1.spring介绍

Spring是一个开放源代码的设计层面框架，他解决的是业务逻辑层和其他各层的松耦合问题，因此它将面向接口的编程思想贯穿整个系统应用。Spring是于2003 年兴起的一个轻量级的Java 开发框架，由Rod Johnson创建。简单来说，Spring是一个分层的JavaSE/EE **full-stack(一站式)** 轻量级开源框架。

1.1 spring 特点

1.方便解耦，简化开发

通过Spring提供的IoC容器，我们可以将对象之间的依赖关系交由Spring进行控制，避免硬编码所造成的过度程序耦合。有了Spring，用户不必再为单实例模式类、属性文件解析等这些很底层的需求编写代码，可以更专注于上层的应用。

2.AOP编程的支持

通过Spring提供的AOP功能，方便进行面向切面的编程，许多不容易用传统OOP实现的功能可以通过AOP轻松应付。

3.声明式事务的支持

在Spring中，我们可以从单调烦闷的事务管理代码中解脱出来，通过声明式方式灵活地进行事务的管理，提高开发效率和质量。

4.方便程序的测试

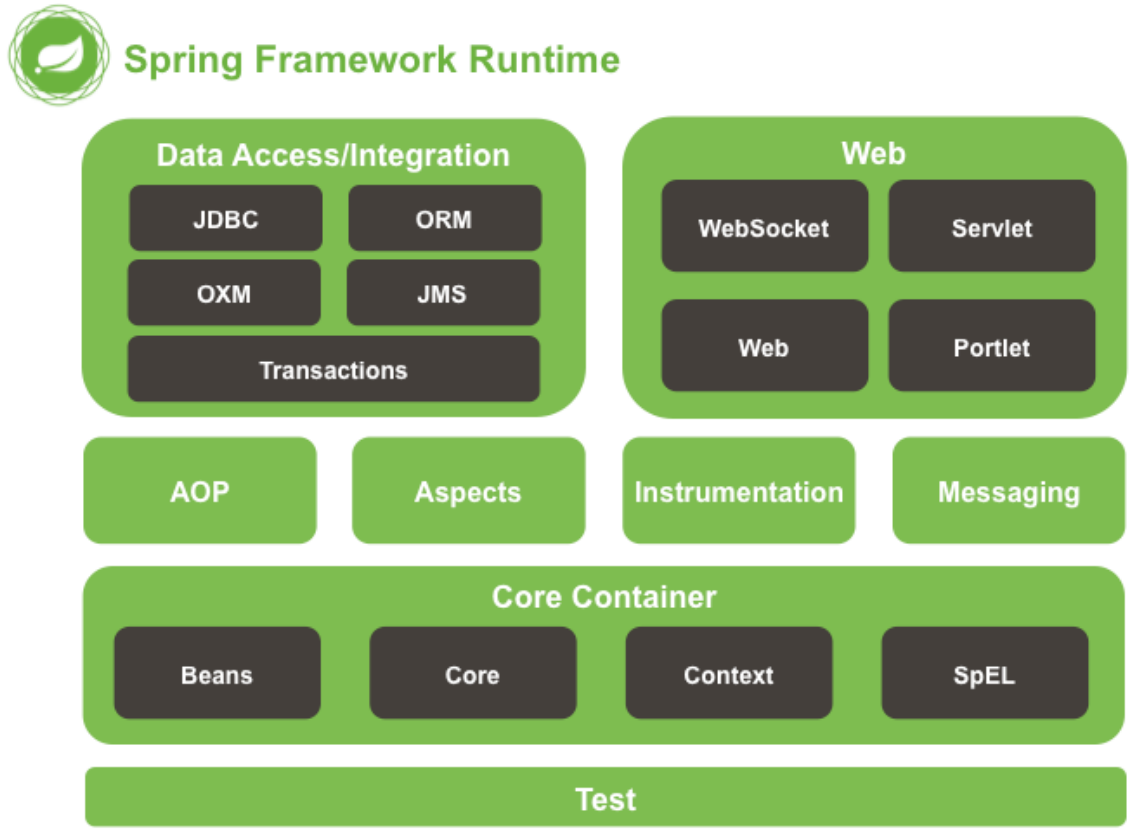
可以用非容器依赖的编程方式进行几乎所有的测试工作，在Spring里，测试不再是昂贵的操作，而是随手可做的事情。例如:Spring对Junit4支持，可以通过注解方便的测试Spring程序。

5.方便集成各种优秀框架

Spring不排斥各种优秀的开源框架，相反，Spring可以降低各种框架的使用难度，Spring提供了对各种优秀框架(如Struts,Hibernate、Hessian、Quartz)等的直接支持。

6.降低Java EE API的使用难度

1.2 spring组织架构



ORM- object relation mapping

OXM-Object xml mapping

JMS - Java消息服务（Java Message Service ，JMS）

WebSocket protocol 是HTML5一种新的协议。它实现了浏览器与服务器全双工通信(full-duplex)。一开始的握手需要借助HTTP请求完成。Socket是传输控制层协议，WebSocket是应用层协议。

Portlet是一种Web组件 - 就像servlets - 是专为将合成页面里的内容聚集在一起而设计的。通常请求一个portal页面会引发多个portlets被调用。每个portlet都会生成标记段，并与别的portlets生成的标记段组合在一起嵌入到portal页面的标记内

spring全家桶:spring,Spring Data、Spring MVC、Spring Boot、Spring Cloud(微服务)

1.3 spring下载

Spring官网:http://spring.io

Spring资源地址:http://repo.spring.io/release/org/springframework/spring

1.4 spring的核心模块

- spring-core：依赖注入IOC与DI的最基本实现

- spring-beans：Bean工厂与bean的装配

- spring-context：spring的context上下文即IoC容器

- spring-context-support

- spring-expression：spring表达式语言

2.spring中的IOC

IOC是 Inverse of Control 的简写，意思是控制反转。是降低对象之间的耦合关系的设计思想。

DI是Dependency Injection的缩写，意思是依赖注入,说的是创建对象实例时，同时为这个对象注入它所依赖的属性

2.1. 实现过程

步骤1:添加jar包

```
<!-- Spring的核心工具包-->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.0.8.RELEASE</version>
        </dependency>
<!--在基础IOC功能上提供扩展服务，还提供许多企业级服务的支持，有邮件服务、
任务调度、远程访问、缓存以及多种视图层框架的支持-->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.0.8.RELEASE</version>
        </dependency>

 <!-- Spring IOC的基础实现，包含访问配置文件、创建和管理bean等 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
            <version>5.0.8.RELEASE</version>
        </dependency>

        <!-- Spring context的扩展支持，用于MVC方面 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context-support</artifactId>
            <version>5.0.8.RELEASE</version>
        </dependency>
        <!-- Spring表达式语言 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-expression</artifactId>
            <version>5.0.8.RELEASE</version>
        </dependency>
```

步骤2:创建配置文件applicationContext.xml

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

步骤3:在配置文件中创建对象

```
<bean id="对象名" class="类的完整路径">
    <property name="属性名" ref="对象的id值"></property>
</bean>
```

