**MySQL多表查询与事务的操作**

# 回顾

## DQL查询

### 排序使用什么子句：order by

* 升序：asc
* 降序：desc

### 聚合函数

|  |  |
| --- | --- |
| **聚合函数** | **作用** |
| **count** | 统计个数 |
| **sum** | 求和 |
| **max** | 最大值 |
| **min** | 最小值 |
| **avg** | 平均 |

### 分页查询

|  |
| --- |
| **limit 起始行数， 返回多少行**  **起始行数从0开始** |

### 分组查询

|  |
| --- |
| **group by 分组的字段 having 筛选的条件** |

## 约束

### 约束的关键字

|  |  |
| --- | --- |
| **约束名** | **约束关键字** |
| **主键** | primary key |
| **唯一** | unique |
| **非空** | not null |
| **默认** | default |
| **外键** | foreign key |

### 自增长的关键字：

|  |
| --- |
| **auto\_increment** |

### 级联操作的语法：

|  |  |
| --- | --- |
| **级联操作语法** | **描述** |
| **on update cascade** | 级联更新 |
| **on delete cascade** | 级联删除 |

## 表与表之间的关系

|  |  |
| --- | --- |
| **表与表之间的三种关系** | **关系如何维护** |
| **一对多** | 通过外键维护 |
| **多对多** | 有一个中间表，中间表与其它两张表是多对一 |
| **一对一** | 1. 特殊的一对多，多方使用唯一约束 2. 从表的主键又是外键 |

## 数据库的三大范式

|  |  |
| --- | --- |
| **范式** | **特点** |
| **1** | 原子性，每列不可再拆分 |
| **2** | 不产生局部依赖 |
| **3** | 不产生传递依赖 |

# 学习目标

1. 能够使用内连接进行多表查询
2. 能够使用左外连接和右外连接进行多表查询
3. 能够使用子查询进行多表查询
4. 能够使用多表进行查询
5. 能够理解事务的概念
6. 能够说出事务的特点
7. 能够在MySQL中使用事务
8. 能够理解脏读、不可重复读、幻读的概念及解决办法
9. 能够使用DCL管理MySQL中的用户

# 表连接查询

## 什么是多表查询

* 数据准备

# 创建部门表

create table dept(

id int primary key auto\_increment,

name varchar(20)

)

insert into dept (name) values ('开发部'),('市场部'),('财务部');

# 创建员工表

create table emp (

id int primary key auto\_increment,

name varchar(10),

gender char(1), -- 性别

salary double, -- 工资

join\_date date, -- 入职日期

dept\_id int,

foreign key (dept\_id) references dept(id) -- 外键，关联部门表(部门表的主键)

)

insert into emp(name,gender,salary,join\_date,dept\_id) values('孙悟空','男',7200,'2013-02-24',1);

insert into emp(name,gender,salary,join\_date,dept\_id) values('猪八戒','男',3600,'2010-12-02',2);

insert into emp(name,gender,salary,join\_date,dept\_id) values('唐僧','男',9000,'2008-08-08',2);

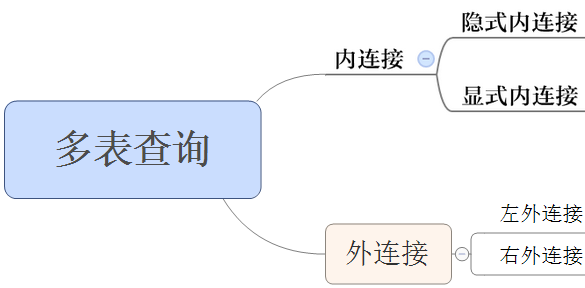
insert into emp(name,gender,salary,join\_date,dept\_id) values('白骨精','女',5000,'2015-10-07',3);

insert into emp(name,gender,salary,join\_date,dept\_id) values('蜘蛛精','女',4500,'2011-03-14',1);

* 多表查询的作用：

在实际的开发过程中，许多查询数据不是来源于一张表，可能需要从多张表中去得到结果。所以需要使用多表查询。

### 多表查询的分类：



## 笛卡尔积现象

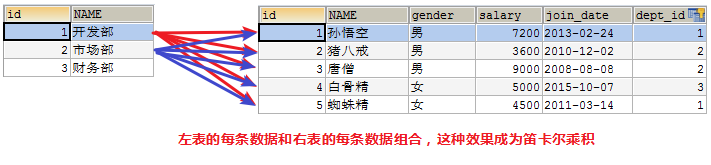
### 什么是笛卡尔积现象

* 什么是笛卡尔积： 如果有两张表创建表连接，左表中每条记录与右表中每条记录都匹配一次，查询的结果的行数是两张表行数的乘积。

-- 需求：查询所有的员工和所有的部门

*select \* from dept, emp;*

* 结果分析：



### 如何清除笛卡尔积现象的影响

我们发现不是所有的数据组合都是有用的，只有**员工表.dept\_id = 部门表.id** 的数据才是有用的。所以需要通过条件过滤掉没用的数据。

-- 设置过滤条件，这就是内连接

select \* from dept d, emp e where d.`id` = e.`dept\_id`;

## 内连接

用左边表的记录去匹配右边表的记录，如果符合条件的则显示。上面看到的就是一种内连接的操作。

### 隐式内连接

* 隐式内连接：看不到JOIN关键字，条件使用WHERE指定。一般是主表主键=从表的外键

|  |
| --- |
| **SELECT 字段名 FROM 左表, 右表 WHERE 条件** |

### 显示内连接

* 显示内连接：使用INNER JOIN ... ON语句, 可以省略INNER

|  |
| --- |
| **SELECT 字段名 FROM 左表 INNER JOIN 右表 ON 条件** |

* 查询唐僧的信息，显示员工id，姓名，性别，工资和所在的部门名称，我们发现需要联合2张表同时才能查询出需要的数据，使用内连接。

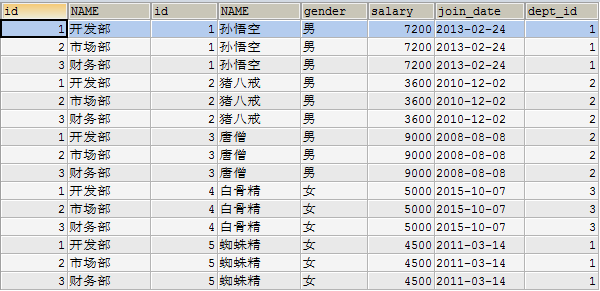


原来表：



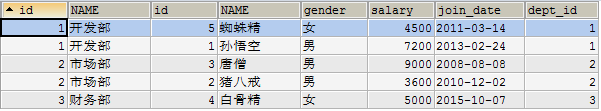
1. 确定查询哪些表

select \* from dept inner join emp;



1. 确定表连接条件，员工表.dept\_id = 部门表.id 的数据才是有效的

select \* from dept inner join emp on dept.`id` = emp.`dept\_id`;



1. 确定查询条件，我们查询的是唐僧的信息，员工表.name='唐僧'

select \* from dept inner join emp on dept.`id` = emp.`dept\_id` where emp.`name` = '唐僧';



1. 确定查询字段，查询唐僧的信息，显示员工id，姓名，性别，工资和所在的部门名称

select emp.`id`, emp.`name`, emp.`gender`, emp.`salary`, dept.`name` from dept inner join emp on dept.`id` = emp.`dept\_id` where emp.`name` = '唐僧';



1. 我们发现写表名有点长，可以给表取别名，显示的字段名也使用别名

select emp.`id` 编号, emp.`name` 姓名, emp.`gender` 性别, emp.`salary` 工资, dept.`name` 部门名称 from dept inner join emp on dept.`id` = emp.`dept\_id` where emp.`name` = '唐僧';



