

Spring4.3.8

作者: 高薪茹

Spring意思是春天,于2003 年兴起的一个轻量级的Java开源框架, 由Rod Johnson(音乐博士) 在其著作Expert One-On-One J2EE Development and Design中阐述的部分理念和原型衍生而来。

框架的主要优势之一就是其分层架构, 分层架构允许您选择使用哪一个组件, 同时为J2EE 应用程序开发提供集成的框架。

它以IoC(控制反转)和AOP(面向切面编程)两种先进的技术为基础, 完美的简化了企业级开发的复杂度。

- [Spring4.3.8](#)
 - ■ ■ [作者: 高薪茹](#)
- [1. Spring 架构图](#)
- [2. 开发 Spring 所需要的工具](#)
 - [2.1 Spring 的 jar 包](#)
 - [2.2 Spring 配置文件](#)
- [3. Spring 基本功能详解](#)
 - [3.1 SpringIoC](#)
 - [3.2 Spring 容器内部对象](#)
 - [3.2.1创建对象的方式](#)
 - [3.2.1.1 无参构造函数](#)
 - [3.2.1.2 静态工厂](#)
 - [3.2.1.3 实例工厂](#)
 - [3.2.2 对象的 scope](#)
 - [3.2.2.1 singleton\(默认\)](#)
 - [3.2.2.2 prototype](#)
 - [3.2.2.3 Request](#)
 - [3.2.2.4 Session](#)
 - [3.2.2.5 Global session](#)
 - [3.2.3 初始化 bean 的时机](#)
 - [3.2.4 init, destroy](#)

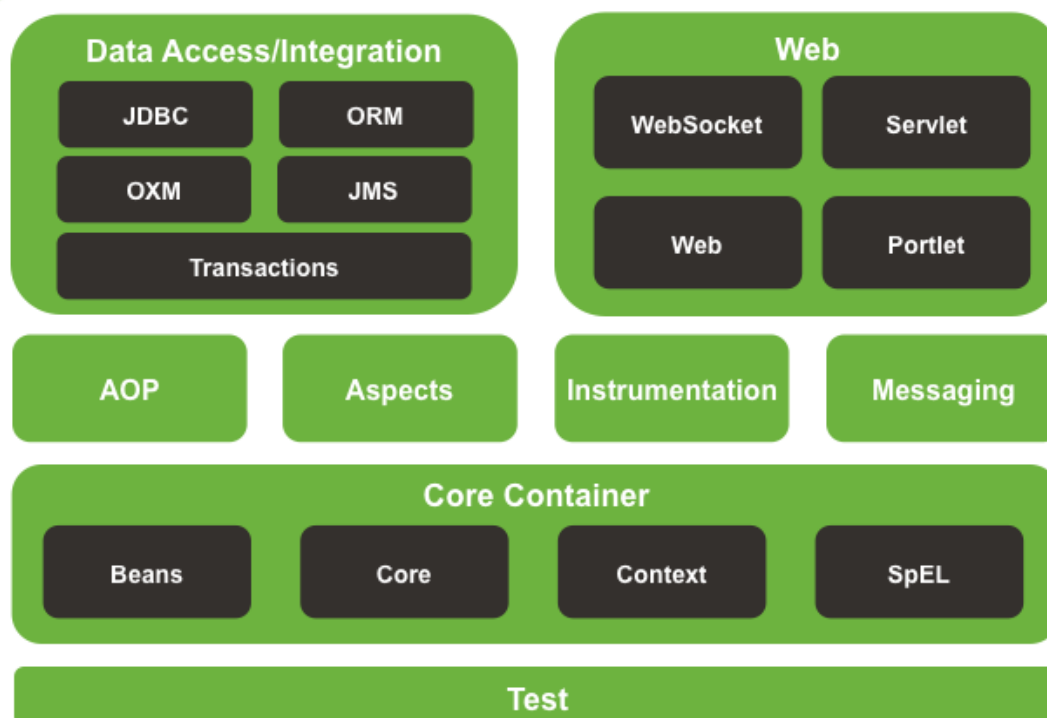
- [3.3 别名](#)
- [3.4 依赖注入\(DI\)](#)
 - [3.4.1 xml 形式](#)
 - [3.4.1.1 使用构造器注入](#)
 - [3.4.1.2 使用属性 setter 方法](#)
 - [3.4.1.3 装配 list 集合](#)
 - [3.4.1.4 装配 set](#)
 - [3.4.1.5 装配 map](#)
 - [3.4.1.5 装配 properties](#)
 - [3.4.1.6 装配 Object\[\]](#)
 - [3.4.2 注解 Annotation](#)
 - [配置bean](#)
 - [3.4.2.1 @Component](#)
 - [3.4.2.2 @Controller](#)
 - [3.4.2.3 @Repository](#)
 - [3.4.2.4 @Service](#)
 - [配置 bean 的属性](#)
 - [3.4.2.5 @Autowired](#)
 - [3.4.2.6 @Qualifier](#)
 - [3.4.2.7 @Resource](#)
 - [3.4.2.8 @Required](#)
 - [3.4.2.8 @PostConstruct & @PreDestroy](#)
 - [3.5 扫描](#)
 - [3.6 Spring 的继承](#)
- [4. 面向切面编程](#)
 - [4.1 代理模式](#) Spring02_1_Proxy
 - [4.1.1 静态代理](#)
 - - [4.1.2 JDK动态代理](#)
 - [4.1.3 CGLIB动态代理](#)
 - [4.2 AOP编程](#)
 - [4.2.1概念](#)
 - - [4.2.2 Spring AOP实现的两种模式](#)
 - [4.2.2.1 xml形式](#)
 - [4.2.2.2 注解形式](#)

- [4.2.3 切入点表达式](#)
- [4.2.4 通知](#)
 - [4.2.4.1 通知种类](#)
 - [4.2.4.2 通知使用细节](#)
- [4.2.5 SpringAOP 细节](#)
- [5.Spring 与 Struts2 整合](#)
 - [5.1 方式一](#)
 - [5.2 方式二](#)
- [6.Spring 与 Hibernate 整合](#)
- [7.S2SH 框架整合](#)
 - ▪ [7.1. 各个框架扮演的角色](#)
 - [7.2. 整合的原则](#)
 - [7.3. 步骤](#)
 - [7.4 启动流程](#)

1. Spring 架构图



Spring Framework Runtime



2. 开发 Spring 所需要的工具

2.1 Spring 的 jar 包

[下载 Spring jar包地址](#)

1. Spring 核心功能(CoreContainer): spring-core, spring-beans, spring-context, spring-expression, commons-logging

2. spring-core 和 spring-beans 提供了基本的依赖注入和控制反转功能

spring-context 在 Core&Beans 的基础上提供基本的框架API, ApplicationContext接口是模块的核心

spring-expression 提供了在运行时操作对象的语言

[不同功能需要依赖不同 jar 包, jar 包详解地址](#)

2.2 Spring 配置文件

默认情况下命名为 applicationContext.xml ,可以建立多个 xml 文件.

