sqoracle数据库

oracle安装

plsql安装

Oracle开启归档

激活码：

product code: ke4tv8t5jtxz493kl8s2nn3t6xgngcmgf3

serial Number: 264452

password: xs374ca

修改用户的权限

 image-20201105174800920

修改对用户的限额

 image-20201105174800920

plsq停止正在输出的数据：shift + esc

---

sql基础

DML: Data Manipulation Language 数据操纵语言

> DML用于查询与修改数据记录，包括如下SQL语句：
>
> INSERT：添加数据到数据库中
>
> lUPDATE：修改数据库中的数据
>
> DELETE：删除数据库中的数据
>
> **SELECT**：选择（查询）数据
>
> SELECT是SQL语言的基础，最为重要

DDL: Data Definition Language 数据定义语言

> DDL用于定义数据库的结构，比如创建、修改或删除数据库对象，包括如下SQL语句：

> **CREATE** TABLE：创建数据库表
>
> **ALTER** TABLE：更改表结构、添加、删除、修改列长度
>
> **DROP** TABLE：删除表
>
> CREATE INDEX：在表上建立索引
>
> DROP INDEX：删除索引

DCL: Data Control Language 数据控制语言

> DCL用来控制数据库的访问，包括如下SQL语句：
>
> **GRANT**：授予访问权限
>
> **REVOKE**：撤销访问权限
>
> **COMMIT**：提交事务处理
>
> **ROLLBACK**：事务处理回退:指定回滚到保持点rollback to (savepoint) 保存点名称
>
> **SAVEPOINT**：设置保存点:savepoint 保存点名称
>
> **LOCK**：对数据库的特定部分进行锁定

---

select基本语句

> SQL 语言**大小写不敏感。**
>
> SQL 可以写在一行或者多行
>
> **关键字不能被缩写也不能分行**
>
> 各子句一般要分行写。
>
> 使用缩进提高语句的可读性
>
> 为列起别名，有四种方式
>
> 由多个单词组成的别名就用双引号引起来
>
> 只有起别名时是双引号其他地方都是单引号
>
> 1、select (to_char(sysdate,'yyyy') * 2) as "今年" from dual;
>
> 2、select (to_char(sysdate,'yyyy') * 2) as 今年 from dual;
>
> 3、select (to_char(sysdate,'yyyy') * 2) 今年 from dual;
>
> 4、select (to_char(sysdate,'yyyy') * 2) "今年" from dual;

定义空值

没有值，再进行运算时就不会进行运算；如果参与运算那么结果就是null

> **空值是无效的，未指定的，未知的或不可预知的值**
>
> **空值不是空格或者0**

算术运算符

数字和日期使用的算数运算符

乘除的优先级高于加减。

同一优先级运算符从左向右执行。

括号内的运算先执行

日期(sysdate)不能不能做乘除

操作符	描述
+	加
-	减
*	乘
/	除

每个员工工资；一年的工资+年终奖1000

```
select last_name, salary, 12 * salary + 1000 from system.employees;
```

查询今天，明天，下个月今天的语句

```
select sysdate 今天, sysdate + 1 明天, sysdate + 30 下个月今天 from dual;

获取年份
select to_char(sysdate,'yyyy') 今年 from dual;

获取年月日
select to_char(sysdate,'yyyy-mm-dd') from dual;

获取时分秒----12小时制
select to_char(sysdate, 'hh:mi:ss') from dual ;  --12小时制

24小时制
select to_char(sysdate,'hh24:mi:ss')  from dual ; --24小时制
```

查看表有多少列

```
desc 表名;    -- desc是sql plus的关键字   desc = describe
```

查询指定的列的数据

```
select 列名1, 列名2, 列名3... from 表名;
```

连接符

> 把列与列，列与字符连接在一起。
>
> 用‘||’表示。
>
> 可以用来‘合成’列。

```
select last_name || ' `s job_id is ' || job_id from system.employees;
```

去重查询

```
select distinct department_id from system.employees;

去重，且不查询空值
select distinct department_id from system.employees where department_id is not
null;
```

过滤和排序数据

匹配日期查询

```
select last_name, hire_date from system.employees where hire_date = '7-6月-1994';
-- 日-月-年；这种方式易出错

-- 一般用这种
select last_name, hire_date from system.employees where to_char(hire_date,
'yyyy-mm-dd') = '1994-06-07';

-- 查询指定年份的数据
select last_name, hire_date from system.employees where hire_date like '%94%';
-- 或者
select last_name, hire_date from system.employees where to_char(hire_date,
'yyyy') = 1994;

-- 将字段转换为yyyyy-mm-dd形式比较
select last_name, hire_date from system.employees where to_char(hire_date,
'yyyy-mm-dd') between '1998-02-01' and '1998-05-01';
```

比较运算

赋值：　:=

操作符	含义
=	等于（不是==）
>	大于
>=	大于、等于
<	小于
<=	小于、等于
<>	不等于（也可以是!=）

工资大于等于4k且小于7k

```
select last_name, hire_date
from system.employees
where salary >= 4000 && salary <7000
```

其它比较运算

操作符	含义
BETWEEN ...AND...	在两个值之间（包含边界）
IN(set)	等于值列表中的一个
LIKE	模糊查询
IS NULL	空值

between　　在...之间　　不在...之间：not between ... and ...

```
-- 工资在4k-7k之间的数据(包含前后)
select last_name, hire_date
from system.employees
where salary between 4000 and 7000

等于下面

select last_name, hire_date
from system.employees
where salary >= 4000 && salary <= 7000
```

in　　包含括号内的某些值

```
-- 部门编号是70-90(包含前后)直接的数据
select last_name, department_id, salary
from system.employees
where department_id in (70, 80, 90)

等于

select last_name, department_id, salary
from system.employees
where department_id = 90 or department_id = 80 or department_id = 70
```

like　字段包含指定的值

```
-- 员工名字包含a的员工信息

select last_name, department_id, salary
from system.employees
where last_name like '%a%'

-- 员工名字第二位是a的员工信息
select last_name, department_id, salary
from system.employees
where last_name like '_a%'

-- 姓名中有字母a和e的员工姓名
select last_name from system.employees where last_name like '%a%e%' or last_name
like '%a%e%';

-- 员工名字包含下划线的员工信息；需要转义；可以是\也可以是#
select last_name, department_id, salary
from system.employees
where last_name like '%\_%' escape '\'
```

is null

```
-- 员工奖金率是null的数据
-- 查询非null就是is not null

select last_name, department_id, salary, commission_pct
from system.employees
where commission_pct is null
```

逻辑运算

操作符	含义
AND	逻辑并
OR	逻辑或
NOT	逻辑否

and

```
select last_name, department_id, salary, commission_pct
from system.employees
where department_id = 80 and salary <= 8000
```

优先级

可以使用括号改变优先级顺序

优先级	
1	算术运算符
2	连接符
3	比较符
4	IS [NOT] NULL**,** LIKE**,** [NOT] IN
5	[NOT] BETWEEN
6	NOT
7	AND
8	OR

order by的desc和asc排序查询

默认是asc由低到高排序

```sql
-- 查询的信息根据工资从高到低排序desc       由低到高是asc

select last_name, department_id, salary, commission_pct
from system.employees
where department_id = 80
order by salary desc


-- 工资相同，就按照名字由低到高排：a-z
select last_name, department_id, salary, commission_pct
from system.employees
where department_id = 80
order by salary desc, last_name asc
```

单行函数

> 单行函数:
>
> - 操作数据对象
> - 接受参数返回一个结果
> - **只对一行进行变换**
> - **每行返回一个结果**
> - 可以转换数据类型
> - 可以嵌套
> - 参数可以是一列或一个值

大小写控制函数

函数	结果
LOWER ('SQL Course')	sql course
UPPER ('SQL Course')	sql COURSE
INITCAP('SQL Course')	sql Course

```sql
-- lower转换为小写        upper转换为大写              initcap把首字母转换为大写
select lower('ATGUIGU'), UPPER('AtGuiGu Java'), initcap('AtGuiGu java')
from dual;


-- 将指定字段都转换为小写，方便查询
select * from system.employees where lower(last_name) = 'king'
```

字符控制函数

函数	结果
CONCAT('Hello', 'World') 字符串连接	HelloWorld
SUBSTR('HelloWorld',1,5) 字符串截取	Hello
LENGTH('HelloWorld') 得到字符串长度	10
INSTR('HelloWorld', 'W') 获得字符在字符串中首次出现的下标(从1开始)	6
LPAD(salary,10,'*') 字段占指定长度，占不满就在左边以*补充	* * * * *24000
RPAD(salary, 10, '*') 字段占指定长度，占不满就在右边以*补充	24000* * * * *
TRIM('H' FROM 'HelloWorld') 去除指定字符串的左右指定字符	elloWorld
REPLACE(' abcd ',' b','m ') 替换指定字符串的所有字符，为指定字符	amcd

```
---     concat将字符串拼成一个       substr下标(从1开始)截取指定下标的后面四个字符串
length获取字符串长度
select concat('hello', 'world'), substr('helloworld', 2, 4)
, length('helloworld') from dual;


-- 获得字符在字符串中首次出现的下标(从1开始)    没有返回0
select instr('helloJava', 'l') from dual;

-- 员工的工资共十位长度，占不满就使用*表示；往左边补          rpad是往右边补
select employee_id, last_name, lpad(salary, 10, '*')
from system.employees

-- 去除指定字符串左右的h字符
select trim('h' from 'hhhhellhoworldhhh') from dual;

-- 替换指定字符串的所有字符，为指定字符
select replace('abcdab', 'b', 'm') from dual
```

