一. 定义

所谓事务，它是一个操作序列，这些操作要么都执行，要么都不执行，它是一个不可分割的工作单位。

官方定义：事务是由一系列对系统中数据进行访问与更新的操作所组成的一个程序执行单元。

在java中：在一次事务中要使用同一个Connection，可以在提交的时候控制回滚。

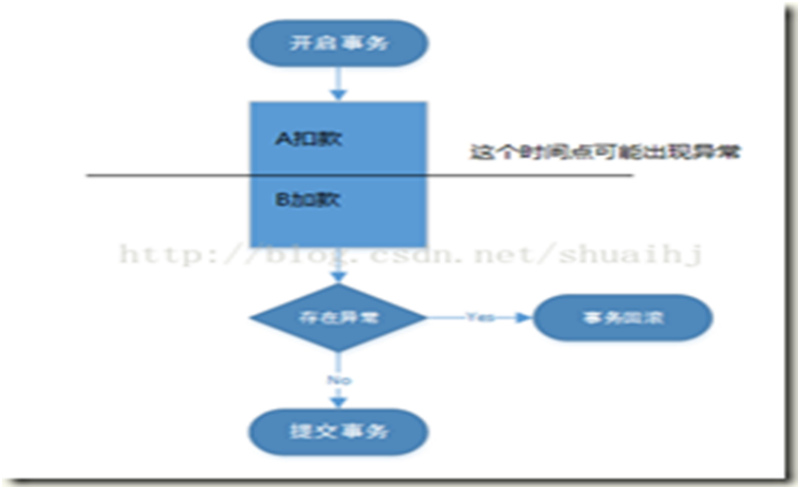
二. 数据库的ACID

- ACID，是指在可靠数据库管理系统（DBMS）中，事务(transaction)所应该具有的四个特性：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性

（Durability）.这是可靠数据库所应具备的几个特性.下面针对这几个特性进行逐个讲解.

三. 原子性

- 原子性是指事务是一个不可再分割的工作单位，事务中的操作要么都发生，要么都不发生。
- A给B转帐100元钱



四. 一致性

- 一致性是指在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。这是说数据库事务不能破坏关系数据的完整性以及业务逻辑上的一致性。

五. 隔离性

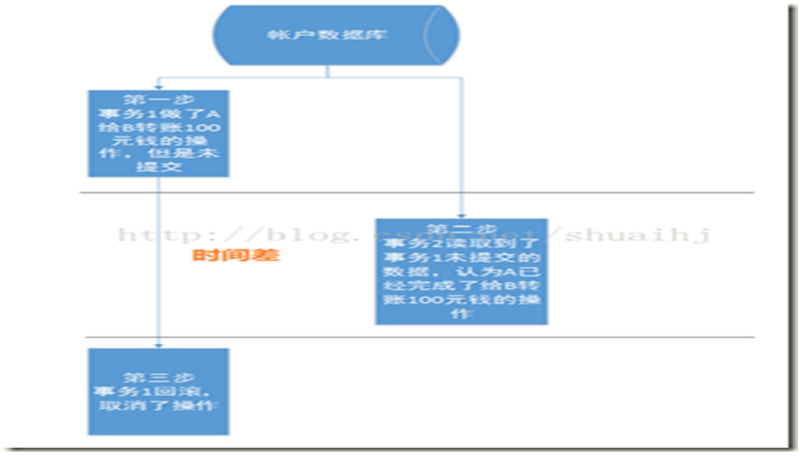
- 多个事务并发访问时，事务之间是隔离的（不是并发的），一个事务不应该影响其它事务运行效果。

