

目录

目录.....	1
译者的话:	14
第一周概貌	16
从这里开始	16
第一天: SQL 简介	17
SQL 简史	17
数据库简史	17
设计数据库的结构.....	21
SQL 总览	23
流行的 SQL 开发工具.....	24
SQL 在编程中的应用.....	27
第二天: 查询——SELECT 语句的使用	30
目标:	30
背景.....	30
一般的语法规则	30
你的第一个查询	33
总结:	37
问与答:	38
校练场	38
练习:	39
第三天: 表达式、条件语句与运算	40
第四天: 函数: 对数据的进一步处理.....	60
目标:	60
汇总函数.....	60
COUNT	61
SUM	61
AVG.....	63
MAX.....	63
MIN.....	64
VARIANCE.....	65
STDDEV	66

日期/时间函数.....	66
ADD_MONTHS.....	67
LAST_DAY.....	68
MONTHS_BETWEEN.....	69
NEW_TIME.....	70
NEXT_DAY.....	71
SYSDATE.....	72
数学函数.....	72
ABS.....	73
CEIL 和 FLOOR.....	73
COS、COSH、SIN、SINH、TAN、TANH.....	73
EXP.....	75
LN and LOG.....	75
MOD.....	76
POWER.....	77
SIGN.....	77
SQRT.....	78
字符函数.....	79
CHR.....	79
CONCAT.....	79
INITCAP.....	80
LOWER 和 UPPER.....	81
LPAD 与 RPAD.....	82
LTRIM 与 RTRIM.....	83
REPLACE.....	84
SUBSTR.....	85
TRANSLATE.....	88
INSTR.....	88
LENGTH.....	89
转换函数.....	89
TO_CHAR.....	90
TO_NUMBER.....	91
其它函数.....	91
GREATEST 与 LEAST.....	91
USER.....	92
总结.....	92
问与答.....	93
校练场.....	93
练习.....	94
第五天：SQL 中的子句.....	95

目标:	95
WHERE 子句	96
STARTING WITH 子句.....	98
ORDER BY 子句	99
GROUP BY 子句	104
HAVING 子句.....	109
子句的综合应用	112
总结.....	117
问与答	117
校练场	117
练习.....	118
第六天: 表的联合.....	119
介绍.....	119
在一个 SELECT 语句中使用多个表.....	119
正确地找到列.....	123
等值联合.....	124
不等值联合	129
外部联合与内部联合	130
表的自我联合.....	132
总结.....	134
问与答	134
校练场	134
练习.....	135
第七天: 子查询: 内嵌的 SQL 子句.....	136
目标:	136
建立一个子查询	136
在子查询中使用汇总函数	140
子查询的嵌套.....	141
相关子查询	144
EXISTS, ANY, ALL 的使用	147

总结.....	151
问与答	151
校练场	152
练习:	153
第一周回顾	154
预览.....	154
第二周概貌	155
这一周都讲些什么.....	155
第八天: 操作数据.....	156
目标.....	156
数据操作语句.....	156
插入语句.....	157
INSERT VALUES 语句.....	157
INSERT SELECT 语句	161
UPDATE 语句	163
DELETE 语句.....	166
从外部数据源中导入和导出数据.....	169
Microsoft Access	170
Microsoft and Sybase SQL Server	171
Personal Oracle7.....	171
总结.....	172
问与答	172
校练场	173
练习.....	173
第九天: 创建和操作表.....	174
目标:	174
CREATE DATABASE 语句.....	174
建立数据库时的选项	175
设计数据库	176
建立数据字典.....	176
建立关键字段.....	177
CREATE TABLE 语句.....	178
表名.....	179

FIRST NAME	179
空值属性	180
唯一属性	181
表的存储与尺寸的调整	183
用一个已经存在的表来建表	184
ALTER TABLE 语句	185
DROP TABLE 语句	186
DROP DATABASE 语句	187
总结	188
问与答	188
校练场	189
练习	190
第 10 天 创建视图和索引	191
目标	191
使用视图	192
列的重命名	196
SQL 对视图的处理过程	197
在 SELECT 语句使用约束	201
在视图中修改数据	201
在视图中修改数据的几个问题	203
通用应用程序的视图	204
删除视图语句	207
使用索引	207
什么是索引?	207
使用索引的技巧	212
对更多的字段进行索引	212
在创建索引时使用 UNIQUE 关键字	214
索引与归并	216
群集（簇）的使用	217
总结	218
问与答:	219
校练场	219
练习:	220
第 11 天: 事务处理控制	221
目标:	221
事务控制	221

银行应用程序.....	222
开始事务处理.....	223
结束事务处理.....	225
取消事务处理.....	228
在事务中使用保存点	231
总结.....	234
问与答	234
校练场	235
练习.....	235
第 12 天：数据库安全	236
前提：数据库管理员	236
流行的数据库产品与安全	237
如何让一个数据库变得安全.....	237
Personal Oracle7 与安全	238
创建用户.....	238
创建角色.....	240
用户权限.....	242
为安全的目的而使用视图	247
总结.....	251
问与答	252
校练场	252
练习.....	253
第 13 天 高级 SQL.....	254
目标：	254
临时表	254
Title	257
游标.....	259
创建游标.....	260
打开游标.....	260
使用游标来进行翻阅	261
测试游标的状态	262
关闭游标.....	263
游标的适用范围	264
创建和使用存贮过程	265
在存贮过程中使用参数.....	267

删除一个存贮过程.....	269
存贮过程的嵌套.....	270
设计和使用触发机制.....	272
触发机制与事务处理.....	273
使用触发机制时的限制.....	275
触发机制的嵌套.....	275
在选择语句中使用更新和删除.....	275
在执行前测试选择语句.....	276
嵌入型 SQL.....	277
静态 SQL 与动态 SQL.....	277
使用 SQL 来编程.....	279
总结.....	280
问与答.....	280
校练场.....	280
练习.....	281
第 14 天：动态使用 SQL.....	282
目标.....	282
快速入门.....	282
ODBC.....	282
Personal Oracle 7.....	283
InterBase SQL (ISQL).....	283
Visual C++.....	284
Delphi.....	284
设置.....	284
创建数据库.....	285
使用 MS QUERY 来完成链接.....	290
将 VISUAL C++与 SQL 结合使用.....	292
将 DELPHI 与 SQL 结合使用.....	296
总结.....	302
问与答.....	303
校练场.....	303
练习.....	303
第二周回顾.....	304
第三周概貌.....	305

应用你对 SQL 的知识.....	305
第 15 天：对 SQL 语句优化以提高其性能.....	306
目标.....	306
让你的 SQL 语句更易读.....	307
全表扫描.....	308
加入一个新的索引.....	309
在查询中各个元素的布局.....	309
过程.....	311
避免使用 OR.....	311
OLAP 与 OLTP 的比较.....	313
OLTP 的调试.....	313
OLAP 的调试.....	314
批量载入与事务处理进程.....	314
删除索引以优化数据的载入.....	316
经常使用 COMMIT 来让 DBA 走开.....	316
在动态环境中重新生成表和索引.....	317
数据库的调整.....	319
性能的障碍.....	322
内置的调整工具.....	323
总结.....	323
问与答.....	324
校练场.....	324
练习.....	324
第 16 天：用视图从数据字典中获得信息.....	326
目标.....	326
数据字典简介.....	326
用户的数据字典.....	327
数据字典中的内容.....	327
Oracle 的数据字典.....	328
Sybase 的数据字典.....	328
ORACLE 数据字典的内部结构.....	328
用户视图.....	328

系统数据库管理员视图.....	336
数据库对象	339
数据库的生长.....	343
动态执行视图.....	347
总结.....	349
问与答	349
校练场	350
练习.....	350
第 17 天：使用 SQL 来生成 SQL 语句	351
目标.....	351
使用 SQL 来生成 SQL 语句的目的.....	351
几个 SQL*PLUS 命令	352
SET ECHO ON/OFF.....	353
SET FEEDBACK ON/OFF.....	353
SET HEADING ON/OFF	353
SPOOL FILENAME/OFF.....	353
START FILENAME.....	354
ED FILENAME.....	354
计算所有的表中的行数.....	354
为多个用户赋予系统权限	359
将你的表的权限赋予其它的用户.....	361
在载入数据时解除对数的约束	363
一次创建多个同义字	364
为你的表创建视图.....	368
在一个计划中清除其所有的表的内容.....	369
使用 SQL 来生成 SHELL 脚本	371
再建表和索引.....	372
总结.....	373
问与答	373
校练场	373
练习.....	374
第 18 天：PL/SQL 简介	376
目标.....	376

入门.....	376
在 PL/SQL 中的数据类型.....	377
字符串类型	377
数值数据类型.....	378
二进制数据类型	378
日期数据类型.....	378
逻辑数据类型.....	378
ROWID.....	379
PL/SQL 块的结构.....	379
注释.....	380
DECLARE 部分.....	380
变量声明.....	380
常量定义.....	381
指针定义.....	381
%TYPE 属性	382
%ROWTYPE 属性	382
%ROWCOUNT 属性.....	383
Procure 部分.....	383
BEGINEND	383
指针控制命令.....	384
条件语句.....	386
LOOPS 循环.....	387
EXCEPTION 部分	390
激活 EXCEPTION（异常）	390
异常的处理	391
将输入返回给用户.....	392
在 PL/SQL 中的事务控制.....	393
让所有的事在一起工作.....	394
示例表及数据.....	394
一个简单的 PL/SQL 语句块	395
又一个程序	398
存储过程、包和触发机制	403
总结.....	406
问与答	407
校练场	407
练习.....	407

第 19 天：TRANSACT-SQL 简介.....	408
目标.....	408
TRANSACT-SQL 概貌.....	408
对 ANSI SQL 的扩展	408
谁需要使用 TRANSACT-SQL	409
TRANSACT-SQL 的基本组件	409
数据类型.....	409
使用 TRANSACT-SQL 来访问数据库.....	411
BASEBALL 数据库	411
定义局部变量.....	414
定义全局变量.....	414
使用变量.....	415
PRINT 命令	417
流控制	417
BEGINEND 语句	418
IFELSE 语句	418
EXIST 条件.....	421
WHILE 循环.....	422
使用 WHILE 循环在表中翻阅	424
TRANSACT-SQL 中的通配符	426
使用 COMPUTE 来生成摘要报告	426
日期转换.....	427
SQL SERVER 的诊断工具 ——SET 命令.....	427
总结.....	428
问与答	428
校练场	429
练习.....	429
第 20 天：SQL*PLUS	430
目标.....	430
简介.....	430
SQL*PLUS 缓存.....	430
DESCRIBE 命令.....	435
SHOW 命令.....	436
文件命令.....	438

SAVE、GET、EDIT 命令	438
运行一个文件.....	439
查询的假脱机输出.....	440
SET 命令	442
LOGIN.SQL 文件	445
CLEAR 命令	446
将你的输出格式化.....	446
TTITLE 与 BTITLE.....	446
格式化列 (COLUMN、HEADING、FORMAT)	447
报表与分类汇总	449
BREAK ON	449
COMPUTE.....	450
在 SQL*PLUS 中使用变量	453
DEFINE	454
ACCEPT	455
NEW_VALUE.....	457
DUAL 表.....	458
DECODE 函数.....	459
日期转换.....	462
运行一系列的 SQL 文件	465
在你的 SQL 脚本中加入注释	466
高级报表.....	467
总结.....	469
问与答	469
校练场	469
练习.....	470
第 21 天：常见的 SQL 错误及解决方法	471
目标:	471
介绍.....	471
常见的错误	471
Table or View Does Not Exist.....	471
Invalid Username or Password	472
FROM Keyword Not Specified.....	473
Group Function Is Not Allowed Here	474
Invalid Column Name.....	475

Missing Keyword	475
Missing Left Parenthesis	476
Missing Right Parenthesis	477
Missing Comma.....	478
Column Ambiguously Defined	478
Not Enough Arguments for Function.....	480
Not Enough Values.....	481
Integrity Constraint Violated--Parent Key Not Found	482
Oracle Not Available	483
Inserted Value Too Large for Column.....	483
TNS:listener Could Not Resolve SID Given in Connect Descriptor	484
Insufficient Privileges During Grants.....	484
Escape Character in Your Statement--Invalid Character	485
Cannot Create Operating System File	485
Common Logical Mistakes.....	485
Using Reserved Words in Your SQL statement	486
The Use of DISTINCT When Selecting Multiple Columns.....	487
Dropping an Unqualified Table	487
The Use of Public Synonyms in a Multischema Database.....	488
The Dreaded Cartesian Product	488
Failure to Enforce File System Structure Conventions	489
Allowing Large Tables to Take Default Storage Parameters.....	489
Placing Objects in the System Tablespace.....	490
Failure to Compress Large Backup Files	491
Failure to Budget System Resources	491
Preventing Problems with Your Data.....	491
Searching for Duplicate Records in Your Database.....	491
总结:	491
校练场.....	492
练习.....	492
第三周回顾	494
附件 A: 在 SQL 中的常见术语.....	495
ALTER DATABASE.....	495
ALTER USER.....	495
BEGIN TRANSACTION	495
CLOSE CURSOR.....	495
COMMIT TRANSACTION.....	496
CREATE DATABASE.....	496
CREATE INDEX.....	496
CREATE PROCEDURE.....	496
CREATE TABLE.....	497
CREATE TRIGGER.....	497

CREATE USER.....	497
CREATE VIEW.....	497
DEALLOCATE CURSOR.....	498
DROP DATABASE.....	498
DROP INDEX.....	498
DROP PROCEDURE.....	498
DROP TABLE.....	498
DROP TRIGGER	499
DROP VIEW.....	499
EXECUTE.....	499
FETCH.....	499
FROM.....	499
GRANT.....	500
GROUP BY	500
HAVING.....	500
INTERSECT.....	500
ORDER BY	500
ROLLBACK TRANSACTION	500
REVOKE.....	500
SELECT	501
SET TRANSACTION.....	501
UNION.....	501
WHERE	501
*	501
附件 B: 在第 14 天中的 C++源代码清单	502
附件 C: 第 14 天中的 Delphi 源代码清单	521
附件 D: 参考内容.....	524
书	524
《Developing Sybase Applications》	524
《Sybase Developer's Guide》	524
《Microsoft SQL Server 6.5 Unleashed, 2E》	524
《Teach Yourself Delphi in 21 Days》	524
《Delphi Developer's Guide》	524
《Delphi Programming Unleashed》	525
《Essential Oracle 7.2》	525
《Developing Personal Oracle7 for Windows 95 Applications》	525
《Teach Yourself C++ Programming in 21 Days》	525
《Teach Yourself Tansact-SQL in 21 Days》	525
《Teach Yourself PL/SQL in 21 Days 》	525
杂志:	526
DBMS	526
Oracle Magazine.....	526

SQL 的互联网资源	526
附件 E: ACSLL 码表	527
附件 F: 问题与练习答案.....	533
第一天: SQL 简介	533
问题答案.....	533
练习答案.....	533
第二天: 查询——SELECT 语句的使用	533
问题答案.....	533
练习答案.....	534
第三天: 表达式、条件语句与运算	535
问题答案.....	535
练习答案.....	535
第四天: 函数: 对获得数据的进一步处理.....	536
问题答案.....	536
练习答案.....	537
第五天: SQL 中的子句	538
问题答案.....	538
练习答案.....	538
第六天: 表的联接.....	540
问题答案.....	540
练习答案.....	541
第 7 天: “子查询: 内嵌的 SELECT 语句”	542
问题答案.....	542
练习答案.....	544
第八天: 操作数据.....	544
问题答案.....	544
练习答案.....	546
第九天: 创建和操作表.....	546
问题答案.....	546
练习答案.....	548
第 10 天 创建视图和索引.....	549
问题答案.....	549
练习答案:	550
第 11 天: 事务处理控制.....	550
问题答案.....	550
练习答案.....	551
第 12 天: 数据库安全	552

问题答案.....	552
练习答案.....	552
第 13 天 高级 SQL.....	553
问题答案.....	553
练习答案.....	553
第 14 天：动态使用 SQL	554
问题答案.....	554
练习答案.....	554
第 15 天：对 SQL 语句优化以提高其性能.....	555
问题答案.....	555
练习答案.....	555
第 16 天：用视图从数据字典中获得信息	557
问题答案.....	557
练习答案.....	557
第 17 天：使用 SQL 来生成 SQL 语句	558
问题答案.....	558
练习答案.....	560
第 18 天：PL/SQL 简介	561
问题答案.....	561
练习答案.....	561
第 19 天：TRANSACT-SQL 简介.....	562
问题答案.....	562
练习答案.....	562
第 20 天：SQL*PLUS	563
问题答案.....	563
练习答案.....	563
第 21 天：常见的 SQL 错误及解决方法	564
问题答案.....	564
练习答案.....	565

译者的话：

大家好，我是笨猪：

由于我的能力有限，所以文章中会存在许多的不当之处，特别是，我也是一个初学者，所以我想通过这一工作来认识更多的朋友。能得到高手的指点也可以让我的学习有一个进步。这也正是我所期望的。

您如果发现文章有错误或不当的地方，请 MAIL 给我。我会对此进行及时的更正。

本人对书中的内容不作任何担保，对使用本书中所述内容进行操作所引起的事故亦不承担任何责任。

这本书我想它可以称为 FREEBOOK。也就是说你可以自由地使用它而不必付出任何费用。但出于对书中内容负责的态度，所以当你认为需要对其进行修改时请先通知我，由我自己来进行修改。

本书目前的版本为 1.0 版，随着日后的修改它的版本会升高，直至最终定稿。

在此，我特别感谢所有为本书提供了发布空间的网站。感谢他们对互联网事业的支持，中华网电脑书库首先为本人敞开了大门，在此特别感谢。

需要说明的是，本书的工作到此并没有结束，虽然它的内容已经完整了。但是正如我所说，还并不完善。所以我希望得到大家的帮助和指点。这也可以看作是你对我的工作的支持吧。

这半年来，我的妻子——边文兰对我的工作非常的支持。没有她的鼓励，你也许就不会看到眼前的东西了!!

网友 ARROWTIME 对本书的部分内容作了修订，对此我非常感谢。

另：所有对本书提出了有价值的修改意见的网友，我均会将你的名字写入本书中，因为这本书中也有您的汗水。

我的信箱是：wyhsillypig@163.com，是一个 8M 的信箱。

王永宏

2001 年 7 月 5 日

第一周概貌

从这里开始

在本周我们将向大家介绍 SQL 的发展历程及其前景，并来学习第一个 SQL 语句——SELECT 语句。它使我们能够用自己的方法来从数据库中检索到自己想要的数据库。同时，在第一周我们也将学习 SQL 的函数、联合查询及子查询（嵌于查询中的查询）。并举出多个例子以帮助理解它们，这些例子是适用于 Oracle7，Sybase SQL Server，Microsoft Access，Microsoft Query，我们会用高亮显示指出它们的相似之处以及不同点，读者们会觉得这些例子更具有适用性和趣味性。

第一天：SQL 简介

SQL 简史

SQL 的诞生于 IBM 公司在加利福尼亚 San Jose 的试验室中，在七十年代 SQL 由这里开发出来。最初它们被称为结构化查询语言 (Structured Query Language)，并常常简称为 `sequeL`。开始时它们是为 IBM 公司的 DB2 系列数据管理系统 (RDBMS —— 关系型数据库管理系统) 而开发的，您在今天仍可以买到在不同平台下运行的该系统，事实上，是 SQL 造就了 RDBMS，它是一种非过程语言，与第三代过程语言如 C 和 COBOL 产生于同一时代。

注：非过程性语言的意思就是指与具体过程无关，举例来说：SQL 描述了对数据进行检索，插入，删除，但它并不说明如何进行这样的操作。

这种特性将 RDBMS 从 DBMS 中区别开来，RDBMS 提供了一整套的针对数据库的语言。而且对于大多数的 RDBMS 来说，这一整套的数据语言就是 SQL。这里一整套的意思就是对数据和处理操作语言是一些过程的集合。

有两个标准化组织，美国国家标准协会 (ANSI) 和国际标准组织 (ISO) 正致力于 SQL 在工业领域的标准化应用工作。本书使用的标准为 ANSI-92。尽管该标准要求所有的数据库设计者应遵守这一标准，然而所有的数据库系统所用的 SQL 均与 ANSI-92 存在一定的差异。此外，大多数数据库系统对 SQL 进行了有针对性的扩展使它们成为了过程型语言。在本书中我们对不同的 RDBMS 系统给出了它们的 SQL 语言例句，希望你能从中发现它们的共性。(我们将要讨论的过程型 SQL 有 PL/SQL 和 Transact-SQL，它们将在第 18 天和第 19 天提到)

数据库简史

对数据库的发展历程有一个简要的了解可以使您更清楚如何使用 SQL 来工作。数据库系统在商业领域应用极为广泛，大到航空机票售票系统，小到孩子们的棒球卡管理系统，数据库将按照我们的意愿来存储和处理这些数据，直到最近几年以前，大型的数据库系统

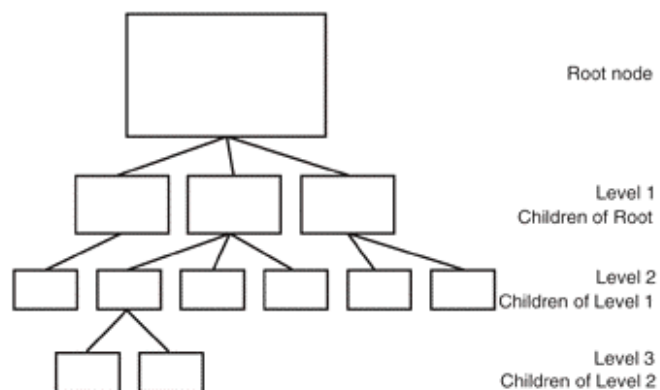
仍只能在大型机上运行，而大型机的运行，维护，使用费用均是非常昂贵的。然而在今天，工作站的能力强大到可以让编程人员以极快的速度和极低的价格来设计和发布软件。

Dr. Codd's 对关系型数据库系统的十二条规则

关系型数据库是最为流行的数据存储模式，它产生于一个名称为《A Relational Model of Data for Large Shared Data Banks》的论文中，SQL 进而发展为关系型的数据库，对于关系型数据库，Dr. Codd's 定义了 12 条规则使之与其他类型的数据库相区别。

0. 关系型数据库必须通过关系来实现对数据的完全管理。
1. 所有在关系型数据库中的信息均可以在表中以数值的形式加以体现。
2. 在关系型数据库中的每一项数据均可以通过库名、键名和列名来准确指定。
3. 关系型数据库系统必须对空值（未知的和违规的数据）提供系统级的支持，有独特的缺省值，而且具有独立域。{这一段不太清楚}
4. 活动的、即时的数据联合 —— 它的意思就是在数据库中的数据应有逻辑表格的行的形式来表达，并且可以通过数据处理语言来访问。
5. 完善的数据子语句 —— 它应该至少支持一种有严格语法规则和功能完善的语言，并且应该支持数据和定义、处理、完整性、权限以及事务等操作。
6. 查看更新规则 —— 所有在理论上可以更新的视图可以通过系统操作来更新。
7. 数据库中数据和插入、更新与删除操作 —— 该数据库系统不仅要支持数据行的访问，还要支持数据和的插入、更新和删除操作。
8. 数据和物理独立性 —— 当数据在物理存储结构上发生变化时应用程序在逻辑上不应受到影响。
9. 数据的逻辑独立性 —— 当改变表的结构时应用程序在最大程度上不受影响。
10. 有效性独立 —— 数据库的语言必须有定义数据完整性规则的能力，数据应即时存储在线目录，而且在处理时必须通过这一五一节。
11. 发布的独立性 —— 当数据第一次发布或当它重新发布时应用程序应不受影响。
12. 任何程序不可能使用更低级的语言从而绕过数据库语言的有效性规则定义。

大多数数据库具有父/子关系，这就是说在父结点中保存有子结点的文件指针。（见下图）



这种方式有优点也有缺点，它的好处在于它使得数据在磁盘上的物理存储结构变得不再重要，编程人员只需存储下一个文件的指针就可以实现对下一个文件的访问，而且数据的添加和删除操作也变得非常容易，可是不同组的信息想要联合为一个新组就变得困难了，这是因为在这种方式下数据在磁盘上的存储格式不能在数据库建立以后再强制性地改变，如果需要这样做那就必须重新建立一个数据库结构。

Codd's 的关系型数据库思想借用的逻辑代数的思想，使得数据的子集与父级之间具有平等的地位。

由于信息可以很自然地组织在不同的表格中，Dr. Codd 也以这种方式来组织他所提出的数据库，在关系模式下，数据被存入类似于表格的结构中，这种表格由独立的数据元组（被称为列或字段）所组合而成，一组数据信息被存储为一行，举例来说，创建一个包括雇员内容的关系型数据库，我们可以很容易地从雇员表开始，而像这样的表在很容易得到的，该表中包含有如下信息：姓名、年龄、职业。这三项数据用作雇员表的字段。整个表如下图所示：

姓名	年龄	职业
Will Williams	25	Electrical Engineer
Dave Davidson	34	Museum Curator
Jan Janis	42	Chef
Bill Jackson	19	Student
Don DeMarco	32	Game programmer

Becky	Boudreaux	25	Model
-------	-----------	----	-------

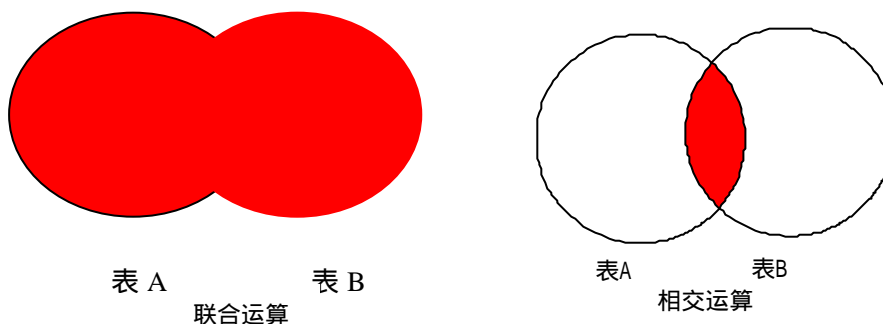
在这个表中有六行记录，为了从中找到特定的记录，举例来说：Dave Davidson，用户可以望知数据库管理系统在数据库中检索满足条件（姓名= Dave Davidson）的记录，如果数据库管理系统已经检索过了全部的数据，那么，它将会把满足条件的的姓名、年龄、职业三项的记录返回给用户，SQL 会通知 DBMS 找什么样的数据，这一检索过程的 SQL 例句如下：

```
SELECT * FROM EMPLOYEE
```

在这里不要刻意去记它的语句，我们在明天将会对它进行更为详细的讨论。

由于通过明显的关系可以特不同的数据项归结在一起（比如雇员的姓名和雇员的年龄），所以关系性数据库管理系统对如何来描述数据之间的关系给出了相当大的弹性，通过精确的的连接和联合运算，关系型数据库管理系统可以非常迅速地不同的表中将所需要的数据联合（见联合运算图）然后返回给用户或程序，这种联合的特性允许数据库的设计者将数据信息存储在不同的表中以减少数据的冗余度。

右图则反映了相交运算，相交运算地意思就是取出两个或多个库所共有的部分。



这里有一个简单的例子来显示数据是如何进行逻辑处理的，表 1-2 是一个被称为《报告》的表，它的里边有两个字段，姓名和工作。

Name	Duties
Becky Boudreaux	Smile
Becky Boudreaux	Walk
Bill Jackson	Study
Bill Jackson	Interview for jobs

在雇员数据库中的年龄和职业字段在每一个记录中均出现重复是不合适的。而且随着时间的进行，这些冗余的数据将会占用大量的磁盘空间且使得数据库管理系统在检索数据

所耗用的时间增多。可是，如果你将姓名和工作另存到一个名字叫《报告》的库中以后，你可以通过联合操作将《报告》与《雇员》通过姓名字段进行联合操作，也就是通知 RDBMS 将《报告》与《雇员》库中姓名与 Becky Boudreaux 相同的记录显示出来，其结果将如下所示：

Name	Age	Occupation	Duties
Becky Boudreaux	25	Model	Smile
Becky Boudreaux	25	Model	Walk

关于联合运算的详细讲述将在第 6 天的《库的联合》中讲述。

设计数据库的结构

在数据库的设计师已经决定了系统的硬件平台和 RDBMS 系统以后，余下事情中最为重要的就是如何来设计数据库的结构了。数据库的结构将会影响到是后运行于该库上的应用程序的性能，这个决定数据库的分配情况及联合运算的过程称之为标准化。

数据库的前景

电脑技术将对今天世办上的商业产生深远的影响，鼠标只要点一下就可以将数据入库或对其进行访问，制造商的国外订货单可以立即接受并执行，尽管在 20 年以前信息的交换还需要大型机的支持，而办公领域处理事务也仍在采用批处理的方式，要完成某一个查询，用户需要将需求提交给服务器上的信息管理系统（MIS），结果将会以最快的速度返回给用户（尽管经常不是足够快）。

此外，随着关系型数据库模型的发展，有两种技术被引用到了在今天被称为服务器/客户机的数据库系统当中，第一项技术就是个人电脑，廉价而又易用的应用程序如 Lotus1-2-3 和 WordPerfect 允许员工（或家庭用户）可以建立文档来快速而准确地处理数据。用户也会经常升级他们的系统以使其速度更快，巧的是这时的系统的价格却在迅速下跌。

第二项技术则是局域网的发展，它导致的世界范围内的办公交叉——虽然用户习惯于采用终端同主机相连，在今天，一个字处理文档可以存储在本地而被任何连接到网络上的电脑访问，然后苹果的 Macintosh 电脑为大家提供了一个友好易用的图形用户界面，使得电脑变得物美价廉，此外，他们可以访问远程站点，并从服务器上传大量的数据。

在这个飞速发展的时期，一种新型的叫作服务器/客户机的系统诞生了，这种系统的处

理过程被分解上了客户机和数据服务器上，新的应用程序取代了基于主机的应用程序，这一体系有着相当多的优点：

- 降低了维护费用
- 减轻的网路负荷（处理过程在服务器上和客户机上均有）
- 多个操作系统可以基于相同了网络协议来共同工作。
- 本地化的数据操作提高了数据的完整性。

对于什么是客户机/服务器型电脑系统，Bernard H. Boar 的定义如下：

客房机/服务器系统就是把单一的任务分解到多个处理器上进行协同处理就像在单个处理器上运行时一样。一个完备的客户机/服务器系统可以将多个处理器捆绑在一起以提供一个单一系统虚拟环境。共享的资源可以被位于远端的客户机通过特殊的服务来访问，这种结构可以逐级递归，所以一级服务器可以在最后转变为客户机进而要求其他的服务器提供服务。就这样一直下去。

这种类型的应用程序在设计时需要全新的程序设计技巧，今天的用户界面都是图形用户界面，不论是微软的 WINDOWS、苹果的 MACINTOSH、IBM 的 OS/2 还是 UNIX 的 X-windows 系统均是如此。用过使用 SQL 和网络，应用程序就可以访问位于远端服务器上的数据库，个人电脑处理能力的提高可以对存放在一系统相关的服务器的数据库作出评定，而这此服务器是可以更换的，而应用程序则只需做出较少的改动甚至无需改动。

交互式语言

本书在许多场合下可以借用 BASIC 的概念，举例来说：Microsoft Access 是基于 windows 的单用房应用程序而 SQL SEVER 则可以允许 100 个用户同时工作，SQL 的最大优越性在于它是一种真正的跨平台的交互式语言，由于它可以被程序员在第四代的编程语言中调用，第四代编程语言可以做用少量的代码做大量的工作。

易于实现

ORACLE 公司是第一个发行基本于 SQL 的关系型数据库管理系统（RDBMS）的公司，虽然它是为 VAX/VMS 系统开发的，ORACLE 公司也是 DOS 下的关系型数据库的供应商之一（ORACLE 现在可以运行在近 70 种平台之上）。在八十年代中期，Sybase 公司发行了

他们的 RDBMS ——SQL Sever 具有客户端数据库访问功能并支持过程存储（将在第 14 天的动态应用 SQL 中提到）和多平台交互工作能力，SQL Sever 作为一个成功的产品，其客户机/服务器工作能力是其最为突出的优势所在，这个强大的数据库系统具有极高的平台适应性，用 C 语言写成的运行于 PC 机的 ORACLE 其实是运行于 VAX 系统上的 ORACLE 的复制。

SQL 与客户机/服务器应用程序开发环境

在使用客户机/服务器电脑来开发客户机/服务器应用程序时 SQL 和关系型数据库的思想遍及始终，在单用户系统中使用这种技术也可以使您的程序更适应未来的发展。

SQL 总览

SQL 是操作和检索关系型数据库的事实上的标准语言，它允许程序员和数据库管理员做如下的工作：

- 更改数据库的结构
- 更改系统的安全设置
- 增加用户对数据库或表的许可权限
- 在数据库中检索需要的信息
- 对数据库的信息进行更新

注：对于 SQL 大家可能还不明白，S 即 Structured（结构），L 即 Language（语言）这是显而易见的，但是 Q 的意思容易让人误解，Q 的意思当然是 Query（查询）——如果你直译的话。可是这只限于你对数据库提问。但是 SQL 能干的不只是查询，通过它可以建立一个库，添加和删除数据，对数据作联合，当数据库改变时触发动作，并把你的查询存储在程序或数据库中。

不幸得很，这对于查询来说似乎是一个缺点，显然，库结构的增加、删除、修改、联合、存储、触发以及查询语言在多用户协同工作时有点烦琐。我们将会在工作中一直与 SQL 打交道，不过现在你应该知道它的功能不只是限于它的名字所指的内容了。

SELECT 语句是 SQL 中应用最多的语句（见第二章：查询——SELECT 语句的使用），它会从数据库中检索需要的数据并把结果返回给用户。上边的雇员表的举例便是一个典型

的 SELECT 语句的使用例子，除了 SELECT 语句之外，SQL 还提供了用以建立新的数据库、表格、字段和索引的语句以及记录的插入和删除语句。你将会发现，ANSI SQL 还提供了对核心数据操作的功能，许多数据库管理系统还提供了确保数据完整性和强制安全性的工具（见第 11 天的传输控制），它允许程序员在当前环境不符合的时候强制性地终止语句组的执行。

流行的 SQL 开发工具

这一部分将介绍一些大众化的 SQL 开发工具，每一种工具都有它的优点和缺点，一些工具是基于 PC 用户的，强调易用性，而另一些则是为超大型数据库提供的。本部分将向您介绍选择这些工具的关键所在。

注：为了 SQL 在实际中的使用，本书中举出了一些 SQL 在实际开发环境中的应用例子，SQL 只有出现在你的代码中，为你真正地解决了问题才说明它是有用的。

Microsoft Access

在一些应用实例中我们将会举一些 Microsoft Access 的例子，Microsoft Access 是一个非常容易使用的基于 PC 机的数据库管理系统，在它的下边你既可以手工输入 SQL 语句也可以使用图形用户界面工具来生成 SQL 语句。

Personal Oracle7

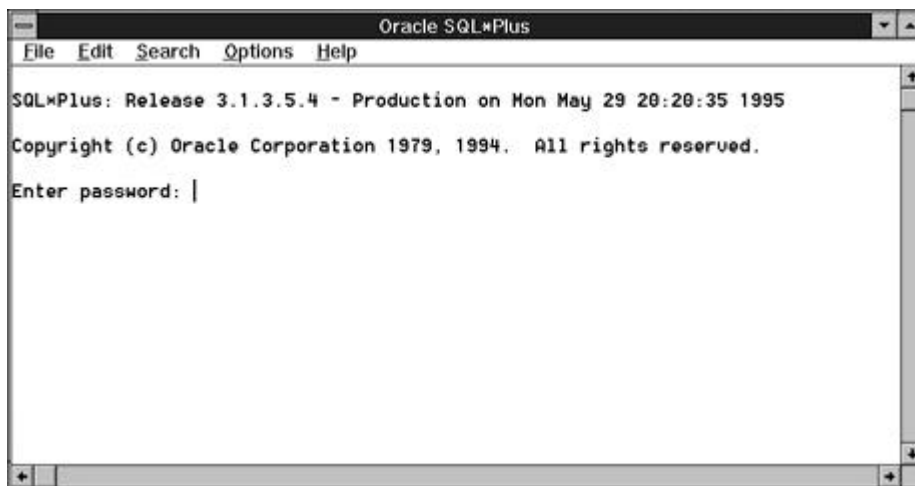
我们使用 Personal Oracle7 来向大家演示 SQL 对大型数据库上的命令行使用方法，当用户对一个数据库或一个操作系统有充分的了解以后，需要在一个孤立的电脑上进行设计时这种方法是十分重要的。在命令行下，用户可以在 SQL PLUS 工具中输入不同的单的 SQL 语句，该工具可以把数据返回给用户，或是对数据库进行适当的操作。

大多数例子是针对初用 SQL 进行程序设计的程序员的，我们从最简单的 SQL 语句开始并进阶到事务处理阶段，为程序设计做好准备，Oracle 的发行版提供一整套的开发工具，它包括 C++和 Visual Basic 函数库(Oracle Objects for OLE)，通过它可以将应用程序与 ORACLE 个人数据库链接在一起，它也可以为数据库、用户或管理员提供图形工具，同 SQL*Loader 一样，它也经常用于从 Oracle 数据库中导出或引入数据。

注：Personal Oracle7 是 Oracle7 server 的不完整版，它只允许单用户操作（就如同它的名字一样），但它在 SQL 的语法使用与大型、更昂贵的 Oracle 版本是相同的，此外，在 Personal Oracle7 中所使用的工具也适用于 Oracle 的其他版本。

我们选择 Personal7 基于以下原因：

- 它几乎用到了本书中将要讨论的所有工具。
- 它是可以在几乎全部的平台运行的，风靡世界的关系型数据库管理系统。
- 可以从 Oracle 公司的服务器上下载一个它的 90 天限时版(<http://www.oracle.com>)。

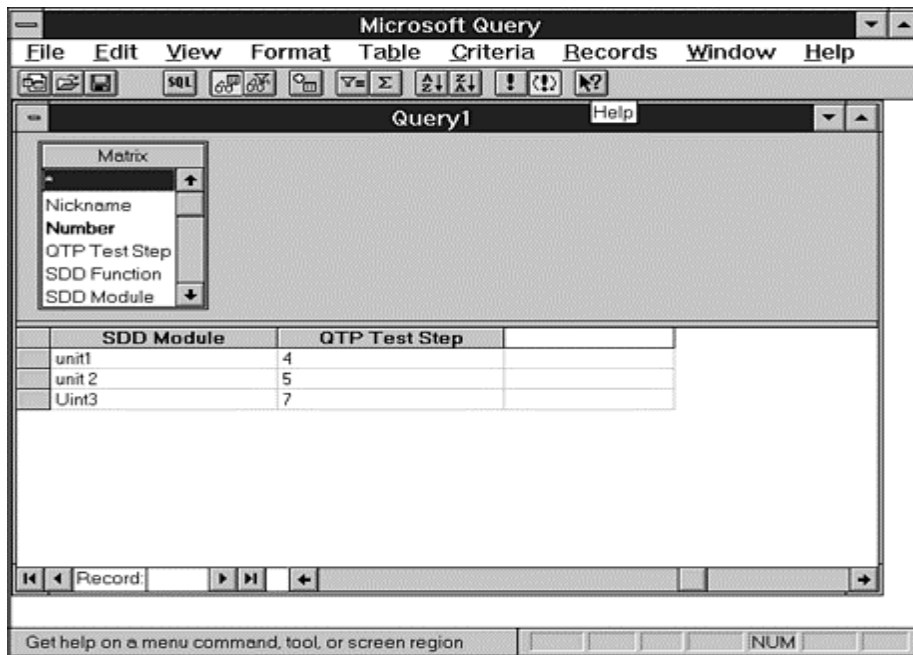


上图显示了 SQL-PLUS 的一个画面。

技巧：注意在本书中所给出的所有 SQL 代码都适用于其它的数据库管理系统，防止在其它的系统中语法出现大的差别，在本书的例子中指出了它在其它系统中的不同之处。

Microsoft Query

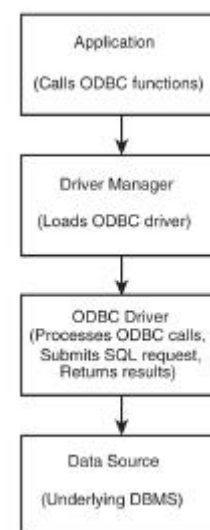
Microsoft Query 是 Microsoft 公司的 Visual C++ 和 Visual Basic 开发工具包中所附带的一个非常有用的查询工具。它可应用在基于 ODBC 标准下的数据库，该查询工具在将查询语句提交给数据库之前会将基保留在驱动器上。



开放型数据库联接（ODBC）

ODBC 是为应用程序接口（API）提供的访问下层数据库所用的函数库，它通过数据库引擎与数据库交流，就像在 Windows 通过打印驱动程序与打印机通信那样。为了访问数据库，可能还会需要网络驱动程序与异地数据库通信，ODBC 的结构如右图所示。

ODBC 的特色在于它不是针对任何一种数据库的，举例来说，你可以用相同的代码来在 Microsoft Access 表中或 Informix database 上运行查询，而无需修改代码或只需做很小的改动。再提醒您一次，第三方数据库供应商可能会在 SQL 的标准之上对其进行扩充，比如 Microsoft 和 Sybase 的 Transact-SQL 以及 Oracle 的 PL/SQL。



当您用一种新型的数据库工作时您应该认真阅读一下它的文档，ODBC 是许多数据库所支持的一种标准，包括 Visual Basic，Visual C++，FoxPro，Borland Delphi 和 PowerBuilder。基于 ODBC 所开发的应用程序有着明显的优势，因为它允许你在编写代码的时候不必考虑是为哪一个数据库所写的。当然，它的运行速度要弱于特定的数据库代码，也就是说在使用 ODBC 的时候其灵活性更强但比起使用 Oracle7 或 Sybase 的函数库时要慢。

SQL 在编程中的应用

SQL 的最初标准是 ANSI-1986，而在 ANSI-1989 中定义的 SQL 在应用程序中的三种接口类型。

- 模块语句——在程序中使用过程，该过程可以通过主调参数向主调函数返回值。
- 内嵌 SQL——可以在编写程序的过程中内嵌 SQL 语句，该方式在经常需要对 SQL 语句进行预编译处理时所需要，在 Pascal，FORTRAN，COBOL，PL/1 中均定义了这样的子句。
- 直接调用——由程序直接实现。

在动态 SQL 发展以前，内嵌 SQL 在编程中应用最为流行。这种方法在今天也仍然在使用。由于静态的 SQL——它的意思就是 SQL 语句已经被编译在了应用程序之中，不能在运行的过程中改变。这与编译程序同解释程序的区别类似，也就是说这种类型的 SQL 速度很快，但是灵活性很差，这在今天的商业应用领域是不适宜的。动态 SQL 这里就不多说了。

ANSI-92 标准将 SQL 语言标准扩展为一种国际化的标准，它定义了 SQL 的三种编译级别：登录调用、内嵌子句和完全编译，主要的新特性如下：

- 联接到数据库
- 移动游标
- 动态 SQL
- 外连接

本书除了这些扩展以外还包括了第三方数据库供应商所提供的扩展，动态 SQL 允许你在运行时修改 SQL 语句，但是它的速度要比内嵌型 SQL 慢，它在调用级接口上为应用程序开发人员提供了相当大的灵活性，ODBC 或 Sybase 的 DB-Library 就是动态 SQL 的例子。

调用级接口对于编程人员来说不是一个新概念，当您在使用 ODBC 时，举例来说：你需要在 SQL 语句提交给数据库时修改子句中的变量参数而使用该功能，在设计阶段可以通过使用其它的函数调用接收错误信息或结果，结果是以数据包的形式返回的。

摘要

在第一天介绍了 SQL 的历史，由于 SQL 与关系型数据库是紧密结合的，所以第一天也简要地介绍了关系型数据库的历史和它的功能，明天我们将讲述 SQL 中最为常用的功能——查询。

问与答

问：为什么我要了解 SQL？

答：到现在为止，如果你不知道如何利用大型数据库来进行工作，如果你要使用客户机/服务器型应用程序开发平台（如：Visual Basic，Visual C++，ODBC，Delphi，和 PowerBuilder）以及一些已经移植到 PC 平台上的大型数据库系统，（如：Oracle 和 Sybase）来进行开发工作。那么你只有学习 SQL 的知识。在今天大多数的商用程序开发都需要你了解 SQL。

问：在学习 SQL 时我为什么要了解关系型数据库系统？

答：SQL 在为关系型数据库系统开发的，不知道点关系型数据库系统的知识你就无法有效地使用 SQL。

问：在 GUI 图形用户界面工具下我只需按按钮就可以写出 SQL 语句，为什么我还需要学习手工写 SQL？

答：GUI 有 GUI 的方法，手工有手工的方法，一般说来手工写出来的 SQL 比 GUI 写出来的更有效率，GUI 的 SQL 语句没有手工写出的易读，而且如果你知道如何用手来写 SQL 的话那么你在使用 GUI 时就会有更高的效率。

问：既然 SQL 是一种标准化语言，那是不是说我可以用它在任何数据库下进行编程？

答：不是，你只能将它用于支持 SQL 的关系型数据库系统，如 MS-Access，Oracle，Sybase 和 Informix。尽管不同的系统在执行时有所差别，但是你只需要对你的 SQL 语句进行很小的调整。

校练场

在校练场里我们提出了一些问题以帮助你巩固自己所学，这些练习可以提高你在学习中的经验，请试着回答和练习附录五（问答与练习）中的内容。

- 1、为什么说 SQL 是一种非过程型语言。
- 2、我如何知道一种数据库系统是不艺机是关系型数据库系统。
- 3、我可以用 SQL 来做什么。
- 4、把数据清楚地分成一个个唯一集的过程叫什么名字。

练习：

确认一下你所使用的数据库系统是否是一个关系型数据库系统。

第二天：查询 ——SELECT 语句的使用

目标：

欢迎来到第二天，在今天你将学习到以下内容：

- 如何写 SQL 的查询。
- 将表中所有的行选择和列出。
- 选择和列出表中的选定列。
- 选择和列出多个表中的选定列。

背景

在第一天中我们简要地介绍了关系型数据库系统所具有的强大功能，在对 SQL 进行了简要的介绍中我们知道了如何同它进行交流。最终，我们将会与计算机用一种非常清楚、果断的话说：“给我看一下所有在本公司中工作十年以上，左撇子，蓝眼睛的外国人”。如果你能够这样做（与计算机交流，而不是查他们的档案）。每一个人都可以用他自己的方法来达到目的，但是你却是用 SQL 的一种重要功能——查询来达到目的。

在第一天中我们说过，查询一词用在 SQL 中并不是很恰当，在 SQL 中查询除了向数据库提出问题之外还可以实现下面的功能。

- 建立或删除一个表。
- 插入、修改、或删除一个行或列。
- 用一个特定的命令从几个表中查找所需要的信息并返回。
- 改变信息的安全性。

SQL 的查询当然也能进行一般的查询工作，在学会使用这个有用的工具之前，我们来学习如何写 SQL 的查询语句。

一般的语法规则

正如你所看到的那样，SQL 有很高的灵活性，尽管在任何程序中都有一定的规则限制，

下而有一个 SQL 中 SELECT 语句使用的简单例子，请注意：在每个 SQL 语句的关键字都是大写的，并且用空格将他们划分出来。

```
SELECT  NAME, STARTTERM, ENDTERM
FROM    PRESIDENTS
WHERE   NAME= 'LINCOLN';
```

在这个例子中每一个字母都是大写的，但是这不是必需的，上边的查询语句完全可以写成这样：

```
select name, startterm, endterm
from  presidents
where name= 'LINCOLN';
```

注意，LINCOLN 在这里仍然是大写的，尽管 SQL 语句对大小写并不敏感，但在数据库中的数据却是大小写敏感的，举例来说：许多公司在储存数据时用大写字母，在这种情况下，所有的字段名也将是大写字母，那么在检索条件为 name='Lincoln'的数据时将不会得到任何结果，这种情况在每个实例应用中都会遇到。

注意：在 SQL 语句中大小写是不敏感的。

现在我们来看另一个例子，在这个例子中的空格有问题吗？不是，这个语句完全可以正常执行。

```
Select name, startterm, endterm from presidents where name='LINCOLN'
```

但是，如果你注意在你的语句中使用空格和大写字母会增强语句的可读性，当它变成你的工程（编程）的一部分时会更便于维护。

另一个重要的特性是分号，当在 SQL 语句中出现分号就意味着本条语句已经结束。

为什么在格式中大小写是不重要的，原因何在。答案是——关键字。关键字是 SQL 语法中的保留字，在 SQL 语句中，关键字是可选择的，但其内容有强制性。在本例中的关键字有：

```
SELECT
FORM
WHERE
```

看一下目录，你会找到需要在其它几天中学习的关键字。

数据报的形成——SELECT 和 FROM

随着对 SQL 的了解，你会发现你键入的 SELECT 和 FROM 在远远多于其它的关键字，它不像 CREATE 那样迷人或像 DROP 那样残忍。但是如果你在同计算机会话并需要计算机返回结果时它们却是必不可少的，这与最初选择何种数据库没有关系？

我们先从 SELECT 开始讨论，因为 SELECT 是在 SQL 中使用最为频繁的语句。

语法：

SELECT <列名>

没有其它的语句可以比 SELECT 语句更简单了，但是 SELECT 语句不从独立工作，如果你只是键入了 SELECT 语句，那么你将收到如下信息：

输入：

```
SQL> SELECT;
```

输出：

```
SELECT
```

```
*
```

```
ERROR at line 1:
```

```
ORA-00936: missing expression
```

当在访问 ORACLE 时会有*出现以表示有事件产生，错误信息的意思是告诉你有一个东西丢了。这个丢失的东西就是 FROM 子句：

语法：

FROM <表名>

当两条语句结合使用时就有了后台访问数据库的能力。

注：你可能会对子句、关键字、或 SQL 语句感到费解。SQL 的关键字是 SQL 中的特定元素，如 SELECT 和 FROM。子句则是 SQL 语句的一部分，如，SELECT column1, column2, ... 就是一个子句，而 SQL 语句则是几个子句的结合，例如你可以将 SELECT 子句和 FROM 子句组合成为一个 SQL 语句。

注：每一个种 SQL 都有其特定的出错信息，例如 Microsoft Query 会显示说它不能运行查询，并引导你发现错误所在；Borland's Interbase 将会弹出一个错误对话框，Personal

Oracle7 的引擎将会出现如前所述的信息, 并给出一个错误号码(所以当你手工输入 SQL 语句时会看到详细的错误信息) 以及对错误的简要诠释。

例子

在进一步学习之前, 我们先来看一个将要在下面的例子中用到的数据库, 这个数据库体现了 SELECT 和 FROM 的基本功能, 在实际应用时我们将会用到在第 8 天(熟练地操作数据) 中讲到的技巧来构建这个数据库。但是我们现在的目的是学习如何使用 SELECT 和 FROM, 所以我们假设数据库已经建好了, 本例中使用 CHECKS 表, 这个表的内容如下:

CHECK#	PAYEE	AMOUNT	REMARKS
1	MaBell	150	Havesonsnexttime
2	ReadingR.R.	245.34	TraintoChicago
3	MaBell	200.32	CellularPhone
4	LocalUtilities	98	Gas
5	JoesStale\$Dent	150	Groceries
6	Cash	25	WildNightOut
7	JoansGas	25.1	Gas

你的第一个查询

输入:

SQL>select * from checks

输出:

CHECK#	PAYEE	AMOUNT	REMARKS
1	Ma Bell	150	Have sons next time
2	Reading R.R	245.34	Train to Chicago
3	Ma Bell	200.32	Cellular Phone
4	Local Utilities	98	Gas
5	Joes Stale \$ Dent	150	Groceries
6	Cash	25	Wild Night Out
7	Joans Gas	25.1	Gas

7 rows selected.

分析:

请看例子的输出结果, 注意第 1 列和第 3 列是右对齐的而第 2 列和第 4 列是左对齐的,

这是因为对于数字类型采用右对齐而对于字符类型则是采用左对齐的。数据的类型将在第 9 天的《表的建立与维护》中讨论。

在 SELECT 中的*表示要返回 FROM 中所指定的表中的所有列，并按照数据库中的固有顺序来排序。

完成一个 SQL 语句

在 SQL 运行时，分号(;)即意味着通知解释程序当前语句已经结束。例如：SQL*PLUS 在没有遇到分号时将不会执行语句。但是在其它的 SQL 解释器中可能不会用到分号，例如：Microsoft Query 和 Borland's ISQL 不需要查询终止符，因为你是编辑框中输入查询语句并且当你在按下按钮以后才开始执行查询。

对列进行排序

在前边的例子中使用了*来选择了选定表格中的所有列，并且是按照其在数据库中的固定顺序来排序的，如果需要对特定的列排序，你应该按下边所写的那样输入：

输入

```
SQL> SELECT payee , remarks , amount , check# from checks;
```

注意在 SELECT 子句中给出了每个列的名字，排序是根据列的先后顺序来进行的，注意将最后列的列名与其后的子句（这里是 FROM）用空格分开，输出的结果如下：

输出：

PAYEE	REMARKS	AMOUNT	CHECK#
Ma Bell	Have sons next time	150	1
Reading R.R.	Train to Chicago	245.34	2
Ma Bell	Cellular Phone	200.32	3
Local Utilities	Gas	98	4
Joes Stale \$ Dent	Groceries	150	5
Cash	Wild Night Out	25	6
Joans Gas	Gas	25.1	7

7 rows selected.

这句话也可以写成下边的形式：

输入：

```
SELECT payee, remarks, amount, check#
```

```
FROM checks;
```

注意，这里的 FROM 子句已经写到第二行了，这是一种个人习惯。其输出的结果如下：

输出：

PAYEE	REMARKS	AMOUNT	CHECK#
Ma Bell	Have sons next time	150	1
Reading R.R.	Train to Chicago	245.34	2
Ma Bell	Cellular Phone	200.32	3
Local Utilities	Gas	98	4
Joes Stale \$ Dent	Groceries	150	5
Cash	Wild Night Out	25	6
Joans Gas	Gas	25.1	7

7 rows selected.

分析：

语句的格式变更不会对输出的结果造成影响，现在你已经知道了如何对输出的结果进行排序，试着将表格的列按照你的要求进行排序。

选择特定的列

如果你不想看到数据库中的每一列，当然，你使用 SELECT * 将所有的列显示出是可行的，但是如果你只想看一下 CHECKS 中的号码与数量列，那么你可以输入如下语句：

输入：

```
SQL> SELECT CHECK#, amount from checks;
```

输出：

CHECK#	AMOUNT
1	150
2	245.34
3	200.32
4	98
5	150
6	25
7	25.1

现在你可以按要求显示所需要的列，注意大小写的使用，它不会对查询的结果造成影响

响。

如何从不同的表中查找到所需要的信息呢？

从不同的表中选择

假设你有一个名为 DEPOSITS 表，其内容如下：

输出：

DEPOSIT#	WHOPAID	AMOUNT	REMARKS
1	Rich Uncle	200	Take off Xmas list
2	Employer	1000	15 June Payday
3	Credit Union	500	Loan

分析：

你需要对数据源作一下简单的改动：

查找不重复的数据

如果你看过原来的 CHECKS 表，你会发现其中有一些数据是重复的，例如，AMOUNT 列。

输入：

```
SQL> select amount from checks;
```

输出：

AMOUNT
150
245.34
200.32
98
150
25
25.1

请注意，150 在这里是重复的，如果你只想查看不重复的数据，可以这样做：

输入：

```
SQL> select DISTINCT amount from checks;
```

输出：

 AMOUNT

25

25.1

98

50

200.32

245.34

 6 rows selected

分析：

注意只有六行数据被选择，这是因为你使用了 **DISTINCT**，所有只有不重复的数据才会被显示，**ALL** 是在 **SELECT** 中默认的关键字，你几乎从来也不会用到 **ALL**，因为 **SELECT** 与 **SELECT ALL** 是等价的。

试一下这个例子，作为你对 SQL 的第一次（也是唯一的一次）实际体验。

输入：

SQL> SELECT ALL AMOUNT

FROM CHECKS;

 输出：

AMOUNT

150

245.34

200.32

98

150

25

25.1

 7rowsselected.

它的结果与 **SELECT <Column>** 是相同的，谁还会再去用这个多余关键字呢？

总结：

关键字 **SELECT** 可以检索数据库并从中返回数据，你可以用一个很长的语句并使用 **SELECT *** 来检索数据库中的所有表，而且你可以对指定表格的结果进行重新排序。而关键

字 `DISTINCT` 则会强制性地要求返回的结果中不能有重复数据。明天我们将学习如何使您的查询更具有选择性。

问与答：

问：这些数据是从哪里来的，我们是如何得到它的？

答：数据是按照第 8 天所讲述的方法创建的，与数据库的联接是依靠你所使用的 SQL，它以传统的命令行方式与数据库进行会话。该数据库原来属于服务器或客户机范畴，但最近它已经被移植到了 PC 机上。

问：可是我用不到这些数据库，那我还可以用 SQL 干什么？

答：你也可以在编程语言中使用 SQL，一般的编程语言都支持内嵌的 SQL，例如 COBOL，你可以在它的环境中写 SQL 并编译，而 Microsoft 公司则提供了应用程序接口函数以允许编程人员在 Visual Basic，C，或 C++ 中使用 SQL。Sybase and Oracle 提供的库也允许你在编程时使用 SQL，Borland 公司则将 SQL 置于 Delphi 中，本书中也将讨论 SQL 在编程中的应用。

校练场

在校练场里我们提出了一些问题以帮助你巩固自己所学，这些练习可以提高你在学习中的经验，请试着回答和练习附录五（问答与练习）中的内容。在开始明天的工作之前要确保你已经知道了这些问题的答案。

1、 下列语句所返回的结果是否相同？

```
SELECT * FROM CHECKS;
```

```
select * from checks;
```

2、为什么下列查询不会工作？

a. Select * b. Select * from checks

c. `Select amount name payee FROM checks;`

34

A select *

From checks:

```
B> select * from checks;
```



```
C> select * from checks  
/
```

练习：

- 1、 使用今天早些时候的 CHECKS 表的数据来写一个查询，返回表中的 number 和 remark 列中的数据。
- 2、 将练习 1 中的查询再写一遍以使得 remark 列出现在第一位。
- 3、 使用 CHECKS 表，写一个查询来返回其中的不重复数据。

第三天：表达式、条件语句与运算

目标：

在第二天我们学习了使用 SELECT 语句和 FROM 语句来对数据进行有趣味性（也是非常有用的）运算，在今天我们将对昨天学习的进行进一步的扩充。我们将把新的方法应用到查询、表和行中，引进新的子句和被称为运算的批量处理方法，在第三天的阳光下，你将学会：

知道什么叫作表达式以及如何来使用它们。

知道什么叫作条件语句以及如何来使用它们。

熟悉基本的子句 WHERE 的使用

可以用算术、比较、字符和逻辑表达式来建立一个运算

学会将多种不同的运算结合在一起使用

注：在今天的学习中我们来使用 PERSONAL ORACLE7 来进行应用举例，其它的 SQL 环境在命令运算以及结果显示上与它稍有不同，但在遵循 ANSI 标准的基础上它们的结果应该是相同的。

表达式：

表达式的定义非常简单：表达式可以返回一个值，表达式的类型非常广泛，它以包括各种类型的数据如数字、字符、以逻辑型等，其实在下列子句（如 SELECT 和 FROM）中所占成分中表达式最大，在下边的例子中 amount 就是一个表达式，它可以返回 amount 列中的数据。

```
SELECT amount FROM checks;
```

而在下列语句中 NAME, ADDRESS, PHONE, ADDRESSBOOK 是表达式：

```
SELECT NAME, ADDRESS, PHONE  
FROM ADDRESSBOOK;
```

现在，请检查一下下边的表达式：

```
WHERE NAME = 'BROWN'
```

这里，NAME = 'BROWN' 是一个条件语句，这是一个逻辑形表达式的实例，NAME = 'BROWN' 将根据=号来返回值 TRUE 或 FALSE。

条件

如果你想在数据库中查找一个或一组特定的信息，你需要使用一个或更多的条件。条件可以包含在 WHERE 子句中，在上一个例子中，条件就是：

```
NAME = 'BROWN'
```

如果你想知道在你们单位中上一个月有谁的工作时间超过了 100 个小时，你可能会写出下边的条件语句：

```
NUMBEROFHOURS > 100
```

条件语句可以让你建立一个选择查询，在大多数情况下，条件中包括变量，常量和比较运算，在第一个例子中的变量是 NAME，常量是'BROWN'，而比较运算符则为=。在第二个例子中变量为 NUMBEROFHOURS,常量为 100，而比较运算符则是>，当您准备写一个条件查询时你需要知道两个元素：WHERE 子句和运算。

WHERE 子句

Where 子句的语法如下：

```
WHERE <SEARCH CONDITION>
```

Select, From 和 Where 在 SQL 中最常使用的三个子句, Where 只是当你的查询具有更大的选择性, 没有 Where 子句, 你可以用查询做得最多的有用工作是显示选定表中的所有记录, 例如:

输入:

SQL> SELECT * FROM BIKES;

这将会将 BIKES 表中的所有数据按行列出。

输出:

NAME	FRAMESIZE	COMPOSITION	MILESRIIDEN	TYPE
TREK 2300	22.5	CARBONFIBER	3500	RACING
BURLEY	22	STEEL	2000	TANDEM
GIANT	19	STEEL	1500	COMMUTER
FUJI	20	STEEL	500	TOURING
SPECIALIZED	16	STEEL	100	MOUNTAIN
CANNONDALE	22.5	ALUMINUM	3000	RACING

假若你想要一台特定型号的自行车, 你应该键入:

SQL> SELECT * FROM BIKES WHERE NAME = 'BURLEY';

你将只会收到一个记录:

输出:

NAME	FRAMESIZE	COMPOSITION	MILESRIIDEN	TYPE
BURLEY	22	STEEL	2000	TANDEM

分析:

这个简单的例子显示出了你可以在数据库返回的数据中加以条件限制。

运算

运算是你需要对从数据库中返回的数据进行数学处理时所用到的元素。运算可以归为六组:

数值型, 比较型, 字符型, 逻辑型和备注型以及混合型。

数值型运算:

数值型运算有加、减、乘、除和取模。前四个不用多说, 取模将返回一个除法结果中商的余数部分, 这里有两个例子:

$5\%2=1$

$6\%2=0$

对于有小数的数据不能应用取模运算, 如实数。

如果你在进行数据运算时应用了几个运算符而没有在其中使用括号, 那么运算进行的次序将是先乘后除再模后加减, 举例来说, 表达式: $2*6+9/3$ 其结果将是 $12+3=15$ 但是, 表达式 $2*(6+9)/3$ 结果则为 $2*15/3=10$ 注意在这里你使用了括号, 有时表达式不会按你所想像的那样得出期望的结果。

加法(+)

你可以在许多场合下使用加号, 下面的语句将显示一个价格表:

输入: SQL> SELECT * FROM PRICE;

输入如下右:

现在请输入:

SQL>SELECT ITEM, WHOLESAL, WHOLESAL+0.15 FROM PRICE; OUTPUT

ITEM	WHOLESAL
TOMATOES	.34
POTATOES	.51
BANANAS	.67

ITEM	WHOLESALE
TURNIPS	.45
CHEESE	.89
APPLES	.23

这里对于下列产品的每一个价格数据加了 15 分：

ITEM	WHOLESALE	WHOLESALE+0.15
TOMATOES	.34	.49
POTATOES	.51	.66
BANANAS	.67	.82
TURNIPS	.45	.60
CHEESE	.89	1.04
APPLES	.23	.38

分析：

请不要忽视最后一列 `WHOLESALE+0.15`，它在原始的数据库表中没有（切记，你在 `SELECT` 中使用了 * 号，这将会显示出所有的列）。SQL 允许你创建一个虚拟列或对已有的列进组合和修改后产生的派生列。

请再输入一次刚才的语句

```
SQL> SELECT * FROM PRICE;
```

右面是从表中返回的结果：

ITEM	WHOLESALE
TOMATOES	.34
POTATOES	.51
BANANAS	.67
TURNIPS	.45
CHEESE	.89
APPLES	.23

分析：

输出的结果有时原始数据并没有被改变，而标题为 `WHOLESALE+0.15` 的列也不是表中的固有列，事实上，由于这个列标题太不容易为人所注意，所以你应该在它的上边再花一些工夫。

请输入：

```
SQL> SELECT ITEM, WHOLESALE, (WHOLESALE+0.15) RETAIL
      FROM PRICE;
```

其结果如右：

ITEM	WHOLESALE	RETAIL
TOMATOES	.34	.49
POTATOES	.51	.66
BANANAS	.67	.82
TURNIPS	.45	.60
CHEESE	.89	1.04
APPLES	.23	.38

分析

真棒，你不但可以创建一个新列，而且还可以对它按自己的需要进行重命名。你可以按语法列名 别名来对任何一个列进行重命名（注意在列名与别名之间有空格）。

例如，输入

```
SQL> SELECT ITEM PRODUCE, WHOLESALE, WHOLESALE+0.25 RETAIL
      FROM PRICE;
```

重命名的列如下：

PRODUCE	WHOLESALE	RETAIL
---------	-----------	--------

TOMATOES	.34	.49
POTATOES	.51	.66
BANANAS	.67	.82
TURNIPS	.45	.60
CHEESE	.89	1.04
APPLES	.23	.38

注：一些 SQL 解释器使用的语法为《列名》=《别名》。所以前一个例子要写成如下格式：

```
SQL> SELECT ITEM=PRODUCE,
WHOLESALE,
WHOLESALE+0.25=RETAIL,
FROM PRICE;
```

请检查你的 SQL 解释器以确认它采用哪一种语法。

你大概想知道当不在命令行状态时应如何使用别名吧！很清楚，你知道报表生成器是如何工作的吗？总有一天，当有人让你写一个报表的生成器时，你就会记住它而且不用却重复 Dr. Codd 和 IBM 已经做过的工作。

到现在为止，你已经看到了两种加号的用法，第一种用法是在 SELECT 子句中使用+号以执行对数据的运算并将结果显示出来。第二种用法是在 WHERE 子句中使用加号，在 WHERE 中使用操作符可以在当你对数据有特定条件时具有更大的灵活性。

在一些解释器中，加号还同时肩负着进行字符运算的责任，在稍后的几天中你将会看到这一点。

减法 (—)

减号也有两种用途，第一种用途是作为负号使用，你可以使用 HILOW 表来验证这项功能。

```
SQL> SELECT * FROM HILOW;
```

输出：

STATE	HIGHTEMP	LOWTEMP
CA	-50	120
FL	20	110
LA	15	99
ND	-70	101
NE	-60	100

例如，这里对数据进行这样的运算：

```
SQL> SELECT STATE, —HIGHTEMP LOWS, —LOWTEMP HIGHS FROM HILOW;
```

STATE	LOWS	HIGHS
CA	50	-120
FL	-20	-110
LA	-15	-99
ND	70	-101
NE	60	-100

第二种用法（很明显）是作为减号从某一系列中减去另一列，例如：

```
SQL> SELECT STATE,
HIGHTEMP LOWS,
LOWTEMP HIGHS,
(LOWTEMP - HIGHTEMP) DIFFERENCE
FROM HILOW;
```

STATE	LOWS	HIGHS	DIFFERENCE
CA	-50	120	170
FL	20	110	90

LA	15	99	84
ND	-70	101	171
NE	-60	100	160

注意这里使用了别名来对输入的错误进行更正，这只不过是一种暂时的补救方法，虽然这不是永久的解决办法，你是以后（第 21 天：常见的 SQL 错误及其解决方案）会看到如何对数据以及输入进行更正，在那里你将学会如何对错误的数据进行更正。

该查询不只是修正（至少看起来是这样）错误的数据库，而且还创建了一个新列以获得每个记录的最高与最低的差价。

如何你在一个字符型字段中意外地使用了减号，你将会看到如下信息：

```
SQL> SELECT -STATE FROM HILOW;
```

```
ERROR: ORA-01722: invalid number
```

```
No rows selected
```

在不同的解释器中错误的号码可能会不同，但是结果是相同的。

除法 (/)

除法只有一种显而易见的应用，在 PRICE 表中它的应用如下：

输入：

```
SQL> SELECT * FROM PRICE;
```

输出：

ITEM	WHOLESALE
TOMATOES	.34
POTATOES	.51
BANANAS	.67
TURNIPS	.45
CHEESE	.89
APPLES	.23

```
rows selected.
```

在下边的例句中你可以成功地将销售价折半

输入/输出：

```
SQL> SELECT ITEM, WHOLESALE, (WHOLESALE/2) SALEPRICE; 2 FROM PRICE;
```

ITEM	WHOLESALE	SALEPRICE
TOMATOES	.34	.170
POTATOES	.51	.255
BANANAS	.67	.335
TURNIPS	.45	.225
CHEESE	.89	.445
APPLES	.23	.115

```
6 rows selected.
```

在这个 SELECT 语句中除法的作用是显而易见的（只不过将商品半价销售有点太难以理解了）。

乘法 (*)

乘法的运算也非常直观，再以 Price 表为例：

输入：

```
SQL> SELECT * FROM PRICE;
```

输出：

ITEM	WHOLESALE
TOMATOES	.34
POTATOES	.51
BANANAS	.67

TURNIPS	.45
CHEESE	.89
APPLES	.23

6 rows selected.

将表中的价格下调 10% 可以用如下方法来实现:

输入/输出:

```
SQL>SELECT ITEM, WHOLESALE, WHOLESALE * 0.9 NEWPRICE
      FROM PRICE;
```

ITEM	WHOLESALE	NEWPRICE
TOMATOES	.34	.306
POTATOES	.51	.459
BANANAS	.67	.603
TURNIPS	.45	.405
CHEESE	.89	.801
APPLES	.23	.207

6 rows selected.

通过这些操作您可以在 SELECT 语句中进行复杂的运算。

取模 (%)

取模运算将返回一个除法的余数部分，以 REMAINS 表举例如下:

输入:

```
SQL> SELECT * FROM REMAINS;
```

输出:

NUMERATOR	DENOMINATOR
10	5
8	3
23	9
40	17
1024	16
85	34

6 rows selected.

你也可以用 NUMERATOR % DENOMINATOR 的结果来建立一个新列:

输入/输出:

```
SQL> SELECT NUMERATOR, DENOMINATOR, NUMERATOR%DENOMINATOR
      REMAINDER FROM REMAINS;
```

NUMERATOR	DENOMINATOR	REMAINDER
10	5	0
8	3	2
23	9	5
40	17	6
1024	16	0
85	34	17

6 rows selected.

在一些 SQL 解释器中取模运算符为 MOD (见第 4 天: 函数——返回数据的再加工) 下边的语句所得到的结果与上边的语句相同:

```
SQL> SELECT NUMERATOR, DENOMINATOR, MOD (NUMERATOR, DENOMINATOR)
      REMAINDER FROM REMAINS;
```

优先级别

在这一部分的例子中主要讲述在 SELECT 语句中的优先级别，数据库 PRECEDENCE 的内容如下:

SQL> SELECT * FROM PRECEDENCE;

N1	N2	N3	N4
1	2	3	4
13	24	35	46
9	3	23	5
63	2	45	3
7	2	1	4

用 PRECEDENCE 来做如下例子：

输入/输出：

SQL> SELECT

2 N1+N2*N3/N4,

3 (N1+N2)*N3/N4,

4 N1+(N2*N3)/N4

5 FROM PRECEDENCE;

N1+N2*N3/N4	(N1+N2)*N3/N4	N1+(N2*N3)/N4
2.5	2.25	2.5
31.26	28.15	31.26
22.8	55.2	22.8
93	975	93
7.5	2.25	7.5

看到了吗？第一例与最后一例的结果是相同的，如果你把第四列改写成为 $N1+N2*(N3/N4)$ ，那么其结果与上边的例子是相同的。

比较运算：

顾名思义，比较运算就是将两个表达式进行比较并返回三个数值中的一个，True,False,Unknow,请等一下，Unknow，True 和 False 的意义无需说明，但是什么是 Unknow 呢？

为了便于理解什么是 Unknow，你需要理解一下什么是 NULL。在数据库领域内 NULL 的意义就是在一个字段之中没有数据，这与在该字段中数据为零或为空的不是同一个概念，为零或为空是一种特殊的数值，而 NULL 则表示在这个字段之中什么也没有，如果你想进行 Field=9 的比较而 Field 字段是空的，那么比较的结果就会返回 Unknow。由于 Unknow 是一种不正常的状态，所以大多数 SQL 都会置其为无效并提供一种叫 IS NULL 的操作来测试 Null 的存在。

输入：

SQL> SELECT * FROM PRICE;

输出：

ITEM	WHOLESALE
TOMATOES	.34
POTATOES	.51
BANANAS	.67
TURNIPS	.45
CHEESE	.89
APPLES	.23
ORANGES	

请注意 WHOLESALE 字段在 ORANGES 处没有输出，这说明在这里的数值是空的，由于这里的 WHOLESALE 字段的属性为数字，所以空值在这里是显而易见的，但是如果空值是出现在 ITEM 列中，那么要将空值与空白值区分开来就是非常重要的了。

请试着找一下空值：

输入/输出：


```
SQL> SELECT * FROM PRICE WHERE WHOLESALE IS NULL;
      ITEM      WHOLESALE
      ORANGES
```

如你所见到的，WHOLESALE 字段中的 ORANGES 是唯一的一个空值，因为它是不可见的，可是当你使用 “=” 这个比较运算符时会有什么结果呢？

输入/输出：

```
SQL> SELECT * FROM PRICE WHERE WHOLESALE = NULL;
No rows selected
```

分析：

你没有得到任何记录因为比较运算在这里返回的结果为 FALSE，所以使用 WHERE SWHLESALE IS NULL 在这里比使用=更恰当，它将会返回所有存在空值的记录。

这个例子也是对使用 “=” 进行的比较操作进行的完全展示，这之中的 WHERE 子句就不用多说了，下面简要说一下等号。

在今天的早些时候你已经看到了在一些 SQL 解释器中等号可以在 SELECT 子句中用以给搜索字段赋以别名，而在 WHERE 子句中它则用于比较操作，并且它是从多个记录中挑选所需要数值的一种有效手段。试一下：

输入：

```
SQL> SELECT * FROM FRIENDS;
```

输出：

LASTNAME	FIRSTNAME	AREACODE	PHONE	ST	ZIP(邮政编码)
BUNDY	AL	100	555-1111	IL	22333
MEZA	AL	200	555-2222	UK	
MERRICK	BUD	300	555-6666	CO	80212
MAST	JD	381	555-6767	LA	23456
BULHER	FERRIS	345	555-3223	IL	23332

现在让我们来找一下 JD.MAST 的记录信息(在我们的这个表中这很容易，但是你的朋友可能不只这些，也许像这样的记录你有成千上万)。

输入/输出：

```
SQL> SELECT * FROM FRIENDS WHERE FIRSTNAME = 'JD';
      LASTNAME  FIRSTNAME  AREACODE  PHONE  ST  ZIP
      MAST      JD      381      555-6767  LA  23456
```

结果如我们所愿，再试一下：

输入/输出：

```
SQL> SELECT * FROM FRIENDS WHERE FIRSTNAME = 'AL';
      LASTNAME  FIRSTNAME  AREACODE  PHONE  ST  ZIP
      BUNDY     AL      100      555-1111  IL  22333
      MEZA      AL      200      555-2222  UK
```

注：你应该看到在这里 “=” 号返回了多个记录，注意第二个记录的邮政编码（ZIP）是空的，邮政编码是一个字符型字段（你将在第 8 天学习如何创建和组装一个表），这个特殊的空字段表明在字符型字段中空字段与空白字段是不同的。

此外还有一个关于敏感性的问题，试一下：

输入/输出：

```
SQL> SELECT * FROM FRIENDS WHERE FIRSTNAME = 'BUD';
      FIRSTNAME
      BUD
```

1 row selected

再试一下：

输入/输出：

```
SQL> select * from friends where firstname = 'Bud';
```

No rows selected

分析：

尽管 SQL 对大小写是不敏感的，但是数据库中的数据对大小写却是敏感的，大多数公司在存储数据时采用大写以保证数据的一致性，所以你应该永远采用大写或小写来存储数据，大小写的混合使用会对你精确地查找数据造成障碍。

大于与大于等于

大于操作的使用方法如下：

输入：

```
SQL> SELECT * FROM FRIENDS WHERE AREACODE > 300;
```

输出：

LASTNAME	FIRSTNAME	AREACODE	PHONE	ST	ZIP
MAST	JD	381	555-6767	LA	23456
BULHER	FERRIS	345	555-3223	IL	23332

分析：

这个操作将显示所有比区号比 300 大的记录，但是不包括 300，如果要包括 300，应写成如下方式：

输入/输出：

```
SQL> SELECT * FROM FRIENDS WHERE AREACODE>=300;
```

LASTNAME	FIRSTNAME	AREACODE	PHONE	ST	ZIP
MERRICK	BUD	300	555-6666	CO	80212
MAST	JD	381	555-6767	LA	23456
BULHER	FERRIS	345	555-3223	IL	23332

当然，你使用 AREACODE>299 时会得到相同的结果。

注：在这个语句中 300 没有使用引号，对于数字型字段是不需要加引号的。

小于与小于等于

如你所料，它们的使用方法与大于和大于等于操作相同，但结果相反。

输入：

```
SQL> SELECT * FROM FRIENDS WHERE STATE< 'LA';
```

输出：

LASTNAME	FIRSTNAME	AREACODE	PHONE	ST	ZIP
BUNDY	AL	100	555-1111	IL	22333
MERRICK	BUD	300	555-6666	CO	80212
BULHER	FERRIS	345	555-3223	IL	23332

注：为什么 STATE 会变成 ST 呢？这是因为这一列只有两个字符宽，所以结果只会返回两个字符，如果列为 COWS，那么它将会显示成 CO，而 AREACODE、和 PHONE 所在列的列宽大于它们自身的名字，所以它们不会被截去。

分析：

请等一下，你现在知道<在字符字段中的用法了吗？当然知道了，你可以在各种数据类型中进行你想要的比较操作，结果会因数据类型的不同而不同，例如：你在下例中使用小写字符。

输入/输出：

SQL>SELECT * FROM FRIENDS WHERE STATE < 'la';

LASTNAME	FIRSTNAME	AREACODE	PHONE	ST	ZIP
BUNDY	AL	100	555-1111	IL	22333
MEZA	AL	200	555-2222	UK	
MERRICK	BUD	300	555-6666	CO	80212
MAST	JD	381	555-6767	LA	23456
BULHER	FERRIS	345	555-3223	IL	23332

分析:

因为大写的字母代码比小写的小，所以大写的字符总是排在小写字符的前面。这里再说一次，为了保证安全，请在执行前检查大小写情况。

技巧：想知道你所进行操作的结果，那你先要检查一下你的电脑所采用的字符编码集，PC 机解释器使用的是 ASCLL 编码，而其它平台则使用 EBCDIC 编码。

要想在结果中显示 Louisiana，键入：

输入/输出：

SQL> SELECT * FROM FRIENDS WHERE STATE<= 'LA';

LASTNAME	FIRSTNAME	AREACODE	PHONE	ST	ZIP
BUNDY	AL	100	555-1111	IL	22333
MERRICK	BUD	300	555-6666	CO	80212
MAST	JD	381	555-6767	LA	23456
BULHER	FERRIS	345	555-3223	IL	23332

不等号 (<>或!=)

如果你想要查找一些除了确定信息以外的其它信息，那你可以使用不等号，由于 SQL 的解释器不同，它可能写做 "<>" 或 "!="。如果你想找除了 AL 以外的人，你可以写出：

输入：

SQL> SELECT * FROM FRIENDS WHERE FIRSTNAME <> 'AL';

输出：

LASTNAME	FIRSTNAME	AREACODE	PHONE	ST	ZIP
MERRICK	BUD	300	555-6666	CO	80212
MAST	JD	381	555-6767	LA	23456
BULHER	FERRIS	345	555-3223	IL	23332

想找一下不在 California 住的人，可以写成：

输入/输出：

SQL> SELECT * FROM FRIENDS WHERE STATE != 'CA';

LASTNAME	FIRSTNAME	AREACODE	PHONE	ST	ZIP
BUNDY	AL	100	555-1111	IL	22333
MEZA	AL	200	555-2222	UK	
MERRICK	BUD	300	555-6666	CO	80212
MAST	JD	381	555-6767	LA	23456
BULHER	FERRIS	345	555-3223	IL	23332

注意：!=和<>符号都表示不等于。

字符操作：

无论数据的输出是否为有条件输出，你都可以对其中的字符串进行操作，本部分将会讲述两个操作符，LIKE 和||，以及字符串连接的概念。

LIKE

如果你想从数据库中选出一部分数据并把它们添到一个模板中，并且不需要非常精确的匹配。你可以用 "=" 来对每一种可能的情况进行操作，但是这一过程烦琐而又耗时，这时你可以使用 LIKE，如下例：

输入：

SQL>SELECT * FROM PARTS;

输出:

NAME	LOCATION	PARTNUMBER
APPENDIX	MID-STOMACH	1
ADAMSAPPLE	THROAT	2
HEART	CHEST	3
SPINE	BACK	4
ANVIL	EAR	5
KIDNEY	MID-BACK	6

你怎样找出其中有 BACK 的记录呢？粗看一下这里有两个记录，可不幸的是它们有一点差别！

请试一下：

输入/输出:

SQL>SELECT * FROM PARTS WHERE LOCATION LIKE '%BACK%';

NAME	LOCATION	PARTNUMBER
SPINE	BACK	4
KIDNEY	MID-BACK	6

你可能注意到了在这条语句的 LIKE 后边使用了%，在 LIKE 表达式中，%是一种通配符，它表示可能在 BACK 中出现的其它信息，如果你输入如下：

输入:

SQL>SELECT * FROM PARTS WHERE LOCATION LIKE 'BACK%';

你将会检索到所有以 BACK 开头的 LOCATION 记录。

输出:

NAME	LOCATION	PARTNUMBER
SPINE	BACK	4

如果你输入:

输入:

SQL> SELECT * FROM PARTS WHERE NAME LIKE 'A%';

你将会得到所有 NAME 中以 A 开头的记录;

输出:

NAME	LOCATION	PARTNUMBER
APPENDIX	MID-STOMACH	1
ADAMS	APPLE THROAT	2
ANVIL	EAR	5

那么 LIKE 语句是否对大小写敏感呢，请看下边的例子:

输入/输出:

SQL> SELECT * FROM PARTS WHERE NAME LIKE 'a%';

no rows selected

分析:

回答是敏感的，当涉及到数据是时候总是大小写敏感的。

如果你想查找在某一确定的位置上有字符的数据时你应该如何做呢？你可以使用另一个通配符——下划线。

下划线 (_)

输入:

SQL> SELECT * FROM FRIENDS;

输出:

LASTNAME	FIRSTNAME	AREACODE	PHONE	ST	ZIP
----------	-----------	----------	-------	----	-----

BUNDY	AL	100	555-1111	IL	22333
MEZA	AL	200	555-2222	UK	
MERRICK	UD	300	555-6666	CO	80212
MAST	JD	381	555-6767	LA	23456
BULHER	FERRIS	345	555-3223	IL	23332
PERKINS	ALTON	911	555-3116	CA	95633
BOSS	SIR	204	555-2345	CT	95633

如果你想查找所有以 C 开头的州，可以使用如下语句：

输入/输出：

```
SQL> SELECT * FROM FRIENDS WHERE STATE LIKE 'C_';
```

LASTNAME	FIRSTNAME	AREACODE	PHONE	ST	ZIP
MERRICK	BUD	300	555-6666	CO	80212
PERKINS	ALTON	911	555-3116	CA	95633
BOSS	SIR	204	555-2345	CT	95633

也可以在一个语句中使用多个下划线，如：

输入/输出：

```
SQL> SELECT * FROM FRIENDS WHERE PHONE LIKE '555-6_6_';
```

LASTNAME	FIRSTNAME	AREACODE	PHONE	ST	ZIP
MERRICK	BUD	300	555-6666	CO	80212
MAST	JD	381	555-6767	LA	23456

这个语句也可以写成如下形式：

输入/输出：

```
SQL> SELECT * FROM FRIENDS WHERE PHONE LIKE '555-6%';
```

LASTNAME	FIRSTNAME	AREACODE	PHONE	ST	ZIP
MERRICK	BUD	300	555-6666	CO	80212
MAST	JD	381	555-6767	LA	23456

看，它们的结果是一样的，这两个通配符也可以联合起来使用，下边的例子将找出所有的第 2 个字母为 L 的记录：

输入 / 输出：

```
SQL> SELECT * FROM FRIENDS WHERE FIRSTNAME LIKE '_L%';
```

LASTNAME	FIRSTNAME	AREACODE	PHONE	ST	ZIP
BUNDY	AL	100	555-1111	IL	22333
MEZA	AL	200	555-2222	UK	
PERKINS	ALTON	911	555-3116	CA	95633

连接 (||)

|| 可以将两个字符串连接起来，例如：

输入：

```
SQL> SELECT FIRSTNAME || LASTNAME ENTIRENAME FROM FRIENDS;
```

输出：

ENTIRE	NAME
AL	BUNDY
AL	MEZA
BUD	MERRICK
JD	MAST
FERRIS	BULHER
ALTON	PERKINS
SIR	BOSS

7 rows selected.

分析：

请注意这里使用的是 || 而不是 + 号，如果你试图使用 + 号来连接两个字符串的话，对于我们使用的 SQL 解释程序(Personal Oracle7)将会产生如下错误信息：

输入/输出：

```
SQL> SELECT FIRSTNAME + LASTNAME ENTIRENAME FROM FRIENDS;
ERROR:
```

```
ORA-01722: invalid number
```

它试图将两个数字做加法运算，但是它没有在表达式中找到任何数字。

注：有一些解释器也采用加号来连接字符串，请检查一下你的解释器。

对于连接字符串这里有更多的实例。

输入/输出：

```
SQL> SELECT LASTNAME || ', ' || FIRSTNAME NAME FROM FRIENDS;
```

```
NAME
-----
BUNDY      ,  AL
MEZA       ,  AL
MERRICK    ,  BUD
MAST       ,  JD
BULHER     ,  FERRIS
PERKINS    ,  ALTON
BOSS       ,  SIR
```

```
7 rows selected.
```

分析：

这条语句在姓与名之间插入了一个逗号。

注：请注意在姓与名之间的多余的空格，这些空格是数据的一部分，对于确定的数据类型，空格将右填充至达到字段的设定宽度，请检查你的解释器，有关数据类型内容将在第 9 天的《表的创建与维护》中讨论。

至现在为止你已经学完了所有的比较操作符，对于一些问题这种方法非常好，可是如果你想找出所有的名字中的第一个字母为 P，并且他的应有的休假时间已经超过了 3 天的人呢？

逻辑运算：

逻辑运算用于 SQL 的 WHERE 子句中将两个或更多条件组合在一起。

休假的时间总是人们在工作时讨论的热门话题，现在我们来为财务部门设计一个名为渡假（VACATION）的表，内容如下：

输入：

```
SQL> SELECT * FROM VACATION;
```

输出：

LASTNAME	EMPLOYEEENUM	YEARS	LEAVETAKEN
ABLE	101	2	4
BAKER	104	5	23
BLEDSON	107	8	45
BOLIVAR	233	4	80
BOLD	210	15	100
COSTALES	211	10	78

```
6 rows selected.
```

假设你的公司的雇员每年可以有 12 天的休假时间，现在使用你所知道的逻辑运算来实现以下要求，名字是以 B 开头并且他的休假时间已经超过了 50 天的员工。

输入/输出：

```
SQL> SELECT LASTNAME, YEARS * 12 - LEAVETAKEN REMAINING
FROM VACATION WHERE LASTNAME LIKE 'B%' AND
YEARS * 12 - LEAVETAKEN > 50;
```

LASTNAME	REMAINING
BLEDSON	51
BOLD	80

分析：

这个查询语句是你迄今为止学到的最为复杂的语句，SELECT 子句中使用了算术运算符来确定每一个员工还有多少天剩余的假期，标准的算式为 $YEARS * 12 - LEAVETAKEN$ ，而更为清楚的表达方法为 $(YEARS * 12) - LEAVETAKEN$ 。

LIKE 中使用了通配符%来发现所有的以 B 开头的员工，而比较运算的 > 则用来发现所有休假时间超过 50 天的员工。

这里我们使用了逻辑运算符 AND 来使查找到的记录同时满足两个条件（带下划线的）。

AND

AND 只有当两个表达式的值都为真的时候才会返回真，如果任意一个表达式的值不是真，那么结果就会是假的，例如：找一下在你的公司中工作不超过 5 年但是剩余的休假时间超过 20 天的员工

输入：

```
SQL> SELECT LASTNAME FROM VACATION WHERE YEARS <= 5 AND
LEAVETAKEN > 20;
```

输出：

LASTNAME
BAKER
BOLIVAR

如果你想知道在你的公司中工作时间 5 年以上人员和休假时间不足已有假期的 50% 的员工呢？你可以写成下边这样：

输入/输出：

```
SQL> SELECT LASTNAME WORKAHOLICS
2 FROM VACATION
3 WHERE YEARS >= 5
4 AND
5 ((YEARS * 12) - LEAVETAKEN) / (YEARS * 12) < 0.50;
```

WORKAHOLICS
BAKER
BLEDSON

给这些人放假吧，也让我们结束对 AND 的学习。

OR

你也可以使用 OR 来对几个条件进行合并，当其中的任一个条件为真时，其结果就会为真值，为了展示它与 AND 的不同，下面我们用 OR 来换掉上一个例子中的 AND。

输入：

```
SQL> SELECT LASTNAME WORKAHOLICS
2 FROM VACATION
3 WHERE YEARS >= 5
4 OR
5 ((YEARS * 12) - LEAVETAKEN) / (YEARS * 12) >= 0.50;
```

输出：

WORKAHOLICS

ABLE

BAKER

BLED SOE

BOLD

COSTALES

分析：

上例中的结果仍然在其中，但是我们又多个几个记录，这几个记录出现的原因是它们满足我们所提出的条件中的一个，OR 只要记录满足其中的一个条件就会把记录返回。

NOT

顾名思义，它对条件取反，条件为假时结果为真，条件为真时结果为假。

下边的 SELECT 子句将返回所有开头的名字不是 B 的员工。

输入：

```
SQL> SELECT *
  2 FROM VACATION
  3 WHERE LASTNAME NOT LIKE 'B%';
```

输出：

LASTNAME	EMPLOYEEENUM	YEARS	LEAVETAKEN
ABLE	101	2	4
COSTALES	211	10	78

当 NOT 应用于 NULL 时可以使用操作符 IS，让我们再来看一下 PRICES 表中 WHOLESALE 列 ORANGES 记录中的空值。

输入/输出：

```
SQL> SELECT * FROM PRICE;
```

ITEM	WHOLESALE
TOMATOES	.34
POTATOES	.51
BANANAS	.67
TURNIPS	.45
CHEESE	.89
APPLES	.23
ORANGES	

7 rows selected.

想找出所有的非空项，可以写出如下语句：

输入/输出：

```
SQL> SELECT * FROM PRICE WHERE WHOLESALE IS NOT NULL;
```

ITEM	WHOLESALE
TOMATOES	.34
POTATOES	.51
BANANAS	.67
TURNIPS	.45
CHEESE	.89
APPLES	.23

6 rows selected.

集合运算 (SET)

在第一天《介绍 SQL》中我们已经知道了 SQL 是基于集合的理论的，下面这一部分将讨论集合运算。

UNION 与 UNION ALL

UNION 将返回两个查询的结果并去除其中的重复部分，下边有两个值勤人员表：

输入:

```
SQL> SELECT * FROM FOOTBALL;
```

输出:

NAME
ABLE
BRAVO
CHARLIE
DECON
EXITOR
FUBAR
GOOBER

7 rows selected.

输入:

```
SQL> SELECT * FROM SOFTBALL;
```

输出:

NAME
ABLE
BAKER
CHARLIE
DEAN
EXITOR
FALCONER
GOOBER

7 rows selected.

在这两个表中有哪些不重复的人员呢?

输入/输出:

```
SQL> SELECT NAME FROM SOFTBALL
```

```
2 UNION
```

```
3 SELECT NAME FROM FOOTBALL;
```

NAME
ABLE
BAKER
BRAVO
CHARLIE
DEAN
DECON
EXITOR
FALCONER
FUBAR
GOOBER

10 rows selected.

UNION 返回了两个表中的 10 个记录，它们是不重复的，但是两个表中共有多少人呢？（包括重复的人员）

输入/输出:

```
SQL> SELECT NAME FROM SOFTBALL
```

```
2 UNION ALL
```

```
3 SELECT NAME FROM FOOTBALL;
```

NAME
ABLE

```

BAKER
CHARLIE
DEAN
EXITOR
FALCONER
GOOBER
ABLE
BRAVO
CHARLIE
DECON
EXITOR
FUBAR
GOOBER

```

14 rows selected.

