

360Oracle 技术服务中心

Oracle 数据库设计文档

目录

第 1 章 Oracle 数据库安装配置.....	5
1.1 配置监听程序.....	5
1.2 配置网络服务名.....	6
第 2 章管理用户访问.....	6
2.1 表空间、用户及方案概述.....	6
2.2 用户访问.....	7
2.3 创建用户.....	8
2.4 修改用户.....	8
2.5 锁定用户账户.....	9
2.6 解锁用户账户.....	9
2.7 默认角色.....	9
2.8 授予用户访问权.....	9
2.9 撤消用户.....	11
2.10 废除用户访问.....	11
2.11 创建角色.....	11
2.12 新建的用户怎样才能成功创建一个表.....	12
第 3 章建立和管理表.....	13
3.1 简介.....	14
3.2 建表.....	15
3.3 修改表.....	16
3.4 截断和删除表.....	17
3.5 显示表信息.....	17
第 4 章 Oracle 操作符.....	18
第 5 章 基本查询.....	22
5.1 简单查询语句.....	22
5.2 排序数据.....	24
第 6 章 复杂查询.....	25
6.1 数据分组.....	25
6.2 连接查询.....	26
6.3 相等连接（包括 SQL：1999 标准内连接、自然连接）.....	27
6.4 不等连接.....	27
6.5 自连接.....	27
6.6 [内连接和]外连接.....	27
6.6.1 内连接.....	28
6.6.2 左外连接.....	28
6.6.3 右外连接.....	28
6.6.4 完全外连接.....	28
6.7 子查询.....	28

6.7.1 单行子查询（单列）	28
6.7.2 多行子查询（单列）	28
6.7.3 多列子查询	29
6.7.4 其他子查询	29
6.8 使用集合操作符	30
第 7 章 常用 SQL 函数	30
第 8 章 操纵数据	44
8.1 插入数据	44
8.2 更新数据	46
8.3 删除数据	47
第 9 章 使用事务	48
9.1 概述	49
9.2 事务分类	49
9.3 回复修改	50
9.4 回复部分事务	51
9.5 事务和锁	52
第 10 章 使用约束	52
10.1 约束简介	52
10.2 定义约束	53
10.3 维护约束	54
10.5 显示约束信息	55
第 11 章 使用视图	56
11.1 视图简介	56
11.2 建立视图	57
11.3 维护视图	58
11.4 显示视图信息	58
第 12 章 使用其它对象（索引序列同义词）	59
12.1 使用索引	59
12.1.1 建立索引	60
12.1.2 维护索引	60
12.1.3 显示索引信息	61
11.2 使用序列	61
11.2.1 建立序列	61
11.2.2 使用序列	61
11.2.3 维护序列	62
11.2.4 显示序列信息	62
11.3 使用同义词	62
11.3.1 建立同义词	62
11.3.2 删除同义词	62
第 13 章 PL/SQL 语句	63
13.1 PL/SQL 简介	63
13.2 PL/SQL 块	65
13.2.1 定义并使用变量	66
13.2.2 异常	67

13.2.2 游标.....	70
13.3 过程函数包	74
第 14 章 过程函数包及触发器.....	74
14.1 过程	74
14.1.1 建立过程.....	74
14.1.2 显示过程代码.....	75
14.2 函数	75
14.2.1 建立函数.....	75
14.2.2 删除函数.....	76
14.2.3 显示函数代码.....	76
14.3 包	76
14.3.1 建立包规范.....	76
14.3.2 建立包体.....	77
14.3.3 删除包.....	78
14.3.4 显示包代码.....	78
14.4 触发器	78
14.4.1 语句触发器.....	78
14.4.2 行触发器.....	79
14.4.3 使用触发器的注意事项.....	80
14.4.4 编译触发器.....	80
14.4.5 删除触发器.....	80
14.4.6 显示触发器代码.....	80
第十五章 使用 EXP 和 IMP.....	80
15.1 使用 EXP.....	81
15.1.1 导出表.....	81
15.1.2 导出方案.....	81
15.1.3 导出数据库.....	81
15.2 使用 IMP.....	82
15.2.1 导入表.....	82
15.2.2 导入方案.....	82
15.2.3 导入数据库.....	82

第 1 章 Oracle 数据库安装配置

当安装 Oracle Database 时，如果没有建立数据库，在安装完成之后可以使用 DBCA 工具建立数据库。数据库配置助手(Database Configuration Assistant)用于建立数据库、配置数据库选项、删除数据库和管理模板。

当建立了 Oracle 数据库之后，为了使得服务器端可以监听该 Oracle 数据库，必须配置监听程序。为了使得客户端可以访问该数据库，必须在客户端配置网络服务名。只有合理地配置了监听程序和网络服务名之后，客户应用才能访问该数据库。配置监听程序和网络服务名可以使用网络管理工具 Net Manager 完成。(源码网整理：www.codepub.com)

1.1 配置监听程序

监听程序用于接收客户端的连接请求。当客户应用访问 Oracle Server 时，监听程序会接

收并检查连接请求，以确定是否可以为该客户应用提供数据服务。在建立了 Oracle 数据库之后，为了使得客户应用可以访问 Oracle 数据库，必须在监听程序中追加该数据库。一个监听程序可以监听多个 Oracle 数据库，多个监听程序也可以监听同一个 Oracle 数据库。当安装数据库产品时，会自动建立默认监听程序 LISTENER。

配置监听程序的具体步骤如下：

- 展开监听程序，并选中 LISTENER 节点，此时在 NET MANAGER 窗口右端会显示默认监听位置，其中“协议”用于指定监听程序要使用的网络协议(默认为 TCP/IP)；“主机”用于指定服务器所在机器的主机名或 IP 地址；“端口”用于指定监听程序要使用的 TCP/IP 端口号（默认 1521）。
- 在 Net Manager 窗口上端的下拉列表中选择数据库服务，此时会显示默认的数据库配置，“全局数据库名”用于指定数据库的全局数据库名；“Oracle 主目录”用于指定 Oracle 数据库软件的安装路径；“SID”用于指数据库例程名。为了监听新建的数据库（如 DEMO），必须追加该数据库。
- 单击“添加数据库”按钮，然后进行相应配置，在“全局数据库名”处输入 DEMO 数据库的初始化参数 SERVICE—HOME 所对应的值，在 SID 处输入 DEMO 数据库的例程名。配置了监听程序之后，保存网络配置信息。
- 保存了监听程序配置之后，为了使得其网络配置生效，必须重新启动监听程序。（通过服务器管理器重新启动监听程序）

1.2 配置网络服务名

- 选中“服务命名”，然后单击+按钮，此时会显示“Net 服务名”界面，建议使用数据库名作网络服务名。
- 选取与监听程序一致的网络协议“TCP/IP”。
- 指定数据库所在主机名及其监听端口号。
- 指定监听程序所配置的全局数据库名或者 SID。
- 测试网络服务名配置是否成功，如果成功则表示网络服务名配置正确。
- 完成网络服务名配置之后，保存网络配置信息。

第 2 章管理用户访问

本章主要内容：

- 表空间、用户及方案概述
- 用户访问
- 创建用户
- 修改用户
- 授予用户访问权
- 撤消用户
- 废除用户访问
- 创建角色

2.1表空间、用户及方案概述

表空间是数据库的逻辑组成部分。从物理上说，数据库数据存放在数据文件中；从

逻辑上说，数据库数据存放在表空间(tablespace)中，并且表空间是由一个或多个数据文件组成的。

一个表空间是由一个或多个数据文件组成的。

用户（也称为帐户）是定义在数据库中的一个名称，它是 Oracle 数据库的基本访问控制机制。当连接到 Oracle 数据库时，默认情况下必须要提供用户名和口令。只有在输入了正确的用户名和口令之后，才能够连接到数据库，并执行各种管理操作和数据访问操作。

方案(Schema)是用户所拥有数据库对象的集合。在 Oracle 数据库中对象是以用户来组织的，用户与方案是一一对应的关系，并且二者名称相同。例 SCOTT 用户所拥有的所有对象都属于 SCOTT 方案，而 SYSTEM 用户所拥有的所有对象都属于 SYSTEM 方案。当访问数据库对象时，有一些注意事项：

- 在同一个方案中不能存在同名对象，但不同方案可以具有同名对象。
- 用户可以直接访问其方案对象，但如果要访问其他方案对象，则必须具有对象权限。如用户 SCOTT 可以直接查询其方案表 EMP，但如果用户 SMITH 要检索 SCOTT 方案的表 EMP，则必须在 EMP 表上具有 SELECT 对象权限。
- 当用户访问其他方案对象时，必须加方案名为前缀。例，如用户 SMITH 要访问 SCOTT 方案的 EMP 表，则必须使用 SCOTT.EMP。

2.2 用户访问

在多用户环境里，一个数据库可能有多个用户同时在访问。当有不同的用户同时访问数据库时，保护数据库安全，防范非授权访问非常重要。因此，必须在数据库里创建用户，并为用户指定用户名和密码，这样可以保证只有经过授权的，即有正确用户名和密码的用户才能访问数据库。数据库管理员（DBA）是最高级别的用户，他可以创建其他用户。在创建了用户后，DBA 需要按用户的需求为用户分配权限。权限指用户执行特定语句的许可，这意味并非所有用户都被允许修改重要数据。例如，某个用户可能只需要有连接数据库和查询某些表的记录的权限。类似地，另一个用户可能要求有创建和修改表的权限。

DBA 有访问数据库的一切权限，并有权为其他用户分配权限。下表列出了部分 DBA 权限：

DBA 权限	有权执行
CREATE USER	创建新用户
DROP USER	撤消用户
DROP ANY TABLE	撤消表
BACKUP ANY TABLE	为表制作备份
SELECT ANY TABLE	查询数据库对象，如表和视图
CREATE ANY TABLE	创建表

为了维护存储在数据库中数据的安全，Oracle 提供了以下数据库安全措施：

- 管理和控制数据库访问
- 用 Oracle 数据字典验证权限
- 为指定用户提供对数据库特定对象（表、视图和序列等）的访问
- 为数据库对象提供同义词

可被数据库操作采用的数据库的安全策略包括：

- 系统安全：系统安全涉及系统级的访问，如允许用户通过指出用户名和密码连接 Oracle，为用户分配磁盘空间，限定用户所能执行的操作。用户能执行的操作包括：查询数据库

对象的内容、创建数据库对象和更改数据库对象。

- 数据安全：数据安全涉及对数据库对象的访问和使用，以及用户在数据库对象上所拥有的权限的程度。

2.3 创建用户

`CREATE USER` 语句用于创建新用户。在创建新用户时，必须为新用户指定用户名和密码。为了使新用户能登录服务器和访问数据库，DBA 必须显式地为用户分配权限。只有拥有 `CREATE USER` 权限的用户才能创建新用户。例如，DBA 创建了新用户 Susan，但 Susan 无权创建其他新用户。只有当 Susan 有了 `CREATE USER` 这个权限后，她才能创建别的新用户。

创建用户的语法示例：

```
CREATE USER user IDENTIFIED BY password
DEFAULT TABLESPACE data01
TEMPORARY TABLESPACE temp
QUOTA 3M ON data01
PASSWORD EXPIRE;
```

其中 `IDENTIFIED BY` 用于指定用户口令；`DEFAULT TABLESPACE` 用于指定用户的默认表空间，当建立表或者索引时，如果不指定 `TABLESPACE` 子句，那么 Oracle 会自动在默认表空间上为这些对象分配空间；`TEMPORARY TABLESPACE` 用于指定用户的临时表空间，当用户执行排序操作时，或临时数据超过 PGA 工作区，则会在该表空间上建立临时段；`QUOTA` 用于指定表空间配额，即用户对象在表空间上可占用的最大空间；`PASSWORD EXPIRE` 用于指定终止口令，最终强制用户在登录时改变口令。当建立了新用户之后，需要注意以下问题：

- 初始创建的数据库用户没有任何权限，不能执行任何数据库操作。
- 如果在建立用户时不指定 `DEFAULT TABLESPACE` 子句，那么 Oracle 会将数据库默认表空间作为用户的默认表空间。在 Oracle Database 10g 之前，如果不指定 `DEFAULT TABLESPACE` 子句，那么 Oracle 会将 `SYSTEM` 表空间作为用户的默认表空间。
- 如果在建立用户时不指定 `TEMPORARY TABLESPACE` 子句，那么 Oracle 会将数据库默认临时表空间作为用户的临时表空间。
- 如果在建立用户时没有为特定表空间指定 `QUOTA` 子句，那么用户在特定表空间上的配额为 0，这样用户将不能在相应表空间上建立数据对象。

2.4 修改用户

修改用户信息是使用 `ALTER USER` 命令完成的。一般情况下，该命令是由 DBA 来执行的，如果以其他用户身份修改用户信息，必须要具有 `ALTER USER` 系统权限。

1. 修改口令

创建用户时为每个用户指定一个初始密码。之后可修改密码，修改密码的方法有两种：

- 管理员修改：`ALTER USER user IDENTIFIED BY password;(password 为用户的新密码)`
- 用户自己修改：用户登录后输入命令：`PASSWORD;`

修改表空间配额

表空间配额用于限制用户对象在表空间上可占用的最大空间。如果用户对象已经占

满了表空间配额所允许的最大空间，那么该用户将不能在该表空间上分配新的空间。此时如果执行了涉及到空间分配的 SQL 操作（如 INSERT、UPDATE、CREATE TABLE 等），则会显示错误，修改表空间语法：

```
ALTER USER user QUOTA 10M ON data01;
```

2.5 锁定用户账户

为了禁止特定数据库用户访问数据库，DBA 可以锁定用户账户，

```
ALTER USER user ACCOUNT LOCK;
```

2.6 解锁用户账户

为了使得数据库用户可以访问数据库，DBA 可以解锁用户账户。

```
ALTER USER user ACCOUNT UNLOCK;
```

2.7 默认角色

当将多个角色授予数据库用户之后，通过使用 ALTER USER 命令可以设置用户的默认角色。（当为用户指定了默认角色后，以该用户身份登录时会自动激活其默认角色，并不激活非默认角色）

```
ALTER USER user DEFAULT ROLE select_role;
```

[补充：]

默认角色和非默认角色的区别是什么？

这个理解有多种，第一种：默认角色可以是我们的创建数据库就可以见的 connect/dba 等，非默认角色需要我们自己创建；第二种：一个用户可以有多个角色，默认的角色登录即生效，非默认的需要激活才能使用。

默认角色和权限集是 Oracle 安装过程中预先定义的。每个版本的默认角色都有所变化。

```
CREATE USER SMIS IDENTIFIED BY SMIS;
grant dba,connect to SMIS;
create role r_px;
grant r_px to SMIS;
select * from dba_role_privs where grantee='SMIS';
alter user SMIS default role all except r_px;
select * from dba_role_privs where grantee='SMIS';
```

2.8 授予用户访问权

用户创建后，数据库管理员需要为他分配权限。权限关系到数据库的安全，它决定了用户在数据库上所能执行的操作。GRANT 语句用于为用户分配权限。语法如下：

```
GRANT privilege TO user;
```

[注释：授予所有系统权限 GRANT ALL PRIVILEGES to test_2;]

[注释：授予所有对象权限 GRANT ALL [PRIVILEGES] ON DEMO.DEPT TO TEST_1;]

可分配给用户的权限有：

- 系统权限：允许用户访问数据库的权限称为系统权限。
- 对象权限：允许用户在数据库对象上执行查询、更新、删除或添加数据等操作的权限，此称为对象权限。

常用的系统权限有：

系统权限	有权执行
CREATE SESSION	连接数据库
CREATE TABLE	创建表
CREATE VIEW	创建视图
CREATE PUBLIC SYNONYM	建立同义词
CREATE SEQUENCE	创建序列
CREATE PROCEDURE	建立过程、函数和包
CREATE TRIGGER	建立触发器
CREATE CLUSTER	建立簇
CREATE TYPE	建立对象类型
CREATE DATABASE LINK	建立数据库链

另外，Oracle 还提供了一类 ANY 系统权限，当用户具有该类系统权限时，可以在任何方案中执行相应操作。例如，如果用户具有 SELECT ANY TABLE 系统权限，那么用户可以查询任何方案的表（除数据字典基表和数据字典视图 DBA_XXX---DBA 和特权用户专访）。

如：GRANT CREATE SESSION,CREATE SEQUENCE,CREATE VIEW TO john;

上述命令执行后，用户 john 将拥有创建会话、视图和序列的系统权限。

*Oracle 提供了 100 多种系统权限（ALTER TABLE，ALTER VIEW，ALTER PROCEDURE，DROP TABLE，DROP VIEW，DROP PROCEDURE 等）。

一般情况下，授予系统权限是由 DBA 来完成的；如果要以其他用户身份授予系统权限，则要求该用户必须具有 GRANT ANY PRIVILEGE 系统权限，或者具有相应系统权限及其转授系统权限选项（WITH ADMIN OPTION），授予系统权限是使用 GRANT 命令来完成的，其语法如下：

```
GRANT system_priv[,system_priv,...]
TO {user | role | public},[,{user | role | public}]...
[WITH ADMIN OPTION];
```

注：user 也可以是用户组 PUBLIC；UNLIMITED TABLESPACE 权限不能被授予角色
下表列出了所有对象权限。

对象权限	适用于
ALTER	表、序列
DELETE	表、视图
EXECUTE	过程
INDEX	表
INSERT	表、视图
REFERENCES	表（基于表建立从表）
SELECT	表、视图、序列
UPDATE	表、视图

如果用户在某个同义词上拥有权限，而这个同义词又需要引用其他基表，那么用户在同义词上拥有的权限会转换为在基表上的权限。

缺省地，用户对他所创建的对象拥有完全的权限。如用户在用户模式上创建了一个表，那么缺省地，他在用户模式上拥有所创建的这个表的所有权限。

授予对象权限一般情况下是由对象所有者或者 DBA 用户来完成的；如果以其他用户身份授予对象权限，则要求用户必须具有该对象权限及转授对象权限选项（WITH GRANT OPTION），语法如下：

```
GRANT { object_priv [ (columns) ][ ,object_priv[(columns) ] ]...
| ALL [ PRIVILEGES ] } ON [schema.]object
TO {user | role | PUBLIC} [, {user | role | PUBLIC}]...
[ WITH GRANT OPTION ];
```

例：grant update on scott.emp to blake;

Grant update(sal) on emp to blake; (只能在 insert、update 和 references 上授予列权限)

2.9 撤消用户

DROP USER 语句用于删除用户。

语法：DROP USER username;

撤消用户时，用户创建的对象并没有被撤消。为了撤消创建对象的用户

语法：DROP USER username CASCADE;

(不指定 CASCADE 不能撤消创建对象的用户)

2.10 废除用户访问

一般情况下，收回**系统权限**是由 DBA 来完成的；如果以其他用户身份收回系统权限，则要求该用户必须具有相应系统权限及其转授系统权限选项(WITH ADMIN OPTION)。收回系统权限是使用 REVOKE 命令来完成的。语法如下：

```
REVOKE system_priv[,system_priv]...
FROM {user | role | PUBLIC }[, {user | role | PUBLIC }]...
```

用户的权限可使用 REVOKE 语句废除。一旦某个用户的权限被撤消，由他创建的所有用户和从他那获得权限的所有用户也都被撤消。如，Jim 把 CREATE TABLE 权限授给 John。当 Jim 的权限被撤消时，同时自动撤消 John 的权限。

如：GRANT CREATE TABLE TO John;

REVOKE CREATE TABLE FROM John; (不能撤消自己的权限)

一般情况下，收回对象权限是由对象所有者或者 DBA 用户来完成的；如果以其他用户身份收回对象权限，则要求用户必须具有该对象权限及转授对象权限选项 (WITH GRANT OPTION)。语法如下：

```
REVOKE { object_priv [ , object_priv ]... | ALL [ PRIVILEGES ] }
ON [ schema. ] object
FROM { user | role | PUBLIC }[, { user | role | PUBLIC }]...
[CASCADE CONSTRAINTS];
```

CASCADE CONSTRAINTS 用于删除任何与该对象相关的约束和对象，例如索引、触发器、权限、完整性约束等。

2.11 创建角色

角色是相关权限的命名集合，使用角色的主要目的是为了简化权限管理。角色可以是权限的组合，也可以是角色的组合。角色包括预定义角色和自定义角色两类。

常用的预定义角色有：(oracle 10g)

- CONNECT Role: 分配给临时用户的角色。通常，为只需要查询材料而无须创建表的分配这个角色。
- RESOURCE Role: 这个角色分配给常规用户
- DBA Role: 这个角色拥有一切系统权限，包括不加限制的表空间配额以及 WITH ADMIN OPTION 选项。默认的 DBA 用户为 SYS 和 SYSTEM，他们可

以将任何系统权限授予其他用户。读者需要注意，DBA 角色不具备 SYSDBA 和 SYSOPER 特权。

创建角色的语法:

CREATE ROLE role [NOT IDENTIFIED]; 不验证—用于公用角色或用户默认角色
或

CREATE ROLE role

IDENTIFIED BY password; 数据库验证

为角色分配密码或修改角色密码:

ALTER ROLE role

IDENTIFIED BY password; (password 为要设置的新密码)

如:

CREATE ROLE Acadre;

ALTER ROLE Acadre IDENTIFIED BY success;

为角色授予权限

GRANT privilege TO role;

例: 只授予 CONNECT 权限给角色 Acadre, 这样他们不可能操作数据库。

GRANT connect TO Acadre;

将角色指派给用户

一个角色可指派给多个用户。类似的, 一个用户也可具有多个角色。当把角色指派给用户时, 赋予该角色的权限也自动分配给用户。为用户分配角色的语法:

GRANT role TO user;

也可以使用 ALTER USER 语句授权用户。

ALTER USER John Default ROLE Student;

授予用户 John 以缺省角色 student。

2.12 新建的用户怎样才能成功创建一个表

当一个用户刚被创建时是不具备任何权限的, 因此要在该用户模式下创建表, 需授予 CREATE SESSION、CREATE TABLE、以及 UNLIMITED TABLESPACE (或分配配额) 权限, 因为: 当用户要连接到数据库时必须拥有 CREATE SESSION 权限

当用户要创建表时必须拥有 CREATE TABLE 权限, 同时用户还需要在表空间中拥有配额或者被授予 UNLIMITED TABLESPACE。现在我们来做一个测试:

1)、创建用户 TEST, 密码为 passwd_1:

```
SQL> CREATE USER test  
IDENTIFIED BY passwd_1
```

用户已创建

2) 当用 TEST 连接数据库时:

```
SQL> conn test/passwd_1
```

ERROR:

```
ORA-01045: user TEST lacks CREATE SESSION privilege; logon denied
```

警告: 您不再连接到 ORACLE。

//因为缺少 CREATE SESSION 的权限, 登陆失败。

3) 利用 SYS 给 TEST 授予 CREATE SESSION 权限:

```
SQL> grant create session to test;
```

授权成功。

4) SQL> conn test/passwd_1

已连接。

5) 在 test 的方案中创建表 exam1:

```
SQL> create table exam1
```

```
(student_id int,
```

```
paper_id int);
```

```
create table exam1
```

```
ERROR 位于第 1 行:
```

```
ORA-01031: 权限不足
```

```
//因为未给 TEST 用户授予 create table 权限, 因此不能够创建表 exam1.
```

6) 给 TEST 用户授予 CREATE TABLE 权限

```
SQL> grant create table to test;
```

授权成功。

7) SQL> create table exam1

```
(student_id int,
```

```
paper_id int);
```

```
create table exam1
```

```
ERROR 位于第 1 行:
```

```
ORA-01950: 表空间 'SYSTEM' 中无权限
```

```
//因为在创建用户时没有指定表空间, 因此默认的表空间是 SYSTEM 表空间, 而 TEST 用户还需要在表空间 SYSTEM 中既没有拥有配额又没有被授予 UNLIMITED TABLESPACE 权限, 因此对于这种情况有两种解决办法:
```

第一种方法:

```
SQL> alter user test
```

```
quota 15m on system;
```

用户已更改。

```
//在 SYSTEM 表空间中, 给用户 TEST 分配 15M 的使用空间
```

```
SQL> create table exam1
```

```
(student_id int,
```

```
paper_id int);
```

表已创建

第二种方法:

```
SQL> grant unlimited tablespace to test
```

授权成功。

```
SQL> create table exam2
```

```
(student_id int,
```

```
paper_id int);
```

第 3 章建立和管理表

本章主要内容:

- 表简介
- 创建表
- 修改表
- 截断和删除表
- 显示表的信息

3.1 简介

表是 Oracle 数据库最基本的对象，它用于存储用户数据。

- 设计表

当设计表时，需要考虑以下因素：

- ✓ 当规划表和列时，应该使用有意义的名称。当定义表名和列名时，只能使用字符（A-Z，a-z）、数字（0-9）、_、\$和#，名称必须以字符开始，并且长度不能超过 30 个字符。
- ✓ 当规划表名和列名时，要使用一致的缩写格式、单数或复数格式。
- ✓ 为了给用户和其他人员提供有意义的帮助信息，应该使用 COMMENT 命令描述表、列的作用
- ✓ 当设计表时，应该使用第一范式（1NF）、第二范式（2NF）和第三范式（3NF）规范化每张数据库表。
- ✓ 当定义表列时，应该选择合适的数据类型和长度。
- ✓ 当定义表列时，为了节省存储空间，应该将 NULL 列放在后面。

- 常用数据类型

当建立表时，不仅需要指定表名、列名，而且要根据情况为列选择合适的数据类型和长度。下面是常用数据类型。

- ✓ CHAR (N) 或 CHAR (N BYTE)：定义固定长度的字符串（以字节为单位），最大长度为 2000 字节。如果 CHAR (100) 或 CHAR (100 BYTE)，表示可存储 100 个字节的字符串，并且占用空间是固定的（100 个字节）
- ✓ CHAR (N CHAR)：定义固定长度的字符串（以字符个数为单位）。如果 CHAR (100 CHAR)，表示该列最多可以存储 100 个字符（单字节或多字节）。如果该列存放的全是汉字，则占用空间最多为 200 个字节；如果存放的全部是英文字符，则占用空间最多为 100 个字节。
- ✓ VARCHAR2 (N) 或 VARCHAR2 (N BYTE)：用于定义变长字符串（以字节为单位），其最大长度为 4000 字节。
- ✓ VARCHAR2 (N CHAR)：用于定义变长字符串（以字符为单位）。
- ✓ NUMBER (P, S)：定义数据类型的数据，P 表示数字的总位数（最大字节个数，而 S 表示小数点后面的位数。当定义整数类型时，可以直接使用 NUMBER 的子类型 INT。
- ✓ DATE：定义日期时间数据，其长度为 7 个字节。当查询 DATE 类型列时，其数据的默认显示格式为 (DD-MON-YY)，如“29-4 月-05”。
- ✓ TIMESTAMP：是 DATE 的扩展，在该数据类型上执行 DML 操作与 DATE 类型完全相同。但当查询时，数据的显示格式为 (DD-MON-YY HH.MI.SS AM)，如“29-4 月-03 04.02.03.000000 下午”。
- ✓ RAW(N)：定义二进制数据，N 的上限值为 2000。
- ✓ 大对象数据类型：早期版本（6，7）中，存储大批量字符数据采用数据类型 LONG，存储大批量二进制数据采用 LONG RAW 类型。从 8 版本开始，建议

使用 CLOB 存储大批量字符，建议使用 BLOB 类型存储大批量二进制数据。
下表列出了这些数据类型之间的区别：

LONG、LONG RAW	LOB(CLOB、BLOB)
表只能有一个 LONG 或 LONG RAW 列	表可以有多个 LOB 列
最大长度：2GB	最大长度：4GB
不支持对象类型	支持对象类型
存放在表段中	小于 4000：存放在表段中 大于 4000：存放到 LOB 段
SELECT：直接返回数据	SELECT：返回定位符
列数据顺序访问	列数据可以随机访问

✓ 伪列 ROWID 和 ROWNUM

➤ ROWID

用于唯一地标识表行，它间接给出了表行的物理位置，并且 ROWID 是定位表行最快的方式。如果某表包含了完全相同的行数据，为了删除重复行，那么可以考虑使用 ROWID 作为条件。当使用 INSERT 语句插入数据时，Oracle 会自动生成 ROWID，并将其值与表数据一起存放到表行。ROWID 与表列一样可以直接查询，如

```
SELECT dname,rowid FROM dept;
```

➤ ROWNUM

用于返回标识行数据顺序的数字值。当执行 SELECT 语句返回数据时，第 1 行的 ROWNUM 为 1，第 2 行的 ROWNUM 为 2，以此类推

3.2 建表

语法：CREATE TABLE [schema.]table_name(
Column_name datatype [DEFAULT expr]
[,...]
);

其中：schema 用于指定方案名（与用户名完全相同），table_name 用于指定表名，column_name 用于指定列名，datatype 用于指定列的数据类型，DEFAULT 子句用于指定列的默认值。（每张表最多可定义 1000 列）

[注释：查询表结构：DESC]

✓ 在当前方案中建表

```
CREATE TABLE dept01(  
dno NUMBER(2),name VARCHAR2(10),loc VARCHAR2(20)  
);
```

✓ 在其他方案中建表

```
CREATE TABLE scott.dept02(  
dno NUMBER(2),name VARCHAR2(10),loc VARCHAR2(20)  
);
```

✓ 在建表时为列指定默认值

```
CREATE TABLE scott.dept03(  
dno NUMBER(2),name VARCHAR2(10),
```

```
loc VARCHAR2(20) DEFAULT '呼和浩特'  
);
```

✓ 使用子查询建表

```
CREATE TABLE emp04(name,salary,job,dno) AS  
SELECT ename,sal,job,deptno FROM emp WHERE deptno=30;
```

✓ 建立临时表

临时表用于存放会话或事务的私有数据。临时表包括事务临时表和会话临时表两种类型，其中事务临时表是指数据只在当时事务内有效的临时表，会话临时表是指数据只在当前会话内有效的临时表。当建立临时表时，需要使用 **CREATE GLOBAL TEMPORARY TABLE** 命令。通过使用 **ON COMMIT DELETE ROWS** 选项可以指定事务临时表，通过使用 **ON COMMIT PRESERVE ROWS** 选项可以指定会话临时表，例：

```
CREATE GLOBAL TEMPORARY TABLE temp1(cola INT)  
ON COMMIT DELETE ROWS
```

说明：当执行了以上语句之后，会建立事务临时表 **TEMP1**。因为事务临时表的数据只在当前事务内有效，所以在事务结束之后会自动清除其数据。

3.3 修改表

如果表结构不符合实际情况，建表之后，可用 **ALTER TABLE** 改变表结构。

- 增加列

语法：ALTER TABLE table_name ADD(
column datatype [DEFAULT expr][,column datatype...]
);

如 ALTER TABLE emp01 ADD eno NUMBER(4);

- 修改列定义

语法：ALTER TABLE table_name MODIFY(
column datatype [DEFAULT expr][,column datatype...]
);

如 ALTER TABLE emp01 MODIFY job VARCHAR2(15);

- 删除列

语法：ALTER TABLE table_name DROP(column);

如：ALTER TABLE emp01 DROP COLUMN dno;

- 修改列名

语法：ALTER TABLE table_name
RENAME COLUMN column_name TO new_column_name;

如：ALTER TABLE emp01 RENAME COLUMN eno TO empno;

- 修改表名

语法：RENAME object_name TO new_object_name;

如：RENAME emp01 TO employee;

- 增加注释

语法：COMMENT ON TABLE table_name IS 'text';
COMMENT ON COLUMN table_name.column IS 'text';

如：COMMENT ON TABLE employee IS '存放雇员信息';
COMMENT ON COLUMN employee.name IS '描述雇员姓名';

3.4 截断和删除表

- 截断表

当表结构必须保留，而表数据不再需要时，可以使用 TRUNCATE TABLE 命令截断表。执行此命令时，会删除表的所有数据，并释放表所占用的空间，但会保留表的结构。

语法：TRUNCATE TABLE table_name;

说明：当删除表的所有数据时，既可以使用 DELETE 语句，也可以使用 TRUNCATE TABLE 命令。注意的是，DELETE (DML) 操作可以回退，但 TRUNCATE TABLE (DDL) 操作不能回退。

如：TRUNCATE TABLE employee;

- 删除表

当表不再需要时，可以使用 DROP TABLE 命令删除表。用此命令，不仅会删除表的所有数据，而且会删除表结构。

语法：DROP TABLE table_name [CASCADE CONSTRAINTS] [PURGE];

CASCADE CONSTRAINTS 用于指定级联删除从表的外键约束，PURGE 用于指定彻底删除表（这个选项是 10g 的新特征）

如：DROP TABLE employee;

- 恢复被删除表

当执行 DROP TABLE 语句删除表时，Oracle 会将删除表存放到数据库回收站。从 Oracle Database 10g 开始，使用 FLASHBACK TABLE 命令可以恢复被删除表。

语法：FLASHBACK TABLE table_name TO BEFORE DROP;

3.5 显示表信息

- USER_TABLES

当建立表时，Oracle 会将表信息存放到数据字典。通过查询数据字典视图 USER_TABLES，可以显示当前用户的所有表信息。

如：conn scott/tiger

```
SELECT table_name FROM user_tables;
```

- USER_OBJECTS

当建立数据库对象（表、视图、索引等）时，Oracle 会将对象信息存放到数据字典中。通过查询数据字典视图 USER_OBJECTS，可显示所有数据库对象。

如：SELECT object_name FROM user_objects

```
WHERE object_type='TABLE';
```

- USER_TAB_COMMENTS

当执行 COMMENT 命令为表、视图增加注释信息时，Oracle 会将注释存放到数据字典中。通过查询数据字典视图 USER_TAB_COMMENTS，可以显示当前用户所有表的注释。

如：SELECT comments FROM user_tab_comments

```
WHERE table_name='EMPLOYEE';
```

- USER_COL_COMMENTS

当执行 COMMENT 命令为列增加注释信息时，Oracle 会将注释存放到数据字典中。通过查询数据字典视图 USER_COL_COMMENTS，可以显示当前用户所有表的列注释。

如：SELECT comments FROM user_col_comments

```
WHERE table_name='EMPLOYEE' AND column_name='NAME';
```

[注释：表名及列名均大写，因为 Oracle 中是以大写字母存储对象名及列名

第 4 章 Oracle 操作符

Oracle 中有很多的操作符，每种操作符都有自己的含义，在使用时需要很好的理解其中的内涵。这些操作符与平时大家见到的一些操作符几乎是一样的，含义也差不多。需要注意一点的是，oracle 中的赋值语句用:=的方式，而不是=。

操作符	使用	描述	例子
=	$a = b$	测试两个操作数相等	SELECT * FROM emp WHERE sal = 500
!=	$a != b$	测试两个操作数不相等	SELECT * FROM emp WHERE sal != 500
^=	$a ^= b$	测试两个操作数不相等	SELECT * FROM emp WHERE sal ^= 500
<>	$a <> b$	测试两个操作数不相等	SELECT * FROM emp WHERE sal <> 500
<	$a < b$	测试操作数 a 比操作数 b 小	SELECT * FROM emp WHERE sal < 500
!<	$a !< b$	测试操作数 a 不小于操作数 b 。这与 \geq 相同	SELECT * FROM emp WHERE sal !< 500
>	$a > b$	测试操作数 a 比操作数 b 大	SELECT * FROM emp WHERE sal > 500
!>	$a !> b$	测试操作数 a 不大于操作数 b 。这与 \leq 相同	SELECT * FROM emp WHERE sal !> 500
<=	$a <= b$	测试操作数 a 小于等于操作数 b	SELECT * FROM emp WHERE <= 500
>=	$a >= b$	测试操作数 a 大于等于操作数 b	SELECT * FROM emp WHERE >= 500

操作符	使用	描述	例子
IN	$a \text{ IN } (b,c,\dots)$	测试操作数 a 在提供的列表中（操作数 b 、操作数 c 等等）至少有一个匹配值	<pre>SELECT * FROM emp WHERE sal IN (500,600,700)</pre>
NOT IN	$a \text{ NOT IN } (b,c,\dots)$	表示操作数 a 与提供的列表中的元素（操作数 b 、操作数 c 等等）没有匹配值	<pre>SELECT * FROM emp WHERE sal NOT IN (500,600,700)</pre>
ANY	$a = \text{ANY } (b,c,\dots)$ $a < \text{ANY } (b,c,\dots)$ $a > \text{ANY } (b,c,\dots)$ 等等	测试指定的关系（例如，=、<>、<、> 等等）在提供的元素清单（操作数 b 、操作数 c 等等）中至少有一个是真。当测试相等时，就等于 IN 的功能	<pre>SELECT * FROM emp WHERE sal = ANY (500,600,700)</pre>
SOME	$a = \text{SOME } (b,c,\dots)$ $a < \text{SOME } (b,c,\dots)$ $a > \text{SOME } (b,c,\dots)$ 等等	测试指定的关系（例如，=、<>、<、> 等等）在提供的元素清单（操作数 b 、操作数 c 等等）中至少有一个是真。当测试相等时，就等于 IN 的功能	<pre>SELECT * FROM emp WHERE sal = SOME (500,600,700)</pre>
ALL	$a = \text{ALL } (b,c,\dots)$ $a < \text{ALL } (b,c,\dots)$ $a > \text{ALL } (b,c,\dots)$ 等等	测试指定的关系（例如，=、<>、<、> 等等）对提供的所有元素清单（操作数 b 、操作数 c 等等）都为真	<pre>SELECT * FROM emp WHERE sal = ALL (500,600,700)</pre>
BETWEEN	$a \text{ BETWEEN } b \text{ and } c$	测试操作数 a 大于等于操作数 b 并小于等于操作数 c	<pre>SELECT * FROM emp WHERE sal BETWEEN 400 AND 600</pre>
NOT BETWEEN	$a \text{ NOT BETWEEN } b \text{ and } c$	测试操作数 a 小于等于操作数 b 并大于等于操作数 c	<pre>SELECT * FROM emp WHERE sal NOT BETWEEN 400 AND 600</pre>

操作符	使用	描述	例子
EXISTS	EXISTS(<i>query</i>)	测试该查询至少返回一行	<pre>SELECT * FROM emp WHERE EXISTS (SELECT deptno FROM dept d WHERE deptno= e.deptno)</pre>
NOT EXISTS	NOT EXISTS(<i>query</i>)	测试查询不返回一行	<pre>SELECT * FROM emp WHERE NOT EXISTS (SELECT deptno FROM dept d WHERE deptno= e.deptno)</pre>
LIKE	<i>a</i> LIKE <i>b</i>	测试操作数 <i>a</i> 匹配操作数 <i>b</i> 的模式。模式可以包含 <code>_</code> 匹配该位置的单个字符，或者用 <code>%</code> 匹配所有的字符	<pre>SELECT * FROM emp WHERE ename LIKE 'SMI%'</pre>
NOT LIKE	<i>a</i> NOT LIKE <i>b</i>	测试操作数 <i>a</i> 不匹配操作数 <i>b</i> 的模式。模式可以包含 <code>_</code> 匹配该位置的单个字符，或者用 <code>%</code> 匹配所有的字符	<pre>SELECT * FROM emp WHERE ename NOT LIKE 'SMI%'</pre>
IS NULL	<i>a</i> IS NULL	测试操作数 <i>a</i> 为 NULL	<pre>SELECT * FROM emp WHERE comm. IS NULL</pre>
IS NOT NULL	<i>a</i> IS NOT NULL	测试操作数 <i>a</i> 不为 NULL	<pre>SELECT * FROM emp WHERE comm. IS NOT NULL</pre>

操作符	描述	例子
UNION	组合每个查询返回的所有行，并删除重复行	<pre>SELECT * FROM emp WHERE deptno=10 UNION SELECT * FROM emp WHERE sal > 500</pre>
UNION ALL	组合每个查询返回的所有行，并包括重复行	<pre>SELECT * FROM emp WHERE deptno=10 UNION ALL SELECT * FROM emp WHERE sal > 500</pre>
MINUS	采用第一个查询返回的行，减去第二个查询中也同样返回的行，最后返回剩下的行	<pre>SELECT * FROM emp MINUS SELECT * FROM emp WHERE sal > 500</pre>
INTERSECT	只返回两个查询共同返回的行	<pre>SELECT * FROM emp WHERE deptno = 10 INTERSECT SELECT * FROM emp WHERE SAL>500</pre>

第 5 章 基本查询

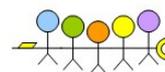
- 简单查询语句
- 限制数据
- 排序数据

5.1 简单查询语句

查询所有列: `select * from dept;`

查询指定列: `desc emp(显示表结构) select empno,ename,sal from emp;`

查询日期列: 日期列是指 `date` 类型列，默认显示格式为 `dd-mon-yy`，不同语言和地区的日期显示结果有所不同。如果想以自己习惯的日期格式显示日期值，必须要用 `to_char` 函数进行转换。当日期语言为 `SIMPLIFIED CHINESE` 时，



格式: 17-12 月-80

```
Select ename,to_char(hiredate,'YYYY-MM-DD' from emp;
```

上述语句显示格式: 1980-12-17

取消重复行: 默认会显示所有行, 但完全相同的查询结果没有实际意义, 因此有时需要取消重复结果

```
Select distinct deptno,job from emp;(显示三列都不相同的记录集)
```

```
Select distinct deptno;(只显示 deptno 不同的记录集)
```

使用算术表达式: 当招行查询操作时, 可在数字列上用算术表达式(+,-,*,/)

```
Select ename,sal,sal*12 from emp;
```

使用列别名: 默认情况下, 列标题是大写格式的列名或表达式, 如果使用列别名, 列别名可跟在列名后, 并且在二者之间可以加 AS 关键字, 若列别名区分大小写、包含特殊字符或空格, 必须用双引号引住

```
Select ename as name,sal*12 "Annual Salary" from emp;
```

