 人员，使用客户端工具，例如 PL/SQL Developer 提供的备份功能是较好的选择。



4.6 习题

1. 请简述 SID 与 ORACLE_SID 的区别。
2. 请简述 Oracle 逻辑备份的主要手段。
3. 请简述 Oracle 物理备份的主要手段。
4. 请简述利用 PL/SQL Developer 进行备份的主要步骤。



第 5 章 Oracle 数据表对象

与其他数据库（如 SQL Server、MySQL）不同，Oracle 数据库的下一层逻辑结构并非数据表，而是表空间；每个数据表都属于唯一的表空间。因此，本章将首先介绍表空间，然后介绍数据表及相关操作。本章的主要内容包括以下几个方面：

- 创建 Oracle 表空间；
- 创建 Oracle 数据表；
- 修改数据表结构；
- 删除数据表；
- 备份/恢复数据表；
- 特殊数据表。

通过本章学习，读者可以了解 Oracle 表空间和数据表的基本概念，掌握 Oracle 表存储的逻辑分层概念。同时，读者可以对创建、修改表空间和数据表的语法有全面的了解。



5.1 Oracle 表空间

表空间（TableSpace）是 Oracle 的开创性理念。表空间使得数据库管理更加灵活，而且极大地提高了数据库性能。

5.1.1 Oracle 表空间简介

与数据表相同，Oracle 表空间是一个逻辑对象，而非物理对象。所谓逻辑对象，是指数据库的组成部分，如表、索引、视图等。当我们使用 SQL 语句对数据库进行操作时，操作的都是逻辑对象，而非直接操作物理文件。Oracle 数据库、表空间与数据表都是逻辑对象，它们之间的关系如图 5-1 所示。

从图 5.1 可以看出，一个数据库可以有多个表空间，而一个表空间也可以有多个数据表。Oracle 表空间是 Oracle 数据库高性能的保证。使用表空间管理数据有以下好处。

1. 避免磁盘空间突然耗尽的风险

在创建表空间时，可以预先指定表空间的最大磁盘空间，避免数据库中数据无限扩张带来的磁盘空间的突然耗尽。

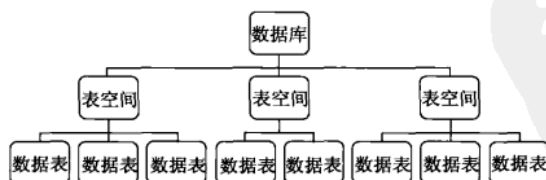


图 5-1 Oracle 数据库、表空间、数据表逻辑关系



2. 规划数据更灵活

当数据库中的数据量较大时，可以自定义划分标准，将各数据表划分到不同的表空间中。这类类似于操作系统的文件夹功能，方便查找和管理。

3. 提高数据库性能

对于访问频繁的数据表，可以将其放入单独的表空间中，并存储于高性能磁盘；将其他访问较少的的数据表规划于单独的表空间，并存储于性能相对较低的磁盘。以实现合理利用服务器资源，最大限度提高数据库性能。

另外一种应用场景为，将更新频繁的数据表规划于单独的表空间，而很少更新的数据表规划于其他的表空间。那么，在备份数据库时，可以针对不同的表空间指定不同的备份周期，在最大程度上减小备份数据库占用的系统资源。

4. 提高数据库安全性

不同的表空间对应着不同的物理文件，当某个表空间的物理文件损坏时，不会影响其他表空间的正常使用，提高了数据库的安全性。

另外，可以将数据库实际数据与日志规划为不同的表空间，并存储于不同的磁盘。即使数据库所在的磁盘出现问题，还可利用另一磁盘中的日志文件恢复数据库，从而有效降低了数据丢失的风险。

5.1.2 创建 Oracle 表空间

要想发挥 Oracle 表空间的强大性能和灵活性，首先应该合理地规划和创建表空间。本节将按照用户标准划分并创建单独的表空间。

1. 创建一个简单的表空间

Oracle 表空间是一个逻辑概念，创建时需要指定物理文件，即为实际数据分配磁盘空间。表空间的物理文件被称为数据文件（datafile）；与此同时，应同时指定数据文件的初始大小。

【范例 5-1】 创建一个名为 USER1 的表空间，其 SQL 语句如下：

```
create tablespace user1 datafile 'f:\database\oracle\user1_data.dbf' size 20M
```

【代码解析】 create tablespace 是创建表空间命令；user1 是新建表空间的名称；'f:\database\oracle\user1_data.dbf' 是表空间对应的数据文件的完整路径；20M 是数据文件的初始化大小。



注意： 数据文件的初始化大小是必需参数，其完整路径必须包含在单引号中。

2. 指定数据文件的可扩展性

表空间设计理念灵活性的一个方面在于数据文件的可扩展性。当存储在某个数据文件中的数据量超过了其初始大小时，数据文件可以进行自动扩展。要想实现该功能，在创建表空间时应该使用 autoextend 选项。

【范例 5-2】 指定数据文件可以自动扩展，其 SQL 语句如下：

```
create tablespace user2 datafile 'f:\database\oracle\user2_data.dbf' size 20M  
autoextend on
```

【代码解析】 autoextend 选项的值为“on”，表空间的数据文件是可以自动扩展的。

当然，也可以关闭自动扩展功能，那么当数据文件中的数据量达到其初始大小时，将无法写入数据。要关闭自动扩展功能，应当将 `autoextend` 选项的值设为“off”，相应的 SQL 语句如下所示：

```
create tablespace user2 datafile 'f:\database\oracle\user2_data.dbf' size 20M
autoextend off
```

3. 指定数据文件的增长幅度

当表空间创建时，使用自动增长数据文件大小的确带来了灵活性。数据文件自动增长时，每次默认增长 64K，但是，当某个表空间数据更新很快，数据量的增长也很快时，就会频繁地要求增加数据文件的大小。此时，应该为每次的增长幅度设定一个合理的值，避免频繁执行增加数据文件大小的动作，影响数据库性能。

【范例 5-3】 设定自动增长幅度应该使用 `next` 选项，相应的 SQL 语句如下：

```
create tablespace user3 datafile 'f:\database\oracle\user3_data.dbf' size 20M
autoextend on next 5M
```

【代码解析】 在本例中，将数据文件自动增长的幅度指定为 5MB（即 5M 字节）。数据文件的大小和自动增长幅度的单位只有 KB 和 MB 两种。

4. 指定数据文件的最大尺寸

数据文件可以自动增长，但是无限制的增长往往带来风险。很多情况下，某台服务器上可能同时运行着多个系统，如邮件服务器和数据库服务器可能为同一台机器，那么如果允许数据库无限增长，当硬盘空间耗尽时，不但会影响数据库的正常使用，邮件系统也不可避免地会受到影响。因此，除非特殊需要，应为每个表空间的数据文件设定最大尺寸。

【范例 5-4】 为数据文件设定最大尺寸，应使用 `maxsize` 选项，相应的 SQL 语句如下所示：

```
create tablespace user4 datafile 'f:\database\oracle\user4_data.dbf' size 20M
Autoextend On Next 5m Maxsize 500M
```

【代码解析】 在本例中，将数据文件的最大尺寸设为 500MB。

如果不限制数据文件的最大尺寸，应该使用 `unlimited` 来代替实际值。相应的代码应该改写为：

```
create tablespace user4 datafile 'f:\database\oracle\user4_data.dbf' size 20M
autoextend on next 5m maxsize unlimited
```



说明：在创建表空间时，还有其他选项，如 `reuse` 等，由于并不常用，读者可查看 Oracle 帮助文档进行了解。

5. 查看表空间是否创建成功

表空间成功创建后，会在数据库系统表中添加相应的记录，并且创建相应的物理文件。可以通过查看视图 `dba_data_files` 中的记录和实际数据文件存在性，来判断表空间是否创建成功。

【范例 5-5】 查看表空间信息的 SQL 语句如下：

```
select file_name, tablespace_name from dba_data_files order by file_name
```

【代码解析】 视图 `dba_data_files` 存储了数据库中所有表空间的数据文件信息。其中 `tablespace_name` 字段存储了表空间的名称；`file_name` 则存储了对应表空间的数据文件的完整路径。



执行范例 5-5 的 SQL 语句，查询结果如图 5-2 所示。

FILE_NAME	TABLESPACE_NAME
D:\ORACLE\PRODUCT\10.1.0\ORADATA\TEST\SYSAUX01.DBF	SYSAUX
D:\ORACLE\PRODUCT\10.1.0\ORADATA\TEST\SYSTEM01.DBF	SYSTEM
D:\ORACLE\PRODUCT\10.1.0\ORADATA\TEST\UNDOTBS01.DBF	UNDOTBS1
D:\ORACLE\PRODUCT\10.1.0\ORADATA\TEST\USERS01.DBF	USERS
F:\DATABASE\ORACLE\USER1_DATA.DBF	USER1
F:\DATABASE\ORACLE\USER2_DATA.DBF	USER2
F:\DATABASE\ORACLE\USER3_DATA.DBF	USER3
F:\DATABASE\ORACLE\USER4_DATA.DBF	USER4

图 5-2 查看表空间信息

检查数据文件的存在性。依据 file_name 字段提供的信息，在 F:\DATABASE\ORACLE 下可以找到与各表空间同名的数据文件，如图 5-3 所示。

地址	名称	大小	类型	修改日期
F:\Database\Oracle	USER1_DATA.DBF	20,488 KB	DBF 文件	2009-5-6 1:07
	USER2_DATA.DBF	20,488 KB	DBF 文件	2009-5-6 0:28
	USER3_DATA.DBF	20,488 KB	DBF 文件	2009-5-6 21:06
	USER4_DATA.DBF	20,488 KB	DBF 文件	2009-5-6 21:06
	USER5_DATA.DBF	20,488 KB	DBF 文件	2009-5-6 21:06
	USER6_DATA.DBF	20,488 KB	DBF 文件	2009-5-6 21:06

图 5-3 验证表空间对应物理文件的存在性



说明：以上创建的表空间 USER1-USER4 只是按照用户原则来规划并创建，并未将其分配给特定用户。如何指定某个用户所创建的数据表存储于特定表空间将在后面的内容中进行介绍。

6. 为一个表空间创建多个数据文件

当然，一个表空间可以有多个数据文件，为一个表空间创建多个数据文件需要指定多个数据文件的完整路径和详细的选项参数。各数据文件参数之间使用逗号（“，”）分隔。

【范例 5-6】 为一个表空间创建多个数据文件的 SQL 语句如下所示：

```
create tablespace multiple_data_file datafile 'f:\database\tmp\data_1.dbf' size 1M , 'f:\database\tmp\data_2.dbf' size 5M
```

【代码解析】 该 SQL 语句创建了一个名为 multiple_data_file 的表空间。它包含了两个数据文件——data_1.dbf 和 data_2.dbf。其中，data_1.dbf 的初始大小为 1MB，data_2.dbf 的初始大小为 5MB。

表空间的相关操作一般包括，查看、修改和删除。下面几节将通过简单的实例来介绍表空间的基本操作。

5.1.3 查看表空间

每个数据库在创建时都会自动创建几个表空间，这些表空间和用户创建的表空间信息，都存储在数据词典中。可以通过查询视图 dba_tablespaces 和视图 dba_data_files 来获得数据库的表空间信息。dba_tablespaces 可以用来查看所有表空间的基本信息；而 dba_data_files 可以用来查看相关数据文件的信息。

【范例 5-7】查看所有表空间信息，可以利用如下 SQL 语句：

```
select tablespace_name, status, allocation_type from dba_tablespaces
```

执行结果如图 5-4 所示。

TABLESPACE_NAME	STATUS	ALLOCATION_TYPE
SYSTEM	ONLINE	SYSTEM
UNDOTBS1	ONLINE	SYSTEM
SYSAUX	ONLINE	SYSTEM
TEMP	ONLINE	UNIFORM
USERS	ONLINE	SYSTEM
USER1	ONLINE	SYSTEM
USER2	ONLINE	SYSTEM
USER3	ONLINE	SYSTEM
USER4	ONLINE	SYSTEM
USER5	ONLINE	SYSTEM
USER6	ONLINE	SYSTEM

图 5-4 查看数据库表空间的相关信息

其中，SYSTEM、UNDOTBS1、SYSAUX、TMP 和 USERS 这 5 个表空间是 Oracle 数据库自动创建的。SYSTEM 是最重要的表空间，其中存储了数据库运行的最基本信息；SYSAUX 是 Oracle 10g 的新特性，用于分担在早期版本中 SYSTEM 表空间的部分工作；UNDOTBS1 是系统回滚段表空间，用于回滚操作时的数据存储；TEMP 表空间为数据库进行排序运算、索引管理、查看视图等操作提供临时运算空间；而 USERS 表空间则是整个数据库的默认表空间，也就是说，如果某个普通用户（非系统用户）创建时没有为其分配表空间，则使用 USERS 这个表空间。

5.1.4 修改数据库默认表空间

默认表空间是相对用户来说的，也就是说，每个用户登录 Oracle 数据库时，都有一个默认的工作空间。当进行与表空间相关操作（例如，创建数据表，每个数据表都隶属于一个表空间）时，如果未显式指定表空间（例如，创建数据表，未显式指定将表创建于哪个表空间中），则该操作将作用于用户的默认表空间。

【范例 5-8】查询每个用户的默认表空间，可以使用如下 SQL 语句：

```
select user_id, username, default_tablespace from dba_users
```

查询的部分结果如图 5-5 所示。

USER_ID	USERNAME	DEFAULT_TABLESPACE
5	SYSTEM	SYSTEM
0	SYS	SYSTEM
46	OLAPSYS	SYSAUX
35	SI_INFORMTH_SCHEMA	SYSAUX
56	MGMT_VIEW	SYSAUX
34	ORDPLUGINS	SYSAUX
51	WKPROXY	SYSAUX
58	TEST	USERS
42	XDB	SYSAUX
54	SYSMAN	SYSAUX
19	DIP	USERS

图 5-5 查询用户与默认表空间



说明：从图 5-5 可以获得每个用户的默认表空间。例如，SYSTEM 和 SYS 这两个系统用户的默认表空间为 SYSTEM

一个数据库除了系统用户外，可能有多个普通用户。如果创建用户时，未为该用户显式指定默认表空间，则它们默认使用统一的表空间——这就是数据库默认表空间。USERS 表空间是整个数据库的默认表空间，如果需要将数据库的默认表空间指定为其他，则可以使用 alter 命令进行重设。

【范例 5-9】修改数据库默认表空间的 SQL 语句如下：

```
alter database default tablespace user1
```



说明：alter database 是修改数据库属性命令，default tablespace user1 则指定了新的表空间 user1。该语句执行后，数据库的默认表空间被设定为 user1。

重设之后，无论是新建的用户还是已有用户，创建时未显式指定默认表空间者，其默认表空间将使用新的数据库默认表空间。

5.1.5 修改表空间名称

在 Oracle 10g 中，新增了修改表空间名称这一特性。修改表空间名称应该使用 rename 命令。

【范例 5-10】演示如何将表空间 user2 更名为 user20。

```
alter tablespace user2 rename to user20
```

【代码说明】user2 为原表空间名称，而 user20 则为新的表空间的名称。

查看更新后的表空间信息：

```
select tablespace_name, status, allocation_type from dba_tablespaces
```

查询结果如图 5-6 所示。

TABLESPACE_NAME	STATUS	ALLOCATION_TYPE
SYSTEM	ONLINE	SYSTEM
UNDOTBS1	ONLINE	SYSTEM
STSAUX	ONLINE	SYSTEM
TEMP	ONLINE	UNIFORM
USERS	ONLINE	SYSTEM
USER1	ONLINE	SYSTEM
USER20	ONLINE	SYSTEM
USER3	ONLINE	SYSTEM
USER4	ONLINE	SYSTEM
USER5	ONLINE	SYSTEM
USER6	ONLINE	SYSTEM

图 5-6 查看表空间更名后的信息



说明：从图 5-6 的查询结果可以看出表空间更名成功。

再次查看数据文件是否进行了相应的更名操作。相应的 SQL 语句如下：

```
select file_name, tablespace_name from dba_data_files order by file_name
```


数据查询结果如图 5-7 所示。

FILE_NAME	TABSPACE_NAME
D:\ORACLE\PRODUCT\10.1.0\ORADATA\TEST\SYSAUX01.DBF	SYSAUX
D:\ORACLE\PRODUCT\10.1.0\ORADATA\TEST\SYSTEM01.DBF	SYSTEM
D:\ORACLE\PRODUCT\10.1.0\ORADATA\TEST\UNDOTBS01.DBF	UNDOTBS1
D:\ORACLE\PRODUCT\10.1.0\ORADATA\TEST\USERS01.DBF	USERS
F:\DATABASE\ORACLE\USER1_DATA.DBF	USER1
F:\DATABASE\ORACLE\USER2_DATA.DBF	USER20
F:\DATABASE\ORACLE\USER3_DATA.DBF	USER3
F:\DATABASE\ORACLE\USER4_DATA.DBF	USER4
F:\DATABASE\ORACLE\USERS_DATA.DBF	USERS
F:\DATABASE\ORACLE\USER6_DATA.DBF	USER6

图 5-7 查看表空间重命名对数据文件的影响



说明：表空间重命名并不对数据文件产生影响。

最后，需要注意的是，不能对数据的系统表空间进行重命名，例如 SYSTEM，SYSAUX 等无法进行重命名。

5.1.6 删除表空间

如果某个表空间没有存在的必要，那么可以执行删除表空间命令，以释放磁盘空间。删除表空间的命令为 drop tablespace。删除表空间有两种方式，一种是仅仅删除其在数据库中的记录，另一种是将记录和数据文件一起删除。

【范例 5-11】 仅删除其在数据库中记录的 SQL 语句如下：

```
drop tablespace user20
```

【代码说明】 drop tablespace 命令用于删除表空间，user20 则为要删除的表空间名称。

再次查看表空间信息：

```
select tablespace_name, status, allocation_type from dba_tablespaces
```

查询结果如图 5-8 所示。

TABSPACE_NAME	STATUS	ALLOCATION_TYPE
SYSTEM	ONLINE	SYSTEM
UNDOTBS1	ONLINE	SYSTEM
SYSAUX	ONLINE	SYSTEM
TEMP	ONLINE	UNIFORM
USERS	ONLINE	SYSTEM
USER1	ONLINE	SYSTEM
USER3	ONLINE	SYSTEM
USER4	ONLINE	SYSTEM
USERS	ONLINE	SYSTEM
USER6	ONLINE	SYSTEM

图 5-8 删除表空间 USER20 后的查询结果



说明：从图 5-8 可以看出，已在数据库中成功删除了表空间 USER20 的信息。

此时，查看其数据文件，会发现仍然存在。只要我们在删除时添加 including 选项，即可



将其数据文件一并删除。

【范例 5-12】删除表空间及其数据文件的 SQL 语句如下：

```
drop tablespace user20 including contents and datafiles
```

【代码说明】including contents and datafiles 表明，当删除该表空间时，应将数据文件一并删除。



5.2 创建 Oracle 数据表

Oracle 表空间的下一层逻辑结构即为数据表。数据表也是各种数据库中共有的、开发人员和 DBA 最常打交道的数据库对象。本节着重介绍如何创建 Oracle 数据表。

5.2.1 利用工具创建数据表

利用工具创建数据表，操作简单、直观、易于掌握。很多数据库管理工具都提供了图形化界面来创建数据表，如 MS SQL Server 企业管理器。针对 Oracle 数据库，PL/SQL Developer 是一个不错的选择。使用 PL/SQL Developer 创建数据表可以遵循以下步骤：

- ① 在左侧对象清单栏中，右键单击【Tables】分支，在弹出的菜单中选择【New】命令。
- ② 在右侧窗口中选择【General】选项卡，并填入新建表的表名，如“T_USER”，如图 5-9 所示。

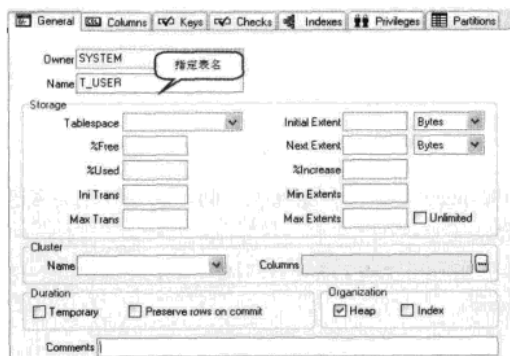


图 5-9 为新建表输入表名



说明：本页中，可以为新表选择存储于哪个表空间中，在此留空，表示使用默认表空间。

- ③ 选择【Columns】选项卡，为表创建多个列。在本例中为表 T_USER 创建“USER_ID”、“USER_NAME”、“USER_EMAIL”三个列，如图 5-10 所示。

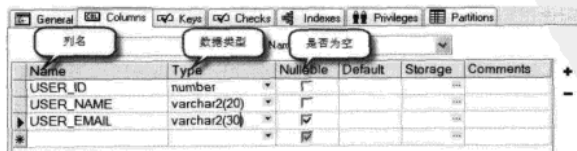



图 5-10 为表创建多个列

在图 5-10 中, Name 为列名; Type 为列的数据类型, number 为数值型, varchar2(20) 表示 USER_NAME 列为可变长度 20 的字符串类型; Nullable 指定列的内容能否为空, 其中 USER_ID 和 USER_NAME 不能为空, 而 USER_EMAIL 允许为空。



说明: 对于已经增加的列, 可以选中该列, 单击右侧的 , 删除该列。

④ 单击窗口底部的【Apply】按钮, 即可创建名为 T_USER 的数据表, 如图 5-11 所示。

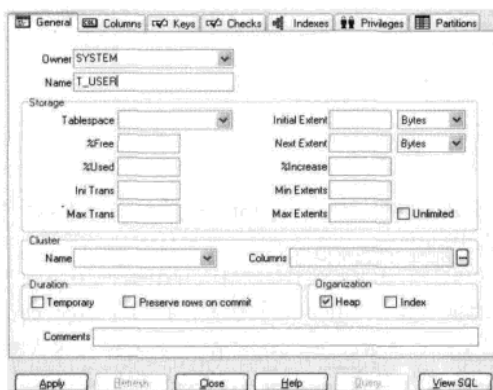


图 5-11 创建数据表

执行完以上步骤, 一个新的数据表 T_USER 创建成功。

5.2.2 利用工具查看数据表

在创建了数据表 T_USER 后, 同样可以在 PL/SQL Developer 中查看该表的信息。操作步骤如下。

① 在 PL/SQL Developer 的左侧窗口中展开【Tables】分支, 拖拽滚动条, 找到 T_USER 表。右击 T_USER 表, 在弹出的菜单中, 选择【Properties】菜单。

② 单击【Properties】命令, 将弹出表 T_USER 的属性信息, 如图 5-12 所示。

从图 5-12 可以看出, 表 T_USER 所属表空间为 SYSTEM。由于登录 Oracle 数据库所用的账号为 system, 创建数据表时, 没有显式指定其所属的表空间, 所以将使用当前用户的默认表空间。正如 5.1.3 节所述, 用户 system 的默认表空间为 SYSTEM, 所以 T_USER 表所属的表空间正是 SYSTEM。

③ 在右侧窗口中展开【T_USER】分支, 可以查看表 T_USER 相关的各数据库对象列表, 如图 5-13 所示。

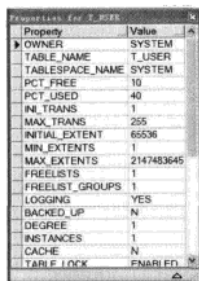


图 5-12 表 T_USER 的属性信息

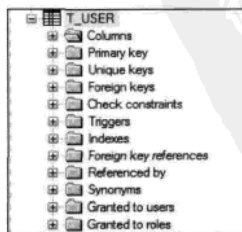


图 5-13 数据表相关对象列表



④ 展开【Columns】分支，可以查看 T_USER 表的列清单，如图 5-14 所示。

⑤ 与数据表属性类似，通过右键单击某个列对象，选择其【Properties】属性，可以查看某列的属性。以 USER_ID 为例，其属性信息如图 5-15 所示。



图 5-14 查看数据表 T_USER 的列清单

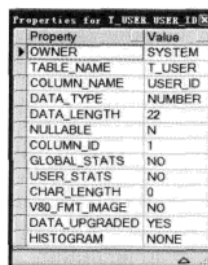


图 5-15 列 USER_ID 的属性信息

从图 5-15 可以看出，该列所属的表为 T_USER；数据类型为数值型（NUMBER）；长度为 22 字节；列的内容不允许为空。



小技巧：通过右键单击【Columns】分支，执行【Copy comma separated】菜单命令，可以将所有列名复制出来，并以逗号作为分隔符。这在数据表的列很多、使用 INSERT 语句时，最为有用。图 5-16 演示了这种用法。

在文本编辑器中，执行“粘贴”命令，即可将所有列名以文本方式复制出来。对于表 T_USER，复制出的列名清单，代码如下：

```
user_id,
user_name,
user_email
```

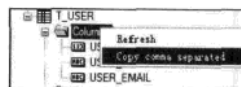


图 5-16 复制列清单

5.2.3 利用命令创建数据表

利用命令同样可以创建数据表，其效果与利用工具完全相同。创建数据表的命令为 CREATE TABLE。

【范例 5-13】创建与 5.2.1 节相同的数据表的命令如下：

```
create table t_user (user_id number not null, user_name varchar2(20) not null,
user_email varchar2(30))
```

【代码说明】create table 向数据库发出建表命令；所有列的描述应该包括在括号中——user_id 为列名，number 为列的数据类型，not null 表示该列不能为空；其他列与 user_id 类似。

5.2.4 利用命令查看表结构

同样可以通过命令方式来获得数据表的信息。例如，可以通过查询语句获得所属表空间。

【范例 5-14】利用 SQL 语句查看数据表所属的表空间的 SQL 语句如下：

```
select table_name, tablespace_name from user_tables where table_name='t_user'
```

【代码说明】视图 user_tables 可以用来查看所有用户表的基本信息。table_name 列为表名，tablespace_name 列为表所属表空间的名称。

查询结果如图 5-17 所示。

【范例 5-15】可以在 SQL Plus 或 PL/SQL Developer 的【Command Window】执行如下 SQL 语句来查看数据表信息。

```
sql> describe t_user;
```

【代码说明】：describe 命令用于描述表结构，且只用于描述表对象，不能用于数据库其他对象。describe 命令可以使用 desc 的简写形式来代替。

执行的结果如图 5-18 所示：

TABLE_NAME	TABLESPACE_NAME
T_USER	SYSTEM

图 5-17 查询表所属的表空间

SQL> DESCRIBE T_USER;				
Name	Type	Nullable	Default	Comments
USER_ID	NUMBER			
USER_NAME	VARCHAR2(20)			
USER_EMAIL	VARCHAR2(30)	Y		

图 5-18 在【Command Window】下查看表结构



5.3 修改 Oracle 数据表结构

数据表一旦创建，并不是一成不变的，修改数据表结构也成为开发人员必不可少的知识。本章将从工具方式和命令方式两个角度讲述如何修改数据表结构。

5.3.1 利用工具修改数据表结构

利用工具修改数据表结构，操作简单、直观。最常用的工具自然是 PL/SQL Developer。以 5.2 节创建的表 T_USER 为例，利用 PL/SQL Developer 修改数据表结构的步骤如下。

- ① 利用 PL/SQL Developer 登录 Oracle 数据，并在左侧对象列表中找到 T_USER 表。
- ② 右键单击 T_USER 表，在弹出的快捷菜单中选择【Edit】菜单项。右侧窗口将出现对 T_USER 表的编辑窗口。默认选项卡为【General】，如图 5-19 所示。

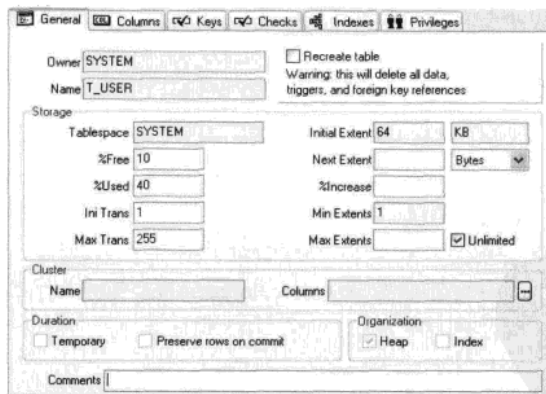


图 5-19 表 T_USER 的【General】信息

从图 5-19 可以看出表 T_USER 的基本信息，包括拥有者（Owner）、表名（Name）、表空间（Tablespace）都是不可更改的。

- ③ 选择【Columns】选项卡，进入列的编辑页面，如图 5-20 所示。



Name	Type	Nullable	Default	Storage	Comments
USER_ID	NUMBER	<input type="checkbox"/>			
USER_NAME	VARCHAR2(20)	<input type="checkbox"/>			
USER_EMAIL	VARCHAR2(30)	<input checked="" type="checkbox"/>			

图 5-20 表 T_USER 的【Columns】编辑页面

在该页面中，可以直接修改列名，例如，将 USER_EMAIL 的列名修改为 EMAIL；可以修改列的数据类型，例如，将 USER_NAME 的长度修改为 15 字节；可以直接增加新列，例如，增加新列 REMARK 作为备注。修改后的表结构如图 5-21 所示。

Name	Type	Nullable	Default	Storage	Comments
USER_ID	NUMBER	<input type="checkbox"/>			
USER_NAME	VARCHAR2(15)	<input type="checkbox"/>			
EMAIL	VARCHAR2(30)	<input checked="" type="checkbox"/>			
REMARK	VARCHAR2(50)	<input checked="" type="checkbox"/>			

图 5-21 修改后的表结构

④ 单击【Apply】按钮，提交对数据表结构的修改。

5.3.2 利用命令修改数据表结构

使用命令方式同样可以修改数据表结构。修改数据表结构的 SQL 命令为 alter table。

【范例 5-16】可以利用 RENAME 选项，对列名进行修改。以修改列名 USER_EMAIL 到 EMAIL 为例，相应的 SQL 语句如下：

```
alter table t_user rename column user_email to email
```

【代码说明】alter table t_user 指定修改的对象为表 t_user，正如 alter database 指定修改的对象为数据库；rename column user_email 指定重命名列 user_email，to email 指定列的新名。



注意：列更名时，不会影响数据库中的记录；但注意不能将列名更新为与其他已存在列名相同。

【范例 5-17】可以利用 modify 选项，对列的属性进行修改。以修改列 USER_NAME 的长度为例，相应的 SQL 语句如下：

```
alter table t_user modify (user_name varchar2(15))
```

【代码说明】modify 选项，不必像 rename column 一样，指定作用的对象为 column，而是默认为修改列。小括号内的 user_name varchar2(15) 指定要修改的列 user_name 和列的新属性 varchar2(15)。

修改列的属性，可能会影响数据库中的数据。以修改列长度为例，如果数据库中已存在的数据长度为 20 字节，那么将列的长度修改为 15 字节，将会导致修改失败。

例如，在 t_user 的列 email 长度为 20 字节的情况下插入一条记录。

```
insert into t_user (user_id, user_name, email) values (1, 'alex', 'alex200905@163.com')
```

尝试将列长度修改为 15 字节：

```
alter table t_user modify (email varchar2(15))
```

将会出现如图 5-22 所示的错误提示。

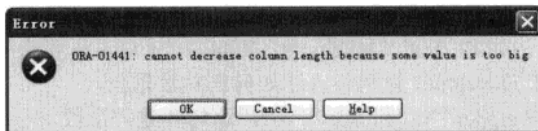


图 5-22 修改列长度不当的错误提示

【范例 5-18】 Oracle 允许一次修改多个列属性。将列 `user_name` 的长度修改为 15 字节，同时将列 `user_email` 长度修改为 30 字节的 SQL 语句如下：

```
alter table t_user modify (user_name varchar2(15), email varchar2(30))
```

【代码说明】 `user_name` 和 `email` 两个列属性之间需要使用逗号进行分隔。

【范例 5-19】 使用 `alter table` 命令同样可以为表添加一列。例如，可以为数据表 `t_user` 增加一个 `remark` 列，相应的 SQL 语句如下：

```
alter table t_user add (remarks varchar2(50))
```

【代码说明】 `add` 选项同样无须使用 `column` 关键字，直接将要增加的列名和期望属性用小括号括起来。与修改列属性相同，用逗号分隔的多个列及属性，可以用来一次性为表添加多个列。

【范例 5-20】 删除表中的某列也是常见操作。删除表中的列需要用到 `drop` 选项。如果需要将范例 5-19 中增加的 `remark` 列删除，那么可以使用如下 SQL 语句。

```
alter table t_user drop column remark
```

【代码说明】 `drop column` 选项用于删除某列，`remark` 为删除的列名。



说明：对于 `add`、`modify` 都无须添加 `column` 关键字，而 `drop` 则需要。这是因为修改一个表时，删除操作有可能针对表的某些约束，所以必须添加 `column` 关键字，表示要删除的是一个列。

【范例 5-21】 `alter table` 命令除了对列进行修改外，还可以对表本身的属性进行修改。例如，可以利用 `rename` 选项修改表名。将表 `t_user` 的表名修改为 `t_users`，可以使用如下 SQL 语句。

```
alter table t_user rename to t_users
```

【代码说明】 `rename to` 选项直接将原表名修改为新表名。注意和修改列名的区别。



注意：不要轻易修改一个表的表名，因为所有针对该表的操作都是以表名作为标识的，修改表名有可能影响已有应用程序的运行。



5.4 删除数据表

对于维护数据库，另一个可执行的操作就是删除数据表。如果不再需要某个数据表，又不想其一直占用数据库资源，即可执行删除操作。本节将结合工具和命令方式，讲述如何删除



数据表。

5.4.1 利用工具删除数据表

许多数据库工具，例如 PL/SQL Developer、TOAD for Oracle 都会包含直观的数据表删除功能。本节将以 PL/SQL Developer 为例，讲述如何删除数据表。

首先在 PL/SQL Developer 中，利用如下 SQL 语句新建一个数据表 t_drop：

```
create table t_drop (a int, b int, c varchar2(20))
```

在 PL/SQL Developer 中删除表 t_drop 的步骤如下：

- ① 在 PL/SQL Developer 左侧对象清单窗口中，选择表 t_drop。
- ② 右键单击该对象，在弹出的快捷菜单中选择【Drop】命令。
- ③ PL/SQL Developer 将弹出确认窗口，如图 5-23 所示。
- ④ 单击【Yes】按钮确认删除。

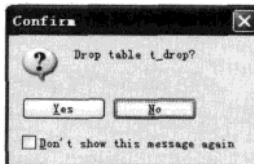


图 5-23 确认删除窗口

5.4.2 利用 SQL 语句删除数据表

同样，可以使用 SQL 语句删除某张表。删除数据表的命令为 drop table。本节将演示如何使用 drop table 命令删除数据表。首先，重新创建表 t_drop：

```
create table t_drop (a int, b int, c varchar2(20))
```

在【SQL Window】或【Command Window】中，执行范例 5-22 所示的 SQL 语句。

【范例 5-22】利用 SQL 语句删除数据表。

```
drop table t_drop
```

【代码说明】drop table 命令用于删除数据表；t_drop 指定了要删除的数据表的名称。drop table 和 delete 命令的区别：drop table 用于删除数据表，一旦删除，该数据表在数据库中将不再存在。而 delete 则用于删除数据表的记录，无论记录有无，数据表仍然是存在的。

有时，某些数据表的列被其他表引用，如外键引用，直接使用 drop table 将无法删除该表。此时，需要使用 cascade constraints 选项。

【范例 5-23】使用 cascade constraints 选项删除表约束的 SQL 语句如下：

```
drop table t_drop cascade constraints
```

【代码说明】cascade constraints 表示删除表时，将一起作用于约束。



5.5 备份/恢复数据表

数据表的备份和恢复是最常用的数据库操作，数据表的备份主要用于以下场合。

- 修改数据表结构之前；
- 修改数据表的数据之前；
- 删除某个数据表之前。

本节将从以下两个角度讲述如何备份 Oracle 数据表。

- 利用工具备份/恢复数据表；
- 利用命令备份/恢复数据表。

5.5.1 利用工具备份/恢复数据表

PL/SQL Developer 是备份/恢复 Oracle 数据表的常用工具, 本节将以 PL/SQL Developer 为例讲述如何备份/恢复数据表。首先建立一个拥有图 5-24 和图 5-25 所示的结构和内容的数据表。

```
SQL> DESC T_USERS;
```

Name	Type	Nullable	Default	Comments
USER_ID	NUMBER			
EMAIL	VARCHAR2(30)	Y		
REMARKS	VARCHAR2(50)	Y		
USER_NAME	VARCHAR2(20)			
AGE	INTEGER	Y		
BIRTHDAY	DATE	Y		

图 5-24 用于演示备份/恢复的 T_USERS 表的结构

USER_ID	USER_NAME	AGE	BIRTHDAY	EMAIL	REMARKS
1	Alex	20	1989-1-1 上午12:00:00	Alex200905@163.com	a goods boy
2	Bob	20	1989-2-1 上午12:00:00	Bob@yahoo.com	a good boy
3	Calina	23	1986-9-18 上午12:00:00	Calina@yahoo.com	a good girl
4	Denny	23	1986-10-7 上午12:00:00	Denny@yahoo.com	a good boy
5	Mike	17	1992-11-3 上午12:00:00	Mike@yahoo.com	a good boy

图 5-25 用于演示备份/恢复的 T_USERS 表的内容

PL/SQL Developer 提供了三种导出/导入方式, 分别是: Oracle 导出/导入方式、SQL 语句方式、PL/SQL 工具本身方式。

1. Oracle 导出/导入方式

Oracle 导出/导入方式实际上是调用 Oracle 本身的导出/导入命令来实现备份/恢复操作。备份/恢复数据表 T_USERS 的步骤如下。

① 选择【Tools】|【Export Tables】命令, 将出现数据表导出窗口, 如图 5-26 所示。

选择列表中的一个, 或者按住 Ctrl/Shift 键, 选择多个数据表。被选中的数据表将作为被导出的对象。

② 在【Oracle Export】页面中, 为【Where clause】添加筛选条件, 即只备份部分数据。在【Output file】中输入或者通过【浏览】获得导出文件的完整路径, 如图 5-27 所示。

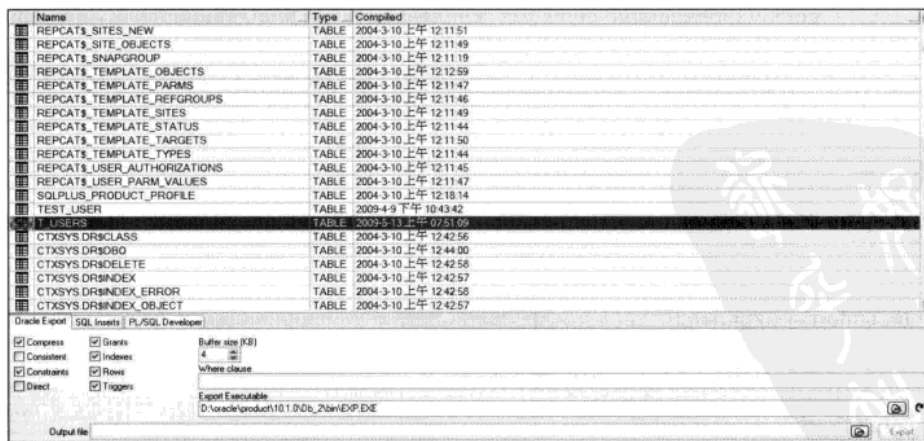


图 5-26 数据表导出窗口

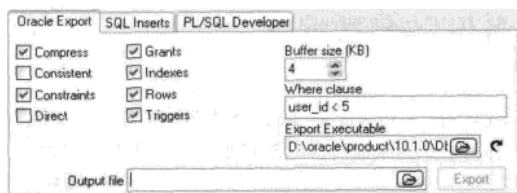


图 5-27 为导出操作填写完整信息

【Export Executable】指定导出操作实际调用的可执行文件。默认为 Oracle 安装路径下的 EXPEXE 文件，一般无须改动。

③ 单击【浏览】按钮，将弹出【另存为】对话框，可以为导出文件选择路径，并指定文件名。单击【保存】按钮，如图 5-28 所示。



图 5-28 为导出文件指定路径与文件名

④ 单击【Export】按钮，将符合条件的记录导出到 D:\temp\oracle\t_users.dmp。

⑤ 利用 DELETE FROM T_USERS 删除表中记录，或者利用 DROP TABLE T_USERS 删除整个数据表。再次对表 T_USERS 进行查询操作，确认表中记录或表本身已被删除。

⑥ 打开【Tools】->【Import Tables】窗口，并选择【Oracle Import】选项卡，如图 5-29 所示。

⑦ 单击【浏览】按钮，将弹出【打开】对话框，可以为导出文件选择路径和文件名。单击【保存】按钮，如图 5-30 所示。

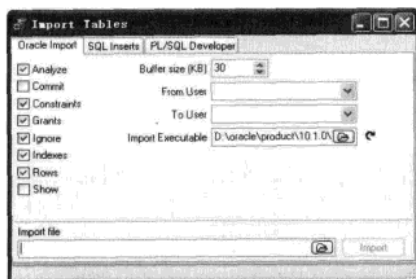


图 5-29 【Oracle 导入】界面



图 5-30 选择导入文件

⑧ 对表 T_USERS 进行查询操作，会发现数据恢复为备份过的数据，如图 5-31 所示。

USER_ID	EMAIL	REMARKS	USER_NAME	AGE	BIRTHDAY
1	Alex200905@163.com	a goods boy	Alex	20	1989-1-1 上午12:00:00
2	Bob@yahoo.com	a good boy	Bob	20	1989-2-1 上午12:00:00
3	Calina@yahoo.com	a good girl	Calina	23	1986-9-18 上午12:00:00
4	Denny@yahoo.com	a good boy	Denny	23	1986-10-7 上午12:00:00

图 5-31 执行数据表恢复后的记录

小技巧：在【Export Tables】窗口中，单击列标题【Name】，将会弹出关于表名的筛选框，如图 5-32 所示。在文本框中输入表名，例如，T_USERS，则会将搜索结果显示在数据表列表中，如图 5-33 所示。

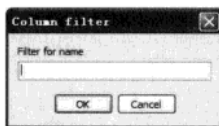


图 5-32 关于表名的筛选框

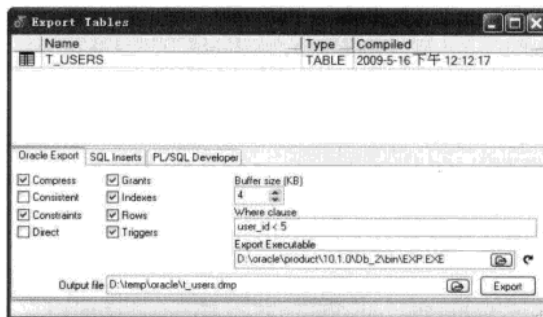


图 5-33 执行表筛选操作后的数据表列表

2. SQL 导出/导入方式

SQL 导出/导入方式，实际上是利用 SQL 语句实现整个数据表的重建。重建的过程，包括了删除表、新建表、删除记录、其他相关对象（如存储过程）的重建等步骤中的任意组合。最后利用 INSERT 语句插入所有数据。

利用 SQL 导出/导入方式实现 T_USERS 表的备份/恢复的步骤如下。

- ① 选择【Tools】|【Export Table】命令，在弹出的窗口中选择要备份的表 T_USERS。
- ② 在窗口底部选择【SQL Inserts】选项卡，将出现 SQL 导出方式对话框，如图 5-34 所示。

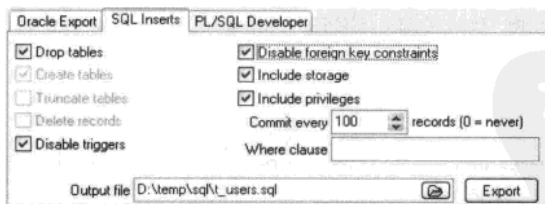


图 5-34 SQL Inserts 窗口

在图 5-34 所示的窗口中，Drop tables 选项代表生成 DROP TABLE 命令，选择了该选项将无法选择 Truncate tables 和 Delete records，因为删除了整个数据表，自然没有必要再执行删除表中记录的命令；Create tables 用于生成 CREATE TABLE 命令；Truncate table 和 Delete records 一般只选择其中一个，用于删除表中数据；Disable triggers，用于禁用与当前表相关的触发器；



Disable foreign key constraints 用于禁用外键约束; Include storage 用于包含与表相关的存储过程; Include privileges 用于将表的权限信息一起输出。

③ 直接输入或者单击浏览按钮获得【Out file】的完整本地文件路径。

④ 单击【Export】按钮, 将整个表输出为 SQL 语句模式。输出的 SQL 语句如下所示。

```
prompt PL/SQL Developer import file
prompt Created on 2009年5月15日 by Administrator
set feedback off
set define off
prompt Dropping T_USERS...
drop table T_USERS cascade constraints;
prompt Creating T_USERS...
create table T_USERS
(
    USER_ID    NUMBER not null,
    EMAIL      VARCHAR2(30),
    REMARKS    VARCHAR2(50),
    USER_NAME  VARCHAR2(20) default ' ' not null,
    AGE        INTEGER,
    BIRTHDAY   DATE
)
tablespace SYSTEM
pctfree 10
pctused 40
initrans 1
maxtrans 255
storage
(
    initial 64K
    minextents 1
    maxextents unlimited
);

prompt Disabling triggers for T_USERS...
alter table T_USERS disable all triggers;
prompt Loading T_USERS...
insert into T_USERS (USER_ID, EMAIL, REMARKS, USER_NAME, AGE, BIRTHDAY)
values (1, 'Alex200905@163.com', 'a goods boy', 'Alex', 20, to_date('01-01-1989',
'dd-mm-yyyy'));
insert into T_USERS (USER_ID, EMAIL, REMARKS, USER_NAME, AGE, BIRTHDAY)
values (2, 'Bob@yahoo.com', 'a good boy', 'Bob', 20, to_date('01-02-1989',
'dd-mm-yyyy'));
insert into T_USERS (USER_ID, EMAIL, REMARKS, USER_NAME, AGE, BIRTHDAY)
values (3, 'Calina@yahoo.com', 'a good girl', 'Calina', 23, to_date('18-09-1986',
'dd-mm-yyyy'));
insert into T_USERS (USER_ID, EMAIL, REMARKS, USER_NAME, AGE, BIRTHDAY)
values (4, 'Denny@yahoo.com', 'a good boy', 'Denny', 23, to_date('07-10-1986',
'dd-mm-yyyy'));
commit;
prompt 4 records loaded
prompt Enabling triggers for T_USERS...
alter table T_USERS enable all triggers;
set feedback on
set define on
prompt Done.
```



注意: 这些输出命令可以运行在 SQL*Plus 或者 PL/SQL Developer 中的【Command Window】中。

⑤ 删除数据表 T_USERS。

⑥ 选择【Tools】|【Import Tables】命令，在弹出的导入窗口中，选择【SQL Inserts】选项卡，如图 5-35 所示。

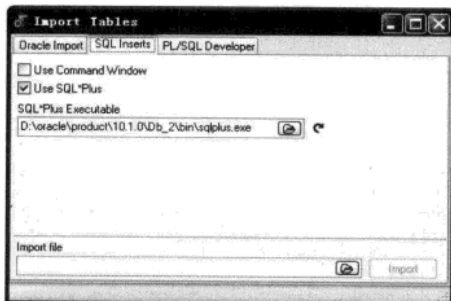


图 5-35 SQL 方式导入窗口

在本窗口中有【Use Command Window】和 Use SQL*Plus 两个选项。二者实际为二选一选项，选择其中任意一个即可。

⑦ 为【Import file】选择合适的 SQL 文件，在此选择导出的文件 D:\temp\sql\t_users.sql，并单击【Import】按钮。

⑧ 再次查询数据表 T_USER，查询结果如图 5-36 所示。

USER_ID	EMAIL	REMARKS	USER_NAME	AGE	BIRTHDAY
1	Alex200905@163.com	a goods boy	Alex	20	1989-1-1 上午12:00:00
2	Bob@yahoo.com	a good boy	Bob	20	1989-2-1 上午12:00:00
3	Calina@yahoo.com	a good girl	Calina	23	1986-9-18 上午12:00:00
4	Denny@yahoo.com	a good boy	Denny	23	1986-10-7 上午12:00:00

图 5-36 利用 SQL 方式恢复后的数据表

3. PL/SQL 导出/导入方式

PL/SQL 方式是 PL/SQL Developer 工具自带的备份/恢复方式。导出的文件为 pde 文件。利用 PL/SQL 方式实现表 T_USERS 的备份/恢复的步骤如下。

① 选择【Tools】|【Export Tables】命令。在弹出的窗口中选择欲备份的表 T_USERS。在底部窗口中选择【PL/SQL Developer】选项卡，将出现 PL/SQL 导出方式对话框，如图 5-37 所示。

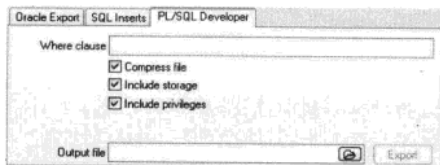


图 5-37 PL/SQL 导出方式窗口

② 在 PL/SQL 导出方式对话框中，【Where clause】用于输入筛选条件，一旦输入了该条件，将只备份数据表中符合条件的记录。【Compress file】用于标识是否将导出的文件进行压缩；【Include storage】/【Include privileges】用于标识导出时，是否包括相关的存储过程/权限信息。【Output file】用于标识导出文件的完整路径。

③ 直接输入或者单击浏览按钮获得【Out file】的完整本地文件路径。



- ④ 在导出窗口中单击【Export】按钮，将整个数据表以 pde 方式导出。
- ⑤ 删除整个数据表。
- ⑥ 选择【Tools】|【Import Tables】命令，在弹出的导入窗口中，选择【PL/SQL Inserts】选项卡，如图 5-38 所示。

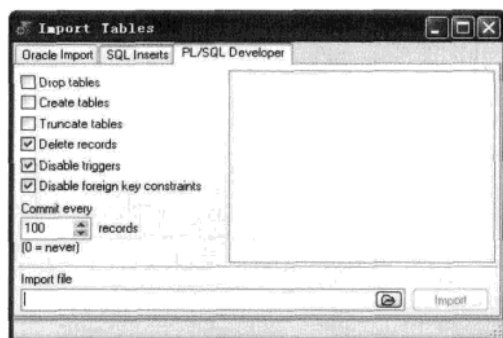


图 5-38 PL/SQL 方式的导入窗口

- ⑦ PL/SQL 导入窗口的选项与 Oracle 方式相同。为【Import file】选择已备份的文件，并单击【导入】按钮，将实现 PL/SQL 方式的导入。

5.5.2 利用命令备份/恢复数据表

对于 Oracle 数据表的备份/恢复操作，最常用的命令为 exp 和 imp。二者可以在 Windows 的命令提示符或者 UNIX/Linux 的命令行执行。exp 命令实现导出操作，imp 实现导入操作。以在 Windows 下执行为例，使用 exp 导出表 T_USERS 的步骤如下。

- ① 单击【开始】菜单，在弹出菜单中选择【运行】。

- ② 在命令提示符下，键入如下命令行：

```
exp system/abc123@test tables=(T_USERS) file=D:\temp\command\T_USERS.dmp
```

【代码说明】exp 是 Oracle 的一个可执行文件，可以在 Oracle 的安装目录下的 bin 子目录找到该文件。Oracle 安装时，已经将该路径添加到系统变量 Path 中，所以可以直接使用 exp 命令。

system/abc123 是登录数据库的用户名和密码。需要注意的是，该用户一定要具有相应的权限，才能执行导出命令。

@test 指定了要登录到的数据库。

tables 指定了要将哪些表导出，表名使用小括号括起来，多个表名之间使用逗号分隔。

file 指定导出数据存放的物理文件的完整路径和文件名。

- ③ 按下 Enter 键，执行该命令。将出现文件导出成功信息。

进入 SQL Plus 或者 PL/SQL Developer 将数据表删除，然后使用 imp 命令将备份的数据恢复。

使用 imp 导入数据表的步骤如下所示：

- ① 打开命令提示符，并键入如下命令：

```
imp system/abc123@test file=D:\temp\command\T_USERS.dmp tables=T_USERS
```

【代码说明】imp 同样是 Oracle 的一个可执行文件，可以在 Oracle 的安装目录的 bin 子目录下找到该文件。

- system/abc123 是登录数据库的用户名/密码。
 - @test 指定了要登录的数据库。
 - file 指定了进行导入操作的源文件。
 - tables 指定了进行导入操作的目标数据表名。
- ② 按下 Enter 键，执行该命令，将出现如图 5-39 所示的导入成功信息。

```

Connected to: Oracle Database 10g Enterprise Edition Release 10.1.0.2.0 - Production
With the Partitioning, OLAP and Data Mining options

Export file created by EXPORT:V10.01.00 via conventional path
Import done in ZHS16GBK character set and AL16UTF16 NCHAR character set
- Importing SYSIBM's objects into SYSIBM
- Importing table "T_USERS" 4 rows imported
Import terminated successfully without warnings.

```

图 5-39 成功导入数据表的提示信息

注意，tables 参数是必需的，而且要与导出时的表名保持一致。这是因为导出的 dmp 文件中保存了表名信息。如果导入时的表名与导出时不一致，将会导致导入失败。例如，将导入时的表名修改为 T_TEST。

```
imp system/abc123@test file=D:\temp\command\T_USERS.dmp tables=T_TEST
```

将会出现图 5-40 所示的错误。

```

Connected to: Oracle Database 10g Enterprise Edition Release 10.1.0.2.0 - Production
With the Partitioning, OLAP and Data Mining options

Export file created by EXPORT:V10.01.00 via conventional path
Import done in ZHS16GBK character set and AL16UTF16 NCHAR character set
- Importing SYSIBM's objects into SYSIBM
IMP-00013: Warning: table "T_TEST" not found in export file
Import terminated successfully with warnings.

```

图 5-40 表名错误导致导入数据表失败



5.6 临时表

Oracle 使用 create table 命令创建的数据表称为永久表或普通表。在 Oracle 中还有另外一种特殊的数据表——临时表。本节将按照以下顺序介绍临时表。

- 临时表简介；
- 会话级临时表；
- 事务级临时表；
- 查看临时表属性信息；
- 临时表的应用场景。

5.6.1 临时表简介

首先需要明确的是，临时表的临时并非指其存在性而言。也就是说，除非使用 DROP TABLE 命令来删除临时表，否则，一旦创建，将一直存在。在存在性上，和普通表没有任何区别。其临时性，指的是所存储数据的临时性。也就是说，临时表虽然一直存在，但其中的数据会在某种条件下被 Oracle 数据库自动清空。

临时表数据清空的条件有两种，一是事务提交或回滚；二是会话结束。

在 Oracle 中，使用某条命令对数据库进行操作时，并不会立即将修改更新到数据库。而是



以事务为单位将数据修改提交到数据库。每个事务以 `commit` 命令结束。也就是说，每个事务是从上一次执行 `commit`，到本次执行 `commit` 之间的操作。这些操作可能是一条或多条 SQL 语句。当使用了 `commit` 命令时，意味着一个事务的提交。事务回滚使用 `rollback` 命令实现，事务回滚将使整个事务的操作全部作废。简而言之，被 `commit` 或 `rollback` 包围的所有数据库操作就是一个事务。

Oracle 中的会话是指每个与用户进行交互的进程。在 PL/SQL Developer 中体现为，每次打开一个新的【SQL Window】或【Command Window】时，将建立一个新的会话。

对应着临时表清空数据的时机，临时表可以分为两类：事务级临时表和会话级临时表。事务级临时表是指当事务提交或事务回滚时，表中数据将被清空；会话级临时表是指当会话结束时，表中数据将被清空，而且同一时刻，两个不同会话无法访问彼此的数据。

5.6.2 会话级临时表

创建临时表应该使用命令 `create global temporary table` 命令。

【范例 5-24】用于创建名为 `tmp_users_session` 的会话级临时表。

```
create global temporary table tmp_users_session (user_id int, user_name varchar2(20),
user_email varchar2(30)) on commit preserve rows
```

【代码说明】`create global temporary table` 用于创建临时表。其中，关键字 `global` 不可省略；`on commit preserve rows` 表示当事务提交时，保留数据。即在整个会话中，数据库不会自动清理表中数据。

在成功创建会话级临时表 `tmp_users_session` 之后，可以通过如下步骤测试临时表在整个会话内保持数据的特点。

(1) 执行 SQL 语句，向临时表 `tmp_users_session` 中插入两条记录。

```
insert into tmp_users_session (user_id, user_name, user_email) values (1, 'Alex',
'Alex@yahoo.com')
insert into tmp_users_session (user_id, user_name, user_email) values (2, 'Bob',
'Bob@yahoo.com')
```

(2) 查询表中数据，以验证是否插入成功。

```
select * from tmp_users_session
```

查询结果如图 5-41 所示。

(3) 提交数据修改，以结束事务。

```
commit
```

(4) 利用查询语句，查看数据表中的记录。

```
select * from tmp_users_session
```

此时的数据仍然存在。

(5) 重新打开一个【SQL Window】，在该窗口中查询 `tmp_users_session` 表中的数据。查询结果如图 5-42 所示。

USER_ID	USER_NAME	USER_EMAIL
1	Alex	Alex@yahoo.com
2	Bob	Bob@yahoo.com

图 5-41 执行 INSERT 语句后的查询结果

USER_ID	USER_NAME	USER_EMAIL

图 5-42 在新的会话中查询临时表 `T_USERS_SESSION` 中的数据

(6) 关闭 PL/SQL Developer, 并重新登录数据库, 相当于重新连接数据库。此时在【SQL Window】中查询临时表 tmp_users_session 中的数据, 结果如图 5-43 所示。

USER_ID	USER_NAME	USER_EMAIL

图 5-43 重新连接数据库并查询临时表

综上所述, 会话级临时表中的数据仅在当前会话存在, 一旦会话结束, 数据库将自动清理其中数据 (相当于使用了 truncate table) 命令。各会话之间的数据也是相互独立、互不影响的。

5.6.3 事务级临时表

与会话级临时表一样, 事务级临时表的创建同样使用 create global temporary table 命令。只是将 on commit preserve rows 变更为 on commit delete rows, 即提交时删除表中记录。

【范例 5-25】 创建事务级临时表, 并插入两条测试数据。

```
create global temporary table tmp_users_transaction (user_id int, user_name
varchar2(20), user_email varchar2(30)) on commit delete rows
insert into tmp_users_transaction (user_id, user_name, user_email) values (1, 'Alex',
'Alex@yahoo.com')
insert into tmp_users_transaction (user_id, user_name, user_email) values (2, 'Bob',
'Bob@yahoo.com')
```

在成功创建会话级临时表 tmp_users_transaction 之后, 可以通过如下步骤测试临时表在整个会话内保持数据的特点。

(1) 查询表中数据, 以验证是否插入成功。查询结果如图 5-44 所示。

USER_ID	USER_NAME	USER_EMAIL
1	Alex	Alex@yahoo.com
2	Bob	Bob@yahoo.com

图 5-44 向事务级临时表中插入数据后的查询结果

(2) 提交数据修改, 以结束事务。

```
commit;
```

(3) 利用查询语句, 查看数据表中的记录。查询结果如图 5-45 所示。

USER_ID	USER_NAME	USER_EMAIL

图 5-45 事务提交后, 临时表中的数据被清空

(4) 再次执行插入语句, 将两条测试记录插入到 tmp_users_transaction 中, 并执行 rollback 操作:

```
rollback
```

(5) 再次查询临时表 tmp_users_transaction, 结果如图 5-46 所示。

USER_ID	USER_NAME	USER_EMAIL

图 5-46 事务回滚后, 临时表中的数据被清空

综上所述, 事务级临时表的数据仅在当前事务有效, 事务一旦结束, 所有数据都会自动清空。



5.6.4 查看临时表在数据库中的信息

一个数据表在数据库中的最主要信息有以下几个方面：所属表空间、包含的列及列类型。在 PL/SQL Developer 左侧窗口的【Tables】分支下，可以找到名为 TMP_USERS_SESSION 和 TMP_USERS_TRANSACTION 的表。像普通表一样，可以通过二者右键菜单中的【Properties】查看其属性信息。通过二者的【Columns】子分支，可以查看列信息。

需要注意的是它们的表空间，执行如下 SQL 语句，查询其表空间信息，并与之前的普通表 t_users 进行比较。

```
select table_name, tablespace_name from user_tables where table_name= 'T_USERS'
or table_name='TMP_USERS_SESSION' or table_name='TMP_USERS_TRANSACTION'
```

查询结果如图 5-47 所示。

TABLE_NAME	TABLESPACE_NAME
TMP_USERS_SESSION	(null)
TMP_USERS_TRANSACTION	(null)
T_USERS	SYSTEM

图 5-47 比较临时表与普通表的表空间

从查询结果可知，临时表的表空间为空。

对于 TMP_USERS_SESSION 和 TMP_USERS_TRANSACTION，二者分属于会话级和事务级临时表。可以通过如下 sql 语句查询二者的不同。

```
select table_name, temporary, duration from user_tables where table_name= 'T_USERS'
or table_name='TMP_USERS_SESSION' or table_name='TMP_USERS_TRANSACTION'
```

查询结果如图 5-48 所示。

TABLE_NAME	TEMPORARY	DURATION
TMP_USERS_SESSION	Y	SYSS\$SESSION
TMP_USERS_TRANSACTION	Y	SYST\$TRANSACTION
T_USERS	N	(null)

图 5-48 比较普通表、会话级临时表和事务级临时表

通过图 5-48 的查询结果可知：temporary 列标识了对应表是否为临时表，TMP_USERS_SESSION 和 TMP_USERS_TRANSACTION 表都为 Y，而 T_USERS 表则为 N；duration 列则可以用来区分临时表的具体类型，SYSS\$SESSION 标识 tmp_users_session 表为会话级临时表，而 SYST\$TRANSACTION 标识 tmp_users_transaction 为事务级临时表。

5.6.5 临时表的应用场景

临时表的特殊性使其在特殊场景下有着特殊的作用。临时表的主要应用场景有以下几种。

1. 大表分割

众所周知，为表创建合适的索引可以在很大程度上提高数据查询的速度。但是，当某个表的数据量相当大，例如数据量为亿级时，那么创建索引将会花费大量时间，而且，查询大的索引表，与直接查询数据表相比，在性能上几乎没有任何优势。此时，一个常用的方法就是分割大表，例如，将大表分割为多个小的临时表，然后对这些小表进行相应操作，最后对所有查询结果进行综合处理。

2. 解决并行问题

当多个进程（客户端）同时对某张表进行操作时，往往会出现并行问题。即不允许多个进

程同时修改同一条记录或同一张表，最常用的方法就是对表进行锁定。但是，当客户端连接众多，而且更新数据表频繁时，对表的锁定将影响整个数据库的性能。此时，可以利用临时表，因为各会话在临时表中的数据都是相互透明的，所以，无须对临时表进行锁定。毫无疑问，使用临时表将极大提高数据库性能。

3. 作为数据缓存

在程序段（例如，存储过程），可能需要对若干数据进行复杂的运算。此时，可以创建一个临时表，并将这些数据存储在临时表中。因为可以像操作普通表一样操作临时表，这样，许多函数和 SQL 语句都可以用来处理这些数据。此时的临时表即作为临时数据的存储空间。



5.7 特殊的表 dual

在所有的 Oracle 数据表中，有一个非常特殊的表——dual。该表是每个数据库创建时默认生成的。该表仅有一列和一行数据，在数据库开发中有着非常特殊的作用。本节将从以下三个角度介绍表 dual。

- 分析 dual 表；
- dual 表的应用场景；
- 为什么不可以随意修改 dual 表。

5.7.1 分析 dual 表

在 PL/SQL Developer 或 SQL*Plus 中，可以利用 DESC 命令查看 dual 表的结构，如图 5-49 所示。

```
SQL> DESC DUAL;
Name      Type          Nullable Default Comments
-----
DUMMY     VARCHAR2(1)   Y
```

图 5-49 利用 DESC 命令查看 dual 表的结构

分析图 5-49 可知，dual 仅有一列——dummy。该列的数据类型为一个字符长度的可变字符串。dummy 列允许为空。

在 PL/SQL Developer 或 SQL*Plus 中查看 dual 表的数据：

```
select * from dual
```

查询结果如下：

```
DUMMY
```

```
1
```

从查询结果可知，dual 表的数据仅有一行。

5.7.2 dual 表的应用场景

在 Oracle 数据库中，dual 表实际上是作为一个虚表的概念存在的。也就是说，dual 表存在的意义并非为了存储数据。更多时候，dual 表用来作为 from 的源表。

因为 Oracle 的查询语句必须满足 select * | [column1 [AS alias1], column2 [AS alias2]] from table 的语法格式，其中的 from 所指向的表是必需的，所以即使某些数据不属于任何表，也必



须有一个强制的表名。dual 表即可用做这个强制的虚表。

利用 dual 表获得日期是最常见的应用场景，范例 5-26 演示了这种情况。

【范例 5-26】使用 dual 表获得当前日期。

```
select sysdate from dual
```

【代码说明】sysdate 是系统函数，返回当前日期。当前日期的值总是在不断变化的，而这些值也不存在于任何表中。所以在此使用了 dual 表。

dual 表可以用来进行数学运算，如范例 5-27 所示。

【范例 5-27】使用 dual 表进行数学运算。

```
select 3*4+17 as result from dual
```

运算结果如下：

RESULT
29

还可以利用 dual 表获得序列的值，如范例 5-28 所示。

【范例 5-28】使用 dual 表实现序列的运算。

```
select t_user_sql.nextval from dual
```

【代码说明】序列可以认为是一个自动增长的序号，每执行一次查询，序号将自动加 1。

5.7.3 修改 dual 表对查询结果的影响

Oracle 针对 dual 表会做一些内部操作，以保证其一行一列的数据结构不发生改变。本节将尝试修改 dual 表中的数据，来查看对应用程序的影响。

(1) 首先使用 SYS 登录数据库 TEST，为 dual 表插入一条数据'Y'，SQL 语句如下所示：

```
insert into dual (dummy) values ('Y')
```

(2) 查询 DUAL 表，查询结果如下：

DUMMY
Y

(3) 再次运行查询当前日期的 SQL 语句：

```
select sysdate from dual
```

查询结果如下：

SYSDATE
2009-5-18 下午10:48:05

从查询结果可以看出，尽管 dual 表已经包含了两条记录，但是针对日期查询语句，仍然只返回一条记录。

(4) 利用如下 SQL 语句删除表 DUAL 中的数据：

```
delete from dual
```

再次查询 DUAL 表中的数据，查询结果如下：

DUMMY

对于日期查询语句，返回的查询结果如下：

SYSDATE

2009-5-18 下午10:53:08

(5) 可见，即使 dual 表不存在数据，仍然不影响使用它作为虚表的查询结果。但是，不可以删除该表，一旦删除该表，所有使用到该表的查询将无法完成。

在 Oracle 10g 之前的版本中，为 dual 表添加或者删除记录，都会影响使用该表作为虚表的查询结果。Oracle 10g 对于 dual 表做了更多的内部工作，无论其真实的数据内容是什么，都保证效果与单条记录一样。这是 Oracle 10g 针对该表的处理机制所做的较大改进。



5.8 本章实例

在本章中，创建表、修改表和删除表等表操作是重点讲述的内容。本节将通过一个范例来综合执行这几项操作。

【范例 5-29】 演示表操作的综合应用。

(1) 首先利用 PL/SQL Developer 登录数据库 TEST。

(2) 创建一个新表，SQL 语句如下：

```
SQL> create table test_table (id number, name varchar2(20));
```

Table created

(3) 查看此时的表结构，代码如下：

```
SQL> describe test_table;
```

Name	Type	Nullable	Default	Comments
ID	NUMBER	Y		
NAME	VARCHAR2(20)	Y		

(4) 为表添加一列 status，代码如下：

```
SQL> alter table test_table add (status varchar2(3));
```

Table altered

(5) 查看此时的表结构，代码如下：

```
SQL> describe test_table;
```

Name	Type	Nullable	Default	Comments
ID	NUMBER	Y		
NAME	VARCHAR2(20)	Y		
STATUS	VARCHAR2(3)	Y		

可见，此时的表 test_table 已经成功添加了列 status。

(6) 删除表中的列 status，代码如下：

```
SQL> alter table test_table drop column status;
```

Table altered

(7) 删除表 test_table，代码如下：



```
SQL> drop table test_table;
```

```
Table dropped
```



5.9 本章小结

本章着重讲述了 Oracle 中数据库中的表空间和表的概念。表空间是 Oracle 数据库的独有概念，这种设计理念，为数据库的灵活配置和空间分配提供了方便。对于数据表，需要着重理解临时表和虚表 `dual` 的作用。临时表在日常开发中并不常用，而 `dual` 表则是不可或缺的。尤其需要注意的是，不要对 `dual` 表进行任何修改操作，否则，将对应用程序带来不可预知的风险。



5.10 习题

1. 请简述创建表空间命令中主要参数的意义。
2. 请简述 Oracle 的数据表在添加列和删除列时，语法形式上的区别。
3. 请简述会话级临时表和事务级临时表的区别。
4. 请简述虚表 `dual` 的主要应用场景。



第6章 约 束

约束是每个数据库必不可少的一部分。约束的根本目的在于保持数据的完整性。数据完整性是指数据的精确性和可靠性。即数据库中的数据都符合某种预定义规则。当用户输入的数据不符合这些规则时，将无法实现对数据库的更改。本章将介绍以下几种约束，并讲解如何利用这些约束保持数据完整性。

- 主键约束；
- 外键约束；
- 唯一性约束；
- 检查约束；
- 默认值约束。

通过本章学习，读者可以掌握 Oracle 中各约束的基本概念，并对如何使用这些约束保证数据完整性有清晰的了解。



6.1 主键约束

主键约束是数据库中最常见的约束。主键约束可以保证数据完整性。即防止数据表中的两条记录完全相同，通过将主键纳入查询条件，可以达到查询结果最多返回一条记录的目的。

6.1.1 主键简介

主键被创建在一个或多个列上，通过这些列的值或者值的组合，唯一地标识一条记录。例如，对于存储了学生信息的 `student` 表，一般会为每个学生分配一个 `student_id`，也就是说将主键建立在 `student_id` 这个列上。`student_id` 将成为每个学生的唯一标识。当向 `student` 表中插入新的学生信息时，如果要插入的 `student_id` 已经存在，数据库将拒绝插入该条记录。这就是主键保证数据完整性的体现。对于主键，有以下几点需要注意。

- 主键列的数据类型并不一定是数值型。很多时候，会见到将自动增长的整数作为主键列的数据类型。自动增长的整数不会出现重复，当然是作为主键的理想选择，但是并非唯一选择。字符串等数据类型照样可以作为主键列的数据类型。
- 主键列不一定只有一列。将 `id` 作为主键是最常见的情况，但是主键列可以有多列。这些列的实际数据的组合将是唯一的。例如，学生 `id` 可以作为学生的唯一性标识，而学生姓名、出生年月、家庭住址的组合照样可以作为学生表的主键，而且能更明确反映出主键规则的真实意义。
- 主键是规则制定者的意志体现，不要将其与现实世界混淆。例如，在 `student` 表中，可以将学生姓名列 `student_name` 作为主键，虽然不符合现实情况，但主键是规则制定者意志的体现——在该表中不能出现同名学生。



6.1.2 创建主键约束

主键约束可以在创建表的同时进行创建，也可以在表存在时进行添加。本节将演示这两种创建方式，并演示主键的作用。

1. 创建主键约束

主键约束作为表结构设计的一部分，一般在建表时创建。创建主键约束使用关键字 `primary key`。以创建学生表 `student` 为例，假设该表所包含的列及其数据类型有以下几种：

- `student_id` `number`
- `student_name` `varchar2(20)`
- `student_birthday` `date`
- `student_address` `varchar2(50)`
- `student_phone` `varchar2(20)`

以下几个范例体现了建表时创建主键的几种情况。

【范例 6-1】 演示只有一列的主键。

将列 `student_id` 作为主键列时，创建表 `student` 的 SQL 语句如下：

```
create table student (student_id number primary key, student_name varchar2(20),
student_birthday date, student_address varchar2(50), student_phone varchar2(20))
```

【代码说明】 `primary key` 出现在列名 `student_id` 的后面，表明要在列 `student_id` 上创建主键约束。

2. 查看主键约束

可以通过 PL/SQL Developer 中表 `student` 对象的【右键】|【VIEW】|【Keys】来查看表的主键信息。另外，所有用户创建的主键的基本信息都存储在表 `user_constraint` 表中。所有的约束都是依附于表存在的，所以每个约束肯定隶属于唯一确定的表。到目前为止，表 `student` 仅有一个主键约束，所以可以通过表名 `student` 作为查询条件来获得其约束信息。

【范例 6-2】 演示如何查看表 `student` 的约束。

```
select table_name, constraint_name, constraint_type, status from user_constraints
where table_name = 'STUDENT'
```

【代码说明】 列 `table_name` 存储了表名信息；列 `constraint_name` 存储了约束名信息；列 `constraint_type` 存储了约束类型；列 `status` 存储了约束的当前状态。

执行范例 6-2 的 SQL 代码，查询结果如图 6-1 所示。

TABLE_NAME	CONSTRAINT_NAME	CONSTRAINT_TYPE	STATUS
STUDENT	SYS_C005121	P	ENABLED

图 6-1 查询 `student` 表的约束

通过查询结果可以看出，表 `student` 仅有一个约束，名为“`SYS_C005121`”；约束类型为“`P`”代表主键 `Primary Key`；`status` 为 `ENABLED`，表明当前约束可用。

在获得了表 `student` 的主键约束 `SYS_C005121` 的基本信息之后，还可以通过查询视图 `user_cons_columns` 来获得该主键被建立在哪些列之上。

【范例 6-3】 演示如何获得主键的作用列。


```
select constraint_name, table_name, column_name from user_cons_columns where
constraint_name = 'SYS_C005121'
```

【代码说明】列 `constraint_name` 存储了约束名；列 `column_name` 则存储了该约束被建立在哪些列上。

查询结果如图 6-2 所示。

CONSTRAINT_NAME	TABLE_NAME	COLUMN_NAME
SYS_C005121	STUDENT	STUDENT_ID

图 6-2 查询主键建立在哪些列上

3. 测试主键约束

在 `student` 表的主键创建之后，可以通过插入重复的主键数据进行测试。

【范例 6-4】演示插入重复主键数据，来测试主键的作用。

```
insert into student (student_id, student_name, student_birthday, student_address,
student_phone) values (1, '张三', to_date('1998-12-01', 'YYYY-MM-DD'), '人民路 96 号',
'010-22415109')
```

【代码说明】SQL 语句 `insert into student` 用于向表 `student` 中插入新的记录。

执行范例 6-4 的 SQL 语句，可以成功插入数据。再次执行以下 SQL 语句。

```
insert into student (student_id, student_name, student_birthday, student_address,
student_phone) values (1, '李四', to_date('1999-06-03', 'YYYY-MM-DD'), '人民路 32 号',
'010-22415032')
```

在上例 SQL 语句中，使用了与范例 6-4 相同的 `student_id`——1，所以无法成功插入。以 PL/SQL Developer 为例，将给出如图 6-3 所示的错误提示。

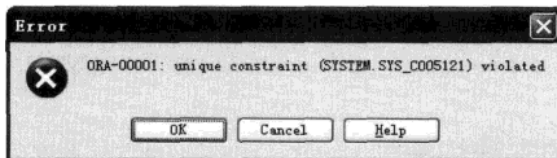


图 6-3 主键约束不允许插入相同键值

4. 显式命名主键约束

在范例 6-1 中，并没有为主键进行显式命名，但是 Oracle 数据库还是为其自动生成了一个名称——`SYS_C005121`。该名称对于用户来说，没有任何实际意义——既不利于识别，也不便于记忆。其实，在主键创建时，还可以显式命名该主键。

【范例 6-5】演示如何显式命名主键。

```
create table student (student_id number constraint pk_student primary key,
student_name varchar2(20), student_birthday date, student_address varchar2(50),
student_phone varchar2(20))
```

【代码说明】在 `student_id` 列名之后使用 `constraint` 关键字，表示在该列上创建约束；`pk_student` 指定约束名；`primary key` 指定约束的类型为主键。

此时查询关于该主键的信息，结果如图 6-4 所示。



TABLE_NAME	CONSTRAINT_NAME	CONSTRAINT_TYPE	STATUS
STUDENT	PK_STUDENT	P	ENABLED

图 6-4 显式命名主键的基本信息

5. 创建主键的另一种写法

正如在 PL/SQL Developer 左侧对象结构树中所显示的, 一个表的子对象中, Columns 分支和 Primary Key 分支是并列的。在创建表时, 同样可以将 Primary Key 与各列并列起来。

【范例 6-6】演示如何将 Primary Key 的描述与列的描述并列, 来创建主键。

```
createtable student (student_id number, student_name varchar2(20), student_birthday
date, student_address varchar2(50), student_phone varchar2(20), constraint pk_student
primary key (student_id))
```

【代码说明】constraint pk_student primary key (student_id) 用于创建主键 pk_student。需要注意的是, 主键所在的列名必须使用小括号括起来。主键的描述与其他列的描述是并列关系, 并使用逗号进行分隔。

6. 创建多列主键

在多列上创建主键的语法, 与范例 6-6 非常相似。以在 student 表的 student_name, student_birthday, student_address 三列上建立主键为例, 相应的 SQL 语句如范例 6-7 所示。

【范例 6-7】演示如何在多列上创建主键。

```
create table student (student_id number, student_name varchar2(20), student_
birthday date, student_address varchar2(50), student_phone varchar2(20),
constraint pk_student primary key (student_name, student_birthday, student_
address))
```

【代码说明】要在表 student 的多列上创建主键, 只要将 primary key 所对应的列 student_id, 替换为 student_name, student_birthday, student_address 的列名列表即可。多个列名之间使用逗号分隔。

6.1.3 修改表的主键约束

表的主键也是作为表的对象存在的, 因此, 同样可以对其进行修改。这其中包括为表添加主键, 删除主键、启用/禁用主键、重命名主键等。

1. 为表添加主键

如果一个表创建时并未指定主键, 那么在表创建之后照样可以为其添加主键。例如, 表 student 在创建后需要在列 student_id 上创建主键。

【范例 6-8】演示如何为已存在的表创建主键。

```
alter table student modify (student_id number primary key)
```

【代码说明】modify 选项实际为修改列属性, 并在修改列属性的同时, 将该列指定为主键列。

2. 为表添加多列主键

当要添加的主键作用于多个列时, 可通过添加主键命令, 而不是通过修改列的属性来添加主键。

【范例 6-9】演示如何为表添加多列主键。

```
alter table student add constraint pk_student primary key (student_name,
student_birthday, student_address)
```

【代码说明】add constraint 用于为表添加约束；pk_student 为约束名；primary key 为约束的类型——主键；括号内的列名列表表明主键将建立在这些列上。

3. 删除主键

与列一样，主键是表的一个对象，删除表的主键与删除列的语法非常相似。

【范例 6-10】演示如何删除表 student 的主键。

```
alter table student drop primary key
```

【代码说明】drop primary key 用于删除表的主键。因为一个表的主键是唯一的，所以无须指定主键名。

当然，如果将主键看做表的一个对象，而且知道主键的名称。那么可以利用删除约束的语法来删除表的主键。相应的代码如范例 6-11 所示。

【范例 6-11】演示如何利用删除约束的语法删除主键。

```
alter table student drop constraint pk_student
```

【代码说明】drop constraint 用于删除表的约束。因为一个表的约束可能有若干个，所以必须指定约束的名称，在这里使用的是主键的名称 pk_student。

4. 启用/禁用主键

在图 6-4 中，列 status 标识了约束的可用性。如果禁用一个主键约束，将可以插入重复主键数据。以表 student 为例，其主键建立在列 student_id 上。

【范例 6-12】演示如何禁用表 student 的主键约束。

```
alter table student disable primary key
```

【代码说明】disable primary key 用于禁用表的主键。因为一个表最多含有一个主键，因此无须指定主键名。

禁用该主键后，范例 6-3 所示的两条 SQL 将可以成功执行。表 student 中的数据内容如图 6-5 所示。

STUDENT_ID	STUDENT_NAME	BIRTHDAY	STUDENT_ADDRESS	STUDENT_PHONE
1	张三	1998-12-01	人民路98号	010-22415109
1	李四	1999-06-03	人民路32号	010-22415032

图 6-5 禁用主键约束后可以插入重复主键数据

此时尝试启用表的主键约束，相应的 SQL 语句如下：

```
alter table student enable primary key
```

Oracle 数据库无法完成该任务，并抛出异常。以 PL/SQL Developer 为例，将抛出如图 6-6 所示的错误。

这是因为 Oracle 在启用主键之前，要首先对主键列的数据进行校验，如果校验符合恢复主键的条件，才能重新启用主键。为了重新启用主键，应该首先修改 student_id 的值。

```
update student set student_id=2 where student_name = '李四'
```

此时，可成功启用主键。

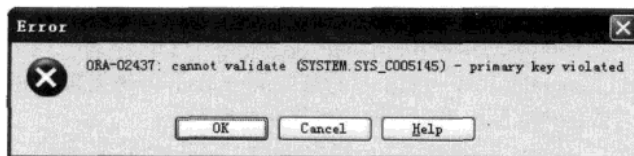


图 6-6 启用主键出现错误

5. 重命名主键

正如在前面的内容中所看到的，表的主键名称，既可以利用 Oracle 自动分配，也可以自行指定。因此，重命名主键也是常用操作。

【范例 6-13】 演示如何使用 rename 关键字重命名主键。

首先通过查询 user_constraints 表获得表 student 的主键，以自动生成的 SYS_C005145 为例，重命名该主键的 SQL 语句如下：

```
alter table student rename constraint SYS_C005145 to PK_STUDENT
```

【代码说明】 rename constraint 用于修改约束的名称；SYS_C0055145 为原约束名；PK_STUDENT 为目标约束名。

6.1.4 主键应用场景

主键是数据表中最常用的约束，其应用场景一般有如下几种。

1. 对于完整性要求比较高的数据表都应该建立主键

主键可以保证数据在主键列上的唯一性，对于特定的业务逻辑，可以指定相应的主键规则，避免出现错误数据。例如，人员信息数据表，可以考虑以人员的身份证号作为主键，避免出现重复。

2. 对于经常按照某列进行查询的数据表，应该考虑建立主键

主键在建立时，会自动在该表中建立一个索引。对于表 student，可以利用如下查询语句获得其索引名称。

```
select table_name, constraint_name, constraint_type, index_name from user_constraints where table_name = 'STUDENT'
```

查询结果如图 6-7 所示。

TABLE_NAME	CONSTRAINT_NAME	CONSTRAINT_TYPE	INDEX_NAME
STUDENT	PK_STUDENT	P	SYS_C005145

图 6-7 表 student 主键的对应索引

利用得当的索引，可以显著提高查询速度。所以对于经常以某列作为查询条件的表，应该考虑在该列上创建主键，以同时获得索引所带来的性能上的提高。

3. 考虑是否对外键有利

外键是数据表中另外一个重要的约束。如果某个表需要为另外一个表提供外键支持，那么应该建立主键，因为外键是建立在主键基础之上的。



6.2 外键约束

外键与主键一样用于保证数据完整性，主键是针对单个表的约束，而外键则描述了表之间的关系。即两个表之间的数据的相互依存性。

6.2.1 外键简介

外键实际是一种关联，描述了表之间的父子关系。即子表中的某条数据与父表中的某条数据有着依附关系。当父表中的某条数据被删除或进行更改时，会影响子表中的相应数据。例如，父表中的数据被删除，则子表中的相应数据也应该被删除；当父表中的数据更新时，子表中的数据也应该做出适当的反应。

外键约束是建立在子表之上的，并要求子表的每条记录必须在父表中有且仅有一条记录与之对应。例如，有两个数据表 `customers` 和 `orders`，其中 `customers` 表存储了客户信息，而 `orders` 表存储了订单信息。订单因客户存在而存在，当通过外键关联去寻找客户信息时，只能找到唯一的客户。这实际演示了一张订单来自一个客户的客观情况。同时，某条 `order` 记录，没有对应的 `customer` 的信息，也是不允许的，即一张订单没有客户是不允许的。

`customers` 表中记录与 `orders` 表中记录并非一一对应。`customers` 表中的一条记录在 `orders` 表中可能有相应的一条或多条记录。也就是说，一个客户可能有多张订单。

外键约束不仅仅限制子表的数据，对父表同样具有约束作用。向 `orders` 表中成功插入一条记录，意味着添加一张订单，在 `customers` 表中有唯一客户信息。但是向 `customers` 表中插入一条记录，并不意味着该客户一定要有订单存在。另外一种可能的情况是，预期修改 `customers` 中的主键，如果在 `orders` 表中已经有一条或多条订单与之关联，也将破坏数据完整性，同样无法成功修改。

6.2.2 创建外键约束

创建外键，首先在父表中创建主键，因为当使用外键寻找父表中记录时，必须要找到唯一一条，所以必须在父表中使用主键来标识记录的唯一性。

1. 建立外键

以 `customers` 表和 `orders` 表为例，首先创建这两个表。使用的 SQL 语句如下：

创建表 `customers`：

```
create table customers (customer_id number primary key, customer_name varchar2(50),
customer_address varchar2(50), customer_phone varchar2(20), email varchar2(20),
contrator varchar2(20)) create table customer (customer_id number primary key,
customer_name varchar2(50), customer_address varchar2(50), customter_phone varchar2(20),
email varchar2(20), contractor varchar2(20))
```

创建表 `orders`：

```
create table orders (order_id number primary key, customer_id number, goods_name
varchar(20), quantity number, unit varchar(10))
```

表 `customers` 和表 `orders` 创建时都建立了主键：表 `orders` 表中有一列 `customer_id`，用于存储当前订单所属客户的 `id`。

此时，要为表 `orders` 建立与表 `customers` 的外键关联，相应的 SQL 语句应该如范例 6-14 所示。



【范例 6-14】演示如何建立 orders 表到 customers 表的外键关联。

```
alter table orders add constraint fk_orders_customers foreign key (customer_id)
references customers (customer_id)
```

【代码说明】外键同样是表的约束，与主键一样，应该使用 add constraint 选项；fk_orders_customers 为新建外键的名称；foreign key 指定创建的约束是一个外键；customer_id 指定了在本表，即 orders 表中使用的列；references customers 指定了要关联到的表；最后的 customer_id 指定了在 customers 表中被关联的列。

注意，references 使用了动词的第三人称的单数形式，不能写为 reference。

2. 查看外键信息

可以通过 PL/SQL Developer 中表 orders 对象的【右键】|【VIEW】|【Keys】来查看所建约束。另外，可以在数据库中查看新建外键的信息。

【范例 6-15】演示如何查看 orders 表的外键信息。

```
select table_name, constraint_name, constraint_type, r_constraint_name from
user_constraints where table_name = 'ORDERS'
```

查询结果如图 6-8 所示。

TABLE_NAME	CONSTRAINT_NAME	CONSTRAINT_TYPE	R_CONSTRAINT_NAME
ORDERS	SYS_C005180	P	(null)
ORDERS	FK_ORDERS_CUSTOMERS	R	SYS_C005181

图 6-8 查看表 orders 的约束信息

从查询结果可以看出，表 orders 有两个约束。一个名为 SYS_C005180，constraint_type 为 P 表示该约束是一个主键约束，由于在创建主键时没有为其指定名称，SYS_C005180 由 Oracle 自动分配；另一个名为 FK_ORDERS_CUSTOMERS，constraint_type 为 R 表示其为外键约束（References 的首字母）。r_constraint_name 用于存储与该约束关联的其他约束名。可以通过如下 SQL 语句查看关联约束的信息。

```
select * from user_constraints where constraint_name = 'SYS_C005181'
```

查询结果如图 6-9 所示。

CONSTRAINT_NAME	CONSTRAINT_TYPE	TABLE_NAME
SYS_C005181	P	CUSTOMERS

图 6-9 外键关联到父表主键

可见，表 orders 的外键 FK_ORDERS_CUSTOMERS 关联到表 customers 的主键上。

3. 验证外键约束的有效性

外键约束用于约束表之间的父子关系，当 customers 和 orders 表都为空时，执行如下 SQL 语句向子表 orders 中插入一条数据。

```
insert into orders (order_id, customer_id, goods_name, quantity, unit) values (1,
1, 'FABRIC', 20, 'BMP')
```

在 PL/SQL 中执行该语句时，将弹出如图 6-10 所示的错误窗口。

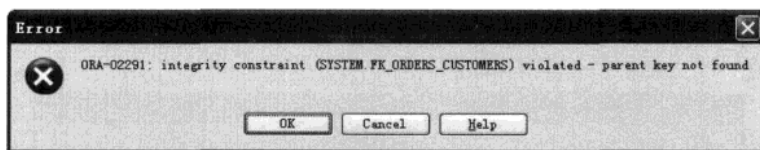


图 6-10 外键限制插入非法数据

该提示给出约束冲突，父表中没有找到相应的记录。错误原因是要插入数据的 `customer_id` 为 1，而在表 `customers` 并不存在 `customer_id` 为 1 的记录。

外键约束并不影响父表中记录的插入，因此可以利用如下 SQL 语句向 `customers` 表中插入一条记录。

```
insert into customers (customer_id, customer_name, customer_address, customer_phone,
email, contractor)
values (1, '新晋制衣', '解放北路 23 号', '010-23458761', 'xinjin@xinjin.com', '李明浩
');
```

