Oracle基础篇

目录

[第一章 Oracle数据库 5](#_Toc28097623)

[1.1 Oracle数据库简介 5](#_Toc28097624)

[1.2 Oracle的数据类型 5](#_Toc28097625)

[1.3 Oracle客户端 5](#_Toc28097626)

[1.3.1 Oracle Enterprise Manager 5](#_Toc28097627)

[1.3.2 SQL\*Plus 和 iSQL\*Plus 6](#_Toc28097628)

[第二章 oracle体系结构 8](#_Toc28097629)

[2.1 oracle存储结构 8](#_Toc28097630)

[2.1.1 物理存储结构 9](#_Toc28097631)

[2.1.2 oracle逻辑结构 10](#_Toc28097632)

[2.2 oracle的内存结构 14](#_Toc28097633)

[2.2.1  系统全局区(SGA) 14](#_Toc28097634)

[2.2.2 程序全局区(PGA) 16](#_Toc28097635)

[2.2.3 用户全局区(UGA) 16](#_Toc28097636)

[2.3  oracle进程结构 16](#_Toc28097637)

[2.3.1 用户进程与服务器进程 16](#_Toc28097638)

[2.3.2 后台进程 17](#_Toc28097639)

[2.3.3 oracle for windows 服务器进程 20](#_Toc28097640)

[2.4 sql的执行过程 20](#_Toc28097641)

[2.5 oracle数据的启动与关闭 23](#_Toc28097642)

[2.5.1 数据库启动 23](#_Toc28097643)

[2.5.2  orace数据库的关闭 24](#_Toc28097644)

[2.5.3   数据库的静默和挂起状态 25](#_Toc28097645)

[2.6 oracle的初始化参数 26](#_Toc28097646)

[第二章 Oracle数据库管理 27](#_Toc28097647)

[2.1 用户管理 27](#_Toc28097648)

[2.1.1 创建一个新用户 27](#_Toc28097649)

[2.1.2 删除或锁定用户 27](#_Toc28097650)

[2.1.3 更改用户密码 27](#_Toc28097651)

[2.2 表空间管理 27](#_Toc28097652)

[2.2.1 创建表空间和临时表空间 28](#_Toc28097653)

[2.2.2 查看表空间 28](#_Toc28097654)

[2.2.3 修改表空间 28](#_Toc28097655)

[2.2.4 删除表空间 29](#_Toc28097656)

[2.3 角色管理 29](#_Toc28097657)

[2.3.1 Oracle的默认角色 29](#_Toc28097658)

[2.3.2 角色的使用 29](#_Toc28097659)

[2.4 权限管理 29](#_Toc28097660)

[2.4.1 系统权限 29](#_Toc28097661)

[2.4.2 对象权限 30](#_Toc28097662)

[2.5 事务管理 30](#_Toc28097663)

[2.5.1 事务控制命令 30](#_Toc28097664)

[2.5.2 事务的隔离级别 31](#_Toc28097665)

[2.6 回收站管理 31](#_Toc28097666)

[2.6.1 开启和关闭回收站 31](#_Toc28097667)

[2.6.2 查看回收站对象 32](#_Toc28097668)

[2.6.3 还原回收站对象 33](#_Toc28097669)

[2.6.4 清空回收站 34](#_Toc28097670)

[2.7 数据库备份与恢复 35](#_Toc28097671)

[2.7.1 逻辑备份-EXP/IMP 35](#_Toc28097672)

[2.7.2 逻辑备份-EXPDP/IMPDP 36](#_Toc28097673)

[2.7.3 物理备份-热备份 39](#_Toc28097674)

[2.7.4 物理备份-冷备份 41](#_Toc28097675)

[2.7.5 其它数据备份恢复方法 41](#_Toc28097676)

[第三章 Oracle数据库对象 42](#_Toc28097677)

[3.1 基本表 42](#_Toc28097678)

[3.1.1 创建表 42](#_Toc28097679)

[3.1.2 删除表 42](#_Toc28097680)

[3.1.3 修改表 42](#_Toc28097681)

[3.2 表的约束 42](#_Toc28097682)

[3.2.1 主键(primary key)约束 42](#_Toc28097683)

[3.2.2 唯一性(unique)约束 43](#_Toc28097684)

[3.2.3 非空(NOT NULL)约束 43](#_Toc28097685)

[3.2.4 默认值(default)约束 43](#_Toc28097686)

[3.2.5 检查(check)约束 44](#_Toc28097687)

[3.2.6 外键(FOREIGN KEY)约束 44](#_Toc28097688)

[3.2.7 删除或禁用约束 44](#_Toc28097689)

[3.2.8 查看约束信息 45](#_Toc28097690)

[3.3 表的视图 45](#_Toc28097691)

[3.3.1 什么是视图？ 45](#_Toc28097692)

[3.3.2 视图的作用 45](#_Toc28097693)

[3.3.3 创建视图 46](#_Toc28097694)

[3.3.4 简单视图 46](#_Toc28097695)

[3.3.5 连接视图 47](#_Toc28097696)

[3.3.6 复杂视图 48](#_Toc28097697)

[3.3.7 更改视图 48](#_Toc28097698)

[3.3.8 删除视图 49](#_Toc28097699)

[3.3.9 查看视图 50](#_Toc28097700)

[附： 在视图上执行DML操作的步骤和原理 50](#_Toc28097701)

[3.4 表的索引 51](#_Toc28097702)

[3.4 Oracle 索引简介 51](#_Toc28097703)

[3.4.1 创建索引 51](#_Toc28097704)

[3.4.2 修改索引 52](#_Toc28097705)

[3.4.3 删除索引 52](#_Toc28097706)

[3.4.4 显示索引信息 52](#_Toc28097707)

[3.5 Oracle序列 53](#_Toc28097708)

[3.5.1 序列的创建 53](#_Toc28097709)

[3.5.2 使用序列 53](#_Toc28097710)

[3.5.3 修改序列 54](#_Toc28097711)

[3.5.5 查询序列 54](#_Toc28097712)

[3.6 Oracle同义词 54](#_Toc28097713)

[3.6.1 创建同义词 55](#_Toc28097714)

[3.6.2 查看同义词 55](#_Toc28097715)

[3.6.3 使用同义词 56](#_Toc28097716)

[3.6.5 编译同义词 56](#_Toc28097717)

[3.8 Oracle数据字典 56](#_Toc28097718)

[3.8.1 数据字典表 56](#_Toc28097719)

[3.8.2 静态数据字典视图 57](#_Toc28097720)

[3.8.3 内部RDBMS（X$）表 57](#_Toc28097721)

[3.8.4 动态性能视图 58](#_Toc28097722)

[第四章 数据的增删改查 61](#_Toc28097723)

[4.1 插入数据 61](#_Toc28097724)

[4.2 删除数据 61](#_Toc28097725)

[4.3 修改数据 61](#_Toc28097726)

[4.4 查询数据 62](#_Toc28097727)

[4.4.1 基本查询 62](#_Toc28097728)

[4.4.2 关联查询 62](#_Toc28097729)

[4.4.3 子查询 63](#_Toc28097730)

[4.4.4 分组查询与聚合函数 66](#_Toc28097731)

[4.4.5 查询结果过滤 66](#_Toc28097732)

[4.4.6 查询结果排序 67](#_Toc28097733)

[4.5 集合运算 68](#_Toc28097734)

[第五章 内置函数与表达式 70](#_Toc28097735)

[5.1 字符类函数 70](#_Toc28097736)

[5.2 数值类函数 71](#_Toc28097737)

[1.3 日期类函数 71](#_Toc28097738)

[5.4 类型转换函数 72](#_Toc28097739)

[5.5 通用函数 73](#_Toc28097740)

[5.6 表达式 73](#_Toc28097741)

[第七章 PLSQL 77](#_Toc28097742)

[第八章 存储过程、存储函数、包 78](#_Toc28097743)

# 第一章 Oracle数据库

## 1.1 Oracle数据库简介

Oracle Database，简称Oracle。是甲骨文公司的一款对象关系型数据库管理系统(RDBMS)，主要应用在传统行业的数据化业务中，比如：银行、金融这样的对可用性、健壮性、安全性，实时性要求极高的业务场景。

安装步骤详情见安装文档!

卸载步骤详情见www.51zxw.net视频

用于卸载的批处理文件：C:\app\hyx\product\11.2.0\dbhome\_1\deinstall\deinstall.bat

## 1.2 Oracle的数据类型

### 1.2.1 字符型

| **数据类型** | **说明** |
| --- | --- |
| varchar2 | 变长字符串，n为最大长度，取值范围1~4000  与varchar(n)的区别：  1) varchar中数字、英文字符占一个字节，而varchar2是两个字节  2) VARCHAR2把空串等同于null处理，而varchar仍按照空串处理  3) VARCHAR2字符要用几个字节存储，要看数据库使用的字符集 |
| nvarchar2 | 用来存储Unicode字符集的变长字符型数据 |
| char | 用于描述定长的字符型数据, n取值范围为1~2000,缺省为1 |
| nchar | 用来存储Unicode字符集的定长字符型数据 |
| long | 可变长字符列，最大长度2GB，但不能进行字符串搜索 |

### 1.2.2 数值型

| **数据类型** | **说明** |
| --- | --- |
| number(p,s) | 可变长的数值列，允许0、正值和负值，m是有效数字位数，n是小数位数，  取值范围分别为1~38和-84~127。 |
| float | 存储的精度是按二进制计算的，精度范围为二进制的1~126，在转化为十进制时需要乘以0.30103 |

### 1.2.3 日期型

| **数据类型** | **说明** |
| --- | --- |
| date | 在内部是按7个字节来保存日期数据，在定义中还包括小时、分、秒，缺省格式为DD-MON-YY，如07-11月-00 表示2000年11月7日。 |
| timestamp | 用来存储日期和时间，时间可以精确到毫秒。此外，使用timestamp存放日期和时间还能够显示当前是上午还是下午 |

### 1.2.3 其它数据类型

| **数据类型** | **说明** |
| --- | --- |
| blob | 存储二进制数据,最大4GB |
| clob | 存储字符串数据，最大4GB |
| bfile | 用来把非结构化的二进制数据存储在数据库以外的操作系统文件中，大小与操作系统有关 |

## 1.3 Oracle客户端

### 1.3.1 Enterprise Manager

安装 Oracle 数据库时，Oracle Universal Installer 还会安装 Oracle Enterprise Manager (Enterprise Manager)。Enterprise Manager 中基于 Web 的 Database Control，可充当管理 Oracle 数据库的主要工具。

1. 启动EM

    $ emctl start dbconsole

2. 关闭EM

    $ emctl stop dbconsole

3. 检查EM状态

    oracle>emctl status dbconsole

4. 重建EM

如果启动EM报错，错误信息如下：OC4J Configuration issue. /oracle/product/11.2.0/db\_1/oc4j/j2ee/OC4J\_DBConsole\_testdb\_kgdb not found.，则需要重建

    $ emca -config dbcontrol db -repos recreate

5.进入EM

    EM启动后，可在浏览器中输入http://host name:port number/em 访问, 如 https://localhost:1158/em

*-- 如果出现“应用程序要求的数据库权限超出了您当前具有的权限”，则需要给用户赋权：****grant select\_catalog\_role  to SCOTT****;*

*-- 在windwos平台上，还需启动xxx服务,如果启动 Enterprise Manager 时出现了问题，请确保启动了监听程序。*

### 1.3.2 SQL\*Plus 和 iSQL\*Plus

除了 Oracle Enterprise Manager 外，还可以使用其它 Oracle 工具（如 SQL\*Plus 和 iSQL\*Plus）发出 SQL 语句。使用这些工具可执行很多的数据库管理操作，还可在数据库中选择、插入、更新或删除数据。

要启动 SQL\*Plus，请执行以下步骤：

1) 打开终端窗口。

2) 在命令行提示符下，按以下格式输入 SQL\*Plus 命令：

            $ sqlplus /nolog

3) 输入 connect，之后输入要以其身份进行连接的用户名。

4) 出现提示后，输入该用户的口令。SQL\*Plus 启动后会连接到默认数据库。

用户登录的语法如下：

    {<username>[/<password>][@<connect\_identifier>] /[AS{SYSDBA | SYSOPER}]

*-- connect\_identifier：数据库连接标识符(服务器名)，不指定则连接到默认数据库。*

*-- SYSDBA ISYSOPER：分别为数据库管理员的权限和数据库操作员的权限。*

例1：

    conn sys/manager @orcl as sysdba (使用@必须启动监听服务)

连接用户：connect hyx/hyx123

退出用户：quit

用户登陆语法格式

注： 1) 可以用sqlplus /nolog来登陆sqlplus，但是不连接数据库。

     2) System用户以sysdba权限登陆后show user出来的是SYS。

     3) conn /as sysdba 可以不用指定用户名和密码。

1. sqlplus的命令模式

connect 切换连接用户，简写为conn，如conn SCOTT/SCOTT123。

show user 显示登陆用户。

host <dos> 在SQLPlus中执行dos命令，如host mkdir C:\testOracle。

spool <file> 导出记录到文件，如spool C:\sql.txt ，<执行SQL> ，spool off

clear screen 清除屏幕，也可用host cls

start <.sql > 执行SQL脚本，与@效果一致。

desc 显示表结构

show err 显示错误信息

quit 退出SQLPlus，也可以用exit。

set linesize xx 设置行尺寸

set pagesize xx 设置页尺寸

set timing on/off 打开与关闭命令记时

set feedback on/off 打开与关闭回显

ed 编辑上一条命令(语句末尾不加分号，编辑完成后用”/”运行)

col DEPTNO for 99999 修改数字的列宽

col ENAME for a20 修改字符的列宽

*--linux下的sqlplus如何使用edit编辑命令*

        1 在sqlplus中 键入 ed

        2 在另外一个终端中，打开afiedt.buf文件 进行修改 保存 退出

        3 回到sqlplus环境中，键入q

        4 在sqlplus中 /

2. 从shell调用sqlplus

  如下是一个shell脚本，通过调用 sqlplus 和使用传递参数的操作系统脚本语法，可从 shell 脚本或 BAT 文件中调用 SQL\*Plus。

在这个示例中，先执行 SELECT、UPATE 和 **COMMIT** 语句，之后 SQL\*Plus 将控制权返回给操作系统。

# Name of this file: batch\_sqlplus.sh

# **Count** employees and give raise.

sqlplus hr/hr <<EOF

select **count**(\*) from employees;

update employees **set** salary = salary\*1.10;

**commit**;

quit

EOF

exit

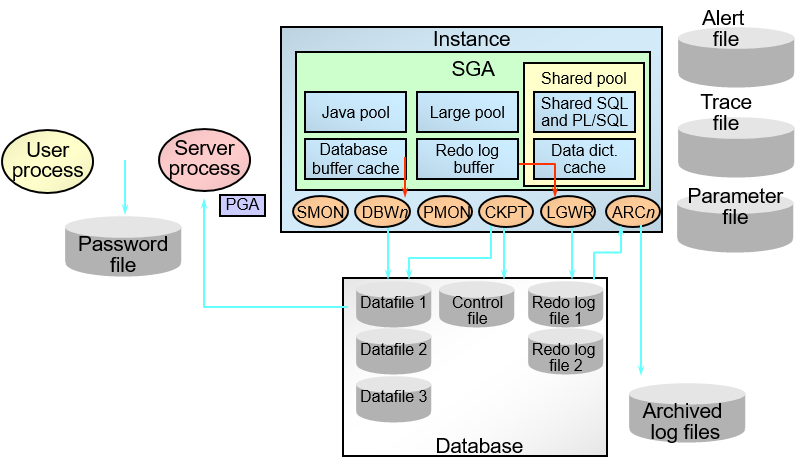
3. 从 SQL\*Plus 调用 SQL 脚本

可以从 SQL\*Plus 中调用现有的 SQL 脚本文件。可在首次调用 SQL\*Plus 时在命令行上完成此操作，如幻灯片所示。只使用“@”运算符也可以从 SQL\*Plus 会话中完成此操作。例如，输入以下内容可从已建立的 SQL\*Plus 会话中运行脚本：

SQL> @script.sql

# 第二章 oracle体系结构

**Oracle的体系结构**是数据库的组成，工作过程，以及数据库中数据的组织与管理机制。一个**oracle服务器**由一个oracle数据库+一个oracle实例组成，为用户提供开放、全面、集成的方法进行信息管理。



1. Oracle数据库

    是各种数据的集合，由操作系统文件组成，这些文件为数据库信息提供实际物理存储区。

2. Oracle实例

    有两个组成部分,分别是:

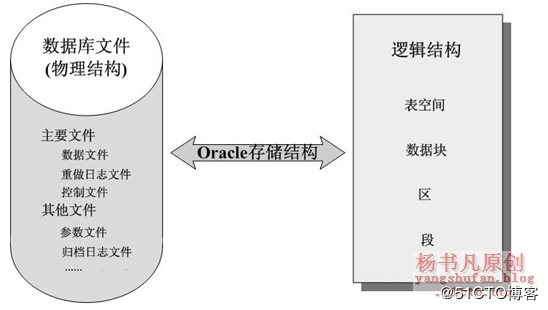
    1)  系统全局内存区(SGA):服务器专门划分给Oracle实例使用的内存块。

    2)  Oracle进程:包括服务器进程和后台进程。

必须启动实例才能访问数据库中的数据。一个实例只能打开并使用一个数据库，但一个数据库可以由多个实例打开。

## 2.1 oracle存储结构

Oracle的存储结构分为物理结构和逻辑结构，这两种存储结构既相互独立又相互联系。



### 2.1.1 物理存储结构

数据库的**物理存储结构**就是oracle数据库创建后使用的一系列操作系统物理文件，主要分为以下几类：

1. 数据文件(data files)

包含用户数据、程序数据、元数据和数据字典的文件,每个数据文件都有自己所属的表空间。

2. 控制文件(control files)

当数据库读取时，要根据控制文件的信息查找数据文件。控制文件记录了以下关键信息：数据文件的位置和大小、重做日志文件的位置及大小、数据库名称及创建时间、日志序列号。控制文件对数据库至关重要。为了安全起见，控制文件至少存在两份。

3. 重做日志文件

它记录了对数据的所有更改信息，并提供一种数据恢复机制，如果数据库服务器发生崩溃，但未丢失任何数据文件，那么实例便可使用这些文件中的信息恢复数据库。

4. 归档日志文件

使用这些文件和数据库备份，可以恢复丢失的数据文件。也就是说，归档日志能够恢复还原的数据文件

5. 口令文件

允许sysdba、sysoper 和sysasm 远程连接到实例并执行管理任务

6. 参数文件

用于定义实例启动时的配置

7. 跟踪文件

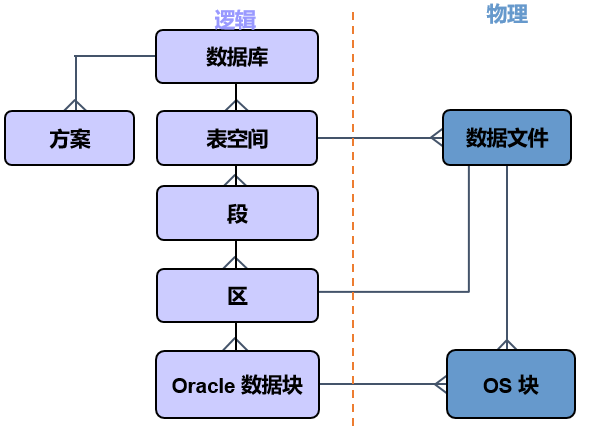
每一个服务器和后台进程都可以对关联的跟踪文件进行写操作。如果在进程中检测到内部错误，进程就会将关于错误的信息转储到进程的跟踪文件。写入跟踪文件的某些信息专门供数据库管理员使用，另一些信息则供 Oracle 支持服务部门使用。

8. 预警日志文件

    这些是特殊的跟踪文件，又称为预警日志。数据库预警日志是按时间顺序列出的消息日志和错误日志。Oracle 建议您查看这些文件。

### 2.1.2 oracle逻辑结构

    Oracle的逻辑结构是一种层次结构。主要由：表空间、段、区和数据块等概念组成。逻辑结构是面向用户的，用户使用Oracle开发应用程序使用的就是逻辑结构。数据库存储层次结构及其构成关系,结构对象也从数据块到表空间形成了不同层次的粒度关系。



1. 表空间（tablespace）

  是数据库的逻辑划分。任何数据库对象在存储时都必须存储在某个表空间中。表空间对应于若干个磁盘文件，即表空间是由一个或多个磁盘文件构成的。表空间相当于操作系统中的文件夹，也是数据库逻辑结构与物理文件之间的一个映射。每个数据库至少有一个表空间（system tablespace），表空间的大小等于所有从属于它的数据文件大小的总和。

1) SYSTEM表空间

    是每个Oracle数据库都必须具备的。其功能是在系统表空间中存放诸如表空间名称、表空间所含数据文件等数据库管理所需的信息。系统表空间的名称是不可更改的。系统表空间必须在任何时候都可以用，也是数据库运行的必要条件。因此，系统表空间是不能脱机的。

    SYSTEM表空间包括数据字典、存储过程、触发器和系统回滚段。为避免系统表空间产生存储碎片以及争用系统资源的问题，应创建一个独立的表空间用来单独存储用户数据。

2) SYSAUX表空间

    是随着数据库的创建而创建的，它充当SYSTEM的辅助表空间，主要存储除数据字典以外的其他对象。SYSAUX也是许多Oracle 数据库的默认表空间，它减少了由数据库和DBA管理的表空间数量，降低了SYSTEM表空间的负荷。

3) 临时表空间

    相对于其他表空间而言，临时表空间（temp tablespace）主要用于存储Oracle数据库运行期间所产生的临时数据。数据库可以建立多个临时表空间。当数据库关闭后，临时表空间中所有数据将全部被清除。除临时表空间外，其他表空间都属于永久性表空间。

4) 撤销表空间

    用于保存Oracle数据库撤销信息，即保存用户回滚段的表空间称之为回滚表空间（或简称为RBS撤销表空间（undo tablespace））。在Oracle8i中是rollback tablespace，从Oracle9i开始改为undo tablespace。在Oracle 10g中初始创建的只有6个表空间sysaux、system、temp、undotbs1、example和users。其中temp是临时表空间，undotbs1是undo撤销表空间。

5) USERS表空间

    用户表空间，用于存放永久性用户对象的数据和私有信息。每个数据块都应该有一个用户表空间，以便在创建用户是将其分配给用户。

2. 段（Segment）

是由多个数据区构成的，它是为特定的数据库对象（如表段、索引段、回滚段、临时段）分配的一系列数据区。段内包含的数据区可以不连续，并且可以跨越多个文件。使用段的目的是用来保存特定对象。 一个Oracle数据库有4种类型的段：

1) 数据段

    数据段也称为表段，它包含数据并且与表和簇相关。当创建一个表时，系统自动创建一个以该表的名字命名的数据段。

2) 索引段

    包含了用于提高系统性能的索引。一旦建立索引，系统自动创建一个以该索引的名字命名的索引段。

3) 回滚段

    包含了回滚信息，并在数据库恢复期间使用，以便为数据库提供读入一致性和回滚未提交的事务，即用来回滚事务的数据空间。当一个事务开始处理时，系统为之分配回滚段，回滚段可以动态创建和撤销。系统有个默认的回滚段，其管理方式既可以是自动的，也可以是手工的。

4) 临时段

    它是Oracle在运行过程中自行创建的段。当一个SQL语句需要临时工作区时，由Oracle建立临时段。一旦语句执行完毕，临时段的区间便退回给系统。

3. 数据区（Extent）

    是一组连续的数据块。当一个表、回滚段或临时段创建或需要附加空间时，系统总是为之分配一个新的数据区。一个数据区不能跨越多个文件，因为它包含连续的数据块。使用区的目的是用来保存特定数据类型的数据，也是表中数据增长的基本单位。在Oracle数据库中，分配空间就是以数据区为单位的。一个Oracle对象包含至少一个数据区。设置一个表或索引的存储参数包含设置它的数据区大小。

4. 数据块（Data Blocks）

    数据块是Oracle最小的存储单位，Oracle数据存放在“块”中。一个块占用一定的磁盘空间。特别注意的是，这里的“块”是Oracle的“数据块”，不是操作系统的“块”。

    Oracle每次请求数据的时候，都是以块为单位。也就是说，Oracle每次请求的数据是块的整数倍。如果Oracle请求的数据量不到一块，Oracle也会读取整个块。所以说，“块”是Oracle读写数据的最小单位或者最基本的单位。

    块的标准大小由初始化参数DB\_BLOCK\_SIZE指定。具有标准大小的块称为标准块(Standard Block)。块的大小和标准块的大小不同的块叫非标准块(Nonstandard Block)。

操作系统每次执行I/O的时候，是以操作系统的块为单位;Oracle每次执行I/O的时候，都是以Oracle的块为单位。Oracle数据块大小一般是操作系统块的整数倍。

1) 数据块的格式(Data Block Format)

块中存放表的数据和索引的数据等，无论存放哪种类型的数据，块的格式都是相同的，如下图五部分组成：

***块头(header)***：存放块的基本信息，如：块的物理地址，块所属的段的类型(是数据段还是索引段)。

***表目录(Table Directory)***：存放表的信息，即：如果一些表的数据被存放在这个块中，那么，这些表的相关信息将被存放在“表目录”中。

***行目录(Row Directory)***：如果块中有行数据存在，则，这些行的信息将被记录在行目录中。这些信息包括行的地址等。

***空余空间(Free Space)***：空余空间是一个块中未使用的区域，这片区域用于新行的插入和已经存在的行的更新。

***行数据(Row Data)***：是真正存放表数据和索引数据的地方。这部分空间是已被数据行占用的空间。

我们把块头、表目录和行目录这三部分合称为***头部信息区(Overhead)***。头部信息区不存放数据，它存放的整个块的信息。一般来说，头部信息区的大小介于84字节(bytes)到107字节(bytes)之间。

2) 数据块中自由空间的使用

当往数据库中插入(INSERT)数据的时候，块中的自由空间会减少;当对块中已经存在的行进行修改(UPDATE)的时候(使记录长度增加)，块中的自由空间也会减少。

DELETE语句和UPDATE语句会使块中的自由空间增加。当使用DELETE语句删除块中的记录或者使用UPDATE语句把列的值更改成一个更小值的时候，Oracle会释放出一部分自由空间。释放出的自由空间并不一定是连续的。

通常情况下，Oracle不会对块中不连续的自由空间进行合并。因为合并数据块中不连续的自由空间会影响数据库的性能。只有当用户进行数据插入(INSERT)或者更新(UPDATE)操作，却找不到连续的自由空间的时候，Oracle才会合并数据块中不连续的自由空间。

对于块中的自由空间，Oracle提供两种管理方式：自动管理和手动管理

3) 块中自由空间的自动管理

    Oracle使用位图(bitmap)来管理和跟踪数据块，这种块的空间管理方式叫“自动管理”，有下面的好处：

    ◆易于使用

    ◆更好地利用空间

    ◆可以对空间进行实时调整

4) 块中自由空间的手动管理

用户可以通过PCTFREE, PCTUSED来调整块中空间的使用，这种管理方式叫手动管理。相对于自动管理，手动管理方式比较麻烦，不容易掌握，容易造成块中空间的浪费。

    PCTUSED也是用于设置一个百分比，当块中已使用的空间的比例小于这个百分比的时候，这个块才被标识为有效状态。只有有效的块才被允许插入数据。

    PCTFREE参数用于指定块中必须保留的最小空闲空间百分例，默认值为10。之所以要预留这样的空间，是因为UPDATE时，需要这些空间。如果UPDATE时，没有空余空间，Oracle就会分配一个新的块，这会产生行迁移(Row Migrating)。

    例如，假定在Create table语句中指定了pctfree为20，则说明在该表的数据段内每个数据块的20%被作为可利用的空闲空间，用于更新已在数据块内存在的数据行其余80%是用于插入新的数据行，直到达到80%为止。显然，pctfree值越小，则为现存行更新所预留的空间越少。因此，如果pctfree设置得太高，则在全表扫描期间增加I/O，浪费磁盘空间；如果pctfree设置得太低，则会导致行迁移。

    pctused参数设置了数据块是否是空闲的界限。当数据块的使用空间低于pctused的值时，此数据块标志为空闲，该空闲空间仅用于插入新的行。如果数据块已经达到了由pctfree所确定的上边界时，Oracle就认为此数据块已经无法再插入新的行。例如，假定在Create table语句中指定pctused为40，则当小于或等于39时，该数据块才是可用的。所以，可将数据块填得更满，这样可节省空间，但却增加了处理开销，因为数据块的空闲空间总是要被更新的行占据，所以对数据块需要频繁地进行重新组织。比较低的pctused增加了数据库的空闲空间，但减少了更新操作的处理开销。所以，如果pctused设置过高，则会降低磁盘的利用率导致行迁移；若pctused设置过低，则浪费磁盘空间，增加全表扫描时的I/O输出。pctused是与pctfree相对的参数。