处理 NULL: NULL 表示未知值, 既不是空格也不是 0。若没为列提供数据且该列无默认值, 则其数据为 NULL。当算术表达式包含 NULL 时, 其结果也是 NULL。

```
Select ename,sal,comm,sal+comm from emp;
```

```
Select ename,sal,comm,sal+nvl(comm,0) from emp;
```

Nvl(comm,0)说明: 如果 comm 存在数值, 则函数返回其原有数值; 如果 comm 列为 NULL, 则函数返回 0。

连接字符串: 连接字符串是使用||操作符完成的。如果在字符串中要加入数字值, 那么在||后可以直接指定数字; 如果在字符串中加入字符和日期值, 则必须用单引号引住。

```
Select ename||'的岗位是'||job "Employee" from emp;
```

```
或 Select ename||'的岗位是'||job AS Employee from emp;
```

[注释: dual 是一个虚拟表, 用来查那些不属于实际表里的内容, 如:

```
select sysdate from dual; select 3+3 from dual;]
```

1. 限制数据 (条件查询)

条件查询中条件表达式中需要使用各种比较操作符, 如 (=, <>(!=), >=, <=, >, <, BETWEEN...AND..., IN(list), LIKE, IS NULL)

使用数值值: `select ename,sal from emp where sal>2000;`

使用字符值: `select ename,sal from emp where job='MANAGER';`//字符值必须用单引号引住

需要注意: 字符值区分大小写

使用日期值: 在 where 条件中使用日期值时, 必须用单引号引住, 并且日期必须符合默认日期显示格式和日期语言。如果不符合默认日期显示格式, 必须使用 to_date 函数进行转换。

```
Select ename,sal,hiredate from emp where hiredate>'01-1 月-82';
```

使用 BETWEEN...AND 操作符:

```
select ename,sal,job from emp where sal between 2000 and 4000
```

使用 LIKE 操作符: 执行模糊查询。%: 通配 0 或多个字符 _: 通配单个字符

```
Select ename,sal from emp where ename like 'S%'//显示首字符为 S 的所有雇员名及其工资
```

```
Select ename,sal from emp where ename like '__o%';
```

当希望使用这两个字符(%,_)招行模糊查询时, 必须使用 ESCAPE

选项和转义符实现。当使用 ESCAPE 选项时，需要在%或_前加转义符，并且在 ESCAPE 选项中指定转义符的名称。例：

```
Select ename,sal from emp where ename like '%a_%' ESCAPE 'a';
```

使用 IN 操作符：select ename,sal,job from emp where job in('CLERK','MANAGER')

使用 IS NULL 操作符：select ename from emp where mgr IS (NOT) NULL;

使用 AND, OR, NOT 操作符：

```
Select ename,sal,job,deptno from emp where deptno=20 and job='CLERK';
```

```
Select ename,sal,job,deptno from emp where sal>2500 or job='MANAGER'
```

```
Select ename,sal,comm,deptno from emp where comm is not null;
```

5.2 排序数据

执行查询操作时，默认情况下 Oracle 以无序方式显示数据。为了更直观地显示数据结果，在实际应用中经常对数据进行排序

升序排序：默认数据以升序方式排列，当以特定列执行升序排序时，如果排序列存在 NULL 行，那么 NULL 行会显示在最后面。

```
Select ename,sal from emp order by sal;
```

降序排序：必须指定 DESC 关键字，如果排序列存在 NULL 行，那么 NULL 行会显示在最前面。

```
Select ename,sal from emp order by sal desc;
```

使用多列排序：当以多列进行排序时，首先按照第一列进行排序，当第一列存在相同数据时，然后以第二列进行排序。

```
SELECT ename,deptno,sal FROM emp ORDER BY deptno ASC,sal DESC;
```

使用非选择列进行排序：执行排序操作时，多数情况下会在选择列表中包含排序列，但选择列表也可以不包含任何排序列。

```
SELECT ename FROM emp ORDER BY sal DESC;
```

使用列别名排序：如果在选择列表中为列或表达式定义了别名，那么当执行排序操作时，既可以使用列或表达式进行排序，也可以使用列别名进行排序。

```
SELECT ename,sal*12 年收入 FROM emp ORDER BY 年收入 DESC;
```

使用列位置排序：排序操作不仅可以指定列名、列别名，也可以按照列或表达式在选择列表中的位置进行排序。另外当使用 UNION、UNION ALL、INTERSECT、MINUS 等集合操作符合并查询结果时，如果选择列表的列名不同，并且希望进行排序，必须使用列位置进行排序。

```
SELECT deptno,dname FROM dept
```

```
UNION
```

```
SELECT empno,ename FROM emp ORDER BY 1;
```

第 6 章 复杂查询

- 数据分组
- 连接查询
- 子查询
- 使用集合操作符

6.1 数据分组

实际应用中，经常需要对数据进行统计。当统计数据时，需要将表的数据划分成几个组，最终统计每个组的数据结果。

1) 数组函数

分组函数用于统计表的数据，与单行函数不同，分组函数作用于多行，并返回一个结果，所以有时也称为多行函数。一般情况下，分组函数要与 **GROUP BY** 子句结合使用。在使用分组函数时，如果忽略了 **GROUP BY** 子句，那么会汇总所有行，并产生一个结果。

- ✓ **MAX** 和 **MIN**：用于取得列或表达式的最大值和最小值。
`SELECT max(sal) 最高工资, min(sal) 最低工资 FROM emp;`
- ✓ **AVG** 和 **SUM**：用于取得表达式的平均值或总和
`SELECT avg(sal) 平均工资, sum(sal) 总计工资 FROM emp;`
- ✓ **COUNT**：用于取得总计行数
`SELECT count(*) 雇员总数 FROM emp;`
[说明]count 函数中还可以引用表达。因为分组函数会忽略 **NULL** 行，所以使用 `count(表达式)` 会显示非 **NULL** 的总计行数。如：
`SELECT count(comm) 补助非空的雇员总数 FROM emp;`
- ✓ **VARIANCE** 和 **STDDEV**：用于取得列或表达式的方差和标准偏差。
`SELECT variance(sal) 方差, stddev(sal) 标准偏差 FROM emp;`

当使用分组函数时，有一些注意事项。

- 分组函数只能出现在选择列表、**ORDER BY** 子句、**HAVING** 子句中。
- 当使用分组函数时，会忽略 **NULL** 行。
- 如果在选择列表中既包含了分组函数，也包含了其他列和表达式，那么这些列或表达式必须出现在 **GROUP BY** 子句中。
- 当使用分组函数时，可以在函数中指定 **ALL** 和 **DISTINCT** 选项。其中 **ALL** 是默认选项，该选项表示统计所有行数据（包括重复值）；如果指定 **DISTINCT**，则只会统计不同行值。下面是常用的分组函数：

2) **GROUP BY** 和 **HAVING** 子句

GROUP BY 子句用于对查询结果进行分组统计，**HAVING** 子句用于限制分组显示结果。如果选择列表同时包含有列、表达式和分组函数，那么这些列和表达式必须出现在 **GROUP BY** 子句中。语法如下：

```
SELECT column,group_function FROM table
[WHERE condition]
[GROUP BY group_by_expression]
[HAVING group_condition]
```

[ORDER BY expression];

- 使用 GROUP BY 进行单列分组
单列分组是指基于单列生成分组统计结果。当进行单列分组时，会基于分组列的每个不同值生成一个统计结果。
如：SELECT deptno 部门代码, avg(sal) 部门平均工资 FROM emp
GROUP BY deptno;
- 使用 GROUP BY 进行多列分组
多列分组是指基于两个或两个以上的列生成分组统计结果。当进行多列分组时，会基于多个列的不同值生成统计结果。
如：SELECT deptno,job,avg(sal),max(sal) FROM emp
GROUP BY deptno,job;
- 使用 HAVING 子句限制分组显示结果
HAVING 子句用于限制分组统计结果，并且 HAVING 子句必须跟在 GROUP BY 子句后面。
如：SELECT deptno,avg(sal),max(sal) FROM emp
GROUP BY deptno
HAVING avg(sal)<2500;
- 使用 ORDER BY 子句改变分组排序结果
当使用 GROUP BY 子句执行分组统计时，默认情况下会以分组列的升序显示统计结果。如果要改变统计结果的显示顺序，那么需要使用 ORDER BY 子句
如：SELECT deptno,avg(sal) FROM emp
GROUP BY deptno ORDER BY avg(sal);
- 使用数据分组的注意事项
当执行数据统计时，需要注意正确使用 GROUP BY 子句、WHERE 子句和分组函数。具体要求如下：
 - ✓ 分组函数只能出现在选择列表、HAVING 子句和 ORDER BY 子句中
 - ✓ 如果在 SELECT 语句中同时包含有 GROUP BY、HAVING 以及 ORDER BY 子句，那么必须将 ORDER BY 子句放在最后。
 - ✓ 如果选择列表包含有列、表达式和分组函数，那么这些列和表达式必须出现在 GROUP BY 子句中，否则会显示错误信息。
如：SELECT deptno,job,avg(sal) FROM emp
GROUP BY deptno; (错误，因为 job 未出现在 GROUP BY 子句中)
 - ✓ 当限制分组显示结果时，必须使用 HAVING 子句，而不能在 WHERE 子句中使用分组函数，否则会显示错误信息。
如：SELECT deptno, AVG(sal) FROM emp
WHERE SUM(sal)>1000
GROUP BY deptno (错误：分组函数不能用在 WHERE 中)

6.2 连接查询

连接查询是指基于两个或两个以上表或视图的查询。在实际应用中，查询单个表可能无法满足应用程序的实际需求（例如显示 SALES 部门位置以及雇员名）。使用连接查询时的注意事项：

- 当使用连接查询时，必须在 FROM 子句后指定两个或两个以上的表。

- 当使用连接查询时，应该在列名前加表名作为前缀。但如果不同表之间列名不同，可以不加表名作前缀。当使用连接查询时，必须在 **WHERE** 子句中指定有效的连接条件，否则会导致生成笛卡儿积。
- 如：`SELECT dept.dname,emp.ename FROM dept,emp WHERE ept.dname='SALES';`
- 当进行连接查询时，使用表别名可以简化连接查询语句。
如：`SELECT d.dname,e.ename FROM dept d,emp e
WHERE d.deptno=e.deptno;`

6.3 相等连接（包括 SQL: 1999 标准内连接、自然连接）

相等连接是指使用相等比较符 (=) 指定连接条件的连接查询，该类连接查询主要用于检索主从表之间的相关数据。语法：

```
SELECT table1.column,table2.column FROM table1,table2  
WHERE table1.column1=table2.column2.;
```

如：`SELECT e.ename,e.sal,d.dname FROM emp e,dept d
WHERE e.deptno=d.deptno(AND d.deptno=10);`

6.4 不等连接

指在连接条件中使用除相等比较符外其他比较操作符的连接查询。主要用于在不同表之间显示特定范围的信息。

如：`SELECT a.ename,a.sal,b.grade FROM emp a,salgrade b
WHERE a.sal BETWEEN b.losal AND b.hisal`

6.5 自连接

自连接是指在同一张表之间的连接查询，它主要用在自参照表上显示上下级关系或者层次关系。自参照表是指在不同列之间具有参照关系或主从关系的表。如下表，

EMPNO (雇员号)	ENAME	MGR (管理者号)
7839	KING	
7566	JONES	7839
7698	BLAKE	7839
7782	CLARK	7839
...

根据 EMPNO 列和 MGR 列的对应关系，可以确定雇员 JONES、BLAKE 和 CLARK 的管理者为 KING。为了显示雇员及其管理者之间的对应关系，可以使用自连接。因为自连接是在同一张表之间的连接，所以必须定义表别名。

如：`SELECT manager.ename FROM emp manager,emp worker
WHERE manager.empno=worker.mgr
AND worker.ename='BLAKE';`

6.6 [内连接和]外连接

内连接用于返回满足连接条件的记录；而外连接则是内连接的扩展，它不仅会返回满足连接条件的所有记录，而且还会返回不满足连接条件的记录。在 Oracle Database 9i 之前，连接语法都是在 **WHERE** 子句中指定的 (+号实现)；从 9i 开始，有专用语法 (SQL: 1999 标准)，格式如下：

```
SELECT table1.column,table2.column
FROM table1 [INNER | NATURAL | LEFT | RIGHT | FULL] JOIN table2
ON table1.column1=table2.column2;
```

6.6.1 内连接

用于返回满足连接条件的所有记录。默认情况下，在执行连接查询时如果没有指定任何连接操作符，那么这些连接查询都属于内连接。

如：SELECT a.dname,b.ename FROM dept a INNER JOIN emp b
ON a.deptno=b.deptno AND a.deptno=10;

从 Oracle Database 9i 开始，如果主表的主键列和从表的外部键列名称相同，那么还可以使用 NATURAL JOIN 关键字自动执行内连接操作。

如：SELECT dname,ename FROM dept NATURAL JOIN emp;

6.6.2 左外连接

不仅返回满足连接条件的所有记录，而且还会返回不满足连接条件的连接操作符左边表的其他行。

如：SELECT a.dname,b.ename FROM dept a LEFT JOIN emp b
ON a.deptno=b.deptno AND a.deptno=10

6.6.3 右外连接

不仅返回满足连接条件的所有记录，而且还会返回不满足连接条件的连接操作符右边表的其他行。

6.6.4 完全外连接

不仅返回满足连接条件的所有记录，而且还会返回不满足连接条件的所有其他行。

注释：外连接可以使用 (+) 操作符，但不建议用。

6.7 子查询

子查询是指嵌入在其他 SQL 语句中的 SELECT 语句，也称为嵌套查询。当在 DDL 语句中使用子查询时，可以带有 ORDER BY 子句；但如果在 WHERE 子句、SET 子句中使用子查询，不能带有 ORDER BY 子句。子查询具有以下作用：

- 通过在 INSERT 或 CREATE TABLE 语句中使用子查询，可以将源表数据追加到目标表。
- 通过在 UPDATE 语句中使用子查询可以修改一列或多列的数据
- 通过在 WHERE、HAVING 子句中使用子查询，可以提供条件值。

根据子查询返回结果的不同，子查询又分为单行子查询、多行子查询和多列子查询。

6.7.1 单行子查询（单列）

指只返回一行数据的子查询语句。当在 WHERE 子句中引用单行子查询时，可以使用单行比较符 (=, <, >, <>, <=>, >=, <>=)。

如：SELECT ename,sal,deptno FROM emp
WHERE deptno=(SELECT deptno FROM emp WHERE ename='SCOTT');

6.7.2 多行子查询（单列）

指返回多行数据的子查询语句。当在 WHERE 子句中使用多行子查询时，必须使用多行比较符 (IN, ALL, ANY)。

IN：匹配于子查询结果的任一个值即可

```
SELECT ename,job,sal,deptno FROM emp WHERE job IN
(SELECT distinct job FROM emp WHERE deptno=10);
```

ALL：必须要符合子查询结果的所有值

```
SELECT ename,sal,deptno FROM emp WHERE sal>all
(SELECT sal FROM emp WHERE deptno=30);
```

ANY: 只要符合子查询结果的任一个值即可

```
SELECT ename,sal,deptno FROM emp WHERE sal>ANY
(SELECT sal FROM emp WHERE deptno=30)
```

6.7.3 多列子查询

单行子查询是指子查询只返回单列单行数据,多行子查询是指子查询返回单列多行数据,二者都是针对单列而言的。

而多列子查询则是指返回多个列数据的子查询语句。当多列子查询返回单行数据时,在 WHERE 子句中可以使用单行比较符;当多列子查询返回多行数据时,有 WHERE 子句中必须使用多行比较符 (IN、ANY、ALL)。

如: SELECT ename,job,sal,deptno FROM emp WHERE (deptno,job)=(SELECT deptno,,job FROM emp WHERE ename='SMITH');

当使用子查询比较多个列的数据时,既可以使用成对比较,也可以使用非成对比较

成对比较示例

```
SELECT ename,sal,comm,deptno FROM emp WHERE(sal,nvl(comm,-1)) IN
(SELECT sal,nvl(comm,-1) FROM emp WHERE deptno=30);
```

非成对比较示例

```
SELECT ename,sal,comm,deptno FROM emp WHERE sal IN (SELECT sal
FROM emp WHERE deptno=30) AND nvl(comm,-1) IN (SELECT nvl(comm,-1)
FROM emp WHERE deptno=30);
```

6.7.4 其他子查询

6.7.4.1 相关子查询 (通过 EXISTS 谓词实现)

```
SELECT ename,job,sal FROM emp WHERE EXISTS
(SELECT 1 FROM dept WHERE dept.deptno=emp.deptno AND dept.loc='NEW
YORK');
```

显示工作在 NEW YORK 的所有雇员的姓名,工作,工资及部门号)

6.7.4.2 在 FROM 子句中使用子查询 (子查询作为视图对待,又称内嵌视图)

```
SELECT ename,job,sal FROM emp,(SELECT deptno,avg(sal) avgsal FROM emp
GROUP BY deptno) d
WHERE emp.deptno=d.deptno and sal>d.avgsal;
```

注: 当在 FROM 子句中使用子查询时,必须给予查询指定别名。

6.7.4.3 在 DML 语句中使用子查询

6.7.4.3.1 在 INSERT 语句中使用子查询

可以将一张表的数据装载到另一张表。

如: INSERT INTO employee(id,name,title,salary) SELECT empno,ename,job,sal FROM emp;

6.7.4.3.2 UPDATE 中

6.7.4.3.3 DELETE 中

6.7.4.4 在 DDL 语句中使用子查询

6.7.4.4.1 在 CREATE TABLE 语句中使用子查询

可以在建立新表的同时复制表数据

如: CREATE TABLE new_emp(id,name,sal,job,deptno) AS SELECT

```
empno,ename,sal,job,deptno FROM emp;
```

6.7.4.4.2 在 CREATE VIEW 语句中使用子查询

```
CREATE OR REPLACE VIEW dept_10 AS SELECT empno,ename,  
job,sal,deptno FROM emp WHERE deptno=10 ORDER BY empno
```

6.8 使用集合操作符

为了合并多个 SELECT 语句的结果，可以使用集合操作符 UNION、UNION ALL，语法如下：

```
SELECT 语句 1  
[UNION | UNION ALL]  
SELECT 语句 2
```

- UNION

用于取得两个结果集的并集，自动去掉结果集中的重复行。

如：SELECT ename,sal,job FROM emp WHERE sal>2500

```
UNION
```

```
SELECT ename,sal,job FROM emp WHERE job='MANAGER'
```

- UNION ALL

用于取得两个结果集的并集，与 UNION 操作符不同，该操作符不会取消重复值

第 7 章 常用 SQL 函数

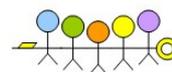
SQL 函数是 Oracle 数据库的内置函数，并且可以在各种 SQL 语句中使用。当单独调用 SQL 函数时，可以使用数据字典 DUAL，该数据字典专门用于取得函数返回值。SQL 函数包括单行函数和多行函数，其中单行函数是指输入一行输出一行的函数；而多行函数是指输入多行输出一行的函数。按照处理功能分类，SQL 函数分为数字函数，字符函数，日期时间函数，转换函数及其它类型函数。

本章介绍常用各种类型的单行函数

1. ASCII

返回与指定的字符对应的十进制数；

```
SQL> select ascii('A') A,ascii('a') a,ascii('0') zero,ascii(' ') space from dual;
```



2. CHR

给出整数, 返回对应的字符;

```
SQL> select chr(54740) zhao, chr(65) chr65 from dual;
```

3. CONCAT 等价与||, 推荐 CONCAT

连接两个字符串;

```
SQL> select concat('010-', '88888888') || '转 23' 联系电话 from dual;
```

4. INITCAP

返回字符串并将字符串的第一个字母变为大写;

```
SQL> select initcap('smith') upp from dual;
```

5. INSTR(C1, C2[, I[, J]]) 和 INSTRB(C1, C2[, I[, J]])

在一个字符串中搜索指定的字符, 返回发现指定的字符的位置;

C1 被搜索的字符串

C2 希望搜索的字符串

I 搜索的开始位置, 默认为 1

J 子串的第 J 次出现的位置, 默认为 1

```
SQL> select instr('oracle traning', 'ra', 1, 2) instrstring from dual;
```

6. LENGTH 和 LENGTHB

返回字符串的长度; LENGTHB 按照字节进行返回

```
SQL> select name, length(name), addr, length(addr), sal, length(to_char(sal)) from  
gao.nchar_tst;
```