数字函数

ROUND: 四舍五入

```
--      保留两位小数       没指定保留的位数，默认是0，只取了整数      负数相当于去掉是小数，且
在整数的后两位进行四舍五入，小于5为0
select round(435.45, 2), round(435.45), round(435.45, -2) from dual;

--      四舍五入435.5       保留整数435      在整数中的最后以为四舍五入为440
select round(435.45, 1), round(435.45), round(435.45, -1) from dual;
```

TRUNC: 截断

```
-- 无论是什么样的数，舍去几个就去掉最后的几个；没有指定就是0；负数相当于去掉小数，在整数的最后面去除指定个数，指定的几位都为0
select trunc(435.45, 1), trunc(435.45), trunc(435.45, -1) from dual;
--          435.4               435             430
```

MOD: 求余

```
-- 1100除300，余200
select mod(1100, 300) from dual;
```

日期函数

函数	描述
MONTHS_BETWEEN	两个日期相差的月数
ADD_MONTHS	向指定日期中加上若干月数
NEXT_DAY	指定日期的下一个星期 * 对应的日期
LAST_DAY	本月的最后一天
ROUND	日期四舍五入
TRUNC	日期截断

Oracle 中的日期型数据实际含有两个值: 日期和时间

日期

```
-- 今天的日期          今天日期 + 1天    今天日期 - 3天
select sysdate, sysdate + 1, sysdate - 3 from dual;

-- 显示指定格式的系统时间
select to_char(sysdate, 'yyyy"年"mm"月"dd"日" hh:mi:ss') from dual
```

日期的数学运算

> 在日期上加上或减去一个数字结果仍为日期。
>
> **两个日期相减返回日期之间相差的 天数。**
>
> 日期不允许做加法运算，无意义
>
> 可以用数字除24来向日期中加上或减去天数。

```
-- 查询员工到公司多少天了
select employee_id, last_name, sysdate - hire_date worked_days
from system.employees

-- 有小数就截取掉
select employee_id, last_name, trunc(sysdate - hire_date) worked_days
from system.employees

-- 查询员工到公司多少个月
select employee_id, last_name, months_between(sysdate, hire_date) from
system.employees;
```

```
-- 系统时间+2个月                    系统时间 - 3个月            往后算最近的星期日的日期
select add_months(sysdate, 2), add_months(sysdate, -3), next_day(sysdate, '星期
日') from dual

-- 当前日期的最后一天
select last_day(sysdate) from dual;


-- 公司员工中，hire_date每个月倒数第二天入职的人员
select last_name, hire_date
from system.employees
where hire_date = last_day(hire_date) - 1

-- 按照当前日期的天数四舍五入，对月份天数过半的规则      也是对月份进行四舍五入      舍弃的掉小
时后面的位数，都为0
select round(sysdate, 'month'), round(sysdate, 'mm'), trunc(sysdate, 'hh')
from dual
```

转换函数

分为隐式和显式

隐式：以下数据类型都可以自动相互转换

源数据类型	目标数据类型
VARVHAR2 or CHAR	NUMBER
VARVAHR2 or CHAR	DATE
NUMBER	VARCHAR2
DATE	VARHCAR2

```
-- 等于15，会自动进行转换为number进行运算
select '12' +  3 from dual;
```

显式----**字符串和日期转换**

```
-- 将指定列，转换为指定字符串格式，进行比较
select employee_id, hire_date
from system.employees
where to_char(hire_date, 'yyyy-mm-dd') = '1994-06-07'

-- 也可以把字符串转换为日期类型进行比较
select employee_id, hire_date
from system.employees
where to_date('1994-06-07', 'yyyy-mm-dd') = hire_date

-- 穿插一些字符，就要用双引号包裹
select employee_id, hire_date
from system.employees
where to_char(hire_date, 'yyyy"年"mm"月"dd"日"') = '1994年06月07日'

-- 要显示想要的格式，在想要转换的字段处转换
select employee_id, to_char(hire_date, 'yyyy"年"mm"月"dd"日"')
from system.employees
where to_char(hire_date, 'yyyy"年"mm"月"dd"日"') = '1994年06月07日'
```

字符串和数字类型转换

```
-- 数字显示为指定字符串格式
select to_char(1234567.89, '999,999,999.99') from dual

-- 需要不足位数时，补充0，就用这种格式
select to_char(1234567.89, '000,999,999.99') from dual

-- 使用$表示金额
select to_char(1234567.89, '$000,999,999.99') from dual

-- 使用本地的符号表示金额
select to_char(1234567.89, 'L000,999,999.99') from dual


-- 将金额类型的字符串，转换为数字
select to_number('￥001,234,567.89', 'L000,000,999.99') from dual
```

通用函数

> 这些函数**适用于任何数据类型，同时也适用于空值**:
>
> - NVL (expr1, expr2)
> - NVL2 (expr1, expr2, expr3)
> - NULLIF (expr1, expr2)
> - COALESCE (expr1, expr2, ..., expr*n*)
>
> **NVL 函数**
>
> **将空值转换成一个已知的值**:
>
> 可以使用的数据类型有日期、字符、数字。

> 函数的一般形式:
>
> - NVL(commission_pct,0)
> - NVL(hire_date,'01-JAN-97')
> - NVL(job_id,'No Job Yet')

```
-- 公司员工的年薪;为空的用0代替
select employee_id, last_name, salary * 12 * (1 + nvl(commission_pct, 0))
from system.employees

-- 先把部门编号转换为字符串类型(如果不转换，就会报无效数字)；在将部门编号为null的替换为指定字符串
select nvl(to_char(employee_id), '没有部门'), last_name from system.employees
```

使用 NVL2 函数

NVL2 (expr1, expr2, expr3) : expr1 不为 NULL， 返回 expr2； 为 NULL， 返回 expr3 。

```
-- 员工的奖金率，为null，返回0.01；若不为空，返回实际奖金率+0.015
select last_name, commission_pct, nvl2(commission_pct, commission_pct + 0.015,
0.01) from system.employees
```

使用 NULLIF 函数

NULLIF (expr1, expr2) : 相等返回NULL，不等返回expr1

```
SELECT first_name, LENGTH(first_name) "expr1",
       last_name,  LENGTH(last_name)  "expr2",
       NULLIF(LENGTH(first_name), LENGTH(last_name)) result
FROM   employees;
```

使用 COALESCE 函数

COALESCE 与 NVL 相比的优点在于 COALESCE 可以同时处理交替的多个值。

如果第一个表达式为空,则返回下一个表达式，对其他的参数进行COALESCE

```
SELECT   last_name,
         COALESCE(commission_pct, salary, 10) comm
FROM     employees
ORDER BY commission_pct;
```

条件表达式

在 SQL 语句中使用IF-THEN-ELSE 逻辑

使用两种方法:

CASE 表达式

DECODE 函数

```sql
-- 查询部门号为 10, 20, 30 的员工信息,
-- 若部门号为 10, 则打印其工资的 1.1 倍, 20 号部门
-- 则打印其工资的 1.2 倍
-- 30 号部门打印其工资的 1.3 倍数
select last_name, employee_id, department_id,
case department_id when 10 then salary * 1.1
                   when 20 then salary * 1.2
                   else salary * 1.3  end new_sal    -- 结束时起了个别名
from system.employees
where department_id in (10, 20, 30)
```

DECODE 函数

在需要使用 IF-THEN-ELSE 逻辑时:

```sql
-- 等效于上一个案例
select last_name, employee_id, department_id,
decode(department_id, 10, salary * 1.1,
                      20, salary * 1.2,
                      salary) new_sal
from system.employees
where department_id in (10, 20, 30)
```

多表查询

笛卡尔集

笛卡尔集会在下面条件下产生:

> - 省略连接条件
> - 连接条件无效
> - 所有表中的所有行互相连接
> -
> - 为了避免笛卡尔集， 可以在 WHERE 加入有效的连接条件。

```sql
-- 叉集：使用cross join 子句使连接的表产生叉集；叉集和笛卡尔积相同
select employee_id, d.department_id, d.department_name
from system.employees e cross join system.departments d
```

内连接

合并具有同一列的两个以上的表的行, **结果集中不包含一个表与另一个表不匹配的行**

分等值(连接) & 不等值(连接)