六. 事务之间的相互影响（如果不考虑隔离性，也就是发生并发问题）

- 不考虑隔离性的情况下，事务之间的相互影响分为几种，分别为：脏读，不可重复读，幻读，丢失更新
- 不考虑隔离性也就相当于并发事务产生的问题。

6.1.脏读

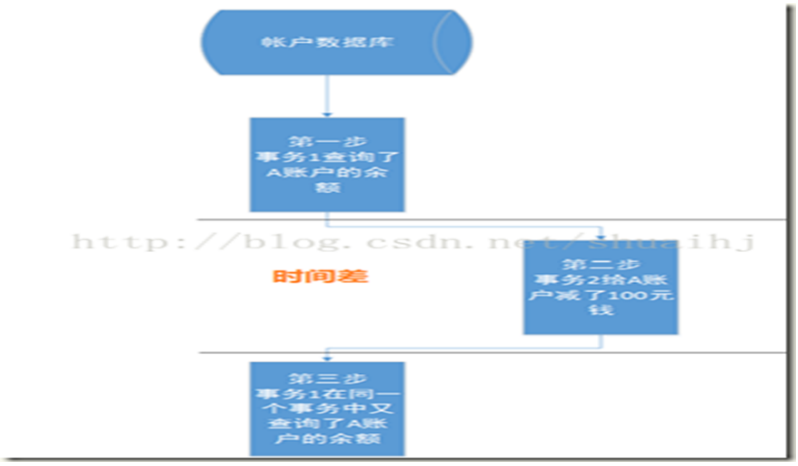
- 脏读意味着事务2读取了事务1中未提交的数据，而这个数据在事务1中是有可能回滚的；如下案例，此时如果事务1回滚，则B账户必将有损失。
  事务2执行时发生脏读，事务1执行完导致账户B有损失



6.2不可重复读

- 不可重复读意味着：在数据库访问中，一个事务范围内两个相同的查询却返回了不同数据。这是由于查询时系统中其他事务修改的提交而

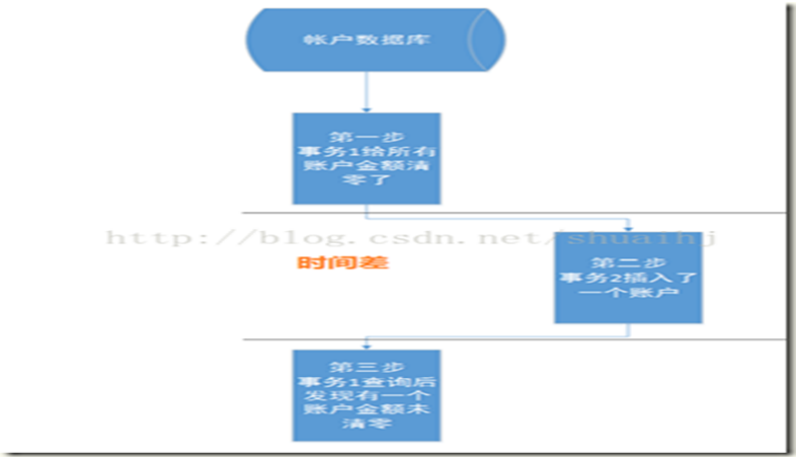
引起的。如下案例，事务1必然会变得糊涂，不知道发生了什么。



- 脏读和不可重读的区别是：脏读是一个事务读取了另一个为未完成的事务执行过程中的数据，不可重读是一个事务执行过程中，另一个事务提交并修改了当前事务正在读取的数据。

6.3幻读（虚读）

- 幻读，是指当事务不是独立执行时发生的一种现象，例如第一个事务对一个表中的数据进行了修改，这种修改涉及到表中的全部数据行。同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象发生了幻觉一样.