该记录被插入后，再次执行向 `orders` 表中插入数据的 SQL 语句，将可以成功插入。

4. 修改子表数据，验证外键有效性

在向 `orders` 表中成功插入数据后，可以修改 `orders` 表中数据以验证外键的有效性。

首先，修改订单信息中的货物数量信息，即更新列 `quantity` 的值。`quantity` 列与外键毫无关系。

```
update orders set quantity = 52 where order_id=1
```

此时查询 `orders` 表，将能够看到更新后的信息，如图 6-11 所示。

ORDER_ID	CUSTOMER_ID	GOODS_NAME	QUANTITY	UNIT
1	1	FABRIC	52	BMP

图 6-11 更新与外键约束无关的列

假设此时需要修改该订单的 `customer` 信息，可以使用如下 SQL 语句。

```
update orders set customer_id = 3 where order_id=1
```

在 PL/SQL Developer 中仍然会给出约束冲突的提示，如图 6-12 所示。

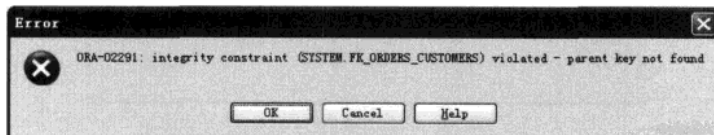


图 6-12 更新外键值不当的错误提示

可见，当修改非外键列的数据时，不会受外键影响；但是当修改外键列为不合理数据时，外键约束将禁止该动作的执行。

5. 修改父表数据，验证外键有效性

外键约束同样对父表数据修改进行验证。例如，需要修改表 `customer` 中的 `customer_id` 数据，此时将会对子表 `order` 产生影响。

【范例 6-16】 演示修改父表 `customers` 对 `orders` 表的影响。

```
update customers set customer_id = 2 where customer_id = 1
```



【代码说明】利用该 SQL 语句，尝试将 customers 表中列 customer_id 的值由 1 修改为 2。在 PL/SQL Developer 中，将给出如图 6-13 所示的错误提示。

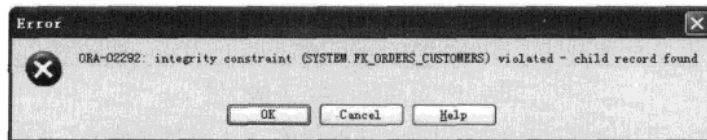


图 6-13 修改父表中的主键值不当的错误提示

图 6-13 所示的错误是因为修改父表的主键列。而主键列又是子表 orders 的外键关联列。目标值与原值不同，造成子表中的某些订单数据丢失客户信息，因此会给出错误提示。

外键校验的过程为，首先通过欲修改的父表的原主键值在子表中进行查找，如果未找到子记录，那么可以直接进行修改。如果找到子记录，那么将验证修改后的值与原值是否相同，如果相同，那么可以成功修改，如果不相同，那么将禁止修改。

同样，如果尝试删除有子记录的父表记录，或者删除整个表，也会出现外键约束给出的错误信息。

可以利用如下 SQL 语句验证删除记录时的错误提示。

```
delete from customers where customer_id = 1
```

错误提示如图 6-14 所示。

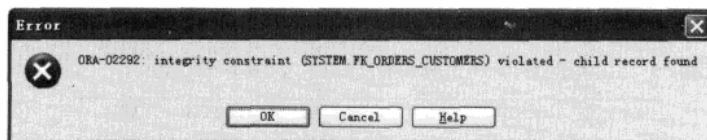


图 6-14 删除具有关联信息的父表记录给出的错误提示

可以利用 SQL 语句验证删除父表的错误提示。

```
DROP TABLE CUSTOMERS
```

错误提示如图 6-15 所示。

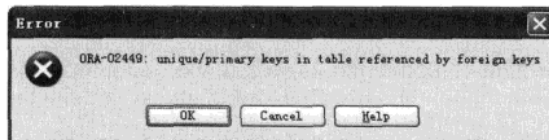


图 6-15 删除具有关联信息的父表的错误提示

即使数据表中没有任何记录，仍然不能在删除子表之前删除父表。这是因为外键约束已经将主表与子表绑定在一起，父表将无法自由地删除。请注意外键约束的两个方面，一方面是对数据的约束，父表的数据和子表的数据必须保持一致；另一方面是对表之间关系的约束，子表依附于父表存在，一旦建立关联关系，父表将不能在删除子表前删除。

6.2.3 级联更新与级联删除

在上一节中可以看到，尝试修改主表中的数据并不一定能够成功。但是有时又的确有这种需求，即修改主表中的主键列的值。当然，子表中的数据也应该同时更新。对于主表中的记录删除亦是如此。但是因为外键约束，造成了两种操作都不能成功进行。这就是级联更新与级联

删除问题的提出背景。

所谓级联更新,是指当父表中的主键列进行修改时,子表的外键列也应该进行相应的修改。级联删除是指当父表中的记录删除时,子表中与之相关的子记录也应该同时删除。

外键约束之所以会限制父表与子表的更新,是以为数据完整性校验无法通过。该校验有两种类型——即时校验(immediate)和延迟校验(deferred)。默认为即时校验,即每执行一条语句,都会进行校验;而延迟校验可以指定校验的时机。

1. 级联更新

其实 Oracle 并未提供命令直接实现级联更新,可以在建立外键时将其设置为延迟校验,然后分别修改主表的主键值和子表的外键列值。

【范例 6-17】演示如何创建延迟校验的外键约束。

```
alter table orders add constraint fk_orders_customers foreign key (customer_id)
references customers (customer_id) deferrable initially deferred
```

【代码说明】defferable 选项用于控制约束校验是否可以执行延迟校验,默认是 not deferrable; initially 用于指定在什么时刻进行校验,有两个备选值: immediate 表示在每条语句(结尾)检查约束(该值为默认值);deferred 表示在事务结尾检查约束。在本例中使用了 deferred,表示在事务结尾处进行约束检查。

【范例 6-18】演示如何修改具有延迟校验外键的表的数据。

```
update customers set customer_id = 3 where customer_id = 1;
update orders set customer_id = 3 where customer_id = 1;
commit ;
```

【代码说明】在本例中,依次修改了主表 customers 的主键 customer_id,子表 orders 的外键列 customer_id 的值,最后执行一次提交动作。

范例 6-18 中的 SQL 语句可以成功执行,可以通过以下语句查看更新结果。

```
select customer_id, customer_name, customer_address, customer_phone, email,
contractor from customers
```

```
select order_id, customer_id, goods_name, quantity, unit from orders
```

查询结果如图 6-16、图 6-17 所示。

CUSTOMER_ID	CUSTOMER_NAME	CUSTOMER_ADDRESS	CUSTOMER_PHONE	EMAIL	CONTRACTOR
3	新晋制衣	解放北路23号	010-23458761	xinjin@xinjin.com	李明浩

图 6-16 更新主表 customers 主键值后的记录

ORDER_ID	CUSTOMER_ID	GOODS_NAME	QUANTITY	UNIT
1	3	FABRIC	52	BMP

图 6-17 更新子表 orders 外键值后的记录

需要注意的是,延迟校验并不等于不进行校验,提交时仍然会对表中数据进行验证。范例 6-19 演示了延迟校验失败的情况。

【范例 6-19】演示提交数据时外键校验失败。

```
UPDATE CUSTOMERS SET CUSTOMER_ID = 2 WHERE CUSTOMER_ID = 1;
```



```
UPDATE ORDERS SET CUSTOMER_ID = 3 WHERE CUSTOMER_ID = 1;
COMMIT ;
```

【代码说明】本例中，将主表中原值为 1 的 customer_id 修改为 2，而将子表 orders 中相应的 customer_id 修改为 3。

在 PL/SQL Developer 中执行该 SQL 语句，将出现如图 6-18 所示的错误提示。

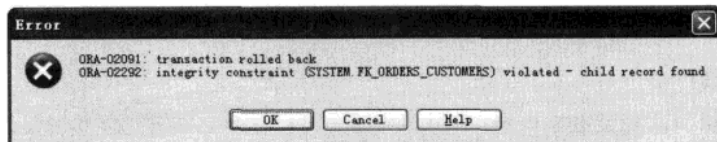


图 6-18 事务结束时外键约束校验数据完整性

从错误提示可以看出，完整性校验失败，事务也被回滚。

2. 级联删除

如果创建外键约束时，显式指定了延迟校验，并且延迟校验的时机为事务提交，那么照样可以通过首先删除主表数据，然后删除子表数据的方式来实现级联更新的效果。另外，Oracle 还提供了一个外键选项 on delete cascade，指定其为级联删除的外键。

on delete cascade 选项指定创建的外键实现级联删除，即删除主表的记录时，子表中的子记录可以被自动删除。删除子表记录的动作是由 Oracle 数据库自动实现的。

【范例 6-20】演示如何创建级联删除的外键约束。

```
alter table orders add constraint fk_orders_customers foreign key (customer_id)
references customers (customer_id) on delete cascade
```

【代码说明】on delete cascade 将新建的外键指定为级联删除。

此时，删除表 customers 中 customer_id 值为 3 的记录。

```
delete from customers where customer_id = 3
```

再次查询表 customers 中的数据，查询结果如图 6-19 所示。

```
select * from customers where customer_id = 3
```

ORDER_ID	CUSTOMER_ID	GOODS_NAME	QUANTITY	UNIT

图 6-19 删除主表中 customer_id 为 3 的数据之后的查询结果

查询表 orders 中的数据，查询结果如图 6-20 所示。

```
select * from orders where customer_id = 3
```

ORDER_ID	CUSTOMER_ID	GOODS_NAME	QUANTITY	UNIT

图 6-20 删除主表中 customer_id 为 3 的数据之后 order 表查询结果

6.2.4 修改外键属性

外键也是约束中的一种，因此可以像修改其他约束一样对其进行修改。修改外键的主要操作有一些几种：重命名、启用/禁用、修改、删除。

1. 重命名外键

重命名外键应该使用 rename 选项。

【范例 6-21】 演示如何将外键 FK_ORDERS_CUSTOMERS 名称修改为 FK_ORDERS。

```
alter table orders rename constraint FK_ORDERS_CUSTOMERS to FK_ORDERS
```

【代码说明】 rename constraint 选项用于重命名约束；FK_ORDERS_CUSTOMERS 为原约束名，FK_ORDERS 为目标约束名。

重新查询表 orders 表的约束信息。

```
select table_name, constraint_name, constraint_type, r_constraint_name from user_constraints where table_name = 'ORDERS'
```

查询结果如图 6-21 所示。

TABLE_NAME	CONSTRAINT_NAME	CONSTRAINT_TYPE	R_CONSTRAINT_NAME
ORDERS	STS_C005180	P	(null)
ORDERS	FK_ORDERS	R	STS_C005181

图 6-21 重命名外键后的约束信息

2. 禁用/启用外键

外键可以被禁用，被禁用后，向表中插入数据或者修改其中数据将不经过约束校验。被禁用的外键可以再次启用，但是在启用时，将执行约束校验。如果其中的数据无法通过约束校验，那么启用外键操作将会失败。可以通过 modify constraint disable/enable 来禁用/启用外键。

【范例 6-22】 演示如何禁用和启用外键约束。

```
alter table orders modify constraint FK_ORDERS disable
```

【代码说明】 modify constraint 选项用于修改约束属性，disable 将使约束失效。

通过以下 SQL 语句查看该外键是否可用。

```
select constraint_name, status from user_constraints where constraint_name = 'FK_ORDERS'
```

查询结果如图 6-22 所示。

CONSTRAINT_NAME	STATUS
FK_ORDERS	DISABLED

图 6-22 外键约束被禁用后的状态

同样，可以使用如下 SQL 语句启用外键约束。

```
alter table orders modify constraint FK_ORDERS enable
```

3. 是否对已有数据进行校验

当禁用了一个外键，用户可能向数据库中插入或者更新了一些数据。当再次启用时，由于数据不一致性，会导致启用失败。但又确实存在这样一种需求，即不需要验证已有的数据，只对以后生成或者修改的数据进行校验，那么，此时可以使用 novalidate/validate 选项。

【范例 6-23】 演示如何利用 novalidate/validate 决定是否验证已有数据。

在表 customers 和表 orders 中当前的数据状态如图 6-23 和图 6-24 所示。



CUSTOMER_ID	CUSTOMER_NAME	CUSTOMER_ADDRESS	CUSTOMER_PHONE	EMAIL	CONTRACTOR
1	新晋制衣	解放北路23号	010-23458761	xinjin@xinjin.com	李明浩

图 6-23 表 customers 的当前数据状态

ORDER_ID	CUSTOMER_ID	GOODS_NAME	QUANTITY	UNIT
1	3	FABRIC	52	BMP

图 6-24 表 orders 的当前数据状态

表 orders 中建立在 customer_id 上的外键处于禁用状态。现使用范例 6-18 中的 SQL 语句启用该外键约束，将会出现启用失败的提示窗口，如图 6-25 所示。

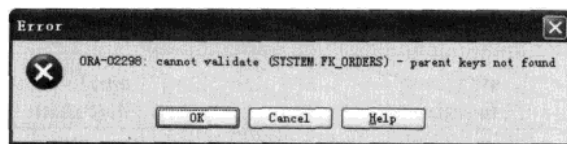


图 6-25 启用外键约束失败

这是因为默认情况下，启用外键时，将会对已有数据进行校验。很明显，customers 表和 orders 表中的数据不符合外键约束规则，所以启用失败。

此时可以使用 novalidate 选项，使 Oracle 不对已有数据进行校验。SQL 语句应该改写为如下形式。

```
alter table orders modify constraint FK_ORDERS enable novalidate
```

在 enable 选项的后面紧跟 novalidate 选项，这使得 Oracle 不对已有数据进行校验，直接启用外键 FK_ORDERS。此时，将能够成功启用外键约束。

需要注意的是，novalidate 选项只是一种临时状态，也就是说，它不会像 enable/disable 一样存储于数据库中，只是在启用外键约束的时刻起作用。当再次启用该约束时，状态仍然是 validate，亦即，仍然需要对已有数据进行外键约束的校验。

启用之后的约束，仍然对 orders 表将来插入的数据起作用。例如，通过如下 SQL 语句向 orders 表中插入非法数据，仍然会抛出错误。

```
insert into orders (order_id, customer_id, goods_name, quantity, unit) values (2, 2, 'FABRIC', 5, 'BMP')
```

4. 删除外键

删除外键使用删除约束的统一语法。即 alter table drop constraint。

【范例 6-24】 演示如何删除 orders 表的外键 FK_ORDER。

```
alter table orders drop constraint FK_ORDERS
```

【代码说明】 drop constraint 为删除约束的统一语法；FK_ORDERS 指定了要删除的外键名称。

6.2.5 外键使用

外键是数据一致性的重要保证，需要注意的问题有以下几个。

1. 严格遵守父子关系的数据表应该使用外键

正如前面 3 节的内容所看到的，形如用户和订单这种具有父子关系的数据表，应该使用外键来建立约束，以防止出现不合理数据。

2. 将应用程序中的父子关系转移到外键约束

很多时候，建立外键的确会将表之间的关系复杂化，但是尽量不要通过应用程序来实现本可由外键实现的完整性检查。因为应用程序实现类似约束，需要考虑的情况比较多，容易出现遗漏。当程序比较复杂时，一旦出现错误，也很难定位错误的位置和原因。而外键约束则由数据库自动处理，对应定位和处理错误非常容易。

3. 不要过分使用外键

合理使用外键可以增加整个应用系统的健壮性，但是不能过分使用外键。首先，使用过多的外键，会增加数据库复杂度，同时降低了设计的可读性。另外，过分使用外键，有可能影响应用系统的效率。例如，在 Hibernate 所映射的表使用了外键，那么在进行查询、修改、删除时，都可能遇到级联问题。对于大数据量的对象，不但要创建这些对象，还要为相应的子记录建立对象，并进行相应处理，效率和性能都会大打折扣。



6.3 唯一性约束

唯一性约束与主键一样，用于唯一标识一行。使用唯一性约束的列或列的组合，其值或值的组合必须是唯一的。

6.3.1 唯一性约束简介

正如在 6.1 节中所描述的那样，主键列上的值都是唯一的，主键是记录唯一性的保证。但是，一个表只能有一个主键。很多时候，对于其他列同样要求列值唯一。例如，在用户表中，列 USER_ID 作为主键可以保证用户的唯一性，但是同时又要求其 E-mail 地址唯一，防止多个用户同时使用同一邮箱。

所以，可以这样理解，主键设计为标识唯一一条记录，而唯一性约束则设计为保证列自身值的唯一性。

6.3.2 创建唯一性约束

唯一性约束和主键约束都是约束的一种，因此具有非常类似的创建语法。本节演示如何创建唯一性约束。

1. 创建唯一性约束

创建唯一性约束，可以在建表时实现。在某个列之后添加 unique 关键字，即可将该列添加唯一性约束。

【范例 6-25】演示如何在创建表时添加唯一性约束。

```
create table users (user_id number primary key, user_name varchar2(50), user_address  
varchar2(50), user_phone varchar2(20) , email varchar2(20) unique, constractor  
varchar2(20))
```

【代码说明】email varchar2(20) unique 用于描述列 email 的属性，unique 关键字为该列添加唯一性约束。



2. 查看唯一性约束

同样可以通过 PL/SQL Developer 中表 USERS 对象的【右键】|【VIEW】|【Index】来查看所建约束，在此不再赘述。另外，可以通过如下 SQL 语句查看表 USERS 的约束。

```
select table_name, constraint_name, constraint_type, status from user_constraints
where table_name = 'USERS'
```

查询结果如图 6-26 所示。

TABLE_NAME	CONSTRAINT_NAME	CONSTRAINT_TYPE	STATUS
USERS	SYS_C005187	P	ENABLED
USERS	SYS_C005188	U	ENABLED

图 6-26 查看表 USERS 的约束

通过查询结果可以看出，名为 SYS_C005188 的约束即为范例 6-25 中创建的唯一性约束。constraint_type 的值为 U，代表 unique。

通过如下 SQL 语句查看该约束所作用的列。

```
select constraint_name, column_name from user_cons_columns where constraint_name
= 'SYS_C005188'
```

查询结果如图 6-27 所示。

CONSTRAINT_NAME	COLUMN_NAME
SYS_C005188	EMAIL

图 6-27 查看唯一性约束作用的列

从图 6-27 的查询结果可以看出，该约束对应的列名为 email。

3. 验证唯一性约束

向表 users 插入两条记录，这两条记录的 email 都为 test@oracle.com。

【范例 6-26】 演示如何向表 users 中插入非法数据，以验证列 email 的唯一性。

```
insert into users (user_id, user_name, user_address, user_phone, email, constructor)
values (1, '张三', '解放路 23 号', '010-22371560', 'test@oracle.com', '张三')
```

在 PL/SQL Developer 中执行该 SQL 语句，将会显示插入成功的信息。再次插入一条新的数据。

```
insert into users (user_id, user_name, user_address, user_phone, email, constructor)
values (2, '李四', '建设路 22 号', '010-83581071', 'test@oracle.com', '李四')
```

在 PL/SQL Developer 中执行该 SQL 语句，将会显示错误信息，如图 6-28 所示。

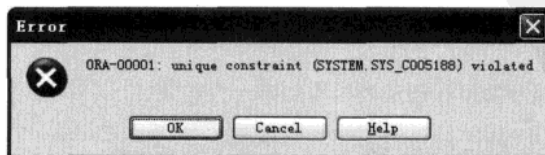


图 6-28 插入违反唯一性约束的数据的错误信息

6.3.3 修改唯一性约束

可以像修改其他约束一样修改唯一性约束，例如添加、删除、重命名、禁用/启用等。

1. 添加唯一性约束

可以在建表之后，添加唯一性约束。

【范例 6-27】 演示如何为已存在的表添加唯一性约束。

```
alter table users add constraint uq_phone unique (customer_phone)
```

【代码说明】add constraint 用于添加一个新的约束；uq_phone 指定了新建约束的名称；unique 关键字指明了该约束的类型为唯一性约束；小括号内的 customer_phone 指明了要在哪个列上添加该约束。



注意：与主键不同，一个唯一性约束只能建立在一个列上，因此，小括号内的列名只能有一个，不能是多个列名的列表。要想在多个列上建立唯一性约束，只能针对每个列分别进行创建。

2. 删除唯一性约束

对于不再需要的唯一性约束，可以使用 drop constraint 选项将其删除。

【范例 6-28】 演示如何删除唯一性约束。

```
alter table users drop constraint uq_phone
```

【代码说明】drop constraint 选项用于删除一个约束；uq_phone 指定了该约束的名称。

3. 重命名唯一性约束

可以重命名已创建的唯一性约束。例如，在范例 6-25 中，创建表的同时创建了列 email 的唯一性约束，但并未为其指定名称。系统自动分配的名称不易识别和记忆，可以通过 rename 选项重命名该约束。

【范例 6-29】 演示如何重命名唯一性约束。

```
alter table users rename constraint sys_c005188 to uq_email
```

【代码说明】rename constraint 选项用于重命名约束；sys_c005188 指定约束的原始名称；uq_email 指定约束的目标名称；原始名称与目标名称之间使用 to 进行连接。

4. 禁用/启用唯一性约束

同样可以对唯一性约束进行禁用/启用操作。

【范例 6-30】 演示如何禁用和启用唯一性约束。

可以使用 disable 选项禁用唯一性约束，SQL 语句如下：

```
alter table users disable constraint uq_email
```

或者

```
alter table users modify constraint uq_email enable
```

禁用 uq_email 之后，再次插入不符合该约束的数据，将能成功进行操作，SQL 语句如下所示：

```
insert into users (user_id, user_name, user_address, user_phone, email, constractor)
```



```
values (2, '李四', '建设路 22 号', '010-83581071', 'test@orale.com', '李四')
```

重新启用该约束的 SQL 语句如下所示：

```
alter table users enable constraint uq_email
```

但是，由于非法数据的存在，将无法成功启用该约束。在 PL/SQL Developer 中的错误提示如图 6-29 所示。

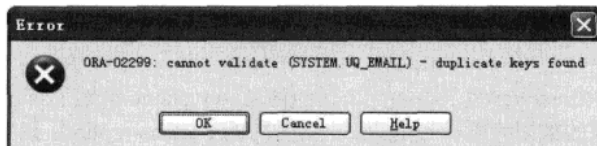


图 6-29 无法成功启用唯一性约束

6.3.4 唯一性约束的使用

唯一性约束可以建立在列或者列的组合上，可以作为主键约束的补充。常见的情形为，主键列为 id，而 id 只是一个自增的整数列，与实际的业务逻辑无关。在业务逻辑上保证记录的唯一性往往使用唯一性约束来实现。



6.4 检查约束

在前面介绍的约束（主键、外键、唯一性约束）实际在定义多个列值之间的关系，例如，主键和唯一性都约束表中的两个列值或列值组合不能相同，而外键则约束了两个表之间的数据保持父子关系。本节介绍的检查约束则是针对列值本身进行限制。

6.4.1 检查约束简介

检查约束对列值进行限制，将表中的一列或多列限制在某个范围内。例如，在学生成绩表中，可能需要将学生单科成绩限制在 0~100 之内，超过 100 分的单科成绩将不能够录入。又如，在员工表中，可能需要限制经理级薪水不能超过 8000，主管级薪水不能超过 5000，普通员工薪水不能超过 4000。这些都可以通过检查约束来实现。

检查约束实际可以看做一个布尔表达式，该布尔表达式如果返回为真，则约束校验将通过，反之，约束校验将无法通过。

6.4.2 创建检查约束

检查约束可以在创建表时进行创建，使用选项 `check`。以创建学生单科成绩不超过 100 为例，范例 6-31 演示了如何创建该约束。

【范例 6-31】 演示如何创建学生单科成绩不超过 100 分的检查约束。

```
create table students (student_id number primary key, student_name varchar2(10),
subject varchar2(20), score number constraint chk_score check (score between 0 and 100))
```

【代码说明】constraint `chk_score` 指定了约束的名称；`check` 指定了约束的类型为检查约束；`score between 0 and 100` 是对约束的描述——`score` 指定列名，`between 0 and 100` 要求 `score` 的值在 0~100 之间。

尝试向其中插入合法数据。


```

insert into scores (score_id, student_name, subject, score) values (1, '张小琴', '
数学', 100)
insert into scores (score_id, student_name, subject, score) values (2, '张小琴', '
语文', 50)
insert into scores (score_id, student_name, subject, score) values (3, '张小琴', '
英语', 0)

```

利用以上 SQL 语句，向 scores 表中插入了三条数据，存储分数的字段 score 的值分别为 100、50 和 0。再向表 scores 中插入新的数据，并使 score 的值分别为 101 和 -1。相应的 SQL 语句如下所示：

```

insert into scores (score_id, student_name, subject, score) values (4, '张小琴', '
物理', 101)
insert into scores (score_id, student_name, subject, score) values (5, '张小琴', '
化学', -1)

```

在 PL/SQL Developer 中，无论执行两条插入语句中的哪一条，都无法通过检查约束的校验。错误提示如图 6-30 所示。

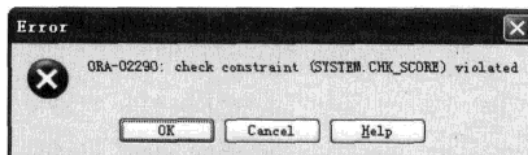


图 6-30 违反检查约束的错误提示

检查约束同样可以建立在多列之上。以薪金控制为例，不但需要将约束的条件建立在人员等级上，同时还要控制薪金的上限。范例 6-32 演示了如何通过多列上建立检查约束实现薪金控制。

【范例 6-32】演示如何利用检查约束限制职位薪金。

```

create table employees (employee_id number primary key, employee_name varchar2(10),
grade varchar2(10), salary number,
constraint chk_salary check (grade in ('MANAGER', 'LEADER', 'STAFF')
and (grade = 'MANAGER' and salary <= 8000 or grade = 'LEADER' and salary <= 5000
or grade = 'STAFF' and salary <= 4000)))

```

【代码说明】constraint chk_salary 用于指定约束的名称；check 用于指定约束的类型为检查约束；grade in ('MANAGER', 'LEADER', 'STAFF') 指定职位必须是“MANAGER”、“LEADER”、“STAFF”中的一种。grade = 'MANAGER' and salary <= 8000 or grade = 'LEADER' and salary <= 5000 or grade = 'STAFF' and salary <= 4000 用于限定不同的职位的薪金上限。

尝试插入合法数据，SQL 语句如下所示：

```

insert into employees (employee_id, employee_name, grade, salary) values (1, '
林新', 'MANAGER', 7800)
insert into employees (employee_id, employee_name, grade, salary) values (2, '
周林', 'LEADER', 4800)
insert into employees (employee_id, employee_name, grade, salary) values (3, '
吴浩', 'MANAGER', 3200)

```

以上 SQL 可以被成功执行。查询表 employees 的数据如图 6-31 所示。

现在尝试向表中插入非法数据：

```

insert into employees (employee_id, employee_name, grade, salary) values (4, '
周兴', 'CEO', 9200)

```



EMPLOYEE_ID	EMPLOYEE_NAME	GRADE	SALARY
1	林新	MANAGER	7800
2	周林	LEADER	4800
3	吴浩	MANAGER	3200

图 6-31 表 employees 的查询结果

在 PL/SQL Developer 中将给出错误提示, 如图 6-32 所示。

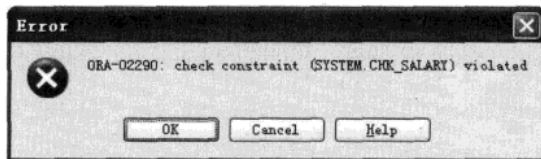


图 6-32 检查约束校验失败时的错误提示

分析插入数据可知, grade 列的期望值为“CEO”, 但是“CEO”并不在“MANAGER”、LEADER”和“STAFF”之中, 所以验证失败。

执行以下 SQL 语句, 尝试插入三条新的记录。

```
insert into employees (employee_id, employee_name, grade, salary) values (4, '王新', 'MANAGER', 8100)
insert into employees (employee_id, employee_name, grade, salary) values (5, '刘军', 'LEADER', 5100)
insert into employees (employee_id, employee_name, grade, salary) values (5, '刘军', 'STAFF', 4100)
```

在 PL/SQL Developer 中执行上述三条中的任意一条 SQL 语句, PL/SQL Developer 都将给出如图 6-32 所示的错误提示。这里需要注意的是, 此时的错误原因不在于插入的 grade 非法, 而是由于每个 grade 的薪金超过了检查约束限制的上限。

检查约束的校验过程。对于更新的每条记录, Oracle 都将计算 check 的布尔值。而以上三条 SQL 语句中插入的记录都将返回 false。

6.4.3 修改检查约束

可以像修改其他约束一样修改检查约束。针对检查约束的操作包括, 添加、删除、重命名和禁用/启用。

1. 添加约束

为表的某个列添加检查约束, 可以使用 add constraint 选项。

【范例 6-33】 演示如何为表 employees 的 employee_name 列添加长度不能超过 4 的约束。

```
alter table employees add constraint chk_name check (length(employee_name) <=4)
```

【代码说明】 chk_name 指定了约束的名称; check 指定约束的类型为检查约束; length(employee_name)<=4 指定 employee_name 的长度不大于 4, 注意这里的长度指的是按字符计算的长度。即无论是单字节还是双字节字符, 都作为一个字符看待。

尝试向表 employees 中插入 employee_name 长度为 5 的记录。

```
insert into employees (employee_id, employee_name, grade, salary) values (5, '欧阳文龙1', 'STAFF', 3100)
```

PL/SQL Developer 将提示如图 6-33 所示的错误。

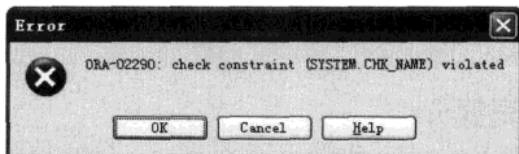


图 6-33 约束 chk_name 校验 employee_name 长度不大于 4

用户会发现员工姓名多输入了一个字符“1”。删除该字符后，重新执行 SQL 语句，可成功插入。

2. 删除检查约束

可以像删除其他约束一样，使用 `drop constraint` 命令来删除一个检查约束。

【范例 6-34】 演示如何删除 `employees` 表的检查约束 `chk_name`。

```
alter table employees drop constraint chk_name
```

【代码说明】 `drop constraint` 选项用于删除一个约束；`chk_name` 指定约束名。

3. 重命名检查约束

可以利用 `rename` 选项来重命名一个检查约束。

【范例 6-35】 演示如何重命名约束 `chk_salary` 为 `chk_grade_salary`。

```
alter table employees rename constraint chk_salary to chk_grade_salary
```

【代码说明】 `rename constraint` 选项用于重命名一个约束；`chk_salary` 为原约束名；`chk_grade_salary` 为目标约束名；二者使用 `to` 进行连接。

4. 禁用/启用检查约束

类似于其他约束，检查约束可以进行禁用/启用操作。

【范例 6-36】 演示如何禁用/启用检查约束。

可以使用 `disable` 选项禁用检查约束。

```
alter table employees disable constraint chk_grade_salary
```

或者：

```
alter table employees modify constraint chk_grade_salary disable
```

可以使用 `enable` 选项启用检查约束。

```
alter table employees enable constraint chk_grade_salary
```

或者：

```
alter table employees modify constraint chk_grade_salary enable
```

6.4.4 检查约束的使用

检查约束可以通过很灵活的约束条件来完成约束任务。但是不能过多使用检查约束，尤其是复杂的约束条件。因为每次更新数据库针对每条记录，都需要进行检查约束的校验，会耗费大量资源。



6.5 默认值约束

默认值约束也是数据库中的常用约束。当向数据表中插入数据时，并不总是将所有字段一



一插入。对于某些特殊字段，其值总是固定或差不多的。用户希望，如果没有显式指定值，就使用某个特定的值进行插入，即默认值。为列指定默认值的操作即为设置默认值约束。

6.5.1 默认值约束简介

就像前面所述，数据表的列可以有非空约束（nullable），所以如果允许列的值为非空的，对于某个字段值不进行显式赋值是允许的。但是，同样可以对其设定默认值约束。一旦设定默认值约束，该列将使用默认值赋值作为空值的替代值。即使未为列指定默认值，那么 Oracle 将隐式使用 null 作为默认值，即 default null。在 Oracle 9i 以前的版本，用户使用默认值只能使用常量值；而 Oracle 9i 及以后版本，用户将可以使用 sysdate 等函数来对列指定默认值。

6.5.2 创建默认值约束

默认值约束的作用对象为列，每个列只能有一个默认值约束。范例 6-37～范例 6-39 演示了如何创建、查看及使用默认值约束。

1. 创建默认值约束

可以在创建表时为列指定默认值。

【范例 6-37】演示如何在创建表时为列指定默认值。

```
create table purchase_order (purchase_order_id number, goods_name varchar(30),
quantity number, price number, status varchar2(3) default 'ACT')
```

【代码说明】status varchar2(3) default 'ACT'用于创建名为 status 的列，该列的数据类型是长度为 3 的可变字符串；default 指定该列的默认值为'ACT'。

2. 查看默认值约束的信息

可以在视图 user_tab_columns 中查看关于列的信息，其中也包含了对列默认值的定义。

【范例 6-38】演示如何查看列的默认值定义。

```
select table_name, column_name, data_type, data_length, data_default from
user_tab_columns where table_name = 'PURCHASE_ORDER' and column_name = 'STATUS'
```

【代码说明】视图 user_tab_columns 中包含了所有用户列的信息：其中，table_name 为表名；column_name 为列名；data_type 为列的数据类型；data_length 为列的长度；data_default 为列的默认值。

执行范例 6-38 的 SQL 语句，查询结果如下所示：

TABLE_NAME	COLUMN_NAME	DATA_TYPE	DATA_LENGTH	DATA_DEFAULT
PURCHASE_ORDER	STATUS	VARCHAR2	3	'ACT'

从查询结果可以看出，列 status 的默认值为“ACT”。

3. 默认值的使用

默认值可以作为保证列值非空的有效手段之一。当向数据表 purchase_order 中插入数据，而未指定 status 列时，将自动使用“ACT”进行填充。

```
insert into purchase_order (purchase_order_id, goods_name, quantity, price) values
(1, 'Sweater', 100, 52)
```

在 PL/SQL Developer 中，查询 purchase_order 中的数据。

```
select * from purchase_order
```

查询结果如下:

PURCHASE_ORDER_ID	GOODS_NAME	QUANTITY	PRICE	STATUS
1	Sweater	100	52	ACT

从查询结果可知, 虽然插入数据时未指定 status 的值, 但数据表中的数据已经默认为“ACT”。

4. 使用函数作为默认值

在 Oracle 9i 及以后的版本, 可以使用函数作为列的默认值。函数 sysdate 用于返回当前日期, 例如, 在销售表可以使用当前日期作为销售时间的默认值。

【范例 6-39】 演示使用 sysdate 函数作为销售时间的默认值。

```
create table sales (id number primary key, product_name varchar2(20), price number,
quantity number, total number, sales_date date default sysdate)
```

【代码说明】 sales_date 是销售时间列, 该列的数据类型为 date, default sysdate 指定使用 sysdate 函数作为 sales_date 的默认值。

向表 sales 中插入两条记录:

```
insert into sales (id, product_name, price, quantity, total) values (1, 'SWEATER',
20, 1, 20)
insert into sales (id, product_name, price, quantity, total) values (2, 'T-SHIRT',
38, 3, 114)
```

在上面的插入语句中, 没有对 sales_date 进行赋值, 查询该表的数据:

```
select * from sales
```

查询结果如下:

ID	PRODUCT_NAME	PRICE	QUANTITY	TOTAL	SALES_DATE
1	SWEATER	20	1	20	2009-5-28 下午 11:33:05
2	T-SHIRT	38	3	114	2009-5-28 下午 11:34:22

分析查询结果可知, 列 sales_date 的值由系统自动生成。

6.5.3 修改默认值约束

对默认值约束的常见操作包括添加、删除、修改等。本节将通过实例进行一一演示。

1. 添加默认值约束

为已有表添加默认值约束, 不能使用 ADD CONSTRAINT 选项, 因为默认值约束实际是列级约束, 所以只能在修改列的同时, 为列添加默认值。

【范例 6-40】 演示如何为已有表添加默认值约束。

```
alter table sales modify quantity number default 1
```

【代码说明】 modify 用于修改表的列, quantity 指定了列名, number 为列的数据类型, default 1 为列指定默认值 1。

向数据表 sales 中插入两条数据:

```
insert into sales (id, product_name, price) values (3, 'SWEATER', 20)
insert into sales (id, product_name, price) values (4, 'T-SHIRT', 38)
```

查询表 sales 中的数据:

```
select * from sales
```



查询结果如下所示:

ID	PRODUCT_NAME	PRICE	QUANTITY	TOTAL	SALES_DATE
1	SWEATER	20	1	20	2009-5-28 下午 11:33:05
2	T-SHIRT	38	3	114	2009-5-28 下午 11:34:22
3	SWEATER	20	1	(null)	2009-5-29 上午 12:05:45
4	T-SHIRT	38	1	(null)	2009-5-29 上午 12:05:48

分析查询结果可知, 新插入的数据 quantity 的值均为 1。

2. 删除默认值约束

删除默认值约束同样不能使用 drop constraint 选项。正如前面所述, 没有添加默认值约束之前, 实际的默认值为 null, 所以, 删除默认值约束, 应该将默认值重新设置为 null。

【范例 6-41】 演示如何删除默认值约束。

```
alter table sales modify quantity number default null
```

【代码说明】 default null 将列 quantity 的默认值设置为 null, 实际上执行了删除默认值约束的功能。

向表中插入新的数据, 如下 SQL 语句所示。

```
insert into sales (id, product_name, price) values (5, 'BLOUSE', 20)
insert into sales (id, product_name, price) values (6, 'BLOUSE', 38)
```

利用如下 SQL 语句查询表 sales 的数据:

```
select * from sales
```

查询结果如下:

ID	PRODUCT_NAME	PRICE	QUANTITY	TOTAL	SALES_DATE
1	SWEATER	20	1	20	2009-5-28 下午 11:33:05
2	T-SHIRT	38	3	114	2009-5-28 下午 11:34:22
3	SWEATER	20	1	(null)	2009-5-29 上午 12:05:45
4	T-SHIRT	38	1	(null)	2009-5-29 上午 12:05:48
5	BLOUSE	20	(null)	(null)	2009-5-29 上午 12:20:27
6	BLOUSE	38	(null)	(null)	2009-5-29 上午 12:20:29

分析查询结果可知, 删除 quantity 的默认值后, 没有指定 quantity 的值时, 实际的插入值为 null。

3. 修改默认值

修改默认值实际应该使用默认值重设功能。例如, 将 sales_date 的默认值, 不保存时间, 只保存日期。

【范例 6-42】 演示只保存日期的默认值。

```
alter table sales modify sales_date default trunc(sysdate, 'DD')
```

【代码说明】 default trunc (sysdate, 'dd') 用于重设默认值, 新的默认值将截断日期, 即时间都使用 12:00。

向表 sales 中插入两条数据。

```
insert into sales (id, product_name, price, quantity, total) values (7, 'TROUSERS',
58, 2, 118)
insert into sales (id, product_name, price, quantity, total) values (8, 'TROUSERS',
60, 1, 60)
```

重新查询表中数据:

```
select * from sales
```

ID	PRODUCT_NAME	PRICE	QUANTITY	TOTAL	SALES_DATE
1	SWEATER	20	1	20	2009-5-28 下午 11:33:05
2	T-SHIRT	38	3	114	2009-5-28 下午 11:34:22
3	SWEATER	20	1	(null)	2009-5-29 上午 12:05:45
4	T-SHIRT	38	1	(null)	2009-5-29 上午 12:05:48
5	BLOUSE	20	(null)	(null)	2009-5-29 上午 12:20:27
6	BLOUSE	38	(null)	(null)	2009-5-29 上午 12:20:29
7	TROUSERS	58	2	118	2009-5-29 上午 12:00:00
8	TROUSERS	60	1	60	2009-5-29 上午 12:00:00

分析查询结果可知, 列 SALES_DATE 的值被默认设为当天日期, 时间都使用了 12:00。



6.6 本章实例

本章讲述了 Oracle 中的几种主要约束, 本节将通过一个实例来演示各种约束的创建及删除过程。

【范例 6-43】 演示各种约束的创建及删除。

(1) 首先利用 pl/sql developer 登录数据库 test, 并打开新的【command window】。

(2) 创建表 teacher, 用于存储教师信息。

```
create table teacher
(id number(3),
teacher_name nvarchar2(30));
```

(3) 为表 teacher 增加各种约束。

```
alter table teacher add constraint pk_teacher primary key(id); --主键约束
alter table teacher add constraint chk_id check(id>0); --check 约束, 要求教师 ID
大于 0
alter table teacher modify teacher_name constraint uq_teachername unique; --唯一
约束
```

(4) 重命名唯一性约束 uq_teachername。

```
alter table teacher rename constraint uq_teachername to uq_teacher;
```

(5) 创建学生表 student, 并为表添加约束。

```
create table student
(id number(10) constraint pk_student primary key, --主键约束
teacher_id number(3) constraint fk_teacher_student references teacher(id), --
外键约束
student_name varchar2(20),
sex varchar2(2) default '男' constraint chk_sex check(sex in ('男', '女')) --check
约束
);
```

(6) 最后, 删除表 student 的所有约束。

```
alter table teacher drop constraint pk_teacher;
alter table teacher drop constraint chk_id;
alter table teacher drop constraint uq_teacher;

alter table student drop constraint pk_student;
alter table student drop constraint fk_teacher_student;
```



```
alter table student drop constraint chk_sex;
```



6.7 本章小结

本章讲述了 Oracle 中的主要的几种约束。注意外键用于约束表之间的关系，而主键和唯一性则约束表中的记录；检查约束和默认值则针对表中列的值。另外，所有约束都具有非常相似的操作，例如，重命名约束、启用/禁用约束等。在学习的过程中，可以进行相互参照，以更好地理解 Oracle 中的约束。



6.8 习题

1. 简述主键的存在意义。
2. 首先创建表 teacher 和 student，并在两个表的 id 之间建立外键约束。
3. 简述唯一性约束与主键的区别。
4. 简述检查约束的作用。
5. 简述默认值约束的作用。



第7章 视图

视图是数据库中特有的对象。视图用于存储查询，但不会存储数据（物化视图除外）。这是视图和数据表的重要区别。可以利用视图进行查询、插入、更新和删除数据。Oracle 中有 4 种视图：

- 关系视图；
- 内嵌视图；
- 对象视图；
- 物化视图。

通过本章的学习，读者可以理解四种视图的概念及各视图的区别，并掌握如何在不同的场合使用不同的视图。



7.1 关系视图

关系视图是四种视图中最简单、同时也最常用的视图。读者可以将关系视图看做对简单或复杂查询的定义。它的输出可以看做一个虚拟的表，该表的数据是由其他基础数据表提供的。由于关系视图并不存储真正的数据，因此占用数据库资源也较少。

7.1.1 建立关系视图

关系视图一旦创建，即可在数据库中查找到其相关信息，也可以作为普通数据表进行查询操作。以下实例演示了如何建立关系视图及相关操作。

1. 创建关系视图

首先创建一个用于测试的数据表 `employees`：

```
create table employees (employee_id number primary key, first_name varchar2(4),  
last_name varchar2(4), province varchar2(10), city varchar2(10), salary number )
```

在该表中，`last_name` 列用于存储员工的姓；`first_name` 列用于存储员工的名；`province` 用于存储员工所在省份；`city` 用于存储员工所在的城市。

表中数据如下：

EMPLOYEE_ID	LAST_NAME	FIRST_NAME	PROVINCE	CITY	SALARY
1	张	军	广东	深圳	3000
2	刘	新	广东	广州	4000
3	王	一帆	广西	桂林	3800
4	周	峰	江苏	苏州	4000
5	徐	佳林	浙江	杭州	2900

为了查询方便，可以创建一个员工视图，该视图可用于直接查询员工的姓名与所在地。创建视图的语法为 `create view`。



【范例 7-1】演示如何创建单纯用于查询的视图。

```
create view vw_employees as
select employee_id, last_name || first_name as employee_name, province || '-' ||
city as location from employees
```

【代码说明】create 用于创建视图；as 关键字表示其后的查询语句为视图定义。last_name || first_name as employee_name 用于连接字符串 last_name 和 first_name，并作为新的列 employee_name 的值；而 province || '-' || city as location 则将省份和城市使用“-”进行连接，并生成新的列 location。

2. 查看视图定义

在创建了视图之后可以通过查询语句查看视图定义。视图 user_views 可用于查询所有用户创建的视图定义。

【范例 7-2】演示如何查询视图 vw_employees 的信息。

```
select text from user_views where view_name = 'VW_EMPLOYEES'
```

【代码说明】view_name 列可用于获取视图的名称；列 text 可用于获取视图的定义。

查询结果如下：

```
TEXT
-----
SELECT EMPLOYEE_ID, LAST_NAME || FIRST_NAME AS EMPLOYEE_NAME, PROVINCE || '-' ||
CITY AS LOCATION
FROM EMPLOYEES
```

3. 查看视图内容

在创建了视图之后，可以通过查询视图内容查看是否达到了预期效果。

【范例 7-3】演示如何查询视图内容。

```
select * from vw_employees
```

查询结果如下：

EMPLOYEE_ID	EMPLOYEE_NAME	LOCATION
1	张军	广东-深圳
2	刘新	广东-广州
3	王一帆	广西-桂林
4	周峰	江苏-苏州
5	徐佳林	浙江-杭州



注意：可以像查询普通表那样来查询视图中的内容。

7.1.2 修改/删除视图

在创建了关系视图 vw_employees 之后，可以对其进行修改和删除操作。

1. 修改视图

修改视图的过程即为重新定义视图的过程，假设现在需要为视图 vw_employees 添加新列 salary，尝试使用如下代码进行重新创建：

```
create view vw_employees as
select employee_id, last_name || first_name as employee_name, province || '-' ||
```

city as location, salary from employees

在 PL/SQL Developer 中执行以上语句，将抛出 Oracle 的 ORA-00995 错误，如图 7-1 所示。

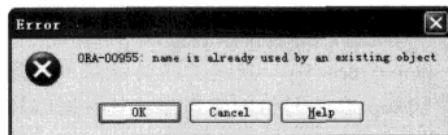


图 7-1 尝试修改视图定义时的错误提示

该错误是因为同名视图，即 `vw_employees` 已存在。可以利用如下 SQL 语句进行修改：

```
create or replace vw_employees as
select employee_id, last_name || first_name as employee_name, province || '-' ||
city as location, salary from employees
```

【代码说明】`create or replace view` 命令用于创建或者替换视图。当同名视图已经存在时，该语句可以执行替换操作。

在 PL/SQL Developer 中成功执行该语句后，可以利用如下 SQL 语句进行查询，以查看是否达到预期效果。

```
select * from vw_employess
```

查询结果如下：

EMPLOYEE_ID	EMPLOYEE_NAME	LOCATION	SALARY
1	张军	广东-深圳	3000
2	刘新	广东-广州	4000
3	王一帆	广西-桂林	3800
4	周峰	江苏-苏州	4000
5	徐佳林	浙江-杭州	2900

2. 删除视图

可以使用 `drop view` 命令删除一个不再需要的视图，以删除视图 `vw_employee` 为例，相应的 SQL 语句如范例 7-4 所示。

【范例 7-4】演示如何删除视图。

```
drop view vw_employees
```

【代码说明】`drop view` 命令用于删除一个视图。需要注意的是，在 Oracle 中删除对象一般使用 `drop` 命令，而 `delete` 命令用于删除表中数据。

在 PL/SQL Developer 中执行以上删除语句后，再次执行查询语句：

```
select * from vw_employess
```

PL/SQL Developer 将抛出如图 7-2 所示的错误。

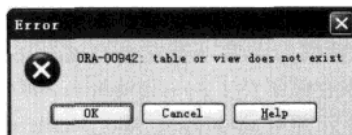


图 7-2 删除视图后，重新进行查询，PL/SQL 将抛出错误

通过错误提示可知，视图 `vw_employees` 已经成功删除。



7.1.3 联接视图

在范例 7-1 中所创建的视图是基于单个基础表的。而在实际应用中，大多数的视图是较为复杂的查询。这也是视图的一大优势，封装复杂查询。

在范例 7-1 所建的表 `employees` 中，描述了雇员的基本信息。现假设现有另外一个表 `employee_sales`，该表用于存储雇员在 3 月份的零售额。其数据结构及实际数据如下：

ID	SALE_MONTH	TOTAL_PRICE	SALE_BY
1	3 月	1200	1
2	3 月	1500	2
3	3 月	3000	3
4	3 月	1480	4
5	3 月	1760	5

该表描述了雇员在 3 月份的销售额。`sale_by` 并未存储雇员的姓名，而是存储了雇员的 id 号。对于每一位雇员，3 月份的实际工资为基本工资+销售额的 10%。那么可以建立名为 `vw_employee_salary` 的视图，用于自动结算员工的月工资。

【范例 7-5】 演示如何创建自动结算员工 3 月份工资的视图。

```
create or replace view vw_employee_salary
as select e.employee_id, e.last_name || e.first_name employee_name, e.salary +
s.total_price * 0.1 salary
from employees e, employee_sales s
where e.employee_id = s.sale_by
and s.sale_month = '3 月'
```

【代码说明】 该视图使用了两个表的联接，联接条件为 `employees` 表的 `employee_id` 的值等于 `employee_sales` 表的 `sale_by` 的值，并且销售月份为 3 月份；实际工资额为基本工资额+销售额*10%。

查询视图 `vw_employee_salary`：

```
select * from vw_employee_salary
```

查询结果如下：

EMPLOYEE_ID	EMPLOYEE_NAME	SALARY
1	张军	3120
2	刘新	4150
3	王一帆	4100
4	周峰	4148
5	徐佳林	3076

在建立了该视图之后，就可以直接利用该视图获得员工 3 月份的应得工资，而无须每次结算都联接数据表。

视图联接时，还可以使用另外一个视图与表或其他视图进行联接。因为视图本身就是作为一个查询定义存在的。例如，数据库中并不存在名为 `employee_sales` 的表，而只含有名为 `employee_sale_details` 的表。该表存储了员工的销售明细表，即每项销售记录分别进行记录。为了获得员工的月结工资，可以首先利用 `employee_sale_details` 表得到一个视图。该视图用于获得月度的销售总额，然后利用表 `employees` 与该视图进行联接，以获得最终的月结工资。

【范例 7-6】 演示如何利用表与视图进行联接获得新的视图。

表 `employee_sale_details` 的数据结构及内容如下：

SALE_DETAILS_ID	GOODS_NAME	PRICE	QUANTITY	SALE_MONTH	SALE_BY
1	西装	300	2	3月	1
2	毛衣	20	20	3月	1
3	西裤	100	2	3月	1
4	运动鞋	80	10	3月	2
5	西装	300	1	3月	2
6	皮鞋	100	4	3月	2
7	运动鞋	80	10	3月	3
8	皮鞋	100	15	3月	3
9	衬衫	70	10	3月	3
10	衬衫	50	8	3月	4
11	T恤	60	8	3月	4
12	皮鞋	100	6	3月	4
13	运动衫	40	9	3月	5
14	羊毛衫	80	10	3月	5
15	西裤	60	10	3月	5

可以创建一个名为 `vw_employee_sales` 的视图，该视图存储了员工月度销售额的查询定义，其 SQL 语句如下：

```
create or replace view vw_employee_sales
as select sale_by, sum(price * quantity) total_price
from employee_sale_details
group by sale_by
```

该视图使用了分组查询，`group by sale_by` 指定按照销售人员的 id 号进行分组；`sum(price * quantity)` 用于获得月度的销售总额。

查询该视图：

```
select * from vw_employee_sales
```

查询结果如下：

SALE_BY	TOTAL_PRICE
1	1200
2	1500
3	3000
4	1480
5	1760

此时，利用该视图与表 `employees` 进行联接来获得员工月结工资，代码如下：

```
create or replace view vw_employee_salary
as select e.employee_id, e.last_name || e.first_name employee_name, e.salary +
s.total_price * 0.1 salary
from employees e, vw_employee_sales s
where e.employee_id = s.sale_by
```

在以上 SQL 代码中，利用了表 `employee` 与视图 `vw_employee_sales` 联接，联接条件为表 `employee` 的 `id` 列的值等于视图 `vw_employee_sales` 的 `sale_by` 列的值。

查询视图 `vw_employee_salary` 的内容：

```
select * from vw_employee_salary
```

查询结果如下：

EMPLOYEE_ID	EMPLOYEE_NAME	SALARY
1	张军	3120
2	刘新	4150



3	王一帆	4100
4	周峰	4148
5	徐佳林	3076

比较两次创建的 `VW_EMPLOYEE_SALARY` 的内容可知, 利用视图与表联接可以获得正确的结果。当然, 也可以利用视图与视图联接来获得第三个视图, 创建视图的方法与范例 7-1 和范例 7-5 相同, 在此不再一一赘述。

7.1.4 编译视图

视图依赖于基础表的存在而存在, 当基础表进行了结构上的调整时, 有可能会对视图产生影响。如要再次使用该视图, 必须对视图进行重新编译。

【范例 7-7】 演示基础表修改后, 对视图进行重编译。

(1) 以范例 7-5 创建的视图 `vw_employee_salary` 为例, 查看状态信息的 SQL 语句如下所示:

```
select object_name, status from user_objects where object_name = 'VW_EMPLOYEE_SALARY'
and object_type = 'VIEW'
```

【代码说明】 `user_objects` 视图可用于查询当前数据库中所有对象的信息, 这些对象包括了表、视图、约束等; `object_name` 列存储了对象名; `status` 列存储了对象的状态。

查询结果如下:

OBJECT_NAME	STATUS
-----	----
VW_EMPLOYEE_SALARY	VALID

分析查询结果可知, 视图 `vw_employee_salary` 的状态为 `VALID` (有效)。

(2) 此时, 修改表 `employees` 的结构, 相应的 SQL 语句如下:

```
alter table employees add (age number)
```

【代码说明】 该 SQL 语句用于为表 `employees` 表添加一个列 `age` (年龄)。

(3) 再次查询视图 `vw_employee_salary` 的状态, 查询结果如下:

OBJECT_NAME	STATUS
-----	----
VW_EMPLOYEE_SALARY	INVALID

因为视图 `vw_employee_salary` 的基础表 `employees` 发生了结构上的变化, 所以视图的状态被 Oracle 置为 `INVALID` (无效)。在重新编译之前, 不可以直接使用该视图。

(4) 编译视图可以利用 `alter view` 命令, 重新编译 `vw_employee_salary` 的 SQL 语句如下所示:

```
alter view vw_employee_salary compile
```

【代码说明】 `alter view` 命令用于修改视图, `compile` 选项表示要重新编译视图。

(5) 在重编译视图 `vw_employee_salary` 之后, 再次查询其状态, 查询结果如下:

OBJECT_NAME	STATUS
-----	----
VW_EMPLOYEE_SALARY	VALID

可见, 重新编译之后, 视图状态被重新置为可用。

(6) 其实, 只要重新执行一次对视图的查询操作, 也可将视图状态置为可用。在视图 `vw_employee_salary` 无效的状态下, 重新查询视图:

```
select * from vw_employee_salary
```

此时, 再次执行视图状态的查询语句:

```
select object_name, status from user_objects where object_name = 'VW_EMPLOYEE_SALARY'
and object_type = 'VIEW'
```

查询结果如下所示:

```
OBJECT_NAME          STATUS
-----
VW_EMPLOYEE_SALARY    VALID
```

这是因为 Oracle 在每次执行真正查询之前, 会自动重新编译视图。

(7) 需要注意的是, 只有修改基础表的结构才会影响视图的有效性, 修改基础表的数据并不影响视图的有效性。以向表 `employees` 中插入一条数据为例, 验证视图的有效性跟数据内容的关系。

```
insert into employees values (6, '秦', '君', '陕西', '西安', 2700, 22)
```

【代码说明】在上面的示例中, 为表 `employees` 添加了一个列 `age`, 因此插入新数据时, 为 `age` 赋值 22。

此时表 `employees` 的数据如下:

EMPLOYEE_ID	LAST_NAME	FIRST_NAME	PROVINCE	CITY	SALARY	AGE
1	张	军	广东	深圳	3000	(null)
2	刘	新	广东	广州	4000	(null)
3	王	一帆	广西	桂林	3800	(null)
4	周	峰	江苏	苏州	4000	(null)
5	徐	佳林	浙江	杭州	2900	(null)
6	秦	君	陕西	西安	2700	22

查询视图 `vw_employee_salary` 的状态, 结果如下所示:

```
OBJECT_NAME          STATUS
-----
VW_EMPLOYEE_SALARY    VALID
```

可见, 修改基础表的数据并不会影响视图的有效性。

(8) 但是, 并非所有针对基础表结构的修改, 都可以在视图进行重编译时顺利通过。例如, 将基础表中的某列进行重命名操作, 而该列又出现在视图定义中, 此时, 视图将无法顺利编译。以修改基础表 `employees` 中的列名 `last_name` 为 `family_name` 为例:

```
alter table employees rename column last_name to family_name
```

在 PL/SQL Developer 的【Command Window】或 SQL*Plus 中查看此时基础表 `employees` 的结构:

```
sql> desc employees;
```

Name	Type	Nullable	Default	Comments
EMPLOYEE_ID	NUMBER			
FAMILY_NAME	VARCHAR2(4)	Y		
FIRST_NAME	VARCHAR2(4)	Y		
PROVINCE	VARCHAR2(10)	Y		
CITY	VARCHAR2(10)	Y		
SALARY	NUMBER	Y	0	
AGE	NUMBER	Y		

分析查询结果可知, 列 `last_name` 的名称已经修改为 `family_name`。此时视图 `vw_employee_salary` 将被置为无效。在 PL/SQL Developer 中, 重新查询视图 `vw_employee_salary` 的数据, 以便其被重新编译:

```
select * from vw_employee_salary compile
```



PL/SQL Developer 将给出错误提示, 如图 7-3 所示。

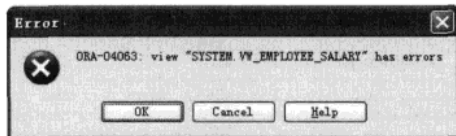


图 7-3 视图无法成功编译, 查询视图将出现错误提示

此时查询视图的状态, 结果如下:

OBJECT_NAME	STATUS
VW_EMPLOYEE_SALARY	INVALID

7.1.5 使用 force 选项强制创建视图

有时, 视图的基础表还没有创建, 但是仍然希望能够创建基于不存在的数据表的视图。这是因为, 数据表的预期创建者和视图的创建者并非同一用户。视图创建者不希望等待基础表创建者的工作完成之后才进行自己的工作。假设, 此时视图创建者已经对基础表的结构有了清晰的概念, 例如, 表 books 将被创建, 以存储图书信息。其预期的表结构如下:

Name	Type	Nullable
BOOK_ID	NUMBER	Y
BOOK_NAME	VARCHAR2(50)	Y
AUTHOR	VARCHAR2(20)	Y
PRESS	VARCHAR2(50)	Y
PUBLISH_DATE	DATE	Y

现要创建一个仅包含 book_name 和 author 信息的视图, 根据前面所讲述的内容。相应的 SQL 语句如下:

```
create or replace view vw_books as
select book_name, author
from books
```

因为基础表 books 在数据库中并不存在, 所以 Oracle 将抛出错误, 以 PL/SQL Developer 为例, 错误如图 7-4 所示。

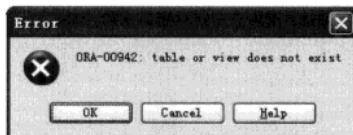


图 7-4 利用不存在的基础表创建视图, 将抛出错误

为了达到目的, 可以强制创建该视图。强制创建视图应该使用 force 选项, 以创建 vw_books 视图为例, 相应的 SQL 语句如范例 7-8 所示。

【范例 7-8】 演示如何强制创建视图 vw_books。

```
create or replace force view vw_books as
select book_name, author
from books
```

【代码说明】 force 选项用于强制创建视图 vw_books。

执行该 SQL 语句, Oracle 将创建视图, 但是仍然会有错误提示, 如下所示:

W (1): Warning: Completed with errors

此时，可以在数据词典中查找到该视图的基本信息。

```
select object_name, status from user_objects where object_name = 'VW_EMPLOYEE_SALARY'
and object_type = 'VIEW'
```

查询结果如下：

OBJECT_NAME	STATUS
VW_EMPLOYEE_SALARY	INVALID

通过查询结果可知，虽然该视图已经创建，但其状态仍为 INVALID（不可用）。当数据表 books 一旦按照预期结构被创建，那么该视图将能够正常使用。

7.1.6 利用视图更新数据表

通过视图，不但可以对基础表中的数据进行查询，而且可以对数据表中的数据进行更新。更新的方式非常简单——直接更新视图中的数据即可对基础表进行相应的更新。当然，并非视图中的所有列都能够进行更新，并反映到基础表中。只有那些直接由基础表获得的列可以进行更新操作，如范例 7-5 中的列 book_name 和列 author；而由基础表中的数据经过运算获得，仅凭视图中的数据无法判断基础表中的数据情况的列，不能进行更新，如范例 7-1 中的 employee_name 列和 location 列。

可以在 user_updatable_columns 中获得用户创建的所有视图的列是否可以更新的信息。

【范例 7-9】演示如何获得视图 vw_employee_salary 所有列的可更新情况。

```
select table_name, column_name, updatable, insertable, deletable from
user_updatable_columns where table_name = 'VW_EMPLOYEE_SALARY'
```

【代码说明】视图 vw_employee_salary 的列 table_name 为表名，也包括了视图名；column_name 为表的列名；updatable、insertable 和 deletable 表征了对应的列是否可进行更新、插入和删除操作。

查询结果如下：

TABLE_NAME	COLUMN_NAME	UPDATABLE	INSERTABLE	DELETABLE
VW_EMPLOYEE_SALARY	EMPLOYEE_ID	NO	NO	NO
VW_EMPLOYEE_SALARY	EMPLOYEE_NAME	NO	NO	NO
VW_EMPLOYEE_SALARY	SALARY	NO	NO	NO

分析查询结果可知，三个列 employee_id、employee_name 和 salary 均不可更改。

(1) 现尝试修改 employee_id 列：

```
update vw_employee_salary set employee_id = employee_id + 1
```

尝试将列 employee_id 的列值增加 1，在 PL/SQL Developer 中将出现如图 7-5 所示的错误提示。

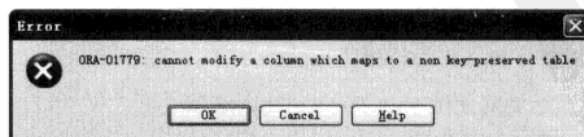


图 7-5 尝试修改列 employee_id 的错误提示

ORA-01779: cannot modify a column which maps to a non-key-preserved table, 该错误是因为



尝试修改的列，在基础表中的映射并没有唯一性约束。在表 `employees` 中，`employee_id` 被设为主键，`employee_id` 值唯一；但是在另外一个基础表 `employee_sale_details` 中 `employee_id` 列实际为 `sale_by`，利用某个 `employee_id` 定位 `employee_sale_details` 中的记录时，可能定位到多条，因为 `sale_by` 并非主键，也没有唯一性约束。

(2) 现要为每个员工加薪 300 元，尝试使用视图进行更新：

```
update vw_employee_salary set salary = salary + 300
```

在 PL/SQL Developer 中，将抛出如图 7-6 所示的错误提示。

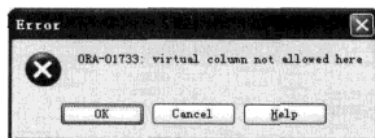


图 7-6 更新虚列时出错

错误 ORA-01733: virtual column not allowed here 显示，`salary` 为一个虚列，无法进行更新。虚列是指，并没有真正的基础表的列与之对应的列。虚列一般由基础表的列进行运算生成。

(3) 接着查看视图 `vw_books` 的列的可更新情况：

```
select table_name, column_name, updatable, insertable, deletable from user_
updatable_columns where table_name = 'VW_BOOKS'
```

查询结果如下：

TABLE_NAME	COLUMN_NAME	UPDATABLE	INSERTABLE	DELETABLE
VW_BOOKS	BOOK_NAME	YES	YES	YES
VW_BOOKS	AUTHOR	YES	YES	YES

通过分析可知，视图 `vw_books` 的所有列均可进行更新、插入和删除的操作。

(4) 现尝试对其进行插入操作：

```
insert into vw_books values ('千年一叹', '余秋雨')
```

在 PL/SQL Developer 中成功执行该语句后，查询基础表 `books` 的内容。

```
select * from books
```

查询结果如下所示：

BOOK_ID	BOOK_NAME	AUTHOR	PRESS	PUBLISH_DATE
(null)	千年一叹	余秋雨	(null)	(null)

分析查询结果可知，通过向视图 `vw_books` 插入数据，的确反映在了基础表上。虽然表的其他列的数据均为 `null`。

7.1.7 with check option 选项

创建视图时，`with check option` 是常用的选项之一。一旦使用了该选项，那么 Oracle 将保证视图在数据更新之后与更新之前的结果集相同。

【范例 7-10】 演示如何使用 `with check option` 选项。

向基础表 `books` 中插入如下数据。

BOOK_ID	BOOK_NAME	AUTHOR	PRESS	PUBLISH_DATE
1	围城	钱钟书	人民文学出版社	2006-4-9

2	千年一叹	余秋雨	作家出版社	2000-5-1
3	平凡的世界	路遥	北京十月文艺出版社	2009-3-1
4	老人与海	海明威	人民文学出版社	1952-8-1
5	中国大历史	黄仁宇	中华书局	2002-3-3

(1) 创建一个视图，该视图中仅存在选择 ID 号为 5 的图书信息。

```
create or replace view vw_books as
select * from books
where book_id = 5
```

查询该视图：

```
select * from vw_books
```

查询结果如下：

BOOK_ID	BOOK_NAME	AUTHOR	PRESS	PUBLISH_DATE
5	中国大历史	黄仁宇	人民文学出版社	2002-3-3

(2) 此时通过视图将 book_id 的值修改为 6，代码如下：

```
update vw_books set book_id = 6
```

此时查询基础表 books，查询结果如下：

BOOK_ID	BOOK_NAME	AUTHOR	PRESS	PUBLISH_DATE
1	围城	钱钟书	人民文学出版社	2006-4-9
2	千年一叹	余秋雨	作家出版社	2000-5-1
3	平凡的世界	路遥	北京十月文艺出版社	2009-3-1
4	老人与海	海明威	人民文学出版社	1952-8-1
6	中国大历史	黄仁宇	中华书局	2002-3-3

可见，此时已经成功将 book_id 由 5 修改为了 6。但是会对视图产生很大的影响，因为视图的查询条中限制了 book_id 为 5。此时，查询视图 vw_books 中的数据：

```
select * from vw_books
```

查询结果如下所示：

BOOK_ID	BOOK_NAME	AUTHOR	PRESS	PUBLISH_DATE

可见，查询结果为空。

(3) 为了防止这种情况，可以利用 with check option 选项。首先将 books 表的内容恢复，然后重新创建视图。

```
update books set book_id = 5 where book_id = 6
```

创建视图的代码应该修改为：

```
create or replace view vw_books as
select * from books
where book_id = 5
with check option
```

【代码说明】with check option 用于保证视图的数据完整性。

(4) 此时尝试更新 book_id 的值：

```
update vw_books set book_id = 6
```

Oracle 将抛出错误，因为修改之后的数据将使视图查询结果发生改变。在 PL/SQL Developer 中的错误提示如图 7-7 所示。

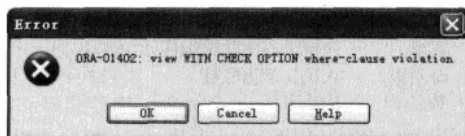


图 7-7 check with option 选项保护视图数据

从错误提示可以看出，因为列 `book_id` 出现在视图定义的 `where` 子句中。修改之后的数据会影响到视图的查询结果，所以 Oracle 禁止其修改。

7.1.8 关系视图小结

通过以上内容的讲述，可知使用关系视图有以下好处。

1. 保障数据安全性

在范例 7-1 中，并没有将员工的 `salary` 列即薪水信息置于视图中。如果将原始的数据表 `employees` 进行保护，而暴露给用户的仅仅是视图 `vw_employees`，那么 `salary` 信息将被有效地保护起来。

2. 数据整合

在试图 `vw_employees` 中，将 `first_name` 与 `last_name` 进行连接，组成了新的 `employee_name`。用户无须每次自行整合数据，即可得到结果数据。当然，可以通过对原数据表的更复杂的运算进行更加强大的数据整合。

3. 数据透明性

在视图中使用自定义运算，可以建立新的列。但是，这些列对用户来说是透明的，用户无须关心视图的实现细节即可使用其中的数据。



7.2 内嵌视图

关系视图作为查询定义，一旦创建，即可存在于数据库中，并可被多次使用。但有时，需要某个视图作为过渡结果集，但在一次使用之后，便不再需要，此时不宜创建关系视图。因为关系视图占用数据库资源，而且也会增加维护成本。此时应该选择使用内嵌视图。

7.2.1 内嵌视图简介

顾名思义，内嵌视图也是视图，只是不会利用 `create view` 进行显式创建。也不能在数据库中查询到其相关信息。一般情况下，被嵌套在查询语句中使用，因此称为内嵌视图。其功能类似于子查询。

当然，内嵌视图也可以出现在更新、插入和删除语句中。内嵌视图具有临时性，它只在被嵌入语句的执行期间有效，并且可以在被嵌入语句的任何地方使用该视图。

7.2.2 内嵌视图的使用

内嵌视图无须显式创建，但有时需要为其指定别名，以便进行引用。本节将通过几个实例来演示如何使用内嵌视图进行查询、更改、插入和删除操作。

1. 利用内嵌视图进行查询

在范例 7-6 中, 为了获得所有员工的月度销售总额, 创建了视图 `vw_employee_sales`。此处, 可以利用内嵌视图来代替关系视图 `vw_employee_sales`。

【范例 7-11】 演示如何利用内嵌视图代替关系视图。

```
select e.employee_id, e.last_name || e.first_name employee_name, e.salary + s.
total_price * 0.1 salary
from employees e,
     (select sale_by, sum(price * quantity) total_price
      from employee_sale_details group by sale_by) s
where e.employee_id = s.sale_by
```

【代码说明】 查询语句 `select sale_by, sum(price * quantity) total_price from employee_sale_details group by sale_by` 用于获得各员工的销售总额。该查询语句即可看做内嵌视图。其后紧跟视图的别名 `s`, 用于在整条语句中引用该视图。

查询结果如下:

EMPLOYEE_ID	EMPLOYEE_NAME	SALARY
1	张军	3120
2	刘新	4150
3	王一帆	4100
4	周峰	4148
5	徐佳林	3076

分析查询结果可知, 该查询语句的最终结果与范例 7-6 中的视图完全一致。

内嵌视图的别名并不是必需的, 范例 7-11 中使用了视图别名是因为整个查询中有另外一个表 `employees`, 为了区别二者的引用而使用视图别名。当 `from` 子句指向的目标只有内嵌视图时, 其别名可以省略。

【范例 7-12】 演示不使用别名的内嵌视图。

现需查找员工基本工资处于前三位的员工资料, 那么, 可以首先对表中数据进行排序, 然后再取前面三条记录。对表中数据按员工基本工资进行排序的 SQL 语句如下所示:

```
select employee_id, last_name || first_name employee_name, salary
from employees
order by salary desc
```

【代码说明】 `order by` 为排序选项, `salary` 为排序标准, `desc` 表示对 `salary` 的降序排列, 即按照工资标准由高到低进行排序。

查询结果如下:

EMPLOYEE_ID	EMPLOYEE_NAME	SALARY
2	刘新	4000
4	周峰	4000
3	王一帆	3800
1	张军	3000
5	徐佳林	2900
6	秦君	2700

此时, 可以将以上查询语句作为内嵌视图置于获取前三位员工的查询语句中, 相应的 SQL 语句如下:

```
select * from
(select employee_id, last_name || first_name employee_name, salary
```



```

from employees
order by salary desc)
where rownum <= 3

```

【代码说明】此处，`select employee_id, last_name || first_name employee_name, salary from employees order by salary desc` 作为内嵌视图存在，并不需要为其指定别名，这是因为 `from` 子句中没有其他表或视图与之产生冲突；`rownum` 指定了查询结果的行号，当 `rownum<=3` 意味着工资由高到低排列的前三条记录。

查询结果如下：

EMPLOYEE_ID	EMPLOYEE_NAME	SALARY
2	刘新	4000
4	周峰	4000
3	王一帆	3800

2. 利用内嵌视图更新数据表

更新关系视图的操作可以更新基础表中的数据。同样，可以利用内嵌视图对数据表进行更新。例如，为所有员工增加 300 元的代码如范例 7-13 所示。

【范例 7-13】演示如何利用内嵌视图为所有员工增加 300 元工资。

```

update
(select salary
 from employees
)
set salary = salary + 300

```

再次查询 `employee` 表的数据。

```
select employee_id, last_name || first_name, salary from employees
```

查询结果如下：

EMPLOYEE_ID	LAST_NAME FIRST_NAME	SALARY
1	张军	3300
2	刘新	4300
3	王一帆	4100
4	周峰	4300
5	徐佳林	3200
6	秦君	3000

范例 7-13 的查询语句当然可以使用 `update employees set salary = salary + 300` 来代替。通过内嵌视图来更新基础表的数据并不常见，但该用法反映了一种情况，即可以通过内嵌视图来更新基础表的数据。

同样，可以利用内嵌视图进行删除和插入数据的操作，在此不再一一赘述。

7.2.3 内嵌视图小结

在讲述了内嵌视图之后，本节对内嵌视图的使用进行总结。

1. 内嵌视图与临时表、关系视图的使用场景

内嵌视图并不真正创建视图，因此，相对于关系视图和临时表，不会进行大量的 I/O 操作，可以节省大量的数据库资源。如果需要在查询中临时使用另外的查询结果，可以考虑使用内嵌视图。但是，如果某个查询会被多处使用，那么可复用性便非常重要，此时，应该考虑使用临时表或者关系视图。

2. 分辨内嵌视图与子查询

Oracle 中允许使用子查询，而且子查询与内嵌视图非常相像。分辨二者的标准为，在 SQL 语句中，代替了表的位置的查询即为内嵌视图。例如，from 和 into 关键字之后的查询语句，即为内嵌视图。注意内嵌视图的定义必须使用小括号括起来。



7.3 对象视图

目前，数据库都是关系型数据库，但是，面向对象编程的思想早已深入人心。Oracle 数据库不仅可以通过关系表来存储数据，同样可以创建对象，以对象的方式进行数据存储。关系视图是由关系表进行查询获得的，而对象视图则是对对象进行查询获得的。

7.3.1 对象视图简介

说到对象视图，自然应该首先提到对象。Oracle 中的对象仍然是一个逻辑概念，虽然可以从对象中获取数据，就像数据真实存储于对象中一样。但是在对象的概念之下，数据仍然是存储于关系表中的。要创建对象，首先要建立对象类型，类似于 Java 或 C# 中类的概念。以创建 employee 对象类型为例，相应的 SQL 语句如下所示：

```
create type employee is object
(employee_id number, employee_name varchar2(20), location varchar2(50), salary
number)
```

【代码说明】create type 命令用于创建新的类型；employee 则指定了新类型的名称；is object 表示该类型为一个对象；(employee_id number, employee_name varchar2(20), location varchar2(50), salary number) 则为对象类型中各属性的具体定义，对比创建表的语法可知，对象类型的中属性的定义与表中列的定义完全一致。

对象视图是基于对象类型来创建的，因此，在创建了对象类型之后，即可创建对象视图。

7.3.2 对象视图简介

利用基础对象类型 employee 来创建对象视图，那么视图中的列将与 employee_type 中的属性保持一致。

【范例 7-14】演示如何创建对象视图。

```
create or replace view ov_employees
of employee
with object oid(employee_id) as
select employee_id, last_name || first_name, province || city, salary
from employees
```

【代码说明】create or replace view 用于创建或者替换一个视图；ov_employees 标识了新视图的名称，在此特地使用了 ov 前缀，表示该视图是一个对象视图；of employee 表示新视图是基于对象类型 employee；with object oid 则用于标识对象中的主键，该主键是 employee_id 属性；select employee_id, last_name || first_name, province || city, salary from employees 则为视图提供数据源，此处选择的列会与对象中的属性按照顺序一一对应。



注意：oid 的值 employee_id 指的是对象 employee 中的属性，并非基础表 employees 中的 employee_id 列。



可以在 PL/SQL Developer 的【Command Window】或 SQL*Plus 中查看视图对象
ov_employees:

```
SQL> desc ov_employees;
Name                Type                Nullable  Default  Comments
-----
EMPLOYEE_ID         NUMBER              Y
EMPLOYEE_NAME       VARCHAR2(8)         Y
LOCATION             VARCHAR2(20)        Y
SALARY              NUMBER              Y
```

在数据库中查询对象视图 ov_employees 的信息, 并对比关系视图 vw_employee_salary:

```
select view_name, view_type, oid_text
from user_views
where view_name = 'OV_EMPLOYEES' or view_name = 'VW_EMPLOYEE_SALARY'
```

【代码说明】利用视图 user_views 可以对所有用户视图信息进行查询; 列 view_name 是关于视图名称的列; view_type 是视图基于的对象类型; 而 oid_text 则指定了对象中的主键。

查询结果如下:

VIEW_NAME	VIEW_TYPE	OID_TEXT
OV_EMPLOYEES	EMPLOYEE	EMPLOYEE_ID
VW_EMPLOYEE_SALARY	(null)	(null)

分析查询结果可知, 关系视图的 view_type 和 oid_text 列均为 null。可以通过这两列判断某个视图是否为对象视图。

另外, 从基础表中获得的各列数据将和对象中的属性按照定义顺序一一对应, 不能颠倒。例如, 将视图 ov_employees 的定义修改如下:

```
create or replace view ov_employees
of employee
with object oid(employee_id) as
select last_name || first_name, province || city, salary, employee_id
from employees
```

在 PL/SQL Developer 中执行该语句, 将抛出错误提示, 如图 7-8 所示。

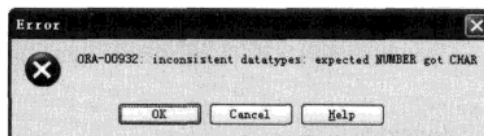


图 7-8 创建对象视图失败的错误提示

这是因为该创建语句将 last_name||first_name 的值填充到对象 employee 的 employee_id 属性。该属性的数据类型为数值型, 而 last_name||first_name 的结果为字符串, 所以无法成功创建。

在创建对象视图 ov_employees 之后, 可以对其直接进行查询操作, 代码如下:

```
select * from ov_employees
```

查询结果如下:

EMPLOYEE_ID	EMPLOYEE_NAME	LOCATION	SALARY
1	张军	广东深圳	3300
2	刘新	广东广州	4300
3	王一帆	广西桂林	4100
4	周峰	江苏苏州	4300
5	徐佳林	浙江杭州	3200

对象视图可以和关系视图进行一样，可以进行增删改查的操作，在此不再一一赘述。



7.4 物化视图

前面所讲述的三种视图——关系视图、内嵌视图和对象视图，实际都是通过定制查询，并利用查询定义来获取数据。三种视图都不会直接存储数据，每次操作时，都会进行编译。本节讲述另外一种视图——物化视图。

7.4.1 物化视图简介

物化视图是物理化视图的简称，顾名思义，该视图存储实际数据，因此，会占用一定的数据库空间。在这一点上，更接近于临时表。但不像临时表那样在某个特定的时机删除数据。物化视图中的数据是可重用的，因此，经常应用于读取频繁的场所。

物化视图对于大数据表的处理显得尤为重要。为了统计一个拥有百万级记录的数据表的总和及平均值问题，将耗费大量数据库资源和时间。可以通过物化视图改善这一状况。即对表进行一次统计，并将统计结果存储在物化视图中，以后每次直接查询该视图即可。但是，物化视图并不适合统计更新频繁的数据，因为每次的更新都连带更新物化视图，所付出的代价是相当大的。

7.4.2 物化视图的使用

物化视图使用的场合为大数据场合，本节首先为表 `employees_sales` 添加若干条数据，以更好地物化视图进行演示。

SALE_DETAILS_ID	GOODS_NAME	PRICE	QUANTITY	SALE_MONTH	SALE_BY
1	西装	300	2	3月	1
2	毛衣	20	20	3月	1
3	西裤	100	2	3月	1
4	运动鞋	80	10	3月	2
5	西装	300	1	3月	2
6	皮鞋	100	4	3月	2
7	运动鞋	80	10	3月	3
8	皮鞋	100	15	3月	3
9	衬衫	70	10	3月	3
10	衬衫	50	8	3月	4
11	T恤	60	8	3月	4
12	皮鞋	100	6	3月	4
13	运动衫	40	9	3月	5
14	羊毛衫	80	10	3月	5
15	西裤	60	10	3月	5
16	西装	300	2	4月	1
17	毛衣	20	20	4月	1
18	西裤	100	2	4月	1
19	运动鞋	80	10	4月	2
20	西装	300	1	4月	2
21	皮鞋	100	4	5月	2
22	运动鞋	80	10	5月	3
23	皮鞋	100	15	5月	3
24	衬衫	70	10	5月	3



25	衬衫	50	8	5 月	4
26	T 恤	60	8	5 月	4
27	皮鞋	100	6	6 月	4
28	运动衫	40	9	6 月	5
29	羊毛衫	80	10	6 月	5
30	西裤	60	10	6 月	5

1. 创建物化视图

在表 `employee_sale_details` 中, 有 30 条销售明细记录。现欲统计每月的销售额, 那么可以利用分组查询, 并将查询结果存储在物化视图中。

【范例 7-15】 演示如何创建物化视图, 存储单月销售额。

```
create materialized view mv_monthly_sales
build immediate
refresh on commit
enable query rewrite
as
select sale_month, sum(price*quantity)
from employee_sale_details
group by sale_month
```

【代码说明】 `create materialized view` 用于创建物化视图, 需要注意的是, 物化视图没有 `replace` 选项, 即不能写为 `create or replace materialized view`; `build immediate` 要求 Oracle 立即编译视图; `refresh on commit` 要求 Oracle 数据库, 一旦基础表的修改被提交, 应自动更新视图, 以便保持视图的数据与基础表一致; `enable query rewrite` 表示启用查询重写功能。

通过如下查询语句查看物化视图 `mv_monthly_sales` 所存储的数据。

```
select * from mv_monthly_sales
```

查询结果如下:

SALE_MONTH	SALE_QUANTITY
3 月	8940
4 月	2300
5 月	4280
6 月	2360

2. 查看物化视图信息

物化视图不同于其他视图, 不能通过查询视图 `user_views` 中查看其信息。其信息可以通过 `user_mvviews` 和 `user_objects` 进行查看。

```
select mview_name, query from user_mvviews where mview_name = 'MV_MONTHLY_SALES'
```

查询结果如下:

MVIEW_NAME	QUERY
MV_MONTHLY_SALES	SELECT SALE_MONTH, SUM(PRICE*QUANTITY) SALE_QUANTITY FROM EMPLOYEE_SALE_DETAILS GROUP BY SALE_MONTH

在 `user_objects` 中查询其相关信息:

```
select object_name, object_type, status from user_objects where object_name='MV_MONTHLY_SALES'
```

查询结果如下:

OBJECT_NAME	OBJECT_TYPE	STATUS
-----	-----	-----

```

MV_MONTHLY_SALES      TABLE          VALID
MV_MONTHLY_SALES      MATERIALIZED VIEW  VALID

```

分析查询结果可知，物化视图在创建视图的同时，也创建了一个同名的物理表。而物理表 `mv_monthly_sales` 是真正存储数据的地方。

7.4.3 物化视图的数据加载

在范例 7-15 创建物化视图的 SQL 语句中使用了 `build immediate`。该选项用于立即加载物化视图的数据。也就是说，在创建物化视图的同时，立即根据定义从基础表中获取数据，并将数据添加到物化视图中。另外一个可用选项为 `build deferred`，表示延迟载入数据。

使用延迟加载是必要的。有时，物化视图的基础表数据量巨大，载入数据会耗费大量资源。直接使用立即加载策略，在数据库使用高峰期，会造成客户端的延迟。但是，后续的开发工作可能使用到该物化视图，那么可以采用延迟加载数据的策略。一旦创建视图，即可在后续开发工作中进行引用。当数据库不再繁忙时，进行数据加载。该情形类似于创建一个空表，并不向表中插入数据，但可以成功地在开发中进行引用。

7.4.4 物化视图的数据更新

物化视图的数据应该根据基础表的更新而更新。范例 7-15 创建物化视图的 SQL 语句中使用了 `refresh on commit`，要求 Oracle 实现自动更新的功能。即基础表的数据更新被提交后，应该自动更新物化视图的数据。

【范例 7-16】 演示物化视图的数据更新。

利用以下 SQL 语句向基础表 `employee_sales` 中插入数据：

```

begin
insert into employee_sale_details values (31, '衬衫', 100, 2, '7月', 1);
insert into employee_sale_details values (32, '西装', 300, 5, '8月', 2);
insert into employee_sale_details values (33, '皮鞋', 130, 6, '9月', 3);
insert into employee_sale_details values (34, '运动鞋', 120, 4, '10月', 4);
insert into employee_sale_details values (35, '羊毛衫', 60, 5, '11月', 5);
insert into employee_sale_details values (36, '衬衫', 50, 6, '11月', 3);
commit;
end;

```

此时查询物化视图 `MV_MONTHLY_SALES` 中的数据：

```
select * from mv_monthly_sales
```

查询结果如下：

SALE_MONTH	SALE_QUANTITY
10 月	480
11 月	600
3 月	8940
4 月	2300
5 月	4280
6 月	2360
7 月	200
8 月	1500
9 月	780

分析查询结果可知，物化视图已经实现了自动更新。

物化视图的自动更新与关系视图的自动更新不同。物化视图的自动更新，是将更新后的数



据存储起来；而关系视图则每次都会重新查询基础表。

但是，物化视图的自动更新对于基础表的大数据量频繁更新是不利的。因为频繁的更新会占用大量数据库资源，因此物化视图的使用应该集中在读取频繁、而更新较少的情形下。

7.4.5 查询重写

在范例 7-15 的创建物化视图 SQL 语句中，使用了 `enable query rewrite` 选项。该选项用于启用查询重写。查询重写是指 Oracle 对基础表的查询，按照优化原则，查找恰当的物化视图，如果获得优化视图，则将查询转化为对物化视图的查询。

【范例 7-17】演示如何利用物化视图实现查询重写。

使用如下 SQL 语句查询：

```
select sale_month, sum(quantity * price) sale_quantity from employee_sale_details
group by sale_month
```

在 PL/SQL Developer 中对该查询语句执行【Explain Plan】。查询语句的执行过程如图 7-9 所示。

Description	Object owner	Object name	Cost	Cardinality	Bytes
SELECT STATEMENT, GOAL = ALL_ROWS			2	9	153
MAT_VIEW REWRITE ACCESS FULL	SYSTEM	MV_MONTHLY_SALES	2	9	153

图 7-9 Oracle 使用物化视图实现查询重写

从图中可以看出，虽然在 SQL 语句中查询的是表 `employee_sale_details`，但是 Oracle 使用了查询重写，直接搜索物化视图 `mv_monthly_sales`。

为了区分不使用查询重写时的执行情况，首先删除物化视图 `mv_monthly_sales`：

```
drop materialized view mv_monthly_sales
```

再次使用【Explain Plan】来分析查询语句的执行情况，结果如图 7-10 所示。

Description	Object owner	Object name	Cost	Cardinality	Bytes
SELECT STATEMENT, GOAL = ALL_ROWS			3	4	40
SORT GROUP BY			3	4	40
TABLE ACCESS FULL	SYSTEM	EMPLOYEE_SALE_DETAILS	2	30	300

图 7-10 Oracle 查询基础表

对比图 7-9 与图 7-10 的 COST 值可知，使用查询重写，在速度上有极大的提高。而这是在基础表的数据量不大的情况下做出的分析。当基础表数据量极大时，使用查询重写的优势将会更加明显。



7.5 本章小结

本章讲述了 Oracle 中的四种视图：关系视图、内嵌视图、对象视图和物化视图。其中关系视图是最常用的视图。关系视图在安全性、组合数据及增强性能方面都有着极为广泛的应用。内嵌视图则无须真正创建一个视图对象，而更接近于临时视图，具有运行时有效的特点，使内嵌视图使用起来非常灵活、方便。对象视图则侧重于以面向对象的思想来处理关系数据表，用户在数据库建模时更加形象。物化视图则提前存储了计算结果，并且可以直接复用这些结果，从而缩短数据库的响应时间。视图的主要功能总结如下：

- 增强安全性；
- 组装数据；

- 封装复杂查询;
- 提供建模模型;
- 提高响应速度。



7.6 本章实例

本章主要讲述了视图的创建和使用。在 Oracle 数据库中, 本身存在若干已创建的视图。这些视图可以用来获取许多重要信息。例如, 为了获得数据库中某个表的所有列, 可利用视图 `user_tab_cols`。

【范例 7-18】 演示如何获得某个表的所有列。

```
select table_name, column_name, data_type from user_tab_cols where lower(table_name)
= 'teacher'
```

【代码说明】 `user_tab_cols` 是 Oracle 已定义的视图; `table_name` 列可以用于获得表名; `column_name` 列可以用于获得列名; `data_type` 列则可以用于获得列的数据类型; Oracle 在存储对象名(表名、列名等)时, 会使用大写形式, 因此在此使用了 `lower()` 函数将列值统一为小写形式。

该 SQL 语句的查询结果如下:

TABLE_NAME	COLUMN_NAME	DATA_TYPE
TEACHER	TEACHER_NAME	NVARCHAR2
TEACHER	ID	NUMBER



7.7 习题

1. 视图共分为哪几种?
2. 何时应当使用内嵌视图?
3. 简述对象视图的主要特点。
4. 简述物化视图的主要特点。
5. 如何获得数据库中某个用户的所有对象名称?