步骤4:加载配置文件，获得对象

```
ApplicationContext app=new ClassPathXmlApplicationContext("spring.xml");
 Users users=(Users)app.getBean("u1");
```

2.2 bean标签的属性介绍

属性	说明
class	指定bean对应类的全路径
name	name是bean对应对象的一个标识
scope	执行bean对象创建模式和生命周期,scope="singleton"和scope="prototype"
id	id是bean对象的唯一标识,不能添加特别字符
lazy-init	是否延时加载 默认值:false。true 延迟加载对象,当对象被调用的时候才会加载，测试的时候，通过getbean()方法获得对象。lazy-init="false" 默认值，不延迟，无论对象是否被使用，都会立即创建对象,测试时只需要加载配置文件即可。注意:测试的时候只留下id,class属性
init-method	只需要加载配置文件即可对象初始化方法
destroy-method	对象销毁方法

2.3 对象创建的方式

(1)无参构造

(2)有参构造

```
public  Person(String  name , Car car){
        this.name = name;
        this.car = car;
        System.out.println("Person的有参构造方法:"+name+car);
     }
   <bean name="person" class="com.xzk.spring.bean.Person">
           <constructor-arg name="name" value="rose"/>
           <constructor-arg name="car"  ref="car"/>
   </bean>
```

(3)静态方法创建对象

//静态工厂模式

```
public class PersonFactory {
    public static Person  createPerson(){
        System.out.println("静态工厂创建Person");
        return  new Person();
    }
}
```

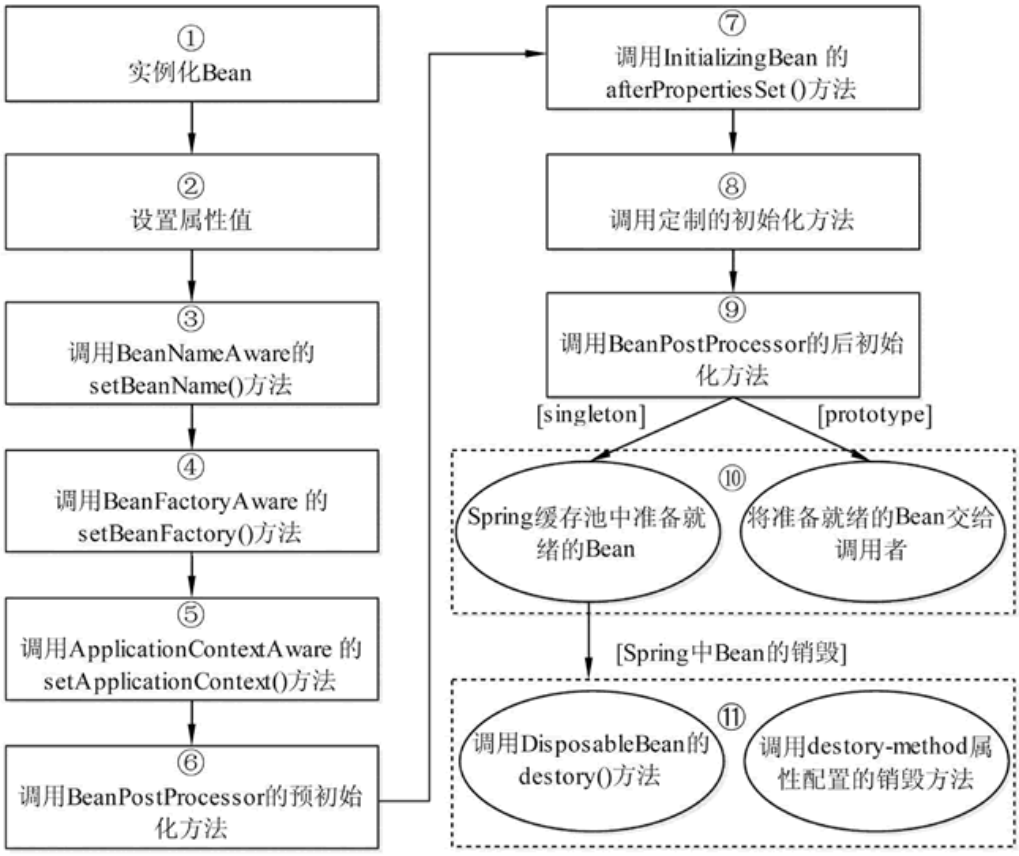
```
<bean name="pf" class="com.xzk.PersonFactory" factory-method="createPerson" />
```

(4)非静态工厂方法

```
public class Users{
   public  Person createPerson1(){
    System.out.println("非静态工厂创建Person");
    return new Person();
   }
}
```

```
<bean id="u2"  class="com.bean.Users"></bean>
<bean id="u3" factory-method="createPerson1" factory-bean="u2"></bean>
```

2.4 springBean的生命周期



Bean 生命周期的整个执行过程描述如下

1) 根据配置情况调用 Bean 构造方法或工厂方法实例化 Bean。

2) 利用依赖注入完成 Bean 中所有属性值的配置注入。

3) 如果 Bean 实现了 BeanNameAware 接口，则 Spring 调用 Bean 的 setBeanName() 方法传入当前 Bean 的 id 值。

4) 如果 Bean 实现了 BeanFactoryAware 接口，则 Spring 调用 setBeanFactory() 方法传入当前工厂实例的引用。

5) 如果 Bean 实现了 ApplicationContextAware 接口，则 Spring 调用 setApplicationContext() 方法传入当前 ApplicationContext 实例的引用。

6) 如果 BeanPostProcessor 和 Bean 关联，则 Spring 将调用该接口的预初始化方法 postProcessBeforeInitialzation() 对 Bean 进行加工操作，此处非常重要，Spring 的 AOP 就是利用它实现的。

7) 如果 Bean 实现了 InitializingBean 接口，则 Spring 将调用 afterPropertiesSet() 方法。初始化bean的时候执行，可以针对某个具体的bean进行配置。afterPropertiesSet 必须实现 InitializingBean 接口。实现 InitializingBean接口必须实现afterPropertiesSet方法。

8) 如果在配置文件中通过 init-method 属性指定了初始化方法，则调用该初始化方法。

9) 如果 BeanPostProcessor 和 Bean 关联，则 Spring 将调用该接口的初始化方法 postProcessAfterInitialization()。此时，Bean 已经可以被应用系统使用了。

10) 如果在 中指定了该 Bean 的作用范围为 scope="singleton"，则将该 Bean 放入 Spring IoC 的缓存池中，将触发 Spring 对该 Bean 的生命周期管理；如果在 中指定了该 Bean 的作用范围为 scope="prototype"，则将该 Bean 交给调用者，调用者管理该 Bean 的生命周期，Spring 不再管理该 Bean。

11) 如果 Bean 实现了 DisposableBean 接口，则 Spring 会调用 destory() 方法将 Spring 中的 Bean 销毁；如果在配置文件中通过 destory-method 属性指定了 Bean 的销毁方法，则 Spring 将调用该方法对 Bean 进行销毁。

