# ORACLE SQL

## 数据库(Database)

数据库(Database DB):

按照数据结构来组织、存储和管理数据的仓库。

数据库管理系统(Database Management System DBMS):

管理数据库的软件。

关系型数据库:描述两个元素之间的关联或者对应关系，并使用关系据模型将数据组织到二维数据表中。

常见:Oracle(甲骨文),DB2(IBM),Sybase,SQL Server(微软),My SQL(免费)

表(Table):一个关系型数据库由多个数据表组成，数据表是关系型数据库的基本存储结构。

表是二维的，由行和列组成:

行(ROW):横排数据，也叫做记录(Record);

列(COLUMN):竖排数据，也叫做字段(Field).

表和表之间存在关联关系。

结构化查询语言(SQL Structured Query Language)

数据库中无论字段是什么类型，默认值都是NULL。

也就是说当我们向表中插入一条数据时，若对应的字段没有给定值，

那么默认就将NULL作为值插入到该字段上。

可以使用DEFAULT为某个字段指定默认值，这样就会用指定的值插入当前字段。

SQL语句本身并不区分大小写，但是行业中使用时，

通常都会将关键字全部大写，其他内容全部小写，形成差异来增加可读性。

但是需要注意，字符串在数据库使用单引号括起来，并且内容区分大小写。

NUMBER 数字型数据类型，常常定义NUMBER(m,n)

m表示数字总位数，n表示小数点后面的位数

n不写代表整数，如果不写参数默认值为1

CHAR 固定长度的字符类型，常常定义为CHAR(n)

n代表占用字节数，最大长度为2000字节

不写参数默认值为1

VARCHAR 变长的字符类型，常常定义为VARCHAR(n)

n必须写，没有默认值，最大长度为4000字节

VARCHAR2 变长的字符类型，Oracle数据库内部使用数据类型。

和VARCHAR一样，参数必须写，没有默认值。Oracle保证在SQL标准改变的情况下保证功能不变。

DATE 时间类型，固定7个字节长度，默认格式是DD-MON-RR

### **数据定义语言(DDL Data Definition Language)**

DDL语言不受事务控制。

1.新建表:CREATE TABLE table\_name(field\_name field\_type)

2.查看表结构:DESC table\_name/view\_name

3.删除:DROP TABLE/VIEW/SEQUENCE/INDEX/CONSTRAINT name

4.修改表名:RENAME old\_name TO new\_name

5修改表结构：

5.1 插入字段:

ALTER TABLE table\_name ADD (field\_name field\_type)

注意：a.尽量不要在表中有数据以后修改表；

b.该语句只能在表的末尾添加,而不是在现有字段中插入一个新字段。

5.2 修改字段名:

ALTER TABLE table\_name

RENAME COLUMN old\_field TO new\_field

5.3.删除现有字段:

尽量不要在表中有数据以后修改表，会导致删除该字段的所有数据

ALTER TABLE table\_name

DROP (field)

5.4 修改现有字段:可以修改字段的长度，类型，默认值

尽量不要在表中有数据以后修改表，原数据比修改大导致修改不成功

ALTER TABLE table\_name

MODIFY

(***Field type [DEFAULT default\_field]***)

5.5 删除表中全部数据，保留表结构

TRUNCATE TABLE table\_name

### **数据操作语言(DML Data Manipulation Language)**

DML语句可以对表中的数据进行"**增、删、改**"

DML语句是伴随着**事务控制**的,只有提交(**COMMIT**)后才能写入硬盘

1.插入语句:字段顺序没有关系

**INSERT**语句可以不指定针对哪些字段插入，若不指定，则是全字段插入

这时候VALUES中给定的值的顺序、类型、个数必须与表中字段一致才可以。

***INSERT INTO*** table\_name

***(field\_name1,filed\_name2……)***

***VALUES***

***(field\_value1,field\_value2……)***;

2.更新语句:

修改表中数据，需要注意，不加WHERE条件，整张表的所有记录都会被修改。

***UPDATE*** table\_name

***SET field\_name1 = field\_value1, filed\_name2 = field\_value……*** --SET =号表示赋值

***WHERE field\_name3 = field\_value3 [AND…OR…]*** --WHERE=号表示判断是否相等

3.删除表中数据:

DELETE语句，逐行删除，数据量大比较慢

DDL语句中的TRUNCATE语句可以快速删除，但是删除的是整张表数据，保留表结构，不受事务控制。

删除表中数据，需要注意，不加WHERE条件，整张表的所有记录都会被删除。

***DELETE*** table\_name

***WHERE*** ***field\_name = field\_value***

### **事务控制语言(TCL Transaction Control Lanuage)**

1.提交:确定已经进行的数据改变操作

COMMIT

2.回滚:取消已经进行的数据改变操作

ROLLBACK

3.保存点:使当前事务回到保存点，用于回退部分改变

SAVINGPOINT

### **数据查询语言(DQL Data Query Lanuage)**

SELECT

### **数据控制语言(DCL Data Control Lanuage)**

1.GRANT:授权

2.REVOE:收回权限

3.CREATE USER:新建用户

## **SQL API(Base type)**

### *PART A. CHAR/VARCHAR/VACHAR2*

1.CONCAT语句:用来连接字符串，有且只有两个参数。

***CONCAT(char1,char2)***

2.||连接符号。连接字符串。

***char1*||*char2*||*char3……***

3.LENGTH 字段长度/个数

***LENGTH(filed)***

4.UPPER/LOWER/INITCAP

DUAL伪表:为了满足SELECT语句要求，但是要查询的数据又与现有的表只能个数据没有关系时使用。

***UPPER***(***char***) –-所有英文字符大写

***LOWER*** (***char***) –-所有英文字符小写

**INITCAP** (***char***) –-所有英文单词首字母大写

5.TRIM/LTRIM/RTRIM

5.1 TRIM:去除字符串左右两面重复的给定的字符。

**char1**参数**只能写一个字符**。

***TRIM****(****char1******FROM******char****)*

*示例：*

SELECT TRIM('   Sample   ');

结果：Sample

5.2 LTRIN/RTRIM:分别去除字符串左边/右边的给定的字符。

可以连续去除左边或者右边出现的给定的字符内容。

给定的字符顺序不关心，只要当前字符串中包含给定的字符就去除。

***LTRIM(char, char1) -–char1*** 可以是如'a'之类的字符

***RTRIM(char, char1) -–char1*** 也可以是如'esa'之类的字符组合

6.补位函数:LPAD/RPAD

***LPAD(str1,n,str2)***

显示str1的内容，要求显示n的长度。

若str1的长度不足n，则在左侧补充若干个str2字符以达到长度。

需要注意，str2必须是单一的字符。

LPAD常被用来实现'右对齐'，RPAD为'左对齐'

当显示的位数不足时补充字符，若显示的内容长度超过要显示的位数时，

只是从左边开始显示给定长度的字符，剩下的会被去除。

***SELECT ename,RPAD(sal,5,' ') FROM emp\_chris***//左对齐，右边补空格

***SELECT ename,LPAD(sal,6,' ') FROM emp\_chris***//右对齐，左边补空格

7.SUBSTR函数:截取字符串

***SUBSTR(str,n,len)***

截取给定的字符串，从指定的位置的字符开始，连续截取指定个数的字符。

**数据库中的下标都是从1开始的，与java不同。**

***n***可以写0，0也是相当于从字符串的第一个字符开始截取。

***n***可以是负数，即从字符串末尾的指定位置的字符开始截取。

***len***可以不写，这时直接截取到字符串末尾。

***len***超过可截取的字符长度时，也是截取到字符串末尾。

***SELECT SUBSTR('THINKING IN JAVA',-4,50) FROM dual***//JAVA

8.INSTR函数:

***INSTR(char1,char2,n,m)***

查看***char2***在***char1***中的位置。

***n***:从第n个字符处开始查找，不写默认值为1.

***m***:从第m次出现的位置开始查找，不写默认值为1.

***SELECT INSTR ('THINKING IN JAVA)','IN',10,2) FROM dual***

若找不着，返回0。

### *PART B. NUMBER*

1.***ROUND(n,m)***

对n进行四舍五入，保留小数点后m位。

***m***可以不写，默认值为0，相当于保留整数位。

***m***可以是负数，保留数字十位以上。

2.**TRUNC(*n*,*m*)**

TRUNC参数的含义和ROUND一致，效果仅仅是不进行四舍五入，只截取数字。

注意m是负数时，整数位只舍不入.

