# 数据库入门

## 引入

方式一： 数据放到内存

优点：

1．擦写数据的效率很高！

缺点：

1．无法永久保存，当程序关闭了，数据丢失

方式二： 数据放到普通文件

优点：

1．可以永久保存

缺点：

1．频繁地进行IO操作，效率比较低

2．查询数据不方便的！！！

方式三： 数据放到数据库软件

优点：

1．数据永久保存下来

2．查询数据非常方便的！！

3．效率不能低

## 市面上常见的数据库软件

数据库软件，存储和管理数据的仓库。

Oracle: 是甲骨文公司的产品。满足中大型应用。和java语言兼容性非常好的。

SQL Server： 是微软公司的产品。和net平台（c#语言）兼容最好。

DB2: 是IBM的产品。

mysql：开源的数据库产品。和java语言兼容非常好。满足中小型应用。

mangoDB： noSQL数据库

mysql: 满足中小型应用

oracle：满足中大型应用

# 管理数据库

## 查看所有数据库

|  |
| --- |
| mysql> show databases; --分号结束。sql语句就会发送给mysql服务器端执行  +--------------------+  | Database |  +--------------------+  | information\_schema | --mysql元数据数据库。  | mysql | --mysql配置数据库。其中user表用于管理mysql用户和密码、权限信息。  | performance\_schema | --mysql性能监控信息数据库。  | test | --测试数据库。  +--------------------+  4 rows in set (0.00 sec) |

## 创建数据库

1．创建数据库，并制定默认字符集

|  |
| --- |
| mysql> create database day15  -> default character set utf8;  Query OK, 1 row affected (0.00 sec) |

**或者直接写：**

|  |
| --- |
| mysql> create database day15 default character set utf8;  Query OK, 1 row affected (0.01 sec) |

查看数据库的状态（默认字符串）：

show create database day15;

2．创建数据库，同时指定默认字符集，以及字符集的默认校验规则。

|  |
| --- |
| mysql> create database day15  -> default character set utf8  -> collate utf8\_general\_ci; --指定默认的校验规则  Query OK, 1 row affected (0.00 sec) |

校验规则：

一个字符集可以同时存在多种校验规则。

a．使用A校验规则：取字符的ASCII码值，进行比较大小

a < b

97 98

b．使用B校验规则：取字符的ASCII码值的负数，进行比较大小

a > b

-97 -98

查看常用的字符串集的校验规则：

show character set;

## 删除数据库

|  |
| --- |
| mysql> drop database day15;  Query OK, 0 rows affected (0.00 sec) |

## 修改数据库

修改数据库默认字符集

|  |
| --- |
| mysql> alter database day15 default character set gbk;  Query OK, 1 row affected (0.00 sec) |

# 管理表

**操作表之前，需要切换数据库： use 数据库名称 例如 use day14;**

## 创建新的表(create table)

|  |
| --- |
| mysql> create table student(id int,name varchar(20),age int);  Query OK, 0 rows affected (0.01 sec)  create table 表名(字段名称 字段类型,字段名称 字段类型.....); |

## 修改表（alter table）

### 添加字段：

|  |
| --- |
| mysql> alter table student add column gender varchar(1);  Query OK, 0 rows affected (0.04 sec)  Records: 0 Duplicates: 0 Warnings: 0 |

### 修改字段类型：

|  |
| --- |
| mysql> alter table student modify column gender varchar(2);  Query OK, 0 rows affected (0.03 sec)  Records: 0 Duplicates: 0 Warnings: 0 |

### 修改字段名称：

|  |
| --- |
| mysql> alter table student change column gender sex varchar(2);  Query OK, 0 rows affected (0.06 sec)  Records: 0 Duplicates: 0 Warnings: 0 |

### 删除字段：

|  |
| --- |
| mysql> alter table student drop column age;  Query OK, 0 rows affected (0.02 sec)  Records: 0 Duplicates: 0 Warnings: 0 |

### 修改表名称：

|  |
| --- |
| mysql> alter table student rename to teacher;  Query OK, 0 rows affected (0.01 sec) |

## 查看表

|  |
| --- |
| mysql> desc student;  +-------+-------------+------+-----+---------+-------+  | Field | Type | Null | Key | Default | Extra |  +-------+-------------+------+-----+---------+-------+  | id | int(11) | YES | | NULL | |  | name | varchar(20) | YES | | NULL | |  | age | int(11) | YES | | NULL | |  +-------+-------------+------+-----+---------+-------+ |

**show tables: 查看当前数据库的所有表**

## 删除表

|  |
| --- |
| mysql> drop table teacher;  Query OK, 0 rows affected (0.01 sec) |

# 管理数据

|  |
| --- |
| -- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 一、 数据增删修改操作 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*---------  -- 1.1 查看表的数据  SELECT \* FROM student;  -- 1.2 插入数据  -- 插入所有列（所有列的值都要按顺序依次插入）  INSERT INTO student VALUES(1,'eric',20,'广州人','男');  INSERT INTO student VALUES(2,'rose',25,'佛山人','女');  -- 插入部分列(列的数据和值的顺序和数量一定保持一致！)  INSERT INTO student(id,NAME,age,remark2) VALUES(3,'jacky',27,'佛山人');  -- 1.3 修改数据  -- 修改所有（用的很少）  UPDATE student SET gender='男';  -- 修改某个某些条件的数据（用的较多）  UPDATE student SET gender='女' WHERE id=2;  -- 修改多个列(SET 列名=值,列名=值...)  UPDATE student SET age=28,remark2='韶关人' WHERE id=2;  -- 1.4 删除数据  -- 删除表的所有数据（用的较少）  -- 注意：  -- 1）只能删除表中的数据，不能影响表中的自增长约束  -- 2）使用delete删除的数据，通过事务可以回滚。  -- 3）后面带条件  DELETE FROM student;  -- 删除部分  DELETE FROM student WHERE id=3;  -- 删除所有数据  -- 注意：  -- 1）既能删除表的数据，也能够把表的子增长约束置为0.  -- 2）使用truncate删除的数据，不能回滚！  -- 3）后面不能带条件  TRUNCATE TABLE student; |

# 查询数据（重点）

## 1．查询所有列

|  |
| --- |
| -- 2.1 查询所有列  SELECT \* FROM student; |

## 2．查询指定列

|  |
| --- |
| -- 2.2 查询指定列(多个列之间以逗号分隔)  SELECT NAME,gender FROM student; |

## 3．查询时指定别名（AS）

|  |
| --- |
| -- 2.3 查询时给列指定别名(AS)  SELECT NAME AS '姓名',gender AS '性别' FROM student;  -- AS可以省略  SELECT NAME '姓名',gender '性别' FROM student; |

## 4．合并列查询

|  |
| --- |
| -- 2.5 合并列查询  -- 添加servet,jsp成绩列  ALTER TABLE student ADD servlet INT,ADD jsp INT;  SELECT \* FROM student;  UPDATE student SET servlet=86,jsp=75 WHERE id=1;  UPDATE student SET servlet=90,jsp=65 WHERE id=2;  UPDATE student SET servlet=78,jsp=50 WHERE id=3;  UPDATE student SET servlet=85,jsp=80 WHERE id=4;  -- 需求： 查询每个学生的servlet和jsp的总成绩  -- 注意： 必须是数值类型  SELECT NAME '姓名',(servlet+jsp) '总成绩' FROM student;  -- 注意，和数值以外合并没有意义  SELECT NAME '姓名',(gender+jsp) FROM student; |

## 5．查询时添加常量列

|  |
| --- |
| -- 2.4 查询时添加常量列  -- 需要： 在查询学生表时都带上一个班级列，内容为"java就业班"  SELECT NAME '姓名',gender '性别',age '年龄','java就业班' AS '班级' FROM student; |

## 6．查询去除重复数据（DISTINCT）

|  |
| --- |
| -- 2.6 查询去除重复数据  -- 需求： 统计学生都来自于哪里  SELECT DISTINCT remark2 FROM student;  -- 这种语法也可以  SELECT DISTINCT(remark2) FROM student; |

## 7．条件查询（WHERE）

|  |
| --- |
| -- 2.7 条件查询(where)  SELECT \* FROM student;  **-- 1) 逻辑条件： and or**  -- 需求: 查询id值为3，且性别为男的学生(交集：两个条件同时满足)  SELECT \* FROM student WHERE id=3 AND gender='男';  -- 需求：查询id为3，或性别为男的学生（并集： 两个条件其中一个满足）  SELECT \* FROM student WHERE id=3 OR gender='男';  **-- 2）比较条件： > < >= <= = between and**  -- 需求： 查询servlet分数大于80分的学生  SELECT \* FROM student WHERE servlet>80;  -- 需求：查询jsp分数小于或等于80分的学生  SELECT \* FROM student WHERE jsp<=80;  SELECT \* FROM student WHERE jsp<80 OR jsp=80;  -- 需求： 查询servlet成绩大于等于80，且小于等于88的学生  SELECT \* FROM student WHERE servlet>=80 AND servlet<=88;  -- 等价于上面  SELECT \* FROM student WHERE servlet BETWEEN 80 AND 88; -- (包前包后)  **-- 3）判空条件: ='' is null <>'' is not null**  UPDATE student SET remark2="" WHERE id=1;  UPDATE student SET remark2=NULL WHERE id=4;  -- 需求： 查询remark字段为空的学生（包含null，也包含字符串）  -- null 和 空字符串的区别：  -- null： 没有数据。 判断null： is null,判断不为null： is not null  -- 空字符： 有数据，数据就是空字符串。判断空字符： =''； 判断不为空字符串： <>''  SELECT \* FROM student WHERE remark2 IS NULL;  SELECT \* FROM student WHERE remark2='';  SELECT \* FROM student WHERE remark2 IS NULL OR remark2='';  -- 需求：查询remark字段不为空的学生  SELECT \* FROM student WHERE remark2 IS NOT NULL AND remark2<>'';  **-- 4) 模糊条件： like**  -- like后面跟上符合  -- %: 表示任意多个字符（0-多个）  -- \_: 表示一个字符  -- 需求： 查询姓张的学生  SELECT \* FROM student WHERE NAME LIKE '张%';  -- 需求： 查询姓名中包含'张'字的学生  SELECT \* FROM student WHERE NAME LIKE '%张%';  -- 需求： 查询姓张，且姓名只有两个字的学生  SELECT \* FROM student WHERE NAME LIKE '张\_\_'; |

## 8．聚合查询

|  |
| --- |
| -- 2.8 聚合查询  SELECT \* FROM student;  DELETE FROM student WHERE id=5;  -- 需求： 查询所有学生servlet的总分 (SUM: 求和函数)  SELECT SUM(servlet) FROM student;  -- 需求： 查询所有学生servlet的平均分（AVG; 平均函数）  SELECT AVG(servlet) FROM student;  -- 需求：查询最高的servlet分数（MAX：最大值函数）  SELECT MAX(servlet) FROM student;  -- 需求：查询最低的servlet分数（MIN：最小值函数）  SELECT MIN(servlet) FROM student;  -- 需求： 一共几个学生（COUNT： 统计数量函数）  SELECT COUNT(\*) FROM student;  SELECT COUNT(id) FROM student; -- 效率会比count(\*)效率稍高  -- 注意： 聚合函数，会排除null值的数据  SELECT COUNT(remark2) FROM student; |

## 9．分页查询（LIMIT）

|  |
| --- |
| -- 2.9 分页查询(limit)  -- limit 起始行数,查询的行数  -- 起始行数从0开始  SELECT \* FROM student;  -- 需求：学生共20条数据，每页显示5条，共4页  -- 看第3页(第11到15条)的学生数据sql：select \* from student limit 10,5;  -- 看第4页(第16到20条)的学生数据sql: select \* from student limit 15,5;    -- 知道： 当前页码，每页显示条数    -- 结论分页查询当前页数据的sql： select \* from student limit (当前页码-1)\*每页显示条数,每页显示条数;  -- 需求： 查询第1,2条数据  SELECT \* FROM student LIMIT 0,2;  -- 需求： 查询第3,4条数据  SELECT \* FROM student LIMIT 2,2;  -- 需求：查询第5,6条数据  SELECT \* FROM student LIMIT 4,2; |

## 10．查询后排序（ORDER BY）

|  |
| --- |
| -- 2.10 查询后排序（order by）  -- desc: 降序。数值从大到小，字母z-a  -- asc: 升序。数值从小到大，字母a-z  -- 默认情况下，按照插入的顺序排序  SELECT \* FROM student;  -- 需求：按照id的升序排序  SELECT \* FROM student ORDER BY id ASC;  -- 需求： 按照servlet成绩降序排序  SELECT \* FROM student ORDER BY servlet DESC;  -- 多个排序条件的情况：先按照前面的条件排序，当出现重复记录，再按照后面的条件排序  -- 需求： 按照age升序，按照servlet成绩升序排序  SELECT \* FROM student ORDER BY age ASC,servlet ASC;  -- select \* from student order by username asc; |

## 11．分组查询（GROUP BY）

|  |
| --- |
| -- 2.11 分组查询(group by)  -- 需求： 查询每个地区有多少人  -- 预期结果：  -- 广州天河 3  -- 广州越秀 1  SELECT address,COUNT(\*) FROM student;  -- 1)对地区进行分组 2)在分组的基础可以进行统计，统计的是每组的数据  SELECT address,COUNT(\*) FROM student GROUP BY address;  -- 需求： 统计男女的人数  -- 注意： where条件必须放在group by分组之前  SELECT gender,COUNT(\*) FROM student WHERE gender IS NOT NULL AND gender<>'' GROUP BY gender; |

## 12．分组后筛选（）

|  |
| --- |
| -- 3.12 分组后筛选(having)  -- 需求： 查询哪些地区的人数大于2个的地区  -- 1）查询哪些地区多少人 2）筛选人数大于2的地区  -- 注意： having使用在group by分组之后的，对分组后的条件进行筛选  SELECT address,COUNT(\*) FROM student GROUP BY address HAVING COUNT(\*)>2 ; |

# 字段类型

## char(20) vs varchar(20)

char(20): 固定长度的字符串。不管实际存储的数据的大小，一定占用20个字符空间

varchar(20): 可变长度的字符串。占用的空间大小就是实际存储的数据大小。

## int vs int(4)

int: 默认最多11位，长度根据实际存储的数值的长度

int(4): 固定的数组长度

**例：**

CREATE TABLE test(

id1 INT,

id2 INT(4) ZEROFILL -- zerofill: 零填充

)

INSERT INTO test VALUES(1,1);

SELECT \* FROM test;