    那么，如何选择pctfree和pctused的值呢？

    有个公式可供参考。显然，pctfree和pctused的之和不能超过100。若两者之和低于100，则空间的利用与系统的I/O之间的最佳平衡点是：pctfree与pctused之和等于100%减去一行的大小占块空间大小的百分比。例如，如果块大小为2048字节，则它需要100个字节的开销，而行大小是390字节（为可用块的20%）。为了充分利用空间，pctfree与pctused之和最好为80%。

    那么，怎样确定数据块大小呢？有两个因素需要考虑：

    一是数据库环境类型。例如，是DSS环境还是OLTP环境？在数据仓库环境（OLAP或DSS）下，用户需要进行许多运行时间很长的查询，所以应当使用大的数据块。在OLTP系统中，用户处理大量的小型事务，采用较小数据块能够获得更好的效果。

    二是SGA的大小。数据库缓冲区的大小由数据块大小和初始化文件的db\_block\_buffers参数决定。最好设为操作系统I/O的整数倍。

5) 行链接(Row Chaining)

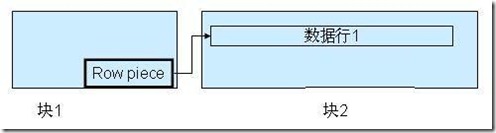
如果我们往数据库中插入(INSERT)一行数据，这行数据很大，以至于一个数据块存不下一整行，Oracle就会把一行数据分作几段存在几个数据块中，这个过程叫行链接。如下图所示：



如果一行数据是普通行，这行数据能够存放在一个数据块中;如果一行数据是链接行，这行数据存放在多个数据块中。

6) 行迁移(Row Migrating)

数据块中存在一条记录，用户执行UPDATE更新这条记录，这个UPDATE操作使这条记录变长，这时候，Oracle在这个数据块中进行查找，但是找不到能够容纳下这条记录的空间，无奈之下，Oracle只能把整行数据移到一个新的数据块。原来的数据块中保留一个“指针”，这个“指针”指向新的数据块。被移动的这条记录的ROWID保持不变。行迁移的原理如下图所示：



无论是行链接还是行迁移，都会影响数据库的性能。Oracle在读取这样的记录的时候，Oracle会扫描多个数据块，执行更多的I/O。

7) 查看标准数据库块大小

SQL> COL NAME FOR A30

SQL> COL **VALUE** FOR A30

SQL> SELECT NAME,**VALUE** FROM V$PARAMETER WHERE NAME=’db\_block\_size’;

## 2.2 oracle的内存结构

内存是影响数据库性能的第一要素，oracle内存存储的主要内容如下：

* + 程序代码
  + 关于已经连接的会话信息，包括当前所有活动会话和非活动会话
  + 程序运行时必须的相关信息，如查询计划
  + Oracle进程之间通信和共享的信息，如锁

按照内存的使用方法不同，oracle数据库的内存又可分为系统全局区（SGA）、程序全局区（PGA）、用户全局区（UGA）：

### 2.2.1  系统全局区(SGA)

SGA是一组为系统分配的共享的内存结构，包含一个数据库实例的数据或控制信息。如果多个用户连接到同一个数据库实例，在实例的SGA中，数据可以被多个用户共享，当数据库实例启动时，SGA的内存被自动分配；当数据库实例关闭时，SGA内存被回收。

SGA是占用内存最大的一个区域，同时也是影响数据库性能的重要因素。SGA按作用不同，分为以下几个部分，各个部分的size信息保存在 v$sga视图中。

1. 数据块缓存区

数据块缓存区(datablockbuffercache)是SGA中的一个高速缓存区域，用来存储从数据库中读取数据段的数据块(如表、索引和簇)。数据块缓存区的大小由数据库服务器init.ora文件中的DB\_LOCK\_BUFFERS参数决定(用数据库块的个数表示)。在调整和管理数据库时，调整数据块缓存区的大小是一个重要的部分

。

因为数据块缓存区的大小固定，并且其大小通常小于数据库段所使用的空间，所以它不能一次装载下内存中所有的数据库段。通常，数据块缓存区只是数据库大小的1%～2%，Oracle使用最近最少使用(LRU，leastrecentlyused)算法来管理可用空间。当存储区需要自由空间时，最近最少使用块将被移出，新数据块将在存储区代替它的位置。通过这种方法，将最频繁使用的数据保存在存储区中。

然而，如果SGA的大小不足以容纳所有最常使用的数据，那么，不同的对象将争用数据块缓存区中的空间。当多个应用程序共享同一个SGA时，很有可能发生这种情况。此时，每个应用的最近使用段都将与其他应用的最近使用段争夺SGA中的空间。其结果是，对数据块缓存区的数据请求将出现较低的命中率，导致系统性能下降。

2. 字典缓存区

    数据库对象的信息存储在数据字典表中，这些信息包括用户帐号数据、数据文件名、段名、盘区位置、表说明和权限，当数据库需要这些信息(如检查用户查询一个表的授权)时，将读取数据字典表并且将返回的数据存储在字典缓存区的SGA中。

    数据字典缓存区通过最近最少使用(LRU)算法来管理。字典缓存区的大小由数据库内部管理。字典缓存区是SQL共享池的一部分，共享池的大小由数据库文件init.ora中的SHARED\_POOL\_SIZE参数来设置。

如果字典缓存区太小，数据库就不得不反复查询数据字典表以访问数据库所需的信息，这些查询称为循环调用(recuesivecall)，这时的查询速度相对字典缓存区独立完成查询时要低。

3. 重做日志缓冲区

重做项描述对数据库进行的修改。它们写到联机重做日志文件中，以便在数据库恢复过程中用于向前滚动操作。然而，在被写入联机重做日志文件之前，事务首先被记录在称作重做日志缓冲区(redologbuffer)的SGA中。数据库可以周期地分批向联机重做日志文件中写重做项的内容，从而优化这个操作。重做日志缓冲区的大小(以字节为单位)由init.ora文件中的LOG\_BUFFER参数决定。

4. SQL共享池

    SQL共享池存储数据字典缓存区及库缓存区(librarycache)，即对数据库进行操作的语句信息。当数据块缓冲区和字典缓存区能够共享数据库用户间的结构及数据信息时，库缓存区允许共享常用的SQL语句。

    SQL共享池包括执行计划及运行数据库的SQL语句的语法分析树。在第二次运行(由任何用户)相同的SQL语句时，可以利用SQL共享池中可用的语法分析信息来加快执行速度。

    SQL共享池通过LRU算法来管理。当SQL共享池填满时，将从库缓存区中删掉最近最少使用的执行路径和语法分析树，以便为新的条目腾出空间。如果SQL共享池太小，语句将被连续不断地再装入到库缓存区，从而影响操作性能。

SQL共享池的大小(以字节为单位)由init.ora文件参数SHARED\_POOL\_SIZE决定。

5. 大池

大池(LargePool)是一个可选内存区。如果使用线程服务器选项或频繁执行备份/恢复操作，只要创建一个大池，就可以更有效地管理这些操作。大池将致力于支持SQL大型命令。利用大池，就可以防止这些SQL大型命令把条目重写入SQL共享池中，从而减少再装入到库缓存区中的语句数量。大池的大小(以字节为单位)通过init.ora文件的LARGE\_POOL\_SIZE参数设置，用户可以使用init.ora文件的LARGE\_POOL\_MIN\_ALLOC参数设置大池中的最小位置。Oracle8i已不用这个参数。作为使用LargePool的一种选择方案，可以用init.ora文件的SHARED\_POOL\_RESERVED\_SIZE参数为SQL大型语句保留一部分SQL共享池。

6. Java池

由其名字可知，Java池为Java命令提供语法分析。Java池的大小(以字节为单位)通过在Oracle8i引入的init.ora文件的JAVA\_POOL\_SIZE参数设置。init.ora文件的JAVA\_POOL\_SIZE参数缺省设置为10MB。

7. 多缓冲池

    可以在SGA中创建多个缓冲池，能够用多个缓冲池把大数据集与其他的应用程序分开，以减少它们争夺数据块缓存区内相同资源的可能性。对于创建的每一个缓冲池，都要规定其LRU锁存器的大小和数量。缓冲区的数量必须至少比LRU锁存器的数量多50倍。

    创建缓冲池时，需要规定保存区(keeparea)的大小和再循环区(recyclearea)的大小。与SQL共享池的保留区一样，保存区保持条目，而再循环区则被频繁地再循环使用。可以通过BUFFER\_POOL\_KEEP参数规定来保存区的大小。例如：

保存和再循环缓冲池的容量减少了数据块缓冲存储区中的可用空间(通过DB\_BLOCK\_BUFFERS参数设置)。对于使用一个新缓冲池的表，通过表的storage子句中的buffer\_pool参数来规定缓冲池的名字。例如，如果需要从内存中快速删除一个表，就把它赋予RECYCLE池。缺省池叫作DEFAULT，这样就能在以后用altertable命令把一个表转移到DEFAULT池。

8. Streams 池：由 Oracle Streams 使用

### 2.2.2 程序全局区(PGA)

  PGA不是实例的一部分，包含单个服务器进程或者单个后台进程所需的数据和控制信息，是在用户进程连接到数据库并创建一个会话时自动分配。

  该区域内保留每个与oracle数据库连接的用户进程所需的内存，当一个用户会话结束，PAG就会释放，可以用 select **sum**(pga\_used\_mem) from v$process 语句来查看当前使用的总PGA大小.

### 2.2.3 用户全局区(UGA)

  用户全局区（UGA）为用户进程存储会话状态

  UGA可以作为SGA或者PGA的一部分。具体位置取决于如何连接Oracle：如果通过一个共享服务器连接，UGA包含在SAG中。如果通过一个专有服务器连接，UGA就包含在专有服务器的PGA中

## 2.3  oracle进程结构

### 2.3.1 用户进程与服务器进程

Oracle的服务器进程由Oracle实例自动创建，用来处理连接到实例的客户端进程发出的请求，用户必须通过连接到Oracle的服务器进程来获取数据库中的信息。对于专用服务器模式，客户端进程和Oracle服务器进程是一一对应的，而在共享服务器模式下，一个Oracle服务器进程可能同时服务多个客户端进程。

服务器进程主要用来执行下列的任务：

* 解析、执行客户端提交的SQL语句。
* 从磁盘数据文件中读取必须的数据块到SGA得数据缓存区。
* 以适当形式返回SQL语句执行结果。

服务器进程用 ps -ef|grep oraclekgdb来查看。

### 2.3.2 后台进程

数据库的物理结构与内存结构之间的交互要通过后台进程来完成。 后台进程用 ps –ef|grep  ora\_ 来查看。Oracle有5个后台进程是必须启动的，否则数据库实例无法启动成功，分别是：进程监控（PMON）进程、系统监控（SMON）进程、数据写入（DBWR）进程、日志写入（LGWR）进程、检查点（CKPT）进程。

1. SMON *--System Monitor Process 系统监控进程*

    1）执行实例恢复。例如当实例是不一致性关闭后启动就需要实例恢复

    2）恢复由于读文件或表空间脱机错误引起中断的事务。

    3）清空未使用的临时段。*--相当于管理tempfile*

4）合并数据字典管理的表空间里的连续的空闲区块  *--管理数据字典管理的表空间*

2. PMON  *--Process Monitor Process 进程监控进程*

     1）监控进程与执行进程的恢复，并周期地检查调度进程（DISPATCHER）和服务器进程的状态，如果已死，则重新启动（不包括有意删除的进程）。

     2）负责清理buffer cache以及释放客户端进程使用的资源。  例如重置事务表的状态，释放锁，删除进程ID等

     3）注册实例信息和调度进程到oracle net service

     4）pmon会调查监听是否启动，若是启动会传递相关参数给listener，若是没有，pmon会定时尝试连接listener

PMON有规律地被呼醒，检查是否需要，或者其它进程发现需要时可以被调用。

3. DBWn *--Database Write Process 数据库写进程*

该进程执行将缓冲区写入数据文件，是负责缓冲存储区管理的一个Oracle后台进程。当缓冲区中的一缓冲区被修改，它被标志为“弄脏”，DBWR的主要任务是将“弄脏”的缓冲区写入磁盘，使缓冲区保持“干净”。由于缓冲存储区的缓冲区填入数据库或被用户进程弄脏，未用的缓冲区的数目减少。当未用的缓冲区下降到很少，以致用户进程要从磁盘读入块到内存存储区时无法找到未用的缓冲区时，DBWR将管理缓冲存储区，使用户进程总可得到未用的缓冲区。

Oracle采用LRU（**LEAST** RECENTLY USED）算法（最近最少使用算法）保持内存中的数据块是最近使用的，使I/O最小。在下列情况预示DBWR 要将弄脏的缓冲区写入磁盘：

    1）当脏缓冲区的数量超过了所设定的限额

    2）所设定的时间间隔已到

    3）当有进程需要数据库高速缓冲区却找不到空闲的缓冲区时

    4）当检查点发生时

    5）当某个表被删除或截断时

    6）当某个表空间被设置为只读时

    7）当对某个表空间进行联机备份时

    8）当某个临时表空间被设置为脱机状态或正常状态时

一个实例最多可配置10个DBWn进程。

查看当前的DBWn数量

SQL> show parameter db\_writer\_processes;

修改数量

SQL> alter system **set** db\_writer\_processes=20 scope=spfile;

alter system **set** db\_writer\_processes=20 scope=both;

4. CKPT  *--Checkpoint Process 检查点进程*

由于Oracle中LGWR和DBWR工作的不一致，Oracle引入了检查点的概念，负责更新控制文件和数据文件的头文件的检查点信息（检查点位置，SCN,online redo **log** 开始恢复的点）, 和触发DBWn写脏数据到磁盘，确保数据缓冲区中所有修改过的数据块都被写入数据库文件的机制，保证数据库日志文件和数据文件的同步。

DBWR将SGA中所有已改变了的数据库高速缓冲区缓存中的数据（包括已提交的和未提交的）写到数据文件中时，将产生检查点事件。保证了所有到检查点为止的变化了的数据都已经写到了数据文件中，在实例恢复时检查点之前的重做日志记录已经不再需要，从而加快了实例的恢复速度。检查点事件发生时，Oracle要将检查点号写入数据文件头中，还要将检查点号、重做日志序列号、归档日志名称和SCN号都写入控制文件中。

过于频繁的检查点会使联机操作受到冲击，因此需要在实例的恢复速度和联机操作之间折中。（大多在20分钟以上）

检查点发生时机：

    1）重做日志的切换

    2）LOG\_CHECKPOINT\_TIMEOUT  这个延迟参数的到达

    3）相应字节(LOG\_CHECKPOINT\_INTERVAL\* size of IO OS blocks)被写到当前的重做日志

    4）ALTER SYSTEM SWITCH LOGFILE， alter system archive **log** current命令

    5）ALTER SYSTEM CHECKPOINT命令

查看数据库的检查点号：

    Select checkpoint\_change# from v$database;

查看数据库当前的SCN号：

    Select current\_scn from v$database;

设置检查点

SQL> alter system checkpoint;

5. LGWR *--Log Write Process 日志写进程*

该进程将日志缓冲区写入磁盘上的一个日志文件，它是负责管理日志缓冲区的一个Oracle后台进程。LGWR进程将自上次写入磁盘以来的全部日志项输出，触发条件有：

    1) 用户提交了事务（**commit**)

    2) 日志切换发生了

    3) 每隔3s

    4) 日志缓冲区redo **log** buffer 1/3 满了 或 达到1MB

    5) DBWn 写脏数据到磁盘前, 缓冲区的数据写入磁盘前

LGWR进程同步地写入到活动的镜象在线日志文件组。如果组中一个文件被删除或不可用，LGWR可继续地写入该组的其它文件。

日志缓冲区是一个循环缓冲区。当LGWR将日志缓冲区的日志项写入日志文件后，服务器进程可将新的日志项写入到该日志缓冲区。LGWR 通常写得很快，可确保日志缓冲区总有空间可写入新的日志项。有时候当需要更多的日志缓冲区时，LWGR在一个事务提交前就将日志项写出，而这些日志项仅当在以后事务提交后才永久化。

ORACLE使用快速提交机制，当用户发出COMMIT语句时，一个COMMIT记录立即放入日志缓冲区，但相应的数据缓冲区改变是被延迟，直到在更有效时才将它们写入数据文件。当一事务提交时，被赋给一个系统修改号（SCN），它同事务日志项一起记录在日志中。由于SCN记录在日志中，以致在并行服务器选项配置情况下，恢复操作可以同步。

6. ARCn  *--归档进程 ：发生日志切换时将重做日志文件复制到归档存储器*

该进程将已填满的在线日志文件拷贝到指定的存储设备。当日志是为ARCHIVELOG使用方式、并可自动地归档时ARCH进程才存在。

7. RECO进程：

该进程是在具有分布式选项时所使用的一个进程，自动地解决在分布式事务中的故障。一个结点RECO后台进程自动地连接到包含有悬而未决的分布式事务的其它数据库中，RECO自动地解决所有的悬而不决的事务。任何相应于已处理的悬而不决的事务的行将从每一个数据库的悬挂事务表中删去。

当一数据库服务器的RECO后台进程试图建立同一远程服务器的通信，如果远程服务器是不可用或者网络连接不能建立时，RECO自动地在一个时间间隔之后再次连接。

RECO后台进程仅当在允许分布式事务的系统中出现，而且DISTRIBUTED C TRANSACTIONS参数是大于0。

8、LCKn进程：

是在具有并行服务器选件环境下使用，可多至10个进程（LCK0，LCK1……，LCK9），用于实例间的封锁。

9、Dnnn进程（调度进程）：

该进程允许用户进程共享有限的服务器进程（SERVER PROCESS）。没有调度进程时，每个用户进程需要一个专用服务进程（DEDICATEDSERVER PROCESS）。对于多线索服务器（MULTI-THREADED SERVER）可支持多个用户进程。如果在系统中具有大量用户，多线索服务器可支持大量用户，尤其在客户\_服务器环境中。

在一个数据库实例中可建立多个调度进程。对每种网络协议至少建立一个调度进程。数据库管理员根据操作系统中每个进程可连接数目的限制决定启动的调度程序的最优数，在实例运行时可增加或删除调度进程。多线索服务器需要SQL\*NET版本2或更后的版本。在多线索服务器的配置下，一个网络接收器进程等待客户应用连接请求，并将每一个发送到一个调度进程。如果不能将客户应用连接到一调度进程时，网络接收器进程将启动一个专用服务器进程。该网络接收器进程不是Oracle实例的组成部分，它是处理与Oracle有关的网络进程的组成部分。在实例启动时，该网络接收器被打开，为用户连接到Oracle建立一通信路径，然后每一个调度进程把连接请求的调度进程的地址给予它的接收器。当一个用户进程作连接请求时，网络接收器进程分析请求并决定该用户是否可使用一调度进程。如果是，该网络接收器进程返回该调度进程的地址，之后用户进程直接连接到该调度进程。有些用户进程不能调度进程通信（如果使用SQL\*NET以前的版本的用户），网络接收器进程不能将此用户连接到一调度进程。在这种情况下，网络接收器建立一个专用服务器进程，建立一种合适的连接。

10. 查看进程：用户进程和服务器进程

1) 查看用户进程和数据库进程

Ps –elf | greporacleocl |grep–v grep

2) 守护进程

Ps –elf |grep oha

3) 查看日志进程LGWR

Ps –elf |grep ora\_ | grep lgwr

### 2.3.3 oracle for windows 服务器进程

Oracle ORCL VSS Writer Service 卷映射拷贝写入服务

OracleDBConsoleorcl Oracle数据库控制台服务，用于运行Enterprise Manager。

OracleJobSchedulerORCL Oracle作业调度（定时器）服务

OracleMTSRecoveryService 服务端控制，非必需启动服务。

OracleOraDb11g home1ClrAgentOracle 数据库.NET扩展服务

OracleOraDb11g home1TNSListener 监听器服务，用于远程访问数据库。

OracleServiceORCL 数据库服务(数据库实例)，Oracle核心服务。

## 2.4 sql的执行过程

1. 客户端与服务端的sql语句传输

用户在客户端输入若干条SQL语句,例如1个普通的存储过程,有读和写的动作. 这条语句通过什么来传输呢,答案就是session啦, 那么服务器上用什么来接受这个sql语句呢,答案就是Server Process。

server process会判断sql语句是否合法(语法+权限)。如果sql语法有错,或者对应的表或视图或Procedure没有权限,就会直接返回错误信息.

2. server process根据sql语句 生成执行计划(execute plan)

Oracle是无法直接执行sql语句的,必须先生成执行计划,然后Oracle就会根据执行计划去执行了。而生成执行计划要访问许多数据库对象, 是1个比较消耗服务器资源(CPU,IO,Memory)的动作.而且因为同一条sql语句可能会有多个用户多次重复的执行, 那么是否每次都生成一次执行计划呢?

这时SGA里面的Shared pool就发挥作用了,说白了它会缓存sql的执行计划.. 所以server process会首先从Shared pool里面查找有无现成的执行计划, 如果有就直接采用. 如果无, 就自己生成1个,然后看情况把这个执行计划放入shared pool.

3. server process根据执行计划去取(写)数据

当server process得到执行计划后就可以去取数据了.Oracle的数据放在哪里呢, 放在数据文件,这个大家都懂,但是server process是不是就直接去访问数据文件呢?

因为计算机的时间消耗主要都在物理IO,所以要尽量避免物理读写,所以SGA里面的database buffer cache起作用了.说明白点, database buffer cache就是用来缓冲Data files里的数据的.? 这样就可以避免了对数据文件的读写.

所以server process得到执行计划后,第一步是首先去database buffer cache去找有没有现成的数据. 如果有最好, 如果无或者缓存中数据不全的话就只能去访问data\_files啦.

从data\_files获得数据后, 也不是直接发给用户客户端,而是根据情况放入database\_buffer\_cache里面, 以便当前或其他用户2次使用啦.

4. 在缓存中修改数据

server process拿出数据放入缓存中,接下来就对数据进行修改啦.因为修改数据很可能会产生大量缓冲数据,所以这个动作是在Database buffer cache里完成的.

修改数据会产生重做日志,这些日志是不是直接就写到日志文件中,写日志文件也是物理读,所以SGA就有个Redo **log** buffer,就是日志缓存啦, 专门实时存放产生的日志数据啦~

5. 返回结果给用户

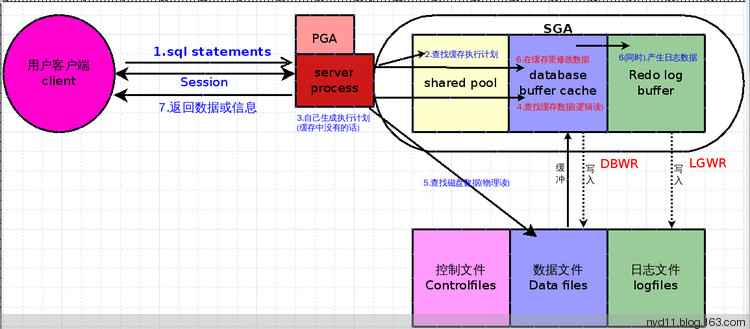
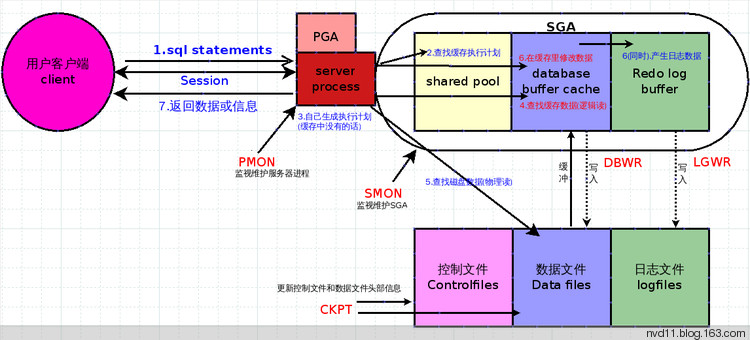
Server Process 做完读取和修改数据的动作后,就会将结果返给用户了。其实到上面那一步为止,? 整个sql语句执行流程已经完成了.

可能有人会问, Server process修改的数据和产生的日志还在SGA里面啊, 它们不用被写入磁盘吗?答案是肯定需要的, 但是这些动作已经不是sql执行流程之内, 而且这些动作也不是server process负责的, 他们分别由DBWR?和?LGWR?这个两个进程负责.

6. 将数据缓存和日志缓存写入磁盘

DBWR: Database writer,后台进程之一,负责将Database buffer cache里被修改的数据写入数据文件.

LGWR: **Log** writer,后台进程之一,负责将Redo **log** buffer里的日志数据写入到日志文件.



附1：逻辑读,物理读与缓存命中率

逻辑读：从缓存一般是内存里读取数据

物理读：从磁盘数据文件里读取

缓存命中率：逻辑读次数/ (逻辑读次数+物理读次数)

读数据的缓存命中率越接近1越好，当然缓存命中率并不是数据库健康的唯一指标，其它诸如每秒物理读(tps)也很重要，在linux下可以用iostat命令来查看当前磁盘的每秒物理读。

附2：为什么Oracle要将server process 和后台进程分开?

如上图,为什么写入数据文件和日志文件要交给后台进程去完成呢.

其实我们在流程可以发现, 服务器与用户打交道的就只有1个进程,就是Server Process啊,? 所以server Process的速度直接影响了用户的感受, 无论后台进程多么繁忙,只要server process响应迅捷, 用户还是觉得数据库很快的. 相反,后台进程没事做,服务器CPU很空闲, 但server process反应慢的话, 用户就觉得数据库慢了.

所以就要尽量精简server process的动作,? 看看后台进程DBWR 和 LGWR进行的是什么动作, 磁盘写动作啊!比物理读还慢.所以这些动作完全可以在sql流程执行完慢慢来嘛.Server Process唯一进行的物理操作就是物理读,? 这个是无办法避免的, 因为数据都在磁盘上嘛... 除非有办法预测用户要提取的数据,提前拿出来.但也没有那么大的内存啊.