3. Spring 基本功能详解

3.1 SpringIoC

Spring的控制反转：把对象的创建、初始化、销毁等工作交给spring容器来做。由spring容器控制对象的生命周期。

IoC : Inverse Of Control(控制反转)

1. 把自己 new 的东西改为由容器提供
 - a) 初始化具体值
 - b) 装配
2. 好处:灵活装配

3.2 Spring 容器内部对象

3.2.1创建对象的方式

3.2.1.1 无参构造函数

```
public class HelloWorld {  
    public HelloWorld(){  
        System.out.println("spring 在默认的情况下，使用默认的构造函数");  
    }  
    public void sayHello(){  
        System.out.println("hello");  
    }  
}
```

<!--

一个bean就是描述一个类

id就是标示符

命名规范：类的第一个字母变成小写，其他的字母保持不变

class为类的全名

-->

```
<bean id="helloWorld" class="com.lanou.domain.HelloWorld"></bean>
```

```

@Test
    public void testHelloWorld(){
        //启动spring容器
        ApplicationContext context=new ClassPathXmlApplicationContext("
applicationContext.xml");
        //从spring容器中把对象提取出来
        HelloWorld helloWorld=(HelloWorld)context.getBean("helloWorld")
;
        helloWorld.sayHello();
    }

```

3.2.1.2 静态工厂

```

public class HelloWorldFactory {
    public static HelloWorld getInstance(){
        return new HelloWorld();
    }
}

```

<!--静态工厂
 静态方法，静态方法不需要实例化对象，
 class指定的就是工厂类型
 factory-method 是工厂里面的静态方法
 -->

```

<bean id="helloWorldFactory" class="com.lanou.domain.HelloWorldFact
ory" factory-method="getInstance"></bean>

```

```

/**
 * 静态工厂
 * 在spring 内部，调用了 HelloWorldFactory 内部的 getInstance 内部方法
 * 而该方法的内容，就是创建对象的过程，是由程序员来完成的
 * 这就是静态工厂
 * */
@Test
    public void testHelloWorldFactory(){
        ApplicationContext context=new ClassPathXmlApplicationContext("
applicationContext.xml");
        HelloWorld helloWorld=(HelloWorld)context.getBean("helloWorldFa
ctory");
        helloWorld.sayHello();
    }

```

```
}
```

3.2.1.3 实例工厂

```
public class HelloWorldFactory2 {  
    public HelloWorld getInstance(){  
        return new HelloWorld();  
    }  
}
```

```
<!--实例工厂  
工厂类创建对象  
工厂类，实例方法，必须先创建工厂  
再创建对象，用factory的实例方法  
-->  
<bean id="helloWorldFactory2" class="com.lanou.domain.HelloWorldFac  
tory2">  
</bean>  
<!--factory-bean 指向实例工厂的bean-->  
<!--factory-method 实例工厂对象的方法-->  
<bean id="helloWorld3" factory-bean="helloWorldFactory2" factory-m  
ethod="getInstance">  
</bean>
```

```
/**  
 * 实例工厂  
 * 1.spring容器（beans）创建了一个实例工厂的bean  
 * 2.该bean 调用了工厂方法的getInstance 方法产生对象  
 * */  
@Test  
public void testHelloWorldFactory2(){  
    ApplicationContext context=new ClassPathXmlApplicationContext("  
applicationContext.xml");  
    HelloWorld helloWorld=(HelloWorld)context.getBean("helloWorld3"  
);  
    helloWorld.sayHello();  
}
```

3.2.2 对象的 scope

简单说就是对象在spring容器（IOC容器）中的生命周期，也可以理解为对象在spring

容器中的创建方式。

目前，scope的取值有5种取值：

在Spring 2.0之前，有singleton和prototype两种；

在Spring 2.0之后，为支持web应用的ApplicationContext，增强另外三种：request，session和global session类型，它们只实用于web程序，通常是和XmlWebApplicationContext共同使用。

3.2.2.1 singleton(默认)

每个Spring IoC 容器中一个bean定义只有一个对象实例(共享)。

此取值时表明容器中创建时只存在一个实例，所有引用此bean都是单一实例。此外，singleton类型的bean定义从容器启动到第一次被请求而实例化开始，只要容器不销毁或退出，该类型的bean的单一实例就会一直存活，典型单例模式，如同servlet在web容器中的生命周期。

3.2.2.2 prototype

允许bean可以被多次实例化(使用一次就创建一个实例) . Spring不能对一个prototype bean的整个生命周期负责. **无论lazy-init 的值是什么, 都在 context.getBean时才创建对象.**

spring容器在进行输出prototype的bean对象时，会每次都重新生成一个新的对象给请求方，虽然这种类型的对象的实例化以及属性设置等工作都是由容器负责的，但是只要准备完毕，并且对象实例返回给请求方之后，容器就不再拥有当前对象的引用，请求方需要自己负责当前对象后继生命周期的管理工作，包括该对象的销毁。也就是说，容器每次返回请求方该对象的一个新的实例之后，就由这个对象“自生自灭”，最典型的体现就是spring与struts2进行整合时，要把action的scope改为prototype。

