1. JDBC

目录

[1 课程计划 1](#_Toc2551706)

[2 JDBC概述 1](#_Toc2551707)

[2.1 什么是JDBC 1](#_Toc2551708)

[2.2 为什么要学习JDBC 1](#_Toc2551709)

[2.3 如何使用JDBC访问数据库 1](#_Toc2551710)

[2.4 JDBC开发细节 1](#_Toc2551711)

[3 JDBC增删改查 1](#_Toc2551712)

[4 PreparedStatement对象 1](#_Toc2551713)

[4.1 模拟用户登陆案例 1](#_Toc2551714)

[4.2 SQL注入攻击 1](#_Toc2551715)

[4.3 防止SQL注入攻击 1](#_Toc2551716)

[5 数据库连接池 1](#_Toc2551717)

[5.1 什么是连接池 1](#_Toc2551718)

[5.2 为什么要使用数据库连接池 1](#_Toc2551719)

[5.3 如何使用C3P0连接池 1](#_Toc2551720)

[6 事务及四大特性 1](#_Toc2551721)

[6.1 什么是事务 1](#_Toc2551722)

[6.2 事务的四大特性 1](#_Toc2551723)

[6.3 MySQL中的事务 1](#_Toc2551724)

[7 事务并发读问题 1](#_Toc2551725)

[7.1 事务并发读问题 1](#_Toc2551726)

[7.2 事务隔离级别 1](#_Toc2551727)

[7.3 设置隔离级别 1](#_Toc2551728)

# 课程计划

1、了解JDBC的作用

2、掌握使用JDBC访问数据库，完成增删改查操作

3、了解SQL注入及掌握PreparedStatement的使用

4、掌握连接池的原理及c3p0连接池的使用

5、了解事务的作用及掌握事务的四大特性

6、熟悉事务的三个读问题及四个隔离级别

# JDBC概述

## 什么是JDBC

JDBC(Java DataBase Connectivity)

Java数据库连接, 其实就是利用Java语言连接并访问数据库的一门技术。

## 为什么要学习JDBC

之前我们可以通过CMD、Navicat等连接并访问数据库

但是，在企业开发中，我们更多的是通过Java程序连接并访问数据库。

使用Java程序连接数据库就需要使用JDBC

包括Mybatis框架、Hibernate框架底层都是JDBC，都是对JDBC做了封装，目的是为了简化JDBC代码。

## 如何使用JDBC访问数据库

1.提出需求:

创建一个 jt\_db 数据库，在库中创建一个Account表，并插入三条记录，然后利用Java程序查询出Account表中所有的记录，并打印在控制台上.

2.开发步骤:

(1)准备数据, 创建jt\_db库, 创建account表

drop database if exists jt\_db;

create database jt\_db charset utf8;

use jt\_db;

create table account(

id int primary key auto\_increment,

name varchar(50),

money double

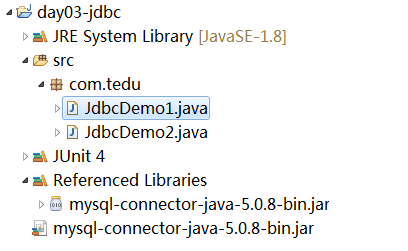
);

insert into account values(null, 'tom', 1000);

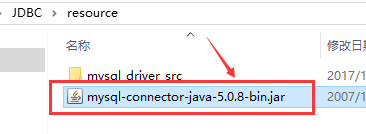
insert into account values(null, 'andy', 1000);

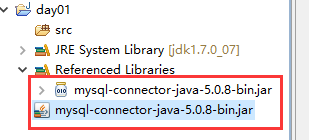
insert into account values(null, 'tony', 1000);