案例一：验证内存结构工作的原理（select语句是如何被执行的？）

当用户执行语句select  \*  from scott.emp的时候，工作原理分为四步，如下：

第一步：先做编译，编译包含了语法的检查和语义的检查。

第二步：如果编译没有问题会进入shared pool(共享池 )中，而shared pool 是由library cache和data dictionary cache（数据字典缓存）组成。Sql语句第一时间会进入library cache中，即library cache中会存放被编译过的正确的sql语句，仅仅是sql语句而已。library cache中没有缓存的sql语句需要进行硬解析（即需要耗费设备的I/0资源从硬盘读取）。假设library cache中有这个sql语句，就不需要做硬解析，接着会查 data dictionary cache。data dictionary cache中缓存了权限和对象数据及属性，所有data dictionary cache检查用户是否对scott.emp表是否具有访问权限，如果有权限继续往下执行，如果没有权限，直接给用户返回一个结果。如果一条语句从来都没有被执行过，这条语句首先被缓存在share pool中的library cache中，下一次被执行的时候直接从share pool中取sql语句。存放sql语句也需要占用空间，share pool的空间也是有限的，为了防止share pool空间不够用，采用先进先出的规则即后执行的sql语句会覆盖先执行的sql语句来释放空间，当然share pool的空间越大越好。

第三步：data buffer cache现在hr用户可以访问scott.emp表了，data buffer cache的作用就是从磁盘或存储中将数据块调入内存，缓存在buffer cache中。总结：buffer cache缓存数据本身，library cache缓存sql语句本身。

第四步：redo **log** buffer<重做日志缓冲区>当用户执行insert,update,delete,create,alter等操作后，数据发生了变化，这些变化了的数据写入数据缓冲区（buffer cache）之前，先写入重做日志缓冲区，同时变化之前的数据也放入重做日志缓存中，这样做，是为了保证在数据恢复时oracle就知道哪些事务需要提交，哪些事务需要撤回。

可以从v$sgainfo中查看shared pool size

测试语句执行时间：

SQL> **set** timing on

SQL> select \* from scott.emp;第一次执行sql语句

已用时间:  00: 00: 00.05

SQL> /再次执行上一条命令所用的时间为00: 00: 00.00

已用时间:  00: 00: 00.00

通过上面的测试发现，缓存提高了查询的速度。

## 2.5 oracle数据的启动与关闭

oracle的启动与关闭需要 sysdba 或 sysoper 验证，因此应先切换到oracle所在的用户，并以sysdba或sysoper权限登录数据库：

    #su - oracle

    $lsnrctl start           *--启动监听，可以用$lsnrctl status查看是否正常启动*

    $sqlplus / as sysdba     *--以dba身份进入sqlplus*

### 2.5.1 数据库启动

由于Oracle数据库的启动过程是分步进行的，因此数据库可以有多种模式启动。不同的启动模式之间能够相互切换。此外，除正常方式启动数据库外，还能够将数据库设置为受限状态或只读状态，以满足特殊管理工作的需要。Oracle启动的过程如下：

    1) 创建并启动与数据库有关的实例             *--    只访问 .ora文件*

    2) 为实例加载数据库                       *--    只访问 .ctl文件*

    3) 将数据库设置为打开状态                  *--    只访问 .dbf文件*

1. 启动数据库所使用的STARTUP命令格式如下：

startup[nomount|mount|open |force] [restrict][pfile=filename]

参数的说明：

    open：启动例程、装载并打开数据库，默认选项

    nomount：启动实例，访问初始化参数文件，分配SGA区，创建实例进程，没有打开控制文件、数据文件、重做日志文件，无法访问数据库。

    mount：启动实例并装载数据库，创建实例进程，打开控制文件，但没打开数据文件和重做日志文件。

    force：强制关闭数据库并重新启动数据库

    pfile：用于指定启动实例时所使用的文本参数文件。

restrict：以受限会话方式启动数据库，只允许具有restricted session或者create session权限的用户访问数据库。

【注意】：nomount和mount两种启动方式仅仅DBA可以进行管理，其他数据库用户都不能访问数据库。

2. 转换启动模式

在进行某些特定的管理和维护操作时，需要使用某种特定的启动模式来启动数据库。如

必须在nomount启动模式下进行的：

* 创建新的数据库
* 重建控制文件

必须在mount启动模式下进行的：

* 重命名数据文件
* 添加删除或重命名重做日志文件
* 执行数据库的完全恢复操作
* 改变数据库的归档模式

转换启动模式命令如下：

startup nomount             *-- 启动实例，但未加载数据库*

alter database mount；       *-- 转换启动模式，已加载数据库*

alter database open；        *-- 转换启动模式，已打开数据库受限状态和非受限状态的切换*

3. 切换受限状态

若是打开数据库的时候设置为受限状态，在完成管理工作后，需要将数据库恢复为非受限状态：

    alter system disable restricted session；

在数据库运行过程中需要将非受限状态切换到受限状态：

    alter system enable restricted session；

4. 切换只读状态

正常启动模式下，数据库默认处于读写状态。但有时候需要将数据库设置为只读状态。

    startup mount

    alter database open read only；

### 2.5.2  orace数据库的关闭

在关闭数据库与实例时，需要使用一个具有SYSDBA权限的用户账户连接到Oracle中，然后使用SHUTDOWN语句执行关闭操作：

    #su - oracle              切换到oracle用户且切换到它的环境

    $lsnrctl stop             停止监听

    $sqlplus / as sysdba      以DBA身份进入sqlplus

与数据库的启动相对应，关闭数据库时也是分步骤进行的：

    关闭数据 -> 实例卸载数据库 -> 终止实例

1. 关闭数据库所使用的SHUTDOWN命令：

shutdown [normal | transactional | immediate | abort]

1) 正常模式（NORMAL）：shutdown normal

如果对关闭数据库的时间没有限制，通常会使用正常方式来关闭数据库。以正常方式关闭数据库时，oracle将执行：

* 阻止任何用户建立新的连接
* 等待用户主动断开连接，已经连接的用户能够继续当前操作。
* 一旦所有用户都断开连接，则立即关闭、卸载数据库，并终止实例。

2) 立即关闭模式（IMMEDIATE）：shutdown immediate

立即方式能够在尽可能短的时间内关闭数据库。以立即方式关闭数据库时，Oracle将执行如下操作：

* 阻止任何用户建立新的连接，同时阻止当前连接的用户开始任何新的事务。
* 任何未提交的事务均被回退。
* Oracle不再等待用户主动断开连接，而是直接关闭、卸载数据库，并终止实例。

3) 事务关闭模式（TRANSACTIONAL）：shutdown transactional

事务方式介于正常关闭方式与立即关闭方式之间，它能够在尽可能短的时间内关闭数据库，但是能够保证当前所有的活动事务都可以被提交。使用事务方式关闭数据库时，Oracle将等待所有未提交的事务完成后再关闭数据库。

4) 终止关闭方式（ABORT）：shutdown abort

如果上述三种关闭方式都无法关闭数据库，说明数据库产生了严重的错误，这时只能使用终止方式来关闭数据库。终止关闭方式将丢失一部分数据信息，会对数据库的完整性造成损害。因此，如果重新启动实例并打开数据库时，后台进程SMON会执行实例恢复。一般情况下，应当尽量避免使用这种方式来关闭数据库。

### 2.5.3   数据库的静默和挂起状态

静默状态和挂起状态是两种特殊的数据库状态，利用这两种数据库状态，数据库管理员可以完成一些特殊的管理和维护操作。

1. 静默状态

在静默状态下，只有SYS和SYSTEM用户才能够在数据库中执行查询、更新操作，运行PL/SQL程序，任何非管理员用户都不能在数据库中执行任何操作。

在数据库运行过程中，执行如下的语句将进入和退出静默状态：

    alter system quiesce restricted；

    alter system unquiesce；

2. 挂起状态

当数据库处于挂起状态时，数据库所有的物理文件（控制文件、数据文件和重做日志文件）的I/O操作都被暂停，这样能够保证数据库在没有任何I/O操作的情况下进行物理备份。

    alter system suspend；

    alter system resume；

挂起状态与静默状态的区别是：挂起状态并不禁止非管理员用户进行数据库操作，只是暂时停止所有用户的I/O操作。

## 2.6 oracle的初始化参数

Oracle在启动实例、打开数据库时将读取初始化参数文件，并从中获取的初始化参数初始化参数用于设置实例和数据库的特征。

oracle11g提供的初始化参数有200多个，常用的如DB\_NAME、DB\_BLOCK\_SIZE等。可以通过show parameter [参数名]命令查看参数信息，如：show parameter db\_block\_size

这些命令保存在spfile<SID>.ora文件中，它是一个二进制的服务器端初始化参数文件，可通过alter system命令修改。

# 第二章 Oracle数据库管理

## 2.1 用户管理

**Oracle**是基于用户的数据管理方案，其一个实例只有一个数据库，一个数据库可以有多个用户，不同的表属于某个用户管理，在oracle实例创建时，会创建若干的默认用户，如

    sys：拥有权限最大的超级用户，相当于SQL server中的sa帐户，只能以系统管理员（sysdba）或系统操作员（sysoper）的权限登录。

    system：通常用来创建一些用于查看管理信息的表或视图，权限比system小，登陆时以normal权限登陆，同样也不建议使用system架构来创建一些与管理无关的表或视图。

    sysman：是Oracle数据库中用于EM管理的用户，可以选择性删除。

scott：Oracle提供的示例用户之一，为普通用户，默认密码为tiger。

还有一些其它用户，可以通过下面语句来解锁

alter **user** scott account unlock

### 2.1.1 创建一个新用户

1. 创建新用户时可以为用户指定默认表空间和临时表空间，语法如下：

create **user** hyx identified by hyx123

    default tablespace ts\_hyx

    temporary tablespace ts\_tmp\_hyx;

用户创建后可以用

    select username from dba\_users 来查看创建的用户。

2. 用户创建后一般需要授权：

    grant connect to hyx;

    grant resource to hyx;

### 2.1.2 删除或锁定用户

1. 删除用户：

    drop **user** hyx cascade;        *-- cascade表示连同其创建的东西一起删除*

2. 锁定用户：

    alter **user** hyx account lock;  *-- 用sys用户执行*

### 2.1.3 更改用户密码

    alter **user** hyx identified by hyx147741

## 2.2 表空间管理

**表空间**实际上是数据库上的逻辑存储结构，可以把表空间理解为在数据库中开辟的一个空间，用于存放我们数据库的对象，一个数据库可以由多个表空间组成，而表空间则由一个或多个数据文件构成的，数据文件的位置和大小可以由我们用户自己来定义。

表空间按功能分可以分为永久表空间、临时表空间以及UNDO表空间。

### 2.2.1 创建表空间和临时表空间

1. 创建默认表空间：

create tablespace ts\_hyx datafile 'c:\app\hyx\oradata\orcl\ts\_hyx.dbf' size 10M;

2. 创建临时表空间：

create temporary tablespace ts\_tmp\_hyx tempfile 'c:\app\hyx\oradata\orcl\ts\_tmp\_hyx.dbf' size 20M;

创建的表空间文件默认保存在C:\APP\HYX\PRODUCT\11.2.0\DBHOME\_1\DATABASE\，该文件可用

select file\_name from dba\_data\_files;

select file\_name from dba\_temp\_files;

来查看，如果创建数据库文件错误，可能是路径写错或所在目录没有权限。

### 2.2.2 查看表空间

1. 查看系统所有表空间：

    select tablespace\_name from dba\_tablespaces;

 查看用户所有表空间：

    select tablespace\_name from user\_tablespaces;

2. 查看SCOTT用户的表空间信息：

   select default\_tablespace,temporary\_tablespace from dba\_users where username = ‘SCOTT’;

 也可登陆scott用户后用：

    select default\_tablespace,temporary\_tablespace from user\_users 查看。

### 2.2.3 修改表空间

1. 修改用户的表空间

修改用户默认表空间和临时表空间：alter **user** hyx default tablespace ts\_hyx temporary tablespace ts\_tmp\_hyx;

*-- ALTER USER username DEFAULT | TEMPORARY TABLESPACE tablespace\_name;*

2. 修改表空间状态

修改表空间状态(联机或脱机)：

    ALTER TABLESPACE tablespace\_name ONLINE | OFFLINE;

设置表空间状态(只读或可读写)：

    ALTER TABLESPACE tablespace\_name READ ONLY | READ WRITE;

修改后可通过select tablespace\_name, status from dba\_tablespaces来查看表空间的状态。

3. 修改表空间数据文件

增加数据文件：

    alter tablespace ts\_hyx add datafile ‘ts\_hyx\_02.dbf’ size 10M;

删除数据文件：

    alter tablespace ts\_hyx drop datafile ‘ts\_hyx\_02.dbf’

*-- 不能删除表空间中的第一个创建的数据文件，如果需要删除的话，我们需要把整个的表空间删掉。*

可通过

    select file\_name from dba\_data\_files where tablespace\_name = ‘TS\_HYX’;(注意大写)来查看表空间的文件信息。

### 2.2.4 删除表空间

DROP TABLESPACE tablespace name [INCLUDING CONTENTS]

例： drop tablespace TS\_TEST including content;

-- including contents表示连同表空间文件也一起删除。

## 2.3 角色管理

**角色**就是一组权限或者说是权限的集合。用户可以给角色赋予指定的权限，然后将角色赋给相应的用户。

### 2.3.1 Oracle的默认角色

**connect(连接角色)**：拥有Connect权限的用户只可以登录Oracle，不可以创建实体，不可以创建数据库结构，如不可以创建表、视图、索引。

**RESOURCE（资源角色）**：拥有Resource权限的用户只可以创建实体，不可以创建数据库结构，如不能创建其它用户。

**DBA（数据库管理员角色）**：拥有全部特权，是系统最高权限，只有DBA才可以创建数据库结构。

一般来说，对于普通用户来说，授予connect、resoure权限，对于DBA管理用户，授予dba权限。

### 2.3.2 角色的使用

1.创建角色

create role role\_01;

2.为角色赋权

grant create table,create view **to** role\_01;

3.将角色赋予用户

grant role\_01 to hyx;

4.收回权限

revoke role\_01 from hyx;

5.删除角色

    drop role role\_01;

## 2.4 权限管理

**权限**指的是执行特定命令或访问数据库对象的权利。对数据库的安全性(系统安全性、数据安全性)有着至关重要的作用。权限可分为系统权限(允许用户执行特定的数据库动作，如创建表、索引、连接实例等)和对象权限(也叫实体权限，允许用户操纵一些特定的对象，如读取视图、更新某些列、执行存储过程等)

### 2.4.1 系统权限

常用的系统权限可通过 select \* from system\_privilege\_map 来查看，列举部分如下：

|  |  |
| --- | --- |
| CREATE SESSION | **创建会话** |
| CREATE SEQUENCE | 创建序列 |
| CREATE TABLE | 创建表 |
| CREATE USER | 创建用户 |
| ALTER USER | 更改用户 |
| DROP USER | 删除用户 |
| CREATE VIEW | 创建视图 |

1. 授予系统权限：

    GRANT privilege I, privilege..] TO **user** I, userl role, PUBLIC.…J

例：grant create table,create sequence to hyx;

2. 回收系统权限：

    REVOKE {privilege I role}FROM {user\_name I role\_name | PUBLIC}

    例：revoke create table, create sequence from hyx;

### 2.4.2 对象权限

常用的对象权限可通过select \* from table\_privilege\_map来查询Oracle所以对象权限，常用的对象权限有select、update、insert、delete、all等。

1. 授予对象权限

    GRANT object\_priv |ALL [(columns)]ON object TO{userlrolelPUBLIC}

例： grant select,update,insert on scott.emp to role\_02;

2. 回收对象权限

    REVOKE { privilege I, privilege... JIALL} ON object FROM { **user**[, **user**...] Irole |PUBLIC}

例： revoke all on scott.emp from hyx;

## 2.5 事务管理

**事务**可以看作是由对数据库的若干操作组成的一个单元，这些操作要么都完成，要么都取消，从而保证数据满足一致性的要求。事务可以是一条或多条DML语句、一条DDL语句或一条DCL语句等。其中DDL和DCL的事务是自动提交的，而DML语句需要使用COMMIT提交或ROLLBACK回滚事务。

当执行事务操作（DML语句）时，Oracle会在被作用表上加表锁，以防止其他用户改变表结构；同时会在被作用行上加行锁，以防止其他事务在相应行上执行DML操作。

举例：如果SCOTT用户：update emp **set** sal = 6000 where sid=7369; 但是没有提交，那么SYS用户：修改表结构alter table scott.emp add remark varchar(200)时会一直等待，因为加了表锁，而update scott.emp st sal = 8000 where sid = 7369时也会等待，因为SCOTT对这一行加了行锁。

### 2.5.1 事务控制命令

1. 提交事务：

**commit**

事务提交后，会删除保存点，释放锁，同时其它会话将可以查看到事务变化后的数据。

2. 回滚事务：

**rollback** [ to savapoint\_name]

对未提交的事务进行撤销。

3. 保存点：

**savepoint** sp1。

事务中的定位点，可用rollback回退到指定的保存点，当事务提交后该会话的所有保存点都将被删除。

### 2.5.2 事务的隔离级别

**事务的隔离级别**是一个事务对数据库的修改与并行的另一个事务的隔离程度。两个并发事务同时访问数据库相同的行时，可能会发生以下问题：

    1) 幻读：事务T1读取一条指定where条件的语句，返回结果集。此时事务T2插入一行新记录，恰好满足T1的where条件。然后T1使用相同的条件再次查询，结果集中可以看到T2插入的记录，这条新纪录就是幻读。

    2) 不可重复读：事务T1读取一行记录，紧接着事务T2修改了T1刚刚读取的记录，然后T1再次查询，发现与第一次读取的记录不同，这称为不可重复读。

    3) 脏读：事务T1更新了一行记录，还未提交所做的修改，这个T2读取了更新后的数据，然后T1执行回滚操作，取消刚才的修改，所以T2所读取的行就无效，也就是脏数据。

为了处理这些问题，SQL标准定义了以下几种事务隔离级别：

    READ UNCOMMITTED：三种都允许

    READ COMMITTED：  不允许脏读

    REPEATABLE READ： 不允许脏读和可不重复读

    SERIALIZABLE：    三种都不允许

但oracle只支持READ COMMITTED (默认)和 SERIALIZABLE这两种事务隔离级别。设置oracle事务隔离级别可以用：

**SET** TRANSACTION ISOLATION LEVEL [READ COMMITTED|SERIALIZABLE]

## 2.6 回收站管理

从ORACLE 10g开始，引入了一个叫**回收站(Recycle Bin)**的概念。它的全称叫Tablespace Recycle Bin。回收站实际是一个逻辑容器（逻辑区域），原理有点类似于WINDOW系统的回收站。它以表空间中现有已经分配的空间为基础，而不是从表空间上物理划出一个固定区域用作回收站。这意味着回收站和表空间中的对象共用存储区域、系统没有给回收站预留空间。因此，当表被DROP后，如果可用空间充足，并且没有对回收站进行清理，那么被DROP掉的对象会一直存在回收站中，但是如果可用空间紧张的情况下，数据库会根据先进先出的顺序覆盖Recycle Bin中的对象。所以回收站机制也不是百分百的保险机制。

另外从原理上来说它就是一个数据字典表，放置用户Drop掉的数据库对象信息。用户进行Drop操作的对象并没有真正被数据库删除，仍然会占用空间。除非是由于用户手工进行Purge或者因为存储空间不够而被数据库清掉。数据库有了这样的功能，能够减少很多不必要的麻烦。当用户、开发人员、甚至DBA误操作删除了表，那么我们不必还原整个数据库或表空间，直接使用ORACLE 10g的闪回(FLASHBACK,闪回)功能来还原被删除的表。这样我们就能避免大量的人工误操作。这是一个对DBA相当有用的功能。

### 2.6.1 开启和关闭回收站

首先你可以通过命令查看数据库是否开启了回收站机制， 如下所示 **VALUE**= ON表示开启了回收站机制。OFF则表示回收站机制关闭。

SQL> SHOW PARAMETER RECYCLEBIN;

NAME              TYPE        **VALUE**

*--------     ----------- ----------*

recyclebin      string         ON

或

SQL> SELECT NAME, **VALUE** FROM V$PARAMETER WHERE NAME='recyclebin';

NAME                      **VALUE**

*----------------- --------------------*

recyclebin                  on

可以通过设置初始化参数recyclebin启用或禁用回收站功能,可以用命令关闭回收站

SQL> ALTER SYSTEM **SET** RECYCLEBIN=OFF;

System altered.

SQL> ALTER SESSION **SET** RECYCLEBIN=OFF;

Session altered.

也可以用命令开启回收站

SQL> ALTER SYSTEM **SET** RECYCLEBIN=ON;

System altered.

SQL> ALTER SESSION **SET** RECYCLEBIN =ON;

Session altered.

### 2.6.2 查看回收站对象

我们先来看看一个例子，如下所示，假如不小心误操作DROP了表test，那么我们如何在回收站查看被DROP的表对象呢？

SQL> show **user**

**USER** is "ODS"

SQL> create table test(name varchar2(16));

Table created.

SQL> insert into test select 'kerry' from dual;

1 row created.

SQL> insert into test select 'ken' from dual;

1 row created.

SQL> **commit**;

**Commit** complete.

SQL> drop table test;

Table dropped.

1. 查看当前用户的回收站

SQL> show recyclebin

ORIGINAL NAME    RECYCLEBIN NAME                OBJECT TYPE  **DROP** TIME

*---------------- ------------------------------ ------------ -------------------*

TEST             BIN$BLmi9vltN3TgUKjAgYxoiA==$0 TABLE        2014-10-06:11:25:38

或

SQL> SELECT \* FROM RECYCLEBIN      *-- 其中RECYCLEBIN是USER\_RECYCLEBIN 的同义词。*

或

SQL> SELECT \* FROM USER\_RECYCLEBIN;

可以通过下面语句来设置sqlplus的显示格式

COL OBJECT\_NAME FOR A30

COL ORIGINAL\_NAME FOR A8

COL OPERATION FOR A9

COL TYPE **FOR** A8

COL DROPTIME FOR A19

COL TS\_NAME FOR A30

2. 查看数据库回收站所有对象，需要相关权限才能查询。

SQL> SELECT \* FROM DBA\_RECYCLEBIN;

3. 为了避免被删除的表与同类对象名称的重复，被删除的表以及相依的对象放到回收站后，ORACLE数据库会对被删除的对象名称进行重命名，例如两次创建了表TEST，然后删除了该表TEST，如下所示，虽然ORIGINAL\_NAME一致，但是RECYCLEBIN NAME则有所不同。

SQL> show recyclebin

ORIGINAL NAME    RECYCLEBIN NAME                OBJECT TYPE  **DROP** TIME

*---------------- ------------------------------ ------------ -------------------*

TEST             BIN$BLmi9vluN3TgUKjAgYxoiA==$0 TABLE        2014-10-06:14:40:52

TEST             BIN$BLmi9vltN3TgUKjAgYxoiA==$0 TABLE        2014-10-06:11:25:38

clip\_image002

RECYCLEBIN NAME的命名规则为BIN$GUID$Version 其中GUID为GlobalUID，是一个全局唯一、24个字符长的标识对象，它是ORACLE内部使用的标识。 其中$version是ORACLE数据库分配的版本号。

### 2.6.3 还原回收站对象

还原回收站被删除的表、索引等对象， 是通过Flashback Drop实现的。

还原回收站中的表

SQL> FLASHBACK TABLE TEST TO BEFORE DROP;

Flashback complete.

SQL> SELECT \* FROM TEST;

NAME

*----------------*

kerry

ken

但是如果出现上面两个TEST表都被删除时，此时的Flashback Drop就有点意思了

SQL> show recyclebin

ORIGINAL NAME    RECYCLEBIN NAME                OBJECT TYPE  **DROP** TIME

*---------------- ------------------------------ ------------ -------------------*

TEST             BIN$BLmi9vlxN3TgUKjAgYxoiA==$0 TABLE        2014-10-06:15:10:25

TEST             BIN$BLmi9vlwN3TgUKjAgYxoiA==$0 TABLE        2014-10-06:15:10:09

SQL> flashback table test to before drop;

Flashback complete.

SQL> show recyclebin

ORIGINAL NAME    RECYCLEBIN NAME                OBJECT TYPE  **DROP** TIME

*---------------- ------------------------------ ------------ -------------------*

TEST             BIN$BLmi9vlwN3TgUKjAgYxoiA==$0 TABLE        2014-10-06:15:10:09

如上所示，如果两个相同名字的表TEST被删除了，此时闪回被DROP的表TEST，实质是闪回最后一个被删除的表（后进先出原则）,如果此时继续闪回操作就会报ORA-38312错误

SQL> flashback table test to before drop;

flashback table test to before drop

\*

ERROR at line 1:

ORA-38312: original name is used by an existing object

此时可以在闪回过程中对表名进行重命名解决问题。。

SQL> flashback table test to before drop rename to test\_2;

Flashback complete.

SQL> select \* from test\_2;

NAME

*----------------*

kerry

ken

另外，如果回收站有两个被DROP掉的表TEST, 如果想闪回第一个被删除的表，那该怎么办呢？

SQL> show recyclebin

ORIGINAL NAME    RECYCLEBIN NAME                OBJECT TYPE  **DROP** TIME

*---------------- ------------------------------ ------------ -------------------*

TEST             BIN$BLmi9vlxN3TgUKjAgYxoiA==$0 TABLE        2014-10-06:15:10:25

TEST             BIN$BLmi9vlwN3TgUKjAgYxoiA==$0 TABLE        2014-10-06:15:10:09

其实这个也很好处理,直接指定RECYCLEBIN NAME进行闪回即可。

SQL> flashback table "BIN$BLmi9vlwN3TgUKjAgYxoiA==$0" to before drop;

Flashback Drop注意事项

    1)  只能用于非系统表空间和本地管理的表空间。在系统表空间中，表对象删除后就真的从系统中删除了，而不是存放在回收站中。

    2)  对象的参考约束不会被恢复，指向该对象的外键约束需要重建。

    3)  对象能否恢复成功，取决于对象空间是否被覆盖重用。

    4)  当删除表时，依赖于该表的物化视图也会同时删除，但是由于物化视图并不会放入recycle binzhong，因此当你执行flashback drop时，并不能恢复依赖其的物化视图。需要DBA手工重建。

    5)  对于回收站（Recycle Bin）中的对象，只支持查询。不支持任何其他DML、DDL等操作。

### 2.6.4 清空回收站

数据库对象删除后，数据库会把它重命名为BIN$开头的对象，你可以通过ORIGINAL\_NAME查看它对应的原始对象名称。记住，将表放在回收站里并不在原始表空间中释放空间。

如果您希望完全删除该表，而不让该表放入回收站，可以使用以下命令永久删除该表。当然这样操作后，你也不能通过使用闪回特性闪回该表了。

DROP TABLE TABLE\_NAME PURGE;

如果数据库中删除表时都放入回收站，因而没有释放所占空间，那么当空闲的空间不足时，已经删除的表是否还会侵占存储空间呢?

答案很简单：当表空间被回收站数据完全占满，以至于必须扩展数据文件来容纳更多数据时，可以说表空间处于“空间压力”情况下。此时，对象以先进先出的方式从回收站中自动清除。在删除表之前，相关对象(如索引)被删除。同样，空间压力可能由特定表空间定义的用户限额而引起。表空间可能有足够的空余空间，但用户可能将其在该表空间中所分配的部分用完了。在这种情况下，Oracle 自动清除该表空间中属于该用户的对象。

此外，有几种方法可以手动控制回收站。如果在删除名为 TEST 的特定表之后需要从回收站中清除它，可以执行

PURGE TABLE TABLE\_NAME;

或者使用其回收站中的名称：

PURGE TABLE "BIN$04LhcpndanfgMAAAAAANPw==$0";

此命令将从回收站中删除表 TEST 及所有相关对象，如索引、约束等，从而节省了空间。

但是，如果要从回收站中永久删除索引，则可以使用以下命令来完成工作：

PURGE INDEX IN\_TEST1\_O1;

此命令将仅仅删除索引，而将表的拷贝留在回收站中。有时在更高级别上进行清除可能会有用。

例如，您可能希望清除表空间 USERS 的回收站中的所有对象。可以执行：

PURGE TABLESPACE USERS;

您也许希望只为该表空间中特定用户清空回收站。在数据仓库类型的环境中，用户创建和删除许多临时表，此时这种方法可能会有用。您可以更改上述命令，限定只清除特定的用户：

PURGE TABLESPACE USERS **USER** SCOTT;

要释放整个回收站占用的空间，您需要使用以下命令清空回收站：

PURGE RECYCLEBIN;

记住PURGE RECYCLEBIN只是清除当前用户回收站中的对象，DBA\_RECYCLEBIN下的的对象并没有删除。

如果你要清除当前数据库回收站的对象，必须使用下面命令（DBA权限）

PURGE DBA\_RECYCLEBIN

## 数据库备份与恢复

### 2.7.1 逻辑备份-EXP/IMP

1. 导入数据

1) 将备份文件导入到数据库

    imp scott/tiger1@orcl file=c:\oracle\_bak\scott\_bak.dmp ignore=y

2) 将scott用户的备份文件导入到yanln用户中

    imp yanln/yanln@orcl fromuser=scott touser=yanln file=c:\oracle\_bak\scott\_bak.dmp

2. 数据的导出

1) 将数据库orcl完全导出

**exp** system/oracle@orcl file=c:\oracle\_baklorcl\_bak.dmp full=y

2) 将数据库中scott用户的所有对象导出

**exp** scott/tiger1@orcl file=c:\oracle\_bak\scott\_bak.dmp owner=scott

3) 将scott用户中表emp，dept导出

**exp** scott/tiger1@orcl file=c:\oracle\_bak\table\_bak.dmp tables=/emp,dept）

3. **EXP**/IMP增量导出／导入

增量导出是一种常用的数据备份方法，它只能对整个数据库来实施，并且必须作为SYSTEM来导出。在进行此种导出时，系统不要求回答任何问题。导出文件名缺省为export.dmp，如果不希望自己的输出文件定名为export.dmp，必须在命令行中指出要用的文件名。

增量导出包括三种类型：

1) 完全增量导出（Complete）即备份三个数据库，比如：

**exp** system/manager inctype=complete file=040731.dmp

2) 增量型增量导出,备份上一次备份后改变的数据，比如：

**exp** system/manager inctype=incremental file=040731.dmp

3) 累积型”增量导出,导出自上次“完全”导出之后数据库中变化了的信息。比如：

**exp** system/manager inctype=cumulative file=040731.dmp

示例：数据库管理员可以排定一个备份日程表，用数据导出的三个不同方式合理高效的完成。

比如数据库的被封任务可以做如下安排：

　　星期一：完全备份（A）

　　星期二：增量导出（B）

　　星期三：增量导出（C）

　　星期四：增量导出（D）

　　星期五：累计导出（E）

　　星期六：增量导出（F）

　　星期日：增量导出（G）

如果在星期日，数据库遭到意外破坏，数据库管理员可按一下步骤来回复数据库：

　　第一步：用命令CREATE DATABASE重新生成数据库结构；

　　第二步：创建一个足够大的附加回滚。

　　第三步：完全增量导入A：

imp system/manager inctype=RESTOREFULL=y FILE=A

　　第四步：累计增量导入E：

imp system/manager inctype=RESTOREFULL=Y FILE=E

　　第五步：最近增量导入F：

imp system/manager inctype=RESTOREFULL=Y FILE=F

### 2.7.2 逻辑备份-EXPDP/IMPDP

EXPDP/IMPDP是 ORCALE10G 提供了新的导入导出工具——数据泵，其主要特性如下：

    1) 支持并行处理导入、导出任务

    2) 支持暂停和重启动导入、导出任务

    3) 支持通过Database Link的方式导出或导入远端数据库中的对象

    4) 支持在导入时自动修改对象属主、数据文件或数据所在表空间。

5) 导入/导出时提供了非常细粒度的对象控制。

但使用是应注意EXPDP和IMPDP是服务端的工具程序，他们只能在ORACLE服务端使用，不能在客户端使用。

在使用EXPDP/IMPDP前可以事先可以创建逻辑目录(oracle)和实际目录(host)创建逻辑目录

create directory dpdata1 as 'd:\test\dump';

目录创建后可以通过 select \* from dba\_directories 语句查看，在使用前把目录的权限授予用户：

grant read,write on directory dpdata1 to scott;

1. 导出数据库

1)按用户导

   expdp scott/tiger@orcl schemas=scott dumpfile=expdp.dmp DIRECTORY=dpdata1;

2)并行进程

   expdp scott/tiger@orcl directory=dpdata1 dumpfile=scott3.dmp

parallel=40 job\_name=scott3

   expdp ananda/abc123 tables=CASES directory=DPDATA1 dumpfile=expCASES\_%U.dmp

parallel=4 job\_name=Cases\_Export

3)按表名导

   expdp scott/tiger@orcl TABLES=emp,dept dumpfile=expdp.dmp DIRECTORY=dpdata1;

4)按查询条件导

   expdp scott/tiger@orcl directory=dpdata1 dumpfile=expdp.dmp

Tables=emp query='WHERE deptno=20';

5)按表空间导

   expdp system/manager DIRECTORY=dpdata1 DUMPFILE=tablespace.dmp

TABLESPACES=temp,example;

6)导整个数据库，只有拥有dba或者 exp\_full\_database和imp\_full\_database权限的用户才能执行

   expdp system/manager DIRECTORY=dpdata1 DUMPFILE=full.dmp FULL=y;

2. 还原数据

1)导到指定用户下

   impdp scott/tiger DIRECTORY=dpdata1 DUMPFILE=expdp.dmp SCHEMAS=scott;

2)改变表的owner

impdp system/manager DIRECTORY=dpdata1 DUMPFILE=expdp.dmp

TABLES=scott.dept REMAP\_SCHEMA=scott:system;

3)导入表空间

    impdp system/manager DIRECTORY=dpdata1 DUMPFILE=tablespace.dmp TABLESPACES=example;

4)导入数据库

impdb system/manager DIRECTORY=dump\_dir DUMPFILE=full.dmp FULL=y;

5)追加数据

impdp system/manager DIRECTORY=dpdata1 DUMPFILE=expdp.dmp

SCHEMAS=system TABLE\_EXISTS\_ACTION

3. 使用参数文件

使用参数文件可以避免繁长的命令行，如

expdp **user**/pwd parfile=xxx.par logfile=a.**log**

使用参数文件同样可以使用参数，参数值以最后出现的为准。

5. 交互方式的使用

使用CTRL+C退出当前模式，连接当前正在执行的备份作业:

expdp bam/bam attach=SYS\_IMPORT\_FULL\_01

作业名称默认为SYS\_XXX,可以用JOB\_NAME = job\_string指定，可以用SELECT \* FROM DBA\_DATAPUMP\_JOBS 查看存在的job.