分析：

可以看到，UNION ALL 与 UNION 一样对表进行了合并，但是它不去掉重复的记录。可是如果我们想知道都有谁同时在两个表中呢？UNION 无法做到这一点，我们需要学习使用 INTERSECT。

INTERSECT（相交）

INTERSECT 返回两个表中共有的行，看下例，它将返回两个表中有存在的员工：

输入：

```

SQL> SELECT * FROM FOOTBALL
      2 INTERSECT
      3 SELECT * FROM SOFTBALL;

```

输出：

```

NAME
ABLE
CHARLIE
EXITOR
GOOBER

```

分析：

这些记录是两个表中都存在的。

MINUS（相减）

MINUS 返回的记录是存在于第一个表中但不存在于第二个表中的记录。例如：

输入：

```

SQL> SELECT * FROM FOOTBALL MINUS SELECT * FROM SOFTBALL;

```

输出：

```

NAME
BRAVO
DECON
FUBAR

```

上例中显示了三个不在垒球队中的足球队员，如果你把语句的次序颠倒，那么你将得到在垒球队中但不在足球队中的队员。

输入：

```

SQL> SELECT * FROM SOFTBALL MINUS SELECT * FROM FOOTBALL;

```

输出：

```

NAME
BAKER
DEAN

```

FALCONER

从属运算 (IN and BETWEEN)

这两个运算符对你已经做过的例子提供了更快捷的操作，如果你想找一个你在 Colorado, California, 和 Louisiana 的朋友，可以输入：

输入：

```
SQL> SELECT * FROM FRIENDS WHERE STATE= 'CA' OR STATE ='CO' OR STATE = 'LA';
```

输出：

LASTNAME	FIRSTNAME	AREACODE	PHONE	ST	ZIP
MERRICK	BUD	300	555-6666	CO	80212
MAST	JD	381	555-6767	LA	23456
PERKINS	ALTON	911	555-3116	CA	95633

也可以输入：

输入/输出：

```
SQL> SELECT * FROM FRIENDS WHERE STATE IN('CA','CO','LA');
```

LASTNAME	FIRSTNAME	AREACODE	PHONE	ST	ZIP
MERRICK	BUD	300	555-6666	CO	80212
MAST	JD	381	555-6767	LA	23456
PERKINS	ALTON	911	555-3116	CA	95633

分析：

第二个实例比第一个更易读和简捷，我想你一会再用以前的方法来工作了吧？在 IN 中也可以使用数字，例如：

输入/输出：

```
SQL> SELECT *
      2 FROM FRIENDS
      3 WHERE AREACODE IN(100,381,204);
```

LASTNAME	FIRSTNAME	AREACODE	PHONE	ST	ZIP
BUNDY	AL	100	555-1111	IL	22333
MAST	JD	381	555-6767	LA	23456
BOSS	SIR	204	555-2345	CT	95633

如果你想要查找符合某一范围的记录，例如：

输入/输出：

```
SQL> SELECT * FROM PRICE WHERE WHOLESAL > 0.25 AND WHOLESAL < 0.75;
```

ITEM	WHOLESAL
TOMATOES	.34
POTATOES	.51
BANANAS	.67
TURNIPS	.45

或使用 BETWEEN，你可以这样写：

输入/输出： **BETWEEN**

```
SQL> SELECT * FROM PRICE WHERE WHOLESAL BETWEEN 0.25 AND 0.75;
```

ITEM	WHOLESAL
TOMATOES	.34
POTATOES	.51
BANANAS	.67
TURNIPS	.45

看，第二个是不是比第一个更清楚和易读。

注：如果批发价为 0.25 的商品在表中存在，那么它们也将被返回，BETWEEN 操作将包括边界值。

摘要：

在第三天我们学会了使用最基本的 SELECT 子句和 FROM 子句，现在我们已经掌握了最为常用的操作来使数据库返回的结果满足你的要求，你学会了使用算术、比较、字符和逻辑操作，这为你进一步学习 SQL 打下了良好的基础。

问与答：

问：如果我不想使用命令行的 SQL 那么学习这些东西对我有什么用？

答：不论你使用内嵌 SQL 的 COBOL 还是微软的 ODBC，它们所使用的 SQL 结构都是一样的，所以你现在学习的东西将会更有助于你以后的学习。

问：既然 SQL 是一种标准，那为什么又种是让我检查一下自己的解释器呢？

答：我们所使用的是 ANSI1992 标准，但大多数供应商对它进行了修改以使它更适用于自己的数据库，我们是以 ANSI1992 标准为基础的，但在具体使用时要注意它们的不同。

校练场

应用下表的内容来回答下列问题：

LASTNAME	FIRSTNAME	AREACODE	PHONE	ST	ZIP
BUNDY	AL	100	555-1111	IL	22333
MEZA	AL	200	555-2222	UK	
MERRICK	BUD	300	555-6666	CO	80212
MAST	JD	381	555-6767	LA	23456
BULHER	FERRIS	345	555-3223	IL	23332
PERKINS	ALTON	911	555-3116	CA	95633
BOSS	SIR	204	555-2345	CT	95633

写一下查询，返回数据库中所有名字以 M 开头的每一个人。

写一个查询，返回数据库 ST 为 LA 且 FIRSTNAME 以 AL 开头的人。

给你两个表 (PART1 和 PART2)，你如何才能找出两个表中的共有元素。请写出查询。

WHERE a >= 10 AND a <= 30 的更便捷写法是什么，请写出来？

下面的查询将返回什么结果？

```
SELECT FIRSTNAME FROM FRIENDS WHERE FIRSTNAME='AL' AND
LASTNAME='BULHER';
```

练习：

用上边给出的表返回下面的结果。

NAME	ST
AL	FROM IL

输入：

```
SQL> SELECT (FIRSTNAME || 'FROM') NAME, STATE
```

```
2 FROM FRIENDS
```

```
3 WHERE STATE = 'IL' AND
```

```
5 LASTNAME = 'BUNDY';
```

输出：

NAME	ST
AL	FROM IL

仍使用上表，返回以下结果

NAME	PHONE
MERRICK, BUD	300-555-6666
MAST, JD	381-555-6767
BULHER, FERRIS	345-555-3223

第四天：函数：对数据的进一步处理

目标：

在今天我们将学习函数，在 SQL 中的函数可以执行一些诸如对某一些进行汇总或或将一个字符串中的字符转换为大写的操作，在今天结束之际，您将学会以下内容：

- 汇总函数
- 日期与时间函数
- 数学函数
- 字符函数
- 转换函数
- 其它函数

这些函数将大大加强你对这一周的早些时间所学习的 SQL 的基本功能所获得的数据的操作能力，开始的五个汇总函数（COUNT、SUM、AVG、MAX、MIN）是由 ANSI 标准所制定的。大多数的 SQL 解释器都对汇总函数进行了扩充，其中有一些今天会提到，在有些解释器这汇总函数的名称与这里所提到的不一样。

汇总函数

这是一组函数，它们返回的数值是基于一系列的（因为你不会对单个的记录求它的平均数），这一部分的例子将使用 TEAMSTATS 表。

输入：

```
SQL>SELECT * FROM TEAMSTATS;
```

输出：

NAME	POS	AB	HITS	WALKS	SINGLES	DOUBLES	TRIPLES	HR	SO
JONES	1B	145	45	34	31	8	1	5	10
DONKNOW	3B	175	65	23	50	10	1	4	15
WORLEY	LF	157	49	15	35	8	3	3	16
DAVID	OF	187	70	24	48	4	0	17	42
HAMHOCKER	3B	50	12	10	10	2	0	0	13

CASEY	DH	1	0	0	0	0	0	0	1
-------	----	---	---	---	---	---	---	---	---

COUNT

该函数将返回满足 WHERE 条件子句中记录的个数，例如你想知道都有哪一个球员的击球数小于 350，可以这样做：

输入/输出：

```
SQL> SELECT COUNT(*) FROM TEAMSTATS WHERE HITS/AB<=.35;
COUNT(*)
4
```

为了使这段代码更易读，可以使用别名：

输入/输出：

```
SQL>SELECT COUNT(*) NUM_BELOW_350 FROM TEAMSTATS
WHERE HITS/AB<=.35;
NUM_BELOW_350
4
```

如果我们用列名来替换掉括号中的星号时会结果与原来有什么不同呢？试一下：

```
SQL> SELECT COUNT(NAME) NUM_BELOW_350 FROM TEAMSTATS
WHERE HITS/AB <= .35;
NUM_BELOW_350
4
```

结果是一样的，因为你所选择的 NAME 列与 WHERE 子句并不相关，如果你在使用 count 时无 WHERE 子句，那么它将会返回表中的所有记录的个数。

输入/输出：

```
SQL> SELECT COUNT(*) FROM TEAMSTATS;
COUNT(*)
6
```

SUM

SUM 就如同它的本意一样，它返回某一列的所有数值的和，如果想知道队员总打点的总和是多少，试一下：

输入:

```
SQL>SELECT SUM(SINGLES) TOTAL_SINGLES FROM TEAMSTATS;
```

输出:

TOTAL_SINGLES
174

如果想得到多个列的和，可按如下所做:

输入/输出:

```
SQL> SELECT SUM(SINGLES) TOTAL_SINGLES , SUM(DOUBLES)
TOTAL_DOUBLES, SUM(TRIPLES) TOTAL_TRIPLES, SUM(HR), TOTAL_HR
FROM TEAMSTATS;
```

TOTAL_SINGLES	TOTAL_DOUBLES	TOTAL_TRIPLES	TOTAL_HR
174	32	5	29

类似地，如果想找一下所有的点数在 300（包括 300）以上的的队员，则语句如下:

输入/输出:

```
SQL>SELECT SUM(SINGLES) TOTAL_SINGLES, SUM(DOUBLES) TOTAL_DOUBLES,
SUM(TRIPLES) TOTAL_TRIPLES, SUM(HR) TOTAL_HR FROM TEAMSTATS
WHERE HITS/AB >=.300;
```

TOTAL_SINGLES	TOTAL_DOUBLES	TOTAL_TRIPLES	TOTAL_HR
164	30	5	29

想估计一下一个球队的平均中球率:

输入/输出:

```
SQL>SELECT SUM(HITS)/SUM(AB) TEAM_AVERAGE FROM TEAMSTATS;
```

TEAM_AVERAGE
.33706294

SUM 只能处理数字，如果它的处理目标不是数字，你将会收到如下信息:

输入/输出:

```
SQL>SELECT SUM(NAME) FROM TEAMSTATS;
```

ERROR:

ORA-01722: invalid number

no rows selected

该错误信息当然的合理的，因为 NAME 字段是无法进行汇总的。

AVG

AVG 可以返回某一列的平均值，如果你想知道漏球的平均数请看下例：

输入：

```
SQL>SELECT  AVG(SO)  AVE_STRIKE_OUTS  FROM  TEAMSTATS;
```

输出：

AVE_STRIKE_OUTS
16.166667

下边的例子反映了 SUM 与 AVG 的不同之处：

输入/输出：

```
SQL>SELECT  AVG(HITS/AB)  TEAM_AVERAGE  FROM  TEAMSTATS;
```

TEAM_AVERAGE
.26803448

分析：

可是在上一个例子中的打中率是.3370629，这是怎么回事呢？AVG 计算的是打中的次数与总打击次数商的平均值，然而在上一个例子中是对打中次数和打击次数分别求和后在进行相除的。举例来说：A 队员打了 100 杆，中了 50 次，那么他的平均值是 0.5，B 队员打了 1 杆，没打中，他的平均值是 0.0，而 0.0 与 0.5 的平均值是 0.25。如果你按打 101 杆中 50 杆计算，那么结果就会是正确的了，下边的例子将会返回正确的击中率。

输入/输出：

```
SQL>SELECT  AVG(HITS)/AVG(AB)  TEAM_AVERAGE  FROM  TEAMSTATS;
```

TEAM_AVERAGE
.33706294

与 SUM 函数一样，AVG 函数也只能对数字进行计算。

MAX

如果你想知道某一列中的最大值，请使用 MAX。例如：你想知道谁的打点最高。

输入：

```
SQL>SELECT  MAX(HITS)  FROM  TEAMSTATS;
```

输出：

MAX(HITS)
70

你能从这里知道是谁打的最多吗？

输入/输出：

```
SQL>SELECT NAME FROM TEAMSTATS WHERE HITS=MAX (HITS);
```

ERROR at line 3:

ORA-00934: group function is not allowed here

很不幸，你不能。这一信息提示你汇总函数无法在 WHERE 子句中使用。但是请别灰心，在第 7 天的《子查询：深入 SELECT 语句》将引入子查询的概念并会给出知道谁是打点最多人解决方案。

如果把它用的非数字场合会有什么情况出现呢？

输入/输出：

```
SQL>SELECT MAX(NAME) FROM TEAMSTATS;
```

MAX(NAME)
WORLEY

这是一个新现象，MAX 返回了最高的字符串（最大的是 z），所以说 MAX 既可以处理数值也可以处理字符。

MIN

MIN 与 MAX 类似，它返回一列中的最小数值。例如：你想知道打杆的最小值是多少？

输入：

```
SQL>SELECT MIN (AB) FROM TEAMSTATS;
```

输出：

MIN(AB)
1

下列语句将返回名字在字母表中排在最前边的：

输入/输出：

```
SQL>SELECT MIN(NAME) FROM TEAMSTATS;
```

 MIN(NAME)

 CASEY

你可以同时使用 MAX 和 MIN 函数以获得数值的界限，例如：

输入/输出：

```
SQL>SELECT  MIN (AB), MAX (AB)  FROM  TEAMSTATS;
```

MIN(AB)	MAX(AB)
1	187

对于统计函数来说这一信息有时非常有用。

注：我们在今天开始曾说过，这五个函数是由 ANSI 标准所定义的。其余的函数也已经成为了事实上的标准。你可以在所有的 SQL 解释器中找到它们。这里，我们使用的它们在 ORACLE7 中的名字，在其它的解释器中它们的名称可能与这里提到的不同。

VARIANCE

VARIANCE（方差）不是标准中所定义的，但它却是统计领域中的一个至关重要的数值，使用方法如下：

输入：

```
SQL>SELECT  VARIANCE (HITS) FROM  TEAMSTATS;
```

输出：

VARIANCE(HITS)
802.96667

如果我们在将它应用于字符串：

输入/输出：

```
SQL>SELECT  VARIANCE (NAME) FROM  TEAMSTATS;
```

ERROR:

ORA-01722: invalid number

No rows selected

可见，VARIANCE 也是一个只应用于数值对象的函数。

STDDEV

这是最后一个统计函数，STDDEV 返回某一列数值的标准差，它的应用示例如下：

输入：

```
SQL>SELECT  STDDEV (HITS) FROM  TEAMSTATS;
```

输出：

STDDEV(HITS)
28.336666

同样，对字符型列应用该函数时会得到错误信息：

输入/输出：

```
SQL>SELECT  STDDEV(NAME)  FROM  TEAMSTATS;
```

ERROR:

ORA-01722: invalid number

no rows selected

这些统计函数也可以在一个语句中同时使用：

输入/输出：

```
SQL>SELECT  COUNT ( AB), AVG(AB), MIN(AB), MAX(AB), STDDEV(AB),
            VARIANCE(AB), SUM(AB)  FROM  TEAMSTATS;
```

COUNT(AB)	AVG(A B)	MIN(A B)	MAX(AB)	STDDEV(AB)	VARIANCE (AB)	SUM(A B)
6	119.167	1	187	75.589	5712.97	715

当你下次见到比赛结果时，你应该知道了 SQL 正在它的后台工作。

日期/时间函数

我们的生活是由日期和时间来掌握的，大多数的 SQL 解释器都提供了对它进行支持的函数。在这一部分我们使用 PROJECT 表来演示日期和时间函数的用法。

输入：

```
SQL> SELECT  *  FROM  PROJECT;
```

输出：

TASK	STARTDATE	ENDDATE
------	-----------	---------

KICKOFFMTG	01-APR-95	01-APR-95
TECHSURVEY	02-APR-95	01-MAY-95
USERMTGS	15-MAY-95	30-MAY-95
DESIGNWIDGET	01-JUN-95	30-JUN-95
CODEWIDGET	01-JUL-95	02-SEP-95
TESTING	03-SEP-95	17-JAN-96

注：这里的数据类型使用日期型，大多数 SQL 解释器都有日期型，但是在语法的细则上有不同之处。

ADD_MONTHS

该函数的功能是将给定的日期增加一个月，举例来说：由于一些特殊的原因，上述的计划需要推迟两个月，那么可以用下面的方法来重新生成一个日程表。

输入：

```
SQL>SELECT TASK, STARTDATE, ENDDATE
        ORIGINAL_END, ADD_MONTHS(ENDDATE,2) FROM PROJECT;
```

输出：

TASK	STARTDATE	ORIGINAL	ADD_MONTH
KICKOFFMTG	01-APR-95	01-APR-95	01-JUN-95
TECHSURVEY	02-APR-95	01-MAY-95	01-JUL-95
USERMTGS	15-MAY-95	30-MAY-95	30-JUL-95
DESIGNWIDGET	01-JUN-95	30-JUN-95	31-AUG-95
CODEWIDGET	01-JUL-95	02-SEP-95	02-NOV-95
TESTING	03-SEP-95	17-JAN-96	17-MAR-96

尽管这种延误不太可能发生，但是实现日程的变动却是非常容易的，ADD_MONTHS 也可能工作在 SELECT 之外，试着输入：

输入：

```
SQL>SELECT TASK TASKS_SHORTER_THAN_ONE_MONTH
        FROM PROJECT WHERE ADD_MONTHS(STARTDATE, 1)> ENDDATE;
```

结果如下所示：

输出：

TASKS_SHORTER_THAN_ONE_MONTH

KICKOFF MTG
TECH SURVEY
USER MTGS
DESIGN WIDGET

分析：

你将会发现这一部分中的几乎所有的函数都可能工作在不只一个地方，但是，如果没有 TO_CHAR 和 TO_DATE 函数的帮助，ADD_MONTH 就无法在字符或数字类型中工作，这将在今天的晚些时候讨论。

LAST_DAY

LAST_DAY 可以返回指定月份的最后一天。例如：如果你想知道在 ENDDATE 列中的给出日期中月份的最后一天是几号时，你可以输入：

输入：

```
SQL>SELECT ENDDATE, LAST_DAY (ENDDATE) FROM PROJECT;
```

结果如下：

输出：

ENDDATE	LAST_DAY(ENDDATE)
01-APR-95	30-APR-95
01-MAY-95	31-MAY-95
30-MAY-95	31-MAY-95
30-JUN-95	30-JUN-95
02-SEP-95	30-SEP-95
17-JAN-96	31-JAN-96

如果是在闰年的最后一天呢？

输入/输出：

```
SQL>SELECT LAST_DAY( '1-FEB-95' ) NON_LEAP, LAST_DAY( '1-FEB-96' )
```

LEAP

```
FROM PROJECT;
```

NON_LEAP	LEAP
28-FEB-95	29-FEB-96
28-FEB-95	29-FEB-96
28-FEB-95	29-FEB-96

28-FEB-95	29-FEB-96
28-FEB-95	29-FEB-96
28-FEB-95	29-FEB-96

分析：

结果当然是正确的，可是为什么它输出了这么多行呢？这是因为你没有指定任何列或给出一个条件，SQL 引擎对数据库中的每一条记录都应用了这一语句。如果你想去掉这些重复的内容可以这样写：

输入：

```
SQL>SELECT DISTINCT LAST_DAY('1-FEB-95') NON_LEAP, LAST_DAY('1-FEB-96')
LEAP FROM PROJECT;
```

在这句话中我们使用了关键字 DISTINCT（参见第二天的《介绍查询——SELECT 语句的使用》）来得到唯一的结果。

输出：

NON_LEAP	LEAP
28-FEB-95	29-FEB-96

虽然在我的电脑上该函数可以正确地识别出闰年来，但是如果你要将它应用于金融领域，那么请在你的解释器上试一下，看一看它是否支持闰年。

MONTHS_BETWEEN

如果你想知道在给定的两个日期中有多少个月，可以像这样来使用 MONTHS_BETWEEN。

输入：

```
SQL>select task, startdate, enddate, months between(startdate,enddate) duration from project;
```

输出：

TASK	STARTDATE	ENDDATE	DURATION
KICKOFF MTG	01-APR-95	01-APR-95	0
TECH SURVEY	02-APR-95	01-MAY-95	-.9677419
USER MTGS	15-MAY-95	30-MAY-95	-.483871
DESIGN WIDGET	01-JUN-95	30-JUN-95	-.9354839
CODE WIDGET	01-JUL-95	02-SEP-95	-2.032258
TESTING	03-SEP-95	17-JAN-96	-4.451613

请等一下，结果看起来不太对劲，再试一下：

输入/输出：

```
SQL> SELECT TASK, STARTDATE, ENDDATE,
           MONTHS_BETWEEN ( ENDDATE,STARTDATE ) DURATION FROM
PROJECT;
```

TASK	STARTDATE	ENDDATE	DURATION
KICKOFF MTG	01-APR-95	01-APR-95	0
TECH SURVEY	02-APR-95	01-MAY-95	.96774194
USER MTGS	15-MAY-95	30-MAY-95	.48387097
DESIGN WIDGET	01-JUN-95	30-JUN-95	.93548387
CODE WIDGET	01-JUL-95	02-SEP-95	2.0322581
TESTING	03-SEP-95	17-JAN-96	4.4516129

分析：

如你所见，MONTHS_BETWEEN 对于你所给出的月份的次序是敏感的，月份值为负数可能并不是一件坏事，例如：你可以利用负值来判断某一日期是否在另一个日期之前，下例将会显示所有在 1995 年 5 月 19 日以前开始的比赛。

输入：：

```
SQL>SELECT * FROM PROJECT
      WHERE MONTHS_BETWEEN ( '19 MAY 95', STARTDATE)> 0;
```

输出：

TASK	STARTDATE	ENDDATE
KICKOFF MTG	01-APR-95	01-APR-95
TECH SURVEY	02-APR-95	01-MAY-95
USER MTGS	15-MAY-95	30-MAY-95

NEW_TIME

如果你想把时间调整到你所在的时区，你可以使用 NEW_TIME，下边给出了所有的时区

简写	时区	简写	时区
AST or ADT	大西洋标准时间	HST or HDT	阿拉斯加_夏威夷时间
BST or BDT	英国夏令时	MST or MDT	美国山区时间

CST or CDT	美国中央时区	NST	新大陆标准时间
EST or EDT	美国东部时区	PST or PDT	太平洋标准时间
GMT	格伦威治标准时间	YST or YDT	Yukon 标准时间

你可以这样来调节时间：

输入：

```
SQL>SELECT ENDDATE EDT, NEW_TIME (ENDDATE, 'EDT', 'PDT') FROM
PROJECT;
```

输出：

EDT		NEW_TIME(ENDDATE, 'EDT', 'PDT')	
01-APR-95	1200AM	31-MAR-95	0900PM
01-MAY-95	1200AM	30-APR-95	0900PM
30-MAY-95	1200AM	29-MAY-95	0900PM
30-JUN-95	1200AM	29-JUN-95	0900PM
02-SEP-95	1200AM	01-SEP-95	0900PM
17-JAN-96	1200AM	16-JAN-96	0900PM

就像变魔术一样，所有的时间和日期都变成以新的时区标准了。

NEXT_DAY

NEXT_DAY 将返回与指定日期在同一个星期或之后一个星期内的，你所要求的星期天数的确切日期。如果你想知道你所指定的日期的星期五是几号，可以这样做：

输入：

```
SQL>SELECT STARTDATE, NEXT_DAY (STARTDATE, 'FRIDAY') FROM
PROJECT;
```

返回结果如下：

输出：

STARTDATE	NEXT_DAY(STARTDATE, 'FRIDAY')
01-APR-95	07-APR-95
02-APR-95	07-APR-95
15-MAY-95	19-MAY-95
01-JUN-95	02-JUN-95
01-JUL-95	07-JUL-95
03-SEP-95	08-SEP-95

分析：

输出的结果告诉了你距你所指定的日期最近的星期五的日期。

SYSDATE

SYSDATE 将返回系统的日期和时间。

输入：

```
SQL> SELECT DISTINCT SYSDATE FROM PROJECT;
```

输出：

SYSDATE

18-JUN-95 1020PM

如果你想知道在今天你都已经启动了哪些项目的话，你可以输入：

输入/输出：

```
SQL> SELECT * FROM PROJECT WHERE STARTDATE>SYSDATE;
```

TASK	STARTDATE	ENDDATE
CODE WIDGET	01-JUL-95	02-SEP-95
TESTING	03-SEP-95	17-JAN-96

现在，你已经看到了项目在今天所启动的部分。

数学函数

大多数情况下你所检索到的数据在使用时需要用到数学函数，大多数 SQL 的解释器都提供了与这里相类似的一些数学函数，这里的例子使用的表名字叫 NUMBERS，内容如下：

输入：

```
SQL> SELECT * FROM NUMBERS;
```

输出：

A	B	A	B
3.1415	4	-57.667	42
-45	.707	15	55
5	9	-7.2	5.3

ABS

ABS 函数返回给定数字的绝对值，例如：

输入：

```
SQL>SELECT ABS (A) ABSOLUTE_VALUE FROM NUMBERS;
```

输出：

ABSOLUTE_VALUE	ABSOLUTE_VALUE
3.1415	57.667
45	15
5	7.2

CEIL 和 FLOOR

CEIL 返回与给定参数相等或比给定参数^大的最小整数。FLOOR 则正好相反，它返回与给定参数相等或比给定参数小的最大整数。例如：

输入：

```
SQL>SELECT B, CEIL (B) CEILING FROM NUMBERS;
```

输出：

B	CEILING	B	CEILING
4	4	42	42
.707	1	55	55
9	9	5.3	6

输入/输出：

```
SQL>SELECT A, FLOOR (A) FLOOR FROM NUMBERS;
```

A	FLOOR	A	FLOOR
3.1415	3	-57.667	-58
-45	-45	15	15
5	5	-7.2	-8

COS、COSH、SIN、SINH、TAN、TANH

COS、SIN、TAN 函数可以返回给定参数的三角函数值，默认的参数认定为弧度制，如果你没有认识到这一点那你会觉得下例所返回的值是错误：

输入：

```
SQL>SELECT A, COS (A) FROM NUMBERS;
```

输出：

A	COS(A)	A	COS(A)
3.1415	-1	-57.667	.437183
-45	.52532199	15	-.7596879
5	.28366219	-7.2	.60835131

分析：

你可能认为 COS (45) 的返回值应该为 0.707 左右，而不应该是 0.525，如果你想让它按照弧度制来计算，那么你需要将弧度制转换成角度制，由于 360 角度为 2 个弧度，所以，我们可以写成：

输入/输出：

```
SQL>SELECT A, COS (A*0.01745329251994) FROM NUMBERS;
```

A	COS(A*0.01745329251994)
3.1415	.99849724
-45	.70710678
5	.9961947
-57.667	.5348391
15	.96592583
-7.2	.9921147

分析：

这里的将角度转换成弧度后的数值，三角函数也可以像下面所写的那样工作：

输入/输出：

```
SQL>SELECT A, COS (A*0.017453), COSH (A*0.017453) FROM NUMBERS;
```

A	COS(A*0.017453)	COSH(A*0.017453)
3.1415	.99849729	1.0015035
-45	.70711609	1.3245977
5	.99619483	1.00381
-57.667	.53485335	1.5507072
15	.96592696	1.0344645
-7.2	.99211497	1.0079058

输入/输出：

SQL> SELECT A, SIN (A*0.017453), SINH (A*0.017453) FROM NUMBERS;

A	SIN(A*0.017453)	SINH(A*0.017453)
3.1415	.05480113	.05485607
-45	-.7070975	-.8686535
5	.08715429	.0873758
-57.667	-.8449449	-1.185197
15	.25881481	.26479569
-7.2	-.1253311	-.1259926

输入/输出:

SQL> SELECT A, TAN (A*0.017453), TANH (A*0.017453) FROM NUMBERS;

A	TAN(A*0.017453)	TANH(A*0.017453)
3.1415	.05488361	.05477372
-45	-.9999737	-.6557867
5	.08748719	.08704416
-57.667	-1.579769	-.7642948
15	.26794449	.25597369
-7.2	-.1263272	-.1250043

EXP

EXP 将会返回以给定的参数为指数，以 e 为底数的幂值，其应用见下列：

输入:

SQL>SELECT A, EXP (A) FROM NUMBERS;

输出:

A	EXP(A)	A	EXP(A)
3.1415	23.138549	-57.667	9.027E-26
-45	2.863E-20	15	3269017.4
5	148.41316	-7.2	.00074659

LN and LOG

这是两个对数函数，其中 LN 返回给定参数的自然对数，例如：

输入:

SQL>SELECT A, LN(A) FROM NUMBERS;

输出：

ERROR:

ORA-01428: argument '-45' is out of range

这是因为我们忽视了参数的取值范围，负数是没有对数的，改写为：

输入/输出：

SQL>SELECT A, LN (ABS (A)) FROM NUMBERS;

A	LN (ABS (A))	A	LN (ABS (A))
3.1415	1.1447004	-57.667	4.0546851
-45	3.8066625	15	2.7080502
5	1.6094379	-7.2	1.974081

分析：

注意，你可以将 ABS 函数嵌入到 LN 函数中使用，第二个对数函数需要两个参数，其中第二个参数为底数，下例将返回以 10 为底的 B 列的对数值：

输入/输出：

SQL> SELECT B, LOG (B, 10) FROM NUMBERS;

B	LOG(B,10)	B	LOG(B,10)
4	1.660964	42	.61604832
.707	-6.640962	55	.57459287
9	1.0479516	5.3	1.3806894

MOD

其实我们已经见过 MOD 函数了，在第三天的《表达式、条件及操作》就有它，我们知道在 ANSI 标准中规定取模运算的符号为 % 在一些解释器中被函数 MOD 所取代。下例的查询就返回了 A 与 B 相除后的余数：

输入：

SQL>SELECT A, B, MOD (A, B) FROM NUMBERS;

输出：

A	B	MOD(A,B)
3.1415	4	3.1415
-45	.707	-.459
5	9	5

-57.667	42	-15.667
15	55	15
-7.2	5.3	-1.9

POWER

该函数可以返回某一个数对另一个数的幂，在使用幂函数时，第一个参数为底数，第二个指数：

输入：

```
SQL>SELECT A, B, POWER (A, B) FROM NUMBERS;
```

输出：

ERROR:

ORA-01428: argument '-45' is out of range

分析：

粗看时你可能会认为它不允许第一个参数为负数，但这个印象是错误的。因为像-4 这样的数是可以做为底数的，可是，如果第一个参数为负数的话，那么第二个参数就必须是整数（负数是不能开方的），对于这个问题可以使用 CEIL（或 FLOOR）函数：

输入：

```
SQL>SELECT A, CEIL (B), POWER (A, CEIL (B)) FROM NUMBERS;
```

输出：

A	CEIL (B)	POWER(A,CEIL(B))
3.1415	4	97.3976
-45	1	-45
5	9	1953125
-57.667	42	9.098E+73
15	55	4.842E+64
-7.2	6	139314.07

现在就可以有正确的结果了。

SIGN

如果参数的值为负数，那么 SIGN 返回-1，如果参数的值为正数，那么 SIGN 返回 1，如果参数为零，那么 SIGN 也返回零。请看下例：

输入：

```
SQL>SELECT A, SIGN (A) FROM NUMBERS;
```

输出：

A	SIGN (A)	A	SIGN (A)
3.1415	1	-57.667	-1
-45	-1	15	1
5	1	-7.2	-1

你也可以在 SELECT WHERE 子句中使用 SIGN:

输入：

```
SQL>SELECT A FROM NUMBERS WHERE SIGN (A) =1;
```

输出：

A
3.1415
5
15

SQRT

该函数返回参数的平方根，由于负数是不能开平方的，所以我们不能将该函数应用于负数。

输入/输出：

```
SQL>SELECT A, SQRT (A) FROM NUMBERS;
```

ERROR:

ORA-01428: argument '-45' is out of range

但是你可以使用绝对值来解除这一限制：

输入/输出：

```
SQL>SELECT ABS (A), SQRT (ABS (A)) FROM NUMBERS;
```

ABS(A)	SQRT(ABS(A))
3.1415	1.7724277
45	6.7082039
5	2.236068
57.667	7.5938791

15	3.8729833
7.2	2.6832816
0	0

字符函数

许多 SQL 解释器都提供了字符和字符串的处理功能，本部分覆盖了大部分字符串处理函数，这一部分的例子使用 CHARACTERS 表。

输入/输出：

```
SQL> SELECT * FROM CHARACTERS;
```

LASTNAME	FIRSTNAME	M	CODE
PURVIS	KELLY	A	32
TAYLOR	CHUCK	J	67
CHRISTINE	LAURA	C	65
ADAMS	FESTER	M	87
COSTALES	ARMANDO	A	77
KONG	MAJOR	G	52

CHR

该函数返回与所给数值参数等当的字符，返回的字符取决于数据库所依赖的字符集。

例如示例的数据库采用了 ASCLL 字符集，示例数据库的代码列的内容为数字。

输入：

```
SQL>SELECT CODE, CHR (CODE) FROM CHARACTERS;
```

输出：

CODE	CH	CODE	CH
32		87	W
67	C	77	M
65	A	52	4

在数值 32 处显示为空白，因为 32 在 ASCLL 码表中是空格。

CONCAT

我们在第 3 天时学到过一个与这个函数所执行的功能相当的操作，|| 符号表示将两个

字符串连接起来，CONCAT 也是完成这个功能的，使用方法如下：

输入：

```
SQL>SELECT  CONCAT (FIRSTNAME, LASTNAME) "FIRST  AND  LAST  NAMES"
        FROM  CHARACTERS;
```

输出：

FIRST AND LAST NAMES	
KELLY	PURVIS
CHUCK	TAYLOR
LAURA	CHRISTINE
FESTER	ADAMS
ARMANDO	COSTALES
MAJOR	KONG

分析：

当用多个词来做为别名时需对它们使用引号，请检查你的解释器，看看它是否支持别名。

需要注意的是尽管在看起来输出似乎是两列，但实际上它仍是一列，这是因为你所连接的 Firstname 字段的宽度为 15，函数取得了该列中的所有数据，包括其中用以补足宽度的空格。

INITCAP

该函数将参数的第一个字母变为大写，此外其它的字母则转换成小写。

输入：

```
SQL>SELECT  FIRSTNAME  BEFORE, INITCAP (FIRSTNAME) AFTER
        FROM  CHARACTERS;
```

输出：

BEFORE	AFTER
KELLY	Kelly
CHUCK	Chuck
LAURA	Laura
FESTER	Fester
ARMANDO	Armando
MAJOR	Major

LOWER 和 UPPER

如你所料，LOWER 将参数转换为全部小写字母而 UPPER 则把参数全部转换成大写字母。

下例是用 LOWER 函数和一个叫 UPDATE 的函数来把数据库的内容转变为小写字母。

输入：

```
SQL>UPDATE CHARACTERS SET FIRSTNAME='kelly'
      WHERE FIRSTNAME='KELLY';
```

输出：

1 row updated.

输入：

```
SQL>SELECT FIRSTNAME FROM CHARACTERS;
```

输出：

FIRSTNAME	FIRSTNAME
kelly	FESTER
CHUCK	ARMANDO
LAURA	MAJOR

然后，请您再输入：

```
SQL>SELECT FIRSTNAME, UPPER (FIRSTNAME), LOWER (FIRSTNAME)
      FROM CHARACTERS;
```

输出：

FIRSTNAME	UPPER(FIRSTNAME)	LOWER (FIRSTNAME)
kelly	KELLY	kelly
CHUCK	CHUCK	chuck
LAURA	LAURA	laura
FESTER	FESTER	fester
ARMANDO	ARMANDO	armando
MAJOR	MAJOR	major

现在你明白这两个函数的作用了吧！

LPAD 与 RPAD

这两个函数最少需要两个参数，最多需要三个参数。每一个参数是需要处理的字符串，第二个参数是需要将字符串扩充的宽度，第三个参数表示加宽部分用什么字符来做填补，第三个参数的默认值为空格，但也可以是单个的字符或字符串，下面的句子中向字段中加入了五个字符。（该字段的定义宽度为 15）

输入：

```
SQL>SELECT LASTNAME, LPAD (LASTNAME, 20, '*' ) FROM CHARACTERS;
```

输出：

LASTNAME	LPAD (LASTNAME, 20, '*')
PURVIS	*****PURVIS
TAYLOR	*****TAYLOR
CHRISTINE	*****CHRISTINE
ADAMS	*****ADAMS
COSTALES	*****COSTALES
KONG	*****KONG

分析：

为什么只添加了 5 个占位字符呢？不要忘记 LASTNAME 列是 15 个字符宽，在可见字符的右方填充着空格以保证字符的定义宽度，请检查一下你所用的解释器。现在再试一下右扩充：

输入：

```
SQL> SELECT LASTNAME, RPAD (LASTNAME, 20, '*' ) FROM CHARACTERS;
```

输出：

LASTNAME	RPAD(LASTNAME,20,'*')
PURVIS	PURVIS*****
TAYLOR	TAYLOR*****
CHRISTINE	CHRISTINE*****
ADAMS	ADAMS*****
COSTALES	COSTALES*****
KONG	KONG*****

分析：

通过这个操作我们可以清楚地看到空格也是该字段内容的一部分这一事实了，下边的两个

函数正是用于这一情况的。

LTRIM 与 RTRIM

LTRIM 和 RTRIM 至少需要一个参数，最多允许两个参数。第一个参数与 LPAD 和 RPAD 类似，是一个字符串，第二个参数也是一个字符或字符串，默认则是空格。如果第二个参数不是空格的话，那么该函数将会像剪除空格那样剪除所指定的字符，如下例：

输入：

```
SQL> SELECT LASTNAME, RTRIM (LASTNAME) FROM CHARACTERS;
```

输出：

LASTNAME	RTRIM(LASTNAME)
PURVIS	PURVIS
TAYLOR	TAYLOR
CHRISTINE	CHRISTINE
ADAMS	ADAMS
COSTALES	COSTALES
KONG	KONG

你可以用下边的语句来确认字符中的空格已经被剪除了：

输入：

```
SQL> SELECT LASTNAME, RPAD (RTRIM (LASTNAME), 20, '*' ) FROM CHARACTERS;
```

输出：

LASTNAME	RPAD(RTRIM(LASTNAME)
PURVIS	PURVIS*****
TAYLOR	TAYLOR*****
CHRISTINE	CHRISTINE*****
ADAMS	ADAMS*****
COSTALES	COSTALES*****
KONG	KONG*****

输出证明的确已经进行了剪除工作，现在请再试一个 LTRIM：

输入：

```
SQL>SELECT LASTNAME, LTRIM (LASTNAME, 'C') FROM CHARACTERS;
```

输出：

LASTNAME	LTRIM(LASTNAME,
PURVIS	PURVIS
TAYLOR	TAYLOR
CHRISTINE	HRISTINE
ADAMS	ADAMS
COSTALES	OSTALES
KONG	KONG

注意，第三行和第五行的 C 已经没有了。

REPLACE

它的工作就如果它的名字所说的那样，该函数需要三个参数，第一个参数是需要搜索的字符串，第二个参数是搜索的内容，第三个参数则是需要替换成的字符串，如果第三个参数省略或者是 NULL，那么将只执行搜索操作而不会替换任何内容。

输入：

```
SQL> SELECT LASTNAME, REPLACE (LASTNAME, 'ST')  REPLACEMENT FROM
CHARACTERS;
```

输出：

LASTNAME	REPLACEMENT
PURVIS	PURVIS
TAYLOR	TAYLOR
CHRISTINE	CHRIINE
ADAMS	ADAMS
COSTALES	COALES
KONG	KONG

如果存在第三个参数，那么在每一个目标字符串中搜索到的内容将会被由第三个参数所指定的字符串替换，例如：

输入：

```
SQL> SELECT  LASTNAME, REPLACE (LASTNAME, 'ST', '**') REPLACEMENT
FROM  CHARACTERS;
```

输出：

LASTNAME	REPLACEMENT
PURVIS	PURVIS

TAYLOR	TAYLOR
CHRISTINE	CHRI**INE
ADAMS	ADAMS
COSTALES	CO**ALES
KONG	KONG

如果没有第二个参数，那么只有将源字符串返回而不会执行任何操作。

输入：

```
SQL> SELECT  LASTNAME, REPLACE (LASTNAME, NULL)  REPLACEMENT
        FROM  CHARACTERS;
```

输出：

LASTNAME	REPLACEMENT
PURVIS	PURVIS
TAYLOR	TAYLOR
CHRISTINE	CHRISTINE
ADAMS	ADAMS
COSTALES	COSTALES
KONG	KONG

SUBSTR

这个函数有三个参数，允许你将目标字符串的一部份输出。第一个参数为目标字符串，第二个字符串是即将输出的子串的起点，第三个参数是即将输出的子串的长度。

输入：

```
SQL>SELECT  FIRSTNAME, SUBSTR (FIRSTNAME, 2, 3) FROM  CHARACTERS;
```

输出：

FIRSTNAME	SUB
kelly	ell
CHUCK	HUC
LAURA	AUR
FESTER	EST
ARMANDO	RMA
MAJOR	AJO

如果第二个参数为负数，那么将会从源串的尾部开始向前定位至负数的绝对值的位置，例如：

输入：

```
SQL> SELECT FIRSTNAME, SUBSTR (FIRSTNAME, -13, 2) FROM CHARACTERS;
```

输出：

FIRSTNAME	SU
kelly	ll
CHUCK	UC
LAURA	UR
FESTER	ST
ARMANDO	MA
MAJOR	JO

分析：

切记 FIRSTNAME 字段的宽度为 15，这也就是为什么参数为-13 时会从第三个开始的原因。因为从 15 算起向前算第 13 个字符正好是第 3 个字符。如果没有第三个参数，将会输出字符串余下的部分：

输入：

```
SQL> SELECT FIRSTNAME, SUBSTR (FIRSTNAME, 3) FROM CHARACTERS;
```

输出：

FIRSTNAME	SUBSTR(FIRSTN
kelly	lly
CHUCK	UCK
LAURA	URA
FESTER	STER
ARMANDO	MANDO
MAJOR	JOR

看，是不是将字符串余下的部分返回了。

现在再来看一个例子：

输入：

```
SQL> SELECT * FROM SSN_TABLE;
```

输出：

SSN_____
300541117
301457111
459789998

如果直接阅读上边的结果是比较困难的，比较好的解决办法是使用下划线，请先想一下下边语句的输出情况：

输入：

```
SQL> SELECT SUBSTR (SSN, 1, 3) || '-' || SUBSTR (SSN, 4, 2) || '-' || SUBSTR (SSN,
6, 4)
SSN FROM SSN_TABLE;
```

输出：

```
SSN_____
300-54-1117
301-45-7111
459-78-9998
```

注：这在当数字特别大（例如：1, 343, 178, 128）需要用逗号分隔时以及区位号码或电话号码需要下划线分隔时特别有效。

这是 SUBSTR 的另一个非常有用的功能，倘若你需要打印一个报表而其中一些列的宽度超过了 50 个字符时，你可以使用 SUBSTR 来减小列宽以使它更接近数据的真实宽度，请看一个下面的这两个例子：

输入：

```
SQL> SELECT NAME, JOB, DEPARTMENT FROM JOB_TBL;
```

输出：

```
NAME_____
JOB_____DEPARTMENT_____
ALVIN SMITH
VICEPRESIDENT MARKETING
1 ROW SELECTED.
```

分析：

注意，这几列已经换行显示了，这例得阅读变行非常困难，现在试一下下边的 SELECT 语句：

输入：

```
SQL> SELECT SUBSTR(NAME, 1,15) NAME, SUBSTR(JOB,1,15) JOB,
```

DEPARTMENT

2 FROM JOB_TBL;

输出：

NAME_____JOB_____DEPARTMENT_____

ALVIN SMITH VICEPRESIDENT MARKETING

是不是变得好多了！

TRANSLATE

这一函数有三个参数：目标字符串、源字符串和目的字符串，在目标字符串与源字符串中均出现的字符将会被替换成对应的目的字符串的字符。

输入：

```
SQL> SELECT FIRSTNAME, TRANSLATE(FIRSTNAME
      '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
      'NNNNNNNNNNAAAAAAAAAAAAAAAAAAAAAAAAAAAA)          FROM
CHARACTERS;
```

输出：

FIRSTNAME	TRANSLATE(FIRST
kelly	kelly
CHUCK	AAAAA
LAURA	AAAAA
FESTER	AAAAAA
ARMANDO	AAAAAAA
MAJOR	AAAAA

6 rows selected.

注意，这个函数对大小写是敏感的。

INSTR

如果需要知道在一个字符串中满足特定的内容的位置可以使用 INSTR，它的第一个参数是目标字符串，第二个参数是匹配的内容，第三和第四个参数是数字，用以指定开始搜

索的起点以及指出第几个满足条件的将会被返回。下例将从字符串的第二个字符开始、搜索并返回第一个以 O 开头的字符的位置：

输入：

```
SQL>SELECT LASTNAME, INSTR (LASTNAME, 'O', 2, 1) FROM CHARACTERS;
```

输出：

LASTNAME	INSTR(LASTNAME,'O',2,1)
PURVIS	0
TAYLOR	5
CHRISTINE	0
ADAMS	0
COSTALES	2
KONG	2
6 rows selected	

分析：

默认第三个与第四个参数的数值均为 1，如果第三个数值为负数，那么将会从后向前搜索。

LENGTH

LENGTH 将返回指定字符串的长度，例如：

输入：

```
SQL>SELECT FIRSTNAME, LENGTH (RTRIM (FIRSTNAME)) FROM CHARACTERS;
```

输出：

FIRSTNAME	LENGTH(RTRIM(FIRSTNAME))
kelly	5
CHUCK	5
LAURA	5
FESTER	6
ARMANDO	7
MAJOR	5

注意：这里使用了函数 RTRIM，否则 LENGTH 将全部返回 15。

转换函数

转换函数有三个，可以使你方便地将数据从一种类型变换为另一种类型，本节的示例

使用表 CONVERSIONS。

输入：

```
SQL> SELECT * FROM CONVERSIONS;
```

输出：

NAME	TESTNUM
40	95
13	23
74	68

NAME 列为字符串，该列的宽度为 15，TESTNUM 列为数字。

TO_CHAR

该函数的最初功能是将一个数字转换为字符型，不同的解释器可能会使用它来转换其它的数据类型，例如日期型转换为字符型或者是拥有更多的参数。下例展示了该函数的基本功能：

输入：

```
SQL> SELECT TESTNUM, TO_CHAR (TESTNUM) FROM CONVERT;
```

输出：

TESTNUM	TO_CHAR(TESTNUM)
95	95
23	23
68	68

稍安勿躁，下例可以证明它确实已经将 TESTNUM 转换为字符型了：

输入：

```
SQL> SELECT TESTNUM, LENGTH (TO_CHAR (TESTNUM)) FROM CONVERT;
```

输出：

TESTNUM	LENGTH(TO_CHAR(TESTNUM))
95	2
23	2
68	2

分析：

如果对数字使用 LENGTH 函数将会返回错误，注意 TO_CHAR 与在先前进过的 CHR 不同，

CHR 返回字符集中给定数字位置的一个字符或符号。

TO_NUMBER

该函数与 TO_CHAR 函数相对应，显而易见，它是将一个字符串型数字转换为数值型，例如：

输入：

```
SQL> SELECT NAME, TESTNUM, TESTNUM*TO_NUMBER (NAME) FROM CONVERT;
```

输出：

NAME	TESTNUM	TESTNUM*TO_NUMBER(NAME)
40	95	3800
13	23	299
74	68	5032

分析：

如果该函数没有将 NAME 转换为数值的话将会返回一个错误信息。

其它函数

这里有三个函数可能对你是有用处的。

GREATEST 与 LEAST

这两个函数将返回几个表达式中最大的和最小的，例如：

输入：

```
SQL> SELECT  GREATEST ('ALPHA', 'BRAVO', 'FOXTROT', 'DELTA')
          FROM  CONVERT;
```

输出：

```
GREATEST
FOXTROT
FOXTROT
FOXTROT
```

分析：

注意 GREATEST 将会返回在字母表中最靠后的字符开头的字符串，虽然看起来似乎没

有必要使用 FROM 子句，可是如果 FROM 子句没有的话，你将会收到一个错误信息。每一个 SELECT 语句都需要 FROM 子句。由于给定的表有三行，所以结果返回了三个。

输入：

```
SQL> SELECT LEAST (34, 567, 3, 45, 1090) FROM CONVERT;
```

输出：

LEAST(34, 567, 3, 45, 1090)
3
3
3

就像你看到的那样，这两个函数也可以对数字进行处理。

USER

该函数返回当前使用数据库的用户的名字。

输入：

```
SQL> SELECT USER FROM CONVERT;
```

输出：

USER
PERKINS
PERKINS
PERKINS

只有我一个人在使用这个数据库。看，SELECT 又一次对表中的每一行都返回了结果，该函数与在今天早些时候提及的日期函数类似，甚至尽管 USER 不是表中确实存在的列，但 SELECT 仍然检索了表中的每一行。

总结

好长的一天啊，我们一共学习了 47 个函数。你无需记住每一个函数，只需要知道它们的大致类型（算术函数、日期/时间函数、字符函数、转换函数、其它函数），而当你写一个查询的时候你有一个明确的目标就够了。

问与答

问：为什么在 ANSI 标准中定义的函数这么少，而不同的解释器又都定义了这么多的函数？

答：ANSI 标准是一个非常宽松的标准，而且如果对所有的解释器生产厂家来说过多的限制会使其走向破产的道路。另一方面，如果 A 公司对 SQL 补充了一个用于统计的函数包而销路特别好的话，那么 B 公司和 C 公司一定也会跟着做的。

问：我认为你对 SQL 的介绍太简单了，我的工作时应如何去使用这些函数呢？

答：这个问题与一个都三角的教师所提出的问题类似，事实上我也不知道如何去求一个等腰三角形的面积。所以，我的回答是：根据你的职业而定，你的工作中需要用到哪些函数你就去使用它，而其它的对你来说则是没有必要掌握的。这一观点也适用于你的查询。

校练场

1、哪个函数是用来将给定字符串的第一个字母变成大写而把其它的字符变成小写的？

2、哪此函数的功能就如同它的名字含义一样？

3、下边的查询将如何工作？

```
SQL> SELECT COUNT(LASTNAME) FROM CHARACTERS;
```

4、下边的查询是干什么的？

```
SQL> SELECT SUM(LASTNAME) FROM CHARACTERS;
```

5、哪个函数可以将 FIRSTNAME 列与 LASTNAME 列合并到一起？

6、在下边的查询中，6 是什么意思？

输入：

```
SQL> SELECT COUNT(*) FROM TEAMSTATS;
```

输出：

```
COUNT(*)
```

```
6
```

7、下列语句将输出什么？

```
SQL> SELECT SUBSTR (LASTNAME,1,5) FROM NAME_TBL;
```

练习

- 1、用今天的 TEAMSTARTS 表来写一个查询，用来显示谁的中球率低于 0.25（中球率的计算方法为 hits/ab）
- 2、用今天的 CHARACTERS 表来写一个查询，要求返回下边的结果

INITIALS_____CODE

K.A.P. 32

1 row selected.

第五天：SQL 中的子句

目标：

今天的主题是子句——它不是你在渡假时的赠品，而是你所学习的 SELECT 语句的一个组成部分，在今天结束以后我们将学会以下子句：

- WHERE
- STARTING WITH
- ORDER BY
- GROUP BY
- HAVING

为了对这些子句有一个大致的印象，请看一下 SELECT 语句的通用语法表达式：

语法：

```
SELECT [DISTINCT | ALL] { *
                        | { [schema.]{table | view | snapshot}.*
                        | expr } [ [AS] c_alias ]
                        [, { [schema.]{table | view | snapshot}.*
                        | expr } [ [AS] c_alias ] ] ... }
FROM [schema.]{table | view | snapshot}[@dblink] [t_alias]
    [, [schema.]{table | view | snapshot}[@dblink] [t_alias] ] ...
[WHERE condition ]
[GROUP BY expr [, expr] ... [HAVING condition] ]
[ {UNION | UNION ALL | INTERSECT | MINUS} SELECT command ]
[ORDER BY {expr|position} [ASC | DESC]
    [, {expr|position} [ASC | DESC]] ...]
```

注：根据我对 SQL 的经验，ANSI 标准确实只是 ANSI 的“建议”。上述的语法格式在大多数的 SQL 引擎下都能够工作，但是你可以发现它们之间存在着一些差别。

你无需对这些复杂的语法花费太多的精力，因为许多人发现语法表比关于新应用的示例更容易让人困惑，本书采用简单的例子展现其特点，不过，如果我们在今天看一下有关它的语法表述会使我们对今天的学习更容易理解。

不要对语法的精确内容太担心，对于不同的解释器它们是不同的。所以，我们应该把精力放到关系上，在最前边的是 SELECT 语句，我们在前几天已经多次使用过了，SELECT 之后应该是 FROM，在每次输入 SELECT 语句时它也应该输入（明天我们将学习 FROM 语句的新用法）之后是 WHERE，GROUP BY，HAVING 和 ORDER BY（其余的子句，包括 UNION，UNION ALL，INTERSECT 和 MINUS 在表中已经在第 3 天时讲过了）每一个子句的在数据的选择和操作时都扮演着它的重要角色。

注：在今天的例子中我们使用两种 SQL 解释器，一种仍然是 SQL 的命令行形式（它属于 Personal Oracle7），而另外一种则不是（它是 BORLAND 公司的 ISQL）。你可以发现输出的结果会随着解释器的不同而不同。

WHERE 子句

仅使用 SELECT 和 FROM 子句，你会受到表中的每一行数据均返回的限制。例如，只在 CHECKS 表中使用这两个关键字，你将会得到表中的全部行（共 7 行）。

INPUT:

```
SQL>SELECT * FROM CHECKS;
```

OUTPUT:

CHECK#	PAYEE	AMOUNT	REMARKS
1	Ma Bell	150	Have sons next time
2	Reading R.R.	245.34	Train to Chicago
3	Ma Bell	200.32	Cellular Phone
4	Local Utilities	98	Gas
5	Joes Stale\$ Dent	150	Groceries
16	Cash 25	Wild	Night Out
17	Joans Gas	25.1	Gas

使用 WHERE 子句将会使你更具有选择性，要想找到你所填写的支票中所有超过 100 元的，你可以这样写：

INPUT:

```
SQL>SELECT * FROM CHECKS WHERE AMOUNT>100;
```

WHERE 子句只返回了符合条件的四条记录。

OUTPUT:

CHECK#	PAYEE	AMOUNT	REMARKS
1	Ma Bell	150	Have sons next time
2	Reading R.R	245.34	Train to Chicago
3	Ma Bell	200.32	Cellular Phone
5	Joes Stale \$ Dent	150	Groceries

使用 WHERE 也可以解决一些其它的难题，下表给出了姓名和位置，你可以提出这样的问题——Waldo 住在哪儿？

INPUT:

```
SQL>SELECT * FROM PUZZLE;
```

OUTPUT:

NAME	LOCATION
TYLER	BACKYARD
MAJOR	KITCHEN
SPEEDY	LIVING ROOM
WALDO	GARAGE
LADDIE	UTILITY CLOSET
ARNOLD	TV ROOM

INPUT:

```
SQL>SELECT LOCATION AS "WHERE'S WALDO?" FROM PUZZLE
      WHERE NAME = 'WALDO';
```

OUTPUT:

WHERE'S WALDO?

GARAGE

好了，我答应以后不再写像这样粗俗的语句了（我已经把它们收录于《SQL BATHROOM HUMOR》——这是一本每个人都想得到的书）。不过，这个查询显示出了在 WHERE 中用于条件的列并没有在 SELECT 语句中出现，本例中你所选择的是 LOCATION 列而条件列是 NAME，这是完全合法的。同时，我们也应该注意到 SELECT 语句中我们使用了 AS，它是一个可以选择的参数，用以指定 LOCATION 的别名，你以后将不会看到 AS，因为它是多余的（ACCESS 中则不可省略——译者）。在大多数 SQL 解释器中我们只需输入：

INPUT:

```
SQL>SELECT  LOCATION  "WHERE'S  WALDO?"  FROM  PUZZLE  WHERE
NAME='WALDO';
```

这里没有使用 AS，但它的结果与上例是完全一样的。

WHERE 是使用频度仅次于 SELECT 和 FROM 的语句。

STARTING WITH 子句

STARTING WITH 子句附加于 WHERE 子句上，它的作用与 LIKE (exp%) 相似，试比较下边的两个查询：

INPUT:

```
SELECT  PAYEE, AMOUNT, REMARKS  FROM  CHECKS  WHERE  PAYEE
LIKE('Ca%');
```

OUTPUT:

PAYEE	AMOUNT	REMARKS
Cash	25	Wild Night Out
Cash	60	Trip to Boston
Cash	34	Trip to Dayton

再看看下边的查询：

INPUT:

```
SELECT  PAYEE, AMOUNT, REMARKS  FROM  CHECKS  WHERE  PAYEE  STARTING
WITH('Ca');
```

OUTPUT:

PAYEE	AMOUNT	REMARKS
Cash	25	Wild Night Out
Cash	60	Trip to Boston
Cash	34	Trip to Dayton

结果是相同的，你甚至可以同时使用它们，例如：

INPUT:

```
SELECT  PAYEE, AMOUNT, REMARKS  FROM  CHECKS
WHERE  PAYEE  STARTING  WITH('Ca')  OR  REMARKS  LIKE  'G%';
```

OUTPUT:

PAYEE	AMOUNT	REMARKS
Local Utilities	98	Gas
Joes Stale \$ Dent	150	Groceries
Cash	25	Wild Night Out
Joans Gas	25.1	Gas
Cash	60	Trip to Boston
Cash	34	Trip to Dayton
Joans Gas	15.75	Gas

警告：STARTING WITH 为许多 SQL 解释器所支持，在你喜欢上它之前请先检查你的 SQL 解释器是否支持它！

ORDER BY 子句

在有些时候你可能会希望查询输出的结果按一定的排序规则来显示。可是，正如你所知道的，SELECT FROM 语句只会给你一个列表，除非你已经定义了关键字（见第 10 天：《创建视图和索引》），否则你查询的结果是依据它们在输入时的次序排列的。请看下表：

INPUT:

SQL>SELECT * FROM CHECKS;

OUTPUT:

CHECK#	PAYEE	AMOUNT	REMARKS
1	Ma Bell	150	Have sons next time
2	Reading R.R.	245.34	Train to Chicago
3	Ma Bell	200.32	Cellular Phone
4	Local Utilities	98	Gas
5	Joes Stale \$ Dent	150	Groceries
16	Cash	25	Wild Night Out
17	Joans Gas	25.1	Gas
9	Abes Cleaners	24.35	X-Tra Starch
20	Abes Cleaners	10.5	All Dry Clean
8	Cash	60	Trip to Boston
21	Cash	34	Trip to Dayton

分析：

请相信我，数据输出的情况的确是按照它们被输入的先后次序排序的，在第 8 天的《数据操作》中我们将知道如何使用 INSERT 来创建一个新的表，那时你就可以试一下，看看

数据是不是如你所猜想的那样来排序。

ORDER BY 子句为你提供了对输出的结果进行排序的方法，例如，将记录按 CHECKS 号进行排序，语句如下：

INPUT:

```
SQL>SELECT * FROM CHECKS ORDER BY CHECK#;
```

OUTPUT:

CHECK#	PAYEE	AMOUNT	REMARKS
1	MaBell	150	Have next sonstime
2	Reading R.R.	245.34	Train to Chicago
3	Ma Bell	200.32	Cellular Phone
4	Local Utilities	98	Gas
5	Joes Stale	\$	Dent 150 Groceries
8	Cash	60	Trip to Boston
9	Abes Claeners	24.35	X-Tra Starch
16	Cash	25	Wild Night Out
17	Joans Gas	25.1	Gas
20	Abes Cleaners	10.5	All Dry Clean
21	Cash	34	Trip to Dayton

现在数据已经按照你的要求进行排序而不是按照它们被输入的次序进行排序了，下边的例子则表明，BY 是 ORDER 不可缺少的组成部分：

INPUT/OUTPUT:

```
SQL> SELECT * FROM CHECKS ORDER CHECK#;
```

ERROR at line 1:

ORA-00924: missing BY keyword

如果你想让数据按降序排列，也就是说数值最大的排在最前边。那么非常幸运，下例中 PAYEEs 表中的 PAYEEs 列就是按降序排列的。

INPUT/OUTPUT:

```
SQL>SELECT * FROM CHECKS ORDER BY PAYEE DESC;
```

CHECK#	PAYEE	AMOUNT	REMARKS
2	Reading R.R.	245.34	Train to Chicago
1	Ma Bell	150	Have sons next time
3	Ma Bell	200.32	Cellular Phone
4	Local Utilities	98	Gas

5	Joes Stale \$ Dent	150	Groceries
17	Joans Gas	25.1	Gas
16	Cash	25	Wild Night Out
8	Cash	60	Trip to Boston
21	Cash	34	Trip to Dayton
9	Abes Cleaners	24.35	X-Tra Starch
20	Abes Cleaners	10.5	All Dry Clean

在 ORDER BY 后边的 DESC 表示用降序排列来代替默认的升序排列，下例则出现了很少使用的关键字 ASC，表示要按升序进行排列：

INPUT:

SQL>SELECT PAYEE, AMOUNT FROM CHECKS ORDER BY CHECK# ASC;

OUTPUT:

PAYEE	AMOUNT
Ma Bell	150
Reading R.R	245.34
Ma Bell	200.32
Local Utilities	98
Joes Stale \$ Dent	150
Cash	60
Abes Cleaners	24.35
Cash	25
Joans Gas	25.1
Abes Cleaners	10.5
Cash	34

输出的结果与最初没有使用 ASC 时是一样的，这是因为 ASC 是默认的选项。本例也表明了用于排序的字段并不一定要出现在 SELECT 子句中。尽管我们选择的是 PAYEE 和 AMOUNT，但是排序却是按 CHECKS 进行的。

ORDER BY 可以使用多个字段，下例是按 PAYEE 和 REMARKS 进行排序的。

INPUT:

SQL>SELECT * FROM CHECKS ORDER BY PAYEE, REMARKS;

OUTPUT:

CHECK#	PAYEE	AMOUN	REMARKS
		T	
20	Abes Cleaners	10.5	All Dry Clean

9	Abes	Cleaners	24.35	X-Tra	Starch
8	Cash		60	Tripto	Boston
21	Cash		34	Tripto	Dayton
16	Cash		25	Wild	Night Out
17	Joans	Gas	25.1	Gas	
5	Joes	Stale\$ Dent	150	Groceries	
4	Local	Utilities	98	Gas	
3	Ma	Bell	200.32	Cellular	Phone
1	Ma	Bell	150	Havesonsnexttime	
2	Reading	R.R.	245.34	Trainto	Chicago

分析：

注意在排序之前 CASH 在表中的输入次序，CHECK 号依次为 16, 8, 21。在 ORDER BY 子句中加入了 REMARK 字段后结果是按照 REMARK 的字母顺序进行排序，在 ORDER BY 中的列的次序对排序的结果会有影响吗？试着掉换一下 PAYEE 和 REMARKS 的次序：

INPUT:

```
SQL>SELECT * FROM CHECKS ORDER BY REMARKS, PAYEE;
```

OUTPUT:

CHECK#	PAYEE	AMOUNT	REMARKS
20	Abes Cleaners	10.5	All Dry Clean
3	Ma Bell	200.32	Cellular Phone
17	Joans Gas	25.1	Gas
4	Local Utilities	98	Gas
5	Joes Stale \$ Dent	150	Groceries
1	Ma Bell	150	Have sons next time
2	Reading R.R.	.245.34	Train to Chicago
8	Cash	60	Trip to Boston
21	Cash	34	Trip to Dayton
16	Cash	25	Wild Night Out
9	Abes Cleaners	24.35	X-Tra Starch

分析：

你大概已经猜出来了，结果是完全不同的。下例显示了如何将一列按字母的正顺排列而把第二列按字母的逆序进行排列。

INPUT/OUTPUT:

```
SQL> SELECT * FROM CHECKS ORDER BY PAYEE ASC, REMARKS DESC;
```


CHECK#	PAYEE	AMOUNT	REMARKS
9	Abes Cleaners	24.35	X-Tra Starch
20	Abes Cleaners	10.5	All Dry Clean
16	Cash	25	Wild Night Out
21	Cash	34	Trip to Dayton
8	Cash	60	Trip to Boston
17	Joans Gas	25.1	Gas
5	Joes Stale \$ Dent	150	Groceries
4	Local Utilities	98	Gas
1	Ma Bell	150	Have sons next time
3	Ma Bell	200.32	Cellular Phone
2	Reading R.R	.245.34	Train to Chicago

分析:

在这个例子中 PAYEE 按正序排列而 REMARK 按逆序排列，请注意 CASH 中的 REMARK 是怎样相对于 PAYEE 排序的。

技巧：假如你已经知道了你想要进行排序的列是表中的第一列的话，那么你可以用 ORDER BY 1 来代替输入列的名字，见下例。

INPUT/OUTPUT:

SQL> SELECT * FROM CHECKS ORDER BY 1;

CHECK#	PAYEE	AMOUNT	REMARKS
1	Ma Bell	150	Have sons next time
2	Reading R.R	.245.34	Train to Chicago
3	Ma Bell	200.32	Cellular Phone
4	Local Utilities	98	Gas
5	Joes Stale \$ Dent	150	Groceries
8	Cash	60	Trip to Boston
9	Abes Cleaners	24.35	X-Tra Starch
16	Cash	25	Wild Night Out
17	Joans Gas	25.1	Gas
20	Abes Cleaners	10.5	All Dry Clean
21	Cash	34	Trip to Dayton

分析:

它的结果与你在今天的早些时候写下的这个语句的结果是一样的。

SELECT * FROM CHECKS ORDER BY CHECK#;

GROUP BY 子句

在第三天时我们学习了汇总类函数 (COUNT, SUM, AVG, MIN, MAX)，如果你想看一下支出的总费用你可以用如下语句：

INPUT:

```
SELECT * FROM CHECKS;
```

下表是修改以后的 CHECKS 表:

CHECKNUM	PAYEE	AMOUNT	REMARKS
1	Ma Bell	150	Have sons next time
2	Reading R.R	.245.34	Train to Chicago
3	Ma Bell	200.33	Cellular Phone
4	Local Utilities	98	Gas
5	Joes Stale \$ Dent	150	Groceries
16	Cash	25	Wild Night Out
17	Joans Gas	25.1	Gas
9	Abes Cleaners	24.35	X-Tra Starch
20	Abes Cleaners	10.5	All Dry Clean
8	Cash	60	Trip to Boston
21	Cash	34	Trip to Dayton
30	Local Utilities	87.5	Water
31	Local Utilities	34	Sewer
25	Joans Gas	15.75	Gas

你会输入如下语句:

INPUT/OUTPUT:

```
SELECT SUM (AMOUNT) FROM CHECKS;
```

```
SUM
```

```
1159.87
```

分析:

这条语句返回了对 AMOUNT 列的合计结果，可是如果你想知道的是对每一个 PAYEE 花了多少钱时又该怎么办呢？使用 GROUP BY 语句可以帮助你。对本例它的使用方法如下：

INPUT/OUTPUT:

```
SELECT PAYEE, SUM (AMOUNT) FROM CHECKS GROUP BY PAYEE;
```

```
PAYEE          SUM
```

Abes Cleaners	34.849998
Cash	119
Joans Gas	40.849998
Joes Stale \$ Dent	150
Local Utilities	219.5
Ma Bell	350.33002
Reading R.R	.245.34

SELECT 子句有一个正常的列，之后是一个汇总函数，如果它的后边只有 FROM CHECKS 子句的话，那么你将会看到：

INPUT/OUTPUT:

```
SELECT PAYEE, SUM (AMOUNT) FROM CHECKS;
```

Dynamic SQL Error

-SQL error code = -104

-invalid column reference

分析：

该信息表明 SQL 无法把正常的列和汇总函数结合在一起，这时就需要 GROUP BY 子句，它可以对 SELECT 的结果进行分组后在应用汇总函数，查询 SELECT * FROM CHECKS 返回了 14 行，而 SELECT PAYEE, SUM (AMOUNT) FROM CHECKS GROUP BY PAYEE; 则把返回的 14 行分成了 7 组，然后对每组应用了汇总函数。

INPUT/OUTPUT:

```
SELECT PAYEE, SUM (AMOUNT), COUNT (PAYEE) FROM CHECKS
```

```
GROUP BY PAYEE;
```

PAYEE	SUM	COUNT
Abes Cleaners	34.849998	2
Cash	119	3
Joans Gas	40.849998	2
Joes Stale \$ Dent	150	1
Local Utilities	219.5	3
Ma Bell	350.33002	2
Reading R.R	.245.34	1

分析：

SQL 现在越来越变得有用了，在上一个例子中，我们只是应用 GROUP BY 对数据结果进行了唯一的分组，注意结果是按 PAYEE 排序的。GROUP BY 也可以像 ORDER BY 那

样工作。如果我们对多个列进行分组会有什么结果呢，请看：

INPUT/OUTPUT：

```
SELECT PAYEE, SUM (AMOUNT), COUNT (PAYEE) FROM CHECKS
```

```
GROUP BY PAYEE, REMARKS;
```

PAYEE	SUM	COUNT
Abes Cleaners	10.5	1
Abes Cleaners	24.35	1
Cash	60	1
Cash	34	1
Cash	25	1
Joans Gas	40.849998	2
Joes Stale \$ Dent	150	1
Local Utilities	98	1
Local Utilities	34	1
Local Utilities	87.5	1
Ma Bell	200.33	1
Ma Bell	150	1
Reading R.R	.245.34	1

分析：

输出结果由原来的将 14 行分成 7 组变成了 13 组，为什么它会多出了这么多组呢？我们来看一下：

INPUT/OUTPUT：

```
SELECT PAYEE, REMARKS FROM CHECKS WHERE PAYEE= 'Joans Gas';
```

PAYEE	REMARKS
Joans Gas	Gas
Joans Gas	Gas

分析：

你可以看到这两个记录的内容是完全一样的，所以在运行 GROUP BY 以后把它们合并成了一个记录，而其它行则是唯一的，所以合并以后仍然是唯一的。

下例是对 REMARKS 进行分组并找出组中的最大值和最小值。

INPUT/OUTPUT：

```
SELECT MIN (AMOUNT), MAX (AMOUNT) FROM CHECKS GROUP BY  
REMARKS;
```

MIN	MAX
245.34	245.34
10.5	10.5
200.33	200.33
15.75	98
150	150
150	150
34	34
60	60
34	34
87.5	87.5
25	25
24.35	24.35

如果我们在分组时指定的列名与 SELECT 中所指定的列名不相同时会有什么情况发生呢？

INPUT/OUTPUT:

```
SELECT PAYEE, MAX (AMOUNT), MIN (AMOUNT) FROM CHECKS
```

```
GROUP BY REMARKS;
```

Dynamic SQL Error

-SQL error code = -104

-invalid column reference

分析:

查询无法对 REMARK 进行分组，当查询在 REMARK 字段中找到了两个重复的数值但它们的 PAYEE 不同，这表明 GAS 有两个 PAYEE，这将会导致错误的产生。

规则是，当要求分组结果返回多个数值时不能在在 SELECT 子句中使用除分组列以外的列，这将会导致错误的返回值。你可以使用在 SELECT 中未列出的列进行分组，例如：

INPUT/OUTPUT:

```
SELECT PAYEE, COUNT (AMOUNT) FROM CHECKS
```

```
GROUP BY PAYEE, AMOUNT;
```

PAYEE	COUNT
Abes Cleaners	1
Abes Cleaners	1
Cash	1

Cash	1
Cash	1
Joans Gas	1
Joans Gas	1
Joes Stale \$ Dent	1
Local Utilities	1
Local Utilities	1
Local Utilities	1
Ma Bell	1
Ma Bell	1
Reading R.R	1

分析：

这个愚蠢的查询显示的记录与你在表中输入的记录数一样多，这表明你可以在 GROUP BY 中使用 AMOUNT，尽管在 SELECT 中没有提到过该字段，现在试着将 AMOUNT 字段从 GROUND 部分移动到 SELECT 部分，如下例：

```
SELECT PAYEE, AMOUNT, COUNT (AMOUNT) FROM CHECKS GROUP BY
PAYEE;
```

Dynamic SQL Error

-SQL error code = -104

-invalid column reference

SQL 不能运行查询，因为在 SELECT 中出现的字段没有在 GROUP BY 中指出，所以我们不得不采用下边的方法进行分组。

INPUT/OUTPUT:

```
SELECT PAYEE, AMOUNT, REMARKS FROM CHECKS WHERE PAYEE=
'Cash';
```

PAYEE	AMOUNT	REMARKS
Cash	25	Wild Night Out
Cash	60	Trip to Boston
Cash	34	Trip to Dayton

如果你的用户要求你将这三行数据输出并按 PAYEE 进行分组的话，那么请问数据并不重复的 REMARKS 字段的内容应该放在哪里？切记，当进行分组以后由于这三行数据是同一组，所以结果只有一行。SQL 无法在同时为你做两种工作，所以它会说：Error #31: Can't do two things at once.

HAVING 子句

如何对你需要进行分组的数据进行限制呢？这里我们使用 ORGCHART 表，内容如下：

INPUT:

```
SELECT * FROM ORGCHART;
```

OUTPUT:

NAME	TEAM	SALARY	SICKLEAVE	ANNUALLEAVE
ADAMS	RESEARCH	34000.00	34	12
WILKES	MARKETING	31000.00	40	9
STOKES	MARKETING	36000.00	20	19
MEZA	COLLECTIONS	40000.00	30	27
MERRICK	RESEARCH	45000.00	20	17
RICHARDSON	MARKETING	42000.00	25	18
FURY	COLLECTIONS	35000.00	22	14
PRECOURT	PR	37500.00	24	24

如果你想对输出的结果进行分组并显示每一组的平均工资，你可以输入如下语句：

INPUT/OUTPUT:

```
SELECT TEAM, AVG (SALARY) FROM ORGCHART GROUP BY TEAM;
```

TEAM	AVG
COLLECTIONS	37500.00
MARKETING	36333.33
PR	37500.00
RESEARCH	39500.00

下边的这条语句的目的是返回分组后平均工资低于 38000 的组：

INPUT/OUTPUT:

```
SELECT TEAM, AVG (SALARY) FROM ORGCHART
WHERE AVG (SALARY) <38000 GROUP BY TEAM;
```

Dynamic SQL Error

-SQL error code = -104

-Invalid aggregate reference

分析：

错误产生的原因是由于汇总函数不能工作在 WHERE 子句中，如果想要让这个查询工作的话，我们需要一些新东西 HAVING 子句，输入下边的查询就会得到你想要的结果了：

INPUT/OUTPUT：

```
SELECT TEAM, AVG (SALARY) FROM ORGCHART GROUP BY TEAM
HAVING AVG (SALARY) <38000;
```

TEAM	AVG
COLLECTIONS	37500.00
MARKETING	36333.33
PR	37500.00

分析：

HAVING 子句允许你将汇总函数作为条件，但是如果 HAVING 后边没有汇总函数时会有什么结果呢，看下例：

INPUT/OUTPUT：

```
SELECT TEAM, AVG (SALARY) FROM ORGCHART GROUP BY TEAM
HAVING SALARY<38000;
```

TEAM	AVG
PR	37500.00

分析：

为什么这一次的结果与上一次的不同，子句 HAVING AVG(SALARY) < 38000 是对每一组的 SALARY 求平均数并将数值大于 38000 的组返回。正像你所想到的那样，HAVING SALARY < 38000 则是用另外一种处理方式，所以就会有不同的结果。根据 SQL 的解释规则，如果用户要求对分组数据执行 HAVING SALARY < 38000，它会对数据库中的每个记录均进行检查，并且剔除 SALARY 大于 38000 的，这样的话就只有 PR 符合条件了，在其它组中都至少有一条 SALARY 大于 38000 的记录。（并不是所有的解释器都执行这条语句，ACCESS 就不能——译者）

INPUT/OUTPUT：

```
SELECT NAME, TEAM, SALARY FROM ORGCHART ORDER BY TEAM;
```

NAME	TEAM	SALARY
FURY	COLLECTIONS	35000.00
MEZA	COLLECTIONS	40000.00

WILKES	MARKETING	31000.00
STOKES	MARKETING	36000.00
RICHARDSON	MARKETING	42000.00
PRECOURT	PR	37500.00
ADAMS	RESEARCH	34000.00
MERRICK	RESEARCH	45000.00

分析：

结果就是除了 PR 外所有的组都被剔除了，事实上你的要求是返回组中内容 SALARY<38000 的组，SQL 完全按照你的要求去做了，所以你不必生气！

警告：在一些解释器中如果在 HAVING 子句中使用了非汇总函数将会导致错误（ACCESS 就是这样——译者）。在没有对你所使用的解释器做认真的检查之前不要认为这样做一定会得到结果！

HAVING 子句允许使用多个条件吗？试一下：

INPUT：

```
SELECT TEAM, AVG (SICKLEAVE), AVG (ANNUALLEAVE) FROM ORGCHART
GROUP BY TEAM
HAVING AVG (SICKLEAVE) >25 AND AVG (ANNUALLEAVE) <20;
```

分析：

下表显示的是按 TEAM 进行分组并符合平均病假大于 25 天和平均年休假少于 20 天的组。

OUTPUT：

TEAM	AVG	AVG
MARKETING	28	15
RESEARCH	27	15

你也可以在 HAVING 中使用在 SELECT 中没有指出的字段进行汇总，如：

INPUT/OUTPUT：

```
SELECT TEAM, AVG (SICKLEAVE), AVG (ANNUALLEAVE) FROM ORGCHART
GROUP BY TEAM HAVING COUNT (TEAM) >1;
```

TEAM	AVG	AVG
COLLECTIONS	26	21
MARKETING	28	15
RESEARCH	27	15

该查询返回了组中成员大于 1 的组，虽然的 SELECT 子句中没有出现 COUNT (TEAM) 语句，但是它还是在 HAVING 子句中起到了它应有的作用。

在 HAVING 子句中也可以使用其它的逻辑操作符，例如：

INPUT/OUTPUT：

```
SELECT TEAM, MIN (SALARY), MAX (SALARY) FROM ORGCHART
GROUP BY TEAM HAVING AVG (SALARY) >37000
```

OR

```
MIN (SALARY) >32000;
```

TEAM	MIN	MAX
COLLECTIONS	35000.00	40000.00
PR	37500.00	37500.00
RESEARCH	34000.00	45000.00

操作符 IN 也可以在 HAVING 子句中使用，如下例：

INPUT/OUTPUT：

```
SELECT TEAM, AVG (SALARY) FROM ORGCHART GROUP BY TEAM
HAVING TEAM IN ('PR','RESEARCH');
```

TEAM	AVG
PR	37500.00
RESEARCH	39500.00

子句的综合应用

这一部分没有什么新的东西，只是通过一些例子来向你演示如何将这些子句进行综合的应用。

例 1：

找出所有 CHECKS 表中对 CASH 和对 GAS 支付的记录，并按 REMARKS 进行排序。

INPUT：

```
SELECT PAYEE, REMARKS FROM CHECKS WHERE PAYEE='Cash'
OR REMARKS LIKE 'Ga%' ORDER BY REMARKS;
```

OUTPUT：

PAYEE	REMARKS
-------	---------

Joans Gas	Gas
Joans Gas	Gas
Local Utilities	Gas
Cash	Trip to Boston
Cash	Trip to Dayton
Cash	Wild Night Out

分析：

这里使用了 LIKE 来查找在 REMARKS 中以 GA 开头的内容，通过使用 OR 来控制 WHERE 返回满足两个条件之一的内容。

如果您有相同的要求，并要求按 PAYEE 进行分组，看下列：

INPUT：

```
SELECT PAYEE, REMARKS FROM CHECKS WHERE PAYEE='Cash'
OR REMARKS LIKE 'Ga%' GROUP BY PAYEE ORDER BY REMARKS;
```

分析：

这个查询将会由于无法对 REMARKS 进行分组而无法工作，切记，无论在什么情况下进行分组，SELECT 语句中出现的字段只能是在 GROUP BY 中出现过的才可以——除非你在 SELECT 子句中不指定任何字段。

例 2：

使用 ORGCHART 表，找出病候天数少于 25 天的人的工资，并按名字进行排序。

INPUT：

```
SELECT NAME, SALARY FROM ORGCHART WHERE SICKLEAVE<25 ORDER BY
NAME;
```

OUTPUT：

NAME	SALARY
FURY	35000.00
MERRICK	45000.00
PRECOURT	37500.00
STOKES	36000.00

这是个非常简单的查询并且使你对 WHERE 和 ORDER BY 子句有了更新的体会。

例 3：

仍使用 ORGCHART 表，对每一个 TEAM，显示 TEAM，AVG (SALARY)，AVG (SICKLEAVE) 和 AVG (ANNUALLEAVE)。

INPUT:

```
SELECT TEAM, AVG (SALARY), AVG (SICKLEAVE), AVG (ANNUALLEAVE) FROM
```

~~CHECKS~~ ORGCHART

GROUP BY TEAM;

OUTPUT:

TEAM	AVG	AVG	AVG
COLLECTIONS	37500.00	26	21
MARKETING	36333.33	28	15
PR	37500.00	24	24
RESEARCH	39500.00	26	15

下边的查询有一些有趣的变化，你想一下它会有什么结果。

INPUT/OUTPUT:

```
SELECT TEAM, AVG(SALARY), AVG(SICKLEAVE), AVG(ANNUALLEAVE)
```

```
FROM ORGCHART GROUP BY TEAM ORDER BY NAME;
```

TEAM	AVG	AVG	AVG
RESEARCH	39500.00	27	15
COLLECTIONS	37500.00	26	21
PR	37500.00	24	24
MARKETING	36333.33	28	15

只使用 ORDER BY 语句可能会为你提供一些线索。

INPUT/OUTPUT:

```
SELECT NAME, TEAM FROM ORGCHART ORDER BY NAME, TEAM;
```

NAME	TEAM
ADAMS	RESEARCH
FURY	COLLECTIONS
MERRICK	RESEARCH
MEZA	COLLECTIONS
PRECOURT	PR
RICHARDSON	MARKETING
STOKES	MARKETING
WILKES	MARKETING

分析:

当 SQL 引擎对结果进行排序时，它使用的是 NAME 列（注意，排序使用 SELECT 未

指的列是完全合法的), 并且忽略了重复的 TEAM 值, 这样就只得到了四个值, 在 ORDER BY 中包括 TEAM 字段是没有必要的, 因为 NAME 本身没有任何重复的记录, 用下列语句也可以得到相同的结果。

INPUT/OUTPUT:

```
SELECT NAME, TEAM FROM ORGCHART ORDER BY NAME;
```

NAME	TEAM
ADAMS	RESEARCH
FURY	COLLECTIONS
MERRICK	RESEARCH
MEZA	COLLECTIONS
PRECOURT	PR
RICHARDSON	MARKETING
STOKES	MARKETING
WILKES	MARKETING

现在你可以对它做一些变化, 例如我们可以让其逆序排列。

INPUT/OUTPUT:

```
SELECT NAME, TEAM FROM ORGCHART ORDER BY NAME DESC;
```

NAME	TEAM
WILKES	MARKETING
STOKES	MARKETING
RICHARDSON	MARKETING
PRECOURT	PR
MEZA	COLLECTIONS
MERRICK	RESEARCH
FURY	COLLECTIONS
ADAMS	RESEARCH

例 4: (大结局)

可能用一个查询来完成每一件工作吗? 是的, 但是在许多时候它们的结果却是令人费解的。

WHERE 子句与 ORDER BY 子句常在对单行进行处理时看到, 如:

INPUT/OUTPUT:

```
SELECT * FROM ORGCHART ORDER BY NAME DESC;
```

NAME	TEAM	SALARY	SICKLEAVE	ANNUALLEAVE
WILKES	MARKETING	31000.00	40	9
STOKES	MARKETING	36000.00	20	19

RICHARDSON	MARKETING	42000.00	25	18
PRECOURT	PR	37500.00	24	24
MEZA	COLLECTIONS	40000.00	30	27
MERRICK	RESEARCH	45000.00	20	17
FURY	COLLECTIONS	35000.00	22	14
ADAMS	RESEARCH	34000.00	34	12

GROUP BY 和 HAVING 子句常用在对数据进行汇总操作上:

INPUT/OUTPUT:

```
SELECT PAYEE, SUM (AMOUNT) TOTAL, COUNT (PAYEE) NUMBER_WRITTEN
FROM CHECKS GROUP BY PAYEE HAVING SUM (AMOUNT) >50;
```

PAYEE	TOTAL	NUMBER_WRITTEN
Cash	119	3
Joes Stale \$ Dent	150	1
Local Utilities	219.5	3
Ma Bell	350.33002	2
Reading R.R	.245.34	1

如果把它们结合起来使用会有出人意料的结果，例如：

INPUT:

```
SELECT PAYEE, SUM (AMOUNT) TOTAL, COUNT (PAYEE) NUMBER_WRITTEN
FROM CHECKS WHERE AMOUNT>=100 GROUP BY PAYEE
HAVING SUM (AMOUNT) >50;
```

OUTPUT:

PAYEE	TOTAL	NUMBER_WRITTEN
Joes Stale \$ Dent	150	1
Ma Bell	350.33002	2
Reading R.R	.245.34	1

将其与下边的结果进行对比:

INPUT/OUTPUT:

```
SELECT PAYEE, AMOUNT FROM CHECKS ORDER BY PAYEE;
```

PAYEE	AMOUNT
Abes Cleaners	10.5
Abes Cleaners	24.35
Cash	25

Cash	34
Cash	60
Joans Gas	15.75
Joans Gas	25.1
Joes Stale \$ Dent	150
Local Utilities	34
Local Utilities	87.5
Local Utilities	98
Ma Bell	150
Ma Bell	200.33
Reading R.R	.245.34

分析：

你使用了 WHERE 子句在分组前将 AMOUNT 小于 50 的记录过滤掉了，我们并不试图告诉你不要结合使用这两种分组，你在以后可能会有这方面的需要。但是请不要随便地结合使用这两种子句。在上例的表中只有为数不多的几行（否则这本书的内容需要用车来拉了），而在你的实际工作中数据库可能有成千上万行，结合使用后造成的变化就不会像现在这样明显了。

总结

在今天我们学习了与扩展 SELECT 语句功能相关的所有子句，切记要认真仔细地去对计算机描述你的需求，我们的基本 SQL 教育到这里就结束了。你已经有足够的能力对单个表进行操作了，明天（第 6 天：归并表格），我们将有机会在多个表中工作。

问与答

问：像这些功能在这一周的早些时候我们已经学习过了，为什么今天还要再学习一次？

答：我们的确在第 3 天就曾经提到过 WHERE 子句，我们在那时使用 WHERE 是为了更加可靠地进行操作，WHERE 在今天出现是因为它是一个子句，而我们在今天讨论的主题是子句。

校练场

1、哪种子句的作用与 LIKE (<exp>%) 相似？

- 2、 GROUP BY 子句的功能是什么，哪种子句的功能与它类似？
- 3、 下面的查询会工作吗？

INPUT:

```
SQL>SELECT NAME, AVG (SALARY), DEPARTMENT FROM PAY_TBL
WHERE DEPARTMENT='ACCOUNTING' ORDER BY NAME
GROUP BY DEPARTMENT, SALARY;
```

- 4、 为什么在使用 HAVING 子句时我们总是同时使用 GROUP BY 子句？
- 5、 你可以使用在 SELECT 语句中没有出现的列进行排序吗？

练习

- 1、 使用上例中的 ORGCHART 表找一下每一个 TEAM 中 SICKLEAVE 天数超过 30 天的人数。
- 2、 使用 CHECKS 表，返回如下结果：

OUTPUT:

CHECK#	PAYEE	AMOUNT
1	MA BELL	150

練習

- 1 SELECT TEAM ,COUNT(TEAM) 人數
FROM ORGCHART
WHERE SICKLEAVE>30
GROUP BY TEAM
- 2 SELECT CHECK#,PAYEE,AMOUNT
FROM CHECKS
WHERE CHECK#=1

第六天：表的联合

今天我们将学习联合操作，这种操作可以让你从多个表中选择数据并对它们进行维护。
在今天结束以后我们将会具有以下能力：

- 执行外部联合
- 执行内部联合
- 执行左联合
- 执行右联合
- 进行等值联合
- 进行不等值联合

介绍

能够从多个表中选择和操作数据是 SQL 的特色之一。如果没有这个功能的话你将不得不将一个应用程序所需的所有数据放在一个表中。如果表不能共享那么你将不得不在多个表中保存相同的数据，而且每当用户需要查询一个新的内容时你就不得不重新设计、和编译你的数据库系统。SQL 中的 JOIN 语句可以让你的设计出比那种庞大的表格更小和更为专业以及更容易使用的表格。

在一个 SELECT 语句中使用多个表

就像多萝茜在《绿野仙踪》中所做的一样，你其实在《第二天：查询——SELECT 语句的使用》中学习过 SELECT 和 FROM 以后就已经具备了联合多个表格的能力了。但是与多萝茜不同，你执行联合操作并不需要将脚后跟磕三下。使用下边的两个表，简单点，不妨就叫 TABEL1 和 TABLE2。

注：在今天的查询使用的是 BORLAND 的 ISQL 产生的结果，你会发现它与我们在本书的早些时候所使用的查询有一些不同之处。例如：它没有 SQL 提示符，而且在语句的末尾也没有分号（在 ISQL 中分号是可选项）。但是查询的基本结构是相同的。

INPUT:

SELECT * FROM TABLE1

OUTPUT:

ROW	REMARKS
row1	Table 1
Row2	Table 1
Row3	Table 1
Row4	Table 1
Row5	Table 1
Row6	Table 1

INPUT:

SELECT * FROM TABLE2

OUTPUT:

ROW	REMARKS
Row1	Table2
Row2	Table2
Row3	Table2
Row4	Table2
Row5	Table2
Row6	Table2

要联合两个表格，可以像下边这样操作：

INPUT:

SELECT *

FROM TABLE1, TABLE2

OUTPUT:

ROW	REMARKS	ROW	REMARKS
row 1	Table 1	row 1	table 2
row 1	Table 1	row 2	table 2
row 1	Table 1	row 3	table 2
row 1	Table 1	row 4	table 2
row 1	Table 1	row 5	table 2
row 1	Table 1	row 6	table 2
row 2	Table 1	row 1	table 2
.....			

总计有 36 行，它们都是从哪来的呢？这又属于哪一种联合类型呢？

分析：

认真看一下，你会发现联合的结果其实就是将 TABLE1 中的每一行与 TABLE2 中的每一行都接合了起来，其中的一个片断如下：

ROW	REMARKS	ROW	REMARKS
row 1	Table 1	row 1	table 2
row 1	Table 1	row 2	table 2
row 1	Table 1	row 3	table 2
row 1	Table 1	row 4	table 2

ROW	REMARKS	ROW	REMARKS
row 1	Table 1	row 5	table 2
row 1	Table 1	row 6	table 2

看，TABEL2 中的每一行均与 TABEL1 中的第一行联合了起来。祝贺你，你已经完成了你的第一个联合。可是它是哪一种联合呢？内部联合？外部联合？还是别的？嗯……其实这种联合应该称为交叉联合（其实就是笛卡尔叉积——译者）。交叉联合在今天不像其它联合那样有用，但是这种联合表明了联合的最基本属性，联合源自表格。

假设你为生计所迫到一家自行车行中卖零件。在你设计一个数据库时，你建立了一个大表，其中囊括了所有的相关列。每当你有一个新的需要时，你就向其中加入了一个新列或者是重新建立一张表，向其中加入所有以前的数据后再建立一个特定的查询。最后，你的数据库将会由于它自身的重量而崩溃——你不想看到这种情况。所以额外的选择就是使用关系模型，你只需所相关的数据放入同一张表中。下边显示的是你的客户表。

INPUT:

```
SELECT * FROM CUSTOMER
```

OUTPUT:

NAME	ADDRESS	STATE	ZIP	PHONE	REMARKS
TRUE WHEEL	550 HUSKER	NE	58702	555-4545	NONE
BIKE SPEC	CPT SHRIVE	LA	45678	555-1234	NONE
LE SHOPPE	HOMETOWN	KS	54678	555-1278	NONE
AAA BIKE	10 OLDTOWN	NE	56784	555-3421	JOHN-MGR
JACKS BIKE	24 EGLIN	FL	34567	555-2314	NONE

分析:

这张表中包括了所有的你需要对顾客进行的描述，而关于你所卖的产品则在另外一张表上。

INPUT:

```
SELECT * FROM PART
```

OUTPUT:

PARTNUM	DESCRIPTION	PRICE
54	PEDALS	54.25
42	SEATS	24.50
46	TIRES	15.25
23	MOUNTAIN BIKE	350.45

76	ROAD BIKE	530.00
10	TANDEM	1200.00

而你的定单则有着它们自己的表：

INPUT：

```
SELECT * FROM ORDERS
```

OUTPUT：

ORDEREDON	NAME	PARTNUM	QUANTITY	REMARKS
15-MAY-1996	TRUE WHEEL	23	6	PAID
19-MAY-1996	TRUE WHEEL	76	3	PAID
2-SEP-1996	TRUE WHEEL	10	1	PAID
30-JUN-1996	TRUE WHEEL	42	8	PAID
30-JUN-1996	BIKE SPEC	54	10	PAID
30-MAY-1996	BIKE SPEC	10	2	PAID
30-MAY-1996	BIKE SPEC	23	8	PAID
17-JAN-1996	BIKE SPEC	76	11	PAID
17-JAN-1996	LE SHOPPE	76	5	PAID
1-JUN-1996	LE SHOPPE	10	3	PAID
1-JUN-1996	AAA BIKE	10	1	PAID
1-JUL-1996	AAA BIKE	76	4	PAID
1-JUL-1996	AAA BIKE	46	14	PAID
11-JUL-1996	JACKS BIKE	76	14	PAID

这样做的好处是你可以用三个专职人员或部门来维护属于他们自己的数据，你也无需与数据库管理员来套交情好让他看管你那庞大的、多部门的数据库。另外的优点就是由于网路的发展，每个表都可以放在不同的机器上，所有它可以在适当的地点由对它的内部数据熟悉的人来进行维护（而不是像大型机那样需要一队的专家来进行维护）。

现在将 PARTS 表与 ORDERS 表进行联合：

INPUT/OUTPUT：

```
SELECT O.ORDEREDON, O.NAME, O.PARTNUM, P.PARTNUM, P.DESCRPTION
FROM ORDERS O, PART P
```

ORDEREDON	NAME	PARTNUM	PARTNUM	DESCRIPTION
15-MAY-1996	TRUE WHEEL	23	54	PEDALS
19-MAY-1996	TRUE WHEEL	76	54	PEDALS
2-SEP-1996	TRUE WHEEL	10	54	PEDALS
30-JUN-1996	TRUE WHEEL	42	54	PEDALS

ORDEREDON	NAME	PARTNUM	PARTNUM	DESCRIPTION
30-JUN-1996	BIKE SPEC	54	54	PEDALS
30-MAY-1996	BIKE SPEC	10	54	PEDALS
30-MAY-1996	BIKE SPEC	23	54	PEDALS
17-JAN-1996	BIKE SPEC	76	54	PEDALS
17-JAN-1996	LE SHOPPE	76	54	PEDALS
1-JUN-1996	LE SHOPPE	10	54	PEDALS
1-JUN-1996	AAA BIKE	10	54	PEDALS
1-JUL-1996	AAA BIKE	76	54	PEDALS
1-JUL-1996	AAA BIKE	46	54	PEDALS
11-JUL-1996	JACKS BIKE	76	54	PEDALS
.....				

分析:

上表只是结果集的一部分，实际上记录数应该有 14（定单行数）×6（零件行数）=84 行，它与今天的早些时候 TABLE1 与 TABLE2 的联合类似，这条语句的结果仍然没有太大的用处。在我们对这种语句深入之前，我们先回想并讨论一下别名的问题。

正确地找到列

当你将 TABLE1 与 TABLE2 联合以后，你使用 SELECT * 来选择了表中的所有列。在联合表 ORDER 和 PART 时，SELECT 看起来不太好懂：

```
SELECT O.ORDEREDON, O.NAME, O.PARTNUM, P.PARTNUM, P.DESCRPTION
```

SQL 可以知道 ORDEREDON 和 NAME 是在 ORDER 表中而 DESCRIPTION 则存在于 PART 表中。但是 PARTNUM 呢，它在两个表中都有啊！如果你想使用在两个表中都存在的列，你必须使用别名来说明你想要的是哪一列。常用的办法为每一个表分配一个简单的字符，就像你在 FROM 子句中所做的那样：

```
FROM ORDERS O, PART P
```

你可以在每一列中都使用这个字符，就像你刚才在 SELECT 中所做的那样，SELECT 子句也可以写成下边的形式：

```
SELECT ORDEREDON, NAME, O.PARTNUM, P.PARTNUM, DESCRIPTION
```

可是不要忘记，有时你会不得不回过头来对查询进行维护，所以让它更具有可读性并没有什么坏处。还是不要使用这种省略的形式吧？

等值联合

下边的表是 ORDERS 与 PARTS 表的联合结果的片断，作为缺货的情况：

30-JUN-1996	TRUEWHEEL	42	54	PEDALS
30-JUN-1996	BIKESPEC	54	54	PEDALS
30-MAY-1996	BIKESPEC	10	54	PEDALS

注意到 PARTNUM 是两个表的共有字段。如果输入如下的语句会有什么结果呢？

INPUT:

```
SELECT O.ORDEREDON, O.NAME, O.PARTNUM, P.PARTNUM, P.DESCRPTION
FROM ORDERS O, PART P WHERE O.PARTNUM=P.PARTNUM
```

OUTPUT:

ORDEREDON	NAME	PARTNUM	PARTNUM	DESCRIPTION
1-JUN-1996	AAA BIKE	10	10	TANDEM
30-MAY-1996	BIKE SPEC	10	10	TANDEM
2-SEP-1996	TRUE WHEEL	10	10	TANDEM
1-JUN-1996	LE SHOPPE	10	10	TANDEM
30-MAY-1996	BIKE SPEC	23	23	MOUNTAIN BIKE
15-MAY-1996	TRUE WHEEL	23	23	MOUNTAIN BIKE
30-JUN-1996	TRUE WHEEL	42	42	SEATS
1-JUL-1996	AAA BIKE	46	46	TIRES
30-JUN-1996	BIKE SPEC	54	54	PEDALS
1-JUL-1996	AAA BIKE	76	76	ROAD BIKE
17-JAN-1996	BIKE SPEC	76	76	ROAD BIKE
19-MAY-1996	TRUE WHEEL	76	76	ROAD BIKE
11-JUL-1996	JACKS BIKE	76	76	ROAD BIKE
17-JAN-1996	LE SHOPPE	76	76	ROAD BIKE

分析：

利用在两个表中都存在的 PARTNUM 列，我们得到了存储在 ORDERS 表中的信息以及在 PARTS 中的与 ORDERS 相关的信息，它表明了您已经定出的零件数量。这种联合操作称为等值联合，因为它只显示第一个表中的数据以及第二个表中的，存在于第一个表中的数值。

你也可以使用 WHERE 子句对其结果进行更大的限制，例如：

INPUT/OUTPUT:

```
SELECT O.ORDEREDON, O.NAME, O.PARTNUM, P.PARTNUM, P.DESCRPTION
```

```
FROM ORDERS O, PART P WHERE O.PARTNUM=P.PARTNUM
AND O.PARTNUM=76
```

ORDEREDON	NAME	PARTNUM	PARTNUMDES	DESCRIPTION
1-JUL-1996	AAABIKE	76	76	ROADBIKE
17-JAN-1996	BIKESPEC	76	76	ROADBIKE
19-MAY-1996	TRUEWHEEL	76	76	ROADBIKE
11-JUL-1996	JACKSBIKE	76	76	ROADBIKE
17-JAN-1996	LESHOPPE	76	76	ROADBIKE

PARTNUM 为 76 的零件描述不是非常准确，你不想把它作为零件出售（我们非常遗憾在发现在许多数据库系统中需要最终用户知道一些非常晦涩的代码，而该代码所代表的东西原本就有着自己的，非常清楚明白的名字。请不要像他们那样做！）。这一行代码也可以写成如下方式：

INPUT/OUTPUT:

```
SELECT O.ORDEREDON, O.NAME, O.PARTNUM, P.PARTNUM, P.DESCRPTION
FROM ORDERS O, PART P WHERE O.PARTNUM=P.PARTNUM
AND P.DESCRPTION= 'ROAD BIKE'
```

ORDEREDON	NAME	PARTNUM	PARTNUMDES	DESCRIPTION
1-JUL-1996	AAABIKE	76	76	ROADBIKE
17-JAN-1996	BIKESPEC	76	76	ROADBIKE
19-MAY-1996	TRUEWHEEL	76	76	ROADBIKE
11-JUL-1996	JACKSBIKE	76	76	ROADBIKE
17-JAN-1996	LESHOPPE	76	76	ROADBIKE

顺着这个思路，我们来看一下一个或多个表是如何进行联合的。在下边的例子中 employee_id 显然是唯一标识列，因为你可以有在同一个公司、有相同薪水并且他们的名字也相同的雇员，但是他们会各自拥有他们自己的 employee_id，所以如果要对这两个表进行联合，我们应该使用 employee_id 列。

EMPLOYEE_TABLE	EMPLOYEE_PAY_TABLE
EMPLOYEE_ID	EMPLOYEE_ID
LAST_NAME	SALARY
FIRST_NAME	DEPARTMENT
MIDDLE_NAME	SUPERVISOR
	MARITAL_STATUS

INPUT:

```
SELECT E.EMPLOYEE_ID, E.LAST_NAME, EP.SALARY FROM EMPLOYEE_TBL E,
EMPLOYEE_PAY_TBL EP WHERE E.EMPLOYEE_ID = EP.EMPLOYEE_ID
AND E.LAST_NAME = 'SMITH';
```

OUTPUT:

E.EMPLOYEE_ID	E.LAST_NAME	EP.SALARY
13245	SMITH	35000.00

技巧：如果你在联合表的时候没有使用 WHERE 子句，你执行的其实是笛卡尔联合（也就是笛卡尔叉积）这种联合会对 FROM 中指出的表进行完全的组合，如果每个表有 200 个记录的话，那么所得到的结果将会有 40000 行（ 200×200 ）。这太大了！所以除非你确实是想对表中的所有记录进行联合，否则一定不要忘记使用 WHERE 子句。

现在回到原来的表中，我们已经对联合进行了充分的准备，可以用它来完成一些实际的工作了：找一下我们卖 road bikes 共卖了多少钱。

INPUT/OUTPUT:

```
SELECT SUM (O.QUANTITY * P.PRICE) TOTAL FROM ORDERS O, PART P
WHERE O.PARTNUM = P.PARTNUM AND P.DESCRPTION = 'ROAD BIKE'
```

TOTAL
19610.00

在这种设置中，销售人员可以保证 ORDERS 表的更新，生产部门则可以保持 PART 表的更新，而你则无需对数据库的底层进行重新设计。

注：注意在 SQL 语句中表以及列的别名的使用，你可能会因为别名多按了许许多多按钮，但是它可以让你语句更具有可读性。

我们可以对更多的表进行联合吗？例如：我们需要生成发票所要的信息，可以这样写：

INPUT/OUTPUT:

```
SELECT C.NAME, C.ADDRESS, (O.QUANTITY * P.PRICE) TOTAL
FROM ORDER O, PART P, CUSTOMER C
WHERE O.PARTNUM = P.PARTNUM AND O.NAME = C.NAME
```

NAME	ADDRESS	TOTAL
TRUE WHEEL	550 HUSKER	1200.00
BIKE SPEC	CPT SHRIVE	2400.00

LE SHOPPE	HOMETOWN	3600.00
AAA BIKE	10 OLDTOWN	1200.00
TRUE WHEEL	55O HUSKER	2102.70
BIKE SPEC	CPT SHRIVE	2803.60
TRUE WHEEL	55O HUSKER	196.00
AAA BIKE	10 OLDTOWN	213.50
BIKE SPEC	CPT SHRIVE	542.50
TRUE WHEEL	55O HUSKER	1590.00
BIKE SPEC	CPT SHRIVE	5830.00
JACKS BIKE	24 EGLIN	7420.00
LE SHOPPE	HOMETOWN	2650.00
AAA BIKE	10 OLDTOWN	2120.00

把语句写成如下格式会更具可读性：

INPUT/OUTPUT：

SELECT C.NAME, C.ADDRESS, O.QUANTITY * P.PRICE TOTAL

FROM ORDERS O, PART P, CUSTOMER C

WHERE O.PARTNUM = P.PARTNUM

AND O.NAME = C.NAME ORDER BY C.NAME

NAME	ADDRESS	TOTAL
AAA BIKE	10 OLDTOWN	213.50
AAA BIKE	10 OLDTOWN	2120.00
AAA BIKE	10 OLDTOWN	1200.00
BIKE SPEC	CPT SHRIVE	542.50
BIKE SPEC	CPT SHRIVE	2803.60
BIKE SPEC	CPT SHRIVE	5830.00
BIKE SPEC	CPT SHRIVE	2400.00
JACKS BIKE	24 EGLIN	7420.00
LE SHOPPE	HOMETOWN	2650.00
LE SHOPPE	HOMETOWN	3600.00
TRUE WHEEL	55O HUSKER	196.00
TRUE WHEEL	55O HUSKER	2102.70
TRUE WHEEL	55O HUSKER	1590.00
TRUE WHEEL	55O HUSKER	1200.00

注：注意当将三个表进行联合的时候（ORDERS、PART、CUSTOMER），ORDERS 表

被使用了两次，而其它的表只使用了一次。通常，根据给定的条件返回行数最少的表会作为驱动表——也就是基表。在查询中除基表以外的其它表通常是向基表联合以便更有效地获得数据。所以，在本例中 ORDERS 表是基表。在大多数的数据库中只有很少的几个基表（直接或间接地）与其它的所有表联合。（见第 15 天“高性能的 SQL 语句流” [for more on base tables.](#)）

在下边的使用中我们通过使用 DESCRIPTION 列来使上述的查询更精确（因而也就更有效）：

INPUT/OUTPUT：

```
SELECT C.NAME, C.ADDRESS, O.QUANTITY * P.PRICE TOTAL, P.DESCRPTION
FROM ORDERS O, PART P, CUSTOMER C
WHERE O.PARTNUM=P.PARTNUM AND O.NAME = C.NAME ORDER BY C.NAME
```

NAME	ADDRESS	TOTAL	DESCRIPTION
AAA BIKE	10 OLDTOWN	213.50	TIRES
AAA BIKE	10 OLDTOWN	2120.00	ROAD BIKE
AAA BIKE	10 OLDTOWN	1200.00	TANDEM
BIKE SPEC	CPT SHRIVE	542.50	PEDALS
BIKE SPEC	CPT SHRIVE	2803.60	MOUNTAIN BIKE
BIKE SPEC	CPT SHRIVE	5830.00	ROAD BIKE
BIKE SPEC	CPT SHRIVE	2400.00	TANDEM
JACKS BIKE	24 EGLIN	7420.00	ROAD BIKE
LE SHOPPE	HOMETOWN	2650.00	ROAD BIKE
LE SHOPPE	HOMETOWN	3600.00	TANDEM
TRUE WHEEL	55O HUSKER	196.00	SEATS
TRUE WHEEL	55O HUSKER	2102.70	MOUNTAIN BIKE
TRUE WHEEL	55O HUSKER	1590.00	ROAD BIKE
TRUE WHEEL	55O HUSKER	1200.00	TANDEM

分析：

这是三个表联合后的结果，我们可以所得到的信息来开发票了。

注：在今天的开始 SQL 曾经联合过 TABEL1 和 TABEL2 并生成了一个新表有 X(TABLE1 的行数)×Y(TABLE2 的行数)列，联合并没有生成确实存在的表格，但它生成了一个虚拟的表格。对两个表联合（包括自我联合）后会根据 WHERE 所指定的条件生成一个新的集合，SELECT 语句减少了显示的列数，但是 WHERE 语句仍然把所有的列全返回了。在今天的例子中我们的表中只有为数不多的几列，而现实生活中的数据可能会

有成千上万列。如果你所使用的平台足够快，那么多表联合可能对系统的性能没有影响。可是如果你工作在一个比较慢的平台上，联合可能会导致死机。

不等值联合

既然 SQL 支持等值联合，你也许会推想它也支持不等值联合，你猜对了！等值联合是在 WHERE 子句中使用等号，而不等值联合则是在 WHERE 子句中使用除了等号以外的其它比较运算符。现下例：

INPUT:

```
SELECT O.NAME, O.PARTNUM, P.PARTNUM, O.QUANTITY * P.PRICE TOTAL
FROM ORDERS O, PART P WHERE O.PARTNUM > P.PARTNUM
```

OUTPUT:

NAME	PARTNUM	PARTNUM	TOTAL
TRUE WHEEL	76	54	162.75
BIKE SPEC	76	54	596.75
LE SHOPPE	76	54	271.25
AAA BIKE	76	54	217.00
JACKS BIKE	76	54	759.50
TRUE WHEEL	76	42	73.50
BIKE SPEC	54	42	245.00
BIKE SPEC	76	42	269.50
LE SHOPPE	76	42	122.50
AAA BIKE	76	42	98.00
AAA BIKE	46	42	343.00
JACKS BIKE	76	42	343.00
TRUE WHEEL	76	46	45.75
BIKE SPEC	54	46	152.50
BIKE SPEC	76	46	167.75
LE SHOPPE	76	46	76.25
AAA BIKE	76	46	61.00
JACKS BIKE	76	46	213.50
TRUE WHEEL	76	23	1051.35
TRUE WHEEL	42	23	2803.60

分析：

上边的表给出了满足条件 WHERE O.PARTNUM > P.PARTNUM 的所有联合内容，结合

你上边的自行车行的例子，这些信息似乎没有太多的意义。在现实世界中等值联合的使用要远远多于不等值联合，但是你的编程时可能会遇到使用不等值联合的情况。

外部联合与内部联合

就像不等值联合与等值联合相对应一样，外部联合是与内部联合相对应的。内部联合是指与个表内的行与本表内的数据相互进行联合，产生的结果行数取决于参加联合的行数，也就是说内部联合的行数取决于 WHERE 子句的结果。外部联合则是表间的联合，如上例中的 ORDERS 表与 PART 表的联合，内部联合的例子如下：

INPUT:

```
SELECT P.PARTNUM, P.DESCRPTION, P.PRICE, O.NAME, O.PARTNUM
FROM PART P JOIN ORDERS O ON ORDERS.PARTNUM = 54
```

OUTPUT:

PARTNUM	DESCRIPTION	PRICE	NAME	PARTNUM
54	PEDALS	54.25	BIKESPEC	54
42	SEATS	24.50	BIKESPEC	54
46	TIRES	15.25	BIKESPEC	54
23	MOUNTAIN BIKE	350.45	BIKESPEC	54
76	ROAD BIKE	530.00	BIKESPEC	54
10	TANDEM	1200.00	BIKESPEC	54

注：在这里你使用的语法中的 JOIN ON 不是 ANSI 标准中所指定的，而是我们所使用的解释器的附加语法，你可以用它来指明是内部联合还是外部联合，大多数解释器对这些都进行了类似的扩充。注意这种类型的联合没有 WHERE 子句。

分析：

结果表明 PART 表中的所有行都与 PARTNUM 为 54 的行进行了组合，再来看一个外部右联合的例子：

INPUT/OUTPUT:

```
SELECT P.PARTNUM, P.DESCRPTION, P.PRICE, O.NAME, O.PARTNUM, FROM PART P
RIGHT OUTER JOIN ORDERS O ON ORDERS.PARTNUM = 54
```

PARTNUM	DESCRIPTION	PRICE	NAME	PARTNUM
---------	-------------	-------	------	---------

PARTNUM	DESCRIPTION	PRICE	NAME		PARTNUM
<null>	<null>	<null>	TRUE	WHEEL	23
<null>	<null>	<null>	TRUE	WHEEL	76
<null>	<null>	<null>	TRUE	WHEEL	10
<null>	<null>	<null>	TRUE	WHEEL	42
54	PEDALS	54.25	BIKE	SPEC	54
42	SEATS	24.50	BIKE	SPEC	54
46	TIRES	15.25	BIKE	SPEC	54
23	MOUNTAIN BIKE	350.45	BIKE	SPEC	54
76	ROAD BIKE	530.00	BIKE	SPEC	54
10	TANDEM	1200.00	BIKE	SPEC	54
<null>	<null>	<null>	BIKE	SPEC	10
<null>	<null>	<null>	BIKE	SPEC	23
<null>	<null>	<null>	BIKE	SPEC	76
<null>	<null>	<null>	LE	SHOPPE	76
<null>	<null>	<null>	LE	SHOPPE	10
<null>	<null>	<null>	AAA	BIKE	10
<null>	<null>	<null>	AAA	BIKE	76
<null>	<null>	<null>	AAA	BIKE	46
<null>	<null>	<null>	JACKS	BIKE	76

分析：

这是一种新型的查询，这里我们第一次使用了 RIGHT OUTER JOIN，它会令 SQL 返回右边表集内的全部记录，如果当 ORDERS.PARTNUM<>54 则补以空值，下边是一个左联合的例子。

INPUT/OUTPUT:

SELECT P.PARTNUM, P.DESCRPTION, P.PRICE, O.NAME, O.PARTNUM,

FROM PART P LEFT OUTER JOIN ORDERS O ON ORDERS.PARTNUM = 54

PARTNUM	DESCRIPTION	PRICE	NAME	PARTNUM
54	PEDALS	54.25	BIKE SPEC	54
42	SEATS	24.50	BIKE SPEC	54
46	TIRES	15.25	BIKE SPEC	54
23	MOUNTAIN BIKE	350.45	BIKE SPEC	54
76	ROAD BIKE	530.00	BIKE SPEC	54
10	TANDEM	1200.00	BIKE SPEC	54

分析：

与内部联合的结果一样，都是六行。因为你使用的是左联合，PART 表决定返回的行数，而 PART 表比 ORDERS 表小，所以 SQL 把其余的行数都扔掉了。

不要对内部联合和外部联合操太多的心，大多数的 SQL 产品会判断应该在你的查询中使用哪一种联合。事实上，如果你在过程中使用它（或在程序内使用这（包括存储过程和将在第 13 天提到的《高级 SQL 使用》））。你无需指明联合类型，解释器会为你选择合适的语法形式，如果你指明的联合类型，解释器会用你指明的类型来代替优化的类型。

在一些解释器中使用+号来代替外部联合，+号的意思就是——显示我的全部内容包括不匹配的内容。语法如下：

SYNTAX:

```
SQL> select e.name, e.employee_id, ep.salary, ep.marital_status from e,ployee_tbl e,
        employee_pay_tbl ep
        where e.employee_id = ep.employee_id(+) and e.name like '%MITH';
```

分析：

这条语句将会联合两个表，标有+号的 employee_id 将会全部显示，包括不满足条件的记录。

表的自我联合

今天的最后一个内容是经常使用的自我联合，它的语法与联合两个表的语法相似。例如，表 1 的自我联合可以写成如下格式：

INPUT:

```
SELECT * FROM TABLE1, TABLE1
```

OUTPUT:

ROW	REMARKS	ROW	REMARKS
row 1	Table 1	row 1	Table 1
row 1	Table 1	row 2	Table 1
row 1	Table 1	row 3	Table 1
row 1	Table 1	row 4	Table 1
row 1	Table 1	row 5	Table 1
row 1	Table 1	row 6	Table 1
row 2	Table 1	row 1	Table 1
row 2	Table 1	row 2	Table 1

ROW	REMARKS	ROW	REMARKS
row 2	Table 1	row 3	Table 1
row 2	Table 1	row 4	Table 1
row 2	Table 1	row 5	Table 1
row 2	Table 1	row 6	Table 1

分析：

如果把这个表的内容全部列出的话它与联合两个有 6 行的表是相同的。这种联合对于检查内部数据的一致性。如果你的零件生产部门的某人犯了迷糊输入了一个已经存在的零件号时将会发生什么呢？这对于每一个人来说都是一个坏消息：发票会开错，你的应用程序会崩溃、会耗掉你许多宝贵的时光。在下表中，重复的 PARTNUM 会导致问题的产生：

INPUT/OUTPUT：

SELECT * FROM PART

PARTNUM	DESCRIPTION	PRICE
54	PEDALS	54.25
42	SEATS	24.50
46	TIRES	15.25
23	MOUNTAIN BIKE	350.45
76	ROAD BIKE	530.00
10	TANDEM	1200.00
76	CLIPPLESS SHOE	65.00<-NOTESAME#

下边的语句会使你的公司从不利的局面中摆脱出来：

INPUT/OUTPUT：

SELECT F.PARTNUM, F.DESCRPTION, S.PARTNUM, S.DESCRPTION

FROM PART F, PART S WHERE F.PARTNUM = S.PARTNUM

AND F.DESCRPTION <> S.DESCRPTION

PARTNUM	DESCRIPTION	PARTNUM	DESCRIPTION
76	ROAD BIKE	76	CLIPPLESS SHOE
76	CLIPPLESS SHOE	76	ROAD BIKE

分析：

直到有人问你为什么这个表的记录会是两个之前你会是一个英雄。你会记得你曾经学习过联合。是语句

WHERE F.PARTNUM = S.PARTNUM AND F.DESCRPTION <> S.DESCRPTION.

