主讲老师：Fox

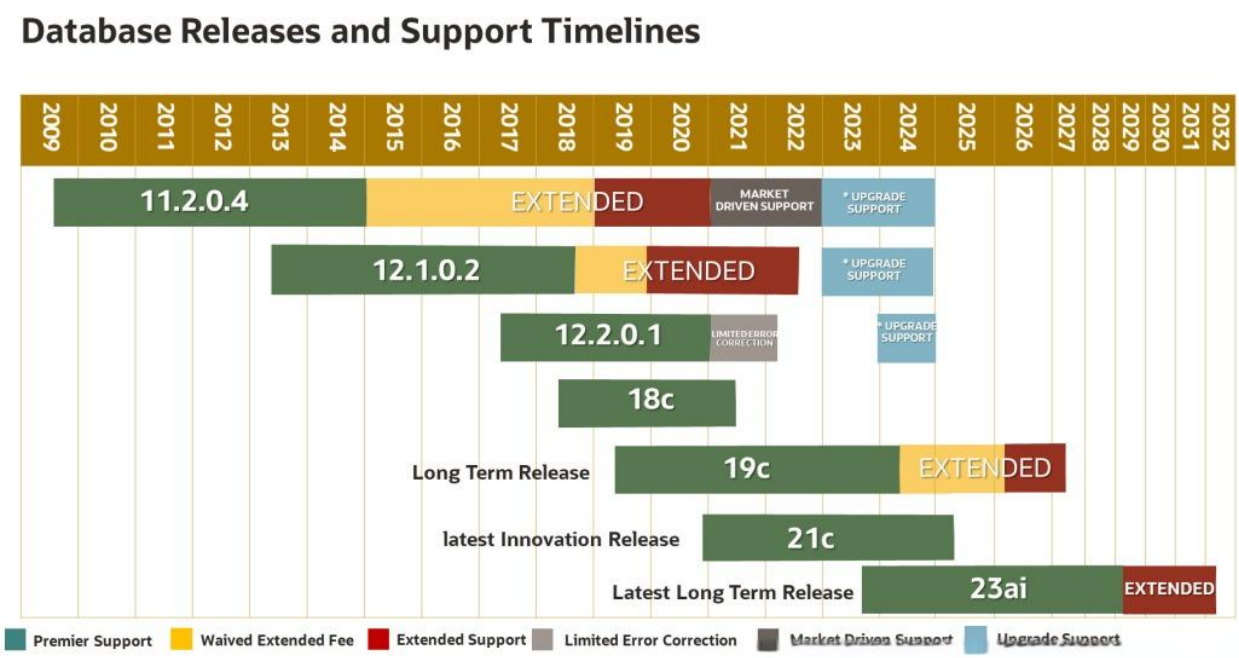
> 版本：Oracle Database 23ai Free
> 官网：https://www.oracle.com/cn/database/free/

1. Oracle介绍

1.1 Oracle简介

Oracle 是甲骨文公司开发的一款关系型数据库，它一款系统可移植性好、使用简单、功能强大的关系型数据库。它为各行业在各类环境下（服务器、虚拟机、微机环境下）可以快速搭建一种**高效率、可靠性好、高吞吐量**的数据库解决方案。

北京时间 2024年 5 月 3 日凌晨，Oracle Database 23c 更名为 Oracle Database 23ai。Oracle Database 23ai 是 Oracle Database 的下一个长期支持版本，它包括300 多项新功能，重点关注人工智能（AI）和开发人员的工作效率。



> 数据库 8i 9i 10g 11g 12c 18c 19c
> 数字代表版本号,12.2这个.2是小版本号
> i是internet的意思，表明当时是internet互联网盛行的年代。
> g是grid，网格运算。为了迎合分布式计算而推出的版本。
> c是cloud，云计算的意思。
> ai 重点是AI和提高开发人员的工作效率

Oracle23ai特性：

1.AI能力：内置向量数据库，模型数据处理，内置的机器学习算法持续增强

2.JSON能力增强
3.图数据处理能力增强
4.True Cache缓存服务
5.支持分片多副本
6.SQL易用性，向MySQL、PostgreSQL靠齐

1.2 应用场景

Oracle的应用场景非常广泛，涵盖了企业级应用、数据仓库和商业智能、电信行业、机构数据管理和决策支持系统、金融行业等多个领域。

- 企业级应用：Oracle适用于企业的核心业务系统，如财务管理、人力资源管理等，提供了高可用性、高性能的数据库服务，为企业提供灵活的数据库解决方案。
- 数据仓库和商业智能：Oracle可以用于构建大规模的数据仓库，支持数据的存储、查询和分析。这对于需要处理大量数据的组织来说尤为重要，可以帮助他们进行决策支持。
- 电信行业：Oracle可以用于电信运营商的计费系统、客户关系管理等，满足电信行业对数据安全性、可靠性和扩展性的高要求。
- 机构数据管理和决策支持系统：Oracle可以用于机构的数据管理和决策支持系统，帮助机构有效地管理和分析数据，支持其日常运营和决策过程。
- 金融行业：Oracle在金融行业的应用尤为突出，包括银行、证券等金融机构的系统。其提供的数据安全和稳定性对于金融行业的核心业务至关重要。
- 此外，Oracle还提供了云数据库服务Oracle Cloud，允许企业在云上部署Oracle Database，进一步扩展了其应用场景。通过使用Oracle Private Cloud Appliance，客户可以创建安全、高度可用的多租户私有云环境，快速部署高性能应用，同时密切控制成本。

综上所述，Oracle以其强大的功能、高可靠性、良好的扩展性以及对大规模数据的高效处理能力，在多个行业和领域中发挥着重要作用。

1.3 Oracle体系结构

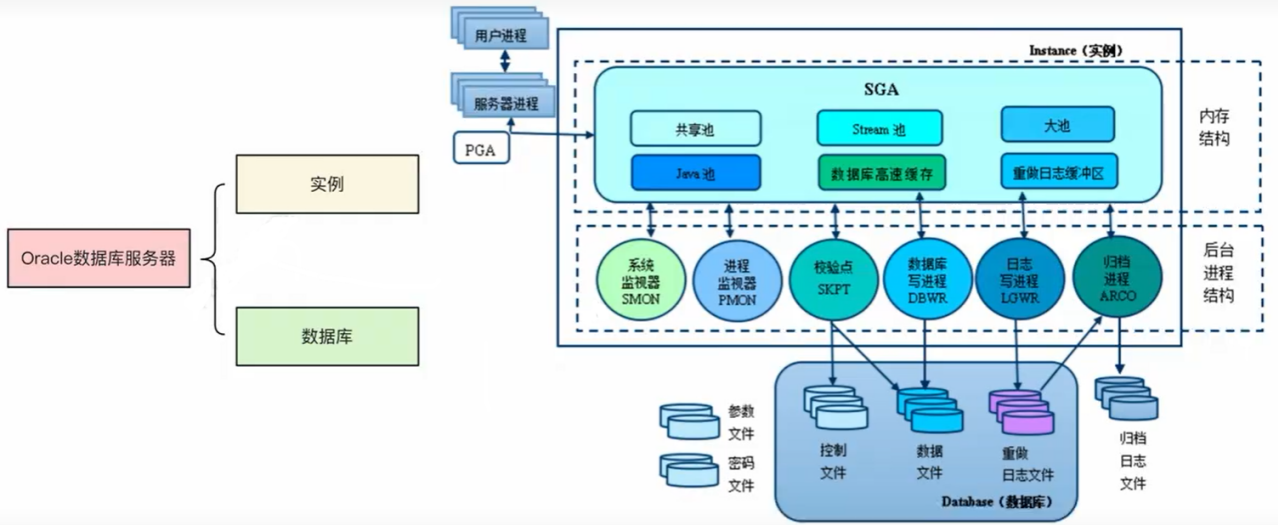
Oracle数据库实际上是一个数据的物理储存系统，这其中包括数据文件（ora/dbf）、参数文件、控制文件、联机日志等。

实例：Oracle实例是Oracle数据库系统的一种运行环境，它由一系列的后台进程（Background Processes）和内存结构（Memory Structures）组成。Oracle实例在运行时会占用一定的系统资源（如CPU, 内存, 和一些后台进程），而Oracle数据库是存储在实例中的一系列逻辑结构（如表，视图，索引等）

数据文件：Oracle数据文件是数据存储的物理单位，数据库的数据是存储在表空间中的。而一个表空间可以由一个或多个数据文件组成，一个数据文件只能属于一个表空间，一旦数据文件被加入到某个表空间后，就不能删除这个文件，如果要删除某个数据文件，只能删除其所属于的表空间才行。

表空间：表空间是Oracle 对物理数据库数据文件（ora/dbf）的逻辑映射。一个数据库在逻辑上被划分成一到若干个表空间，每个表空间由同一磁盘上的一个或多个数据文件（datafile）组成，一个数据文件只能属于一个表空间。

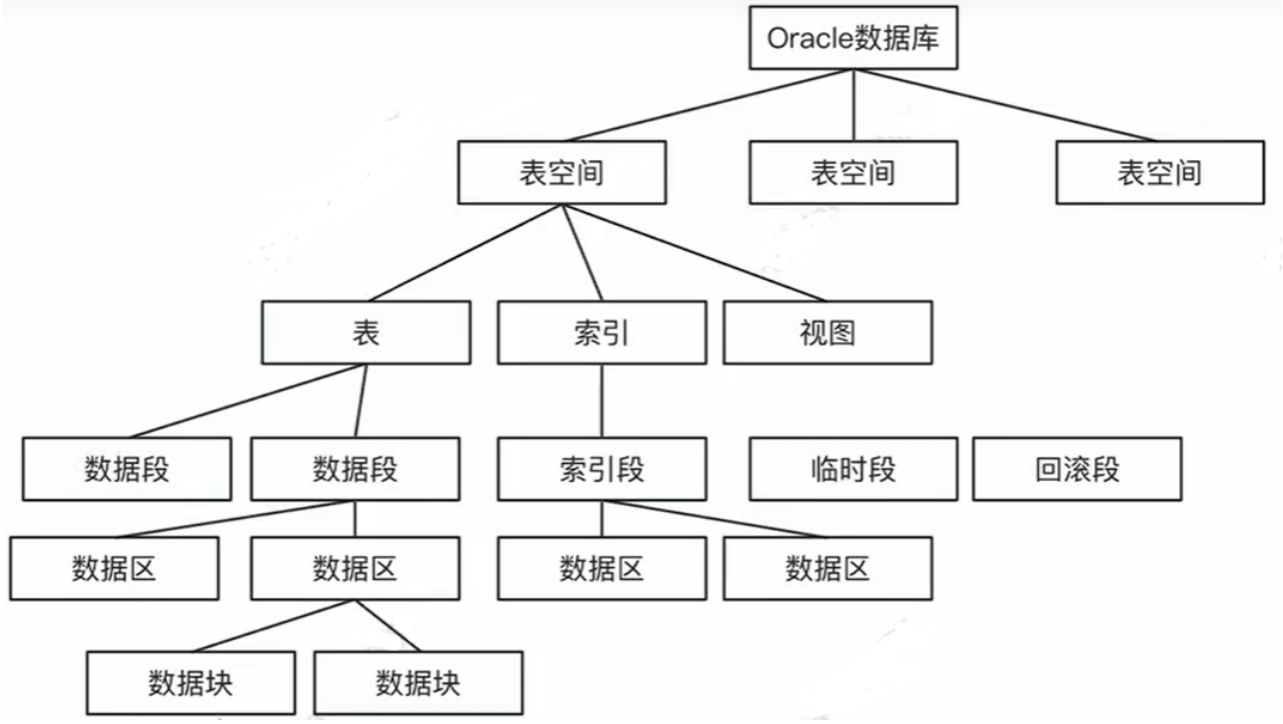
oracle用户：表当中的数据是由Oracle用户放入到表空间当中的，而这些表空间会随机的把数据放入到一个或者多个数据文件当中。Oracle对表数据的管理是通过用户对表的管理去查询，而不是直接对数据文件或表空间进行查询。因为不同用户可以在同一个表空间上面建立相同的表名。但是通过不同的用户管理自己的表数据。



逻辑存储结构

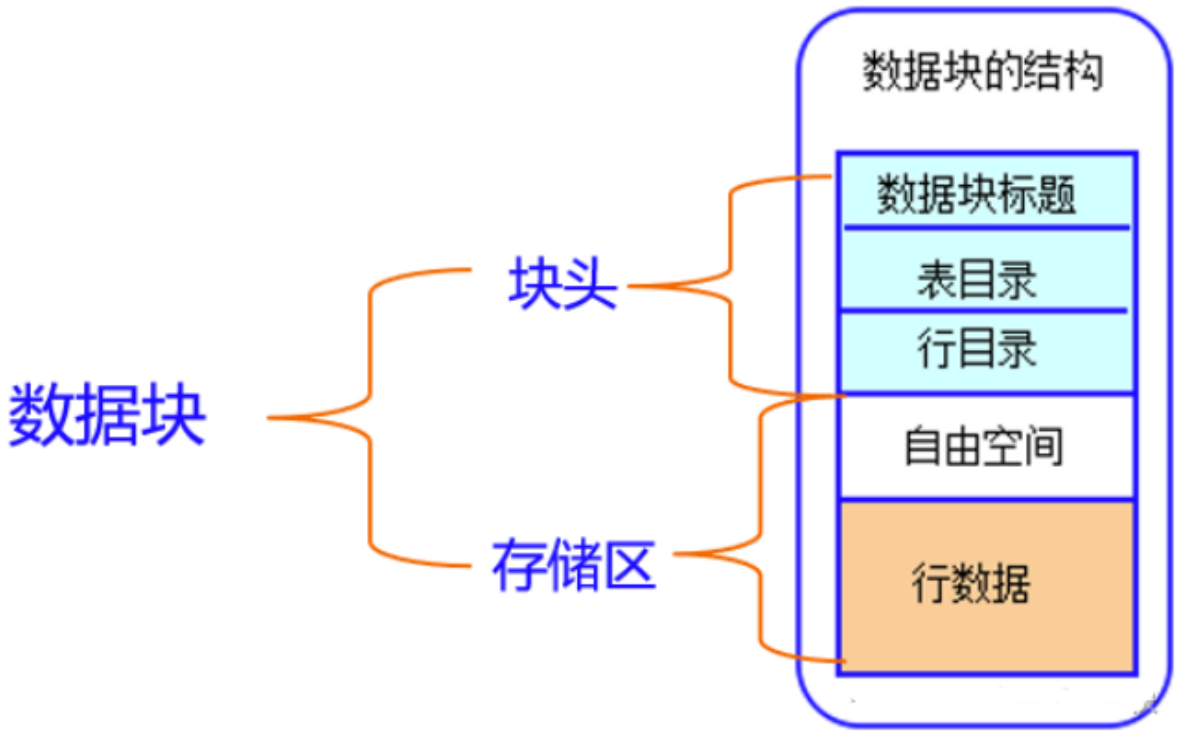
从逻辑上来看：

1. 数据库是由一个或者多个表空间等组成。
2. 一个表空间(tablespace)由一组段组成
3. 一个段(segment)由一组区组成
4. 一个区(extent)由一批数据库块组成
5. 一个数据库块(block)对应一个或多个物理块



数据块

- 数据块是Oracle逻辑存储结构中最小的逻辑单位
- 一个数据库块对应一个或者多个物理块，大小由参数DB_BLOCK_SIZE决定
- 当用户从表中选择数据时，选择操作从数据库文件中以块为单位读取或者提取数据。例如Oracle块的大小为8kb，即使只想检索4kb的字符的名字，也必须读取含有这4个字符的整个8kb的块。
- 数据块的结构包括块头和存储区两个部分



通过下面的语句查看当前数据块设置的大小：

```
1 show parameter db_block_size
```

数据区

是数据库存储空间分配的一个逻辑单位，它由连续数据块所组成。第一个段是由一个或多个盘区组成。当一段中间所有空间已完全使用，oracle为该段分配一个新的范围。

- 数据区是由连续的数据块结合而成的
- 数据区是Oracle存储分配的最小单位

段

一个段是分配给一个逻辑结构（一个表、一个索引或其他对象）的一组区，是数据库对象使用的空间的集合；段可以有表段、索引段、回滚段、临时段和高速缓存段等。

- 数据段：存储表中所有数据
- 索引段：存储表上最佳查询的所有索引数据
- 临时段：存储表排序操作期间建立的临时表的数据
- 回滚段：存储修改之前的位置和值

表空间

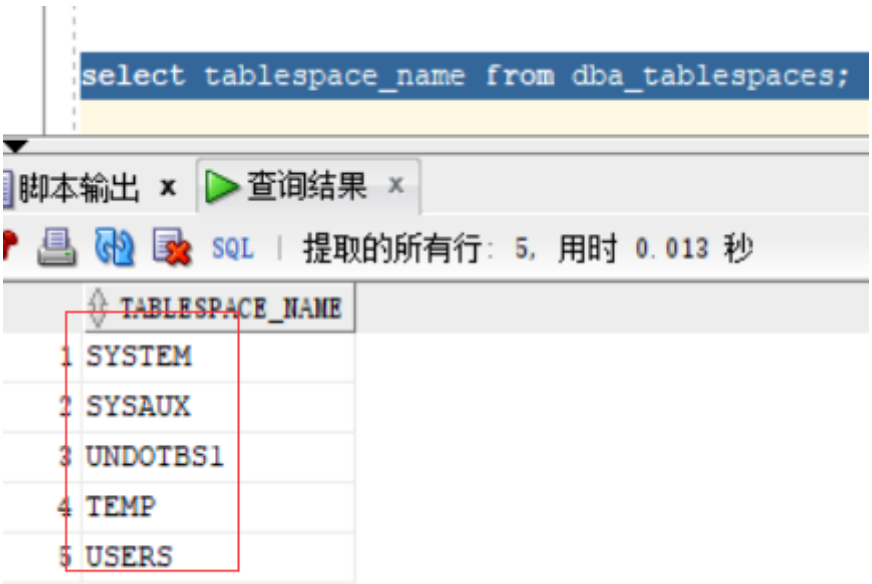
- 表空间是数据库的最大逻辑划分区域
- 一个表空间由一个或多个数据文件组成，一个数据文件只属于一个表空间
- 表空间的大小是它所对应的数据文件大小的总和
- 默认创建的表空间：系统表空间、辅助表空间、撤销表空间、用户表空间，这些表空间通常用于存储Oracle系统内部数据和提供样例所需要的逻辑空间

Oracle默认的表空间及其说明如下表所示：

表 空 间	说 明
EXAMPLE	如果安装时选择“实例方案”，则此表空间存储各样例的数据
SYSAUX	SYSTEM 表空间的辅助空间。一些选件的对象都存储在此表空间内，这样可以减少 SYSTEM 表空间的负荷
SYSTEM	存储数据字典，包括表、视图、存储过程的定义等
TEMP	存储 SQL 语句处理的表和索引的信息，如数据排序就占用此空间
UNDOTBS1	存储撤销数据的表空间
USERS	通常用于存储应用系统所使用的数据库对象

可以通过下面的语句查看当前数据库的表空间信息：

```
1 select tablespace_name from dba_tablespaces;
```



物理存储结构

物理结构包含的三种数据文件：

- 控制文件
- 数据文件
- 重做日志文件

控制文件（.CTL）

控制文件是oracle的物理文件之一，每个oracle数据库都必须至少有一个控制文件，它记录了数据库的名字、数据文件的位置等信息。在启动数据实例时，oracle会根据初始化参数定位控制文件，然后oracle会根据控制文件在实例和数据库之间建立关联。控制文件的重要性在于，一旦控制文件损坏，数据库将会无法启动。

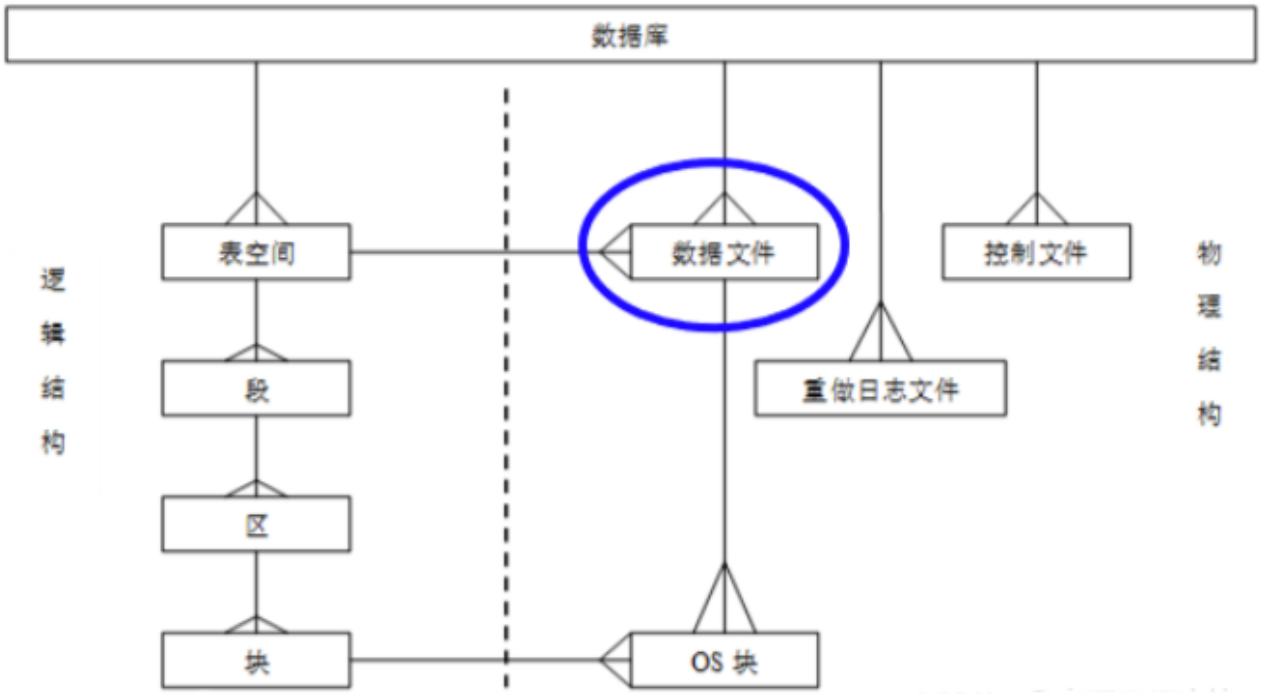
- 控制文件是数据库中最小的文件
- 控制文件是数据库中最重要的文件
- 控制文件在数据库创建时被自动创建，并在数据库发生物理变化时会同时更新。

数据文件（包括数据字典）（.DBF）

数据文件是用来存储实际数据的。一个数据库可以由多个数据文件组成的，数据文件是真正存放数据库数据的。一个数据文件就是一个操作系统文件。数据库的对象(表和索引)物理上是被存放在数据文件中的。当我们要查询一个表的数据的时候，如果该表的数据没有在内存中，那么oracle就要读取该表所在的数据文件，然后把数据存放到内存中。

数据文件和表空间的关系：

- 一个表空间可以包含几个数据文件
- 一个数据文件只能对应一个表空间



数据文件的种类：

- 系统数据文件（sysaux01.dbf和system01.dbf)
- 回滚数据文件（undotbs01.dbf)
- 用户数据文件（users01.dbf)
- 临时数据文件 （temp01.dbf)

通过下面的语句可以查看当前存在的数据文件和对应的表空间：

```
1 select file_name,tablespace_name from dba_data_files;
2 select file_name,tablespace_name from dba_temp_files;
```

```
select file_name,tablespace_name from dba_data_files;
```

查询结果 x

SQL | 提取的所有行：4，用时 0.004 秒

	FILE_NAME	TABLESPACE_NAME
1	/opt/oracle/oradata/FREE/system01.dbf	SYSTEM
2	/opt/oracle/oradata/FREE/sysaux01.dbf	SYSAUX
3	/opt/oracle/oradata/FREE/undotbs01.dbf	UNDOTBS1
4	/opt/oracle/oradata/FREE/users01.dbf	USERS

```
select file_name,tablespace_name from dba_temp_files;
```

询结果 x

SQL | 提取的所有行：1，用时 0.049 秒

	FILE_NAME	TABLESPACE_NAME
1	/opt/oracle/oradata/FREE/temp01.dbf	TEMP

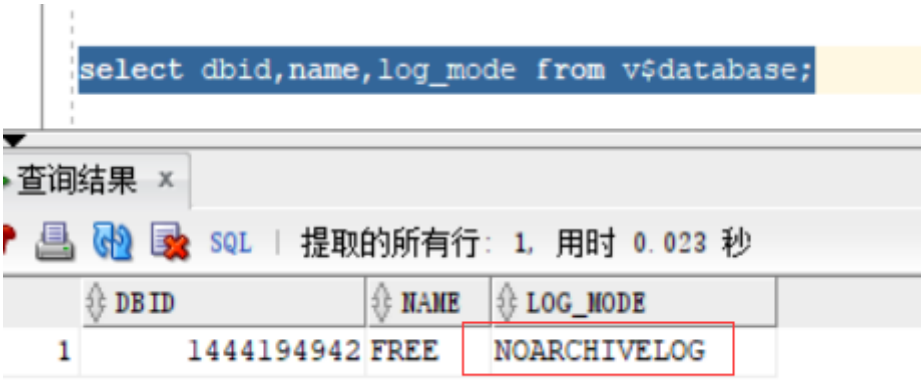
日志文件（.LOG）

在oracle数据库中，重做日志文件用于记录用户对数据库所做的各种变更操作所引起的数据变化，此时，所产生的操作会先写入重做日志缓冲区，当用户提交一个事务的时候，LGWR进程将与该事务相关的所有重做记录写入重做日志文件，同时生成一个“系统变更数”,scn会和重做记录一起保存到重做日志文件组，以标识与该事务提交成功。如果某个事务提交出现错误，可以通过重做记录找到数据库修改之前的内容，进行数据恢复。

- 重做日志文件特点：
  - a．**记录所有的数据变化**
  - b．提供恢复机制
- 归档日志文件：是重做日志文件的历史备份

默认情况下，oracle数据库处于非归档日志模式，即当重做日志文件写满的时候，直接覆盖里面的内容，原先的日志记录不会被写入到归档日志文件中。根据oracle数据库对应的应用系统不同，数据库管理员可以把数据库的日志模式在归档模式和非归档模式之间进行切换。


```
1  select dbid,name,log_mode from v$database;
```



可以通过alter database archivelog或noarchivelog语句实现数据库在归档模式与非归档模式之间进行切换。

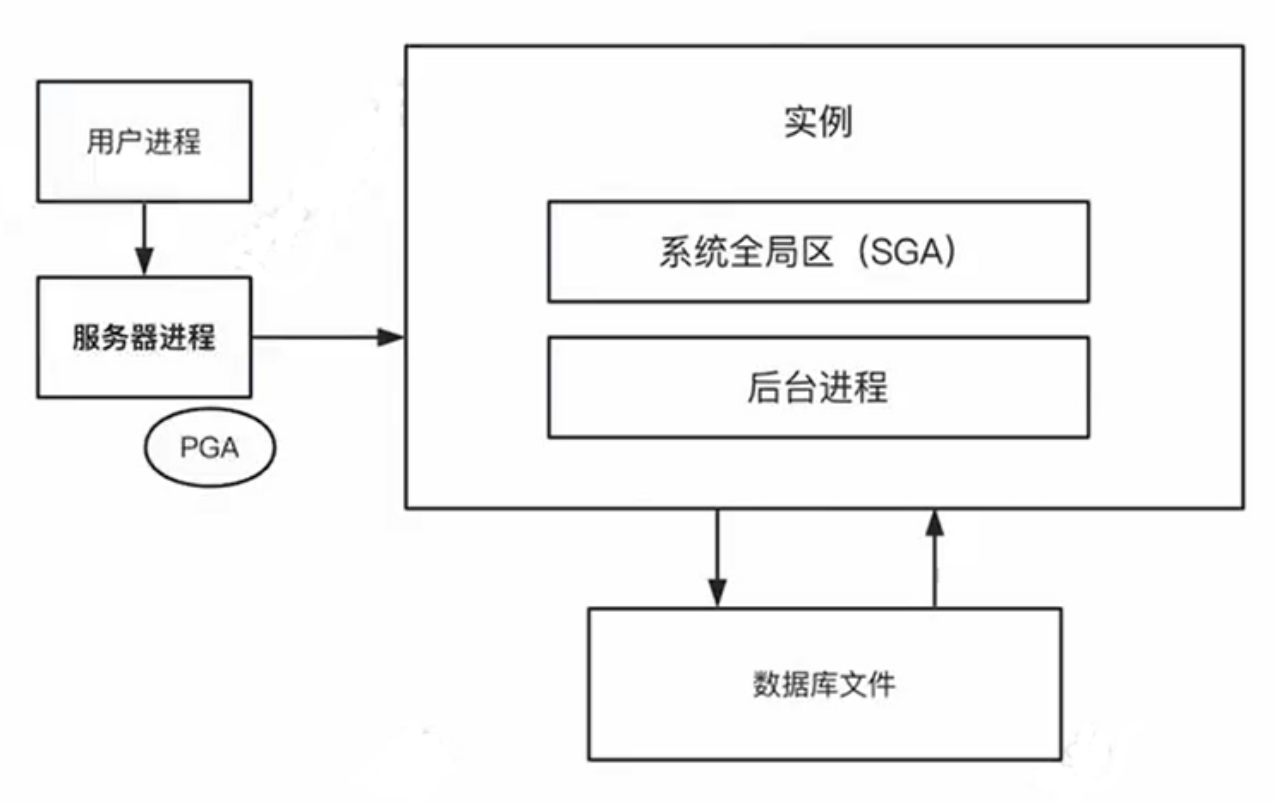
切换步骤如下：

```
1.关闭数据库
shutdown immediate;
2.将数据库启动到加载状态
startup mount;
3.改为归档模式
alter database archivelog;
4.重新打开数据库
alter database open;
```

