

依赖注入  
DI

1

面向切面编程  
AOP

2

目录

3

声明式事务  
DT

4

SSH  
整合



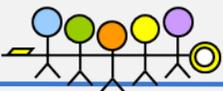
声明式事务介绍



事务管理器依赖树



配置代码示例



# 事务特性

事务必须具备ACID的特性：

## ■原子性(Atomic)

整个事务中的所有操作，是一个不可分割的整体。

要么一起成功，要么一起失败，不可能停滞在中间某个环节。

## ■一致性(Consistency)

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。

## ■隔离级别(Isolation)

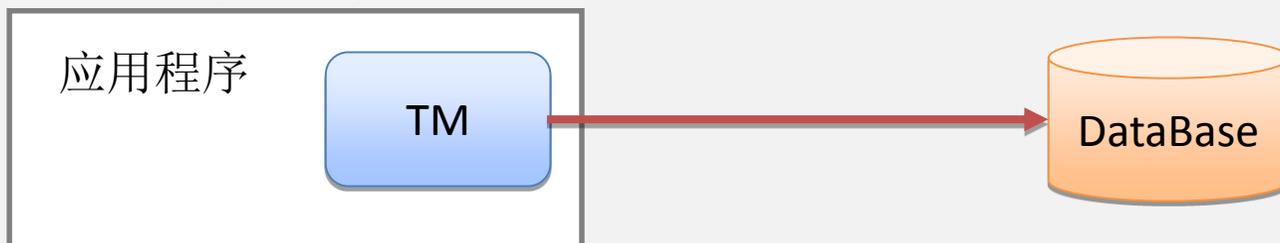
并发的事务是相互隔离的，如果有两个事务，运行在相同的时间内，执行相同的功能，事务的隔离性将确保每一事务在系统中认为只有该事务在使用系统。

