Spring事务

事务的特性：
原子性
一致性
隔离性
持久性
原子性：一组操作是不可分割的最小单元操作
一致性Consistent：事务处理前后数据的完整性必须保持一致。
隔离性：多个事务之间不应该互相影响 多个用户并发访问数据库时，一个用户的事务不能被其他用户的事务所干扰，多个并发事务之间数据要相互隔离。
持久性：一旦一个事务执行完之后，就永远持久化，执行完之后的数据不再受影响 事务处理前后数据的完整性必须保持一致。

Spring中的事务管理
在Spring中事务管理中，提供了一组接口
spring事务管理的3个接口
PlatformTransactionManager平台事务管理器 包含了提交，回滚
TransactionDefine事务定义接口 定义隔离级别 传播行为，是否超时，是否只读等
TransactionStatus事务状态接口 具体的运行的状态

Spring为不同的持久层框架提供了不同的PlatformTransactionManager接口实现

事务	说明
org.springframework.jdbc.datasource.DataSourceTransactionManager	使用Spring JDBC或iBatis 进行持久化数据时使用
org.springframework.orm.hibernate3.HibernateTransactionManager	使用Hibernate3.0版本进行持久化数据时使用
org.springframework.orm.jpa.JpaTransactionManager	使用JPA进行持久化时使用
org.springframework.jdo.JdoTransactionManager	当持久化机制是Jdo时使用
org.springframework.transaction.jta.JtaTransactionManager	使用一个JTA实现来管理事务，在一个事务跨越多个资源时必须使用

下面是另外一个接口

事务的隔离级别 四种
TransactionDefine
如果不考虑隔离性，会引发如下的安全问题：
1.脏读：一个事务读取了另一个事务改写但还未提交的数据，如果这些数据被回滚，则读到的数据是无效的。
2.不可重复读：在同一个事务中，多次读取同一数据返回的结果有所不同。
3.幻读（虚读）：一个事务读取了几行记录后，另一个事务插入一些记录，幻读就发生了。再后来的查询中，第一个事务就会发现有些原来没有的记录。

隔离级别	含义
DEFAULT	使用后端数据库默认的隔离级别(spring中的的选择项)
READ_UNCOMMITED	允许你读取还未提交的改变了的数据。可能导致脏、幻、不可重复读
READ_COMMITTED	允许在并发事务已经提交后读取。可防止脏读，但幻读和 不可重复读仍可发生
REPEATABLE_READ	对相同字段的多次读取是一致的，除非数据被事务本身改变。可防止脏、不可重复读，但幻读仍可能发生。
SERIALIZABLE	完全服从ACID的隔离级别，确保不发生脏、幻、不可重复读。这在所有的隔离级别中是最慢的，它是典型的通过完全锁定在事务中涉及的数据表来完成的。

除了如图的数据库提供的事务隔离级别，spring提供了Default隔离级别，该级别表示spring使用后端数据库默认的隔离级别。

MySQL默认事务隔离级别：REPATABLE_READ(可能出现幻读)
Oracle默认：READ_COMMITTED(可能出现不可重复读和幻读)

事务的传播行为：
1.保证事务在一个事务里
2.保证事务不在一个事务里
3.事务的嵌套

事务的传播行为：主要解决的是业务层方法之间的相互调用行为

事务传播行为类型	说明
PROPAGATION_REQUIRED	支持当前事务，如果不存在 就新建一个
PROPAGATION_SUPPORTS	支持当前事务，如果不存在，就不使用事务
PROPAGATION_MANDATORY	支持当前事务，如果不存在，抛出异常
PROPAGATION_REQUIRES_NEW	如果有事务存在，挂起当前事务，创建一个新的事务
PROPAGATION_NOT_SUPPORTED	以非事务方式运行，如果有事务存在，挂起当前事务
PROPAGATION_NEVER	以非事务方式运行，如果有事务存在，抛出异常
PROPAGATION_NESTED	如果当前事务存在，则嵌套事务执行

这个1-3是一组
就是如果第一个service有事务的话，就使用事务，如果没有事务的话，就按不同的来
然后 4-6
如果service 1 service2不在一个事务里面的话，如果service1有的话，1就新建一个，2就挂起，3就报异常
然后最后一个
是一种嵌套的事务 可以自己控制 就是service1执行完之后，设置保存点，然后如果service2发生异常之后，我们可以让service2回滚到保存点的位置，或者最初始的状态，