等值使用（=）连接

不等值使用等号之外的操作符：<>、>、<、>=、<=、LIKE、IN、BETWEEN...AND。

> **使用表名前缀在多个表中区分相同的列。**
>
> 在不同表中具有相同列名的列可以用**表的别名**加以区分
>
> 连接n个表，至少需要n-1个连接条件

```
-- 当两个多个表中的字段名是唯一的，那么可以不加别名；当然建议加上，可读性更高
select e.employee_id, d.department_id, d.department_name
from system.employees e, system.departments d
where e.department_id = d.department_id

-- 使用inner join on内连接方式（inner可选择性省略）
select e.employee_id, d.department_id, d.department_name
from system.employees e inner join system.departments d
on e.department_id = d.department_id

-- 查询三个表的信息
select e.employee_id, d.department_id, d.department_name, l.city
from system.employees e, system.departments d, system.locations l
where e.department_id = d.department_id and d.location_id = l.location_id

-- 查询公司员工的工资属于a-f哪种（属于非等值条件，在多个值之间使用bewteen方式连接）
select employee_id, last_name, salary, grade_level
from system.employees e, system.job_grades j
where e.salary between j.lowest_sal and j.highest_sal
```

使用 ON 子句创建连接 （常用,属于内连接）

> 自然连接中是以具有相同名字的列为连接条件的。
>
> **可以使用 ON 子句指定额外的连接条件。**
>
> 这个连接条件是与其它条件分开的。
>
> **ON 子句使语句具有更高的易读性**

```
-- 两个表进行查询（省略了inner）
select employee_id, d.department_id, d.department_name
from system.employees e join system.departments d
on e.department_id = d.department_id


-- 三个表进行查询（省略了inner）
select employee_id, d.department_id, d.department_name,l.city
from system.employees e join system.departments d
on e.department_id = d.department_id
join system.locations l
on d.location_id = l.location_id
```

外连接

两个表在连接过程中除了返回满足连接条件的行以外**还返回左（或右）表中不满足条件的行，这种连接称为左（或右） 外连接。**没有匹配的行时, 结果表中相应的列为空(NULL). 外连接的 WHERE 子句条件类似于内部连接, 但**连接条件中没有匹配行的表的列后面**要加**外连接运算符，即用圆括号括起来的加号 (+) .**

使用外连接可以查询不满足连接条件的数据。

外连接的符号是 (+);,条件中右边有(+)是左外，左边有(+)是右外

只能右左外联或右外联；没有两边都有(+)的情况；会报错

letft/right outer ... join ... on...　　outer可选择性省略

```
-- 左外连接
select e.last_name, e.employee_id, d.department_id, d.department_name
from system.employees e, system.departments d
where e.department_id = d.department_id(+)

-- 左外连接的另一种写法；多张表就在on条件后继续加left outer join 表名 on条件
select employee_id, d.department_id, d.department_name
from system.employees e left outer join system.departments d
on e.department_id = d.department_id


-- 右外连接
select e.last_name, e.employee_id, d.department_id, d.department_name
from system.employees e, system.departments d
where e.department_id(+) = d.department_id

-- 右外连接的另一种写法；多张表就在on条件后继续加right outer join 表名 on条件
select employee_id, d.department_id, d.department_name
from system.employees e right outer join system.departments d
on e.department_id = d.department_id
```

```
-- 全连接
select employee_id, d.department_id, d.department_name
from system.employees e full outer join system.departments d
on e.department_id = d.department_id
```

自然连接

> NATURAL JOIN 子句，**会以两个表中具有相同名字的列为条件创建等值连接。**
>
> 在表中查询满足等值条件的数据。
>
> 如果只是列名相同而**数据类型不同**，则会产生错误。
>
> 一般不使用这种方式，不推荐，有多个列名相同的情况，会自动连接

```
-- 不能再字段名前，加表名或表名的别名；否则会报错
select employee_id, department_id, department_name
from system.employees natural join system.departments
```

使用 USING 子句创建连接

> 在NATURAL JOIN 子句创建等值连接时，可以**使用 USING 子句指定等值连接中需要用到的列。**
>
> 使用 USING 可以在有多个列满足条件时进行选择。
>
> **不要给选中的列中加上表名前缀或别名。**
>
> **JOIN 和 USING 子句经常同时使用。**
>
> **缺点：两个表的连接字段名称必须相同,且数据类型也必须相同，否则连接不上，报错**

```
select employee_id, department_id, department_name
from system.employees join system.departments
using(department_id)
```

自连接

一张表里面查；在他一张表中有查询条件有返回时，且根据返回添加继续查询的话可以使用

```
-- 查询公司员工‘chen’的manager(管理者)的信息
-- chen的上级的编号
select last_name, manager_id
from system.employees
where lower(last_name) = 'chen'

-- 再根据查到的编号，查询指定的管理
select last_name, salary, email
from system.employees
where employee_id = 108



-- 使用自连接的方式查询
-- 需要查询的字段  员工的名字   管理的名字        管理的工资        管理的邮箱
select emp.last_name, manager.last_name, manager.salary, manager.email
-- 从一张表中，分出两次查询
from system.employees emp, system.employees manager
-- 管理的id =  通过员工的名字是chen的查到    员工表中的管理id
where emp.manager_id = manager.employee_id and lower(emp.last_name) = 'chen'
```

分组函数

组函数

avg(平均值)、 count(总个数)、 max(最大值)、 min(最小值)、 stddev(方差)、 sum(总和)

avg和sun只能使用number类型的数据；且avg只计算非空的数据

```
-- 工资的平均值          最高工资          最低工资      总发放的工资
select avg(salary), max(salary), min(salary), sum(salary) from system.employees;

-- 字符串的最大值就是首字母z      最小是a      日期最大值是最接近现在的日期
select max(last_name), min(last_name), max(hire_date), min(hire_date) from
system.employees

-- 查询id个数                  名字个数              日期个数
select count(employee_id), count(last_name), count(hire_date) from
system.employees;

-- avg = sum / count(进度更大点)
select avg(salary), sum(salary) / count(salary) from system.employees
```

count

count(*) 和 count(1)和count(列名)区别

count(*)包括了所有的列，相当于行数，在统计结果的时候，不会忽略列值为NULL

count(1)包括了忽略所有列，用1代表代码行，在统计结果的时候，不会忽略列值为NULL

count(列名)只包括列名那一列，在统计结果的时候，会忽略列值为空（这里的空不是只空字符串或者0，而是表示null）的计数，即某个字段值为NULL时，不统计

(1)这个表示的表里有1字段(假设)这里用1

> **效率**
>
> 列名为主键，count(列名)会比count(1)快
> 列名不为主键，count(1)会比count(列名)快
> 如果表多个列并且没有主键，则 count（1） 的执行效率优于 count（）
> *如果有主键，则 select count（主键）的执行效率是最优的*
> *如果表只有一个字段，则 select count（）最优。*

```
select count(1), count(2), count(3), count(-1) from system.employees;
```

分组查询

在SELECT列表中所有未包含在组函数中的列都应该包含在GROUP BY子句中。

包含在GROUP BY子句中的列不必包含在SELECT列表中

不能在HERE子句中使用组函数。

可以在HAVING子句中使用组函数————————**having放在group前后都行**

```sql
-- 员工表中各个部门的平均工资
select department_id, avg(salary)
from system.employees
group by department_id

-- 只查询指定部门
select department_id, avg(salary)
from system.employees
where department_id in (40, 60, 80)
group by department_id

-- 各个部门中平均工资大于6k的部门，以及平均工资
select department_id, avg(salary)
from system.employees
having avg(salary) > 6000
group by department_id

-- 员工表中，平均工资最大的部门
select max(avg(salary))
from system.employees
group by department_id
```

子查询

子查询(内查询)在主查询之前一次执行完成。

子查询的结果被主查询(外查询)使用。

> **注意事项**
>
> - 子查询要包含在括号内。
> - 将子查询放在比较条件的右侧。
> - 单行操作符对应单行子查询，多行操作符对应多行子查询(子查询返回多条结果，给外部提供条件)。

单行子查询

只返回一行。

使用单行比较操作符

操作符	含义
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to

```
-- 通过自连接方式，查询比Abel工资高的
select e2.last_name, e2.salary
from system.employees e1, system.employees e2
where e1.salary < e2.salary and lower(e1.last_name) = 'abel'

-- 通过子查询方式，查询比Abel工资高的(一条语句返回的值，给另一个语句作为条件查询)
select last_name, salary
from system.employees
where salary > (select salary
                        from system.employees
                        where last_name = 'Abel')

-- 查询员工位chen的管理者的id，通过子查询的方式
select last_name, department_id, salary
from system.employees
where employee_id = (select manager_id
                        from system.employees
                        where lower(last_name) = 'chen')

-- 单行子查询，只返回员工结果
-- 返回job_id与141号员工相同，salary比143号员工多的员工姓名，job_id 和工资
select last_name "名称", job_id "工号", salary "工资"
from system.employees
where job_id = (select job_id
                        from system.employees
                        where employee_id = 141)
              and salary > (select salary
                        from system.employees
                        where employee_id = 143)
-- 返回公司工资最少的员工的last_name,job_id和salary
select last_name "名称", job_id "工号", salary "工资"
from system.employees
where salary = (select min(salary)
                        from system.employees
                        )

-- 查询工资最低的员工信息：last_name,salary
select last_name "名字", salary "工资"
from system.employees
where salary = (select min(salary)
                    from system.employees)
```