(2)创建JAVA工程

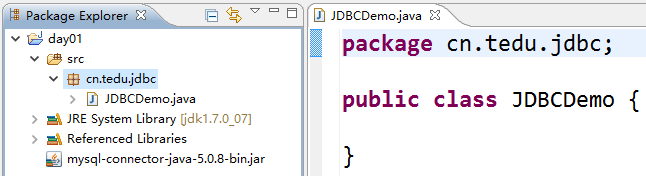


(3)导入jar包





(4)创建类并实现JDBC程序(六个步骤)



代码实现:

**public** **void** testJdbc() **throws** Exception {

//1.注册数据库驱动

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

//2.获取数据库连接

Connection conn = DriverManager

.*getConnection*(

"jdbc:mysql://localhost:3306/jt\_db",

"root","root");

//3.获取传输器

Statement stat = conn.createStatement();

//4.利用传输器执行sql语句,返回执行结果

String sql = "select \* from account";

ResultSet rs = stat.executeQuery( sql );

//5.处理执行结果

**while**(rs.next()) {

**int** id = rs.getInt("id");

String name = rs.getString("name");

**double** money = rs.getDouble("money");

System.***out***.println(

id+":"+name+":"+money

);

}

//6.释放资源

rs.close();

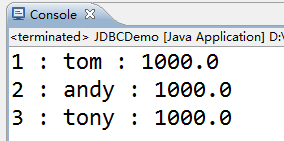
stat.close();

conn.close();

System.***out***.println("JdbcDemo1.testJdbc()");

}

3.执行结果:



## JDBC开发细节

**1、注册数据库驱动**

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

所谓的注册驱动，就是让JDBC程序加载mysql驱动程序，并管理驱动

驱动程序实现了JDBC API定义的接口以及和数据库服务器交互的功能，加载驱动是为了方便使用这些功能。

**2、获取连接之数据库URL**

Connection conn = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/jt\_db",

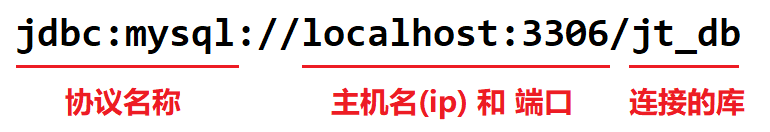
"root", "root"

);

DriverManager.getConnection() 用于获取数据连接, 返回的Connection连接对象是JDBC程序连接数据库至关重要的一个对象。

参数2和参数3分别是所连接数据库的用户名和密码。

参数1 "jdbc:mysql://localhost:3306/jt\_db" 是连接数据库的URL，用于指定访问哪一个位置上的数据库服务器及服务器中的哪一个数据库，其写法为:



当连接本地数据库，并且端口为3306，可以简写为如下形式：

jdbc:mysql:///jt\_db

**3、Statement传输器对象**

Statement stat = conn.createStatement();

Statement传输器对象用于向数据库服务器发送sql语句，该对象上提供了发送sql的方法：

executeQuery(String sql) – 用于向数据库发送查询类型的sql语句，返回一个ResultSet对象中

executeUpdate(String sql) – 用于向数据库发送更新(增加、删除、修改)类型的sql语句，返回一个int值，表示影响的记录行数

4、ResultSet结果集对象

ResultSet对象用于封装sql语句查询的结果，也是一个非常重要的对象。该对象上提供了遍历数据及获取数据的方法。

(1)遍历数据行的方法

next() – 使指向数据行的索引向下移动一行

(2)获取数据的方法

getInt(int columnIndex)

getInt(String columnLable)

getString(int columnIndex)

getString(String columnLable)

getDouble(int columnIndex)

getDouble(String columnLable)

getObject(int columnIndex)

getObject(String columnLable)

...

5、释放资源

rs.close();

stat.close();

conn.close();

此处释放资源必须按照一定的顺序释放，越晚获取的越先关闭。所以先关闭 rs对象，再关闭stat对象，最后关闭conn对象。

另，为了避免上面的程序抛出异常，释放资源的代码不会执行，应该把释放资源的代码放在finally块中.

try{

...

