MySQL 多表查询与事务的操作

第1节 回顾

1.1 DQL 查询

1.1.1 排序使用什么子句：order by

- 升序：asc
- 降序：desc

1.1.2 聚合函数

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

1.1.3 分页查询

```
limit 起始值从 0 开始，长度
```

1.1.4 分组查询

```
group by 分组列 having 过滤条件
```

1.2 约束

1.2.1 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

1.2.2 自增长的关键字：

```
auto_increment
```

1.2.3 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

1.3 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	通过主外键约束
多对多	通过中间表，中间表与两个表是多对一
一对一	1. 特殊的一对多，多方加唯一约束

	2. 从表的主键同时又是外键

1.4 数据库的三大范式

范式	特点
第 1 范式	原子性，每列不可再拆分
第 2 范式	不产生局部依赖，表中每一列都完全依赖于主键。
第 3 范式	不产生传递，表中每一列都直接依赖于主键

第2节 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能够使用多表进行查询
4) 能够理解事务的概念
5) 能够说出事务的特点
6) 能够在 MySQL 中使用事务
7) 能够理解脏读、不可重复读、幻读的概念及解决办法
8) 能够使用 DCL 管理 MySQL 中的用户

第3节 表连接查询

3.1 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);
```

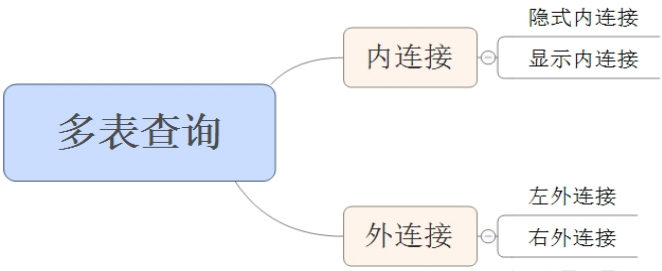
```
insert into emp(name,gender,salary,join_date,dept_id) values('唐僧','男',9000,'2008-08-08',2);
insert into emp(name,gender,salary,join_date,dept_id) values('白骨精','女',5000,'2015-10-07',3);
insert into emp(name,gender,salary,join_date,dept_id) values('蜘蛛精','女',4500,'2011-03-14',1);
```

- 多表查询的作用：

  比如：我们想查询孙悟空的名字和他所在的部门的名字，则需要使用多表查询。

  如果一条 SQL 语句查询多张表，因为查询结果在多张不同的表中。每张表取 1 列或多列。

3.1.1 多表查询的分类：



3.2 笛卡尔积现象

3.2.1 什么是笛卡尔积现象

- 什么是笛卡尔积：

```
-- 需求：查询所有的员工和所有的部门
select * from emp,dept;
```

- 结果分析：

id	NAME
1	开发部
2	市场部
3	财务部

id	NAME	gender	salary	join_date	dept_id
1	孙悟空	男	7200	2013-02-24	1
2	猪八戒	男	3600	2010-12-02	2
3	唐僧	男	9000	2008-08-08	2
4	白骨精	女	5000	2015-10-07	3
5	蜘蛛精	女	4500	2011-03-14	1

左表的每条数据和右表的每条数据组合，这种效果成为笛卡尔乘积

3.2.2 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 Column 'id' in where clause is ambiguous
select * from emp,dept where id=5;

select * from emp,dept where emp.`dept_id` = dept.`id`;

-- 查询员工和部门的名字
select emp.`name`, dept.`name` from emp,dept where emp.`dept_id` = dept.`id`;
```

3.3 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。如：从表.外键=主表.主键

3.3.1 隐式内连接

- 隐式内连接：看不到 JOIN 关键字，条件使用 WHERE 指定

```
SELECT 字段名 FROM 左表, 右表 WHERE 条件
```

```
select * from emp,dept where emp.`dept_id` = dept.`id`;
```

id	name	gender	salary	join_date	dept_id	id	name
1	孙悟空	男	7200	2013-02-24	1	1	开发部
2	猪八戒	男	3600	2010-12-02	2	2	市场部
3	唐僧	男	9000	2008-08-08	2	2	市场部
4	白骨精	女	5000	2015-10-07	3	3	财务部
5	蜘蛛精	女	4500	2011-03-14	1	1	开发部

3.3.2 显式内连接

- 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

id	NAME
1	开发部
2	市场部
3	财务部
4	销售部

id	NAME	gender	salary	join_date	dept_id
1	孙悟空	男	7200	2013-02-24	1
2	猪八戒	男	3600	2010-12-02	2
3	唐僧	男	9000	2008-08-08	2
4	白骨精	女	5000	2015-10-07	3
5	蜘蛛精	女	4500	2011-03-14	1

1) 确定查询哪些表

```
select * from emp inner join dept;
```

id	NAME	id	NAME	gender	salary	join_date	dept_id
1	开发部	1	孙悟空	男	7200	2013-02-24	1
2	市场部	1	孙悟空	男	7200	2013-02-24	1
3	财务部	1	孙悟空	男	7200	2013-02-24	1
1	开发部	2	猪八戒	男	3600	2010-12-02	2
2	市场部	2	猪八戒	男	3600	2010-12-02	2
3	财务部	2	猪八戒	男	3600	2010-12-02	2
1	开发部	3	唐僧	男	9000	2008-08-08	2
2	市场部	3	唐僧	男	9000	2008-08-08	2
3	财务部	3	唐僧	男	9000	2008-08-08	2
1	开发部	4	白骨精	女	5000	2015-10-07	3
2	市场部	4	白骨精	女	5000	2015-10-07	3
3	财务部	4	白骨精	女	5000	2015-10-07	3
1	开发部	5	蜘蛛精	女	4500	2011-03-14	1
2	市场部	5	蜘蛛精	女	4500	2011-03-14	1
3	财务部	5	蜘蛛精	女	4500	2011-03-14	1