在子查询中使用组函数

```
-- 查询最低工资大于50号部门最低工资的部门id和其最低工资
select department_id "部门id", min(salary) "最低工资"
from system.employees
group by department_id
having min(salary) > (select min(salary)
                      from system.employees
                      where department_id = 50)
```

多行子查询

返回多行。

使用多行比较操作符

操作符	含义
IN	**等于列表中的 任意一个**
ANY	**和子查询返回的 某一个 值比较**
ALL	**和子查询返回的 所有 值比较**

```
-- any操作符
-- 返回其它部门中比job_id为‘IT_PROG’部门任一工资低的员工的员工号、姓名、job_id 以及salary
select employee_id "员工id", last_name "姓名", job_id "工作id", salary "工资"
from system.employees
where lower(job_id) <> lower('IT_PROG')
      and salary < any(select salary from system.employees
                  where lower(job_id) = lower('IT_PROG'))

-- all操作符
-- 返回其它部门中比job_id为‘IT_PROG’部门所有工资都低的员工的员工号、姓名、job_id 以及salary
select employee_id "员工id", last_name "姓名", job_id "工作id", salary "工资"
from system.employees
where lower(job_id) <> lower('IT_PROG')
      and salary < all(select salary from system.employees
                  where lower(job_id) = lower('IT_PROG'))


-- 查询平均工资最低的部门信息---having必须要跟group by查询
    -- 先查询到所有部门中的平均工资中最低的一个部门
    -- 在根据指定部门，在部门表查询到部门所有信息
select *
from system.departments
where department_id = (select department_id "部门id"
from system.employees
having avg(salary) = (select min(avg(salary))
                    from system.employees
                    group by department_id)
```

```
group by department_id)

-- 查询平均工资最低的部门信息和该部门的平均工资
    -- 先查询到所有部门中的平均工资中最低的一个部门
    -- 在根据指定部门，在部门表查询到部门所有信息
    -- 为查询表起个别名，根据查到的部门id，再员工表查询平均工资
select sd.*, (select avg(salary) from system.employees where department_id =
sd.department_id)
from system.departments sd
where department_id = (select department_id "部门id"
from system.employees
having avg(salary) = (select min(avg(salary))
                      from system.employees
                      group by department_id)
group by department_id)


-- 查询平均工资最高的job信息
select *
from system.jobs
where job_id = (select job_id
                from system.employees
                having avg(salary) = (select max(avg(salary))
                                      from system.employees
                                      group by job_id)
                 group by job_id)

-- 查询平均工资高于公司平均工资的部门有哪些?
select department_id, avg(salary)
from system.employees
having avg(salary) > (select avg(salary)
                      from system.employees)
group by department_id

-- 各个部门中最高工资，中最低的那个部门的，最低工资是多少
select min(salary)
from system.employees
where department_id = (select department_id
                       from system.employees
                       having max(salary) = (select min(max(salary))
                                             from system.employees
                                             group by department_id)
                       group by department_id)

-- 查询平均工资最高的部门的manager的详细信息: last_name，department_id,email,salary
select last_name, department_id, email, salary
from system.employees
where employee_id in (select manager_id
                      from system.employees
                      where department_id = (select department_id
from system.employees
group by department_id
having avg(salary) = (select max(avg(salary))
                from system.employees
                group by department_id)))

-- 查询1999年来公司的员工中的最高工资的那个员工的信息.
select *
```

```
from system.employees
where to_char(hire_date, 'yyyy') = '1999' and salary = (
                                    select max(salary)
                                    from system.employees
                                    where to_char(hire_date, 'yyyy') = '1999')

-- 查询各部门中工资比本部门平均工资高的员工的员工号，姓名和工资
select employee_id, last_name, salary
from system.employees e1
where salary > (select avg(salary)
                from system.employees e2
                where e1.department_id = e2.department_id
                group by department_id)
```

创建和管理表

DDL

常见的数据库对象

对象	描述
表	**基本的数据存储集合，由行和列组成。**
视图	**从表中抽出的逻辑上相关的数据集合**
序列	**提供有规律的数值。**
索引	**提高查询的效率**
同义词	**给对象起别名**

创建表

> 表名和列名:
>
> - 必须**以字母开头**
> - 必须在 1–30 个字符之间
> - 必须只能包含 A–Z, a–z, 0–9, _, $, 和 **#**
> - 必须不能和用户定义的其他对象重名
> - 必须不能是Oracle 的保留字

数据类型

数据类型	描述
VARCHAR 2 (size)	可变长字符数据
CHAR(size)	定长字符数据
NUMBER(p , s)	可变长数值数据
DATE	日期型数据
LONG	可变长字符数据，最大可达到 2G
CLOB	字符数据，最大可达到 4G
RAW (LONG RAW)	原始的二进制数据
BLOB	二进制数据，最大可达到 4G
BFILE	存储外部文件的 二进制数据，最大可达到 4G
ROWID	行地址

创建表语句

> 必须具备:
>
> - CREATE TABLE权限
> - 存储空间
>
> 必须指定
>
> - 表名
> - 列名, 数据类型, **尺寸**

第一种方式

CREATE TABLE 语句

```
CREATE TABLE [schema.]table
        (column datatype [DEFAULT expr][, ...]);

-- 语法
CREATE TABLE emp1 (
        id  NUMBER(10),
        name    VARCHAR2(20),       -- 字符串，长度为20
        salary  NUMBER(10, 2),      -- 数字类型，长度为10，有两个是小数
        hire_date date
);
Table created   -- 创建成功的表示

-- 查看表结构
desc 表名
--  确认创建
DESCRIBE dept
```

第二种方式(依托于现有表)

使用子查询创建表

```
-- 根据employees里面的列创建表，可以起新的列名，也可以不起，保持employees的列名；但是这种方式会把数据也带过来
CREATE TABLE emp2
AS
SELECT employee_id id, last_name name, hire_date, salary
from employees;

-- 只要80号部门的员工数据
CREATE TABLE emp2
AS
SELECT employee_id id, last_name name, hire_date, salary
from employees
where department_id = 80;


-- 不要数据的方式；只要where条件为false就不会带数据；默认不写where条件是true
CREATE TABLE emp2
AS
SELECT employee_id id, last_name name, hire_date, salary
from employees
where 1 = 2;
```

查看用户创建的表

```
SELECT * FROM user_tables
```

查看用户定义的表

```
select table_name from user_tables
```

查看用户定义的各种数据库对象

```
select distinct object_type from user_objects
```

查看用户定义的表，视图，同义词和序列

```
select * from user_catalog;
```

ALTER TABLE 语句

rollback执行后可以回到上一步；除了对表操作外：创建表，修改表，清空表，删除表

> 使用 ALTER TABLE 语句可以:
>
> - 追加新的列

- 使用 ADD 子句追加一个新列
- 新列是表中的最后一列

```
alter table emp1
add (email varchar2(20))     -- 新增email列，字符串类型长度为20
```

- 修改现有的列，没有数据的情况
  - 可以修改列的**数据类型，尺寸和默认值**
  - 对默认值的修改**只影响今后对表的修改**

```
alter table emp1
mpdify (id NUMBER(15)) --    修改id为数字类型，长度为15
```

- 为新追加的列定义默认值

```
-- 增加一个默认值
alter table emp1
mpdify (salary NUMBER(20, 2) default 2000)  -- 默认值为2000
```

- 设置指定列不可用，desc 表名，并不会显示此列

```
alter table 表名
set unused cloumn 列名

-- 把不可见的这列删除
alter table 表名
drop unused columns
```

- 删除一个列
  - 使用 DROP COLUMN 子句删除不再需要的列

```
alter table emp1
drop column email        -- 删除email列
```

- 重命名表的一个列名
  - 使用 RENAME COLUMN [table_name] TO子句重命名列

```
alter table emp1
rename column salary to sal -- 把salary修改为sal
```

删除表

> **数据和结构都被删除**
>
> 所有正在运行的相关事务被提交
>
> 所有相关索引被删除

> **DROP TABLE 语句不能回滚**

```
drop table 表名
```

清空表

> **TRUNCATE TABLE** 语句:
>
> - 删除表中所有的数据
> - 释放表的存储空间
> - **TRUNCATE**语句**不能回滚**
> - 可以使用 **DELETE** 语句删除数据，**可以回滚**

```
TRUNCATE TABLE 表名;
```

改变对象的名称

> 执行**RENAME**语句**改变表**, 视图, 序列, 或同义词的**名称**
>
> 必须是对象的拥有者

```
RENAME emp1 TO emp2;
```

常用ddl语句

以下这些 DDL 的命令，操作外，皆不可回滚

CREATE TABLE	**创建表**
ALTER TABLE	**修改表结构**
DROP TABLE	**删除表**
RENAME TO	**重命名表**
TRUNCATE TABLE	**删除表中的所有数据，并释放存储空间**