7. LOWER

返回字符串, 并将所有的字符小写

```
SQL> select lower('AaBbCcDd') AaBbCcDd from dual;
```

8. UPPER

返回字符串, 并将所有的字符大写

```
SQL> select upper('AaBbCcDd') upper from dual;
```

9. RPAD 和 LPAD(粘贴字符) 字符串填充

RPAD 在列的右边粘贴字符

LPAD 在列的左边粘贴字符

LPAD(char1, n, char2): 在字符串 char1 的左端填充字符串 char2, 直至字符串总长度为

n, char2 的默认值为空格, 如果 char1 长度大于 n, 则该函数返回 char1 左端的前 n 个字符。

如果输入参数值存在 NULL, 则返回 NULL

```
SQL> select lpad(rpad('gao', 10, '*'), 17, '*') from dual;
```

10. LTRIM 和 RTRIM

LTRIM 删除左边出现的字符串

RTRIM 删除右边出现的字符串

LTRIM(char1[, set]): 去掉 char1 左端所包含的 set 中的任何字符。Oracle 从左端第一个字符开始扫描, 逐一去掉在 set 中出现的字符, 当遇到不是 set 中的字符时终止, 然后返回剩余结果。

```
SQL> select ltrim(rtrim(' gao qian jing ', ' '), ' ') from dual;
```

11. SUBSTR(string, start[, count]) 和 SUBSTRB(string, start[, count])

取子字符串, 从 start 开始, 取 count 个。如果 m 为 0, 则从首字符开始; 如果 m 是负数, 则从尾部开始。

```
SQL> select substr('13088888888', 3, 8) from dual;
```

12. REPLACE('string', 's1', 's2')

string 用于指定字符串

s1 用于指定要被替换的子串

s2 用于指定替换后的子串

如果 s1 为 NULL, 则返回原有字符串, 如果 s2 为 NULL, 则会去掉指定子串。

```
SQL> select replace('he love you', 'he', 'i') from dual;
```

13. SOUNDEx(char)

用于返回字符串的语音表示, 参数 char 用于指定英文字符串, 通过使用函数 SOUNDEx, 可以比较两个英文单词的发音是否相同。

```
SQL> create table table1(xm varchar(8));
```

```
SQL> insert into table1 values('weather');
```

```
SQL> insert into table1 values('wether');
```

```
SQL> insert into table1 values('gao');
```

```
SQL> select xm from table1 where soundex(xm)=soundex('weather');
```

```
14. TRIM({trim_char|LEADING trim_char|TRAILING trim_char|BOTH trim_char} FROM trim_source);
```



用于从字符串中截断特定字符。参数 trim_char 用于指定要截去的字符，参数 trim_source 用于指定源字符串，LEADING 用于指定截去头部字符，TRAILING 用于指定截去尾部字符，BOTH 用于指定截去头部和尾部字符（默认选项）

```
Select TRIM('s' from 'string') from DUAL;
```

15. ABS(n)：用于返回数字 n 的绝对值，如果输入为 NULL，则返回值也是 NULL

返回指定值的绝对值

```
SQL> select abs(100),abs(-100) from dual;
```

16. ACOS(n)

给出反余弦的值，输入值的范围是 $-1 \sim 1$ ，输出值为弧度，如果输入 NULL，返回 NULL

```
SQL> select acos(-1) from dual;
```

17. ASIN(n)

给出反正弦的值，输入值的范围是 $-1 \sim 1$ ，输出值为弧度，若输入为 NULL，则返回 NULL。

```
SQL> select asin(0.5) from dual;
```

18. ATAN(n)

返回一个数字的反正切值，输入值可以是任何数字，输出值为弧度，若输入为 NULL，则返回 NULL

```
SQL> select atan(1) from dual;
```

19. CEIL(n)

返回大于等于数字 n 的最小整数，若输入 NULL，则返回为 NULL

```
SQL> select ceil(3.1415927) from dual;
```

20. COS(n)

返回数字 n（以弧度表示的角度值）的余弦值，若输入 NULL，则返回 NULL

```
SQL> select cos(-3.1415927) from dual;
```

21. ACOS(n)

返回一个数字反余弦值，输入值的范围是 $-1 \sim 1$ ，输出值为弧度。如果输入值为 NULL，则返回值也为 NULL。

```
SQL> select acos(1) from dual;
```

22. EXP(n)

返回一个数字 e 的 n 次方根

```
SQL> select exp(2),exp(1) from dual;
```

23. FLOOR

返回小于等于数字 n 的最大整数，若输入 NULL，则返回 NULL

```
SQL> select floor(2345.67) from dual;
```

24. LN(n)

返回数字 n 的自然对数值，其中 n 必须大于 0，如果输入 NULL，则返回 NULL

```
SQL> select ln(1),ln(2),ln(2.7182818) from dual;
```

25. LOG(n1, n2)

返回一个以 n1 为底 n2 的对数

```
SQL> select log(2, 1),log(2, 4) from dual;
```

26. MOD(n1, n2)

返回一个 n1 除以 n2 的余数

```
SQL> select mod(10, 3),mod(3, 3),mod(2, 3) from dual;
```

27. POWER(n1, n2)

返回 n1 的 n2 次方根，n1, n2 可以是任意数字。但如果数字 m 为负数，则数字 n 必须是正数。若输入为 NULL，则返回 NULL。

```
SQL> select power(2, 10),power(3, 3) from dual;
```

28. ROUND 和 TRUNC(x, m, n)

ROUND(n[, m]): 返回四舍五入，其中 n 可为任意数字，m 必须为整数。如果省略 m，则四舍五入到整数位；如果 m 是负数，则四舍五入到小数点前的第 m 位；如果 m 为正数，那么四舍五入到小数点后的第 m 位。若输入 NULL，则输出 NULL

TRUNC(n[, m]): 该函数用于截取数字，其中 n 可以是任意数字，m 必须是整数。若 m 省略，则会将数字 n 的小数部分截去；如果数字 m 是正数，那么会将数字 n 截取至小数点后的第 m 位；如果数字 m 是负数，那么会将数字 n 截取至小数点前的第 m 位。

```
SQL> select round(55.5),round(-55.4),trunc(55.5),trunc(-55.5) from dual;
```

29. SIGN(n) 符号函数

用于检测数字的正负。如果数字 n 小于 0，则函数的返回值为 -1；如果数字 n 等于 0，则返回 0；如果大于 0，则函数的返回值为 1。若输入值为 NULL，则返回 NULL

```
SQL> select sign(123),sign(-100),sign(0) from dual;
```

30. SIN(n)

返回数字 n（以弧度表示的角）的正弦值，输入 NULL，则输出 NULL

```
SQL> select sin(1.57079) from dual;
```

31. SINH(n)

返回数字 n 的双曲正弦值，输入 NULL，则输出也为 NULL

```
SQL> select sin(20),sinh(20) from dual;
```

32. SQRT(n)

返回数字 n 的平方根，数字 n 必须大于等于 0，输入 NULL，则输出也为 NULL

```
SQL> select sqrt(64),sqrt(10) from dual;
```

33. TAN(n)

返回数字 n（以弧度表示的角）的正切值，输入 NULL，则输出也为 NULL

```
SQL> select tan(20),tan(10) from dual;
```

34. TANH(n)

返回数字 n（以弧度表示的角）的双曲正切值，输入 NULL，则输出也为 NULL

```
SQL> select tanh(20),tan(20) from dual;
```

35. TRUNC

按照指定的精度截取一个数

```
SQL> select trunc(124.1666,-2) trunc1,trunc(124.16666,2) from dual;
```

36. ADD_MONTHS(d, n)

返回特定日期时间之后或之前的几个月所对应的日期时间。

D 用于指定日期时间数据。N 可以是任意整数。当 n 为负整数时，返回特定日期之前几个月对应的日期时间；当 n 为正整数时，返回特定日期之后几个月对应的日期时间。

```
SQL> select to_char(add_months(to_date('199912','yyyymm'),2),'yyyymm') from dual;
```

```
SQL> select to_char(add_months(to_date('199912','yyyymm'),-2),'yyyymm') from dual;
```

37. LAST_DAY(d)

返回特定日期所在月份的最后一天，参数 d 用于指定日期值。

```
SQL> select to_char(sysdate,'yyyy.mm.dd'),to_char((sysdate)+1,'yyyy.mm.dd') from dual;
```

```
SQL> select last_day(sysdate) from dual;
```

38. MONTHS_BETWEEN(d1, d2)

返回日期 d1 和 d2 之间相差的月数，如果 d1 小于 d2，则返回负数。如果日期 d1 和 d2 的天数相同或都是月底，则返回整数；否则 Oracle 以每月 31 天为准，计算结果的小数部分。

```
SQL> select months_between('19-12月-1999', '19-3月-1999') mon_between from dual;
```

```
SQL> select months_between(to_date('2000.05.20', 'yyyy.mm.dd'), to_date('2005.05.20', 'yyyy.mm.dd')) mon_betw from dual;
```

39. NEW_TIME(date, zone1, zone2)

返回特定时区的日期时间在其他时区中的日期时间。参数 date 用于指定日期时间值，参数 zone1 用于指定时区一，参数 zone2 用于指定时区二。

```
SQL> select to_char(sysdate, 'yyyy.mm.dd hh24:mi:ss') bj_time, to_char(new_time  
2 (sysdate, 'PDT', 'GMT'), 'yyyy.mm.dd hh24:mi:ss') los_angles from dual;
```

40. NEXT_DAY(date, char)

返回特定日期之后的第一个工作日所对应的日期。参数 date 用于指定日期时间值，参数 char 用于指定工作日。工作日必须与日期语言匹配。

```
SQL> select next_day('18-5月-2001', '星期五') next_day from dual;
```

41. SYSDATE

用来得到当前系统日期

```
SQL> select to_char(sysdate, 'dd-mm-yyyy day') from dual;
```

42. CHARTOROWID

将字符数据类型转换为 ROWID 类型

```
SQL> select rowid, rowidtochar(rowid), ename from scott.emp;
```

43. CONVERT(c, dset, sset)

将源字符串 sset 从一个语言字符集转换到另一个目的 dset 字符集

```
SQL> select convert('strutz', 'we8hp', 'f7dec') "conversion" from dual;
```

44. HEXTORAW

将一个十六进制构成的字符串转换为二进制

45. RAWTOHEXT

将一个二进制构成的字符串转换为十六进制

46. ROWIDTOCHAR

将 ROWID 数据类型转换为字符类型

47. TO_CHAR(date[, 'format' [, 'nlsparams']]): 将日期值转变为 VARCHAR2 类型的数据。Date 为指定日期值; format 用于指定日期格式, 如果省略, 则使用默认日期显示格式; nlsparams' 用于指定日期显示语言, 指定方式为 'NLS_DATE_LANGUAGE=language', 如果省略, 则使用会话默认日期语言。

```
SQL> select to_char(sysdate, 'yyyy/mm/dd hh24:mi:ss') from dual;
```

48. TO_DATE(string, 'format')

将字符串转化为 ORACLE 中的一个日期

49. TO_MULTI_BYTE

将字符串中的单字节字符转化为多字节字符

```
SQL> select to_multi_byte('高') from dual;
```

50. TO_NUMBER

将给出的字符转换为数字

```
SQL> select to_number('1999') year from dual;
```

51. BFILENAME(dir, file)

指定一个外部二进制文件

```
SQL> insert into file_tbl values(bfilename('lob_dir1', 'image1.gif'));
```

52. CONVERT('x', 'desc', 'source')

将 x 字段或变量的源 source 转换为 desc

53. DUMP(s, fmt, start, length)

DUMP 函数以 fmt 指定的内部数字格式返回一个 VARCHAR2 类型的值

```
SQL> col global_name for a30
```

```
SQL> col dump_string for a50
```

```
SQL> set lin 200
```

```
SQL> select global_name, dump(global_name, 1017, 8, 5) dump_string from global_name;
```

54. EMPTY_BLOB() 和 EMPTY_CLOB()

这两个函数都是用来对大数据类型字段进行初始化操作的函数

55. GREATEST

返回一组表达式中的最大值, 即比较字符的编码大小。

```
SQL> select greatest('AA', 'AB', 'AC') from dual;
```

```
SQL> select greatest('啊', '安', '天') from dual;
```

56. LEAST

返回一组表达式中的最小值

```
SQL> select least('啊','安','天') from dual;
```

57. UID

返回标识当前用户的唯一整数

```
SQL> show user
```

```
SQL> select username,user_id from dba_users where user_id=uid;
```

58. USER

返回当前用户的名字

```
SQL> select user from dual;
```

59. USREVN

返回当前用户环境的信息, opt 可以是:

ENTRYID, SESSIONID, TERMINAL, ISDBA, LABLE, LANGUAGE, CLIENT_INFO, LANG, VSIZE

ISDBA 查看当前用户是否是 DBA 如果是则返回 true

```
SQL> select userenv('isdba') from dual;
```

SESSION

返回会话标志

```
SQL> select userenv('sessionid') from dual;
```

ENTRYID

返回会话人口标志

```
SQL> select userenv('entryid') from dual;
```

INSTANCE

返回当前 INSTANCE 的标志

```
SQL> select userenv('instance') from dual;
```

LANGUAGE

返回当前环境变量

```
SQL> select userenv('language') from dual;
```

LANG

返回当前环境的语言的缩写

```
SQL> select userenv('lang') from dual;
```

TERMINAL

返回用户的终端或机器的标志

```
SQL> select userenv('terminal') from dual;
```

VSIZE(X)

返回 X 的大小(字节)数

```
SQL> select vsize(user),user from dual;
```

60. AVG(DISTINCT|ALL)

all 表示对所有的值求平均值, distinct 只对不同的值求平均值

```
SQL> create table table3(xm varchar(8), sal number(7, 2));
```

```
SQL> insert into table3 values('gao', 1111.11);
```

```
SQL> insert into table3 values('gao', 1111.11);
```

```
SQL> insert into table3 values('zhu', 5555.55);
```

```
SQL> commit;
```

```
SQL> select avg(distinct sal) from table3;
```

```
SQL> select avg(all sal) from gao.table3;
```

61. MAX(DISTINCT|ALL)

求最大值, ALL 表示对所有的值求最大值, DISTINCT 表示对不同的值求最大值, 相同的只取一次

```
SQL> select max(distinct sal) from scott.emp;
```

62. MIN(DISTINCT|ALL)

求最小值, ALL 表示对所有的值求最小值, DISTINCT 表示对不同的值求最小值, 相同的只取一次

```
SQL> select min(all sal) from gao.table3;
```

63. STDDEV(distinct|all)

求标准差, ALL 表示对所有的值求标准差, DISTINCT 表示只对不同的值求标准差

```
SQL> select stddev(sal) from scott.emp;
```

```
SQL> select stddev(distinct sal) from scott.emp;
```

64. VARIANCE(DISTINCT|ALL)

求协方差

```
SQL> select variance(sal) from scott.emp;
```

65. GROUP BY

主要用来对一组数进行统计

```
SQL> select deptno, count(*), sum(sal) from scott.emp group by deptno;
```

66. HAVING

对分组统计再加限制条件

```
SQL> select deptno, count(*), sum(sal) from scott.emp group by deptno having  
count(*)>=5;
```

```
SQL> select deptno, count(*), sum(sal) from scott.emp having count(*)>=5 group by  
deptno ;
```

67. ORDER BY

用于对查询到的结果进行排序输出

```
SQL> select deptno, ename, sal from scott.emp order by deptno, sal desc;
```

68. DECODE

条件判断赋值，很有用处，相当于 IF 判断。可以简化查询、数据转换、提高性能等功效

```
DECODE(expr, adjust1, value1, ....., default value)
```

```
SELECT deptno, ename, sal, decode(deptno, 10, sal*1.2, 20, sal*1.1, sal)
```

```
“New Salary” FROM emp ORDER BY deptno;
```

69. NVL (expr1, expr2)

该函数用于将 NULL 转变为实际值。如果 expr1 是 NULL，则返回 expr2；如果 expr1 不是 NULL，则返回 expr1。参数 expr1 和 expr2 可以是任何类型，但二者数据类型必须要匹配。

```
Select Nvl(write_sect_no, ' 无抄表区段' ) from user_files
```

70. SUM(exprseeion_to_be_added_together)

将传递给它作为参数的表达式中的所有值累加在一起，表达式既可以是列的名字，也可以是一个计算的结果。它仅统计实际的值，NULL 值将被忽略。

```
Select Sum(Total_power) from df_money_files where total_money >0
```

71. to_char(n[, fmt]) 数字转换成字符

其实：to_number, to_char, to_date 等转换函数都可以在很多数据类型之间进行转换，to_lob 一般只能将 long、long raw 转换为 clob、blob、nclob 类型

数字格式：

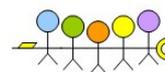
格式元素	功能
9	用于控制显示的有意义的数字的编号
0	用于在结果中标记希望开始显示前面的0的位置。它代替了一个9。最左边的格式字符串是0最通用的位置，但也可以放置在任何地方
\$	在数字前显示\$符号
,	在输出中放置逗号
.	标记小数点的位置
B	强制将0值显示为空白
MI	在格式字符串的结尾使用，使后面的负号显示负数值
S	可以用在格式字符串的开始或末尾，并显示出符号；“+”用于标记正数，“-”用于标记负数；当使用S时，就显示符号
PR	使负数值显示在角括号中。例如，-123.99将被显示为<123.99>。正数值将在前面和后面放置空格代替角括号
D	标记小数点的位置
G	在输出中放置分组分割符（通常是逗号）
C	标记想显示ISO货币指示符的地方。对美元，将使用USD
L	标记想要出现本地货币显示符的位置。对美元，将使用“\$”字符。不能在相同格式中使用L和C
V	用于显示缩放值。在V右边显示的数字表示显示数字之前小数点的右边将有多少位被移动
EEEE	使得SQL*Plus使用科学表示法来显示值。大家必须使用确切的4个E，而且必须出现在格式字符串的最右边
RN	允许大家使用罗马数字显示数字。这是大小写会引起差别的唯一格式元素。大写字母“RN”产生大写的罗马数字，而小写的字母“rn”产生小写的罗马数字。用罗马数字显示数字，必须是整数，而且必须在1~3 999之间
DATE	使得SQL*Plus假定数字代表儒略历日期，而且显示为mm/dd/yy格式

72. round/trunc(d [,fmt])

Round(d[,fmt]):返回日期时间的四舍五入结果。参数d用于指定日期时间值，参数fmt用于指定四舍五入的方式。如果设置fmt为YEAR，则7月1日为分界线；如果设置fmt为MONTH，则16日为分界线；如果设置fmt为DAY，则中午12:00时为分界线。

Trunc(d[,fmt]):用于截断日期时间数据。Fmt用于指定截断日期时间数据的方法。如果设置fmt为YEAR，则结果为本年度的1月1日；如果设置fmt为MONTH，则结果为本月1日。

```
SQL> select round(sysdate,'YEAR') from dual;
```



日期格式:

格式元素	功能
- / , . : :	标点符号可以包括在日期格式字符串的任何地方, 而且将被包括在输出中
'text'	被引用的文本也可以包括在日期格式字符串中, 而且将在输出中重复产生
AD 或 A.D. BC 或 B.C.	在日期显示中包括公元和公元前
AM 或 A.M. PM 或 P.M.	无论在什么时候应用时间都显示 AM 或 PM
CC	世纪数字。该数字用 20 代表 1900 年到 1999 年
SCC	与 CC 相同, 但公元前的日期是负数
D	一周中一天的数字 (1 到 7)
DAY	天的名称 (星期六、星期天和星期一等等)
DD	月的天数
DDD	年的天数
DY	天的缩写名 (Sat、Sun 和 Mon 等等)
HH	天中的小时, 这里是 1~12。
HH12	天中的小时, 这里是 1~12, 与 HH 相同
HH24	24 小时时钟下天的小时数 (0~23)
IW	一年中的周 (1~53)
IYYY	4 位数字年份
IYY	年的最后 3 位数
IY	年的最后两位数
I	年的最后一位数字
J	儒略历天。第 1 天等于公元前 4712 年 1 月 1 日
MI	分钟
MM	月数
MON	3 个字母的月份缩写

MONTH	月名，完全拼写出来
Q	年的季度。1 季度是 1~3 月，2 季度是 4~6 月等等
RM	罗马数字的月份数
RR	在使用 TO_CHAR 时，返回年的最后 2 位数字
RRRR	在使用 TO_CHAR 时，返回年的 4 位数字
SS	秒
SSSS	从午夜开始的秒数
WW	一年中的周
W	月中的周。第一周从月的第一天开始算起，第 2 周从月的第 8 天开始算起等等
Y.YYY	4 位数的年份，在第 1 位数字后有逗号
YEAR	用单词的形式拼写出的年
SYEAR	用单词的形式拼写出的年，在表示公元前的年份时，前面有一个负号
YYYY	4 位数字年份
SYYYY	4 位数字年份，当表示公元前的年份时，前面有一个负号
YYY	年数的最后 3 位数字
YY	年数的最后 2 位数字
Y	年数的最后 1 位数字

73. SYS_CONTEXT(context, attribute)

返回应用上下文的特定属性值，其中参数 context 用于指定应用上下文名，参数 attribute 用于指定属性名。（源码网整理：www.codepub.com）