第 8 章 函数与存储过程

Oracle 数据库中不仅可以使⤵用单条语句对数据库进行增、删、改、查操作，而且可以多条语句组成一个语句块，并一起执行。这些语句块可以进行显式命名，并被其他应用调用。这些命名的语句块被称为函数与存储过程。本章将重点介绍函数与存储过程的使用。

本章的主要内容包括：

- Oracle 中的自定义函数；
- Oracle 中的存储过程；
- 包装函数与存储过程——程序包。

通过本章的学习，读者可以了解 Oracle 中函数与存储过程的编写，并理解与编程语言中函数和过程的区别。在此基础上，读者可以掌握函数包这一概念，并可以在实际开发中应用程序包来封装函数和存储过程。



8.1 函数

函数是 Oracle 数据库中常用对象之一，与其他编程语言的函数一样，Oracle 中的函数也必须返回一个值。这也是函数区别于存储过程的重要特征。

8.1.1 函数简介

函数可以包含传入参数，函数的语句块进行实际的处理动作，并最终返回值。Oracle 中的函数分为两类：一类为系统函数，即 Oracle 已经定义好的函数；另一类是自定义函数，即用户根据需要自行定义的函数。自定义函数时，应注意以下几个方面。

1. 函数与功能的划分

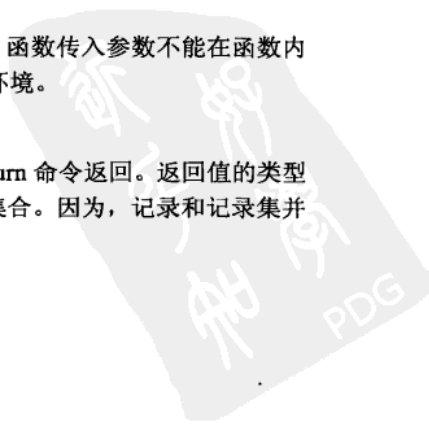
应该首先明确函数的功能。每个独立的功能应该使用单独的函数实现。多个功能在同一个函数中实现，不仅会使代码复杂化，增加维护成本，更重要的是，不易最大化实现复用。因此，首先要将功能划分清晰，针对功能实现函数。

2. 函数的参数

函数的传入参数可以没有，如果有，一定要明确其数据类型。函数传入参数不能在函数内部进行修改，因此不必担心函数内部对参数的修改会影响调用者环境。

3. 函数的返回值

函数必须有返回值，并且返回值必须在函数的结尾处使用 `return` 命令返回。返回值的类型可以是字符串、数值型、数组或者对象类型，但是不能返回记录集合。因为，记录和记录集并不是 Oracle 的数据类型。



8.1.2 创建函数

在了解了函数的基本信息之后，本节将通过两个范例来讲解如何创建及查看自定义函数。

1. 创建函数

与创建表和视图的命令类似，可以利用 `create or replace function` 命令来创建一个函数。

【范例 8-1】 演示如何创建最简单的函数。

```
create or replace function get_hello_msg
return varchar2 as
begin
    return 'hello world';
end get_hello_msg;
```

【代码说明】 `create or replace function` 用于创建或者替换一个函数；`get_hello_msg` 指定函数名称；`return varchar2` 指定函数的返回类型为 Oracle 的可变字符串类型；处于 `begin` 和 `end get_hello_msg` 之间的代码为函数的定义；`return 'hello world'` 返回一个字符串 “hello world”。

需要注意的是，每条语句应该以 “;” 结尾，但是 `begin` 和 `end` 是一个语句块，因此只能在 `end get_hello_msg` 之后使用分号。

2. 在数据字典中查看函数的信息

可以利用如下 SQL 语句查询函数 `get_hello_world` 的基本信息。

```
select object_name, object_type, status from user_objects where lower (object_name)
= 'get_hello_msg'
```

查询结果如下：

OBJECT_NAME	OBJECT_TYPE	STATUS
GET_HELLO_MSG	FUNCTION	VALID

分析查询结果可知，函数在数据库中的对象类型被标识为 “FUNCTION”。需要注意的是，虽然创建时函数名使用了小写形式，但数据字典的存储都是大写形式，即 “`get_hello_msg`” 被存储为 “GET_HELLO_MSG”。所以，应该首先使用 `lower()` 函数将列 `OBJECT_NAME` 的值转换为小写形式，再进行比较。

同样，函数的定义也是存储在数据字典中的。可以通过视图 `user_source` 进行查看，代码如下：

```
select name, type, line, text from user_source where lower(name) = 'get_hello_msg'
```

【代码说明】 `user_source` 用于查询数据库中定义的函数和存储过程的代码。其中，`name` 列标识了函数或存储过程的名称；`type` 列标识类型（函数/存储过程）；`line` 列标识了代码的行号，即每行源码都会存储为一条记录；`user_source` 则为每行源码的实际内容。

查询结果如下：

NAME	TYPE	LINE	TEXT
GET_HELLO_MSG	FUNCTION	1	function get_hello_msg
GET_HELLO_MSG	FUNCTION	2	return varchar2 as
GET_HELLO_MSG	FUNCTION	3	begin
GET_HELLO_MSG	FUNCTION	4	return 'hello world';
GET_HELLO_MSG	FUNCTION	5	end get_hello_msg;

需要注意的是，第 4 行的源码 “`return 'hello world';`” 中的空格是源码中输入的，数据字典按照原有模式进行存储。



3. 查看函数的返回值

函数都有返回值,在 Oracle 中可以直接调用函数。范例 8-2 演示了如何在 PL/SQL Developer 的【Command Window】中查看函数的返回值。

【范例 8-2】 演示如何查看函数的返回值。

(1) 首先开启服务端输出。

```
SQL> set serverout on;
```

(2) 开启了服务端输出之后,在 PL/SQL Developer 中输入以下代码:

```
SQL> declare msg varchar2(20);  
2 begin  
3     msg := get_hello_msg;  
4     dbms_output.put_line(msg);  
5 end;
```

【代码说明】 Oracle 中使用 declare 命令声明一个变量 msg,其类型为可变长度 20 的字符串;msg:=get_hello_msg 用于为变量 msg 赋值,其值为函数 get_hello_msg 的返回值;dbms_output.put_line(msg)用于在控制台上打印出 msg 的内容。

(3) 在【Common Window】中执行该代码块,结果如下:

```
SQL> /;
```

```
hello world
```

```
PL/SQL procedure successfully completed
```

可见 get_hello_msg 返回了字符串“hello world”。

(4) 同样,一旦函数被创建,可以像使用其他函数一样使用该函数,例如,select sysdate from dual 可以返回当前日期。在 PL/SQL Developer 中的【SQL Window】中使用如下语句查看 get_hello_msg 的值:

```
select get_hello_msg msg from dual
```

【代码说明】 直接使用 select 语句来查询函数 get_hello_msg 的返回值,并指定别名 msg。

查询结果如下:

```
MSG  
-----  
hello world
```

8.1.3 函数中的括号

其他标准编程语言中,函数的括号都是必需的,但是范例 8-2 中,函数 get_hello_msg 并没有使用小括号。当函数需要传入参数时,参数列表必须使用小括号括起来,但是当函数没有参数时,小括号可以省略。当函数没有小括号时,在形式上和变量相同,那么有可能会产生变量冲突。

【范例 8-3】 演示函数与变量冲突。

将范例 8-2 中的代码修改为:

```
SQL> declare  
2     msg varchar2(20);  
3     get_hello_msg varchar2(20);  
4 begin  
5     get_hello_msg := 'welcome message';  
6     msg := get_hello_msg;
```



```

7      dbms_output.put_line(msg);
8  end;

```

【代码说明】declare 命令不仅声明了变量 msg，而且声明了一个与函数 get_hello_msg 同名的变量。get_hello_msg:=‘welcome message’ 将变量的值置为 “welcome message”。

再次在【Command Window】中查看其输出，代码如下：

```

SQL> /

welcome message

PL/SQL procedure successfully completed;

```

可见，系统的输出并非函数 get_hello_msg() 的返回值，而是变量 get_hello_msg 的值。Oracle 在遇到变量时，如果该变量可用，则使用该变量；如果变量不可用，才尝试将其解释为函数名。

为了防止产生冲突，尽量不要省略函数的小括号。例如，将范例 8-3 的代码修改如下：

```

declare msg varchar2(20);
begin
    msg := get_hello_msg();
    dbms_output.put_line(msg);
end;

```

即不省略函数 get_hello_msg() 的小括号。

另外，需要注意的是，不要声明一个与函数同名的变量，否则，系统将无法正确引用该函数。范例 8-4 演示了这种情况。

【范例 8-4】变量与函数的同名冲突。

```

SQL> declare
2      msg varchar2(20);
3      get_hello_msg varchar2(20);
4  begin
5      get_hello_msg := 'welcome message';
6      msg := get_hello_msg();
7      dbms_output.put_line(msg);
8  end;

```

【代码说明】该代码声明了名为 get_hello_msg 的变量，同时利用 get_hello_msg() 调用同名函数，PL/SQL Developer 仍然会抛出如下所示的错误：

```
PLS-00222: no function with name 'GET_HELLO_MSG' exists in this scope
```

8.1.4 函数的参数

在 8.1.2 节中创建的函数 get_hello_msg 是无参数函数，本节将演示带参数函数的创建和使用。

【范例 8-5】演示如何计算员工应发工资的函数。

```

SQL> create or replace
2      function get_tax(p_salary number)
3      return number as
4  begin
5      declare tax_salary number;
6      begin
7          tax_salary := p_salary - 2000;
8
9          if tax_salary<=0 then
10             return 0;
11         end if;
12

```



```
13     if tax_salary<=500 then
14         return tax_salary*5/100;
15     end if;
16
17     if tax_salary<=2000 then
18         return tax_salary*10/100 - 25;
19     end if;
20
21     if tax_salary<=5000 then
22         return tax_salary*15/100 - 125;
23     end if;
24
25     if tax_salary<=20000 then
26         return tax_salary*20/100 - 375;
27     end if;
28
29     if tax_salary<=40000 then
30         return tax_salary*25/100 - 1375;
31     end if;
32
33     if tax_salary<=60000 then
34         return tax_salary*30/100 - 3375;
35     end if;
36 end;
37 end get_tax;
38 /
```

【代码说明】在函数 `get_tax` 的定义中，使用了参数 `p_salary` 传入员工的工资，并根据超过起征点的额度计算应缴的所得税。

以某人的工资，扣除社保等项，剩余 6000 元为例，可以利用该函数计算应缴所得税。

```
select get_tax(6000) tax from dual
```

返回的结果如下：

```
TAX
---
475
```

8.1.5 函数的确定性

每次调用函数，Oracle 总是根据传入的参数，执行相同的步骤，并返回最终值。函数的确定性是指，传入的参数一定，无论函数被调用多少次，都会返回相同的值。例如，范例 8-5 所定义的函数，每次输入相同的工资额，返回值都不会改变。

对于具有确定性的函数，在定义时，可以使用 `deterministic` 选项，以告知 Oracle 创建确定性函数。

【范例 8-6】演示如何创建确定性函数。

```
SQL> create or replace
2     function get_tax(p_salary number)
3     return number
4     deterministic as
5     begin
6         declare tax_salary number;
7         .
8         .
9         .
10        .
11    end get_tax;
12
```



【代码说明】deterministic 选项用于指定新建函数为确定函数。注意，该选项应该置于返回值的定义之后，否则，将抛出编译错误。

对于使用频繁的函数，确定性将在很大程度上提高数据库性能。Oracle 在调用函数时，会查找以前是否使用相同的参数调用过该函数。如果查找到，那么将直接使用先前的执行结果，而不会再次执行该函数定义中的操作步骤。例如，在员工数量众多时，很多员工的工资都是相同的，利用 get_tax() 函数的确定性，能够提高计算性能，节省数据库服务器资源。

8.1.6 典型函数举例

行转列问题是一个常见的问题，即将多行数据转换为一列。例如，在学生表中，存储了很多学生资料，现欲获得所有学生的姓名列表，常见做法是将所有学生姓名串联起来，即多行转一列。

【范例 8-7】演示如何获得所有学生姓名。

学生表 students 的数据如下：

STUDENT_ID	STUDENT_NAME	STUDENT_AGE
1	金瑞	14
2	钟君	15
3	王山	14
4	刘迪	14
5	钟会	14
6	张玉	14
7	柳青	14
8	胡东	14
9	商乾	14
10	周明	14

为了获得所有学生姓名，必须要对数据表中的数据循环处理，以获得每个学生的姓名，并将所有学生姓名的字符串串联起来。可以创建一个函数来处理该过程，相应的代码如下：

```
SQL> create or replace
2  function get_student_string
3  return varchar2
4  as
5  begin
6  declare cursor cu_student is
7  select student_name from students
8  order by student_id;
9  student_name varchar2(10);
10 rowString varchar2(500);
11
12 begin
13  open cu_student;
14  fetch cu_student into student_name;
15
16  while cu_student%found loop
17  rowString:=rowString || student_name || ', ';
18  fetch cu_student into student_name;
19  end loop;
20
21  return substr(rowString,1, length(rowString) - 1);
22  end;
23 end get_student_string;
24 /
```



【代码说明】`declare cursor cu_student is select student_name from students order by student_id` 用于声明一个游标，该游标可用于循环获得数据表中所有学生姓名记录；`fetch cu_student into student_name` 则用于将游标所指向的当前记录的数据赋值给 `student_name`；`while cu_student%found loop` 用于循环处理游标所指向的记录；`rowString:=rowString || student_name ||` '，' 用于将变量 `student_name` 的值添加到 `rowString` 的末尾。

通过如下语句查看函数 `get_student_string()` 的返回值：

```
select get_student_string() from dual
```

查询结果如下：

```
GET_STUDENT_STRING()
```

```
-----  
金瑞, 钟君, 王山, 刘迪, 钟会, 张玉, 柳青, 胡东, 商乾, 周明
```



8.2 存储过程

存储过程 (Store Procedure) 对应于其他编程语言中的过程。存储过程不需返回值，但是可以有参数。本节将详细讲述存储过程的创建及使用。

8.2.1 存储过程简介

存储过程不必返回值，但是可以有参数。存储过程的参数分三种：IN（输入）参数、OUT（输出）参数和 IN OUT（输入输出）参数。函数适用于复杂的统计和计算，最终将结果返回；而存储过程则更适合执行对数据库的更新，尤其是大量数据的更新。

使用存储过程有如下好处。

1. 提高数据库执行效率

在编程语言中使用 SQL 接口更新数据库，如果更新复杂而频繁，那么，可能需要频繁连接数据库。众所周知，连接数据库是非常耗时和消耗资源的。如果将所有工作都交由一个存储过程来完成，那么将大大减少数据库连接频率，从而提高数据库执行效率。

2. 提高安全性

存储过程是作为对象存在于数据库中的。可以通过对存储过程分配权限来控制整个操作的安全性。同时，使用存储过程实际上实现了数据库操作从编程语言转移到了数据库中。只要数据库不遭到破坏，这些操作将被一直保留。

3. 可复用

可复用性也是函数的重要属性之一，同样，存储过程的提出，在某种程度上也是由于可复用性的需求。

8.2.2 创建存储过程

存储过程中可以实现数据库的增删改查等操作，也可以实现复杂的运算，但不能够直接执行数据库定义语言，即 DDL 操作。

1. 创建存储过程

可以通过存储过程实现数据库的增删改查等操作。范例 8-8 演示了如何创建一个简单的存储过程。

【范例 8-8】 演示如何将所有学生的年龄修改为 10。

```
SQL> create or replace procedure update_students
2   as
3   begin
4       update students set STUDENT_AGE = 10;
5       commit;
6   end update_students;
7   /
```

【代码说明】 create or replace procedure update_students 用于创建一个名为 update_students 的存储过程；begin 与 end update_students 之间的部分为存储过程的定义；该存储过程用于更新 students 表中所有学生的年龄为 10。

2. 查看存储过程在数据字典中的信息

同样，可以在 user_objects 和 user_source 中查找存储过程 update_procedure 的信息。

在 user_objects 中查找该存储过程的基本信息：

```
select object_name, object_type, status from user_objects where lower (object_name)
= 'update_students'
```

查询结果如下所示：

OBJECT_NAME	OBJECT_TYPE	STATUS
UPDATE_STUDENTS	PROCEDURE	VALID

可见，在数据字典中，存储过程的对象类型被标识为 PROCEDURE。

在 user_source 中查找该存储过程的信息：

```
select * from user_source where lower(name) = 'update_students'
```

查询结果如下所示：

NAME	TYPE	LINE	TEXT
UPDATE_STUDENTS	PROCEDURE	1	procedure update_students
UPDATE_STUDENTS	PROCEDURE	2	as
UPDATE_STUDENTS	PROCEDURE	3	begin
UPDATE_STUDENTS	PROCEDURE	4	update students set STUDENT_AGE = 10;
UPDATE_STUDENTS	PROCEDURE	5	commit;
UPDATE_STUDENTS	PROCEDURE	6	end update_students;

3. 执行存储过程

在 PL/SQL Developer 中，可以利用 execute 命令执行存储过程 update_procedure。

【范例 8-9】 演示如何利用 execute 命令调用函数。

```
SQL> execute update_students;
```

PL/SQL procedure successfully completed

查看表 students 中的数据，以查看存储过程的执行情况：

```
select * from students
```

查询结果如下：

STUDENT_ID	STUDENT_NAME	STUDENT_AGE
------------	--------------	-------------



1	金瑞	10
2	钟君	10
3	王山	10
4	刘迪	10
5	钟会	10
6	张玉	10
7	柳青	10
8	胡东	10
9	商乾	10
10	周明	10

分析查询结果可知，存储过程成功执行。

当然，也可以像调用函数一样调用存储过程，如范例 8-10 所示。

【范例 8-10】 演示如何在代码块中调用存储过程。

```
SQL> begin
2   update_students;
3 end;
4 /
```

PL/SQL procedure successfully completed

【代码说明】 包含在 begin-end 块内的 update_students 语句用于调用存储过程。这实际上也演示了如何在一个存储过程或函数中调用另外一个存储过程或者函数。

存储过程的参数用法比函数要复杂得多。存储过程共有三种参数：IN 参数、OUT 参数和 IN OUT 参数。

8.2.3 存储过程的参数——IN 参数

IN 参数，顾名思义，是指传入参数，即只进不出的参数。它由调用者传递给存储过程之后，存储过程在执行过程中，无论怎样使用该参数，都无法改变该参数的值。该参数对于存储过程来说，是只读的。例如，在更新学生信息的存储过程 update_stdudents 中，可以传入一个年龄参数，用于标识需要将学生年龄修改为多少岁。

【范例 8-11】 演示传入参数的使用。

```
SQL> create or replace
2   procedure update_students(in_age in number) as
3   begin
4       update students set student_age = in_age;
5       commit;
6   end update_students;
7 /
```

此时向存储过程传入年龄为 12 的参数，代码如下：

```
SQL> begin
2   update_students(12);
3 end;
4 /
```

PL/SQL procedure successfully completed

在成功执行存储过程之后，再次查询表 students，以确认是否修改成功。

```
SQL> select * from students;
```

查询结果如下：

STUDENT_ID	STUDENT_NAME	STUDENT_AGE
1	金瑞	12
2	钟君	12
3	王山	12
4	刘迪	12
5	钟会	12
6	张玉	12
7	柳青	12
8	胡东	12
9	商乾	12
10	周明	12

如果在存储过程中尝试修改 IN 参数的值，则会出现编译错误，如范例 8-12 所示。

【范例 8-12】 演示尝试修改 IN 参数的值将会抛出的错误。

```
SQL> create or replace
2  procedure update_students(in_age in number) as
3  begin
4      update students set student_age = in_age;
5      in_age := in_age + 10;
6      commit;
7  end update_students;
8  /
```

在 PL/SQL Developer 中编译该存储过程，将出现如下所示的错误。

```
Error: PLS-00363: expression 'IN_AGE' cannot be used as an assignment target
Line: 5
Text: in_age := in_age + 10;
```

```
Error: PL/SQL: Statement ignored
Line: 5
Text: in_age := in_age + 10;
```

Error: PLS-00363: expression 'IN_AGE' cannot be used as an assignment target 亦即无法为表达式 p_page 赋值。原因是，该参数为一个 IN 参数。

8.2.4 存储过程的参数——OUT 参数

函数可以有返回值，存储过程并没有显式的返回值。但是可以通过 OUT 参数获得存储过程的处理结果。在范例 8-11 中，更新了表 students 中的学生年龄，可以通过 OUT 参数返回更新之后的值，以便验证更新是否成功。

【范例 8-13】 演示如何使用 OUT 参数获得返回值。

首先，存储过程 update_students 的代码修改如下：

```
SQL> create or replace
2  procedure update_students(in_age in number, out_age out number) as
3  begin
4      update students set student_age = in_age;
5      select student_age into out_age from students where student_id = 1;
6      commit;
7  end update_students;
8  /
```

【代码说明】 in_age 参数是一个传入参数，而 out_age 是一个输出参数；代码 select student_age into out_age from students where student_id = 1; 用于将 id 为 1 的学生年龄赋给输出参数 out_age。



此时，可以在 PL/SQL Developer 中声明一个变量，并在调用存储过程时，将变量置于输出参数 out_age 的位置上。那么 out_age 的输出值，将直接赋给该变量。

(1) 开启服务器输出：

```
SQL> set serverout on;
```

(2) 输入如下代码块，并按下 Enter 键执行：

```
SQL> declare updated_age number;
2  begin
3  update_students(20, updated_age);
4  dbms_output.put_line(updated_age);
5  end;
6  /
```

```
20
```

```
PL/SQL procedure successfully completed
```

【代码说明】declare updated_age number;用于声明一个数值型变量；update_students(20, updated_age);则将该变量传入存储过程 update_students; dbms_output.put_line(updated_age);则用于输出变量 updated_age。

(3) 通过打印出的变量 updated_age 的值可知，updated_age 被赋值为 20，证明已经成功从存储过程中输出了变量。

(4) 需要注意的是，一定要为输出参数指定输出的变量名，如范例 8-12 中的 updated_age。不能为输出参数指定常量。如下所示的调用语句，将出现错误：

```
SQL> begin
2  'update_students(20,30);
3  end;
4  /
```

```
begin
  update_students(20,30);
end;
```

```
ORA-06550: line 3, column 22:
PLS-00363: expression '30' cannot be used as an assignment target
ORA-06550: line 3, column 3:
PL/SQL: Statement ignored
```

错误原因是 30 为一个常量，在存储过程中，不能将参数的值输出给一个常量。

(5) 另外，不能够随意缺失变量，代码如下：

```
SQL> begin
2  update_students(20);
3  end;
4  /
```

```
begin
  update_students(20);
end;
```

```
ORA-06550: line 3, column 3:
PLS-00306: wrong number or types of arguments in call to 'UPDATE_STUDENTS'
ORA-06550: line 3, column 3:
PL/SQL: Statement ignored
```

错误原因是在调用存储过程时，只传入了一个参数，但存储过程实际需要两个参数。



8.2.5 存储过程的参数——IN OUT 参数

IN OUT 参数既可以作为输入参数，也可以作为输出参数。因此，IN OUT 参数一般用于对参数的值进行处理，并处理结果输出。一个典型实例就是交换两个变量的值。

【范例 8-14】 演示如何利用 IN OUT 参数交换两个变量的值。

```
SQL> create or replace procedure swap(in_out_param1 in out number, in_out_param2
in out number) as
2  begin
3  declare param number;
4  begin
5  param := in_out_param1;
6  in_out_param1 := in_out_param2;
7  in_out_param2 := param;
8  end;
9  end;
10 /
```

【代码说明】in_out_param1 in out number 和 in_out_param2 in out number 用于为参数提供两个数值型的输入输出参数；存储过程的 begin-end 块中的代码首先声明一个名为 param 的数值型变量，并利用该变量交换两个参数的值。

在 PL/SQL Developer 中检验存储过程 swap 的正确性：

```
SQL> declare param1 number:=25;
2  param2 number:=35;
3  begin
4  swap(param1, param2);
5  dbms_output.put_line('param1 = ' || param1);
6  dbms_output.put_line('param2 = ' || param2);
7  end;
8  /
```

```
param1 = 35
param2 = 25
```

PL/SQL procedure successfully completed

变量 param1 和 param2 在声明时，即被赋值为 25 和 35。经过存储过程 swap 的处理之后，二者的值被修改为 35 和 25。

IN OUT 参数既可以输入也可以输出的特性，的确带来了很大便利性，但是也有其弊端。首先，存储过程可能为多个用户调用，那么针对输出参数的变量，将被频繁且无规律的更新。控制该变量将变得非常困难。另外，因为 IN OUT 参数具有可输出的性质，那么，将不能够使用常量来传入参数，否则，将会出现编译错误。因此，除非必要，应该首先选择单向参数（IN 或者 OUT 参数）。

8.2.6 存储过程的参数——参数顺序

像其他编程语言一样，存储过程的参数顺序同样重要。在以上范例中，所有参数在调用时的值都是按照顺序分配给存储过程。那么顺序就显得格外重要，如果顺序颠倒，不仅得不到正确结果，而且有可能返回不可预知的错误。

【范例 8-15】 演示参数顺序不正确将导致错误。

```
create or replace
procedure update_students(in_age in number, in_name in varchar2) as
begin
```



```
update students set student_age = in_age where student_name = in_name;
commit;
end update_students;
/
```

【代码说明】in_age 和 in_name 分别用于传入参数年龄和姓名；存储过程的作用为将某个学生的年龄进行修改。

在 PL/SQL Developer 中使用如下语句块调用存储过程：

```
SQL> begin
2   update_students('柳青', 19);
3 end;
4 /
```

调用存储过程时，第一个参数为姓名，第二个参数为年龄。因为参数类型的关系，Oracle 将抛出如下所示的错误。

```
begin
  update_students();
end;

ORA-06550: line 3, column 3:
PLS-00306: wrong number or types of arguments in call to 'UPDATE_STUDENTS'
ORA-06550: line 3, column 3:
PL/SQL: Statement ignored
```

该错误表明，对于存储过程 update_students 使用了错误类型的参数值。

对于 IN 类型的参数，如果不希望严格遵循其顺序，那么可以使用另外一种方式进行参数值的传入——名称表示法。

对于范例 8-15 中的存储过程，可以用如范例 8-15 所示的方式进行调用。

【范例 8-16】演示如何利用名称表示法为存储过程传入参数值。

```
SQL> begin
2   update_students(in_name=>'柳青', in_age=>19);
3 end;
4 /
```

PL/SQL procedure successfully completed

【代码说明】in_name=>'柳青' 表示，将参数 in_name 的值赋为“柳青”；in_age 表示，将参数 in_age 的值赋为“19”。此时，虽然参数顺序和存储过程定义的顺序不同，但由于使用了参数名称，因此，仍然可以成功处理。

查询存储过程的处理结果：

```
select * from students where student_name = '柳青';
```

查询结果如下所示：

STUDENT_ID	STUDENT_NAME	STUDENT_AGE
7	柳青	19

对于 IN 参数来说，虽然可以利用名称表示法不受参数顺序的约束。但是，一旦使用了名称表示法，那么其后的参数也必须使用名称表示法。

【范例 8-17】演示名称表示法与位置表示法的混合使用。

```
create or replace
```

```

procedure update_students(in_age in number, in_name in varchar2, out_age out number) as
begin
    update students set student_age = in_age where student_name = in_name;
    select student_age into out_age from students where student_name = '柳青';
    commit;
end update_students;

```

【代码说明】该存储过程共有三个参数：IN 参数 in_age、in_name 和 OUT 参数 out_age。在范例 8-17 所示的存储过程中，尝试在名称表示法之后使用位置表示法，将会抛出错误：

```

SQL> declare age number;
2  begin
3      update_students(in_name=>'柳青', in_age=>19, age);
4      dbms_output.put_line('柳青的年龄是' || age);
5  end;
6  /

declare age number;
begin
    update_students(in_name=>'柳青', in_age=>19, age);
    dbms_output.put_line('柳青的年龄是' || age);
end;

ORA-06550: line 4, column 48:
PLS-00312: a positional parameter association may not follow a named association
ORA-06550: line 4, column 3:
PL/SQL: Statement ignored

```

update_students(in_name=>'柳青', in_age=>19, age);用于调用存储过程 update_students。参数列表中，in_name 和 in_age 使用了名称表示法，其后的 age 使用了位置表示法。执行存储过程时，抛出错误 PLS-00312: a positional parameter association may not follow a named association，即位置表示法的参数不能跟在名称表示法参数之后。

可以对 OUT 参数 out_age 也使用名称表示法，代码块修改如下：

```

SQL> declare age number;
2  begin
3      update_students(in_name=>'柳青', in_age=>19, out_age=>age);
4      dbms_output.put_line('柳青的年龄是' || age);
5  end;
6  /

```

柳青的年龄是 19

PL/SQL procedure successfully completed

out_age=>age 利用了名称表示法将 out_age 这个参数与变量 age 绑定。

但是，可以先使用位置表示法，然后使用名称表示法进行参数传递，代码如下：

```

SQL> declare age number;
2  begin
3      update_students(19, out_age=>age, in_name=>'柳青');
4      dbms_output.put_line('柳青的年龄是' || age);
5  end;
6  /

```

柳青的年龄是 19

PL/SQL procedure successfully completed



`update_students(19, out_age=>age, in_name=>'柳青');`用于调用存储过程 `update_students`, 19 是利用了位置表示法来传递参数, 而之后的 `out_age=>age` 和 `in_name=>'柳青'`则利用了名称表示法。可见, 首先使用位置表示法, 然后使用名称表示法是合法的。

8.2.7 存储过程的参数——参数的默认值

有时, 存储过程的参数有很多个。对于用户来说, 部分参数并非必需的, 那么, 在定义存储过程时应该为可选参数设定默认值, 以允许用户不为该参数传值。需要注意的是, 默认值是仅对 IN 参数而言的, OUT 和 IN OUT 参数没有默认值。范例 8-18 演示了如何使用 IN 参数的默认值。

【范例 8-18】 演示如何使用 IN 参数的默认值。

```
SQL> create or replace
2   procedure insert_student(in_student_id in number, in_student_name in
varchar2,
3   in_student_age in number default 20) as
4   begin
5       insert into students values(in_student_id, in_student_name,
in_student_age);
6       commit;
7   end insert_student;
8   /
```

【代码说明】 `in_student_age in number default 20` 指定了一个名为 `in_student_age` 的参数, 该参数的默认值为 20; 整个存储过程将向数据库中插入一条数据。

省略 `in_student_age` 参数, 尝试调用存储过程 `insert_student`:

```
SQL> begin
2   insert_student(11, '王蒙');
3   end;
4   /
```

PL/SQL procedure successfully completed

`insert_student(11, '王蒙')`表明, 当调用存储过程时, 只输入了 `in_student_id` 和 `in_student_name` 的值, 存储过程仍然能够成功执行。

查询表 `students` 的数据:

```
select * from students
```

查询结果如下:

STUDENT_ID	STUDENT_NAME	STUDENT_AGE
1	金瑞	20
2	钟君	20
3	王山	20
4	刘迪	20
5	钟会	20
6	张玉	20
7	柳青	19
8	胡东	20
9	商乾	20
10	周明	20
11	王蒙	20



分析查询结果可知，虽然没有输入为 `in_student_age` 输入值，但在插入过程中使用了默认值 20。

8.2.8 存储过程的参数——参数顺序总结

在讲解了三种参数的用法之后，可以对参数的顺序总结如下：

具有默认值的参数应该置于参数列表的末尾，因为有时用户需要省略该参数；没有默认值的参数可以遵循“IN 参数”——>“OUT 参数”——>“IN OUT 参数”。

【范例 8-19】 演示存储过程的参数顺序。

```
SQL> create or replace procedure insert_employee(in_employee_id in number,
2          in_employee_name in varchar2,
3          in_employee_sex in varchar2 default '男',
4          out_count out number,
5          in_out_salary number) as
6   begin
7     null;
8 end insert_employee;
9 /
```

【代码说明】 在参数列表中，`in_employee_id`、`in_employee_name`、`out_count` 和 `in_out_salary` 是必需参数；参数 `in_employee_sex` 有默认值“男”，因此是可选参数。

当用户调用该参数时，如果没有输入 `in_employee_sex` 参数，将只能使用名称表示法。因此，应当将 `in_employee_sex` 置于参数列表的末尾，那么，用户将可以直接使用位置表示法。存储过程的定义语法应该修改为如下所示的代码：

```
SQL> create or replace procedure insert_employee(in_employee_id in number,
2          in_employee_name in varchar2,
3          out_count out number,
4          in_out_salary number,
5          in_employee_sex in varchar2 default '男') as
6   begin
7     null;
8 end insert_employee;
9 /
```



8.3 程序包

程序包可以将若干个函数或者存储过程组织起来，作为一个对象进行存储。程序包通常由两部分构成：规范（specification）和主体（body）。程序包也可以包含常量和变量，包中的所有函数和存储过程都能够使用这些变量和常量。

8.3.1 规范

规范是程序包的公共接口。规范中一般定义公用的变量、需要组织到程序包中所有函数和存储过程都会出现在规范中。其出现形式为，仅定义名称和参数列表，但不包括实际处理语句。规范相当于面向对象编程中的接口——只定义方法名称和参数，并不真正实现方法。

1. 创建程序包规范

可以利用 `create or replace package` 命令创建一个程序包规范。

【范例 8-20】 演示如何创建基于表 `students` 的规范。



```

SQL> create or replace package pkg_students as
2   studentString varchar2(500);
3   studentAge number:=18;
4   function get_student_string return varchar2;
5   procedure update_student(in_student_id in number);
6   procedure insert_student(in_student_id in number, in_student_name in
varchar2,
7                           in_student_age in varchar2);
8   procedure delete_student(in_student_id in number);
9 end pkg_students;
10 /

```

【代码说明】studentString varchar2(500)用于定义一个名为 studentString 的可变字符串类型的变量；studentId number:=3 用于定义一个名为 studentId 的数值型变量，并将变量赋值为 3，需要注意的是，这里的变量声明不需要使用 declare 命令；function get_student_string(p_student_id number) return number; 则声明了一个名为 get_student_string 的函数；procedure update_student(in_student_id in number) 等则声明了存储过程。

2. 在数据字典中查看程序包规范的信息

可以通过视图 user_objects 查看其基本信息：

```
select object_name, object_type, status from user_objects where lower(object_name) = 'pkg_students'
```

查询结果如下所示：

OBJECT_NAME	OBJECT_TYPE	STATUS
PKG_STUDENTS	PACKAGE	VALID

分析查询结果可知，程序包规范在数据字典中被标识为“PACKAGE”类型。

可以通过视图 user_source 查看其代码信息：

```
select * from user_source where lower(name) = 'pkg_students'
```

查询结果如下：

NAME	TYPE	LINE	TEXT
PKG_STUDENTS	PACKAGE	1	package pkg_students as
PKG_STUDENTS	PACKAGE	2	studentString varchar2(500);
PKG_STUDENTS	PACKAGE	3	studentAge number:=18;
PKG_STUDENTS	PACKAGE	4	function get_student_string return varchar2;
PKG_STUDENTS	PACKAGE	5	procedure update_student(in_student_id in number);
PKG_STUDENTS	PACKAGE	6	procedure insert_student (in_student_id in number, in_student_name in varchar2,
PKG_STUDENTS	PACKAGE	7	in_student_age in varchar2);
PKG_STUDENTS	PACKAGE	8	procedure delete_student(in_student_id in number);
PKG_STUDENTS	PACKAGE	9	end pkg_students;
PKG_STUDENTS	PACKAGE	10	

8.3.2 主体

主体是真正实现功能的地方。但是主体必须遵从规范，实现规范中定义的函数和存储过程。类似于面向对象编程中的类必须实现接口中的所有方法一样，主体必须实现对应规范的所有函数和存储过程，否则，将出现编译错误。

1. 创建程序包主体

可以通过 `create or replace package body` 来创建一个程序包主体。在主体中再一一实现规范中的所有函数和存储过程。Oracle 系统会自动寻找与主体同名的规范，并查看主体是否符合规范。范例 8-21 演示了如何创建 `pkg_students` 的主体。

【范例 8-21】 演示如何创建程序包主体。

```
create or replace package body pkg_students as

end pkg_students;
```

【代码说明】`create or replace package body` 用于创建一个程序包的主体，该语法比创建规范时多了一个 `body` 关键字。

尝试在 PL/SQL Developer 中编译该创建语句，会出现如下错误：

Compilation errors for PACKAGE BODY SYSTEM.PKG_STUDENTS

Error: PLS-00323: subprogram or cursor 'GET_STUDENT_STRING' is declared in a package specification and must be defined in the package body

Line: 3

Text: end pkg_students;

Error: PLS-00323: subprogram or cursor 'UPDATE_STUDENT' is declared in a package specification and must be defined in the package body

Line: 3

Text: end pkg_students;

Error: PLS-00323: subprogram or cursor 'INSERT_STUDENT' is declared in a package specification and must be defined in the package body

Line: 3

Text: end pkg_students;

Error: PLS-00323: subprogram or cursor 'DELETE_STUDENT' is declared in a package specification and must be defined in the package body

Line: 3

Text: end pkg_students;

错误原因在于，`GET_STUDENT_STRING`、`UPDATE_STUDENT`、`INSERT_STUDENT` 和 `DELETE_STUDENT` 这四个函数/存储过程，已经在规范中声明了，但是并未在程序包中实现。可以通过如下步骤来实现程序包主体的修正。

(1) 补齐函数 `GET_STUDENT_STRING` 的实现，代码如下：

```
SQL> create or replace package body pkg_students as
2
3   function get_student_string
4     return varchar2 is
5   begin
6     declare cursor cu_student is
7       select student_name from students
8       order by student_id;
9     student_name varchar2(10);
10    rowString varchar2(500);
11
```



```
12      begin
13          open cu_student;
14          fetch cu_student into student_name;
15
16          while cu_student%found loop
17              rowString:=rowString || student_name || ', ';
18              fetch cu_student into student_name;
19          end loop;
20
21          return substr(rowString,1, length(rowString) - 1);
22      end;
23  end get_student_string;
24
25 end pkg_students;
26 /
```

【代码说明】在程序包主体中实现函数时，不需要 `create or replace` 命令，因为程序包已经具备了该命令；函数的参数列表和返回值必须和规范中相同。

(2) 重新编译该代码，会发现错误提示出现变化，代码如下：

```
Compilation errors for PACKAGE BODY SYSTEM.PKG_STUDENTS

Error: PLS-00323: subprogram or cursor 'UPDATE_STUDENT' is declared in a package
specification and must be defined in the package body
Line: 5
Text: begin

Error: PLS-00323: subprogram or cursor 'INSERT_STUDENT' is declared in a package
specification and must be defined in the package body
Line: 6
Text: declare cursor cu_student is

Error: PLS-00323: subprogram or cursor 'DELETE_STUDENT' is declared in a package
specification and must be defined in the package body
Line: 8
Text: order by student_id;
```

分析错误提示可知，函数 `GET_STUDENT_STRING` 缺失的提示已不再出现，证明主体中已正确实现了该函数。

(3) 在主体中补全所有存储过程的实现，代码如范例 8-22 所示。

【范例 8-22】演示如何在主体中补全所有存储过程的实现。

```
SQL> create or replace package body pkg_students as
2
3      function get_student_string
4          return varchar2 is
5          begin
6              declare cursor cu_student is
7                  select student_name from students
8                  order by student_id;
9                  student_name varchar2(10);
10                 rowString varchar2(500);
11
12          begin
13              open cu_student;
14              fetch cu_student into student_name;
15
16              while cu_student%found loop
17                  rowString:=rowString || student_name || ', ';
18                  fetch cu_student into student_name;
19              end loop;
```




```

20
21     return substr(rowString,1, length(rowString) - 1);
22     end;
23 end get_student_string;
24
25 procedure update_student(in_student_id number) as
26     begin
27         update students set student_age = studentAge where student_id =
in_student_id;
28         commit;
29     end update_student;
30
31     procedure insert_student(in_student_id in number, in_student_name in
varchar2,
32                             in_student_age in varchar2) as
33     begin
34         insert into students values(in_student_id, in_student_name,
in_student_age);
35         commit;
36     end insert_student;
37
38     procedure delete_student(in_student_id in number) as
39     begin
40         delete from students where student_id = in_student_id;
41         commit;
42     end delete_student;
43
44 end pkg_students;
45 /

```

(4) 此时，重新编译该程序包主体，会返回如下信息：

Compiled successfully

2. 在数据字典中查看该程序包主体的信息

同样可以通过视图 `user_objects` 查看程序包主体的信息：

```

select object_name, object_type, status from user_objects where lower(object_name)
= 'pkg_students'

```

查询结果如下所示：

OBJECT_NAME	OBJECT_TYPE	STATUS
PKG_STUDENTS	PACKAGE	VALID
PKG_STUDENTS	PACKAGE BODY	VALID

分析查询结果可知，在创建了程序包主体之后，有两个名为“PKG_STUDENTS”的对象；程序包主体的对象类型被标识为 PACKAGE BODY。

8.3.3 调用程序包中的函数/存储过程

对于程序包中的函数，可以直接在 `select` 语句进行调用。调用的格式为 `package_name.function_name()`。以调用程序包 `pkg_students` 中的函数 `get_student_string()` 为例，相应的代码如范例 8-23 所示。

【范例 8-23】 演示如何调用程序包中的函数。

```

select pkg_students.get_student_string() from dual

```

【代码说明】 `pkg_students.get_student_string()` 用于获得程序包中函数 `get_student_string()` 的返回值。



查询结果如下所示:

```
PKG_STUDENTS.GET_STUDENT_STRING()
```

金瑞, 钟君, 王山, 刘迪, 钟会, 张玉, 柳青, 胡东, 商乾, 周明, 王蒙

对于程序包中的存储过程, 可以在任何代码块中进行调用。调用格式为 `package_name.procedure_name()`。以调用程序包 `pkg_students` 中的 `delete_student()` 为例, 欲删除 ID 值为 10 的学生信息, 相应的代码如范例 8-24 所示。

【范例 8-24】 演示如何调用程序包中的存储过程。

```
SQL> begin
2   pkg_students.delete_student(10);
3 end;
4 /
```

PL/SQL procedure successfully completed

【代码说明】 `pkg_students.delete_student(10);` 用于执行程序包中的 `delete_student` 过程。

查询此时的表 `students` 的内容, 结果如下所示:

STUDENT_ID	STUDENT_NAME	STUDENT_AGE
1	金瑞	20
2	钟君	20
3	王山	20
4	刘迪	20
5	钟会	20
6	张玉	20
7	柳青	19
8	胡东	20
9	商乾	20
11	王蒙	20

8.3.4 程序包中的变量

程序包中的变量一般声明在规范中, 而且可以被主体中所有函数/存储过程共享。在范例 8-20 所创建的程序包规范中, 定义了变量 `studentAge`, 其值为 18; 而在范例 8-22 所创建的程序包主体的 `update_student` 过程中使用了该变量。可以利用范例 8-25 试验该变量在存储过程中的有效性。

【范例 8-25】 演示如何在存储过程中使用程序包变量。

```
SQL> begin
2   pkg_students.update_student(9);
3 end;
4 /
```

PL/SQL procedure successfully completed

【代码说明】 `pkg_students.update_student(9)` 向存储过程 `update_student()` 传入了参数 9; 该存储过程的主要作用为更新某个学生的年龄, 参数 9 即为学生的 ID。

重新查询 ID 为 9 的学生的年龄:

```
select * from students where student_id = 9
```

查询结果如下所示:

STUDENT_ID	STUDENT_NAME	STUDENT_AGE
------------	--------------	-------------

9 商乾 18

分析查询结果可知, 学生年龄已被更新为 18。可见, 在程序包规范中声明的变量 `studentAge` 可在主体的存储过程中使用。



8.4 本章实例

存储过程的应用非常广泛, 可以用来处理非常复杂的问题。其中比较常用的一种应用为循环处理。有时, 将一条语句可以处理的问题, 使用存储过程来解决, 反而是一种更好的策略。

【范例 8-26】 演示如何利用存储过程实现数据表迁移。

(1) 以视图 `user_objects` 中的数据为例, 假设其数据为原表。

(2) 创建新表 `target` 作为目标数据表。

```
create table target (object_id number, object_name varchar2(30), object_type
varchar2(20), previous_name varchar2(30), status varchar2(30));
```

【代码说明】 `target` 表中包含了 `object_id`、`object_name`、`object_type`、`previous_name` 和 `status` 列。

(3) 现获得视图 `user_objects` 中的对应列, 并将数据插入表 `target` 中。其中 `previous_name` 列是指, 所有记录按照 `object_id` 进行升序排列, 处于当前记录之前的那条记录的 `object_name` 的列值。一种可能的解决思路为: 利用 SQL 语句的批量插入, 代码如下:

```
insert into target
(object_name, object_id, object_type, status, previous_name)
select object_name,
       object_id,
       object_type,
       status,
       (select object_name
        from user_objects
        where object_id = (select max(object_id)
                          from user_objects
                          where object_id < u.object_id))
from user_objects u
```

【代码说明】 该插入语句从视图 `user_objects` 中获得所有记录集合, 并插入到表 `target` 中。值得注意的是, 列 `previous_name` 的获取利用了比较复杂的查询——对于每条记录, 首先获得小于其 `object_id` 的所有 `object_id` 中的最小值, 然后从视图 `user_objects` 中获得最小 `object_id` 所对应的 `object_name`。

在笔者的测试环境下, 550 行的插入动作, 所花费的时间约为 12.7 秒, 如图 8-1 所示。

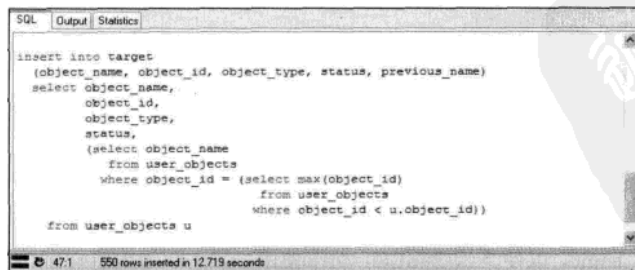


图 8-1 利用批量查询的执行结果



(4) 尝试取消复杂查询，而为 `previous_name` 列赋以固定值 “object”，那么相应的代码应该修改为：

```
insert into target
(object_name, object_id, object_type, status, previous_name)
select object_name,
       object_id,
       object_type,
       status,
       'object'
from user_objects u
```

此时，执行该插入语句，执行结果如图 8-2 所示。

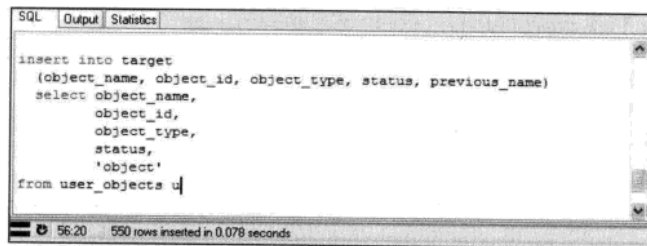


图 8-2 取消复杂查询之后的执行结果

在同样的测试环境下，插入同样 550 条记录，只需 0.078 秒。毫无疑问，复杂查询是影响效率的主要原因。

(5) 尝试创建一个存储过程，利用存储过程解决该问题。

```
create or replace procedure insert_objects as
begin
  declare
    cursor cu_objects is
      select * from user_objects order by object_id;

    obj          user_objects%rowtype;
    previous_name user_objects.object_name%type;

  begin
    open cu_objects;
    fetch cu_objects into obj;
    while cu_objects%found loop
      insert into target
      values
        (obj.object_id,
         obj.object_name,
         obj.object_type,
         previous_name,
         obj.status);

      previous_name := obj.object_name;
      fetch cu_objects
      into obj;
    end loop;
  end;
end insert_objects;
```

【代码说明】在该存储过程中利用游标来循环处理视图 `user_objects` 中的每条记录，并记录当前记录的 `object_name`，以便下次 `INSERT` 动作使用（第 9 章的内容详细讲述了游标的使用）。问题的重点在于使用存储过程的方式，避免了每次进行全表扫描。执行该存储过程，会发

现耗时明显减少，如图 8-3 所示。

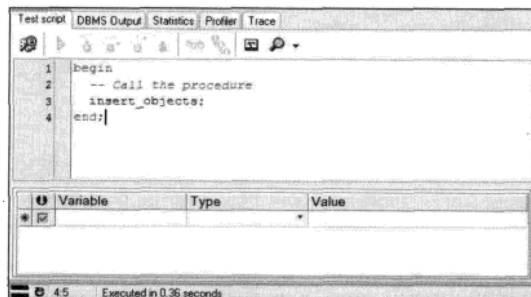


图 8-3 使用存储过程耗时明显减少

保持环境不变，此时的耗时为 0.36 秒。

(6) 通过本例可以看出，虽然使用一条语句处理插入操作看似简单，但实际上并不实用。尤其对于大数据库的迁移，更应该重点考虑使用存储过程，而非复杂的查询来实现。



8.5 本章小结

本章主要讲述了数据库中两个重要的对象，函数与存储过程。二者是 Oracle 与其他编程语言非常相近的地方。函数和存储过程本质是复杂的 SQL 语句的组合，在 Oracle 编程中起了至关重要的作用。用户编写的代码可以存储并重用，这使得 Oracle 编程具有编程语言的基本特点。另外，Oracle 提供了许多安全机制便于控制函数和存储过程的访问权限，也使得函数和存储过程具有了更加灵活的应用。程序包和主体的概念，则体现了 Oracle 编程照样可以具有面向接口编程的特点，使得 Oracle 编程更加规范、可靠。



8.6 习题

1. 什么是函数的确定性？
2. 简述存储过程的三种参数的特点。
3. 程序包由哪两部分构成？二者的关系如何？
4. 函数和存储过程的最大区别是什么？



第9章 游 标

Oracle 编程中，可以使用函数、存储过程等，这在很大程度上接近于编程语言。Oracle 的本质还是作为一个数据库存在的，因此，Oracle 也必须提供方便地访问数据的方法。这正是游标概念的本质——允许用户针对某个结果集进行逐行访问。本章重点介绍游标的概念，主要包括：

- 声明和使用显式游标；
- 使用隐式游标；
- 使用动态游标。

通过本章的学习，读者可以对 Oracle 中各种游标有清晰的了解，并在日常开发中使用它们。



9.1 游标简介

游标类似于编程语言中的指针，游标可以进行位置的移动，以循环访问结果集中每条记录。通过游标可以方便地访问当前记录。

游标分两类：显式游标和隐式游标。显式游标可以被用户显式创建、打开、访问、关闭，即用户可以控制游标的整个生命周期。而隐式游标无须用户的全程控制，即可进行访问。



9.2 显式游标

显式游标在使用时，应该遵循声明、打开、访问、关闭的步骤。本节将通过几个范例讲解如何创建和使用游标。

9.2.1 声明游标

如同声明变量一样，声明游标也应该使用 `declare` 命令。游标的内容一般利用 SQL 查询语句来定义。因为游标的本质就是用来处理结果集中的每条记录。范例 9-1 演示了获取表 `students` 中所有学生姓名的游标声明。

【范例 9-1】 演示查询表 `students` 中的所有学生姓名的游标声明。

```
SQL> declare cursor cu_student_name is
2   select student_name
3   from students;
```

【代码说明】 `declare cursor` 用于声明一个游标；`cu_student_name` 指定了游标的名称；`select student_name from students` 用于选择表 `students` 中的 `student_name` 列的数据，并将游标指向该结果集。

在声明游标的同时，一般还应当声明一个相应的变量。该变量专门用于获取游标中的数据。针对范例 9-1 所示的游标，添加了变量声明的 PL/SQL 语句，如范例 9-2 所示。



【范例 9-2】 演示如何声明与游标对应的局部变量。

```
SQL> declare cursor cu_student_name is
2   select student_name
3   from students;
4
5   student_name varchar2(20);
```

【代码说明】 变量 `student_name` 被定义为长度为 20 的可变字符串。

如果游标中含有多列，常见做法是为每列声明一个变量，以便获取游标时进行临时存取。

【范例 9-3】 演示如何在游标内部声明变量。

```
SQL> declare cursor cu_student_id_name is
2   select student_id, student_name
3   from students;
4
5   student_id number;
6   student_name varchar2(20);
```

【代码说明】 `student_id` 可以用于游标中 `student_id` 的存取；而 `student_name` 则可以用于游标中 `student_name` 的存取。

以上实例的演示实际上处于这样一种环境下——创建者对表 `students` 的列 `student_id` 和 `student_name` 非常熟悉，预先知道其类型分别为 `number` 和 `varchar2`。另外一种可能的情况是创建者并不关心二者的实际数据类型，那么在声明时可以利用范例 9-4 所示的语法：

【范例 9-4】 演示如何直接使用列类型声明变量。

```
SQL> declare cursor cu_student_id_name is
2   select student_id, student_name
3   from students;
4
5   student_id students.student_id%type;
6   student_name students.student_name%type;
```

【代码说明】 `student_id students.student_id%type`，该语法用于声明名为 `student_id` 的变量，该变量的类型与表 `students` 的 `student_id` 列的类型相同；与此类似，`student_name students.student_name%type`；则声明了一个与表 `students` 的 `student_name` 列类型相同的局部变量。

虽然可以用多个变量实现与游标列的一一对应，但是，有的游标所获得的列非常多，使用变量对应列的方式非常烦琐，此时，应该使用行类型的变量来临时存储游标的内容。

【范例 9-5】 演示如何声明行类型变量。

```
SQL> declare cursor cu_student is
2   select *
3   from students;
4
5   student students%rowtype;
```

【代码说明】 `student students%rowtype` 用于声明一个行类型的变量，该变量的实际类型与表 `students` 的行保持一致。

游标声明时，同样可以传递参数。利用带有参数的游标，可以声明灵活的游标，以免频繁地修改游标定义。带有参数的游标，可以在打开时传入参数。例如，现需要声明一个游标，该游标存储了某个年龄段的学生信息。

【范例 9-6】 演示如何声明带有参数的游标。

```
SQL> declare cursor cu_student(minAge in number, maxAge in number) is
```



```
2  select *
3  from students;
4  where student_age >= minAge
5  and student_age <= maxAge
6
7  student students%rowtype;
```

【代码说明】declare cursor cu_student(minAge in number, maxAge in number)用于声明游标 cu_student，该游标含有两个传入参数 minAge 和 maxAge，二者均为数值型；与存储过程的参数类似，in 代表两个参数都为传入参数。

9.2.2 使用游标

游标的使用主要有三个步骤：打开游标、通过游标获取数据和关闭游标。其中尤其需要注意的是关闭游标。当使用完游标之后，应该立即关闭，否则将会占用数据库资源，增加数据库负担。本节将通过几个范例来讲述如何使用游标。

【范例 9-7】演示如何使用变量获取游标信息。

```
SQL> set serverout on;
SQL> declare cursor cu_student_id_name is
2  select student_id, student_name
3  from students;
4
5  student_id students.student_id%type;
6  student_name students.student_name%type;
7  begin
8  open cu_student_id_name;
9  fetch cu_student_id_name into student_id, student_name;
10
11  while cu_student_id_name %found loop
12  dbms_output.put_line(student_id || ':' || student_name);
13  fetch cu_student_id_name into student_id, student_name;
14  end loop;
15
16  close cu_student_id_name;
17 end;
```

【代码说明】open cu_student_id_name 用于打开游标，注意，打开游标并不意味着真正开始使用游标，而只是使游标处于可用状态；fetch cu_student_id_name into student_id, student_name 用于从游标中获取数据，利用 into 关键字，将游标的列值按顺序赋值给变量 student_id 和 student_name；while cu_student_id_name %found loop 用于循环处理，cu_student_id_name %found 返回一个布尔值，即游标的当前位置能否获得记录，每次对游标使用 fetch 动作时，都会更新其 %found 属性；dbms_output.put_line(student_id || ':' || student_name) 用于输出当前变量 student_id 和 student_name 的信息；fetch cu_student_id_name into student_id, student_name 将游标位置向下移动，并获取移动后的记录；end loop 是循环的闭合语句；close cu_student_id_name 用于关闭游标。

执行范例 9-7 的代码：

/

执行结果如下：

```
1:金瑞
2:钟君
3:王山
4:刘迪
```




```
5:钟会
6:张玉
7:柳青
8:胡东
9:商乾
11:王蒙
```

PL/SQL procedure successfully completed

范例 9-7 中的代码使用了变量来获得游标信息，需要注意的是，局部变量必须与游标所获得的列一一对应。而范例 9-8 则演示了如何利用行类型，以避免使用过多局部变量。

【范例 9-8】 演示如何利用行类型获取游标信息。

```
SQL> set serverout on;
SQL> declare cursor cu_student is
2   select *
3   from students;
4   student students%rowtype;
5   begin
6   open cu_student;
7   fetch cu_student into student;
8
9   while cu_student%found loop
10      dbms_output.put_line(student.student_id || ':' || student.student_name);
11      fetch cu_student into student;
12   end loop;
13
14   close cu_student;
15 end;
```

【代码说明】 fetch cu_student into student 用于获取游标，并将所指向的记录赋值给变量 student; dbms_output.put_line(student.student_id || ':' || student.student_name) 用于输出学生的 ID 号和学生姓名，student.student_id 可以直接用来访问当前学生的 ID 号，因为 student 是一个行类型，因此 student.student_id 相当于访问了当前行的 student_id 列值。

执行范例 9-8 所示的代码：

```
SQL> /
```

执行结果如下：

```
1:金瑞
2:钟君
3:王山
4:刘迪
5:钟会
6:张玉
7:柳青
8:胡东
9:商乾
11:王蒙
```

PL/SQL procedure successfully completed

同样，可以向游标中传递参数。例如，需要获得 19~20 岁的学生信息，可以创建一个带有参数的游标。

【范例 9-9】 演示如何使用带有参数的游标。

```
SQL> declare cursor cu_student(minAge in number, maxAge in number) is
```



```
2  select *
3  from students
4  where student_age>=minAge and student_age<=maxAge;
5  student students%rowtype;
6
7  begin
8      open cu_student(19,20);
9      fetch cu_student into student;
10
11     while cu_student%found loop
12         dbms_output.put_line(student.student_name || ':' || student.student_age
|| '岁');
13         fetch cu_student into student;
14     end loop;
15
16     close cu_student;
17 end;
```

【代码说明】open cu_student(19,20)语句相当于在打开游标的同时，向游标传递两个参数，19 和 20；而游标指向由 select * from students where student_age>=19 and student_age<=20 查询出的所有记录。

执行范例 9-9 所示的代码块：

SQL> /

执行结果如下：

金瑞:20 岁
钟君:20 岁
王山:20 岁
刘迪:20 岁
钟会:20 岁
张玉:20 岁
柳青:19 岁
胡东:20 岁
王蒙:20 岁

PL/SQL procedure successfully completed

分析执行结果可知，该游标返回了 19~20 岁的所有学生的记录。

在以上范例中，需要注意的是，每次循环调用时，都应该使用 fetch 命令获得游标的下一条记录。这样，游标位置下移，才能获得新的记录。如果使用了循环，但并未使用 fetch，游标位置总处于首条记录处，那么游标的 found 属性就永远为真，最终造成死循环。



9.3 隐式游标

隐式游标是相对于声明游标变量的显式游标而言的。显式游标通常使用 declare 命令来声明游标；而隐式游标则无须 declare 命令即可直接使用。隐式游标不能直接被用户控制和使用——即不能执行打开（open），获取游标数据（fetch）、关闭（close）等。隐式游标有两种：使用 Oracle 预定义的名 SQL 的隐式游标和使用 cursor for loop 进行循环的隐式游标。

9.3.1 sql 隐式游标

Oracle 为每个 PL/SQL 的会话都定义了一个名为 sql 的游标变量。可以在 PL/SQL Developer 中直接调用该变量。

【范例 9-10】 演示如何在 PL/SQL Developer 中直接调用游标变量 sql。

```
SQL> begin
2   if sql%rowcount>0 then
3       dbms_output.put_line('sql 游标变量的 rowcount 属性大于 0');
4   end if;
5 end;
```

【代码说明】 if sql% rowcount>0 then 用于获得游标变量 sql 的 rowcount 属性，即游标的行数。根据游标变量是否能够抓取记录进行条件判断，并输出相应的提示。

在 PL/SQL Developer 中执行范例 9-10 所示的代码，代码如下：

```
SQL> /
```

```
PL/SQL procedure successfully completed
```

虽然没有任何输出，但是很明显，sql 变量是存在的，并且是一个游标变量。因为它可以被成功调用，并且使用游标变量的属性。

每次执行数据库操作（DML）时，Oracle 都会自动更新该变量。

【范例 9-11】 演示 Oracle 自动更新游标变量。

```
SQL> begin
2   update students
3   set student_age = student_age -1 ;
4
5   dbms_output.put_line('共更新了' || sql%rowcount || '条记录');
6 end;
```

【代码说明】 update students set student_age = student_age-1 用于更新所有学生的年龄，更新后的年龄比现在年龄小一岁。

在 PL/SQL Developer 中执行范例 9-11 的代码，结果如下：

```
SQL> /
```

```
共更新了 10 条记录
```

```
PL/SQL procedure successfully completed
```

此时，查询表 students 的内容，查询结果如下：

```
SQL> select * from students;
```

STUDENT_ID	STUDENT_NAME	STUDENT_AGE
1	金瑞	19
2	钟君	19
3	王山	19
4	刘迪	19
5	钟会	19
6	张玉	19
7	柳青	18
8	胡东	19
9	商乾	17
11	王蒙	19

```
10 rows selected
```

对比以前的数据可知，表 students 的内容更新成功。

SQL 变量是被 Oracle 自动声明的，但是并不能被用户控制。这反映了其隐式变量的特点。

**【范例 9-12】**演示尝试控制 sql 游标变量。

```
SQL> declare
2   student students%rowtype;
3   begin
4   update students set student_age = student_age;
5   fetch sql into student;
6   while sql%found loop
7       dbms_output.put_line(student.student_id || ':' || student.student_name ||
'' || student.student_age);
8       fetch sql into student;
9   end loop;
10  end;
```

【代码说明】declare student students%rowtype 用于声明变量 student，该变量的类型为表 students 的行类型；update students set student_age = student_age 用于更新学生年龄，以更新隐式变量 sql 的内容；fetch sql into student 利用游标变量抓取记录，并存储到变量 student 中；while sql%found loop 用于循环处理游标变量，处理条件为游标可以获得当前记录。

在 PL/SQL Developer 中执行代码 9-12 所示的代码，结果如下：

```
declare
  student students%rowtype;
begin
  update students set student_age = student_age;
  fetch sql into student;
  while sql%found loop
      dbms_output.put_line(student.student_id || ':' || student.student_name || ''
|| student.student_age);
      fetch sql into student;
  end loop;
end;
ORA-06550: line 6, column 9:
PLS-00103: Encountered the symbol "SQL" when expecting one of the following:

    <an identifier> <a double-quoted delimited-identifier>
    <a bind variable>
The symbol "<an identifier>" was inserted before "SQL" to continue.
ORA-06550: line 9, column 11:
PLS-00103: Encountered the symbol "SQL" when expecting one of the following:

    <an identifier> <a double-quoted delimited-identifier>
    <a bind variable>
The symbol "<an identifier>" was inserted before "SQL" to continue.
```

分析错误提示可知，游标变量 sql 并不能使用 fetch 命令进行显式操作。SQL 隐式变量只能用于更新、删除等操作之后的属性信息获取。

9.3.2 cursor for 游标

sql 游标可以应用于更新及删除数据表中的数据，为了能够处理 select 语句获得的记录集合，Oracle 提供了另外一种隐式游标——cursor for 游标。利用该游标，用户可以像使用普通循环语句一样来循环处理 select 语句所获得的每一条记录。

【范例 9-13】演示 cursor for 游标的使用。

```
SQL> begin
2
3   for student in (select * from students) loop
4       dbms_output.put_line(student.student_id || ':' || student.student_name ||
': ' || student.student_age || '岁');
```

```

5   end loop;
6
7   end;

```

【代码说明】for student in (select * from students) loop 用于创建循环，student 为循环中的临时变量，就像其他编程语言 for (i=0; i<count; i++) 中的 i 一样；select * from students 用于获得要处理的结果集；dbms_output.put_line(student.student_id || ':' || student.student_name || ':' || student.student_age || '岁') 用于输出学生的基本信息，student.student_id 用于获得 student 变量的 student_id 属性。

在 PL/SQL Developer 中执行范例 9-13 的代码，结果如下：

SQL> /

```

1:金瑞:19 岁
2:钟君:19 岁
3:王山:19 岁
4:刘迪:19 岁
5:钟会:19 岁
6:张玉:19 岁
7:柳青:18 岁
8:胡东:19 岁
9:商乾:17 岁
11:王蒙:19 岁

```

PL/SQL procedure successfully completed

利用 for 循环可以处理 select 命令获得的结果集；利用 sql 隐式变量可以处理 update、delete 等操作的结果，二者相互配合可以实现与显式游标相同的所有功能。

9.3.3 隐式游标和显式游标

隐式游标和显式游标都可以实现对结果集的操作，但是，相比之下，隐式游标不需要用户控制游标的声明、打开、获取和关闭，因此用户可以利用更少的代码实现同样的功能。而且，隐式游标的执行速度更快，在对游标的显式控制要求不高时，应尽量选择隐式游标。



9.4 游标属性

游标属性反映了游标的当前状态，游标属性对于 PL/SQL 编程有着极为重要的作用。例如逻辑判断等，都可以使用游标属性。本节将讲述 4 个常用的游标属性：found、not found、rowcount 和 isopen。

- found 属性用于标识当前游标从结果中获取记录时，是否成功找到了记录。
- not found 属性是 found 属性的对立面。当不能成功获取记录时，将返回 true，否则返回 false。
- rowcount 属性用于返回当前时刻已经获得了多少条记录。
- isopen 属性用于判断游标是否处于打开状态，如果游标打开，则返回 true，否则，将返回 false。该属性被游标的 open 和 close 动作（对于隐式游标，Oracle 会自动执行该动作）更新。

前三个属性，均被游标的 fetch 动作（对于隐式游标，Oracle 会自动执行该动作）更新。

【范例 9-14】演示如何利用 4 种属性监视游标状态。



```
SQL> declare
2   cursor students
3   is select * from students;
4
5   student students%rowtype;
6
7 begin
8
9   if students%isopen then
10    dbms_output.put_line('声明游标之后, 游标已经打开');
11   else
12    dbms_output.put_line('声明游标之后, 游标尚未打开');
13   end if;
14   open students;
15   if students%isopen then
16    dbms_output.put_line('执行 OPEN 命令之后, 游标已经打开');
17   else
18    dbms_output.put_line('执行 OPEN 命令之后, 游标尚未打开');
19   end if;
20
21   fetch students into student;
22   dbms_output.put_line('第一次执行 FETCH 命令之后, 游标的 ROWCOUNT 属性值为: ' ||
students%rowcount);
23   if students%found then
24    dbms_output.put_line('第一次执行 FETCH 命令之后, 游标的 FOUND 属性值为: TRUE');
25   else
26    dbms_output.put_line('第一次执行 FETCH 命令之后, 游标的 NOTFOUND 属性值为: TRUE');
27   end if;
28
29   dbms_output.put_line('-----');
30
31   loop
32     if students%found then
33       dbms_output.put_line('循环执行..., 游标的 ROWCOUNT 属性值为: ' ||
students%rowcount);
34       fetch students into student;
35     else
36       dbms_output.put_line('游标的 FOUND 属性为: ' || students%found);
37       dbms_output.put_line('循环完毕..., 游标的 ROWCOUNT 属性值为: ' ||
students%rowcount);
38       exit;
39     end if;
40   end loop;
41
42   dbms_output.put_line('-----');
43
44   close students;
45   if students%isopen then
46    dbms_output.put_line('执行 CLOSE 命令之后, 游标仍然打开');
47   else
48    dbms_output.put_line('执行 CLOSE 命令之后, 游标已经关闭');
49   end if;
50
51 end;
```

【代码说明】if students%isopen 用于判断游标 students 是否已经打开, 如果打开将输出“声明游标之后, 游标已经打开”, 否则将输出“游标尚未打开”; loop 用于创建一个循环, if students%found 用于判断游标是否能过获得记录, 若能, 将输出 rowcount 属性, 否则, 将输出“循环完毕”及最终的 rowcount 属性的值。

在 PL/SQL Developer 中执行范例 9-14 所示的代码，执行结果如下：

```
SQL> /
```

```
声明游标之后，游标尚未打开
```

```
执行 OPEN 命令之后，游标已经打开
```

```
第一次执行 FETCH 命令之后，游标的 ROWCOUNT 属性值为：1
```

```
第一次执行 FETCH 命令之后，游标的 FOUND 属性值为：TRUE
```

```
-----
```

```
循环执行...，游标的 ROWCOUNT 属性值为：1
```

```
循环执行...，游标的 ROWCOUNT 属性值为：2
```

```
循环执行...，游标的 ROWCOUNT 属性值为：3
```

```
循环执行...，游标的 ROWCOUNT 属性值为：4
```

```
循环执行...，游标的 ROWCOUNT 属性值为：5
```

```
循环执行...，游标的 ROWCOUNT 属性值为：6
```

```
循环执行...，游标的 ROWCOUNT 属性值为：7
```

```
循环执行...，游标的 ROWCOUNT 属性值为：8
```

```
循环执行...，游标的 ROWCOUNT 属性值为：9
```

```
循环执行...，游标的 ROWCOUNT 属性值为：10
```

```
游标的 NOTFOUND 属性为 TRUE
```

```
循环完毕...，游标的 ROWCOUNT 属性值为：10
```

```
-----
```

```
执行 CLOSE 命令之后，游标已经关闭
```

```
PL/SQL procedure successfully completed
```

分析输出结果可知，仅仅使用了 `declare` 命令声明游标之后，游标并未打开；只有使用了 `open` 命令，游标才真正处于打开状态；每次使用 `fetch` 命令获取游标记录之后，如果成功抓取，则 `rowcount` 属性自动加 1，而 `found` 属性保持为 `true`；当抓取最后一条记录后，再次尝试抓取记录，`found` 属性将被置为 `false`，而 `notfound` 属性将被置为 `true`，而 `rowcount` 属性不再改变；当执行 `close` 命令之后，游标才被真正关闭，`isopen` 属性将被置为 `false`。



9.5 动态游标

无论显式游标还是隐式游标，都具有一个特点，即游标在打开时，其定义已经确定。在整个程序的运行过程中，游标定义不能进行更改。因此显式游标和隐式游标被称为静态游标。为了增加游标的灵活性，Oracle 提供了另外一种游标——动态游标，即其定义在游标声明时没有设定，在打开时，可以进行动态修改。

动态游标又分为两类：强类型动态游标和弱类型动态游标。本节将通过几个范例讲述如何使用这两种动态游标。

9.5.1 强类型动态游标

强类型动态游标是指当游标声明时，虽未设定其查询定义，但是已经指定了游标的返回类型。游标的返回类型可以是 Oracle 内置类型，也可以是自定义类型。

声明一个强类型游标首先自定义一个 `ref cursor` 的游标类型，然后利用该自定义类型，声明一个游标变量。例如，现需一个打印学生信息的存储过程。用户可以向该存储过程传递一个年龄参数，该参数决定了打印哪些学生的信息。如果所传入的年龄参数小于等于 0，则打印所有学生的信息；如果传入的参数大于 0，则打印年龄与参数相同的学生的信息。

【范例 9-15】 演示如何利用强类型动态游标打印学生信息。



```
create or replace procedure printStudents(in_age in number) as
begin
  declare
    type student_type is record(
      id number,
      name varchar2(20),
      age number
    );

    type students_type is ref cursor
      return student_type;

    students students_type;
    student student_type;

  begin
    if in_age <= 0 then
      open students for
        select * from students;
    else
      open students for
        select * from students where student_age = in_age;
    end if;

    fetch students into student;

    while students%found loop
      dbms_output.put_line(student.id || ':' || student.name || ':' || student.age);
      fetch students into student;
    end loop;

    close students;
  end;
end printStudents;
```

【代码说明】declare type student_type is record(id number, name varchar2(20), age number)用于声明自定义类型 student_type，该类型的基类是一个 record 类型，它包含了三个属性 id、name 和 age；代码 type students_type is ref cursor return student_type 用于声明自定义类型，ref cursor 表明该自定义类型是一个动态游标，return student_type 表明该动态游标是强类型动态游标，返回结果集的类型是 student_type，注意，students_type 是游标类型，student_type 是结果集类型，因为 students_type 引用到了 student_type，所以，必须首先声明 student_type，然后声明 students_type，二者顺序不可颠倒；students students_type 用于声明变量 students，该变量是一个动态游标；student student_type 则声明了变量 student，该变量是一个 record 类型；if in_age <= 0 用于判断存储过程的传入参数是否为 0；open students for select * from students 表示当传入的年龄参数为 0 时，将游标打开为获得表 students 的所有记录；else open students for select * from students where student_age = in_age 表示，当传入的年龄参数大于 0 时，按照实际年龄进行查询；fetch students into student；用于获取游标中的记录，并存储到变量 student 中；while students%found loop 用于创建循环，循环执行的条件可以从游标 students 中获得记录；dbms_output.put_line(student.id || ':' || student.name || ':' || student.age)利用变量 student 输出学生信息。

在 PL/SQL Developer 中测试该存储过程（使用小于 0 的参数）。

```
SQL> set serverout on;
SQL> begin
2   printStudents(-1);
```



```

3 end;
4 /

1:金瑞:19
2:钟君:19
3:王山:19
4:刘迪:19
5:钟会:19
6:张玉:19
7:柳青:18
8:胡东:19
9:商乾:17
11:王蒙:19

```

PL/SQL procedure successfully complet

可见，当使用了小于 0 的参数执行存储过程时，将输出所有学生信息。

在 PL/SQL Developer 中测试该存储过程（使用大于 0 的参数）：

```

SQL> begin
2   printStudents(19);
3 end;
4 /

1:金瑞:19
2:钟君:19
3:王山:19
4:刘迪:19
5:钟会:19
6:张玉:19
8:胡东:19
11:王蒙:19

```

PL/SQL procedure successfully completed

可见，当使用了 19 作为参数调用存储过程时，将输出年龄为 19 岁的学生信息。

9.5.2 弱类型动态游标

众所周知，编程语言有强类型和弱类型之分，例如，VB、JavaScript 为弱类型，在 JavaScript 中使用 var 关键字即可声明一个变量，该变量既可以存储字符串也可以存储数字；而 Java、C 等属于强类型语句，变量类型在声明时确定，而且一旦确定将不能改变。弱类型动态游标的概念与此类似，在声明游标时不使用 return 关键字指定游标的返回类型，那么在以后的程序中，可以对其使用不同的返回类型。例如，对于学生表，用户可能只希望获得学生姓名或学生的年龄，那么可以创建一个存储过程，向存储过程传递一个参数，该参数标识用户希望获得的数据。此时，可以利用弱类型游标来实现此功能。

【范例 9-16】 演示如何使用弱类型游标。

```

create or replace procedure printStudentsByFlag(in_flag in varchar2) as
begin
  declare
    type name_type is record(
      id number,
      name varchar2(20)
    );

```



```
type age_type is record(
    id number,
    age number
);

type students_type is ref cursor;

name name_type;
age age_type;
students students_type;

begin
    if upper(in_flag)='NAME' then
        open students for
            select student_id, student_name from students;

        fetch students into name;

        while students%found loop
            dbms_output.put_line(name.id || '号学生的姓名是: ' || name.name);
            fetch students into name;
        end loop;

    elsif upper(in_flag)='AGE' then
        open students for
            select student_id, student_age from students;

        fetch students into age;

        while students%found loop
            dbms_output.put_line(age.id || '号学生的年龄是: ' || age.age);
            fetch students into age;
        end loop;
    end if;

    if students%isopen then
        close students;
    end if;
end;
end printStudentsByFlag;
```

【代码说明】

- declare type name_type is record(id number,name varchar2(20))用于声明自定义类型 name_type, 该类型的基类为 record, 并包含两个属性——id 和 name。
- type age_type is record(id number,age number)用于声明自定义类型 age_type, 该类型包含两个属性——id 和 age。
- type students_type is ref cursor;则声明了自定义类型 students_type, 该类型为弱游标类型。
- name name_type 声明一个 name_type 类型的、名为 name 的变量。
- age age_type;声明一个 age_type 类型的、名为 age 的变量。
- students students_type;声明一个名为 students、类型为 students_type 的游标变量; if upper(in_flag)='NAME'用于判断用户输入的参数是不是“NAME”。
- open students for select student_id, student_name from students 在打开游标时, 将游标的定义设定为获取 student_id 和 student_name 的值; fetch students into name 获取游标 students 的当前记录, 并将记录存储于变量 name 中。
- dbms_output.put_line(name.id || '号学生的姓名是: ' || name.name);用于输出学生学号及

姓名。

- `elsif upper(in_flag)='AGE'`用于判断用户输入的参数是否为“AGE”。
- `open students for select student_id, student_age from students` 在打开游标 `students` 时，将其定义设定为获取学生学号和学生年龄。
- `fetch students into age` 用于获取游标 `students` 的当前记录，并将记录存储于变量 `age` 中。
`dbms_output.put_line(age.id || '号学生的年龄是: ' || age.age)`用于输出学生的学号及年龄信息。
- `if students%isopen` 用于判断游标 `students` 是否处于打开状态，如果打开，则利用 `close students` 关闭游标，因为用户输入的参数不是“NAME”或者“AGE”时，将不会打开游标，关闭一个没有打开的游标，Oracle 将会抛出错误提示，因此，此处的判断是必需的。

在 PL/SQL Developer 中测试该存储过程（输入参数为 NAME）：

```
SQL> begin
2   printStudentsByFlag('NAME');
3 end;
4 /
```

1 号学生的姓名是：金瑞
2 号学生的姓名是：钟君
3 号学生的姓名是：王山
4 号学生的姓名是：刘迪
5 号学生的姓名是：钟会
6 号学生的姓名是：张玉
7 号学生的姓名是：柳青
8 号学生的姓名是：胡东
9 号学生的姓名是：商乾
11 号学生的姓名是：王蒙

PL/SQL procedure successfully completed

在 PL/SQL Developer 中测试该存储过程（输入参数为 AGE）：

```
SQL> begin
2   printStudentsByFlag('AGE');
3 end;
4 /
```

1 号学生的年龄是：19
2 号学生的年龄是：19
3 号学生的年龄是：19
4 号学生的年龄是：19
5 号学生的年龄是：19
6 号学生的年龄是：19
7 号学生的年龄是：18
8 号学生的年龄是：19
9 号学生的年龄是：17
11 号学生的年龄是：19

PL/SQL procedure successfully completed

通过测试结果可知，当使用不同的参数时，可以为同一个游标使用不同的返回值类型，以便实现灵活的功能。



9.5.3 比较两种动态游标

通过以上范例可知，强类型动态游标在使用时，必须声明其类型，在以后的使用过程中，虽然游标的定义可以修改，但是返回值类型是一定的。而弱类型则无须声明返回值类型，但在使用过程中，必须保证每次用于获取记录的类型都能够正确接收来自游标的数据，因此，也存在着一定的风险。应尽量避免使用弱类型游标。



9.6 本章实例

相对于显式游标，隐式游标往往被开发者忽略。而 `cursor for` 游标使用简单、方便，因此，更值得关注。

【范例 9-17】演示 `cursor for` 游标的使用。

```
SQL> set serverout on;
SQL> begin
  2
  3   for student in (select * from students where student_age >= 18 order by student_age
desc) loop
  4     dbms_output.put_line(student.student_name || ':' || student.student_age ||
'岁');
  5   end loop;
  6
  7 end;
  8 /
```

```
金瑞:19 岁
钟君:19 岁
王山:19 岁
刘迪:19 岁
钟会:19 岁
王蒙:19 岁
张玉:19 岁
柳青:18 岁
胡东:18 岁
```

PL/SQL procedure successfully completed

【代码说明】`select * from students where student_age >= 18 order by student_age desc` 用于获得表 `students` 中成年人的信息，并且将这些信息按照年龄进行降序排列。`for student in` 则用于对这些信息进行循环处理。

通过本例可以看出，`cursor for` 游标的作用对象可以是任意复杂的查询语句。



9.7 本章小结

本章讲述了 Oracle 中各种游标的使用。其中显式游标最符合用户的思维习惯——声明游标、打开游标、操作游标、关闭游标。尤其需要注意的是，在使用游标结束之后，一定要执行关闭操作。而隐式游标不允许用户进行生命周期的控制，这些控制都由 Oracle 自行完成。显式游标和隐式游标具有静态性，二者一旦定义，无法进行修改。游标所获得的结果集也是确定不变的。动态游标则提供了游标定义不确定的工作策略，从而使游标具有更强的灵活性。



9.8 习题

1. 请简述显式游标使用的一般步骤。
2. 请简述隐式游标与显式游标的区别。
3. 请列举游标的主要属性。
4. 请简述强类型动态游标与弱类型动态游标的区别。



第 10 章 触 发 器

触发器是数据库常用对象之一。触发器的主要部分是代码块，一旦创建了触发器，在条件成立时，代码块将自动执行。触发器的好处在于，用户只需建立触发器，而无须对其进行任何人为控制，数据库将会精确地完成触发器任务。实际上，用户也不能显式调用触发器。触发器针对不同的动作和对象也分为若干类，本章将详细讲述各种常见触发器的用法，主要包括：

- 语句触发器的创建和使用；
- 行触发器的创建和使用；
- INSTEAD OF 触发器的创建和使用；
- 系统和用户事件触发器的创建和使用。

通过本章学习，读者可以了解各种触发器的创建和使用。尤其对于语句触发器和行触发器的触发规则，以及二者之间的区别有清晰的认识。



10.1 触发器简介

在结构上，触发器非常类似于存储过程，都是为实现特殊的功能而执行的代码块。不过，触发器不允许用户显式传递参数，不能够返回参数值，也不允许用户调用触发器。触发器只能由 Oracle 在合适的时机自动调用。

在功能上，触发器非常类似于面向切面编程中的拦截器。可以在动作执行之前或执行之后，再进行其他自定义操作。

触发器按照触发事件类型和对象不同，可以分为以下几类：语句触发器、行触发器、instead of 触发器、系统事件触发器和用户事件触发器。

其中，语句触发器、行触发器和 instead of 触发器所针对的对象一般是数据表，而系统事件和用户事件触发器更侧重于针对数据库和用户操作触发。换句话说，对数据表的 insert、update、delete 等 DML 操作应该使用前三类触发器；除此以外，数据库的系统事件，例如，数据库启动、关闭等一般使用数据库的系统事件触发器，用户的 drop、alter 等 DDL 操作一般使用用户事件触发器。



10.2 创建和使用触发器

本节将通过一个简单的实例，讲述如何创建和使用触发器。为了方便演示，首先创建名为 t_employees 和 t_salary 的数据表。其结构及数据内容如下。

t_employees 表中的数据如下：

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	5	ACT
2	钟君	5	ACT
3	王山	5	ACT

4	刘迪	4	ACT
5	钟会	3	ACT
6	张玉	3	ACT
7	柳青	3	ACT
8	胡东	3	ACT
9	商乾	3	ACT
10	王蒙	1	ACT

与表 `t_employees` 对应的、用于存储第一季度员工工资的表 `t_salary` 如下：

SALARY_ID	EMPLOYEE_ID	MONTH	SALARY
1	1	1 月	8000
2	2	1 月	7000
3	3	1 月	7000
4	4	1 月	7000
5	5	1 月	6000
6	6	1 月	5500
7	7	1 月	5000
8	8	1 月	4000
9	9	1 月	4000
10	10	1 月	3000
11	1	2 月	8000
12	2	2 月	7000
13	3	2 月	7000
14	4	2 月	7000
15	5	2 月	6000
16	6	2 月	5500
17	7	2 月	5000
18	8	2 月	4000
19	9	2 月	4000
20	10	2 月	3000
21	1	3 月	8000
22	2	3 月	7000
23	3	3 月	7000
24	4	3 月	7000
25	5	3 月	6000
26	6	3 月	5500
27	7	3 月	5000
28	8	3 月	4000
29	9	3 月	4000
30	10	3 月	3000

10.2.1 创建触发器

创建触发器语法如下：

```
CREATE TRIGGER [trigger_name] ON [dbo].[Table]
FOR INSERT, update, delete
AS
Sql_statement
```

以表 `t_employees` 为例，现欲向其中插入一条新的数据，以表示新加入的员工。但是在插入之前，需要首先将其服务年限字段——`work_yeas` 设为 0，以防插入者错误输入。那么，可以创建一个触发器，在插入动作之前执行，强制将 `work_yeas` 置为 0。



【范例 10-1】演示如何利用触发器强制设置新进员工的服务年限为 0。

```
create or replace trigger tr_before_insert_employee
before insert
on t_employees
for each row

begin
:new.work_years := 0;
end;
```

【代码说明】create or replace trigger tr_before_insert_employee 用于创建名为 tr_before_insert_employee 的触发器；before insert 表明该触发器是在 insert 动作之前执行的；on t_employees 表明该触发器所作用的对象为表 t_employees；for each row 表明触发器的作用目标细化为表 t_employees 的每一行，即行级触发器；begin end 块之间的部分为触发器执行时的代码块；:new.work_years := 0 将新插入行的 work_years 字段值更新为 0。

注意，在创建语句中，自 create or replace trigger 起至 end 语句止，是一个完整的创建语法——描述触发器、定义触发器的代码块。在触发器描述和代码块中间不能出现分号。以下代码是一个错误的创建语句：

```
create or replace trigger tr_before_insert_employee
before insert
on t_employees
for each row;

begin
:new.work_years := 0;
end;
```

上述代码在 for each row 的结尾处使用了分号。此时创建触发器将会导致 Oracle 抛出错误，如图 10-1 所示。

可以在 PL/SQL Developer 中测试该触发器的作用。因为触发器不能够被显式调用，所以，可以向表 t_employees 中插入一条新的数据，以触发 insert 事件，执行触发器操作。

```
SQL> insert into t_employees values (11, '周兴', 5, 'ACT');

1 row inserted
```

在成功插入数据后，可以从表中查询新插入数据：

```
SQL> select * from t_employees where employee_id = 11;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
11	周兴	0	ACT

虽然在插入时 work_years 的值为 5，但由于触发器的作用，新加数据的 work_years 被设为 0。

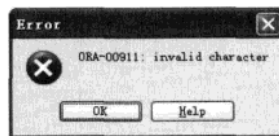


图 10-1 创建触发器出错

10.2.2 触发器的作用级别

在范例 10-1 中，使用了 for each row，将触发器的作用级别定义为行级。也只有此时的行级触发器才能使用:new 变量。尝试将 for each row 删除，并使用:new 变量，Oracle 将抛出错误提示。

【范例 10-2】演示变量:new 只能应用于行级触发器。

```
create or replace trigger tr_before_insert_employee
```



```

before insert
on t_employees

begin
:new.work_years := 0;
end;

```

此时，尝试创建此触发器，Oracle 将抛出如图 10-2 所示的错误提示。

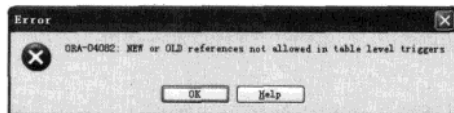


图 10-2 不能在表级触发器使用 new 引用

因此，在创建触发器时，应该特别注意其级别。表级触发器的常见应用场景并不在于限制数据，而是更多地应用于整个表的控制。例如，当向数据表中插入数据时，可以限制当前用户是某个用户才可以进行插入操作。

【范例 10-3】 演示如何利用表级触发器限制插入操作的用户。

```

create or replace trigger tr_before_insert_employee
before insert
on t_employees

begin
if user != 'ADMIN' then
raise_application_error(-20001, '权限不足，不能向数据表中插入数据');
end if;
end;

```

【代码说明】 if user != 'ADMIN' 用于判断当前用户是否为“ADMIN”。如果不是，将抛出错误，并拒绝用户修改。

10.2.3 在多个事件上定义触发器

在范例 10-1 中，为表 t_employees 添加了触发器。该触发器将在 insert 动作之前执行。其实，可以为多个事件同时定义触发器。例如，在表 t_employees 中，含有 status 字段，该字段标识了当前的员工信息是否有效。为了保证该数据表中的 status 字段值都为大写形式，可以对 update 和 insert 两个事件同时定义触发器，将 status 字段的新值转换为大写形式。

【范例 10-4】 演示如何在 UPDATE 和 INSERT 事件上同时定义触发器。

```

create or replace trigger tr_insert_update_employee
before insert or update
on t_employees
for each row

begin
:new.status := upper(:new.status);
end;

```

【代码说明】 before insert or update 指定触发器触发的时机——在插入或更新数据之前；on t_employees 则指定了触发器的作用对象为表 t_employees；for each row 指定了该触发器为行级触发器，每行的插入和更新动作都会触发触发器行为；:new.status := upper(:new.status) 用于将新行的 status 列值修改为大写形式。

尝试向表 t_employees 中插入新的数据，status 列值使用小写形式。



```
SQL> insert into t_employees values(12, '王军', 0, 'act');
```

```
1 row inserted
```

在 PL/SQL Developer 中，向表 `t_employees` 中插入新的数据，其中，列 `status` 的值为小写形式的“act”。查询新插入的数据，结果如下：