2) 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 emp e inner join dept d on e.`dept_id` = d.`id`;
```

id	NAME	id	NAME	gender	salary	join_date	dept_id
1	开发部	5	蜘蛛精	女	4500	2011-03-14	1
1	开发部	1	孙悟空	男	7200	2013-02-24	1
2	市场部	3	唐僧	男	9000	2008-08-08	2
2	市场部	2	猪八戒	男	3600	2010-12-02	2
3	财务部	4	白骨精	女	5000	2015-10-07	3

3) 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 emp e inner join dept d on e.`dept_id` = d.`id` where e.`name`='唐僧
';
```

id	NAME	id	NAME	gender	salary	join_date	dept_id
2	市场部	3	唐僧	男	9000	2008-08-08	2

4) 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 e.`id`,e.`name`,e.`gender`,e.`salary`,d.`name` from emp e inner join dept d
on e.`dept_id` = d.`id` where e.`name`='唐僧';
```

id	NAME	gender	salary	NAME
3	唐僧	男	9000	市场部

5) 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 e.`id` 编号,e.`name` 姓名,e.`gender` 性别,e.`salary` 工资,d.`name` 部门名字 from
emp e inner join dept d on e.`dept_id` = d.`id` where e.`name`='唐僧';
```

员工编号	员工姓名	性别	工资	部门名称
3	唐僧	男	9000	市场部

3.3.3 总结内连接查询步骤：

1) 确定查询哪些表
2) 确定表连接的条件
3) 确定查询的条件
4) 确定查询的字段

3.4 左外连接

- 左外连接：使用 LEFT OUTER 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(左表是部门，右表员工)

```
-- 在部门表中增加一个销售部
insert into dept (name) values ('销售部');
select * from dept;

-- 使用内连接查询
select * from dept d inner join emp e on d.`id` = e.`dept_id`;

-- 使用左外连接查询
select * from dept d left join emp e on d.`id` = e.`dept_id`;
```

id	NAME	id	NAME	gender	salary	join_date	dept_id
1	开发部	1	孙悟空	男	7200	2013-02-24	1
1	开发部	5	蜘蛛精	女	4500	2011-03-14	1
2	市场部	2	猪八戒	男	3600	2010-12-02	2
2	市场部	3	唐僧	男	9000	2008-08-08	2
3	财务部	4	白骨精	女	5000	2015-10-07	3
4	销售部	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)

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

3.5 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

```
-- 在员工表中增加一个员工
```

id	NAME	gender	salary	join_date	dept_id
1	孙悟空	男	7200	2013-02-24	1
2	猪八戒	男	3600	2010-12-02	2
3	唐僧	男	9000	2008-08-08	2
4	白骨精	女	5000	2015-10-07	3
5	蜘蛛精	女	4500	2011-03-14	1
6	沙僧	男	6666	2013-02-24	(NULL)

```
-- 在员工表中增加一个员工
insert into emp values (null, '沙僧','男',6666,'2013-12-05',null);
select * from emp;
-- 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`;
```

id	NAME	id	NAME	gender	salary	join_date	dept_id
1	开发部	1	孙悟空	男	7200	2013-02-24	1
2	市场部	2	猪八戒	男	3600	2010-12-02	2
2	市场部	3	唐僧	男	9000	2008-08-08	2
3	财务部	4	白骨精	女	5000	2015-10-07	3
1	开发部	5	蜘蛛精	女	4500	2011-03-14	1
(NULL)	(NULL)	6	沙僧	男	6666	2013-02-24	(NULL)

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

第4节 子查询

4.1 什么是子查询

```
-- 需求：查询开发部中有哪些员工
```

```
select * from emp;

-- 通过两条语句查询
select id from dept where name='开发部' ;
select * from emp where dept_id = 1;
-- 使用子查询
select * from emp where dept_id = (select id from dept where name='市场部');
```

- 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) 有查询的嵌套，内部的查询称为子查询
3) 子查询要使用括号

4.2 子查询结果的三种情况：

1) 子查询的结果是单行单列

max(salary)
9000

2) 子查询的结果是多行单列

dept_id
1
2

3) 子查询的结果是多行多列

id	NAME	gender	salary	join_date	dept_id
1	孙悟空	男	7200	2013-02-24	1
4	白骨精	女	5000	2015-10-07	3
5	蜘蛛精	女	4500	2011-03-14	1
6	沙僧	男	6666	2013-02-24	*(NULL)*

4.3 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）;
```

4.3.1 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 = (select max(salary) from emp);
```

4.3.2 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);
```

4.4 子查询结果是多行单列的时候

- 子查询结果是单例多行，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）;

4.4.1 查询工资大于 5000 的员工，来自于哪些部门的名字

```
-- 先查询大于 5000 的员工所在的部门 id
select dept_id from emp where salary > 5000;
-- 再查询在这些部门 id 中部门的名字  Subquery returns more than 1 row
select name from dept where id = (select dept_id from emp where salary > 5000);

select name from dept where id in (select dept_id from emp where salary > 5000);
```

4.4.2 查询开发部与财务部所有的员工信息

```
-- 先查询开发部与财务部的 id
select id from dept where name in('开发部','财务部');
-- 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 in('开发部','财务部'));
```

4.5 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

4.5.1 查询出 2011 年以后入职的员工信息，包括部门名称