## date vs datetime vs timestamp

-- date: 只记录日期 2019-08-29

-- datetime： 记录日期+时间 2019-08-29 14:02:45

-- timestamp: 时间戳,用于记录当前数据的插入或更新的时间（只要有修改就会更新）

# 总结：

mysql基础：

1）管理数据库：

增： create database 数据库名 default character set 字符集;

修： alter database 数据库名 default character set 新的字符集;

删： drop database 数据库名;

查： show databases;

2）管理表：

增： create table 表名(字段名1 字段类型,字段名2 字段类型......);

修：

增加字段： alter table 表名 add column 字段名 字段类型;

修改字段类型： alter table 表名 modify column 字段名 新的字段类型;

修改字段名： alter table 表名 change column 旧字段名 新的字段名 字段类型;

修改表名： alter table 表名 rename to 新表名;

删： drop table 表名;

查： show tables; desc 表名;

3)管理数据：

增： insert into 表名(字段名1,字段名2....) values(值1，值2.....);

修： update 表名 set 字段名1=值1,字段名2=值2...... where 条件;

删： delete from 表名 where 条件;

truncate table 表名;

查： (12中查询)

a）所有字段：select \* from 表名;

b）指定字段： select 字段名1,字段名2 from 表名;

c）指定别名： select 字段名1 as 别名1,字段名2 as 别名2 from 表名;

d）添加常量列： select 常量值 as 别名 from 表名;

e）合并列： select (字段名1+字段名2+....) as 别名 from 表名;

f）去除重复： select distinct 字段名 from 表名;

g）条件查询：

逻辑条件： where 条件1 and/or 条件2;

比较条件： where 字段名 >/>=/</<=/=/<> 值

where 字段名 between 值1 and 值2;

判空条件：

null; where 字段名 is null/is not null;

空字符串： where 字段名=''/<>''

模糊条件： where 字段名　like 值

%： 代表任意个字符

\_: 代表一个字符

h）聚合查询：

max(): 最大值

min（）：最小值

avg（）:平均值

count(): 统计数量

i）分页查询：

limit 起始行，查询行数

j）排序：

order by 字段名 asc/desc

asc: 升序

desc: 降序

h）分组查询：

group by 字段名

k）分组后筛选：

group by 字段名 having 条件;

sql语句分类：

1）数据定义语句（DDL）：

create/alter/drop

2）数据操作语句（DML）

insert/update/delete/truncate

3）数据查询语句（DQL）

select/show

# 数据约束

## 简介

给表添加数据约束，从而约束用户操作表数据的行为。

## 默认值约束

注意： 当前没有插入默认值字段的时候，默认值才会起作用

|  |
| --- |
| -- 1.1 默认值(**default**)  CREATE TABLE test(  NAME VARCHAR(20),  gender VARCHAR(2) DEFAULT '男'  )  INSERT INTO test(NAME) VALUES('张三');  INSERT INTO test(NAME,gender) VALUES('张三',NULL);  SELECT \* FROM test;  -- 需求：当不插入gender的时候，分配一个‘男’的默认值  -- 注意： 1）当没有插入gender字段的时候，分配一个默认值 |

## 非空约束

|  |
| --- |
| -- 1.2 非空（not null）  CREATE TABLE test(  NAME VARCHAR(20) **NOT NULL**,  gender VARCHAR(2)  )  -- 需求： name字段一定要有值（不能不插入数据，不能是null），这是给name添加非空约束  -- 1)非空约束，不能不插入值  INSERT INTO test(gender) VALUES('男'); -- ERROR 1364 (HY000): Field 'NAME' doesn't have a default value  -- 2)非空约束，不能插入null  INSERT INTO test(NAME,gender) VALUES(NULL,'男'); -- ERROR 1048 (23000): Column 'NAME' cannot be null |

## 唯一约束

|  |
| --- |
| CREATE TABLE test(  id INT UNIQUE,  NAME VARCHAR(20)  )  -- 需求： id的值不能出现重复值。这时就要给id添加一个唯一约束。  INSERT INTO test(id,NAME) VALUES(1,'张三');  -- 1)**不能插入重复的值**  INSERT INTO test(id,NAME) VALUES(1,'李四'); -- Duplicate entry '1' for key 'id'  INSERT INTO test(id,NAME) VALUES(2,'李四');  -- 2)**唯一约束，可以插入多个null。所以唯一约束不能约束null**  INSERT INTO test(id,NAME) VALUES(NULL,'王五');  INSERT INTO test(id,NAME) VALUES(NULL,'陈六'); |

## 主键约束

|  |
| --- |
| -- 1.4 主键约束(primary key)（**唯一+非空**）  -- 注意：  -- 1）通常情况下，我们会给每张表都会设置一个主键字段，用来标记记录的唯一性  -- 2）但是不建议把业务含义字段作为主键，因为随着业务的变化，业务字段可能会出现重复。  -- 3）建议给每张张独立添加一个叫id的字段，把这个id字段设置成主键，用来作为记录的唯一性。  CREATE TABLE test(  id INT **PRIMARY KEY**,  NAME VARCHAR(20)  )  -- 1)唯一性  INSERT INTO test(id,NAME) VALUES(1,'张三');  INSERT INTO test(id,NAME) VALUES(1,'张三'); -- Duplicate entry '1' for key 'PRIMARY'  -- 2）非空性  INSERT INTO test(id,NAME) VALUES(NULL,'张三'); -- ERROR 1048 (23000): Column 'id' cannot be null |

## 自增长约束

作用： 自动递增

|  |
| --- |
| -- 1.5 自增长（auto\_increment**）**  CREATE TABLE student(  id INT(4) ZEROFILL PRIMARY KEY **AUTO\_INCREMENT**, -- 自增长，从0开始 ZEROFILL 零填充  NAME VARCHAR(20)  )  -- 自增长字段可以不赋值，自动递增  INSERT INTO student(NAME) VALUES('张三');  INSERT INTO student(NAME) VALUES('李四');  INSERT INTO student(NAME) VALUES('王五');  SELECT \* FROM student;  -- 不能影响自增长约束  DELETE FROM student;  -- 可以影响自增长约束  TRUNCATE TABLE student; |

作用：约束两种表的数据

出现两种表的情况：

解决数据冗余高问题： 独立出一张表

例如： 员工表 和 部门表

问题出现：在插入员工表数据的时候，员工表的部门ID字段可以随便插入！！！！！

使用外键约束：约束插入员工表的部门ID字段值

解决办法： 在员工表的部门ID字段添加一个外键约束

|  |
| --- |
| -- 部门表（主表）  CREATE TABLE dept(  id INT PRIMARY KEY,  deptName VARCHAR(20)  )  -- 修改员工表（副表/从表）  CREATE TABLE employee(  id INT PRIMARY KEY,  empName VARCHAR(20),  deptId INT,-- 把部门名称改为部门ID  -- 声明一个外键约束  CONSTRAINT emlyee\_dept\_fk FOREIGN KEY(deptId) REFERENCES dept(id)  -- 外键名称 外键 参考表(参考字段)  ) |

注意：

1）被约束的表称为副表，约束别人的表称为主表，外键设置在副表上的！！！

2）主表的参考字段通用为主键！

**3）添加数据： 先添加主表，再添加副表**

**4）修改数据： 先修改副表，再修改主表**

**5）删除数据： 先删除副表，再删除主表**

|  |
| --- |
| -- 1.6 外键约束  -- 员工表  CREATE TABLE employee(  id INT PRIMARY KEY,  empName VARCHAR(20),  deptName VARCHAR(20) -- 部门名称  )  INSERT INTO employee VALUES(1,'张三','软件开发部');  INSERT INTO employee VALUES(2,'李四','软件开发部');  INSERT INTO employee VALUES(3,'王五','应用维护部');  SELECT \* FROM employee;  -- 添加员工，部门名称的数据冗余高  INSERT INTO employee VALUES(4,'陈六','软件开发部');  -- 解决数据冗余高的问题：给冗余的字段放到一张独立表中  -- 独立设计一张部门表  CREATE TABLE dept(  id INT PRIMARY KEY,  deptName VARCHAR(20)  )  DROP TABLE employee;  -- 修改员工表  CREATE TABLE employee(  id INT PRIMARY KEY,  empName VARCHAR(20),  deptId INT,-- 把部门名称改为部门ID  -- 声明一个外键约束  CONSTRAINT emlyee\_dept\_fk FOREIGN KEY(deptId) REFERENCES dept(id) ON UPDATE CASCADE  -- 外键名称 外键 参考表(参考字段)  )  INSERT INTO dept(id,deptName) VALUES(1,'软件开发部');  INSERT INTO dept(id,deptName) VALUES(2,'应用维护部');  INSERT INTO dept(id,deptName) VALUES(3,'秘书部');  INSERT INTO employee VALUES(1,'张三',1);  INSERT INTO employee VALUES(2,'李四',1);  INSERT INTO employee VALUES(3,'王五',2);  INSERT INTO employee VALUES(4,'陈六',3);  -- 问题: 该记录业务上不合法，员工插入了一个不存在的部门数据  INSERT INTO employee VALUES(5,'陈六',4); -- 违反外键约束： Cannot add or update a child row: a foreign key constraint fails (`day16`.`employee`, CONSTRAINT `emlyee\_dept\_fk` FOREIGN KEY (`deptId`) REFERENCES `dept` (`id`))  -- 1）**当有了外键约束，添加数据的顺序： 先添加主表，再添加副表数据**  -- 2）**当有了外键约束，修改数据的顺序： 先修改副表，再修改主表数据**  -- 3）**当有了外键约束，删除数据的顺序： 先删除副表，再删除主表数据**  -- 修改部门(不能直接修改主表)  UPDATE dept SET id=4 WHERE id=3;  -- 先修改员工表  UPDATE employee SET deptId=2 WHERE id=4;    -- 删除部门  DELETE FROM dept WHERE id=2;  -- 先删除员工表  DELETE FROM employee WHERE deptId=2;  SELECT \* FROM dept;  SELECT \* FROM employee; |

## 级联操作

级联： 当有了外键的时候，我们希望修改或删除数据的时候，修改或删除了主表的数据，同时能够影响副表的数据，这时就可以使用级联。

|  |
| --- |
| CREATE TABLE employee(  id INT PRIMARY KEY AUTO\_INCREMENT,  NAME VARCHAR(20),  deptId INT,  -- 添加级联修改： ON UPDATE CASCADE  -- 添加级联删除： ON DELETE CASCADE  CONSTRAINT employee\_dept\_fk FOREIGN KEY(deptId) REFERENCES dept(id) ON UPDATE CASCADE ON DELETE CASCADE  -- 外键名称 外键字段 参考  )  **注意： 级联操作必须在外键基础上使用** |

|  |
| --- |
| -- 级联修改（修改）  -- 直接修改部门  UPDATE dept SET id=5 WHERE id=4;  -- 级联删除  -- 直接删除部门  DELETE FROM dept WHERE id=1; |

# 数据库设计

## 引入

需求分析： 原始需求 -> 业务需求

客户 业务模型：客户（姓名 登记时间 邮箱 微信号 ）

需求设计：

业务模型（客户业务模型） -> 抽取实体模型： class Customer{name email weixin}

业务模型（客户业务模型） -> 数据模型：设计数据库表（customer表： 字段）

## 数据库设计的三大范式

第一范式： 要求表的每个字段必须独立的不可分割的单元

学生表： student name --违反第一范式

张三|狗娃

王含|张小含

学生表： student name old\_name --符合第一范式

张三 狗娃

王含 张小含

第二范式： 在第一范式的基础上，要求表的除主键以外的字段都和主键有依赖关系的。

一张表应该只表达一个意思！！！！

员工表：employee

员工编号 员工姓名 部门名称 订单名称 --违反第二范式

员工表：employee

员工编号 员工姓名 部门名称 --符合第二范式

订单表：

订单编号 订单名称

第三范式： 在第二范式基础上，要求表的除主键外的字段都只能和主键有直接决定的依赖关系。

员工表：employee --不符合第三范式（出现数据冗余）

**员工编号** 员工姓名 部门编号 部门名称

1 张三 1 软件开发部

2 李四 1 软件开发部

员工表：employee --符合第三范式（降低数据冗余）

**员工编号** 员工姓名 部门编号

1 张三 1

2 李四 1

部门表： dept

部门编号 部门名称

1 软件开发部

数据库存储空间 和 数据查询效率

# 多表查询（关联查询）

|  |
| --- |
| -- 1.1 交叉连接查询（笛卡尔乘积: 4\*3=12，产生笛卡尔积的原因是没有足够的连接条件）  -- 需求： 员工及其部门  SELECT employee.name,dept.name FROM employee,dept;  -- 多表查询的步骤： 1）确定查询哪些表 2）确定查询哪些字段 3）确定连接条件(规则： 表数量-1)  -- 内连接效果：只有满足连接条件的数据才会显示出来  -- 1.2 内连接查询（使用最多）  SELECT e.name,d.name  FROM employee e,dept d  WHERE e.deptId=d.id;  -- 另一种语法  SELECT e.name,d.name  FROM employee e  INNER JOIN dept d  ON e.deptId=d.id;    -- 1.3 左外连接查询（其次）  -- 需求： 查询部门及其部门的员工（部门全部显示）  -- 左外连接查询效果： 左表（部门表）的数据全部显示，右表（员工）的数据当满足连接条件的时候，  -- 就显示满足条件的的数据，但是如果不满足连接条件，则显示null  -- 预期结果：  -- 软件开发部 张三  -- 软件开发部 王五  -- 软件维护部 李四  -- 软件维护部 陈六  -- 秘书部 null  SELECT d.name,e.name  FROM dept d  LEFT OUTER JOIN employee e  ON d.id=e.deptId;  -- 1.4 右外连接查询  -- 右外连接查询效果： 右表（部门表）的数据全部显示，左表（员工）的数据当满足连接条件的时候，  -- 就显示满足条件的数据，但是如果不满足连接条件，则显示null  SELECT d.name,e.name  FROM employee e  RIGHT OUTER JOIN dept d  ON d.id=e.deptId;      -- 1.5 自连接查询  -- 需求： 查询员工姓名及其上司姓名(没有上司的员工也显示)  -- 预期结果：  员工 上司  -- 张三 null  -- 李四 张三  -- 王五 李四  -- 陈六 王五  SELECT e.name AS '员工',b.name AS '上司'  FROM employee e  LEFT OUTER JOIN employee b  ON e.bossId=b.id; |