Namespace 命名空间一般由 create context 创建的，也可制定默认的 USERENV

```
Select SYS_CONTEXT('USERENV','TERMINAL') terminal From dual;
```

```
Select
```

```
SYS_CONTEXT('userenv','session_user'),sys_context('userenv','os_u  
ser') from dual;
```

74. SYS_GUID()

用于生成类型为 RAW 的 16 或 32 字节的唯一标识符，一般为主机 ID、进程 ID 和序列号的组合值，每次调用该函数都会生成不同的 RAW 数据。

第 8 章 操纵数据

使用 DML 语句（INSERT，UPDATE，DELETE）可以操纵表和视图的数据。

8.1 插入数据

当给表增加数据时，可以使用 INSERT 语句。使用 INSERT 语句既可以为表插入单行数据，也可以通过子查询将一张表的多行数据插入到另一张表。从 Oracle Database 9i 开始，Oracle 还提供了多表插入功能，即使用一条 INSERT 语句同时为多张表插入数据。但使用 INSERT 语句有以下一些注意事项：

- 如果为数字列插入数据，则可以直接提供数字值；如果为字符列或日期列插入数据，则必须用单引号引住
- 当插入数据时，数据必须要满足约束规则，并且必须为主键列和 NOT NULL 列提供数据
- 当插入数据时，数据必须要与列的个数和顺序保持一致

1. 使用 VALUES 子句插入数据

```
INSERT INTO <table> [ (column[, column, ...]) ] VALUES (value[, value, ...])
```

- ✓ 不使用列列表插入数据

```
INSERT INTO dept VALUES(50, ' TRAIN' , ' BOSTON' )
```

- ✓ 使用列列表插入单行数据

```
INSERT INTO emp(empno,ename, job,hiredate)
VALUES (1234, ' JOHN' , ' CLERK' , ' 01-3 月-86' );
```

- ✓ 使用特定格式插入日期值

```
INSERT INTO emp(empno,ename, job hiredate)
VALUES (1356, ' MARY' , ' CLERK' ,
to_date( '1983-10-20' , ' YYYY-MM-DD' ));
```

- ✓ 使用 DEFAULT 提供数据

```
INSERT INTO dept VALUES(60, ' MARKET' , DEFAULT);
```

- ✓ 使用替代变量插入数据（建立如下的 sql 脚本 a.sql）

```
ACCEPT no PROMPT '请输入雇员号： '
ACCEPT name PROMPT '请输入雇员名： '
```

```
ACCEPT title PROMPT '请输入雇员岗位: '  
ACCEPT d_no PROMPT '请输入部门号: '  
INSERT INTO emp(empno, ename, job, hiredate, deptno)  
VALUES(&no, ' &name' , ' &title' , SYSDATE, &d_no)  
运行该脚本
```

2. 使用子查询插入数据

```
INSERT INTO <table> [(column[, column, ...])] subQuery
```

✓ 使用子查询插入数据

```
INSERT INTO employee(empno, ename, sal, deptno)  
SELECT empno, ename, sal, deptno FROM emp  
WHERE deptno=20;
```

✓ 使用子查询执行直接装载

```
INSERT /*+APPEND*/ INTO employee (empno, ename, sal, deptno)  
SELECT empno, ename, sal, deptno FROM emp  
WHERE deptno=20;
```

说明：尽管以上两种语句执行结果一样，但第二条语句使用/*+APPEND*/表示采用直接装载方式，当装载大批量数据时，采用第二种方法装载数据的速度要远远优于第一种方法。

3. 使用多表插入

```
INSERT ALL insert_into_clause[value_clause] subquery;  
INSERT FIRST insert_into_clause[value_clause] subquery;
```

其中 insert_into_clause 用于指定 INSERT 子句，value_clause 用于指定值子句，subquery 用于指定提供数据的子查询

✓ 使用 ALL 操作符执行多表插入

```
INSERT ALL  
WHEN deptno=10 THEN INTO dept10  
WHEN deptno=20 THEN INTO dept20  
WHEN deptno=30 THEN INTO dept30  
WHEN job=' CLERK' THEN INTO clerk  
ELSE INTO other
```

```
SELECT * FROM emp;
```

当执行以上插入语句之后，会将部门 10 的雇员信息插入到 DEPT10 表，将部门 20 的雇员信息插入到 DEPT20 表，将部门 30 的雇员信息插入到 DEPT30 表，将岗位 CLERK 的所有雇员插入到 CLERK 表，将其他行插入到 OTHER 表

- ✓ 使用 FIRST 操作符执行多表插入

```
INSERT FIRST  
  
WHEN deptno=10 THEN INTO dept10  
  
WHEN deptno=20 THEN INTO dept20  
  
WHEN deptno=30 THEN INTO dept30  
  
WHEN job=' CLERK' THEN INTO clerk  
  
ELSE INTO other  
  
SELECT * FROM emp;
```

当使用 FIRST 操作符执行多表插入时，如果数据已经满足了先前条件，并且已经被插入到某表，那么该行数据在后续插入中将不会被再次使用。示例如下：

```
create table test3(a number(2), b number(2));  
create table test3(a number(2), b number(2));  
create table test3(a number(2), b number(2));  
insert into test1 values(10,20);  
commit  
insert first  
  when a=10 then into test2  
  when b=20 then into test3 values(a,b)  
select * from test1;
```

8.2 更新数据

当更新表行的数据时，可以使用 UPDATE 语句。当使用 UPDATE 语句时，既可以使用表达式更新列值，也可以使用子查询更新一列或多列的数据，但使用 UPDATE 语句有以下注意事项：

- 如果要更新数字列，则可以直接提供数据值；如果要更新字符列或日期列，则数据必须用单引号引住。
- 当更新数据时，数据必须要满足约束规则
- 当更新数据时，数据必须与列的数据类型匹配

1. 使用表达式更新数据

语法: UPDATE <table|view>

```
SET <column>=<value>| [,<column>=<value>]
```

```
[WHERE <condition>];
```

- ✓ 更新单列数据

```
UPDATE emp SET sal=2460 WHERE ename=' SCOTT' ;
```

- ✓ 更新多列数据

```
UPDATE emp SET sal=sal*1.1,comm.=sal*0.1
```

```
WHERE deptno=20;
```

- ✓ 更新日期列数据

```
UPDATE emp SET hiredate=TO_DATE( '1984/01/01' , ' YYYY/MM/DD' )
```

```
WHERE empno=7788;
```

- ✓ 使用 DEFAULT 选项更新数据

```
UPDATE emp SET job=DEFAULT WHERE ename=' SCOTT' ;
```

- ✓ 更新违反约束规则的数据

```
UPDATE emp SET deptno=55 WHERE empno=7788;ERROR,未找到父项关键字
```

2. 使用子查询更新数据

当使用 UPDATE 语句更新数据时, 某些情况下, 使用子查询执行效率更好。

- ✓ 更新关联数据

当更新关联数据时, 使用子查询可以降低网络开销 (简化 SQL 语句个数)。

```
UPDATE emp SET (job, sal, comm)=
```

```
(SELECT job, sal, comm. FROM emp WHERE ename=' SMITH' )
```

```
WHERE ename=' SCOTT' ;
```

- ✓ 复制表数据

当使用触发器复制表数据时, 如果表 A 的数据被修改, 那么表 B 的数据也应该修改。

通过使用子查询, 可以基于一张表修改另一张表的数据。

```
UPDATE employee SET deptno=
```

```
(SELECT deptno FROM emp WHERE empno=7788);
```

```
WHERE job=(SELECT job FROM emp WHERE empno=7788);
```

8.3 删除数据

使用 DELETE 语句可删除一行或多行数据，语法如下：

```
DELETE FROM <table|view> [WHERE <condition>];
```

- ✓ 删除满足条件的数据

```
DELETE FROM emp WHERE ename=' SMITH' ;
```

- ✓ 删除表的所有数据

```
DELETE FROM emp;
```

- ✓ 使用 TRUNCATE TABLE 截断表

TRUNCATE TABLE emp, 它不仅可删除表的所以数据，还会释放表段所占用的空间。

- ✓ 使用子查询删除数据

```
DELETE FROM emp WHERE deptno=(SELECT deptno FROM dept WHERE dname=' SALES' );
```

- ✓ 删除主表数据的注意事项

当删除主表数据时，必须确保从表不存在相关记录，否则会显示错误信息。

日期类型数据处理：

客户端是中文环境，月份不能用英文的月份写法，必须用中文的“六月”

如果不想修改 sql 语句运行的话，就需要在执行该语句之前，使用 alter session 命令将

nls_date_language 修改为 american，如下：

```
alter session set nls_date_language='american' --以英语显示日期或
```

```
alter session set nls_date_language='simplified chinese'
```

```
alter session set nls_date_format='yyyy-mm-dd';
```

第 9 章 使用事务

本章将学习的内容：

- 创建用户定义的事务
- 提交事务
- 回滚事务
- 使用保存点有选择地丢弃变动
- 了解保护数据的上锁模式

9.1 概述

在 RDBMS 中，数据的变化必须同时在所有相关的表中得到反映。如果一个表中的数据改变了，而这一变化没有在相关的表中反映出来，那么这将导致数据库里数据的不一致。如银行转帐。因此，数据的改变必须要么在相关的表中同时得到反映，要么不在任何表中得到反映。Oracle 使用事务确保数据的一致性。

事务可定义为把一串一起执行的操作作为单个逻辑工作单元处理。单个工作单元必须具有四个属性，即：原子性、一致性、隔离性和持久性。

- 原子性：指所有的数据修改，要么全部执行，要么一个也不执行。
- 一致性：指通过事务进行的所有数据修改的状态必须同时在所有相关的表中得到反映。有关关系型数据库中的数据完整性的所有规则，必须应用到事务修改中以维护数据完整性。
- 隔离性：是指事务应在另一个事务的修改过程开始前对数据进行访问，或者等另一个事务修改完数据后再访问。处于修改之中的数据是不可被访问的。因此，事务不能在另一个事务修改数据时访问数据的中间状态
- 持久性：持久性保证事务对数据所做的变动是持久的。即使系统发生故障，数据中变动也不会丢失。

使用事务修改数据库的优点：

- 它保证了数据的一致性。
- 使用事务时，数据修改更灵活而且修改过程是可控的。
- 即使在用户处理失败或者系统发生故障的情况下，数据仍然是安全的。
- 事务保证 DML（数据操纵语言）语句对数据所做的变动是一致的。

9.2 事务分类

- 显式事务

显式事务指必须对所做的数据修改明确地表示接受或丢弃。显式事务又称为用户定义事务。为了接受显式事务对数据所做的修改，我们需要用到提交（COMMIT）语句。

提交语句：

完成显式事务。它结束当前事务，并使所有的修改持久有效。COMMIT 语句允许用户查看其他用户对数据所做的修改。

在 Oracle 里，COMMIT 语句写在事务的最后。COMMIT 语句执行前的数据操作不是持久有效的，因为保存的数据库在缓冲区里。即使当前用户能看到数据的变动，但其他用户看不到这种变化。Oracle 服务器默认在行级上锁，这意味着当一个用户正在修改数据时，其他用户不可能操纵被锁定行中数据。只执行 COMMIT 语句后数据才是永久地存储在数据库中。数据一旦被提交，以前的数据就彻底消失了。

COMMIT 语句使你能：

- 保证数据的一致性
- 可在永久地更新数据前预览修改
- 将逻辑相关的所有操作组合起来

语法:

```
SQL statement1;  
SQL statement2;  
COMMIT;
```

其中, COMMIT 保证 statement1 和 statement2 所做的修改持久有效。

- 隐式事务

隐式事务是自动地提交数据变动的系统事务。使用 AUTOCOMMIT 语句可实现隐式事务。AUTOCOMMIT 语句可设置为 ON 或 OFF。只有当 AUTOCOMMIT 被设置为 ON 时, 数据修改才是持久有效的。如果 AUTOCOMMIT 被设置为 OFF, 必须显式地提交对数据所做的修改。按缺省方式, AUTOCOMMIT 被设置为 OFF。验证 AUTOCOMMIT 状态的语法如下:

```
SHOW AUTOCOMMIT
```

当 AUTOCOMMIT 被禁用时, 结果返回 “autocommit OFF”; 反之, 结果返回 “autocommit IMMEDIATE”。

设想这样一种情况: 你执行了两条 DDL 语句, 而 AUTOCOMMIT 的状态为 OFF。这时, 你必须显式地提交该语句对数据库所做的修改。

语法:

```
SET AUTOCOMMIT ON
```

该语句将启用 AUTOCOMMIT。这可通过说明 SHOW AUTOCOMMIT 语句来验证。

也可以执行一定个数的语句之后, 设置 AUTOCOMMIT 以提交数据库中的变动。为此, 必须指明提交数据库中的变动的语句个数。语法如下:

```
SET AUTOCOMMIT value
```

其中, value: 确定提交修改前执行的语句个数。

如:

```
UPDATE Employee Set cCurrentPosition='0001'  
WHERE cEmployeeCode='000006';  
Update Position Set nCurrentStrength=nCurrentStrength+1  
WHERE cPositionCode='0001';  
UPDATE Positon Set nCurrentStrength=cCurrentStrength-1  
WHERE cPositionCode='0016';
```

在 AUTOCOMMIT 为 OFF 情况下, 当出现以下情况时会自动提交事务:

- 当使用 DDL 语句时会自动提交事务, 例如 CREATE TABLE、ALTER TABLE、DROP TABLE 等语句
- 当执行 DCL 语句 (GRANT、REVOKE) 时会自动提交事务
- 当正常退出 SQL*Plus 时 (quit 或 exit) 会自动提交事务。

9.3 回复修改

回滚事务

有时, 可能在修改了表中数据后发现: 所做的修改是无紧要的或不正确的。也可能要按照你的要求进行修改。在这种情况下, 需要回复所做的修改。回复修改时, 可使用 ROLLBACK 语句。ROLLBACK 语句终止当前事务, 使数据库返回到以前的状态。

当在表中作出变动时, 如果没有指定 COMMIT 语句, 那么此变动只是存储在数据

库缓冲区里。当说明 **ROLLBACK** 语句时，将丢弃缓冲区里的数据。然而，如果说明了 **COMMIT** 语句，那么数据将在各自的表里更新。说明了 **COMMIT** 语句后，**ROLLBACK** 语句不会丢弃修改了。

使用 **ROLLBACK** 语句时：

- 自上一个 **COMMIT** 语句执行以来的所有数据修改都被取消。
- 进行数据修改的行上的锁被解开。

使用 **ROLLBACK** 语句的优点有：

- 对数据所做的修改是一致的，因为使用 **ROLLBACK** 语句，缓冲区里的数据将被除去。
- 可恢复数据到以前的状态
- 如果不小心删错了需要的行，可检索被删除的行。
- 万一执行 **SQL** 查询时发生异常，原始数据仍可得到恢复。

语法：

SQL statement1;

SQL statement2;

ROLLBACK;

其中，**ROLLBACK**：丢弃自上一个 **COMMIT** 语句执行以来所做的数据修改。

自动回滚

在下列情况下，数据修改自动被回滚：

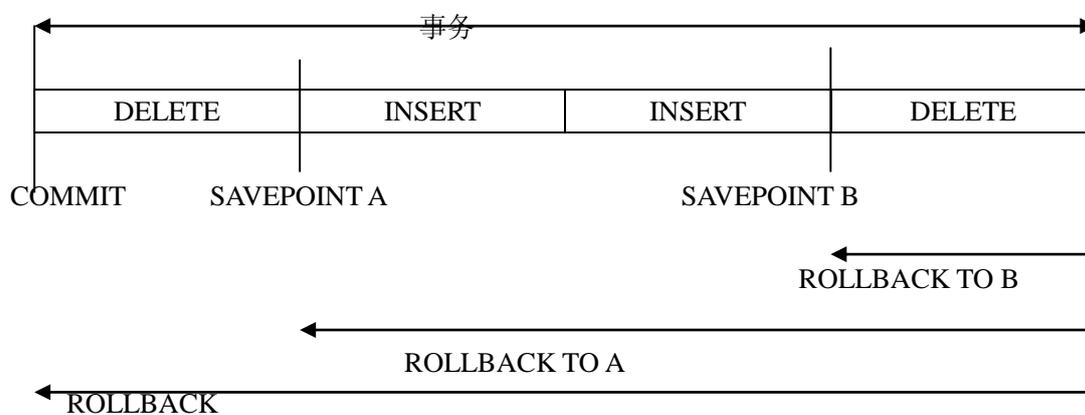
- 系统崩溃或发生故障
- **SQL*Plus** 意外终止。可使用窗口中 **CLOSE** 按钮终止 Oracle，这认为是非正常终止。退出 Oracle 的常规方法是在 **SQL** 揭示符处输入 **EXIT** 命令。

语句级回滚：

一个事务

9.4 回复部分事务

当执行 **ROLLBACK** 命令时，通过指定保存点可以取消部分事务，如下图所示。



1. 设置保存点

设置保存点是使用 SQL 命令 SAVEPOINT 来完成的。另外，在编写应用程序时，开发人员可以使用包 DBMS_TRANSACTION 的过程 SAVEPOINT 设置保存点。示例如下：

```
SAVEPOINT A; 或
```

```
Exec dbms_transaction.savepoint('a');
```

2. 取消部分事务

为了取消部分事务，可以回退到保存点。回退到保存点既可以作用 ROLLBACK 命令，也可以使用包 DBMS_TRANSACTION 的过程 ROLLBACK_SAVEPOINT。示例如下：

```
Rollback to a; 或
```

```
Exec dbms_transaction.rollback_savepoint('a');
```

9.5 事务和锁

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。假定会话 A 更新 EMP 表行的数据，那么会在表 EMP 上加表锁；此时如果其他会话修改表结构，则会显示错误信息。

如：

会话 A：

```
UPDATE emp SET sal=sal*1.1 WHERE ename='SCOTT';
```

已更新 1 行。

会话 B：

```
ALTER TABLE emp ADD remark VARCHAR2(100);
```

ERROR 位于第 1 行：

```
ORA-00054: 资源正忙，要求指定 NOWAIT
```

在 Oracle 数据库中，为了确保数据为数据的读一致性，不允许其他用户读取脏数据（未提交事务）。假定会话 A 将雇员 SCOTT 工资修改为 2000（未提交），那么其他会话只能查询到原来工资；只有在会话 A 提交了事务之后，其他会话才能查询到新工资。

第 10 章 使用约束

本章主要内容

- 约束简介
- 定义约束
- 维护约束
- 显示维护信息

为了使得数据库数据能够满足商业逻辑或者企业规则，在 Oracle 数据库中可以使用约束、触发器和应用代码（过程、函数）三种方法。在这三种方法中，约束易于维护，并且具有最好的性能，因此实现数据完整性首选约束。

10.1 约束简介

约束用于确保数据库数据满足特定的商业逻辑或者企业规则。如果定义了约束，并且数据不符合约束规则，那么 DML 操作（INSERT、UPDATE、DELETE）将不能成功执行。约束包括 NOT NULL、UNIQUE、PRIMARY KEY、FOREIGN KEY 以及 CHECK 等五种类型。

- NOT NULL：用于确保列不能为 NULL。
- UNIQUE（惟一约束）：用于惟一地标识列的数据（允许列为 NULL）。当定义惟一约束时，默认情况下 Oracle 会自动基于惟一约束列建立惟一索引，并且索引名与约束名完全一致。
- PRIMARY KEY（主键约束）：用于惟一地标识表行的数据（主键约束列值不能重复，也不能为 NULL）。当定义主键约束时，Oracle 会自动基于主键约束列建立惟一索引，并且索引名与约束名完全一致。注：一张表最多只能有一个主键约束
- FOREIGN KEY（外键约束）：用于定义主从表之间的关系。外部键约束要定义在从表上，但主表必须具有主键约束或惟一约束。
- CHECK（检查约束）：用于强制表行数据必须要满足的条件。如在 sal 列上定义了 CHECK 约束，并且要求 sal 列值必须在 1000-5000 之间

10.2 定义约束

当执行 CREATE TABLE 语句建表时，必须要提供表和列的信息，另外也可以在建表的同时定义约束，语法如下：

```
CREATE TABLE [schema.]table_name(
    column_name datatype [DEFAULT expr] [column_constraint],
    ...
    [table_constraint][,...]
```

DEFAULT 子句用于指定列的默认值，column_constraint 用于在列级定义约束，table_constraint 用于在表级定义约束。列级定义是指在定义列的同时定义约束，所有约束都可以在列级定义；表级定义是指在定义了所有列之后定义的约束。语法如下：

列级约束：column [CONSTRAINT constraint_name] constraint_type

表级约束：[CONSTRAINT constraint_name] constraint_type(column,...)