## ■持久性(Durable)

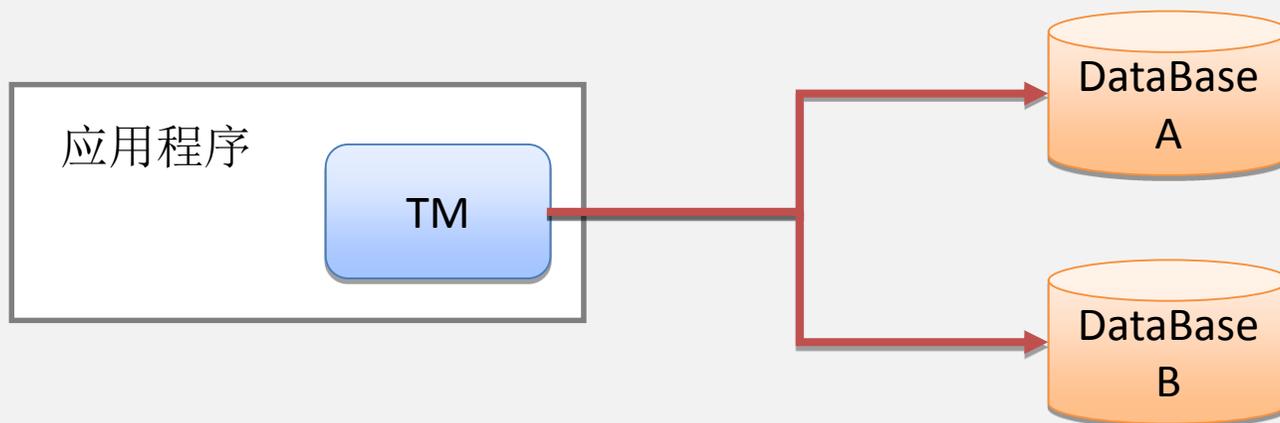
事务一旦提交，数据将会被持久化。

如果按数据来源分类，有以下两类：

- 本地事务(Local)，只连接一个数据库。



- 全局事务(Global)，可以跨越多个数据库。



如果按事务的实现方式分类，有以下两类：

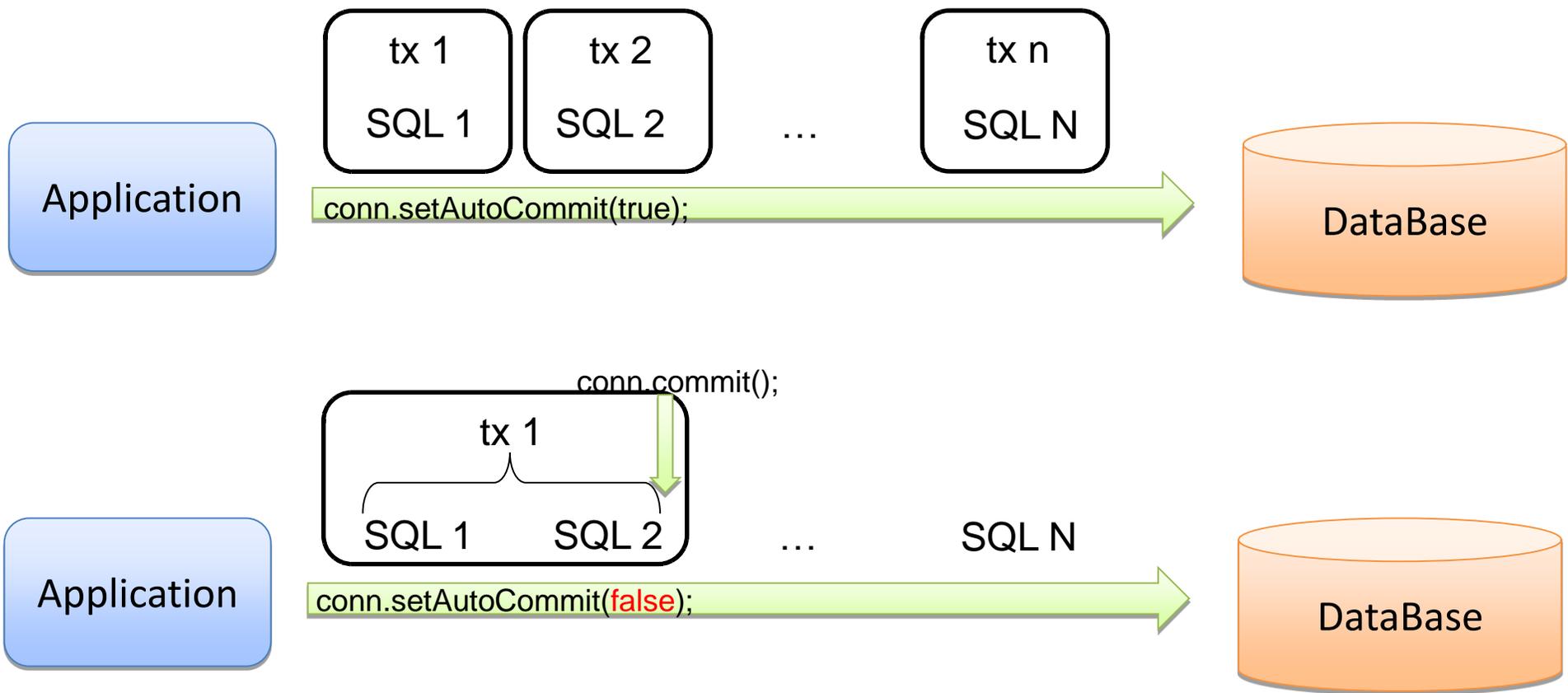
- 编程式事务通过coding来控制事务的开启，提交，和回滚。

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
try {
    session.save(user);
    tx.commit();
} catch (Exception e) {
    tx.rollback();
    e.printStackTrace();
} finally {
    session.close();
}
```

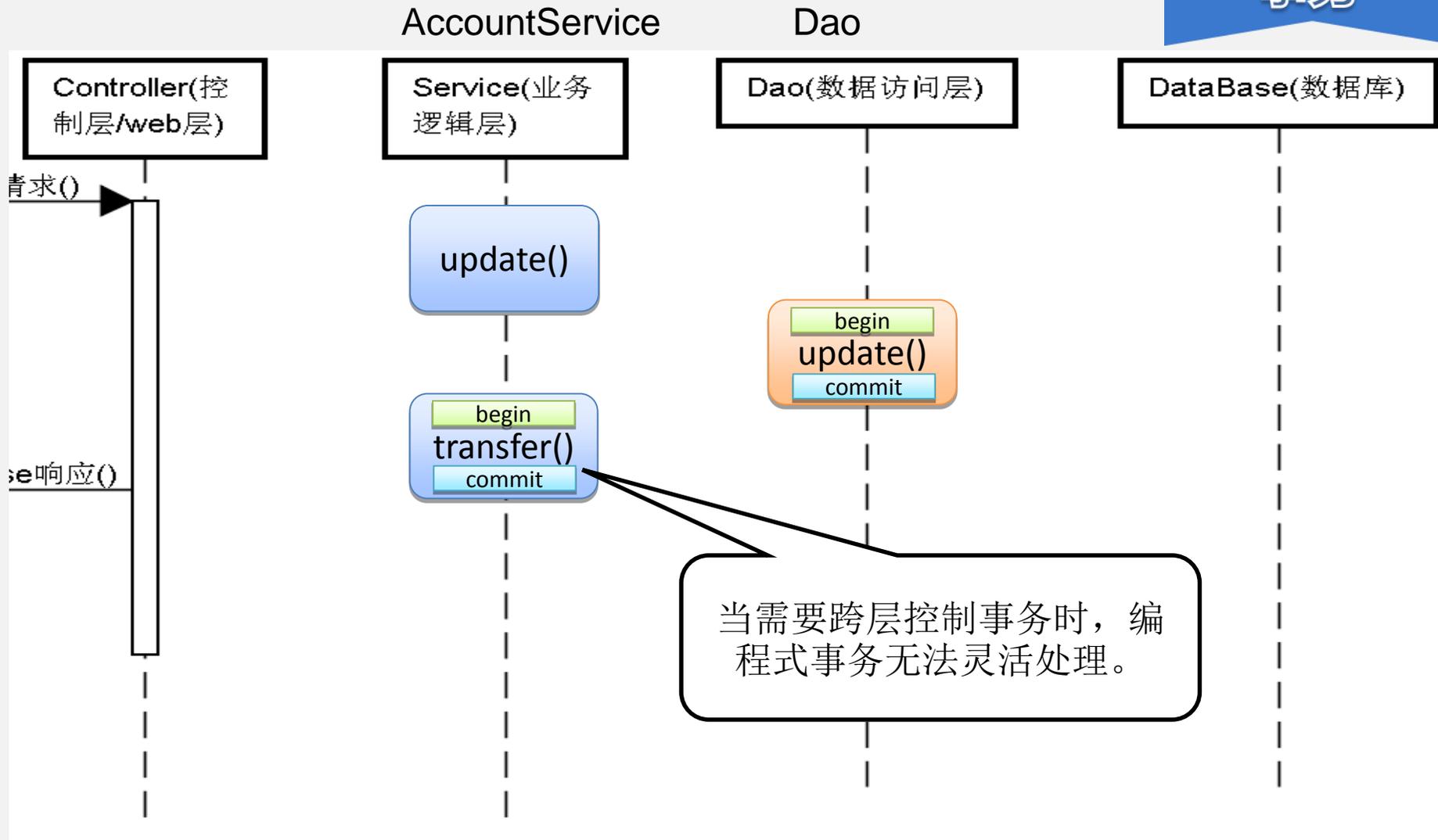
- 声明式事务，通过声明定义哪些类的哪些方法怎么来参与事务。(推荐)

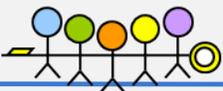
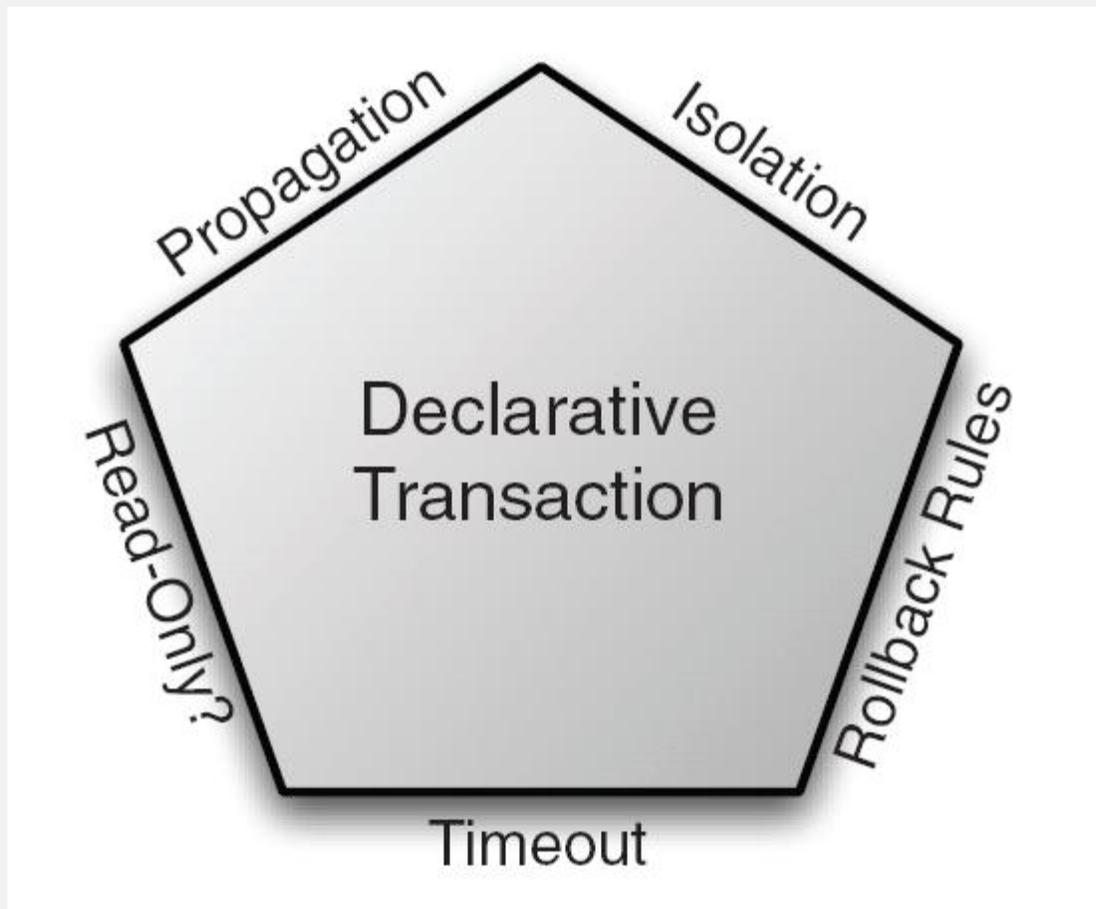
# 控制事务的本质

控制事务，即控制事务的边界。



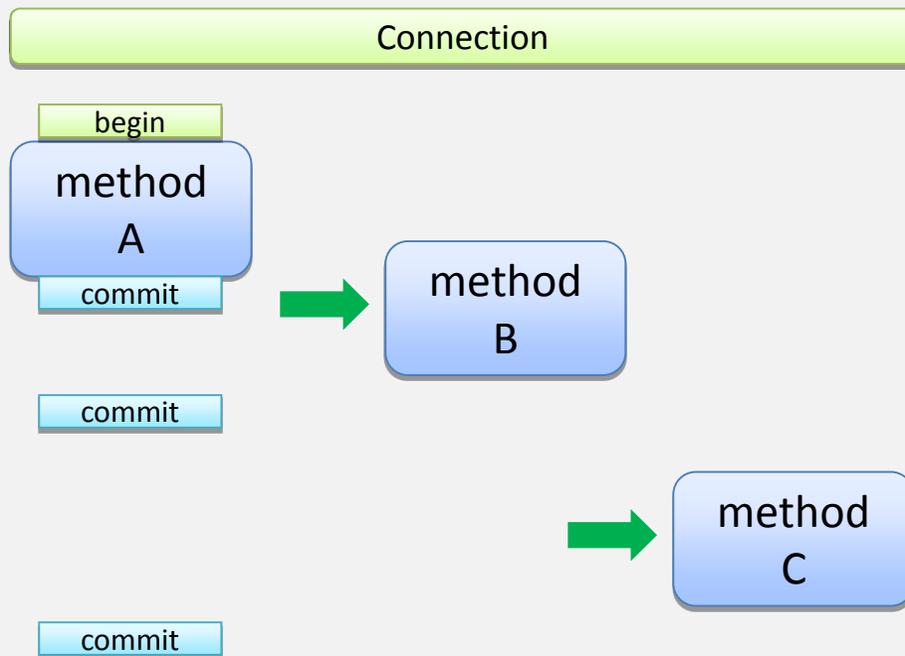
# 程式式事务不足





# Propagation(传播行为)

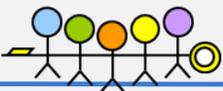
被定义了声明式事务的方法，有权知道当前是否有事务开启，并有权决定是否加入当前事务。某个方法增加到当前事务，相当于事务被“传播了”。  
传播的实质，也就是对事务的边界，在程序运行期间“动态地”进行控制。



传播行为有以下7种设定值：

设定值	说明
REQUIRED (默认值)	如果没有事务，则开始新的事务，事务已经存在，则加入当前事务。