# mysql存储过程

## 引入

存储过程，其实就是带逻辑的（多个）sql语句。也是sql编程。

## 存储过程的特点

1．存储过程保存到数据库服务器端，通过数据库客户端工具调用存储过程

2．存储过程的效率会非常高！因为存储过程是在数据库服务器端执行。

3．存储过程的移植性非常差的！

## mysql数据库三种变量：

1．全局变量。mysql内置的变量，mysql程序关闭的时候全局变量才会失效！！

-- character\_set\_client: mysql接收的客户端的数据编码

-- character\_set\_results： mysql使用什么编码输出给客户端数据

-- 查看某个全局变量： select @@变量名

-- 修改某个全局变量： set @@变量名=值

2．会话变量。变量只在某次登录的会话中有效！退出连接，会话变量数据失效！！

-- 查看某个会话变量： select @变量名

-- 修改/定义某个会话变量： set @变量名=值

3．局部变量：在存储过程中定义的变量。存储过程结束局部变量失效！！

-- 查看某个局部变量： select 变量名

-- 修改某个局部变量： set 变量名=值

-- 定义某个局部变量： declare 变量名 数据类型;

## 存储过程语法

|  |
| --- |
| 创建存储过程  -- 定义结束符号  **DELIMITER 结束符号（一般是$）**  **CREATE PROCEDURE 存储过程名称 (形式参数列表)**  **BEGIN**  **多个sql语句**  **END 结束符号**  -- 调用存储过程  CALL 存储过程名称(实际参数列表);  参数类型：  IN： 输入参数，可以携带数据到存储过程中  OUT: 输出参数，可以携带数据到存储过程外面。  INOUT: 输入输出参数。 |

|  |
| --- |
| -- 3.1 带有输入参数的存储过程  -- 需求： 传入员工id查询对应的员工  **DELIMITER $**  **CREATE PROCEDURE pro\_testByIn(IN eid INT) -- 参数类型（IN） 参数名称 数据类型(int)**  **BEGIN**  **SELECT \* FROM employee WHERE id=eid;**  **END $**  -- 调用  CALL pro\_testByIn(2);  -- 3.2 带有输出参数的存储过程  **DELIMITER $**  **CREATE PROCEDURE pro\_testByOut(OUT n VARCHAR(20))**  **BEGIN**  **-- 修改变量n**  **SET n = '输出参数';**  **END $**  -- 问题： 如何接收存储过程的输出参数？？？  -- 定义变量去接收输出参数数据。  **-- 定义会话变量去接收输出参数数据**  **-- set @n='eric';**  **CALL pro\_testByOut(@n);**  **-- 查看会话变量n**  **SELECT @n;**  -- 3.3 带有输入输出参数的存储过程  **DELIMITER $**  **CREATE PROCEDURE pro\_testByInOut(INOUT n VARCHAR(20))**  **BEGIN**  **-- 查看n变量**  **SELECT n;**  **-- 修改n变量**  **SET n = '500';**  **END $**  -- 定义会话变量调用存储过程  **SET @n='100';**  **CALL pro\_testByInOut(@n);**  **-- 查看n**  **SELECT @n;**  -- 3.4 带有判断条件的存储过程  -- 需求： 输入一个num整数，num=1 ,输出‘星期一’,num=2,输出‘星期二’，num=3,输出‘星期三’，否则，输出‘错误参数’  **DELIMITER $**  **CREATE PROCEDURE pro\_testByIf(IN num INT,OUT str VARCHAR(20))**  **BEGIN**  **IF num=1 THEN**  **SET str = '星期一';**  **ELSEIF num= 2 THEN**  **SET str ='星期二';**  **ELSEIF num=3 THEN**  **SET str = '星期三';**  **ELSE**  **SET str = '错误参数';**  **END IF;**  **END $**  CALL pro\_testByIf(5,@str);  SELECT @str;  -- 3.5 带有循环条件的存储过程  -- 需求： 输入一个num，计算从1到num的总和。  **DELIMITER $**  **CREATE PROCEDURE pro\_testByWhile(IN num INT,OUT score INT)**  **BEGIN**  **-- int result =0;**  **-- for(int i=1;i<=100;i++){**  **-- result += i;**  **-- }**    **-- 定义局部变量**  **DECLARE i INT DEFAULT 1;**  **DECLARE result INT DEFAULT 0;**  **WHILE i<=num DO**  **SET result = result + i;**  **SET i = i+1;**  **END WHILE;**  **SET score = result;**  **END $**  CALL pro\_testByWhile(200,@score);  SELECT @score;  -- 3.6 携带数据库的数据给输出参数(INTO)  -- 需求： 传入员工id，查询对应的员工，输出员工姓名  **DELIMITER $**  **CREATE PROCEDURE pro\_testByData(IN eid INT,OUT sname VARCHAR(20))**  **BEGIN**  **SELECT NAME INTO sname FROM employee WHERE id=eid;**  **END $**  CALL pro\_testByData(2,@sname);  SELECT @sname;ss |

# 触发器

## 触发器作用

当操作了某张表时，希望同时触发一些动作/行为，可以使用触发器完成！！

例如： 当向员工表插入一条记录时，希望同时往日志表插入数据

|  |
| --- |
| -- 需求： 当向员工表插入一条记录时，希望mysql自动同时往日志表插入数据  -- 创建触发器(添加)  **CREATE TRIGGER tri empAdd AFTER INSERT ON employee FOR EACH ROW** -- 当往员工表插入一条记录时  INSERT INTO test\_log(content) VALUES('员工表插入了一条记录');    -- 插入数据  INSERT INTO employee(id,empName,deptId) VALUES(7,'扎古斯',1);  INSERT INTO employee(id,empName,deptId) VALUES(8,'扎古斯2',1);  -- 创建触发器(修改)  **CREATE TRIGGER tri\_empUpd AFTER UPDATE ON employee FOR EACH ROW** -- 当往员工表修改一条记录时  INSERT INTO test\_log(content) VALUES('员工表修改了一条记录');    -- 修改  UPDATE employee SET empName='eric' WHERE id=7;    -- 创建触发器(删除)  **CREATE TRIGGER tri\_empDel AFTER DELETE ON employee FOR EACH ROW** -- 当往员工表删除一条记录时  INSERT INTO test\_log(content) VALUES('员工表删除了一条记录');    -- 删除  DELETE FROM employee WHERE id=7; |

# mysql权限问题

|  |
| --- |
| -- \*\*\*\*\*\*\*\*\*\*\*五、mysql权限问题\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  -- mysql数据库权限问题：root ：拥有所有权限（可以干任何事情）  -- 权限账户，只拥有部分权限（CURD）例如，只能操作某个数据库的某张表  -- 如何修改mysql的用户密码？  -- password: md5加密函数(单向加密)  SELECT PASSWORD('root'); -- \*81F5E21E35407D884A6CD4A731AEBFB6AF209E1B    -- mysql数据库，用户配置 : user表  USE mysql;  SELECT \* FROM USER;  -- 修改密码  **UPDATE USER SET PASSWORD=PASSWORD('123456') WHERE USER='root';**  -- 分配权限账户  **GRANT SELECT ON day16.employee TO 'eric'@'localhost' IDENTIFIED BY '123456';**  **GRANT DELETE ON day16.employee TO 'eric'@'localhost' IDENTIFIED BY '123456';** |

# 总结：

数据约束：

默认值： default 默认值

唯一： unique

非空： not null

主键：primary key （唯一+非空 ）

自增长： auto\_increment

外键： foreign key

级联操作：

级联更新： on update cascade

级联删除： on delete　cascade

多表查询：

1．内连接查询： inner join

作用： 当两种表满足了连接条件时的数据才会显示。

2．左外连接： left outer join

作用：左表完全显示。右表的数据如果满足了连接条件，则显示对应的数据，如果不满足条件，那么显示null。

3．右外连接： right outer join

作用：右表完全显示。左表的数据如果满足了连接条件，则显示对应的数据，如果不满 足条件，那么显示null。

数据库设计：

第一范式：要求表的每个字段都必须独立的不可分割的最小单元。

第二范式：要求表的除主键外的其他字段都和主键有依赖关系。（一张表表达一个意思）

第三范式：要求表的除主键外的其他字段都只能由主键决定。

# jdbc入门

## 简介

使用java程序访问（操作）数据库（发送sql语句），这叫用到了jdbc技术！！！！

## 使用java程序访问数据库的前提

1．先登录到数据库：

数据库的主机地址（ip地址）

端口

数据库用户名

数据库用户密码

连接的数据库

2．发送sql语句

## Jdbc的核心api

|-Driver接口： 驱动程序接口。

|-Connection connect() 用于连接数据库的方法

可以使用驱动程序管理类获取连接：

DriverManager.getConnection(url,user,pasword);

|-**Connection**接口： 代表和数据库的连接

|- Statement createStatement() 创建Statement接口的对象。

|- PreparedStatement prepareStatement(String sql) 创建PreparedStatement接口的对象。

|- CallableStatement prepareCall(String sql)创建CallableStatement接口的对象。

|-**Statement**接口：用于执行静态 SQL 语句。

|- int executeUpdate(String sql) 执行DDL和DML语句（更新sql语句）

|- ResultSet executeQuery(String sql) 执行DQL语句(查询sql语句)

|-**PreparedStatement**接口：用于执行预编译的 SQL 语句

|- int executeUpdate()：执行DDL和DML语句（更新sql语句）

|- ResultSet executeQuery() 执行DQL语句(查询sql语句)

|-**CallableStatement**接口： 用于执行存储过程的SQL语句

|- ResultSet executeQuery() 存储过程只能执行查询sql

|-**ResultSet**接口： 表示数据库结果集。

|- boolean next() 将光标移至下一行

|- getXXX(): 获取结果集中的每列的值

## jdbc操作步骤

1）注册驱动程序

2）获取连接对象

3）准备sql语句(DDL+DML)

4）创建Statement对象( Statement,PreparedStatment,CallableStement )

5）执行sql语句

DDL+DML: executeUpdte(sql)

DQL; executeQuery()

6）返回结果，处理结果

ResultSet

7）关闭资源

### 实例：

public class Demo1 {

//连接数据库的字符串

private static String url = "jdbc:mysql://localhost:3306/day16";

// jdbc协议+数据库协议+主机地址+端口+连接的数据库

//用户名

private static String user = "root";

//密码

private static String password = "root";

public static void main(String[] args) throws Exception {

//t1();

//t2();

//t3();

//1.驱动程序(只注册一次)

Class.forName("com.mysql.jdbc.Driver");

//2.使用驱动管理类获取连接

Connection conn = DriverManager.getConnection(url, user, password);

System.out.println(conn);

}

private static void t3() throws ClassNotFoundException, SQLException {

/\*\*

1．直接使用驱动程序连接

private static void t1() throws SQLException {

//1.创建驱动程序的实现类对象

Driver driver = new com.mysql.jdbc.Driver(); //新版本

//Driver driver = new org.gjt.mm.mysql.Driver(); 旧版本

Properties prop = new Properties();

prop.setProperty("user", user); //用户名

prop.setProperty("password", password);

//2.连接数据库

Connection conn = driver.connect(url, prop);

}

\*/

/\*\*

2．使用驱动管理类DriverManager，连接数据库

private static void t2() throws SQLException {

Driver driver = new com.mysql.jdbc.Driver(); //mysql

//1.注册驱动程序

DriverManager.registerDriver(driver);

//2.获取连接数据库

Connection conn = DriverManager.getConnection(url, user, password);

//通过url识别需要连接数据库

}

\*/

/\*\*

\* **3．使用反射：获取类的对象（推荐使用）**

\*/

Class.forName("com.mysql.jdbc.Driver"); //执行Driver类中的静态代码块

//2.获取连接数据库

Connection conn = DriverManager.getConnection(url, user, password);//通过url识别需要连接数据库

}

}

### 最优操作步骤

//1)注册驱动程序

Class.forName("com.mysql.jdbc.Driver");

//2)获取连接

Connection conn = DriverManager.getConnection(url, user, password);

//3)准备sql语句

String sql = "CREATE TABLE student(id INT,NAME VARCHAR(20),age INT)";

//4)在连接基础上，创建Statement

Statement stmt = conn.createStatement();

//5)执行sql,返回结果

int count = stmt.executeUpdate(sql);

System.out.println("影响了"+count+"行！");

//6)关闭资源.后打开的先关闭

stmt.close();

conn.close();

# Statement对象执行SQL操作

## 执行DDL操作

|  |
| --- |
| /\*\*  \* 通过jdbc执行DDL语句  \*  \*/  **public** **class** Demo1 {  //数据库的连接的URL  **private** **static** String *url* = "jdbc:mysql://localhost:3306/day17";  //数据库用户名  **private** **static** String *user* = "root";  //数据库密码  **private** **static** String *password* = "root";    **public** **static** **void** main(String[] args){  Connection conn = **null**;  Statement stmt = **null**;  **try** {  //1.驱动驱动程序  Class.*forName*("com.mysql.jdbc.Driver");  //2.从驱动程序管理类获取连接  conn = DriverManager.*getConnection*(*url*, *user*, *password*);  //3.通过Connection对象获取Statement对象  stmt = conn.createStatement();  //4.准备sql语句  String sql = "CREATE TABLE student(id INT PRIMARY KEY AUTO\_INCREMENT,NAME VARCHAR(20),gender VARCHAR(2))";  //5.执行sql语句，返回结果  **int** count = stmt.executeUpdate(sql);    System.*out*.println("影响了"+count+"行");  } **catch** (Exception e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  } **finally**{  //6.关闭资源（先关闭statement，再关闭connection）  **if**(stmt!=**null**)  **try** {  stmt.close();  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  **if**(conn!=**null**)  **try** {  conn.close();  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }  }  } |