服务器结构

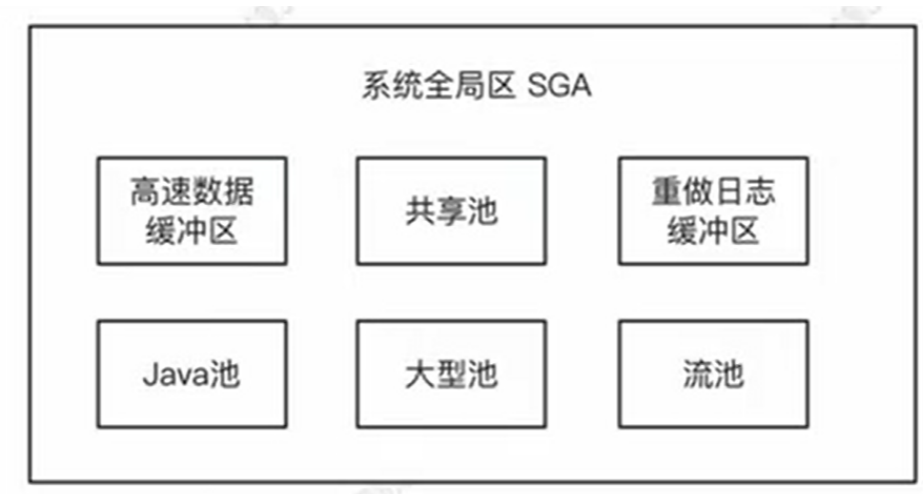
Oracle服务器主要有以下几部分组成：

- 前台进程
  - 用户进程: 使用SQL Plus连接成功后生成。包含两个重要概念:连接和会话
  - 服务器进程: 处理用户会话过程中的SQL语句和SQL Plus命令
- 程序全局区（PGA)
- 实例
  - 系统全局区（SGA)
  - 后台进程
- 数据库



系统全局区SGA

系统全局区（System Global Area，SGA）是Oracle为系统分配的一组共享的内存结构，可以包含一个数据库实例的数据或控制信息。 在一个数据库实例中，可以有多个用户进程，这些用户进程可以共享系统全局区的数据。SGA是Oracle Instance的基本组成部分，在实例启动时分配。



高速数据缓冲区

作用: 用来存放Oracle系统最近访问过的数据块。

当用户请求数据时，oracle会从高速缓存区中检索，如果检索到了对应的数据块即缓存命中，oracle便会直接从缓存区中读取数据。如果没有命中，oracle的读进程会从数据文件中读取对应的数据块，将对应的数据块加入到缓存区中。

经常或最近被访问的数据块会被放置到高速数据缓冲区的前端，不经常被访问的数据块会被放置到高速数据缓冲区的后端

共享池

SGA中的共享池（shared pool）是由库高速缓存（library cache）和数据字典高速缓存（data dictionary cache）两部分所组成。服务器进程将SQL或者PL/SQL语句的正文和编译后的代码以及执行计划都放在共享池的库高速缓存中，在进行编译时，服务器进程首先会在共享池中搜索是否有相同的SQL或者PL/SQL语句，如果有就不进行任何后续的编译处理，而是直接使用已存在的编译后的代码和执行计划。

作用: 存储最近执行过的SQL 语句和最近使用过的数据定义

共享池包含:

- 库高速缓冲区

库高速缓存包含了共享SQL区和共享PL/SQL区两部分，它们分别存放SQL和PL/SQL语句以及相关的信息。

SQL或PL/SQL语句要是能共享的通用代码

因为Oracle是通过比较SQL或PL/SQL语句的正文来决定两个语句是否相同的，只有当两个语句的正文完全相同时，Oracle才能重用已存的编译后的代码和执行计划。如果只是字母大小写不一样，那么字母大小写Oracle会自动转换，而且多余的空格或者制表键Oracle也会自动地压缩。

例如:

```
SELECT * FROM emp WHERE sal >= 10000;
SELECT * FROM emp WHERE sal >= 12000;
```

上面代码因为SQL语句的不同的，所以不是能共享的，那么我们可以使用通用代码来达到共享；我们可以通过绑定变量的方式:

```
SELECT * FROM emp WHERE sal >= :g_sal;
```

因为变量不是在编译阶段而是在运行阶段赋值的。

- 字典高速缓冲区

当Oracle在执行SQL语句时，服务器进程将把数据文件、表、索引、列、用户和其他的数据对象的定义和权限的信息放入数据字典高速缓存。如果在这之后，有用户进程需要同样的信息，如表和列的定义，那么所有的这些信息都将从数据字典高速缓存中获得。

```
#设置共享池的大小，受限于SGA_MAX_SIZE
ALTER SYSTEM SET SHARED_POOL_SIZE = 250M
```

程序全局区PGA

程序全局区（Program Global Area, PGA），是服务器进程(server process)使用的一块包含数据和控制信息的内存区域。PGA是非共享的内存,在服务器进程启动或创建时分配(在系统运行时,排序,连接等操作也可能需要进行更进一步的PGA分配),并为server process排他访问。PGA中包含了会话的特定信息，如游标状态、排序和哈希操作的内存，以及用于管理游标共享和会话内内存分配的数据结构。

每个Oracle服务器进程只拥有自己的那部分PGA资源。

2.安装Oracle Database 23ai Free

> 文档：https://www.oracle.com/database/free/get-started/#quick-start

2.1 安装的环境要求

Oracle Database Free 系统要求

要求	价值
操作系统	请参阅《适用于 Linux 的 Oracle 数据库安装指南》，了解每个 x86-64 Linux 平台支持的 Linux 发行版列表和最低操作系统要求。
网络协议	支持以下协议： • IPC • UDP • TCP/IP • TCP/IP with SSL
内存	至少 1 GB RAM。建议使用 2 GB RAM。
磁盘空间	至少 10 GB。

服务器组件内核参数要求

Oracle 数据库预安装 RPM 会检查系统的内核参数设置。如果系统的内核参数值小于此表中列出的值，则 Oracle Database 预安装 RPM 会为您设置建议的最小内核参数值。在/etc/sysctl.d/97-oracle-database-sysctl.conf中设置的值在系统重新启动时将保留。

> 参考：手动更改内核参数值

2.2 rpm安装

1）准备Oracle Linux9系统环境

> 参考：安装Oracle Linux9 操作系统

2) 下载oracle-database-preinstall-23ai-1.0-2.el9.x86_64.rpm和oracle-database-free-23ai-1.0-1.el9.x86_64.rpm数据库安装包

3）安装 Oracle 数据库预安装 RPM

```
dnf -y install oracle-database-preinstall-23ai-1.0-2.el9.x86_64.rpm
```

> 文档：DNF命令参考

4) 安装数据库软件

```
dnf -y install oracle-database-free-23ai-1.0-1.el9.x86_64.rpm
```

5) 创建和配置 Oracle 数据库
数据库使用默认设置进行配置。除非您有特殊要求，否则无需修改这些参数。在修改配置文件 /etc/sysconfig/oracle-free-23ai.conf 之前，先复制该文件。在 RPM 安装之后和配置数据库之前进行修改。
/etc/sysconfig/oracle-free-23ai.conf 配置文件设置以下内容：

- LISTENER_PORT：数据库侦听器的有效侦听器数字端口值。不要为自动端口分配指定任何值。
- CHARSET：数据库的字符集。这设置为 AL32UTF8。
- DBFILE_DEST 数据库文件目录。默认情况下，数据库文件存储在 Oracle base /opt/oracle/oradata 子目录中。您还可以创建自己的数据库文件目录。但是，该文件路径的权限应该由oracle用户拥有。
- SKIP_VALIDATIONS：跳过内存和磁盘空间的验证。默认为 false。
- CONFIGURE_TDE：设置 CONFIGURE_TDE=true 以配置 TDE。默认值为 false。
- ENCRYPT_TABLESPACES：将此值保留为空，仅用于用户表空间。将此值设置为 ALL 以加密所有表空间。

```
/etc/init.d/oracle-free-23ai configure
```

在命令提示符下，为 SYS、 SYSTEM 和 PDBADMIN 管理用户帐户指定密码。Oracle 建议您的密码长度应至少为8个字符，包含至少1个大写字符、1个小写字符和1个数字[0-9]。

配置、数据库文件和日志位置

文件名和位置	作用
/opt/oracle	Oracle 基础。这是 Oracle Database Free 目录树的根目录。
/opt/oracle/product/23ai/dbhomeFree	Oracle 主页。此主页是安装 Oracle Database Free 的位置。它包含 Oracle Database Free 可执行文件和网络文件的目录。
/opt/oracle/oradata/FREE	数据库文件。
/opt/oracle/diag子目录	诊断日志。数据库警报日志是/opt/oracle/diag/rdbms/free/FREE/trace/alert_FREE.log
/opt/oracle/cfgtoollogs/dbca/FREE	数据库创建日志。该文件包含数据库创建脚本执行的结果。FREE.log
/etc/sysconfig/oracle-free-23ai.conf	配置默认参数。
/etc/init.d/oracle-free-23ai	配置和服务脚本。

6）设置 Oracle Database Free 环境变量

以 Oracle 用户身份登录并运行以下命令：

```
su oracle
export ORACLE_SID=FREE
export ORAENV_ASK=NO
. /opt/oracle/product/23ai/dbhomeFree/bin/oraenv
```

全局配置环境变量

```
vim /etc/profile
# 添加下面的配置
export ORACLE_HOME=/opt/oracle/product/23ai/dbhomeFree
export ORACLE_SID=FREE
export PDB_NAME=FREEPDB1
export NLS_LANG=AMERICAN_AMERICA.AL32UTF8
export NLS_DATE_FORMAT="YYYY-MM-DD HH24:MI:SS"
export LD_LIBRARY_PATH=${ORACLE_HOME}/lib:/lib:/usr/lib
export PATH=${ORACLE_HOME}/bin:${ORACLE_HOME}/OPatch:$OGG_HOME:${PATH}
export HOST=`hostname | cut -f1 -d"."`
export LD_LIBRARY_PATH=$ORACLE_HOME/lib:$ORACLE_HOME/lib32:$OGG_HOME:/lib/usr/lib:/usr/local/lib
# General exports and vars
export PATH=$ORACLE_HOME/bin:$PATH
LSNR=$ORACLE_HOME/bin/lsnrctl
SQLPLUS=$ORACLE_HOME/bin/sqlplus

#使配置生效
source /etc/profile
```

7) 连接到数据库

```
# 切换到oracle用户
su oracle
# 以DBA身份登录oracle
sqlplus / as sysdba
> select instance_name from v$instance;

#关闭防火墙
systemctl stop firewalld
systemctl disable firewalld
#远程连接 ， 需要先关闭防火墙
sqlplus system/Oracle23ai@192.168.65.198:1521
# 连接到默认 PDB：FREEPDB1
sqlplus system/Oracle23ai@192.168.65.198:1521/FREEPDB1
```

```
[oracle@192-168-65-198 root]$ sqlplus / as sysdba

SQL*Plus: Release 23.0.0.0.0 - Production on Thu Jun 20 10:55:31 2024
Version 23.4.0.24.05

Copyright (c) 1982, 2024, Oracle.  All rights reserved.


???:
Oracle Database 23ai Free Release 23.0.0.0.0 - Develop, Learn, and Run for Free
Version 23.4.0.24.05

SQL> select instance_name from v$instance;

INSTANCE_NAME
----------------
FREE
```

Oracle Database Free 的 Net Services 数据库监听器允许您通过 TCP/IP 从同一台计算机或网络上的其他计算机连接到数据库。可以使用从命令提示符运行的以下命令查看监听器的配置：

```
lsnrctl status
```

```
Connecting to (DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=192-168-65-198)(PORT=1521)))
STATUS of the LISTENER
------------------------
Alias                     LISTENER
Version                   TNSLSNR for Linux: Version 23.0.0.0.0 - Production
Start Date                20-JUN-2024 09:40:16
Uptime                    0 days 2 hr. 13 min. 27 sec
Trace Level               off
Security                  ON: Local OS Authentication
SNMP                      OFF
Default Service           FREE
Listener Parameter File   /opt/oracle/product/23ai/dbhomeFree/network/admin/listener.ora
Listener Log File         /opt/oracle/diag/tnslsnr/192-168-65-198/listener/alert/log.xml
Listening Endpoints Summary...
  (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=192-168-65-198)(PORT=1521)))
  (DESCRIPTION=(ADDRESS=(PROTOCOL=ipc)(KEY=EXTPROC1521)))
Services Summary...
Service "1b49bcdddf63bcd3e065f8dcbd290b00" has 1 instance(s).
  Instance "FREE", status READY, has 1 handler(s) for this service...
Service "FREE" has 1 instance(s).
  Instance "FREE", status READY, has 1 handler(s) for this service...
Service "FREEXDB" has 1 instance(s).
```

lsnrctl 命令的输出显示许多重要参数的值:

- 监听器监听的端口
- 向监听器注册的服务列表
- 监听器使用的配置文件的名称
- 日志文件的名称

8）开机自启动设置

Oracle 建议您将系统配置为在系统启动时自动启动 Oracle Database Free，并在系统关闭时自动关闭系统。自动关闭数据库可防止不正确的数据库关闭。
要自动启动和关闭监听器和数据库，请按以下方式运行以下命令：

```

systemctl daemon-reload
/usr/lib/systemd/systemd-sysv-install enable oracle-free-23ai
```

使用 Configuration Services 脚本关闭和启动

```
/etc/init.d/oracle-free-23ai status

systemctl start oracle-free-23ai
systemctl stop  oracle-free-23ai
systemctl restart oracle-free-23ai
```

2.3 Docker镜像安装

1）安装docker环境

2）拉取oracle镜像

```
docker pull container-registry.oracle.com/database/free:latest
```

3）运行镜像

```
docker run -d --name oracl23ai -p 1521:1521 container-
registry.oracle.com/database/free:latest
#以超级管理员身份登录并查看实例
docker exec -it oracl23ai sqlplus / as sysdba
SQL> select instance_name from v$instance;
```

2.4 导入VM VirtualBox虚拟机安装

Oracle 官方提供了一个已经在虚拟机上安装好了的 Oracle 23 ai,并将其打包成 ova 文件，可直接在官方下载，然后安装 VM VitualBox。完了将下载好的 ova 文件导入到虚拟机中。

1）安装VirtualBox

> 下载地址：Downloads – Oracle VM VirtualBox

2）下载Oracle_Database_23ai_Free_Developer.ova，并导入VirtualBox虚拟机

2.5 安装Oracle SQL Developer

Oracle SQL Developer 是同类优秀的 Oracle Database 管理工具，它为用户提供了三种界面：桌面端、Web 浏览器和命令行。

> https://www.oracle.com/cn/database/sqldeveloper/

下载完成后，解压缩并运行 sqldeveloper.exe，即可启动 SQL Developer。

建立连接

3.Oracle使用

3.1 用户管理

Oracle用户的概念对于Oracle数据库至关重要，在现实环境当中一个服务器一般只会安装一个Oracle实例，一个Oracle用户代表着一个用户群，他们通过该用户登录数据库，进行数据库对象的创建、查询等开发。

每一个用户对应着该用户下的N多对象，因此，在实际项目开发过程中，不同的项目组使用不同的Oracle用户进行开发，不相互干扰。也可以理解为一个Oracle用户即是一个业务模块，这些用户群构成一个完整的业务系统，不同模块间的关联可以通过Oracle用户的权限来控制，来获取其它业务模块的数据和操作其它业务模块的某些对象。

用户模式

在Oracle数据库中，为了便于管理用户创建的数据库对象(如数据表、索引、视图等)，引入了模式的概念，某个用户创建的数据库对象都属于该用户模式。

在一个模式内部不可以直接访问其他模式的数据库对象，即使在具有访问权限的情况下，也需要指定模式名称才可以访问其他模式的数据库对象。

比如在scott账号下创建的表dept，查询这个表时使用下面的语句：

```
SELECT * FROM dept;
```

当在其他用户账号下，比如system账号下，访问dept表时，需要在表名前面加上该表的所属账号scott；

```
SELECT * FROM scott.dept;
```

用户登录数据库

SQLPlus是 Oracle 默认安装的自带的一个客户端工具。如果Oracle安装在 Windows下，在cmd 命令行中输入“sqlplus”命令，就能够启动该工具了；如果Oracle安装在Linux系统中，在Linux的命令界面中输入“sqlplus”命令，就能启动该工具。

连接身份代表着改用户连接数据库后享受的权限，oracle 有三种身份如下：

- **sysdba:**数据库管理员身份。权限：打开（关闭）数据库服务器、备份（恢复）数据库、日志功能、会话限制、数据库管理功能等。
- **sysoper:**数据库操作员身份 。权限：打开（关闭）数据库服务器、备份（恢复）数据库、日志功能、会话限制。
- **normal:**普通用户。权限：操作该用户下的数据对象和数据查询，默认的身份是normal用户。

注意:sys用户必须用sysdba才能登陆(使用：sys as sysdba)，system用户(数据库管理员身份，直接使用：system)。

sqlplus登录用户语法：sqlplus username/password@serviceName [as 连接身份]

以系统管理员登录数据库：

```
sqlplus system/Oracle23ai@FREE
```

切换用户：connect username/password@serviceName [as 连接身份]

```
connect sys/Oracle23ai@free as sysdba;
```

- 查看当前用户：show user;

SQLPlus令行工具提供了和Oracle数据库交互能力，不仅仅可以连接本地数据库，也可以连接远程数据库。

语法：conn 用户名/密码@服务器连接字符串 as 连接身份(此身份需要解锁)

案例：connect sys/Oracle23ai@192.168.65.198:1521/free as sysdba;

基本概念

- CDB——Container Database，即容器数据库；

- PDB——Pluggable Database，即可插拔数据库；
- 公共用户——CDB公共用户，必须以C##或c##开头，Oracle会在每个PDB中同时创建该用户；
- 本地用户——PDB的用户，本地用户所在PDB中必须是唯一的，每个PDB可以有不同的本地用户；

CDB和PDB是数据库中的两种重要概念。
CDB是一个容器数据库，名称为 CDB$ROOT，其作用就是系统数据库，sys等账号都保存在里面，同时它可以管理PDB数据库。可以容纳多个PDB，类似于一个父容器。在CDB环境中，用户称为公共用户，而在PDB环境中，用户称为本地用户。
PDB是一种可插拔数据库，它可以在CDB中独立地创建、停止和启动。每个PDB都有自己的数据文件、控制文件和日志文件，并且可以独立地进行备份和恢复操作。PDB提供了更高的隔离性和安全性，因为每个PDB都有自己的安全上下文，并且只能通过指定的用户访问。

常用命令

```
--打开PDB
alter pluggable database pdb名称 open;
--关闭PDB
alter pluggable database pdb名称 close;
--切换PDB
alter session set container=pdb名称(指定容器);
--查看当前所在容器
show con_name;

--查看所有的PDB
show pdbs;

--切换到mypdb
alter session set container=mypdb;
```

创建用户

- 每个用户都有自己的用户名和密码，以及权限和角色，用户访问和管理数据库中的对象。
- 使用CREATE和DROP来创建和删除用户。用户对它所拥有的对象具有所有的增删改的权限。
- 可以使用 ALTER USER 命令来修改用户的属性，如密码、默认表空间以及存储配额。
- 用户可以被授予不同的权限或者角色。

```
sqlplus system/Oracle23ai@free  -- 连接数据库(管理员角色)


# 在CDB（Container Database）容器中创建用户时，前面必须添加C## ,而PDB（Pluggable Database）数据库不需要加前缀
#COMMON USERS(普通用户)：经常建立在CDB层，用户名以C##或c##开头；
#LOCAL USERS(本地用户)：仅建立在PDB层，建立的时候得指定CONTAINER。

#看数据库是否为 CDB
select CDB from v$database;

#创建用户
create user C##fox identified by Oracle23ai;

#配置权限
grant dba,connect,resource,create view to C##fox;
grant create session to C##fox;
grant select any table to C##fox;
grant update any table to C##fox;
grant insert any table to C##fox;
grant delete any table to C##fox;



#连接登陆测试
conn c##fox/Oracle23ai@free


--修改用户密码、锁定状态
alter user c##fox
  identified by ******  --修改密码
  account lock;--修改用户处于锁定状态或者解锁状态 （LOCK|UNLOCK ）

#删除用户
drop user c##fox;
```

虽然创建了用户，但还不能使用，需要给用户授予数据库角色和设置用户权限。

数据库角色

Oracle数据库角色是若干系统权限的集合，给Oracle用户进行授数据库角色，就是等于赋予该用户若干数据库系统权限。

常用的数据库角色如下：

- CONNECT角色：

CONNECT角色是Oracle用户的基本角色，CONNECT权限代表着用户可以和Oracle服务器进行连接，建立Session（会 话）。

- RESOURCE角色：

RESOURCE角色是开发过程中常用的角色。RESOURCE给用户提供了可以创建自己的对象，包括：表、视图、序列、过程、触发器、索引、包、类型等。

- DBA角色：

DBA角色是管理数据库管理员该有的角色。它拥护系统了所有权限，和给其他用户授权的权限。SYSTEM用户就具有DBA权限。

三个数据库角色，对应有三个连接身份。

用户权限

Oracle数据库用户权限分为：系统权限和对象权限两种。

- 系统权限：

比如：create session可以和数据库进行连接权限、create table、create view 等具有创建数据库对象权限。

- 对象权限：

比如：对表中数据进行增删改查操作，拥有数据库对象权限的用户可以对所拥有的对象进行相应的操作。

因此，在实际开发过程当中可以根据需求，把某个角色或系统权限赋予某个用户。

- 系统权限只能通过DBA用户授权！所以通常用于连接的用户（sqlplus system/Oracle23ai as sysdba; ）
- 对象权限由拥有该对象权限的对象授权！
- 数据库角色选择授权一般也只能通过DBA用户授权！（sqlplus system/Oracle23ai as sysdba;）
- 用户不能自己给自己授权！（所以授权可以统一的：sqlplus system/Oracle23ai as sysdba;）

授权操作

```
--GRANT 数据库角色 to 用户
grant connect to C##fox;--授权connect角色（必须）
grant resource to C##fox;--授予resource角色
grant dba to C##fox; -- 授予管理员dba角色

--GRANT 用户的系统权限 to 用户
grant create session to C##fox; -- 授予用户登录数据库的权限
-- 授予用户操作表空间的权限：
grant unlimited tablespace to C##fox;  -- 授予用户无限制的操作表空间的权限
grant create tablespace to C##fox;
grant alter tablespace to C##fox;
grant drop tablespace to C##fox;
grant manage tablespace to C##fox;
-- 授予用户操作表的权限：
grant create table to C##fox; (包含有create index权限, alter table, drop table权限)
-- 授予用户操作视图的权限：
grant create view to C##fox; (包含有alter view, drop view权限)
-- 授予用户操作触发器的权限：
grant create trigger to C##fox; (包含有alter trigger, drop trigger权限)
 -- 授予用户操作存储过程的权限：
grant create procedure to C##fox;(包含有alter procedure, drop procedure 和function 以及package权限)
-- 授予用户操作序列的权限：
grant create sequence to C##fox; (包含有创建、修改、删除以及选择序列)
-- 授予用户回退段权限：
grant create rollback segment to C##fox;
grant alter rollback segment to C##fox;
grant drop rollback segment to C##fox;
-- 授予用户同义词权限：
grant create synonym to C##fox;(包含drop synonym权限)
grant create public synonym to C##fox;
grant drop public synonym to C##fox;
-- 授予用户关于用户的权限：
grant create user to C##fox;
grant alter user to C##fox;
grant become user to C##fox;
grant drop user to C##fox;
-- 授予用户关于角色的权限：
```

```
grant create role to C##fox;
-- 授予用户操作概要文件的权限
grant create profile to C##fox;
grant alter profile to C##fox;
grant drop profile to C##fox;
-- 允许从sys用户所拥有的数据字典表中进行选择
grant select any dictionary to C##fox;


--GRANT 用户的对象权限 on 对象 TO 用户
grant select, insert, update, delete on emp to C##fox;
--把emp表的全部操作权限授予fox用户
grant all on emp to C##fox;
```

查看用户的权限或角色

```
（一）、查看用户
show user; //查看当前用户名
1.查看所有用户：
select * from dba_users;
select * from all_users;
select * from user_users;    //查看当前用户

（二）、查看角色
1.当前用户被激活的全部角色
select * from session_roles;
2.当前当前用户被授予的角色
select * from user_role_privs;
3.全部用户被授予的角色
select * from dba_role_privs;
4.查看某个用户所拥有的角色
select * from dba_role_privs where grantee='用户名';
5.一个角色包含的系统权限
select * from dba_sys_privs where grantee='角色名'
select * from dba_sya_privs where grantee='CONNECT'; connect要大写
或者
select * from role_sys_privs where role='角色名'
select * from role_sys_privs where grantee='CONNECT'; connect要大写
6.一个角色包含的对象权限
select * from dba_tab_privs where grantee='角色名'
7.查看所有角色
select * from dba_roles;

（三）、查看权限
1.基本权限查询：
select * from session_privs; --当前用户所拥有的全部权限
select * from user_sys_privs;--当前用户的系统权限
select * from user_tab_privs;--当前用户的对象权限
select * from dba_sys_privs ;--查询某个用户所拥有的系统权限
select * from role_sys_privs;--查看角色(只能查看登陆用户拥有的角色)所包含的权限
2. 查看用户的系统权限(直接赋值给用户或角色的系统权限)
select * from dba_sys_privs;
select * from user_sys_privs;
3.查看用户的对象权限：
```

```
select * from dba_tab_privs;
select * from all_tab_privs;
select * from user_tab_privs;
4.查看哪些用户有sysdba或sysoper系统权限(查询时需要相应权限)
select * from v$pwfile_users;

补充
1、以下语句可以查看Oracle提供的系统权限
select name from sys.system_privilege_map
2、查看一个用户的所有系统权限(包含角色的系统权限)
select privilege from dba_sys_privs where grantee='SYSTEM'
union
select privilege from dba_sys_privs where grantee in (select granted_role from
dba_role_privs where grantee='SYSTEM' );
3、 查询当前用户可以访问的所有数据字典视图。
select * from dict where comments like '%grant%';
4、显示当前数据库的全称
select * from global_name;
```

取消用户权限

```
-- Revoke 对象权限 on 对象 from 用户
revoke select, insert, update, delete on emp from C##fox;

-- Revoke 系统权限  from 用户
grant create session to C##fox; -- 授予用户登录数据库的权限
revoke SELECT ANY TABLE from C##fox;

-- Revoke 角色（role） from 用户
revoke RESOURCE from C##fox;
```