使你有了英雄的称号，当然，表中重复的记录内容将会被更正。

总结

今天你学习了对选择的表进行了所有的可能的联合。所得的结果是满足你所提出的选择的信息的内容。

没有了——现在你已经学习了关于 SELECT 语句的几乎所有的内容，剩下的一个内容——子查询将会在明天提到（第 7 天：内嵌了 SELECT 语句）。

问与答

问：既然我不可能用到这些联合，那为什么还要讲它们呢？

答：一知半解是危险的，你为无知付出的代价是昂贵的。现在，你已经有了足够的知识来了解 SQL 引擎在优化你的查询时所做的基本工作。

问：我可以对多少个表进行联合？

答：依解释器而定，有一些解释器有 25 个表的限制，另外一些则没有这个限制。但是请不要忘记，你联合的表越多，系统的响应就会越慢。为安全起见，请检查你的解释器看一看它最多允许同时使用多少个表。

问：为什么说将多个表联合为一个表的说法是不对的？

答：很简单，因为并没有这种事情发生，当你联合时，你只是从多个表中选出了特定的列。

校练场

1、如果一个表有 50000 行而另一个表有 100000 行时联合的结果会有多少行？

2、下边的联合属于哪一种类型的联合？

```
SELECT E.NAME, E.EMPLOYEE_ID, EP.SALARY FROM EMPLOYEE_TBL E, EMPLOYEE_PAY_TBL EP
WHERE E.EMPLOYEE_ID = EP.EMPLOYEE_ID; 等值联合
```

3、下边的查询语句能否工作？

```
A. SELECT NAME, EMPLOYEE_ID, SALARY FROM EMPLOYEE_TBL E, EMPLOYEE_PAY_TBL EP
WHERE EMPLOYEE_ID = EMPLOYEE_ID AND NAME LIKE '%MITH';
```



```
B. SELECT E.NAME, E.EMPLOYEE_ID, EP.SALARY FROM EMPLOYEE_TBL E, EMPLOYEE_PAY_TBL EP
WHERE NAME LIKE '%MITH';
```

```
C. SELECT E.NAME, E.EMPLOYEE_ID, EP.SALARY FROM EMPLOYEE_TBL E,EMPLOYEE_PAY_TBL EP
WHERE E.EMPLOYEE_ID = EP.EMPLOYEE_ID AND E.NAME LIKE '%MITH';
```

4、 是否在联合语句中 WHERE 子句中的第一个条件应该是联合条件？

5、 联合的限制为一列， 是否可以有更多的列？

练习

1、 在表的自我联合这部分， 最后的一个例子返回了两个结果， 请重写这个查询使它对多余的记录只返回一个结果。

2、 重写下边的查询使它更可读和简炼。

INPUT:

```
select orders.orderedon, orders.name, part.partnum,part.price, part.description
from orders, part
where orders.partnum = part.partnum and orders.orderedon
between '1-SEP-96' and '30-SEP-96' order by part.partnum;
```

3、 使用 ORDERS 表和 PART 表， 返回下边的结果。

OUTPUT:

ORDEREDON	NAME	PARTNUM	QUANTITY
2-SEP-96	TRUE WHEEL	10	1

第七天：子查询：内嵌的 SQL 子句

目标：

子查询是一种把查询的结果作为参数返回给另一个查询的一种查询。子查询可以让你将多个查询绑定在一起，到今天结束以后，你将掌握以下内容：

- 建立一个子查询
- 在你的子查询中使用 EXIST，ANY 和 ALL 关键字
- 建立和使用子查询的关联

注：今天的例子是使用 BORLAND 公司的 ISQL 建立的，我们在第六天使用的也是这种解释器。切记，这种查询没有 SQL>提示符以及行号。

建立一个子查询

简而言之，子查询可以让你把查询的结果与另一个查询绑定在一起，通用的语法格式如下：

SYNTAX:

```
SELECT * FROM TABLE1 WHERE TABLE1.SOMECOLUMN =  
(SELECT SOMEOTHERCOLUMN FROM TABLE2  
WHERE SOMEOTHERCOLUMN = SOMEVALUE)
```

注意一下第二个查询是如何嵌入到第一个查询之中的，这里用 ORDERS 和 PART 表来举一个实例：

INPUT:

```
SELECT * FROM PART
```

OUTPUT:

PARTNUM	DESCRIPTION	PRICE
54	PEDALS	54.25
42	SEATS	24.50
46	TIRES	15.25

PARTNUM	DESCRIPTION	PRICE
23	MOUNTAIN BIKE	350.45
76	ROAD BIKE	530.00
10	TANDEM	1200.00

INPUT/OUTPUT:

SELECT *

FROM ORDERS

ORDEREDON	NAME	PARTNUM	QUANTITY	REMARKS
15-MAY-1996	TRUE WHEEL	23	6	PAID
19-MAY-1996	TRUE WHEEL	76	3	PAID
2-SEP-1996	TRUE WHEEL	10	1	PAID
30-JUN-1996	TRUE WHEEL	42	8	PAID
30-JUN-1996	BIKE SPEC	54	10	PAID
30-MAY-1996	BIKE SPEC	10	2	PAID
30-MAY-1996	BIKE SPEC	23	8	PAID
17-JAN-1996	BIKE SPEC	76	11	PAID
17-JAN-1996	LE SHOPPE	76	5	PAID
1-JUN-1996	LE SHOPPE	10	3	PAID
1-JUN-1996	AAA BIKE	10	1	PAID
1-JUL-1996	AAA BIKE	76	4	PAID
1-JUL-1996	AAA BIKE	46	14	PAID
11-JUL-1996	JACKS BIKE	76	14	PAID

分析:

两表的共有字段是 PARTNUM，假如你不知道（或者是不想知道）这个字段，但是你又想用 PART 表的 description 字段来工作，这时可以使用子查询，语句如下：

INPUT/OUTPUT:

SELECT * FROM ORDERS WHERE PARTNUM =

(SELECT PARTNUM FROM PART WHERE DESCRIPTION LIKE "ROAD%")

ORDEREDON	NAME	PARTNUM	QUANTITY	REMARKS
19-MAY-1996	TRUE WHEEL	76	3	PAID
17-JAN-1996	BIKE SPEC	76	11	PAID
17-JAN-1996	LE SHOPPE	76	5	PAID
1-JUL-1996	AAA BIKE	76	4	PAID
11-JUL-1996	JACKS BIKE	76	14	PAID

分析:

更进一步，如果你使用了在第六天中的概念，你可以使 PARTNUM 列带有 DESCRIPTION，这样就会使那些对 PARTNUM 还不太清楚的人看得更明白些，如下例：

INPUT/OUTPUT：

```
SELECT O.ORDEREDON, O.PARTNUM, P.DESCRPTION, O.QUANTITY, O.REMARKS
FROM ORDERS O, PART P WHERE O.PARTNUM = P.PARTNUM
AND O.PARTNUM =(SELECT PARTNUM FROM PART
WHERE DESCRIPTION LIKE "ROAD%")
```

ORDEREDON	PARTNUM	DESCRIPTION	QUANTITY	REMARKS
19-MAY-1996	76	ROAD BIKE	3	PAID
1-JUL-1996	76	ROAD BIKE	4	PAID
17-JAN-1996	76	ROAD BIKE	5	PAID
17-JAN-1996	76	ROAD BIKE	11	PAID
11-JUL-1996	76	ROAD BIKE	14	PAID

分析：

查询的第一部分非常熟悉：

```
SELECT O.ORDEREDON, O.PARTNUM, P.DESCRPTION, O.QUANTITY,
O.REMARKS FROM ORDERS O, PART P
```

这里使用了别名 O 和 P 来指定了在 ORDERS 和 PART 表中你所感兴趣的 5 列，对于你要访问的在两个表中的名字唯一的列别名是没有必要的，可是它可以使你的语句更具有可读性，你看到的第一个 WHERE 子句内容如下：

```
WHERE O.PARTNUM = P.PARTNUM
```

它是将 ORDERS 与 PART 表进行归并的标准语句，如果你没有使用 WHERE 子句，那么你会得到两个表的记录的所有可能的组合，接下来就是子查询语句，内容如下：

```
AND O.PARTNUM =(SELECT PARTNUM FROM PART WHERE DESCRIPTION LIKE "ROAD%")
```

增加的限制使你的 PARTNUM 内容必须与你的子查询所返回的结果相等，子查询则非常简单，它要求返回以 ROAD%相符的 PARTNUM，使用 LIKE 语句是一种懒人的办法，使得你不必键入 ROAD BIKE。但是这只是你侥幸，如果在 PART 表中加入了一个新的记录名字为 ROADKILL 时呢？这时 PART 表的内容如下：

INPUT/OUTPUT：

```
SELECT * FROM PART
```

PARTNUM	DESCRIPTION	PRICE
---------	-------------	-------

PARTNUM	DESCRIPTION	PRICE
54	PEDALS	54.25
42	SEATS	24.50
46	TIRES	15.25
23	MOUNTAIN BIKE	350.45
76	ROAD BIKE	530.00
10	TANDEM	1200.00
77	ROADKILL	7.99

如果你没有觉察到这些改变而仍然使用原来的查询的话，你将会得到如下信息：

multiple rows in singleton select

你没有得到任何结果，SQL 的响应信息可能不会相同，但是你会同样地得不到任何结果。

想知道为什么会有这样的结果，请想一个 SQL 引擎的处理规则，你需要重新核查一下你的子查询，请输入：

INPUT/OUTPUT：

SELECT PARTNUM FROM PART WHERE DESCRIPTION LIKE "ROAD%"

PARTNUM

76

77

你会把这个结果赋给 O.PARTNUM =，就是这一步导致的错误。

分析：

PARTNUM 怎么能同时匹配 76 和 77 呢，解释器一定会给你这样的信息的，因为你是一个懒家伙！当你使用 LIKE 子句的时候，你就已经开始了犯错误的道路，如果你想使用比较运算符如>、<和=时，你必须确保你的查询结果是唯一的。在我们现在的这个例子中，应该使用=号来代替 LIKE，如下：

INPUT/OUTPUT：

SELECT O.ORDEREDON, O.PARTNUM, P.DESCRPTION, O.QUANTITY,

O.REMARKS FROM ORDERS O, PART P WHERE O.PARTNUM = P.PARTNUM

AND O.PARTNUM = (SELECT PARTNUM FROM PART

WHERE DESCRIPTION = "ROAD BIKE")

ORDEREDON	PARTNUM	DESCRIPTION	QUANTITY	REMARKS
19-MAY-1996	76	ROAD BIKE	3	PAID
1-JUL-1996	76	ROAD BIKE	4	PAID

ORDEREDON	PARTNUM	DESCRIPTION	QUANTITY	REMARKS
17-JAN-1996	76	ROAD BIKE	5	PAID
17-JAN-1996	76	ROAD BIKE	11	PAID
11-JUL-1996	76	ROAD BIKE	14	PAID

分析：

这个子查询将会返回唯一的结果，因为使用=会返回唯一的结果。当你需要唯一的结果时如何才能避免子查询返回多个结果呢？

首先是不是使用 LIKE，再就是设计表的时候就要保证你要搜索的字段内容是唯一的。你可以使用表自我归并的方法（昨天讲过）来检查给定字段的内容是否是唯一的。如果表是你自己设计的，你要让你搜索的列是表中的唯一列，你也可以使用 SQL 只返回单一结果的部分——汇总函数。

在子查询中使用汇总函数

像 SUM、AVG、COUNT、MIN 和 MAX 等汇总函数均返回单一的数值。如果想知道定单的平均金额可以用如下语句：

INPUT：

```
SELECT AVG(O.QUANTITY * P.PRICE)
FROM ORDERS O, PART P
WHERE O.PARTNUM = P.PARTNUM
```

OUTPUT：

```
AVG
2419.16
```

分析：

这条语句只返回一个平均值，如果你想找一下都有哪些定单的金额高于平均值的话，可以将上述语句使用子查询使用，完整的语句内容如下：

INPUT/OUTPUT：

```
SELECT O.NAME, O.ORDEREDON, O.QUANTITY * P.PRICE TOTAL
FROM ORDERS O, PART P WHERE O.PARTNUM = P.PARTNUM AND
O.QUANTITY * P.PRICE > (SELECT AVG(O.QUANTITY * P.PRICE)
FROM ORDERS O, PART P WHERE O.PARTNUM = P.PARTNUM)
```

NAME	ORDEREDON	TOTAL
LE SHOPPE	1-JUN-1996	3600.00
BIKE SPEC	30-MAY-1996	2803.60
LE SHOPPE	17-JAN-1996	2650.00
BIKE SPEC	17-JAN-1996	5830.00
JACKS BIKE	11-JUL-1996	7420.00

分析：

在这个例子中的 SELECT/FROM/WHERE 子句的区别不太明显。

```
SELECT O.NAME, O.ORDEREDON, O.QUANTITY * P.PRICE TOTAL
```

```
FROM ORDERS O, PART P WHERE O.PARTNUM = P.PARTNUM
```

这是归并两个表的常用方法，这种归并是必须的，因为单价在 PART 表上而数量则在 ORDERS 表上。WHERE 子句用来检测相关性错误（指一个主关键字对应两条记录的情况）。

之后则是子查询语句：

```
AND
```

```
O.QUANTITY * P.PRICE > (SELECT AVG(O.QUANTITY * P.PRICE)
```

```
FROM ORDERS O, PART P WHERE O.PARTNUM = P.PARTNUM)
```

第一个比较表达式是将每一条记录的金额与子查询中的平均金额进行比较。注意子查询中使用归并的原因与主查询是相同的，它严格地遵循着归并的语法。在子查询中没有什么秘密可言，它与单独的查询具有相同的语法格式。事实上，大多数子查询都是作为独立查询经过测试确定其只返回一个值以后才作为子查询使用的。

子查询的嵌套

嵌套就是将一个子查询嵌入到另一个子查询中去，例如：

```
Select * FROM SOMETHING WHERE ( SUBQUERY(SUBQUERY(SUBQUERY)));
```

子查询可被嵌套的深度依你的需要而定。例如：如果你想给那些花费超过了平均价格的客户发一个特别通知，你将会使用 CUSTOMERS 表中的如下信息：

INPUT：

```
SELECT * FROM CUSTOMER
```

OUTPUT：

NAME	ADDRESS	STATE	ZIP	PHONE	REMARKS
TRUE WHEEL	55O HUSKER	NE	58702	555-4545	NONE
BIKE SPEC	CPT SHRIVE	LA	45678	555-1234	NONE
LE SHOPPE	HOMETOWN	KS	54678	555-1278	NONE
AAA BIKE	10 OLDTOWN	NE	56784	555-3421	JOHN-MGR
JACKS BIKE	24 EGLIN	FL	34567	555-2314	NONE

你只需要对上边你的查找定单的查询做一点改动即可：

INPUT/OUTPUT：

SELECT ALL C.NAME, C.ADDRESS, C.STATE, C.ZIP FROM CUSTOMER C

WHERE C.NAME IN

(SELECT O.NAME FROM ORDERS O, PART P

WHERE O.PARTNUM = P.PARTNUM

AND

O.QUANTITY * P.PRICE > (SELECT AVG(O.QUANTITY * P.PRICE)

FROM ORDERS O, PART P

WHERE O.PARTNUM = P.PARTNUM))

NAME	ADDRESS	STATE	ZIP
BIKE SPEC	CPTSHRIVE	LA	45678
LE SHOPPE	HOMETOWN	KS	54678
JACKS BIKE	24EGLIN	FL	34567

分析：

注意一下圆括号最里边的内容，你会发现类似的语句：

SELECT AVG(O.QUANTITY * P.PRICE)

FROM ORDERS O, PART P WHERE O.PARTNUM = P.PARTNUM

结果传入的语句与你以前使用的 SELECT 语句有一些不同之处。

SELECT O.NAME FROM ORDERS O, PART P WHERE O.PARTNUM = P.PARTNUM

AND O.QUANTITY * P.PRICE > (...)

注意 SELECT 子句已经被改为返回单一的 NAME 列，运行该查询你会得到下表：

NAME
LE SHOPPE
BIKE SPEC
LE SHOPPE

NAME
BIKE SPEC
JACKS BIKE

我们曾经花过一些时间来讨论为什么子查询应该只返回一个数值，而这个查询返回了多个数值则是显而易见的。

将上述结果引入下边的语句：

```
SELECT C.NAME, C.ADDRESS, C.STATE, C.ZIP FROM CUSTOMER C
WHERE C.NAME IN (...)
```

分析：

头两行没有什么特别的内容，在第三行时再次引入了关键字 IN，看一下第二天的《查询简介：SELECT 语句的使用》，IN 是一种允许你在子查询中进行多行输出的工具。就像你原来记得的那样，它将返回与所列内容相匹配的记录，它的列出内容如下：

LE SHOPPE
BIKE SPEC
LE SHOPPE
BIKE SPEC
JACKS BIKE

根据子查询的条件得到了下边的内容：

NAME	ADDRESS	STATE	ZIP
BIKE SPEC	CPT SHRIVE	LA	45678
LE SHOPPE	HOMETOWN	KS	54678
JACKS BIKE	24 EGLIN	FL	34567

在子查询中使用关键字 IN 是非常普遍的，因为 IN 可以与一组数据进行对比，而且它不会使 SQL 的引擎检查出其中有冲突或是不合适的地方。

子查询也可以使用 GROUP BY 和 HAVING 子句，见下例：

INPUT/OUTPUT：

```
SELECT NAME, AVG (QUANTITY) FROM ORDERS
GROUP BY NAME HAVING AVG (QUANTITY) > (SELECT AVG (QUANTITY)
FROM ORDERS)
```

NAME	AVG
BIKE SPEC	8
JACKS BIKE	14

分析：

让我们来看一下这个查询在引擎中的工作过程。首先，请看子查询：

INPUT/OUTPUT：

```
SELECT AVG (QUANTITY) FROM ORDERS
```

该查询返回的结果为 6

而主查询的结果如下：

INPUT/OUTPUT：

```
SELECT NAME, AVG (QUANTITY) FROM ORDERS GROUP BY NAME
```

NAME	AVG
AAA BIKE	6
BIKE SPEC	8
JACKS BIKE	14
LE SHOPPE	4
TRUE WHEEL	5

在经过 HAVING 子句的检查后，该查询给出了两条大于平均 QUANTITY 的记录：

INPUT/OUTPUT：

```
HAVING AVG (QUANTITY) > (SELECT AVG (QUANTITY) FROM ORDERS)
```

NAME	AVG
BIKE SPEC	8
JACKS BIKE	14

相关子查询

到现在为止，我们所写出的子查询都是独立的，它们都没有涉及到其它的子查询。相关子查询可以接受外部的引用从而得到一些令人惊奇的结果。请看下边的这个查询：

INPUT：

```
SELECT * FROM ORDERS O WHERE 'ROAD BIKE' =
      (SELECT DESCRIPTION FROM PART P
       WHERE P.PARTNUM = O.PARTNUM)
```

OUTPUT：

ORDEREDON	NAME	PARTNUM	QUANTITY	REMARKS
19-MAY-1996	TRUE WHEEL	76	3	PAID

ORDEREDON	NAME	PARTNUM	QUANTITY	REMARKS
19-MAY-1996	TRUE WHEEL	76	3	PAID
17-JAN-1996	BIKE SPEC	76	11	PAID
17-JAN-1996	LE SHOPPE	76	5	PAID
1-JUL-1996	AAA BIKE	76	4	PAID
11-JUL-1996	JACKS BIKE	76	14	PAID

该查询实际上执行了类似下边的归并操作：

INPUT：

```
SELECT O.ORDEREDON, O.NAME, O.PARTNUM, O.QUANTITY, O.REMARKS
FROM ORDERS O, PART P WHERE P.PARTNUM = O.PARTNUM
AND P.DESCRPTION = 'ROAD BIKE'
```

OUTPUT：

ORDEREDON	NAME	PARTNUM	QUANTITY	REMARKS
19-MAY-1996	TRUE WHEEL	76	3	PAID
1-JUL-1996	AAA BIKE	76	4	PAID
17-JAN-1996	LE SHOPPE	76	5	PAID
17-JAN-1996	BIKE SPEC	76	11	PAID
11-JUL-1996	JACKS BIKE	76	14	PAID

分析：

事实上，除了命令不同以外，结果是一样的。相关查询的执行情况与归并非常相似。

这个相关查询查找了在子查询中指定的内容，在本例中相关查询的由于语句所确定：

```
WHERE P.PARTNUM = O.PARTNUM
```

当在子查询内部对 P.PARTNUM 和子查询外部的 O.PARTNUM 进行比较时，由于 O.PARTNUM 对于每一行均有一个不同的值，因此相关查询对每一行都执行了这一语句，

下面是 ORDERS 表的详细内容：

INPUT/OUTPUT:

```
SELECT * FROM ORDERS
```

ORDEREDON	NAME	PARTNUM	QUANTITY	REMARKS
15-MAY-1996	TRUE WHEEL	23	6	PAID
19-MAY-1996	TRUE WHEEL	76	3	PAID
2-SEP-1996	TRUE WHEEL	10	1	PAID
30-JUN-1996	TRUE WHEEL	42	8	PAID
30-JUN-1996	BIKE SPEC	54	10	PAID

ORDEREDON	NAME	PARTNUM	QUANTITY	REMARKS
30-MAY-1996	BIKE SPEC	10	2	PAID
30-MAY-1996	BIKE SPEC	23	8	PAID
17-JAN-1996	BIKE SPEC	76	11	PAID
17-JAN-1996	LE SHOPPE	76	5	PAID
1-JUN-1996	LE SHOPPE	10	3	PAID
1-JUN-1996	AAA BIKE	10	1	PAID
1-JUL-1996	AAA BIKE	76	4	PAID
1-JUL-1996	AAA BIKE	46	14	PAID
11-JUL-1996	JACKS BIKE	76	14	PAID

它的内容对应着下边的查询：

```
SELECT DESCRIPTION FROM PART P WHERE P.PARTNUM = O.PARTNUM
```

分析：

该操作将返回符合条件 $P.PARTNUM = O.PARTNUM$ 的 PART 表中的每一个 DESCRIPTION，之后 DESCRIPTION 又与下边的语句进行比较操作：

```
WHERE 'ROAD BIKE' =
```

由于每一行都进行了检查，所以这个相关查询可能会返回不止一条内容，不要以为返回多列在 WHERE 中就是允许的了，它仍会被拒绝！（本例中只所以会运行是因为 DESCRIPTION 是内容是唯一的）如下例，你试图将 PRICE 与“ROAD BIKE”进行比较，那么你会收到如下信息：

INPUT/OUTPUT：

```
SELECT * FROM ORDERS O WHERE 'ROAD BIKE' =
      (SELECT PRICE FROM PART P WHERE P.PARTNUM = O.PARTNUM)
```

conversion error from string "ROAD BIKE"

这是又一个不能运行的例子：

```
SELECT * FROM ORDERS O WHERE 'ROAD BIKE' =
      (SELECT * FROM PART P WHERE P.PARTNUM = O.PARTNUM)
```

分析：

在 WINDOWS 操作系统中将会导致一个一般保护性错误，SQL 引擎无法用=来关联所有的行。

相关查询也可以使用 GROUP BY 和 HAVING 子句，下边的查询是查找特定 PART 的总价并对其按 PARTNUM 进行分组：

INPUT/OUTPUT:

```
SELECT O.PARTNUM, SUM (O.QUANTITY*P.PRICE), COUNT (PARTNUM)
FROM ORDERS O, PART P WHERE P.PARTNUM = O.PARTNUM GROUP BY O.PARTNUM
HAVING SUM (O.QUANTITY*P.PRICE) > (SELECT AVG (O1.QUANTITY*P1.PRICE)
                                FROM PART P1, ORDERS O1
                                WHERE P1.PARTNUM = O1.PARTNUM
                                AND P1.PARTNUM = O.PARTNUM)
```

PARTNUM	SUM	COUNT
10	8400.00	4
23	4906.30	2
76	19610.00	5

分析:

子查询中不能只有

AVG(O1.QUANTITY*P1.PRICE)

因为子查询与主查询之间需要下边的关联语句

AND P1.PARTNUM = O.PARTNUM

将会计算每一组的平均值然后再与 HAVING SUM(O.QUANTITY*P.PRICE)>进行比较。

技巧：当在相关查询中使用 GROUP BY 和 HAVING 子句时，在 HAVING 子句中的列必需在 SELECT 或 GROUP BY 子句中存在，否则你将会收到一行非法引用的信息，因为这时与子查询对应的是每一组而不是每一行，对于组你无法进行比较操作。

EXISTS, ANY, ALL 的使用

EXISTS、ANY 和 ALL 关键字的用法不像它看上去那么直观，如果子查询返回的内容为非空时 EXISTS 返回 TRUE，否则返回 FALSE。例如：

INPUT/OUTPUT:

```
SELECT NAME, ORDEREDON FROM ORDERS WHERE EXISTS
(SELECT * FROM ORDERS WHERE NAME = 'TRUE WHEEL')
```

NAME	ORDEREDON
TRUE WHEEL	15-MAY-1996
TRUE WHEEL	19-MAY-1996

NAME	ORDEREDON
TRUE WHEEL	2-SEP-1996
TRUE WHEEL	30-JUN-1996
BIKE SPEC	30-JUN-1996
BIKE SPEC	30-MAY-1996
BIKE SPEC	30-MAY-1996
BIKE SPEC	17-JAN-1996
LE SHOPPE	17-JAN-1996
LE SHOPPE	1-JUN-1996
AAA BIKE	1-JUN-1996
AAA BIKE	1-JUL-1996
AAA BIKE	1-JUL-1996
JACKS BIKE	11-JUL-1996

分析：

与你所想的并不一样，在 EXISTS 中的子查询在这个例子中只返回一个值，因为从子查询中返回的行数至少有一行，EXIST 返回为 TRUE，这就使得表中的所有记录都被显示了出来，如果你把查询改成下边的形式，你将不会得到任何结果。

```
SELECT NAME, ORDEREDON FROM ORDERS
```

```
WHERE EXISTS (SELECT * FROM ORDERS WHERE NAME ='MOSTLY HARMLESS')
```

分析：

EXISTS 求得的结果为 FALSE，因为 MOSTLY HARMLESS 并不是 NAME 中的内容。

注：注意在 EXIST 所属的子查询中使用了 SELECT *，EXIST 并不管返回了多少列。

你可以使用 EXIST 来检查查询是否确实存在输出，从而实现对查询确实有结果才输出的控制。

如果你在相关查询中使用 EXISTS 关键字，它将会检查你所指出的每一种情况，例如：

INPUT/OUTPUT：

```
SELECT NAME, ORDEREDON FROM ORDERS O WHERE EXISTS
```

```
(SELECT * FROM CUSTOMER C WHERE STATE ='NE' AND C.NAME = O.NAME)
```

NAME	ORDEREDON
TRUE WHEEL	15-MAY-1996
TRUE WHEEL	19-MAY-1996
TRUE WHEEL	2-SEP-1996

NAME	ORDEREDON
TRUE WHEEL	30-JUN-1996
AAA BIKE	1-JUN-1996
AAA BIKE	1-JUL-1996
AAA BIKE	1-JUL-1996

与上例相比只有微小的改动，不相关的查询返回了所有在内布拉斯加州售出的已有订单的自行车。下边的子查询将返回与 CUSTOMER 和 ORDERS 相关的所有记录。

```
(SELECT * FROM CUSTOMER C WHERE STATE = 'NE' AND C.NAME = O.NAME)
```

分析：

对于相关的记录，如果 STATE 为 NE 则 EXISTS 返回为 TRUE，否则返回 FALSE。

与 EXISTS 相关的关键字有 ALL、ANY 和 SOME，ANY 与 SOME 具有同样的功能，乐观的人认为它给用户提供了一种选择，而悲观的人则认为它使得条件更加复杂化，见下例：

INPUT：

```
SELECT NAME, ORDEREDON FROM ORDERS WHERE NAME = ANY
```

```
(SELECT NAME FROM ORDERS WHERE NAME = 'TRUE WHEEL')
```

OUTPUT：

NAME	ORDEREDON
TRUE WHEEL	15-MAY-1996
TRUE WHEEL	19-MAY-1996
TRUE WHEEL	2-SEP-1996
TRUE WHEEL	30-JUN-1996

分析：

ANY 与子查询中的每一行与主查询进行比较，并对子查询中的每一行返回一个 TRUE 值。

```
(SELECT NAME FROM ORDERS WHERE NAME = 'TRUE WHEEL')
```

将 ANY 用 SOME 替换将会得到同样的结果。

INPUT/OUTPUT：

```
SELECT NAME, ORDEREDON FROM ORDERS WHERE NAME = SOME
```

```
(SELECT NAME FROM ORDERS WHERE NAME = 'TRUE WHEEL')
```

NAME	ORDEREDON
TRUE WHEEL	15-MAY-1996

NAME	ORDEREDON
TRUE WHEEL	19-MAY-1996
TRUE WHEEL	2-SEP-1996
TRUE WHEEL	30-JUN-1996

分析：

你可能注意到了它与 IN 有些类似，使用 IN 的相同查询语句如下：

INPUT/OUTPUT：

```
SELECT NAME, ORDEREDON FROM ORDERS WHERE NAME IN
(SELECT NAME FROM ORDERS WHERE NAME ='TRUE WHEEL')
```

NAME	ORDEREDON
TRUE WHEEL	15-MAY-1996
TRUE WHEEL	19-MAY-1996
TRUE WHEEL	2-SEP-1996
TRUE WHEEL	30-JUN-1996

分析：

你已经看到了，IN 返回的结果与 SOME 或 ANY 是相同的，是不是它就没有用了呢？

当然不是，IN 能像下边这样使用吗？

INPUT/OUTPUT：

```
SELECT NAME, ORDEREDON FROM ORDERS WHERE NAME > ANY
(SELECT NAME FROM ORDERS WHERE NAME ='JACKS BIKE')
```

NAME	ORDEREDON
TRUE WHEEL	15-MAY-1996
TRUE WHEEL	19-MAY-1996
TRUE WHEEL	2-SEP-1996
TRUE WHEEL	30-JUN-1996
LE SHOPPE	17-JAN-1996
LE SHOPPE	1-JUN-1996

回答是否定的，IN 只相当于多个等号的作用，而 ANY 和 SOME 则可以使用其它的比较运算符如大于或小于。它是你的一个新增工具。

ALL 关键字的作用在于子查询中的所有结果均满足条件时它才会返回 TRUE，奇怪吗？

ALL 常起双重否定的作用，见下例：

INPUT/OUTPUT：

```
SELECT NAME, ORDEREDON FROM ORDERS WHERE NAME <> ALL
```


(SELECT NAME FROM ORDERS WHERE NAME = 'JACKS BIKE')

NAME	ORDEREDON
TRUE WHEEL	15-MAY-1996
TRUE WHEEL	19-MAY-1996
TRUE WHEEL	2-SEP-1996
TRUE WHEEL	30-JUN-1996
BIKE SPEC	30-JUN-1996
BIKE SPEC	30-MAY-1996
BIKE SPEC	30-MAY-1996
BIKE SPEC	17-JAN-1996
LE SHOPPE	17-JAN-1996
LE SHOPPE	1-JUN-1996
AAA BIKE	1-JUN-1996
AAA BIKE	1-JUL-1996
AAA BIKE	1-JUL-1996

分析：

该语句返回了除 JACKS BIKE 的所有人，<>ALL 只有当左边的内容不存在于右边时才会返回 TRUE 值。

总结

今天我们做了不少关于子查询的练习，你已经学习了使用 SQL 中的重要的一部分。你也已经处理了 SQL 中最为困难的一部分——相关查询，相关查询会在查询与子查询之间建立一个关系，并对子查询中的每一个实例加以关系限制。不要为查询的长度所制约，你可以马上从它们中把子查询找出来。

问与答

问：在一些 SQL 实例中对于同一个问题给出了多个解决的方法，这是否会导致混乱？

答：不，事实上不是这样，多种方法得到相同的结果可以使你的查询更简练，这是 SQL 的优点。

校练场

1、在嵌套查询部分，有一个例子中子查询返回了以下几个数值：

```
LE SHOPPE
BIKE SPEC
LE SHOPPE
BIKE SPEC
JACKS BIKE
```

其中有一些结果是重复的，为什么在最终的结果中没有出现重复？

2、下面的话是对还是错？

- (1) 汇总函数如 SUM、AVG、COUNT、MAX、MIN 都返回多个数值。
- (2) 子查询最多允许嵌套两层。
- (3) 相关查询是完全的独立查询。

3、下边的子查询中哪一个是使用 ORDERS 表和 PART 表工作的？

INPUT/OUTPUT:

SQL> SELECT * FROM PART;

PARTNUM	DESCRIPTION	PRICE
54	PEDALS	54.25
42	SEATS	24.50
46	TIRES	15.25
23	MOUNTAIN BIKE	350.45
76	ROAD BIKE	530.00
10	TANDEM	1200.00

INPUT/OUTPUT:

SQL> SELECT * FROM ORDERS;

ORDEREDON	NAME	PARTNUM	QUANTITY	REMARKS
15-MAY-96	TRUE WHEEL	23	6	PAID
19-MAY-96	TRUE WHEEL	76	3	PAID
2-SEP-96	TRUE WHEEL	10	1	PAID
30-JUN-96	BIKE SPEC	54	10	PAID
30-MAY-96	BIKE SPEC	10	2	PAID
30-MAY-96	BIKE SPEC	23	8	PAID
17-JAN-96	BIKE SPEC	76	11	PAID
17-JAN-96	LE SHOPPE	76	5	PAID

1-JUN-96	LE SHOPPE	10	3	PAID
1-JUN-96	AAA BIKE	10	1	PAID
1-JUN-96	AAA BIKE	76	4	PAID
1-JUN-96	AAA BIKE	46	14	PAID
11-JUL-96	JACKS BIKE	76	14	PAID

A、 SQL> SELECT * FROM ORDERS WHERE PARTNUM =

SELECT PARTNUM FROM PART

WHERE DESCRIPTION = 'TRUE WHEEL';

B、 SQL> SELECT PARTNUM FROM ORDERS WHERE PARTNUM =

(SELECT * FROM PART

WHERE DESCRIPTION = 'LE SHOPPE');

C、 SQL> SELECT NAME, PARTNUM FROM ORDERS WHERE EXISTS

(SELECT * FROM ORDERS

WHERE NAME = 'TRUE WHEEL');

练习：

应用 ORDERS 表来写一个查询，返回所以字母顺序排列在 JACKS BIKE 之后的 NAMES 和 ODEREDON 数据。

第一周回顾

在第一周中我们简要地回顾了一下 SQL 的历史，第一周的重要是在 SQL 的 SELECT 语句上，下边给出的内容是关于 SELECT 语句的语法与我们学习的前后对照情况表。

- SELECT [DISTINCT | ALL] (Day 2)--Columns (Day 1), Functions (Day 4)
- FROM (Day 2)--Tables or Views (Day 1), Aggregate Functions (Day 4)
- WHERE (Day 5)--Condition (Day 3), Join (Day 6), Subquery (Day 7)
- GROUP BY (Day 5)--Columns (Day 3)
- HAVING (Day 5)--Aggregate Function (Day 4)
- UNION | INTERSECT (Day 3)--(Placed between two SELECT statements)
- ORDER BY (Day 5)--Columns (Day 1)

假如你需要在你的程序中建立成千上万个查询的话，你会发现它们中的 80%将会以 SELECT 开头，而余下的 20%将在第二周中提到。

预览

在第二周中我们将会学习到数据库管理的新技能，我们将会学习如何：

- 建立和删除一个表
- 为你的朋友提供访问权限和有效地防止你的敌人
- 更新和删除表中的数据

第二周概貌

这一周都讲些什么

在第一周我们讲了使用 SELECT 语句进行 SQL 的最基本的查询，从最简单的 SELECT 语句开始，我们学习了如何从数据库中取得数据，然后我们学习了 SQL 中的函数，它们的用处很大，例如在转换数据的格式或财务领域。然后你很快就学到了用不同的方法从数据库中取得数据。子句如 WHERE、ORDER BY 和 GROUP BY 允许你对查询进行定制以返回有特定要求的数据记录，你学习了归并操作以从不同的表中返回数据，当你需要运行多个查询且每个查询都需要前一个查询返回的内容时子查询是特别有用处的。

在第二周我们将进一步地学习 SQL 的使用。

- 第 8 天教你如何修改数据库中的数据。你也许很害怕向数据库中录入数据，但是手工输入数据并不总是必须的。现代的数据库系统大多都支持你从其它的数据库格式中导入或导出数据。此外，SQL 还提供了几个很有用的语句使你可以熟练地操纵数据库中的数据。
- 第 9 天你将学习如何建立和维护数据库中的表。你也将学习如何建立数据库和管理数据库的磁盘空间。
- 第 10 天将学习如何建立、维护、使用数据库的视图和索引。
- 第 11 天将涉及事务处理。对数据库进行处理以及撤消对它的改动，它对于在线事务处理程序非常有用。
- 第 12 天的重点是数据库的安全性，拥有数据库安全性知识可以让你更有效地管理数据库。
- 第 13 天将学习如何在大型的应用程序中使用 SQL，内嵌型 SQL 常常运行于宿主语言如 C 或 COBOL 上。此外，开放数据联接（ODBC）可以让你在应用程序中写出在通过数据库驱动在不同的数据库系统上运行的代码。也将会提到一些 SQL 的高级特性。
- 在第 14 天将会讨论如何使用动态 SQL。并用几个例子来演示如何在应用程序中使用 SQL。

第八天：操作数据

目标

今天我们来讨论一下数据操作问题，在今天中我们将学习以下内容：

- 如何使用 INSERT、UPDATE 和 DELETE 来处理数据。
- 在操作数据时使用 WHERE 子句的重要性。
- 从外部数据源中导入和导出数据的基本方法。

数据操作语句

到现在为止我们已经学习了从数据库出取得数据的每一种可能的操作。当获得数据以后，你可以在应用程序中使用或编辑它。在第一周主要讨论获得数据。但是，可能你会对如何向数据库中输入数据感兴趣，你也可能会很想知道如何来编辑数据。今天，我们将讨论三个关于如何对数据库中的表中的数据进行操作的三条语句，这三条语句是：

INSERT 语句

UPDATE 语句

DELETE 语句

在过去，你也许使用过基于 PC 机的数据库系统，如 ACCESS、dBASE IV 和 FOXPRO。这些产品提供了很好的输入，编辑和删除数据的工具。这就是 SQL 为什么提供了对数据进行编辑的基本语句可又允许用户使用应用程序自带的工具进行编辑的原因。SQL 的程序员应该具有将数据送入数据库中的能力。此外，大型的数据库系统常常不能按照数据库设计和编程人员的意图来进行设计。因为这些数据库系统是为大容量和多用户环境准备的，所以它的设计重点放在了如何优化查询和数据引擎上了。

大多数商业化的数据库系统都提供了导入导出数据工具。数据被存储在有分隔符的文本文件之中，格式化的文本文件常常存储着与表相关的信息。相关的工具如 ORACLE 的 SQL*Loader、SQL Server's bcp (bulk copy) 以 Microsoft Access 的导入和导出数据的工具都将在今天提到。

注：今天的例子是用 Personal Oracle7 做的，请注意它与其它的命令解释器在语句上和数据

返回的形式上的不同之处。

插入语句

INSERT 语句允许你向数据库中输入数据，它有两种写法：

INSERT VALUES 和 INSERT SELECT

INSERT VALUES 语句

该语句每次向表中输入一条记录，如何操作的规模小，只有几条语句需要输入时它是非常有用的。该语句的语法形式如下：

SYNTAX:

```
INSERT INTO table_name (col1, col2...) VALUES (value1, value2...)
```

该语句的作用是向表中加以一个新的记录，其数值为你所指定的数值。使用该语句向表中插入数据时你必须遵循以下三条规则：

- 你所要插入的数值与它所对应的字段必须具有相同的数据类型。
- 数据的长度必须小于字段的长度。例如：你不能向一个长 40 个字符的字段中插入一个长 80 个字符的字符串。
- 插入的数值列表必须与字段的列表相对应。（也就是说第一个数值在第一个字段，第二个数值在第二个字段）

例 8.1:

假定你有一个 COLLECTION 的表中存储着你所收集的材料，你可以用下边的语句来查看其中的内容：

INPUT:

```
SQL> SELECT * FROM COLLECTION;
```

OUTPUT:

ITEM	WORTH	REMARKS
NBA ALL STAR CARDS	300	SOME STILL IN BIKE SPOKES
MALIBU BARBIE	150	TAN NEEDS WORK
STAR WARS GLASS	5.5	HANDLE CHIPPED
LOCK OF SPOUSES HAIR	1	HASN'T NOTICED BALD SPOT YET

如果你想向表中加入一个新记录，你可以像这样写：

INPUT/OUTPUT:

```
SQL> INSERT INTO COLLECTION (ITEM, WORTH, REMARKS)
```

```
VALUES('SUPERMANS CAPE', 250.00, 'TUGGED ON IT');
```

1 row created.

你可以用一个简单的 SELECT 语句来验证插入的结果:

INPUT/OUTPUT:

```
SQL> SELECT * FROM COLLECTION;
```

ITEM	WORTH	REMARKS
NBA ALL STAR CARDS	300	SOME STILL IN BIKE SPOKES
MALIBU BARBIE	150	TAN NEEDS WORK
STAR WARS GLASS	5.5	HANDLE CHIPPED
LOCK OF SPOUSES HAIR	1	HASN'T NOTICED BALD SPOT YET
SUPERMANS CAPE	250	TUGGED ON IT

分析:

INSERT 语句并不需要列的名字, 如果列的名字没有给出, SQL 会把数据添入对应的列号中。也就是说: SQL 会把第一个值插入到第一列中, 把第二个值插入到第二列中, 依此类推。

例 8.2

下边的语句将会像例 8.1 中的表中插入数值:

INPUT:

```
SQL> INSERT INTO COLLECTION VALUES
```

```
2 ('STRING', 1000.00, 'SOME DAY IT WILL BE VALUABLE');
```

1 row created

分析:

为了表明它与例 8.1 的效果是相同的, 你可以用下边的例子来对其进行验证:

INPUT:

```
SQL> SELECT * FROM COLLECTION;
```

OUTPUT:

ITEM	WORTH	REMARKS
NBA ALL STAR CARDS	300	SOME STILL IN BIKE SPOKES
MALIBU BARBIE	150	TAN NEEDS WORK

ITEM	WORTH	REMARKS
STAR WARS GLASS	5.5	HANDLE CHIPPED
LOCK OF SPOUSES HAIR	1	HASN'T NOTICED BALD SPOT YET
SUPERMANS CAPE	250	TUGGED ON IT
STRING	1000	SOME DAY IT WILL BE VALUABLE

插入空值

在第几天的《建立和操作表》中你将会学到如何使用 SQL 的 CREATE TABLE 语句来创建一个表。现在你需要知道的是当一个列被创建以后，它可能有一定的规则限制，其中之一就是它应该（或不应该）包含空值的存在。空值的意思就是该处数值为空，但不是零——这属于整数范畴、或是空格——这属于字符串范畴。而是说在空值处根本就没有数据存在，如果列被定义为 NOT NULL（这时列中不允许有空值存在），则当你使用 INSERT 语句时必须在此列插入一个数值，如果你违反了 this 规则，那么你将收到一个错误的信息。

警告：你可以在对应的空值列（规则上不允许为空值）插入空格，它不会被看为空值，而且看起来该处并没有数据。

INPUT:

```
SQL> insert into collection values
```

```
2 ('SPORES MILDEW FUNGUS', 50.00, '');
```

OUTPUT:

```
1 row inserted.
```

分析:

在使用空格来代替空值以后，你可以在选择语句中使用空格:

INPUT/OUTPUT:

```
SQL> select * from collection
```

```
2 where remarks = ' ';
```

ITEM	WORTH	REMARKS
SPORES MILDEW FUNGUS	50.00	

分析:

返回的结果就好像在那里有一个空值一样，只从输出上区别这里是空格还是空值是不太可能的。

如果 REMARKS 列被定义为不允许空值，那么当输入下边的语句：

INPUT/OUTPUT：

```
SQL> INSERT INTO COLLECTION
```

```
2 VALUES('SPORES MILDEW FUNGUS',50.00,NULL);
```

你将会得到一个错误信息：

```
INSERT INTO COLLECTION
```

```
          *
```

ERROR at line 1:

ORA-01400: mandatory (NOT NULL) column is missing or NULL during insert

注：请注意语法，数字和空值不需要引号，而字符型数据则需要引号。

插入唯一值

在许多数据库管理系统中都允许你建立一个具有唯一值属性的列。这个属性的意思就是在当前的表中当前列的内容不得出现重复。这个属性在向一个已有的表中插入或更新数据时可能会导致问题的产生，见下例：

INPUT：

```
SQL> INSERT INTO COLLECTION VALUES('STRING', 50, 'MORE STRING');
```

OUTPUT：

```
INSERT INTO COLLECTION VALUES('STRING', 50, 'MORE STRING')
```

```
          *
```

ERROR at line 1:

ORA-00001: unique constraint (PERKINS.UNQ_COLLECTION_ITEM) violated

分析：

在本例中你试图在 COLLECTION 表的 ITEM 列中插入另外一个叫 STRING 的项目。由于 ITEM 列已经被定义为一个唯一的值，所以返回了一个错误结果。对于这个问题 ANSI SQL 没有提供解决方法，但许多商业化的解释器会对此进行扩充，如下例：

```
IF NOT EXISTS (SELECT * FROM COLLECTION WHERE NAME = 'STRING')
```

```
INSERT INTO COLLECTION VALUES('STRING', 50, 'MORE STRING')
```

这一例子在 Sybase 系统中是支持的。

一个正当的，标准化的表中应该有一个唯一值列或关键字列，这一字段在归并表格的时候非常有用，如果你使用索引的话它也可以大幅度地提高你查询的速度（见第 10 天《创建视图和索引》）。

注：下边的这个插入语句将会向表中插入一个新的雇员：

```
SQL> insert into employee_tbl values  
      ('300500177', 'SMITHH', 'JOHN');
```

在按下回车键以后，你发现你把 SMITH 拼错了。别担心！你可以使用 ROLLBACK 命令来回溯操作，而数据则并不会被插入。关于 ROLLBACK 语句的详细使用方法请参见第 11 天的《事务处理控制》。

INSERT SELECT 语句

INSERT VALUE 语句在向表中插入几个数据的时候非常有用，但显然这是不够的。如果你想向表中插入 25,000 行数据时怎么办？在这种情况下 INSERT SELECT 语句就非常有效。它允许程序员拷贝一个或一组表的信息到另外一个表中。你可以在下边这几种情况下使用该语句。需要查询的表经常产生利润的增加、需要查询的表可以从多个数据库或表中获得外部数据，由于多个表的查询要比单一表的查询速度慢得多，因此对单个表的查询速度要远远高于复杂而缓慢的多个表查询。在服务器/客户机系统上需要查询的表的数据经常存储在客户机上以减少网络中的数据传输速度。

见下例：

INPUT：

```
SQL> insert into tmp_tbl  
      2  select * from table;
```

OUTPUT：

19,999 rows inserted.

分析：

你可以将所有的数据都插入到一个临时表中。

注：并不是所有的数据库管理系统都支持临时表。请检查你的数据库系统的文档，看看它是否支持临时表，在第 14 天中你将会知道对于它的更详细的内容。

INSERT SELECT 语句的语法格式如下：

语法：

```
INSERT INTO table_name (col1, col2...)
```

```
SELECT col1, col2... FROM tablename WHERE search_condition
```

本质上来说它是将一个 SELECT 语句的输出结果在输入到另一个表格中去，在 INSERT VALUE 中的规则也适用于 INSERT SELECT 语句。如果想把表 COLLECTION 中的内容复制到另一个叫 INVENTORY 的表中去，你可以使用例 8.3 中的语句。

例 8.3：

本例将创建一个叫 INVENTORY 的表：

INPUT：

```
SQL> CREATE TABLE INVENTORY
```

```
2 (ITEM CHAR(20),
```

```
3 COST NUMBER,
```

```
4 ROOM CHAR(20),
```

```
5 REMARKS CHAR(40));
```

OUTPUT：

Table created.

下边的语句将向表中插入 COLLECTION 表中的数据：

INPUT/OUTPUT：

```
SQL> INSERT INTO INVENTORY (ITEM, COST, REMARKS)
```

```
2 SELECT ITEM, WORTH, REMARKS
```

```
3 FROM COLLECTION;
```

6 rows created.

你可以使用 SELECT 语句来检验 INSERT 的结果：

INPUT/OUTPUT：

```
SQL> SELECT * FROM INVENTORY;
```

ITEM	COSTROOM	REMARKS
NBA ALL STAR CARDS	300	SOME STILL IN BIKE SPOKES
MALIBU BARBIE	150	TAN NEEDS WORK
STAR WARS GLASS	5.5	HANDLE CHIPPED
LOCK OF SPOUSES HAIR	1	HASN'T NOTICED BALD SPOT YET
SUPERMANS CAPE	250	TUGGED ON IT

ITEM	COSTROOM	REMARKS
STRING	1000	SOME DAY IT WILL BE VALUABLE

注：数据已经出现在表中了，可是在你使用 COMMIT 语句之前它并不会真正生效，事务处理工作可以由 COMMIT 确认或只是简单地放弃。关于 COMMIT 的详细内容可见第 11 天。

分析：

你已经成功了，虽然有些费力，但是你已经将 COLLECTION 中的数据复制到了 INVERTORY 表中。

INSERT SELECT 语句要求你遵循如下规则：

- SELECT 语句不能从被插入数据的表中选择行。
- INSERT INTO 中的列数必须与 SELECT 语句返回的列数相等。
- INSERT INTO 中的数据类型要与 SELECT 语句返回的数据类型相同。

INSERT SELECT 语句的另外一个用处是当你需要对表进行重新定义时对表进行备份。这时需要你通过选择原始表中的所有数据并将其插入到一个临时表中来完成。例如：

```
SQL> insert into copy_table
2 select * from original_table;
```

然后你就可以放心地对原始表进行变更了。

注：在今天的早些时候你将会学习如何向一个表中导入其它数据库中的数据，几乎每一种商用的数据库都有它们自己的数据存储格式，编程人员经常需要对其进行格式转换。你将学习有关这方法的通用方法。

UPDATE 语句

该语句的作用是将已存在的记录的内容改变，语法格式如下：

SYNTAX:

```
UPDATE table_name SET columnname1 = value1 [, columnname2 = value2]...
```

```
WHERE search_condition
```

UPDATE 语句首先要检查 WHERE 子句，对于符合 WHERE 子句条件的记录将会用给定的数据进行更新。

例 8.4:

INPUT:

SQL> UPDATE COLLECTION SET WORTH = 900 WHERE ITEM = 'STRING';

OUTPUT:

1 row updated.

下边的查询可以用来验证确实已经进行了更新操作。

INPUT/OUTPUT:

SQL> SELECT * FROM COLLECTION WHERE ITEM = 'STRING';

ITEM	WORTH	REMARKS
STRING	900	SOME DAY IT WILL BE VALUABLE

下边是一个对多个记录进行更新的例子:

INPUT/OUTPUT:

SQL> update collection set worth = 900, item = ball where item = 'STRING';

1 row updated.

注：你所使用的解释器的对多个记录进行更新的语法可能会与这里给出的并不相同。

注：注意在 900 上没有加引号，因为它是数值类型。而在 STRING 上则有引号，因为它是一个字符串。

例 8.5:

如果在 UPDATE 语句中省略了 WHERE 子句，那么给定表中的所有记录都会被更新。

INPUT/OUTPUT:

SQL> UPDATE COLLECTION SET WORTH = 555;

6 rows updated.

下边的 SELECT 查询表明了表中的所有记录都已经被更新了。

INPUT/OUTPUT:

SQL> SELECT * FROM COLLECTION;

ITEM	WORTH	REMARKS
NBA ALL STAR CARDS	555	SOME STILL IN BIKE SPOKES
MALIBU BARBIE	555	TAN NEEDS WORK
STAR WARS GLASS	555	HANDLE CHIPPED

ITEM	WORTH	REMARKS
LOCK OF SPOUSES HAIR	555	HASN'T NOTICED BALD SPOT YET
SUPERMANS CAPE	555	TUGGED ON IT
STRING	555	SOME DAY IT WILL BE VALUABLE

当然你也应该检查一下它是否也对具有唯一值属性的列进行了更新操作。

警告：如果你在 UPDATE 语句中没有使用 WHERE 子句，那么所有给定表中的记录都会被更新。

一些数据库管理系统对标准的 UPDATE 语句进行了扩展。SQL SERVER 的 Transact-SQL 就是它们中的一个例子，它允许使用 FROM 子句实现对给定表的记录用其它表中的数据来进行更新操作。其语法表达如下：

SYNTAX:

```
UPDATE table_name SET columnname1 = value1 [, columnname2 = value2]...
FROM table_list WHERE search_condition
```

例 8.6:

这是一个实例:

INPUT:

```
SQL> UPDATE COLLECTION SET WORTH = WORTH * 0.005;
```

INPUT/OUTPUT:

```
SQL> SELECT * FROM COLLECTION;
```

ITEM	WORTH	REMARKS
NBA ALL STAR CARDS	2.775	SOME STILL IN BIKE SPOKES
MALIBU BARBIE	2.775	TAN NEEDS WORK
STAR WARS GLASS	2.775	HANDLE CHIPPED
LOCK OF SPOUSES HAIR	2.775	HASN'T NOTICED BALD SPOT YET
SUPERMANS CAPE	2.775	TUGGED ON IT
STRING	2.775	SOME DAY IT WILL BE VALUABLE

分析:

该语法在当给定表需要更新的内容源自于其它多个表的时候非常有用。切记该语法不是标准语法，在使用它之前请先查看一下你所使用的数据库的文档看一看它是否为你的数据库系统所支持。

UPDATE 语句也可以用一个数学运算式的结果来对给定数据进行更新操作，当使用这项技术时，必须注意你所使用的表达式结果与需要更新的数据字段为同一种数据类型。而

且其长度也要与被更新字段的定义长度相符。

当使用计算值时可能会有两个问题产生：截断和溢出。例如当将一个小数转换为整数时可能会有截断的情况产生。而当计算的结果超过了该字段的定义数据长度时会导致溢出，这会使你的数据库返回一个错误。

注：一些数据库系统可以为你处理溢出问题，ORACLE 7 可以在这时将其转成指数形式以避免错误，但是你要清楚在使用数据类型时这种错误存在的可能性。

技巧：如果你在你更新列的以后发现了错误，你可以使用 ROLLBACK 语句来取消更新操作（就像你对 INSERT 所做的那样）。关于该命令在第 11 天会有更多的介绍。

DELETE 语句

与向数据库中加入数据相对应，你可能需要删除数据库中的数据。DELETE 语句的语法格式如下：

SYNTAX:

```
DELETE FROM tablename WHERE condition
```

对于 DELETE 命令你需要注意的第一件事就是它不会出现确认提示。而用户似乎已经习惯于确认提示了。举例来说，当我们在操作系统中删除了某个文件或目录时，Are you sure? (Y/N)经常会在命令执行之前出现。在使用 SQL 时，当你告诉 DBMS 从表中删除一组记录时，它会执行你的命令而不提问。也就是说，当你用 SQL 的 DELETE 命令删除记录时，它确实已经执行了删除操作。

在第 11 天中我们将会学习到事务处理控制，事务控制是一种数据库处理机制，它允许编程人员确认或撤消对数据库的改变。该操作对于在线事务处理程序中采用批处理方式对数据库进行改动时非常有效，然而如果同一时间又有其他的用户也在进行数据修改操作时将会导致引用完整性错误。现在，假设不存在事务处理机制。

注：对于一些解释器，例如 ORACLE，当你在即出 SQL 的时候会自动地调用确认操作。

通过 DELETE 语句和 WHERE 子句，DELETE 语句可以完成下边的工作。

- 删除单一的行。
- 删除多个行。
- 删除所有的行。

- 什么也不删除。

在使用 DELETE 语句时需要注意以下几点：

- DELETE 不能删除个别的字段，它对于给定表只能整个记录整个记录地删除。
- 与 INSERT 和 UPDATE 一样，删除一个表中的记录可能会导致与其它表的引用完整性问题。当对数据库进行修改时一定在头脑中有这个概念。
- DELETE 语句只会删除记录，不会删除表。如果要删除表需使用 DROP TABLE 命令（参见第 9 天）。

例 8.7：

下例显示了如何删除 COLLECTION 表中的 WORTH 小于 275 的所有记录。

INPUT：

```
SQL> DELETE FROM COLLECTION WHERE WORTH < 275;
```

4 rows deleted.

之后表的内容如下：

INPUT/OUTPUT：

```
SQL> SELECT * FROM COLLECTION;
```

ITEM	WORTH	REMARKS
NBA ALL STAR CARDS	300	SOME STILL IN BIKE SPOKES
STRING	1000	SOME DAY IT WILL BE VALUABLE

注：与 UPDATE 语句一样，如果你省略了 WHERE 子句，那么表中的所有记录都会被删除。

例 8.8 中则使用了三种数据操作语句来完成一个数据操作过程。

例 8.8：

INPUT：

```
SQL> INSERT INTO COLLECTION VALUES('CHIA PET', 5, 'WEDDING GIFT');
```

OUTPUT：

1 row created.

INPUT：

```
SQL> INSERT INTO COLLECTION
```

```
2 VALUES('TRS MODEL III', 50, 'FIRST COMPUTER');
```

OUTPUT：

1 row created.

现在，建立一个新表并向其中复制数据：

INPUT/OUTPUT：

SQL> CREATE TABLE TEMP (NAME CHAR(20), VALUE NUMBER, REMARKS CHAR(40));

Table created.

INPUT/OUTPUT：

SQL> INSERT INTO TEMP(NAME, VALUE, REMARKS)

2 SELECT ITEM, WORTH, REMARKS FROM COLLECTION;

4 rows created.

INPUT/OUTPUT：

SQL> SELECT * FROM TEMP;

NAME	VALUE	REMARKS
NBA ALL STAR CARDS	300	SOME STILL IN BIKE SPOKES
STRING	1000	SOME DAY IT WILL BE VALUABLE
CHIA PET	5	WEDDING GIFT
TRS MODEL III	50	FIRST COMPUTER

现在，改变其中的数值：

INPUT/OUTPUT：

SQL> UPDATE TEMP SET VALUE = 100 WHERE NAME = 'TRS MODEL III';

1 row updated.

INPUT/OUTPUT：

SQL> UPDATE TEMP SET VALUE = 8 WHERE NAME = 'CHIA PET';

1 row updated.

INPUT/OUTPUT：

SQL> SELECT * FROM TEMP;

NAME	VALUE	REMARKS
NBA ALL STAR CARDS	300	SOME STILL IN BIKE SPOKES
STRING	1000	SOME DAY IT WILL BE VALUABLE
CHIA PET	8	WEDDING GIFT
TRS MODEL III	100	FIRST COMPUTER

然后将这些数据更新回原始表中：

INPUT:

```
INSERT COLLECTION SELECT * FROM TEMP;
```

```
DROP TABLE TEMP;
```

分析:

关于 CREATE TABLE 和 DROP TABLE 语句将在第 9 天作详细讨论。现在，这些语句的作用基本上与它们的名字是一样的，CREATE TABLE 会按照你给的格式建立一个新表。而 DROP TABLE 则会删除表，切记 DROP TABLE 会删除中而 DELETE 只会删除表中的记录。

为了验证你的工作，你可以选出 COLLECTION 表中的内容，你会看到你改动后的结果。

INPUT/OUTPUT:

```
SQL> SELECT * FROM COLLECTION;
```

NAME	VALUE	REMARKS
NBA ALL STAR CARDS	300	SOME STILL IN BIKE SPOKES
STRING	1000	SOME DAY IT WILL BE VALUABLE
CHIA PET	8	WEDDING GIFT
TRS MODEL III	100	FIRST COMPUTER

分析:

上边的例子使用了所有的操作数据的三个语句，INSERT、UPDATE 和 DELETE 来对一个表完成一组操作。DELETE 是这三个语句中最容易使用的。

警告：切记对数据的操作可能会导致引用完整性问题，你要认真地检查数据库中所有表的所有记录以确保正确无误。

从外部数据源中导入和导出数据

INSERT、UPDATE 和 DELETE 语句对于数据库程序而言是非常有用的，它与 SELECT 语句一起为你将要进行的其它数据操作奠定了基础，然而，SQL 作为一种语言并不提供从外部数据源中导入和导出数据的方法。例如：以前你的办公室用了多年的 dBASE 数据库而现在不准备再使用了。你的领导想把它转为具有服务器/客户机功能的 ORACLE 的 RDBMS 系统。显然，INSERT、UPDATE 和 DELETE 语句将会帮助你完成移植工作，前提是你想输入 300,000 个记录。幸运的是 ORACLE 和其他的数据库制造商已经为你提供了完

成这种任务的工具。

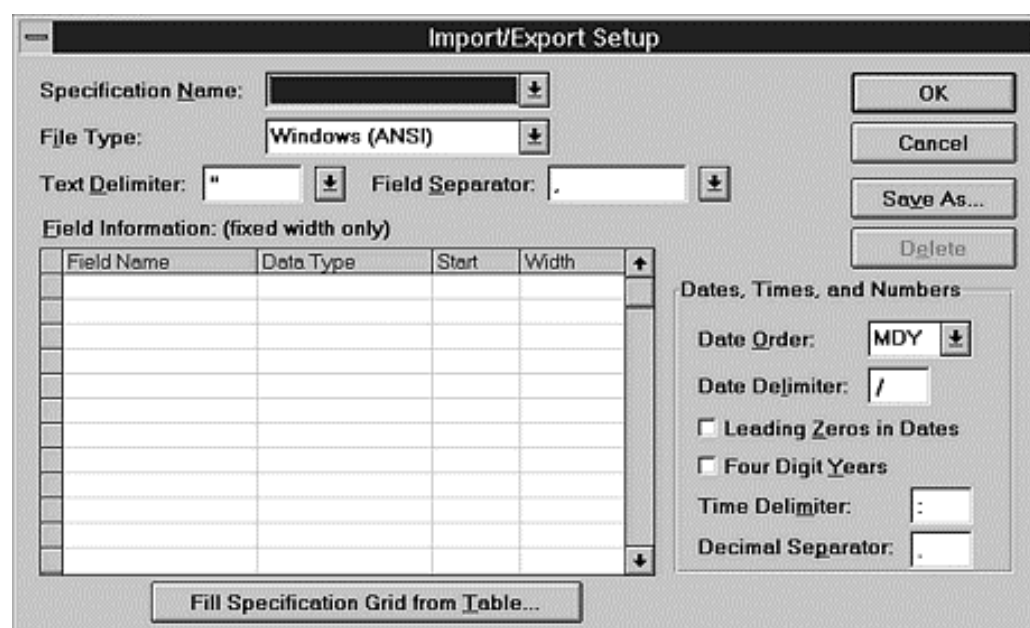
几乎所有的数据库系统都可以导入或导出 ASCII 码的文本文件，尽管 SQL 语言没有这个功能，SQL 不会做得比从一个空数据库开始更好。我们可以使用下列产品中的导入和导出工具，它们是：Microsoft Access，Microsoft 和 Sybase SQL Server 以及 Personal Oracle7。

Microsoft Access

Microsoft Access 是一个其于 PC 的数据库产品，它具有许多关系数据库管理系统的特点，Microsoft Access 也有强大的报表功能、与 Visual Basic 类似的宏语言以及从其他的数据库系统和文本文件中导入或导出数据的能力，本部分讲最后边的一种——从分界文本文件中导入或导出数据，分界的意思就为每个字段采用特殊的分界符来划定界限，一般常用的分界符是逗号、引号和空格。

Access 允许你从其他的数据库系统中导入和导出数据，这些数据库系统包括 dBASE FoxPro，SQL 数据库，其中 SQL 数据库其实是 ODBC 数据链接（Microsoft ODBC 将在第 13 天的《高级 SQL 主题》中提到）本例中我们讨论文本文件的导入导出过程。

在打开 Access 数据库以后（使用文件|打开），选择导出，此时将会出现一个对话框。选择文本文件（宽度固定）选项，Access 将会把数据库中的表导出到一个每一个数据类型都有固定宽度的文本文件中。例如：如果字符字段宽度为 30 就会向文本文件中输出一个长为 30 的字符串，如果字符串不足 30，就会用空格补足。最后，会问你文本文件的存放地点，下图显示了导入/导出对话框。



注意在这个对话框中你可以选择文本宽度和字段分隔符，最后一步是保存与使用有关的注释内容，注释会存储在数据库的内部。

Microsoft and Sybase SQL Server

Microsoft 与 Sybase 公司共同开发了新一代的功能强大的服务/客户数据库系统，它就是 SQL Server，Microsoft 已经同意了在一些平台上开发 RDBMS 版本，SyBase 则在其他平台上已经开发了他的数据库版本（通常是大型化的），虽然在近几年协议有所改变，但提到这个协议可以让我们避免对近几年的数据库系统的版本混淆。

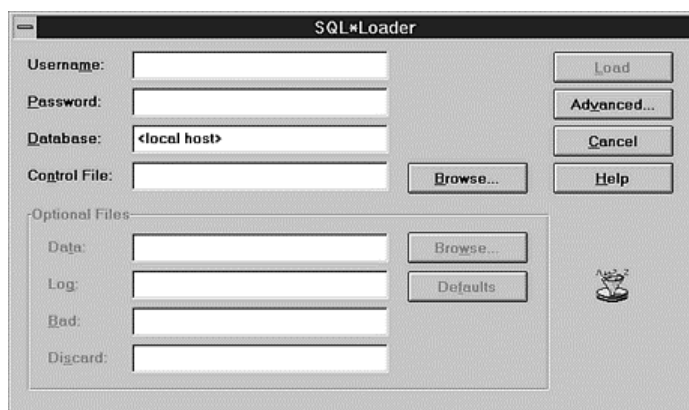
SQL Server 提供的数据库导入和导出的工具叫 BCP，BCP 是“BULK COPY”的缩写。它的主要内容与 ACCESS 的相同，但是不幸的是 BCP 需要你在提示符下输入命令而不是在窗口中使用对话框。

BCP 可以导出固定宽度的文本文件，在 SQL Server 中使用它导入文件要比使用 ACCESS 直接，但 ACCESS 更容易使用。BCP 使用格式化文本（通常扩展名为.FMT）来存储导出的说明。说明文件告诉 BCP 导出文件的列名，字段宽度以及字段分隔符。你可以当数据库建好后在 SQL 内部使用 BCP 来建立一个关于数据库结构的说明。

Personal Oracle7

Personal Oracle7 允许你导出文本文件，文本文件的字段宽度与源库定义的字段宽度相同。使用的工具是 SQL*Loader。这个图形工具使用一个控制文件（扩展名为.CTL）。这个文件类似与 SQL Server 的格式文件（.FMT），它的内容是告诉 SQL*Loader 数据文件的位置。

SQL*Loader 的界面见下图：



总结

SQL 对于操作数据提供了三条语句。

INSERT 语句有二个变体，INSERT VALUES 可以插入一个记录，而 INSERT SELECT 则可以根据给定的一个或多个表来插入一个或多个记录，SELECT 语句可以归并多个表，并把归并的结果加入到另外一个表中。

UPDATE 语句可以改变符合条件列的值，UPDATE 语句可以用计算或表达式的结果作为需要更新的内容。

DELETE 语句是这三个语句中最简单的，它会删除符合 WHERE 条件的所有记录，如果没有 WHERE 子句，它会删除表中的所有记录。

现代的数据库系统提供了许多的数据库操作的工具，其中一些工具可以让开发人员从外部数据源中导入或导出数据，这在当数据库向大系统或小系统上移植时非常有用，Microsoft Access，Microsoft 和 Sybase 的 SQL Server 以及 Personal Oracle7 都提供了这样的移植工具。

问与答

问：SQL 有导入或导出操作的语句吗？

答：没有，这一操作是解释器的附加功能，也就是说 ANSI 委员会允许制造商进行他们认为是需要的工作。

问：我可以使用 INSERT 语句从一个表中拷贝数据到它自身吗？我想复制所有记录的内容而只对其中一个字段的内容作更改。

答：不能，INSERT 语句中的表不能与 FROM 中的表相同，但是，你可以将它复制到一个临时的表格中（这将在第 14 天讨论），然后对临时表格的内容作修改后再将它复制回原始表。要检查你的表的具有唯一值属性的字段，唯一值属性将只允许一个数值在该列中出现一次。

问：我注意到了关于 INSERT、UPDATE 和 DELETE 语句的警告。是否我可以对我所犯的错误进行修正，如果是的话，是那条语句可以完成这种工作？

答：是的，例如：你可以使用 ROLLBACK 来撤消 INSERT、UPDATE 或 DELETE 的操作。

但是如果你向表中插入了多个记录后没有发现错误就使用了 COMMIT 命令，几个星期后别人发现了错误，这时你可能要花上两个星期的时间来对数据库的数据进行逐条的检查，大多数情况下你可能不知道错在哪里，所以你只好恢复数据库。

校练场

1、下边的语句有什么错误？

```
DELETE COLLECTION;
```

2、下边的语句有什么错误？

```
INSERT INTO COLLECTION SELECT * FROM TABLE_2
```

3、下边的语句有什么错误？

```
UPDATE COLLECTION ("HONUS WAGNER CARD", 25000, "FOUND IT");
```

4、如果执行下边的语句会有什么结果？

```
SQL> DELETE * FROM COLLECTION;
```

5、如果执行下边的语句会有什么结果？

```
SQL> DELETE FROM COLLECTION;
```

6、如果执行下边的语句会有什么结果？

```
SQL> UPDATE COLLECTION SET WORTH = 555
```

```
SET REMARKS = 'UP FROM 525';
```

7、下边的语句是否会工作？

```
SQL> INSERT INTO COLLECTION SET VALUES = 900 WHERE ITEM = 'STRING';
```

8、下边的语句是否会工作？

```
SQL> UPDATE COLLECTION SET VALUES = 900 WHERE ITEM = 'STRING';
```

练习

1、试着向一个表中插入一个不正确的数据类型，看一下出错信息，然后再插入一个正确的数据类型。

2、试着使用你的数据库系统将某个表导出为其他库格式，然后再把它导入。熟悉一下你的数据库系统的导入与导出操作。并试着用其它数据库操作导出文件。

第九天：创建和操作表

目标：

在今天我们将学习创建数据库的知识，在第九天我们将学习 CREATE DATABASE 、 CREATE TABLE、 ALTER TABLE、 DROP TABLE 以及 DROP DATABASE 语句。它们被子统称为数据定义语句，（与之相对应，SELECT、 UPDATE、 INSERT、 DELETE 被称为数据操作语句）。到今天结束之时，我们将学会以下内容：

- 建立关键字段
- 建立一个数据库以及在其中建表
- 表的建立、修改与删除
- 向数据库中添加数据
- 修改数据库中的数据
- 删除数据库

从它的语法开始，你现在已经知道了许多与 SQL 有关的词汇并且知道对 SQL 的一部分已经有了详尽的了解。在第二天的《查询简介：SELECT 语句的使用》中我们学会了如何从数据库中检索数据，在第 8 天《操作数据》中我们学习了如何从数据库中添加、更新和删除数据。现在，在第九天的学习过程中，你将会知道数据库是怎么来的。为了简单起见，我们忽略了数据库和表的建立过程。我们假定这些东西已经存在于你们的系统之上，现在我们来建立它。

CREATE 语句的语法可以非常简单或复杂，它依据你的数据库管理系统的支持和你对所要建立的数据库要求而定。

注：今天的例子使用的是 PERSONAL ORACLE 7，请查看你所用的解释器的文档资料以了解它们与本书中的例子在语法细节上的不同。

CREATE DATABASE 语句

在任何数据库项目中管理数据的第一步工作就是建立数据库，根据你的要求和你的数据库管理系统的情况，这个工作可以很简单也可以很复杂。许多现代的数据库系统（包括

PERSONAL ORACLE 7) 都提供了图形工具例你可以通过按鼠标按键来完成数据库的建立工作。这对于节省时间是相当有益处的，但是你应该知道 SQL 是如何响应鼠标的操作的。

典型的数据库创建语句如下：

SYNTAX:

```
CREATE DATABASE database_name
```

由于语法根据系统的不同差别很大，所以我们不对 CREATE 的语法作更深入的讨论。许多数据库系统甚至不支持数据库的创建命令。但是，几乎所有的流行了，功能强大的关系型数据库系统都支持它。所以我们用创建数据库的选择来代替创建数据库命令的语法讨论工作。

建立数据库时的选项

CREATE DATABASE 语法的差别很大，许多 SQL 教材中都没有提到它而是直接进行了 CREATE TABLE 语句。由于你在创建表之前必须先创建数据库。而当一个开发人员在建立数据库时必须要考虑一些事情。第一件事情就是你的权限级别，如果你使用的关系型数据库管理系统支持权限，你必须确认你是否具有系统管理设置权限或系统管理员已经允许你有创建数据库的权限，请参见你的数据库文档以获得更多的知识。

大多数关系型数据库系统允许你指定数据库的默认大小，它通常受你的硬盘容量的限制。你需要清楚你所使用的数据库系统是如何将数据根据你指定的大小存在你的硬盘上的。对空间的管理责任由每个系统管理人员负责，所以你的本地的数据库管理人员为你建立一个测试用的数据库是可能的。

不要对 CREATE DATABASE 过于关心，它非常简单，你可以像下边这样来建立一个名字叫 PAYMENTS 的数据库。

SYNTAX:

```
SQL> CREATE DATABASE PAYMENTS;
```

注：再一次请你参考你所使用的数据库管理系统的说明书来建立数据库，因为 CREATE DATABASE 语句在不同的解释器之间的差别是很大的，每一种解释器都有它自己的一些特点。

设计数据库

设计数据库对于成功的应用来说是非常重要的。它将涉及到我们在第一天学习的关系型数据库理论和标准化数据库理论。

标准化过程要求你将数据分解成不同的部分以减少数据的冗余度。规划你的数据将是一个非常复杂的过程，有相当多的数据库设计工具可以让你将这一过程变得更合乎逻辑。

许多因素会对你设计数据库造成影响，如下边的：

- 安全问题
- 磁盘的可用空间
- 数据检索及响应的速度
- 数据更新的速度
- 多表归并返回数据的速度
- RDBMS 对临时表的支持

磁盘空间是一个最为重要的因素，尽管你可能会认为当磁盘的容量以 GB 计时这是不重要的。不要忘记你的数据库越大，返回数据库时间就越长。如果你的表的设计工作做得非常糟糕，那么你可能会在其中存入许多无用的数据。

相反的问题也会产生，你可能已经建立了相当多的表所以看起来它们已经非常的合乎标准化的要求了，尽管你的数据库结构非常合理，但是在这个数据库上进行的查询工作将会花费相当多的时间。数据库的设计风格有时会因为数据结构并不能清楚地表达设计人的意图而使维护工作难于进行。所以当你在进行代码和数据库的结构设计时将你的数据结构与你当时的意图进行详细的记录是非常重要的，在数据库设计行业中，它被称为数据字典。

建立数据字典

数据字典是数据库设计人员非常重要的文档资料，它可以有以下功能：

- 数据库的设计意图、谁将会使用它。
- 数据库的自身资料：用什么创建的数据库，数据库的大小是多少，日志文件的尺寸是多少（在一些 RDBMS 中它存储着数据库的操作信息）。
- 任何数据库安装或反安装的 SQL 原代码记录，包括导入/导出数据库的文档资料记录，就像第八天所说的那样。

- 对每一个表的详细描述以及它的最终目的是什么。
- 每一个表的内部结构的资料，包括表中的所有字段，数据类型以及注释情况以及所有的索引和所有的视图。（见第 10 天《创建视图及索引》）
- 对于每一个存储过程的原代码和触发机制。
- 说明数据库是否具有唯一值及非空值约束，并说明这些约束是关系型数据库管理系统强加的还是数据库编制人员设定的，以及约束在原代码中的作用范围。

许多计算机辅助软件工程工具会在建立数据字典的过程中为你提供帮助，例如：MicroSoft Access 在将数据库打包会生成一个文档，在文档中对数据库中的每一个细节都进行了详细的描述。在第 17 天中的《使用 SQL 来生成 SQL 语句》中你了解数据字典的详细内容。

注：大多数 RDBMS 中都提供了生成数据字典的工具包或有说明来告诉你如何安装它。

建立关键字段

在接下来的数据库设计工作中，最主要的目标就是建立你的表的结构，它包括主关键字和外关键字。其中主关键字用于完成下列目标：

保证表中的第一条记录都是唯一的（没有一条记录的内容完全与另一条相同，至少主键不能相同）

对于一个特定的记录，它的所有列都是必须的（列的内容不应出现重复）

在第二个目标中，如果列的内容在表中从头至尾都没有重复，那它就是主关键字。外关键字则是在自己的关系中不唯一标识记录，但在其它关系中可用作对匹配字段链接的一种关键字。下边的例子可以帮助你来区分这两种情况：

假定你有三个表，BILLS，BANK_ACCOUNTS，COMPANY。它们的结构如下：

Table 9.1. Table structure for the PAYMENTS database.

Bills	Bank_Accounts	Company
NAME, CHAR(30)	ACCOUNT_ID, NUMBER	NAME, CHAR(30)
AMOUNT, NUMBER	TYPE, CHAR(30)	ADDRESS, CHAR(50)
ACCOUNT_ID, NUMBER	BALANCE, NUMBER	CITY, CHAR(20)
	BANK, CHAR(30)	STATE, CHAR(2)

在 Bills 表中的主关键字段为 NAME 字段，这一字段是没有重复的，因为你只可能有一个支票的来对应 AMOUNT（实际上你可能通过支票号以及日期来保证它的唯一性，但

在这里我们假定 NAME 能做到这一点)，而 ACCOUNT_ID 字段则是 BANK_ACCOUNT 的主关键字，COMPANY 表中的 NAME 字段是主关键字。

在本例中外部关键字应该是很容易发现的，在 BILL 表中的 ACCOUNT_ID 与 BANK_ACCOUNT 表相关连，而 NAME 字段则将 BILL 表与 COMPANY 表相关连。如果这是一个完备的数据库，你可能会有更多的表和数据分类。例如：BANK 字段在 BANK_ACCOUNT 表中可以说明一个银行的信息如地址和电话号码等。COMPANY 表可以与其它表或（数据库事件）相关连以得到公司及有关产品的信息。

例 9.1:

同样用 BILLS，BANK_ACCOUNTS 和 COMPANY 表，我们来看一个不正确的数据库设计。初学者可能犯的错误就是不会正确地对数据进行尽可能的逻辑分组，一个不好的 BILL 设计如下：

Column Names	Comments
NAME, CHAR(30)	Name of company that bill is owed to
AMOUNT, NUMBER	Amount of bill in dollars
ACCOUNT_ID, NUMBER	Bank account number of bill (linked to BANK_ACCOUNTS table)
ADDRESS, CHAR(30)	Address of company that bill is owed to
CITY, CHAR(15)	City of company that bill is owed to
STATE, CHAR(2)	State of company that bill is owed to

结果看上去是正确的，但是这只是在数据输入的开始时。在几个月以后，你已经在 NAME 字段中输入了许多公司的账单，每当向 BILL 表中加入一个新记录的时候，公司的 ADDRESS、CITY 以及 STATE 就有可能出现重复。当记录增加到成千上万时重复的数据也在相应的增加，类似的情况也可能在 10、20 或 30 个表中出现。你现在知道数据库标准化设计的重要性了吧？

在你向表中输入数据之前，你应该知道如何来创建一个表。

CREATE TABLE 语句

建立表的过程比建立数据库的过程更不标准，它的基本语法如下：

SYNTAX:

```
CREATE TABLE table_name      (field1 datatype [ NOT NULL ],
                                field2 datatype [ NOT NULL ],
```

field3 datatype [NOT NULL]...)

它的一个简单的例子如下：

INPUT/OUTPUT：

```
SQL>CREATE TABLE BILLS (  
2  NAME CHAR(30),  
3  AMOUNT NUMBER,  
4  ACCOUNT_ID NUMBER);
```

Table created.

分析：

该语句创建了一个名字叫 BILL 的表，在 BILL 表中有三个字段，NAME、ACCOUNT 和 ACCOUNT_ID。其中 NAME 字段为字符类型可以存储长度 30 的字符串，而 AMOUNT 和 ACCOUNT_ID 则只参存储数字。

下边的部分对 CREATE TABLE 命令作进一步的解释。

表名

当使用 PERSONAL ORACLE 来创建一个表的时候，对表的命名要遵从几个约束。首先，表的名字不得超过 30 个字符长。由于 ORACLE 对大小写不敏感，所以在写名字时你可以根据需要采用大写或小写的方式。但是表的第一个字符必须是字母（A-Z），其余的字符则还可以有下划线，#，\$，@。当然，在本工程中表的名字不应该有重复，表的名字也不可以是 ORACLE 的保留字（如 SELECT）。

注：你可以在不同的所有者或工程中使用相同的表的名字，但在同一个工程中表的名字必须保证唯一。

FIRST NAME

如果你有过用任何一种语言进编程的经验，你会有类似的关于数据类型的概念。指定的字段只能存放特定的数据类型。例如：字符型字段只能存放字符型数据类型。表 9.2 显示了 ORACLE 支持的数据类型。

ORACLE 所支持的数据类型

数据类型	说 明
CHAR	可以存储长度为 1~255 个字符的字符串，空格会被填充到字符串的右边以保证其内容满足定义的长度。
DATE	包括日期的世纪、年、月、日时、分、秒
LONG	可以支持长达 2G 的字符串（见下注）
LONG RAW	可以存储长达 2G 的二进制内容（见下注）
NUMBER	零、正值或负值的定点或浮点数
RAW	可以存储长不过 255 个字节的二进制代码
ROWID	用一个十六进制的数来标明当前行在表内的唯一地址（见下注）
VARCHAR2	变长的字母或数字，长度可以从 1 到 2000。

注：LONG 数据类型在其它的数据库系统中常被称为备注类型，它主要用于存储大量的可以在稍后返回的文本内容。

LONG RAW 类型在其它数据库系统中常被称为大二进制类型（BLOB），它可以用来存储图形，声音、视频数据，尽管关系型数据库管理系统最初不是为它们而设计的，但是多媒体数据可以存储在 BLOB（或 LONG RAW）类型的字段内。

ROWID 常用在可以将你的表中的每一条记录都加以唯一标识的场合，许多关系型数据库管理系统用 COUNTER（如 ACCESS）或 IDENTITY（SQL SERVER）来表达这个概念。

注：请检查你的解释器看它们是否对数据类型的支持有所变化。

空值属性

SQL 也可以让你鉴别在一个列中是否已经存入的数值，NULL 只是一个修饰，因为如果一个字段的内容为 NULL 的话实际上是说这个字段中没有东西也没有。

在建立表的时候，大多数数据库管理系统允许你用 NOT NULL 来指明字段是否为非空属性，NOT NULL 的意思就是在当前表的该字段中不能有任何记录存在空值，也就是说在当前表中的该字段的每一个记录中都应该确实存在数值。下例给出了 NOT NULL 的用法。

INPUT:

```
SQL>CREATE TABLE BILLS (
2  NAME CHAR(30) NOT NULL,
```

```
3  AMOUNT NUMBER,  
4  ACCOUNT_ID NOT NULL);
```

分析：

在这个例子中如果你想把公司的账转到你自己的名下，如果 NAME 区和 ACCOUNT_ID 区没有内容，那么这种存储是没有任何意义的，你也许可以在记录中给出账单号，但是无法收取。

下例中的第一个语句插入正确的数据以便为支付 JOE 的\$25 电脑服务费。

INPUT/OUTPUT：

```
SQL> INSERT INTO BILLS VALUES("Joe's Computer Service", 25, 1);  
  
1 row inserted.
```

INPUT/OUTPUT：

```
SQL> INSERT INTO BILLS VALUES("", 25000, 1);  
  
1 row inserted.
```

分析：

注意，在上边的第二个例子中没有给出 NAME 的名字，（你也许会认为这是一件好事，因为没有收款人却收取了 25000 元，但是我们不这样认为）。如果 NAME 字段在创建时指定的非空属性，那么在第二个例子中就会产生一个错误。

唯一属性

你在设计表时的一个目标是要保证在表中有一个列的值是唯一的。这列或这个字段被称为关键字，在一些数据库管理系统中允许你将某一列设成唯一值属性，如 ORACLE 和 SQL Server，可以让你对一个字段加以唯一值索引（见第 10 天），这一特性可以保证你不在该字段中插入重复的数值。

在选择主关键字段时有几个需要注意的问题，我们曾经说过，ORACLE 提供了一个 ROWID 字段，它对于每一行均会自动递增。因此默认情况下它总是一个唯一的数值，将 ROWID 字段作为主关键字有许多理由。首先，对于整数值的归并操作要远远快于对一个长度为 80 个字符的字符串的归并操作。因为整数的存储长度小于字符串，所以最终归并的结果集也将小于字符串的归并结果集。此外的好处是使用 ROWID 字段你可以看到表的组织情况，而字符则会产生数字输入的问题。例如，当一个人输入了 111 First Street 而

另一人输入了 111 1st Street 时会有什么情况发生，如果又有一个输入了 111 1st St.呢？在今天的图形用户界面环境下，正确的字符串会被输入到一个列表框中，当用户从列表框中选择的时候，代码会将字符串变换成为一个唯一的 ID 号并将这个号码存储在数据库中。

到现在为止，你可以用你在今天所学过的东西来创建一个表了。随后我们将在今天使用这些表，所以你应该在表中输入一些数据。使用昨天的 INSERT 命令可以向表中加入表 9.3，9.4 和 9.5 中的数据。

INPUT/OUTPUT:

```
SQL>create database PAYMENTS;
```

Statement processed.

```
SQL>create table BILLS (
```

```
2  NAME CHAR(30) NOT NULL,
```

```
3  AMOUNT NUMBER,
```

```
4  ACCOUNT_ID NUMBER NOT NULL);
```

Table created.

```
SQL> create table BANK_ACCOUNTS (
```

```
2  ACCOUNT_ID NUMBER NOT NULL,
```

```
3  TYPE CHAR(30),
```

```
4  BALANCE NUMBER,
```

```
5  BANK CHAR(30));
```

Table created.

```
SQL> create table COMPANY (
```

```
2  NAME CHAR(30) NOT NULL,
```

```
3  ADDRESS CHAR(50),
```

```
4  CITY CHAR(30),
```

```
5  STATE CHAR(2));
```

Table created.

Table 9.3. Sample data for the BILLS table.

Name	Amount	Account_ID
Phone Company	125	1
Power Company	75	1

Name	Amount	Account_ID
Record Club	25	2
Software Company	250	1
Cable TV Company	35	3

Table 9.4. Sample data for the BANK_ACCOUNTS table.

Account_ID	Type	Balance	Band
1	Checking	500	First Federal
2	Money Market	1200	First Investor's
3	Checking	90	Credit Union

Table 9.5. Sample data for the COMPANY table

Name	Address	City	State
Phone Company	111 1st Street	Atlanta	GA
Power Company	222 2nd Street	Jacksonville	FL
Record Club	333 3rd Avenue	Los Angeles	CA
Software Company	444 4th Drive	San Francisco	CA
Cable TV Company	555 5th Drive	Austin	TX

表的存储与尺寸的调整

大多数 RDBMS 都设定了表的默认大小和存储的定位，如果你没有指定表的大小和存储大小它就会采用默认值。它可能是非常不合适的，特别对于大型的表来说更是如此，默认大小根据解释器和不同而不同，下边是一个在创建表时使用 STORAGE 子句的例子。（对于 ORACLE 而言）

INPUT:

```
SQL> CREATE TABLE TABLENAME
2  (COLUMN1    CHAR    NOT NULL,
3  COLUMN2    NUMBER,
4  COLUMN3    DATE)
5  TABLESPACE TABLESPACE NAME
6  STORAGE
7  INITIAL SIZE,
8  NEXT SIZE,
9  MINEXTENTS value,
```

```
10    MAXEXTENTS value,  
11    PCTINCREASE value);
```

OUTPUT:

Table created.

分析:

在 ORACLE 中你可以指定需要存放的表的大小，定夺的依据是可用空间的大小，经常是由数据库管理人员来决定。INITIAL SIZE 表的初始长度（最初的分配空间），NEXT SIZE 是指追加的长度，MINEXTENTS 和 MAXEXTENTS 用于指定表的最小和最大长度，PCTINCREASE 则指明表每次追加的百分比或进行下一次追加。

用一个已经存在的表来建表

CREATE TABLE 是最为通用的建表的方法。然而，在一些数据库管理系统中提供了一种可供选择的方法——使用已经存在的表中的格式和数据。当你对表进行临时改动需要将数据选出时这种方法是很有用的。当你要创建的表与已有的表类似并且其内容也类似时它也非常有用（你不必须重新输入这些信息）。在 ORACLE 中它的语法如下：

SYNTAX:

```
CREATE TABLE NEW_TABLE(FIELD1, FIELD2, FIELD3)  
    AS (SELECT FIELD1, FIELD2, FIELD3  
    FROM OLD_TABLE <WHERE...>
```

它的语法允许你建立一个字段类型与已有表中选出的字段类型相同的新表，你也可以对新表中的字段进行重命名。

INPUT/OUTPUT:

```
SQL> CREATE TABLE NEW_BILLS(NAME, AMOUNT, ACCOUNT_ID)  
    AS (SELECT * FROM BILLS WHERE AMOUNT < 50);  
Table created.
```

分析:

上边的语句用 BILL 表中 AMOUNT 小于 50 的记录创建了一个新表。

在一些数据库系统中你也可以使用下边的语法:

SYNTAX:

```
INSERT NEW_TABLE
```

```
SELECT <field1, field2... | *> from OLD_TABLE <WHERE...>
```

上边的语法格式将会严格地按照原有表的字段格式和数据建立一个新表。下边用 SQL Server 的 Transact-SQL 来对它进行举例。

INPUT:

```
INSERT NEW_BILLS
```

```
1> select * from BILLS where AMOUNT < 50
```

```
2> go
```

go 语句在 SQL SERVER 中是处理 SQL 缓冲区内指令的命令，它的作用等同于在 ORACLE 中的分号。

ALTER TABLE 语句

没有道理为每件事对你的数据库进行多次的设计，但又确实需要对数据库和应用程序进行改动。那么 ALTER TABLE 语句可以让数据库的设计者或设计人员在表创建以后修改它的结构。

ALTER TABLE 语句可以帮助你做两件事：

- 加入一列到已经存在的表中。
- 修改已经存在的表中的某一列。

ALTER TABLE 语句的语法如下：

SYNTAX:

```
ALTER TABLE table_name <ADD column_name data_type; |  
MODIFY column_name data_type;>
```

下边的命令会将 BILL 表中的 NAME 字段改为长度 40 个字符。

INPUT/OUTPUT:

```
SQL> ALTER TABLE BILLS MODIFY NAME CHAR(40);
```

```
Table altered.
```

注：你可以增加或减少某一列的长度，但是你不能将它减少到使修改后的长度小于其中的已有数据长度。

下边的语句是向 NEW_BILLS 表中加入一个新列：

INPUT/OUTPUT:

```
SQL> ALTER TABLE NEW_BILLS
```

```
2 ADD COMMENTS CHAR(80);
```

Table altered.

分析:

这条语句会加入一个叫 COMMENTS 的长度为 80 个字符的字段，该字段会加在已有字段的右边。

在使用 ALTER TABLE 时会有许多限制，你不能用它来对一个数据库增加或删除字段。它可以将一个列由 NOT NULL 改变为 NULL 而不必使用其它方法。但是如果想把列由 NULL 改变为 NOT NULL 时则要求指定的字段中不能有 NULL 值。想把某一列由 NOT NULL 改变为 NULL 可以使用下边的语法。

SYNTAX:

```
ALTER TABLE table_name MODIFY (column_name data_type NULL);
```

如果能把一列由 NULL 改变为 NOT NULL，你必需经过以下步骤：

- 1、确认要改变的列中有没有 NULL 值。
- 2、删掉你所发现的任何 NULL 值（删除该记录、更新这一记录等等）。
- 3、使用 ALTER TABLE 命令。

注：在一些数据库管理系统中允许使用 MODIFY 子句，另外一些则不可以。而又有
一些在 ALTER TABLE 中加入了其它的子句。在 ORACLE 中，你甚至可以修
改表的存储参数，请检查你的解释器以找出它对 ALTER TABLE 的确实用法。

DROP TABLE 语句

SQL 提供了一个可以从数据库去彻底地移去某个表的命令，DROP TABLE 可以从数据库中删除一个指定的表以及与之相关联的索引和视图（在第 10 天会进行更详细的讨论）。一旦这个命令发出以后，就没有办法可以彻消它。它最常用在你创建一个临时表，并且已经进行完毕了你的全部计划工作的时候。DROP TABLE 语句的语法格式如下：

SYNTAX:

```
DROP TABLE table_name;
```

下边是如何删掉一个叫 NEW_BILLS 表的实例：

INPUT/OUTPUT:

SQL>DROP TABLE NEW_BILLS;

Table dropped.

分析:

请注意，系统没有给出你提示，该命令不会问你 Are you sure? (Y/N)。但是删除操作已经执行，表已经永远地删除了。

警告:

如果你执行了

SQL> DROP TABLE NEW_BILLS;

如果你想正确地删除一个表，那么在删除表的时候最好给出它的所有都（所属的数据库）或工程的名字，推荐的使用方法如下：

SQL> DROP TABLE OWNER.NEW_BILLS;

我只所以强调这种使用方法是因为我曾经挽救过一个被错误删除了的表格，它被删除的原因是因为没有准确地给出其所属工程的名字，修复那个数据库用了八个小时，我们一直工作到了深夜。

DROP DATABASE 语句

一些数据库管理系统也提供了删除数据库（DROP DATABASE）的语句，它的使用方法与 DROP TABLE 相同，语法如下：

DROP DATABASE database_name

请不要删掉 BILLS 数据库，我们在今天的后边还要用它，而且在第 10 天也要用。

注：不同的关系数据库解释器提供了不同的删除数据库的方法，在数据库被删除以后，我们需要清理掉操作系统用以构建数据库的文件。

练习 9.2

创建一个数据库并在其中创建一个表，试验 DROP TABLE 和 DROP DATABASE 命令。数据库系统是否会允许你这样做？单文件的数据库系统如 ACCESS 是不支持这个命令的。数据库包含在一个单一的文件中，如果想建立一个数据库，你必须用系统提供的菜单选项，如果想删掉它，只需简单地从系统中删掉这个文件就可以了。

总结

第 9 天讲述了数据处理语言 (DML) 的主要内容，具体说来是你学习了五个新的语句：CREATE DATABASE、CREATE TABLE、ALTER TABLE、DROP TABLE、DROP DATABASE。在今天的课程中我们也讨论了一个好的数据库设计方案的重要性。

当你在创建并设计一个数据库的时候数据字典是一个重要的文档资料，字典中有对数据库的详细的描述，包括：表、字段、视图、索引、存储过程、触发机制等等。一个完备的数据字典中应该有对数据库中的每一个内容的详细注释。每当你数据库进行过修改以后你都应及时的更新数据字典。

在使用数据库处理语句时，设计一个好的数据库是非常重要的，把数据分组逻辑组并建立主关键字以使其它的逻辑组正确地识别它。可以使用外部关键字来指向该表的主关键字或在该表中用外部关键字与其它的表相关联。

我们已经知道了建立数据库语句不是一个数据库系统的必需内容，因为不同的数据库供应商有不同的数据库组织形式，每一种解释器都有它们自己的特点和选项，从而导致了建立数据库语句的截然不同，只使用 CREATE DATABASE database_name 可以在大多数系统中用默认的参数来创建一个默认的数据库。而 DROP DATABASE 语句则可以永久地删除一个数据库。

使用 CREATE TABLE 语句可以建立一个新的表，使用该命令你可以建立字段并定义它们的数据类型，在一些数据库管理系统中你还可以指定字段的其它属性，例如是否它可以接受空值以及它的内容是否在本表中应该是唯一的。而 ALTER TABLE 语句可以对已存在的表的结构进行修改。DROP TABLE 语句可以永久地删除一个表格。

问与答

问：为什么 CREATE DATABASE 语句在不同的数据库中使用方法是不同的？

答：这是因为不同的数据库系统在建立数据库时的实际过程是不同的，基于 PC 机的小型数据库系统通常依赖文件来建立某些应用程序。而在大型服务器上运行的分布式数据库中的需用数据库文件通常是分布在多个磁盘驱动器上的，当你的代码访问数据库的时候，运行于电脑上的数据库程序不会像访问你的磁盘上的文件那样直接，更大的数据库系统还要对磁盘的空间进行估算以支持一些特性如安全性、

传输控制以及内嵌于数据库的存储过程。当你的程序访问数据库时，数据库的服务程序通常需要对你的请求（通常与其它的请求一起）经过复杂的中间过程才会返回数据。这一主题将在第 3 周讨论。现在，你应该知道为什么不同的数据库系统建立和管理数据库的方法不同的。

问：我能否建立一个临时表并且的工作完成后它会自动地删除？

答：可以，许多数据库管理支持临时表的概念，该类型的表可以在你的过程运行结束或你使用 DROP TABLE 语句后删除，我们将在第 14 天的《动态应用 SQL》中讨论临时表。

问：我是否可以用 ALTER TABLE 语句来删除一个表？

答：不行！该语句只可以用来增加或修改表中的某一列，如果你想删除一列，你可以建立一个新表并将旧表中的数据有选择地复制到新表中然后再删除旧表。

校练场

- 1、ALTER DATABASE 语句经常用在修改已有表的结构上，对不对？
- 2、DROP TABLE 语句与 DELETE FROM <table_name>的作用是相同的，对不对？
- 3、可以使用 CREATE TABLE 命令向数据库中加入一个新表，对不对？
- 4、为什么下边的语句是错误的？

INPUT:

```
CREATE TABLE new_table (  
    ID NUMBER,  
    FIELD1 char(40),  
    FIELD2 char(80),  
    ID char(40);
```

- 5、为什么下边的语句是错误的？

INPUT:

```
ALTER DATABASE BILLS (  
    COMPANY char(80));
```

- 6、当一个表建立时，谁是它的所有者？
- 7、如果字符型列的长度在不断变化，如何才能做出最佳的选择？
- 8、表名是否可以重复？

练习

- 1、用你喜欢的格式向 `BILLS` 数据库中加入两个表，名字分别叫 `BANK` 和 `ACCOUNT_TYPE`。 `BANK` 表中应该包含有 `BANK_ACCOUNT` 表中 `BANK` 字段的信息， `ACCOUNT_TYPE` 表中也应该包含有 `BANK_ACCOUNT` 表中 `ACCOUNT_TYPE` 字段的信息，试着尽可能地减少数据的数量。
- 2、使用你已经创建的五個表， `BILLS`、 `BANK_ACCOUNTS`、 `COMPANY`、 `BANK`、 `ACCOUNT_TYPE`，改为表的结构以用整数型字段作为关键字以取代字符型字段作为关键字。
- 3、使用你所知道的 SQL 的归并知识（见第 6 天：表的归并），写几个查询来归并 `BILLS` 数据库中的几个表。

第 10 天 创建视图和索引

目标

今天我们将要讨论的内容对于一些有 SQL 有一定了解的程序员或数据库管理人员来说可能是新东西。从第 1 天到第 8 天，我们主要学习了如何使用 SQL 在关系数据库中进行基本的工作。在第 9 天我们讨论了数据库的设计，表的创建以及其它的数据处理语句。所有这些内容的对象（表、数据库、记录、字段）的共同之处在于——它们是存在于磁盘上的物理对象。今天我们来学习 SQL 的两个新的特性，它允许你以于数据在磁盘上的存储不同的方式来显示数据。这两个特性就是记录和索引，到今天的结束，你将学习以下内容：

- 如何区别索引与视图
- 如何创建视图
- 如何创建索引
- 如何用视图来修改数据
- 索引可以做什么

视图常常被称为虚表，它是用 CREATE VIEW 语句来建立的。在视图建立以后你可以对视图采用如下命令：

- SELECT
- INSERT
- INPUT
- UPDATE
- DELETE

索引是与磁盘上数据的存储方式不同的另外一种组织数据的方法，索引的特例是表中记录依据其在磁盘上的存储位置显示，索引可以在表内创建一个列或列的组合。当应用索引以后，数据会按照你使用 CREATE INDEX 语句所定义的排序方式返回给用户，通过对正确的、特定的两个表的归并字段进行索引可以获得明显的好处。

注：视图与索引是两个完全不同的对象，但是它们有一点是相同的：它们都与一个表或数据库相关联，尽管每一个对象只能与一个特定的表相关联，但它们还是通过对数据的

预排序和预定义显著地提高了表的工作性能。

注：在今天的例子中我们使用的是 PERSONAL ORACLE 7，对于你使用的解释器请参阅它的文档与找出它们在语法上的不同之处。

使用视图

你可以对封装的复合查询应用视图或虚表。当对一组数据建立视图以后，你可以像处理另外一个表一样去处理视图。但是，在视图中修改数据时要受到一些限制。当表中的数据改变以后，你将会在查询视图时发现相应的改变，视图并不占用数据库或表的物理空间。

CREATE VIEW 和语法如下：

SYNTAX：

```
CREATE VIEW <view_name> [(column1, column2...)] AS
```

```
SELECT <table_name column_names>
```

```
FROM <table_name>
```

与通常一样，语法看起来不太容易使人明白。但是对于今天的内容我们有许多例子来展示视图的用法和优点。该命令通知 SQL 去创建一个视图（用你给出的名字）及其列（如果你想指定的话），SQL 的 SELECT 语句可以判定列所对应的字段及其数据类型。没错，就是九天以来你一直使用的 SELECT 语句。

在你用视图进行任何有用的工作之前，你需要对 BILLS 数据库再添加一些数据。如果你已经用 DROP DATABASE 语句对它进行了试验，那么你需要重新建立它（数据见表 10.1、10.2 和 10.3）。

INPUTOUTPUT：

```
SQL> create database BILLS;
```

```
Statement processed.
```

INPUTOUTPUT：

```
SQL> create table BILLS (
```

```
2  NAME CHAR(30) NOT NULL,
```

```
3  AMOUNT NUMBER,
```

```
4  ACCOUNT_ID NUMBER NOT NULL);
```

Table created.

INPUTOUTPUT:

```
SQL> create table BANK_ACCOUNTS (
2  ACCOUNT_ID NUMBER NOT NULL,
3  TYPE CHAR(30),
4  BALANCE NUMBER,
5  BANK CHAR(30));
```

Table created.

INPUTOUTPUT:

```
SQL> create table COMPANY (
2  NAME CHAR(30) NOT NULL,
3  ADDRESS CHAR(50),
4  CITY CHAR(30),
5  STATE CHAR(2));
```

Table created.

Table 10.1. Sample data for the BILLS table.

Name	Amount	Account_ID
Phone Company	125	1
Power Company	75	1
Record Club	25	2
Software Company	250	1
Cable TV Company	35	3
Joe's Car Palace	350	5
S.C. Student Loan	200	6
Florida Water Company	20	1
U-O-Us Insurance Company	125	5
Debtor's Credit Card	35	4

Table 10.2. Sample data for the BANK_ACCOUNTS table.

Account_ID	Type	Balance	Bank
1	Checking	500	FirstFederal
2	MoneyMarket	1200	FirstInvestor's
3	Checking	90	CreditUnion

Account_ID	Type	Balance	Bank
5	Checking	2500	SecondMutual
6	Business	4500	Fidelity

Table 10.3. Sample data for the COMPANY table.

Name	Address	City	State
Phone Company	111 1st Street	Atlanta	GA
Power Company	222 2nd Street	Jacksonville	FL
Record Club	333 3rd Avenue	Los Angeles	CA
Software Company	444 4th Drive	San Francisco	CA
Cable TV Company	555 5th Drive	Austin	TX
Joe's Car Palace	1000 Govt. Blvd	Miami	FL
S.C. Student Loan	25 College Blvd	Columbia	SC
Florida Water Company	1883 Hwy 87	Navarre	FL
U-O-Us Insurance Company	295 Beltline Hwy	Macon	GA
Debtor's Credit Card	115 2nd Avenue	Newark	NJ

现在你已经成功地使用 CREATE DATABASE、CREATE TABLE 和 INSERT 命令输入了所有的这些信息，现在我们开始对视图作进一步的讨论。

简单视图

让我们从最简单的视图开始，假设由于一些未知的原因，我们需要在 BILLS 中创建视图。它看上去与 BILLS 表相同但是名字（叫 DEBTS）不相同，语句如下：

INPUT:

```
SQL> CREATE VIEW DEBTS AS
```

```
SELECT * FROM BILLS;
```

可以用下边的语句来确认上边的结果。

INPUT/OUTPUT:

```
SQL> SELECT * FROM DEBTS;
```

NAME	AMOUNT	ACCOUNT_ID
Phone Company	125	1
Power Company	75	1
Record Club	25	2
Software Company	250	1
Cable TV Company	35	3
Joe's Car Palace	350	5

NAME	AMOUNT	ACCOUNT_ID
S.C. Student Loan	200	6
Florida Water Company	20	1
U-O-Us Insurance Company	125	5
Debtor's Credit Card	35	4

你甚至可以已经存在的视图来创建一个新的视图。当从视图中创建视图时要仔细，尽管操作是可以接受的，但是它使得维护工作变得复杂，假设你的视图有三级，如表的视图的视图的视图。那么当表中的第一级视图被删除时会有什么情况发生？另外两个视图会仍然存在，但是在第一个视图恢复之前它们是没有用处的，切记！当创建一个视图后,它实际上是一个虚表。

INPUT:

```
SQL> CREATE VIEW CREDITCARD_DEBTS AS
```

```
2  SELECT * FROM DEBTS
```

```
3  WHERE ACCOUNT_ID = 4;
```

```
SQL> SELECT * FROM CREDITCARD_DEBTS;
```

OUTPUT:

NAME	AMOUNT	ACCOUNT_ID
Debtor's Credit Card	35	4

CREATE VIEW 也允许你从表中选择特定的列到视图中，下例是从 COMPANY 表中选择了 NAME 和 STATE 列。

INPUT:

```
SQL> CREATE VIEW COMPANY_INFO (NAME, STATE) AS
```

```
2  SELECT * FROM COMPANY;
```

```
SQL> SELECT * FROM COMPANY_INFO;
```

OUTPUT:

NAME	STATE
Phone Company	GA
Power Company	FL
Record Club	CA
Software Company	CA
Cable TV Company	TX
Joe's Car Palace	FL

NAME	STATE
S.C. Student Loan	SC
Florida Water Company	FL
U-O-Us Insurance Company	GA
Debtor's Credit Card	NJ

注：用户可以通过创建视图来查询特定的数据，如果你的表有 50 列且有成千上万个记录但是你只需要其中两列的话。你可以创建视图来选择这两列，然后从视图中查询，你会发现查询在数据返回时间上与原来有相当大的不同。

列的重命名

视图继承了已有列的名字，此外视图还可以有自己的名字，SQL 的 CREATE VIEW 允许你对所选择的列进行重命名。它与前边的例子非常相似，如果你想把 COMPANY 中的 ADDRESS、CITY 和 STATE 字段组合起来并打印到信封上时该如何做呢？请看下边的例子，它使用了 SQL 的+操作符将地址字段与逗号和空格组合起来。

INPUT:

```
SQL> CREATE VIEW ENVELOPE (COMPANY, MAILING_ADDRESS) AS
```

```
2  SELECT NAME, ADDRESS + " " + CITY + ", " + STATE
```

```
3  FROM COMPANY;
```

```
SQL> SELECT * FROM ENVELOPE;
```

OUTPUT:

COMPANY	MAILING_ADDRESS
Phone Company	111 1st Street Atlanta, GA
Power Company	222 2nd Street Jacksonville, FL
Record Club	333 3rd Avenue Los Angeles, CA
Software Company	444 4th Drive San Francisco, CA
Cable TV Company	555 5th Drive Austin, TX
Joe's Car Palace	1000 Govt. Blvd Miami, FL
S.C. Student Loan	25 College Blvd. Columbia, SC
Florida Water Company	1883 Hwy. 87 Navarre, FL
U-O-Us Insurance Company	295 Beltline Hwy. Macon, GA
Debtor's Credit Card	115 2nd Avenue Newark, NJ

分析：

当在视图中使用 SQL 的计算功能时 SQL 会要求你给出一个虚字段的名字，这是可以理解的，因为像 COUNT(*)或 AVG(PAYMENT)是不能作为名字的。

注：检查你的解释器看它是否支持+操作符。

SQL 对视图的处理过程

视图可以以比数据在数据库表中的存储情况更为便捷的方式来返回数据。当需要连续进行几个复合的查询时（例如在存储过程和应用程序中时）视图也是非常方便的。为了进一步地说明视图和 SELECT 语句，下边的例子分别使用了 SQL 的查询方法和视图方法以作对比。假设你需要经常去运行一个查询，例如：你需要例行公事地将 BILLS 表与 BANK_ACCOUNT 表进行归并以得到支付信息。

INPUT:

```
SQL> SELECT BILLS.NAME, BILLS.AMOUNT, BANK_ACCOUNTS.BALANCE
      2  BLANCE, BANK_ACCOUNTS.BANK BANK FROM BILLS, BANK_ACCOUNTS
      3  WHERE BILLS.ACCOUNT_ID = BANK_ACCOUNTS.ACCOUNT_ID;
```

OUTPUT:

BILLS.NAME	BILLS.AMOUNT	BALANCE	BANK
Phone Company	125	500	First Federal
Power Company	75	500	First Federal
Record Club	25	1200	First Investor's
Software Company	250	500	First Federal
Cable TV Company	35	90	Credit Union
Joe's Car Palace	350	2500	Second Mutual
S.C. Student Loan	200	4500	Fidelity
Florida Water Company	20	500	First Federal
U-O-Us Insurance Company	125	2500	Second Mutual

这一过程用视图来表达则语句如下：

INPUT/OUTPUT:

```
SQL> CREATE VIEW BILLS_DUE (NAME, AMOUNT, ACCT_BALANCE, BANK) AS
      2  SELECT BILLS.NAME, BILLS.AMOUNT, BANK_ACCOUNTS.BALANCE,
      3  BANK_ACCOUNTS.BANK FROM BILLS, BANK_ACCOUNTS
```

```
4 WHERE BILLS.ACCOUNT_ID = BANK_ACCOUNTS.ACCOUNT_ID;
```

View created.

如果你对 BILLS_DUE 视图执行查询是使用了一些条件，如下语句所示：

INPUT/OUTPUT：

```
SQL> SELECT * FROM BILLS_DUE
```

```
2 WHERE ACCT_BALANCE > 500;
```

NAME	AMOUNT	ACCT_BALANCE	BANK
Record Club	25	1200	FirstInvestor's
Joe's Car Palace	350	2500	SecondMutual
S.C. Student Loan	200	4500	Fidelity
U-O-Us Insurance Company	125	2500	SecondMutual

分析：

在上述语句中 SQL 执行了好几步操作，因为 BILLS_DUE 是一个视图，不是一个真实的表。SQL 首先查找一个名字叫 BILLS_DUE 的表，但是没有找到，SQL 的过程可能会从系统表中发现 BILLS_DUE 原来是一个视图（这依据你所使用的数据库而定），于是它对视图进行了诠释并形成了如下的查询语句：

```
SQL> SELECT BILLS.NAME, BILLS.AMOUNT, BANK_ACCOUNTS.BALANCE,
```

```
2 BANK_ACCOUNTS.BANK FROM BILLS, BANK_ACCOUNTS
```

```
3 WHERE BILLS.ACCOUNT_ID = BANK_ACCOUNTS.ACCOUNT_ID
```

```
4 AND BANK_ACCOUNTS.BALANCE > 500;
```

例 10.1：

构造一个视图以显示所有需要发送账单的州，同时要求显示每个州的账单金额总数和账单的总数。

首先，你知道 CREATE VIEW 语句看起来应该是下边的样子：

```
CREATE VIEW EXAMPLE (STATE, TOTAL_BILLS, TOTAL_AMOUNT) AS...
```

现在你必须决定 SELECT 语句的内容，你要清楚根据需要你要选择 STATE 字段并应该使用 SELECT DISTINCT 语法以显示账单需要发送的州，例如：

INPUT：

```
SQL> SELECT DISTINCT STATE FROM COMPANY;
```

OUTPUT：

```
STATE GA FL CA TX SC NJ（为节约宽度这里用制表符取代了段落标记）
```


除了选择州字段以外，你还需要知道发往这个州的账单总数。因此，你需要归并 `BILLS` 表和 `COMPANY` 表。

INPUT/OUTPUT:

```
SQL> SELECT DISTINCT COMPANY.STATE, COUNT(BILLS.*) FROM BILLS, COMPANY
```

```
2 GROUP BY COMPANY.STATE
```

```
3 HAVING BILLS.NAME = COMPANY.NAME;
```

STATE	COUNT(BILLS.*)
GA	2
FL	3
CA	2
TX	1
SC	1
NJ	1

现在你已经成功地返回了需要的三分之二的结果，你可以用下边的语句来最终结束这条语句，加入 `SUM` 语句以返回每个州的金额总数：

INPUT/OUTPUT:

```
SQL> SELECT DISTINCT COMPANY.STATE, COUNT(BILLS.NAME), SUM(BILLS.AMOUNT)
```

```
2 FROM BILLS, COMPANY
```

```
3 GROUP BY COMPANY.STATE
```

```
4 HAVING BILLS.NAME = COMPANY.NAME;
```

STATE	COUNT(BILLS.*)	SUM(BILLS.AMOUNT)
GA	2	250
FL	3	445
CA	2	275
TX	1	35
SC	1	200
NJ	1	35

最后一步，你可以将它与最初的 `CREATE VIEW` 语句组合在一起。

INPUT/OUTPUT:

```
SQL> CREATE VIEW EXAMPLE (STATE, TOTAL_BILLS, TOTAL_AMOUNT) AS
```

```
2 SELECT DISTINCT COMPANY.STATE, COUNT(BILLS.NAME),SUM(BILLS.AMOUNT)
```

```
3 FROM BILLS, COMPANY
```

```

4 GROUP BY COMPANY.STATE
5 HAVING BILLS.NAME = COMPANY.NAME;

```

INPUT/OUTPUT:

```
SQL> SELECT * FROM EXAMPLE;
```

STATE	TOTAL_BILLS	TOTAL_AMOUNT
GA	2	250
FL	3	445
CA	2	275
TX	1	35
SC	1	200
NJ	1	35

上边的例子向你展示了如何设计 CREATE VIEW 语句和 SELECT 语句，SELECT 语句测试代码的作用在于检查数据的返回结果是不是符合你的要求，然后才将其最终组合来创建一个视图。

例 10.2:

假设你的债权人因为你推迟付款加收 10% 的服务费，而且不幸的是你在这个月的每件事都需要推迟，因此你想看一下需要推迟付款的债主的账号：

归并语句在这里是非常简单的（因为你不需要使用像 SUM 或 COUNT 之类的语句）。可是你会第一次发现使用视图的好处，在视图中你可以将增加的 10% 的服务费在视图中作为一个字段，由于这一点，你可以在视图中使用 SELECT 语句来为你计算总计的结果，语句如下：

INPUT:

```

SQL> CREATE VIEW LATE_PAYMENT (NAME, NEW_TOTAL, ACCOUNT_TYPE) AS
2 SELECT BILLS.NAME, BILLS.AMOUNT * 1.10, BANK_ACCOUNTS.TYPE
3 FROM BILLS, BANK_ACCOUNTS
4 WHERE BILLS.ACCOUNT_ID = BANK_ACCOUNTS.ACCOUNT_ID;

```

OUTPUT:

View created.