## 执行DML操作

|  |
| --- |
| /\*\*  \* 通过jdbc执行DML语句(insert/update/delete)  \* **@author** APPle  \*  \*/  **public** **class** Demo2 {  //数据库的连接的URL  **private** **static** String *url* = "jdbc:mysql://localhost:3306/day17";  //数据库用户名  **private** **static** String *user* = "root";  //数据库密码  **private** **static** String *password* = "root";  /\*\*  \* 执行插入操作  \*/  @Test  **public** **void** test1(){  Connection conn = **null**;  Statement stmt = **null**;  **try** {  //注册驱动  Class.*forName*("com.mysql.jdbc.Driver");  //获取连接  conn = DriverManager.*getConnection*(*url*, *user*, *password*);  //创建Statment对象  stmt = conn.createStatement();  //准备sql  String sql = "INSERT INTO student(NAME,gender) VALUES('张三','男')";  //执行sql,返回结果  **int** count = stmt.executeUpdate(sql);  System.*out*.println("影响了"+count+"行");  } **catch** (Exception e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  } **finally**{  //关闭资源  **if**(stmt!=**null**)  **try** {  stmt.close();  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  **if**(conn!=**null**)  **try** {  conn.close();  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }  }    /\*\*  \* 执行更新操作  \*/  @Test  **public** **void** test2(){  Connection conn = **null**;  Statement stmt = **null**;  //声明外部变量  String name = "陈六";  **int** id=2;  **try** {  //注册驱动  Class.*forName*("com.mysql.jdbc.Driver");  //获取连接  conn = DriverManager.*getConnection*(*url*, *user*, *password*);  //创建Statment对象  stmt = conn.createStatement();  //准备sql  String sql = "UPDATE student SET NAME='"+name+"' WHERE id="+id+""; //变量和String拼接sql  //执行sql,返回结果  **int** count = stmt.executeUpdate(sql);  System.*out*.println("影响了"+count+"行");  } **catch** (Exception e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  } **finally**{  //关闭资源  **if**(stmt!=**null**)  **try** {  stmt.close();  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  **if**(conn!=**null**)  **try** {  conn.close();  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }  }    /\*\*  \* 执行删除操作  \*/  @Test  **public** **void** test3(){  Connection conn = **null**;  Statement stmt = **null**;  //声明外部变量  **int** id=2;  **try** {  //注册驱动  Class.*forName*("com.mysql.jdbc.Driver");  //获取连接  conn = DriverManager.*getConnection*(*url*, *user*, *password*);  //创建Statment对象  stmt = conn.createStatement();  //准备sql  String sql = "DELETE FROM student WHERE id="+id+""; //变量和String拼接sql  //执行sql,返回结果  **int** count = stmt.executeUpdate(sql);  System.*out*.println("影响了"+count+"行");  } **catch** (Exception e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  } **finally**{  //关闭资源  **if**(stmt!=**null**)  **try** {  stmt.close();  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  **if**(conn!=**null**)  **try** {  conn.close();  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }  }  } |

## 执行DQL查询操作

|  |
| --- |
| /\*\*  \* 使用jdbc执行DQL语句（select）  \* **@author** APPle  \*  \*/  **public** **class** Demo3 {  //数据库的连接的URL  **private** **static** String *url* = "jdbc:mysql://localhost:3306/day17";  //数据库用户名  **private** **static** String *user* = "root";  //数据库密码  **private** **static** String *password* = "root";    **public** **static** **void** main(String[] args) {  Connection conn = **null**;  Statement stmt = **null**;  ResultSet rs = **null**;  **try** {  //注册驱动  Class.*forName*("com.mysql.jdbc.Driver");  //获取连接  conn = DriverManager.*getConnection*(*url*, *user*, *password*);  //创建Statement对象  stmt = conn.createStatement();  //准备sql  String sql = "SELECT \* FROM student";    rs = stmt.executeQuery(sql);    //移动光标到下一行  //rs.next();  /\*\*  \* 注意：  \* 1）如果光标在第一行之前，使用rs.getXX()获取列值，报错：Before start of result set  \* 2）如果光标在最后一行之后，使用rs.getXX()获取列值，报错：After end of result set  \*/    //获取列值  /\*if(rs.next()){  //使用列索引    int id = rs.getInt(1);  String name = rs.getString(2);  String gender = rs.getString(3);    //使用列名称  int id = rs.getInt("id");  String name = rs.getString("name");  String gender = rs.getString("gender");  System.out.println(id+"\t"+name+"\t"+gender+"\t");  }\*/    //迭代结果集  **while**(rs.next()){  //使用列索引  /\*  int id = rs.getInt(1);  String name = rs.getString(2);  String gender = rs.getString(3);  \*/  //使用列名称  **int** id = rs.getInt("id");  String name = rs.getString("name");  String gender = rs.getString("gender");  System.*out*.println(id+"\t"+name+"\t"+gender+"\t");  }      } **catch** (Exception e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  } **finally**{  **if**(rs!=**null**)  **try** {  rs.close();  } **catch** (SQLException e1) {  e1.printStackTrace();  **throw** **new** RuntimeException(e1);  }  //关闭资源  **if**(stmt!=**null**)  **try** {  stmt.close();  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  **if**(conn!=**null**)  **try** {  conn.close();  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }    }  } |

# PreparedStatement对象执行SQL操作

|  |
| --- |
| /\*\*  \* 使用PreparedStatement执行sql语句  \* **@author** APPle  \*  \*/  **public** **class** Demo1 {  /\*\*  \* 插入操作  \*/  @Test  **public** **void** test1(){  Connection conn = **null**;  PreparedStatement stmt = **null**;  **try**{  //获取连接  conn = JdbcUtil.*getConnection*();  String sql = "INSERT INTO student(NAME,gender) VALUES(?,?)"; //预编译sql：使用？号代替参数值。一个？号代表一个参数值  //创建PreparedStatement对象,执行预编译的sql语句  stmt = conn.prepareStatement(sql);  //设置参数  /\*\*  \* 参数一： 参数位置。从1开始  \* 参数二： 参数实际值  \* 注意： 所有参数必须要赋值  \*/  stmt.setString(1, "rose");  stmt.setString(2, "女");  //发送参数，执行sql语句  **int** count = stmt.executeUpdate();  System.*out*.println(count);  }**catch**(Exception e){  e.printStackTrace();  **throw** **new** RuntimeException(e);  }**finally**{  //关闭资源  JdbcUtil.*close*(conn, stmt, **null**);  }  }    /\*\*  \* 修改操作  \*/  @Test  **public** **void** test2(){  Connection conn = **null**;  PreparedStatement stmt = **null**;  //声明变量  String name = "jacky";  **int** id = 8;  **try**{  //获取连接  conn = JdbcUtil.*getConnection*();  String sql = "UPDATE student SET NAME=? WHERE id=?"; //预编译sql：使用？号代替参数值。一个？号代表一个参数值  //创建PreparedStatement对象,执行预编译的sql语句  stmt = conn.prepareStatement(sql);  //设置参数  stmt.setString(1,name);  stmt.setInt(2, id);  //发送参数，执行sql语句  **int** count = stmt.executeUpdate();  System.*out*.println(count);  }**catch**(Exception e){  e.printStackTrace();  **throw** **new** RuntimeException(e);  }**finally**{  //关闭资源  JdbcUtil.*close*(conn, stmt, **null**);  }  }    /\*\*  \* 删除操作  \*/  @Test  **public** **void** test3(){  Connection conn = **null**;  PreparedStatement stmt = **null**;  //声明变量  **int** id = 8;  **try**{  //获取连接  conn = JdbcUtil.*getConnection*();  String sql = "DELETE FROM student WHERE id=?"; //预编译sql：使用？号代替参数值。一个？号代表一个参数值  //创建PreparedStatement对象,执行预编译的sql语句  stmt = conn.prepareStatement(sql);  //设置参数  //任何类型都可以使用setObject进行赋值  stmt.setObject(1, id);  //发送参数，执行sql语句  **int** count = stmt.executeUpdate();  System.*out*.println(count);  }**catch**(Exception e){  e.printStackTrace();  **throw** **new** RuntimeException(e);  }**finally**{  //关闭资源  JdbcUtil.*close*(conn, stmt, **null**);  }  }    /\*\*  \* 查询操作  \*/  @Test  **public** **void** test4(){  Connection conn = **null**;  PreparedStatement stmt = **null**;  //声明变量  String name = "张%";  **try**{  //获取连接  conn = JdbcUtil.*getConnection*();  String sql = "SELECT \* FROM student WHERE NAME LIKE ?";  //创建PreparedStatement，预编译sql语句  stmt = conn.prepareStatement(sql);  //设置参数  stmt.setObject(1, name);  //发送参数，执行sql，返回结果集  ResultSet rs = stmt.executeQuery();  //遍历结果集  **while**(rs.next()){  **int** id = rs.getInt("id");  String nameStr = rs.getString("name");  String gender = rs.getString("gender");  System.*out*.println(id+"\t"+nameStr+"\t"+gender+"\t");  }  }**catch**(Exception e){  e.printStackTrace();  **throw** **new** RuntimeException(e);  }**finally**{  //关闭资源  JdbcUtil.*close*(conn, stmt, **null**);  }  }    } |

# Statement和PreparedStatement的区别

一、语法结构不同

1．Statment执行静态sql语句，且sql可以拼接。

2．PreparedStatement可以先执行预编译的sql语句，在预编译sql语句中使用？进行参数占位，后面在进行参数赋值

二、原理不同

1．Statement不能进行sql缓存

2．而PreparedStatement可以进行sql缓存，执行效率会比Statement快！！！

三、安全性不同

1．Statement存在sql注入的风险

2．而PreparedStatement可以有效防止用户注入。

# CallableStatement对象执行存储过程

## 执行带输入参数的存储过程

|  |
| --- |
| /\*\*  \* 执行带有输入参数存储过程  \*/  @Test  **public** **void** test1(){  Connection conn = **null**;  CallableStatement stmt = **null**;  ResultSet rs = **null**;  **try**{  //获取连接  conn = JdbcUtil.*getConnection*();  //创建CallableStatement对象  String sql = "CALL pro\_findById(?)";//预编译sql、可以带？号  //执行预编译的sql  stmt = conn.prepareCall(sql);  //设置参数  stmt.setInt(1, 4);  //发送参数，执行sql,返回结果  rs = stmt.executeQuery();// 注意： 执行存储过程必须使用exeuteQuery！！！！  //遍历结果  **while**(rs.next()){  **int** id = rs.getInt("id");  String name = rs.getString("name");  String gender = rs.getString("gender");  System.*out*.println(id+"\t"+name+"\t"+gender+"\t");  }  }**catch**(Exception e){  e.printStackTrace();  **throw** **new** RuntimeException(e);  }**finally**{  //关闭资源  JdbcUtil.*close*(conn, stmt, rs);  }  } |

## 执行带有输出参数的存储过程

|  |
| --- |
| /\*\*  \* 执行带有输出参数存储过程  \*/  @Test  **public** **void** test2(){  Connection conn = **null**;  CallableStatement stmt = **null**;  **try**{  //获取连接  conn = JdbcUtil.*getConnection*();  String sql = "CALL pro\_findById2(?,?)"; // 第一个参数时输入参数，第二个参数是输出参数  //创建CallableStatement对象  stmt = conn.prepareCall(sql);  //设置输入参数  stmt.setObject(1, 4);  //注册一个输出参数  /\*\*  \* 参数一： 参数位置  \* 参数二： 表示存储过程中的OUT参数的数据库类型  \*/  stmt.registerOutParameter(2, java.sql.Types.*VARCHAR*);    //发送参数,执行存储过程  stmt.executeQuery();    /\*\*  \* 如何获取存储过程的返回值：OUT参数值。使用getXXX方法  \*/  String name = stmt.getString(2);//和预编译语句中的参数位置保持一致！！！！  System.*out*.println("结果："+name);  }**catch**(Exception e){  e.printStackTrace();  **throw** **new** RuntimeException(e);  }**finally**{  //关闭资源  JdbcUtil.*close*(conn, stmt, **null**);  }  } |

# 使用类路径加载资源文件

|  |
| --- |
| /\*\*  \* 使用类路径的方式加载db.properties文件  \*/  //1.得到字节码对象  Class clazz = JdbcUtil.**class**;  //2.通过方法加载文件，使用类路径方式  // / : 表示类路径的根目录（放class字节码文件目录）  // java项目： bin目录下  // web项目： WEB-INF/classes目录下  InputStream in = clazz.getResourceAsStream("/db.properties");  // 从类路径的根目录下开始找  //InputStream in = clazz.getResourceAsStream("db.properties");  // 文件和当前类处于同一个目录  prop.load(in); |

Class.getResourceAsStream("/db.properties");

Class.getResourceAsStream("db.properties");

这里的 斜杠 / 表示当前项目的类路径的根目录。

当前项目是java项目，/ 在项目的bin目录下

当前项目是web项目，/在项目的WEB-INF/classes目录下

注意： 无论是java项目还是web项目，在开发时如果把资源文件放在src下，那么src下的文件都会拷贝到类路径的根目录下。

相对路径： . 这个点代表当前目录。当前目录本质上是java命令运行的目录

java项目： 在ecplise中，当前目录指向项目的根目录。

web项目： 当前目录指向%tomcat%/bin目录

1．结论： 在web项目不能使用相对路径

web项目中加载配置文件： ServletContext.getRealPath() / getResourceAsStream() 这种方式对于jdbcUtil这个工具而言，放到java项目中找不到ServletContext对象，不通用的！

2．不能使用ServletContext读取文件

3．使用类路径方式读取配置文件

使用类路径方式读取配置文件：

Class clazz = JdbcUtil.**class**;

InputStream in = clazz.getResourceAsStream("/db.properties");

总结：

1．jdbc：使用java程序操作数据库的技术。（一套接口）

2．jdbc步骤：

2.1 注册驱动

Class.forName(驱动类名称)

2.2 获取连接

Connection conn = DriverManger.getConnection(url,user,password);

2.3 创建Statement/PreparedStatement/CallableStament

conn.createStatement()

conn.preparedStaement(sql);

conn.preparedCall(sql)

2.4 如果使用PreparedStatement/CallableStament，设置参数

stmt.setXXX(参数位置,参数值)

2.5 执行sql/发送参数

stmt.executeUpdate() DDL+DML

ResultSet rs = stmt.executeQuery() DQL

2.6 处理结果集

rs.next()

rs.getXXX() 列索引和列名称

2.7 关闭资源

rs.close();

stmt.close();

conn.close();

# jdbc执行批处理

## 引入

之前：使用jdbc每次只能发送1条sql语句,如果同时执行2000次插入，向数据库服务器发送2000条插入语句，这种做法的效率不高！

现在：使用jdbc的批处理执行2000次插入，但每次向数据库的服务器发送20条sql，一共只需要发送100次就可以了！那么这种的做法的效率会比之前的做法高！！！

**批处理：一次发送多条SQL语句**

## 批处理的API

Statement的批处理：

void addBatch(String sql) 把sql添加到缓存区中(没有发送的)

int[] executeBatch() 执行批处理缓存中sql语句(发送到数据库执行)

void clearBatch() 清空缓存区sql语句

PreparedStatement的批处理：

void addBatch() 把参数添加到缓存区中

int[] executeBatch() 执行批处理缓存中所有参数(发送到数据库执行)

void clearBatch() 清空缓存区参数列表

### 例：

/\*\*

\* 使用批处理的PrepaedStatement的情况

\* mysql:耗时为：9379 oracle: 耗时为：1391

\*/

public static void testByPreparedStaementBatch(){

Connection conn = null;