3.2.2.3 Request

再次说明 request，session和global session类型只适用于web程序，通常是和XmlWebApplicationContext共同使用,作用域仅在基于web的Spring ApplicationContext情形下有效。

request表示该针对每一次HTTP请求都会产生一个新的bean，同时该bean仅在当前**HTTP request** 内有效。当处理请求结束，request作用域的bean实例将被销毁。

3.2.2.4 Session

session作用域表示该针对每一次HTTP请求都会产生一个新的bean，同时该bean仅在当前**HTTP session**内有效。

3.2.2.5 Global session

在一个全局的HTTP Session中，一个bean定义对应一个实例。典型情况下，仅在使用portlet context的时候有效。该作用域仅在基于web的Spring ApplicationContext情形下有效。

3.2.3 初始化 bean 的时机

Spring 默认在启动时将所有singleton bean提前进行实例化。提前实例化意味着作为初始化的一部分,ApplicationContext 会自动创建并配置所有的singleton bean. 通常情况下这是件好事。因为这样在配置中有任何错误能立即发现。

```
lazy-init="true" OR "false"
```

Lazy-init 为false, spring容器将在启动的时候报错（比较好的一种方式）

Lazy-init 为true, spring容器将在调用该类的时候出错。

```
<bean id="per1" class="com.lanou.domain.Person"/>
<bean id="per2" class="com.lanou.domain.Person" lazy-init="true"/>
```

可以打断点查看无参构造函数的执行

```
@Test
public void testPerson_lazy_init_true(){
    ApplicationContext context = new ClassPathXmlApplicationContext("ap
plicationContext.xml");
    Person person = (Person) context.getBean("per1");
    /* lazy-init="true"
    * 在 context.getBean("per2") 时才创建该对象
    * */
    Person person2 = (Person) context.getBean("per2");
}
```

如果想对所有bean都应用延迟初始化，可以在根节点beans设置default-lazy-init="true"，如下：

```
<beans default-lazy-init="true" ...>
```

3.2.4 init, destroy

Spring初始化bean或销毁bean时，有时需要作一些处理工作，因此spring可以在创建和拆卸bean的时候调用bean的两个生命周期方法。

```
public class Person {

    public void init(){
        System.out.println("初始化");
    }

    public void destroy(){
        System.out.println("销毁");
    }
}
```

```
<bean id="per1" class="com.lanou.domain.Person" init-method="init" destroy-method="destroy"/>
```

当 per1 被载入到Spring容器中时调用init-method方法。当 per1 从容器中删除时调用destroy-method（scope = singleton有效）

```
/**
 * 在构造方法之后立即执行 init 方法，
 * 如果 spring 执行close方法，关闭之后执行 destroy 方法
 */

@Test
public void testPerson_init_destroy() {
    ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
    Person person = (Person) context.getBean("per1");
    ((ClassPathXmlApplicationContext) context).close(); //spring 容器关闭
}
```

3.3 别名

```
<bean id="car" name="BMCAR" class="com.lanou.domain.Car"></bean>
<!-- name的属性值和id对应-->
<alias name="car" alias="QQ"/>
<alias name="car" alias="劳斯莱斯"/>
```

```
ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
```

```
Car car1 = (Car) context.getBean("car");
Car car2 = (Car) context.getBean("QQ");
Car car3 = (Car) context.getBean("劳斯莱斯");
// 都是同一个对象
System.out.println(car1);
System.out.println(car2);
System.out.println(car3);
```

3.4 依赖注入(DI)

控制反转，我们可以把它看作是一个概念。而依赖注入(Dependency Injection)是控制反转的一种实现方法。

James Shore给出了依赖注入的定义：依赖注入就是将实例变量传入到一个对象中去 (Dependency injection means giving an object its instance variables)。

3.4.1 xml 形式

3.4.1.1 使用构造器注入

通过参数顺序, 类型等

```
<bean id="person" class="com.lanou.domain.Person">
    <!--根据参数名称注入-->
    <constructor-arg name="name" value="大老王"/>
    <!--根据索引index注入-->
    <constructor-arg index="1" value="24"/>
    <!--根据构造器参数的类型-->
    <constructor-arg type="java.lang.String" value="men"/>
</bean>
```

3.4.1.2 使用属性 setter 方法

```
public class Person {
    private String name;
    private int age;
    private String gender;
    private Car car;
    private List list;
    private Set set;
    private Map map;
    private Properties properties;
    private Object[] objects;
    [属性 setter 方法]
```



```
}
```

简单Bean的注入

```
<bean name="person" class="com.lanou.domain.Person">
  <!-- property 就是一个bean 的属性
        name 用来描述属性的名称
        value 就是属性值，一般类型(基本类型和 String)
  -->
  <property name="name" value="狗蛋儿"/>
  <property name="age" value="22"/>
  <property name="gender" value="men"/>
</bean>
```

引用其他Bean

```
<bean name="car" class="com.lanou.domain.Car"></bean>
<bean name="person" class="com.lanou.domain.Person">
  <!--spring 容器内部创建的car 对象给 Person 的 car属性赋值-->
  <property name="car" ref="car"/>
</bean>
```

3.4.1.3 装配 list 集合

```
<property name="list">
  <list>
    <value>list1</value>
    <ref bean="car"/>
  </list>
</property>
```

3.4.1.4 装配 set

```
<property name="set">
  <set>
    <value>set1</value>
    <ref bean="car"/>
  </set>
</property>
```


3.4.1.5 装配 map

```
<property name="map">
  <map>
    <entry key="m1">
      <value>map1</value>
    </entry>
    <entry key="m2">
      <ref bean="car"/>
    </entry>
  </map>
</property>
```

3.4.1.5 装配 properties

```
<property name="properties">
  <props>
    <prop key="p1">pp1</prop>
    <prop key="p2">pp2</prop>
  </props>
</property>
```

3.4.1.6 装配 Object[]

```
<property name="objects">
  <list>
    <value>obj1</value>
    <ref bean="car"/>
  </list>
</property>
```

3.4.2 注解 Annotation

注解的注入会在XML之前，因此后者配置将会覆盖前者。

配置bean

3.4.2.1 @Component

@Component是所有受 Spring 管理组件的通用形式，泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。