3.2 SQL基本语法

SQL语句介绍

在 Oracle 开发中，客户端把 SQL 语句发送给服务器，服务器对 SQL 语句进行编译、执行，把执行的结果返回给客户端。常用的SQL语句大致可以分为五类：

1. 数据定义语言（DDL），包括 CREATE（创建）命令、 ALTER（修改）命令、 DROP（删除）命令等。
2. 数据操纵语言（DML），包括 INSERT（插入）命令、 UPDATE（更新）命令、 DELETE（删除）命令等。
3. 数据查询语言（DQL），包括基本查询语句、 Order By 子句、 Group By 子句等。
4. 事务控制语言（TCL），包括 COMMIT（提交）命令、 SAVEPOINT（保存点）命令、ROLLBACK（回滚）命令。
5. 数据控制语言（DCL）， GRANT（授权）命令、 REVOKE（撤销）命令。

SQL语言的编写规则

- SQL关键字不区分大小写，既可以使用小写格式，也可以使用大写格式，或者大小写格式混用。
- 对象名和列名不区分大小写，既可以使用大写格式，也可以使用小写格式，或者大小写格式混用。
- 字符值区分大小写。当在SQL语句中引用字符值时，必须给出正确的大小写数据，否则不能得到正确的查询结果。

Oracle字段数据类型

数据类型	类型解释
VARCHAR2(length)	字符串类型：存储可变的长度的字符串，length:是字符串的最大长度，默认不填的时候是1，最大长度不超过4000。
CHAR(length)	字符串类型：存储固定长度的字符串，length:字符串的固定长度大小，默认是1，最大长度不超过2000。
NUMBER(a,b)	数值类型：存储数值类型，可以存整数，也可以存浮点型。a代表数值的最大位数：包含小数位和小数点，b代表小数的位数。例子：
	number(6,2)，输入321.456789，实际存入：321.45 。
	number(4,2)，输入12312.345，实际存入：错误，超过存储的指定的精度。
DATA	时间类型：存储的是日期和时间，包括年、月、日、时、分、秒。例子：
	内置函数sysdate获取的就是DATA类型
TIMESTAMP	时间戳类型：存储的不仅是日期和时间，还包含了时区。例子：
	内置函数systimestamp获取的就是timestamp类型
CLOB	大字段类型：存储的是大的文本，比如：非结构化的txt文本，字段大于4000长度的字符串。
BLOB	二进制类型：存储的是二进制对象，比如图片、视频、声音等转换过来的二进制对象

Oracle算术运算符

+、-、*、/、mod()

```
select 5 + 3 from dual;
select 5 - 3 from dual;
select 5 * 3 from dual;
select 5 / 3 from dual;
select mod(5,3) from dual;
```

Oracle关系运算符

符号	解释	符号	解释
=	等于	<>或者!=	不等于
>	大于	>=	大于或者等于
<	小于	<=	小于或者等于

Oracle的逻辑运算符

AND、OR、NOT

字符串连接符||

```
select 'w' || 123 || 'abc' from dual;
```

创建表

```
1
2  -- Create table
3  create table stuinfo     --用户名.表名
4  (
5    stuid      varchar2(11) not null,--学号: ①
6    stuname    varchar2(50) not null,--学生姓名
7    sex        char(1) not null,--性别
8    age        number(2) not null,--年龄
9    classno    varchar2(7) not null,--班号:
10   stuaddress varchar2(100) default '地址未录入',--地址 ②
11   grade      char(4) not null,--年级
12   enroldate  date,--入学时间
13   idnumber   varchar2(18) default '身份证未采集' not null--身份证
14 )
15 tablespace USERS --③
16   storage
17   (
18     initial 64K
19     minextents 1
20     maxextents unlimited
21   );
22 -- Add comments to the table
23 comment on table stuinfo --④
24   is '学生信息表';
25 -- Add comments to the columns
26 comment on column stuinfo.stuid -- ⑤
27   is '学号';
28 comment on column stuinfo.stuname
29   is '学生姓名';
30
31
32
33
34
35
36
```

代码说明:

①处：not null 表示学号字段（stuid）不能为空。

②处：default 表示字段stuaddress不填时候会默认填入‘地址未录入’值。

③处：表示表stuinfo存储的表空间是users，storage表示存储参数：区段(extent)一次扩展64k，最小区段数为1，最大的区段数不限制。

④处：comment on table 用户名.表名 is "注释内容"；是给表名进行注释。

⑤处：comment on column 用户名.表名.字段名 is "注释内容"；是给表字段进行注释

添加约束

```
-- 非空（NOT NULL）约束
ALTER TABLE teacher MODIFY subject NOT NULL;
-- 主键(PRIMARY KEY)约束
ALTER TABLE stuinfo ADD CONSTRAINT pk_stuinfo_stuid PRIMARY KEY(stuid);
-- 唯一（UNIQUE）约束
ALTER TABLE teacher ADD CONSTRAINT uk_teacher_idnumber UNIQUE (idnumber);
-- 外键（FOREIGN KEY）约束
-- dept为主表，emp为从表（外键表），emp中的外键列deptno引用dept中的主键
ALTER TABLE emp ADD CONSTRAINT fk_scott_emp_teptno FOREIGN KEY(deptno) REFERENCES
scott.dept(deptno);
-- 条件（CHECK）约束
--给字段年龄age添加约束，学生的年龄只能0-50岁之内的
alter table STUINFO add constraint ch_stuinfo_age check(age>0 and age<=50);
-- 限定sex的值
alter table STUINFO add constraint ch_stuinfo_sex check(sex='1' or sex='0');
-- 限定年级的范围
alter table STUINFO add constraint ch_stuinfo_grade check (grade>='1900' and
grade<='2999');

```

查询语法

用户对表或视图最常进行的操作就是检索数据。检索数据通过SELECT语句来实现，该语句由多个子句组成，通过这些子句可以完成筛选、投影和连接等各种数据操作，最终得到用户想要的查询结果。
SELECT语句的基本语法格式如下：

```
SELECT {[ DISTINCT | ALL ] columns | *}
[INTO table_name]
FROM {tables | views | other select}
[WHERE conditions]
[GROUP BY columns]
[HAVING conditions]
[ORDER BY columns]
```

- SELECT子句：用于选择数据表、视图中的列。
- INTO子句：用于将原表的结构和数据插入新表中。
- FROM子句：用于指定数据来源，包括表、视图和其他SELECT语句。
- WHERE子句：用于对检索的数据进行筛选。
- GROUP BY子句：用于对检索结果进行分组显示。
- HAVING子句：用于从使用GROUP BY子句分组后的查询结果中筛选数据行。
- ORDER BY子句：用于对结果集进行排序（包括升序和降序）。

简单查询

只包含SELECT子句和FROM子句的查询就是简单查询。SELECT子句和FROM子句是SELECT语句的必选项，即每个SELECT语句都必须包含这两个子句。
创建部门信息表dept、员工信息表emp、工资等级表salgrade，并插入测试数据。

```
CREATE TABLE DEPT
       (DEPTNO NUMBER(2) CONSTRAINT PK_DEPT PRIMARY KEY,
        DNAME VARCHAR2(14) ,
        LOC VARCHAR2(13) ) ;
CREATE TABLE EMP
       (EMPNO NUMBER(4) CONSTRAINT PK_EMP PRIMARY KEY,
        ENAME VARCHAR2(10),
        JOB VARCHAR2(9),
        MGR NUMBER(4),
        HIREDATE DATE,
        SAL NUMBER(7,2),
        COMM NUMBER(7,2),
        DEPTNO NUMBER(2) CONSTRAINT FK_DEPTNO REFERENCES DEPT);
CREATE TABLE salgrade (
        grade NUMBER,
        losal NUMBER,
        hisal NUMBER );

-- 插入测试数据 —— dept
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');
INSERT INTO dept VALUES (30,'SALES','CHICAGO');
INSERT INTO dept VALUES (40,'OPERATIONS','BOSTON');

-- 插入测试数据 —— emp
INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,to_date('17-12-1980','dd-mm-yyyy'),800,NULL,20);
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,to_date('20-2-1981','dd-mm-yyyy'),1600,300,30);
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,to_date('22-2-1981','dd-mm-yyyy'),1250,500,30);
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,to_date('2-4-1981','dd-mm-yyyy'),2975,NULL,20);
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,to_date('28-9-1981','dd-mm-yyyy'),1250,1400,30);
INSERT INTO emp VALUES (7698,'BLAKE','MANAGER',7839,to_date('1-5-1981','dd-mm-yyyy'),2850,NULL,30);
INSERT INTO emp VALUES (7782,'CLARK','MANAGER',7839,to_date('9-6-1981','dd-mm-yyyy'),2450,NULL,10);
```

```
INSERT INTO emp VALUES (7788,'SCOTT','ANALYST',7566,to_date('13-07-87','dd-mm-yyyy')-85,3000,NULL,20);
INSERT INTO emp VALUES (7839,'KING','PRESIDENT',NULL,to_date('17-11-1981','dd-mm-yyyy'),5000,NULL,10);
INSERT INTO emp VALUES (7844,'TURNER','SALESMAN',7698,to_date('8-9-1981','dd-mm-yyyy'),1500,0,30);
INSERT INTO emp VALUES (7876,'ADAMS','CLERK',7788,to_date('13-07-87','dd-mm-yyyy')-51,1100,NULL,20);
INSERT INTO emp VALUES (7900,'JAMES','CLERK',7698,to_date('3-12-1981','dd-mm-yyyy'),950,NULL,30);
INSERT INTO emp VALUES (7902,'FORD','ANALYST',7566,to_date('3-12-1981','dd-mm-yyyy'),3000,NULL,20);
INSERT INTO emp VALUES (7934,'MILLER','CLERK',7782,to_date('23-1-1982','dd-mm-yyyy'),1300,NULL,10);

-- 插入测试数据 —— salgrade
INSERT INTO salgrade VALUES (1,700,1200);
INSERT INTO salgrade VALUES (2,1201,1400);
INSERT INTO salgrade VALUES (3,1401,2000);
INSERT INTO salgrade VALUES (4,2001,3000);
INSERT INTO salgrade VALUES (5,3001,9999);

-- 事务提交
COMMIT;
```

检索所有列

```
--SELECT后跟星号 * 检索所有列
SELECT * FROM 表名
```

FROM子句的后面还可以指定多张数据表，每张数据表名之间使用逗号(,)分隔：

```
SELECT * FROM 表名1,表名2,表名3,...,表名n
```

检索指定的列

可以指定查询表中的某些列（也称为投影操作），而不是全部列，并且被指定列的顺序不受限制。

```
SELECT 列名1,列名2,列名3,...,列名n FROM 表名

```

在Oracle数据库中，有一个标识行中唯一特性的行标识符ROWID。ROWID是Oracle数据库内部使用的隐藏列，由于该列实际上并不是定义在表中，因此也被称为伪列。伪列ROWID长度为18位字符，包含该行数据在Oracle数据库中的物理地址。

```
--查询emp表中的rowid，job，ename三列数据
select rowid,job,ename from emp;
```

查询日期列

日期列是指数据类型为DATE的列。查询日期列与查询其他列没有任何区别，但日期列的默认显示格式为DD-MON-RR。

可以指定查询数据中日期的显式格式，如下：

1. 以简体中文格式显示日期结果

```
--将sql*plus的nls_date_language参数设置为中文格式
alter session set nls_date_language = 'SIMPLIFIED CHINESE';
--查询emp表
select ename,hiredate from emp;
```

以美式英语格式显示日期结果

```
--将sql*plus的nls_date_language参数设置为美式英语格式
alter session set nls_date_language = 'AMERICAN';
--查询emp表
select ename,hiredate from emp;
```

以特定格式显示日期结果

```
--将sql*plus的nls_date_format参数设置为××××年××月××日格式
alter session set nls_date_format = 'YYYY"年"MM"月"DD"日"';
--查询emp表
select ename,hiredate from emp;
```

带有表达式的SELECT子句

在SELECT语句中，数字和日期数据可以使用(+)、减(-)、乘(*)、除(/)和括号等算术运算符；

```
--将emp表的sal列的数值显式为原值的1.1倍后的值
select sal*(1+0.1),sal from emp;
```

为列指定别名

在Oracle系统中，为列指定别名既可以使用AS关键字，也可以不使用任何关键字而直接指定。

```
--使用as指定列别名
select empno as "员工编号",ename as "员工名称",job as "职务"  from emp;
--不使用任何关键字直接指定列别名
select empno "员工编号",ename "员工名称",job "职务"  from emp;
```

显示不重复记录

在SELECT语句中，可以使用DISTINCT关键字来限制显示重复的数据，该关键字用在SELECT子句的列表前面。

```
--查询emp表中的job列，可以看到有重复值
select job from emp;
--使用DISTINCT来去除重复值
select distinct job from emp;
```

处理NULL值

NULL表示未知值，它既不是空格，也不是0。插入数据时，如果没有为特定列提供数据，并且该列没有默认值，那么其结果为NULL。

当算术表达式中包含NULL时，如果不处理NULL，显示结果将为空。

```
--comm列值为NULL的，计算sal+comm的值也为NULL
select ename,sal,comm,sal+comm from emp;
```

为了避免出现上面这种情况，可以使用NVL函数处理NULL；

```
--使用NVL函数处理comm列，如果值为数值，返回原数值，如果值为NULL转换成0
select ename,sal,comm,sal+nvl(comm,0) from emp;
```

连接字符串

连接字符串可以使用“||”操作符或者CONCAT函数。当连接字符串时，如果是在字符串中加入数字值，可以**直接指定数字值**；而如果是在字符串中加入**字符值或者日期值**，那么**必须将值放在单引号中**。

```
--使用||将ename列与job列进行连接
select ename||''''||'s job is '||job from emp;

```

在字符串中使用单引号，需要使用两个连续的单引号来表示一个单引号字符。

```
--使用concat连接ename列和sal列
select concat(concat(ename, '''s salary is '),sal) from emp;
```

筛选查询

在SELECT语句中使用WHERE子句实现对数据行的筛选操作，只有满足WHERE子句判断条件的行才会显示在结果集中。

```
SELECT columns_list
FROM table_name
WHERE conditional_expression

```

- columns_list——字段列表；
- table_name——表名；
- conditional_expression——筛选条件表达式；

比较筛选

可以在WHERE子句中使用比较运算符来筛选数据，基本的“比较筛选”操作主要有以下6种情况：

- A=B：比较A与B是否相等。
- A!B或A< >B：比较A与B是否不相等。
- A>B：比较A是否大于B。
- A<B：比较A是否小于B。
- A>=B：比较A是否大于或等于B。
- A<=B：比较A是否小于或等于B。

```
--查询emp表中工资（sal）大于2000的数据记录
select empno,ename,sal from emp where sal > 2000;
```

除了基本的“比较筛选”操作，还有以下两个特殊的“比较筛选”操作：

- A{operator}ANY(B)：表示A与B中的任何一个元素进行operator运算符的比较，只要有一个比较值为TRUE，就返回数据行。
- A{operator}ALL(B)：表示A与B中的所有元素进行operator运算符的比较，只有与所有元素比较值都为TRUE，才返回数据行。

```
--使用all关键字过滤工资（sal）同时不等于3000、5000、800的员工记录
select empno,ename,sal from emp where sal <> all(3000,5000,800);
```

注意：在进行比较筛选的过程中，字符串和日期的值必须使用单引号标识，否则Oracle会提示标识符无效。

使用特殊关键字筛选

SQL语言提供了LIKE、IN、BETWEEN和IS NULL等关键字来筛选数据；

LIKE关键字——字符串模式匹配或字符串模糊查询

LIKE关键字需要使用通配符在字符串内查找指定的模式，主要使用以下两个通配符：

- %：代表0个或多个字符。
- _：代表一个且只能是一个字符。

```
--查询emp表中以A开头的员工名称
select empno,ename,job from emp where ename like 'A%';
```

在LIKE关键字前面加上NOT，表示否定的判断，如果LIKE为真，则NOT LIKE为假。另外，也可以**在IN、BETWEEN、IS NULL和IS NAN等关键字前面加上NOT来表示否定的判断**。

```
--查询工作是MANAGER的员工，但是不记得MANAGER的准确拼写了，仅记得第一个字母是M，第三个字母是N，第五个字母是G
select empno,ename,job from emp where job like 'M_N_G%';
```

LIKE关键字还可以帮助简化某些WHERE子句；例如：

```
--在emp表中，显示1981年入职的员工信息
select empno,ename,sal,hiredate from emp where hiredate like '%81%';
```

如果要查询的字符串中含有“%”或“_”，可以使用转义escape关键字实现查询；

```
--创建一张与dept表的结构和数据都相同的表dept_temp
create table dept_temp  as  select * from dept;
--插入一条数据
insert into dept_temp  values(60,'IT_RESEARCH','SHANGHAI');
--提交
commit;

```

显示临时表dept_temp中所有部门名称以IT_开头的数据行，代码如下:

```
--通过转义字符\将_转义为本来的含义下划线而不再是单字符通配符
select * from dept_temp where dname like 'IT\_%' escape '\';
```

没有必要一定使用“\”字符作为转义符，可以使用任何字符来作为转义符。许多Oracle的专业人士之所以经常使用“\”字符作为转义符，是因为该字符在UNIX操作系统和C语言中就是转义符。

```
--使用a作为转义字符，查询出的结果是一样的
select * from dept_temp where dname like 'ITa_%' escape 'a';
```

IN关键字——测试一个数据值是否匹配一组目标值中的一个

IN关键字的格式是**IN（目标值1,目标值2,目标值3,...）**

```
--emp表中，使用IN关键字查询职务为CLERK、MANAGER或ANALYST的员工信息
select empno,ename,job from emp where job in('CLERK','MANAGER','ANALYST');
```

NOT IN表示查询指定的值不在某一组目标值中，这种方式在实际应用中也很常见。

```
--查询职务不为CLERK、MANAGER或ANALYST的员工信息
select empno,ename,job from emp where job not in('CLERK','MANAGER','ANALYST');
```

BETWEEN关键字——用于比较一个值是否在指定范围内

通常使用BETWEEN…AND和NOT…BETWEEN…AND来指定范围条件。

使用BETWEEN…AND查询条件时，指定的**第一个值必须小于第二个值**。因为BETWEEN…AND实质是查询条件“大于或等于第一个值，并且小于或等于第二个值”的简写形式，即BETWEEN…AND要**包括两端的值**，等价于比较运算符(>=…<=)。

```
--在emp表中，查询工资(sal)为1000～2000的员工信息
select empno,ename,sal from emp where sal between 1000 and 2000;
```

NOT BETWEEN…AND语句用于返回在两个指定值范围以外的某个数据值，并且不包括两个指定的值。

```
--在emp表中，查询工资(sal)不在1000～2000范围内的员工信息
select empno,ename,sal from emp where sal not between 1000 and 2000;
```

IS NULL关键字——筛选出特定列的值为空的行

空值(NULL)从技术上来说就是未知的、不确定的值，但空值与空字符串不同，因为空值是不存在的值，而空字符串是长度为0的字符串。
因为空值代表的是未知的值，所以并不是所有的空值都相等。例如，student表中有两名学生的年龄未知，但无法证明这两名学生的年龄相等，因此不能用“=”运算符来检测空值。SQL引入了IS NULL关键字来检测特殊值之间的等价性，IS NULL关键字通常在WHERE子句中使用。

```
--IS NULL关键字查询emp表中没有奖金(comm为空值)的员工信息
select empno,ename,sal,comm from emp where comm is null;
```

逻辑筛选

逻辑筛选是指在WHERE子句中使用逻辑运算符AND、OR和NOT进行数据筛选操作，这些逻辑运算符可以把多个筛选条件组合起来，便于获取更加准确的数据记录。
AND逻辑运算符表示两个逻辑表达式之间是“逻辑与”的关系，可以**使用AND运算符加比较运算符来代替BETWEEN...AND关键字**。

```
--在emp表中，使用AND运算符查询工资(sal)为1000～2000的员工信息
select empno,ename,sal from emp where sal >= 1000 and sal <= 2000;

--在emp表中，查询工资(sal)为1000～2000的员工信息
select empno,ename,sal from emp where sal between 1000 and 2000;

```

上面两条语句执行结果是一样的；
OR逻辑运算符表示两个逻辑表达式之间是“逻辑或”的关系，两个表达式的**结果中有一个为TRUE，则这个逻辑或表达式的值就为TRUE**。

```
--在emp表中，使用OR逻辑运算符查询工资低于1000或工资高于2000的员工信息
select empno,ename,sal from emp where sal < 1000 or sal > 2000;
```

NOT逻辑运算符用于对表达式执行逻辑非运算，即**对条件取反**，如果条件为true，使用NOT运算符后该条件将变为false，反之亦然。

分组查询

在查询结果集中使用GROUP BY子句对记录进行分组。在SELECT语句中，GROUP BY子句位于FROM子句之后，其语法格式如下：

```
SELECT columns_list
FROM table_name
[WHERE conditional_expression]
GROUP BY columns_list

```

- columns_list：字段列表，在GROUP BY子句中也可以指定多个列分组。
- table_name：表名。
- conditional_expression：筛选条件表达式。

使用GROUP BY子句进行单列分组

单列分组是指基于列生成分组统计结果。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。

```
--在emp表中，按照部门编号(deptno)列进行分组，并显示每个部门有几个岗位
select deptno,count(*) as 岗位数 from emp group by deptno order by deptno;
```

GROUP BY子句经常与聚集函数一起使用。使用GROUP BY子句和聚集函数，可以实现对查询结果中每一组数据进行分类统计。

```
--在emp表中，使用GROUP BY子句对工资记录进行分组，并计算平均工资(AVG)、所有工资的总和(SUM)、最高工资(MAX)和各组的行数(COUNT)
select job 岗位,avg(sal) 平均工资,sum(sal) 工资总和,max(sal) 最高工资,count(job) 岗位数 from emp group by job;
```

使用GROUP BY子句时的注意事项：

- 在SELECT子句的后面只可以有统计函数和进行分组的列名两类表达式。
- SELECT子句中的列名必须是进行分组的列，但是GROUP BY子句后面的列名可以不出现在SELECT子句中。
- 默认按照GROUP BY子句指定的分组列升序排列，可以使用ORDER BY子句指定新的排列顺序。
- 如果在一个查询中使用了分组函数，则查询中的任何不在分组函数中的列或表达式必须在GROUP BY子句中出现。

下面是一个错误示例：

```
--查询岗位（job）和岗位的平均工资
select job,avg(sal) from emp;
```

没有使用GROUP BY子句，出现错误，提示job不是单分组函数；查询的job是多条数据，而avg(sal)只有一条数据，这两个查询要求是矛盾的；
使用GROUP BY子句改正上面的错误：

```
--查询岗位（job）和岗位的平均工资
select job,avg(sal) from emp group by job;

```

通过group by子句将job列按相同数据进行分组，然后avg(sal)计算出每一组的平均工资；

使用GROUP BY子句进行多列分组

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。

```
--查询emp表，显示每个部门每种岗位的平均工资和最高工资
select deptno,job,avg(sal),max(sal) from emp group by deptno,job;
```

使用ORDER BY子句改变分组排序结果

GROUP BY子句执行分组统计时，会自动基于分组列进行升序排列。使用ORDER BY子句可以改为降序；

```
--查询每个部门的部门编号和工资总额按默认排序排列
select deptno,sum(sal) from emp group by deptno
--查询每个部门的部门编号和工资总额并按降序排列
select deptno,sum(sal) from emp group by deptno order by sum(sal) desc;
```

使用HAVING子句限制分组结果

GROUP BY子句进行分组的结果可使用HAVING子句对分组的结果做进一步的筛选。

```
--查询部门平均工资高于1000的部门编号和平均工资
select deptno as 部门编号,avg(sal) as 平均工资
from emp
group by deptno
having avg(sal) > 1000 ;
```

在GROUP BY子句中使用ROLLUP和CUBE操作符

1. 使用ROLLUP操作符创建多层次的汇总数据

ROLLUP操作符在Oracle中用于生成分组汇总报表，它可以在GROUP BY子句中指定多个列，并生成这些列的所有可能组合的汇总行。

```
-- 显示每个部门的平均工资、所有员工的平均工资
select deptno as 部门编号, job as 岗位, avg(sal) as 平均工资
from emp
group by rollup(deptno,job) ;
```

2. 使用CUBE操作符根据指定的列表生成不同的排列组合，并根据每一种组合结果生成统计汇总

```
--显示各岗位的平均工资、每个部门的平均工资、所有员工的平均工资
select deptno as 部门编号, job as 岗位, avg(sal) as 平均工资
from emp
group by cube(deptno,job) ;
```

3. 使用GROUPING函数确定统计结果是否使用了特定列

当使用ROLLUP或者CUBE操作符生成统计结果时，某个统计结果行可能用到一列或者多列，也可能没有使用任何列。使用GROUPING函数确定统计结果是否使用了特定列。如果该函数返回0，则表示统计结果使用了该列；如果函数返回1，则表示统计结果没有使用该列。

```
--使用GROUPING函数确定统计结果所使用的列
select deptno,job, sum (sal),grouping(deptno),grouping(job)
from emp
group by rollup(deptno,job) ;
```

4．在ROLLUP操作符中使用复合列

复合列被看作一个逻辑单元的列组合，当引用复合列时，需要用括号括住相关列。通过在ROLLUP操作符中使用复合列，可以略过ROLLUP操作符的某些统计结果。

例如，子句GROUP BY ROLLUP(a,b,c)的统计结果等同于GROUP BY(a,b,c)、GROUP BY(a,b)、GROUP BY a以及GROUP BY()的并集；而如果将(b,c)作为复合列，那么子句GROUP BY ROLLUP(a,(b,c))的结果等同于GROUP BY(a,b,c)、GROUP BY a以及GROUP BY()的并集。

```
--在emp表中显示特定部门特定岗位的工资总额以及所有员工的工资总额
select deptno,job, sum (sal) from emp group by rollup((deptno,job)) ;
--显示特定部门特定岗位的工资总额、部门的工资总额及所有员工的工资总额
select deptno,job, sum (sal) from emp group by rollup(deptno,job) ;

```

group by rollup((deptno,job)) 等同于group by(deptno,job)与group by()的并集，即特定部门特定岗位的工资总额与所有员工的工资总额的并集，不包括部门工资总额；

5．在CUBE操作符中使用复合列

通过在CUBE操作符中使用复合列，可以略过CUBE操作符的某些统计结果。

例如，子句GROUP BY CUBE(a,b,c)的统计结果等同于GROUP BY(a,b,c)、GROUP BY(a,b)、GROUP BY(a,c)、GROUP BY(b,c)、GROUP BY a、GROUP BY b、GROUP BY c以及GROUP BY()的并集；而如果将(a,b)作为复合列，那么子句GROUP BY CUBE((a,b),c)的结果等同于GROUP BY(a,b,c)、GROUP BY(a,b)、GROUP BY c以及GROUP BY()的并集。

```
--在CUBE操作符中使用复合列，在emp表中显示特定部门特定岗位的工资总额以及所有员工的工资总额
select deptno,job, sum (sal) from emp group by cube ((deptno,job)) ;
```

6．使用GROUPING SETS操作符合并多个分组的统计结果

```
--使用部门编号(deptno)执行分组统计每个部门的平均工资
select deptno,avg (sal) from emp group by deptno;
--使用岗位（job）显示每个岗位的平均工资
select job,avg (sal) from emp group by job ;
--使用grouping sets显示部门的平均工资和岗位的平均工资
select deptno,job,avg (sal) from emp group by grouping sets(deptno,job);

```

使用grouping sets(deptno,job)既显示了部门平均工资，又同时显示了岗位平均工资；

排序查询

在SELECT语句中，可以使用ORDER BY子句对检索的结果集进行排序，该子句位于FROM子句之后，其语法格式如下：

```
SELECT columns_list
FROM table_name
[WHERE conditional_expression]
[GROUP BY columns_list]
ORDER BY { order_by_expression [ ASC | DESC ] } [ ,...n ]

```

- columns_list：字段列表，在GROUP BY子句中也可以指定多个列分组。
- table_name：表名。
- conditional_expression：筛选条件表达式。
- order_by_expression：表示要排序的列名或表达式。关键字ASC表示按升序排列，这也是默认的排序方式；关键字DESC表示按降序排列。

ORDER BY子句可以根据查询结果中的一个列或多个列对查询结果进行排序，并且第一个排序项是主要的排序依据，其他的是次要的排序依据。

```
--使用默认排序查询
select deptno,empno,ename from emp;
--按deptno升序排序（单列排序）
select deptno,empno,ename from emp order by deptno;
--按deptno升序排序后，再按empno升序排序（多列排序）
select deptno,empno,ename from emp order by deptno,empno;