PreparedStatement stmt = null;

try{

conn = JdbcUtil.getConnection();

stmt = conn.prepareStatement("INSERT INTO student VALUES(?,?,?,?)");

for(int i=1;i<=2000;i++){

//参数赋值

stmt.setInt(1, i);

stmt.setString(2, "张三");

stmt.setInt(3, 20);

stmt.setString(4, "男");

//把参数添加到缓存区

stmt.addBatch();

//每20次发送一次参数

if(i%20==0){

//执行批处理命令

stmt.executeBatch();

//清空缓存区的参数

stmt.clearBatch();

}

}

}catch(Exception e){

e.printStackTrace();

}finally{

JdbcUtil.close(stmt, conn);

}

}

# Jdbc处理大容量数据

## 什么是大容量

**mysql:**

字符串： varchar char 65535

大文本数据： tinytext ， longtext ，text

字节： bit

大字节文件： tinyblob（255byte）， blob（64kb），MEDIUMBLOB（约16M） longblob（4GB）

**oracle：**

字符串： varchar2 char 65535

大文本数据： clob

字节： bit

大字节文件： blob

### 字符：

存储字符内容： mysql： char varchar 长度有限的。 65535

大容量的字符字段：

mysql： text longtext(4G字符内容)

oracle : clob longclob

#### 例：

|  |
| --- |
| /\*\*  \* 对大文本数据处理  \* **@author** APPle  \*  \*/  **public** **class** Demo1 {  /\*\*  \* 文件保存到数据中  \*/  @Test  **public** **void** testWrite(){  Connection conn = **null**;  PreparedStatement stmt = **null**;  **try**{  //获取连接  conn = JdbcUtil.*getConnection*();  //创建PreparedStatement  String sql = "INSERT INTO test1(content) VALUES(?)";  stmt =conn.prepareStatement(sql);  //设置参数  /\*\*  \* 参数一： 参数位置  \* 参数二： 输入字符流  \*/  /\*\*  \* 读取本地文件，返回输入字符流  \*/  FileReader reader = **new** FileReader(**new** File("e:/Demo1.java"));  stmt.setClob(1, reader);  //执行sql  **int** count = stmt.executeUpdate();  System.*out*.println("影响了"+count+"行");  }**catch**(Exception e){  e.printStackTrace();  }**finally**{  JdbcUtil.*close*(conn, stmt, **null**);  }  }    /\*\*  \* 从数据中读取文本内容  \*/  @Test  **public** **void** testRead(){  Connection conn = **null**;  PreparedStatement stmt = **null**;  ResultSet rs = **null**;  **try**{  //获取连接  conn = JdbcUtil.*getConnection*();  String sql = "SELECT \* FROM test1 where id=?";  stmt = conn.prepareStatement(sql);  //设置参数  stmt.setInt(1, 2);  //执行sql,返回结果集  rs = stmt.executeQuery();  **if**(rs.next()){  //方式一：当做字符串取出数据  /\*  String content = rs.getString("content");  System.out.println(content);  \*/    //方式二：返回输入流形式  Clob clob = rs.getClob("content");  Reader reader = clob.getCharacterStream();  //写出到文件中  FileWriter writer = **new** FileWriter(**new** File("e:/Demo2.java"));  **char**[] buf = **new** **char**[1024];  **int** len = 0;  **while**( (len=reader.read(buf))!=-1){  writer.write(buf, 0, len);  }  //关闭流  writer.close();  reader.close();  }      }**catch**(Exception e){  e.printStackTrace();  }**finally**{  JdbcUtil.*close*(conn, stmt, rs);  }  }  } |

### 字节：

mysql： blob（65kb） mediumblob（16mb） longblog（4GB）

oracle: blob

#### 例1：

|  |
| --- |
| /\*\*  \* 对字节文件处理  \* **@author** APPle  \*  \*/  **public** **class** Demo2 {  /\*\*  \* 文件保存到数据库中  \*/  @Test  **public** **void** testWrite(){  Connection conn = **null**;  PreparedStatement stmt = **null**;  **try**{  //获取连接  conn = JdbcUtil.*getConnection*();  String sql = "insert into test2(content) values(?)";  stmt = conn.prepareStatement(sql);  //设置参数  /\*\*  \* 参数一：参数位置  \* 参数二:输入字节流  \*/  /\*\*  \* 读取本地文件  \*/  InputStream in = **new** FileInputStream(**new** File("e:/abc.wmv"));  //stmt.setBlob(1, in);  stmt.setBinaryStream(1, in);  //执行  stmt.executeUpdate();  }**catch**(Exception e){  e.printStackTrace();  }**finally**{  JdbcUtil.*close*(conn, stmt, **null**);  }  }    /\*\*  \* 注意： mysql数据库默认情况下，只能存储不超过1m的文件，由于max\_allowed\_packet变量的限制  \* 可以修改： %mysql%/my.ini文件, 修改或添加max\_allowed\_packet变量，然后重启mysql即可！！  \*/    /\*\*  \* 从数据中读取字节内容  \*/  @Test  **public** **void** testRead(){  Connection conn = **null**;  PreparedStatement stmt = **null**;  ResultSet rs = **null**;  **try**{  //获取连接  conn = JdbcUtil.*getConnection*();  String sql = "SELECT \* FROM test2 where id=?";  //获取PreparedStatement  stmt = conn.prepareStatement(sql);  //设置参数  stmt.setInt(1, 1);  //执行sql  rs = stmt.executeQuery();  **if**(rs.next()){  //返回输入流  //InputStream in = rs.getBinaryStream("content");  InputStream in = rs.getBlob("content").getBinaryStream();  //写出文件中  FileOutputStream out = **new** FileOutputStream(**new** File("e://3.jpg"));  **byte**[] buf = **new** **byte**[1024];  **int** len = 0;  **while**((len=in.read(buf))!=-1){  out.write(buf, 0, len);  }  //关闭流  out.close();  in.close();  }  }**catch**(Exception e){  e.printStackTrace();  }**finally**{  JdbcUtil.*close*(conn, stmt, rs);  }  }  } |

# Jdbc获取自增长的值

## 如何获取自增长的生成的值?

1．设置是否可以返回自增长的值

RETURN\_GENERATED\_KEYS: 可以返回自增长生成值

NO\_GENERATED\_KEYS:不能返回自增长的值

2．获取自增长的值

注意： 该ResultSet只包含了自增长的字段

## 例：

|  |
| --- |
| /\*\*  \* 获取自增长值  \* **@author** APPle  \*  \*/  **public** **class** Demo1 {  @Test  **public** **void** test(){  //插入部门表数据  String deptSql = "INSERT INTO dept(deptName) VALUES(?)";  //插入员工表数据  String empSql = "INSERT INTO employee(empName,deptId) VALUES(?,?)";  Connection conn = **null**;  PreparedStatement stmt = **null**;  ResultSet rs = **null**;  **try**{  //获取连接  conn = JdbcUtil.*getConnection*();    /\*\*  \* 1）插入部门  \*/  //预编译部门sql  //stmt = conn.prepareStatement(deptSql);  //  /\*\*  \* 1.1 使用两个参数的prepareStatement（）方法，指定可以返回自动增长的键值  \* Statement.RETURN\_GENERATED\_KEYS: 可以返回自动增长值  \* Statement.NO\_GENERATED\_KEYS: 不能返回自动增长值  \*/  stmt = conn.prepareStatement(deptSql, Statement.*RETURN\_GENERATED\_KEYS*);  //设置参数  stmt.setString(1, "秘书部");  //执行部门sql  stmt.executeUpdate();  /\*\*  \* 1.2 获取自增长的值  \*/  rs = stmt.getGeneratedKeys();  Integer deptId = **null**;  **if**(rs.next()){  deptId = rs.getInt(1);//得到第一行第一列的值  }    /\*\*  \* 2)插入员工，员工在刚添加的部门中  \*/  stmt = conn.prepareStatement(empSql);  //设置参数  stmt.setString(1, "李四");  stmt.setInt(2, deptId); //如何获取刚刚添加的部门ID？？  //执行员工sql  stmt.executeUpdate();    }**catch**(Exception e){  e.printStackTrace();  }**finally**{  JdbcUtil.*close*(conn, stmt, rs);  }  }  } |

# 数据库事务

## 什么是事务？

所谓的事务，如果把多条sql语句看做一个事务，那么这个事务要么一起成功，要么一起失败！！

## mysql事务操作命令

set autocommit =0 / 1; 设置是否自动提交事务

1： 表示自动提交事务，每执行一条sql语句，自动提交事务。

0： 表示关闭自动提交事务。

start transaction; 开启事务

commit; 提交事务，一旦提交事务不能回滚

rollback; 回滚事务。回滚到事务的起始点。

## jdbc事务操作

Connection.setAutoCommit(false) 开启事务

Connection.commit(); 成功执行，最后提交事务

Connection.rollback(); 一旦遇到错误，回滚事务

|  |
| --- |
| /\*\*  \* 模拟银行转账  \* **@author** APPle  \*  \*/  **public** **class** Demo1 {  /\*\*  \* 从eric的账户转2000元到rose的账户上  \*/  @Test  **public** **void** testTransfer(){  //从eric账户上扣除2000元  String delSql = "UPDATE account SET BALANCE=BALANCE-2000 WHERE NAME='eric'";  //向rose账户打入2000元  String addSql = "UPDATE account SET BALANCE=BALANCE+2000 WHERE NAME='rose'";  Connection conn = **null**;  PreparedStatement stmt = **null**;  **try**{  //获取连接  conn = JdbcUtil.*getConnection*();  //开启事务  conn.setAutoCommit(**false**); // 等价于set autocommit=0;    //预编译delSQL  stmt = conn.prepareStatement(delSql);  //执行delSQL  stmt.executeUpdate();    //发生异常  **int** i = 100/0;    //预编译addSql  stmt = conn.prepareStatement(addSql);  //执行addSql  stmt.executeUpdate();    //提交事务  conn.commit();// 等价于commit;    }**catch**(Exception e){  e.printStackTrace();  //回滚事务  **try** {  conn.rollback(); // 等价于rollback;  } **catch** (SQLException e1) {  e1.printStackTrace();  }  }**finally**{  JdbcUtil.*close*(conn, stmt, **null**);  }  }  } |

## 事务的四大特性 （ACID）

**原子性**： 一个事务的操作要么一起成功，要么一起失败！

**一致性**： 在一个事务执行后，数据库从一个一致性的状态转变为另一个一致性。

**隔离性**： 多个事务并发过程中，希望能够相互隔离的。

多个事务在并发产生以下的现象：

**脏读：** 一个事务读到了另一个事务没有提交的数据

**不可重复读：**一个事务读到了另一个事务已经提交的更新数据（update）

**幻读：** 一个事务读到了另一个事务已经提交的插入数据（insert）

数据库可以设置不同的隔离级别来防止以上现象：（是否能够防止现象）

脏读 不可重复读 幻读

read uncommitted： 否 否 否

read committed： 是 否 否 oracle默认的隔离级别

repeatable-read： 是 是 否 mysql默认的隔离级别

serializable： 是 是 是

**持久性： 事务一旦提交，应该永久保存下来**

总结：

1）批处理

2）大容量数据（应用）

3）获取自增长的值（了解）

4）事务

4.1 什么是事务？

4.2 数据库事务如何使用？

4.3 jdbc中如何使用事务？

4.4 事务有哪些特性？

# 连接池

## 什么是连接池？

之前jdbc步骤： 获取连接 -> 得到Statement-> 发送sql -> 关闭连接

通讯录系统：

CURD 四次

问题： 获取连接对象需要消耗比较多的资源，而每次操作都要重新获取新的连接对象，执行一次操作就把连接关闭，这样连接对象的使用效率并不高！！！

办法： 能不能让连接对象反复使用？？？如果可以的话，连接对象使用率提高！

这就可以用连接池实现！！

## 自定义连接池

自定义连接池实现：

1．连接池中初始化若干个连接对象

2．当连接池中连接不足，重新连接数据库获取新的连接对象

3．总连接对象不能超过最大连接数

|  |
| --- |
| /\*\*  \* 自定义的连接池  \* **@author** APPle  \*  \*/  **public** **class** MyPool {  //连接参数  **private** **static** String *url* = "jdbc:mysql://localhost:3306/day19";  **private** **static** String *user* = "root";  **private** **static** String *password* = "root";  **private** **static** String *driverClass* = "com.mysql.jdbc.Driver";  //使用集合存储多个连接对象  LinkedList<Connection> pool = **new** LinkedList<Connection>();  /\*\*  \* 连接池的初始连接数  \*/  **private** **int** initSize = 5;  /\*\*  \* 连接池的最大连接数  \*/  **private** **int** maxSize = 8;  /\*\*  \* 连接池中的当前连接数,用于记录连接中有几个连接对象？  \*/  **private** **int** currentSize = 0;    **public** MyPool(){  /\*\*  \* 创建若干个连接对象，放入到集合中  \*/  **for**(**int** i=1;i<=initSize;i++){  pool.addLast(createConnection());  currentSize++;  }  }    **static**{  **try** {  //注册驱动  Class.*forName*(*driverClass*);  } **catch** (ClassNotFoundException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }    /\*\*  \* 创建连接方法  \*/  **private** Connection createConnection(){  **try** {  Connection conn = DriverManager.*getConnection*(*url*, *user*, *password*);  **return** conn;  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }    /\*\*  \* 对外提供获取连接的方法  \*/  **public** Connection getConnection(){  /\*\*  \* 1)判断连接池中是否还有连接对象，池中有连接，直接取出池中连接  \*/  **if**(pool.size()>0){  /\*\*  \* 直接返回连接池中的一个连接对象  \*/  **return** pool.removeFirst();  }    /\*\*  \* 2）如果初始连接数不足的情况下，需要获取新的连接对象。但不能超过最大连接数  \*/  **if**(currentSize<maxSize){  Connection conn = createConnection();  currentSize++;  **return** conn;  }    /\*\*  \* 3)超过最大连接数时，需要处理（抛出异常或让用户等待）  \*/  **throw** **new** RuntimeException("超过了最大连接数，请稍后再来");  }    /\*\*  \* 对外提供释放连接的方法  \*/  **public** **void** releaseConnection(Connection conn){  /\*\*  \* 把用完的连接对象放回到连接池中  \*/  pool.addLast(conn);  }    } |

## 解决Connection.close方法释放连接问题

1．静态代理：

1.1 创建Connection接口的实现类（代理类）

1.2 在代理类中，重写close方法

注意： 使用静态代理，需要把原有对象的方法全部实现。

2．动态代理：

使用jdk自带的动态代理类

**Proxy类**

用于创建动态代理类对象：

static Object newProxyInstance(

ClassLoader loader,

Class<?>[] interfaces,

InvocationHandler h ) {

Object invoke(

Object proxy, 代理类对象

Method method, 代理类对象调用的方法。

Object[] args 调用代理类对象方法时传入的参数列表 ){

}

}

参数一：类加载器。

参数二：代理类实现的接口列表

参数三： 接口 InvocationHandler： 代理类的调用处理程序的接口。（代理完代理对象之后，对其中的方法如何处理？？？）

Object invoke(

Object proxy, 代理类对象

Method method, 代理类对象调用的方法。

Object[] args 调用代理类对象方法时传入的参数列表

)

如果以上自定义连接要拿到实际应用中使用，还需要不断优化。

1．比如线程并发问题

2．达到最大连接数应该先等待，再抛出异常

3．当连接空闲超过一定时间，连接池要回收连接

....