);

- 1) 定义 NOT NULL 约束：只能在列级定义，不能在表级定义。

```
CREATE TABLE emp01(
    Eno INT NOT NULL,          没指定约束名
    Name VARCHAR2(10) CONSTRAINT nn_name NOT NULL,
    Salary NUMBER(6,2)
);
```

测试用例：INSERT INTO emp01 VALUES(1,NULL,1000);

- 2) 定义 UNIQUE 约束：既可在列级定义，也可在表级定义。

```
CREATE TABLE emp02(
    eno INT,name VARCHAR2(10),salary NUMBER(6,2),
    CONSTRAINT u_name UNIQUE(name)
);
```

测试用例：

```
INSERT INTO emp02 VALUES(1,'SCOTT',1000);
```

```
INSERT INTO emp02 VALUES(2,'SCOTT',1000);
```

- 3) 定义 **PRIMARY KEY** 约束: 既可在列级定义, 也可在表级定义。一张表最多只能具有一个主键约束。

```
CREATE TABLE dept04(  
    dno INT PRIMARY KEY,  
    dname VARCHAR2(10),loc VARCHAR2(20)  
);
```

测试用例:

```
INSERT INTO dept04 VALUES(1,'SALES','DALLAS');
```

```
INSERT INTO dept04 VALUES(1,'ADMIN','DALLAS');
```

- 4) 定义 **FOREIGN KEY** 约束: 既可在列级定义, 也可在表级定义。

外键约束用于定义主从表之间的一对多关系。当定义了外键约束之后, 要求外部键列的数据必须在主表的主键列(或唯一列)中存在, 或者为 NULL。定义外键约束需要使用以下关键字:

FOREIGN KEY: 该选项用于指定在表级定义外部键约束。当在表级定义外部键约束时必须指定该选项, 在列级定义外部键约束时不需要指定该选项。

REFERENCES: 用于指定主表名及其主键列, 必须指定

ON DELETE CASCADE: 用于指定级联删除选项。若在定义外部键约束时指定了该选项, 则当删除主表数据时会级联删除从表的相关数据。

ON DELETE SET NULL: 用于指定转换相关的外部键值为 NULL。若在定义外部键约束时指定了该选项, 则当删除主表数据时会将从表外部键列的数据设置为 NULL。

```
CREATE TABLE emp04(  
    eno INT,name VARCHAR2(10),salary NUMBER(6,2),  
    dno INT CONSTRAINT fk_dno REFERENCES dept04(dno)  
);
```

测试用例:

```
INSERT INTO emp04 VALUES(1111,'SCOTT',1000,2);
```

上例中如果在表 dept04 中不存在 2, 即如果在 dno 列上提供了不存在的部门号, 则会显示错误信息。

- 5) 定义 **CHECK** 约束: 既可在列级定义, 也可在表级定义。**CHECK** 约束允许列为 NULL。

```
CREATE TABLE emp05(  
    Eno INT,name VARCHAR2(10),salary NUMBER(6,2),  
    CHECK(salary BETWEEN 1000 AND 5000)  
);
```

测试用例:

```
INSERT INTO emp05 VALUES(1111,'SCOTT',800);
```

- 6) 定义复合约束: 基于多列定义复合约束。复合约束只能在表级定义。

```
CREATE TABLE item(  
    order_id NUMBER(3),item_id NUMBER(3), product VARCHAR2(20),  
    PRIMARY KEY(order_id,item_id)  
);
```

10.3 维护约束

1) 增加约束

若在建表时没有定义必须的约束,可在建表之后使用 ALTER TABLE 增加约束。需注意的是:如果增加 UNIQUE、PRIMARY KEY、FOREIGN KEY 和 CHECK 约束,必须使用 ALTER TABLE 语句的 ADD 子句;如果增加 NOT NULL 约束,必须使用 ALTER TABLE 语句的 MODIFY 子句。语法如下:

```
ALTER TABLE table ADD [CONSTRAINT constraint_name]
    constraint_type(column,...);
```

```
ALTER TABLE table MODIFY column
    [CONSTRAINT constraint_name] NOT NULL
```

● 增加 NOT NULL 约束

```
ALTER TABLE emp02 MODIFY name NOT NULL
```

● 增加 UNIQUE 约束

```
ALTER TABLE emp04 ADD CONSTRAINT u_emp04 UNIQUE(name)
```

● 增加 PRIMARY KEY 约束

```
ALTER TABLE emp01 ADD PRIMARY KEY(dno);
```

● 增加 FOREIGN KEY 约束

```
ALTER TABLE emp01 ADD FOREIGN KEY(dno)
REFERENCES dept01(dno)
```

● 增加 CHECK 约束

● ALTER TABLE emp01 ADD CHECK(sal BETWEEN 800 AND 5000);

2) 修改约束名

```
ALTER TABLE table RENAME CONSTRAINT old_constraint_name
    TO new_constraint_name;
```

3) 删除约束: 约束不再需要时,可删除。语法:

```
ALTER TABLE table DROP
    CONSTRAINT constraint_name | PRIMARY KEY[CASCADE];
CASCADE 用于指定级联删除从表的外部键约束。
```

```
ALTER TABLE emp01 DROP CONSTRAINT ch_emp01_salary;
```

注意:当删除特定表的主键约束时,如果该表具有相关的从表,那么在删除主键约束时必须带有 CASCADE 选项,否则会出现错误信息。

如: ALTER TABLE dept01 DROP PRIMARY KEY CASCADE;

4) 禁止约束: 是指使约束临时失效。语法如下:

```
ALTER TABLE table
    DISABLE CONSTRAINT constraint_name[CASCADE];
CASCADE 用于指定级联禁止从表的外部键约束。
```

```
ALTER TABLE emp05 DISABLE CONSTRAINT SYS_C005022;
```

测试用例

```
INSERT INTO emp05 VALUES(1,'SCOTT',800); 可插入
```

5) 激活约束: 是指使约束重新生效。语法:

```
ALTER TABLE table ENABLE CONSTRAINT constraint_name;
```

10.5 显示约束信息

1) USER_CONSTRAINTS: 查询该视图可显示当前用户表的所有约束信息。

```
SELECT constraint_name,constraint_type FROM user_constraints
```

- ```
WHERE table_name='EMP';
```
- 2) USER\_CONS\_COLUMNS: 查询该视图, 可以显示当前用户约束所对应的表列
- ```
SELECT column_name FROM user_cons_columns
WHERE constraint_name='PK_EMP';
```

第 11 章 使用视图

视图是一个或多个表的逻辑表示, 它对应于一条 SELECT 语句, 并且其查询结果会被作为表对待, 因此视图也被称为虚表, 而其 SELECT 语句所对应的表则被称为视图基表。

如:

```
CREATE VIEW vu_emp AS
SELECT empno,ename,sal,job,deptno FROM emp;
```

11.1 视图简介

视图是基于其他表或者其他视图的逻辑表, 它本身没有任何数据, 视图上的 SELECT、UPDATE 和 DELETE 等操作实际都是针对视图基表来完成的。

1. 视图的作用

- ✧ 限制数据访问。因为视图定义对应于 SELECT 语句, 所以当访问视图时只能访问 SELECT 语句所涉及到的列
- ✧ 简化复杂查询。如果经常需要在多个表之间执行复杂查询操作, 那么可以基于该复杂查询语句建立视图。这样, 当查询该视图时, Oracle 内部会执行视图所对应的复杂查询语句。

2. 视图分类

- ✧ 简单视图: 它是指基于单个表所建立的, 不包含任何函数、表达式以及分组数据的视图。
- ✧ 复杂视图: 它是指包含函数、表达式或者分组数据的视图, 使用复杂视图的主要目的是为了简化查询操作。
- ✧ 连接视图: 它是指基于多个表所建立的视图, 使用连接视图的主要目的是为了简化连接查询。
- ✧ 只读视图: 它是指只允许执行 SELECT 操作, 而禁止任何 DML 操作的视图。

3. 在视图上执行 DML 操作的原则

当建立了表之后, 用户可以在表上执行 SELECT、INSERT、UPDATE 和 DELETE 操作, 在建立了视图之后用户同样也可以执行这些操作。对于表来说, 只要数据符合约束规则, 那么用户就可以执行相应的 DML 操作; 而对于视图来说, 当执行 DML 操作时, 不仅要求数据符合约束规则, 而且还必须要满足一些其他原则如下:

- ✧ DELETE 操作原则: 如果视图包含有 GROUP BY 子句、分组函数、DISTINCT 关键字和 ROWNUM 伪列, 那么不能在该视图上执行 DELETE 操作。
- ✧ UPDATE 操作原则: 如果视图包含有 GROUP BY 子句、分组函数、DISTINCT 关键字和 ROWNUM 伪列以及使用表达式所定义的列, 那么不能在该视图上

执行 UPDATE 操作。

- ✧ INSERT 操作原则：如果视图包含有 GROUP BY 子句、分组函数、DISTINCT 关键字、ROWNUM 伪列以及使用表达式所定义的列，或者在视图上没有包含视图基表的 NOT NULL 列，那么不能在该视图上执行 INSERT 操作。

11.2 建立视图

建立视图是使用 CREATE VIEW 命令来完成的。为了在当前方案中建立视图，要求数据库用户必须具有 CREATE VIEW 系统权限；为了在其他方案中建立视图，要求数据库用户必须具有 CREATE ANY VIEW 系统权限，建立视图的语法如下：

```
CREATE VIEW view [(alias[,alias]...)]
```

```
AS subquery
```

```
[WITH CHECK OPTION [CONSTRAINT constraint]]
```

```
[WITH READ ONLY]
```

其中 view 指视图名，alias 用于指定视图列的别名，subquery 用于指定视图所对应的子查询语句，WITH CHECK OPTION 子句用于在视图上定义 CHECK 约束；WITH READ ONLY 子句用于定义只读视图。需注意的是，当建立视图时，如果不提供视图列别名，Oracle 会自动使用子查询的列名或者列别名；如果视图子查询包含有函数或表达式，那么必须要为其定义列别名。

- ✧ 建立简单视图：在简单视图上可执行 SELECT，INSERT，UPDATE 及 DELETE 操作，如

```
CREATE VIEW emp_vu AS
```

```
SELECT empno,ename,sal,job,deptno FROM emp;
```

测试用例：

```
INSERT INTO emp_vu VALUES(1234,'MARY',1000,'CLERK',30);
```

```
UPDATE emp_vu SET sal=2000 WHERE empno=1234;
```

```
DELETE FROM emp_vu WHERE empno=1234;
```

```
SELECT * FROM emp_vu WHERE empno=7788;
```

- ✧ 建立复杂视图

```
CREATE VIEW job_vu AS
```

```
SELECT job,avg(sal) avgsal,sum(sal) sumsal,max(sal) maxsal,min(sal) minsal FROM emp GROUP BY job;
```

测试用例：

```
SELECT * FROM job_vu WHERE job='CLERK';
```

- ✧ 建立连接视图

```
CREATE VIEW dept_emp_vu20 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=20;
```

测试用例：

```
SELECT * FROM dept_emp_vu20;
```

- ✧ 建立只读视图

```
CREATE VIEW emp_vu20 AS
```

```
SELECT * FROM emp WHERE deptno=20 WITH READ ONLY;
```

测试用例：

```
SELECT ename,sal,deptno FROM emp_vu20;
```

UPDATE emp_vu20 SET sal=1000 WHERE ename='FORD'; 错误!

- ✧ 在建立视图时定义 CHECK 约束。当在视图上定义了 CHECK 约束之后,如果在视图上执行 INSERT 和 UPDATE 操作,那么要求新数据必须是视图子查询的查询结果。如:

```
CREATE VIEW emp_vu10 AS
SELECT * FROM emp WHERE DEPTNO=10
WITH CHECK OPTION CONSTRAINT chk_vu10;
```

测试用例:

```
INSERT INTO emp_vu10 (empno,ename,sal,job,deptno)
VALUES(1235,'PETER',1000,'CLERK',20); 错误,原因:当基于视图 emp_vu10
执行 INSERT 操作时,DEPTNO 列的值必须设置为 10;当基于该视图执行 UPDATE
操作时,只能修改除 DEPTNO 列之外的其他列。
```

- ✧ 在建立视图时定义列别名

```
CREATE VIEW emp_view(name,salary,title)
AS SELECT ename,sal,job FROM emp;
```

测试用例:

```
SELECT name FROM emp_view WHERE title='CLERK';
```

11.3 维护视图

- ✧ 修改视图定义

建立了视图之后,如果要改变视图所对应的子查询语句,那么可以使用 CREATE OR REPLACE VIEW 语句修改视图定义。语法如下:

```
CREATE OR REPLACE VIEW view [(alias[,alias]...)]
AS subquery;
```

例:

```
CREATE OR REPLACE VIEW emp_vu(name,salary,title) AS
SELECT ename,sal,job FROM emp;
```

- ✧ 重新编译视图

当改变了视图基表的定义(例如增加列、删除列等)之后,视图会被标记为无效状态。当用户访问视图时,Oracle 会自动重新编译视图。另外,也可执行 ALTER VIEW 语句手工编译视图,语法如下:

```
ALTER VIEW view COMPILE;例:
ALTER VIEW emp_vu COMPILE;
```

- ✧ 删除视图

当视图不再需要时,可删除视图,语法如下:

```
DROP VIEW view;如:
DROP VIEW emp_vu;
```

11.4 显示视图信息

- ✧ USER_VIEWS

通过查询该视图,可显示当前用户所有视图的详细信息。示例如下:

```
SELECT text FROM user_views WHERE view_name='EMP_VU';
```

- ✧ USER_UPDATABLE_COLUMNS

通过查询该视图,可以显示是否允许在特定视图列上执行 DML 操作。示例如下:

```
SELECT column_name,insertable,updatable,deletable
FROM user_updatable_columns
WHERE table_name='DEPT_EMP_VU20' AND
column_name IN('DNAME','ENAME');
```

第 12 章 使用其它对象（索引序列同义词）

本章主要内容：

- ◇ 索引
- ◇ 序列
- ◇ 同义词

12.1 使用索引

索引用于加快数据定位速度。通过使用索引，可以大大降低 I/O 次数，从而提高 SQL 语句的访问性能。

● 单列索引和复合索引

按索引列个数，可将索引划分为单列索引和复合索引两类。单列索引是指基于单个列建立的索引，复合索引是指基于两列或多列建立的索引。同一张表中可建立多个索引，但要求列的组合必须不同，使用以下语句建立的两个索引是合法的

```
CREATE INDEX emp_idx1 ON emp(ename, job);
CREATE INDEX emp_idx1 ON emp(job, ename);
```

如果索引列顺序完全相同，则是不合法的。

● 惟一索引和非惟一索引

按索引列值的惟一性，可将索引划分为惟一索引和非惟一索引。惟一索引是指索引列值不能重复的索引，非惟一索引是指索引列值可以重复的索引，当定义主键约束或惟一约束时，Oracle 会自动在相应的约束列上建立惟一索引。

● 使用索引的指导方针

➤ 索引正确的表和列。

当建立和规划索引时，必须选择合适的表和列。如果选择了不合适的表和列，那么不仅无法提高查询速度，反而会极大地降低 DML 操作的速度。建立索引的指导方针如下：

- ◇ 索引应该建立在 WHERE 子句经常引用的表列上。如果在大表上频繁使用某列或某几列作为条件执行检索操作，并且检索行数低于总行数的 15%，那么应该考虑在这些列上建立索引。
- ◇ 为了提高多表连接的性能，应该在连接列上建立索引
- ◇ 如果经常需要基于某列或某几列执行排序操作，那么通过在这些列上建立索引，可以加快数据排序的速度。
- ◇ 不要在小表上建立索引

➤ 限制表的索引个数

索引主要用于加速查询速度，但会降低 DML 操作的速度。索引越多，DML 操作的速度将会越慢，尤其会极大地影响 INSERT 操作和 DELETE 操作的速度。因此，在规划索引时，必须要仔细权衡查询和 DML 的需求。

- 删除不需要的索引
因为索引会降低 DML 速度, 所以应该删除不合理或不需要的索引。以下情况, 该考虑删除索引:
 - ✧ 删除在小表上建立的索引。如果表很小, 使用索引不会加快查询速度。
 - ✧ 删除查询语句不会引用的索引。

12.1.1 建立索引

一般情况下建立索引是由表的所有者来完成的, 但如果以其他用户身份建立索引, 则要求用户必须具有 CREATE ANY INDEX 系统权限或在相应表上的 INDEX 对象权限。语法:

```
CREATE [UNIQUE] INDEX index ON table(column[,column,...]);
```

- 建立单列索引

如果经常在 WHERE 子句中引用某个列, 那么应该考虑在该列上建立单列索引。

如: CREATE INDEX i_ename ON emp(ename)

下述语句将引用该索引:

```
SELECT sal,job FROM emp WHERE ename='SCOTT';
```

```
UPDATE emp SET sal=3000 WHERE ename='SCOTT';
```

```
DELETE FROM emp WHERE ename='SCOTT'
```

- 建立复合索引

如果经常在 WHERE 子句需要引用同一个表的多个列定位数据, 那么可以考虑在这些列上建立复合索引。当建立复合索引时, 索引列不能超过 32 个。如:

```
CREATE INDEX i_deptno_job ON emp(deptno,job);
```

建立了上述索引后, 如果在 WHERE 子句中使用 AND 谓词引用 deptno 和 job 列, 会使用该索引; 如果在 WHERE 子句中单独引用 deptno 列, 也会引用该索引, 如下述两语句都会使用上述索引:

```
SELECT ename,sal FROM emp WHERE deptno=20 AND job='CLERK';
```

```
SELECT ename,sal FROM emp WHERE deptno=20;
```

但需注意: 如果在 WHERE 子句中使用 OR 谓词引用 deptno 和 job 列, 或者单独引用 job 列, 都不会使用上述索引, 例:

```
SELECT ename FROM emp WHERE deptno=20 OR job='CLERK';
```

```
SELECT ename FROM emp WHERE job='CLERK';
```

- 建立非惟一索引

非惟一索引是指索引列值可以重复的索引。当建立索引时, 如果不指定 UNIQUE 选项, 则默认会建立非惟一索引。

如: CREATE INDEX i_dname ON dept(dname)

- 建立惟一索引

当定义主键约束或惟一约束时, Oracle 会自动基于约束列建立惟一索引; 建立索引时, 还可通过指定 UNIQUE 选项建立惟一索引。

如: CREATE UNIQUE INDEX i_dname ON dept(dname)

12.1.2 维护索引

当重新组织表之后, 会导致其索引转变为无效状态, 在这种情况下需重新建立索引; 当索引不再需要时, 应该删除索引。

重建索引语法: ALTER INDEX index REBUILD[ONLINE];

删除索引语法: DROP INDEX index;

- 重建索引
当执行 DELETE 操作时，会删除表数据，但在索引上只是进行逻辑删除，其所占用空间不能供其他插入操作使用。只有当索引块的所有索引入口被全部删除之后，该索引块上的空间才能使用。如果在索引列上频繁执行 UPDATE 或 DELETE 操作，那么应该定期重建索引，以提高其空间利用率。如：

```
ALTER INDEX i_job REBUILD;
```

- 联机重建索引
当使用 REBUILD 选项重建索引时，若其他用户正在表上执行 DML 操作，那么重建索引将会失败，为了最小化 DML 操作的影响，可以指定 ONLINE 选项。如：

```
ALTER INDEX i_job REBUILD ONLINE;
```

- 删除索引
索引主要用于提高查询速度，但会降低 DML 操作的速度，Oracle 曾经做过一种统计，分别在无索引的表上和存在三个索引的表上执行 INSERT 操作，前者插入速度要比后者插入速度快 10 倍。因此，如果索引很少使用，那么应该删除该索引。如：

```
DROP INDEX i_job;
```

12.1.3 显示索引信息

- USER_INDEXES
当建立索引时，Oracle 会将索引信息存放到数据字典。通过查询数据字典视图 USER_INDEXES，可以显示当前用户的所有索引。Oracle 为该数据视图提供了同义词 IND，所以可用 IND 取得索引信息，如：

```
SELECT index_name,uniqueness,status FROM ind  
WHERE table_name='EMP';
```

- USER_IND_COLUMNS
当建立索引时，Oracle 会将索引列的信息写入数据字典。通过查询数据字典视图 USER_IND_COLUMNS，可以显示当前用户索引列的信息。如：

```
SELECT column_name,column_position FROM user_ind_columns  
WHERE index_name='i_deptno_job';
```

11.2 使用序列

序列是一种用于生成惟一数字的数据库对象。序列生成器会自动生成顺序递增或递减的序列号，从而帮我们提供惟一的主键值。

11.2.1 建立序列

语法：CREATE SEQUENCE sequence
[INCREMENT BY n] 默认为 1，n 为正数，序列号递增，否递减
[START WITH n]
[**MAXVALUE** n | **NOMAXVALUE**]
[**CYCLE** | **NOCYCLE**]
[**CACHE** n | **OCACHE**];指内存中可预分配的序列号个数，默认 20

➤ 建立序列

```
CREATE SEQUENCE deptno_seq START WITH 50 INCREMENT BY 10  
MAXVALUE 99 CACHE 10;
```