- 幻读和不可重复读的相同点：都是读取了另一条已经提交的事务（与脏读不同）；不同点：不可重读中两个事务查询的是同一个数据项，幻读针对的是一批整体数据（比如数据个数）。

6.4 丢失更新

- 两个事务同时读取同一条记录，A先修改记录，B也修改记录（B 是不知道A修改过），B提交数据后B的修改结果覆盖了A的修改结果。

七. 数据库的隔离级别

点击此处有博客园中的一篇相关文章

- 隔离事务之间的影响是通过锁来实现的，通过阻塞来阻止事务之间的影响。不同的隔离级别是通过加不同的锁（下表中的级别越来越高），造成阻塞来实现 的，所以会以付出性能作为代价；安全级别越高，处理效率越低； 安全级别越低，效率高。

隔离级别	脏读	丢失更新	不可重复读	幻读	并发模型	更新冲突检测
未提交读：Read Uncommited	是	是	是	是	悲观	否
已提交读：Read commited	否	是	是	是	悲观	否
可重复读：Repeatable Read	否	否	否	是	悲观	否
可串行读：Serializable	否	否	否	否	悲观	否

- **未提交读：** 在读数据时不会检查或使用任何锁。因此，在这种隔离级别中可能读取到没有提交的数据。
  在这种隔离级别中，所有事务都可以看到其他未提交事务的执行结果。
- **已提交读：**只读取提交的数据并等待其他事务释放排他锁。读数据的共享锁在读操作完成后立即释放。已提交读是SQL Server 的默认隔离级别。
  这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）。
  在这种隔离级别中，满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变。
  这种隔离级别会出现不可重读取、幻读的场景：事务1执行多次查询操作（查询同一条数据），事务2执行修改操作（被修改的数据是事务1多次查询的数据），在事务1执行到一半的时候事务2执行并提交了，接着事务1执行就可能会出现这两种情况
- **可重复读：** 像已提交读级别那样读数据，但会保持共享锁直到事务结束。（MySQL的默认隔离级别）。
  在这种隔离级别中，确保同一事物的多个实例再并发读取数据时，会看到同样的数据行。不过理论上，这会导致另一个棘手的问题：幻读（Phantom Read）。
- **可串行读：**工作方式类似于可重复读。但它不仅会锁定受影响的数据，还会锁定这个范围。这就阻止了新数据插入查询所涉及的范围。这是最高的隔离级别，它通过强制事务排序，是指不可能相互冲突，从而解决幻读问题；简而言之，他是在每个读的数据行上加上共享锁；在这个级别，可能会导致大量的超时和锁竞争

八.持久性

- 持久性，意味着在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。
- 即使出现了任何事故比如断电等，事务一旦提交，则持久化保存在数据库中。
- SQL SERVER通过write-ahead transaction log来保证持久性。write-ahead transaction log的意思是，事务中对数据库的改

变在写入到数据库之前，首先写入到事务日志中。而事务日志是

按照顺序排号的（LSN）。当数据库崩溃或者服务器断电时，重启动SQL SERVER，SQLSERVER首先会检查日志顺序号，将本应对数据库做更改而未做的部分持久化到数据库，从而保证了持久性。

九. 关于事务的示例代码

> *场景：转账*

9.1.存在问题的转账代码：

9.1.1 相关配置

在pom.xml文件中使用了依赖：

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.16.18</version>
    <scope>provided</scope>
</dependency>
```

SpringConfig 配置类（创建ApplicationContext容器就是根据这个类）

```
// @ComponentScan：com.bruce 父包，该包的子包也会被扫描
@Configuration//不加这个注解也可以
@ComponentScan(value = "com.bruce")
public class SpringConfig {}
```

db.properties文件

```
jdbc.driver=com.mysql.cj.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/test?
useSSL=false&allowPublicKeyRetrieval=true&serverTimezone=U
TC
jdbc.username=root
jdbc.password=271441
```

Acount实体类：

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Account {

    private Integer id;
    private String name;
    private Double balance;
}
```

PropertiesConfig配置类：

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@PropertySource(value =
"classpath:db.properties",ignoreResourceNotFound =
true,encoding = "UTF-8")
@Configuration
public class PropertiesConfig {

    @Value("${jdbc.driver}")
    private String driver;

    @Value("${jdbc.url}")
    private String url;

    @Value("${jdbc.username}")
    private String username;