所以这时就使用别人写的连接池工具！！

# 连接池工具

## DBCP连接池

DBCP: DataBase Connection Pool

特点：

1．Apache旗下的软件，开源连接池  
 2．Tomcat的服务器的连接池默认实现

使用步骤：

1．导入dbcp的jar包

commons-dbcp-1.4.jar 核心包

commons-pool-1.5.6.jar 辅助包

2．创建连接池对象BasicDataSource对象

3．设置连接参数（url，user，password，dirverClass）

4．设置连接池参数（初始连接数，最大连接数，最大等待时间）

5．获取连接对象（getConnection() 方法）

|  |
| --- |
| **public** **void** test1(){  **try** {  //1.创建连接池对象  BasicDataSource ds = **new** BasicDataSource();  //2.设置连接参数  ds.setUrl("jdbc:mysql://localhost:3306/day19");  ds.setUsername("root");  ds.setPassword("root");  ds.setDriverClassName("com.mysql.jdbc.Driver");    //3.设置连接池参数  ds.setInitialSize(5);//初始连接数  ds.setMaxActive(8);//最大连接数  ds.setMaxWait(3000);//超过最大连接数时，最大等待时间  ds.setMaxIdle(3000);//最大空闲时间    //4.获取连接  **for**(**int** i=1;i<=9;i++){  Connection conn = ds.getConnection();  System.*out*.println(conn.hashCode());  **if**(i==5){  //释放连接(不是真正的关闭连接对象，而是把连接对象放回连接池)  conn.close();  }  }  } **catch** (SQLException e) {  e.printStackTrace();  }    } |

### 使用配置文件加载

#### 1．配置文件(dbcp.properties)

|  |
| --- |
| url=jdbc:mysql://localhost:3306/day19  username=root  password=root  driverClassName=com.mysql.jdbc.Driver    initialSize=5  maxActive=8  maxWait=3000  maxIdle=3000 |

#### 2．程序

|  |
| --- |
| @Test  **public** **void** test2(){  **try** {  Properties prop = **new** Properties();  prop.load(Demo.**class**.getResourceAsStream("/dbcp.properties"));  //1.创建连接池对象  BasicDataSource ds = (BasicDataSource)BasicDataSourceFactory.*createDataSource*(prop);    //2.获取连接  **for**(**int** i=1;i<=9;i++){  Connection conn = ds.getConnection();  System.*out*.println(conn.hashCode());  **if**(i==5){  //释放连接(不是真正的关闭连接对象，而是把连接对象放回连接池)  conn.close();  }  }  }**catch** (Exception e) {  e.printStackTrace();  }  } |

## C3P0连接池

特点：

1．开源连接池

2．Hibernate框架，默认推荐使用C3P0作为连接池实现

使用步骤：

1．导入c3p0的jar包

c3p0-0.9.1.2.jar 核心包

2．创建连接池对象ComboPooledDataSource对象

3．设置连接参数（url，user，password，dirverClass）

4．设置连接池参数（初始连接数，最大连接数，最大等待时间）

5．获取连接对象（getConnection() 方法）

|  |
| --- |
| /\*\*  \* 使用C3p0连接池  \* **@author** APPle  \* 注意： DataSource接口是Sun公司设计的用于规范连接池实现的接口。  \*  \*/  **public** **class** Demo {  @Test  **public** **void** test1(){  **try** {  //1.创建连接池对象  ComboPooledDataSource ds = **new** ComboPooledDataSource();  //2.设置连接参数  ds.setDriverClass("com.mysql.jdbc.Driver");  ds.setJdbcUrl("jdbc:mysql://localhost:3306/day19");  ds.setUser("root");  ds.setPassword("root");    //3.设置连接池参数  ds.setInitialPoolSize(5);// 初始化连接数  ds.setMaxPoolSize(8);//最大连接数  ds.setCheckoutTimeout(3000);//最大等待时间    //4.获取连接  **for**(**int** i=1;i<=9;i++){  Connection conn = ds.getConnection();  System.*out*.println(conn);  **if**(i==5){  /\*\*  \* 把连接对象放回连接池  \*/  conn.close();  }  }  } **catch** (Exception e) {  e.printStackTrace();  }  }    /\*\*  \* 使用配置文件方式读取参数信息  \* 注意： c3p0会默认查询类路径的c3p0-config.xml文件，文件名不能错！！  \*/  @Test  **public** **void** test2(){  **try** {  //1.创建连接池对象（方式一： 使用默认配置（default-config））  //ComboPooledDataSource ds = new ComboPooledDataSource();    //2.创建连接池对象（方式二： 使用命名配置（named-config：mysql\_day18））  ComboPooledDataSource ds = **new** ComboPooledDataSource("mysql\_day18");    //2.获取连接  **for**(**int** i=1;i<=9;i++){  Connection conn = ds.getConnection();  System.*out*.println(conn);  **if**(i==5){  /\*\*  \* 把连接对象放回连接池  \*/  conn.close();  }  }  } **catch** (SQLException e) {  e.printStackTrace();  }  }  } |

# BeanUtils工具

## BeanUtils工具的作用

BeanUtils工具可以方便开发者对象javabean进行操作

### JavaBean规范：

1．必须有无参的构造方法

2．把属性进行私有化

3．提供公开的getter和setter方法

### 使用步骤：

1．导入beanutills的jar包

commons-beanutils-1.8.3.jar 核心包

commons-logging-1.1.3.jar 辅助包

## JavaBean操作

### 1．对JavaBean的属性进行赋值

|  |
| --- |
| /\*\*  \* 1）对javabean的属性操作  \*/  @Test  **public** **void** test1() **throws** Exception{  /\*  //1.创建javabean对象  Student student = new Student();  //2.赋值  student.setId(1);  student.setName("eric");  student.setGender(true);  student.setScore(79.43);  student.setBirth(new Date());  \*/    //1.使用反射创建javabean对象  Object student = Class.*forName*("gz.itcast.d\_beanutils.Student").newInstance();  //2.对javabean进行赋值  /\*\*  \* 该方法是给Javabean的一个属性赋值  \* 参数一：需要赋值的javabean对象  \* 参数二： 需要赋值的属性名称  \* 参数三： 属性值  \* 注意：  \* 1)如果基本数据类型（Integer,Double,Float,Long,Char,Boolean），可以由字符串类型自动转换过去  \*/  /\*\*  \* 注册一个日期转换器  \* 参数一： 注册的转换器对象  \* 参数二： 转换后的目标类型  \*/  ConvertUtils.*register*(**new** DateLocaleConverter(), java.util.Date.**class**);  BeanUtils.*copyProperty*(student, "name", "jacky");//给Student的name属性赋值  BeanUtils.*copyProperty*(student, "id", "2");  BeanUtils.*copyProperty*(student, "gender", "true");  BeanUtils.*copyProperty*(student, "score", 84.78);  BeanUtils.*copyProperty*(student, "birth", "2014-10-01"); //默认情况下，不能把字符串的日期转换为Date类型  /\*\*  \* 注意：  \* 1）拷贝的方法能够自动将非String的类型转换对应的类型的属性.例如 ：  \* "2"（String） -> int(String->int)  \* "97.23"(String) -> double (String->double)  \* 2）如果要支持从String转为Date，那么需要手动注册一个日期转换器  \*/  System.*out*.println(student);  } |

### 2．对JavaBean的对象赋值

|  |
| --- |
| /\*\*  \* 拷贝整个javabean的对象（所有属性一次性拷贝）  \* **@throws** Exception  \*/  @Test  **public** **void** test2() **throws** Exception{  Student student = **new** Student();  student.setId(1);  student.setName("eric");  student.setGender(**true**);  student.setScore(79.43);  student.setBirth(**new** Date());    //1.使用反射构造对象  Object student2 = Class.*forName*("gz.itcast.d\_beanutils.Student").newInstance();  //2.把student的属性拷贝到student2中  /\*\*  \* 参数一： 目标的javabean  \* 参数二： 拷贝的源javabean  \*/  BeanUtils.*copyProperties*(student2, student);    System.*out*.println(student2);  } |

### 3．把Map数据赋值给JavaBean对象

|  |
| --- |
| /\*\*  \* 把map集合的数据拷贝到javabean中  \* **@throws** Exception  \*/  @Test  **public** **void** test3() **throws** Exception{  Map map = **new** HashMap();  map.put("id", 2);  map.put("name", "rose");  map.put("gender", **true**);  map.put("score", 90.32);  map.put("birth", **new** Date());    //1.使用反射构造对象  Object student2 = Class.*forName*("gz.itcast.d\_beanutils.Student").newInstance();  //2.把map数据拷贝到student2中  // 1）只拷贝javabean存在的哪些属性(setXXX方法)  // 2)需要拷贝的数据是数组类型，那么只拷贝数组中的第一个元素。  BeanUtils.*copyProperties*(student2, map);    System.*out*.println(student2);  } |

## 应用案例

在web项目中，经常需要拷贝请求中的参数到JavaBean对象中，这时可以使用BeanUtils工具，把请求的Map结构的所有参数一次拷贝到javaben中。

|  |
| --- |
| /\*\*  \* 优化后的方法,请求参数拷贝到javabean对象中  \* **@param** request  \* **@param** obj  \*/  **public** **static** **void** copyRequestToBean(HttpServletRequest request,Object obj){  System.*out*.println("优化后的拷贝方法...");  //1.得到请求的所有参数  /\*\*  \* map： 键是参数的名称，值是参数内容  \*/  Map requestMap = request.getParameterMap();  //2.把请求map对象拷贝到obj中  **try** {  **BeanUtils.*copyProperties*(obj, requestMap);**  }**catch** (Exception e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  } |

# 元数据的使用

## 什么是元数据？

希望知道以下：

### 一、数据库相关的信息：（封装到数据库元对象： DataBaseMetaData）

|  |
| --- |
| /\*\*  \* 数据库元数据  \* @author APPle  \*  \*/  ComboPooledDataSource ds = new ComboPooledDataSource();    Connection conn = ds.getConnection();    //System.out.println(conn);    /\*\*  \* **使用数据库元数据（DatabaseMetaData）**  \*/  DatabaseMetaData dbmd = conn.getMetaData();    //**指定哪个数据库**  **System.out.println(dbmd.getDatabaseProductName());**    //**数据库版本**  **System.out.println(dbmd.getDatabaseMajorVersion());**  **System.out.println(dbmd.getDatabaseMinorVersion());**    //**使用驱动程序的版本**  **System.out.println(dbmd.getDriverMajorVersion());**  **System.out.println(dbmd.getDriverMinorVersion());** |

### 二、参数相关的信息：(封装到参数元对象：ParameterMetaData)

|  |
| --- |
| /\*\*  \* 参数元数据（ParameterMetaData）  \*/  ComboPooledDataSource ds = new ComboPooledDataSource();  Connection conn = ds.getConnection();  String sql = "insert into student(id,name,gender,scroe,birth) values(?,?,?,?,?)";    //预编译  PreparedStatement stmt = conn.prepareStatement(sql);    //参数赋值  /\*stmt.setInt(1, 1);  stmt.setString(2, "eric");  stmt.setString(3, "男");  stmt.setDouble(4, 87.65);  stmt.setString(5, "2015-06-24");\*/    //知道预编译的sql有几个参数  //**得到参数元数据**  **ParameterMetaData pmd = stmt.getParameterMetaData();**  //**得到sql中有几个参数**  **int count = pmd.getParameterCount();**    //System.out.println(count);    //**可以把参数值放入数组中**  **Object[] values = new Object[]{2,"jacky","女",90.23,"2015-06-25"};**  **//新的赋值方式**  **for(int i=1;i<=count;i++){**  **//setObject可以设置任何类型参数**  **stmt.setObject(i, values[i-1]);**  **}** |

### 三、结果相关的信息：(封装到结果集元对象： ResultSetMetaData)

|  |
| --- |
| /\*\*  \* 结果集元数据（ParameterMetaData）  \*/  ComboPooledDataSource ds = new ComboPooledDataSource();  Connection conn = ds.getConnection();    String sql = "select \* from student";  PreparedStatement stmt = conn.prepareStatement(sql);  ParameterMetaData pmd = stmt.getParameterMetaData();  //得到参数数量  int paramCount = pmd.getParameterCount();  //Object[] values = new Object[]{1};  Object[] values = null;  if(values!=null){  //参数赋值  for(int i=1;i<=paramCount;i++){  stmt.setObject(i, values[i-1]);  }  }  ResultSet rs = stmt.executeQuery();    //遍历结果集  /\*while(rs.next()){// 行  //得到列值  int id = rs.getInt(1);  String name = rs.getString(2);  System.out.println(id+"\t"+name);  }\*/    //问题:得到表有几列，每个列的名称是什么  //得到结果集的元数据  ResultSetMetaData rsmd = rs.getMetaData();  //得到列数量  int columnCount = rsmd.getColumnCount();    while(rs.next()){//行  //遍历每列  for(int i=1;i<=columnCount;i++){  Object value = rs.getObject(i);  System.out.print(value+"\t");  }  System.out.println();  } |

