OCP (051)课堂笔记

第一部分：SQL 语言基础

第二部分：用户及数据库对象

第三部分：SQL 语言的扩展

第一章、Oracle 命令类别：

数据操纵语言：DML: select; insert; delete; update; merge.

数据定义语言：DDL: create; alter; drop; truncate; rename; comment.

事务控制语言：TCL: commit; rollback; savepoint.

数据控制语言：DCL: grant; revoke.

第二章、SQL 的基本函数

2.1 单行函数与多行函数

单行函数：指一行数据输入，返回一个值的函数。所以查询一个表时，对选择的每一行数据都返回一个结果。

SQL>select empno,lower(ename) from emp;

多行函数：指多行数据输入，返回一个值的函数。所以对表的群组进行操作，并且每组返回一个结果。（典型的是聚合函数）

SQL>select sum(sal) from emp;

2.2 单行函数的几种类型

2.2.1 字符函数
lower('SQL Course')----->sql course 返回小写
upper('sql course')----->SQL COURSE 返回大写
initcap('SQL course')-----> Sql Course 返回首字母大写
concat('good','string')---->good string 拼接　　//只能拼接 2 个字符串
substr('String',1,3)---->Str 从第 1 位开始截取 3 位数
instr('t#i#m#r#a#n#','#',3) --->从第 3 位起始找#字符在那个绝对位置
length('String')---->6 长度
lpad('first',10,'$')左填充
rpad(676768,10,'*')右填充
replace('JACK and JUE','J','BL')---->BLACK and BLUE
trim('m' from 'mmtimranm')---->timran

2.2.2 数值函数
round 对指定的值做四舍五入,round(p,s) s 为正数时，表示小数点后要保留的位数，s 也可以为负数，但意义不大。
round:按指定精度对十进制数四舍五入,如:round(45.923, 1),结果,45.9
round(45.923, 0),结果,46
round(45.923, -1),结果,50

trunc 对指定的值取整 trunc(p,s)
trunc:按指定精度截断十进制数,如:trunc(45.923, 1),结果,45.9
trunc(45.923),结果,45
trunc(45.923, -1),结果, 40

mod 返回除法后的余数
SQL> select mod(100,12) from dual;

```
MOD(100,12)
-----------
          4
```

2.2.3 日期函数

因为日期在 oracle 里是以数字形式存储的，所以可对它进行加减运算，计算是以天为单位。
缺省格式：DD-MON-RR.
可以表示日期范围：（公元前）4712 至（公元）9999

时间格式

```
SQL>select to_date('2003-11-04 00:00:00' ,'YYYY-MM-DD HH24:MI:SS') FROM dual;

SQL> select sysdate+2 from dual;    //当前时间+2day

SQL> select sysdate+2/24 from dual; //当前时间+2hour

SQL> select (sysdate-hiredate)/7 week from emp;    //两个 date 类型差，结果是以天为整数位的实数。

MONTHS_BETWEEN //计算两个日期之间的月数
SQL>select months_between('1994-04-01','1992-04-01') mm from dual;

查找 emp 表中参加工作时间>30 年的员工
SQL>select * from emp where months_between(sysdate,hiredate)/12>30;
```

考点：很容易认为单行函数返回的数据类型与函数类型一致，对于数值函数类型而言的确如此，但字符和日期函数可以返回任何数据类型的值。比如 instr 函数是字符型的，months_between 函数是日期型的，但它们返回的都是数值。

```
ADD_MONTHS      //给日期增加月份
SQL>select add_months('1992-03-01',4) am from dual;

LAST_DAY      //日期当前月份的最后一天
SQL>select last_day('1989-03-28') l_d from dual;

NEXT_DAY      //NEXT_DAY 的第 2 个参数可以是数字 1-7，分别表示周日--周六(考点)，比如要取下一个星期六，则应该是：

SQL>select next_day(sysdate,7) FROM DUAL;
```

ROUND(p,s)，TRUNC(p,s)在日期中的应用，如何舍入要看具体情况，s 是 MONTH 按 30 天计，应该是 15 舍 16 入，s 是 YEAR 则按 6 舍 7 入计算。

```
SQL>
SELECT  empno, hiredate,
     round(hiredate,'MONTH') AS round,
     trunc(hiredate,'MONTH') AS trunc
 FROM     emp
 WHERE    empno=7788;

SQL>
SELECT  empno, hiredate,
     round(hiredate,'YEAR') AS round,
     trunc(hiredate,'YEAR') AS trunc
FROM emp
WHERE empno=7839;
```

2.2.4 几个有用的函数

1）decode 函数和 case 表达式：

实现 sql 语句中的条件判断语句，具有类似高级语言中的 if 语句的功能。
decode 函数源自 oracle，case 表达式源自 sql 标准，实现功能类似，decode 语法更简单些。

decode 函数用法：

```
SQL>
SELECT job, sal,
        DECODE(job, 'ANALYST', SAL*1.1,
                    'CLERK',   SAL*1.15,
                    'MANAGER', SAL*1.20,
                     SAL)
                    REVISED_SALARY
FROM emp
/
```

case 表达式第一种用法：

```
SQL>
select job, sal,case job
   when  'ANALYST' then SAL*1.1
   when  'CLERK'   then SAL*1.15
   when  'MANAGER' then SAL*1.20
   else sal end
   REVISED_SALARY
from emp
/
```

case 表达式第二种用法：

```
SQL>
select job, sal,
   case when  job='ANALYST' then SAL*1.1
    when  job='CLERK'   then SAL*1.15
    when  job='MANAGER' then SAL*1.20
   else sal end
   REVISED_SALARY
from emp
/
```

以上三种写法结果都是一样的：

JOB	SAL	REVISED_SALARY
CLERK	800	920
SALESMAN	1600	1600
SALESMAN	1250	1250
MANAGER	2975	3570
SALESMAN	1250	1250
MANAGER	2850	3420
MANAGER	2450	2940
ANALYST	3000	3300
PRESIDENT	5000	5000
SALESMAN	1500	1500
CLERK	1100	1265
CLERK	950	1092.5
ANALYST	3000	3300
CLERK	1300	1495

case 第二种语法比第一种语法增加了搜索功能。形式上第一种 when 后跟定值，而第二种还可以使用比较符。看一个例子

```
SQL>
select ename,sal,
   case when sal>=3000 then '高级'
        when sal>=2000 then '中级'
   else '低级' end
   级别
from emp
```

/

```
ENAME                SAL 级别
---------- ---------- ----
SMITH                800 低级
ALLEN               1600 低级
WARD                1250 低级
JONES               2975 中级
MARTIN              1250 低级
BLAKE               2850 中级
CLARK               2450 中级
SCOTT               3000 高级
KING                5000 高级
TURNER              1500 低级
ADAMS               1100 低级
JAMES                950 低级
FORD                3000 高级
MILLER              1300 低级
```

2）DISTINCT(去重)限定词的用法：　　//distinct 貌似多行函数，严格来说它不是函数。

```
SQL> select distinct job from emp;      //消除表行重复值。

JOB
---------
CLERK
SALESMAN
PRESIDENT
MANAGER
ANALYST

SQL> select distinct job,deptno from emp;   //重复值是后面的字段组合起来考虑的

JOB            DEPTNO
--------- ----------
MANAGER            20
PRESIDENT          10
CLERK              10
SALESMAN           30
ANALYST            20
MANAGER            30
MANAGER            10
```

```
CLERK                30
CLERK                20
```

3）CHR()函数和 ASCII()函数

chr()函数将 ASCII 码转换为字符： ASCII 码 -》 字符

ascii()函数将字符转换为 ASCII 码：字符 -》 ASCII 码

在 oracle 中 chr()函数和 ascii()是一对反函数。

求字符对应的 ASCII 值

```
SQL> select ASCII('A') FROM dual;

ASCII('A')
----------
        65

SQL> select chr(65) from dual;

C
-
A
```

4) sys_context 获取环境上下文的函数（很有用）

scott 远程登录

```
select SYS_CONTEXT('USERENV','IP_ADDRESS') from dual;
--------------------------------------------------------------------------------
192.168.0.136

SQL> select sys_context('userenv','sid') from dual;

SYS_CONTEXT('USERENV','SID')
--------------------------------------------------------------------------------
129

SQL> select sys_context('userenv','terminal') from dual;

SYS_CONTEXT('USERENV','TERMINAL')
--------------------------------------------------------------------------------
TIMRAN-222C75E5
```

5）处理空值的几种函数（见第四章）

第三章、SQL 的数据类型

3.1 四种基本的常用数据类型

1、字符型， 2、数值型，3、日期型，4、大对象型

3.1.1 字符型：

char 固定字符，最长 2000 个

varchar2 可变长字符，最长 4000 个,最小值是 1

nchar/nvarchar2 NCHAR/NVARCHAR2 类型的列使用国家字符集

raw 和 long raw 固定/可变长度的二进制数据长度 最 2G,可存放多媒体图象声音等。（老类型，逐步淘汰）

LONG 可变长的字符串数据，最长 2G，LONG 具有 VARCHAR2 列的特性，一个表中最多一个 LONG 列。（老类型，逐步淘汰）。

3.1.2 数值型：

number（p,s） 实数类型，以可变长度的内部格式来存储数字。这个内部格式精度可以高达 38 位。

int 整数型，number 的子类型，范围同上

3.1.3 日期型：

date 日期的普通形式，表示精度只能到秒级。

timestamp 日期的扩展形式，表示精度可达秒后小数点 9 位（10 亿分之 1 秒）。

timestamp with timezone 带时区

timestamp with local timezone 时区转换成本地日期

系统安装后，默认日期格式是 DD-MON-RR, RR 和 YY 都是表示两位年份，但 RR 是有世纪认知的，它将指定日期的年份和当前年份比较后确定年份是上个世纪还是本世纪（如表）。

当前年份	指定日期	RR 格式	YY 格式
1995	27-OCT-95	1995	1995
1995	27-OCT-17	2017	1917
2001	27-OCT-17	2017	2017
2013	27-OCT-95	1995	2095

3.1.4 LOB 型：大对象是 10g 引入的，在 11g 中又重新定义，在一个表的字段里存储大容量数据，所有大对象最大都可能达到 4G

CLOB: 用来存储单字节的字符数据，包含在数据库内。
NCLOB:用来存储多字节的字符数据
BLOB：用于存储二进制数据，包含在数据库内。
BFILE:存储在数据库之外的二进制文件中，这个文件中的数据只能被只读访问。

CLOB，NCLOB，BLOB 都是内部的 LOB 类型，没有 LONG 只能有一列的限制(考点)。

保存图片或电影使用 BLOB 最好、如果是小说则使用 CLOB 最好。
虽然 LONG RAW 也可以使用，但 LONG 是 oracle 将要废弃的类型，因此建议用 LOB。
当然说将要废弃，但还没有完全废弃，比如 oracle 11g 里的重要视图 dba_views，对于 text(视图定义）仍然沿用了 LONG 类型。

Oracle 11g 重新设计了大对象,推出 SecureFile Lobs 的概念,相关的参数是 db_securefile,采用 SecureFile Lobs 的前提条件是 11g 以上版本,ASSM 管理等,符合这些条件的 BasicFile Lobs 也可以转换成 SecureFile Lobs。较之过去的 BasicFile Lobs, SecureFile Lobs 有几项改进：

1）压缩，2）去重，3）加密。

当 create table 定义 LOB 列时，也可以使用 LOB_storage_clause 指定 SecureFile Lobs 或 BasicFile Lobs
而 LOB 的数据操作则使用 Oracle 提供的 DBMS_LOB 包，通过编写 PL/SQL 块完成 LOB 数据的管理。

3.2 数据类型的转换

隐性类型转换和显性类型转换。

3.2.1 隐性类型转换：

是指 oracle 自动完成的类型转换。在一些带有明显意图的字面值上，可以由 Oracle 自主判断进行数据类型的转换。 如：

```
SQL> select empno,ename from emp where empno='7788';    //empno 本来是数值类型的，这里字符'7788'隐性转换成数值 7788

     EMPNO ENAME
---------- ----------
      7788 SCOTT

SQL> select length(sysdate) from dual;       //将 data 型隐转成字符型后计算长度

LENGTH(SYSDATE)
```

```
---------------
             19

SQL> SELECT '12.5'+11 FROM dual;          // 将字符型‘12.5’隐转成数字型再求和

 '12.5'+11
----------
      23.5

SQL> SELECT 10+('12.5'||11) FROM dual;       //将数字型 11 隐转成字符与‘12.5’合并，其结果再隐转数字型与 10 求和

10+('12.5'||11)
---------------
         22.511
```

3.2.2 显性类型转换是强制完成类型转换（推荐），转换函数形式有三种：

TO_CHAR
TO_DATE
TO_NUMBER

1）日期-->字符

```
SQL> select ename, hiredate, to_char(hiredate, 'DD-MON-YY') month_hired from emp where ename='SCOTT';

ENAME      HIREDATE            MONTH_HIRED
---------- ------------------- --------------
SCOTT      1987-04-19 00:00:00 19-4 月 -87
```

fm 压缩空格或左边的'0'

```
SQL> select ename, hiredate, to_char(hiredate, 'fmyyyy-mm-dd') month_hired from emp where ename='SCOTT';

ENAME      HIREDATE            MONTH_HIRED
---------- ------------------- ------------
SCOTT      1987-04-19 00:00:00 1987-4-19
```

//其实 DD-MM-YY 是比较糟糕的一种格式，因为当日期中天数小于 12 时，DD-MM-YY 和 MM-DD-YY 容易造成混乱。

2）数字-->字符：9 表示数字，L 本地化货币字符

```
SQL> select ename, to_char(sal, 'L99,999.99') Salary from emp where ename='SCOTT';

ENAME      SALARY
---------- --------------------
SCOTT                ￥3,000.00
```

3）字符-->日期

```
SQL> select to_date('1983-11-12', 'YYYY-MM-DD') tmp_DATE from dual;

TMP_DATE
--------------------
1983-11-12 00:00:00
```

4）字符-->数字：

```
SQL> SELECT to_number('$123.45','$9999.99') result FROM dual;

    RESULT
----------
    123.45
```

考点：使用 to_number 时如果使用较短的格式掩码转换数字，就会返回错误。不要混淆 to_number 和 to_char 转换。

例如：

```
SQL> select to_number('123.56','999.9') from dual;
select to_number(123.56,'999.9') from dual
                 *
第 1 行出现错误:
ORA-01722: 无效数字

SQL> select to_char(123.56,'999.9') from dual;

TO_CHA
------
 123.6
```

第四章、WHERE 子句中常用的运算符

4.1 运算符及优先级：

算数运算符
*，/，+，-，

逻辑运算符
not，and ,or

比较运算符

单行比较运算 =,>，>=,<,<=，<>
多行比较运算 >any,>all,<any,<all,in，not in
模糊比较 like（配合“%”和“_”）
特殊比较 is null

（）优先级最高

SQL>select ename，job，sal ,comm from emp where job='SALESMAN' OR job='PRESIDENT' AND sal> 1500;

考点：条件子句使用比较运算符比较两个选项，重要的是要理解这两个选项的数据类型。

4.2 用 BETWEEN AND 操作符来查询出在某一范围内的行.

SQL> SELECT ename，sal FROM emp WHERE sal BETWEEN 1000 AND 1500;

//between 低值 and 高值， 包括低值和高值。

4.3 模糊查询及其通配符：

在 where 字句中使用 like 谓词，常使用特殊符号"%"或"_"匹配查找内容，也可使用 escape 可以取消特殊符号的作用。

```
SQL>
create table test (name char(10));
insert into test values ('sFdL');
insert into test values ('AEdLHH');
insert into test values ('A%dMH');
commit;
```

SQL> select * from test;

```
NAME
----------
sFdL
AEdLHH
A%dMH

SQL> select * from test where name like 'A\%%' escape '\';

NAME
----------
A%dMH
```

4.4 ' '和" "的用法：

''内表示字符或日期数据类型，而" " 一般用于别名中有大小写、保留字、空格等场合，引用 recyclebin 中的《表名》也需要" ".

单引号的转义：连续两个单引号表示转义.

```
SQL> select empno||' is Scott''s empno' from emp where empno=7788;

EMPNO||'ISSCOTT''SEMPNO'
--------------------------------------------------------
7788 is Scott's empno
```

4.5 用 IN 操作符来检验一个值是否在一个列表中

```
SQL> SELECT empno, ename, sal, mgr FROM emp WHERE mgr IN (7902, 7566, 7788);
```

4.6 交互输入变量符&和&&的用途：

```
SQL> select empno,ename from emp where empno=&empnumber;
输入 empnumber 的值:  7788

     EMPNO ENAME
---------- ----------
      7788 SCOTT
```

&后面是字符型的，注意单引号问题，可以有两种写法：

```
SQL> select empno,ename from emp where ename='&emp_name';
输入 emp_name 的值:  SCOTT

     EMPNO ENAME
```

```
---------- ----------
      7788 SCOTT

SQL> select empno,ename from emp where ename=&emp_name;
输入 emp_name 的值:  'SCOTT'

     EMPNO ENAME
---------- ----------
      7788 SCOTT
```

&&存储了第一次输入值，使后面的相同的&不再提问，自动取代。

```
SQL> select empno,ename,&&salary from emp where deptno=10 order by &salary;
输入 salary 的值:  sal

     EMPNO ENAME             SAL
---------- ---------- ----------
      7934 MILLER           1300
      7782 CLARK            2450
      7839 KING             5000
```

注：上面给的&salary 已经在当前 session 下存储了，可以使用 undefine salary 解除。

define(定义变量）和 undefine 命令（解除变量）

```
SQL> define --显示当前已经定义的变量（包括默认值）
```

SQL> set define on|off 可以打开和关闭&。

```
SQL> define emp_num=7788    --定义变量
SQL> select empno,ename,sal from emp where empno=&emp_num;

     EMPNO ENAME             SAL
---------- ---------- ----------
      7788 SCOTT            3000

SQL>undefine emp_num        --取消变量
```

如果不想显示“原值”和“新值”的提示，可以使用 set verify on|off 命令

4.7 使用逻辑操作符: AND; OR; NOT

AND 两个条件都为 TRUE ，则返回 TRUE

```
SQL> SELECT empno,ename,job,sal FROM emp WHERE sal>=1100 AND job='CLERK';

     EMPNO ENAME      JOB              SAL
---------- ---------- --------- ----------
      7876 ADAMS      CLERK            1100
      7934 MILLER     CLERK            1300
```

OR 两个条件中任何一个为 TRUE，则返回 TRUE

```
SQL> SELECT empno,ename,job,sal FROM emp WHERE sal>=1100 OR job='CLERK';

     EMPNO ENAME      JOB              SAL
---------- ---------- --------- ----------
      7369 SMITH      CLERK             800
      7499 ALLEN      SALESMAN         1600
      7521 WARD       SALESMAN         1250
      7566 JONES      MANAGER          2975
      7654 MARTIN     SALESMAN         1250
......
```

已选择 14 行。

NOT 如果条件为 FALSE，返回 TRUE

```
SQL> SELECT ename,job FROM emp WHERE job NOT IN ('CLERK','MANAGER','ANALYST');

ENAME      JOB
---------- ---------
ALLEN      SALESMAN
WARD       SALESMAN
MARTIN     SALESMAN
KING       PRESIDENT
TURNER     SALESMAN
```

第五章 分组函数

5.1 最重要的五个分组函数

sum(); avg(); count(); max(); min().

数值类型可以使用所有组函数

```
SQL> select sum(sal) sum, avg(sal) avg, max(sal) max, min(sal) min, count(*) count
from emp;
```

```
       SUM        AVG        MAX        MIN      COUNT
---------- ---------- ---------- ---------- ----------
     29025 2073.21429       5000        800         14
```

MIN(),MAX(),count()可以作用于日期类型和字符类型

```
SQL> select min(hiredate), max(hiredate), min(ename),max(ename),count(hiredate)
from emp;

MIN(HIREDATE)       MAX(HIREDATE)       MIN(ENAME) MAX(ENAME) COUNT(HIREDATE)
------------------- ------------------- ---------- ---------- ---------------
1980-12-17 00:00:00 1987-05-23 00:00:00 ADAMS      WARD                    14
```

COUNT(*)函数返回表中行的总数，包括重复行与数据列中含有空值的行，而其他分组函数的统计都不包括空值的行。（考点）
COUNT(comm)返回该列所含非空行的数量。

```
SQL> select count(*),count(comm) from emp;

  COUNT(*) COUNT(COMM)
---------- -----------
        14           4
```

5.2 在组函数中使用 NVL 函数

```
SQL> select deptno, avg(nvl(comm,0)) from emp group by deptno;

    DEPTNO AVG(NVL(COMM,0))
---------- -----------------
        30        366.666667
        20                 0
        10                 0

SQL> select deptno, avg(comm) from emp group by deptno;

    DEPTNO  AVG(COMM)
---------- ----------
        30         550
        20
        10
```

想一想上面两个例子结果为何不一样？

5.3 GROUP BY 创建组

```
SQL>select deptno, avg(nvl(sal,0)) from emp group by deptno;
```

group by 后面的列也叫分组特性，一旦使用了 group by, select 后面只能有两种列，一个是组函数列，而另一个是分组特性列（可选）。

对分组结果进行过滤

```
SQL>select deptno, avg(sal) avgcomm from emp group by deptno having avg(sal)>2000;
SQL>select deptno, avg(sal) avgcomm from emp where avg(sal)>2000 group by deptno;
//错误的,应该使用 HAVING 子句
```

对分组结果排序

```
SQL>select deptno, avg(nvl(sal,0)) avgcomm from emp group by deptno order by
avg(nvl(sal,0));
```

```
    DEPTNO    AVGCOMM
---------- ----------
        30 1566.66667
        20       2175
        10 2916.66667
```

排序的列不在 select 投影选项中也是可以的，这是因为 order by 是在 select 投影前完成的。

*考点：确保 SELECT 列表中除了组函数的项以外，所有列都包含在 GROUP BY 子句中。

5.4 分组函数的嵌套

单行函数可以嵌套任意层，但分组函数最多可以嵌套两层。（考点）

比如：count(sum(avg)))会返回错误“ORA-00935：group function is nested too deeply”.