```

注意：当在SELECT语句中同时包含多个子句(WHERE、GROUP BY、HAVING、ORDER BY)时，ORDER BY必须是最后一个子句。

ORDER BY子句不仅可以按照列名、列别名进行排序，也可以按照列或表达式在选择列表中的位置进行排序；但是不推荐使用列号进行排序，这种用法可读性很差；

```
--查询员工的年薪，并按年薪降序排列,这里计算年薪在第3列，直接使用列号排序
select empno,ename,sal*12 年薪 from emp order by 3 desc;
```

多表关联查询

多表关联查询是在关系型数据库中，通过使用多个表之间的关联条件来查询相关联的数据。这种查询通常用于解决复杂的数据关系问题，涉及多个表之间的关联和数据整合表的别名
在多表关联查询时，如果多张表之间存在同名的列，则必须使用表名来限定列的引用。与列别名类似，可以给表指定一个别名，使用简短的表别名可以替代原有较长的表名称，可以大大缩减语句的长度;

```
--指定emp表别名为e，dept表别名为d，查询岗位MANAGER的员工信息及部门
select e.empno as 员工编号, e.ename as 员工名称, d.dname as 部门名称
from emp e,dept d
where e.deptno=d.deptno and e.job='MANAGER';
```

使用表的别名的注意事项如下：

- 表的别名在FROM子句中定义，别名放在表名之后，它们之间用空格隔开。
- 别名一经定义，在整个查询语句中就只能使用表的别名而不能再使用表名。
- 表的别名只在所定义的查询语句中有效。
- 应该选择有意义的别名，表的别名最长为30个字符，但越短越好。

内连接

内连接是一种常用的多表关联查询方式，一般使用关键字INNER JOIN来实现。INNER关键字可以省略，当只使用JOIN关键字时，语句只表示内连接操作。内连接使用JOIN指定用于连接的两张表，使用ON指定连接表的连接条件。**内连接只显示与连接条件匹配的行**，语法格式如下：

```
SELECT columns_list
FROM table_name1 [INNER] JOIN table_name2 ON join_condition;

```

- columns_list：字段列表。

- table_name1和table_name2：两张要实现内连接的表。
- join_condition：实现内连接的条件表达式。

```
--通过deptno字段来内连接emp表和dept表，并检索这两张表中相关字段的信息
select e.empno as 员工编号, e.ename as 员工名称, d.dname as 部门
from emp e inner join dept d on e.deptno=d.deptno;
```

外连接

外连接分为以下3类：

- 左外连接：关键字为LEFT OUTER JOIN或LEFT JOIN。
- 右外连接：关键字为RIGHT OUTER JOIN或RIGHT JOIN。
- 完全外连接：关键字为FULL OUTER JOIN或FULL JOIN。

外连接不只列出与连接条件匹配的行，还能够列出左表（左外连接时）、右表（右外连接时）或两张表（全部外连接时）中所有符合搜索条件的数据行。

左外连接

左外连接的查询结果中不仅包含满足连接条件的数据行，还包含左表中不满足连接条件的数据行。左外连接以左边的表为主表，显示主表所有数据。

```
--向emp表中插入新记录（没有为deptno和dname列插入值，即它们的值为NULL）
insert into emp(empno,ename,job) values(9527,'EAST','SALESMAN');
--通过deptno进行emp表和dept表的左外连接
select e.empno,e.ename,e.job,d.deptno,d.dname
from emp e left join dept d on e.deptno=d.deptno;

```

可以看到deptno为空的数据也被查询出来，左外连接就是以左边的表emp为主表，显示emp主表的所有记录，以条件e.deptno=d.deptno匹配dept表中的数据，即使EMPNO=9527的记录没有匹配上dept表的记录也会显示出来；

右外连接

右外连接的查询结果中不仅包含满足连接条件的数据行，还包含右表中不满足连接条件的数据行。右外连接以右边的表为主表，显示主表所有数据。

```
--通过deptno进行emp表和dept表的右外连接
select e.empno,e.ename,e.job,d.deptno,d.dname
from emp e right join dept d on e.deptno=d.deptno;
```

在外连接中也可以使用外连接的连接运算符，**外连接的连接运算符为“(+)”**，该连接运算符可以放在等号的左边，也可以放在等号的右边，但一定要放在显示较少行（完全满足连接条件行）的一端；
上面的右外连接查询语句还可以这么写，代码如下：

```
select e.empno,e.ename,e.job,d.deptno,d.dname
from emp e , dept d
where e.deptno(+)=d.deptno;

```

(+)在哪个表的列名后面，则另一个表为主表，e.deptno(+)=d.deptno则dept为主表，即语句为右连接，e.deptno=d.deptno(+)则emp为主表，即语句为左连接；
使用“(+)”操作符时应注意以下3点：

- 当使用“(+)”操作符执行外连接时，如果在WHERE子句中包含多个条件，则必须在所有条件中都包含“(+)”操作符。
- “(+)”操作符只适用于列，而不能用在表达式上。
- “(+)”操作符不能与ON和IN操作符一起使用。

完全外连接

在执行完全外连接时，Oracle会执行一个完整的左外连接和右外连接查询，然后将查询结果合并，并消除重复的记录行。

```
select e.empno,e.ename,e.job,d.deptno,d.dname
from emp e full join dept d
on e.deptno=d.deptno;
```

自然连接

自然连接是指在检索多张表时，Oracle会将第一张表中的列与第二张表中具有相同名称的列进行自动连接。在自然连接中，用户不需要明确指定进行连接的列，这个任务由Oracle系统自动完成，自然连接使用NATURAL JOIN关键字。

```
--查询工资（sal字段）高于1000的记录，并实现emp表与dept表的自然连接
select empno,ename,job,dname
from emp natural join dept
where sal > 1000;
```

自然连接强制要求表之间具有相同的列名称，容易在设计表时出现不可预知的错误，所以在实际应用系统开发中很少用到自然连接。

自连接

自连接主要用在自参照表上，显示上下级关系或者层次关系。自参照表是指在同一张表的不同列之间具有参照关系或主从关系的表。自连接是在同一张表之间的连接查询，必须定义表别名。

```
--查询所有管理者所管理的下属员工信息
select em2.ename 上层管理者,em1.ename as 下属员工
from emp em1 left join emp em2
on em1.mgr=em2.empno
order by em1.mgr;
```

交叉连接

交叉连接实际上就是不需要任何连接条件的连接，它使用CROSS JOIN关键字来实现，其语法格式如下：

```
SELECT colums_list
FROM table_name1 CROSS JOIN table_name2

```

- colums_list：字段列表。
- table_name1和table_name2：两张实现交叉连接的表。

交叉连接的执行结果是一个笛卡尔积，这种查询结果是非常冗余的，但可以通过WHERE子句来过滤出有用的记录信息。

笛卡尔积（Cartesian product）是指在没有明确指定连接条件的情况下，将两个或多个表中的所有行进行组合。这种组合操作不考虑表之间的关联关系，只是简单地将每一行与其他表中的每一行进行组合。

```
--计算dept表与emp表的记录两两组合的行数
select count(*) from dept cross join emp;
```

可以看到dept表中有4条数据，emp表中有15条数据，交叉连接将dept表中的每一条数据都与emp表中数据进行组合，最终结果为4 * 15=60条数据；

在执行数据操作（包括查询、添加、修改和删除等）的过程中，如果某个操作需要依赖于另一个SELECT语句的查询结果，那么可以把SELECT语句嵌入该操作语句中，从而形成一个子查询。

子查询和关联子查询

子查询是在SQL语句内的另一条SELECT语句，也被称为内查询或内SELECT语句。在SELECT、INSERT、UPDATE或DELETE命令中允许是一个表达式的地方都可以包含子查询，子查询甚至可以包含在另一个子查询中。

```
--在emp表中查询部门名称为SALES的员工信息
select empno,ename,job from emp
where deptno=(select deptno from dept where dname='SALES');

```

在emp表中是不存在dname字段（部门名称），但emp表中存在deptno字段（部门代码）；dname字段、deptno字段存在于dept表中，所以deptno为两张表之间的关联字段。
也可以通过多表关联来实现在emp表中查询部门名称为SALES的员工信息；子查询与多表关联查询能实现一样的查询功能，**子查询易读，更容易理解，但多表关联查询效率高于子查询**；

```
select empno,ename,job from emp
join dept on emp.deptno=dept.deptno
where dept.dname = 'SALES';
```

在执行子查询操作的语句中，子查询也被称为内查询，包含子查询的查询语句也被称为外查询或主查询。
在一般情况下，外查询语句检索一行，子查询语句需要检索一遍数据，然后判断外查询语句的条件是否满足。如果条件满足，则外查询语句将检索到的数据行添加到结果集中；如果条件不满足，则外查询语句继续检索下一行数据。所以子查询相对多表关联查询要慢一些。
在使用子查询时，应注意以下规则：

- 子查询必须用括号“()”括起来。
- 子查询中不能包括ORDER BY子句。
- 子查询允许嵌套多层，但不能超过255层。

在Oracle中，通常把子查询再细化为**单行子查询、多行子查询和关联子查询**3种。

单行子查询

单行子查询是指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，可以使用单行比较运算符（=、>、<、>=、<=和<>）。

```
--查询emp表中既不是最高工资，也不是最低工资的员工信息
select empno,ename,sal from emp
where sal > (select min(sal) from emp)
and sal < (select max(sal) from emp);
```

多行子查询

多行子查询是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、ANY、ALL)。

使用IN运算符

当在多行子查询中使用IN运算符时，外查询会尝试与子查询结果中的任何一个结果进行匹配，只要有一个匹配成功，则外查询返回当前检索的记录。

```
--查询不是研发部门（RESEARCH）的员工信息
select empno,ename,job
from emp where deptno in
(select deptno from dept where dname<>'RESEARCH');
```

使用ANY运算符

ANY运算符必须与单行操作符结合使用，并且返回行只要匹配子查询的任何一个结果即可。

```
--查询工资高于部门编号为20的任意一个员工工资的其他部门的员工信息
select deptno,ename,sal from emp where sal > any
(select sal from emp where deptno = 20) and deptno <> 20;
```

使用ALL运算符

ALL运算符必须与单行运算符结合使用，并且返回行必须匹配所有子查询结果。

```
--查询工资高于部门编号为30的所有员工工资的员工信息
select deptno,ename,sal from emp where sal > all
(select sal from emp where deptno = 30);
```

关联子查询

在单行子查询和多行子查询中，内查询和外查询是分开执行的，也就是说，内查询的执行与外查询的执行是没有关系的，外查询仅仅是使用内查询的最终结果。

在一些特殊需求的子查询中，内查询的执行需要借助外查询，而外查询的执行又离不开内查询的执行，这时，内查询和外查询是相互关联的，这种子查询就被称为关联子查询。

```
--使用关联子查询检索工资高于同职位的平均工资的员工信息
select empno,ename,sal
from emp f
where sal > (select avg(sal) from emp where job = f.job)
order by job;

```

在上述查询语句中，内层查询使用关联子查询计算每个职位的平均工资，而关联子查询必须知道职位的名称。为此，外层查询就使用f.job字段值为内层查询提供职位名称，以便于计算出某个职位的平均工资。如果外层查询正在检索的数据行的工资高于平均工资，则会对该行的员工信息进行显示；否则不显示。

注意：在执行关联子查询的过程中，必须遍历数据表中的每条记录，因此如果被遍历的数据表中有大量数据记录，则关联子查询的执行速度会比较缓慢。

Oracle集合运算

Oracle集合运算就是把多个查询结果组合成一个查询结果，oralce的集合运算包括：INTERSECT(交集)、UINION ALL(交集重复)、UINION(交集不重复)、MINUS(补集)。表取别名时不要用as ,直接用空格。

- 1、intersect(交集)，返回两个查询共有的记录。
- 2、union all(并集重复)，返回各个查询的所有记录，包括重复记录。

- 3、union(并集不重复)，返回各个查询的所有记录，不包括重复记录 （重复的记录只取一条）。
- 4、minus(补集)，返回第一个查询检索出的记录减去第二个查询检索出的记录之后剩余的记录。

当我们使用Oracle集合运算时，要注意每个独立查询的字段名的列名尽量一致（列名不同时，取第一个查询的列名）、列的数据类型、列的个数要一致，不然会报错

```
-- 0、数据准备
create table emp01 as select * from emp;
select * from emp;  -- 14条记录
select * from emp01; -- 复制emp表的14条记录

-- 1、INTERSECT(交集)，返回两个查询共有的记录。
select * from emp
intersect  --交集
select * from emp01; --两个表相同的记录，这里也是14条

-- 2、UNION ALL(并集重复)，返回各个查询的所有记录，包括重复记录。
select * from emp
union all    --并集重复
select * from emp01; --28条记录

-- 3、UNION(并集不重复)，返回各个查询的所有记录，不包括重复记录 （重复的记录只取一条）。
select * from emp
union  --并集不重复
select * from emp01;  --14条记录

-- 4、MINUS(补集)，返回第一个查询检索出的记录减去第二个查询检索出的记录之后剩余的记录
select * from emp
minus --补集
select * from emp01;  --0条记录
```

数据操纵语言DML

插入、删除和更新操作使用的SQL语言，称为数据操纵语言(data manipulation language，DML)，它们分别对应INSERT、DELETE和UPDATE这3种语句。在Oracle中，DML除了包括上述提到的3种语句，还包括TRUNCATE、CALL、LOCKTABLE和MERGE等语句。

插入数据（INSERT语句）

Oracle数据库通过INSERT语句来实现插入数据记录，该语句既可以实现向数据表中一次插入一条记录，也可以使用SELECT子句将查询结果集批量插入数据表中。

使用INSERT语句有以下注意事项：

- 当为数字列增加数据时，可以直接提供数字值，或者用单引号引住。
- 当为字符列或日期列增加数据时，必须用单引号引住。
- 当增加数据时，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。
- 当增加数据时，数据必须与列的个数和顺序保持一致。

插入单条数据

插入单条数据是INSERT语句最基本的用法，语法格式如下：

```
INSERT INTO table_name [(column_name1[,column_name2]…)] VALUES(express1[,express2]…)

```

- table_name：表示要插入的表名。
- column_name1和column_name2：指定表的完全或部分列名称，如果指定多个列，那么列之间用逗号分开。
- express1和express2：表示要插入的值列表。

当使用INSERT语句插入数据时，既可以指定列，也可以不指定列。如果不指定列，那么在VALUES子句中必须为每一列提供数据，并且数据顺序必须与列表顺序完全一致；如果指定列，则只需要为相应列提供数据。

指定列增加数据

在INSERT INTO子句中指定添加数据的列，并在VALUES子句中为每列提供一个值是最常用的形式。

```
--向dept表中的deptno,dname两列插入数据
insert into dept(deptno,dname) values(90,'abc');

```

在上述示例中，INSERT INTO子句中指定添加数据的列，既可以是数据表的全部列，也可以是部分列。在指定部分列时，需要注意不允许为空(NOT NULL)的列必须被指定出来，并且在VALUES子句中的对应赋值也不允许为NULL，否则系统显示“无法将NULL插入”的错误信息提示。

不指定列增加数据

在向表的所有列中添加数据时，也可以省略INSERT INTO子句后面的列表清单，使用这种方法时，必须根据表中定义的列的顺序，为所有的列提供数据。

```
--不指定列，向dept中插入数据
insert into dept values(88,'design','beijing');

```

在SQL * Plus中使用desc dept命令查看dept的表结构和列的顺序，可以看到只有deptno、dname、loc三列，所以上述insert into语句的values给定了三个值；

使用特定格式插入日期值

当增加日期数据时，默认情况下日期值必须匹配于日期格式和日期语言，否则在插入数据时会出现错误信息。如果希望使用习惯方式插入日期数据，那么必须使用TO_DATE函数进行转换。

```
--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

当增加数据时，可以使用DEFAULT提供数值。当指定DEFAULT时，如果列存在默认值，则会使用其默认值；如果列不存在默认值，则自动使用NULL。

```
--使用DEFAUT插入数据
insert into dept values(60, 'MARKET',DEFAULT);
--查询deptno=60的部门信息
select * from dept where deptno = 60;

```

可以看到LOC列没有默认值，DEFAULT自动使用了NULL空值；

批量插入数据

可以使用SELECT语句替换原来的VALUES子句，这样由SELECT语句提供添加的数值，通过INSERT向表中添加一组数据。其语法格式如下：

```
INSERT INTO table_name [(column_name1[,column_name2]…)] selectSubquery

```

- table_name：表示要插入的表名称。
- column_name1和column_name2：表示指定的列名。
- selectSubquery：任何合法的SELECT语句，其所选列的个数和类型要与语句中的column对应。

```
--创建一个与EMP表结构一样的EMP_TEMP表
create table EMP_TEMP
(
  empno    NUMBER(4) not null,
  ename    VARCHAR2(10),
  job      VARCHAR2(9),
  mgr      NUMBER(4),
  hiredate DATE,
  sal      NUMBER(7,2),
  comm     NUMBER(7,2),
  deptno   NUMBER(2)
)
--将emp表中sal大于等于3000的数据插入emp_temp表
insert into emp_temp select * from emp where sal >= 3000

```

INSERT INTO子句指定的列名可以与SELECT子句指定的列名不同，但它们之间的数据类型必须是兼容的，即SELECT语句返回的数据必须满足INSERT INTO表中列的约束。

更新数据（UPDATE语句）

在更新数据时，更新的列数可以由用户自己指定，列与列之间用逗号(,)分隔；更新的条数可以通过WHERE子句来加以限制，使用WHERE子句时，系统只更新符合WHERE条件的记录信息。UPDATE语句的语法格式如下：

```
UPDATE table_name SET {column_name1=express1[,column_name2=express2...] | (column_name1
[,column_name2…])=(selectSubquery)} [WHERE condition]

```

- table_name：表示要修改的表名。
- column_name1和column_name2：表示指定要更新的列名。
- selectSubquery：任何合法的SELECT语句，其所选列的个数和类型要与语句中的column对应。
- condition：筛选条件表达式，只有符合筛选条件的记录才被更新。

使用UPDATE语句有以下注意事项：

- 更新数字列时，可以直接提供数字值，或者用单引号引住。
- 更新字符列或日期列时，必须用单引号引住。
- 更新数据时，数据必须要满足约束规则。
- 更新数据时，数据必须与列的数据类型匹配。

更新单列数据

当更新单列数据时，SET子句后只需要提供一个列。

```
--将emp表中员工名称ename为SCOTT的工资sal调整为6000
update emp set sal = 6000 where ename='SCOTT';

```

更新多列数据

当修改多列时，列之间用逗号分开。

```
--将emp表中员工名称ename为JONES的工资sal上调20%，部门编号deptno调整为30
update emp set sal = sal*1.2,deptno=30 where ename='JONES';

```

更新日期列数据

当更新日期列数据时，数据格式要与日期格式和日期语言匹配，否则会显示错误信息，可以使用TO_DATE函数进行日期格式转换；

```
--将员工编号7788的入职时间hiredate修改为1986/01/01
update emp set hiredate = TO_DATE('1986/01/01', 'YYYY/MM/DD') where empno=7788;

```

使用DEFAULT选项更新数据

可以使用DEFAULT选项提供的数据来更新数据。使用此方式时，如果列存在默认值，则会使用默认值更新数据；如果列不存在默认值，则使用NULL。

```
--更新员工姓名为SCOTT的岗位为默认值
update emp set job = DEFAULT where ename = 'SCOTT';

```

使用子查询更新数据

UPDATE语句也可以与SELECT语句组合使用来达到更新数据的目的。

```
--将工资sal低于2000的员工工资调整为管理者的平均工资水平
update emp set sal = (select avg(sal) from emp where job = 'MANAGER')
where sal < 2000;

```

注意：在将UPDATE语句与SELECT语句组合使用时，必须保证SELECT语句返回单一的值，否则会出现错误提示，导致更新数据失败。

删除数据（DELETE语句和TRUNCATE语句）

从数据库中删除记录可以使用DELETE语句和TRUNCATE语句，但这两种语句还是有很大区别的，下面分别进行讲解。

DELETE语句

DELETE语句用来删除数据库中的所有记录和指定范围的记录。

```
DELETE FROM table_name [WHERE condition]

```

- table_name：表示要删除记录的表名。
- condition：筛选条件表达式，是一个可选项。当该筛选条件存在时，只有符合筛选条件的记录才会被删除。

```
--删除员工姓名ename为SCOTT的员工信息
delete from emp where ename='SCOTT';

--删除emp表中的所有数据
delete from emp;

```

使用DELETE语句删除数据时，Oracle系统会产生回滚记录，所以这种操作可以使用ROLLBACK语句来撤销。

TRUNCATE语句

如果确定要删除表中的所有记录，Oracle建议使用TRUNCATE语句。使用TRUNCATE语句删除表中的所有记录要比DELETE语句快得多，这是因为**使用TRUNCATE语句删除数据时，它不会产生回滚记录**。当然，执行了TRUNCATE语句的操作也就**无法使用ROLLBACK语句撤销**。

```
--删除emp_temp表的所有数据
truncate table emp_temp;

```

在TRUNCATE语句中还可以使用REUSE STORAGE关键字或DROP STORAGE关键字，前者表示删除记录后仍然保存记录所占用的空间，后者表示删除记录后立即回收记录占用的空间。默认情况下TRUNCATE语句使用DROP STORAGE关键字。

常用系统函数

Oracle字符型函数

函数	说明	案例
ASCII(X)	求字符X的ASCII码	select ASCII('A') FROM DUAL;
CHR(X)	求ASCII码对应的字符	select CHR(65) FROM DUAL;
LENGTH(X)	**求字符串X的长度**	**select LENGTH('ORACLE数据库')from DUAL;**
CONCATA(X,Y)	**返回连接两个字符串X和Y的结果**	**select CONCAT('ORACLE','数据库') from DUAL;**
INSTR(X,Y[,START])	查找字符串X中字符串Y的位置，可以指定从Start位置开始搜索，不填默认从头开始	SELECT INSTR('ORACLE数据库','数据') FROM DUAL;
LOWER(X)	把字符串X中大写字母转换为小写	SELECT LOWER('ORACLE数据库') FROM DUAL;
UPPER(X)	把字符串X中小写字母转换为大写	SELECT UPPER('Oracle数据库') FROM DUAL;
INITCAP(X)	把字符串X中所有单词首字母转换为大写，其余小写。	SELECT INITCAP('ORACLE and mysql ') FROM DUAL;
LTRIM(X[,Y])	去掉字符串X左边的Y字符串，Y不填时，默认的是字符串X左边去空格	SELECT LTRIM('--ORACLE数据库','-') FROM DUAL;
RTRIM(X[,Y])	去掉字符串X右边的Y字符串，Y不填时，默认的是字符串X右边去空格	SELECT RTRIM('ORACLE数据库--','-') FROM DUAL;
TRIM(X[,Y])	去掉字符串X两边的Y字符串，Y不填时，默认的是字符串X左右去空格	SELECT TRIM('--ORACLE数据库--','-') FROM DUAL;
REPLACE(X,old,new)	**查找字符串X中old字符，并利用new字符替换**	**select replace('ORACLE数据库','ORACLE','关系型') fromdual;**
SUBSTR(X,start[,length])	**截取字符串X，从start位置（其中start是从1开始）开始截取长度为length的字符串，length不填默认为截取到字符串X末尾**	**SELECT SUBSTR('ORACLE数据库',1,6) FROM DUAL;**
RPAD(X,length[,Y])	对字符串X进行右补字符Y使字符串长度达到length长度	SELECT RPAD('ORACLE',9,'-') from DUAL;
LPAD(X,length[,Y])	对字符串X进行左补字符Y使字符串长度达到length长度	SELECT LPAD('ORACLE',9,'-') from DUAL;

Oracle日期函数

1、SYSDATE函数：该函数没有参数，可以得到系统的当前时间。

```
select sysdate from dual;
```

2、SYSTIMESTAMP函数：该函数没有参数，可以得到系统的当前时间，该时间包含时区信息，精确到微秒。

```
select systimestamp from dual;
```

3、DBTIMEZONE函数：该函数没有输入参数，返回数据库时区。

```
select dbtimezone from dual;
```

4、to_char函数和to_date函数：转换日期时间格式

```
select to_char(sysdate,'yyyy-mm-dd hh24:mi:ss') from dual; --日期转指定格式的字符串
select to_date('2019-11-12','yyyy-mm-dd') from dual;  --字符串转日期格式
```

5、ADD_MONTHS（r,n）函数：该函数返回在指定日期r上加上月份数n后的日期。其中

r：指定的日期。n：要增加的月份数，如果N为负数，则表示减去的月份数。

```
select to_char(add_months(sysdate,2),'yyyy-mm-dd') from dual;
```

6、LAST_DAY(r)函数：返回指定r日期的当前月份的最后一天日期。

```
select to_char(last_day(sysdate),'yyyy-mm-dd') from dual;
```

7、NEXT_DAY(r,c)函数：返回指定R日期的后一周的与r日期字符（c：表示星期几,1：周天，2：周1）对应的日期。

```
select to_char(next_day(sysdate,1),'yyyy-mm-dd') from dual;
```

8、EXTRACT（time）函数：返回指定time时间当中的年、月、日、分等日期部分。

```
select
extract(year from sysdate) as year0,
extract(year from timestamp '2018-11-12 15:36:01') as year,
extract(month from timestamp '2018-11-12 15:36:01') as month,
extract(day from timestamp '2018-11-12 15:36:01') as day,
extract(hour from timestamp '2018-11-12 15:36:01') as hour,
extract(minute from timestamp '2018-11-12 15:36:01') as minute,
extract(second from timestamp '2018-11-12 15:36:01') as second
from dual;
```

9、substr(to_char(）)--截取年、月、日、时、分、秒

```
    select
    substr(to_char(sysdate,'yyyy-mm-dd  hh24:mi:ss'),1,4) s_year ,
    substr(to_char(sysdate,'yyyy-mm-dd  hh24:mi:ss'),6,2) s_month,
    substr(to_char(sysdate,'yyyy-mm-dd  hh24:mi:ss'),9,2) s_day,
    substr(to_char(sysdate,'yyyy-mm-dd  hh24:mi:ss'),13,2) s_hour,
    substr(to_char(sysdate,'yyyy-mm-dd  hh24:mi:ss'),16,2) s_minute,
    substr(to_char(sysdate,'yyyy-mm-dd  hh24:mi:ss'),19,2) s_second
    from dual;
```

10、MONTHS_BETWEEN(r1,r2)函数：该函数返回r1日期和r2日期直接的月份。当r1>r2时，返回的是正数，假如r1和r2是不同月的同一天，则返回的是整数，否则返回的小数。当r1<r2时，返回的是负数

```
select months_between(to_date('2019-10-1','yyyy-mm-dd'),to_date('2019-8-5','yyyy-mm-dd')) from dual;
11、ROUND（r[,f]）函数：将日期r按f的格式进行四舍五入。如果f不填，则四舍五入到最近的一天。
select sysdate, --当前时间
       round(sysdate, 'yyyy') as year, --按年
       round(sysdate, 'mm') as month, --按月
       round(sysdate, 'dd') as day, --按天
       round(sysdate) as mr_day, --默认不填按天
       round(sysdate, 'hh24') as hour --按小时
  from dual;
12、TRUNC（r[,f]）函数：将日期r按f的格式进行截取。如果f不填，则截取到当前的日期。
select sysdate, --当前时间
       trunc(sysdate, 'yyyy') as year, --按年
       trunc(sysdate, 'mm') as month, --按月
       trunc(sysdate, 'dd') as day, --按天
       trunc(sysdate) as mr_day, --默认不填按天
       trunc(sysdate, 'hh24') as hour --按小时
  from dual;
```

Oracle数值型函数

函数	解释	案例
ABS(X)	求数值X的绝对值	select abs(-5) from dual;
COS(X)	求数值X的余弦	select cos(0.5) from dual;
ACOS(X)	求数值X的反余弦	select acos(0.5) from dual;
CEIL(X)	求大于或等于数值X的最小整数	select ceil(5.5) from dual;
FLOOR(X)	求小于或等于数值X的最大整数	select floor(5.5) from dual;
log(x,y)	求x为底y的对数	select log(2,32) from dual;
mod(x,y)	求x除以y的余数	select mod(15,10) from dual;
power(x,y)	求x的y次幂	select power(2,5) from dual;
sqrt(x)	求x的平方根	select sqrt(25) from dual;
round(x[,y])	求数值x在y位进行四舍五入。 y不填时，默认为y=0; 当y>0时，是四舍五入到小数点右边y位。 当y<0时，是四舍五入到小数点左边|y|位。	select round(5.615, 2), round(5.615), round(6.45,-1) from dual;
trunc(x[,y])	求数值x在y位进行直接截取 y不填时，默认为y=0; 当y>0时，是截取到小数点右边y位。 当y<0时，是截取到小数点左边|y|位。	select trunc(5.615, 2), trunc(5.615), trunc(5.615,-1) from dual;

Oracle转换函数

1、to_char()函数：将DATE或者NUMBER转换为字符串

2、 to_date()函数：将number、char转换为date

3、 to_number()函数：将char转换为number

4、CAST(expr AS type_name)函数：用于将一个内置数据类型或集合类型转变为另一个内置数据类型或集合类型。expr为列名或值，type_name数据类型。

SELECT CAST('123.4' AS int) from dual; 结果： 123 可进行四舍五入操作： SELECT CAST('123.447654' AS decimal(5,2)) as result from dual;decimal(5,2)表示值总位数为5，精确到小数点后2位。结果：123.45

5、TO_MULTI_BYTE(c1)函数：将字符串c1中的半角转化为全角。TO_MULTI_BYTE和TO_SINGLE_BYTE是相反的两个函数.