### jdbc步骤：

1．连接数据库，获取Connection对象 （得到DataBaseMetaData）

2．创建Statment，预编译sql语句 （得到ParameterMetaData）

3．设置参数

4．执行sql，返回结果集 （得到ResultSetMetaData）

5．遍历结果集

6．关闭连接

## 提示：使用元数据

|  |
| --- |
| /\*\*  \* jdbc的工具方法  \* @author APPle  \*  \*/  public class SQLUtil {  \* 通用的更新方法  \*/  public static void update(String sql,Object[] values){  try {  ComboPooledDataSource ds = new ComboPooledDataSource();  Connection conn = ds.getConnection();  PreparedStatement stmt = conn.prepareStatement(sql);    ParameterMetaData pmd = stmt.getParameterMetaData();  //得到sql中有几个参数  int count = pmd.getParameterCount();    //可以把参数值放入数组中  //新的赋值方式  for(int i=1;i<=count;i++){  //setObject可以设置任何类型参数  stmt.setObject(i, values[i-1]);  }    stmt.executeUpdate();  } catch (SQLException e) {  e.printStackTrace();  }  }    /\*\*  \* 通用的查询方法  \*/  public static <T> List<T> query(String sql,Object[] values,Class<T> clazz){  try {  ComboPooledDataSource ds = new ComboPooledDataSource();  Connection conn = ds.getConnection();  PreparedStatement stmt = conn.prepareStatement(sql);  ParameterMetaData pmd = stmt.getParameterMetaData();  //得到参数数量  int paramCount = pmd.getParameterCount();  //如果是null，则不赋值了  if(values!=null){  //参数赋值  for(int i=1;i<=paramCount;i++){  stmt.setObject(i, values[i-1]);  }  }  ResultSet rs = stmt.executeQuery();  //问题:得到表有几列，每个列的名称是什么  //得到结果集的元数据  ResultSetMetaData rsmd = rs.getMetaData();  //得到列数量  int columnCount = rsmd.getColumnCount();  List list = new ArrayList();  while(rs.next()){//行  //每一行数据就是一个对象  //构造对象  Object obj = clazz.newInstance();  //遍历每列  for(int i=1;i<=columnCount;i++){  //得到表的列值  Object value = rs.getObject(i);  //通过结果集元数据可以得到字段名称  String columnName = rsmd.getColumnName(i);  //表中的每列的值就封装到对象的每个属性中  //约定一个前提： 表中的每个字段名称和javabean对象中属性名称保持一致！！！！  //把值拷贝到javabean对象中  BeanUtils.copyProperty(obj, columnName, value);  }  list.add(obj);  }  return list;  } catch (Exception e) {  e.printStackTrace();  throw new RuntimeException(e);  }  }  } |

# DBUtls工具

## 简介DBUtil工具

DBUtils是Apache的工具，是一个对jdbc的简单封装的工具。提供了一些通用的jdbc操作方法。

## 使用步骤

1)导入jar包

commons-dbutils-1.2.jar

2）使用的API

**QueryRunner类**： 通过此类可以执行更新操作或者查询操作。

update(.....): 用于更新操作（DDL、DML）

query(.....): 用于查询操作（DQL）

**ResultSetHandler接口**：用于封装查询之后的结果。

Object handle(ResultSet rs) ： 用于封装数据

## 常用的实现类：

[ArrayHandler](file:///C:\Users\APPle\Desktop\day19_dbutils\03.资料和工具\02.DBUtils-1.2\apidocs\org\apache\commons\dbutils\handlers\ArrayHandler.html)： 把结果集的第一行的数据封装成对象数组。

ArrayListHandler：把结果集的每一行数据封装对象数组，把这个对象数组放入List中

**BeanHandler**： 把结果集的第一行数据封装成javabean

**BeanListHandler:** 把结果集的每一行数据封装成javabean，把这个javabean放入LIst中

**ScalarHandler**： 把结果集的第一行第一列取出。通常用于聚合函数查询。例如（count()/max()）

如果表的字段名称和javabean的属性名称不一致时，需要自定义ResultSetHandler的实现类

### 1．更新方法

|  |
| --- |
| /\*\*  \* 使用dbutils工具  \* @author APPle  \*  \*/  public class Demo1 {  public void testInsert() throws Exception{  ComboPooledDataSource ds = new ComboPooledDataSource();  **//1.创建QueryRunner对象**  **QueryRunner qr = new QueryRunner(ds);**  **//2.执行操作**  //qr.update("INSERT INTO student(NAME,age,address) VALUES('张三11',20,'广州天河')");  **qr.update("INSERT INTO student(NAME,age,address) VALUES(?,?,?)", new Object[]{"eric11",20,"广州天河"});**  }  } |

### 2．查询方法

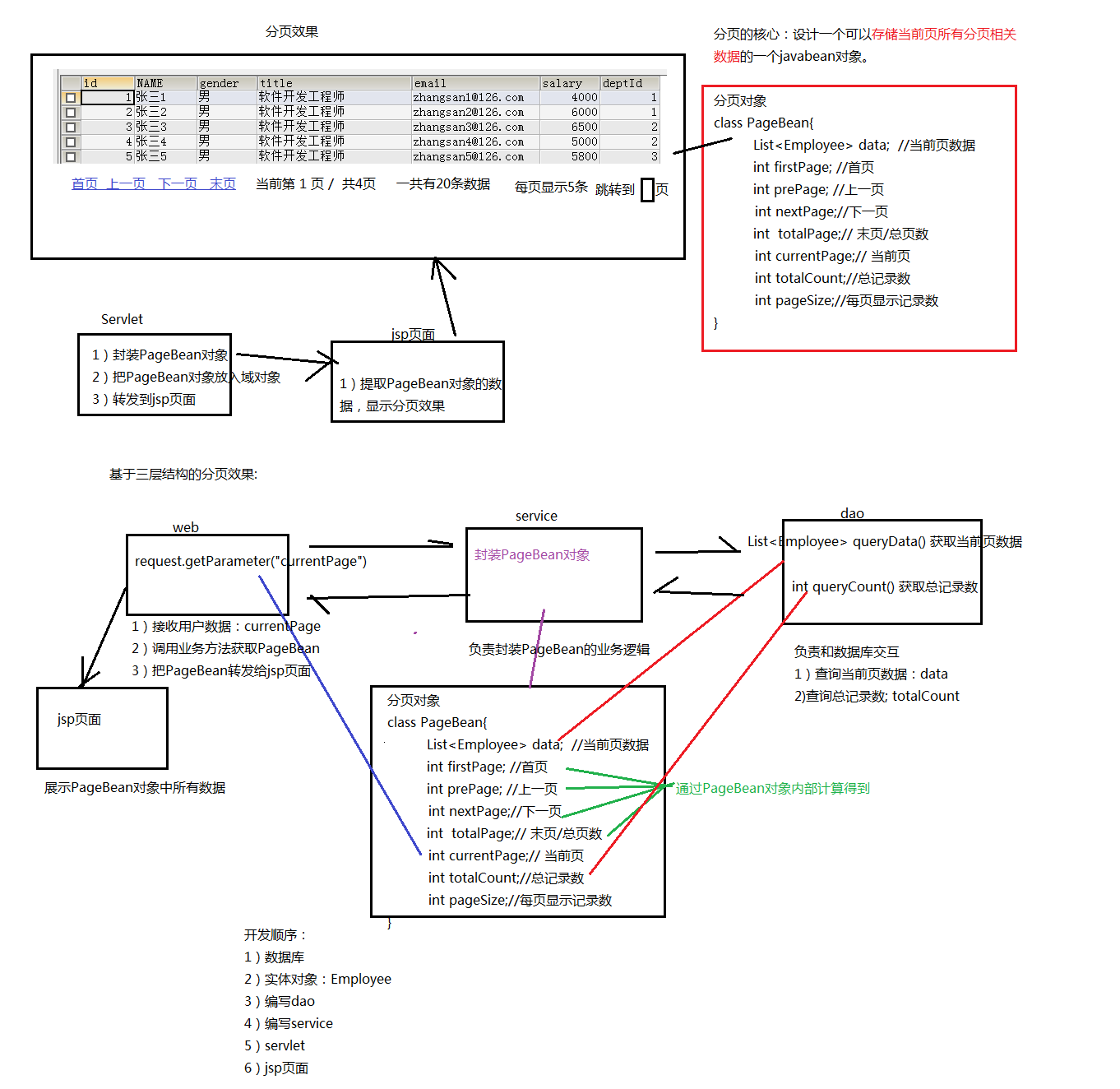
|  |
| --- |
| /\*\*  \* dbutils执行查询操作  \* @author APPle  \*  \*/  public class Demo2 {  /\*\*  \* ArrayHandler: 把结果集的第一行的数据封装成对象数组。  \*/  @Test  public void test1() throws Exception{  ComboPooledDataSource ds = new ComboPooledDataSource();  **//1.创建QueryRunner**  **QueryRunner qr = new QueryRunner(ds);**  **//2.执行sql**  **Object[] arr = (Object[])qr.query("select \* from student where id=?", new ArrayHandler(),new Object[]{2});**  for(Object obj:arr){  System.out.println(obj);  }  }    /\*\*  \* **ArrayListHandler: 把结果集的每一行数据封装对象数组，把这个对象数组放入List中**  \* @throws Exception  \*/  @Test  public void test2() throws Exception{  ComboPooledDataSource ds = new ComboPooledDataSource();  //1.创建QueryRunner  QueryRunner qr = new QueryRunner(ds);  //2.执行sql  List<Object[]> list = (List<Object[]>)qr.query("select \* from student", **new ArrayListHandler()**);  for(Object[] arr:list){//一行  //一列  for(Object obj:arr){  System.out.print(obj+"\t");  }  System.out.println();  }  }    /\*\*  \* **BeanHandler： 把结果集的第一行数据封装成javabean**  \* 约定前提： 表的字段名称和javabean的属性名称保持一致！！  \*/  @Test  public void test3() throws Exception{  ComboPooledDataSource ds = new ComboPooledDataSource();  //1.创建QueryRunner  QueryRunner qr = new QueryRunner(ds);  //2.执行sql  Student student = (Student)qr.query("select \* from student", **new BeanHandler(Student.class)**);  System.out.println(student);  }    /\*\*  \* **BeanListHandler: 把结果集的每一行数据封装成javabean，把这个javabean放入LIst中**  \* 约定前提： 表的字段名称和javabean的属性名称保持一致！！  \*/  @Test  public void test4() throws Exception{  ComboPooledDataSource ds = new ComboPooledDataSource();  //1.创建QueryRunner  QueryRunner qr = new QueryRunner(ds);  //2.执行sql  List<Student> list = (List<Student>)qr.query("select \* from student", **new BeanListHandler(Student.class)**);  for (Student student : list) {  System.out.println(student);  }  }    /\*\*  \* **ScalarHandler： 把结果集的第一行第一列取出。通常用于聚合函数查询。例如（count()/max()）**  \*/  @Test  public void test5() throws Exception{  ComboPooledDataSource ds = new ComboPooledDataSource();  //1.创建QueryRunner  QueryRunner qr = new QueryRunner(ds);  //2.执行sql  Long count = (Long)qr.query("select count(id) from student", **new ScalarHandler(1)**);  System.out.println("行数： "+count);  }    /\*\*  \* **如果表的字段名称和javabean的属性名称不一致时，需要自定义ResultSetHandler的实现类**  \*/  @Test  public void test6() throws Exception{  ComboPooledDataSource ds = new ComboPooledDataSource();  //1.创建QueryRunner  QueryRunner qr = new QueryRunner(ds);  List<Student> list = (List<Student>)qr.query("select \* from student2", **new MyStudentHandler()**);  for (Student student : list) {  System.out.println(student);  }  }  }  /\*\*  \* 自定义ResultSetHandler （ **MyStudentHandler()** ）  \*/  class MyStudentHandler implements ResultSetHandler{  @Override  public Object handle(ResultSet rs) throws SQLException {  List<Student> list = new ArrayList<Student>();  while(rs.next()){  Student s = new Student();  s.setId(rs.getInt("sid"));  s.setName(rs.getString("sname"));  s.setAge(rs.getInt("sage"));  s.setAddress(rs.getString("saddress"));  list.add(s);  }  return list;  }  } |

# 分页查询

## 分页核心

设计一个用于封装当前页所有分页相关的数据的对象，叫分页对象**PageBean**

|  |
| --- |
| /\*\*  \* 分页对象。用于封装当前页的分页相关的所有数据  \* **@author** APPle  \*  \*/  **public** **class** PageBean {  **private** List<Employee> data;//当前页的数据  **private** Integer firstPage;//首页  **private** Integer prePage;//上一页  **private** Integer nextPage;//下一页  **private** Integer totalPage;//末页、总页数  **private** Integer currentPage;//当前页  **private** Integer totalCount;//总记录数  **private** Integer pageSize;//每页显示的记录数  } |



## 分页的实现步骤

## 1．编写分页对象和实体对象

|  |
| --- |
| /\*\*  \* 分页对象。用于封装当前页的分页相关的所有数据  \* **@author** APPle  \*  \*/  **public** **class** PageBean {  **private** List<Employee> data;//当前页的数据  **private** Integer firstPage;//首页  **private** Integer prePage;//上一页  **private** Integer nextPage;//下一页  **private** Integer totalPage;//末页、总页数  **private** Integer currentPage;//当前页  **private** Integer totalCount;//总记录数  **private** Integer pageSize;//每页显示的记录数  **public** List<Employee> getData() {  **return** data;  }  **public** **void** setData(List<Employee> data) {  **this**.data = data;  }  **public** Integer getFirstPage() {  **return** 1;  }  **public** **void** setFirstPage(Integer firstPage) {  **this**.firstPage = firstPage;  }  /\*\*  \* 计算上一页  \* **@return**  \*/  **public** Integer getPrePage() {  **return** **this**.getCurrentPage()==**this**.getFirstPage() ? 1 : **this**.getCurrentPage()-1;  }  **public** **void** setPrePage(Integer prePage) {  **this**.prePage = prePage;  }  /\*\*  \* 计算下一页  \* **@return**  \*/  **public** Integer getNextPage() {  **return** **this**.getCurrentPage()==**this**.getTotalPage()?  **this**.getTotalPage()  : **this**.getCurrentPage()+1;  }  **public** **void** setNextPage(Integer nextPage) {  **this**.nextPage = nextPage;  }  **public** Integer getTotalPage() {  **return** **this**.getTotalCount()%**this**.getPageSize()==0 ?  **this**.getTotalCount()/**this**.getPageSize()  :**this**.getTotalCount()/**this**.getPageSize()+1;  }  **public** **void** setTotalPage(Integer totalPage) {  **this**.totalPage = totalPage;  }  **public** Integer getCurrentPage() {  **return** currentPage;  }  **public** **void** setCurrentPage(Integer currentPage) {  **this**.currentPage = currentPage;  }  **public** Integer getTotalCount() {  **return** totalCount;  }  **public** **void** setTotalCount(Integer totalCount) {  **this**.totalCount = totalCount;  }  **public** Integer getPageSize() {  **return** pageSize;  }  **public** **void** setPageSize(Integer pageSize) {  **this**.pageSize = pageSize;  }    } |

|  |
| --- |
| /\*\*  \* 员工对象  \* **@author** APPle  \*  \*/  **public** **class** Employee {  **private** **int** id;  **private** String name;  **private** String gender;  **private** **int** age;  **private** String title;  **private** String phone;  **private** String email;  **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** String getGender() {  **return** gender;  }  **public** **void** setGender(String gender) {  **this**.gender = gender;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  **public** String getTitle() {  **return** title;  }  **public** **void** setTitle(String title) {  **this**.title = title;  }  **public** String getPhone() {  **return** phone;  }  **public** **void** setPhone(String phone) {  **this**.phone = phone;  }  **public** String getEmail() {  **return** email;  }  **public** **void** setEmail(String email) {  **this**.email = email;  }  **public** Employee(**int** id, String name, String gender, **int** age, String title,  String phone, String email) {  **super**();  **this**.id = id;  **this**.name = name;  **this**.gender = gender;  **this**.age = age;  **this**.title = title;  **this**.phone = phone;  **this**.email = email;  }  **public** Employee() {  **super**();  // **TODO** Auto-generated constructor stub  }  @Override  **public** String toString() {  **return** "Employee [age=" + age + ", email=" + email + ", gender="  + gender + ", id=" + id + ", name=" + name + ", phone=" + phone  + ", title=" + title + "]";  }      } |