}catch(Exception e){

...

}finally{

if (rs != null) {

try {

rs.close();

} catch (SQLException e) {

e.printStackTrace();

} finally {

rs = null;

}

}

if (stat != null) {

try {

stat.close();

} catch (SQLException e) {

e.printStackTrace();

} finally {

stat = null;

}

}

if (conn != null) {

try {

conn.close();

} catch (SQLException e) {

e.printStackTrace();

} finally {

conn = null;

}

}

}

# JDBC增删改查

完成JDBC对数据库中数据的增删改查操作!!

**1、(自己完成)增加：往account表中插入一条新的记录，name为 "john"，money为30000**

/\*\* 1.新增: 往account表中插入一条新的记录，

\* name为 "john"，money为30000 \*/

@Test

**public** **void** testAdd() {

Connection conn = **null**;

Statement stat = **null**;

ResultSet rs = **null**;

**try** {

//注册驱动

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

//获取连接

conn = DriverManager.*getConnection*(

"jdbc:mysql:///jt\_db",

"root", "root");

//获取传输器

stat = conn.createStatement();

//执行sql语句,返回执行结果

String sql = "insert into account "

+ "values(null, 'john', 30000)";

**int** rows = stat.executeUpdate(sql);

//处理结果

System.***out***.println("影响的行数:"+rows);

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

//释放资源

JDBCUtil.*close*(conn, stat, rs);

}

}

**2、(自己完成)修改：修改account表中name为 "john" 的记录，将金额改为2500**

/\*\* 2.修改: 修改account表中name为 "john" 的

\* 记录，将金额改为2500 \*/

@Test

**public** **void** testUpdate() { }

**3、(自己完成)删除：删除account表中name为 "john" 的记录**

/\*\* 3.删除: 删除account表中name为 "john" 的

\* 记录 \*/

@Test

**public** **void** testDelete() { }

**4.查询：查询account表中id为1的记录**

/\*\* 4.查询: 查询account表中id为1的记录 \*/

@Test

**public** **void** testFindById() {

Connection conn = **null**;

Statement stat = **null**;

ResultSet rs = **null**;

**try** {

//注册驱动

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

//获取连接

conn = DriverManager.*getConnection*(

"jdbc:mysql:///jt\_db",

"root", "root");

//获取传输器

stat = conn.createStatement();

//执行sql语句,返回执行结果

String sql = "select \* from "

+ "account where id=1";

rs = stat.executeQuery(sql);

**if**(rs.next()) {

**int** id = rs.getInt("id");

String name = rs.getString("name");

**double** money = rs.getDouble("money");

System.***out***.println(

id+":"+name+":"+money

);

}

//处理结果

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

//释放资源

JDBCUtil.*close*(conn, stat, rs);

}

}

# PreparedStatement对象

在上面的增删改查的操作中，使用的是Statement传输器对象，而在开发中我们用的更多的传输器对象是PreparedStatement对，PreparedStatement是Statement的子接口，比Statement更加安全，并且能够提高程序执行的效率。

## 模拟用户登陆案例

(1)准备数据

use jt\_db;

create table user(

id int primary key auto\_increment,

username varchar(50),

password varchar(50)

);

insert into user values(null,'张三','123');

insert into user values(null,'李四','234');

(2)创建LoginUser 类，提供 main 方法 和 login 方法。

**public** **static** **void** main(String[] args) {

Scanner sc = **new** Scanner(System.***in***);

//1.提示用户登录

System.***out***.println("请登录:");

//2.提示用户输入用户名并接收用户名

System.***out***.println("请输入用户名:");

String username = sc.nextLine();

//3.提示用户输入密码并接收密码

System.***out***.println("请输入密码:");

String password = sc.nextLine();

//4.调用login方法,并传入用户名和密码

*loginByPS*(username, password);