3.***MOD(m,n)***:求余数

返回m除以n的余数(m%n),n为0直接返回m

4.***CEIL(n)/FLOOR(n)***:

参数**n**一般是浮点数，整数返回本身

CEIL:返回大于给定参数的最小整数(向上取整)

FLOOR:返回小于给定参数的最大整数(向下取整)

### *PART C. DATE*

**SYSDATE** 关键字

该关键字对应oracle的一个内部函数，表示一个DATE类型的当前系统时间的值

***SELECT SYSDATE FROM dual***

**TIMESTAMP** 时间戳，精确到纳秒,占用数据库7～11位字节

***SELECT SYSTIMESTAMP FROM dual***

1.***TO\_DATE(DATE,format)***

将一个字符串按照给定日期格式转换为DATE 类型的值

***SELECT TO\_DATE('2008-08-08 20:08:00','YYYY-MM-DD HH24:MI:SS')***

***FROM dual***

若日期格式字符串中出现了非英文与符号的其他字符时，这些字符需要使用

**双引号**括起来，否则会抛出时间格式无法识别的错误。

***SELECT TO\_DATE('2008年08月08日 20时08****分****00****秒****',***

***'YYYY"年"MM"月"DD"日" HH24"时"MI"分"SS"秒"')***

***FROM dual***

**注意:**数据库中的日期是可以比较大小的，**时间晚**的大，**时间早**的小。

可以对一个日期的值进行加法操作，加上一个数字等于加上了对应的天数。

结果为计算后的日期。

两个日期时间也可以进行减法操作，结果为相差的天数。

2.***TO\_CHAR(DATE,format)***

可以将给定的日期按照给定的格式转换为一个字符串

***SELECT TO\_CHAR(SYSDATE,'YYYY-MM-DD HH:MI:SS') FROM dual***

--RR格式，根据系统时间返回一个合理的时间值

***SELECT TO\_CHAR(TO\_DATE('50-02-20','RR-MM-DD'),'YYYY-MM-DD') FROM dual***

3.***LAST\_DAY(DATE)***

返回给定日期所在月的最后一天日期，返回值也是DATE类型

查看当月月底到底哪天

***SELECT LAST\_DAY(SYSDATE) FROM dual***

4.**ADD\_MONTHS(*DATE*,*n*)**

对给定的日期加上给定的月

第二个参数**n**是负数则减去指定月

第二个参数**n**是小数则截取整数部分计算，没有四舍五入

5.**MONTHS\_BETWEEN(*DATE1*,*DATE2*)**

计算两个日期之间相差多少个月。

结果是按照**DATE1-DATE2**差计算的

6.**NEXT\_DAY(*DATE*,*char*)**

返回距离给定日期最近的还没有过的周几是哪天。

若给定日期与要计算的周几正好是一天，则计算结果是下周

第二个参数建议使用数字表示周几

数字(1～7)分别代表周日、周一...周五、周六

***SELECT NEXT\_DAY(SYSDATE,2)*** //下个周一是哪天

***FROM dual***

7.***LEAST/GREATEST(char1,char2......)***

返回给定参数中最小的/最大的，参数多少个都可以。

**注意**：这里的参数可以是任何类型，但是参数类型必须一致。

若不一致则在第二个参数之后隐式转换成第一个参数类型进行比较，不能转换则报错。

NUMBER和DATE型按大小比较。

CHAR/VARCHAR/VARCHAR2按首字母a-z顺序比较

8.***EXTRACT(date FROM datetime)***

从datetime中提取参数date指定的年月日

从TIMESTAMP参数中可以提取到秒

***SELECT EXTRACT(YEAR FROM SYSDATE)***

***FROM dual***

### *PART ANOTHER. NULL*

NULL,空值

任何数据类型在数据库中默认都是NULL，表示不存在

NOT NULL 非空约束，表示该类型值不能为空

设置或者panduan 条件中，不能 =NULL，这点与java不同，

只能写成 **IS NULL/IS NOT NULL**

**NULL与任何数字的运算结果还是NULL**

**空值函数**:

1.***NVL(arg1,arg2)***

判断arg1是否是NULL，若是，该函数返回agr2的值。

否则则返回arg1，所以该函数的作用是**将NULL值替换为非NULL值**

arg1与arg2可以是任何类型，但类型必须一致。

2.***NVL2(arg1,arg2,arg3)***

当arg1不为NULL时，返回arg2，否则返回arg3

参数类型同NVL。

## **SQL API(for SELECT)**

### *PART A 基础查询语句: SELECT*

1.**别名**: ***AS char***

当SELECT语句中要查询的内容不是单纯的字符段，而是一个函数或者表达式时，

查询出来的结果集中对应的该字段的字段名就是这个函数或者表达式，这样不利于读取，

为此我们会为这样的列加一个**别名**，使得查询出来的该字段的名字为这个**别名**，

增加可读性，甚至在**子查询**中对这样的情况还要求**必须写别名**。

别名可以在字段之后使用空格分隔开，然后定义或者在字段名之后使用'AS'的后面定义。

**通常可以不用写AS**

别名不区分大小写，且不能使用空格。

若有需要在别名中**区分大小写或者加空格**，那么别名需要被**双引号括**起来。

SELECT ename,sal\*12 sal你

FROM emp\_chris

2.**WHERE** **条件判断**

**WHERE子句**，与DML中使用一致，WHERE用来限定条件。

在SELECT中使用，可以将只满足WHERE要求的记录查询出来。

WHERE子句中可以使用函数或者表达式的结果作为过滤条件

3.**条件判断符号**:**>,<,>=,<=,!=,<>,=**

<>表示不等于，与！=等价，数据库中**建议使用<>**

4.***AND/OR*** **关键字** 逻辑操作符 **与/或**

**AND优先级高于OR**，所以要提高OR的优先级可以使用括号

5.**LIKE** **关键字**(模糊查询)

用于模糊查询字符串内容，支持两个通配符

**%** 零到多个个任意字符

**\_** 一个任意字符

6.***IN(list)/NOT IN(list)***

**IN(*list*)**:判断指定内容的值是否等于列表中的任意一项

**NOT IN(*list*)**:判断指定内容的值不等于列表中的任意一项。

参数list一般为SELECT查询结果集

IN常被用作判断**指定值是否与子查询的结果内容一致**，这是OR不可替代的。

7.***BETWEEN...AND...***

判断指定内容的值在一个区间范围内。两个传入参数**小的在前，大的在后**

8.***IS NULL/IS NOT NULL***

判断是否空值，不能使用符号 =

9.***ANY(list)/ALL(list)***

ANY与ALL 判定指定内容>,>=,<,<=一个列表内容时使用。

**>ANY**:大于列表中其中之一，所以大于最小的即可

**>ALL**:大于列表中所有，所以必须大于最大的

**<ANY**:小于最大的

**<ALL**:小于最小的

ANY和ALL常用于判断指定值与子查询的结果内容之间的大小关系

10.**DISTINCT** **关键字**

adj.独特的，明显的，有区别的

去除重复行，DISTINCT 后面的字段值相同的部分将在查询结果集中被去除。

**DISTINCT 关键字必须紧跟在 SELECT 关键字之后，两者之间除了空格不能有其他字段。**

DISTINCT 后面可以有多个字段，若是多个字段，那么则是这些字段值的组合没有重复的。

查看公司有哪些职位

***SELECT DISTINCT job FROM emp***

***SELECT DISTINCT job,deptno FROM emp***  --job和deptno组合没有重复

11.**ORDER BY** **子句**

用来排序查询结果集，**ORDER BY 必须写在 SELECT 语句的最后面。**

ORDER BY 可以根据后面指定的字段的值进行升序或者降序排列。

**升序使用 ASC，降序使用 DESC。**但升序的 ASC**通常不写**，**默认为升序**。

多个字段之间排序有**优先级关系**:先按照第一个字段的排序方式进行排列，然后在第一个

字段值排序后的相同值区间，再按照第二个字段的排序方式进行排列，以此类推。

查看公司员工的工资排名:

***SELECT ename,deptno,sal FROM emp\_chris***

***ORDER BY* *deptno* DESC,*sal* DESC** –先按septno排序，其值区间按sal排序

### *PART B 聚合函数*

**聚合函数**又叫做多行函数，分组函数

作用：将若干记录进行**统计**，然后产生一个结果。

1.**MAX()/MIN()**

统计指定内容中所有参与统计记录中的最大值/最小值。

参数可以是任何类型。

2.**AVG()/SUM()**

统计平均值和总和。

只能操作数字类型。

聚合函数统计时**忽略NULL值。**

若希望NULL值的记录参与统计，可以使用NVL将这些记录转换为非NULL值再参与统计

3.**COUNT()**

该函数统计的是**非空的记录**有多少条，而不关系具体每条记录的值为多少

查看公司总共有多少人（整张表有多少条记录）

***SELECT COUNT(\*) count FROM emp\_chris***

4.**GROUP BY** 子句

**GROUP BY** 关键字后面可以跟若干字段，作用是根据给定的字段值相同的记录看作一组，

若是多个字段，则是这些字段值的组合的记录看作一组

**SELECT语句中若使用了组函数，那么不在组函数中的其他字段必须出现在 GROUP BY 子句中**！

查看每个部门的最高工资、最低工资、平均工资、以及工资总和：

***SELECT***

***MAX(sal) max,MIN(sal) min,***

***TRUNC(AVG(sal)) avg,SUM(sal) SUM,deptno***

***FROM***

***emp***

***GROUP BY***

***deptno***;

5.**HAVING** 子句

查看平均工资高于2000的部门的最高工资和最低工资:

**WHERE 子句中不允许使用分组函数。**

**原因**：过滤时机不同。

若我们向判断平均工资大于2000，这说明我们已经将表里的数据查询出来，并先进行统计，在统计的结果基础上再进行判断。而 WHERE 的过滤时机是在查询表的过程中进行的，表中的数据查询出哪条记录是由 WHERE 的判断结果为准，所以这里 WHERE 已经在实际需要判断的地方之前就执行完毕了。

**HAVING 子句，该子句必须跟在 GROUP BY 子句之后，不能单独定义。**

作用是在分组后做统计，并对分组结果进行进一步限制。

***SELECT MAX(sal),MIN(sal),deptno***

***FROM emp***

***GROUP BY deptno***

***HAVING AVG(sal)>2000***

**注意：查询语句的执行顺序(重要)**

**1.FROM子句:执行顺序为从后往前、从右到左。**

**数据较少的表尽量放在后面**

**2.WHERE子句:执行顺序为自下而上，从右到左**

**将能过滤掉最大数量记录的条件写在最右**

**3.GROUP BY字句:执行顺序从左往右分组**

**在使用前最好能用WHERE过滤掉不需要的记录。**

**4.HAVING子句:消耗资源**

**尽量避免使用。HAVING子句会在检索出所有记录之后才对记录进行过滤。**

**需要排序等操作。**

**5.SELECT子句:少用\*号，尽量取字段名**

**ORACLE在解析的过程中会通过查询数据字典将\*逐一转换为字段名，消耗时间。**

**DISTINCT 必须紧跟在SELECT后，中间除了空格没有其他字符**

**6.ORDER BY:执行顺序从左到右，消耗资源。**

**该子句必须放在SELECT语句的最后。**

### *\*\*\*PART C 关联查询（重点）\*\*\**

1.**联合多张表进行查询**。

需要注意的是，**N**张表关联查询至少要有**N-1**个连接条件。

连接条件的目的在于让数据库知道表中记录与另一张表中的记录是如何对应上的。

查询SALES部门的员工名字:(传统方式86)

***SELECT emp.ename***

***FROM emp,dept***

***WHERE emp.deptno = dept.deptno*** --连接条件必须有

***AND dept.dname = 'SALES'*** --过滤条件根据需要写上

查询每个员工的名字以及所在部门的名称以及所在地

表名也可以加别名 使用方式同AS

**SELECT *e.ename,d.dname,d.loc***

**FROM *emp e,dept d***

**WHERE *e.deptno* = *d.deptno*** --连接条件必须有

**笛卡尔积**

表A的记录x\*表B的记录Y，即所有记录全部查出

不添加连接条件，或者连接条件无效，都会产生笛卡尔积。

笛卡尔积会产生大量无用的关联，对于系统资源产生严重破坏，

严重时可导致系统瘫痪。

2.**内连接**(**[INNER]JOIN...ON...**):(92标准)

***SELECT e.ename,d.dname,d.loc***

***FROM emp e JOIN dept d***

***ON e.deptno = d.deptno***

***WHERE*** ***ename*** = ***'SCOTT'***

这样写的好处在于连接条件定义在ON子句中，WHERE里只定义过滤条件即可，条理更**清晰**。

3.**外连接**:

除了满足连

接条件的数据查询出来以外，还可以将不满足连接条件的数据也都查询出来。

外连接分为:**左外连接**，**右外连接**，**全外连接**。

**左外连接**(**LEFT [OUTER] JOIN...ON...**):以左边的表为主，左边表中的所有记录都查询出来。

当有不满足连接条件的记录时，来自右边表中的字段全部为NULL。

**右外连接**(**RIGHT [OUTER]JOIN...ON...**):右表为主，左边补NULL。

**全外连接**(**FULL [OUTER] JOIN...ON...**):那边不满足条件，哪边补NULL。

**注意**:在使用外连接时，过滤条件如果也写在ON中，不满足过滤条件的记录也能显示出来

如果写在WHERE中，则在ON执行完之后才过滤，不满足过滤条件的记录不会被显示.

内连接没有这种情况。

查看每个员工的名字，部门号，部门名称，没有部门，也要将员工信息列出来

***SELECT e.ename,e.deptno,d.dname***

***FROM emp e LEFT OUTER JOIN dept d***

***ON e.deptno = d.deptno***

***SELECT e.ename,e.deptno,d.dname***

***FROM emp e RIGHT OUTER JOIN dept d***

***ON e.deptno = d.deptno***

***SELECT e.ename,e.deptno,d.dname***

***FROM emp e FULL OUTER JOIN dept d***

***ON e.deptno = d.deptno***

4.**自连接:**

自己表中的一条数据对应自己表中的多条数据。

自连接为了解决数据一样，但是数据之间又存在父子关系，

而且层级关系中有多少层不确定的情况。

**自连接**简单的说就是解决了**树状结构的数据关系**。

例如:员工表、都是员工、但是员工之间有存在上下级关系

商品类别表，都是类别，但是也存在上下级关系

查看员工以及其上司的名字:

***SELECT es.ename "员工名字",er.ename "上司名字"***

***FROM emp es LEFT JOIN emp er***

***ON es.mgr = er.empno***

补充:关于连接查询中的字段名问题，一般要查询的字段名都加上**"**表名.**"**，表示查询于哪个表的字段。如果两个表的字段没有相同值，那么可以直接写字段名，只需要将重复的那个字段标上**"**表名.**"**即可。

### *PART D 子查询*

子查询的作用是为了其他SQL语句提供数据，以便其他SQL可以根据该数据进行操作。

子查询可以嵌套在DQL，DML，DDL中使用。

最常见的地方是在DQL中使用。

谁的工资比CLARK的工资高

***SELECT ename,sal***

***FROM emp\_chris***

***WHERE sal > (SELECT sal***

***FROM emp\_chris***

***WHERE ename = 'CLARK')***;

查看与JONES相同部门的员工

***SELECT ename,deptno***

***FROM emp\_chris***

***WHERE deptno = (SELECT deptno***

***FROM emp\_chris***

***WHERE ename = 'JONES')***

***AND ename <> 'JONES'***

1.子查询可以用于DDL语句中**快速建表**

***CREATE TABLE myemp\_chris***

***AS***

***SELECT e.empno,e.ename,e.sal,e.job,e.deptno,d.dname,d.loc***

--新表的字段来自于SELECT后写的字段，如果用别名，新字段为别名

***FROM emp\_chris e JOIN dept\_chris d***

***ON e.deptno = d.deptno***

2.子查询用于DML语句

***DELETE emp\_chris***

***WHERE deptno = (SELECT deptno***

***FROM emp\_chris***

***WHERE ename = 'CLARK')***;

3.1子查询是**单行单列**的情况:

由于结果只有一个，所以可以把它当作一个值看待

提高CLARK所在部门所有员工的工资20%

***UPDATE emp\_chris***

***SET sal = sal\*1.2***

***WHERE deptno = (SELECT deptno***

***FROM emp\_chris***

***WHERE ename = 'CLARK')***

查找薪水比公司平均薪水高的员工

***SELECT ename,sal***

***FROM emp\_chris***

***WHERE sal > (SELECT AVG(sal) FROM emp\_chris)***

3.2子查询是**多行单列**的情况:

由于一般结果是一个字段列表，故可以使用IN、ANY、ALL之类的逻辑判断

查看与SALESMAN不同职位但是相同部门的其他员工

***SELECT ename,job,deptno***

***FROM emp\_chris***

***WHERE deptno IN (SELECT deptno***

***FROM emp\_chris***

***WHERE job = 'SALESMAN')***

***AND job <> 'SALESMAN'***

查看谁的工资比20号部门随便一个员工工资高

***SELECT ename,sal,deptno***

***FROM emp\_chris***

***WHERE sal > ANY (SELECT sal***

***FROM emp\_chris***

***WHERE deptno = 20)***

3.3.子查询在 FROM 语句中

**多列子查询**，通常当作一张表看待，定义在外层查询的FROM子句中使用

查出薪水比本部门平均薪水高的员工信息

***SELECT e.ename,e.sal,e.deptno***

***FROM emp\_chris e JOIN (SELECT AVG(sal) avgsal,deptno***

***FROM emp\_chris***

***GROUP BY deptno) d***

***ON e.deptno = d.deptno AND e.sal > d.avgsal --两种都可以***

***--WHERE e.sal > d.avgsal***

4. **EXISTS** 关键字

用在 WHERE 子句中，其后要跟一个子查询，

**EXISTS**是否返回true取决于后面的子查询是否能查询到数据。

**NOT EXISTS** 作用与 EXISTS 相反

查看有员工的部门信息:

***SELECT deptno,dname***

***FROM dept\_chris d***

***WHERE******EXISTS ( SELECT \****

***FROM emp\_chris e***

***WHERE d.deptno = e.deptno)***

5.子查询在 HAVING 语句中

查询所有部门最低薪水大于30号部门最低薪水

***SELECT MIN(sal),deptno***

***FROM emp\_chris***

***GROUP BY deptno***

***HAVING MIN(sal) > (SELECT MIN(sal) FROM emp\_chris WHERE deptno = 30);***

7.子查询在 SELECT 子句中，是**外连接的一种写法**

***SELECT e.ename,e.sal,(SELECT d.dname***

***FROM dept\_chris d***

***WHERE e.deptno = d.deptno) dname***

***FROM emp\_chris e***

### *PART E 分页查询:*

分页的目的是，当数量过大时，若一次性全部查询出来，会对系统资源造成不必要的开销，

而且处理速度会变慢，为此我们可以分段式的将数据一点一点的取出。

分页的语句SQL没有定义，所以不同数据库分页语句不一样。

1.**ROWNUM**

**ROWNUM** 伪列，并不是表中的一个真实字段，但是在任何表的查询中都可以使用该列。

该列的值为结果集的每一条记录编一条行号。

只要可以查询出一条记录，该字段的值就会+1。行号从1开始。

由于只有从表中查询出一条数据，ROWNUM 才会使用1进行编号，然后自动涨为2.

所以，**不要在第一次查询数据使用 ROWNUM 编号的时候在 WHERE 中做大于1以上**

**的数字判断，否则不会查出任何数据**！

ROWNUM一开始不大于1，所以WHERE不满足条件，不满足WHERE条件就查不出数据，

查询不出数据ROWNUM就不会涨，不涨就不大于1.....

***SELECT ROWNUM,ename,sal,job,deptno***

***FROM emp\_chris***

***WHERE ROWNUM>1*** --查不到任何信息

要正确使用ROWNUM功能可以先查询，再把查询结果作为一个表，再来进行其他操作。

注意ROWNUM不能直接调用，要起一个别名。

***SELECT \****

***FROM (SELECT ROWNUM rn,ename,sal,deptno,job***

***FROM emp\_chris) d***

***WHERE d.rn BETWEEN 6 AND 10***

2.分页排序

分页的同时，数据还有排序需求

查看公司中工资排名的6-10位员工信息:

由于使用ROWNUM编号是在查询的过程中完成的，而排序是在查询出数据以后进行的，

所以这就导致排序之前的编号失去意义了。

***SELECT \****

***FROM （SELECT ROWNUM rn,ename,sal,job,deptno***

***FROM emp\_chris***

***ORDER BY sal DESC)***

***WHERE rn BETWEEN 6 AND 10*** --排序无效

所以若有排序需求，分页需要三次查询才能实现。

1)排序;2)编号;3)取范围

***SELECT \****

***FROM (SELECT ROWNUM rn,t.\****

***FROM (SELECT ename,sal,job,deptno***

***FROM emp\_chris***

***ORDER BY sal DESC***

***)t***

***)***

***WHERE rn BETWEEN 6 AND 10***

分页时，BETWEEN中的两个页数公式:

page：页数

pagesize：每页显示的条数

根据上面两个值计算

***start:(page-1)\*pagesize+1***

***end:page\*pagesize***

### *PART F DECODE函数*

1.***DECODE(a,b1,c1,b2,c2,b3,c3,.....,default)***

当a=b1,返回c1；

当a=b2,返回c2；

......

当a不等于任何值，则返回default，没有default则返回NULL

***SELECT ename,job,sal,***

***DECODE(job,***

***'MANAGER',sal\*1.2,***

***'ANALYST',sal\*1.1,***

***'SALESMAN',sal\*1.05,***

***sal)bouns***

***FROM emp\_chris***

2.**CASE**函数，可以实现DECODE类似的功能

***SELECT ename,job,sal,***

***CASE job WHEN 'MANAGER' THEN sal\*1.2***

***WHEN 'ANALYST' THEN sal\*1.1***

***WHEN 'SALESMAN' THEN sal\*1.05***

***ELSE sal END***

***bouns***

***FROM emp\_chris***

3.**分组查询中的应用**

GROUP BY 中使用DECODE函数，可以将字段值不同的记录看作一组，

只要使用DECODE将需要划为一组的不同值转变为相同值即可。

***SELECT COUNT(\*) vip\_cut,***

***DECODE(job,'MANAGER','VIP','ANALYST','VIP','OPERATIONS') vip***

***FROM emp\_chris***

***GROUP BY DECODE(job,'MANAGER','VIP','ANALYST','VIP','OPERATIONS')***

ORDER BY 排序也可以用DECODE函数,NULL在排序中认为是无限大

***SELECT deptno,dname,loc***

***FROM dept\_chris***

***ORDER BY DECODE(dname,'OPERATIONS',1,'ACCOUNTING',2,'SALES',3)***

### *PART G 排序函数*

1.**ROW\_NUMBER**函数，可以根据指定的字段分组，再根据指定的字段排序，

然后生成组内连续且唯一的数字。

***ROW\_NUMBER() OVER(PARTITION BY part1 ORDER BY part2)***

按part1分组，part2排序，也可以不分组只排序。

查看每个部门的工资排名情况

***SELECT ename,sal,deptno,***

***ROW\_NUMBER()/RANK/DENSE\_RANK OVER(PARTITION BY deptno***

***ORDER BY sal DESC) bouns***

***FROM emp\_chris***

2.**RANK**函数，功能和**ROW\_NUMBER**一样，只是生成组内不连续且不唯一的数字。

3.**DENSE\_RANK** 函数，功能和**ROW\_NUMBER**一样，只是生成组内连续且不唯一的数字。

### *PART H 集合操作*

对查询结果集的操作。

对于操作的两个SELECT结果集要求：

**字段的个数和数据类型必须匹配**。

1.**UNION/UNION ALL** :**并集/全并集**

显示满足要求的结果集。

前者要求符合两个结果集其一即可，重复的记录去除；

后者要求满足两个结果集其一即可，不去除重复记录。

2.**INTERSECT**：交集

显示符合两个结果集要求的记录。

3.**MINUS**：差集

显示符合前一个结果集但不符合后一个结果集的记录。

***SELECT ename,job,sal FROM emp\_chris***

***WHERE job = 'MANAGER'***

***UNION*/*INTERSECT*/*MINUS***

***SELECT ename,job,sal FROM emp\_chris***

***WHERE sal > 2500***

### *PART I 高级分组函数*

1.**ROLLUP()**

**ROLLUP** **用在 GROUP BY 子句中**，ROLLUP函数中可以传入若干参数。

作用是参数逐个递减，每次进行一回分组并统计结果，然后将这些结果并在一起显示。

**GROUP BY *ROLLUP*(*a,b,c*)**

相当于：

**...GROUP BY (*a,b,c*)**

**UNION**

**...GROUP BY (*a,b*)**

**UNION**

**...GROUP BY *a***

**UNION**

**...**

2.**CUBE()**

会将给定字段的每一种组合都进行一次分组，然后将所有结果并在一起显示。

分组的次数是2的参数个数幂。使用方法同**ROLLUP**。

3.**GROUPING SETS()**

可以按照给定的分组方式进行分组，然后结果并在一起显示。

每个参数表示一种分组方式，参数可以使用括号将要分组的字段括起来表示一种。

使用方法同**ROLLUP**。

### *PART J 递归查询函数(for PostgreSQL)*

**postgreSql**提供**with recursive**函数做递归查询

例如：查询当前部门树的当前节点以及以下所有节点

**WITH RECURSIVE dept AS (**

**SELECT \***

**FROM tb\_sys\_dept**

**WHERE dept\_id = 2**

**UNION ALL**

**SELECT tb\_sys\_dept.\***

**FROM tb\_sys\_dept,dept**

**WHERE tb\_sys\_dept.parent\_dept\_id = dept.dept\_id**

**)**

**SELECT dept\_id**

**FROM dept**

**WHERE dept.delete\_flag = 0**

查询部门树的当前节点以及以上所有节点

**WITH RECURSIVE dept AS (**

**SELECT \***

**FROM tb\_sys\_dept**

**WHERE dept\_id = 2**

**UNION ALL**

**SELECT tb\_sys\_dept.\***

**FROM tb\_sys\_dept,dept**

**WHERE tb\_sys\_dept.dept\_id = dept.parent\_dept\_id**

**)**

**SELECT dept\_id**

**FROM dept**

**WHERE dept.delete\_flag = 0 AND dept.parent\_dept\_id = 0**

### *PART K CAST函数*

**Cast函数**

vi.计算，把几个数字加起来

语法：**CAST** (expression **AS** data\_type)