在分组函数内可以嵌套单行函数，如：要计算各个部门 ename 值的平均长度之和

```
SQL> select sum(avg(length(ename))) from emp group by deptno;

SUM(AVG(LENGTH(ENAME)))
-----------------------
             14.9666667
```

第六章、数据限定与排序

6.1 简单查询语句执行顺序

from, where, group by, having, order by, select

where 限定 from 后面的表或视图，限定的选项只能是表的列或列单行函数或列表达式，where 后不可以直接使用分组函数

```
SQL> select empno,job from emp where sal>2000;
SQL> select empno,job from emp where length(job)>5;
SQL> select empno,job from emp where sal+comm>2000;
```

having 限定 group by 的结果，限定的选项必须是 group by 后的聚合函数或分组列，不可以直接使用 where 后的限定选项。

```
SQL> select sum(sal) from emp group by deptno having deptno=10;
SQL> select deptno,sum(sal) from emp group by deptno having sum(sal)>9000;
```

如果要使用 group by 及 having，有条件的话先使用 where 筛选。

6.2 排序(order by)

1）位置：order by 语句总是在一个 select 语句的最后面。

2）排序可以使用列名，列表达式，列函数，列别名，列位置编号等都没有限制，select 的投影列可不包括排序列，除指定的列位置标号外。

3）升序和降序，升序 ASC（默认），降序 DESC。有空值的列的排序，缺省（ASC 升序）时 null 排在最后面（考点）。

4）混合排序，使用多个列进行排序，多列使用逗号隔开，可以分别在各列后面加升降序。

```
SQL> select ename,sal from emp order by sal;
SQL> select ename,sal salary from emp order by salary;
SQL> select ename,sal salary from emp order by 2;
SQL> select ename,sal,sal+100 from emp order by sal+comm;
SQL> select deptno,avg(sal) from emp group by deptno order by avg(sal) desc;
SQL> select ename,job,sal+comm from emp order by 3 desc nulls first;
SQL> select ename,deptno,job from emp order by deptno asc,job desc;
```

6.3 空值（null）

空值既不是数值 0,也不是字符″″，null 表示不确定。

6.3.1 空值参与运算或比较时要注意几点：

1）空值（null）的数据行将对算数表达式返回空值

```
SQL> select ename,sal,comm,sal+comm from emp;

ENAME                SAL       COMM    SAL+COMM
---------- ---------- ---------- ----------
SMITH                800
ALLEN               1600        300        1900
WARD                1250        500        1750
JONES               2975
MARTIN              1250       1400        2650
BLAKE               2850
CLARK               2450
SCOTT               3000
KING                5000
TURNER              1500          0        1500
ADAMS               1100
JAMES                950
FORD                3000
MILLER              1300

已选择 14 行。

SQL> select sum(sal),sum(sal+comm) from emp;     //为什么 sal+comm 的求和小于 sal 的求和?

  SUM(SAL) SUM(SAL+COMM)
---------- -------------
     29025          7800
```

2）比较表达式选择有空值（null）的数据行时，表达式返回为“假”， 结果返回空行。

```
SQL>  select ename,sal,comm from emp where sal>=comm;

ENAME                SAL       COMM
---------- ---------- ----------
ALLEN               1600        300
WARD                1250        500
TURNER              1500          0
```

3）非空字段与空值字段做"||"时，null 值转字符型""，合并列的数据类型为 varchar2。

```
SQL> select ename, sal||comm from emp;

ENAME      SAL||COMM
---------- --------------------------------------------------------------------
-----------
SMITH      800
ALLEN      1600300
WARD       1250500
JONES      2975
MARTIN     12501400
BLAKE      2850
CLARK      2450
SCOTT      3000
KING       5000
TURNER     15000
ADAMS      1100
JAMES      950
FORD       3000
MILLER     1300
```

已选择 14 行。

4）not in 在子查询中的空值问题（见第八章）

5）外键值可以为 null

6）空值在 where 子句里使用“is null”或“is not null”

```
SQL> select ename,mgr from emp where mgr is null;

SQL> select ename,mgr from emp where mgr is not null;
```

7）空值在 update 语句和 insert 语句可以直接使用“=null” 赋值

```
SQL> update emp set comm=null where empno=7788;
```

6.3.2 处理空值的几种函数方法：

1）nvl(expr1,expr2)
当第一个参数不为空时取第一个值，当第一个值为 NULL 时，取第二个参数的值。

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

  NVL(1,2)
----------
         1

SQL> select nvl(null,2) from dual;

NVL(NULL,2)
-----------
         2
```

*考点：nvl 函数可以作用于数值类型，字符类型，日期类型，但数据类型尽量匹配。

```
NVL(comm,0)
NVL(hiredate,'1970-01-01')
NVL(ename,'no manager')
```

2）nvl2(expr1,expr2,expr3)

当第一个参数不为 NULL，取第二个参数的值，当第一个参数为 NULL，取第三个数的值。

```
SQL> select nvl2(1,2,3) from dual;

NVL2(1,2,3)
-----------
         2

SQL> select nvl2(null,2,3) from dual;

NVL2(NULL,2,3)
--------------
            3

SQL> select ename,sal,comm,nvl2(comm,SAL+COMM,SAL) income,deptno from emp where
deptno in (10,30);

ENAME             SAL       COMM     INCOME     DEPTNO
---------- ---------- ---------- ---------- ----------
ALLEN            1600        300       1900         30
WARD             1250        500       1750         30
MARTIN           1250       1400       2650         30
BLAKE            2850                  2850         30
```

CLARK	2450		2450	10
KING	5000		5000	10
TURNER	1500	0	1500	30
JAMES	950		950	30
MILLER	1300		1300	10

*考点：nvl 和 nvl2 中的第二个参数不是一回事。

3）NULLIF（expr1,expr2） /*比对两个值是否一样，一样就返回为空，否则不会为空*/
当第一个参数和第二个参数相同时，返回为空，当第一个参数和第二个数不同时，返回第一个参数值，第一个参数值不允许为 null

```
SQL> select nullif(2,2) from dual;

NULLIF(2,2)
-----------

SQL> select nullif(1,2) from dual;

NULLIF(1,2)
-----------
          1
```

4）coalesce（expr1,expr2.....）返回从左起始第一个不为空的值，如果所有参数都为空，那么返回空值。

```
SQL> select coalesce(1,2,3,4) from dual;


COALESCE(1,2,3,4)
-----------------
                1

SQL> select coalesce(null,2,null,4) from dual;


COALESCE(NULL,2,3,4)
--------------------
                   2
```

第七章、复杂查询之一：多表连接技术

7.1 简单查询的解析方法：

全表扫描：指针从第一条记录开始，依次逐行处理，直到最后一条记录结束；

横向选择+纵向投影=结果集

7.2 多表连接

交叉连接（笛卡尔积）
非等值连接（内连）
等值连接(内连)
外连接　（内连的扩展，左外，右外，全连接）
自连接
自然连接（内连，隐含连接条件,自动匹配连接字段）
集合运算 （多个结果集进行并、交、差）

范例：

```
create table a (id int, name char(10));
create table b (id int, loc char(10));

insert into a values (1,'a');
insert into a values (2,'b');
insert into a values (2,'c');
insert into a values (4,'d');

insert into b values (1,'A');

insert into b values (2,'B');
insert into b values (3,'C');
commit;

SQL> select * from a;

        ID NAME
---------- ----------
         1 a
         2 b
         2 c
         4 d

SQL> select * from b;

        ID LOC
```

---------- ----------
1 A
2 B
3 C

7.2.1 交叉连接（笛卡尔积）
连接条件无效或被省略，两个表的所有行都发生连接，所有行的组合都会返回（n*m）

SQL99 写法：
SQL> select * from a cross join b;

oracle 写法：
SQL> select * from a,b;

SQL> select * from a cross join b;

ID	NAME	ID	LOC
1	a	1	A
2	b	1	A
2	c	1	A
4	d	1	A
1	a	2	B
2	b	2	B
2	c	2	B
4	d	2	B
1	a	3	C
2	b	3	C
2	c	3	C
4	d	3	C

已选择 9 行。

非等值连接：（连接条件非等值，也属于内连范畴）
SQL> select empno,ename,sal,grade,losal,hisal from emp,salgrade where sal between losal and hisal;

EMPNO	ENAME	SAL	GRADE	LOSAL	HISAL
7369	SMITH	800	1	700	1200
7900	JAMES	950	1	700	1200
7876	ADAMS	1100	1	700	1200
7521	WARD	1250	2	1201	1400

7654	MARTIN	1250	2	1201	1400
7934	MILLER	1300	2	1201	1400
7844	TURNER	1500	3	1401	2000
7499	ALLEN	1600	3	1401	2000
7782	CLARK	2450	4	2001	3000
7698	BLAKE	2850	4	2001	3000
7566	JONES	2975	4	2001	3000
7788	SCOTT	3000	4	2001	3000
7902	FORD	3000	4	2001	3000
7839	KING	5000	5	3001	9999

7.2.2 等值连接，典型的内连接

SQL99 写法：

```
SQL> select * from a inner join b on a.id=b.id;
```

oracle 写法：

```
SQL> select * from a,b where a.id=b.id;
```

ID	NAME	ID	LOC
1	a	1	A
2	b	2	B
2	c	2	B

7.2.3 外连接包括左外连接，右外连接，全外连接

1）左外连接

SQL99 语法:

```
SQL> select * from a left join b on a.id=b.id;
```

oracle 语法:

```
SQL> select * from a,b where a.id=b.id(+);
```

结果：

ID	NAME	ID	LOC
1	a	1	A
2	c	2	B
2	b	2	B
4	d		

2）右外连接

SQL99 语法:

```
SQL>select * from a right join b on a.id=b.id;
```

oracle 语法:

```
SQL> select * from a,b where a.id(+)=b.id;
```

结果

```
        ID NAME                ID LOC
---------- ---------- ---------- ----------
         1 a                   1 A
         2 b                   2 B
         2 c                   2 B
                               3 C
```

3）全外连接

SQL99 语法:

```
SQL> select * from a full join b on a.id=b.id;
```

```
        ID NAME                ID LOC
---------- ---------- ---------- ----------
         1 a                   1 A
         2 b                   2 B
         2 c                   2 B
         4 d
                               3 C
```

oracle 语法:

```
SQL> select * from a,b where a.id=b.id(+)
     union
     select * from a,b where a.id(+)=b.id;
```

```
        ID NAME                ID LOC
---------- ---------- ---------- ----------
         1 a                   1 A
         2 b                   2 B
         2 c                   2 B
         4 d
                               3 C
```

7.2.4 自连接
sql99 语法：

```
SQL> select * from a cross join a;
```

oracle 语法:

```
SQL> select * from a a1,a a2;
```

```
        ID NAME                ID NAME
---------- ---------- ---------- ----------
         1 a                    1 a
         1 a                    2 b
         1 a                    2 c
         1 a                    4 d
         2 b                    1 a
         2 b                    2 b
         2 b                    2 c
         2 b                    4 d
         2 c                    1 a
         2 c                    2 b
         2 c                    2 c
         2 c                    4 d
         4 d                    1 a
         4 d                    2 b
         4 d                    2 c
         4 d                    4 d
```

已选择 16 行。

7.2.5 自然连接（属于内连中等值连接）

在 oralce 中使用 natural join,也就是自然连接。

先看自然连接：

```
SQL> select * from a natural join b;
```

```
        ID NAME       LOC
---------- ---------- ----------
         1 a          A
         2 b          B
         2 c          B
```

-----将两个表分别再加一个列 ABC 后，则有两个公共列（ID 列和 ABC 列），添加数据后，再尝试自然连接如何匹配。

```
SQL> select * from a;

        ID NAME       ABC
---------- ---------- ----------
         1 a          s
         2 b          t
         2 c          u
         4 d          v

SQL> select * from b;

        ID LOC        ABC
---------- ---------- ----------
         1 A          w
         2 B          t
         3 C          r

SQL> select * from a natural join b;

        ID ABC        NAME       LOC
---------- ---------- ---------- ----------
         2 t          b          B
```

在自然连接中可以使用 using 关键字：

当使用 natraul join 关键字时，如果两张表中有多个字段，它们具有相同的名称和数据类型，那么这些字段都将被 oracle 自作主张的将他们连接起来。但如果名称相同，类型不同，或者当你需要在多个字段同时满足连接条件的情况下，想人为指定某个（些）字段做连接，那么可以使用 using 关键字。

在 oracle 连接(join)中使用 using 关键字

```
SQL> select id,a.abc,name,loc from a join b using(id);

        ID ABC        NAME       LOC
---------- ---------- ---------- ----------
         1 s          a          A
         2 t          b          B
         2 u          c          B
```

using 里未必只能有一列

```
SQL> select id,abc,name,loc from a join b using(id,abc);

        ID ABC        NAME       LOC
---------- ---------- ---------- ----------
         2 t          b          B
```

总结：

1、使用 using 关键字时，如果 select 的结果列表项中包含了 using 关键字所指明的那个关键字，那么，不要指明该关键字属于哪个表（考点）。
2、using 中可以指定多个列名。
3、natural 和 using 关键字是互斥的，也就是说不能同时出现。

在实际工作中看，内连接，左外连接，以及自然连接用的较多，而且两表连接时一般是多对一的情况居多，即 a 表行多，b 表行少， 从连接字段来看，b 表为父表，其连接字段做主键，a 表为子表，其连接字段为外键。

典型的就是 dept 表和 emp 表的关系，两表连接字段是 deptno，建有主外键关系。

这与数据库设计要求符合第三范式有关。

7.3 集合运算

Union，对两个结果集进行并集操作，不包括重复行，同时进行默认规则的排序；

Union All，对两个结果集进行并集操作，包括所有重复行，不进行排序；

Intersect，对两个结果集进行交集操作，不包括重复行，同时进行默认规则的排序；

Minus，对两个结果集进行差操作，不包括重复行，同时进行默认规则的排序。

集合操作有　并，交，差　3 种运算。

举例：

```
SQL> create table dept1 as select * from dept where rownum <=1;

SQL> insert into dept1 values (80, 'MARKTING', 'BEIJING');
SQL> select * from dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
```

```
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON


SQL> select * from dept1;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        80 MARKTING       BEIJING
```

7.3.1 union

```
SQL>
select * from dept
union
select * from dept1;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        80 MARKTING       BEIJING
```

7.3.2 union all

```
SQL>
select * from dept
union all
select * from dept1;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        10 ACCOUNTING     NEW YORK
        80 MARKTING       BEIJING
```

特别注意：可以看出只有 union all 的结果集是不排序的。

7.3.3 intersect

```
SQL>
select * from dept
intersect
select * from dept1;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
```

7.3.4 minus （注意谁 minus 谁）

```
SQL>
select * from dept
minus
select * from dept1;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON


SQL>
select * from dept1
minus
select * from dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        80 MARKTING       BEIJING
```

7.4 集合运算中的几点注意事项

1） 集合运算中列名不必相同，但要类型匹配且顺序要对应，大类型对上就行了，比如 char 对 varchar2，date 对 timestamp 都可以，字段数要等同，不等需要补全。

```
create table a (id_a int,name_a char(10));
create table b (id_b int,name_b char(10),sal number(10,2));

insert into a values (1, 'sohu');
insert into a values (2, 'sina');

insert into b values (1, 'sohu', 1000);
insert into b values (2, 'yahoo', 2000);
commit;

SQL> select * from a;

      ID_A NAME_A
---------- ----------
         1 sohu
         2 sina

SQL> select * from b;

      ID_B NAME_B            SAL
---------- ---------- ----------
         1 sohu             1000
         2 yahoo            2000

SQL>
select id_a,name_a from a
union
select id_b,name_b from b;

      ID_A NAME_A
---------- ----------
         1 sohu
         2 sina
         2 yahoo
```

2）多表可以复合集合运算，四种运算符按自然先后顺序，如有特殊要求可以使用（）。

3）关于 order by 使用别名排序的问题：

a）缺省情况下，集合运算后的结果集是按所有字段的组合进行排序的（除 union all 外）

如果不希望缺省的排序，也可以使用 order by 显示排序

```
select id_a, name_a name from a
```

```
union
select id_b, name_b name from b
order by name;

      ID_A NAME
---------- ----------
         2 sina
         1 sohu
         2 yahoo

select id_a, name_a from a
union
select id_b, name_b from b
order by 2;

      ID_A NAME_A
---------- ----------
         2 sina
         1 sohu
         2 yahoo
```

b) 显式 order by 是参照第一个 select 语句的列元素。所以，order by 后的列名只能是第一个 select 使用的列名、别名、列号（考点）。

如果是补全的 null 值需要 order by，则需要使用别名。

```
SQL>
select id_a, name_a name,to_number(null) from a
union
select id_b, name_b name,sal from b
order by sal;

ORA-00904: "SAL": 标识符无效


SQL>
select id_a, name_a name,to_number(null) from a
union
select id_b, name_b name,sal from b
order by 3;


      ID_A NAME       TO_NUMBER(NULL)
---------- ---------- ---------------
```

```
         1 sohu                    1000
         2 yahoo                   2000
         1 sohu
         2 sina

SQL>
select id_b, name_b name,sal from b
union
select id_a, name_a name,to_number(null) from a
order by sal;

      ID_B NAME                 SAL
---------- ---------- ----------
         1 sohu                1000
         2 yahoo               2000
         1 sohu
         2 sina

SQL>
select id_a, name_a name,to_number(null) aa from a
union
select id_b, name_b name,sal aa from b
order by aa;

      ID_A NAME                  AA
---------- ---------- ----------
         1 sohu                1000
         2 yahoo               2000
         1 sohu
         2 sina
```

4)，排序是对结果集的排序（包括复合集合运算），不能分别对个别表排序，order by 只能出现一次且在最后一行；

```
SQL>
select id_a, name_a from a order by id_a
union
select id_b, name_b from b order by id_b;

ORA-00933: SQL 命令未正确结束
```

第八章、复杂查询(下)：子查询

子查询返回的值可以被外部查询使用，这样的复合查询等效与执行两个连续的查询。

8.1 单行单列子查询　采用单行比较运算符是（>，<，=，<>,>=，<=)

内部 SELECT 子句只返回一行结果

```
SQL>select ename,sal
from emp
where sal > (
     select sal from emp
     where ename='JONES')
/
```

和员工 7369 从事相同工作并且工资大于员工 7876 的员工的姓名和工作

```
SQL>select ename,job, sal
from emp
where job=(
        select job
        from emp
        where empno=7369
        )
and
       sal > (
        select sal
        from emp
        where empno=7876
        )
/
```

8.2 多行单列子查询，多行比较运算符是（all, any, in，not in）

all　（>大于最大的，<小于最小的）

```
SQL> select ename,sal from emp where sal >all (2000,3000,4000);

ENAME             SAL
---------- ----------
KING              5000
```

查找高于所有部门的平均工资的员工（>比子查询中返回的列表中最大的大才行）

```
SQL> select ename, job, sal from emp where sal > all(select avg(sal) from emp
group by deptno);
```

```
ENAME      JOB              SAL
---------- --------- ----------
JONES      MANAGER         2975
SCOTT      ANALYST         3000
KING       PRESIDENT       5000
FORD       ANALYST         3000
```

```
SQL> select avg(sal) from emp group by deptno;   //子查询结果

  AVG(SAL)
----------
1566.66667
      2175
2916.66667
```

8.3 在多行子查询中使用 any （>大于最小的，<小于最大的）

>any 的意思是：比子查询中返回的列表中最小的大就行，注意和 all 的区别，all 的条件苛刻，any 的条件松阔，
any 强调的是只要有任意一个符合就行了，所以>any 只要比最小的那个大就行了，没必要比最大的还大。

```
select ename, sal from emp where sal >any (2000,3000,4000);
```

```
ENAME             SAL
---------- ----------
JONES             2975
BLAKE             2850
CLARK             2450
SCOTT             3000
KING              5000
FORD              3000
```

8.4 在多行子查询中使用 in （逐个比较是否有匹配值）

```
SQL> select ename, sal from emp where sal in (800,3000,4000);
```

```
ENAME             SAL
---------- ----------
SMITH              800
SCOTT             3000
FORD              3000
```

NOT 运算操作符可以使用在 IN 操作上,但不能使用在 ANY,ALL 操作。

```
SQL> select ename, sal from emp where sal not in (800,3000,4000);

ENAME                SAL
---------- ----------
ALLEN              1600
WARD               1250
JONES              2975
MARTIN             1250
BLAKE              2850
CLARK              2450
KING               5000
TURNER             1500
ADAMS              1100
JAMES               950
MILLER             1300

已选择 11 行。
```

8.5 多行多列子查询，多列子查询是返回多列结果

以 SELECT 主查询的 WHERE 子句中的多个列合并作为成对比较条件。

实验准备

```
SQL>create table emp1 as select * from emp;

SQL>update emp1 set sal=1600,comm=300 where ename='SMITH';  //SMITH是20部门的员工

SQL>update emp1 set sal=1500,comm=300 where ename='CLARK';       //CLARK 是 10 部门的员工

SQL> select * from emp1;

     EMPNO ENAME      JOB              MGR HIREDATE                 SAL
COMM     DEPTNO
---------- ---------- --------- ---------- -------------------- ---------- ------
---- ----------
      7369 SMITH      CLERK             7902 1980-12-17 00:00:00         1600
300         20
      7499 ALLEN      SALESMAN          7698 1981-02-20 00:00:00         1600
```

```
300          30
      7521 WARD          SALESMAN             7698 1981-02-22 00:00:00          1250
500          30
      7566 JONES         MANAGER              7839 1981-04-02 00:00:00          2975
20
      7654 MARTIN        SALESMAN             7698 1981-09-28 00:00:00          1250
1400         30
      7698 BLAKE         MANAGER              7839 1981-05-01 00:00:00          2850
30
      7782 CLARK         MANAGER              7839 1981-06-09 00:00:00          1500
300          10
      7788 SCOTT         ANALYST              7566 1987-04-19 00:00:00          3000
20
      7839 KING          PRESIDENT                 1981-11-17 00:00:00          5000
10
      7844 TURNER        SALESMAN             7698 1981-09-08 00:00:00          1500
0          30
      7876 ADAMS         CLERK                7788 1987-05-23 00:00:00          1100
20
      7900 JAMES         CLERK                7698 1981-12-03 00:00:00           950
30
      7902 FORD          ANALYST              7566 1981-12-03 00:00:00          3000
20
      7934 MILLER        CLERK                7782 1982-01-23 00:00:00          1300
10

已选择 14 行。
```

查询条件：查找 emp1 表中是否有与 30 部门的员工工资和奖金相同的其他部门的员工。
（注意看一下：现在 20 部门的 SIMTH 符合这个条件，它与 30 部门的 ALLEN 有相同的工资和奖金）

多列子查询
特点是主查询每一行中的列都要与子查询返回列表中的相应列同时进行比较，只有各列完全匹配时才显示主查询中的该数据行。

分解一下：

第一步，我们可以先找出 emp1 表中 30 号部门的工资和奖金的结果集，（此例没有对 comm 的空值进行处理）

```
SQL> select sal,comm from emp1 where deptno=30;
```

```
       SAL       COMM
---------- ----------
      1600        300
      1250        500
      1250       1400
      2850
      1500          0
       950
```

已选择 6 行。

第二步，列出 emp1 表中属于这个结果集的所有员工。

```
SQL> select * from emp1 where (sal,comm) in (select sal,comm from emp1 where
deptno=30);

     EMPNO ENAME      JOB              MGR HIREDATE                 SAL
COMM     DEPTNO
---------- ---------- --------- ---------- -------------------- ---------- ------
---- ----------
      7499 ALLEN      SALESMAN         7698 1981-02-20 00:00:00         1600
300         30
      7369 SMITH      CLERK            7902 1980-12-17 00:00:00         1600
300         20
      7521 WARD       SALESMAN         7698 1981-02-22 00:00:00         1250
500         30
      7654 MARTIN     SALESMAN         7698 1981-09-28 00:00:00         1250
1400         30
      7844 TURNER     SALESMAN         7698 1981-09-08 00:00:00         1500
0         30
```

可以这样想：相当于谓词 in（...)中有上面这 6 行内容，所以上面句子相当于：