数据操作———DML

只要没有commit(提交)；都可以rollback(回滚)

> DML(Data Manipulation Language – 数据操纵语言) 可以在下列条件下执行:
>
> - 向表中插入数据
> - 修改现存数据
> - 删除现存数据

> 事务是由完成若干项工作的DML语句组成的

insert插入语句

使用 INSERT 语句向表中插入数据

> - 为每一列添加一个新值。
> - 按列的默认顺序列出各个列的值。
> - 在 INSERT 子句中随意列出列名和他们的值。
> - **字符和日期型数据应包含在 单引号 中**

使用这种语法一次只能向表中插入**一条**数据

```
-- 语法，不写列名的话，必须严格按照列的顺序和类型进行插入
INSERT INTO table [(column [, column...])]
VALUES      (value [, value...]);


-- 创建新表
create table emp1
as
select employee_id,   last_name, hire_date, salary
from system.employees where 1 = 2;

-- 插入数据
insert into emp1
values(1001, 'AA', sysdate, 10000)

-- 插入日期
insert into emp1
values(1002, 'BB', to_date(to_char(sysdate, 'yyyy-mm-dd'), 'yyyy-mm-dd'), 20000)

-- 插入时，有空值----->显式
insert into emp1
values(1003, 'CC', to_date(to_char(sysdate, 'yyyy-mm-dd'), 'yyyy-mm-dd'), null)
```

指定列名赋值

```
-- 按照指定列的顺序和类型进行赋值，没有指定的列，默认是null----->注意列的约束是否为空:desc 表名可查看
insert into emp1(last_name, employee_id, hire_date)
values('DD', 1004, to_date(to_char(sysdate, 'yyyy-mm-dd'), 'yyyy-mm-dd'))
```

从其他表拷贝数据

- 在 INSERT 语句中加入子查询
- **不必书写 VALUES 子句。**
- 子查询中的值列表应与 INSERT 子句中的列名对应

```
-- 不要写values；但是列和列位置和类型必须对应，不添加所有数据，可以加上过滤条件
insert into emp1(employee_id, hire_date, last_name, salary)
select employee_id, hire_date, last_name, salary
from system.employees
where department_id = 80
```

使用脚本的方式

```
-- 按照正常的顺序和类型，需要加引号的还是要加上
insert into emp1(employee_id, last_name, salary, hire_date)
values(&id, '&name', &salary, '&hire_date')
```

update更新语句

- 使用 UPDATE 语句更新数据
- 可以一次更新**多条**数据

```
-- 语法
UPDATE      table
SET     column = value [, column = value, ...]
[WHERE      condition];

-- 不加where更新所有的工资；一般都是指定修改
update emp1
set salary = 22000
```

带条件更新

- 使用 **WHERE** 子句指定需要更新的数据
- 如果**省略 WHERE** 子句，则表中的**所有数据**都将**被更新**

```
-- 修改指定员工的工资
update emp1
set salary = 12000
where employee_id = 179
```

在 UPDATE 语句中使用子查询

```
-- 赋值一张employees表
create table employees1
as
select * from system.employees;

-- 更新114号员工的工作和工资使其与205号员工相同。---->where条件一定不能忘
update employees1
set job_id = (select job_id
              from employees1
              where employee_id = 205),
    salary = (
              select salary
              from employees1
              where employee_id = 205)
```

```
where employee_id = 114

-- 调整与employee_id为200的员工job_id相同的，员工的department_id为employee_id为100的员工的department_id.

update employees1
set department_id = (select department_id
              from employees1
              where employee_id = 100)
where job_id = (select job_id
                from employees1
                where employee_id = 200)
```

delete删除数据

使用 DELETE 语句从表中删除数据

```
-- 没有where条件，所以会清空表的数据
delete from 表名
```

WHERE 子句删除指定的记录

如果省略 WHERE 子句，则表中的全部数据将被删除

```
delete from 表名 where 列名 = 值
```

在 DELETE 中使用子查询

```
-- 从employees1表中删除部门名称中含Public字符的部门id
delete from employees1
where department_id = (select department_id
                       from system.departments
                       where department_name like '%Public%')
```

数据库事务

> 以第一个 DML 语句的执行作为开始
>
> 以下面的其中之一作为结束:
>
> - **COMMIT 或 ROLLBACK 语句**
> - DDL 语句（**自动提交**）
> - 用户会话正常结束
> - 系统异常终止

COMMIT 和 ROLLBACK 语句的优点

> 使用COMMIT 和 ROLLBACK语句,我们可以:
>
> - 确保数据完整性。
> - 数据改变被提交之前预览。
> - 将逻辑上相关的操作分组。

回滚到报留点

- 使用 **SAVEPOINT** 语句在当前事务中创建保存点。
- 使用 **ROLLBACK TO SAVEPOINT** 语句回滚到创建的保存点
- ```
  savepoint  保持点名称 -- 设置保存点
  rollback to (savepoint) 保存点名称 -- 回滚到指定保存点
  ```

事务进程

> 自动提交在以下情况中执行:
>
> - DDL 语句。
> - DCL 语句。
> - 不使用 COMMIT 或 ROLLBACK 语句提交或回滚，正常结束会话。
>
> 会话异常结束或系统异常会导致自动回滚

提交(commit)或回滚(rollback)前的数据状态

> 改变前的数据状态是可以恢复的
>
> 执行 DML 操作的用户可以通过 SELECT 语句查询之前的修正
>
> **其他用户不能看到当前用户所做的改变，直到当前用户结束事务。**
>
> **DML 语句所涉及到的行被锁定， 其他用户不能操作**

提交后的数据状态

> 数据的改变已经被保存到数据库中。
>
> 改变前的数据已经丢失。
>
> 所有用户可以看到结果。
>
> **锁被释放，其他用户可以操作涉及到的数据。**
>
> 所有保存点被释放

数据回滚后的状态

> 使用 ROLLBACK 语句可使数据变化失效:
>
> - 数据改变被取消。
> - 修改前的数据状态被恢复。
> - 锁被释放
  ```

语句	功能
INSERT	插入
UPDATE	修改
DELETE	删除
COMMIT	提交
SAVEPOINT	保存点
ROLLBACK	回滚

约束

> 约束是表级的强制规定
>
> 有以下五种约束:
>
> - **NOT NULL**:非空约束
> - **UNIQUE** :唯一约束，运行一个或多个null
> - **PRIMARY KEY**:主键约束：非空+唯一约束
> - **FOREIGN KEY**:外键约束
> - **CHECK**:检查条件

注意事项

> 如果不指定约束名， Oracle server 自动按照 SYS_C*n* 的格式指定约束名
>
> **创建**和修改约束:
>
> - **建表的同时**
> - **建表之后**：alter的方式
>
> 可以在**表级**或**列级**定义约束
>
> 可以通过数据字典视图查看约束

表级约束和列级约束

> **作用范围**:
>
> - 列级约束只能作用在一个列上
> - 表级约束可以作用在多个列上（当然表级约束也 可以作用在一个列上）
>
> **定义方式**：列约束必须跟在列的定义后面，表约束不与列一起，而是单独定义。
>
> **非空 (not null) 约束只能定义在列上**

```
-- 创建表，添加非空约束
```

```
create table emp2(
id number(10) constraint emp2_id_nn not null,  -- 非空约束给约束起名：表名_列名_约束名
name varchar2(20) not null, -- 非空约束
salary number(10, 2)
)

-- 创建表，添加唯一约束
create table emp3(
-- 列级约束
id number(10) constraint emp3_id_uk unique,
name varchar2(20) constraint emp3_name_nn not null,
salary number(10, 2),
email varchar2(20),
-- 表级约束，在表结构最后
constraint emp3_email_uk unique(email)      -- 表级约束，作用于email列上
)


-- 创建表，添加主键约束
create table emp4(
id number(10) constraint emp4_id_pk primary key,
name varchar2(20) constraint emp4_name_nn not null,
salary number(10, 2),
email varchar2(20),
-- 表级约束，在表结构最后
constraint emp4_email_uk unique(email)      -- 表级约束，作用于email列上
)


-- 添加表级的主键约束
create table emp5(
id number(10),
name varchar2(20) constraint emp5_name_nn not null,
salary number(10, 2),
email varchar2(20),
-- 表级约束，在表结构最后
constraint emp5_email_uk unique(email),      -- 表级约束，作用于email列上
constraint emp4_id_pk primary key(id)
)