}

/\*\*

\* 根据用户名和密码进行登录

\* **@param** username

\* **@param** password

\*/

**private** **static** **void** login(

String username, String password) {

Connection conn = **null**;

Statement stat = **null**;

ResultSet rs = **null**;

**try** {

//1.注册驱动并获取连接

conn = JDBCUtil.*getConn*();

//2.获取传输器

stat = conn.createStatement();

//3.执行SQL并返回执行结果

String sql = "select \* from user "

+ "where username='"+username+"' and "

+ "password='"+password+"'";

System.***out***.println(sql);

rs = stat.executeQuery(sql);

//4.处理结果

**if**(rs.next()) {//true:用户名密码正确

System.***out***.println("恭喜您登录成功!");

}**else** {

System.***out***.println("用户名或密码错误!");

}

} **catch** (Exception e) {

e.printStackTrace();

System.***out***.println("登录失败!!");

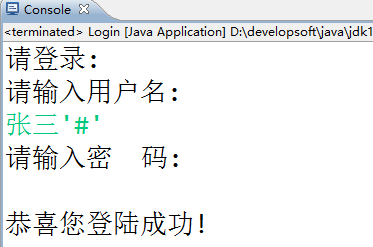
} **finally** {

JDBCUtil.*close*(conn, stat, rs);

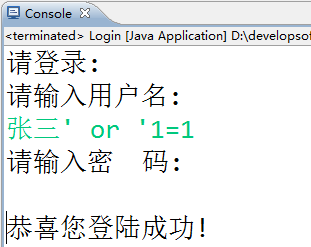
}

}

**执行时，输入：**



**或输入**



## SQL注入攻击

通过上面的案例，我们发现在执行时，不输入密码只输入用户名也可以登陆成功。这就是SQL注入攻击。

SQL注入攻击：由于后台的SQL语句是拼接而来的。其中的参数是由用户提交的，如果用户在提交参数时，在其中掺杂了一些SQL关键字或者特殊符号，就可能会导致SQL语句的语意发生变化。从而执行一些意外的操作。

## 防止SQL注入攻击

如果防止SQL注入攻击？ 使用PreparedStatement对象来替代Statement对象。

添加loginByPreparedSatement方法，在方法中，使用PreparedStatement来代替Statement作为传输器对象使用！

/\*\*

\* 根据用户名和密码进行登录

\* **@param** username

\* **@param** password

\*/

**private** **static** **void** loginByPS(

String username, String password) {

Connection conn = **null**;

PreparedStatement ps = **null**;

ResultSet rs = **null**;

**try** {

//1.注册驱动并获取连接

conn = JDBCUtil.*getConn*();

//2.获取传输器

//>>获取传输器并传递SQL骨架

String sql = "select \* from user"

+ " where username=? and"

+ " password=?";

ps = conn.prepareStatement(sql);

//>>设置SQL参数值

ps.setString(1, username);

ps.setString(2, password);

//3.执行SQL并返回执行结果

rs = ps.executeQuery();

//4.处理结果

**if**(rs.next()) {//true:用户名密码正确

System.***out***.println("恭喜您登录成功!");

}**else** {

System.***out***.println("用户名或密码错误!");

}

} **catch** (Exception e) {

e.printStackTrace();

System.***out***.println("登录失败!!");

} **finally** {

JDBCUtil.*close*(conn, ps, rs);

}

}

再次执行程序，按照上面的操作登录。此时，已经成功的防止了SQL注入攻击问题了。

使用PreparedStatement对象可以防止SQL注入攻击

而且通过方法设置参数更加的方便且不易出错!

还可以从某些方面提高程序执行的效率!

# 数据库连接池

## 什么是连接池

池: 就是一个容器(比如, 数组, 集合), 其中可以存放资源

连接池: 就是将连接存放在容器中, 目的是为了让整个程序共享连接, 实现连接的复用, 减少连接创建和关闭的次数!!