```
-- 查询出 2011 年以后入职的员工信息，包括部门名称
-- 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';

-- 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 d, (select * from emp where join_date >='2011-1-1') e where d.`id`= e.dept_id ;
```

-- 也可以使用表连接：

```
select * from emp inner join dept on emp.`dept_id` = dept.`id` where join_date >='2011-1-1';
select * from emp inner join dept on emp.`dept_id` = dept.`id` and join_date >='2011-1-1';
```

4.6 子查询小结

- 子查询结果只要是单列，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

第5节 事务

5.1 事务的应用场景说明

- 什么是事务： 在实际的开发过程中，一个业务操作如：转账，往往是要多次访问数据库才能完成的。转账是一个用户扣钱，另一个用户加钱。如果其中有一条 SQL 语句出现异常，这条 SQL 就可能执行失败。

事务执行是一个整体,所有的 SQL 语句都必须执行成功。如果其中有 1 条 SQL 语句出现异常,则所有的 SQL 语句都要回滚，整个业务执行失败。

- 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 ('张三', 1000), ('李四', 1000);
```

模拟张三给李四转 500 元钱，一个转账的业务操作最少要执行下面的 2 条语句：
张三账号-500
李四账号+500

```
-- 张三账号-500
update account set balance = balance - 500 where name='张三';
-- 李四账号+500
update account set balance = balance + 500 where name='李四';
```

假设当张三账号上-500 元,服务器崩溃了。李四的账号并没有+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。

5.2 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

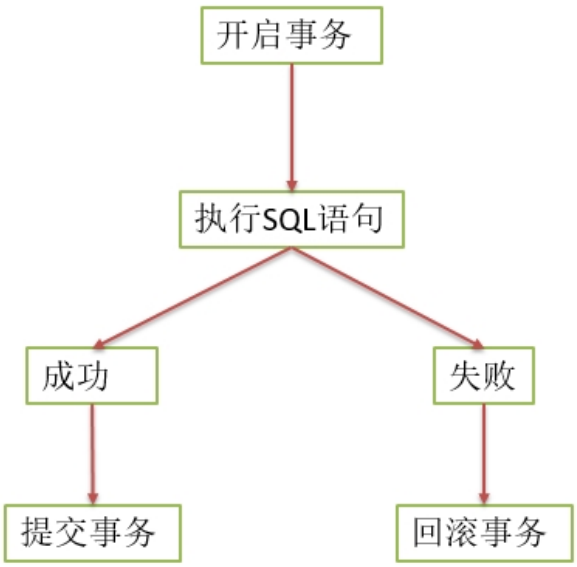
5.2.1 手动提交事务的 SQL 语句

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

5.2.2 手动提交事务使用过程：

1) 执行成功的情况： 开启事务 → 执行多条 SQL 语句 → 成功提交事务

2) 执行失败的情况： 开启事务 → 执行多条 SQL 语句 → 事务的回滚



5.2.3 案例演示 1：事务提交

模拟张三给李四转 500 元钱（成功） 目前数据库数据如下：

id	name	balance
1	张三	1000
2	李四	1000

1) 使用 DOS 控制台进入 MySQL
2) 执行以下 SQL 语句： 1.开启事务， 2.张三账号-500， 3.李四账号+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 提交事务：
5) 使用 SQLYog 查看数据库：发现数据改变

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> update account set balance = balance - 500 where name='张三';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> update account set balance = balance + 500 where name='李四';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from account;
+----+------+---------+
| id | NAME | balance |
+----+------+---------+
|  1 | 张三     |     500 |
|  2 | 李四     |    1500 |
+----+------+---------+
2 rows in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

5.2.4 案例演示 2：事务回滚

模拟张三给李四转 500 元钱（失败） 目前数据库数据如下：

id	name	balance
1	张三	1000
2	李四	1000

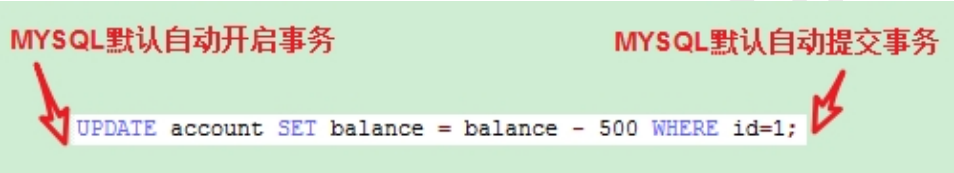
1) 在控制台执行以下 SQL 语句：1.开启事务， 2.张三账号-500
2) 使用 SQLYog 查看数据库：发现数据并没有改变
3) 在控制台执行 rollback 回滚事务：
4) 使用 SQLYog 查看数据库：发现数据没有改变

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> update account set balance = balance - 500 where name='张三';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> update account set balance = balance + 500 where name='李四';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from account;
+----+------+---------+
| id | NAME | balance |
+----+------+---------+
|  1 | 张三     |     500 |
|  2 | 李四     |    1500 |
+----+------+---------+
2 rows in set (0.00 sec)

mysql> rollback;
```

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。

5.3 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，语句执行完毕自动提交事务，MySQL 默认开始自动提交事务



5.3.1 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

id	NAME	balance
1	张三	500
2	李四	1500

```
mysql> update account set balance = balance + 500 where name='李四';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from account;
+----+------+---------+
| id | NAME | balance |
+----+------+---------+
|  1 | 张三     |    1000 |
|  2 | 李四     |    1500 |
+----+------+---------+
```

5.3.2 取消自动提交

- 查看 MySQL 是否开启自动提交事务