**参数说明：**  
**expression**：任何有效的SQServer表达式。  
**AS**：用于分隔两个参数，在AS之前的是要处理的数据，在AS之后是要转换的数据类型。  
**data\_type**：目标系统所提供的数据类型，包括bigint和sql\_variant，不能使用用户定义的数据类型。

**使用CAST函数进行数据类型转换时，在下列情况下能够被接受：**  
（1）两个表达式的数据类型完全相同。  
（2）两个表达式可隐性转换。  
（3）必须显式转换数据类型。

**如果试图进行不可能的转换（例如，将含有字母的 char 表达式转换为 int 类型），SQServer 将显示一条错误信息。**  
**如果转换时没有指定数据类型的长度，则SQServer自动提供长度为30**

例如：

SELECT CAST('12' AS int) 返回整数类型12

SELECT CAST('12.5' AS int) 则报错

**SELECT** \*

**FROM** tb\_sys\_user

**WHERE** **CAST**(tb\_sys\_user.department **AS** INTEGER) = 1

## **VIEW、SEQUENCE、INDEX、CONSTRAINT**

### *视图(VIEW)*

--和拟表(dual)不同

也称为**虚表**，虚拟的表。

视图在SQL语句中体现的地方和表一致，但是它不是数据库中真实存在的表，

而是一个 SELECT 语句的查询结果集。所以视图中有哪些字段，对应的都有哪些记录，

完全取决于 SELECT 查询的结果。

**取名**: **v\_表名\_要查询的数据**

***CREATE VIEW v\_emp\_chris\_10***

***AS***

***SELECT empno,ename,sal,deptno***

***FROM emp\_chris***

***WHERE deptno = 10;***

当我们通过SELECT语句查询视图时，数据库会先使用该视图对应的SELECT语句将数据查询出来。

然后将这个结果集当成一张表，再进行我们的SELECT语句，将要查询的数据查询出来。

所以视图就是在FROM中使用子查询。

DESC v\_emp\_chris\_10

SELECT \* FROM v\_emp\_chris\_10

视图中对应的子查询中，查询的字段若含有表达式函数，那么必须要添加别名。

单纯的字段也可以添加别名。这样创建出来的视图对应的字段名就是别名。

**CREATE *OR REPLACE* VIEW name**，可以达到修改视图的效果，如果视图不存在则创建。

**简单视图**:基于**单表**建立，不包含任何函数运算、表达式或分组函数，

**可以使用DML语句，对视图进行增删改操作**。

对视图进行DML操作就是对视图数据来源的基表进行的。

**视图的DML规则**：

1.向视图插入(INSERT)数据，数据实际上是插入视图来源的基表中。

而且，新增的记录中只有视图看得到的字段有值，其他字段会插入默认值。

若基表中有NOT NULL 字段且视图没有该字段，则不能插入。

视图有可能插入一条视图自己看不见的数据，这会到基表产生数据污染！

2.更新(UPDATE)视图数据，就是更新基表中对应的数据。

与INSERT一样，更新视图数据有可能导致视图无法再次查询到该数据。

3.删除(DELETE)视图数据，就是删除表中对应数据。

但是删除只能删除视图看得到的数据。

并且基表的数据也被删除了。

**复杂视图**:包含函数运算、表达式或分组函数，**不能增删改(DML语句)**。

其中SELECT基于多个表的称之为连接视图。

创建一个复杂视图:

CREATE VIEW v\_emp\_chris\_salary

AS

SELECT d.dname,AVG(e.sal) avg\_sal,

SUM(e.sal) sum\_sal,

MAX(e.sal) max\_sal,

MIN(e.sal) min\_sal

FROM emp\_chris e JOIN dept\_chris d

ON e.deptno = d.deptno

GROUP BY d.dname

**删除视图不会影响基表的记录，但是删除视图数据会把基表的数据也删除**

DROP VIEW v\_emp\_chris\_salary

DROP VIEW v\_emp\_chris\_10

**视图的作用：简化查询复杂度，限制数据访问**

**视图创建的语法**:

**CREATE [OR REPLACE] VIEW v\_table\_data**

**AS**

**...**

**[WITH CHECK OPTION]/[WITH READ ONLY]**

当视图增加 OR REPLACE 时，可以达到修改视图的效果，如果视图不存在则创建。

当视图增加 WITH CHECK OPTION 后：

(1).对视图进行插入操作(INSERT):必须插入的数据视图可见。

(2).对视图进行更新操作(UPDATE):必须更新后视图对其可见。

当视图增加 WITH READ ONLY 后，视图变成只读，不可进行DML操作（增删改）。

**数据字典**:***user\_objects***,***user\_views***,***user\_tables***,***user\_sequence***

所有用户创建对象/所有用户创建视图/所有用户创建表/所有用户创建序列

### *序列(SEQUENCE)*

根据设定的方式，生成一组数字。

通常序列被用来为**表中的主键字段提供值(\*)**。

主键:一般是表的第一个字段，要求非空约束且值唯一

***CREATE SEQUENCE [schema.] sequence\_name***

[***START WITH***] 起始值--默认1

[***INCREMENT BY***] 步长(进)值--默认1

[***MAXVALUE***] 最大值

[***NOMAXVALUE***] 没有最大值

[***MINVALUE***] 最小值

[***NOMINVALUE***] 没有最小值

[***CYCLE***] 循环

[***NOCYCLE***] 不可循环

[***CACHE***] 缓存--默认20 用户取值时系统一次性生成的序列数字个数，使用完后再去维护数字字典，生成下一组数字

[***NOCACHE***] 没有缓存

**序列**(**SEQUENCE**)有两个伪列:

***NEXTVAL***:获取下一个序列值/向序列要下一个数字

若序列还没有生成过数字，则第一次获取的数字为START WITH指定的数字。

否则则是之前生成过的数字加上步长得到的。

**一旦获取了下一个数字，就无法再通过序列获取之前生成的数字了！**

***CURRVAL***:获取当前的序列值/向序列要最后一次生成的数字

该伪列可以调用若干次，每次返回的数字一样，并不会导致序列获取下一个值。

除非使用NEXTVAL。需要注意，序列刚创建完毕后，至少要调用一次NEXTVAL，

生成数字后才可以使用该伪列。

### *索引(INDEX)*

提高表的**查询效率**。

索引的算法和应用完全由数据库自行决定。

给单个字段添加索引:

***CREATE INDEX index\_emp\_chris ON emp\_chris(ename)***;

给多个字段添加索引称为多列索引:

***CREATE INDEX index\_emp\_chris\_job\_sal ON emp\_chris(job,sal)***;

给函数添加索引:

***CREATE INDEX index\_emp\_chris\_upper ON emp\_chris(UPPER(ename))***;

重建索引:

***ALTER INDEX index\_name*** ***REBUILD***

***ALTER INDEX index\_emp\_chris*** ***REBUILD***

删除索引:

***DROP INDEX index\_emp\_chris\_upper***

合理建立索引：

1.为经常出现在WHERE子句中的列建立索引；

2.为经常出现在ORDER BY、DISTINCT 后面的字段建立索引；

如果建立的是复合索引，索引后面的字段顺序要与这些关键字后面的字段顺序一致；

3.为经常作为表的连接条件上的列建立索引。

不要建立索引:

1.不要在经常做DML操作的表上建立索引。

2.不要在小表上建立索引；

其他要求：

1.限制表上的索引数目，索引并不是越多越好；

2.删除很少被使用的、不合理的索引

### *约束(CONSTRAINT)*

约束条件、完整性约束条件

1.NOT NULL:非空约束。列级约束：定义列的同时使用。

要求该约束的字段值不能为空。

在建表时使用，也可以在建表后的修改表的属性时使用。

完整写法： ***CONSTRAINT*** 约束名 ***NOT NULL***

缩写：***NOT NULL***

CREATE TABLE employees\_chris(

eid NUMBER(6),

name VARCHAR(30) ***NOT NULL***,

salary NUMBER(7,2),

hiredate DATE CONSTRAINT employees\_chris\_hiredate\_nn NOT NULL

);

ALTER TABLE employees\_chris

MODIFY(

eid NUMBER(6) ***CONSTRAINT*** employees\_chris\_eid\_nn ***NOT NULL***

);