```
SQL> select * from t_employees where employee_id = 12;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
12	王军	0	ACT

分析查询结果可知，虽然插入语句中使用了小写形式，但是实际插入的数据为大写形式。尝试将所有员工信息的状态设置为无效，以验证针对 `update` 动作的触发器。

```
SQL> update t_employees set status = 'cxl';
```

```
12 rows updated
```

在更新语句中，状态列 `status` 被赋值为“cxl”，表示“CANCLE”。成功更新数据之后，再次执行查询，结果如下：

```
SQL> select * from t_employees;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	5	CXL
2	钟君	5	CXL
3	王山	5	CXL
4	刘迪	4	CXL
5	钟会	3	CXL
6	张玉	3	CXL
7	柳青	3	CXL
8	胡东	3	CXL
9	商乾	3	CXL
10	王蒙	1	CXL
11	周兴	0	CXL
12	王军	0	CXL

可见，利用 `or` 连接多个事件（例如 `insert or insert`）并应用于触发器之后，Oracle 会同时监视数据表的这些事件，并分别调用触发器进行处理。

10.2.4 为同一事件定义多个触发器

如果为同一对象的同一事件定义了多个触发器，那么，这些触发器都将被触发。触发的顺序按照触发器的创建时间，排在较前位置的将首先触发。

在范例 10-1 中，为表 `t_employees` 每行的 `before insert` 创建了触发器，该触发器用于将所有新进员工的服务年限强制设为 0；在范例 10-4 同样为每行的 `before insert` 创建了触发器，该触发器可以将 `status` 的值转换为大写形式。那么尝试向其中插入如下数据。

```
SQL> insert into t_employees values(13, '周亚', 5, 'act');
```

```
1 row inserted
```

在插入语句中，服务年限列 `work_yeas` 的值为 5，而列 `status` 的值则为小写的“act”。

插入成功后，查询新添加的数据。

```
SQL> select * from t_employees where employee_id = 13;
```



EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
13	周亚	0	ACT

可见，新添加数据的 `work_years` 为 0，`status` 为“ACT”，两个触发器都被触发，并成功执行。

10.2.5 触发器限制

在前面的实例中，所有的触发器都是针对整个表或针对表中所有行的。其实触发器同样可以指定限制条件，以确定触发器是否应该触发。表 `t_employees` 中的现有数据如下：

```
SQL> select * from t_employees;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	5	CXL
2	钟君	5	CXL
3	王山	5	CXL
4	刘迪	4	CXL
5	钟会	3	CXL
6	张玉	3	CXL
7	柳青	3	CXL
8	胡东	3	CXL
9	商乾	3	CXL
10	王蒙	1	CXL
11	周兴	0	CXL
12	王军	0	CXL
13	周亚	0	ACT

但是，目前该表的数据并不符合实际情况。现欲增加一个逻辑判断：在更新表中的数据之前，如果被更新的员工信息中，服务年限字段 `work_yeas` 大于 0，并且 `status` 是“CXL”，应该将其修改为“ACT”，以表明该员工的状态仍为有效。那么，可以为表 `t_employees` 增加一个触发器，在触发器的操作中，将 `status` 的新值修改为“ACT”，但是必须为该触发器增加条件——原状态是“CXL”并且服务年限大于 0。

【范例 10-5】演示触发器限制。

```
create or replace trigger tr_before_update_employee
before update
on t_employees
for each row
when (old.status = 'CXL' and old.work_years > 0)

begin
:new.status := 'ACT';
end;
```

【代码说明】`create or replace trigger tr_before_update_employee` 用于创建一个名为 `tr_before_update_employee` 的触发器；`before update` 表示该触发器的触发时机为更新事件之前；`on t_employees` 表示触发器的作用对象为表 `t_employees`；`for each row` 表示该触发器是一个行级触发器，所有行的更改都会调用触发器；`when (old.status = 'CXL' and old.work_years > 0)` 为触发器的触发提供进一步的限制，`old.status = 'CXL'` 指定条件——原数据的 `status` 列值为“CXL”，`old.work_years > 0` 指定另外一个条件——原数据的 `work_years` 列值大于 0；`:new.status := 'ACT'` 则将列 `status` 的新值修改为“ACT”。



需要注意的是，在触发器描述语句中，原数据使用 `old` 进行引用，例如，`when (old.status = 'CXL' and old.work_years > 0)`；而在触发器的操作语句中，原数据使用 `:old` 进行引用。与之相同，应注意区分 `new` 和 `:new` 的使用场合。

成功创建触发器后，使用如下语句更新表 `t_employees`：

```
SQL> update t_employees set employee_id = employee_id;
```

```
13 rows updated
```

尽管使用了 `update` 语句，但实际上该 SQL 语句并没有试图修改表 `t_employees` 的内容。但是由于触发器的存在，表 `t_employees` 的内容发生了其他变化。查询该表数据，代码如下：

```
SQL> select * from t_employees;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	5	ACT
2	钟君	5	ACT
3	王山	5	ACT
4	刘迪	4	ACT
5	钟会	3	ACT
6	张玉	3	ACT
7	柳青	3	ACT
8	胡东	3	ACT
9	商乾	3	ACT
10	王蒙	1	ACT
11	周兴	0	CXL
12	王军	0	CXL
13	周亚	0	ACT

分析查询结果可知，员工 ID 自 1 至 10 的 `status` 列值已经被修改为“ACT”。这是因为修改前，其状态为“CXL”，并且 `work_years` 值大于 0。



10.3 语句触发器

语句触发器是指建立在表或视图上的、由表的特定操作触发的触发器。这些操作可以是 `insert`、`update` 或者 `delete`。语句触发器是表级触发器，即无论操作影响了多少行数据，语句触发器只被调用一次。

10.3.1 创建语句触发器

在 10.2 节介绍创建简单触发器时，实际上已经介绍了语句触发器，例如范例 10-3 中限制用户的触发器。本节将利用语句触发器实现另一个常见功能——记录表的修改日志。对于表 `t_employees`，如果修改频繁，那么记录下其修改日志以便追踪，是一个常见的功能需求。

【范例 10-6】 演示如何利用触发器记录表的修改日志。

(1) 首先创建一个名为 `employee_log` 的表，其结构如下：

```
SQL> desc employee_log
Name          Type          Nullable  Default  Comments
-----
UPDATE_BY    VARCHAR2(20)  Y
UPDATE_AT    DATE          Y
```

(2) 创建针对表 `t_employees` 的语句触发器：

```

create or replace trigger tr_employee_log
before insert or update
on t_employees

begin
insert into employee_log values(user, sysdate);
end;

```

【代码说明】create or replace trigger tr_employee_log 用于创建名为 tr_employee_log 的触发器；before insert or update 表示该触发器的触发时机为 insert 或 update 动作之前；on t_employees 表示触发器建立在表 t_employees 之上；insert into employee_log values(user, sysdate) 用于向表 employee_log 中插入数据，user 和 sysdate 直接调用了系统变量，相当于 select user from dual 和 select sysdate from dual。

(3) 尝试向表 t_employees 中插入新的数据：

```

SQL> insert into t_employees values (14, '王娟', 0, 'ACT');

1 row inserted

```

(4) 此时，可以在表 t_employees 中获得新插入的数据：

```

SQL> select * from t_employees where employee_id = 14;

EMPLOYEE_ID  EMPLOYEE_NAME  WORK_YEARS  STATUS
-----
14           王娟           0           ACT

```

(5) 查询表 employee_log 中数据：

```

SQL> select update_by, to_char(update_at, 'yyyy-MM-dd') update_at from employee_log;

UPDATE_BY      UPDATE_AT
-----
SYSTEM        2009-06-15

```

可见，在向表 t_employees 中插入一条数据时，由于触发器的作用，同样向表 employee_log 中插入了一条数据。

(6) 现在尝试更新表 t_employees 中的数据，更新规则为：将所有员工的服务年限加 1。

```

SQL> update t_employees set work_years = work_years + 1;

14 rows updated

```

(7) 此时，表 t_employees 中的数据如下：

```

SQL> select * from t_employees;

EMPLOYEE_ID  EMPLOYEE_NAME  WORK_YEARS  STATUS
-----
1            金瑞           6           ACT
2            钟君           6           ACT
3            王山           6           ACT
4            刘迪           5           ACT
5            钟会           4           ACT
6            张玉           4           ACT
7            柳青           4           ACT
8            胡东           4           ACT
9            商乾           4           ACT
10           王蒙           2           ACT
11           周兴           1           CXL
12           王军           1           CXL

```



13	周亚	1	ACT
14	王娟	1	ACT

(8) 查询结果表明, 表 `t_employees` 中的 14 条数据都被更新。此时查询表 `employee_log` 中的数据:

```
SQL> select update_by, to_char(update_at, 'yyyy-MM-dd') update_at from employee_log;
```

UPDATE_BY	UPDATE_AT
SYSTEM	2009-06-15
SYSTEM	2009-06-15

可见, 对于语句触发器, 无论事件影响到的数据是一条还是多条, 该触发器只被调用一次。

10.3.2 触发器谓词

对于激活触发器的动作, Oracle 提供了谓词, 来判断触发动作的类型。常用的谓词包括: `inserting`、`updating` 和 `deleting`。这三种谓词都会返回一个布尔值, 以表示激活动作是否为 `insert` (插入)、`update` (更新) 和 `deleting` (删除)。某个触发器被激活时, Oracle 总是自动更新这三种谓词的值, 需要注意的是, 三种谓词只能有一个为真, 因为每次触发器被激活总是由于某个特定动作。

在范例 10-6 中, 使用了触发器来记录数据表的更新日志。但是该日志过分简单, 现要为日志增加一个名为 `action` 的列, 以表示用户进行了怎样的更改数据库操作。即执行了 `insert`、`update` 还是 `delete`。

【范例 10-7】 演示如何利用谓词记录用户的实际操作。

(1) 修改表 `employee_log` 的结构:

```
SQL> alter table employee_log add (action varchar2(10));
```

Table altered

(2) 修改触发器 `tr_employee_log` 的定义:

```
create or replace trigger tr_employee_log
before insert or update or delete
on t_employees

begin
    if inserting then
        insert into employee_log values(user, sysdate, '插入数据');
    end if;

    if updating then
        insert into employee_log values(user, sysdate, '更新数据');
    end if;

    if deleting then
        insert into employee_log values(user, sysdate, '删除数据');
    end if;
end;
```

【代码说明】 `before insert or update or delete` 用于指定触发器激活的时机——在插入操作、更新操作或删除操作之前; `on t_employees` 指定触发器的作用对象——表 `t_employees`; `if inserting` 用于判断谓词 `inserting` 是否为真, 即激活动作是否为插入操作; `if updating` 用于判断激活动作是否为更新操作; `if deleting` 用于判断激活动作是否为删除操作。

(3) 尝试向表 `t_employees` 中插入新的数据:

```
SQL> insert into t_employees values(15, '刘建', 5, 'ACT');
```

1 row inserted

(4) 确认插入成功:

```
SQL> select * from t_employees where employee_id = 15;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
15	刘建	0	ACT

(5) 查询 `employee_action` 中的数据:

```
SQL> select update_by, to_char(update_at, 'yyyy-MM-dd') update_at, action from employee_log where length(action)>0;
```

UPDATE_BY	UPDATE_AT	ACTION
SYSTEM	2009-06-15	插入数据

利用 `length(action)>0` 来获得包含了触发动作信息的记录, 可见, 向表 `t_employees` 中插入数据时, 相应的 `action` 为“插入数据”。

(6) 尝试更新和删除表 `t_employees` 的记录:

```
SQL> begin
2   update t_employees set status = 'ACT' where status = 'CXL';
3   delete t_employees where work_years = 0;
4 end;
5 /
```

PL/SQL procedure successfully completed

(7) 此时, 检验 `employee_log` 中含有 `action` 信息的记录:

```
SQL> select update_by, to_char(update_at, 'yyyy-MM-dd') update_at, action from employee_log where length(action)>0;
```

UPDATE_BY	UPDATE_AT	ACTION
SYSTEM	2009-06-15	插入数据
SYSTEM	2009-06-15	更新数据
SYSTEM	2009-06-15	删除数据

(8) 综合表 `employee_log` 中的信息可知, 触发器 `employee_log` 实现了记录用户动作的功能。

10.3.3 触发时机

在前面所有的实例中, 触发器的触发时机均在实际操作之前——`before` 执行。另外 Oracle 提供了另外一种触发时机——`after`, 即在实际操作之后执行。利用 `after` 关键字代替前面实例中的 `before` 关键字即可实现实际操作之后激活触发器。

在使用 `before` 和 `after` 之前, 首先应该分析使用场景。例如, 对于验证用户权限的触发器应该使用 `before` 关键字, 因为用户操作执行完毕再进行权限校验是没有任何意义的; 对于记录 `log` 的触发器, 则应该使用 `after` 关键字, 这是因为 `update`、`insert`、`delete` 等操作有可能返回错误, 事务需要回滚, 那么用户没有进行实际操作, 不必记录 `log`, 所以触发时机应该选择 `after`。

通过以上分析, 应该将触发器 `tr_employee_log` 的定义修改为如下所示的代码:



```

create or replace trigger tr_employee_log
after insert or update or delete
on t_employees

begin
  if inserting then
    insert into employee_log values(user, sysdate, '插入数据');
  end if;

  if updating then
    insert into employee_log values(user, sysdate, '更新数据');
  end if;

  if deleting then
    insert into employee_log values(user, sysdate, '删除数据');
  end if;
end;

```

【代码说明】after insert or update or delete 用于执行在 insert、update 或 delete 动作之后激活触发器。

10.3.4 触发器级联

触发器级联是指当某个事件激活了触发器时，该触发器的操作可能涉及另外一个表，而针对涉及表的操作可能激活建立在该表之上的触发器。如果该触发器的操作再次激活最初的触发器，将会造成死循环。因此，触发器的级联往往会造成不可预知的问题。

【范例 10-8】演示如何进行触发器级联。

(1) 为表 t_salary 创建对应的日志表——salary_log，其结构如下：

```

SQL> desc salary_log;
Name          Type          Nullable      Default      Comments
-----
UPDATE_BY     VARCHAR2(20)   Y
UPDATE_AT     DATE           Y
ACTION        VARCHAR2(10)   Y

```

(2) 为表 t_salary 创建触发器：

```

create or replace trigger tr_salary_log
after insert or update or delete
on t_salary

begin
  if inserting then
    insert into salary_log values(user, sysdate, '插入数据');
  end if;

  if updating then
    insert into salary_log values(user, sysdate, '更新数据');
  end if;

  if deleting then
    insert into salary_log values(user, sysdate, '删除数据');
  end if;
end;

```

【代码说明】after insert or update or delete 表明触发器 tr_salary_log 在 insert、update 或 delete 操作之后激活；该触发器的操作为记录表 salary_log 的修改日志。

(3) 此时，修改表 t_employees 的触发器：


```

create or replace trigger tr_employee_log
after insert or update or delete
on t_employees

begin
    if inserting then
        insert into employee_log values(user, sysdate, '插入数据');
    end if;

    if updating then
        insert into employee_log values(user, sysdate, '更新数据');
    end if;

    if deleting then
        insert into employee_log values(user, sysdate, '删除数据');
    end if;

    delete from t_salary where employee_id not in (select employee_id from t_employees
where status = 'ACT');
end;

```

【代码说明】delete from t_salary where employee_id not in (select employee_id from t_employees where status = 'ACT')用于在记录完表 t_employees 的修改日志之后，删除表 t_salary 中的无效记录。即 employee_id 不是表 t_employees 中的有效员工 ID。

(4) 尝试删除表 t_employee 中的数据：

```
SQL> delete from t_employees where status = 'CXL';
```

```
0 rows deleted
```

(5) 虽然没有任何的记录被删除，但是该动作已经执行了，因此在表 employee_log 中会有相应的记录。但更重要的是，表 salary_log 也被添加了记录：

```
SQL> select update_by, to_char(update_at, 'yyyy-MM-dd') update_at, action from salary_log;
```

UPDATE_BY	UPDATE_AT	ACTION
SYSTEM	2009-06-15	删除数据

通过以上分析可知，触发器级联对于用户来说是完全透明的，一旦触发器数量较多，并且关系复杂，跟踪触发器级联的轨迹将是十分困难的。因此，应当尽量避免触发器级联。



10.4 行触发器

与语句触发器不同，行触发器在表的每行上进行操作时，都会激活并执行一次代码。行触发器必须包含 for each row 子句，并且可以引用每一行上的数据。因此，行触发器可以实现的功能更加精细和强大。

10.4.1 行触发器与引用

行触发器可以建立在 insert 操作之上，但此时触发器只能使用 new 引用，而不能使用 old，因为 insert 操作并不存在旧的数据；但对于 update 操作，new 引用和 old 引用都可以使用；delete 操作则只能使用 old 引用。

在表 t_employees 中插入新的数据时，除了要格式化 action 列的值为大写形式、将 work_years 设置为 0 之外，还有另外一个需求——employee_id 的值保持顺序递增。综合以上三



种需求，可以利用行触发器及:new 引用来实现该功能。

【范例 10-9】 演示如何利用行触发器与:new 引用保证插入数据的合法性。

```
create or replace trigger tr_before_insert_employee
before insert
on t_employees
for each row

begin
  select (max(employee_id)+1) into :new.employee_id from t_employees;
  :new.status := upper(:new.status);
  :new.work_years := 0;
end;
```

【代码说明】 for each row 表明该触发器是一个行级触发器；select (max(employee_id)+1) into :new.employee_id from t_employees;用于更新新行的 employee_id 的值为已有 employee_id 最大值的下一个整数；:new.status := upper(:new.status)用于格式化新行的 status 列值为大写形式；:new.work_years := 0 用于设置新行的 work_years 列值为 0。

向表 t_employees 中插入新的数据：

```
SQL> insert into t_employees values(0, '钟君', 19, 'act');
```

```
1 row inserted
```

利用内嵌视图查找新插入的数据：

```
SQL> select * from (select * from t_employees order by employee_id desc) where rownum=1;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
15	钟君	0	ACT

分析新插入的数据可知，存储过程已经实现了预期功能。

对于:new 和:old 引用，可以使用 referencing 子句来指定别名，以便在使用时可以直接使用别名。对于更新表 t_employees 的操作，可能需要更新 status 状态。如果该员工在工资表中出现了，那么 status 状态应该修改为“ACT”，否则应该修改为“CXL”。以下代码演示了如何利用触发器和 reference 子句实现该功能。

```
create or replace trigger tr_before_update_employee
before update
on t_employees
referencing new as new_value
old as old_value
for each row

begin
  declare num number;
  begin
    select count(1) into num from t_salary where employee_id = :old_value.
employee_id;
    if num>0 then
      :new_value.status := 'ACT';
    else
      :new_value.status := 'CXL';
    end if;
  end;
end;
```

【代码说明】 referencing new as new_value old as old_value 用于为引用 new 和 old 分别指定

别名: new_value 和 old_value; declare num number 用于声明一个数值型变量 num; select count(1) into num from t_salary where employee_id = :old_value.employee_id 用于获取表 t_salary 中所有 employee_id 等于表 t_employees 中当前行的 employee_id 的记录总数; if num>0 用于判断自表 t_salary 中获取的记录数是否大于 0, 若大于 0, 则证明该员工信息为有效记录, 利用: new_value.status 将新行数据修改为 “ACT”, 否则修改为 “CXL”。

注意, 在本例中, 使用了 new_value 和 old_value 来代替引用 new 和 old; 而且这两个别名的指定语句 referencing new as new_value old as old_value 必须置于 for each row 之前。

尝试更新表 t_employees 中的数据, 例如, 将 STATUS 列值修改为 “NEW”。

```
SQL> update t_employees set status = 'NEW';
```

15 rows updated

查询表 t_employees 中的数据:

```
SQL> SELECT * FROM T_EMPLOYEES ORDER BY EMPLOYEE_ID;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	6	ACT
2	钟君	6	ACT
3	王山	6	ACT
4	刘迪	5	ACT
5	钟会	4	ACT
6	张玉	4	ACT
7	柳青	4	ACT
8	胡东	4	ACT
9	商乾	4	ACT
10	王蒙	2	ACT
11	周兴	1	CXL
12	王军	1	CXL
13	周亚	1	CXL
14	王娟	1	CXL
15	钟君	0	CXL

10.4.2 触发时机与引用

因为行触发器可以使用 new 和 old 引用, 因此可以利用该触发器记录历史数据。范例 10-10 演示了实现方法。

【范例 10-10】 演示如何利用行触发器记录日志。

(1) 创建表 t_employees_history, 其数据结构如下:

```
SQL> DESC T_EMPLOYEES_HISTORY;
```

Name	Type	Nullable	Default	Comments
EMPLOYEE_HISTORY_ID	NUMBER			
EMPLOYEE_ID	NUMBER			
EMPLOYEE_NAME	VARCHAR2(10)	Y		
WORK_YEARS	NUMBER	Y		
STATUS	VARCHAR2(3)	Y		
ACTION	VARCHAR2(3)	Y		
UPDATE_BY	VARCHAR2(20)	Y		
UPDATE_AT	DATE	Y		

(2) 当更新表 t_employees 后, 利用如下触发器向表 t_employees_history 插入数据。

```
create or replace trigger tr_employee_history
```



```

after update or delete
on t_employees
for each row

begin
  declare historyId number;
begin
  select (nvl(max(employee_history_id), 0) + 1) into historyId from
t_employees_history;
  if updating then
    insert into t_employees_history (employee_history_id, employee_id,
employee_name, work_years, status, action, update_by, update_at)
values(historyId, :old.employee_id, :old.employee_name, :old.work_years, :old.status,
'修改数据', user, sysdate);
    end if;

    if deleting then
      insert into t_employees_history (employee_history_id, employee_id,
employee_name, work_years, status, action, update_by, update_at)
values(historyId, :old.employee_id, :old.employee_name, :old.work_years, :old.status,
'删除数据', user, sysdate);
    end if;
  end;
end;

```

【代码说明】create or replace trigger tr_employee_history 用于创建触发器 tr_employee_history; after update or delete 表明该触发器的触发时机为插入或更新操作之后; on t_employees 表明该触发器建立在表 t_employees 之上; select (nvl(max(employee_history_id), 0) + 1) into historyId from t_employees_history 用于获得 employee_history 表中最大 ID 的下一整数值; if updating 用于判断激活触发器的操作是否为 update, 如果是, 则向表 t_employees_history 中插入原数据, 并存储此时的动作、修改者和修改时间; delete 操作的执行动作与 update 类似。

(3) 尝试更新表 t_employees 中的数据:

```
SQL> update t_employees set status = 'ACT' where status = 'CXL';
```

5 rows updated

(4) 在更新了 5 条数据之后, 查看表 t_employees 中的数据:

```
SQL> select employee_history_id, employee_id, employee_name, work_years, status,
action from t_employees_history;
```

EMPLOYEE_HISTORY_ID	EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS	ACTION
1	11	周兴	1	CXL	修改数据
2	12	王军	1	CXL	修改数据
3	13	周亚	1	CXL	修改数据
4	15	钟君	0	CXL	修改数据
5	14	王娟	1	CXL	修改数据

可见, 在更新表 t_employees 中的 5 条数据之后, 历史表 t_employees_history 中也被插入了 5 条数据。这 5 条数据均为修改前的数据。

(4) 尝试删除表 t_employees 中的数据:

```
SQL> delete from t_employees where employee_id>10;
```

5 rows deleted

(5) 在删除成功后，查询此时历史表中的记录：

```
SQL> select employee_history_id, employee_id, employee_name, work_years, status,
action from t_employees_history;
```

EMPLOYEE_HISTORY_ID	EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS	ACTION
1	11	周兴	1	CXL	修改数据
2	12	王军	1	CXL	修改数据
3	13	周亚	1	CXL	修改数据
4	15	钟君	0	CXL	修改数据
5	14	王娟	1	CXL	修改数据
6	11	周兴	1	CXL	删除数据
7	12	王军	1	CXL	删除数据
8	13	周亚	1	CXL	删除数据
9	15	钟君	0	CXL	删除数据
10	14	王娟	1	CXL	删除数据

分析查询结果可知，触发器自动记录了被删除的记录。

(6) 按照正常逻辑，当数据表被更新之后（即 `after update`），那么更新前的数据应该丢失；在删除表的记录之后（即 `after delete`），被删除的数据也应该丢失。但在范例 10-10 中，成功使用了行触发器来实现历史表功能。这证明，无论是 `before` 还是 `after`，都能使用 `old` 引用和 `new` 引用来获得每行数据更新前和更新后的状态。

10.4.3 触发时机与瞬态

瞬态是针对表来说的，当表被修改时，其状态是瞬态的。正是因为其不确定性，Oracle 不允许访问一个瞬态表。在 Oracle 的用户手册中描述了触发时机与瞬态的关系：当使用行级触发器时，如果使用了 `before` 关键字，那么使用单条插入的方式可以访问触发器的关联表，如果使用批量插入的方式，将不能访问触发器的关联表，因为此时的触发器关联表是瞬态的。

【范例 10-11】 演示单条插入和批量插入对触发器的影响。

```
create or replace trigger tr_before_insert_employee
before insert
on t_employees
for each row

begin
  declare num number;
  begin
    select count(1) into num from t_employees;
  end;
end;
```

【代码说明】 `create or replace trigger tr_before_insert_employee` 用于创建触发器 `tr_before_insert_employee`；`before insert` 表明该触发器的触发时机为 `insert` 动作之前；`for each row` 表明该触发器是一个行级触发器；`select count(1) into num from t_employees` 用于统计表 `t_employees` 中的记录数，并将记录数存储到变量 `num`。该语句的真实用意在于实现访问表 `t_employees`，即触发器的关联表。

(1) 尝试向表 `t_employees` 中插入单条数据。

```
SQL> insert into t_employees values(11, '王武', 0, 'ACT');

1 row inserted
```



可见，插入成功，触发器也成功执行。

(2) 尝试向表 `t_employees` 中插入批量数据。这里所谓的批量插入是指，插入利用 `select` 子句获得记录集合。即使该集合仅有一条数据，Oracle 也将其看做批量插入。

```
SQL> insert into t_employees (select 12 employee_id, '王武', 0, 'ACT' from dual);

insert into t_employees (select 12 employee_id, '王武', 0, 'ACT' from dual)

ORA-04091: table SYSTEM.T_EMPLOYEES is mutating, trigger/function may not see it
ORA-06512: at "SYSTEM.TR_BEFORE_INSERT_EMPLOYEE", line 4
ORA-04088: error during execution of trigger 'SYSTEM.TR_BEFORE_INSERT_EMPLOYEE'
```

(select 12 employee_id, '王武', 0, 'ACT' from dual)用于获得记录集，该记录集仅含有一条数据。尝试向表中插入该记录集时，Oracle 将抛出错误。table SYSTEM.T_EMPLOYEES is mutating, trigger/function may not see it 说明表 `t_employees` 是一个瞬态表，该表不能被触发器或函数访问。

(3) 可以这样理解错误的产生原因：在触发器中访问一个表时，要求该表状态是确定的。对于 `before insert` 是存在这种机会的，因为在插入数据之前，表的状态是确定的，所以使用单条插入的方式是可行的。但是，对于批量插入的方式，由于触发器会被每行数据触发一次，每次都可能改变数据表的实际数据，很难保证针对每次 `before insert` 数据表的状态是确定的，因此 Oracle 不允许批量插入的方式用于 `before insert` 触发器。

(4) 对于 `after insert` 触发器，既不允许单条插入，也不允许批量插入。可以创建一个如下代码所示的触发器：

```
create or replace trigger tr_after_insert_employee
after insert
on t_employees
for each row

begin
  declare num number;
  begin
    select count(1) into num from t_employees;
  end;
end;
```

【代码说明】`create or replace trigger tr_after_insert_employee` 用于创建触发器 `tr_after_insert_employee`；`after insert` 表明该触发器的触发时机为 `insert` 动作之后；`for each row` 表明该触发器是一个行级触发器；`select count(1) into num from t_employees` 用于统计表 `t_employees` 中的记录数，并将记录数存储到变量 `num`，该语句的真实用意在于实现访问表 `t_employees`，即触发器的关联表。

(5) 尝试使用单条插入的方式向表 `t_employees` 中插入数据。

```
SQL> insert into t_employees values(11, '王武', 0, 'ACT');

insert into t_employees values(11, '王武', 0, 'ACT')

ORA-04091: table SYSTEM.T_EMPLOYEES is mutating, trigger/function may not see it
ORA-06512: at "SYSTEM.TR_AFTER_INSERT_EMPLOYEE", line 4
ORA-04088: error during execution of trigger 'SYSTEM.TR_AFTER_INSERT_EMPLOYEE'
```

(6) 可见，使用单条插入的方式，仍然不能够成功执行触发器操作。

(7) 尝试使用批量插入方式向表 `t_employees` 中插入数据。

```
SQL> insert into t_employees (select 12 employee_id, '王武', 0, 'ACT' from dual);

insert into t_employees (select 12 employee_id, '王武', 0, 'ACT' from dual)
```

```
ORA-04091: table SYSTEM.T_EMPLOYEES is mutating, trigger/function may not see it
ORA-06512: at "SYSTEM.TR_BEFORE_INSERT_EMPLOYEE", line 4
ORA-04088: error during execution of trigger 'SYSTEM.TR_BEFORE_INSERT_ EMPLOYEE'
```

(8) 同样，使用批量插入方式，仍然不能在触发器中访问关联表。

(9) 对于单条插入方式来说，在 `after insert` 的触发时机，仍然不能够访问关联表，这是因为该表已经被修改，但修改尚未提交，仍然处于瞬态。



10.5 instead of 触发器

语句触发器和行触发器与触发动作（如 `insert`、`update`、`delete`）之间是一种附属关系。触发器依赖于触发动作，但触发动作本身所执行的操作仍然被执行。二者的效果是叠加起来作用于数据表的。而 `instead of` 触发器则用于代替触发动作，例如，`insert` 动作的触发器不再进行 `insert` 操作，而是转而执行触发器动作。

10.5.1 创建和使用 `instead of` 触发器

`instead of` 触发器的提出是有必要的。因为对于有些对象，并不能进行直接的 `insert` 和 `update` 等动作，但有时又需要按照一定的逻辑来执行这些操作。此时，便可以使用 `instead of` 触发器。在本章的开始，创建了表 `t_employees` 和表 `t_salary` 来存储员工第一季度的工资标准。针对二者有一个如下所示的视图，来存储员工的合计工资。

```
create or replace
view vw_employee_salary as

select e.employee_id, e.employee_name, sum(s.salary) as total
from t_employees e, t_salary s
where e.employee_id = s.employee_id
group by e.employee_id, e.employee_name
```

查看视图 `vw_employee_salary` 的内容：

```
SQL> select * from vw_employee_salary;
```

EMPLOYEE_ID	EMPLOYEE_NAME	TOTAL
1	金瑞	24000
2	钟君	21000
3	王山	21000
4	刘迪	21000
5	钟会	18000
6	张玉	16500
7	柳青	15000
8	胡东	12000
9	商乾	12000
10	王蒙	9000

希望通过视图 `vw_employee_salary` 中的列 `total` 来修改员工的实际工资。工资的差额按月平均分摊到表 `t_salary` 中的 `salary` 列。此时，对视图 `vw_employee_salary` 直接进行 `update` 操作是不可行的，因为视图中没有任何一列对应于表 `t_salary` 中的 `salary` 列。此时可以利用 `instead of` 触发器来代替视图的 `update` 动作。

【范例 10-12】 演示如何利用 `instead of` 触发器更新视图。



```

create or replace trigger tr_update_employee_salary
instead of update
on vw_employee_salary

begin
  declare totalMonth number;
  begin
    select count(distinct(month)) into totalMonth
    from t_salary
    where employee_id = :old.employee_id;

    update t_salary
    set salary = salary + (:new.total - :old.total)/totalMonth
    where employee_id = :old.employee_id;
  end;
end;

```

【代码说明】create or replace trigger tr_update_employee_salary 用于创建名为 tr_update_employee_salary 的触发器；instead of update 表明该触发器是一个 instead of 触发器，并用来代替 update 动作；on vw_employee_salary 表明该触发器的作用对象为视图 vw_employee_salary；declare totalMonth number 用于声明一个名为 totalMonth 的变量，该变量的类型是数值型；select count(distinct(month)) into totalMonth from t_salary where employee_id = :old.employee_id 用于获得每位员工发放工资的月数；update t_salary set salary = salary + (:new.total - :old.total)/totalMonth where employee_id = :old.employee_id 用于更新表 t_salary，将工资差额的平均值更新到每个月份的 salary 列。

(1) 查看 employee_id 为 1 的员工在视图 vw_employee_salary 和表 t_salary 中工资状况。

```
SQL> select * from vw_employee_salary where employee_id=1;
```

EMPLOYEE_ID	EMPLOYEE_NAME	TOTAL
1	金瑞	24000

```
SQL> select * from t_salary where employee_id=1;
```

SALARY_ID	EMPLOYEE_ID	MONTH	SALARY
1	1	1 月	8000
11	1	2 月	8000
21	1	3 月	8000

(2) 尝试利用视图将 employee_id 为 1 的员工的工资总额修改为 30000。

```
SQL> update vw_employee_salary set total=30000 where employee_id=1;
```

```
1 row updated
```

(3) 在更新成功之后，再次查看 employee_id 为 1 的员工在视图 vw_employee_salary 和表 t_salary 中的工资状况。

```
SQL> select * from vw_employee_salary where employee_id=1;
```

EMPLOYEE_ID	EMPLOYEE_NAME	TOTAL
1	金瑞	30000

```
SQL> select * from t_salary where employee_id=1;
```

SALARY_ID	EMPLOYEE_ID	MONTH	SALARY
1	1	1 月	10000
11	1	2 月	10000
21	1	3 月	10000



1	1	1月	10000
11	1	2月	10000
21	1	3月	10000

可见, 触发器 `tr_update_employee_salary` 已经成功实现了预期功能。

(4) 需要注意的是, 触发器对表的修改, 仍然被当做对视图的修改看待。如果对视图 `vw_employee_salary` 的修改没有提交, 那么对表 `t_salary` 的修改也可以执行回滚操作。

```
SQL> rollback;
```

```
Rollback complete
```

(5) 此时, 查看 `employee_id` 为 1 的员工在视图 `vw_employee_salary` 和表 `t_salary` 中工资状况。

```
SQL> select * from vw_employee_salary where employee_id=1;
```

EMPLOYEE_ID	EMPLOYEE_NAME	TOTAL
1	金瑞	24000

```
SQL> select * from t_salary where employee_id=1;
```

SALARY_ID	EMPLOYEE_ID	MONTH	SALARY
1	1	1月	8000
11	1	2月	8000
21	1	3月	8000

分析查询结果可知, 随着 `update vw_employee_salary set total=30000 where employee_id=1` 的回滚操作, 触发器的操作也一并回滚。

(6) 由于触发器 `tr_employee_salary` 的存在, 将无法通过视图 `vw_employee_salary` 修改表的其他列, 如 `employee_name`。欲对列 `employee_name` 进行处理, 需要将该列添加到触发器的处理语句中。

【范例 10-13】 演示如何在触发器中更新多列。

```
create or replace trigger tr_update_employee_salary
instead of update
on vw_employee_salary

begin
  declare totalMonth number;
  begin
    update t_employees
    set employee_name = :new.employee_name
    where employee_id = :old.employee_id;

    select count(distinct(month)) into totalMonth
    from t_salary
    where employee_id = :old.employee_id;

    update t_salary
    set salary = salary + (:new.total - :old.total)/totalMonth
    where employee_id = :old.employee_id;
  end;
end;
```

【代码说明】 `update t_employees set employee_name = :new.employee_name where employee_id = :old.employee_id` 用于更新表 `t_employees` 中的 `employee_name` 列。

10.5.2 instead of 触发器与引用

instead of 触发器可以看做行级触发器，而不必使用 for each row 进行限制。另外，instead of 触发器虽然可以访问 new 引用，但是不能修改引用的值。对于范例 10-13 所示的触发器，尝试在其中修改:new 引用的值。

```
create or replace trigger tr_update_employee_salary
instead of update
on vw_employee_salary

begin
  declare totalMonth number;
  begin

    :new.total := :new.total + 300;

    update t_employees
    set employee_name = :new.employee_name
    where employee_id = :old.employee_id;

    select count(distinct(month)) into totalMonth
    from t_salary
    where employee_id = :old.employee_id;

    update t_salary
    set salary = salary + (:new.total - :old.total)/totalMonth
    where employee_id = :old.employee_id;
  end;
end;
```

【代码说明】:new.total := :new.total + 300 用于尝试在 instead of 触发器中修改 new 引用的值。在 PL/SQL Developer 中编译此触发器，Oracle 将抛出如图 10-3 所示的错误。

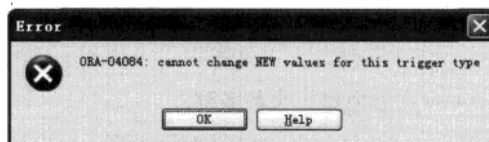


图 10-3 不能在 instead of 触发器中修改 new 引用的值



10.6 系统事件与用户事件触发器

系统事件是指 Oracle 数据库本身的动作所触发的事件。这些事件主要包括：数据库启动、数据库关闭、系统错误等。用户事件是相对于用户的所执行的表（视图）等 DML 操作而言的。常见的用户事件包括：create 事件、truncate 事件、drop 事件、alter 事件、commit 事件和 rollback 事件。系统事件与用户事件触发器并非常用触发器，因此本节只对二者进行简单介绍。

10.6.1 系统事件触发器

系统事件触发器的应用对象是数据库，而创建方法与创建其他触发器类似。以创建数据库启动触发器为例，代码如下：

```
create or replace trigger tr_database_startup
after startup
on database
```



```
begin
    null;
end;
```

【代码说明】**after startup** 表明该触发器的触发时机为启动之后；**on database** 表明该触发器的作用对象为数据库。在这里，只需指定 **database** 即可，无须指定数据库名称，因为该触发器存在于数据库中，这里的 **database** 即为当前数据库。

同样，可以利用 **before** 关键字为数据库的关闭动作创建触发器，代码如下：

```
create or replace trigger tr_database_shutdown
before shutdown
on database

begin
    null;
end;
```

【代码说明】**before shutdown** 表明该触发器的触发时机为关闭之前；**on database** 表明该触发器的作用对象为当前数据库。

对于数据库事件，只能为 **startup** 事件创建 **after** 类型的触发器。数据库的 **startup** 事件没有 **before** 类型的触发器。这是因为触发器也是数据库的对象之一，在数据库启动之前，触发器是不能工作和捕捉事件的。如下代码尝试为数据库的 **startup** 事件创建 **before** 类型的触发器，Oracle 将会抛出错误。

```
create or replace trigger tr_database_startup
before startup
on database

begin
    null;
end;
```

Oracle 将抛出如图 10-4 所示的错误。

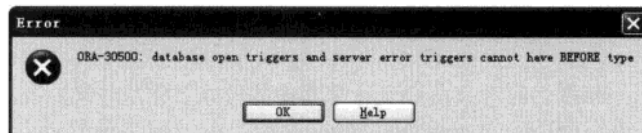


图 10-4 Oracle 不能为数据库的 startup 事件创建 before 类型的触发器

同样，不能够为 **shutdown** 事件创建 **after** 类型的触发器。因为当数据库关闭之后，所有数据库对象，包括触发器都将无法使用。以下代码尝试为数据库的 **shutdown** 事件创建 **after** 类型的触发器。

```
create or replace trigger tr_database_startup
after SHUTDOWN
on DATABASE

begin
    null;
end;
```

在 PL/SQL Developer 中编译该代码，将抛出如图 10-5 所示的错误。

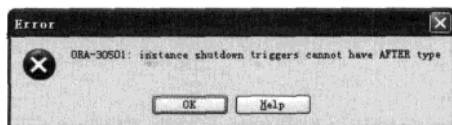


图 10-5 Oracle 不能为数据库的 shutdown 事件创建 after 类型的触发器

10.6.2 用户事件触发器

用户事件触发器的作用对象一般是 `user.schema`，即将触发器建立在该用户及用户所拥有的所有对象之上。例如，为了监视用户 `system` 利用 `truncate` 命令删除表中数据的操作，可以首先建立一个日志表和一个用户事件触发器。范例 10-14 演示了实现的具体细节。

【范例 10-14】 演示如何记录用户 `truncate` 操作的日志。

(1) 日志表 `truncate_log` 的结构如下：

```
SQL> desc truncate_log;
Name                Type                Nullable  Default  Comments
-----
OBJECT_NAME         VARCHAR2(20)        Y
OBJECT_TYPE         VARCHAR2(20)        Y
TRUNCATE_DATE       DATE                Y
```

(2) 创建一个针对用户 `truncate` 操作的触发器：

```
create or replace trigger tr_system_truncate
after truncate
on system.schema

begin
insert into truncate_log values (ora_dict_obj_name, ora_dict_obj_type, sysdate);
end;
```

【代码说明】 `create or replace trigger tr_system_truncate` 用于创建名为 `tr_system_truncate` 的触发器；`after truncate` 表明触发器的触发时机为 `truncate` 操作之后；`on system.schema` 指定该触发器的作用对象为用户 `system` 的所有对象；`insert into truncate_log values (ora_dict_obj_name, ora_dict_obj_type, sysdate)` 用于向日志表中插入日志信息，`ora_dict_obj_name` 和 `ora_dict_obj_type` 分别获取当前对象在 Oracle 字典中的对象名称和对象类别，直接使用它们如同使用了 `select ora_dict_obj_name from dual` 和 `select ora_dict_obj_type from dual`。

(3) 在 PL/SQL Developer 中创建一个名为 `tmp` 的临时表，该表的数据和结构复制自表 `t_employees`。

```
SQL> create table tmp as select * from employees where 1=2;
```

Table created

```
SQL> select * from tmp;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	5	ACT
2	钟君	5	ACT
3	王山	5	ACT
4	刘迪	4	ACT
5	钟会	3	ACT
6	张玉	3	ACT
7	柳青	3	ACT

8	胡东	3	ACT
9	商乾	3	ACT
10	王蒙	1	ACT

(4) 对 tmp 表进行 truncate 操作。

```
SQL> truncate table tmp;
```

Table truncated

(5) 查询表 tmp 的内容。

```
SQL> select * from tmp;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
-----	-----	-----	-----

(6) 可见, 针对表 tmp 的 truncate 操作已经成功完成。此时查询日志表 truncate_log 的数据, 如下所示:

```
SQL> select * from truncate_log;
```

OBJECT_NAME	OBJECT_TYPE	TRUNCATE_DATE
-----	-----	-----
TMP	TABLE	2009-6-18 上午

(7) 分析表 truncate_log 的查询结果可知, 触发器 tr_system_truncate 已经成功记录了用户 system 对其对象的 truncate 操作。

(8) 值得注意的是, 即使是对日志表 truncate_log 进行 truncate 操作, 触发器仍然被执行, 并将操作记录到 truncate_log 中。

```
SQL> truncate table truncate_log;
```

Table truncated

(9) 此时日志表 truncate_log 的内容如下:

```
SQL> select * from truncate_log;
```

OBJECT_NAME	OBJECT_TYPE	TRUNCATE_DATE
-----	-----	-----
TRUNCATE_LOG	TABLE	2009-6-18 上午

(10) 用户 system 首先对表 truncate_log 执行 truncate 命令, 将表中所有数据删除。接着触发器 tr_system_truncate 激活并执行, 将针对表 truncate_log 的 truncate 动作记录到日志表 truncate_log 中, 因此该表中仍然含有一条数据。



10.7 启用和禁用触发器

如同使用 alter 命令来修改表和约束的属性一样, 可以使用 alter 命令来修改触发器的属性。而最常用的操作为启用和禁用触发器。

10.7.1 启用和禁用触发器的场景

启用触发器和禁用触发器是必要的。触发器一旦创建, 默认有效, 那么每个相应的条件都会激活触发器。但需要注意的是, 触发器的执行也是会耗费大量系统资源的。尤其是针对大数据表的行级触发器。以建立在某个表上的 before insert 类型的行级触发器为例, 触发器所执行的操作有可能超过了 insert 操作本身所耗费的资源。当执行大数据量的插入时, 这些数据又被



认为无须经过触发器操作，例如数据库迁移。此时即需用到禁用触发器，当数据库插入完毕，再次启用触发器即可。

10.7.2 禁用触发器

禁用触发器应该使用 `alter trigger trigger_name disable` 命令。

【范例 10-15】 演示如何禁用触发器。

(1) 创建新表 `week_day`:

```
create table week_day(week_day varchar2(10));
```

(2) 创建针对该表的触发器 `tr_insert_week_day`。该触发器的作用为将每次插入的数据转换为大写形式。

```
create trigger tr_insert_week_day
before insert
on week_day
for each row

begin
:new.week_day := upper(:new.week_day);
end;
```

(3) 向表 `week_day` 中插入小写形式的数据，以检验触发器的有效性。

```
SQL> insert into week_day values ('monday');
```

```
1 row inserted
```

(4) 查询表 `week_day` 中的数据。

```
SQL> select * from week_day;
```

```
WEEK_DAY
-----
MONDAY
```

(5) 可见，触发器的确达到了预期的效果。此时，禁用触发器。

```
SQL> alter trigger tr_insert_week_day disable;
```

```
Trigger altered
```

(6) 再次向表 `week_day` 中插入小写形式的数据。

```
SQL> insert into week_day values ('tuesday');
```

```
1 row inserted
```

(7) 此时，查询表 `week_day` 中的数据。

```
SQL> select * from week_day;
```

```
WEEK_DAY
-----
MONDAY
tuesday
```

(8) 在禁用触发器之后向表 `week_day` 中插入的小写形式的数据不会被转换为大写形式，证明触发器失效。

10.7.3 启用触发器

在触发器失效之后，可以利用 `alter trigger trigger_name enable` 命令来启用触发器。以范例



10-15 中的失效触发器为例，启用该触发器的代码如下所示：

```
SQL> alter trigger tr_insert_week_day enable;
```

```
Trigger altered
```

启用触发器之后，向表 `week_day` 中插入小写形式的数据库：

```
SQL> insert into week_day values('wednesday');
```

```
1 row inserted
```

查询此时表中数据：

```
SQL> select * from week_day;
```

```
WEEK_DAY
-----
MONDAY
tuesday
WEDNESDAY
```

可见，启用触发器之后，触发器状态重新有效。

10.7.4 触发器信息

为了获得当前触发器的信息（如触发器是否有效），可以利用视图 `user_objects` 和 `user_triggers`。

【范例 10-16】 演示如何利用视图 `user_objects` 查询触发器信息。

```
SQL> select object_name, object_type, status from user_objects where lower(object_name) = 'tr_insert_week_day';
```

OBJECT_NAME	OBJECT_TYPE	STATUS
TR_INSERT_WEEK_DAY	TRIGGER	VALID

【代码说明】 `object_type` 列标识了对象类型，触发器的对象类型被标识为“TRIGGER”；`STATUS` 为 `VALID`，表明该触发器处于有效状态。

利用视图 `user_triggers` 可以获得更加详细的触发器信息。

【范例 10-17】 演示如何利用视图 `user_triggers` 来获得触发器的详细信息。

```
SQL> select trigger_name, trigger_type, triggering_event, table_name, status from user_triggers where lower(trigger_name) = 'tr_insert_week_day';
```

TRIGGER_NAME	TRIGGER_TYPE	TRIGGERING_EVENT	TABLE_NAME	STATUS
TR_INSERT_WEEK_DAY	BEFORE EACH ROW	INSERT	WEEK_DAY	ENABLED

【代码说明】 列 `trigger_name` 标识了触发器的名称；`trigger_type` 列标识了触发器的类型——行级、BEFORE 类型的触发器；`trigger_event` 列标识了触发器的触发事件——INSERT 动作；`table_name` 列标识了触发器的建立在哪个表上——表 `WEEK_DAY`；`status` 列标识了触发器的状态——ENABLED（可用）。



10.8 本章实例

触发器的应用往往不仅限于两个对象之间产生关联动作。例如，在收入表 `income` 中存储



了各项收入的详细记录；在表 `payout` 中存储了各项支出的详细记录。二者中的任何一个数据发生了变化都会影响到表 `balance` 中的余额。因此，表 `income` 和表 `payout` 应该各有一个触发器，而且两个触发器的实际操作都指向表 `balance`。

【范例 10-18】 演示利用触发器实现收支的余额计算。

(1) 表 `income`、表 `payout` 和表 `balance` 的结构如下：

```
SQL> describe income;
Name      Type      Nullable  Default  Comments
-----
ID         NUMBER    Y
INCOME     NUMBER    Y
```

```
SQL> describe payout;
Name      Type      Nullable  Default  Comments
-----
ID         NUMBER    Y
PAYOUT     NUMBER    Y
```

```
SQL> describe balance;
Name      Type      Nullable  Default  Comments
-----
BALANCE   NUMBER    Y
```

(2) 为表 `income` 创建触发器：

```
create or replace trigger tr_income
after insert or update or delete on income
for each row
begin
    if inserting then
        update balance set balance = balance + :new.income;
    elsif updating then
        update balance set balance = balance - :old.income + :new.income;
    elsif deleting then
        update balance set balance = balance - :old.income;
    end if;
end;
```

【代码说明】 触发器 `tr_income` 包含了对三种 DML 操作——INSERT、UPDATE 和 DELETE 的处理。针对 INSERT 操作，则直接为 `balance` 表中的 `balance` 列值添加收入；针对 UPDATE 操作，先减去原收入，再加上现收入；针对 DELETE 操作，直接减去收入。

(3) 测试触发器的作用：

```
SQL> select * from balance;

BALANCE
-----
0

SQL> insert into income values (1, 20);

1 row inserted

SQL> select * from balance;

BALANCE
-----
20
```



```
SQL> update income set income = 30 where id = 1;
```

```
1 row updated
```

```
SQL> select * from balance;
```

```
BALANCE
-----
30
```

```
SQL> delete from income where id = 1;
```

```
1 row deleted
```

```
SQL> select * from balance;
```

```
BALANCE
-----
0
```

通过对表 `income` 执行 `INSERT`、`UPDATE` 和 `DELETE` 操作，可以看出触发器已经成功发挥作用。

(4) 同样，为表 `payout` 创建触发器。该触发器的动作与表 `income` 的触发器的动作反向。

```
create or replace trigger tr_payout
after insert or update or delete on payout
for each row

begin
  if inserting then
    update balance set balance = balance - :new.payout;
  elsif updating then
    update balance set balance = balance + :old.income - :new.payout;
  elsif deleting then
    update balance set balance = balance + :old.payout;
  end if;
end;
```



10.9 本章小结

本章通过翔实的范例讲述了如何创建及使用触发器。着重讲述了表级触发器和行级触发器的使用。这两种触发器也是日常开发中最常用的触发器。对于行触发器需要注意的是，由于每行数据的操作都会触发触发器操作，因此最有可能影响数据库的工作效率。`instead of` 触发器直接代替了数据库对象的原有动作，因此，一定要注意触发器操作的全面性，不仅仅完成特殊需求，还应注意原操作中的必需操作。



10.10 习题

1. Oracle 中的触发器共有哪几类？
2. 语句触发器和行触发器的主要区别是什么？
3. 简述 `instead of` 触发器的主要特点。
4. 列举两个触发器的主要应用场景。

第 11 章 序 列

在数据库中，ID 往往作为数据表的主键。ID 的创建规则又往往使用自增的整数。在 SQL Server 和 MySQL 中提供了自增的字段类型，但是 Oracle 中并未提供该用法。在范例 10-9 中，为了获得下一条记录的 EMPLOYEE_ID，不得不首先获得数据表中已有 ID 的最大值，并为之增加 1。本章将讲述 Oracle 提供的另外一种更加灵活方便的策略——建立序列对象。本章的主要内容包括：

- 创建序列；
- 使用序列；
- 修改序列属性。

通过本章的学习，读者可以明确序列的概念，并掌握在开发中如何使用序列。



11.1 创建和使用序列

序列（SEQUENCE）像其他数据库对象（表、约束、视图、触发器等）一样，是实实在在的数据库对象。一旦创建，即可存在于数据库中，并可在适用场合进行调用。序列总是从指定整数开始，并按照特定步长进行累加，以获得新的整数。本节着重讲解序列的创建和使用。

11.1.1 创建序列

创建序列，应该使用 `create sequence` 命令。范例 11-1 演示了如何创建一个用于生成表 employee 主键 ID 的序列。

【范例 11-1】 演示如何创建序列。

```
create sequence employee_seq
```

【代码说明】 `create sequence` 命令用于创建序列，序列名称为 `employee_seq`。习惯上，使用 `SEQ` 后缀来标识序列。

因为创建的序列是一个实实在在的数据库对象，因此，可以利用视图 `user_objects` 和 `user_sequences` 进行查看。

```
SQL> select object_name, object_type, status from user_objects where  
lower(object_name) = 'employee_seq';
```

OBJECT_NAME	OBJECT_TYPE	STATUS
EMPLOYEE_SEQ	SEQUENCE	VALID

```
SQL> select sequence_name, min_value, max_value, increment_by from  
user_sequences where lower(sequence_name) = 'employee_seq';
```

SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY
EMPLOYEE_SEQ	1	1000000000	1

EMPLOYEE_SEQ	1	1E27	1
--------------	---	------	---

在视图 `user_sequences` 的查询结果中，会发现 `min_value`、`max_value` 和 `increment_by` 列，其值均为创建序列时的默认值。`min_value` 指定序列的最小值；`max_value` 指定最大值，这里使用了科学计数法的 `1E27`，即十进制的 10^{27} ；`increment_by` 指定序列每次增长的步长。

11.1.2 使用序列

对于序列，有两个重要的属性——`currval` 和 `nextval`。其中 `currval` 用于获得序列的当前值，而 `nextval` 则用于获得序列的下一个值。每次调用 `nextval`，都会使序列的当前值增加单位步长（默认步长为 1）。

序列的调用方法为 `seq.currval` 和 `seq.nextval`。但是，需要注意的是，在序列创建之后，应该首先使用 `seq.nextval`，然后才能够使用 `seq.currval`。

【范例 11-2】 演示首次使用 `seq.currval` 时，Oracle 抛出的错误。

```
Select employee_seq.currval from dual;
```

【代码说明】 `select employee_seq.currval from dual` 用于获得序列 `employee_seq` 的当前值。Oracle 将抛出如图 11-1 所示的错误。

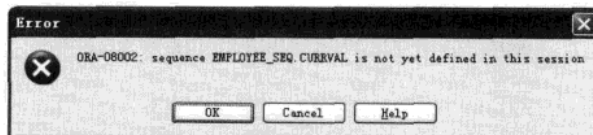


图 11-1 首次使用序列的 `currval` 属性

可以首先使用 `seq.nextval`，再尝试使用 `currval` 属性。

```
SQL> select employee_seq.nextval from dual;
```

```
NEXTVAL
-----
1
```

【代码说明】 `select employee_seq.nextval from dual` 用于获得序列 `employee_seq` 的下一个值。此时可以再次使用序列 `employee_seq` 的 `currval` 属性：

```
SQL> select employee_seq.currval from dual;
```

```
CURRVAL
-----
1
```

可见，此时 `employee_seq` 的 `currval` 属性已经可以使用，其值为 1。

因为序列的 `nextval` 属性的自动增长性，可以尝试利用该属性生成表 `T_EMPLOYEE` 的主键 ID。其步骤如下所示：

(1) 首先保证表 `t_employee` 的 `employee_id` 列为主键，并禁用影响插入操作的触发器。

```
SQL> alter table t_employees modify (employee_id number primary key);
```

```
Table altered
```

```
SQL> alter trigger tr_after_insert_employee disable;
```



Trigger altered

(2) 向表 `t_employees` 中插入新的员工信息:

```
SQL> insert into t_employees values (employee_seq.nextval, '陆逊', 0, 'ACT');
```

```
insert into t_employees values (employee_seq.nextval, '陆逊', 0, 'ACT')
```

ORA-00001: unique constraint (SYSTEM.SYS_C005326) violated

(3) 利用 `employee_seq` 的 `currval` 属性查看在插入语句中的实际值。

```
SQL> select employee_seq.currval from dual;
```

CURRVAL

2

(4) 查看此时表 `t_employee` 中的数据:

```
SQL> select * from t_employees;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	6	ACT
2	钟君	6	ACT
3	王山	6	ACT
4	刘迪	5	ACT
5	钟会	4	ACT
6	张玉	4	ACT
7	柳青	4	ACT
8	胡东	4	ACT
9	商乾	4	ACT
10	王蒙	2	ACT
11	王武	0	ACT

在表 `t_employee` 已经存在了 `employee_id` 为 2 的记录, 因此插入失败。此时的解决方法有两种: 一是利用适当次数的 `select employee_seq.nextval from dual` 语句将 `employee_seq` 的 `nextval` 属性修改至 12; 二是删除现有序列 `employee_seq`, 并重建该序列, 重建时指定初始值。

对于第一种方法, 当数据量较少时, 可以很快执行完毕。但是当已有数据量很大时, 试图通过此方法将变得非常不方便。

11.1.3 序列初始值 start with

为了解决 `employee_seq` 的 `nextval` 属性不适合表 `t_employee` 的问题。可以首先删除序列, 然后重建该序列, 并在重建时指定初始值。

【范例 11-3】 演示如何删除序列, 并利用 `start with` 选项重建。

```
SQL> drop sequence employee_seq;
```

Sequence dropped

```
SQL> create sequence employee_seq
2 start with 12
3 /
```

Sequence created

【代码说明】drop sequence employee_seq 用于删除序列 employee_seq; create sequence employee_seq 用于创建序列 employee_seq; start with 12 指定该序列的值自 12 开始。需要注意的是，第一次使用 employee_seq.nextval 获得的值是 start with 的设定值。

此时，再次向表中插入新员工信息：

```
SQL> insert into t_employees values (employee_seq.nextval, '陆逊', 0, 'ACT');
```

1 row inserted

查询 t_employees 中的数据，以验证插入是否成功：

```
SQL> select * from t_employees;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	6	ACT
2	钟君	6	ACT
3	王山	6	ACT
4	刘迪	5	ACT
5	钟会	4	ACT
6	张玉	4	ACT
7	柳青	4	ACT
8	胡东	4	ACT
9	商乾	4	ACT
10	王蒙	2	ACT
11	王武	0	ACT
12	陆逊	0	ACT

可见，在创建序列时，已经成功利用 start with 指定序列的初始值。



11.2 修改序列属性

像其他数据库对象一样，可以通过 alter 命令修改序列属性。可修改的属性包括 minvalue、maxvalue、increment_by、cycle 和 cache。

11.2.1 修改 minvalue 和 maxvalue

minvalue 和 maxvalue 用于指定序列的最小值和最大值。序列最小值的意义在于限定 start with 和循环取值时的起始值；而最大值则用于限制序列所能达到的最大值。序列最小值不能大于序列的当前值。例如，尝试将序列 employee_start with 的最小值设置为 20，Oracle 将会抛出错误提示。

```
SQL> alter sequence employee_seq minvalue 20;
```

```
alter sequence employee_seq minvalue 20
```

```
ORA-04007: MINVALUE cannot be made to exceed the current value
```

【代码说明】alter sequence employee_seq minvalue 20 尝试修改序列 employee_seq 的最小值为 20；minvalue cannot be made to exceed the current value 意即，最小值不能大于当前值。

相较于最小值，最大值往往更值得修改。以下代码用于修改序列的最大值。



```
SQL> alter sequence employee_seq maxvalue 99999;
```

```
Sequence altered
```

【代码说明】alter sequence employee_seq maxvalue 99999 用于将序列的最大值修改为 99999。

有些情况下，甚至需要将某个序列的最大值设为无限。以序列 employee_seq 为例，相应的代码如下：

```
SQL> alter sequence employee_seq nomaxvalue;
```

```
Sequence altered
```

【代码说明】alter sequence employee_seq nomaxvalue 用于将序列 employee_seq 的最大值修改为无限：nomaxvalue 用来代替选项 maxvalue n。

11.2.2 修改 increment by

increment by 相当于编程语言 for 循环中的步长。即每次使用 nextval 时，在当前值累加该步长来获得新值。序列的默认步长为 1，可以通过 alter 命令和 increment by 选项来修改序列步长。

(1) 创建一个新的序列。

```
SQL> create sequence test_seq;
```

```
Sequence created
```

(2) 测试此时的步长。

```
SQL> select test_seq.nextval from dual;
```

```
NEXTVAL
```

```
-----  
1
```

```
SQL> select test_seq.nextval from dual;
```

```
NEXTVAL
```

```
-----  
2
```

通过两次调用序列 test_seq 的 nextvalue 可知，序列的默认步长为 1。

(3) 利用 alter 命令修改步长为 5。

```
SQL> alter sequence test_seq increment by 5;
```

```
Sequence altered
```

(4) 测试此时 test_seq 的步长。

```
SQL> select test_seq.nextval from dual;
```

```
NEXTVAL
```

```
-----  
7
```

```
SQL> select test_seq.nextval from dual;
```





```
NEXTVAL
```

```
-----
12
```

分析查询结果可知，序列 `test_seq` 的步长已经成功修改为 5。

(5) 需要特别注意的是，修改序列的步长，很可能会影响获得最大值。例如，将序列 `test_seq` 的步长修改为 5 之后，将其最大值修改为 20。

```
SQL> alter sequence test_seq maxvalue 20;
```

```
Sequence altered
```

(6) 再次连续获取序列 `test_seq` 的 `nextval`，期望获得最大值 20。

```
SQL> select test_seq.nextval from dual;
```

```
NEXTVAL
```

```
-----
17
```

```
SQL> select test_seq.nextval from dual;
```

```
select test_seq.nextval from dual
```

```
ORA-08004: sequence TEST_SEQ.NEXTVAL exceeds MAXVALUE and cannot be
instantiated
```

(7) 可见，尝试获得最大值 20 的操作将会失败。因为此时 `select test_seq.nextval from dual` 的值已经为 22，而数值 22 超过了最大值 20。序列 `test_seq.nextval` 能够获得的最大值为 17。

(8) 应该将序列 `test_seq` 的步长修改为 1，可以获得 `test_seq` 的最大值 20。但获得最大值 20 之后，再次使用 `test_seq.nextval`，Oracle 仍会抛出错误：

```
.SQL> select test_seq.nextval from dual;
```

```
NEXTVAL
```

```
-----
20
```

```
SQL> select test_seq.nextval from dual;
```

```
select test_seq.nextval from dual
```

```
ORA-08004: sequence TEST_SEQ.NEXTVAL exceeds MAXVALUE and cannot be
instantiated
```

(9) 对于最大值的下一个值的获取，Oracle 提供了一种处理机制——循环获取。

11.2.3 修改 cycle

`cycle` 选项用于指定序列在获得最大值的下一个值时，从头开始获取。这里的“头”即为 `minvalue` 指定的值。为了说明 `cycle` 的功能及 `start with` 与 `minvalue` 的区别，首先创建该序列，并为各选项指定特定值。

```
SQL> create sequence test_seq
2   start with 5
3   minvalue 1
4   maxvalue 30
```



```
5  increment by 1
6  /
```

Sequence created

(1) 首次获取 `test_seq.nextval`, 其值为 5。

```
SQL> select test_seq.nextval from dual;
```

NEXTVAL

5

(2) 在经历多次测试之后, 可以获得其最大值 30。

```
SQL> select test_seq.nextval from dual;
```

NEXTVAL

30

(3) 此时, 将序列 `test_seq` 修改为可循环。

```
SQL> alter sequence test_seq cycle;
```

Sequence altered

(4) 再次尝试获取其 `nextval` 属性。

```
SQL> select test_seq.nextval from dual;
```

NEXTVAL

1

(5) 可见, 此时的 `nextval` 属性的值为 1, 即序列的最小值。

(6) 要关闭该选项, 不允许序列的循环取值, 可利用 `nocycle` 选项:

```
SQL> alter sequence test_seq nocycle;
```

Sequence altered

11.2.4 修改 cache

顾名思义, `cache` 是序列缓存, 其实际意义为, 每次利用 `nextval`, 并非直接操作序列, 而是一次性获取多个值的列表到缓存。使用 `nextval` 获得的值, 实际是从缓存抓取。抓取的值, 依赖于序列的 `currval` 和步长 `increment by`。默认缓存的大小为 20, 可以通过 `alter` 命令修改缓存大小。可以通过如下步骤测试 `cache` 的存在。

(1) 尝试将序列 `test_seq` 的 `increment by` 属性修改为 2。

```
SQL> alter sequence test_seq increment by 2;
```

```
alter sequence test_seq increment by 2
```

ORA-04013: number to CACHE must be less than one cycle

修改时, Oracle 将会抛出错误——缓存的值 (在这里, 即容量 20) 必须小于一次循环所能获得的数目。因为步长为 2, 最大值为 30, 所以一次循环所能获得的 `nextval` 的数目为 15 (实际为 15.5, 因为第一次为 1, 第二次为 3, 依此类推)。Oracle 不能一次抓取 20 条记录。



(2) 现将尝试将序列 `test_seq` 的最大值 `maxvalue` 修改为临界值 39, 并再次将步长修改为 2, 仍然会导致失败。

```
SQL> alter sequence test_seq maxvalue 39;
```

```
Sequence altered
```

```
SQL> alter sequence test_seq increment by 2;
```

```
alter sequence test_seq increment by 2
```

```
ORA-04013: number to CACHE must be less than one cycle
```

因为 Oracle 在一次性抓取 20 条记录之后, 发现已经达到了序列的最大值, 不符合 `less than` 的条件。

(3) 将序列 `test_seq` 的最大值修改为 40, 再次尝试将步长修改为 2。

```
SQL> alter sequence test_seq maxvalue 40;
```

```
Sequence altered
```

```
SQL> alter sequence test_seq increment by 2;
```

```
Sequence altered
```

(4) 通过以上示例的执行结果可知, 序列的 `cache` 的存在会对修改其他选项产生影响。当然, 可以通过 `alter` 命令修改 `cache` 的大小, 以适应具体序列的要求, 代码如下所示:

```
SQL> alter sequence test_seq cache 10;
```

```
Sequence altered
```

【代码说明】`alter sequence test_seq cache 10` 用于将序列缓存的容量修改为 10。需要注意的是, 不能够将序列的缓存容量设置过大, 其原因与修改 `increment by` 选项的原因相同。



11.3 本章实例

对于范例 11-2 中的单条插入, 使用序列可以正常获得流水号。其实, 对于批量插入, 同样可以使用序列来获得相同的效果。

【范例 11-4】演示利用序列为批量插入操作获得流水号。

(1) 首先创建序列 `object_seq`:

```
create sequence target_seq;
```

该序列使用了默认初始值和默认步长 1。

(2) 清空测试表 `target` 中的数据:

```
truncate table target;
```

(3) 利用视图 `user_objects` 获得列 `object_name`、`object_type` 及 `status`, 并将数据插入到表 `target` 中。id 使用 `object_seq` 进行重新编排。

```
insert into target (object_id, object_name, object_type, status)
select target_seq.nextval, object_name, object_type, status from
user_objects;
```



【代码说明】`target_seq.nextval` 用在子查询中，针对每条记录自动获取流水号。

(4) 查询此时表 `target` 中的内容：

```
SQL> select * from target;
```

OBJECT_ID	OBJECT_NAME	OBJECT_TYPE	PREVIOUS_NAME	STATUS
1	SYSCATALOG	SYNONYM		VALID
2	CATALOG	SYNONYM		VALID
3	TAB	SYNONYM		VALID
4	COL	SYNONYM		VALID
5	TABQUOTAS	SYNONYM		VALID
.				
.				
.				

可见，序列可以在子查询中直接使用。



11.4 本章小结

本章通过实例讲解了如何创建和使用一个序列，并详细讲解了如何修改序列的各个属性。在修改属性时应该注意：无论修改哪个选项值（`minvalue`、`maxvalue`、`increment by`、`cache`）都应该遵循一个原则——Oracle 能够一次性抓取序列中与 `cache` 容量相同数目的值，并且序列中仍然存在 `cache` 所含数值之外的值。



11.5 习题

1. 创建序列时，默认 `start with` 和 `increment by` 的值是多少？
2. 简述 `start with` 和 `minvalue` 的区别。
3. 简述 `cycle` 的作用。
4. 序列的主要应用场景是什么？



第 12 章 用户角色与权限控制

每次登录 Oracle 数据库，用户和密码是最基本的信息。当使用特定的用户登录数据库时，该用户便具有自己的特点和操作权限。角色则是权限的集合，角色可以分配给用户，相当于一次性将某个特定权限集合分配给用户。Oracle 正是通过这三个重要的对象来实现数据库操作的安全策略。本章的主要内容包括：

- 用户及用户的创建；
- 权限及权限的分配；
- 角色及角色的使用。

通过本章的学习，读者将清晰地了解用户、权限和角色这三者的概念及相互关系，并掌握如何使用这三种对象。



12.1 用户

用户是数据库中最基本的对象之一。在前面的内容中，登录数据库一直使用了 `system` 用户。而该用户是管理员级别的用户，拥有数据库大多数对象的操作权限。在正式开发过程中，使用该用户将是不安全的。一旦操作失当，有可能对数据库造成严重损害。本节将讲述 Oracle 中用户的基本情况以及如何创建用户。

12.1.1 Oracle 中的用户

Oracle 中的用户可以分为两类：一类是 Oracle 数据库创建时，由系统自动创建的用户，称为系统用户，如 `sys` 和 `system`；另一类用户是利用系统用户创建的用户，称为普通用户。可以通过查询视图 `dba_users` 来查看当前数据库的所有用户状况。

```
select username, account_status, default_tablespace, temporary_tablespace from dba_users
```

查询结果如下：

USERNAME	ACCOUNT_STATUS	DEFAULT_TABLESPACE	TEMPORARY_TABLESPACE
SYSTEM	OPEN	SYSTEM	TEMP
SYS	OPEN	SYSTEM	TEMP
OLAPSYS	EXPIRED & LOCKED	SYSAUX	TEMP
SI_INFORMTN_SCHEMA	EXPIRED & LOCKED	SYSAUX	TEMP
MGMT_VIEW	OPEN	SYSAUX	TEMP
ORDPLUGINS	EXPIRED & LOCKED	SYSAUX	TEMP
WKPROXY	EXPIRED & LOCKED	SYSAUX	TEMP
TEST	OPEN	USERS	TEMP
XDB	EXPIRED & LOCKED	SYSAUX	TEMP
SYSMAN	OPEN	SYSAUX	TEMP
DIP	EXPIRED & LOCKED	USERS	TEMP
OUTLN	EXPIRED & LOCKED	SYSTEM	TEMP
ANONYMOUS	EXPIRED & LOCKED	SYSAUX	TEMP
CTXSYS	EXPIRED & LOCKED	SYSAUX	TEMP



MDDATA	EXPIRED & LOCKED	USERS	TEMP
WK_TEST	EXPIRED & LOCKED	SYSAUX	TEMP
WKSYS	EXPIRED & LOCKED	SYSAUX	TEMP
TEST1	OPEN	USERS	TEMP
WMSYS	EXPIRED & LOCKED	SYSAUX	TEMP
SCOTT	EXPIRED & LOCKED	USERS	TEMP
DBSNMP	OPEN	SYSAUX	TEMP
DMSYS	EXPIRED & LOCKED	SYSAUX	TEMP
EXFSYS	EXPIRED & LOCKED	SYSAUX	TEMP
ORDSYS	EXPIRED & LOCKED	SYSAUX	TEMP
MDSYS	EXPIRED & LOCKED	SYSAUX	TEMP

【代码说明】列 USERNAME 标识了用户的登录名；列 ACCOUNT 标识了账号的当前状态，处于 OPEN 状态的账号为可用账号；而处于 EXPIRED&LOCKED 状态的账号则是过期和锁定账号，此状态的用户不能登录数据库；DEFAULT_TABLESPACE 和 TEMPORARY_TABLESPACE 列分别标识了用户的默认和临时表空间。

12.1.2 创建新用户

可以在 Oracle 中创建新的用户。创建用户应该使用 create user 命令，在创建普通用户的同时，应为其分配一个具体的表空间。

【范例 12-1】演示如何创建一个名为 tiger 的普通用户。

```
SQL> create user tiger
2 identified by abc
3 default tablespace users
4 /
```

User created

【代码说明】create user tiger 用于创建一个名为 tiger 的用户；identified by abc 用于指定该用户的密码为 abc；default tablespace users 指定该用户的默认表空间为 users。

可以在视图 dba_users 中查看新建用户的信息。

```
select username, account_status, default_tablespace, temporary_tablespace from
dba_users
```

查询结果如下：

```
SQL> select username, account_status, default_tablespace, temporary_tablespace
from dba_users where lower(username) = 'tiger';
```

USERNAME	ACCOUNT_STATUS	DEFAULT_TABLESPACE	TEMPORARY_TABLESPACE
TIGER	OPEN	USERS	TEMP

通过查询结果可知，用户 tiger 已经成功创建。创建后的状态为 OPEN，默认表空间为 users，临时表空间为 temp。

12.1.3 用户与模式 (Schema)

模式是指用户所拥有的所有对象的集合。这些对象包括：表、索引、视图、存储过程等。每个用户都会有独立的模式信息。当然，对于新建用户，在没有创建任何对象时，所拥有的对象集合为空，Schema 同样为空。但是，Schema 必须依赖于用户的存在而存在，即不存在不属于任何用户的 Schema 对象。

可以利用以下 SQL 语句来查看当前用户及其 Schema 信息。

```
SQL> select sys_context('userenv', 'current_user') current_user, sys_context
```

```
('userenv', 'current_schema') current_schema from dual;
```

```
CURRENT_USER    CURRENT_SCHEMA
```

```
SYSTEM          SYSTEM
```

可见, 使用 system 登录之后, 当前的 Schema 即为 system。

Schema 的意义在于标识某个对象的所有者 (OWNER)。Schema+对象名的组合可以限制数据库中唯一对象。例如, 查询语句 `select * from t_employees`, 在执行时, 实际被 Oracle 翻译为 `select * from system.t_employees`。

在不同的 Schema 中可以存在同名对象。虚表 dual 属于 Schema sys, 所以用户 system 仍然可以在自己的 Schema 下创建名为 dual 的表。但是, 此时使用 `select * from dual`, 执行的 SQL 语句为 `select * from system.dual`, 而并非 `select * from sys.dual`。

用户可以跨 Schema 进行操作, 例如, `select * from sys.dual`。这是因为用户 system 具有访问 sys.dual 的权限。如果用户在当前的 Schema 中找不到操作对象, 则会在其他具有操作权限的 Schema 中进行查找。

12.1.4 系统用户 sys 和 system

系统用户 sys 和 system 是 Oracle 数据库常用的两个系统用户。其中 sys 是 Oracle 数据库中最高权限用户, 其角色为 SYSDBA (数据库管理员); 而 system 用户的权限仅次于 sys 用户, 其角色为 SYSOPER (数据库操作员)。在权限的范围上, sys 可以创建数据库, 而 system 则不可以。

system 用户密码丢失是一个常见问题。例如, 当多次输入错误的密码之后, Oracle 会锁定 system 账号, 不允许用户再次登录。

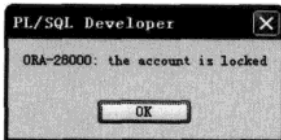


图 12-1 Oracle 锁定账户

图 12-1 演示了利用 PL/SQL Developer 登录数据库, 但 system 账号被锁定的情形。

此时, 可以利用 sys 来进行密码重设。

【范例 12-2】 演示如何利用 sys 用户重设 system 用户的密码。

(1) 利用 sys 账号以 DBA 角色登录数据库。

```
sqlplus /@test as sysdba
```

【代码说明】 sqlplus 用于启动 sqlplus; /实际上代表了用户名和密码, 如果以 sysdba 身份登录, 可以不输入该密码; @test 指定要登录的数据库的 Net 服务名, 因为本书的所有实例都建立在数据库 test 上, 因此, 此处使用了数据库 test 的 Net 服务名——test; as sysdba 表示以系统管理员的身份登录。

(2) 成功登录数据库后, 检查数据库名称, 以保证正在修改的数据库是预期修改的。

```
SQL> show parameter db_name;
```

NAME	TYPE	VALUE
db_name	string	test

【代码说明】 show parameter db_name 用于显示名为 db_name (即数据库名称) 的参数信息。分析显示结果可知, 此时的数据库名称为 test, 正是预期修改的数据库。

(3) 查看此时用户 system 的账户状态。

```
SQL> select username, account_status from dba_users where lower(username) = 'system';
```



USERNAME	ACCOUNT_STATUS
SYSTEM	LOCKED (TIMED)

(4) 可以利用 **alter** 命令来修改用户 **system** 的属性，首先将其从锁定状态中恢复。

```
SQL> alter user system account unlock;
```

User altered.

(5) 再次查看此时用户 **system** 的账户状态。

```
SQL> select username, account_status from dba_users where lower(username) = 'system';
```

USERNAME	ACCOUNT_STATUS
SYSTEM	OPEN

(6) 利用 **alter** 命令和 **identified by** 选项重设 **system** 账户的密码。

```
SQL> alter user system identified by abc123;
```

User altered.

【代码说明】**alter user system identified by abc123** 用于修改 **system** 账户的密码为 **abc123**。注意，虽然 Oracle 的账户不区分大小写，但是密码是区分大小写的。因此，**abc123** 和 **ABC123** 是完全不同的密码。

(7) 重新使用 **system** 账号登录数据库 **test**，会发现已经可以成功登录。



12.2 权限

权限 (Privilege) 的最终作用对象是用户。即所有用户在数据库内的操作对象和可执行的动作都是受到限制的。Oracle 中共有两种权限：系统权限和对象权限。

12.2.1 系统权限

系统权限是指针对数据库中特定操作 (例如，创建数据表) 的许可。与对象权限不同，对象权限是指对已存在对象 (例如，表、视图等) 的访问和操作权限。

1. 获得系统权限的相关信息

可以通过视图 **dba_sys_privs** 来获得系统权限的相关信息。

```
SQL> select distinct privilege from dba_sys_privs;
```

PRIVILEGE
ADMINISTER ANY SQL TUNING SET
ADMINISTER DATABASE TRIGGER
ADMINISTER RESOURCE MANAGER
ADMINISTER SQL TUNING SET
ADVISOR
ALTER ANY CLUSTER
ALTER ANY DIMENSION
ALTER ANY EVALUATION CONTEXT
ALTER ANY INDEX
ALTER ANY INDEXTYPE
.
.
BACKUP ANY TABLE



```
BECOME USER
COMMENT ANY TABLE
CREATE ANY CLUSTER
.
.
.
CREATE USER
CREATE VIEW
DEBUG ANY PROCEDURE
DEBUG CONNECT SESSION
DELETE ANY TABLE
DEQUEUE ANY QUEUE
.
.
.
DROP TABLESPACE
DROP USER
ENQUEUE ANY QUEUE
EXECUTE ANY CLASS
EXECUTE ANY EVALUATION CONTEXT
EXECUTE ANY INDEXTYPE
.
.
.
SELECT ANY DICTIONARY
SELECT ANY SEQUENCE
SELECT ANY TABLE
SELECT ANY TRANSACTION
.
.
.
UPDATE ANY TABLE
```

虽然在查询结果中省略了部分权限名称，但仍然可以总结出系统权限的主要组成部分：

- 用于创建新的对象。这是与对象权限极大的区别，因为对象权限总是针对已存在对象。
- 包括了对对象的各种操作。
- 并不针对特定对象，而是使用了 `any` 关键字，表示针对所有对象。例如，`select any table` 表示针对所有表的 `select` 权限。

2. 为用户分配权限

对于新建用户 `tiger` 来说，该用户在创建时没有分配任何系统权限。现尝试使用该对象执行创建表的操作。

【范例 12-3】 演示如何为用户分配权限。

(1) 利用该用户登录数据库 `test`，会发现 Oracle 给出错误提示。

```
sqlplus tiger/abc@test
```

```
ERROR:
```

```
ORA-01045: user TIGER lacks CREATE SESSION privilege; logon denied
```

这是因为用户 `tiger` 没有 `create session`——即与数据库建立会话的权限。

(2) 重新使用 `system` 用户登录 `test` 数据库，并赋予用户 `tiger` 创建会话的权限。

```
sqlplus system/abc123@test
```

```
SQL> grant create session to tiger;
```

```
Grant succeeded.
```



(3) 此时, 可以利用用户 `tiger` 登录数据库 `test`。尝试创建数据表 `tiger`。

```
SQL> create table tiger(name varchar2(20));
create table tiger(name varchar2(20))
*
ERROR at line 1:
ORA-01031: insufficient privileges
```

(4) 分析错误提示可知, 用户 `tiger` 不具备创建表的权限。因此, 再次利用 `system` 账户为其分配建表权限。

```
SQL> grant create table to tiger;
```

Grant succeeded.

(5) 在成功分配创建新表的权限之后, 尝试利用 `tiger` 账号创建新表。

```
SQL> create table tiger(name varchar2(20));
create table tiger(name varchar2(20))
*
ERROR at line 1:
ORA-01950: no privileges on tablespace 'USERS'
```

错误显示, 当前用户在其默认表空间 `users` 上, 权限不足。这是因为没有在表空间中为其分配有效的可用空间。

(6) 可以利用 `system` 账户进行空间分配。

```
SQL> alter user tiger
2 quota 10M on users
3 quota 2M on temp;
```

User altered.

【代码说明】`quota` 选项用于为用户 `tiger` 分配可用空间, 其中在 `users` 表空间中分配了 10MB, 在 `temp` 表空间中分配了 2MB。

(7) 此时, 利用 `tiger` 账号成功创建表。

```
SQL> create table tiger(name varchar2(20));
```

Table created.