3.DI注入值

分类:一种是调取属性的set方法赋值，第二种使用构造方法赋值

3.1 set注入值

3.1.1 基本属性类型值注入

```
<property name="name" value="jeck" />
```

3.1.2 引用属性类型值注入

```
<property name="car" ref="car"></property>
```

3.2 构造注入:

3.2.1 可以通过name属性，按照参数名赋值

```
public  Person(String  name , Car car){
        this.name = name;
        this.car = car;
        System.out.println("Person的有参构造方法:"+name+car);
     }
   <bean name="person" class="com.xzk.spring.bean.Person">
           <constructor-arg name="name" value="rose"/>
           <constructor-arg name="car"  ref="car"/>
   </bean>
```

3.2.2 可以通过index属性，按照参数索引注入

```
<bean name="person2" class="com.xzk.spring.bean.Person">
        <constructor-arg name="name" value="helen" index="0"></constructor-arg>
        <constructor-arg name="car" ref="car" index="1"></constructor-arg>
</bean>
```

3.2.3 使用type注入

```
 public Person(Car car, String name) {
        super();
        System.out.println("Person(Car car, String name)");
        this.name = name;
        this.car = car;
    }

    public Person(Car car, Integer name) {
        super();
        System.out.println("Person(Car car, Integer name)");
        this.name = name + "";
        this.car = car;
    }
 <bean name="person2" class="com.xzk.spring.bean.Person">
     <constructor-arg name="name" value="988" type="java.lang.Integer">
</constructor-arg>
     <constructor-arg name="car" ref="car" ></constructor-arg>
  </bean>
```

3.3 spel spring表达式

```
<bean name="car"  class="com.xzk.spring.bean.Car" >
           <property name="name" value="mime" />
           <property name="color" value="白色"/>
  </bean>
   <!--利用spel引入car的属性 -->
   <bean  name="person1"  class="com.xzk.spring.bean.Person"  p:car-ref="car">
            <property name="name" value="#{car.name}"/>
            <property name="age" value="#{person.age}"/>
   </bean>
```

3.4 p命名空间注入值

使用p:属性名 完成注入,走set方法

- 基本类型值: p:属性名="值"

- 引用类型值: P:属性名-ref="bean名称"

实现步骤:配置文件中 添加命名空间p

```
xmlns:p="http://www.springframework.org/schema/p"
```

实例:

```
<bean id="u6" class="com.entity.Users" p:age="30" p:name="李四" p:student-
ref="stu1"></bean>
```

3.5 复杂类型注入

Object[],list,set,map,java.util.Properties

```
    <!-- 数组变量注入 -->
            <property name="arrs">
                <list>
                    <value>数组1</value>
                    <!--引入其他类型-->
                    <ref bean="car"/>
                </list>
            </property>

            <!-- 集合变量赋值-->
            <property name="list">
                 <list>
                     <value>集合1</value>
                     <!--集合变量内部包含集合-->
                     <list>
                         <value>集合中的集合1</value>
                         <value>集合中的集合2</value>
                         <value>集合中的集合3</value>
                     </list>
                     <ref bean="car" />
                 </list>
            </property>

            <!--map赋值 -->
            <property name="map">
                <map>
                    <entry key="car" value-ref="car" />
                    <entry key="name" value="保时捷" />
                    <entry key="age"  value="11"/>
                </map>

            </property>

            <!-- properties赋值 -->
            <property name="properties">
                 <props>
                     <prop key="name">pro1</prop>
                     <prop key="age">111</prop>
                 </props>
            </property>
```

3.6 自动注入(由程序自动给属性赋值)

autowire：

no 不自动装配(默认值)

byName 属性名=id名 ，调取set方法赋值

byType 属性的类型和id对象的类型相同，当找到多个同类型的对象时报错，调取set方法赋值

constructor 构造方法的参数类型和id对象的类型相同,当没有找到时，报错。调取构造方法赋值

示例:

```
<bean id="service" class="service.impl.UserServiceImpl" autowire="constructor">
</bean>
```

配置全局自动装配:

```
<beans   default-autowire="constructor/byName/byType/no">
```

4.注解实现IOC

（1）配置文件中添加约束

参考文件位置:

spring-framework-5.0.8.RELEASE\docs\spring-framework-reference\html\xsd-configuration.html

```
xmlns:context="http://www.springframework.org/schema/context"
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
```

(2)配置注解扫描:指定扫描包下所有类中的注解,扫描包时,会扫描包所有的子孙包

```
<!--扫描包设置-->
<context:component-scan base-package="com.xzk.spring.bean"></context:component-scan>
```

(3)注解

3.1 添加在类名上

```
@Component("对象名")
@Service("person")  // service层
@Controller("person")   // controller层
@Repository("person")   // dao层
@Scope(scopeName="singleton")  //单例对象
@Scope(scopeName="prototype") //多例对象
```

3.2 添加在属性上:

```
@Value("属性值")
private String name;


@Autowired //如果一个接口类型，同时有两个实现类，则报错，此时可以借助@Qualifier("bean
name")
@Qualifier("bean name")
private Car car;


//说明:@Resource 是java的注释,但是Spring框架支持,@Resource指定注入哪个名称的对象
//@Resource(name="对象名") == @Autowired  + @Qualifier("name")
@Resource(name="baoma")
private Car car;
```

3.3 添加在方法上

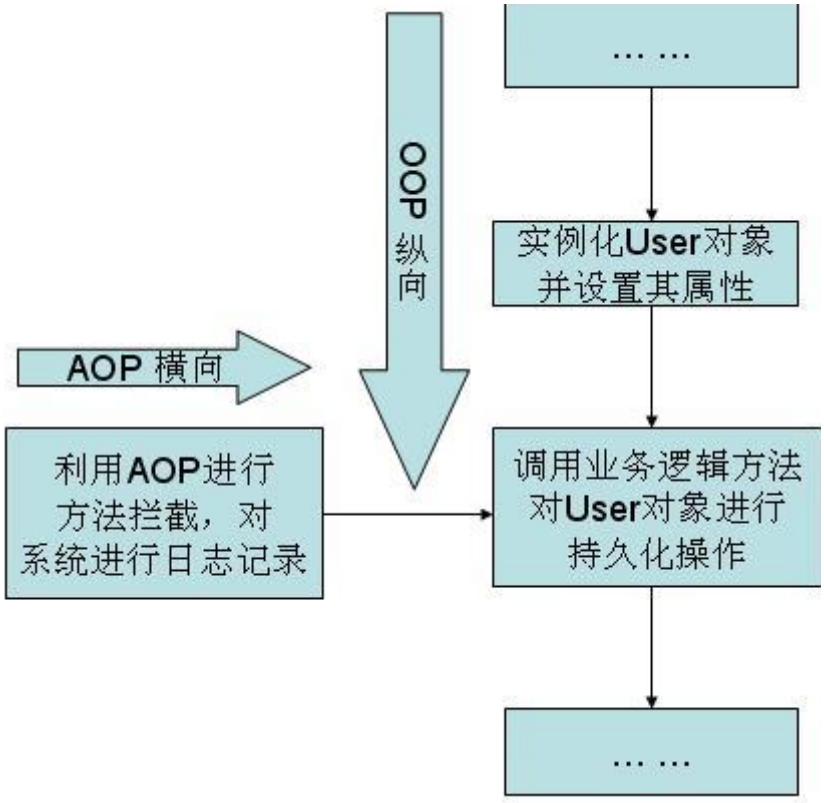
```
@PostConstruct    //等价于init-method属性
 public  void init(){
   System.out.println("初始化方法");
}

 @PreDestroy  //等价于destroy-method属性
 public void destroy(){
    System.out.println("销毁方法");
 }
```