-- 创建表，添加外键约束
create table emp6(
id number(10),
name varchar2(20) constraint emp6_name_nn not null,
salary number(10, 2),
email varchar2(20),
department_id number(10),
-- 表级约束，在表结构最后
constraint emp6_email_uk unique(email),      -- 表级约束，作用于email列上
constraint emp6_id_pk primary key(id),
--           约束名          外键约束     作用于某个列           参考于
constraint emp6_dept_id_fk foreign key(department_id) references
system.departments(department_id)
)
```

FOREIGN KEY 约束的关键字

- FOREIGN KEY: 在表级指定子表中的列
- REFERENCES: 标示在父表中的列
- **ON DELETE CASCADE(级联删除)**: 当父表中的列被删除时，子表中相对应的列也被删除
    - ```
      create table emp6(
      id number(10),
      name varchar2(20) constraint emp6_name_nn not null,
      salary number(10, 2),
      email varchar2(20),
      department_id number(10),
      -- 表级约束，在表结构最后
      constraint emp6_email_uk unique(email),      -- 表级约束，作用于email列上
      constraint emp6_id_pk primary key(id),
      constraint emp6_dept_id_fk foreign key(department_id) references system.departments(department_id) on delete set cascade
      )
      ```

- **ON DELETE SET NULL(级联置空)**: 子表中相应的列置空
    - ```
      create table emp6(
      id number(10),
      name varchar2(20) constraint emp6_name_nn not null,
      salary number(10, 2),
      email varchar2(20),
      department_id number(10),
      -- 表级约束，在表结构最后
      constraint emp6_email_uk unique(email),      -- 表级约束，作用于email列上
      constraint emp6_id_pk primary key(id),
      constraint emp6_dept_id_fk foreign key(department_id) references system.departments(department_id) on delete set null
      )
      ```

check约束

```
create table emp7(
id number(10),
name varchar2(20) constraint emp7_name_nn not null,
-- 要求工资必须大于1500，并且不能大于3w
salary number(10, 2) constraint emp7_salary_che check(salary > 1500 and salary <
30000),
email varchar2(20),
department_id number(10),
-- 表级约束，在表结构最后
constraint emp7_email_uk unique(email),      -- 表级约束，作用于email列上
constraint emp7_id_pk primary key(id),
constraint emp7_dept_id_fk foreign key(department_id) references
system.departments(department_id) on delete set cascade
)
```

添加约束，已经创建过的表

> 使用 ALTER TABLE 语句:
>
> - 添加或删除约束,**但是不能修改约束**
> - 有效化或无效化约束
> - **添加 NOT NULL 约束要使用 MODIFY 语句**

```
-- 语法
  ALTER TABLE table
  ADD [CONSTRAINT constraint] type (column);


-- 添加约束
alter table emp5
add constraint emp5_nameuk unique(name)

alter table emp5
modify (salary number(10, 2) not null)

-- 删除约束
alter table emp5
drop constraint emp5_name_nn    -- 约束名字

-- 无效化约束：DISABLE
alter table emp3
disable constraint emp3_email_uk

-- 激活约束
-- 当定义或激活UNIQUE 或 PRIMARY KEY 约束时系统会自动创建UNIQUE 或 PRIMARY KEY索引
alter table emp3
enable constraint emp3_email_uk


-- 查询约束
select constraint_name, constraint_type, search_condition
from user_constraints
where table_name = 'EMP3'   -- 这里的表名必须大写
```

```sql
-- 查询定义约束的列
SELECT  constraint_name, column_name
FROM  user_cons_columns
WHERE  table_name = 'EMP3'
```

视图

对象	描述
表	基本的数据存储集合，由行和列组成
视图	从表中抽出的逻辑上相关的数据集合
序列	提供有规律的数值
索引	提高查询的效率
同义词	给对象起别名

> 使用视图的优势
>
> - 控制数据访问
> - **简化查询**
> - 避免重复访问相同的数据
>
> 视图是一种虚表。
>
> **视图建立在已有表的基础上**, 视图赖以建立的这些表称为**基表**。
>
> 向视图提供数据内容的语句为 SELECT 语句, 可以将视图理解为**存储起来的 SELECT 语句**
>
> 视图向用户提供基表数据的另一种表现形式
>
> **修改视图数据后，基表也会跟着修改**

简单视图和复杂视图

特性	简单视图	复杂视图
表的数量	一个	一个或多个
函数	没有	有
分组	没有	有
DML操作	可以	有时可以

创建视图

```
create view empview
as
select employee_id, last_name, salary   -- 需要提取的字段，两个需要为视图起新名字，就和表起别名方式一样
from system.employees   -- 视图的基表
where department_id = 80


-- 多个表的数据，放在一个视图中
create view empview2
as
select employee_id id, last_name name, salary, department_name
from system.employees e, system.departments d
where e.department_id = d.department_id
```

查看视图

```
select * from empview

-- 查询视图结构
desc 视图名称
```

修改视图

```
-- 修改视图中的指定员工的工资；基表的数据也会跟着修改(包括删除，增加)
update empview
set salary = 20000
where employee_id = 179


-- 创建或者是替换；如果已经存在则会对已有的视图覆盖
create or replace view empview2
as
select employee_id id, last_name name, department_name
from system.employees e, system.departments d
where e.department_id = d.department_id


-- 只允许此视图可查看，不了进行增删改操作
create or replace view empview2
as
select employee_id id, last_name name, department_name
from system.employees e, system.departments d
where e.department_id = d.department_id
with read only
```

创建复杂视图

```
create or replace view empview3
as
select department_name, avg(salary) avg_sal -- 组函数必须要起别名
from system.employees e, system.departments d
where e.department_id = d.department_id
group by department_name    -- 查询条件没有出现在组函数中，那么就需要放在group by中
```

> **视图中使用 DML 的规定**
>
> 可以在简单视图中执行 DML 操作
>
> 当视图定义中包含以下元素之一时不能使用delete:
>
> - 组函数
> - GROUP BY 子句
> - DISTINCT 关键字
> - ROWNUM 伪列
>
> 当视图定义中包含以下元素之一时不能使用update:
>
> - 组函数
> - GROUP BY子句
> - DISTINCT 关键字
> - ROWNUM 伪列
> - 列的定义为表达式
>
> 当视图定义中包含以下元素之一时不能使insert:
>
> - 组函数
> - GROUP BY 子句
> - DISTINCT 关键字
> - ROWNUM 伪列
> - 列的定义为表达式
> - 表中非空的列在视图定义中未包括
>
> **屏蔽 DML 操作**
>
> - 可以使用 **WITH READ ONLY** 选项屏蔽对视图的DML 操作
> - 任何 DML 操作都会返回一个Oracle server 错误

```
-- 只允许此视图可查看，不了进行增删改操作
create or replace view empview2
as
select employee_id id, last_name name, department_name
from system.employees e, system.departments d
where e.department_id = d.department_id
with read only
```

删除视图

删除视图只是删除视图的定义，并不会删除基表的数据

```
drop view empview3;
```

Top-N分析

类似mysql的分页查询

> **注意：**
>
> **对 ROWNUM 只能使用 < 或 <=, 而用 =, >, >= 都将不能返回任何数据。**

```
-- 工资最高的前10个人的信息
-- rownum属于伪列；类似excel的最左边的数字列
select rownum, employee_id, last_name, salary
from (
     select employee_id, last_name, salary
     from system.employees
     order by salary desc
)
where rownum <= 10


-- 查询工资逆序排行40-50之间的随机
select rn, employee_id, last_name, salary
from (-- 类似里面也是一个表
     select rownum rn, employee_id, last_name, salary
      from (
           select employee_id, last_name, salary
           from system.employees
           order by salary desc
      )
      where rownum <= 50
)
where rn > 40 and rn <= 50
```

分页规律总结：每页显示m条数据，查询第n页数据

```
select * from (select rownum r, e.* from 要分页的表 e where rownum <= m * n) t
where r > m *n - m ;
```

其他数据库对象

对象	描述
表	基本的数据存储集合，由行和列组成
视图	从表中抽出的逻辑上相关的数据集合
序列	提供有规律的数值
索引	提高查询的效率
同义词	给对象起别名

序列

类似mysql的自增长

> 序列: 可供多个用户用来产生唯一数值的数据库对象
>
> - 自动提供唯一的数值
> - 共享对象
> - **主要用于提供主键值**
> - 将序列值装入内存可以提高访问效率

创建序列

> 创建序列 DEPT_DEPTID_SEQ为表 DEPARTMENTS 提供主键
>
> 不使用 CYCLE 选项

```
-- 语法
CREATE SEQUENCE sequence
       [INCREMENT BY n]  --每次增长的数值
       [START WITH n]    --从哪个值开始
       [{MAXVALUE n | NOMAXVALUE}]
       [{MINVALUE n | NOMINVALUE}]
       [{CYCLE | NOCYCLE}]      --是否需要循环
       [{CACHE n | NOCACHE}];  --是否缓存登录


-- 创建序列
create sequence empseq
increment by 10 -- 每次增长10
start with 10   -- 从10开始增长
-- 不设置最大值，默认是1E28：后面的部分E表示后面有多少个零。
maxvalue 100    -- 最大值为100----->当序列超过最大值，则会循环为开始的值+1，下次就是当前值
+10
cycle           -- 需要循环
nocache         -- 不需要缓存登录
```

查询序列

> 查询数据字典视图 **USER_SEQUENCES** 获取序列定义信息
>
> 如果指定NOCACHE 选项，则列LAST_NUMBER 显示序列中**下一 个有效**的值