INPUT/OUTPUT:

```
SQL> SELECT * FROM LATE_PAYMENT;
```

NAME	NEW_TOTAL	ACCOUNT_TYPE
Phone Company	137.50	Checking
Power Company	82.50	Checking
Record Club	27.50	MoneyMarket
Software Company	275	Checking
Cable TV Company	38.50	Checking
Joe's Car Palace	385	Checking
S.C. Student Loan	220	Business
Florida Water Company	22	Checking
U-O-Us Insurance Company	137.50	Business
Debtor's Credit Card	38.50	Savings

在 SELECT 语句使用约束

在视图的 SELECT 语句中使用约束是必然的，在使用 SELECT 语句中可以应用下边这两个规则：

- 你不能使用 UNION 操作。
- 你不能使用 ORDER BY 子句，但是在视图中使用 GROUP BY 子句可以有 ORDER BY 子句相同的功能。

在视图中修改数据

正如同你所学习过的，你可以在数据库的一个或多个表中使用视图，你也可以在 SQL 和数据库应用程序中使用虚表。在使用 CREATE VIEW SELECT 创建视图以后，你可以用在第八天《操作数据》学习过的 INSERT，UPDATE 和 DELETE 语句来更新、插入、删除视图中的数据。

我们在稍后讨论在视图中操作数据的限制，下边的例子显示了如何在视图中操作数据。

将例 10.2 中的工作继续，更新 BILLS 表中的那不幸的 10% 的费用。

INPUT/OUTPUT:

```
SQL> CREATE VIEW LATE_PAYMENT AS
```

```
2 SELECT * FROM BILLS;
```

```
SQL> UPDATE LATE_PAYMENT
```

```
2 SET AMOUNT = AMOUNT * 1.10;
```

SQL> SELECT * FROM LATE_PAYMENT;

NAME	NEW_TOTAL	ACCOUNT_ID
Phone Company	137.50	1
Power Company	82.50	1
Record Club	27.50	2
Software Company	275	1
Cable TV Company	38.50	3
Joe's Car Palace	385	5
S.C. Student Loan	220	6
Florida Water Company	22	1
U-O-Us Insurance Company	137.50	5
Debtor's Credit Card	38.50	4

为了验证结果确实已经进行了更新，我们在一次在 BILLS 表中运行查询：

INPUT/OUTPUT:

SQL> SELECT * FROM BILLS;

NAME	NEW_TOTAL	ACCOUNT_ID
Phone Company	137.50	1
Power Company	82.50	1
Record Club	27.50	2
Software Company	275	1
Cable TV Company	38.50	3
Joe's Car Palace	385	5
S.C. Student Loan	220	6
Florida Water Company	22	1
U-O-Us Insurance Company	137.50	5
Debtor's Credit Card	38.50	4

现在我们从视图中删除一行：

INPUT/OUTPUT:

SQL> DELETE FROM LATE_PAYMENT

2 WHERE ACCOUNT_ID = 4;

SQL> SELECT * FROM LATE_PAYMENT;

NAME	NEW_TOTAL	ACCOUNT_ID
Phone Company	137.50	1
Power Company	82.50	1
Record Club	27.50	2

Software Company	275	1
Cable TV Company	38.50	3
Joe's Car Palace	385	5
S.C. Student Loan	220	6
Florida Water Company	22	1
U-O-Us Insurance Company	137.50	5

最后一步是测试 UPDATE 函数，为 BILLS 表中所有的 NEW_TOTAL 中大于 100 的增加 10。

INPUT/OUTPUT:

```
SQL> UPDATE LATE_PAYMENT
```

```
2 SET NEW_TOTAL = NEW_TOTAL + 10
```

```
3 WHERE NEW_TOTAL > 100;
```

```
SQL> SELECT * FROM LATE_PAYMENT;
```

NAME	NEW_TOTAL	ACCOUNT_ID
Phone Company	147.50	1
Power Company	82.50	1
Record Club	27.50	2
Software Company	285	1
Cable TV Company	38.50	3
Joe's Car Palace	395	5
S.C. Student Loan	230	6
Florida Water Company	22	1
U-O-Us Insurance Company	147.50	5

在视图中修改数据的几个问题

你大概已经看到了，视图其实就是一组表的映射，所以想要修改下层表的数据并不会总是像上例那样直接，下面给出了你在使用视图进行工作时常用会遇到的限制。

- 对于多表视图你不能使用 DELETE 语句。
- 除非底层表的所有非空列都已经在视图中出现，否则你不能使用 INSERT 语句，有这个限制的原因是 SQL 不知道应该将什么数据插入到 NOT COLUMNS 限制列中（没有在视图中出现的）。
- 如果对一个归并的表格插入或更新记录，那么所有被更新的记录必须属于同一个

物理表。

- 如果你在创建视图时使用了 `DISTINCT` 子句，那么你就不能插入或更新这个视图中的记录。
- 你不能更新视图中的虚拟列（它是用计算字段得到了）。

通用应用程序的视图

下边有几个视图需要完成的任务。

- 提供了用户安全功能。
- 可以进行单位换算。
- 创建一个新的虚拟表格式。
- 简单的结构化复合查询。

视图与安全性

尽管我们需要在第 12 天的《数据库安全》中才会完全地讨论数据库的安全性，但是在本题目下我们先来大致谈一谈如何使用视图的安全性功能。

所有的关系型数据库在今天都有着完善的内置的安全性特性。数据库系统的用户通常会根据他们所使用的数据库来分成不同的组，常用组的类型有数据库管理员（database administrators）、数据库开发员（database developers）、数据录入人员（data entry personnel）和大众用户，不同的组在使用数据库时有着不同的权限，数据库管理员具有系统的完全控制权限，包括更新（`UPDATE`）、插入（`INSERT`）、删除（`DELETE`）、修改（`ALTER`）数据库的特权。而大众用户则只有使用 `SELECT` 语句的权利——或许是只有对特定的数据库使用特定的 `SELECT` 语句的权利。

视图通常用在对用户访问数据进行控制的场所。例如：如果你只想让用户访问 `BILLS` 表中的 `NAME` 字段，你需要创建一个名字叫 `BILLS_NAME` 的视图。

INPUT/OUTPUT:

```
SQL> CREATE VIEW BILLS_NAME AS SELECT NAME FROM BILLS;
```

具有系统管理员权限的人也可以使用具有公共组 `SELECT` 权限的 `BILLS_NAME`，该组没有任何对下层 `BILLS` 表的权限，如你所料，SQL 也提供了可以使用的数据安全语句，现在你要知道的是视图对于实现数据库的安全有相当大的用处。

在单位换算中使用视图

视图在你提供给用户的数据与数据库中的真实数据不同时也相当有用。例如：如果 AMOUNT 字段实际上存储于美国，加拿大的用户不想频繁地进行美元与加拿大元之间的转换工作。那么你可以创建一个叫 CANADA_BILLS 的视图。

INPUT/OUTPUT:

```
SQL> CREATE VIEW CANADIAN_BILLS (NAME, CAN_AMOUNT) AS
```

```
2  SELECT NAME, AMOUNT / 1.10
```

```
3  FROM BILLS;
```

```
SQL> SELECT * FROM CANADIAN_BILLS;
```

NAME	CAN_AMOUNT
Phone Company	125
Power Company	75
Record Club	25
Software Company	250
Cable TV Company	35
Joe's Car Palace	350
S.C . Student Loan	200
Florida Water Company	20
U-O- Us Insurance Company	125

分析:

当进行类似这样的单位转换时，要注意当计算字段创建一个列时修改底层表的数据时可能带来的问题。与往常一样，你应该查看你的数据库系统的相关文档看一看你的系统上的 CREATE VIEW 命令是如何执行了。

在视图中使用简单的结构化复合查询

视图在你需要按次序运行一系列查询以后得到某个结果的情况下也很有用，下边的例子显示了如何在这种情况下使用视图。

如果想找出所有发给德克萨斯州的账单金额少于 50 美元的银行的名字，你可以把这个问题分解成如下的两个问题。

- 得到所有发给德克萨斯州的账单。

- 找出账单中金额小于 50 美元的记录。

让我们用两个分开的视图 BILLS_1 和 BILLS_2 来解决这两个问题：

INPUT/OUTPUT：

```
SQL> CREATE TABLE BILLS1 AS
```

```
2 SELECT * FROM BILLS
```

```
3 WHERE AMOUNT < 50;
```

```
SQL> CREATE TABLE BILLS2 (NAME, AMOUNT, ACCOUNT_ID) AS
```

```
2 SELECT BILLS.* FROM BILLS, COMPANY
```

```
3 WHERE BILLS.NAME = COMPANY.NAME AND COMPANY.STATE = "TX";
```

分析：

因为你想找的是所有发给德州的账单和所有账单中小于 50 美元的账单，你现在可以使用 SQL 中的 IN 子句来找出所有在 BILLS1 中发往德州的账单，这个信息来创建一个名字叫 BILLS3 的视图：

INPUT/OUTPUT：

```
SQL> CREATE VIEW BILLS3 AS
```

```
2 SELECT * FROM BILLS2 WHERE NAME IN
```

```
3 (SELECT * FROM BILLS1);
```

现在将上述查询与 BANK_ACCOUNT 表进行合并以得到最初想要的结果：

INPUT/OUTPUT：

```
SQL> CREATE VIEW BANKS_IN_TEXAS (BANK) AS
```

```
2 SELECT BANK_ACCOUNTS.BANK
```

```
3 FROM BANK_ACCOUNTS, BILLS3
```

```
4 WHERE BILLS3.ACCOUNT_ID = BANK_ACCOUNTS.ACCOUNT_ID;
```

```
SQL> SELECT * FROM BANK_IN_TEXAS;
```

```
BANK
```

```
Credit Union
```

分析：

如你所见，当把一个查询分解成几个视图以后，最后的查询就非常简单了，当然，使用一个视图也经常是必需的。

删除视图语句

就像每一个 CREATE 语句一样，CREATE VIEW 语句对应的也与 DROP VIEW 语句相对应，其语法形式如下：

```
SQL> DROP VIEW view_name;
```

在使用它的时候需要记住，DROP VIEW 命令会使所有与 DROP 视图相关联的视图不能正常运行，一些数据库系统甚至会将所有与要 DROP 的视图相关联的视图也删除掉。在 Personal Oracle7 中，如果你将 BILLS1 删除，那么最终的查询将会返回下边的错误：

INPUT/OUTPUT:

```
SQL> DROP VIEW BILLS1;
```

```
View dropped.
```

```
SQL> SELECT * FROM BANKS_IN_TEXAS;
```

```
ERROR at line 1:
```

```
ORA-04063: view "PERKINS.BANKS_IN_TEXAS" has errors
```

注：你可以删除一个视图而不影响任何一个真实的表，这也就是为什么我们将视图称为虚表的原因。（虚体也使用了相同的逻辑）

使用索引

使用索引是另外一种让数据提供给用户的形式与它在数据库中不同的方法，此外，索引可以让存储于磁盘上的数据进行重新排序（这是一些视图不具有的功能）。

在 SQL 中使用索引是其于以下几个原因：

- 在使用 UNIQUE 关键字时强制性地保证数据的完整性。
- 可以容易地用索引字段或其它字段进行排序。
- 提高查询的执行速度。

什么是索引？

可以用两种方法从数据库中获得数据，第一种方法常被称为顺序访问方式，它需要 SQL 检查每一个记录以找到与之相匹配的。这种查找的方法效率很低。但它是使记录准确定位的唯一方法。回想一下以前图书馆的卡片档案系统，假设卡片是按字母的顺序排列的，那

么在将它们抽出来后在放回卡片柜时。那么当你来到书柜的旁边以后，那么你能只能从头开始，然后一张卡片一张卡片地看，直到找到你所需要的（当然，也许你碰巧很快就找到了）。

现在假设图书管理员将书的标题按字母顺序排列，那么通过查看目录中的书的字母顺序你可以很快地找到你想要的书。

进一步设想如果管理员非常勤劳，他不但将书按标题进行了排序，而且还另外制作了不同的卡片柜，在那个卡片柜中他是按照作者的名字或其他的方式进行排序的。那么这对于你，一个图书馆的读者来说检索信息就有了相当大的灵活性，而且你只需要很短的时间就能找到你所需的内容。

在数据库中使用索引可以让 SQL 使用直接访问方式。SQL 采用树形结构来存储和返回索引数据，用以指示的数据存储在树的最末端（也就是叶子），它们被称为结点（也可以叫叶子）。每一个结点中有一个指向其它结点的指针，结点左边的值只是它的双亲结点，结点右边的值则是孩子结点或叶子。

SQL 将从根结点开始直到找到所需的数据。

注：当查询没有使用索引的表时查询通常是全表搜索后才会得到结果，全表搜索会让数据库服务程序遍历过表中的所有记录然后返回给定条件的记录，这种方法就好比从图书馆的第一号书架的第一本书找起，直到找到了你所需要的书一样。你或许会使用卡片柜以更快地找到所需的书，索引可以让数据库服务程序快速地定位到表中的确定行。

幸运的是这个树结构不需要由你来制作，你甚至不必去写从数据库的表中存储和读的过程，基本的 SQL 索引的语法形式如下：

INPUT/OUTPUT:

```
SQL> CREATE INDEX index_name  
2 ON table_name(column_name1, [column_name2], ...);
```

像你以前多次看到的那样，索引的语法对于不同的数据库系统差别很大，例如：CREATE INDEX 语句在 ORACLE7 中的形式如下：

SYNTAX:

```
CREATE INDEX [schema.]index  
ON { [schema.]table (column [!!under!!ASC|DESC]  
    [, column [!!under!!ASC|DESC]] ...)  
    | CLUSTER [schema.]cluster }
```

[INTRANS integer] [MAXTRANS integer]

[TABLESPACE tablespace]

[STORAGE storage_clause]

[PCTFREE integer]

[NOSORT]

而它在 Sybase SQL Server 中的语法形式则如下：

SYNTAX:

```
create [unique] [clustered | nonclustered]
      index index_name
on [[database.]owner.]table_name (column_name
      [, column_name]...)
[with {fillfactor = x, ignore_dup_key, sorted_data,
      [ignore_dup_row | allow_dup_row]}]
[on segment_name]
```

Informix SQL 解释器的命令形式则如下：

SYNTAX:

```
CREATE [UNIQUE | DISTINCT] [CLUSTER] INDEX index_name
ON table_name (column_name [ASC | DESC],
      column_name [ASC | DESC]...)
```

注意到所有这些解释器有几点是相同的，它们的基本开始语句都是：

```
CREATE INDEX index_name ON table_name (column_name, ...)
```

SQL Server 和 ORACLE 允许你创建成簇的索引，这将在稍后讨论。ORACLE 和 Informix 允许你指明列名是按升序排列还是按降序排列。我们不喜欢听到被打断的声音，但是请再一次，参考你的数据库管理系统以得到明确的关于 CREATE INDEX 的指示。

例如：要对 BILLS 表中的 ACCOUNTID 字段创建索引，其 CREATE INDEX 语句如下：

INPUT:

```
SQL> SELECT * FROM BILLS;
```

OUTPUT:

NAME	AMOUNT	ACCOUNT_ID
Phone Company	125	1
Power Company	75	1
Record Club	25	2
Software Company	250	1
Cable TV Company	35	3
Joe's Car Palace	350	5
S.C. Student Loan	200	6
Florida Water Company	20	1
U-O-Us Insurance Company	125	5
Debtor's Credit Card	35	4

INPUT/OUTPUT:

```
SQL> CREATE INDEX ID_INDEX ON BILLS( ACCOUNT_ID );
```

```
SQL> SELECT * FROM BILLS;
```

NAME	AMOUNT	ACCOUNT_ID
Phone Company	125	1
Power Company	75	1
Software Company	250	1
Florida Water Company	20	1
Record Club	25	2
Cable TV Company	35	3
Debtor's Credit Card	35	4
Joe's Car Palace	350	5
U-O-Us Insurance Company	125	5
S.C. Student Loan	200	6

直至索引被 DROP INDEX 语句删除之前，BILLS 表是按照 ACCOUNT_ID 的顺序进行排序的，DROP INDEX 语句是非常清楚的：

SYNTAX:

```
SQL> DROP INDEX index_name;
```

当索引被删除以后的结果是什么样呢？

INPUT/OUTPUT:

```
SQL> DROP INDEX ID_INDEX;
```

Index dropped.

```
SQL> SELECT * FROM BILLS;
```

NAME	AMOUNT	ACCOUNT_ID
Phone Company	125	1
Power Company	75	1
Record Club	25	2
Software Company	250	1
Cable TV Company	35	3
Joe's Car Palace	350	5
S.C. Student Loan	200	6
Florida Water Company	20	1
U-O-Us Insurance Company	125	5
Debtor's Credit Card	35	4

分析：

现在的 BILLS 表是它原本的形态，使用索引不会对表中的物理存储造成影响。

你也许想知道为什么数据库提供了索引而又允许你使用 ORDER BY 子句吧？

INPUT/OUTPUT：

SQL> SELECT * FROM BILLS ORDER BY ACCOUNT_ID;

NAME	AMOUNT	ACCOUNT_ID
Phone Company	125	1
Power Company	75	1
Software Company	250	1
Florida Water Company	20	1
Record Club	25	2
Cable TV Company	35	3
Debtor's Credit Card	35	4
Joe's Car Palace	350	5
U-O-Us Insurance Company	125	5
S.C. Student Loan	200	6

分析：

它与使用 ID_INDEX 语句的结果是一样的，不同之处在于当你使用 ORDER BY 子句时每次运行它都需要重新进行排序，而当你使用索引的时候，数据库会建立一个物理索引对象（就是前边提到的树结构）而在你每次运行查询时都访问同一个索引。

警告：当表被删除时，所有与表相关的索引也将被删除。

使用索引的技巧

这里给出了几个在使用索引时需要记住的技巧：

- 对于小表来说，使用索引对于性能不会有任何提高。
- 当你的索引列中有极多的不同的数据和空值时索引会使性能有极大的提高。
- 当查询要返回的数据很少时索引可以优化你的查询（比较好的情况是少于全部数据的 25%）。如果你要返回的数据很多时索引会加大系统开销。
- 索引可以提高数据的返回速度，但是它使得数据的更新操作变慢，在对记录 and 索引进行更新时请不要忘记这一点。如果要进行大量的更新操作，在你执行更新操作时请不要忘记先删除索引，当执行完更新操作后，只需要简单的恢复索引即可，对于一次特定的操作，系统可以保存删除的索引 18 个小时，在这个时间内数据更新完后你可以恢复它。
- 索引会占用你的数据库的空间，如果你的数据库管理系统允许你管理数据库的磁盘空间，那么在设计数据库的可用空间时要考虑索引所占用的空间。
- 对字段的索引已经对两个表进行了归并操作，这一技术可以极大地提高归并的速度。
- 大多数数据库系统不允许你对视图创建索引，如果你的数据库系统允许这样做，那么可以使用这种方法来在 SELECT 语句中对视图的数据进行排序（很不巧，一些数据库系统中也不允许在视图中使用 ORDER BY 子句）。
- 不要创建对经常需要更新或修改的字段创建索引，更新索引的开销会降低你所期望获得的性能。
- 不要将索引与表存储在同一个驱动器上，分开存储会去掉访问的冲突从而使结果返回得更快。

对更多的字段进行索引

SQL 也允许你对多个字段进行索引，这种索引被称为复合索引，下边的代码是一个简单的复合索引的例子，注意虽然是对两个字段进行索引，但索引在物理结构上只有一个。

INPUT/OUTPUT:

```
SQL> CREATE INDEX ID_CMPD_INDEX ON BILLS( ACCOUNT_ID, AMOUNT );
```

```
Index created.
```

SQL> SELECT * FROM BILLS;

NAME	AMOUNT	ACCOUNT_ID
Florida Water Company	20	1
Power Company	75	1
Phone Company	125	1
Software Company	250	1
Record Club	25	2
Cable TV Company	35	3
Debtor's Credit Card	35	4
U-O-U's Insurance Company	125	5
Joe's Car Palace	350	5
S.C. Student Loan	200	6

SQL> DROP INDEX ID_CMPD_INDEX;

Index dropped.

分析:

选择唯一值最多的列建立索引可以达到你所希望的性能。例如，在 BILLS 表中 NAME 字段中的每一个值都是唯一的。当使用复合索引时，要把最可能选择的字段放在前边，也就是说，把你最经常在查询中使用是字段放在最前边（在 CREATE INDEX 中列的出现次序不必与表中的次序一致）如果你经常使用下边的语句：

SQL> SELECT * FROM BILLS WHERE NAME = "Cable TV Company";

为了想达到所期望的性能，你必须在索引中将 NAME 字段放在第一位，这里有两个例子：

SQL> CREATE INDEX NAME_INDEX ON BILLS (NAME, AMOUNT);

或

SQL> CREATE INDEX NAME_INDEX ON BILLS (NAME);

在这两个例子中 NAME 都在索引字段的最左边，所以这两个索引可以提高对 NAME 的查询的性能。

复合索引也可以根据他们自己的选择性来对两个以上的字段进行索引。作为一个选择性的例子，请看一下下边的这个表。

ACCOUNT_ID	TYPE	BALANCE	BANK
1	Checking	500	First Federal
2	Money Market	1200	First Investor's

ACCOUNT_ID	TYPE	BALANCE	BANK
3	Checking	90	Credit Union
4	Savings	400	First Federal
5	Checking	2500	Second Mutual
6	Business	4500	Fidelity

请注意输出的六个记录，checking 值在这里出现了三次，所以它的选择性要低于 ACCOUNT_ID，请注意，每一个 ACCOUNT_ID 的值都是唯一的，要想提高你的索引的选择性，你可以将 TYPE 字段与 ACCOUNT_ID 字段组合在一起建立一个索引。这将创建一个唯一的索引值（当然，这也是你所能得到的最高的选择性）。

注：一个索引可以包含多个列通常是指复合索引，复合索引的性能与单个字段的索引相比是无法断定的。以 ORACLE 为例，如果你在查询条件中经常指定某一特定的列那个你可以创建这个列的索引。而当你的查询需要复合条件时你可以创建复合索引，当创建多个索引的时候你需要参考你所选定的解释器的帮助信息以从中得到确定的复合索引的用法。

在创建索引时使用 UNIQUE 关键字

复合索引通常使用 UNIQUE 关键字来防止有相同数据的多个记录多次出现。例如，如果你想要 BILLS 表具有下边的规则：每一个账单的交付公司都必须有不同的银行账号。你需要创建一个包括 NAME 和 ACCOUNT_ID 的唯一索引，不幸的是 ORACLE7 不支持 UNIQUE 语法，它是用 UNIQUE 完整性约束来达到内容唯一这一特性的。下边的例子中给出了在 Sybase 的 Transact-SQL 语言中 UNIQUE 关键字的用法：

INPUT:

```
1> create unique index unique_id_name
2> on BILLS(ACCOUNT_ID, NAME)
3> go

1> select * from BILLS
2> go
```

OUTPUT:

NAME	AMOUNT	ACCOUNT_ID
Florida Water Company	20	1

NAME	AMOUNT	ACCOUNT_ID
Power Company	75	1
Phone Company	125	1
Software Company	250	1
Record Club	25	2
Cable TV Company	35	3
Debtor's Credit Card	35	4
U-O-Us Insurance Company	125	5
Joe's Car Palace	350	5
S.C. Student Loan	200	6

现在，我们试着向表中插入一个已经存在的记录：

INPUT：

```
1> insert BILLS (NAME, AMOUNT, ACCOUNT_ID)
```

```
2> values("Power Company", 125, 1)
```

```
3> go
```

分析：

你会收到了个错误信息告诉你插入操作是不允许的，这个错误可以为应用程序所捕获，从而告知用户他插入了一个不合法的数据。

例 10.3：

在 BILLS 表中创建一个索引以对 AMOUNT 字段进行降序排列：

INPUT/OUTPUT：

```
SQL> CREATE INDEX DESC_AMOUNT
      ON  BILLS(AMOUNT DESC);
```

Index created.

分析：

这是我们第一次使用 DESC 操作，它将告诉 SQL 将索引降序排列（通常情况下是升序排列）。现在来看一下结果：

INPUT/OUTPUT：

```
SQL> SELECT * FROM BILLS;
```

NAME	AMOUNT	ACCOUNT_ID
Joe's Car Palace	350	5
Software Company	250	1

NAME	AMOUNT	ACCOUNT_ID
S.C. Student Loan	200	6
Phone Company	125	1
U-O-Us Insurance Company	125	5
Power Company	75	1
Cable TV Company	35	3
Debtor's Credit Card	35	4
Record Club	25	2
Florida Water Company	20	1

分析：

这个例子对 AMOUNT 列使用 DESO 操作创建了一个索引，注意输出的顺序是从大到小。

索引与归并

当在查询中使用了复杂的归并时，你的 SELECT 语句会耗用很长的时间。对于大表来说，所用的时间可能会达到好几秒钟（与你通常需要等待几毫秒相对比）。这样的性能在客户机/服务器环境中常会令你的用户对使用你的应用程序感到不耐烦。在归并时对字段创建索引可以显著地提高你的查询反映速度。但是，如果你创建太多的索引，就会使你的系统的性能下降而不是提高，我们推荐你在几个大表中进行索引试验（对数以千计的数据排序）。这样的试验可以让你更深入地理解 SQL 查询的优化。

注：大多数的解释器有捕获查询耗用时间的机制，ORACLE 将这种特性称为 timing，请察看你所使用的解释器的相关信息。

下边的例子对 BILLS 表与 BANK_ACCOUNT 表根据 ACCOUNT_ID 字段创建了索引：

INPUT/OUTPUT：

```
SQL> CREATE INDEX BILLS_INDEX ON BILLS(ACCOUNT_ID);
```

```
Index created.
```

```
SQL> CREATE INDEX BILLS_INDEX2 ON BANK_ACCOUNTS(ACCOUNT_ID);
```

```
Index created.
```

```
SQL> SELECT BILLS.NAME NAME, BILLS.AMOUNT AMOUNT,
        BANK_ACCOUNTS.BALANCE ACCOUNT_BALANCE
        FROM BILLS, BANK_ACCOUNTS
```

WHERE BILLS.ACCOUNT_ID = BANK_ACCOUNTS.ACCOUNT_ID;

NAME	AMOUNT	ACCOUNT_BALANCE
Phone Company	125	500
Power Company	75	500
Software Company	250	500
Florida Water Company	20	500
Record Club	25	1200
Cable TV Company	35	90
Debtor's Credit Card	35	400
Joe's Car Palace	350	2500
U-O-Us Insurance Company	125	2500
S.C. Student Loan	200	4500

分析：

这个例子中首次在相关的表中为 ACCOUNT_ID 字段创建了索引，在每一个表中均对 ACCOUNT_ID 字段创建了索引以后，归并就可以更快地访问特定行的数据。作为一个规则，你应该对表中的唯一属性的字段或你用以归并操作的字段来创建索引。

群集（簇）的使用

尽管在开始的时候我们曾经说过索引只是提供给用户的一种与数据的物理存在不同的查看方式。但是这话并不是绝对的，在许多数据管理系统中都支持一种特殊的、可以允许数据库管理员或开发人员对数据进行群集的索引。当使用群集索引时，数据在表中的物理排列方式将会被修改，使用群集索引通常比传统的不使用群集的索引速度要快。但是，许多数据库管理系统（如 Sybase 的 SQL Server）只允许一个表有一个群集索引，用于创建群集索引的字段常常是主关键字。用 Sybase 的 Transact-SQL 你可以对 BANK_ACCOUNT 的 ACCOUNT_ID 字段创建一个群集的，不重复的索引，语法如下：

SYNTAX:

```
create unique clustered index id_index on BANK_ACCOUNTS(ACCOUNT_ID)
```

```
go
```

ORACLE 中群集的概念与此不同，当使用 ORACLE 关系数据库系统时，群集就是一个像数据或表一样的对象，群集一般是存储了表的共有字段以提高对表的访问速度。

这是一个 ORACLE7 中创建群集的例子：

SYNTAX:

```
CREATE CLUSTER [schema.]cluster (column datatype [,column datatype] ... )  
[PCTUSED integer] [PCTFREE integer] [SIZE integer [K|M] ]  
[INITRANS integer] [MAXTRANS integer] [TABLESPACE tablespace]  
[STORAGE storage_clause] [!!under!!INDEX | [HASH IS column] HASHKEYS integer]
```

你随后创建的其于该表的群集的索引会被加入到群集中，然后把表也加入群集中。你应该只将经常需要归并的表加入到群集，不要向群集中加入只需要用简单的 SELECT 语句进行个别访问的表。

很明显，群集是 SQL 的第三方特性，所以我们不准备详细地讨论创建和使用它的语法的细节问题。但是，你要查看你的数据库系统的文档看它是否支持这一有用的特性。

总结

视图是一种虚表，视图是提供给用户的数据与其在数据库的真实面貌不相同的一种方法，CREATE VIEW 语法的语法使用了标准的 SELECT 语法来创建了一个视图（除了一些小差别），你可以将视图视为一个常规的表来执行插入、删除、更新和选择操作，我们也简要地提到了视图是实现数据库安全的一种重要方法，有关数据的安全性将在第 12 天作更详细的讨论。

基本的创建视图的语法如下：

```
CREATE VIEW view_name AS SELECT field_name(s) FROM table_name(s);
```

视图主要用于以下方面：

- 提高用户数据的安全性
- 进行单位换算
- 创建一个新格式的虚表
- 使复杂查询的构筑简单化

索引也是一种数据库设计和 SQL 编程的工具，索引是一种存储在你的数据库管理系统中的物理对象，它可以让你的查询更快地从数据库中返回数据。此外，索引是可以定制的，正确地在查询中使用索引可以使性能显著地提高。

创建索引的基本语法如下：

```
CREATE INDEX index_name ON table_name ( field_name (s));
```

在一些数据库系统中提供了一些非常有用的如 UNIQUE 和 CLUSTER 关键字附加选项。

问与答：

问：如果数据已经在我的数据库中进行了排序，我是否还有必须在表中使用索引？

答：索引通过在树结构中查找关键值而提高你的数据库查询性能，它比对数据库的顺序访问方式快得多。记住：SQL 不需要知道你的数据库是否已经进行了排序。

问：我可以创建一个包括多个表中字段的索引吗？

答：你不能！但是，以 ORACLE 为例，它允许你创建一个群集，你可以将表放入群集中以根据表的共有字段创建一个群集索引，这是它的一个例外，所有你应该查看你所使用的解释器的文档以找到这方法的详细解答。

校练场

1、当在一个不唯一的字段中创建一个唯一值索引会有什么结果？

2、下边的话是对是错

视图和索引都会占用数据库的空间，所以在设计数据库空间时要考虑到这一点。

如果一个从更新了一个已经创建视图的表，那么视图必须进行同样的更新才会看到相同的数据。

如果你的磁盘空间够而你想加快你的查询的速度，那么索引越多越好。

3、下边的 CREATE 语句是否正确？

```
SQL> create view credit_debts as (select all from debts where account_id = 4);
```

4、下边的 CREATE 语句是否正确？

```
SQL> create unique view debts as select * from debts_tbl;
```

5、下边的 CREATE 语句是否正确？

```
SQL> drop * from view debts;
```

6、下边的 CREATE 语句是否正确？

```
SQL> create index id_index on bills (account_id);
```

练习：

- 1、检查你所使用的数据库系统，它是否支持视图？允许你在创建视图时使用哪些选项？用它的语法来写一个简单的创建视图语句，并对其进行如 SELECT 和 DELETE 等常规操作后再删除视图。
- 2、检查你所使用的数据库系统看它是否支持索引？它有哪些选项？在你的数据库系统中的一些已经存在的表中试一下这些选项。进一步，确认在你的数据库系统中是否支持 UNIQUE 和 CLUSTER 索引。
- 3、如果可能的话，在一个表中输入几千条记录，用秒表或钟来测定一下你的数据库系统对特定操作的反映时间，加入索引是否使性能提升了？试一下今天提到的技巧。

第 11 天：事务处理控制

前十天中我们学习了实际上我们可以对关系数据库系统中的数据所做的每一件事。例如：我们已经知道了如何使用 SQL 的 SELECT 语句根据用户给定的条件从一个或多个表中获得数据，我们也有机会体验了数据修改语句如 INSERT、DELETE、UPDATE。在今天，我们将成为中级 SQL 用户，如果有必要，我们将建立一个数据库及其相关的表，每一个表中都包括几个不同类型的字段，通过合适的设计方法，我们会成为从数据库到应用程序的桥梁。

目标：

如果你是一个临时用户，只需要偶而使用 SQL 从数据库获得数据的话，那么前十天的主题已经为你提供了足够的内容。但是，如果你想开发可以在使用数据库系统下运行的专业应用程序（这在当前是很普遍的），那么你在今后四天中讲到的内容（事务控制、安全、内嵌 SQL 语句、数据库过程）将会对你有很大的帮助，我们先从事务控制开始，到今天的结束，我们将学会以下内容：

- 基本的事务控制
- 如何确认或终止某一项事务
- Sybase 与 Oracle 在事务处理上的不同之处

注：在今天的例子中我们使用 PERSONAL ORACLE 7 和 SYBASE SQL SERVER，对于你所使用的解释器请查看相应的帮助文档以找出它们的不同之处。

事务控制

事务控制或者说事务处理是指关系数据库系统执行数据库事务的能力，事务是指在逻辑上必须完成的一命令序列的单位，单元工作期是指事务的开始和结束时期。如果在事务中产生的错误，那么整个过程可以根据需要被终止，如果每一件事都是正确的，那么结果将会被保存到数据库中。

日后你也许会运行其于网络的多用户应用程序，客户/服务环境就是为它而设计的，传

统上的服务器（例如数据库服务器）支持多个与它连接的工作站，与其它技术一样，新特性提高了数据库的复杂程度，下边的几段描述了一个银行所使用的应用程序。

银行应用程序

假定你受雇于联邦银行并负责为他们设计一个支票管理系统，你已经设计了一个非常完美的数据库，并且经常测试检验证明是正确无误的。你在应用程序中调用它以后，你从账号中支取了 20 元并进行验证，数据库中确实已经少了 20 元，你又从帐号中存入了 50.25 元并进行验证，结果也与所期望的相同。于是你骄傲在告诉你的老板系统可以运行了，几台计算机接入了程序并开始工作。

几分钟以后，你注意到了一个你没有预见的问题，一个出纳员向帐号中存入了一张支票而另一个出纳则从相同的帐号中提出了一部分钱。在分钟之内，由于多用户的同时操作就导致的帐目无法平衡。很不幸，由于他们之间互相进行更新和写入操作，你的应用程序很快就因为过负荷而断线，我们假定出现这个问题的数据库名字叫 CHECKING，它有两个表，其内容如下所示：

表 11.1

Name	Address	City	State Zip	Customer_ID
Bill Turner	725 N. Deal Parkway	Washington	DC	20085 1
John Keith	1220 Via De Luna Dr.	Jacksonville	FL	33581 2
Mary Rosenberg	482 Wannamaker Avenue	Williamsburg	VA	23478 3
David Blanken	405 N. Davis Highway	Greenville	SC	29652 4
Rebecca Little	7753 Woods Lane	Houston	TX	38764 5

表 11.2

Average_Bal	Curr_Bal	Account_ID
1298.53	854.22	1
5427.22	6015.96	2
211.25	190.01	3
73.79	25.87	4
1285.90	1473.75	5
1234.56	1543.67	6
345.25	348.03	7

假定你的应用程序为 BILL Turner 运行了 SELECT 查询并得到如下结果：

OUTPUT:

NAME: Bill Turner

ADDRESS: 725 N. Deal Parkway

CITY: Washington

STATE: DC

ZIP: 20085

CUSTOMER_ID: 1

当返回数据的时候，另外一个用户连接到了数据库并更新了 BILL Turner 的住址信息：

INPUT:

```
SQL> UPDATE CUSTOMERS SET Address ="11741 Kingstowne Road"
      WHERE Name = "Bill Turner";
```

你现在看到了，如果你执行 SELECT 语句当中出现的更新操作的话那么你所得到的结果将是不正确的。如果你的应用程序可以生成一个信件给 Bill Turner，那么由于地址是错误的，如果信已经发送了，那你是不能对地址进行修改的。但是如果你使用了事务处理机制，那么你就可以对检测到错误的数据进行修改，你所进行的所有操作也都可以撤消。

开始事务处理

事务处理在执行上是非常简单的，你需要检查你所执行的语法是 Oracle RDBMS SQL 语法还是 Sybase SQL Server SQL 语法。

所有支持事务处理的系统都必须以一种准确的语法来告诉系统一项事务是如何开始的（不要忘记事务处理只是工作的逻辑分组，它有自己的开始和结束），在使用 PERSONAL ORACLE7 时，它的语法形式如下：

SYNTAX:

```
SET TRANSACTION {READ ONLY | USE ROLLBACK SEGMENT segment}
```

SQL 标准要求每一种数据库的 SQL 解释器都必须支持语句级的读一致，这也就是说，当某一条语句运行的时候数据必需保持不变。但是，在许多情况下在一个工作过程中必须要求数据保持有效，而不仅仅是对单个语句。ORACLE 允许用户用 SET TRANSACTION 来指定事务的开始，如果你想检查 BILL TUNER 的信息并且要保证数据在这之中是不能改变的，那么你可以使用如下语句：

INPUT:

```
SQL> SET TRANSACTION READ ONLY;
```

```
SQL> SELECT * FROM CUSTOMERS
```

```
WHERE NAME = 'Bill Turner';
```

```
SQL> COMMIT;
```

我们将在今天的早些时候来讨论 COMMIT 语句，这里的 SET TRANSACTION READ ONLY 允许你锁定一个记录集直到事务结束，你可以在下列语句中使用 READ ONLY 选项。

```
SELECT
```

```
LOCK TABLE
```

```
SET ROLE
```

```
ALTER SESSION
```

```
ALTER SYSTEM
```

选项 USE ROLLBACK SEGMENT 告诉 ORACLE 数据库提供数据回溯的存储空间段，这一选项是 ORACLE 对标准的 SQL 的扩展，如果需要维护你的数据库请参见 ORACLE 的帮助文档以获得更多的帮助信息。

SQL Server's Transact-SQL 语言用下边的方法实现了开始事务处理的命令。

SYNTAX:

```
begin {transaction | tran} [transaction_name]
```

它的实现方法与 ORACLE 的有一些不同，（SYBASE 不允许你指定 READ ONLY 选项）但是，SYBASE 允许你给出事务处理的名字，从最早的事务到最近发生的事务处理都可以一次退回。

INPUT:

```
1> begin transaction new_account
```

```
2> insert CUSTOMERS values ("Izetta Parsons", "1285 Pineapple Highway", "Greenville", "AL"  
32854, 6)
```

```
3> if exists(select * from CUSTOMERS where Name = "Izetta Parsons")
```

```
4> begin
```

```
5> begin transaction
```

```
6> insert BALANCES values(1250.76, 1431.26, 8)
```

```
7> end
```

```
8> else
```

```
9> rollback transaction

10> if exists(select * from BALANCES where Account_ID = 8)

11> begin

12> begin transaction

13> insert ACCOUNTS values(8, 6)

14> end

15> else

16> rollback transaction

17> if exists (select * from ACCOUNTS where Account_ID = 8 and Customer_ID = 6)

18> commit transaction

19> else

20> rollback transaction

21> go
```

现在，请不要担心 ROLLBACK TRANSACTION 和 COMMIT TRANSACTION 语句，重要的问题是这是一个内嵌的事务处理，或者说是事务处理之中还有事务处理。

注意，最开始的事务处理在第 1 行，之后是插入语句，你检查了插入确实已经执行了以后，第二个事务处理在第 5 行开始，这种在事务之中的事务在术语上称为内嵌事务。

有一些数据库支持 AUTOCOMMIT 选项，它可以在 SET 命令中使用，如下例：

```
SET AUTOCOMMIT [ON | OFF]
```

默认情况上 SET AUTOCOMMIT ON 命令在启动时是自动运行的，它告诉 SQL 自动确认你所运行的所有的语句，如果你不想让这个命令自动运行，那么请将它的参数设为 NO。

```
SET AUTOCOMMIT OFF
```

注：请检查你的数据库文档确认在你的数据库系统中一项事务处理是如何开始的。

结束事务处理

在 ORACLE 语法中结束事务处理语句的语法如下：

SYNTAX:

```
COMMIT [WORK]
```

```
[ COMMENT 'text']
```

| FORCE 'text' [, integer]] ;

它的命令语法与 Sybase 的语法是相同的。

语法：

COMMIT (TRANSACTION | TRAN | WORK) (TRANSACTION_NAME)

COMMIT 命令将保存在一项事务中所进行的所有的改变，在开始一项事务处理之前要先运行 COMMIT 命令以确保在之前没有事务未被确认。

在下边的例子中，如果 COMMIT 没有收到任何系统错误的情况下它将会执行确认。

INPUT：

SQL> COMMIT;

SQL> SET TRANSACTION READ ONLY;

SQL> SELECT * FROM CUSTOMERS

WHERE NAME = 'Bill Turner';

---Do Other Operations---

SQL> COMMIT;

在 ORACLE 中 COMMIT 语句的使用方法如下：

INPUT：

SQL> SET TRANSACTION;

SQL> INSERT INTO CUSTOMERS VALUES

("John MacDowell", "2000 Lake Lunge Road", "Chicago", "IL", 42854, 7);

SQL> COMMIT;

SQL> SELECT * FROM CUSTOMERS;

CUSTOMER 表的内容如下：

Name	Address	City	State	Zip	Customer_ID
Bill Turner	725 N. Deal Parkway	Washington	DC	20085	1
John Keith	1220 Via De Luna Dr.	Jacksonville	FL	33581	2
Mary Rosenberg	482 Wannamaker Avenue	Williamsburg	VA	23478	3
David Blanken	405 N. Davis Highway	Greenville	SC	29652	4
Rebecca Little	7753 Woods Lane	Houston	TX	38764	5
Izetta Parsons	1285 Pineapple Highway	Greenville	AL	32854	6
John MacDowell	2000 Lake Lunge Road	Chicago	IL	42854	7

而 Sybase SQL 使用 COMMIT 的语法方式如下：

INPUT:

1>begin transaction

2>insert into CUSTOMERS values

("John MacDowell", "2000 Lake Lunge Road", "Chicago", "IL", 42854, 7)

3>commit transaction

4>go

1>select * from CUSTOMERS

2>go

Name	Address	City	State	Zip	Customer_ID
Bill Turner	725 N. Deal Parkway	Washington	DC	20085	1
John Keith	1220 Via De Luna Dr.	Jacksonville	FL	33581	2
Mary Rosenberg	482 Wannamaker Avenue	Williamsburg	VA	23478	3
David Blanken	405 N. Davis Highway	Greenville	SC	29652	4
Rebecca Little	7753 Woods Lane	Houston	TX	38764	5
Izetta Parsons	1285 Pineapple Highway	Greenville	AL	32854	6
John MacDowell	2000 Lake Lunge Road	Chicago	IL	42854	7

上边的语句完成了与 ORACLE7 相同的功能，但是，在使用 COMMIT 确认事务处理之前，你应该确保在该事务中的工作是正确无误的。

注：COMMIT WORD 命令与 COMMIT 命令的作用是相同的（或 Sybase 中的 COMMIT TRANSACTION）它与 ANSI SQL 的语法一样的简单。

切记 COMMIT 语句一定要与之前的 SET TRANSCATION 或 BEGIN TRANSCATION 语句一致。注意，在下边的语句中你将会收到错误信息。

Oracle SQL:

INPUT:

SQL> INSERT INTO BALANCES values (18765.42, 19073.06, 8);

SQL> COMMIT WORK;

Sybase SQL:

INPUT:

1> insert into BALANCES values (18765.42, 19073.06, 8)

2> commit work

取消事务处理

在一个事务处理的过程中，常常会运行一些错误检查以确认在过程中是否语句是运行成功。你可以使用 ROLLBACK 语句来撤消事务中所做的每一项工作，即便工作是成功的你也可以撤消。但是，这必须是在 COMMIT 之前，ROLLBACK 语句必须在一个事务之中运行，它可以一直撤消到事务的开始，也就是说：数据库会一直返回到事务处理刚开始的状态，在 ORACLE 7 中它的语法形式如下：

SYNTAX:

```
ROLLBACK [WORK]
[ TO [SAVEPOINT] savepoint
| FORCE 'text' ]
```

如你所见，该命令可以设置事务的 SAVEPOINT，我们将在今天的晚些时候来讨论这项技术。

Sybase Transact-SQL's 的 ROLLBACK 语句与 COMMIT 语句非常相似。

SYNTAX:

```
rollback {transaction | tran | work} [transaction_name | savepoint_name]
```

一个 ORACLE 的命令序列如下：

INPUT:

```
SQL> SET TRANSACTION;
```

```
SQL> INSERT INTO CUSTOMERS VALUES
```

```
("Bubba MacDowell", "2222 Blue Lake Way", "Austin", "TX", 39874, 8);
```

```
SQL> ROLLBACK;
```

```
SQL> SELECT * FROM CUSTOMERS;
```

Name	Address	City	State	Zip	Customer_ID
Bill Turner	725 N. Deal Parkway	Washington	DC	20085	1
John Keith	1220 Via De Luna Dr.	Jacksonville	FL	33581	2
Mary Rosenberg	482 Wannamaker Avenue	Williamsburg	VA	23478	3
David Blanken	405 N. Davis Highway	Greenville	SC	29652	4
Rebecca Little	7753 Woods Lane	Houston	TX	38764	5
Izetta Parsons	1285 Pineapple Highway	Greenville	AL	32854	6
John MacDowell	2000 Lake Lunge Road	Chicago	IL	42854	7

而 A Sybase SQL 的命令序列则如下：

INPUT：

1> begin transaction

2> insert into CUSTOMERS values

("Bubba MacDowell", "2222 Blue Lake Way", "Austin", "TX", 39874, 8)

3> rollback transaction

4> go

1> SELECT * FROM CUSTOMERS

2> go

Name	Address	City	State	Zip	Customer_ID
Bill Turner	725 N. Deal Parkway	Washington	DC	20085	1
John Keith	1220 Via De Luna Dr.	Jacksonville	FL	33581	2
Mary Rosenberg	482 Wannamaker Avenue	Williamsburg	VA	23478	3
David Blanken	405 N. Davis Highway	Greenville	SC	29652	4
Rebecca Little	7753 Woods Lane	Houston	TX	38764	5
Izetta Parsons	1285 Pineapple Highway	Greenville	AL	32854	6
John MacDowell	2000 Lake Lunge Road	Chicago	IL	42854	7

你也看到了，由于使用了 ROLLBACK 命令撤消了 INSERT 命令，新的记录并没有被加入到表中。

如果你写了一个图形用户界面的应用程序，比如 MICRO WINDOWS。你可以做一个数据库查询对话框以便让用户在其中输入数值，如果用户按下了《确定》按钮，那么数据库将会保存所做的改动，如果用户按下了《取消》按钮，那么所有的更改就会被取消。显然，这种情况给予了你使用事务处理的机会。

注：下边的代码给出的 ORACLE SQL 中的使用方法，注意这里有 SQL>并且有行号，在随后给出的 Sybase SQL syntax 中则没有 SQL>提示符。

当对话框载入后，这些 SQL 语句将会运行：

INPUT：

SQL> SET TRANSACTION;

SQL> SELECT CUSTOMERS.NAME, BALANCES.CURR_BAL, BALANCES.ACCOUNT_ID

2 FROM CUSTOMERS, BALANCES

3 WHERE CUSTOMERS.NAME = "Rebecca Little"

```
4 AND CUSTOMERS.CUSTOMER_ID = BALANCES.ACCOUNT_ID;
```

该对话框允许用户更改当前的结算账号，所以你需要将该数据返回给数据库。

当按下 OK 按钮以后，UPDATE 将会运行：

INPUT：

```
SQL> UPDATE BALANCES SET CURR_BAL = 'new-value' WHERE ACCOUNT_ID = 6;
```

```
SQL> COMMIT;
```

如果用户按下了 CANCEL，那么将会运行 ROLLBACK 命令。

INPUT：

```
SQL> ROLLBACK;
```

当该对话框在 Sybase SQL 中被载入以后，将会运行下边的语句：

INPUT：

```
1> begin transaction
```

```
2> select CUSTOMERS.Name, BALANCES.Curr_Bal, BALANCES.Account_ID
```

```
3> from CUSTOMERS, BALANCES
```

```
4> where CUSTOMERS.Name = "Rebecca Little"
```

```
5> and CUSTOMERS.Customer_ID = BALANCES.Account_ID
```

```
6> go
```

该对话框允许用户改变当前的结算账号，当你将该数据返回给数据库以后并按下 OK 按钮时，UPDATE 语句将会运行：

INPUT：

```
1> update BALANCES set Curr_BAL = 'new-value' WHERE Account_ID = 6
```

```
2> commit transaction
```

```
3> go
```

如果用户选择了 CANCEL 按钮，那么将会执行 ROLLBACK 的语句：

INPUT：

```
1> rollback transaction
```

```
2> go
```

ROLLBACK 语句将会终止整个事务，当存在嵌套事务时，ROLLBACK 将会终止掉全部事务，系统将会返回到事务开始的最初状态。

如果当前没有活动的事务时，ROLLBACK 或 COMMIT 语句将不会对数据库产生任何

作用（你可以认为这是一个无效的命令）

在 COMMIT 语句运行以后，在事务中的所有动作都会得到确认，这时在使用 ROLLBACK 命令就太晚了。

在事务中使用保存点

在事务中使用 ROLLBACK 可以取消整个的事务，但是你也可以在你的事务当中使用语句进行“部分地确认”。在 Sybase 和 Oracle 中都允许你在当前事务中设一个保存点，从这一点开始，如果你使用了 ROLLBACK 命令，那么系统将会回到保存点时的状态，而在保存点之前的语句将会得到确认，在 ORACLE 中创建一个保存点的语法格式如下：

SYNTAX:

```
SAVEPOINT savepoint_name;
```

在 SYBASE 中创建保存点的语法格式如下：

SYNTAX:

```
save transaction savepoint_name
```

下边是使用 ORACLE 语法的例子：

INPUT:

```
SQL> SET TRANSACTION;
```

```
SQL> UPDATE BALANCES SET CURR_BAL = 25000 WHERE ACCOUNT_ID = 5;
```

```
SQL> SAVEPOINT save_it;
```

```
SQL> DELETE FROM BALANCES WHERE ACCOUNT_ID = 5;
```

```
SQL> ROLLBACK TO SAVEPOINT save_it;
```

```
SQL> COMMIT;
```

```
SQL> SELECT * FROM BALANCES;
```

结算平衡表的内容如下：

Average_Bal	Curr_Bal	Account_ID
1298.53	854.22	1
5427.22	6015.96	2
211.25	190.01	3
73.79	25.87	4
1285.90	25000.00	5

1234.56	1543.67	6
345.25	348.03	7
1250.76	1431.26	8

下边是使用 Sybase 语法的例子：

INPUT:

1> begin transaction

2> update BALANCES set Curr_Bal = 25000 where Account_ID = 5

3> save transaction save_it

4> delete from BALANCES where Account_ID = 5

5> rollback transaction save_it

6> commit transaction

7> go

1> select * from BALANCES

2> go

Average_Bal	Curr_Bal	Account_ID
1298.53	854.22	1
5427.22	6015.96	2
211.25	190.01	3
73.79	25.87	4
1285.90	25000.00	5
1234.56	1543.67	6
345.25	348.03	7
1250.76	1431.26	8

在上边的例子中创建了一个叫 SAVE_IT 的保存点，UPDATE 语句更新了结算平衡表中的 CURR_BAL 列，你在其后设置了一个保存点，在保存之后，你又运行了 DELETE 命令，系统退回到了保存点处，之后你对事务用 COMMIT 命令进行了确认，结果所有在保存点之前的命令得到了确认。

如果你在其后又使用了 ROLLBACK 命令，那么将会取消当前的事务而不会有任何的改变。

在 ORACLE 中的例子如下：

INPUT:

SQL> SET TRANSACTION;

```
SQL> UPDATE BALANCES SET CURR_BAL = 25000 WHERE ACCOUNT_ID = 5;
```

```
SQL> SAVEPOINT save_it;
```

```
SQL> DELETE FROM BALANCES WHERE ACCOUNT_ID = 5;
```

```
SQL> ROLLBACK TO SAVEPOINT save_it;
```

```
SQL> ROLLBACK;
```

```
SQL> SELECT * FROM BALANCES;
```

BALANCE 表的内容如下:

Average_Bal	Curr_Bal	Account_ID
1298.53	854.22	1
5427.22	6015.96	2
211.25	190.01	3
73.79	25.87	4
1285.90	1473.75	5
1234.56	1543.67	6
345.25	348.03	7
1250.76	1431.26	8

Sybase SQL 语法的例子如下:

INPUT:

```
1>begin transaction
```

```
2>update BALANCES set Curr_Bal = 25000 where Account_ID = 5
```

```
3>save transaction save_it
```

```
4>delete from BALANCES where Account_ID = 5
```

```
5>rollback transaction save_it
```

```
6>rollback transaction
```

```
7>go
```

```
1>select * from BALANCES
```

```
2>go
```

Average_Bal	Curr_Bal	Account_ID
1298.53	854.22	1
5427.22	6015.96	2
211.25	190.01	3
73.79	25.87	4

1285.90	1473.75	5
1234.56	1543.67	6
345.25	348.03	7
1250.76	1431.26	8

总结

事务可以被定义为一个有组织的工作单元，事务通常会执行一系列的以前学过的操作。如果由于一些原因使得操作没有如所期望地执行，那么可以在事务中取消这些操作，反之，如果操作全部正确执行了，那么事务中的工作可以确认。

可以使用 ROLLBACK 命令来取消事务，确认事务的命令为 COMMIT，SQL 用非常相似的语法来支持这两类过程，

SYNTAX:

BEGIN TRANSACTION

statement 1

statement 2

statement 3

ROLLBACK TRANSACTION

或

SYNTAX:

BEGIN TRANSACTION

statement 1

statement 2

statement 3

COMMIT TRANSACTION

问与答

问：如果我有一组事务，其中一个是不成功的，我是否可以确认其它的事务过程？

答：不可以，必须整组的事务都是成功的才可以。

问：在使用的 COMMIT 命令以后，我发现我犯了一个错误，那么我怎样才能更正这个错

误？

答：使用 DELETE, INSERT 或 UPDATE 语句，ROLLBACK 在这时是不行的。

问：在怎个事务结束以后我都必须使用 COMMIT 命令确认吗？

答：不必，但是在确认没有错误而且在之前没有事务在运行时使用 COMMIT 会更安全。

校练场

- 1、 在嵌套的事务中，是否可以使用 ROLLBACK 命令来取消当前事务并回退到上级事务中，为什么？
- 2、 使用保存点是否可以保存事务的一部分，为什么？
- 3、 COMMIT 命令是否可以单独使用，它一定要嵌套吗？
- 4、 如果你在 COMMIT 命令后发现的错误，你是否还可以使用 ROLLBACK 命令？
- 4、 在事务中使用保存点是否可以自动地将之前的改动自动地保存？

练习

- 1、 使用 PERSONAL ORACLE7 的语法来更正下边的语法

```
SQL> START TRANSACTION INSERT INTO CUSTOMERS VALUES ('SMITH', 'JOHN')
```

```
SQL> COMMIT;
```

- 2、 使用 PERSONAL ORACLE7 的语法来更正下边的语法

```
SQL> SET TRANSACTION;
```

```
UPDATE BALANCES SET CURR_BAL = 25000;
```

```
SQL> COMMIT;
```

- 3、 使用 PERSONAL ORACLE7 的语法来更正下边的语法

```
SQL> SET TRANSACTION;
```

```
INSERT INTO BALANCES VALUES ('567.34', '230.00', '8');
```

```
SQL> ROLLBACK;
```

第 12 天：数据库安全

今天我们来讨论一下数据库的安全问题，我们已经很清楚地看到不同的 SQL 语句可以使具有管理关系型数据库系统的能力，与我们到今天为止所学习的其它主题一样，一个数据库管理系统是如何在不同的产品中实现安全的呢？对于这个问题我们今天将以流行的 ORACLE 7 数据库系统为例。到今天的结束时我们将具有以下能力：

- 创建用户
- 更改密码
- 创建角色
- 为安全的目的而使用视图
- 在视图使用同义词

前提：数据库管理员

安全问题在数据库的设计过程中常常会被忽略，许多计算机工作人员在进入计算机领域时有计算机的编程知识或硬件知识，并且他们也会将精力注重于这一方面。例如：如果你的老板要求你开展一个新的项目，而这个项目很明显地需要一个关系型的数据库时，那你第一步想要做的工作是什么？在确定的相应的硬件和软件平台以后，你也许会开始设计项目中的基本的数据库结构了，这一阶段会分给许多人去做，其中一些人是图形用户界面设计者，另一些底层组件的设计者。也许你在读过本书以后会要求编写用于提供给客户应用程序所使用查询的 SQL 代码，而与这项任务随之而来的是你可能会成为一个数据库的管理和维护人员。

在很多时候，当我们在真正实施这个项目时我们需要考虑或计划到一点，那就是当众多的用户通过广域网使用你的应用程序时会有什么情况发生？根据今天个人计算机的强大软件和硬件以及微软的开放数据库联接（ODBC），任何连接于网络的用户都会有办法得到你的数据库。（我们不想在当我们公司的计算机准备联入 internet 时或类似的大型网络时有麻烦出现）那么我们应该如何面对这一情况？

非常幸运！软件供应商已经为你处理安全问题提供了许多的工具。每一个新的网络操作系统都会面对着比它的上一代更为严格的安全性要求。此外，许多的数据库供应商也都

在它们的数据库系统提供了不同程度的、与你的网络操作系统安全相独立的安全性，所以想在不同的产品中实现安全性的方法是非常广泛的。

流行的数据库产品与安全

就像你所知道的那样，许多数据库管理系统为你的生意而存在着商业竞争。在一个项目的开发过程中，你可能会只购买少数的许可证用以测试、开发以及其它的目的，但是，你的产品的实际许可证要求可能会是成百上千个。此外，当你决定采用某一种数据库管理系统时，你可能会在这个产品中渡过几年的时间，所以，当你在检查下边的数据库管理系统时头脑中要有下边的概念：

Microsoft FoxPro 数据库管理系统是一个非常强大的基于单用户环境的数据库管理系统，它只使用了有限的 SQL 标准的子集，在该数据库系统中没有提供安全性措施，同时，它使用了 Xbase 的文件格式，每一个文件中都只有一个表，索引文件存储于单独的表中。

Microsoft Access 数据库管理系统提供了更多的 SQL 实现，尽管它内部已经包括了基本的安全系统，但它仍然是一个基于 PC 平台的数据库管理系统。该数据库系统允许你创建查询并把它们存储在数据库之中，此外，全部的数据库及其对象均存在于同一个文件之中。

Oracle 7 数据库管理系统支持全部的标准的 SQL，此外，它还对标准的 SQL 进行了称之为 PL*SQL 的扩充，它拥有全部的安全特性，包括在数据库中创建角色以及为数据库对象分配权限的能力。

Sybase SQL 拥有与 Oracle 7 类似的能力与特性，它也提供了极大范围内的安全特性，它对 SQL 的扩充被称为 Transact-SQL。

对这些产品进行描述的目的是想说明并不是所有的软件都适用于每一个应用程序，如果你的程序用于商业目的，那么你的选择将会受到限制，成本与性能的因素是非常重要的。然而，如果没有足够的安全手段，那么任何在创建数据库之初时的费用节约都将被安全问题吃掉。

如何让一个数据库变得安全

到现在为止你还没有因为数据库的安全问题而不高兴，但是否你曾经因为你不让其它的用户登录进入你的数据库系统造成破坏而不得不非常小心的登录？如果有一天早上你

登录进行系统后发现你辛苦努力的成果都被人删除了你的反应会怎样？（还记得 DROP DATABASE 语句是没有记录的吗？）我们来学习一个流行的数据库管理系统（Personal Oracle 7）是如何实现安全的，当你已经确认你的系统中没有选择 Oracle 7 时你也可能会将这些信息中的大多数应用到其它的数据库管理系统中。

技巧：要带着下边的问题去规划你数据库系统的安全性：

- 谁应该得到数据库管理员权限？
 - 有多少个用户需要访问数据库系统？
 - 每个用户应该得到什么样的权限与角色？
 - 当一个用户不再访问数据库时应该如何去删除它？
-

Personal Oracle7 与安全

Personal Oracle7 通过下边的三个结构来实现安全性：

- 用户
- 角色
- 权限

创建用户

用户就是允许登录进入系统的账号名，创建用户使用的 SQL 语法格式如下：
语法：

```
CREATE USER user  
  
IDENTIFIED {BY password | EXTERNALLY}  
  
[DEFAULT TABLESPACE tablespace]  
  
[TEMPORARY TABLESPACE tablespace]  
  
[QUOTA {integer [K|M] | UNLIMITED} ON tablespace]  
  
[PROFILE profile]
```

如果你选择了需要密码选项，当用户每次登录进行系统时系统都会要求他输入密码，
作为一个例子，下面请为你自己创建一个用户名：

INPUT/OUTPUT：

```
SQL> CREATE USER Bryan IDENTIFIED BY CUTIGER;
```


User created

每次我登录进行系统的名字是 Bryan，我会被要求输入密码 GUTIGER。

如果选择了 EXTERNALLY 选项，ORACLE 将会依赖于你登录进入计算机系统的用户名和密码，也就是说当你登录进行计算机时你就已经登录进行了 ORACLE。

注：一些解释器允许你使用外部的，比如操作系统的密码作为 SQL 的密码（IDENTIFIED externally）。但是，我们推荐你使用 IDENTIFIED BY 子句强制用户在登录进行系统时输入密码（IDENTIFIED BY PASSWORD）。

就像你在 CREATE USER 的其它部分所看到的一样，ORACLE 允许你指定默认的表空间及配额，你可以在 ORACLE 的附带文档中学习到更多的与之相关的内容。

与你在本书中学到的其它的 CREATE 语句一样，它也有 ALTER USER 命令，其语法格式如下：

语法：

```
ALTER USER user
[IDENTIFIED {BY password | EXTERNALLY}]
[DEFAULT TABLESPACE tablespace]
[TEMPORARY TABLESPACE tablespace]
[QUOTA {integer [K|M] | UNLIMITED} ON tablespace]
[PROFILE profile]
[DEFAULT ROLE { role [, role] ...
| ALL [EXCEPT role [, role] ...] | NONE}]
```

你可以使用该命令来改变所有的用户选项，包括密码和配置文件。例如：如果想改变 Bryan 的密码，你可以用下边的语句：

INPUT/OUTPUT：

```
SQL> ALTER USER Bryan
      2 IDENTIFIED BY ROSEBUD;
```

User altered

如果想改变默认的表空间，可以用下边的语句：

INPUT/OUTPUT：

```
SQL> ALTER USER RON
```

```
2 DEFAULT TABLESPACE USERS;
```

User altered.

如果想删除一个用户，只需简单地使用 DROP USER 命令即可，它将会把用户从数据库的清除，该命令的语法格式如下：

SYNTAX:

```
DROP USER user_name [CASCADE];
```

如果你使用了 CASCADE 选项，那么所有与用户账号相关的对象也将会被删除，否则的话对象将仍归该用户所有，但是用户将不再有效。这有点让人迷糊，但是当你只想删除用户时它很有用。

创建角色

角色是允许用户在数据库中执行特定功能的一个或一组权限，将角色应用于用户的语法如下：

SYNTAX:

```
GRANT role TO user [WITH ADMIN OPTION];
```

如果你使用了 WITH ADMIN OPTION 选项，那么该用户可以为其它用户赋予权限，功能是不是很强大？

如果想删除角色，可以使用 REVOKE 命令：

SYNTAX:

```
REVOKE role FROM user;
```

当你使用你早些时候创建的账号登录进入系统时，你会为你的许可权问题而费尽精力。你可以登录进入，但这也是你所能做的全部内容。ORACLE 可以让你用下边的三个角色之一进行注册：

- Connect
- Resource
- DBA（也就是数据库管理员）

这三个角色有着不同程度的权限。

注：如果你有适当的权限，你可以创建你自己的角色，为你的角色赋予权限，然后将角色应用于你的用户以取得更高的安全性。

Connect 角色

你可以将 Connect 角色理解为登录级角色，被赋予该角色的用户可以登录进入系统并做允许他/她们做的工作。

INPUT/OUTPUT:

```
SQL> GRANT CONNECT TO Bryan;
```

```
Grant succeeded.
```

Connect 角色允许用户从表中插入，更新，删除属于其它用户的记录（在取得了适当的许可权限以后）用户也可以创建表、视图、序列、簇和同义词。

Resource 角色

该角色允许用户对 ORACLE 数据库进行更多的访问，除了可以赋予 Connect 角色的权限以外，它还有创建过程，触发机制和索引的权限。

INPUT/OUTPUT:

```
SQL> GRANT RESOURCE TO Bryan;
```

```
Grant succeeded.
```

DBA 角色

DBA 角色包括了所有的权限，赋予了该角色的用户可以在数据库中做他们想做的任何事。为了保证系统的完整性你应该将具有该角色的用户数量保持在仅有的少数几个上。

INPUT/OUTPUT:

```
SQL> GRANT DBA TO Bryan;
```

```
Grant succeeded.
```

在经过了这三步之后，用户 Bryan 分别应用了 connect、resource 和 DBA 角色，这似乎是多余的，因为 DBA 角色就包括了其它的两个角色，所有现在我们可以把它删除掉。

INPUT/OUTPUT:

```
SQL> REVOKE CONNECT FROM Bryan;
```

```
Revoke succeeded.
```

```
SQL> REVOKE RESOURCE FROM Bryan;
```

```
Revoke succeeded.
```

拥有 DBA 角色的 Bryan 现在可以做他想做的任何工作。

用户权限

在你决定好对你的用户应用何种角色以后，你需要决定让你的用户具有使用哪种数据库对象的权利（ORACLE 中称之为许可权限）该权限依据你为用户所定的角色的不同而不同，如果你创建了一个对象，你可以将该对象的许可权赋予其它的用户以使他们也具有访问权，ORACLE 定义了两种可以赋予用户的权利——系统许可权与对象许可权。（见表 12.1 和表 12.2）

系统许可权应用于整个系统，赋予系统许可权的语法如下：

SYNTAX:

```
GRANT system_privilege TO {user_name | role | PUBLIC}
[WITH ADMIN OPTION];
```

如果使用了 WITH ADMIN OPTION 选项就允许拥有该权限的人将该权限应用给其它的用户。

用户使用视图的权利

下边的命令将允许系统中的所有用户都具有在自己的模块中创建视图和访问视图的能力。

INPUT:

```
SQL> GRANT CREATE VIEW TO PUBLIC;
```

OUTPUT:

```
Grant succeeded.
```

分析:

PUBLIC 关键字的意思就是每个人都有创建视图的权利，很明显，系统权限允许受权人访问几乎全部的系统的设置。所有系统权限只应只能给予特定的人或需要使用系统权限的人，表 12.1 显示出了你可以在 ORACLE 7 的帮助文件中找到的系统权限。

警告：当赋予权限给 PUBLIC 时必须小心，它可能会给所有用户以访问数据库的权限，尽管有一些人你是不想让他拥有这一权限的。

表 12.1 在 ORACLE 7 中的系统权限

System Privilege	Operations Permitted
ALTER ANY INDEX	允许授权人更改任何模块中的索引。
ALTER ANY PROCEDURE	允许授权人更改任何的存储过程、函数并打包到任何模块中。
ALTER ANY ROLE	允许授权人更改数据库中的任何角色。
ALTER ANY TABLE	允许授权人更改模块中的表和视图。
ALTER ANY TRIGGER	允许授权人能够、不能或编译任何模块中的触发机制。
ALTER DATABASE	允许授权人改变数据库。
ALTER USER	允许授权人更改用户，该权限允许授权人改变其它用户的密码或鉴定方法，指定配额或表空间，设置默认的或临时的表空间。指定一个profile或默认角色。
CREATE ANY INDEX	允许授权人在任何模块的任何表中创建索引。
CREATE ANY PROCEDURE	允许授权人创建存储过程，函数并打包到任何模块中。
CREATE ANY TABLE	允许授权人在任何模块内创建表，创建表的模块的所有者必须拥有配额及表空间以存放表。
CREATE ANY TRIGGER	允许授权人在与任何模块相关联的表的模块中创建触发机制。
CREATE ANY VIEW	允许授权人在任何模块中创建视图
CREATE PROCEDURE	允许授权人创建存储过程，函数并打包到他们自己的模块中。
CREATE PROFILE	允许授权人创建profiles。
CREATE ROLE	允许授权人创建角色。
CREATE SYNONYM	允许授权人在他们自己的模块中创建同义字
CREATE TABLE	允许授权人在他们自己的模块中创建表，要创建一个表，授权人必须有表空间配额以存放表。
CREATE TRIGGER	允许授权人在他们自己的模块中创建数据库触发机制。
CREATE USER	允许授权人创建用户，该权限也允许创建人指定表空间配额，设置默认和临时的表空间，并指定profile作为CREATE USER语句的一部分。
CREATE VIEW	允许授权人在他们自己的模块中创建视图。
DELETE ANY TABLE	允许授权人从任何模块的表和视图中删除行或截表。
DROP ANY INDEX	允许授权人从任何模块中删除索引。
DROP ANY PROCEDURE	允许授权人删除任何模块中的包，存储过程及函数。
DROP ANY ROLE	允许授权人删除角色。
DROP ANY SYNONYM	允许授权人删除任何模块中的同义字权限。
DROP ANY TABLE	允许授权人删除任何模块中的表。
DROP ANY TRIGGER	允许授权人删除任何模块中的数据库触发机制。
DROP ANY VIEW	允许授权人删除任何模块中的视图。
DROP USER	允许授权人删除用户。
EXECUTE ANY PROCEDURE	允许授权人执行过程或函数（独立的或打包的）或引用任何模块中的全局变量。
GRANT ANY PRIVILEGE	使授权人具有全部系统特权。
GRANT ANY ROLE	使授权人具有全部数据库角色。
INSERT ANY TABLE	允许授权人在任何模块的表或视图中插入行。
LOCK ANY TABLE	允许授权人锁定任何模块中的表和视定。
SELECT ANY SEQUENCE	允许授权人引用其它模块中的序列。
SELECT ANY TABLE	允许授权人查询任何模块中的表、视图或进行映像。
UPDATE ANY ROWS	允许授权人更新表中的行。

对象权限是指可以在数据库中使用的对象的权限，表 12.2 中给出了所有了 ORACLE 中的对象权限。

ALL
ALTER
DELETE
EXECUTE
INDEX
INSERT
REFERENCES
SELECT
UPDATE

你可以使用下边的 GRANT 语句来对其它用户授权访问你的表。

SYNTAX:

```
GRANT {object_priv | ALL [PRIVILEGES]} [ (column
[, column]...) ]
[, {object_priv | ALL [PRIVILEGES]} [ (column
[, column] ...) ] ] ...
ON [schema.]object
TO {user | role | PUBLIC} [, {user | role | PUBLIC}] ...
[WITH GRANT OPTION]
```

如果你想取消对某个对象对于某人的授权，你可以使用 REVOKE 语句，语法如下：

SYNTAX:

```
REVOKE {object_priv | ALL [PRIVILEGES]}
[, {object_priv | ALL [PRIVILEGES]} ]
ON [schema.]object
FROM {user | role | PUBLIC} [, {user | role | PUBLIC}]
[CASCADE CONSTRAINTS]
```

从建表到角色授权

创建一个名字为 SALARIES 的表，结构如下：

INPUT:

```
SQL> CREATE TABLE SALARIES (
2  NAME CHAR(30),
3  SALARY NUMBER,
4  AGE NUMBER);
```

OUTPUT:

Table created.

现在，来创建两个用户，Jack 的 Jill:

INPUT/OUTPUT:

```
SQL> create user Jack identified by Jack;
```

User created.

```
SQL> create user Jill identified by Jill;
```

User created.

```
SQL> grant connect to Jack;
```

Grant succeeded.

```
SQL> grant resource to Jill;
```

Grant succeeded.

分析：

到现在为止，你已经创建了两个用户，并且为每个用户分了不同的角色。因此，当他们在数据库中进行工作时有着不同的能力。在最初创建的表中有如下内容：

INPUT/OUTPUT：

```
SQL> SELECT * FROM SALARIES;
```

NAME	SALARY	AGE
JACK	35000	29
JILL	48000	42
JOHN	61000	55

在本例中你可以按着自己的意愿为该表分配不同的权限，我们假定你具有 DBA 角色因而具有系统中的一切权利，即使你不是 DBA 角色，你仍然可以对 SALARIES 表进行对象授权，因为你是它的所有者。

由于 JACK 的角色了 Connect，所以你想让他有使用 SELECT 语句的权利。

INPUT/OUTPUT：

```
SQL> GRANT SELECT ON SALARIES TO JACK;
```

Grant succeeded.

因为 JILL 的角色为 Resource，你允许他对表进行选择 and 插入一些数据，或是严格一些，允许 JILL 修改 SALARIES 表中 SALARY 字段的值。

INPUT/OUTPUT：

```
SQL> GRANT SELECT, UPDATE(SALARY) ON SALARIES TO Jill;
```

Grant succeeded.

现在表和用户都已经创建了，你需要看一下创建的用户在访问表时的不同之处，JACK 和 JILL 都有对 SALARIES 表执行 SELECT 的权限，可是，如果是 JACK 访问表，他可能会被告知该表不存在，因为 ORACLE 的表名之前需要知道表所有的用户名或计划名。

使用表时的限制

这里需要说明一下，你在创建表的时候所使用的用户名假定为 Byran，当 JACK 想从 SALARIES 表中选择数据库，他必须使用该用户名：

INPUT：

```
SQL> SELECT * FROM SALARIES;
```

OUTPUT：

ERROR at line 1:

ORA-00942: table or view does not exist

这里 JACK 被告知该表并不存在，现在对表使用用户名来加以标识：

INPUT/OUTPUT：

```
SQL> SELECT * FROM Bryan.SALARIES;
```

NAME	SALARY	AGE
JACK	35000	29
JILL	48000	42
JOHN	61000	55

分析：

你可以看到现在查询已经工作了，现在我们来测试一下 JILL 的访问权限，退出 JACK 的登录并以 JILL 的身份登录：

INPUT/OUTPUT：

```
SQL> SELECT * FROM Bryan.SALARIES;
```

NAME	SALARY	AGE
JACK	35000	29
JILL	48000	42
JOHN	61000	55

工作正常，现在试着向表中插入一个新的记录：

INPUT/OUTPUT：

```
SQL> INSERT INTO Bryan.SALARIES VALUES('JOE',85000,38);
```

ERROR at line 1:

ORA-01031: insufficient privileges

分析：

该操作并没有被执行，因为 JILL 没有在 SALARIES 表中使用 INSERT 语句的权限。

INPUT/OUTPUT:

```
SQL> UPDATE Bryan.SALARIES SET AGE = 42 WHERE NAME = 'JOHN';
```

ERROR at line 1:

ORA-01031: insufficient privileges

分析:

还是不能执行，JILL 只能做他权利范围之内的事情，事实上，ORACLE 非常快地捕捉到了错误并反馈给了她。

INPUT/OUTPUT:

```
SQL> UPDATE Bryan.SALARIES SET SALARY = 35000 WHERE NAME = 'JOHN';
```

1 row updated.

```
SQL> SELECT * FROM Bryan.SALARIES;
```

NAME	SALARY	AGE
JACK	35000	29
JILL	48000	42
JOHN	35000	55

分析:

你看到了，JILL 可以进行她所有权限范围内的更新工作。

为安全的目的而使用视图

我们在第十天的《创建视图和索引》中曾经提到过视图其实是一种虚表，它可以为用户提供一种以真实的数据并不相同的显示方式。今天，我们将学习更多的使用视图来实现安全性的方法。但是，首先，我们需要解释一下视图为什么可以让 SQL 的语句简单化。

在早些时候我们曾经学过当一个用户访问一个不为他所有的表的时候，目标必须引用它所属用户的名字方可正常访问，就像你所想的一样，当你将多个 SQL 语句写在一行时它将会变得非常的冗长，更重要的是，初学者在可以查看表的内容之前要先知道表的所属用户名，这并不是你想让你的用户做的工作。下边给出了一个非常简单的解决方案。

使用表或视图时限制的解决方法

假如你是以 JACK 的身份登录进行系统的，你从早些时候的内容中了解到如果你想查看表中的内容，你必须使用下边的语句：

INPUT：

```
SQL> SELECT * FROM Bryan.SALARIES;
```

OUTPUT：

NAME	SALARY	AGE
JACK	35000	29
JILL	48000	42
JOHN	35000	55

如果你创建了一个名字叫 SELECT_VIEW 和视图，那么用户可以非常简单地使用这个视图。

INPUT/OUTPUT：

```
SQL> CREATE VIEW SALARY_VIEW AS SELECT * FROM Bryan.SALARIES;
```

View created.

```
SQL> SELECT * FROM SALARY_VIEW;
```

NAME	SALARY	AGE
JACK	35000	29
JILL	48000	42
JOHN	35000	

分析：

上边的查询返回的结果与使用用户名的返回结果是相同的。

用同义词取代视图

SQL 还提供了一种叫同义词的对象，同义词可以为表提供一个别名以将击键的次数减到最小。同义词有两种，公有的和私有的。任何一个具有 Resource 角色的用户都可以创建私有类型的同义词，与之相对应的是只有 DBA 角色的用户才能够创建公有类型的同义词。

创建公有类型的同义词的语法如下：

SYNTAX：

```
CREATE [PUBLIC] SYNONYM [schema.]synonym FOR [schema.]object[@dblink]
```

针对前一个例子，你可以使用下边的语句来取得相同的效果：

INPUT/OUTPUT：

```
SQL> CREATE PUBLIC SYNONYM SALARY FOR SALARIES
```

Synonym created.

然后再以 JACK 的身份登录并输入：

INPUT/OUTPUT：

```
SQL> SELECT * FROM SALARY;
```

NAME	SALARY	AGE
JACK	35000	29
JILL	48000	42
JOHN	35000	55

使用视图来解决安全问题

假定你现在又改变主意了，你不想让 JACK 和 JILL 看到 SALARIES 表的全部内容，你可以使用视图来达到只使他们看到属于他们自己的信息这一目的。

INPUT/OUTPUT：

```
SQL> CREATE VIEW JACK_SALARY AS
```

```
2 SELECT * FROM BRYAN.SALARIES
```

```
3 WHERE NAME = 'JACK';
```

View created.

INPUT/OUTPUT：

```
SQL> CREATE VIEW JILL_SALARY AS
```

```
2 SELECT * FROM BRYAN.SALARIES
```

```
3 WHERE NAME = 'JILL';
```

View created.

INPUT/OUTPUT：

```
SQL> GRANT SELECT ON JACK_SALARY TO JACK;
```

Grant succeeded.

INPUT/OUTPUT：

```
SQL> GRANT SELECT ON JILL_SALARY TO JILL;
```

Grant succeeded.

INPUT/OUTPUT:

SQL> REVOKE SELECT ON SALARIES FROM JACK;

Revoke succeeded.

INPUT/OUTPUT:

SQL> REVOKE SELECT ON SALARIES FROM JILL;

Revoke succeeded.

现在，以 JACK 的身份登录并测试你为他创建的视图。

INPUT/OUTPUT:

SQL> SELECT * FROM Bryan.JACK_SALARY;

NAME	SALARY	AGE
Jack	35000	29

INPUT/OUTPUT:

SQL> SELECT * FROM PERKINS.SALARIES;

ERROR at line 1:

ORA-00942: table or view does not exist

退出 JACK 登录并以 JILL 身份登录来测试 JILL:

INPUT/OUTPUT:

SQL> SELECT * FROM Bryan.JILL_SALARY;

NAME	SALARY	AGE
Jill	48000	42

ANALYSIS:

你可以看到对 SALARIES 表的访问将完全受到视图的控制，SQL 允许你创建这些视图并把它提供给你的用户，这项技术为你提供了相当大的灵活性:

删除同义词的语法如下:

SYNTAX:

SQL> drop [public] synonym synonym_name;

注：到现在为止，你应该明白保持有 DBA 角色的用户最少的重要性了吧！因为具有这一角色的用户可以运行数据库中的任何命令及操作。但是请注意，在 ORACLE 和 Sybase 中你只有成为 DBA 角色的用户才可以从数据库中引入或导出数据。

使用 WITH GRANT OPTION 子句

如果 JILL 想把她的 UPDATE 权限赋给 JACK 时需要谁来完成这项工作？最初你可能会认为应该由 JILL 来完成，因为她有 UPDATE 权限。她应该可以为其他用户授予这个权限。但是，如果你使用早些时候的 GRANT 语句，JILL 并不能为其他用户授权。

```
SQL> GRANT SELECT, UPDATE(SALARY) ON Bryan.SALARIES TO Jill;
```

下边是我们在今天早些时候讲过的 GRANT 语句的语法：

SYNTAX:

```
GRANT {object_priv | ALL [PRIVILEGES]} [ (column [, column]...) ]  
[, {object_priv | ALL [PRIVILEGES]} [ (column[, column] ...) ] ] ...  
ON [schema.]object TO {user | role | PUBLIC} [, {user | role | PUBLIC}] ...  
[WITH GRANT OPTION]
```

看到在末尾的 WITH GRANT OPTION 语句了吗？当在给对象授权时如果使用了这个选项，那么该权限就可以被传给其他的用户。所以，如果你想让 JILL 具有给 JACK 授权的能力，那么你应该像下边这样使用 GRANT 语句：

INPUT:

```
SQL> GRANT SELECT, UPDATE(SALARY)  
2 ON Bryan.SALARIES TO JILL  
3 WITH GRANT OPTION;
```

OUTPUT:

```
Grant succeeded.
```

当以 JILL 的身份登录时就可以使用下边的语句：

INPUT/OUTPUT:

```
SQL> GRANT SELECT, UPDATE(SALARY) ON Bryan.SALARIES TO JACK;  
Grant succeeded.
```

总结

如果数据库的管理人员如果对数据库的安全考虑不周常常会导致许多问题，幸运的是 SQL 提供了几个非常有用的命令来实现数据库的安全性。

用户最初可由 CREATE USER 命令来创建，这里可以设定用户名和密码，当用户的帐号生效以后，必须为其指定角色以使其可以工作，在 ORACLE 中有三种可用的角色，分别了 Connect、Resource 和 DBA 角色，每种角色有不同的访问数据库的资格，Connect 最少而 DBA 则拥有全部的访问能力。

GRANT 命名可以对用户进行授权，REVOKE 命令则可以取消对用户的授权。权限可分为对象权限和系统权限。系统权限应该严格控制，不能授予没有使用经验的用户。如果他们得到了这种权限他们就可能（也许是不经意之间）毁坏你辛苦构建的数据库。对象权限则可以让用户具有访问个别的由已存在用户所创建的模块中的对象的能力。

所有的这些技术和语句都为 SQL 的用户提供了相当广泛的用以设置安全性的工具，尽管我们讨论的主要是 ORACLE 7 的安全特性，但是你可以在设计你自己的数据库时用到这些信息，切记不论你使用哪一种数据库产品，它都提供了一定程度的安全性。

问与答

问：我知道安全性是需要的，但是否 ORACLE 做的太多了？

答：不，一点也不多！尤其是当在大型的多用户应用场合时更是这样，由于使用数据库的不同用户所做的工作也并不相同，所以你需要限制用户让他们能做什么或不能做什么。用户应该只能进行他所处角色和权限许可内的工作。

问：看来 DBA 用户在创建我的帐号时已经知道了我的密码，是否是这样？这是一个安全问题！

答：确实是这样，DBA 创建了你的用户和密码，所以你应该在收到创建信息以后立即使用 ALTER USER 命令来更改 ID 和密码。

校练场

1、下边的语句是否是错误的？

```
SQL> GRANT CONNECTION TO DAVID;
```

2、对与错：当删除用户时所有属于用户对对象都会随之删除。

3、如果你创建了一个表并对它使用了 SELECT 权限对象为 PUBLIC 时会有什么问题？

4、下边的 SQL 语句是否正确？

```
SQL> create user RON identified by RON;
```

5、下边的 SQL 语句是否正确？

```
SQL> alter RON identified by RON;
```

6、下边的 SQL 语句是否正确？

```
SQL> grant connect, resource to RON;
```

7、如果表为你所有，别人如何才能从表中选择数据？

练习

作为数据库安全性的练习，请你创建一个表，然后再创建一个用户，为该用户设置不同的安全性并测试。

第 13 天 高级 SQL

目标：

在之前的 12 天中，我们学习了许多关于如何写出强大的 SQL 查询以从数据库中获得数据。我们也简要地学习了如何进行数据库的设计以数据库安全性问题。而在今天的高级 SQL 部分，我们将主要学习以下内容：

- 临时表
- 游标
- 存储过程
- 触发机制
- 内嵌 SQL

注：在今天的例子中我们使用 ORACLE 的 PL/SQL 和 MicroSoft SQL Server 的 Transact-SQL 来实现，我们尽可能使我们的例子可以适用于这两种风格的 SQL。你不必为此需要得到一个 ORACLE 或 SQL Server 的副本，实际的数据库版本的选择是依据你的需要而定的。（如果你已经阅读了足够的可以创建一个项目所需要的知识，那你是没有选择的机会的。）

注：尽管你可以在大多数的数据库产品上应用本书中所给出的例子，但是今天的内容并不适用于这句话。许多数据库供应商仍没有提供临时表、存储过程以及触发机制（请检查你的数据库文档，看你所喜爱的数据库系统是否支持这些特性）。

临时表

我们要讨论的第一个高级主题是临时表的用法，这是一种简单的临时存在于数据库系统当中的表格，当结束数据库的联接或退出登录以后它们会被自动地删除。Transact-SQL 在 TempDB 中创建临时表，这个数据库是在你安装 SQL-SERVER 时创建的，创建临时表可以使用两种语法格式。

SYNTAX:

SYNTAX 1:


```
create table #table_name (field1 datatype,  
.  
.  
.  
fieldn datatype)
```

语法 1 用以在 TempDB 中创建一个表。该表由 Create Table 命令以及创建表时的日期和时间组合而成一个唯一的表名，临时表只可由它的创建者使用，五十个用户可以在同时运行下边的命令：

```
1> create table #albums (  
2> artist char(30),  
3> album_name char(50),  
4> media_type int)  
5> go
```

表名开头的#标志是 SQL 用以标识临时表的标志，五十个用户中每一个都可以获得一个他可以使用的表，每一个用户都可以放心地插入、更新、删除表中的数据而不必担心其它的用户使该表中的数据失效。该表也可以使用下边的命令来手动删除：

```
1> drop table #albums  
2> go
```

当用户退出 SQL-SERVER 时该表也可以被自动地删除，如果你是在自态 SQL 联接情况下使用该语句（例如 SQL-SERVER 的 DB-LIBRARY），那么当动态联接被终止时该表也会被自动地删除。

语法 2 给出了另一种在 SQL-SERVER 中创建临时表的方法，该语法与使用语法 1 有着不同的结果，所以你要注意这两种语法之间的差别：

SYNTAX:

```
SYNTAX 2:  
  
create table tempdb..tablename (field1 datatype,  
.  
.  
fieldn datatype)
```

使用语法 2 来创建临时表的结果与使用语法 1 相同，临时表的名称格式也与语法 1 的相同，它们的不同之处在于当用户退出 SQL 或切断联接时该表不会被自动地删除。用户必

须使用 DROP 命令在 TEMPDB 中将其手动删除。

技巧：另外一种将使用语法 2 创建的临时表删除的方法是将 SQL-SERVER 关闭并重新启动，这将会把所有在 TEMPDB 中的表都删除掉。

例 13.1 和 13.2 表明使用这两种格式的临时表的确是货真价实的临时的表，在这两个例子以后，例 13.3 给出的临时的最为通常的用途，用于暂时存贮从查询中返回的数据。这些数据可以在其它的查询中使用。

为了验证这些例子你需要创建一个数据库，在 MUSIC 数据库中需要创建以下三个表：

- ARTISTS
- MEDIA
- RECORDINGS

创建这些表需要使用下边的 SQL 语句：

INPUT：

```

1> create table ARTISTS (                                4> price float)
2> name char(30),                                       5> go
3> homebase char(40),                                  1> create table RECORDINGS (
4> style char(20),                                       2> artist_id int,
5> artist_id int)                                       3> media_type int,
6> go                                                  4> title char(50),
1> create table MEDIA (                                5> year int)
2> media_type int,                                       6> go
3> description char(30),

```

注：表 13.1、13.2、13.3 给出的这些表中的示例数据。

表 13.1

Name	Homebase	Style	Artist_ID
Soul Asylum	Minneapolis	Rock	1
Maurice Ravel	France	Classical	2
Dave Matthews Band	Charlottesville	Rock	3
Vince Gill	Nashville	Country	4
Oingo Boingo	Los Angeles	Pop	5
Crowded House	New Zealand	Pop	6
Mary Chapin-Carpenter	Nashville	Country	7

Edward MacDowell	U.S.A.	Classical	8
------------------	--------	-----------	---

表 13.2

Media_Type	Description	Price
1	Record	4.99
2	Tape	9.99
3	CD	13.99
4	CD-ROM	29.99
5	DAT	19.99

表 13.3

Artist_Id	Media_Type	Title	Year
1	2	Hang Time	1988
1	3	Made to Be Broken	1986
2	3	Bolero	1990
3	5	Under the Table and Dreaming	1994
4	3	When Love Finds You	1994
5	2	Boingo	1987
5	1	Dead Man's Party	1984
6	2	Woodface	1990
6	3	Together Alone	1993
7	5	Come On, Come On	1992
7	3	Stones in the Road	1994
8	5	Second Piano Concerto	1985

例 13.1

你可以在 TEMPDB 数据库中创建一个临时表，在向这些表中插入一些虚拟的数据以后，退出登录，然后再重新登录 SQL SERVER，试着从表中选取临时的数据，注意结果：

INPUT:

```
1> create table #albums (
2> artist char(30),
3> album_name char(50),
4> media_type int)
5> go
1> insert #albums values ("The Replacements", "Pleased To Meet Me", 1)
2> go
```

现在请使用 EXIT（或者 QUIT）来退出 SQL SERVER 的联接，在重新登录并选择了你在上一次时使用的数据库以后，试一下下边的命令：

INPUT:

```
1> select * from #albums
```

```
2> go
```

分析：

在当前的数据库中并不存在该表。

例 13.2

现在使用语法 2 来创建表：

INPUT:

```
1> create table tempdb..albums (
```

```
2> artist char(30),
```

```
3> album_name char(50),
```

```
4> media_type int)
```

```
5> go
```

```
1> insert #albums values ("The Replacements", "Pleased To Meet Me", 1)
```

```
2> go
```

在退出登录并重新登录进入以后，切换到你在 CREATE TABLE TEMPDB..ALBUMS ()

命令中指定的数据库，然后请验证下边的命令：

INPUT:

```
1> select * from #albums
```

```
2> go
```

这次，你会得到下边的结果：

OUTPUT:

Artist	Album_name	media_type
The Replacements	Pleased To Meet Me	1

例 13.3

本例给出一临时表的最为通常的用法：在复合查询中存贮查询的结果为之后的查询使用。

INPUT:

```
1> create table #temp_info (  
2> name char(30),  
3> homebase char(40),  
4> style char(20),  
5> artist_id int)  
6> insert #temp_info  
7> select * from ARTISTS where homebase = "Nashville"  
8> select RECORDINGS.* from RECORDINGS, ARTISTS  
9> where RECORDINGS.artist_id = #temp_info.artist_id  
10> go
```

上边的这一组命令选出了所有的居住在 Nashville 的艺术家的记录信息。

下边的这此命令则是例 13.3 语句的另外一种写法：

```
1> select ARTISTS.* from ARTISTS, RECORDINGS where ARTISTS.homebase = "Nashville"  
2> go
```

游标

数据库指针类似于字处理程序中的指针。当你按下方向键时，游标依次从各行文本中滚动。按一下向上键游标向上跳一行，而按 PageUp 和 PageDown 则会向一次翻阅几行，数据库游标的操作也类似。

数据库游标允许你选择一组数据，通过翻阅这组数据记录（通常被称为数据集），检查每一个游标所在的特定的行。你可以将游标和局部变量组合在一起对每一个记录进行检查，当游标移动到下一个记录时来执行一些外部操作。

游标的另一个常见的用法是保存查询结果以备以后使用。一个游标结果集是通过执行 SELECT 查询来建立的。如果你的应用程序或过程需要重复使用一组记录，那么第一次建立游标以后再重复使用将会比多次执行查询快得多。（而且你还有在查询的结果集中翻阅的好处）

下边是创建、使用和关闭数据库游标的例子：

1. Create the cursor.
2. Open the cursor for use within the procedure or application.

3. Fetch a record's data one row at a time until you have reached the end of the cursor's records.
4. Close the cursor when you are finished with it.
5. Deallocate the cursor to completely discard it.

创建游标

如果使用 Transcat-SQL 来创建游标，其语法如下：

SYNTAX:

```
declare cursor_name cursor
    for select_statement
    [for {read only | update [of column_name_list]]]
```

使用 ORACLE7 的 SQL 来创建和语法格式则如下：

SYNTAX:

```
DECLARE cursor_name CURSOR
    FOR {SELECT command | statement_name | block_name}
```

在执行 DECLARE cursor_name CURSOR 语句时，你必须同时定义将要在你的所有的游标操作中使用的结果集，一个游标有两个重要的部分：游标结果集和游标的位置。

下边的语句将创建一个基于 ARTIST 表的结果集：

INPUT:

```
1> create Artists_Cursor cursor
2> for select * from ARTISTS
3> go
```

分析：

你现在已经有了一个名字为 ARTIST_Cursor 游标，它包括了所有的 ARTIST 表的内容。但是首先你必须打开游标。

打开游标

最简单的打开游标命令如下：

SYNTAX:

```
open cursor_name
```

运行下列命令打开 ARTIST_Cursor 游标：

```
1> open Artists_Cursor
```

```
2> go
```

现在你可以使用游标来翻阅结果集了。

使用游标来进行翻阅

要想在游标结果集中进行翻阅操作，Transcat-SQL 提供了 FETCH 命令。

SYNTAX:

```
fetch cursor_name [into fetch_target_list]
```

ORACLE SQL 则提供了下边的语法：

```
FETCH cursor_name { INTO : host_variable
                    [[INDICATOR] : indicator_variable]
                    [,  : host_variable
                    [[INDICATOR] : indicator_variable] ]...
                    | USING DESCRIPTOR descriptor }
```

每次当 FETCH 命令运行时，游标指针的好处是每次可以在结果集中移动一行，如果需要，移动到行的数据可以被填充到 fetch_target_list 变量中。

注：Transcat-SQL 允许程序员通过下边的命令来实现一次移动多行：

```
set cursor rows number for cursor_name
```

该命令不能使用 INTO 子句，但是，当向前跳动的行数已知时用它来代替重复执行 FETCH 命令则很有用。

下边的语句将从 ARTIST_Cursor 的结果集中获得数据并把它返回给程序变量：

INPUT:

```
1> declare @name char(30)
```

```
2> declare @homebase char(40)
```

```
3> declare @style char(20)
```

```
4> declare @artist_id int
```

```
5> fetch Artists_Cursor into @name, @homebase, @style, @artist_id
```

```
6> print @name
```

```
7> print @homebase
8> print @style
9> print char(@artist_id)
10> go
```

你可以使用 WHILE 循环来循环查看整个结果集，但是你是如果知道已经到达了最后一个记录的呢？

测试游标的状态

Transcat-SQL 可以让你在任何时候通常维护：@@sqlstatus 和 @@rowcount 这两个全局变量来检查当前游标的状态。

变量@@sqlstatus 返回最后一次运行 FETCH 语句的状态信息，（Transcat-SQL 规定除了 FETCH 命令以外其他的命令不得修改：@@sqlstatus 变量）该变量可以取下表三个值中的一个。下表是在 Transcat-SQL 参考手册中给出的。

Status	Meaning
0	Successful completion of the FETCH statement.
1	The FETCH statement resulted in an error.
2	There is no more data in the result set.

而变量：@@rowcount 则返回上次一 FETCH 命令设置的行号，你可以用它来确定当前游标结果集的行数。

下边的代码给出了 FETCH 命令的扩充使用方法，你现在可以使用 While Loop 命令和变量@@sqlstatus 来翻阅当前的游标。

INPUT:

```
1> declare @name char(30)
2> declare @homebase char(40)
3> declare @style char(20)
4> declare @artist_id int
5> fetch Artists_Cursor into @name, @homebase, @style, @artist_id
6> while (@@sqlstatus = 0)
7> begin
8>     print @name
```



```
9>      print @homebase

10>      print @style

11>      print char(@artist_id)

12>      fetch Artists_Cursor into @name, @homebase, @style, @artist_id

13> end

14> go
```

分析：

现在你已经有了一个全功能的游标，下边要做的工作就是关闭游标。

关闭游标

关闭游标是一个非常简单的工作，它的语句如下：

SYNTAX:

```
close cursor_name
```

这时游标依然存在；但是，它必须被再次打开方可使用。关闭一个游标从本质上来说是关闭了它的结果集，而并不是它的全部内容。如果你已经完全结束了对一个游标的使用的话，DEALLOCATE 命令将释放让游标所占用的内存并且可以让游标的名字可以被再次使用。这是该命令的语法格式：

SYNTAX:

```
deallocate cursor cursor_name
```

例 13.4 给出了用 Transcat-SQL 写的创建、使用、关闭、释放一个游标的完整过程：

Example 13.4

INPUT:

```
1> declare @name char(30)

2> declare @homebase char(40)

3> declare @style char(20)

4> declare @artist_id int

5> create Artists_Cursor cursor

6> for select * from ARTISTS

7> open Artists_Cursor
```

```

8> fetch Artists_Cursor into @name, @homebase, @style, @artist_id
9> while (@@sqlstatus = 0)
10> begin
11>     print @name
12>     print @homebase
13>     print @style
14>     print char(@artist_id)
15>     fetch Artists_Cursor into @name, @homebase, @style, @artist_id
16> end
17> close Artists_Cursor
18> deallocate cursor Artists_Cursor
19> go

```

注：下边是示例所用的数据。

OUTPUT:

Soul Asylum	Minneapolis	Rock	1
Maurice Ravel	France	Classical	2
Dave Matthews Band	Charlottesville	Rock	3
Vince Gill	Nashville	Country	4
Oingo Boingo	Los Angeles	Pop	5
Crowded House	New Zealand	Pop	6
Mary Chapin-Carpenter	Nashville	Country	7
Edward MacDowell	U.S.A.	Classical	8

游标的适用范围

与表、索引以及其它的对象如触发机制和存贮过程不同，游标在创建以后并不作为一个数据库对象来看待，所以，游标的使用会受到一些限制。

警告：切记，无论何时要注意游标分配过的内存，尽管它的名字可能已经不存在了。当不在使用游标的时候（或在进行游标能力之外的工作时），一定要记得关闭游标并将它释放掉。

可以在下列三种情况下创建游标：

- 在会话中——会话在用户登录以后开始。如果用户在登录进行 SQL SERVER 以后创建了一个游标，那么游标的名字将一直存在到用户退出登录，用户不能再一次使用在本

次登录中创建的游标名。

- 在存贮过程中——游标在存贮过程的内部创建的好处在于只有当过程运行时它才真正起作用，一旦过程退出了。则游标的名字将不再有效。
- 在触发机制中——在触发机制中创建游标与在存贮过程中创建游标所受到的限制是相同的。

创建和使用存贮过程

存贮过程是一个专业数据库编程人员必须掌握的概念，存贮过程可以在最大程序上发挥出 SQL 的潜能，该功能可以被如 C、FORTRAN、或 VISUAL BASIC 象调用或执行自己的函数一样地调用或执行。存贮过程应该是一组经过压缩处理的经常使用的一组命令（如交叉表的查询、更新和插入操作）。存贮过程允许程序员简单地将该过程作为一个函数来调用而不是重复地执行过程内部的语句。而且，存贮过程还有些附加的优点。

SyBase 工业有限公司是使用存贮过程的先驱，它早在 1980 年就在它的 SQL SERVER 中提供了存贮过程功能，这种过程是作为数据库的一部分被创建的，它与表、索引一样是存贮在数据库的内部的。Transcat-SQL 允许在过程调用中提供输入或输出的参数。这种机制可以让你写出通用的存贮过程并将变量传递给它。

使用存贮过程的一个最大的优点在于它可以在设计的阶段执行，当在一个网站中执行大批量的 SQL 语句时，你的应用程序会不停地不 SQL SERVER 进行通讯，这会使得网站的负荷迅速增大，在多用户环境下通讯将异常繁忙，你的服务器将变得越来越慢，而使用存贮过程可以在最大程序中减轻通讯负荷。

当存贮过程执行时，SQL 语句将在服务器中继续运行，一些数据信息将会返回给用户的电脑直至过程执行完毕。这会极大地提高性能并带来了附加的好处，存贮过程在第一次执行时在数据库经过了编译操作，编译的映象将存贮在服务器的过程中。因此，你不必在每一次执行它的时候都对它进行优化，这也使性能得到了提高。

使用 Transcat-SQL 来创建存贮过程的语法如下：

SYNTAX:

```
create procedure procedure_name  
    [( )@parameter_name  
        datatype [(length) | (precision [, scale])
```

```

        [= default][output]
    [, @parameter_name
        datatype [(length) | (precision [, scale])
        [= default][output]]...[]]]
    [with recompile]
as SQL_statements

```

运行存贮过程的 EXECUTE 命令的语法如下：

SYNTAX:

```

execute [@return_status = ]
    procedure_name
    [[@parameter_name =] value |
    [@parameter_name =] @variable [output]...]]
    [with recompile]

```

例 13.5

本例使用例 13.4 的内容来创建一个简单的过程。

INPUT:

```

1> create procedure Print_Artists_Name
2> as
3> declare @name char(30)
4> declare @homebase char(40)
5> declare @style char(20)
6> declare @artist_id int
7> create Artists_Cursor cursor
8> for select * from ARTISTS
9> open Artists_Cursor
10> fetch Artists_Cursor into @name, @homebase, @style, @artist_id
11> while (@@sqlstatus = 0)
12> begin
13>     print @name
14>     fetch Artists_Cursor into @name, @homebase, @style, @artist_id

```

```
15> end

16> close Artists_Cursor

17> deallocate cursor Artists_Cursor

18> go
```

你可以使用 EXECUTE 命令来执行 Print_Artists_Name 过程：

INPUT：

```
1> execute Print_Artists_Name

2> go
```

OUTPUT：

```
Soul Asylum

Maurice Ravel

Dave Matthews Band

Vince Gill

Oingo Boingo

Crowded House

Mary Chapin-Carpenter

Edward MacDowell
```

例 13.5 是一个很小的存贮过程；但是，一个存贮过程中可以包含许多条语句，也就是说你不必逐条地执行这些语句。

在存贮过程中使用参数

例 13.5 是重要的第一步，因为它给出的 CREATE PROCEDURE 语句的最简单的用法。但是，在看过它的语法以后你会发现 CREATE PROCEDURE 语句有着比例 13.5 更多的内容。存贮过程也可以接受参数并把它们输入到其中的 SQL 语句中。此外，数据可以通过输出参数从存贮过程中返回。

输入参数必须以 @ 提示符开始，而且这些参数必须是 Transcat-SQL 的合法数据类型。输出参数也必须以 @ 提示符开始，此外，OUTPUT 关键字必须紧跟着输出参数的名字。（当你在运行存贮过程时必须给出 OUTPUT 关键字）

例 13.6 给出了在存贮过程中使用输入参数的用法。

例 13.6

下面的存贮过程将选用所有发行媒体为 CD 的艺术家的名字：

```
1> create procedure Match_Names_To_Media @description char(30)
2> as
3>   select ARTISTS.name from ARTISTS, MEDIA, RECORDINGS
4>   where MEDIA.description = @description and
5>   MEDIA.media_type = RECORDINGS.media_type and
6>   RECORDINGS.artist_id = ARTISTS.artist_id
7> go
1> execute Match_Names_To_Media "CD"
2> go
```

运行该语句将会得到下边的结果集：

OUTPUT:

NAME

Soul Asylum

Maurice Ravel

Vince Gill

Crowded House

Mary Chapin-Carpenter

例 13.7:

本例中给出的输出参数的用法，在该例中将使用艺术家的 HOMEBASE 作为输入，过程会将艺术家的名字作为输出：

INPUT:

```
1> create procedure Match_Homebase_To_Name @homebase char(40), @name char(30) output
2> as
3>   select @name = name from ARTISTS where homebase = @homebase
4> go
1> declare @return_name char(30)
2> execute Match_Homebase_To_Name "Los Angeles", @return_name = @name output
3> print @name
4> go
```

OUTPUT:

Oingo Boingo

删除一个存贮过程

现在，你大概会猜到应该如何删除一个存贮过程了。如果你猜测是使用 DROP 命令，那你是绝对正确的。下边的语句将会从数据库中删除存贮过程。

SYNTAX:

```
drop procedure procedure_name
```

DROP 语句是经常使用的：当一个存贮过程被重新创建之前，旧的存贮过程以及它的名字必须被删除掉。根据我个人的经验，只有极少的存贮过程在创建之后是不需要修改的。有许多次，在语句中产生的错误会上传至过程，我们推荐你在创建存贮过程时使用 SQL 脚本文件来存贮你在过程中的所有语句，你可以在数据库服务器中使用这个脚本文件来得到你想要的结果或者是对存贮过程进行重新编译。该技术允许你使用通用的文本编辑器如 VI 或 WINDOWS 的记事本，但是，当你在运行脚本进行新的创建之前时，你一定要记得将原有的存贮过程及相关表先删除。如果你忘记了执行 DROP 命令，你将会收到一个错误信息。

在用 SQL 的脚本来创建数据库对象时经常会用到下边的语法：

SYNTAX:

```
if exists (select * from sysobjects where name = "procedure_name")
begin
    drop procedure procedure_name
end
go
create procedure procedure_name
as
.....
```

该命令会检查 SYSOBJECTS 表（这里边存贮着 SQL SERVER 的数据库对象信息）来查看该对象是否存在，如果存在，就在创建新对象之前先将它删除。在创建脚本文件并最后运行之前上面的工作将会花掉你大量的时间（因为可能有太多的潜在错误出现）。

存贮过程的嵌套

存贮过程也可以被嵌套调用以增强程序的模块化，一个存贮过程也可以被其它的存贮过程调用，它也可以调用其它的存贮过程。嵌套的存贮过程在很多情况下是一个非常好的办法。

- 嵌套存贮过程可以在函数级上将你的复合查询减到最小（如果在一行中需要运行 12 个查询，你或许可以将这 12 个查询精简为 3 个嵌套过程调用，当然，这要视情况而定）
- 嵌套存贮过程可以提高性能，查询优化将会更优化，许多的简明语句组可能会比大的语句组更有效。

当嵌套存贮过程时，所有的在存储过程内部创建的变量和数据库对象对于将要调用它的过程来说都是可见的。全体局部变量或临时对象（如临时表）将由最后创建这些元素的存贮过程来删除。

当准备一个大型的 SQL 脚本文件时，你可以要对表或数据库对象来运行它以检证问题。在调用它们之前你必须先创建一个嵌套的存贮过程来调用它。但是，主调过程可能会创建临时表或游标以在调用过程中使用它们。而被调用的过程则不知道这些将在脚本文件的稍后时创建临时表和游标。对于这个问题最容易的办法是在创建所有的存贮过程之前先创建这些临时对象，然后在存贮过程再次创建它们之前删除这些临时的对象（在脚本文件中）。你是不是对些有点迷糊了？例 13.8 将会帮助你明白这一过程。

例 13.8:

INPUT:

```
1> create procedure Example13_8b
2> as
3>     select * from #temp_table
4> go
1> create procedure Example13_8a
2> as
3>     create #temp_table (
4>         data char(20),
5>         numbers int)
```



```
6>      execute Example13_8b

7>      drop table #temp_table

8> go
```

分析：

正如你所看到的，过程 Example13_8b 使用了。但是#temp_table 并没有被创建（它是在过程 Example13_8a）中创建的，结果将会产生错误。事实上，Example13_8b 并没有被创建（因为并没有临时表#temp_table），而过程 Example13_8a 也不会被创建（因为没有过程 Example13_8b）。

下边的代码通过在创建第一个过程之前先创建#temp_table 来修正这一错误，其实#temp_table 在第二个过程创建之前被删除了。

INPUT：

```
1> create #temp_table (

2> data char(20),

3> numbers int)

4> go

1> create procedure Example13_8b

2> as

3>      select * from #temp_table

4> go

1> drop table #temp_table

2> go

1> create procedure Example13_8a

2> as

3>      create #temp_table (

4>          data char(20),

5>          numbers int)

6>      execute Example13_8b

7>      drop table #temp_table

8> go
```

设计和触发机制

触发机制从本质上来说是一种特殊类型的存储过程，它可以在下列的三种情况之一发生时自动运行。

- 更新
- 插入
- 删除

Transcat-SQL 创建触发机制的语法格式如下：

SYNTAX:

```
create trigger trigger_name  
  
on table_name  
  
for {insert, update, delete}  
  
as SQL_Statements
```

ORACLE 7 SQL 则使用下边的语法来创建触发机制：

SYNTAX:

```
CREATE [OR REPLACE] TRIGGER [schema.]trigger_name  
  
    {BEFORE | AFTER}  
  
    {DELETE | INSERT | UPDATE [OF column[, column]...]}  
  
[OR {DELETE | INSERT | UPDATE [OF column [, column] ...]}]...  
  
ON [schema.]table  
  
[[REFERENCING { OLD [AS] old [NEW [AS] new]  
  
    | NEW [AS] new [OLD [AS] old]}]  
  
FOR EACH ROW  
  
[WHEN (condition)] ]  
  
pl/sql statements...
```

触发机制对于强制执行引用完整性非常有用，我们在第 9 天《创建和维护表》中学习如何创建表时曾经提到过。强制执行引用完整性可以保证在多表交叉访问时数据的有效性，如果用户输入了下边的命令：

INPUT:

```
1> insert RECORDINGS values (12, "The Cross of Changes", 3, 1994)
```

```
2> go
```

分析：

这是一个有效的在表 RECORDINGS 表中插入新记录的命令。可是，对 ARTISTS 表进行一下快速的检查后你会发现并没有 ARTIST_ID=12 的记录，用户所拥有的在 RECORDINGS 表中插入记录的权利可以彻底地破坏你的数据引用完整性。

注：尽管有许多数据库系统都可以通过在创建表的时候设置约束来强制执行数据的引用完整性，但是触发机制却提供了更为灵活的解决方法。约束将会把系统的错误信息返回给用户，并且（你现在可能已经知道了）这些错误信息有时对你没有多大的帮助。而作为另外一种方法，触发机制可以打印出错误信息、调用其它的存储过程，如果有必要，它还可以修正错误信息。

触发机制与事务处理

触发机制所进行的活动是被默认为事务处理的一部分进行的，主要的事件次序如下：

- 1、默认地自动运行 BEGIN TRANSACTION 语句（对于表和触发机制而言）
- 2、当插入、更新、删除操作发生时。
- 3、触发机制被调用，其中的语句被自动执行。
- 4、由触发机制自动的完成事务处理的取消或确认操作。

例 13.9

本例解决了早些时候更新 RECORDINGS 表时所带来的问题。

INPUT：

```
1> create trigger check_artists
2> on RECORDINGS
3> for insert, update as
4>     if not exists (select * from ARTISTS, RECORDINGS
5>         where ARTISTS.artist_id = RECORDINGS.artist_id)
6>         begin
7>             print "Illegal Artist_ID!"
8>             rollback transaction
9>         end
```

```
10> go
```

分析：

类似的问题也可能在删除 RECORDINGS 表中的记录时出现，如果你只是将 RECORDINGS 表中将某个艺术家删除了。你可能也想同时删除 ARTIST 表中的艺术家记录。如果在触发机制激活之前记录已经被删除了，那么如何才能知道哪一个 ARTIST_ID 记录才是需要删除的记录呢？对于这个问题有两种解决的办法：

- 将 ARTIST 表中的所有不在 RECORDINGS 表中存在记录的艺术家删除。（见例 13.10a）
- 检查被删除过的逻辑表。Transcat-SQL 可以维护两个表：DELETED 和 INSERTED。这两个表中保存着对真实表的最近所做的改动，它与触发机制所创建的表有着相同的结构。因此，你可以从 DELETE 表中获得 ARTIST_ID 的内容并将它从 ARTIST 中删除。（见例 13.10b）

例 13.10a

INPUT：

```
1> create trigger delete_artists
2> on RECORDINGS
3> for delete as
4> begin
5>     delete from ARTISTS where artist_id not in
6>     (select artist_id from RECORDINGS)
7> end
8> go
```

例 13.10b

```
1> create trigger delete_artists
2> on RECORDINGS
3> for delete as
4> begin
5>     delete ARTISTS from ARTISTS, deleted
6>     where ARTIST.artist_id = deleted.artist_id
7> end
```

```
8> go
```

使用触发机制时的限制

- 当你在使用触发机制时你必须要知道它有如下的使用限制：
- 不能在临时表中创建触发机制。
- 触发机制必须在当前的表所在的数据库中创建。
- 不能在视图中创建触发机制。
- 当表被删除以后，所有与之相关的触发机制会被自动地删除。

触发机制的嵌套

触发机制也可以被嵌套，比如说你可以创建一个触发机制来执行删除动作。例如：如果触发机制自己删除了一个记录，数据库服务器可以据此激活另一个触发机制。结果将会不停地循环，直到表中的所有记录都被删除掉（或一些其他的触发条件被激活）。嵌套机制不是默认的，可是，环境中必须提供这个功能，对于这个主题你可以参考你的数据库文档来得到更多的内容。

在选择语句中使用更新和删除

这是复合使用更新和删除语句的命令。

INPUT:

```
SQL> UPPDATE EMPLOYEE_TBL  
  
SET LAST_NAME = 'SMITH'  
  
WHERE EXISTS (SELECT EMPLOYEE_ID  
  
FROM PAYROLL_TBL  
  
WHERE EMPLOYEE_ID = 2);
```

OUTPUT:

```
1 row updated.
```

分析:

EMPLOYEE 表中有一个雇员的名字是不正确的，我们只有当薪水表中出现的错误的 ID

时才会更新雇员表。

INPUT/OUTPUT:

```
SQL> UPDATE EMPLOYEE_TABLE  
  
      SET HOURLY_PAY = 'HOURLY_PAY * 1.1  
  
      WHERE EMPLOYEE_ID = (SELECT EMPLOYEE_ID  
  
      FROM PAYROLL_TBL  
  
      WHERE EMPLOYEE_ID = '222222222');
```

1 row updated.