TransactionStatus接口介绍
这个是一种事务的一种状态
提供了一组方法，可以获得到事务的状态，是否有保存点，是否完成，是不是一个新的事务。。。

Spring 支持俩种方式事务管理

编程式的事务管理，
在实际应用中应用很少
通过TransactionTemplate手动事务管理
使用XML配置声明式的事务管理
开发中推荐使用
Spring的声明式事务是通过AOP实现的

下面是配置一个案例的配置准备方法：

```
<!-- 引入外部的属性文件 -->
<context:property-placeholder location="jdbc.properties"/>
<!--配置c3p0的连接池-->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="driverClass" value="${jdbc.driverClass}"></property>
    <property name="jdbcUrl" value="${jdbc.url}"></property>
    <property name="user" value="${jdbc.username}"></property>
    <property name="password" value="${jdbc.password}"></property>
```

其中有C3P0的连接池，是负责加载连接的地方可能
C3P0是一个开源的JDBC连接池，它实现了数据源和JNDI绑定，支持JDBC3规范和JDBC2的标准扩展。 目前使用它的开源项目有Hibernate，Spring等。

```
<!--配置业务层类-->
<bean  id="acoutService" class="cn.moke.spring.demo1.AccoutServiceImpl">
</bean>
<!--配置DAO的类-->
<bean id="acoutDAO" class="cn.moke.spring.demo1.AcoutDAOImpl">
<!--注入连接池-->
<property name="dataSource" ref="dataSource"></property>
</bean>
```

这是配置业务层，DAO层的Bean,由于我们在DAO层继承了一个连接池的Bean。至于为什么：
在下面这个JDBC的源码中，我们可以看到，这里面设置了一个方法
关于连接池的方法

```
public final void setDataSource(DataSource dataSource) {
if(this.jdbcTemplate == null || dataSource != this.jdbcTemplate.getDataSource()) {
this.jdbcTemplate = this.createJdbcTemplate(dataSource);
        this.initTemplateConfig();
}
```

```
}
```

所以我们只需要在xml中，进行配置，就自动创建模板了

```
this.jdbcTemplate = this.createJdbcTemplate(dataSource);
```

然后我们实现就行了，实现的代码如下

```
@Override
public void outMoney(String out, Double money) {
 String sql="update account set money =money - ? where name =?";
     this.getJdbcTemplate().update(sql,money,out);
}
/*@param in: 转入的账号
     @param out: 转入的金额
    * */
@Override
public void inMoney(String in, Double money) {
 String sql="update account set money = money + ?  where name = ?";
    this.getJdbcTemplate().update(sql,money,in);
}
```

```
@Override
public void transfer(String out, String in, double money) {
accoutDAO.outMoney(out,money);
accoutDAO.inMoney(in,money);

}
```

但是这样就导致了一个问题，就是下面我们涉及到的，事务失败的问题

1 编程式的事务管理
在ActiontService中使用TransactionTemplate
TransactionTemplate依赖DataSourceTransactionManager
DataSourceTransactionManager依赖DataSource

事务管理器，需要引用连接池获取连接connection对象
是用事务管理模板来操作事务管理器，然后在需要事务管理的业务类注入事务管理模板对象，调用指定方法，把需要事务操作的代码写在它的匿名内部类的方法中即可，这就是编程式事务管理，通过代码的方式实现

为什么这是编程式的事务管理，因为是需要我们手动的编写代码来实现的

在下面的实现中，我们需要实现一个匿名内部类，同时也要注意，由于我们使用了外面的变量，需要将外面的变量设为final
这样子我们设置的俩个语句，就是属于一个事务中的了，所以说如果出错，就自动回滚，就不会再出现事务的出错。
这是XML

```
<!--配置业务层类-->
<bean  id="accoutService" class="cn.moke.spring.demo1.AccountServiceImpl">
    <property name="accoutDAO"  ref="accoutDAO"></property>
<!--注入事务管理的模板-->
<property name="transactionTemplate" ref="transactionTemplate"></property>
</bean>
<!--配置DAO的类-->
<bean id="accoutDAO" class="cn.moke.spring.demo1.AccountDAOImpl">
<!--注入连接池-->
<property name="dataSource" ref="dataSource"></property>
</bean>
<!--配置事务管理器-->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceT
ransactionManager">
    <property name="dataSource" ref="dataSource"></property><!--我们加入这个连接池，这样
子这个事务管理才能管理的到-->
</bean>
<!--配置事务管理的模板：Spring为了简化事务管理而设置的类  -->
<bean id="transactionTemplate" class="org.springframework.transaction.support.Transa
ctionTemplate">
    <property name="transactionManager" ref="transactionManager"></property><!--这里
我们需要加入事务管理器，
    因为真正进行事务管理的bean不是这个，这个只是为了简化事务管理的-->
</bean>
```