```
-- 必须先使用nextval获取一次值
select empseq.nextval from dual;

-- 才能使用currval获取当前的值
select empseq.currval from dual;


-- 查看当前序列的有效值
SELECT  sequence_name, min_value, max_value,
    increment_by, last_number
FROM    user_sequences;
```

NEXTVAL 和 CURRVAL 伪列

- NEXTVAL 返回序列中下一个有效的值，任何用户都可以引用
- CURRVAL 中存放序列的当前值
- **NEXTVAL 应在 CURRVAL 之前指定**，否则会报CURRVAL 尚未在此会话中定义的错误。

使用序列

> 将序列值装入内存可提高访问效率
>
> 序列在下列情况下出现**裂缝**:
>
> 回滚
>
> 系统异常
>
> 多个表同时使用同一序列
>
> 如果不将序列的值装入内存(NOCACHE), 可使用表 USER_SEQUENCES 查看序列当前的有效值

```
-- 创建一张(空)表
create table emp01
as
select employee_id, last_name, salary
from system.employees
where 1 = 2

-- 使用序列，自动增加id
insert into emp01
values(empseq.nextval, 'AA', 2300)
```

修改序列

修改序列的增量, 最大值, 最小值, 循环选项, 或是否装入内存

注意事项

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

```
alter sequence empseq
increment by 1  --  每次增长1
nocycle         -- 不循环

-- 删除默认最大值
alter sequence empseq
nomaxvalue  --  没有最大值
nocycle     -- 没有循环
```

删除序列

使用 DROP SEQUENCE 语句删除序列

删除之后，序列不能再次被引用

```
drop sequence 序列名称
```

索引

> - 一种独立于表的模式对象, 可以存储在与表不同的磁盘或表空间中
> - 索引被删除或损坏, 不会对表产生影响, 其影响的只是**查询的速度**
> - 索引一旦建立, Oracle 管理系统会对其进行自动维护, 而且由 Oracle 管理系统决定何时使用索引。用户不用在查询语句中指定使用哪个索引
> - 在删除一个表时,所有基于该表的索引会自动被删除
> - 通过指针**加速 Oracle 服务器的查询速度**
> - 通过快速定位数据的方法，减少磁盘 I/O
>
> 索引不需要用，只是说我们在用name进行查询的时候，速度会更快。当然查的速度快了，插入的速度就会慢。因为插入数据的同时，还需要维护一个索引

什么时候创建索引

> 以下情况可以创建索引:
>
> - 列中数据值分布范围很广
> - 列经常在 WHERE 子句或连接条件中出现
> - 表经常被访问而且数据量很大 ， 访问的数据大概占数据总量的2%到4%

创建索引

> •**自动创建**: 在定义 PRIMARY KEY 或 UNIQUE 约束后系统自动在相应的列上创建**唯一性**索引
>
> •**手动创建**: 用户可以在其它列上创建非唯一的索引，以加速查询

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

-- 在表emp01的employee_id上创建索引
create index emp01_id_ix     -- 索引名称
on emp01(employee_id)        -- 多个列，就在小括号里面用逗号隔开，在添加其他列名
```

查询索引

可以使用数据字典视图 USER_INDEXES 和 USER_IND_COLUMNS 查看索引的信息

```
SELECT  ic.index_name, ic.column_name,
    ic.column_position col_pos,ix.uniqueness
FROM    user_indexes ix, user_ind_columns ic
WHERE   ic.index_name = ix.index_name
AND ic.table_name = 'EMPLOYEES'; -- 需要大写
```

删除索引

只有索引的拥有者或拥有DROP ANY INDEX 权限的用户才可以删除索引

删除操作是不可回滚的

```
-- 使用DROP INDEX 命令删除索引
drop INDEX 索引名称

-- 删除索引UPPER_LAST_NAME_IDX
DROP INDEX emp01_id_ix
```

什么时候不要创建索引

> 下列情况不要创建索引:
>
> - 表很小
> - 列不经常作为连接条件或出现在WHERE子句中
> - 查询的数据大于2%到4%
> - 表经常更新

同义词

> 使用同义词访问相同的对象:
>
> - 方便访问其它用户的对象
> - 缩短对象名字的长度

```
-- 语法：SYNONYM关键字
CREATE [PUBLIC] SYNONYM synonym
FOR object;
```

创建和删除同义词

```
-- 为视图DEPT_SUM_VU 创建同义词
create synonym e for emp01  -- select * from e  也可以查到emp01的数据

-- 删除同义词
DROP SYNONYM 同义词名称;
DROP SYNONYM e;
```

控制用户权限

权限

> 数据库安全性:
>
> - 系统安全性
> - 数据安全性
>
> 系统权限: 对于数据库的权限
>
> 对象权限: 操作数据库对象的权限

系统权限

> 超过一百多种有效的权限
>
> 数据库管理员具有高级权限以完成管理任务，例如:
>
> - 创建新用户
> - 删除用户
> - 删除表
> - 备份表

创建用户

```
-- 语法
CREATE USER user
IDENTIFIED BY   password;


-- 创建的用户at01    密码是at01
create user at01
identified by at01;

-- 赋予系统权限：创建会话权限session---->必要的，否则不能登录
grant create session to at01;
```

用户的系统权限

用户创建之后, DBA 会赋予用户一些系统权限

```
-- 语法
GRANT privilege [, privilege...]
TO user [, user| role, PUBLIC...];

-- 以应用程序开发者为例，一般具有下列系统权限
CREATE SESSION  --  （创建会话）
CREATE TABLE    --  （创建表）
CREATE SEQUENCE --  （创建序列）
CREATE VIEW     --  （创建视图）
CREATE PROCEDURE    --  （创建过程）
```

赋予系统权限

DBA 可以赋予用户特定的权限

```
-- grant：授予
GRANT  create session, create table,
       create sequence, create view
TO     scott;

-- 赋予at01创建表的权限
grant create table
to at01;
```

创建用户表空间

用户拥有create table权限之外，还需要分配相应的表空间才可开辟存储空间用于创建的表

```
-- 语法/也可以修改
ALTER USER atguigu01 QUOTA UNLIMITED
ON users


-- 给at01用户，表空间权限；UNLIMITED表示无限制
ALTER USER at01 QUOTA UNLIMITED(5m)
ON users    -- 表空间
```

创建角色并赋予权限

创建角色

角色权限可以赋给用户权限

```
-- 语法
CREATE ROLE manager;

-- 创建my_role角色
create role my_role;
```

为角色赋予权限

```
-- 语法
GRANT create table, create view
TO manager;

-- 赋予创建会话，创建表，创建视图三个系统权限给指定角色
grant create session, create table, create view to my_role;
```

将角色赋予用户

```
-- 语法
GRANT manager TO DEHAAN, KOCHHAR;

-- 把角色赋给at02用户---->at02用户就有了指定角色的所有权限
grant my_role to at02;
```

修改密码

DBA 可以创建用户和修改密码

用户本人可以使用 ALTER USER 语句修改密码

```
-- 语法
ALTER USER scott
IDENTIFIED BY lion;
```

对象权限

用户的一些数据库权限赋给其他用户

- 不同的对象具有不同的对象权限
- 对象的拥有者拥有所有权限
- 对象的拥有者可以向外分配权限

```
GRANT  object_priv [(columns)]
ON     object
TO     {user|role|PUBLIC}
[WITH GRANT OPTION];   -- 允许to后面的用户授予给其他用户
```

分配对象权限

```
-- 分配表 EMPLOYEES 的查询权限
GRANT  select
ON     employees
TO     sue, rich;

-- 把当前用户所拥有的表的权限的指定权限，赋予给指定用户
grant select, update     --  查询，更新
on emp01
to at01;


-- 分配表中各个列的更新权限
GRANT  update
ON     scott.departments
TO     atguigu
```

WITH GRANT OPTION 和 PUBLIC 关键字

```
-- WITH GRANT OPTION 使用户同样具有分配权限的权利
GRANT  select, insert
ON     departments
TO     scott
WITH   GRANT OPTION;

-- 向数据库中所有用户分配权限
GRANT  select
ON    alice.departments
TO    PUBLIC;
```

查询权限分配情况

数据字典视图	描述
ROLE_SYS_PRIVS	角色拥有的系统权限
ROLE_TAB_PRIVS	角色拥有的对象权限
USER_ROLE_PRIVS	用户拥有的角色
USER_TAB_PRIVS_MADE	用户分配的关于表对象权限
USER_TAB_PRIVS_RECD	用户拥有的关于表对象权限
USER_COL_PRIVS_MADE	用户分配的关于列的对象权限
USER_COL_PRIVS_RECD	用户拥有的关于列的对象权限
USER_SYS_PRIVS	用户拥有的系统权限

```
-- 查看用户拥有的表对象权限
select * from user_tab_privs_recd;
```

收回对象权限

- 使用 REVOKE 语句收回权限
- 使用 WITH GRANT OPTION 子句所分配的权限同样被收回

```
-- 语法
REVOKE {privilege [, privilege...]|ALL}
ON    object
FROM   {user[, user...]|role|PUBLIC}
[CASCADE CONSTRAINTS];

-- 从at01用户中收回，emp01表的查询权限
revoke select
on emp01
from at01;
```