```
select * from emp1 where (sal,comm) in(
(1600,300),(1250,500),(1250,1400),(2850,null),(1500,0),(950,null)
);
```

第三步， 再去掉 30 号部门后，就显示出了在 emp1 表中与 30 部门中任意一个员工的工资和奖金完全相同的，
但该员工不是来自 30 部门的员工信息。

```
SQL>
select ename,deptno,sal,comm from emp1
  where (sal,comm) in (select sal,comm from emp1 where deptno=30)
and deptno<>30
```

```
/

ENAME           DEPTNO        SAL        COMM
---------- ---------- ---------- ----------
SMITH               20       1600        300
```

考点：1）成对比较是不能使用>any 或>all 等多行单列比较符的。2）成对比较时的多列顺序和类型必须一一对应。

8.6 与非成对比较（含布尔运算）的区别

```
SQL>select ename,deptno,sal,comm
from emp1
where sal in(
        select sal
        from emp1
        where deptno=30)
and
        nvl(comm,0) in (
        select nvl(comm,0)
        from emp1
        where deptno=30)
and deptno<>30
/

ENAME           DEPTNO        SAL        COMM
---------- ---------- ---------- ----------
SMITH               20       1600        300
CLARK               10       1500        300
```

两个子查询返回的值分别与主查询中的 sal 和 comm 列比较，
如果员工的工资与 30 部门任意一个员工相同，同时，奖金也与 30 部门的其他员工相同，那么得到了两个员工的信息。
可见，成对比较(使用 where （列，列))比非成对比较（使用 where 列 and 列）更为严苛。

8.7 关于布尔运算符 not

8.7.1 not 就是否定后面的比较符，基本的形式如下

```
where empno=7788              where NOT (empno=7788)
where ename LIKE 'S%'             where ename NOT LIKE 'S%'
where deptno IN (20,30)           where deptno NOT IN (20,30)
```

where sal BETWEEN 1500 AND 3000　　where sal NOT BETWEEN 1500 AND 3000
where comm IS NULL　　where comm IS NOT NULL
where EXISTS (select 子查询)　　where NOT EXISTS (select 子查询)

8.7.2 not in 在子查询中的空值问题：

"in"与"not in"遇到空值时情况不同，对于"not in" 如果子查询的结果集中有空值，那么主查询得到的结果集也是空。

查找出没有下属的员工,即普通员工,(该员工号不在 mgr 之列的)

SQL>select ename from emp where empno not in (select mgr from emp);

no rows selected

上面的结果不出所料，主查询没有返回记录。这个原因是在子查询中有一个空值，而对于 not in 这种形式，一旦子查询出现了空值，则主查询记录结果也就返回空了。

注意：not 后不能跟单行比较符，只有 not in 组合，没有 not any 和 not all 的组合，但 not 后可以接表达式 如：

where empno not in(...) 与 where not empno in(...)两个写法都是同样结果，前者是 not in 组合，后者是 not 一个表达式。

例：排除空值的影响

SQL>select ename from emp where empno not in (select nvl(mgr,0)from emp);

8.8 from 子句中使用子查询(这个也叫内联视图)

例：员工的工资大于他所在的部门的平均工资的话，显示其信息。

分两步来考虑：

第一步，先看看每个部门的平均工资，再把这个结果集作为一个内联视图。

SQL> select deptno,avg(sal) salavg from emp group by deptno;

DEPTNO	SALAVG
30	1566.66667
20	2175
10	2916.66667

第二步，把这个内联视图起一个别名 b，然后和 emp 别名 e 做连接，满足条件即可。

```
SQL>
select e.ename, e.sal, e.deptno, b.salavg
from emp e, (select deptno,avg(sal) salavg from emp group by deptno) b
where e.deptno=b.deptno and e.sal > b.salavg
/
```

ENAME	SAL	DEPTNO	SALAVG
ALLEN	1600	30	1566.66667
JONES	2975	20	2175
BLAKE	2850	30	1566.66667
SCOTT	3000	20	2175
KING	5000	10	2916.66667
FORD	3000	20	2175

8.9 关联子查询与非关联子查询

从主查询（外部）调用子查询（内部）来看，可以有关联与非关联子查询之分

8.9.1 非关联子查询：

子查询部分可以独立执行，Oracle 的 in 子查询一般用于非关联子查询，执行过程是这样的，首先执行子查询，并将获得的结果列表存放在一个加了索引的临时表中。也就是说在执行子查询之前，系统先将主查询挂起，待子查询执行完毕，存放在临时表中以后再执行主查询。

例：得到 30 号部门工资高于本部门平均工资的员工的信息

```
SQL> select ename,sal,deptno from emp where deptno=30 and sal>(select avg(sal)
from emp where deptno=30);
```

8.9.2 关联子查询：

其子查询（内部，inner)会引用主查询(外部，outer)查询中的一列或多列。在执行时,外部查询的每一行都被一次一行地传递给子查询.子查询依次读取外部查询传递来的每一值,并将其用到子查询上,直到外部查询所有的行都处理完为止.然后返回查询结果.。

```
SQL> create table emp1 as (select e.empno,e.ename,d.loc,d.deptno from emp e,dept
d where e.deptno=d.deptno(+));
SQL> update emp1 set loc=null;
```

```
SQL> commit;
```

如何通过关联查询再将 emp1 表更新回原值。

```
SQL> update emp1 e set loc=(select d.loc from dept d where e.deptno=d.deptno);
```

在关联查询中使用 EXISTS 和 NOT EXISTS

EXISTS 关心的是在子查询里能否找到一个行值（哪怕有 10 行匹配，只要找到一行就行），如果子查询有行值，则立即停止子查询的搜索，然后返回逻辑标识 TRUE, 如果子查询没有返回行值，则返回逻辑标识 FALSE， 子查询要么返回 T，要么返回 F，以此决定了主查询的调用行的去留，然后主查询指针指向下一行，继续调用子查询...

EXISTS 的例子：显示出 emp 表中那些员工不是普通员工（属于大小领导的）。

```
SQL> select empno,ename,job,deptno from emp outer where exists(select 'X' from
emp where mgr=outer.empno);

     EMPNO ENAME      JOB              DEPTNO
---------- ---------- --------- ----------
      7566 JONES      MANAGER            20
      7698 BLAKE      MANAGER            30
      7782 CLARK      MANAGER            10
      7788 SCOTT      ANALYST            20
      7839 KING       PRESIDENT          10
      7902 FORD       ANALYST            20

已选择 6 行。
```

说明：exists 子查询中 select 后的‘X’只是一个占位，它返回什么值无关紧要，它关心的是子查询中否‘存在’，即子查询的 where 条件能否有‘结果’，一旦子查询查到一条记录满足 where 条件，则立即返回逻辑‘TRUE’，(就不往下查了)。否则返回‘FALSE’。

NOT EXISTS 的例子：显示 dept 表中还没有员工的部门。

```
SQL> select deptno,dname from dept d where not exists (select 'X' from emp where
deptno=d.deptno);

    DEPTNO DNAME
---------- --------------
        40 OPERATIONS
```

对于关联子查询，在某种特定的条件下，比如子查询是个大表，且连接字段建立了索引，那

么使用 exists 比 in 的效率可能更高。

8.10 关于别名的使用

有表别名和列别名，表别名用于多表连接或子查询中，列别名用于列的命名规范。

如果别名的字面值有特殊字符，需要使用双引号。如："AB C"

8.10.1 必须使用别名的地方：

1）两表连接后，select 投影中有相同命名的列，必须使用表别名区别标识（自然连接中的公共列则使用相反原则）

```
select ename,d.deptno from emp e,dept d where e.deptno=d.deptno;
```

2）使用 create * {table |view} as select ...语句创建一个新的对象，其字段名要符合对象中字段的规范，不能是表达式或函数等非规范字符，而使用别名可以解决这个问题。

```
create table emp1 as select deptno,avg(sal) salavg from emp group by deptno;
create view v as select deptno,avg(sal) salavg from emp group by deptno;
```

3）使用内联视图时， 若 where 子句还要引用其 select 中函数的投影， 使用别名可以派上用场。

```
select * from (select avg(sal) salavg from emp) where salavg>2000;
```

4）当以内联视图作为多表连接，主查询投影列在形式上不允许单行字段（或函数）与聚合函数并列，解决这个问题是使在内联视图中为聚合函数加别名，然后主查询的投影中引用其别名。

```
select e.ename,e.sal,b.deptno,b.salavg
       from emp e,(select deptno,avg(sal) salavg from emp group by deptno) b
       where e.deptno=b.deptno;
```

5）rownum 列是 Oracle 的伪列，加别名可以使它成为?一个表列，这样才可以符合 SQL99 标准中的连接和选择。

```
SQL> select * from (select ename,rownum rn from emp) where rn>5;
```

8.10.2 不能使用别名的地方：

在一个独立的 select 结构的投影中使用了列别名，不能在其后的 where 或 having 中直接引用该列别名（想想为什么？）。

```
select ename,sal salary from emp where salary>2000;                    --错

select deptno,avg(sal) salavg from emp group by deptno having salavg>2000; --错
```

8.11 简单查询与复杂查询练习题：

1）列出 emp 表工资最高的前三名员工信息

```
select * from (select * from emp order by sal desc) where rownum < 4;
```

关于 rownum 伪列使用特别需要注意两点：

1，rownum>时不会返回任何行
2，rownum< 和 and 并用时，是在另一个条件基础上的 rownum< ，而不是两个独立条件的并集（intersect）

体会一下：

```
SQL> select ename,sal,deptno from emp where deptno=10;

ENAME             SAL     DEPTNO
---------- ---------- ----------
CLARK             2450         10
KING              5000         10
MILLER            1300         10

SQL> select ename,sal,deptno from emp where rownum <=1;

ENAME             SAL     DEPTNO
---------- ---------- ----------
SMITH              800         20

SQL> select ename,sal,deptno from emp where rownum <=1 and deptno=10;

ENAME             SAL     DEPTNO
---------- ---------- ----------
CLARK             2450         10
```

2) 列出 emp 表第 5-第 10 名员工（按 sal 大--小排序）的信息（结果集的分页查询技术）

```
SQL> select * from (select t1.*, rownum rn from (select * from emp order by sal
```

```
desc) t1) where rn between 5 and 10;

EMPNO ENAME      JOB            MGR HIREDATE            SAL       COMM DEPTNO
RN
----- ---------- --------- ----- ---------- --------- --------- ------ -------
--
 7698 BLAKE      MANAGER      7839 1981-5-1       2850.00                30
5
 7782 CLARK      MANAGER      7839 1981-6-9       2450.00                10
6
 7499 ALLEN       SALESMAN    7698 1981-2-20       1600.00     300.00     30
7
 7844 TURNER     SALESMAN     7698 1981-9-8        1500.00       0.00     30
8
 7934 MILLER     CLERK        7782 1982-1-23      1300.00                10
9
 7521 WARD        SALESMAN    7698 1981-2-22       1250.00     500.00     30
10

6 rows selected
```

3）从列出 emp 表中显示员工和经理对应关系表。(emp 自连,利用笛卡尔积)

```
select a.empno,a.ename,a.mgr , b.empno,b.ename, b.mgr
from emp a, emp b
where a.mgr=b.empno;
```

4)要求列出 emp 表中最高工资的员工所在工作地点，(emp+dept 左外)

```
select a.ename, d.loc from
(select * from emp where sal=
(select max(sal) from emp)) a left join dept d on a.deptno=d.deptno ;
```

5）CTAS 方法建立 dept1，将 dept1 表增加一列 person_count,要求根据 emp 表填写 dept1 表的各部门员工合计数（典型个关联查询)。

```
SQL> create table dept1 as select * from dept;
SQL> alter table dept1 add person_count int;
SQL> update dept1 d set person_count=(select count(*) from emp e where
e.deptno=d.deptno);
SQL> select * from dept1;
```

```
    DEPTNO DNAME          LOC            PERSON_COUNT
---------- -------------- -------------- ------------
        10 ACCOUNTING     NEW YORK                  3
        20 RESEARCH       DALLAS                    5
        30 SALES          CHICAGO                   6
        40 OPERATIONS     BOSTON
```

6) 复杂 select 查询，以 HR 用户的几个表为例，显示欧洲地区员工的平均工资及人数（使用多表连接及内联视图）

```
SQL>
select avg(salary),count(salary) from
  (select    e.first_name,e.salary,d.department_id,l.location_id,    c.country_id,
r.region_id,r.region_name
          from employees e,departments d,locations l,countries c,regions r
            where               e.department_id=d.department_id               and
d.location_id=l.location_id
              and l.country_id=c.country_id and c.region_id=r.region_id)
          where region_name='Europe';

AVG(SALARY) COUNT(SALARY)
----------- -------------
 8916.66667            36
```

7）同上题（使用嵌套子查询技术）

```
SQL>
select avg(salary),count(*) from employees where department_id in
(select department_id from departments where location_id in
 (select location_id from locations where country_id in
   (select country_id from countries where region_id=
    (select region_id from regions where region_name='Europe')
   )
 )
);

AVG(SALARY) COUNT(SALARY)
----------- -------------
 8916.66667            36
```

分解如下：

```
SQL> select region_id from regions where region_name='Europe';
```

```
 REGION_ID
----------
         1


SQL> select country_id from countries where region_id=1;

CO
--
BE
CH
DE
DK
FR
IT
NL
UK

SQL>  select  location_id  from  locations  where  country_id  in
('BE','CH','DE','DK','FR','IT','NL','UK');

LOCATION_ID
-----------
       1000
       1100
       2400
       2500
       2600
       2700
       2900
       3000
       3100


SQL>  select  department_id  from  departments  where  location_id  in
(1000,1100,2400,2500,2600,2700,2900,3000,3100);

DEPARTMENT_ID
-------------
           40
           70
           80
```

```
SQL> select avg(salary),count(*) from employees where department_id in(40,70,80);

AVG(SALARY)    COUNT(*)
-----------  ----------
 8916.66667          36
```

第九章 用户访问控制

9.1 创建和管理数据库用户账户

sys 查看数据库里有多少用户？
select username from dba_users;

9.1.1 用户的缺省表空间

每个用户账户都可以指定默认的表空间，用户创建的任何对象（如表或索引）将缺省保存在此表空间中，如果创建用户时没有指定默认表空间，那么属于数据库级的默认表空间（DBCA 创建数据库时默认是 users 表空间）将应用于该账户。

```
SQL> select * from database_properties;          //查看数据库级的默认表空间

SQL> alter database default tablespace tablespace_name; //可以更改
```

9.1.2 空间配额的概念：
配额（quota)是表空间中为用户的对象使用的空间量，dba 建立用户时就应该考虑限制用户的磁盘空间配额，否则无限制配额的用户可能把的表空间撑爆（甚至损坏 system 表空间)。

```
ALTER USER tim QUOTA 10m ON test_tbs;   --设置限额=10m

ALTER USER tim QUOTA                    --不受限制

ALTER USER tim QUOTA 0 ON test_tbs; --收回限额
```

考点：

1）要创建表，用户必须具有执行 create table 的权限，而且拥有在该表使用的表空间上的配额。
2）用户不需要临时表空间上的配额,UNOD 表空间也不能针对用户设置配额。

9.2 管理概要文件(profile)(PPT-I-270-272)

作用是对用户访问数据库做一些限制。有几个要点：

1）概要文件（profile)具有两个功能，一个是实施口令限制，另一个是限制会话可以占用的资源。
2）始终要实施口令控制，而对于资源限制，则只有实例参数 RESOURE_LIMIT 为 TRUE 时（默认是 FALSE)才会实施（考点）。
3）系统自动使用概要文件，有一个默认的 default profile,限制很松，作用较小。
4）可以使用 create profile 为用户创建它自己的概要文件，没有说明的参数就从 default profile 的当前版本中提取。

Password_parameter 部分：

Failed_login_attempts：指定在帐户被锁定之前所允许尝试登陆的的最大次数。
Password_lock_time：在到达 Failed_login_attempts 后锁定账户的天数。
Password_life_time：口令过期前的天数，口令在过期后仍可以使用，具体取决于 Password_grace_time
Password_grace_time：口令过期（将生成更改口令的提示）后第一次成功登录后的天数，在此期间，旧口令仍然可用。
Password_reuse_time：可以重新使用口令前的天数
password_reuse_max：可以重新使用口令的次数
Password_verify_function：更改口令时运行的函数名，此函数一般用于检查新口令所需的复杂程度。

Resource_parameter 部分

Session_per_user：对同一个用户账户执行的并发登录数。
Cpu_per_session：在强制终止会话前，允许会话的服务器进程使用的 CPU 时间（厘秒)。
Cpu_per_call： 在强制终止某条 SQL 语句前，允许会话的服务器进程用于执行此语句的 CPU 时间（厘秒)。
Connect_time： 在强制终止会话前，会话的最长持续时间（分钟)。
Idle_time： 在强制终止会话前，允许会话处于闲置状态的最长时间（分钟)。
Logical_reads_per_session： 在强制终止会话前，会话可读取的块数（无论块在数据缓冲区还是磁盘)。
Logical_read_per_call： 在强制终止单个语句前，此语句可读取的块数（无论块在数据缓冲区还是磁盘)。
Private_sga： 对于通过共享服务器体系结构连接的会话，允许会话的会话数据在 SGA 中占用的字节数(考点)。
Composite_limit：前面几个参数的加权和。这是一个高级功能，其需要的配置不在 OCP 考试范围。

例：

1）创建一个概要文件，如果出现两次口令失误，那么将账户锁定。

```
SQL> create profile two_error limit failed_login_attempts 2;
```

2）将概要文件分配给 tim 用户

```
SQL> alter user tim profile two_error;

SQL> select username,PROFILE from dba_users where username='TIM';

USERNAME                       PROFILE
------------------------------ ------------------------------
TIM                            TWO_ERROR
```

3）tim 尝试两次登录使用错误密码，系统报出 ORA-28000 错误

```
SQL> conn tim/fdfd
ERROR:
ORA-28000: 帐户已被锁定
```

4）sys 为 tim 解锁

```
SQL> conn / as sysdba
SQL> alter user tim account unlock;
```

5）sys 删掉了 two_error 概要文件

```
SQL> drop profile two_error cascade;        //删除 two_error 后，tim 用户又绑定到 default profile 上。
```

//profile 这部分操作参数较多，使用命令有些啰嗦，可以使用 OEM 方式来管理，比较方便。

9.3 更改密码

sys 建立用户时给用户一个密码，用户的密码保存在数据字典中，用户登录户再更改密码。这些密码是加密形式存在的，DBA 也无法知晓。

sys 不知道用户密码，又想临时登录用户账户，但不想打扰用户。怎样做：

```
SQL> select name,password from user$ where name='SCOTT';

NAME                           PASSWORD
------------------------------ ------------------------------
SCOTT                          F894844C34402B67

SQL> alter user scott identified by tiger;

用户已更改。
```

SQL> conn scott/tiger
已连接。
SQL> conn / as sysdba

SQL> alter user scott identified by values 'F894844C34402B67';

用户已更改。

SQL>

9.4 系统权限，对象权限，角色

9.4.1 权限的引入：

数据库安全分为系统安全和数据安全

系统安全：用户名和口令，分配给用户的磁盘空间及用户的系统操作,如 profile 等
数据库安全：对数据库对象的访问及操作

用户具备系统权限才能够访问数据库
具备对象权限才能访问数据库中的对象

简而言之：权限（privilege）：system privilege and object privilege

1） system privilege：针对于 database 的相关权限

系统权限通常由 DBA 授予 （11g 有 200 多种，select distinct privilege from dba_sys_privs;也可被其他用户或角色授予）

典型 DBA 权限
CREATE USER
DROP USER
BACKUP ANY TABLE
SELECT ANY TABLE
CREATE ANY TABLE

典型用户需要的系统权限
CREATE SESSION
CREATE TABLE
CREATE SEQUENCE
CREATE VIEW
CREATE PROCEDURE

2）object privilege：针对于 schema（用户）的 object

对象权限有 8 种：ALTER, DELETE, EXECUTE, INDEX, INSERT, REFERENCES, SELECT, UPDATE

对象权限	表	视图	序列	过程
ALTER	*		*	
DELETE	*	*		
EXECUTE				*
INDEX		*		
INSERT		*	*	
REFERENCES	*			
SELECT		*	*	*
UPDATE		*	*	

9.4.2 权限的授权

授予系统权限语法：

GRANT sys_privs,[role] TO user|role|PUBLIC [WITH ADMIN OPTION]

//授予角色与系统权限的语法格式是一样的,所以可以并列在一个句子里赋权

授予对象权限语法
GRANT object_privs ON object TO user|role|PUBLIC [WITH GRANT OPTION]

创建和删除角色
CREATE role myrole;
DROP role myrole;

9.4.3 角色的引入：

系统权限太繁杂，将系统权限打包成角色，Oracle 建议通过角色授权权限，目的就是为了简化用户访问管理

sys:
SQL> create user tim identified by tim;　　//建一个 tim 用户
SQL> conn tim/tim
ERROR:
ORA-01045: 用户 TIM 没有 CREATE SESSION 权限; 登录被拒绝

警告: 您不再连接到 ORACLE。
SQL> conn / as sysdba

已连接。
SQL> grant create session to tim;　　//授予 tim 系统权限 create session

SQL> conn tim/tim
已连接。

SQL> select * from tab;

未选定行

SQL> create table a (id int);
create table a (id int)
*
第 1 行出现错误:
ORA-01031: 权限不足
sys:
SQL> grant create table to tim;　　//授予 tim 系统权限 create table

tim:
SQL> create table a (id int);
create table a (id int)
*
第 1 行出现错误:
ORA-01950: 对表空间 'USERS' 无权限

sys:
SQL> grant unlimited tablespace to tim;　　// 授 予 tim 系 统 权 限 unlimited tablespace，可以无限制的使用任何表空间

SQL> alter user tim quota 5m on users;　　//仅对于使用 users 表空间加上了磁盘限额。

tim:
SQL> create table a (id int);

表已创建。

SQL> select * from session_privs;　　//这个语句最常用，但其中不包括该用户的对象权限（考点）

PRIVILEGE

CREATE SESSION
UNLIMITED TABLESPACE

CREATE TABLE

列出 oracle 所有系统权限;
SQL> select distinct privilege from dba_sys_privs;

sys:
SQL>drop user tim cascade;
SQL>create user tim identified by tim;
SQL>grant connect,resource to tim;

SQL> select * from session_privs;

PRIVILEGE

CREATE SESSION
UNLIMITED TABLESPACE
CREATE TABLE
CREATE CLUSTER
CREATE SEQUENCE
CREATE PROCEDURE
CREATE TRIGGER
CREATE TYPE
CREATE OPERATOR
CREATE INDEXTYPE

已选择 10 行。

需要注意两点：

1）在 resource 角色里包含了 unlimited tablesapce 系统权限，意思是不限制用户使用任何表空间，此权限太大，它包括可以访问 system 表空间，在实际应用中一般要将此权限收回，然后再对用户限制表空间配额（quoto）。
2）sys 将 resource 角色授权给用户，其中有 9 个系统权限，只有 unlimited tablespace 可以单独收回（从角色里收回系统权限的特例）。

SQL> create tablespace test_tbs datafile '/u01/oradata/timran11g/test01.dbf' size 10m;
SQL> create user tim identified by tim default tablespace test_tbs;
SQL> grant connect, resource to tim;
SQL> revoke unlimited tablespace from tim;
SQL> alter user tim quota 10m on test_tbs;

查看用户表空间配额：
SQL> select tablespace_name,username,max_bytes from DBA_TS_QUOTAS where

username='TIM';

9.4.4 几个有关权限的考点

1）sys, system 拥有普通用户的所有对象权限，并有代理授权资格。
2）系统权限里的 any 含义：
sys:
SQL> grant create any table to tim;
tim:
SQL> create table scott.t100 (id int);

3）对象权限一般由对象拥有者授予，也可以由 sys 或 system 代理授予。
sys:
grant select on scott.emp to tim;

可以使 update 对象权限精确到列：

scott:
SQL> grant select, update(sal) on emp to tim;

SQL> revoke update(sal) on emp from tim;
revoke update(sal) on emp from tim
*
第 1 行出现错误:
ORA-01750: UPDATE/REFERENCES 只能从整个表而不能按列 REVOKE

SQL> revoke update on emp from tim;

授予 Tim 对 scott.emp 的所有对象权限
SQL> create all on scott.emp to tim;

几个需要注意的地方：

系统权限和对象权限语法格式不同，不能混合使用 grant create table,select on emp to tim 错(考点)
系统权限和角色语法相同可以并列授权 grant connect,create table to tim 对
可以一条语句并列授予多个用户 grant connect to tim,ran 对
可以通过授权建立用户,如 ran 用户不存在， grant connect,resource to ran identified by ran; 对

4）系统权限的传递与回收：WITH ADMIN OPTION 选项

sys: 先建立两个测试用户 tim 和 ran

CREATE USER tim IDENTIFIED BY tim;
CREATE USER ran IDENTIFIED BY ran;

GRANT create session TO tim WITH ADMIN OPTION;

tim:
GRANT create session TO ran

收回 tim 系统权限，ran 的系统权限没有收回!
sys:
REVOKE create session FROM tim;

conn tim/tim 可以成功

5）对象权限的传递与回收：WITH GRANT OPTION 选项

scott：为 tim 用户授权

GRANT SELECT ON emp TO tim WITH GRANT OPTION;
tim：
GRANT SELECT ON scott.emp TO ran;

检查 tim 和 ran 都能否访问 scott.emp ....

scott：回收 tim 对象权限
REVOKE SELECT ON emp FROM tim;

检查 tim 和 ran 都不能访问 scott.emp 了，ran 的对象权限也收回!

9.4.5 对象权限在存储过程中的使用

scott 将存储过程 proc1 的 execute 权限赋给 tim，proc1 中包含了一些 tim 没有权限的 DML 操作，那么 tim 能成功地执行存储过程 proc1 吗？这个问题涉及到了 create procedure 时 invoker_rights_clause 的两个选项：（考点）

1、AUTHID CURRENT_USER //执行存储过程时，要检查用户是否有 DML 操作的对象权限。
2、AUTHID DEFINER（默认）

测试：

sys:
SQL> create table scott.a (d1 date);
SQL> grant connect,resource to tim identified by tim;