### 内连接查询步骤：

1. 确定查询哪些表 dept, emp
2. 确定表连接的条件 on
3. 确定查询的条件where
4. 确定显示哪些列

## 左外连接

* 左外连接：使用LEFT JOIN ... ON，OUTER可以省略

|  |
| --- |
| **SELECT 字段名 FROM 左表 LEFT OUTER JOIN 右表 ON 条件** |

用左边表的记录去匹配右边表的记录，如果符合条件的则显示；否则，显示NULL

可以理解为：在内连接的基础上保证左表的数据全部显示

-- 在部门表中增加一个销售部，将部门表设置成左表，员工表设置成右表

select \* from dept;

select \* from emp;

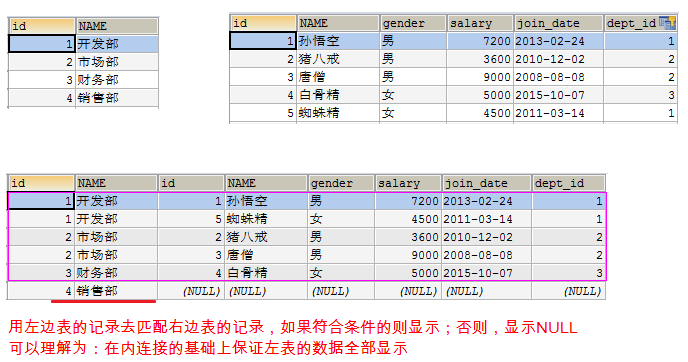
insert into dept values (null, '销售部');

-- 使用内连接查询，发现少了销售部

select \* from dept inner join emp on emp.`dept\_id` = dept.`id`;

-- 使用左外连接查询

select \* from dept left join emp on emp.`dept\_id` = dept.`id`;



## 右外连接

* 右外连接：使用RIGHT [OUTER] JOIN ... ON，OUTER可以省略

|  |
| --- |
| **SELECT 字段名 FROM 左表 RIGHT OUTER JOIN 右表 ON 条件** |

用右边表的记录去匹配左边表的记录，如果符合条件的则显示；否则，显示NULL

可以理解为：在内连接的基础上保证右表的数据全部显示，左表是部门表，右表是员工表

-- 在员工表中增加一个员工:'沙僧','男',6666,'2013-02-24',null

-- 不在部门中

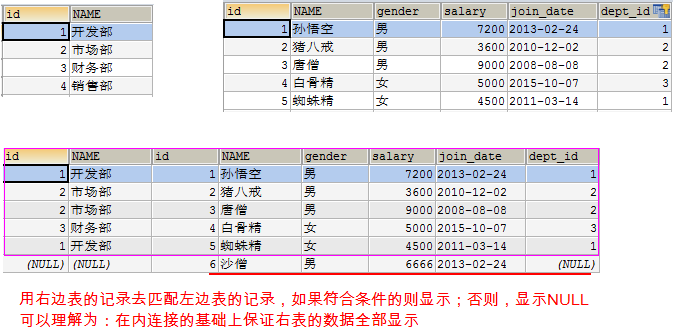
insert into emp values (null,'沙僧','男',6666,'2013-02-24',null);

-- 使用内连接查询

select \* from dept inner join emp on dept.`id` = emp.`dept\_id`;

-- 使用右外连接查询

select \* from dept right join emp on dept.`id` = emp.`dept\_id`;



# 子查询

## 什么是子查询

-- 需求：查询开发部中有哪些员工

select \* from emp;

-- 得到开发部的id

select id from dept where name='开发部';

-- 查询员工表，dept\_id等于1

select \* from emp where dept\_id = 1;

-- 把2句写成一句

select \* from emp where dept\_id = (select id from dept where name='开发部');

* 子查询的概念：

1. 一个查询的结果做为另一个查询的条件
2. SQL语句有查询的嵌套，内部的查询称为子查询
3. 子查询条件必须使用括号

## 子查询结果的三种情况：

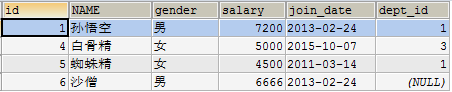
1. 单行单列的情况， 一个值

fig:

1. 多行单列，多个值，类似于数组或集合

fig:

1. 多行多列，可以进行再次查询，类似于一张虚拟表。



## 子查询的结果是一个值的时候

* 子查询结果只要是单行单列，肯定在WHERE后面作为条件，父查询使用：比较运算符，如：> 、<、<>、= 等

|  |
| --- |
| **SELECT 查询字段 FROM 表 WHERE 字段=（子查询）;** |

-- 案例：查询工资最高的员工是谁？

-- 1. 查询最高工资是多少

select max(salary) from emp;

-- 2. 根据最高工资到员工表查询到对应的员工信息

select \* from emp where salary = 9000;

-- 使用子查询

select \* from emp where salary = (select max(salary) from emp);



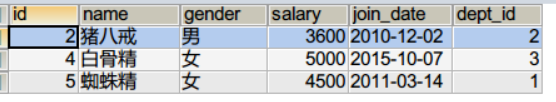
-- 案例：查询工资小于平均工资的员工有哪些？

-- 1. 查询平均工资是多少

select avg(salary) from emp;

-- 2. 到员工表查询小于平均的员工信息

select \* from emp where salary < (select avg(salary) from emp);



## 子查询结果是多行单列的时候

* 子查询结果是多行单列，结果集类似于一个数组，父查询使用IN运算符

|  |
| --- |
| **SELECT 查询字段 FROM 表 WHERE 字段 IN （子查询）;** |

-- 查询工资大于5000的员工，来自于哪些部门的名字

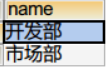
-- 1. 先查询大于5000的员工所在的部门id

select dept\_id from emp where salary > 5000;

-- 2. 再查询在这些部门id中部门的名字, Subquery returns more than 1 row

select dept.`name` from dept where id = (select dept\_id from emp where salary > 5000);

select dept.`name` from dept where id in (select dept\_id from emp where salary > 5000);



-- 查询开发部与财务部所有的员工信息

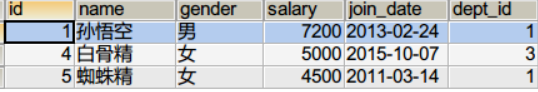
-- 1. 先查询开发部与财务部的id

select id from dept where name='开发部' or name='财务部';

select id from dept where name in('开发部','财务部');

-- 2. 再查询在这些部门id中有哪些员工

select \* from emp where dept\_id in (select id from dept where name='开发部' or name='财务部');



## 子查询的结果是多行多列

子查询结果只要是多列，肯定在FROM后面作为虚拟表

|  |
| --- |
| **SELECT 查询字段 FROM （子查询） 表别名 WHERE 条件;** |

子查询作为表需要取别名，否则这张表没有名称则无法访问表中的字段

-- 查询出2011年以后入职的员工信息，包括部门名称

-- 1. 在员工表中查询2011-1-1以后入职的员工

select \* from emp where join\_date > '2011-1-1';

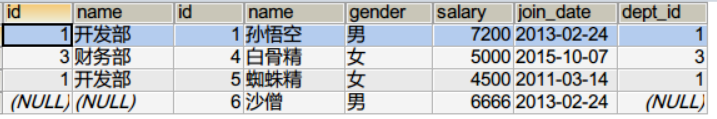
-- 2. 查询所有的部门信息，与上面的虚拟表中的信息组合，找出所有部门id等于的dept\_id

select \* from dept, (select \* from emp where join\_date > '2011-1-1') e where dept.`id` = e.dept\_id;

select \* from dept right join (select \* from emp where join\_date > '2011-1-1') e on dept.`id` = e.dept\_id;