分析:

我们将该雇员的小时报酬增加了 10%。

INPUT/OUTPUT:

```
SQL> DELETE FROM EMPLOYEE_TBL  
  
      WHERE EMPLOYEE_ID = (SELECT EMPLOYEE_ID  
  
      FROM PAYROLL_TBL  
  
      WHERE EMPLOYEE_ID = '222222222');
```

1 row deleted.

分析:

我们将雇员 ID 号为 222222222 的雇员删除了。

在执行前测试选择语句

如果你正在创建报表（比如你使用的是 SQL*PLUS）而且报表是比较大的，你也许会在运行之前先检查一个空格、列、标题。这会浪费你许多的时间。一个比较好的检查方法是在你的 SQL 语句中使用 `add where rownum < 3;`

SYNTAX:

```
SQL> select *  
  
      from employee_tbl  
  
      where rownum < 5;
```

分析:

这时你可以得到表中的前四行，用它你可以检查是否拼写、空格和看起是否合适。否

则，在你发现报表中的拼写错误或不正确的空格时你的报表已经返回了成百上千行。

技巧：实现你的顾客的真正的需要是你的一项重要的工作，它大概会占你所有工作中的一半。在特定的工作中拥有好的沟通手段和知识将会是你编程技能的有益补充。例如：如果你为一个小汽车代理商工作，它的经理想要知道在未来一段时间内他将会有多少小汽车进帐。你认为（只是你自己）：“对此进行计数将会很好，他问的是他有多少小汽车。”但是，其实经理人真正想知道的是他有多少种类型的汽车（小汽车、卡车），汽车的型号、生产的时间等等。是不是他的要求会浪费你许多的时间，或者说你给他的是不是他所需要的？

嵌入型 SQL

本书中的嵌入型 SQL 这一术语在用 SQL 来编写大型程序时经常用到。它的意思就是：可以将存贮过程嵌入到数据库之中并且它可以被应用程序来调用以处理一些任务。一些数据库系统提供了一整套的工具可以让你将 SQL 与程序设计语言结合在一起用以创建简单的屏幕和菜单对象。SQL 代码被嵌入到这些代码之中。

静态 SQL 与动态 SQL

静态 SQL 的意思就是指在程序中直接写入 SQL 代码。这些代码在运行的时候不能被更新。事实上，大多数静态 SQL 解释器需要将你的 SQL 语句在运行之前进行预编译处理。ORACLE 7 与 INFORMIX 都为他们的数据库系统开发了静态 SQL 包，这些经过预编译的产品可以在几种语言环境中使用，主要有以下几种语言：

- C
- PASCAL
- ADA
- COBLE
- FORTRAN

静态 SQL 的好处在于：

- 提高运行时的速度
- 经过了编译错误检查

它的缺点是：

- 灵活性差
- 需要更多的代码（因为查询不能在运行时进行变更）
- 它对于其它的数据库系统来说使用起来不方便（这一因素你必须要考虑到的）

如果你将这些代码打印出来的话，SQL 的语言将会出现在 C 语言的代码（或者是你所使用的那种语言），在进行预编译处理时字段要受到程序变量的限制。例 13.11 是一个简单的静态 SQL 的使用实例。

动态 SQL 是另外一种方法，它可以让程序员在运行时构建 SQL 语句并把这些语句提交给数据库引擎，引擎再将数据返回给程序变量，这也是在运行时实现的，这一主题已经在第 12 天进行过非常彻底的讨论了。

例 13.11

本例给出了静态 SQL 的 C 语言中的使用方法，注意到这里的语法并不是完全依照 ANSI 标准来实施的，静态 SQL 不同于任何商业化的产品，尽管它的语法与商业化的产品类似。

INPUT:

```
BOOL Print_Employee_Info (void)
```

```
{
```

```
int Age = 0;
```

```
char Name[41] = "\0";
```

```
char Address[81] = "\0";
```

```
/* Now Bind Each Field We Will Select To a Program Variable */
```

```
#SQL BIND(AGE, Age)
```

```
#SQL BIND(NAME, Name);
```

```
#SQL BIND(ADDRESS, Address);
```

```
/* The above statements "bind" fields from the database to variables from the program.
```

```
After we query the database, we will scroll the records returned
```

```
and then print them to the screen */
```

```
#SQL SELECT AGE, NAME, ADDRESS FROM EMPLOYEES;
```

```
#SQL FIRST_RECORD
```



```
if (Age == NULL)
{
    return FALSE;
}
while (Age != NULL)
{
    printf("AGE = %d\n", Age);
    printf("NAME = %s\n", Name);
    printf("ADDRESS = %s\n", Address);
    #SQL NEXT_RECORD
}
return TRUE;
}
```

分析：

当你输入你的代码并把它们保存到文件之中以后，这些代码通常要经过某种类型的预编译处理方可运行，预编译会将带有#SQL 的指令转换为真正在 C 代码指令，然后将它与你其它的 C 语言代码一同编译。

如果你从来没有看过或写过 C 语言的程序，那么不要对例 13.11 中的语法过份留心。（在早些时候，静态的 SQL 只是一些伪指令，请参阅在你的产品中的关于静态 SQL 的真正语法。）

使用 SQL 来编程

到目前为止，我们已经讨论了两种在编程中使用的 SQL。第一种方法来写查询和更新数据已经在本书的第 12 天中进行了详细的讨论。第二种特点可以在第 3 代或第 4 代编程语言中使用。显然，如果你想知道该语言和一般的数据库编程知识那么你可以使用第一种语言。我们已经讨论过了静态 SQL 与动态 SQL 相比的优点和缺点。在第 18 天《PL/SQL 简介》和第 19 天的《Transcat-SQL 简介》将会包括两种扩充的 SQL 来取代嵌入型 SQL 来完成在这一部分讨论的相同类型的工作。

总结

在通用的编程环境如 VISUAL BASIC，DELPHI 和 POWERBUILDER 中为数据库编程人员提供了许多的工具来对数据库进行查询和更新数据库的工作。但是，随着你对数据库的日益深入，你将会发现在今天讨论的主题中和使用这些工具的优点。不幸的是，有关于游标，触发机制和存贮过程等概念是在最近的数据库系统中提出的，它们的标准化程序还不高。但是，基本的使用的理论在几乎所有的数据库系统中都提供了。

临时表是一种存在于用户会话过程中的表，这种表是存在于一个特殊的数据库之中（在 SQL SERVER 中称之为 TEMPDB）。FETCH 语句用以从游标中获得指定的记录或使游标移向下一个记录，它经常将日期和时间作为其名称的唯一标识，临时表可以存贮查询的结果以供以后的查询使用。如果多用户在同时在同时创建和使用了临时表，由于在 TEMPDB 中有大量的活动从而会导致系统性能的下降。

存贮过程是一种可以将多个 SQL 语句结合在一起作为一个函数的数据库对象，存贮过程可以接受和返回参数值并且可以被其它的存贮过程调用，该过程是运行于数据库服务器中并且是经过编译的，使用存贮过程与直接使用 SQL 语句相比可以提高系统的性能。

内嵌型 SQL 可以将 SQL 代码用于实际编程中，内嵌型 SQL 可分为静态 SQL 与动态 SQL 两类，静态 SQL 不能是运行时进行修改，而动态 SQL 可以允许一定程序的修改。

问与答

问：如果我创建了一个临时表，是否其它的用户可以使用我的表？

答：不能，临时表只能由它的创建者使用。

问：为什么我必须关闭和释放游标？

答：内存仍为游标所占用，即使它的名字不存在了也依然是这样。

校练场

- 1、MICIRSOFT VISUAL C++可以让程序员直接调用 ODBC 的 API 函数，对不对？
- 2、ODBC 的 API 函数只能由 C 语言直接调用，对不对？
- 3、动态 SQL 需要进行预编译，对不对？
- 4、临时表中的#提示符是干什么用的？

- 5、在将游标从内存中关闭后必须做什么？
- 6、能不能是 SELECT 语句中使用触发机制？
- 7、如果你在表中创建了触发机制然后你把表删除了，那么触发机制还存在吗？

练习

- 1、创建一个示例数据库应用程序（在今天我们使用了音乐收藏数据库作为示例）并对应用程序进行合理的数据分组。
- 2、列出你想要在数据库中完成的查询。
- 3、列出你要在维护数据库中需要的各种规则。
- 4、为你在第一步创建的数据库逻辑给创建不同的数据库计划。
- 5、将第二步中的查询转变为存贮过程。
- 6、将第三步中的规则转变为触发机制。
- 7、将第 4 步与第 5 步结合起来，与第 6 步一起生成一个脚本，其中包括所有的与该数据库相关联的过程。
- 8、插入一些示例数据（这一步可以作为第 7 步生成脚本的一部分）
- 9、执行你所创建的这些过程并验证它的功能。

第 14 天：动态使用 SQL

目标

今天这一课的目的是把我们到现在为止所学习的付诸于应用。在今天的內容中我们将学习如何在实际编程中使用 SQL，我们主要讨论在 WINDOWS 下的编程环境，但是编程的原理也适用于其它的软件平台，在今天之中我们将学习：

- 不同的商业产品 ——Personal Oracle 7 、 开放数据库联接 (ODBC)、 InterBase ISQL、 MicroSoft Visual C++、 和 BORLAND DELPHI 中 SQL 的差别。
- 如何正确的为使用 SQL 来设置你的环境。
- 如何使用 ORACLE 7 ， MicroSoft Query 和 InterBase ISQL 来创建数据库。
- 如何在 VISUAL C++和 DELPHI 中使用 SQL。

快速入门

这一部分将主要介绍一些在 WINDOWS 操作系统下的商业产品以及它们是如何与 SQL 产生联系的。所涉及的基本原理如果在产品中没有进行特别的说明那么也可以在其它的软件平台中使用。

ODBC

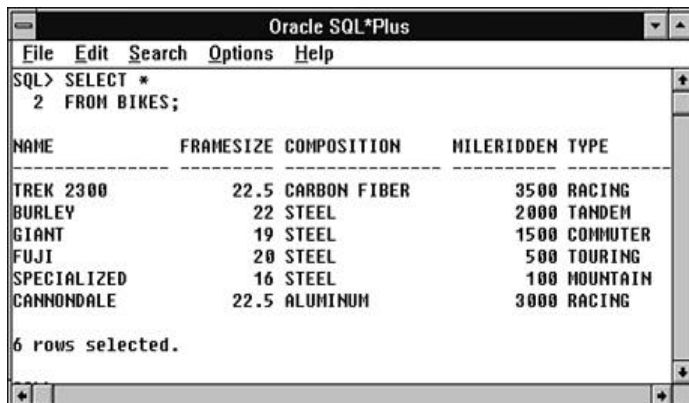
ODBC 是 WINDOWS 操作系统的一种底层技术，它可以让基本 WINDOWS 的程序通过驱动程序来访问数据库，与每种数据库系统所提供的用户界面不同，你可以用它来将你所要完成的工作完成的非常好，你可以通过你所选择的驱动来联接进行数据库，ODBC 的概念与 WINDOWS 的打印机驱动的概念非常相似 ——它可以让你写的程序与具体的打印机无关。与此相对应的是在 DOS 下你不得不自己来考虑和编写相关的打印机驱动程序。所以你可以将你更多的精力来投入到你的程序上，而不是写打印驱动。

ODBC 将这一思想应用到了数据库上，你可以在 WINDOWS 3.1， 3.11 和 95 的控制面板中找到它。在 WINDOWS NT 中它有着自己的程序组。

我们将在今天创建数据库的时候讨论更多的关于 ODBC 的细节。

Personal Oracle 7

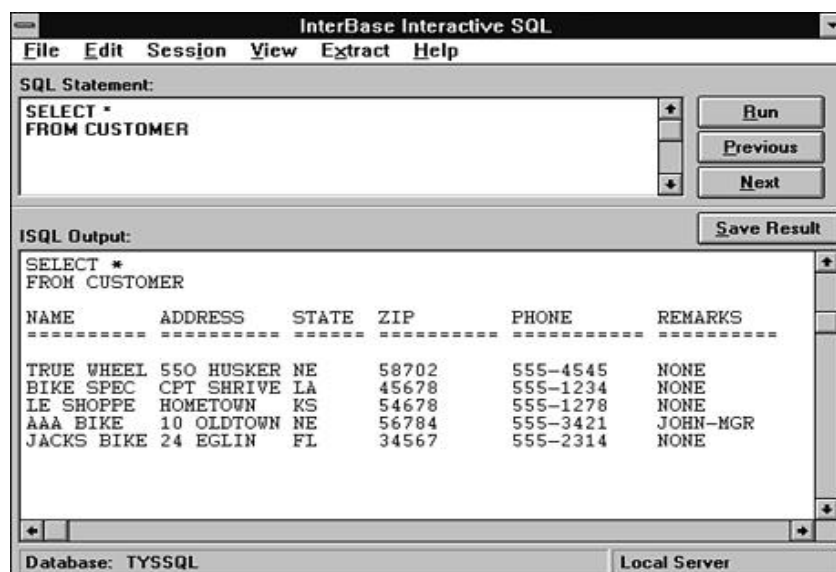
Personal Oracle 7 是近来在 PC 数据库市场上最流行的数据库系统，不必安装 Personal Oracle 7 的全部程序，我们在前几天中所使用的例子只是用了 ORACLE DATABASE MANGER 和 SQL*PLUS 3.3。SQL*PLUS 的如下图所示。



InterBase SQL (ISQL)

一些其它的例子所使用的工具是 Borland's ISQL。它在本质上与 ORACLE7 是相同的，只不过 ORACLE 7 是字符型界面而 ISQL 则更具有 WINDOWS 的风格。

在下图中给出了 ISQL 的界面，你可以在上边的编辑框中输入 SQL 语句，结果将会出现在下边的编辑框中，按向上和向下按钮则可以翻阅你在一次会话中的所有查询。



Visual C++

关于 Visual C++ 的书有几十本，在本书的例子中我们使用的是它的 1.52 版，我们所使用的过程可以应用于它的 32 位版的 C++ 2.0。在这里使用它是应用它有一个 ODBC 的简单界面，它不但具有编译联接到数据库的能力，而且如果你想使用其它的编译器的话，那么这里将会是一个非常好的出发点。

Visual C++ 提供了不少的工具，我们这里只使用其中的两个：编译器和原代码编辑器。

Delphi

我们最后要说的工具是 Borland's Delphi，它是许多新书中讨论的主题，它为不同的数据库提供了一个可以升级的界面。

我们使用它的两个程序：InterBase Server 和 Windows ISQL (WISQL)。

设置

在进行的足够的介绍以后我们来开始工作，在你安装完你的 SQL 引擎或 ODBC 兼容的编译器以后，在演员开始使用材料进行工作之前你必须指定舞台。无论是 Oracle 7 还是 InterBase，你都需要进行登录并为你自己设立一个帐号。这一过程在本质上来说是相同的，最困难的工作是为默认的口令分配硬拷贝和在线文档。这两种系统都有默认的系统管理员账号。见下图：



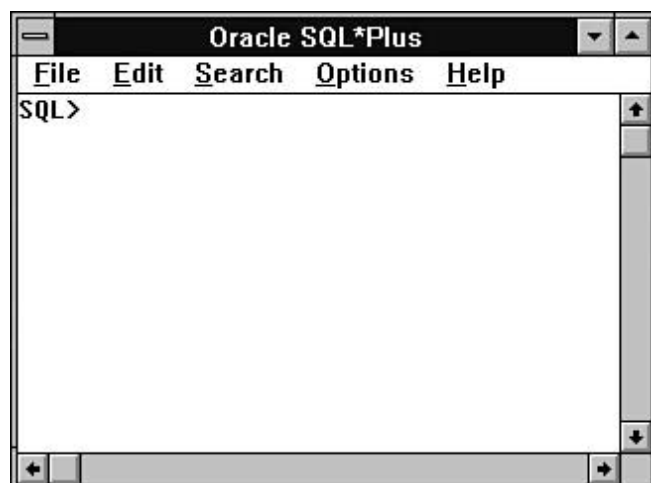
在登录和创建了用户账号以后，我们就可以创建数据库了。

创建数据库

从这一步开始，我们对 SQL 所学习的将开始得到回报，首先，你需要启动一个你想要使用的数据库，下图表明 Oracle 7 默认是停止状态的。



当你看到绿灯以后，你就可以启动如下图所示的 SQL*PLUS 了：



在这里你可以使用 CREATE 和 INSERT 命令来创建表和输入你想使用的数据了。另一种常用的方法是用脚本来创建表和输入数据，脚本通常是一个包含适当的 SQL 命令集合的文本文件，请看下边这个用于 Oracle 7 的脚本文件：

```
-----  
-- Script to build seed database for Personal Oracle  
-----  
  
-- NTES  
    Called from buildall.sql  
  
-- MODIFICATIONS
```

```
-- rs 12/04/94 - Comment, clean up, resize, for production

-----

startup nomount pfile=%rdbms71%\init.ora

-- Create database for Windows RDBMS

create database oracle

    controlfile reuse

    logfile '%oracle_home%\dbs\wdblog1.ora' size 400K reuse,

        '%oracle_home%\dbs\wdblog2.ora' size 400K reuse

    datafile '%oracle_home%\dbs\wdbsys.ora' size 10M reuse

    character set WE8ISO8859P1;
```

实际的 SQL 语法随着你所使用的数据库的不同而有一些差别，所以你应该看一下你的随机文档，并在你的 SQL 引擎中选择《文件》➔《打开》来装入脚本文件。

Borland's InterBase 使用相同的方法来装入数据，下边是从一个文件中摘出的插入数据的片断：

```
/*    Add countries.    */

INSERT INTO country (country, currency) VALUES ('USA',      'Dollar');
INSERT INTO country (country, currency) VALUES ('England',  'Pound');
INSERT INTO country (country, currency) VALUES ('Canada',   'CdnDlr');
INSERT INTO country (country, currency) VALUES ('Switzerland','SFranc');
INSERT INTO country (country, currency) VALUES ('Japan',    'Yen');
INSERT INTO country (country, currency) VALUES ('Italy',    'Lira');
INSERT INTO country (country, currency) VALUES ('France',   'FFranc');
INSERT INTO country (country, currency) VALUES ('Germany',  'D-Mark');
INSERT INTO country (country, currency) VALUES ('Australia', 'ADollar');
INSERT INTO country (country, currency) VALUES ('Hong Kong', 'HKDollar');
INSERT INTO country (country, currency) VALUES ('Netherlands','Guilder');
INSERT INTO country (country, currency) VALUES ('Belgium',   'BFranc');
INSERT INTO country (country, currency) VALUES ('Austria',   'Schilling');
INSERT INTO country (country, currency) VALUES ('Fiji',      'fdollar');
```

分析：

在本例中是向 COUNTRY 表中插入国家的名字和该国家所使用的货币类型（关于插入语句的使用请参阅第 8 天《维护数据》）。

这里边并没有什么魔术，程序员总可以找到一些方法来减少击键的次数，如果你是在家中进行尝试，那么请试着输入下边的表：

INPUT：

```
/* Table: CUSTOMER, Owner: PERKINS */

CREATE TABLE CUSTOMER (NAME CHAR(10),
    ADDRESS CHAR(10),
    STATE CHAR(2),
    ZIP CHAR(10),
    PHONE CHAR(11),
    REMARKS CHAR(10));
```

INPUT：

```
/* Table: ORDERS, Owner: PERKINS */

CREATE TABLE ORDERS (ORDEREDON DATE,
    NAME CHAR(10),
    PARTNUM INTEGER,
    QUANTITY INTEGER,
    REMARKS CHAR(10));
```

INPUT：

```
/* Table: PART, Owner: PERKINS */

CREATE TABLE PART (PARTNUM INTEGER,
    DESCRIPTION CHAR(20),
    PRICE NUMERIC(9, 2));
```

然后向这些表中输入下边的数据：

INPUT/OUTPUT：

```
SELECT * FROM CUSTOMER
```

NAME	ADDRESS	STATE	ZIP	PHONE	REMARKS
TRUE WHEEL	55O HUSKER	NE	58702	555-4545	NONE
BIKE SPEC	CPT SHRIVE	LA	45678	555-1234	NONE

LE SHOPPE	HOMETOWN	KS	54678	555-1278	NONE
AAA BIKE	10 OLDTOWN	NE	56784	555-3421	JOHN-MGR
JACKS BIKE	24 EGLIN	FL	34567	555-2314	NONE

INPUT/OUTPUT:

SELECT * FROM ORDERS

ORDEREDON	NAME		PARTNUM	QUANTITY	REMARKS
15-MAY-1996	TRUE	WHEEL	23	6	PAID
19-MAY-1996	TRUE	WHEEL	76	3	PAID
2-SEP-1996	TRUE	WHEEL	10	1	PAID
30-JUN-1996	TRUE	WHEEL	42	8	PAID
30-JUN-1996	BIKE	SPEC	54	10	PAID
30-MAY-1996	BIKE	SPEC	10	2	PAID
30-MAY-1996	BIKE	SPEC	23	8	PAID
17-JAN-1996	BIKE	SPEC	76	11	PAID
17-JAN-1996	LE	SHOPPE	76	5	PAID
1-JUN-1996	LE	SHOPPE	10	3	PAID
1-JUN-1996	AAA	BIKE	10	1	PAID
1-JUL-1996	AAA	BIKE	76	4	PAID
1-JUL-1996	AAA	BIKE	46	14	PAID
11-JUL-1996	JACKS	BIKE	76	14	PAID

INPUT/OUTPUT:

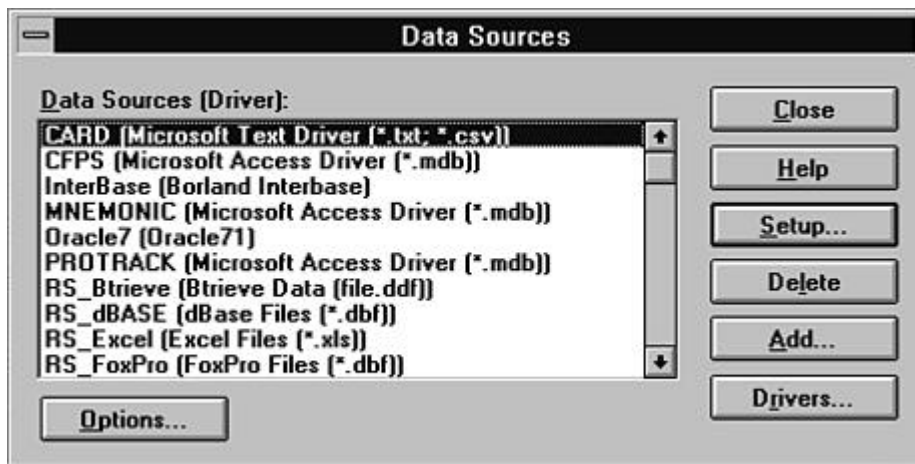
SELECT * FROM PART

PARTNUM	DESCRIPTION	PRICE
54	PEDALS	54.25
42	SEATS	24.50
46	TIRES	15.25
23	MOUNTAIN BIKE	350.45
76	ROAD BIKE	530.00
10	TANDEM	1200.00

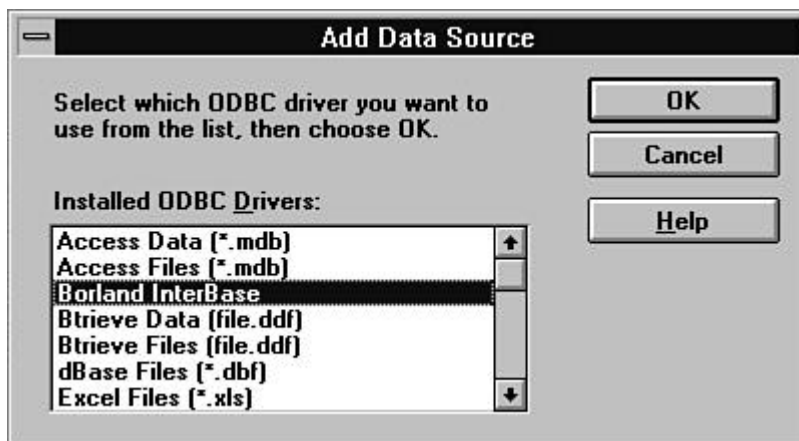
当你输入数据以后，下一步是创建一个 ODBC 联接，请打开控制面板（如果你使用的是 WINDOWS 95、WINDOWS 3.1 或 3.11）并双击 ODBC 图标。

注：有好几种风格的 SQL 引擎可以装放 ODBC，VISUAL C++，DELPHI 和 ORACLE 7 是在安装的时候将 ODBC 作为其一部分的。幸运的是 ODBC 已经变得和打印机驱动一样普遍了。

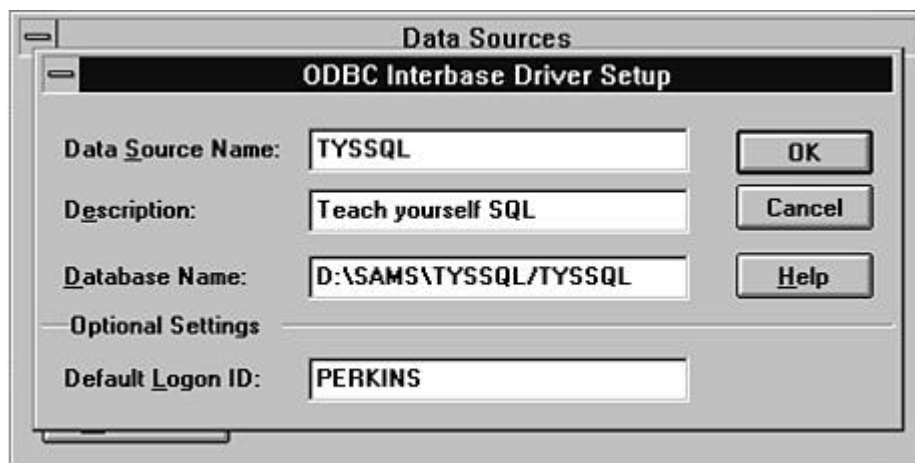
最初的 ODZBC 界面如下图所示：



这一屏幕给出的当前的 ODBC 链接，你想创建一个新的链接，假设你使用的是 InterBase 并且数据库的名字叫 TYSSQL（你是将你的工资让调 10% 的那个数据库），那那么请按 ADD 按钮并选择 InterBase 驱动，如下图所示：



从这里你可以转到设置屏幕，并进行如下图的填写：



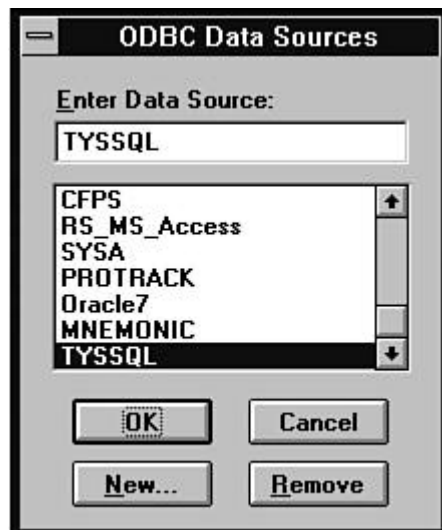
你可以使用你自己的名字或是一些比较容易输入的东西，这要依据你为你自己设置的账号而定。在这里有一个技巧，至少对于我是这样的，就是 InterBase 想用哪一个名字作为数据库。如果你是 PC 用户或小型的数据库后台可能还需要不得不使用路径名，路径句可以告诉数据库引擎在子网中的哪一台计算机中可以找到数据库。

使用 MS QUERY 来完成链接

现在我们已经建立了一个数据链接，我们需要使用一个更为方便的叫作 MS QUERY 的工具，该工具是与 VISUAL C++ 一同载入的，我们通过使用它来解决足够的数据库和代码问题以补偿多次编译带来的费用。QUERY 通常安装在一个独立的程序组中，你可以找到它的，它的外观如下图所示：



选择《FILE|NEW QUERY》。你的 TTSSQL 链接将不会出现，所以我们需要按《OTHER》按钮来调出 ODBC 数据源对话框，然后从中选择 TYSSQL，如下图：



可以小型数据库的用户会不习惯进行登录，可是在这里你必需输入密码才能通过这一屏幕。

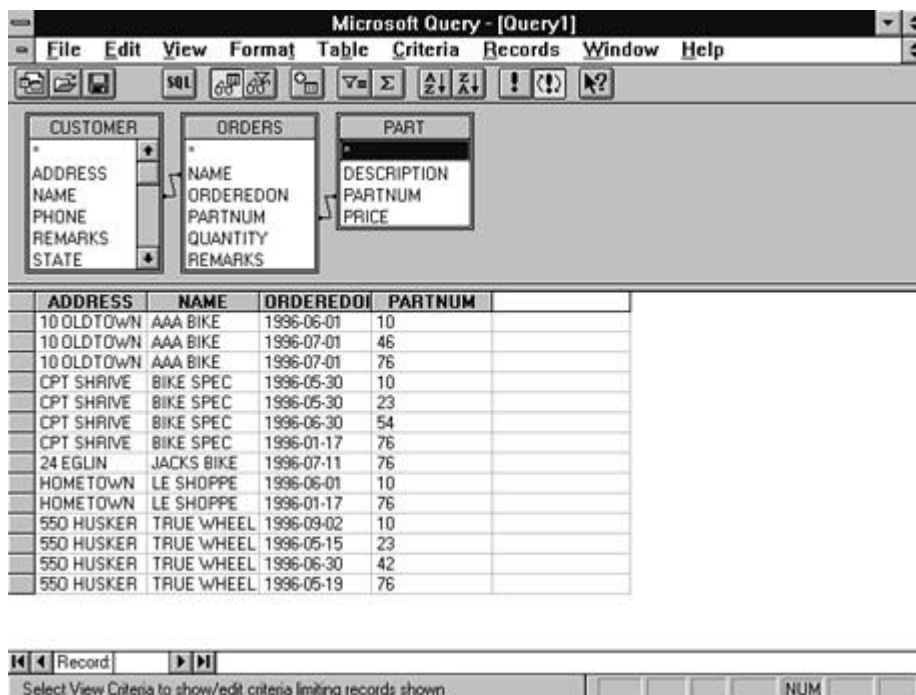
加入表的对话框如下图所示，这里的表是与你所建立的数据库相关联的，选择 PART、ORDERS 和 CUSTOMER 然后按关闭按钮。



该工具有两个功能，首先是对 ODBC 链接作检查，如果它可以工作就会在程序中工作，这一步可以让你确定问题是出在数据库方还是在程序一方。其次是生成查询并对其进行检查，在 SQL 中输入下边的语句并按 OK 按钮：

WHERE CUSTOMER.NAME = ORDERS.NAME AND PART.PARTNUM = ORDERS.PARTNUM

下图的结果是返回的结果：



你只不过完成了归并操作，但是你归并的字段已经以图形的方式返回了。（注意在 NAME 与 PARTNUM 之间的链接）

QUERY 是你在 WINDOWS 中工作的一个重要工具，它可以让你维护表和查询，你也可以使用它来创建表和维护数据。如果你使用 WINDOWS 的 ODBC 和 SQL 工作，你可以为你或你的电脑来购买它，它不像联网的 DOOM 那样有趣，但是它可以节省你的时间和

金钱，现在我们已经建立了 ODBC 链接，我们也可以在编程中使用它。

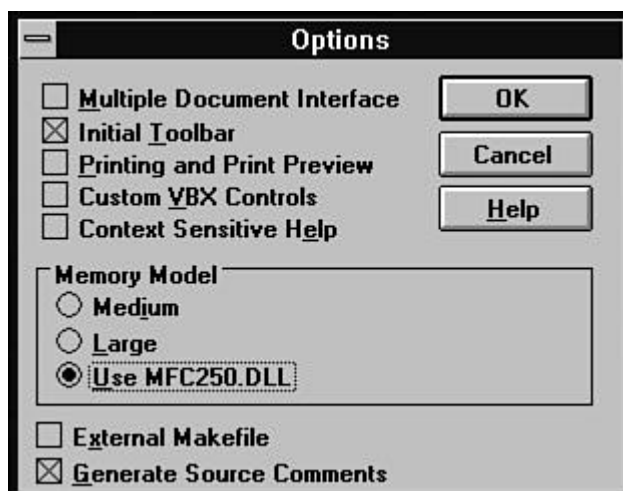
将 VISUAL C++ 与 SQL 结合使用

注：在附件 B 中有本例的源代码。

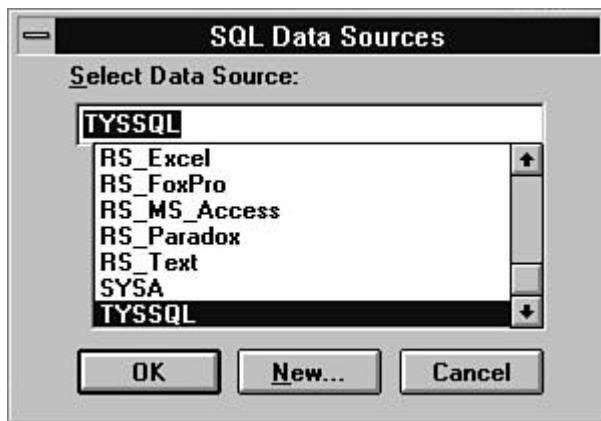
启动 Visual C++ 并调用 AppWizard，如下图所示，你的工程名字和子目录的名字可能是不一样的。



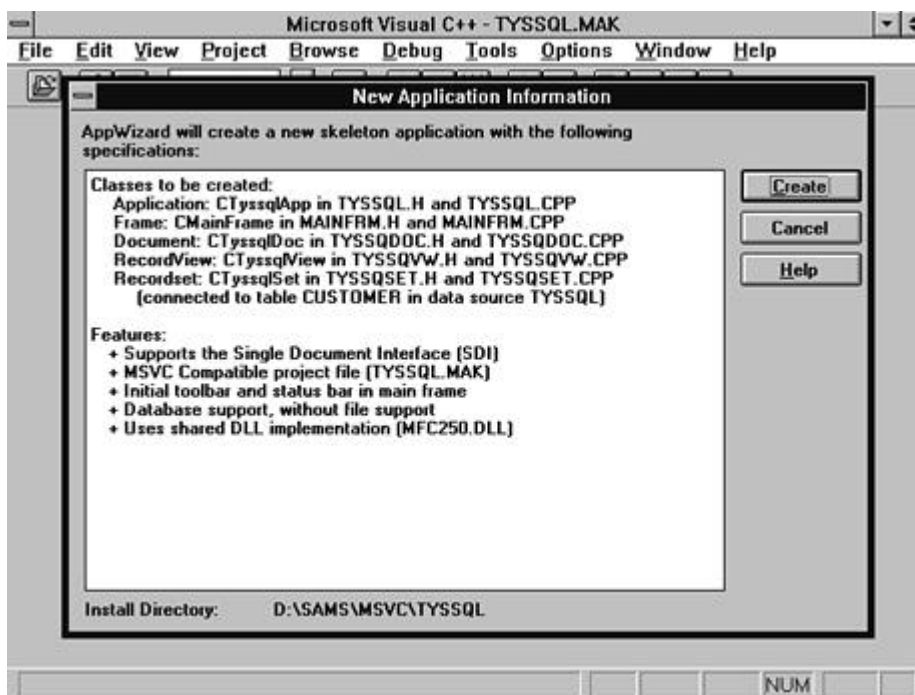
按 OPTION 按钮并填写下图：



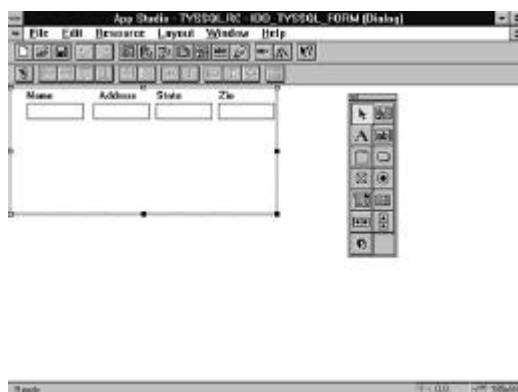
按 Data Source 按钮并对下图作出选择：



现在你可以从 TYSSQL 数据库中选择 CUSTOMER 表了，退回到 AppWizard 的基本屏幕并按两下 OK 按钮，再按一下 OK 后会显示 New Application Information，如下图所示：

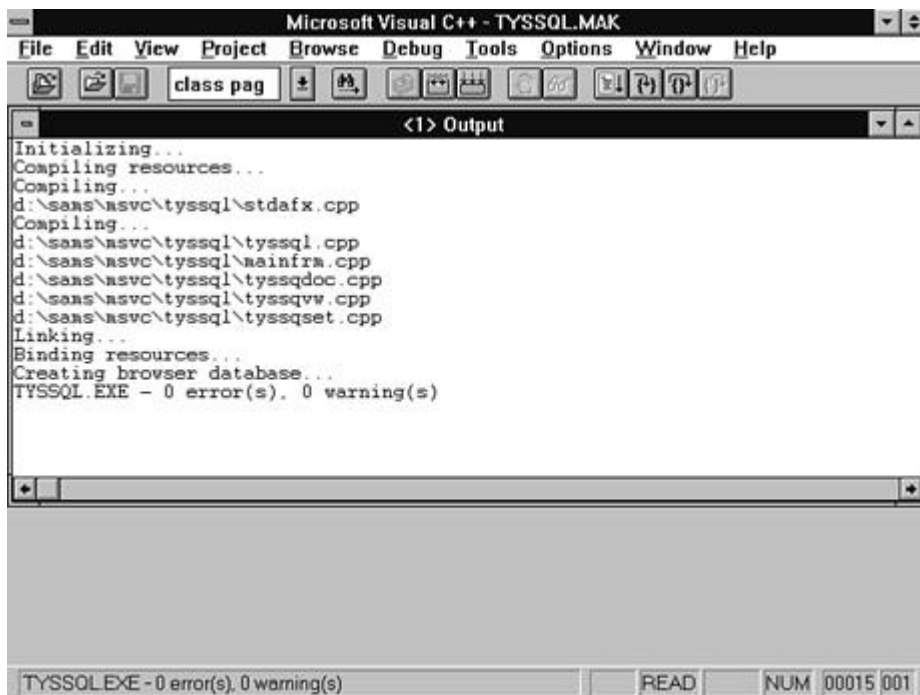


当程序生成以后，你需要使用原代码编辑器来设计你的主屏幕。选择 Tools | App Studio 来装入 App Studio，你需要设计的表单是很简单的，只要你能在翻阅时显示足够的表列就行，你可以参照下图来设计表单：



注：这个程序对于你所链接的表来说是很完美的，这也是使用 Microsoft Wizard 或 Borland Expert 的好处之一。

将你的工作保存，然后按 ALT+TAB 键回到编译器来编译这个程序，如果一切正常，你将会得到如下图所示的输出，如果你没有得到，那么你需要回头检查并再次尝试。



多么好的一个程序啊，要知道你现在还没有写一行代码，用箭头回退到数据库处，你可以发现数据的排序与它的输出是相同的。它们并没有按字母的次序排列（除非你已经使用了这个方法），你该如何让它们排序呢？

你所链接的数据库被封装在一个叫 CTyssqlset 的组中，它是 wizard 为你创建了，请看头文件：

```
// tyssqset.h : interface of the CTyssqlSet class
//
//
//
class CTyssqlSet : public CRecordset
{
    DECLARE_DYNAMIC(CTyssqlSet)
public:
    CTyssqlSet(CDatabase* pDatabase = NULL);
// Field/Param Data
```



```

//{{AFX_FIELD(CTyssqlSet, CRecordset)

CString    m_NAME;

CString    m_ADDRESS;

CString    m_STATE;

CString    m_ZIP;

CString    m_PHONE;

CString    m_REMARKS;

//}}AFX_FIELD

// Implementation

protected:

virtual CString GetDefaultConnect();// Default connection string

virtual CString GetDefaultSQL();// default SQL for Recordset

virtual void DoFieldExchange(CFieldExchange* pFX);// RFX support

};

```

分析:

要知道在表中所有列的成员都是可变的，请注意下边的 `GetDefaultConnect` 和 `GetDefaultSQL` 函数。这里是 `tyssqlset.cpp` 的 `implementations` 部分:

```

CString CTyssqlSet::GetDefaultConnect()

{

return ODBC;DSN=TYSSQL;";

}

CString CTyssqlSet::GetDefaultSQL()

{

return "CUSTOMER";

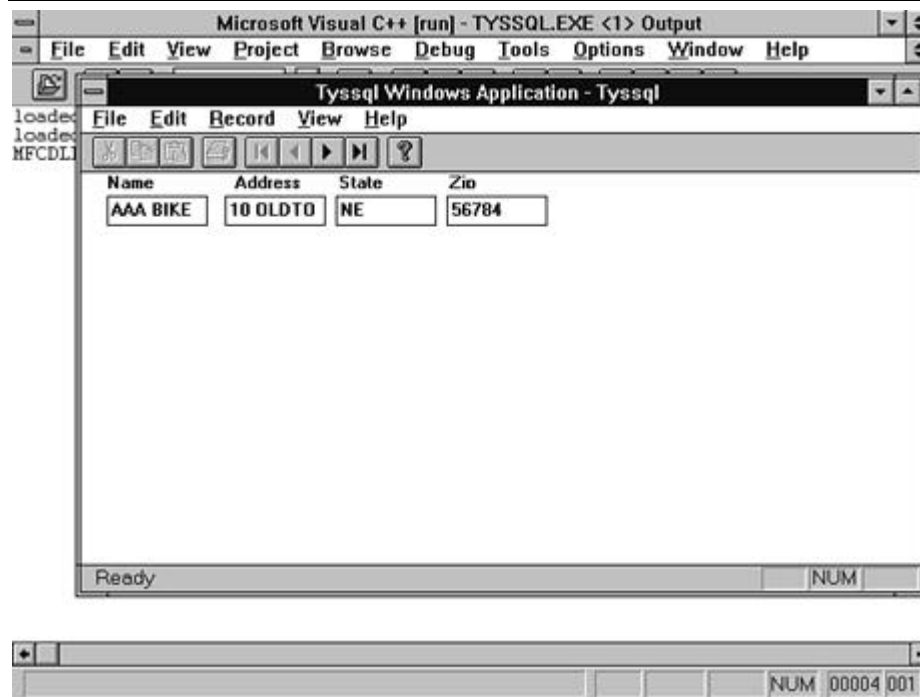
}

```

`GetDefaultConnect` 用以确认数据库链接，你不能改变它，但是，`GetDefaultSQL` 则可以让你作一些你所感兴趣的事，可以像下边这样改变它:

```
return "SELECT * FROM CUSTOMER ORDER BY NAME";
```

重新编译以后，你会发现结果已经排序了，如下图所示:



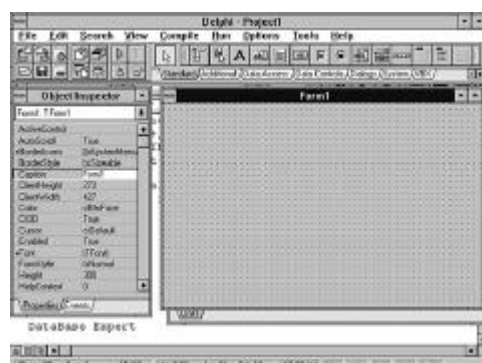
由于没有进行 Microsoft Foundation Class 的漫游，所以我们只能说你已经会操作 CRecordSet 和 CDatabase 对象了，归并与删除表，更新和插入记录，以及所有使用 SQL 进行的有趣的工作，对于将 SQL 与 VISUAL C++ 进行组合应用，你已经学得足够了，你可以进一步学习 CRecordSet 和 CDatabase（在线书籍已经成为 C++ 软件的一部分了）。ODBC API 和 AIPS 则由 ORACLE 和 SYBASE 提供。

将 DELPHI 与 SQL 结合使用

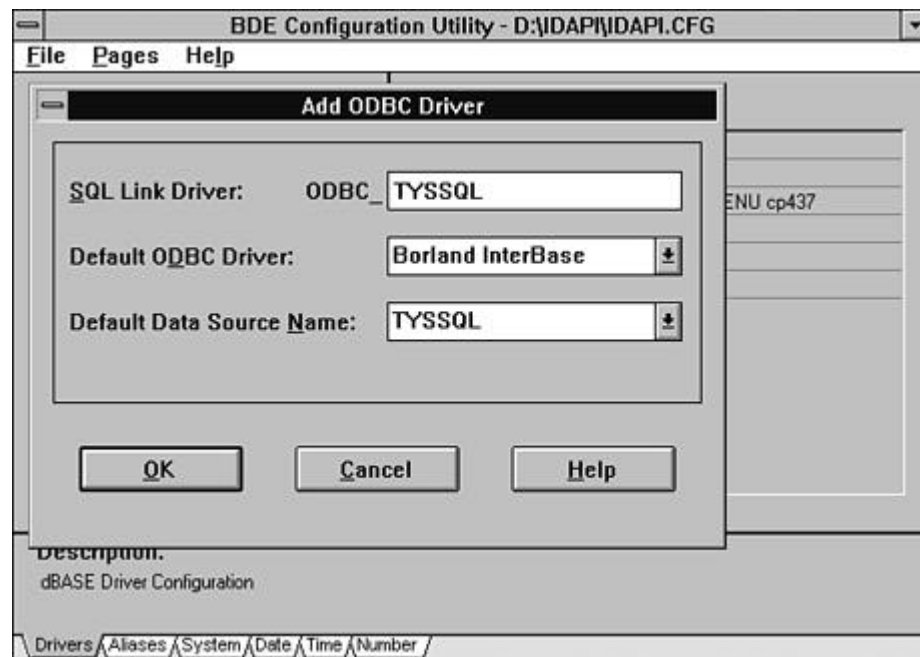
另外的一个 WINDOWS 下的重要的软件平台是 DELPHI，在 DELPHI 中 ORACLE 是作为一个画面被载入的，围绕的字符型的 SQL，在 C++ 的例子中你可以在线改写代码，在使用 DELPHI 时，你可以归并两个表而无需写哪怕是最简单的代码：

注：该程序的代码在附件 C 中提供。

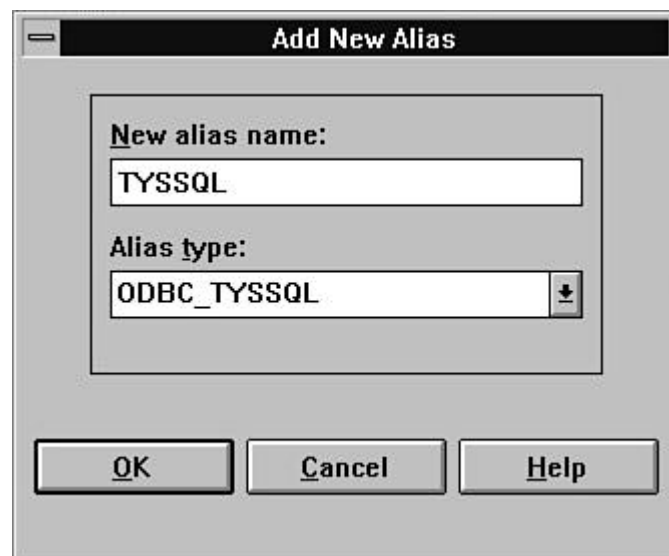
双击 DELPHI 图标来启动 DELPHI，程序的外观如下图所示：



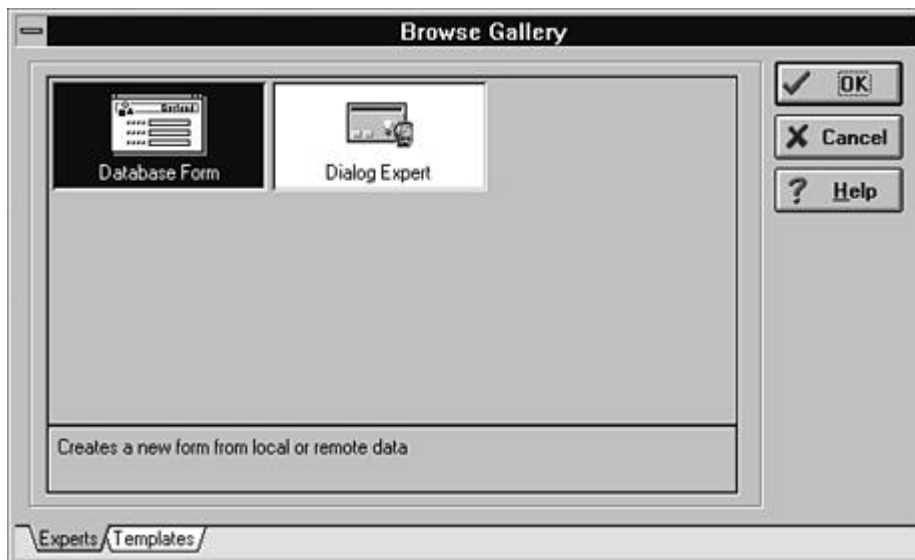
当你开始进行编程时你需要对 ODBC 联接进行注册，从工具菜单中选择 BDE（Borland Database 环境）并填写下图的对话框：



双击图下部的 ALIASES 标签，为 TYSSQL 分配别名，如下图：



选择文件|新表单进行下边的选择，从 EXPORT 表中选择 DATABASE FORM，如下图所示：



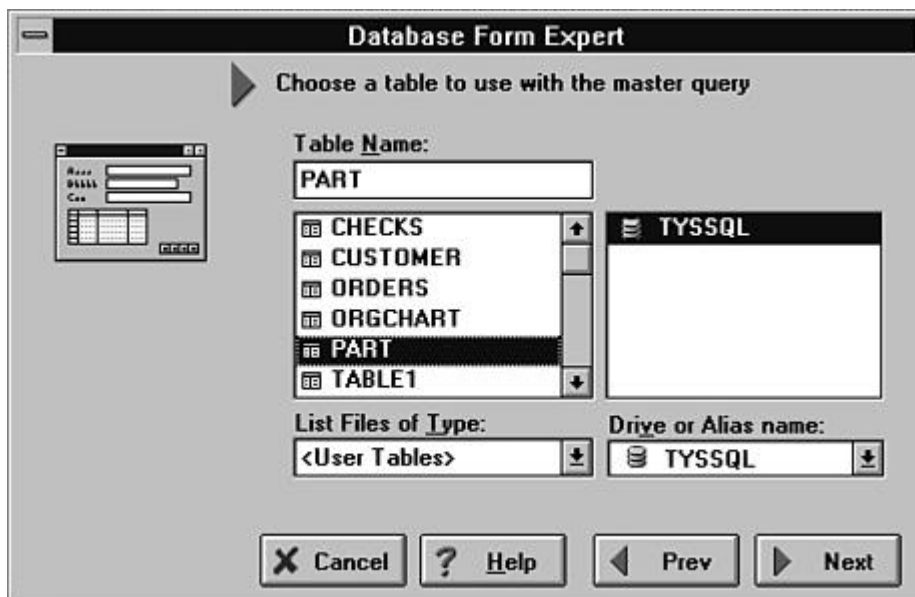
选择 master/detail form 和 TQuery objects，如下图所示：



现在选择你早些时候的 TYSSQL 数据源，如下图所示：



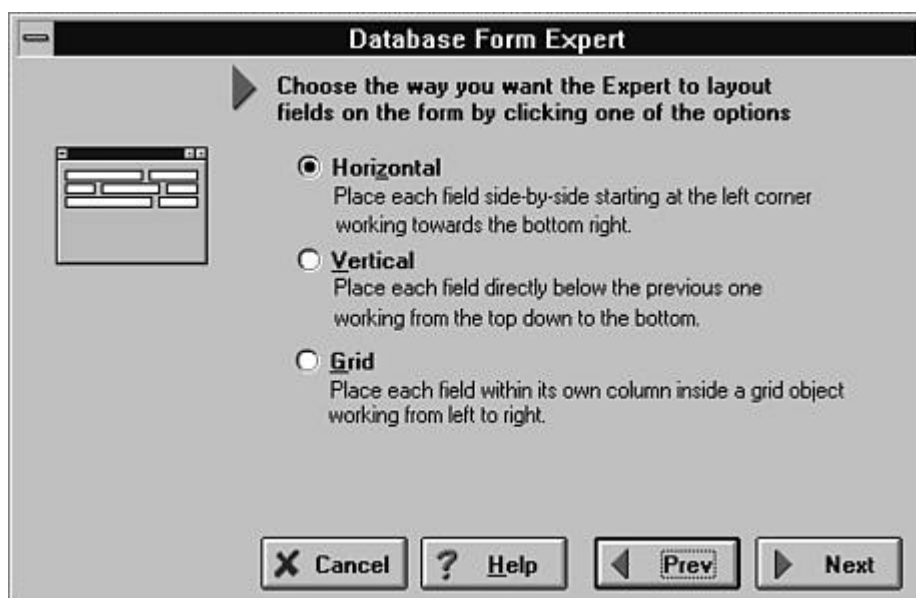
将 PART 表作为主表:



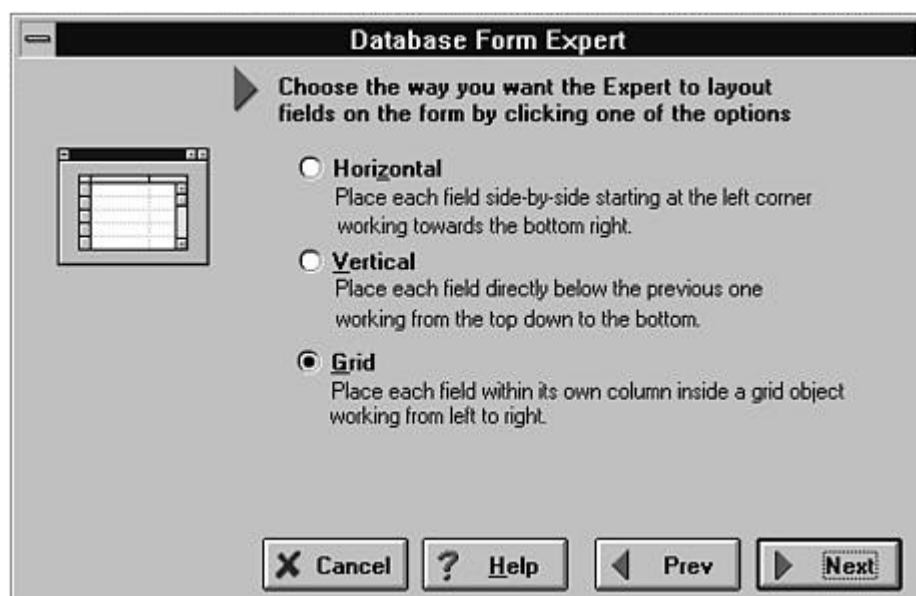
选择其中的所有字段，如下图:



选择水平显示模式，如下图：



然后选择 ORDERS 表，并选择其中的所有字段，显示模式选择为 GRID，如下边的三个图所示：



ORDEREDON	NAME	PARTNUM	QUANTITY	REMARKS
9/2/96	TRUE WHEEL	10	1	PAID
5/30/96	BIKE SPEC	10	2	PAID
6/1/96	LE SHOPPE	10	3	PAID
6/1/96	AAA BIKE	10	1	PAID

关闭工程并单击一个或全部表单上的查询对象，当你单击一个对象以后，对象浏览器将会显示它们的属性的不同：

```

Select
PART.PARTNUM,
PART.DESCRPTION,
PART.PRICE
From PART
  
```

这个查询已经试验成功了，但是我们并没有写一行代码。

总结

在今天我们学习了 SQL 在应用程序中的一般应用，每一天的学习都会让我们找到硬盘之中的一些新内容，最好的学习方法是你来学习如何去写查询，进行你能进行的查询试验。

问与答

问：为什么 ODBC 的 API 与 SYBASE 和 ORACLE 的 API 会不相同？

答：在从函数到函数级的调用上，ORACLE 与 SYBASE 的功能是非常类似的，但与此不同的是，许多数据库供应商都让它的数据库系统离开的通用的标准，ODBC 的 API 通常性更好——它并不针对任何特定的数据库，如果你需要特定的数据库并要求它提供高的性能，那你就需要使用数据库对你提供的 API 函数库。

问：在所有的可用的产品中，我如何知道该使用哪一个？

答：作为商业的环境，产品的选择通常从可管理性和可学习性两方面来考虑，管理性决定的产品的价格，而可学习性而可以让它更容易使用，在最好的程序环境中你可以解决问题非常的迅速而有效。

校练场

- 1、在 VISUAL C++ 中提供了几个 SQL 对象。
- 2、在 DELPHI 中提供了几个 SQL 对象。
- 3、什么是 ODBC。
- 4、DELPHI 可以做什么？

练习

- 1、在 C++ 的例子中如何对 STATE 字段进行正序或逆序的排序操作。
- 2、在向前一步，找到所有需要使用 SQL 的程序并使用它。

第二周回顾

- 在第一周我们主要把时间用在了介绍一个非常重要的主题——SELECT 语句。
- 在第二周则对 SQL 的进行了分门别类的详细介绍。
- 在第 8 天主要介绍的是数据维护语言，通过这些语句你可以修改数据库中的数据，三个最为通用的命令是 INSERT、DELETE 和 UPDATE。
- 第九天介绍了如何设计和创建一个数据库，主要命令有 CREATE DATABASE 和 CREATE TABLE，在表中可以创建任意数量的字段，它们中的每一个都有着由数据库供应商所定义的数据类型，而 DROP DATABASE 和 DROP TABLE 语句的作用则是删除数据库和表。
- 第十天讲了两种显示数据的方法：视图和索引。视图是用 SELECT 语句的输出所创建的一个虚拟表，索引则可以其于给定的一个或一组字段对数据库中的记录进行排序。
- 第 11 天的主要内容是事务管理，在这里你第一次体验到了使用数据库进行编程，事务用 BEGIN TRANSCAT 语句来开始，COMMIT TRANSCAT 语句用来确认事务而 ROLLBACK TRANSCAT 语句则用来取消事务。
- 第 12 天的主要内容集中在数据库安全上，尽管数据库安全的实现方法因数据库产品的不同而不同，但大多数是用 GRANT 和 REVOKE 语句来实现的，GRANT 语句用以给用户授权，REVOKE 则用以去除这些权限。
- 第 13 天主要内容是在应用程序开发环境中使用 SQL，静态 SQL 需要进行预编译在运行时是不会改变的，动态 SQL 则由于它的灵活性在近几年应用很广，用动态 SQL 进行编程的例子是 VISUAL C++和 DELPHI 开发工具。
- 第 14 天 SQL 的高级内容，游标可以在一个记录集中翻阅，存储过程是一种数据库对象，它可以在一行中执行多个 SQL 语句，它也可以接受和返回数值，触发机制是一种特殊类型的存储过程，它可以在向数据库中插入、删除和更新数据时自动运行。

第三周概貌

应用你对 SQL 的知识

欢迎来到第三周，到现在为止你已经学习了足够的 SQL 基本原理并且已经学习了 SQL 在实际生活中的应用。本周的内容将建立在基于头两周的内容之上。

- 第 15 天告诉你如何使用 SQL 流来提高 SQL 的性能。
- 第 16 天中告诉你如何使用关系数据库系统中的数据字典和系统目录而且会告诉你如何来获得有价值的信息。
- 第 17 天对数据字典的概念进行了扩充并讲解如何使用数据字典来用 SQL 语句生成 SQL 语句，你会学习到这项技术的优点并学习如何用它来提高你工作的效率。
- 第 18 天：主要是 Oracle 的 PS/SQL 或 Oracle 7 的过程语言，PL/SQL 是许多对标准 SQL 的扩展之一。
- 其它的扩展如 SYBASE 或 MICROSOFT SERVER 的 TRANSCAT SQL 它将在第 19 天提到。
- 第 20 天：回到 ORACLE 中学习 SQL*PLUS 它可以让你使用高级命令来与数据库进行会话，它也可以让你以一种让人满意的方法来格式化查询所生成的报表，你可以将它与 PL/SQL 协同工作。
- 第 21 天：则介绍了关系数据库用户常见的逻辑错误和错误信息，我们对这些错误提供了比较好的解决方法以及如何避免这些错误的技巧。

第 15 天：对 SQL 语句优化以提高其性能

对 SQL 语句的流化是在设计和调试数据库时影响基应用性能的重要部分。不管数据库规划的如何合理或数据设计考虑的如何健全，你都不会对你的查询返回数据的及时性感到满意，即便是错误也是如此。那么你的客户呢，如果你不遵循一些基本的指导方针，那么你的客户也不会感到满意，所以你的领导对你也不会满意。

目标

你已经知道了在关系数据库中的主要 SQL 组件并且知道了如何与数据库进行通信。现在是将你的知识应用到实际生活中以提高性能的时候了。今天的目标是告诉你一些提高流与 SQL 语句性能的方法。在今天中你将学习到以下内容：

- 明白 SQL 语句流的概念。
- 明白批装入和事务过程的区别以及它们对数据库性能的影响。
- 能够对查询的条件进行操作以加快数据获得。
- 熟悉一些影响数据库调试阶段和整个过程的性能底层元素。

这里有一个类似的比喻可以帮助你明白 SQL 语句流的含义：迅游运动员必须在尽可能短的时间之内完成项目，否则他就有可能被淘汰。那么这些游泳者必须具有足够的技术。并且发挥他的全部的身体的固定能力，以使得他们可以以鱼一样的速度来在水中运行。而且在他的每一次划水和呼吸时都必须如此，游泳者需要在每一时刻都保持他身体的流线型以减少水的阻力。

你的 SQL 查询也与此相似，你应该总是准确地知道你的目标是什么并尽可能地用最快的方法来实现这一目标。你要将大部分时间花在计划上，少部分时间会花在稍后对它的调整上。你的目标应该总是获得正确的数据和用尽可能少的时间。最终用户等待一个缓慢查询的感觉就像一个饥肠辘辘的人等一份久候不至的大餐一样。虽然你可以将大多数的查询用几种方法来输写，但对查询中元素的安排的不同导致的最终时间差异可能有几秒、几分钟、甚至是几个小时，SQL 语句的流化过程可以找到在你的查询中排列这些元素的最优结果。

除了流化你的 SQL 语句以外，你还应该考虑到一些其他的影响你的数据库性能的因素。

素，例如：在数据库中的并发事务控制，表的索引以及对数据库的深层调试等等。

注：在今天的例子中我们使用的是 Personal ORACLE 7，其工具可以在 ORACLE 7.3 关系数据库管理系统中找到。但今天所讨论的概念不局限于 ORACLE，它们也可以应用到其它的数据库管理系统中。

让你的 SQL 语句更易读

尽管实际上易读性不会影响 SQL 语句的性能，好的程序员会习惯于调用易读的代码，当你在 WHERE 子句中存在多个条件的时候尤为重要。任何人读到这个子句的时候都可以确切地知道表是否已经被正确地归并了，并且也可以确切地知道条件的次序。

试着读下边的语句：

```
SQL>SELECT  EMPLOYEE_TBL.EMPLOYEE_ID ,  EMPLOYEE_TBL.NAME ,
EMPLOYEE_PAY_TBL.SALARY, EMPLOYEE_PAY_TBL.HIRE_DATE

FROM EMPLOYEE_TBL, EMPLOYEE_PAY_TBL

WHERE  EMPLOYEE_TBL.EMPLOYEE_ID=EMPLOYEE_PAY_TBL.EMPLOYEE_ID
AND  EMPLOYEE_PAY_TBL.SALARY>30000  OR  (EMPLOYEE_PAY_TBL.SALARY
BETWEEN 25000 AND 30000 AND EMPLOYEE_PAY_TBL.HIRE_DATE < SYSDATE -
365);
```

下边是相同的查询的更易读的写法：

```
SQL> SELECT E.EMPLOYEE_ID, E.NAME, P.SALARY, P.HIRE_DATE
2  FROM EMPLOYEE_TBL E,
3      EMPLOYEE_PAY_TBL P
4  WHERE E.EMPLOYEE_ID = P.EMPLOYEE_ID
5      AND P.SALARY > 30000
6      OR (P.SALARY BETWEEN 25000 AND 30000
7      AND P.HIRE_DATE < SYSDATE - 365);
```

注：注意在上边的查询中使用了表的别名，在第 2 行语句中 EMPLOYEE_TBL 被赋予了简单的别名 E，在第 3 行中 EMPLOYEE_PAY 被赋予了别名 P。你可以看到在第 4、5、6 行中 E 和 P 已经取代了表的名字。别名需要输入的字符比输入全名时要少许多。更重要的是，使用别名以后查询变得更有组织和易读性，而使用表的命名则引出不必要的混乱。

这两个查询是一样的，但是第二个显然是更易读的，它非常具有结构性，那就是：查询的各个成份被回车和回车合理地分开了，你可以很容易地找到哪里是查询的 SELECT 部分（在 SELECT 子句中）和都有哪些表被访问（在 FROM 子句中），哪些是需要指定的条件（在 WHERE 子句中）。

全表扫描

当数据库服务为执行某一个 SQL 语句需要对表中的每一个记录进行检查时就会发生全表扫描。它通常在执行 SELECT 语句时发生，但有时也会在更新和删除记录时发生。全表扫描通常是因为在 WHERE 子句中使用了索引中没有的字段时发生。它就像对一本书一页一页地来看以找到所需内容一样，在大多数情况下，我们使用索引。

我们通常通过对经常在 WHERE 子句中使用的字段建立索引来避免全表扫描。索引所提供的找数据的方法与通过目录找书中的指定内容方法一样。使用索引可以提高数据的访问速度。

尽管程序员们并不赞成使用全表扫描，但是有时使用它也是适当的，例如：

- 你选择了一个表中的大多数行的时候
- 你在对表中的每一行记录进行更新的时候。
- 表非常小的时候。

对于头两种情况索引的效率是非常低的，因为数据库服务程序不得不频繁地读表和索引内容。也就是说索引只有在你所要找的数据只在表中所占比率很小的时候才会非常地有效。通常不会超过表中全部数据量的 10% 到 15%。

此外，最好在大型表中使用索引，当你设计表和索引的时候你要考虑表的大小，合适的索引应该是建立在对数据的熟悉上，知道那一系列数据是最经常引用的。如果想让索引工作得好你需要做一些试验。

注：当说到“大表”的时候，这里的大是相对而言的，一个表比某个表相比可以说很大，

而它与另一个表相比时却又很小，表的大小的概念是与数据库中其它表的大小、可用的磁盘空间、可用的磁盘的数量以及类似的因素相关的。很明显：2GB 的表很大而 16KB 的表是小的，如果一个数据库中表的平均大小是 100MB，那么一个 500MB 的表就是大的。

加入一个新的索引

你经常会发现一些 SQL 语句运行的时间长得不合情理，尽管其它的语句运行的性能看起来是可以接受的；例如：当数据的检索条件改变或表的结构改变以后。

当我们加入一个 WINDOWS 的应用前端时我们也会发现速度的下降，对于这种情况你首先要检查的是所用的目标表是否存在索引。然后大多数情况下我们会发现表是有索引的，但是在 WHERE 子句中所使用的新条件没有索引，看一下 SQL 语句中的 WHERE 子句，我们要问的是：“是否可以加入其它的索引？”。如果是在下列条件下，那么答案是肯定的：

- 最大的限制条件返回表的数据库小于表总数据量的 10%。
- 最大的限制条件在 SQL 语句中是经常使用的。
- 条件列的索引将会返回一个唯一的值。
- 列经常被 ORDER BY 或 GROUP BY 子句所引用。

也可以使用复合索引，复合索引是基于表中两个或更多列的索引，如果在 SQL 语句中经常将两列一起使用时这种索引会比单列索引更有效。如果在一起的索引列经常是分开使用的，特别是在其它的查询中，那么单列索引则是更合适的，所以你要经过试验来判断在你的数据库中使用哪一种索引会是更合适的。

在查询中各个元素的布局

在你的查询中最好的元素布局，尤其是在 WHERE 子句中，是根据解释器处理 SQL 语句的步骤和次序而定的。在条件中安排被索引过的列，这样的条件将会查找最少的记录。

你不一定非要在 WHERE 子句中使用已经被索引过的列，但是显然这样做会更有效。试着调整 SQL 语句以使它返回的记录数最少。在一个表中返回记录数最少的条件就是最大的限制条件。在通常的语句中，你应该把最大的条件限制语句放在 WHERE 子句的最后（ORACLE 查询优化会对 WHERE 子句从后向前读，所以它会最先处理我们放置的条件语句）

当优化器首先读到最大条件限制语句以后，它就将为以后的条件所提供的结果集缩减至最小了，下一个条件将不再搜索整个表，而是搜索经过最大条件限制过的子集。所以，数据的返回就会更快。在复杂查询中的多个查询、子查询、计算以及使用逻辑条件（AND、OR、NOT）中最大限制条件可能并不清晰。

技巧：请查看你的数据库文档资料，看一看解释器是如何处理你的 SQL 语句的。

下边的测试是我们对用两种不同的方法来查询相同的内容所耗用时间的差异，该例子使用 ORACLE 7 关系数据库系统。切记，在解释器中的优化是从后向前进行的。

在创建一个 SELECT 语句之前，对于每一个条件我们都选择了独立的行，下边是不同的条件给出的数值。

Condition	Distinct Values
calc_ytd = '-2109490.8'	13,000 +
dt_stmp = '01-SEP-96'	15
output_cd = '001'	13
activity_cd = 'IN'	10
status_cd = 'A'	4
function_cd = '060'	6

注：最大限制条件就是最显著的值。

下边的例子中在 WHERE 子句中使用了最大限制条件放在最前。

INPUT:

```
SQL> SET TIMING ON

2  SELECT COUNT(*)
3  FROM FACT_TABLE
4  WHERE CALC_YTD = '-2109490.8'
5     AND DT_STMP = '01-SEP-96'
6     AND OUTPUT_CD = '001'
7     AND ACTIVITY_CD = 'IN'
8     AND STATUS_CD = 'A'
9     AND FUNCTION_CD = '060';
```

OUTPUT:

```
COUNT(*)

8

1 row selected.

Elapsed: 00:00:15.37
```

INPUT/OUTPUT:


```
SQL> SET TIMING ON

2  SELECT COUNT(*)
3  FROM FACT_TABLE
4  WHERE FUNCTION_CD = '060'
5     AND STATUS_CD = 'A'
6     AND ACTIVITY_CD = 'IN'
7     AND OUTPUT_CD = '001'
8     AND DT_STMP = '01-SEP-96'
9     AND CALC_YTD = '-2109490.8';
```

OUTPUT:

```
      COUNT(*)
          8

1 row selected.

Elapsed:  00:00:01.80
```

分析:

注意所使用的时间的不同，只要简单地改变所给出的统计的次序，第二个查询比第一个查询快了 14 秒，那么可以设想一下当查询的结构设计不好的时候耗用几个小时的情形。

过程

如果所使用的查询有规律可循，那么你可以试着使用过程。过程可以调用很大的一组 SQL 的语句（参见第 13 天《高级 SQL 主题》）

过程是被数据库的引擎编译后运行的，与 SQL 语句不同，数据库引擎在执行过程的时候不需要进行优化。过程相对于独立的多个 SQL 语句，它对于用户来说更容易使用而对于数据库来说更为有效。

避免使用 OR

如果可能的话应该在查询尽量避免使用逻辑操作符 OR，OR 会不可避免的根据表的大小降低查询的速度。我们发现 IN 通常比 OR 要快。当然，优化器的文档中并不是这样说的。可是，下边的例子中使用了多个 OR。

INPUT:

```
SQL> SELECT *  
2 FROM FACT_TABLE  
3 WHERE STATUS_CD = 'A'  
4    OR STATUS_CD = 'B'  
5    OR STATUS_CD = 'C'  
6    OR STATUS_CD = 'D'  
7    OR STATUS_CD = 'E'  
8    OR STATUS_CD = 'F'  
9 ORDER BY STATUS_CD;
```

下边是使用子串和 IN 写成的相同的查询:

INPUT:

```
SQL> SELECT *  
2 FROM FACT_TABLE  
3 WHERE STATUS_CD IN ('A','B','C','D','E','F')  
4 ORDER BY STATUS_CD;
```

分析:

你可以自己进行一些类似的测试，尽管书中的代码是最为标准 and 直接的，但是你会发现你经常可以得到你自己的结论，特别是在性能方面。

下边是使用子串和 IN 的又一个例子，注意，第一个查询结合使用了 LIKE 和 OR。

INPUT:

```
SQL> SELECT *  
2 FROM FACT_TABLE  
3 WHERE PROD_CD LIKE 'AB%'  
4    OR PROD_CD LIKE 'AC%'  
5    OR PROD_CD LIKE 'BB%'  
6    OR PROD_CD LIKE 'BC%'  
7    OR PROD_CD LIKE 'CC%'  
8 ORDER BY PROD_CD;
```

```
SQL> SELECT *  
2 FROM FACT_TABLE  
3 WHERE SUBSTR(PROD_CD,1,2) IN ('AB','AC','BB','BC','CC')  
4 ORDER BY PROD_CD;
```

第二个例子不仅避免的使用 OR，而且也避免了使用 LIKE 与 OR 的联合操作，你可以试一下这个例子，你会看到对于你的数据它们在实际运行时性能上的不同。

OLAP 与 OLTP 的比较

当你在调试一个数据库的时候，你首先要决定的是它应该经常由谁来使用，在线的分析处理（OLAP）的数据库是一个对最终用户的查询进行统计和汇总的系统。在这种环境下返回的数据经常用与统计报告给决策管理过程提供帮助。而在线事务过程（OLTP）的数据库则是一个将主要的功能提供给为最终用户输入服务的环境的系统，包括用户日复一日的查询，OLTP 系统经常用在以日为基本单位在数据库中操作数据的使用场合，数据仓库与 DSS 可以从在线的事务处理数据库中得到它们所需的数据，有时也可以从其它的 OLAP 数据库中得到数据。

OLTP 的调试

一个事务处理数据库是一个精密的系统，它的访问任务是由繁重的按日而定的大量的事务与查询组成。但是，OLTP 通常不需要巨大的分类区域，至少它的需要没有 OLAP 的那么大，大多数 OLTP 的反应很迅速但是不会进行分类。

在事务处理数据库中最明显的例子就是 ROLLBACK 语句，需要撤消的内容的量与尺寸是与当前有多少用户在访问数据库相关的。与在每一个事务中进行的工作一样，最好的办法是在一个事务处理的环境中有多数 ROLLBACK 命令。

在事务处理环境中另一个涉及的问题是事务历史记录完整性，它在每一个事务结束后都会写出，LOGS 是为恢复的目的而存在的。所以，每一种 SQL 解释器都需要有一种方法来对 LOGS 进行备份以用于“恢复点”，SQL SERVER 使用的是 DUMP DEVICE，ORACLE 则使用一种被称为 ARCHIVELOG 模式的数据库，事务记录也会涉及到性能，因为对记录的备份是额外的负荷。

OLAP 的调试

如果要调试 OLAP 系统，例如数据仓库或决策支持的系统，与调试 OLTP 系统有着相当大的不同，一般说来，它需要较大的空间用以进行分类。

由于这种系统的目的是获得有用的用以决策的数据，所以你可以想象得到它有着相当多的复合查询并且通常要涉及到对数据的分组和排序，与事务处理数据库相比较而言，OLAP 系统是将较多的空间用于对数据的分类和排序而把较少的空间用于撤消区域的代表。

在 OLAP 系统中的大多数事务是作为批处理进程中的一部分存在的，代之以为用户输入提供大量的输入撤消区域，你会采用一个很大的撤消区域用以加载，这样可以实现离线工作以减少工作负荷。

批量载入与事务处理进程

对于 SQL 语句和数据库而言的一个重要的性能因素是处理的类型和它在数据库中所占用的空间，一种处理类型为 OLTP，它已经在今天的早些时候讨论过了，当我们谈到事务处理过程时，我们是指两种输入方法：用户输入和批量载入。

正常情况下用户的输入是由 INSERT、UPDATE、DELETE 语句组成的，这种类型的事务其性能通常是依据最终用户或客户而定的，最终用户通常使用前端应用程序如 PowerBuilder 来与数据库进行交互，所以他们很少能够看到 SQL 语句，然而，SQL 代码仍然通过他们所使用的前端应用程序产生了。

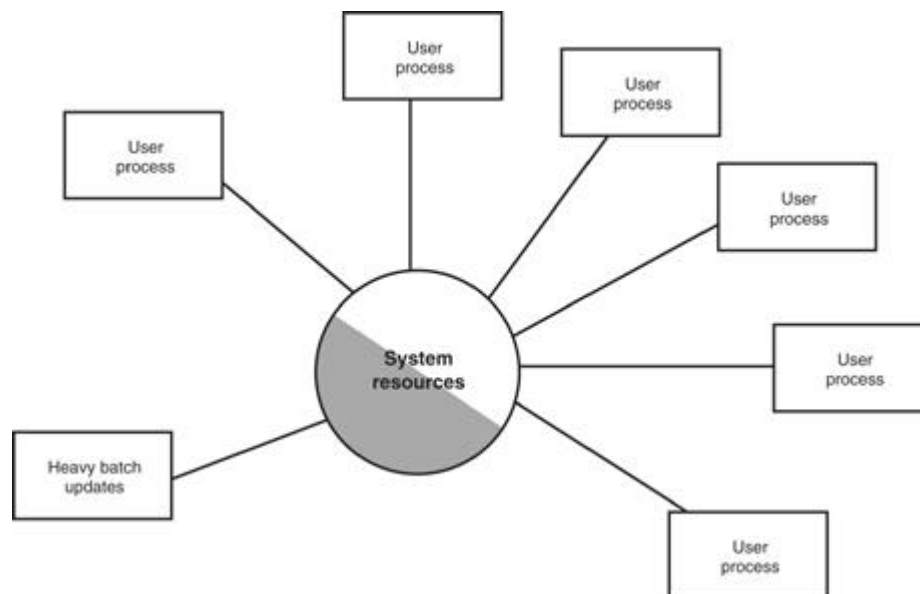
当我们优化数据库的性能时我们的重点应该在最终用户的事务上。毕竟，“没有用户”就是“没有数据库”，你也将会失业，你一定要力争让你的用户感到高兴，即使他们对数据库或系统的期望有些时候是不合情理的，对最终用户需要考虑的是最终用户输入的并发数量，在你的数据库中用户并发操作越多，数据库性能下降的可能性就越大。

什么是批量载入呢？批量载入就是在一次事务中完成对数据库所进行的任务，例如：如果你想把上一年的记录存入一个很大的历史表中，你需要在你的历史表中插入成千上万条的记录，你大概不想手工来完成这个工作，所以你会创建一个批任务或是一个脚本来自动完成这个工作（对于批量装入数据是有很多的技术可以使用的），批量载入由于它对系统资源和数据库资源占用而名声不好，这些数据库资源包括表的访问、系统目录的访问、数

数据库回退、和排序空间。系统资源则包括 CPU 和共享内存的占用。根据你所使用的操作系统和数据库服务，可能还有许多麻烦的工作。

最终用户事务与批量载入对于大多数数据库来说是成功的必备，但是当这两种类型的过程死锁时你的数据库系统会面临严重的性能考验。所以你应该知道它们之间的不同，如果可能最好对它们进行隔离。例如：当最终用户活动处于高峰时你不应该向数据库中装入大量的数据，数据库的响应会因为并发用户的增多而变慢，你要在最终用户访问最少的时候执行批量载入任务，许多公司都选择在夜间或是早上执行批量载入任务以避免与日间的进程产生冲突。

对于大量的批量载入你一定要安排好时间，要认真的避开数据库可能进行常规用户访问的时间，下表给出的当重载批处理任务进行时又有多个用户进行访问所引出资源引用冲突的情况。



正如你所看到的，许多进程在争用的系统的资源，重载的批处理任务已经打破了这种平等的情况，系统将不能为每一个用户平均地分配资源，批处理任务已经大量地占用了它们，这种情况只是资源争夺的开始，如果批处理任务进行下去，用户的进程可能会在最后被迫退出这副图，这在生意上是非常不利的，即使系统中只有一个用户。这种竞争也还是会出现。

使用批处理进程的另外一个问题是当另一个用户访问它所访问的表时可能会死锁，如果一个表被锁住了，用户将会被拒绝访问直到批处理进程解除对该表的锁定，这可能会是几个小时的时间，如果可能，批处理进程应该是系统处理最佳的时候发生，不要让用户与批处理进程进行竞赛，没有人会在这样的比赛中获胜。

删除索引以优化数据的载入

一种可以加快批量更新速度的方法是删除索引，设想一下如果历史表的记录有上千条，而且它还可能有一个甚至更多的索引。你认为索引会怎样，你通常会认为索引可以加快表的访问速度，但是在批量载入的时候，将索引删除的好处可能会更大。

当你通过索引向表中装放数据的时候，你通常会希望尽可能地使用索引，尤其是当你要更新的记录在表中所占的比率很高的时候。那么让我们来看一下这种方法，如果我们学习一本以后续指引的书的话，你也许会发现从头至尾地看这本书要比用索引来定位你的关注点更快。（如果你所要关注的东西所占的比重占全书中的比重比较小的时候索引是更有效的）

为了让数据占总表比重相当的批量载入达到最大效率，你可以使用下边的三步来使索引失去作用。

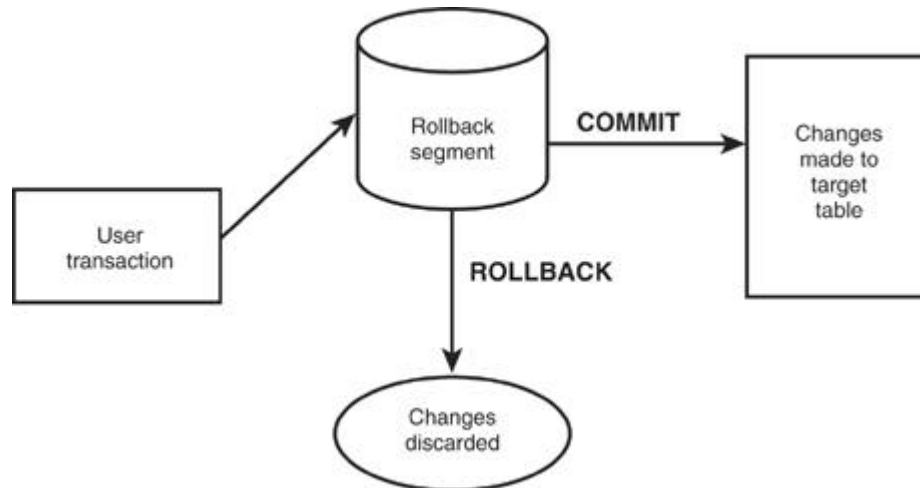
- 1、删除适当的索引。
- 2、装入或更新表中的数据。
- 3、重新生成表的索引。

经常使用 COMMIT 来让 DBA 走开

当你在执行一个批量事务时，你必须知道要多长时间执行一回 COMMIT 命令。就像你在第 11 天《事务控制》所学到的那样，COMMIT 可以结束一个事务。COMMIT 可以将事务中所作的任何改变写到实际的表中，但是在后台，它做的工作不只如此。在数据库中，有一个区域是用以存储全部的写到实际表中之前的事务数据的。ORACLE 将这一区域叫做 ROLLBACK 段，当你执行一个 COMMIT 命令以后，与你的 SQL 相关联的事务会将 ROLLBACK 段中的内容写到实际的表中，然后更新这一区域，ROLLBACK 段中的原有内容就被删除了，ROLLBACK 命令是另一种清除 ROLLBACK 段的命令方法，只是它不将所做的改动写到目标表中。

如你所料，如果你一直不执行 COMMIT 或 ROLLBACK 命令，那么事务就会一直保存在 ROLLBACK 段中。随之而来的是，如果你要装入的数据大小比 ROLLBACK 段的可用空间还要大，数据库将会终止并挂起所有的活动事务，不运行 COMMIT 命令是通用程序的一个缺陷，有规律地使用 COMMIT 命令将会例数据库系统输入的性能稳定。

对 ROLLBACK 的管理是数据库管理员（DBA）的一项复杂而重要的责任，因为事务对 ROLLBACK 段的影响是动态的，随后是像个别的 SQL 语句一样影响数据库的整体性能。所以当你批量载入大量数据的时候，要确保按一定的规律执行 COMMIT 命令，由你的数据库管理进行检查并告诉你应该很久执行一次 COMMIT 命令。（见下图）



你在上图中也看到了，当用户执行一个事务的时候，所做的改动是保存在 ROLLBACK 段中的。

在动态环境中重新生成表和索引

在大型数据库系统中动态数据库环境一词的意思就是状态在不断地改变，我们在批处理进程和日常事务处理过程中会经常使用这种改变，动态数据库通常很定于 OLTP 系统，但是也可以在 DSS 或数据仓库中引用它。这要视需载入的数据的量和频度而定。

结果是数据库中持续不断的大量数据的改变，从而造成大量的碎片，如果管理不当这些碎片就很容易失去控制，ORACLE 在表最初生成时为它分配了一个长度，当数据载入并填充完初始长度以后，初创建的表会得到下一个分配的长度。

表和索引的大小的数据库管理员的工作，而且它对 SQL 语句性能的影响是很大的，首先要进行正确的管理，所分配的空间应该足够表在一天中所增加的尺寸，同时也应该制定一个计划以按一定的规律对数据库进行碎片的清理工作，如果可能最好这成为每周的例行工作，清除关系型数据库中的表和索引的碎片在基本概念上的非常麻烦的。

- 1、对表和索引进行完善的备份。
- 2、删除表和索引。
- 3、用新的分配空间来重新生成表和索引。

- 4、将数据恢复到新建的表中。
- 5、如有必须，重新生成索引。
- 6、对该表重新分派用户的规则和权限。
- 7、直到你已经确认了新生成的表是完全正常的，否则请保留备份，如果你选择了放弃对原始表的备份，你要在新表的数据完全恢复后马上做一个备份。

警告：当你还没有确认新表已经完全正常之前千万不要丢弃原有的备份。

下边给出了一个 ORACLE 数据库中邮件清单表的实际的真实例子。

INPUT:

```
CREATE TABLE MAILING_TBL_BKUP AS
SELECT * FROM MAILING_TBL;
```

OUTPUT:

Table Created.

INPUT:

```
drop table mailing_tbl;
```

OUTPUT:

Table Dropped.

INPUT:

```
CREATE TABLE MAILING_TBL
( INDIVIDUAL_ID          VARCHAR2(12)          NOT NULL,
  INDIVIDUAL_NAME        VARCHAR2(30)          NOT NULL,
  ADDRESS                 VARCHAR(40)           NOT NULL,
  CITY                   VARCHAR(25)            NOT NULL,
  STATE                   VARCHAR(2)             NOT NULL,
  ZIP_CODE                VARCHAR(9)            NOT NULL,
) TABLESPACE TABLESPACE_NAME
STORAGE ( INITIAL         NEW_SIZE,
          NEXT             NEW_SIZE);
```

OUTPUT:

Table created.

INPUT:

```
INSERT INTO MAILING_TBL

select * from mailing_tbl_bkup;

93,451 rows inserted.

CREATE INDEX MAILING_IDX ON MAILING TABLE

( INDIVIDUAL_ID )

TABLESPACE TABLESPACE_NAME

STORAGE ( INITIAL          NEW_SIZE,

          NEXT             NEW_SIZE );
```

OUTPUT:

Index Created.

INPUT:

```
grant select on mailing_tbl to public;
```

OUTPUT:

Grant Succeeded.

INPUT:

```
drop table mailing_tbl_bkup;
```

OUTPUT:

Table Dropped.

分析:

重新生成表和索引可以让你的存储空间达到最优化，这将会提高整体的性能。切记，只有当你确认新生成的表正确无误以后才可以删除原有表的备份，而且你也应该知道你用另一种方法达到了相同的结果，请检查数据库文档看一下它的可以选项。

数据库的调整

调整数据库就是将数据库的服务优化的过程。如果你刚开始使用 SQL，那么除非你是数据库管理员或是刚刚接触关系型数据库的数据库管理员，否则你不公直接面对数据库的调整。不管你是不是数据库管理员，也不管理你会不会使用 SQL 来在应用程序中编程，如果想让所有的部分协同工作，通常的调整技巧有以下几点。

将数据库所需的全部大小最小化。

在设计时预留出增长的空间很好，但是不要对此过于迷恋，不要将你以后数据库的增长空间与资源强行捆绑。

对用户进程进行可用时间片试验。

这样你可以为不同的用户分配不同的可控时间。

优化你所使用的应用的网络包的尺寸。

要通过网络传送大量的数据，应该设定网络的最大包尺寸，你参考你的数据库和网络文档来查看更多的内容。

将大表分配于不同的磁盘上。

如果并发用户访问位于多个磁盘上的大表，等待系统资源的机会就会减少。

将数据库的排序区，系统目录区，撤消区分布于不同的磁盘上。见下图

这些区域是用户访问最多的区域，把它放在多个硬盘上你可以最有效地利用资源。

增加 CPU

这样系统管理及数据库的性能将会得到极大的提高，增加 CPU 可以提高数据处理的速度是显而易见的，如果你在机器上将有多 CPU，那么你将有机会行使并行处理策略，关于并行处理请参考你的数据库相关文档中的更详细的信息——如果它在你的环境中是可用的话。

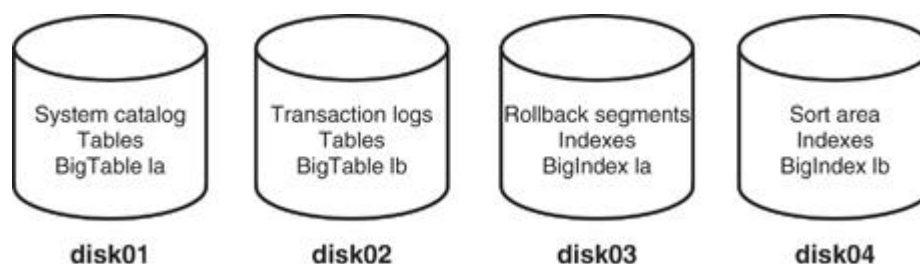
增加内存

一般说来，这样会更好。

将表和索引存贮在不同的硬盘上。

只要有可能，你就应该将索引和与它相关的表存储在不同的硬盘上。这种安排可以在读表的同时访问与该表相关的索引，存储在多个硬盘上的对象的实际能力取决于有多少块硬盘连接于控制器上。

下图给出的如何隔离数据库各个主要区域的简单例子：



在上图中使用了四个设备，从 DISK01 到 DISK04。这样做的目的是保证每个盘都有较

高的访问率。

DISK01-系统目录中存储着关于表、索引、用户、统计信息、数据文件的名字、大小、增加长度以及其它的相关数据，它在事务中被访问的次数相当多。

DISK02-每天的每一次对表的内容的改变（插入、更新、删除）都会更新事务的历史记录信息，事物记录是在线事务的一项重要功能，但它在只读型的数据库如 DSS 或数据仓库中的意义不大。

DISK03-ROLLBACK 段对于事务处理环境的意义也很大，可是，如果很少进行事务活动的话，ROLLBACK 段就不会被经常使用。

DISK04-这里是数据库排序区，也就是说它是 SQL 语句过程和排序数据（如 GROUP BY 或 ORDER BY 子句）的临时存放区域。DSS 和数据仓库是排序区域的典型代表。但是，在事务处理环境中也考虑使用排序区域。

技巧：你也应该注意到数据库的表和索引是如何入到每一个磁盘上的，应该尽可能地使表和索引分开。

注意在上图中表和索引是被安排在不同的设备上的。你也可以看看“大表”和索引在不同设备上的交互访问，这种技术将大表分为可以访问的小段，交互访问表和索引是一种控制碎片的办法，在这种情况下可以在参考相关索引的同时进行读表，这就会提高数据的整体访问速度。

这个例子的确是非常简单的，根据你所使用的数据库的函数、大小和相互关系，你可以找到类似的方法对系统进行优化以提高工作性能，如果钱不是问题那么完美的方法是将每一个数据库实体都放在一个硬盘上，包括大表和索引。

注：DBA 与系统管理员应该协同工作以平衡数据库的使用空间和优化服务器的可用存储空间。

对数据库的调整要依据你所使用的数据库系统而定，显而易见，优化查询要比调整数据库的作用更明显，也就是说当你还没有将 SQL 应用本身调整好时你不要期望在对数据库的调整上得到太多的回报，以调整数据库为专职的人员会专攻一种数据库产品以学习尽可能多的关于它的特点和特性。尽管数据库的调整经常被认为是一项痛苦的工作，但是它可以真正了解它的人对它更充分地利用。

性能的障碍

我们已经说不许许多的对一个数据库性能产生影响的可能方面。最具有代表性的瓶颈涉及到系统级的维护、数据库的维护和对 SQL 语句的管理。

这里总结了常见的大多数对系统性能和数据库响应时间有影响的因素。

没有很好地利用服务器上的可用设备——电脑可能基于某种原因已经购买了多个磁盘，如果你没有将一些数据库的重要组件分布在多个磁盘上的话性能就会受到影响。要让系统的能力和数据库服务的能力尽可能地发挥。

没有经常性的执行 COMMIT 命令——懒于使用 COMMIT 命令和 ROLLBACK 命令的结果会导致你的服务器出现瓶颈。

允许批量载入于日常进程同时工作，当数据库还存在其它用户时运行批量载入会对每个人都产生影响，批处理进行会由于成为最终用户和系统资源之间的瓶颈。

在创建 SQL 语句时不认真——不负责任地创建复合语句会比标准语句耗用更多的系统时间。

技巧：你可以用不同的来优化你的 SQL 语句的结构，具体的步骤依据 SQL 服务器处理你的过程的步骤而定。

对于有索引的表运行批处理任务时——你可以会需要几天几夜才能完成一个批处理的任任务。而一般的批量载入任务也许只要几个小时！索引会对批量载入任务访问表中更多的行造成障碍。

可分配的存储空间中有更多的并发用户，随着当前数据库用户和系统用户的增多，你可以会因为过程的共享而需要更多的内存。

创建索引的列只存在不多的可选数值——对如性别这样的列创建索引并不会非常有效，所以，你所创建的索引返回的行数应该占表的并行数比较低。

对小表创建了索引——这时使用全表扫描可能会更有效。

没有有效地管理系统的资源——对系统资源的管理不善如数据库初始化、表的创建、不受控制的碎片、和不正确的系统维护会浪费大量的数据空间。

表和索引的大小不合适——在大型数据库中对表和索引的大小估计错误片如果护理不当会带来严重的碎片问题，并会由于而引发更严重的问题。

内置的调整工具

在确认需要进行调整时要征询 DBA 和你的数据库供应商以得知哪些工具对你可用。你可以使用调整工具来确定你的数据库在数据访问上的不足之处，此外，这些工具对于特定的 SQL 语句提出更改建议以提高系统的性能。

ORACLE 有两个流行的工具用以管理 SQL 语句的性能，这两个工具是 *EXPLAIN PLAN* 和 *TKPROF*。*EXPLAIN PLAN* 可以在 SQL 语句执行时检查它的访问路径，*TKPROF* 可以给出每个 SQL 语句在执行时所需的时间，ORACLE 公司也提供了其它的工具用以对 SQL 语句和数据库进行分析，但是这两种是最流行的，如果你想测试一下一个 ORACLE 中的单个的 SQL 语句的运行时间，你可以使用 SQL*PLUS 中的命令 SET TIMING ON。

这个命令和其它更多的 SET 命令会在第 20 天的《SQL*PLUS》中提及。

SYBASER 的 SQL SERVER 也有用于诊断 SQL 的工具，你可以将 SET 命令作为选项加入到你的 SQL 语句中（这些命令与 ORACLE 的命令类似）。一些常用的命令 SET SHOWPLAN ON，SET STATISTIC IO ON 和 SET STATISTICS TIME ON。这些 SET 命令可以在查询执行时显示每一步的输出内容，查询的读写要求和一些常用的 SQL 语句分析用的信息，SQL SERVER 的 SET 命令将会在第 19 天的《TRANSCAT SQL 简介》中提到。

总结

流化（或者说是调整）中两个主要的因素可以直接地影响 SQL 语句的性能：对应用的调整和对数据库的调整。它们中的每一个都有自己的规则，但是没有另外一方它们自己是不能完全调整工作的，首先你要让技术队伍与系统工程师一同为平衡资源和让数据库充分发挥它自己的特性而工作以有助于系统性能的提高。其中的许多特性已经被数据库软件的供应商内置于其中了。

应用程序开发人员必需知道这些数据，知道应用数据是优化数据库设计的关键所在，开发人员和编程人员必须知道什么时候就使用索引，什么时候应该增加索引，什么时候允许批量载入运行。一定要让批量载入与日常进行的工作分开。

对数据库的调整可以提高特定的应用程序访问它的速度，数据库管理员必须了解每天的操作和数据库的性能。此外，在后台调试时必须小心，DBA 通常会给出一些创造性的建议以使对数据的访问更加有效，比如使用索引或重新构造 SQL 语句。DBA 也应该熟悉在

数据库软件中所给出的工具来测量性能和提出建议。

问与答

问：如果我对我的 SQL 进行了优化，可以得到多大的性能提升？

答：性能的增加是与你的表的大小相关的，与表有没有索引还是其它的相关数据无关，在大型的表中，复杂的查询可能由几个小时减为只需几分钟，在事务处理过程中，对 SQL 语句的优化可能为最终用户节省更多的时间。

问：我应该在什么时候运行我的批量载入任务？

答：听你的 DBA 的建议，DBA 需要知道你需要插入、更新或删除多少数据，在其它批量载入任务活动时也应该经常执行 COMMIT 命令。

问：我是否应该将我的表分于不同的磁盘？

答：这在你的表很大或你的表访问任务很重时有用。

校练场

- 1、SQL 语句的流化是什么意思？
- 2、表和它的索引是否应该放在同一个磁盘上？
- 3、为什么说对 SQL 语句中各个元素的安排是非常重要的？
- 4、当全表扫描时会发生什么情况？
- 5、你如何才能避免全表扫描？
- 6、常见的对性能的障碍有哪些？

练习

- 1、让下边的 SQL 语句更易读。

```
SELECT      EMPLOYEE.LAST_NAME,      EMPLOYEE.FIRST_NAME,
EMPLOYEE.MIDDLE_NAME,EMPLOYEE.ADDRESS,      EMPLOYEE.PHONE_NUMBER,
PAYROLL.SALARY,      PAYROLL.POSITION,EMPLOYEE.SSN,      PAYROLL.START_DATE
FROM  EMPLOYEE,  PAYROLL  WHEREEMPLOYEE.SSN  =  PAYROLL.SSN  AND
EMPLOYEE.LAST_NAME LIKE 'S%' AND  PAYROLL.SALARY > 20000;
```

- 2、重新安排下边的查询条件以减少数据返回所需要的时间，并使用下边的统计（对整个表）以决定这些条件的次序。

593 individuals have the last name SMITH.

712 individuals live in INDIANAPOLIS.

3,492 individuals are MALE.

1,233 individuals earn a salary \geq 30,000.

5,009 individuals are single.