(8) 利用 `system` 账户登录数据库 `test`, 并查询用户 `tiger` 所拥有的系统权限。

```
SQL> select * from dba_sys_privs where lower(grantee) = 'tiger';
```

GRANTEE	PRIVILEGE	ADMIN_OPTION
TIGER	CREATE TABLE	NO
TIGER	CREATE SESSION	NO

3. 收回用户的系统权限

在为用户分配了若干系统权限之后, 可以利用 `revoke` 命令收回权限。

【范例 12-4】演示如何收回用户 `tiger` 的 `create table` 权限。

```
SQL> revoke create table from tiger;
```

Revoke succeeded

【代码说明】`revoke` 命令用于收回权限; `create table` 为欲收回的权限; `from tiger` 则指定收回用户 `tiger` 的权限。需要注意的是, 此操作的执行者为系统用户 `system`。

在成功收回用户 `tiger` 的 `create table` 权限之后, 再次尝试创建新表。

```
SQL> create table test(test_data varchar2(2));
```




```
create table test(test_data varchar2(2))
```

```
ORA-01031: insufficient privileges
```

【代码说明】insufficient privileges 表明在 create table 权限收回之后，用户 tiger 不再具有创建表的权限。

12.2.2 对象权限

对象权限是指用户对已有对象的操作权限。这些权限包括以下几种：

- select: 可用于查询表、视图和序列
- insert: 向表或视图中插入新的记录
- update: 更新表中数据
- delete: 删除表中数据
- execute: 函数、存储过程、程序包等的调用或执行
- index: 为表创建索引
- references: 为表创建外键
- alter: 修改表或者序列的属性

可以利用视图 dba_tab_privs 来查看某个用户所具有的对象权限。以用户 tiger 为例，相应的代码如下所示：

```
SQL> select grantee, table_name, privilege from dba_tab_privs where lower(grantee) = 'tiger';
```

GRANTEE	TABLE_NAME	PRIVILEGE

对于新建用户 tiger，还没有任何的对象权限。

为用户赋予对象权限应该使用 grant 命令。本节将通过几个范例讲述如何为用户赋予对象权限。

1. 为用户赋予 select 权限

【范例 12-5】演示如何为用户赋予 select 权限。

(1) 使用用户 tiger 登录数据库 test，尝试查询表 dual。

```
SQL> select * from dual;
```

```
select * from dual
```

```
ORA-00942: table or view does not exist
```

(2) 这是因为用户 tiger 没有表 dual 的访问权限。因此，应该利用 SYS 账号（因为表 dual 的所有者是 sys）登录数据库 test，并为用户 tiger 赋予表 dual 的 select 权限。

```
SQL> grant select on dual to tiger;
```

```
Grant succeeded
```

【代码说明】grant 用于分配权限；select 指定权限类别为查询；on dual 指定权限的对象——当前 Schema 中的 dual 表；to tiger 指定将 select 权限赋予用户 tiger。

(3) 再次利用用户 tiger 查询表 dual。

```
SQL> select * from dual;
```



```
DUMMY
-----
X
```

2. 为用户赋予 insert 权限

【范例 12-6】 演示如何为用户赋予 insert 权限。

在用户 system 的 Schema 下，有名为 t_employee 的表。现尝试利用 tiger 向其中插入新的数据。

(1) 利用账号 system 为用户 tiger 赋予表 t_employees 的查询权限。

```
SQL> grant select on t_employees to tiger;
```

Grant succeeded

(2) 那么，可以利用用户 tiger 查询表 t_employees。

```
SQL> select * from system.t_employees order by employee_id;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	6	ACT
2	钟君	6	ACT
3	王山	6	ACT
4	刘迪	5	ACT
5	钟会	4	ACT
6	张玉	4	ACT
7	柳青	4	ACT
8	胡东	4	ACT
9	商乾	4	ACT
10	王蒙	2	ACT
11	王武	0	ACT
12	陆逊	0	ACT

12 rows selected

(3) 现尝试向其中插入新的数据。

```
SQL> insert into system.t_employees values (13, '张俊', 10, 'ACT');
```

```
insert into system.t_employees values (13, '张俊', 10, 'ACT')
```

ORA-00942: table or view does not exist

(4) 利用 system 账号为其分配 insert 权限。

```
SQL> grant insert on t_employees to tiger;
```

Grant succeeded

(5) 再次利用用户 tiger 向表中插入数据，则可以成功插入。

```
SQL> insert into system.t_employees values (13, '张俊', 10, 'ACT');
```

1 row inserted

3. 为用户赋予 update 权限

【范例 12-7】 演示如何为用户赋予 update 权限。

对象（在这里指表和视图）的 update 权限可分为两类：一类为对象中部分列的更新；另一类为对象中所有列的更新。



(1) 首先利用用户 `system` 为用户 `tiger` 赋予表 `t_employees` 中 `employee_name` 列的更新权限。

```
SQL> grant update(employee_name) on t_employees to tiger;
```

【代码说明】`grant update(employee_name) on t_employees to tiger` 用于赋予用户 `tiger` 更新表 `t_employees` 的权限，小括号中的列名限制了仅能更新 `employee_name` 列。

```
Grant succeeded
```

(2) 尝试利用用户 `tiger` 更新 `system.t_employees` 中的 `employee_name` 列。

```
SQL> update system.t_employees set employee_name = '李俊' where employee_id = 13;
```

```
1 row updated
```

(3) 如果更新表 `t_employees` 中，除 `employee_name` 之外的列，将会导致更新失败。

```
SQL> update system.t_employees set status = 'CXL' where employee_id = 13;
```

```
update system.t_employees set status = 'CXL' where employee_id = 13
```

```
ORA-01031: insufficient privileges
```

(4) 可以为用户 `tiger` 重新分配权限，如下所示：

```
SQL> grant update on t_employees to tiger;
```

```
Grant succeeded
```

【代码说明】`grant update on t_employees to tiger` 为用户 `tiger` 分配表 `t_employees` 的所有列的更新权限。

(5) 再次利用用户 `tiger` 更新 `status` 列的数据，则可以成功完成更新动作。

```
SQL> update system.t_employees set status = 'CXL' where employee_id = 13;
```

```
1 row updated
```

为用户分配对象的其他权限，包括 `delete`、`references` 等，与分配 `select`、`update` 权限类似。在此，不再一一赘述。

4. 为用户赋予 all 权限

在经历了以上的为 `tiger` 分配表 `t_employees` 的单个权限之后，可以查看此时该表的权限信息。

```
SQL> select table_name, grantee, grantor, privilege from dba_tab_privs where table_name = 'T_EMPLOYEES';
```

TABLE_NAME	GRANTEE	GRANTOR	PRIVILEGE
T_EMPLOYEES	TIGER	SYSTEM	INSERT
T_EMPLOYEES	TIGER	SYSTEM	SELECT
T_EMPLOYEES	TIGER	SYSTEM	UPDATE

查询结果与以上的权限分配动作完全一致。但是，如果预期一个用户对其他 Schema 中的对象拥有所有权限，那么进行分配将是非常耗时和费力的工作。此时，可以利用 `all` 关键字，来一次性分配所有权限。以为用户 `tiger` 分配 `system.t_employees` 所有权限为例，相应的代码如范例 12-8 所示。

【范例 12-8】演示如何利用 `all` 关键字分配所有权限。

```
SQL> grant all on t_employees to tiger;
```



Grant succeeded

【代码说明】grant all on t_employees to tiger 为用户 tiger 分配表 t_employees 的所有对象权限。

此时，查看表 t_employees 的权限分配信息。

```
SQL> select table_name, grantee, grantor, privilege from dba_tab_privs where
table_name = 'T_EMPLOYEES';
```

TABLE_NAME	GRANTEE	GRANTOR	PRIVILEGE
T_EMPLOYEES	TIGER	SYSTEM	ALTER
T_EMPLOYEES	TIGER	SYSTEM	DELETE
T_EMPLOYEES	TIGER	SYSTEM	INDEX
T_EMPLOYEES	TIGER	SYSTEM	INSERT
T_EMPLOYEES	TIGER	SYSTEM	SELECT
T_EMPLOYEES	TIGER	SYSTEM	UPDATE
T_EMPLOYEES	TIGER	SYSTEM	REFERENCES
T_EMPLOYEES	TIGER	SYSTEM	ON COMMIT REFRESH
T_EMPLOYEES	TIGER	SYSTEM	QUERY REWRITE
T_EMPLOYEES	TIGER	SYSTEM	DEBUG
T_EMPLOYEES	TIGER	SYSTEM	FLASHBACK

利用 delete 动作来测试针对表 t_employees 的权限。

```
SQL> delete from system.t_employees where employee_id = 13;
```

1 row deleted

可见，利用 all 关键字的确将所有操作权限一次性分配给了用户 tiger。

5. 收回用户的对象权限

如同收回用户的系统权限，可以利用 revoke 命令收回用户的对象权限，其语法略有不同。

【范例 12-9】all 收回用户 tiger 在 system.t_employees 上的 select 权限。

```
SQL> revoke select on t_employees from tiger;
```

Revoke succeeded

【代码说明】revoke 命令用于收回权限；select on t_employees 表明欲收回的权限是表 t_employees 的 select 权限；from tiger 则指定了收回的目标用户为 tiger。

此时，利用用户 tiger 来查询 system.t_employees 中的数据，将会抛出权限不足的错误提示，如下所示：

```
SQL> select * from system.t_employees;
```

```
select * from system.t_employees
```

```
ORA-01031: insufficient privileges
```

对于利用 all 进行分配的权限，同样可以通过 revoke all 命令来一次性收回所有权限。

```
SQL> revoke all on t_employees from tiger;
```

Revoke succeeded

此时，查看表 t_employees 的权限信息，并与范例 11-7 进行对比，会发现用户 tiger 所有基于表 system.t_employees 的权限都不复存在了。

```
SQL> select table_name, grantee, grantor, privilege from dba_tab_privs where
table_name = 'T_EMPLOYEES';
```

TABLE_NAME	GRANTEE	GRANTOR	PRIVILEGE



12.3 角色

虽然可以利用 `grant` 命令为所有用户分配权限，但是如果数据库的用户众多，而且权限关系复杂，因此为用户分配权限的工作量将变得十分巨大。因此，Oracle 提出了角色的概念。

角色是指系统权限或者对象权限的集合。Oracle 允许首先创建一个角色，然后将角色信息赋予用户，从而间接地将权限信息添加给用户。因为角色的可复用性，因此，可以将角色再次分配给其他用户，从而减少了重复工作。

就角色的创建来说，可以利用继承的特性，从简单的角色衍生出复杂的角色，这无疑大大提高了权限分配工作的效率。

12.3.1 创建和使用角色

创建角色可以使用命令 `create role`，并可以利用 `grant` 命令为角色分配权限，最后将角色信息赋予用户。

【范例 12-10】 演示如何创建和使用角色。

(1) 对于表 `system.t_employees`，现欲创建一个名为 `role_employee` 的角色，该角色只具有查询权限，那么可以利用账号 `system` 登录数据库，并创建该角色。

```
SQL> create role role_employee;
```

```
Role created
```

【代码说明】 `create role` 命令用于创建新的角色，新角色的名称为 `role_employee`。

(2) 在成功创建角色之后，可以利用 `grant` 命令对角色进行权限分配。

```
SQL> grant select on t_employees to role_employee;
```

```
Grant succeeded
```

(3) 此时，查看表 `t_employees` 的权限信息。

```
SQL> select table_name, grantee, grantor, privilege from dba_tab_privs where
table_name = 'T_EMPLOYEES';
```

TABLE_NAME	GRANTEE	GRANTOR	PRIVILEGE
T_EMPLOYEES	ROLE_EMPLOYEE	SYSTEM	SELECT

(4) 此时的 `grantee` (被分配者) 是角色 `role_employee`。可以利用 `grant` 命令将角色的权限信息赋予用户 `tiger`。

```
SQL> grant role_employee to tiger;
```

```
Grant succeeded
```

【代码说明】 `grant` 命令用于权限分配；`role_employee` 是角色，封装了权限信息；`to tiger` 则是权限分配的目标。

(5) 尝试利用 `tiger` 用户查询表 `t_employees`。

```
SQL> select * from system.t_employees order by employee_id;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS



1	金瑞	6	ACT
2	钟君	6	ACT
3	王山	6	ACT
4	刘迪	5	ACT
5	钟会	4	ACT
6	张玉	4	ACT
7	柳青	4	ACT
8	胡东	4	ACT
9	商乾	4	ACT
10	王蒙	2	ACT
11	王武	0	ACT
12	陆逊	0	ACT

12 rows selected

【代码说明】通过查询结果可知，用户 tiger 具有查询表 system.t_employees 的权限。这里需要注意的是，在使用了 grant 命令将角色信息赋予用户之后，用户 tiger 需要重登录数据库，权限信息的更新才能生效。

12.3.2 继承角色

角色继承是指一个角色可以继承其他角色的权限信息，从而减少自身使用 grant 的机会。范例 12-11 演示了如何继承一个角色。

【范例 12-11】演示如何继承角色。

(1) 在已有角色 role_employee 之后，希望能够增加一个新的角色 role_hr，该角色的权限为允许对表 t_employees 的查询、修改和插入操作。那么，首先应该创建该角色，因为 role_employee 已经包含了对表 t_employees 的查询操作的权限，所以，role_hr 可以继承 role_employee。

```
SQL> create role role_hr;
```

Role created

接着赋予角色 role_hr 适当的权限信息。

```
SQL> grant role_employee to role_hr;
```

Grant succeeded

```
SQL> grant update, insert on t_employees to role_hr;
```

Grant succeeded

【代码说明】grant role_employee to role_hr 首先将角色 role_employee 的权限信息赋予角色 role_hr; grant update, insert on t_employees to role_hr 为角色 role_hr 赋予更新、插入表 t_employees 数据的权限。

(2) 查询此时表 t_employees 的权限信息。

```
SQL> select table_name, grantee, grantor, privilege from dba_tab_privs where
table_name = 'T_EMPLOYEES';
```

TABLE_NAME	GRANTEE	GRANTOR	PRIVILEGE
T_EMPLOYEES	ROLE_EMPLOYEE	SYSTEM	SELECT
T_EMPLOYEES	ROLE_HR	SYSTEM	INSERT
T_EMPLOYEES	ROLE_HR	SYSTEM	UPDATE

【代码说明】因为表 `t_employees` 的查询权限是通过角色 `role_employee` 间接赋予角色 `role_hr` 的，因此，并未在查询结果中反映出来。

(3) 为了验证角色 `role_hr` 的有效性，并避免与用户 `tiger` 已有角色的混淆，可以重新创建一个用户 `cat`。

```
SQL> create user cat identified by abc;
```

User created

(4) 为用户 `cat` 赋予 `create session` 的系统权限，以保证其正常登录。

```
SQL> grant create session to cat;
```

Grant succeeded

(5) 将 `role_hr` 的权限信息赋予用户 `cat`。

```
SQL> grant role_hr to cat;
```

Grant succeeded

(6) 利用用户 `cat` 登录数据库，并测试其权限信息。

```
SQL> select * from system.t_employees order by employee_id;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	6	ACT
2	钟君	6	ACT
3	王山	6	ACT
4	刘迪	5	ACT
5	钟会	4	ACT
6	张玉	4	ACT
7	柳青	4	ACT
8	胡东	4	ACT
9	商乾	4	ACT
10	王蒙	2	ACT
11	王武	0	ACT
12	陆逊	0	ACT

12 rows selected

(7) 可见，通过角色 `role_employee` 传递给角色 `role_hr` 的 `select` 权限信息，已成功赋予了用户 `cat`。

(8) 当然，也需要测试用户 `cat` 针对表 `system.t_employees` 的插入和更新权限。

```
SQL> insert into system.t_employees values(13, '王俊', 0, 'ACT');
```

1 row inserted

```
SQL> update system.t_employees set status = 'CXL' where employee_id = 13;
```

1 row updated

(9) 在这里，利用 `role_hr` 继承角色 `role_employee` 的权限信息，似乎显得意义不大。但是如果 `role_employee` 的权限信息相当复杂，那么省略对角色 `role_hr` 的重复赋予权限的操作将会显示出极大的优势。



12.3.3 禁用和启用角色

每个用户登录数据库时，都可以获得其默认角色。可以通过查询视图 `session_roles` 来获得当前会话下该用户的默认角色。管理员可以禁用用户的默认角色，一旦禁用，则用户从该角色获得的权限将不再有效。

【范例 12-12】 演示如何禁用和启用角色。

```
SQL> alter user cat default role none;
```

```
User altered
```

【代码说明】 `alter user` 用于修改用户信息；`default role none` 表示将用户的默认角色全部禁用。

(1) 此时，利用用户 `cat` 登录数据库，并查询当前用户角色。

```
SQL> select * from session_roles;
```

```
ROLE
```

(2) 分析查询结果可知，当前角色信息为空。此时，利用 `cat` 用户登录数据库，并尝试查询表 `system.t_employee`，Oracle 会抛出错误提示。

```
SQL> select * from system.t_employees order by employee_id;
```

```
select * from system.t_employees order by employee_id
```

```
ORA-00942: table or view does not exist
```

(3) 要想启用数据库角色，则可以使用 `set` 命令，如下所示：

```
SQL> set role role_hr;
```

```
Role set
```

【代码说明】 `set role` 用于启用角色；`role_hr` 则为要启用的角色。该参数也可以使一个角色列表，各角色之间使用逗号分隔。

(4) 此时，查询视图 `session_roles` 以获取当前会话的有效角色。

```
SQL> select * from session_roles;
```

```
ROLE
```

```
ROLE_HR
```

```
ROLE_EMPLOYEE
```

【说明】 虽然没有显式启用角色 `role_employee`，但是 `role_hr` 是依赖于角色 `role_employee` 的。因此，启用角色 `role_hr`，同时连带启用了角色 `role_employee`。

(5) 尝试查询表 `system.t_employees` 的数据。

```
SQL> select * from system.t_employees order by employee_id;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	6	ACT
2	钟君	6	ACT
3	王山	6	ACT
4	刘迪	5	ACT
5	钟会	4	ACT
6	张玉	4	ACT

7	柳青	4	ACT
8	胡东	4	ACT
9	商乾	4	ACT
10	王蒙	2	ACT
11	王武	0	ACT
12	陆逊	0	ACT
13	王俊	0	ACT

13 rows selected

(6) 同样，可以验证其他两个权限 `update` 和 `insert` 都已经成功启用。



12.4 本章实例

Oracle 数据库内置了几个默认角色。这些角色包括：`RESOURCE`、`CONNECT` 和 `DBA`。在许多应用系统的开发中，往往为开发人员创建一个新的用户，并为用户分配 `CONNECT` 和 `RESOURCE` 角色。本节将查看这两种角色的主要权限。

【范例 12-13】 演示 `RESOURCE` 和 `CONNECT` 角色的权限信息。

(1) 首先利用 `PL/SQL Developer` 登录数据库 `TEST`。

(2) 查看角色 `CONNECT` 的权限信息：

```
SQL> select * from dba_sys_privs where lower(grantee) = 'connect';
```

GRANTEE	PRIVILEGE	ADMIN_OPTION
CONNECT	CREATE VIEW	NO
CONNECT	CREATE TABLE	NO
CONNECT	ALTER SESSION	NO
CONNECT	CREATE CLUSTER	NO
CONNECT	CREATE SESSION	NO
CONNECT	CREATE SYNONYM	NO
CONNECT	CREATE SEQUENCE	NO
CONNECT	CREATE DATABASE LINK	NO

8 rows selected

`CONNECT` 角色并不像其字面意义那样，只具有连接权限，而是包括了创建表、创建视图等其他权限。

(3) 获得角色 `RESOURCE` 的权限信息：

```
SQL> select * from dba_sys_privs where lower(grantee) = 'resource';
```

GRANTEE	PRIVILEGE	ADMIN_OPTION
RESOURCE	CREATE TYPE	NO
RESOURCE	CREATE TABLE	NO
RESOURCE	CREATE CLUSTER	NO
RESOURCE	CREATE TRIGGER	NO
RESOURCE	CREATE OPERATOR	NO
RESOURCE	CREATE SEQUENCE	NO
RESOURCE	CREATE INDEXTYPE	NO
RESOURCE	CREATE PROCEDURE	NO

8 rows selected

`RESOURCE` 角色除了以上列出的权限之外，还有另外一个隐含的权限——`UNLIMITED TABLESPACE`。该权限使用户不局限于表空间磁盘配额的限制，可以任意扩展大小，因此具有



一定的危险性。



12.5 本章小结

本章详细讲述了用户、权限和角色的相互关系，并通过多个实例来剖析如何创建用户、分配权限和使用角色。尤其需要注意的是角色的使用，合理使用角色可以极大地提高数据库权限管理的效率，减轻数据库 DBA 的工作负担。



12.6 习题

1. 简述用户与模式的关系。
2. 简述系统权限与对象权限的区别。
3. 简述角色的意义。
4. 如何实现两个角色之间的继承？



第三篇 Oracle 中的 SQL

第 13 章 Oracle 数据类型

本章将进入 Oracle 数据类型的学习。Oracle 中的数据类型主要应用场景为数据表中列的类型、函数、存储过程的变量定义。这些数据类型大致可以分为 4 种：字符型（character）、数值型（number）、日期型（date）和大对象型（LOB）。本章的主要内容包括：

- 字符型简介；
- 数值型简介；
- 日期时间型简介；
- lob 类型简介；
- 特殊数据简介。

通过本章的学习，读者将对 Oracle 中的基本数据类型有清晰的了解，并掌握几种特殊数据的用法。



13.1 Oracle 中的数据类型

4 种基本数据类型各有多个实际类型。这些实际类型才可以直接应用于数据表列和变量定义。本节将简要介绍 4 种基本数据类型的各个实际类型。

13.1.1 字符型

字符型是最常用的数据类型，一般用于存储字符串数据。字符串类型包括以下几种实际的数据类型。

1. char(*n*)

char 数据类型用于标识固定长度的字符串。小括号内的数字 *n* 即代表了字符串的长度。当实际数据不足定义长度时，Oracle 将使用空格补全右边不足位；当实际数据的长度大于其固定长度时，Oracle 将不允许数据存储在对应的列或者变量中，并抛出错误。

char 数据类型也有最大长度。即 *n* 的最大值不能大于 2000，同时不能小于 1。

2. varchar(*n*)

varchar(*n*) 类型，是可变字符串类型。当将某个列或变量指定为 varchar(*n*) 的类型时，意味着该列或变量的最大长度不大于 *n*，但当其实际数据小于 *n* 时，Oracle 并不在其右端补齐空格。这样，减少了存储时所占用的实际资源。

3. varchar2(*n*)

varchar2 同样是可变数据类型。与 varchar 类型的区别在于，varchar 为 SQL 标准规定的、数据库必须实现的数据类型，所以 Oracle 数据库中必须存在该数据类型。而 varchar2 则是 Oracle 在 varchar 的基础上自行定义的可变长度的字符串类型，并使用了 varchar2 这个名称。varchar2 当被用做列的数据类型时，其最大长度可被定义为 4000，当用做变量的数据类型时，其长度可



以达到 32 767。但是,需要注意的是,varchar 和 varchar2 的实际数据长度,都不可以大于其定义长度 n 。

varchar 是符合工业标准 (SQL 标准) 的数据类型。该类型可以存储空字符串,而 varchar2 则不存储空字符串,而是将字符串转换为 NULL 进行存储。为了保证数据库的向后兼容性,Oracle 建议使用 varchar2,而非 varchar。

13.1.2 数值型

Oracle 中的数值型可以用于存储整数 (integer)、浮点数 (float) 和实数 (real number)。而所有这些数值类型都被统一为 number 型。Oracle 中的 number 数据类型具有精度和小数位数。精度是数值中有效数字的总位数,最大位数为 38 位。小数位数则是小数点之后的位数。以下为数值型实例:

- number 表示 number(38), 即最大位数为 38 的整数。
- number (7, 2) 表示小数位数最大为 2, 整数部分最大位数为 5 的数值。7 代表了小数部分和整数部分的总和。
- number(3)表示最大 3 位的整数。

Oracle 并没有定义整型 (integer 或者 int), 但是 Oracle 允许某个字段定义为整型, 这是因为整型是工业标准的强制性要求。但是 Oracle 会自动将 integer 类型转换为 number 类型进行存储。Oracle 推荐所有的数值型都使用 number, 因为使用包括 integer、float 在内的多种数据类型实际上增加了用户在设计数据库时的复杂性。

13.1.3 日期时间型

日期时间型主要用来存储日期和时间格式的数据。Oracle 中最常用的日期型为 date, 该类型中包含了以下信息:

- Century: 世纪信息;
- Year: 年份信息;
- Month: 月份信息;
- Day: 天数信息;
- Hour: 小时信息;
- Minutes: 分钟信息;
- Second: 秒数信息。

在使用时, 为日期型指定固定的格式, 例如, YYYY 代表四位年份; MM 代表月份; DD 代表当月中第几天; HH 代表小时数; MI 代表分钟数; SS 代表秒数。

由于 Oracle 中不区分大小写, 因此 yyyy-mm-dd 等同于 YYYY-MM-DD, 也等同于 yyyy-MM-dd。需要注意的是 date 类型中不仅包含了日期信息, 同时也包含了时间信息。如下实例演示了 date 类型的使用。

```
SQL> create table test_date(test_data date);
```

```
Table created
```

向表 test_data 中插入新的数据, 所插入的数据为当前时间。

```
SQL> insert into test_date values(sysdate);
```

```
1 row inserted
```

查询此时表中数据:

```
SQL> select * from test_date
```

```
2009-6-22 下午 10:33:40
```

除了最常用的 `date` 类型之外, Oracle 还提供了其他日期时间类型, 例如 `timestamp`。 `timestamp` 可以提供精度更高的时间, 它可以将时间精确到毫秒。由于 `date` 以外的日期时间型并不常用, 因此不再一一赘述。

13.1.4 lob 类型

`lob` 类型主要用于存储大对象 (Large Object) 类型。例如, 大量的文本信息 (因为 `varchar2` 最大长度只能达到 4000)、二进制文件等。 `lob` 类型的最大存储容量为 4GB, 数据的存储形式可以为数据库, 也可以是外部数据文件。 `lob` 类型有以下几种具体类型:

- `clob`: 用于存储大型文本数据, 例如, 备注信息。
- `blob`: 用于存储二进制数据, 例如, 图片文件的二进制数据内容。
- `bfile`: 作为单独文件存在的二进制数据。



注意: 在 Oracle 中并不存在布尔类型, 因为存储布尔型是没有任何意义的。如果需要, 可以利用字符串或者利用数值型 (1 或 0) 来代替。



13.2 Oracle 中的特殊数据

4 种基本数据类型保证了 Oracle 可以处理数据存储、变量使用等大部分的工作。另外, Oracle 中存在着一些特殊的数据值得注意。

13.2.1 rowid

`rowid` 是用于标识数据物理地址的列。该列是一个伪列, 它并非用户创建, 而是由数据库自动为表添加, 且只可供数据库内部使用。 `rowid` 的组成通常为 10 个字节。以查看表 `t_employees` 的 `row_id` 及数据表信息为例, 相应代码如范例 13-1 所示。

【范例 13-1】 演示如何查看 `rowid`。

```
SQL> select t.*, t.rowid from t_employees t;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS	ROWID
1	金瑞	6	ACT	AAAMbjAABAAAN/CAAA
2	钟君	6	ACT	AAAMbjAABAAAN/CAAB
3	王山	6	ACT	AAAMbjAABAAAN/CAAC
4	刘迪	5	ACT	AAAMbjAABAAAN/CAAD
5	钟会	4	ACT	AAAMbjAABAAAN/CAAE
6	张玉	4	ACT	AAAMbjAABAAAN/CAAF
7	柳青	4	ACT	AAAMbjAABAAAN/CAAG
8	胡东	4	ACT	AAAMbjAABAAAN/CAAH
9	商乾	4	ACT	AAAMbjAABAAAN/CAAI
10	王蒙	2	ACT	AAAMbjAABAAAN/CAAJ
12	陆逊	0	ACT	AAAMbjAABAAAN/CAAK
13	王俊	0	ACT	AAAMbjAABAAAN/CAAM
11	王武	0	ACT	AAAMbjAABAAAN/CAAP

```
13 rows selected
```



rowid 的前 6 个字符代表数据对象编号, (例如 AAAMbj); 其后的三个字符代表文件编号 (例如 AAB); 接下来的 5 个字符代表块编号 (例如 AAAN/); 最后的 4 个字符代表行的编号 (例如 CAAA、CAAB)。分析表中各记录的 rowid 可知, 表中的 rowid 是唯一的。一般情况下, 可以认为在查询语句中没有指定排序标准时, 将以 rowid 作为默认排序标准。

需要注意的是, rowid 不能作为记录插入数据表的先后标准。rowid 位置靠后的记录不一定是晚插入的记录。因为, Oracle 总是查找空闲的空间进行插入动作。某些记录被删除之后, Oracle 即可释放相应空间, 之后插入的数据有可能存储于删除记录所带来的空闲空间中。其 rowid 将小于现有某些记录的 rowid。

13.2.2 null 与空字符串

Oracle 中将空字符串视为 null。在 Oracle 中判断某列的值是否为空, 不能将该列的值与空字符串"进行比较, 而应该使用 is null。

【范例 13-2】 演示如何测试 null 与空字符串的区别。

```
create table test_data (id varchar2(10), name varchar2(20))
```

【代码说明】 create table test_data 用于创建名为 test_data 的数据表; 该表含有两个 varchar2 类型的列——id 和 name。

现向表 test_data 中插入 4 组数据:

```
begin
  insert into test_data values (1, '张三');
  insert into test_data values (2, '李四');
  insert into test_data values (3, '');
  insert into test_data values (4, null);
  commit;
end;
```

【代码说明】 在这四条数据中, ID 为 3 和 4 的记录的 name 列值分别为空字符串"和 null。

查询表 test_data 的内容:

```
SQL> select * from test_data;
```

ID	NAME
1	张三
2	李四
3	
4	

分析查询结果可知, 对于列值, null 和空字符串是没有区别的, 都显示为空。

现尝试使用比较空字符串的方式来获得记录:

```
SQL> select * from test_data where name = '';
```

ID	NAME
----	------

查询结果表明利用="无法正确判断列值为空的情况。即使在插入时使用了空字符串"。

尝试使用 is null 来获得记录:

```
SQL> select * from test_data where name is null;
```

ID	NAME
3	
4	



分析查询结果可知, 空字符"和 null 值都应该使用 is null 进行比较。

进行相反的比较应该使用 is not null, 表示列值不为空:

```
SQL> select * from test_data where name is not null;
```

ID	NAME
1	张三
2	李四

13.2.3 单引号与双引号

与某些编程语言不同 (例如, Javascript), Oracle 中的单引号与双引号有着截然不同的意义。Oracle 中的单引号用于界定字符串。而双引号则用于标识对象名称。

在 SQL 语句中单引号内为字符串, 即普通数据。例如, select * from test_data where name = '张三' 中的单引号。要想获得单引号的原义字符, 需要使用两个连续单引号。例如, 将单引号的原义字符存储到表 test_data 中的 name 列, 则可以使用如下 SQL 语句:

```
SQL> insert into test_data values(5, ''周军');
```

【代码说明】连续的三个单引号, 第一个为字符串界定符, 第二个为转义符, 第三个是被转义的字。

查询插入后的结果如下:

```
SQL> select * from test_data;
```

ID	NAME
1	张三
2	李四
3	
4	
5	'周军'

当然, 也可以在结尾处插入新的单引号:

```
SQL> update test_data set name = ''周军'' where id=5;
```

【代码说明】在字符串结尾处使用了三个连续的单引号, 其中, 前两个用来转义并获得原义单引号; 第三个为字符串界定符的结尾符。

此时, 查询 id 为 5 的列 name 的值:

```
SQL> select * from test_data where id = 5;
```

ID	NAME
5	'周军'

但是, 执行如下 SQL 语句, Oracle 将会抛出错误:

```
SQL> insert into test_data values (6, '张军');
2 /
```

```
insert into test_data values (6, '张军');
```

```
ORA-01756: quoted string not properly terminated
```

错误提示为: 引号没有正常结束。

Oracle 处理多个单引号的规则为: 当遇到字符串的第一个单引号时, Oracle 将其解析为字符



串界定符的开始符；当遇到第二个单引号时，Oracle 将其解释为界定符的结束符，但是，当此单引号后紧跟第三个单引号时，第二和第三个单引号被解释为原义单引号；依此类推，Oracle 尝试将寻找到的下一个单引号解释为界定符的结束符，但当该单引号后紧跟另一但单引号时，Oracle 将两个单引号一起解释为原义单引号；总之，Oracle 将一直循环此过程，直至找到真正的界定符的结束符。

Oracle 在解释“张军”时，第一个单引号肯定为字符串界定符的开始符，而第二个单引号之后的字符为“张”，因此，第二个单引号被解释为字符串界定符的结束符，而整个字符串将被认作非法字符串。

双引号在 Oracle 中则有着截然不同的意义。其主要作用为标识对象，而这些作用主要表现在两个方面。

1. 用做特殊的列名

Oracle 中数据表的列名具有一定限制，最典型的为列名中不能含有空格。例如，以下修改表列名的 SQL 语句将会抛出错误。

```
SQL> alter table test_data rename column name to user name;

alter table test_data rename column name to user name

ORA-00904: : invalid identifier
```

【代码说明】rename column 选项用于修改列名；name 为原列名，而 user name 为目标列名。Oracle 将抛出非法标识符的错误。这是因为 user name 中含有空格，因此不能作为列名使用。

为了解决此种需求，可以为 user name 添加双引号，以表示 user name 为一个完整的标识符：

```
SQL> alter table test_data rename column name to "user name";

Table altered
```

【代码说明】“user name”表示 user name 为一个完整的标识符，并将该标识符作为新的列名。此时，查询表 test_data 的内容：

```
SQL> select * from test_data;

ID      user name
----      -
1         张三
2         李四
3
4
5         '周军'
```

通过查询结果可知，已经成功将列名修改为 user name。

2. 控制列名的大小写形式

众所周知，一般情况下，Oracle 会自动将所有列名修改为大写形式：

```
SQL> alter table test_data add age number;

Table altered
```

【代码说明】alter table test_data add age number 用于为表 test_data 添加一列；列名为 age，类型为 number。

查看此时 test_data 的内容：

```
SQL> select * from test_data;
```


ID	user name	AGE
1	张三	
2	李四	
3		
4		
5	'周军'	

虽然在 SQL 语句中使用了小写形式,但是 Oracle 执行时,仍然将其转换为大写形式。为了控制该列名为小写形式,可以将列名使用双引号括起来:

```
SQL> alter table test_data rename column age to "age";

Table altered
```

查看此时 test_data 的内容:

```
SQL> select * from test_data;
```

ID	user name	age
1	张三	
2	李四	
3		
4		
5	'周军'	

可见,在使用了双引号之后,Oracle 不会改变列名的大小写形式。

对于使用双引号创建或修改的列名,在引用时,也必须为其添加双引号,例如,直接查询表 test_data 的 age 列,将会出现列不存在的错误提示:

```
SQL> select age from test_data;
```

```
select age from test_data
```

```
ORA-00904: "AGE": invalid identifier
```

只有为其添加双引号之后,才能正常查询:

```
SQL> select "age" from test_data;
```

```
age
-----
```



13.3 本章实例

varchar2 是 Oracle 中最常用的数据类型,其长度的计算方式为字节数,而非字符长度。对于 Oracle 来说,使用字符长度时无法判断用户将要输入的为单字节字符还是双字节字符,因此无法预先划分存储区域给变量或表中的列。

【范例 13-3】用于验证 varchar2 字符串的长度。

(1) 首先创建测试表 data:

```
SQL> create table data(data varchar2(2));
```

```
Table created
```

【代码说明】该代码创建了名为 data 的表。该表含有一个长度为 2 的 varchar2 类型的列 data。

(2) 尝试向表中插入两个英文字符:



```
SQL> insert into data values('hi');
```

```
1 row inserted
```

很明显，向表中插入 2 字节长度的字符串，可以成功操作。

(3) 尝试向表中插入两个中文字符：

```
SQL> insert into data values('你好');
```

```
insert into data values('你好')
```

```
ORA-12899: value too large for column "SYSTEM"."DATA"."DATA" (actual: 4, maximum: 2)
```

可见，由于两个中文字符的长度为 4 个字节，已经超出了列的最大长度 2，导致插入失败。

(4) 对于表中的列，如果类型为 `varchar2`，那么，所允许的最大长度为 4000。例如，尝试将列 `data` 的长度修改为 4001，将会导致修改失败。

```
SQL> alter table data modify (data varchar2(4001));
```

```
alter table data modify (data varchar2(4001))
```

```
ORA-00910: specified length too long for its datatype
```

(5) 将长度由 4001 修改为 4000，再次尝试修改：

```
SQL> alter table data modify (data varchar2(4000));
```

```
Table altered
```

(6) 在 PL/SQL 环境下，如果变量定义为 `varchar2` 类型，那么其最大长度不能超过 32767。

```
SQL> declare data varchar2(32768);
```

```
2 begin
3   data := 'hi';
4 end;
5 /
```

```
declare data varchar2(32768);
```

```
begin
  data := 'hi';
end;
```

```
ORA-06550: line 2, column 23:
```

```
PLS-00215: String length constraints must be in range (1...32767)
```

【代码说明】`declare data varchar2(32768)` 尝试声明一个名为 `data` 的变量，该变量的类型为长 32767 字节的 `varchar2` 类型。此时，PL/SQL 将提示字符串的长度超出了允许范围 (1...32767)。

(7) 在声明变量 `data` 时，将其长度指定为 32767，那么，将可以成功执行。

```
SQL> declare data varchar2(32767);
```

```
2 begin
3   data := 'hi';
4 end;
5 /
```

```
PL/SQL procedure successfully completed
```



13.4 本章小结

本章简要讲述了 Oracle 中的 4 种基本数据类型：字符串型、数值型、日期型和 `lob` 型。尤



其需要注意的是 `varchar2` 类型的理解。对于特殊的数据，需要着重理解的是 Oracle 如何分析和解释多个单引号的字符串。而双引号的应用也不同于其他编程语言，在 Oracle 中，双引号主要用于界定特殊标识符，并将其作为对象名。



13.5 习题

1. Oracle 的数据类型主要有哪几类？
2. 简述 `rowid` 的意义。
3. 简述 Oracle 中空字符串与 `null` 的关系。
4. 简述 Oracle 中双引号的作用。



第 14 章 Oracle 中的函数与表达式

Oracle 中提供了大量的内置函数，以处理各种形式的运算。这些函数涵盖了字符串运算、数值运算、日期运算等方面。同样，Oracle 允许使用数值运算、逻辑运算等基本的表达式运算，另外，提供了 SQL 标准所规定的特殊判式，本章将详细讲述这些函数与运算表达式。本章的主要内容包括：

- Oracle 中的字符串函数；
- Oracle 中的数学函数；
- Oracle 中的日期函数；
- Oracle 中的聚合函数；
- Oracle 中的运算表达式；
- Oracle 中的特殊判式；
- Oracle 中的高级函数——分析函数与窗口函数。

通过本章的学习，读者将掌握 Oracle 中各种常用内置函数，并掌握运算表达式与特殊判式的使用。另外，可以运用分析函数和窗口函数来实现复杂统计功能。



14.1 Oracle 中的字符串函数

Oracle 提供了丰富的字符串函数，本节将通过实例讲述 Oracle 中各字符串函数的使用。

14.1.1 lpad()函数

lpad()函数用于左补全字符串。在某些情况下，预期的字符串为固定长度，而且格式统一，此时可以考虑使用 lpad()函数。例如，深市股票代码都以 0 开头，并且都为 6 位，可以利用 lpad 格式化股票代码，以保证股票代码的格式。

【范例 14-1】演示如何格式化股票代码。

```
SQL> select lpad('21',6,'0') stock_code from dual;
```

```
STOCK_CODE  
-----  
000021
```

【代码说明】lpad('21',6,'0')用于格式化股票代码；21 为原始数据；6 为格式化后的位数；0 为用来填充的字符。该代码将填充字符串'21'为 6 位，并使用 0 填充左边的不足位数。格式化结果为'000021'。

需要注意的是，当原字符串的长度大于预期长度时，实际上进行的是截取字符串操作，代码如下：

```
SQL> select lpad('1234567',6,'0') stock_code from dual;
```

```
STOCK_CODE
```

```
-----
123456
```

【代码说明】在 `lpad('1234567',6,'0')` 中，原字符串长度为 7，而预期长度为 6，`LPAD()` 函数将原字符串从左端开始截取 6 位，返回结果为 123456。

14.1.2 rpad()函数

与 `lpad()` 函数相反，`rpadd()` 函数从右端补齐字符串。

【范例 14-2】演示如何利用 `rpadd()` 函数从右端补齐字符串。

```
SQL> select rpad('abc', 10, '*') from dual;
```

```
RPAD('ABC',10,'*')
```

```
-----
abc*****
```

【代码说明】`rpadd('abc', 10, '*')` 用于从右端补齐字符串 `abc`，预期长度为 10，* 为用来填充的字符。

与 `lpad()` 函数相同，如果预期长度小于原字符串长度，那么，`rpadd()` 函数将会从左端截取字符串，代码如下：

```
SQL> select rpad('abcdefg', 6, '*') from dual;
```

```
RPAD('ABCDEFG',6,'*')
```

```
-----
abcdef
```



注意：`lpad()` 和 `rpadd()` 都用于填充字符串，`lpad()` 从左端进行填充，而 `rpadd()` 从右端进行填充，但是，二者在最终截取字符串时，都是从左端开始截取的。

14.1.3 lower()函数——返回小写字符串

`lower()` 函数用于返回字符串的小写形式。`lower()` 函数在查询语句中经常扮演重要角色。例如，对于用户名和密码的校验来说，用户名一般并不区分大小写，用户无论输入了大写还是小写形式，都被认为是合法用户。因此，在数据库查询时，应该将数据库中用户名与用户输入的用户名进行统一。

【范例 14-3】演示如何使用 `lower()` 函数统一用户名格式。

```
SQL> select user_id, user_name from t_users where lower(user_name) = lower('Alex');
```

```
USER_ID  USER_NAME
```

```
-----
1        Alex
```

【代码说明】`lower(user_name) = lower('Alex')` 用以统一查询条件中的字符串为小写，那么无论用户输入的是“Alex”，还是“alex”，都可以返回正常的记录。

14.1.4 upper()函数——返回大写字符串

`upper()` 函数用于返回字符串的大写形式。与 `lower()` 函数类似，`upper()` 函数也可以用在查询语句中，以统一数据库和查询条件的一致性。范例 14-3 的代码，可以修改为范例 14-4 所示。



【范例 14-4】 演示如何利用 upper() 函数统一用户名格式。

```
SQL> select user_id, user_name from t_users where upper(user_name) = upper('ALEX');
```

```
USER_ID  USER_NAME
-----
1        Alex
```



注意：upper() 函数和 lower() 函数只针对英文字符起作用，因为只有英文字符才有大小写之分。

14.1.5 initcap() 函数——单词首字母大写

initcap() 函数实现单词的首字母大写。

【范例 14-5】 演示如何利用 initcap() 函数实现单词首字母大写。

```
SQL> select initcap('big') from dual;
```

```
INITCAP('BIG')
-----
Big
```

【代码说明】 initcap('big') 将单词 'big' 的首字母转换为大写形式，转换后的结果为 “Big”。

需要注意的是，initcap() 函数不能自动识别单词，例如：

```
SQL> select initcap('bigbigtiger') from dual;
```

```
INITCAP('BIGBIGTIGER')
-----
Bigbigtiger
```

虽然本意是希望返回 BigBigTiger，但是 initcap() 函数并不能自动识别单词，而是将整个字符串当做了个单词，从而返回了 Bigbigtiger。

initcap() 函数会将参数中的非单词字符作为单词分隔符，如下所示：

```
SQL> select initcap('big_big_tiger') from dual;
```

```
INITCAP('BIG_BIG_TIGER')
-----
Big_Big_Tiger
```

```
SQL> select initcap('big/big/tiger') from dual;
```

```
INITCAP('BIG/BIG/TIGER')
-----
Big/Big/Tiger
```

```
SQL> select initcap('big big tiger') from dual;
```

```
INITCAP('BIGBIGTIGER')
-----
Big Big Tiger
```

【代码说明】 在本例中分别使用了 “big_big_tiger”、“big/big/tiger” 和 “big big tiger” 作为 initcap() 函数的参数，而其中的 “_”、“/” 和 “ ” 等非单词字符被视为单词分隔符。

14.1.6 length() 函数——返回字符串长度

length() 函数用于返回字符串的长度。

【范例 14-6】演示如何利用 length() 函数返回字符串长度。

```
SQL> select length('abcd ') from dual;

LENGTH('ABCD')
-----
8
```

【代码说明】length('abcd ') 用于返回字符串 “abcd ” 长度。需要注意的是，空格也是有效字符。length() 返回的长度是包含了空格符的长度。

空字符串的长度不是 0，而是 null。因为空字符串被视为 null，所以，length(null) 返回的仍然是 null。如下代码所示：

```
SQL> select length('') from dual;

LENGTH('')
-----
```

对于其他数据类型，照样可以通过 length() 函数来获得其长度。length() 函数会首先将参数转换为字符串，然后计算其长度，如下代码所示：

```
SQL> select length(12.51) from dual;

LENGTH(12.51)
-----
5
```

length(12.51) 将返回数值型的转换为字符串之后的长度——5。

14.1.7 substr() 函数——截取字符串

substr() 函数用于截取字符串。该函数可以指定截取的起始位置，截取长度，可以实现灵活的截取操作，因此，成为字符串操作中最常用的函数之一。

例如，对于字符串 “1234567890”，现欲截取自第 5 位开始的 4 个字符，相应代码如范例 14-7 所示。

【范例 14-7】演示如何利用 substr() 函数截取字符串。

```
SQL> select substr('1234567890', 5, 4) from dual;

SUBSTR('1234567890',5,4)
-----
5678
```

【代码说明】substr('1234567890', 5, 4) 用于获得字符串 “1234567890” 自第 5 位开始的 4 个字符。需要注意的是，Oracle 中字符位置从 1 开始，而不是像某些编程语言（如 Java）那样从 0 开始。

如果不指定长度，那么 substr() 函数将获取起始位置参数至字符串结尾处的所有字符。

```
SQL> select substr('1234567890', 5) from dual;

SUBSTR('1234567890',5)
-----
567890
```

14.1.8 instr() 函数——获得字符串出现的位置

instr() 函数用于获得子字符串在父字符串中出现的位置。



【范例 14-8】演示如何利用 instr() 函数获得字符串出现的位置。

```
SQL> select instr('big big tiger', 'big') from dual;

INSTR('BIGBIGTIGER','BIG')
-----
1
```

【代码说明】第一个参数，字符串“big big tiger”为被搜索的父字符串；第二个参数，字符串“big”为期望搜索到的子字符串；该函数将返回“big”首次出现的位置——1。

可以指定额外的参数，以命令该函数从指定位置开始搜索，例如：

```
SQL> select instr('big big tiger', 'big', 2) from dual;

INSTR('BIGBIGTIGER','BIG',2)
-----
5
```

【代码说明】参数 2 代表将从被搜索字符串“big big tiger”第 2 个字符开始搜索，当搜索到字符串“big”时，其位置为 5。

还可以指定出现次数参数，以指定是第几次搜索到子字符串：

```
SQL> select instr('big big tiger', 'big', 2, 2) from dual;

INSTR('BIGBIGTIGER','BIG',2,2)
-----
0
```

【代码说明】第 4 个参数 2 代表第 2 次搜索到字符串“big”，因为自“big big tiger”的第二个字符开始，只有一次“big”出现，因此该函数将返回 0，表示不存在符合条件的子字符串“big”。

14.1.9 ltrim() 函数——删除字符串首部空格

ltrim() 中的 l 代表 left。该函数用于删除字符串左端的空白符。

【范例 14-9】演示如何利用 ltrim() 函数删除字符串首部空格。

```
SQL> select ltrim(' abc') from dual;

LTRIM('ABC')
-----
abc
```

需要注意的是，空白符不仅仅包括了空格符，还包括 Tab 键、回车符和换行符。

14.1.10 rtrim() 函数——删除字符串尾部空格

rtrim() 中的 r 代表 right。该函数用于删除字符串右端空白符。删除字符串首尾空白符可以结合使用 ltrim() 和 rtrim() 函数。

【范例 14-10】演示如何利用 rtrim() 结合 ltrim() 删除首尾空格。

```
SQL> select rtrim(ltrim(' abc ')) from dual;

RTRIM(LTRIM('ABC'))
-----
abc
```


14.1.11 trim()函数——删除字符串首尾空格

trim()函数可用于删除首尾空格，相当于 ltrim()和 rtrim()的组合。

【范例 14-11】演示如何利用 trim()删除首尾空格。

```
SQL> select trim(' abc ') from dual;
```

```
TRIM('ABC')
-----
abc
```

14.1.12 to_char()函数——将其他类型转换为字符类型

to_char()函数用于将其他数据类型的数据转换为字符型，这些类型主要包括数值型、日期型。

1. 将数值型转换为字符串

将数值型转换为字符串时，可以通过指定预期格式来实现灵活的转换。转换格式中，可以包括数字 0 和 9、千位分隔符“,”和小数点。

【范例 14-12】演示如何利用 to_char()函数将数值型转换为字符串。

```
SQL> select to_char(120, '99999') result from dual;
```

```
RESULT
-----
120
```

【代码说明】to_char(120, '99999')用于转换数字 120 到“99999”的格式。在数字格式中，“9”代表 0~9 之内的任意数值。Oracle 会尝试将 120 转换为长度为 5 的字符串，当实际位数不足时，将不会进行填充。因此，只返回了实际值“120”。

如果要转换的数值是一个小数，应为其指定小数点及小数点前后的位数。

```
SQL> select to_char(0.96, '9.99') result from dual;
```

```
RESULT
-----
.96
```

【代码说明】分析转换格式可知，当使用 9 来代表数值位时，数值参数中某位为 0，则会忽略此位上的数据。此时，应该使用 0 来代替 9，如下所示：

```
SQL> select to_char(0.96, '0.00') result from dual;
```

```
RESULT
-----
0.96
```

【代码说明】0 用于强制保留位置上的数据，若无对应数据或数据为 0，则进行 0 填充或者保留 0。

千位分隔符也是常用格式之一。在 Oracle 中，默认的千位分隔符为逗号(,)。以下实例演示了如何使用千位分隔符。

```
SQL> select to_char(5897.098, '999,999,999.000') result from dual;
```

```
RESULT
-----
5,897.098
```



【代码说明】预期格式“999,999,999.000”表示，整数部分每三位使用千位分隔符，若无对应数值，则忽略；小数部分为三个 0，用于强制保留三位小数。

可以在格式中指定货币符号，以适应获得货币格式的需求：

```
SQL> select to_char(5987.098, '$999,999,999.000') result from dual;
```

RESULT

\$5,987.098

【代码说明】\$为货币符号，\$在使用时强制添加于字符串的左端。

可以指定中文的货币符号，如下所示：

```
SQL> select to_char(5987.098, 'U999,999,999.000') result from dual;
```

RESULT

¥5,987.098

2. 将日期型转换为字符串

日期格式中含有各个部分（年、月、日）的代码。只要将这些代码进行任意组合，那么就可以获得灵活多变的日期格式。最常见的日期格式莫过于 yyyy-mm-dd 了。

【范例 14-13】演示如何利用 to_char() 将日期型转换为字符串。

```
SQL> select to_char(sysdate, 'yyyy-mm-dd') result from dual;
```

RESULT

2009-06-25

【代码说明】to_char(sysdate, 'yyyy-mm-dd') 将当前日期转换为年-月-日的格式。yyyy 代表四位年份；mm 代表两位的月份；dd 代表两位的日期。Oracle 不区分日期格式大小写，因此，使用 select to_char(sysdate, 'YYYY-MM-DD') result from dual 将获得同样的结果。

表 14-1 包含了 Oracle 中常用的日期格式、格式说明及转换当前日期的结果。

表 14-1 Oracle 日期格式及格式说明

格 式	说 明	转换结果实例
YEAR	获得年份的全拼	two thousand nine
YYYY	四位年份	2009
YYY	年份的后三位	009
YY	年份的后两位	09
Y	年份的最后一位	9
Q	季度	2
MM	两位月份	06
MON	月份的缩写	6 月（与当前系统语言区域有关）
MONTH	月份	6 月（与当前系统语言区域有关）
WW	一年中的第几周	26
W	一月中的第几周	4
D	一周中的第几天	5（星期天为一周的第一天）
DAY	一周中的星期几	星期四（与当前系统语言区域有关）
DD	一月中的第几天	25

续表

格 式	说 明	转换结果实例
DDD	一年中的第几天	176
DY	一周中星期几的缩写	星期四（与当前系统语言区域有关）
HH	某时刻的小时数	11
HH12	某时刻的 12 进制小时数	11
HH24	某时刻的 24 进制小时数	23
MI	某时刻的分钟数	40
SS	某时刻的秒数	30
FF	某时刻的毫秒数	121

表中列举的所有格式，都可以进行任意的组合。只要 Oracle 能够识别其中的格式部分，即可成功转换为字符串。

```
SQL> select to_char(sysdate, 'YYYY-MON-DD') from dual;
```

```
TO_CHAR(SYSDATE, 'YYYY-MON-DD')
```

```
-----
2009-6 月 -25
```

【代码说明】YYYY 为 4 位年份；MON 为月份名称；DD 为日期中的天数。可见，三者的组合生成了字符串 2009-6 月-25。

14.1.13 chr()函数——将 ascii 码转换为字符串

chr()函数用于将 ascii 码转换为字符串。通过 chr()函数，可以对不宜直接输入的字符进行操作。例如，将回车换行符插入到数据中。

【范例 14-14】演示如何利用 chr()函数插入特殊字符。

```
SQL> insert into test_data values (6, '周林'||chr(13)||chr(10)||'梁军', 20);
```

```
1 row inserted
```

【代码说明】chr(13)为回车符，chr(10)为换行符，该 SQL 语句将两个姓名插入到数据库，姓名之间使用了回车换行符。

在插入成功之后，可以查看插入结果：

```
SQL> select * from test_data where id = 6;
```

```
ID          user name          age
-----
6           周林
           梁军                20
```

可见，利用 chr()函数成功向数据库中插入了回车换行符。

14.1.14 translate()函数——替换字符

translate()函数用于替换字符串。替换的规则类似于翻译的过程。

【范例 14-15】演示如何利用 translate()函数替换字符。

```
SQL> select translate('56338', '1234567890', 'avlihemogr') result from dual;
```

```
RESULT
```



```
-----
hello
```

【代码说明】translate('56338','1234567890','avlihemogr')用于依次替换字符串“56338”的每个字符。替换规则为，从第二个参数“1234567890”中查找欲替换的字符，若无法找到，则不执行任何操作，否则记录其位置，并根据该位置在第三个字符串“avlihemogr”中获得对应的字符。该字符即用来替换第一个参数中的字符。在本例中，字符 5 在字符串“1234567890”中的位置为 5，第三个参数“avlihemogr”中位置是 5 的字符为 h，依此类推，最后获得新的字符串“hello”。可见该过程类似于翻译的过程。

需要注意的是，当字符不能被成功“翻译”时，Oracle 将使用空字符替换它。利用此特性，可以使用 translate()函数来删除一个含有数字和英文字母的字符串中的所有字母：

```
SQL> select          translate('21343yuiioioizf899dasiwpe58595oda0j098',
'#abcdefghijklmnopqrstuvwxyz',' ') reulst
from dual;

REULST
-----
21343899585950098
```

【代码说明】#abcdefghijklmnopqrstuvwxyz 以一个非数字，非英文字母的字符开始，并包含了 26 个英文字母，那么在被替换字符中，可以保证所有英文字母均可被查找到。而第三个参数只有一个字符，并且该字符为空格。该空格字符对应“#”，这保证了所有的英文字母均不能成功“翻译”。因此，最终结果返回了所有数字。

需要注意的是，第三个参数不能使用 null，因为 Oracle 的绝大多数函数（包括 translate()）一旦使用了 null 作为函数，那么最终结果也会返回 null，代码如下：

```
SQL> select          translate('21343yuiioioizf899dasiwpe58595oda0j098',
'abcdefghijklmnopqrstuvwxyz','') reulst from dual;

REULST
-----
```



14.2 Oracle 中的数学函数

Oracle 提供的数学函数可以处理日常使用到的大多数数学运算。本节将讲述 Oracle 中常用的几种数学函数。

14.2.1 abs()函数——返回数字的绝对值

abs()函数的参数只能是数值型，该参数用于返回参数的绝对值。

【范例 14-16】演示如何利用 abs()函数返回数字的绝对值。

```
SQL> select abs(-2.1) from dual;

ABS(-2.1)
-----
2.1
```

14.2.2 round()函数——返回数字的“四舍五入”值

round()函数用于返回某个数字的四舍五入值。为了使用该函数，除了提供原始值之外，还应提供精确到的位数。精确位数可以为正整数、0 和负整数。

【范例 14-17】 演示如何使用 round() 函数对数字进行四舍五入处理。

```
SQL> select round(2745.173, 2) result from dual;
```

```
RESULT
-----
2745.17
```

【代码说明】 round(2745.173, 2) 中的参数 2745.173 指定原数据，而参数 2 指定要精确到小数点之后两位。

如果不使用第二个参数，那么，相当于使用了参数 0，即精确到整数。

```
SQL> select round(2745.173) result from dual;
```

```
RESULT
-----
2745
```

```
SQL> select round(2745, 0) result from dual;
```

```
RESULT
-----
2745
```

如果第二个参数为负数，那么，相当于将数值精确到小数点之前的位数。

```
SQL> select round(2745, -1) result from dual;
```

```
RESULT
-----
2750
```

可见，当使用 -1 作为精确位数时，原数值将被精确到 10 位。依此类推，可以利用负数参数实现对数值整数部分的处理。

14.2.3 ceil() 函数——向上取整

ceil() 函数只能有一个参数。该函数将参数向上取整，以获得大于等于该参数的最小整数。

【范例 14-18】 演示 ceil() 函数的使用。

```
SQL> select ceil(21.897) result from dual;
```

```
RESULT
-----
22
```

需要注意的是该函数针对负数的运算：

```
SQL> select ceil(-21.897) result from dual;
```

```
RESULT
-----
-21
```

因为 ceil() 函数返回的是大于等于参数的最小整数，所以，该函数返回的并非 -22，而是 -21。

14.2.4 floor() 函数——向下取整

与 ceil 函数相反，floor() 函数用于返回小于等于某个数值的最大整数。

【范例 14-19】 演示 floor() 函数的使用。

```
SQL> select floor(21.897) result from dual;
```



```
RESULT
-----
21

SQL> select floor(-21.897) result from dual;

RESULT
-----
-22
```

14.2.5 mod ()函数——取模操作

mod()函数有两个参数，第一个参数为被除数，第二个参数为除数。mod()函数的实际功能为获得两数相除之后的余数。

【范例 14-20】 演示 mod()函数的使用。

```
SQL> select mod(5,2) result from dual;

RESULT
-----
1
```

【代码说明】 mod(5,2)的第一个参数 5 为被除数，第二个参数 2 为除数。Mod(5,2)将返回两个参数的余数 1。

与除法规则不同的是，该函数的第二个参数可以为 0。

```
SQL> select mod(5,0) result from dual;

RESULT
-----
5
```

14.2.6 sign()函数——返回数字的正负性

sign()函数只有一个参数。该函数将返回参数的正负性。若返回值为 1，表示该参数大于 0；若返回值为-1，表示该参数小于 0；若返回值为 0，表示该参数等于 0。

【范例 14-21】 演示如何利用 sign()函数返回数字的正负性。

```
SQL> select sign(8) result from dual;

RESULT
-----
1

SQL> select sign(-8) result from dual;

RESULT
-----
-1

SQL> select sign(0) result from dual;

RESULT
-----
0
```

sign()函数为判断两个数值的大小关系提供了方便。因为在 oracle 中，利用类似 if else 的结构来判断两个数值之间的大小关系，并不像编程语言中那样方便，而且极易造成代码的复杂化。

14.2.7 sqrt()函数——返回数字的平方根

sqrt()函数也只有一个参数。该函数用于返回参数的平方根。可以利用 round()函数和 sqrt()函数返回某个数值的近似平方根。

【范例 14-22】演示如何利用 round()函数和 sqrt()函数返回数值的近似平方根。

```
SQL> select round(sqrt(2), 3) result from dual;
```

```
RESULT
```

```
-----  
1.414
```

【代码说明】sqrt(2)用于返回 2 的平方根；round(sqrt(2), 3)将获得的平方根保留三位小数。

14.2.8 power()函数——乘方运算

power()函数有两个参数。该函数用于实现数值的乘方运算。

【范例 14-23】演示如何利用 power()函数实现乘方运算。

```
SQL> select power(6, 2) result from dual;
```

```
RESULT
```

```
-----  
36
```

【代码说明】power(6,2)中，6 为底数，2 为指数；power(6,2)用于获得 6 的 2 次方。

14.2.9 trunc()函数——截取数字

trunc()函数用于截取部分数字。其工作机制非常类似于 round()函数。与 round()函数不同的是，该函数不对数值做四舍五入处理，而是直接截取。

【范例 14-24】演示 trunc()函数的使用。

```
SQL> select trunc(2745.173, 2) result from dual;
```

```
RESULT
```

```
-----  
2745.17
```

【代码说明】trunc(2745.173, 2)中的参数 2745.173 为原始数值；参数 2 为预期保留的位数——小数点后两位。

保留位数的值可以为 0，当该参数的值为 0 时，将保留到整数。

```
SQL> select trunc(2745.173) result from dual;
```

```
RESULT
```

```
-----  
2745
```

当保留位数小于 0 时，表示保留到小数点之前的位数。

```
SQL> select trunc(2745.173, -1) result from dual;
```

```
RESULT
```

```
-----  
2740
```



14.2.10 vsize()函数——返回数据的存储空间

vsize()函数根据数据库的存储格式，来返回其所占用的存储空间的字节数。

【范例 14-25】 演示如何利用 vsize()返回数据所占存储空间。

```
SQL> select vsize('abc123') from dual;
```

```
VSIZE('ABC123')
-----
6
```

```
SQL> select vsize(123) from dual;
```

```
VSIZE(123)
-----
3
```

```
SQL> select vsize(sysdate) from dual;
```

```
VSIZE(SYSDATE)
-----
7
```



注意： vsize()函数返回的是 Oracle 实际存储数据的字节数，在实际开发中使用的几率也较小。读者可以不必了解 Oracle 本身的存储机制。

14.2.11 to_number()函数——将字符串转换为数值类型

to_number()函数可以将字符串转换为数值型。

【范例 14-26】 演示如何将字符串转换为数值。

```
SQL> select to_number('257.90') result from dual;
```

```
RESULT
-----
257.9
```

需要注意的是，被转换的字符串必须符合数值类型格式。如果被转换的字符串不符合数值型格式，Oracle 将抛出错误提示。

```
SQL> select to_number('a') result from dual;
```

```
select to_number('a') result from dual
```

```
ORA-01722: invalid number
```



14.3 Oracle 中的日期函数

Oracle 提供了丰富的日期函数。利用日期函数可以灵活地对日期进行运算。

14.3.1 to_date()函数——将字符串转换为日期型

to_date()函数用于将字符串转换为日期。被转换的字符串必须符合特定的日期格式。

【范例 14-27】 演示如何利用 to_date() 函数将字符串转换为日期型。

```
SQL> select to_date('12/02/09', 'mm/dd/yy') result from dual;
```

```
RESULT
```

```
-----  
2009-12-2
```

14.3.2 add_months() 函数——为日期加上特定月份

add_months() 函数将为日期添加特定月份，并获得新的日期。

【范例 14-28】 演示 add_months() 函数的使用。

```
SQL> select to_char(add_months(sysdate, 2), 'yyyy-mm-dd') result from dual;
```

```
RESULT
```

```
-----  
2009-08-27
```

【代码说明】 add_months(sysdate, 2) 中的第一个参数 sysdate 用于获得当前日期，2 为要添加的月份。

14.3.3 last_day() 函数——返回特定日期所在月的最后一天

last_day() 函数将接受一个日期参数。该函数首先获得日期参数所在月的信息，然后获得该月最后一天的日期。

【范例 14-29】 演示如何利用 last_day() 函数获得特定日期所在月的最后一天。

```
SQL> select to_char(last_day(sysdate), 'yyyy-mm-dd') result from dual;
```

```
RESULT
```

```
-----  
2009-06-30
```

【代码说明】 last_day(sysdate) 用于获得当前日期所在月的最后一天。

可以综合利用 add_months() 函数来获得若干月之后的月份的最后一天。

```
SQL> select to_char(last_day(add_months(sysdate, 3)), 'yyyy-mm-dd') result from dual;
```

```
RESULT
```

```
-----  
2009-09-30
```

14.3.4 months_between() 函数——返回两个日期所差的月数

months_between() 函数用于获取两个日期所间隔的月数。该函数的返回值是一个实数。

【范例 14-30】 演示 months_between() 函数的使用。

```
SQL> select months_between(sysdate, to_date('2009-02-08', 'yyyy-mm-dd')) result from dual;
```

```
RESULT
```

```
-----  
4.62952172
```

【代码说明】 months_between(sysdate, to_date('2009-02-08', 'yyyy-mm-dd')) 用于获得当前日期与 2009-02-08 之间相差的月数。可以看出该函数返回的是一个小数，证明两个参数相差的月数



不是整数。

如果第一个日期早于第二个日期，那么返回值将是负值。

```
SQL> select months_between(to_date('2009-02-08', 'yyyy-mm-dd'), to_date('2009-03-08', 'yyyy-mm-dd')) result from dual;
```

```
RESULT
```

```
-----  
-1
```

14.3.5 current_date()函数——返回当前会话时区的当前日期

current_date()函数用于返回当前会话时区的当前日期。

【范例 14-31】 演示 current_date()函数的使用。

```
SQL> select sessiontimezone, to_char(current_date, 'yyyy-mm-dd hh:mi:ss') result from dual;
```

```
SESSIONTIMEZONE RESULT
```

```
-----  
+08:00          2009-06-27 12:47:46
```

【代码说明】 sessiontimezone 用于获得当前会话的时区；而 current_date 则获得该时区的当前日期。

从查询结果可知，当前会话时区为东八区，当前时间实际为东八区时间。



注意： current_date 等无参数函数作为 Oracle 的关键字存在。在使用时，不能为其添加小括号。即 select current_date() from dual 是错误的 SQL 语句。

14.3.6 current_timestamp()函数——返回当前会话时区的当前时间戳

current_timestamp()函数用于返回当前会话时的区时间戳。可以结合 sessiontimezone 来看其用法。

【范例 14-32】 演示 current_timestamp()函数的使用。

```
SQL> select sessiontimezone, current_timestamp from dual;
```

```
SESSIONTIMEZONE CURRENT_TIMESTAMP
```

```
-----  
+08:00          27-6月 -09 12.51.09.796000 下午 +08:00
```

【代码说明】 sessiontimezone 用于返回当前会话的时区，+08:00 表示当前为东八区；而 current_timestamp 返回当前时区的时间戳。

14.3.7 extract()函数——返回日期的某个域

日期由若干域组成，例如年、月、日、小时等。extract()函数可以返回这些域的具体值。为了使用该函数，除了要指定原日期外，还应该指定要返回的域名。

【范例 14-33】 演示如何使用 extract()函数获得日期的年份。

```
SQL> select extract(year from sysdate) result from dual;
```

RESULT

2009

【代码说明】extract(year from sysdate)用于获得日期的年份部分; year 为域的格式代码; from sysdate 则指定原始日期为当前日期。

需要注意的是, year、month、day 域只能从日期(如 sysdate)中获得, 而 hour、minute、second 只能从时间型(如 systimestamp)中获得。表 14-2 描述了日期中的域名格式及当前日期的对应域的值。

表 14-2 Oracle 日期中的域名

格 式	说 明	转换结果实例
year	获得年份	2009
month	获得月份	6
day	获得天数	27
hour	获得小时数	13
minute	获得分钟数	9
second	获得秒数	2
timezone_hour	获得当前时区的小时数	8



14.4 Oracle 中的聚合函数

所谓聚合函数是指针对多条记录的函数。Oracle 最常用的聚合函数包括, max()、min()、avg()、sum()和 count()函数。本节将讲述这些函数的用法。

本节使用表 t_employees 及表 t_salary 作为测试聚合函数的用例表。二者的结构及数据如下所示。

表 t_employees 的结构及数据内容如下:

```
Select * from t_employees;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	6	ACT
2	钟君	6	ACT
3	王山	6	ACT
4	刘迪	5	ACT
5	钟会	4	ACT
6	张玉	4	ACT
7	柳青	4	ACT
8	胡东	4	ACT
9	商乾	4	ACT
10	王蒙	2	ACT

表 t_salary 的结构及数据内容如下:

```
select * from t_salary
```

SALARY_ID	EMPLOYEE_ID	MONTH	SALARY
1	1	1 月	8000
2	2	1 月	7000
3	3	1 月	7000
4	4	1 月	7000
5	5	1 月	6000



6	6	1月	5500
7	7	1月	5000
8	8	1月	4000
9	9	1月	4000
10	10	1月	3000
11	1	2月	8000
12	2	2月	7000
13	3	2月	7000
14	4	2月	7000
15	5	2月	6000
16	6	2月	5500
17	7	2月	5000
18	8	2月	4000
19	9	2月	4000
20	10	2月	3000
21	1	3月	8000
22	2	3月	7000
23	3	3月	7000
24	4	3月	7000
25	5	3月	6000
26	6	3月	5500
27	7	3月	5000
28	8	3月	4000
29	9	3月	4000
30	10	3月	3000

14.4.1 max()函数——求最大值

max()函数用于获得记录集在某列的最大值。例如，为了返回员工最高工资，可以利用 max() 函数。

【范例 14-34】 演示如何利用 max()函数获得员工的最高工资。

```
SQL> select max(salary) max_salary from t_salary;
```

```
MAX_SALARY
-----
8000
```

【代码说明】 整个查询是针对表 t_salary 的所有记录，max(salary)将返回所有记录中 salary 列的最大值。

需要注意的是，聚合函数往往返回记录集的统计值，因此，不能与其中的单条记录同时出现。例如，不能将 max(salary)与具体列一起查询。

```
SQL> select employee_id, max(salary) max_salary from t_salary;
```

```
select employee_id, max(salary) max_salary from t_salary
```

```
ORA-00937: not a single-group group function
```

【代码说明】 该查询语句将 employee_id 与 max(salary)一起使用，即将列值与聚合函数一起进行查询，Oracle 将抛出 employee_id 并不是分组函数的错误。

为了查询工资最高员工的信息，可以利用如下方法进行查询：

```
SQL> select distinct e.employee_name, s.salary
2 from t_employees e, t_salary s
3 where e.employee_id = s.employee_id and s.salary = (select max(salary) from
```

```
t_salary)
4 /
```

EMPLOYEE_NAME	SALARY
金瑞	8000

【代码说明】因为 `select max(salary) from t_salary` 将返回单行数据，因此，可以用来与 `salary` 列进行比较运算。

14.4.2 min()函数——求最小值

`min()`函数可以用来获得记录集在某列上的最小值，其功能与 `max()`函数相反。

【范例 14-35】演示如何利用 `min()`函数获得工资最低的员工的信息。

```
SQL> select distinct e.employee_name, s.salary
2   from t_employees e, t_salary s
3   where e.employee_id = s.employee_id and s.salary = (select min(salary) from
t_salary)
4 /
```

EMPLOYEE_NAME	SALARY
王蒙	3000

【代码说明】`select min(salary) from t_salary` 将返回工资表中，`salary` 列的最小值。利用该最小值，可以获得表中最低工资的员工信息。

14.4.3 avg()函数——求平均值

`avg()`函数用于获得记录集在某列上的平均值。

【范例 14-36】演示如何利用 `avg()`函数获得第一季度每个员工的平均工资。

```
SQL> select e.employee_name, avg(salary)
2   from t_employees e, t_salary s
3   where e.employee_id = s.employee_id
4   group by e.employee_id, e.employee_name
5 /
```

EMPLOYEE_NAME	AVG(SALARY)
金瑞	8000
钟君	7000
王山	7000
刘迪	7000
钟会	6000
张玉	5500
柳青	5000
胡东	4000
商乾	4000
王蒙	3000

10 rows selected

【代码说明】`group by e.employee_id, e.employee_name` 将所有记录按照员工进行分组，每个分组都被看做一个记录集；`avg(salary)`返回每个结果集的 `salary` 列的平均值。



14.4.4 sum()函数——求和

sum()函数用于获得结果集上某列值的和。

【范例 14-37】演示如何利用 sum()函数来获得每个员工在第一季度获得的工资总额。

```
SQL> select e.employee_name, sum(salary)
  2   from t_employees e, t_salary s
  3   where e.employee_id = s.employee_id
  4   group by e.employee_id, e.employee_name
  5   /
```

EMPLOYEE_NAME	SUM(SALARY)
金瑞	24000
钟君	21000
王山	21000
刘迪	21000
钟会	18000
张玉	16500
柳青	15000
胡东	12000
商乾	12000
王蒙	9000

10 rows selected

【代码说明】与范例 14-36 相同，group by e.employee_id, e.employee_name 将所有记录按照员工进行分组，每个分组都被看做一个记录集；sum(salary)返回每个结果集的 salary 列的总和。

14.4.5 count()函数——获得记录数

count()函数的作用对象同样为记录集。与其他聚合函数不同的是，count()函数可以有三种方式来进行计数：count(*)——计算行数、count(column)——计算某列和 count(1)——累加 1。

为了区别这三种用法，首先为表 t_employees 添加一条记录：

```
SQL> insert into t_employees values (11, null, null, null);

1 row inserted
```

在表 t_employees 中插入新记录，其 employee_id 为 11，其他列均为 null。

(1) 使用 count(*)获得员工数目。

```
SQL> select count(*) from t_employees;
```

```
COUNT(*)
-----
11
```

【代码说明】count(*)计数标准为所有列，可见，其结果为 11。

(2) 利用 count(employee_id)进行计数。

```
SQL> select count(employee_id) from t_employees;
```

```
COUNT(EMPLOYEE_ID)
-----
11
```

【代码说明】count(employee_id)表明，计数标准为列 employee_id 的值，其结果同样为 11。

(3) 利用 count(employee_name)进行计数。

```
SQL> select count(employee_name) from t_employees;
```

```
COUNT(EMPLOYEE_NAME)
-----
10
```

【代码说明】count(employee_name)表明，计数标准为列 employee_name 的值。因为最后一行中的 employee_name 列的值为 null，所以，最终的计数结果为 10。

(4) 利用 count(1)进行计数。

```
SQL> select count(1) from t_employees;
```

```
COUNT(1)
-----
11
```

【代码说明】count(1)是利用常量 1 进行计数。在结果集中，每查询到一条记录，都会累加 1，因此，最终计数结果为 11。

一般来说，利用 count(1)进行计数的速度最快，但是特别注意的是预期的结果是针对整行数据还是某列的数据。



14.5 Oracle 中的其他函数

除了数值函数、字符串函数、日期函数和聚合函数外，Oracle 还提供了其他功能性更强的函数。本节将介绍 decode()、nvl()和 cast()函数。

14.5.1 decode()函数——多值判断

decode()函数用于多值判断。其执行过程类似于解码操作。该函数最常见的应用为，实现类似 if else 的功能。例如，可以利用 decode()函数为员工工资添加标识，工资大于 6000 者为高收入，其余的为一般收入。

【范例 14-38】演示 decode()函数的使用。

```
SQL> select e.employee_id, e.employee_name, decode(sign(avg(s.salary) - 6000),1,
'高收入', '一般收入') incomming
2 from t_employees e, t_salary s
3 where e.employee_id = s.employee_id
4 group by e.employee_id, e.employee_name
5 /
```

EMPLOYEE_ID	EMPLOYEE_NAME	INCOMMING
1	金瑞	高收入
2	钟君	高收入
3	王山	高收入
4	刘迪	高收入
5	钟会	一般收入
6	张玉	一般收入
7	柳青	一般收入
8	胡东	一般收入
9	商乾	一般收入
10	王蒙	一般收入





10 rows selected

【代码说明】`decode(sign(avg(s.salary) - 6000),1, '高收入', '一般收入')`用于判断员工收入状况：`decode()`函数的第一个参数 `sign(avg(s.salary)-6000)`，用于获得员工的平均工资与 6000 元的差值的正负性；第二个参数 1，用于与第一个参数进行比较；当第一个参数与第二个参数的值相等时，`decode()`函数将返回第三个参数的值（高收入），否则，将返回第四个参数的值（一般收入）。

当然，`decode()`函数还实现了 `if elseif else` 的功能。此时，`decode()`的参数个数将发生变化。例如，现欲将员工收入状况分为三个等级：6000 元以上为高收入，6000 元为标准收入，6000 元以下为低收入，那么范例 14-38 所示的代码应该修改为：

```
SQL> select e.employee_id, e.employee_name, decode(sign(avg(s.salary) - 6000),1,
'高收入', 0, '标准收入', '一般收入') incomming
2 from t_employees e, t_salary s
3 where e.employee_id = s.employee_id
4 group by e.employee_id, e.employee_name
5 /
```

EMPLOYEE_ID	EMPLOYEE_NAME	INCOMMING
1	金瑞	高收入
2	钟君	高收入
3	王山	高收入
4	刘迪	高收入
5	钟会	标准
6	张玉	一般收入
7	柳青	一般收入
8	胡东	一般收入
9	商乾	一般收入
10	王蒙	一般收入

10 rows selected

【代码说明】`decode(sign(avg(s.salary) - 6000),1, '高收入', 0, '标准收入', '一般收入')`在原代码的基础上插入了两个新的参数：第四个参数——0、第五个参数——‘标准收入’；当 `sign()`函数返回 1 时，`decode()`函数返回字符串“高收入”；当 `sign()`函数返回 0 时，`decode()`函数将返回“标准收入”；其他情况下，`decode()`函数将返回“一般收入”。

14.5.2 nvl()函数——处理空值

`nvl()`函数用于处理某列的值。该函数有两个参数，第一个参数为要处理的列。如果其值为空，则返回第二个参数的值，否则，将返回列值。

【范例 14-39】演示函数 `nvl()` 的使用。

```
SQL> select employee_id, nvl(employee_name, '未知') employee_name from t_employees;
```

EMPLOYEE_ID	EMPLOYEE_NAME
1	金瑞
2	钟君
3	王山
4	刘迪
5	钟会
6	张玉




```

7      柳青
8      胡东
9      商乾
10     王蒙
11     未知

```

11 rows selected

【代码说明】`nvl(employee_name, '未知')`用于判断列 `employee_name` 的值是否为空，如果为空，则返回字符串“未知”。

`nvl()`函数更常见的用途为判断数值是否为空。因为 `sum()`等函数往往会返回 `null`，例如，表示汇率的列一旦为 `null`，那么最终的货币结算额度也为 `null`，所以，必须对汇率列进行 `nvl()`处理。在统计员工工资时，`null` 同样是不受欢迎的结果，那么可以利用 `nvl()`函数进行处理。

【范例 14-40】演示如何利用 `nvl()`函数处理员工工资。

```

SQL> select e.employee_id, nvl(e.employee_name, '未知') employee_name, nvl(sum
(s.salary), 0) salary
2   from t_employees e, t_salary s
3   where e.employee_id = s.employee_id(+)
4   group by e.employee_id, e.employee_name
5   /

```

EMPLOYEE_ID	EMPLOYEE_NAME	SALARY
1	金瑞	24000
2	钟君	21000
3	王山	21000
4	刘迪	21000
5	钟会	18000
6	张玉	16500
7	柳青	15000
8	胡东	12000
9	商乾	12000
10	王蒙	9000
11	未知	0

11 rows selected

【代码说明】`nvl(sum(s.salary), 0)`用于判断员工工资总额是否为空，如果为空，则返回 0；`e.employee_id = s.employee_id(+)`则是一个外联接，保证了表 `t_employees` 中的每条记录都会出现在查询结果中。

14.5.3 cast()函数——强制转换数据类型

`cast()`函数用于强制转换数据类型。Oracle 会根据操作符来自动进行数据类型的转换，例如：

```
SQL> select '123' + 200 result from dual;
```

```

RESULT
-----
323

```

【代码说明】Oracle 会根据运算符“+”将‘123’转换为数值型 123。

```
SQL> select '123' || 200 result from dual;
```

```

RESULT
-----

```



123200

【代码说明】Oracle 会根据运算符 “||” 将数字 200 转换为字符串 '200'。

```
SQL> select * from t_salary where salary like '%8%';
```

SALARY_ID	EMPLOYEE_ID	MONTH	SALARY
1	1	1 月	8000
11	1	2 月	8000
21	1	3 月	8000

【代码说明】Oracle 会根据运算符 “like” 将列 salary 的值转换为字符串。

cast() 函数最常用的场景是转换列的数据类型，以创建新表。

【范例 14-41】演示如何利用 cast() 函数强制转换列的数据类型。

```
SQL> create table tmp_salary as
2 select cast(salary_id as varchar2(20)) salary_id,
3        cast(employee_id as varchar2(20)) employee_id,
4        cast(month as varchar2(20)) month,
5        cast(salary as varchar2(20)) salary
6 from t_salary
7 /
```

Table created

【代码说明】create table tmp_salary as 用于创建新表 tmp_salary; cast(salary_id as varchar2(20)) salary_id 将表 t_salary 的列 salary_id 转换为可变长度 20 的字符串类型，并作为新列 salary_id; 其他列与此相同。

查看生成的新表 tmp_salary:

```
select * from t_salary
```

SALARY_ID	EMPLOYEE_ID	MONTH	SALARY
1	1	1 月	8000
2	2	1 月	7000
3	3	1 月	7000
4	4	1 月	7000
5	5	1 月	6000
6	6	1 月	5500
7	7	1 月	5000
8	8	1 月	4000
9	9	1 月	4000
10	10	1 月	3000
11	1	2 月	8000
12	2	2 月	7000
13	3	2 月	7000
14	4	2 月	7000
15	5	2 月	6000
16	6	2 月	5500
17	7	2 月	5000
18	8	2 月	4000
19	9	2 月	4000
20	10	2 月	3000
21	1	3 月	8000
22	2	3 月	7000
23	3	3 月	7000
24	4	3 月	7000





25	5	3 月	6000
26	6	3 月	5500
27	7	3 月	5000
28	8	3 月	4000
29	9	3 月	4000
30	10	3 月	3000

新表中的数据与原表相同，各列的数据类型已经转换为 VARCHAR2(20)。

```
SQL> desc tmp_salary;
```

Name	Type	Nullable	Default	Comments
SALARY_ID	VARCHAR2(20)	Y		
EMPLOYEE_ID	VARCHAR2(20)	Y		
MONTH	VARCHAR2(20)	Y		
SALARY	VARCHAR2(20)	Y		



14.6 Oracle 中的运算表达式

Oracle 中的常用运算包括：数学运算、逻辑运算和按位运算。本节将通过范例着重讲述这三种运算的常用运算符和运算规则。

14.6.1 数学运算

数学运算是最常用的运算方式，Oracle 中的数学运算符包括：+、-、*、/，分别代表了加、减、乘、除运算。在使用数学运算时，Oracle 会自动将其他数据类型转换为数值型，然后再参与运算。

【范例 14-42】 演示 4 种数学运算的用法。

```
SQL> select 5+3 result from dual;
```

```
RESULT
```

```
-----
      8
```

```
SQL> select 5-3 result from dual;
```

```
RESULT
```

```
-----
      2
```

```
SQL> select 5*2 result from dual;
```

```
RESULT
```

```
-----
     10
```

```
SQL> select 5/2 result from dual;
```

```
RESULT
```

```
-----
     2.5
```

需要注意的是，任何一种运算符与 null 的运算结果均为 null。

```
SQL> select 5+null result from dual;
```

```
RESULT
```

```
-----
```



```
SQL> select 5-null result from dual;
```

```
RESULT
-----
```

```
SQL> select 5*null result from dual;
```

```
RESULT
-----
```

```
SQL> select 5/null result from dual;
```

```
RESULT
-----
```

因此，对于表查询语句中的复杂的运算，要注意其中间结果是否需要处理空值。

另外，除法运算中的除数不能为 0，否则，Oracle 将抛出错误提示：

```
SQL> select 5/0 result from dual;
```

```
select 5/0 result from dual
```

```
ORA-01476: divisor is equal to zero
```

14.6.2 逻辑运算

Oracle 中的逻辑运算如下。

- >: 大于运算，可用于数值型、日期型和字符串型；
- >=: 大于等于运算，可用于数值型、日期型和字符串型；
- <: 小于运算，可用于数值型、日期型和字符串型；
- <=: 大于等于运算，可用于数值型、日期型和字符串型；
- =: 等于，可用于数值型、日期型和字符串型；
- <>: 不等于，可用于数值型、日期型和字符串型；
- !=: 与<>用法相同；
- NOT: 取反操作；
- AND: 布尔值的与操作；
- OR: 布尔值的或操作。

需要注意的是，Oracle 中的逻辑运算符只能作为条件判断，并不返回值。为了查询工资在 5000~7000 之间的记录，可以利用逻辑运算符来组合查询条件。

【范例 14-43】 演示如何利用逻辑运算符组合查询条件。

```
SQL> select * from t_salary where salary>=5000 and salary<=7000;
```

SALARY_ID	EMPLOYEE_ID	MONTH	SALARY
2	2	1 月	7000
3	3	1 月	7000
4	4	1 月	7000
5	5	1 月	6000
6	6	1 月	5500
7	7	1 月	5000
12	2	2 月	7000
13	3	2 月	7000

14	4	2 月	7000
15	5	2 月	6000
16	6	2 月	5500
17	7	2 月	5000
22	2	3 月	7000
23	3	3 月	7000
24	4	3 月	7000
25	5	3 月	6000
26	6	3 月	5500
27	7	3 月	5000

18 rows selected

【代码说明】`salary>=5000 and salary<=7000` 用于组成生成工资大于等于 5000 元，并且小于等于 7000 元的查询条件。

对于 null 值，需要特别注意的是，无论使用哪种运算符，结果都会返回 null。当比较的结果为 null，并作为条件出现时，Oracle 都会将其解释为 false。

```
SQL> select 1 result from dual where 1=1;
```

```
RESULT
-----
1
```

```
SQL> select 1 result from dual where 1=null;
```

```
RESULT
-----
```

```
SQL> select 1 result from dual where 1<>null;
```

```
RESULT
-----
```

```
SQL> select 1 result from dual where null=null;
```

```
RESULT
-----
```

```
SQL> select 1 result from dual where null<>null;
```

```
RESULT
-----
```

从查询结果可以看出，无论是等于运算 (=)，还是不等于 (<>)，只要逻辑运算中出现了 null，结果都会返回 false。

如同其他编程语言一样，Oracle 中的逻辑运算符 NOT、AND、OR 遵循优先级由高至低的顺序。

14.6.3 位运算

从 Oracle 8i 开始，系统已经提供了位运算符。最常用的莫过于 `bitand` 运算符。

【范例 14-44】演示 `bitand` 运算符的使用。

```
SQL> select bitand(192, 100) result from dual;
```

```
RESULT
-----
```



【代码说明】bitand 将两个参数 192 和 100 首先转换为二进制数据(11000000 和 01100100), 并将各位进行与运算。当两个二进制数字同时为 1 时, 结果为 1, 否则为 0, 运算结果为 01000000。最后转换为十进制数据, 结果为 64。



14.7 Oracle 中的特殊判式

除了逻辑运算之外, Oracle 提供了一些特殊判式。这些判式可以用来生成更加复杂和灵活的查询条件。本节将着重介绍以下几种判式。

- Between: 取值范围。
- In: 集合成员测试。
- Like: 模式匹配。
- is null: 空值判断。
- all, some, any: 数量判断。
- exists: 存在性判断。

14.7.1 between——范围测试

between 判式, 用于判断某个值是否在另外两个值之间。这些值可以为数值型、字符串和日期型。范例 14-45 演示了如何使用 between 判式来获得 ID 号在 1~5 之间的员工信息。

【范例 14-45】演示 between 判式的使用。

```
SQL> select * from t_employees where employee_id between 1 and 5;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	6	ACT
2	钟君	6	ACT
3	王山	6	ACT
4	刘迪	5	ACT
5	钟会	4	ACT

【代码说明】employee_id between 1 and 5 作为查询条件, between 1 and 5 表明 employee_id 的值在 1 到 5 之间。相当于 employee_id>=1 and employee_id<=5。

between 判式同样可以应用于字符串和日期型。字符串是按照字母表的顺序进行比较的, 而日期型是按照日期的先后顺序进行比较的。

```
SQL> select * from t_employees where 'b' between 'b' and 'c';
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	6	ACT
2	钟君	6	ACT
3	王山	6	ACT
4	刘迪	5	ACT
5	钟会	4	ACT
6	张玉	4	ACT
7	柳青	4	ACT
8	胡东	4	ACT
9	商乾	4	ACT
10	王蒙	2	ACT
11			

11 rows selected

【代码说明】'b' between 'b' and 'c' 作为条件判式，用以返回字符串“b”是否处于字符串“b”和“c”之间。按照字母表顺序，该判式永远返回为真，因此，查询结果为表 t_employees 中的所有记录。

将该判式修改，并重新进行查询，代码如下：

```
SQL> select * from t_employees where 'b' between 'bc' and 'c';
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	6	ACT
2	钟君	6	ACT
3	王山	6	ACT
4	刘迪	5	ACT
5	钟会	4	ACT
6	张玉	4	ACT
7	柳青	4	ACT
8	胡东	4	ACT
9	商乾	4	ACT
10	王蒙	2	ACT

【代码说明】按照字母表顺序，字符串“b”处于“bc”之前，因此，该判式的结果为假，查询结果为空。



注意：between 判式与 >=、<= 的组合是等价关系。但是，效率上要比后者差。

14.7.2 in——集合成员测试

in 用于判断某个值是否一个集合的成员。

【范例 14-46】演示 in 判式的使用。

```
SQL> select * from t_employees where status in('NEW', 'ACT');
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	6	ACT
2	钟君	6	ACT
3	王山	6	ACT
4	刘迪	5	ACT
5	钟会	4	ACT
6	张玉	4	ACT
7	柳青	4	ACT
8	胡东	4	ACT
9	商乾	4	ACT
10	王蒙	2	ACT

【代码说明】status in('NEW', 'ACT') 作为条件判式，如果列 status 的值是“NEW”、“ACT”之一，判式结果为真。

值得注意的是，in 判式中的集合的成員的数据类型可以不一致，例如，select * from t_employees where status in('NEW', 'ACT', sysdate, 1) 中的数据类型包含了字符串、日期型和数值型。

14.7.3 like——模式匹配

like 判式的最大特点在于，可以使用通配符。其通常的应用场景为处理模糊查询。

【范例 14-47】演示如何利用 like 判式获得钟姓员工的信息。

```
SQL> select * from t_employees where employee_name like '钟%';
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
2	钟君	6	ACT
5	钟会	4	ACT



【代码说明】like '钟%'表示以字符“钟”开头，其后为任何长度、任何字符组合的字符串。因此，根据该判式将查询出所有钟姓员工的信息。

如果要求字符串中含有原义字符“%”，例如，含有百分比的字符串。那么，like 判式应写为：like '钟\%' escape '\'. Oracle 会首先解释 escape 关键字，并将其后的字符“\”解释为转义字符。那么在“钟\%”中的“%”不再表示通配符，而是表示原义字符“%”。

“_”（下画线）是可用于 like 判式的另一个通配符，该通配符表示一个任意的字符。

14.7.4 is null——空值判断

在逻辑判断中，对于列值为空的判断，不能使用=或者<>。oracle 对与空值的判断提供了专门的判式——is null。例如，为了获取表 t_employees 中员工信息不全的记录，可以利用范例 14-48 所示的查询语句。

【范例 14-48】演示 is null 判式的使用。

```
SQL> select * from t_employees
2  where employee_id is null
3      or employee_name is null
4      or work_years is null
5      or status is null;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
11			

【代码说明】employee_id is null 用于判断列 employee_id 的值是否为空；其他判式与之相同。

14.7.5 exists——存在性判断

in 判式用于判断表的列值是否存在于列表（集合）中。而 exists 判式则可用于判断查询结果集合是否为空。例如，为了查询出表 t_employees 所存储的员工信息中，哪些员工存在于工资表中，即可利用 exists 判式。

【范例 14-49】演示如何利用 exists 判式判断记录的存在性。

```
SQL> select * from t_employees e
2  where exists(select * from t_salary where employee_id = e.employee_id);
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	6	ACT
2	钟君	6	ACT
3	王山	6	ACT
4	刘迪	5	ACT
5	钟会	4	ACT
6	张玉	4	ACT
7	柳青	4	ACT
8	胡东	4	ACT
9	商乾	4	ACT
10	王蒙	2	ACT

【代码说明】exists(select * from t_salary where employee_id = e.employee_id) 针对 t_employees 中的每条记录的 employee_id，将在表 t_salary 中查找 employee_id 列值与之相等的记录，如果找到，将返回真，否则返回假。

14.7.6 all, some, any——数量判断

all, some 和 any 判式的作用对象为记录集合。all 表示, 记录集中的所有记录, some 表示其中的一些记录, any 判式则表示其中的任意记录。例如, 在员工工资表 t_salary 中, 为了查找高于 id 为 4 和 5 的工资信息, 即可使用 all 判式。

【范例 14-50】演示 all 判式的使用。

首先查看表 t_salary 中员工 id 为 4 和 5 的员工的工资信息:

```
SQL> select * from t_salary where employee_id = 4 or employee_id = 5;
```

SALARY_ID	EMPLOYEE_ID	MONTH	SALARY
4	3	1 月	7000
5	4	1 月	6000
14	3	2 月	7000
15	4	2 月	6000
24	3	3 月	7000
25	4	3 月	6000

6 rows selected

然后利用 all 判式进行查询:

```
SQL> select * from t_salary where salary > all(select distinct salary from t_salary
where employee_id = 4 or employee_id = 5);
```

SALARY_ID	EMPLOYEE_ID	MONTH	SALARY
1	1	1 月	8000
11	1	2 月	8000
21	1	3 月	8000

【代码说明】在这里, all 判式等效于 and 运算, 因为子查询使用了 distinct 关键字, 所以, 此处的查询条件相当于 salary>6000 and salary>7000。

some 和 any 判式是等价的, 比如:

```
SQL> select * from t_salary where salary > some(select distinct salary from t_salary
where employee_id = 4 or employee_id = 5);
```

SALARY_ID	EMPLOYEE_ID	MONTH	SALARY
1	1	1 月	8000
11	1	2 月	8000
21	1	3 月	8000
2	2	1 月	7000
12	2	2 月	7000
22	2	3 月	7000
24	4	3 月	7000
23	3	3 月	7000
13	3	2 月	7000
3	3	1 月	7000
4	4	1 月	7000
14	4	2 月	7000

12 rows selected

【代码说明】此时的 some 判式实际上相当于逻辑运算中的 or 运算, 即 salary>6000 or



salary>7000。此时，使用 any 判式，将返回同样的结果。



14.8 Oracle 高级函数——分析函数与窗口函数

Oracle 中的分析函数具有非常强大的功能。分析函数往往与另一类函数——窗口函数同时使用。窗口函数总是为查询过程中的当前记录提供一个相关记录集，而且随着当前记录的推移，相应的记录集也会随之改变，这非常类似于“滑动窗”的概念。分析函数的操作对象即为“滑动窗”所指定的记录集合。本节将通过实例来讲述分析函数和窗口函数的使用。

14.8.1 排名

分析函数中的排名函数可以针对窗口中的记录生成排序序号。常用的排名函数有 rank()、dense_rank()和 row_number()。

rank()函数用于返回当前记录在窗口函数所指定的记录集中的排名。rank()函数在排名过程中，具有跳跃的特点。

【范例 14-51】 演示 rank()函数的使用。

用于测试的表 students 中的数据如下：

```
SQL> select * from students;
```

STUDENT_ID	STUDENT_NAME	STUDENT_AGE
1	金瑞	19
2	钟君	19
3	王山	19
4	刘迪	19
5	钟会	19
6	张玉	19
7	柳青	18
8	胡东	18
9	商乾	17
11	王蒙	19

10 rows selected

在表 students 中存储了学生的基本信息，包括了学生姓名及年龄。现欲对学生年龄按照由小到大的顺序进行排列，并返回各个学生在排序中的位置，则可以使用分析函数 rank()和窗口函数 over()。

```
SQL> select student_name, rank() over (order by student_age) position from students;
```

STUDENT_NAME	POSITION
商乾	1
柳青	2
胡东	2
金瑞	4
钟君	4
王蒙	4
王山	4
刘迪	4
钟会	4

张玉 4

10 rows selected

【代码说明】`over(order by student_age)`用于指定当前记录的窗口——首先对表 `students` 中所有记录进行排序，然后选定自第 1 条记录至当前排名的所有记录；`rank()`函数则用于返回当前记录在窗口中的排序序号。

分析查询结果可知，对于排名相同的记录，`rank()`函数返回相同的排序序号；当出现多个排名相同的记录时，下一排名序号，将根据前一排名个数进行跳跃。

相对于 `rank()`函数，`dense_rank()`函数所获得的排名不会出现跳跃。

【范例 14-52】演示 `dense_rank()`函数的使用。

```
SQL> select student_name, dense_rank() over(order by student_age) position from students;
```

STUDENT_NAME	POSITION
商乾	1
柳青	2
胡东	2
金瑞	3
钟君	3
王蒙	3
王山	3
刘迪	3
钟会	3
张玉	3

10 rows selected

值得注意的是，`rank()`函数和 `dense_rank()`函数对于排名相同的多条记录，返回相同的排名序号，而函数 `row_number()`函数每条记录都会存在唯一的排名序号。

【范例 14-53】演示 `row_number()`函数的使用。