语句	功能
create user	创建用户（通常由 DBA 完成）
grant	分配权限
reate role	创建角色（通常 由 DBA 完成）
alter user	修改用户密码
revoke	收回权限

set运算符

SET 操作符

> **UNION** （联合）**/ UNION ALL**(全部联合)
>
> **INTERSECT**(交集)
>
> **MINUS**(减)
>
> 排序：**ORER BY**

> - 除 UNION ALL之外，系统会自动将重复的记录删除
> - 系统将第一个查询的列名显示在输出中
> - 除 UNION ALL之外，系统自动按照第一个查询中的第一个列的升序排列

UNION 操作符

UNION 操作符返回两个查询的结果集的并集

上下表的列需要对应，个数相同，并且是同一种类型，以上面表的名称为主，包括别名,默认以第一个的列排序

```
SELECT employee_id, job_id
FROM   employees
UNION
SELECT employee_id, job_id
FROM   job_history;


-- 创建两个新表
create table employee01
as
select * from system.employees
where department_id in (70, 80)

create table employee02
as
select * from system.employees
where department_id in (80, 90)

-- union操作，取两个表的数据，如果有相同的就只取一个
select employee_id, department_id
from employee01
union
select employee_id, department_id
from employee02

-- 查询10，50，20号部门pjob_id，department_id并且department_id按10，50，20的顺序排列
select job_id, department_id, 1 "a_de"
from system.employees
where department_id = 10
UNION
select job_id, department_id, 2
from system.employees
where department_id = 50
```

```
UNION
select job_id, department_id, 3
from system.employees
where department_id = 20
order by 1

-- 不让后面排序的数字显示
column a_de noprint;



-- 查询所有员工的Last_name ,department_id和department_name
select last_name, department_id, to_char(null)
from system.employees
UNION
select to_char(null), department_id, department_name
from system.departments
```

UNION ALL 操作符

UNION ALL 操作符返回两个查询的结果集的并 集。对于两 个结果集的重复 部分，不 去 重 。

上下表的列需要对应，个数相同，并且是同一种类型，以上面表的名称为主，包括别名,默认以第一个的列排序

```
SELECT employee_id, job_id, department_id
FROM   employees
UNION ALL
SELECT employee_id, job_id, department_id
FROM   job_history
ORDER BY  employee_id;

-- union all操作，取两个表的数据
select employee_id, department_id
from employee01
union all
select employee_id, department_id
from employee02

-- 指定排序方式
select employee_id, department_id
from employee01
union all
select employee_id, department_id
from employee02
order by employee_id desc
```

INTERSECT 操作符

INTERSECT 操作符返回两个结果集的交集

```
SELECT employee_id, job_id
FROM   employees
INTERSECT
SELECT employee_id, job_id
FROM   job_history;

-- 创建的两个表之间的交集
select employee_id, department_id
from employee01
INTERSECT
select employee_id, department_id
from employee02
(order by employee_id desc)
```

MINUS 操作符

MINUS 操作符：返回 两个结果集 的差集

```
SELECT employee_id,job_id
FROM   employees
MINUS
SELECT employee_id,job_id
FROM   job_history;

-- 创建的两个表的差集
select employee_id, department_id
from employee01
MINUS
select employee_id, department_id
from employee02


-- 查询部门的部门号，其中不包招job_id是"ST_CLERK”的部门号
select department_id
from system.departments
MINUS
select department_id
from system.departments
where job_id = 'ST_CLERK'
```

使用 SET 操作符注意事项

在SELECT 列表中的列名和表达式在**数量**和**数据类型**上要相对应

括号可以改变执行的顺序

ORDER BY 子句:

- 只能在语句的最后出现

- 可以使用第一个查询中的列名, 别名或相对位置

```
-- 相对位置的方式
select employee_id, department_id
from employee01
union
select employee_id, department_id
from employee02
order by 1 desc     -- 第一个字段的倒叙
```

```
-- 当两个表对应不上时的解决方式
select employee_id, department_id, to_char(null)
from employee01
UNION
select to_number(null), department_id, department_name
from system.departments



-- 以以下方式输出
-- I want to
-- study at
-- atguigu.com
select 'study at' as "My Dream", 2 "a_study"
from dual
union
select 'I want to', 1
from dual
union
select 'atguigu.com', 3
from dual
order by 2 asc

-- 不让后面排序的数字显示
column a_study noprint;
```

高级子查询

子查询是嵌套在 SQL 语句中的另一个SELECT 语句

- 子查询 (内查询) 在主查询执行之前执行
- 主查询(外查询)使用子查询的结果

```
SELECT  select_list
FROM    table
WHERE   expr operator (SELECT select_list
                       FROM table);

-- 查询last_name为chen的manager的信息
select employee_id, last_name
from system.employees
```

```
where employee_id = (
                    select manager_id
                    from system.employees
                    where lower(last_name) = 'chen'
                    )

-- 查询工资大于149号员工工资的员工信息
select employee_id, last_name, salary
from system.employees
where salary > (
                    select salary
                    from system.employees
                    where employee_id = 149
                    )
```

多 列子查询

主查询与子查询返回的 多个列 进行比较

- 多列子查询中的比较分为两种:
    - 成对比较
    - 不成对比较

成对比较举例

```
-- 查询与141号或174号员工的manager_id和department_id相同的其他员工的employee_id,
manager_id, department_id

-- 以下成对写法
SELECT  employee_id, manager_id, department_id  -- 需要查询的字段
FROM  system.employees                                                    -- 查询的
表
WHERE  (manager_id, department_id) IN                          -- WHERE条件
                      (SELECT manager_id, department_id -- emp_id包含141和174的管理
员id和所属部门id；为一个假表
                       FROM   system.employees
                       WHERE  employee_id IN (141,174))
AND  employee_id NOT IN (141,174)                                    -- 并且emp_id不包
含141和174本身
```

不成对比较举例

```
-- 查询与141号或174号员工的manager_id和department_id相同的其他员工的employee_id,
manager_id, department_id
-- 不成对写法
select employee_id, manager_id, department_id   -- 需要查询的字段
from system.employees                                                    -- 查询的
表
where manager_id in (
```

```
                    select manager_id                    -- 管理员id包含141和174员工
的管理员id
                    from system.employees e2
                    where employee_id in (141, 174)
                    )
AND department_id in (
                     select department_id            -- 部门id包含141和174的部门id
                    from system.employees
                    where department_id in (select department_id from
system.employees where employee_id in (141, 174))
                     )
and employee_id not in (141, 174)                          -- 并且不包含141和174的
信息
```

在 FROM 子句中使用子查询

```
-- 返回比本部门平均工资高的员工的last_name, department_id, salary及平均工资

SELECT last_name, department_id, salary, (select AVG(salary) from
system.employees e3 WHERE e3.department_id = e1.department_id GROUP BY
department_id)
FROM  system.employees e1
WHERE  salary > (
                SELECT AVG(salary)
                FROM   system.employees e2
                WHERE  e1.department_id = e2.department_id   -- 表示e1和当前e2是一个
部门
                GROUP BY department_id
                )

-- 使用from子句的方式
SELECT last_name, e1.department_id, e1.salary, e2.avg_sal
FROM  system.employees e1,
(-- 使用子查询创建了一个假表
      SELECT department_id, AVG(salary) avg_sal
      FROM   system.employees
      GROUP BY department_id
) e2
WHERE e1.department_id = e2.department_id
AND e1.salary > e2.avg_sal
```

单列 子查询表达式

- 单列子查询表达式是在一行中只返回一列的子查询
- Oracle8*i* 只在下列情况下可以使用, 例如:
  - SELECT 语句 (FROM 和 WHERE 子句)
  - INSERT 语句中的VALUES列表中
- Oracle9*i*中单列子查询表达式可在下列情况下使用:
  - DECODE 和 CASE
  - SELECT 中除 GROUP BY 子句以外的所有子句中

在 CASE 表达式中使用单列子查询

在 ORDER BY 子句中 使用单列子查询

相关 子查询

相关子查询按照一行接一行的顺序执行，主查询的每一行都执行一次子查询

EXISTS 操作符

- EXISTS 操作符检查在子查询中是否存在满足条件的行
- **如果在子查询中存在满足条件的行：**
  - 不在子查询中继续查找
  - 条件返回 TRUE
- **如果在子查询中不存在满足条件的行：**
  - 条件返回 FALSE
  - 继续在子查询中查找

NOT EXISTS 操作符

子查询更新

使用相关子查询依据一个表中的数据更新另一个表的数据

子查询删除

使用相关子查询依据一个表中的数据删除另一个表的 数据

```
```

with子句

- 使用 WITH 子句, 可以避免在 SELECT 语句中重复书写相同的语句块
- WITH 子句将该子句中的语句块执行一次并存储到用户的临时表空间中
- 使用 WITH 子句可以提高查询效率

```
```