    @Value("${jdbc.password}")
    private String password;
}
```

9.1.2 AccountDaoImpl类：（只有更新数据库的代码）

```
@Repository(value = "accountDao")
@Scope("singleton")  //默认值
public class AccountDaoImpl implements AccountDao {

    //注入配置类PropertiesConfig获得数据库连接信息
    @Autowired
    PropertiesConfig propertiesConfig;
    /**
    更新表操作；
    每次调用这个方法都会创建一个Connection，这也是导致转账操作时，
导致的原子性问题的根源；
    存在两个问题：
      1.两次更新数据库操作使用的是两个Connection
      2.JDBC底层会自动提交事务，导致每一次更新数据库操作都会持久化
到数据库，来不及执行回滚
    */
    public int updateBalance(String name, Double balance)
{
        int count=0;
        Connection conn=null;
        PreparedStatement ps=null;
        try {
            Class.forName(propertiesConfig.getDriver());
```

```
            conn =
DriverManager.getConnection(propertiesConfig.getUrl(),
propertiesConfig.getUsername(),
propertiesConfig.getPassword());
            //关闭自动提交
            conn.setAutoCommit(false);

            ps = conn.prepareStatement("update account set
balance=balance+? where name=?");
            ps.setObject(1,balance);
            ps.setObject(2,name);
            count= ps.executeUpdate();

            //即使这里使用手动提交也不会成功，因为最后的结果还是：
用户每调用一次updateBalance方法都会提交一次数据，无法在两次更新数据
库之间回滚
            //手动提交
            conn.commit();
        } catch (Exception e) {
            try {
                //手动回滚
                conn.rollback();
            } catch (SQLException ex) {
                ex.printStackTrace();
            }
            e.printStackTrace();
        } finally {
            try {
                if(ps!=null){
                    ps.close();
                }
                if(conn!=null){
                    conn.close();
                }
            } catch (SQLException e) {
                e.printStackTrace();
            } finally {
            }
        }
        return count;
    }
```

9.1.3 AccountServiceImpl类：

```
@Service(value = "accountService")
public class AccountServiceImpl implements AccountService
{

    @Autowired
    AccountDao accountDao;
```

```
    /**
     * 转账
     * @param fromName
     * @param toName
     * @param money
     * @return
     */
    public int transferMoney(String fromName, String
toName, Double money) {

        int count1 = accountDao.updateBalance(fromName,
money * (-1));
        System.out.println(count1>0?fromName+"钱减
少":fromName+"转账失败");

        System.out.println(1000/0); //模拟异常

        int count2 = accountDao.updateBalance(toName,
money);
        System.out.println(count1>0?toName+"钱增
加":fromName+"收款失败");

        return count1+count2;
    }
```

9.1.3 分析

这样在转账的时候会出现问题，原因是每次调用updateBalance方法的时候都会创建一个Connection。
在测试代码的转账操作中，每次在这两个Connection上对数据库的操作都会被提交，而事务要求的是两次对数据库的操作都是在一个Connection上进行操作，如果两次操作都成功，则提交，如果两次操作中某次出现了错误，就会回滚。

```
问题的根源：
  1.两次更新数据库操作使用的是两个Connection
  2.JDBC底层会自动提交事务，导致每一次更新数据库操作都会持久化到数据
库，来不及执行回滚
```

9.2.使用事务的转账代码：

修改9.1中的代码

增加两个工具类：ConnectionUtils和TransactionMamager，修改AccountDaoImpl、AccountServiceImpl类

9.2.1 ConnectionUtils类：

```
/**
把9.1.2节中回去数据库连接的内容提取出来作为一个工具类
*/
```

```
@Component
public class ConnectionUtils {

    @Autowired
    PropertiesConfig propertiesConfig;

    //本地局部变量，线程局部变量，一个线程一个
    ThreadLocal<Connection> tl=new ThreadLocal<Connection>
();