-- 也可以使用表连接，很多的子查询都可以使用相应的表连接来替换

select \* from dept right join emp on dept.`id` = emp.`dept\_id` where emp.`join\_date` > '2011-1-1';



## 子查询小结

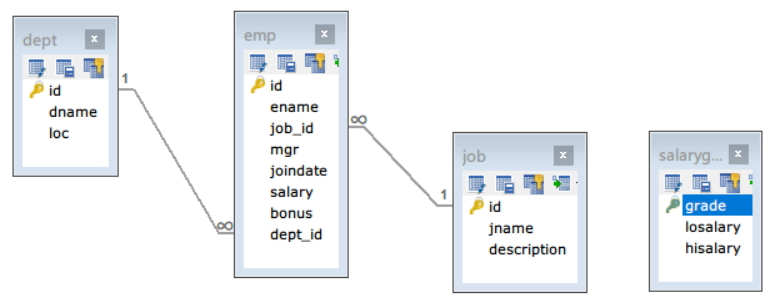
* 子查询结果只要是单列(1值，多个值)，则在WHERE后面作为条件
* 子查询结果只要是多列，则在FROM后面作为表进行二次查询

# 多表查询的案例

## 准备数据

-- 部门表  
CREATE TABLE dept (  
 id INT PRIMARY KEY PRIMARY KEY, -- 部门id  
 dname VARCHAR(50), -- 部门名称  
 loc VARCHAR(50) -- 部门所在地  
);  
  
-- 添加4个部门  
INSERT INTO dept(id,dname,loc) VALUES   
(10,'教研部','北京'),  
(20,'学工部','上海'),  
(30,'销售部','广州'),  
(40,'财务部','深圳');  
  
-- 职务表，职务名称，职务描述  
CREATE TABLE job (  
 id INT PRIMARY KEY,  
 jname VARCHAR(20),  
 description VARCHAR(50)  
);  
  
-- 添加4个职务  
INSERT INTO job (id, jname, description) VALUES  
(1, '董事长', '管理整个公司，接单'),  
(2, '经理', '管理部门员工'),  
(3, '销售员', '向客人推销产品'),  
(4, '文员', '使用办公软件');  
  
-- 员工表  
CREATE TABLE emp (  
 id INT PRIMARY KEY, -- 员工id  
 ename VARCHAR(50), -- 员工姓名  
 job\_id INT, -- 职务id  
 mgr INT , -- 上级领导  
 joindate DATE, -- 入职日期  
 salary DECIMAL(7,2), -- 工资  
 bonus DECIMAL(7,2), -- 奖金  
 dept\_id INT, -- 所在部门编号  
 CONSTRAINT emp\_jobid\_ref\_job\_id\_fk FOREIGN KEY (job\_id) REFERENCES job (id),  
 CONSTRAINT emp\_deptid\_ref\_dept\_id\_fk FOREIGN KEY (dept\_id) REFERENCES dept (id)  
);  
  
-- 添加员工  
INSERT INTO emp(id,ename,job\_id,mgr,joindate,salary,bonus,dept\_id) VALUES   
(1001,'孙悟空',4,1004,'2000-12-17','8000.00',NULL,20),  
(1002,'卢俊义',3,1006,'2001-02-20','16000.00','3000.00',30),  
(1003,'林冲',3,1006,'2001-02-22','12500.00','5000.00',30),  
(1004,'唐僧',2,1009,'2001-04-02','29750.00',NULL,20),  
(1005,'李逵',4,1006,'2001-09-28','12500.00','14000.00',30),  
(1006,'宋江',2,1009,'2001-05-01','28500.00',NULL,30),  
(1007,'刘备',2,1009,'2001-09-01','24500.00',NULL,10),  
(1008,'猪八戒',4,1004,'2007-04-19','30000.00',NULL,20),  
(1009,'罗贯中',1,NULL,'2001-11-17','50000.00',NULL,10),  
(1010,'吴用',3,1006,'2001-09-08','15000.00','0.00',30),  
(1011,'沙僧',4,1004,'2007-05-23','11000.00',NULL,20),  
(1012,'李逵',4,1006,'2001-12-03','9500.00',NULL,30),  
(1013,'小白龙',4,1004,'2001-12-03','30000.00',NULL,20),  
(1014,'关羽',4,1007,'2002-01-23','13000.00',NULL,10);  
  
-- 工资等级表  
CREATE TABLE salarygrade (  
 grade INT PRIMARY KEY, -- 级别  
 losalary INT, -- 最低工资  
 hisalary INT -- 最高工资  
);  
  
-- 添加5个工资等级  
INSERT INTO salarygrade(grade,losalary,hisalary) VALUES   
(1,7000,12000),  
(2,12010,14000),  
(3,14010,20000),  
(4,20010,30000),  
(5,30010,99990);

* 分析4张表的关系：



## 练习1

### 需求：查询所有员工信息。显示员工编号，员工姓名，工资，职务名称，职务描述

### 具体操作：

1. 确定要查询哪些表：员工表和职务表emp e, job j
2. 确定表连接条件： e.job\_id=j.id
3. 确定查询字段：员工编号，员工姓名，工资，职务名称，职务描述

select e.`id`,e.`ename`,e.`salary`,j.`jname`,j.`description` from emp e inner join job j on j.`id` = e.`job\_id`

### 查询结果：



## 练习2

### 需求：查询所有员工信息。显示员工姓名，工资，工资等级

### 具体操作：

1. 确定要查询哪些表，emp e, salarygrade s
2. 确定表连接条件：员工表中工资数额与工资等级表之间的关联条件是：工资在最低与最高工资之间
3. 确定查询字段：员工姓名，工资，工资等级

select e.`ename`, e.`salary`,s.grade from emp e inner join salarygrade s on e.`salary` between s.losalary and s.hisalary;

### 查询结果：



## 练习3

### 需求：查询经理的信息。显示经理姓名，工资，职务名称，职务描述，部门名称，部门位置，工资等级

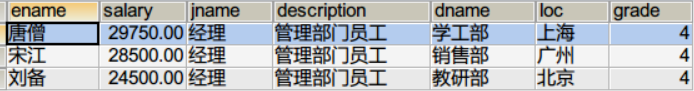
### 具体操作：

1. 确定要查询哪些表，emp e, job j, dept d, salarygrade s
2. 确定表连接条件  
   e.job\_id=j.id and e.dept\_id=d.id and e.salary between s.losalary and hisalary  
   额外条件：只需要查询经理的信息（j.jname='经理'）
3. 确定查询字段：员工姓名，工资，职务名称，职务描述，部门名称，部门位置，工资等级

select e.`ename`,e.`salary`,j.`jname`,j.`description`,d.`dname`,d.`loc`,s.`grade` from emp e inner join job j inner join dept d inner join salarygrade s

on e.`job\_id`=j.`id` and d.`id` = e.`dept\_id` and e.`salary` between s.`losalary` and s.`hisalary` where j.`jname` = '经理';

### 查询结果



## 多表查询规律小结：

1. 关联字段：不管我们查询几张表，表连接查询会产出笛卡尔积，需要消除笛卡尔积，拿到正确的数据。需要找到表与表之间通过哪个字段关联起来的（通常是外键=主键）
2. 关联的条件数：消除笛卡尔积规律：2张表需要1个条件，3张表需要2个条件，4张表需要3个条件。（条件数量=表的数量-1），每张表都要参与进来。
3. 多表连接查询步骤：
4. 确定要查询哪些表
5. 确定表连接条件
6. 确定查询字段

## 练习4

### 需求：查询出部门编号、部门名称、部门位置、部门人数

### 具体操作：

1. 查员工表，按部门编号进行分组，找到每个部门的人数和部门id
2. 将上面的查询结果做为虚拟表再和部门表进行表连接查询
3. 显示对应的字段

-- 查询出部门编号、部门名称、部门位置、部门人数

select \* from dept;