@Component 不推荐使用。

```
@Component
public class Person {
}
```

3.4.2.2 @Controller

@Controller对应表现层的Bean，也就是Action, 例:

```
@Controller
@Scope("prototype")
public class UserAction {
}
```

3.4.2.3 @Repository

用于标注数据访问组件，即DAO组件

```
@Repository
public class UserDao {
}
```

3.4.2.4 @Service

用于标注业务层组件

```
@Service
public class UserServiceImpl {
}
```

对于扫描到的组件，Spring有默认的命名策略：

1) 使用非限定类名，第一个字母小写

(UserServiceImpl -> userServiceImpl)

2) 在注解中通过 value 属性值标识组件的名称

(通常可以将UserServiceImpl -> userService, 可以将Impl拿掉，这是一个习惯)

```
@Service("userService")
```

```
public class UserServiceImpl {  
}
```

(3) 当在组件类上使用了特定的注解之后,还需要在 Spring 的配置文件中声明

`<context:component-scan>` :

base-package 属性指定一个需要扫描的基类包, Spring 容器将会扫描这个基类包里及其子包中的所有类.

当需要扫描多个包时,可以使用逗号分隔.

配置 bean 的属性

3.4.2.5 @Autowired

使用@Autowired 注解自动装配, @Autowired 默认按类型装配.

可以在普通setter方法上使用@Autowired注解

```
public class UserServiceImpl implements UserService{  
    private UserDao userDao;  
  
    @Autowired  
    public void setUserDao(UserDao userDao) {  
        this.userDao = userDao;  
    }  
}
```

可以将注解应用到构造器和字段上

```
public class UserServiceImpl implements UserService{  
    private UserDao userDao;  
  
    @Autowired  
    public UserServiceImpl(UserDao userDao) {  
        this.userDao = userDao;  
    }  
}
```

```
public class UserServiceImpl implements UserService{  
    @Autowired  
    private UserDao userDao;  
}
```


可以将注解应用到任意名称或参数的方法上

```
public class UserServiceImpl implements UserService{
    private UserCatalog userCatalog;
    private UserDao userDao;

    @Autowired
    public void prepare(UserCatalog catalog, UserDao dao){
        this.userCatalog = catalog;
        this.userDao = dao;
    }
}
```

可以把指定类型的所有Bean提供给一个数组类型,集合类型的变量

```
public class UserServiceImpl implements UserService{
    @Autowired
    private UserCatalog[] catalogs;
}
```

xml 中只定义了几个 对应bean

```
<bean id="userService" class="com.lanou.service.impl.UserServiceImpl"/>
<bean id="dao" class="com.lanou.dao.impl.UserDaoImpl"/>
<bean id="catalog" class="com.lanou.domain.UserCatalog"/>
```

@Autowired注解是按类型装配依赖对象，默认情况下它要求依赖对象必须存在，一旦找不到对应的Bean，自动注入就会失败。若要改变默认的策略或允许null值，可以设置它required属性为false。

```
public class UserServiceImpl implements UserService{
    // xml 中没有定义UserFinder的 bean
    @Autowired(required = false)
    private UserFinder userFinder;
}
```

3.4.2.6 @Qualifier

使用@Qualifier调整基于注解的自动注入。

在指定参数上使用@Qualifier，可以缩小类型匹配的范围，更容易为参数找到指定的Bean。

[想使用按名称装配, 可以结合@Qualifier注解一起使用]

```
public class UserServiceImpl implements UserService{
    @Qualifier("userD")
    @Autowired
    private UserDao userD;
}
```

对应的Bean定义如下所示。

标识值为“userDe”的Bean将会被标记为@Qualifier(“userDe”)的参数所注入:

```
<bean id="dao" class="com.lanou.dao.impl.UserDaoImpl">
    <qualifier value="userD"/>
</bean>
```

如果全部使用注解, 可以在UserDaoImpl中使用Repository配置

```
@Repository("userDe")
public class UserDaoImpl implements UserDao{
}
```

3.4.2.7 @Resource

@Resource 注解和@Autowired一样, 也可以标注在字段或属性的setter方法上.

@Resource注解默认按名称装配。

名称可以通过@Resource的name属性指定, 如果没有指定name属性,
当注解标注在字段上, 即默认取字段的名称作为bean名称寻找依赖对象;
当注解标注在属性的setter方法上, 即默认取属性名作为bean名称寻找依赖对象。

```
public class UserServiceImpl implements UserService{
    @Qualifier("userD")
    @Autowired
    private UserDao userD;

    @Resource(name = "daoU")
    private UserDao userDao;
}
```

注: 如果没有指定name属性, 并且按照默认的名称找不到依赖对象时, @Resource注解会回退到按类型装配。但一旦指定了name属性, 就只能按名称装配了。

3.4.2.8 @Required

@Required注解应用于Bean属性的setter方法, 如下:

```
public class UserServiceImpl implements UserService{
    private UserDao userDao;
    @Required
    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }
}
```

这个表明Bean的属性值必须在配置中指定, 可以通过显示的在Bean定义中指定, 也可以使用自动注入。若没有为这个属性指定值, 那么容器会抛出一个异常。

```
<bean id="userService" class="com.lanou.service.impl.UserServiceImpl">
    <property name="userDao" ref="dao"/>
</bean>
<bean id="dao" class="com.lanou.dao.impl.UserDaoImpl"/>
```

3.4.2.8 @PostConstruct & @PreDestroy

生命周期注解

之前的做法是在xml中定义init-method 和 destroy-method方法

```
public class User {
    public User() {
        System.out.println("user .. 构造函数");
    }
    @PostConstruct
    public void init(){
        System.out.println("I'm init method using @PostConstruct....");
    }

    @PreDestroy
    public void destroy(){
        System.out.println("I'm destroy method using @PreDestroy....");
    }
}
```

3.5 扫描

spring2.5为我们引入了组件自动扫描机制，它可以在类路径底下寻找标注了@Component、@Service、@Controller、@Repository注解的类，并把这些类纳入进spring容器中管理。

它的作用和在xml文件中使用bean节点配置组件是一样的。要使用自动扫描机制，我们需要打开以下配置信息：

1. 引入context命名空间 需要在xml配置文件中配置以下信息
2. 在配置文件中添加context:component-scan标签

```
<?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:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/context
         http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- 自动扫描指定包及其子包下的所有Bean类 -->
    <context:component-scan base-package="com.lanou.domain"/>
```