```
SQL>
scott:
create or replace procedure proc1 as
begin
 insert into scott.a values(sysdate);
 commit;
end;
/

SQL> grant execute on proc1 to tim;

tim:
SQL> exec scott.proc1;

PL/SQL 过程已成功完成。                    //有执行存储过程的权限，但没有 insert 对象权限，竟然也能执行成功。


scott:                          //使用 invoker_rights_clause 加入 AUTHID CURRENT_USER 参数再试试。
create or replace procedure proc1 AUTHID CURRENT_USER
as
begin
 insert into scott.a values(sysdate);
 commit;
end;
/

tim:
SQL> exec scott.proc1;
报错!

scott:
SQL> grant all on a to tim;

tim:
SQL> exec scott.proc1;

PL/SQL 过程已成功完成。                    //在 execute 权限和 insert 权限都具备的情况下使操作成功，这可能是我们想要的。
```

9.4.6 与权限有关的数据字典

SESSION_PRIVS　　//用户当前会话拥有的系统权限
USER_ROLE_PRIVS　　//用户被授予的角色
ROLE_SYS_PRIVS　　//用户当前拥有的角色的系统权限
USER_SYS_PRIVS　　//直接授予用户的系统权限
USER_TAB_PRIVS　　//授予用户的对象权限
ROLE_TAB_PRIVS　　//授予角色的表的权限

练习：要掌握权限与角色的关系，以及如何查看信息，

数据字典
dba_xxx_privs
all_xxx_privs
user_xxx_privs

其中 xxx：role 表示角色，sys 表示系统权限，tab 表示对象权限。

从哪个角度看，非常重要！
我们举个例子：三个用户，分别是 sys,scott,和 tim，
sys:
1)建立 myrole 角色，把 connect 角色和 create table 系统权限以及 update on scott.emp 对象权限放进 myrole。
2)把 myrole 角色授给 tim。
3)把 create table 系统权限授给 tim。

scott:
把 select on emp 表的对象权限授给 tim

如此 tim 用户有了如下角色和权限：

myrole(connect,create table，update on scott.emp)
create table
select on emp

我们从三个角度分析一下，如何里数据字典里查看 tim 拥有的角色和权限信息。

从 dba 角度看：

看用户 tim 所拥有的系统权限
select * from dba_sys_privs where grantee='TIM';

看用户 tim 所拥有的对象权限
select * from dba_tab_privs where grantee='TIM';

看用户 tim 所拥有的角色（不包含角色里的角色）

```
select * from dba_role_privs where grantee='TIM';
```

查看这个角色里包含的角色

```
select * from dba_role_privs where grantee='MYROLE';
```

查看这个角色里包含的系统权限

```
select * from dba_sys_privs where grantee='MYROLE';
```

查看这个角色里包含的对象权限

```
select * from dba_tab_privs where grantee='MYROLE';
```

从当前用户角度看：

查看和自己有关的角色（不含角色中含有的角色）

```
select * from user_role_privs;
```

查看和自己有关的系统权限（不含角色中的系统权限）

```
select * from user_sys_privs;
```

查看和自己有关的对象权限（不含角色中的对象权限）

```
select * from user_tab_privs;
```

角色里包含的角色

```
select * from role_role_privs;
```

角色里包括的系统权限

```
select * from role_sys_privs;
```

角色里包括的对象权限

```
select * from role_tab_privs;
```

查看和自己有关的系统权限(包括角色里的权限）

```
select * from session_privs;
```

从 scott 用户看是个什么情况

```
select * from all_tab_privs where grantee='TIM';
select * from all_tab_privs where table_name='EMP';
```

第十章 Oracle 的事务和锁(PPT-I-283-293)

10.1 什么是事务

必须具备以下四个属性，简称 ACID 属性：
原子性（Atomicity）：事务是一个完整的操作。事务的各步操作是不可分的（原子的）；要么都执行，要么都不执行。
一致性（Consistency）：一个查询的结果必须与数据库在查询开始时的状态保持一致（读不等待写，写不等待读）。
隔离性（Isolation）：对于其他会话来说，未完成的（也就是未提交的）事务必须不可见。
持久性（Durability）：事务一旦提交完成后，数据库就不可以丢失这个事务的结果，数据库通过日志能够保持事务的持久性。

10.2 事务的开始和结束

10.2.1 事务采用隐性的方式，起始于 session 的第一条 DML 语句，

10.2.2 事务结束于：
1）COMMIT（提交）或 ROLLBACK（回滚）
2）DDL 语句被执行（提交）
3）DCL 语句被执行（提交）
4）用户退出 SQLPLUS（正常退出是提交，非正常退出是回滚）
5）服务器故障或系统崩溃（回滚）
6）shutdowm immediate(回滚）

考点：在一个事务里如果某个 DML 语句失败，之前其他任何 DML 语句将保持完好，而且不会提交！

10.3 Oracle 的事务保存点功能

savepoint 命令允许在事务进行中设置一个标记（保存点），这个标记可以控制 rollback 的效果，即在一个事务中回滚掉最近的部分 dml 语句，保留下保存点之前的的 dml 语句，并使事务本身继续执行(考点)。也就是说回滚到保存点这个动作并不使事务结束。

SAVEPOINT 实验
```
savepoint sp1;
delete from emp1 where empno=7900;
savepoint sp2;
update emp1 set ename='timran' where empno=7788;
select * from emp1;
rollback to sp2;
select * from emp1;
rollback to sp1;
```

//rollback to XXX 不会使事务结束。

10.4 SCN 的概念

SCN 全称是 System Change Number
它是一个不断增长的整数，相当于 Oracle 内部的一个时钟，只要数据库一有变更，这个 SCN 就会+1，Oracle 通过 SCN 记录数据库里事务的一致性。SCN 涉及了实例恢复和介质恢复的核心概念，它几乎无处不在：控制文件，数据文件，日志文件都有 SCN，包括 block 上也有 SCN，

实际上，我们所说的保证同一时间点一致性读的概念，其背后是物理层面的 block 读，Oracle 会依据你发出 select 命令，记录下那一刻的 SCN 值，然后以这个 SCN 值去同所读的每个 block 上的 SCN 比较，如果读到的块上的 SCN 大于 select 发出时记录的 SCN，则需要利用 Undo 段，在内存中构造 CR 块(Consistent Read)。

得到当前 SCN 有两个办法：

```
SQL> conn / as sysdba
SQL> select current_scn from v$database;

CURRENT_SCN
-----------
    7222678

SQL> select dbms_flashback.get_system_change_number from dual;

GET_SYSTEM_CHANGE_NUMBER
------------------------
                 7222708
```

10.5 共享锁与排他锁的基本原理：

排他锁，排斥其他排他锁和共享锁。
共享锁，排斥其他排他锁，但不排斥其他共享锁。

因为有事务才有锁的概念。Oracle 数据库锁可以分为以下几大类：

DML 锁（data locks，数据锁），用于保护数据的完整性。
DDL 锁（dictionary locks，数据字典锁），用于保护数据库对象的结构，如表、索引等的结构定义。
SYSTEM 锁（internal locks and latches），保护数据库的内部结构。

考点：
1）当一个用户对某表做 DML 操作时，也会加 DDL 锁，这样在事务未结束前，可防止另一个用户对该表做 DDL 操作。初始化参数 ddl_lock_timeout 可以设定了 DDL 锁的等待时间。时间过后如果事务仍未结束，则显示资源正忙。
2）当一个用户对某表做 DDL 操作时，也会加 DML 锁（EXCLUSIVE 排他锁），这样可以防止另

一个用户对该表做 DML 操作

我们探讨的是 Oracle 的 DML 锁(又叫数据锁)，它包括 TM 和 TX 两种

TM 是面向对象的锁，它表示你锁定了系统中的一个对象，在锁定期间不允许其他人对这个对象做 DDL 操作。TM 锁首先产生，目的就是为了实施 DDL 保护。
TX 是面向事务的锁，表示发起了一个事务，是否有事务产生，这是根据是否使用 UNDO 段作为评判标准的。

比如一个 update 语句，有表级锁（即 TM)和行锁（即 TX 锁)。Oracle 是先申请表级锁 TM（其中的 RX 锁）， 获得后系统再自动申请行锁(TX)， 并将实际锁定的数据行的锁标志置位（即指向该 TX 锁)。

对于 DML 操作

行锁(TX)只有一种
表锁(TM)共有五种，分别是 RS,RX,S,SRX,X。

10.6 五种 TM 表锁的含义：

ROW SHARE 行共享(RS)，允许其他用户同时更新其他行，允许其他用户同时加共享锁，不允许有独占（排他性质）的锁
ROW EXCLUSIVE 行排他(RX)，允许其他用户同时更新其他行，只允许其他用户同时加行共享锁或者行排他锁
SHARE 共享(S)，不允许其他用户同时更新任何行，只允许其他用户同时加共享锁或者行共享锁
SHARE ROW EXCLUSIVE(SRX) 共享行排他，不允许其他用户同时更新其他行，只允许其他用户同时加行共享锁
EXCLUSIVE (X)排他，其他用户禁止更新任何行，禁止其他用户同时加任何排他锁。

sql 语句	加锁模式	许可其他用户的加锁模式
select * from table_name	无	RS,RX,S,SRX,X
insert, update, delete(DML 操作）	RX	RS,RX
select * from table_name for update	RX	RS,RX
lock table table_name in row share mode	RS	RS,RX,S,SRX

lock table table_name in row exclusive mode RX RS,RX

lock table table_name in share mode S RS,S

lock table table_name in share row exclusive mode SRX RS

lock table table_name in exclusive mode X 无

10.7 加锁模式

第一种方式：自动加锁

做 DML 操作时，如 insert，update，delete，以及 select....for update 由 oracle 自动完成加锁

session1 scott: //用 for update 加锁
SQL> select * from dept where deptno>20 for update;

DEPTNO	DNAME	LOC
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

session2 sys: //试探，以防被锁住
SQL>select * from scott.dept for update nowait;
SQL>select * from scott.dept for update wait 5;

session1 scott:
SQL> select * from emp where deptno=30 for update;

session2 sys: 跳过加锁的记录，锁定其他记录。
SQL> select * from scott.emp for update skip locked;

注意：
1）对整个表 for update 是不锁 insert 语句的。
2）wait 5：等 5 秒自动退出。nowait：不等待。skip locked：跳过。都可起到防止自己被挂起的作用。

第二种方式：人工方式加锁，用 lock 命令以显式的方式加锁。

lock table 表名 in exclusive mode.（一般限于后三种表锁）

观察锁的动态视图 v$lock

观察锁的静态视图 dba_locks

```
select * from v$lock;

select * from dba_locks where session_id=149;
```

10.8 死锁和解锁

10.8.1 Oracle 自动侦测死锁，自动解决锁争用。

制作死锁案例：

```
scott:

SQL> select * from a;

      ID_A
----------
         1
         2

brain::

SQL> select * from b;

      ID_B
----------
       100
       200

ORA-00060: deadlock detected while waiting for resource

scott:   //改自己,不提交

update table a set id=11 where id=1;
brain:   //改自己,不提交
update table b set id=1100 where id=100;
scott:   //改对方,被锁住
update table brain.b id=1000 where id=100;
brain:   //改对方,造成死锁
update table brain.b id=1000 where id=100;
```

10.8.2 管理员如何解鎖

可以根据以下方法准确定位要 kill session 的 sid 号和 serial#号，

```
SQL> select * from v$lock where type in ('TX','TM');

ADDR     KADDR            SID TYPE        ID1        ID2       LMODE     REQUEST
CTIME      BLOCK
-------- -------- ---------- ---- ---------- ---------- ---------- ---------- --
-------- ----------
38B66D60 38B66D8C         127 TX        327680       1042          0          6
2985          0
00567BAC 00567BDC         134 TM         71090          0          3          0
2996          0
00567BAC 00567BDC         127 TM         71090          0          3          0
2985          0
37960894 379608D4         134 TX        327680       1042          6          0
2996          1
```

```
SQL> select a.sid,a.serial#,b.sql_text from v$session a,v$sql b where
a.prev_sql_id=b.sql_id and a.sid=127;

       SID    SERIAL# SQL_TEXT
---------- ---------- ------------------------------------------------------------
----------------------
       127       2449 update emp1 set sal=8000 where empno=7788
```

```
SQL> select sid,serial#,blocking_session,username,event from v$session where
blocking_session_status='VALID';

       SID    SERIAL# BLOCKING_SESSION USERNAME                       EVENT
---------- ---------- ---------------- ------------------------------ ----------
--------------------------------
       127       2449              134 SCOTT                          enq: TX -
row lock contention
```

也可以根据 v$lock 视图的 block 和 request 确定 session 阻塞关系,确定无误后再杀掉这个 session

```
SQL>ALTER SYSTEM KILL SESSION '127,2449';
```

更详细的信息，可以从多个视图得出，相关的视图有：v$session,v$process,vsql,vlocked,v$sqlarea 等等

阻塞(排队）从 OEM 里看的更清楚 OEM-->Performance-->Additional Monitoring Links-->Blocking Sessions(或 Instance Locks)

第十一章，索引

11.1 索引结构及特点

11.1.1 B 树索引结构（图），介绍根节点，分支节点，叶子节点，以及表行，rowid，键值，双向链等概念。

考点：
1）叶块之间使用双向链连接，
2）删除表行时索引叶块也会更新，但只是逻辑更改，并不做物理的删除叶块。
3）索引叶块中不保存表行键值的 null 信息。

11.1.2 位图索引结构：

位图索引适用于离散度较低的列，它的叶块中存放 key, start rowid-end rowid,并应用一个函数把位图中相应 key 值置 1，位图索引在逻辑 or 时效率最高。

```
SQL>create bitmap index job_bitmap on emp1(job);

值/行        1  2  3  4  5  6  7  8  9  10 11 12 13 14
--------------------------------------------------------------
ANALYST      0  0  0  0  0  0  0  1  0  0  0  0  1  0
CLERK           1  0  0  0  0  0  0  0  0  0  1  1  0  1
MANAGER         0  0  0  1  0  1  1  0  0  0  0  0  0  0
PRESIDENT       0  0  0  0  0  0  0  0  1  0  0  0  0  0
SALESMAN     0  1  1  0  1  0  0  0  0  0  0  0  0  0

SQL>select count(*) from emp1 where job = 'CLERK' or job = 'MANAGER';

值/行        1  2  3  4  5  6  7  8  9  10 11 12 13 14
--------------------------------------------------------------
CLERK           1  0  0  0  0  0  0  0  0  0  1  1  0  1
MANAGER         0  0  0  1  0  1  1  0  0  0  0  0  0  0
--------------------------------------------------------------
or 的结果         1  0  0  1  0  1  1  0  0  0  1  1  0  1
```

以上操作使用 autotrace 可以看到优化器使用了 bitmap，

B 树索引要比位图索引应用更广泛，下面我们重点关注 B 树索引。

索引是与表相关的一个可选结构，在逻辑上和物理上都独立于表的数据，索引能优化查询，不能优化 DML 操作，Oracle 自动维护索引，频繁的 DML 操作反而会引起大量的索引维护。

如果 SQL 语句仅访问被索引的列，那么数据库只需从索引中读取数据，而不用读取表，如果该语句同时还要访问除索引列之外的列，那么，数据库会使用 rowid 来查找表中的行，通常，为检索表数据，数据库以交替方式先读取索引块，然后读取相应的表块。

11.2 B 树索引和位图索引的适用环境

B 树适合情况	位图适合情况
大表，返回行数<5%	同左
经常使用 where 子句查询的列	同左
离散度高的列	离散度低的列
更新键值代价低	更近键值代价高
逻辑 AND 效率高	逻辑 OR 效率高
用于 OLTP	用于 OLAP

11.3 索引的类型与选项：

1）B 树索引，2）位图索引

常用的 B 树索引类型：

唯一或非唯一索引 (Unique or non_unique)：唯一索引指键值不重复。

```
SQL> create unique index empno_idx on emp1(empno);
```

或

```
SQL> create index empno_idx on emp1(empno);
```

组合索引(Composite)：绑定了两个或更多列的索引。

```
SQL> create index job_deptno_idx on emp1(job,deptno);
```

反向键索引(Reverse)：将字节倒置后组织键值。当使用序列产生主键索引时，可以防止叶节点出现热块现象（考点）。

```
SQL> create index mgr_idx on emp1(mgr) reverse;
```

函数索引(Function base)：以索引列值的函数值为键值去组织索引

```
SQL> create index fun_idx on emp1(lower(ename));
```

压缩（Compress)：重复键值只存储一次，就是说重复的键值在叶块中就存一次，后跟所有与之匹配的 rowid 字符串。

```
SQL> create index comp_idx on emp1(sal) compress;
```

升序或降序（Ascending or descending)：叶节点中的键值排列默认是升序的。
SQL> create index deptno_job_idx on emp1(deptno desc, job asc);

可以更改索引属性：
alter index xxx ....

索引相关的数据字典

USER_INDEXES //索引主要信息
USER_IND_CULUMNS //索引列的信息

11.4 优化器使用索引的扫描方式

Oracle 的执行计划常见的四种索引扫描方式：

1）索引唯一扫描(index unique scan)
通过唯一索引查找一个数值返回单个 ROWID。对于唯一组合索引，要在 where 的谓词“=”后包含所有列的“布尔与”。

2）索引范围扫描(index range scan)
在非唯一索引上，可能返回多行数据，所以在非唯一索引上都使用索引范围扫描。
使用 index rang scan 的 3 种情况：
(a) 在唯一索引列上使用了 range 操作符(> < <> >= <= between)
(b) 在唯一组合索引上，对组合索引使用部分列进行查询(含引导列)，导致查询出多行
(c) 对非唯一索引列上进行的任何查询。不含‘布尔或’

3）索引全扫描(index full scan)
对整个 index 进行扫描，并且顺序的读取其中数据。
全 Oracle 索引扫描只在 CBO 模式下才有效。 CBO 根据统计数值得知进行全 Oracle 索引扫描比进行全表扫描更有效时，才进行全 Oracle 索引扫描，而且此时查询出的数据都必须从索引中可以直接得到。

4）索引快速扫描(index fast full scan)
扫描索引中的所有的数据块，与 index full scan 很类似，但是一个显著的区别是 full scan 是根据叶子块的双向列表顺序读取，读取的块是有顺序的，也是经过排序的，所以返回的列表也是排序的。而 fast full scan 在读取叶子块时的顺序完全由物理存储位置决定，并采取多块读，每次读取 DB_FILE_MULTIBLOCK_READ_COUNT 个块。

//分析器是根据要访问的数据量和索引的聚簇因子等属性判断使用 RANGE SCAN 或 INDEX FULL SCAN

聚簇因子(CLUSTERING_FACTOR)： 堆表的表行物理的存储在数据块是无序的，这与插入一行记录首选空闲块的策略有关，而索引的键值又是有序的，当这两者差异越大，聚簇因子的值

就越高。

举例这四种索引扫描方式：

如果你的 scott 不能使用 autotrace，做一下几步。

```
SQL> conn / as sysdba
SQL> @$ORACLE_HOME/rdbms/admin/utlxplan
SQL> @$ORACLE_HOME/sqlplus/admin/plustrce
SQL> grant plustrace to scott,hr;
SQL>
```

试试 scott 下能否使用 autotrace

```
SQL> create table emp1 as select * from emp;

SQL> set autotrace traceonly explain;

SQL> select empno from emp1;

执行计划
----------------------------------------------------------
Plan hash value: 2226897347

--------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    14 |   182 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| EMP1 |    14 |   182 |     3   (0)| 00:00:01 |
--------------------------------------------------------------------------------
```

例：索引唯一扫描(index unique scan)

```
SQL> create unique index emp1_idx on emp1(empno);

索引已创建。

SQL> select empno from emp1 where empno=7788;

执行计划
----------------------------------------------------------
Plan hash value: 1995401140

--------------------------------------------------------------------------------
| Id  | Operation          | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
```

```
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |          |     1 |    13 |     0   (0)| 00:00:01 |
|*  1 |  INDEX UNIQUE SCAN| EMP1_IDX |     1 |    13 |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

SQL> drop index emp1_idx;

例：索引范围扫描(index range scan)

SQL> create index emp1_idx on emp1(empno);

SQL> select empno from emp1 where empno=7788;

执行计划

```
----------------------------------------------------------
Plan hash value: 253836959

-------------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |     1 |    13 |     1   (0)| 00:00:01 |
|*  1 |  INDEX RANGE SCAN| EMP1_IDX |     1 |    13 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

3）索引全扫描(index full scan)

当你要查询出的数据全部可以从索引中直接得到，也就是说仅读索引块而不需要读表块，这时会选择 index (fast) full scan

SQL> alter table emp1 modify (empno not null); --因索引的叶子块不存空值，使 empno 字段非空。

SQL> select empno from emp1; --数据库仅访问索引本身的数据。而无需访问表。

显示结果同上

例：索引快速扫描(index fast full scan)

SQL> insert into emp1 select * from emp1;

已创建 14 行。

SQL> /

.....

已创建 14336 行。

SQL> select count(empno) from emp1;

执行计划

```
----------------------------------------------------------
Plan hash value: 593157948

--------------------------------------------------------------------------------
--
| Id  | Operation             | Name      | Rows  | Bytes | Cost (%CPU)| Time
|
--------------------------------------------------------------------------------
--
|   0 | SELECT STATEMENT      |           |     1 |     3 |    27   (0)| 00:00:01
|
|   1 |  SORT AGGREGATE       |           |     1 |     3 |            |
|
|   2 |   INDEX FAST FULL SCAN| EMP1_IDX  | 28672 | 86016 |    27   (0)| 00:00:01
|
--------------------------------------------------------------------------------
--
```

对比 index full scan，当行数较多时，会使用 index fast full scan

行数较多时 index fast full scan 比 index full scan 计划成本要低的多，所以 CBO 优化器会优选 index fast full scan

两项再比较一下：

SQL> select empno from emp1;

执行计划

```
----------------------------------------------------------
Plan hash value: 1892048637

--------------------------------------------------------------------------------
-
| Id  | Operation             | Name      | Rows  | Bytes | Cost (%CPU)| Time
|
--------------------------------------------------------------------------------
-
```

```
|   0 | SELECT STATEMENT      |           | 28033 |   355K|    29   (0)| 00:00:01
|
|   1 |  INDEX FAST FULL SCAN| EMP1_IDX | 28033 |   355K|    29   (0)| 00:00:01
|
--------------------------------------------------------------------------------
-
Note
-----
   - dynamic sampling used for this statement
```

可以加一行 hint,强制 oracle 使用 index full scan 的执行计划，得到 cost 是 100.