-- 1) 查员工表，按部门编号进行分组，找到每个部门的人数和部门id

select dept\_id, count(\*) from emp group by dept\_id;

-- 2) 将上面的查询结果做为虚拟表再和部门表进行表连接查询

select \* from dept d inner join (select dept\_id, count(\*) from emp group by dept\_id) e on d.`id` = e.dept\_id;

-- 3) 显示对应的字段

select d.`id` 编号,d.`dname` 部门名 ,d.`loc` 所在地,IFNULL(e.num, 0) 人数 from dept d left join (select dept\_id, count(\*) num from emp group by dept\_id) e on d.`id` = e.dept\_id;

### 查询结果



## 练习5

### 需求：

列出所有员工的姓名及其直接上级的姓名,没有领导的员工也需要显示。自连接

### 具体操作：

1. 确定要查询哪些表，emp e, emp m
2. 确定表连接条件 e.mgr=m.id

发现员工表中少了一条数据，因为罗贯中是董事长没有上司，没有领导的员工也需要显示，所以左表的数据需要全部显示。应该改成左外连接。

1. 确定查询字段：员工的姓名及其直接上级的姓名

* 注：IFNULL函数的作用是，如果前面的参数有值，则显示原有的值，如果没有值，则显示后面的参数

select e.`ename` 员工名, IFNULL(m.`ename`,'大BOSS') 上司 from emp e left join emp m on e.`mgr` = m.`id`;

### 查询结果：



## 练习6

### 需求：

查询工资高于公司平均工资的所有员工列：显示员工信息，部门名称，上级领导，工资等级

### 具体操作：

-- 查询工资高于公司平均工资的所有员工列：显示员工信息，部门名称，上级领导，工资等级

-- 1) 本例要使用子查询，先统计公司平均工资

select avg(salary) from emp;

-- 2) 确定要查询哪些表：emp e, emp m, dept d, salarygrade s

select \* from emp e inner join emp m inner join dept d inner join salarygrade s;

-- 3) 确定表连接条件

select \* from emp e inner join emp m inner join dept d inner join salarygrade s on e.`mgr` = m.`id`

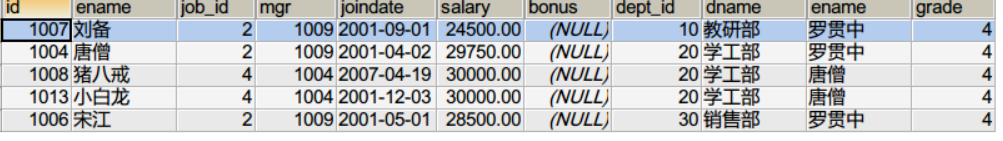
and d.`id` = e.`dept\_id` and e.`salary` between s.`losalary` and s.`hisalary` where e.`salary` > ( select avg(salary) from emp);

-- 4) 确定查询字段：员工所有信息，部门名称，上级领导，工资等级

select e.\*, d.`dname`,m.`ename`,s.`grade` from emp e inner join emp m inner join dept d inner join salarygrade s on e.`mgr` = m.`id`

and d.`id` = e.`dept\_id` and e.`salary` between s.`losalary` and s.`hisalary` and e.`salary` > ( select avg(salary) from emp);

### 查询结果



# 事务

## 事务的应用场景说明

* 什么是事务：

在实际开发过程中，一个业务操作，可能需要多次访问数据库。如果其中一条SQL语句失败，就会导致整个业务出现问题。事务中每条SQL语句都必须执行成功，如果有一条语句失败。则所有的SQL语句都应该回滚，全部执行失败。

* 转账的操作

-- 创建数据表  
CREATE TABLE account (  
 id INT PRIMARY KEY AUTO\_INCREMENT,  
 NAME VARCHAR(10),  
 balance DOUBLE  
);  
  
-- 添加数据  
INSERT INTO account (NAME, balance) VALUES ('Jack', 1000), ('Rose', 1000);

模拟Jack给Rose转500元钱，一个转账的业务操作最少要执行下面的2条语句：

Jack账号-500

Rose账号+500

-- 还原

update account set balance = 1000;

-- jack转账200给rose

-- jack扣钱

update account set balance = balance - 500 where name = 'jack';

-- rose加钱

update account set balance = balance + 500 where name = 'rose';

假设当Jack账号上-500元,服务器崩溃了。Rose的账号并没有+500元，数据就出现问题了。我们需要保证其中一条SQL语句出现问题，整个转账就算失败。只有两条SQL都成功了转账才算成功。这个时候就需要用到事务。

## 手动提交事务

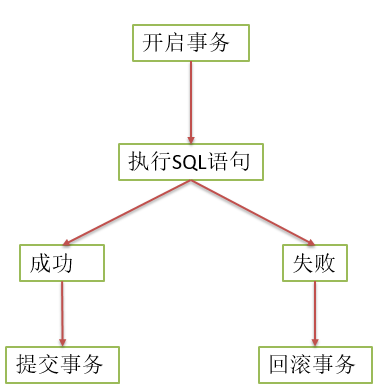
MYSQL中可以有两种方式进行事务的操作：

1. 手动提交事务
2. 自动提交事务

### 手动提交事务的SQL语句

|  |  |
| --- | --- |
| **功能** | **SQL语句** |
| **开启事务** | **start transaction** |
| **提交事务** | **commit** |
| **回滚事务** | **rollback** |

### 手动提交事务使用过程：

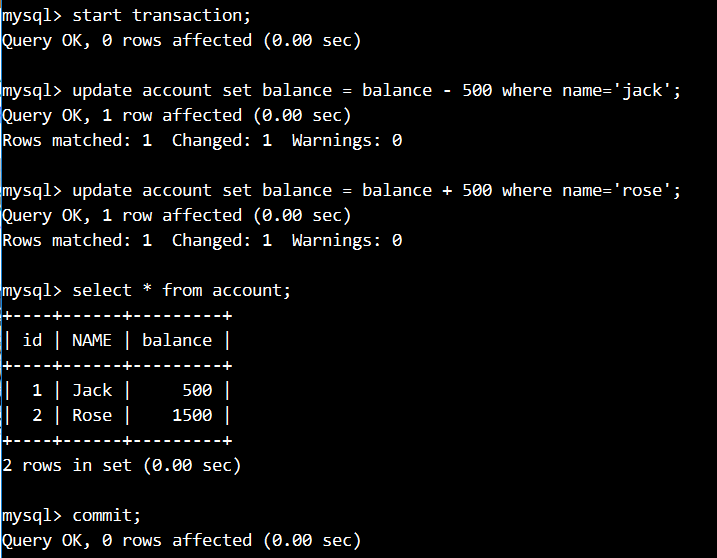


### 案例演示1：事务提交

模拟Jack给Rose转500元钱（成功） 目前数据库数据如下：



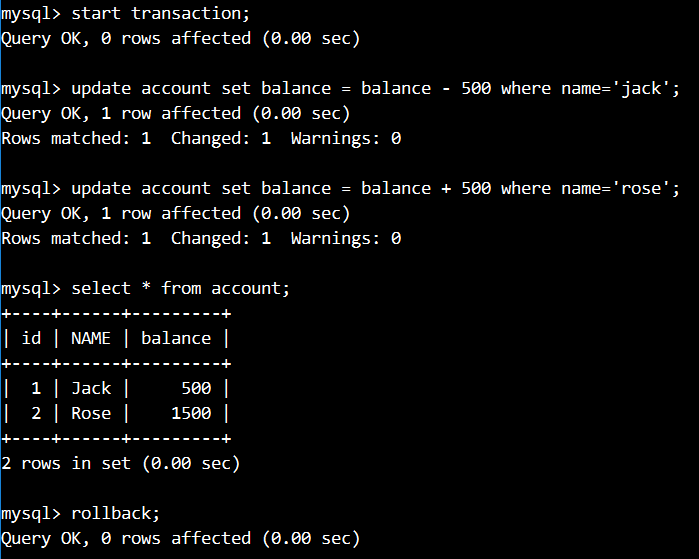
1. 使用DOS控制台进入MySQL
2. 执行以下SQL语句： 1.开启事务， 2.Jack账号-500， 3.Rose账号+500
3. 使用SQLYog查看数据库：发现数据并没有改变
4. 在控制台执行commit提交任务：
5. 使用SQLYog查看数据库：发现数据改变