```
mysql> select @@autocommit;
+--------------+
| @@autocommit |
+--------------+
|            1 |
+--------------+
1 row in set (0.00 sec)
```

@@表示全局变量，1 表示开启，0 表示关闭

- 取消自动提交事务

```
mysql> set @@autocommit = 0;
Query OK, 0 rows affected (0.00 sec)

mysql> select @@autocommit;
+--------------+
| @@autocommit |
+--------------+
|            0 |
+--------------+
1 row in set (0.00 sec)
```

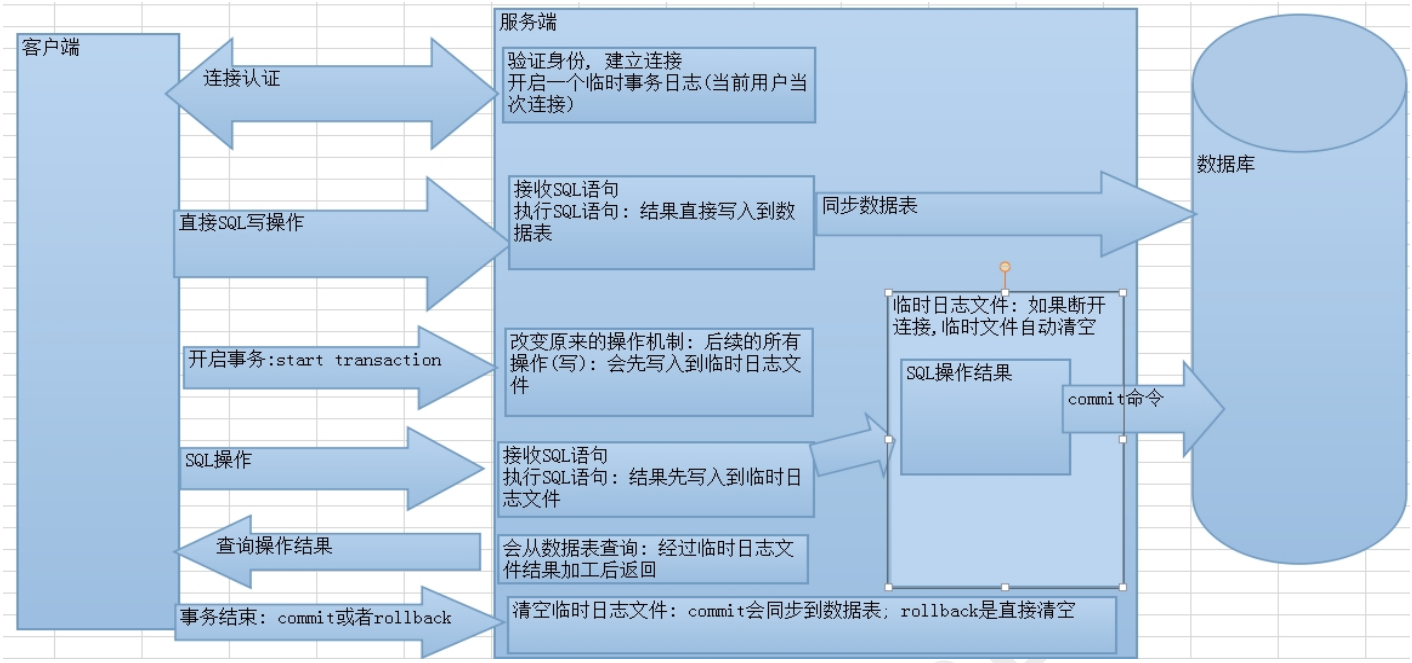
- 执行更新语句，使用 SQLYog 查看数据库，发现数据并没有改变
- 在控制台执行 commit 提交任务

```
mysql> select * from account;
+----+------+---------+
| id | NAME | balance |
+----+------+---------+
|  1 | 张三     |    1000 |
|  2 | 李四     |    1500 |
+----+------+---------+
2 rows in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

5.4 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)

5.4.1 原理图：



5.4.2 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) 如果事务提交则将日志文件中的数据写到表中，否则清空日志文件。

5.5 回滚点

5.5.1 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。

5.5.2 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

5.5.3 具体操作：

1) 将数据还原到 1000
2) 开启事务
3) 让张三账号减 3 次钱，每次 10 块
4) 设置回滚点：savepoint three_times;
5) 让张三账号减 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。

5.6 事务的隔离级别

5.6.1 事务的四大特性 ACID

事务特性	含义
原子性（Atomicity）	每个事务都是一个整体，不可再拆分，事务中所有的 SQL 语句要么都执行成功，要么都失败。
一致性（Consistency）	事务在执行前数据库的状态与执行后数据库的状态保持一致。如：转账前 2 个人的总金额是 2000，转账后 2 个人总金额也是 2000
隔离性（Isolation）	事务与事务之间不应该相互影响，执行时保持隔离的状态。
持久性（Durability）	一旦事务执行成功，对数据库的修改是持久的。就算关机，也是保存下来的。

5.6.2 事务的隔离级别

事务在操作时的理想状态： 所有的事务之间保持隔离，互不影响。因为并发操作，多个用户同时访问同一个数据。可能引发并发访问的问题：

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

5.6.3 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
						`mysql> select @@tx_isolation;`
						`+-----------------+`
						`| @@tx_isolation |`
						`+-----------------+`
						`| REPEATABLE-READ |`
						`+-----------------+`

4	串行化	serializable	否	否	否	

✧ 隔离级别越高，性能越差，安全性越高。

5.6.4 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;

5.6.5 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;
```

```
C:\Users\zp>mysql -uroot -proot
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5
Server version: 5.5.49 MySQL Community Server (GPL)

Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> set global transaction isolation level read uncommitted;
Query OK, 0 rows affected (0.00 sec)
```

2. 打开 B 窗口,AB 窗口都开启事务

```
use day23;
start transaction;
```

A窗口 - mysql -uroot -proot

```
1 row in set (0.00 sec)

mysql> use day23;
Database changed
mysql> start transaction;  A窗口开启事务
Query OK, 0 rows affected (0.00 sec)

mysql>
```

B窗口 - mysql -uroot -proot

```
Type 'help;' or '\h' for help. Type '\c'

mysql> use day23;
Database changed
mysql> start transaction;  B窗口开启事务
Query OK, 0 rows affected (0.00 sec)

mysql>
```

3. A 窗口更新 2 个人的账户数据，未提交

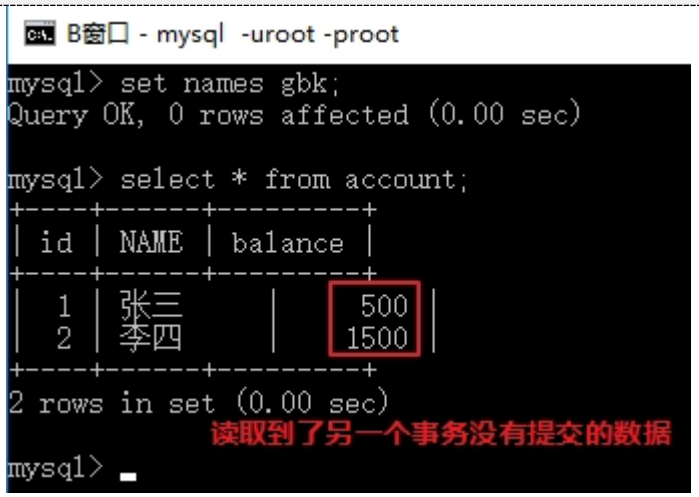
```
update account set balance=balance-500 where id=1;
update account set balance=balance+500 where id=2;
```

A窗口 - mysql -uroot -proot

```
1 row in set (0.00 sec)

mysql> use day23;
Database changed
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> update account set balance=balance-500 where id=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>  update account set balance=balance+500 where id=2;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>
```

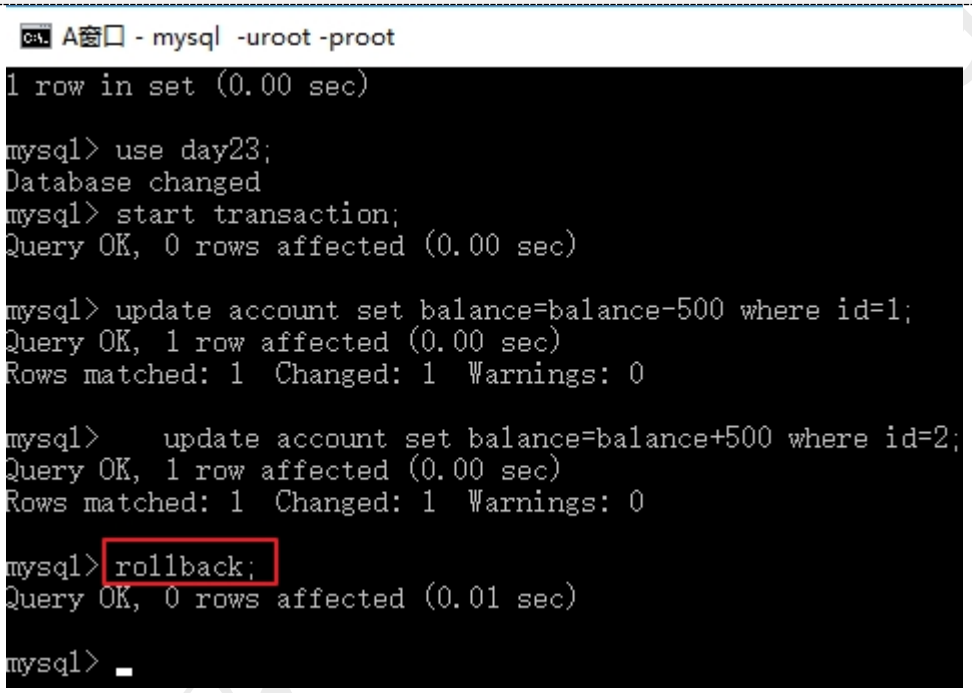
4. B 窗口查询账户