11.2.2 使用序列

必须通过伪列 NEXTVAL 和 CURRVAL 引用，如：

```
SELECT deptno_seq.CURRVAL(NEXTVAL) FROM dual;
```

```
INSERT INTO dept (deptno,dname,loc)
```

```
VALUES(deptno_seq.NEXTVAL,'DEVELOPMENT',DEFAULT);
```

➤ 使用序列的注意事项

✧ 通过 **CACHE** 选项建立序列，可以设置在内存中预分配的序列号个数。该选项设置越大，序列的访问性能会越好，但也会占用更多的内存空间。

✧ 当执行 **ROLLBACK** 时，会导致出现序列缺口。

11.2.3 维护序列

✧ 修改序列

```
ALTER SEQUENCE sequence
```

```
  [INCREMENT BY n]
```

```
  [START WITH n]
```

```
  [{MAXVALUE n | NOMAXVALUE}]
```

```
  [{CYCLE | NOCYCLE}]
```

```
  [{CACHE n | NOCACHE}];
```

如：ALTER SEQUENCE deptno_seq MAXVALUE 200 CACHE 20;

✧ 删除序列

```
DROP SEQUENCE sequence;
```

若删除其他方案的序列，用户必须有 **DROP ANY SEQUENCE** 系统权限

11.2.4 显示序列信息

USER_SEQUENCES，同义词为 **SEQ**

```
SELECT increment_by,cache_size,max_value,last_number
```

```
FROM seq WHERE sequence_name='DEPTNO_SEQ'
```

11.3 使用同义词

同义词是方案对象的别名。通过使用同义词，一方面可简化对象访问，另一方面可以提高对象访问的安全性。

同义词包括公共同义词和私有同义词，公共同义词是指所有用户者可直接引用的同义词，并且这种同义词由 **PUBLIC** 用户组所拥有；私有同义词是指只能由其方案用户直接引用的同义词。

11.3.1 建立同义词

✧ 建立公共同义词

```
CREATE PUBLIC SYNONYM public_emp FOR scott.emp
```

因为该同义词属于 **PUBLIC** 用户组，所以任何用户都可以直接引用该同义词。需要注意的是，如果用户要使用该同义词，那必须具有访问 **SCOTT.EMP** 表的权限。

使用：UPDATE public_emp SET sal=3100 WHERE ename='SCOTT';

✧ 建立私有同义词

```
CREATE SYNONYM private_emp FOR [schema.]emp;
```

在当前方案中建立私有同义词，数据库用户须具有 **CREATE SYNONYM** 系统权限；在其他方案中建立私有同义词，数据库用户需具有 **CREATE ANY SYNONYM** 系统权限。

使用：SELECT ename,sal,job FROM scott.private_emp

11.3.2 删除同义词

- ✧ 删除公共同义词
用户必须具有 `DROP PUBLIC SYNONYM` 系统权限，语法如下：
`DROP PUBLIC SYNONYM synonym;`
- ✧ 删除私有同义词
用户可直接删除自身方案的同义词，如果要删除其他方案的同义词，必须有 `DROP ANY SYNONYM` 系统权限，语法如下：
`DROP SYNONYM synonym;`
- 显示同义词（视图 `USER——SYNONYMS`，同义词 `SYN`）
`SELECT table_owner,table_name FROM syn`
`WHERE synonym_name='PRIVATE_EMP';`

第 13 章 PL/SQL 语句

本章重点：

PL/SQL 简介

PL/SQL 块

13.1 PL/SQL 简介

PL/SQL 是 Oracle 在标准 SQL 语言上的过程性扩展，当编写 PL/SQL 应用模块时，开发人员不仅需要掌握 SQL 语句的编写方法，而且必须要掌握 PL/SQL 语句及其语法规则。

PL/SQL 不仅允许嵌入 SQL 语句，而且允许定义变量和常量，允许过程语言结构（条件分支语句和循环语句），允许使用例外处理 Oracle 错误等，在运行 Oracle 的任何平台中应用开发人员都可以使用 PL/SQL。

PL/SQL 具有以下一些优点和特征：

- 提高应用程序的运行性能
- 提供了模块化的程序设计功能
- 允许定义标识符
- 具有过程语言控制结构
- 具有良好的兼容性
- 处理运行错误

在 oracle 中，其 pl/sql 语句控制与普通高级语言类似。也有 if、loop、while、go to 等语句控制，下面只是简单的列一下他们的格式，需要在使用过程中进行体会。

IF 语句：

```
IF <Expr> THEN
```

```
Statement;[Statement Block]
```

```
[ELSIF <Expr> THEN
```

```
Statement;[Statement Block]]
```

```
[ELSE
    Statement;[Statement Block]]
END IF ;
```

CASE 语句:

```
CASE selector
    WHEN expression1 THEN sequence_of_statements1;
    WHEN expression2 THEN sequence_of_statements2;
    ...
    WHEN expressionN THEN sequence_of_statementsN;
END CASE;
```

基本循环语句:

```
LOOP
    Statement;[Statement Block]
    [EXIT WHEN Condition;]
END LOOP;
```

WHILE 循环:

```
WHILE Condition LOOP
    Statement;[Statement Block]
END LOOP;
```

FOR 循环:

```
FOR Counter IN [REVERSE] start..end LOOP
    Statement;[Statement Block]
END LOOP;
```

[注释: Counter 是循环控制变量, 并且该变量由 Oracle 隐含定义, 不需要显式定义; start 和 end 分别对应于循环控制变量的下界值和上界值。默认情况下, 当使用 FOR 循环时, 每次循环时循环控制变量会自动增一; 如果指定 REVERSE 选项, 那么循环控制变量会自动减一。]

嵌套循环和标号

在编写 PL/SQL 块时, 可以使用《label_name》定义标号, 如:

```
set serveroutput on
DECLARE
    result INT;
```

```
BEGIN
  <<outer>>
  FOR i IN 1..100 LOOP
    <<inter>>
    FOR j IN 1..100 LOOP
      result:=i*j;
      EXIT outer WHEN result=1000;
      EXIT WHEN result=500;
    END LOOP inner;
    dbms_output.put_line(result);
  END LOOP outer;
  dbms_output.put_line(result);
END;
```

13.2 PL/SQL 块

PL/SQL 块是 PL/SQL 的基本执行单元，编写 PL/SQL 程序实际就是编写 PL/SQL 块。

PL/SQL 块由定义部分、执行部分和例外处理部分组成。其中，定义部分用于定义常量、变量、游标、例外、复杂数据类型等；执行部分用于实现应用模块功能，该部分包含了要执行的 PL/SQL 语句和 SQL 语句；例外处理部分用于处理执行部分可能出现的运行错误。

PL/SQL 语句块的基本结构如下：

```
DECLARE
[Declare Variable]
BEGIN
Statement;[PL/SQL Statement Block]
[EXCEPTION
  WHEN Exception THEN
    Statement;]
END ;
```

示例 1：只包含执行部分的 PL/SQL 块

```
Set serveroutput on
BEGIN
--
/*
*Dbms_output.put_line( 'Hello, everyone!' );
*/
```

```
END;
```

```
/
```

示例 2: 只包含定义部分和执行部分的 PL/SQL 块

```
DECLARE
```

```
    v_name VARCHAR2(5);
```

```
BEGIN
```

```
    SELECT ename INTO v_ename FROM emp WHERE empno=&no;
```

```
    Dbms_output.put_line('雇员名: ' || v_ename);
```

```
END;
```

```
/
```

示例 3: 包含定义部分、执行部分和异常处理部分的 PL/SQL 块

```
DECLARE
```

```
    v_ename VARCHAR2(5)
```

```
BEGIN
```

```
    SELECT ename INTO v_ename FROM emp WHERE empno=&no;
```

```
    Dbms_output.put_line('雇员名: ' || v_ename);
```

```
EXCEPTION
```

```
    WHEN NO_DATA_FOUND THEN
```

```
        Dbms_output.put_line('请输入正确的雇员号!');
```

```
END;
```

```
/
```

当使用 PL/SQL 开发应用模块时, 根据要实现的应用模块功能, 可以将 PL/SQL 块划分为匿名块、命名块、子程序 (存储过程、函数、包) 和触发器等 4 种类型

13.2.1 定义并使用变量

当编写 PL/SQL 块时, 为了临时存储数据, 需要定义变量和常量。为了在应用环境和子程序之间传递数据, 需要为子程序定义参数。当定义变量、常量和参数时, 需要指定合适的 PL/SQL 数据类型, PL/SQL 包括标题类型、复合类型、参照类型和 LOB 类型等 4 种类型。

如 name VARCHAR2(10)

name emp.ename%TYPE; 等

当在 SQL*Plus 或应用程序（如 Pro*C/C++）中与 PL/SQL 块之间进行数据交换时，需要使用 SQL*Plus 变量或应用程序变量来完成。当使用 SQL*Plus 变量时，需要使用 VAR[TABLE] 命令定义变量，并且使用 PRINT 命令可以输出变量内容；当在 PL/SQL 块中引用非 PL/SQL 变量时，需要在非 PL/SQL 变量前加冒号（:）。用法如下：

- 使用 SQL*Plus 变量

当使用 SQL*Plus 变量时，需要使用 VAR[TABLE] 命令定义变量，并且使用 PRINT 命令可以输出变量内容。如

```
SQL>var name varchar2(10)

SQL>BEGIN

2  SELECT ename INTO :name FROM emp

3  WHERE empno=&eno;

4  END;

5  /

SQL>PRINT name
```

- 使用 Pro*C/C++变量

当在 PL/SQL 块中引用 Pro*C/C++程序的宿主变量时，需要定义宿主变量，并且使用 printf（）函数可以输出变量。如

```
Char name[10];

EXEC SQL EXECUTE

    BEGIN

        SELECT ename INTO :name FROM emp

        WHERE empno=7788;

    END;

END-EXEC;

Printf(“雇员名: %s\n”, name);
```

13.2.2 异常

异常是一种 PL/SQL 标识符，用于处理 PL/SQL 程序的运行错误。为了提高 PL/SQL 程序的健壮性，开发人员必须要考虑 PL/SQL 程序可能出现的各种错误，并编写相应的例外处理部分。如果不进行错误处理，那么当执行 PL/SQL 块时会将错误传递到调用块或 PL/SQL 运行环境，终止 PL/SQL 程序的运行，并显示错误信息。

为了处理 PL/SQL 应用程序的各种错误，开发人员可以使用各种类型的例外。Oracle 提供了预定义例外、非预定义例外和自定义例外等三种例外类型，其中预定义例外用于处理

常见的 Oracle 错误，非预定义例外用于处理预定义例外所不能处理的 Oracle 错误，自定义例外用于处理与 Oracle 错误无关的其他情况。在 PL/SQL 块中捕捉并处理需要使用例外处理部分来完成，语法如下：

```
EXCEPTION
  WHEN exception1 [OR exception2...] THEN
    statement1;
    statement2;
    ...
  WHEN exception3 [OR exception4...] THEN
    statement1;
    statement2;
    ...
  WHEN OTHERS THEN
    statement1;
    statement2;
    ...
WHEN OTHERS 必须是例外处理部分的最后一条子句
```

1) 处理预定义例外

预定义例外是指由 PL/SQL 所提供的系统例外，PL/SQL 为开发人员提供了 20 多个预定义例外，每个预定义例外都对应一个 Oracle 错误。下面列出常用的预定义例外。

- NO_DATA_FOUND: 对应 ORA-01403 错误。当执行 SELECT INTO 未返回行，或者引用未初始化的 PL/SQL 表元素时，会隐含地触发该例外
- TOO_MANY_ROWS: 对应 ORA-01422 错误。当执行 SELECT INTO 语句时，如果返回超过一行，则会触发该例外
- DUP_VAL_ON_INDEX: 对应 ORA-00001 错误。当在惟一索引所对应的列上键入重复值时，会隐含地触发该例外
- ZERO_DIVIDE: 对应于 ORA-01476 错误。当运行 PL/SQL 块时，如果使用数字值除 0，则会隐含地触发该例外。
- INVALID_CURSOR: 对应 ORA-01001 错误。当试图在不合法的游标上执行操作时，会隐含地触发该例外。

当使用预定义例外处理 Oracle 错误时，可以直接在例外处理部分引用预定义例外。

如：

```
DECLARE
  v_ename emp.ename%TYPE;
BEGIN
  SELECT ename INTO v_ename FROM emp
  WHERE empno=&no;
  dbms_output.put_line('雇员名：' || v_ename);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    dbms_output.put_line('该雇员不存在');
END;
```

2) 处理非预定义例外

非预定义例外用于处理与预定义例外无关的 Oracle 错误。当使用预定义例外时，只能处理 21 个 Oracle 错误。为了处理其他 Oracle 错误，必须使用非预定义例外。使用步骤如下：



使用非预定义例外包括三步：首先在定义部分定义例外名，然后在例外和 Oracle 错误之间建立关联，最终在例外处理部分捕捉并处理例外。当定义 Oracle 错误和例外之间的关联关系时，需要使用 EXCEPTION_INIT。如：

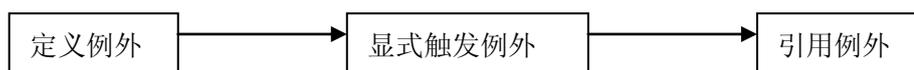
```

DECLARE
  e_integrity EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_integrity,-2291);
BEGIN
  UPDATE emp SET deptno=&dno WHERE empno=&eno;
EXCEPTION
  WHEN e_integrity THEN
    dbms_output.put_line(' 该部门不存在');
END;
  
```

PRAGMA 由编译器控制，或者是对于编译器的注释。**PRAGMA** 在编译时处理，而不是在运行时处理。**EXCEPTION_INIT** 告诉编译器将异常名与 **ORACLE** 错误码结合起来，这样可以通过名字引用任意的内部异常，并且可以通过名字为异常编写一适当的异常处理器。

3) 处理自定义例外

自定义例外是指由 PL/SQL 开发人员所定义的例外。预定义例外和非预定义例外都与 Oracle 错误有关，并且当出现 Oracle 错误时会隐含触发相应例外；而自定义例外与 Oracle 错误没有任何关联，它是由开发人员为特定情况所定义的例外。使用自定义例外的步骤如下：



当使用自定义例外时，首先需要在定义部分（DECLARE）定义例外，然后在执行部分（BEGIN）触发例外（使用 RAISE 语句），最后在例外处理部分（EXCEPTION）捕捉并处理例外。如：

```

DECLARE
  e_integrity EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_integrity,-2291);
  e_no_employee EXCEPTION;
BEGIN
  UPDATE emp SET deptno=&dno WHERE empno=&eno;
  IF SQL%NOTFOUND THEN
    RAISE e_no_employee;
  END IF;
EXCEPTION
  
```

```
WHEN e_integrity THEN
    dbms_output.put_line(' 该部门不存在');
WHEN e_no_employee THEN
    dbms_output.put_line(' 该雇员不存在');
END;
[SQL%NOTFOUND 是隐式游标的属性]
```

13.2.2 游标

当在 PL/SQL 块中执行查询语句 (SELECT) 和数据操纵语句 (DML) 时, Oracle 会为其分配上下文区, 游标是指向上下文区的指针。对于数据操纵语句和单行 SELECT INTO 语句来说, Oracle 会为它们分配隐含游标。为了处理 SELECT 语句返回的多行数据, 开发人员需要使用显式游标。

1. 显式游标

显式游标专门用于处理 SELECT 语句返回的多行数据。

1) 使用显式游标

使用显式游标包括定义游标、打开游标、提取数据和关闭游标四个阶段

a) 定义游标

使用显式游标之前, 必须首先在定义部分定义游标。定义游标用于指定游标所对应的 SELECT 语句, 语法如下:

```
CURSOR cursor_name IS select_statement;
```

b) 打开游标

当打开游标时, Oracle 会执行游标所对应的 SELECT 语句, 并且将 SELECT 语句的结果存放到游标结果集。语法如下:

```
OPEN cursor_name;
```

c) 提取数据

打开游标之后, SELECT 语句的结果临时存放到游标结果集。为了处理游标结果集的数据, 需要使用 FETCH 语句提取游标数据, 语法

```
FETCH cursor_name INTO variable1, variable2, ...;
```

FETCH INTO 只能提取单行数据, 为了处理所有数据, 必须使用循环语句。

d) 关闭游标

提取并处理了结果集的所以数据之后, 就可以关闭游标并释放其结果集了。语法如下:

```
CLOSE cursor_name;
```

2) 显式游标属性

显式游标属性用于返回显式游标的执行信息，这些属性包

括%ISOPEN、%FOUND、%NOTFOUND 和%ROWCOUNT。

%ISOPEN：用于确定游标是否已经打开。如果游标已经打开，则返回 TRUE，否则返回 FALSE

%FOUND：用于检测是否从结果集中提取到数据。取到返回 TRUE

%NOTFOUND：用于检测是否未从结果集中提取到数据。取到返回 FALSE

%ROWCOUNT：用于返回到当前行为止已经取到的实际行数

示例：

```
DECLARE

    CURSOR emp_cursor IS

        SELECT ename, sal FROM emp WHERE deptno=1;

    v_ename emp.ename%TYPE;

    v_sal emp.sal%TYPE;

BEGIN

    IF NOT emp_cursor%ISOPEN THEN

        OPEN emp_cursor;

    END IF;

    LOOP

        FETCH emp_cursor INTO v_ename, v_sal;

        EXIT WHEN emp_cursor%NOTFOUND;

        dbms_output.put_line(v_ename||': '||v_sal);

    END LOOP;

    CLOSE emp_cursor;

END;
```

2. 参数游标

参数游标是指带参数的游标。在定义了参数游标之后，当使用不同参数值多次打开游标时，可以生成不同的结果集。定义参数游标的语法如下：

```
CURSOR cursor_name(parameter_name datatype)

    IS select_statement;
```

示例如下:

```
DECLARE

    CURSOR emp_cursor(no NUMBER) IS

        SELECT ename FROM emp WHERE deptno=no;

    v_ename emp.ename%TYPE;

BEGIN

    OPEN emp_cursor(&n);

    LOOP

        FETCH emp_cursor INTO v_ename;

        EXIT WHEN emp_cursor%NOTFOUND;

        dbms_output.put_line('雇员名: '||v_ename);

    END LOOP;

    CLOSE emp_cursor;

END;
```

3. 游标 FOR 循环

游标 FOR 循环是在 PL/SQL 块中使用游标的最简单方式, 它简化了对游标的处理。当使用游标 FOR 循环时, Oracle 会隐含地打开游标、提取游标数据并关闭游标。使用游标 FOR 循环的语法如下:

```
FOR record_name IN cursor_name LOOP

    statement1;

    statement2;

    ...

END LOOP;
```

其中, `cursor_name` 是已经定义的游标名, `record_name` 是 Oracle 隐含定义的记录变量名。示例如下:

```
DECLARE

    CURSOR emp_cursor IS

        SELECT ename, sal FROM emp;

BEGIN

    FOR emp_record IN emp_cursor LOOP
```

```
dbms_output.put_line('雇员名: ' || emp_record.ename ||  
    ', 工资: ' || emp_record.sal);  
END LOOP;  
END;
```

4. 使用游标更新或删除数据

通过使用显式游标，不仅可以一行一行地处理 SELECT 语句的结果，而且可以更新或删除当前游标行的数据。需要注意的是，如果通过游标更新或删除数据，在定义游标时必须带有 FOR UPDATE 子句。语法如下：

```
CURSOR cursor_name IS select_statement  
    FOR UPDATE[OF column_reference] [NOWAIT]
```

其中，FOR UPDATE 子句用于指定在游标结果集上加锁，OF 子句用于指定在特定表上加锁，NOWAIT 子句用于指定不等待锁。在提取了游标数据之后，为了更新或删除当前游标行数据，必须在 UPDATE 或 DELETE 语句中引用 WHERE CURRENT OF 子句。语法如下：

```
UPDATE table_name SET column=. WHERE CURRENT OF cursor_name;  
DELETE table_name WHERE CURRENT OF cursor_name;
```

示例如下：

```
DECLARE  
    CURSOR emp_cursor IS  
        SELECT * FROM emp FOR UPDATE;  
BEGIN  
    FOR emp_record IN emp_cursor LOOP  
        IF emp_record.deptno=3 THEN  
            UPDATE emp SET sal=sal+200 WHERE CURRENT OF emp_cursor;  
            dbms_output.put_line('雇员名: ' || emp_record.ename ||', 部门号:  
                ' || emp_record.deptno ||  
                ', 原工资: ' || emp_record.sal ||', 新工资: ' || (emp_record.sal+200));  
        ELSE  
            dbms_output.put_line('姓名: ' || emp_record.ename ||', 部门号:  
                ' || emp_record.deptno ||', 工资: ' ||  
                emp_record.sal);  
        END IF;  
    END LOOP;  
END;
```

```
END IF;  
  
END LOOP;  
  
END;
```

13.3 过程函数包

对于过程、函数、包等形式的语句块，有自己的定义方式。对于包，大家可以多使用，因为包有很多的优点：1、全局变量定义；2、函数重载；3、便于维护和存储。包是函数、过程、变量、游标等的一个集合体。