在 base-package 指明一个包,表明自动扫描目录 com.lanou 包及其子包中的注解，当需要扫描多个包时,可以使用逗号分隔。

其他用法:

```
<!-- 通过resource-pattern指定扫描的资源 -->
<!--只扫描repository下的类-->
<context:component-scan base-package="com.lanou" resource-pattern="dao/*.class"></context:component-scan>

<!-- 不扫描repository注解的类 。可以配置多个。-->
<context:component-scan base-package="com.lanou" >
    <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Repository"/>
</context:component-scan>

<!-- 只扫描repository注解的类。可以配置多个。 -->
<context:component-scan base-package="com.lanou" use-default-filters="false">
    <context:include-filter type="annotation" expression="org.springframework.stereotype.Repository"/>
</context:component-scan>
```



```

<!-- 不扫描某个接口。可以配置多个。使用assignable类型 -->
<context:component-scan base-package="com.lanou" use-default-filters="
true">
    <context:exclude-filter type="assignable" expression="com.lanou.dao
.UserDao"/>
</context:component-scan>

<!-- 只扫描某个接口。 -->
<context:component-scan base-package="com.lanou" use-default-filters="
false">
    <context:include-filter type="assignable" expression="com.lanou.dao
.UserDao"/>
</context:component-scan>

```

3.6 Spring 的继承

```

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

public class Student extends People {
}

```

```

<bean name="people" class="com.lanou.domain.People">
    <property name="name" value="111"/>
</bean>
<bean name="student" class="com.lanou.domain.Student"></bean>

```

```

public void testExtents(){
    ApplicationContext context = new ClassPathXmlApplicationContext("ap
plicationContext.xml");
    Student student = (Student) context.getBean("student");
    // student 打印出来的名字为 null
    System.out.println(student.getName());
}

```


配置文件需要更改

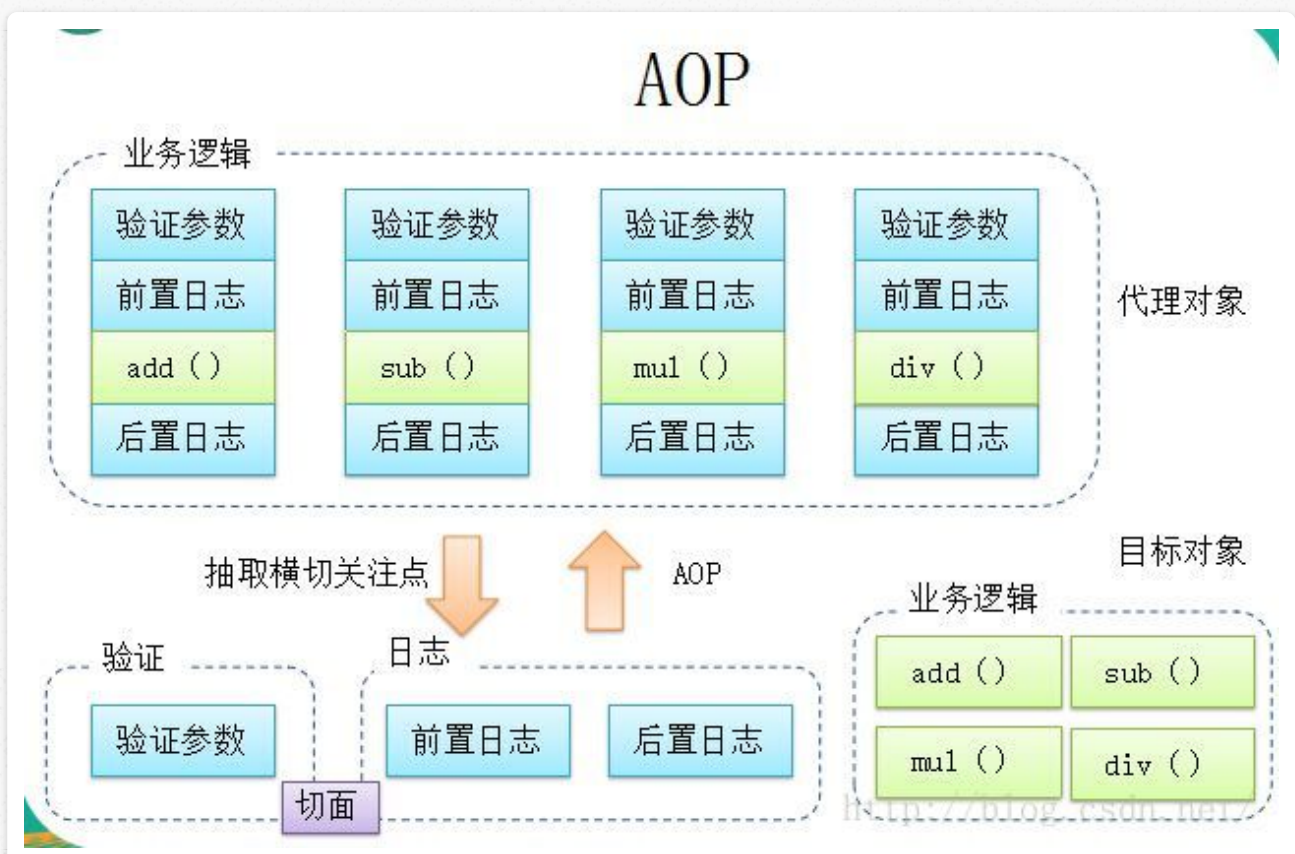
```
<!-- parent 实现了 spring 容器内部的继承关系
      在父类中进行赋值，子类继承父类内容
-->
<bean name="people" class="com.lanou.domain.People">
  <property name="name" value="111"/>
</bean>
<bean name="student" class="com.lanou.domain.Student" parent="people"><
/bean>
```

还有其他方式解决 student 的名字赋值问题么？

student 也继承了 people 的 name setter 方法

4. 面向切面编程

这里有一张网上看到的图我觉得也蛮适合理解的, 可以看完下面的代理之后返回来仔细看看这张图能够更容易理解:



4.1 代理模式 Spring02_1_Proxy

静态代理: 由程序员创建代理类或特定工具自动生成源代码再对其编译。在程序运行前代理类的.class文件就已经存在了。

动态代理: 在程序运行时运用反射机制动态创建而成。

我们在 Hibernate 中处理增删改功能都要添加事务,代码如下:

```
public class UserDaoImpl implements UserDao {
    @Override
    public void insertUser() {
        Session session = sessionFactory.openSession();
        Transaction transaction = session.beginTransaction();

        System.out.println("插入用户");

        transaction.commit();
        session.close();
    }

    @Override
    public void delete(Integer uid) {
        Session session = sessionFactory.openSession();
        Transaction transaction = session.beginTransaction();

        System.out.println("删除用户");

        transaction.commit();
        session.close();
    }
}
```

事务和逻辑的处理混杂在一起,如果再增添方法还需要继续重复有关事务的重复代码. 或者需求发生了变化, 要求项目中所有的类在执行方法时输出执行耗时。最直接的办法是修改源代码

缺点:

1、工作量特别大, 如果项目中有多个类, 多个方法, 则要修改多次。

2、违背了一些设计原则:

开闭原则 (OCP) [对扩展开放, 对修改关闭, 而为了增加功能把每个方法都修改了, 也不便于维护。]

单一职责 (SRP) [每个方法除了要完成自己本身的功能, 还要计算耗时、延时; 每一个方法引起它变化的原因就有多种。]

4.1.1 静态代理

1、定义UserDao接口,抽象主题

```
public interface UserDao{
    void insertUser();
    void delete(Integer uid);
}
```

2、实现UserDao接口,被代理的目标对象,真实主题

```
public class UserDaoImpl implements UserDao {
    @Override
    public void insertUser() {
        System.out.println("插入用户");
    }
    @Override
    public void delete(Integer uid) {
        System.out.println("删除用户");
    }
}
```

3、代理类,静态代理类

```
public class UserDaoProxy implements UserDao{
    //被代理的对象
    private UserDao userDao;
    private Transaction transaction;

    public UserDaoProxy(UserDao userDao, Transaction transaction) {
        this.userDao = userDao;
        this.transaction = transaction;
    }

    @Override
    public void insertUser() {
        transaction.beginTransaction();
        userDao.insertUser();
        transaction.commitTransaction();
    }

    @Override
    public void delete(Integer uid) {
        transaction.beginTransaction();
    }
}
```



```
        userDao.insertUser();
        transaction.commitTransaction();
    }
}
```

4、测试

```
public class ProxyTest {
    @Test
    public void testProxy(){
        UserDao dao = new UserDaoImpl();
        Transaction transaction = new Transaction();
        UserDaoProxy proxy = new UserDaoProxy(dao, transaction);
        proxy.insertUser();
    }
}
```

通过静态代理可以解决这些设计原则问题,但是不能解决:

如果项目中有多类,则需要编写多个代理类,工作量大,不好修改,不好维护,不能应对变化.

如果要解决上面的问题,可以使用动态代理

4.1.2 JDK动态代理

JDK的动态代理主要涉及 `java.lang.reflect` 包中的两个类: `Proxy`和`InvocationHandler`。其中, `InvocationHandler`是一个接口,可以通过实现该接口定义横切逻辑,并通过反射机制调用目标类的代码,动态地将横切逻辑和业务逻辑编织在一起。而`Proxy`利用 `InvocationHandler`动态创建一个符合某一接口的实例,生成目标类的代理对象。

只需要一个代理类,而不是针对每个类编写代理类。

1、在上一个示例中修改代理类 `UserDaoProxy` 如下:

JDK动态代理需要实现 `InvocationHandler` 接口,这个类其实应该是拦截器类

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

/**
 * 拦截器:
 *     1. 目标类导入
```



```

*      2. 事务导入
*      3. invoke完成:
*          1.开启事务
*          2.调用目标对象方法
*          3.事务提交
*/
public class UserDaoProxy implements InvocationHandler {
    private Object target;
    private Transaction transaction;

    public UserDaoProxy(Object target, Transaction transaction) {
        this.target = target;
        this.transaction = transaction;
    }

    /**
     * 当用户调用对象中的每个方法时都通过下面的方法执行，方法必须在接口
     * proxy 被代理后的对象
     * method 将要被执行的方法信息（反射）
     * args 执行方法时需要的参数
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        String methodName = method.getName();
        if (!methodName.equals("getUser")){
            this.transaction.beginTransaction();
            method.invoke(target, args); // 目标对象调用目标方法
            this.transaction.commitTransaction();
        } else {
            method.invoke(target, args);
        }
        return null;
    }

    /**
     * 获得被代理后的对象
     * @param object 被代理的对象
     * @return 代理后的对象
     *
     * loader:一个ClassLoader对象，定义了由哪个ClassLoader对象来生成代理对象进行加载
     * interfaces:一个Interface对象的数组，表示的是我将要给我需要代理的对象提供一组什么接口，如果我提供了一组接口给它，那么这个代理对象就宣称实现了该接口(多态)，这样我就能调用这组接口中的方法了
     * h:一个InvocationHandler对象，表示的是当我这个动态代理对象在调用方法的时候，会关联到哪一个InvocationHandler对象上，间接通过invoke来执行
     */
    public Object getProxyObject(Object object){
        this.target = object;
        return Proxy.newProxyInstance(

```

```

        target.getClass().getClassLoader(), //类加载器
        target.getClass().getInterfaces(), //获得被代理对象的所有接口

        this); //拦截器: InvocationHandler对象
    }
}

```

2、测试一下:

```

public class JDKProxyTest {
    @Test
    public void testProxy(){
        /*
            1. 创建一个目标类
            2. 创建一个事务
            3. 创建一个拦截器
            4. 动态产生代理对象
        */
        Object target = new UserDaoImpl();
        Transaction transaction = new Transaction();
        List<Interceptor> interceptorList = new ArrayList<>();
        interceptorList.add(transaction);
        UserDaoProxy interceptor = new UserDaoProxy(target, transaction);