```
select * from account;
```



5. A 窗口回滚

```
rollback;
```



6. B 窗口查询账户，钱没了

脏读非常危险的，比如张三向李四购买商品，张三开启事务，向李四账号转入 500 块，然后打电话给李四说钱已经转了。李四一查询钱到账了，发货给张三。张三收到货后回滚事务，李四的再查看钱没了。

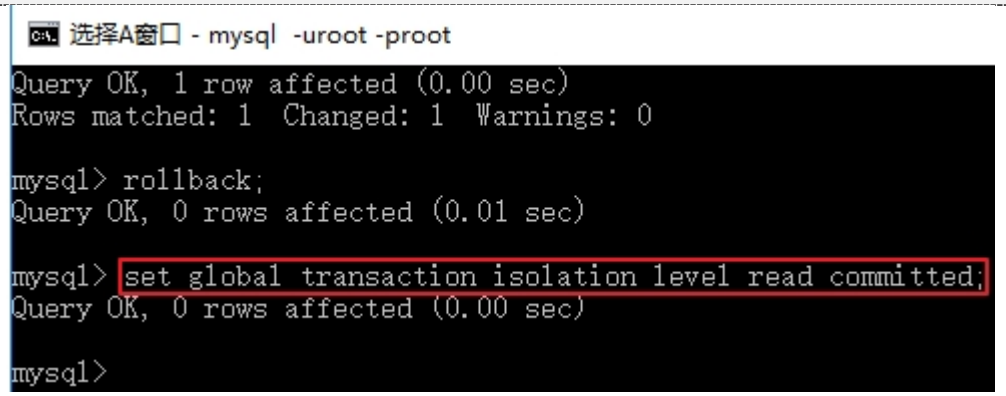
解决脏读的问题：将全局的隔离级别进行提升

将数据进行恢复：

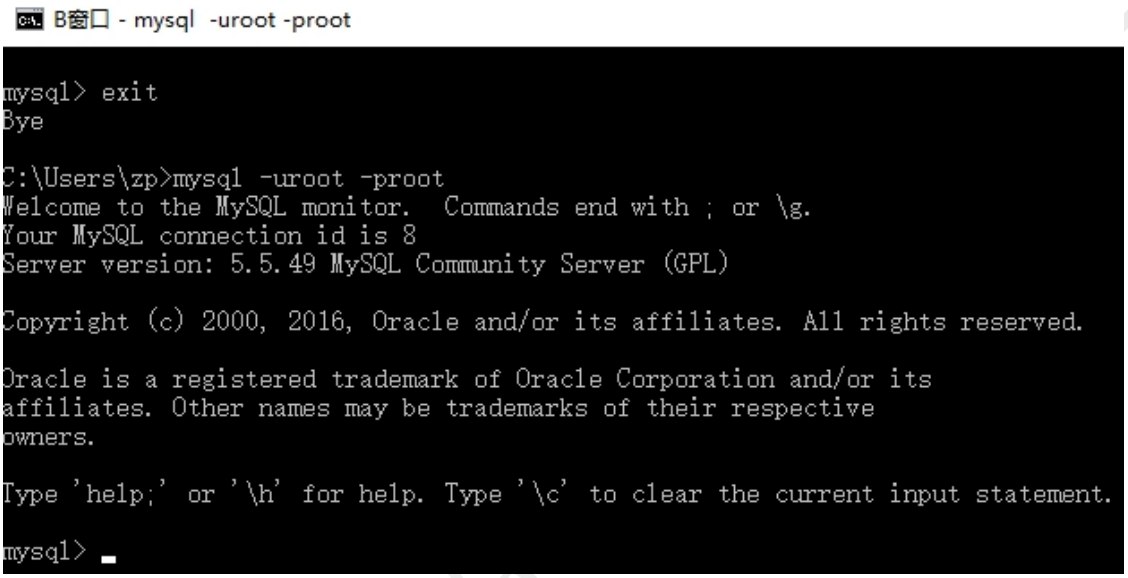
```
UPDATE account SET balance = 1000;
```

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

2. 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;
```

A窗口 - mysql -uroot -proot

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> update account set balance=balance-500 where id=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> update account set balance=balance+500 where id=2;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>
```

3. B 窗口查询账户

B窗口 - mysql -uroot -proot

```
mysql> select * from account;
+----+------+---------+
| id | NAME | balance |
+----+------+---------+
|  1 | 张三 |    1000 |
|  2 | 李四 |    1000 |
+----+------+---------+
2 rows in set (0.00 sec)

mysql>
```

没有读取到另一个事务未提交的事务

A 窗口 commit 提交事务

A窗口 - mysql -uroot -proot

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> update account set balance=balance-500 where id=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> update account set balance=balance+500 where id=2;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> commit;
Query OK, 0 rows affected (0.01 sec)

mysql>
```

4. B 窗口查看账户

B窗口 - mysql -uroot -proot

```
mysql> select * from account;
+----+------+---------+
| id | NAME | balance |
+----+------+---------+
|  1 | 张三 |    1000 |
|  2 | 李四 |    1000 |
+----+------+---------+
2 rows in set (0.00 sec)

mysql> select * from account;
+----+------+---------+
| id | NAME | balance |
+----+------+---------+
|  1 | 张三 |     500 |
|  2 | 李四 |    1500 |
+----+------+---------+
2 rows in set (0.00 sec)

mysql>
```

另一个事务提交后的数据才读取到

结论：read committed 的方式可以避免脏读的发生

5.6.6 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;
```

A窗口 - mysql -uroot -proot

```
mysql> set global transaction isolation level read committed;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

2. 开启 B 窗口，在 B 窗口开启事务

```
start transaction;
select * from account;
```

```
mysql> select * from account;
+----+------+---------+
| id | NAME | balance |
+----+------+---------+
|  1 | 张三 |    1000 |
|  2 | 李四 |    1000 |
+----+------+---------+
2 rows in set (0.00 sec)
```

3. 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;
```

A窗口 - mysql -uroot -proot

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> update account set balance=balance+500 where id=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> commit;
Query OK, 0 rows affected (0.01 sec)

mysql>
```

4. B 窗口查询

```
select * from account;
```

B窗口 - mysql -uroot -proot

```
mysql> select * from account;
+----+------+---------+
| id | NAME | balance |
+----+------+---------+
|  1 | 张三 |    1000 |
|  2 | 李四 |    1000 |
+----+------+---------+
2 rows in set (0.00 sec)

mysql> select * from account;
+----+------+---------+
| id | NAME | balance |
+----+------+---------+
|  1 | 张三 |    1500 |
|  2 | 李四 |    1000 |
+----+------+---------+
2 rows in set (0.00 sec)

mysql>
```