第 14 章 过程函数包及触发器

本章内容：

- ✧ 过程
- ✧ 函数
- ✧ 包
- ✧ 触发器

14.1 过程

过程用于执行特定操作。如果应用程序经常需要执行特定操作，那么可以基于这些操作建立特定的过程。通过使用过程，不仅可以简化客户端应用程序的开发和维护，而且可以提高应用程序的运行性能。

14.1.1 建立过程

建立过程是使用 `CREATE PROCEDURE` 语句来完成的。如果用户要建立过程，那么要求该用户必须具有 `CREATE PROCEDURE` 系统权限。建立过程的语法：

```
CREATE [OR REPLACE] PROCEDURE procedure_name  
(argument1 [model] datatype1 [:=|DEFAULT initial_value ], argument2 [mode2]  
datatype2,...)  
IS[AS]  
PL/SQL Block;
```

其中，`procedure name` 用于指定过程名称，`argument` 用于指定过程参数，`IS` 或 `AS` 用于开始 `PL/SQL` 块。注意的是，当定义过程参数时，只能指定数据类型，不能指定长度。另外，当建立过程时，既可以指定输入参数（`IN`），也可以指定输出参数（`OUT`）及输入输出参数（`IN OUT`）。通过在过程中使用输入参数，可以将应用环境的数据传递到执行部分；通过使用输出参数，可以将执行部分的数据传递到应用环境。当定义参数时，如果不指定参数模式，则默认为输入参数；如果定义输出参数，那么需要指定 `OUT` 关键字；如果定义输入输出参数，那么需要指定 `IN OUT`

关键字。如果过程已经建立，那么使用 CREATE OR REPLACE PROCEDURE 可以替换过程代码。如

```
CREATE OR REPLACE PROCEDURE add_emp
(eno NUMBER,name VARCHAR2,sal NUMBER,job VARCHAR2 DEFAULT
'CLERK',dno NUMBER)
IS
E_integrity EXCEPTION;
PRAGMA EXCEPTION_INIT(e_integrity,-2291);
BEGIN
    INSERT INTO emp(empno,ename,sal,job,deptno)
        VALUES(eno,name,sal,job,dno);
EXCEPTION
    WHEN DUP_VAL_ONINDEX THEN
        RAISE_APPLICATION_ERROR(-20000,'雇员号不能重复');
    WHEN E_integrity THEN
        RAISE_APPLICATION_ERROR(-20001,'部门号不存在');
END;
```

因为在建立过程 ADD_EMP 时所有参数都没有指定参数模式，所以这些参数全部都是输入参数。当调用该过程时，除了具有默认值的参数之外，其他参数必须要提供数值。调用示例如下：

```
exec add_emp(1111,'MARY',2000,'MANAGER',10)
SELECT ename,sal FROM emp WHERE empno=1111;
```

1) 删除过程

语法：

```
DROP PROCEDURE procedure_name;
```

14.1.2 显示过程代码

通过查询数据字典视图 USER_SOURCE，可以显示过程源代码。示例如下：

```
SELECT text FROM user_source WHERE name='ADD_EMP'
```

14.2 函数

函数用于返回特定数据。如果应用程序经常需要执行 SQL 语句返回特定数据，那么可以基于这些操作建立特定的函数。通过使用函数，不仅可以简化客户端应用程序的开发和维护，而且可以提高应用程序的执行性能。

14.2.1 建立函数

建立函数是使用 CREATE FUNCTION 语句来完成的。如果用户要建立函数，则要求该用户必须具有 CREATE PROCEDURE 系统权限。建立函数的语法如下：

```
CREATER [OR REPLACE] FUNCTION function_name
(argument1 [mode1] datatype1,argument2 [mode2] datatype2,...)
RETURN datatype
IS | AS
PL/SQL Block;
```

其中，function_name 用于指定函数名称，argument 用于指定函数的参数，IS 或 AS 用于开始 PL/SQL 块。需要注意的是，当建立函数时，在函数头部必须要带

有 RETURN 子句，在函数体内至少要包含一条 RETURN 语句。另外，当建立函数时，既可以指定输入参数（IN），也可以指定输出参数（OUT）及输入输出参数（IN OUT）。另外，如果函数已经建立，那么使用 CREATE OR REPLACE FUNCTION 可以替换函数代码。例：

```
CRATE OR REPLACE FUNCTION get_sal(name IN VARCHAR2)
RETURN NUMBER
AS
    v_sal emp.sal%TYPE;
BEGIN
    SELECT sal INTO v_sal FROM emp
        WHERE upper(ename)=upper(name);
    RETURN v_sal;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        Raise_application_error(-20000,'该雇员不存在');
END;
```

当建立了函数 get_sal 以后，就可以在应用程序中调用该函数了。调用该函数的示例如下：

```
var sal NUMBER
exec :sal:=get_sal('scott')
print sal
```

14.2.2 删除函数

14.2.3 显示函数代码

14.3 包

包用于逻辑组合相关的 PL/SQL 类型、项和子程序，它由包规范（Package Specification）和包体（Package Body）两部分组成。当建立包时，道德建立包规范，然后建立包体。建立包规范是使用 CREATE PACKAGE 来完成的，建立包体是使用 CREATE PACKAGE BODY 命令来完成的。如果用户要建立包，则要求用户必须具有 CREATE PROCEDURE 系统权限。

14.3.1 建立包规范

包规范实际是包与应用程序之间的接口，它用于定义包的公用组件，包括常量、变量、游标、过程和函数等。在包规范中所定义的公用组件不仅可以在包内引用，而且可以由其他的子程序引用。假定在定义包规范 EMP_PACKAGE 时定义了公用变量 g_deptno、公用过程 add_employee 和 fire_employee，以及公用函数 get_sal，那么它们不仅可以在包 EMP_PACKAGE 内引用，而且也可以由其他子程序引用。建立包规范语法如下：

```
CREATE [OR REPLACE] PACKAGE package_name
IS | AS
    public type and item declarations
    subprogram specifications
END package_name;
```

示例如下:

```
CREATE OR REPLACE PACKAGE emp_package IS
    g_deptno NUMBER(3):=30;
    PROCEDURE add_employee(eno NUMBER,name VARCHAR2,salary
        NUMBER,dno NUMBER DEFAULT g_deptno);
    PROCEDURE fire_employee(eno NUMBER);
    FUNCTION get_sal(eno NUMBER) RETURN NUMBER;
END emp_package;
/
```

当执行了以上命令之后, 会建立包规范 `emp_package`, 并且定义了所有公用组件, 但因为只定义了过程和函数的头部, 没有编写过程和函数的执行代码, 所以公用的过程和函数只有在建立了包体之后才能调用。

14.3.2 建立包体

包体用于实现包规范所定义的过程和函数。当建立包体时, 也可以单独定义私有组件, 包括变量、常量、过程和函数等, 但在包体中所定义的私有组件只能在包内使用, 而不能由其它子程序引用。假定在建立包体 `EMP_PACKAGE` 时定义了函数 `validate_deptno`, 那么该函数只能在包 `EMP_PACKAGE` 内使用, 而不能由其他子程序调用。

当建立包时, 为了实现信息隐藏, 应该在包体内定义私有组件; 为了实现包规范中所定义的公用过程和函数, 必须要建立包体。建立包体是使用命令 `CREATE PACKAGE BODY` 来完成的, 语法如下:

```
CREATE [OR REPLACE] PACKAGE BODY package_name
IS | AS
    private type and item declarations
    subprogram bodies
END package_name;
```

示例如下:

```
CREATE OR REPLACE PACKAGE BODY emp_package IS
    FUNCTION validate_deptno(v_deptno NUMBER)
        RETURN BOOLEAN
    IS
        v_temp INT;
    BEGIN
        SELECT 1 INTO v_temp FROM dept WHERE deptno=v_deptno;
        RETURN TRUE;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RETURN FALSE;
    END;
    PROCEDURE add_employee(eno NUMBER,name VARCHAR2,
        Salary NUMBER,dno NUMBER DEFAULT g_deptno)
    IS
    BEGIN
        IF validate_deptno(dno) THEN
```

```
INSERT INTO emp(empno,ename,sal,deptno)
VALUES(eno,name,salary,dno);
ELSE
  Raise_application_error(-20010,'不存在该部门');
END IF;
EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    raise_application_error(-20011,'该雇员已存在');
END;
PROCEDURE fire_employee(eno NUMBER) IS
BEGIN
  DELETE FROM emp WHERE empno=eno;
  IF SQL%NOTFOUND THEN
    raise_application_error(-20012,'该雇员不存在');
  END IF;
END;
FUNCTION get_sal(eno NUMBER) RETURN NUMBER
IS
  v_sal emp.sal%TYPE;
BEGIN
  SELECT sal INTO v_sal FROM emp WHERE empno=eno;
  RETURN v_sal;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    raise_application_error(-20012,'该雇员不存在');
END;
END emp_package;
```

14.3.3 删除包

语法如下：

```
DROP PACKAGE package_name; 删除包规范和包体
```

```
DROP PACKAGE BODY package_name; 删除包体
```

14.3.4 显示包代码

通过查询 USER_SOURCE，可以显示包规范或者包体的代码。如：

```
SELECT text FROM user_source
```

```
WHERE name='EMP_PACKAGE' AND type='PACKAGE';
```

14.4 触发器

触发器是指被隐含执行的存储过程，它可以使用 PL/SQL、Java 和 C 进行开发。在建立了触发器之后，如果发生了相应的 DML 操作，那么会自动执行触发器的相应代码。触发器包括语句触发器和行触发器两种类型。在触发器只能包含 SELECT 和 DML 语句，不能包含 DDL、DCL 和事务控制语句。

14.4.1 语句触发器

语句触发器是指当执行 DML 语句时被隐含执行的触发器。如果在表上针对某种 DML 操作建立了语句触发器，那么当执行 DML 操作时会自动执行触

发器的相应代码。当审计 DML 操作，或者确保 DML 操作安全执行时，可以使用语句触发器。当使用语句触发器时，不能记录列数据的变化。建立语句触发器的语法如下：

```
CREATE [OR REPLACE] TRIGGER trigger_name
    timing event1 [OR event2 OR event3]
    ON table_name
    PL/SQL block;
```

其中，`timing` 用于指定触发时机（`BEFORE` 或 `AFTER`）；`event` 用于指定触发事件（`INSERT`、`UPDATE` 和 `DELETE`）；`table_name` 用于指定 DML 操作所对应的表名。示例：禁止工作人员在休息日改变雇员信息

```
CREATE OR REPLACE TRIGGER tr_sec_emp
BEFORE INSERT OR UPDATE OR DELETE ON emp
BEGIN
    IF to_char(sysdate,'DY','nls_date_language=AMERICAN')
        IN('SAT','SUN') THEN
        raise_application_error(-20001,'不能在休息日改变雇员信息');
    END IF;
END;
/
```

当建立了该触发器之后，如果星期六、星期日的在 EMP 表上执行 DML 操作，则会显示错误信息。

14.4.2 行触发器

行触发器是指当执行 DML 操作时，每作用一行被触发一次的触发器。当审计数据变化时，可以使用行触发器。建立行触发器语法：

```
CREATE [OR REPLACE] TRIGGER trigger trigger_name
    timing event1 [OR event2 OR event3]
    ON table_name
    [REFERENCING OLD AS old | NEW AS new]
    FOR EACH ROW
    [WHEN condition]
    PL/SQL block;
```

其中，`timing` 用于指定触发时机（`BEFORE` 或 `AFTER`）；`event` 用于指定触发事件（`INSERT`、`UPDATE`、`DELETE`）；`REFERENCING` 子句用于指定引用新、旧数据的方式，默认情况下使用 `old` 修饰符引用旧数据，使用 `new` 修饰符引用新数据；`table_name` 用于指定 DML 操作所对应的表；`FOR EACH ROW` 表示建立行触发器；`WHEN` 子句（可选）用于指定触发条件。示例如下：

```
CREATE TABLE audit_emp_change(
    name VARCHAR2(10),oldsal NUMBER(6,2),
    newsal NUMBER(6,2),time DATE);
```

```
CREATE OR REPLACE TRIGGER tr_sal_change
AFTER UPDATE OF sal ON emp
FOR EACH ROW
DECLARE
```

```
v_temp INT;
BEGIN
  SELECT count(*) INTO v_temp FROM audit_emp_change
    WHERE name=:old.ename;
  IF v_temp=0 THEN
    INSERT INTO audit_emp_change
      VALUES(:old.ename,:old.sal,:new.sal,SYSDATE);
  ELSE
    UPDATE audit_emp_change
      SET oldsal=:old.sal,newsal=:new.sal,time=SYSDATE
      WHERE name=:old.ename;
  END IF;
END;
/
```

当建立了触发器 `tr_sal_change` 之后，当修改雇员工资时，会将每个雇员的工资变化全部写入到审计表 `audit_emp_change` 中。示例如下：

```
UPDATE emp SET sal=sal*1.1 WHERE deptno=30;
```

14.4.3 使用触发器的注意事项

当编写 DML 触发器时，触发器代码不能从触发器所对应的基表中读取数据

14.4.4 编译触发器

当修改表结构时，会导致触发器转变为无效状态。为了使得触发器生效，必须重新编译触发器，语法如下：

```
ALTER TRIGGER tr_sal_change COMPILE;
```

14.4.5 删除触发器

语法：

```
DROP TRIGGER trigger_name;
```

14.4.6 显示触发器代码

通过查询数据字典视图 `USER_TRIGGERS`，可以显示触发器详细信息。如：

```
SELECT trigger_body FROM user_triggers
  WHERE trigger_name='TR_SAL_CHANGE'
```

第十五章 使用 EXP 和 IMP

EXP（导出）和 IMP（导入）的作用：

EXP 和 IMP 不仅可用于实现逻辑备份和逻辑恢复，而且还可以用于实现许多其他功能。导出和导入具有以下作用：

- 使用导出和导入可以重新组织表。例如，使用 EXP 和 IMP 可以删除行迁移。

- 使用导出和导入可以在用户之间移动对象。例如，使用 EXP 和 IMP 可以将 SCOTT 用户的对象移动到 SMITH 用户中。
- 使用导出和导入可以在数据库之间移动对象。例如，使用 EXP 和 IMP 可以将 OLTP 系统的对象移动到 DSS 系统中。
- 使用导出和导入可以升级数据库到其他平台。例如，使用 EXP 和 IMP 可以将 Windows 平台中的数据库对象移动到 Solaris 平台中。
- 使用导出和导入可以升级数据库到更高版本。例如，使用 EXP 和 IMP 可以将 8I 版本的数据库对象升级到 9I 数据库中。
- 使用导出和导入可以实现逻辑备份和恢复。例如，使用 EXP 可以导出数据库对象及数据到 OS 文件中；而当误删除了数据库对象之后，可以使用 IMP 将对象及数据导入到数据库中。

15.1 使用 EXP

导出是指使用实用工具 EXP 将数据库对象的结构及其数据转储到特定 OS 文件中的过程，导出又包括导出表、导出方案和导出数据库等三种方式。

EXP 是客户端的工具，该工具不仅可以在 Oracle 客户端使用，也可以在 Oracle 服务器端使用。当在 Oracle 客户端使用 EXP 工具时，必须要带有连接字符串；当在 Oracle 服务器端使用 EXP 工具时，可以不带连接字符串。导出包括导出表、导出方案、导出数据库等三种模式。

15.1.1 导出表

导出表是指使用 EXP 工具将一个或多个表的结构和数据存储到 OS 文件中，导出表是使用 TABLES 选项来完成的。

普通用户可以导出其方案的所有表，但如果要导出其他方案的表，则要求该用户必须具有 EXP_FULL_DATABASE 角色或者 DBA 角色。另外当导出表时，默认情况下会导出相应表上的所有索引、触发器、约束。下面以 SYSTEM 用户和 SCOTT 用户分别导出 SCOTT.DEPT 和 SCOTT.EMP 表为例，说明导出表的方法。示例如下：

```
Exp system/manager TABLES=scott.dept,scott.emp FILE=tab1.dmp
```

```
Exp scott/tiger TABLES=dept,emp FILE=tab2.dmp
```

15.1.2 导出方案

导出方案是指使用 EXP 工具将一个或多个方案中的所有对象及数据存储到 OS 文件中，导出方案是使用 OWNER 选项来完成的。

普通用户可以导出其自身方案，但如果要导出其他方案，则要求该用户必须具有 EXP_FULL_DATABASE 角色或者 DBA 角色。当用户要导出其自身方案的所有对象时，可以不指定 OWNER 选项。下面分别以 SYSTEM 用户和 SCOTT 用户导出 SCOTT 方案的所有对象为例，说明导出方案的方法，示例如下：

```
Exp system/manager OWNER=scott FILE=schema1.dmp
```

```
Exp scott/tiger FILE=schema2.dmp
```

15.1.3 导出数据库

导出数据库是指使用工具 EXP 将所有数据库对象及其数据存储到特定 OS 文件中，导出数据库是使用 FULL 选项来完成的。

读者需要注意，导出数据库要求用户必须具有 EXP_FULL_DATABASE 角色或者 DBA 角色，并且导出数据库不会导出 SYS 方案的任何对象。下面以导出 DEMO 数据库的所有对象为例，说明导出数据库的方法。示例如下：

```
Exp system/manager FULL=y FILE=full.dmp
```

15.2 使用 IMP

导入是指使用实用工具 IMP 将 EXP 导出文件中的对象及其数据装载到 Oracle 数据库中，导入包括导入表、导入方案、导入数据库等三种模式。

15.2.1 导入表

导入表是指使用 IMP 工具将 EXP 文件中的表结构及其数据装载到数据库中，导入表是使用 TABLES 选项来完成的。

普通用户可以直接导入其所拥有的表，但如果要将表导入到其他用户中，则要求该用户必须具有 EXP_FULL_DATABASE 角色或者 DBA 角色。读者需要注意，如果要将表导入到其他用户中，那么需要指定 FROMUSER 和 TOUSER 选项。示例如下：

```
imp scott/tiger FILE=tab2.dmp TABLES=dept,emp
imp system/manager FILE=tab1.dmp TABLES= dept, emp
FROMUSER=scott TOUSER=system
```

15.2.2 导入方案

导入方案是指使用 IMP 工具将 EXP 文件中特定方案的所有对象及数据装载到数据库中。

普通用户可以导入其自身方案，并且在导入时只需要提供 USERID 和 FILE 选项即可。但如果要将一个方案的所有对象导入到其他方案中，那么要求该用户必须具有 EXP_FULL_DATABASE 角色或者 DBA 角色，并且必须要提供 FROMUSER 和 TOUSER 选项，示例如下：

```
imp scott/tiger FILE=schema2.dmp
imp system/manager FILE=schema1.dmp FROMUSER=scott TOUSER=system
```

15.2.3 导入数据库

导入数据库是指使用工具 IMP 将 EXP 文件中所有用户的对象及数据装载到 Oracle 数据库中，导入数据库是使用 FULL 选项来完成的。

导入数据库要求用户必须具有 EXP_FULL_DATABASE 角色或者 DBA 角色，读者需要注意，因为在导出文件中没有包含 SYS 方案的对象，所以在导入时也不会包含 SYS 方案的对象。示例如下：

```
imp system/manager FILE=full.dmp FULL=y
```

decode()函数的用法

```
select decode(x, 1, 'x is 1', 2, 'x is 2', 'others') from dual
```

当 x 等于 1 时，则返回'x is 1'。

当 x 等于 2 时，则返回'x is 2'。

否则，返回 others'。

需要，比较 2 个值的时候，可以配合 SIGN()函数一起使用。

```
SELECT DECODE( SIGN(5 - 6), 1 'Is Positive', -1, 'Is Nagative', 'Is Zero')
```

同样，也可以用 CASE 实现：

```
SELECT CASE SIGN(5 - 6)
WHEN 1 THEN 'Is Positive'
WHEN -1 THEN 'Is Nagative'
ELSE 'Is Zero' END
FROM DUAL
```

自己补充:

1、oracle with as 用法

with

```
sql1 as (select to_char(a) s_name from test_tempa),
```

```
sql2 as (select to_char(b) s_name from test_tempb where not exists (select s_name from sql1  
where rownum=1))
```

```
select * from sql1
```

```
union all
```

```
select * from sql2
```

```
union all
```

```
select 'no records' from dual
```

```
where not exists (select s_name from sql1 where rownum=1)
```

```
and not exists (select s_name from sql2 where rownum=1);
```

再举个简单的例子

```
with a as (select * from test)
```

```
select * from a;
```

其实就是把一大堆重复用到的 SQL 语句放在 with as 里面，取一个别名，后面的查询就可以用它

这样对于大批量的 SQL 语句起到一个优化的作用，而且清楚明了

2、start with connect by prior 的用法

该语句用来实现递归查询

Prior 不能省略

Connect by prior a=b 不同于 connect by prior b=a 两者递归方向不同，一个向上，一个向下。

源码网整理,www.codepub.com