交互窗口常用命令如下：

1) 查看JOB的运行状态

    Export> status

2) 回退到命令行

    Export> continue\_client

3) 增加并行

    Export> parallel=4

4) 停止JOB

    Export> stop\_job

5) 启动JOB

    Export> start\_job

6) 杀掉JOB

    Export> kill\_job

7) 退出交互模式

    Export> exit\_client

8) 指定文件大小

    Export> filesize=1G

9) 帮助

    Export> Help

6. 重要参数的使用

1) Query = [Schema.][Table\_name:] Query\_clause

    EG:Query=TABLE\_A:”Where id<5″,TABLE\_B:”Where name=’a’”

*-- 如果Where条件前没有指定Schema名或者表名的话，默认就是针对当前所有要被导出的表。*

2) INCLUDE=[object\_type]:[name\_clause],[object\_type]:[name\_clause]

INCLUDE=PROCEDURE:"LIKE 'PROC\_U%'"              *--包含以PROC\_U开头的所有存储过程(\_ 符号代表任意单个字符)*

INCLUDE=TABLE:"> 'E' "                          *--包含大于字符E的所有表对象*

3）EXCLUDE=[object\_type]:[name\_clause],[object\_type]:[name\_clause]

EXCLUDE=SEQUENCE,VIEW                           **--**过滤所有的SEQUENCE,VIEW

EXCLUDE=TABLE:"IN ('EMP','DEPT')"               *--过滤表对象EMP,DEPT*

EXCLUDE=SEQUENCE,VIEW,TABLE:"IN ('EMP','DEPT')" *--过滤所有的SEQUENCE,VIEW以及表对象EMP,DEPT*

EXCLUDE=INDEX:"= 'INDX\_NAME'"                   *--过滤指定的索引对象INDX\_NAME*

注意：INCLUDE和EXCLUDE不可以连用，并且为了避免符号转义，一般使用参数文件形式

Windows平台需要对双引号进行转义，如 INCLUDE=TABLE:\"IN ('EMP', 'DEPT')\"

Unix平台所有的符号都需要进行转义，包括括号，双引号，单引号等，如INCLUDE=TABLE:\"IN \(\'EMP\', \'DEP\'\)\"

4) CONTENT = ALL|DATA\_ONLY|METADATA\_ONLY    *-- 对象定义+数据 | 仅数据 | 仅对象定义*

5) Table\_exists\_action = SKIP|APPEND|TRUNCATE|**REPLACE**   *-- 表示跳过 | 追加 | 截断 | 重建*

*-- 如果同时指定了CONTENT参数为Data\_only的话，那么SKIP变为apped，REPLACE参数无效*

6) REMAP\_SCHEMA=Source\_schema:Target\_schema[,Source\_schema:Target\_schema]

EG:Remap\_schema=a:b,c:d

7) REMAP\_TABLESPACE=Source\_tablespace:Target\_tablespace[,Source\_tablespace:Target\_tablespace]

*-- 使用Remap\_tablespace参数，则要保证导入的用户对目标表空间有读写权限*

8) parallel

    在导出时应根据filesize和导出后实际文件的大小来设置parallel，如某用户对象占用了4G左右的空间，实际导出后的DUMP文件约为3G，我们尝试在导出该用户时指定并行度为4，设置单个文件不超过500M：

    expdp **user**/pwd directory=dump\_file dumpfile=expdp\_20100820\_%U.dmp logfile=expdp\_20100820.**log** filesize=500M parallel=4

    在导入时参数设置为几，则认为同时将几张表的内容导入到库中，如：

    impdp **user**/pwd directory=dump\_file dumpfile=expdp\_20100820.dmp logfile=impdp\_20100820.**log** parallel=10

6. NETWORK\_LINK导出文件到本地

    1) 在本地建立dblink到远程

SQL> create database link dl\_monitor connect to monitor identified by monitor using 'lobom55';

    2) 数据泵导出

F:\Documents and Settings\Administrator>expdp network\_link=dl\_monitor dumpfile=n

etwork\_monitor.dmp

    3) 检查结果：

SQL> **set** head off

SQL> select \* from dba\_directories where directory\_name='DATA\_PUMP\_DIR';

### 2.7.3 物理备份-热备份

热备份也叫联机备份，它是指数据库处于open状态下，对数据库的数据文件、控制文件、参数文件、密码文件等进行一系列备份操作，它要求数据库处在归档模式下。

由于数据文件处于备份状态时重做日志后台进程要将这些文件的所有的变化数据块写到重做日志文件中，这对重做日志缓冲区和重做日志文件的压力都增大了，所以需要注意几下加点：

    1) 重做日志缓冲区和重做日志文件适当增大

　　2) 在联机备份时，每次只备份一个表空间

　　3) 在DML最少的时候做备份

1. 热备份的优点：

　　1) 可在表空间或数据库文件级备份，备份的时间短。

　　2) 备份时数据库仍可使用。

　　3) 可达到秒级恢复（恢复到某一时间点上）。

　　4) 可对几乎所有数据库实体做恢复

　　5) 恢复是快速的，在大多数情况下数据库仍工作时恢复。

2. 热备份的不足：

    1) 不能出错，否则后果严重

　　2) 若热备份不成功，所得结果不可用于时间点的恢复

　　3) 因难于维护，所以要特别仔细小心，不允许“以失败告终”。

3. 热备份表空间

1) 设置表空间为备份状态，表空间的名字和状态可用 select file\_id,tablespace\_name,status from dba\_data\_files 查看

SQL> alter tablespace PDMS\_DATA  begin backup;

2) 备份表空间的数据文件

SQL> host cp /opt/oracle/app/oradata/pdms/PDMS\_DATA.dbf /u03/backup/hotbak

3) 恢复表空间为正常状态，

alter tablespace PDMS\_DATA end **backup**;

tips: 备份完成后可用 select \* from v$backup 查看数据文件的备份状态

5. 热备份全数据库

    1) alter database begin backup;

    2) 拷贝所有的datafile到备份目录

    3) alter database end **backup**;

6. 热备份控制文件

SQL> alter database backup controlfile to '/u03/backup/hotbak/control\_01.ctl';  *--控制文件的完整备份*

SQL> alter database backup controlfile to trace as '/u03/backup/hotbak/control\_02.ctl';     *--用于创建控制文件的语句，丢失了部分信息，可以直接查看*

8. 热备份参数文件

SQL> create pfile = '/u03/backup/hotbak/initdbsrv1.ora' from spfile;

9. 临时表空间的数据文件、日志文件不需要备份

### 2.7.4 物理备份-冷备份

冷备份也叫脱机备份，其在关闭数据库后，将关键性文件拷贝到另外的位置，是最快和最安全的方法。注意其必须拷贝全部文件的备份，包括参数文件、所有控制文件、所有数据文件、所有联机重做日志文件。

1. 冷备份的优点：

    1) 是非常快速的备份方法（只需拷贝文件）

    2) 容易归档（简单拷贝即可）

    3) 容易恢复到某个时间点上（只需将文件再拷贝回去）

    4) 能与归档方法相结合，作数据库“最新状态”的恢复。

    5) 低度维护，高度安全。

2. 冷备份的不足：

    1) 单独使用时，只能提供到“某一时间点上”的恢复。

    2) 在实施备份的全过程中，数据库必须要作备份而不能作其它工作。也就是说，在脱机备份过程中，数据库必须是关闭状态。

    3) 若磁盘空间有限，只能拷贝到其它外部存储设备上，速度会很慢。

    4) 不能按表或按用户恢复。

3. 冷备份的步骤：

    1) 以sys用户或者system身份登录数据库，以immediate方式关闭数据库。

    2) 把需要备份的文件拷贝到指定的位置。

    3) 拷贝完成后，如果继续让用户使用数据库，需要以open方式启动数据库。

4. 脱机恢复

数据库恢复就是把数据库中备份出来的数据重新还原给原来的数据库。数据库恢复技术分为完全恢复和不完全恢复.完全恢复是指把数据库恢复到数据库失败时的数据库状态;不完全恢复是指将数据库恢复到数据库失败前的某一时刻的数据库状态。脱机恢复的步骤分为三步：

    1) 以sys用户或者system身份登录数据库，以immediate方式关闭数据库。

    2) 把所有备份文件全部拷贝到原来的位置。

    3) 恢复完成后，如果继续让用户使用数据库，需要以open方式启动数据库。

### 2.7.5 其它数据备份恢复方法

1. 导出数据为.sql文件(不能导出表结构)

# 第三章 Oracle数据库对象

## 3.1 基本表

### 3.1.1 创建表

1. 可以从零定义一个表结构

CREATE TABLE student (

    sid        NUMBER(8, 0),

    name       VARCHAR2(20),

    sex        CHAR(2),

    birthday   DATE,

    address    VARCHAR2(50)

);

2. 也可以复制另一个表的结构，同时复制其数据

create table student

as

select \* from emp;

注：可以在创建表时指定表空间，直接在最后加tablespace <tablespace\_name>即可。

表创建后可以查询该表的段的表空间，区id等信息

SQL> select tablespace\_name,extent\_id,bytes,file\_id,block\_id from dba\_extents where segment\_name=’HEHE’;

### 3.1.2 删除表

1. 删除数据库对象一般都用drop

drop table student

### 3.1.3 修改表

1. 表级的修改，如修改表名

rename student to student\_info;

2. 列级的修改，修改表结构一般都以 alter table 开头

增加列：alter table student add tel varchar2(11);

修改列：alter table student modify tel number(11,0);

删除列：alter table student drop column tel;

## 3.2 表的约束

**约束**是ORACLE提供的自动保持数据库完整性的一种方法，它通过限制字段中数据、记录中数据和表之间的数据来保证数据的完整性。

### 3.2.1 主键(primary key)约束

**primary key**约束用于定义基本表的主键，它是唯一确定表中每一条记录的标识符，其值不能为NULL，也不能重复，以此来保证实体的完整性。

表中主键只能有一个，但可以由多个列构成。如：primary key（学号，科目编号）。

1. 可以在创建表的同时添加主键约束，如

create table student(

    sid number(8,0),

    name varchar2(20),

    sex char(2),

    birthday date,

    constraint pk\_sid primary key(sid)

);

2. 也可以通过修改表结构来添加主键约束，如

alter table student add constraint pk\_sid primary key(sid);

*--其中 constraint pk\_sid 可以省略*

### 3.2.2 唯一性(unique)约束

**唯一性约束**用于指定一个或者多个列的组合值具有唯一性，以防止在列中输入重复的值。使用唯一性约束的列允许空值。

1. 可以在创建表的时候添加唯一性约束，如

create table student(

    sid number(8,0),

    email varchar2(50) constraint uk\_email unique,

    cardid varchar2(18),

    constraint uk\_cardid unique(cardid)

);

2. 也可以通过修改表结构来添加唯一性约束，如

alter table student add constraint uk\_cardid unique(cardid);

### 3.2.3 非空(NOT NULL)约束

**NOT NULL约束**用于确保列不能为NULL。如果在列上定义了NOT NULL约束，那么当插入数据时，必须为该列提供数据；当更新列数据时，不能将其值设置为NULL。

1. 可以在创建表时添加列级非空约束，如

create table student(

    sid number(8,0)

    name varchar2(20) not null,                 *--匿名非空约束*

    sex char(2) constraint nn\_sex not null,     *--非空约束*

    birthday date,

);

*-- 特别注意，NOT NULL 约束必须定义为列级约束*

2. 也可以通过修改表结构来添加非空约束，如

alter table student modify(name varchar2(20) constraint nn\_name not null);

*--注意：添加非空约束采用的是 modify ，而非 add ，且非空约束的类型为 check*

### 3.2.4 默认值(default)约束

**默认值约束**用于指定列的默认值，在插入数据且没有输入该列的值时生效。可防止意外的空值输入。

1. 可以在创建表的时候添加默认值约束，如

create table student(

    sid number(8,0),

    sex varchar2(2) default '男',

);

2. 也可以通过修改表结构来添加默认值约束，如

alter table student modify (sex varchar(2) default ‘男’);

*-- 与非空约束类似，添加默认值约束用modify，而非add*

设置默认值后，插入数据时也可以显示使用默认值，如

    insert into student values(20150001,’王五’,default,21);

### 3.2.5 检查(check)约束

**检查约束**对输入列或者整个表中的值设置检查条件，以限制输入值，保证数据库数据的完整性。

1. 可以在创建表的时候添加检查约束，如

create table student(

    sid number(8,0),

    name varchar2(20),

    sex char(2) constraint ck\_sex check(sex in('男', '女')),    *-- 列级*

    constraint ck\_sid check(sid > 1000 and sid < 10000)         *--表级*

);

2. 也可以通过修改表结构来添加检查约束，如

alter table student add constraint ck\_sex check(sex in('男', '女'));

### 3.2.6 外键(FOREIGN KEY)约束

**外键（FOREIGNKEY）**是用于建立和加强两个表数据之间的链接的一列或多列。外键约束是唯一涉及两个表关系的约束。

1. 可以在创建表时添加外键约束，如

create table student(

    sid number(8,0),

    name varchar2(20),

    depid varchar2(10) references department(depid),

    course varchar2(12),

    address    VARCHAR2(50),

    constraint fk\_course foreign key(course) references Course(course)

);

*-- student是从表，department、Course是主表，应先创建主表再创建从表，插入数据也类似。*

2.也可以通过修改表结构添加外键约束，如

alter table student add constraint fk\_depid foreign key(depid)

references department(depid) on delete cascade;

*-- on delete cascade 表示在删除主表记录时级联删除从表记录，类似的还有级联置空*

*--注意：*

*--1 设置外键约束时主表的字段必须是主键列（或唯一列）。*

*--2 主从表中相应字段必须是同一数据类型。*

*--3 从表中外键字段的值必须来自主表中的相应字段的值，或者为NULL值。*

### 3.2.7 删除或禁用约束

1. 删除约束：

    alter table student drop constraint constraint\_name

2. 删除主键约束：

    alter table student drop constraint key

3. 禁用/激活约束：

alter table student disable | enable constraint constraint\_name

### 3.2.8 查看约束信息

1. 查看表的约束

    select \* from user\_constraints where table\_name = 'INT\_RATE'

2. 查看约束列

select \* from User\_Cons\_Columns  where table\_name = 'INT\_RATE';

## 3.3 表的视图

### 3.3.1 什么是视图？

    1) 视图(view)，也称虚表, 不占用物理空间，这个也是相对概念，因为视图本身的定义语句还是要存储在数据字典里的。视图只有逻辑定义。每次使用的时候,只是重新执行SQL。

    2) 视图是从一个或多个实际表中获得的，这些表的数据存放在数据库中。那些用于产生视图的表叫做该视图的基表。一个视图也可以从另一个视图中产生。

    3) 视图的定义存在数据库中，与此定义相关的数据并没有再存一份于数据库中。通过视图看到的数据存放在基表中。

    4) 视图看上去非常象数据库的物理表，对它的操作同任何其它的表一样。当通过视图修改数据时，实际上是在改变基表中的数据；相反地，基表数据的改变也会自动反映在由基表产生的视图中。

    5) 还有一种视图：物化视图（MATERIALIZED VIEW ），也称实体化视图，快照 （8i 以前的说法） ，它是含有数据的，占用存储空间。

    6) 查询视图没有什么限制, 插入/更新/删除视图的操作会受到一定的限制; 所有针对视图的操作都会影响到视图的基表; 为了防止用户通过视图间接修改基表的数据, 可以将视图创建为只读视图(带上with read only选项)

### 3.3.2 视图的作用

    1)提供各种数据表现形式, 可以使用各种不同的方式将基表的数据展现在用户面前, 以便符合用户的使用习惯(主要手段: 使用别名)；

    2)隐藏数据的逻辑复杂性并简化查询语句, 多表查询语句一般是比较复杂的, 而且用户需要了解表之间的关系, 否则容易写错; 如果基于这样的查询语句创建一个视图, 用户就可以直接对这个视图进行"简单查询"而获得结果. 这样就隐藏了数据的复杂性并简化了查询语句.这也是oracle提供各种"数据字典视图"的原因之一,all\_constraints就是一个含有2个子查询并连接了9个表的视图(在catalog.sql中定义)；

    3)执行某些必须使用视图的查询. 某些查询必须借助视图的帮助才能完成. 比如, 有些查询需要连接一个分组统计后的表和另一表, 这时就可以先基于分组统计的结果创建一个视图, 然后在查询中连接这个视图和另一个表就可以了；

    4)提供某些安全性保证. 视图提供了一种可以控制的方式, 即可以让不同的用户看见不同的列, 而不允许访问那些敏感的列, 这样就可以保证敏感数据不被用户看见；

    5)简化用户权限的管理. 可以将视图的权限授予用户, 而不必将基表中某些列的权限授予用户, 这样就简化了用户权限的定义。

### 3.3.3 创建视图

要在当前方案中创建视图, 用户必须具有create view系统权限; 要在其他方案中创建视图, 用户必须具有create any view系统权限. 视图的功能取决于视图拥有者的权限.创建视图的语法如下：

    create [ or **replace** ] [ force ] view [schema.]view\_name [ (column1,column2,...) ]

        as select ...

    [ with check option ] [ constraint constraint\_name ]

    [ with read only ];

tips:

 1) or **replace**: 如果存在同名的视图, 则使用新视图"替代"已有的视图

 2) force: "强制"创建视图,不考虑基表是否存在,也不考虑是否具有使用基表的权限

 3) column1,column2,...：视图的列名, 列名的个数必须与select查询中列的个数相同; 如果select查询包含函数或表达式, 则必须为其定义列名.此时, 既可以用column1, column2指定列名, 也可以在select查询中指定列名.

 4) with check option: 指定对视图执行的dml操作必须满足“视图子查询”的条件即,对通过视图进行的增删改操作进行"检查",要求增删改操作的数据, 必须是select查询所能查询到的数据,否则不允许操作并返回错误提示. 默认情况下, 在增删改之前"并不会检查"这些行是否能被select查询检索到.

 5) with read only：创建的视图只能用于查询数据, 而不能用于更改数据.

### 3.3.4 简单视图

简单视图定义：是指基于单个表建立的，不包含任何函数、表达式和分组数据的视图。

1. 创建简单视图

    SQL> conn /as sysdba

    SQL> grant create view **to** scott;

    SQL> conn scott/tiger

    SQL> create view **vw\_emp** as select empno,ename,job,hiredate,deptno from emp;

    SQL> desc vw\_emp

2. 对简单视图进行DML操作,基表也会发生相应的更改

    SQL> insert into vw\_emp values(1,'a','aa','05-JUN-88',10);

    SQL> update vw\_emp **set** ename='cc' where ename='KING';

    SQL> delete vw\_emp where ename='cc';

    SQL> select \* from vw\_emp where deptno=10;

3. 创建只读视图, 只读视图只能查询，无法进行更改。更新基表，只读视图会发生相应的更改

SQL> create view **vw\_emp\_readonly** as select empno,ename,job,hiredate,deptno from emp with read only;

4. 创建检查约束视图, 对通过视图进行的增删改操作进行检查，要求增删改操作的数据必须是select查询所能查询到的数据

Sql>create view **vw\_emp\_check** as select empno,ename,job,hiredate,deptno from emp where deptno=10 with check option;

SQL> insert into vw\_emp\_check values('2','c','cc','02-JAN-55',10);

SQL> insert into vw\_emp\_check values('3','d','dd','02-JAN-65',20);

    ORA-01402: view **WITH** CHECK OPTION where-clause violation

*-- 20号部门不在查询范围内，违反检查约束,所以无法插入；*

SQL> delete vw\_emp\_check where empno=2;

    1 row deleted.

*--所删除的数据在查询范围内，不违反检查约束*

### 3.3.5 连接视图

连接视图是指基于多个表所创建的视图，即，定义视图的查询是一个连接查询。 主要目的是为了简化连接查询；

1. 创建连接视图

示例1： 查询部门编号为10和30的部门及雇员信息

create view **vw\_dept\_emp**

    as select a.deptno,a.dname,a.loc,b.empno,b.ename,b.sal

            from dept a,emp b

            where a.deptno=b.deptno and a.deptno in(10,30);

2. 连接视图上的DML操作：在视图上进行的所有DML操作，最终都会在基表上完成， select 视图没有什么限制，但insert/delete/update有一些限制。

SQL> insert into vw\_dept\_emp values(10,'aaa','aaaa',22,'a',5000);

    ORA-01779: cannot modify a column which maps to a non key-preserved table

3. 键值保存表

如果连接视图中的一个“基表的键”(主键、唯一键)在它的视图中仍然存在，并且“基表的键”仍然是“连接视图中的键”(主键、唯一键)；即，某列在基表中是主键|唯一键，在视图中仍然是主键|唯一键，则称这个基表为“键值保存表”。 一般地，由主外键关系的2个表组成的连接视图，外键表就是键值保存表，而主键表不是。

4. 连接视图的更新准则

(一) 一般准则

    1) 任何DML操作，只能对视图中的键值保存表进行更新, 即，“不能通过连接视图修改多个基表”;

    2) 在DML操作中，“只能使用连接视图定义过的列”;

    3) “自连接视图”的所有列都是可更新(增删改)的

(二) insert准则

    1) 在insert语句中不能使用“非键值保存表”中的列(包括“连接列”)；

    2) 执行insert操作的视图，至少应该“包含”键值保存表中所有设置了约束的列；

    3) 如果在定义连接视图时使用了WITH CHECK OPTION 选项，则“不能”针对连接视图执行insert操作

(三) update准则

    1) 键值保存表中的列是可以更新的；

    2) 如果在定义连接视图时使用了WITH CHECK OPTION 选项，则连接视图中的连接列(一般就是“共有列”)和基表中的“其他共有列”是“不可”更新的，连接列和共有列之外的 其他列是“可以”更新的

(四) delete准则

    1) 如果在定义连接视图时使用了WITH CHECK OPTION 选项，依然“可以”针对连接视图执行delete操作

5. 可更新连接视图

如果创建连接视图的select查询“不包含”如下结构，并且遵守连接视图的“更新准则”，则这样的连接视图是“可更新”的：

  1) 集合运算符(union,intersect,minus)

  2) DISTINCT关键字

  3) GROUP BY，ORDER BY，CONNECT BY或START WITH子句

  4) 子查询

  5) 分组函数

  6) 需要更新的列不是由“列表达式”定义的

  7) 基表中所有NOT NULL列均属于该视图

### 3.3.6 复杂视图

复杂视图是指包含函数、表达式、或分组数据的视图,主要用于执行查询操作,目的是为了简化查询,并不用于执行DML操作。

  注意：当视图的select查询中包含函数或表达式时，必须为其定义列别名。

1. 创建复杂视图

 示例1：查询目前每个岗位的平均工资、工资总和、最高工资和最低工资。

create view **vw\_emp\_job\_sal**(job,avgsal,sumsal,maxsal,minsal)

    as select job,**avg**(sal),**sum**(sal),**max**(sal),**min**(sal) from emp group by job;

2. 强制创建视图

强制视图定义：正常情况下，如果基表不存在，创建视图就会失败。但是可以使用force选项强制创建视图(前提：创建视图的语句没有语法错误！)，此时该视图处于失效状态。

SQL> create force view **vw\_test\_tab**

  2  as select c1,c2 from test\_tab;

Warning: View **created** with compilation errors. *-- 警告: 创建的视图带有编译错误。*

此时的视图不可查看：

SQL> select \* from vw\_test\_tab;

*-- ERROR at line 1:*

*-- ORA-04063: view "SCOTT.VW\_TEST\_TAB" has errors*

可用在user\_objects中查看视图的状态：

SQL> select object\_name,status from user\_objects where object\_name='VW\_TEST\_TAB';

OBJECT\_NAME                    STATUS

*------------------------------ -------*

VW\_TEST\_TAB                    INVALID

可以看到 视图状态 为 不可用

3. 复杂视图上执行DML操作的原则

视图包含下列元素之一不能执行delete操作：

    1) Group by子句

    2) 分组函数

    3) Distinct关键字

    4) Rownum伪列

包括以下之一不能执行update操作：

    1) delete所有限制

    2) 使用表达式定义的列

包含以下之一不能执行insert操作：

    1) update所有限制

    2) 视图上没有包含基表的NOT NULL列

### 3.3.7 更改视图

在对视图进行更改(或重定义)之前，需要考虑如下几个问题：

    1) 由于视图只是一个虚表，其中没有数据，所以更改视图只是改变数据字典中对该视图的定义信息，视图的所有基础对象都不会受到任何影响

    2) 更改视图之后，依赖于该视图的所有视图和PL/SQL程序都将变为INVALID(失效)状态

    3) 如果以前的视图中具有with check option选项，但是重定义时没有使用该选项，则以前的此选项将自动删除。

1. 更改视图的定义

方法—— 执行create or **replace** view语句。这种方法代替了先删除(“权限也将随之删除”)后创建的方法，会保留视图上的权限，但与该视图相关的存储过程和视图会失效。

EG:将视图改为改为只读

create or replace view **vw\_emp**

    as select empno,ename,job,hiredate,deptno from emp with read only;

2. 视图的重新编译

当视图依赖的基表改变后，视图会“失效”。为了确保这种改变“不影响”视图和依赖于该视图的其他对象，应该使用 alter view 语句“明确的重新编译”该视图，从而在运行视图前发现重新编译的错误。视图被重新编译后，若发现错误，则依赖该视图的对象也会失效；若没有错误，视图会变为“有效”。

    语法：alter view 视图名 compile;