当使用非空约束的时候要取消约束，直接加个NULL，不要 DEFAULT NULL

ALTER TABLE employees\_chris

MODIFY(

eid NUMBER(6) NULL

);

2.UNIQUE：唯一性约束。**既是一个列级约束，也是一个表级约束**。

保证约束的字段及其组合的值不能重复，保证约束的列的值不能重复。

NULL除外。

它可以在建表的同时建立，也可以在建表之后建立。

**在唯一性约束建立之后，系统会自动生成唯一索引。**

CREATE TABLE employee\_chris(

eid NUMBER(6) ***UNIQUE***,

name VARCHAR(30),

salary NUMBER(7,2),

email VARCHAR2(30),

hiredate DATE,

***CONSTRAINT*** employee\_email\_uk ***UNIQUE***(email)

);

ALTER TABLE employee\_chris

ADD ***CONSTRAINT*** employee\_chris\_name\_uk ***UNIQUE***(name)

3.**PRIMARY KEY** 主键约束：

主键字段可以是单字段或者多字段组合，约束条件是既不能重复也不能为空。

一个表上只能建立一个主键。

**主键约束在建立时同时拥有唯一性索引。**

(1).**主键应该是对于系统无意义的数据**；

(2).永远不要更新主键，让主键除了**唯一标识**一行以外，再无其他用途；

(3).主键不应包含动态数据，如时间戳(TIMESTAMP)

(4).主键最好**自动生成**(**SEQUENCE用来添加主键值**)，不要人为干预，以免带上除了唯一

标识一行以外的意义;

(5).主键尽量建立在单列上。

CREATE TABLE employee\_chris(

eid NUMBER(6) **PRIMARY KEY**,

name VARCHAR(30),

salary NUMBER(7,2),

email VARCHAR2(30),

hiredate DATE

);

ALTER TABLE employee\_chris

ADD **CONSTRAINT** employees\_chris\_eid\_pk **PRIMARY KEY**(eid);

INSERT INTO employee\_chris

(eid,name)

VALUES

(1,'DMB')

4.**FOREIGN KEY** 外键约束：

定义在两个表的字段或者表的两个字段上，用于保证两个字段的关系。

**FOREIGN KEY** 外键 **REFERENCE** 主键

定义在辅表上

5.**CHECK**(条件) 检查约束：

强制被修饰的字段必须满足CHECK中的定义的条件

ALTER TABLE employee\_chris

ADD **CONSTRAINT** employee\_chris\_salary\_ck

**CHECK**(salary>2000)

删除约束条件：

**ALTER TABLE 表名**

**DROP CONSTRAINT 约束名**

## JDBC

### **JDBC的标准流程**

概述:

**JDBC**是JAVA提供的连接数据库的解决方案，其提供了一套用于连接数据库的通用接口。

不同的数据库提供商负责编写实现类来满足java连接该数据库。

数据库提供商会将这一组实现类打包成一个jar包，我们也称这个jar包为该数据库的驱动包。

JDBC每个接口都定义了相关的抽象方法，无论连接哪个数据库，过程都是一样的，并且我们只需要面向JDBC提供的接口即可，

无需了解不同数据库驱动包中的实现类。

**连接数据库的标准流程：**

**1.注册驱动 Class.forName()**

**2.加载驱动建立数据库连接 DriveManager**

**3.通过连接创建执行SQL语句的对象 Connection**

**4.使用语句对象执行SQL Statement/PreparedStatement**

**5.若是查询语句，可以获取结果集 ResultSet**

**6.使用完毕后关闭连接释放资源 close()**

***java.sql.Connection*** 是一个接口，表示与特定数据库的连接。

不同数据库驱动包中有该接口的实现了类。

**1.注册驱动**

将数据库厂商提供的jar包中JDBC要求的驱动类导入环境中以便JVM可以执行它。

由于不同厂商的jar包的目录结构和实现类名字不同，这里的字符串内容不一样。

这里使用的是Oracle数据库的字符串。

执行这句话，可能出现:ClassNotFoundException

常见的两种情况:

(1).字符串内容拼写有误；

(2).驱动jar包没有导入环境变量中(build path)

Class.forName("oracle.jdbc.driver.OracleDriver");

**2:通过DriverManager加载驱动并与数据库建立连接**，不同的数据库连接地址格式不一样。

调用静态方法getConnection()时需要传入3个参数:

1).数据库url

不同数据库地址不一样，这里使用oracle地址:

jdbc:oracle:thin:@host:port:sid

2).数据库用户名

3).数据库密码

Connection conn = DriverManager.getConnection("jdbc:oracle:thin:@host:port:sid",username,password);

\*

JDBC默认自动提交事务，在例如转账业务中，我们会进行两条UPDATE语句。

只有两个更改都成立才应该认为转账成功，否则业务应该失败！

这时候我们需要自行控制事务。

Connection提供了一个方法:

void setAutoCommit(boolean auto)

当参数传入false时，就关闭了自动提交事务

在生成Statement实例之前就要设置好。

\*

**3.创建用于执行SQL语句的Statement实例**，同样的Statement也是接口，驱动包中会有对应的实现类。

不过我们不需要关心，只当它是Statement用即可。

Statement state = conn.createStatement();

**4~5 执行SQL语句**

Statement：

Statement针对不同的SQL语句提供了不同的方法，常用的有:

boolean execute(String sql)

什么语句都执行，但DML,DQL由不同的方法执行，所以该方法普遍用来执行DDL。

返回值若为TRUE说明执行该SQL后有结果集。

由于用它执行DDL，所以返回值应该是FALSE。

若SQL语句执行有误会抛出异常。

int executeUpdate(String sql)

执行DML语句，返回值为执行完毕后，影响了数据库多少条数据

JDBC默认是自动提交事务的，意思就是说：每当我们执行一条DML操作，就自动commit！

ResultSet executeQuery(String sql)

执行DQL语句，返回值为查询的结果集

执行查询语句使用executeQuery方法，会返回一个查询结果集

ResultSet rs = state.executeQuery(sql);

遍历结果集的方式:

结果集默认指向第一条记录之上。

ResultSet遍历结果集的常用方法：

1).boolean next()

该方法的作用是让结果集表示下一条记录。

若结果集有下一条记录，则返回值为true，并表示该记录，否则则返回false

2).字段名 get字段名(String colName)

根据字段名返回当前记录该字段的值，不同的字段类型要调用不同类型的方法。

varchar类数据类型要调用:

String getString(String colName)

注意：

使用Statement执行SQL语句时，若SQL语句中含有动态数据，那么这种拼接后的SQL语句在执行时，

会存在SQL注入式攻击的风险。

原因:动态数据中若包含SQL敏感字符，那么 拼接后的SQL语句可能语法含义就会改变。

这样再通过Statement发送给数据库后，数据库看到的就是被改变语法含义的SQL而执行了。

Statement将SQL语句发送给数据库后，数据库首先要理解该SQL语句的含义，然后指定一个计划，这个过程是比较耗时耗资源的，当指定完毕后，数据库就会执行这个计划然后将结果返回。所以，若使用Statement执行若干条语义相同但是数据不同的SQL语句时，数据库会认为它们是不同的SQL语句，而为每一条SQL语句都制定计划，这对数据库来讲开销是非常大的！

再加上Statement还存在SQL注入攻击的风险，所以得出结论为：

Statement只适合执行静态SQL语句(SQL语句中不存在动态数据)。

缓存原理:使用Statemaent本身作为key并将执行计划存入Statement对应的缓存中

对于曾经使用过的Statement，再执行计划时将重用，减少开销。

PreparedStatement：

当SQL语句中含有动态信息时，由于Statement不适合做，所以JDBC提供了另一种专门来执行动态SQL语句的Statement。

PreparedStatement

PreparedStatement是Statement的子接口。其需要执行预编译的SQL语句(动态数据用？代替)，然后每次执行SQL语句时

只需要将动态信息传给数据库即可，数据库会重用执行计划。在执行含有动态数据的SQL语句时，能防止注入式危险。

PreparedStatement ps = conn.prepareStatement(sql);

创建PreparedStatement的同时，就要先将预编译的SQL语句发送给数据库，这时，数据库就会理解该SQL的含义，并制定计划，由于？的存在，该SQL还不能执行。但好处是我们每次只要将？需要的值传递给数据库即可，这样可以重用该执行计划(因为语义没变)，减小了数据库开销，也提高了SQL效率。

预编译SQL的注意事项:

?不需要加单引号，不需要在这里关系数据类型；

?只能是"值"的占位替换，不能作为其他内容的占位。原因是预编译的SQL语义必须确定。

?需要填写什么类型调用对应类型的set方法

?号顺序从左右到，从上到下

ps.setString(1,String str);

ps.setInt(2,int in);

ps.setDouble(3,double dou);

......