```
SQL> select /*+ index(emp1 emp1_idx) */ empno from emp1;

执行计划
----------------------------------------------------------
Plan hash value: 4252953140

--------------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT |           | 28033 |   355K|   100   (0)| 00:00:02 |
|   1 |  INDEX FULL SCAN | EMP1_IDX | 28033 |   355K|   100   (0)| 00:00:02 |
--------------------------------------------------------------------------------

Note
-----
   - dynamic sampling used for this statement
```

可以看出：CBO 在满足一定条件时会先选 index fast full scan,而不是 index full scan，因为前者的 cost 是 29，比后者 100 低许多。

CBO 优化器何时决定 INDEX FULL SCAN 与 INDEX FAST FULL SCAN：

共性：当仅从索引表中就可以得到所要的查询结果，省去了第二步扫描表块。
个性：INDEX FAST FULL SCAN 可以使用多块读，多块读由参数 db_file_multiblock_read_count 指定,适用于表行多时 IO 效率更高,而对于索引列上 order by 之类的操作又几乎总是使用 INDEX FULL SCAN。

```
SQL>
```

11.5 索引的碎片问题

由于对基表做 DML 操作，便导致对索引表块的自动更改操作，尤其是基表的 delete 操作会引

起 index 表的 index entries 的逻辑删除，注意，只有当一个索引块中的全部 index entry 都被删除了，这个块才会被收回。如果 update 基表索引列，则索引块会发生 entry delete，再 entry insert，这都些动作都可能产生索引碎片。

```
SQL> create table t (id int);
SQL> create index ind_1 on t(id);
SQL>
begin
  for i in 1..1000000 loop
    insert into t values (i);
        if mod(i, 100)=0 then
        commit;
        end if;
  end loop;
end;
/

SQL> analyze index ind_1 validate structure;
SQL> select name,HEIGHT,PCT_USED,DEL_LF_ROWS/LF_ROWS from index_stats
SQL> delete t where rownum<700000;
SQL> alter index ind_1 rebuild [online] [tablespace name];
```

在 Oracle 文档里并没有清晰的给出索引碎片的量化标准,Oracle 建议通过 Segment Advisor(段顾问）解决表和索引的碎片问题（053 课程会涉及），如果你想自行解决，可以通过查看 index_stats 视图，当以下三种情形之一发生时，说明积累的碎片应该整理了（仅供参考)。

1.HEIGHT >=4
2 PCT_USED< 50%
3 DEL_LF_ROWS/LF_ROWS>0.2

联机重建索引通常比删除后再重建要更实用，Oracle9i 和 10g 一直提供联机索引重建功能--rebuild online,但由于涉及到使用表的排他锁，通常一段时间内其他用户的 DML 操作需要等待。

在 Oracle11g 中有了改进，实现了最终意义上的联机索引重建(rebuild index online)的特性，它重新设计了锁的模式，因此容许 DML 操作不受干扰。

```
SQL> alter index ind_1 coalesce;     //索引融合，比 rebuild 动作轻，可以合并一些块中的 index entries;
```

11.6 索引不可见（invisible),11g 新特性

在 11g 里， Oracle 提供了一个新特性来降低直接删除索引或禁用索引的风险，那就是索引

不可见（Index Invisible)。我们可以在创建索引时指定 invisible 属性或者用 alter 语句来修改索引为 invisible(visible)（考点）

```
SQL> create table test (id int,name char(10));
SQL> create index test_idx on test(id)
SQL> alter index test_idx invisible;
SQL> select index_name,status,VISIBILITY from user_indexes;
```

注意 索引被设定为 invisible 后，视图 user_indexes 中 status 字段仍然是 VALID，实际上就是指该索引对于优化器不可见，而索引的正常更新仍然会由有 Oracle 自动完成的。对比 11g 前的 unusable，（保存索引定义，不删除索引，也不更新索引)。

```
SQL> alter index test_idx unusable;
SQL> select index_name,status VISIBILITY from user_indexes;
```

索引被设定为 unusable 后，视图 user_indexes 中 status 字段是 unusable(考点）

查询索引的两个动态视图：

```
select * from dba_indexes;
select * from dba_ind_columns

SQL>
```

第十二章 约束

12.1 什么是约束

约束是数据库能够实施业务规则以及保证数据遵循实体-关系模型的一种手段。

考点：如果违反约束，将自动回滚出现问题的整个语句，而不是语句中的单个操作，也不是整个事务。

12.2 约束的语法：

列级定义：只能引用一个列，表中可以有多个列级约束。
表级定义：引用一个或多个列，通常用来定义主键。
追加定义：建表后，再通过 alter table 命令追加的约束。

查看约束的两个数据字典视图

```
select * from user_constraints;
select constraint_name,column_name,table_name from user_cons_columns;
```

12.3 五种约束的语法

12.3.1 非空约束

列级定义：

```
create table stud (id number(2) not null, name varchar(4))
```

追加非空约束：
not null 约束比较特殊，一般只是列级定义和表外定义，当使用表外（追加）时，要使用 modify 关键字（考点）

如：alter table emp1 modify ename not null;
或 alter table emp1 modify ename constraint xyz not null;

12.3.2.唯一性约束 （唯一性约束允许列中输入空值）

列级定义
```
create table a1(id number(2) unique,name varchar2(4));
```

表级定义
```
create table a2(id number(2),name varchar2(4),constraint id_uk unique(id));
```

追加定义
```
alter table a2 add CONSTRAINT id_uk UNIQUE (id);
```

考点：唯一加非空约束可以多列。unique 和 not null 之间没有“,”
```
SQL> create table a (id int unique not null, name char(10) unique not null);
```

12.3.3 主键约束

考点：

1）每个表只能建立一个主键约束，primary key=unique key + not null，主键约束可以是一列，也可以是组合多列。
2）主键列上需要索引，如果该列没有索引会自动建立一个 unique index，如果该列上已有索引（非唯一也可以），那么就借用这个索引，由于是借用的索引，当主键约束被删除后，借用的索引不会被删除。同理，多列组合的主键，需要建立多列组合索引，而多列主键的单列上还可以另建单列索引。
3）主键约束和唯一约束不能同时建立在一个列上。

主键约束的六种写法

列级定义

1）create table u1 (id char(10) primary key , name char(20)); -- 主键名字，Oracle 起的
2）create table u2 (id char(10) constraint pk_u2 primary key , name char(20)); -- 主键名字，自己指定

表级定义
3）create table u3 (id char(10) , name char(20) , primary key(id)); -- 主键名字，Oracle 起的
4）create table u4 (id char(10) , name char(20) , CONSTRAINTS pk_u4 primary key (id)); --主键名字，自己指定

追加定义
create table u5 (id char(10) , name char(20));

5）alter table u5 add primary key(id); // desc u5; 会发现 u5 已经自动加上了 not null 属性

6）alter table u5 add CONSTRAINT pk_u5 primary key (id) ; --表外，后来加上主键。

关于主键和索引关联的问题:（这个地方考点较多）

```
SQL>
create table t (id int, name char(10));
insert into t values (1, 'sohu');
insert into t values (2, 'sina');
commit;

SQL> create index t_idx on t(id);
```

下面这两句话是一样的效果，因为缺省情况下 id 列已经有索引 t_id 了，建主键时就会自动用这个索引（考点）。

```
SQL> alter table t add constraint pk_id primary key (id);
SQL> alter table t add constraint pk_id primary key (id) using index t_idx;

SQL> select CONSTRAINT_NAME,TABLE_NAME,INDEX_NAME  from user_constraints;

CONSTRAINT_NAME                TABLE_NAME                     INDEX_NAME
------------------------------ ------------------------------ ------------------
------------
FK_DEPTNO                      EMP
PK_DEPT                        DEPT                           PK_DEPT
PK_EMP                         EMP                            PK_EMP
```

```
PK_ID                          T                              T_IDX
```

SQL> alter table t drop constraint pk_id; //删除了约束，索引还在，本来就是借用的索引。

```
SQL> select index_name from user_indexes;

INDEX_NAME
------------------------------
PK_EMP
PK_DEPT
T_IDX
```

SQL> drop table t purge; //t_idx 是和 t 表关联的，关键字 purge 使表和索引一并永久删除了。

也可以使用 using 子句在建表建、建约束、建索引一条龙下来，当然 primary key 也会自动使用这个索引(考点)。删除该约束，索引还存在。

```
SQL> create table t (id int,name char(10),constraint pk_id primary key(id) using
index
(create index t_idx on t(id)));

SQL> select CONSTRAINT_NAME,TABLE_NAME,INDEX_NAME  from user_constraints;

CONSTRAINT_NAME                TABLE_NAME                     INDEX_NAME
------------------------------ ------------------------------ ------------------
------------
FK_DEPTNO                      EMP
PK_DEPT                        DEPT                           PK_DEPT
PK_EMP                         EMP                            PK_EMP
PK_ID                          T                              T_IDX

SQL> select index_name from user_indexes;

INDEX_NAME
------------------------------
PK_EMP
PK_DEPT
T_IDX
```

12.3.4.外键约束 （引用完整性约束）

作用：是为了和同一个表或其他表的主关键字(或唯一关键字)建立连接关系，外键值必须和

父表中的值匹配或者为空值。

考点：
1）外键约束和 unique 约束都可以有空值。
2）外键需要参考主键约束，但也可以参考唯一键约束。
3）外键和主键一般分别在两个表中，但也可以同处在一个表中。

```
SQL> create table emp1 as select * from emp;
SQL> create table dept1 as select * from dept;
```

列级定义

```
SQL> alter table dept1 add constraint pk_dept1 primary key(deptno);
SQL> create table emp100 (empno int,deptno int references dept1(deptno),deptno2
int);
```

//外键的列级定义有点特殊，使用 references 不使用 foreign key 关键字。

表级定义

```
SQL> create table emp200 (empno int,deptno int,sal int,foreign key(deptno)
references dept1(deptno));
```

追加定义

```
ALTER TABLE emp1 ADD CONSTRAINT fk_emp1 FOREIGN KEY(deptno) REFERENCES
dept1(deptno);
```

关于 ON DELETE CASCADE 关键字

测试：

```
delete from dept1 where deptno=30
*
ERROR at line 1:
ORA-02292: integrity constraint (SCOTT.E_FK) violated - child record found
```

删除外键约束

```
SQL>alter table emp1 drop constraint fk_emp1;
```

使用 ON DELETE CASCADE 关键字重建外键,

```
SQL>ALTER TABLE emp1 ADD CONSTRAINT fk_emp1 FOREIGN KEY(deptno) REFERENCES
dept1(deptno) ON DELETE CASCADE

SQL>select constraint_name,constraint_type,status,delete_rule from
```

user_constraints;

注意：delete_rule 列会显示 CASCADE,否则显示 NO ACTION(考点)

测试：
delete from dept1 where deptno=30

再查看 emp1 表的 deptno 已经没有 30 号部门了，如果再对 dept1 的操作进行 rollback，emp1 的子记录也随之 rollback

ON DELETE CASCADE 要慎用，父表中删除一行数据就可能引发子表中大量数据丢失。

为此，还有 on delete set null 子句，顾名思义是子表不会删除（丢失）记录，而是将外键的值填充 null。

如果 disable dept1 主键约束并使用级联 cascade 关键字，则 emp1 的外键也会 disable, 若再次 enable dept1 主键，则 emp1 外键任然保持 disable.
SQL> alter table dept1 disable constraints pk_dept1 cascade;
SQL> alter table dept1 enable constraints pk_dept1;

SQL>select constraint_name,constraint_type,status,delete_rule from user_constraints;

SQL> drop table dept1 purge;
drop table dept1 purge
*
第 1 行出现错误:
ORA-02449: 表中的唯一/主键被外键引用

SQL> drop table dept1 cascade constraint purge;

表已删除。

注意：这时外键约束也被删除了（考点）

12.3.5.CHECK 约束

列级定义
SQL> create table emp100 (empno int,sal int check (sal>0),comm int);

表级定义
SQL> create table emp200 (empno int,sal int,comm int,check(sal>1000));

追加定义

```
SQL> alter table emp200 add constraint e_no_ck check (empno is not null);
```

验证

```
SQL> insert into emp200 values(null,1,1);
insert into emp200 values(null,1,1)
*
第 1 行出现错误:
ORA-02290: 违反检查约束条件 (SCOTT.E_NO_CK)
```

12.3.6 check 约束中的表达式中不能使用变量日期函数（考点）。

```
SQL> alter table emp1 add constraint chk check(hiredate<sysdate);
alter table emp1 add constraint chk check(hiredate<sysdate)
                                                    *
第 1 行出现错误:
ORA-02436: 日期或系统变量在 CHECK 约束条件中指定错误

SQL> alter table emp1 add constraint chk check(hiredate<to_date('2000-01-01','yyyy-mm-dd'));     //这句是可以的
```

12.3.7 级联约束（考点）

测试表

```
CREATE TABLE test2 (
    pk NUMBER PRIMARY KEY,
    fk NUMBER,
    col1 NUMBER,
    col2 NUMBER,
    CONSTRAINT fk_constraint FOREIGN KEY (fk) REFERENCES test2,
    CONSTRAINT ck1 CHECK (pk > 0 and col1 > 0),
    CONSTRAINT ck2 CHECK (col2 > 0)
    )
/
```

当删除列时，看看会发生什么?

```
ALTER TABLE test2 DROP (col2);  //这句可以执行

ALTER TABLE test2 DROP (fk);    //这句可以执行

ALTER TABLE test2 DROP (pk);    //这句不能执行，在 constraint ck1 中使用了该列，
```

约束级联问题

ALTER TABLE test2 DROP (col1); //这句不能执行，在 constraint ck1 中使用了该列，约束级联问题

如果一定要删除级联约束的列，带上 cascade constraints 才行

ALTER TABLE test2 DROP (pk) cascade constraint; //所有与 pk 列有关的约束统统随该列被删掉。

或(ALTER TABLE test2 DROP (col1) cascade constraints;)

CASCADE CONSTRAINTS 将丢弃在删除列上的唯一键或主键约束。

//注意：constraint ck1 约束只能是表级约束（为什么？）

12.3.8 约束的四种状态

enable validate :无法输入违反约束的行，而且表中所有行都要符合约束
enable novalidate :表中可以存在不合约束的状态,,但对新加入数据必须符合约束条件.
disable novalidate :可以输入任何数据，表中或已存在不符合约束条件的数据.
disable validate :不能对表进行插入/更新/删除等操作,相当于对整个表的 read only 设定.

更改约束状态是一个数据字典更新，将对所有 session 有效。

举例：

1)enable novalidate 这种组态的用法

常用于当在表中输入了一些测试数据后，而上线后并不想去清除这些违规数据，但想从此开始才执行约束。

假设已经建立了一个 emp1 表，也插入了数据，如果有一天想在 empno 上加入 primary key 但是之前有不符合 (not null+unique)约束的，怎样才能既往不咎呢？

create table emp1 as select * from emp;(CTAS 语句使约束并没有考过来)

update emp1 set empno=7788 where empno=7369;(设置一个重号)

alter table emp1 add constraint pk_emp1 primary key (empno);因要检查主键唯一索引，拒绝建立此约束。

任何违反（not null+unique)的 update 或 insert 操作将被拒绝。

alter table emp1 add constraint pk_emp1 primary key (empno) enable novalidate;（这句话也不行，为什么？原因是唯一索引在捣乱）

create index empno_index on emp1(empno);建一个索引，一定要一个普通索引，不能是唯一索引，普通索引不受 unquie 的限制）

alter table emp1 add constraint pk_emp1 primary key (empno) enable novalidate;（这句话可以了)。

从此之后，这个列的 DML 操作还是要符合（not null+unique)。

2) disable validate 组态的特点：

```
SQL> alter table emp1 add constraint pk_emp1 primary key(empno);
SQL> select index_name from user_indexes;

INDEX_NAME
------------------------------
PK_EMP1
PK_EMP
PK_DEPT

SQL> alter table emp1 modify constraint pk_emp1 disable validate;   //主键索引将自动删除（考点）
SQL> select index_name from user_indexes;

INDEX_NAME
------------------------------
PK_EMP
PK_DEPT

SQL> update emp1 set sal=8000;
update emp1 set sal=8000
*
第 1 行出现错误:
ORA-25128: 不能对带有禁用和验证约束条件 (SCOTT.PK_EMP1) 的表进行插入/更新/删除

SQL> alter table emp1 modify constraint pk_emp1 enable validate;

表已更改。

SQL> select index_name from user_indexes;
```

```
INDEX_NAME
------------------------------
PK_EMP1
PK_EMP
PK_DEPT
```

3)将 disable novalidate，enable novalidate 和 enable validate 三种状态组合起来的用法：

这种组合，可以避免因有个别不符合条件的数据而导致大数据量的传输失败。
假设有 a 表是源数据表，其中有空值，b 表是 a 表的归档表，设有非空约束,现要将 a 表数据（远程）大批量的插入到 b 表（本地）。

```
alter table b modify constraint b_nn1 disable novalidate;   //先使 B 表非空约束无效。
insert into b select * from a;                    //大批数据可以无约束插入，空值也插进 B 表里了。
alter table b modify constraint b_nn1 enable novalidate;    //既往不咎，但若新输入数据必须符合要求。
update b set channel='NOT KNOWN'where channel is null;      //将所有空值填充了，新老数据都符合要求了。
alter table b modify constraint b_nn1 enable validate;          //最终是约束使能+验证生效，双管齐下。
```

12.3.9 延迟约束

可延迟（deferrable）可以通过查询 User_Constraints 视图获得当前所有关于约束的系统信息.

查看 user_constraints 中的两个字段

```
Deferrable  //是否为延迟约束 值为:Deferrable 或 Not Deferrable(缺省）.
Deferred    //是否采用延迟 值为:Immediate（缺省）或 Deferred.
```

关于 Deferrable 可延迟，提醒以下几点：

1）约束的默认方式下是：enable/validate 和 Not Defferrable。

2）如果创建约束时没有指定 deferrable 那么无法在后来使约束成为延迟约束（只有通过重建约束时再指定它是延迟约束）

3）一个约束只有被定义成 deferrable，那么这个约束 session 级才可以在 deferred 和 immediate 两种状态间相互转换

例：

```
SQL> alter table emp1 add constraint chk_sal check(sal>500) deferrable;
```

已经将 chk_sal 延迟设为可延迟了，在此基础上可以有两种面向 session 的方案

```
SQL>set constraint chk_sal immediate;      //约束不延迟，插入数据立刻检查约束
SQL>set constraint chk_sal deferred;       //约束延迟，提交时将整个事务一起检查约束
```

也可以在建立约束时一次性指定系统级的延迟约束

```
alter table emp1 add constraint chk_sal check(sal>500) deferrable initially immediate;
```

或

```
alter table emp1 add constraint chk_sal check(sal>500) deferrable initially deferred;
```

考点：

1）使用 set immediate 和 deferred 的切换只影响当前会话，而 initially 状态将应用于所有会话。
2）延迟约束时，一旦有一条 DML 语句违反了约束，整个提交都将失败，全军覆没。

第十三章 视图

13.1 为什么使用视图

1）限制数据的存取：用户只能看到基表的部分信息。方法：赋予用户访问视图对象的权限，而不是表的对象权限。
2）使得复杂的查询变得容易（内联视图）：
3）提供数据的独立性

13.2 简单视图和复杂视图

特性	简单视图	复杂视图
表的个数	一个	一个或多个
含函数	无	有
含组函数	无	有
含 DISTINCT	无	有
DML 操作	可以	不一定

13.2 语法

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW view
  [(alias[, alias]...)]
AS subquery
[WITH CHECK OPTION [CONSTRAINT constraint]]
[WITH READ ONLY];
```

13.2.1 FORCE 作用：可以先建视图，后建基表

```
create force view view1 as select * from test1;
```

13.2.2 WITH CHECK OPTION 作用：对视图 where 子句进行约束。不允许对限定关键字修改。

```
SQL>create view view2 as select * from emp where deptno=10 with check option;
```

13.2.3 WITH READ ONLY 作用：禁止对视图执行 DML 操作

```
SQL>create view view3 as select * from emp where deptno=10 with read only;
```

13.2.4 关于视图注意事项（考点）

1）只要视图中的数据不是来自基表的原始数据，就尽量不要对该数据做 DML 操作。
2）视图的 DML 操作
含有如下情况，则不能删除视图中的数据:
含有聚合函数
含有 GROUP BY 子句
含有 DISTINCT 关键字
含有 ROWNUM 这个伪列