    权限：为了重新编译其他模式中的视图，必须拥有alter any table系统权限。

    注意：当访问基表改变后的视图时，oracle会“自动重新编译”这些视图。

 示例1：

SQL> select last\_ddl\_time,object\_name,status from user\_objects where object\_name='VW\_TEST\_TAB';

LAST\_DDL\_ OBJECT\_NAME                    STATUS

*--------- ------------------------------ -------*

23-AUG-14 VW\_TEST\_TAB                    VALID     *-----视图的状态:有效*

SQL> alter table test\_tab modify(c2 varchar2(30)); ——修改基表，c2列的长度

SQL> select last\_ddl\_time,object\_name,status from user\_objects where object\_name='VW\_TEST\_TAB';

LAST\_DDL\_ OBJECT\_NAME                    STATUS

*--------- ------------------------------ -------*

23-AUG-14 VW\_TEST\_TAB                    INVALID     ——视图的状态:失效

SQL> alter view **vw\_test\_tab** compile;                 *---重新编译*

SQL> select last\_ddl\_time,object\_name,status from user\_objects where object\_name='VW\_TEST\_TAB';

LAST\_DDL\_ OBJECT\_NAME                    STATUS

*--------- ------------------------------ -------*

23-AUG-14 VW\_TEST\_TAB                    VALID         *---视图有效*

思考：若上述代码修改的不是列长，而是表名，结果又会如何？

   <警告：更改的视图带有编译错误；视图状态：失效>

### 3.3.8 删除视图

    1) 一般用户可以删除当前模式中的任何视图；如果要删除其他模式中的视图，必须拥有DROP ANY VIEW系统权限；

    2) 视图被删除后，该视图的定义会从词典中被删除，并且在该视图上授予的“权限”也将被删除。

    3) 视图被删除后，其他引用该视图的视图及存储过程等都会失效。

示例1：drop view **vw\_test\_tab**;

### 3.3.9 查看视图

使用数据字典查看视图

    1) dba\_views            *-- DBA视图描述数据库中的所有视图*

    2) all\_views            *-- ALL视图描述用户“可访问的”视图*

    3) user\_views           *-- USER视图描述“用户拥有的”视图*

    4) dba\_tab\_columns      *--DBA视图描述数据库中的所有视图的列(或表的列)*

    5) all\_tab\_columns      *-- ALL视图描述用户“可访问的”视图的列(或表的列)*

    6) user\_tab\_columns     *-- USER视图描述“用户拥有的”视图的列(或表的列)*

    7) dba\_updatable\_columns   *-- 显示数据库所有视图中的所有列的可更新状态*

    8) all\_updatable\_columns   *-- 显示用户可访问的视图中的所有列的可更新状态*

    9) user\_updatable\_columns  *-- 显示用户拥有的视图中的所有列的可更新状态*

示例1：查询当前方案中所有视图的信息

SQL> select view\_name,text from user\_views;

示例2：查询当前方案中指定视图(或表)的列名信息

  select \* from user\_tab\_columns where table\_name='VW\_DEPT';

示例3：查询视图的可更新列

SQL> select table\_name,column\_name,insertable,updatable,deletable from user\_updatable\_columns;

### 附： 在视图上执行DML操作的步骤和原理

第一步：将针对视图的SQL语句与视图的定义语句(保存在数据字典中)“合并”成一条SQL语句

第二步：在内存结构的共享SQL区中“解析”(并优化)合并后的SQL语句

第三步：“执行”SQL语句

示例：假设视图v\_emp的定义语句如下：

create view **v\_emp**

as select empno,ename,loc from employees emp,departments dept

where emp.deptno=dept.deptno and dept.deptno=10;

当用户执行如下查询语句时：

select ename from v\_emp where empno=9876;

oracle将把这条SQL语句与视图定义语句“合并”成如下查询语句：

select ename from employees emp,departments dept

    where emp.deptno=dept.deptno and dept.deptno=10 and empno=9876;

然后，解析(并优化)合并后的查询语句，并执行查询语句;

## 3.4 表的索引

### 3.4 Oracle 索引简介

在 Oracle 数据库中，存储的每一行数据都有一个 rowID 来标识。当 Oracle 中存储着大量的数据时，意味着有大量的 rowID ，此时想要快速定位指定的 rowID ，就需要使用索引对象。

当对 Oracle 表执行指定条件的查询时，常规的方法是将所有的记录取出来，然后再把每一条记录与查询条件作对比，最后返回满足条件的记录。这样操作不仅耗费时间并耗费资源。当有了索引之后，只需要在索引中找到符合查询条件的索引字段值，就可以通过保存在索引中的 rowID 快速找到表中对应的记录。

用户可以根据情况不同创建多种类型的索引。按照索引的存储方式将索引分为 B 树索引、位图索引、反向索引和基于函数的索引。

创建索引时需要注意以下几点：

* 索引应该建立在 where 子句频繁引用、排序以及分组的列上，如果选择的列不合适将无法提升查询速度；
* 限制索引的个数。索引只要提升查询速度，但会降低 DML 操作的速度；
* 指定索引块空间的使用参数。基于表建立索引时， Oracle 会将相应表添加到索引块。为索引添加数据时， Oracle 会按照 pctfree 参数在索引块上预留部分空间。如果将来在表上执行大量的 insert 操作，那么应该在建立索引时设置较大的 pctfree ；
* 将表和索引部署到相同的表空间，可以简化表空间的管理；将表和索引部署到不同的表空间，可以提高访问性能（ Oracle 能够并行读取不同硬盘的数据）；

### 3.4.1 创建索引

       在创建索引时， Oracle 首先对将要建立索引的字段进行排序，然后将排序后的字段值和对应记录的 rowID 存储在索引段中。查询时，根据索引查询指定条件的 rowID ，再根据 rowID 提取数据行。

1. 创建 B树索引

B 树索引是 Oracle 最常用的索引类型（也是默认类型），是以 B 树结构组织并存放索引数据。默认情况下 B 树索引中的数据是以升序方式排列的。 B 树索引是由根块、分支块和叶块组成。

    例：create index index\_test on emp(deptno) Pctfree 25 Tablespace users;

子句 pctfree 指定为将来 insert 操作所预留的空闲空间,子句 tablespace 指定索引段所在的表空间.

2. 创建位图索引

当需创建索引列包含的取值太少时，如对性别列创建索引，取值只能是“男”或者“女”，使用 B 树索引取出来的值任然太多，失去了索引的意义，这种情况则需要使用位图索引.

例： create bitmap index index\_test on emp(salary) tablespace users;

初始化参数 create\_bitmap\_area\_size 用于指定建立位图索引时分配的位图区大小，默认值为8 MB ，该参数越大建立位图索引的速度越快。

修改该参数语句为：

              alter system **set** create\_bitmap\_area\_size =8388608 Scope = spfile ；修改后需要重新启动数据库方可生效；

3. 创建反向键索引

       在单调递增的列上使用 B 树索引时，若用户对表中数据做了删除操作，将导致对某一边的叶子节点的大量占用。 Oracle 提供另一种索引机制，及反向键索引，它可以将添加的数据随机分散到索引中。反向键索引是一种特殊的 B 树索引，在顺序递增数列上建立索引非常有用。反向键索引在原理和存储结构方面和 B 树索引类似。当用户插入记录时，将列值进行反向操作后进行索引，此时数据不在是递增的，所以新数据在值的范围上分布通常比原来的有序树更均匀。

例：create index index\_test on emp(deptno) reverse tablespace users ;

如果该列上已经建立了 B 树索引，那么可以将其修改为反向键索引：

              Alter index index\_test rebulid reverse;

4. 基于函数的索引

       用户在使用数据库时，最常遇到的问题是大小写字符敏感。例如在 emp 表中 Job 字段有 MANAGER 的记录，当用户使用小写搜索时则无法找到该记录，只能通过函数 **upper** 对应进行转换，在使用转换后的数据进行检查。但是这样查询是，即便 job 列有普通索引， Oracle 也会执行全表搜索，并为遇到的各个行计算 **upper** 函数。这种情况可以使用建立基于函数的索引，通常只是常规 B 树索引，但是它存放的数据是由表中数据应用函数后得到的，而不是直接存放表中的数据本身。

如果习惯使用小写字符串，可以创建如下索引：

       create index index\_test on emp( **lower**(job) );

5. 索引的唯一性

  在创建索引时，可以限制索引关联字段在表中的记录是唯一的，语法如下：

    create unique index ...

6. 创建过滤索引

create unique\_index CUACCT\_LOG\_UI2 on CUACCT\_LOG (

    case when BIZ\_CODE IN ('20040011','20040019','20040116','20040123') then VOUCHER\_NO else **to\_char**(SERIAL\_NO) end **ASC**,

    ORIGIN\_SUBSYS\_SN ASC,

    SETT\_DATE ASC

)

create unique index CUBSBLOG\_GI3 on CUBSB\_LOG (

    case when CUBSB\_TRD\_ID IN ('01','02') then EXT\_SERIAL\_NO else **to\_char**(SERIAL\_NO) end **ASC**,

    EXT\_ORG ASC,

    SETT\_DATE ASC

)

### 3.4.2 修改索引

       修改索引使用 alter index 完成。为表建立索引后，随着对表不断进行更替、插入和删除动作，索引中会产生越来越多的存储碎片，导致索引工作效率降低。这是可以采取重建索引和合并索引清除碎片。合并索引只是将 B 树中叶子节点的存储碎片合并在一起，并不会改变索引的物流组织结构。

1. 合并索引

alter index index\_test **Coalesce** deallocate unused ;

2. 重建索引

alter index index\_test rebuild tablespace **user** 1；

重建索引时可以更改索引的类型，存储表空间等

### 3.4.3 删除索引

删除索引使用 drop index 语句。当索引过于碎片化，或者不经常被用到时，既可以删除索引：

drop index index\_test ；

    注：删除表是也会删除其相应的索引。

### 3.4.4 显示索引信息

为了显示索引的信息， Oracle 提供了一系列数据字典视图，使用户了解索引的各方面信息。

1. 显示表的所有索引：显示 emp 的所有索引

Select \* from dba\_indexs where owner = ’EMP’

2. 显示索引列：显示 index\_test 所使用的索引列：

Select \* from user\_ind\_columns where index\_name = 'INDEX\_TEST'

3. 显示索引位置及大小

Select \* from user\_segments where segment\_name = 'INDEX\_TEST'

4. 显示函数索引

Select \* from user\_ind\_expressions where index\_name = 'INDEX\_TEST'

## 3.5 Oracle序列

**序列(Sequence)**是oracle提供的用于产生一系列唯一数字的数据库对象。由于oracle中没有设置自增列的方法，所以我们在oracle数据库中主要用序列来实现主键自增的功能。

### 3.5.1 序列的创建

1.序列创建的语法格式如下：

CREATE SEQUENCE sequence      *-- 创建序列名称*

[INCREMENT BY n]          *-- 递增的序列值是 n 如果 n 是正数就递增,如果是负数就递减 默认是 1*

[START WITH n]                     *-- 开始的值,递增默认是 minvalue 递减是 maxvalue*

[{MAXVALUE n | NOMAXVALUE}]        *-- 最大值*

[{MINVALUE n | NOMINVALUE}]        *-- 最小值*

[{CYCLE | NOCYCLE}]                *-- 循环/不循环*

[{CACHE n | NOCACHE}];             *-- 分配并存入到内存中*

2. 创建一个最小值为9，最大值为500，从10开始，步长为10的循环序列，每次加载50个值放入内存待使用。

create sequence SEQ\_CACHE1

increment by 10

start with 10

maxvalue 500

minvalue 9

cycle

cache 50;

3. 使用cache应注意两点

  a)序列cache可能造成序列值不连续：

  b)需保证 最大值-最小值 >= (缓存值-1)\*每次循环的值。

可以用 alter system flush shared\_pool 清空 cache 中缓存的序列值.

### 3.5.2 使用序列

序列创建后,可以使用序列的NEXTVAL来获取序列的下一个值,使用CURRVAL来查看当前值。第一次使用必须先使用NEXTVAL来产生一个值后才可以使用CURRVAL进行查看。

1. 序列调用 产生一个新的序列

select seq\_test.nextval from dual

2. 查看当前序列的值

select seq\_test.currval from dual

3. 序列在下列情况下会出现裂缝:

* 回滚
* 使用cache
* 多个表同时使用同一序列
* 系统异常

### 3.5.3 修改序列

1. 可以修改序列的增量, 最大值, 最小值, 循环选项, 或是否装入内存

alter SEQUENCE sequence            *-- 创建序列名称*

       [INCREMENT BY n]        *-- 递增的序列值是n 如果n是正数就递增,如果是负数就递减 默认是1*

       [START WITH n]           *-- 开始的值,递增默认是minvalue 递减是maxvalue`这里写代码片`*

       [{MAXVALUE n | NOMAXVALUE}] *-- 最大值*

       [{MINVALUE n | NOMINVALUE}] *-- 最小值*

       [{CYCLE | NOCYCLE}]         *-- 循环/不循环*

       [{CACHE n | NOCACHE}];      *-- 分配并存入到内存中*

例如：

alter sequence seqEmp maxvalue 5;

aLTER SEQUENCE emp\_sequence INCREMENT BY 10 MAXVALUE 10000 CYCLE *-- 到10000后从头开始 NOCACHE ;*

2. 修改序列的注意事项：

* 必须是序列的拥有者或对序列有 ALTER 权限
* 只有将来的序列值会被改变
* 改变序列的初始值只能通过删除序列之后重建序列的方法实现

3.5.4 删除序列

使用DROP SEQUENCE 语句删除序列，删除之后，序列不能再次被引用,如：

drop sequence t1\_seq;

### 3.5.5 查询序列

1. 通过数据字典USER\_OBJECTS可以查看用户拥有的序列。

select object\_name,object\_type from user\_objects;

2. 通过数据字典USER\_SEQUENCES可以查看序列的设置。

SELECT SEQUENCE\_NAME,MIN\_VALUE,MAX\_VALUE,INCREMENT\_BY,LAST\_NUMBER FROM USER\_SEQUENCES;

## 3.6 Oracle同义词

Oracle的**同义词（synonyms）**是数据库对象的一个别名，在使用同义词时，Oracle数据库将它翻译成对应方案对象的名字。与视图类似，同义词并不占用实际存储空间，只在数据字典中保存同义词的定义。在Oracle数据库中的大部分数据库对象，如表、视图、物化视图、序列、函数、存储过程、包、同义词等等，数据库管理员都可以根据实际情况为他们定义同义词。

Oracle同义词有两种类型，分别是Oracle公用同义词与Oracle私有同义词。普通用户创建的同义词一般都是私有同义词，公有同义词一般由DBA创建。

   a) **Oracle私有同义词**:一般由普通用户创建，跟公用同义词所对应，由创建他的用户所有，同义词的创建者，可以通过授权控制其他用户是否有权使用属于自己的私有同义词。

   b) **Oracle公用同义词**:一般由DBA创建，由一个特殊的用户组Public所拥有，数据库中所有的用户都可以使用公用同义词，往往用来标示一些比较普通的数据库对象，这些对象往往大家都需要引用。

同义词可以简化对象访问和提高对象访问的安全性。在多用户协同开发中，可以屏蔽对象的名字及其持有者，如果没有同义词，当操作其他用户的表时，必须通过user名.object名的形式，采用了Oracle同义词之后就可以隐蔽掉user名，同时也简化了SQL语句。

同义词还可以为分布式数据库的远程对象提供位置透明性。Oracle同义词在数据库链接中的作用：数据库链接是一个命名的对象，说明一个数据库到另一个数据库的路径，通过其可以实现不同数据库之间的通信。访问对象要通过 object名@数据库链名。同义词在数据库链中的作用就是提供位置透明性。

### 3.6.1 创建同义词

普通用法如下所示：

CREATE [OR **REPLACE**] [PUBLIC] SYNONYM [ schema.] 同义词名称 FOR [ schema.] object [ @dblink ];

1. 创建专有（私有）同义词

CREATE SYNONYM SYSN\_TEST FOR TEST;

CREATE SYNONYM SCOTT.EM FOR SOCTT.EMP;    *--创建其它模式的同义词，需要CREATE ANY SYNONYM权限*

2. 创建公共同义词

CREATE PUBLIC SYNONYM PUBLIC\_TEST FOR TEST;     *-- 公共同义词是和用户的schema无关的，但是公共的意思并不是所有的用户都可以访问它，必须被授权后才能进行*

3. 创建远程同义词

如果要创建一个远程的数据库上的某张表的同义词，需要先创建一个Database Link(数据库连接)来扩展访问：

Create database link 数据库链名 connect to user名 identified by 口令 using ‘Oracle连接串’;

然后再使用如下语句创建数据库同义词：

create synonym table\_name for table\_name@DB\_Link;

4. 同义词权限管理：

与同义词相关的权限有CREATE SYNONYM、CREATE ANY SYNONYM、CREATE PUBLIC SYNONYM权限。

a)用户在自己的模式下创建私有同义词，这个用户必须拥有CREATE SYNONYM权限，否则不能创建私有同义词。

b)如果需要在其它模式下创建同义词，则必须具有CREATE ANY SYNONYM的权限。

c)创建公有同义词则需要CREATE PUBLIC SYNONYM系统权限。

### 3.6.2 查看同义词

1. 查看系统所有同义词

       SELECT \* FROM DBA\_SYNONYMS

2. 查看用户同义词

       SELECT \* FROM USER\_SYNONYMS

3. 查看同义词状态

SELECT OBJECT\_NAME, STATUS  FROM ALL\_OBJECTS WHERE OBJECT\_NAME='T';

### 3.6.3 使用同义词

使用同义词可以保证当数据库的位置或对象名称发生改变时，应用程序的代码保持稳定不变，仅需要改变同义词；

当使用一个没有指定schema的同义词时，首先在用户自己的schema中寻找，然后再公共同义词中寻找。

       SELECT \* FROM SYSN\_TEST;

#### 3.6.4 删除同义词

语法格式如下：

DROP [ PUBLIC ] SYNONYM [ schema. ] 同义词名称 [ FORCE ];

例：

DROP SYNONYM SYSN\_TEST;

DROP PUBLIC SYNONYM PUBLIC\_TEST;   *--注意：当同义词的原对象被删除时，同义词并不会被级联删除*

### 3.6.5 编译同义词

对原对象进行DDL操作后，同义词的状态会变成INVALID；当再次引用这个同义词时，同义词会自动编译，状态会变成VALID，无需人工干预，当然前提是不改变原对象的名称

ALTER  SYNONYM T COMPILE; *--当同义词的原对象被重新建立时，同义词需要重新编译*

tips：

数据库对象、私有同义词、公有同义词同名现象

a) 不能创建和数据库对象同名的私有同义词

b) 数据库对象(或私有同义词)和公有同义词同名时，优先访问数据库对象(或私有同义词)

## 3.8 Oracle数据字典

数据字典是Oracle来管理和展现数据库信息，数据字典通常储存数据库的元数据，其用来描述用户数据，是数据库的“数据库”。通常说的数据字典由4部分组成：内部RDBMS（X$）表、数据字典表、动态性能视图（V$）和（静态）数据字典视图。（两表两视图）。

数据字典的相关信息也由表来保存，其中dictionary　保存了全部数据字典表的名称和解释，它有一个同义词dict；dict\_column保存了全部数据字典表里字段名称和解释，如

我们想查询跟索引有关的数据字典时，可以用下面这条SQL语句:

SQL>select \* from dictionary where **instr**(comments,'index')>0;

如果我们想知道user\_indexes表各字段名称的详细含义，可以用下面这条SQL语句:

SQL>select column\_name,comments from dict\_columns where table\_name='USER\_INDEXES';

依此类推，就可以轻松知道数据字典的详细名称和解释，不用查看ORACLE的其它文档资料了。

### 3.8.1 数据字典表

**数据字典表（Data dictionary table）**用以存储表、索引、约束以及其它数据库结构的信息，这些对象通常以“$”结尾（例如：TAB$、OBJ$、TS$等），在创建数据库的时候通过运行$ORACLE\_HOME/rdbms/admin/sql.bsq脚本来创建。

sql.bsq是一个非常重要的文件，其中包含了数据字典表的定义及注释说明，应仔细阅读研究。比如我们经常使用的V$DBA\_OBJECT视图就是基于V$OBJ创建的；再比如，当用户创建一张表的时候，Oracle将会在后台执行一系列的内部操作，比如向objV$表中插入数据、向tab$表中记录表数据等。

Oracle通过将DDL解析成DML操作，并且将这些操作记录在数据字典表中，那么我们当然可以通过反向解析得到原始的创建语句，从Oracle 9i 开始，一个新的工具包 DBMS\_METADATA 就可以完成该功能：

select dbms\_metadata.get\_ddl('TABLE','EMP') from dual;

### 3.8.2 静态数据字典视图

由于X$表和数据字典表通常不能直接被用户访问，Oracle创建了静态数据字典视图来提供用户对于数据字典信息的访问，由于这些信息通常相对稳定，不能直接修改，所以又被称为静态数据字典视图。静态数据字典视图在创建数据库时由$ORACLE\_HOME/rdbms/admin/catagory.sql脚本创建。

静态数据字典视图按照前缀的不同通常分成三类：

     ●**USER\_**：用户所拥有的相关对象信息；

**● ALL\_**：用于有权限访问的所有对象的信息；

     ● **DBA\_**：数据库所有相关对象的信息，用于需要有 select any table 的权限才能访问。

通过三类视图在本质上是为了实现权限控制。在Oracle数据库中，每个用户与方案（Schema）是对应的，Schema是用户所拥有的对象的集合。数据库通过Schema将不同用户的对象隔离开来，用户可以自由的访问自己的对象，但是要访问其他Schema对象就需要相关的授权。

常用的静态数据字典视图有下：

|  |  |
| --- | --- |
| 数据字典名称 | 说明 |
| dba\_tablespaces | 关于表空间的信息 |
| dba\_ts\_quotas | 所有用户表空间限额 |
| dba\_free\_space | 所有表空间中的自由分区 |
| dba\_segments | 描述数据库中所有段的存储空间 |
| dba\_extents | 数据库中所有分区的信息 |
| dba\_tables | 数据库中所有数据表的描述 |
| dba\_tab\_columns | 所有表、视图以及簇的列 |
| dba\_views | 数据库中所有视图的信息 |
| dba\_synonyms | 关于同义词的信息查询 |
| dba\_sequences | 所有用户序列信息 |
| dba\_constraints | 所有用户表的约束信息 |
| dba\_indexs | 关于数据库中所有索引的描述 |
| dba\_ind\_columns | 在所有表及簇上压缩索引的列 |
| dba\_triggers | 所有用户的触发器信息 |
| dba\_source | 所有用户存储过程信息 |
| dba\_data\_files | 查询关于数据库文件的信息 |
| dba\_tab\_grants/privs | 查询关于对象授权的信息 |
| dba\_objects | 数据库中所有的对象 |
| dba\_users | 关于数据库中所有用户的信息 |

### 3.8.3 内部RDBMS（X$）表

**X$**表示Oracle数据库的核心部分，这些表用于跟踪数据库内部信息，维持数据库的正常运行。X$表是加密命名的，而且Oracle不做文档说明。X$表是Oracle数据库的运行基础，在数据库启动时由Oracle应用程序动态创建。比如我们熟知的X$BH、X$KSMSP等。研究这些表的最好的办法就是借用Oracle的autotrace功能（**set** autotrace trace explain）。

### 3.8.4 动态性能视图

**动态性能视图**记录了数据库运行时信息和统计数据，大部分动态性能视图被实时更新以及反映数据库当前状态。在数据库启动时，Oracle动态创建X$表，在此基础上，Oracle创建了GV$和V$视图，GV$即Global V$，除了一些特例外，每个V$都对应一个GV$。GV$产生是为了OPS/RAC环境的需要，每个V$都是基于GV$的，只是GV$多了INST\_ID列来显示实例ID。

1. 如何查看一个视图的底层创建语句？

Oracle提供了一些特殊的视图用以记录其他视图的创建方式，v$fixed\_view\_definition就是其中之一。

 select \* from v$fixed\_view\_definition

2. 查询V$视图时，报的错却是V\_$视图不存在，V\_$视图不是基于V$视图创建的吗，怎么反向报错了？

为了防止普通用户的误操作，Oracle对通过软件机制对V$视图的访问做了限制，它不允许普通用户直接访问V$视图，但我们平时不是经常访问吗？这其实是Oracle引入了V\_$视图。在建立V$视图后，Oracle就建立了V\_$视图，随后为V\_$视图建立了与V$视图同名的共用同义词。这些工作都是通过catalog.sql脚本（该脚本位于$ORACLE\_HOME/rdbms/admin/目录下）实现的。

如：

     create or replace view **v\_**$fixed\_table as select \* from v$fixed\_table;

     create or replace public synonym v$fixed\_table for v\_$fixed\_table;    *--同名同义词*

也就是说，大部分用户访问的V$对象，并不是视图，而是执行V\_$视图的同义词；而V\_$视图时基于真正V$视图创建的。

常用的动态性能视图有下：

|  |  |
| --- | --- |
| 数据字典名称 | 说 明 |
| v$database | 描述关于数据库的相关信息 |
| v$datafile | 数据库使用的数据文件信息 |
| v$log | 从控制文件中提取有关重做日志组的信息 |
| v$logfile | 有关实例重置日志组文件名及其位置的信息 |
| v$archived\_log | 记录归档日志文件的基本信息 |
| v$archived\_dest | 记录归档日志文件的路径信息 |
| v$controlfile | 描述控制文件的相关信息 |
| v$instance | 记录实例的基本信息 |
| v$system\_parameter | 显示实例当前有效的参数信息 |
| v$sga | 显示实例的 SGA 区的大小 |
| v$sgastat | 统计 SGA 使用情况的信息 |
| v$parameter | 记录初始化参数文件中所有项的值 |
| v$lock | 通过访问数据库会话，设置对象锁的所有信息 |
| v$session | 有关会话的信息 |
| v$sqltext | 记录 SQL 语句的详细信息 |
| v$sqltext | 记录 SQL 语句的语句信息 |
| v$bgprocess | 显示后台进程信息 |
| v$process | 当前进程的信息 |

# 第四章 数据的增删改查

## 4.1 插入数据

1. 插入单行所有列

insert into DEPT(DEPTNO,DNAME,LOC)

    values(50, 'Public', 'changsha');

*-- 此时 (DEPTNO,DNAME,LOC) 部分可以省略*

2. 插入单行部分列

insert into DEPT(DEPTNO,DNAME)

values(60, null);  *-- DNAME为显式插入空值，LOC为隐式插入空值*

3. 插入多行数据(拷贝表)

insert into DEPT2

select \* from DEPT where DEPTNO = 20;

4. SQLPlus交互式插入数据

insert into DEPT(DEPTNO,DNAME,LOC)

values(&deptno,&dname, '默认');

## 4.2 删除数据

1.

delete [from] DEPT

    where DEPTNO >30;   *-- 删除表数据的限定条件*

2.

delete 也可以换成trunc，两者的区别是：

    delete：是DML语言，逐条删除，不释放空间，会产生碎片，可以回滚和闪回

**trunc**：是DDL语言，先摧毁再重建，会释放空间，不产生碎片，不可以回滚和闪回

## 4.3 修改数据

### 1.3.1 update

update DEPT

**set** DEPTNO = 70 , DNAME = 'HH', LOC = 'BeiJing'

    where DEPTNO = 80;

*-- 一定要不要忘了限定条件*

### 1.3.2 update set from

Oracle没有update **set** from语法，可以通过两种写法实现同样的功能：

1. 通过子查询实现

1)一次子查询更新一个列

UPDATE EMP1

**SET** SAL = (SELECT SAL+1 FROM EMP WHERE EMPNO = 7568)

    WHERE EMPNO = 7568

2) 一次子查询更新多个列

UPDATE EMP1

**SET** (SAL,COMM) = (SELECT SAL+1,**nvl**(COMM,0)+1 FROM EMP WHERE EMPNO = 7568)

    WHERE EMPNO = 7568

3) 一个sqlserver update **set** from改为oracle语法的例子：

UPDATE t

**SET** t.EXT\_SUBSYS\_SN = tt.SUBSYS\_SN

    FROM CUACCT\_LOG t,CUACCT tt

    WHERE t.CUACCT\_CODE = tt.CUACCT\_CODE

        AND t.SETT\_DATE = @SETT\_DATE

        AND t.BIZ\_NO = @BIZ\_NO

        AND tt.SUBSYS\_SN > 0;

改为：

UPDATE CUACCT\_LOG t

**SET** t.EXT\_SUBSYS\_SN = (select tt.SUBSYS\_SN from CUACCT tt WHERE t.CUACCT\_CODE = tt.CUACCT\_CODE and tt.SUBSYS\_SN > 0)

    WHERE t.SETT\_DATE = @SETT\_DATE

        AND t.BIZ\_NO = @BIZ\_NO

        AND exists (select 1 from CUACCT tt WHERE t.CUACCT\_CODE = tt.CUACCT\_CODE and tt.SUBSYS\_SN > 0);