该语句段的方法调用取决于SQL语句类型，不需要传入SQL语句

ps.executeUpdate();

获取结果集中的元数据

通过元数据，可以得知结果集中有多少字段，字段叫什么名，字段是什么类型等信息

ResultSet ResultSetMetaData getMetaData();

通过元数据查看结果集中总共有多少字段

ResultSetMetaData int getColumn();

通过元数据查看结果集中字段名

ResultSetMetaData String getColumnName(int colCount)

### **BasicDataSource:**

**连接池**

引入第三方jar包的连接池类

***@see org.apache.commons.dbcp.BasicDataSource***

**连接池的业务流程**:

1.读取配置文件

@see java.util.Properties用来读取配置文件

Properties prop = new Properties();

通过文件流读取并解析配置文件内容

prop.load(new FileInputStream(config.properties));

Properties String getProperty(String key)

2.根据配置文件中每一项key,(等号左边的内容),获取对应的值(等号右边的内容).

驱动包名

String driverName = prop.getProperty("driverName");

数据库地址

String url = prop.getProperty("url");

数据库用户名

String username = prop.getProperty("username");

数据库登录密码

String password = prop.getProperty("password");

最大连接数

int maxActive = Integer.parseInt(prop.getProperty("maxActive"));

超时等待时间

int maxWait = Integer.parseInt(prop.getProperty("maxWait"));

3.初始化基本参数

BasicDataSourcebds = new BasicDataSource();

初始化驱动

bds.setDriverClassName(driverName);

初始化数据库地址

bds.setUrl(url);

初始化数据库用户名

bds.setUsername(username);

初始化数据库登录密码

bds.setPassword(password);

初始化最大连接数

bds.setMaxActive(maxActive);

初始化超时等待时间，bds.getConnection()时有效

bds.setMaxWait(maxWait);

4.连接池提供了一个方法:

BasicDataSource Connection getConnection();

该方法会将当前连接池中的一个空闲连接返回。

由于连接池在创建时可以设置超时时间，该时间的作用就在这里体现。

当连接池中没有空闲连接时，该方法会进入阻塞状态，等待控线连接。

若设置的超时时间经过后，连接池依然没有空闲连接可用时，该方法会抛出异常。

### **Transaction & DAO:**

#### *事务 Transaction ACID Atomicity-Consistency-Isolation-Durability*

保证交易可靠的机制,为了解决并发业务的冲突

在一个原子业务操作中，有多条SQL语句(DML,DQL)的时候，需要使用事务，将业务过程进行保护。需要保证:所有SQL语句都正确执行，或者都回退，不能出现执行一半的情况。

1.原子性:事务必须是原子工作单元，对于数据的修改，要么全部执行，要么全部不执行;

原子业务的标准流程

try{...

设置自动提交关闭

I.setAutoCommit(false)

SQL语句:

II.DML、DQL语句多条(UPDATE,INSERT.....)

检查余额不足，抛异常

提交操作

III.commit();

}catch(Exception e){

一旦异常就回滚操作

IV.rollback();

}finally{

关闭连接

V.close()

}

2.一致性:事务完成时，所有的数据要保持一致

3.隔离性:每个并发事务进行的数据修改必须与其他的隔离

4.持久性:事务完成时，对系统影响是永久性的(即必须提交或者回滚)

JDBC的批量操作，一般需要使用事务机制保证程序的一次性执行或者不执行

Statement 一般处理多条SQL语句的批处理

Statement void addBatch(String sql)

添加多条SQL语句到Statement对象的SQL语句缓冲区，每添加一条执行一次。

Statement void cancelBatch();放弃缓冲区的SQL不再执行

Statement void clearBatch()；清空当前SQL语句列表

Statement int[] executeBatch();

将Statement的对象中的SQL语句成批的发送到数据库去执行。

返回值大于等于0的数表示成功执行。是给出执行命令所影响数据库中行数的更新计数

PreparedStatement 一般处理成批替换预处理SQL语句参数的批处理

PreparedStatement void addBatch()；

添加多条SQL语句的参数到PreparedStatement对象的SQL语句缓冲区，每添加一组参数执行一次。

现

PreparedStatement cancelBatch(); 放弃缓冲区的SQL不再执行

PreparedStatement clearBatch(); 清空当前SQL语句列

PreparedStatement int[] executeBatch(); 发送SQL语句给数据库进行处理

批量提交参数到数据库

执行结果中的-2表示执行成功，-3表示失败。

1.返回大于等于0的数:指示成功处理了命令，是给出执行命令所影响数据库中行数的更新计数；

2.返回SUCCES\_NO\_INFO的值-2，成功执行了命令，但受影响的行数是未知的；

3.返回EXECUTE\_FAILED的值-3，未能成功执行命令，仅当命令失败后驱动程序继续处理命令时出表。

注意:

批处理添加过多而不执行可能会导致OUT OF MEMORY，所以在添加一定量以后要先执行再清空。

批量执行SQL(批量替换参数)，执行性能好于单独执行SQL。

自动增加ID的获取的标准流程：

案例背景:对于两个表emp和dept。emp是主表，dept是从表。

dept的deptno由Senquence生成，获取刚生成的deptno，添加进emp的deptno字段里。

prepareStatement(String sql,String[] seqFiled)

获取自动生成的序列 需要追加传入的参数是生成列的列名

PreparedStatement ps = conn.prepareStatement(sql1,new String[]{"deptno"});

获取自动生成的序列deptno，使用PreparedStatement ResultSet getGeneratedKeys()

ResultSet rs = ps.getGeneratedKeys();

这里只能使用序号获取结果集的值

int deptno = -1;

while(rs.next()){

deptno = rs.getInt(1);

#### *DAO:*

Data Access Object:

数据访问对象。

建立在数据库与业务层之间，封装所有对数据访问的方法。

目的是为了数据访问逻辑与业务逻辑分开。

DAO业务的标准项:

1.DAO接口:interface ObjectDao

2.数据传递对象类:class Object

3.DAO实现类:class ObjectDaoImpl

4.DAO工具类:class Tools

5.数据库连接配置文件:config.properties

DAO接口的标准业务内容:

增加

public abstract void save(Object obj);

删除：

public abstract void delete(Object obj);

修改：

public abstract void update(Object obj);

查找：

public abstract Object findById(Filed field);

public abstract List<Object> findAll();

# MYSQL

## 存储过程

1.创建：CREATE PRODUCER/FUNCTION/TRIGGER

开始结束：BEGIN……END;

语句结束符号：DELIMITER，默认是分号“;”，可以自由设置

eg:

DELIMITER $$ -- 确保将$$作为语句终结条件；

DORP PROCEDURE IF EXISTS HelloWorld$$

CREATE PROCEDURE HelloWorld()

BEGIN

SELECT ‘Hello World’;

END$$

2.调用存储过程CALL

eg：HelloWorld();

声明变量 DECLARE var\_name var\_type;

参数输入输出模式：IN（只读模式，默认可以不填）、INOUT（可读写模式）、OUT（只写模式）

使用程序时，对于存储过程的变量，如果是out类型的，使用@+varname作为其变量名

eg：

DELIMITER $$

DORP PROCEDURE IF EXISTS my\_sqrt$$

CREATE PROCEDURE my\_sqrt(**in** input\_number int, **out** output\_number float)

BEGIN

SET out\_number = SQRT(input\_number);

END$$

DELIMITER ;

CALL my\_sqrt(12, **@**out\_number);

SELECT @out\_number;

输入结果即为执行过程my\_sqrt，输出到@out\_number;

3.IF…THEN..ELSEIF….ELSE….END IF;

条件语句，IF后接条件 THEN语句后执行满足IF条件的语句，ELSEIF执行另外一个条件语句，ELSE不满足以上IF条件则执行该语句，END IF结束条件语句。

eg(默认在begin..end内)：

DECLARE a NUMERIC(8,2);

DECLARE b NUMERIC(8,2);

**IF** (a>500) **THEN** b=a\*.8;

**ELSEIF** (a>100) **THEN** b=a\*.9;

**ELSE** b=a;

**END IF**;

4.LOOP….END LOOP;

循环语句，loop\_name:LOOP…..END LOOP loop\_name;

LEAVE loop\_name字句跳出循环

eg(默认在begin..end内)：

DECLARE counter INT DEFAULT 0;

my\_simple\_loop:**LOOP**

SET counter = counter + 1;

IF (counter = 10) THEN **LEAVE** my\_simple\_loop;

END IF;

**END LOOP** my\_simple\_loop;

5.SELECT…INTO…..