两次查询输出的结果不同，到底哪次是对的？

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;
```

```
mysql> set global transaction isolation level repeatable read;
Query OK, 0 rows affected (0.00 sec)
```

2. B 窗口退出 MySQL，B 窗口再进入 MySQL

```
start transaction;
select * from account;
```

B窗口 - mysql -uroot -proot

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from account;
+----+------+---------+
| id | NAME | balance |
+----+------+---------+
|  1 | 张三 |    1000 |
|  2 | 李四 |    1000 |
+----+------+---------+
2 rows in set (0.00 sec)
```

3. A 窗口更新数据

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

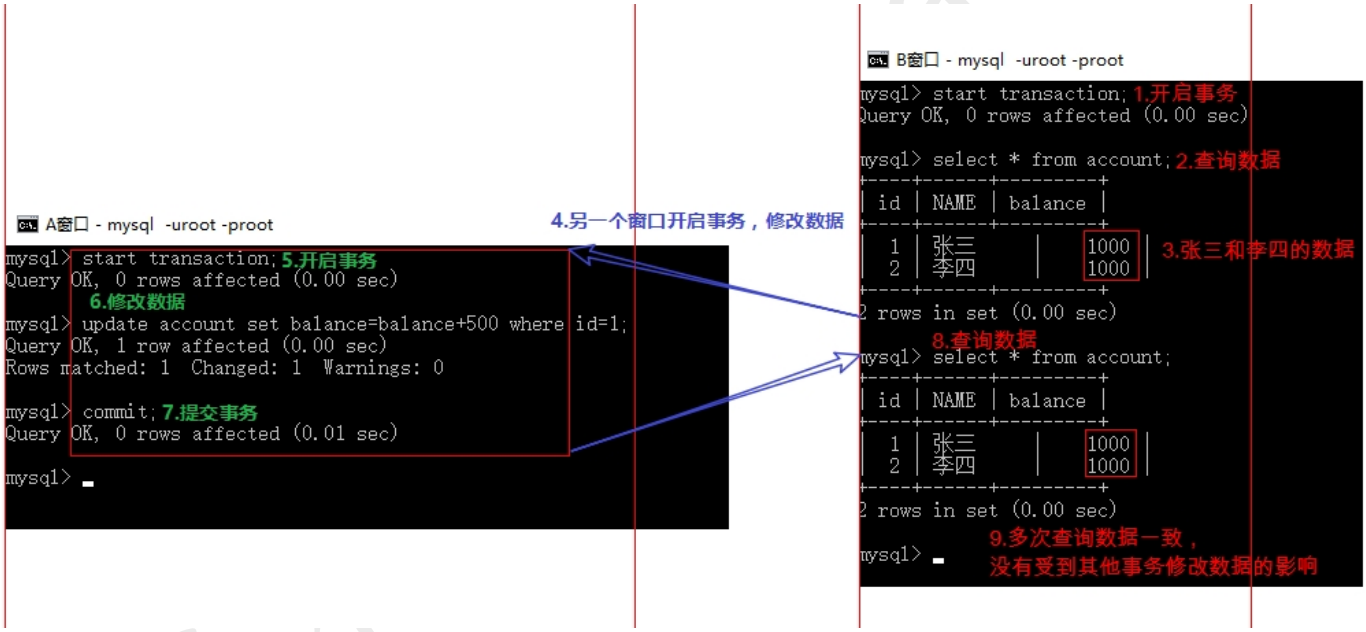
A窗口 - mysql -uroot -proot

```
mysql> set global transaction isolation level repeatable read;
Query OK, 0 rows affected (0.00 sec)

mysql> use day23;
Database changed
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> update account set balance=balance+500 where id=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> commit;
Query OK, 0 rows affected (0.01 sec)

mysql>
```

4. 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



5.6.7 幻读的演示

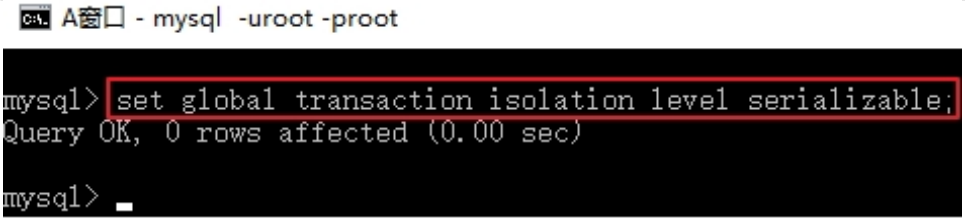
在 MySQL 中无法看到幻读的效果。

但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
```



2. A 窗口退出 MySQL，A 窗口重新登录 MySQL

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

A窗口 - mysql -uroot -proot

```
mysql> set global transaction isolation level serializable;
Query OK, 0 rows affected (0.00 sec)

mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select count(*) from account;
+----------+
| count(*) |
+----------+
|        2 |
+----------+
1 row in set (0.00 sec)

mysql>
```

3. 再开启 B 窗口，登录 MySQL

4. 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);
```

B窗口 - mysql -uroot -proot

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into account (name,balance) values ('LaoWang', 500);
这时会发现这个操作无法进行，光标一直闪烁。
```

5. 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

A窗口 - mysql -uroot -proot

```
Database changed
mysql>  select count(*) from account;
+----------+
| count(*) |
+----------+
|        2 |
+----------+
1 row in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