## 为什么要使用数据库连接池

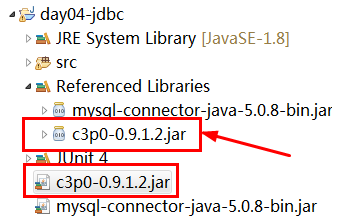
在传统方式中, 不使用连接池, 每次访问数据库, 都需要创建连接对象和关闭连接对象, 而创建连接和关闭连接是非常耗资源和耗时间的操作. 而且连接没有复用, 因此效率低下!

所以,我们可以再程序启动时,创建一批连接放在一个容器(连接池)中, 当用户需要连接时, 就从容器中获取一个连接, 当用户用完连接后, 再把连接还回池中, 这样用来用去都是池中的这批连接, 可以实现连接的复用, 减少连接开关的次数, 提高程序执行的效率!!

## 如何使用C3P0连接池

使用C3P0连接池开发步骤：

1、导入开发包



2、在程序中,创建数据库连接池对象(池子/容器)



3、设置数据库连接的基本信息

(1)方式一：(不推荐)

//设置连接数据库的基本信息

pool.setDriverClass(

"com.mysql.jdbc.Driver");

pool.setJdbcUrl(

"jdbc:mysql:///jt\_db?characterEncoding=utf-8");

pool.setUser("root");

pool.setPassword("root");

(2)方式二：(推荐)

在src源码目录下提供一个名称为c3p0-config.xml的文件, 文件内容配置如下：

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<c3p0-config>

<default-config>

<property name=*"driverClass"*>

com.mysql.jdbc.Driver

</property>

<property name=*"jdbcUrl"*>

jdbc:mysql:///jt\_db?characterEncoding=utf-8

</property>

<property name=*"user"*>

root

</property>

<property name=*"password"*>

root

</property>

</default-config>

</c3p0-config>

Xml文件对语法要求非常严格, 在编写时要注意语法细节:

(3)方式三：(推荐)

在src源码目录下, 提供一个名称为c3p0.properties的文件, 文件内容配置如下:

c3p0.driverClass=com.mysql.jdbc.Driver

c3p0.jdbcUrl=jdbc:mysql:///jt\_db?characterEncoding=utf-8

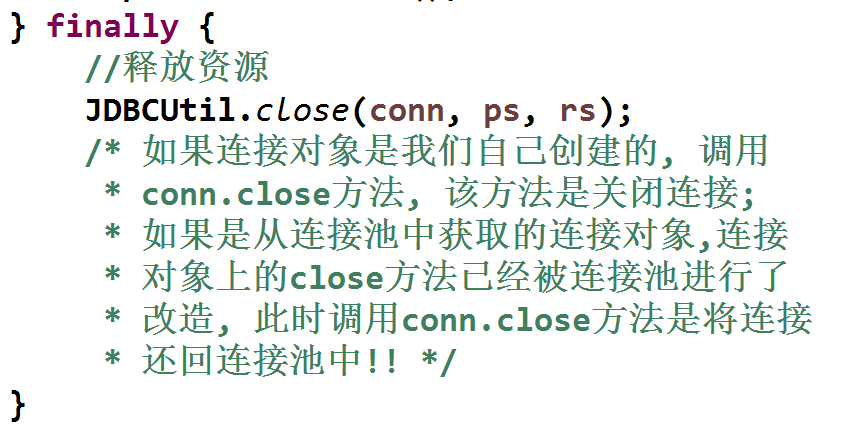
c3p0.user=root

c3p0.password=root

4、通过连接池获取一个连接对象

pool.getConnection();

5、用完连接后，将连接还回连接池中



# 事务及四大特性

## 什么是事务

事务: 将一堆SQL绑定在一起执行, 要么全部执行成功, 要么全部执行失败!!