含有如下情况，不能修改该视图中的数据:
上面提到的任何一种情况。
列是由表达式来进行定义的

含有如下情况，不能增加该视图中的数据:
上面提到的任何一种情况。
在基表中包含有 NOT NULL 约束的列，然而该列并没有在视图中出现

13.3 复杂视图的更新，键保留表概念。

```
select * from user_views
```

如果建立了视图 想查看其中的定义,可以访问如下视图 dba_views 中的 text 字段(long 型);

自建一 pl/sql 过程，参照一下。

```
declare
v_text  dba_views.text%type;
v_name  dba_views.view_name%type;
begin
select text, view_name into v_text,v_name FROM dba_views WHERE view_name='V1';
dbms_output.put_line(v_name||' define is :'||v_text);
end;
/
```

第十四章 同义词

从字面上理解就是别名的意思，和视图的功能类似。就是一种映射关系。

14.1 私有同义词;

一般是普通用户自己建立的同义词，创建者需要 create synonym 权限。

```
sys:
SQL> grant create synonym to scott;     //sys 授权给 scott 建立私有同义词权限
scott:
SQL> create synonym abc for emp;        //scott 建立了一个私有同义词
SQL> select * from abc              //scott 可以使用这个私有同义词了

如何让 tim 用户也使用这个私有同义词
SQL> grant select on scott.emp to tim;      //把访问基表的对象权限给 tim
tim:
SQL> select * from scott.abc;           //tim 使用同义词时加模式名前缀，OK。
```

14.2 公有同义词;

一般是由 DBA 创建，使所有用户都可使用, 创建者需要 create public synonym 权限。

```
create public synonym syn2 for ...
```

（新用户要访问 public synonym（代表 emp 表）, 需要有访问基表的权限。

select * from tab; (tab 是什么？)； 用下面的查看语句:

```
select * from all_objects where object_name='TAB';
select * from all_synonyms where synonym_name='TAB';
```

查看同义词的视图：dba_synonyms
删除私有同义词：drop synonym 同义词名
删除公有同义词：drop public synonym 同义词名

SQL>select * from dba_synonyms where synonym_name='SYN1';

14.3 关于同义词的几个要点：

1）私有同义词是模式对象，一般在自己的模式中使用，如其他模式使用则必须用模式名前缀限定。

2）公有同义词不是模式对象，不能用模式名做前缀。（考点）

3）私有和公有同义词同名时，如果指向不同的对象，私有同义词优先。

4）引用的同义词的对象（表或视图）被删除了，同义词仍然存在，这同视图类似，重新创建该对象名，下次访问同义词时自动编译。（考点）

第十五章 序列

15.1 序列是生成唯一整数值的结构，它的典型用途是用于主键值。

结合真题演示
伪列 nextval, currval 用法

```
CREATE SEQUENCE dept_deptno
INCREMENT BY 10
START WITH 50
MAXVALUE 100
CYCLE
NOCACHE
```

第一次要引用一下 nextval 伪列
select dept_deptno.nextval from dual;
以后就有 currval 伪列值了。
select dept_deptno.nextval from dual;

15.2 几点说明：

1）最简单的建立序列只需要 create sequence 序列名就可以，注意缺省值，起始是 1，步长也是 1。
2) 如果启用 cache，缺省只有 20 个号，经验表明这个数量会不够，可以设置多一些，根据需要 10000 个也可以。

3) cycle 其实没有什么意义，因为它使序列发出重复值，这对于基于序列是主键值的用法是个问题。
4) 创建序列后可以使用 alter 命令再进行修改。alter 命令和 create 命令基本相同，只有一点区别：alter 命令不能设置起始值。如果要重启该序列，唯一的办法是删除并重新创建它。
5) 循环后初始是从 1 开始的， 不管原来的值是如何设的(考点)。

第十六章 外部表

外部表：数据存储在 OS 上，元数据（metadata)存储在数据库中，特点：只读。

16.1 读取外部表的方法

OS 下建目录，
$mkdir -p /u01/oradata/timran11g/aaa_dir

建立 txt 外部表文件，并放到 aaa_dir 下
假如有如下的数据文件:

t1.txt 文件内容：

7566,JONES,2975 ,20
7698,BLAKE,2850 ,30
7788,SCOTT,3000 ,20
7839,KING ,9000 ,10
7902,FORD ,3000 ,20

2)，创建目录,并用 DBA 进行授权;
sys:
sql>create directory test_dir as '/u01/oradata/timran11g/aaa_dir';
sql>grant read,write on directory test_dir to scott;

3).使用被授权的用户 scott 创建外部表:

```
create table test_table
(empno int,
ename char(20),
sal int,
deptno int)
ORGANIZATION EXTERNAL
(
TYPE ORACLE_LOADER
DEFAULT DIRECTORY test_dir
```

```
ACCESS Parameters
(
RECORDS DELIMITED BY NEWLINE
badfile 'bad_dev.txt'
LOGFILE 'log_dev.txt'
FIELDS TERMINATED BY ','
MISSING FIELD VALUES ARE NULL
(empno,ename,sal,deptno)
)
LOCATION('t1.txt')
)
;
```

4),进行 SELECT 操作看是否正确;

```
SQL>select * from test_table
SQL>select * from test_table where deptno=10;
SQL>create table abc as select * from test_table where deptno=20;
SQL>create view v2 as select * from test_table where deptno=20;
```

查看外部表的两个字典视图

```
DBA_EXTERNAL_TABLES;
DBA_EXTERNAL_LOCATIONS;
```

```
SQL>SELECT OWNER,TABLE_NAME,DEFAULT_DIRECTORY_NAME,ACCESS_PARAMETERS FROM
DBA_EXTERNAL_TABLES;
```

考点：

1）外部表中 DEFAULT DIRECTORY 子句是强制的，不能省略的。

2）外部表不能做 DML 操作，不能建立索引、约束、LOB 大对象，但可以使用外部表建立表（CTAS）或视图（CVAS）以及同义词。

3)有两种不同类型的驱动，ORACLE_LOADER and ORACLE_DATAPUMP，前者引用的是 SQL*Loader 特性。

第二册部分

第十七章 insert 语句总结

17.1 第一类，insert 语句：一次插入一行

1）SQL> create table a (id int,name char(10) default 'aaa'); //name 列指定了 default 值

2）SQL> insert into a values(1,'abc'); //表 a 后没有所选列，values 必须指定所有字段的值。

3）SQL> insert into a values(2,default); //同上，name 字段用 default 占位。

4）SQL> insert into a values(3,null); //表 a 后没有所选列，name 字段用 null 占位。

5）SQL> insert into a (id) values(4); //表 a 后有选择字段，未选定的字段如果指定了 default,则以 default 的值代替 null

6）SQL> insert into (select id from a) values (5); //这种形式本质同上，只不过表 a 的形式以结果集代之。

7)SQL> insert into a values(6,(select dname from dept where deptno=10)); //values 里的某列使用了 subquery 引用另一个表的数据。

注意：

1）insert 语句会有约束的问题，不符合约束条件的 insert 不能成功。
2）default 不但可以用于 insert 语句，也可以用于 update 语句（考点）
3）values 后面不可以跟多列子查询。

```
SQL> insert into a values(select deptno,dname from dept where deptno=10);
insert into a values(select deptno,dname from dept where deptno=10)
                     *
第 1 行出现错误:
ORA-00936: 缺失表达式
```

更正一下：

```
SQL> insert into a values((select deptno from dept where deptno=10), (select dname
from dept where deptno=10));

已创建 1 行。

SQL> select * from a;

        ID NAME
---------- ----------
         1 abc
         2 aaa
         3
         4 aaa
         5 aaa
         6 ACCOUNTING
```

```
SQL> commit;
```

7）insert WITH CHECK OPTION 的用法

```
SQL> insert into (select id from a where id<100  WITH CHECK OPTION) values (20);

SQL> select * from a;

        ID NAME
---------- ----------
         1 abc
         2 aaa
         3
         4 aaa
         5 aaa
         6 ACCOUNTING
        20 aaa

SQL> rollback;

SQL> insert into (select id from a where id<100  WITH CHECK OPTION) values (101);
insert into (select id from a where id<100  WITH CHECK OPTION) values (101)
                             *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

看看这句话的另一种情况：

```
SQL> insert into (select name from a where id<100  WITH CHECK OPTION) values
('NBA');
insert into (select name from a where id<100  WITH CHECK OPTION) values ('NBA')
                               *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

上例是想说明如果插入的列有不在 subquery 作为检查的 where 条件里，那么也会不允许插入（考点）。

```
SQL> insert into (select id,name from a where id<100  WITH CHECK OPTION) values
(10,'tim');

SQL> insert into (select name from a where id<100) values ('NBA'); //不加 WITH
CHECK OPTION 则在插入时不会检查。
```

总结：

这样的语法看起来很特殊，其实 select 子句是不会真正执行的，它只是规定了 insert 语句对某些字段的约束形式而已。即如果不满足 subquery 里的 where 条件的话，就不允许插入。关于 DML 语句中嵌入 subquery 的用法，要注意有些指定是强制的。

比如这一句

SQL> update (select empno,job,sal from emp) set sal=8000 where job=(select job from emp where empno=7788);

update 后接 subquery，用于限制修改列。这里 sal 列和 job 列必须包含在 subquery 内，否则 set sal=8000 和 where job=()就无法识别了。

下面的例子也类似，where 中限定 sal 和 deptno, 那么在 subquery 中也必须声明一下。

SQL> delete (select empno,sal,deptno from emp) where sal>3000 and deptno=10;

17.2 第二类，insert 一次插入多行 语法上去掉了 values 选项。

SQL> create table b as select * from a where 1>2; //建立一个空表 b。结构来自 a 表， where 1>2 使没有符合的记录被筛选出来.

SQL> insert into b select * from a where name='aaa'; //插入的是结果集，注意没有 values 选项。

SQL> select * from b;

```
        ID NAME
---------- ----------
         2 aaa
         4 aaa
         5 aaa
```

SQL> insert into b(id) select id from a where id in(1,3); //使用子查询（结果集）插入，对位， 注意 b 表没有 default。

SQL> select * from b;

```
        ID NAME
---------- ----------
         2 aaa
         4 aaa
```

```
         5 aaa
         1
         3
```

17.3 第三类，Multitable insert 一条 INSERT 语句可以完成向多张表的插入任务。

insert all 与 insert first

1.创建表 T 并初始化测试数据，此表作为数据源。

```
create table t (x number(10), y varchar2(10));
insert into t values (1,'a');
insert into t values (2,'b');
insert into t values (3,'c');
insert into t values (4,'d');
insert into t values (5,'e');
insert into t values (6,'f');
commit;
```

2.查看表 T 的数据

```
SQL>select * from t;

X          Y
---------- ----------
1          a
2          b
3          c
4          d
5          e
6          f

6 rows selected.
```

3.创建表 T1 和 T2，作为我们要插入的目标表。

```
SQL>create table t1 as select * from t where 0=1;
Table created.
SQL>create table t2 as select * from t where 0=1;
Table created.
```

17.3.1 第一种多表插入方法 INSERT ALL

unconditional insert all（无条件 insert all）

1）完成 INSERT ALL 插入

```
SQL>insert all into t1 into t2 select * from t;
12 rows created.
```

这里之所以显示插入了 12 条数据，实际上表示在 T1 表中插入了 6 条，T2 表插入了 6 条，一共是 12 条数据。

2）验证 T1 表中被插入的数据。

```
SQL>select * from t1;
X Y
---------- ----------
1 a
2 b
3 c
4 d
5 e
6 f
6 rows selected.
```

3）验证 T2 表中被插入的数据。

```
SQL>select * from t2;
X Y
---------- ----------
1 a
2 b
3 c
4 d
5 e
6 f

6 rows selected.
```

OK，完成 INSERT ALL 命令的使命。

conditional insert all（有条件 insert all）

```
SQL> insert all when x>=3 then into t1 when x>=2 then into t2 select * from t;
```

已创建 9 行。

```
SQL> select * from t1;

         X Y
---------- ----------
```

```
         3 c
         4 d
         5 e
         6 f

SQL> select * from t2;

         X Y
---------- ----------
         2 b
         3 c
         4 d
         5 e
         6 f
```

17.3.2 第二种多表插入方法 INSERT FIRST

conditional insert first(有条件 insert first)

1）清空表 T1 和 T2

```
SQL> truncate table t1;
SQL> truncate table t2;
```

2）完成 INSERT FIRST 插入

```
SQL>insert first when x>=5 then into t1 when x>=2 then into t2 select * from t;
5 rows created.
```

处理逻辑是这样的，首先检索 T 表查找 X 列值大于等于 5 的数据（这里是“5，e”和“6，f”）插入到 T1 表，然后将前一个查询中出现的数据排除后再查找 T 表，找到 X 列值大于等于 2 的数据再插入到 T2 表（这里是“2，b”、“3，c”和“4，d”）。注意 INSERT FIRST 的真正目的是将同样的数据只插入一次。

3）验证 T1 表中被插入的数据。

```
SQL>select * from t1;
X          Y
---------- ----------
5          e
6          f
```

4）验证 T2 表中被插入的数据。

```
SQL>select * from t2;
X          Y
---------- ----------
```

```
2           b
3           c
4           d
```

5)为真实的反映“数据只插入一次”的目的，我们把条件颠倒后再插入一次。

```
SQL>delete from t1;
SQL>delete from t2;

SQL> insert first when x>=2 then into t1 when x>=5 then into t2 select * from t;
5 rows created.

SQL>select * from t1;
X           Y
---------- ----------
2           b
3           c
4           d
5           e
6           f

SQL>select * from t2;
no rows selected
```

OK，目的达到，可见满足第二个条件的数据已经包含在第一个条件里，所以不会有数据插入到第二张表。

同样的插入条件，我们把“INSERT FIRST”换成“INSERT ALL”，对比一下结果。

```
SQL>delete from t1;
5 rows deleted.
SQL>delete from t2;
0 rows deleted.

SQL>insert all when x>=2 then into t1 when x>=5 then into t2 select * from t;
//conditional insert all
7 rows created.

SQL>select * from t1;
X           Y
---------- ----------
2           b
3           c
4           d
5           e
```

```
6          f

SQL>select * from t2;
X          Y
---------- ----------
5          e
6          f
```

17.3.3 第三种，旋转 Insert (pivoting insert)

Pivoting INSERT 有行变列的功能

```
create table sales_source_data (
employee_id number(6),
week_id number(2),
sales_mon number(8,2),
sales_tue number(8,2),
sales_wed number(8,2),
sales_thur number(8,2),
sales_fri number(8,2)
);

insert into sales_source_data values (176,6,2000,3000,4000,5000,6000);

create table sales_info (
employee_id number(6),
week number(2),
sales number(8,2)
);
```

看上面的表结构,现在将要 sales_source_data 表中的数据转换到 sales_info 表中,这种情况就需要使用旋转 Insert

示例如下:

```
insert all
into sales_info values(employee_id,week_id,sales_mon)
into sales_info values(employee_id,week_id,sales_tue)
into sales_info values(employee_id,week_id,sales_wed)
into sales_info values(employee_id,week_id,sales_thur)
into sales_info values(employee_id,week_id,sales_fri)
select employee_id,week_id,sales_mon,sales_tue,
sales_wed,sales_thur,sales_fri
from sales_source_data;
```

```
SQL> select * from sales_info;

EMPLOYEE_ID       WEEK      SALES
----------- ---------- ----------
        176          6       2000
        176          6       3000
        176          6       4000
        176          6       5000
        176          6       6000
```

从该例子可以看出,所谓旋转 Insert 是无条件 insert all 的一种特殊应用, 将一个表中的行转换成另一个表中的列，这种应用被 oracle 官方,赋予了一个 pivoting insert 的名称,即旋转 insert.

第十八章 DML 和 DDL 语句的其他用法

18.1 DML 语句-MERGE

把数据从一个表复制到另一个表，插入新数据或替换掉老数据是每一个 ORACLE DBA 都会经常碰到的问题。

Oracle 10g 中 MERGE 有如下一些改进：

1、UPDATE 或 INSERT 子句是可选的
2、UPDATE 和 INSERT 子句可以加 WHERE 子句
3、ON 条件中使用常量过滤谓词来 insert 所有的行到目标表中,不需要连接源表和目标表
4、UPDATE 子句后面可以跟 DELETE 子句来去除一些不需要的行

首先创建示例表：

```
create table PRODUCTS
    (
    PRODUCT_ID INTEGER,
    PRODUCT_NAME VARCHAR2(30),
    CATEGORY VARCHAR2(30)
    );

insert into PRODUCTS values (1501, 'VIVITAR 35MM', 'ELECTRNCS');
insert into PRODUCTS values (1502, 'OLYMPUS IS50', 'ELECTRNCS');
insert into PRODUCTS values (1600, 'PLAY GYM', 'TOYS');
insert into PRODUCTS values (1601, 'LAMAZE', 'TOYS');
insert into PRODUCTS values (1666, 'HARRY POTTER', 'DVD');
commit;
```

```
create table NEWPRODUCTS
    (
    PRODUCT_ID INTEGER,
    PRODUCT_NAME VARCHAR2(30),
    CATEGORY VARCHAR2(30)
    );

insert into NEWPRODUCTS values (1502, 'OLYMPUS CAMERA', 'ELECTRNCS');
insert into NEWPRODUCTS values (1601, 'LAMAZE', 'TOYS');
insert into NEWPRODUCTS values (1666, 'HARRY POTTER', 'TOYS');
insert into NEWPRODUCTS values (1700, 'WAIT INTERFACE', 'BOOKS');
commit;

SQL>select * from products;

PRODUCT_ID PRODUCT_NAME                    CATEGORY
---------- ------------------------------ ------------------------------
      1501 VIVITAR 35MM                   ELECTRNCS
      1502 OLYMPUS IS50                   ELECTRNCS
      1600 PLAY GYM                       TOYS
      1601 LAMAZE                         TOYS
      1666 HARRY POTTER                   DVD

SQL> select * from newproducts;

PRODUCT_ID PRODUCT_NAME                    CATEGORY
---------- ------------------------------ ------------------------------
      1502 OLYMPUS CAMERA                 ELECTRNCS
      1601 LAMAZE                         TOYS
      1666 HARRY POTTER                   TOYS
      1700 WAIT INTERFACE                 BOOKS
```

下面例子我们从表 NEWPRODUCTS 中合并行到表 PRODUCTS 中， 但删除 category 为 ELECTRNCS 的行.

```
SQL>MERGE INTO products p
USING newproducts np
ON (p.product_id = np.product_id)
WHEN MATCHED THEN
UPDATE
SET p.product_name = np.product_name,p.category = np.category
DELETE WHERE (p.category = 'ELECTRNCS')
```

```
WHEN NOT MATCHED THEN
INSERT
VALUES (np.product_id, np.product_name, np.category);

4 rows merged.

SQL>select * from products;

PRODUCT_ID PRODUCT_NAME                    CATEGORY
---------- ------------------------------ ------------------------------
      1501 VIVITAR 35MM                    ELECTRNCS
      1600 PLAY GYM                        TOYS
      1601 LAMAZE                          TOYS
      1666 HARRY POTTER                    TOYS
      1700 WAIT INTERFACE                  BOOKS
```

为什么 1502 不在了，但 1501 还在？ 因为 1502 是 matched,先被 update,然后被 delete, 而 1501 是 not matched.

注意几点：
1）例子里有 update,delete 和 insert。它们是否操作是取决于 on 子句的，首先两个表如果符合 on 条件就是匹配，不符合就是不匹配。
2）匹配了就更新，不匹配则插入。这是最初 9i 的原则，10g 后加入了 delete 语句，这个语句必须在匹配条件下出现。它是一种补充。
3）你必须对操作的表有对象权限。
4）ON 子句里的字段不能被 update 子句更新（考点）

18.2 with 语句

我们可以使用一个关键字 WITH， 给一个子查询块（subquery block）起一个别名. 然后在后面的查询中引用这个子查询块的别名。

好处：
1、使用 with 语句，可以避免在 select 语句中重复书写相同的语句块。
2、with 语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。
3、使用 with 语句可以避免重复解析，提高查询效率。

举例：这个 with 语句完成三个动作

建立一个 dept_costs，保存每个部门的工资总和，
建立一个 avg_cost，根据 dept_costs 求出所有部门总工资的平均值，
最后显示出部门总工资值小于部门总工资平均值的那些部门的信息(dname)。

```
WITH
```

```
dept_costs AS (
SELECT d.dname, SUM(e.sal) AS dept_total
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY d.dname ),

avg_cost AS (
SELECT SUM(dept_total)/COUNT(*) AS dept_avg FROM dept_costs )
SELECT * FROM dept_costs
WHERE dept_total <
(SELECT dept_avg FROM avg_cost)
ORDER BY dname
/


DNAME          DEPT_TOTAL
-------------- ----------
ACCOUNTING           8750
SALES                9400
```

可以分三个部分来看:
第一 AS 建立 dept_costs，保存每个部门的工资总和。
第二个 AS 建立 avg_cost，根据第一个 AS dept_costs 求出所有部门总工资的平均值（两个 with 子程序用逗号分开，第二个使用了第一个别名）。
最后是查询主体，SELECT * FROM... 调用了前两个 with 下的别名（子查询），显示部门总工资值小于部门总工资平均值的那些部门的信息。

考点：
1）with 语句中只能有 select 子句，没有 DML 子句（注意和 merge 的区别）。
2）一般将主查询放在最后描述，因为查询主体中要引用的 with 别名必须在之前定义过。

18.3 DDL 操作及名称空间

18.3.1 模式与对象名称空间的关系

模式(Schema)是一种逻辑结构，它对应于用户，每建一个用户就有一套模式与之对应。
我们通常说对象的唯一标识符是前缀为模式名加上对象名称，如 scott.emp。
同一模式下的同类对象是不可以重名的。比如在 scott 模式里，表 emp 是唯一的，但在不同模式下可以重名。

名称空间定义了一组对象类型，同一个名称空间里的不同对象不能同名，而不同的名称空间中的不同对象可以共享相同的名称。

1）表，视图，序列，同义词是不同类型的对象，但它们属于同一名称空间，因此在同一模式下也是不可以重名的，比如 scott 下不可以让一个表名和一个视图名同名。
2）索引、约束有自己的名称空间，所以在 scott 模式下，可以有表 A,索引 A 和约束 A 共存。

考点：在同一个模式中，表、视图和同义词不能同名。

18.3.2 使用子查询创建表

关于 CTAS 结构：create table as subquery;

如：create table 表 2 as select * from 表 1;是最简洁的复制表的命令，但是要注意几点：

1）表 1 的索引不会被复制给表 2，
2）表 1 中的约束只有 not null 约束能够带到表 2 里来（考点），
3）表 1 中如果有 default 也不会被复制到表 2。
4）由于需要建立表 2，则表 2 列的命名必须符合规范。通常对表 2 或表 1 有选择的指定别名。（考点）
5) 表 2 可以有 default 选项。（考点）