```
SQL> select student_name, row_number() over(order by student_age) position from students;
```

STUDENT_NAME	POSITION
商乾	1
柳青	2
胡东	3
金瑞	4
钟君	5
王蒙	6
王山	7
刘迪	8
钟会	9
张玉	10

10 rows selected

分析查询结果可知，`row_numbe()`函数每次均返回唯一的排序序号。



14.8.2 分区窗口

对于窗口函数，利用 `partition by` 关键字可以指定分区窗口。

【范例 14-54】 演示如何利用 `partition by` 进行分区。

在表 `salary` 中存储了员工姓名、所属部门及工资的基本信息，如下所示：

```
SQL> select * from salary;
```

EMPLOYEE_ID	EMPLOYEE_NAME	DEPARTMENT	SALARY
1	张小琴	开发部	5000
2	刘明	开发部	5500
3	周欣欣	开发部	4500
4	李周宇	工程部	3000
5	刘金龙	工程部	3500
6	武铨	工程部	3800
7	陆军	工程部	2800
8	张伦	工程部	3200
9	王静	工程部	3000
10	张敏	人事部	2300
11	李云	人事部	3500
12	张辉	财务部	2500
13	刘军	财务部	3000

13 rows selected

现欲统计各员工的工资在各自部门的高低情况，则可以利用 `partition by` 进行分区，然后利用分析函数对分区内的记录进行统计，代码如下：

```
SQL> select t.*, dense_rank() over(partition by department order by salary) position
2 from salary t
3 order by t.employee_id
4 /
```

EMPLOYEE_ID	EMPLOYEE_NAME	DEPARTMENT	SALARY	POSITION
1	张小琴	开发部	5000	2
2	刘明	开发部	5500	1
3	周欣欣	开发部	4500	3
4	李周宇	工程部	3000	4
5	刘金龙	工程部	3500	2
6	武铨	工程部	3800	1
7	陆军	工程部	2800	5
8	张伦	工程部	3200	3
9	王静	工程部	3000	4
10	张敏	人事部	2300	2
11	李云	人事部	3500	1
12	张辉	财务部	2500	2
13	刘军	财务部	3000	1

13 rows selected

【代码说明】`over(partition by department order by salary)` 为当前记录获得分区窗口。分区列为 `department`；当前记录所在的分区被指定为当前窗口；`order by salary` 用于将窗口中的记录按照 `salary` 进行排序；`dense_rank()` 则获得当前记录在分区窗口中，经过排序之后的位置。

通过查询结果可知，员工张小琴的工资在本部门中处于第 2 位，而员工刘明的工资处于第

1 位、周欣欣处于第 3 位。

另外一种常见需求为，在获得员工工资的同时，也需要部门所有员工的工资总额，相应的 SQL 语句如下：

```
SQL> select t.*,
2 sum(salary) over(partition by department) total_salary,
3 round(avg(salary) over(partition by department)) average_salary
4 from salary t
5 order by employee_id
6 /
```

EMPLOYEE_ID	EMPLOYEE_NAME	DEPARTMENT	SALARY	TOTAL_SALARY	AVERAGE_SALARY
1	张小琴	开发部	5000	15000	5000
2	刘明	开发部	5500	15000	5000
3	周欣欣	开发部	4500	15000	5000
4	李周宇	工程部	3000	19300	3217
5	刘金龙	工程部	3500	19300	3217
6	武铨	工程部	3800	19300	3217
7	陆军	工程部	2800	19300	3217
8	张伦	工程部	3200	19300	3217
9	王静	工程部	3000	19300	3217
10	张敏	人事部	2300	5800	2900
11	李云	人事部	3500	5800	2900
12	张辉	财务部	2500	5500	2750
13	刘军	财务部	3000	5500	2750

13 rows selected

【代码说明】sum(salary)对窗口函数所获得的记录集进行求和操作；avg(salary)对窗口函数所获得的记录集进行求平均值操作。



注意：avg(salary) over(partition by department)是不可分割的一个整体。对于数据表 salary 中每条记录都会返回单个值，因此，当使用 round()函数时，函数的作用对象应为 avg(salary) over(partition by department) 这个整体，而不能使用诸如 round(avg(salary)) over(partition by department)等形式。另外，利用 partition by 进行分区之后，当前记录总是处于某个分区中，此时的窗口即为该分区。

14.8.3 窗口子句

对于每条记录，一旦使用了窗口函数，都会为其产生一个可操作的记录集合。而对于该记录集，可以使用窗口子句，来进一步限制窗口范围。常用的窗口子句包括两类：

- 利用 rows 子句的行方式进行限制；
- 利用 range 子句的值方式进行限制。

1. rows 子句

rows 子句的原理在于，在已确定的窗口中，各记录已经按照某种标准进行了排序；而当前记录处于窗口中的特定位置；rows 子句以当前记录为参照物，可以向前、向后推移，并形成新的结果集，作为最终的操作窗口。在表 salary 中，在获得员工信息的同时，获得相邻三个员工（前一员工、当前员工、后一员工）的工资总和，即可利用 rows 子句实现该窗口。

**【范例 14-55】**演示如何利用 rows 子句进一步限制窗口大小。

```
SQL> select employee_id, employee_name,
       2 sum(salary) over (order by employee_id rows between 1 preceding and 1 following)
three_total
       3 from salary
       4 /
```

EMPLOYEE_ID	EMPLOYEE_NAME	THREE_TOTAL
1	张小琴	10500
2	刘明	15000
3	周欣欣	13000
4	李周宇	11000
5	刘金龙	10300
6	武铉	10100
7	陆军	9800
8	张伦	9000
9	王静	8500
10	张敏	8800
11	李云	8300
12	张辉	9000
13	刘军	5500

13 rows selected

【代码说明】order by employee_id 将当前窗口中的记录按照 employee_id 进行升序排列。因为没有 partition by 等其他限制，此时的窗口为排序之后，表 salary 中第一条记录至当前排名的所有记录；rows between 1 preceding and 1 following 进一步限制窗口大小——当前记录的前一条记录和当前记录的下一条记录之间的所有记录；sum(salary)则返回窗口中所有记录的 salary 列值的和。



注意：rows between 1 preceding and 1 following 不一定返回 3 条记录。例如，对于 employee_id 为 1 的记录，排序之后，该记录为第一条记录，不存在前一条记录，因此只返回两条记录，而求和操作返回的实际上为 employee_id 为 1 和 2 的员工的工资总和 10500。

rows 子句因为和位置相关，因此，在窗口函数中必须含有排序子句 order by。如果未使用 order by 子句，而直接使用 rows 子句，Oracle 将抛出错误提示，如下所示：

```
SQL> select employee_id, employee_name,
       2 sum(salary) over (rows between 1 preceding and 1 following) three_total
       3 from salary
       4 order by employee_id
       5 /

select employee_id, employee_name,
sum(salary) over (rows between 1 preceding and 1 following) three_total
from salary
order by employee_id
```

ORA-30485: missing ORDER BY expression in the window specification

2. range 子句

range 子句按照列值进行窗口的进一步限制。

【范例 14-56】演示如何利用 range 子句限制窗口大小。

```
SQL> select employee_id, employee_name,
2 count(1) over(partition by department order by salary range between 500 preceding
and 500 following)
3 as employee_count
3 from salary
4 order by employee_id
5 /
```

EMPLOYEE_ID	EMPLOYEE_NAME	EMPLOYEE_COUNT
1	张小琴	3
2	刘明	2
3	周欣欣	2
4	李周宇	5
5	刘金龙	5
6	武铨	2
7	陆军	4
8	张伦	5
9	王静	5
10	张敏	1
11	李云	1
12	张辉	2
13	刘军	2

13 rows selected

【代码说明】partition by department 首先为当前记录创建分区——与当前记录具有相同 department 值的所有记录；order by salary 对分区中的所有记录按照 salary 列值进行升序排列；range between 500 preceding and 500 following 以当前记录的 salary 值为参照物 x，salary 的值大于等于 x-500、且小于等于 x+500 的记录被筛选为最终窗口。

以 employee_id 为 2 的员工为例，该员工隶属于开发部，那么利用 partition by department 分区之后，窗口中含有三条记录：

EMPLOYEE_ID	EMPLOYEE_NAME	DEPARTMENT	SALARY
1	张小琴	开发部	5000
2	刘明	开发部	5500
3	周欣欣	开发部	4500

利用 order by salary 进行排序之后，窗口记录形式如下所示：

EMPLOYEE_ID	EMPLOYEE_NAME	DEPARTMENT	SALARY
3	周欣欣	开发部	4500
1	张小琴	开发部	5000
2	刘明	开发部	5500

此时，salary 的值为 5500，当使用 range between 500 preceding and 500 following 限制之后，将筛选 salary 处于 5000 (5500-500) 与 6000 (5500+500) 之间的记录作为最终窗口。针对该窗口，count(1)将返回 2。

3. unbounded 和 current row

在 rows 和 range 子句中，除了使用具体的数值来决定窗口的大小之外，还可以使用关键字 unbounded 和 current row。unbounded 可以直接代替数值，表示没有任何限制；current row 则直



接代表当前行。对于 `order by` 子句，如果没有显式指定 `rows` 和 `range` 子句，那么相当于使用了 `rows between unbounded preceding and current row` 或者 `range between unbounded preceding and current row`。但是，当排序后存在相同排名的记录时，情况会有些差异。。

【范例 14-57】 演示使用 `order by` 子句的窗口变化情况。

```
SQL> select student_name,  
2 count(1) over (order by student_age) count  
3 from students;
```

STUDENT_NAME	COUNT
商乾	1
柳青	3
胡东	3
金瑞	10
钟君	10
王蒙	10
王山	10
刘迪	10
钟会	10
张玉	10

10 rows selected

【代码说明】 `order by student_age` 用于将窗口中的记录，按照列 `student_age` 的值进行升序排列；`count(1)` 则用于统计窗口中的记录数。

可见，随着记录的推移，窗口中的记录数为排序的第一条记录至当前排名位置的总记录数。如果排名相同的记录有多条，那么所有相同记录都将被筛选到窗口中。

但是 `current row` 总是针对某条特定记录，排序之后的记录，虽然存在着相同的序号，但是 Oracle 总会将这些记录进行区分。

```
SQL> select student_name,  
2 count(1) over (order by student_age rows between unbounded preceding and current  
row) count  
3 from students;
```

STUDENT_NAME	COUNT
商乾	1
柳青	2
胡东	3
金瑞	4
钟君	5
王蒙	6
王山	7
刘迪	8
钟会	9
张玉	10

10 rows selected

因此，只使用了 `order by` 子句所指定的窗口为第一条记录至当前排名的所有记录，相较于 `rows between unbounded preceding and current row`，前者更为准确。



14.8.4 主要的分析函数

分析函数作用对象为窗口函数所捕获的记录集，因此，分析函数具有聚合函数的特点，大多数的聚合函数，如 `sum()`、`count()`、`max()` 等都能作为分析函数出现。Oracle 还提供了专门针对窗口函数的分析函数，本节着重讲述常用的几种——`first_value()`、`last_value()`、`lag()` 和 `lead()`。

1. `first_value()` 函数的使用

`first_value()` 函数需要一个表达式参数，即 `first_value(expression)`。该函数用于获得窗口函数所捕获的记录集中第一条记录，并根据第一条记录返回表达式参数的值。例如，当表达式参数为列名时，那么 `first_value()` 函数将返回该记录的列值。

在 `salary` 表中，需要获得每个部门中工资最低员工的信息。按照传统方法，必须首先对表 `salary` 按照 `department` 列进行分组，并获得各组中的最小 `salary` 值；然后根据 `department` 和 `salary` 的信息获得员工信息。

```
SQL> select s.department, s.employee_name, s.salary from (
2  select department, min(salary) min_salary from salary
3  group by department ) t1
4  left join salary s
5  on t1.department = s.department
6  and t1.min_salary = s.salary
7  /
```

DEPARTMENT	EMPLOYEE_NAME	SALARY
财务部	张辉	2500
工程部	陆军	2800
开发部	周欣欣	4500
人事部	张敏	2300

【代码说明】`select department, min(salary) min_salary from salary group by department` 用于将所有记录按 `department` 进行分组，获得每个组的 `department` 及工资最小值，并生成内嵌视图 `t1`；`left join salary s on t1.department = s.department and t1.min_salary = s.salary` 将获得内嵌视图与表 `department` 进行外联接，以获得每个 `department` 和最小 `salary` 所对应的员工信息。

另一种解决方案为使用 `first_value()` 函数。

【范例 14-58】演示如何利用 `first_value()` 函数获得最低工资员工信息。

```
SQL> select distinct department,
2  first_value(employee_name) over(partition by department order by salary)
employee_name,
3  first_value(salary) over(partition by department order by salary) salary
4  from salary
5  /
```

DEPARTMENT	EMPLOYEE_NAME	SALARY
财务部	张辉	2500
工程部	陆军	2800
开发部	周欣欣	4500
人事部	张敏	2300

【代码说明】`over(partition by department order by salary)` 对表 `salary` 中的数据按 `department` 列进行分区，并在分区内部按 `salary` 列进行升序排列；`first_value(employee_name)` 用于获得窗口中的第一条记录的 `employee_name` 列的值；`first_value(salary)` 用于获得窗口中的第一条记录



的 salary 列的值; distinct 关键字是必要的, 对于每个部门中的每条记录都会返回相同的信息——部门名称、部门最低工资和部门最低工资员工。



注意: 两段代码实际还是有区别的, 利用 left join 的方法, 可以获得本部门工资最低的所有员工。因为处于最低工资标准的员工有可能多于一个, 使用 first_value() 函数则仅获得其中一个样本。

2. last_value()函数的使用

last_value()函数与 first_value()函数的用法相同, 不同的是, 该函数将返回窗口中最后一条记录的相关信息。在范例 14-58 中, 如果使用了 last_value()函数, 则可以获得部门工资最高的员工信息, 如范例 14-59 所示。

【范例 14-59】 演示 last_value()函数的使用。

```
SQL> select distinct department,
2 last_value(employee_name) over(partition by department order by salary
3 rows between unbounded preceding
4 and unbounded following) employee_name,
5 last_value(salary) over(partition by department order by salary
6 rows between unbounded preceding
7 and unbounded following) salary
8 from salary
9 /
```

DEPARTMENT	EMPLOYEE_NAME	SALARY
财务部	刘军	3000
工程部	武铨	3800
开发部	刘明	5500
人事部	李云	3500

【代码说明】 partition by department order by salary rows between unbounded preceding and unbounded following 用于将表 salary 中的所有记录以 department 分区, 并按照 salary 进行排序; last_value(salary)用于获取窗口中工资最大值。



注意: 这里使用 rows between unbounded preceding and unbounded following 是必要的。这是因为, 默认情况下, order by 的窗口为第一条记录至当前记录, 所以必须选定所有记录, 才能获得整个部门的最后一位员工的信息。

3. lead()函数的使用

lead()函数也可以定位一条记录。对于通过排序之后的窗口集合, lead()函数自当前记录向下推移, 获得新的记录。lead()函数的完整形式为 lead(expression, offset, defaultvalue)。其中, expression 是一个表达式, 该表达式根据定位的记录返回值; offset 是一个整数, 表示自当前记录向下的偏移量; defaultvalue 表示, 当无法获取新的记录时, 用于代替 expression 作为 lead()函数的返回值。例如, 对于表 salary, 可以在部门内部按工资进行排序, 并获得排序之后每位员工的下一员工信息, 如范例 14-60 所示。

【范例 14-60】 演示 lead()函数的使用。

```
SQL> select employee_id, employee_name, salary,
2 lead(employee_name, 1, 'N/A')
```

```

3 over (partition by department order by salary) prev_name
4 from salary
5 order by employee_id
6 /

```

EMPLOYEE_ID	EMPLOYEE_NAME	SALARY	PREV_NAME
1	张小琴	5000	刘明
2	刘明	5500	N/A
3	周欣欣	4500	张小琴
4	李周宇	3000	王静
5	刘金龙	3500	武铨
6	武铨	3800	N/A
7	陆军	2800	李周宇
8	张伦	3200	刘金龙
9	王静	3000	张伦
10	张敏	2300	李云
11	李云	3500	N/A
12	张辉	2500	刘军
13	刘军	3000	N/A

13 rows selected

【代码说明】partition by department order by salary 将表 salary 中的记录以 department 进行分区，并在分区内部按照 salary 排序；lead(employee_name, 1, 'N/A')自当前记录开始，向下推移 1，并定位新的记录，以获得当前员工的下一条记录的员工姓名；如果该记录不存在，则使用 N/A 来代替。

分析查询结果可知，刘明工资高于员工张小琴，而在同一部门中，没有员工的工资高于刘明。

4. lag()函数的使用

lag()函数与 lead()函数具有完全相同的语法规则。在排序之后，lag()函数自当前记录向上搜索。范例 14-61 演示了使用 lag()函数来获得在同部门中，按 salary 排序之后，每位员工的前一位员工的信息。

【范例 14-61】演示 lag()函数的使用。

```

SQL> select employee_id, employee_name, salary,
2 lag(employee_name, 1, 'N/A')
3 over (partition by department order by salary) prev_name
4 from salary
5 order by employee_id
6 /

```

EMPLOYEE_ID	EMPLOYEE_NAME	SALARY	PREV_NAME
1	张小琴	5000	周欣欣
2	刘明	5500	张小琴
3	周欣欣	4500	N/A
4	李周宇	3000	陆军
5	刘金龙	3500	张伦
6	武铨	3800	刘金龙
7	陆军	2800	N/A
8	张伦	3200	王静
9	王静	3000	李周宇
10	张敏	2300	N/A



11	李云	3500	张敏
12	张辉	2500	N/A
13	刘军	3000	张辉

13 rows selected

【代码说明】partition by department order by salary 将表 salary 中的记录以 department 进行分区，并在分区内部按照 salary 排序；lag(employee_name, 1, 'N/A')自当前记录开始，向上推移 1，并定位新的记录，以获得当前员工的上一条记录的员工姓名；如果该记录不存在，则使用 N/A 来代替。



14.9 本章实例

Oracle 虽然内置了很多函数，但是并不能满足日常开发中的应用。此时，需要开发者自定义函数，而内置函数往往成为自定义函数的基石。

【范例 14-62】演示如何利用 to_date()函数创建自定义函数 is_date()。

```
SQL> create or replace function is_date (param varchar2)
2  return varchar2
3  is
4  val date;
5  begin
6    val := to_date (nvl (param, ' '), 'yyyy-mm-dd hh24:mi:ss');
7    return 'Y';
8  exception
9  when others then
10   return 'N';
11 end;
12 /
```

Function created

【代码说明】nvl (param, ' ')用于处理传入参数为非空的情况，如果传入参数为空，则使用空格符进行处理；to_date (nvl (param, ' '), 'yyyy-mm-dd hh24:mi:ss')尝试将处理之后的参数转换为日期时间型；return 'Y'在实现正常转换之后，将返回字符“Y”；exception 用于捕获异常；return 'N'当捕获异常之后，将返回字符“N”。

在该自定义函数 is_date()中，实际用到了两个内置函数——nvl()和 to_date()函数。当所传入的字符串符合正常的日期格式时，将返回字符“Y”；否则，将返回“N”。

在 PL/SQL Developer 中进行测试：

```
SQL> select is_date('abc') from dual;
```

```
IS_DATE('ABC')
```

```
N
```

```
SQL> select is_date('') from dual;
```

```
IS_DATE('')
```

```
N
```

```
SQL> select is_date('2009-09-01') from dual;
```

```
IS_DATE('2009-09-01')
```



Y

```
SQL> select is_date('2009-09-01 12:30:00') from dual;
```

```
IS_DATE('2009-09-0112:30:00')
```

Y



14.10 本章小结

本章通过详尽的实例讲述了 Oracle 中常用的内置函数、表达式及特殊判式。对于内置函数，特别需要注意的是聚合函数的使用。聚合函数最常用的场景为分组查询；聚合函数不能与单条记录的列并列作为查询结果。对于特殊判式，尤其应该注意的是 like 判式的使用，like 判式使用中，通配符只有“%”和“_”两种，要注意通配符和正则表达式的区别。

在 Oracle 高级函数中，重点介绍了分析函数和窗口函数，这两个函数总是结合使用，为数据表中单条记录提供新的结果集的方法。对于窗口函数，要重点理解分区和排序的工作流程，尤其需要注意的是，对于排序中，具有相同排名的记录的处理。相较之下，分析函数非常类似于聚合函数，比较容易理解和掌握。在统计和生成复杂报表时，分析函数和窗口函数有着广泛的应用，尤其对于复杂统计，利用这两种函数往往可以起到事半功倍的效果。



14.11 习题

1. 简述 null 作为函数参数时的特点。
2. 简述 like 判式的使用方法。
3. 简述 is null 判式的意义。
4. 简述窗口函数的特点。



第 15 章 Oracle 中的控制语句

所有编程语言都离不开控制语句，Oracle 也为 PL/SQL 编程提供了基本的控制语句。本章将针对最主要的两种控制语句：条件控制语句和循环语句进行讲解。本章的内容如下。

- 条件语句：if else;
- 条件语句：case when;
- 循环语句：无条件循环;
- 循环语句：while 循环;
- 循环语句：for 循环。

通过本章的学习，读者将掌握 Oracle 中的各种控制语句，并扩展 PL/SQL 编程的思路。



15.1 Oracle 中的条件语句

条件语句是指通过条件判断改变程序流程的语句。Oracle 中提供了两种主要的条件语句：if else 语句和 case 语句。

15.1.1 利用 if else 进行条件判断

if else 语句是编程语言中最常见的条件控制语句。if 关键字之后需要紧跟条件表达式，当该条件表达式返回为真时，将执行其后的语句，否则，将执行 else 之后的语句。

【范例 15-1】演示如何使用 if else 语句。

```
SQL> set serverout on;
```

```
SQL> declare employee_num number;  
2 begin  
3   select count(*) into employee_num from t_employees where status = 'ACT';  
4   if employee_num>0 then  
5     dbms_output.put_line('表 T_EMPLOYEES 中存在记录');  
6   else  
7     dbms_output.put_line('表 T_EMPLOYEES 中不存在记录');  
8   end if;  
9 end;  
10 /
```

表 T_EMPLOYEES 中存在记录

PL/SQL procedure successfully completed

【代码说明】set serverout on 用于打开缓冲，将 dbms_output.put_line 所输出的信息打印到命令行或者控制台；select count(*) into employee_num from t_employees where status = 'ACT' 用于统计表 t_employees 中 status 列为“ACT”的记录数目，并将该数目存储到变量 employee_num 中；if employee_num>0 then 用于判断变量 employee_num 是否大于 0，若该变量的值大于 0，则输出“表 t_employees 中存在有效记录”字样；else 表示以上条件不符合时，将输出“表

T_EMPLOYEES 中不存在有效记录”字样；end if 表示 if 语句的结尾，在 PL/SQL 编程中，if 语句必须以 end if 结束。

如果条件判断中，含有两种或两种以上情况，则可以使用 elsif 来表示后续的条件语句，如范例 15-2 所示。

【范例 15-2】 演示如何利用 elsif 进行条件判断。

```
SQL> declare employee_num number;
2 begin
3   select count(*) into employee_num from t_employees where status = 'ACT';
4   if employee_num>1 then
5     dbms_output.put_line('表 T_EMPLOYEES 中存在多条有效记录');
6   elsif employee_num = 1 then
7     dbms_output.put_line('表 T_EMPLOYEES 中仅存在一条有效记录');
8   else
9     dbms_output.put_line('表 T_EMPLOYEES 中不存在记录');
10  end if;
11 end;
12 /
```

表 T_EMPLOYEES 中存在多条有效记录

PL/SQL procedure successfully completed

【代码说明】 elsif employee_num = 1 then 用于判断变量 employee_num 的值是否为 1，当上一个条件判式 if employee_num>1 不成立时，才会执行此判断。



注意：此处条件判断的写法，elsif 不能写做 else if。

15.1.2 利用 case when 进行分支判断

使用 if else，当条件分支很多时，将不得不使用多个 elsif 分支。从 Oracle 9i 开始，可以利用另外一个条件判断的语句来实现多分支判断，这就是 case when 语句。可以利用 case when 语句来改写范例 15-2 中的代码。

【范例 15-3】 演示如何利用 case when 替换 if else 判断。

```
SQL> declare employee_num number;
2 begin
3   select count(*) into employee_num from t_employees where status = 'ACT';
4   case
5     when employee_num>1 then
6       dbms_output.put_line('表 T_EMPLOYEES 中存在多条有效记录');
7     when employee_num = 1 then
8       dbms_output.put_line('表 T_EMPLOYEES 中仅存在一条有效记录');
9     else
10      dbms_output.put_line('表 T_EMPLOYEES 中不存在记录');
11  end case;
12 end;
13 /
```

表 T_EMPLOYEES 中存在多条有效记录

PL/SQL procedure successfully completed

【代码说明】 case 作为 case 判断的起始关键字；when employee_num>1 then 用于判断变量



employee_num 的值是否大于 1; when employee_num = 1 then 用于判断变量 employee_num 的值是否等于 1; else 则表示以上条件都不成立时的情形; end case 如同 end if 一样是 case 语句的结束关键字。

case when 还有另外一种形式, 即 case expression when value else。范例 15-3 中的代码可以利用这种形式进行改写:

【范例 15-4】 演示 case expression when value 的判断形式。

```
SQL> declare employee_num number;
2 begin
3   select count(*) into employee_num from t_employees where status = 'ACT';
4   case employee_num
5     when 0 then
6       dbms_output.put_line('表 T_EMPLOYEES 中不存在记录');
7     when 1 then
8       dbms_output.put_line('表 T_EMPLOYEES 中仅存在一条有效记录');
9     else
10      dbms_output.put_line('表 T_EMPLOYEES 中存在多条有效记录');
11   end case;
12 end;
13 /
```

表 T_EMPLOYEES 中存在多条有效记录

【代码说明】 case employee_num 指定要对哪个变量的值进行判断; when 0 then 用于匹配变量 employee_num 为 0 的情况; when 1 then 用于匹配变量 employee_num 为 1 的情况; else 用于当所有列出的数据都无法正常匹配时。



注意与说明:

1. case 语句, 在一种情况满足时, 将不会继续匹配下面的条件, 而是直接跳出判断语句。
2. 第二种 case 方式, 只适用于表达式的值在有限且确定的范围之内进行匹配; 而第一种方式则是使用逻辑判断。

case 语句的另一种用法, 则可以作为表达式出现在 select 语句中。例如在学生成绩表中, 有如下数据:

```
SQL> select * from scores;
```

SCORE_ID	STUDENT_NAME	SUBJECT	SCORE
1	张小琴	数学	100
2	张小琴	语文	50
3	张小琴	英语	0
4	刘明	数学	80
5	刘明	语文	70
6	刘明	英语	86
7	周欣欣	语文	78
8	周欣欣	数学	89
9	周欣欣	英语	87
10	李周宇	数学	90
11	李周宇	英语	86
12	李周宇	语文	76

12 rows selected



现欲将单科成绩单转换为学生的综合成绩单，那么可以利用范例 15-5 所示的代码。

【范例 15-5】 演示如何利用 case 语句统计学生成绩。

```
SQL> select student_name,
2      sum(case when subject='数学' then score else 0 end) "数学",
3      sum(case when subject='语文' then score else 0 end) "语文",
4      sum(case when subject='英语' then score else 0 end) "英语"
5 from scores
6 group by student_name
7 /
```

STUDENT_NAME	数学	语文	英语
李周宇	90	76	86
刘明	80	70	86
张小琴	100	50	0
周欣欣	89	78	87

【代码说明】group by student_name 首先将表 scores 中的记录按照学生姓名进行分组；sum() 函数作为聚合函数处理分组中的多条记录；case when subject='数学' then score else 0 end 用于判断单条记录中的科目 (subject) 是否为“数学”，如果是则返回其成绩，如果为否，将返回 0；在 sum() 对分组中的数学成绩统计后，生成新的列“数学”；“语文”和“英语”列的处理方式相同。



注意与说明：

1. case 语句在这里是作为表达式出现的，其目的是为了返回单科成绩。
2. then 之后是一个表达式（数值也可以看做表达式），并非 DML 操作语句。
3. 此处的 case 语句以 end 结尾，并非 end case。



15.2 Oracle 中的循环语句

循环语句是除了条件语句之外另一种常用的控制语句。Oracle 提供了三种主要的循环语句：无条件循环、while 循环和 for 循环。

15.2.1 无条件循环

Oracle 中的无条件循环是指循环本身并不提供循环条件，而是由 exit 语句来控制何时跳出循环。

【范例 15-6】 演示如何使用无条件循环打印员工姓名。

```
SQL> set serverout on;
SQL> declare num number:=0;
2      v_name varchar2(20);
3 begin
4 loop
5     if num>=5 then
6         exit;
7     end if;
8
9     num := num + 1;
10
```



```
11      select employee_name into v_name from t_employees where employee_id = num;
12
13      dbms_output.put_line(num || '号员工是: ' || v_name);
14  end loop;
15 end;
16 /
```

1 号员工是: 金瑞
2 号员工是: 钟君
3 号员工是: 王山
4 号员工是: 刘迪
5 号员工是: 钟会

PL/SQL procedure successfully completed

【代码说明】loop 标识循环开始，end loop 标识循环结束，二者之间的部分为循环的主体；num := num + 1 用于每次将 num 变量累加 1；if num >= 5 then exit 用于判断变量 num 的值是否大于等于 5，如果大于等于 5，则退出循环。

可以利用 exit when 来代替 if 判断，修改后代码如范例 15-7 所示。

【范例 15-7】演示如何利用 exit when 跳出循环。

```
SQL> declare num number:=0;
2      v_name varchar2(20);
3  begin
4      loop
5          exit when num>=5;
6
7          num := num + 1;
8
9          select employee_name into v_name from t_employees where employee_id = num;
10         dbms_output.put_line(num || '号员工是: ' || v_name);
11     end loop;
12 end;
13 /
```

1 号员工是: 金瑞
2 号员工是: 钟君
3 号员工是: 王山
4 号员工是: 刘迪
5 号员工是: 钟会

PL/SQL procedure successfully completed

【代码说明】exit when num >= 5 表示在 num 大于等于 5 时退出循环。

15.2.2 while 循环

while 循环也是利用 loop 循环的形式。只是在 loop 循环之前添加条件判断，while 即用来作为条件的标识。可以利用 while 循环改写范例 15-7 的代码。

【范例 15-8】演示 while 循环的使用。

```
SQL> declare num number:=0;
2      v_name varchar2(20);
3  begin
4      while num < 5 loop
5          num := num + 1;
6          select employee_name into v_name from t_employees where employee_id = num;
```

```
7      dbms_output.put_line(num || '号员工是: ' || v_name);
8  end loop;
9  end;
10 /
```

```
1 号员工是: 金瑞
2 号员工是: 钟君
3 号员工是: 王山
4 号员工是: 刘迪
5 号员工是: 钟会
```

PL/SQL procedure successfully completed

【代码说明】while num < 5 loop 在 loop 循环之前添加了条件判断, 要求变量 num 小于 5。

15.2.3 for 循环

for 循环适用于循环次数确定的情况。for 循环会首先评估循环次数, 然后针对每次循环都会将循环计数器累加 1, 直至达到循环次数。

【范例 15-9】演示如何利用 for 循环获得 ID 号 1~5 的员工姓名。

```
SQL> declare v_name varchar2(20);
2  begin
3      for i in 1 .. 5 loop
4          select employee_name into v_name from t_employees where employee_id = i;
5          dbms_output.put_line(i || '号员工是: ' || v_name);
6      end loop;
7  end;
8  /
```

```
1 号员工是: 金瑞
2 号员工是: 钟君
3 号员工是: 王山
4 号员工是: 刘迪
5 号员工是: 钟会
```

PL/SQL procedure successfully completed

【代码说明】for i in 1 .. 5 loop 用于创建 for 循环, 其中 i 作为循环计数器, 1..5 指定了循环计数的范围——从 1 开始, 至 5 结束。

for 循环的循环范围可以利用变量来指定。例如, 要查询所有员工姓名, 可以利用范例 15-10 所示的代码。

【范例 15-10】演示如何利用变量指定 for 循环的范围。

```
SQL> declare v_name varchar2(20);
2      v_count number;
3
4  begin
5      select count(*) into v_count from t_employees;
6      for i in 1 .. v_count loop
7          select employee_name into v_name from t_employees where employee_id=i;
8          dbms_output.put_line(i || '号员工是: ' || v_name);
9      end loop;
10 end;
11 /
```

```
1 号员工是: 金瑞
```



2 号员工是: 钟君
 3 号员工是: 王山
 4 号员工是: 刘迪
 5 号员工是: 钟会
 6 号员工是: 张玉
 7 号员工是: 柳青
 8 号员工是: 胡东
 9 号员工是: 商乾
 10 号员工是: 王蒙
 11 号员工是:

PL/SQL procedure successfully completed

【代码说明】select count(*) into v_count from t_employees 用于将表 t_employees 的记录数存储到变量 v_count 中; for i in 1 .. v_count loop 用于指定循环范围——从 1 至变量 v_count 所指定的范围。



15.3 本章实例

在 Oracle 中, 同样可以使用嵌套循环, 如范例 15-11 所示。

【范例 15-11】演示 Oracle 中的嵌套循环。

```
SQL> declare num number:=0;
       2   v_name varchar2(20);
       3   begin
       4     while num <5 loop
       5       num := num + 1;
       6       select employee_name into v_name from t_employees where employee_id = num;
       7       dbms_output.put_line(num || '号员工是: ' || v_name);
       8       for i in 1..num loop
       9         dbms_output.put('*');
      10       end loop;
      11       dbms_output.put_line('');
      12     end loop;
      13 end;
      14 /
```

1 号员工是: 金瑞
 *
 2 号员工是: 钟君
 **
 3 号员工是: 王山

 4 号员工是: 刘迪

 5 号员工是: 钟会

PL/SQL procedure successfully completed

【代码说明】while num <5 loop 用于当 num 变量的值小于 5 时, 进行循环; for i in 1..num loop 则用于对 1 至 num 变量的当前值进行循环处理; dbms_output.put(*) 用于在第二层循环内部打印 “*”, put 命令不同于 put_line 命令, put 命令不输入换行符。

通过本例可以看出, Oracle 中的嵌套循环与其他语言中的嵌套循环用法一致。当然, 开发者也可以使用其他任意两种循环形式进行嵌套循环。



15.4 本章小结

本章通过几个实例，介绍了如何使用循环语句。Oracle 中所有循环的实质都是 loop 循环。只是根据是否在 loop 之前指定条件和条件的具体格式，分为无条件循环和 while 循环、for 循环。值得注意的是，无条件循环内部对于 exit 的处理，一定要保证 exit 能够成功执行，否则，将会造成死循环。



15.5 习题

1. 试用 case when 语句改写以下语句。

```
if course = '0' then
    dbms_output.put_line('数学');
elsif course = '1' then
    dbms_output.put_line('语文');
elsif course = '2' then
    dbms_output.put_line('英语');
else
    dbms_output.put_line('其他');
end if;
```

2. 试用无条件循环打印表 students 的雇员信息（包括雇员 ID、雇员姓名）。
3. 利用 while 循环来完成上题的功能。
4. 利用 for 循环计算 1~100 之间所有奇数的和。



第 16 章 SQL 查询

数据库开发中，SQL 查询无处不在。在以前的内容中，已经用到了许多查询的实例。本章将详细讲述 Oracle 中的 SQL 查询。本章的主要内容如下。

- 基本查询：主要讲述查询语句及各种子句的使用；
- 子查询：理解和使用子查询；
- 联合语句：针对多个查询结果集合的运算；
- 关联语句：表之间的关联关系；
- 层次化查询：connect by。

通过本章学习，读者可以掌握 SQL 查询的方方面面，并且可以了解高级应用——层次化查询。



16.1 基本查询

Oracle 中的数据查询可以非常复杂。复杂的查询是由基本查询构成的，因此，理解基本查询是必需的，本节将着重讲述 Oracle 中的基本查询。

16.1.1 select 语句查询执行步骤

select 是查询中的首要关键字。select 用于指定查询所获得的结果列。select 列表之后，需要紧跟 from 子句。from 子句指定查询的数据源。这里的数据源可以是一个表，也可以是一个临时记录集合，如内嵌视图或者多个子查询的集合操作获得的记录集合。以表 t_employees 为例，利用如下语句可以查询表中所有员工姓名。

```
SQL> select employee_name from t_employees;
```

```
EMPLOYEE_NAME
```

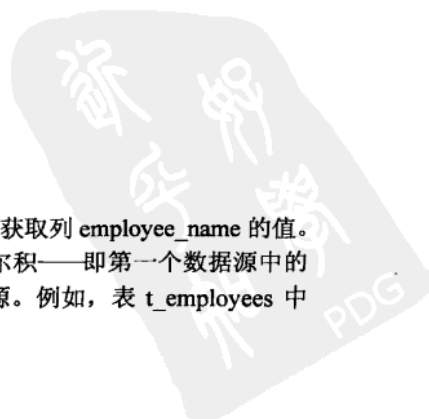
```
-----
```

```
金瑞  
钟君  
王山  
刘迪  
钟会  
张玉  
柳青  
胡东  
商乾  
王蒙
```

```
10 rows selected
```

Oracle 会遍历表 t_employees 中的每条记录，并利用 select 语句获取列 employee_name 的值。

当 from 子句中含有多个数据源时，这些数据源将实现笛卡尔积——即第一个数据源中的每条记录与第二个数据源中所有记录组合，最终形成新的数据源。例如，表 t_employees 中



含有 10 条记录,而表 t_salary 中含有 30 条记录,那么查询语句 `select * from t_employees, t_salary` 的数据源将有 $10 \times 30 = 300$ 条记录。

【范例 16-1】 演示如何获得笛卡尔积。

SQL> select * from t_employees, t_salary;

employee_id	employee_name	work_years	status	salary_id	employee_id	month	salary
1	金瑞	6	ACT	1	1	1月	8000
1	金瑞	6	ACT	2	2	1月	7000
1	金瑞	6	ACT	3	3	1月	7000
1	金瑞	6	ACT	4	4	1月	7000
1	金瑞	6	ACT	5	5	1月	6000
1	金瑞	6	ACT	6	6	1月	5500
1	金瑞	6	ACT	7	7	1月	5000
1	金瑞	6	ACT	8	8	1月	4000
1	金瑞	6	ACT	9	9	1月	4000
1	金瑞	6	ACT	10	10	1月	3000
1	金瑞	6	ACT	11	1	2月	8000
1	金瑞	6	ACT	12	2	2月	7000
1	金瑞	6	ACT	13	3	2月	7000
1	金瑞	6	ACT	14	4	2月	7000
1	金瑞	6	ACT	15	5	2月	6000
1	金瑞	6	ACT	16	6	2月	5500
1	金瑞	6	ACT	17	7	2月	5000
1	金瑞	6	ACT	18	8	2月	4000
1	金瑞	6	ACT	19	9	2月	4000
1	金瑞	6	ACT	20	10	2月	3000
.							
.							
.							
10	王蒙	2	ACT	1	1	1月	8000
10	王蒙	2	ACT	2	2	1月	7000
10	王蒙	2	ACT	3	3	1月	7000
10	王蒙	2	ACT	4	4	1月	7000
10	王蒙	2	ACT	5	5	1月	6000
10	王蒙	2	ACT	6	6	1月	5500
10	王蒙	2	ACT	7	7	1月	5000
10	王蒙	2	ACT	8	8	1月	4000
10	王蒙	2	ACT	9	9	1月	4000
10	王蒙	2	ACT	10	10	1月	3000
10	王蒙	2	ACT	11	1	2月	8000
10	王蒙	2	ACT	12	2	2月	7000
10	王蒙	2	ACT	13	3	2月	7000
10	王蒙	2	ACT	14	4	2月	7000
10	王蒙	2	ACT	15	5	2月	6000
10	王蒙	2	ACT	16	6	2月	5500
10	王蒙	2	ACT	17	7	2月	5000
10	王蒙	2	ACT	18	8	2月	4000
10	王蒙	2	ACT	19	9	2月	4000
10	王蒙	2	ACT	20	10	2月	3000

300 rows selected



在获得了 from 子句的最终数据源之后，select 语句再捕获预期列。

16.1.2 where 子句

where 子句用于过滤 from 子句所获得的数据源。可以为范例 16-1 所示的查询语句添加 where 子句，如范例 16-2 所示。

【范例 16-2】 演示如何为查询添加 where 子句。

```
SQL> select e.employee_id, e.employee_name, s.salary
  2  from t_employees e, t_salary s
  3  where e.employee_id = s.employee_id
  4  /
```

EMPLOYEE_ID	EMPLOYEE_NAME	SALARY
1	金瑞	8000
2	钟君	7000
3	王山	7000
4	刘迪	7000
5	钟会	6000
6	张玉	5500
7	柳青	5000
8	胡东	4000
9	商乾	4000
10	王蒙	3000
1	金瑞	8000
2	钟君	7000
3	王山	7000
4	刘迪	7000
5	钟会	6000
6	张玉	5500
7	柳青	5000
8	胡东	4000
9	商乾	4000
10	王蒙	3000
1	金瑞	8000
2	钟君	7000
3	王山	7000
4	刘迪	7000
5	钟会	6000
6	张玉	5500
7	柳青	5000
8	胡东	4000
9	商乾	4000
10	王蒙	3000

30 rows selected

【代码说明】from t_employees e, t_salary s 首先指定了数据源为两个表的笛卡尔积，并为两个表定义别名，因为两个表中都有列 employee_id，所以，必须指定表别名，以区分两个表中的同名列；where e.employee_id = s.employee_id 用于过滤数据源中的数据，该条件为 e.employee_id 等于 s.employee_id，Oracle 会搜索范例 16-1 所示的查询结果中的每条记录，并依据查询条件获得记录集；select e.employee_id, e.employee_name, s.salary 捕获该记录集中的列值。



注意：当 from 子句中含有多个数据表（或其他数据源）时，为各数据表指定别名是一个良好的习惯，即使 select 语句所要捕获的列不会引起列名的混淆。

16.1.3 利用 distinct 获得唯一性记录

distinct 关键字可用于获得唯一性记录，被 distinct 限制的既可以是单个列，也可以是多个列的组合。

【范例 16-3】演示如何利用 distinct 关键字获得工资表中的所有月份。

```
SQL> select distinct month from t_salary;
```

```
MONTH
```

```
-----
```

```
1 月
```

```
2 月
```

```
3 月
```

【代码说明】select distinct month 用于限制捕获的 moth 列值唯一。

可以获得员工的工资的唯一性记录，以表现员工的工资变化，代码如下：

```
SQL> select distinct e.employee_name, s.salary
2  from t_employees e, t_salary s
3  where e.employee_id = s.employee_id
4  /
```

```
EMPLOYEE_NAME  SALARY
```

```
-----
```

```
胡东          4000
```

```
金瑞          8000
```

```
刘迪          7000
```

```
柳青          5000
```

```
商乾          4000
```

```
王蒙          3000
```

```
王山          7000
```

```
张玉          5500
```

```
钟会          6000
```

```
钟君          7000
```

```
10 rows selected
```

【代码说明】select distinct e.employee_name, s.salary 中含有两个列——e.employee_name 和 s.salary，那么 select 语句将捕获唯一的 employee_name 和 salary 的组合。

16.1.4 order by 子句

order by 子句可以将查询的结果，按照一定的顺序进行排序。例如，在范例 16-3 中，获得了所有员工的工资信息，那么可以利用 order by 子句将查询结果排序。

【范例 16-4】演示如何使用 order by 子句。

```
SQL> select distinct e.employee_name, s.salary
2  from t_employees e, t_salary s
3  where e.employee_id = s.employee_id
4  order by s.salary;
```

```
EMPLOYEE_NAME  SALARY
```



王蒙	3000
胡东	4000
商乾	4000
柳青	5000
张玉	5500
钟会	6000
刘迪	7000
王山	7000
钟君	7000
金瑞	8000

10 rows selected

【代码说明】`order by s.salary` 子句将 `select` 语句获得的最终结果集合进行排序，排序字段为 `salary`；`order by` 提供的排序规则有两种——升序（`asc`）和降序排列（`desc`），默认情况下为升序排列。范例 16-5 演示了降序排列的使用。

【范例 16-5】演示降序排列的使用。

```
SQL> select distinct e.employee_name, s.salary
  2  from t_employees e, t_salary s
  3  where e.employee_id = s.employee_id
  4  order by s.salary desc
  5  /
```

EMPLOYEE_NAME	SALARY
金瑞	8000
刘迪	7000
王山	7000
钟君	7000
钟会	6000
张玉	5500
柳青	5000
胡东	4000
商乾	4000
王蒙	3000

10 rows selected

【代码说明】`order by s.salary desc` 用于将查询结果按照工资的降序排列。



注意：当排序列的数据类型是字符串时，将按照字符串在字母表中的顺序进行排列。

16.1.5 group by 子句

`group by` 子句用于对记录集合进行分组。一旦使用了分组之后，`select` 语句的真实操作目标即为各个分组数据，每次循环处理的也是各个分组，而不再是单条记录。

【范例 16-6】演示如何利用 `group by` 统计每位员工工资。

```
SQL> select e.employee_name, sum(s.salary)
  2  from t_employees e, t_salary s
  3  where e.employee_id = s.employee_id
  4  group by e.employee_id, e.employee_name;
```

该查询语句的执行过程如下:

(1) from t_employees e, t_salary s 定义了查询数据源——表 t_employees 与表 t_salary 的笛卡尔积, 如范例 16-1 中的 330 条记录。

(2) where e.employee_id = s.employee_id, 将 from 子句所定义的数据源进行进一步的筛选。条件为表 t_employees 的列 employee_id 的值等于表 t_salary 中 employee_id 的值。数据源进一步缩小为如下所示的 30 条记录。

employee_id	employee_name	work_years	status	salary_id	employee_id	month	salary
1	金瑞	6	ACT	1	1	1 月	8000
2	钟君	6	ACT	2	2	1 月	7000
3	王山	6	ACT	3	3	1 月	7000
4	刘迪	5	ACT	4	4	1 月	7000
5	钟会	4	ACT	5	5	1 月	6000
6	张玉	4	ACT	6	6	1 月	5500
7	柳青	4	ACT	7	7	1 月	5000
8	胡东	4	ACT	8	8	1 月	4000
9	商乾	4	ACT	9	9	1 月	4000
10	王蒙	2	ACT	10	10	1 月	3000
1	金瑞	6	ACT	11	1	2 月	8000
2	钟君	6	ACT	12	2	2 月	7000
3	王山	6	ACT	13	3	2 月	7000
4	刘迪	5	ACT	14	4	2 月	7000
5	钟会	4	ACT	15	5	2 月	6000
6	张玉	4	ACT	16	6	2 月	5500
7	柳青	4	ACT	17	7	2 月	5000
8	胡东	4	ACT	18	8	2 月	4000
9	商乾	4	ACT	19	9	2 月	4000
10	王蒙	2	ACT	20	10	2 月	3000
1	金瑞	6	ACT	21	1	3 月	8000
2	钟君	6	ACT	22	2	3 月	7000
3	王山	6	ACT	23	3	3 月	7000
4	刘迪	5	ACT	24	4	3 月	7000
5	钟会	4	ACT	25	5	3 月	6000
6	张玉	4	ACT	26	6	3 月	5500
7	柳青	4	ACT	27	7	3 月	5000
8	胡东	4	ACT	28	8	3 月	4000
9	商乾	4	ACT	29	9	3 月	4000
10	王蒙	2	ACT	30	10	3 月	3000

(3) group by e.employee_id, e.employee_name, 将现有数据源按照 t_employee 表的 employee_id 列和 employee_name 列的组合进行分组。分组后的数据源如下:

employee_id	employee_name	work_years	status	salary_id	employee_id	month	salary
(1 组)							
1	金瑞	6	ACT	1	1	1 月	8000
1	金瑞	6	ACT	11	1	2 月	8000
1	金瑞	6	ACT	21	1	3 月	8000
(2 组)							
2	钟君	6	ACT	2	2	1 月	7000
2	钟君	6	ACT	12	2	2 月	7000
2	钟君	6	ACT	22	2	3 月	7000



(3 组)							
3	王山	6	ACT	3	3	1 月	7000
3	王山	6	ACT	13	3	2 月	7000
3	王山	6	ACT	23	3	3 月	7000
(4 组)							
4	刘迪	5	ACT	4	4	1 月	7000
4	刘迪	5	ACT	14	4	2 月	7000
4	刘迪	5	ACT	24	4	3 月	7000
(5 组)							
5	钟会	4	ACT	5	5	1 月	000
5	钟会	4	ACT	15	5	2 月	6000
5	钟会	4	ACT	25	5	3 月	6000
(6 组)							
6	张玉	4	ACT	6	6	1 月	5500
6	张玉	4	ACT	16	6	2 月	5500
6	张玉	4	ACT	26	6	3 月	5500
(7 组)							
7	柳青	4	ACT	7	7	1 月	5000
7	柳青	4	ACT	17	7	2 月	5000
7	柳青	4	ACT	27	7	3 月	5000
(8 组)							
8	胡东	4	ACT	8	8	1 月	4000
8	胡东	4	ACT	18	8	2 月	4000
8	胡东	4	ACT	28	8	3 月	4000
(9 组)							
9	商乾	4	ACT	9	9	1 月	4000
9	商乾	4	ACT	19	9	2 月	4000
9	商乾	4	ACT	29	9	3 月	4000
(10 组)							
10	王蒙	2	ACT	10	10	1 月	3000
10	王蒙	2	ACT	20	10	2 月	3000
10	王蒙	2	ACT	30	10	3 月	3000

(4) select e.employee_name, sum(s.salary), 作用对象为每个组, 因为在分组时, 分组标准中含有 e.employee_name 列, 因此, 可以将其作为查询目标列。sum(s.salary)用于获得单个分组中 salary 列值的和。最终查询结果如下:

EMPLOYEE_NAME	SUM(S.SALARY)

金瑞	24000
钟君	21000
王山	21000
刘迪	21000
钟会	18000
张玉	16500
柳青	15000
胡东	12000
商乾	12000
王蒙	9000



10 rows selected

order by 子句是对最终结果进行排序的，因此，如果一条查询语句中同时存在着 **group by** 子句和 **order by** 子句，那么，应该将 **order by** 子句置于 **group by** 子句之后，并且 **order by** 子句的排序标准不能出现 **group by** 定义之外的列。例如，范例 16-6 中的查询语句是针对 **employee_id** 和 **employee_name** 进行分组的，那么尝试使用 **work_years** 进行排序，oracle 将会抛出错误提示。

```
SQL> select e.employee_name, sum(s.salary)
2 from t_employees e, t_salary s
3 where e.employee_id = s.employee_id
4 group by e.employee_id, e.employee_name
5 order by e.work_years
6 /
```

```
select e.employee_name, sum(s.salary)
from t_employees e, t_salary s
where e.employee_id = s.employee_id
group by e.employee_id, e.employee_name
order by e.work_years
```

ORA-00979: not a GROUP BY expression

【代码说明】**order by e.work_years** 用于排序查询结果集合，排序标准为 **e.work_years**。Oracle 将会抛出 **work_year** 不是分组表达式的错误提示。

这里所说的分组表达式，不仅仅是 **group by** 子句所定义的列（**e.employee_id** 和 **e.employee_name**），还包括了聚合函数。例如，可以将查询结果按照员工的总工资由低到高进行排序，相应的代码如范例 16-7 所示。

【范例 16-7】演示在 **order by** 子句中使用聚合函数。

```
SQL> select e.employee_name, sum(s.salary)
2 from t_employees e, t_salary s
3 where e.employee_id = s.employee_id
4 group by e.employee_id, e.employee_name
5 order by sum(s.salary);
```

EMPLOYEE_NAME	SUM(S.SALARY)
王蒙	9000
胡东	12000
商乾	12000
柳青	15000
张玉	16500
钟会	18000
钟君	21000
王山	21000
刘迪	21000
金瑞	24000

10 rows selected

16.1.6 having 子句

where 子句会对 **from** 子句所定义的数据源进行条件过滤，但是针对 **group by** 子句形成的分组之后的结果集，**where** 子句将无能为力。为了过滤 **group by** 子句所生成的结果集，可以使用 **having** 子句。



【范例 16-8】 演示如何筛选工资总额大于 15000 的员工信息。

```
SQL> select e.employee_name, sum(s.salary)
  2   from t_employees e, t_salary s
  3   where e.employee_id = s.employee_id
  4   group by e.employee_id, e.employee_name
  5   having sum(s.salary)>15000
  6   order by sum(s.salary);
```

EMPLOYEE_NAME	SUM(S.SALARY)
张玉	16500
钟会	18000
钟君	21000
王山	21000
刘迪	21000
金瑞	24000

6 rows selected

【代码说明】 having sum(s.salary)>15000 用于过滤分组集合；sum(s.salary)是聚合函数，将返回分组集合中 salary 列的和。



16.2 子查询

子查询是指嵌套在查询语句中的查询语句。子查询出现的位置一般为条件语句，如 where 条件。Oracle 会首先执行子查询，然后执行父查询。

16.2.1 理解子查询

子查询是完整的查询语句。子查询首先生成结果集，并将结果集应用于条件语句。本节所要讲述的子查询与内嵌视图不同。内嵌视图也可以看做临时查询结果，但是内嵌视图出现在 from 子句中，并与其他数据源（数据表、视图等）形成笛卡尔积运算。而子查询单独运算，不会与其他数据源进行笛卡尔积运算。

子查询可以出现在插入、查询、更新和删除语句中。建立子查询的目的在于更加有效地限制 where 子句中的条件，并可以将复杂的查询逻辑梳理得更加清晰。

子查询可以访问父查询中的数据源（数据表、视图等），但是父查询不能够访问子查询 from 子句所定义的数据源。当子查询访问父查询中的数据源时，子查询的查询结果集合可能随着记录的推移而发生变化，因为此时的子查询是针对父查询中的每条记录执行的。

16.2.2 子查询使用实例

在表 t_employees 中，可以利用子查询获得实际领取过工资的员工信息。

【范例 16-9】 演示如何利用子查询获得实际领取过工资的员工信息。

```
SQL> select * from t_employees
  2   where employee_id in (select employee_id from t_salary)
  3   /
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	6	ACT

2	钟君	6	ACT
3	王山	6	ACT
4	刘迪	5	ACT
5	钟会	4	ACT
6	张玉	4	ACT
7	柳青	4	ACT
8	胡东	4	ACT
9	商乾	4	ACT
10	王蒙	2	ACT

10 rows selected

【代码说明】select employee_id from t_salary 是一个不引用父查询的子查询，它将生成一个包含所有 employee_id 的结果集；where employee_id in (select employee_id from t_salary)作为父查询的条件，判断 employee_id 的值是否在子查询生成的结果集中。

这里需要注意的是，含有子查询的查询语句与普通查询语句具有完全相同的执行过程——都是针对表 t_employees 中每条记录，进行一次迭代，判断是否符合 where 定义的查询条件，并利用 select 语句将其捕获。

当然，子查询可以引用父查询中的数据源。例如，可以利用 exists 判式来代替 in 判式，实现范例 16-9 所示的查询。

【范例 16-10】演示如何在子查询中引用父查询中的数据源。

```
SQL> select * from t_employees e
2  where exists (select employee_id from t_salary where employee_id=e.employee_id)
3  /
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	6	ACT
2	钟君	6	ACT
3	王山	6	ACT
4	刘迪	5	ACT
5	钟会	4	ACT
6	张玉	4	ACT
7	柳青	4	ACT
8	胡东	4	ACT
9	商乾	4	ACT
10	王蒙	2	ACT

10 rows selected

【代码说明】select employee_id from t_salary where employee_id = e.employee_id 作为一个子查询，引用了父查询中的表 t_employees 的 employee_id 列。其查询过程如下所示：

(1) from 子句的数据源为表 t_salary 的所有记录。

对于表 t_employees 的第一条记录，子查询实际执行的 sql 语句为：select employee_id from t_salary where employee_id = 1，查询结果如下：

```
EMPLOYEE_ID
-----
1
1
1
```

而 exists 判式将返回为真，因此父查询的 select 语句将捕获第一条记录。



(2) 同理，对于表 `t_employees` 中的其他记录子查询都会执行一次，并依据 `exists` 判式的值来决定是否应该捕获记录。



16.3 联合语句

联合语句是指两个或多个 `select` 语句是并列关系，并且对这些 `select` 语句所捕获的记录集进行集合操作，以获得最终的结果集。这些联合语句包括以下几种：`union` 查询、`union all` 查询、`intersect` 查询和 `minus` 查询。

16.3.1 union 查询

`union` 查询是指两个查询结果集进行并集操作，并将重复记录剔除，即相当于并集操作之后，再执行一次 `distinct` 操作。

在一个 `web` 页面上经常会用到下拉框，下拉框中的选项往往从数据库中获得。例如，为了获得所有员工列表，可以利用范例 16-11 所示的查询语句。

【范例 16-11】 演示如何利用 `union` 查询获得 `web` 页面中员工列表的数据源。

```
SQL> select 0 value, '请选择' display_value
2 from dual
3 union
4 select employee_id value, employee_name display_value
5 from t_employees
6 where status = 'ACT'
7 /
```

VALUE	DISPLAY_VALUE
0	请选择
1	金瑞
2	钟君
3	王山
4	刘迪
5	钟会
6	张玉
7	柳青
8	胡东
9	商乾
10	王蒙

11 rows selected

【代码说明】`select 0 value, '请选择' display_value from dual` 用于获得单条记录，该记录有两列，列名分别为“`value`”和“`display_value`”，其值分别为 0 和“请选择”；`select employee_id value, employee_name display_value from t_employees where status = 'ACT'` 用于捕获表 `t_employees` 中所有有效员工的员工 `id` 和 `employee_name` 列，并分别为之赋予别名“`value`”和“`display_value`”；`union` 对两个结果集进行并集操作。

进行 `union` 操作需要注意的是各个结果集的对应列必须具有相同的数据类型。例如，在范例 16-11 中，尝试使用字符串“0”来代替数字 0，`oracle` 将会抛出错误提示。

```
SQL> select '0' value, '请选择' display_value
2 from dual
3 union
```



```

4 select employee_id value, employee_name display_value
5 from t_employees
6 where status = 'ACT'
7 /

```

```

select '0' value, '请选择' display_value
from dual
union
select employee_id value, employee_name display_value
from t_employees
where status = 'ACT'

```

ORA-01790: expression must have same datatype as corresponding expression

查询语句 `select '0' value, '请选择' display_value` 中的 `value` 列的数据类型为字符串类型，而查询语句 `select employee_id value, employee_name display_value from t_employees where status = 'ACT'` 中的 `value` 列的数据类型为数值型，两个结果集在合并时将会出现数据类型不匹配的误差。

此外，`union` 操作的各结果集，只要保证列数相同，并且各列的数据类型一致即可，并不要求其具有统一的列名。并集操作的最终结果的列名将统一使用第一个查询语句所获得的列名。

【范例 16-12】 演示使用不同列名的结果集的并集。

```

SQL> select 0 value, '请选择' display_value
2 from dual
3 union
4 select employee_id, employee_name
5 from t_employees
6 where status = 'ACT'
7 /

```

VALUE	DISPLAY_VALUE
0	请选择
1	金瑞
2	钟君
3	王山
4	刘迪
5	钟会
6	张玉
7	柳青
8	胡东
9	商乾
10	王蒙

11 rows selected

【代码说明】 `select employee_id, employee_name from t_employees where status = 'ACT'` 表明查询结果的列名分别为“`employee_id`”和“`employee_name`”，而最终的查询结果使用了第一个查询结果的列名“`value`”和“`display_value`”。

16.3.2 union all 查询

`union all` 查询与 `union` 同为并集操作，但 `union all` 查询并不删除最终结果集中的重复记录，因此 `union all` 的操作要快于 `union`。为了演示 `union all` 的查询结果，首先创建一个名为



t_managers 的表，该表存储了经理人员的信息。

```
SQL> select * from t_managers;
```

MANAGER_ID	MANAGER_NAME	WORK_YEARS	STATUS
1	刘建	7	ACT
2	吴新	7	ACT
3	金瑞	6	ACT

为了获得经理人员与普通员工的信息，可以利用如范例 16-13 所示的 SQL 语句。

【范例 16-13】 演示 union all 的使用。

```
SQL> select manager_name name, status
2   from t_managers
3   where status = 'ACT'
4   union all
5   select employee_name, status
6   from t_employees
7   where status = 'ACT'
8   /
```

NAME	STATUS
刘建	ACT
吴新	ACT
金瑞	ACT
金瑞	ACT
钟君	ACT
王山	ACT
刘迪	ACT
钟会	ACT
张玉	ACT
柳青	ACT
胡东	ACT
商乾	ACT
王蒙	ACT

13 rows selected

【代码说明】select manager_name name, status from t_managers where status = 'ACT'用于查询所有有效经理人员信息；select employee_name, status from t_employees where status = 'ACT'用于查询所有有效的员工信息；union all 用于获得二者的并集。

分析查询查询结果可知，员工“金瑞”出现了重复。尝试使用 union 操作代替 union all，如范例 16-14 所示。

【范例 16-14】 演示 union 查询将删除重复记录。

```
SQL> select manager_name name, status
2   from t_managers
3   where status = 'ACT'
4   union
5   select employee_name, status
6   from t_employees
7   where status = 'ACT'
8   /
```

NAME	STATUS
------	--------



胡东	ACT
金瑞	ACT
刘迪	ACT
刘建	ACT
柳青	ACT
商乾	ACT
王蒙	ACT
王山	ACT
吴新	ACT
张玉	ACT
钟会	ACT
钟君	ACT

12 rows selected

分析查询结果可知, 利用 union 查询代替 union all 查询之后, 最终结果将不会出现重复记录。

16.3.3 intersect 查询

intersect 查询用于获得两个结果集的交集。例如, 为了获得既存在于表 t_employees 中, 又存在于表 t_managers 中的员工信息, 可以利用范例 16-15 所示的代码。

【范例 16-15】 演示 intersect 查询的使用。

```
SQL> select employee_name, status
2   from t_employees
3   where status = 'ACT'
4   intersect
5   select manager_name, status
6   from t_managers
7   where status = 'ACT'
8   /
```

EMPLOYEE_NAME	STATUS

金瑞	ACT

【代码说明】 select employee_name, status from t_employees where status = 'ACT' 用于获取表 t_employees 中的所有有效记录; select manager_name, status from t_managers where status = 'ACT' 用于获取表 t_managers 中的所有有效记录; intersect 操作用于获取二者的交集, 在本例中, 两个结果集的交集仅含有 1 条记录。

16.3.4 minus 查询

minus 查询可以看做集合间的减法运算, 该操作的第一个集合看做被减数, 而第二个集合看做减数, 那么 minus 操作将返回第一个结果集中存在, 而第二个结果集中不存在的记录。

【范例 16-16】 演示 minus 查询的使用。

```
SQL> select employee_name, status
2   from t_employees
3   where status = 'ACT'
4   minus
5   select manager_name, status
6   from t_managers
7   where status = 'ACT'
8   /
```



EMPLOYEE_NAME	STATUS
胡东	ACT
刘迪	ACT
柳青	ACT
商乾	ACT
王蒙	ACT
王山	ACT
张玉	ACT
钟会	ACT
钟君	ACT

9 rows selected

【代码说明】select employee_name, status from t_employees where status = 'ACT'用于获取表 t_employees 中的所有有效记录；select manager_name, status from t_managers where status = 'ACT'用于获取表 t_managers 中的所有有效记录；minus 操作用于获取存在于表 t_employees 中，而不存在于表 t_managers 表中的有效记录，在本例中，两个结果集的交集含有 9 条记录。



16.4 联接

联接用于指定多数据源（表、视图）之间如何组合，以形成最终的数据源。联接对于查询语句有着不可或缺的作用。如果未显式指定联接，那么将获得多个数据源的笛卡尔积。对于查询的真实目的来说，笛卡尔积往往是没有任何实际意义的。Oracle 中主要包括以下几种联接关系。

- 自然联接；
- 内连接；
- 外联接：左联接；
- 外联接：右联接；
- 外联接：完全联接。

16.4.1 自然联接

自然联接将两个数据源中具有相同名称的列进行联接。用户不必明确指定执行联接的列。自然联接应该使用 natural join 关键字。

【范例 16-17】演示自然连接的使用。

```
SQL> select *
  2  from t_employees natural join t_managers
  3  /
```

WORK_YEARS	STATUS	EMPLOYEE_ID	EMPLOYEE_NAME	MANAGER_ID	MANAGER_NAME
6	ACT	1	金瑞	3	金瑞
6	ACT	2	钟君	3	金瑞
6	ACT	3	王山	3	金瑞

【代码说明】select * from t_employees natural join t_salary 首先将表 t_employees 与表 t_salary 进行自然联接，然后选择联接结果集中的所有列。

查询结果是一个毫无意义的结果集，获得该结果集的过程如下所示。

(1) Oracle 首先检索表 `t_employees` 与表 `t_managers` 表中的共有列——`work_years`、`status`。

(2) 在表 `t_employees` 中，第一条记录如下所示：

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	6	ACT

在表 `t_managers` 中，含有一条记录，其 `work_years` 列等于 6，`status` 列等于 “ACT”。

MANAGER_ID	MANAGER_NAME	WORK_YEARS	STATUS
3	金瑞	6	ACT

那么二者将生成一条记录：

WORK_YEARS	STATUS	EMPLOYEE_ID	EMPLOYEE_NAME	MANAGER_ID	MANAGER_NAME
6	ACT	1	金瑞	3	金瑞

(3) 表 `t_employees` 中，第二条记录如下：

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
2	钟君	6	ACT

在表 `t_managers` 中，含有一条记录，其 `work_years` 列等于 6，`status` 列等于 “ACT”。

MANAGER_ID	MANAGER_NAME	WORK_YEARS	STATUS
3	金瑞	6	ACT

二者将生成一条记录：

WORK_YEARS	STATUS	EMPLOYEE_ID	EMPLOYEE_NAME	MANAGER_ID	MANAGER_NAME
6	ACT	2	钟君	3	金瑞

(4) 依此类推，在搜索完表 `t_employees` 表中的所有记录后，即可捕获最终的结果集。

WORK_YEARS	STATUS	EMPLOYEE_ID	EMPLOYEE_NAME	MANAGER_ID	MANAGER_NAME
6	ACT	1	金瑞	3	金瑞
6	ACT	2	钟君	3	金瑞
6	ACT	3	王山	3	金瑞

自然联接要求各个数据源中的联接列必须具有完全相同的名称。因此，这种联接的使用更多地停留在理论阶段，并不具有太多实际意义。

16.4.2 内联接

内联接像自然联接不同，需要在 `from` 子句中使用联接条件。但是，用户可以自行制定所要联接的各数据源的列。这克服了自然联接要求联接列必须同名的限制。

【范例 16-18】 演示内联接的使用。

```
SQL> select *
  2 from t_employees e inner join t_managers m
  3 on e.employee_name = m.manager_name
  4 /
```

employee_id	employee_name	work_years	status	manager_id	manager_name	work_years	status
1	金瑞	6	ACT	3	金瑞	6	ACT

【代码说明】 `select from t_employees e inner join t_managers m` 用于查询的数据源——表



t_employees 与表 t_managers 的内联接；on e.employee_name = m.manager_name 指定内联接条件为表 t_employees 的列 employee_name 的值等于表 t_managers 的列 manager_name 的值。

以上的内联接完全可以使用 where 子句来代替，代码如下：

```
SQL> select *
  2 from t_employees e, t_managers m
  3 where e.employee_name = m.manager_name
  4 /
employee_id employee_name work_years status manager_id manager_name work_years status
-----
1          金瑞          6      ACT          3          金瑞          6      ACT
```

内联接虽然可以指定两个数据源的联接列，但是完全可以利用 where 子句来代替。而且使用内联接之外，往往还需要使用 where 子句，代码如下：

```
SQL> select e.employee_name, e.work_years, e.status
  2 from t_employees e inner join t_managers m
  3 on e.employee_name = m.manager_name
  4 where e.work_years>5
  5 /
```

```
EMPLOYEE_NAME WORK_YEARS STATUS
-----
金瑞          6      ACT
```

【代码说明】on e.employee_name = m.manager_name 用于指定内联接的列；where e.work_years>5 在获得最终数据源的基础上，添加了过滤条件——服务年限大于 5 年。该 SQL 语句可只使用 where 子句来指定查询条件。

```
SQL> select e.employee_name, e.work_years, e.status
  2 from t_employees e, t_managers m
  3 where e.employee_name = m.manager_name
  4 and e.work_years>5
  5 /
```

```
EMPLOYEE_NAME WORK_YEARS STATUS
-----
金瑞          6      ACT
```

相对于单纯使用 where 子句，同时使用 inner join 和 where 子句更容易引起混淆，因此，内联接在实际开发中并不常用。

16.4.3 外联接——左联接

外联接与内联接不同的是，内联接中的两个数据源是并列关系，二者具有平等的地位。而外联接将其中一个数据源指定为基表（或者说主表），另一个数据源可以看做附表。在最终的数据源中，一定含有基表中的数据，而附表中的数据是否出现，则依具体的联接条件而定。

外联接分为三类：左连接、右联接和完全联接。本节着重讲述左联接的使用。

在表 t_employees 中存储了所有员工信息，表 t_salary 存储了员工的工资，为了获得所有员工的工资情况，可以利用左联接来实现该查询功能。

【范例 16-19】演示左联接的使用。

首先向表 t_employees 中插入新的员工记录：

```
SQL> insert into t_employees values (11, '周兵', 0, 'ACT');
```

```
insert into t_employees values (11, '周兵', 0, 'ACT')
```

此时，表 t_employees 中数据如下：

```
SQL> select * from t_employees;
```

EMPLOYEE_ID	EMPLOYEE_NAME	WORK_YEARS	STATUS
1	金瑞	6	ACT
2	钟君	6	ACT
3	王山	6	ACT
4	刘迪	5	ACT
5	钟会	4	ACT
6	张玉	4	ACT
7	柳青	4	ACT
8	胡东	4	ACT
9	商乾	4	ACT
10	王蒙	2	ACT
11	周兵	0	ACT

利用左联接来获得所有员工的工资状况。

```
SQL> select e.employee_id, e.employee_name, s.salary
2 from t_employees e
3 left outer join t_salary s
4 on e.employee_id = s.employee_id
5 order by e.employee_id
6 /
```

EMPLOYEE_ID	EMPLOYEE_NAME	SALARY
1	金瑞	8000
1	金瑞	8000
1	金瑞	8000
2	钟君	7000
2	钟君	7000
2	钟君	7000
3	王山	7000
3	王山	7000
3	王山	7000
4	刘迪	7000
4	刘迪	7000
4	刘迪	7000
5	钟会	6000
5	钟会	6000
5	钟会	6000
6	张玉	5500
6	张玉	5500
6	张玉	5500
7	柳青	5000
7	柳青	5000
7	柳青	5000
8	胡东	4000
8	胡东	4000
8	胡东	4000
9	商乾	4000
9	商乾	4000
9	商乾	4000
10	王蒙	3000
10	王蒙	3000



10	王蒙	3000
11	周兵	

31 rows selected

【代码说明】from t_employees e left outer join t_salary s 用于指定数据源——表 t_employees 与表 t_salary 的左联接；on e.employee_id = s.employee_id 指定外联接的联接条件——表 t_employees 的 employee_id 列值与表 t_salary 的 employee_id 列值相等。

分析查询结果可知，左联接的查询结果以 left outer join 的左侧数据表 t_employees 为基表，右侧数据表 t_salary 为附表。当左侧数据表的某条记录在右侧表中有多条记录与之对应时，将生成多条记录，以保证不会出现数据丢失。如表 t_employees 中 employee_id 为 1 的记录，在表 t_salary 中有三条记录与之对应，则左联接之后的结果，将包含三条记录。

EMPLOYEE_ID	EMPLOYEE_NAME	SALARY
1	金瑞	8000
1	金瑞	8000
1	金瑞	8000

当左侧数据表中的某条记录，在右侧数据表中没有记录与之对应，将利用 null 来填补空缺的查询结果。如 employee_id 为 11 的记录，在表 t_salary 中没有记录与之对应，则左联接之后的结果仍然有一条记录：

EMPLOYEE_ID	EMPLOYEE_NAME	SALARY
11	周兵	

由于内联接没有左右之分，因此，左联接实际成为左外联接的常用说法。而语法 left outer join 也可以简写为 left join，那么可以对范例 16-19 中的示例代码进行改写。

```
SQL> select e.employee_id, e.employee_name, s.salary
2   from t_employees e
3   left join t_salary s
4   on e.employee_id = s.employee_id
5   order by e.employee_id
6   /
```

【代码说明】left join 代替 left outer join，可以实现相同的功能。

对于左联接，Oracle 还提供了另外一种简写的方式——在 where 子句条件中添加 (+) 来指定附表。可以利用这种语法格式改写范例 16-19 中的联接语句。

【范例 16-20】演示如何利用 (+) 改写左联接语句。

```
SQL> select e.employee_id, e.employee_name, s.salary
2   from t_employees e, t_salary s
3   where e.employee_id = s.employee_id(+)
4   order by e.employee_id
5   /
```

【代码说明】在本例中省略了 left join 及 on 子句，转而使用 where e.employee_id = s.employee_id(+) 来代替；s.employee_id(+) 表明该查询是一个外联接查询，带有 (+) 的列表表示该表为一个附表，基表为另一个数据表。

16.4.4 外联接——右联接

右联接与左联接的执行过程非常相似，二者的区别在于基表的选择。右联接应该使用 right (outer) join 关键字，而基表即处于该关键字右侧的数据表，对于范例 16-19 中的代码，如果使



用右联接，修改后的代码及查询结果如范例 16-21 所示。

【范例 16-21】 演示右联接的使用。

```
SQL> select e.employee_id, e.employee_name, s.salary
2  from t_employees e
3  right join t_salary s
4  on e.employee_id = s.employee_id
5  order by e.employee_id
6  /
```

EMPLOYEE_ID	EMPLOYEE_NAME	SALARY
1	金瑞	8000
1	金瑞	8000
1	金瑞	8000
2	钟君	7000
2	钟君	7000
2	钟君	7000
3	王山	7000
3	王山	7000
3	王山	7000
4	刘迪	7000
4	刘迪	7000
4	刘迪	7000
5	钟会	6000
5	钟会	6000
5	钟会	6000
6	张玉	5500
6	张玉	5500
6	张玉	5500
7	柳青	5000
7	柳青	5000
7	柳青	5000
8	胡东	4000
8	胡东	4000
8	胡东	4000
9	商乾	4000
9	商乾	4000
9	商乾	4000
10	王蒙	3000
10	王蒙	3000
10	王蒙	3000

30 rows selected

【代码说明】 `from t_employees e right join t_salary s` 用于指定 `from` 子句的数据源——表 `t_employees` 与表 `t_salary` 的右联接；`on e.employee_id = s.employee_id` 指定了右联接的条件为两个表的 `employee_id` 列相等。

分析查询结果可知，右联接的基表为表 `t_salary`，因此，对于 `t_employees` 表中 `employee_id` 为 11 的记录，最终查询结果中没有记录与之对应。

同样，该右联接可以简写为如下形式：

```
SQL> select e.employee_id, e.employee_name, s.salary
2  from t_employees e, t_salary s
3  where e.employee_id(+) = s.employee_id
```



```

4 order by e.employee_id
5 /

```

【代码说明】where e.employee_id(+) = s.employee_id 表示该查询是一个外联查询；e.employee_id(+)表明该联接的基表是另一张表——t_salary。

16.4.5 外联接——完全联接

完全联接实际是一个左联接和右联接的组合，即首先执行一个左联接，然后执行一个右联接，最后将两个结果集执行 union 操作（union 操作会消除重复记录），从而获得最终的数据源。

【范例 16-22】演示完全联接的使用。

```

SQL> select e.employee_id, e.employee_name, s.salary
2 from t_employees e full join t_salary s
3 on e.employee_id = s.employee_id
4 order by e.employee_id
5 /

```

【代码说明】from t_employees e full join t_salary s 指定 from 子句的最终数据源为表 t_employees 与表 t_salary 的完全外联接；on e.employee_id = s.employee_id 指定外联接的联接条件为两个表的 employee_id 列值相等。

范例 16-22 的代码实际执行过程如下：

(1) 实现表 t_employees 和表 t_salary 关于列 employee_id 列的左联接。

```

SQL> select *
2 from t_employees e left join t_salary s
3 on e.employee_id = s.employee_id
4 /

```

employee_id	employee_name	work_years	status	salary_id	employee_id	month	salary
1	金瑞	6	ACT	1	1	1 月	8000
2	钟君	6	ACT	2	2	1 月	7000
3	王山	6	ACT	3	3	1 月	7000
4	刘迪	5	ACT	4	4	1 月	7000
5	钟会	4	ACT	5	5	1 月	6000
6	张玉	4	ACT	6	6	1 月	5500
7	柳青	4	ACT	7	7	1 月	5000
8	胡东	4	ACT	8	8	1 月	4000
9	商乾	4	ACT	9	9	1 月	4000
10	王蒙	2	ACT	10	10	1 月	3000
1	金瑞	6	ACT	11	1	2 月	8000
2	钟君	6	ACT	12	2	2 月	7000
3	王山	6	ACT	13	3	2 月	7000
4	刘迪	5	ACT	14	4	2 月	7000
5	钟会	4	ACT	15	5	2 月	6000
6	张玉	4	ACT	16	6	2 月	5500
7	柳青	4	ACT	17	7	2 月	5000
8	胡东	4	ACT	18	8	2 月	4000
9	商乾	4	ACT	19	9	2 月	4000
10	王蒙	2	ACT	20	10	2 月	3000
1	金瑞	6	ACT	21	1	3 月	8000
2	钟君	6	ACT	22	2	3 月	7000
3	王山	6	ACT	23	3	3 月	7000
4	刘迪	5	ACT	24	4	3 月	7000
5	钟会	4	ACT	25	5	3 月	6000
6	张玉	4	ACT	26	6	3 月	5500

7	柳青	4	ACT	27	7	3月	5000
8	胡东	4	ACT	28	8	3月	4000
9	商乾	4	ACT	29	9	3月	4000
10	王蒙	2	ACT	30	10	3月	3000
11	周兵	0	ACT				



狂想：在执行左联接时，SELECT 语句所捕获的列为两个表的所有列，而非 e.employee_id, e.employee_name, s.salary 这三列。

(2) 实现表 t_employees 和表 t_salary 关于列 employee_id 列的右联接。

```
SQL> select *
2 from t_employees e right join t_salary s
3 on e.employee_id = s.employee_id
4 /
```

employee_id	employee_name	work_years	status	salary_id	employee_id	month	salary
1	金瑞	6	ACT	1	1	1月	8000
2	钟君	6	ACT	2	2	1月	7000
3	王山	6	ACT	3	3	1月	7000
4	刘迪	5	ACT	4	4	1月	7000
5	钟会	4	ACT	5	5	1月	6000
6	张玉	4	ACT	6	6	1月	5500
7	柳青	4	ACT	7	7	1月	5000
8	胡东	4	ACT	8	8	1月	4000
9	商乾	4	ACT	9	9	1月	4000
10	王蒙	2	ACT	10	10	1月	3000
1	金瑞	6	ACT	11	1	2月	8000
2	钟君	6	ACT	12	2	2月	7000
3	王山	6	ACT	13	3	2月	7000
4	刘迪	5	ACT	14	4	2月	7000
5	钟会	4	ACT	15	5	2月	6000
6	张玉	4	ACT	16	6	2月	5500
7	柳青	4	ACT	17	7	2月	5000
8	胡东	4	ACT	18	8	2月	4000
9	商乾	4	ACT	19	9	2月	4000
10	王蒙	2	ACT	20	10	2月	3000
1	金瑞	6	ACT	21	1	3月	8000
2	钟君	6	ACT	22	2	3月	7000
3	王山	6	ACT	23	3	3月	7000
4	刘迪	5	ACT	24	4	3月	7000
5	钟会	4	ACT	25	5	3月	6000
6	张玉	4	ACT	26	6	3月	5500
7	柳青	4	ACT	27	7	3月	5000
8	胡东	4	ACT	28	8	3月	4000
9	商乾	4	ACT	29	9	3月	4000
10	王蒙	2	ACT	30	10	3月	3000

(3) 实现左联接和右联接的 union 操作。

```
SQL> select *
2 from t_employees e left join t_salary s
3 on e.employee_id = s.employee_id
4 union
5 select *
6 from t_employees e right join t_salary s
7 on e.employee_id = s.employee_id
```



在本例中，左联接与右联接的 **union** 操作返回的结果集与左联接返回的结果集相同。

(4) 实现最终的选择语句和筛选条件，查询结果如下所示。

EMPLOYEE_ID	EMPLOYEE_NAME	SALARY
1	金瑞	8000
1	金瑞	8000
1	金瑞	8000
2	钟君	7000
2	钟君	7000
2	钟君	7000
3	王山	7000
3	王山	7000
3	王山	7000
4	刘迪	7000
4	刘迪	7000
4	刘迪	7000
5	钟会	6000
5	钟会	6000
5	钟会	6000
6	张玉	5500
6	张玉	5500
6	张玉	5500
7	柳青	5000
7	柳青	5000
7	柳青	5000
8	胡东	4000
8	胡东	4000
8	胡东	4000
9	商乾	4000
9	商乾	4000
9	商乾	4000
10	王蒙	3000
10	王蒙	3000
10	王蒙	3000
11	周兵	

31 rows selected

由于完全联接将首先实现左关联，然后实现右关联，最后再进行 **union** 操作，因此完全联接的开销很大。除非必要，否则，尽量避免使用完全关联。



16.5 层次化查询

在讲述了 Oracle 中的基本查询之后，本节着重讲述一种特殊查询——层次化查询。数据库中的数据具有平面的特点，即每条记录都是独立存在的、相互间的关系是平等和并列的。而层次化是指同一个数据表中多条数据之间存在着父子关系，进而形成了树状结构。例如，在数据表 **market** 中存储了关于市场的信息。

```
SQL> select * from market;
```

MARKET_ID	MARKET_NAME	PARENT_MARKET_ID
1	全球	0

2	亚洲	1
3	欧洲	1
4	美洲	1
5	中国	2
6	韩国	2
7	朝鲜	2
8	英国	3
9	德国	3
10	法国	3
11	美国	4
12	墨西哥	4
13	巴西	4
14	北京	5
15	天津	5
16	上海	5

16 rows selected

在表 market 中,列 market_id 标识了当前 market 的 ID;而 market_name 则标识了当前 market 的名称;parent_market_id 则标识了上级 market 的 ID。例如,market_id 为 13 的记录,名称为“巴西”,其 parent_market_id 为 4——代表“美洲”;market_id 为 4 的记录,其 parent_id 为 1——“全球”。该实例演示了表中数据之间的“父子”关系,而这种“父子”关系,往往被形象地形容为“树”,表中的每条记录都可看做树的节点。

16.5.1 利用 connect by 进行层次化查询

Oracle 9i 及以后版本都提供了 connect by 查询。connect by 可以按照某种规则,来获得节点路径上的节点集合。

【范例 16-23】演示如何查询“北京”的所有父级市场。

```
SQL> select * from market
2 start with market_id = 14
3 connect by prior parent_market_id = market_id;
```

MARKET_ID	MARKET_NAME	PARENT_MARKET_ID
14	北京	5
5	中国	2
2	亚洲	1
1	全球	0/

MARKET_ID	MARKET_NAME	PARENT_MARKET_ID
14	北京	5
5	中国	2
2	亚洲	1
1	全球	0

【代码说明】该查询用于获得“北京”市场的所有“父级”市场;start with market_id = 14 表示从 market_id 为 14 的记录开始进行查询;connect by 用于建立层级关系,递归获得所有记录。

该过程实际是一个递归的过程。当前记录,通过某种条件,来获得下一条记录。新获得的记录,再次利用相同的条件来获得新的记录,直至不再满足条件。在范例 16-23 中所使用的条



件为 `prior parent_market_id=market_id`。其中，被 `prior` 修饰的列，表示已获得的记录，而没有使用 `prior` 修饰的列是指尝试获取的新记录。`prior parent_market_id = market_id` 表示，从 `start with` 所指定的记录开始，获取新记录，新记录的 `market_id` 等于旧记录的 `parent_market_id`。

对于 `connect by` 来说，关键字 `prior` 是必不可少的，否则，将无法执行递归查询动作。但是，无论递归是否正常执行，结果集合中，都至少包含 `start with` 条件所获得记录。

【范例 16-24】 演示不含 `prior` 关键字的 `connect by` 查询。

```
SQL> select * from market
  2 start with market_id = 14
  3 connect by parent_market_id = market_id
  4 /
```

MARKET_ID	MARKET_NAME	PARENT_MARKET_ID
14	北京	5

【代码说明】在 `connect by` 的子句中直接使用了 `parent_market_id = market_id`，并未包含 `prior` 关键字，但是仍然获得了 `start with` 所指定的 `market_id` 等于 14 的记录。

`prior` 关键字仅仅用来标识记录的前后关系，因此，`prior` 关键字可以置于比较运算符“=”之后。范例 16-25 将实现与范例 16-23 相同的功能。

【范例 16-25】 演示 `prior` 关键字仅用来修饰记录的前后关系。

```
SQL> select * from market
  2 start with market_id = 14
  3 connect by market_id = prior parent_market_id
  4 /
```

MARKET_ID	MARKET_NAME	PARENT_MARKET_ID
14	北京	5
5	中国	2
2	亚洲	1
1	全球	0

【代码说明】`connect by prior parent_market_id = market_id` 与 `connect by market_id = prior parent_market_id` 是相同的条件。

对于 `connect by` 指定的递归条件，可以像 `where` 子句中的条件一样，使用 `and` 或者 `or` 等运算符来指定多个条件。在范例 16-25 中，获得了自 `market_id` 等于 14 至顶级市场的信息。现欲将市场控制在洲级以下，则可以为 `connect by` 的递归条件添加新的限制，代码修改如下：

```
SQL> select * from market
  2 start with market_id = 14
  3 connect by market_id = prior parent_market_id and instr(market_name,'洲') = 0
  4 /
```

MARKET_ID	MARKET_NAME	PARENT_MARKET_ID
14	北京	5
5	中国	2

【代码说明】`connect by market_id = prior parent_market_id and instr(market_name,'洲') = 0` 用于指定递归条件——当前记录的 `market_id` 等于上一条记录的 `parent_market_id`，并且当前记录的 `market_name` 的列值中不含字符“洲”。因此，最终的获得的结果集合仅为“城市”和“国家”级别。



16.5.2 connect by 的使用场景

表 `market` 可以看做一个树状节点集合。每个市场都可以尝试请求自己的“父市场”和“子市场”。现有一个名为 `market_customer` 的数据表，其结构和数据如下：

```
SQL> select * from market_customer;
```

CUSTOMER_ID	CUSTOMER_NAME	CUSTOMER_ADDRESS	MARKET_ID
1	Air USA	Waston	11
2	飞卢财经	北京	5
3	晓金公司	北京	14
4	中国五金	天津	15
5	申业公司	上海	16

在表 `market_customer` 中存储了 `customer` 的基本信息，并且指定了客户所属的市场 ID。这其中每个客户所使用的市场 ID 并不同级，例如，第二条记录所属的市场实际为“中国”，而第 3~5 条记录所属的市场分别为“北京”、“天津”和“上海”。此时，如果用户在应用程序的客户端输入“中国”作为查询条件，一般说来，查询语句如下：

```
SQL> select * from market_customer
2 where market_id in (
3 select market_id from market
4 where market_name = '中国'
5 )
6 /
```

CUSTOMER_ID	CUSTOMER_NAME	CUSTOMER_ADDRESS	MARKET_ID
2	飞卢财经	北京	5

但是，此处的查询结果，并不符合用户的初衷，因为，用户希望查看到所有中国市场的客户信息。因此，范例 16-26 所示的 SQL 语句更能反映用户的需求。

【范例 16-26】 演示如何利用 `connect by` 获得所有中国市场的客户信息。

```
SQL> select * from market_customer
2 where market_id in (
3 select market_id from market
4 start with market_name = '中国'
5 connect by prior market_id = parent_market_id
6 )
7 /
```

CUSTOMER_ID	CUSTOMER_NAME	CUSTOMER_ADDRESS	MARKET_ID
2	飞卢财经	北京	5
3	晓金公司	北京	14
4	中国五金	天津	15
5	申业公司	上海	16

【代码说明】 `start with market_name = '中国'` 指定了 `connect by` 查询的起始节点；`connect by prior market_id = parent_market_id` 则用于递归获得其他记录——新记录的 `parent_market_id` 等于上一条记录的 `market_id`，最终将获得中国市场及其下的所有子市场信息。

查询结果表明，利用 `connect by` 已经获得了用户真正需要的信息——所有属于中国市场的客户信息。



16.5.3 sys_connect_by_path()函数的使用

sys_connect_by_path()函数与 connect by 子句有着非常密切的关系。只有含有 connect by 子句的查询中才可以使用 sys_connect_by_path()函数。connect by 子句为每条起始记录,通过递归条件,生成一个结果集合,而 sys_connect_by_path()则可以将这些结果集合的列值串联为字符串。

【范例 16-27】 演示 sys_connect_by_path 函数的使用。