```
select to_multi_byte('高A') text from dual;
```

6、to_single_byte(c1)函数：将字符串c1中的全角转化为半角。

```
select to_single_byte('高Ａ') text from dual;
```

7、TIMESTAMP_TO_SCN(timestamp)函数：用于根据输入的timestamp返回所对应的scn值，其中timestamp用于指定日期时间。作为对于闪回操作(flashback)的一个增强，Oracle10g提供了函数对于SCN和时间戳进行相互转换。

```
select timestamp_to_scn(sysdate) scn from dual;
```

结果：2138728

8、SCN_TO_TIMESTAMP(number)函数：根据输入的scn值返回对应的大概日期时间，其中number用于指定scn值.

```
select to_char(scn_to_timestamp(2138728), 'yyyy-mm-dd hh24:mi:ss') from dual;
```

结果：2024-06-20 17:28:09

9、CONVERT(string,dest_set[,source_set])函数：将字符串string从source_set所表示的字符集转换为由dest_set所表示的字符集.如果source_set没有被指定,它缺省的被设置为数据库的字符集.
ZHS16GBK表示采用GBK编码格式、16位（两个字节）简体中文字符集
WE8ISO8859P1(西欧、8位、ISO标准8859P1编码)
AL32UTF8（其中AL代表ALL,指适用于所有语言）、zhs16cgb231280

```
select convert('中国','US7ASCII','WE8ISO8859P1') "conversion" from dual;
```

结果：0??u

10、TRANSLATE(str1 USING zfj)函数：将字符串转变为数据库字符集(char_cs)或民族字符集(nchar_cs)

```
Select TRANSLATE('中国' using nchar_cs) from dual;
```

结果：中国

11、ASCIISTR(s)函数：将任意字符集的字符串转变为数据库字符集的ASCII字符串。

```
Select ASCIISTR ('1A_中文')  from dual;
```

结果：1A_\4E2D\6587

12、UNISTR(str1)函数：输入字符串返回相应的UNICODE字符

```
Select UNISTR ('\4E2D'),UNISTR ('\6587'),UNISTR ('\0300'),UNISTR ('\00E0') from dual;
```

结果：中 文 ` à

13、COMPOSE(string)函数：这个函数以UNICODE字符串为参数，返回一个规范化的字符串。比如，它可以接受一个字母和一个组合标记，比如说'a'（Unicode 字符0097）和沉音符（Unicode 字符0300），然后创建一个单独的由两个标记组合而成的字符(à)。

```
Select COMPOSE('a'||unistr('\0300'))  from dual;
```

结果：à

14、DECOMPOSE(string)函数：返回一个Unicode字符串。它是string的规范分解。

```
SELECT DECOMPOSE ('Châteaux') FROM DUAL;
```

结果：Cha^teaux

15、CHARTOROWID(c1)函数：将字符数据类型CHAR或VARCHAR2转换为ROWID值.参数c1是长度为18的字符串，必须符合rowid格式.CHARTOROWID是ROWIDTOCHAR的反函数.

在Oracle中，每一条记录都有一个rowid，rowid在整个数据库中是唯一的，rowid确定了每条记录是在Oracle中的哪一个数据文件、块、行上。在重复的记录中，可能所有列的内容都相同，但rowid不会相同.

```
SELECT chartorowid('AAAADeAABAAAAZSAAA') FROM DUAL;
```

结果：AAAADeAABAAAAZSAAA

16、ROWIDTOCHAR(rowid)函数：转换rowid值为varchar2类型，rowid固定参数，返回长度为18的字符串。

```
SELECT ROWIDTOCHAR(rowid) FROM DUAL;
```

结果：AAAAB0AABAAAAOhAAA

17、INTERVAL 'integer [- integer]' {YEAR | MONTH} [(precision)][TO {YEAR | MONTH}]函数：该数据类型常用来表示一段时间差, 注意时间差只精确到年和月. precision为年或月的精确域, 有效范围是0到9, 默认值为2.

```
select INTERVAL '123-2' YEAR(3) TO MONTH from dual;
```

表示: 123年2个月, "YEAR(3)" 表示年的精度为3, 可见"123"刚好为3为有效数值, 如果该处YEAR(n), n<3就会出错, 注意默认是2

结果：+123-02

18、HEXTORAW(string)函数：将string一个十六进制构成的字符串转换为二进制RAW数值.String中的每两个字符表示了结果RAW中的一个字节.HEXTORAW和RAWTOHEX为相反的两个函数.当出现比f大的字母时(以a最小z最大)就会报错

```
select hextoraw('abcdef') from dual;
```

结果：ABCDEF

19、RAWTOHEX(rawvalue)函数：将raw串转换为十六进制. rawvalue中的每个字节都被转换为一个双字节的字符串.

```
select rawtohex('AA') from dual;
```

结果：4141 结果之所以是4141是因为A的ASCII为65，65转换为十六进制就是41。

20、TO_LOB (long_column)函数：将LONG或LONG ROW列的数据转变为相应的LOB类型。但需要注意的是，TO_LOB一般只用在CREATE TABLE或INSERT TABLE语句后面的子查询中。在其他地方使用会报错，比如UPDATE语句。

Oracle聚合函数

函数	说明	备注
count(col)	计数	
sum(col)	求和	
avg(col)	均值	
min(col)/max(col)	最小/最大值	

Oracle分析(窗口)函数

分析(窗口)函数是Oracle专门用于解决复杂报表统计需求的功能强大的函数。
它可以在数据中进行分组然后计算基于组的某种统计值，并且每一组的每一行都可以返回一个统计值。

分析函数语法

```
function_name(<argument>,<argument>...)
over(<partition_Clause>
      <order by_Clause>
      <windowing_Clause>);
```

- function_name()：函数名称
- argument：参数
- over( )：开窗函数
- partition_Clause：分区子句，数据记录集分组，group by...
- order by_Clause：排序子句，数据记录集排序，order by...
- windowing_Clause：开窗子句，定义分析函数在操作行的集合，三种开窗方式：rows、range、Specifying
- 注：使用开窗子句时一定要有排序子句！！！

聚合、累计类型分析函数使用

```
--(1)count(...) over(...) ;求各部门员工数
select
distinct e.deptno deptno,
d.dname dname,
count(*)over(partition by e.deptno,d.dname) totar
from emp e join dept d on e.deptno=d.deptno;

--(2)sum(...) over(...);求各部门员工递加的工资总和
select
ename,
deptno,
sum(sal) over(partition by deptno order by ename) sum_sal
from emp;

--(3)avg(...) over(...);求各部门的平均工资
select
distinct deptno,
avg(sal)over(partition by deptno) avg_sal
from emp;

--（4）min(...)/max(...)over(...);求各职位的最低和最高薪资
select
distinct job,
min(sal)over(partition by job) min_sal,
max(sal)over(partition by job) max_sal
from emp;
```

排名类型分析函数使用

```
--(1)、整体排名：rank()/row_number()/dense_rank() over(...);按照薪资降序整体排名
select emp.* ,
rank()over(order by sal desc) rank, --占空排名，跳跃排名，如 1-2-2-4-5
row_number()over(order by sal desc) row_number,--顺序递增（减）排名，如 1-2-3-4-5
dense_rank()over(order by sal desc) dense_rank --不占空排序，如1-2-2-3-4
from emp;

--(2)、组内排名：rank()/row_number()/dense_rank() over(...);按照各部门内部薪资降序排名
select emp.* ,
rank()over(partition by deptno order by sal desc) rank, --占空排名，跳跃排名，如 1-2-2-4-5
row_number()over(partition by deptno order by sal desc) row_number,--顺序递增（减）排名，如 1-2-3-4-5
dense_rank()over(partition by deptno order by sal desc) dense_rank --不占空排序，如1-2-2-3-4
from emp;
```

还有更多分析函数，使用过程中再去查找使用吧！！！

Oracle行转列

case when语法结构

```
case 列名
when 条件值1 then 选项1
when 条件值2 then 选项2
.......
else    默认值 end
```

案例

```
--  Oracle 行列转换
    -- 数据，使用fox用户的emp表数据
    select * from emp t where rownum<=50;
    -- 表字段说明：
    -- empno,    ename,   job,  mgr,     hiredate, sal, comm, deptno
    -- 员工编号，员工姓名，岗位，领导编号，入职日期，薪资，奖金，部门编号

-- 需求：查询emp表中，每个部门的人数，并进行行转列显示
select
max(case when deptno=10
    then total else 0 end) "10",
max(case when deptno=20
    then total else 0 end) "20",
max(case when deptno=30
    then total else 0 end) "30"
from
  (select
      deptno,
      count(*) total
  from emp
  group by deptno
  );
```

Oracle数据合并（存在则更新，不存在就插入）

```
-- Merge into
MERGE INTO table_name alias1
USING  table|view|sub_query alias2
ON (join condition)
WHEN MATCHED THEN
  UPDATE SET
  col1 = col_val1,
  col2 = col2_val
WHEN NOT MATCHED THEN
  INSERT (column_list)
  VALUES (column_values);
```

案例

```
---创建表
create table test_merge_A(
empno number,
ename varchar2(30),
sal number);
---创建表
create table test_merge_B(
empno number,
ename varchar2(30),
sal number);

------插入数据
insert into test_merge_A values(1122,'AA',1500);
insert into test_merge_A values(1133,'BB',1600);
insert into test_merge_A values(1144,'CC',1700);
insert into test_merge_A values(1155,'DD',1800);

insert into test_merge_B values(1144,'DD',2500);
insert into test_merge_B values(1166,'EE',5000);
insert into test_merge_B values(1177,'FF',3000);
insert into test_merge_B values(1122,'AA',3000);
-----查看数据
select * from test_merge_A;
select * from test_merge_B;

--将B表的数据合并到A表，如果存在则更新，如果不存在则插入
merge into test_merge_A a
using test_merge_B b
on (a.empno=b.empno)
when matched then
update set a.ename=b.ename,a.sal=b.sal
when not matched then
insert (a.empno,a.ename,a.sal)
  values (b.empno,b.ename,b.sal);

-----查看数据
select * from test_merge_A;
```

Oracle递归用法

递归语法:

```
select
  [level],
  column,
  expr,...
from table
[where condition]
start with condition
connect by [prior column1 = column2   向下
          |column1 = prior  column2];  向上
```

- level:伪列。用于返回层次查询的层次(1:根行 2:第2级行 3:第3级行...)
- start with:用于指定层次关系查询的根行； 决定了爬树的起点
- connect by:用于指定父行和子行的关系 当定义父行和子行的关系时,必须使用prior关键字，决定了爬树的方向:
- prior:用于指定哪个是父级列

案例

```sql
-- Oracle 递归查找
  -- 数据，使用fox用户的emp表数据
  select * from emp t
  where rownum<=50;
  -- 表字段说明：
  -- empno,ename,job, mgr, hiredate, sal, comm, deptno
  -- 员工编号，员工姓名，岗位，领导编号，入职日期，薪资，奖金，部门编号

-- 需求1：向上递归查出empno=7369的所有上级领导的编号和姓名
   -- 7369-->7902-->7566--> 7839
select
  empno,
  ename
from emp
start with empno = 7369
connect by empno = prior mgr;

-- 需求2：向下递归查出7839的所有‘10’部门的下属及层级
select
  empno,
  ename,
  deptno
from emp
where deptno = 10
start with empno = 7839
connect by prior empno = mgr;
```

3.3 PL/SQL编程

PL/SQL(procedural language/SQL)是Oracle在数据库中引入的一种过程化编程语言。PL/SQL构建于SQL之上，可以用来编写包含SQL语句的程序。

PL/SQL是一种过程化语言，在PL/SQL中可以通过IF语句或LOOP语句控制程序的执行流程，甚至可以定义变量，在语句之间传递数据信息，从而操控程序处理的细节过程，以实现比较复杂的业务逻辑。

PL/SQL是Oracle的专用语言，是对标准SQL语言的扩展，它允许在其内部嵌套普通的SQL语句，因此能将SQL语句的数据操纵能力、数据查询能力和PL/SQL的过程处理能力结合在一起，达到取长补短的目的。

PL/SQL块结构

PL/SQL程序以块(block)为基本单位，整个PL/SQL块分为3部分，即声明部分（用DECLARE开头）、执行部分（以BEGIN开头）和异常处理部分（以EXCEPTION开头）。其中，执行部分是必需的，其他两个部分可选。

标准PL/SQL块的语法格式如下：

```
[DECLARE]
--声明部分，可选
BEGIN
--执行部分，必需
[EXCEPTION]
--异常处理部分，可选
END
```

1. 声明部分

由关键字**DECLARE开始，到BEGIN关键字结束**。在这里可以声明PL/SQL程序块中用到的**变量、常量和游标**等。注意，在某个PL/SQL块中声明的内容只能在当前块中使用，而在其他PL/SQL块中是无法被引用的。

2. 执行部分

以关键字BEGIN开始，它的结束方式通常有两种：如果PL/SQL块中的代码在运行时出现异常，则执行完异常处理部分的代码就结束；如果没有使用异常处理或PL/SQL块未出现异常，则以关键字END结束。**执行部分是整个PL/SQL块的主体，主要的逻辑控制和运算都在这部分被完成**，所以在执行部分可以包含多个PL/SQL语句和SQL语句。

3. 异常处理部分

以关键字EXCEPTION开始，在该关键字所包含的代码被执行完毕后，整个PL/SQL块也将结束。在执行PL/SQL代码（主要是执行部分）的过程中，可能会产生一些意想不到的错误，如除数为零、空值参与运算等，这些错误都会导致程序被中断运行，可以在异常处理部分通过编写一定量的代码来纠正错误或者给用户提供一些错误信息提示，甚至是将各种数据操作回退到异常产生之前的状态，以备重新运行代码块。另外，对于可能出现的多种异常情况，可以使用WHEN THEN语句来实现多分支判断，然后在每个分支下通过编写代码来处理相应的异常。

对于PL/SQL块中的语句，**每一条PL/SQL语句都必须以分号结束，而每条PL/SQL语句均可以被写成多行的形式，同样必须使用分号来结束**；另外，**一行中也可以有多条PL/SQL语句，但是它们之间必须以分号分隔。**接下来通过一个简单的例子来看PL/SQL块的完整应用。

定义一个PL/SQL块，计算两个整数的和与这两个整数的差的商，代码如下（SQL * Plus中运行代码）：

```
--实现在服务端显示执行结果
set serveroutput on
--PL/SQL块
declare
  a int:=100;
  b int:=200;
  c number;
begin
  c:=(a+b)/(a-b);
  dbms_output.put_line(c);
exception
  when zero_divide then
  dbms_output.put_line('除数不允许为零! ');
end;
/
```

代码注释和标识符

注释用于对程序代码进行解释说明，它能够增强程序的可读性，使程序更易于理解。注释编译时会被PL/SQL编译器忽略掉，注释有**单行注释**和**多行注释**两种情况。

单行注释

单行注释由两个连接字符“--”开始，后面紧跟着注释内容。

```
set serveroutpu on

declare
Num_sal number;           --声明一个数值变量
Var_ename varchar2(20);   --声明一个字符串变量
begin
select ename,sal into Var_ename,Num_sal from emp  --检索指定的值并存储到变量中
where empno=7369;
dbms_output.put_line(Var_ename||'的工资是'||Num_sal);  --输出变量中值
end;
/
```

注意：如果注释超过一行，就必须在每一行的开头使用连接字符(--)。

多行注释

多行注释由“/*”开头，以“*/”结尾，这种多行注释的方法在大多数的编程语言中是相同的。

```
set serveroutpu on

declare
Num_sal number;           /*声明一个数值变量*/
Var_ename varchar2(20);   /*声明一个字符串变量*/
begin
/*检索指定的值并存储到变量中*/
select ename,sal into Var_ename,Num_sal from emp
where empno=7369;
/*输出变量中值*/
dbms_output.put_line(Var_ename||'的工资是'||Num_sal);
end;
```

标识符

标识符(identifier)用于定义PL/SQL块和程序单元的名称；通过使用标识符，可以定义常量、变量、异常、显式游标、游标变量、参数、子程序以及包的名称。当使用标识符定义PL/SQL块或程序单元时，需要满足以下规则：

- 当定义变量、常量时，每行只能定义一个变量或者常量（行终止符为“;”）
- 当定义变量、常量时，名称必须以英文字符(A～Z、a～z)开始，并且最大长度为30个字符。如果以其他字符开始，那么必须使用双引号引住。
- 当定义变量、常量时，名称只能使用字符A～Z、a～z、0～9以及符号_、$和#。如果使用其他字符，那么必须用双引号引住。
- 当定义变量、常量时，名称不能使用Oracle关键字。例如，不能使用SELECT、UPDATE等作为变量名。如果要使用Oracle关键字定义变量、常量，那么必须使用双引号引住。

所有的PL/SQL程序元素（如关键字、变量名、常量名等）都是由一些字符序列组合而成的，而这些字符序列中的字符都必须取自PL/SQL所允许使用的字符集，这些合法的字符集主要包括以下内容：

- 大写和小写字母：A～Z或a～z。
- 数字：0～9。
- 非显示的字符：制表符、空格和按Enter键。
- 数学符号：+、-、*、/、>、<、=等。

- 间隔符：()、{}、[]、?、!、;、:、@、#、%、$、&等。

除了由引号引起来的字符串，PL/SQL不区分字母的大小写。标准PL/SQL字符集是ASCII字符集的一部分，ASCII是一个单字节字符集，这就是说每个字符可以表示为一个字节的数据，该性质将字符总数限制在最多为256个。

文本

文本是指实际数值的文字，包括数字文本、字符文本、布尔文本、日期时间文本、字符串文本等。

- 数字文本：整数或者浮点数。编写PL/SQL代码时，可以使用科学记数法和幂操作符(**)表示，如100、2.45、3e3、5E6、6*\10**3等。科学记数法和幂操作符只适用于PL/SQL语句，而不适用于SQL语句。
- 字符文本：用单引号引住的单个字符。这些字符可以是PL/SQL支持的所有可打印字符，包括英文字符(A～Z、a～z)、数字字符(0～9)及其他字符（<、>等），如'A'、'9'、'<'、' '、'%'等。
- 布尔文本：通常指BOLLEAN值（TRUE、FALSE和NULL），主要用在条件表达式中。
- 日期时间文本：指日期时间值。日期文本必须用单引号引住，并且日期值必须与日期格式和日期语言匹配，如'10-NOV-91'、'1997-10-2213:01:01' 、'09-10-月-03'等。
- 字符串文本：由两个或两个以上字符组成的多个字符值。字符串文本必须用单引号引住，如'Hello World'、'$9600'、'10-NOV-91'等。

在Oracle Database 10g之前，如果字符串文本包含单引号，必须使用两个单引号表示。例如，要为某个变量赋值“I’m a string,you’re a string.”，字符串文本必须要采用以下格式。

```
string_var:= 'I''m a string,you''re a string.';
```

在Oracle Database 10g之后，如果字符串文本中包含单引号，既可以使用原有格式赋值，也可以使用其他分隔符（[]、{}、<>等）赋值。
注意使用分隔符[]、{}、<>为字符串赋值时，不仅需要在分隔符前后加单引号，而且需要带有前缀q。
例如：

```
string_var:= q'[I'm a string,you're a string.]';
```

基本数据类型

数据类型本质上是一种描述数据存储的内存结构，用它来决定变量中所存储数据的类型。而变量本质上是一种用名称进行识别的标识符号，可以存储不同类型的数据。根据不同的数据类型，定义不同名称的变量，这样就可以存储不同类型的数据。变量在程序运行的过程中，其值可以发生变化。与变量对应的就是常量，常量就是指在程序运行的过程中，其值不会发生变化的量。

数组类型

数值类型包括NUMBER、BINARY_INTEGER和PLS_INTEGER 3种基本类型。

- NUMBER类型的变量可以存储整数或浮点数；
- BINARY_INTEGER和PLS_INTEGER类型的变量只能存储整数。

NUMBER类型还可以通过NUMBER(p,s)的形式来格式化数字，其中，参数p表示精度，参数s表示刻度范围。精度是指数值中所有有效数字的个数，而刻度范围是指小数点右边小数位的个数，在这里精度和刻度范围都是可选的。

```
--声明一个精度为8且刻度范围为3的标识金额的变量Num_Money
Num_Money NUMBER(8,3);
```

PL/SQL子类型，是与NUMBER类型等价的类型别名，甚至可以说是NUMBER类型的多种重命名形式；这些等价的子类型主要包括DEC、DECIMAL、DOUBLE、INTEGER、INT、NUMERIC、SMALLINT、BINARY_INTEGER、PLS_INTEGER等。

字符类型

字符类型主要包括VARCHAR2、CHAR、LONG、NCHAR和NVARCHAR2，用来存储字符串或字符数据。

- VARCHAR2类型：PL/SQL中的VARCHAR2类型和数据库类型中的VARCHAR2比较类似，用于存储可变长度的字符串，其语法格式如下：

```
VARCHAR2(maxlength)
```

参数maxlength表示可存储字符串的最大长度，定义变量时必须给出（因为VARCHAR2类型没有默认的最大长度），最大值是32 767字节。

注意：数据库类型中的VARCHAR2的最大长度是4000字节，所以一个长度大于4000字节的PL/SQL中的类型VARCHAR2变量不可以赋值给数据库中的一个VARCHAR2变量，而只能赋值给LONG类型的数据库变量。

- CHAR类型：表示指定长度的字符串，其语法格式如下：

```
CHAR(maxlength)
```

CHAR类型的默认最大长度为1。与VARCHAR2不同，maxlength可以不指定，默认为1。如果赋给CHAR类型的值不足maxlength，则会在其后面用空格补全，这也是不同于VARCHAR2的地方。

注意：数据库类型中的CHAR只有2000字节，所以如果PL/SQL中的CHAR类型变量的长度大于2000字节，则不能赋给数据库中的CHAR。

- LONG类型：表示可变的字符串，最大长度是32 767字节。数据库类型中的LONG最大长度可达2 GB，所以几乎任何字符串变量都可以赋值给它。
- NCHAR和NVARCHAR2类型：PL/SQL 8.0以后加入的类型，它们的长度要根据各国字符集来确定，只能具体情况具体分析。

日期类型

日期类型只有一种，即DATE类型，用来存储日期和时间信息。DATE类型的存储空间是7字节，分别使用1字节存储世纪、年、月、天、小时、分钟和秒。

布尔类型

布尔类型也只有一种，即BOOLEAN类型，主要用于程序的流程控制和业务逻辑判断，其变量值可以是TRUE、FALSE或NULL中的一种。

特殊数据类型

%TYPE类型

使用%TYPE关键字可以声明一个与指定列相同的数据类型，它通常紧跟在指定列名的后面。

```
--声明一个与emp表中job列的数据类型完全相同的变量var_job
declare
var_job emp.job%type;
```

使用%TYPE定义变量有两个好处：

- 用户不必查看表中各个列的数据类型，就可以确保所定义的变量能够存储检索的数据；
- 如果对表中已有列的数据类型进行修改，则用户不必考虑对已定义的变量所使用的数据类型进行更改，因为%TYPE类型的变量会根据列的实际类型自动调整自身的数据类型。

使用%TYPE类型的变量输出emp表中编号为7369的员工名称和职务信息，代码如下:

```
set serveroutput on

declare
  var_ename emp.ename%type;    --声明与ename列类型相同的变量
  var_job emp.job%type;        --声明与job列类型相同的变量
begin
  select ename,job
  into var_ename,var_job
  from emp
  where empno=7369;          --检索数据，并保存在变量中
dbms_output.put_line(var_ename||'的职务是'||var_job);  --输出变量的值
end;
/
```

注意：由于INTO子句中的变量只能存储一个单独的值，因此要求SELECT子句只能返回一行数据，这个由WHERE子句进行了限定。若SELECT子句返回多行数据，则代码运行后会返回错误信息。

在上述代码中使用了INTO子句，它位于SELECT子句的后面，用于设置将从数据库中检索的数据存储到哪个变量中。

RECORD类型

注意：由于INTO子句中的变量只能存储一个单独的值，因此要求SELECT子句只能返回一行数据，这个由WHERE子句进行了限定。若SELECT子句返回多行数据，则代码运行后会返回错误信息。

在上述代码中使用了INTO子句，它位于SELECT子句的后面，用于设置将从数据库中检索的数据存储到哪个变量中。

```
TYPE record_type IS RECORD
(
var_member1 data_type [not null] [:=default_value],
…
var_membern data_type [not null] [:=default_value])
```

record_type：表示要定义的记录类型名称。

- var_member1：表示该记录类型的成员变量名称。
- data_type：表示成员变量的数据类型。

从上述语法结构中可以看出，记录类型的声明类似于C或C++中的结构类型，并且成员变量的声明与普通PL/SQL变量的声明相同。

声明一个记录类型emp_type，然后使用该类型的变量存储emp表中的一条记录信息，并输出这条记录信息，代码如下:

```
declare
  type emp_type is record  --声明RECORD类型emp_type
  (
    var_ename varchar2(20),  --定义字段/成员变量
    var_job varchar2(20),
    var_sal number
   );
  empinfo  emp_type;    --定义变量
begin
  select ename,job,sal
  into empinfo
  from emp
  where empno=7369;  --检索数据
  /*输出员工信息*/
  dbms_output.put_line('员工'||empinfo.var_ename||'的职务是'||empinfo.var_job||'、工资是'||empinfo. var_sal);
end;
/
```

%ROWTYPE类型

%ROWTYPE类型的变量结合了“%TYPE类型”和“记录类型”变量的优点，它可以根据数据表中行的结构定义一种特殊的数据类型，用来存储从数据表中检索到的一行数据。它的语法格式如下：

```
rowVar_name table_name%ROWTYPE;
```

- rowVar_name：表示可以存储一行数据的变量名。
- table_name：指定的表名。

首先声明一个%ROWTYPE类型的变量rowVar_emp，然后使用该变量存储emp表中的一行数据，代码如下：

```
declare
    rowVar_emp emp%rowtype;   --定义能够存储emp表中一行数据的变量rowVar_emp
begin
    select *
    into rowVar_emp
    from emp
    where empno=7369;   --检索数据
    /*输出员工信息*/
    dbms_output.put_line('员工'||rowVar_emp.ename||'的编号是'||rowVar_emp.empno||',职务是'||rowVar_emp. job);
end;
/
```

定义变量和常量

定义变量

变量是指其值在运行程序过程中可以改变的数据存储结构，定义变量必需的元素就是变量名和数据类型，另外还有可选择的初始值，其标准语法格式如下：

```
<变量名> <数据类型> [(长度):=<初始值>];
```

定义一个用于存储国家名称的可变字符串变量var_countryname，该变量的最大长度是50，并且该变量的初始值为“中国”，代码如下：

```
var_countryname varchar2(50):='中国';
```

定义常量

常量是指其值在程序运行过程中不可被改变的数据存储结构，定义常量必需的元素包括常量名、数据类型、常量值和CONSTANT关键字，其标准语法格式如下：