### 案例演示2：事务回滚

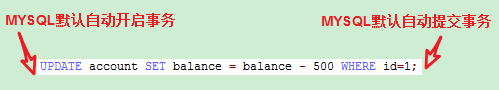
模拟Jack给Rose转500元钱（失败）

1. 在控制台执行以下SQL语句：1.开启事务， 2.Jack账号-500
2. 使用SQLYog查看数据库：发现数据并没有改变
3. 在控制台执行rollback回滚事务：
4. 使用SQLYog查看数据库：发现数据没有改变



总结: 如果事务中SQL语句没有问题，commit提交事务，会对数据库数据的数据进行改变。 如果事务中SQL语句有问题，rollback回滚事务，会回退到开启事务时的状态。

## 自动提交事务

MySQL默认每一条DML(增删改)语句都是一个单独的事务，每条语句都会自动开启一个事务，执行完毕自动提交事务，MySQL默认开始自动提交事务 

### 案例演示3：自动提交事务

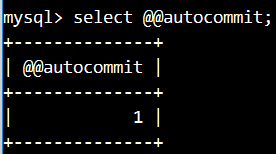
1. 将金额重置为1000
2. 更新其中某一个账户
3. 使用SQLYog查看数据库：发现数据已经改变



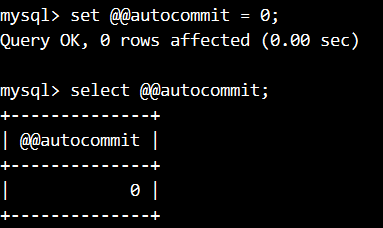
### 取消自动提交

* 查看MySQL是否开启自动提交事务

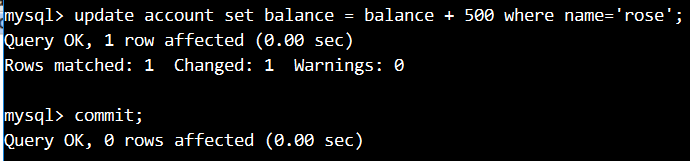
select @@autocommit; -- 只要是@@开头，表示mysql中的全局变量



* 取消自动提交事务



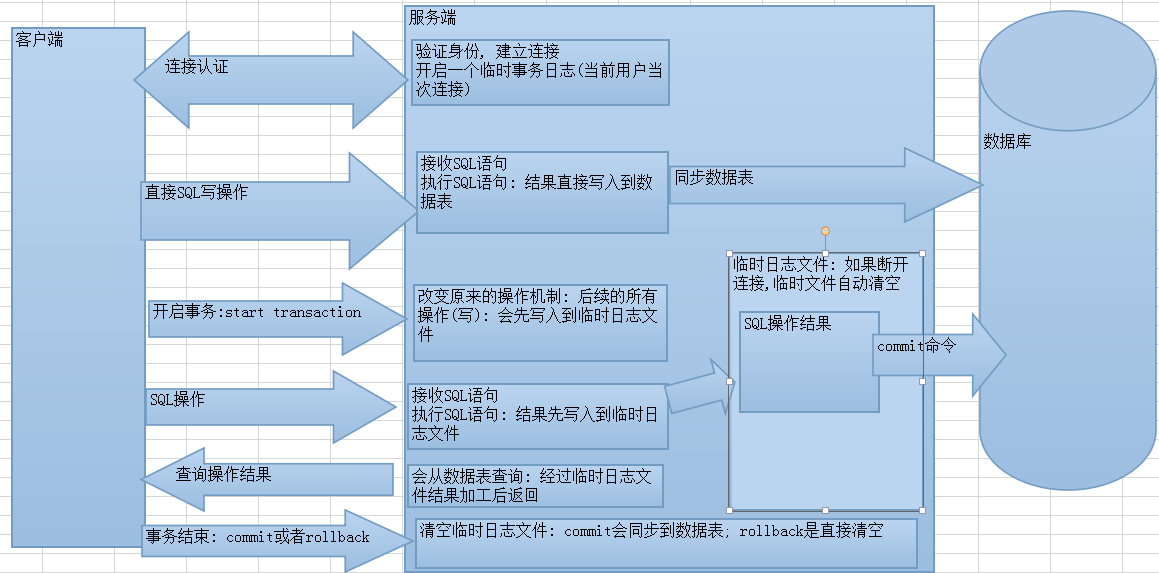
* 执行更新语句，使用SQLYog查看数据库，发现数据并没有改变
* 在控制台执行commit提交任务



## 事务原理

事务开启之后, 所有的操作都会临时保存到事务日志中, 事务日志只有在得到commit命令才会同步到数据表中，其他任何情况都会清空事务日志(rollback，断开连接)

### 原理图：



### 事务的步骤：

1. 客户端连接数据库服务器，创建连接时创建此用户临时日志文件
2. 开启事务以后，所有的操作都会先写入到临时日志文件中
3. 所有的查询操作从表中查询，但会经过日志文件加工后才返回
4. 如果事务提交commit则将日志文件中的数据写到表中，rollback否则清空日志文件。

## 回滚点

### 什么是回滚点

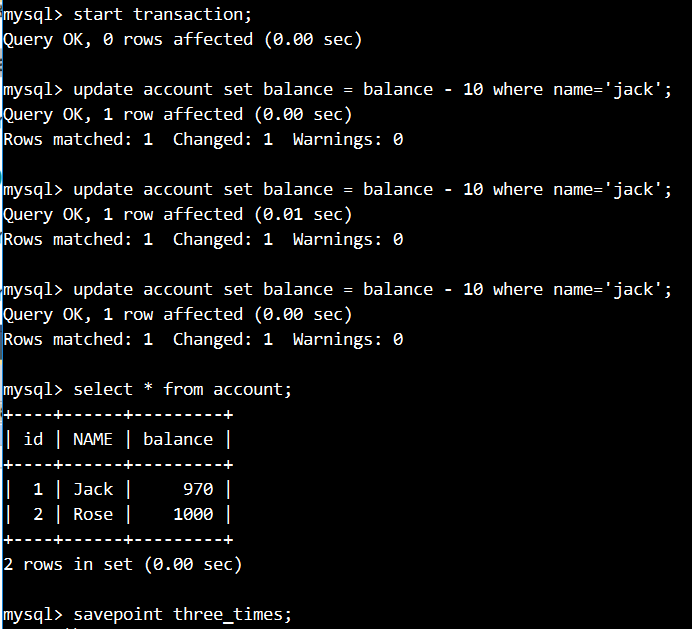
在某些成功的操作完成之后，后续的操作有可能成功有可能失败，但是不管成功还是失败，前面操作都已经成功，可以在当前成功的位置设置一个回滚点。可以供后续失败操作返回到该位置，而不是返回所有操作，这个点称之为回滚点。