这是JAVA代码

```
public void setAccoutDAO(AccountDAO accoutDAO) {
```

```
this.accoutDAO = accoutDAO;
}

@Override
public void transfer( final String out, final String in,final double money) {
transactionTemplate.execute(new TransactionCallbackWithoutResult() {
@Override
protected void doInTransactionWithoutResult(TransactionStatus transactionStatus) {
accoutDAO.outMoney(out,money);
            int i=1/0;
accoutDAO.inMoney(in,money);
}
    });

}
```

这个在我们以后中不是经常用的，因为编程式的事务管理是需要手动的来改代码的

下面这个声明式的事务管理是基于AOP来实现的
这个是采用第一种方式，也就是配置代理类的方式

```
TransactionProxyFactoryBean
```

配置这个类的方式
事务管理器还是需要配置的，这个事务管理器是真的用来事务管理的一个类
所以我们需要用它来进行相应的配置

```
<property name="transactionManager" ref="transactionManage
<property name="transactionAttributes">
  <props>
    <prop key="insert*">PROPAGATION_REQUIRED</prop>
    <prop key="update*">PROPAGATION_REQUIRED</prop>
    <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
  </props>
</property>
```

如果单单写*就代表这个类下面的所有方法

这个一定要看好啊，我这个在配置的时候出现了bug，经过了调试终于发现是写错了。。。一定要注意
这是原始方式

```
<!-- 引入外部的属性文件 -->
<context:property-placeholder location="jdbc.properties"/>
<!--配置c3p0的连接池-->
<beanid="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="driverClass" value="${jdbc.driverClass}"></property>
    <property name="jdbcUrl" value="${jdbc.url}"></property>
    <property name="user" value="${jdbc.username}"></property>
    <property name="password" value="${jdbc.password}"></property>
</bean>

<!--配置业务层类-->
<beanid="accoutService" class="cn.moke.spring.demo2.AccountServiceImpl">
    <property name="accoutDAO"  ref="accoutDAO"></property>
<!--注入事务管理的模板-->
</bean>
<!--配置DAO的类-->
<bean id="accoutDAO" class="cn.moke.spring.demo2.AccountDAOImpl">
<!--注入连接池-->
<property name="dataSource" ref="dataSource"></property>
</bean>
<!--事务管理器-->
<beanid="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTr
ansactionManager">
    <property name="dataSource" ref="dataSource"></property>
</bean>
<!--配置业务层的代理-->
<beanid="accoutServiceProxy" class="org.springframework.transaction.interceptor.Tran
sactionProxyFactoryBean">
<!--配置目标对象-->
<property name="target" ref="accoutService"/>
<!--注入事务管理器-->
<property name="transactionManager" ref="transactionManager"/>
<!--注入的事务的一些属性-->
<property name="transactionAttributes">
        <props>
<!--pro的格式:
                    *PROPAGATION :事务的传播行为
                    *ISOLATION   :事务的隔离级别
```

```
                    *readOnly      :只读(不可以进行删除，修改的操作的)
                    *-EXCEPTION    :发生哪些异常回滚事务
                    *+EXCEPTION    :发生哪些异常事务不回滚
                -->
                <!--<prop key="transfer">PROPAGATION_REQUIRED,+java.lang.ArithmeticExce
ption</prop>-->
<prop key="transfer">PROPAGATION_REQUIRED </prop>
            </props>
        </property>

</bean>
```

```
public  void demo2(){
    ApplicationContext context=
new ClassPathXmlApplicationContext("applicationContext2.xml");
AccountService accountService= (AccountService) context.getBean("accoutServiceProxy"
);
accountService.transfer("aaa","bbb",200d);
}
```

这种比较原始的方法，这种不经常使用，这个就是因为我们需要为每一个要去事务管理的类，都去配置一个transactionProxyFactoryBean
。这就会配置很多transactionProxyFactoryBean，这样开发起来很麻烦

这是原始的方式

下面是基于AspectJ XML的一种配置方式

其实AspectJ本身是一种的AOP层的框架，
Spring为了方便开发，就引入过来了 。
跟上面不一样的是，我们getBean这块不用再改了，因为这是种自动代理，在类的生成过程当中，这个类的本身就是一个代理对象了，不需要再像上面一样注入了