```
<常量名> CONSTANT <数据类型>:=<常量值>;

```

对于一些固定的数值，如圆周率、光速等，为了防止其不慎被改变，最好定义成常量。

```
--定义一个常量con_day，用来存储一年的天数
con_day constant integer:=365;

```

变量的初始化

一般而言，如果变量的取值可以被确定，那么最好为其初始化一个数值。PL/SQL定义了一个未初始化变量应该存储的内容，其被赋值为NULL。NULL意味着“未定义或未知的取值”；换句话讲，NULL可以被默认地赋值给任何未经过初始化的变量，这是PL/SQL的一个独到之处，许多其他程序设计语言没有定义未初始化变量的取值。

PL/SQL表达式

表达式不能独立构成语句，表达式的结果是一个值，如果不给这个值安排一个存储的位置，则表达式本身毫无意义。通常，表达式作为赋值语句的一部分出现在赋值运算符的右边，或者作为函数的参数等。例如，123*23-24+33就是一个表达式，它是由运算符串连起来的一组数，按照运算符的意义运算会得到一个运算结果，这就是表达式的值。
“操作数”是运算符的参数。根据所拥有的参数个数，PL/SQL运算符可分为一元运算符（一个参数）和二元运算符（两个参数）。表达式按照操作对象的不同，也可以分为字符表达式和布尔表达式两种。

字符表达式

唯一的字符运算符就是并置运算符“||”，它的作用是把几个字符串连在一起，如表达式'Hello'||'World'||'!'的值等于'Hello World!'。

布尔表达式

PL/SQL控制结构都涉及布尔表达式。布尔表达式是一个判断结果为真还是为假的条件表达式，它的值只有TRUE、FALSE或NULL；
布尔表达式有3个布尔运算符，即AND、OR和NOT，与高级语言中的逻辑运算符一样，它们的操作对象是布尔变量或者表达式。
此外，BETWEEN操作符可以划定一个范围，在范围内则为真，否则为假；
IN操作符判断某一元素是否属于某个集合，属于则为真，不属于则为假。

流程控制语句

流程控制语句是所有过程性程序设计语言的关键，因为只有能够进行流程控制才能灵活地实现各种操作和功能，PL/SQL也不例外，其主要控制语句及其说明如下表所示：

选择语句

选择语句也被称为条件语句，它的主要作用是根据条件的变化选择执行不同的代码，主要分为以下4种语句。

IF…THEN语句

IF…THEN语句是选择语句中最简单的一种形式，它只做一种情况或条件的判断，其语法格式如下：

```
IF <condition_expression> THEN
plsql_sentence
END IF;

```

condition_expression为条件表达式，其值为true时，程序将会执行IF下面的PL/SQL语句（即plsql_sentence语句）；其值为false时，程序将会跳过IF下面的语句而直接执行END IF后面的语句。定义两个字符串变量，然后赋值，接着使用IF…THEN语句比较两个字符串变量的长度，并输出比较结果，代码如下：

```
set serveroutput on

declare
    var_name1 varchar2(50);   --定义两个字符串变量
    var_name2 varchar2(50);
begin
    var_name1:='Wast';        --给两个字符串变量赋值
    var_name2:='xiaofei';
    if length(var_name1) < length(var_name2) then  --比较两个字符串的长度大小
    /*输出比较后的结果*/
    dbms_output.put_line('字符串“'||var_name1||'”的长度比字符串“'||var_name2||'”的长度小');
    end if;
end;
/
```

如果IF后面的条件表达式存在“并且”“或者”“非”等逻辑运算，则可以使用AND、OR、NOT等逻辑运算符。另外，如果要判断IF后面的条件表达式的值是否为空值，则需要在条件表达式中使用is和null关键字，如下面的代码：

```
if last_name is null then
…;
end if;
```

IF…THEN…ELSE语句

在编写程序的过程中，IF…THEN…ELSE语句是最常用到的一种选择语句，它可以实现判断两种情况，只要IF后面的条件表达式为FALSE，程序就会执行ELSE语句下面的PL/SQL语句，其语法格式如下：

```
IF < condition_expression> THEN
plsql_sentence1;
ELSE
plsql_sentence2;
END IF;

```

condition_expression为条件表达式，若该条件表达式的值为TRUE，则执行IF下面的PL/SQL语句，即plsql_sentence1语句；否则，程序将执行ELSE下面的PL/SQL语句，即plsql_sentence2语句。

通过IF…THEN…ELSE语句实现只有年龄大于或等于60岁，才可以申请退休的功能，否则程序会提示不可以申请退休，代码如下：

```
set serveroutput on

declare
    age int:=55;  --定义整型变量并赋值
begin
    if age >= 60 then  --比较年龄是否大于或等于60岁
    dbms_output.put_line('您可以申请退休了！');--输出可以退休信息
    else
    dbms_output.put_line('您小于60岁，不可以申请退休！');  --输出不可退休信息
    end if;
end;
/
```

IF…THEN…ELSIF语句

IF…THEN…ELSIF语句实现了多分支判断选择，它使程序的判断选择条件更加丰富，更加多样化。如果该语句中的哪个判断分支的表达式为TRUE，那么程序就会执行其下面对应的PL/SQL语句，其语法格式如下：

```
IF < condition_expression1 > THEN
plsql_sentence_1;
ELSIF < condition_expression2 >
THEN plsql_sentence_2;
…
ELSE plsql_sentence_n;
END IF;

```

- condition_expression1：第一个条件表达式，若其值为FALSE，则程序继续判断condition_expression2表达式。
- condition_expression2：第二个条件表达式，若其值为FALSE，则程序继续判断下面的ELSIF语句后面的表达式；若再没有ELSIF语句，则程序将执行ELSE语句下面的PL/SQL语句。
- plsql_sentence_1：第一个条件表达式的值为TRUE时，将要执行的PL/SQL语句。
- plsql_sentence_2：第二个条件表达式的值为TRUE时，将要执行的PL/SQL语句。
- plsql_sentence_n：当其上面所有的条件表达式的值都为FALSE时，将要执行的PL/SQL语句。

首先指定一个月份数值，然后使用IF…THEN…ELSIF语句判断它所属的季节，并输出季节信息，代码如下：

```
set serveroutput on

declare
 month int:=6;      --定义整型变量并赋值
begin
 if month >= 0 and month <= 3  then      --判断春季
  dbms_output.put_line(month||'月是春季');
 elsif  month >= 4 and month <= 6 then   --判断夏季
  dbms_output.put_line(month||'月是夏季');
 elsif  month >= 7 and month <= 9  then  --判断秋季
  dbms_output.put_line(month||'月是秋季');
 elsif  month >= 10 and month <= 12 then --判断冬季
  dbms_output.put_line(month||'月是冬季');
 else
  dbms_output.put_line('对不起，月份不合法! ');
 end if;
end;
/
```

注意：在IF…THEN…ELSIF语句中，多个条件表达式之间不能存在逻辑上的冲突，否则程序将判断出错。

CASE语句

CASE语句的执行方式与IF…THEN…ELSIF语句十分相似。在CASE关键字的后面有一个选择器，它通常是一个变量，程序就从这个选择器开始执行，接下来是WHEN子句，并且在WHEN关键字的后面是一个表达式，程序将根据选择器的值去匹配每个WHEN子句中的表达式的值，以实现执行不同的PL/SQL语句的功能，其语法格式如下：

```
CASE < selector>
WHEN <expression_1> THEN plsql_sentence_1;
WHEN <expression_2> THEN plsql_sentence_2;
…
WHEN <expression_n> THEN plsql_sentence_n;
[ELSE plsql_sentence;]
END CASE;

```

- selector：一个变量，用来存储要检测的值，通常被称为选择器。该选择器的值需要与WHEN子句中的表达式的值进行匹配。
- expression_1：第一个WHEN子句中的表达式，它通常是一个常量，当选择器的值等于该表达式的值时，程序将执行plsql_sentence_1语句。
- expression_2：第二个WHEN子句中的表达式，它通常也是一个常量，当选择器的值等于该表达式的值时，程序将执行plsql_sentence_2语句。
- expression_n：第n个WHEN子句中的表达式，它通常也是一个常量，当选择器的值等于该表达式的值时，程序将执行plsql_sentence_n语句。
- plsql_sentence：一个PL/SQL语句，当没有与选择器匹配的WHEN常量时，程序将执行该PL/SQL语句，其所在的ELSE语句是一个可选项。

首先指定一个季度数值，然后使用CASE语句判断它所包含的月份信息并输出，代码如下：

```
set serveroutput on

declare
    season int:=3;            --定义整型变量并赋值
    aboutInfo varchar2(50);   --存储月份信息
begin
    case season    --判断季度
    when 1 then    --若是1季度
    aboutInfo := season||'季度包括1，2，3月份';
    when 2 then    --若是2季度
    aboutInfo := season||'季度包括4，5，6月份';
    when 3 then    --若是3季度
    aboutInfo := season||'季度包括7，8，9月份';
    when 4 then    --若是4季度
    aboutInfo := season||'季度包括10，11，12月份';
    else           --若季度不合法
    aboutInfo := season||'季节不合法';
    end case;
    dbms_output.put_line(aboutinfo);  --输出该季度所包含的月份信息
end;
/

```

在进行多种情况判断时，建议使用CASE语句替换IF…THEN…ELSIF语句，因为CASE语句的语法更加简洁明了，易于阅读。

循环语句

当程序需要反复执行某一操作时，就必须使用循环结构。PL/SQL中的循环语句主要包括LOOP语句、WHILE语句和FOR语句3种。

LOOP语句

LOOP语句会先执行一次循环体，然后判断EXIT WHEN关键字后面的条件表达式的值是TRUE还是FALSE。如果是TRUE，程序会退出循环体；否则，程序将再次执行循环体。这样就使得程序至少能够执行一次循环体，其语法格式如下：

```
LOOP
plsql_sentence;
EXIT WHEN end_condition_exp
END LOOP;

```

- plsql_sentence：循环体中的PL/SQL语句，可能是一条语句，也可能是多条，这是循环体的核心部分，这些PL/SQL语句至少会被执行一遍。
- end_condition_exp：循环结束表达式，当该表达式的值为TRUE时，程序会退出循环体，否则程序将再次执行循环体。

使用LOOP语句计算前100个自然数的和，并输出到屏幕上，代码如下：

```
set serveroutput on

declare
    sum_i int:= 0;  --定义整数变量，存储整数和
    i int:= 0;      --定义整数变量，存储自然数
begin
    loop            --循环累加自然数
    i:=i+1;         --得出自然数
    sum_i:= sum_i+i;    --计算前n个自然数的和
    exit when i = 100;  --当循环100次时，程序退出循环体
    end loop;
    dbms_output.put_line('前100个自然数的和是：'||sum_i); --计算前100个自然数的和
end;
/
```

WHILE语句

WHILE语句根据它的条件表达式的值执行零次或多次循环体，在每次执行循环体之前，首先要判断条件表达式的值是否为TRUE，若为TRUE，则程序执行循环体；否则退出WHILE循环，然后继续执行WHILE语句后面的其他代码，其语法格式如下：

```
WHILE condition_expression LOOP
plsql_sentence;
END LOOP;

```

condition_expression为条件表达式，当其值为TRUE时，程序执行循环体，否则程序退出循环体。程序每次在执行循环体之前，都要首先判断该表达式的值是否为TRUE。

使用WHILE语句计算前100个自然数的和，并输出到屏幕上，代码如下：

```
set serveroutput on

declare
    sum_i int:= 0;    --定义整数变量，存储整数和
    i int:= 0;        --定义整数变量，存储自然数
begin
    while i<=99 loop  --当i的值等于100时，程序退出WHILE循环
    i:=i+1;           --得出自然数
    sum_i:= sum_i+i;  --计算前n个自然数的和
    end loop;
    dbms_output.put_line('前100个自然数的和是：'||sum_i); --计算前100个自然数的和
end;
/
```

FOR语句

FOR语句是一个可预置循环次数的循环控制语句，它有一个循环计数器，通常是一个整型变量，通过这个循环计数器来控制循环执行的次数。该计数器可以从小到大进行记录，也可以相反，从大到小进行记录。另外，该计数器值的合法性由上限值和下限值控制，若计数器值在上限值和下限值的范围内，则程序执行循环；否则，终止循环。其语法格式如下：

```
FOR variable_counter_name in [REVERSE] lower_limit..upper_limit LOOP
plsql_sentence;
END LOOP;

```

- variable_counter_name：表示一个变量，通常为整数类型，用来作为计数器。默认情况下，计数器的值会循环递增，当在循环中使用REVERSE关键字时，计数器的值会随循环递减。
- lower_limit：计数器的下限值，当计数器的值小于下限值时，程序终止FOR循环。
- upper_limit：计数器的上限值，当计数器的值大于上限值时，程序终止FOR循环。
- plsql_sentence：表示PL/SQL语句，作为FOR语句的循环体。

使用FOR语句计算前100个自然数中偶数之和，并输出到屏幕上，代码如下：

```
set serveroutput on

declare
    sum_i int:= 0;    --定义整数变量，存储整数和
begin
    for i in reverse 1..100 loop  --遍历前100个自然数
    if mod(i,2)=0 then            --判断是否为偶数
    sum_i:=sum_i+i;               --计算偶数和
    end if;
    end loop;
    dbms_output.put_line('前100个自然数中偶数之和是：'||sum_i);
end;
/

```

在上述FOR语句中，由于使用了关键字REVERSE，表示计数器i的值为递减状态，即i的初始值为100，随着每次递减1，最后一次FOR循环时，i的值变为1。如果在FOR语句中不使用关键字REVERSE，则表示计数器i的值为递增状态，即i的初始值为1。

GOTO语句

GOTO语句的语法格式如下：

```
GOTO label;

```

这是个无条件转向语句。当执行GOTO语句时，控制程序会立即转到由标签标识的语句中。其中，label是在PL/SQL中定义的符号。标签使用双箭头括号(<< >>)括起来。
使用GOTO语句的例子，代码如下：

```
… --程序其他部分
<<goto_mark>>  --定义了一个转向标签goto_mark
… --程序其他部分
IF no>98050 THEN
    GOTO goto_mark;  --如果条件成立，则转向goto_mark继续执行
… --程序其他部分

```

在使用GOTO语句时需要十分谨慎。不必要的GOTO语句会使程序代码复杂化，容易出错，而且难以理解和维护。事实上，几乎所有使用GOTO的语句都可以使用其他PL/SQL控制结构（如循环或条件结构）重新编写。

游标

游标提供了一种从表中检索数据并进行操作的灵活手段，游标主要用在服务器上，处理由客户端发送给服务器端的SQL语句，或是批处理、存储过程、触发器中的数据处理请求。游标的作用就相当于指针，通过游标PL/SQL程序可以一次处理查询结果集中的一行，并可以对该行数据执行特定操作，从而为用户在处理数据的过程中提供了很大方便。
在Oracle中，通过游标操作数据主要使用显式游标和隐式游标。另外，还包括具有引用类型特性的REF游标。

基本原理

在PL/SQL块中执行SELECT、INSERT、UPDATE和DELETE语句时，Oracle会在内存中为其分配上下文区(context area)，即一个缓冲区。游标是指向该区的一个指针，或是命名一个工作区(work area)，或是一种结构化数据类型。游标为应用程序提供了一种具有对多行数据查询结果集中的每一行数据分别进行单独处理的方法，是设计嵌入式SQL语句的应用程序的常用编程方法。
游标分为显式游标和隐式游标两种；

1. 显式游标是由用户声明和操作的一种游标；
2. 隐式游标是Oracle为所有数据操纵语句（包括只返回单行数据的查询语句）自动声明和操作的一种游标。

在每个用户会话中，可以同时打开多个游标，其数量由数据库初始化参数文件中的open cursors参数定义。

显式游标

显式游标是由用户声明和操作的一种游标，通常用于操作查询结果集（即由SELECT语句返回的查询结果），使用它处理数据的步骤包括声明游标、打开游标、读取游标和关闭游标4个步骤。其中，读取游标可能需要反复操作，因为游标每次只能读取一行数据，所以对于多条记录，需要反复读取，直到游标读取不到数据为止。其操作过程如下图所示：

声明游标需要在块的声明部分进行，其他3个步骤都在执行部分或异常处理中进行。

声明游标

声明游标主要包括指定游标名称和为游标提供结果集的SELECT语句，语法格式如下：

```
CURSOR cur_name[(input_parameter1[,input_parameter2]…)] [RETURN ret_type]
IS select_ sentence;

```

- cur_name：表示所声明的游标名称。
- ret_type：表示执行游标操作后的返回值类型，这是一个可选项。
- select_ sentence：游标所使用的SELECT语句，它为游标的反复读取提供了结果集。
- input_parameter1：作为游标的“输入参数”，可以有多个，这是一个可选项。它指定用户在打开游标后向游标中传递的值，该参数的定义和初始化格式如下：

```
para_name [IN] DATATYPE [{:= | DEFAULT} para_value]

```

其中，para_name表示参数名称，其后面的关键字IN表示输入方向，可以省略；DATATYPE表示参数的数据类型，但数据类型不可以指定长度；para_value表示该参数的初始值或默认值，它也可以是一个表达式；para_name参数的初始值既可以以常规的方式赋值(:=)，也可以使用关键字DEFAULT初始化默认值。
与声明变量一样，声明游标也应该放在PL/SQL块的declare部分;
声明游标，用来读取emp表中职务为销售员(SALESMAN)的员工信息，代码如下:

```
declare
  cursor cur_emp(var_job in varchar2:='SALESMAN')
  is select empno,ename,sal
     from emp
     where job=var_job;

```

在上述代码中，首先声明了一个名称为cur_emp的游标，并定义了一个输入参数var_job（类型为varchar2，但不可以指定长度，如varchar2(10)，否则程序报错），该参数用来存储员工的职务（初始值为SALESMAN）；然后使用SELECT语句检索得到职务是销售员的结果集，以等待游标逐行读取它。

打开游标

在游标声明完毕之后，必须打开游标才能使用。打开游标的语法格式如下：

```
OPEN cur_name[(para_value1[,para_value2]…)];

```

- cur_name：要打开的游标名称。
- para_value1：指定“输入参数”的值，根据声明游标时的实际情况，可以是多个或一个，这是一个可选项。如果在声明游标时定义了“输入参数”，并初始化其值，而在此处省略“输入参数”的值，则表示游标将使用“输入参数”的初始值；若在此处指定“输入参数”的值，则表示游标将使用这个指定的“参数值”。

打开游标就是执行定义的SELECT语句。执行完毕，将查询结果装入内存，游标停在查询结果的首部（注意，并不是第一行）。打开一个游标时，会完成以下几件事：

- 检查联编变量的取值。
- 根据联编变量的取值，确定活动集。
- 活动集的指针指向第一行。

```
--打开游标cur_emp，给游标的“输入参数”赋值为“MANAGER”
OPEN cur_emp('MANAGER');
```

这里可以省略“('MANAGER')”，这样表示“输入参数”的值仍然使用其初始值（即SALESMAN）；

读取游标

当打开一个游标之后，就可以读取游标中的数据了，读取游标就是逐行将结果集中的数据保存到变量中。读取游标使用FETCH…INTO语句，其语法格式如下：

```
FETCH cur_name INTO {variable};
```

- cur_name：要读取的游标名称。
- variable：一个变量列表或“记录”变量（RECORD类型），Oracle使用“记录”变量来存储游标中的数据，要比使用变量列表方便得多。

在游标中包含一个数据行指针，它用来指向当前数据行。刚刚打开游标时，指针指向结果集中的第一行，当使用FETCH…INTO语句读取数据完毕之后，游标中的指针将自动指向下一行数据。这样，就可以在循环结构中使用FETCH…INTO语句来读取数据，每一次循环都会从结果集中读取一行数据，直到指针指向结果集中最后一条记录之后为止（实际上，最后一条记录之后是不存在的，是空的，这里只是表示遍历完所有的数据行），这时游标的%FOUND属性值为FALSE。
声明一个检索emp表中员工信息的游标，然后打开游标，并指定检索职务是MANAGER的员工信息，接着使用FETCH…INTO语句和WHILE循环语句读取游标中的所有员工信息，最后输出读取的员工信息，代码如下：

```
set serveroutput on
declare
    /*声明游标，检索员工信息*/
    cursor cur_emp (var_job in varchar2:='SALESMAN')
    is select empno,ename,sal
       from emp
       where job=var_job;
    type record_emp is record  --声明一个记录类型（RECORD类型）
    (
      /*定义当前记录的成员变量*/
      var_empno emp.empno%type,
      var_ename emp.ename%type,
      var_sal emp.sal%type
     );
     emp_row record_emp;    --声明一个record_emp类型的变量
begin
     open cur_emp('MANAGER');        --打开游标
        fetch cur_emp into emp_row;  --先让指针指向结果集中的第一行，并将值保存到emp_row中
     while cur_emp%found loop
        dbms_output.put_line(emp_row.var_ename||'的编号是'||emp_row.var_empno||',工资是'||emp_row.var_sal);
        fetch cur_emp into emp_row;  --让指针指向结果集中的下一行，并将值保存到emp_row中
     end loop;
     close cur_emp;   --关闭游标
end;
/
```

关闭游标

当所有的活动集都被检索以后，游标就应该被关闭。PL/SQL程序将被告知对于游标的处理已经结束，与游标相关联的资源可以被释放了。这些资源包括用来存储活动集的存储空间，以及用来存储活动集的临时空间。关闭游标的语法格式如下：

```
CLOSE cur_name;

```

参数cur_name表示要关闭的游标名称。一旦关闭了游标，SELECT操作就会被关闭，并释放占用的内存区。如果再从游标提取数据就是非法的，这样做会产生以下两种Oracle错误：

```
ORA-1001: Invalid CUSOR --非法游标
ORA-1002: FETCH out of sequence --超出界限
```

隐式游标

在执行一个SQL语句时，Oracle会自动创建一个隐式游标，这个游标是内存中处理该语句的工作区域。隐式游标主要是处理数据操作语句（如UPDATE、DELETE语句）的执行结果，当然在特殊情况下，也可以处理SELECT语句的查询结果。由于隐式游标也有属性，因此当使用隐式游标的属性时，需要在属性前面加上隐式游标的默认名称—SQL。

在实际的PL/SQL编程中，经常使用隐式游标来判断更新数据行或删除数据行的情况。

把emp表中销售员（即SALESMAN）的工资上调20%，然后使用隐式游标SQL的%ROWCOUNT属性输出上调工资的员工数量，代码如下：

```
set serveroutput on

begin
    update emp
    set sal=sal*(1+0.2)
    where job='SALESMAN';     --把销售员的工资上调20%
    if sql%notfound then      --若UPDATE语句没有影响到任何一行数据
    dbms_output.put_line('没有员工需要上调工资');
    else   --若UPDATE语句至少影响到一行数据
    dbms_output.put_line('有'||sql%rowcount||'个员工工资上调20%');
    end if;
end;
/

```

在上述代码中，标识符SQL就是UPDATE语句在更新数据过程中所使用的隐式游标，它通常处于隐藏状态，是由Oracle系统自动创建的。当需要使用隐式游标的属性时，标识符SQL就必须显式地添加到属性名称之前。另外，无论是隐式游标还是显式游标，它们的属性总是反映最近的一条SQL语句的处理结果。因此，在一个PL/SQL块中出现多个SQL语句时，游标的属性值只能反映出紧挨着它的上一条SQL语句的处理结果。

游标的属性

无论是显式游标还是隐式游标，都具有%FOUND、%NOTFOUND、%ROWCOUNT和%ISOPEN 4个属性，通过这4个属性可以获知SQL语句的执行结果以及该游标的状态信息。

游标属性只能用在PL/SQL的流程控制语句内，而不能用在SQL语句内。下面对这4个属性的功能进行讲解。

- %FOUND：布尔型属性，如果SQL语句至少影响到一行数据，则该属性为TRUE，否则为FALSE。
- %NOTFOUND：布尔型属性，与%FOUND属性的功能相反。
- %ROWCOUNT：数字型属性，返回受SQL语句影响的行数。
- %ISOPEN：布尔型属性，当游标已经打开时返回TRUE，当游标关闭时返回FALSE。

是否找到游标(%FOUND)

%FOUND属性表示当前游标是否指向有效一行，若是则值为TRUE，否则值为FALSE。检查此属性可以判断是否结束游标使用。

```
open cur_emp;                              --打开游标
fetch cur_emp into var_ename,var_job;  --将第一行数据放入变量中，游标后移
loop
  exit when not cur_em%found;          --使用了%FOUND属性
end loop;

```

在隐式游标中%FOUND属性的引用方法是SQL %FOUND。使用SQL %FOUND，代码如下：

```
delete from emp where empno = emp_id;--emp_id为一个有值变量
if sql%found then--如果删除成功，则将该行员工编号写入success表中
   insert into success values(empno);
else --如果删除不成功，则将该行员工编号写入fail表中
   insert into fail values(empno);
end if;

```

是否没找到游标(%NOTFOUND)

%NOTFOUND属性与%FOUND属性类似，但其值恰好相反。

```
open cur_emp;  --打开游标
fetch cur_emp into var_ename,var_job;  --将第一行数据放入变量中，游标后移
loop
    exit when cur_em%notfound;  --使用了%NOTFOUND属性
end loop;

```

在隐式游标中%NOTFOUND属性的引用方法是SQL %NOTFOUND。

```
delete from emp where empno = emp_id; --emp_id为一个有值变量
if sql %notfound then  --如果删除不成功，则将该行员工编号写入fail表中
  insert into fail values(empno);
else    --如果删除成功，则将该行员工编号写入success表中
  insert into success values(empno);
end if;

```

游标行数(%ROWCOUNT)

%ROWCOUNT属性记录了游标抽取过的记录行数，也可以理解为当前游标所在的行号。这个属性在循环判断中也很有用，使得不必抽取所有记录行就可以中断游标操作。

```
loop
  fetch cur_emp into var_empno,var_ename,var_job;
  exit when cur_emp%rowcount = 10; --只抽取10条记录
  …
end loop;

```

还可以用FOR语句控制游标的循环，系统隐含地定义了一个数据类型为ROWCOUNT的记录，作为循环计数器，将隐式地打开和关闭游标。

游标是否打开(%ISOPEN)

%ISOPEN属性表示游标是否处于打开状态。在实际应用中，使用一个游标前，第一步往往是检查它的%ISOPEN属性，看其是否已打开，若没有，要打开游标再向下操作。这是防止程序运行过程中出错的关键一步。

```
if cur_emp%isopen tneh
  fetch cur_emp into var_empno,var_ename,var_job;
else
  open cur_emp;
end if;

```

在隐式游标中此属性的引用方法是SQL %ISOPEN。**隐式游标中SQL %ISOPEN属性总为TRUE**，因此在隐式游标使用中不用打开和关闭游标，也不用检查其打开状态。

参数化游标

在定义游标时，可以带上参数，使得在使用游标时，根据参数不同所选中的数据行也不同，达到动态使用的目的。

声明一个游标，用于检索指定员工编号的员工信息，然后使用游标的%FOUND属性来判断是否检索到指定员工编号的员工信息，代码如下：

```
set serveroutput on
declare
  var_ename varchar2(50);  --声明变量，用来存储员工名称
  var_job varchar2(50);    --声明变量，用来存储员工的职务
  /*声明游标，检索指定员工编号的员工信息*/
  cursor cur_emp           --定义游标，检索指定编号的记录信息
  is select ename,job
     from emp
     where empno=7499;
begin
  open cur_emp;                           --打开游标
  fetch cur_emp into var_ename,var_job;  --读取游标，并存储员工名称和职务
  if cur_emp%found then                   --若检索到数据记录，则输出员工信息
     dbms_output.put_line('编号是7499的员工名称为：'||var_ename||'，职务是：'||var_job);
  else
     dbms_output.put_line('无数据记录');  --提示无记录信息
  end if;
end;
/
```

使用显式游标时，需要注意以下事项：

- 使用前必须用%ISOPEN检查其打开状态，只有此值为TRUE的游标才可使用，否则要先将游标打开。
- 在使用游标的过程中，每次都要用%FOUND或%NOTFOUND属性检查是否成功返回，即是否还有要操作的行。
- 将游标中行取至变量组中时，对应变量个数和数据类型必须完全一致。
- 使用完游标必须将其关闭，以释放相应内存资源。

游标变量

如同常量和变量的区别一样，前面所讲的游标都是与SQL语句相关联的，它是静态的，并且在编译该块时此语句已经是可知的。而游标变量可以在运行时与不同的语句相关联，它是动态的。游标变量被用于处理多行的查询结果集。在同一个PL/SQL块中，游标变量不同于特定的查询绑定，而是在打开游标时才能确定所对应的查询。因此，游标变量可以一次对应多个查询。
使用游标变量之前，必须先声明它，然后在运行时必须为其分配存储空间。游标变量是REF类型的变量，类似于高级语句中的指针。

声明游标变量

游标变量是一种引用类型。当程序运行时，它们可以指向不同的存储单元。如果要使用引用类型，首先要声明该变量，然后相应的存储单元必须被分配。PL/SQL中的引用类型变量通过下述语法进行声明：