交互。将select语句查询的单个记录放到into后面的**本地变量**中

6.CURSOR FOR….OPEN…CLOSE

游标，返回的select语句中查询的多列结果集。

eg：将游标中的a,b,c查询结果分别放入ax，bx，cx中

**DECLARE** curl **CURSOR FOR** select a, b, c from abc;

**DECLARE** done **INT DEFAULT** 0;

**DECLARE** **CONTINUE HANDLER FOR NOT FOUND SET** done = 1;

**OPEN** curl;

exp\_loop:**LOOP**

**FETCH** curl **INTO** ax, bx, cx

**IF** done = 1 **THEN** **LEAVE** abc;

**END IF**;

**END LOOP**;

**CLOSE** curl**;**

常用变量类型：

M 指出最大的显示尺寸。最大的合法的显示尺寸是 255 。  
D 适用于浮点类型并且指出跟随在十进制小数点后的数码的数量。最大可能的值是30，但是应该不大于M-2。  
方括号(“[”和“]”)指出可选的类型修饰符的部分。  
  
注意，如果你指定一个了为ZEROFILL，MySQL将为该列自动地增加UNSIGNED属性。  
  
TINYINT[(M)] [UNSIGNED] [ZEROFILL]  
一个很小的整数。有符号的范围是-128到127，无符号的范围是0到255。  
  
SMALLINT[(M)] [UNSIGNED] [ZEROFILL]  
一个小整数。有符号的范围是-32768到32767，无符号的范围是0到65535。  
  
MEDIUMINT[(M)] [UNSIGNED] [ZEROFILL]  
一个中等大小整数。有符号的范围是-8388608到8388607，无符号的范围是0到16777215。  
  
INT[(M)] [UNSIGNED] [ZEROFILL]  
一个正常大小整数。有符号的范围是-2147483648到2147483647，无符号的范围是0到4294967295。  
  
INTEGER[(M)] [UNSIGNED] [ZEROFILL]  
这是INT的一个同义词。  
  
BIGINT[(M)] [UNSIGNED] [ZEROFILL]  
  
一个大整数。有符号的范围是-9223372036854775808到9223372036854775807，无符号的范围是0到  
  
18446744073709551615。注意，所有算术运算用有符号的BIGINT或DOUBLE值完成，因此你不应该使用大于9223372036854775807（63位)的有符号大整数，除了位函数！注意，当两个参数是INTEGER值时，-、+和\*将使用BIGINT运算！这意味着如果你乘2个大整数(或来自于返回整数的函数)，如果结果大于9223372036854775807，你可以得到意外的结果。一个浮点数字，不能是无符号的，对一个单精度浮点数，其精度可以是<=24，对一个双精度浮点数，是在25 和53之间，这些类型如FLOAT和DOUBLE类型马上在下面描述。FLOAT(X)有对应的FLOAT和DOUBLE相同的范围，但是显示尺寸和小数位数是未定义的。在MySQL3.23中，这是一个真正的浮点值。在更早的MySQL版本中，FLOAT(precision)总是有2位小数。该句法为了ODBC兼容性而提供。  
  
FLOAT[(M,D)] [ZEROFILL]  
一个小(单精密)浮点数字。不能无符号。允许的值是-3.402823466E+38到-1.175494351E-38，0 和1.175494351E-38到3.402823466E+38。M是显示宽度而D是小数的位数。没有参数的FLOAT或有<24 的一个参数表示一个单精密浮点数字。  
  
DOUBLE[(M,D)] [ZEROFILL]  
一个正常大小(双精密)浮点数字。不能无符号。允许的值是-1.7976931348623157E+308到-2.2250738585072014E-308、 0和2.2250738585072014E-308到1.7976931348623157E+308。M是显示宽度而D是小数位数。没有一个参数的DOUBLE或FLOAT(X)（25 < = X < = 53）代表一个双精密浮点数字。  
  
DOUBLE PRECISION[(M,D)] [ZEROFILL]  
  
REAL[(M,D)] [ZEROFILL]  
这些是DOUBLE同义词。  
  
DECIMAL[(M[,D])] [ZEROFILL]  
一个未压缩(unpack)的浮点数字。不能无符号。行为如同一个CHAR列：“未压缩”意味着数字作为一个字符串被存储，值的每一位使用一个字符。小数点，并且对于负数，“-”符号不在M中计算。如果D是0，值将没有小数点或小数部分。DECIMAL值的最大范围与DOUBLE相同，但是对一个给定的DECIMAL列，实际的范围可以通过M和D的选择被限制。如果D被省略，它被设置为0。如果M被省掉，它被设置为10。注意，在MySQL3.22里，M参数包括符号和小数点。  
  
NUMERIC(M,D) [ZEROFILL]  
这是DECIMAL的一个同义词。  
  
DATE  
一个日期。支持的范围是'1000-01-01'到'9999-12-31'。MySQL以'YYYY-MM-DD'格式来显示DATE值，但是允许你使用字符串或数字把值赋给DATE列。  
  
DATETIME  
一个日期和时间组合。支持的范围是'1000-01-01 00:00:00'到'9999-12-31 23:59:59'。MySQL以'YYYY-MM-DD HH:mm:ss'格式来显示DATETIME值，但是允许你使用字符串或数字把值赋给DATETIME的列。  
  
TIMESTAMP[(M)]  
一个时间戳记。范围是'1970-01-01 00:00:00'到'2038-01-19 3:14:07'。由于mysql以4字节存储timestamp，故最大时间从1970年+2的31次方，即为2038年。

MySQL以YYYYMMDDHHmmss、YYMMDDHHmmss、YYYYMMDD或YYMMDD格式来显示TIMESTAMP值，取决于是否M是14（或省略)、12、8或6，但是允许你使用字符串或数字把值赋给TIMESTAMP列。一个TIMESTAMP列对于记录一个INSERT或UPDATE操作的日期和时间是有用的，因为如果你不自己给它赋值，它自动地被设置为最近操作的日期和时间。你以可以通过赋给它一个NULL值设置它为当前的日期和时间。  
TIME  
一个时间。范围是'-838:59:59'到'838:59:59'。MySQL以'HH:MM:SS'格式来显示TIME值，但是允许你使用字符串或数字把值赋给TIME列。Time类型不仅可以用于表示一天的时间（必须小于24小时）,还可能为某个过去的时间或两个事件的时间间隔（可以大于24小时，或者为负）  
  
YEAR[(2|4)]  
一个2或4位数字格式的年(缺省是4位)。允许的值是1901到2155，和0000（4位年格式），如果你使用2位，1970-2069( 70-69)。MySQL以YYYY格式来显示YEAR值，但是允许你把使用字符串或数字值赋给YEAR列。（YEAR类型在MySQL3.22中是新类型。）  
  
CHAR(M) [BINARY]  
一个定长字符串，当存储时，总是是用空格填满右边到指定的长度。M的范围是1 ～ 255个字符。当值被检索时，空格尾部被删除。CHAR值根据缺省字符集以大小写不区分的方式排序和比较，除非给出BINARY关键词。NATIONAL CHAR（短形式NCHAR)是ANSI SQL的方式来定义CHAR列应该使用缺省字符集。这是MySQL的缺省。CHAR是CHARACTER的一个缩写。  
  
[NATIONAL] VARCHAR(M) [BINARY]  
一个变长字符串。注意：当值被存储时，尾部的空格被删除(这不同于ANSI SQL规范)。M的范围是1 ～ 255个字符。 VARCHAR值根据缺省字符集以大小写不区分的方式排序和比较，除非给出BINARY关键词值。 VARCHAR是CHARACTER VARYING一个缩写。  
**注：**mysql 5.0.3 之前：  
0--255字节   
varchar（20）中的20表示字节数，如果存放urf8编码的话只能放6个汉字。  
MySQL 5.0.3 之后：  
0--65535字节   
varchar（20）表示字符数，不管什么编码，既汉字也能放20个。但最多占65532字节（两个字节存放长度，小于255字节用1个字节存放长度）  
  
TINYBLOB  
  
TINYTEXT  
一个BLOB或TEXT列，最大长度为255(2^8-1)个字符。

BLOB  
  
TEXT  
一个BLOB或TEXT列，最大长度为65535(2^16-1)个字符。  
  
MEDIUMBLOB  
  
MEDIUMTEXT  
一个BLOB或TEXT列，最大长度为16777215(2^24-1)个字符。  
LONGBLOB  
  
LONGTEXT  
一个BLOB或TEXT列，最大长度为4294967295(2^32-1)个字符。  
  
ENUM('value1','value2',...)  
枚举。一个仅有一个值的字符串对象，这个值式选自与值列表'value1'、'value2', ...,或NULL。一个ENUM最多能有65535不同的值。  
  
SET('value1','value2',...)  
一个集合。能有零个或多个值的一个字符串对象，其中每一个必须从值列表'value1', 'value2', ...选出。一个SET最多能有64个成员。