    /**
     * 获取Connection
     * @return
     */
    public Connection getCurrentConnection(){
        Connection connection = tl.get();
        try {
            if(connection==null){

 Class.forName(propertiesConfig.getDriver());
                connection =
DriverManager.getConnection(propertiesConfig.getUrl(),
propertiesConfig.getUsername(),
propertiesConfig.getPassword());
                tl.set(connection);
            }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
        }
        return connection;
    }

    /**
     本意是关闭Connection，但是这个方法啥也没干
     @return
     */
    public void realseCurrentConnection(){
        Connection connection = tl.get();
        try {
            if(connection!=null){
                //connection.close();
            }
            //tl.remove();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
        }
    }
}
```

9.1.2 AccountDaoImpl类（修）：

```
@Repository(value = "accountDao")
@Scope("singleton")  //默认值
public class AccountDaoImpl implements AccountDao {

    @Autowired
    ConnectionUtils connectionUtils;
    /**
    更新表操作；在这个方法内部不会再创建Connection连接，而是把创建Connection连接的操作提取到ConnectionUtils类中，这个方法中只需要调用ConnectionUtils类中的方法就可以连接数据库

    */
    public int updateBalance(String name, Double balance)
{
        PreparedStatement ps=null;
        try {
            Connection conn =
connectionUtils.getCurrentConnection();
            ps = conn.prepareStatement("update account set
balance=balance+? where name=?");
            ps.setObject(1,balance);
            ps.setObject(2,name);
            return ps.executeUpdate();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            //这段代码可以没有，这个方法内只是通过Connection操作数据库，并不是创建和管理Connection。

 connectionUtils.realseCurrentConnection();
        }
        return 0;
    }
```

9.1.3 TransactionManager类：

```
/**
事务管理类，这个类中封装了ConnectionUtils类。
这个类对AccountDaoImpl类的单独使用不会有影响；如果想要开启事务，就使用这个类调用openTransaction方法，创建Connection的实例并设置自动提交为手动，并且在开启事务之后可以管理这个Connection，如果不想开启事务，可以直接使用AccounntDaoImpl类来创建Connection。
*/
@Component
public class TransactionManager {

    @Autowired
```

```
    ConnectionUtils connectionUtils;

    /**
     01-开启事务，也就是获得Connection，并关闭自动提交
     */
    public void openTransaction(){
        try {
            Connection currentConnection =
connectionUtils.getCurrentConnection();
            //关闭自动提交
            currentConnection.setAutoCommit(false);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    /**
     * 02-提交事务
     */
    public void commitTransaction(){
        try {
            Connection currentConnection =
connectionUtils.getCurrentConnection();
            //提交
            currentConnection.commit();
            //关闭Connection
            currentConnection.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }


    /**
     * 03-回滚事务
     */
    public void rollbackTransaction(){
        try {
            Connection currentConnection =
connectionUtils.getCurrentConnection();
            //回滚事务
            currentConnection.rollback();
            //关闭Connection
            currentConnection.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

9.1.4 AccountServiceImpl类（修）：

```java
@Service(value = "accountService")
public class AccountServiceImpl implements AccountService
{

    @Autowired
    TransactionManager tx;

    @Autowired
    AccountDao accountDao;
    public int transferMoney(String fromName, String
toName, Double money) {
        int count1 = 0;
        int count2 = 0;
        try {
            //开启事务
            tx.openTransaction();

            count1 = accountDao.updateBalance(fromName,
money * (-1));
            System.out.println(count1>0?fromName+"钱减
少":fromName+"转账失败");

            System.out.println(1000/0); //模拟异常

            count2 = accountDao.updateBalance(toName,
money);
            System.out.println(count1>0?toName+"钱增
加":fromName+"收款失败");
            //提交事务
            tx.commitTransaction();
        } catch (Exception e) {
            //回滚事务
            tx.rollbackTransaction();
            e.printStackTrace();
        }
        return count1+count2;
    }
```