```
REF type
```

type是已经被定义的类型。REF关键字指明新的类型必须是一个指向经过定义的类型的指针。因此，游标变量可以使用的类型就是REF CURSOR。
定义一个游标变量类型的完整语法如下：

```
TYPE <类型名> IS REF CURSOR
RETURN <返回类型>
```

其中，<类型名>是新的引用类型的名字，而<返回类型>是一个记录类型，它指明了最终由游标变量返回的选择列表的类型。
游标变量的返回类型必须是一个记录类型。它可以被显式声明为一个用户定义的记录，或者隐式使用%ROWTYPE进行声明。在定义了引用类型以后，就可以声明该变量了。
在声明部分，给出用于游标变量的不同游标，代码如下：

```
set serveroutput on

DECLARE
  TYPE t_StudentRef IS REF CURSOR --定义使用%ROWTYPE
  RETURN STUDENTS%ROWTYPE;
  TYPE t_AbstractstudentsRecord IS RECORD( --定义新的记录类型
     sname STUDENTS.sname%TYPE,
     sex STUDENTS.sex%type);
  v_AbstractStudentsRecord t_AbstractStudentsRecord;
  TYPE t_AbstractStudentsRef IS REF CURSOR  --使用记录类型的游标变量
  RETURN t_AbstractStudentsRecord;
  TYPE t_NameRef2 IS REF CURSOR   --另一类型定义
  RETURN v_AbstractStudentsRecord%TYPE;
  v_StudentCV t_StudentsRef;      --声明上述类型的游标变量
  v_AbstractStudentCV t_AbstractStudentsRef;

```

在上面代码中极少的游标变量是受限的，它的返回类型只能是特定类型。而在PL/SQL语句中，还有一种非受限游标变量，它在声明时没有RETURN子句。一个非受限游标变量可以为任何查询打开。
定义游标变量，代码如下：

```
DECLARE
  --定义非受限游标变量
  TYPE t_FlexibleRefIS REF CURSOR;
  --游标变量
  V_CURSORVar t_FlexibleRef;

```

打开游标变量
如果要将一个游标变量与一个特定的SELECT语句相关联，需要使用OPEN FOR语句，其语法格式如下：

```
OPEN<游标变量>FOR<SELECT语句>;

```

如果游标变量是受限的，则SELECT语句的返回类型必须与游标变量所受限的记录类型匹配，如果不匹配，则Oracle会返回错误ORA_6504。
打开游标变量v_StudentSCV，代码如下：

```
DECLARE
  TYPE t_StudentRef IS REF CURSOR  --定义使用%ROWTYPE
  RETURN STUDENTS%ROWTYPE;
  v_StudentSCV t_StudentRef;       --定义新的记录类型
BEGIN
  OPEN v_StudentSCV FOR
     SELECT * FROM STUDENTS;
END;

```

关闭游标变量

游标变量的关闭和静态游标的关闭类似，均使用CLOSE语句，这会释放查询所使用的空间。关闭已经关闭的游标变量是非法的。

通过FOR语句循环游标

在使用隐式游标或显式游标处理具有多行数据的结果集时，可以配合使用FOR语句来完成。在使用FOR语句遍历游标中的数据时，可以把它的计时器看作一个自动的RECORD类型的变量。
在FOR语句中遍历隐式游标中的数据时，通常在关键字IN的后面提供由SELECT语句检索的结果集，在检索结果集的过程中，Oracle系统会自动提供一个隐式的游标SQL。
使用隐式游标和FOR语句检索出职务是销售员(SALESMAN)的员工信息并输出，代码如下：

```
set serveroutput on

begin
  for emp_record in (select empno,ename,sal from emp where job='SALESMAN') --遍历隐式游标中的记录
  loop
    dbms_output.put('员工编号：'||emp_record.empno);       --输出员工编号
    dbms_output.put('；员工名称：'||emp_record.ename);     --输出员工名称
    dbms_output.put_line('；员工工资：'||emp_record.sal);  --输出员工工资
  end loop;
end;
/
```

在FOR语句中遍历显式游标中的数据时，通常在关键字IN的后面提供游标的名称，其语法格式如下：

```
FOR var_auto_record IN cur_name LOOP
plsqlsentence;
END LOOP;

```

- var_auto_record：自动的RECORD类型的变量，可以是任意合法的变量名称。
- cur_name：指定的游标名称。
- plsqlsentence：PL/SQL语句。

使用显式游标和FOR语句检索出部门编号是30的员工信息并输出，代码如下：

```
set serveroutput on
declare
  cursor cur_emp is
    select * from emp
    where deptno = 30; --检索部门编号为30的员工信息
begin
  for emp_record in cur_emp  --遍历员工信息
  loop
    dbms_output.put('员工编号：'||emp_record.empno);      --输出员工编号
    dbms_output.put('；员工名称：'||emp_record.ename);    --输出员工名称
    dbms_output.put_line('；员工职务：'||emp_record.job); --输出员工职务
   end loop;
end;
/
```

注意：在使用游标（包括显式和隐式）的FOR循环中，可以声明游标，但不用进行打开游标、读取游标和关闭游标等操作，这些由Oracle系统自动完成。

异常处理

异常处理方法

在编写PL/SQL程序时，不可避免地会发生一些错误，可能是程序设计人员自己造成的，也可能是操作系统或硬件环境出错，如出现除数为零、磁盘I/O错误等情况。对于出现的这些错误，Oracle采用异常机制来处理，异常处理代码通常放在PL/SQL的EXCEPTION代码块中。根据异常产生的机制和原理，可将Oracle系统异常分为以下两大类：

- 预定义异常：Oracle系统自身为用户提供了大量的、可在PL/SQL中使用的预定义异常，以便检查用户代码失败的一般原因，它们都定义在Oracle的核心PL/SQL库中，用户可以在自己的PL/SQL异常处理部分使用名称对其进行标识，对这种异常情况的处理，用户无须在程序中定义，它们由Oracle自动引发。
- 自定义异常：有时候可能会出现操作系统错误或机器硬件故障，这些错误Oracle系统自身无法知晓，也不能控制。例如，操作系统因病毒破坏而产生故障、磁盘损坏、网络突然中断等。另外，因业务的实际需求，程序设计人员需要自定义一些错误的业务逻辑，而PL/SQL程序在运行过程中就可能会触发到这些错误的业务逻辑。那么，对于以上这些异常情况的处理，就需要用户在程序中自定义异常，然后由Oracle自动引发。

异常的处理方法有两种：预定义异常处理和用户自定义异常处理。

预定义异常处理

每当PL/SQL程序违反了Oracle的规则或超出系统的限制时，系统就自动地产生内部异常。每个Oracle异常都有一个号码，但异常必须按名处理。因此，PL/SQL对那些常见的异常预定义了异常名。

用户自定义异常处理

用户自定义异常处理方法分为如下3个部分：

- 异常声明：用户定义异常包括预定义异常和用户自定义异常。用户定义的异常只能在PL/SQL块的声明部分进行声明，声明方式与变量声明类似。
- 抛出异常：用户定义的异常使用RAISE语句显式地提出。
- 为内部异常命名：在PL/SQL中，必须使用OTHERS处理程序或用伪命令EXCEPTION_INIT来处理未命名的内部异常。

注意：异常是一种状态而不是一个对象，因此异常名不能出现在赋值语句或SQL语句中。PRAGMA EXCEPTION_INIT的作用是将一个异常名与一个Oracle错误号码联系起来。因此，用户就可以按名称引用任何内部异常，并为它编写一个特定的处理程序。

异常处理语法

异常处理语法主要分为如下5个部分。

声明异常

声明异常的代码如下：

```
exception_name EXCEPTION
```

其中，exception_name为用户定义的异常名。

为内部异常命名

为内部异常命名的代码如下：

```
PRAGE EXCEPTION_INIT(exception_name,ORA_errornumber);
```

其中，ORA_errornumber为用户定义的Oracle错误号。

异常定义

异常定义的代码如下：

```
DECLARE
 exception_name EXCEPTION;
BEGIN
 IF condition THEN
  RAISE exception_name;
 END IF;
 EXCEPTION
  WHERE exception_name THEN
  statement;
END;

```

异常处理

异常处理的代码如下：

```
SET SERVEROUTPUT ON
EXCEPTION
 WHEN exception1 THEN
  statement1
 WHEN exception2 THEN
  statement2
 ...
 WHEN OTHERS THEN
  statement3

```

使用SQLCODE和SQLERRM函数定义提示信息

使用SQLCODE和SQLERRM函数定义提示信息的代码如下：

```
DBMS_OUTPUT.PUT_LINE('错误号：'||SQLCODE); DBMS_OUTPUT.PUT_LINE('错误号：'||SQLERRM);

```

预定义异常

当PL/SQL程序违反Oracle系统内部规定的设计规范时，将会自动引发一个预定义的异常。例如，当除数为零时，将会引发ZERO_DIVIED异常。Oracle系统常见的预定义异常标识符如下：

- ACCESS_INTO_NULL：该异常对应于ORA-06530错误。为了引用对象属性，必须首先初始化对象。当直接引用未初始化的对象属性时，将会触发该异常。
- CASE_NOT_FOUND：该异常对应于ORA-06592错误。当CASE语句的WHEN子句没有包含必须条件分支或者ELSE子句时，将会触发该异常。
- COLLECTION_IS_NULL：该异常对应于ORA-06531错误。在给嵌套表变量或者VARRAY变量赋值之前，必须首先初始化集合变量。如果没有初始化集合变量，则将会触发该异常。
- CURSOR_ALREADY_OPEN：该异常对应于ORA-06511错误。当在已打开游标上执行OPEN操作时，将会触发该异常。
- INVALID_CURSOR：该异常对应于ORA-01001错误。当视图从未打开游标提取数据，或者关闭未打开游标时，将会触发该异常。
- INVALID_NUMBER：该异常对应于ORA-01722错误。当内嵌SQL语句不能将字符转变成数字时，将会触发该异常。
- LOGIN_DENIED：该异常对应于ORA-01017错误。当连接到Oracle数据库时，如果提供不正确的用户名或者口令，则将会触发该异常。
- NO_DATA_FOUND：该异常对应于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，将会触发该异常。
- NOT_LOGGED_ON：该异常对应于ORA-01012错误。当没有连接到Oracle数据库时，如果执行内嵌SQL语句，则将会触发该异常。
- PROGRAM_ERROR：该异常对应于ORA-06501错误。如果出现该错误，则表示存在PL/SQL内部问题，在这种情况下需要重新安装数据字典视图和PL/SQL包。
- ROWTYPE_MISMATCH：该异常对应于ORA-016504错误。当执行赋值操作时，如果宿主变量和游标变量具有不兼容的返回类型，则将会触发该异常。
- SELF_IS_NULL：该异常对应于ORA-30625错误。当使用对象类型时，如果在NULL实例上调用成员方法，则将会触发该异常。
- STORAGE_ERROR：该异常对应于ORA-06500错误。当执行PL/SQL块时，如果超出内存空间或者内存被破坏，则将会触发该异常。
- SUBSCRIPT_BEYOND_COUNT：该异常对应于ORA-06533错误。当使用嵌套表或者VARRAY元素时，如果下标超出嵌套表或者VARRAY元素的范围，则将会触发该异常。
- SUBSCRIPT_OUTSIDE_LIMIT：该异常对应于ORA-06532错误。当使用嵌套表或者VARRAY元素时，如果元素下标为负值，则将会触发该异常。
- SYS_INVALID_ROWID：该异常对应于ORA-01410错误。当字符串被转变为ROWID时，如果使用无效字符串，则将会触发该异常。
- TIMEOUT_ON_RESOURCE：该异常对应于ORA-00051错误。当等待资源时，如果出现超时错误，则将会触发该异常。

- TOO_MANY_ROWS：该异常对应于ORA-01422错误。当执行SELECT INTO语句时，如果返回超过一行，则将会触发该异常。
- VALUE_ERROR：该异常对应于ORA-06502错误。当执行赋值操作时，如果变量长度不足以容纳实际数据，则将会触发该异常。
- ZERO_DIVIDE：该异常对应于ORA-01476错误。当使用数字值除以0时，将会触发该异常。

使用SELECT INTO语句检索emp表中部门编号为10的员工记录信息，然后使用TOO_MANY_ROWS预定义异常捕获错误信息并输出，代码如下：

```
set serveroutput on

declare
  var_empno number;        --定义变量，存储员工编号
  var_ename varchar2(50); --定义变量，存储员工名称
begin
  select empno,ename into var_empno,var_ename
  from emp
  where deptno=10;     --检索部门编号为10的员工信息
  if sql%found then    --若检索成功，则输出员工信息
    dbms_output.put_line('员工编号：'||var_empno||'；员工名称'||var_ename);
  end if;
exception    --捕获异常
  when too_many_rows then      --若SELECT INTO语句的返回记录超过一行
    dbms_output.put_line('返回记录超过一行');
  when no_data_found then      --若SELECT INTO语句的返回记录为0行
    dbms_output.put_line('无数据记录');
end;
/

```

自定义异常

Oracle系统内部的预定义异常只有20个左右，而实际程序运行过程中可能会产生几千种异常情况。因此，Oracle经常使用错误编号和相关描述输出异常信息。另外，程序设计人员可以根据实际的业务需求定义一些特殊异常，Oracle的自定义异常可以分为**错误编号异常**和**业务逻辑异常**两种。

错误编号异常

错误编号异常是指在Oracle系统发生错误时，系统会显示错误号和相关描述信息的异常。虽然直接使用错误编号也可以完成异常处理，但错误编号较为抽象，不易于用户理解和记忆。对于这种异常，首先在PL/SQL块的声明部分（DECLARE部分）使用EXCEPTION类型定义一个异常变量名，然后使用语句PRAGMA EXCEPTION_INIT为“错误编号”关联“这个异常变量名”，接下来就可以像对待系统预定义异常一样处理了。
下面通过一个具体的例子来演示如何为Oracle系统的“错误编号”做自定义异常处理。首先向dept表中插入一条部门编号为10的记录（部门编号10已经存在于dept表中，并且部门编号为dept表的唯一主键），然后执行INSERT语句，将会得到如下图所示的运行结果。可以看到，程序执行中断而崩溃，并显示错误信息为“ORA-00001”（即错误编号为“00001”）；

```
insert into dept values(10,'研发部','QINGDAO');
```

对于Oracle捕获到的上面这个异常可以通过下面的示例来解决；
首先定义错误编号为“00001”的异常变量，然后向dept表中插入一条能够“违反唯一约束条件”的记录，最后在EXCEPTION代码体中输出异常提示信息，代码如下：

```
set serveroutput on

declare
  primary_iterant exception;  --定义一个异常变量
  pragma exception_init(primary_iterant,-00001); --关联错误号和异常变量名
begin
  /*向dept表中插入一条与已有主键值重复的记录，以便引发异常*/
  insert into dept values(10,'研发部','青岛');
exception
  when primary_iterant then   --若Oracle捕获到的异常为-00001异常
    dbms_output.put_line('主键不允许重复！');  --输出异常描述信息
end;
/
```

使用异常处理机制，可以防止Oracle系统因引发异常而导致程序崩溃，使程序有机会自动纠正错误，而且自定义异常容易理解和记忆，方便用户的使用。

业务逻辑异常

在实际的应用中，程序开发人员可以根据具体的业务逻辑规则自定义一个异常。这样，当用户操作违反业务逻辑规则时，就会引发一个自定义异常，从而中断程序的正常执行，并转到自定义的异常处理部分。

无论是预定义异常，还是错误编号异常，都是由Oracle系统判断的错误，但业务逻辑异常是Oracle系统本身无法知道的，这样就需要有一个引发异常的机制，引发业务逻辑异常通常使用RAISE语句来实现。当引发一个异常时，控制就会转到EXCEPTION异常处理部分执行异常处理语句。业务逻辑异常首先要在DECLARE部分使用EXCEPTION类型声明一个异常变量，然后在BEGIN部分根据一定的业务逻辑规则执行RAISE语句（在RAISE关键字后面跟着异常变量名），最后在EXCEPTION部分编写异常处理语句。

自定义一个异常变量，在向dept表中插入数据时，若判断loc字段的值为NULL，则使用RAISE语句引发异常，并将程序的执行流程转入EXCEPTION部分中进行处理，代码如下：

```
set serveroutput on

declare
  null_exception exception;  --声明一个EXCEPTION类型的异常变量
  dept_row dept%rowtype;     --声明ROWTYPE类型的变量dept_row
begin
  dept_row.deptno := 66;       --给部门编号变量赋值
  dept_row.dname := '公关部'; --给部门名称变量赋值
  insert into dept
  values(dept_row.deptno,dept_row.dname,dept_row.loc); --向dept表中插入一条记录
  if dept_row.loc is null then    --如果判断loc变量的值为NULL
    raise null_exception;         --引发NULL异常，程序转入EXCEPTION部分
  end if;
exception
  when null_exception then        --当RAISE引发的异常是NULL_EXCEPTION时
  dbms_output.put_line('loc字段的值不许为null');  --输出异常提示信息
  rollback;     --回滚插入的数据记录
end;
/

```

使用DESC命令查看dept表的设计情况，可以看到loc字段允许为NULL，但实际应用中loc字段的值（部门位置）可能会被要求必须填写，这样程序设计人员就可以通过自定义业务逻辑异常来限制loc字段的值不许为空。

3.4 存储过程

命名的PL/SQL块可以被独立编译并存储在数据库中，Oracle提供了**4种可以存储的PL/SQL块，即过程、函数、触发器和包。**

存储过程是一种命名的PL/SQL块，它既可以没有参数，也可以有若干个输入、输出参数，甚至可以有多个既作为输入又作为输出的参数，但它通常没有返回值。存储过程被保存在数据库中，它不可以被SQL语句直接执行或调用，只能通过EXECUT命令执行或在PL/SQL块内部被调用。由于存储过程是已经编译好的代码，因此在被调用或引用时，其执行效率非常高。

创建存储过程

创建一个存储过程与编写一个普通的PL/SQL块有很多相似之处。例如，两者都包括声明部分、执行部分和异常处理3部分。但这两者之间的实现细节尚有很多差别，例如，创建存储过程需要使用PROCEDURE关键字，在关键字后面就是过程名和参数列表；创建存储过程不需要使用DECLARE关键字，而是使用CREATE或REPLACE关键字。其基本语法格式如下：

```
CREATE [OR REPLACE] PROCEDURE pro_name [(parameter1[,parameter2]…)] IS|AS
BEGIN
 plsql_sentences;
[EXCEPTION]
 [dowith _ sentences;]
END [pro_name];

```

- pro_name：存储过程的名称。如果数据库中已经存在此名称，则可以指定OR REPLACE关键字，这样新的存储过程将覆盖原来的存储过程。
- parameter1：存储过程的参数。若是输入参数，则需要在其后指定IN关键字；若是输出参数，则需要在其后指定OUT关键字。在IN或OUT关键字的后面是参数的数据类型，但不能指定该类型的长度。
- plsql_sentences：PL/SQL语句，它是存储过程功能实现的主体。
- dowith _ sentences：异常处理语句，也是PL/SQL语句，这是一个可选项。

注意：上述语法中的parameter1是存储过程被调用 / 执行时用到的参数，而不是存储过程内定义的内部变量，内部变量要在IS|AS关键字后面定义，并使用分号(;)结束。

创建一个存储过程，该存储过程实现向dept表中插入一条记录，代码如下：

```
create procedure pro_insertDept is
begin
  insert into dept values(66,'营销部','JINAN');  --插入数据记录
  commit;  --提交数据
  dbms_output.put_line('插入新记录成功! '); --提示插入记录成功
end pro_insertDept;
/
```

在当前模式下，如果数据库中存在同名的存储过程，则要求新创建的存储过程覆盖已存在的存储过程；如果不存在同名的存储过程，则可直接创建，代码如下：

```
create or replace procedure pro_insertDept is
begin
  insert into dept values(66,'营销部','JINAN');  --插入数据记录
  commit;  --提交数据
  dbms_output.put_line('插入新记录成功! ');  --提示插入记录成功
end pro_insertDept;
/

```

无论在数据库中是否存在名称为pro_insertDept的存储过程，上述代码都可以成功地创建一个存储过程。如果在创建存储过程中发生了错误，则用户还可以使用SHOW ERROR命令来查看错误信息。
上述两个存储过程中的主体代码都可以实现向数据表dept中插入一行记录，但主体代码INSERT语句仅仅是被编译了，并没有被执行。若要执行该INSERT语句，则需要在SQL* Plus环境中使用**EXECUTE命令**来执行该存储过程，或者在PL/SQL块中调用该存储过程。
使用EXECUTE命令的执行方式比较简单，只需要在该命令后面输入存储过程名即可；在SQL* Plus环境中，使用EXECUTE命令执行pro_insertDept存储过程，代码如下：

```
execute pro_insertDept;
```

代码中的EXECUTE命令也可简写为EXEC。有时候需要在一个PL/SQL块中调用某个存储过程，比如，在PL/SQL块中首先调用存储过程pro_insertDept，然后执行PL/SQL块，具体代码如下：

```
set serverout on
begin
  pro_insertDept;
end;
/

```

因为前面已经执行过一次pro_insertDept存储过程，已经向dept表插入一条deptno=66的数据了，不能再插入deptno=66的数据，提示违反唯一约束；
修改一下pro_insertDept存储过程插入的内容，然后再次执行PL/SQL块；

```
create or replace procedure pro_insertDept is
begin
  insert into dept values(77,'运营部','QINGDAO');  --插入数据记录
  commit;  --提交数据
  dbms_output.put_line('插入新记录成功！');  --提示插入记录成功
end pro_insertDept;
/
```

注意：在创建存储过程的语法中，对于IS关键字，也可以使用AS关键字来替代，效果是相同的。

存储过程的参数

前面所创建的存储过程都是简单的存储过程，都没有涉及参数。Oracle为了增强存储过程的灵活性，提供向存储过程传入参数的功能。参数是一种向程序单元输入和输出数据的机制，存储过程可以接受多个参数，参数模式包括IN、OUT和IN OUT共3种；

IN模式参数

IN模式参数是一种输入类型的参数，参数值由调用方传入，并且只能被存储过程读取。这种参数模式是最常用的，也是默认的参数模式，关键字IN位于参数名称之后。
创建一个存储过程，并定义3个IN模式的变量，然后将这3个变量的值插入dept表中，代码如下：

```
create or replace procedure insert_dept(
  num_deptno in number,   --定义IN模式的变量，它存储部门编号
  var_dname in varchar2,  --定义IN模式的变量，它存储部门名称
  var_loc in varchar2) is --定义IN模式的变量，它存储部门位置
begin
  insert into dept
  values(num_deptno,var_dname,var_loc);  --向dept表中插入记录
  commit;    --提交数据库
end insert_dept;
/

```

上述代码成功创建了一个存储过程，需要注意的是，参数的类型不能指定长度。

在调用或执行这种IN模式的存储过程时，用户需要向存储过程传递若干参数值，以保证执行部分（即BEGIN部分）有具体的数值参与数据操作。向存储过程传入参数可以有如下3种方式：

指定名称传递

指定名称传递是指在向存储过程传递参数时，需要指定参数名称，即参数名称在左侧，中间是赋值符号“=>”，右侧是参数值，其语法格式如下：

```
pro_name(parameter1=>value1[,parameter2=>value2]…)

```

- parameter1：参数名称。在传递参数值时，这个

参数名称与存储过程中定义的参数顺序无关。

- value1：参数值。在它的左侧不是常规的赋值符号“=”，而是一种新的赋值符号“=>”，需要注意参数值的类型要与参数的定义类型兼容。

在PL/SQL块中调用存储过程insert_dept，然后使用“指定名称”的方式传入参数值，最后执行当前的PL/SQL块，代码及运行结果如下：

```
begin
  insert_dept(var_dname=>'财务部',var_loc=>'烟台',num_deptno=>15);
end;
/
```

在创建存储过程时，其参数的定义顺序是num_deptno、var_dname、var_loc；而在执行存储过程时，参数的传递顺序是var_dname、var_loc、num_deptno。通过对比可以看到，使用“指定名称”的方式传递参数值与参数的定义顺序无关，但与参数个数有关。

按位置传递

指定名称传递参数虽然直观易读，但也有缺点，就是参数过多时，会显得代码冗长，反而变得不容易阅读。可以采取按位置传递参数，采用这种方式时，提供的参数值顺序必须与存储过程中定义的参数顺序相同。

在PL/SQL块中调用存储过程insert_dept，然后使用“按位置传递”的方式向其传入参数值，最后执行当前的PL/SQL块，代码及运行结果如下：

```
begin
  insert_dept(11,'工程部','威海');
end;
/
```

有时候参数过多，不容易记住参数的顺序和类型，可以通过DESC命令来查看存储过程中参数的定义信息，这些信息包括参数名、参数定义顺序、参数类型和参数模式等。

混合方式传递

混合方式就是将前两种方式结合到一起，可以兼顾二者的优点。

在PL/SQL块中调用存储过程insert_dept，然后使用按位置传递方式传入第一个参数值，使用指定名称传递方式传入剩余的两个参数值，最后执行当前的PL/SQL块，代码及运行结果如下：

```
exec insert_dept(12,var_loc=>'济南',var_dname=>'测试部');
```

注意：在某个位置使用指定名称传递方式传入参数值后，其后面的参数值也要使用指定名称传递，因为指定名称传递方式有可能已经破坏了参数原始的定义顺序。

OUT模式参数

OUT模式参数是一种输出类型的参数，表示这个参数在存储过程中已经被赋值，并且这个参数值可以被传递到当前存储过程以外的环境中，关键字OUT位于参数名称之后。
创建一个存储过程，要求定义两个OUT模式的字符类型的参数，将在dept表中检索到的一行部门信息存储到这两个参数中，代码如下：

```
create or replace procedure select_dept(
  num_deptno in number,           --定义IN模式变量，要求输入部门编号
  var_dname out dept.dname%type, --定义OUT模式变量，可以存储部门名称并输出
  var_loc out dept.loc%type) is  --定义OUT模式变量，可以存储部门位置并输出
begin
  select dname,loc
  into var_dname,var_loc
  from dept
  where deptno = num_deptno;  --检索某个部门编号的部门信息
exception
  when no_data_found then     --若SELECT语句无返回记录
     dbms_output.put_line('该部门编号不存在');  --输出信息
end select_dept;
/

```

在上述存储过程（即select_dept）中，定义了两个OUT参数，由于存储过程要通过OUT参数返回值，因此当调用或执行这个存储过程时，需要定义变量来保存这两个OUT参数值；

①在PL/SQL块中调用OUT模式的存储过程

这种方式需要在PL/SQL块的DECLARE部分定义与存储过程中OUT参数兼容的若干变量。
在PL/SQL块中声明若干变量，然后调用select_dept存储过程，并将定义的变量传入该存储过程中，以便接收OUT参数的返回值，代码如下：

```
set serverout on
declare
  var_dname dept.dname%type;  --声明变量，对应过程中的OUT模式的var_dname
  var_loc dept.loc%type;      --声明变量，对应过程中的OUT模式的var_loc
begin
  select_dept(77,var_dname,var_loc); --传入部门编号，然后输出部门名称和位置信息
  dbms_output.put_line(var_dname||'位于：'||var_loc); --输出部门信息
end;
/

```

在上述代码中，把声明的两个变量传入存储过程中，当存储过程执行时，其中的OUT参数会被赋值；当存储过程执行完毕，OUT参数的值会在调用处返回，这样定义的两个变量就可以得到OUT参数被赋予的值，最后这两个值就可以在存储过程外任意使用了。

②使用EXEC命令执行OUT模式的存储过程

当使用EXEC命令时，需要在SQL* Plus环境中使用VARIABLE关键字声明两个变量，用以存储OUT参数的返回值。

使用VARIABLE关键字声明两个变量，分别用来存储部门名称和位置信息，然后使用EXEC命令执行存储过程，并传入声明的两个变量来接收OUT参数的返回值，代码如下：

```
variable var_dname varchar2(50);
variable var_loc varchar2(50);
exec select_dept(11,:var_dname,:var_loc);
```

使用SELECT语句检索并输出变量var_dname和var_loc的值，代码如下:

```
select :var_dname,:var_loc from dual;
```

注意：如果在存储过程中声明了OUT模式的参数，则在执行存储过程时，必须为OUT参数提供变量，以便接收OUT参数的返回值；否则，程序执行后将出现错误。

IN OUT模式参数

在执行存储过程时，IN参数不能被修改，它只能根据被传入的指定值（或是默认值）为存储过程提供数据；而OUT类型的参数只能等待被赋值，它不能像IN参数那样为存储过程本身提供数据。但IN OUT参数可以兼顾其他两种参数的特点，在调用存储过程时，可以从外界向该类型的参数传入值；在执行完存储过程之后，可以将该参数的返回值传给外界。

创建一个存储过程，其中定义一个IN OUT参数，该存储过程用来计算这个参数的平方或平方根，代码如下：

```
create or replace procedure pro_square(
  num in out number,  --计算它的平方或平方根，这是一个IN OUT参数
  flag in boolean) is --计算平方或平方根的标识，这是一个IN参数
  i int := 2;         --表示计算平方，这是一个内部变量
begin
  if flag then           --若为TRUE
    num := power(num,i); --计算平方
  else
    num := sqrt(num);    --计算平方根
  end if;
end;

```

在上述存储过程中，首先定义了一个IN OUT参数，该参数在存储过程被调用时会传入一个数值，然后与另一个IN参数相结合来判断所进行的运算方式（平方或平方根），最后将计算后的平方或平方根保存到此IN OUT参数中。

调用存储过程pro_square，计算某个数的平方或平方根，代码如下：

```
set serverout on
declare
  var_number number;  --存储要进行运算的值和运算后的结果
  var_temp number;    --存储要进行运算的值
  boo_flag boolean;   --平方或平方根的逻辑标记
begin
  var_temp :=9;       --变量赋值
  var_number :=var_temp;
  boo_flag := false;  --FALSE表示计算平方根；TRUE表示计算平方
  pro_square(var_number,boo_flag);  --调用存储过程
  if boo_flag then
    dbms_output.put_line(var_temp ||'的平方是：'||var_number); --输出计算结果
  else
    dbms_output.put_line(var_temp ||'平方根是：'||var_number);
  end if;
end;
/

```

变量var_number在调用存储过程之前是9，而在存储过程被执行完毕之后，该变量的值变为其平方根，这是因为该变量作为存储过程的IN OUT参数被传入和返回。

IN参数的默认值

IN参数的值是在调用存储过程中被传入的，实际上，Oracle支持在声明IN参数的同时给其初始化默认值，这样在存储过程调用时，如果没有向IN参数传入值，则存储过程可以使用默认值进行操作。
创建一个存储过程，定义3个IN参数，并将其中的两个参数各设置一个初始默认值，然后将这3个IN参数的值插入dept表中，代码如下：

```
create or replace procedure insert_dept(
  num_deptno in number,                    --定义存储部门编号的IN参数
  var_dname in varchar2 default '行政部',   --定义存储部门名称的IN参数，并初始化默认值
  var_loc in varchar2 default '青岛') is
begin
  insert into dept values(num_deptno,var_dname,var_loc);  --插入一条记录
end;
/

```

在上述存储过程中，IN参数var_dname和var_loc都有默认值，所以在调用insert_dept存储过程时，可以不向这两个参数传入值，而是使用其默认值（当然也可以传入值）。这种方法建议使用“指定名称传递”的方式传值，这样就不会因为顺序不固定的问题出现混乱。
在PL/SQL块中调用insert_dept存储过程，并且只向该存储过程中传入两个参数值，代码如下：

```
set serverout on
declare
  row_dept dept%rowtype;      --定义行变量，与dept表的一行类型相同
begin
  insert_dept(57,var_loc=>'太原');  --调用insert_dept存储过程，传入参数
  commit;                           --提交数据库
  select *  into row_dept from dept where deptno=57; --查询插入的记录
  dbms_output.put_line('部门名称是：《'||row_dept.dname||'》,位置是：《'||row_dept.loc||'》');
end;
/

```

存储过程insert_dept有3个IN参数，这里只传入两个参数（num_deptno和var_loc）的值，而var_dname参数的值使用默认值“行政部”。

删除存储过程

当一个过程不再被需要时，要将此过程从内存中删除，以释放相应的内存空间，代码如下：

```
DROP PROCEDURE count_num;

```

删除存储过程insert_dept，代码如下:

```
DROP PROCEDURE insert_dept;

```

当一个存储过程已经过时，想重新定义时，不必先删除再创建，只需在CREATE语句后面加上OR REPLACE关键字即可，代码如下:

```
CREATE OR REPLACE PROCEDURE count_num
```

3.5 函数

函数一般用于计算和返回一个值，可以将经常需要使用的计算或功能写成一个函数。函数的调用是表达式的一部分，而过程的调用是一条PL/SQL语句。函数与过程在创建的形式上有些相似，也是编译后放在内存中供用户使用，只不过调用函数时要用表达式，而不像过程只需要调用过程名。另外，函数必须要有一个返回值，过程则没有。

创建函数

函数的创建语法与存储过程比较类似，也是一种存储在数据库中的命名程序块。函数可以接收零或多个输入参数，并且必须有返回值（这一点存储过程是没有的）。定义函数的语法格式如下：

```
CREATE [OR REPLACE] FUNCTION fun_name[(parameter1[,parameter2]…) RETURN data_type IS [inner_variable]
BEGIN
  plsql_ sentence;
[EXCEPTION]
  [dowith _ sentences;]
END [fun_name];

```

- fun_name：函数名称，如果数据库中已经存在了此名称，则可以指定OR REPLACE关键字，这样新的函数将覆盖原来的函数。
- parameter1：函数的参数，这是一个可选项，因为函数可以没有参数。
- data_type：函数的返回值类型，这是一个必选项。前面要使用RETURN关键字来标明。
- inner_variable：函数的内部变量，它有别于函数的参数，这是一个可选项。
- plsql_ sentence：PL/SQL语句，它是函数主要功能的实现部分，也就是函数的主体。
- dowith _ sentences：异常处理代码，也是PL/SQL语句，这是一个可选项。

由于函数有返回值，因此在函数主体部分（即BEGIN部分）必须使用RETURN语句返回函数值，并且要求返回值的类型要与函数声明时的返回值类型（即data_type）相同。
定义一个函数，用于计算emp表中指定部门的平均工资，代码如下：

```
create or replace function get_avg_pay(num_deptno number) return number is
  num_avg_pay number;  --保存平均工资的内部变量
begin
  select avg(sal) into num_avg_pay from emp where deptno=num_deptno; --某个部门的平均工资
  return(round(num_avg_pay,2));  --返回平均工资
exception
  when no_data_found then    --若此部门编号不存在
    dbms_output.put_line('该部门编号不存在');
    return(0);    --返回平均工资为0
end;
/

```

调用函数

由于函数有返回值，因此在调用函数时，必须使用一个变量来保存函数的返回值，这样函数和这个变量就组成了一个赋值表达式。
调用函数get_avg_pay，计算部门编号为10的员工的平均工资并输出，代码如下：

```
set serveroutput on
declare
  avg_pay number;   --定义变量，存储函数返回值
begin
  avg_pay:=get_avg_pay(10);  --调用函数，并获取返回值
  dbms_output.put_line('平均工资是：'||avg_pay);  --输出返回值，即员工平均工资
end;
/
```

删除函数

删除函数的操作比较简单，使用DROP FUNCTION命令，其后面跟着要删除的函数名称，其语法格式如下：

```
DROP FUNCTION fun_name;

```

参数fun_name表示要删除的函数名称。
使用DROP FUNCTION命令删除函数get_avg_pay，代码如下：

```
drop function get_avg_pay;

```

当一个函数已经过时，想重新定义时，也不必先删除再创建，同样只需要在CREATE语句后面加上OR REPLACE关键字即可，代码如下:

```
CREATE OR REPLACE FUNCTION fun_name;

```

3.6 触发器

触发器可以看作一种特殊的存储过程，它定义了一些在数据库相关事件（如INSERT、UPDATE、CREATE等事件）发生时应执行的“功能代码块”，通常用于管理复杂的完整性约束，或监控对表的修改，或通知其他程序，甚至可以实现对数据的审计功能。

触发器简介

触发器是通过触发事件来执行的（存储过程的调用或执行是由用户或应用程序进行的）。能够引起触发器运行的操作就被称为**触发事件**，如执行DML语句（使用INSERT、UPDATE、DELETE语句对表或视图执行数据处理操作）、执行DDL语句（使用CREATE、ALTER、DROP语句在数据库中创建、修改、删除模式对象）、引发数据库系统事件（如系统启动或退出、产生异常错误等）、引发用户事件（如登录或退出数据库操作）等，以上这些操作都可以引起触发器的运行。
触发器的语法格式：

```
CREATE [OR REPLACE] TRIGGER tri_name
  [BEFORE | AFTER | INSTEAD OF] tri_event
  ON table_name | view_name | user_name | db_name
   [FOR EACH ROW [WHEN tri_condition]
 BEGIN
 plsql_sentences;
 END tri_name;

```

- TRIGGER：表示创建触发器的关键字，就如同创建存储过程的关键字PROCEDURE一样。
- BEFORE | AFTER | INSTEAD OF：表示触发时机的关键字。BEFORE表示在执行DML等操作之前触发，这种方式能够防止某些错误操作发生而便于回滚或实现某些业务规则；AFTER表示在DML等操作之后发生，这种方式便于记录该操作或某些事后处理信息；INSTEAD OF表示触发器为替代触发器。
- ON：表示操作的数据表、视图、用户模式和数据库等，当对它们执行某种数据操作（如对表执行INSERT、ALTER、DROP等操作）时，将引起触发器的运行。
- FOR EACH ROW：指定触发器为行级触发器，当DML语句对每一行数据进行操作时都会引起该触发器的运行。如果未指定该条件，则表示创建语句级触发器，这时无论数据操作影响多少行，触发器都只会执行一次。
- tri_name：触发器的名称，如果数据库中已经存在此名称，则可以通过指定“or replace”关键字将原来的触发器用新的触发器覆盖。
- tri_event：触发事件，如常用的有INSERT、UPDATE、DELETE、CREATE、ALTER、DROP等。
- table_name | view_name | user_name | db_name：分别表示操作的数据表、视图、用户模式和数据库，当对它们执行某些操作时，将引起触发器的运行。

- WHEN tri_condition：这是一个触发条件子句，其中WHEN是关键字，tri_condition表示触发条件表达式。只有当该表达式的值为TRUE时，遇到触发事件才会自动执行触发器，使其执行触发操作，否则即便是遇到触发事件也不会执行触发器。
- plsql_sentences：PL/SQL语句，它是触发器功能实现的主体。

Oracle支持的触发器分为以下5种类型：

- 行级触发器：当DML语句对每一行数据进行操作时都会引起该触发器的运行。
- 语句级触发器：无论DML语句影响多少行数据，其所引起的触发器仅执行一次。
- 替换触发器：该触发器是定义在视图上的，而不是定义在表上，它是用来替换所使用实际语句的触发器。
- 用户事件触发器：是指与DDL操作或用户登录、退出数据库等事件相关的触发器，如用户登录到数据库或使用ALTER语句修改表结构等事件的触发器。
- 系统事件触发器：是指在Oracle数据库系统的事件中进行触发的触发器，如Oracle实例的启动与关闭。

语句级触发器

语句级触发器，是针对一条DML语句引起的触发器执行。在语句级触发器中，不使用FOR EACH ROW子句，也就是说无论数据操作影响多少行，触发器都只会执行一次。
使用触发器在scott模式下针对dept表的各种操作进行监控，为此首先需要创建一个日志表dept_log，它用于存储对dept表的各种数据操作信息，如操作种类（如插入、修改、删除操作）、操作时间等，在scott模式下创建dept_log数据表，并在其中定义两个字段，分别用来存储操作种类信息和操作日期，代码如下：

```
create table dept_log
(
 operate_tag varchar2(10),  --定义字段，存储操作种类信息
 operate_time date          --定义字段，存储操作日期
);

```

创建一个触发器tri_dept，该触发器在INSERT、UPDATE和DELETE事件下都可以被触发，操作的数据对象是dept表，并且在触发器执行时输出对dept表所做的具体操作，代码如下：

```
create or replace trigger tri_dept
  before insert or update or delete
  on dept   --创建触发器，当dept表发生插入、修改、删除操作时，将引起该触发器执行
declare
  var_tag varchar2(10);    --声明一个变量，存储对dept表执行的操作类型
begin
  if inserting then      --当触发事件是INSERT时
    var_tag := '插入';    --标识插入操作
  elsif updating then    --当触发事件是UPDATE时
    var_tag := '修改';    --标识修改操作
  elsif deleting then    --当触发事件是DELETE时
    var_tag := '删除';    --标识删除操作
  end if;
  insert into dept_log
  values(var_tag,sysdate); --向日志表中插入对dept表的操作信息
end tri_dept;
/

```

在上述代码中，使用BEFORE关键字来指定触发器的“触发时机”，它指定当前的触发器在DML语句执行之前被触发，这使得它非常适合于强化安全性、启用业务逻辑和进行日志信息记录。当然也可以使用AFTER关键字，它通常被用于记录该操作或者做某些事后处理工作。具体使用哪一种关键字，要根据实际需要而定。

在数据表dept中实现插入、修改、删除3种操作，以便引起触发器tri_dept的执行，代码如下：

```
insert into dept values(55,'业务部','上海');
update dept set loc='杭州' where deptno=55;
delete from dept where deptno=55;

commit;
select * from dept_log;
```

上述代码对dept表执行了3次DML操作，这样根据tri_dept触发器自身的设计情况，其会被触发3次，并且会向dept_log表中插入3条操作记录。

行级触发器

行级触发器会针对DML操作所影响的每一行数据都执行一次触发器，创建这种触发器时，必须在语法中使用FOR EACH ROW这个选项；使用行级触发器的一个典型应用就是给数据表生成主键值。
为了使用行级触发器生成数据表中的主键值，首先需要创建一个带有主键列的数据表。
在scott模式下，创建一个用于存储商品种类的数据表，其中包括商品序号列和商品名称列，代码如下；

```
create table goods
(
  id int primary key,  --设置id为主键
  good_name varchar2(50)
);

```

为了给goods表的id列生成不能重复的有序值，这里需要创建一个序列，使用CREATE SEQUENCE语句创建一个序列，命名为seq_id，代码如下：

```
create sequence seq_id;
```

上述代码创建了序列seq_id，用户可以在PL/SQL程序中调用它的NEXTVAL属性来获取一系列有序的数值，这些数值就可以被作为goods表的主键值。
创建一个行级触发器，该触发器在向数据表goods中插入数据时被触发，并且在该触发器的主体中实现设置goods表中的id列的值，代码如下：

```
create or replace trigger tri_insert_good
  before insert
  on goods        --关于goods数据表，在向其插入新记录之前，引起该触发器的运行
  for each row    --创建行级触发器
begin
  select seq_id.nextval
  into :new.id
  from dual;     --从序列中生成一个新的数值，赋值给当前插入行的id列
end tri_insert_good;
/

```

在上述代码中，为了创建行级的触发器，使用了FOR EACH ROW选项；为了给goods表中的当前插入行的id列赋值，这里使用了:new.id关键字—也被称为“列标识符”，这个列标识符用来指向新行的id列，给它赋值，就相当于给当前行的id列赋值。

在行级触发器中，可以通过“列标识符”来访问当前正在受到影响（添加、删除、修改等操作）的数据行，列标识符可以分为“原值标识符”和“新值标识符”。原值标识符用于标识当前行的某个列的原始值，记作:old.column_name（如:old.id），通常在UPDATE语句和DELETE语句中被使用，因为在INSERT语句中新插入的行没有原始值；新值标识符用于标识当前行的某个列的新值，记作:new.column_name（如:new.id），通常在INSERT语句和UPDATE语句中被使用，因为在DELETE语句中被删除的行无法产生新值。

在触发器创建完毕之后，可以通过向goods表中插入数据来验证触发器是否被执行，同时能够验证该行级触发器是否能够使用序列为表的主键赋值。

向goods表中插入两条记录，其中一条记录不指定id列的值，由序列seq_id来产生；另一条记录指定id的值，代码如下：

```
insert into goods(good_name) values('香蕉');
insert into goods(id,good_name) values(9,'柚子');


commit;
select * from goods;

```

可以看到，无论是否指定id列的值，数据的插入都是成功的，而且即使第二次插入数据指定了id的值为9，也没有起作用，这是因为在触发器中将序列seq_id的NEXTVAL属性值赋给了:new.id列标识符，这个列标识符的值就是当前插入行的id列的值，并且NEXTVAL属性值是连续不间断的。

替换触发器

替换触发器即INSTEAD OF触发器，它的“触发时机”关键字是INSTEAD OF，而不是BEFORE或AFTER。与其他类型触发器不同的是，替换触发器是定义在视图上的，而不是定义在表上。由于视图是由多张基表连接组成的逻辑结构，因此一般不允许进行DML操作（如INSERT、UPDATE、DELETE等操作），当为视图编写替换触发器后，对视图的DML操作实际上就变成了执行触发器中的PL/SQL块，这样就可以通过在替换触发器中编写适当的代码对构成视图的各张基表进行操作。
在scott模式下创建一个检索员工信息的视图，该视图的基表包括dept表（部门表）和emp表（员工表），代码如下：

```
--创建视图 view_emp_dept
create view view_emp_dept
  as select empno,ename,dept.deptno,dname,job,hiredate
     from emp,dept
     where emp.deptno = dept.deptno;

```

尝试向view_emp_dept视图中插入数据，会提示“ORA-01776: 无法通过联接视图修改多个基表”；

创建一个关于view_emp_dept视图的替换触发器，在该触发器的主体中实现向emp表和dept表中插入两行相互关联的数据，代码如下：

```
create or replace trigger tri_insert_view
  instead of insert
  on view_emp_dept    --创建一个关于view_emp_dept视图的替换触发器
  for each row        --是行级视图
declare
  row_dept number;
begin
  SELECT count(*) INTO row_dept FROM dept WHERE deptno = :new.deptno;--检索指定部门编号的记录行
  if row_dept = 0 then     --判断是否存在该部门编号的记录
     insert into dept(deptno,dname)
     values(:new.deptno,:new.dname);   --向dept表中插入数据
  end if;
  insert into emp(empno,ename,deptno,job,hiredate)
  values(:new.empno,:new.ename,:new.deptno,:new.job,:new.hiredate);  --向emp表中插入数据
end tri_insert_view;
/

```

在上述触发器的主体代码中，如果新插入行的部门编号(deptno)不在dept表中，则首先向dept表中插入关于新部门编号的数据行，然后再向emp表中插入记录行，这是因为emp表的外键值(emp.deptno)是dept表的主键值(dept.deptno)。

成功创建触发器tri_insert_view之后，在向view_emp_dept视图中插入数据时，Oracle就不会产生错误信息，而是引起触发器tri_insert_view的运行，从而实现向emp表和dept表中插入两行数据。

向视图view_emp_dept中插入一条记录，然后检索插入的记录行，代码如下：

```
--向视图插入一条数据
insert into view_emp_dept (empno,ename,deptno,dname,job,hiredate) values (7777,'小明',99,'测试部','测试工程师',sysdate);

--查询deptno=99的数据
select * from dept where deptno=99;
--查询empno=7777的数据
select * from emp where empno=7777;

```

dept表之前是没有部门编码deptno=99的记录的，触发器中的程序向dept表中插入deptno=99的数据，然后又向emp表中插入一条记录。

用户事件触发器

用户事件触发器是因进行DDL操作或用户登录、退出等操作而引起运行的一种触发器，引起该类型触发器运行的常见用户事件包括CREATE、ALTER、DROP、ANALYZE、COMMENT、GRANT、REVOKE、RENAME、TRUNCATE、SUSPEND、LOGON和LOGOFF等。
使用CREATE TABLE语句创建一个日志信息表，该表保存的日志信息包括数据对象、数据对象类型、操作行为、操作用户和操作日期等，代码如下：

```
create table ddl_oper_log
(
  db_obj_name varchar2(20),  --数据对象名称
  db_obj_type varchar2(20),  --对象类型
  oper_action varchar2(20),  --具体ddl行为
  oper_user varchar2(20),    --操作用户
  oper_date date             --操作日期
);

```

创建一个关于scott用户的DDL操作（这里包括CREATE、ALTER和DROP）的触发器，然后将DDL操作的相关信息插入ddl_oper_log日志表中，代码如：

```
create or replace trigger tri_ddl_oper
  before create or alter or drop
  on scott.schema    --在scott模式下，在创建、修改、删除数据对象之前将引发该触发器运行
begin
  insert into ddl_oper_log values(
    ora_dict_obj_name,     --操作的数据对象名称
    ora_dict_obj_type,     --对象类型
    ora_sysevent,          --系统事件名称
    ora_login_user,        --登录用户
    sysdate);
end;
/

```

当向日志表ddl_oper_log中插入数据时，使用了若干个事件属性，其含义如下：

- ora_dict_obj_name：获取DDL操作所对应的数据库对象。
- ora_dict_obj_type：获取DDL操作所对应的数据库对象的类型。
- ora_sysevent：获取触发器的系统事件名。
- ora_login_user：获取登录用户名。

通过上述4个事件属性值和sysdate系统属性就可以将scott用户的DDL操作信息获取出来，然后再把这些信息保存到ddl_oper_log日志表中。

在scott模式下，首先创建一张数据表和一个视图，然后删除视图和修改数据表，最后使用SELECT语句查看ddl_oper_log日志表中的DDL操作信息，代码如下：

```
--创建tb_test表
create table tb_test(id number);
--创建view_test视图
create view view_test as select empno,ename from emp;
--修改tb_test表
alter table tb_test add(name varchar2(10));
--删除视图view_test
drop view view_test;
--查询ddl_oper_log表
select * from ddl_oper_log;

```

可以看到，用户scott的DDL操作信息都被存储到ddl_oper_log日志表中，这些信息就是由DDL操作引起触发器运行而保存到日志表中的。

删除触发器

当一个触发器不再使用时，要从内存中删除它，例如：

```
--删除名为tri_dept的触发器
DROP TRIGGER tri_dept;

```

当一个触发器已经过时，想重新定义时，不必先删除再创建，只需在CREATE语句后面加上OR REPLACE关键字即可，代码如下：

```
CREATE OR REPLACE TRIGGER my_trigger;
```

3.7 程序包

程序包由PL/SQL程序元素（如变量、类型）和匿名PL/SQL块（如游标）、命名PL/SQL块（如存储过程和函数）组成。程序包可以被整体加载到内存中，从而大大加快其组成部分的访问速度。
在PL/SQL程序中使用DBMS_OUTPUT. PUT_LINE语句就是程序包的一个具体应用。其中，DBMS_OUTPUT是程序包，而PUT_LINE就是其中的一个存储过程。程序包通常由规范和包主体组成。

程序包的规范

程序包规范规定了在程序包中可以使用哪些变量、类型、游标和子程序（指各种命名的PL/SQL块），需要注意的是，程序包一定要在包主体之前被创建。其语法格式如下：

```
CREATE [OR REPLACE ] PACKAGE pack_name IS [declare _variable];
[declare _type];
[declare _cursor];
[declare _function];
[declare _procedure];
END [pack_name];
```

- pack_name：程序包的名称，如果数据库中已经存在了此名称，则可以指定OR REPLACE关键字，这样新的程序包将覆盖原来的程序包。
- declare _variable：规范内声明的变量。
- declare _type：规范内声明的类型。
- declare _cursor：规范内定义的游标。
- declare _function：规范内声明的函数，但仅定义参数和返回值类型，不包括函数体。
- declare _procedure：规范内声明的存储过程，但仅定义参数，不包括存储过程主体。

创建一个程序包规范，首先在该程序包中声明一个可以获取指定部门的平均工资的函数，然后声明一个可以实现按照指定比例上调指定职务的工资的存储过程，代码如下：

```
create or replace package pack_emp is
  function fun_avg_sal(num_deptno number) return number;                --获取指定部门的平均工资
  procedure pro_regulate_sal(var_job varchar2,num_proportion number);   --按照指定比例上调指定职务的工资
end pack_emp;
/
```

从上述代码中可以看到，在规范中声明的函数和存储过程只有头部的声明，而没有函数体和存储过程主体。

注意：只定义了规范的程序包还不可以被使用，此时如果在PL/SQL块中通过程序包的名称来调用其中的函数或存储过程，则Oracle将会产生错误提示。

程序包的主体

程序包主体包含了在规范中声明的游标、过程和函数的实现代码，另外，也可以在程序包的主体中声明一些内部变量。程序包主体的名称必须与规范的名称相同，这样通过这个相同的名称，Oracle就可以将规范和主体结合在一起组成程序包，并实现一起编译代码。在实现函数或存储过程主体时，可以将每一个函数或存储过程作为一个独立的PL/SQL块来处理。

与创建规范不同的是，创建程序包主体使用CREATE PACKAGE BODY语句，而不是CREATE PACKAGE语句，这一点需要读者注意。创建程序包主体的代码如下：

```
CREATE [OR REPLACE] PACKAGE BODY pack_name IS
  [inner_variable]
  [cursor_body]
  [function_title]
  {BEGIN
    fun_plsql;
  [EXCEPTION]
    [dowith _ sentences;]
  END [fun_name]}
  [procedure_title]
  {BEGIN
    pro_plsql;
  [EXCEPTION]
    [dowith _ sentences;]
  END [pro_name]}
…
END [pack_name];

```

- pack_name：程序包的名称，要求与规范对应的程序包名称相同。
- inner_variable：程序包主体的内部变量。
- cursor_body：游标主体。
- function_title：从规范中引入的函数头部声明。
- fun_plsql：PL/SQL语句，这里是函数主要功能的实现部分。从BEGIN到END部分就是函数的BODY。
- dowith _ sentences：异常处理语句。
- fun_name：函数的名称。
- procedure_title：从规范中引入的存储过程头部声明。
- pro_plsql：PL/SQL语句，这里是存储过程主要功能的实现部分。从BEGIN到END部分就是存储过程的BODY。
- pro_name：存储过程的名称。

创建程序包pack_emp的主体，在该主体中实现与规范中声明的函数和存储过程对应，代码如下：

```
create or replace package body pack_emp is
  function fun_avg_sal(num_deptno number) return number is  --引入“规范”中的函数
    num_avg_sal number;    --定义内部变量
  begin
   select avg(sal)
   into num_avg_sal
   from emp
   where deptno = num_deptno;  --计算某个部门的平均工资
   return(num_avg_sal);        --返回平均工资
  exception
   when no_data_found then     --若未发现记录
     dbms_output.put_line('该部门编号不存在员工记录');
   return 0;                   --返回0
  end fun_avg_sal;
  procedure pro_regulate_sal(var_job varchar2,num_proportion number) is --引入“规范”中的存储过程
  begin
   update emp
   set sal = sal*(1+num_proportion)
   where job = var_job;   --为指定的职务调整工资
  end pro_regulate_sal;
end pack_emp;
/

```

在创建了程序包的规范和主体之后，就可以像普通的存储过程和函数一样实施调用了。
创建一个匿名的PL/SQL块，然后通过程序包pack_emp调用其中的函数fun_avg_sal和存储过程pro_regulate_sal，并输出函数的返回结果，代码如下：

```
set serveroutput on

declare
  num_deptno emp.deptno%type;     --定义部门编号变量
  var_job emp.job%type;           --定义职务变量
  num_avg_sal emp.sal%type;       --定义工资变量
  num_proportion number;          --定义工资调整比例变量
begin
  num_deptno:=10;                 --设置部门编号为10
  num_avg_sal:=pack_emp.fun_avg_sal(num_deptno);    --计算部门编号为10的平均工资
  dbms_output.put_line(num_deptno||'号部门的平均工资是：'||num_avg_sal);  --输出平均工资
  var_job:='SALESMAN';            --设置职务名称
  num_proportion:=0.1;            --设置调整比例
  pack_emp.pro_regulate_sal(var_job,num_proportion); --调整指定部门的工资
end;
 /

```

删除程序包

与函数和过程一样，当一个程序包不再被使用时，要从内存中删除它，删除程序包pack_emp，代码如下：

```
drop package pack_emp;

```

当一个程序包已经过时，想重新定义时，不必先删除再创建，同样只需在CREATE语句后面加上OR REPLACE关键字即可，例如：

```
create or replace package my_package
```

3.8 事务

https://note.youdao.com/s/5YdqK6dz

3.9 SQL优化

https://note.youdao.com/s/IHezR7hr

4. Oracle Database23ai新特性

https://note.youdao.com/s/WrfU26zz