```
SQL> select market_id, market_name, sys_connect_by_path(market_name, '/')
market_path
2 from market
3 start with market_name = '天津'
4 connect by prior parent_market_id = market_id
5 /
```

MARKET_ID	MARKET_NAME	MARKET_PATH
15	天津	/天津
5	中国	/天津/中国
2	亚洲	/天津/中国/亚洲
1	全球	/天津/中国/亚洲/全球

【代码说明】 start with market_name = '天津', 指定递归查询的起始点为市场名称为“天津”的记录; connect by prior parent_market_id = market_id 则用于递归获得其他记录——新记录的 parent_market_id 等于上一条记录的 market_id; sys_connect_by_path(market_name, '/')用于将起始记录至当前记录的 market_name 列值进行串联,并使用“/”作为分隔符。



注意: sys_connect_by_path 函数返回的字符串中,第一个字符即为分隔符。

范例 16-28 演示了如何获得“天津”市场至“全球”顶级市场的完整路径。

【范例 16-28】 演示如何利用 max()和 sys_connect_by_path()函数获得完整路径。

```
SQL> select max(market_path) from (
2 select sys_connect_by_path(market_name, '/') market_path
3 from market
4 start with market_name = '天津'
5 connect by prior parent_market_id = market_id
6 /
```

```
MAX(MARKET_PATH)
-----
/天津/中国/亚洲/全球
```

【代码说明】 max(market_path)用于获得 market_path 的最大值,对于字符串类型,Oracle 总是按照字母表顺序进行比较大小的操作,因此,将返回“天津”市场至“全球”顶级市场的完整路径。



16.6 本章实例

Oracle 查询中,分页查询是一个常用查询操作。例如,在表 students 中含有 10 条记录,如下所示:

```
SQL> select * from students;
```


STUDENT_ID	STUDENT_NAME	STUDENT_AGE
1	金瑞	19
2	钟君	19
3	王山	19
4	刘迪	19
5	钟会	19
6	张玉	19
7	柳青	18
8	胡东	18
9	商乾	17
11	王蒙	19

10 rows selected

【范例 16-29】演示 Oracle 中的分页查询。

现欲以 student_name 列的字母表顺序进行排列, 并进行分页, 每页含有 5 条数据。实现分页, 则必须借助于 Oracle 的 rownum 函数, 例如为了获得第 2 页的内容, 尝试使用如下代码。

```
SQL> select rownum, s.* from students s order by student_name;
```

ROWNUM	STUDENT_ID	STUDENT_NAME	STUDENT_AGE
8	8	胡东	18
1	1	金瑞	19
4	4	刘迪	19
7	7	柳青	18
9	9	商乾	17
10	11	王蒙	19
3	3	王山	19
6	6	张玉	19
5	5	钟会	19
2	2	钟君	19

10 rows selected

从查询结果可以看出, 对于带有 order by 子句的查询语句来说, 首先获得 rownum 的值, 然后再进行排序输出。为了获得 student_name 排序, 并且处于第 2 页的所有记录, 应该首先获得排序之后的数据, 然后再获得 rownum 的值。

```
SQL> select * from (
2  select rownum rn, s.* from (
3  select * from students order by student_name) s
4  ) where rn>5 and rn<=10
5  /
```

RN	STUDENT_ID	STUDENT_NAME	STUDENT_AGE
6	11	王蒙	19
7	3	王山	19
8	6	张玉	19
9	5	钟会	19
10	2	钟君	19

【代码说明】select * from students order by student_name)用于获得一个内嵌视图, 该内嵌视图是将表 students 中的记录按照 student_name 进行排序之后的结果集合; select rownum rn, s.*



则从内嵌视图 s 中获得所有列的数据及此时每条记录的 rownum; where rn>5 and rn<=10 用于获得第 2 页的数据。

对于大数据量的表, 使用以下代码进行优化:

```
SQL> select * from (
2  select rownum rn, s.* from (
3  select * from students order by student_name) s
4  where rownum<=10
5  ) where rn>5
6  /
```

RN	STUDENT_ID	STUDENT_NAME	STUDENT_AGE
6	11	王蒙	19
7	3	王山	19
8	6	张玉	19
9	5	钟会	19
10	2	钟君	19

【代码说明】内层 SQL 中的 rownum<=10 用于限制当前记录集合中 rownum<=10 的所有记录; 以缩小外层 SQL 中所扫描的记录集合的大小。这里需要注意的是, 只能将 rownum<=10 置于内层 SQL, 而 rn>5 只能置于外层 SQL, 二者的位置不能颠倒。这是因为 rownum 总是从 1 开始的, 如果在内层 SQL 中使用 rownum>5 的条件是无法获得任何记录的。



16.7 本章小结

本章讲述了常用的查询语句, 并按照复杂程度, 采用从最简单的查询语句开始, 依次增加各功能子句的方式, 剖析了各子句的作用。对于复杂的查询语句, 最重要的是理解其查询所执行的步骤和各子句执行的顺序。另外, 对于结果集之间的操作进行了详细的介绍, 尤其需要注意区分 union 和 union all 的细微差别。对于设计上保证不存在重复记录的集合运算, 应该使用 union all 以保证高效率。

联接也是本章的重点内容, 本章通过多个实例详细讲解了各种联接的用法及执行过程, 读者在使用过程中应该进行甄别, 并使用合适的联接方式。尽管有时候简化方式, 如使用(+), 会简化代码, 但在查询较为复杂时, 往往会造成代码的可读性较差, 因此, 读者也应当根据自己的习惯进行取舍。

在本章的最后, 介绍了层次化查询 connect by 的使用。对于 connect by 查询, 最重要的是理解 prior 关键字的真实意义。另外, connect by 查询还提供了其他伪列, 例如 level。这些伪列的使用比较简单, 读者可以查看 Oracle 的相关资料进行学习。



16.8 习题

1. 在表 students 中, 如何获取同名学生的信息?
2. 简述 union 与 union all 的区别。
3. 利用外联接从表 students 与表 student_course 中获得学生与所选课程的信息。
4. 利用 connect by 获得中国市场下的所有子市场。



第 17 章 SQL 更新数据

Oracle 中可以利用 DML 更新数据。其 DML 语句与其他数据库的 SQL 语法完全一致——都遵守了工业标准。与查询操作不同，更新数据将导致数据库状态的变化，因此，Oracle 同样提供了提交与回滚操作来保证数据库状态的一致性。Oracle 常见的更新操作包括：

- 插入数据；
- 修改数据；
- 删除数据。

本章将系统讲述 Oracle 以上三种更新数据操作，并讲述数据更新的提交与回滚操作。



17.1 插入数据

插入数据即向数据表中插入新的记录，插入数据应该使用 `insert` 命令。插入数据的主要途径包括：通过指定各列的值直接插入、通过子查询插入、通过视图插入等。对于通过视图插入的方式，大多数应该使用 `instead of` 触发器来进行处理，因此，本章将着重讲述前两种插入方式。

17.1.1 insert 语句向表中插入数据

利用 `insert` 语句可以直接向表中插入新的数据。例如，创建一个如下结构的数据表：

```
SQL> create table people (id number primary key, name varchar2(20) not null, status
varchar2(3));
```

Table created

【代码说明】表 `people` 中，列 `id` 为主键，数值型；列 `name` 为可变字符串类型，非空；列 `status` 为可变字符串类型。

创建用于生成该表主键的序列：

```
SQL> create sequence people_seq minvalue 1 maxvalue 99999 start with 1 increment
by 1;
```

`insert` 语句插入数据表有两种方式，一是为所有列显式赋值，这种方式下不需要罗列列名，其语法规则如下：

```
insert into table_name values (value1, value2,...)
```

【范例 17-1】演示如何为所有列显式赋值，以插入新的数据。

```
SQL> insert into people values(people_seq.nextval, '张文', 'ACT');
```

1 row inserted

【代码说明】`insert into` 命令用于向数据表中插入数据；`people` 指定了要插入的数据表；`values(people_seq.nextval, '张文', 'ACT')` 用于指定表中各列的值，这里罗列了所有列的值。

查询此时表中数据：



21 天学通 Oracle

```
SQL> select * from people;
```

ID	NAME	STATUS
1	张文	ACT

当数据表的列非常多时，而有的列值又不是必需的，此时可以同时指定要插入的列名列表和列值列表。其语法规则如下所示：

```
insert into table_name (column1, column2,...) values (value1, value2,...)
```

【范例 17-2】 演示如何同时指定列名列表和列值列表，以插入新的数据。

```
SQL> insert into people (id, name) values(people_seq.nextval, '柳平');
```

```
1 row inserted
```

【代码说明】(id, name) 指定了要插入的新数据的列名列表；values(people_seq.nextval, '柳平')用于指定要插入的新数据的列值列表。

此时，查询表 people 中的数据：

```
SQL> select * from people;
```

ID	NAME	STATUS
1	张文	ACT
2	柳平	



注意：当使用第二种方式插入数据时，列名列表和列值列表必须保持一致，即每个列的数据类型和实际插入类型保持一致。

17.1.2 利用子查询批量插入数据

Oracle 可以利用子查询向表中批量插入数据。此时的 SQL 语句除了包含 insert into 命令之外，还应该包含一个查询语句。其语法规则如下所示：

```
insert into table_name select ...
```

【范例 17-3】 利用子查询批量插入数据。

```
SQL> insert into people select employee_id, employee_name, status from t_employees  
where employee_id>=3;
```

```
9 rows inserted
```

【代码说明】insert into people 用于向表 people 中插入新的数据；select employee_id, employee_name, status from t_employees where employee_id>=3 用于获得表 t_employees 中 employee_id 大于等于 3 的所有记录；子查询处于 insert into 语句之后，可以将该子查询所获得的所有记录插入到 insert into 语句所指定的数据表中。

此时，查询表 people 中的数据：

```
SQL> select * from people;
```

ID	NAME	STATUS
1	张文	ACT
2	柳平	
3	王山	ACT
4	刘迪	ACT

5	钟会	ACT
6	张玉	ACT
7	柳青	ACT
8	胡东	ACT
9	商乾	ACT
10	王蒙	ACT
11	周兵	ACT

11 rows selected

17.1.3 insert 语句与默认值

当向数据表中插入数据时，如果表中某列含有默认值约束，对于该列又没有显式赋值，那么默认值将作为列值进行插入。

【范例 17-4】利用 insert 语句插入默认值。

首先为表 people 的 status 列添加默认值：

```
SQL> alter table people modify (status varchar2(3) default 'ACT');
```

Table altered

向表 people 中插入新的数据，但不为列 status 显式赋值：

```
SQL> insert into people (id, name) values (12, '殷商');
```

1 row inserted

查看表中的数据：

```
SQL> select * from people where id = 12;
```

ID	NAME	STATUS
12	殷商	ACT

分析查询结果可知，status 列虽未显式赋值，但是仍然被插入数据“ACT”。

17.1.4 insert 语句与唯一性约束

当使用 insert 语句时，需要注意的是唯一性约束。当插入的列值违反了唯一性约束时，Oracle 将抛出错误。例如，表 people 中，列 id 为表的主键，尝试向其中插入数据：

```
SQL> insert into people values(11, '张三', 'ACT');
```

```
insert into people values(11, '张三', 'ACT')
```

ORA-00001: unique constraint (SYSTEM.SYS_C005393) violated

对于数值型主键来说，在插入数据时，使用序列来获得主键值是一个好的选择。

17.1.5 insert 语句与外键约束

insert 语句更新数据表时，同样会引起外键约束的检查。在表 t_salary 上建立外键约束，如下所示：

```
SQL> alter table t_salary add
2 constraint fk_employee_id foreign key (employee_id)
3 references t_employees (employee_id)
4 /
```



Table altered

向表 t_salary 中插入新的数据:

```
SQL> insert into t_salary (salary_id, employee_id, month, salary)
      2 values (31, 20, '1月', 5000);
```

```
insert into t_salary (salary_id, employee_id, month, salary)
values (31, 20, '1月', 5000)
```

ORA-02291: integrity constraint (SYSTEM.FK_EMPLOYEE_ID) violated - parent key not found

【代码说明】尝试向表 t_salary 中插入的数据 employee_id 列值为 20，但是该 employee_id 的值破坏了外键约束。



17.2 修改数据

像其他数据库一样，Oracle 使用 update 命令来修改数据。update 修改数据一般有以下几种情况：直接修改单列的值、直接修改多列的值、利用 where 子句限制修改范围和利用视图修改数据。利用视图修改数据往往需要利用 instead of 触发器实现，因此本节将着重讲述前三种更新方式。

17.2.1 利用 update 修改单列的值

update 可以修改单列的值。其语法规则如下所示：

```
update table_name set column = value
```

例如，为了将表 people 中的 status 列值修改为“CXL”，可以利用范例 17-5 所示的代码。

【范例 17-5】演示如何利用 update 修改单列的值。

```
SQL> update people set status = 'CXL';
```

12 rows updated

【代码说明】update people 用于发出修改命令，并指定更新的目标表为 people；set status 指定要修改的列名；更新后的目标值为“CXL”。

查询表 people 的内容，以验证更新是否成功：

```
SQL> select * from people;
```

ID	NAME	STATUS
1	张文	CXL
2	柳平	CXL
3	王山	CXL
4	刘迪	CXL
5	钟会	CXL
6	张玉	CXL
7	柳青	CXL
8	胡东	CXL
9	商乾	CXL
10	王蒙	CXL
11	周兵	CXL
12	殷商	CXL



12 rows selected

17.2.2 利用 update 修改多列的值

update 命令既可以修改单列值，也可以同时修改多列的值。其语法规则如下所示：

```
update table_name set column1 = value1, column2 = value2 ...
```

例如，有时为了合并两个表的数据，需要为其中的一个主键 id 添加一个基数，以避免两个表中主键的重复。此时，需要修改表中所有 id 的值。以表 people 为例，在修改列 id 的值的同时，也可以修改 status 列的值。

【范例 17-6】利用 update 命令同时修改多列的值。

```
SQL> update people set id = (20000+id), status = 'ACT';
```

12 rows updated

【代码说明】update people 指定要修改的目标为表 people；set id = (20000+id) 将列 id 的值增加基数 20000；status = 'ACT' 同时修改 status 列值为“ACT”。

查询表 people 的内容，以验证更新是否成功：

```
SQL> select * from people;
```

ID	NAME	STATUS
20001	张文	ACT
20002	柳平	ACT
20003	王山	ACT
20004	刘迪	ACT
20005	钟会	ACT
20006	张玉	ACT
20007	柳青	ACT
20008	胡东	ACT
20009	商乾	ACT
20010	王蒙	ACT
20011	周兵	ACT
20012	殷商	ACT

12 rows selected

17.2.3 利用 where 子句限制修改范围

where 子句是 update 命令最常用的子句。不使用 where 子句的 update 命令是不安全的。因为不使用 where 子句将一次性修改表中所有记录，这将带来极大的安全隐患。为了将表 people 中，id 大于 20010 的 status 列修改为“CXL”则可以利用如范例 17-7 所示的 SQL 语句。

【范例 17-7】利用 where 子句限制修改范围。

```
SQL> update people set status = 'CXL' where id > 20010;
```

2 rows updated

【代码说明】update people set status = 'CXL' 用于将表 people 中的 status 列值修改为“CXL”；where id > 20010 用于限制修改范围——id 列值大于 20010。

查询表 people 的内容，以验证更新是否成功：

ID	NAME	STATUS
----	------	--------



```
20011    周兵      CXL
20012    殷商      CXL
```

```
2 rows selected
```

当然，也可以利用较为复杂的条件来限制修改范围。

【范例 17-8】 利用 in 判式作为条件，限制数据修改范围。

```
SQL> update people set status = 'CXL'
      2 where name in (select employee_name from t_employees)
      3 /
```

```
9 rows updated
```

【代码说明】 where name in (select employee_name from t_employees) 用于限制修改条件为：表 people 的列 name 的值与表 employee 中列 employee_name 的某个值相等。



17.3 删除数据

数据删除的目标是数据表中的记录，而不是针对列来进行的。删除数据应该使用 delete 命令或者 truncate 命令。其中 delete 命令的作用目标是表中的某些记录，而 truncate 命令的作用目标是整个数据表。

17.3.1 用 delete 命令删除数据

像 update 命令一样，delete 命令经常与 where 子句一起出现，以删除数据表中的某些数据。其语法规则如下所示：

```
delete from table_name where ...
```

【范例 17-9】 演示 delete 命令如何与 where 子句结合，删除数据表中部分数据。

```
SQL> delete from people p
      2 where exists(select 1 from t_employees e where e.employee_name = p.name)
      3 /
```

```
9 rows deleted
```

【代码说明】 delete 命令用于删除表中数据；from people p 用于指定删除的目标表为 people，并指定该表的别名为 p；where exists(select 1 from t_employees e where e.employee_name = p.name) 用于指定删除记录的过滤条件——在表 t_employees 中存在着一记录，该记录的 employee_name 列值等于表 people 的当前记录的名称列值；该删除语句用于保证表 people 中，所有的姓名不再存在于表 t_employees 中。



注意： delete 命令是针对表中的整条记录，因此，其后不需要指定列名或者*。例如，执行 delete * from people，Oracle 将抛出错误提示。

```
SQL> delete * from people;

delete * from people

ORA-00903: invalid table name
```

【代码说明】 该 SQL 语句将抛出非法表名的错误提示。此处，Oracle 将*解析为表名，这是因为 Oracle 的 delete 命令允许省略 from 关键字，如下代码将实现与范例 17-9 相同的作用：



```
SQL> delete people p
      2 where exists(select 1 from t_employees e where e.employee_name = p.name)
      3 /

0 rows deleted
```

【代码说明】delete people p 相当于 delete from people p; 因为范例 17-9 已经删除了 where 子句所限制的记录，因此，该语句删除的记录行数为 0。

17.3.2 用 truncate 命令删除数据

truncate 命令删除数据和 delete 命令删除数据主要有三点不同。

- truncate 命令属于 DDL（数据库定义语言）范畴，而 delete 命令是 DML（数据库操作语言）范畴。
- truncate 命令将一次性删除数据表的所有数据，而 delete 语句将对数据表中所有记录进行循环处理。
- truncate 命令删除的数据将不能回滚，而 delete 语句在提交修改之前，仍然可以回滚操作。

truncate 命令的语法规则如下：

```
truncate table table_name
```

【范例 17-10】利用 truncate 命令删除数据表数据。

```
SQL> truncate table people;

Table truncated
```

【代码说明】由于 truncate 命令属于 DDL，因此 truncate 命令之后为 table 关键字，表示该命令的作用目标是一个数据表。

在命令执行成功之后，查看表 people 中的数据：

```
SQL> select * from people;

   ID      NAME      STATUS
-----

```



17.4 数据提交与回滚

Oracle 中有回滚段的概念。Oracle 中的回滚段是指当 DML 修改数据库时，用于存储原数据影像的存储空间。当 DML 修改数据库中的数据（例如，update 和 delete 命令）之后，执行提交之前，如果执行了回滚操作，Oracle 将利用回滚段中的数据影像将数据库恢复到修改前的状态。

17.4.1 回滚动作

回滚动作有两种情况，一是用户在提交动作之前，手动执行 rollback 命令，以放弃该事务对数据库的修改；二是事务执行失败，数据库自动执行 rollback 命令，来恢复事务对数据库的修改。

【范例 17-11】演示事务的手动回滚。

查询数据表 people 的当前状态：

```
SQL> select * from people;
```



ID	NAME	STATUS
----	------	--------

向表 **people** 中插入新的数据，并执行查询动作：

```
SQL> insert into people values (1, '周璇', 'ACT');
```

```
1 row inserted
```

```
SQL> select * from people;
```

ID	NAME	STATUS
1	周璇	ACT

利用 **rollback** 命令回滚插入操作：

```
SQL> rollback;
```

```
Rollback complete
```

再次查看表中数据：

```
SQL> select * from people;
```

ID	NAME	STATUS
----	------	--------

可见，当使用 **rollback** 命令回滚插入操作之后，数据表将恢复到修改前的状态。

17.4.2 提交动作

提交动作是指将数据库的修改操作反映到数据库，不再允许使用回滚操作。

【范例 17-12】 演示如何进行事务提交。

```
SQL> insert into people values (1, '周璇', 'ACT');
```

```
1 row inserted
```

```
SQL> commit;
```

```
Commit complete
```

【代码说明】 **insert into people values (1, '周璇', 'ACT')** 用于向表 **people** 中插入新的数据；**commit** 命令用于提交事务。

此时查询表 **people** 中的数据：

```
SQL> select * from people;
```

ID	NAME	STATUS
1	周璇	ACT

尝试执行回滚操作：

```
SQL> rollback;
```

```
Rollback complete
```

在回滚动作成功执行之后，再次查询表 **people** 中的数据：

```
SQL> select * from people;
```

ID	NAME	STATUS
----	------	--------



1 周璇 ACT

分析查询结果可知，在事务提交之后，rollback 命令将无法回滚数据库的修改。rollback 失效的原因在于回滚段信息已经被释放。

17.4.3 PL/SQL Developer 中的回滚与提交

在 PL/SQL 中，同样可以利用图形化界面来处理回滚与提交动作。图 17-1 演示了在执行修改数据库操作之前的提交/回滚按钮。



图 17-1 执行数据库操作之前，回滚提交按钮不可用

在 PL/SQL Developer 中的【Command Window】中执行以下语句：

```
insert into people values (2, '刘祥', 'ACT');
```

此时，PL/SQL Developer 中的提交/回滚按钮将变为可用状态，如图 17-2 所示。

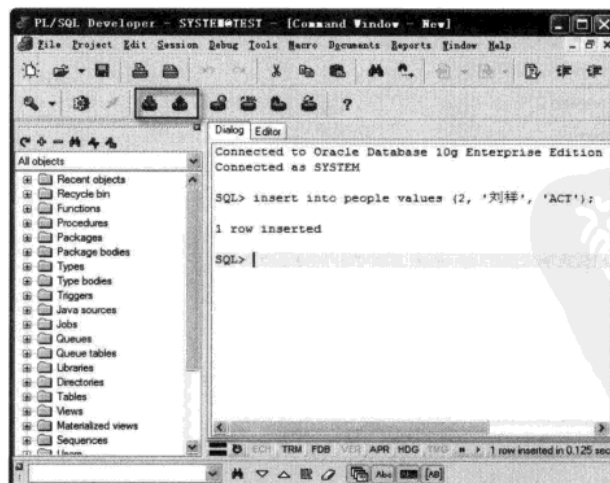


图 17-2 执行数据库修改之后，提交/回滚按钮将变为有效状态



查询此时的数据表状态，如图 17-3 所示。

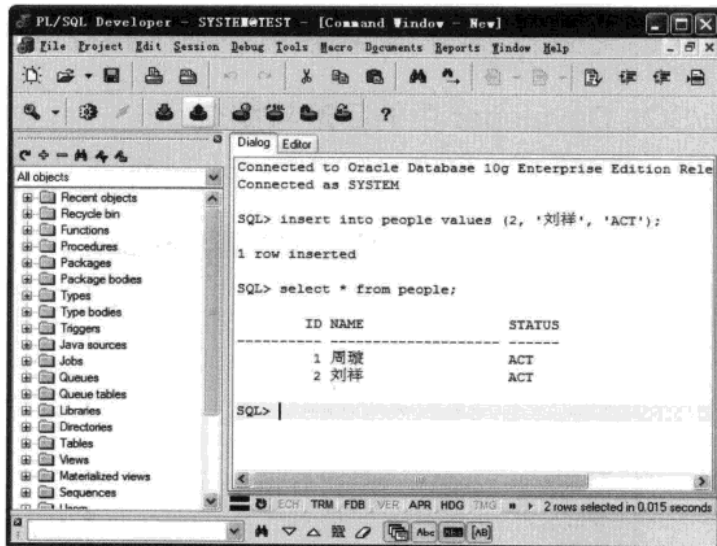


图 17-3 插入数据后，表 PEOPLE 的状态

此时，单击回滚按钮，可以回滚数据表的修改，如图 17-4 所示。

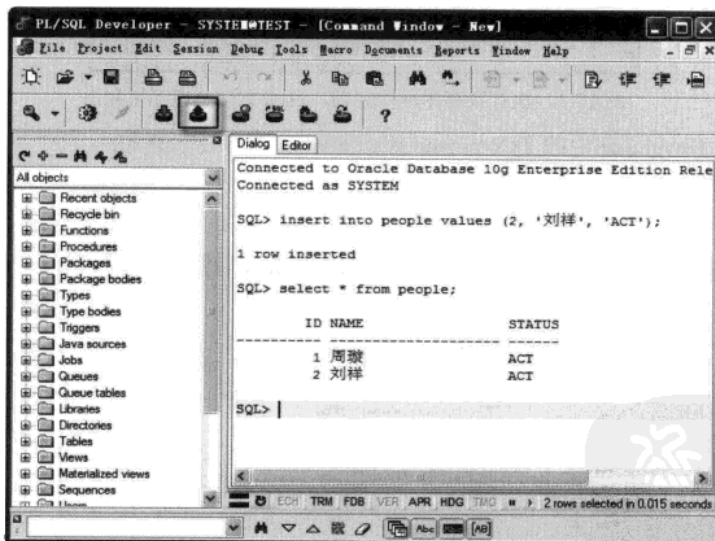


图 17-4 单击回滚按钮，以回滚数据库修改

回滚数据之后，会发现提交回滚按钮均被置为不可用状态，而数据表恢复到修改之前的状态，如图 17-5 所示。

如果在修改数据表之后，单击提交按钮，那么将无法执行回滚操作，读者可以自行试验。

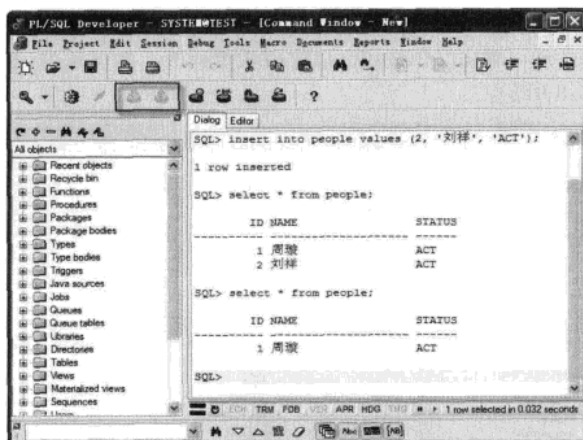


图 17-5 单击回滚按钮之后的数据表状态



17.5 本章实例

本章重点讲述了如何使用命令进行数据更新。但是很多时候，开发人员使用工具直接修改数据会更加直观。例如，可以利用 PL/SQL Developer 直接进行数据修改。

【范例 17-13】 演示如何利用 PL/SQL Developer 直接进行数据修改。

(1) 创建用于测试的临时数据表 tmp_people。

```
SQL> create table test_people as select * from people;
```

Table created

(2) 新建【SQL Window】，并在其中执行 SQL 语句：

```
select * from test_people order by id for update;
```

【代码说明】 for update 用于锁定表中记录。

注意，此时 PL/SQL 工具栏中的【提交】、【回滚】按钮已经变为可用状态，表明锁定成功，如图 17-6 所示。

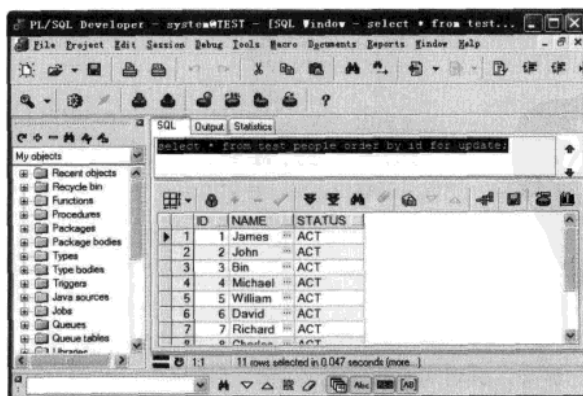





图 17-6 锁定表的记录

- (3) 单击列名（例如 ID）上方的  按钮，以直接编辑数据，如图 17-7 所示。
- (4) 更新完数据之后，单击  按钮，以完成数据修改，然后单击  按钮，使数据恢复到只读状态，如图 17-8 所示。

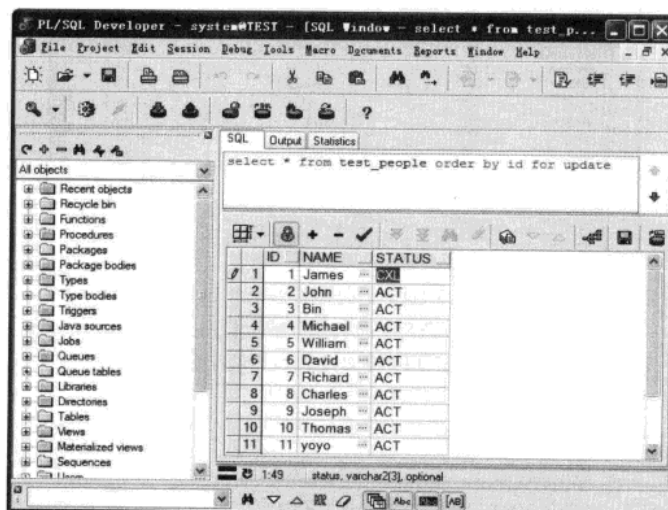


图 17-7 将列 status 的值，直接修改为“CXL”

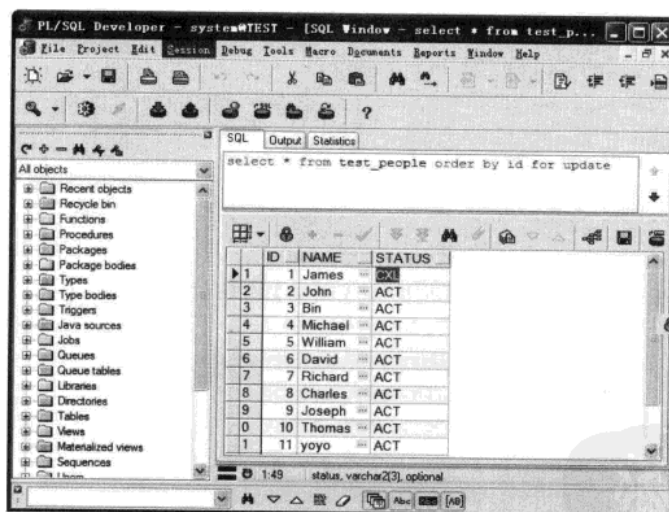




图 17-8 完成数据修改之后，将数据恢复到只读状态

- (5) 单击工具栏中的【提交】按钮，将数据修改提交到数据库。
- (6) 同样，可以为查询语句添加过滤条件，以修改部分数据，如下所示。
`select * from test_people where id<5 order by id for update`
- (7) 利用这种方式除了可以修改数据之外，还可以通过单击  按钮进行数据插入，通过单击  按钮进行数据删除，读者可自行试验。



17.6 本章小结

由于利用 `update` 更新数据表和利用 `delete` 删除表中数据，语法简单，而且为大多数读者所熟悉，所以本章只进行了简单介绍。需要注意的是利用 `truncate` 删除表中数据和利用 `delete` 删除数据的区别：`truncate` 命令是 DDL，数据删除之后，不可回滚；而 `delete` 命令是 DML，数据删除之后，可以利用 `rollback` 进行回滚。就速度而言，`truncate` 命令要快于 `delete` 命令，因此，当确定要删除表中所有数据，只保留数据表结构时，应该使用 `truncate` 命令（这往往发生在数据清理时，例如，删除测试数据）；而删除表中部分数据应该使用 `delete` 命令。



17.7 习题

1. 如何利用 `insert` 命令实现批量插入？
2. 如何利用 `update` 命令修改多列的值？
3. 简述利用 `delete` 命令与 `truncate` 命令删除数据的区别。
4. 简述提交动作与回滚动作的区别。



第四篇 Oracle 编程高级应用

第 18 章 数据库速度优化与数据完整性

对于大型应用中的数据库系统，性能和安全是两个重要的方面。这里的性能指的是数据库处理和响应客户端的速度；而保证数据完整性则是安全的体现。本章将详细讲述如何利用索引来提高数据库性能，并综合利用各种约束来保证数据完整性。本章的主要内容包括：

- 索引的原理；
- 利用索引提高数据库性能；
- 数据库完整性简介；
- 利用约束保持数据库完整性。

通过本章的学习，读者可以了解索引的工作原理及索引的应用场景，并进一步了解数约束与数据完整性的关系。



18.1 利用索引加快数据引用

在数据库系统中，索引是非常重要的一个对象，尤其是面对大型数据表时，索引能大大提高数据检索的速度。本节将介绍索引的原理及索引的使用。

18.1.1 索引的原理

索引在现实世界中最典型的例子莫过于字典检索了。用户在使用字典时，可以使用两种方式，一是逐页翻查，以获得需要查找的目标；二是根据字典的检索目录来获得目标所在的页数，然后直接在该页获得查找目标。毫无疑问，利用检索目录（索引）检索目标是更为高效的方式。

现有的主流数据库都提供了索引这一概念，Oracle 也不例外。一旦在数据表的某列上建立了索引，Oracle 将另辟新的空间，以存储该列上所有值与其记录的 rowid 的对应关系。当用户试图以索引列作为搜索条件时，Oracle 将利用索引来获得相应的 rowid，并捕获该记录。

在表 people 中含有以下记录：

```
SQL> select * from people order by id;
```

ID	NAME	STATUS
1	James	ACT
2	John	ACT
3	Robert	ACT
4	Michael	ACT
5	William	ACT
6	David	ACT
7	Richard	ACT
8	Charles	ACT
9	Joseph	ACT
10	Thomas	ACT



10 rows selected

欲获得表 `people` 中 `name` 为 `David` 的信息，可以利用如下 SQL 语句：

```
SQL> select * from people where name = 'David';
```

ID	NAME	STATUS
6	David	ACT

当 Oracle 处理该查询语句时，将执行全表扫描。当搜索到第 6 条记录时，会发现该条记录符合搜索条件，并将该记录纳入结果集合。但是搜索并不会停止，因为 Oracle 并不知道在后面的记录中是否仍然存在符合条件的记录。直至搜索完整个表，Oracle 才会返回最终的结果集合。

但是，如果预先在表 `people` 的 `name` 列上创建了一个索引，那么，搜索的顺序将完全不同。创建索引的语法如下所示：

```
create index idx_people_name
on people(name)
```

一旦索引创建，那么表中所有数据将按照字母表顺序进行分块处理，例如，以每 5 条记录作为一个数据块（当然，实际数据块将大得多）。分块后的数据结构如图 18-1 所示。

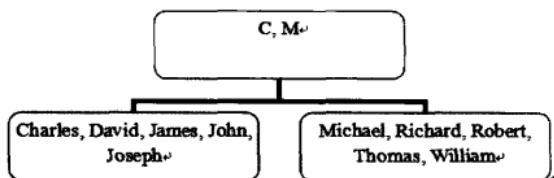


图 18-1 索引数据分块示意图

此时执行搜索语句 `select * from people where name = 'David'` 时，Oracle 只会在第一个数据块中进行搜索，因为数据库知道，第二个数据块都是 M 以后的数据。当搜索到 `David` 之后，如果下一条记录的 `name` 列的值不是 `David`，Oracle 也将停止搜索，因为列中所有值都是按照字母表顺序进行排列的，所有的“David”位置肯定是相邻的。一旦搜索到“David”，Oracle 会马上获得其对应的 `rowid`，并根据 `rowid` 快速定位该记录。

对于如下 SQL 语句，索引的作用会反映得更加清晰。

```
SQL> select * from people where name = 'Michael';
```

ID	NAME	STATUS
4	Michael	ACT

因为要搜索的条件为 `name` 列的值为“Michael”，所以，将会直接跳转到第二个数据块进行搜索，因为第二个数据块才是以 M 开头的数据。而且，当获得 `Michael` 的下一条记录为“Richard”时，将停止搜索，返回结果集合。

18.1.2 利用索引提高数据库性能

索引应用于大数据量的数据表时，将在很大程度上提高查询速度。为了演示这种情况，首先来创建一个大的数据表。视图 `dba_objects` 可以获得数据库中所有对象的基本信息：

```
select * from dba_objects
```

其中的部分数据如图 18-2 所示。



OWNER	OBJECT_NAME	SUBOBJECT_NAME	OBJECT_ID	DATA_OBJECT_ID	OBJECT_TYPE	CREATED	LAST_DDL_TIME
SYS	ICOLS	(null)	20	2	TABLE	2004-3-9 下午11:57:57	2004-3-10 上午
SYS	I_USERS	(null)	44	44	INDEX	2004-3-9 下午11:57:58	2004-3-9 下午
SYS	CODE	(null)	28	28	TABLE	2004-3-9 下午11:57:57	2004-3-10 上午
SYS	UNDO\$	(null)	15	15	TABLE	2004-3-9 下午11:57:57	2004-3-9 下午
SYS	C_CORP\$	(null)	29	29	CLUSTER	2004-3-9 下午11:57:57	2004-3-9 下午
SYS	I_ORL\$	(null)	3	3	INDEX	2004-3-9 下午11:57:57	2004-3-9 下午
SYS	PROXY_ROLE_DATAS	(null)	25	25	TABLE	2004-3-9 下午11:57:57	2004-3-9 下午
SYS	I_ORP1	(null)	39	39	INDEX	2004-3-9 下午11:57:58	2004-3-9 下午
SYS	I_ORP2	(null)	51	51	INDEX	2004-3-9 下午11:57:58	2004-3-9 下午
SYS	I_PROXY_ROLE_DATAS_1	(null)	26	26	INDEX	2004-3-9 下午11:57:57	2004-3-9 下午
SYS	FILES	(null)	17	17	TABLE	2004-3-9 下午11:57:57	2004-3-9 下午
SYS	UTIS	(null)	13	8	TABLE	2004-3-9 下午11:57:57	2004-3-9 下午
SYS	I_FILES_BLOCKS	(null)	9	9	INDEX	2004-3-9 下午11:57:57	2004-3-9 下午
SYS	I_FILES1	(null)	41	41	INDEX	2004-3-9 下午11:57:58	2004-3-9 下午
SYS	I_COM1	(null)	48	48	INDEX	2004-3-9 下午11:57:58	2004-3-9 下午

图 18-2 视图 dba_objects 的部分数据

利用该视图创建一个大的数据表：

```
SQL> create table test_objects as
2 select * from dba_objects;
```

Table created

```
SQL> select count(1) from test_objects;
```

```
COUNT(1)
-----
47736
```

对于查询语句：

```
select * from test_objects where object_name = 'PEOPLE'
```

在 PL/SQL Developer 中运行执行计划（选中查询语句，按 F5 执行），结果如图 18-3 所示。

Description	Object owner	Object name	Cost	Cardinality	Bytes
SELECT STATEMENT, GOAL = ALL_ROWS			146	7	1239
TABLE ACCESS FULL	SYSTEM	TEST_OBJECTS	146	7	1239

图 18-3 全表搜索时的执行计划

可以看出，将表 test_objects 作为普通表处理时，将执行全表搜索，其花费基数为 146。

现为该表在 object_name 列上创建索引。

```
SQL> create index idx_test_object
2 on test_objects(object_name);
```

Index created

此时，再次对同一条 SQL 语句，查看其执行计划。

```
select * from test_objects where object_name = 'PEOPLE'
```

结果如图 18-4 所示。

Description	Object owner	Object name	Cost	Cardinality	Bytes
SELECT STATEMENT, GOAL = ALL_ROWS			2	1	177
TABLE ACCESS BY INDEX ROWID	SYSTEM	TEST_OBJECTS	2	1	177
INDEX RANGE SCAN	SYSTEM	IDX_TEST_OBJECT	1	1	

图 18-4 利用索引的执行计划

分析此时的执行计划可知，搜索模式由原来的全表搜索变为利用索引进行搜索；而花费基数已由原来的 146 降为 2。这极大地提高了数据库性能。

这里需要注意的是，搜索条件中包含索引列才能真正使用到索引所带来的好处。例如，利用：

```
select * from test_objects where owner = 'SCOTT'
```



无法通过索引来提高效率，其执行计划如图 18-5 所示。

Description	Object owner	Object name	Cost	Cardinality	Bytes
SELECT STATEMENT, GOAL = ALL_ROWS			146	7	1239
TABLE ACCESS FULL	SYSTEM	TEST_OBJECTS	146	7	1239

图 18-5 搜索条件中不含索引的执行计划

可以看出，该 SQL 语句仍然使用了全表搜索的模式，花费基数仍然为 146。

另外一种情况是，搜索条件中不仅存在着主键列，而且存在着非主键列。那么，应当将主键列的条件置于最前面，将非主键列置于后面。将能够提高查询的速度。例如，查询语句：

```
select * from test_objects where object_name = 'EMP' and owner = 'SCOTT'
```

将稍快于查询语句：

```
select * from test_objects where owner = 'SCOTT' and object_name = 'EMP'
```

18.1.3 索引对 DML 的影响

到目前为止，所看到的都是索引所带来的好处。但在表上创建索引也会带来部分负面效果。例如，针对数据的添加、删除和修改都会影响到索引。

1. 添加数据对索引的影响

以表 `people` 为例，一旦在其上创建了索引，向其中插入数据时，数据库将会重新组织索引。

```
insert into people (ID, NAME, STATUS) values (11, 'Zoey', 'ACT');
```

在以上 SQL 语句中，向表 `people` 中插入名为“Kelly”的用户信息。此时数据库除了正常的插入操作开销之外，还需要为该记录添加索引项，而索引已有数据块无法满足存储要求，因此，数据库将为其分配新的存储空间，如图 18-6 所示。

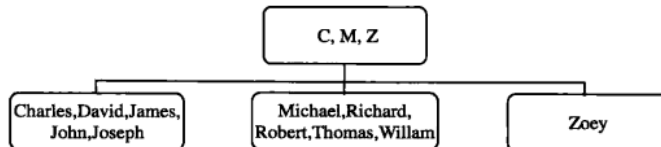


图 18-6 新插入的数据必须反映到索引中

在这里，新建的索引项似乎并未引起太多的动作，但是考虑如下 SQL 语句：

```
insert into people (ID, NAME, STATUS) values (12, 'Bill', 'ACT');
```

此时，除了要针对数据表的插入操作之外，对索引的修改将变得更加烦琐。因为“Bill”需要插入到 Charles 之后、David 之前，而巧合的是，第一个数据块已经是满的状态，那么就需要从第一个数据块中为“Bill”空出一个位置，结果将造成连锁反应，此时的索引结构应该如图 18-7 所示。

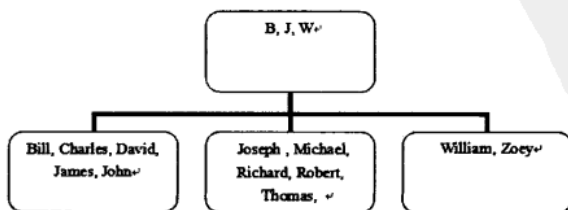


图 18-7 插入“Bill”之后，索引结构图



分析此时图中索引结构可知,该结构已经“颠覆”了原结构,其开销远远大于将索引项添加到末尾的操作。而由于插入数据的不可预知性,因此,对于插入操作来说,带有索引的表的开销,将是普通表的两倍以上。

2. 修改数据对索引的影响

修改数据的操作对索引的影响稍稍不同,例如,将表 `people` 中原名“James”修改为“Winne”,那么可以利用如下 SQL 语句。

```
update people set name = 'Winne' Where name = 'James';
```

修改数据会直接反映在数据表中,但是索引项将不会被删除。原索引项“James”只是被标记为不可用,而为“Winne”新添索引项,如图 18-8 所示。

从图中可以看出,“James”被添加了删除标记,而在最后一个数据块中添加了索引项“Winne”。Oracle 采取该策略是为了最大程度上减小数据库操作的开销。但这同时带来了空间上的浪费,因为索引项“James”仍然占用索引空间。新的索引项仍然不能够插入第一个数据块中。

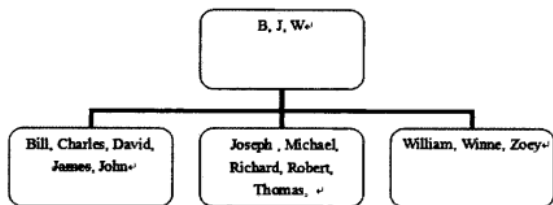


图 18-8 修改数据对索引的影响

3. 删除数据对索引的影响

删除数据对索引的影响相对简单,因为数据删除之后,相应的索引项只是被打上了删除标记,表征该索引项不可用。例如,利用如下 SQL 语句删除一条记录。

```
delete from people where name = 'Robert';
```

此时的索引项被修改为如图 18-9 所示的样子。

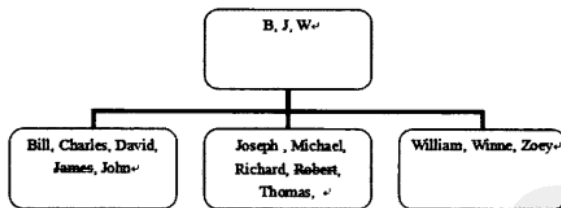


图 18-9 删除数据对索引的影响

分析此时的索引结构可知,删除操作是 DML 操作中针对索引的开销最小的一种。但是,仍然不可避免地形成了对存储空间的浪费。

18.1.4 索引的使用时机

通过前面的内容可知,索引有弊有利,因此,分析索引的使用时机十分重要。

1. 小数据量的表不宜使用索引

小数据量的表,首先要创建索引,并寻找数据块,然后才进行数据的实际搜索,因此,对



于小数据量的表，使用全表搜索往往会快于使用索引。

2. 频繁使用插入、修改、删除等 DML 操作的数据表不宜使用索引

针对表的插入、修改、删除等操作非常频繁时，将会降低数据库性能。因为每次的动作都可能会触发两倍甚至两倍以上成本来完成索引操作。



18.2 利用约束保持数据完整性

数据完整性 (Data Integrity) 是指数据的精确性 (Accuracy) 和可靠性 (Reliability)。它是应防止数据库中存在不符合语义规定的数据和防止因错误信息的输入造成无效操作或错误信息而提出的。约束是数据库中重要的对象之一，使用约束的目的就是保证数据完整性。

18.2.1 数据库完整性的重要性

数据库中的数据庞杂复杂，各表之间的关系也可能存在着这样那样的关系。无论是通过人工录入还是自动生成都不可避免地破坏数据完整性。例如，可能出现无客户信息的订单，也可能出现订单数量不符合实际情况。数据库作为基础性数据，对于这样的错误是不能容忍的，而依靠人工维护数据完整性则是不现实的。因此，约束成为保证数据完整性的重要手段。

18.2.2 保持数据库完整性的重要方面

数据完整性分为四类：实体完整性 (Entity Integrity)、域完整性 (Domain Integrity)、参照完整性 (Referential Integrity)、用户定义的完整性 (User-defined Integrity)。

1. 实体完整性

实体完整性指表中行的完整性。要求表中的所有行都有唯一的标识符，存储标识符的列称为主键列。

2. 参照完整性

参照完整性用于表述表之间的关系。这种关系一般表现为主从关系。附表中的数据依附于主表数据的存在而存在。例如，对于客户表与订单表，客户表为主表，而订单表为附表。订单表中不可能存在某张订单，该订单的客户不在客户表中。如果要删除某个客户，那么该客户的订单也应该随之删除。

3. 域完整性

域完整性是对数据表中列属性的约束，如数据类型、格式、值域范围、是否允许空值等约束。域完整性也与实际的逻辑相联系。例如，针对订单的到货日期，对应列值不能为空。

4. 用户自定义完整性

前三种的约束都是由数据库自动完整的，而用户定义完整性则是由用户自定义的约束。这极大增强了数据库的约束的灵活性。例如，针对订单状态只能存在“有效”和“无效”两种。

18.2.3 利用约束保持数据完整性

针对数据完整性的 4 个方面，可以分别利用相应的约束来实现。



1. 保证实体完整性

保证实体完整性一般使用主键实现。一旦创建主键，主键将能够唯一标识一条记录。需要注意的是，主键值不能为空，因为利用 `select * from table_name where primary_key = NULL` 是无法获得任何记录的。另外，主键可以是多个列的组合，只要这些列的组合值唯一，同样可以保证实体完整性。

关于主键的详细信息，可以参考 6.1 节的内容。

2. 保证参照完整性

参照完整性一般通过外键约束来实现。外键约束定位于表之间的关系，而且强制实现由子表向主表的对应。这在很大程度上解放了开发者通过复杂的逻辑来保证参照完整性。

关于外键的详细信息，可以参考 6.2 节的内容。

3. 保证域完整性

这里的域实际指的是数据表的列。最常见的域约束为非空约束和默认值约束。

非空约束是指，针对表中某列数据不能为 NULL。值得注意的是，在 Oracle 9i 之后，NULL 与空字符串都被当做 NULL 进行处理，因此，将不能使用空字符串来填充非空列。非空约束是非常有意义的，例如，对于货运订单，需要限制其到货日期不能为空。即可针对该列使用非空约束。对于某列使用非空约束，可以在列声明时，添加 `not null` 选项。

【范例 18-1】 演示如何利用 `not null` 指定非空约束。

```
create table purchase_order (purchase_order_id integer primary key, goods_name
varchar2(20), status varchar2(3), receive_date date not null)
```

【代码说明】 `receive_date date not null`，用于限制列 `receive_date` 的值不能为空。

默认值约束则指定某列在插入新数据、又没有指定实际值时，所使用的值。默认值约束弥补了非空约束的呆板性。关于默认值约束，可以参考 6.5 节的内容。

4. 保证用户自定义完整性

用户自定义完整性一般通过检查约束来实现。检查约束可以灵活定制检查条件，这正符合了用户自定义完整性的要求。例如，为了限制订单状态仅能为有效或无效，可以利用检查约束。

【范例 18-2】 演示如何利用检查约束指定自定义条件。

```
create table purchase_order (purchase_order_id integer primary key, goods_name
varchar2(20), status varchar2(3) check(status in ('ACT', 'CXL')), receive_date date not
null)
```

【代码说明】 `check(status in ('ACT', 'CXL'))` 中的 `check` 通过判断其参数 `status in ('ACT', 'CXL')` 的真假来决定新插入或更新的数据是否能够通过校验。

在新建表 `purchase_order` 中，`status` 列仅允许输入“ACT”或者“CXL”。



18.3 本章实例

一旦在数据表的某个或某些列上建立了索引，那么查询时也不一定能够使用。本节通过几个实例查看索引使用时的误区。



18.3.1 使用比较运算符不当

Oracle 是通过比较索引值来实现搜索的，但是，如果使用了不当的比较运算符，有可能仍会使用全表扫描。

【范例 18-3】演示使用“不等于”运算引起 Oracle 忽略索引。

在表 test_objects 中定义了列 object_name 作为索引，但使用如下 SQL 语句，将不会使用索引。

```
select * from test_objects where object_name <> 'PEOPLE'
```

在 PL/SQL Developer 中查看执行计划，如图 18-10 所示。

Description	Object owner	Object name	Cost	Cardinality	Bytes
SELECT STATEMENT, GOAL = ALL_ROWS			147	47734	4439262
TABLE ACCESS FULL	SYSTEM	TEST_OBJECTS	147	47734	4439262

图 18-10 使用<>比较，引起索引被忽略

18.3.2 函数的使用

对于索引列，一旦使用了函数，那么将在查询时无法使用索引。

【范例 18-4】演示在索引列上使用函数，将导致忽略索引。

```
select * from test_objects where instr(object_name, '_') = 0
```

【代码说明】instr(object_name, '_')=0 指定了过滤条件——列 object_name 中不含字符“_”。此时，Oracle 将忽略索引，如图 18-11 所示。

Description	Object owner	Object name	Cost	Cardinality	Bytes
SELECT STATEMENT, GOAL = ALL_ROWS			147	477	44361
TABLE ACCESS FULL	SYSTEM	TEST_OBJECTS	147	477	44361

图 18-11 对索引列使用函数，将导致索引被忽略

18.3.3 联合索引

索引不仅可以建立在单列之上，而且可以建立在多列之上。建立在多列之上的索引被称为联合索引。使用联合索引列中的部分列作为查询条件，将无法使用到联合索引所带来的性能提升。

【范例 18-5】演示联合索引被忽略的情况。

(1) 在表 test_objects 的 object_name 和 owner 列上建立联合索引。

```
create index idx_owner_type on test_objects(owner, object_type);
```

(2) 尝试使用 owner 作为查询条件：

```
select * from test_objects where owner = 'TIGER'
```

此时，在 PL/SQL Developer 中查看执行计划，如图 18-12 所示。



```
select * from test_objects where owner = 'TIGER'
```

Description	Object owner	Object name	Cost	Cardinality
SELECT STATEMENT, GOAL=ALL_ROWS			146	2808
TABLE ACCESS FULL	SYSTEM	TEST_OBJECTS	146	2808

Return all rows from a table

图 18-12 在 PL/SQL Developer 中查看任务计划

(3) 只有同时使用列 `owner` 和列 `object_type` 作为查询条件, 才能发挥联合索引 `idx_owner_type` 的作用。

```
select * from test_objects where owner = 'TIGER' and object_type='TABLE'
```

此时, 在 PL/SQL Developer 中查看执行计划, 如图 18-13 所示。

```
select * from test_objects where owner = 'TIGER' and object_type='TABLE'
```

Description	Object owner	Object name	Cost	Cardinality	Bytes
SELECT STATEMENT, GOAL=ALL_ROWS			6	104	9672
TABLE ACCESS BY INDEX ROWID	SYSTEM	TEST_OBJECTS	6	104	9672
INDEX RANGE SCAN	SYSTEM	IDX_OWNER_TYPE	1	104	

Select a range of values from an index in ascending order

图 18-13 在 PL/SQL Developer 中查看任务计划

总结所有使用索引失败的例子可知, 有的索引列的确出现在了查询条件中, 但也并不一定能够使用到索引所带来的好处。Oracle 会自行分析所给出的查询条件, 并根据查询条件判断是否可以使用索引。



18.4 本章小结

本章简要讲解了数据库性能与数据完整性两个方面的内容。对于数据库性能来说, 建立索引是最常用的方法。但是使用索引的方便性和对于性能的有效提高, 往往使索引的使用走入误区, 本章列举了各种 DML 操作语句对索引的影响来提醒读者——避免索引的滥用。数据完整性是数据库的重要研究方向, 本章列举了数据完整性的主要方面及实现途径。关于如何利用索引来保证数据完整性, 读者可以参考相关章节的具体内容。



18.5 习题

1. 请简述 Oracle 索引的使用场景。
2. 请简述 Oracle 数据完整性的主要方面。
3. 在列 `student_name` 上为表 `students` 创建索引。
4. 在列 `student_name` 和列 `student_age` 上为表 `students` 创建联合索引。



第 19 章 数据一致性与事务管理

数据库中，除了数据的完整性之外，数据的一致性同样是一个重要的话题。而事务是保证数据一致性的重要手段。本章主要讲解数据的一致性及事务的管理。本章的主要内容如下。

- 数据一致性简介；
- 事务简介；
- Oracle 中的事务处理；
- 事务处理的原则。

通过本章的学习，读者可以了解 Oracle 中的数据一致性，并对事务处理原则及在 Oracle 中的具体体现有清晰的认识。



19.1 什么是数据一致性和事务

数据一致性是指数据库的数据在每一时刻都是稳定且可靠的状态。而事务是保证数据一致性的主要手段。

19.1.1 数据一致性

对于一个数据库来说，其中的数据可能每时每刻都在发生着变化，而在数据变化的同时，也无时无刻不伴随着数据的读取。这就对数据库的状态产生了比较高的要求——数据库的每次改变都是可被接受的，而每次读取的数据也都是正常的。这就是数据一致性的体现。

例如，在某个用户的银行账户中，现有存款 100 元。此时，如果一个用户读取其中的数据为 100 元，那么，此时的 100 元是可以接受的数据。但在同一时刻，另一用户针对同一账户进行了以下操作，向其中存入了 100 元，但是还没有提交动作，那么，此时第一个用户有可能会读到 200 元的余额。事实上，由于某些原因，第二个用户的提交没有成功，那么第一个用户所读取的 200 元，并非数据库的真实和可靠的状态。这就造成了读不一致性。

另外一种情形，如果第一个用户读取了 200 元的同时，取出了 100 元，那么计算获得余额为 100 元。而第二个用户的存款动作失败，最后，第一个用户的余额重新覆盖数据库中的数据，那么就会将 100 再次写回数据库，这造成了取出 100 元，最后余额还为 100 元的状态，此时的数据库也不符合一致性的状态。

19.1.2 事务

在数据库中，提出了事务的概念来保证数据库中数据的一致性。事务往往包括一个或多个处理步骤。例如，在超市购物包括，选购商品、放入购物车、付款、个人现金或银行账户余额减少、超市账户余额增加、商品库中商品信息更新等步骤。这些步骤组成了一个事务，当其中任何步骤出现异常，并且不能正常进行下去，都会影响其他所有步骤。例如，付款阶段不能正常进行，那么将不会对个人的现金或银行账户造成影响，超市的账户也不可能进行余额的增加，商品库中也不能更新该商品的信息。



同样的，如果个人现金或账户余额不足，那么也无法完成整个交易。在此之前所做的所有动作：选购商品、放入购物车等都将无效，商品应当被重新放回货架。



19.2 Oracle 中的事务处理

Oracle 中的事务应当使用关键字 `transaction`。一个事务的生命周期应当包括：事务开始、事务执行和事务结束。需要注意的是，Oracle 中并不能显式开始一个事务，也不存在这样的语句。事务的开始总是隐式进行的，而事务的结束则可以利用 `commit` 或者 `rollback` 命令进行终止。Oracle 中控制事务的常用命令包括：

- `Commit;`
- `roll back;`
- `savepoint;`
- `roll back to <savepoint>;`
- `set transaction;`
- `set constraint.`

本节将通过实例来详细讲述这些命令的用法。

19.2.1 commit 命令

`commit` 命令用于提交事务，并将事务中对数据库的修改进行持久化，即将数据库修改为另外一种状态，而这种状态是可接受的、可靠的状态。范例 19-1 所示的 SQL 语句是来自存储过程的片段。

【范例 19-1】 演示 `commit` 命令的使用。

```
while i<1000 loop
  update people set salary = salary + 10*i where id = i;
  i := i+1;
end loop;
commit;
```

【代码说明】 该片段中，依次更新 `id` 列与变量 `i` 相同的工资列，最后进行了 `commit` 操作。

对于开发者来说，最安全的方式是显式进行数据的提交或者回滚，以结束事务。但很多时候，许多开发者并未注意该问题，而是依靠开发工具来进行提交或回滚。例如，PL/SQL Developer 提供了 `Commit` 和 `Roll Back` 按钮，如图 19-1 所示。



图 19-1 PL/SQL Developer 提供的 `Commit` 和 `roll Back` 按钮

此时需要注意的是，如果用户未提交对数据库的修改，而关闭了回话，或者数据库连接在提交之前断开，那么针对该数据库的所有操作都将执行回滚操作。

另外，需要明确的概念是，在提交之前，数据库已经进行了实际更新，不过，并未得到数据库认可，因此提交动作只是一个获得认可的过程，其花费的数据库资源非常少。而且，提交一条数据与提交 1000 条数据所花费的数据库资源是相同的。因此，当实现大数据量的数据修改或者插入操作时，应当采取最后一次性提交的策略，如范例 19-1 所示的提交方式。如果使用多次提交，将会给数据库带来相对较大的负担，如范例 19-2 所示的 SQL 代码。

**【范例 19-2】** 演示多次提交。

```
while i<1000 loop
  update people set salary = salary + 10*i where id = i;
  i := i+1;
  commit;
end loop;
```

【代码说明】 在该代码片段中，每次更新数据，都有一次提交动作，将造成数据库资源的浪费。

19.2.2 roll back 命令

roll back 命令用于回滚用户操作。在某些时机，例如程序代码段中出现异常或错误，或者用户直接发出撤销命令，需要回滚操作。回滚操作将终止事务处理，并撤销用户在当前事务中进行的更改。范例 19-3 演示了如何在异常发生时回滚所有操作。

【范例 19-3】 捕获异常，回滚事务。

```
begin
  while i<1000 loop
    update people set salary = salary + 10*i where id = i;
    i := i+1;
  end loop;
exception
  roll back;
end;
```

【代码说明】 该代码片段中，依次更新 id 值与变量 i 相同的工资列。但是，有可能会因为列约束等原因造成更新失败，从而抛出异常，因此，利用 exception 捕获异常，并回滚所有操作。

回滚操作，首先要读取回滚段信息，并利用这些信息将数据库中已发生的修改重新恢复。例如，对于使用了 update 操作的列，则需要将其恢复到原值，而使用了 delete 操作的行，则需要再次执行插入操作。因此，回滚操作所需要的时间和花费的资源，依赖于在事务中所执行的数据库更改，并与之成正比。

通常情况下，回滚操作是非常耗费时间和资源的，因此，回滚往往被用于处理异常，而不用做终端用户的可操作选项。一旦终端用户经常性使用回滚操作，那么将给数据库带来非常大的负担。

应当保证终端用户在提交事务之前进行确认，来代替允许用户执行回滚，从而实现提交与回滚操作的平衡。

19.2.3 savepoint 和 roll back to savepoint 命令

通过对 roll back 的描述可知，roll back 命令走向了一个极端——回滚整个事务的所有操作。而有时，大部分的工作都是必须执行的，仅有少量的工作有可能出现异常并需要回滚。那么，除了利用多个事务进行处理之外，还可以使用 save point 在事务中建立标记点，并允许用户只回滚标记点之后的动作。回滚标记点之后的操作，需要使用 roll back to savepoint 命令。

【范例 19-4】 演示如何使用 savepoint 和 roll back to savepoint 命令。

```
update people set id = id -1;

begin
  savepoint insert_people;
```



```
update people set staus = 'CXL';
insert into people (id, name, status)
values (2, 'allen', 'ACT');
exception
when others then
rollback to insert_people;
end;
```

【代码说明】`update people set id = id -1` 用于更新表 `people` 中的数据；`savepoint insert_people` 用于建立标记点，标记点的名称为 `insert_people`；`exception` 用于处理异常，当 `begin-end` 块之内的语句出现异常时，将数据库状态回滚到 `insert_people` 标记点之前。

为了验证该实例，首先查询表 `people` 的数据状态：

```
SQL> select * from people;
```

ID	NAME	STATUS
2	James	ACT
3	John	ACT
4	Robert	ACT
5	Michael	ACT
6	William	ACT
7	David	ACT
8	Richard	ACT
9	Charles	ACT
10	Joseph	ACT
11	Thomas	ACT

10 rows selected

然后在 PL/SQL Developer 中执行范例 19-4 所示的代码。由于主键的作用，语句 `insert into people (id, name, status) values (2, 'allen', 'ACT')` 一定会抛出异常，并被 Oracle 捕获，进而执行 `rollback to insert_people`。此时在同一个会话中，查询表 `people` 中的数据。

```
SQL> select * from people;
```

ID	NAME	STATUS
1	James	ACT
2	John	ACT
3	Robert	ACT
4	Michael	ACT
5	William	ACT
6	David	ACT
7	Richard	ACT
8	Charles	ACT
9	Joseph	ACT
10	Thomas	ACT

10 rows selected

可见，语句 `update people set id = id -1` 已经成功执行，而 `update people set staus = 'CXL'` 则被回滚。

这里需要注意的是，回滚操作虽然回滚了数据库状态，但是并不会改变程序的运行轨迹，也就是说，程序继续执行 `rollback to insert_people` 的下一条语句。这一点区别于其他编程语言中的 `go to` 语句（虽然 `goto` 语句饱受诟病，但是其思想仍然在影响着开发者）。

从 PL/SQL Developer 的状态可知，此时事务并未结束，因为没有最后的 `commit` 或者 `rollback` 来结束事务，如图 19-2 所示。



图 19-2 roll back to savepoint 命令不会结束事务

所以完整的语句块，应该在末尾添加提交命令，如范例 19-5 所示。

【范例 19-5】 演示在 rollback to savepoint 之后使用 commit 结束事务。

```
update people set id = id -1;

begin
  savepoint insert_people;
  update people set staus = 'CXL';
  insert into people (id, name, status)
  values (2, 'allen', 'ACT');
exception
  when others then
    rollback to insert_people;
end;
commit;
```

【代码说明】 commit 命令用于提交本事务的所有更改，并结束事务。

19.2.4 事务的属性和隔离级别

事务本身存在着一些属性，这可以保证事务以某种特定的规则运行。所谓隔离，是指将事务所能看到的数据库状态与其他事务分隔开来。这具有很强的现实意义。因为数据库总是以动态改变的形式存在的，而事务往往处理时，要求有比较稳定的状态。以下 4 种事务的属性和隔离级别值得关注：

- read only 属性；
- read write 属性；
- serializable 隔离级别；
- read committed 隔离级别。

1. read only 属性

一旦将某个事务设置为 read only，将无法在事务内进行修改数据库的操作。

【范例 19-6】 演示 read only 的事务无法进行数据库修改。

```
set transaction read only;
insert into people(id, name, status)
values (13, 'youyou', 'ACT');
commit;
```

【代码说明】 set transaction read only 用于设置当前事务为只读事务；而 insert 语句则用于向表 people 中插入新的数据。

在 PL/SQL Developer 中执行该代码，Oracle 将抛出错误提示，如图 19-3 所示。

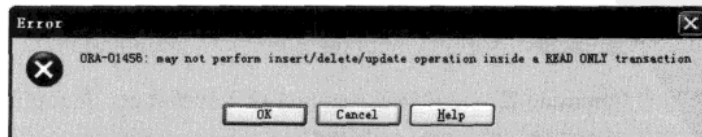


图 19-3 不能在 read only 的事务中进行数据库修改



同样，不能够首先执行插入语句，然后才将事务设置为只读。这是因为，事务属性的设置语句，只能出现在事务的开头。

【范例 19-7】演示事务属性的设置只能出现在事务的第一条语句。

```
insert into people(id, name, status)
values (13, 'youyou', 'ACT');
set transaction read only;
commit;
```

【代码说明】该代码段中，首先向表 `people` 中插入新的数据，然后才设置当前事务的属性。在 PL/SQL Developer 中执行该代码，Oracle 将抛出错误提示，如图 19-4 所示。

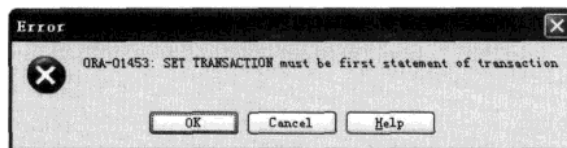


图 19-4 事务属性设置语句，必须处于事务的开头

一旦将事务设置为只读，那么在事务开始执行时，数据库状态将被冻结。因为事务本身不可能修改数据库，而其他事务对数据库的修改，对于当前事务来说，都是透明的。也就是说，在该事务的执行过程中，数据库的状态与事务开始时的状态一致。

【范例 19-8】演示使用只读事务冻结数据库状态。

(1) 利用 PL/SQL Developer 新建一个 Command 窗口，并在其中键入如下命令：

```
set transaction read only;
select * from people;
```

执行该代码段，其查询结果如下：

```
SQL> set transaction read only;
```

```
Transaction set
```

```
SQL> select * from people;
```

ID	NAME	STATUS
1	James	ACT
2	John	ACT
3	Robert	ACT
4	Michael	ACT
5	William	ACT
6	David	ACT
7	Richard	ACT
8	Charles	ACT
9	Joseph	ACT
10	Thomas	ACT

```
10 rows selected
```

【代码说明】在该 Command 窗口中，首先将事务设置为只读，然后查询此时表 `people` 中的数据。

(2) 新建第二个 Command 窗口，并向表 `people` 中插入新的数据，然后提交。

```
SQL> insert into people (id, name, status)
2 values (11, 'yoyo', 'ACT');
```

```
1 row inserted
```

```
SQL> commit;
```

```
Commit complete
```

(3) 新建第三个 Command 窗口，并在其中查询表 `people` 中的数据：

```
SQL> select * from people;
```

ID	NAME	STATUS
1	James	ACT
2	John	ACT
3	Robert	ACT
4	Michael	ACT
5	William	ACT
6	David	ACT
7	Richard	ACT
8	Charles	ACT
9	Joseph	ACT
10	Thomas	ACT
11	yoyo	ACT

```
11 rows selected
```

可见，此时数据表 `people` 中已经成功插入了新的数据，并被新建事务所识别。

(4) 在第一个事务中，由于既未使用 `commit` 命令提交事务，也未使用 `roll back` 命令回滚事务，因此该事务仍未结束。在第一个事务窗口中查询此时表 `people` 中的数据：

```
SQL> select * from people order by id;
```

ID	NAME	STATUS
1	James	ACT
2	John	ACT
3	Robert	ACT
4	Michael	ACT
5	William	ACT
6	David	ACT
7	Richard	ACT
8	Charles	ACT
9	Joseph	ACT
10	Thomas	ACT

```
10 rows selected
```

此时的事务中，所能识别的数据表 `people` 仍然停留在事务开始时的状态。事实上，不止表 `people`，包括整个数据库，当前事务所能识别的状态都是事务开始时的状态。

只读事务的这种特性，可以提供一种稳定的状态，从而处理大量数据查询工作。例如，在生成复杂报表时，需要查询大量数据，而这些数据又是频繁变更的，在处理报表的过程中，可以利用只读事务来提供稳定的环境，以使生成的报表有意义。

2. read write 属性

`read write` 属性可以将事务设置为可读、可写状态。这实际是事务的默认状态，因此，该属性的显式设定，并没有太大的现实意义。

需要注意的是，一旦使用 `set transaction read write` 命令，那么该命令之前，不能出现 `set transaction` 命令之外的其他命令。



3. serializable 隔离级别

serializable 隔离级别是指串行化事务。串行化事务可以实现与只读事务实现相同的功能——隔离其他事务对数据库状态的影响。但是串行化事务允许在其中执行任何 DML 操作，包括删除、修改、插入数据等。

在默认情况下，一个事务可以识别其他事务针对数据库的修改，而这种修改仅限于已经提交到数据库的修改。使用了串行化事务，那么，其他事务已经提交的修改也将被隔离。这里所说的隔离，实际是对于查询操作来说，也就是串行化事务的处理过程中，无法查看到其他事务的修改。

【范例 19-9】演示串行化事务的使用。

(1) 首先在 PL/SQL Developer 中新建 Command 窗口，并设置当前事务为串行化事务。

```
SQL> set transaction isolation level serializable;
```

```
Transaction set
```

set transaction isolation level serializable 用于将当前事务的隔离级别设置为串行化。此时，查询表 people 中的数据。

```
SQL> select * from people order by id;
```

ID	NAME	STATUS
1	James	ACT
2	John	ACT
3	Robert	ACT
4	Michael	ACT
5	William	ACT
6	David	ACT
7	Richard	ACT
8	Charles	ACT
9	Joseph	ACT
10	Thomas	ACT
11	yoyo	ACT

```
11 rows selected
```

(2) 新建一个 Command 窗口，并在其中执行插入语句，向表 people 中插入新的数据，最后执行提交动作。

```
SQL> insert into people (id, name, status)
2 values (12, 'raid', 'ACT');
```

```
1 row inserted
```

```
SQL> commit;
```

```
Commit complete
```

(3) 此时，在第一个 Command 窗口中查询表 people 中的数据。

```
SQL> select * from people order by id;
```

ID	NAME	STATUS
1	James	ACT
2	John	ACT
3	Robert	ACT
4	Michael	ACT
5	William	ACT



6	David	ACT
7	Richard	ACT
8	Charles	ACT
9	Joseph	ACT
10	Thomas	ACT
11	yoyo	ACT

11 rows selected

可以看出, 由于该事务为一个串行事务, 因此, 不能识别其他事务对数据库的修改, 即使这些修改已经进行了提交。

(4) 虽然串行事务不能识别其他事务的修改, 但并不意味着可以在该事务中无视修改的存在。例如, 尝试向表 `people` 中插入数据, 并故意违反主键约束。

```
SQL> insert into people (id, name, status)
2 values (12, 'rain', 'ACT');
```

```
insert into people (id, name, status)
values (12, 'rain', 'ACT')
```

ORA-00001: unique constraint (SYSTEM.SYS_C005393) violated

尝试向表 `people` 中插入 `id` 为 12 的新记录, 但是, 由于违反了主键的唯一性约束, 将导致插入失败。

4. read committed 隔离级别

`read committed` 隔离级别是事务的默认隔离级别, 即当只能读取其他事务已经提交的修改。对于尚未提交的修改, 只有实现该修改的事务本身可以进行读取。



19.3 事务处理原则

事务处理的原则可以概括为 ACID。ACID 是以下单词的首字母。

- 原子性 (Atomicity): 整个事务不可分割, 要么全部执行, 要么都不执行。
- 一致性 (Consistency): 事务一旦提交生效, 会将数据库从一种状态转变为另一种状态。而且转变之后的状态是合理和可接受的。
- 隔离性 (Isolation): 在事务处理的过程中, 事务处理的效果对于其他事务, 是完全透明的。只有提交之后, 其效果才能为其他事务所识别。即前面提到的 `read committed` 的隔离级别。
- 持久性 (Durability): 一旦提交了事务处理, 那么事务处理的效果将永久生效。

以上 4 点原则是数据库事务的基本属性。本节对其进行一一阐述。

19.3.1 原子性

Oracle 中的事务处理遵循原子性。无论事务中包含多少语句, 要么提交所有工作, 要么废除所有工作。需要注意的是, 利用 `save point` 和 `roll back savepoint` 回滚部分操作并非特例, 这是因为利用 `roll back savepoint` 也是用户指定的动作。可以看做事务处理中的普通命令。

【范例 19-10】 演示事务原子性。

(1) 创建用于测试的数据表 T:

```
SQL> create table t(data varchar2(2));
```

Table created



(2) 向该表中插入两条数据:

```
SQL> insert into t values('a');
```

```
1 row inserted
```

```
SQL> insert into t values('b');
```

```
1 row inserted
```

(3) 在插入数据之后, 回滚事务:

```
SQL> rollback;
```

```
Rollback complete
```

(4) 此时查询表 T 中的数据:

```
SQL> select * from t;
```

```
DATA
```

```
----
```

通过查询结果可知, 两条插入语句都没有成功执行, 这表明了回滚操作包括了两条插入语句。

(5) 再次向表中插入数据, 并提交:

```
SQL> insert into t values('a');
```

```
1 row inserted
```

```
SQL> insert into t values('b');
```

```
1 row inserted
```

```
SQL> commit;
```

```
Commit complete
```

(6) 事务提交之后, 再次查看表中数据:

```
SQL> select * from t;
```

```
DATA
```

```
----
```

```
a
```

```
b
```

分析查询结果可知, 此时的数据已经全部提交。

19.3.2 一致性

Oracle 中的事务必须保证数据的一致性。数据一致性的一个典型实例为外键约束。例如, 在外键约束时, 首先更新父表中的主键, 如果在子表中有数据与该父表记录相关联, 那么父表的更新将被终止。同样地, 如果试图更新子表的外键列, 也会造成更新失败。因为这两个动作都破坏了数据的一致性。无论哪种操作, 如果成功执行, 都会使数据库中的数据处于逻辑上的不可接受状态。此时的数据一致性的保持出现在语句级, 即每条 DML 语句都会进行校验。

而事务可以包含多条语句, 并可以在事务开始执行时将约束设置为延迟校验。以外键约束为例, 可以成功避开无法更新父表主键和子表外键的尴尬。但是, 提交时, 延迟校验将生效, 以保持数据一致性。

**【范例 19-11】** 演示事务的一致性。

(1) 创建测试表:

```
SQL> create table vendor(vendor_id integer primary key, vendor_name varchar2(20));
```

Table created

```
SQL> create table po(po_id integer, vendor_id integer, constraint fk_vendor_po  
foreign key(vendor_id) references vendor deferrable);
```

Table created

(2) 向表中插入测试数据:

```
SQL> insert into vendor values(1, 'HK');
```

1 row inserted

```
SQL> insert into po values(1, 1);
```

1 row inserted

(3) 尝试更新表 vendor 中的主键:

```
update vendor set vendor_id = 2
```

ORA-02292: integrity constraint (SYSTEM.FK_VENDOR_PO) violated - child record found

此时, 由于外键约束的作用, 更新父表中的主键列, 将会抛出错误提示。当然, 如果更新子表中的外键列, Oracle 照样会抛出错误提示。

```
update po set vendor_id = 2
```

ORA-02291: integrity constraint (SYSTEM.FK_VENDOR_PO) violated - parent key not found

(4) 将外键约束设置为延迟校验, 然后分别修改父表与子表中的数据, 最后提交修改。

```
SQL> set constraints fk_vendor_po deferred;
```

Constraints set

```
SQL> update vendor set vendor_id = 2;
```

1 row updated

```
SQL> update po set vendor_id = 2;
```

1 row updated

```
SQL> commit;
```

Commit complete

【代码说明】set constraints fk_vendor_po deferred 用于将外键 fk_vendor_po 设置为延迟校验。

(5) 此时, 查询表中数据:

```
SQL> select * from vendor;
```

VENDOR_ID	VENDOR_NAME
2	HK

```
SQL> select * from po;
```



PO_ID	VENDOR_ID
1	2

分析查询结果可知，通过延迟校验规则，可以分别修改父表和子表中的主键和外键列。

(6) 另外，当事务提交时，校验规则仍然起作用，以保证数据的一致性。例如，以下 DML 语句将无法成功执行。

```
SQL> set constraints fk_vendor_po deferred;
```

```
Constraints set
```

```
SQL> update vendor set vendor_id = 3;
```

```
1 row updated
```

```
SQL> update po set vendor_id = 4;
```

```
1 row updated
```

在以上代码中，首先将约束设置为延迟校验，然后将父表 vendor 中的主键 vendor_id 修改为 3，接着将子表 po 中的外键 vendor_id 修改为 4。这两条 DML 语句均可成功执行，但是此时，该事务并未结束，因为并没有出现 commit 或者 roll back。尝试提交修改：

```
SQL> commit;
```

```
commit
```

```
ORA-02091: transaction rolled back
```

```
ORA-02291: integrity constraint (SYSTEM.FK_VENDOR_PO) violated - parent key not found
```

从错误提示可知，当提交时，由于不能保证数据的一致性，事务被回滚。这正是事务一致性的体现。

19.3.3 隔离性

事务的隔离性是指，在事务的处理过程中，针对各事务之间的相互影响，所提出的对应策略。因为各事务所执行的动作不同，因此，如果各事务之间相互影响，将使得最终的数据库处于不可预知的状态。SQL92 归纳出了三种读取方式来概括可能出现的情况。

1. 脏读取 (Dirty Read)

脏读取意味着，可以读取来自外界其他动作对数据库的修改，而这种修改尚未提交，未提交的数据有可能回滚。也就是说，读取的数据并非真正有效的数据，这将直接破坏数据一致性。

2. 不可重读

不可重读意味着，如果用户在某一时刻读取了一条记录，那么，在下一时刻再次读取时，该记录已发生改变。其状态类似于“人不能两次踏入同一条河流”。不可重读并非不可接受的，相反，不可重读是一种正常的数据库状态。这与脏读取是有本质区别的。

3. 影像读取

影像读取意味着，如果用户在某一时刻执行了一个查询，在下一时刻再次执行相同查询时，可能会有新的数据加入。但是，已经读取的数据是不会改变的，只是查询所获得结果集更大而已。

ANSI (美国国家标准委员会) 在 SQL 标准中包含了对隔离性的定义。而 SQL 92 标准定

义了 4 种层次的事务处理的隔离性。这 4 种隔离层次是针对三种读取方式是否允许而设定的。它们分别是：未提交读取（Read Uncommitted）、已提交读取（Read Committed）、可重复读取（Repeatable Read）和串行读取（Serializable）。表 19-1 概括了 4 种隔离层次与 3 种读取方式的关系。

表 19-1 隔离层次与读取方式关系表

隔离层次	脏读取	不可重读	影像读取
非提交读取（Read Uncommitted）	允许	允许	允许
已提交读取（Read Committed）	禁止	允许	允许
可重复读取（Repeatable Read）	禁止	禁止	允许
串行读取（Serializable）	禁止	禁止	禁止

三种读取方式实际是针对能否读取数据库中的变化而设定的。从表中可以看出，随着隔离层次的提高，读取方式的范围越来越小。这意味着隔离层次越高，对于数据库变化的反映越迟钝。而串行读取则不允许数据库中的任何变化反映到当前会话中。

Oracle 中的事务使用了两种 SQL 标准的隔离层次——已提交读取和串行读取。已提交读取是 Oracle 事务的默认隔离层次，即 19.2.4 节提到的 read committed 级别。串行读取则是 19.2.3 节提到的 serializable 级别。

19.3.4 持久性

持久性是指，当事务一旦提交，其改变将会生效，并将信息存储在磁盘上。当系统再次重新启动或者故障时，这些信息不会丢失。

对于提交动作来说，用户见到提交成功提示时，Oracle 并未完成持久化工作。但是，Oracle 数据库利用 redo 日志文件来保证系统的持久性。redo 文件在事务提交的前一刻生成，其中记录了一旦提交时，系统崩溃，那么该如何将事务的工作再次执行，以保证事务真正执行完毕，并持久化到数据库。



19.4 本章实例

本节将通过一个实例查看事务嵌套的处理原则。事务嵌套是指，在事务的处理过程中调用了其他子程序，子程序所完成的功能也是一个事务。

【范例 19-12】 本范例用于演示事务嵌套的处理原则。

(1) 创建测试表 records:

```
create table records(record varchar2(20));
```

【代码说明】 表 records 仅有一列 record，列的数据类型为 varchar2 (20)。

(2) 创建存储过程 local:

```
create or replace procedure local is
counter number := -1;
begin
select count(*) into counter from records;
dbms_output.put_line('子程序, 当前记录数为: ' || counter);

insert into records values ('该记录来自 local');
commit;
end;
```



【代码说明】该存储过程的作用为：首先计算表 `records` 中的记录数，并将当前记录数打印出来；接着向表 `records` 中插入新的数据。

(3) 在 PL/SQL 的【Command Window】中键入如下代码：

```
declare
    counter number := -1;
begin
    delete from records ;
    commit;
    insert into records values ('该记录来自 Global');
    local;
    select count(*) into counter from records;
    dbms_output.put_line('主程序, 当前记录数为: ' || counter);
    rollback;

    local;
    insert into records values ('该记录来自 Gobal');
    commit;

    local;
    select count(*) into counter from records;
    dbms_output.put_line('主程序, 当前记录数为: ' || counter);
end;
```

【代码说明】`counter number := -1` 定义了名为 `counter` 的变量；`insert into records values ('该记录来自 Global')` 用于向表 `records` 中插入记录；`local` 则用于调用存储过程 `local`；`select count(*) into counter from records` 将表 `records` 中的记录数存储到变量 `counter` 中；`dbms_output.put_line('主程序, 当前记录数为: ' || counter)` 则打印表 `records` 中的记录数。

其执行结果如下：

SQL> /

```
子程序, 当前记录数为: 1
主程序, 当前记录数为: 2
子程序, 当前记录数为: 2
子程序, 当前记录数为: 4
主程序, 当前记录数为: 5
```

PL/SQL procedure successfully completed

在输出结果中，第一行输出是由子程序 `local` 输出的，此时只有主程序向表 `records` 中插入了一条数据。虽然主程序的提交动作尚未执行，但是子程序已经可以看到主程序对 Oracle 数据的修改。这证明子程序和主程序被当做同一个事务进行处理；第二行输出表的记录数为 2，这是因为子程序又向表 `records` 中插入了新的数据；第三行的输出是主程序进行了回滚，并再次调用子程序时输出的，可见，经过了子程序的提交动作，主程序的回滚动作将不再有效。

以此类推，分析整个输出结果可知，在主程序中调用子程序，而子程序中实现了事务控制，那么无论是在主程序还是子程序中提交/回滚，都将结束到目前为止所执行的事务。即主/子程序的事务之间并不存在真正意义上的事务嵌套。`COMMIT` 和 `ROLLBACK` 的位置无论是在主程序，还是在子程序中都会影响到整个当前事务。

当然，可以通过将子程序设置为自治事务，来实际解决这个问题。其使用语法为：

```
pragma autonomous_transaction;
```

存储过程 `local` 的代码改写如下：

```
create or replace procedure local is
```

```
pragma autonomous_transaction;
counter number := -1;
begin
    select count(*) into counter from records;
    dbms_output.put_line('子程序, 当前记录数为: ' || counter);

    insert into records values ('该记录来自 local');
    commit;
end;
```

再次执行主程序, 执行结果如下:

SQL> /

子程序, 当前记录数为: 0
主程序, 当前记录数为: 2
子程序, 当前记录数为: 1
子程序, 当前记录数为: 3
主程序, 当前记录数为: 4

PL/SQL procedure successfully completed

通过执行结果可知, 主程序和子程序的两个事务之间已经实现了隔离, 子程序也实现了事务自治。



19.5 本章小结

本章讲述了 Oracle 数据库事务的基本处理原则。Oracle 中如何提交和回滚事务, 并讲述了 save point 命令的使用。事务处理中, 重点要理解是隔离层次。隔离层次隔离实际上是数据库的状态变化。SQL 标准制定了 4 种隔离层次, Oracle 实现了其中的两种, 并且自定了 read only 的隔离层次。



19.6 习题

1. 什么是事务?
2. 利用 save point 和 roll back to savepoint 实现回滚标记点。
3. 简述 Oracle 中事务的隔离级别。
4. 简述事务的处理原则。



第 20 章 并发控制

并发是指多用户同时访问数据库。并发能力是衡量数据库性能的重要指标之一，数据库允许并发数量越大，标志着该数据库的性能越好。另一方面，并发会给保持数据库一致性带来挑战。Oracle 在并发方面有着卓越的性能，并有着完善的并发控制机制。本章着重讲述 Oracle 的并发控制，主要包括：

- 并发的概念；
- 悲观锁定与乐观锁定；
- 利用锁定进行并发控制。

通过本章的学习，读者可以对 Oracle 数据库中对于并发和死锁有清晰的了解，并掌握常用的并发控制策略。



20.1 并发与锁定

当多用户同时访问和修改数据库时，将产生并发。而并发将带来数据库不一致性的风险。锁定是现有数据库控制并发的常用手段。Oracle 提供了各种不同层次的锁定，例如行级锁定，表级锁定。大多数情况下，Oracle 都会自动执行锁定控制，并不需要用户显式使用锁定。例如，当某个存储过程在执行时，该存储过程总是处于锁定状态，此时，不允许其他用户对其执行重编译。

【范例 20-1】 演示存储过程执行时，自动锁定。

利用 PL/SQL Developer 的浏览页面中找到已存在的存储过程，例如 printStudents。右键单击该存储过程，在弹出的快捷菜单中选择【Test】命令，并使用调试模式来执行该存储过程。在调试时，使用单步调试功能，此时，存储过程将处于执行状态，如图 20-1 所示。

此时打开该存储过程的编辑窗口，并尝试编译该存储过程。由于执行时的锁定，Oracle 将禁止该存储过程的重编译，并抛出错误提示，如图 20-2 所示。

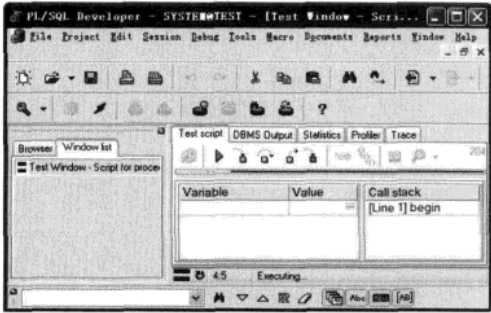


图 20-1 处于执行状态的存储过程

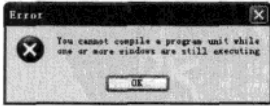


图 20-2 不能编译正在执行中的存储过程



存储过程的锁定不允许其他用户或会话对其进行修改和重编译。在其他用户或会话尝试修改时会给出错误提示，并禁止继续执行。这自然不会出现问题，但并非所有锁定都使用这种机制，因此，往往会造成死锁。

死锁是指用户 1 锁定资源 1，并进行相应的操作，但是并未释放资源；用户 2 锁定了资源 2，并进行了相应的操作，但也未释放该资源。此时，用户 1，尝试锁定资源 2，由于用户 2 已锁定资源 2，用户 1 的操作将被阻塞。用户 1 无法完成自己的工作，只好等待资源 2 的释放。如果用户 2 此时执行完毕，并释放了资源 2，那么用户 1 将有机会锁定资源 2，并继续自己的工作。但是，用户 2 并未执行完毕，而是请求锁定资源 1，此时，用户 1 和用户 2 实际在相互等待。而且，如果没有相关机制进行处理，这种“对峙”的状态将一直持续下去，二者都无法完成自己的工作——即产生了死锁。

Oracle 处理死锁的机制为，为其中某个用户的会话抛出错误，并要求该会话进行提交或者回滚，以释放资源。而另外一个用户的会话将会立即锁定所需资源，并完成自己的工作。这意味着，其中一方的任务以失败告终。

【范例 20-2】 演示 Oracle 中的死锁及处理机制。

(1) 创建表 a 和表 b 用于测试：

```
SQL> create table a (data integer);
```

Table created

```
SQL> create table b (data integer);
```

Table created

(2) 分别向表中插入数据，并进行提交：

```
SQL> insert into a (data) values(5);
```

1 row inserted

```
SQL> insert into b (data) values(8);
```

1 row inserted

```
SQL> commit;
```

Commit complete

(3) 将该会话作为会话 1，并在该会话中更新表 a 中的数据——将 data 列的值修改为 6：

```
SQL> update a set data = 6;
```

1 row updated

(4) 新建【Command Window】，并将其作为会话 2。在会话 2 中更新表 b 的数据——将 data 列的值修改为 9：

```
SQL> update b set data = 9;
```

1 row updated

(5) 接着，在会话 2 中更新表 a 的数据——尝试将 data 列的值修改为 7：