        UserDao dao = (UserDao) interceptor.getProxyObject(target);
        dao.delete("10");
    }
}

```

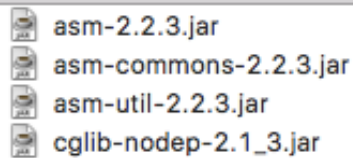
使用内置的Proxy实现动态代理有一个问题: 被代理的类必须实现接口, 未实现接口则没办法完成动态代理。

如果项目中有些类没有实现接口, 则不应该为了实现动态代理而刻意去抽出一些没有实例意义的接口, 通过cglib可以解决该问题。

4.1.3 CGLIB动态代理

CGLIB(Code Generation Library)是一个开源项目,是一个强大的, 高性能, 高质量的Code生成类库, 它可以在运行期扩展Java类与实现Java接口, 通俗说cglib可以在运行时动态生成字节码。

1、引用 cglib jar 包, 通过maven或者直接下载 jar 包添加



- asm-2.2.3.jar
- asm-commons-2.2.3.jar
- asm-util-2.2.3.jar
- cglib-nodep-2.1_3.jar

下载地址: <http://download.csdn.net/download/javawebxy/6849703>

maven:

```
<!-- https://mvnrepository.com/artifact/cglib/cglib -->
<dependency>
<groupId>cglib</groupId>
<artifactId>cglib</artifactId>
<version>2.2.2</version>
</dependency>
```

2、修改 拦截器类

```
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;
import java.lang.reflect.Method;

/*
 * 动态代理类
 * 实现了一个方法拦截器接口
 */
public class MyInterceptor implements MethodInterceptor {
    // 被代理对象
    private Object target;
    private Transaction transaction;

    public MyInterceptor(Transaction transaction) {
        this.transaction = transaction;
    }

    //动态生成一个新的类，使用父类的无参构造方法创建一个指定了特定回调的代理实例
    public Object getProxyObject(Object object) {
        this.target = object;
        // 代码增强类
        Enhancer enhancer = new Enhancer();
        // 回调方法
        enhancer.setCallback(this); // 参数就是拦截器
        //设置生成类的父类类型
        enhancer.setSuperclass(target.getClass()); // 生成的代理类父类是目标
        //动态生成字节码并返回代理对象
        return enhancer.create();
    }
}
```



```

@Override
public Object intercept(Object o, Method method, Object[] objects,
MethodProxy methodProxy) throws Throwable {
    this.transaction.beginTransaction();
    method.invoke(target, objects);
    this.transaction.commitTransaction();
    return null;
}
}

```

3、测试

```

/**
 * 通过 CGLIB 产生的代理类是目标类的子类
 */
public class CGLIBProxyTest {
    @Test
    public void testCGLIB(){
        Object target = new UserDaoImpl();
        Transaction transaction = new Transaction();
        MyInterceptor interceptor = new MyInterceptor(transaction);

        UserDaoImpl dao = (UserDaoImpl) interceptor.getProxyObject(target);
        dao.insertUser();

        //另一个被代理的对象,不再需要重新编辑代理代码
        PersonDaoImpl personDao = (PersonDaoImpl) interceptor.getProxyObject(new PersonDaoImpl());
        personDao.delete();
    }
}

```

4.2 AOP编程


4.2.1概念

使用 JDK 动态代理的代码进行对应说明

1. Aspect(切面)

比如说事务、权限等，与业务逻辑没有关系的部分


```
public class Transaction{
```

 切面

```
    public void beginTransaction(){  
        System.out.println("开启事务");  
    }
```

```
    public void commitTransaction(){  
        System.out.println("提交事务");  
    }
```

2. joinPoint(连接点)

目标类的目标方法。（由客户端在调用的时候决定）

```

public class JDKProxyTest {
    @Test
    public void testProxy() {
        Object target = new UserDaoImpl();
        Transaction transaction = new Transaction();
        List<Interceptor> interceptorList = new
        ArrayList<>();
        interceptorList.add(transaction);
        UserDaoProxy interceptor = new UserDaoProxy
        (target, transaction);
        UserDao dao = (UserDao) interceptor
        .getProxyObject(target);
        dao.delete( uid: "10");
    }
}

```

连接点
在客户端调用哪个
方法, 那个方法就是
连接点

3. pointCut(切入点)

所谓切入点是指我们要对那些拦截的方法的定义.
被纳入spring aop中的目标类的方法。

4. Advice(通知)

所谓通知是指拦截到joinpoint之后所要做的事情就是通知.通知分为前置通知,后置通知,
异常通知,最终通知,环绕通知(切面要完成的功能)

```

public class Transaction{

    public void beginTransaction(){
        System.out.println("开启事务");
    }

    public void commitTransaction(){
        System.out.println("提交事务");
    }
}

```

通知

5. Target(目标对象)

代理的目标对象

6. Weaving(织入)

是指把切面应用到目标对象来创建新的代理对象的过程.切面在指定的连接点织入到目标对象

对比表格查看

JDKProxy代理	SpringAop
目标对象	目标对象 Target
拦截器类	切面 Aspect
拦截器类中的方法	通知 Advice
被拦截到的目标类中方法的集合	切入点 pointCut
在客户端调用的方法（目标类目标方法）	连接点 joinPoint
代理类	AOP代理 AOP Proxy
代理类的代理方法生成的过程	织入 Weaving

4.2.2 Spring AOP实现的两种模式

4.2.2.1 xml形式

还以事务为例,准备 Transaction 切面类,目标接口类 UserDao,目标类 UserDaoImpl

```
/**
 * 切面
 */
public class Transaction {
    public void beginTransaction(){
        System.out.println("Begin Transaction");
    }

    public void commit(){
        System.out.println("Commit Transaction");
    }
}

-----

public interface UserDao {
    void saveUser();
}

-----

public class UserDaoImpl implements UserDao {
    @Override
    public void saveUser() {
        System.out.println("save Person");
    }
}
```

剩余的我们需要配置 applicationContext.xml

1、引入 aop 命名空间

```
<?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"
```



```
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/aop http://www.springframe
work.org/schema/aop/spring-aop-4.3.xsd" >
```

2、配置 两个 bean