转账: A🡪B 1000元 100元

A: update acc set money=money-100 where name=’A’;

B: update acc set money=money+100 where name=’B’;

## 事务的四大特性

1、**原子性**：事务中的所有操作都是一个整体，不可以再分割，其中的所有SQL语句，要么全都执行成功， 要么全都执行失败！！！

2、**一致性**：事务前后的业务数据总和是保持一致的

转账: A🡪B 1000元 100元

A: 1000-100=900

B: 1000+100=1100;

A: 1000;

B: 1000;

3、**隔离性**：多个事务之间是相互隔离开来的,一个事务不会访问到另外一个事务正在执行中的操作。

事务1:转账: 1000-100=900; 1000+100=1100;

事务2:查询A.B的账号金额

隔离性等级高, 可以保证事务之间完全隔离, 否则可能会出现, 事务之间并不完全隔离的情况!!

4、**持久性**: 如果事务一旦提交(事务中的SQL语句全部都执行成功了), 修改后的数据就真的被保存到了数据库中.

开启事务----

1000-100=900; 900

1000+100=1100; 1100

提交/回滚

## MySQL中的事务

在默认情况下，MySQL每执行一条SQL语句，都是一个单独的事务。如果需要在一个事务中包含多条SQL语句，那么需要开启事务和结束事务。

* 开启事务：start transaction；
* 结束事务：commit（提交事务）或rollback（回滚事务）。

在执行SQL语句之前，先执行strat transaction，这就开启了一个事务（事务的起点），然后可以去执行多条SQL语句，最后要结束事务，**commit**表示提交，即事务中的多条SQL语句所做出的影响会持久化到数据库中。或者**rollback**，表示回滚，即回滚到事务的起点，之前做的所有操作都被撤消了！

下面演示A账户给B账户转账的例子：

\*\* 准备数据：

-- 1、创建数据库jt\_db数据库(如果不存在才创建)

create database if not exists jt\_db charset utf8;

use jt\_db; -- 选择jt\_db数据库

-- 2、在 jt\_db 库中创建 acc 表(银行账户表), 要求有id(主键),name(姓名),money(账户金额)

drop table if exists acc;

create table acc(

id int primary key auto\_increment,

name varchar(50),

money double

);

-- 3、往 acc 表中, 插入2条记录

insert into acc values(null,'A',1000);

insert into acc values(null,'B',1000);

-- 查询acc表中的所有记录

select \* from acc;

下面分别演示事务开启及执行一系列SQL之后，回滚事务、提交事务及中断操作的效果。

rollback（回滚事务）：

-- 查询acc账户表中A和B的金额

select \* from acc;

-- 开启事务

start transaction;

-- 开始转账，A账户减去100元

update acc set money=money-100 where name='A';

-- 查询acc账户表中A和B的金额

select \* from acc;

-- B账户增加100元

update acc set money=money+100 where name='B';

-- 查询acc账户表中A和B的金额

select \* from acc;

-- 回滚事务

rollback;

-- 再次查询acc账户表中A和B的金额

select \* from acc;

commit（提交事务）：

-- 将上面的操作再做一次，最后将rollback替换为commit，即提交事务

commit;

中断操作

-- 将上面的操作再做一次，最后将rollback替换为quit，即中断操作

quit;

# 事务并发读问题

## 事务并发读问题

多个事务对相同的数据同时进行操作，这叫做事务并发。

在事务并发时，如果没有采取必要的隔离措施，可能会导致各种并发问题，破坏数据的完整性等。这些问题中，其中有三类是读问题，分别是：脏读、不可重复读、幻读。

(1)**脏读**（dirty read）：读到另一个事务的未提交更新数据，即读取到了脏数据；

例如：A给B转账100元但未提交事务，在B查询后，A做了回滚操作，那么B查询到了A未提交的数据，就称之为脏读。

(2)**不可重复读**（unrepeatable read）：对同一记录的两次读取不一致，因为另一事务对该记录做了修改（是针对修改操作）

例如：在事务1中，前后两次查询A账户的金额，在两次查询之间，另一事物2对A账户的金额做了修改，此种情况可能会导致事务1中，前后两次查询的结果不一致。这就是不可重复度