Individual_id is the primary key for both tables.

```
SELECT M.INDIVIDUAL_NAME, M.ADDRESS, M.CITY, M.STATE, M.ZIP_CODE,  
       S.SEX, S.MARITAL_STATUS, S.SALARY  
FROM MAILING_TBL M, INDIVIDUAL_STAT_TBL S  
WHERE M.NAME LIKE 'SMITH%'  
  
      AND M.CITY = 'INDIANAPOLIS'  
  
      AND S.SEX = 'MALE'  
  
      AND S.SALARY  $\geq$  30000  
  
      AND S.MARITAL_STATUS = 'S'  
  
      AND M.INDIVIDUAL_ID = S.INDIVIDUAL_ID;
```

第 16 天：用视图从数据字典中获得信息

目标

今天我们来讨论数据字典，也就是通常所说的系统目录，在今天中我们会对以下内容有充分的了解。

- 数据字典的定义。
- 数据字典中都包括有哪些方面的信息。
- 在数据字典中有几种不同类型的表。
- 从数据字典中获得有用信息的有效方法。

数据字典简介

每一种数据库都有它自己的数据字典（或系统目录）（在今天的介绍中这两个词我们都会使用）数据字典是在数据库环境中的一个系统区域，它包含着关于数据库成份的信息，数据字典中包括的信息如数据库设计、存储的 SQL 代码、用户统计、数据库过程、数据库的增长情况和对数据库性能的统计。

数据字典中包括有数据库设计信息的表，它们是由数据库设计语言（DDL）如 CREATE TABLE 所创建的，这一部分的系统目录存储着善于表列和属性的信息、表的大小、表的权限和表的增长情况信息，其它的对象也存储在了数据字典中如索引、触发机制、过程、包、和视图。

使用统计表可以给出对于特定的用户的数据库库联接和权限信息，权限主要可以分成两个部分：系统级权限和对象级权限。具有创建其它用户的权限是系统权限，但是访问表的权限属于对象权限，在数据库中也强制性地使用了任务来确保安全性，这些信息也都在数据字典中存储着。

数据字典是数据库中众多有用工具之一，它是一种保证数据库组织的方法，而且特别像库存零售中的明细账，它是一种确保数据库完整性的机制。例如，当你创建表的时候，数据库是如何知道你所创建的表在数据库中是否有已经存在的相同名字的呢？当你在创建一个查询从表中选择数据的时候，数据库又是如何知道你是否访问这个表的权限的呢？

数据字典，是数据库的心脏，所以你应该知道如何去使用它。

用户的数据字典

最终用户，系统工程师和数据库管理员都在使用数据字典，那么他们是否对它了解呢，他们的访问是直接的还是间接的呢？

最终用户，通常是为了他们而创建数据库的客户，是间接地访问系统的目录的。当一个用户尝试去登录数据库的时候，数据字典将会将该用户的名字、密码、和权限做为联接数据库的参考，数据库也会通过它来确定是否用户具有访问特定数据的权限。最终用户最常用的访问数据库的方法是通过前端应用程序。现在已经开发出了许多的图形用户界面工具可以允许让用户非常容易地构建 SQL 语句。当登录到数据库中以后，前端应用程序会立即执行对数据字典的查询以定义用户可以访问的表。然后前端应用程序可能其于从数据字典中获得的数据来创建一个本地化的系统目录。用户则可以用这个本地化的目录来从他或好想要查询的表中获得数据。

系统工程师是有着创建和设计数据库的模块，应用程序的开发以及应用程序的管理任务的用户（在一些公司中会使用另外一种称呼，如程序员、程序分析员、数据模块师来称呼他们的系统工程师）。系统工程师是直接地使用数据字典来管理和开发过程的，当访问一个已经存在的工程的时候，访问也可以通过数据库前终应用程序、开发工具以及计算机工程帮助工具来获得，对于这些用户来说常用的系统目录是查询处于他的计划组中的模块，查询应用任务及权限和查询模块开发情况的统计，在特定的计划中系统工程师也可以将数据字典应用于系统工程师的专用对象上。

数据库管理员（DBAS）是数据字典中所定义的具有最大权限的用户，与它的两组用户偶而也会使用系统目录不同，DBAS 将使用数据字典作为他们的日常工作，访问通常是通过查询的，但也可以通过数据库管理工具如 ORACLE 的 SERVER MANGER。DBA 使用数据字典中的信息来管理用户和资源以达到数据库优化运行的目的。

如你所见，所有的数据库用户都需要使用数据字典，更为重要的是，关系型数据库系统没有数据字典就无法存在了。

数据字典中的内容

这一部分使用两种关系型数据库系统的系统目录，Oracle 和 Sybase 的。尽管这两种数

数据库系统都有着他自己的特点，但是他们提供的功能是相同的，不要关心它们在名字上的不同，你只要明白数据字典的概念和它的内容就行了。

Oracle 的数据字典

由于每一个表都必须有一个所有者，系统目录在数据字典中的所有者是 SYS，Oracle 的数据字典可以分为三个基本类，用户访问视图、DBA 视图、以及动态执行表——它也会以视图的形式出现。视图可以查询用户在数据字典中的用户账号信息，如权限和目录表的创建。DBA 可以帮助数据库管理员完成日常的工作。它允许 DBA 来管理用户和数据库中的其它对象。在 ORACLE 中的动态执行表也可以为 DBA 所使用并提供了对数据库的更深层的监视。这些视图提供了诸如对过程的执行，对 ROLLBACK 段的动态使用，内存的使用等诸如此类的统计信息。动态执行表都以 V\$ 为前缀。

Sybase 的数据字典

与 ORACLE 一样，Sybase 中系统表的所有者也是 SYS，该表可以被分为两个部分：系统表和数据库表。

系统表只能为数据库所有者所拥有。这些表定义的对象（如表和索引）为多个数据库所共有，另一部分表在 Sybase SQL Server 中的称为数据库表。这些表只与每个特定的数据库相关联。

ORACLE 数据字典的内部结构

这一部分的例子会告诉你如何从数据字典中获得信息以及如何将它应用于广大关系型数据库的用户也就是，最终用户，系统工程师和 DBA。ORACLE 数据字典中有大量的关于所有类型的数据库用户的表和视图，这是为什么我们要进一步研究 ORACLE 数据字典的原因。

用户视图

用户视图是在数据字典中的可以为全部用户所拥有的视图，一个用户对其它的用户只有 CREATE SESSION 的权限，所有的用户都是这样。

你是谁？

在进行对数据库的无穷尽的知识的探险之前，你应该确切地知道你是谁（在数据库中的字眼）和你能做什么。下边的两个例子中给出两个表中的 SELECT 语句：其中一个会告诉你你是谁而另一个会告诉你可以使用哪些数据库。

INPUT:

```
SQL> SELECT * FROM USER_USERS;
```

OUTPUT:

USERNAME	USER_ID	DEFAULT_TABLESPACE	TEMPORARY	TABLESPACE	CREATED
JSMITH	29	USERS	TEMP	14-MAR-97	

分析:

USER_USERS 视图可以告诉你你在 ORACLE 中的 ID 是如何设置的，它也可以显示其它用户的详细情况，以及对用户的统计。默认的表大小和临时表的大小也可以被显示。默认的用户表的表的大小是指由该用户所创建的表的大小。临时表大小是对 JSMITH 所指定的进行排序和分组空间的大小。

INPUT/OUTPUT:

```
SQL> SELECT * FROM ALL_USERS;
```

USERNAME	USER_ID	CREATED
SYS	0	01-JAN-97
SYSTEM	5	01-JAN-97
SCOTT	8	01-JAN-97
JSMITH	10	14-MAR-97
TJONES	11	15-MAR-97
VJOHNSON	12	15-MAR-97

正如你在上边的查询结果中所看到的那样，你可以使用 ALL_USERS 视图你可以看到所有存在于数据库中的用户。但是，它不会给出像上边的那个视图那么详细的信息。因为对于这一级用户来说是没有必要的。只有对于系统级用户才会需要更详细的信息。

你有哪些权限

现在你已经知道你是谁了，如果你可以知道你能做什么就太好了。有好几个视图都可以为你提供这样的信息。USER_SYS_PRIVS 视图和 USER_ROLE_PRIVS 视图可以给你最

为权威的信息。

你可以使用 `USER_SYS_PRIVS` 来查看系统的权限。切记，系统权限是指你对于特定数据库的整体权限，这些权限并不针对任何一个对象或对象集。

INPUT:

```
SQL> SELECT * FROM USER_SYS_PRIVS;
```

OUTPUT:

USERNAME	PRIVILEGE	ADM
JSMITH	UNLIMITED TABLESPACE	NO
JSMITH	CREATE SESSION	NO

分析:

JSMITH 已经被赋予了两种系统级权限，除了获准的任务以外。请注意第二部分，创建会话（CREATE SESSION），它包括在 ORACLE 的标准任务 CONNECT，这将会在下边的例子中提到。

你可以使用 `USER_ROLE_PRIVS` 视图来查看你在数据库中被允许的任务信息。数据库任务与系统任务非常相似，当任务为权限所许可之后，可以将任务许可给该用户。切记，在任务中可能会有对象级权限。

INPUT/OUTPUT:

```
SQL> SELECT * FROM USER_ROLE_PRIVS;
```

USERNAME	GRANTED_ROLE	ADM	DEF	OS_
JSMITH	CONNECT	NO	YES	NO
JSMITH	RESOURCE	NO	YES	NO

分析:

`USER_ROLE_PRIVS` 视图可以让你查看允许你执行的任务，在早些时候曾经说过，CONNECT 包含着系统权限 CREATE SESSION。与其它的权限一样，RESOURCE 也有为数不多的权限，你可以查看一下给予用户的缺省任务，用户不能将这些任务再给予其它的用户，这对于 ADM 来说非常重要，而且该任务也不能为操作系统所允许。（参见第 12 天的《数据库安全》）

你可以访问哪些东西

现在你也许会问，我可以访问哪些内容呢？我已经知道了我是谁，我也知道了我的权

限，但是我在哪里可以将到我的数据？你可以通常查看在数据字典中的不同的可用用户视图来回答这个问题。这一部分的内容对于一些视图来说很有帮助。

也许最为基本的用户视图就是 USER_CATALOG 了，它是一个表、视图、同义字和当前所有的次序的简明目录。

INPUT:

```
SQL> SELECT * FROM USER_CATALOG;
```

OUTPUT:

TABLE_NAME	TABLE_TYPE
MAGAZINE_TBL	TABLE
MAG_COUNTER	SEQUENCE
MAG_VIEW	VIEW
SPORTS	TABLE

分析:

这个例子中给出了为你所有的所有表和相关对象的清单，出于简明的目的你也可以使用 USER_CATALOG 中的公共同义字。那就是：试一下 SELECT * FROM CAT；

另外一个有用的视图是 ALL_CATALOG，它可以让你看到其他人拥有的表。

INPUT/OUTPUT:

```
SQL> SELECT * FROM ALL_CATALOG;
```

OWNER	TABLE_NAME	TABLE_TYPE
SYS	DUAL	TABLE
PUBLIC	DUAL	SYNONYM
JSMITH	MAGAZINE_TBL	TABLE
JSMITH	MAG_COUNTER	SEQUENCE
JSMITH	MAG_VIEW	VIEW
JSMITH	SPORTS	TABLE
VJOHNSON	TEST1	TABLE
VJOHNSON	HOBBIES	TABLE
VJOHNSON	CLASSES	TABLE
VJOHNSON	STUDENTS	VIEW

分析:

作为用户你可以会拥有比上表更多的对象（系统表会加入许多的表），我们可以很容易在将这个清单简化，ALL_CATALOG 视图与 the USER_CATALOG 视图一样，但是它显示给你所有的你可以访问的表、视图、顺序、和同义字。

INPUT:

```
SQL> SELECT SUBSTR(OBJECT_TYPE,1,15) OBJECT_TYPE,
2         SUBSTR(OBJECT_NAME,1,30) OBJECT_NAME,
3         CREATED,
4         STATUS
5 FROM USER_OBJECTS
6 ORDER BY 1;
```

OUTPUT:

OBJECT_TYPE	OBJECT_NAME	CREATED	STATUS
INDEX	MAGAZINE_INX	14-MAR-97	VALID
INDEX	SPORTS_INX	14-MAR-97	VALID
INDEX	HOBBY_INX	14-MAR-97	VALID
TABLE	MAGAZINE_TBL	01-MAR-97	VALID
TABLE	SPORTS	14-MAR-97	VALID
TABLE	HOBBY_TBL	16-MAR-97	VALID

分析:

你可以使用 USER_OBJECTS 视图来获得关于用户所拥有对象的通用信息如名字、类型、数据的创建、数据的更新以及对象的状态，在上一个查询中，我们对每一个数据对象的创建进行了确认。

INPUT/OUTPUT:

```
SQL> SELECT TABLE_NAME, INITIAL_EXTENT, NEXT_EXTENT
2 FROM USER_TABLES;
```

TABLE_NAME	INITIAL_EXTENT	NEXT_EXTENT
MAGAZINE_TBL	1048576	540672
SPORTS	114688	114688

分析:

从 USER_TABLES 中可以选择非常多的有用信息，就看你想要知道什么。大部分的数据都是由存储信息组成。

注：注意最初的和最大的输出值是字节，在一些解释器中你可能会看到在列之间加以空格以更具可读性。见第 19 天的《事务处理 SQL》和第 20 天的《SQL*PLUS》。

ALL_TABLES 与 USER_TABLES 的关系与 ALL_CATALOG 和 USER_CATALOG 关系

一样，也就是说，ALL_TABLES 可以让你查看所有的你可以访问的表，而不只是你所拥有的表。ALL_TABLES 也可以包括在其它用户目录中存在的表。

INPUT/OUTPUT:

```
SQL> SELECT SUBSTR(OWNER,1,15) OWNER,
2         SUBSTR(TABLE_NAME,1,25) TABLE_NAME,
3         SUBSTR(TABLESPACE_NAME,1,13) TABLESPACE
4 FROM ALL_TABLES;
```

OWNER	TABLE_NAME	TABLESPACE
SYS	DUAL	SYSTEM
JSMITH	MAGAZINE_TBL	USERS
SMITH	SPORTS	USERS
VJOHNSON	TEST1	USERS
VJOHNSON	HOBBIES	USERS
VJOHNSON	CLASSES	USERS

分析:

你又一次选择了只是你想得到的信息，在 ALL_TABLES 的许多附加列中也包含着有用的信息。

作为一个数据库用户，你可以通过查询 USER_SEGMENTS 视图来监视你的表和索引的增长情况，如同它的名字一样，USER_SEGMENTS 给你了关于每个段的信息。如存储信息等等，一个段可以会由一个表、索引、ROLLBACK 簇、临时表或缓存组成，下边的例子告诉你如何从这个视图获得有用的信息。

INPUT/OUTPUT:

```
SQL> SELECT SUBSTR(SEGMENT_NAME,1,30) SEGMENT_NAME,
2         SUBSTR(SEGMENT_TYPE,1,8) SEG_TYPE,
3         SUBSTR(TABLESPACE_NAME,1,25) TABLESPACE_NAME,
4         BYTES, EXTENTS
5 FROM USER_SEGMENTS
6 ORDER BY EXTENTS DESC;
```

SEGMENT_NAME	SEG_TYPE	TABLESPACE_NAME	BYTES	EXTENTS
MAGAZINE_TBL	TABLE	USERS	4292608	7
SPORTS_INX	INDEX	USERS	573440	4

SPORTS	TABLE	USERS	344064	2
MAGAZINE_INX	INDEX	USERS	1589248	1

分析：

上边的输出是按 EXTENTS 递减的原则进行了排序的，段增长最多的将会出现于第一个行中。

现在你已经知道了都有哪些表是你访问的，你也许会想知道对于每个表你都可以做什么，你的查询是否受限？或你有权更新表的内容吗？ALL_TAB_PRIVS 视图可以告诉你作为一个数据库用户你在每一个可以使用的表中的权限。

INPUT/OUTPUT：

```
SQL> SELECT SUBSTR(TABLE_SCHEMA,1,10) OWNER,
2          SUBSTR(TABLE_NAME,1,25) TABLE_NAME,
3          PRIVILEGE
4 FROM ALL_TAB_PRIVS;
```

OWNER	TABLE_NAME	PRIVILEGE
SYS	DUAL	SELECT
JSMITH	MAGAZINE_TBL	SELECT
JSMITH	MAGAZINE_TBL	INSERT
JSMITH	MAGAZINE_TBL	UPDATE
JSMITH	MAGAZINE_TBL	DELETE
JSMITH	SPORTS	SELECT
JSMITH	SPORTS	INSERT
JSMITH	SPORTS	UPDATE
JSMITH	SPORTS	DELETE
VJOHNSON	TEST1	SELECT
VJOHNSON	TEST1	INSERT
VJOHNSON	TEST1	UPDATE
VJOHNSON	TEST1	DELETE
VJOHNSON	HOBBIES	SELECT
VJOHNSON	CLASSES	SELECT

分析：

如你所见，你可以操纵在一些表中的数据，然后对于其它的一些表你则只拥有只读访问权限。

当你创建一个对象的时候，如果你不想把它存于默认的地点你通常要知道你可以把

它存放于何处。ORACLE 的表空间的分散的，它们中的每一个都可以存储对象。每一个表空间都被分配了一定数据的磁盘空间，这要根据你的系统的可用性而定。磁盘空间通常是从数据管理员处获得的。

下面的查询是从一个叫 USER_TABLESPACES 的视图中进行查询的，它会给出你访问的表空间，分配对象的初始大小和它们的下一个大小以及它们的状态。

INPUT/OUTPUT:

```
SQL> SELECT SUBSTR(TABLESPACE_NAME,1,30) TABLESPACE_NAME,
2      INITIAL_EXTENT,
3      NEXT_EXTENT,
4      PCT_INCREASE,
5      STATUS
6 FROM USER_TABLESPACES;
```

TABLESPACE_NAME	INITIAL_EXTENT	NEXT_EXTENT	PCT_INCREASE	STATUS
SYSTEM	32768	16384	1	ONLINE
RBS	2097152	2097152	1	ONLINE
TEMP	114688	114688	1	ONLINE
TOOLS	32768	16384	1	ONLINE
USERS	32768	16384	1	ONLINE

分析:

这在当你创建一个需要存储空间的对象如表和索引时，这个查询是非常有用的。当表或查询建立以后，如果没有在 DDL 中指定它的初始和后续存储参数，这些表和索引会采用这些数值的默认值。相同的概念也可以应用于 PCT INCREASE，它是一个 ORACLE 参数用以指定当一个对象的大小在增长时它为它分配空间的百分比。如果当表和索引在创建是没有指定这个数值，数据库服务程序会为它分派默认的数值，通过查看这些默认值你可以决定是否应该在创建表和索引时使用存储子句来指定它的初始及后续大小。

但是在有些时候，你需要对你访问的表空间知道得更多，那就是在底层生成一个表的时候。例如：你需要知道你在表空间方面所受的限制以使你可以更好地创建和组合你的对象，USER_TS_QUOTAS 视图会提供你所需要的信息，下边的查询给出了在数据库中创建对象时用户空间的限制。

INPUT/OUTPUT:

```
SQL> SELECT SUBSTR(TABLESPACE_NAME,1,30) TABLESPACE_NAME,
```

2 BYTES, MAX_BYTES

3 FROM USER_TS_QUOTAS;

TABSPACE_NAME	BYTES	MAX_BYTES
SYSTEM	0	0
TOOLS	5242880	16384
USERS	573440	-1

分析:

上边的输出是典型的从 ORACLE 数据字典中的输出，以字节的形式给出了可由用户支配的表空间。MAX_BYTES 给出了用户配额的最大值，在这一列中的头两个数据无需说明，在第三列中的-1 的意思是最大值不受限制，也就是说用户使用的表空间没有最大限制。

注：在 SUBSTR 函数中出现了许多上边的查询中的数据字典视图，在数据返回后你可以使用许多你以前学过的函数来对返回的数据进行控制以使其更具有可读性。你也可以对你所返回数据的输出长度进行限制，就像我们在这些例子中所做的那样。

所有的这些例子给出的常规的用户如何从数据字典中取得有用的信息，这些视图只是存在于 ORACLE 数据字典中的很少的一部分，检查你的数据库解释器以看它是否在你的数据库字典中可用是非常重要的。切记：你应该使用数据字典来管理你的数据库，尽管对于不同的数据库系统目录是不相同的，你只需要明白这个概念以及如何取得你所需要的数据来支持你的工作就行了。

系统数据库管理员视图

在 ORACLE 数据字典中 DBA 视图通常是最为重要的，在大多数情况下它是由 DBA 来访问的，这些视图对于 DBA 来说是至关重要的，如果 DBA 没有这些视图就如同木匠没有的锤子。

如你所料，你必须具有 SELECT_ANY_TABLE 的系统权限才能访问这些表，该权限包含在 DBA 规则中。例如：如果你是 JSMITH，那么你就没有访问 DBA 表的权限。

INPUT:

SQL> SELECT *

2 FROM USER_ROLE_PRIVS;

OUTPUT:

USERNAME	GRANTED_ROLE	ADM	DEF	OS_
----------	--------------	-----	-----	-----

JSMITH	CONNECT	NO	YES	NO
JSMITH	RESOURCE	NO	YES	NO

INPUT/OUTPUT:

```
SQL> SELECT *
      2 FROM SYS.DBA_ROLES;

FROM SYS.DBA_ROLES;

ERROR at line 2:

ORA-00942: table or view does not exist
```

分析:

如果你没有得到合适的权限却想访问这些表，那么返回的错误会说这个表不存在。这个信息容易让人产生误解，事实上这个表不存在是因为该用户不能“看到”这个表，要解决这个问题你需要将 DBA 规则应用于 JSMITH，当然这必须要得到 DBA 的许可。

数据库用户信息

USER_USER 与 ALL_USER 视图将会给你最少的关于用户的信息，DBA 视图被称为 DBA_USERS（它为 SYS 所拥有）所有的关于所有用户的信息，前提是你有 DBA 规则或 SELECT_ANY_TABLE 权限，如下例如示：

INPUT:

```
SQL> SELECT * FROM SYS.DBA_USERS;
```

OUTPUT:

USERNAME	USER_ID	PASSWORD	DEFAULT_TABLESPACE
SYS	0	4012DA490794C16B	SYSTEM
JSMITH	5	A4A94B17405C10B7	USERS

（实际为一个表格，这里由于排版需要改为两个）

USERNAME	TEMPORARY_TABLESPACE	CREATED	PROFILE
SYS	TEMP	06-JUN-96	DEFAULT
JSMITH	TEMP	06-JUN-96	DEFAULT

分析:

当你选择了 DBA_USERS 视图中的所有内容，你会看到对于每个用户来说至为重要的信息，注意这里的 PASSWORD 是经过加密处理的。这个视图是 DBA 管理其它用户的重要

视图。

数据库安全

在数据字典中有三个基本的视图与数据库安全相关的，尽管想得到完整的信息你还要查阅其它相关的信息。这三个视图与数据库的规则、规则对用户的授权以及系统权限对用户的授权相关，这三个视图会在 DBA_ROLES、DBA_ROLE_PRIVS 和 DBA_SYS_PRIVS 这三个部分中介绍。下边的查询显示了如何去获得相关的数据库安全信息。

INPUT:

```
SQL> SELECT * FROM SYS.DBA_ROLES;
```

OUTPUT:

ROLE	PASSWORD
CONNECT	NO
RESOURCE	NO
DBA	NO
EXP_FULL_DATABASE	NO
IMP_FULL_DATABASE	NO
END_USER_ROLE	NO

分析:

视图 DBA_ROLES 给出了所有在数据库中创建的规则的信息。它给出的规则的名字以及规则是否存在密码。

INPUT:

```
SQL> SELECT *
2 FROM SYS.DBA_ROLE_PRIVS
3 WHERE GRANTEE = 'RJENNINGS';
```

GRANTEE	GRANTED_ROLE	ADM	DEF
RJENNINGS	CONNECT	NO	YES
RJENNINGS	DBA	NO	YES
RJENNINGS	RESOURCE	NO	YES

分析:

DBA_ROLE_PRIVS 给出的关于赋给用户的数据库规则的信息。第一列为被授权人，第二列显示了被授予的规则。注意，对用户所授予的每一个规则都会占用表中的一个记录，

ADM 通过后边的 ADM 选项来确定这个规则是否是允许的。也就是说该用户是否有权将该规则授予其它的用户。最后一列为默认，表明这个规则是否是用户的默认规则。

分析：

DBA_SYS_PRIVS 给了所有授予用户的系统权限。这个视图与 DBA_ROLE_PRIVS 视图类似，你可以通过把系统权限授予规则来把这些系统权限包括在规则中。就像你是一个用户一样。

数据库对象

数据库对象是 DBA 所关心了又一个重要内容，在数据字典中有几个视图提供了有关数据库对象如表和索引的信息，通常这些视图你可以获得常用的信息或详细的信息。

INPUT：

```
SQL> SELECT *
      2 FROM SYS.DBA_CATALOG
      3 WHERE ROWNUM < 5;
```

OUTPUT：

OWNER	TABLE_NAME	TABLE_TYPE
SYS	CDEF\$	TABLE
SYS	TAB\$	TABLE
SYS	IND\$	TABLE
SYS	CLU\$	TABLE

分析：

DBA_CATALOG 所做的事与 USER_CATALOG 是一样的。只显示了所有者的表，与之相对比，USER_CATALOG 则用于显示当前用户的表，用 DBA_CATALOG 视图 DBA 可以快速地查看所有的表。

下边的查询给出了对于特定的数据库都存在哪种类型的对象。

技巧：出于测试的目的你可以使用 ROWNUM 来限制查询返回特定数目的行。它与 ROWID 一样可以在任何数据库表和视图使用。

INPUT/OUTPUT：

```
SQL> SELECT DISTINCT(OBJECT_TYPE)
      2 FROM SYS.DBA_OBJECTS;
```

OBJECT_TYPE	OBJECT_TYPE
CLUSTER	PROCEDURE
DATABASE LINK	SEQUENCE
FUNCTION	SYNONYM
INDEX	TABLE
PACKAGE	TRIGGER
PACKAGE BODY	VIEW

分析：

在上边的查询中 DISTINCT 功能可以让存在于数据库中的对象只出现一次。在数据库的设计和开发时使用这个查询来发现数据库中都有哪些对象是非常好的。

DBA_TABLES 视图给出了关于数据库表和大多数与存储有关的数据库对象的信息。

INPUT/OUTPUT：

```
SQL> SELECT SUBSTR(OWNER,1,8) OWNER,
2         SUBSTR(TABLE_NAME,1,25) TABLE_NAME,
3         SUBSTR(TABLESPACE_NAME,1,30) TABLESPACE_NAME
4 FROM SYS.DBA_TABLES
5 WHERE OWNER = 'JSMITH';
```

OWNER	TABLE_NAME	TABLESPACE_NAME
JSMITH	MAGAZINE_TBL	USERS
JSMITH	HOBBY_TBL	USERS
JSMITH	ADDRESS_TBL	SYSTEM
JSMITH	CUSTOMER_TBL	USERS

分析：

除了 ADDRESS_TBL 以外，所有的表都在 USER 表空间中，ADDRESS_TBL 是在系统表空间中的。因为这是唯一的一个你可能存储于系统表空间中的表。作为 DBA 需要知道这件事。这个查询对你的帮助不小。

JSMITH 应该马上把这个表移到其它符合条件的表空间中去。

DBA_SYNONYMS 给出了所有存在于数据库中的同义字的清单，DBA_SYNONYMS 给出了所有数据库用户的同义字清单，与 USER_SYNONYMS 不同——它只是给出了当前用户私有的同义字。

INPUT/OUTPUT：

```
SQL> SELECT SYNONYM_NAME,
```

```

2      SUBSTR(TABLE_OWNER,1,10) TAB_OWNER,
3      SUBSTR(TABLE_NAME,1,30) TABLE_NAME
4 FROM SYS.DBA_SYNONYMS
5 WHERE OWNER = 'JSMITH';

```

SYNONYM_NAME	TAB_OWNER	TABLE_NAME
TRIVIA_SYN	VJOHNSON	TRIVIA_TBL

分析:

下边的例子表明 JSMITH 有对 VJOHNSON 所有的 TRIVIA_TBL 表有一个名字叫 TRIVIA_SYN 的同义字。

现在假设你想得到所有的 JSMITH 的表和索引的清单，你也许会写出像下边这样的查询，它使用了 DBA_INDEXES。

INPUT/OUTPUT:

```

SQL> SELECT SUBSTR(TABLE_OWNER,1,10) TBL_OWNER,
2      SUBSTR(TABLE_NAME,1,30) TABLE_NAME,
3      SUBSTR(INDEX_NAME,1,30) INDEX_NAME
4 FROM SYS.DBA_INDEXES
5 WHERE OWNER = 'JSMITH'
6 AND ROWNUM < 5
7 ORDER BY TABLE_NAME;

```

TBL_OWNER	TABLE_NAME	INDEX_NAME
JSMITH	ADDRESS_TBL	ADDR_INX
JSMITH	CUSTOMER_TBL	CUST_INX
JSMITH	HOBBY_TBL	HOBBY_PK
JSMITH	MAGAZINE_TBL	MAGAZINE_INX

分析:

像上边这样一个很容易的查询可以给出一个计划（工程）和与之相中所使用的表。

INPUT/OUTPUT:

```

SQL> SELECT SUBSTR(TABLE_NAME,1,15) TABLE_NAME,
2      SUBSTR(INDEX_NAME,1,30) INDEX_NAME,
3      SUBSTR(COLUMN_NAME,1,15) COLUMN_NAME,
4      COLUMN_POSITION

```

```

5 FROM SYS.DBA_IND_COLUMNS
6 WHERE TABLE_OWNER = 'JSMITH'
7 AND ROWNUM < 10
8 ORDER BY 1,2,3;

```

TABLE_NAME	INDEX_NAME	COLUMN_NAME	COLUMN_POSITION
ADDRESS_TBL	ADDR_INX	PERS_ID	1
ADDRESS_TBL	ADDR_INX	NAME	2
ADDRESS_TBL	ADDR_INX	CITY	3
CUSTOMER_TBL	CUST_INX	CUST_ID	1
CUSTOMER_TBL	CUST_INX	CUST_NAME	2
CUSTOMER_TBL	CUST_INX	CUST_ZIP	3
HOBBY_TBL	HOBBY_PK	SAKEY	1
MAGAZINE_TBL	MAGAZINE_INX	ISSUE_NUM	1
MAGAZINE_TBL	MAGAZINE_INX	EDITOR	2

现在你已经选择一在每一个表中的索引过的列并根据索引出现的次序进行了排序。你已经学习了表，但是如何才能更好地把握表呢？表空间比表和索引之类的对象更高级，表空间是 ORACLE 为数据库分配空间的机制，要分配空间，你必须知道当前有多少空间可供使用，你可以对 DBA_TABLESPACES 视图执行一个查询来查看所有的表空间以及它们的状态，如下例所示：

INPUT/OUTPUT：

```
SQL> SELECT TABLESPACE_NAME, STATUS
```

```
2 FROM SYS.DBA_TABLESPACES;
```

TABLESPACE_NAME	STATUS
SYSTEM	ONLINE
RBS	ONLINE
TEMP	ONLINE
TOOLS	ONLINE
USERS	ONLINE
DATA_TS	ONLINE
INDEX_TS	ONLINE

分析：

上边的例子告诉了你当前有多少表空间处于在线状态，也就是说它对于你来说是可用的，如果它是离线状态，那么在它之中的数据库对象（也就是表）将是不可访问的。

那么对于 JSMITH 来说有多少配额可供他访问呢？也就是说，对于 JSMITH 来说他可以使用多少空间来存储数据库对象呢？

INPUT/OUTPUT:

```
SQL> SELECT TABLESPACE_NAME,
2         BYTES,
3         MAX_BYTES
4 FROM SYS.DBA_TS_QUOTAS
5 WHERE USERNAME = 'JSMITH';
```

TABLESPACE_NAME	BYTES	MAX_BYTES
DATA_TS	134111232	-1
INDEX_TS	474390528	-1

分析:

JSMITH 对于可供他访问和使用的表空间是没有限制的，在这种情况下空间的使用是本着先入为主的原则进行的。举例来说，如果 JSMITH 在 DATA_TS 中使用了所有的表空间，那么其它的用户就不能在这里创建对象了。

数据库的生长

在这一部分有两个视图可以对管理数据库的生长情况进行控制。它们是 DBA_SEGMENTS 和 DBA_EXTENTS。DBA_SEGMENTS 提供了关于每一段或数据对象如存储分配，空间使用和扩展的信息。每次当表和索引的增长超过了预先的指定就会开始下一次的扩展。采用这种方法增长的表通常会有碎片的产生。DBA_EXTENTS 则给出的段每次扩展的信息。

INPUT:

```
SQL> SELECT SUBSTR(SEGMENT_NAME,1,30) SEGMENT_NAME,
2         SUBSTR(SEGMENT_TYPE,1,12) SEGMENT_TYPE,
3         BYTES,
4         EXTENTS,
5 FROM SYS.DBA_SEGMENTS
6 WHERE OWNER = 'TWILLIAMS'
7 AND ROWNUM < 5;
```

OUTPUT:

SEGMENT_NAME	SEGMENT_TYPE	BYTES	EXTENTS
INVOICE_TBL	TABLE	163840	10
COMPLAINT_TBL	TABLE	4763783	3
HISTORY_TBL	TABLE	547474996	27
HISTORY_INX	INDEX	787244534	31

分析:

看一下从 DBA_SEGMENTS 的输出, 你可以很容易地通常 EXTENTS 的数字来确定增长最多的表的情况。在本例子 HISTORY_TBL 和 HISTORY_INX 的增长要比另外的两个表快得多。

现在你可以看一下表每次增长的情况, 我们以 INVOICE_TBL 为例:

INPUT/OUTPUT:

```
SQL> SELECT SUBSTR(OWNER,1,10) OWNER,
2         SUBSTR(SEGMENT_NAME,1,30) SEGMENT_NAME,
3         EXTENT_ID,
4         BYTES
5 FROM SYS.DBA_EXTENTS
6 WHERE OWNER = 'TWILLIAMS'
7 AND SEGMENT_NAME = 'INVOICE_TBL'
8 ORDER BY EXTENT_ID;
```

OWNER	SEGMENT_NAME	EXTENT_ID	BYTES
TWILLIAMS	INVOICE_TBL	0	16384
TWILLIAMS	INVOICE_TBL	1	16384
TWILLIAMS	INVOICE_TBL	2	16384
TWILLIAMS	INVOICE_TBL	3	16384
TWILLIAMS	INVOICE_TBL	4	16384
TWILLIAMS	INVOICE_TBL	5	16384
TWILLIAMS	INVOICE_TBL	6	16384
TWILLIAMS	INVOICE_TBL	7	16384
TWILLIAMS	INVOICE_TBL	8	16384
TWILLIAMS	INVOICE_TBL	9	16384

分析:

这个例子显示了该表每次的增长、EXTEND_ID 和每次增长的字节大小情况。这里每

次的增长只有 16K，而且已经增长了 10 次了，你也许应该重建数据库并重新生成表并增加 initial_extent 的大小以优化空间的使用，重新生成表将可以将表中的数据放放一个单一的片段中，所以也就还会有碎片产生了。

空间分配

ORACLE 是使用数据文件来分配数据库的空间的，空间在逻辑上以表空间的形式存在，但是它是数据文件的物理形式存在于磁盘上的，在许多的解释器中，在数据文件中也可直接包括数据，尽管这些文件以能会以其它的名字来引用。视图 DBA_DATA_FILES 可以让你看到表空间的实际分配。

INPUT/OUTPUT:

```
SQL> SELECT SUBSTR(TABLESPACE_NAME,1,25) TABLESPACE_NAME,
2          SUBSTR(FILE_NAME,1,40) FILE_NAME,
3          BYTES
4 FROM SYS.DBA_DATA_FILES;
```

TABSPACE_NAME	FILE_NAME	BYTES
SYSTEM	/disk01/system0.dbf	41943040
RBS	/disk02/rbs0.dbf	524288000
TEMP	/disk03/temp0.dbf	524288000
TOOLS	/disk04/tools0.dbf	20971520
USERS	/disk05/users0.dbf	20971520
DATA_TS	/disk06/data0.dbf	524288000
INDEX_TS	/disk07/index0.dbf	524288000

分析:

你现在可以看到在数据库中存在的每个表空间实际上被分配了多大的空间。注意数据库文件的名称与它所属的表空间是一一对应的。

可用空间

就像下边的例子所显示的一样，DBA_FREE_SPACE 视图可以告诉你在每个表格空间中还有多少自由的空间可以使用。

INPUT:

```
SQL> SELECT TABLESPACE_NAME, SUM(BYTES)
2 FROM SYS.DBA_FREE_SPACE
3 GROUP BY TABLESPACE_NAME;
```

OUTPUT:

TABLESPACE_NAME	SUM(BYTES)
SYSTEM	23543040
RBS	524288000
TEMP	524288000
TOOLS	12871520
USERS	971520
DATA_TS	568000
INDEX_TS	1288000

分析:

上边的例子给出的所有的每个表空间的自由空间，如果你只是使用 SELECT 语句而没有使用 SUM 函数的话你也会看到每一段的自由空间情况。

ROLLBACK 段

为事务所预留的 ROLLBACK 区域对数据库的性能影响是非常大的，你需要知道有多少的 ROLLBACK 段是可用的，视图 DBA_ROLLBACK_SEGS 可以为你提供这些信息。

INPUT:

```
SQL> SELECT OWNER,
2 SEGMENT_NAME
3 FROM SYS.DBA_ROLLBACK_SEGS;
```

OUTPUT:

OWNER	SEGMENT_NAME
SYS	SYSTEM
SYS	R0
SYS	R01
SYS	R02
SYS	R03
SYS	R04
SYS	R05

分析：

这个例子中通过运行一个简单的查询列出的所有 ROLLBACK 段的名称，其实它还有更多的对你有帮助的数据。

动态执行视图

ORACLE 的数据库管理经常会访问动态执行视图因为它对提供了比其它的数据字典视图更为详细的对内部性能的量度。（在 DBA 视图中也包括了一些相同的信息）

这些视图涉及到了相当多的细节——依据特定的解释器而定，这一部分给出在数据字典中包括的大体上的信息。

会话信息

对 V\$SESSION 视图执行 DESCRIBE（在第 20 天的《SQL*PLUS》中会详细讨论）命令，你可以看到这个视图中的详细内容。

INPUT：

```
SQL> DESCRIBE V$SESSION
```

OUTPUT：

Name	Null?	Type
SADDR		RAW(4)
SID		NUMBER
SERIAL#		NUMBER
AUDSID		NUMBER
PADDR		RAW(4)
USER#		NUMBER
USERNAME		VARCHAR2(30)
COMMAND		NUMBER
TADDR		VARCHAR2(8)
LOCKWAIT		VARCHAR2(8)
STATUS		VARCHAR2(8)
SERVER		VARCHAR2(9)
SCHEMA#		NUMBER
SCHEMANAME		VARCHAR2(30)
OSUSER		VARCHAR2(15)

Name	Null?	Type
PROCESS		VARCHAR2(9)
MACHINE		VARCHAR2(64)
TERMINAL		VARCHAR2(10)
PROGRAM		VARCHAR2(48)
TYPE		VARCHAR2(10)
SQL_ADDRESS		RAW(4)
SQL_HASH_VALUE		NUMBER
PREV_SQL_ADDR		RAW(4)
PREV_HASH_VALU		NUMBER
E		
MODULE		VARCHAR2(48)
MODULE_HASH		NUMBER
ACTION		VARCHAR2(32)
ACTION_HASH		NUMBER
CLIENT_INFO		VARCHAR2(64)
FIXED_TABLE_SEQ		NUMBER
UENCE		
ROW_WAIT_OBJ#		NUMBER
ROW_WAIT_FILE#		NUMBER
ROW_WAIT_BLOC		NUMBER
K#		
ROW_WAIT_ROW#		NUMBER
LOGON_TIME		DATE
LAST_CALL_ET		NUMBER

如果你想得到关于当前数据库的会话信息。你可以对 V\$SESSION 写像下边这样的
一个查询。

INPUT/OUTPUT:

```
SQL> SELECT USERNAME, COMMAND, STATUS
2 FROM V$SESSION
3 WHERE USERNAME IS NOT NULL;
```

USERNAME	COMMAND	STATUS
TWILLIAMS	3	ACTIVE
JSMITH	0	INACTIVE

分析:

TWILLIAMS 已经登录到了数据库中并且对数据进行了选择，这可以从他执行了命令

3 看出来。

JSMITH 只不过是登录到了数据库中，他的会话不是处于活动状态的，他也没有执行任何命令，你可以参考你的数据库文档以发现在数据字典中都有哪些命令的定义，这些命令包括 SELECT、INSERT、UPDATE、DELETE、CREATE TABLE 和 DROP TABLE。

运行统计

也可以对用户的会话进行执行统计，它比今天讨论的其它视图更加依赖于具体的解释器。

执行的统计包括如数据库读写速率、对表的成功命中、系统全局区域的使用、内存缓冲的使用、ROLLBACK 的详细信息，事务历史记录的信息以及表的锁定和等待情况，这几乎全是底层知识。

计划表

计划表是 ORACLE 的 SQL 语言工具所使用的默认表，EXPLAIN PLAN（见第 15 天）表是由被称为 UTLXPLAN.SQL 的 ORACLE 脚本所创建的。当软件被安装时拷贝到的服务器上，数据是由 EXPLAIN PLAN 工具来产生，

总结

尽管具体的数据字典在不同的解释器中是不同的，但这些概念适用于所有的关系数据库系统，你必须遵守你的数据库管理系统的语法和规则。但是今天的例子将会给你对数据字典进行查询使用的信息。which populates the PLAN table with information about the object being accessed and the steps in the execution plan of an SQL statement.

注：对数据字典进行探险是很危险的，你需要进行更有效的学习才行。

问与答

问：为什么在数据字典中我应该使用表和视图？

答：使用视图是在数据字典中找到关于你的数据库的信息的最有效的方法，表可以告

诉你可以访问什么以及你的权限如何，它也可以帮助你监视其它的数据库事件如用户过程和数据库性能等。

问：数据字典是怎样创建的？

答：数据字典是在数据库初始化的时候创建的，在创建数据库的时候 ORACLE 提供了许多的脚本来运行，对于特定的数据库系统目录这些脚本会创建必要的表和视图。

问：数据字典是怎样更新的？

答：在进行日常操作时数据字典由关系数据库系统在内部进行更新，当我们改变一个表的结构的时候，数据字典也将会做出相应的改变，你不要尝试手工去修改数据字典，这可能会导致数据库的崩溃。

问：我如何才能发现谁对数据库做了些什么？

答：一般来说，在系统目录中的表和视图可以让你对用户的活动进行审核。

校练场

- 1、在 ORACLE 中，你如何才能知道哪些表和视图是为你所有的？
- 2、在数据字典中存储有哪些信息？
- 3、你如何才能进行性能统计？
- 4、数据库对象都有哪些？

练习

假设你管理了一个中小型的数据库系统，你的职责是开发和管理数据库，某人向表中插入了大量的数据并收到了一个空间不足的错误信息。你必须断定问题产生的原因。是对该用户配额的表空间增加还是你需要增加为表空间分配的磁盘空间。要一步一步地列出你需要从数据字典中得到的信息——不必给出具体的表和视图的名称。

第 17 天：使用 SQL 来生成 SQL 语句

目标

在今天你将学习从后台生成更多的 SQL 语句的概念，在今天中你将会明白以下内容：

- 从查询中生成 SQL 的好处
- 如何从其它的 SQL 语句的输出中生成查询
- 如何在 SQL 语句方法中使用数据字典和数据库表。

使用 SQL 来生成 SQL 语句的目的

从 SQL 中生成 SQL 语句的意思简而言之就是写一个 SQL 语句，它可以其它形式的 SQL 语句或命令。到目前为止，你所学所用的 SQL 语句，如对表中数据的操作，或产生一行结果，或可以生成一些报告。今天，你将学习写一个可以生成其它查询或 SQL 语句的查询。

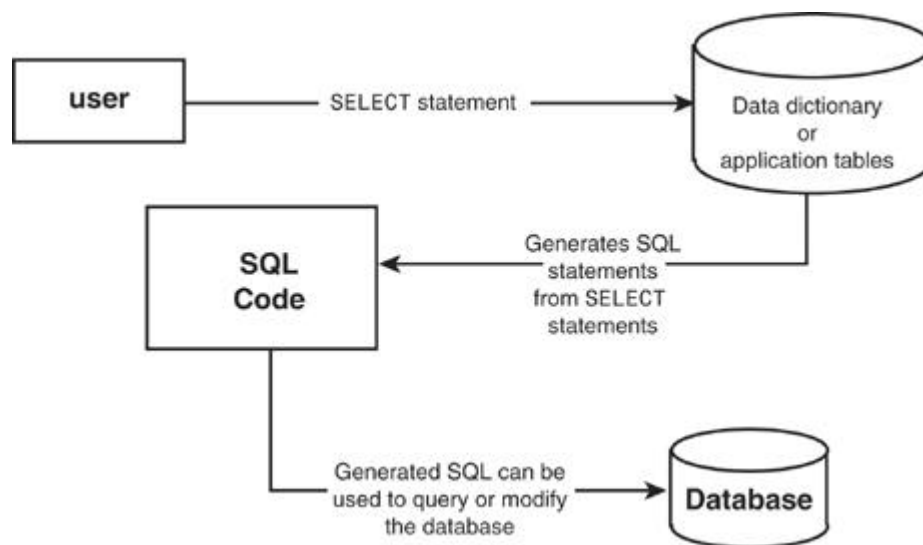
为什么我们需要从查询中生成 SQL 语句呢？最初，这样做的目的是为了简单和有效，你并不一定需要生成 SQL 语句，但是如果你不这样做你可能会忽视 SQL 的最为强大的特性。有许多人甚至根本就不知道有这样的功能存在。

生成 SQL 语句并不是必须的要求，因为你完全可以手工写并运行全部的 SQL 语句——尽管对于特定的工作这是单调和乏味的。可是如果你的工作期限很紧迫的话你可以考虑生成 SQL 语句。举例来说：如果你的老板批准了 90 个在市场部工作的人可以访问一些新的表的话（而你还想回家去吃晚饭）。由于在数据库中的一些用户并不是工作在市场部的，所以你不能简单地将表的访问权限设为可以公共访问。当你有多个组的用户并且他们的访问权限不同的时候，你可能会想用强制性的安全规则——这是一种内建的对用户的访问进行控制的方法。在这种情况下，你可以创建一个 SQL 语句来对每一个在市场部工作的人生成一个 GRANT 语句，也就是说，它对每一个在市场部工作的人赋予了相同的规则。

你可以在许多的情况下发现从一个 SQL 语句中生成 SQL 语句的优点。例如：你可能要执行一组由许多条相似的 SQL 语句组成的语句组或是你想生成针对数据字典的 DDL，当从一个 SQL 语句中生成其它的 SQL 语句时，你会经常从你的输出或数据字典或在数据库中的计划表中来取得数据。下图给出了这个过程。

如你在下图所见，可以对数据库执行一个 SELECT 语句，它输出的结果或是数据字典中的内部或是在数据库中应用表的内容。你的语句可以将这些结果嵌入到一个或更多的 SQL 语句中。如果在你的语句中返回了 100 条记录，那么你就会得到 100 条语句。如果你成功地从数据库中生成了这些 SQL 代码，那么你还可以对数据库来应用这些 SQL 语句，它们将对数据库执行一系列的查询工作。

今天剩下的时间我们主要集中于向你讲解如何从一个 SQL 语句中生成其它的 SQL 语句的例子。你的大多数的信息将来源于数据字典，所以你应该回想一下你昨天所学习的内容（见第 16 天《使用视图从数据字典中获得有用的信息》）。



注：在今天的例子中我们使用 PERSONAL ORACLE7。与以往一样，你应该将在今天所讲座的内容与你所使用的数据库解释器的语法结合起来。

几个 SQL*PLUS 命令

在今天的例子中我们要使用几个新的命令，也就是 SQL*PLUS 命令——针对 PERSONAL ORACLE 的用以对输出的格式进行控制的命令。（见第 20 天的《SQL*PLUS》）这些命令也是在 SQL>提示符下运行的，它们也可以在文件中使用。

注：尽管这些命令是针对 PERSONAL ORACLE7 的，但是在其它的数据库解释器中也有类似的命令。例如：在 Transact-SQL 中就有（参见第 19 天《Transact-SQL 简介》）

SET ECHO ON/OFF

当你 set echo on 时，你会在执行的时候看到你的 SQL 语句，set echo off 的意思就是你不想在执行的时候看到 SQL 语句——你只想看到输出的结果。

```
SET ECHO [ ON | OFF ]
```

SET FEEDBACK ON/OFF

FEEDBACK 就是你的查询所输入的行数，例如：如果你运行的 SELECT 语句返回 30 行数据，那么你的 FEEDBACK 将会是：

```
30 rows selected
```

SET FEEDBACK ON 会显示对行的计数，SET FEEDBACK OFF 则在你的结果输出时不会对行进行计数。

```
SET FEEDBACK [ ON | OFF ]
```

SET HEADING ON/OFF

HEADING 就是指你的 SELECT 语句的输出结果的头部，诸如 LAST_NAME 或 CUSTOMER_ID 的就是。SET HEADING ON 是默认的，当然，OFF 的时候就是在输出的时候不输出列标头。

```
SET HEADING [ ON | OFF ]
```

SPOOL FILENAME/OFF

SPOOL 可以将你的查询结果直接地输入到一个文件中，要想打开 SPOOL 文件，你需要输入：

```
spool filename
```

如果想关掉 SPOOL 文件，你应该输入：

```
spool off
```

START FILENAME

大多数的我们所学习的 SQL 命令都是在 SQL> 下运行的，另外的一种运行 SQL 语句的方法是创建 SQL 执行文件，在 SQL*PLUS 中，运行 SQL 文件的命令是 START FILENAME。

START FILENAME

ED FILENAME

ED 是 ORACLE 7 的用以打开文件（文件是已经存在的）的命令。当你使用 ED 打开一个文件以后，你就进入了一个全屏幕的编辑环境，它的使用要比在 SQL 提示符下输入 SQL 语句更容易。你可以使用它来修改你的 SPOOL 文件。你将会发现当你在创建 SQL 脚本的时候你会非常频繁地使用它，因为你出于定制的目的你不得不经常地修改脚本的内容。但是，你的大多数定制工作是用 SQL*PLUS 来完成的。

ED FILENAME

计算所有的表中的行数

在第一个例子中我们会向你演示如何去除你的 SPOOL 文件中的不正确的行。这样你的 SQL 脚本在运行的时候就不会返回错误了。

注：请注意在本例中我们所使用的编辑技术，因为我们在之后的例子中将不再进行这一步工作的演示。我们假定到目前为止你已经掌握了基本的 SQL 语法，此外，你也可以使用不同的编辑方法来编辑你的 SPOOL 文件。

现在回想一下对一个表中的所有行进行计数的函数：COUNT (*)。你应该已经知道了如何对一个表中的所有行进行计数。例如：

INPUT:

```
SELECT COUNT(*) FROM TBL1;
```

OUTPUT:

```
COUNT(*)
```

```
29
```

实现的方法很容易，可是如果你想对你的工程中所有属于你的表的行数进行计数呢？

例如：属于你的表如下表所列：

INPUT/OUTPUT:

```
SELECT * FROM CAT;
```

TABLE_NAME	TABLE_TYPE
ACCT_PAY	TABLE
ACCT_REC	TABLE
CUSTOMERS	TABLE
EMPLOYEES	TABLE
HISTORY	TABLE
INVOICES	TABLE
ORDERS	TABLE
PRODUCTS	TABLE
PROJECTS	TABLE
VENDORS	TABLE

10 rows selected.

分析:

如果你想知道你的表有几行，你可能会对每一个表使用 COUNT (*) 函数，FEEDBACK 的结果如下:

10 rows selected

下边的 SELECT 语句将会生成更多的 SELECT 语句来得到上表中的每一个表的行数。

INPUT/OUTPUT:

```
SQL> SET ECHO OFF
```

```
SQL> SET FEEDBACK OFF
```

```
SQL> SET HEADING OFF
```

```
SQL> SPOOL CNT.SQL
```

```
SQL> SELECT 'SELECT COUNT(*) FROM ' || TABLE_NAME || ';
```

```
2 FROM CAT
```

```
3 /
```

```
SELECT COUNT(*) FROM ACCT_PAY;
```

```
SELECT COUNT(*) FROM ACCT_REC;
```

```
SELECT COUNT(*) FROM CUSTOMERS;
```

```
SELECT COUNT(*) FROM EMPLOYEES;
```

```
SELECT COUNT(*) FROM HISTORY;
```

```
SELECT COUNT(*) FROM INVOICES;
```

```
SELECT COUNT(*) FROM ORDERS;
SELECT COUNT(*) FROM PRODUCTS;
SELECT COUNT(*) FROM PROJECTS;
select count(*) FROM VENDORS;
```

分析:

在上例中的第一个动作使用了 SQL*PLUS 中的命令，设置 echo off、feedback off 和 heading off 是将实际的输出结果加以简化。然后使用 SPOOL 命令来把输出的结果重定位到一个叫 CNT.SQL 的文件中。最后的工作是执行这个 SELECT 语句，它将会生成其它形式的 SQL 语句。注意在需要生成的 SELECT 语句上要使用单引号，成对的单引号和联接符可以让你把实际的数据和字符串结合在一起成为一个新的 SQL 语句。本例中从数据字典中选择了数据。命令 SPOOL OFF 则关闭了输出文件。

技巧：在你创建了文件并把你的文件提交运行之前一定把编辑它以使它的语法更加符合你的要求。

INPUT:

```
SQL> SPOOL OFF
SQL> ED CNT.SQL
```

OUTPUT:

```
SQL> SELECT 'SELECT COUNT(*) FROM '||TABLE_NAME||',' FROM CAT;
SELECT COUNT(*) FROM ACCT_PAY;
SELECT COUNT(*) FROM ACCT_REC;
SELECT COUNT(*) FROM CUSTOMERS;
SELECT COUNT(*) FROM EMPLOYEES;
SELECT COUNT(*) FROM HISTORY;
SELECT COUNT(*) FROM INVOICES;
SELECT COUNT(*) FROM ORDERS;
SELECT COUNT(*) FROM PRODUCTS;
SELECT COUNT(*) FROM PROJECTS;
SELECT COUNT(*) FROM VENDORS;
SQL> SPOOL OFF
```

分析:

SPOOL OFF 命令关闭了输出的文件，然后我们使用 ED 来编辑这个文件。这时你就会进行你所创建的文件的内部分了。你应该从这个文件中去掉不必要的行，例如已经知道了结果的 SELECT 语句以及在文件末尾的 SPOOL OFF 命令。

你的文件在编辑完之后应该如下所示，注意：其中的每一行都是一个完整的 SQL 语句。

```
SELECT COUNT(*) FROM ACCT_PAY;

SELECT COUNT(*) FROM ACCT_REC;

SELECT COUNT(*) FROM CUSTOMERS;

SELECT COUNT(*) FROM EMPLOYEES;

SELECT COUNT(*) FROM HISTORY;

SELECT COUNT(*) FROM INVOICES;

SELECT COUNT(*) FROM ORDERS;

SELECT COUNT(*) FROM PRODUCTS;

SELECT COUNT(*) FROM PROJECTS;

SELECT COUNT(*) FROM VENDORS;
```

现在，我们来执行这个文件。

INPUT/OUTPUT:

```
SQL> SET ECHO ON

SQL> SET HEADING ON

SQL> START CNT.SQL

SQL> SELECT COUNT(*) FROM ACCT_PAY;

COUNT(*)

7

SQL> SELECT COUNT(*) FROM ACCT_REC;

COUNT(*)

9

SQL> SELECT COUNT(*) FROM CUSTOMERS;

COUNT(*)

5

SQL> SELECT COUNT(*) FROM EMPLOYEES;

COUNT(*)
```

10

SQL> SELECT COUNT(*) FROM HISTORY;

COUNT(*)

26

SQL> SELECT COUNT(*) FROM INVOICES;

COUNT(*)

0

SQL> SELECT COUNT(*) FROM ORDERS;

COUNT(*)

0

SQL> SELECT COUNT(*) FROM PRODUCTS;

COUNT(*)

10

SQL> SELECT COUNT(*) FROM PROJECTS;

COUNT(*)

16

SQL> SELECT COUNT(*) FROM VENDORS;

COUNT(*)

22

SQL>

分析:

Set echo on 可以让你在执行文件的时候看到其中的每一条语句，Set heading on 则可以显示每个 SELECT 语句的列标头 COUNT(*)。如果你还输入了 set feedback on 的话那么

1 row selected

将会出现在每个计数结果的后边。在这个例子中我们使用 SQL*PLUS 中的 START 命令来执行了 SQL 脚本。可是，如果你要处理的表不是 10 个而是 50 个时又会怎样呢？

注：一定要正确地使用单引号以保证生成的 SQL 脚本的正确。你要把你生成的全部语句都用单引号括起来。在本例中，你用单引号括起来的 SQL 语句如, 'SELECT COUNT(*) FROM' 和 ';' 不会从表中选择数据。

为多个用户赋予系统权限

作为一个数据库管理员或有维护其它用户任务的个体，你经常会收到一些用户的请求。然后将不得不将某些权限赋予一些用户以使它们可以进行正常的数据库访问。你也会由于他们工作的变动而相应地改变他们的权限。这可以通过今天的方法来生成 GRANT 语句为多个用户赋予系统权限和规则。

INPUT:

```
SQL> SET ECHO OFF

SQL> SET HEADING OFF

SQL> SET FEEDBACK OFF

SQL> SPOOL GRANTS.SQL

SQL> SELECT 'GRANT CONNECT, RESOURCE TO ' || USERNAME || '
2  FROM SYS.DBA_USERS
3  WHERE USERNAME NOT IN ('SYS','SYSTEM','SCOTT','RYAN','PO7','DEMO')
4  /
```

OUTPUT:

```
GRANT CONNECT, RESOURCE TO KEVIN;
GRANT CONNECT, RESOURCE TO JOHN;
GRANT CONNECT, RESOURCE TO JUDITH;
GRANT CONNECT, RESOURCE TO STEVE;
GRANT CONNECT, RESOURCE TO RON;
GRANT CONNECT, RESOURCE TO MARY;
GRANT CONNECT, RESOURCE TO DEBRA;
GRANT CONNECT, RESOURCE TO CHRIS;
GRANT CONNECT, RESOURCE TO CAROL;
GRANT CONNECT, RESOURCE TO EDWARD;
GRANT CONNECT, RESOURCE TO BRANDON;
GRANT CONNECT, RESOURCE TO JACOB;
```

INPUT/OUTPUT:

```
SQL> spool off
```

```
SQL> start grants.sql

SQL> GRANT CONNECT, RESOURCE TO KEVIN;

Grant succeeded.

SQL> GRANT CONNECT, RESOURCE TO JOHN;

Grant succeeded.

SQL> GRANT CONNECT, RESOURCE TO JUDITH;

Grant succeeded.

SQL> GRANT CONNECT, RESOURCE TO STEVE;

Grant succeeded.

SQL> GRANT CONNECT, RESOURCE TO RON;

Grant succeeded.

SQL> GRANT CONNECT, RESOURCE TO MARY;

Grant succeeded.

SQL> GRANT CONNECT, RESOURCE TO DEBRA;

Grant succeeded.

SQL> GRANT CONNECT, RESOURCE TO CHRIS;

Grant succeeded.

SQL> GRANT CONNECT, RESOURCE TO CAROL;

Grant succeeded.

SQL> GRANT CONNECT, RESOURCE TO EDWARD;

Grant succeeded.

SQL> GRANT CONNECT, RESOURCE TO BRANDON;

Grant succeeded.

SQL> GRANT CONNECT, RESOURCE TO JACOB;

Grant succeeded.
```

分析:

除非你愿意手工地一条一条地去输入，否则在本例中你通过使用一个 SQL 语句来生成多个 GRANT 语句从而避免了大量的枯燥的击键工作。

注：下边的例子将省略掉你编辑输出文件的工作，你可以认为该文件已经编辑处理过了。

将你的表的权限赋予其它的用户

将一个表的权限赋予其它的用户就像从表中选择一行一样地简单，可是，如果你有多个表需要对它们赋予访问规则和用户时，你可以使用 SQL 来为你生成一个脚本——除非你喜欢打字。

首先，回顾一下为单个表赋予权限：

INPUT：

```
SQL> GRANT SELECT ON HISTORY TO BRANDON;
```

OUTPUT：

```
Grant succeeded.
```

你准备好了吗？下边的语句将会创建一组 GRANT 语句来为你的计划中的 10 个表赋予权限。

INPUT/OUTPUT：

```
SQL> SET ECHO OFF
```

```
SQL> SET FEEDBACK OFF
```

```
SQL> SET HEADING OFF
```

```
SQL> SPOOL GRANTS.SQL
```

```
SQL> SELECT 'GRANT SELECT ON ' || TABLE_NAME || ' TO BRANDON;'
```

```
2 FROM CAT
```

```
3 /
```

```
GRANT SELECT ON ACCT_PAY TO BRANDON;
```

```
GRANT SELECT ON ACCT_REC TO BRANDON;
```

```
GRANT SELECT ON CUSTOMERS TO BRANDON;
```

```
GRANT SELECT ON EMPLOYEES TO BRANDON;
```

```
GRANT SELECT ON HISTORY TO BRANDON;
```

```
GRANT SELECT ON INVOICES TO BRANDON;
```

```
GRANT SELECT ON ORDERS TO BRANDON;
```

```
GRANT SELECT ON PRODUCTS TO BRANDON;
```

```
GRANT SELECT ON PROJECTS TO BRANDON;
```

```
GRANT SELECT ON VENDORS TO BRANDON;
```

分析：

GRANT 语句已经自动地为每一个表生成了，BRANDON 现在可以访问你的每一个表了。

现在我们使用 SPOOL 命令来关闭输出文件，并假定该文件已经被编辑过了，该文件已经为运行做好了准备。

INPUT/OUTPUT：

```
SQL> SPOOL OFF
```

```
SQL> SET ECHO ON
```

```
SQL> SET FEEDBACK ON
```

```
SQL> START GRANTS.SQL
```

```
SQL> GRANT SELECT ON ACCT_PAY TO BRANDON;
```

Grant succeeded.

```
SQL> GRANT SELECT ON ACCT_REC TO BRANDON;
```

Grant succeeded.

```
SQL> GRANT SELECT ON CUSTOMERS TO BRANDON;
```

Grant succeeded.

```
SQL> GRANT SELECT ON EMPLOYEES TO BRANDON;
```

Grant succeeded.

```
SQL> GRANT SELECT ON HISTORY TO BRANDON;
```

Grant succeeded.

```
SQL> GRANT SELECT ON INVOICES TO BRANDON;
```

Grant succeeded.

```
SQL> GRANT SELECT ON ORDERS TO BRANDON;
```

Grant succeeded.

```
SQL> GRANT SELECT ON PRODUCTS TO BRANDON;
```

Grant succeeded.

```
SQL> GRANT SELECT ON PROJECTS TO BRANDON;
```

Grant succeeded.

```
SQL> GRANT SELECT ON VENDORS TO BRANDON;
```

Grant succeeded.

分析：

将 ECHO 和 FEEDBACK 设置为 ON 是很好的。设置了 FEEDBACK 我们就可以在执行的时候看到 Grant succeeded。没用多少力气，我们就为 BRANDON 赋予了 10 个表的访问权限。而且，你所做的工作可能会不只是 10 个表。

在载入数据时解除对数的约束

当你向表中载入数据的时候，有时你会不得不解除对表的约束，假定你的表已经被删减过或有损坏了，更有可能的是你的表存有如外部关键字之类的引用完整性约束。这时数据库不会允许你向表中插入不在其它表中存在相关关系的数据（如果引用列在其它的表中不存在的话）。你在最初装入数据的时候你可能会不得不解除对表的约束。当然，当载入成功以后，你应该将这些约束恢复。

INPUT：

```
SQL> SET ECHO OFF

SQL> SET FEEDBACK OFF

SQL> SET HEADING OFF

SQL> SPOOL DISABLE.SQL

SQL> SELECT 'ALTER TABLE ' || TABLE_NAME ||
2      'DISABLE CONSTRAINT ' || CONSTRAINT_NAME || ';'
3 FROM SYS.DBA_CONSTRAINTS
4 WHERE OWNER = 'RYAN'
5 /
```

OUTPUT：

```
ALTER TABLE ACCT_PAY DISABLE CONSTRAINT FK_ACCT_ID;
ALTER TABLE ACCT_REC DISABLE CONSTRAINT FK_ACCT_ID;
ALTER TABLE CUSTOMERS DISABLE CONSTRAINT FK_CUSTOMER_ID;
ALTER TABLE HISTORY DISABLE CONSTRAINT FK_ACCT_ID;
ALTER TABLE INVOICES DISABLE CONSTRAINT FK_ACCT_ID;
ALTER TABLE ORDERS DISABLE CONSTRAINT FK_ACCT_ID;
```

分析：

这个对象已经生成了一系列的 ALTER TABLE 语句来解除所有为 RYAN 所拥有的表，在连接符末尾的分号是为了保证每个语句的完整。

INPUT/OUTPUT:

```
SQL> SPOOL OFF

SQL> SET ECHO OFF

SQL> SET FEEDBACK ON

SQL> START DISABLE.SQL

Constraint Disabled.

Constraint Disabled.

Constraint Disabled.

Constraint Disabled.

Constraint Disabled.

Constraint Disabled.
```

分析:

注意在这里 ECHO 被设置为 OFF，也就是说你在执行不会看到对应的语句，但是由于 FEEDBACK 设置为 ON，所以你可以看到结果。

```
Constraint Disabled.
```

如果 ECHO 和 FEEDBACK 都设置成 OFF 了，在执行时将什么都不会显示，只是经过一段时间的暂停以后又回到了 SQL> 的提示符下。

现在你可以放心地载入你的数据而不必担心因为约束而导致的错误了。约束是好事，但它在数据载入的时候会造成障碍。你也可以使用相同的思想来恢复对表的约束。

一次创建多个同义字

另外一项令人烦躁和费力的工作是创建多个同义字，不管是公共的还是私有的。只有 DBA 可以创建共同的同义字，但是任何用户都可以创建私有的同义字。

下边的例子是对所有为 RYAN 所拥有的表创建公共同义字。

INPUT:

```
SQL> SET ECHO OFF

SQL> SET FEEDBACK OFF
```

```
SQL> SET HEADING OFF

SQL> SPOOL PUB_SYN.SQL

SQL> SELECT 'CREATE PUBLIC SYNONYM ' || TABLE_NAME || ' FOR ' ||
2         OWNER || '.' || TABLE_NAME || ';'
3 FROM SYS.DBA_TABLES
4 WHERE OWNER = 'RYAN'
5 /
```

OUTPUT:

```
CREATE PUBLIC SYNONYM ACCT_PAY FOR RYAN.ACCT_PAY;
CREATE PUBLIC SYNONYM ACCT_REC FOR RYAN.ACCT_REC;
CREATE PUBLIC SYNONYM CUSTOMERS FOR RYAN.CUSTOMERS;
CREATE PUBLIC SYNONYM EMPLOYEES FOR RYAN.EMPLOYEES;
CREATE PUBLIC SYNONYM HISTORY FOR RYAN.HISTORY;
CREATE PUBLIC SYNONYM INVOICES FOR RYAN.INVOICES;
CREATE PUBLIC SYNONYM ORDERS FOR RYAN.ORDERS;
CREATE PUBLIC SYNONYM PRODUCTS FOR RYAN.PRODUCTS;
CREATE PUBLIC SYNONYM PROJECTS FOR RYAN.PROJECTS;
CREATE PUBLIC SYNONYM VENDORS FOR RYAN.VENDORS;
```

现在，运行这个文件

INPUT/OUTPUT:

```
SQL> SPOOL OFF

SQL> ED PUB_SYN.SQL

SQL> SET ECHO ON

SQL> SET FEEDBACK ON

SQL> START PUB_SYN.SQL

SQL> CREATE PUBLIC SYNONYM ACCT_PAY FOR RYAN.ACCT_PAY;

Synonym created.

SQL> CREATE PUBLIC SYNONYM ACCT_REC FOR RYAN.ACCT_REC;

Synonym created.

SQL> CREATE PUBLIC SYNONYM CUSTOMERS FOR RYAN.CUSTOMERS;
```

Synonym created.

```
SQL> CREATE PUBLIC SYNONYM EMPLOYEES FOR RYAN.EMPLOYEES;
```

Synonym created.

```
SQL> CREATE PUBLIC SYNONYM HISTORY FOR RYAN.HISTORY;
```

Synonym created.

```
SQL> CREATE PUBLIC SYNONYM INVOICES FOR RYAN.INVOICES;
```

Synonym created.

```
SQL> CREATE PUBLIC SYNONYM ORDERS FOR RYAN.ORDERS;
```

Synonym created.

```
SQL> CREATE PUBLIC SYNONYM PRODUCTS FOR RYAN.PRODUCTS;
```

Synonym created.

```
SQL> CREATE PUBLIC SYNONYM PROJECTS FOR RYAN.PROJECTS;
```

Synonym created.

```
SQL> CREATE PUBLIC SYNONYM VENDORS FOR RYAN.VENDORS;
```

Synonym created

分析:

几乎是马上，所有的数据库用户都可以通过公共同义字来访问为 RYAN 所拥有的表了。现在用户无需在执行查询的时候对表加以限制（限制的意思就是必须指定表的所有者，如 RYAN.VENDORS）

但是如果公共同义字并不存在呢？假如 BRANDON 想访问所有为 RYAN 所拥有的表来创建私有同义字时该如何做呢？

INPUT/OUTPUT:

```
SQL> CONNECT BRANDON
```

```
ENTER PASSWORD: *****
```

```
CONNECTED.
```

```
SQL> SET ECHO OFF
```

```
SQL> SET FEEDBACK OFF
```

```
SQL> SET HEADING OFF
```

```
SQL> SPOOL PRIV_SYN.SQL
```

```
SQL> SELECT 'CREATE SYNONYM ' || TABLE_NAME || ' FOR ' ||
```



```
2      OWNER || '.' || TABLE_NAME || ';
```

```
3 FROM ALL_TABLES
```

```
4 /
```

```
CREATE SYNONYM DUAL FOR SYS.DUAL;
```

```
CREATE SYNONYM AUDIT_ACTIONS FOR SYS.AUDIT_ACTIONS;
```

```
CREATE SYNONYM USER_PROFILE FOR SYSTEM.USER_PROFILE;
```

```
CREATE SYNONYM CUSTOMERS FOR RYAN.CUSTOMERS;
```

```
CREATE SYNONYM ORDERS FOR RYAN.ORDERS;
```

```
CREATE SYNONYM PRODUCTS FOR RYAN.PRODUCTS;
```

```
CREATE SYNONYM INVOICES FOR RYAN.INVOICES;
```

```
CREATE SYNONYM ACCT_REC FOR RYAN.ACCT_REC;
```

```
CREATE SYNONYM ACCT_PAY FOR RYAN.ACCT_PAY;
```

```
CREATE SYNONYM VENDORS FOR RYAN.VENDORS;
```

```
CREATE SYNONYM EMPLOYEES FOR RYAN.EMPLOYEES;
```

```
CREATE SYNONYM PROJECTS FOR RYAN.PROJECTS;
```

```
CREATE SYNONYM HISTORY FOR RYAN.HISTORY;
```

INPUT/OUTPUT:

```
SQL> SPOOL OFF
```

```
SQL>
```

```
SQL> SET ECHO OFF
```

```
SQL> SET FEEDBACK ON
```

```
SQL> START PRIV_SYN.SQL
```

```
Synonym created.
```

```
Synonym created.
```

```
Synonym created.
```

```
Synonym created.
```

```
Synonym created.
```

```
Synonym created.
```

```
Synonym created.
```

```
Synonym created.
```

Synonym created.

Synonym created.

Synonym created.

Synonym created.

Synonym created.

分析：

几乎没做什么工作，BRANDON 就有了所有为 RYAN 所拥有的表的同义字，他不再需要对表进行限制了。

为你的表创建视图

如果你想为一组表创建视图，你需要做与下列类似的工作：

INPUT：

```
SQL> SET ECHO OFF
```

```
SQL> SET FEEDBACK OFF
```

```
SQL> SET HEADING OFF
```

```
SQL> SPOOL VIEWS.SQL
```

```
SQL> SELECT 'CREATE VIEW ' || TABLE_NAME || '_VIEW AS SELECT * FROM ' ||
```

```
2      TABLE_NAME || ';
```

```
3 FROM CAT
```

```
4 /
```

OUTPUT：

```
CREATE VIEW ACCT_PAY_VIEW AS SELECT * FROM ACCT_PAY;
```

```
CREATE VIEW ACCT_REC_VIEW AS SELECT * FROM ACCT_REC;
```

```
CREATE VIEW CUSTOMERS_VIEW AS SELECT * FROM CUSTOMERS;
```

```
CREATE VIEW EMPLOYEES_VIEW AS SELECT * FROM EMPLOYEES;
```

```
CREATE VIEW HISTORY_VIEW AS SELECT * FROM HISTORY;
```

```
CREATE VIEW INVOICES_VIEW AS SELECT * FROM INVOICES;
```

```
CREATE VIEW ORDERS_VIEW AS SELECT * FROM ORDERS;
```

```
CREATE VIEW PRODUCTS_VIEW AS SELECT * FROM PRODUCTS;
```

```
CREATE VIEW PROJECTS_VIEW AS SELECT * FROM PROJECTS;
```

```
CREATE VIEW VENDORS_VIEW AS SELECT * FROM VENDORS;
```

INPUT/OUTPUT:

```
SQL> SPOOL OFF
```

```
SQL> SET ECHO OFF
```

```
SQL> SET FEEDBACK ON
```

```
SQL> START VIEWS.SQL
```

View Created.

View Created.

View Created.

View Created.

View Created.

View Created.

View Created.

View Created.

View Created.

View Created.

分析:

在上边的 SQL 语句中生成了一个叫 VIEW.SQL 的文件。输出文件变成了一个 SQL 文件，其中包括着对指定的表所创建的视图。在运行这个文件之后，我们可以看到视图已经创建了。

在一个计划中清除其所有的表的内容

清除表是在开发环境中出现的工作，为了有效地开发测试数据载入和 SQL 的性能。数据是经常需要重新载入的。这一过程可以确定和清除 BUG。经常测试后的应用程序才会转入生产环境中。

下边的例子将清除一个工作中的所有的表的内容:

INPUT:

```
SQL> SET ECHO OFF
```

```
SQL> SET FEEDBACK OFF

SQL> SET HEADING OFF

SQL> SPOOL TRUNC.SQL

SQL> SELECT 'TRUNCATE TABLE ' || TABLE_NAME || ';'
      2  FROM ALL_TABLES
      3  WHERE OWNER = 'RYAN'
      4  /
```

OUTPUT:

```
TRUNCATE TABLE ACCT_PAY;
TRUNCATE TABLE ACCT_REC;
TRUNCATE TABLE CUSTOMERS;
TRUNCATE TABLE EMPLOYEES;
TRUNCATE TABLE HISTORY;
TRUNCATE TABLE INVOICES;
TRUNCATE TABLE ORDERS;
TRUNCATE TABLE PRODUCTS;
TRUNCATE TABLE PROJECTS;
TRUNCATE TABLE VENDORS;

如果你敢的话你就运行一下这个脚本。
```

INPUT/OUTPUT:

```
SQL> SPOOL OFF

SQL> SET FEEDBACK ON

SQL> START TRUNC.SQL

Table Truncated.

Table Truncated.

Table Truncated.

Table Truncated.

Table Truncated.

Table Truncated.

Table Truncated.
```

Table Truncated.

Table Truncated.

Table Truncated.

分析：

你已经把所有的为 RYAN 所属的表的内容清除了，这很容易。如果你准备重新构建你的表数据时你可以使用这项技术。

技巧：当在一个计划中运行诸如清除表之类的操作时，你必须对你将要清理的表做一个很好的备份。即便你肯定你再也不需要它的时候也要这样做（是后也许会有人坚持要求你恢复原来的数据）。

使用 SQL 来生成 SHELL 脚本

你也可以使用 SQL 来生成其它形式的脚本，比如说 SHELL 脚本。例如：ORACLE RDBMS 服务可能会运行于 UNIX 环境下，这是比 PC 环境更大的代表。所以，UNIX 需要对文件进行更为有效的管理。通过创建脚本你可以很容易地管理数据库的文件。

下边的工作是删除在一个数据库中的表空间，尽管可以使用 SQL 来删除表空间，但是与表空间相关的事实的数据文件却必须在操作系统中才能删除。

下一步是生成一个 SQL 脚本来删除表空间：

INPUT：

```
SQL> SET ECHO OFF
```

```
SQL> SET FEEDBACK OFF
```

```
SQL> SET HEADING OFF
```

```
SQL> SPOOL DROP_TS.SQL
```

```
SQL> SELECT 'DROP TABLESPACE ' || TABLESPACE_NAME || ' INCLUDING
CONTENTS;'
```

```
2 FROM SYS.DBA_TABLESPACES
```

```
3 /
```

OUTPUT：

```
DROP TABLESPACE SYSTEM INCLUDING CONTENTS;
```

```
DROP TABLESPACE RBS INCLUDING CONTENTS;
```

```
DROP TABLESPACE TEMP INCLUDING CONTENTS;
```

```
DROP TABLESPACE TOOLS INCLUDING CONTENTS;
```

```
DROP TABLESPACE USERS INCLUDING CONTENTS;
```

然后你需要生成一个脚本文件以在表空间删除后在操作系统中删除数据库文件。

INPUT/OUTPUT:

```
SQL> SPOOL OFF
```

```
SQL> SPOOL RM_FILES.SH
```

```
SQL> SELECT 'RM -F ' || FILE_NAME
```

```
2 FROM SYS.DBA_DATA_FILES
```

```
3 /
```

```
rm -f /disk01/orasys/db01/system0.dbf
```

```
rm -f /disk02/orasys/db01/rbs0.dbf
```

```
rm -f /disk03/orasys/db01/temp0.dbf
```

```
rm -f /disk04/orasys/db01/tools0.dbf
```

```
rm -f /disk05/orasys/db01/users0.dbf
```

```
SQL> spool off
```

```
SQL>
```

分析:

现在你已经生成了两个脚本，你可以运行脚本来删除表空间，然后在操作系统中运行 SHELL 脚本来删除相关的数据文件，你会发现有很多种方法来用 SQL 生成和管理非 SQL 的脚本。

再建表和索引

尽管有许多工具可以帮助你再建表和索引，然而对于这目的你可以直接使用 SQL 来完成。你可以从数据字典中获得所有的你需要重建的表和索引的信息。但是要有效地完成这个工作没有过程语言（如 PL/SQL 或 SHELL 脚本）的支持是不行的。

我们可以在一个 SHELL 脚本中使用内嵌的 SQL，过程语言函数中需要插入适当的语法成份，比如逗号。脚本必须足够地聪明以知道哪里是最后一个逗号，因为在最后一个逗号的后边就没有逗号了。脚本也必须知道在哪些地方需要放置括号。在用数据字典重新生

成对象的时候 SEEK 工具是很有用的。不管你是使用 C、Perl、shell scripts、COBOL 还是 PL/SQL。

总结

直接地从数据库中生成语句可以避免许多单调乏味的输入 SQL 语句的工作。不管你做什么工作，使用 SQL 生成技术都可以在你的某个工作阶段减少你的工作量。

如果你已经学习了今天的基本内容——尽管这些例子是使用 ORACLE 来完成的，你可以在其它的关系数据库系统中应用这一概念，请检查你的数据库解释器和数据字典的结构以找出它们与本书中例子的不同之处。如果你开明的话，你会发现许多生成 SQL 脚本的方法，从简单的语句到高度复杂的系统管理中都有。

问与答

问：我如何才能决定是手工写 SQL 还是来生成它？

答：问你自己下边的问题

你解决问题时使用该语句的频繁程度如何？

在写母语句是否比写子语句耗用更多的时间？

问：生成 SQL 语句时我有哪些表可供选择？

答：你可以选择任何一个你可以访问的表，这不管这些表是否为你所有或它是否在数据字典中，你可以选择所有在你数据库中的有效对象，比如视图或快捷方式。

问：我在生成 SQL 语句是是否有限制？

答：对于大多数的语句你既可以手工写出也可以生成。请检查你的解释器看一下看它是否有输出到文件选项以及输出格式是否是你所需要的。切记，在生成输出文件后你应该手工修改一下文件的内容。

校练场

1、你生成 SQL 的来源有哪两个？

2、下边的 SQL 语句是否可以工作？它会输出什么？

```
SQL> SET ECHO OFF
```

```
SQL> SET FEEDBACK OFF
```

```
SQL> SPOOL CNT.SQL
```

```
SQL> SELECT 'COUNT(*) FROM ' || TABLE_NAME || '
```

```
2 FROM CAT
```

```
3 /
```

3、下边的 SQL 语句是否可以工作？它会输出什么？

```
SQL> SET ECHO OFF
```

```
SQL> SET FEEDBACK OFF
```

```
SQL> SPOOL GRANT.SQL
```

```
SQL> SELECT 'GRANT CONNECT DBA TO ' || USERNAME || '
```

```
2 FROM SYS.DBA_USERS
```

```
3 WHERE USERNAME NOT IN ('SYS','SYSTEM','SCOTT')
```

```
4 /
```

4、下边的 SQL 语句是否可以工作？它会输出什么？

```
SQL> SET ECHO OFF
```

```
SQL> SET FEEDBACK OFF
```

```
SQL> SELECT 'GRANT CONNECT, DBA TO ' || USERNAME || '
```

```
2 FROM SYS.DBA_USERS
```

```
3 WHERE USERNAME NOT IN ('SYS','SYSTEM','SCOTT')
```

```
4 /
```

5、在运行生成的 SQL 时最为将 FEEDBACK 设置为 ON，对不对？

6、从 SQL 中生成 SQL 语句时，必须将输入的结果重新定向到一个文件中，对不对？

7、在生成 SQL 语句用以对表的内容进行删减时，你必须先确认自己已经对所删减的表作了很好的备份。对不对？

8、什么是 ED 命令？

9、SPOOL OFF 命令是做什么的？

练习

1、使用 SYS.DBA_USERS 视图（在 PERSONAL ORACLE 7 中）写一个语句来生成一

系列 GRANT 语句为下边的五个用户：John、Kevin、Ryan、Ron 和 Chris 授权，使他们可以访问 History_tbl 表，在写语句时对应的用户名用 USERNAME 来代替。

2、用本章给出的例子作指引，写一些 SQL 来创建一些可以生成你能使用的 SQL 语句。

第 18 天：PL/SQL 简介

目标

PL/SQL 是一项 ORACLE 的技术，它可以让 SQL 像过程型语言一样工作，到今天的结束，你将：

- 对 PL/SQL 有一个基本的了解。
- 明白 PL/SQL 的特点以及它与标准的 SQL 的区别。
- 知道 PL/SQL 编写程序的基本原理。
- 可以写简单的 PL/SQL 程序。
- 知道如何处理在 PL/SQL 编程中的错误。
- 知道如何在真实的世界中使用 PL/SQL。

入门

从对标准的 SQL 进行介绍是一种入门 PL/SQL（或 SQL）的方法，SQL 是一种可以让关系数据库用户以一种直接的方式来访问数据库的方法。你可以使用 SQL 来对数据库进行查询和修改其中的表。在你写 SQL 语句的时候，你可以告诉数据库你要做什么，而不是如何去做。优化器会自己决定采用一种最有效的方法来执行你的语句，如果你向数据库中发送了一系列标准的 SQL 语句，服务将会按语句的先后次序来执行它们。

PL/SQL 是 ORACLE 的过程型语言，它由标准的 SQL 语句和一系列可以让你在不同的情况下对 SQL 语句的执行进行控制的命令组成。PL/SQL 也可以在运行时捕获错误。诸如 LOOP 和 IF.....THEN.....ELSE 语句让 PL/SQL 具有了第三代编程语言的能力。PL/SQL 也可以以交互方式写出，用户友好型界面会把数值赋给它的变量。你可以使用许多种预定义的包，它们可以将信息显示给用户。

今天的内容主要包括 PL/SQL 的以下特性：

- 程序员可以定义在语句中使用的变量。
- 程序员可以使用错误控制例程来防止程序的意外中断和退出。
- 程序员可以写交互式程序来接受来自用户的输入。

- 程序员可以将功能划分为不同的逻辑代码块，编程的模块化为应用程序开发环境提供了更大的灵活性。
- SQL 语句可以并行执行以获得最优的性能。

在 PL/SQL 中的数据类型

大多数数据的类型是相似的，但是每一种解释器都有自己独特的存储方式和内部过程需要。在写 PL/SQL 的语句块时，你将会定义变量，它们必须是真实的数据类型，下边的小标题对 PL/SQL 中可用的变量进行了详细的介绍。

在 PL/SQL 中也提供了数据类型的子类型。例如：数据类型 NUMBER 的子类型叫 INTEGER，你可以在 PL/SQL 程序中使用子类型来保证它与其它程序如 COBOL 中数据类型的一致，尤其是在其它的程序中内嵌的 PL/SQL 代码时更要如此。子类型只是 ORACLE 中数据类型的别名，所以它必须遵循与之相关联的数据类型的规则。

注：在大多数的 SQL 解释器中语法的大小写是不敏感的，PL/SQL 同样允许在它的语句中使用大小写（它也是大小写不敏感的）。

字符串类型

在 PL/SQL 中的字符串类型所你所想的一样，是一种常见的数据类型定义，它允许在其中有字母和数字。像名字，代码，描述，序列号等都可以包括在字符串中。

CHAR 中存储着固定长度的字符串，CHAR 在最大长度为 32767 个字节，尽管很难想象在哪一个表中的字符串会有这么长。

语法：

CHAR (max_length)

子类型： CHARACTER

VARCHAR2 则存储着长度可变的字符串，你会经常用 VARCHAR2 来存储长度可变的字符串，比如某人的名字，VARCHAR2 所允许的最大长度为 32767 个字节。

语法：

VARCHAR2 (max_length)

子类型： VARCHAR, STRING

LONG 也可以存储变长的字符串，它的最大长度为 32760 个字节，LONG 是典型的用以存储长文本如备注，尽管 VARCHAR2 也能做同样的工作。

数值数据类型

NUMBER 用以存储在 ORACLE 数据库中的任何类型的数值。

语法：

NUMBER (max_length)

你也可以使用下边的语法来指定数据类型的精度。

NUMBER (precision, scale)

子类型：DEC、DECIMAL、DOUBLE PRECISION、INTEGER、INT、NUMERIC、REAL、SMALLINT、FLOAT

PLS_INTEGER 定义的整数是可以带有符号的，例如：负数。

二进制数据类型

二进制数据类型可以以二进制形式来存储数据，如图形或图像，这种数据类型包括 RAW 和 LONGRAW。

日期数据类型

DATE 是在 ORACLE 中可以有效存储的数据类型，如果你将某一列定义为 DATE 类型，你就不能指定它的长度，DATE 数据类型的长度是默认的，ORACLE 数据类型是像 01-OCT-97 这样的。

逻辑数据类型

逻辑数据类型可以存储下列数值：TRUE、FALSE 和 NULL。与 DATE 相类似，BOOLEAN 在作为列或变量的类型在定义时也不需要参数。

ROWID

ROWID 是存在于 ORACLE 数据库的每一个表中的预定义列。ROWID 以二进制格式存储和确定表中的每一列，索引就是使用 ROWID 指向数据的。

PL/SQL 块的结构

PL/SQL 是一种块结构语言，也就是说 PL/SQL 的程序可以分成逻辑块来写。在一个块的内部可以有像数据操作或查询之类的过程。下边将会对 PL/SQL 的块进行详细的讨论。

- 在 DECLARE 部分包括了定义的变量和它的对象，如常量和指针，这一部分在 PL/SQL 块中是可以选择的。
- PROCEDURE 部分包括条件语句和 SQL 语句，块可以对它进行控制。它是 PL/SQL 的必须部分。
- EXCEPTION 告诉了 PL/SQL 如何处理指定的错误并按用户的定义进行处理。它也是 PL/SQL 的可选择部分。

注：块是 PL/SQL 代码的逻辑单元，包括至少一个 PROCEDURE 部分和可以选择的 DECLARE 以及 EXCEPTION 部分。

这里是 PL/SQL 块的基本结构：

SYNTAX:

```
BEGIN          -- optional, denotes beginning of block

  DECLARE      -- optional, variable definitions

  BEGIN        -- mandatory, denotes beginning of procedure section

  EXCEPTION    -- optional, denotes beginning of exception section

  END          -- mandatory, denotes ending of procedure section

END            -- optional, denotes ending of block
```

请注意必要部分在第二个 BEGIN——END 对中的，它构成了一个过程部分。当然，在它们中间是有语句的。如果你使用了第一个 BEGIN，那么你必须同时使用第二个 BEGIN，反之亦然。

注释

如果一个程序没有注释会怎样？编程语言提供了命令使你可以在你的代码中放放注释。PL/SQL 当然不例外。你可以在需要注释的行前加一短线，公认的 PL/SQL 注释形式如下（多行注释则与 C 语言的注释方法相同）：

语法：

```
-- This is a one-line comment.
```

```
/* This is a
```

```
multiple-line comment.*/
```

注：PL/SQL 可以直接支持数据操作言语（DML）命令和数据查询。然而，它不支持数据字典语言（DDL）命令。你通常会使用 PL/SQL 维护在数据库结构中的数据。但是并不能维护这些结构。

DECLARE 部分

在 PL/SQL 的 DECLARE 部分包括了变量、常量、指针和特殊数据类型的定义。作为一个 PL/SQL 程序员，你可以在你的代码块中定义所有类型的变量。但是你必须指定数据类型，而且每一个变量都必须与 ORACLE 中所定义的数据类型一致。变量也要符合 ORACLE 的对象命名标准。

变量声明

在 PL/SQL 语句块中变量的数据是可变以。在变量声明时必须对它加以定义，如有必须，还要进行初始化工作。下例中定义了一组在 PL/SQL 的 DECLARE 部分的变量。

```
DECLARE
```

```
    owner char(10);
```

```
    tablename char(30);
```

```
    bytes number(10);
```

```
    today date
```

分析：

DECLARE 部分不能自动运行。DECLARE 部分以 DECLARE 语句开头，每一个变量

占一行，注意在每一个定义的变量后边都有一个分号。

在 DECLARE 部分也可以对变量进行初始化工作，例如：

```
DECLARE

customer char(30);

fiscal_year number(2) := '97';
```

你可以使用：=符号来进行初始化工作。或者说是赋给它一个初值。对于被定义为 NOT NULL 的变量你必须赋初值。

```
DECLARE

customer char(30);

fiscal_year number(2) NOT NULL := '97';
```

分析：

NOT NULL 子句在这里的作用与它在 CREATE TABLE 中所起的作用类似。

常量定义

定义常量的方法与定义变量相同，但是常量的数值是静态的。他们不能改变。在上个例子中，fiscal_year 可能是常量。

注：在每一个变量的定义都必须以分号结束。

指针定义

指针是 PL/SQL 中的另一种类型的变量。在你通常所认为的变量中保存的是数值。而指针型变量则指向了查询结果中的某一行数据。在查询的结果有多个时，为了分析数据你需要在每个记录之间进行翻阅。在查看 PL/SQL 块中的查询结果时，它使用指针来指向返回的每一行。下边是一个在 PL/SQL 语句块中定义指针的例子。

INPUT：

```
DECLARE

cursor employee_cursor is

select * from employees;
```

指针与视图类似，通过在 PROCEDURE 部分使用 LOOP（循环）。你可以翻阅指针。这

项技术曾简要地提到过。

%TYPE 属性

%TYPE 可以返回表中给定列的变量属性，除了查看在 PL/SQL 中的数据类型定义代码，你可以使用 %TYPE 来保持在你的块中的代码的一致性。

INPUT:

```
DECLARE

    cursor employee_cursor is

        select emp_id, emp_name from employees;

    id_num employees.emp_id%TYPE;

    name employees.emp_name%TYPE;
```

分析:

在这个例子中表中 ID_NUM 变量与 EMP_ID 变量具有相同的数据类型。%TYPE 所定义的变量 NAME 具有与 EMPLOYEES 中的 emp_name 具有相同的数据类型。

%ROWTYPE 属性

变量不仅限于单一的数值。如果你所定义的变量与一个指针相关联的话，你可以使用 %ROWTYPE 属性来声明变量与保证它与游标所在行的类型相同，在 ORACLE 的词典中 %ROWTYPE

INPUT:

```
DECLARE

    cursor employee_cursor is

        select emp_id, emp_name from employees;

    employee_record employee_cursor%ROWTYPE;
```

分析:

在上例中定义了一个叫 employee_record 的变量，%ROWTYPE 定义了这个变量的使它 与 employee_cursor 所在行的数据类型相同，这个 %ROWTYPE 属性定义的变量也称为集合变量。

%ROWCOUNT 属性

在 PL/SQL 中 %ROWCOUNT 属性可以保证在特定的 SQL 语句块中的游标行数。

INPUT:

DECLARE

```
cursor employee_cursor is  
  
    select emp_id, emp_name from employees;  
  
records_processed := employee_cursor%ROWCOUNT;
```

分析:

在上例中变量 records_processed 将会返回 PL/SQL 语句所访问的 employee_cursor 的行数。

警告：在定义变量的时候要小心以防止它和表的名字相冲突。例如：如果你定义的变量与你的 PL/SQL 语句块中所访问的表的名字是相同的，那么变量的名字会优先于表的名字。

Procdure 部分

PROCEDURE 部分是 PL/SQL 语句块中的必须部分，在这一部分将会使用变量和用户指针来操作数据库中的数据。PROCEDURE 部分是一个块的主要部分，它包括条件语句和 SQL 语句。

BEGIN……END

在一个语句块中，BEGIN 标明了过程的开始。与此类似，END 则标明了语句块的结束。下边的例子显示了一个 PROCEDURE 部分的基本结构。

语法:

```
BEGIN  
  
    open a cursor;  
  
    condition1;  
  
        statement1;  
  
    condition2;  
  
        statement2;
```

```
        condition3;

        statement3;

        .....

    close the cursor;

END
```

指针控制命令

现在你将学习如何在 PL/SQL 的语句块中定义一个指针，你需要知道如何来访问一个定义过的指针。这一部分诠释了基本的指针控制命令：DECLARE、OPEN、FETCH 和 CLOSE。

DECLARE

在今天的早些时候你已经学习了如何在 DELCARE 部分定义一个指针，DECLARE 语句包括下列指针控制命令。

OPEN

现在你已经定义了一个指针，但是你应该如何去使用它呢？如果你不使用打开这本书你无法看到它的内容的。同样，如果你不使用 OPEN 命令来打开一个指针你也将无法使用它。例如：

语法：

```
BEGIN

    open employee_cursor;

    statement1;

    statement2;

    .....

END
```

FETCH

FETCH 可以从一个指针中取得变量的值。这里有两个使用 FETCH 的例子：一个是取得一个集合变量，另外了个则取得特定的变量。

INPUT:

```
DECLARE

    cursor employee_cursor is

        select emp_id, emp_name from employees;

        employee_record employee_cursor%ROWTYPE;

BEGIN

    open employee_cursor;

    loop

        fetch employee_cursor into employee_record;

    end loop;

    close employee_cursor;

END
```

分析:

上一个例子中将一个指针所指的行的数据赋给了一个名字叫 employee_record 的变量，它使用 LOOP 循环来移动指针，当然，这个块事实上没有做任何工作。

```
DECLARE

    cursor employee_cursor is

        select emp_id, emp_name from employees;

        id_num employees.emp_id%TYPE;

        name employees.emp_name%TYPE;

BEGIN

    open employee_cursor;

    loop

        fetch employee_cursor into id_num, name;

    end loop;

    close employee_cursor;
```

END

分析：

在这个例子中是把当前的指针所指的行的对应数据填入变量 `id_num` 和 `name` 中，这两个变量是在 `DECLARE` 部分中定义的。

CLOSE

如果你已经在一个块中结束了对指针的使用时，你应该将这个指针关闭，与你通常在读完一本书以后要将书合上一样，你应该使用 `CLOSE` 命令来关闭指针。

语法：

```
BEGIN

    open employee_cursor;

    statement1;

    statement2;

    .....

    . close employee_cursor;

END
```

分析：

当关闭一个指针以后，查询的结果集就不复存在了。如果你想访问结果集中的数据你必须重新打开指针才行。

条件语句

现在我们得到了非常有用的东西来控制我们的 `SQL` 语句的运行，在 `PL/SQL` 中的条件语句与大多数第三代编程语言是类似的。

IFTHEN

在大多数编程中 `IF.....THEN` 语句可能是得常用的语句了，它决定了对于特定的条件应当执行哪一部分的操作，其结构如下：

语法：

```
IF condition1 THEN  
    statement1;  
  
END IF;
```

如果你需要对这两种情况进行分别处理，那么你可以将语句形式写成下边的样子：

语法：

```
IF condition1 THEN  
    statement1;  
  
ELSE  
    statement2;  
  
END IF;
```

如果你需要进行判断的条件多于两个，那么语句可以写成下边样子：

语法：

```
IF condition1 THEN  
    statement1;  
  
ELSIF condition2 THEN  
    statement2;  
  
ELSE  
    statement3;  
  
END IF;
```

分析：

在最后一个例子中，如果满足条件 1，那么就执行语句 1，如果满足条件 2，那么就会执行语句 2，否则的话就会执行语句 3，条件语句也可以嵌于其他语句和 LOOP 循环中。

LOOPS 循环

LOOPS 在 PL/SQL 的语句块中将不断地执行过程直到指定的条件满足为止，一共有三种循环。

LOOP 本身是一个无限的循环，它经常在指针中使用，如果你想终止这种循环，你必须指定在什么时候退出。例如：在循环中翻阅指针的时候你可以指定当指针处于最后一行的时候退出循环。见下例：

输入：

```
BEGIN

open employee_cursor;

LOOP

    FETCH employee_cursor into employee_record;

    EXIT WHEN employee_cursor%NOTFOUND;

    statement1;

    .....

END LOOP;

close employee_cursor;

END;
```

%NOTFOUND 是指针的一种属性，它表明在当前指针中没有任何数据。在上一个例子中如果指针没有发现数据就会退出循环。假如你在循环中忽略了这条语句，循环将会一直进行下去。

WHILE-LOOP 则是在当条件满足时执行特定的语句，而当条件不再满足时就会从循环中退出转而执行下一条语句。

输入：

```
DECLARE

cursor payment_cursor is

    select cust_id, payment, total_due from payment_table;

    cust_id payment_table.cust_id%TYPE;

    payment payment_table.payment%TYPE;

    total_due payment_table.total_due%TYPE;

BEGIN

    open payment_cursor;

    WHILE payment < total_due LOOP

        FETCH payment_cursor into cust_id, payment, total_due;

        EXIT WHEN payment_cursor%NOTFOUND;

        insert into underpay_table

        values (cust_id, 'STILL OWES');
```

```
END LOOP;  
  
close payment_cursor;
```

分析：

在上一个例子中使用了 WHILE-LOOP 来对指针进行翻阅，并且在当条件 payment<total_due 为真时一直进行循环。

在上一个例子中你也可以使用 FOR-LOOP 来限定当前的指针所指定的行的数值处于已给定的数值内。

输入：

```
DECLARE  
  
cursor payment_cursor is  
  
select cust_id, payment, total_due from payment_table;  
  
cust_id payment_table.cust_id%TYPE;  
  
payment payment_table.payment%TYPE;  
  
total_due payment_table.total_due%TYPE;  
  
BEGIN  
  
open payment_cursor;  
  
FOR pay_rec IN payment_cursor LOOP  
  
IF pay_rec.payment < pay_rec.total_due THEN  
  
insert into underpay_table  
  
values (pay_rec.cust_id, 'STILL OWES');  
  
END IF;  
  
END LOOP;  
  
close payment_cursor;  
  
END;
```

分析：

在这个例子中使用了 FOR-LOOP 翻阅指针。它是默认的 FETCH 下执行（该语句被省略了），而且在这里%NOTFOUND 属性也被省略了。该属性在 FOR-LOOP 循环中也是默认的。所以这个例子与上一个例子的结果将是相同的。

EXCEPTION 部分

在 PL/SQL 语句块中这一部分是可以选择的。如果在这一部分被省略而遇到异常的时候，该语句块就会终止了。由于一些异常在出现时也许不需要将语句块终止，所以 EXCEPTION 可以用一定的方式来捕获指定的异常或用户定义的异常。异常可以由用户来定义，尽管许多的异常已经在 ORACLE 中进行了预定义。

激活 EXCEPTION（异常）

在语句块中的异常可以由 RAISE 语句来激活，异常可以由程序员进行准确地激活，然而当数据库产生内部错误时它会被自动激活或由默认的数据库服务来调用。

语法：

```
BEGIN

    DECLARE

        exception_name EXCEPTION;

    BEGIN

        IF condition THEN

            RAISE exception_name;

        END IF;

    EXCEPTION

        WHEN exception_name THEN

            Statement;

    END;

END;
```

分析：

这个例子给出了对异常进行准确激活的基本方法，首先 exception_name 要在 EXCEPTION 语句中进行了定义。在 PROCEDURE 部分，当给定的条件满足时这个异常就由 RAISE 语句激活了。然后 RAISE 语句将会引用 EXCEPTION 语句中的对应异常部分以进行适当的工作。

异常的处理

在上例中捕获了在 EXCEPTION 部分中的一个异常，在 PL/SQL 中错误是很容易捕获的，在进行了异常处理以后，PL/SQL 可以在错误状态下继续运行或是以下种合理的方式来终止。

语法：

```
EXCEPTION

    WHEN exception1 THEN

        statement1;

    WHEN exception2 THEN

        statement2;

    WHEN OTHERS THEN

        statement3;
```

分析：

这个例子告诉了你当你有多于一个的异常时应该如何的在 EXCEPTION 部分中进行安排。在这个例子中的块在运行时有两个预料中的异常 (exception1 和 exception2)，如果在该语句中产生了其他的异常就会调用 WHEN OTHERS，它对你的语句块中的其它的错误进行了处理。

运行一个 PL/SQL 语句块

PL/SQL 通常在一个主机的编辑器上创建而它的运行则与常规的 SQL 脚本文件一样。PL/SQL 在语句块中的每一条语句——从变量的赋值到数据操作命令——使用分号结束。

在 SQL 的脚本中正斜线 (/)，但是在 PL/SQL 也使用正斜线来表明脚本的结束，最为容易的运行 PL/SQL 语句块的方法是使用 START 命令，或简写为 STA 或 @。

你的 PL/SQL 脚本可能会像下边这样：

语法：

```
/* This file is called proc1.sql */

BEGIN

    DECLARE

        ...

    BEGIN
```

```
...  
statements;  
  
...  
  
EXCEPTION  
  
...  
  
END;  
  
END;  
  
/
```

你可以像下边这样执行 PL/SQL 脚本：

```
SQL> start proc1    or  
  
SQL> sta proc1      or  
  
SQL> @proc1
```

注：PL/SQL 脚本语言可以使用 START 命令或@字符来运行，它也可以被其它的 PL/SQL 脚本、SHELL 脚本或其它的程序调用。

将输入返回给用户

尤其是在捕获了错误的时候，你可以会希望输出信息给用户告诉他出现了什么错误，你可以转送已有的错误信息。你也可以显示你所定制的错误信息，对于用户来说这会与错误代码更容易理解。也许你想的是当在错误产生时让他们与数据库管理员联系而不是给他们尽可以准确的信息。

PL/SQL 在它的语法部分中并没有提供直接的方式来显示输出，但是它可以让你来调用一个对该语句块服务的包，这个包是由 DBMS_OUTPUT 来调用的。

```
EXCEPTION
```

```
WHEN zero_divide THEN
```

```
DBMS_OUTPUT.put_line('ERROR: DIVISOR IS ZERO. SEE YOUR DBA.');
```

分析：

ZERO_DIVIDE 是 ORACLE 的一个预定义的异常，有许多在程序运行中产生的常见的错误都被预定义为异常并且可以被默认地激活（也就是说你不必在编程的过程中手动将其激活）。

如果在这个语句块运行的过程中产生的异常，用户将会看到：

INPUT：

```
SQL> @block1

ERROR: DIVISOR IS ZERO. SEE YOUR DBA.

PL/SQL procedure successfully completed.

是不是这样的错误信息比下边的错误信息更友好：
```

输入/输出：

```
SQL> @block1

begin

*

ERROR at line 1:

ORA-01476: divisor is equal to zero

ORA-06512: at line 20
```

在 PL/SQL 中的事务控制

在第 11 天中的《事务控制》中，我们已经讨论了事务控制命令 COMMIT、ROLLBACK、SAVEPOINT。这些命令可以让程序员在在事务向数据库中进行写操作时加以控制。在多数时候所进行的操作是需要撤消。

语法：

```
BEGIN

DECLARE

...

BEGIN

statements...

IF condition THEN

COMMIT;

ELSE

ROLLBACK;

END IF;
```

```
...  
EXCEPTION  
  
...  
END;  
  
END;
```

PL/SQL 的一个好处就是你可以用自动地执行事务控制命令来代替对大型事务的不断监控——这是非常单调和乏味的。

让所有的事在一起工作

到目前为止，我们已经介绍了 PL/SQL，你已经熟悉了它所支持的数据类型以及 PL/SQL 语句块的主要特性。你已经知道了如何定义一个局部变量、常量和指针。你也已经知道了如何在一个 PROCEDURE 部分，指针的操作部分和异常部分嵌入 SQL 语句。当指针在使用时，你应该清楚地知道如何在异常部分捕获它。现在你已经可以使用 BEGIN……END 语句块来进行了实际工作。在今天的结束部分，你将会彻底明白 PL/SQL 语句块之间的相同关系。

示例表及数据

在我们创建的 PL/SQL 语句块中使用两个表。PAYMENT_TABLE 确定了一个客户，她/他的付款是多少、应得的总数是多少。PAY_STATUS_TABLE 最初实际上没有任何数据。数据将会依据 PAYMENT_TABLE 中的特定条件插入到 PAY_STATUS_TABLE 表中。

输入：

```
SQL> select * from payment_table;
```

输出：

CUSTOMER	PAYMENT	TOTAL_DUE
ABC	90.50	150.99
AAA	79.00	79.00
BBB	950.00	1000.00
CCC	27.50	27.50
DDD	350.00	500.95
EEE	67.89	67.89

FFF	555.55	455.55
GGG	122.36	122.36
HHH	26.75	0.00

输入：

```
SQL> describe pay_status_table
```

输出：

Name	Null?	Type
CUST_ID	NOT NULL	CHAR(3)
STATUS	NOT NULL	VARCHAR2(15)
AMT_OWED		NUMBER(8,2)
AMT_CREDIT		NUMBER(8,2)

分析：

DESCRIBE 是一个 ORACLE SQL，它可以不通过查询数据字典就可以显示一个表的结构，它与其它的 ORACLE SQL*PLUS 命令将会在第 20 天《SQL*PLUS》中提到。

一个简单的 PL/SQL 语句块

这个 PL/SQL 的脚本内容如下所示：

输入：

```
set serveroutput on

BEGIN

DECLARE

    AmtZero EXCEPTION;

    cCustId payment_table.cust_id%TYPE;

    fPayment payment_table.payment%TYPE;

    fTotalDue payment_table.total_due%TYPE;

    cursor payment_cursor is

        select cust_id, payment, total_due

        from payment_table;

    fOverPaid number(8,2);

    fUnderPaid number(8,2);

BEGIN
```

```
open payment_cursor;

loop

    fetch payment_cursor into

        cCustId, fPayment, fTotalDue;

    exit when payment_cursor%NOTFOUND;

    if ( fTotalDue = 0 ) then

        raise AmtZero;

    end if;

    if ( fPayment > fTotalDue ) then

        fOverPaid := fPayment - fTotalDue;

        insert into pay_status_table (cust_id, status, amt_credit)

            values (cCustId, 'Over Paid', fOverPaid);

    elsif ( fPayment < fTotalDue ) then

        fUnderPaid := fTotalDue - fPayment;

        insert into pay_status_table (cust_id, status, amt_owed)

            values (cCustId, 'Still Owes', fUnderPaid);

    else

        insert into pay_status_table

            values (cCustId, 'Paid in Full', null, null);

    end if;

end loop;

close payment_cursor;

EXCEPTION

    when AmtZero then

        DBMS_OUTPUT.put_line('ERROR: amount is Zero. See your supervisor.');
```

```
    when OTHERS then

        DBMS_OUTPUT.put_line('ERROR: unknown error. See the DBA');

END;

END;

/
```

分析：

DECLARE 部分定义了六个局部变量，与被称为 payment_cursor 指针一样。PROCEDURE 从第二个 BEGIN 语句开始并先打开了这个游标并开始循环，而 FETCH 命令则将当前指针所指向的记录的内容存入在 DECLARE 部分定义的变量中。当在指针中发现记录以后，语句会将客户的支付与他应付的总数进行比较，根据支付的的数量来计算已付款的人数和未付款的人数。然后将计算过的数据插入到 PAY_STATUS_TABLE 表中。当循环终止以后，关闭指针，异常用于处理在这一部分可以发生的错误。

现在，我们来运行一个这个脚本看一个他的结果：

INPUT:

SQL> @block1

OUTPUT:

Input truncated to 1 characters

ERROR: amount is Zero. See your supervisor.

PL/SQL procedure successfully completed.

现在看来你在应付总数上有一个数值是不正确的，你应该修正这个数量然后再运行脚本。

输入/输出：

SQL> update payment_table set total_due = 26.75 where cust_id = 'HHH';

1 row updated.

SQL> commit;

Commit complete.

SQL> truncate table pay_status_table;

Table truncated.

注：在上例中我们清除了 pay_status_table 中的内容，在下一次运行这个语句块时表将会被重新写入。你也许会想把清除语句也加入到语句块中吧！

输入/输出：

SQL> @block1

Input truncated to 1 characters

PL/SQL procedure successfully completed.