## 2．编写DAO层代码（查询总记录数和查询当前页数据）

|  |
| --- |
| /\*\*  \* 员工的DAO类  \* **@author** APPle  \*  \*/  **public** **class** EmpDao {  /\*\*  \* 提供一个查询当前页员工的方法  \*/  **public** List<Employee> queryCurrentData(Integer currentPage,Integer pageSize){  **try** {  //1.创建QueryRunner对象  QueryRunner qr = **new** QueryRunner(JdbcUtil.*getDataSource*());  //2.执行查询sql操作  //计算查询的起始行  **int** startNo = (currentPage-1)\*pageSize;  List<Employee> list = (List<Employee>)qr.query("SELECT \* FROM employee LIMIT ?,?",  **new** BeanListHandler(Employee.**class**),  **new** Object[]{startNo,pageSize});  **return** list;  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }    /\*\*  \* 提供查询总记录数的方法  \* **@param** args  \*/  **public** Integer queryTotalCount(){  **try** {  //1.创建QueryRunner  QueryRunner qr = **new** QueryRunner(JdbcUtil.*getDataSource*());  //2.执行sql查询  Long count = (Long)qr.query("SELECT COUNT(\*) FROM employee", **new** ScalarHandler(1));  **return** count.intValue();  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }  } |

## 3．编写Service层代码（封装PageBean对象）

|  |
| --- |
| /\*\*  \* 员工的业务类  \* **@author** APPle  \*  \*/  **public** **class** EmpService {  /\*\*  \* 提供用于封装PageBean对象方法（处理业务逻辑）  \*/  **public** PageBean queryPageBean(Integer currentPage,Integer pageSize){  //封装PageBean分页对象数据  PageBean pageBean = **new** PageBean();    //设置当前页  pageBean.setCurrentPage(currentPage);    //设置每页显示的记录数  pageBean.setPageSize(pageSize);    EmpDao empDao = **new** EmpDao();  /\*\*  \* 从数据库中查询出总记录数  \*/  **int** totalCount = empDao.queryTotalCount();  //设置总记录数  pageBean.setTotalCount(totalCount);    //设置当前页的数据  /\*\*  \* 从数据库中查询出当前页的员工数据  \*/  List<Employee> list = empDao.queryCurrentData(pageBean.getCurrentPage(), pageBean.getPageSize());  pageBean.setData(list);    **return** pageBean;  }  } |

## 4．编写Servlet代码（接收用户输入）

|  |
| --- |
| **public** **class** PageServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  /\*\*\*\*\*\*\*\*\*\*\*一、获取用户输入\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  //设置当前页(获取用户的输入)  String currentPageStr = request.getParameter("currentPage");  //如果用户没有输入，就是默认第1页  **if**(currentPageStr==**null** || currentPageStr.equals("")){  currentPageStr = "1";  }    //设置每页显示的记录数(获取用户的输入)  String pageSizeStr = request.getParameter("pageSize");  **if**(pageSizeStr==**null** || pageSizeStr.equals("")){  pageSizeStr = "5";  }    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*二、调用业务方法，获取PageBean对象\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  EmpService empService = **new** EmpService();  PageBean pageBean = empService.queryPageBean(Integer.*parseInt*(currentPageStr), Integer.*parseInt*(pageSizeStr));      /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*三、得到业务数据，跳转视图\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  //把PageBean数据发送到jsp页面中显示  request.setAttribute("pageBean", pageBean);  //转发  request.getRequestDispatcher("/list.jsp").forward(request, response);  }  } |

## 5．编写jsp页面代码（显示分页效果）

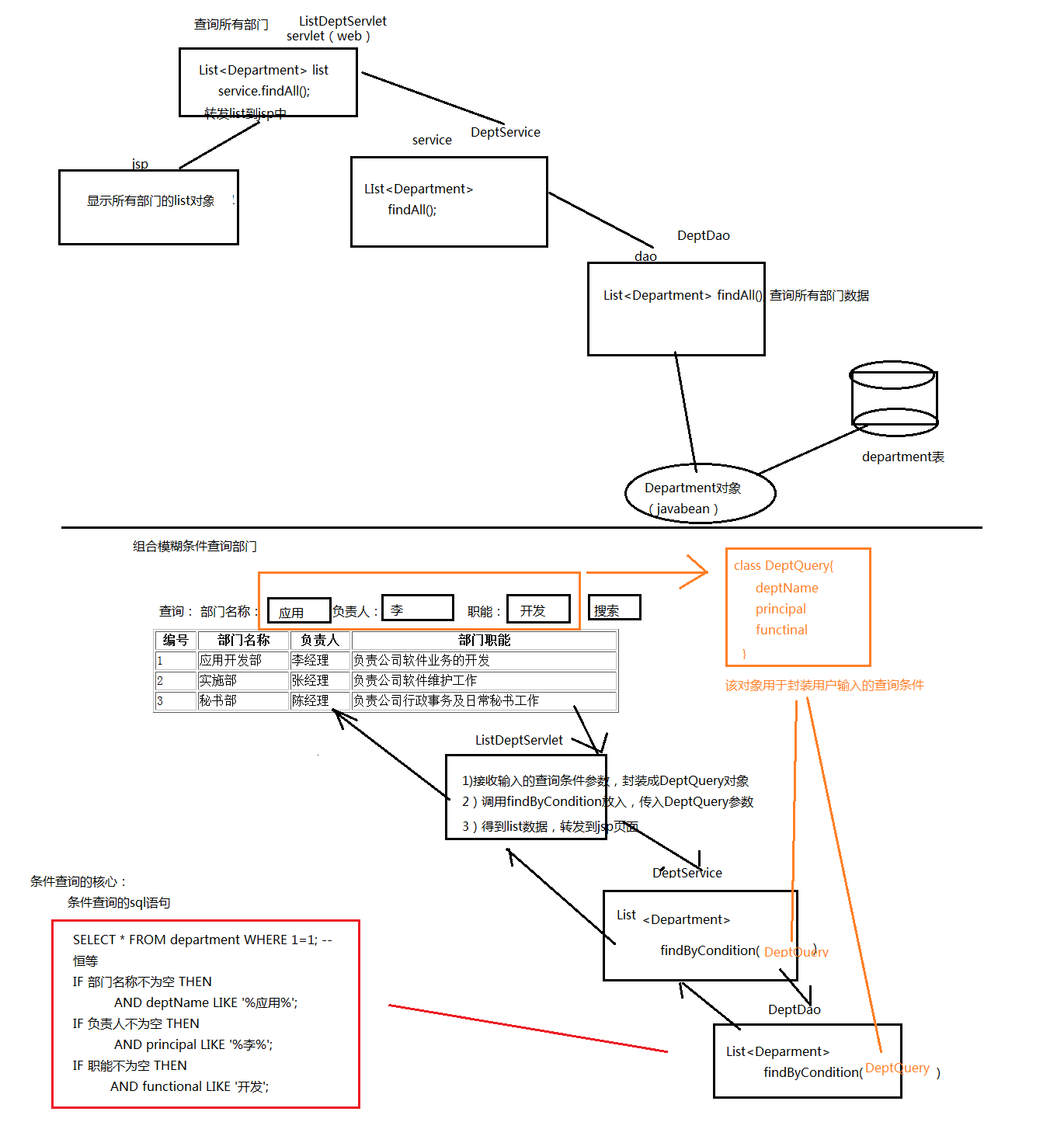
|  |
| --- |
| <%@taglib uri=*"http://java.sun.com/jsp/jstl/core"* prefix=*"c"*%>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>分页显示效果</title>  </head>    <body>  <table border=*"1"* align=*"center"* width=*"700px"*>  <tr>  <th>编号</th>  <th>姓名</th>  <th>性别</th>  <th>年龄</th>  <th>职位</th>  <th>电话</th>  <th>邮箱</th>  </tr>  <c:forEach items="${requestScope.pageBean.data}" var=*"emp"*>  <tr>  <td>${emp.id }</td>  <td>${emp.name }</td>  <td>${emp.gender }</td>  <td>${emp.age}</td>  <td>${emp.title }</td>  <td>${emp.phone }</td>  <td>${emp.email }</td>  </tr>  </c:forEach>  <tr>  <td colspan=*"7"* align=*"center"*>  <%--  1）如果当前页是首页，则不显示“首页”和“上一页”的连接  2）如果当前页是末页，则不显示“末页”和“下一页”的连接  --%>  <c:choose>  <c:when test="${pageBean.currentPage==pageBean.firstPage}">  首页&nbsp;  上一页  </c:when>  <c:otherwise>  <a href=*"*${pageContext.request.contextPath }*/PageServlet?currentPage=*${pageBean.firstPage }*&pageSize=*${pageBean.pageSize}*"*>首页</a>&nbsp;  <a href=*"*${pageContext.request.contextPath }*/PageServlet?currentPage=*${pageBean.prePage }*&pageSize=*${pageBean.pageSize}*"*>上一页</a>&nbsp;  </c:otherwise>  </c:choose>    <c:choose>  <c:when test="${pageBean.currentPage==pageBean.totalPage}">  下一页&nbsp;  末页  </c:when>  <c:otherwise>  <a href=*"*${pageContext.request.contextPath }*/PageServlet?currentPage=*${pageBean.nextPage}*&pageSize=*${pageBean.pageSize}*"*>下一页</a>&nbsp;  <a href=*"*${pageContext.request.contextPath }*/PageServlet?currentPage=*${pageBean.totalPage }*&pageSize=*${pageBean.pageSize}*"*>末页</a>&nbsp;  </c:otherwise>  </c:choose>  当前第${pageBean.currentPage }页/共${pageBean.totalPage }页，  &nbsp;共${pageBean.totalCount }条&nbsp;每页显示 <input type=*"text"* name=*"pageSize"* id=*"pageSize"* size=*"2"* value=*"*${pageBean.pageSize }*"* onblur=*"changePageSize()"*/> 条  </td>  </tr>  </table>    <script type=*"text/javascript"*>  //改变每页显示记录数的方法  **function** changePageSize(){  //获取用户输入的记录数  **var** pageSize = document.getElementById("pageSize").value;  //判断是否输入的数值  **var** reg = /^[1-9][0-9]?$/;  **if**(!reg.test(pageSize)){  alert("请输入数组类型！");  **return**;  }  //把记录数发送到后台  **var** url = "${pageContext.request.contextPath}/PageServlet?pageSize="+pageSize;  window.location.href=url;  }  </script>  </body>  </html> |

# 条件查询

## 条件查询的核心

根据用户的查询条件组装sql语句：

|  |
| --- |
| /\*\*  \* 根据条件查询部门  \* **@param** args  \*/  **public** List<Department> findByCondition(DeptQuery query){  **try** {  //根据查询条件拼凑一条sql语句  StringBuffer sql = **new** StringBuffer("select \* from department where 1=1 ");  **if**(query!=**null**){  //部门名称不为空，则往sql添加条件(trim():去掉两边空格)  **if**(query.getDeptName()!=**null** && !query.getDeptName().trim().equals("")){  sql.append(" and deptName like '%"+query.getDeptName()+"%'");  }  //负责人  **if**(query.getPrincipal()!=**null** && !query.getPrincipal().trim().equals("")){  sql.append(" and principal like '%"+query.getPrincipal()+"%'");  }  //职能  **if**(query.getFunctional()!=**null** && !query.getFunctional().trim().equals("")){  sql.append(" and functional like '%"+query.getFunctional()+"%'");  }  }  System.***out***.println(sql.toString());  QueryRunner qr = **new** QueryRunner(JdbcUtil.*getDataSource*());  **return** (List<Department>)qr.query(sql.toString(), **new** BeanListHandler(Department.**class**));  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  } |



# 总结

## jdbc通用功能：

### 1）分页查询：

核心： 设计一个用于封装当前页分页相关数据的javabean对象

当前页的数据

首页

上一页

下一页

末页/总页数

当前页码

总记录数

每页显示记录数

三层结构：

1）dao层：

查询当前页的数据

总记录数

2）service层；

封装分页javabean对象

（

首页

上一页

下一页

末页/总页数

）

3）web层：

当前页码

每页显示记录数

### 2）条件查询

核心： 根据不同的条件拼凑条件查询的sql语句

条件查询sql语句：

select \* from 表 where 1=1 (恒成立)

if 条件1

and 字段1 like 内容

if 条件2

and 字符2 like 内容

……

### 3）分页+条件查询

在分页的基础上，修改两条sql语句

3.1 查询当前页数据的sql语句

select \* from 表 where 1=1

if 条件1

and 字段1 like 内容

if 条件2

and 字符2 like 内容

……

limit 起始行,每页查询行数;

3.2 查询总记录数的sql语句

select count(\*) from 表 where 1=1

if 条件1

and 字段1 like 内容

if 条件2

and 字符2 like 内容

……