(3)**幻读（虚读）**（phantom read）：对同一张表的两次查询不一致，因为另一事务插入了一条记录(是针对插入或删除操作)；

注意:mysql默认的是不允许出现脏读和不可重复读，所以在下面演示之前需要设置mysql允许出现脏读、不可重复读等。

set tx\_isolation='read-uncommitted';

脏读示例：

-- 在窗口1中，开启事务，执行A给B转账100元

set tx\_isolation='read-uncommitted'; -- 允许脏读、不可重复读、幻读

use jt\_db; -- 选择jt\_db库

start transaction; -- 开启事务

update acc set money=money-100 where name='A';

update acc set money=money+100 where name='B';

-- 在窗口2中，开启事务，查询B的账户金额

set tx\_isolation='read-uncommitted'; -- 允许脏读、不可重复读、幻读

use jt\_db; -- 选择jt\_db库

start transaction; -- 开启事务

select \* from acc where name='B'; -- 出现脏数据

-- 切换到窗口1，回滚事务，撤销转账操作。

rollback; -- 回滚事务

-- 切换到窗口2，查询B的账户金额

select \* from acc where name='B';

在窗口2中，B看到自己的账户增加了100元（此时的数据A操作事务并未提交），此种情况称之为"脏读"。

不可重复读示例：

-- 在窗口1中，开启事务，查询A账户的金额

set tx\_isolation='read-uncommitted'; -- 允许脏读、不可重复读、幻读

use jt\_db; -- 选择jt\_db库

start transaction; -- 开启事务

select \* from acc where name='A';

-- 在窗口2中，开启事务，查询A的账户金额减100

set tx\_isolation='read-uncommitted'; -- 允许脏读、不可重复读、幻读

use jt\_db; -- 选择jt\_db库

start transaction; -- 开启事务

update acc set money=money-100 where name='A'; -- A账户减去100

select \* from acc where name='A';

commit; -- 提交事务

-- 切换到窗口1，再次查询A账户的金额。

select \* from acc where name='A'; -- 前后查询结果不一致

在窗口1中，前后两次对同一数据(账户A的金额)查询结果不一致，是因为在两次查询之间，另一事务对A账户的金额做了修改。此种情况就是"不可以重复读"

幻读示例：

-- 在窗口1中，开启事务，查询账户表中是否存在id=3的账户

set tx\_isolation='read-uncommitted'; -- 允许脏读、不可重复读、幻读

use jt\_db; -- 选择jt\_db库

start transaction; -- 开启事务

select \* from acc where id=3;

-- 在窗口2中，开启事务，往账户表中插入了一条id为3记录，并提交事务。

-- 设置mysql允许出现脏读、不可重复度、幻读

set tx\_isolation='read-uncommitted';

use jt\_db; -- 选择jt\_db库

start transaction; -- 开启事务

insert into acc values(3, 'C', 1000);

commit; -- 提交事务

-- 切换到窗口1，由于上面窗口1中查询到没有id为3的记录，所以可以插入id为3的记录。

insert into acc values(3, 'C', 1000); -- 插入会失败!

在窗口1中，查询了不存在id为3的记录，所以接下来要执行插入id为3的记录，但是还未执行插入时，另一事务中插入了id为3的记录并提交了事务，所以接下来窗口1中执行插入操作会失败。

探究原因，发现账户表中又有了id为3的记录（感觉像是出现了幻觉）。这种情况称之为"幻读"

以上就是在事务并发时常见的三种并发读问题，那么如何防止这些问题的产生？

可以通过设置事务隔离级别进行预防。

## 事务隔离级别

事务隔离级别分四个等级，在相同数据环境下，对数据执行相同的操作，设置不同的隔离级别，可能导致不同的结果。不同事务隔离级别能够解决的数据并发问题的能力也是不同的。

set tx\_isolation='read-uncommitted';