现在你可以对 PAY_STATUS_TABLE 表执行 SELECT 语句，看一下每一个客户的支付情况：

输入/输出：

```
SQL> select * from pay_status_table order by status;
```

CUSTOMER	STATUS	AMT_OWED	AMT_CREDIT
FFF		OverPaid	100.00
AAA		PaidinFull	
CCC		PaidinFull	
EEE		PaidinFull	
GGG		PaidinFull	
HHH		PaidinFull	
ABC		StillOwes	60.49
DDD		StillOwes	150.95
BBB		StillOwes	50.00

分析：

从 PAYMENT_TABLE 获得的数据中有一行被插入到了 PAY_STATUS_TABLE 表中，如果用户的付款金额比应付金额更多，那么差额将会被输入到 AMT_CREDIT 列中，如果用户的支付金额少于应付金额，那么将会在 AMT_OWED 中输入一个信息。如果客户已经全部支付完毕，那么这两列中就不会有任何数据。

又一个程序

在这个例子中使用的表叫 PAY_TABLE：

输入：

```
SQL> desc pay_table
```

输出：

Name	Null?	Type
NAME	NOT NULL	VARCHAR2(20)
PAY_TYPE	NOT NULL	VARCHAR2(8)
PAY_RATE	NOT NULL	NUMBER(8,2)
EFF_DATE	NOT NULL	DATE
PREV_PAY		NUMBER(8,2)

先来看一下数据：

输入：

```
SQL> select * from pay_table order by pay_type, pay_rate desc;
```

输出：

NAME	PAY_TYPE	PAY_RATE	EFF_DATE	PREV_PAY
SANDRA SAMUELS	HOURLY	12.50	01-JAN-97	
ROBERT BOBAY	HOURLY	11.50	15-MAY-96	
KEITH JONES	HOURLY	10.00	31-OCT-96	
SUSAN WILLIAMS	HOURLY	9.75	01-MAY-97	
CHRISSY ZOES	SALARY	50000.00	01-JAN-97	
CLODE EVANS	SALARY	42150.00	01-MAR-97	
JOHN SMITH	SALARY	35000.00	15-JUN-96	
KEVIN TROLLBERG	SALARY	27500.00	15-JUN-96	

现实情况：由于销售情况很好，你需要给为你工作的时间超过了六个月的个人增加薪金。符合条件的钟点工的薪金增加 4%，而符合全条件的雇员的薪金需要增加 5%。

今天的日期是：

输入/输出：

```
SQL> select sysdate from dual;
```

SYSDATE

20-MAY-97

在对下边的 PL/SQL 语句块进行检查之前，我们要对 PAY_TABLE 表进行手工的选择以找出都有哪些人需要增加薪金。

输入：

```
SQL> select name, pay_type, pay_rate, eff_date,
```

```
2         'YES' due
```

```
3   from pay_table
```

```
4  where eff_date < sysdate - 180
```

```
5  UNION ALL
```

```
6  select name, pay_type, pay_rate, eff_date,
```

```
7         'No' due
```

```
8   from pay_table
```

```
9  where eff_date >= sysdate - 180
```

10 order by 2, 3 desc;

输出：

NAME	PAY_TYPE	PAY_RATE	EFF_DATE	DUE
SANDRA SAMUELS	HOURLY	12.50	01-JAN-97	No
ROBERT BOBAY	HOURLY	11.50	15-MAY-96	YES
KEITH JONES	HOURLY	10.00	31-OCT-96	YES
SUSAN WILLIAMS	HOURLY	9.75	01-MAY-97	No
CHRISSY ZONES	SALARY	50000.00	01-JAN-97	No
CLODE EVANS	SALARY	42150.00	01-MAR-97	No
JOHN SMITH	SALARY	35000.00	15-JUN-96	YES
KEVIN TROLLBERG	SALARY	27500.00	15-JUN-96	YES

DUE 列的内容是确定每一个人是否有增加工资资格。下边是 PL/SQL 的脚本：

输入：

```

set serveroutput on

BEGIN

DECLARE

    UnknownPayType exception;

    cursor pay_cursor is

        select name, pay_type, pay_rate, eff_date,

               sysdate, rowid

        from pay_table;

    IndRec pay_cursor%ROWTYPE;

    cOldDate date;

    fNewPay number(8,2);

BEGIN

    open pay_cursor;

    loop

        fetch pay_cursor into IndRec;

        exit when pay_cursor%NOTFOUND;

        cOldDate := sysdate - 180;

        if (IndRec.pay_type = 'SALARY') then

```

```

        fNewPay := IndRec.pay_rate * 1.05;

    elsif (IndRec.pay_type = 'HOURLY') then

        fNewPay := IndRec.pay_rate * 1.04;

    else

        raise UnknownPayType;

    end if;

    if (IndRec.eff_date < cOldDate) then

        update pay_table

        set pay_rate = fNewPay,

            prev_pay = IndRec.pay_rate,

            eff_date = IndRec.sysdate

        where rowid = IndRec.rowid;

        commit;

    end if;

end loop;

close pay_cursor;

EXCEPTION

when UnknownPayType then

    dbms_output.put_line('=====');

    dbms_output.put_line('ERROR: Aborting program. ');

    dbms_output.put_line('Unknown Pay Type for Name');

when others then

    dbms_output.put_line('ERROR During Processing. See the DBA. ');

END;

END;

/

```

你是否已经决定了要给这四个雇员增加工资？（在上边的 SELECT 语句中有四个人有 YES 标记）。为什么不呢？让我们给所有的这四个人加薪吧！你可以通过运行名字叫 block2.sql 的脚本来自动为这四个人进行合理的加薪。

输入/输出：

SQL> @block2

Input truncated to 1 characters

PL/SQL procedure successfully completed.

你可以作一个快速的检查也确定对于每个人的薪金的增加比率是多少。

输入：

SQL> select * from pay_table order by pay_type, pay_rate desc;

输出：

NAME	PAY_TYPE	PAY_RATE	EFF_DATE	PREV_PAY
SANDRA SAMUELS	HOURLY	12.50	01-JAN-97	
ROBERT BOBAY	HOURLY	11.96	20-MAY-97	11.5
KEITH JONES	HOURLY	10.40	20-MAY-97	10
SUSAN WILLIAMS	HOURLY	9.75	01-MAY-97	
CHRISSY ZONES	SALARY	50000.00	01-JAN-97	
CLODE EVANS	SALARY	42150.00	01-MAR-97	
JOHN SMITH	SALARY	36750.00	20-MAY-97	35000
KEVIN TROLLBERG	SALARY	28875.00	20-MAY-97	27500

分析：

四个雇员的薪金已经增加了，如果将现在的输出和原来的 SELECT 的输出做比较的话，你会发现相应的改变，当前的薪金率的变化反映和薪金的增加。原有的薪金率被插入到了 PREV_PAY 列中，而有效日期则被更新为当前的日期，没有符合资格的人的情况则没有任何变化。

请等一下！我们没有看到定义的异常工作的机会，你可以向 PAY_TABLE 表中插入一个不合法的记录来对异常部分进行检测。

输入：

SQL> insert into pay_table values

2 ('JEFF JENNINGS','WEEKLY',71.50,'01-JAN-97',NULL);

输出：

1 row created.

输入/输出：

SQL> @block2

```
Input truncated to 1 characters
```

```
=====
```

```
ERROR: Aborting program.
```

```
Unknown Pay Type for: JEFF JENNINGS
```

```
PL/SQL procedure successfully completed.
```

分析：

错误信息表明 JEFF JENNINGS 的薪金支付方式不是 HOURLY 和 SALARY。这就是异常所捕获到的错误信息。

存储过程、包和触发机制

使用 PL/SQL，你可以创建存储对象来代替日复一日的输入单调和枯燥的代码。过程是一些可以执行一些特定类型的存储工作的代码块，相关的过程可以组合和存储在一起，这称为包。触发机制是一种在其它的事务中使用的数据库对象，你也许对一个叫 ORDERS 的表建立了一个触发机制以使得每次当 ORDERS 表接受到数据时都向 HISTORY 表中插入数据。这些对象的基本语法如下：

过程示例：

语法：

```
PROCEDURE procedure_name IS
```

```
    variable1 datatype;
```

```
    ...
```

```
BEGIN
```

```
    statement1;
```

```
    ...
```

```
EXCEPTION
```

```
    when ...
```

```
END procedure_name;
```

示例包：

语法：

```
CREATE PACKAGE package_name AS

    PROCEDURE procedure1 (global_variable1 datatype, ...);

    PROCEDURE procedure2 (global_variable1 datatype, ...);

END package_name;

CREATE PACKAGE BODY package_name AS

    PROCEDURE procedure1 (global_variable1 datatype, ...) IS

        BEGIN

            statement1;

            ...

        END procedure1;

    PROCEDURE procedure2 (global_variable1 datatype, ...) IS

        BEGIN

            statement1;

            ...

        END procedure2;

END package_name;
```

示例触发机制

SYNTAX:

```
CREATE TRIGGER trigger_name

    AFTER UPDATE OF column ON table_name

    FOR EACH ROW

    BEGIN

        statement1;

        ...

    END;
```

下边的例子在当对 PAY_TABLE 表的数据进行更新时使用触发机制向一个事务表中插入数据。事务表如下所示：

INPUT:

SQL> describe trans_table

OUTPUT:

Name	Null?	Type
ACTION(10)		VARCHAR2
NAME		VARCHAR2(20)
PREV_PAY		NUMBER(8,2)
CURR_PAY		NUMBER(8,2)
EFF_DATE		DATE

示例行的数据如下：

输入/输出：

SQL> select * from pay_table where name = 'JEFF JENNINGS';

NAME	PAY_TYPE	PAY_RATE	EFF_DATE	PREV_PAY
JEFF JENNINGS	WEEKLY	71.50	01-JAN-97	

现在，创建一个触发机制：

SQL> CREATE TRIGGER pay_trigger

```

2  AFTER update on PAY_TABLE
3  FOR EACH ROW
4  BEGIN
5  insert into trans_table values
6  ('PAY CHANGE', :new.name, :old.pay_rate,
7   :new.pay_rate, :new.eff_date);
8  END;
9  /
```

然后对 PAY_TABLE 进行更新操作，这会导致触发机制的运行：

输入/输出：

SQL> update pay_table

```

2  set pay_rate = 15.50,
3  eff_date = sysdate
```

```
4 where name = 'JEFF JENNINGS';
```

```
SQL> select * from pay_table where name = 'JEFF JENNINGS';
```

NAME	PAY_TYPE	PAY_RATE	EFF_DATE	PREV_PAY
JEFF JENNINGS	WEEKLY	15.50	20-MAY-97	

```
SQL> select * from trans_table;
```

ACTION	NAME	PREV_PAY	CURR_PAY	EFF_DATE
PAY CHANGE	JEFFJENNINGS	71.5	15.5	20-MAY-97

分析：

在 PAY_TABE 表中的 PREV_PAY 中是空的，但是在 TRANS_TABLE 中则存在数值。你是不是糊涂了。PAY_TABLE 是不需要 PREV_PAY 的，因为每小时的薪金为 71.5 在这里很明显是一个错误的数值。由于更新操作是一个事务，所以我们将 PREV_PAY 的数值插入到了 TRANS_TABLE 表中，它的目的是为所以的用 PAY_TABLE 工作的表保存记录。

注：如果你工作在类似的网络环境中，你也许会注意到 PL/SQL 与 JAVA 的存储过程有一些类似。但是，你要注意到他们的不同之处，PL/SQL 是对标准的 SQL 的增强，它是一种过程型语言。JAVA 比它有更多的先进的特性，它允许程序写出比 PL/SQL 更为复杂的程序，PL/SQL 是基于指定的数据库的增强型 SQL。而 JAVA 则在 CPU 级上工作的程序。大多数的过程型语言，如 PL/SQL 是针对特定的平台开发的，而 JAVA 则比过程型语言更高级，它可以在交叉的平台上工作并可以实现标准化。

总结

PL/SQL 对标准的 SQL 进行了扩展，PL/SQL 所执行的基本功能与第三代语言相同，它可以使用局部变量来支持动态代码，也就是说块内的数值可以根据用户的输入、指定的条件、和指针的内容的变化而变化，PL/SQL 使用标准的过程语言来对语句进行控制，IF…… THEN 和 LOOP 可以让你按指定的条件搜索。你也可以使用 LOOP 来对指定的指针的内容进行翻阅。

在任何程序中都会有各种错误产生，PL/SQL 通过异常可以让你对产生错误后的行为进行控制。许多异常是预定义过的，如被零除错误，异常可以在程序运行时根据指定的条件激活并按程序员所定义的方式进行处理。

在今天也介绍一些对 PL/SQL 的实际应用，数据库对象如触发机制、存储过程、包可以自动完成许多功能，在今天的例子中我们也应用了一些在前一天中所提到的概念。

问与答

问：在第 18 天中我是否已经学习了我需要 PL/SQL 所掌握的所有内容？

答：当然不是，像今天的介绍只是提及到了一些表层的一些与 SQL 相关的东西，我们只是提及了一些 SQL 的非常明显的特性使你对 PL/SQL 有一个基本的了解。

问：我不用 PL/SQL 行不行？

答：当然，你不使用它也是可以的，但是如果你不使用它你会为达到相同的目的而不得不在第三代编程语言中使用更多的时间和代码。如果你没有使用 ORACLE，那么请检查你的解释器以找到与 PL/SQL 类似的过程方法。

校练场

- 1、如何在数据库中使用触发机制？
- 2、是否可以将相关的过程存储在一起？
- 3、可以在 PL/SQL 中使用数据操作语言，对不对？
- 4、可以在 PL/SQL 中使用数据定义语言，对不对？
- 5、在 PL/SQL 的语法中是否支持直接的文本输出？
- 6、给出 PL/SQL 语句块的三个主要部分？
- 7、请给出与指针控制相关的命令？

练习

- 1、请定义一个变量，使它可以接受的最大数值为 99.99。
- 2、请定义一个指针，它的内容包括 CUSTOMER_TABLE 表中的所有 CITY 为 INDIANAPOLIS 的客户。
- 3、定义一个名字为 UnknownCode 的异常。
- 4、请写一个语句，使得在 AMOUNT_TABLE 中的 AMT 当 CODE 为 A 时其值为 10，当 CODE 为 B 时其值为 20，当 CODE 既不是 A 也不是 B 时激活一个名字叫 UnknownCode 的异常，表中的内容只有一行。

第 19 天：TRANSACT-SQL 简介

目标

与 TRANSACT-SQL 是对标准 SQL 的补充一样，今天的内容是对前几天内容的补充。
今天的目标是：

- 知道一种对 SQL 的流行的扩展。
- 知道 TRANSACT-SQL 的主要特性。
- 给出一些特殊的例子让你知道如何去使用 TRANSACT-SQL。

TRANSACT-SQL 概貌

在第 13 天的《高级 SQL》中我们简要地提到过静态 SQL。在第 13 天的例子中我们也描述了如何在第三代编程语言如 C 中写内嵌的 SQL 语句的方法。由于采用这种方法时嵌入的 SQL 语句是无法改变的所以它的灵活性就受到了限制，而如果我们采用动态的 SQL 语言编程来完成相同的工作时，就允许 SQL 代码的条件在运行时改变。

在本书中我们其实已经讨论过了相关的每一个主题，几乎每一个数据库供应商都在它的语言进行了相应的扩展，TRANSACT-SQL 是 SYBASE 和 MICROSOFT SQL SERVER 的产品，而 ORACLE 的产品是 PL/SQL。这里的每一种语言可以完成全部的到目前为止我们所讨论的每一件事。此外，每一种产品都对标准的 SQL 进行了相应的扩展。

对 ANSI SQL 的扩展

为了演示使用这些扩展来创建实际的程序，我们使用了 SYBASE 和 MICROSOFT SQL SERVER 的 TRANSACT-SQL。它具有大多数的在第三代编程语言中具有的结构。对于针对 SQL-SERVER 的特性它也提供了许多便利的工具用以进行数据库编程（在其它的数据库供应商中也提供了与之类似和更多的特性）。

谁需要使用 TRANSACT-SQL

任何一个读过本书的人都会使用 TRANSACT-SQL。如果是一个程序员它偶尔会用它来写一个查询，如果是开发人员则可以用它写应用程序以创建对象如触发机制和存贮过程等。

注：SYBASE 和 MICROSOFT SQL SERVER 的用户如果想开发实际上的关系数据库应用程序就必须使用 TRANSACT-SQL 的特性。

TRANSACT-SQL 的基本组件

对 SQL 的扩展已经超过了 SQL 作为一种过程型语言的限制。例如：TRANSACT-SQL 可以让你对数据库的事务进行紧密的控制并且可以写出数据库过程程序以把编程人员从冗重的代码中解放出来。

- 在第 19 天我们主要会提到 TRANSACT-SQL 的以下主要特性：
- 提供了更大范围的数据类型以优化数据的存贮。
- 程序流控制命令如 IF-THEN 和 LOOP 语句。
- 在 SQL 语句中使用变量。
- 使用 COMPUTATION 生成摘要报告。
- 对 SQL 语句的诊断和分析特性。
- 对标准的 SQL 语句提供了许多其它的选项。

数据类型

在第 9 天的《创建和操作表》中我们讨论过数据类型，当使用 SQL 创建表时，我们必须为每一列指定数据类型。

注：在不同的 SQL 解释器中数据的类型是不同的，因为每一种数据库服务存储数据的方法都是各不相同的。举例来说：ORACLE 有它自己选定的数据类型，而 SYBASE 和 MICROSOFT SQL SERVER 则有他们自己的数据类型。

SYBASE 和 MICROSOFT SQL SERVER 支持下列数据类型：

字符串

char 用以存储长度固定的字符串，例如 STATE 的缩写——你知道这一列只有两个字符。

Varchar 用以存储长度可变的字符串，如人名，它是无法对其长度进行预先指定的。例如：AL RAY 与 WILLIAM STEPHENSON。

Text 存储的字符长度几乎是不受限制的例如一种服务的备注和描述字段。

数字类型

int 存储的整型数值范围为-2,147,483,647 到+2,147,483,647。

Smallint 存储的整型数值的范围为-32,768 到 32,767。

Tinyint 存储的整型数值的范围为 0 到 255。

Float 可以存储有精度要求的浮点数，数值范围为+2.23E-308 和+1.79E308。

Real 可以存储的数据的精度为 1.18E-38 to +3.40E38。

日期类型

datetime 可以存储的数值从 Jan 1, 1753 到 Dec 31, 9999。

Smalldatetime 可以存储的数值从 Jan 1, 1900 到 Jun 6, 2079。

货币类型

money 的存储数值上限为+922,337,203,685,477.5808。

smallmoney 的存储数值上限为+214,748.3647。

在向表中输入 MONEY 数值类型时要使用美元符号\$，例如：

```
insert payment_tbl (customer_id, paydate, pay_amt)
values (012845, "May 1, 1997", $2099.99)
```

二进制串

binary 存储着长度固定的二进制串。

Varbinary 存储着长度可变的二进制串。

Image 则用于存储非常大的二进制类型，例如图形和图像。

位：逻辑数据类型

bit 数据类型经常用在表中对某行数据进行标识，存储在 bit 数据类型中的数据只有 0 和 1。例如，如果 1 代表符合某个条件，那么 0 就表示不符合这个条件。在下例中使用了 bit 数据类型来创建了一个包括个人测试成绩的表格。

```
create table test_flag  
( ind_id int not null,  
  test_results int not null,  
  result_flag bit not null)
```

分析：

在这里，result_flag 列被定义为 bit 类型，因为这时只需要返回是及格还是不及格，如果及格就为 1 否则为 0。

在今天的余下的时间里，你应该注意在确定的表格中数据类型的使用以及如果写 TRANSACT-SQL 代码。

注：在今天的例子中的代码既可以是大写也可以是小写，尽管在大多数的 SQL 解释器中 SQL 的关键字大小写是不敏感的，但是你还是应该检查你的解释器以进行最终的确定。

使用 TRANSACT-SQL 来访问数据库

好了，说得够多了，如果你想运行今天的实例，你需要在名字为 BASEBALL 的数据库中创建下边的表。

BASEBALL 数据库

BASEBALL 数据库使用了三个表来对 BASEBALL 的信息进行追踪。这三个表分别是 BATTER、PITCHERS 和 TEAM 表，在今天的剩余时间中我们将一直使用这个例子。

BASEBALL 表

NAME char(30)

TEAM int

AVERAGE float

HOMERUNS int

RBIS int

这个表可以使用下边的 TRANSACT-SQL 语句来创建:

输入:

```
1> create database BASEBALL on default
```

```
2> go
```

```
1> use BASEBALL
```

```
2> go
```

```
1> create table BATTERS (
```

```
2> NAME char(30),
```

```
3> TEAM int,
```

```
4> AVERAGE float,
```

```
5> HOMERUNS int,
```

```
6> RBIS int)
```

```
7> go
```

分析:

在第 1 行创建了数据库, 你创建了名字为 BASEBALL 的数据库, 然后在它的下面用创建了一个 BASEBALL 的表。

向表中输入下表的数据。

注: 在上例中的每个 TRANSACT-SQL 语句之后的 GO 命令并不是 TRANSACT-SQL 的组成部分, 它只是将 SQL 语句从前端送到的服务上。

Name	Team	Average	Homeruns	RBIs
Billy Brewster	1	0.275	14	46
John Jackson	1	0.293	2	29
Phil Hartman	1	0.221	13	21
Jim Gehardy	2	0.316	29	84

Tom Trawick	2	0.258	3	51
Eric Redstone	2	0.305	0	28

PITCHERS 表

可以用下边的 TRANSACT-SQL 语句来创建 PITCHERS 表。

INPUT:

1> use BASEBALL

2> go

1> create table PITCHERS (

2> NAME char(30),

3> TEAM int,

4> WON int,

5> LOST int,

6> ERA float)

7> go

向其中输入下表的数据:

Name	Team	Won	Lost	Era
Tom Madden	1	7	5	3.46
Bill Witter	1	8	2	2.75
Jeff Knox	2	2	8	4.82
Hank Arnold	2	13	1	1.93
Tim Smythe	3	4	2	2.76

TEAM 表

可以用下边的 TRANSACT-SQL 语句来创建 TEAM 表。

输入:

1> use BASEBALL

2> go

1> create table TEAMS (

2> TEAM_ID int,

3> CITY char(30),

```

4> NAME char(30),
5> WON int,
6> LOST int,
7> TOTAL_HOME_ATTENDANCE int,
8> AVG_HOME_ATTENDANCE int)
9> go

```

Team_ID	City	Name	Won	Lost
1	Portland	Beavers	72	63
2	Washington	Representatives	50	85
3	Tampa	Sharks	99	36

定义局部变量

每一个编程语言都为你提供了一种方法使你可以创建局部（或全局）变量以存储一些数据。TRANSACT-SQL 当然也不例外。在 TRANSACT-SQL 中定义变量是非常容易的事情，关键是你必须要使用 DECLARE 关键字。语法的形式如下：

语法：

```
declare @variable_name data_type
```

如果你想定义一个字符串来存储 PLAYER 的名字，那么你可以使用下边的语句。

```

1> declare @name char(30)
2> go

```

注意在变量之前有一个@符号，这个符号在查询中用以标识变量。

定义全局变量

如果你对 TRANSACT-SQL 看得更多，你会知道 @@提示符会优先于系统级的变量，这个语法用以表明在 SQL SERVER 中定义的存储信息的全局变量。

在使用存储过程的时候你可以自己定义全局变量是非常有用的。SQL SERVER 也提供了几种系统全局变量，对于数据库的系统用户来说它可能是有用的。下表中给出了这些变量的全部清单，你可以在 SQL SERVER SYSTEM10 的文档中找到它。

变量名	作用
@@char_convert	如果字符转换成功时其值为0
@@client_csid	客户机所使用字符集的ID

@@client_csname	客户机的字符集的名字
@@connections	从SQL Server启动以来的登录次数
@@cpu_busy	从SQL Server启动以来的CPU“忙”的时间总数
@@error	错误的状态
@@identity	插入到确定列中的最后一个值
@@idle	从SQL Server启动以来的总时间数
@@io_busy	SQL Server用于I/O操作的时间
@@isolation	当前的Transact-SQL程序的隔离级别
@@langid	定义了本地语言的ID号
@@language	定义了本地语言的名称
@@maxcharlen	字符的最大长度
@@max_connections	可与SQL SERVER进行连接的最大数量。
@@ncharsize	Average length of a national character.
@@nestlevel	当前进程的嵌套级别
@@pack_received	从SQL Server启动以来的读入的数据包的数量
@@pack_sent	从SQL Server所发出的输出包的数量
@@packet_errors	从SQL Server启动以来产生错误的数量
@@procid	当前正在运行的存储过程的ID号
@@rowcount	上一个命令所涉及的行数
@@servername	本地local SQL Server的名字
@@spid	当前正在处理的进程ID号
@@sqlstatus	存储状态信息
@@textsize	由SELECT语句所返回的文本映像的最大长度。
@@thresh_hysteresis	Change in free space required to activate a threshold.
@@timeticks	Number of microseconds per tick.
@@total_errors	在读写过程中产生的错误数
@@total_read	在SQL Server启动以来读磁盘的次数
@@total_write	在SQL Server启动以来写磁盘的次数
@@tranchained	在Transact-SQL程序中当前事务的模式
@@tranccount	事务的嵌套级别
@@transtate	当一个语句运行后当前事务的状态。
@@version	当前SQL Server的版本日期。

使用变量

使用 `DECLARE` 关键字你也可以在一行中同时定义多个变量（尽管在稍后你在看你的代码的时候可以能糊涂），见下例：

```
1> declare @batter_name char(30), @team int, @average float
```

```
2> go
```

接下来的部分将会说明如何使用变量进行有用的程序操作。

使用变量来存储数据

变量只有在当前的语句块中有效，在 `TRANSACTION-SQL` 中如果你想运行一个语句块，

你需要使用 GO 语句（在 ORACLE 中的分号有相同的作用）变量只能在当前事务中被引用。

你不能使用=来简单地初始化变量，你可以试一下下边的语句，你将会发现它会返回错误。

输入：

```
1> declare @name char(30)
2> @name = "Billy Brewster"
3> go
```

你将会收到一个错误信息告诉你第二行的语法是不正确的。正确的初始化变量的方法是使用 SELECT 语句（你已经会使用这个命令了），现在用正确的语法来重复上边的例子：

输入：

```
1> declare @name char(30)
2> select @name = "Billy Brewster"
3> go
```

这个语句将会正确的执行，如果你在 GO 语句之前又插入了其它的语句，那么该变量就可以在其它的语句中使用。

将数据存入局部变量

变量经常用来存储从数据库中获得的数据，它们也可以在通常的 SQL 语句中使用，如：SELECT、INSERT、UPDATE 和 DELETE，下例中给出了使用方法。

例 19.1

这个例子将会返回在 BASEBALL 表中的在 Portland Beavers.的平均击球成功率最高的 PLAYER 的名字。

输入：

```
1> declare @team_id int, @player_name char(30), @max_avg float
2> select @team_id = TEAM_ID from TEAMS where CITY = "Portland"
3> select @max_avg = max(AVERAGE) from BATTERS where TEAM = @team_id
4> select @player_name = NAME from BATTERS where AVERAGE = @max_avg
```

```
5> go
```

在本例中分成了三个查询以给出变量的使用演示。

PRINT 命令

TRANSACT-SQL 的另一个有用的功能就是 PRINT，它可以让你向显示设备上输出信息，该命令的语法格式如下：

语法：

```
PRINT character_string
```

尽管 PRINT 命令只可以输入字符串，但是在 TRANSACT-SQL 中提供了很多的函数可以让你把其它的数据类型转化为字符串（而且还可以逆向转换）。

例 19.2

本例重复了上边的例子，但是在最后则可以将 PLAYER 的名字打印出来。

输入：

```
1> declare @team_id int, @player_name char(30), @max_avg float
2> select @team_id = TEAM_ID from TEAMS where CITY = "Portland"
3> select @max_avg = max(AVERAGE) from BATTERS where TEAM = @team_id
4> select @player_name = NAME from BATTERS where AVERAGE = @max_avg
5> print @player_name
6> go
```

注意在 WHERE 子句（或其它的子句）中也可以使用变量，就好象它们是常量一样。

流控制

对于 TRANSACT-SQL 的最为强大的功能也许就是它具有程序流的控制功能。如果你使用其它的语言如 C、COBOL、PASCAL、VISUAL BASIC 编过程序，那么你也许会见过类似于 IF……THEN 或循环的控制命令。这一部分包括了一些主要的程序流控制命令。

BEGIN.....END 语句

TRANSACT-SQL 使用 BEGIN 和 END 来标记一个程序语句块的开始和结束。在其它的语言中则可能使用括号或其它的操作符来标记程序块的开始与结束。它也经常与 IF.....ELSE 和 WHILE 循环。使用 BEGIN 和 END 的例子如下：

语法：

```
BEGIN
    statement1
    statement2
    statement3...
END
```

IF.....ELSE 语句

IF.....ELSE 是最基本的编程语句结构之一，几乎每一种编程语言都支持这种结构。而它在用于对从数据库返回的数据进行检查是非常有用的。TRANSACT-SQL 使用 IF.....ELSE 的例子如下：

语法：

```
if (condition)
begin
    (statement block)
end
else if (condition)
begin
    statement block)
end
.....
else
begin
    (statement block)
```

end

注意，当所指定的条件为真时，对应的 BEGIN……END 语句块就会被执行。同时，你也应该注意将每一个语句缩进一定量的空格是一种很好的编程习惯。它可以极大的提高你的程序的易读性和由于易读性不好所导致的错误。

例 19.3

本例对上例增加了对平均击中率的检查，如果选手的平均分大于.300。业主将会将他升级，否则业主对他仍然不会留意。

例 19.3 使用了 IF……THEN 对所给条件进行判断。如果第一个条件为真，就会执行对应的文本，否则就继续进行其它的判断。

输入：

```
1> declare @team_id int, @player_name char(30), @max_avg float
2> select @team_id = TEAM_ID from TEAMS where CITY = "Portland"
3> select @max_avg = max(AVERAGE) from BATTERS where TEAM = @team_id
4> select @player_name = NAME from BATTERS where AVERAGE = @max_avg
5> if (@max_avg > .300)
6> begin
7>     print @player_name
8>     print "Give this guy a raise!"
9> end
10> else
11> begin
12>     print @player_name
13>     print "Come back when you're hitting better!"
14> end
15> go
```

例 19.4

下边的例子对数据库所进行的查询使用了 IF 语句并加入了一些逻辑语句，本例对让例的分支进行了细化处理。

输入：

```
1> declare @team_id int, @player_name char(30), @max_avg float
2> select @team_id = TEAM_ID from TEAMS where CITY = "Portland"
3> select @max_avg = max(AVERAGE) from BATTERS where TEAM = @team_id
4> select @player_name = NAME from BATTERS where AVERAGE = @max_avg
5> if (@max_avg > .300)
6> begin
7>     print @player_name
8>     print "Give this guy a raise!"
9> end
10> else if (@max_avg > .275)
11> begin
12>     print @player_name
13>     print "Not bad. Here's a bonus!"
14> end
15> else
16> begin
17>     print @player_name
18>     print "Come back when you're hitting better!"
19> end
20> go
```

TRANSACT-SQL 也可以让你对 IF 所使用的条件进行检查，这一功能可以对特定的条件值进行测试，如果测试的条件为真，那么 IF 分支就会被执行，否则的话如果有 ELSE 部分就执行 ELSE 部分，就像你在上边的例子中看到的一样。

EXIST 条件

EXIST 命令可以确保从 SELECT 语句中返回数值，如果返回了数值，那么 IF 语句就会被执行，例 19.5 给出了示例。

例 19.5

在本例就 EXIST 关键字对 IF 的条件进行了测试，该条件是由 SELECT 语句所指定的。

输入：

```
1> if exists (select * from TEAMS where TEAM_ID > 5)
2> begin
3>     print "IT EXISTS!!"
4> end
5> else
6> begin
7>     print "NO ESTA AQUI!"
8> end
```

测试查询的结果

IF 语句也可以对 SELECT 语句返回的查询结果进行测试，例 19.6 给出的这样的例子，它是对平均打点最高的选手进行测试。

例 19.6

本例所 19.5 的例子类似，也是使用 SELECT 语句来定义条件。但是这一回我们使用大于号来进行检查。

输入：

```
1> if (select max(AVG) from BATTERS) > .400
2> begin
3>     print "UNBELIEVABLE!!"
4> end
5> else
6>     print "TED WILLIAMS IS GETTING LONELY!"
```

```
7> end
```

我们推荐你在你的解释器中对 IF 语句进行试验。如果对于 BASEBALL 数据库你有多
个条件要进行测试，那么你可以像在本次的应用中那样在 IF 语句中使用查询。

WHILE 循环

为 TRANSACT-SQL 所运行的另一种流行的编程结构是 WHILE 循环，该命令的语法
如下：

语法：

```
WHILE logical_expression  
    statement(s)
```

例 19.7

WHILE 将一直循环到所给出的逻辑表达式值为假时为止，本例中使用了一个简单的
WHILE 循环来增加一个名字为 COUNT 的局部变量。

输入：

```
1> declare @COUNT int  
2> select @COUNT = 1  
3> while (@COUNT < 10)  
4> begin  
5>     select @COUNT = @COUNT + 1  
6>     print "LOOP AGAIN!"  
7> end  
8> print "LOOP FINISHED!"
```

注：上例实际上是一个简单的 FOR 循环，在一些其它的解释器如 ORACLE 的 PL/SQL 中
确实提供一名称为 FOR 的循环方式，你应该检查你的系统看它是否支持这种循环方式。

BREAK 命令

注意 BREAK 语句应该在 IF 条件语句中出现。

输入：

```
1> declare @COUNT int
2> select @COUNT = 1
3> while (@COUNT < 10)
4> begin
5>     select @COUNT = @COUNT + 1
6>     if (@COUNT = 8)
7>         begin
8>             break
9>         end
10>     else
11>         begin
12>             print "LOOP AGAIN!"
13>         end
14> end
15> print "LOOP FINISHED!"
```

分析：

在这个例子中当 COUNT 的值为 8 时 BREAK 将导致循环的退出。

COUNTINUE 命令

COUNTINUE 命令也是一个可以在 WHILE 循环中执行的特殊命令，它会令循环立即从 BEGIN 处开始，也就是说这时 WHILE 将不再执行其模块中余下的部分而是立即从开始执行了。也 BREAK 命令一样，COUNTINUE 也应该在 IF 条件语句中出现。如例 19.9 所示。

例 19.9

注意：这里的 COUNTINUE 也是放在 IF 之中的。

输入：

```
1> declare @COUNT int
2> select @COUNT = 1
3> while (@COUNT < 10)
4> begin
5>     select @COUNT = @COUNT + 1
6>     if (@COUNT = 8)
7>         begin
8>             continue
9>         end
10>     else
11>         begin
12>             print "LOOP AGAIN!"
13>         end
14> end
15> print "LOOP FINISHED!"
```

分析：

除了用 CONTINUE 替换了 BREAK 以外，例 19.9 与 19.8 是一样的。与当 COUNT=8 时即出循环不同，本例只是跳到的循环的起始部分继续执行。

使用 WHILE 循环在表中翻阅

SQL SERVER 与与其它数据库系统都有一个特殊类型的对象——游标，它可以让你一次一行地翻阅表中的记录（参见第 13 天）。但是，有一些数据库系统（包括 SQL SERVER PER-SYSTEM 10）都不支持翻阅游标的使用。在例 19.10 中我们给出了如果系统不支持游标是如何使用 WHILE 循环来实现简单的游标功能。

例 19.10

你可以使用 WHILE 循环来一次一个地翻阅表中的记录，TRANSACTION-SQL 所存储的 ROWCOUNT 数值可以告诉 SQL SERVER 一次从查询中返回一条记录，如果你使用的是其它的数据库产品，那么你需要确认一下该产品是否有与之类似的设置，通过设置

ROWCOUNT 的值为 1（默认值为 0，也就是不受限制）SQL SERVER 可以一次只从查询中返回一条记录，你可以使用这条记录来进行你想要的任何操作。当将一个表的内容存入临时表然后将其删除时，你就可以一次选择一行，在你工作结束后将它删除，当表中的所有行都被选择出以后，你就将表中的所有行都删除了。（所以我们说它只实现了非常简单的游标功能）现在让我们来运行一个这个例子。

输入：

```
1> set rowcount 1

2> declare @PLAYER char(30)

3> create table temp_BATTERS (

4> NAME char(30),

5> TEAM int,

6> AVERAGE float,

7> HOMERUNS int,

8> RBIS int)

9> insert temp_BATTERS

10> select * from BATTERS

11> while exists (select * from temp_BATTERS)

12> begin

13>     select @PLAYER = NAME from temp_BATTERS

14>     print @PLAYER

15>     delete from temp_BATTERS where NAME = @PLAYER

16> end

17> print "LOOP IS DONE!"
```

分析：

注意，当你设置了 ROWCOUNT 的数值以后，你只是改变了从 SELECT 语句中返回的行数。如果 DELETE 命令的 WHERE 子句返回了 5 条记录的话，那么 5 行都将被删除，同时也要注意 ROWCOUNT 也可以在循环重设，因此，在这个循环中，你可以通过重新设置 ROWCOUNT 的值来查询数据库中的一些其它的信息。

TRANSACT-SQL 中的通配符

在 SQL 中使用通配符的内容是在第 3 天的《表达式、条件与操作》中介绍的。LIKE 可以让你在 SQL 语句中使用通配符，通配符的使用增加了灵活性，在 TRANSACT-SQL 中可供使用的通配符如下：

- 下划线可以代表任何单个的字符，例如：_MITH 就是查询 5 个字符并且是以 MITH 结尾的。
- 百分号可以代表一个或多个字符，例如：WILL% 通配符可以返回 WILLIAMS 和 WILL。
- 中括号可以匹配在括号内的字符，例如：[ABC] 可以查询包括 A、B、C 的字符串。
- 如果在中括号中使用了脱字符^，那就是说匹配所有的字符但不包括中括号中给出的。例如：[^ABC] 表示查询所有的字符，除了 A、B、C。

使用 COMPUTE 来生成摘要报告

TRANSACT-SQL 也具有生成摘要报告的能力，其命令为 COMPUTE。它的语法与它在 SQL*PLUS 的语法极为相似（见第 20 天的《SQL*PLUS》）

下边的查询将会产生关于所有击球手的报告，包括每个击球手的本垒数和所有击球手总的本垒数。

输入：

```
select name, homeruns from batters compute sum(homeruns)
```

分析：

在上一个例子中，COMPUTE 独自对报告进行了估算，然而 COMPUTE BY 则可以对整个报告进行分组评估。如下例所示：

语法：

```
COMPUTE FUNCTION(expression) [BY expression]
```

where the FUNCTION might include SUM, MAX, MIN, etc. and

EXPRESSION is usually a column name or alias.

日期转换

SYBASE 与 MICROSOFT SQL SERVER 可以向表中插入不同格式的日期，它们也可以用不同的格式来表达日期，在这一部分将告诉你如何使用 SQL SERVER 的 CONVERT 来实现多种形式的日期显示方法：

语法：

CONVERT (datatype [(length)], expression, format)

在 SQL SERVER 中使用 CONVERT 时有下列日期格式是可用的：

Format code	Format picture
100	mon dd yyyy hh:miAM/PM
101	mm/dd/yy
102	yy.mm.dd
103	dd/mm/yy
104	dd.mm.yy
105	dd-mm-yy
106	dd mon yy
107	mon dd, yy
108	hh:mi:ss
109	mon dd, yyyy hh:mi:ss:mmAM/PM
110	mm-dd-yy
111	yy/mm/dd
112	yymmdd

输入：

```
select "PayDate" = convert(char(15), paydate, 107) from payment_table where customer_id = 012845
```

输出：

PayDate
May 1, 1997

分析：

在上例中使用的 CONVERT 转换格式中的 107，从上表中可以知道，107 将以 MON、DD、YY 的形式显示日期。

SQL SERVER 的诊断工具——SET 命令

TRANSACT-SQL 提供了一系列的 SET 命令可以让你打开不同的选项以帮助分析 TRANSACT-SQL 的语句，这里给出了一些常见的 SET 命令。

- SET STATISTICS IO ON 可以让服务器返回请示的物理和逻辑页数。

- SET STATISTICS TIME O 可以让服务器返回语句的运行时间。
- SET SHOWPLAN ON 可以让服务器返回当前正在运行的计划中的查询。
- SET NOEXEC ON 可以让服务器编译设计过的查询但不运行。
- SET PARSONLY ON 可以让服务器对所设计的查询进行语法检查，但并不运行。

TRANSACT-SQL 也提供下边的命令来帮助你对输出的显示进行控制。

- SET ROWCOUNT N 可以让服务器只返回查询中的前 N 行。
- SET NOCOUNT ON 不必报告查询所返回的行数。

注：如果你对 TRANSACT-SQL 的语句调整比较关心 SQL，那么请参见第 15 天的《对 SQL 语句的优化以提高性能》。

总结

今天所讨论的主题增加了你对使用 SQL 编程的一些知识，关于基本的 SQL 内容你已经在前些天学习过了，它为你进行数据库编程打下了一个良好的基础。但是，这些内容只是基本的东西，SQL 过程语言的概念已经在昨天解释过了，今天你又学习了一种基本的 SQL。这对于你——数据库程序员——在访问数据库时提供了强大的功能。

在 SYBASE 和 MICROSOFT 的 SQL SERVER 中的 TRANSACT-SQL 提供了许多你可以在第三代编程语言和第四代编程语言中可以找到的编程结构。这包括 IF 条件与 WHILE 循环以及局部和全局变量的定义能力。

需要明白的是今天所介绍的内容只是 TRANSACT-SQL 的基本特性和技术，并对所有你可用的工具有了一个感性的认识，对于它的更详细的内容你需要参见 MICROSOFT SQL SERVER 的 TRANSACT-SQL 文档。

问与答

问：在 SQL 中是否提供了 FOR 循环？

答：像 FOR 循环、WHILE 循环以及 CASE 分支等都是对 ANSI 标准的 SQL 的扩展。所以对于不同的数据库系统它们的使用方法有很大的不同。例如：在 ORACLE 中提供了 FOR 循环，而在 TRANSACT-SQL 中就没有提供（当然，可以在 WHILE 循环都使用一个变量来模拟 FOR 循环）。

问：如果我写一个 GUI 风格的 WINDOWS（或 MACINTOSH）应用程序，其中有对话框之类的元素。这时我是否可以使用 PRINT 语句来向用户输出信息？

答：SQL 与平台是完全独立的，所以 PRINT 语句不能将信息输出至对话框中，如果你想输出信息给用户，你的 SQL 过程将会返回一个预定义的值来表示成功或失败，而用户则可以从查询的状态中得到通知（PRINT 命令在调试时最有用，因为在存储过程中的 PRINT 命令将不会总是向屏幕输出。）

校练场

问：在 ORACLE 的 PL/SQL 中与在 TRANSACT-SQL 中 SQL 字的使用方法与 ANSI 标准的 SQL 是完全相同的，对不对？

问：静态的 SQL 比动态的 SQL 灵活性差，尽管它的性能要比动态的好，对不对？

练习

- 1、如果你没有使用 SYBASE 或 MICROSOFT 的 SQL SERVER，那么请你比较一个你的产品对 SQL 的扩展与今天所讲的有何不同？
- 2、写一组 SQL 语句，它可以对一些已知的条件进行检测，如果条件为真，执行一些操作，否则的话执行另一些操作？

第 20 天：SQL*PLUS

目标

今天你将会学习 SQL*PLUS，这种 SQL 是针对 ORACLE 的 RDBMS 的。在今天的结束，你将会明白 SQL*PLUS 的下述内容。

- 如何使用 SQL*PLUS 的缓存
- 如何格式化报表
- 如何操作日期
- 如何创建交互式查询
- 如何构建高级报表
- 如何使用强大的 DECODE 函数

简介

我们介绍 SQL*PLUS 的原因是由于 ORACLE 数据库在关系数据库市场中所处的优势地位以及 SQL*PLUS 为用户提供的强大功能及灵活性。SQL*PLUS 在许多方面与 TRANSACT-SQL 类似，（见第 19 天《TRANSACT-SQL 简介》）。它们都实现了 ANSI 标准的 SQL（它仍是所有 SQL 解释器的骨架）的大部能力。

SQL*PLUS 命令可以增强 SQL 会话能力并对从数据库返回的查询的结果进行格式上的增强。它也可以像一个专业化的报表生成器对报表进行格式化。SQL*PLUS 对 ANSI 标准的 SQL 和 PL/SQL 进行了补充以帮助关系型数据库程序员们取得满意的数据格式。

SQL*PLUS 缓存

SQL*PLUS 缓存是为你特定的 SQL 会话所指定的命令存储区域。这些命令包括大多数最近执行过的命令以及你用于定制 SQL 会话的如格式化命令和变量赋值之类的命令。缓存就像随机存储器一样。这里给出了一些对缓存的最常用的命令。

- LIST line_number ——可以列出缓存中的命令并可以通过行号将它指定为当前行。

- CHANGE/old_value/new_value ——将缓存当前行的旧数值改为新数值。
- APPEND text ——向缓存所在的当前行中追加文本。
- DEL ——将缓存中的当前行删除。
- SAVE newfile ——将缓存中的 SQL 语句保存到文件中。
- GET filename ——将某文件中的内容送到缓存中。
- / ——运行缓存中的语句。

让我们用一个简单的 SQL 语句来开始：

输入：

```
SQL> select * from products where unit_cost > 25;
```

输出：

PRO	PRODUCT_NAME	UNIT_COST
P01	MICKEY MOUSE LAMP	29.95
P06	SQL COMMAND REFERENCE	29.99
P07	BLACK LEATHER BRIEFCASE	99.99

LIST 命令可以列出在缓存中最近执行过的命令，输出是非常简单的。

```
SQL> list
```

```
1  select *
2  from products
3* where unit_cost > 25
```

分析：

注意，在每一行的前边都有一个数字，这个数字对于缓存来说非常重要，它就像一个指针一样可以让你通过 SQL*PLUS 来对指定行进行修改。SQL*Plus buffer 不是全屏幕编辑的，当你按下回车后，你不能使用游标来回到上一行。如下例所示：

输入：

```
SQL> select *
2  from products
3  where unit_cost > 25
4  /
```

注：与 SQL 命令一样，你在 SQL*PLUS 中既可以用大写字符也可以用小写字符。

技巧：SQL*PLUS 中的大部分命令都可以缩写，例如 LIST 可以缩写为 L。

你可以在 L 后边加上一个数字来跳转到缓存中的指定行。

输入：

```
SQL> l3
```

```
3* where unit_cost > 25
```

分析：

注意 3 后边有一个星号，星号表明了当前的行号，在今天的例子中你应该注意星号，如果一行被标为*号，那么你就可以对它进行编辑。

由于你知道了当前的行是第 3 行，你可以对它进行随意的改变。CHANGE 的语法格式如下：

语法：

```
CHANGE/old_value/new_value 或 C/old_value/new_value
```

输入：

```
SQL> c/>/<
```

输出：

```
3* where unit_cost < 25
```

输入：：

```
SQL> l
```

输出：

```
1 select *
```

```
2 from products
```

```
3* where unit_cost < 25
```

分析：

在第 3 行中的小于号已经被改成大于号了。注意在改变以后的新行会被显示，如果你使用 LIST 命令或 L。你就可以看到完整的语句，现在我们来运行一下语句：

输入：

```
SQL> /
```

输出：

PRO	PRODUCT_NAME	UNIT_COST
-----	--------------	-----------

P02	NO 2 PENCILS - 20 PACK	1.99
P03	COFFEE MUG	6.95
P04	FAR SIDE CALENDAR	10.5
P05	NATURE CALENDAR	12.99

分析：

在 SQL>后边的正斜线的意思就是运行处于缓冲区中的任何语句。

输入：

```
SQL>L
```

输出：

```
1 select *
2 from products
3* where unit_cost < 25
```

现在，你可以通常在 SQL>后边输入行号和相应的文字来增加对应行中的内容。当你增加完以后，你就得到了一个完整的语句组，如下例：

输入：

```
SQL> 4 order by unit_cost
SQL> 1
```

输出：

```
1 select *
2 from products
3 where unit_cost < 25
4* order by unit_cost
```

分析：

如果想删除一行比增加一行还容易，例如你输入 del4 就删除了缓冲区中的第 4 行，现在再看一下语句清单，看看语句是否真的删除了。

输入：

```
SQL> DEL4
SQL> 1
```

输出：

```
1 select *
```

```
2 from products
3* where unit_cost < 25
```

加入一行或多行语句的另一种方法是使用 INPUT 命令，就像你在上边的例子中所看到的那样，当前行现在为 3，输入 INPUT 后按回车，然后你就可以输入文本了，这时你每按一次回车就会增加一行，如果你按下两次回车，你就又回到了 SQL> 提示符下。这时如果你看一下语句的列表，就像下边的例子一样，你可以看到第 4 行已经被加入了。

输入：

```
SQL> input
4i and product_id = 'P01'
5i
SQL> l
```

输出：

```
1 select *
2 from products
3 where unit_cost < 25
4 and product_id = 'P01'
5* order by unit_cost
```

如果你想向当前行中追加文本，你可以在 APPEND 命令的后边写上文本，将输出与上一个例子做一下比较，在下边的例子中当前行是第 5 行。

输入：

```
SQL> append desc
```

输出：

```
5* order by unit_cost desc
```

现在使用 LIST 命令来看一下完整的语句：

输入：

```
SQL> l
```

输出：

```
1 select *
2 from products
3 where unit_cost < 25
```

```
4    and product_id = 'P01'
```

```
5* order by unit_cost desc
```

你也可以使用 `CLEAR BUFFER` 来清除缓冲区，就像你在稍后所见到的那样，你可以使用 `CLEAR` 命令来清除指定的缓冲区的内容。

输入：

```
SQL> clear buffer
```

输出：

```
buffer cleared
```

输入：

```
SQL> 1
```

输出：

```
No lines in SQL buffer.
```

分析：

很明显，你不能从空的缓冲区中得到任何信息的，尽管你不是管理人员，但是你却有足够灵活的办法来管理缓冲区中的命令

DESCRIBE 命令

这个命令可以让你不用访问数据字典就可以非常方便地看到表的结构。

语法：

```
DESC[RIBE] table_name
```

现在来看一下我们将要在今天所使用的表的结构。

输入：

```
SQL> describe orders
```

输出：

Name	Null?	Type
ORDER_NUM	NOT NULL	NUMBER(2)
CUSTOMER	NOT NULL	VARCHAR2(30)
PRODUCT_ID	NOT NULL	CHAR(3)
PRODUCT_QTY	NOT NULL	NUMBER(5)
DELIVERY_DATE		DATE

下边的语句将使用 DESC 来代替 DESCRIBE:

输入:

```
SQL> desc products
```

输出:

Name	Null?	Type
PRODUCT_ID	NOT NULL	VARCHAR2(3)
PRODUCT_NAME	NOT NULL	VARCHAR2(30)
UNIT_COST	NOT NULL	NUMBER(8,2)

分析:

DESC 给出了每个列的名字、列中是否必需有数据 (NULL 或 NOT NULL) 以及每个列的数据类型。如果你写了多个查询, 你会发现你很少有不用这个命令的时候, 在相当长的时间中该命令可以帮助你节省许多的编程时间。没有 DESCRIBE 命令, 你将不得不从数据库文档甚至是数据库的操作手册中搜索相应的数据字典来找到你所需要的信息。

SHOW 命令

SHOW 命令显示了当前会话的设置情况, 从命令的格式到你是谁的信息都有。SHOW ALL 则会显示所有的设置, 在这一部分将会讨论最常见的设置。

输入:

```
SQL> show all
```

输出:

appinfo is ON and set to "SQL*Plus"	cmdsep OFF
arraysize 15	compatibility version NATIVE
autocommit OFF	concat "." (hex 2e)
autoprint OFF	copycommit 0
autotrace OFF	copytypecheck is ON
blockterminator "." (hex 2e)	crt ""
btitle OFF and is the 1st few characters	define "&" (hex 26)
of the next SELECT statement	echo OFF
closecursor OFF	editfile "afiedt.buf"
colsep " "	embedded OFF

escape OFF	serveroutput OFF
feedback ON for 6 or more rows	showmode OFF
flagger OFF	spool OFF
flush ON	sqlcase MIXED
heading ON	sqlcode 1007
headsep " " (hex 7c)	sqlcontinue "> "
linesize 100	sqlnumber ON
lno 6	sqlprefix "#" (hex 23)
long 80	sqlprompt "SQL> "
longchunksize 80	sqlterminator ";" (hex 3b)
maxdata 60000	suffix "SQL"
newpage 1	tab ON
null ""	termout ON
numformat ""	time OFF
numwidth 9	timing OFF
pagesize 24	trimout ON
pause is OFF	trimspool OFF
pno 1	tttitle OFF and is the 1st few characters of
recsep WRAP	the next SELECT statement
recsepchar " " (hex 20)	underline "-" (hex 2d)
release 703020200	user is "RYAN"
repheader OFF and is NULL	verify ON
repfooter OFF and is NULL	wrap : lines will be wrapped

SHOW 命令显示了与登录用户有关的详细的设置，如果你是多用户数据库的用户，而你想知道你是如何登录的，那么你可以使用下边的命令。

输入：

```
SQL> show user
```

输出：

```
user is "RYAN"
```

如果你想知道当前 LINE 的大小，你可以输入：

输入：

```
SQL> show linesize
```

输入：

```
linesize 100
```

文件命令

在 SQL*PLUS 中有许多命令可以帮助你来操作文件，这些命令包括创建文件、使用全屏编辑软件来编辑文件以及将输出重定向到一个文件等等。你也会知道在创建了一个 SQL*PLUS 文件如何去运行它。

SAVE、GET、EDIT 命令

SAVE 命令可以将 SQL 缓冲区的内容保存到你所指定名字的文件中，例如：

输入：

```
SQL> select *  
2  from products  
3  where unit_cost < 25  
  
SQL> save query1.sql
```

输出：

```
Created file query1.sql
```

分析：

当文件被保存以后，你可以使用 GET 命令来查看文件，GET 命令与 LIST 命令非常相似，但是 GET 是处理被保存到文件中的 SQL 语句的，而 LIST 则是处理处于缓冲区中的 SQL 语句的。

输入：

```
SQL> get query1
```

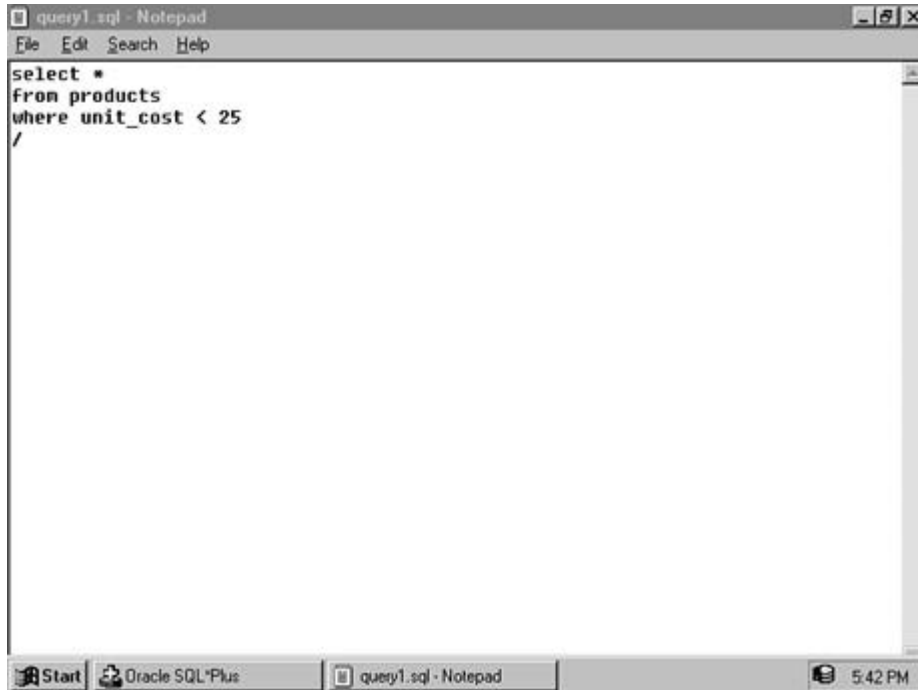
输出：

```
1  select *  
2  from products  
3*  where unit_cost < 25
```


你可以使用 EDIT 命令来创建一个文件或编辑一个已经存在的文件，当运行这个命令以后，你将进入一个全屏幕的编辑环境，它与 WINDOWS 的记事本非常相似，你会发现用它来修改文件要比对缓冲区进行编辑容易得多。特别是当你在处理又大又复杂的语句时，下图给出了 EDIT 命令的例子：

输入：

```
SQL> edit query1.sql
```



运行一个文件

现在你已经知道了如何创建和编辑一个文件，如果你想执行它是非常简单的，你可以使用下边的格式：

语法：

```
START filename 或 STA filename 或 @filename
```

技巧：命令对大小写是不敏感的。

输入：

```
SQL> start query1.sql
```

输出：

PRO	PRODUCT_NAME	UNIT_COST
P02	NO 2 PENCILS - 20 PACK	1.99

P03	COFFEE MUG	6.95
P04	FAR SIDE CALENDAR	10.5
P05	NATURE CALENDAR	12.99

注：在 SQL*PLUS 中你想启动一个文件时不必一定要输入它的扩展名，数据库默认你要执行的文件是有扩展名的，与此类似，当你在 SQL 提示符下创建编辑一个文件如 SAVE、GET、EDIT，你也不必指定文件的扩展名。

输入：

```
SQL> @query1
```

输出：

PRO	PRODUCT_NAME	UNIT_COST
P02	NO 2 PENCILS - 20 PACK	1.99
P03	COFFEE MUG	6.95
P04	FAR SIDE CALENDAR	10.5
P05	NATURE CALENDAR	12.99

输入：

```
SQL> run query1
```

输出：

```
1 select *
2 from products
3* where unit_cost < 25
```

PRO	PRODUCT_NAME	UNIT_COST
P02	NO 2 PENCILS - 20 PACK	1.99
P03	COFFEE MUG	6.95
P04	FAR SIDE CALENDAR	10.5
P05	NATURE CALENDAR	12.99

注意，当你使用 RUN 来运行查询时，语句是会回显到屏幕上的。

查询的假脱机输出

从屏幕上查询输出的结果是非常方便的，但是如果你想把结果保存下来以供将来参考或是想把它打印出来的话。SPOOL 命令可以让你把输出保存到一个指定的文件中，如果这

个文件不存在，它将会被创建，如果它存在，它将会被覆盖，见下图：

输入：

```
SQL> spool prod.lst
```

```
SQL> select *
```

```
2 from products;
```

输出：

PRO	PRODUCT_NAME	UNIT_COST
P01	MICKEY MOUSE LAMP	29.95
P02	NO 2 PENCILS - 20 PACK	1.99
P03	COFFEE MUG	6.95
P04	FAR SIDE CALENDAR	10.5
P05	NATURE CALENDAR	12.99
P06	SQL COMMAND REFERENCE	29.99
P07	BLACK LEATHER BRIEFCASE	99.99

输入：

```
SQL> spool off
```

```
SQL> edit prod.lst
```

在下图中的输出是一个 SQL*PLUS 文件，你必须使用 SPOOL OFF 才能停止向一个文件中的重定向输出。当你退出 SQL*PLUS 时，SPOOL OFF 会自动执行，但是如果你没有退出 SQL*PLUS，那么重定向将一直持续到你使用 SPOOL OFF 时为止。

```

prod - Notepad
File Edit Search Help
SQL> select *
2 from products;

PRO PRODUCT_NAME          UNIT_COST
-----
P01 MICKEY MOUSE LAMP      29.95
P02 NO 2 PENCILS - 20 PACK 1.99
P03 COFFEE MUG             6.95
P04 FAR SIDE CALENDAR      10.5
P05 NATURE CALENDAR        12.99
P06 SQL COMMAND REFERENCE  29.99
P07 BLACK LEATHER BRIEFCASE 99.99

7 rows selected.
SQL> spool off
  
```

SET 命令

在 ORACLE 的 SQL*PLUS 中 SET 命令可以改变对会话的设置。通过使用这些命令，你可以定制你的工作环境并使它的输出更符合你的要求。你可以通过 SET 命令来把相应的选项打开和关闭。

为了演示 SET 命令是如何工作的，可以简单地执行一下 SELECT 命令。

输入：

```
SQL> select *  
      2  from products;
```

输出：

PRO	PRODUCT_NAME	UNIT_COST
P01	MICKEY MOUSE LAMP	29.95
P02	NO 2 PENCILS - 20 PACK	1.99
P03	COFFEE MUG	6.95
P04	FAR SIDE CALENDAR	10.5
P05	NATURE CALENDAR	12.99
P06	SQL COMMAND REFERENCE	29.99
P07	BLACK LEATHER BRIEFCASE	99.99

7 rows selected.

分析：

输出的最后一行为：

7 rows selected.

它被称为 FEEDBACK，它可以通过 SQL 的设置来更改，默认的设置 ON。如果你想更改，你可以输入：

```
SET FEEDBACK ON
```

当运行 SELECT 语句时，假如你不想看到回显，如下例：

输入：

```
SQL> set feedback off  
  
SQL> select *  
      2  from products;
```

输出：

PRO	PRODUCT_NAME	UNIT_COST
P01	MICKEY MOUSE LAMP	29.95
P02	NO 2 PENCILS - 20 PACK	1.99
P03	COFFEE MUG	6.95
P04	FAR SIDE CALENDAR	10.5
P05	NATURE CALENDAR	12.99
P06	SQL COMMAND REFERENCE	29.99
P07	BLACK LEATHER BRIEFCASE	99.99

分析：

在输出中这一列已经被去除了，只有事实上的数据被显示。

你可以有大量的设置来控制你的输出的显示方式。其中一个选项为 `LINESIZE` 可以让你指定每一行的输出长度，如果行长小的时候你的输出换行可能性就大。如果行长超过了默认的 80 以后增加行长可能是必要的。如果你使用的是宽行打印纸，通过增加行的宽度可以让你的打印输出更加漂亮。下边的例子显示了 `LINESIZE` 的用法。

输入：

```
SQL> set linesize 40
```

```
SQL> /
```

输出：

```
P01 MICKEY MOUSE LAMP
29.95

P02 NO 2 PENCILS - 20 PACK
1.99

P03 COFFEE MUG
6.95

P04 FAR SIDE CALENDAR
10.5

P05 NATURE CALENDAR
12.99

P06 SQL COMMAND REFERENCE
29.99

P07 BLACK LEATHER BRIEFCASE
```

99.99

分析：

你也可以通过设置 PAPERSIZE 的大小来调节每一页的尺寸。如果你只是想在屏幕上看到输出的结果，那么最好将 PAPERSIZE 设置为 23，这样在显示多页的时候不会有分页情况。在下边的例子中 PAGESIZE 设置的比较小，你可以清楚地看到它的分页情况。

输入：

```
SQL> set linesize 80
```

```
SQL> set heading on
```

```
SQL> set pagesize 7
```

```
SQL> /
```

输出：

PRO	PRODUCT_NAME	UNIT_COST
P01	MICKEY MOUSE LAMP	29.95
P02	NO 2 PENCILS - 20 PACK	1.99
P03	COFFEE MUG	6.95
P04	FAR SIDE CALENDAR	10.5

PRO	PRODUCT_NAME	UNIT_COST
P05	NATURE CALENDAR	12.99
P06	SQL COMMAND REFERENCE	29.99
P07	BLACK LEATHER BRIEFCASE	99.99

分析：

通过将 PAGESIZE 设置为 7，现在在每一页上显示的最大行数为 7，在每一页都会自动地打印出新的列标头。

TIME 则会把时间作为 SQL 提示符的一部分显示。

输入：

```
SQL> set time on
```

输出：

```
08:52:02 SQL>
```

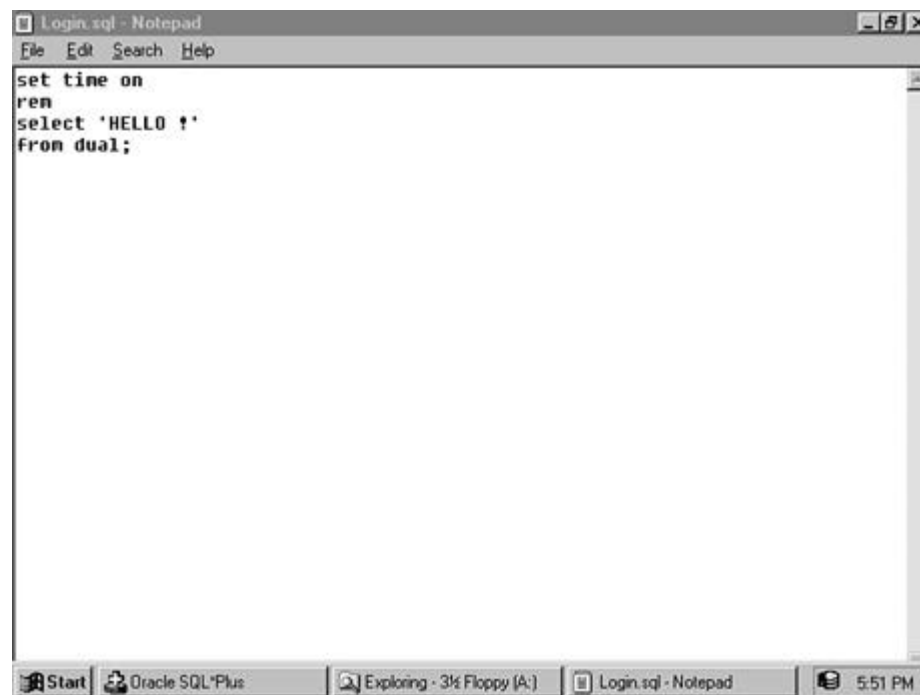
这些只是 SET 选项中很少的一部分，但的所有操作基本上是相同的。你已经在 SHOW ALL 中看到的巨大的 SET 选项的列表，要定制 SQL 会话你有相当多的选项。你最好把它

们中的每一个都试一下。对于许多选项你也许会使用它们的默认设置。但是你会基于自己的要求而经常地改变其中的某些选项。

LOGIN.SQL 文件

在你退出 SQL*PLUS 时，你所有的会话设置都会被清除，而你在重新登录时，如果你不使用 LOGIN.SQL 文件你就不得不对它进行重新的初始化工作。这个文件会在你登录到 SQL*PLUS 时自动运行，它与计算机中的 AUTOEXEC.BAT 文件或 UNIX KORN SHELL 下的 .PROFILE 类似。

在 PERSONAL ORACLE 7 中你可以使用 EDIT 命令来创建你自己的 LOGIN.SQL 文件，见下图：



当你在登录到 SQL*PLUS 以后，你将会看到：

SQL*Plus: Release 3.3.2.0.2 - Production on Sun May 11 20:37:58 1997

Copyright (c) Oracle Corporation 1979, 1994. All rights reserved.

Enter password: ****

Connected to:

Personal Oracle7 Release 7.3.2.2.0 - Production Release

With the distributed and replication options

PL/SQL Release 2.3.2.0.0 - Production

```
'HELLO!
```

```
-----
```

```
HELLO !
```

```
20:38:02 SQL>
```

CLEAR 命令

在 SQL*PLUS 中，设置是在 LONGOUT 或退出 SQL*PLUS 后自动清除的，而其中的一些设置你也可以用 CLEAR 命令来清除。如下例所示：

输入：

```
SQL> clear col
```

输出：

```
columns cleared
```

输入：

```
SQL> clear break
```

输出：

```
breaks cleared
```

输入：

```
SQL> clear compute
```

输出：

```
computes cleared
```

将你的输出格式化

SQL*PLUS 中也有许多的命令可以让你将你的输出格式化。这一部分包括了报表头、列标头及格式的基本的格式化命令。

TTITLE 与 BTITLE

TTITLE 与 BTITLE 可以让你创建报表的报头，在前些天我们提到过查询和输出，但是使用 SQL*PLUS 你也可以将结果输出到预设的报表中。TTITLE 命令可以让你在每一页

输出的顶部放置页眉，BTITLE 则可以在每一页的底部设置页脚。对于这些命令有许多的选项可以使用。但是今天我们只讲一些基本的。TTITLE 与 BTITLE 的语法格式如下：

语法：

```
TTITLE [center|left|right] 'text' [&variable] [skip n]
```

```
BTITLE [center|left|right] 'text' [&variable] [skip n]
```

输入：

```
SQL> title 'A LIST OF PRODUCTS'
```

```
SQL> btitle 'THAT IS ALL'
```

```
SQL> set pagesize 15
```

```
SQL> /
```

输出：

```
Wed May 07
```

```
page      1
```

A LIST OF PRODUCTS		
PRO	PRODUCT_NAME	UNIT_COST
P01	MICKEY MOUSE LAMP	29.95
P02	NO 2 PENCILS - 20 PACK	1.99
P03	COFFEE MUG	6.95
P04	FAR SIDE CALENDAR	10.5
P05	NATURE CALENDAR	12.99
P06	SQL COMMAND REFERENCE	29.99
P07	BLACK LEATHER BRIEFCASE	99.99

```
THAT IS ALL
```

```
7 rows selected.
```

分析：

标题会显示在每一页的顶部和底部，许多人会使用页脚来标记报表数据的改变。使用页眉来标记报表的日期和页数。

格式化列（COLUMN、HEADING、FORMAT）

格式化的列是从 SELECT 语句中返回的列。命令 column、heading、format 可以对从报表中返回的数据的列标头进行重命名并控制数据显示的方式。

在 HEADING 与 FORMAT 命令中常使用 COL[NUM], COLUMN 用于定义你想格式化的列。你所定义的列必须在 SELECT 语句中要出现。在这个命令中你也可以列的别名来代替它的全名。

当使用 HEADING 命令时，你必须使用 COLUMN 命令来定义要列在 HEADING 中的布置。

当使用 FORMAT 命令时，你也必须使用 COLUMN 命令来定义你想要格式化的列。

使用这三个列的基本语法见下，注意 HEADING 与 FORMAT 命令是可选的。在 FORMAT 的语法中，如果数据使用的字符类型或 0 到 9 的数字时你必须要使用一个 a，DECIMAL 也可以会用于数字的值上，在 a 的右边是你所允许的数字的宽度。

语法：

```
COL[UMN] column_name HEA[DING] "new_heading" FOR[MAT] [a|99.99]
```

下边的 SELECTY 语句显示了列的格式，所指定的列是数值类型的，你想显示的数据格式为十进制带美元符号的。

输入：

```
SQL> column unit_cost heading "PRICE" format $99.99;
```

```
SQL> select product_name, unit_cost from products;
```

输出：

PRODUCT_NAME	PRICE
MICKY MOUSE LAMP	\$29.95
NO 2 PENCILS - 20 PACK	\$1.99
COFFEE MUG	\$6.95
FAR SIDE CALENDAR	\$10.50
NATURE CALENDAR	\$12.99
SQL COMMAND REFERENCE	\$29.99
BLACK LEATHER BRIEFCASE	\$99.99

分析：

因为我们使用了格式 99.99，所以允许显示的最大数字为 99.99。

现在试着简化一下这个命令，这可以让你的 HEADING 命令显得整洁。

输入：

```
SQL> col unit_cost hea "UNIT|COST" for $09.99
```

```
SQL> select product_name, unit_cost from products;
```

输出：

PRODUCT_NAME	UNITCOST
MICKEY MOUSE LAMP	\$29.95
NO 2 PENCILS - 20 PACK	\$01.99
COFFEE MUG	\$06.95
FAR SIDE CALENDAR	\$10.50
NATURE CALENDAR	\$12.99
SQL COMMAND REFERENCE	\$29.99
BLACK LEATHER BRIEFCASE	\$99.99

分析：

在 HEADING 命令中的管道符号 (|) 可以强制在它后边的文本跳到下一个输出区。你也许会使用多个管道符号，当你的报表受到最大行长的限制时这是很方便的。注意单价的格式是 09.99。最大的允许显示的数字仍然是 99.99。但是现在零会出现在所有的小于 10 的数字的前边。你也许会更喜欢这种格式因为它会使美元的格式看起来更整齐一些。

报表与分类汇总

如果报表没有汇总和计算还有什么用呢？那只能说明我们是一个失败的程序员。在 SQL*PLUS 中有一些命令可以让你将报表的数据进行分组并对每一组进行汇总计算。BREAK 与标准的 SQL 的分组功能有一些不同。如 COUNT () 和 SUM ()。这些函数在报表和分类汇总中应用以提供更完备的报表。

BREAK ON

BREAK ON 命令可以把从 SQL 中返回的数据分成一个或多个组。如果你依据用户的名字进行分类，那么默认的客户名字将只会打印一次，以后对应的客户名字就以空白代替了。下边的 BREAK ON 的基本使用语法：

语法：

```
BRE[AK] [ON column1 ON column2...][SKIP n|PAGE][DUP|NODUP]
```

你也可以使用 BREAK ON REPORT 或 BREAK ON ROW。ON REPORT 时会对整个报表进行计算。ON ROW 时则是对每一组的行进行计算。

SKIP 选项可以让你跳过每一组的指定行数或页数。DUP 或 NODUP 可以让你确定是

否进行双面打印。默认是单面的。

见下例：

输入：

```
SQL> col unit_cost head 'UNIT|COST' for $09.99

SQL> break on customer

SQL> select o.customer, p.product_name, p.unit_cost
2   from orders o,
3        products p
4  where o.product_id = p.product_id
5  order by customer;
```

输出：

CUSTOMER	PRODUCT_NAME	UNITCOST
JONESandSONS	MICKEYMOUSELAMP	\$29.95
	NO2PENCILS-20PACK	\$01.99
	COFFEEMUG	\$06.95
PARAKEETCONSULTINGGROUP	MICKEYMOUSELAMP	\$29.95
	NO2PENCILS-20PACK	\$01.99
	SQLCOMMANDREFERENCE	\$29.99
	BLACKLEATHERBRIEFCASE	\$99.99
	FARSIDECALNDAR	\$10.50
PLEWSKYMOBILECARWASH	MICKEYMOUSELAMP	\$29.95
	BLACKLEATHERBRIEFCASE	\$99.99
	BLACKLEATHERBRIEFCASE	\$99.99
	NO2PENCILS-20PACK	\$01.99
	NO2PENCILS-20PACK	\$01.99

每一个客户只被打印了一次，当打印多个客户名时这个报表是非常易读的，排序是与你使用 BREAK 时所指定的排序列是相同的。

COMPUTE

在 BREAK ON 命令中也可以使用 COMPUTE 命令，它可以让你对整个报表或报表中的每个组进行不同的计算。

语法：

COMP[UTE] function OF column_or_alias ON column_or_row_or_report

常用的一些函数有：

- AVG-对每个组的平均值进行计算。
- COUNT-计算每个组中的数据个数。
- SUM-计算每个组的总计结果。

如果你想创建一个报表，它打印 PRODUCT 表的内容并在报表中对产品的平均价格进行计算。

输入：

```
SQL> break on report
```

```
SQL> compute avg of unit_cost on report
```

```
SQL> select *
```

```
2 from products;
```

输出：

PRO	PRODUCT_NAME	UNIT_COST
P01	MICKEY MOUSE LAMP	29.95
P02	NO 2 PENCILS - 20 PACK	1.99
P03	COFFEE MUG	6.95
P04	FAR SIDE CALENDAR	10.50
P05	NATURE CALENDAR	12.99
P06	SQL COMMAND REFERENCE	29.99
P07	BLACK LEATHER BRIEFCASE	99.99
Avg		27.48

分析：

你可以通过对报表的分组和对平均价格的计算来得到你想要的信息。

还记得 CLEAR 命令吗？现在清除上一次的计算结果并再运行一次，这次你想计算的是每个客户的付费情况，由于你不想再见到平均值，所以你应该清除计算的结果。

输入：

```
SQL> clear compute
```

输出：

```
computes cleared
```

现在，将上一次的 BREAK 清除（在这种情况下你其实不需要 BREAK 因为你还是想

要对报表进行 BREAK 的)。

输入：

```
SQL> clear break
```

输出：

```
breaks cleared
```

然后你按照你的需要对它们重新进行 BREAK 和计算，你也许需要重新格式化 COST_UNIT 列以使得当你进行求和时它可以接受更大的数字。你需要给它更大的显示空间，这样它们看起来才会漂亮。所以你需要在十进数的左边增加一些空间。

输入：

```
SQL> col unit_cost hea 'UNIT|COST' for $099.99
```

```
SQL> break on report on customer skip 1
```

```
SQL> compute sum of unit_cost on customer
```

```
SQL> compute sum of unit_cost on report
```

现在，我们来看一下在缓冲区中的上一个语句。

输入：

```
SQL> 1
```

输出：

```
1  select o.customer, p.product_name, p.unit_cost
2  from orders o,
3       products p
4  where o.product_id = p.product_id
5* order by customer
```

现在你可以验证情况是否如你所期望的那样，你可以运行它：

输入：

```
SQL> /
```

输出：

CUSTOMER	PRODUCT_NAME	UNIT COST
JONESandSONS	MICKEY MOUSE LAMP	\$029.95
	NO 2 PENCILS-20PACK	\$001.99

	COFFEEMUG	\$006.95
sum		\$038.89
PARAKEET CONSULTING GROUP	MICKEY MOUSE LAMP	\$029.95
	NO 2 PENCILS-20PACK	\$001.99
	SQL COMMAND REFERENCE	\$029.99
	BLACK LEATHER BRIEFCASE	\$099.99
	FAR SIDE CALENDAR	\$010.50
sum		\$172.42
PLEWSKY MOBILE CARWASH	MICKEYMOUSELAMP\$029.95	
	BLACK LEATHER BRIEFCASE	\$099.99
	BLACK LEATHER BRIEFCASE	\$099.99
	NO 2 PENCILS-20PACK	\$001.99
	NO 2 PENCILS-20PACK	\$001.99
CUSTOMER	PRODUCT_NAME	UNIT
		COST
sum		\$233.91
sum		\$445.22

分析:

这个例子对每个客户的付款情况进行了计算总对所以客户的付款情况进行了汇总。

现在你应用明白的基本的列格式命令、数据分组命令和对每一组执行计算的内容了。

在 SQL*PLUS 中使用变量

尽管没有进行事实上的程序设计语言中，但是你仍然可以在你的 SQL 语句中定义变量，在 SQL*PLUS 中你可以使用一些特定的选项（将在这一部分中讲述）来在你的程序中设定参数来接受用户的输入。

置换变量（&）

在 SQL 脚本中&表示变量的值，如果变量没有进行预定义，那么用户会收到输入数值的提示。

输入:

```
SQL> select * from &TBL
```

```
1 /
```

```
Enter value for tbl: products
```

```
用户输入数值"products."
```

输出：

old 2: from &TBL

new 2: from products

PRO	PRODUCT_NAME	UNIT_COST
P01	MICKEY MOUSE LAMP	29.95
P02	NO 2 PENCILS - 20 PACK	1.99
P03	COFFEE MUG	6.95
P04	FAR SIDE CALENDAR	10.5
P05	NATURE CALENDAR	12.99
P06	SQL COMMAND REFERENCE	29.99
P07	BLACK LEATHER BRIEFCASE	99.99

分析：

在这个交互式查询中 PRODUCT 替换了原来的&TBL。

DEFINE

在 SQL 的脚本中你可以使用 DEFINE 来对一个变量赋值，如果在你的 SQL 脚本中定义了变量，那么在运行时用户不会像你使用&时那样被提示输入数值，在下一个例子中使用与上一个例子相同的 SELECT 语句，但是这次 TBL 的值是在脚本中定义的。

输入：

SQL> define TBL=products

SQL> select * from &TBL;

输出：

old 2: from &TBL

new 2: from products

PRO	PRODUCT_NAME	UNIT_COST
P01	MICKEY MOUSE LAMP	29.95
P02	NO 2 PENCILS - 20 PACK	1.99
P03	COFFEE MUG	6.95
P04	FAR SIDE CALENDAR	10.5
P05	NATURE CALENDAR	12.99
P06	SQL COMMAND REFERENCE	29.99
P07	BLACK LEATHER BRIEFCASE	99.99

分析：

这两个查询的结果是相同的，下边的部分将向您介绍对于脚本参数的另外一种向用户提示的办法。

ACCEPT

ACCEPT 可以在运行时允许用户向变量中输入数值，它所作的工作与没有 DEFINE 的 &相同，但它的可控性更好，它可以给用户一个更友好提示。

下边的例子将从清除缓冲区开始。

输入：

```
SQL> clear buffer
```

输出：

```
buffer cleared
```

然后我们使用 INPUT 命令来向缓冲区中输入 SQL 语句，如果你输入语句时没有从 INPUT 开始，你开始会被提示为 NEWTITLE 输入一个值。当然，你也可以建立一个新文件并在其中输入你的语句。

输入：

```
SQL> input
1 accept newtitle prompt 'Enter Title for Report: '
2 title center newtitle
3 select *
4 from products
5
SQL> save prod
```

输出：

```
File "prod.sql" already exists.
Use another name or "SAVE filename REPLACE".
```

分析：

咦？这个文件怎么已经存在了？这就是说你已经有一个叫 PROD.SQL 的文件了，如果你不想保留它，你可以使用覆盖选项来保存缓冲区中的内容。注意在上一个语句中的

PROMPT, 它将会向用户显示文本信息以准确地告诉用户应该输入什么。

输入:

```
SQL> save prod replace
```

输出:

```
Wrote file prod
```

现在你可以使用 START 命令来运行这个文件了。

输入:

```
SQL> start prod
```

```
Enter Title for Report: A LIST OF PRODUCTS
```

输出:

A LIST OF PRODUCTS	
PRO PRODUCT_NAME	UNIT_COST
P01 MICKEY MOUSE LAMP	29.95
P02 NO 2 PENCILS - 20 PACK	1.99
P03 COFFEE MUG	6.95
P04 FAR SIDE CALENDAR	10.5
P05 NATURE CALENDAR	12.99
P06 SQL COMMAND REFERENCE	29.99
P07 BLACK LEATHER BRIEFCASE	99.99

```
7 rows selected.
```

分析:

你输入的文字成了当前报表的题头。

下边的例子向你显示了你如果替换在语句中的任何变量的值。

输入:

```
SQL> input
```

```
1 accept prod_id prompt 'Enter PRODUCT ID to Search for: '
```

```
2 select *
```

```
3 from products
```

```
4 where product_id = '&prod_id'
```

```
5
```

```
SQL> save prod1
```

输出：

Created file prod1

输入：

SQL> start prod1

Enter PRODUCT ID to Search for: P01

输出：

old 3: where product_id = '&prod_id'

new 3: where product_id = 'P01'

A LIST OF PRODUCTS

PRO PRODUCT_NAME	UNIT_COST
P01 MICKEY MOUSE LAMP	29.95

分析：

你可以在许多的需要中见到变量的使用，例如：为你想要重新定向输出的文件命名或为 ORDER BY 子句指定一个表达式。其中的方法之一是使用变量替换，一种使用变量替换的情况是在事务处理诊断报告中在 WHERE 子句中输入日期。如果你的查询设计成要求在特定的时间内得到信息的话，你也许需要设置一个替换变量来与表中的 SSN 列进行比较。

NEW_VALUE

NEW_VALUE 命令可以将 SELECT 语句中的数值返回给一个没有经你明确定义的变量。语法格式如下：

语法：

COL[UMN] column_name NEW_VALUE new_name

你可以使用&符号来调用这个值，例如：

&new_name

在 NEW_VALUE 命令中必须使用 COLUMN 命令。

注意&符号与 COLUMN 在下边的 SQL*PLUS 文件中是一起使用的，在这个文件中出现了 GET 命令。

输入：

SQL> get prod1

输出：

line 5 truncated.

```
1  title left 'Report for Product:  &prod_title' skip 2
2  col product_name new_value prod_title
3  select product_name, unit_cost
4  from products
5* where product_name = 'COFFEE MUG'
```

输入：

SQL> @prod1

输出：

Report for Product: COFFEE MUG

PRODUCT_NAME	UNIT_COST
COFFEE MUG	6.95

分析：

PRODUCT_NAME 列的值已经通过 NEW_VALUE 方法被存入了变量 PROD_TITLE 中。该变量中的值将在稍后被 TTITLE 命令调用。

对于在 SQL 中更多的变量信息，请参见第 18 天的《PL/SQL：简介》以及第 19 天的《TRANSACTION SQL 简介》

DUAL 表

DUAL 表是在每一个 ORACLE 数据库中都存在的虚拟表，它只有一个叫 DUMMY 的列和一行值为 X 的数据。这个表可以由所有的用户出于通用的目标如进行计算（这时它可以像一个计算器一样使用）或维护 SYSDATE 的格式而使用。

输入：

SQL> desc dual;

输出：

Name	Null?	Type
DUMMY		VARCHAR2(1)

输入：

SQL> select * from dual;

输出：

D
X

让我们来看一对使用 DUAL 表的例子。

输入：

```
SQL> select sysdate from dual;
```

输出：

```
SYSDATE  
  
08-MAY-97
```

输入：

```
SQL> select 2 * 2 from dual;
```

输出：

```
2*2
```

非常简单，第一条语句从 DUAL 表中选择了 SYSDATE 以取得今天的日期，第二个例子显示了如何用这个表来进行乘法计算，我们的答案是 2*2 的结果为 4。

DECODE 函数

DECODE 函数是 SQL*PLUS 中众多功能强大的函数之一——也许它的功能是最强大的。标准的 SQL 中没有过程函数，它是包括在如 COBOL 和 C 语言中的。

DECODE 语句与程序语言中的 IF·····THEN 语句类似，对于复杂的报表来说灵活性是必需的，DECODE 可以弥补标准的 SQL 与过程语言函数之间的缺陷。

语法：

```
DECODE(column1, value1, output1, value2, output2, output3)
```

语法中的例子是对 column1 列执行 DECODE 函数，如果 column1 有一个值为 value1，那么将会用 output1 来代替当前值，如果 column1 的值为 value2，那么就会用 OUTPUT2 来代替当前值，如果 column1 中哪两个值都不是，那么就会用 OUTPUT3 来代替当前值。

实际应用的例子呢？我们先来对一个新表运行一下 SELECT 语句：

输入：

```
SQL> select * from states;
```

输入：

ST	IL
IN	OH
FL	CA
KY	NY

7 rows selected.

现在来用一下 DECODE 命令：

输入：

```
SQL> select decode(state,'IN','INDIANA','OTHER') state from states;
```

输出：

STATE	OTHER
INDIANA	OTHER
OTHER	OTHER
OTHER	OTHER

7 rows selected.

分析：

符合条件（州为 IN）的记录只有一条，所以只有一行显示为 INDIANA。其它的记录看上去都不符合，因为它们的显示为 OTHER。

下边的例子为表中的每一个值提供了输入字符串，只有当你的州不在列表中时，你才会收到 OTHER 信息。

输入：

```
SQL> select decode(state,'IN','INDIANA',
2          'FL','FLORIDA',
3          'KY','KENTUCKY',
4          'IL','ILLINOIS',
5          'OH','OHIO',
6          'CA','CALIFORNIA',
7          'NY','NEW YORK','OTHER')
8 from states;
```

输出：

 DECODE(STATE)

INDIANA

FLORIDA

KENTUCKY

ILLINOIS

OHIO

CALIFORNIA

 NEW YORK

7 rows selected.

这实在太容易了，在下一个例子中将引入一个 PAY 表，这个表可以显示职 DECODE 函数更为强大的能力。

输入：

```
SQL> col hour_rate hea "HOURLY|RATE" for 99.00
```

```
SQL> col date_last_raise hea "LAST|RAISE"
```

```
SQL> select name, hour_rate, date_last_raise
```

```
2 from pay;
```

输出：

HOURLY LAST		
NAME	RATE	RAISE
JOHN	12.60	01-JAN-96
JEFF	8.50	17-MAR-97
RON	9.35	01-OCT-96
RYAN	7.00	15-MAY-96
BRYAN	11.00	01-JUN-96
MARY	17.50	01-JAN-96
ELAINE	14.20	01-FEB-97

7 rows selected.

准备好了吗？现在是给在 PAY 表中的每个人长工资的时候了。如果某个人上次长工资的时间是 1997 年，那么将他的工资上浮 10%，如果他上次长工资是在 1996 年，那么将他的工资上浮 20%，此外，还要显示在这两种情况下工资上调的百分率。

输入：

```
SQL> col new_pay hea 'NEW PAY' for 99.00
```

```
SQL> col hour_rate hea 'HOURLY|RATE' for 99.00
```

```
SQL> col date_last_raise hea 'LAST|RAISE'
```

```
SQL> select name, hour_rate, date_last_raise,
2         decode(substr(date_last_raise,8,2),'96',hour_rate * 1.2,
3                                     '97',hour_rate * 1.1) new_pay,
4         decode(substr(date_last_raise,8,2),'96','20%',
5                                     '97','10%',null) increase
6 from pay;
```

输出：

HOURLY LAST				
NAME	RATE	RAISE	NEW PAY	INC
JOHN	12.60	01-JAN-96	15.12	20%
JEFF	8.50	17-MAR-97	9.35	10%
RON	9.35	01-OCT-96	11.22	20%
RYAN	7.00	15-MAY-96	8.40	20%
BRYAN	11.00	01-JUN-96	13.20	20%
MARY	17.50	01-JAN-96	21.00	20%
ELAINE	14.20	01-FEB-97	15.62	10%

7 rows selected.

分析：

根据输出情况，除了 JEFF 和 ELAINE 每个人的工资都上涨了 20%，他们在今年已经涨过工资了。

日期转换

如果你想对日期的显示加一些附加的格式，那么你可以使用 TO_CHAR 来改变日期的显示方式。例子中首先得到的是今天的日期。

输入：

```
SQL> select sysdate from dual;
```

输出：

```
SYSDATE
08-MAY-97
```

当将日期转换为字符串时，你可以使用 TO_CHAR 函数功能的下列语法：

语法：

```
TO_CHAR(sysdate,'date picture')
```

Date picture 是你想要看到的日期形式，最常见的日期形式如下，

Mon	当前月简写
Day	当前日期是本周中的第几天
mm	当前的月份
yy	当前年的最后两位数
dd	当前日期是当前月的第几天
Yyyy	当前年
ddd	当前日期是当前年中的第几天
hh	当前时间是当前天的第几小时
mi	当前时间是当前小时的第几分钟
ss	当前时间是当前分钟的第几秒
a.m.	显示 PM 或 AM

在 date picture 中的引号中也可以包括逗号和其它的文本字符。

输入：

```
SQL> col today for a20
```

```
SQL> select to_char(sysdate,'Mon dd, yyyy') today from dual;
```

输出：

```
TODAY
May 08, 1997
```

分析：

注意在 COLUMN 列中如何使用 today 的别名。

输入：

```
SQL> col today hea 'TODAYs JULIAN DATE' for a20
```

```
SQL> select to_char(sysdate,'ddd') today from dual;
```

输出：

```
TODAYs JULIAN DATE
128
```

分析：

有一些公司更喜欢使用乌利尤斯·恺撒日期并带有两位年份数字的写法，你的日期看

起来就是'yyddd'的样子。

假定你已经写了一个小脚本并假定它的名字为 day，下边的例子将打开这个文件，查看它的内容，并执行它以获得不同格式的日期转换信息。

输入：

```
SQL> get day
```

输出：

```
line 10 truncated.

1  set echo on
2  col day for a10
3  col today for a25
4  col year for a25
5  col time for a15

6  select to_char(sysdate,'Day') day,
7         to_char(sysdate,'Mon dd, yyyy') today,
8         to_char(sysdate,'Year') year,
9         to_char(sysdate,'hh:mi:ss a.m.') time
10* from dual
```

现在，运行一下这个脚本：

输入：

```
SQL> @day
```

输出：

```
SQL> set echo on

SQL> col day for a10

SQL> col today for a25

SQL> col year for a25

SQL> col time for a15

SQL> select to_char(sysdate,'Day') day,
2         to_char(sysdate,'Mon dd, yyyy') today,
3         to_char(sysdate,'Year') year,
4         to_char(sysdate,'hh:mi:ss a.m.') time
```

```
5 from dual;
```

DAY	TODAY	YEAR	TIME
Thursday	May 08, 1997	Nineteen Ninety-Seven	04:10:43 p.m.