6. 在 A 窗口中接着查询，发现数据不变

```
select count(*) from account;
```

A窗口 - mysql -uroot -proot

```
mysql> select count(*) from account;
+----------+
| count(*) |
+----------+
|        2 |
+----------+
1 row in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql> select count(*) from account;
+----------+
| count(*) |
+----------+
|        2 |
+----------+
1 row in set (0.00 sec)
```

7. B 窗口中 commit 提交当前事务

B窗口 - mysql -uroot -proot

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into account (name,balance) values ('LaoWang', 500);
Query OK, 1 row affected (12.86 sec)

mysql> commit;
Query OK, 0 rows affected (0.01 sec)

mysql>
```

8. A 窗口就能看到最新的数据

A窗口 - mysql -uroot -proot

```
mysql> select count(*) from account;
+----------+
| count(*) |
+----------+
|        2 |
+----------+
1 row in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql> select count(*) from account;
+----------+
| count(*) |
+----------+
|        2 |
+----------+
1 row in set (0.00 sec)

mysql> select count(*) from account;
+----------+
| count(*) |
+----------+
|        3 |
+----------+
1 row in set (0.00 sec)
```

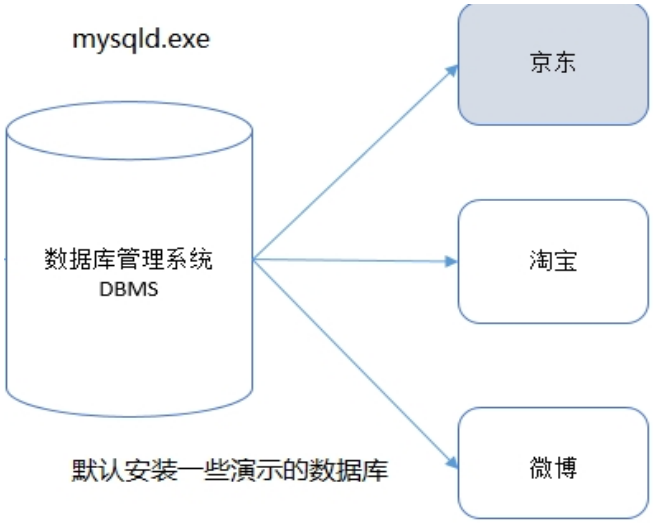
结论：使用 serializable 隔离级别，一个事务没有执行完，其他事务的 SQL 执行不了，可以挡住幻读

第6节 DCL (Data Control Language)

- DDL： create / alter / drop
- DML：insert /update/delete
- DQL ：select /show
- DCL ：grant /revoke

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。

注：mysqld 是 MySQL 的主程序，服务器端。mysql 是 MySQL 的命令行工具，客户端。



6.1 创建用户

6.1.1 语法：

```
CREATE USER '用户名'@'主机名' IDENTIFIED BY '密码';
```

6.1.2 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

6.1.3 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

```
create user 'user1'@'localhost' identified by '123';
```

- 创建 user2 用户可以在任何电脑上登录 mysql 服务器，密码为 123

```
create user 'user2'@'%' identified by '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。

Host	User	Password
localhost	root	*81F5E21E35407D884A6CD4A731AEBFB6AF209E1B
localhost	user1	*23AE809DDACAF96AF0FD78ED04B6A265E05AA257
%	user2	*23AE809DDACAF96AF0FD78ED04B6A265E05AA257

6.2 给用户授权

用户创建之后，没什么权限！需要给用户授权

6.2.1 语法：

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

6.2.2 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 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，注：有 2 对单引号

6.2.3 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';
```

✧ 注：用户名和主机名要与上面创建的相同，要加单引号。

2. 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'@'%';
```

6.3 撤销授权

6.3.1 语法：

```
REVOKE 权限 1, 权限 2... ON 数据库.表名 revoke all on test.* from 'user1'@'localhost'; '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 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

6.3.2 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

6.4 查看权限

6.4.1 语法：

```
SHOW GRANTS FOR '用户名'@'主机名';
```

6.4.2 具体操作：

- 查看 user1 用户的权限

```
mysql> show grants for 'user1'@'localhost';
+-----------------------------------------------------------------------------------------------------------------+
| Grants for user1@localhost                                                                                      |
+-----------------------------------------------------------------------------------------------------------------+
| GRANT USAGE ON *.* TO 'user1'@'localhost' IDENTIFIED BY PASSWORD '*23AE809DDACAF96AF0FD78ED04B6A265E05AA257' |
+-----------------------------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> show grants for 'user2'@'%';
+-----------------------------------------------------------------------------------------------------------------+
| Grants for user2@%                                                                                              |
+-----------------------------------------------------------------------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'user2'@'%' IDENTIFIED BY PASSWORD '*23AE809DDACAF96AF0FD78ED04B6A265E05AA257' |
```

✧ 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）。

6.5 删除用户

6.5.1 语法

```
DROP USER '用户名'@'主机名';
```

6.5.2 具体操作：

- 删除 user2

```
drop user 'user2'@'%';
```

```
C:\>mysql -uuser2 -p123;
ERROR 1045 (28000): Access denied for user 'user2'@'localhos
YES)
```

6.6 修改管理员密码

6.6.1 语法

```
mysqladmin -uroot -p password 新密码
```

✧ 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。

6.6.2 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

```
C:\>mysqladmin -uroot -p password 123456
Enter password: ****

C:\>mysql -uroot -proot
ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: YES)

C:\>mysql -uroot -p123456
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 34
Server version: 5.5.40 MySQL Community Server (GPL)
```

6.7 修改普通用户密码

6.7.1 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。

6.7.2 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

```
mysql> set password for 'user1'@'localhost'=password('66666');
Query OK, 0 rows affected (0.00 sec)
```