5.Aop介绍

AOP（Aspect Oriented Programming)即面向切面编程。即在不改变原程序的基础上为代码段增加新的功能。应用在权限认证、日志、事务。

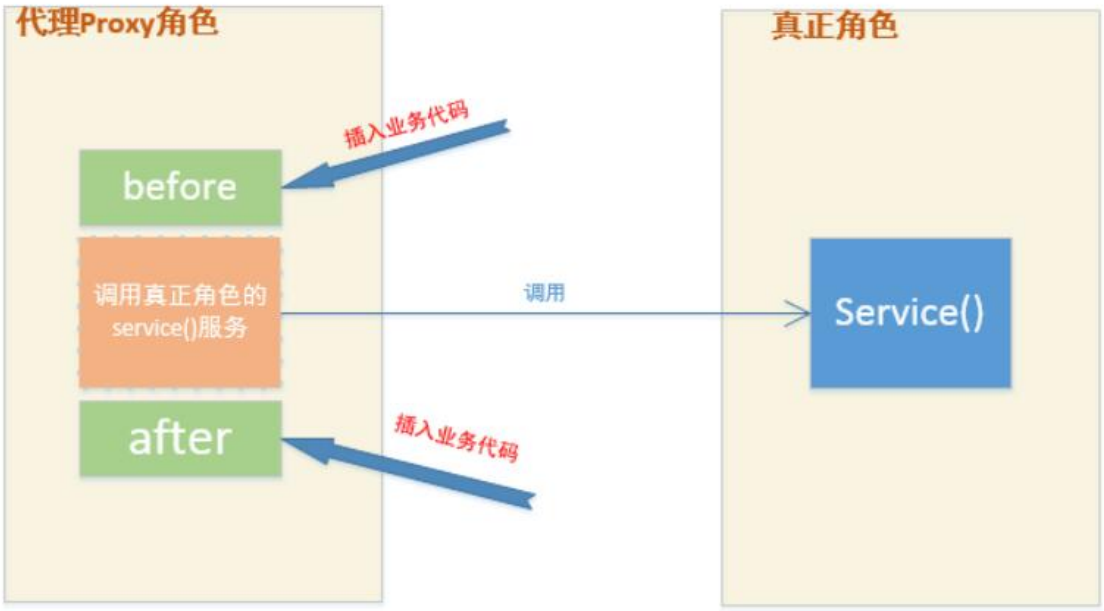
AOP的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。



6.AOP的实现机制

- JDK 的动态代理：针对实现了接口的类产生代理。InvocationHandler接口

- CGlib 的动态代理：针对没有实现接口的类产生代理，应用的是底层的字节码增强的技术 生成当前类的子类对象，MethodInterceptor接口



6.1 JDK动态代理实现

1. 创建接口和对应实现类

```
public interface UserService {
    public void login();
}
```

//实现类

```
public class UserServiceImpl implements  UserService {
    public void login(){}
}
```

2. 创建动态代理类，实现InvocationHandler接口

```
public class agency implements InvocationHandler {
    private  UserService target; //目标对象
    public  agency(UserService target){
        this.target = target;
    }
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        //本方法中的其他输出输入增强
        //proxy    代理方法被调用的代理实例
        System.out.println("方法触发了");
        //执行被代理类 原方法
        Object invoke = method.invoke(target, args);
        System.out.println("执行完毕了");
        return invoke;
    }
}
```

测试

```
 @Test
 public  void  test1(){
     //测试JDK动态代理技术
      UserService us = new UserServiceImpl();
      agency ag = new agency(us);
//这里不能转换成一个实际的类，必须是接口类型
    UserService uservice = (UserService)
Proxy.newProxyInstance(us.getClass().getClassLoader(),
us.getClass().getInterfaces(),ag);
      uservice.login();
 }
```

测试结果: 在调用接口方法的前后都会添加代理类的方法!

2.2 CGlib实现代理

> 使用JDK创建代理有一个限制,它只能为接口创建代理实例.这一点可以从Proxy的接口方法newProxyInstance(ClassLoader loader,Class [] interfaces,InvocarionHandler h)中看的很清楚

> 第二个入参 interfaces就是需要代理实例实现的接口列表.

> 对于没有通过接口定义业务方法的类,如何动态创建代理实例呢? JDK动态代理技术显然已经黔驴技穷,CGLib作为一个替代者,填补了这一空缺.

> CGLib采用底层的字节码技术,可以为一个类创建子类,在子类中采用方法拦截的技术拦截所有父类方法的调用并顺势织入横切逻辑.

添加依赖包:

```
 <dependency>
    <groupId>cglib</groupId>
    <artifactId>cglib</artifactId>
    <version>3.2.5</version>
</dependency>
```

1.创建普通类

```
  public  class Users{
    public void login(){}
  }
```

2. 创建创建CGLib代理器

```
 class CgProxy implements MethodInterceptor {
    public Object intercept(Object o, Method method, Object[] objects,
MethodProxy methodProxy) throws Throwable {
      System.out.println("输出语句1");
//参数：Object为由CGLib动态生成的代理类实例，Method为上文中实体类所调用的被代理的方法
//引用，Object[]为参数值列表，MethodProxy为生成的代理类对方法的代理引用。
       Object obj=   methodProxy.invokeSuper(o,objects);
      System.out.println("输出语句2");
            return obj;
    }
}
```

测试

```
    public static void main(String[] args) {
        //1.创建真实对象
        Users users = new Users();
        //2.创建代理对象
       Enhancer enhancer = new Enhancer();
       enhancer.setSuperclass(users.getClass());
       enhancer.setCallback(new CglibProxy());
       Users o = (Users) enhancer.create();//代理对象
       o.login();
    }
```

结论:spring同时使用了这两种方式，底层会自行判断应该使用哪种

两种代理方式的区别:

1、jdk动态代理生成的代理类和委托类实现了相同的接口；

2、cglib动态代理中生成的字节码更加复杂，生成的代理类是委托类的子类，且不能处理被final关键字修饰的方法；

3、jdk采用反射机制调用委托类的方法，cglib采用类似索引的方式直接调用委托类方法；

7.spring中使用aop

(1)添加jar包

```
  <dependency>
```

```
            <groupId>aopalliance</groupId>
            <artifactId>aopalliance</artifactId>
            <version>1.0</version>
        </dependency>

        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.8.13</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aspects</artifactId>
            <version>5.0.8.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>5.0.8.RELEASE</version>
        </dependency>

         <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.0.8.RELEASE</version>
        </dependency>

         <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.0.8.RELEASE</version>
        </dependency>
         <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
            <version>5.0.8.RELEASE</version>
        </dependency>

         <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context-support</artifactId>
            <version>5.0.8.RELEASE</version>
        </dependency>
```

(2)添加项目原有的调取过程

(3)创建增强类(本质上就是一个普通类)

//前置通知：目标方法运行之前调用 aop:before

//后置通知(如果出现异常不会调用)：在目标方法运行之后调用 aop:after-returning

//环绕通知：在目标方法之前和之后都调用 aop:around

//最终通知(无论是否出现 异常都会调用)：在目标方法运行之后调用 aop:after

//异常增强:程序出现异常时执行(要求：程序代码中不要处理异常) aop:after-throwing

环绕增强:

```
public  Object around(ProceedingJoinPoint point) throws Throwable{
         point.proceed();
}
```

(4)添加aop命名空间

```
   xmlns:aop="http://www.springframework.org/schema/aop"
   http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop-4.2.xsd
```

(5)设置配置文件

```
   <!--1.创建目标类对象-->
    <bean name="userService" class="com.xzk.spring.service.UserServiceImpl" />
    <!--2.配置增强类对象-->
    <bean name="myAdvice" class="com.xzk.spring.aop.MyAdivce" />
    <!-- 3.配置将增强织入目标对象-->
    <aop:config>
        <aop:pointcut id="pc"
            expression="execution(* com.xzk.spring.service.ServiceImpl.*.*
(..))"/>
        <aop:aspect ref="myAdvice">
                <aop:before method="before" pointcut-ref="pc" />
                <aop:after-returning method="afterReturning" pointcut-ref="pc"
/>
                <aop:around method="around" pointcut-ref="pc" />
                <aop:after-throwing method="afterException" pointcut-ref="pc"
/>
                <aop:after method="after" pointcut-ref="pc" />
        </aop:aspect>
    </aop:config>
```

注意:(1)环绕增强需要使用ProceedingJoinPoint 作为参数(2)注意标签顺序

8.切入点方法的定义

表达式匹配规则举例：

public * addUser(com.pb.entity.User)：“*”表示匹配所有类型的返回值。

示例:

```
    public int addUser(User u);
    public String addUser(User u);
```

public void *(com.pb.entity.User)：“*”表示匹配所有方法名。

示例:

```
    public void selectUser(User u);
    public void a(User u);
```

public void addUser (..)：“..”表示匹配所有参数个数和类型。

示例:

```
public void addUser(int a)
public void addUser(int b,int c)
```

* com.pb.service.*.*(..)：匹配com.pb.service 包下所有类的所有方法。

示例：

```
public  void com.pb.service.A.a();
public  String  com.pb.service.B.a();
```

* com.pb.service..*(..)：匹配com.pb.service 包及子包下所有类的所有方法

9.如何获取切入点信息

通过JoinPoint对象获取信息:

```
System.out.println("切入点对象:"+jp.getTarget().getClass().getSimpleName());
System.out.println("切入点方法：+jp.getSignature());
System.out.println("切入点的参数：+jp.getArgs()[0]);
```

10.特殊的前置增强-->Advisor前置增强实现步骤

1.创建增强类,要求该类实现MethodBeforeAdvice接口

2.修改applicationContext.xml文件

(1)创建增强类对象

(2)定义增强和切入点的关系:

```
<aop:config>
          <!-- 表达式是被切入的方法的表达式 -->
         <aop:pointcut expression="execution(* biz.impl.*.*(..))"
id="mypoint"/>
         <aop:advisor advice-ref="增强类对象的id" pointcut-ref="切入点对象的id"/>
</aop:config>
```

11.使用了AspectJ依赖注解开发

spring AOP的注解方式：

注:1 ）增强类也需要创建对象(使用@Component)

2)要启动扫描spring注解包的代码:

```
<context:component-scan base-package="com.xzk"></context:component-scan>
```

<1> 除了启动spring的注解之外 还要启动aspectJ的注解方式

```
<aop:aspectj-autoproxy/>
```

<2> 在切面类(增强类)上添加：@Aspect

<3> 定义一个任意方法

```
@Pointcut("execution(* com.*.*(..))")
public void anyMethod(){}
```

为什么要定义一个任意方法？？？ 因为@Pointcut 要求要放在一个方法

<5>用法：

```
@Pointcut("execution(* com.*.*(..))")
    public void anyMethod(){}

    @Before("anyMethod()")
    public void log(JoinPoint){
    System.out.println("myAspect....log....before");
    }


    @Around("anyMethod()")
    public void aroundTest(ProceedingJoinPoint pjp){
        System.out.println("around...before....");

          try {
                  pjp.proceed();//执行目标方法
          } catch (Throwable e) {

               e.printStackTrace();
          }

          System.out.println("around...after....");
    }
```