2. 通过行列子集视图实现

UPDATE (SELECT E1.EMPNO,E1.SAL SAL1, E.EMPNO, E.SAL SAL2  FROM EMP E,EMP1 E1 WHERE E.EMPNO = E1.EMPNO AND E.EMPNO = 7568)

**SET** SAL1 = SAL2

注意：update只涉及一个表且视图列中包含了被更新的表的整个主键，否则不能更新视图的基表

## 4.4 查询数据

### 4.4.1 基本查询

1. 查询所有列

select \* from DEPT;

2. 查询部分列

select EMPNO,ENAME from EMP;

3. 去除重复列

select DISTINCT EMPNO,ENAME from EMP;

4. 给列起别名

select EMPNO 员工编号,ENAME 员工姓名from EMP;

*-- 如果别名中带有空格或是纯数字，则需要用双引号括起来*

5. 列参与运算

eg：查询员工的编号、姓名、薪水及奖金，并计算年收入

select EMPNO 员工编号,ENAME 姓名,SAL 薪水，COMM 奖金，(SAL+COMM)\*12 年收入

from EMP;

*-- 出现年收入为空的情况？*

*-- 因为COMM存在null，null不是空格或0，而是一个不可预知的值，参与运算会导致结果也为空*

*-- 解决方案：进行滤空修正 COMM 修改为 nvl(COMM,0)*

### 4.4.2 关联查询

1. 内连接(等值连接和非等值连接)

内连接（INNER JOIN）：分显式的和隐式的，返回连接表中符合连接条件和查询条件的数据行。

隐式的内连接没有INNER JOIN，形成的中间表为两个表的笛卡尔积。

select E.EMPNO 员工编号,E.ENAME 姓名,D.DNAME 部门名称

    from EMP E,DEPT D

    where E.DEPTNO = D.DEPTNO;

显式的内连接有INNER JOIN，形成的中间表为两个表经过ON条件过滤后的笛卡尔积。

select E.EMPNO 员工编号,E.ENAME 姓名,D.DNAME 部门名称

    from EMP E

    inner join DEPT D on E.DEPTNO = D.DEPTNO;

如果内连接的连接条件是=，则称为等值连接，否则称为非等值连接，如

eg: 查询员工表中的员工编号、姓名、薪水和薪水表中的薪水级别

select E.EMPNO 员工编号,E.ENAME 姓名,E.SAL 薪水, S.GRADE 薪水级别

    from EMP E,SALGRADE S

    where E.SAL between S.LOSAL and S.HISAL;

也可以写成

select E.EMPNO 员工编号,E.ENAME 姓名,E.SAL 薪水, S.GRADE 薪水级别

    from EMP E

inner join SALGRADE S on E.SAL between S.LOSAL and S.HISAL;

2. 外连接

外连接分为左外连接、右外连接以及全连接，其中左外连接表示除了返回自连接的结果外，还额外返回左表中不满足连接条件的行，此时结果集中这些行右表的数据为null.

eg: 查询部门表中的部门编号、部门名称和统计各部门人数（员工表）

select D.DEPTNO 部门编号, D.DNAME 部门名称, **count**(E.EMPNO) 部门人数

    from DEPT D,EMP E

    where D.DEPTNO = E.DEPTNO(+)

    group by D.DEPTNO,D.DNAME;

*--要想显示DEPT表中所有的部门编号，则要在=对面加上(+)，表示左外连接*

上述语句也可写为

select D.DEPTNO 部门编号, D.DNAME 部门名称, **count**(E.EMPNO) 部门人数

    from DEPT D

    left join EMP E on D.DEPTNO = E.DEPTNO

    group by D.DEPTNO,D.DNAME;

右外连接与左外连接类似，不同的是额外返回右表中不满足连接条件的行，全连接表示既返回左表又返回右表中不满足条件的行，关键字分别为right join和full join。

3. 自连接

一种特殊的连接，即对同一个表进行连接操作， 这个连接叫做自连接。

eg: 查询员工的老板信息

select E1.ENAME 员工姓名 || '的老板是' || **nvl**(E2.ENAME, '他自己')

    from EMP E1,EMP E2;

    where E1.MGR = E2.EMPNO(+);

*-- E1表中KING的MGR为空，不可能满足条件，因此要用外连接显示E1表中所有的行*

*-- 即使E1表中的所有行显示出来了，但KING的老板为空，应进行滤空修正*

### 4.4.3 子查询

1. 单行子查询与多行子查询

查询薪水大于SCOTT的员工编号和姓名

select EMPNO 员工编号, ENAME 员工姓名

    from EMP

    where SAL > (select SAL from EMP where ENAME = 'SCOTT');

*-- 单行子查询只能用单行比较操作符 (= < >)*

查询部门编号不是SALES或ACCOUNTING的员工编号和姓名

select EMPNO 员工编号, ENAME 员工姓名

    from EMP

    where DEPTNO not in (select DEPTNO from DEPT

                            where DNAME = 'SALES' or dname='ACCOUNTING');

*-- 多行子查询只能用多行比较操作符 (in any all)*

查询薪水 比30号部门 所有员工薪高的员工信息

select EMPNO 员工编号, ENAME 员工姓名

    from EMP

    where SAL > all(select SAL from EMP where DEPTNO = 30);

一切all都可以用>max替代，如上例只要大于30部门薪水的最大值即可

    where SAL > (select **max**(SAL) from EMP where DEPTNO = 30 );

查询薪水 比30号部门 任意一个员工薪高的员工信息

select EMPNO 员工编号, ENAME 员工姓名

    from EMP

    where SAL > any( select SAL from EMP where DEPTNO = 30 );

一切any都可以用> **min** 替代，如上例只要大于30部门薪水的最小值即可

    where SAL > (

        select **min**(SAL)

            from EMP

            where DEPTNO = 30

        );

2. 相关子查询与非相关子查询

上述例子都是非相关子查询，即子查询返回值给主查询使用，另一种子查询叫相关子查询，即主查询 通过别名  把主查询的值 传递给 子查询，每一行处理的过程分为三步解析：

    1) 主查询传字段值值给子查询

    2) 子查询使用主表字段

3) 主查询使用子查询的返回结果

EG:查找 员工表中 薪水 大于 本部门平均薪水 的员工

select E.EMPNO 员工编号,E.ENAME 员工姓名,E.SAL 薪水,

( select **avg**(sal) from EMP where DEPTNO = E.DEPTNO ) 平均薪水

    from EMP E              *--1 主查询中：给EMP表起个别名，方便传值*

    where SAL > (

        select **avg**(SAL) from EMP

            where DEPTNO = E.DEPTNO );  *--2 子查询中：引用主查询中的EMP表*

*--3 子查询返回结果后主查询继续执行*

相关子查询难以理解，可以改写为 一般子查询 + 多表查询

select E.EMPNO 员工编号,E.ENAME 员工姓名,E.SAL 薪水,D.AVGSAL

    from EMP E, (select DEPTNO, **avg**(SAL) from EMP

                    group by DEPTNO ) D

    where E.DEPTNO = D.DEPTNO and E.SAL > D.AVGSAL

3. 使用子查询的一些注意事项：

    1) 不能在order by和group by后放子查询，也不能在select语句后放多行子查询，但from、where、having后都可接子查询

    2) 子查询有时可以用多表查询替代，并且效率更高，如

    EG:查询部门是SALES的员工信息：

select EMPNO 员工编号, ENAME 员工姓名

    from EMP

    where DEPTNO = (

        select DEPTNO

            from DEPT

            where DNAME = 'SALES');

可以替换为：

select EMPNO 员工编号, ENAME 员工姓名

    from EMP,DEPT

    where EMP.DEPTNO = DEPT.DEPTNO and DEPT.DNAME = 'SALES';

4. 子查询应用：TOP-N问题

EG:查询薪水排行前三的员工信息

select rownum, EMPNO,  ENAME, SAL

from ( select EMPNO, ENAME, SAL

            from EMP

            order by SAL desc)

where rownum<=3;

*--rownum是oracle的伪列，*

注意：

    1) TOP-N问题如果需要排序，应采用两层结构，因为rownum是属于服务器返回的结果集的固有属性，排序会打乱rownum的顺序

    2) rownum只能使用where rownum<=3 ，而不能使用  where rownum>=3，因为没有产生第1行、第2行，就不可能出现满足条件的第3行、第4行

    3) 与rownum类似的还有结果集的行地址rowid，用法如下：

        select rowid, empno, ename from emp;

        rowid相当于一个门牌地址,是一个指针;在索引中存放的就是行地址

5. 子查询应用：分页问题

EG:查询薪水排行为5-8的员工信息

select   r, EMPNO, ENAME, SAL

    from (select rownum r, EMPNO,  ENAME, SAL

            from (select EMPNO, ENAME, SAL

                    from EMP

                    order by SAL desc)

            where rownum<=8)

    where r>=5;

分页问题解决思路：

    内层：排序

    中层：使用rownum选择前n条；并给rownum指定一个别名，以供最外层过滤使用

    外层：去掉前m条结果

### 4.4.4 分组查询与聚合函数

1. 依据单列分组

eg：求出EMP表中各部门的平均工资

select DEPTNO 部门编号, **avg**(SAL)

    from EMP

    group  by DEPTNO;

2. 依据多列分组

eg：按部门 不同的职位 统计平均工资

select DEPTNO 部门编号,JOB 职位，**avg**(SAL)

    from EMP

    group  by DEPTNO,JOB

    order by 1;

3. 分组过滤(having)

eg：查询平均工资大于2000的部门

select DEPTNO, **avg**(SAL)

    from EMP

    group by DEPTNO

    having **avg**(SAL) > 2000;

*-- 分组过滤语句having与where语句的区别：where语句效率高，但是不能使用组函数*

4. 常用的聚合函数

**1)MAX** **MIN** **AVG** **SUM** 函数

EG:求员工表中 最高工资,最低工资 平均工资 所有工资

select **max**(SAL) 最高工资,**min**(SAL) 最低工资, **avg**(SAL) 平均工资,**sum**(SAL)总工资

    from EMP;

2)count函数

EG: 返查询员工表中的经理总人数(去除重复)

select **count**(distinct MGR) 经理总人数

    from EMP;

*-- 组函数默认忽略空值，可手动调用滤空函数来处理*

### 4.4.5 查询结果过滤

1. 空值过滤

查询奖金为空/不为空的员工编号和姓名

select EMPNO 员工编号, ENAME 员工姓名, COMM 奖金

from EMP

where COMM is NULL;

*--where COMM is not NULL;*

2. 离散过滤

1) 查询部门编号是/不是20的员工编号和姓名

select EMPNO 员工编号, ENAME 员工姓名, DEPTNO 部门编号

from EMP

where DEPTNO = 20;

*--where DEPTNO != 20;*

2) 查询部门编号是/不是20或30的员工编号和姓名

select EMPNO 员工编号, ENAME 员工姓名, DEPTNO 部门编号

from EMP

where DEPTNO in (20,30);

*--where DEPTNO not in (20,30);*

*-- in后带有null将会被忽略，而not in后带有null，会导致表达式始终不成立*

3. 连续过滤

查询薪水在/不在1000到2000之间的员工编号和姓名

select EMPNO 员工编号, ENAME 员工姓名, SAL 薪水

from EMP

where SAL between 1000 and 2000;    *-- between and是双闭区间*

*--where SAL >=1000 and SAL <=2000;*

where SAL not between 1000 and 2000; *-- not between and是双开区间*

*--where SAL < 1000 OR SAL > 2000;*

4. 模糊过滤

查询名字以S开头的员工编号和姓名

select EMPNO 员工编号，ENAME 姓名

from EMP

where ENAME like ‘S%’;

查询名字是四个字母的员工编号和姓名

select EMPNO 员工编号，ENAME 姓名

from EMP

where ENAME like ‘\_\_\_\_’;

查询名字含有下划线的员工编号和姓名 (查询的内容含有转义字符)

select EMPNO 员工编号，ENAME 姓名

  from EMP

  where ENAME like ‘%\\_%’ escape ‘\’;

*-- escape ‘\’为转义声明*

### 4.4.6 查询结果排序

1. 依据单列排序

查询员工的编号、姓名和薪水，并根据薪水升序

select EMPNO 员工编号，ENAME 姓名，SAL 薪水

    from EMP

    order by SAL asc;           *-- asc可以省略*

    #order by 薪水 asc;           *-- 用别名来代替列名排序*

    #order by 3 asc;            *-- 用序号代替列名排序*

    #order by SAL\*12 asc;       *-- 列名也可以进行运算*

查询员工的编号、姓名和奖金，并根据奖金降序

select EMPNO 员工编号，ENAME 姓名，COMM 奖金

    from EMP

    order by COMM desc

nulls **last**;                 *-- 排序列为空的行置后*

2. 依据多列排序

查询员工的编号、姓名、部门编号和薪水，先根据部门编号升序，在根据薪水降序

select EMPNO 员工编号，ENAME 姓名，DEPTNO 部门编号，SAL 薪水

    from EMP

    order by DEPTNO asc,SAL desc;

## 4.5 集合运算

1. union：返回两个集合去掉重复元素后的所有记录

eg：查询E1和E2表中的所有员工信息，去掉重复项

select EMPNO, JOB

    from EMP1

union

select EMP, JOB

    from EMP2;

*-- union all:不去重复项*

2. intersect: 运算符返回同时属于两个集合的记录

EG: 显示薪水同时位于级别1（700~1300）和级别2（1201~1400）的员工信息

select ENAME,SAL

    from EMP

    where SAL between 700 and 1300

intersect

select ENAME,SAL

    from EMP

    where SAL between 1201 and 1400;

3. minus:返回属于第一个集合，但不属于第二个集合的记录

EG: 显示薪水同时位于级别1（700~1300），但不属于级别2（1201~1400）的员工信息

select ENAME,SAL

    from EMP

    where SAL between 700 and 1300

minus

select ENAME,SAL

    from emp

    where SAL between 1201 and 1400;

4. 集合运算的注意事项

    1) select语句中参数类型和个数要一致。

    2) 可以使用括号改变集合执行的顺序

    3) 如果有order by子句，必须放到每一句查询语句后

    4) 集合运算采用第一个语句的表头作为表头

# 第五章 内置函数与表达式

## 5.1 字符类函数

1. 大小写转换

select **lower**('HeLlo wOrld') 全小写,

**upper**('hello world') 全大写,

**initcap**('hello woRld') 首字母大写

  from dual;

2. 字符串连接

select **concat**('aaaa', **concat**('bbbb', 'cccc')) 字符串连接

  from dual;

也可以用连接符||来实现

select 'aaaa' || 'bbbb' || 'cccc' 字符串连接

  from dual;

3. 子串截取

select **substr**('abcd黄原鑫hijklmn', 3, 10) 子串截取

  from dual;

*-- 从3开始，截取10个字符，参数2可以缺省*

4. 子串查找

select **instr**('aaaabbbbb', 'ab')

  from dual;

*-- 返回参数1中ab首次出现的位置，或0*

5. 子串替换

select **replace**('abcd', 'bc', 'cb')

  from dual;

*-- 把参数1中的参数2替换为参数3*

6. 字符长度（汉字占一个字符）

select **length**('中国abc') 字符数

  from dual;

7. 字节长度（汉字占两个字节）

select lengthb('中国abc') 字节数

  from dual;

8. 字符串填充

select **lpad**('abcd', 10, '\*') 左填充 , **rpad**('abcd', 10, '\*') 右填充

  from dual

*-- 参数1：字符串  参数2：总长度  参数3：填充值*

9. 字符串修剪

select **trim**('H' from 'HellowroHldHHHHH')

  from dual ;

*--两端修剪执行结果为：ellowroHld*

## 5.2 数值类函数

1. 数值四舍五入

select **round**(45.923,2) as "2", **round**(45.923,0) as "0", **round**(45.925,-1) as "-1"

  from dual;

*--参数2：表示舍入位数，执行结果为[ 45.92 46 50]*

2. 数值截断

select **trunc**(45.923,2) "2",**trunc**(45.923,0) "0",**trunc**(45.925,-1) "-1"

  from dual;

*--参数2：表示舍入位数，执行结果为[ 45.92 45 40]*

3. 数值求余

select **mod**(10,3)

from dual ;

*-- 表示10%3，执行结果为1*

## 日期类函数

1. 两个日期相差的月数(精确月)

select **months\_between**(**sysdate**,HIREDATE)

  from EMP;

2. 向指定日期加上指定月数

select **add\_months**(**sysdate**,1) 加1月

  from dual;

3. 指定日期的下一个日期

select **next\_day**(**sysdate**,'星期五')

  from dual;

4. 本月最后一天

select **last\_day**(**sysdate**) 本月最后一天

  from dual;

5. 日期四舍五入

select **round**(**sysdate**,'month')   *-- 日期按月四舍五入*

  from dual;

select **round**(**sysdate**,'year')  *-- 日期按年四舍五入*

  from dual;

6. 日期截断

select **trunc**(**sysdate**,'month') 日期按月四舍截断

  from dual;

select **trunc** (**sysdate**,'year') 日期按年四舍截断

  from dual;

7. 日期的数学运算

select (**sysdate** – HIREDATE)/7 距今周数

  from EMP;

*-- 在Oracle中，日期和整数可以直接运算，整数的单位为天*

*--            日期 – 日期得到相差的天数*

**说明：** 日期列的默认显示格式是DD-MON-RR，而RR与YY有一定区别，如下：



**设置会话的日期结果显示**

中文日期格式：alter session SET nls\_language=’SIMPLIFIED CHINESE’;

英文日期格式：alter session SET nls\_language=’AMERICAN’;

指定日期格式：alter session SET nls\_date\_format=’YYYY-MM-DD’;

## 5.4 类型转换函数

1. 转换到字符串(从时间)

**TO\_CHAR**(date, 'format\_model')

date：可以选择 **sysdate** **SYSTIMESTAMP** 等

'format\_model'：可以是以下的任意组合

    YYYY  MM  DY/DAY  YEAR            MONTH   DD  HH24 MI:SS.FF3  [AM]

    2019  04  星期一  twenty eleven   4月     21  19:20:05.301     [PM]

*-- 使用双引号可以在时间和日期中添加字符串，如*

  DD “of” MONTH 12 of OCTOBER

2. 转换到字符串(从数值)

**TO\_CHAR**(number, 'format\_model')

3. 转换到日期(从字符串)

**TO\_DATE**(char[, 'format\_model'])

4. 转换到数值(从字符串)

**TO\_NUMBER**(char[, 'format\_model'])

*-- 关于货币的格式化*

    9         0         $     L       .       ,

    整数占位  小数占位   美元  本地货币 小数点 千位符

## 5.5 通用函数

1. 滤空函数

**NVL** (expr1, expr2)

*-- 若expr1为空值，则转换为expr2（日期、数字、字符串）*

**NVL2** (expr1, expr2, expr3)

*-- 若expr1为空值，转换为expr3，否则转换为expr2*

**COALESCE** (expr1, expr2, …, exprn)

*-- 返回参数列表中第一个不为空的expr*

2. 等值判断函数

**NULLIF** (expr1, expr2)

*-- 若expr1与expr2相等，则返回NULL，不等返回expr1*

## 5.6 表达式

1. SQL99表达式

select EMPNO 员工编号, ENAME 姓名, SAL 涨前薪水,

    case JOB when  ‘IT\_PROG’    then      1.1\*SAL

             when ‘ST\_CLERK’    then      1.2 \*SAL

             when ‘SA\_REP’      then      1.5\*SAL

    else                                  SAL

    end  涨后薪水

  from EMP;

2. Oracle的DECODE 函数

select EMPNO 员工编号, ENAME 姓名, SAL 涨前薪水,

**decode**(JOB, ‘IT\_PROG’,  1.10\*SAL,

                      ‘ST\_CLERK’, 1.15\*SAL,

                      ‘SA\_REP’,   1.20\*SAL,

              SAL) 涨后薪水

from   EMP;

# 第七章 PLSQL语法

**PL/SQL**是一种程序语言，叫做过程化SQL语言(PL/SQL是面向过程语言)，是Oracle数据库对SQL语句的扩展，在普通SQL语句的使用上增加了编程语言的特点(PL/SQL是对SQL语言的扩展)。

    PL/SQL是Oracle系统的核心语言，现在Oracle的许多部件都是由PL/SQL写成的,具有简单、高效、灵活、实用的特点。

## 7.1 PL/SQL的语法结构

### 7.1.1 PL/SQL块

**块(block)**是PL/SQL的基本程序处理单元，由三部分组成，分别是声明部分、执行部分以及异常处理部分。

    DECLARE

*--声明部分：声明变量、常量、复杂数据类型、游标等*

    BEGIN

*-- 执行部分：PL/SQL语句和SQL语句*

    EXCEPTION

*-- 异常处理部分：处理运行错误*

    END;

### 7.1.2 第一个PL/SQL程序

例：打印hello word！

    SQL> **SET** SERVEROUTPUT ON

    SQL> BEGIN

      2     **DBMS\_OUTPUT.PUT\_LINE**('HELLO WORD');

      3  END;

      4  /

HELLO WORD

### 7.1.3 PL/SQL分类

    1) 匿名块：动态构造、只能执行一次。

    2) 子程序：存储在数据库中的存储过程、函数及包等。当在数据库上建立好后可以在其它程序中调用它们。

3) 触发器：当数据库发生操作时，会触发一些事件，从而自动执行相应的程序。

## 7.2 PL/SQL的变量

### 7.2.1 PL/SQL标识符

当编写PL/SQL块时，为了临时存储数据，需要定义变量和常量。变量和常量的定义是需要满足标识符的限制要求的：

* + 标识符名不能超过30个字符
  + 第一个字符必须为字母
  + 不分大小写
  + 不能用减号 "-"

*-- 注意：尽量不把变量名声明和表中字段名一样。*

### 7.2.2 变量的类型

    1) 数值类型：NUMBER(p,s)以及子类型INT、FLOAT 等

    2) 字符类型：CHAR(n)、VARCHAR2(n) 等

    3) 日期类型：DATE 等

    4) 布尔类型：BOOLEAN

### 7.2.3变量的命名规范

为了提高代码的可读性，建议遵从以下编码规则：

    标识符     命名规则        例子

    程序变量    v\_name          v\_sal

    程序常量    c\_name          c\_pi

    游标变量    name\_curror     emp\_curror

    异常标识    e\_name          e\_integrity\_error

    记录类型    name\_record     emp\_record

### 7.2.4 变量的大小写规范

当编写sql语句和PL/SQL语句时，既可以采用大写格式，也可以采用小写格式。但是为了程序的可读性，应尽量按照以下规则：

* SQL关键字采用大写格式，如：SELECT，UPDATE等
* PL/SQL关键字采用大写格式，如：DECLARE，BEGIN，END等
* 数据类型采用大写格式，如：INT、DATE等
* 标识符和参数采用小写格式，如：v\_sal等
* 数据库对象和列采用小写格式，如：emp，sal等

### 7.2.5 代码缩进/注释

    单行注释：*--注释内容*

    多行注释：*/\*注释内容\*/*

### 7.2.5 引用型变量与记录型变量

在许多情况下，PL/SQL变量可以用来存储在数据库表中的数据。在这种情况下，变量应该拥有与表列相同的类型，如：

    SQL> DECLARE

      2     v\_name varchar(10);

      3     v\_sal  number(7,2);

      4  BEGIN

      5     SELECT ename, sal INTO v\_name, v\_sal FROM emp WHERE empno = 7788;

      6     *-- 打印姓名和薪水*

      7     **DBMS\_OUTPUT.PUT\_LINE**(v\_name || '的工资是：' || v\_sal);

      8  END;

      9  /

    SCOTT的工资是：3000

引用型变量是指其数据类型与已经定义的某个数据变量的类型相同，或者与数据库表的某个列的数据类型相同。例：

    SQL> DECLARE

      2     v\_name emp.ename%TYPE;

      3     v\_sal emp.sal%TYPE;

      4  BEGIN

      5     SELECT  ename, sal

      6       INTO  v\_name,v\_sal

      7       FROM  emp

      8       WHERE empno = 7788;

      9     **DBMS\_OUTPUT.PUT\_LINE**(v\_name || '的工资是：' || v\_sal);

     10  END;

     11  /

    SCOTT的工资是：3000

记录型变量：PL/SQL提供%ROWTYPE操作符，返回一个记录类型，其数据类型和数据库表的数据结构相一致。举例：

SQL> DECLARE

  2  emp\_record emp%ROWTYPE;

  3  BEGIN

  4  SELECT \* INTO emp\_record FROM emp WHERE empno = 7788;

  5  **DBMS\_OUTPUT.PUT\_LINE**(emp\_record.ename||'的工资是：'||emp\_record.sal);

  6  END;

  7  /

SCOTT的工资是：3000

## 7.3 PL/SQL运算符

### 7.3.1 算术运算符

SQL> DECLARE

  2    v\_num1 NUMBER(3) := 10;

  3    v\_num2 NUMBER(3) := 2;

  4  BEGIN

  5    **DBMS\_OUTPUT.PUT\_LINE**('加法：' || (v\_num1 +  v\_num2));

  6    **DBMS\_OUTPUT.PUT\_LINE**('减法：' || (v\_num1 -  v\_num2));

  7    **DBMS\_OUTPUT.PUT\_LINE**('乘法：' || (v\_num1 \*  v\_num2));

  8    **DBMS\_OUTPUT.PUT\_LINE**('除法：' || (v\_num1 /  v\_num2));

  9    **DBMS\_OUTPUT.PUT\_LINE**('乘方：' || (v\_num1 \*\* v\_num2));

 10  END;

 11  /

加法：12

减法：8

乘法：20

除法：5

乘方：100

### 7.3.2 关系运算符

    SQL> DECLARE

      2    v\_num1 NUMBER(2) := &n1;

      3    v\_num2 NUMBER(2) := &n2;

      4  BEGIN

      5    IF(v\_num1 != v\_num2) THEN

      6      **DBMS\_OUTPUT.PUT\_LINE**('num1 != num2');

      7    ELSIF (v\_num1 > v\_num2) THEN

      8      **DBMS\_OUTPUT.PUT\_LINE**('num1 > num2');

      9    END IF;

     10  END;

     11  /

    输入 n1 的值:  10

    原值    2:   v\_num1 NUMBER(2) := &n1;

    新值    2:   v\_num1 NUMBER(2) := 10;

    输入 n2 的值:  5

    原值    3:   v\_num2 NUMBER(2) := &n2;

    新值    3:   v\_num2 NUMBER(2) := 5;

    num1 != num2

*--其它的关系运算符还有：=  <>  ~=  ^=  >=  <=*

### 7.3.3 比较运算符

SQL> DECLARE

  2    v\_num1 NUMBER(2) := &n1;

  3  BEGIN

  4    IF(v\_num1 BETWEEN 5 AND 10) THEN

  5      **DBMS\_OUTPUT.PUT\_LINE**('num1在5到10之间');

  6    ELSE

  7      **DBMS\_OUTPUT.PUT\_LINE**('num1不在5到10之间');

  8    END IF;

  9

 10    IF(v\_num1 NOT IN(3,8)) THEN

 11      **DBMS\_OUTPUT.PUT\_LINE**('num1不为3或8');

 12    END IF;

 13

 14    IF(v\_num1 IS NOT NULL) THEN

 15      **DBMS\_OUTPUT.PUT\_LINE**('num1不为NULL');

 16    END IF;

 17  END;

 18  /

输入 n1 的值:  4

原值    2:   v\_num1 NUMBER(2) := &n1;

新值    2:   v\_num1 NUMBER(2) := 4;

num1不在5到10之间

num1不为3或8

num1不为NULL

### 7.3.4 逻辑运算符

SQL> DECLARE

  2    v\_b1 BOOLEAN := &n1;

  3    v\_b2 BOOLEAN := &n2;

  4  BEGIN

  5    IF(v\_b1 AND v\_b2) THEN

  6      **DBMS\_OUTPUT.PUT\_LINE**('AND-TURE');

  7    END IF;

  8

  9    IF(v\_b1 OR v\_b2) THEN

 10      **DBMS\_OUTPUT.PUT\_LINE**('OR-TURE');

 11    END IF;

 12

 13    IF( NOT v\_b1) THEN

 14      **DBMS\_OUTPUT.PUT\_LINE**('FALSE-TURE');

 15    END IF;

 16  END;

 17  /

输入 n1 的值:  TRUE

原值    2:   v\_b1 BOOLEAN := &n1;

新值    2:   v\_b1 BOOLEAN := TRUE;

输入 n2 的值:  FALSE

原值    3:   v\_b2 BOOLEAN := &n2;

新值    3:   v\_b2 BOOLEAN := FALSE;

OR-TURE

*-- 注意：逻辑运算符只可以取TURE、FALSE或NULL，*

### 7.3.5 NULL的运算规则

1) NULL与算术运算符

如果算术表达式的任一输入为空，则该算术表达式（涉及诸如 +、-、\* 或 / 的算术运算）结果为空

2) NULL与关系运算符

SQL将涉及空值的任何比较运算的结果视为unknown（既不是谓词is null，也不是is not null）。unknown是在SQL中除true和false之外的第三个逻辑值。特别的，"null = null"会返回unknown，而不是true。

3) NULL与逻辑运算符

    And：true and unknown = unknown， false and unknown = false， unknown and unknown = unknown

or：

true or unknown = true， false or unknown = unknown， unknown or unknown = unknown

not：

not unknown = unknown

4) NULL与比较运算符

   in后带有null将会被忽略，而not in后带有null，会导致表达式始终不成立

5) NULL与集合运算

在集合运算中，{('A', null), ('A', null)}中的两个元素会被认为是相同的元素，因此在使用distinct子句只会保留这样的相同的元组的一份拷贝。

6) NULL与字符串连接

NULL||字符串 -> 字符串

## 7.4 PL/SQL的流程控制

### 7.4.1 IF语句

1) 简单IF语句：IF...THEN... END IF

例：新入员工号，判断员工工资，显示工资小于3000的员工姓名及工资

    DECLARE

      v\_name emp.ename%TYPE;

      v\_sal  emp.sal%TYPE;

    BEGIN

      SELECT  ename,sal

        INTO  v\_name,v\_sal

        FROM  emp

        WHERE empno=&no;

      IF v\_sal < 3000 THEN

**DBMS\_OUTPUT.PUT\_LINE**(v\_name || '的工资是：'|| v\_sal);

    END IF;

    END;

2) 二重IF语句：IF... THEN... ELSE... END IF

例：输入员工号，判断员工工资将工资小于3000的员工工资涨200，并显示涨工资的员工姓名，其他员工显示员工姓名及工资。

    DECLARE

      v\_name  emp.ename%TYPE;

      v\_sal   emp.sal%TYPE;

      v\_empno emp.empno%TYPE := &no;

    BEGIN

      SELECT  ename, sal

        INTO  v\_name, v\_sal

        FROM  emp

        WHERE empno=v\_empno;

      IF v\_sal < 3000 THEN

        UPDATE emp **SET** sal = sal + 200 WHERE empno = v\_empno;

**COMMIT**;

**DBMS\_OUTPUT.PUT\_LINE**(v\_name || '涨工资了');

      ELSE

**DBMS\_OUTPUT.PUT\_LINE**(v\_name || '的工资是：' || v\_sal);

      END IF;

    END;

3) 三重IF语句：  IF... THEN... ELSIF... THEN... ELSE... END IF

例3：输入员工号，判断员工工资工资小于2000，显示低收入，工资小于6000，显示中等收入，其它显示高收入

DECLARE

  v\_name emp.ename%TYPE;

  v\_sal  emp.sal%TYPE;

BEGIN

  SELECT ename,sal INTO v\_name, v\_sal FROM emp WHERE empno = &no;

  IF v\_sal < 2000 THEN

**DBMS\_OUTPUT.PUT\_LINE**(v\_name || '的工资是：' || v\_sal || '属于低收入');

  ELSIF v\_sal < 6000 THEN

**DBMS\_OUTPUT.PUT\_LINE**(v\_name || '的工资是：' || v\_sal || '属于中收入');

  ELSE

**DBMS\_OUTPUT.PUT\_LINE**(v\_name || '的工资是：' || v\_sal || '属于高收入');

  END IF;

END;

### 7.4.2 CASE语句

1) 等值比较

例：输入成级等级，判断属于哪个层次，并打印输出。

DECLARE

 v\_grade CHAR(1) := &n;

BEGIN

  CASE v\_grade

    WHEN 'A' THEN

**DBMS\_OUTPUT.PUT\_LINE**('优秀');

    WHEN 'B' THEN

**DBMS\_OUTPUT.PUT\_LINE**('中等');

    WHEN 'C' THEN

**DBMS\_OUTPUT.PUT\_LINE**('一般');

    ELSE

**DBMS\_OUTPUT.PUT\_LINE**('输入有误');

    END CASE;

END;

*--注意输入字符要带单引号*

2) 非等值比较

例：输入员工号，获取员工工资，判断工资，如果工资小于1500，补助加100，如果工资小于2500，补助加80，如果工资小于5000，补助加50.