SUPPORTS	如果事务已经存在，则加入当前事务，如果没有事务，不会开始新事务。
MANDATORY	必须有事务存在，并加入当前事务，否则抛出异常。
REQUIRES_NEW	每次都开始一个全新事务
NOT_SUPPORTED	不会开始或加入事务
NESTED	如果事务已经存在，则运行在一个嵌套的事务中。如果没有活动事务，则开始新的事务。
NEVER	表示不会开始或加入事务，如果事务已存在，抛出异常

	当前事务状态	被声明的方法
REQUIRED	Tx1	Tx1
	None	Tx1
SUPPORTS	Tx1	Tx1
	None	None
MANDATORY	Tx1	Tx1
	None	Exception
REQUIRES_NEW	Tx1	Tx2
	None	Tx1
NOT_SUPPORTED	Tx1	None
	None	None
NESTED	Tx1	Tx2
	None	Tx1
NEVER	Tx1	Exception
	None	None



# 隔离级别

**READ\_UNCOMMITTED**

——表示会读取到未提交的数据

**READ\_COMMITTED (默认值)**

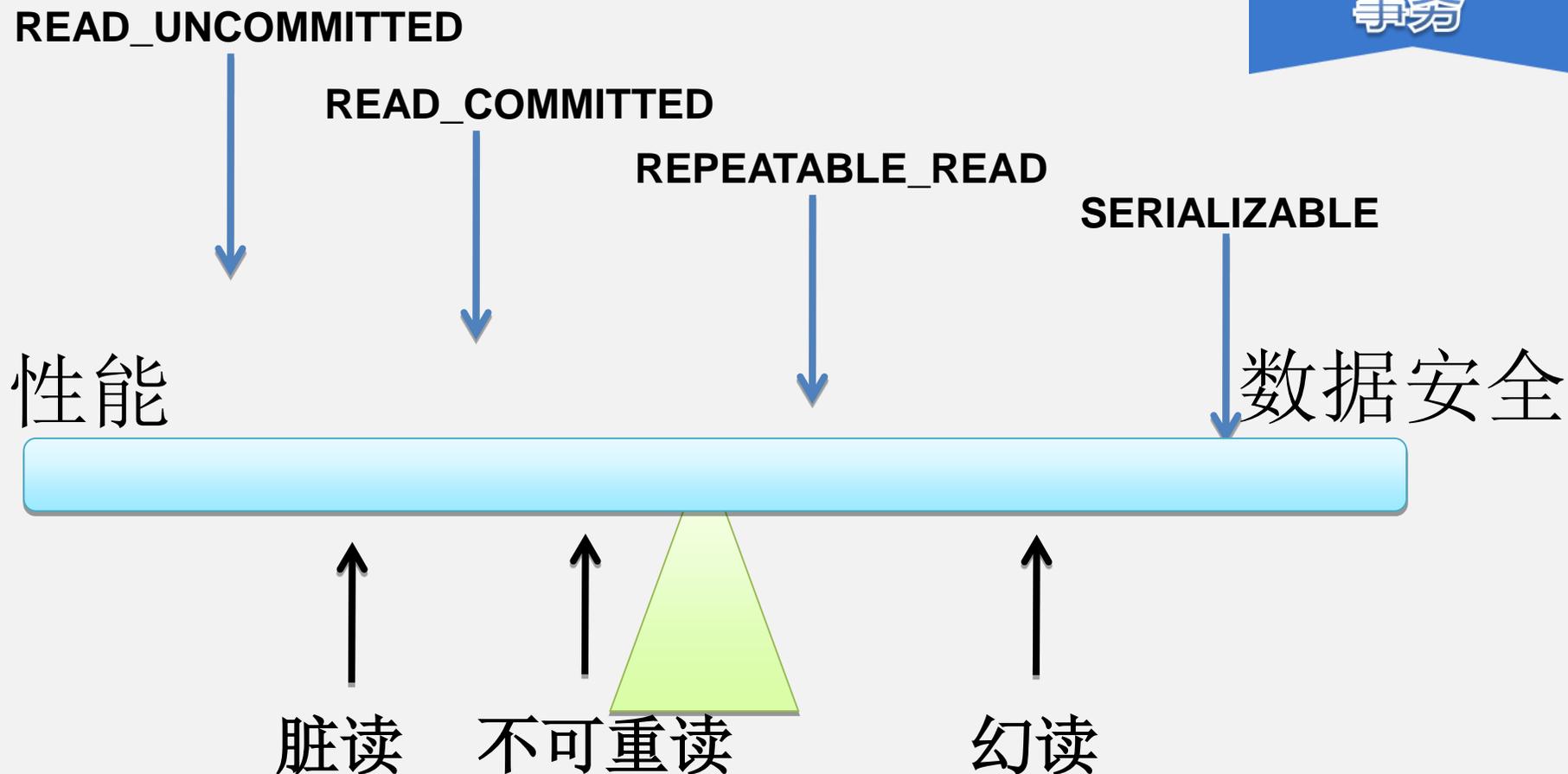
——表示读取到的是已经提交的数据

**REPEATABLE\_READ**

——表示多次读取的数据是相同的

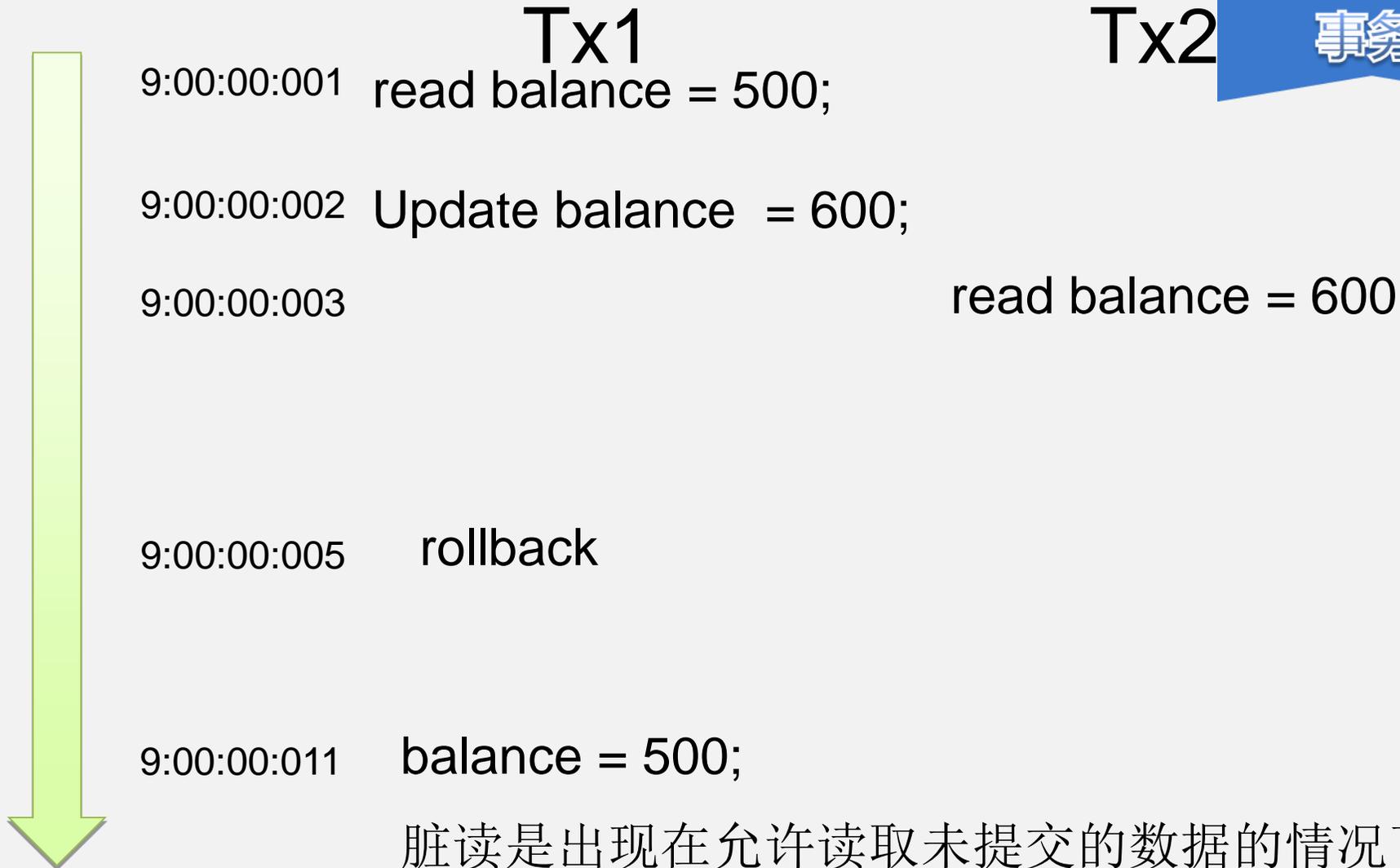
**SERIALIZABLE**

——表示就好像没有别的用户在修改数据库中的数据一样



没有哪个隔离级别是最好的，实际项目开发，需要根据需求来决定合适的。

# 脏读



# 不可重读

Tx1

9:00:00:001

9:00:00:002 Update balance = 600;

9:00:00:003 commit

9:00:00:004

Tx2

read balance = 500

read balance = 600

不可重读是短时间内读取2次,但结果不一致.  
第二次读取之前,数据已被另一事务更新了.

Tx1

Tx2