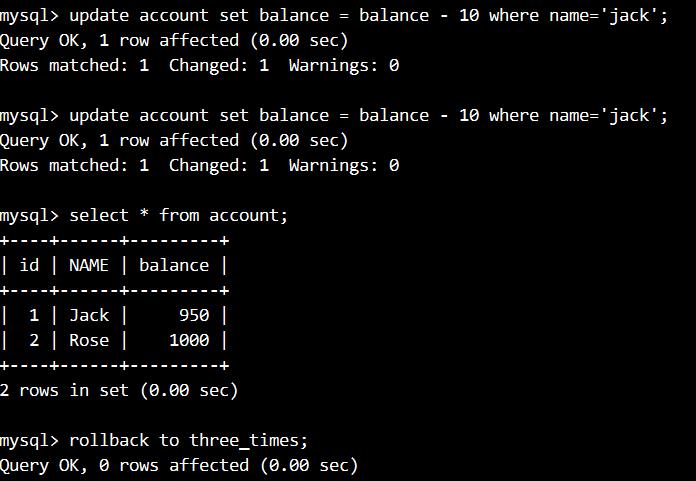
### 回滚点的操作语句

|  |  |
| --- | --- |
| **回滚点的操作语句** | **语句** |
| **设置回滚点** | savepoint 名字 |
| **回到回滚点** | rollback to 名字 |

### 具体操作：

1. 将数据还原到1000
2. 开启事务
3. 让Jack账号减3次钱，每次10块
4. 设置回滚点：savepoint three\_times;
5. 让Jack账号减4次钱，每次10块
6. 回到回滚点：rollback to three\_times;
7. 分析执行过程







* 总结：设置回滚点可以让我们在失败的时候回到回滚点，而不是回到事务开启的时候。

## 事务的隔离级别

### 事务的四大特性ACID

|  |  |
| --- | --- |
| **事务特性** | **含义** |
| **原子性（Atomicity）** | **事务不要再拆分，整个事务是一个整体，要么全部成功，要么全部失败** |
| **一致性（Consistency）** | **事务开启前和事务结束后，对数据库影响状态是一致。如：转账前总金额是2000，转账后总金额也是2000.** |
| **隔离性（Isolation）** | **如果同时运行多个事务，事务之间不能相互影响。** |
| **持久性（Durability）** | **只要事务提交，对数据库的影响是持久的。关机也不会改变。** |

### 事务的隔离级别

事务在操作时的理想状态：所有的事务在执行的过程中都不会互相影响。以下是三个会出现的问题：

|  |  |
| --- | --- |
| **并发访问的问题** | **含义** |
| **脏读** | **一个事务读取到了另一个事务中尚未提交的数据** |
| **不可重复读** | **一个事务中两次读取的数据内容不一致，要求的是一个事务中多次读取时数据是一致的，这是事务update时引发的问题** |
| **幻读** | **一个事务中两次读取的数据的数量不一致，要求在一个事务多次读取的数据的数量是一致的，这是insert或delete时引发的问题** |

### MySQL数据库有四种隔离级别

上面的级别最低，下面的级别最高。“是”表示会出现这种问题，“否”表示不会出现这种问题。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **级别** | **名字** | **隔离级别** | **脏读** | **不可重复读** | **幻读** | **数据库默认隔离级别** |
| **1** | 读未提交 | read uncommitted | 是 | 是 | 是 |  |
| **2** | 读已提交 | read committed | 否 | 是 | 是 | **Oracle和SQL Server** |
| **3** | 可重复读 | repeatable read | 否 | 否 | 是 | **MySQL** |
| **4** | **串行化** | **serializable** | **否** | **否** | **否** |  |

### MySQL事务隔离级别相关的命令

* 查询全局事务隔离级别

|  |  |
| --- | --- |
| **查询隔离级别** | **select @@tx\_isolation;** |

* 设置事务隔离级别，需要退出MySQL再重新登录才能看到隔离级别的变化

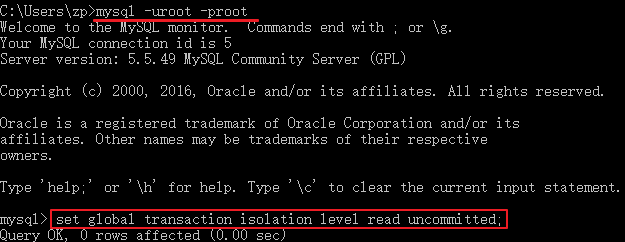
|  |  |
| --- | --- |
| **设置隔离级别** | **set global transaction isolation level 级别字符串;** |

### 脏读的演示

将数据进行恢复：UPDATE account SET balance = 1000;

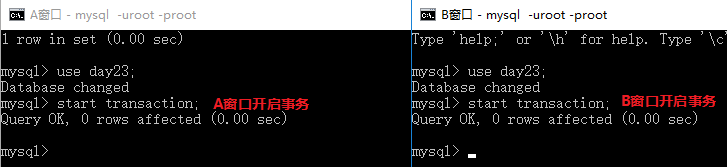
1. 打开A窗口登录MySQL，设置全局的隔离级别为最低

mysql -uroot -proot  
set global transaction isolation level read uncommitted;



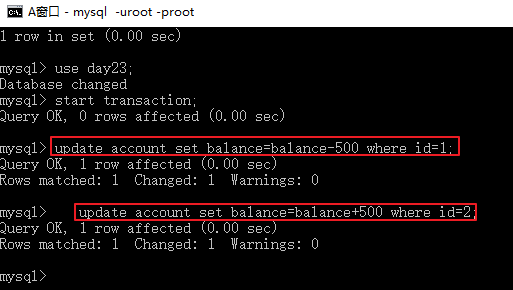
1. 打开B窗口,AB窗口都开启事务

use day23;  
start transaction;



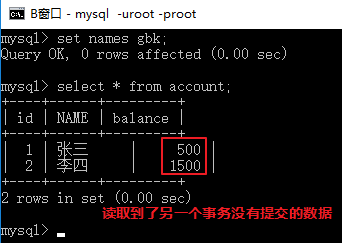
1. A窗口更新2个人的账户数据，未提交

update account set balance=balance-500 where id=1;  
update account set balance=balance+500 where id=2;



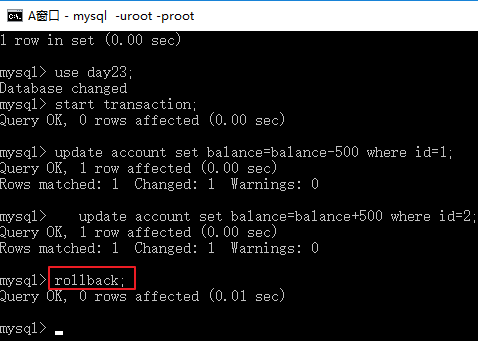
1. B窗口查询账户

select \* from account;



1. A窗口回滚

rollback;



1. B窗口查询账户，钱没了



脏读非常危险的，比如Jack向Rose购买商品，Jack开启事务，向Rose账号转入500块，然后打电话给Rose说钱已经转了。Rose一查询钱到账了，发货给Jack。Jack收到货后回滚事务，Rose的再查看钱没了。

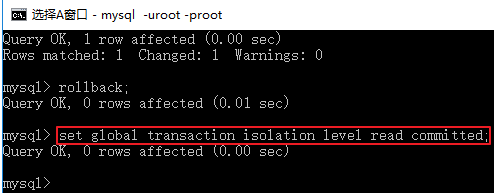
**解决脏读的问题**：将全局的隔离级别进行提升

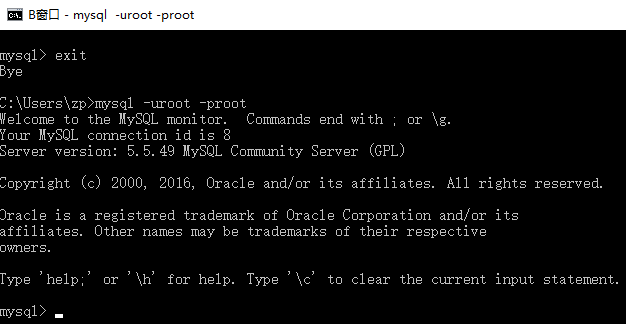
将数据进行恢复：

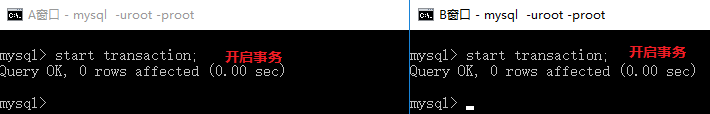
UPDATE account SET balance = 1000;