DECLARE

  v\_sal emp.sal%TYPE;

  v\_empno emp.empno%TYPE := &no;

BEGIN

  SELECT sal INTO v\_sal FROM emp WHERE empno = v\_empno;

  CASE

    WHEN v\_sal < 1500 THEN

      UPDATE emp **SET** comm = **nvl**(comm,0) + 100 WHERE empno = v\_empno;

    WHEN v\_sal < 2500 THEN

      UPDATE emp **SET** comm = **nvl**(comm,0) + 80 WHERE empno = v\_empno;

    WHEN v\_sal < 5000 THEN

      UPDATE emp **SET** comm = **nvl**(comm,0) + 50 WHERE empno = v\_empno;

**COMMIT**;

    END CASE;

END;

### 7.4.3 循环语句

1) LOOP循环

例：打印1~10

DECLARE

 v\_cnt INT := 1;

BEGIN

  LOOP

**DBMS\_OUTPUT.PUT\_LINE**(v\_cnt);

    EXIT WHEN v\_cnt = 10;

    v\_cnt := v\_cnt + 1;

  END LOOP;

END;

2) WHILE循环

例：打印1~10

DECLARE

  v\_cnt INT := 1;

BEGIN

  WHILE v\_cnt <= 10 LOOP

**DBMS\_OUTPUT.PUT\_LINE**(v\_cnt);

    v\_cnt := v\_cnt + 1;

  END LOOP;

END;

3) FOR... IN...循环

例：打印1~10

BEGIN

  FOR i IN 1..10 LOOP

**DBMS\_OUTPUT.PUT\_LINE**(i);

  END LOOP;

END;

4) 嵌套循环与标号

嵌套循环是指在一个循环语句中嵌入另一个循环语句,标号用于标记嵌套块或嵌套循环,使用可以<<label\_name>>定义标号

DECLARE

 v\_result INT;

BEGIN

  <<outter>> FOR i IN 1..5 LOOP

    <<inter>> FOR j IN 1..5 LOOP

      v\_result := i;

      EXIT outter WHEN i = 4;

      END LOOP inter;

**DBMS\_OUTPUT.PUT\_LINE**('内:' || v\_result);

  END LOOP outter;

**DBMS\_OUTPUT.PUT\_LINE**('外:' || v\_result);

END;

写成C语言如下：

for(int i = 1;i <5; i++)

{

   for(int j= 1;j < 5;j++)

   {

       v\_result = i;

       if (i == 4) goto END;

   }

   printf("内：%d", v\_result);

}

END：

   printf("外：%d", v\_result);

5)循环退出

    EXIT语句用于直接退出当前循环

    EXIT WHEN语句用于在满足特定条件时退出当前循环

    CONTINUE语句是oracle11g的新特性，用于直接结束当前循环并继续下一组循环。

    CONTINUE WHEN语句用于在满足特定条件时结束当前循环语句并继续下一组循环语句。

### 7.4.4 其它流程控制语句

1) GOTO语句

GOTO语句用于跳转到特定标号处执行语句。语法格式：GOTO label\_name；

注意：当使用GOTO跳转到特定标号时，标号后至少要包含一条执行语句。

    DECLARE

      v\_cnt INT := 1;

    BEGIN

      LOOP

**DBMS\_OUTPUT.PUT\_LINE**(v\_cnt);

        IF v\_cnt = 10 THEN

*--EXIT：*

        GOTO end\_loop;

        END IF;

        v\_cnt := v\_cnt + 1;

      END LOOP;

    <<end\_loop>>

      NULL;

END;

2) NULL语句

    NULL语句不会执行任何操作，并且会直接将控制传递到下一个语句，使用该语句的主要目的是提高PL/SQL块的可读性。

## 7.5 游标

**游标**是SQL的一个内存工作区，由系统或用户以变量的形式定义。游标的作用就是用于临时存储从数据库中提取的数据块，通俗的来讲游标就是一个结果集。

### 7.5.1 显示游标

**显示游标**是用户自定义的，显示创建的游标，主要是用于对查询语句的处理。显示游标的使用一般分为定义游标 -> 打开游标 -> 提取数据 -> 关闭游标四个步骤。语法如下：

例：查询所有员工的员工号、姓名和职位的信息。

    DECLARE

*--1 定义游标*

        CURSOR emp\_cursor IS SELECT empno,ename,job FROM emp;

        v\_empno emp.empno%TYPE;

        v\_ename emp.ename%TYPE;

        v\_job   emp.job%TYPE;

    BEGIN

*--2 打开游标，执行查询*

        OPEN emp\_cursor;

*--3 提取数据*

        LOOP

        FETCH emp\_cursor INTO v\_empno,v\_ename,v\_job;

**DBMS\_OUTPUT.PUT\_LINE**('员工号：'||v\_empno||'姓名:'||v\_ename||'职位：'||v\_job);

*-- 在适时退出数据提取*

        EXIT WHEN emp\_cursor**%NOTFOUND**;

        END LOOP;

*--4 关闭游标*

        CLOSE emp\_cursor;

    END;

### 7.5.2  游标属性

1) %FOUND：该属性用于检测游标结果集是否存在数据，如果存在数据，返回TRUE.

2) %NOTFOUND：该属性用于检测游标结果集是否不存在数据，如果不存在数据，返回TRUE.

3) %ISOPEN：该属性用于检测游标是否已经打开，如果已经打开返回TRUE.

4) %ROWCOUNT：该属性用于返回已提取的实际行数。

例1：上例的退出语句还可以写成如下：

    EXIT WHEN NOT emp cursor**%FOUND**

    或者提取5行后退出：

    EXIT WHEN emp\_cursor**%ROWCOUNT** = 5;

例2：检测游标是否打开可用下面语句：

  IF emp\_cursor**%ISOPEN** THEN

**DBMS\_OUTPUT.PUT\_LINE**('游标已打开!');

  ELSE

**DBMS\_OUTPUT.PUT\_LINE**('游标未打开!');

  END IF;

### 7.5.3 游标FOR循环

当使用**游标FOR循环**时，Oracle会隐含地打开游标，提取数据并关闭游标。

如上例可以改写为以下：

    DECLARE

      CURSOR emp\_cursor IS SELECT empno, ename, job FROM emp;

    BEGIN

      FOR emp\_record IN emp\_cursor LOOP

**DBMS\_OUTPUT.PUT\_LINE**('员工号：'||emp\_record.empno||'姓名：'||emp\_record.ename||'职位：'||emp\_record.job);

      END LOOP;

    END;

进一步，还可以在FOR IN中利用子查询

    BEGIN

      FOR emp\_record IN (SELECT empno,ename, job FROM emp) LOOP

**DBMS\_OUTPUT.PUT\_LINE**('员工号：'||emp\_record.empno||'姓名：'||emp\_record.ename||'职位：'||emp\_record.job);

      END LOOP;

    END;

### 7.5.4 参数游标

**参数游标**是指带有参数的游标。通过使用参数游标，使用不同参数值可以生成不同的游标结果集。

例：查询指定部门的员工号、姓名和职位

    DECLARE

    CURSOR emp\_cursor(dno NUMBER) IS SELECT empno, ename, job FROM emp WHERE deptno = dno;

    BEGIN

    FOR emp\_record IN emp\_cursor(&no) LOOP

**DBMS\_OUTPUT.PUT\_LINE**('员工号：'||emp\_record.empno||'姓名：'||emp\_record.ename||'职位：'||emp\_record.job);

    END LOOP;

END;

也可以直接写成这样

    BEGIN

    FOR emp\_record IN (SELECT empno, ename, job FROM emp WHERE deptno = &no) LOOP

**DBMS\_OUTPUT.PUT\_LINE**('员工号：'||emp\_record.empno||'姓名：'||emp\_record.ename||'职位：'||emp\_record.job);

    END LOOP;

END;

### 7.5.5 隐式游标

**隐式游标**是由系统隐含创建的游标。主要用于对非查询语句，如修改，删除等操作，有Oracle系统自动地为这些操作设置游标并创建其工作区，对于隐式游标的操作，如定义、打开、取值及关闭操作，都有Oracle系统自动完成，无需用户进行处理。

隐式游标的名字为SQL，这是由Oracle系统定义的。当系统使用一个隐式游标时，可以通过隐式游标的属性来了解操作的状态和结果，进而控制程序的流程。

注意：通过SQL游标名总是只能访问前一个DML操作或单行SELECT操作的游标属性。

例：利用SQL%FOUND判断UPDATE执行是否成功，并打印影响行数

    BEGIN

    UPDATE emp **SET** sal = sal+100 WHERE deptno = &no;

    IF SQL**%FOUND** THEN

**DBMS\_OUTPUT.PUT\_LINE**('成功修改员工工资，影响了'|| SQL**%ROWCOUNT**||'行');

**COMMIT**;

    ELSE

**DBMS\_OUTPUT.PUT\_LINE**('成功修改员工工资');

**ROLLBACK**;

    END IF;

    END;

### 7.5.6 使用游标修改或删除数据

如果创建的游标需要执行更新或删除的操作必须带有FOR UPDATE子句。FOR UPDATE子句会将游标提取出来的数据进行行级锁定，这样在本会话更新期间，其他用户的会话就不能对当前游标中的数据行进行更新操作。

例：如果没有获取到锁，打开游标就会失败，同理，获得锁后，一直到提交前，别的会话都无法获得锁。

    DECLARE

    CURSOR emp\_cursor IS SELECT empno,job FROM emp FOR UPDATE;

    BEGIN

    FOR emp\_record IN emp\_cursor LOOP

**DBMS\_OUTPUT.PUT\_LINE**(emp\_record.empno || '---' ||emp\_record.job);

        IF emp\_record.job = 'PERSIDENT' THEN

        UPDATE emp **SET** sal = sal+1000 WHERE CURRENT OF emp\_cursor;

        ELSIF emp\_record.job = 'MANAGER' THEN

        UPDATE emp **SET** sal = sal+500 WHERE CURRENT OF emp\_cursor;

        ELSE

        UPDATE emp **SET** sal = sal+300 WHERE CURRENT OF emp\_cursor;

        END IF;

    END LOOP;

**COMMIT**;

END;

1. NOWAIT参数

用于指定不等待锁，如果发现所操作的数据行已经锁定，将不会等待，立即返回错误。

用法：CURSOR emp\_cursor IS SELECT empno,job FROM emp FOR UPDATE NOWAIT;

2. of子句

用于在特定表上加行共享锁。当游标子查询涉及到多张表时，如果在特定表上加行共享锁，那么需要使用OF子句。

例：输入部门号，显示该部门的部门名称及员工的姓名，并删除该部门下的这些员工。

    DECLARE

    CURSOR empnew\_cursor IS

        SELECT d.dname dname, e.ename ename FROM empnew e JOIN dept d ON e.deptno = d.deptno WHERE e.deptno = &deptno

        FOR UPDATE OF e.deptno;

    BEGIN

        FOR empnew\_record IN empnew\_cursor LOOP

**DBMS\_OUTPUT.PUT\_LINE**('部门名称：'||empnew\_record.dname||'员工名：'||empnew\_record.ename);

        DELETE FROM empnew WHERE CURRENT OF empnew\_cursor;

        END LOOP;

**COMMIT**;

END;

## 7.6 异常处理

**异常**是程序在正常执行过程中发生的未预料的事件。异常处理是为了提高程序的健壮性，使用异常处理部分可

以有效地解决程序正常执行过程中可能出现的各种错误，使程序正常运行。

### 7.6.1 预定义异常

预定义异常是指由PL/SQL所提供的系统异常，每个预定义异常对应一个特定的Oracle错误，当PL/SQL块出现这些Oracle错误时，会隐含地触发相应的预定义异常。

常见的预定义异常如下：

|  |  |  |
| --- | --- | --- |
| 错误号 | 异常错误信息名称 | 说明 |
| ORA-00001 | DUP\_VAL\_ON\_INDEX | 试图破坏一个唯一性限制 |
| ORA-00051 | TIMEOUT\_ON\_RESOURCE | 在等待资源时发生超时 |
| ORA-01001 | INVALID\_CURSOR | 试图使用一个无效的游标 |
| ORA-01012 | NOT\_LOGGED\_ON | 没有连接到ORACLE |
| ORA-01017 | LOGIN\_DENIED | 无效的用户名及口令 |
| ORA-01403 | NO\_DATA\_FOUND | SELECT INTO没有找到数据 |
| ORA-01422 | TWO\_MANY\_ROWS | SELECT INTO 返回多行 |
| ORA-01410 | SYS\_INVALID\_ROWID | 从字符串向ROWID转换发生错误 |
| ORA-01476 | ZERO\_DIVIDE | 数字值除零时触发的异常 |
| ORA-01722 | INVALID\_NUMBER | 转换一个数字失败 |
| ORA-06500 | STORAGE\_ERROR | 内存不够引发的内部错误 |
| ORA-06501 | PROGRAM\_ERROR | 存在PL/SQL内部问题 |
| ORA-06502 | VALUE\_ERROR | 转换或截断错误 |
| ORA-06504 | ROWTYPE\_MISMATCH | 宿主游标变量与 PL/SQL 游标变量的返回类型不兼容 |
| ORA-06511 | CURSOR\_ALREADY\_OPEN | 游标已经打开 |
| ORA-06530 | ACCESS\_INTO\_NULL | 未定义对象 |
| ORA-06531 | COLLECTION\_IS\_NULL | 集合元素未初始化 |
| ORA-06532 | SUBSCRIPT\_OUTSIDE\_LIMIT | 使用嵌套表或 VARRAY 时，将下标指定为负数 |
| ORA-06533 | SUBSCRIPT\_BEYOND\_COUNT | 元素下标超过嵌套表或 VARRAY 的最大值 |
| ORA-06592 | CASE\_NOT\_FOUND | CASE 中若未包含相应的 WHEN ，并且没有设置 |
| ORA-30625 | SELF\_IS\_NULL | 使用对象类型时，在 null 对象上调用对象方法 |

对于预定义异常情况的处理，无需在程序中定义，只需在PL/SQL块的异常处理部分，直接引用相应的异常情况名，并对其完成相应的异常错误处理即可。

例：根据输入的工资，查询员工的姓名,并输出员工的姓名及工资。如果没有该工资，则捕获NO\_DATA\_FOUND异常，如果发现重复记录，则捕获TOO\_MANY\_ROWS异常

  DECLARE

    v\_name emp.ename%TYPE;

    v\_sal  emp.sal%TYPE := &sal;

  BEGIN

    SELECT ename INTO v\_name FROM emp WHERE sal = v\_sal;

**DBMS\_OUTPUT.PUT\_LINE**(v\_name || '的工资是：'|| v\_sal );

    EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

**DBMS\_OUTPUT.PUT\_LINE**('没有该员工的工资!' );

    WHEN TOO\_MANY\_ROWS THEN

**DBMS\_OUTPUT.PUT\_LINE**('多个员工具有该工资!' );

    WHEN OTHERS THEN

**DBMS\_OUTPUT.PUT\_LINE**('其它异常!' );

  END;

### 7.6.2 非预定义异常

用于处理预定义异常所不能处理的ORACLE错误。此种异常需要在程序中定义。

例：删除dept表中指定部门的信息。在PL/SQL中定义非预定义异常，并关联oracle错误(-2292异常是违反外键约束的oracle错误码)，在末尾捕捉并处理异常

  DECLARE

*--1 定义非预定义异常的标氓符*

    e\_fk EXCEPTION;

*--2 把oracle错误与异常建立关联*

**PRAGMA EXCEPTION\_INIT**(e\_fk,-2292); *-- -2292异常是违反外键约束的oracle错误码*

  BEGIN

    DELETE FROM dept WHERE deptno = &deptno;

  EXCEPTION

    WHEN e\_fk THEN

*--3 捕捉并处理异常*

**DBMS\_OUTPUT.PUT\_LINE**('此部门下有员工，不能删除！');

     WHEN OTHERS THEN

**DBMS\_OUTPUT.PUT\_LINE**(SQLCODE || '###'||**SQLERRM**);

  END;

*-- SQLCODE与SQLERRM是oracle的异常处理函数，分别用来获取oracle错误号和错误信息，通过在异常处理部分引用，可以取得未预计到的oracle错误。*

### 7.6.3 自定义异常

如果你想在某个特定事件发生时向应用程序的用户发出一些警告信息。而事件本身不会抛出Oracle内部异常，这个异常是属于应用程序的特定异常。那么就需要自定义异常。用户定义的异常错误是通过显式使用RAISE语句来触发。当引发一个异常错误时，控制就转向到EXCEPTION块异常错误部分，执行错误处理代码。

例：给输入员工号，给员工工资加100。在声明部分定义自定义异常，适时抛出异常，并捕捉和处理异常

DECLARE

  v\_empno  emp.empno%TYPE := &empno;

*--1 定义自定义异常*

  e\_no\_result EXCEPTION;

BEGIN

  UPDATE emp **SET** sal = sal+100 WHERE empno = v\_empno;

  IF SQL**%NOTFOUND** THEN

*--2 适时抛出异常*

    RAISE e\_no\_result;

  ELSE

**COMMIT**;

  END IF;

EXCEPTION

*--3 捕获并处理异常*

  WHEN e\_no\_result THEN

**DBMS\_OUTPUT.PUT\_LINE**('数据更新失败！');

  WHEN OTHERS THEN

**DBMS\_OUTPUT.PUT\_LINE**('其他错误！');

END;

### 7.6.4 RAISE\_APPLICATION\_ERROR

  虽然我们通过DBMS\_OUTPUT.PUT\_LINE来输出异常信息，但是在实际的应用中，需要把异常信息返回给调用的客户端。而 **RAISE\_APPLICATION\_ERROR** 将应用程序专有的错误从服务器端转达到客户端应用程序(其他机器上的SQLPLUS或者其他前台开发语言)

其函数声明如下：

PROCEDURE **RAISE\_APPLICATION\_ERROR**( error\_number\_in IN NUMBER, error\_msg\_in IN VARCHAR2[,{true|| false}]);

*-- 错误号的范围是-20001到-20999；错误信息是文本字符串，最多为2048字节；true和false表示是添加(true)进错误堆(error stack)还是覆盖(overwrite)错误堆(false),缺省情况下是false。​​​​*

例1：

CREATE OR REPLACE PROCEDURE **change\_sal**(p\_empno NUMBER,p\_sal NUMBER) IS

BEGIN

  UPDATE EMP **SET** SAL = p\_sal WHERE EMPNO = p\_empno;

  IF SQL**%NOTFOUND** THEN

**RAISE\_APPLICATION\_ERROR**(-20002, '该员工不存在');

  ELSE

**DBMS\_OUTPUT.PUT\_LINE**('更新成功');

**COMMIT**;

  END IF;

END;

例2：

CREATE OR REPLACE PROCEDURE **new\_emp**(p\_empno in EMP.EMPNO%TYPE, p\_name in EMP.ENAME%type, p\_sal in EMP.SAL%TYPE, p\_job in EMP.JOB%TYPE,p\_deptno in EMP.DEPTNO%TYPE, p\_mgr in EMP.MGR%TYPE, p\_hired in EMP.HIREDATE%TYPE := **SYSDATE**)

IS

  e\_invalid\_manager EXCEPTION;

**PRAGMA EXCEPTION\_INIT**(e\_invalid\_manager, -2291);  *-- -2291表示违反完整性约束*

BEGIN

  IF **TRUNC**(p\_hired) > **TRUNC**(**SYSDATE**) THEN

**RAISE\_APPLICATION\_ERROR**(-20000, 'NEW\_EMP:hiredate cannot be in the future');

  END IF;

  INSERT INTO EMP(EMPNO,ENAME,SAL,JOB,DEPTNO,MGR,HIREDATE) VALUES(p\_empno, p\_name, p\_sal, p\_job, p\_deptno,p\_mgr,**TRUNC**(p\_hired));

EXCEPTION

  WHEN dup\_val\_on\_index THEN

**RAISE\_APPLICATION\_ERROR**(-20001, 'NEW\_EMP:employee called'|| p\_name||'already exists',TRUE);

  WHEN e\_invalid\_manager THEN

**RAISE\_APPLICATION\_ERROR**(-20002, 'NEW\_EMP:'||p\_mgr||'is not a valid manager');

END;

# 第八章 PL/SQL对象

## 8.1 存储过程与存储函数

  存储过程和存储函数是存储在数据库中的被命名的PLSQL块，供所有用户程序调用，完成特定功能的子程序。

### 8.1.1 存储过程的创建

创建存储过程的语法如下：

CREATE [OR **REPLACE**] PROCEDURE **Procedure\_name**  [(argment1[{IN | OUT | IN OUT}] Type，argment2[{IN|OUT |IN OUT}] Type，…)]

{IS|AS}

    声明部分

BEGIN

    执行部分

EXCEPTION

    异常错误处理部分

END;

1.创建无参数的存储过程

例：打印'存储过程'四个字

CREATE OR REPLACE PROCEDURE **first\_proc**

IS

BEGIN

**DBMS\_OUTPUT.PUT\_LINE**('存储过程!');

END;

2. 创建带输入参数的存储过程

例：输入员工号，删除该员工

CREATE OR REPLACE PROCEDURE **Proc\_IN**(v\_empno IN EMP.EMPNO%TYPE)

IS

BEGIN

  DELETE FROM EMP WHERE EMPNO = v\_empno;

  IF SQL**%NOTFOUND** THEN

**RAISE\_APPLICATION\_ERROR**(-20008,'指定删除的员工不存在！');

  ELSE

**DBMS\_OUTPUT.PUT\_LINE**('删除成功!');

  END IF;

END;

3. 创建带输出参数的存储过程

例：输入部门编号，输出该部门平均薪水和人数

CREATE OR REPLACE PROCEDURE **Proc\_OUT**(v\_deptno IN NUMBER, v\_avgsal OUT NUMBER, v\_cnt OUT NUMBER)

IS

BEGIN

  SELECT **AVG**(sal),**COUNT**(1) INTO v\_avgsal,v\_cnt FROM EMP WHERE DEPTNO = v\_deptno;

EXCEPTION

  WHEN NO\_DATA\_FOUND THEN

**DBMS\_OUTPUT.PUT\_LINE**('没有此部门！');

  WHEN OTHERS THEN

**DBMS\_OUTPUT.PUT\_LINE**(**SQLERRM**);

END;

4. 创建带输入输出参数的存储过程

例：调用存储过程，交换两个变量的值

CREATE OR REPLACE PROCEDURE **Proc\_IN\_OUT**(v\_num1 IN OUT NUMBER, v\_num2 IN OUT NUMBER)

AS

  v\_temp NUMBER := 0;

BEGIN

  v\_temp := v\_num1;

  v\_num1 := v\_num2;

  v\_num2 := v\_temp;

END;

### 8.1.2 存储函数的创建

创建存储函数的语法如下：

CREATE [OR **REPLACE**] FUNCTION **Function\_name** [(argment1[{INIOUTIIN OUT}]Type，argment2[{IN I OUTIIN OUT}]Type，…）]

RETURN return\_type

{IS|AS}

    声明部分

BEGIN

    执行部分

EXCEPTION

    异常错误处理部分

END;

1.创建无参数的存储函数

例：返回'存储函数'四个字

CREATE OR REPLACE FUNCTION **First\_func** RETURN VARCHAR2

IS

BEGIN

  RETURN'存储函数';

END;

2. 创建带输入参数的存储函数

例：输入部门编号，返回该部门的总工资

CREATE OR REPLACE FUNCTION **Func\_IN**(v\_deptno IN NUMBER) RETURN NUMBER

IS

  v\_sumsal NUMBER;

BEGIN

  SELECT **SUM**(SAL) INTO v\_sumsal FROM EMP WHERE DEPTNO = v\_deptno;

  RETURN v\_sumsal;

EXCEPTION

  WHEN NO\_DATA\_FOUND THEN

**DBMS\_OUTPUT.PUT\_LINE**('没有此部门!');

  WHEN OTHERS THEN

**DBMS\_OUTPUT.PUT\_LINE**(**SQLERRM**);

END;

3.创建带输出参数的存储函数

例：输入员工工号，输出员工的姓名和员工工资，并返回员工年收入

CREATE OR REPLACE FUNCTION **Func\_OUT**(v\_empno IN EMP.EMPNO%TYPE, v\_name OUT EMP.ENAME%TYPE,v\_sal OUT EMP.SAL%TYPE) RETURN NUMBER

IS

  v\_yearsal NUMBER;

BEGIN

  SELECT ENAME,SAL,(SAL+**NVL**(COMM,0))\*12 INTO v\_name,v\_sal,v\_yearsal FROM EMP WHERE EMPNO = v\_empno;

  RETURN v\_yearsal;