9:00:00:001 Update all account  
balance = 0;

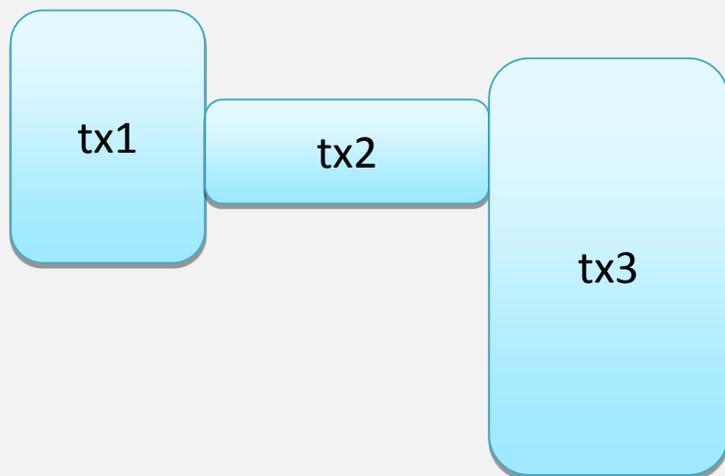
9:00:00:002

Inert into new account  
Balance = 500;

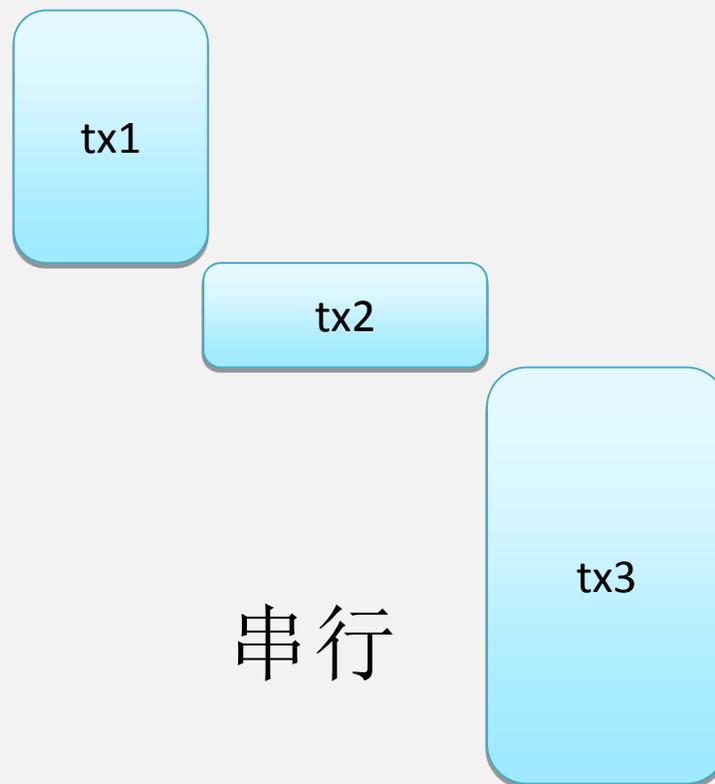
9:00:00:003 Select \* from account;

幻读是由于事务的并发性引起的。

# 并发 vs 串行



并发



串行

声明式事务,是由Spring中的事务管理器管理的.

Spring由于需要为多种数据层的实现提供集成支持,针对不同的情况,定义了各种事务管理器,一定要选择正确的事务管理器

比如:

DataSourceTransactionManager 对应JDBC事务

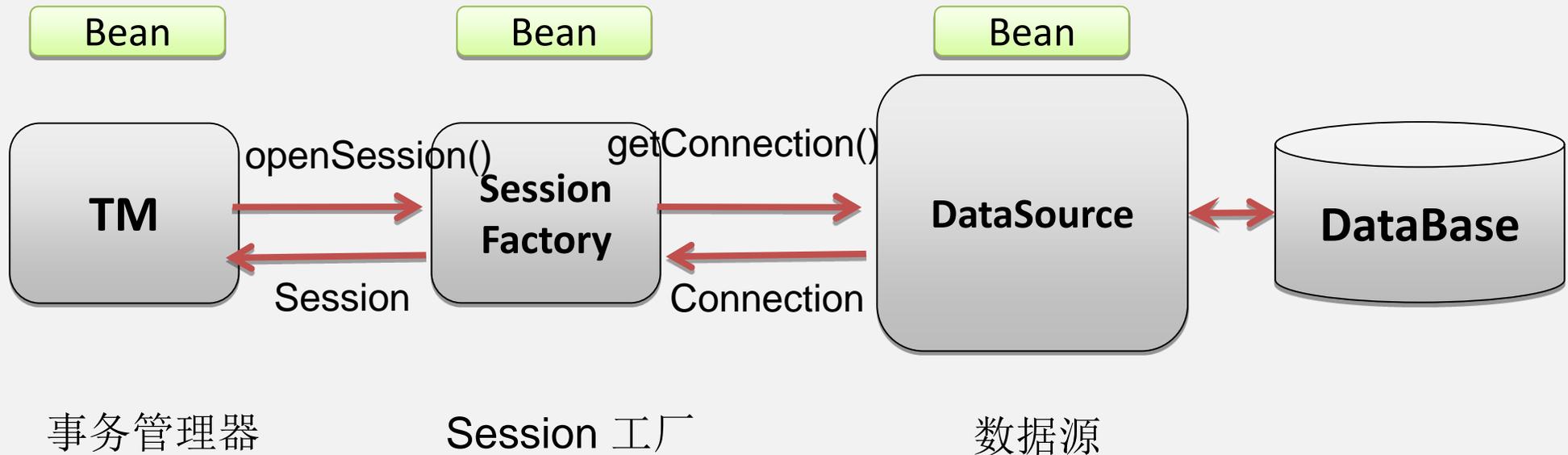
HibernateTransactionManager 对应Hibernate提供的事务(学习目标)

JtaTransactionManager 对应JTA事务

备注: 大多数事务管理器都需要设置dataSource (数据源) 属性或 sessionFactory 属性

# HibernateTransactionManager

在Spring配置如下的依赖树:



Spring支持用两种方式配置声明式事务：

- 1) 注解配置(上课演示)
- 2) XML配置(扩展阅读)

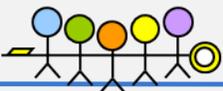
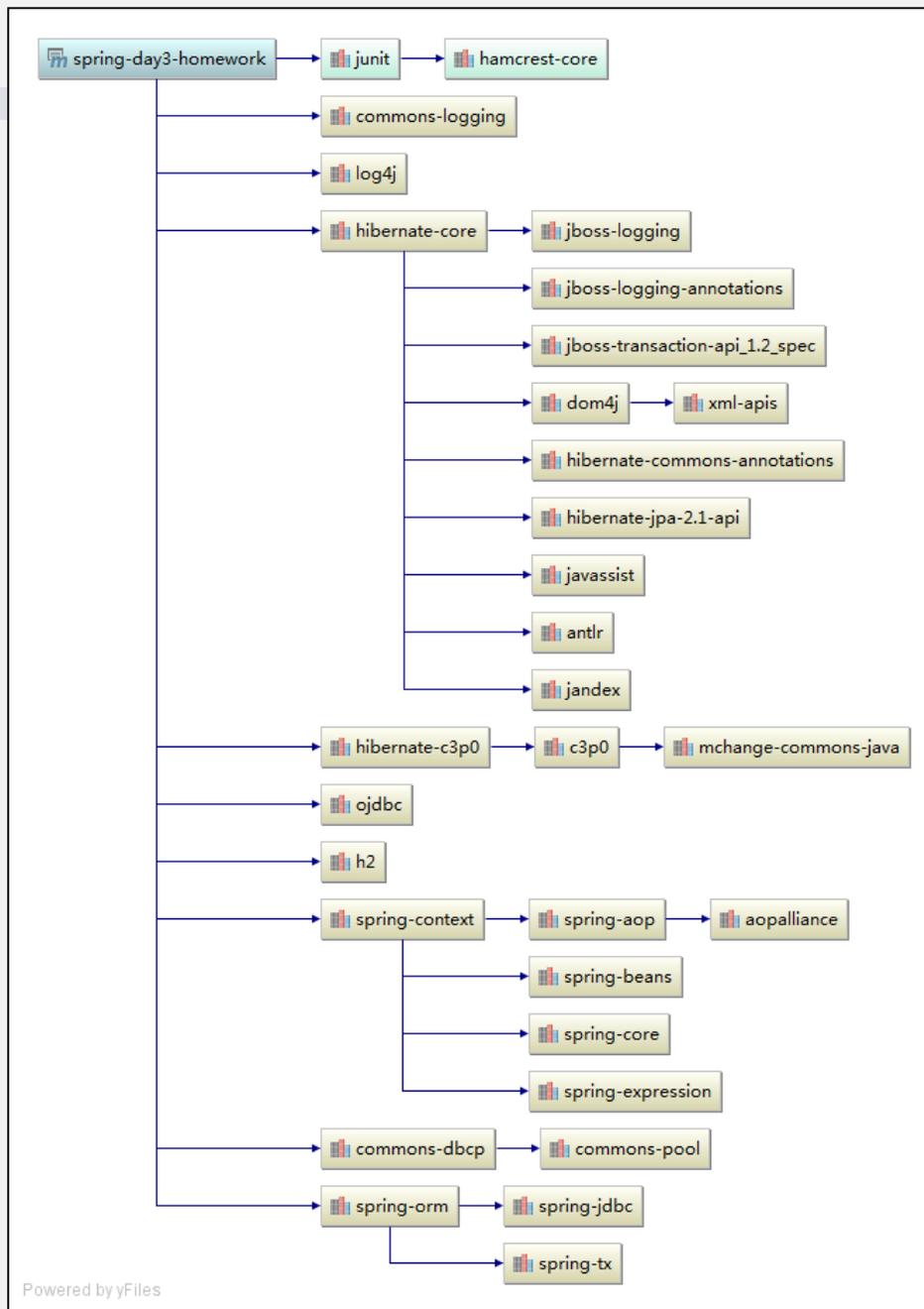
都是实现同一个目标的不同方式

## @注解配置

1. 建立项目
2. pom.xml 配置 spring DT 依赖
3. 写配置信息类 AppConfig
4. 将TM依赖树，被切组件配置到 spring 容器
5. 使用@Transactional注解配置事务边界
6. 写单元测试并调试

# 项目依赖图

## 第三章 声明式 事务