1、**READ UNCOMMITTED**（读未提交数据）

安全级别最低, 可能出现任何事务并发问题(比如脏读、不可以重复读、幻读等)

性能最好（不使用!!）

2、**READ COMMITTED**（读已提交数据）（Oracle默认）

防止**脏读**，没有处理不可重复读，也没有处理幻读；

性能比REPEATABLE READ好

3、**REPEATABLE READ**（可重复读）（MySQL默认）

防止**脏读**和**不可重复读**，不能处理幻读问题；

性能比SERIALIZABLE好

4、**SERIALIZABLE**（串行化）

不会出现任何并发问题，因为它是对同一数据的访问是串行的，非并发访问的；

性能最差；

MySQL的默认隔离级别为REPEATABLE READ，即可以防止脏读和不可重复读

## 设置隔离级别

0、MySQL查询当前的事务隔离级别

select @@tx\_isolation;

**1、MySQL设置事务隔离级别（了解）**

(1) set tx\_isolation='read-uncommitted';

安全性最差，容易出现**脏读**、**不可重复读**、**幻觉读**，但性能最高

(2) set tx\_isolation='read-committed';

安全性一般，可防止**脏读**，但容易出现**不可重复读**、**幻觉读**

(3) set tx\_isolation='repeatable-read';

安全性较好，可防止**脏读**、**不可重复读**，但是容易出现**幻读**

(4) set tx\_isolation='serialiable';

安全性最好，可以防止一切事务并发问题，但是性能最差。

**2、JDBC设置事务隔离界别**

JDBC中通过Connection提供的方法设置事务隔离级别：

Connection.setTransactionIsolation(int level)

参数可选值如下：

Connection.TRANSACTION\_READ\_UNCOMMITTED 1（读未提交数据）

Connection.TRANSACTION\_READ\_COMMITTED 2（读已提交数据）

Connection.TRANSACTION\_REPEATABLE\_READ 4（可重复读）

Connection.TRANSACTION\_SERIALIZABLE 8（串行化）

Connection.TRANSACTION\_NONE 0（不使用事务）

提示：在开发中，一般情况下不需要修改事务隔离级别

**3、JDBC中实现转账例子**

提示：JDBC中默认是自动提交事务，所以需要关闭自动提交，改为手动提交事务

也就是说, 关闭了自动提交后, 事务就自动开启, 但是执行完后需要手动提交或者回滚!!

(1)执行下面的程序，程序执行没有异常，转账成功！A账户减去100元，B账户增加100元。

(2)将第4步、5步中间的代码放开，再次执行程序，在转账过程中抛异常，转账失败！由于事务回滚，所以A和B账户金额不变。

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

Connection conn = **null**;

Statement stat = **null**;

ResultSet rs = **null**;

**try** {

//1.获取连接

conn = JDBCUtil.*getConn*();

//2.关闭JDBC自动提交事务（默认开启事务）

conn.setAutoCommit(**false**);

//3.获取传输器

stat = conn.createStatement();

/\* \*\*\*\*\* A给B转账100元 \*\*\*\*\* \*/

//4.A账户减去100元

String sql = "update acc set money=money-100 where name='A'";

stat.executeUpdate(sql);

//int i = 1/0; // 让程序抛出异常，中断转账操作

//5.B账户加上100元

sql = "update acc set money=money+100 where name='B'";

stat.executeUpdate(sql);

//6.手动提交事务

conn.commit();

System.***out***.println("转账成功！提交事务...");

} **catch** (Exception e) {

e.printStackTrace();

//一旦其中一个操作出错都将回滚，使两个操作都不成功

conn.rollback();

System.***out***.println("执行失败！回滚事务...");

} **finally**{

JDBCUtil.*close*(conn, stat, rs);

}

}