EXCEPTION

  WHEN NO\_DATA\_FOUND THEN

**DBMS\_OUTPUT.PUT\_LINE**('没有此员工!');

  WHEN OTHERS THEN

**DBMS\_OUTPUT.PUT\_LINE**(**SQLERRM**);

END;

4. 创建带输入和输出参数的函数

例：输入两个数，输出其平方，并返回其平方和

CREATE OR REPLACE FUNCTION **Func\_IN\_OUT**(n1 IN OUT NUMBER, n2 IN OUT NUMBER) RETURN NUMBER

IS

BEGIN

  n1 := n1\*n1;

  n2 := n2\*n2;

  RETURN n1+n2;

END;

### 8.1.3 存储过程的调用和删除

1. 调用无参数的存储过程

  BEGIN

    FIRST\_PROC;

  END;

2. 调用带输入参数的存储过程

  BEGIN

    PROC\_IN(7598);

  END;

3.  调用带输出参数的存储过程

  DECLARE

    v\_avgsal NUMBER;

    v\_cnt    NUMBER;

  BEGIN

    PROC\_OUT(10,v\_avgsal,v\_cnt);

**DBMS\_OUTPUT.PUT\_LINE**('平均工资:'||v\_avgsal);

**DBMS\_OUTPUT.PUT\_LINE**('总人数：'||v\_cnt);

  END;

4. 调用带输入输出参数的存储过程

  DECLARE

    v\_n1 NUMBER := 5;

    v\_n2 NUMBER := 10;

  BEGIN

    PROC\_IN\_OUT(v\_n1,v\_n2);

**DBMS\_OUTPUT.PUT\_LINE**('N1：'||v\_n1);

**DBMS\_OUTPUT.PUT\_LINE**('N2：'||v\_n2);

  END;

5. 删除存储过程

DROP PROCEDURE **FIRST\_PROC**;

6. 使用EXEC来调用存储过程

  EXEC方式调用存储过程本质上是在存储过程外嵌套一层BEGIN...END;如：

  SQL> EXEC first\_proc;

  SQL> EXEC PROC\_IN(1234);

*--注意设置：SET SERVEROUTPUT ON*

### 8.1.4 存储函数的调用和删除

1. 调用无参的存储函数

BEGIN

**DBMS\_OUTPUT.PUT\_LINE**(FIRST\_FUNC);

END;

2. 调用带输入参数的存储函数

BEGIN

**DBMS\_OUTPUT.PUT\_LINE**('部门工资的总额：'||FUNC\_IN(10));

END;

3. 调用带输出参数的存储函数

DECLARE

  v\_name   EMP.ENAME%TYPE;

  v\_sal    EMP.SAL%TYPE;

  v\_yearsal NUMBER;

BEGIN

  v\_yearsal := FUNC\_OUT(7568,v\_name,v\_sal);

**DBMS\_OUTPUT.PUT\_LINE**('姓名：'||v\_name);

**DBMS\_OUTPUT.PUT\_LINE**('工资：'||v\_sal);

**DBMS\_OUTPUT.PUT\_LINE**('年收入：'||v\_yearsal);

END;

4. 调用带输入输出参数的存储函数

DECLARE

 v\_num1 NUMBER := 5;

 v\_num2 NUMBER := 10;

 v\_sum NUMBER;

BEGIN

  v\_sum := FUNC\_IN\_OUT(v\_num1,v\_num2);

**DBMS\_OUTPUT.PUT\_LINE**('N1的平方：'||v\_num1);

**DBMS\_OUTPUT.PUT\_LINE**('N2的平方：'||v\_num2);

**DBMS\_OUTPUT.PUT\_LINE**('N1、N2的平方和：'||v\_sum);

END;

5. 删除函数

DROP FUNCTION **FIRST\_FUNC**;

### 8.1.5 存储过程和存储函数的比较

1) 关键字不同。存储过程为PROCEDURE，存储函数为FUNCTION。

2) 调用方式不同。存储过程一般为独立的过程调用语句，而存储函数一般以表达式方式调用。

3) 有无返回值。存储过程没有返回值。

4) 存储过程一般用来完成一系列的数据处理，而存储函数的目的是获得函数返回值。

5) 存储过程和存储函数都有参数默认值。

如下例：求部门的年收入

CREATE OR REPLACE FUNCTION **FUNC\_DEFAULT**(v\_deptno IN NUMBER DEFAULT 10) RETURN NUMBER

IS

  v\_yearsalsum NUMBER;

BEGIN

  SELECT **SUM**((SAL+**NVL**(COMM,0))\*12) INTO v\_yearsalsum FROM EMP WHERE DEPTNO = v\_deptno;

  RETURN v\_yearsalsum;

EXCEPTION

  WHEN NO\_DATA\_FOUND THEN

**DBMS\_OUTPUT.PUT\_LINE**('没有此部门！');

  WHEN OTHERS THEN

**DBMS\_OUTPUT.PUT\_LINE**(**SQLERRM**);

END;

DECLARE

  v\_totalsal NUMBER;

BEGIN

  v\_totalsal := FUNC\_DEFAULT;

**DBMS\_OUTPUT.PUT\_LINE**(v\_totalsal);

END;

## 8.2 包

  包是一组相关过程、函数、变量、常量和游标等PL/SQL程序设计元素的组合。它具有面向对象程序设计语言的特点，是对PL/SQL程序设计元素（过程、函数、变量等）的封装。

  一个包由两个分开的部分组成：

**包规范（包定义）**：用于定义包的公用组件，包括常量、变量、游标、过程和函数等。

**包体（包主体）**：用于实现包规范所定义的公用过程和函数。包体不仅可用于实现公用过程和函数，而且还可以定义包的私有组件（变量、游标、过程、函数等）。

下面是一个简单的包：

*-- 创建包*

CREATE OR REPLACE PACKAGE **First\_package**

IS

  v\_no EMP.DEPTNO%TYPE := 10;

  PROCEDURE **query\_emp**(v\_avgsal OUT NUMBER, v\_cnt OUT NUMBER,v\_deptno IN NUMBER DEFAULT v\_no);

END **First\_package**;

*-- 创建包体*

CREATE OR REPLACE PACKAGE BODY **First\_package** IS

  PROCEDURE **query\_emp**(v\_avgsal OUT NUMBER, v\_cnt OUT NUMBER,v\_deptno IN NUMBER DEFAULT v\_no) IS

  BEGIN

    SELECT **AVG**(sal),**COUNT**(\*) INTO v\_avgsal,v\_cnt FROM EMP WHERE DEPTNO = v\_deptno;

  EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

**DBMS\_OUTPUT.PUT\_LINE**('没有此部门');

    WHEN OTHERS THEN

**DBMS\_OUTPUT.PUT\_LINE**(**SQLERRM**);

    END;

END **First\_package**;

*-- 调用*

DECLARE

  v\_avgsal NUMBER(10,2);

  v\_cnt    NUMBER;

BEGIN

  First\_package.v\_no := 20;

  First\_package.query\_emp(v\_avgsal,v\_cnt);

**DBMS\_OUTPUT.PUT\_LINE**('平均工资：'||v\_avgsal);

**DBMS\_OUTPUT.PUT\_LINE**('总人数：'||v\_cnt);

END;

### 8.2.1 包的创建

一般来说，创建包分为创建包规范和创建包体两步：

1. 创建包规范语法格式如下：

CREATE [OR **REPLACE**] PACKAGE **package\_name**

IS|AS

*--在此处定义公用常量、变量、游标、过程、函数等*

END [package\_name];

例1：创建一个包规范，有两个过程，分别是添加员工信息和删除员工信息

CREATE OR REPLACE PACKAGE **emp\_package**

IS

  PROCEDURE **add\_emp\_pro**(v\_empno IN EMP.EMPNO%TYPE,v\_ename IN EMP.ENAME%TYPE,v\_job IN EMP.JOB%TYPE,MGR IN EMP.MGR%TYPE,

                                   v\_hiredate IN EMP.HIREDATE%TYPE,v\_sal IN EMP.SAL%TYPE,v\_comm IN EMP.COMM%TYPE,v\_deptno IN EMP.DEPTNO%TYPE);

  PROCEDURE **del\_emp\_pro**(v\_empno IN EMP.EMPNO%TYPE);

END **emp\_package**;

2. 创建包体的语法格式下：

CREATE [OR **REPLACE**] PACKAGE BODY **package\_name**

IS|AS

*--在此处定义私有常量、变量、游标、过程和函数等*

*--实在此处现公用过程和函数*

END [package\_name];

例1：实现上面包规范

CREATE OR REPLACE PACKAGE BODY **emp\_package**

IS

  PROCEDURE **add\_emp\_pro**(v\_empno IN EMP.EMPNO%TYPE,v\_ename IN EMP.ENAME%TYPE,v\_job IN EMP.JOB%TYPE,v\_mgr IN EMP.MGR%TYPE,

                                   v\_hiredate IN EMP.HIREDATE%TYPE,v\_sal IN EMP.SAL%TYPE,v\_comm IN EMP.COMM%TYPE,v\_deptno IN EMP.DEPTNO%TYPE)

  IS

    e\_2291 EXCEPTION;

**PRAGMA EXCEPTION\_INIT**(e\_2291,-2291);

  BEGIN

    INSERT INTO EMP VALUES(v\_empno,v\_ename,v\_job,v\_mgr,v\_hiredate,v\_sal,v\_comm,v\_deptno);

  EXCEPTION

    WHEN DUP\_VAL\_ON\_INDEX THEN

**RAISE\_APPLICATION\_ERROR**(-20001,'员工已存在！');

    WHEN e\_2291 THEN

**RAISE\_APPLICATION\_ERROR**(-20001,'部门号不存在');

    WHEN OTHERS THEN

**DBMS\_OUTPUT.PUT\_LINE**(v\_empno||' 插入成功!');

  END;

  PROCEDURE **del\_emp\_pro**(v\_empno IN EMP.EMPNO%TYPE)

  IS

  BEGIN

    DELETE FROM EMP WHERE EMPNO = v\_empno;

    IF SQL**%NOTFOUND** THEN

**RAISE\_APPLICATION\_ERROR**(-20008,'员工不存在');

    ELSE

**DBMS\_OUTPUT.PUT\_LINE**(v\_empno||' 已删除!');

    END IF;

  END;

END **emp\_package**;

例2：根据员工号查询工资，如果工资小于等于3000，工资涨500.

*-- 创建包规范*

CREATE OR REPLACE PACKAGE **emp\_sal\_pa**

IS

  FUNCTION **get\_sal**(p\_empno NUMBER) RETURN NUMBER;

  PROCEDURE **upd\_sal**(p\_empno NUMBER);

END;

*-- 创建包体*

CREATE OR REPLACE PACKAGE BODY **emp\_sal\_pa**

IS

  FUNCTION **get\_sal**(p\_empno NUMBER) RETURN NUMBER

  IS

    v\_sal  EMP.SAL%TYPE := 0;

  BEGIN

    SELECT SAL INTO v\_sal FROM EMP WHERE EMPNO = p\_empno;

    RETURN v\_sal;

  EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

**RAISE\_APPLICATION\_ERROR**(-20008,'该员工不存在！');

  END;

  PROCEDURE **upd\_sal**(p\_empno NUMBER)

  IS

    v\_sal  EMP.SAL%TYPE;

  BEGIN

    SELECT SAL INTO v\_sal FROM EMP WHERE EMPNO = p\_empno;

    IF v\_sal <= 3000 THEN

      UPDATE EMP **SET** SAL = SAL+500 WHERE EMPNO = p\_empno;

    END IF;

  EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

**RAISE\_APPLICATION\_ERROR**(-20008,'该员工不存在！');

  END;

END **emp\_sal\_pa**;

### 8.2.2 包的调用与删除

对包内共有元素(公共组件)一般采用[包名.元素名称]来进行调用

例1：调用emp\_package添加员工

DECLARE

  v\_empno EMP.EMPNO%TYPE := &empno;

  v\_ename EMP.ENAME%TYPE := '&name';

  v\_sal   EMP.SAL%TYPE := &sal;

  v\_deptno EMP.DEPTNO%TYPE := &deptno;

  e\_dup\_val EXCEPTION;

  e\_no\_dept EXCEPTION;

**PRAGMA EXCEPTION\_INIT**(e\_dup\_val,-20001);

**PRAGMA EXCEPTION\_INIT**(e\_no\_dept, -20002);

BEGIN

  emp\_package.add\_emp\_pro(v\_empno,v\_ename,'SALES',7839,**sysdate**,v\_sal,NULL,v\_deptno);

**COMMIT**;

EXCEPTION

  WHEN e\_dup\_val THEN

**DBMS\_OUTPUT.PUT\_LINE**(**SQLERRM**);

  WHEN e\_no\_dept THEN

**DBMS\_OUTPUT.PUT\_LINE**(**SQLERRM**);

**ROLLBACK**;

END;

例2：调用emp\_package中删除指定员工的过程

DECLARE

  v\_empno EMP.EMPNO%TYPE := &EMPNO;

  e\_no\_emp EXCEPTION;

**PRAGMA EXCEPTION\_INIT**(e\_no\_emp, -20008);

BEGIN

  emp\_package.del\_emp\_pro(v\_empno);

**COMMIT**;

EXCEPTION

  WHEN e\_no\_emp THEN

    DBMS\_OUTPUT. put\_line(**SQLERRM**);

**ROLLBACK**;

END;

例3：调用emp\_sal\_pkg更新员工工资

DECLARE

  v\_empno EMP.EMPNO%TYPE :=&EMPNO;

  v\_sal   EMP.SAL%TYPE;

  e\_no\_emp EXCEPTION;

**PRAGMA EXCEPTION\_INIT**(e\_no\_emp,-20008);

BEGIN

  emp\_sal\_pa.upd\_sal(v\_empno);

**COMMIT**;

  v\_sal := emp\_sal\_pa.get\_sal(v\_empno);

  DBMS\_OUTPUT. put\_line('员工工资变为：'||v\_sal);

EXCEPTION

  WHEN e\_no\_emp THEN

    DBMS\_OUTPUT. put\_line(**SQLERRM**);

END;

例4：使用EXEC调用包

SQL> VAR v\_empno NUMBER

SQL> EXEC :v\_empno := &no

PL/SQL procedure **successfully** completed

v\_empno

*---------*

1009

SQL> VAR v\_sal NUMBER

SQL> EXEC :v\_sal := emp\_sal\_pa.get\_sal(:v\_empno);

PL/SQL procedure **successfully** completed

v\_sal

*---------*

3500

v\_empno

*---------*

1009

例5：包的删除

可以使用DROPPACKAGE命令对不需要的包进行删除，语法如下：

DROP PACKAGE [BODY][**user**.]package\_name；

*-- 删除包和包体*

DROP PACKAGE **emp\_sal\_pa**

*--仅删除包体*

DROP PACKAGE BODY **emp\_package**

### 8.2.3 子程序重载

所谓重载是指两个或多个子程序有相同的名称，但拥有不同的参数变量、参数顺序或参数数据类型。

例：获取或删除员工信息，可根据员工号或姓名来操作，使用子程序重载

*-- 创建包规范*

CREATE OR REPLACE PACKAGE **overload\_pkg**

IS

  FUNCTION **get\_info**(p\_empno NUMBER) RETURN EMP%ROWTYPE;

  FUNCTION **get\_info**(p\_ename VARCHAR2) RETURN EMP%ROWTYPE;

  PROCEDURE **del\_emp**(p\_empno NUMBER);

  PROCEDURE **del\_emp**(p\_ename VARCHAR2);

END;

*--创建包体*

CREATE OR REPLACE PACKAGE BODY **overload\_pkg**

IS

  FUNCTION **get\_info**(p\_empno NUMBER) RETURN EMP%ROWTYPE

  IS

    emp\_record EMP%ROWTYPE;

  BEGIN

    SELECT \* INTO emp\_record FROM EMP WHERE EMPNO = p\_empno;

    RETURN emp\_record;

  EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

**RAISE\_APPLICATION\_ERROR**(-20002,'没有此员工！');

  END;

  FUNCTION **get\_info**(p\_ename VARCHAR2) RETURN EMP%ROWTYPE

  IS

    emp\_record EMP%ROWTYPE;

  BEGIN

    SELECT \* INTO emp\_record FROM EMP WHERE ENAME = p\_ename;

    RETURN emp\_record;

    EXCEPTION

  WHEN NO\_DATA\_FOUND THEN

**RAISE\_APPLICATION\_ERROR**(-20002,'没有此员工！');

  END;

  PROCEDURE **del\_emp**(p\_empno NUMBER)

  IS

  BEGIN

    DELETE EMP WHERE EMPNO = p\_empno;

    IF SQL**%NOTFOUND** THEN

**RAISE\_APPLICATION\_ERROR**(-20002,'没有此员工！');

    END IF;

  END;

  PROCEDURE **del\_emp**(p\_ename VARCHAR2)

  IS

  BEGIN

    DELETE EMP WHERE ENAME = p\_ename;

    IF SQL**%NOTFOUND** THEN

**RAISE\_APPLICATION\_ERROR**(-20002,'没有此员工！');

    END IF;

  END;

END;

*-- 调用：查询员工信息*

DECLARE

  emp\_record EMP%ROWTYPE;

  e\_no\_emp EXCEPTION;

**PRAGMA EXCEPTION\_INIT**(e\_no\_emp,-20002);

BEGIN

*-- emp\_record := overload\_pkg.get\_info(&no);*

  emp\_record := overload\_pkg.get\_info('&name');

**DBMS\_OUTPUT.PUT\_LINE**('员工号：'||emp\_record.empno || '员工姓名：'||emp\_record.ename||' 员工工资：'||emp\_record.sal);

  EXCEPTION

    WHEN e\_no\_emp THEN

**DBMS\_OUTPUT.PUT\_LINE**(**SQLERRM**);

END;

*-- 调用：删除员工信息*

DECLARE

  e\_no\_emp EXCEPTION;

**PRAGMA EXCEPTION\_INIT**(e\_no\_emp,-20002);

BEGIN

*-- overload\_pkg.del\_emp(&no);*

  overload\_pkg.del\_emp('&name');

**COMMIT**;

EXCEPTION

  WHEN e\_no\_emp THEN

**DBMS\_OUTPUT.PUT\_LINE**(**SQLERRM**);

**ROLLBACK**;

END;

## 8.3 触发器

触发器是指存放在数据库中，并且被隐含执行的存储过程。当发生特定事件时，Oracle会自动执行触发器的相应代码。

触发器的类型有：DML触发器、DDL触发器、替代（instead of）触发器以及系统触发器。

触发器的组成：

  触发事件：即在何种情况下触发TRIGGER

  触发时间：即该TRIGGER是在触发事件发生之前（BEFORE）还是之后（AFTER）触发

  触发器本身：即该TRIGGER被触发之后的目的和意图，正是触发器本身要做的事情。

  触发频率：说明触发器内定义的动作被执行的次数。

一个简单的触发器如下：

*--当我们对empnew执行删除操作之后，它就会出现一个提示信息，提示：这是删除操作！*

CREATE TRIGGER **First\_TRIGGER**

AFTER DELETE ON EMP

BEGIN

**DBMS\_OUTPUT.PUT\_LINE**('EMP表发生了删除操作！');

END;

### 8.3.1 DDL触发器

当创建、修改或者删除数据库对象时，也会引起相应的触发器操作事件，而此时就可以利用触发器来对这些数据库对象的DDL操作进行监控。

创建DDL触发器的语法格式如下：

  CREATE[OR **REPLACE**]TRIGGER触发器的名称

  [BEFORE | AFTER | INTEAD OF] [DDL事件] ON [DATABASE | SCHEMA]

  [WHEN 触发条件]

  [DECLARE]

    [程序的声明部分]

  BEGIN

    程序的代码部分;

  END;

常见地DDL事件如下：

|  |  |  |  |
| --- | --- | --- | --- |
| NO. | DDL事件触发时机 | 描述 |  |
| 1 | ALTER | BEFORE/AFTER | 修改对象的结构时触发 |
| 2 | ANALYZE | BEFORE/AFTER | 分析数据库对象时触发 |
| 3 | ASSOCIATE STATISTICS | BEFORE/AFTER | 启动统计数据库对象时触发 |
| 4 | AUDIT | BEFORE/AFTER | 开启审核数据库对象时触发 |
| 5 | COMMENT | BEFORE/AFTER | 对数据库对象做注释时触发 |
| 6 | CREATE | BEFORE/AFTER | 创建数据库对象时触发 |
| 7 | DDL | BEFORE/AFTER | 针对出现的所用DDL事件都会触发 |
| 8 | DISASSOCIATE STATISTICS | BEFORE/AFTER | 关闭统计数据库对象时触发 |
| 9 | DROP | BEFORE/AFTER | 删除数据库对象时触发 |
| 10 | GRANT | BEFORE/AFTER | 通过SQL的GRANT命令赋权时触发 |
| 11 | NOAUDIT | BEFORE/AFTER | 禁用审核数据库对象时触发 |
| 12 | RENAME | BEFORE/AFTER | 通过SQL的RENAME命令对对象重命名时触发 |
| 13 | REVOKE | BEFORE/AFTER | 通过SQL的REVOKE语句撤销授权时触发 |
| 14 | TRUNCATE | BEFORE/AFTER | 通过SQL的TRUNCATE语句截断表时触发 |

例1：禁止SCOTT用户的所有DDL操作

CREATE OR REPLACE TRIGGER **scot\_trigger**

BEFORE DDL ON SCHEMA

BEGIN

**RAISE\_APPLICATION\_ERROR**(-20005,'SCOTT用户不允许任何DDL操作！');

END;

例2：实现对数据库对象操作的日志记录

*-- 创建数据库对象DDL操作日志记录表*

SQL> CONN SYS /AS SYSDBA

CREATE TABLE object\_log(

logid NUMBER CONSTRAINT pk\_logid PRIMARY KEY,

operate\_date DATE NOT NULL,

object\_type VARCHAR2(50) NOT NULL,

object\_owner VARCHAR2(50) NOT NULL

);

*-- 创建实现对数据库对象DDL操作记录的触发器*

CREATE SEQUENCE object\_1og\_seq;

CREATE OR REPLACE TRIGGER **object\_trigger**

AFTER CREATE OR DROP OR ALTER ON DATABASE

BEGIN

  INSERT INTO object\_log values(object\_10g\_seq.nextval,**sysdate**,ora\_dict\_obj\_type,ora\_dict\_obj\_owner );

END;

*-- 测试*

SQL> CONN SCOTT/SCOTT

CREATE SEQUENCE test\_seq2;

SQL> CONN SYS /AS SYSDBA

SELECT \* FROM object\_log;

### 8.3.2 DML触发器

DML触发器是指基于DML操作所建立的触发器。DML触发器可用于实现数据安全保护、数据审计、数据完整性、参照完整性、数据复制等功能。

DML触发器又分为：

  语句触发器：在指定的操作语句之前或之后执行一次，不管这条语句影响了多少行。

  行触发器：触发语句作用的每一条记录都被触发，在行级触发器中使用 :old 和 :new 伪记录变量，识别值的状态。

创建DML触发器的语法格式如下：

CREATE [OR **REPLACE**] TRIGGER **trigger\_name**

{BEFORE | AFTER}

{DELETE | INSERT | UPDATE [ OF 列名]} ON 表名

[FOR EACH ROW [WHEN（条件）]

  >>PLSQL块<<

例1：实现数据安全保护（数据的安全性检查）,如禁止在休息日改变EMP表的数据

CREATE OR REPLACE TRIGGER **emp\_TRIGGER1**

BEFORE INSERT OR UPDATE OR DELETE ON EMP

BEGIN

  IF **TO\_CHAR**(**SYSDATE**,'day') IN ('星期六','星期日') THEN

**RAISE\_APPLICATION\_ERROR**(-20006,'不能在休息日改变员工信息!');

  END IF;

END;

*-- 测试*

DELETE EMP WHERE EMPNO = 7569

SELECT \* FROM EMP;

例2：实现数据审计，如 审计员工信息表数据的变化，审计删除时间，以及被删除的雇员名。

*--创建审计表*

CREATE TABLE delete\_emp\_audit(

name VARCHAR2(10),

delete\_time DATE

);

*--创建触发器*

CREATE OR REPLACE TRIGGER **del\_emp\_trigger**

AFTER DELETE ON EMP FOR EACH ROW

BEGIN

  INSERT INTO delete\_emp\_audit VALUES(:old.ename,**SYSDATE**);

END;

*--测试*

DELETE FROM EMP WHERE empno=7566;

select \* from delete\_emp\_audit;

例3：实现数据完整性（数据确认），如：要求员工涨后工作不能低于原来的工资，并且所涨的工资不能超过原工资的50%。

*-- 数据完整性用于确保数据满足商业逻辑或企业规则，实现数据完整性首选约束，约束无法实现的，可以使用触发器实现数据完整性。*

CREATE OR REPLACE TRIGGER **TR\_CHECK\_SAL**

BEFORE UPDATE OF SAL ON EMP FOR EACH ROW

WHEN (new.sal < old.sal OR new.sal > old.sal\*1.5)

BEGIN

**RAISE\_APPLICATION\_ERROR**(-20028,'工资只升不降，并且升幅不能超过50%！');

END;

*-- 测试*

UPDATE EMP **SET** SAL = SAL\*1.8 WHERE EMPNO = 7788;

示例4：实现参照完整性（比如级联更新）

参照完整性是指在两张表之间具有主从关系。当删除主表数据时，需要首先删除从表的相关数据；当更新主表主键列时，需要首先更新从表相关数据。

CREATE OR REPLACE TRIGGER **upd\_cascade\_trigger**

AFTER UPDATE OF DEPTNO ON DEPT FOR EACH ROW

BEGIN

  UPDATE EMP **SET** DEPTNO = :new.deptno WHERE DEPTNO = :old.deptno;

END;

*--测试*

UPDATE dept **SET** deptno=50 WHERE deptno=10;

例5：利用触发器实现系统的同步备份

*--创建员工装的务份表*

CREATE TABLE emp\_bak

AS

SELECT \*FROM emp;

*-- 创建触发器来实现数据的同步务份:如同步删除*

CREATE OR REPLACE TRIGGER **emp\_bak\_trigger**

AFTER DELETE ON EMP FOR EACH ROW

BEGIN

  DELETE FROM emp\_bak WHERE EMPNO = :old.empno;

END;

### 8.3.3 替代触发器

在简单视图上往往可以执行INSERT、UPDATE、DELETE操作。但在复杂视图上执行INSERT、UPDATE、DLETE操作是有限制的。为了在这些复杂的视图上执行DML操作，需要建立INSTEAD OF触发器。

INSTEAD OF 触发器的限制

  替代触发器只适用于视图

  替代触发器不能指定BEFORE和AFTER选项

  不能在具有WITH CHECK OPTION选项的视图上建立替代触发器

  替代触发器必须包含FOR EACH ROW选项

例：给EMP\_DEPT视图创建一个替代触发器

*-- 创建EMP\_DEPT视图*

CREATE OR REPLACE VIEW **emp\_dept**

AS

SELECT d.deptno,d.dname,e.empno, e.ename

  FROM DEPT d, EMP e

  WHERE d.deptno = e.deptno;

*-- 在视图上创建替代触发器*

CREATE OR REPLACE TRIGGER **instead\_of\_trigger**

INSTEAD OF INSERT ON EMP\_DEPT FOR EACH ROW

DECLARE

  v\_temp INT;

BEGIN

  SELECT **COUNT**(\*) INTO v\_temp FROM DEPT WHERE DEPTNO = :new.deptno;

  IF v\_temp = 0 THEN

    INSERT INTO DEPT(DEPTNO, DNAME) VALUES(:new.deptno, :new.dname);

  END IF;

  SELECT **COUNT**(\*) INTO v\_temp FROM EMP WHERE EMPNO = :new.empno;

  IF v\_temp = 0 THEN

    INSERT INTO EMP(EMPNO, ENAME, DEPTNO) VALUES(:new.empno, :new.ename, :new.deptno);

  END IF;

END;

*-- 在连接视图上插入数据*

INSERT INTO EMP\_DEPT VALUES(50,'DEVELOPMENT',2222,'ALICE');

### 8.3.4 系统触发器

系统触发器是由特定系统事件(与例程或方案相关的数据库事件)所触发的触发器。常见的系统事件有：

STARTUP事件触发器：在启动数据库后触发。

SHUTDOWN事件触发器：在关闭数据之前触发。

DB\_ROLE\_CHANGE事件触发器：在改变角色后第一次打开数据库时触发。

SERVERERROR事件触发器：在发生Oracle错误时触发

*-- 创建一个事件表*

SQL> CONN /as sysdba

CREATE TABLE event\_table(

  event VARCHAR2(50),

  event\_time DATE

)

*-- 创建一个系统触发器：在开机后写入事件*

CREATE OR REPLACE TRIGGER **startup\_trigger**

AFTER STARTUP ON DATABASE

BEGIN

  INSERT INTO event\_table VALUES(ora\_sysevent, **SYSDATE**);

END;

三范式与反范式