声明式事务管理第二种方式：通过使用aop的方式，spring配置文件定义一个通知，制定事务管理器和事务传播行为等，再配置apo的切入点和切面，切面引用切入点，制定切入的方法等

```
xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop-3.1.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
      http://www.springframework.org/schema/task
http://www.springframework.org/schema/task/spring-task-3.1.xsd">
<!-- 引入外部的属性文件 -->
<context:property-placeholder location="jdbc.properties"/>
<!--配置c3p0的连接池-->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
     <property name="driverClass" value="${jdbc.driverClass}"></property>
     <property name="jdbcUrl" value="${jdbc.url}"></property>
     <property name="user" value="${jdbc.username}"></property>
     <property name="password" value="${jdbc.password}"></property>
 </bean>

<!--配置业务层类-->
<bean id="accoutService" class="cn.moke.spring.demo3.AccountServiceImpl">
     <property name="accoutDAO" ref="accoutDAO"></property>
<!--注入事务管理的模板-->
</bean>
<!--配置DAO的类-->
<bean id="accoutDAO" class="cn.moke.spring.demo3.AccountDAOImpl">
<!--注入连接池-->
<property name="dataSource" ref="dataSource"></property>
 </bean>
<!--配置事务管理器，这个是必须的-->
<bean id="transactionManager2" class="org.springframework.jdbc.datasource.DataSource
TransactionManager">
     <property name="dataSource" ref="dataSource"></property>
 </bean>
<!--配置的是事务的一个通知（事务的增强）-->
<tx:advice id="txAdvice"  transaction-manager="transactionManager2">
   <tx:attributes>
<!--
       propagation: 事务的传播级别
       isolation : 事务的隔离级别
       read-only: 只读
       rollback-for: 发生哪些异常回滚
```

```
        no-rollback-for: 发生哪些异常不回滚
        timeout: 过期信息
        -->
<tx:method name="transfer" propagation="REQUIRED"/>
   </tx:attributes>
 </tx:advice>
<!--配置AOP的切面-->
<aop:config>
     <aop:pointcut id="pointcut1" expression="execution(* cn.moke.spring.demo3.AccountService+.*(..))"></aop:pointcut>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="pointcut1"></aop:advisor>
 </aop:config>
```

```
ApplicationContext context=
new ClassPathXmlApplicationContext("applicationContext2.xml");
cn.moke.spring.demo2.AccountService accountService= (cn.moke.spring.demo2.AccountService) context.getBean("accoutServiceProxy");
accountService.transfer("aaa","bbb",200d);
```

基于注解方式的配置

关于上面的设置的一样的属性，是可以在注解中的属性声明的

这个是XML配置的方面

```
<!--配置事务管理器-->
<bean id="transactionManager3" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
          <property name="dataSource" ref="dataSource"/>
    </bean>
<!--开启注解事务-->
<tx:annotation-driven transaction-manager="transactionManager3"/>
```

```
/**
 * 转账业务层的实现类
 * Created by Administrator on 2016/5/25.
 *//*
 * @Transactional注解中的属性
 * propagation  事务的传播行为
 * isolation  事务的隔离级别
 * readOnly  只读信息
 *rollbackFor  发生哪些异常回滚
 * noRollbackFor  发生哪些异常不回滚
 * noRollbackForClassName  这个是安装类的名称来设置，如果不写，就是默认值
 * */
@Transactional(propagation = Propagation.REQUIRED ,isolation = Isolation.DEFAULT,readOnly = true ,noRollbackForClassName = )
public class AccountServiceImpl implements AccountService {
//   注入转账的一个类
private AccountDAO accoutDAO;
```

总结：Spring将事务管理

分为俩种：
1 编程式的事务管理
需要手动编写代码，在以后的开发中很少使用

2 声明式的事务管理，
1 基于TransactionProxyFactory 的方式 这也是很少使用的，配置和管理都很麻烦
2 基于AspectJ的事务管理（经常使用）
一旦配置好后，类上不需要任何东西
3 基于注解的事务管理（经常使用）配置简单，需要在业务层上类上去添加注解，找起来麻烦

但是每个都是需要配置事务管理器

```
<!--配置事务管理器，这个是必须的-->
```

```
<bean id="transactionManager2" class="org.springframework.jdbc.datasource.DataSource
TransactionManager">
    <property name="dataSource" ref="dataSource"></property>
</bean>
```