1. 在A窗口设置全局的隔离级别为read committed

set global transaction isolation level read committed;

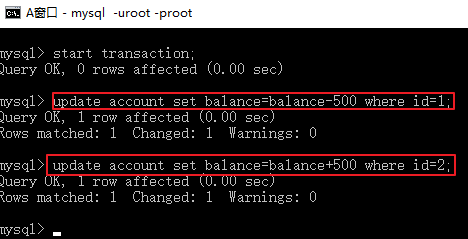


B窗口退出MySQL，B窗口再进入MySQL 

AB窗口同时开启事务 

1. A更新2个人的账户，未提交

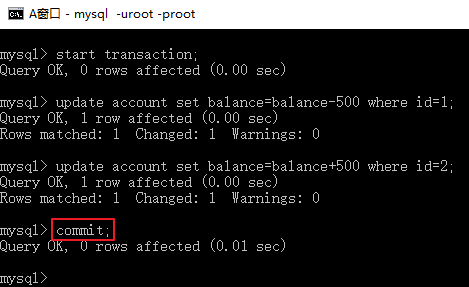
update account set balance=balance-500 where id=1;  
update account set balance=balance+500 where id=2;



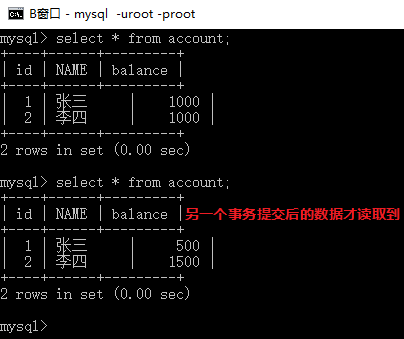
1. B窗口查询账户



A窗口commit提交事务



1. B窗口查看账户



结论：read committed的方式可以避免脏读的发生

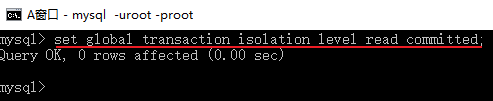
### 不可重复读的演示

将数据进行恢复：

UPDATE account SET balance = 1000;

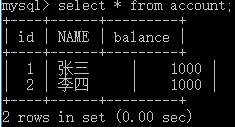
1. 开启A窗口

set global transaction isolation level read committed;



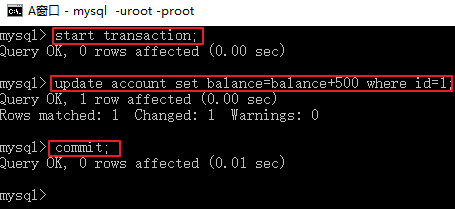
1. 开启B窗口，在B窗口开启事务

start transaction;  
select \* from account;



1. 在A窗口开启事务，并更新数据

start transaction;  
update account set balance=balance+500 where id=1;  
commit;



1. B窗口查询

select \* from account;



两次查询输出的结果不同，到底哪次是对的？不知道以哪次为准。 很多人认为这种情况就对了，无须困惑，当然是后面的为准。我们可以考虑这样一种情况，比如银行程序需要将查询结果分别输出到电脑屏幕和发短信给客户，结果在一个事务中针对不同的输出目的地进行的两次查询不一致，导致文件和屏幕中的结果不一致，银行工作人员就不知道以哪个为准了。

**解决不可重复读的问题**：

将全局的隔离级别进行提升为：repeatable read

将数据进行恢复：

UPDATE account SET balance = 1000;

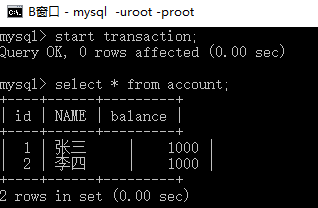
1. A窗口设置隔离级别为：repeatable read

set global transaction isolation level repeatable read;



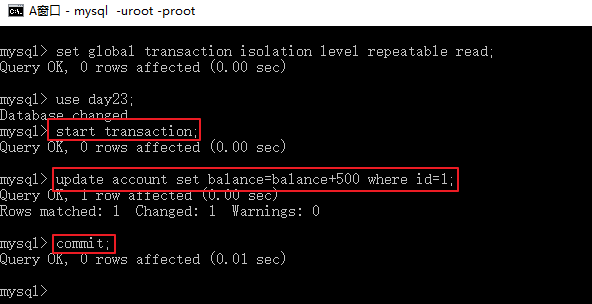
1. B窗口退出MySQL，B窗口再进入MySQL

start transaction;  
select \* from account;



1. A窗口更新数据

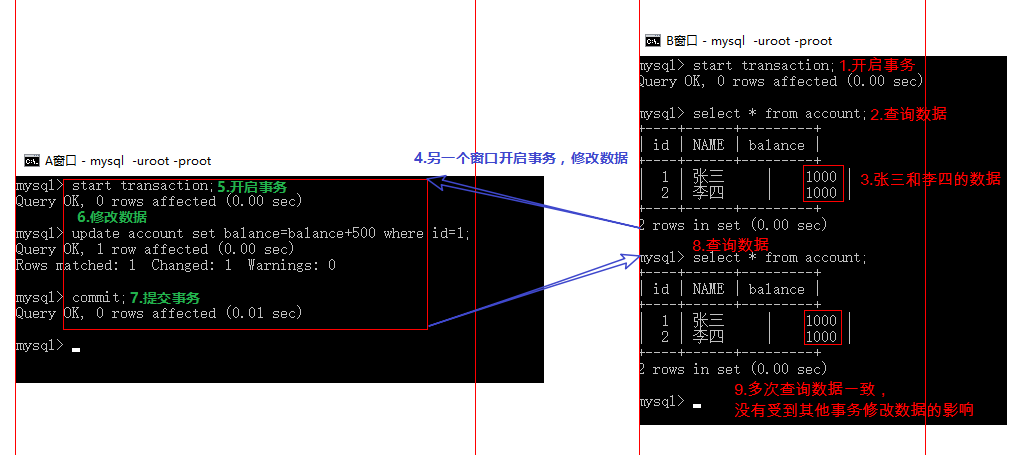
start transaction;  
update account set balance=balance+500 where id=1;  
commit;



1. B窗口查询

select \* from account;



结论：同一个事务中为了保证多次查询数据一致，必须使用repeatable read隔离级别 

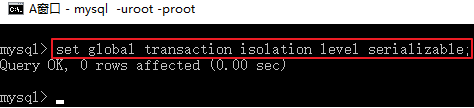
### 幻读的演示

我们可以将事务隔离级别设置到最高，以挡住幻读的发生 将数据进行恢复：

UPDATE account SET balance = 1000;