//注解方式中注解的顺序问题

1.没有异常情况下

环绕开始。。。。

前置增强开始执行

insert----------

环绕结束。。。。

最终增强

后置增强开始执行

相对顺序固定，注解换位置时不影响结果顺序

2.有异常

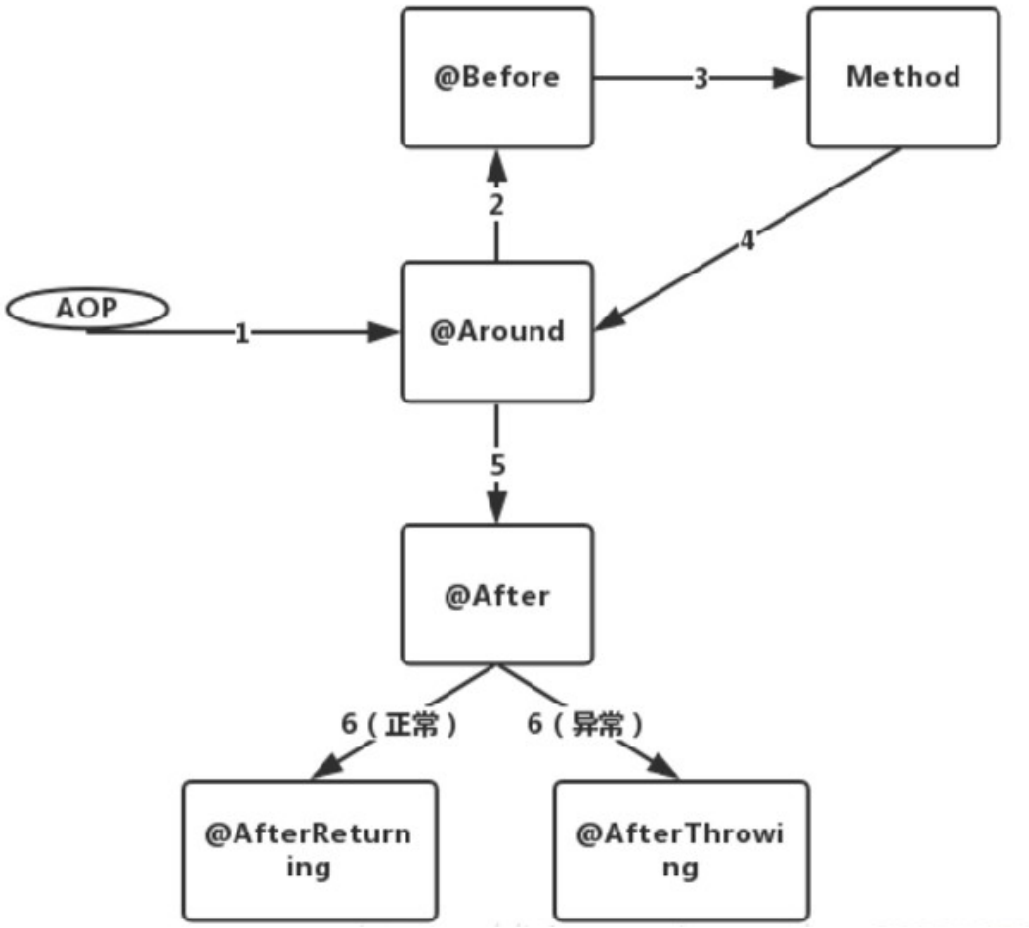
前置增强开始执行

insert----------

最终增强

异常增强

注意：不要使用环绕增强，使用的话，异常增强不执行



aop的应用场景:事务底层实现，日志，权限控制,mybatis中sql绑定,性能检测

12.Spring-JDBC 数据访问

12.1 使用spring-jdbc操作数据库

主要内容:学习使用JdbcTemplate API和 如何使用Spring管理 JdbcTemplate

步骤1. 引入jar包

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.0.8.RELEASE</version>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.46</version>
</dependency>
```

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>5.0.8.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.0.8.RELEASE</version>
</dependency>
<dependency>
    <groupId>com.mchange</groupId>
    <artifactId>c3p0</artifactId>
    <version>0.9.5.2</version>
</dependency>
```

2.测试

```
 public  void test1() throws Exception {
        //TODO 测试jdbcTemplate简单使用
        //1.创建c3p0链接池
        ComboPooledDataSource dataSource  = new ComboPooledDataSource();
        dataSource.setDriverClass("com.mysql.jdbc.Driver");
        dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/springjdbc");
        dataSource.setUser("root");
        dataSource.setPassword("111");

        //创建jdbcTemplate对象
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        //创建sql语句
        String sql = "insert into role (rid , rname ,alias) value (? , ?,?);";
        jdbcTemplate.update(sql,"3","visitor","游客");
 }
```

12.2. Spring管理JdbcTemplate

注意: 可以自己在RoleDaoImpl中添加 JdbcTemplate变量,如果不自动装载记得添加变量的set方法,

标准的操作,我们可以让RoleDaoImpl 继承 JdbcDaoSupport

示例：

```
public class RoleDaoImpl extends JdbcDaoSupport implements RoleDao {
    public void save(Role role) {
       String sql = "INSERT INTO role (rname,alias) value (?,?) ;";
       getJdbcTemplate().update(sql,role.getRname(),role.getAlias());
    }
}
```

配置文件:需要创建数据源和给RoleDaoImpl中的jdbcTemplate赋值

```
<bean name="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
              p:jdbcUrl="jdbc:mysql://localhost:3306/xzk"
              p:driverClass="com.mysql.jdbc.Driver"
              p:user="root"
              p:password="111"
        />
```

```
        <!-- bean jdbcTemplate -->
        <bean name="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
            <property name="dataSource" ref="dataSource"/>
        </bean>
       <bean name="roleDao" class="com.xzk.spring.dao.impl.RoleDaoImpl">
            <property name="jdbcTemplate" ref="jdbcTemplate"/>
        </bean>
```

12.3 创建db.properties属性文件

修改配置文件:

```
<context:property-placeholder  location="db.properties">
</context:property-placeholder>
```

```
p:jdbcUrl="${jdbc.jdbcUrl}"
p:driverClass="${jdbc.driverClass}"
```

关于属性文件username的问题:

解决方式:

```
<context:property-placeholder location="db.properties"
 system-properties-mode="FALLBACK"/>
```

FALLBACK --- 默认值，不存在时覆盖

NEVER --- 不覆盖

12.4 crud

JdbcTemplate常用方法:

JdbcTemplate.update(sql,ArgsObj....); //多个参数值时，可以使用对象数组

//DQL 查询单个

jdbcTemplate.queryForObject(String var1, RowMapper var2, Object... var3);

RowWapper 将结果封装的处理器; 得到Result解析成实体类对象即可!

jdbcTemplate.query(String var1, RowMapper var2, Object... var3); //查询所有

getJdbcTemplate().query(sql,new BeanPropertyRowMapper<类名>(类名.class));//查询(单行+多行)

实例:

处理查询结果的自定义方法(也可以自定义一个类，该类实现RowMapper接口的方式处理结果)

```
private Users chuliresult(ResultSet resultSet){
         Users users=new Users();
        try {
            users.setUsername(resultSet.getString("username"));
            users.setPassword(resultSet.getString("password"));
            users.setUserid(resultSet.getInt("userid"));
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return users;
    }
```

单行查询

```
 public Users findbyid(int uid) {
        String sql="select * from users where userid=?";
        Users users=
                getJdbcTemplate().queryForObject(sql, new Object[]{uid}, new
RowMapper<Users>() {
                    public Users mapRow(ResultSet resultSet, int i) throws
SQLException {
                        return chuliresult(resultSet);
                    }
                });
        return users;
    }
```

多行查询

```
 public List findall() {
        String  sql="select * from users";
        List<Users> list=getJdbcTemplate().query(sql, new RowMapper<Users>() {
            public Users mapRow(ResultSet resultSet, int i) throws SQLException
{
               // System.out.println(resultSet+",,,"+i);
                return chuliresult(resultSet);
            }
        });
        return list;
    }
```

实例：查询行数

```
String sql="select count(*) from users";
Integer integer = getJdbcTemplate().queryForObject(sql, Integer.class);
```

查询多列

```
Map<String, Object> map =
        jdbcTemplate.queryForMap("select count(*),max(roleid),min(roleid) from
role");
Set<Map.Entry<String, Object>> entrySet = map.entrySet();
Iterator<Map.Entry<String, Object>> iterator = entrySet.iterator();
while(iterator.hasNext()){
    System.out.println(iterator.next());
}
```

查询单列值

```
String sql="select username from users";
List<String> list = getJdbcTemplate().queryForList(sql, String.class);
```

13.Spring事务管理

13.1 什么是事务(Transaction)