18.3.3 改变表结构的 DDL 操作

1）在数据库打开的情况下，可以使用 DDL 语句更改表列，如增加（add），删除(drop),修改（midify),更名（rename)等。

drop 和 rename 语法上要加 column 关键字。

原理：清除掉字典信息（撤消存储空间），不可恢复。

2)当想要 add 一列,并约束该列为 not null 时,如果该表已经有数据了,加的列本身是 null,则与 not null 约束矛盾，报错 。

```
SQL> select * from a;

        ID NAME
---------- ----------
         1 a
         2 b

SQL> alter table a add C number(5) not null;
alter table a add C number(5) not null
            *
第 1 行出现错误:
```

ORA-01758: 要添加必需的 (NOT NULL) 列, 则表必须为空

SQL> alter table a add C number(5) default 0 not null;

修改成功了，可以看到 C 列全是 0，这样才能使 C 列的约束为 not null （考点）

3）要删除某一个表格上的某个字段，但是由于这个表格拥有非常大量的资料，如果你在尖峰时间直接执行 ALTER TABLE ABC DROP （COLUMN)；可能会收到 ORA-01562 - failed to extend rollback segment number string，这是因为在这个删除字段的过程中你可能会消耗光整个 RBS。

Oracle 推荐：使用 SET UNUSED 选项标记一列（或多列），使该列不可用。
然后，当业务量下降后再使用 DROP UNUSED column 选项删除被被标记为不可用的列。SET UNUSED COLUMNS 用于 drop 多列时效率更高，

SET UNUSED COLUMNS 方法系统开销比较小，速度较快，但效果等同于直接 drop column。就是说这两种方法都不可逆，无法再还原该字段及其内容了。

语法：
ALTER TABLE table SET UNUSED [column] (COLlist 多个)
ALTER TABLE table DROP UNUSED [COLUMN];

考点：
1）如果 set unused 某列，该列上有索引，约束，并定义了视图，引用过序列，结果如何，索引和约束自动删除，序列无关，视图保留定义。
2）无法删除属于 SYS 的表中的列，会报 ORA-12988 错误，哪怕你是 sys 用户都不可以。

实验：scott 下
```
create table a (id int, name char(10));
create index id_idx on a(id);
alter table a add constraint unq_id unique(id);
create sequence a_id start with 1 increment by 1;
create view v as select id from a;
insert into a values(a_id.nextval,'tim');
insert into a values(a_id.nextval,'ran');
commit;
```

SQL> select * from a;

```
        ID NAME
---------- ----------
         1 tim
         2 ran
```

```
SQL> select index_name from user_indexes where table_name='A';      --查看有关索引

INDEX_NAME
------------------------------
ID_IDX

SQL> select constraint_name from user_constraints where table_name='A'; --查看有关约束

CONSTRAINT_NAME
------------------------------
UNQ_ID

SQL> select object_name from user_objects where object_type='SEQUENCE'; --查看有关序列

OBJECT_NAME
----------------------------------------
A_ID

SQL> select text from user_views where view_name='V';           --查看有关视图

TEXT
--------------------------------------------------------------------------------
select id from a

SQL> alter table a drop column id;
```

重复查看有关信息，索引和约束随该列数据虽然被删除，但序列和视图的定义还在。

第十九章 通过 group by 产生统计报告

group by rollup，
group by cube，
grouping 和 grouping_id 函数
grouping set

Oracle 数据库中的 rollup 配合 group by 命令使用，可以提供信息汇总功能（与"小计"相似）
CUBE，也是 GROUP BY 子句的一种扩展，可以返回每一个列组合的小计记录，同时在末尾加上总计记录。

示例如下：

```
SQL> select job,deptno,sal from emp;

JOB           DEPTNO        SAL
--------- ---------- ----------
CLERK             20        800
SALESMAN          30       1600
SALESMAN          30       1250
MANAGER           20       2975
SALESMAN          30       1250
MANAGER           30       2850
MANAGER           10       2450
ANALYST           20       3000
PRESIDENT         10       5000
SALESMAN          30       1500
CLERK             20       1100
CLERK             30        950
ANALYST           20       3000
CLERK             10       1300

14 rows selected.
```

19.1 rollup 的用法

```
SQL> select job,deptno,sum(sal) total_sal from emp group by rollup(job,deptno);

JOB           DEPTNO  TOTAL_SAL
--------- ---------- ----------
CLERK             10       1300
CLERK             20       1900
CLERK             30        950
CLERK                      4150
ANALYST           20       6000
ANALYST                    6000
MANAGER           10       2450
MANAGER           20       2975
MANAGER           30       2850
MANAGER                    8275
SALESMAN          30       5600
SALESMAN                   5600
PRESIDENT         10       5000
PRESIDENT                  5000
                          29025

15 rows selected.
```

19.2 cube 的用法

```
SQL> select job,deptno,sum(sal) total_sal from emp group by cube(job,deptno);

JOB           DEPTNO  TOTAL_SAL
--------- ---------- ----------
                         29025
                 10        8750
                 20       10875
                 30        9400
CLERK                      4150
CLERK            10        1300
CLERK            20        1900
CLERK            30         950
ANALYST                    6000
ANALYST          20        6000
MANAGER                    8275
MANAGER          10        2450
MANAGER          20        2975
MANAGER          30        2850
SALESMAN                   5600
SALESMAN         30        5600
PRESIDENT                  5000
PRESIDENT        10        5000

18 rows selected.
```

可以看出,用了 rollup 的 group by 子句所产生的所谓的超级聚合就是指在在产生聚合时会从右向左逐个对每一列进行小结，并在结果中生成独立的一行，同时也会对聚合列生成一个合计列。

select deptno,job,sum(sal) from emp group by deptno,job;
会对每一个不同的 dept,job 生成一行独立的结果。

而 select deptno,job,sum(sal) from emp group by rollup(deptno,job);
的结果中除了上述的结果结果之外，还会对每一个 deptno 进行一个小结，并单独生成一行，除此之外还会对所有的 sal 求和并生成一列。

这里的 group by 后面我们仅仅接了 2 列，实际上我们可以使用更多列的，这样的话 oracle 就会以从右向左的方式来进行逐个小结。

这里需要注意的是使用了 group by 和 rollup 后，其后面的列要用括号括起来，否则将会出现 ORA-00933: SQL 命令未正确结束的错误。

看看 grouping、grouping_id 函数是什么？

19.3 grouping 函数和 grouping_id 函数

GROUPING 函数可以接受一列，返回 0 或者 1。如果列值为空，那么 GROUPING()返回 1；如果列值非空，那么返回 0。GROUPING 只能在使用 ROLLUP 或 CUBE 的查询中使用。当需要在返回空值的地方显示某个值时，GROUPING()就非常有用。

grouping_id 函数可以返回 0,1,2,3...可以分别表示小计，合计等信息。

```
SQL>
SQL> select job,deptno,sum(sal) total_sal,grouping(job) job_grp, grouping(deptno)
deptno_grp,grouping_id(job,deptno) total_grp from emp group by cube(job,deptno);
```

JOB	DEPTNO	TOTAL_SAL	JOB_GRP	DEPTNO_GRP	TOTAL_GRP
		29025	1	1	3
	10	8750	1	0	2
	20	10875	1	0	2
	30	9400	1	0	2
CLERK		4150	0	1	1
CLERK	10	1300	0	0	0
CLERK	20	1900	0	0	0
CLERK	30	950	0	0	0
ANALYST		6000	0	1	1
ANALYST	20	6000	0	0	0
MANAGER		8275	0	1	1
MANAGER	10	2450	0	0	0
MANAGER	20	2975	0	0	0
MANAGER	30	2850	0	0	0
SALESMAN		5600	0	1	1
SALESMAN	30	5600	0	0	0
PRESIDENT		5000	0	1	1
PRESIDENT	10	5000	0	0	0

已选择 18 行。

但是我们大多数情况下需要在查询的结果集的汇总列加上“合计”，怎么办呢？用 grouping 和 grouping_id 函数，然后再用 decode 函数判断一下是否为空就可以了

```
SQL>select grouping_id(job,deptno) as group_col,sum(sal) total_sal from emp group
by rollup(job,deptno);
```

GROUP_COL	TOTAL_SAL
0	1300
0	1900
0	950
1	4150
0	6000
1	6000
0	2450
0	2975
0	2850
1	8275
0	5600
1	5600
0	5000
1	5000
3	29025

已选择 15 行。

```
SQL>select  decode(grouping_id(job,deptno),1,'合计',3,'总计',job||deptno)  as
group_col,sum(sal) total_sal
from emp group by rollup(job,deptno);
```

GROUP_COL	TOTAL_SAL
CLERK10	1300
CLERK20	1900
CLERK30	950
合计	4150
ANALYST20	6000
合计	6000
MANAGER10	2450
MANAGER20	2975
MANAGER30	2850
合计	8275
SALESMAN30	5600
合计	5600
PRESIDENT10	5000
合计	5000
总计	29025

19.4 grouping sets

grouping sets 用于在一个 select 语句中定义多个 grouping, 相当于将 grouping set 中的多个 grouping 组合后再 UNION ALL。

例 1：

```
SELECT deptno, job, AVG(sal)
FROM emp
GROUP by (deptno,job);
```

例 2：

```
SELECT job,mgr, AVG(sal)
FROM emp
GROUP BY(job,mgr);
```

例 3 = （例 1）union all（例 2）

```
SELECT null,job,mgr, AVG(sal)
FROM emp
GROUP BY(job,mgr)
union all
SELECT deptno, job, null,AVG(sal)
FROM emp
GROUP by (deptno,job);
```

例 4=例 3, 但使用了 grouping sets

```
SELECT deptno,job,mgr, AVG(sal)
FROM emp
GROUP BY grouping sets ((deptno,job),(job,mgr));
```

DEPTNO	JOB	MGR	AVG(SAL)
	CLERK	7902	800
	PRESIDENT		5000
	CLERK	7698	950
	CLERK	7788	1100
	CLERK	7782	1300
	SALESMAN	7698	1400
	MANAGER	7839	2758.33333
	ANALYST	7566	3000
20	CLERK		950
30	SALESMAN		1400
20	MANAGER		2975

```
        30 CLERK                        950
        10 PRESIDENT                   5000
        30 MANAGER                     2850
        10 CLERK                       1300
        10 MANAGER                     2450
        20 ANALYST                     3000
```

已选择 17 行。

Oracle 给出了一个小表，可以表明 grouping sets 和 union all 的关系：

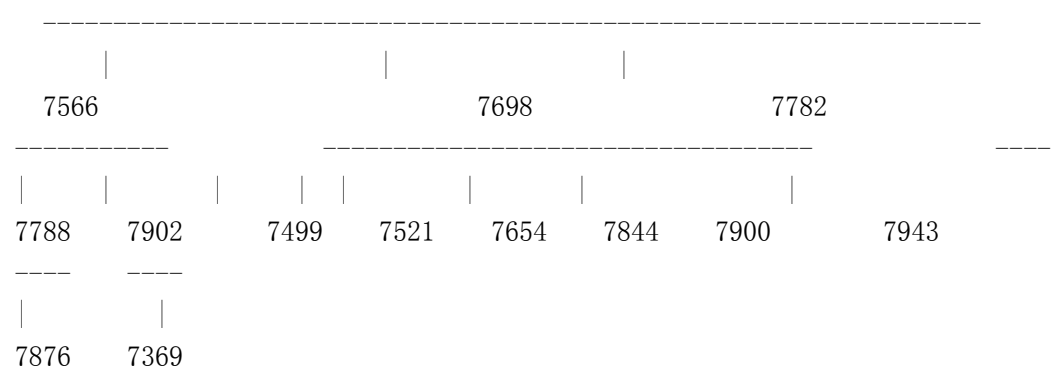
```
GROUPING SETS Statements            Equivalent GROUP BY Statements
--------------------------------------------------------------------------------
GROUP BY GROUPING SETS(a, b, c)          GROUP BY a UNION ALL
                         GROUP BY b UNION ALL
                         GROUP BY c

GROUP BY GROUPING SETS(a, b,(b, c))      GROUP BY a UNION ALL
                         GROUP BY b UNION ALL
                         GROUP BY b, c

GROUP BY GROUPING SETS((a, b, c))        GROUP BY a, b, c

GROUP BY GROUPING SETS(a, (b), ())       GROUP BY a UNION ALL
                         GROUP BY b UNION ALL
                         GROUP BY ()

GROUP BY GROUPING SETS(a,ROLLUP(b, c))       GROUP BY a UNION ALL
                         GROUP BY ROLLUP(b, c)
```

第二十章 ORACLE 分层查询 start with&connect by

20.1 树结构查询

ORACLE 是一个关系数据库管理系统,它用表的形式组织数据,在某些表中的数据还呈现出树型结构的联系。例如,我们现在讨论雇员信息表 EMP,其中含有雇员编号(EMPNO)和经理(MGR)两列，通过这两列反映出来的就是雇员之间领导和被领导的关系。他们之间的这种关系就是一种树结构。

图 1.1 EMP 表树结构图



遍历有两个方向

top--down 自上而下
说白了父亲找儿子，一个父亲可能有几个儿子，一个儿子可能有几个孙子，自上而下遍历是一个不能丢儿子的过程，顺序以左为先。

down--top 自底向上
简单说就是儿子找父亲，一个儿子只能有一个父亲，所以自底向上的顺序应该是：孙子->儿子-->父亲-->爷爷。

20.2 树结构的描述

树结构的数据存放在表中，数据之间的层次关系即父子关系，在表的每一行中都有一个表示父节点的 MGR（除根节点外）。

在 SELECT 命令中使用 CONNECT BY 和 START WITH 子句可以查询表中的树型结构关系。其命令格式如下：

```
SELECT ...

CONNECT BY {PRIOR 列名 1=列名 2|列名 1=PRIOR 列名 2}

[START WITH];
```

20.3 关于 CONNECT BY 子句

CONNECT BY 子句非常关键，它包含两个概念

如：CONNECT BY PRIOR 列名 1=列名 2：
1）top--down
2）列名 1 是父， 列名 2 是子

如：CONNECT BY 列名 1=PRIOR 列名 2：

1）down--up
2）列名 1 是子， 列名 2 是父

也就是说 PRIOR 运算符在的一侧表示父节点，在另一侧表示子节点，而在=号左边还是右边确定查找树结构是的顺序是自顶向下还是自底向上。在连接关系中，除了可以使用列名外，还允许使用列表达式。

START WITH 子句为可选项，用来标识哪个节点作为查找树型结构的根节点。若该子句被省略，则表示所有满足查询条件的行作为根节点。

例 1 以树结构方式显示 EMP 表的数据。

```
SQL>select empno,ename,mgr from emp
connect by prior empno=mgr
start with empno=7839
/
```

EMPNO	ENAME	MGR
7839	KING	
7566	JONES	7839
7788	SCOTT	7566
7876	ADAMS	7788
7902	FORD	7566
7369	SMITH	7902
7698	BLAKE	7839
7499	ALLEN	7698
7521	WARD	7698
7654	MARTIN	7698
7844	TURNER	7698
7900	JAMES	7698
7782	CLARK	7839
7934	MILLER	7782

14 rows selected.

仔细看 empno 这一列输出的顺序，就是上图树状结构每一条分支（从根节点开始）的结构。

例 2 从 SMITH 节点开始自底向上查找 EMP 的树结构。

```
SQL>select empno,ename,mgr
from emp
```

```
connect by empno=prior mgr
start with empno=7369
/

     EMPNO ENAME                MGR
---------- ---------- ----------
      7369 SMITH               7902
      7902 FORD                7566
      7566 JONES               7839
      7839 KING

SQL>
```

在这种自底向上的查找过程中，只有树中的一枝被显示。

20.4 定义查找起始节点

在自顶向下查询树结构时，不但可以从根节点开始，还可以定义任何节点为起始节点，以此开始向下查找。这样查找的结果就是以该节点为开始的结构树的一枝。

例 3 查找 7566(JONES)直接或间接领导的所有雇员信息。

```
SQL＞SELECT EMPNO,ENAME,MGR
FROM EMP
CONNECT BY PRIOR EMPNO=MGR
START WITH EMPNO=7566
/

     EMPNO ENAME                MGR
---------- ---------- ----------
      7566 JONES               7839
      7788 SCOTT               7566
      7876 ADAMS               7788
      7902 FORD                7566
      7369 SMITH               7902
```

START WITH 不但可以指定一个根节点，还可以指定多个根节点。

例 4 查找由 FORD 和 BLAKE 领导的所有雇员的信息。

```
SQL>SELECT EMPNO,ENAME,MGR
FROM EMP
```

```
CONNECT BY PRIOR EMPNO=MGR
START WITH ENAME IN ('FORD','BLAKE')
/

     EMPNO ENAME                MGR
---------- ---------- ----------
      7902 FORD                7566
      7369 SMITH               7902
      7698 BLAKE               7839
      7499 ALLEN               7698
      7521 WARD                7698
      7654 MARTIN              7698
      7844 TURNER              7698
      7900 JAMES               7698

8 rows selected.
```

20.5 使用 LEVEL

在查询中，可以使用伪列 LEVEL 显示每行数据的有关层次。LEVEL 将返回树型结构中当前节点的层次。

伪列 LEVEL 为数值型，可以在 SELECT 命令中用于各种计算。

例 5 使用 LEVEL 改变查询结果的显示形式。

```
SQL> COLUMN LEVEL FORMAT A20

SQL> SELECT LPAD(LEVEL,LEVEL*3,' ')
as "LEVEL",EMPNO,ENAME,MGR
FROM EMP
CONNECT BY PRIOR EMPNO=MGR
START WITH ENAME='KING'
/

LEVEL                     EMPNO ENAME      MGR
-------------------- ---------- ---------- ----------
  1                        7839 KING
     2                     7566 JONES            7839
        3                  7788 SCOTT            7566
           4               7876 ADAMS            7788
        3                  7902 FORD             7566
           4               7369 SMITH            7902
     2                     7698 BLAKE            7839
```

```
         3                7499 ALLEN           7698
         3                7521 WARD            7698
         3                7654 MARTIN          7698
         3                7844 TURNER          7698
         3                7900 JAMES           7698
      2                   7782 CLARK           7839
         3                7934 MILLER          7782

14 rows selected.
```

在 SELECT 使用了函数 LPAD，该函数表示以 LEVEL*3 个空格进行填充，由于不同行处于不同的节点位置，具有不同的 LEVEL 值，因此填充的空格数将根据各自的层号确定，空格再与层号拼接，结果显示出这种层次关系。

只查看 2 级的员工信息：

```
SQL> select t1.* from (select level LNUM ,ename,mgr from emp connect by prior
empno=mgr start with ename='KING') t1 where LNUM=2;

      LNUM ENAME      MGR
---------- ---------- ----------
         2 JONES      7839
         2 BLAKE      7839
         2 CLARK      7839
```

20.6 节点和分支的裁剪

在对树结构进行查询时，可以去掉表中的某些行，也可以剪掉树中的一个分支，使用 WHERE 子句来限定树型结构中的单个节点，以去掉树中的单个节点，但它却不影响其后代节点（自顶向下检索时）或前辈节点（自底向顶检索时）。

```
SQL>SELECT LPAD(LEVEL,LEVEL*3,' ')
as "LEVEL",EMPNO,ENAME,MGR
FROM EMP
WHERE ENAME<>'SCOTT'
CONNECT BY PRIOR EMPNO=MGR
START WITH ENAME='KING'
/

LEVEL                EMPNO ENAME      MGR
-------------------- ----------  ---------- ----------
  1                  7839 KING
     2               7566 JONES      7839
```

```
         4                7876 ADAMS           7788
      3                   7902 FORD            7566
         4                7369 SMITH           7902
   2                      7698 BLAKE           7839
      3                   7499 ALLEN           7698
      3                   7521 WARD            7698
      3                   7654 MARTIN          7698
      3                   7844 TURNER          7698
      3                   7900 JAMES           7698
   2                      7782 CLARK           7839
      3                   7934 MILLER          7782

13 rows selected.
```

在这个查询中，仅剪去了树中单个节点 SCOTT。若希望剪去树结构中的某个分支，则要用 CONNECT BY 子句。CONNECT BY 子句是限定树型结构中的整个分支，既要剪除分支上的单个节点，也要剪除其后代节点（自顶向下检索时）或前辈节点（自底向顶检索时）。

例 8. 显示 KING 领导下的全体雇员信息，除去 SCOTT 领导的一支。

```
SQL>SELECT LPAD(LEVEL,LEVEL*3,' ')
as "LEVEL",EMPNO,ENAME,MGR
FROM EMP
CONNECT BY PRIOR EMPNO=MGR
AND ENAME!='SCOTT'
START WITH ENAME='KING'
/

LEVEL                  EMPNO ENAME      MGR
-------------------- ------- ---------- -------
  1                     7839 KING
     2                  7566 JONES      7839
        3               7902 FORD       7566
           4            7369 SMITH      7902
     2                  7698 BLAKE      7839
        3               7499 ALLEN      7698
        3               7521 WARD       7698
        3               7654 MARTIN     7698
        3               7844 TURNER     7698
        3               7900 JAMES      7698
     2                  7782 CLARK      7839
        3               7934 MILLER     7782

12 rows selected
```

这个查询结果就除了剪去单个节点 SCOTT 外，还将 SCOTT 的子节点 ADAMS 剪掉，即把 SCOTT 这个分支剪掉了。

当然 WHERE 子句可以和 CONNECT BY 子句联合使用，这样能够同时剪掉单个节点和树中的某个分支。

例 9. 显示 KING 领导全体雇员信息，除去雇员 SCOTT，以及 BLAKE 领导的一支。

（这个留给大家实践吧：）

在使用 SELECT 语句来报告树结构报表时应当注意,CONNECT BY 子句不能作用于出现在 WHERE 子句中的表连接。如果需要进行连接，可以先用树结构建立一个视图，再将这个视图与其他表连接，以完成所需要的查询。

第二十一章 Oracle 时间类型及 Timezone

21.1 Oracle 的六种时间类型

DATE
TIMESTAMP
TIMESTAMP WITH TIME ZONE
TIMESTAMP WITH LOCAL TIME ZONE
INTERVAL YEAR TO MONTH
INTERVAL DAY TO SECOND

21.2 DATE 和 TIMESTAMP 类型

1）DATE 只能精确到秒

```
SQL> select sysdate from dual;

SYSDATE
-------------------
2013-03-10 07:07:00
```

2）TIMESTAMP 类型是 DATE 型的扩展

```
SQL> select localtimestamp from dual;

LOCALTIMESTAMP
---------------------------------------------------------------------------
```

2013-06-17 10:53:32.201058

3）DATE 类型与 TIMESTAMP 类型，可以通过 CAST 函数互转。

```
SQL>select sysdate,CAST(sysdate AS TIMESTAMP) "date" from dual;

SYSDATE             date
------------------- ---------------------------------------------------------
---------------
2013-03-10 07:16:28 2013-03-10 07:16:28.000000
```

4）TIMESTAMP 类型可以表示秒小数后 9 位

```
SQL> select to_char(systimestamp, 'yyyy-mm-dd hh24:mi:ss:ff9 TZR') as
"systimestamp" from dual;

systimestamp
-------------------------------------------------------------
2013-03-10 07:14:31:062292000 +08:00
```