```
SQL> update a set data = 7;
```

此时，该动作将被阻塞，如图 20-3 所示。

(6) 对比会话 1 和会话 2 的状态可知，会话 2 中的动作将被阻塞，其状态总是处于执行状



态：会话 1 的状态仍然处于正常，如图 20-4 所示。

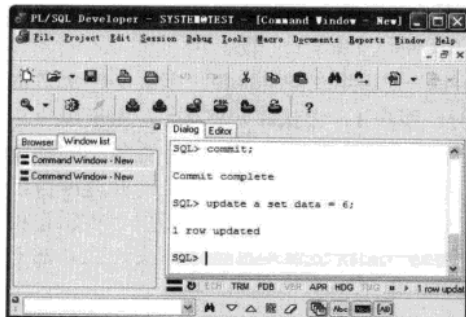


图 20-3 会话 2 中的动作将被阻塞

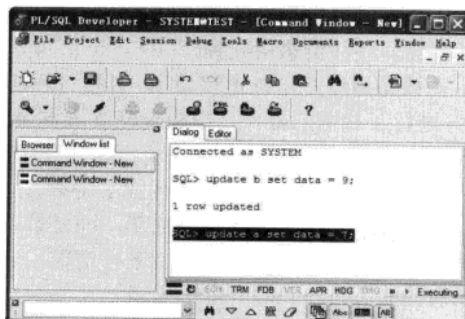


图 20-4 会话 1 处于正常状态

(7) 会话 1 处于正常状态，是因为该会话没有请求已锁定的资源。但是，此时在会话 1 中更新表 b 中的数据，将会引起死锁，并引发会话 1 的阻塞，如图 20-5 所示。

(8) 此时，Oracle 会立即检测到死锁的存在，并作出反应——在会话 2 中抛出错误，要求该会话进行提交或者回滚，如图 20-6 所示。

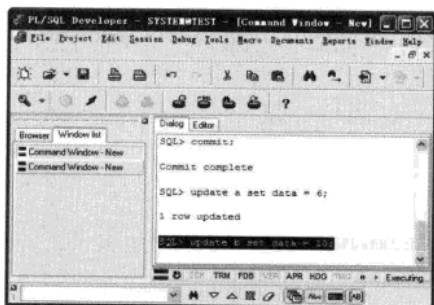


图 20-5 会话 1 的阻塞

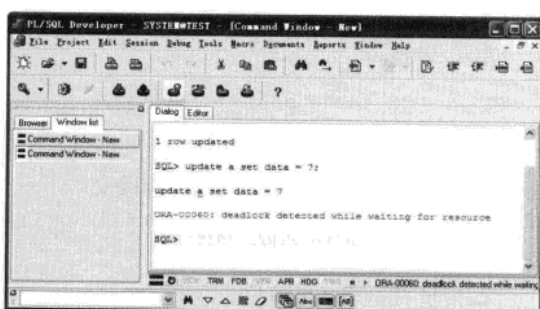


图 20-6 会话 2 抛出错误，并被强制进行提交或者回滚

提交会话 2 的事务：

```
SQL> commit;
```

```
Commit complete
```

(9) 在事务提交之后，查看表 a 和表 b 中的数据：

```
SQL> select * from a;
```

```
DATA
-----
5
```

```
SQL> select * from b;
```

```
DATA
-----
9
```

可见，此时表 b 中的数据已经成功更新。这表示，会话 2 虽然不能获得表 a 的锁定，但是对其已经锁定的表 b 仍然具有控制权。当然，在事务提交之后，会释放对表 b 的锁定。

(10) 此时，会话 1 的状态已经恢复正常，并更新了表 b 中的数据，如图 20-7 所示。

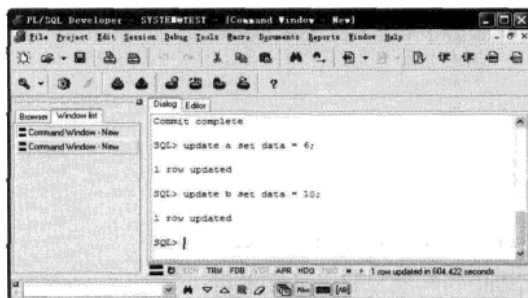


图 20-7 Oracle 处理死锁之后，会话 1 恢复正常

(11) 提交会话 1 中的事务，并查看表 a 与表 b 中的数据。

```
SQL> commit;

Commit complete

SQL> select * from a;

DATA
-----
6

SQL> select * from b;

DATA
-----
10
```

通过本例的分析，可以看出 Oracle 中关于死锁的处理机制——牺牲一个会话来换取解锁和另一个会话的正常执行。



20.2 数据锁定

数据库中的锁定一般分为两种：悲观锁定与乐观锁定。另外，与其他数据库不同的是，Oracle 不提供锁定升级，而是使用了锁定的转换来替代。本节将讲述悲观锁定与乐观锁定的应用，以及 Oracle 中的锁定转换。

20.2.1 悲观锁定

锁定的意义在于存在冲突。当多用户访问和试图修改同一资源时，就会发生冲突。悲观锁定假定冲突在每个用户的每次修改时都存在，因此，在修改之前都首先尝试将资源进行锁定。

Oracle 中，在 select 语句末尾使用 for update 选项指定悲观锁定。

【范例 20-3】 演示悲观锁定的使用。

在表 people 中含有如下数据：

```
SQL> select * from people order by id;
```

ID	NAME	STATUS
1	James	ACT
2	John	ACT
3	Robert	ACT



4	Michael	ACT
5	William	ACT
6	David	ACT
7	Richard	ACT
8	Charles	ACT
9	Joseph	ACT
10	Thomas	ACT
11	yoyo	ACT
12	raid	ACT

12 rows selected

(1) 首先利用 for update 语句锁定 id 为 1 的数据:

```
SQL> select * from people where id = 1 for update;
```

ID	NAME	STATUS
1	James	ACT



注意: 此时 PL/SQL Developer 中, 关于当前会话的【Commit】和【Roll Back】按钮将变为有效状态, 如图 20-8 所示。

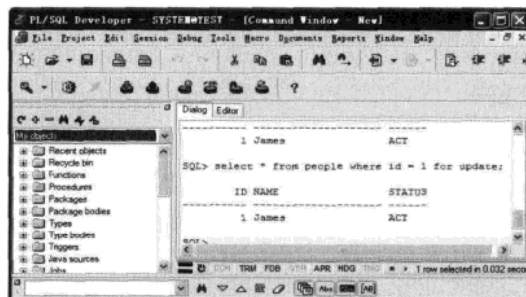


图 20-8 使用 for update 选项指定悲观锁

(2) 在新的【Command Window】中, 使用相同的 DML 语句, 尝试锁定同一行:

```
SQL> select * from people where id = 1 for update;
```

此时, Oracle 将会使该会话中的事务处于阻塞状态, 如图 20-9 所示。

(3) for update 锁定是行级锁定, 这意味着在会话中 1 仅仅锁定了 id 为 1 的行。因此, 如果在会话 2 中尝试锁定 id 为 2 的行, 将能够成功完成, 如图 20-10 所示。

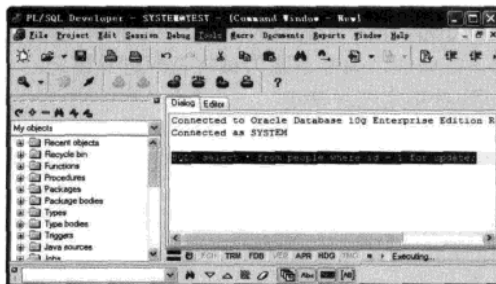


图 20-9 尝试锁定已经被悲观锁锁定的记录, 将使事务处于阻塞状态

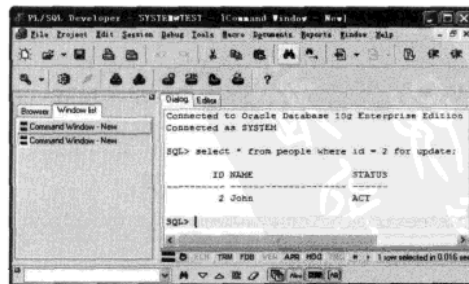


图 20-10 在会话 2 中锁定 ID 为 2 的行



从以上实例可以看出,如果尝试使用悲观锁来锁定已被锁定的记录。会话将会一直处于阻塞状态,用户不知道什么时候才能获得锁。相比之下,用户可能更希望能马上返回错误提示,然后转而执行其他动作。此时,可以使用 `no wait` 选项来要求 Oracle 不必等待,此时 Oracle 将立即抛出错误。例如,在会话 2 中利用 `no wait` 选项来锁定 id 为 1 的记录。

```
SQL> select * from people where id = 1 for update nowait;

select * from people where id = 1 for update nowait

ORA-00054: resource busy and acquire with NOWAIT specified
```

【代码说明】如果使用了 `nowait` 选项,一旦锁定失败,立即抛出异常,而不必等待。

20.2.2 乐观锁定

与悲观锁定不同,乐观锁定假定每次更新数据时,都假定可以获得对相应数据的锁定。而在真正更新时,首先要判断要更新的数据是否仍然存在。如果存在,才进行更新,否则将导致更新失败。这种更新实际上是通过 `where` 子句来实现的。在 `where` 子句中包含了所有列作为筛选条件,这样将能够判断数据是否仍然一致。

【范例 20-4】演示乐观锁定。

两个不同的用户同时使用了如下 SQL 语句:

```
SQL> select * from people where id = 3;
```

ID	NAME	STATUS
3	Robert	ACT

那么在前台页面中,将同时显示出 id 为 3 的人员的基本信息。此时,两个用户都尝试对其进行修改,例如,用户 1 试图将 name 列值“Robert”修改为“Bin”,而用户 2 试图将 name 列值“Robert”修改为“Charles”。如果使用常规语句来更新表,将造成更新遗失。可以在 PL/SQL Developer 中利用两个独立的【Command Window】进行演示。

(1) 在第一个【Command Window】中执行更新命令,并提交:

```
SQL> update people set name = 'Bin' where id = 3;
```

```
1 row updated
```

```
SQL> commit;
```

```
Commit complete
```

(2) 接着在第二个【Command Window】中执行用户 2 的更新命令,并提交。

```
SQL> update people set name = 'Charles' where id = 3;
```

```
1 row updated
```

```
SQL> commit;
```

```
Commit complete
```

(3) 很明显,此时表 `people` 中 id 为 3 的记录,其 name 列值为“Charles”:

```
SQL> select * from people where id = 3;
```

ID	NAME	STATUS
3	Charles	ACT



用户 1 在自己的更新之后，看到了更新成功信息。但是，紧接着用户 2 同样更新了该记录，那么，当用户 1 再次查询时，其获得的信息为“Charles”，而非“Bin”。这就造成了更新遗失，而且这种更新遗失是在数据库中完成的，用户 1 和用户 2 都不会察觉到。

为了避免这种更新遗失，可以在 update 语句的 where 子句中添加所有列作为筛选条件。例如，用户 1 和用户 2 都利用查询语句获得 id 为 3 的记录，并将其显示到前台页面中。当更新时，利用页面中的值来作为查询条件。例如，在第一个【Command Window】中执行以下 SQL 语句，并进行提交。

```
SQL> update people set name = 'Bin' where id = 3 and name = 'Robert' and status = 'ACT';
```

```
1 row updated
```

```
SQL> commit;
```

```
Commit complete
```

此时，用户 2 在前台页面所见到的信息仍然为：id=1, name='Robert', status='ACT'。如果该用户试图将 name 列修改为“Charles”，那么相应的 SQL 语句如下所示：

```
SQL> update people set name = 'Charler' where id = 3 and name = 'Robert' and status = 'ACT';
```

```
0 rows updated
```

用户 1 已经将该记录的 name 修改为“bin”，因此，通过 where 语句的过滤，用户 2 所更新的记录数为 0。而此时，应用程序应该为用户返回“原记录已被其他用户更改”的提示。以便用户 2 再次进行查询，并进行修改，或者执行其他动作。

20.2.3 悲观锁定与乐观锁定的比较

悲观锁定与乐观锁定都是处理多用户并发问题的机制。二者区别如下。

1. 理念不同

悲观锁定在执行更改之前就已经假定所请求的资源已经被其他用户锁定，所以需要首先请求锁定。锁定成功之后才进行更改。而乐观锁定在执行更改之前假定所请求的资源处于未锁定状态，直接进行更新。

2. 性能不同

因为要首先尝试进行锁定，悲观锁定要耗费更多的资源。而乐观锁定直接进行更新，所需资源要少于悲观锁定。

3. 风险不同

对于悲观锁定，如果在尝试锁定时失败，而又使用了等待模式，那么用户很可能在极短的时刻成功获得资源的控制权。即使未能在较短时间内获得，同样可以使用超时来限制所等待的时间。

对于乐观锁定，当一次更新失败之后，用户会在自定义的时间之后再次尝试更新。而这种更新，则面临着再次更新失败的风险。更糟糕的情况是，用户会面临多次更新失败的风险，而对应用系统的功能产生怀疑。

4. 如何取舍

Oracle 虽然将悲观锁定限制在行级，但是，当系统运行繁忙，用户量较大时，悲观锁定



仍然会极大地影响系统性能。而乐观锁定虽然承受了较大的风险，但是性能较佳。而现在有很多更好的策略来代替在 where 子句中使用表的全部列。例如，在 Hibernate 中，提出了在表中增加名为 version 的列，每次更新时都会修改 version 的版本号；在 Oracle 也可以通过时间戳函数来判断记录是否已经被更新。当然这些替代途径与乐观锁定的基本处理方式在本质上是相同的。

总之，悲观锁定侧重于安全性，而乐观锁定则侧重于较高的性能。

20.2.4 锁定转换

在 SQL Server 等其他数据库中，对于锁的维护是通过链表的数据结构。而链表容量都会出现瓶颈，一旦链表的容量超过了最大限制，将导致锁定的升级。例如，将某个表上的 100 个行级锁定升级为表级锁定。如果升级成功，那么整个表都被锁定，其他用户将无法访问该表。当然，如果其他用户也在其他行上设置了独占锁，那么锁定升级将失败。数据库系统只能花费更多的时间来解决锁定升级问题。

Oracle 中，并不执行锁定升级，而是使用另外一种处理策略——锁定转换。锁定转换与锁定升级最大的不同在于，锁定转换总是尽可能将锁定限制在较低的级别上。例如，在使用 for update 语句锁定表中的某些行之后，除了在被锁定行上具有独占锁之外，也会在整个表上放置 row share table lock（在表上放置共享锁的目的是，其他会话无法获得该表的独占锁，防止获得表独占锁的会话修改表结构。因为修改了表结构，将影响针对数据的 update 操作）。这样，并不影响其他用户针对该表的查询等操作。当实际进行数据更新时，将表的 row share table lock 转换为 row exclusive table lock，并执行更新操作。



20.3 并发控制的其他方法

并发控制除了进行锁定之外，仍然有其他间接方法可以解决该问题。例如，创建临时表，并在表中创建一个用于标识用户的列。当进行数据更新时，可以首先更新临时表中数据，数据真正被确认之后，再从临时表中搬迁到实际表。

【范例 20-5】 演示利用临时表来实现并发控制。

(1) 创建一个临时表 tmp_people:

```
SQL> create table tmp_people (id number, name varchar2(20), status varchar2(3),  
user_id integer);
```

Table created

在表 tmp_people 中，除了具有与表 people 相同的列外，还有额外列 user_id，该列用于存储用户的 id。

(2) 在 PL/SQL Developer 中新建一个【Command Window】，并假定为用户 1 所用。那么当用户 1 试图更新表中数据时，则可以利用以下 SQL 语句：

```
SQL> delete from tmp_people where user_id = 1;
```

0 rows deleted

```
SQL> insert into tmp_people (id, name, status, user_id)  
2 values (13, 'Bruce', 'ACT', 1);
```

1 row inserted



```
SQL> commit;
```

```
Commit complete
```

(3) 当用户 1 保存记录时, 首先删除表 tmp_people 中所有 user_id 为 1 的记录, 然后将新的记录插入, 并进行提交。当用户提交时, 才真正将数据插入或者更新到表 people 中。

(4) 对于用户 2, 采用同样的操作策略。那么, 在 tmp_people 中实现了利用 user_id 来隔离多个用户。而当用户提交更改时, 才会将数据更新到真实的数据表中。

(5) 利用临时表处理并发, 实际上是将多用户并发的部分工作转嫁到临时表中, 从而减轻真实数据表的并发负担。另一方面, 当用户保存数据之后, 在提交之前又放弃了修改, 采用临时表策略, 不会操作真实的数据表, 因此, 也在一定程度上减轻了真实数据表的负担。



20.4 本章实例

Oracle 会自动处理锁定及锁的转换, 本节将通过一个实例来查看锁定及锁定转换。

【范例 20-6】 演示锁定及锁的转换。

(1) 锁 SHARE TABLE LOCK 是一个表级锁。当对某个表加载了 SHARE TABLE LOCK, 那么该表将成为只读表。在 PL/SQL Developer 中新建【Command Window】, 用以模拟 Session 1。

```
SQL> lock table people in share mode;
```

```
Table(s) locked
```

【代码说明】 lock table people 用于为表 people 手动加锁; in share mode 表明该锁是一个共享锁 (SHARE TABLE LOCK)。

(2) 在 PL/SQL Developer 中新建【Command Window】, 用以模拟 Session 2, 并尝试对表 people 进行 DML 操作。

```
SQL> select * from people order by id;
```

ID	NAME	STATUS
1	James	ACT
2	John	ACT
3	Bin	ACT
4	Michael	ACT
5	William	ACT
6	David	ACT
7	Richard	ACT
8	Charles	ACT
9	Joseph	ACT
10	Thomas	ACT
11	yoyo	ACT
12	Jordan	ACT
13	Oracle	ACT
14	Owen	ACT

```
14 rows selected
```

可见, 在加载了 SHARE TABLE LOCK 之后, 可以进行正常的查询操作。当执行修改表中数据时, 将会导致阻塞, 如图 20-11 所示。

当然, 用户可以在 Session 1 中更新表 people。

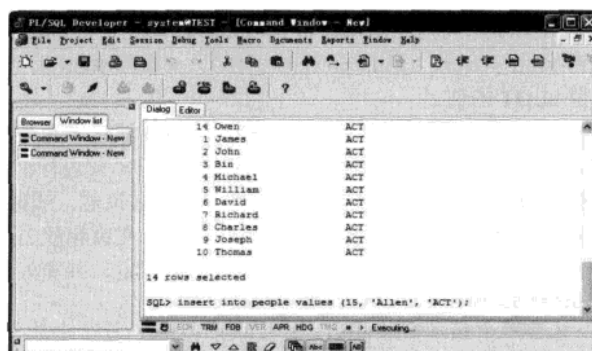


图 20-11 SHARE TABLE LOCK 使表 people 对其他事务只读

(3) 在 Session 1 中回滚事务，以释放表 people 上的 SHARE TABLE LOCK。

```
SQL> rollback;
```

Rollback complete

当 Session1 回滚事务之后，表 people 上的 SHARE TABLE LOCK 锁被释放，那么 Session 2 的插入操作将能够成功执行。同样回滚 Session 2 的操作。

```
SQL> rollback;
```

Rollback complete



注意：以上操作只是为了说明 SHARE TABLE LOCK 锁的作用。

(4) 在 Session 1 中使用 for update 来锁定表 people 中的记录。

```
SQL> select * from people where id = 1 for update;
```

ID	NAME	STATUS
1	James	ACT

(5) 查看此时数据库中锁的情况。

```
SQL> select s.username,
2 s.sid,
3 trunc(l.id1/power(2,16)) rbs,
4 mod(l.id1, power(2,16)) slot,
5 l.id2 seq,
6 l.type,
7 l.lmode,
8 l.request,
9 l.block
10 from v$lock l, v$session s
11 where l.sid = s.sid
12 and s.username = 'SYSTEM'
12 /
```

USERNAME	SID	RBS	SLOT	SEQ	TYPE	LMODE	REQUEST	BLOCK
SYSTEM	145	0	51644	0	TM	2	0	0
SYSTEM	145	4	4	7888	TX	6	0	0

【代码说明】v\$lock 实际存储了当前数据库中锁的信息；v\$session 实际存储了当前数据库



的会话信息；而锁的创建则是在事务中形成，一个事务实际由三个参数唯一确定，即（RBS, SLOT 和 SID），本例中使用了 $\text{trunc}(\text{lid1}/\text{power}(2,16))$ 来获得 RBS 的值，利用 $\text{mod}(\text{lid1}, \text{power}(2,16))$ 来获得 SLOT 的值。

分析查询结果可知，在当前数据库中，用户“SYSTEM”创建了两个锁——TYPE 分别为 TM 和 TX，LMODE 分别为 2 和 6。其中 TM 代表表级锁，而 TX 代表行级锁；LMODE 为 2 代表该锁是一个 ROW SHARE TABLE LOCK，即表中某些行已被锁定，或称部分锁定，LMODE 为 6 代表该锁是一个 ROW EXCLUSIVE LOCK，即行独占锁，在锁释放之前，其他事务将无法获得该行上的独占锁。REQUEST 列的值均为 0，代表已成功锁定，并非处于排队请求锁状态。



注意：1. 注意两个锁之前的区别，一个为表级锁，另一个为行级锁。

2. 毫无疑问，此时的两个锁都是创建在表 people 之上的。当然，也可以通过如下 SQL 语句进行验证。

3. Oracle 数据表的每一行都会有一个锁定标志位。Oracle 在获得行级锁时，会首先寻找到该行，并查看行的锁标志位。

```
SQL> select
2  o.object_name,
3  o.object_type
4  from v$locked_object l, all_objects o
5  where l.session_id = 145 and l.object_id = o.object_id
6  /
```

OBJECT_NAME	OBJECT_TYPE
PEOPLE	TABLE

(6) 此时，在 Session 2 中可以为表添加 SHARE TABLE LOCK，因为表 people 已有的锁 ROW SHARE TABLE LOCK 是不会阻止为表添加 SHARE TABLE LOCK 的。

```
SQL> lock table people in share mode;
```

Table(s) locked

(7) 此时，在 Session 1 中，尽管已经锁定了表 people 中的第一行，在该记录上获得了独占锁，但是此时的表 people 已经被放置了 SHARE TABLE LOCK 锁，因此，更新操作将会被阻塞，如图 20-12 所示。

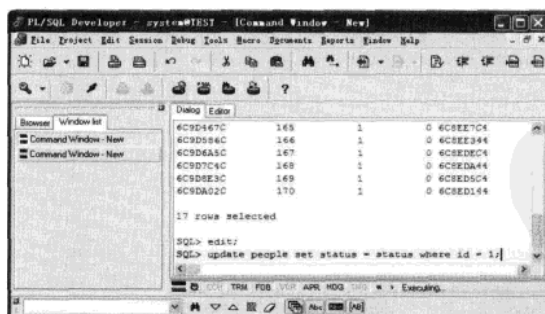


图 20-12 SHARE TABLE LOCK 对 Session 1 的影响

(8) 在 Session 2 中回滚操作，以释放表 people 的 SHARE TABLE LOCK。

```
SQL> rollback;
```

Rollback complete

(9) Session 1 中的更新操作将成功执行。

```
SQL> update people set status = status where id = 1;
```

1 row updated

(10) 重新查询数据库中锁的状态。

```
SQL> select s.username,
2 s.sid,
3 trunc(l.id1/power(2,16)) rbs,
4 mod(l.id1, power(2,16)) slot,
5 l.id2 seq,
6 l.type,
7 l.lmode,
8 l.request,
9 l.block
10 from v$sql l, v$session s
11 where l.sid = s.sid
12 and s.username = 'SYSTEM'
12 /
```

USERNAME	SID	RBS	SLOT	SEQ	TYPE	LMODE	REQUEST	BLOCK
SYSTEM	145	0	51644	0	TM	3	0	0
SYSTEM	145	4	4	7888	TX	6	0	0

分析查询结果可知，此时锁的状态发生了少许变化——表级锁的 LMODE 由 2 变为 3，这意味着 Oracle 执行了锁的自动转换。真实的情况是由 ROW SHARE TABLE 锁转换为了 ROW EXCLUSIVE TABLE 锁。

(11) 验证 ROW EXCLUSIVE TABLE 锁与 ROW SHARE TABLE 锁的区别。

各种锁的不同在于允许执行的操作不同。在步骤(6)中已经看到，可以为具有 ROW SHARE TABLE 锁的表添加 SHARE TABLE LOCK，现再次尝试为表 people 添加 SHARE TABLE LOCK，如图 20-13 所示。

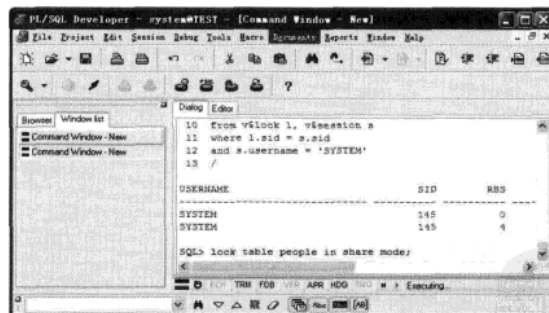


图 20-13 为表 people 添加 SHARE TABLE LOCK 被阻塞

可见，当执行锁转换之后，不能为表添加 SHARE TABLE LOCK。这是合理的设计，当数据表已经在某个事务中进行了修改，当然不能将其再次设置为只读（SHARE TABLE LOCK）。



注意：在表级锁 row share table lock 转换为 row exclusive table lock 之后的影响，不止阻塞为表添加 SHARE TABLE LOCK。此处只是以 SHARE TABLE LOCK 为例，说明如何锁定转换的影响。



20.5 本章小结

本章首先讲述了并发与锁定的概念。在锁定中，又讲述了悲观锁定与乐观锁定的区别。需要注意的是，Oracle 中不存在锁定升级，这也是其性能高于其他数据库系统的一个重要原因。最后，提出了处理高并发问题的临时表策略，并对该策略进行了简要分析。



20.6 习题

1. 什么是死锁？
2. 什么是悲观锁定和乐观锁定？
3. 比较悲观锁定与乐观锁定。
4. 如何查看对某个表执行更新操作时，数据库中相关锁的情况？



第 21 章 Oracle 中的正则表达式

在各种编程语言中，正则表达式都有着广泛的应用。Oracle 10g 及以后的版本中也支持正则表达式，并提供了非常实用的函数。本章首先讲述正则表达式的基本知识，然后讲述 Oracle 中的正则表达式函数。本章的主要内容包括：

- 正则表达式基础知识；
- Oracle 中的正则表达式函数。

通过本章的学习，读者可以掌握正则表达式的基础知识和 Oracle 中的主要正则表达式函数。这对其他编程语言的学习同样具有参考意义。



21.1 正则表达式简介

正则表达式，就是以某种模式来匹配一类字符串。一旦概括了某类字符串，那么正则表达式即可用于针对字符串的各种相关操作。例如，判断匹配性，进行字符串的重新组合等。正则表达式提供了字符串处理的快捷方式。

22.1.1 正则表达式与通配符

正则表达式是一个模式，正则表达式可以看做一个模糊的字符串匹配；而通配符也可以看做模糊的字符串匹配。

正则表达式与通配符虽有相似之处，但却有着本质的区别。正则表达式中不仅存在着代表模糊字符的特殊字符，而且存在着量词等修饰字符，使得模式的控制更加灵活和强大。另外，通配符的使用一般是在特定的环境下，例如 Windows 的查询功能等，在不同的环境下，通配符有可能不同；而正则表达式，不但广泛应用于各种编程语言，而且在各种编程语言中，保持了语法的高度一致性。

22.1.2 正则表达式与编程语言

在各种编程语言中，可以说，正则表达式是无处不在的。正则表达式在各种编程语言中高度统一，都遵循一致的语法。因此，一旦熟悉了一种编程语言中的正则表达式，那么，可以直接应用于其他编程语言。编程语言同时也是正则表达式的载体，没有编程语言，正则表达式的作用将无从发挥。

需要注意的是，正则表达式提出的本意是处理文本，因此在编程语言中，与之相关的函数基本包括：文本匹配、文本截取、文本替换、文本分割等。文本分割是指将一个字符串按照分割规则生成新的复杂数据结构。一种常见的应用为，利用逗号将字符串分割为数组。但是 Oracle 中并不存在复杂数据结构，因此，在 Oracle 中，正则表达式的应用主要包括：文本匹配、文本截取和文本替换。



21.2 正则表达式基础知识

正则表达式简单易学。本节将从以下几方面讲述正则表达式的基础知识。

- 元字符和普通字符;
- 量词;
- 字符转义与字符类;
- 字符组;
- 正则表达式分支。

21.2.1 元字符和普通字符

编程语言与普通文本文件的区别在于, 编程语言中定义了保留字。这些保留字, 都被编译器当做特殊命令来处理。其余才被当做普通字符串进行处理。

正则表达式的定义是以字符为基本单位的。这些字符也可以分为“保留字符”与普通字符两类, 分别称为元字符和普通字符。元字符是指在正则表达式中表示特殊含义的字符。正则表达式中的元字符包括“.”、“\”、“?”、“*”、“+”、“|”、“{”、“}”、“^”、“\$”、“[”、“]”。

例如, 元字符中的“.”用于匹配任何单字符(换行符除外); “\”可以与普通字符组合来表示特殊字符, 也可与元字符组合使用来获得元字符的原义字符。元字符“^”和“\$”用于匹配输入字符串的开始和结束。默认情况下, 这里的开始和结束是指整个字符串的开头和结尾。

普通字符是指除元字符外的所有 Unicode 字符。普通字符匹配其本身。例如, 字符“a”代表原义字符“a”。

21.2.2 量词

量词用来指定量词的前一个字符出现的次数。量词的形式主要有“?”、“*”、“+”、“{ }”。

- 元字符“?”作为量词出现, 用来匹配 0 个或 1 个字符。例如 A?, 表示 0 个或 1 个字符“A”。
- 元字符“*”作为量词出现, 用来匹配 0 个或多个字符。例如 A*, 表示 0 个或多个字符“A”。
- 元字符“+”作为量词出现, 用来匹配 1 个或多个字符。例如 A+, 表示 1 个或多个字符“A”。
- 元字符“{”和“}”同时出现, 用来匹配指定个数的字符, 其形式有三种情况: A{3}表示匹配三个字符“A”; A{3,}表示匹配三个或更多个字符“A”; A{3,5}表示匹配 3 到 5 个字符“A”。

量词在用于匹配字符串时, 默认遵循贪婪原则。贪婪原则是指尽可能多地匹配字符。例如字符串“Function(p),(OK)”, 如果使用正则表达式“(\\(.*)”进行匹配, 则得到字符串“(p),(OK)”, 而非“(p)”; 若欲得到“(p)”, 则必须取消量词的贪婪原则, 此时只需要为量词后追加另外一个数量词“?”即可。如上面的正则表达式应该改为“(\\(.?))”。

21.2.3 字符转义与字符类

字符转义是指通过元字符“\”与其他字符进行组合。这里的“其他字符”包括了元字符和普通字符。

元字符在正则表达式中有特殊含义。如果需要使用其原义, 则需要用到字符转义。字符转

义使用字符“\”来实现。其语法模式为：“\”+元字符。例如，“\.”表示普通字符“.”；“\doc”匹配字符串“.doc”；而普通字符“\”需要使用“\\”来表示。

字符类是可选的字符集合。字符转义是实现字符类的一种模式。字符转义实现的字符类可以分为两类，一类为单字符匹配，另一类为多字符匹配。单字符匹配是指字符集合中仅含有一个字符，而多字符匹配是指字符集合中含有多个字符。单字符匹配有以下几种。

- “\n”：用于匹配换行符（x0A）。
- “\r”：用于匹配回车符（x0D）。
- “\t”：用于匹配制表符（x09）。
- 元字符“.”、“\”、“?”、“*”、“+”、“|”、“{”、“}”、“^”、“\$”、“[”、“]”可分别加前缀字符“\”来实现转义，表示字符本身。

多字符匹配有以下几种。

- “\s”：用于匹配空白符。这里所说的空白符包括空格（x20）、制表符（x09）、回车符（x13）、换行符（x0A）。
- “\d”：用于匹配数字。例如，`matches("1","d")`将返回 `true`，而 `matches("a","d")`将返回 `false`。
- “\w”：用于匹配可用来组成单词的字符。例如，`matches("Z","w")`将返回 `true`，而 `matches("?", "d")`将返回 `false`。

除此之外，还可以使用“[]”来自定义字符类。例如，`[az]`可用于匹配字符 `a` 或 `z`，`[a-z]`用于匹配字符“a”到字符“z”的任意字符，`[a-z0-9]`用于匹配字符“a”到字符“z”或字符“0”到字符“9”中的任意字符。

“[]”实际上定义了某个范围内的字符。如果需要表示该范围之外的字符集合，可以使用字符“^”。例如，`[^a-z]`可用于表示除“a”到“z”之外的所有 Unicode 字符。需要注意，这里的“^”与行首匹配符“^”的区别。行首匹配符必须出现在正则表达式的开头，而表示补集的“^”必须出现在“[]”之内。

21.2.4 字符组的使用

字符组是指将模式中的某些部分作为一个整体。这样，量词可以用来修饰字符组，从而提高正则表达式的灵活性。例如，使用元字符和普通字符的组合，很难针对字符串“abc123abc123abc123”来归纳正则表达式。此时可以利用字符组来实现，相应的正则表达式可以表示为“(abc123)+”，表示连续的多个“abc123”组合。

字符组的另一个重要作用在于，许多编程语言中，可以利用“\$1”、“\$2”等来获取第一个、第二个字符组，即所谓的后向引用。在 Oracle 中，引用格式为“\1”、“\2”。

21.2.5 正则表达式分支

可以利用“|”来创建多个正则表达式分支。例如，“\d{4}\w{4}”可以看做两个正则表达式——“\d{4}”和“\w{4}”，匹配其中任何一个正则表达式的字符串都被认为匹配整个正则表达式。如果该字符串两个正则表达式分支都匹配，那么将被处理为匹配第一个正则表达式分支。

21.2.6 Oracle 中正则表达式的特殊性

在 Oracle 中，正则表达式的用法与标准用法略有不同。这种不同主要体现在对于字符类的定义上。Oracle 中不使用字符“\”与普通字符组合的形式来实现字符类，而是具有其特殊语法。

- `[[:alpha:]]`：表示任意字母，正则表达式的一般语法为 `\w`。



- `[:digit:]`: 表示任意数字, 正则表达式的一般语法为 `\d`。
- `[:alnum:]`: 表示任意字母和数字。
- `[:space:]`: 表示任意空白字符, 正则表达式的一般语法为 `\s`。
- `[:upper:]`: 表示任意大写字母。
- `[:lower:]`: 表示任意小写字母。
- `[:punct:]`: 表示任意标点符号。
- `[:xdigit:]`: 表示任意 16 进制的数字, 相当于 `[0-9a-fA-F]`。



21.3 正则表达式在 Oracle 中的应用

Oracle 10g 及以后的版本都提供了对正则表达式的支持。而这种支持, 主要是通过提供函数来体现的。Oracle 中共有四个正则表达式相关函数, 它们分别是:

- `regexp_like()`
- `regexp_instr()`
- `regexp_substr()`
- `regexp_replace()`

21.3.1 `regexp_like()` 的使用

`regexp_like()` 函数的常用形式为 `regexp_like(expression, regexp)`。其中第一个参数为字符串, 可以是数据表中的列或者表达式获得的字符串; 第二个参数也是一个字符串, 该参数用于表示进行匹配的正则表达式。函数的返回值为一个布尔值。如果第一个参数匹配第二个参数所代表的正则表达式, 那么将返回真, 否则将返回假。

【范例 21-1】 演示 `regexp_like()` 函数的使用。

```
SQL> select * from people where regexp_like(name, '^J.*$');
```

ID	NAME	STATUS
1	James	ACT
2	John	ACT
9	Joseph	ACT

【代码说明】 `regexp_like(name, '^J.*$')` 用于判断列 `name` 的值是否匹配正则表达式 `“^J.*$”`; 该正则表达式表示以字符 “J” 开头, 后跟任意个数的任意字符。

可以利用 `like` 判式来实现同样的功能, 但是写法较为拖沓:

```
SQL> select * from people where name like 'J%';
```

ID	NAME	STATUS
1	James	ACT
2	John	ACT
9	Joseph	ACT

【代码说明】 “J%” 表示以字符 “J” 开头, 后跟任意字符组合。需要注意的是, 此处使用的是通配符, 而非正则表达式语法。

21.3.2 `regexp_instr()` 的使用

`regexp_instr()` 函数的常用形式为 `regexp_instr(expression, regexp, startindex, times)`。其中, 第



一个参数 `expression` 为字符串，可以是表中的列或者字符串表达式；第二个参数 `regexp` 为字符串，表示要进行匹配的正则表达式；第三个参数 `startindex` 表示开始进行匹配比较的位置；第四个参数表示第几次匹配作为最终匹配结果。其中第三个和第四个参数为可选参数，默认值均为 1。该函数用于获得正确匹配时，指针在字符串 `expression` 中的位置。

【范例 21-2】 `regexp_instr()` 函数的用法。

```
SQL> select regexp_instr('12.158', '\.') position from dual;
```

```
POSITION
-----
3
```

【代码说明】 `regexp_instr('12.158', '\.')` 用于获取第一个小数点的位置。正则表达式 “\.” 表示普通字符 “.”。

21.3.3 regexp_substr() 的使用

`regexp_substr()` 函数的常用形式为 `regexp_substr(expression, regexp)`。该函数的第一个参数为字符串，可以是表中的列或者字符串表达式；第二个参数为正则表达式。该函数用于返回第一个字符串参数中，与第二个正则表达式参数相匹配的子字符串。

【范例 21-3】 演示 `regexp_substr()` 的使用。

首先创建测试表，并向其中插入测试数据：

```
SQL> create table html(id integer, html varchar2(2000));
```

```
Table created
```

```
SQL> insert into html values (1, '<a href="http://mail.google.com/2009/1009.html">mail link</a>');
```

```
1 row inserted
```

```
SQL> select * from html;
```

```

      ID                                HTML
-----
1      <a href="http://mail.google.com/2009/1009.html">mail link</a>

```

表 `html` 中存储了 HTML 标签及内容。现欲从标签 `<a>` 中获得链接的 `url`，那么可以利用 `regexp_substr()` 函数。

```
SQL> select id, regexp_substr(html, 'http[a-zA-Z0-9\.:/*]*') url from html;
```

```

      ID                                URL
-----
1      http://mail.google.com/2009/1009.html

```

【代码说明】 正则表达式 `http[a-zA-Z0-9\.:/*]*` 表示，以字符串 “http” 开头，后跟任意个数的大小写字母、数字、小数点、反斜杠或冒号；`regexp_substr()` 函数返回符合正则表达式的子字符串。

21.3.4 regexp_replace() 的使用

`regexp_replace()` 函数的常用形式为 `regexp_replace(expression, regexp, replacement)`。其中，第一个参数 `expression` 为字符串类型，可以是表中的列或者字符串；第二个参数 `regexp` 为一个



字符串，是用来进行匹配操作的正则表达式；第三个参数 `replacement` 是一个字符串，该字符串用于替换 `expression` 中的匹配部分。

值得注意的是，在参数 `replacement` 中，可以含有后向引用，以便将正则表达式中的字符组重新捕获。例如，某些国家和地区的日期格式可能为“MM/DD/YYYY”，那么可以利用 `regexp_replace()` 函数来转换日期格式。

【范例 21-4】 演示 `regexp_replace()` 的使用。

```
SQL> select regexp_replace('09/29/2008', '^([0-9]{2})/([0-9]{2})/([0-9]{4})$',
'\3-\1-\2') replace from dual;
```

```
REPLACE
-----
2008-09-29
```

【代码说明】 正则表达式 “`^([0-9]{2})/([0-9]{2})/([0-9]{4})$`” 实际用于匹配整个字符串，因为正则表达式以 `^` 开始，以 `$` 结束。该正则表达式使用了三个分组，然后利用 “`\3-\1-\2`” 将三个分组从缓存中输出，并将三个字符组重新排序，以替换原有字符串。其中的 “`-`” 为普通字符。



21.4 本章实例

在进行正则表达式匹配时，还可以忽略字符大小写形式进行匹配。这是比使用 `LIKE` 判式更加灵活和方便之处。

【范例 21-5】 演示如何在正则表达式匹配时忽略大小写形式。

```
SQL> select * from people where regexp_like(name, 'or');
```

```
  ID      NAME      STATUS
-----
  12      Jordan      ACT
```

【代码说明】 `regexp_like(name, 'or')` 用于对 `name` 列值进行匹配，匹配的正则表达式实际是一个字符串 “`or`”。

为了忽略大小写，为 `regexp_like` 指定第三个参数，代码修改如下：

```
SQL> select * from people where regexp_like(name, 'or', 'i');
```

```
  ID      NAME      STATUS
-----
  12      Jordan      ACT
  13      Oracle       ACT
```

【代码说明】 `regexp_like(name, 'or', 'i')` 中的第三个参数 “`i`”，代表 “`ignore`”，表示进行正则表达式匹配时忽略字符的大小写形式。

分析查询结果可知，此时所获得的记录中含有 “`Oracle`”，其中的 “`Or`” 即为忽略大小写形式之后的匹配结果。



21.5 本章小结

本章讲述了正则表达式的基本知识。正则表达式语法在各种编程语言中具有高度的一致性。只是在不同的编程环境中略有少许差异。因此，学习时，应当注重把握其一般语法，再结合



环境的特殊性。本章还讲述了正则表达式与通配符的区别，并展示了正则表达式的灵活性和不同用法。其中，特别值得关注的是后向引用的用法。



21.6 习题

1. 简述正则表达式与通配符的区别。
2. 简述正则表达式匹配时的贪婪原则。
3. 简述 Oracle 正则表达式的特殊性。
4. 利用 `regexp_replace()` 函数将 YYYY-MM-DD 格式的字符串，转换为 MM/DD/YYYY 格式。



第五篇 Oracle 与编程语言 综合使用实例

第 22 章 Oracle 在 Java 开发 中的应用

Oracle 是数据库系统，因此，应用程序要调用 Oracle 数据库，需要首先进行连接，并根据特定语法进行操作。编程语言的不同，决定了与 Oracle 的相关操作的不同。本章主要讲述 Oracle 在 Java 开发中的应用。本章的主要内容包括：

- 利用 JDBC 连接 Oracle 数据库；
- 利用 JDBC 操作 Oracle 数据库；
- Hibernate 简介；
- 利用 Hibernate 操作 Oracle 数据库。

通过本章的学习，读者可以掌握利用两种不同的方式——JDBC 和 Hibernate 来操作 Oracle 数据库。



22.1 通过 JDBC 使用 Oracle

一种数据库可以为多种编程语言所用；同样，一种编程语言中也可以使用多种数据库。在 Java 语言中，提供了 JDBC 方式来访问和操作各种数据库。

22.1.1 JDBC 简介

JDBC 全称为 Java DataBase Connectivity standard，它是一个面向对象的应用程序接口 (API)，通过它可访问各类关系数据库。JDBC 也是 Java 核心类库的一部分。

JDBC 相当于访问数据库的模板，它独立于具体的关系数据库。Java 提供了若干类来处理数据库操作。例如提供了 Connection 类来获得数据库连接，提供了 Statement 来封装 SQL 语句，提供了 ResultSet 类来存储由数据库返回的结果集合。

针对不同的数据库，需要不同的数据库驱动支持。而这些支持类大多由数据库厂商提供。数据库驱动中封装了数据库相关操作的所有类。

通常，Java 程序首先使用 JDBC API 来与 JDBC Driver Manager 交互。由 JDBC Driver Manager 载入指定的 JDBC 驱动，然后建立数据库连接，最后通过 JDBC API 来操作数据库。

22.1.2 准备工作

在进行实际的开发之前，需要按照以下步骤完成准备工作。

- ① 安装 JDK 及 JRE 环境，并设置 JAVA_HOME 和 CLASSPATH 环境变量。本例中使用的 JDK 版本为 1.6。
- ② 安装 Eclipse，并在 Eclipse 中创建新的测试项目——test，如图 22-1 所示。
- ③ 在互联网上下载 Oracle 10g 的 JDBC 驱动程序的 jar 包，在本例中文件名为 ojdbc14_10g.jar。
- ④ 在项目目录 test 下创建名为 lib 的子目录，并将 ojdbc14_10g.jar 复制到该目录下，如图 22-2 所示。

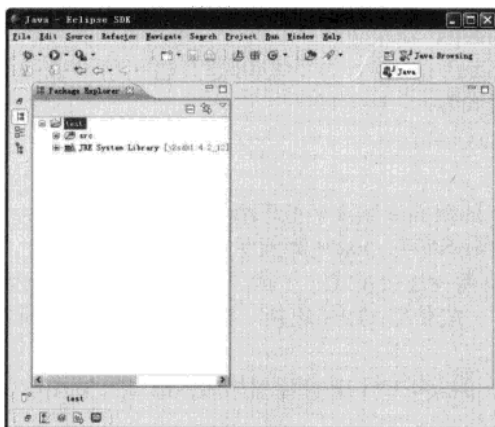


图 22-1 在 Eclipse 中创建测试项目——test

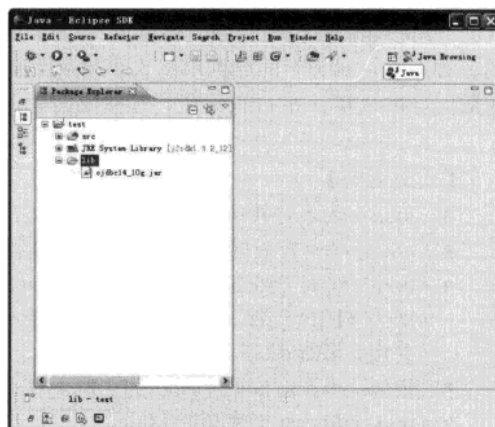


图 22-2 复制 Oracle JDBC 驱动包

- ⑤ 将 ojdbc14_10g.jar 添加到 test 项目的 build path 中，如图 22-3 所示。
- ⑥ 此时，项目 test 的结构如图 22-4 所示。

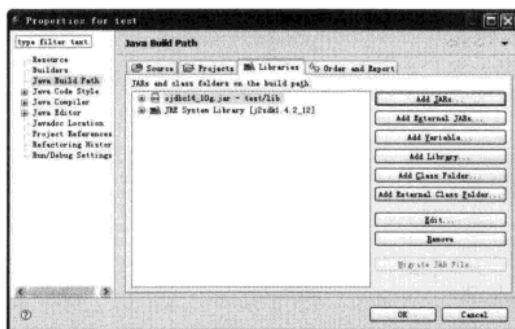


图 22-3 将 jdbc 驱动程序添加到 build path 中

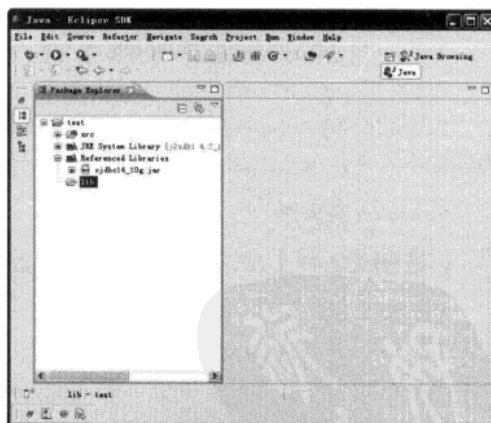


图 22-4 准备工作完成后的 test 项目

22.1.3 JDBC 连接 Oracle

在操作 Oracle 之前，首先要与 Oracle 数据库建立连接。利用 JDBC 建立于 Oracle 连接的步骤如下。



1. 加载 JDBC 驱动

对于下载获得的 JDBC 驱动，必须首先将其加载到 JVM 并执行初始化。因此，应该利用如下 Java 代码加载 JDBC 载驱动。

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

【代码说明】Class.forName 用于加载 Java 类到 JVM 中；oracle.jdbc.driver.OracleDriver 指定了要加载的 Java 类的完整路径。

2. 连接字符串的使用

JDBC 连接任何数据库，都需要一个连接字符串。连接字符串包括了针对数据库连接所需的各种信息。对于不同数据厂商只要使用不同的连接字符串，而使用相同的代码，即可实现不同数据库的连接。在 Oracle 中，连接字符串的格式为：

```
String url = "jdbc:oracle:thin:@host:port:sid
```

【代码说明】

- jdbc：表示连接方式为 JDBC 连接。所有的数据库连接字符串都以“jdbc”开始。
- oracle：代表数据库类型，表示要连接的数据库为 Oracle 数据库。
- thin：代表 Oracle 的连接方式。Oracle 有两种连接模式，一种是 OCI 连接，另外一种即为 THIN 连接。对于 JDBC 中的连接，一般使用 THIN 连接，因为这种连接方式简单易用，稳定性高。
- @host：表示数据库所在的主机。在此，可以为主机 IP 或者局域网内的主机名称。
- port：表示数据库监听端口。
- sid：表示要连接的数据库 SID。

在本例中，使用的连接字符串如下：

```
String url = "jdbc:oracle:thin:@192.168.1.97:1521:test";
```

3. 获得数据库连接

可以利用 DriverManager 类的静态方法来获得数据库连接，代码如下：

```
Connection connection = DriverManager.getConnection(url, username, password);
```

【代码说明】DriverManager 类提供了静态方法 getConnection() 来获得数据库连接；其中 url 为连接字符串；username 为登录数据所使用的用户名；password 为用户名所对应的密码。

4. Java 类实例

【范例 22-1】演示如何获得 Oracle 数据库连接。

```
package com.test.oracle;

import java.sql.Connection;
import java.sql.DriverManager;

public class OracleTest {
    public static void main(String[] args) {
        String url = "jdbc:oracle:thin:@192.168.1.97:1521:test";
        String username = "system";
        String password = "abc123";

        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection connection = DriverManager.getConnection(url, username,
password);
```



```

        System.out.println(connection);
        connection.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

【代码说明】对于创建数据库连接的动作，需要使用 try-catch 语句块来捕获异常；在获得数据库连接之后，利用 System.out.println() 来打印该连接；另外，在 connection 使用完之后，需要利用 connection.close() 将其关闭，否则，该数据库连接将造成服务器资源的浪费。

在 Eclipse 中，利用 Run As Application 来查看其运行效果，如图 22-5 所示。

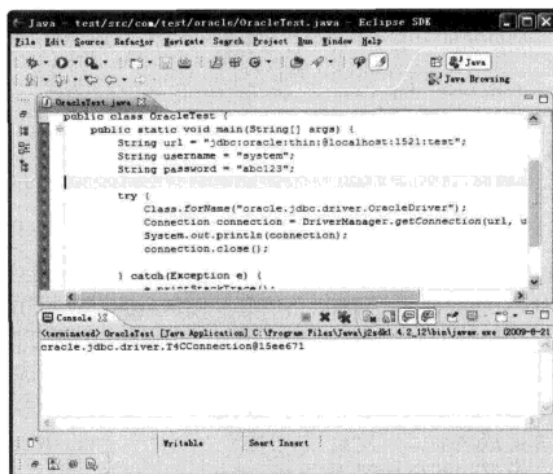


图 22-5 获得数据库连接

从运行结果可知，通过加载驱动、定义连接字符串，最后利用 DriverManager 即可获得数据库连接。

22.1.4 利用 JDBC 查询数据

在获得了数据库连接之后，可以利用该连接来创建表达式对象。表达式对象可以用于处理查询请求，并返回结果集合。

【范例 22-2】演示如何利用 JDBC 查询数据。

```

package com.test.oracle;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class OracleTest {
    public static void main(String[] args) {
        String url = "jdbc:oracle:thin:@192.168.1.97:1521:test";
        String username = "system";
        String password = "abc123";
        Connection connection = null;
    }
}

```



```
try {
    Class.forName("oracle.jdbc.driver.OracleDriver");
    connection = DriverManager.getConnection(url, username, password);

    Statement st = connection.createStatement();
    ResultSet rs = st.executeQuery("select * from people order by id");
    while (rs.next()) {
        System.out.println(rs.getString("ID") + ":" + rs.getString
("NAME"));
    }

} catch (Exception e) {
    e.printStackTrace();
} finally {
    if (connection != null) {
        try {
            connection.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

【代码说明】Statement st = connection.createStatement()利用 Connection 对象来创建表达式对象；ResultSet rs = st.executeQuery("select * from people order by id")则利用表达式对象执行 SQL 查询，查询的结果存储于结果集对象 rs 中；while (rs.next())则用于循环处理结果集中的记录；System.out.println(rs.getString("ID") + ":" + rs.getString("NAME"))打印记录中的 id 列与 name 列的信息；最后的 finally 语句块则是针对 connection 对象的较严密的处理形式，即保证 connection 在数据库操作完成之后，一定会被关闭。

在 Eclipse 中运行该程序，结果如图 22-6 所示。

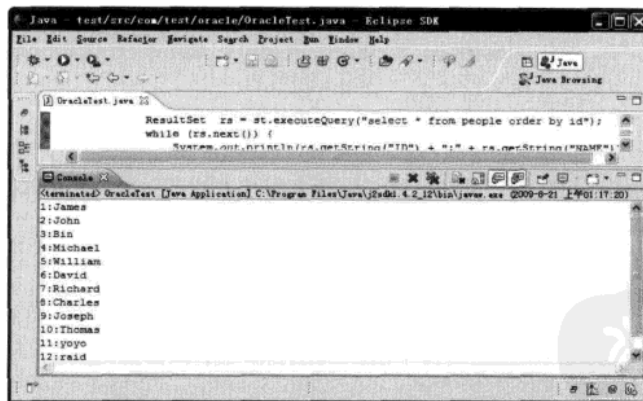


图 22-6 利用 JDBC 查询数据库

22.1.5 利用 JDBC 更新数据

数据的更新包括插入和修改。利用 JDBC 更新数据库的步骤与查询步骤基本相同。不同点在于，查询操作使用表达式对象的 executeQuery()方法，而更新操作，则使用表达式对象的 executeUpdate()方法。

**【范例 22-3】** 演示如何利用 JDBC 更新数据。

```
package com.test.oracle;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class OracleUpdate {
    public static void main(String[] args) {
        String url = "jdbc:oracle:thin:@192.168.1.97:1521:test";
        String username = "system";
        String password = "abc123";
        Connection connection = null;
        Statement st = null;
        ResultSet rs = null;

        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            connection = DriverManager.getConnection(url, username, password);

            st = connection.createStatement();
            st.executeUpdate("update people set name = 'Steven' where id = 12");
            st.executeUpdate("insert into people (id, name, status) values(13, 'Oracle', 'ACT') ");
            rs = st.executeQuery("select * from people where id = 12 or id = 13");
            while (rs.next()) {
                System.out.println(rs.getString("ID") + ":" + rs.getString("NAME"));
            }

            rs.close();
            st.close();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (connection != null) {
                try {
                    connection.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

【代码说明】 `st.executeUpdate("update people set name = 'Steven' where id = 12")` 通过表达式对象将 `people` 表中, `id` 为 12 的人员姓名修改为“Steven”; `st.executeUpdate("insert into people (id, name, status) values(13, 'Oracle', 'ACT')")` 则向表 `people` 中插入新的数据; `ResultSet rs = st.executeQuery("select * from people where id = 12 or id = 13")` 用于查询更新之后的数据。

在 Eclipse 中运行该实例程序, 结果如图 22-7 所示。

22.1.6 总结 JDBC 操作数据库

利用 JDBC 操作数据库, 首先要建立数据库连接。而不同的数据库, JDBC 提供的连接字符串模式不同, 因此, 应该根据不同数据库, 使用不同的连接字符串。

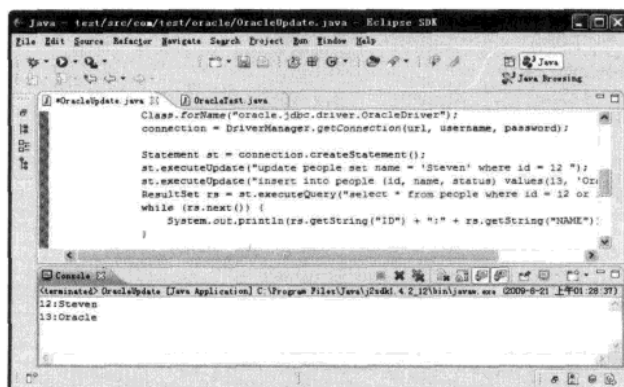


图 22-7 利用 JDBC 更新数据

在建立数据库连接之后，可以利用该连接来获得表达式对象。表达式对象可以向数据库传递操作语句，以实现与数据库的沟通。这里需要注意的是查询数据与更新数据，需要使用不同的方法。

对于数据库连接，在使用完毕之后，需要进行关闭操作。如果使用了 `ResultSet` 与 `Statement`，那么关闭的顺序应该是：`ResultSet`→`Statement`→`Connection`。

当然，利用 JDBC 数据库还有更多丰富的内容，例如，利用预编译的 `PreparedStatement` 对象，利用调用存储过程的 `CallableStatement` 对象等，本节不再进行详述。



22.2 通过 Hibernate 操作 Oracle 数据库

Hibernate 的底层同样利用了 JDBC 技术。Hibernate 框架的真实意义在于，将本来针对关系型数据库的操作，转化为 Java 中面对对象的操作。这样做的好处是，使面对对象的程序员的开发更加得心应手。另一方面，利用面对编程语言的特点，例如继承，可以使代码的重用性得到极大提高，提高开发效率。

22.2.1 准备工作

要使用 Hibernate，必须首先下载支持包。Hibernate 支持包的下载地址为：<http://www.hibernate.org>。Hibernate 的核心包为 `hibernate.jar`。除了 `hibernate.jar` 文件，还需要其他辅助文件。图 22-8 列出了需要下载的包。

名称	大小	类型	修改日期
antlr-2.7.5SH3.jar	424 KB	Executable Jar ...	2009-9-22 16:38
asm.jar	26 KB	Executable Jar ...	2009-9-22 16:07
cglib-2.1.2.jar	276 KB	Executable Jar ...	2009-9-22 16:06
commons-collections-2.1.1.jar	172 KB	Executable Jar ...	2009-9-22 16:04
commons-logging-1.0.4.jar	36 KB	Executable Jar ...	2009-9-22 16:04
dom4j-1.6.1.jar	307 KB	Executable Jar ...	2009-9-22 16:58
hibernate3.jar	2,222 KB	Executable Jar ...	2009-9-22 15:40
jta.jar	9 KB	Executable Jar ...	2009-9-22 16:35
ojdbc4_10g.jar	1,501 KB	Executable Jar ...	2009-9-22 16:05

图 22-8 Hibernate 应用支持包

22.2.2 配置 Hibernate

Hibernate 实现了从关系型数据库向 Java 对象的映射。而这种映射则是通过配置文件来实现的。在 Eclipse 中创建项目，并实现 Hibernate 配置的步骤如下所示。

- ① 在 Eclipse 中创建新的 Java 项目，并将该项目命名为“HibernateOracle”。
- ② 将所需 jar 包添加到项目的 Build Path 中，如图 22-9 所示。

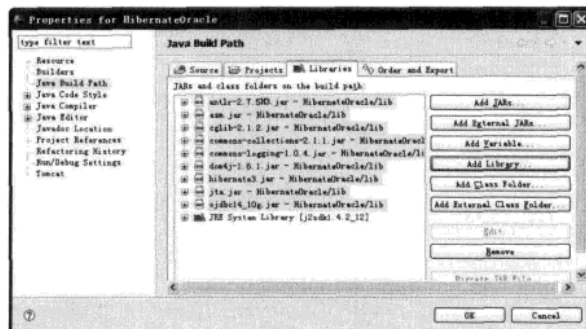


图 22-9 项目 HibernateOracle 的 Build Path

- ③ 在项目 HibernateOracle 的 src 目录下创建新的包 com.example.bean，并在该包下创建名为 People 的 Java 类。其源代码如下：

```
package com.example.bean;

public class People {
    private int id;
    private String name;
    private String status;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getStatus() {
        return status;
    }
    public void setStatus(String status) {
        this.status = status;
    }
}
```

【代码说明】该类是一个简单的 Java Bean。在该类中，定了三个属性 id、name 和 status，并为这三个属性提供 getter 与 setter 方法。该类提供了关系型数据库—对象映射中的对象端。

- ④ 在项目目录中的 src 子目录下创建名为 people.hbm.xml 的文件，该文件的源码如下：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0/EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.example.bean">
```



```

<class name="com.example.bean.People" table="people">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name"/>
    <property name="status"/>
</class>
</hibernate-mapping>

```

【代码说明】该文件实际是一个XML文件。在该XML文件中根元素为<hibernate-mapping>，根元素下包含了所有数据库——对象的映射配置；<class>元素的 name 属性指定了进行映射的类，table 属性指定了与该类进行映射的数据表；<id>元素指定该类的主键属性，name="id"指定该类的主键为属性 id；<generator>元素指定主键的生成规则，class="increment"指定主键的生成规则为自动增加；<property>元素指定属性与表中列的映射关系，name="name"指定类中的 name 属性将与数据表 people 中的同名列 name 实现对应；同样<property name="status"/>指定类 People 中的 status 属性与表 people 中的同名列进行对应；如果要实现非同名映射，则需要添加 column 属性，例如，<property name="status" column="column_name"/>。

⑤ 在项目目录下的 src 子目录中创建名为 hibernate.cfg.xml 的配置文件。该文件的源码如下：

```

<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
    PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <property name="hibernate.connection.driver_class">oracle.jdbc.
driver.OracleDriver</property>

        <property name="hibernate.connection.url">jdbc:oracle:thin:
@192.168.1.97:1521:test</property>

        <property name="hibernate.connection.username">system</property>
        <property name="hibernate.connection.password">abc123</property>
        <property name="hibernate.connection.pool_size">100</property>
        <property name="show_sql">>false</property>
        <property name="dialect">org.hibernate.dialect.OracleDialect </property>

        <!-- Mapping files -->
        <mapping resource="people.hbm.xml"/>

    </session-factory>

</hibernate-configuration>

```

【代码说明】该文件是一个 XML 文件；该文件的根元素是<hibernate-configuration>；<session-factory>用于配置会话工厂，会话工厂将提供创建会话的基本配置；<property name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDriver</property> 指 定



Hibernate 连接所使用的数据库驱动，不同的数据库所使用的驱动也不相同，`oracle.jdbc.driver.OracleDriver` 实际对应了 `hibernate.jar` 文件中的 `OracleDriver.class` 文件，当然不同的数据库应该使用不同的数据库驱动类；`<property name="hibernate.connection.url">jdbc:oracle:thin:@192.168.1.97:1521:test</property>` 指定了数据库连接字符串，该字符串与使用 JDBC 连接完全一致，这也印证了 Oracle 在底层仍然使用了 JDBC 连接；`<property name="hibernate.connection.username">system</property>` 指定了登录数据库所使用的用户名；`<property name="hibernate.connection.password">abc123 </property>` 指定了登录数据库所用的用户名对应的密码。`<property name="hibernate.connection.pool_size">100</property>` 指定了 Hibernate 连接池的大小；`<property name="show_sql">>false</property>` 指定了是否是在控制台上打印 SQL 语句，Hibernate 虽然将对数据库的操作转化为对象的操作，但是本质上仍然是针对数据库的操作，`false` 指定不打印实际执行的 SQL 语句；`<property name="dialect"> org.hibernate.dialect.OracleDialect</property>` 指定当前数据库所使用的方言，`org.hibernate.dialect.OracleDialect` 也是 `hibernate.jar` 中的 class 文件。`<mapping resource="people.hbm.xml"/>` 则指定了需要对哪些文件进行映射，`people.hbm.xml` 正是在步骤 3 中配置的文件。

⑥ 新建名为 `com.example.util` 的包，并在该包下创建名为 `DbUtil` 的 Java 类。该类的源码如下：

```
package com.example.util;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class DbUtil {

    private static final SessionFactory sessionFactory;
    public static final ThreadLocal session = new ThreadLocal();

    static {
        try {
            sessionFactory = new Configuration().configure(). BuildSession
Factory();
        } catch (Throwable ex) {
            ex.printStackTrace();
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static Session currentSession() throws HibernateException {
        Session s = (Session) session.get();
        if (s == null || !s.isOpen()) {
            s = sessionFactory.openSession();
            session.set(s);
        }
        return s;
    }

    public static void closeSession() throws HibernateException {
        Session s = (Session) session.get();
        session.set(null);
        if (s != null)
            s.close();
    }
}
```



```
    public SessionFactory getSessionFactory() {  
        return sessionFactory;  
    }  
}
```

【代码说明】该类使用了 `ThreadLocal` 类存储当前线程的 `Session` 对象；`new Configuration().configure().buildSessionFactory()` 通过默认配置文件 `hibernate.cfg.xml` 创建 `SessionFactory`；`currentSession()` 方法用于获取当前线程中的 `Session` 对象，这里的 `Session` 是 `Hibernate` 的会话；`closeSession()` 用于关闭当前线程中的 `Session` 对象。

22.2.3 利用 Hibernate 查询数据

在 `Hibernate` 配置成功之后，可以创建测试类来执行查询数据库。在项目目录下的 `src` 子目录下创建名为 `com.example.test` 的包，并在其下创建名为 `TestQuery` 的 `Java` 类，其源码如下：

```
package com.example.test;  
  
import java.util.List;  
  
import org.hibernate.HibernateException;  
import org.hibernate.Query;  
import org.hibernate.Session;  
import org.hibernate.Transaction;  
  
import com.example.bean.People;  
import com.example.util.DbUtil;  
  
public class TestQuery {  
  
    public static void main(String[] args) {  
        People people = null;  
  
        Session session = DbUtil.currentSession();  
  
        Transaction tx = null;  
        try {  
            tx = session.beginTransaction();  
            Query query = session.createQuery("from People ");  
            List list = query.list();  
  
            for (int i=0; i<list.size(); i++) {  
                people = (People) list.get(i);  
                System.out.println(people.getName());  
            }  
            tx.commit();  
        } catch (HibernateException e) {  
            if (tx != null)  
                tx.rollback();  
            throw e;  
        }  
        session.close();  
    }  
}
```

【代码说明】`Session session = dbUtil.currentSession()` 用于获取 `Hibernate` 的会话；`tx = session.beginTransaction()` 用于新事务的开始，这里的事务对应于数据库中的事务；`Query query = session.createQuery("from People ")` 用于创建新的查询，这里的查询是以对象形式的出现的，查询具体功能为获取所有 `People` 对象；`List list = query.list()` 将查询获得的对象存储到 `list` 对象中；

for (int i=0; i<list.size(); i++)用于循环处理 list 中的对象; people = (People) list.get(i)用于获得 list 中的对象, 并强制转换为 People 类型; System.out.println(people.getName())用于输出人员的姓名信息; session.close()用于关闭 Hibernate 会话。

在 Eclipse 运行该代码, 控制台的输出结果如图 22-10 所示。

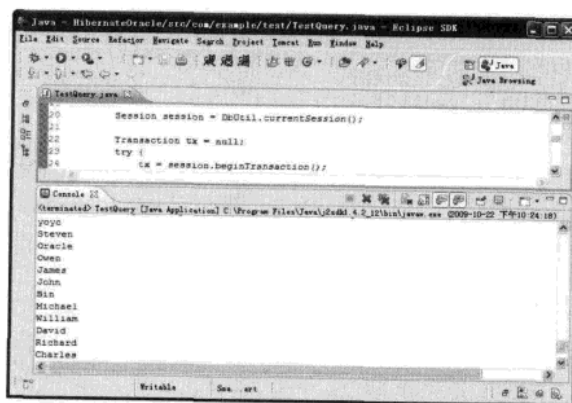


图 22-10 利用 Hibernate 查询表 People 中的所有数据

这里需要注意的是, Query 的作用对象为类 People, 而不是表 people。

22.2.4 利用 Hibernate 更新数据

使用 Hibernate 更新数据, 首先应该获得要更新的对象, 并修改对象的属性。在对象变更之后, 再将对象的变更反映到数据库中。在包 com.example.test 中创建名为 TestUpdate 的 Java 类, 源码如下:

```
package com.example.test;

import java.util.List;

import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;

import com.example.bean.People;
import com.example.util.DbUtil;

public class TestUpdate {

    public static void main(String[] args) {
        People people = null;

        Session session = DbUtil.currentSession();

        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            Query query = session.createQuery("from People where id=:id ");
            query.setParameter("id", new Integer(12));

            List list = query.list();
            if (list.size() > 0) {
                people = (People) list.get(0);
            }
        } catch (HibernateException e) {
            e.printStackTrace();
        } finally {
            tx.commit();
            session.close();
        }
    }
}
```



```
        System.out.println("ID: 12, 原名: " + people.getName());
    }

    people.setName("Jordan");
    session.update(people);

    query = session.createQuery("from People where id=:id ");
    query.setParameter("id", new Integer(12));
    list = query.list();
    if (list.size() > 0) {
        people = (People) list.get(0);
        System.out.println("ID: 12, 现名: " + people.getName());
    }

    tx.commit();
} catch (HibernateException e) {
    if (tx != null)
        tx.rollback();
    throw e;
}
session.close();
}
```

【代码说明】Query `query = session.createQuery("from People where id=:id ")` 用于创建一个 Hibernate 查询, 在该查询中使用了参数 `id`, 其形式为 `:id`; `query.setParameter("id", new Integer(12))` 向查询传递参数 `id`, 该参数的值为 12; `people.setName("Jordan")` 将查询获得的对象 `people` 的 `name` 属性值设置为 “Jordan”; `session.update(people)` 重新保存对象 `people`, 以将对象的修改反映到数据库中。

在 Eclipse 中运行该 Java 类, 可以获得修改之前和修改之后, ID 为 12 的人员姓名, 如图 22-11 所示。

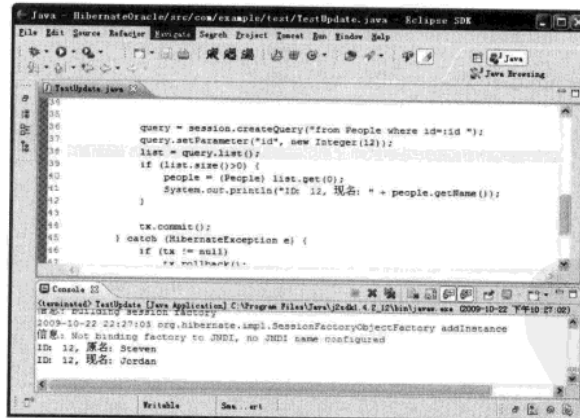


图 22-11 利用 Hibernate 修改数据

22.2.5 利用 Hibernate 插入数据

同样, 可以利用 Hibernate 向数据库中插入数据。向数据库中插入数据, 需要首先创建对象, 然后为该对象组装数据, 最后利用 Hibernate 会话的 `save()` 方法来保存对象。在



com.example.test 包下创建名为 TestInsert 的 Java 类，源码如下：

```
package com.example.test;

import java.util.List;

import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;

import com.example.bean.People;
import com.example.util.DbUtil;

public class TestInsert {

    public static void main(String[] args) {
        People people = null;

        Session session = DbUtil.currentSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();

            people = new People();
            people.setName("Owen");
            people.setStatus("ACT");

            session.save(people);

            Query query = session.createQuery("from People order by id");
            List list = query.list();
            if (list.size() > 0) {
                people = (People) list.get(list.size() - 1);
                System.out.println("新增人员: ID = " + people.getId() + "; NAME
= " + people.getName());
            }

            tx.commit();
        } catch (HibernateException e) {
            if (tx != null)
                tx.rollback();
            throw e;
        }
        session.close();
    }
}
```

【代码说明】people = new People()用于创建新的 People 对象；people.setName("Owen")和 people.setStatus("ACT")用于为 people 对象的 name 属性和 status 属性赋值；session.save(people)用于将新建对象 people 存储到数据库中；people = (People) list.get(list.size() - 1)用于获得数据表中最后一个人对象；System.out.println("新增人员: ID = " + people.getId() + "; NAME = " + people.getName())用于打印获得的人员信息。

需要注意的是，在创建 people 对象之后，不必设置 id 属性。这是因为在配置文件中 id 属性的生成规则为 increment（即自增）。该规则会首先获得数据表中最大的 id，并自增 1，作为新建对象的 id。

在 Eclipse 中执行该 Java 类，运行结果如图 22-12 所示。

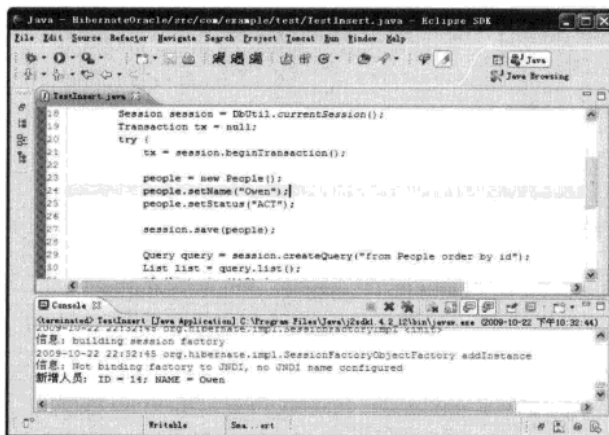


图 22-12 利用 Hibernate 插入数据



22.3 本章小结

本章讲述了如何通过 JDBC 和 Hibernate 两种不同方式操作数据库。利用 JDBC 进行操作，简单易学，但是代码可重用性不高；利用 Hibernate 则可以充分利用面向对象编程的种种优势，从而开发出层次更加明晰，可重用性更高的代码。需要注意的是，Hibernate 的底层仍然是通过 JDBC 处理数据库的，因此，相对于单纯使用 JDBC 的方式，Hibernate 方式在速度上并不占据优势。



22.4 习题

1. 什么是 JDBC?
2. JDBC 连接 Oracle 数据库的字符串及字符串各参数的意义是什么?
3. Hibernate 的主要设计意图是什么?
4. 简述利用 Hibernate 插入数据的主要操作步骤。



第 23 章 Oracle 在 C#开发中的应用

Oracle 不仅能够应用在 Java 开发中,而且在其他开发语言中应用也比较广泛。本章将讲述 Oracle 在 C#开发中的应用。本章的主要内容包括:

- 在 C#中连接 Oracle 数据库;
- 在 C#中操作 Oracle 数据库;
- 在 C#中使用 Oracle 数据库事务。

通过本章的学习,读者可以了解 Oracle 在 C#开发中的应用,并掌握与 Java 环境下开发的异同。



23.1 在 C#中连接 Oracle 数据库

在 C#中连接 Oracle 数据库,与在 Java 中利用 JDBC 连接 Oracle 基本相同。都是利用连接字符串来实现的。在具体实现之前,应该在应用程序的服务器上安装 Oracle 客户端。

【范例 23-1】演示在 C#中连接 Oracle 数据库的代码片段。

```
using System;
using System.Text;
using System.Data;
using System.Data.OracleClient;

namespace WebApp
{
    public class People
    {
        /// <summary>
        /// 获得数据库连接
        /// </summary>
        /// <returns></returns>
        public OracleConnection getConnection()
        {
            //数据库连接字符串
            string connectionString = "Server=192.168.0.178;Data Source= test;User
Id=test;Password=abc123;";

            //创建数据库连接
            using (OracleConnection connection = new OracleConnection
(connectionString));

            return connection;
        }
    }
}
```

【代码说明】string connectionString = "Server=127.0.0.1;Data Source=test;User Id=test;Password=abc123;"用于指定连接字符串;其中 Server 指定数据库所在主机的 IP 地址;Data



Source 指定数据库名称; User Id 用于指定登录数据库的用户名; Password 则指定用户名所对应的密码; using (OracleConnection connection = new OracleConnection(connectionString))根据连接字符串来创建 Oracle 数据库连接。



23.2 在 C#中操作 Oracle 数据库

一旦在 C#中成功创建 Oracle 数据库连接,即可利用该连接向 Oracle 服务器端发送数据库命令,以实现数据库的操作。

【范例 23-2】演示如何在 C#中操作 Oracle 数据库。

```
using System;
using System.Text;
using System.Data;
using System.Data.OracleClient;

namespace WebApp
{
    public class People
    {
        /// <summary>
        /// 读取数据
        /// </summary>
        /// <returns></returns>
        public string ReadData()
        {
            StringBuilder texts = new StringBuilder();

            //数据库查询字符串
            string queryString = "SELECT id,name,status FROM people ";

            //数据库连接字符串
            string connectionString = "Server=127.0.0.1;DataSource=test;UserId=test;
Password=abc123;";

            //创建数据库连接
            using (OracleConnection connection = new OracleConnection
(connectionString))
            {
                OracleCommand command = new OracleCommand(queryString, connection);
                //打开数据库连接
                connection.Open();
                //读取数据
                OracleDataReader reader = command.ExecuteReader();
                try
                {
                    while (reader.Read())
                    {
                        texts.Append(reader.GetInt32(0).ToString());
                        texts.Append(", ");
                        texts.Append(reader.GetString(1));
                        texts.Append(", ");
                        texts.Append(reader.GetString(2));
                        texts.Append("<br>");
                    }
                }
                finally
            }
        }
    }
}
```

```

        {
            reader.Close();
        }
    }

    return texts.ToString();
}

/// <summary>
/// 添加数据
/// </summary>
/// <returns></returns>
public void AddData(int id, string name, string status)
{
    //插入数据字符串
    string insertString = string.Format("INSERT TABLE people (id, name, status)
VALUES ({0}, '{1}', '{2}')" , id, name, status);

    //数据库连接字符串
    string connectionString = "Server=127.0.0.1;Data Source=test; User Id=test;
Password=abc123;";

    //创建数据库连接
    using (OracleConnection connection = new OracleConnection
(connectionString))
    {
        OracleCommand command = new OracleCommand(insertString, connection);
        //打开数据库连接
        connection.Open();
        //执行添加命令
        command.ExecuteNonQuery();
    }

    /// <summary>
    /// 更新数据
    /// </summary>
    /// <returns></returns>
    public void UpdateData(int id, string name, string status)
    {
        //更新数据字符串
        string updateString = string.Format("UPDATE people SET name = '{0}', status
= '{1}' WHERE id = {2}", name, status, id);

        //数据库连接字符串
        string connectionString = "Server=127.0.0.1;Data Source=test; User
Id=test;Password=abc123;";

        //创建数据库连接
        using (OracleConnection connection = new OracleConnection
(connectionString))
        {
            OracleCommand command = new OracleCommand(updateString, connection);
            //打开数据库连接
            connection.Open();
            //执行添加命令
            command.ExecuteNonQuery();
        }
    }
}

```



```

    /// <summary>
    /// 删除数据
    /// </summary>
    /// <returns></returns>
    public void DeleteData(int id)
    {
        //删除数据字符串
        string deleteString = string.Format("DELETE FROM people WHERE id = {0}",
id);

        //数据库连接字符串
        string connectionString = "Server=127.0.0.1;Data Source=test; User Id=test;
Password=abc123;";

        //创建数据库连接
        using (OracleConnection connection = new OracleConnection
(connectionString))
        {
            OracleCommand command = new OracleCommand(deleteString, connection);
            //打开数据库连接
            connection.Open();
            //执行添加命令
            command.ExecuteNonQuery();
        }
    }
}

```

【代码说明】在 C# 中提供了名为 `OracleCommand` 的类来保存 Oracle 命令的信息；其中的信息包含了两方面的内容，一为连接信息，二为要执行的命令。这里需要注意的是，对于查询操作，需要使用 `OracleDataReader` 将查询获得的数据集合存储起来，而对于其他 DML 操作，直接使用 `OracleCommand` 的 `ExecuteNonQuery()` 方法将操作指令反映到数据库。



23.3 在 C# 中使用 Oracle 数据库事务

在 C# 中，同样可以使用 Oracle 数据库事务。一旦使用了事务，那么在事务开始和结束之间所执行的所有 C# 代码块对 Oracle 数据库所产生的影响将遵循数据库事务规则——要么全部执行成功，要么全部回滚。

【范例 23-3】演示如何在 C# 中使用 Oracle 数据库事务。

```

using System;
using System.Text;
using System.Data;
using System.Data.OracleClient;

namespace WebApp
{
    public class People
    {
        /// <summary>
        /// 事务处理
        /// </summary>
        /// <returns></returns>
        public void TestTransaction()
        {
            //数据库连接字符串
            string connectionString = "Server=127.0.0.1;Data Source=test; User

```



```

Id=test;Password=abc123;";

        //创建数据库连接
        OracleConnection connection = new OracleConnection(connection String);

        //打开数据库连接
        connection.Open();

        //创建并同时启动一个事务
        OracleTransaction myOracleTransaction = connection.Begin Transaction();

        //创建一个 Oracle 命令对象
        OracleCommand myOracleCommand = myOracleConnection.CreateCommand();

        //为 myOracleCommand 设置命令文本
        myOracleCommand.CommandText = "INSERT INTO PEOPLE VALUES(16, 'Allen',
'ACT')";

        //执行 myOracleCommand 中存储的 SQL 语句
        myOracleCommand.ExecuteNonQuery();

        //为 myOracleCommand 设置另外一个命令
        myOracleCommand.CommandText = "UPDATE PEOPLE SET STATUS = 'ACT'";

        //执行 myOracleCommand 中存储的 SQL 语句
        myOracleCommand.ExecuteNonQuery();

        //提交事务
        myOracleTransaction.Commit();

        //关闭数据库连接
        connection.close();
    }
}

```

【代码说明】在该示例代码中，首先创建了一个数据库连接，并利用数据库连接启动了一个事务。那么，利用该数据库连接所执行的动作，都将自动处于该事务之下。最后，利用 myOracleTransaction.Commit() 来提交事务。

该范例所执行的动作，相当于执行了如下 SQL 语句：

```

SQL> insert into people values(16, 'Allen', 'ACT');

1 row inserted

SQL> update people set status = 'ACT';

15 rows updated

SQL> commit;

Commit complete

```

当然，也可以利用 OracleTransaction.Rollback() 来回滚一个事务，如以下代码所示。

```

using System;
using System.Text;
using System.Data;
using System.Data.OracleClient;

namespace WebApp
{

```



```
public class People
{
    /// <summary>
    /// 事务处理
    /// </summary>
    /// <returns></returns>
    public void TestTransaction()
    {
        //数据库连接字符串
        string connectionString = "Server=127.0.0.1;Data Source=test; User Id=test;
        Password=abc123;";

        //创建数据库连接
        OracleConnection connection = new OracleConnection (connectionString);

        //打开数据库连接
        connection.Open();

        //创建并同时启动一个事务
        OracleTransaction myOracleTransaction = connection.Begin Transaction();

        try {
            //创建一个 Oracle 命令对象
            OracleCommand myOracleCommand = myOracleConnection.CreateCommand();

            //为 myOracleCommand 设置命令文本
            myOracleCommand.CommandText = "INSERT INTO PEOPLE VALUES (16, 'Allen',
            'ACT')";

            //执行 myOracleCommand 中存储的 SQL 语句
            myOracleCommand.ExecuteNonQuery();

            //为 myOracleCommand 设置另外一个命令
            myOracleCommand.CommandText = "UPDATE PEOPLE SET STATUS = 'ACT'";

            //执行 myOracleCommand 中存储的 SQL 语句
            myOracleCommand.ExecuteNonQuery();

            //提交事务
            myOracleTransaction.Commit();

            //捕获异常
        } catch (Exception ex) {

            //回滚事务
            myOracleTransaction.Rollback();

        } finally {

            //关闭数据库连接
            connection.close();
        }
    }
}
```

【代码说明】本例使用了 try-catch 语句来处理数据库操作语句；一旦捕获异常，那么通过 myOracleTransaction.Rollback() 来回滚所有操作；finally 语句用于保证数据库连接一定被关闭。



23.4 本章小结

本章简单讲述了如何在 C# 中连接和操作 Oracle 数据库。在 C# 中连接和操作数据库与在 Java# 中利用 JDBC 操作数据库具有相同的工作原理和开发模式。本章给出了示例代码，读者可以参考示例代码，并结合 JDBC 模式进行学习。



23.5 习题

1. 简述 C# 中连接 Oracle 数据库字符串中各参数的意义。
2. 简述 C# 中操作 Oracle 数据库的一般步骤。
3. 简述 C# 中如何使用 Oracle 数据库事务。
4. 举例说明 C# 中对 Oracle 数据库进行事务处理的一般做法。





电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview®
WWW.BROADVIEW.COM.CN

Csdn.NET

技术凝聚实力 专业创新出版

博文视点 (www.broadview.com.cn) 资讯有限公司是电子工业出版社、CSDN.NET、《程序员》杂志联合打造的专业出版平台, 博文视点致力于——IT专业图书出版, 为IT专业人士提供真正专业、经典的好书。

请访问 www.dearbook.com.cn (第二书店) 购买优惠价格的博文视点经典图书。

请访问 www.broadview.com.cn (博文视点的服务平台) 了解更多更全面的出版信息; 您的投稿信息在这里将会得到迅速的反馈。

博文视点精品汇聚



加密与解密 (第三版)

段钢 编著
ISBN 978-7-121-06644-3
定价: 69.00元

畅销书升级版, 出版一月销售10000册。
看雪软件安全学院众多高手, 合力历时4年精心打造。



疯狂Java讲义

新东方IT培训广州中心
软件教学总监 李刚 编著
ISBN 978-7-121-06646-7
定价: 99.00元 (含光盘1张)

用案例驱动, 将知识点融入实际项目的开发。
代码注释非常详细, 几乎每两行代码就有一行注释。



Windows驱动开发技术详解

张帆 等编著
ISBN 978-7-121-06846-1
定价: 65.00元 (含光盘1张)

原创经典, 威盛一线工程师倾力打造。
深入驱动核心, 剖析操作系统底层运行机制。



Struts 2权威指南

李刚 编著
ISBN 978-7-121-04853-1
定价: 79.00元 (含光盘1张)

可以作为Struts 2框架的权威手册。
通过实例演示Struts 2框架的用法。



你必须知道的.NET

王涛 著
ISBN 978-7-121-05891-2
定价: 69.80元

来自于微软MVP的最新技术心得和感悟。
将技术问题以生动易懂的语言展开, 层层深入, 以例说理。



Oracle数据库精讲与疑难解析

赵振平 编著
ISBN 978-7-121-06189-9
定价: 128.00元

754个故障重现, 件件源自工作的经验教训。
为专业人士提供的速查手册, 遇到故障不求人。



SOA原理·方法·实践

IBM资深架构师毛新生 主编
ISBN 978-7-121-04264-5
定价: 49.8元

SOA技术巅峰之作!
IBM中国开发中心技术经典呈现!



VC++深入详解

孙鑫 编著
ISBN 7-121-02530-2
定价: 89.00元 (含光盘1张)

IT培训专家孙鑫经典畅销力作!

博文视点资讯有限公司

电话: (010) 51260888 传真: (010) 51260888-802

E-mail: market@broadview.com.cn (市场)

editor@broadview.com.cn jsj@phei.com.cn (投稿)

通信地址: 北京市万寿路173信箱 北京博文视点资讯有限公司

邮编: 100036

电子工业出版社发行部

发行部: (010) 88254055

门市部: (010) 68279077 68211478

传真: (010) 88254050 88254060

通信地址: 北京市万寿路173信箱

邮编: 100036

博文视点·IT出版旗舰品牌



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

Broadview[®]
www.broadview.com.cn

《21 天学通 Oracle》读者交流区

尊敬的读者：

感谢您选择我们出版的图书，您的支持与信任是我们持续上升的动力。为了使您能通过本书更透彻地了解相关领域，更深入的学习相关技术，我们将特别为您提供一系列后续的服务，包括：

1. 提供本书的修订和升级内容、相关配套资料；
2. 本书作者的见面会信息或网络视频的沟通活动；
3. 相关领域的培训优惠等。

请您抽出宝贵的时间将您的个人信息和需求反馈给我们，以便我们及时与您取得联系。

您可以任意选择以下三种方式与我们联系，我们都将记录和保存您的信息，并给您提供不定期的信息反馈。

1. 短信

您只需编写如下短信：B10626+您的需求+您的建议

发送到1066 6666 789（本服务免费，短信资费按照相应电信运营商正常标准收取，无其他信息收费）

为保证我们对您的服务质量，如果您在发送短信24小时后，尚未收到我们的回复信息，请直接拨打电话（010）88254369。

2. 电子邮件

您可以发邮件至jsj@phei.com.cn或editor@broadview.com.cn。

3. 信件

您可以写信至如下地址：北京万寿路173信箱博文视点，邮编：100036。

如果您选择第2种或第3种方式，您还可以告诉我们更多有关您个人的情况，及您对本书的意见、评论等，内容可以包括：

- （1）您的姓名、职业、您关注的领域、您的电话、E-mail地址或通信地址；
- （2）您了解新书信息的途径、影响您购买图书的因素；
- （3）您对本书的意见、您读过的同领域的图书、您还希望增加的图书、您希望参加的培训等。

如果您在后期想退出读者俱乐部，停止接收后续资讯，只需发送“B10626+退订”至10666666789即可，或者编写邮件“B10626+退订+手机号码+需退订的邮箱地址”发送至邮箱：market@broadview.com.cn 亦可取消该项服务。

同时，我们非常欢迎您为本书撰写书评，将您的切身感受变成文字与广大书友共享。我们将挑选特别优秀的作品转载在我们的网站（www.broadview.com.cn）上，或推荐至CSDN.NET等专业网站上发表，被发表的书评的作者将获得价值50元的博文视点图书奖励。

我们期待您的消息！

博文视点愿与所有爱书的人一起，共同学习，共同进步！

通信地址：北京万寿路 173 信箱 博文视点（100036） 电话：010-51260888

E-mail: jsj@phei.com.cn, editor@broadview.com.cn

www.phei.com.cn
www.broadview.com.cn