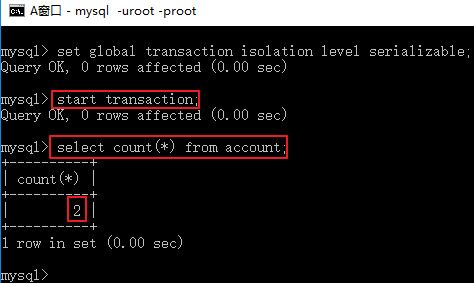
1. 开启A窗口

set global transaction isolation level serializable; -- 设置隔离级别为最高



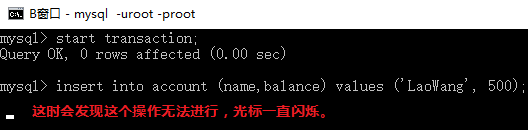
1. A窗口退出MySQL，A窗口重新登录MySQL

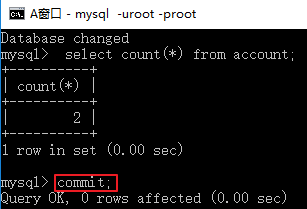
start transaction;  
select count(\*) from account;



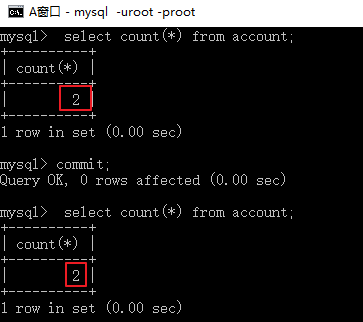
1. 再开启B窗口，登录MySQL
2. 在B窗口中开启事务，添加一条记录

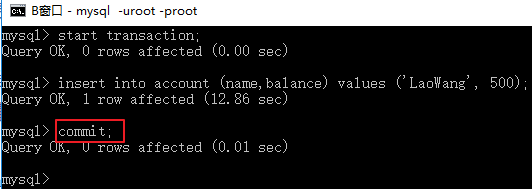
start transaction; -- 开启事务  
insert into account (name,balance) values ('LaoWang', 500);

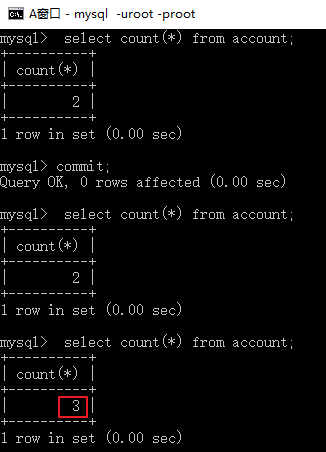


1. 在A窗口中commit提交事务，B窗口中insert语句会在A窗口事务提交后立马运行 
2. 在A窗口中接着查询，发现数据不变

select count(\*) from account;



1. B窗口中commit提交当前事务 
2. A窗口就能看到最新的数据



结论：使用serializable隔离级别，一个事务没有执行完，其他事务的SQL执行不了，可以挡住幻读

# DCL (Data Control Language)

我们现在默认使用的都是root用户，超级管理员，拥有全部的权限。但是，一个公司里面的数据库服务器上面可能同时运行着很多个项目的数据库。所以，我们应该可以根据不同的项目建立不同的用户，分配不同的权限来管理和维护数据库。

## 创建用户

### 语法：

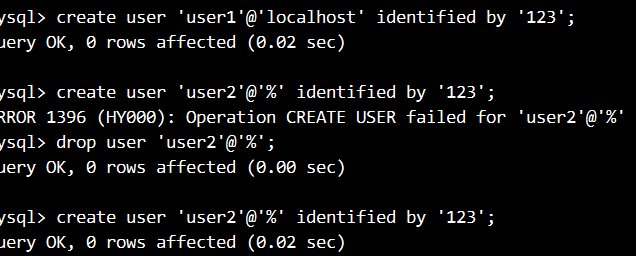
|  |
| --- |
| **CREATE USER '用户名'@'主机名' IDENTIFIED BY '密码';** |
| **注：用户名、主机名和密码都应该加上单引号** |

### 关键字说明：

|  |  |
| --- | --- |
| **关键字** | **说明** |
| **'用户名'** | 将创建的用户名 |
| **'主机名'** | 指定该用户在哪个主机上可以登陆，如果是本地用户可用localhost，如果想让该用户可以从任意远程主机登陆，可以使用通配符% |
| **'密码'** | 该用户的登陆密码，密码可以为空，如果为空则该用户可以不需要密码登陆服务器 |

### 具体操作：

* 创建user1用户，只能在localhost这个服务器登录mysql服务器，密码为123
* 创建user2用户可以在任何电脑上登录mysql服务器，密码为123



* + 注：创建的用户名都在mysql数据库中的user表中可以查看到，密码经过了加密。

## 给用户授权

用户创建之后，没什么权限！需要给用户授权

### 语法：

|  |
| --- |
| **GRANT 权限1, 权限2... ON 数据库名.表名 TO '用户名'@'主机名';** |

### 关键字说明：

|  |  |
| --- | --- |
| **关键字** | **说明** |
| **GRANT…ON…TO** | 授权关键字 |
| **权限** | 授予用户的权限，如CREATE、ALTER、SELECT、INSERT、UPDATE、DELETE等，如果要授予所有的权限则使用ALL。 |
| **数据库名.表名** | 该用户可以操作哪个数据库的哪些表。如果要授予该用户对所有数据库和表的相应操作权限则可用\*表示，如\*.\* |
| **'用户名'@'主机名'** | 给哪个用户授权，要加上单引号。与创建用户时的用户名和主机名要相同。 |

### 具体操作：

1. 给user1用户分配对test这个数据库操作的权限：创建表，修改表，插入记录，更新记录，查询

grant create,alter,insert,update,select on test.\* to 'user1'@'localhost';

1. 给user2用户分配所有权限，对所有数据库的所有表

grant all on \*.\* to 'user2'@'%';

## 撤销授权

### 语法：

|  |
| --- |
| **REVOKE 权限1, 权限2... ON 数据库.表名 FROM '用户名'@'主机名';** |

|  |  |
| --- | --- |
| **关键字** | **说明** |
| **REVOKE…ON…FROM** | 撤销授权的关键字 |
| **权限** | 用户的权限，如CREATE、ALTER、SELECT、INSERT、UPDATE、SELECT等，所有的权限则使用ALL |
| **数据库名.表名** | 对哪些数据库的哪些表，如果要取消该用户对所有数据库和表的操作权限则可用\*表示，如\*.\* |
| **'用户名'@'主机名'** | 给哪个用户撤销，要加上单引号。与创建用户时的用户名和主机名要相同。 |

### 具体操作：

* 撤销user1用户对test数据库所有表的操作的权限

revoke all on test.\* from 'user1'@'localhost';

* + 注：用户名和主机名要与创建时相同，各自要加上单引号

## 查看权限

### 语法：

|  |
| --- |
| **SHOW GRANTS FOR '用户名'@'主机名';** |

### 具体操作：

* 查看user1用户的权限

show grants for 'user1'@'localhost';

* + 注：usage是指连接（登陆）权限，建立一个用户，就会自动授予其usage权限（默认授予）。

## 删除用户

### 语法

|  |
| --- |
| **DROP USER '用户名'@'主机名';** |

### 具体操作：

* 删除user2

drop user 'user2'@'%';

## 修改管理员密码

### 语法

|  |
| --- |
| **mysqladmin -uroot -p password 新密码** |
| **回车后要输出原密码，才能更改成功。如果原密码不正确，则修改失败** |

* + 注意：需要在未登陆MySQL的情况下操作，新密码不需要加上引号。

### 具体操作：

1. 将root管理员的新密码改成123456
2. 要求输入旧密码
3. 使用新密码登录

mysqladmin -uroot -p password 12345

## 修改普通用户密码

### 语法：

|  |
| --- |
| **set password for '用户名'@'主机名' = password('新密码');** |

* + 注意：需要在登陆MySQL的情况下操作，新密码要加单引号。

### 具体操作：

1. 将'user1'@'localhost'的密码改成'666666'
2. 使用新密码登录，老密码登录不了

set password for 'user1'@'localhost' = password('345');