```
/**
 * 配置数据源实现类
 */
@Bean
public DataSource getDataSource() {
    LOGGER.debug("getDataSource() run...");
    //实例化BasicDataSource实例
    BasicDataSource basicDataSource = new BasicDataSource();
    //设置各种参数
    basicDataSource.setDriverClassName("oracle.jdbc.OracleDriver");
    basicDataSource.setUrl("jdbc:oracle:thin:@localhost:1521:XE");
    basicDataSource.setUsername("jsd1502");
    basicDataSource.setPassword("jsd1502");
    basicDataSource.setInitialSize(10);
    return basicDataSource;
}
```

这里演示的是BasicDataSource,  
只要是实现了javax.sql.DataSource  
的实现类都可配置!

# @PropertySource

```
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://127.0.0.1:3306/exam?useUnicode=true&characterEncoding=UTF-8
username=root
password=

dialect=org.hibernate.dialect.
```

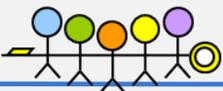
```
@Configuration
@PropertySource("classpath:app.properties")
@ComponentScan
@EnableAspectJAutoProxy
@EnableTransactionManagement
public class AppConfig {

    private static final Logger LOGGER = Logger.getLogger(AppConfig.class);

    @Autowired
    private Environment env;

    /**
     * 配置数据源实现类
     */
    @Bean
    public DataSource getDataSource(){
        LOGGER.debug("getDataSource() run...");
        //实例化BasicDataSource实例
        BasicDataSource basicDataSource = new BasicDataSource();
        //设置各种参数
        LOGGER.debug(env.getProperty("driver"));

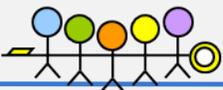
        basicDataSource.setDriverClassName(env.getProperty("driver"));
        basicDataSource.setUrl(env.getProperty("url"));
        basicDataSource.setUsername(env.getProperty("username"));
        basicDataSource.setPassword(env.getProperty("password"));
        basicDataSource.setInitialSize(10);
        return basicDataSource;
    }
}
```



# SessionFactory

```
/**
 * SessionFactory的封装类(FactoryBean)
 * 当被使用时,对外所暴露的是 org.hibernate
 */
@Bean
public LocalSessionFactoryBean getLocalSessionFactoryBean() {
    LOGGER.debug("getLocalSessionFactoryBean() run...");
    //实例化
    LocalSessionFactoryBean localSessionFactoryBean = new LocalSessionFactoryBean();
    //1.设置数据源
    localSessionFactoryBean.setDataSource(this.getDataSource());
    //2.方言
    Properties props = new Properties();
    props.setProperty("hibernate.dialect", "org.hibernate.dialect.Oracle10gDialect");
    localSessionFactoryBean.setHibernateProperties(props);
    //3.注册实体类
    localSessionFactoryBean.setPackagesToScan("com.java.entity");
    return localSessionFactoryBean;
}
```

其中封装了  
**org.hibernate.SessionFactory.**  
所需配置的信息量一点没少,  
只不过改为Spring的配置方式



# HibernateTransactionManager

```
@Bean
@Autowired
public HibernateTransactionManager getHibernateTransactionManager(SessionFactory
 sessionFactory){
    LOGGER.debug("PlatformTransactionManager() run...");
    HibernateTransactionManager htm =
        new HibernateTransactionManager(sessionFactory);
    return htm;
}
```

底层基于AOP的实现，在这些方法运行之前，使用声明式事务的传播行为，将事务边界进行灵活的扩展。  
学员可以借助之前写过的MyTxAdvice来帮助理解。

# @Transactional

位置:方法名或类名上方

作用:该方法具有声明式事务,或该类的所有方法具有声明式事务

效果:

在方法运行前,自动开启事务,将session存入线程;

在方运行成功后,自动提交事务;

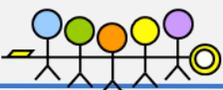
在方法异常后,自动回滚事务.

细节:

- 方法级别的定义,可以覆盖类级别的定义。
- 其中可以自定义传播行为,隔离级别,只读等属性。

将声明式事务管理简化到了极致

```
@Transactional(propagation = Propagation. REQUIRED  
                , isolation = Isolation. READ_COMMITTED)
```



# @EnableTransactionManagement

位置:配置信息类的类名上方

作用:启用声明式事务功能

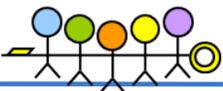
```
@Configuration
@ComponentScan
@EnableTransactionManagement //启用基于注解的 声明式事务
public class AccountConfig {

    private static final Logger LOGGER = Logger.getLogger(AccountConfig.class);