21.3 Timezone 引入的背景

Oracle 中有很多特性支持国际化，如字符集、时区等等。和时区相关的两个时间类型是：

```
TIMESTAMP WITH TIME ZONE
TIMESTAMP WITH LOCAL TIME ZONE
```

1）Database 的 timezone 可以在创建数据库的时候指定，如：

```
CREATE DATABASE db01
...
SET TIME_ZONE='+08:00';
```

查看数据库时区信息：

```
SQL> select dbtimezone from dual;

DBTIME
------
+08:00
```

2）session 的 timezone 可以简单通过 alter session 语句修改：

ALTER SESSION SET TIME_ZONE='+08:00';

查看 session 时区信息：

SQL> select sessiontimezone from dual;

SESSIONTIMEZONE

+08:00

21.4 范例： 模拟北京、东京、伦敦三地的时区，进一步理解 Timezone。

全球的一个统一的时间应由 时区+时刻来指定，
比如 2005-4-6 14:00:00.000 并不能说清楚到底这是北京的下午 2 点还是东京的下午两点。两地时差差一个小时。

2005-4-6 14:00:00.000 +8:00 才是北京时间
2005-4-6 14:00:00.000 +9:00 则是东京时间

假设总部伦敦有 online meeting system, 已经由分部登记了两个网络会议，一个在北京，另一个在东京。

DB 服务器在英国（用 vbox 主控台模拟） alter session set dbtimezone ='+0:00' //查看 select dbtimezone from dual;
管理客户端 c-en 在英国 （SecureCRT 模拟） alter session set time_zone ='+0:00' //查看 select sessiontimezone from dual;
一个客户端 c-cn 在中国 （PL/SQL 模拟） alter session set time_zone ='+8:00' //查看 select sessiontimezone from dual;
一个客户端 c-jp 在日本 （CMD 模拟） alter session set time_zone ='+9:00' //查看 select sessiontimezone from dual;

为了显示时区格式一致，分别在三个 session 下执行下面语句

alter session set nls_timestamp_format='yyyy-mm-dd hh24:mi';
alter session set NLS_TIMESTAMP_TZ_FORMAT='yyyy-mm-dd HH24:MI:SS TZR';

假设伦敦服务器里有一个会议表:(这个表只有三个日期类型的字段，代表三种不同的日期类型)

scott:
create table meeting_table (ctime1 timestamp,ctime2 timestamp with time zone, ctime3 timestamp with local time zone);

北京用户登记了第一个会议，要在当地早上 8 点开会
东京用户登记了第二个会议，也要当地早上 8 点开会

两地分别插入以下记录

```
insert into meeting_table values (
to_timestamp('2005-06-29 8:00:00', 'yyyy-mm-dd hh24:mi:ss'),
to_timestamp('2005-06-29 8:00:00', 'yyyy-mm-dd hh24:mi:ss'),
to_timestamp('2005-06-29 8:00:00', 'yyyy-mm-dd hh24:mi:ss')
);
commit;
```

为了显示日期格式统一，在三地分别执行以下设置

```
col ctime1 for a20;
col ctime2 for a30;
col ctime3 for a25;
```

查看三地显示结果：

北京：

```
SQL> select * from meeting_table;

CTIME1               CTIME2                         CTIME3
-------------------- ------------------------------ -------------------------
2005-06-29 08;00     2005-06-29 08:00:00 +08:00     2005-06-29 08;00
2005-06-29 08;00     2005-06-29 08:00:00 +09:00     2005-06-29 07;00
```

东京：

```
SQL> select * from meeting_table;

CTIME1               CTIME2                         CTIME3
-------------------- ------------------------------ -------------------------
2005-06-29 08;00     2005-06-29 08:00:00 +08:00     2005-06-29 09;00
2005-06-29 08;00     2005-06-29 08:00:00 +09:00     2005-06-29 08;00
```

伦敦：

```
SQL> select * from meeting_table;

CTIME1               CTIME2                         CTIME3
-------------------- ------------------------------ -------------------------
```

2005-06-29 08;00　　2005-06-29 08:00:00 +08:00　　2005-06-29 00;00
2005-06-29 08;00　　2005-06-29 08:00:00 +09:00　　2005-06-28 23;00

这个例子总结几点：
1）timestamp 无法区分哪个是北京的会，哪个是东京的会（都是 8 点）
2）timestamp with time zone 给出了时区，可以识别在哪里开会，但伦敦的管理员要在伦敦本地出席其中的网络会议，需要人工换算时间。
3）timestamp with local time zone，时间自动转换成了本地时间，显示的比较友好。

21.5（INTERVAL YEAR TO MONTH）和（INTERVAL DAY TO SECOND）时间间隔数据类型

从 Oracle 9i 开始，按照 SQL99 标准，增加了时间间隔型数据 INTERVAL YEAR TO MONTH 和 INTERVAL DAY TO SECOND。
用 YEAR TO MONTH 表示时间间隔大小时要在年和月之间用一个连字符(-) 连接。
而 DAY TO SECOND 表示时间间隔大小时要在日和时间之间用一个空格连接。

举个例子来说，下面是 2 年 6 个月的时间间隔的表示方法：
INTERVAL ″2-6″ YEAR TO MONTH

下面的例子表示 3 天 12 个小时 30 分钟 6.7 秒：
INTERVAL ″3 12:30:06.7″ DAY TO SECOND(1)

时间间隔可以为正，也可以为负。它们可以从各种 TIMESTAMP 数据类型中加上或者减去，从而得到一个新的 TIMESTAMP 数据类型。

因为有精度问题，相对来讲，INTERVAL DAY TO SECOND 比 INTERVAL YEAR TO MONTH 要复杂一些
看看下面时间间隔关于 INTERVAL DAY TO SECOND 的字面含义说明

INTERVAL ‘3’ DAY　　//时间间隔为 3 天
INTERVAL ‘2’ HOUR　　//时间间隔为 2 小时
INTERVAL ‘25’ MINUTE　　//时间间隔为 25 分钟
INTERVAL ‘45’ SECOND　　//时间间隔为 45 秒
INTERVAL ‘3 2’ DAY TO HOUR　　//时间间隔为 3 天零 2 小时
INTERVAL ‘3 2:25’ DAY TO MINUTE　　//时间间隔为 3 天零 2 小时 25 分
INTERVAL ‘3 2:25:45’ DAY TO SECOND　　//时间间隔为 3 天零 2 小时 25 分 45 秒
INTERVAL ‘123 2:25:45.12’ DAY(3) TO SECOND(2) //时间间隔为 123 天零 2 小时 25 分 45.12 秒； 天的精度是 3 位，秒的部分的精度是 2 位.
INTERVAL ‘-3 2:25:45’ DAY TO SECOND　　//时间间隔为负数，值为 3 天零 2 小时 25 分 45 秒
INTERVAL ‘1234 2:25:45’ DAY(3) TO SECOND //时间间隔无效，因为天的位数超过了指定的精度 3
INTERVAL ‘123 2:25:45.123’ DAY TO SECOND(2) //时间间隔无效，因为秒的小数部分

的位数超过了指定的精度 2

21.6 关于 numtoyminterval 函数和 numtodsinterval 函数

numtoyminterval 用于产生一个指定的时间间隔，可以作为 interval year to month 型的数据插入到表中。

```
SQL>                                                                    select
sysdate,sysdate+numtoyminterval(1,'month'),sysdate+numtoyminterval(1,'year')
from dual;

SYSDATE             SYSDATE+NUMTOYMINTE SYSDATE+NUMTOYMINTE
------------------- ------------------- -------------------
2012-07-09 08:18:59 2012-08-09 08:18:59 2013-07-09 08:18:59
```

可以看出 numtoyminterval 产生了一个月的时间间隔和一个年的时间间隔

```
SQL>                                                                    select
sysdate,sysdate+numtodsinterval(1,'day'),sysdate+numtodsinterval(1,'second')
from dual;

SYSDATE             SYSDATE+NUMTODSINTE SYSDATE+NUMTODSINTE
------------------- ------------------- -------------------
2012-07-09 09:09:06 2012-07-10 09:09:06 2012-07-09 09:09:07
```

可以看出 numtodsinterval 产生了一天的时间间隔和一秒的时间间隔

21.7 关于 to_yminterval 和 to_dsinterval 转换函数

TO_YMINTERVAL 把 CHAR、VARCHAR2、NCHAR、NVARCHAR2 字符串转换 INTERVAL YEAR TO MONTH 类型。

Years 属于 integer，取值 0-999999999

Months 也属于 integer，取值 0-11

```
SQL> select to_yminterval('15-12') event_time from dual;
select to_yminterval('15-12') event_time from dual
                     *
第 1 行出现错误:
ORA-01843: 无效的月份

SQL> select to_yminterval('15-11') event_time from dual;
```

```
EVENT_TIME
--------------------------------------------------------------------------------
+000000015-11
```

第二十二章：全球化特性与字符集

数据库的全球化特性是数据库发展的必然结果，位于不同地区、不同国家、不用语言而使用同一数据库越来越普遍。Oracle 数据库提供
了对全球化数据库的支持，消除不同文字、语言环境、历法货币等所带来的差异、使得更容易、更方便来使用数据库。

22.1 Oracle 全球化特性内容

Language support
Territory support
Character set support
Linguistic sorting
Message support
Date and time formats
Numeric formats
Monetary formats

在 Oracle 全球化特性中最重要的则是字符集，即使用何种字符集将数据存储在数据库中

22.2 什么是字符集

主要是讨论两个问题，一是字符如何存储，二是字符如何显示。比如单个英文字符、单个阿拉伯数据字，#、$等，美国 ANSI 使用的标准字符集则使用了一个单字节(7 位)来表示。由于世界各国和各个地区使用的符号的多样性，仅有 2 的 7 次方（128）个单字节的码点是不够用的，因此就有需要多字节来表示各自不同的字符。
正是由于上述原因产生了不同的字符集的概念， 如美国使用的为 US7ASCII,西欧则使用的是 WE8ISO8859P1,中国则是 ZHS16GBK。

为了统一世界各国字符编码，统一编码字符集的概念应运而生，这就是 Unicode。

在 Oracle 中，几种常用的 Unicode 为 UTF-8，AL16UTF16，AL32UTF8

22.3 Oracle 所支持的字符集及分类

Oracle 支持两百多种字符集，包含了单字节、可变字节以及通用字符集等。
字符集通常根据使用的字节数来分类,主要分为以下几类

a.单字节字符集，如 US7ASCII(7bit)，WE8ISO8859P1(8bit)，WE8DEC(8bit)
b.可变长多字节字符集，如 JEUC，CGB2312-80
c.固定长多字节字符集，AL16UTF16

22.4 Oracle 数据库支持的 Unicode 字符集

1）数据库字符集和国家字符集特性

Database Character Sets
主要是用作描述字符如何保存。

National Character Set：
主要是用于辅助 Database Character Set。因为早期的数据库中很多使用了单字节字符集，但随着业务的发展，需要使用到诸如 nchar,nvarchar 等 Unicode 字符或者需要扩展到世界各地存储不同的字符，因此辅助字符集应运而生。

两者的比较：

Database Character Set	National Character Set
创建时被定义	创建时被定义
创建后不能被改变，仅有很少列外	有限的可以被改变
可存储列的类型为 char,varchar2,clob,long	可存储的类型为 NCHAR,NVARCHAR2,NCLOB

2) 通用字符集特性

Character Set	Unicode Encoding	Database Character Set	National Character Set
UTF8	UTF-8	Yes	Yes(Oracle 9i and 10g only)
AL32UTF8	UTF-8	Yes	No
AL16UTF16	UTF-16	No	Yes

3) 字符集影响的数据类型

对于二进制数据类型，字符集的设置不影响该类型数据的存储，如视频、音频等

受影响的数据类型为：char,varchar2,nchar,nvarchar2,blob,clob,long,nclob

4)设置字符集的五个级别,优先级依次递增。

Database Server《 Instance《 Initialzation parameter《 Client Environment variable 《 alter session command

22.5 相关 NLS 参数的设定

1.查看 NLS 参数

a.查看本次会话中设定及使用的 NLS 参数，nls_session_parameters 视图决定了 session 显示信息的形式：

```
col parameter format a28
col value format a35
SQL> select * from nls_session_parameters;

PARAMETER                    VALUE
---------------------------- -----------------------------------
NLS_LANGUAGE                 SIMPLIFIED CHINESE
NLS_TERRITORY                CHINA
NLS_CURRENCY                 ￥
NLS_ISO_CURRENCY             CHINA
NLS_NUMERIC_CHARACTERS       .,
NLS_CALENDAR                 GREGORIAN
NLS_DATE_FORMAT              YYYY-MM-DD HH24:MI:SS
NLS_DATE_LANGUAGE            SIMPLIFIED CHINESE
NLS_SORT                     BINARY
NLS_TIME_FORMAT              HH.MI.SSXFF AM
NLS_TIMESTAMP_FORMAT         yyyy-mm-dd HH24:MI:SSXFF
NLS_TIME_TZ_FORMAT           HH.MI.SSXFF AM TZR
NLS_TIMESTAMP_TZ_FORMAT      yyyy-mm-dd HH24:MI:SSXFF  TZR
NLS_DUAL_CURRENCY            ￥
NLS_COMP                     BINARY
NLS_LENGTH_SEMANTICS         BYTE
NLS_NCHAR_CONV_EXCP          FALSE
已选择 17 行。
```

b.查看数据库服务器中设定的 NLS 参数使用 nls_database_parameters 视图;

```
SQL>  select * from nls_database_parameters;

PARAMETER                    VALUE
---------------------------- -----------------------------------
```

```
NLS_LANGUAGE                    AMERICAN
NLS_TERRITORY                   AMERICA
NLS_CURRENCY                    $
NLS_ISO_CURRENCY                AMERICA
NLS_NUMERIC_CHARACTERS          .,
NLS_CHARACTERSET                ZHS16GBK
NLS_CALENDAR                    GREGORIAN
NLS_DATE_FORMAT                 DD-MON-RR
NLS_DATE_LANGUAGE               AMERICAN
NLS_SORT                        BINARY
NLS_TIME_FORMAT                 HH.MI.SSXFF AM
NLS_TIMESTAMP_FORMAT            DD-MON-RR HH.MI.SSXFF AM
NLS_TIME_TZ_FORMAT              HH.MI.SSXFF AM TZR
NLS_TIMESTAMP_TZ_FORMAT         DD-MON-RR HH.MI.SSXFF AM TZR
NLS_DUAL_CURRENCY               $
NLS_COMP                        BINARY
NLS_LENGTH_SEMANTICS            BYTE
NLS_NCHAR_CONV_EXCP             FALSE
NLS_NCHAR_CHARACTERSET          AL16UTF16
NLS_RDBMS_VERSION               11.1.0.6.0
已选择 20 行。
```

几个重要的参数：
1）语言参数，nls_language：
受影响的参数有：

```
NLS_DATE_LANGUAGE
NLS_SORT
```

2）区域参数，nls_territory：
受影响的参数有：

```
NLS_CURRENCY
NLS_DUAL_CURRENCY
NLS_ISO_CURRENCY
NLS_NUMERIC_CHARACTERS
NLS_DATE_FORMAT
NLS_TIMESTAMP_FORMAT
NLS_TIMESTAMP_TZ_FORMAT
```

通常，在 nls_language 设定后，应为 nls_territory 设定合理的值，假如语言设定为简体中文，地区设定为澳大利亚则不太合理
对于使用同样的语言不同国家或地区,比如英语,澳大利亚和英国,则 nls_territory 设定不同,则同样影响相关参数如 currency 等

3）排序参数，nls_sort：

Order by 指定字段的排序方法，缺省的是 Binary, 一般是支持单字节字符集 而多字节的字符集排序则引入 Linguistic Sorting

基于 Binary 排序，根据 encode 的二进制代码排序。
基于语言排序，又分单一语言和多重语言

NLSSORT 函数，可以在 sql 语句中定义，优先级最高，覆盖其他级别的定义。

```
ALTER SESSION SET NLS_SORT=BINARY;
SELECT num,word,def FROM list ORDER BY NLSSORT(word,'NLS_SORT=FRENCH_M');
```

4）NLS_LANG 参数的设定

NLS_LANG 为一个总控参数，控制了前面描述的 nls_language 和 nls_territory 的行为
该参数可以用于设定服务器和客户端的 language 和 territory 值，也可设置客户端输入数据和显示的字符集
只要设定了该参数，则其它参数就确定了。当然也可以只设定其中的一部分，另外，特别注意 NLS_LANG 只能在环境变量中设定。
该参数的格式为： NLS_LANG = language_territory.charset 如：NLS_LANG=French_France.UTF8

在我们的虚机环境下，环境变量文件/home/oracle/.bash_profile 中描述了作为客户端的 NLS_LANG

NLS_LANG="simplified chinese"_china.zhs16gbk，该参数分为几个部分来设定

第一部分为 language，为 simplified chinese。
第二部分为 territory，为 china。一二两部分必须用下划线连接。
第三部分为 character set，为 zhs16gbk 二三两部分必须用小数点连接。

其含义是语言是简体中文，区域是中国，数据库字符集是 zhs16gbk。

另外日期格式缺省的是 DD-MON-RR，我们单独定义了适合中国人使用的格式'YYYY-MM-DD HH24:MI:SS'

22.6 改变字符集

9i 之前无法改变字符集，9i 后 Oracle 提供了扫描字符集的工具，但无法保证其有效。

对于 Database Character Set 在 Unix 平台上 Oracle 提供的实用工具是：

数据库字符扫描工具 csscan
语言与文字扫描工具 lcsscan

如：csscan system/systempassword full=y tocher=utf8

1）转换数据库字符集，前提是 csscan 成功
使用 alter database character set 命令。

2）转换国家字符集，前提是转换后的字符集必须是转换前的字符集的超集。
使用 alter database national character Set 命令。

第二十三章 正则表达式

22.1 ORACLE 中的支持正则表达式的函数主要有下面四个：

1，REGEXP_LIKE ： 与 LIKE 的功能相似
2，REGEXP_INSTR ： 与 INSTR 的功能相似
3，REGEXP_SUBSTR ： 与 SUBSTR 的功能相似
4，REGEXP_REPLACE ： 与 REPLACE 的功能相似

它们在用法上与 Oracle SQL 函数 LIKE、INSTR、SUBSTR 和 REPLACE 用法相同，
但是它们使用 POSIX 正则表达式代替了老的百分号（%）和通配符（_）字符。

22.2 POSIX 正则表达式由标准的元字符（metacharacters）所构成：

’^’ 匹配输入字符串的开始位置，在方括号表达式中使用，此时它表示不接受该字符集合。
’$’ 匹配输入字符串的结尾位置。如果设置了 RegExp 对象的 Multiline 属性，则 $ 也匹配 ’\n’ 或 ’\r’。
’.’ 匹配除换行符之外的任何单字符。
’?’ 匹配前面的子表达式零次或一次。
’+’ 匹配前面的子表达式一次或多次。
’*’ 匹配前面的子表达式零次或多次。
’|’ 指明两项之间的一个选择。例子’^([a-z]+|[0-9]+)$’表示所有小写字母或数字组合成的字符串。
’( )’ 标记一个子表达式的开始和结束位置。
’[]’ 标记一个中括号表达式。

’{m,n}’ 一个精确地出现次数范围，m=<出现次数<=n，’{m}’表示出现 m 次，’{m,}’表示至少出现 m 次。
\num 匹配 num，其中 num 是一个正整数。对所获取的匹配的引用。

22.3 字符簇：
[[:alpha:]] 任何字母。

[[:digit:]] 任何数字。
[[:alnum:]] 任何字母和数字。
[[:space:]] 任何白字符。
[[:upper:]] 任何大写字母。
[[:lower:]] 任何小写字母。
[[:punct:]] 任何标点符号。
[[:xdigit:]] 任何 16 进制的数字，相当于[0-9a-fA-F]。

各种操作符的运算优先级
\转义符
(), (?:), (?=), [] 圆括号和方括号
*, +, ?, {n}, {n,}, {n,m} 限定符
^, $, anymetacharacter 位置和顺序
|
*/

22.4 Oracle REGEXP_LIKE 介绍和例子

```
--创建表
create table fzq (id varchar2(4),value varchar2(10));
--数据插入
insert into fzq values ('1','1234560');
insert into fzq values ('2','1234560');
insert into fzq values ('3','1b3b560');
insert into fzq values ('4','abc');
insert into fzq values ('5','abcde');
insert into fzq values ('6','ADREasx');
insert into fzq values ('7','123 45');
insert into fzq values ('8','adc de');
insert into fzq values ('9','adc,.de');
insert into fzq values ('10','1B');
insert into fzq values ('10','abcbvbnb');
insert into fzq values ('11','11114560');
insert into fzq values ('11','11124560');

--regexp_like
--查询 value 中以 1 开头 60 结束的记录并且长度是 7 位
select * from fzq where value like '1____60';
select * from fzq where regexp_like(value,'1....60');

--查询 value 中以 1 开头 60 结束的记录并且长度是 7 位并且全部是数字的记录。
--使用 like 就不是很好实现了。
select * from fzq where regexp_like(value,'1[0-9]{4}60');

-- 也可以这样实现，使用字符集。
```

```
select * from fzq where regexp_like(value,'1[[:digit:]]{4}60');

-- 查询 value 中不是纯数字的记录
select * from fzq where not regexp_like(value,'^[[:digit:]]+$');

-- 查询 value 中不包含任何数字的记录。
select * from fzq where regexp_like(value,'^[^[:digit:]]+$');

--查询以 12 或者 1b 开头的记录.不区分大小写。
select * from fzq where regexp_like(value,'^1[2b]','i');

--查询以 12 或者 1b 开头的记录.区分大小写。
select * from fzq where regexp_like(value,'^1[2B]');

-- 查询数据中包含空白的记录。
select * from fzq where regexp_like(value,'[[:space:]]');

--查询所有包含小写字母或者数字的记录。
select * from fzq where regexp_like(value,'^([a-z]+|[0-9]+)$');

--查询任何包含标点符号的记录。
select * from fzq where regexp_like(value,'[[:punct:]]');
```

22.5 REGEXP_REPLACE(字符串替换函数)

REPLACE 函数是用另外一个值来替代串中的某个值。例如，可以用一个匹配数字来替代字母的每一次出现。REPLACE 的格式如下所示：

原型:regexp_replace(x,pattern[,replace_string[,start[,occurence[match_option]]]])

每个参数的意思分别是：

x 待匹配的函数
pattern 正则表达式元字符构成的匹配模式
replace_string 替换字符串
start 开始位置
occurence 匹配次数
match_option 匹配参数，这里的匹配参数和 regexp_like 是完全一样的，可参考前面的一篇文章。

举例来讲：

SQL> select regexp_replace('hello everybody,051courses will be over

```
soon,thanks.','b[[:alpha:]]{3}','one') from dual;

REGEXP_REPLACE('HELLOEVERYBODY,047COURSESWILLBEOVER
--------------------------------------------------
hello everyone,051courses will be over soon,thanks.
```