通过sql将逻辑相关的一组操作绑定在一起，以便服务器 保持数据的完整性(准确性)。

事务通常是以begin transaction开始，以commit或rollback结束。

事务执行的流程:开启事务->执行insert,update,delete->commit/rollback

设想网上购物的一次交易，其付款过程至少包括以下几步数据库操作：

1、更新客户所购商品的库存信息

2、保存客户付款信息--可能包括与银行系统的交互

3、生成订单并且保存到数据库中

4、更新用户相关信息，例如购物数量等等

数据库事务正是用来保证这种情况下交易的平稳性和可预测性的技术

13.2 为什么要使用事务？

(1)为了提高性能

(2)为了保持业务流程的完整性

(3)使用分布式事务

13.3 事务的特性ACID

1 - 原子性（atomicity）

事务是数据库的逻辑工作单位，而且是必须是原子工作单位，对于其数据修改，要么全部执行，要么全部不执行。

2、一致性（consistency）

事务在完成时，必须是所有的数据都保持一致状态。在相关数据库中，所有规则都必须应用于事务的修改，以保持所有数据的完整性。

3、隔离性（isolation）

一个事务的执行不能被其他事务所影响。企业级的数据库每一秒钟都可能应付成千上万的并发访问，因而带来了并发控制的问题。

4、持久性（durability）

一个事务一旦提交，事务的操作便永久性的保存在DB中。即使此时再执行回滚操作也不能撤消所做的更改

13.4 事务的嵌套->传播行为propagation

事务的传播机制

事务的第一个方面是传播行为（propagation　/,prɒpə'geɪʃən/　behavior）。

当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。

规定了事务方法和事务方法发生嵌套调用时事务如何进行传播

例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。

Spring定义了七种传播行为：

事务传播行为类型	说明
PROPAGATION_REQUIRED	如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。这是最常见的选择。
PROPAGATION_SUPPORTS	支持当前事务，如果当前没有事务，就以非事务方式执行。
PROPAGATION_MANDATORY	使用当前的事务，如果当前没有事务，就抛出异常。
PROPAGATION_REQUIRES_NEW	新建事务，如果当前存在事务，把当前事务挂起。
PROPAGATION_NOT_SUPPORTED	以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
PROPAGATION_NEVER	以非事务方式执行，如果当前存在事务，则抛出异常。
PROPAGATION_NESTED	如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION_REQUIRED类似的操作。

propagtion_nested:

嵌套的事务可以独立于当前事务进行单独地提交或回滚。

如果当前事务不存在，那么其行为与PROPAGATION_REQUIRED一样。

注意各厂商对这种传播行为的支持是有所差异的。可以参考资源管理器的文档来确认它们是否支持嵌套事务

事务传播行为失效的情况

spring事务是基于代理来实现的，所以：

（1）private、final、static 方法无法被代理，所以添加事务无效

（2）当绕过代理对象, 直接调用添加事务管理的方法时, 事务管理将无法生效。比如直接new出的对象。

（3）在同一个类下，有2个方法，A、B，A没有事务，B有事务，但是A调用B时，方法B被标记的事务无效。 究其原因，因为此类的调用对象为代理对象，代理方法A调用真正的被代理方法A后，在被代理方法A中才会去调用方法B，此时this对象为被代理的对象，所以是不会通知到代理对象，也就变成了第二种情况，绕过了代理对象。所以无效

13.5 事务的隔离级别

MySQL数据库共定义了四种隔离级别：

1. Serializable(串行化)：可避免脏读、不可重复读,幻读情况的发生。
2. Repeatable read(可重复读)：可避免脏读、不可重复读情况的发生。
3. Read committed(读已提交)：可避免脏读情况发生。
4. Read uncommitted(读未提交)：最低级别，以上情况均无法保证。

Isolation.DEFAULT:为数据源的默认隔离级别

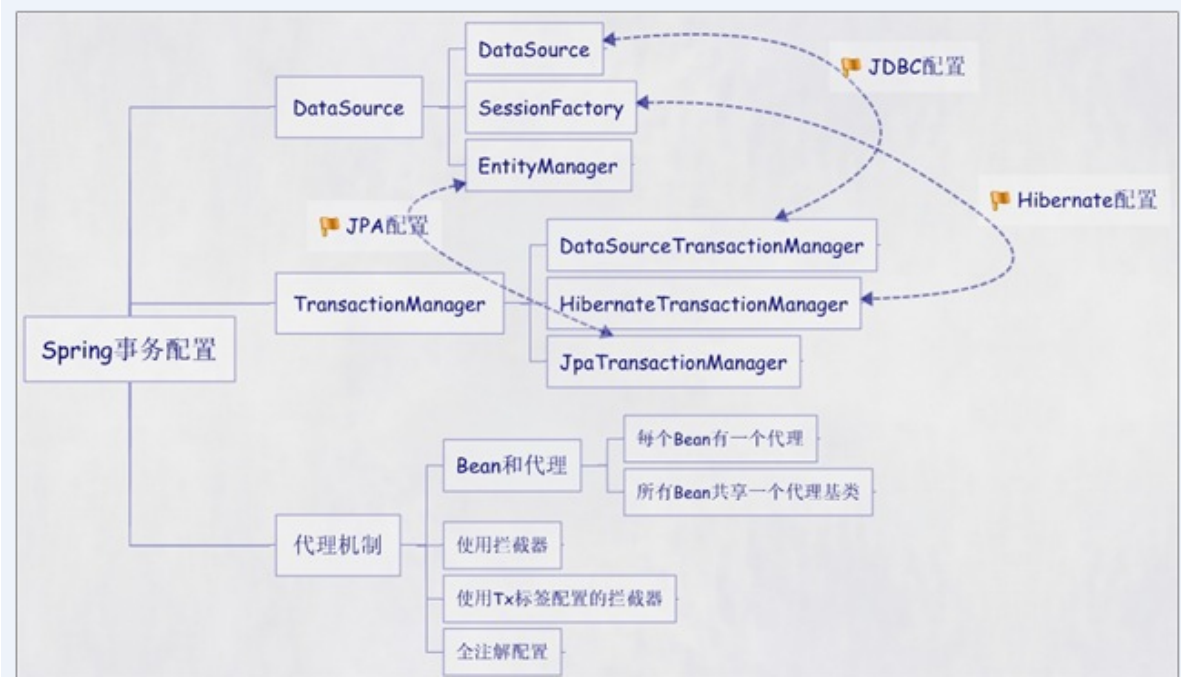
√: 可能出现　×: 不会出现

	脏读	不可重复读	幻读
Read uncommitted	√	√	√
Read committed	×	√	√
Repeatable read	×	×	√
Serializable	×	×	×