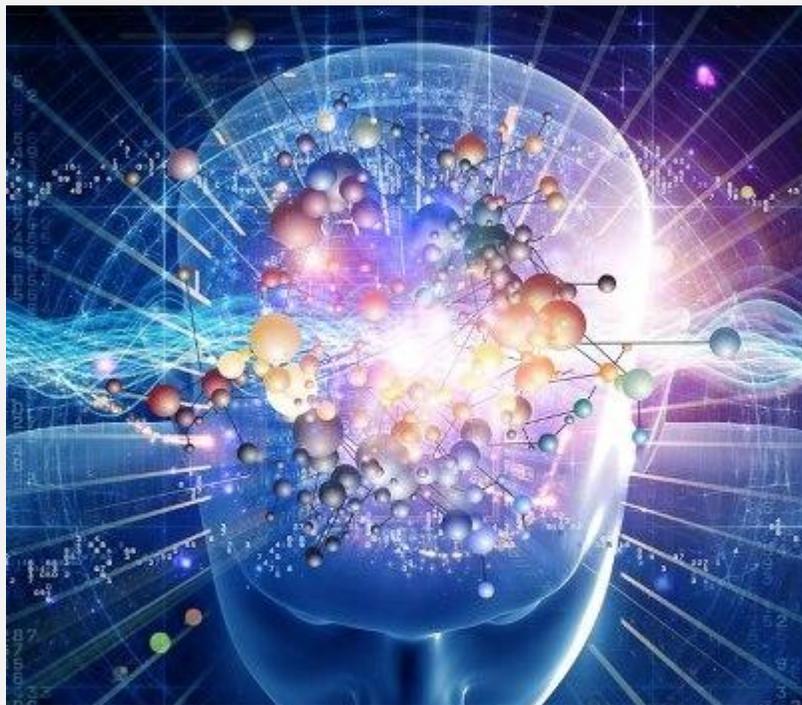
    /**
     * 配置数据源实现类
     */
    @Bean
    public DataSource getDataSource() {...}

    /**
     * sessionFactory的封装类(FactoryBean)
     * 当被使用时,对外所暴露的是 org.hibernate.SessionFactory
     */
    @Bean
    public LocalSessionFactoryBean getLocalSessionFactoryBean() {...}

    @Bean
    @Autowired
    public PlatformTransactionManager getPlatformTransactionManager(SessionFactory sessionFactory) {...}
}
```



# 形象化思维要求



因为事务控制的代码,都已被精简掉了。  
开发者眼中看到的是代码,  
脑海中要出现事务的数量,边界,和传播的次数。

## XML 配置

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/tx
  http://www.springframework.org/schema/tx/spring-tx.xsd
  http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop.xsd">

</beans>
```

本页内容来自spring开发手册（[spring-framework-reference](http://spring-framework-reference.com)）  
如有差异，请以开发手册为准。

BasicDataSource / ComboPooledDataSource

LocalSessionFactoryBean

HibernateTransactionManager

service & dao

tx:advice

aop:config



搭积木式的装配方式

### C3P0数据源

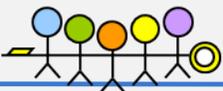
```
<!-- 1.dataSource -->  
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  
  <property name="driverClass" value="oracle.jdbc.driver.OracleDriver"/>  
  <property name="jdbcUrl" value="jdbc:oracle:thin:@localhost:1521:XE"/>  
  <property name="user" value="root"/>  
  <property name="password" value="root"/>  
</bean>
```

### BasicDataSource数据源

```
<!-- 1.BasicDataSource -->  
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">  
  <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>  
  <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE"/>  
  <property name="username" value="root"/>  
  <property name="password" value="root"/>  
</bean>
```

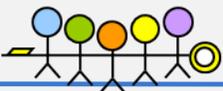
任选一个数据源实现类即可，无需两个都配置。

```
<!-- 2.SessionFactory -->
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
  <!-- a)数据库连接信息(DataSource) -->
  <property name="dataSource" ref="dataSource"/>
  <!-- b)方言等属性 -->
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect</prop>
      <prop key="hibernate.show_sql">>true</prop>
    </props>
  </property>
  <!-- c)注册ORM -->
  <property name="packagesToScan" value="com.tz.spring.entity"/>
</bean>
```



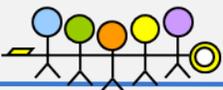
```
<!-- 3.TM -->  
<bean id="tm"  
      class="org.springframework.orm.hibernate4.HibernateTransactionManager">  
  <property name="sessionFactory" ref="sessionFactory"/>  
</bean>
```

```
<!-- 4.被切的组件 (service & dao) -->  
<bean id="accountDao" class="com.tz.spring.dao.AccountDaoImpl">  
  <property name="sessionFactory" ref="sessionFactory"/>  
</bean>  
<bean id="accountService" class="com.tz.spring.service.AccountServiceImpl">  
  <property name="accountDao" ref="accountDao"/>  
</bean>
```



```
<!-- 5.txAdvice(内置实现好,稍作配置即可) -->  
<tx:advice id="txAdvice" transaction-manager="tm">  
  <tx:attributes>  
    <!-- 定义切面涉及的方法,怎么来参与声明式事务 -->  
    <tx:method name="*" isolation="READ_COMMITTED" propagation="REQUIRED"/>  
  </tx:attributes>  
</tx:advice>
```

```
<!-- 6.配置 声明式事务切面 -->  
<aop:config>  
  <aop:advisor advice-ref="txAdvice"  
    pointcut="execution(* com.tz.spring.service.*ServiceImpl.*(..))"/>  
</aop:config>
```



```
public class AccountDaoImpl implements AccountDao {  
  
    private SessionFactory sessionFactory;  
    public void setSessionFactory(SessionFactory sessionFactory) {  
        this.sessionFactory = sessionFactory;  
    }  
  
    private Session getSession(){  
        //获得一个和当前线程捆绑的Session  
        return sessionFactory.getCurrentSession();  
    }  
  
    @Override  
    public Account get(int id) {  
        return (Account)getSession().get(Account.class, id);  
    }  
  
    @Override  
    public void update(Account acc) {  
        getSession().update(acc);  
    }  
}
```

使用声明式事务的写法，  
Dao层代码极度精简。



commons-dbcp是 **Apache** 下的开源数据库连接池jar包。提供了一个轻量级、简单高效、多功能的一站式实现。可以满足基本的需求，节省开发者自行开发连接池的时间。

创建一个BasicDataSource实例，必须设置以下属性：

`protected String driverClassName = null;` 驱动类名

`protected String url = null;` url

`protected String username = null;` 用户名

`protected volatile String password = null;` 密码

该jar包运行期间需要依赖commons-pool.jar包

FactoryBean对于Spring来说具有重要的地位，用户可以通过实现该工厂接口定制实例化Bean的逻辑。

```
org.springframework.beans.factory
```

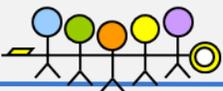
```
public interface FactoryBean<T>
```

Interface to be implemented by objects used within a [BeanFactory](#) which are themselves factories. If a bean implements this interface, it is used as a factory for an object to expose, not directly as a bean instance that will be exposed itself.

**NB: A bean that implements this interface cannot be used as a normal bean.** A FactoryBean is defined in a bean style, but the object exposed for bean references ([getObject\(\)](#)) is always the object that it creates.

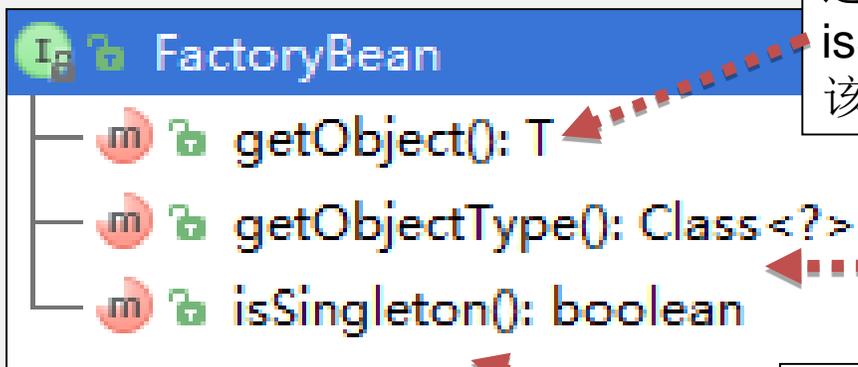
FactoryBeans can support singletons and prototypes, and can either create objects lazily on demand or eagerly on startup. The [SmartFactoryBean](#) interface allows for exposing more fine-grained behavioral metadata.

This interface is heavily used within the framework itself, for example for the AOP [org.springframework.aop.framework.ProxyFactoryBean](#) or the [org.springframework.jndi.JndiObjectFactoryBean](#). It can be used for application components as well; however, this is not common outside of infrastructure code.