分析:

因为 ECHO 设置为 ON, 所以在这个例子中运行之前会显示整个的语句。此外, SYSDATE 会被填入到四列中并被转换为四种格式。

TO_DATE 函数还可以将文本转换为日期格式, 基本的语法与 TO_CHAR 类似。

语法:

```
TO_DATE(expression,'date_picture')
```

Try a couple of examples:

输入:

```
SQL> select to_date('19970501','yyyymmdd') "NEW DATE" from dual;
```

输出:

```
NEW DATE
01-MAY-97
```

输入:

```
SQL> select to_date('05/01/97','mm"/"dd"/"yy') "NEW DATE" from dual;
```

输出:

```
NEW DATE
01-MAY-97
```

分析:

注意, 这里的字符串用引号括起来了。

运行一系列的 SQL 文件

一个 SQL 脚本文件可以包括你在 SQL 提示下输入到缓冲区中的任何内容。甚至可以是运行其它 SQL 脚本的命令, 也就是说你可以从一个 SQL 脚本中执行另一个 SQL 脚本。下图显示了使用 EDIT 命令创建的脚本。这个文件中包括了多条的 SQL 语句和运行其它的 SQL 脚本的命令。

输入:

```
SQL> edit main.sql
```

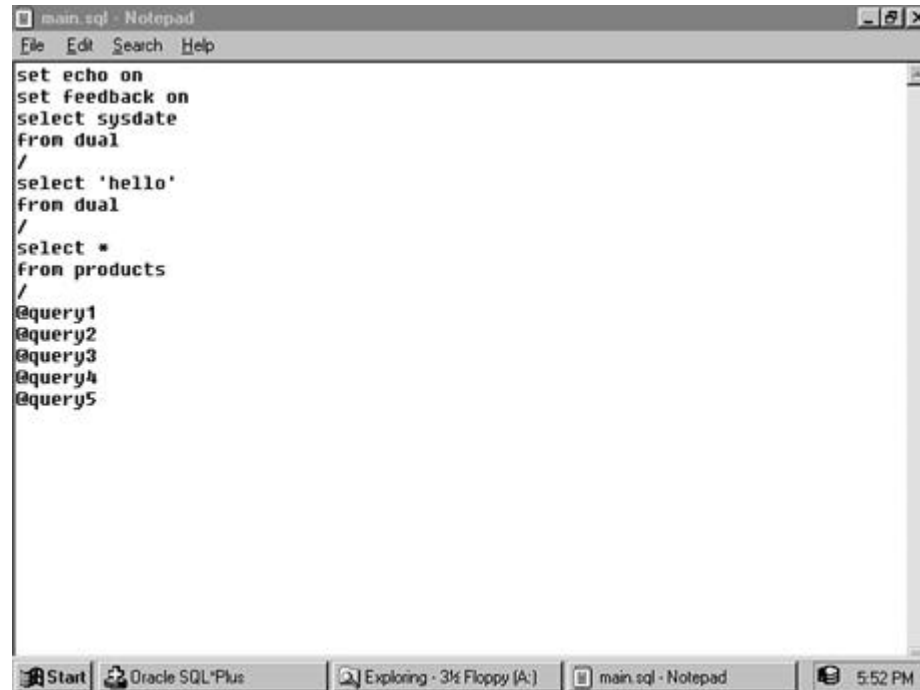
输出：

```
SQL> @main
```

分析：

在启动 main.sql 以后，你可以运行在这个文件中的每一个 SQL 命令，query1 到 query5

将会被执行，次序如下图所示：



```
main.sql - Notepad
File Edit Search Help
set echo on
set feedback on
select sysdate
from dual
/
select 'hello'
from dual
/
select *
from products
/
@query1
@query2
@query3
@query4
@query5
```

在你的 SQL 脚本中加入注释

SQL*PLUS 为你提供了三种在脚本文件中加入注释的方法：

---一次可以注释一行语句。

REMARK 也可以一次注释一行语句。

/* */可以注释多行语句。

请学习下边的例子：

输入：

```
SQL> input
```

- 1 REMARK this is a comment
- 2 -- this is a comment too
- 3 REM
- 4 -- SET COMMANDS

```
5 set echo on
6 set feedback on
7 -- SQL STATEMENT
8 select *
9 from products
10
```

SQL>

如果想看一个脚本文件是如何注释的，请输入：

SQL> edit query10

高级报表

现在我们来做一个游戏，通过今天你已经学习的以及你在此之前学习过的概念，你现在可以创建一个非常奇特的报表，假设你的脚本名字为 report1.sql，运行它以后，坐下来看它的结果：

输入：

SQL> @report1

输出：

```
SQL> set echo on
SQL> set pagesize 50
SQL> set feedback off
SQL> set newpage 0
SQL> col product_name hea 'PRODUCT|NAME' for a20 trunc
SQL> col unit_cost hea 'UNIT|COST' for $99.99
SQL> col product_qty hea 'QTY' for 999
SQL> col total for $99,999.99
SQL> spool report
SQL> compute sum of total on customer
SQL> compute sum of total on report
SQL> break on report on customer skip 1
```

```

SQL> select o.customer, p.product_name, p.unit_cost,
2         o.product_qty, (p.unit_cost * o.product_qty) total
3   from orders o,
4        products p
5  where o.product_id = p.product_id
6  order by customer
7  /

```

CUSTOMER	PRODUCT NAME	UNIT COST	QTY	TOTAL
JONES and SONS	MICKEY MOUSE LAMP	\$29.95	50	\$1,497.50
	NO 2 PENCILS - 20 PA	\$1.99	10	\$19.90
	COFFEE MUG	\$6.95	10	\$69.50
Sum				\$1,586.90
PARAKEET CONSULTING GROUP	MICKEY MOUSE LAMP	\$29.95	5	\$149.75
	NO 2 PENCILS - 20 PA	\$1.99	15	\$29.85
	SQL COMMAND REFERENC	\$29.99	10	\$299.90
	BLACK LEATHER BRIEFC	\$99.99	1	\$99.99
	FAR SIDE CALENDAR	\$10.50	22	\$231.00
Sum				\$810.49
PLEWSKY MOBILE CARWASH	MICKEY MOUSE LAMP	\$29.95	1	\$29.95
	BLACK LEATHER BRIEFC	\$99.99	5	\$499.95
	BLACK LEATHER BRIEFC	\$99.99	1	\$99.99
	NO 2 PENCILS - 20 PA	\$1.99	10	\$19.90
	NO 2 PENCILS - 20 PA	\$1.99	10	\$19.90
Sum				\$669.69
Sum				\$3,067.08

SQL> Input truncated to 9 characters

spool off

分析:

在这个脚本中做了许多的工作，如果你看到了实际的 SQL 语句，你可以知道它是从两个表中选择了数据并对它们进行了计算功能，语句在 WHERE 中归并了两个表并将它们按照客户的名字进行了排序，这只是基本的。此外，SQL*PLUS 按照你所看到的那样对日期进行了格式化处理，BREAK 命令对报表进行了分组，并对每一组和全部的数据进行了汇总。

总结

今天主要讲述了 ORACLE 对标准的 SQL 的扩展，这些命令只是你在 SQL*PLUS 的可用命令中的一小部分，如果你使用 ORACLE 的产品，那么请检查你的数据库文档，那里有你今天学习的内容，而且你还可以从中知道更多的内容。你会发现你几乎可以用 SQL*PLUS 来完成任何报表工作而无需面向过程型语言的帮助。如果你没有使用 ORACLE 的产品，那么今天的所学将会对你从数据库中获得数据的方法得到提高。许多的 SQL 解释器都对标准的 SQL 进行了扩展和增强。

问与答

问：我既然可以直接用 SQL 来得到结果，那么为什么我还要在 SQL*PLUS 上花这么多的时间？

答：如果你需要生成简单的报表，那么直接用 SQL 就行了，但是使用 SQL*PLUS 你可以做得更快，而且可以肯定的是你的报表可以需要的内容是非常多的。

问：如果 DUAL 表没有 COLUMN 我应该如何选择它？

答：因为 SYSDATE 是一个预定义的列，所以你可以从 DUAL 中选择 SYSDATE 或任何其它的有效的表。

问：我是否可以在 DECODE 中使用其它的 DECODE？

答：当然可以，它是可以嵌套的，在 SQL 中你可以在函数中运行其它的函数来得到你想要的结果。

校练场

- 1、哪些命令可以改变你的 SQL 会话的性能？
- 2、你可以在 SQL 的脚本中提示用户输入参数并根据输入的参数运行吗？
- 3、如果你对 CUSTOMERS 表创建了一个汇总报表，你如果在你的报表中对你的数据进行分组？
- 4、你在使用 LOGIN.SQL 文件时有哪些限制？
- 5、DECODE 函数与过程语言中的 LOOP 功能是等价的。对不对？
- 6、如果你将查询重新定向到一个已经存在的文件中，你的输出将追加到这个文件，对不对？

练习

- 1、利用在今天开始时的 PRODUCTS 表，写一个查询选择其中的所有数据并对记录的个数进行汇总，要生成报表并且不得使用 SET FEEDBACK ON 命令。
- 2、假如今天是 1998 年 5 月 12 日星期一，写一个查询产生下边的输出：
Today is Monday, May 12 1998
- 3、试一下下边的语句：

```
1  select *  
2  from orders  
3  where customer_id = '001'  
4* order by customer_id;
```

不需要在缓冲区中重新输入这些语句，将 FROM 子句中的表改为 CUSTOMERS。

在 ORDER BY 子句中加入 DESC。

第 21 天：常见的 SQL 错误及解决方法

目标：

欢迎来到第 21 天，在今天你将学习下边的内容：

- 几种典型的错误及它们的解决方案。
- SQL 用户常犯的逻辑错误。
- 防止再犯日常错误的办法。

介绍

今天你将会看到一些人们（不论是菜鸟还是老鸟）在使用 SQL 常犯的错误，你也不可避免地要犯这些错误，但是对这些错误熟悉了以后会帮助你在尽可能短的时间内解决它们。

注意：我们使用 PERSONAL ORACLE 7 来进行我们的例子，在你的解释器中也会有与之类似的错误，只是错误的代码和名字可能会不相同。我们在 SQL*PLUS 中运行这些语句并将 ECHO 和 FEEDBACK 设置为 ON 以看到这些语句。

要知道一些错误确实可以产生的错误信息，然而逻辑语法的不充分可以在运行时导致更大的错误。通过仔细的校审，你会避免大多数的错误，尽管你会经常被错误卡住。

常见的错误

在这一部分你将会见到在所有的 SQL 语句中看到的一些常见的错误。大多数的错误简单到你甚至想打自己几个耳光，可是也有一些错误看上去很明显但却很容易误码率解。

Table or View Does Not Exist

当你在试图访问一个并不存在的表时产生的错误，这是很明显的，例如：

输入：

```
SQL> @tables.sql
```

输出：

```
SQL> spool tables.lst

SQL> set echo on

SQL> set feedback on

SQL> set pagesize 1000

SQL> select owner|| '.' || table_name
      2  from sys.dba_table
      3  where owner = 'SYSTEM'
      4  order by table_name
      5  /

      from sys.dba_table
      *

ERROR at line 2:

ORA-00942: table or view does not exist

SQL> spool off

SQL>
```

分析：

注意在表的名字后边的星号，正确的名是 sys.dba_tables。表中少了一个 S。

但是如果你已经知道了表是存在的而你仍然收到了错误信息呢？有时你收到这个信息是因为表并不存在，但是也可以是由于安全原因——也就是说，表是存在的，但是你没有权限访问它。这个错误用数据库服务人员的话来精确地说就是“你没有权限来访问这个表”。

技巧：在着急之前，请立即先确认在 DBA 的帐号或可用的工程帐号中这个表是否存在，

你会经常发现表不存在的原因是因为用户没有正当的访问这个表的权限。

Invalid Username or Password

输入：

```
SQL*Plus: Release 3.2.3.0.0 - on Sat May 10 11:15:35 1997

Copyright (c) Oracle Corporation 1979, 1994. All rights reserved.

Enter user-name: rplew

Enter password:
```

输出：

ERROR: ORA-01017: invalid username/password; logon denied

Enter user-name:

这个错误的原因是因为用户名或密码不正确。再试一次，如果还不成功。那么你的密码已经变了，如果你确认你输入的用户名和密码是正确的。那么在你访问多个数据库时确认你要联接的数据库是正确的。

FROM Keyword Not Specified

输入：

SQL> @tblspc.sql

输出：

SQL> spool tblspc.lst

SQL> set echo on

SQL> set feedback on

SQL> set pagesize 1000

SQL> select substr(tablespace_name,1,15) a,

2 substrfile_name, 1,45) c, bytes

3 from sys.dba_data_files

4 order by tablespace_name;

 substrfile_name, 1,45) c, bytes

*

ERROR at line 2:

ORA-00923: FROM keyword not found where expected

SQL> spool off

SQL>

分析：

这个错误容易让人误解，关键字 FROM 是有的，但是你在第二行的 SUBSTR 中缺了一个括号，这个错误也可能是由于在 SELECT 语句中缺少列名导致的，如果在 SELECT 中的列名之后没有逗号，那么查询的处理机制会认为没有 FROM 关键字，上边的语句可以更

正如下：

```
SQL> select substr(tablespace_name,1,15) a,  
2         substr(file_name,1,45) c, bytes  
3   from sys.dba_data_files  
4  order by tablespace_name;
```

Group Function Is Not Allowed Here

输入：

```
SQL> select count(last_name), first_name, phone_number  
2   from employee_tbl  
3  group by count(last_name), first_name, phone_number  
4   /
```

输出：

```
group by count(last_name), first_name, phone_number  
*  
ERROR at line 3:  
ORA-00934: group function is not allowed here  
SQL>
```

分析：

与任何组函数一样，COUNT 不可能在 GROUP BY 子句中使用，你可能在 GROUP BY 子句中使用了列或不具备分组功能的函数，如 SUBSTR。

技巧：COUNT 函数是在分组查询的执行段中使用的。

上边的语句已经使用正确的语法更正了：

```
SQL> select count(last_name), first_name, phone_number  
2   from employee_tbl  
3  group by last_name, first_name, phone_number;
```

Invalid Column Name

输入:

```
SQL> @tables.sql
```

输出:

```
SQL> spool tables.lst
```

```
SQL> set echo on
```

```
SQL> set feedback on
```

```
SQL> set pagesize 1000
```

```
SQL> select owner|| '.' || tablename
```

```
2  from sys.dba_tables
```

```
3  where owner = 'SYSTEM'
```

```
4  order by table_name
```

```
5  /
```

```
select owner|| '.' || tablename
```

```
*
```

```
ERROR at line 1:
```

```
ORA-00904: invalid column name
```

```
SQL> spool off
```

```
SQL>
```

分析:

在第 1 行中的 tablename 列名是不正确的，正确的列名是 table_name。下划线少了。如果想看正确的列，应该使用 DESCRIBE 命令。这个错误也可能会由于在 SELECT 语句中用错误的表名对列进行限制时产生。

Missing Keyword

输入:

```
SQL> create view emp_view
```

```
2  select * from employee_tbl
```

```
3 /
```

输出:

```
select * from employee_tbl
*
```

ERROR at line 2:

ORA-00905: missing keyword

SQL>

分析:

这里的语法是不正确的，当你使用命令时语法与所要求的不一致会出现这种错误。如果你使用了命令的可选项，那么可以是这个选项需要确定的关键字。在这个例子中是缺少了关键字 AS。正确的语句如下：

```
SQL> create view emp_view as
2 select * from employee_tbl
3 /
```

Missing Left Parenthesis

输入:

```
SQL> @insert.sql
```

输出:

```
SQL> insert into people_tbl values
2 '303785523', 'SMITH', 'JOHN', 'JAY', 'MALE', '10-JAN-50')
3 /
'303785523', 'SMITH', 'JOHN', 'JAY', 'MALE', '10-JAN-50')
*
```

ERROR at line 2:

ORA-00906: missing left parenthesis

SQL>

分析:

在第 2 行中圆括号没有出现，正确的语法应该如下：

```
SQL> insert into people_tbl values
2 ('303785523', 'SMITH', 'JOHN', 'JAY', 'MALE', '10-JAN-50')
3 /
```

Missing Right Parenthesis

输入:

```
SQL> @tblspc.sql
```

输出:

```
SQL> spool tblspc.lst
SQL> set echo on
SQL> set feedback on
SQL> set pagesize 1000
SQL> select substr(tablespace_name,1,15) a,
2         substr(file_name, 1,45) c, bytes
3   from sys.dba_data_files
4  order by tablespace_name;
        select substr(tablespace_name,1,15) a,
                *
```

ERROR at line 1:

ORA-00907: missing right parenthesis

```
SQL> spool off
```

```
SQL>
```

分析:

在第 1 行的 SUBSTR 中缺少右括号，正确的语法如下:

```
SQL> select substr(tablespace_name,1,15) a,
2         substr(file_name,1,45) c, bytes
3   from sys.dba_data_files
4  order by tablespace_name;
```

Missing Comma

输入:

```
SQL> @ezinsert.sql
```

输出:

```
SQL> spool ezinsert.lst
```

```
SQL> set echo on
```

```
SQL> set feedback on
```

```
SQL> insert into office_tbl values
```

```
2 ('303785523' 'SMITH', 'OFFICE OF THE STATE OF INDIANA, ADJUTANT GENERAL')
```

```
3 /
```

```
('303785523' 'SMITH', 'OFFICE OF THE STATE OF INDIANA, ADJUTANT GENERAL')
```

```
*
```

```
ERROR at line 2:
```

```
ORA-00917: missing comma
```

```
SQL> spool off
```

```
SQL>
```

分析:

在第 2 行的 SMITH 与安全数字之间缺了逗号。

Column Ambiguously Defined

输入:

```
SQL> @employee_tbl
```

输出:

```
SQL> spool employee.lst
```

```
SQL> set echo on
```

```
SQL> set feedback on
```

```
SQL> select p.ssn, name, e.address, e.phone
```

```
2 from employee_tbl e,
```



```
3 payroll_tbl p
4 where e.ssn =p.ssn;

select p.ssn, name, e.address, e.phone
      *
```

ERROR at line 1:

ORA-00918: column ambiguously defined

SQL> spool off

SQL>

分析:

在第一行中的 name 列没有定义。表已经被给了别名 e 和 p。根据它的别名可以决定使用那一个表。

SQL Command Not Properly Ended

输入:

```
SQL> create view emp_tbl as
2 select * from employee_tbl
3 order by name
4 /
```

输出:

```
order by name
```

```
*
```

ERROR at line 3:

ORA-00933: SQL command not properly ended

SQL>

分析:

为什么命令没有正确地结束呢？你知道你可以使用 / 来结束 SQL 语句，可以在 CREATE VIEW 语句中是不能使用 ORDER BY 子句的，要用 GROUP BY 来代替。查询处理器要在 ORDER BY 子句之前寻找结束标志（分号或正斜线）。因为处理器认为 ORDER BY 不是 CREATE VIEW 的一部分。而由于在 ORDER BY 之前没有结束标志，所以错误返回并指明是在 ORDER BY 子句所在行。Missing Expression

输入：

```
SQL> @tables.sql
```

输出：

```
SQL> spool tables.lst
```

```
SQL> set echo on
```

```
SQL> set feedback on
```

```
SQL> set pagesize 1000
```

```
SQL> select owner|| '.' || table,
2  from sys.dba_tables
3  where owner = 'SYSTEM'
4  order by table_name
5  /

      from sys.dba_tables
      *
```

```
ERROR at line 2:
```

```
ORA-00936: missing expression
```

```
SQL> spool off
```

```
SQL>
```

分析：

注意在第一行的表后边的逗号，因为它，处理机制会认为在 SELECT 语句中还有其它列，这时它不会去找 FROM 子句。

Not Enough Arguments for Function

输入：

```
SQL> @tblspc.sql
```

输出：

```
SQL> spool tblspc.lst
```

```
SQL> set echo on
```

```
SQL> set feedback on
```

```
SQL> set pagesize 1000

SQL> select substr(tablespace_name,1,15) a,
2          decode(substr(file_name,1,45)) c, bytes
3  from sys.dba_data_files
4  order by tablespace_name;

          decode(substr(file_name,1,45)) c, bytes
          *
```

ERROR at line 2:

ORA-00938: not enough arguments for function

SQL> spool off

SQL>

分析:

在 DECODE 函数中没有足够的参数，你需要检查你的解释器看一下它的正确语法。

Not Enough Values

输入:

```
SQL> @ezinsert.sql
```

输出:

```
SQL> spool ezinsert.lst

SQL> set echo on

SQL> set feedback on

SQL> insert into employee_tbl values
2  ('303785523', 'SMITH', 'JOHN', 'JAY', 'MALE')
3  /

insert into employee_tbl values
          *
```

ERROR at line 1:

ORA-00947: not enough values

SQL> spool off

SQL>

分析：

列值不存在，你需要执行 DESCRIBE 命令来找到丢失的列，你只可以向你所指定的列中插入数值，如下例：

输入：：

```
SQL> spool ezinsert.lst
```

```
SQL> set echo on
```

```
SQL> set feedback on
```

```
SQL> insert into employee_tbl (ssn, last_name, first_name, mid_name, sex)
```

```
2 values ('303785523', 'SMITH', 'JOHN', 'JAY', 'MALE')
```

```
3 /
```

Integrity Constraint Violated--Parent Key Not Found

输入：

```
SQL> insert into payroll_tbl values
```

```
2 ('111111111', 'SMITH', 'JOHN')
```

```
3 /
```

输出：

```
insert into payroll_tbl values
```

```
*
```

```
ERROR at line 1:
```

```
ORA-02291: integrity constraint (employee_cons) violated - parent
```

```
key not found
```

```
SQL>
```

分析：

这个错误的原因是因为试图向一个表中插入数据，而对应的数据在父表中不存在时出现，你需要查询父表中是否有正确的数据，如果没有，你必须在向子表中插入数据之前先向它的父表中插入数据。

Oracle Not Available

输入:

```
(sun_su3)/home> sqlplus
```

```
SQL*Plus: Release 3.2.3.0.0 - Production on Sat May 10 11:19:50 1997
```

```
Copyright (c) Oracle Corporation 1979, 1994. All rights reserved.
```

```
Enter user-name: rplew
```

```
Enter password:
```

输出:

```
ERROR: ORA-01034: ORACLE not available
```

```
ORA-07318: smsget: open error when opening sgadef.dbf file.
```

分析:

你没有得到 SQL*PLUS 的提示，数据库可能当掉了。检查数据库的状态。同样，如果你在访问多个数据库的话你要确认你的连接是正确的。

Inserted Value Too Large for Column

输入:

```
SQL> @ezinsert.sql
```

输出:

```
SQL> spool ezinsert.lst
```

```
SQL> set echo on
```

```
SQL> set feedback on
```

```
SQL> insert into office_tbl values
```

```
2 ('303785523', 'SMITH', 'OFFICE OF THE STATE OF INDIANA, ADJUTANT  
GENERAL')
```

```
3 /
```

```
insert into office_tbl values
```

```
*
```

```
ERROR at line 1:
```

```
ORA-01401: inserted value too large for column
```

```
SQL> spool off
```

```
SQL>
```

分析：

要插入的数值对于列来说太大。在表中使用 DESCRIBE 命令来更正数据的长度。如有必要，你可以执行 ALTER TABLE 命令来长宽表的列宽。

TNS:listener Could Not Resolve SID Given in Connect Descriptor

输入：

```
SQLDBA> connect rplew/xxxx@database1
```

输出：

```
ORA-12505: TNS:listener could not resolve SID given in connect descriptor
```

```
SQLDBA> disconnect
```

```
Disconnected.
```

```
SQLDBA>
```

分析：

在 ORACLE 数据库中这个错误很常见， listener 在请求客户与远程服务通讯时产生了错误。这里是你试图连接数据库，不论是数据库名字错误还是 LISTENER 当掉都是有可能的。你检查一个数据库的名字，然后再输入一次，如有必要，将这个问题告知数据库管理员。

Insufficient Privileges During Grants

输入：

```
SQL> grant select on people_tbl to ron;
```

输出：

```
grant select on people_tbl to ron
```

*

ERROR at line 1:

ORA-01749: you may not GRANT/REVOKE privileges to/from yourself

SQL>

输入:

SQL> grant select on demo.employee to ron;

输出:

grant select on demo.employee to ron

*

ERROR at line 1:

ORA-01031: insufficient privileges

SQL>

这个错误出现在当你试图授予其它的用户表权限而你没有这么做的权利时出现。你必需有表授权权限才可以。在 Oracle 中你的权限选项是管理员选项，这就是说你有将指定的权限授予其它用户的能力。请检查你的解释器以找到特定的授权选项。

Escape Character in Your Statement--Invalid Character

当你在调试 SQL 语句时不能使用脱字符，这种情况在你输入了 SQL 语句到缓冲区或文件中以后使用退格键时产生。有时退格键会在你的语句中输入不合法的字符，这要根据你的具体的键盘情况而定。尽管你看不到这些字符。

Cannot Create Operating System File

这个错误的产生有好几个原因，最常见的原因是磁盘满了或文件系统的许可权不正确。如果磁盘满了。你必须删除一些不需要的文件，如果是权限不正确。请正确地设置它们。这个错误更可能是操作系统的错误，所以你需要征询系统管理员的建议。

Common Logical Mistakes

到现在我们已经讨论了全部的在 SQL 语句中的实际上的错误信息。它们中的大多数很明显。也很容易断定。下边的错误更可能是逻辑错误，它们在稍后产生问题——不是马上。

Using Reserved Words in Your SQL statement

输入:

```
SQL> select sysdate DATE from dual;
```

输出:

```
select sysdate DATE
```

```
          *
```

```
ERROR at line 1:
```

```
ORA-00923: FROM keyword not found where expected
```

分析:

在这个例子中查询处理器并不会处理 DATE 字, 因为它是一个保留字。由于在 SYSDATE 后边没有逗号。所以处理认为下一个子句应该是 FROM。

输入:

```
SQL> select sysdate "DATE"
```

```
2 from dual;
```

输出:

```
DATE
```

```
15-MAY-97
```

分析:

注意在这里是如何解决保留字问题的——用双引号将它括起来。它可以让你将 DATE 作为别名字符串处理。

注: 你应该检查你的数据库文档中的保留字列表, 因为不同的解释器中的保留字是不同的。

当对一个列使用别名时你可能使用双引号也可能不使用, 在下边的例子中你就不必使用双引号, 因为 TODAY 不是一个保留字。所以应该检查你的解释器再进行判断。

输入:

```
SQL> select sysdate TODAY
```

```
2 from dual;
```

输出:

```
TODAY
```

```
15-MAY-97
```


SQL>

The Use of DISTINCT When Selecting Multiple Columns

输入:

```
SQL> select distinct(city), distinct(zip)
      2  from address_tbl;
```

输出:

```
select distinct(city), distinct(zip)
*

```

ERROR at line 1:

ORA-00936: missing expression

SQL>

分析:

CITY 可能比州代码更多，作为一个规则，你应该使用 DISTINCT 命令来选择唯一值。

Dropping an Unqualified Table

无论何时你删除表时都要使用工程或所有人的名字。以避免在数据库中表的名字会重复，如果你使用了所有者或工程的名字。那么错误就不会发生。

删除一个表的语法是非常危险的。

语法:

```
SQL> drop table people_tbl;
```

下边的句子会更安全，因为它使用了所有人来限定所要删除的表。

语法:

```
SQL> drop table ron.people_tbl;
```

警告：在删除表时进行资格限定是一个好习惯，尽管有时这一步工作可能不是必需的。再
你没有进行你所连接的数据进行用户验证时千万不要删除表。

The Use of Public Synonyms in a Multischema Database

同义字对于用户来说会更容易。但是，公共的同义字可能会把你不想让所有的人都看到的表打开。所以在使用公共同义字，尤其是在多工程情况下使用公共同义字要非常小心。

The Dreaded Cartesian Product

输入：

```
SQL> select a.ssn, p.last_n  
2   from address_tbl a,  
3       people_tbl p;
```

输出：

SSN	LAST_NAME
303785523	SMITH
313507927	SMITH
490552223	SMITH
312667771	SMITH
420001690	SMITH
303785523	JONES
313507927	JONES
490552223	JONES
312667771	JONES
420001690	JONES
303785523	OSBORN
313507927	OSBORN
490552223	OSBORN
312667771	OSBORN
420001690	OSBORN
303785523	JONES
313507927	JONES
490552223	JONES
312667771	JONES
420001690	JONES

这个错误会在你没有在 WHERE 子句连接表的时候产生，注意选择了多少行，在前边

的表是 4 行，所以你可能想要返回的是 4 行而不是 16 行，由于没有在 WHERE 中使用联接，所以第一个表的中每一行都会分别与第二个表中的每一行相匹配。结果就是 4 乘 4 行了。不幸的是你的许多表可能不只是 4 行，而是成千上万行，在这种情况下，在这种情况下不会返回数以亿计的结果，而是你的查询看起来像是停止了反应。Failure to 坚持标准的输入

坚持使用标准的输入方法是一个公认的好习惯。没有输入员对数据的经常检查，你将会使数据库在非常危险的情况下运行，一个好的办法就是对使用 SQL 创建的 QA 报表的质量进行控制。让它们在一定的时间内运行，在数据提交之前尽可能地更正错误。

Failure to Enforce File System Structure Conventions

在非标准的文件系统下工作会浪费你大量的时间，你应该检查你的解释器，看一看它向你推荐的文件系统。

Allowing Large Tables to Take Default Storage Parameters

存储参数的定义随解释器的不同而不同，但是它们通常是比较小的，当在默认的存储空间中生成大的动态表时，常会导致碎片的产生。这使数据库的性能受到了很大的影响。在表设计时进行较好的计划可以避免这一问题。下边的例子使用的是 ORACLE 的存储参数。

INPUT:

```
SQL> create table test_tbl
2  (ssn    number(9) not null,
3  name    varchar2(30) not null)
4  storage
5  (initial extent 100M
6   next extent    20M
7   minextents 1
8   maxextents 121
9   pctincrease 0);
```

Placing Objects in the System Tablespace

下边的例子给出了一个在系统表空间中创建的表，尽管语句并没有返回任何错误，但是它将会在以后导致问题。

输入：

```
SQL> create table test_tbl
2  (ssn    number(9) not null,
3  name    varchar2(30) not null)
4  tablespace SYSTEM
5  storage
6  (initial extent 100M
7  next extent      20M
8  minextents 1
9  maxextents 121
10 pctincrease 0);
```

下边的例子更正了这个所谓的错误：

输入：

```
SQL> create table test_tbl
2  (ssn    number(9) not null,
3  name    varchar2(30) not null)
4  tablespace linda_ts
5  (initial extent 100M
6  next extent      20M
7  minextents 1
8  maxextents 121
9  pctincrease 0);
```

分析：

在 Oracle 中，系统表空间是由系统对象如数据字典所使用的。如果你在这个空间中放置了动态表，由于它们会增大，你的运行将是危险的，至少它要占用自由空间，这可能会导致你的数据库崩溃，这时数据库可能是无法恢复的，所以你应该将应用程序和用户表

保存在指定的表空间中。

Failure to Compress Large Backup Files

如果你导出文件很大且没有压缩的话，你需要存储的文件可能会超过磁盘空间的大小，一定要压缩导出文件，如果你是在磁盘上存有历史记录文件而不是在磁带上，这些文件可以被压缩以节省磁盘空间

Failure to Budget System Resources

你应该在创建你的数据库时对你的系统进行预先估测，没有进行估测的结果将会使你的数据库性能极差，你应该知道数据库是使用事务，数据仓库还是只有查询，数据库的功能对 ROLLBACK 的大小有直接的影响，而数据库的用户数则对 USERS 和 TEMP 表空间的大小有影响，你是否有足够的空间来存贮你的大型表呢？表和索引应该被存储在不同的磁盘上以减少磁盘访问冲突，你也应该将数据表空间和历史记录文件分布于不同的设备以减少磁盘访问量，这都是在估测系统资源时要考虑的内容。

Preventing Problems with Your Data

你的数据处理中心应该有系统备份功能，如果你的数据库是中小型的，你可以使用 EXPORT 命令来将数据预先导出并确保数据备份，你应该将导出的备份文件存储于另一个安全的地点。由于这些文件很大，所以它可能要占用较多的磁盘空间。

Searching for Duplicate Records in Your Database

如果你的数据库进行了很好的预先规划，那就不会出现冗余记录问题，你应该通过使用约束、外部关键字和唯一值索引来避免冗余记录。

总结：

错误的类型很多——可能有几百个——会出现你的工作道路上和你的数据中。幸运的是大多数错误和过失不会造成灾难而且容易处理。但是，其中一些可能是非常严重的，

所以你认真对待它。如果你不深究它可能会复杂化。如果你犯了错误，你应该明确的采用你所学习的经验。

技巧：我们更喜欢从数据库的文档中找错误信息，尤其是对于偶然发现的一些不常见的错误，错误文件和故障排查的价值是不可估量的。

注：今天我们给出了在 PERSONAL ORACLE 7 中的一些常见的错误及解决方案，对于更详细的内容你要参阅你的数据库相关文档。

问与答：

问：如果我每个错误都可以处理，那我还担心什么呢？

答：对，大多数错误很容易处理，但是如果你删除了一个在使用环境中的表呢？你也许需要几个小时或几天才能恢复它。这段时间数据库不能工作，为了修复它你耽搁了许多人的时间，你的老板会高兴吗？

问：有什么建议可以避免错误？

答：只要是人，你就不可能避免所有的错误，但是，你通过练习并汲取经验你可以避免它们中的大多数。从而使你的工作更轻松。

校练场

1. 一个用户打电话说：“我不能登录数据库了，昨天还能呢。你能帮帮我吗？你该如何做？
2. 为什么表在存储子句中有表空间项。

练习

1. 如果你以 SYSTEM 身份登录了数据库，你想删除你的计划中的一个名字叫 HISTORY 的表，如果你的用户 ID 是 JSMITH，那么正确的语法是怎样的？
2. 更正下边语句的错误

INPUT:

```
SQL> select sysdate DATE
```

```
2 from dual;
```

OUTPUT:

```
select sysdate DATE
```

```
*
```

ERROR at line 1:

ORA-00923: FROM keyword not found where expected

第三周回顾

这是多产的一周。在这一周中你看到了 SQL 的灵活性，并向你讲解了如何在现实世界中应用这些特点。并对一些流行的 SQL 进行了介绍，你应该知道了如何使用解释器中提供了工具来让你的代码更加易读。在现在为止，你已经知道了所有的 SQL 解释器的一些共有概念，尽管在语法上它们可能会不同。

你也对数据字典有了一个清楚的认识，知道了在它之中存储了些什么，以及如何从它们中获得有用的数据。如果你知道了如何从 SQL 语句中生成 SQL 语句，那么你的应用能力将会飞跃到一个新的高度。

什么是错误？你不可能避免语法和逻辑上的错误，但是通过在 SQL 中的实际试验，你将学习到如何避免大多数的错误。而且错误也是一个非常好的学习机会。

附件 A：在 SQL 中的常见术语

ALTER DATABASE

ALTER DATABASE database_name;

ALTER DATABASE 命令可以改变数据库的设计尺寸，它的语法根据数据库系统的不同差别很大

ALTER USER

ALTER USER user

ALTER USER 可以改变像密码之类的用户的系统设置。

BEGIN TRANSACTION

1> BEGIN TRANSACTION transaction_name

2> transaction type

3> if exists

4> begin

BEGIN TRANSACTION 语句表明要开始一个用户事务，事务在遇到 COMMITTED 命令（参见[COMMIT TRANSACTION](#)）或 CANCELED（参见 ROLLBACK TRANSACTION）时终止，一个事务是一个逻辑上的工作单位。ion is a logical unit of work.

CLOSE CURSOR

close cursor_name

CLOSE cursor_name 语法将关闭游标并清除掉它们中的数据，要想彻底地删除游标，需要使用 DEALLOCATE CURSOR 语句。

COMMIT TRANSACTION

SQL> COMMIT;

COMMIT TRANSACTION 语句将保存所有的从一个事务开始以后（也就是自从[BEGIN TRANSACTION](#)语句运行以后）所做的工作

CREATE DATABASE

SQL> CREATE DATABASE database_name;

database_name 是要创建的数据库的名字，在创建数据库时有许多不同的如设备等选项可以应用。并可以对数据库的大小进行初始化。

CREATE INDEX

CREATE INDEX index_name ON table_name(column_name1, [column_name2], ...);

创建索引字段的内容。

CREATE PROCEDURE

create procedure procedure_name

[(*@parameter_name*

datatype [(length) | (precision [, scale])

[= default][output]

[, *@parameter_name*

datatype [(length) | (precision [, scale])

[= default][output]]...[*output*])]

[with recompile]

as SQL_statements

CREATE PROCEDURE 语句可以在数据库中创建一个新的存储过程，这个存储过程可以由 SQL 语句组成并通过使用 EXECUTE 命令来运行。存储过程支持输入和输出参数并可以返回一个整数值用以进行状态检测。

CREATE TABLE

```
CREATE TABLE table_name  
( field1 datatype [ NOT NULL ],  
  field2 datatype [ NOT NULL ],  
  field3 datatype [ NOT NULL ]...)
```

CREATE TABLE 可以在数据库中创建一个新的表，每一个可选的字段都为数据库提供了一个确定的字段名和数据类型。

CREATE TRIGGER

```
create trigger trigger_name  
  on table_name  
  for {insert, update, delete}  
  as SQL_Statements
```

CREATE TRIGGER 语句可以创建一个触发机制，它可以在数据库进行插入、更新和删除操作时自动执行。它也可以调用存储过程以运行一些复杂的任务。

CREATE USER

```
CREATE USER user
```

CREATE USER 语句创建一个包括用户名和密码的新用户帐号。

CREATE VIEW

```
CREATE VIEW <view_name> [(column1, column2...)] AS  
SELECT <table_name column_names>  
FROM <table_name>
```

使用 CREATE VIEW 语句创建视图以后，你就可以使用它来查询数据并对视图内的数据进行更改。

DEALLOCATE CURSOR

```
deallocate cursor cursor_name
```

DEALLOCATE CURSOR 语句可以彻底地从内存中将游标删除并释放游标的名字使它可以为其它的游标使用，在释放它之前你应该先使用[CLOSE CURSOR](#)命令把游标关闭。

DECLARE CURSOR

```
declare cursor_name cursor
```

```
for select_statement
```

DECLARE CURSOR 语句可以从 SELECT 语句中创建一个新的游标，[FETCH](#)语句可以翻阅游标中的数据直到变量载入，然后游标跳到下一个记录上。

DROP DATABASE

```
DROP DATABASE database_name;
```

DROP DATABASE 语句可以彻底地删除数据库，包括数据库中的数据 and 它在磁盘上的物理结构。

DROP INDEX

```
DROP INDEX index_name;
```

DROP INDEX 可以将表的索引删除。

DROP PROCEDURE

```
drop procedure procedure_name
```

DROP PROCEDURE 语句可以从数据库中删除一个存储过程；它的功能与[DROP TABLE](#)和[DROP INDEX](#)语句相似。

DROP TABLE

```
DROP TABLE table_name;
```

DROP TABLE 语句可以从数据库中删除表。

DROP TRIGGER

DROP TRIGGER trigger_name

DROP TRIGGER 可以从数据库中删除触发机制。

DROP VIEW

DROP VIEW view_name;

DROP VIEW 语句可以从数据库中删除视图。

EXECUTE

execute [@return_status =]

procedure_name

[[@parameter_name =] value |

[@parameter_name =] @variable [output] ...]

EXECUTE 命令可以运行一个包含有 SQL 语句的存储过程。在存储过程中可以输入参数。如果使用了 output 关键字的话数据还可以从参数中返回。

FETCH

fetch cursor_name [into fetch_target_list]

FETCH 命令可以将游标的内容装填到提供的程序变量中。在变量载入以后，游标就会跳跃到下一条记录。

FROM

FROM <tableref> [, <tableref> ...]

FROM 指定了联接的是哪一个表。

GRANT

GRANT role TO user 或 GRANT system_privilege TO {user_name | role | PUBLIC}

GRANT 命令可以给由命令[CREATE USER](#)所创建的用户授予规则权限。

GROUP BY

GROUP BY <col> [, <col> ...]

GROUP BY 语句可以将所以列名相同的行组织在一起。

HAVING

HAVING <search_cond>

HAVING 只有在 [GROUP BY](#)下有效，它用以限制选择的组要满足指定的搜索条件。

INTERSECT

INTERSECT

INTERSECT 返回两个[SELECT](#)语句中的所有公共元素。

ORDER BY

ORDER BY <order_list>

ORDER BY 语句可以通过指定列句来对内容进行排序。

ROLLBACK TRANSACTION

ROLLBACK TRANSACTION 语句的作用是使一个事务（从[BEGIN TRANSACTION](#)语句运行时起）中的所有工作全部取消。

REVOKE

REVOKE role FROM user;

或 REVOKE {object_priv | ALL [PRIVILEGES]}

[, {object_priv | ALL [PRIVILEGES]}] ...

ON [schema.]object

FROM {user | role | PUBLIC} [, {user | role | PUBLIC}] ...

REVOKE 命令将删除一个用户的所有数据库权限——无论是系统权限还是规则。

SELECT

SELECT [DISTINCT | ALL]

SELECT 语句是每一个获得数据的语句的开始，修正字 DISTINCT 可以指定让它返回一个重复的数值，ALL 是默认的，返回全部数据。

SET TRANSACTION

SQL> SET TRANSACTION (READ ONLY | USE ROLLBACK SEGMENT);

SET TRANSACTION 可以让用户指定什么时候事务应该开始，READ ONLY 选项会锁定一组记录集直到事务结束以确保在这一过程中数据没有被改变过。

UNION

UNION

UNION 语句会返回两个[SELECT](#)语句中的所有元素。

WHERE

WHERE <search_cond>

WHERE 语句限制返回的行必须满足指定的条件。

*

* 可以代替一个表中的所有列。

附件 B：在第 14 天中的 C++ 源代码清单

```
// tyssqvw.h : interface of the CTyssqlView class
//
///////////////////////////////////////////////////////////////////

class CTyssqlSet;

class CTyssqlView : public CRecordView
{
protected: // create from serialization only
    CTyssqlView();
    DECLARE_DYNCREATE(CTyssqlView)
public:
   //{{AFX_DATA(CTyssqlView)
    enum { IDD = IDD_TYSSQL_FORM };
    CTyssqlSet* m_pSet;
    //}}AFX_DATA
// Attributes
public:
    CTyssqlDoc* GetDocument();
// Operations
public:
    virtual CRecordset* OnGetRecordset();
// Implementation
public:
    virtual ~CTyssqlView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
#endif
```



```

protected:

    virtual void DoDataExchange(CDataExchange* pDX);// DDX/DDV support

    virtual void OnInitialUpdate(); // called first time after construct

// Generated message map functions
protected:

   //{{AFX_MSG(CTyssqlView)

        // NOTE - the ClassWizard will add and remove member functions here.

        // DO NOT EDIT what you see in these blocks of generated code !

   //}}AFX_MSG

    DECLARE_MESSAGE_MAP()

};

#ifdef _DEBUG // debug version in tyssqvw.cpp

inline CTyssqlDoc* CTyssqlView::GetDocument()

    { return (CTyssqlDoc*)m_pDocument; }

#endif

////////////////////////////////////

// tyssql.h : main header file for the TYSSQL application

//

#ifdef __AFXWIN_H__

    #error include 'stdafx.h' before including this file for PCH

#endif

#include "resource.h"        // main symbols

////////////////////////////////////

// CTyssqlApp:

// See tyssql.cpp for the implementation of this class

//

class CTyssqlApp : public CWinApp

{

public:

    CTyssqlApp();

```

```

// Overrides

    virtual BOOL InitInstance();

// Implementation

   //{{AFX_MSG(CTyssqlApp)

afx_msg void OnAppAbout();

    // NOTE - the ClassWizard will add and remove member functions here.

    // DO NOT EDIT what you see in these blocks of generated code !

    }}AFX_MSG

    DECLARE_MESSAGE_MAP()

};

/////////////////////////////////////////////////////////////////

// tyssqlset.h : interface of the CTyssqlSet class

//

/////////////////////////////////////////////////////////////////

class CTyssqlSet : public CRecordset
{
    DECLARE_DYNAMIC(CTyssqlSet)

public:

    CTyssqlSet(CDatabase* pDatabase = NULL);

// Field/Param Data

    {{{AFX_FIELD(CTyssqlSet, CRecordset)

        CString    m_NAME;

        CString    m_ADDRESS;

        CString    m_STATE;

        CString    m_ZIP;

        CString    m_PHONE;

        CString    m_REMARKS;

    }}}AFX_FIELD

// Implementation

protected:

```

```

    virtual CString GetDefaultConnect();    // Default connection string

    virtual CString GetDefaultSQL();        // default SQL for Recordset

    virtual void DoFieldExchange(CFieldExchange* pFX);    // RFX support
};

// tyssqdoc.h : interface of the CTyssqlDoc class
//
//
/////////////////////////////////////////////////////////////////

class CTyssqlDoc : public CDocument
{
protected: // create from serialization only

    CTyssqlDoc();

    DECLARE_DYNCREATE(CTyssqlDoc)

// Attributes

public:

    CTyssqlSet m_tyssqlSet;

// Operations

public:

// Implementation

public:

    virtual ~CTyssqlDoc();

#ifdef _DEBUG

    virtual void AssertValid() const;

    virtual void Dump(CDumpContext& dc) const;

#endif

protected:

    virtual BOOL OnNewDocument();

// Generated message map functions

protected:

   //{{AFX_MSG(CTyssqlDoc)

    // NOTE - the ClassWizard will add and remove member functions here.

```

```

// DO NOT EDIT what you see in these blocks of generated code !

//}}AFX_MSG

DECLARE_MESSAGE_MAP()

};

////////////////////////////////////

// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//

#include <afxwin.h>          // MFC core and standard components
#include <afxext.h>          // MFC extensions (including VB)
#include <afxdb.h>           // MFC database classes

////////////////////////////////////

//{{NO_DEPENDENCIES}}
// App Studio generated include file.
// Used by TYSSQL.RC
//

#define IDR_MAINFRAME                2
#define IDD_ABOUTBOX                 100
#define IDD_TYSSQL_FORM              101
#define IDP_FAILED_OPEN_DATABASE     103
#define IDC_NAME                     1000
#define IDC_ADDRESS                   1001
#define IDC_STATE                     1002
#define IDC_ZIP                       1003

// Next default values for new objects
//

#ifdef APSTUDIO_INVOKED
#ifdef APSTUDIO_READONLY_SYMBOLS

#define _APS_NEXT_RESOURCE_VALUE        102

```

```

#define _APS_NEXT_COMMAND_VALUE      32771

#define _APS_NEXT_CONTROL_VALUE      1004

#define _APS_NEXT_SYMED_VALUE        101

#endif

#endif

////////////////////////////////////

// mainfrm.h : interface of the CMainFrame class

//

////////////////////////////////////

class CMainFrame : public CFrameWnd
{
protected: // create from serialization only

    CMainFrame();

    DECLARE_DYNCREATE(CMainFrame)

// Attributes

public:

// Operations

public:

// Implementation

public:

    virtual ~CMainFrame();

#ifdef _DEBUG

    virtual void AssertValid() const;

    virtual void Dump(CDumpContext& dc) const;

#endif

protected: // control bar embedded members

    CStatusBar  m_wndStatusBar;

    CToolBar    m_wndToolBar;

// Generated message map functions

protected:

```

```

//{{AFX_MSG(CMainFrame)

afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);

    // NOTE - the ClassWizard will add and remove member functions here.

    //    DO NOT EDIT what you see in these blocks of generated code!

//}}AFX_MSG

DECLARE_MESSAGE_MAP()

};

////////////////////////////////////

// tyssqvw.cpp : implementation of the CTyssqlView class
//

#include "stdafx.h"

#include "tyssql.h"

#include "tyssqset.h"

#include "tyssqdoc.h"

#include "tyssqvw.h"

#ifdef _DEBUG

#undef THIS_FILE

static char BASED_CODE THIS_FILE[] = __FILE__;

#endif

////////////////////////////////////

// CTyssqlView

IMPLEMENT_DYNCREATE(CTyssqlView, CRecordView)

BEGIN_MESSAGE_MAP(CTyssqlView, CRecordView)

   //{{AFX_MSG_MAP(CTyssqlView)

        // NOTE - the ClassWizard will add and remove mapping macros here.

        //    DO NOT EDIT what you see in these blocks of generated code!

   //}}AFX_MSG_MAP

END_MESSAGE_MAP()

////////////////////////////////////

// CTyssqlView construction/destruction

```

```

CTyssqlView::CTyssqlView()
    : CRecordView(CTyssqlView::IDD)
{
   //{{AFX_DATA_INIT(CTyssqlView)
    m_pSet = NULL;
   //}}AFX_DATA_INIT
    // TODO: add construction code here
}

CTyssqlView::~CTyssqlView()
{
}

void CTyssqlView::DoDataExchange(CDataExchange* pDX)
{
    CRecordView::DoDataExchange(pDX);
   //{{AFX_DATA_MAP(CTyssqlView)
    DDX_FieldText(pDX, IDC_ADDRESS, m_pSet->m_ADDRESS, m_pSet);
    DDX_FieldText(pDX, IDC_NAME, m_pSet->m_NAME, m_pSet);
    DDX_FieldText(pDX, IDC_STATE, m_pSet->m_STATE, m_pSet);
    DDX_FieldText(pDX, IDC_ZIP, m_pSet->m_ZIP, m_pSet);
   //}}AFX_DATA_MAP
}

void CTyssqlView::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_tyssqlSet;
    CRecordView::OnInitialUpdate();
}

////////////////////////////////////

// CTyssqlView diagnostics
#ifdef _DEBUG

void CTyssqlView::AssertValid() const

```

```

{
    CRecordView::AssertValid();
}

void CTyssqlView::Dump(CDumpContext& dc) const
{
    CRecordView::Dump(dc);
}

CTyssqlDoc* CTyssqlView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CTyssqlDoc)));

    return (CTyssqlDoc*)m_pDocument;
}

#endif // _DEBUG

////////////////////////////////////

// CTyssqlView database support

CRecordset* CTyssqlView::OnGetRecordset()
{
    return m_pSet;
}

////////////////////////////////////

// CTyssqlView message handlers

// tyssqlset.cpp : implementation of the CTyssqlSet class
//

#include "stdafx.h"

#include "tyssql.h"

#include "tyssqlset.h"

////////////////////////////////////

// CTyssqlSet implementation

IMPLEMENT_DYNAMIC(CTyssqlSet, CRecordset)

CTyssqlSet::CTyssqlSet(CDatabase* pdb)

```



```
        : CRecordset(pdb)
    {
       //{{AFX_FIELD_INIT(CTyssqlSet)
        m_NAME = "";
        m_ADDRESS = "";
        m_STATE = "";
        m_ZIP = "";
        m_PHONE = "";
        m_REMARKS = "";
        m_nFields = 6;

       //}}AFX_FIELD_INIT
    }

CString CTyssqlSet::GetDefaultConnect()
{
    return "ODBC;DSN=TYSSQL;";
}

CString CTyssqlSet::GetDefaultSQL()
{
    return "SELECT * FROM CUSTOMER ORDER BY NAME";
}

void CTyssqlSet::DoFieldExchange(CFieldExchange* pFX)
{
   //{{AFX_FIELD_MAP(CTyssqlSet)
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Text(pFX, "NAME", m_NAME);
    RFX_Text(pFX, "ADDRESS", m_ADDRESS);
    RFX_Text(pFX, "STATE", m_STATE);
    RFX_Text(pFX, "ZIP", m_ZIP);
    RFX_Text(pFX, "PHONE", m_PHONE);
    RFX_Text(pFX, "REMARKS", m_REMARKS);
    }}
```

```

        //}}AFX_FIELD_MAP

    }

// tyssql.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"

#include "tyssql.h"

#include "mainfrm.h"

#include "tyssqset.h"

#include "tyssqdoc.h"

#include "tyssqvw.h"

#ifdef _DEBUG

#undef THIS_FILE

static char BASED_CODE THIS_FILE[] = __FILE__;

#endif

////////////////////////////////////

// CTyssqlApp

BEGIN_MESSAGE_MAP(CTyssqlApp, CWinApp)

   //{{AFX_MSG_MAP(CTyssqlApp)

    ON_COMMAND(ID_APP_ABOUT, OnAppAbout)

    // NOTE - the ClassWizard will add and remove mapping macros here.

    // DO NOT EDIT what you see in these blocks of generated code!

    //}}AFX_MSG_MAP

END_MESSAGE_MAP()

////////////////////////////////////

// CTyssqlApp construction

CTyssqlApp::CTyssqlApp()

{

    // TODO: add construction code here,

    // Place all significant initialization in InitInstance

}

```

```

////////////////////////////////////
// The one and only CTyssqlApp object
CTyssqlApp NEAR theApp;
////////////////////////////////////
// CTyssqlApp initialization
BOOL CTyssqlApp::InitInstance()
{
    // Standard initialization

    // If you are not using these features and wish to reduce the size
    // of your final executable, you should remove from the following
    // the specific initialization routines you do not need.

    SetDialogBkColor();      // Set dialog background color to gray

    LoadStdProfileSettings(); // Load standard INI file options (including MRU)

    // Register the application's document templates. Document templates
    // serve as the connection between documents, frame windows and views.

    CSingleDocTemplate* pDocTemplate;

    pDocTemplate = new CSingleDocTemplate(

        IDR_MAINFRAME,

        RUNTIME_CLASS(CTyssqlDoc),

        RUNTIME_CLASS(CMainFrame),      // main SDI frame window

        RUNTIME_CLASS(CTyssqlView));

    AddDocTemplate(pDocTemplate);

    // create a new (empty) document

    OnFileNew();

    if (m_lpCmdLine[0] != '\0')
    {
        // TODO: add command line processing here
    }

    return TRUE;
}

```

```

////////////////////////////////////

// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
    CAboutDlg();

// Dialog Data
   //{{AFX_DATA(CAboutDlg)
    enum { IDD = IDD_ABOUTBOX };
    //}}AFX_DATA

// Implementation
protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support

   //{{AFX_MSG(CAboutDlg)
        // No message handlers
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
   //{{AFX_DATA_INIT(CAboutDlg)
    //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
   //{{AFX_DATA_MAP(CAboutDlg)
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)

```

```

   //{{AFX_MSG_MAP(CAboutDlg)

        // No message handlers

   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// App command to run the dialog
void CTyssqlApp::OnAppAbout()
{
    CAboutDlg aboutDlg;

    aboutDlg.DoModal();
}

/////////////////////////////////////////////////////////////////

// CTyssqlApp commands

// tyssqdoc.cpp : implementation of the CTyssqlDoc class
//

#include "stdafx.h"

#include "tyssql.h"

#include "tyssqset.h"

#include "tyssqdoc.h"

#ifdef _DEBUG

#undef THIS_FILE

static char BASED_CODE THIS_FILE[] = __FILE__;

#endif

/////////////////////////////////////////////////////////////////

// CTyssqlDoc

IMPLEMENT_DYNCREATE(CTyssqlDoc, CDocument)

BEGIN_MESSAGE_MAP(CTyssqlDoc, CDocument)

   //{{AFX_MSG_MAP(CTyssqlDoc)

        // NOTE - the ClassWizard will add and remove mapping macros here.

        // DO NOT EDIT what you see in these blocks of generated code!

   //}}AFX_MSG_MAP

```

```

END_MESSAGE_MAP()

////////////////////////////////////

// CTyssqlDoc construction/destruction
CTyssqlDoc::CTyssqlDoc()
{
    // TODO: add one-time construction code here
}

CTyssqlDoc::~CTyssqlDoc()
{
}

BOOL CTyssqlDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    // TODO: add reinitialization code here

    // (SDI documents will reuse this document)
    return TRUE;
}

////////////////////////////////////

// CTyssqlDoc diagnostics
#ifdef _DEBUG

void CTyssqlDoc::AssertValid() const
{
    CDocument::AssertValid();
}

void CTyssqlDoc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
}

#endif // _DEBUG

```

```

////////////////////////////////////
// CTyssqlDoc commands

// stdafx.cpp : source file that includes just the standard includes
// stdafx.pch will be the pre-compiled header
// stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"

// mainfrm.cpp : implementation of the CMainFrame class
//

#include "stdafx.h"

#include "tyssql.h"

#include "mainfrm.h"

#ifdef _DEBUG

#undef THIS_FILE

static char BASED_CODE THIS_FILE[] = __FILE__;

#endif

////////////////////////////////////

// CMainFrame

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)

   //{{AFX_MSG_MAP(CMainFrame)

        // NOTE - the ClassWizard will add and remove mapping macros here.

        // DO NOT EDIT what you see in these blocks of generated code !

        ON_WM_CREATE()

    }}AFX_MSG_MAP

END_MESSAGE_MAP()

////////////////////////////////////

// arrays of IDs used to initialize control bars

// toolbar buttons - IDs are command buttons

static UINT BASED_CODE buttons[] =

{

```

```

    // same order as in the bitmap 'toolbar.bmp'

    ID_EDIT_CUT,

    ID_EDIT_COPY,

    ID_EDIT_PASTE,

        ID_SEPARATOR,

    ID_FILE_PRINT,

        ID_SEPARATOR,

    ID_RECORD_FIRST,

    ID_RECORD_PREV,

    ID_RECORD_NEXT,

    ID_RECORD_LAST,

        ID_SEPARATOR,

    ID_APP_ABOUT,
};

static UINT BASED_CODE indicators[] =
{
    ID_SEPARATOR,          // status line indicator

    ID_INDICATOR_CAPS,

    ID_INDICATOR_NUM,

    ID_INDICATOR_SCRL,

};

////////////////////////////////////

// CMainFrame construction/destruction

CMainFrame::CMainFrame()
{
    // TODO: add member initialization code here
}

CMainFrame::~CMainFrame()
{
}

```



```

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

    if (!m_wndToolBar.Create(this) ||
        !m_wndToolBar.LoadBitmap(IDR_MAINFRAME) ||
        !m_wndToolBar.SetButtons(buttons,
            sizeof(buttons)/sizeof(UINT)))
    {
        TRACE("Failed to create toolbar\n");
        return -1;        // fail to create
    }

    if (!m_wndStatusBar.Create(this) ||
        !m_wndStatusBar.SetIndicators(indicators,
            sizeof(indicators)/sizeof(UINT)))
    {
        TRACE("Failed to create status bar\n");
        return -1;        // fail to create
    }

    return 0;
}

////////////////////////////////////

// CMainFrame diagnostics
#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
    CFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{

```

```
        CFrameWnd::Dump(dc);
    }

#ifdef _DEBUG
////////////////////////////////////
// CMainFrame message handlers
```

附件 C：第 14 天中的 Delphi 源代码清单

```
program Tyssql;

uses

    Forms,

    Unit1 in 'UNIT1.PAS' {Form1},
    Unit2 in 'UNIT2.PAS' {Form2};
{$R *.RES}

begin
    Application.CreateForm(TForm2, Form2);
    Application.CreateForm(TForm1, Form1);
    Application.Run;
end.

unit Unit1;

interface

uses

    SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
    Forms, Dialogs;

type
    TForm1 = class(TForm)
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    Form1: TForm1;

implementation

{$R *.DFM}
```

```
end.  
  
unit Unit2;  
  
interface  
  
uses  
  
    SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,  
    StdCtrls, Forms, DBCtrls, DB, DBGrids, DBTables, Grids, Mask, ExtCtrls;  
  
type  
  
    TForm2 = class(TForm)  
  
        ScrollBox: TScrollBox;  
  
        Label1: TLabel;  
  
        EditPARTNUM: TDBEdit;  
  
        Label2: TLabel;  
  
        EditDESCRIPTION: TDBEdit;  
  
        Label3: TLabel;  
  
        EditPRICE: TDBEdit;  
  
        DBGrid1: TDBGrid;  
  
        DBNavigator: TDBNavigator;  
  
        Panel1: TPanel;  
  
        DataSource1: TDataSource;  
  
        Panel2: TPanel;  
  
        Panel3: TPanel;  
  
        Query1: TQuery;  
  
        Query2: TQuery;  
  
        DataSource2: TDataSource;  
  
        procedure FormCreate(Sender: TObject);  
  
    private  
  
        { private declarations }  
  
    public  
  
        { public declarations }  
  
    end;
```

var

Form2: TForm2;

implementation

{\$R *.DFM}

procedure TForm2.FormCreate(Sender: TObject);

begin

Query1.Open;

Query2.Open;

end;

end.

附件 D：参考内容

书

《Developing Sybase Applications》

发行社： Sams

作者： Daniel J. Worden

ISBN: 0-672-30700-6

《Sybase Developer's Guide》

发行社： Sams

作者： Daniel J. Worden

ISBN: 0-672-30467-8

《Microsoft SQL Server 6.5 Unleashed, 2E》

发行社： Sams

作者： David Solomon, Ray Rankins, et al.

ISBN: 0-672-30956-4

《Teach Yourself Delphi in 21 Days》

发行社： Sams

作者： Andrew Wozniewicz

ISBN: 0-672-30470-8

《Delphi Developer's Guide》

发行社： Sams

作者： Steve Teixeira and Xavier Pacheco

ISBN: 0-672-30704-9

《Delphi Programming Unleashed》

出版社: Sams

作者: Charlie Calvert

ISBN: 0-672-30499-6

《Essential Oracle 7.2 》

出版社: Sams

作者: Tom Luers

ISBN: 0-672-30873-8

《 Developing Personal Oracle7 for Windows 95 Applications 》

出版社: Sams

作者: David Lockman

ISBN: 0-672-31025-2

《Teach Yourself C++ Programming in 21 Days 》

出版社: Sams

作者: Jesse Liberty

ISBN: 0-672-30541-0

《Teach Yourself Tansact-SQL in 21 Days 》

出版社: SAMS

作者: Bennett Wm. McEwan and David Solomon

ISBN: 0-672-31045-7

《Teach Yourself PL/SQL in 21 Days 》

出版社: SAMS

作者: Tom Luers, Timothy Atwood, and Jonathan Gennick

ISBN: 0-672-31123-2

请访问着站点www.mcp.com以得到更多的信息

杂志：

DBMS

P.O Box 469039
Escondido, CA 92046-9039
800-334-8152

Oracle Magazine

500 Oracle Parkway
Box 659510 Redwood Shores, CA 94065-1600
415-506-5304

SQL 的互联网资源

<http://www.aslaninc.com/>

Aslan 计算机有限公司：提供的 SQL 数据库，WINDOWS 开发工具，WINDOWS NT 组网以互联网服务技术

<http://www.radix.net/~ablaze/>

Ablaze 商业有限公司：主要提供 Microsoft 的 Visual Basic, MS Server, PowerBuilder, 以及 Internet 解决方案.

<http://www.fourgen.com/>

FourGen：提供支持 Windows, 4GL, UNIX, SQL, OLE 的开放系统标准.

<http://www.innovision1.com/steelep4/ddi.html>

Digital Dreamshop: 提供新型的客户机/服务器应用程序, 计算机图形服务和 Visual Basic, Access, Transact-SQL, C++, and Delphi 的通信软件程序

<http://www.novalink.com/bachman/index.html>

Bachman 信息系统：为 Sybase 和 Microsoft SQL Server 数据库提供设计工具和开发工具的供应商。

<http://www.everyware.com/>

EveryWare Development Corp: Butler SQL 的开发商，这种 SQL 数据库是 Macintosh 操作系统下的。

<http://www.edb.com/nb/index.html>

Netbase: 提供在 UNIX 下的底价的客户机/服务器型的 SQL。

<http://www.quadbase.com/quadbase.htm>

Quadbase: Quadbase-SQL 是有很高的性能和全部的、工业化的 RDBMS 数据库的 SQL。

<http://www.sagus.com/>

Software AG of North America (SAGNA): 市场和开发开放，多个平台下的解决方案

(ENTIRE)，应用程序引擎 (NATURAL)，SQL 查询和报表 (ESPERANT)，数据库管理 (ADABAS) 和数据仓库。

<http://www.nis.net/sqlpower/>

Sql Power Tools: 为 SQL 开发人员和管理员提供二次开发工具。

<http://world.std.com/~engwiz/>

English Wizard: 可以将英语翻译可访问你的数据库的 SQL 语句。

<http://www.microsoft.com/SQL/>

Microsoft 官方 SQL 网站。

http://www.jcc.com/sql_stnd.html

SQL Standards: 关于 SQL 标准化的资源信息和 SQL 当前的状况权威站点。

<http://www.sybase.com/WWW/>

联接着 Sybase SQL Server 的万维网站点。

<http://www.ncsa.uiuc.edu/SDG/People/jason/pub/gsql/starthere.html>

GSQL: 一个 Mosaic-SQL 网关。

FTP 站点

<ftp://ftp.cc.gatech.edu/pub/gvu/www/pitkow/gsql-oracle/oracle-backend.html>

GSQL: Oracle Backend.

新闻组

news:comp.databases.oracle

Usenet: The SQL database products of the Oracle Corporation.

news:comp.databases.sybase

Usenet: Implementations of the SQL Server.

附件 E: ACSLL 码表

Dec X_{16}	Hex X_{16}	Binary X_2	ASCII	Dec X_{16}	Hex X_{16}	Binary X_2	ASCII
000	00	0000 0000	null	026	1A	0001 1010	.
001	01	0000 0001	⊕	027	1B	0001 1011	,
002	02	0000 0010	⊕	028	1C	0001 1100	_
003	03	0000 0011	¥	029	1D	0001 1101	..
004	04	0000 0100	+	030	1E	0001 1110	^
005	05	0000 0101	♣	031	1F	0001 1111	~
006	06	0000 0110	♣	032	20	0010 0000	space
007	07	0000 0111	*	033	21	0010 0001	!
008	08	0000 1000	☐	034	22	0010 0010	"
009	09	0000 1001	◦	035	23	0010 0011	#
010	0A	0000 1010	☐	036	24	0010 0100	\$
011	0B	0000 1011	◊	037	25	0010 0101	%
012	0C	0000 1100	♀	038	26	0010 0110	&
013	0D	0000 1101	†	039	27	0010 0111	'
014	0E	0000 1110	♠	040	28	0010 1000	(
015	0F	0000 1111	♠	041	29	0010 1001)
016	10	0001 0000	-	042	2A	0010 1010	*
017	11	0001 0001	-	043	2B	0010 1011	+
018	12	0001 0010	;	044	2C	0010 1100	>
019	13	0001 0011	!!	045	2D	0010 1101	-
020	14	0001 0100	¶	046	2E	0010 1110	.
021	15	0001 0101	§	047	2F	0010 1111	/
022	16	0001 0110	-	048	30	0011 0000	0
023	17	0001 0111	;	049	31	0011 0001	1
024	18	0001 1000	†	050	32	0011 0010	2
025	19	0001 1001	!	051	33	0011 0011	3

Dec X_{10}	Hex X_{16}	Binary X_2	ASCII	Dec X_{10}	Hex X_{16}	Binary X_2	ASCII
052	34	0011 0100	4	078	4E	0100 1110	N
053	35	0011 0101	5	079	4F	0100 1111	O
054	36	0011 0110	6	080	50	0101 0000	P
055	37	0011 0111	7	081	51	0101 0001	Q
056	38	0011 1000	8	082	52	0101 0010	R
057	39	0011 1001	9	083	53	0101 0011	s
058	3A	0011 1010	:	084	54	0101 0100	T
059	3B	0011 1011	;	085	55	0101 0101	U
060	3C	0011 1100	<	086	56	0101 0110	V
061	3D	0011 1101	=	087	57	0101 0111	W
062	3E	0011 1110	>	088	58	0101 1000	X
063	3F	0011 1111	?	089	59	0101 1001	Y
064	40	0100 0000	@	090	5A	0101 1010	Z
065	41	0100 0001	A	091	5B	0101 1011	[
066	42	0100 0010	B	092	5C	0101 1100	\
067	43	0100 0011	C	093	5D	0101 1101]
068	44	0100 0100	D	094	5E	0101 1110	^
069	45	0100 0101	E	095	5F	0101 1111	-
070	46	0100 0110	F	096	60	0110 0000	`
071	47	0100 0111	G	097	61	0110 0001	a
072	48	0100 1000	H	098	62	0110 0010	b
073	49	0100 1001	I	099	63	0110 0011	c
074	4A	0100 1010	J	100	64	0110 0100	d
075	4B	0100 1011	K	101	65	0110 0101	e
076	4C	0100 1100	L	102	66	0110 0110	f
077	4D	0100 1101	M	103	67	0110 0111	g

Dec X_{10}	Hex X_{16}	Binary X_2	ASCII	Dec X_{10}	Hex X_{16}	Binary X_2	ASCII
104	68	0110 1000	h	130	82	1000 0010	é
105	69	0110 1001	i	131	83	1000 0011	ê
106	6A	0110 1010	j	132	84	1000 0100	ë
107	6B	0110 1011	k	133	85	1000 0101	ü
108	6C	0110 1100	l	134	86	1000 0110	ä
109	6D	0110 1101	m	135	87	1000 0111	q
110	6E	0110 1110	n	136	88	1000 1000	è
111	6F	0110 1111	o	137	89	1000 1001	é
112	70	0111 0000	p	138	8A	1000 1010	ê
113	71	0111 0001	q	139	8B	1000 1011	ï
114	72	0111 0010	r	140	8C	1000 1100	î
115	73	0111 0011	s	141	8D	1000 1101	ï
116	74	0111 0100	t	142	8E	1000 1110	Ä
117	75	0111 0101	u	143	8F	1000 1111	Å
118	76	0111 0110	v	144	90	1001 0000	Ê
119	77	0111 0111	w	145	91	1001 0001	æ
120	78	0111 1000	x	146	92	1001 0010	Æ
121	79	0111 1001	y	147	93	1001 0011	ø
122	7A	0111 1010	z	148	94	1001 0100	ö
123	7B	0111 1011	{	149	95	1001 0101	ö
124	7C	0111 1100		150	96	1001 0110	û
125	7D	0111 1101	}	151	97	1001 0111	ü
126	7E	0111 1110	~	152	98	1001 1000	ÿ
127	7F	0111 1111	Δ	153	99	1001 1001	Ö
128	80	1000 0000	Ç	154	9A	1001 1010	Û
129	81	1000 0001	ü	155	9B	1001 1011	ø

Dec X ₁₀	Hex X ₁₆	Binary X ₂	ASCII	Dec X ₁₀	Hex X ₁₆	Binary X ₂	ASCII
156	9C	1001 1100	£	182	B6	1011 0110	
157	9D	1001 1101	¥	183	B7	1011 0111	π
158	9E	1001 1110	ℙ	184	B8	1011 1000	¶
159	9F	1001 1111	ƒ	185	B9	1011 1001	
160	A0	1010 0000	ı	186	BA	1011 1010	
161	A1	1010 0001	í	187	BB	1011 1011	ŋ
162	A2	1010 0010	đ	188	BC	1011 1100	Ɔ
163	A3	1010 0011	ă	189	BD	1011 1101	„
164	A4	1010 0100	ä	190	BE	1011 1110	ƶ
165	A5	1010 0101	Ħ	191	BF	1011 1111	γ
166	A6	1010 0110	º	192	CO	1100 0000	ˆ
167	A7	1010 0111	°	193	C1	1100 0001	⊥
168	A8	1010 1000	ı	194	C2	1100 0010	┐
169	A9	1010 1001	ı	195	C3	1100 0011	┐
170	AA	1010 1010	ı	196	C4	1100 0100	—
171	AB	1010 1011	‰	197	C5	1100 0101	+
172	AC	1010 1100	‰	198	C6	1100 0110	┐
173	AD	1010 1101	ı	199	C7	1100 0111	┐
174	AE	1010 1110	α	200	C8	1100 1000	ℓ
175	AF	1010 1111	α	201	C9	1100 1001	ℓ
176	B0	1011 0000	■	202	CA	1100 1010	≡
177	B1	1011 0001	■	203	CB	1100 1011	≡
178	B2	1011 0010	■	204	CC	1100 1100	
179	B3	1011 0011		205	CD	1100 1101	=
180	B4	1011 0100	†	206	CE	1100 1110	
181	B5	1011 0101	†	207	CF	1100 1111	⊥

Dec X_{10}	Hex X_{16}	Binary X_2	ASCII	Dec X_{10}	Hex X_{16}	Binary X_2	ASCII
208	D0	1101 0000	„	234	EA	1110 1010	Ω
209	D1	1101 0001	〒	235	EB	1110 1011	δ
210	D2	1101 0010	π	236	EC	1110 1100	∞
211	D3	1101 0011	Ł	237	ED	1110 1101	ø
212	D4	1101 0100	Ł	238	EE	1110 1110	€
213	D5	1101 0101	ƒ	239	EF	1110 1111	∩
214	D6	1101 0110	π	240	F0	1110 0000	≡
215	D7	1101 0111	‖	241	F1	1111 0001	±
216	D8	1101 1000	φ	242	F2	1111 0010	≥
217	D9	1101 1001	Ј	243	F3	1111 0011	≤
218	DA	1101 1010	ƒ	244	F4	1111 0100	ƒ
219	DB	1101 1011	■	245	F5	1111 0101	ƒ
220	DC	1101 1100	■	246	F6	1111 0110	+
221	DD	1101 1101	■	247	F7	1111 0111	=
222	DE	1101 1110	■	248	F8	1111 1000	*
223	DF	1101 1111	■	249	F9	1111 1001	*
224	E0	1110 0000	α	250	FA	1111 1010	*
225	E1	1110 0001	β	251	FB	1111 1011	√
226	E2	1110 0010	Γ	252	FC	1111 1100	∞
227	E3	1110 0011	π	253	FD	1111 1101	∞
228	E4	1110 0100	Ł	254	FE	1111 1110	■
229	E5	1110 0101	σ	255	FF	1111 1111	
230	E6	1110 0110	μ				
231	E7	1110 0111	γ				
232	E8	1110 1000	Φ				
233	E9	1110 1001	θ				

附件 F：问题与练习答案

第一天：SQL 简介

问题答案

1、为什么说 SQL 是一种非过程型语言。

SQL 决定了应该做什么而不是如何去做。数据库必需实现 SQL 的要求，这种特性在交叉平台和交叉语言开发环境中非常有利。

2、我如何知道一种数据库系统是不是关系型数据库系统？

根据 Dr. Codd's 12（我们知道共有 13 条）的规则。

3、我可以用 SQL 来做什么？

SQL 可以让你选择、插入、修改和删除数据库中的信息，执行系统安全功能和设置用户对表和数据库的访问权限。对应用程序的在线事务过程进行控制，创建存储过程和触发机制以减少应用程序代码并可以在不同的数据库中传输数据。

4、把数据清楚地分成一个个唯一集的过程叫什么名字。

标准化，它可以减少数据的冗余度和数据库结构的复杂性。

练习答案

确认一下你所使用的数据库系统是否是一个关系型数据库系统。（你自己回答）

第二天：查询——SELECT 语句的使用

问题答案

4、 下列语句所返回的结果是否相同？

```
SELECT * FROM CHECKS;
```

```
select * from checks;
```

这两条语句只有大小写是不同的，一般来说 SQL 对于大小写是不敏感的，所以它们返回相同的结果。但是涉及到具体的数据时应该注意大小写。

5、 为什么下列查询不会工作？

a. `Select *`

没有 FROM 子句，对于是个 SELECT 语句来说 SELECT 和 FROM 都是必须的。

b. `Select * from checks`

没有分号，它是用来标明语句是否结束的，

c. `Select amount name payee FROM checks;`

你需要在每个列名之间使用逗号，`Select amount, name, payee FROM checks;`

3、 下列的查询中哪一个可以工作？

a. `select *`

`from checks;`

b. `select * from checks;`

c. `select * from checks`

/

它们都可以工作。

练习答案

1、 使用今天早些时候的 CHECKS 表的数据来写一个查询，返回表中的 number 和 remark 列中的数据。

```
SELECT CHECK#, REMARKS FROM CHECKS;
```

2、 将练习 1 中的查询再写一遍以使得 remark 列出现在第一位。

```
SELECT REMARKS, CHECK# FROM CHECKS;
```

3、 使用 CHECKS 表，写一个查询来返回其中的不重复数据。

```
SELECT DISTINCT REMARKS FROM CHECKS;
```


第三天：表达式、条件语句与运算

问题答案

应用下表的内容来回答下列问题：

LASTNAME	FIRSTNAME	AREACODE	PHONE	ST	ZIP
BUNDY	AL	100	555-1111	IL	22333
MEZA	AL	200	555-2222	UK	
MERRICK	BUD	300	555-6666	CO	80212
MAST	JD	381	555-6767	LA	23456
BULHER	FERRIS	345	555-3223	IL	23332
PERKINS	ALTON	911	555-3116	CA	95633
BOSS	SIR	204	555-2345	CT	95633

- 1、写一下查询，返回数据库中所有名字以 M 开头的每一个人。

```
SELECT * FROM FRIENDS WHERE LASTNAME LIKE 'M%';
```

- 2、写一个查询，返回数据库 ST 为 LA 且 FIRSTNAME 以 AL 开头的人。

```
SELECT * FROM FRIENDS WHERE STATE = 'IL' AND FIRSTNAME = 'AL';
```

- 3、给你两个表（PART1 和 PART2），你如何才能找出两个表中的共有元素。请写出查询。

使用 INTERSECT，切记，INTERSECT 将返回两个查询中的公共行

```
SELECT PARTNO FROM PART1
```

```
INTERSECT
```

```
SELECT PARTNO FROM PART2;
```

- 4、WHERE a >= 10 AND a <=30 的更便捷写法是什么，请写出来？

```
WHERE a BETWEEN 10 AND 30;
```

- 5、下面的查询将返回什么结果？

```
SELECT FIRSTNAME FROM FRIENDS WHERE FIRSTNAME='AL' AND  
LASTNAME='BULHER';
```

什么也不会返回，没有同时满足这两个条件的记录。

练习答案

- 1、用上边给出的表返回下面的结果。

```
NAME          ST
```

AL FROM IL

INPUT:

SQL> SELECT (FIRSTNAME || 'FROM') NAME, STATE

2 FROM FRIENDS

3 WHERE STATE = 'IL' AND

5 LASTNAME = 'BUNDY';

OUTPUT:

NAME	ST
AL FROM IL	

2、仍使用上表，返回以下结果

NAME	PHONE
MERRICK, BUD	300-555-6666
MAST, JD	381-555-6767
BULHER, FERRIS	345-555-3223

INPUT:

SQL> SELECT LASTNAME || ' ' || FIRSTNAME NAME,

2 AREACODE || '-' || PHONE PHONE

3 FROM FRIENDS

4 WHERE AREACODE BETWEEN 300 AND 400;

第四天：函数：对获得数据的进一步处理

问题答案

8、哪个函数是用来将给定字符串的第一个字母变成大写而把其它的字符变成小写的？

INITCAP

9、哪些函数的功能就如同它的名字含义一样？

Group 函数了 aggregate 函数都是。

10、下边的查询将如何工作？

SQL> SELECT COUNT(LASTNAME) FROM CHARACTERS;

它将返回表中的总记录数。

11、 下边的查询是干什么的？

```
SQL> SELECT SUM(LASTNAME) FROM CHARACTERS;
```

由于 LASTNAME 是字符串类型，所以这个查询将不会工作。

12、 哪个函数可以将 FIRSTNAME 列与 LASTNAME 列合并到一起？

CONCAT 函数和||符号都可以

13、 在下边的查询中，6 是什么意思？

INPUT:

```
SQL> SELECT COUNT(*) FROM TEAMSTATS;
```

OUTPUT:

```
COUNT(*)
```

```
6
```

6 是指表中的记录个数。

14、 下列语句将是否会工作？

```
SQL> SELECT SUBSTR LASTNAME,1,5 FROM NAME_TBL;
```

由于在 lastname,1,5 周围没有括号，所以它会认为这是列的别名，正确的语句如下：

```
SQL> SELECT SUBSTR(LASTNAME,1,5) NAME FROM NAME_TBL;
```

练习答案

3、 用今天的 TEAMSTATS 表来写一个查询，用来显示谁的中球率低于 0.25（中球率

的计算方法为 hits/ab）

INPUT:

```
SQL> SELECT NAME FROM TEAMSTATS
```

```
2 WHERE (HITS/AB) < .25;
```

OUTPUT:

```
NAME
```

```
HAMHOCKER
```

```
CASEY
```

4、 用今天的 CHARACTERS 表来写一个查询，要求返回下边的结果

```
INITIALS_____CODE
```

```
K.A.P.          32
```

```
1 row selected.
```

```
SQL> select substr(firstname,1,1)||'.'||
```

```
substr(middlename,1,1)||'.'||
```

```
substr(lastname,1,1)||'.' INITIALS, code
```

```
from characters  
where code = 32;
```

第五天：SQL 中的子句

问题答案

6、哪种子句的作用与 LIKE (<exp>%) 相似？

STARTING WITH

7、GROUP BY 子句的功能是什么，哪种子句的功能与它类似？

GROUP BY 子句可以将其它函数返回的数据进行分组，它与 ORDER BY 子句在行为上类似，ORDER BY 子句是将查询的结果像 GROUP BY 子句一样根据给出的列进行排序。

8、下面的查询会工作吗？

INPUT:

```
SQL>SELECT NAME, AVG (SALARY), DEPARTMENT FROM PAY_TBL  
WHERE DEPARTMENT='ACCOUNTING' ORDER BY NAME  
GROUP BY DEPARTMENT, SALARY;
```

语法不正确，GROUP BY 必须在 ORDER BY 之前，而且所有选择的列也必需在 GROUP BY 中给出。

9、在使用 HAVING 子句时是否要同时使用 GROUP BY 子句？

是的

10、你可以使用在 SELECT 语句中没有出现的列进行排序吗？

可以，排序时要排序的列不必一定要在 SELECT 语句中出现。

练习答案

3、使用上例中的 ORGCHART 表找一下每一个 TEAM 中 SICKLEAVE 天数超过 30 天的人数。

先来看一下每组有多少个人：

INPUT:

```
SELECT TEAM, COUNT(TEAM)
```

FROM ORGCHART

GROUP BY TEAM;

OUTPUT:

TEAM	COUNT
COLLECTIONS	2
MARKETING	3
PR	1
RESEARCH	2

将它与下边的答案比较

INPUT:

SELECT TEAM, COUNT(Team)

FROM ORGCHART

WHERE SICKLEAVE >=30

GROUP BY TEAM;

OUTPUT:

TEAM	COUNT
COLLECTIONS	1
MARKETING	1
RESEARCH	1

输出显示了每一个组中病假数超过 30 天的人。

4、使用 CHECKS 表，返回如下结果：

OUTPUT:

CHECK#	PAYEE	AMOUNT
1	MA BELL	150

INPUT:

SQL> SELECT CHECK#, PAYEE, AMOUNT

FROM CHECKS

WHERE CHECK# = 1;

有多种方法可以完成这个问题，你能找出其它的吗？

第六天：表的联接

问题答案

6、如果一个表有 50000 行而另一个表有 100000 行时联接的结果会有多少行？

5,000,000,000 行

7、下边的联接属于哪一种类型的联接？

```
SELECT E.NAME, E.EMPLOYEE_ID, EP.SALARY FROM EMPLOYEE_TBL E, EMPLOYEE_PAY_TBL EP  
WHERE E.EMPLOYEE_ID = EP.EMPLOYEE_ID;
```

这是一个等值联接，你可以从中发现所以存在于两个表中的雇员 ID。

8、下边的查询语句能否工作？

```
A. SELECT NAME, EMPLOYEE_ID, SALARY FROM EMPLOYEE_TBL E, EMPLOYEE_PAY_TBL EP  
WHERE EMPLOYEE_ID = EMPLOYEE_ID AND NAME LIKE '%MITH';
```

不会，列句和表名不正确，要注意使用列和表的别名。

```
select e.name, e.employee_id, ep.salary  
from employee_tbl e,  
employee_pay_tbl ep  
where name like '%MITH';
```

```
B. SELECT E.NAME, E.EMPLOYEE_ID, EP.SALARY FROM EMPLOYEE_TBL E, EMPLOYEE_PAY_TBL EP  
WHERE NAME LIKE '%MITH';
```

不会，在 WHERE 子句中没有联接命令。

```
select e.name, e.employee_id, ep.salary  
from employee_tbl e,  
employee_pay_tbl ep  
where e.employee_id = ep.employee_id  
and e.name like '%MITH';
```

```
C. SELECT E.NAME, E.EMPLOYEE_ID, EP.SALARY FROM EMPLOYEE_TBL E,EMPLOYEE_PAY_TBL EP  
WHERE E.EMPLOYEE_ID = EP.EMPLOYEE_ID AND E.NAME LIKE '%MITH';
```

正确的。

9、是否在联接语句中 WHERE 子句中的第一个条件应该是联接条件？

联接命令应该在条件的前边

10、 联接是否限制为一列，是否可以有更多的列？
可以

练习答案

4、在表的自我联接这部分，最后的一个例子返回了两个结果，请重写这个查询使它对多余的记录只返回一个结果。

INPUT/OUTPUT:

```
SELECT F.PARTNUM, F.DESCRPTION,
S.PARTNUM,S.DESCRPTION
FROM PART F, PART S
WHERE F.PARTNUM = S.PARTNUM
AND F.DESCRPTION <> S.DESCRPTION
AND F.DESCRPTION > S.DESCRPTION
```

PARTNUM	DESCRIPTION	PARTNUM	DESCRIPTION
76	ROAD BIKE	76	CLIPPLESS SHOE

5、重写下边的查询使它更可读和简炼。

INPUT:

```
select orders.orderedon, orders.name, part.partnum,part.price, part.description
from orders, part
where orders.partnum = part.partnum and orders.orderedon
between '1-SEP-96' and '30-SEP-96' order by part.partnum;
```

答: select o.orderedon ORDER_DATE, o.name NAME, p.partnum PART#,
p.price PRICE, p.description DESCRIPTION
from orders o, part p
where o.partnum = p.partnum
and o.orderedon like '%SEP%'
order by ORDER_DATE;

6、使用 ORDERS 表和 PART 表，返回下边的结果。

OUTPUT:

ORDEREDON	NAME	PARTNUM	QUANTITY
2-SEP-96	TRUE WHEEL	10	1

答: Select o.orderedon ORDEREDON, o.name NAME,

p.partnum PARTNUM, o.quantity QUANTITY

from orders o,part p

where o.partnum = p.partnum

and o.orderedon like '%SEP%';

还有许多种写法。

第 7 天：“子查询：内嵌的 SELECT 语句”

问题答案

1、在嵌套查询部分，有一个例子中子查询返回了以下几个数值：

```
LE SHOPPE
BIKE SPEC
LE SHOPPE
BIKE SPEC
JACKS BIKE
```

其中有一些结果是重复的，为什么在最终的结果中没有出现重复？

由于查询调用了子查询，所以在最后的结果中没有出现重复

```
SELECT ALL C.NAME, C.ADDRESS, C.STATE,C.ZIP
FROM CUSTOMER C
WHERE C.NAME IN (.....)
```

只会返回 NAME 的州为 IN 的记录，这里不要为 IN 在复杂语句里的简写所迷糊。

2、下面的话是对还是错？

(4) 汇总函数如 SUM、AVG、COUNT、MAX、MIN 都返回多个数值。(不对，它们都返回多个数值)

(5) 子查询最多允许嵌套两层。(不对，嵌套层数的限制是根据你的解释器而定的)

(6) 相关子查询是完全的独立查询。(不对，相关子查询中你使用外部引用)

3、下边的子查询中哪一个是使用 ORDERS 表和 PART 表工作的？

INPUT/OUTPUT:

SQL> SELECT * FROM PART;

PARTNUM	DESCRIPTION	PRICE
54	PEDALS	54.25
42	SEATS	24.50
46	TIRES	15.25
23	MOUNTAIN BIKE	350.45
76	ROAD BIKE	530.00
10	TANDEM	1200.00

INPUT/OUTPUT:

SQL> SELECT * FROM ORDERS;

ORDEREDON	NAME	PARTNUM	QUANTITY	REMARKS
15-MAY-96	TRUE WHEEL	23	6	PAID
19-MAY-96	TRUE WHEEL	76	3	PAID
2-SEP-96	TRUE WHEEL	10	1	PAID
30-JUN-96	BIKE SPEC	54	10	PAID
30-MAY-96	BIKE SPEC	10	2	PAID
30-MAY-96	BIKE SPEC	23	8	PAID
17-JAN-96	BIKE SPEC	76	11	PAID
17-JAN-96	LE SHOPPE	76	5	PAID
1-JUN-96	LE SHOPPE	10	3	PAID
1-JUN-96	AAA BIKE	10	1	PAID
1-JUN-96	AAA BIKE	76	4	PAID
1-JUN-96	AAA BIKE	46	14	PAID
11-JUL-96	JACKS BIKE	76	14	PAID

A、 SQL> SELECT * FROM ORDERS WHERE PARTNUM =

SELECT PARTNUM FROM PART

WHERE DESCRIPTION = 'TRUE WHEEL';

不对，在子查询的周围没有括号

B、 SQL> SELECT PARTNUM FROM ORDERS WHERE PARTNUM =

(SELECT * FROM PART

WHERE DESCRIPTION = 'LE SHOPPE');

不会，SQL 引擎在=操作后边关联表的所有了

```
C> SQL> SELECT NAME, PARTNUM FROM ORDERS WHERE EXISTS
      (SELECT * FROM ORDERS
      WHERE NAME = 'TRUE WHEEL');
对的，这个查询是正确的。
```

练习答案

应用 ORDERS 表来写一个查询，返回所以字母顺序排列在 JACKS BIKE 之后的 NAMES 和 ORDEREDON 数据。

INPUT/OUTPUT:

```
SELECT NAME, ORDEREDON
FROM ORDERS
WHERE NAME >
      (SELECT NAME
      FROM ORDERS
      WHERE NAME ='JACKS BIKE')
```

NAME	ORDEREDON
TRUE WHEEL	15-MAY-1996
TRUE WHEEL	19-MAY-1996
TRUE WHEEL	2-SEP-1996
TRUE WHEEL	30-JUN-1996
LE SHOPPE	17-JAN-1996
LE SHOPPE	1-JUN-1996

第八天：操作数据

问题答案

9、下边的语句有什么错误？

```
DELETE COLLECTION;
```

如果你想删除 COLLECZTION 表中的所有记录，你必需使用下边的语法：

```
DELETE FROM COLLECTION;
```

要注意这条语句会删除表中的所有的记录，你可以使用下边的语法来有选择地删除表中的记录。

```
DELETE FROM COLLECTION
```

```
WHERE VALUE = 125
```

这条语句将会删除所有 VALUE 为 125 的记录，

10、 下边的语句有什么错误？

```
INSERT INTO COLLECTION SELECT * FROM TABLE_2
```

这条语句的目的是将 TABLE_2 中的所有记录都拷贝到 COLLECTION 表中。

这里存在的主要问题是 IN INSERT 语句中使用了 INTO 关键字。在把一个表中的数据拷贝到另一个表中时，你必需使用下边的语法：

```
INSERT COLLECTION
```

```
SELECT * FROM TABLE_2;
```

同时，要记住 TABLE_2 中的数据类型和字段次序与 COLLECTION 表中的一样。

11、 下边的语句有什么错误？

```
UPDATE COLLECTION ("HONUS WAGNER CARD", 25000, "FOUND IT");
```

这条语句把 UPDATE 与 INSERT 弄混了，如果想更新 COLLECTIONS 表中的数值，你应该使用下边的语法：

```
UPDATE COLLECTIONS
```

```
SET NAME = "HONUS WAGNER CARD",
```

```
VALUE = 25000,
```

```
REMARKS = "FOUND IT";
```

12、 如果执行下边的语句会有什么结果？

```
SQL> DELETE * FROM COLLECTION;
```

由于语法不正确，什么也不会删除，这里不需要*号。

13、 如果执行下边的语句会有什么结果？

```
SQL> DELETE FROM COLLECTION;
```

在 COLLECTION 表中的所有记录都会被删除

14、 如果执行下边的语句会有什么结果？

```
SQL> UPDATE COLLECTION SET WORTH = 555
```

```
SET REMARKS = 'UP FROM 525';
```

在 COLLECTION 表中的所有 WORTH 现在都变成了 555，而且所有的 REMARKS 都变成了 UP FROM 525。这可不是一件好事!!

15、 下边的语句是否会工作?

```
SQL> INSERT INTO COLLECTION SET VALUES = 900 WHERE ITEM = 'STRING';
```

不会工作，INSERT 与 SET 不可能在一起工作。

16、 下边的语句是否会工作?

```
SQL> UPDATE COLLECTION SET VALUES = 900 WHERE ITEM = 'STRING';
```

可以工作，语法完全正确。

练习答案

- 3、 试着向一个表中插入一个不正确的数据类型，看一下出错信息，然后再插入一个正确的数据类型。

无论你使用何种类型的解释器，你都会收到一个所插入数据的类型与表中对应列的数据类型不匹配的错误信息。

- 4、 试着使用你的数据库系统将某个表导出为其他库格式，然后再把它导入。熟悉一下你的数据库系统的导入与导出操作。并试着用其它数据库操作导出文件。

对于确切的导入和导出数据的语法请参见你的文档，如果你重复导入的话你可以想删除导入表中的所有数据，在真正进行数据操作之前你应该先熟悉一下导入导出命令，如果在你的表中存在唯一约束列那么你的导入操作可能会失败。你将会收到大量的关于唯一约束限制的错误。

第九天：创建和操作表

问题答案

- 9、 ALTER DATABASE 语句经常用在修改已有表的结构上，对不对?

不对，大多数数据系统中没有 ALTER DATABASE 命令，修改已有表的结构应该用 ALTER TABLE 命令。

10、 DROP TABLE 语句与 DELETE FROM <table_name>的作用是相同的，对不对？

不对，这两个命令并不等价，DROP TABLE 将会把表的记录及结构从数据库中全部删除。而 DELETE FROM <table_name>则只是将表中的记录全部删除，表的结构在数据库中依然存在。

11、 可以使用 CREATE TABLE 命令向数据库中加入一个新表，对不对？（对）

12、 为什么下边的语句是错误的？

INPUT:

```
CREATE TABLE new_table (  
    ID NUMBER,  
    FIELD1 char(40),  
    FIELD2 char(80),  
    ID char(40);
```

该语句存在着两个问题，首先是 ID 的语句在表中重复了，而它们的数据类型并不同，字段名重复使用是非法的。其次是在语句的末尾没有对应的圆括号，正确的语句应该如下：

INPUT:

```
CREATE TABLE new_table (  
    ID NUMBER,  
    FIELD1 char(40),  
    FIELD2 char(80));
```

13、 为什么下边的语句是错误的？

INPUT:

```
ALTER DATABASE BILLS (  
    COMPANY char(80));
```

更新字段名称和长度应该使用 ALTER TABLE 命令，而不是 ALTER DATABASE 命令。

14、 当一个表建立时，谁是它的所有者？

它的创建人，如果你用你的 ID 登录，那么所有都是你的 ID，如果你用 SYSTEM 登录，那么所有都是 SYSTEM。

15、 如果字符型列的长度在不断变化，如何才能做出最佳的选择？

VARCHAR2 是最好的选择，它可以允许存储在其中的字符串的长度变化。

16、 表名是否可以重复？

可以，只要所有者或计划不同就行。

练习答案

4、 用你喜欢的格式向 BILLS 数据库中加入两个表，名字分别叫 BANK 和 ACCOUNT_TYPE，BANK 表中应该包含有 BANK_ACCOUNT 表中 BANK 字段的信息，ACCOUNT_TYPE 表中也应该包含有 BANK_ACCOUNT 表中 ACCOUNT_TYPE 字段的信息，试着尽可能地减少数据的数量。

你应该使用 CREAETE TABLE 命令来创建表，可能的语句形式如下：

```
SQL> CREATE TABLE BANK
```

```
2  ( ACCOUNT_ID      NUMBER(30)          NOT NULL,
    BANK_NAME        VARCHAR2(30)        NOT NULL,
    ST_ADDRESS        VARCHAR2(30)        NOT NULL,
    CITY              VARCHAR2(15)        NOT NULL,
    STATE             CHAR(2)             NOT NULL,
    ZIP               NUMBER(5)           NOT NULL;
```

```
SQL> CREATE TABLE ACCOUNT_TYPE
```

```
( ACCOUNT_ID      NUMBER(30)          NOT NULL,
  SAVINGS          CHAR(30),
  CHECKING         CHAR(30);
```

5、 使用你已经创建的五個表，BILLS、BANK_ACCOUNTS、COMPANY、BANK、ACCOUNT_TYPE，改为表的结构以用整数型字段作为关键字以取代字符型字段作为关键字。

```
SQL> ALTER TABLE BILLS DROP PRIMARY KEY;
```

```
SQL> ALTER TABLE BILLS ADD (PRIMARY KEY (ACCOUNT_ID));
```

```
SQL> ALTER TABLE COMPANY ADD (PRIMARY KEY (ACCOUNT_ID));
```

6、 使用你所知道的 SQL 的联接知识（见第 6 天：表的联接），写几个查询来联接 BILLS 数据库中的几个表

由于在上一个练习中我们已经修改了表的结构使其关键字段为 ACCOUNT_ID 列，所以的表都可以根据该列进行联接。你可以对表进行任何联接。你也可以联接所有的表。别忘了对你的表中的列进行限制。

第 10 天 创建视图和索引

问题答案

7、 当在一个不唯一的字段中创建一个唯一值索引会有什么结果？

根据你所使用的数据库，你将会收到不同类型的错误表达，索引将不会建立，唯一值索引所建立的字段的内容必须是唯一的。

8、 下边的话是对是错

视图和索引都会占用数据库的空间，所以在设计数据库空间时要考虑到这一点。

（不对，只有索引才会占用数据库的存储空间）

如果一个人更新了一个已经创建视图的表，那么视图必须进行同样的更新才会看到相同的数据。（不对，表只要更新了，那么在视图中就可以看到更新了的数据）

如果你的磁盘空间够而你想加快你的查询的速度，那么索引越多越好。

（不对，有时索引反而会降低查询的速度。）

9、 下边的 CREATE 语句是否正确？

```
SQL> create view credit_debts as (select all from debts where account_id = 4);
```

（不对，你不应该使用括号，而且 ALL 应该是*）

10、 下边的 CREATE 语句是否正确？

```
SQL> create unique view debts as select * from debts_tbl;
```

（不对，唯一值视图不能这样做）

11、 下边的 CREATE 语句是否正确？

```
SQL> drop * from view debts;
```

（不对，正确的语法应该是 DROP VIEW DEBTS;）

12、 下边的 CREATE 语句是否正确？

```
SQL> create index id_index on bills (account_id); （正确）
```

练习答案：

- 4、检查你所使用的数据库系统，它是否支持视图？允许你在创建视图时使用哪些选项？用它的语法来写一个简单的创建视图语句，并对其进行如 SELECT 和 DELETE 等常规操作后再删除视图。（检查你的解释器的数据字典，看它是否支持将表的查询开放为视图）
- 5、检查你所使用的数据库系统看它是否支持索引？它有哪些选项？在你的数据库系统中的一些已经存在的表中试一下这些选项。进一步，确认在你的数据库系统中是否支持 UNIQUE 和 CLUSTER 索引。

（ACCESS 可以让开发人员以图形向表中加入索引，索引可以组合多个字段，也可以在图形用户界面下设置排序情况。而其它的系统则需要你在命令行下输入 CREATE INDEX 语句）

- 6、如果可能的话，在一个表中输入几千条记录，用秒表或钟来测定一下你的数据库系统对特定操作的反映时间，加入索引是否使性能提升了？试一下今天提到的技巧。

（只有当操作返回的数据量很少时索引才会对性能的提高有帮助，如果查询返回的数据占表的固有数据的比例较大，使用索引反而会使性能降低）

第 11 天：事务处理控制

问题答案

- 5、在嵌套的事务中，是否可以使用 ROLLBACK 命令来取消当前事务并回退到上级事务中，为什么？

（不可以，当事务嵌套时，任何 ROLLBACK 命令都将取消当前进程中的所有事务，所有在当前事务中所进行的改动都不会真正保存，直到外部事务被确认）

- 6、使用保存点是否可以保存事务的一部分，为什么？

（可以，保存点可以让程序员保存事务中的语句，如果需要，ROLLBACK 可以退回到保存点而不是事务的开始）

- 7、COMMIT 命令是否可以单独使用，它一定要嵌套吗？

（COMMIT 命令可以在事务中使用也可以单独使用）

- 8、如果你在 COMMIT 命令后发现的错误，你是否还可以使用 ROLLBACK 命令？

（也行也不行，你可以使用这个命令，但是你无法撤消所做的改动）

9、在事务中使用保存点是否可以自动地将之前的改动自动地保存？

（不是，保存点只有在使用 ROLLBACK 命令时才发挥作用，这时只有保存点以后的改动可以撤消）

练习答案

4、使用 PERSONAL ORACLE7 的语法来更正下边的语法

```
SQL> START TRANSACTION INSERT INTO CUSTOMERS VALUES ('SMITH', 'JOHN')
```

```
SQL> COMMIT;
```

答：

```
SQL> SET TRANSACTION;
```

```
INSERT INTO CUSTOMERS VALUES
```

```
('SMITH', 'JOHN');
```

```
SQL> COMMIT;
```

5、使用 PERSONAL ORACLE7 的语法来更正下边的语法

```
SQL> SET TRANSACTION;
```

```
UPDATE BALANCES SET CURR_BAL = 25000;
```

```
SQL> COMMIT;
```

答：

```
SQL> SET TRANSACTION;
```

```
UPDATE BALANCES SET CURR_BAL = 25000;
```

```
SQL> COMMIT;
```

该语句被更正后会很好地工作，但是，你只是更新了每个人的当前 BALANCE 为\$25,000!

6、使用 PERSONAL ORACLE7 的语法来更正下边的语法

```
SQL> SET TRANSACTION;
```

```
INSERT INTO BALANCES VALUES ('567.34', '230.00', '8');
```

```
SQL> ROLLBACK; （该语法完全正确，无需更改）
```

第 12 天：数据库安全

问题答案

8、下边的语句是否是错误的？

```
SQL> GRANT CONNECTION TO DAVID;
```

（没有 CONNECTION 规则，正确的语法是：

```
SQL> GRANT CONNECT TO DAVID;）
```

9、对与错：当删除用户时所有属于用户对对象都会随之删除。

（如果在删除用户时 CASCADE 语句运行了这句话就是对的，CASCADE 选项将通知系统删除所有为被删除用户所有的表）

10、如果你创建了一个表并对它使用了 SELECT 权限对象为 PUBLIC 时会有什么问题？

（任何人都可以从你的表中检索数据，即便是你不想让他看到内容的用户）

11、下边的 SQL 语句是否正确？

```
SQL> create user RON identified by RON;
```

（正确，该语句会创建一个用户，但是用户的设置是默认的，这可能不会让人满意，关于设置部分，你应该检查你的解释器文档）

12、下边的 SQL 语句是否正确？

```
SQL> alter RON identified by RON;
```

（不对，没有给出用户，正确的语法是：SQL> alter user RON identified by RON;）

13、下边的 SQL 语句是否正确？

```
SQL> grant connect, resource to RON; （正确）
```

14、如果表为你所有，别人如何才能从表中选择数据？

（只有对你的表拥有选择权限的用户才可以从你的表中选择数据）

练习答案

作为数据库安全性的练习，请你创建一个表，然后再创建一个用户，为该用户设置不同的安全性并测试。（自己回答）

第 13 天 高级 SQL

问题答案

8、MICIRSOFT VISUAL C++可以让程序员直接调用 ODBC 的 API 函数，对不对？

（不对，它将 ODBC 库的压缩内容做为自己的组件，组件提供了对 ODBC 的高级接口，所以可以更方便地使用这个功能。但是其功能的完全发挥受到了限制，如果你购买了 ODBC 的软件开发环境（SDK），那么你可以直接从 VISUAL C++应用程序中直接调用 API）

9、ODBC 的 API 函数只能由 C 语言直接调用，对不对？

（不对，ODBC 的 API 是以 DLL 形式驻留的，它可以为许多语言调用，如 VISUAL BASIC 和 BORLAND OBJECT PASCAL）

10、动态 SQL 需要进行预编译，对不对？

（不对，静态的需要预编译，动态的 SQL 由于是动态的，它只有在使用时才编译处理）

11、临时表中的#提示符是干什么用的？

（SQL SERVER 使用#来标识临时表）

12、在将游标从内存中关闭后必须做什么？

（你必须释放游标，语法为：SQL> deallocate cursor cursor_name;）

13、能不能是 SELECT 语句中使用触发机制？

（不用，它会在使用 UPDATE、DELETE 或 INSERT 时自动运行）

14、如果你在表中创建了触发机制然后你把表删除了，那么触发机制还存在吗？

（不会，表被删除后触发机制就自动删除了）

练习答案

10、创建一个示例数据库应用程序（在今天我们使用了音乐收藏数据库作为示例）并对应用程序进行合理的数据分组。

11、列出你想要在数据库中完成的查询。

12、列出你要在维护数据库中需要的各种规则。

13、为你在第一步创建的数据库逻辑给创建不同的数据库计划。

- 14、 将第二步中的查询转变为存贮过程。
- 15、 将第三步中的规则转变为触发机制。
- 16、 将第 4 步与第 5 步结合起来，与第 6 步一起生成一个脚本，其中包括所有的与该数据库相关联的过程。
- 17、 插入一些示例数据（这一步可以作为第 7 步生成脚本的一部分）
执行你所创建的这些过程并验证它的功能。
（自己回答）

第 14 天：动态使用 SQL

问题答案

- 5、在 VISUAL C++ 中
（是 CRecordSet 对象中的 GetDefaultSQL 成员，切记，你可以在操作你的表时改变这里的字符串）
- 6、在 DELPHI 中哪一个对象用来存放 SQL。
（TQUERY 对象）
- 7、什么是 ODBC。
（ODBC 基于开放数据库联连接，该技术可以让基于 WINDOWS 的应用程序通过驱动来访问数据库）
- 8、DELPHI 可以做什么？
（DELPHI 对于不同的数据库提供了一致的界面）

练习答案

- 3、在 C++ 的例子中如何对 STATE 字段进行正序或逆序的排序操作。
（CString CTyssqlSet::GetDefaultSQL()
 {
 return " SELECT * FROM CUSTOMER ORDER DESC BY STATE ";
 }
）
- 4、在向前一步，找到一个需要使用 SQL 的程序并使用它。（自己做）

第 15 天：对 SQL 语句优化以提高其性能

问题答案

7、SQL 语句的流化是什么意思？

（流化的意思就是认真地去安排在你的 SQL 语句中各子句中元素位置使其效率最大）

8、表和它的索引是否应该放在同一个磁盘上？

（绝对不可以，如果可能，应该把它们放在不同的磁盘上以减少磁盘的交叉访问）

9、为什么说对 SQL 语句中各个元素的安排是非常重要的？

（这样可以让数据的访问更有效，目标是减少反应时间）

10、当全表扫描时会发生什么情况？

（不是先读索引然后读数据，而是逐行逐行地读数据）

11、你如何才能避免全表扫描？

（通过创建索引和合理地安排被索引过的 SQL 语句中的元素位置）

12、常见的对性能的障碍有哪些？

（常见的性能瓶颈有：

共享内存不足

磁盘驱动器不足

磁盘的可用空间安排不当

无计划地运行大型的批量载入

懒于使用 COMMIT 和 ROLLBACK 命令

表和索引的大小不合适）

练习答案

3、让下边的 SQL 语句更易读。

```
SELECT      EMPLOYEE.LAST_NAME,      EMPLOYEE.FIRST_NAME,
EMPLOYEE.MIDDLE_NAME,EMPLOYEE.ADDRESS,      EMPLOYEE.PHONE_NUMBER,
PAYROLL.SALARY,   PAYROLL.POSITION,EMPLOYEE.SSN,   PAYROLL.START_DATE
FROM  EMPLOYEE,  PAYROLL  WHEREEMPLOYEE.SSN  =  PAYROLL.SSN  AND
```

```
EMPLOYEE.LAST_NAME LIKE 'S%' AND PAYROLL.SALARY > 20000;
```

(你应该重新格式这个 SQL 语句，根据你想要做的工作而定：

```
SELECT E.LAST_NAME, E.FIRST_NAME, E.MIDDLE_NAME,
       E.ADDRESS, E.PHONE_NUMBER, P.SALARY,
       P.POSITION, E.SSN, P.START_DATE
FROM EMPLOYEE E,
       PAYROLL P
WHERE E.SSN = P.SSN
      AND E.LAST_NAME LIKE 'S%'
      AND P.SALARY > 20000;
)
```

- 4、重新安排下边的查询条件以减少数据返回所需要的时间，并使用下边的统计（对整个表）以决定这些条件的次序。

593 individuals have the last name SMITH.

712 individuals live in INDIANAPOLIS.

3,492 individuals are MALE.

1,233 individuals earn a salary >= 30,000.

5,009 individuals are single.

Individual_id is the primary key for both tables.

```
SELECT M.INDIVIDUAL_NAME, M.ADDRESS, M.CITY, M.STATE, M.ZIP_CODE,
       S.SEX, S.MARITAL_STATUS, S.SALARY
FROM MAILING_TBL M, INDIVIDUAL_STAT_TBL S
WHERE M.NAME LIKE 'SMITH%'
      AND M.CITY = 'INDIANAPOLIS'
      AND S.SEX = 'MALE'
      AND S.SALARY >= 30000
      AND S.MARITAL_STATUS = 'S'
      AND M.INDIVIDUAL_ID = S.INDIVIDUAL_ID;
```

答：根据统计信息，你的新查询应该与下边的答案相像，NAME 与 'SMITH%' 是最严格的条件，因为它返回的行数最少：

```
SELECT M.INDIVIDUAL_NAME, M.ADDRESS, M.CITY, M.STATE, M.ZIP_CODE,
       S.SEX, S.MARITAL_STATUS, S.SALARY
FROM MAILING_TBL M,
     INDIVIDUAL_STAT_TBL S
WHERE M.INDIVIDUAL_ID = S.INDIVIDUAL_ID
      AND S.MARITAL_STATUS = 'S'
      AND S.SEX = 'MALE'
      AND S.SALARY >= 30000
      AND M.CITY = 'INDIANAPOLIS'
      AND M.NAME LIKE 'SMITH%';
```

第 16 天：用视图从数据字典中获得信息

问题答案

5、 在 ORACLE 中，你如何才能知道哪些表和视图是为你所有的？

（用 SELECT FROM USER_CATALOG 或 CAT，在数据字典中对象的名字根据解释器的不同而不同，但是所有的版本在关于像表和视图这的的对象的基本信息是相同的）

6、 在数据字典中存储有哪些信息？

（数据库的设计，用户的统计，过程，对象，对象的增长情况，性能统计，存储 SQL 代码以及数据库安全信息）

7、 你如何才能进行性能统计？

（对性能的统计给出的建议的方法通过修改数据库的参数和流化 SQL 来提高数据库的性能，也可能是用索引的方法来使结果更有效）

8、 数据库对象都有哪些？

（表、索引、同义字、簇、视图）

练习答案

假设你管理了一个中小型的数据库系统，你的职责是开发和管理数据库，某人向表中

插入了大量的数据并收到了一个空间不足的错误信息。你必须断定问题产生的原因。是对该用户配额的表空间增加还是你需要增加为表空间分配的磁盘空间。要一步一步地列出你需要从数据字典中得到的信息——不必给出具体的表和视图的名称。

- (1、查找在你的数据库文档中的错误
- (2、在数据字典中查询关于表的信息，它的当前大小，用户的表空间配额，表空间的分配。
- (3、确定用户要完成插入操作需要多少空间。
- (4、哪一个真正的问题，是用户的表空间配额需要增加还是需要分配更多的表空间
- (5、如果用户的配额不足，那么增加配额，如果是当前的表空间满了，你需要为当前表分配更多的表空间)
- (6、你也可能既不增加表空间配额也不增加表空间，这时你不得不删除一些旧的数据或将一些数据归档到磁带上。

这些步骤不是必需的，你的工作要根据你公司的政策和你个人的素质来决定。

第 17 天：使用 SQL 来生成 SQL 语句

问题答案

- 10、你生成 SQL 的来源有哪两个？

(你可以从数据库的表和数据字典中生成脚本)

- 11、下边的 SQL 语句是否可以工作？它会输出什么？

```
SQL> SET ECHO OFF
```

```
SQL> SET FEEDBACK OFF
```

```
SQL> SPOOL CNT.SQL
```

```
SQL> SELECT 'COUNT(*) FROM ' || TABLE_NAME || ';
```

```
2 FROM CAT
```

```
3 /
```

(该语句将会生成一个脚本，但是生成的脚本不会工作，你需要在 count(*)之前使用 SELECT 'SELECT'

```
SELECT 'SELECT COUNT(*) FROM ' || TABLE_NAME || ';
```

否则你的输出将会如下：


```
COUNT(*) FROM TABLE_NAME;
```

这不是有效的 SQL 语句)

- 12、 下边的 SQL 语句是否可以工作? 它会输出什么?

```
SQL> SET ECHO OFF
```

```
SQL> SET FEEDBACK OFF
```

```
SQL> SPOOL GRANT.SQL
```

```
SQL> SELECT 'GRANT CONNECT DBA TO ' || USERNAME || ';
```

```
2 FROM SYS.DBA_USERS
```

```
3 WHERE USERNAME NOT IN ('SYS','SYSTEM','SCOTT')
```

```
4 /
```

(回答仍是(是也不是), 该语句将会生成 SQL 脚本, 但是由于生成的 SQL 语句不完整, 你需要在 CONNECT 和 DBA 之间加入逗号

```
SELECT 'GRANT CONNECT, DBA TO ' || USERNAME || ';
```

```
)
```

- 13、 下边的 SQL 语句是否可以工作? 它会输出什么?

```
SQL> SET ECHO OFF
```

```
SQL> SET FEEDBACK OFF
```

```
SQL> SELECT 'GRANT CONNECT, DBA TO ' || USERNAME || ';
```

```
2 FROM SYS.DBA_USERS
```

```
3 WHERE USERNAME NOT IN ('SYS','SYSTEM','SCOTT')
```

```
5 /
```

(该语句是正确的, 生成的 SQL 语句将会给所有的用户以 CONNECT 和 DBA 权限)

- 14、 在运行生成的 SQL 时最为将 FEEDBACK 设置为 ON, 对不对?

(不对, 如果你不关心有多少行被选出时可以把它关闭, 它不是你要生成的 SQL 语句的一部分)

- 15、 从 SQL 中生成 SQL 语句时, 必须将输入的结果重新定向到一个列表或 LOG 文件中, 对不对?

(不对, 你应该重新定向到一个 SQL 文件中或任何你所命名的文件中)

- 16、 在生成 SQL 语句用以对表的内容进行删减时, 你必须先确认自己已经对所删减

的表作了很好的备份。对不对？（对，这样最安全）

17、 什么是 ED 命令？

（ED 命令是一个全屏幕的编辑器，所 UNIX 的 VI 或 WINDOWS 下的记事本类似）

18、 SPOOL OFF 命令是做什么的？

（关闭一个假脱机文件）

练习答案

3、 使用 SYS.DBA_USERS 视图（在 PERSONAL ORACLE 7 中）写一个语句来生成一系列 GRANT 语句为下边的五个用户：John、 Kevin、 Ryan、 Ron 和 Chris 授权，使他们可以访问 History_tbl 表，在写语句时对应的用户名用 USERNAME 来代替。

```
SQL> SET ECHO OFF
```

```
SQL> SET FEEDBACK OFF
```

```
SQL> SPOOL GRANTS.SQL
```

```
SQL> SELECT 'GRANT SELECT ON HISTORY_TBL TO ' || USERNAME || ';
```

```
2      FROM SYS.DBA_USERS
```

```
3      WHERE USERNAME IN ('JOHN','KEVIN','RYAN','RON','CHRIS')
```

```
4      /
```

```
grant select on history_tbl to JOHN;
```

```
grant select on history_tbl to KEVIN;
```

```
grant select on history_tbl to RYAN;
```

```
grant select on history_tbl to RON;
```

```
grant select on history_tbl to CHRIS;
```

4、 用本章给出的例子作指引，写一些 SQL 来创建一些可以生成你能使用的 SQL 语句。

（自己做）

警告：除非你已经彻底地明白了本章的概念，否则一定要注意生成 SQL 语句可能会修改数据或数据库的结构

第 18 天：PL/SQL 简介

问题答案

8、如何在数据库中使用触发机制？

（数据库的触发机制可以在指定表中的数据发生变化时进行执行一些特定的操作。例如：如果你改变了一个表，那么触发机制可以将被改变的数据存入历史记录表中以备后查）

9、是否可以将相关的过程存储在一起？

（相关的过程可以存储在包中）

10、可以在 PL/SQL 中使用数据操作语言，对不对？（对）

11、可以在 PL/SQL 中使用数据定义语言，对不对？

（不对，DDL 不能在 PL/SQL 中使用，用自动的方法来改变数据库的结构不是好主意）

12、在 PL/SQL 的语法中是否支持直接的文本输出？

（它不被 PL/SQL 所直接支持，但是，它可以通过标准的包 DBMS_OUTPUT 来输出）

13、给出 PL/SQL 语句块的三个主要部分？

（DECLARE 部分、PROCEDURE 部分、EXCEPTION 部分）

14、请给出与游标控制相关的命令？

（DECLARE, OPEN, FETCH, CLOSE）

练习答案

5、请定义一个变量，使它可以接受的最大数值为 99.99。

```
(DECLARE
    HourlyPay number(4,2);
)
```

6、请定义一个指针，它的内容包括 CUSTOMER_TABLE 表中的所有 CITY 为 INDIANAPOLIS 的客户。

```
DECLARE
    cursor c1 is
    select * from customer_table
    where city = 'INDIANAPOLIS';
```

7、定义一个名字为 UnknownCode 的异常。

```
DECLARE
    UnknownCode EXCEPTION;
```

8、请写一个语句，使得在 AMOUNT_TABLE 中的 AMT 当 CODE 为 A 时其值为 10，当 CODE 为 B 时其值为 20，当 CODE 既不是 A 也不是 B 时激活一个名字叫 UnknownCode 的异常，表中的内容只有一行。

```
( IF ( CODE = 'A' ) THEN
    update AMOUNT_TABLE
    set AMT = 10;
ELSIF ( CODE = 'B' ) THEN
    update AMOUNT_TABLE
    set AMT = 20;
ELSE
    raise UnknownCode;
END IF;
)
```

第 19 天：TRANSACT-SQL 简介

问题答案

问：在 ORACLE 的 PL/SQL 中与在 TRANSACT-SQL 中 SQL 字的使用方法与 ANSI 标准的 SQL 是完全相同的，对不对？

答：不对，这些关键字不受版权的保护，产品在许多内容上都与 ANSI 标准一致，但不是每样东西都一致。

问：静态的 SQL 比动态的 SQL 灵活性差，尽管它的性能要比动态的好，对不对？

答：对，静态的 SQL 需要进行预编译，而不是动态编译，所以它的灵活性不如动态 SQL。但是由于查询已经被处理过，所以它的性能要更好些。

练习答案

3、如果你没有使用 SYBASE 或 MICROSOFT 的 SQL SERVER，那么请你比较一个你的产品对 SQL 的扩展与今天所讲的有何不同？

（由于今天所讲的几乎全是 TRANSACT-SQL，我们并没有对 ANSI 标准的其它扩展进行

探讨，大多数的数据库随机文档都标明了它所提供了对 SQL 的有效的扩展，这些扩展在其它数据库中使用是不同的)

- 4、写一组 SQL 语句，它可以对一些已知的条件进行检测，如果条件为真，执行一些操作，
否则的话执行另一些操作

(该操作需要使用 IF 语句，具有的逻辑结构今天已经讨论过了)

第 20 天：SQL*PLUS

问题答案

- 7、哪些命令可以改变你的 SQL 会话的性能?

(SET 命令可以改变你的会话设置)

- 8、你可以在 SQL 的脚本中提示用户输入参数并根据输入的参数运行吗?

(可以，你的脚本能从用户输入接受参数并把它赋给变量)

- 9、如果你对 CUSTOMERS 表创建了一个汇总报表，你如何在你的报表中对你的数据进行
分组?

(你可能会依据客户进行分组，因为你的数据是从客户表中选择的)

- 10、 你在使用 LOGIN.SQL 文件时有哪些限制?

(它的限制就是你输入到其中的内容必须是 SQL 或 SQL*PLUS 的有效命令)

- 11、 DECODE 函数与过程语言中的 LOOP 功能是等价的。对不对?

(不对，它与 IF.....THEN 类似)

- 12、 如果你将查询重新定向到一个已经存在的文件中，你的输出将追加到这个文件，对不
对?

(不对，新文件将会覆盖原文件的内容)

练习答案

- 4、利用在今天开始时的 PRODUCTS 表，写一个查询选择其中的所有数据并对记录的个数
进行汇总，要生成报表并且不得使用 SET FEEDBACK ON 命令。

```
compute sum of count(*) on report  
break on report
```

```
select product_id, product_name, unit_cost, count(*)
from products
group by product_id, product_name, unit_cost;
```

5、假如今天是 1998 年 5 月 12 日星期一，写一个查询产生下边的输出：

Today is Monday, May 12 1998

答：

```
set heading off
select to_char(sysdate,' "Today is "Day, Month dd yyyy')
from dual;
```

6、试一下下边的语句：

```
1 select *
2 from orders
3 where customer_id = '001'
4* order by customer_id;
```

不需要在缓冲区中重新输入这些语句，将 FROM 子句中的表改为 CUSTOMERS。

在 ORDER BY 子句中加入 DESC。

```
l2
c/orders/customer
现在你 DESC 加入到 ORDER BY 子句中
l4
append DESC
```

第 21 天：常见的 SQL 错误及解决方法

问题答案

1. A user calls and says, "I can't sign on to the database. But everything was working fine yesterday. The error says invalid user/password. Can you help me?" What steps should you take? At first you would think to yourself, yeah sure, you just forgot your password. But this error can be returned if a front-end application cannot connect to the database. However, if you know the database is up and functional, just change the password by using the ALTER USER command and tell the user what the new password is.

2. Why should tables have storage clauses and a tablespace destination?

In order for tables not to take the default settings for storage, you must include the storage clause. Otherwise medium to large tables will fill up and take extents, causing slower performance. They also may run out of space, causing a halt to your work until the DBA can fix the space problem.

1. Suppose you are logged on to the database as SYSTEM, and you wish to drop a table called HISTORY in your schema. Your regular user ID is JSMITH. What is the correct syntax to drop this table?

Because you are signed on as SYSTEM, be sure to qualify the table by including the table owner. If you do not specify the table owner, you could accidentally drop a table called HISTORY in the SYSTEM schema, if it exists.

```
SQL> DROP TABLE JSMITH.HISTORY;
```

2. Correct the following error:

INPUT:

```
SQL> select sysdate DATE
      2      from dual;
```

OUTPUT:

```
select sysdate DATE
*

```

ERROR at line 1:

ORA-00923: FROM keyword not found where expected

DATE is a reserved word in Oracle SQL. If you want to name a column heading DATE, then you must use double quotation marks: "DATE".

3. 一个用户打电话说：“我不能登录数据库了，昨天还能呢。你能帮帮我吗？你该如何做？

首先，你要问他是否忘记了密码。但是这个错误也可能是因为前端无法联接到数据库产生的。如果你可以确认数据正常，那么你可以更改用户属性，并告诉他新的密码。

4. 为什么表在存储子句中有表空间项。

这是为了可以让表不使用默认的存储空间选项，否则如果表很大时，系统的性能就会变慢。也可能导致运行超时，系统挂死直到 DBA 修正了这个错误。

练习答案

3. 如果你以 SYSTEM 身份登录了数据库，你想删除你的计划中的一个名字叫 HISTORY 的表，如果你的用户 ID 是 JSMITH，那么正确的语法是怎样的？

因为你是以 SYSTEM 身份登录的，所以要确认表的所有者对表的限制，如果你不是指定的表的所有者，你也可以在 SYSTEM 计划中删除了这个表。

4. 更正下边语句的错误

INPUT:

```
SQL> select sysdate DATE
```

```
2 from dual;
```

OUTPUT:

```
select sysdate DATE
```

```
*
```

ERROR at line 1:

ORA-00923: FROM keyword not found where expected

DATE 是 ORACLE SQL 的保留字。如果你想把列命名为 DATE 你应该在它的两边加上单引号。