级别越高，数据越安全，但性能越低。

13.6 事务的实现

Spring XML配置声明事务



13.6.1. TransactionManager

在不同平台，操作事务的代码各不相同，因此spring提供了一个 TransactionManager 接口：

- DateSourceTransactionManager 用于 JDBC 的事务管理

- HibernateTransactionManager 用于 Hibernate 的事务管理

13.6.2 接口的定义

事务的属性介绍：这里定义了传播行为、隔离级别、超时时间、是否只读

```
package org.springframework.transaction;
public interface TransactionDefinition {
```

```
    int PROPAGATION_REQUIRED = 0; //支持当前事务，如果不存在，就新建一个
    int PROPAGATION_SUPPORTS = 1; //支持当前事务，如果不存在，就不使用事务
    int PROPAGATION_MANDATORY = 2; //支持当前事务，如果不存在，就抛出异常
    int PROPAGATION_REQUIRES_NEW = 3;//如果有事务存在，挂起当前事务，创建一个新的事物
    int PROPAGATION_NOT_SUPPORTED = 4;//以非事务方式运行，如果有事务存在，挂起当前事务
    int PROPAGATION_NEVER = 5;//以非事务方式运行，如果有事务存在，就抛出异常
    int PROPAGATION_NESTED = 6;//如果有事务存在，则嵌套事务执行

    int ISOLATION_DEFAULT = -1;//默认级别，MYSQL: 默认为REPEATABLE_READ级别
SQLSERVER: 默认为READ_COMMITTED
    int ISOLATION_READ_UNCOMMITTED = 1;//读取未提交数据(会出现脏读, 不可重复读) 基本不
使用
    int ISOLATION_READ_COMMITTED = 2;//读取已提交数据(会出现不可重复读和幻读)
    int ISOLATION_REPEATABLE_READ = 4;//可重复读(会出现幻读)
    int ISOLATION_SERIALIZABLE = 8;//串行化

    int TIMEOUT_DEFAULT = -1;//默认是-1，不超时，单位是秒

    //事务的传播行为
    int getPropagationBehavior();
    //事务的隔离级别
    int getIsolationLevel();
    //事务超时时间
    int getTimeout();
    //是否只读
    boolean isReadOnly();
    String getName();
}
```

13.6.3 添加tx命名空间

13.6.4 添加事务相关配置

修改applicationContext.xml

```
<!-- 平台事务管理器 -->
<bean id="transactionManager"
       class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
       <property name="dataSource" ref="dataSource"/>
 </bean>

  <!-- 通知 -->
  <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
        <!-- 传播行为:propagation  不是必须的,默认值是REQUIRED  -->
        <!-- REQUIRED：如果有事务，则在事务中执行；如果没有事务，则开启一个新的事务 -->
        <tx:method name="save*" propagation="REQUIRED" />
        <!-- SUPPORTS：如果有事务，则在事务中执行；如果没有事务，则不会开启事务 -->
        <tx:method name="find*" propagation="SUPPORTS" read-only="true" />
        </tx:attributes>
 </tx:advice>

 <aop:config>
       <aop:pointcut id="txPointCut" expression="execution(* com.service.*.*
(..))" />
       <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointCut"/>
  </aop:config>
```

> isolation 设置隔离机制,不是必须的 默认值DEFAULT

> timeout 不是必须的 默认值-1(永不超时) 表示事务超时的时间（以秒为单位）

> read-only 不是必须的 默认值false不是只读的 表示事务是否只读？

> rollback-for 不是必须的 表示将被触发进行回滚的 Exception(s)；以逗号分开。

如：'com.ityhp.MyBusinessException,ServletException'

> no-rollback-for 不是必须的 表示不被触发进行回滚的 Exception(s)；以逗号分开。

如：'com.foo.MyBusinessException,ServletException'

需要包:

```
<dependency>
     <groupId>org.aspectj</groupId>
     <artifactId>aspectjweaver</artifactId>
     <version>1.8.13</version>
</dependency>
```

14. 使用注解方式添加事务

14.1 使用@Transactional注解-添加tx命名空间

```
@Transactional     //对业务类进行事务增强的标注
@Service("accountService")
public class AccountServiceImpl implements AccountService {}
```

配置xml

```
  <bean id="transactionManager"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
              <property name="dataSource" ref="dataSource"/>
       </bean>

       <!--① 对标注@Transactional注解的Bean进行加工处理,以织入事物管理切面 -->
      <tx:annotation-driven transaction-manager="transactionManager" />
```

在默认情况, <tx:annotation-driven /> 中transaction-manager属性会自动使用名为 "transactionManager" 的事务管理器.

所以,如果用户将事务管理器的id定义为 `transactionManager` , 则可以进一步将①处的配置简化为 <tx:annotation-driven />.

14.2 @Transactional其他方面介绍

属 性 名	说 明
propagation	事务传播行为，通过以下枚举类提供合法值： org.springframework.transaction.annotation.Propagation 例如：@Transactional(propagation=Propagation.REQUIRES_NEW)
isolation	事务隔离级别，通过以下枚举类提供合法值： org.springframework.transaction.annotation.Isolation 例如：@Transactional(isolation=Isolation.READ_COMMITTED)
readOnly	事务读写性，布尔型。例如：@Transactional(readOnly=true)
timeout	超时时间，int 型，以秒为单位。例如：@Transactional(timeout=10)
rollbackFor	一组异常类，遇到时进行回滚，类型为：Class<? extends Throwable>[]，默认值为{}。例如： @Transactional(rollbackFor={SQLException.class})。多个异常之间可用逗号分隔
rollbackForClassName	一组异常类名，遇到时进行回滚，类型为 String[]，默认值为{}。例如： @Transactional(noRollbackForClassName={"Exception"})
noRollbackFor	一组异常类，遇到时不回滚，类型为 Class<? extends Throwable>[]，默认值为{}
noRollbackForClassName	一组异常类名，遇到时不回滚，类型为 String[]，默认值为{}

* 关于@Transactional的属性

基于@Transactional默认的属性.

* 事务传播行为: PROPAGATION_REQUIRED.

* 事务隔离级别: ISOLATION_DEFAULT.

* 读写事务属性:读/写事务.

* 超时时间:依赖于底层的事务系统默认值

* 回滚设置:任何运行期异常引发回滚,任何检查型异常不会引发回滚.

默认值可能适应大部分情况,但是我们依然可以可以自己设定属性,具体属性表如下:

* 在何处标注@Transactional注解?

@Transactional注解可以直接用于接口定义和接口方法,类定义和类的public方法上.

但Spring建议在业务实现类上使用@Transactional注解,当然也可以添加到业务接口上,

但是这样会留下一些容易被忽视的隐患,因为注解不能被继承,所以业务接口中标注的@Transactional注解不会被业务类实现继承.

14.3 使用不同的事务管理器

一般情况下,一个应用仅需要使用一个事务管理器.如果希望在不同的地方使用不同的事务管理,@Transactional注解同样支持!

实现代码:

```
@Transactional("事务管理器的名字") //此处添加指定事务管理器的名字
@Service("accountService")
public class AccountServiceImpl implements AccountService {}
```

对应事务查找事务管理器的名字应该在xml中进行定义!如下:

```
  <!--声明一个事务管理器 -->
  <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
                <property name="dataSource" ref="dataSource"/>
            <qualifier value="定义事务管理器的名字,可以被注解查找" />
  </bean>
```