# FactoryBean API

FactoryBean接口定义了三个方法：



返回由FactoryBean创建的Bean的实例，如果isSingleton（）方法返回true,是单例的实例，该实例将放入Spring的缓冲池中

返回FactoryBean创建的Bean的类型

确定由FactoryBean创建的Bean的作用域是singleton还是prototype

# FactoryBean 示例 - 1

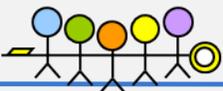
```
public class Car {  
  
    private String name;  
  
    public Car() {  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

Car Bean

```
public class CarFactoryBean implements FactoryBean<Car>{  
  
    @Override  
    public Car getObject() throws Exception {  
        System.out.println(" FactoryBean getObject() run...");  
        Car car = new Car();  
        car.setName("myCar");  
        return car;  
    }  
  
    @Override  
    public Class getObjectType() {  
        return Car.class;  
    }  
  
    @Override  
    public boolean isSingleton() {  
        return true;  
    }  
}
```

实现getObject方法，返回Car实例。

定义Car的实例为单例



# FactoryBean 示例 - 2

将CarFactoryBean配置在XML中

```
<bean id="car" class="com.tz.spring.CarFactoryBean"/>
```

单元测试:

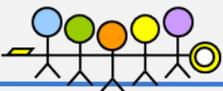
```
@Test
public void testUser() {
    ApplicationContext ac = new ClassPathXmlApplicationContext("ap
    Car car = (Car)ac.getBean("car");
    System.out.println(car);
    Car car2 = (Car)ac.getBean("car");
    System.out.println(car2);
    Car car3 = (Car)ac.getBean("car");
    System.out.println(car3);
}
```



运行结果

```
FactoryBean getObject() run...
com.tz.spring.Car@2c7f8123
com.tz.spring.Car@2c7f8123
com.tz.spring.Car@2c7f8123
```

CarFactoryBean产生了Car的实例，且Car的实例是单例模式。

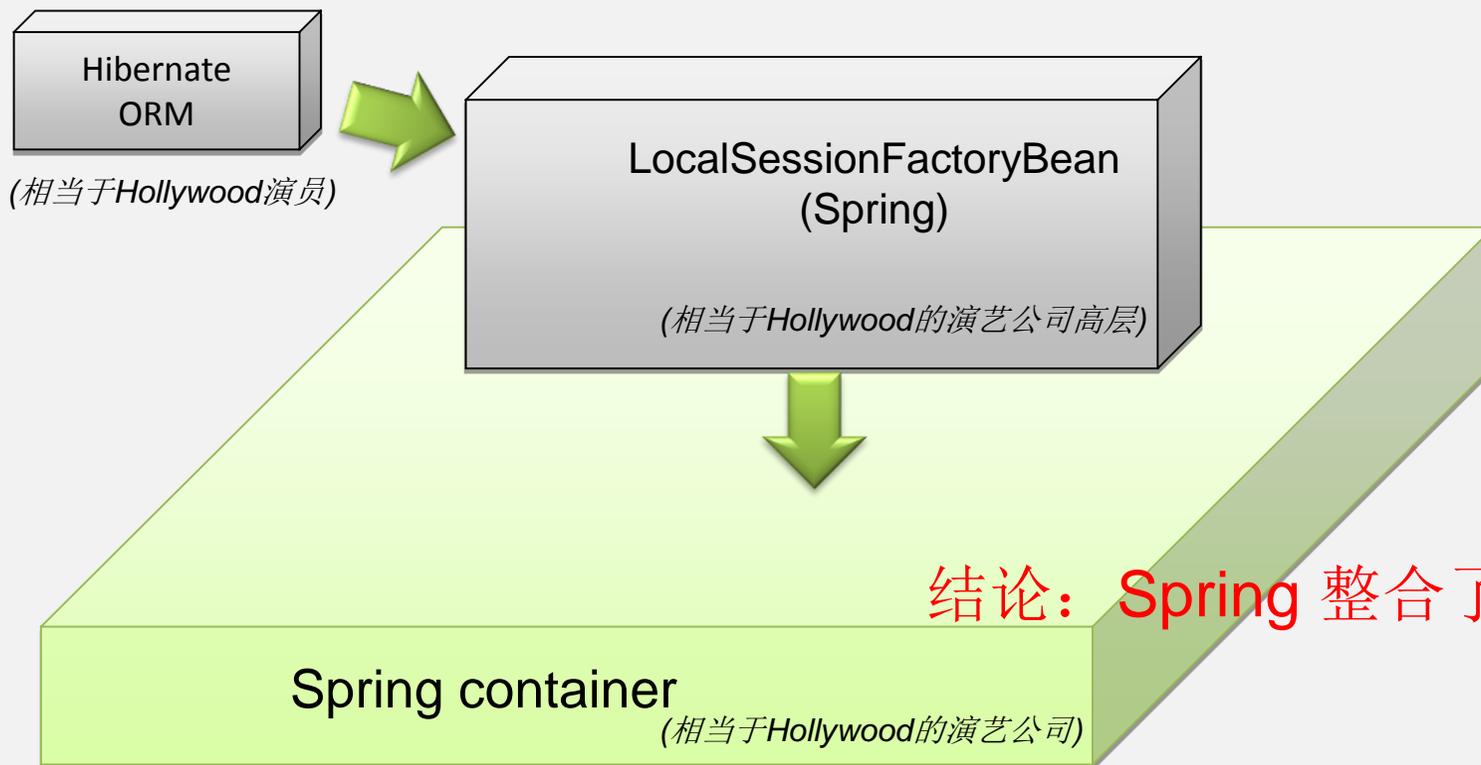


# LocalSessionFactoryBean

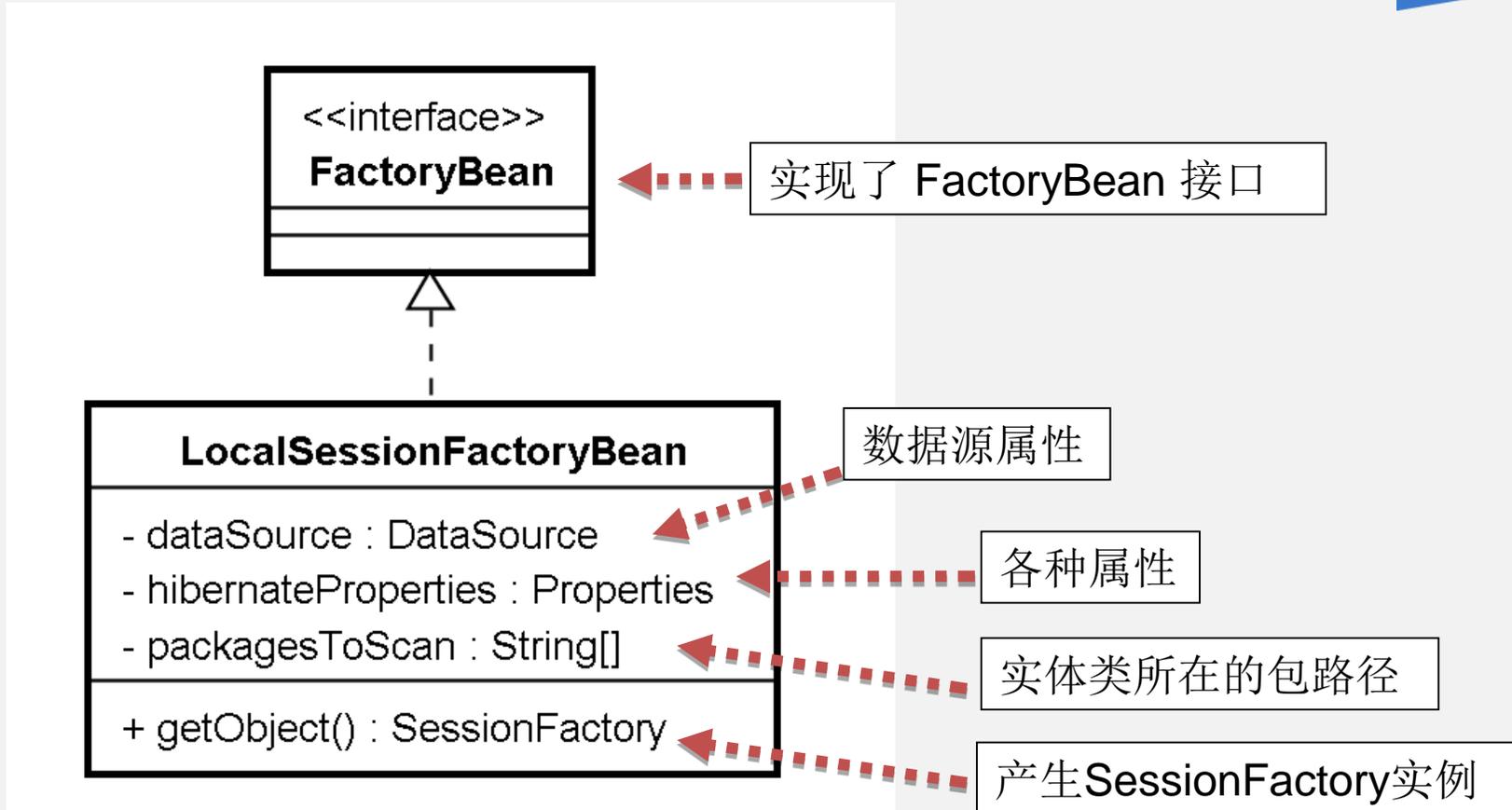
Spring容器框架，提供了对主流开发框架的整合支持。

其中，spring-orm.jar包是负责对ORM框架的支持，比如:Hibernate。

是由 org.springframework.orm.hibernate4.LocalSessionFactoryBean来完成，见下图：



# LocalSessionFactoryBean



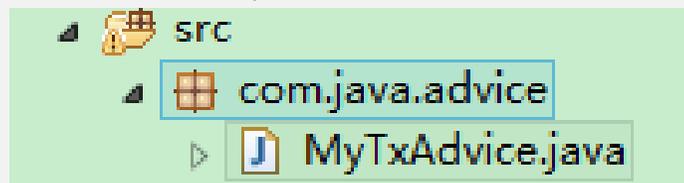
本质：将Hibernate中的SessionFactory对象，以Spring的规范进行实例化！

# <tx:method/>

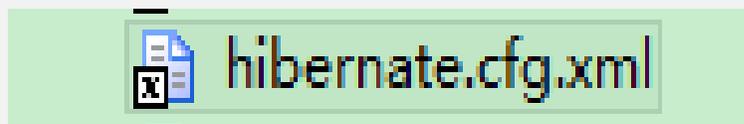
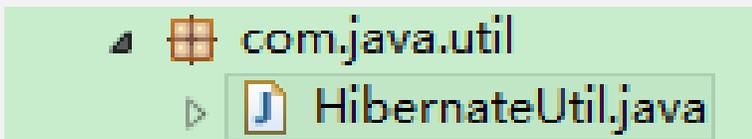
属性	是否必须	默认值	说明
name	Yes		定义方法名，可以使用*通配符。 比如: 'get*', 'handle*', 'on*Event'等。
propagation	No	REQUIRED	定义传播行为
isolation	No	DEFAULT	定义隔离级别
timeout	No	-1	定义超时的值(单位:秒)，超时事务自动回滚。
read-only	No	false	是否只读事务，把查询的操作定义成只读事务，可以减少开销，优化性能。
rollback-for	No		定义会触发事务回滚的异常类型，用逗号分隔。 比如: 'com.foo.MyBusinessException,ServletException'

# 手动版 vs 自动版

1. 前者中的 MyTxAdvice, 被后者中的 `<tx:advice />` 标签和 TM 事务管理器替代。



2. 前者中的 HibernateUtil 和 hibernate.cfg.xml 负责产生一个单例的 SessionFactory, 被后者中的 LocalSessionFactoryBean 替代。



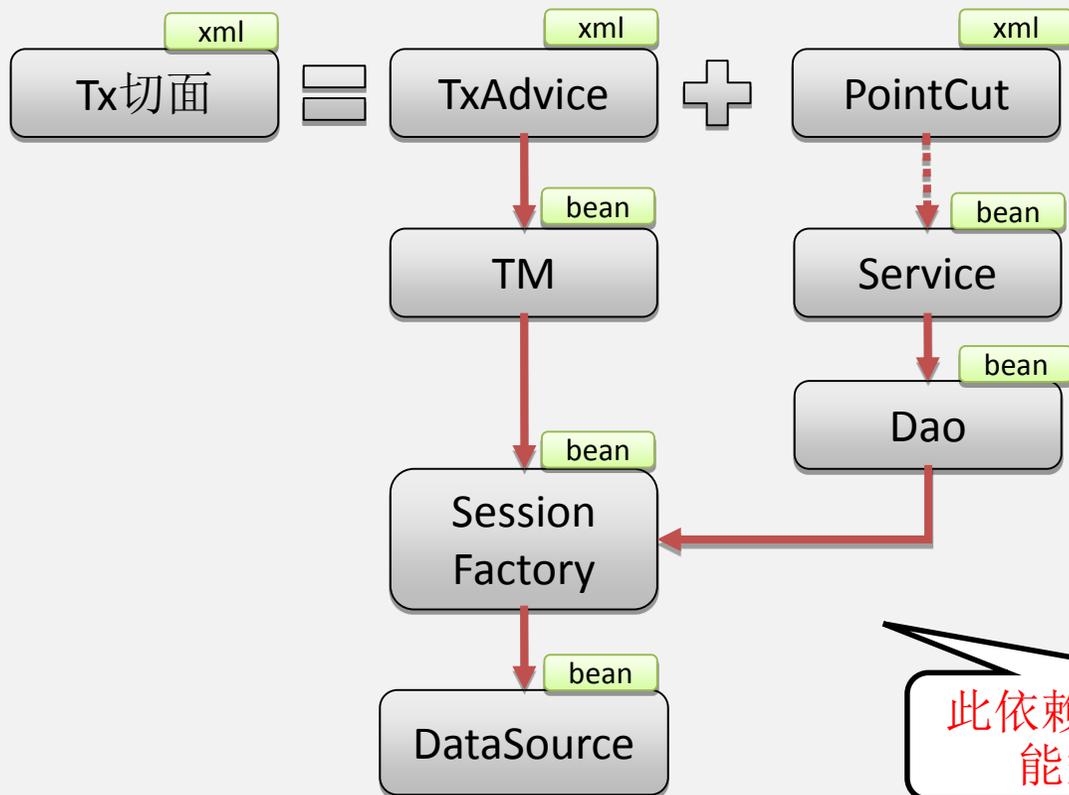
3. 前者中的 `<aop:aspect />` 标签, 被后者的 `<aop:advisor />` 标签替代。

```
<aop:config>
  <aop:aspect ref="myTxAdvice">
    <aop:around method="tx"
               pointcut="execu
```

# getCurrentSession()

openSession()和getCurrentSession()区别:

- 1)前者适用于编程式事务,后者适用于声明式事务;
- 2)前者每次调用,都返回一个新的session;  
后者调用,仅返回一个和当前线程捆绑的session;如果当前线程没有session,则抛错No Session found for current thread.
- 3)前者必须手动的调用close()方法,才能释放;  
后者在事务提交时,会自动释放.



此依赖树必须深刻记忆，  
能空手写出为止。

# 疑问 - 1

为什么LocalSessionFactoryBean  
可以返回org.hibernate.SessionFactory

因为实现了  
org.springframework.beans.factory.FactoryBean  
接口

```
public class LocalSessionFactoryBean extends H  
    implements FactoryBean<SessionFactory>
```

可以返回一个其泛型定义的  
org.hibernate.SessionFactory的实例，  
请详见本PPT中关于FactoryBean的说明



# 疑问 - 2

为什么要将 DataSource 和 SessionFactory 解耦合

因为 LocalSessionFactoryBean 声明依赖 javax.sql.DataSource 接口

```
public class LocalSessionFactoryBean extends  
    implements FactoryBean<SessionFactory>  
  
    private DataSource dataSource;  
  
    @ javax.sql.DataSource
```

因此可以根据开发需要，  
任意选择 DataSource 实现类  
和 SessionFactory 组合使用



# 疑问 - 3

删除了hibernate.cfg.xml文件后，如何使用  
SchemaExport工具类？

```
//& + beanId: 可以返回封装类对象本身
LocalSessionFactoryBean lsfb =
    (LocalSessionFactoryBean)ac.getBean("&sessionFactory");
Configuration cfg = lsfb.getConfiguration();
SchemaExport se = new SchemaExport(cfg);
// 第一个参:是否打印到控制台
// 第2个参:是否提交到数据库
se.create(true, true);
```

- 理解声明式事务的传播行为
- 了解隔离级别
- 能背出依赖树并熟练进行配置

把课堂例子的代码看明白，

运用到自己的项目中。

将之前基于Hibernate的编程式事务，

改为基于Spring的声明式事务。

# Q & A