```
<bean id="userDao" class="com.lanou.spring.aop.transaction.UserDaoImpl"
/>
<bean id="transaction" class="com.lanou.spring.aop.transaction.Transact
ion"/>
```

3、aop 的配置

```
<!-- aop 配置-->
<aop:config>
    <!-- 切入点表达式，确认目标类 -->
    <aop:pointcut id="pointcut" expression="execution(* com.lanou.spring.aop.transaction.UserDaoImpl.*(..))"/>
    <!-- ref 指向的对象就是切面 -->
    <aop:aspect ref="transaction">
        <!-- 前置通知-->
        <aop:before method="beginTransaction" pointcut-ref="pointcut" /
    >
        <!-- 正常返回通知-->
        <aop:after-returning method="commit" pointcut-ref="pointcut"/>
    </aop:aspect>
</aop:config>
```

4、测试

```
public class TransactionTest {
    @Test
    public void testTransaction(){
        ApplicationContext context = new ClassPathXmlApplicationContext
("applicationContext.xml");
        UserDao userDao = (UserDao) context.getBean("userDao");
        userDao.saveUser();
    }
}
```

5、报错, Spring缺少 **aspectjweaver.jar** 包, 添加 jar 包重新运行

```
Caused by: java.lang.ClassNotFoundException: org.aspectj.weaver.reflect
.ReflectionWorld$ReflectionWorldException
    at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
    ... 70 more
```

Spring AOP 原理:

1. 当spring容器启动的时候, 加载两个bean,对两个bean进行实例化
2. 当spring容器对配置文件解析到 `<aop:config>` 的时候
3. 把切入点表达式解析出来, 按照切入点表达式匹配spring容器内容的bean
4. 如果匹配成功, 则为该bean创建代理对象
5. 当客户端利用context.getBean获取一个对象时, 如果该对象有代理对象, 则返回代理对象; 如果没有代理对象, 则返回对象本身
6. 切入点表达式如与springbean 没有一个匹配就会报错

4.2.2.2 注解形式

1. 首先需要导入两个 jar 包 : aspectjrt & aspectjweaver . 需要注意如果你的 jdk 是1.7 就找这两个 jar 包的1.7.+版本; 我是jdk1.8, 使用了aspectjrt1.8.5 & aspectjweaver1.8.5
2. 在 applicationContext.xml 中添加context,aop 的命名空间.扫描, 添加自动创建代理设置

```
<?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:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/aop http://www.springframe
work.org/schema/aop/spring-aop-4.3.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd"
    >

    <!-- 扫描包 -->
    <context:component-scan base-package="com.lanou.transaction"/>
```

```
<!-- 自动创建代理 -->
<aop:aspectj-autoproxy/>
</beans>
```

3.添加 UserDaoImpl, Transaction 的注解

```
@Repository("userDao")
public class UserDaoImpl implements UserDao {...}
-----
@Component
public class Transaction {...}
```

4.在 Transaction 切面类中添加注解

```
@Component
@Aspect
public class Transaction {
    @Pointcut("execution(* com.lanou.transaction.UserDaoImpl.*(..))")
    private void method(){} //方法签名

    @Before("method()")
    public void beginTransaction(JoinPoint joinpoint){
        System.out.println("Begin Transaction");
    }

    @AfterReturning(value = "method()", returning = "returnVal")
    public void commit(JoinPoint joinpoint, Object returnVal)
    {
        System.out.println("返回值: " + returnVal);
        System.out.println("Commit Transaction");
    }

    @After("method()")
    public void finallyMethod(JoinPoint joinpoint){
        System.out.println("Finally Method");
    }

    @AfterThrowing(value = "method()", throwing = "throwable")
    public void throwingMethod(JoinPoint joinpoint, Throwable throwable
){
        System.out.println("异常: " + throwable.getMessage());
    }

    @Around("method()")
    public void round(ProceedingJoinPoint pjp) throws Throwable {

        System.out.println(" before .. before ");
    }
}
```

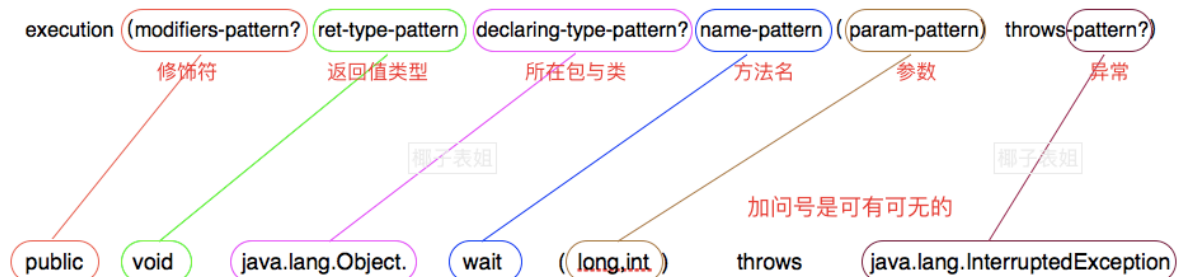


```

    pjp.proceed(); // 调用目标方法
    System.out.println(" after .. after ");
}
}

```

4.2.3 切入点表达式



execution (public * * (..)) : 任意公共方法的执行
 execution (* set* (..)) : 任何一个名字以 set 开始的方法的执行
 execution (* com.lanou.spring.aop.transaction.service.* * (..)) : 在 service 包中定义的任意方法的执行
 execution (* com.lanou.spring.transaction.service..* * (..)) : 在 service 包或其子包中定义的任意方法的执行
 execution (* com.lanou.spring.aop..service..* * (..)) : 在 aop 包及子包一直到 service 包,再子包下的所有类所有方法

4.2.4 通知

4.2.4.1 通知种类

名称	解释
前置通知 [Before advice]	在连接点前面执行，前置通知不会影响连接点的执行，除非此处抛出异常
正常返回通知 [After returning advice]	在连接点正常执行完成后执行，如果连接点抛出异常，则不会执行
异常返回通知 [After	

throwing advice]	在连接点抛出异常后执行
返回通知 [After (finally) advice]	在连接点执行完成后执行， 不管是正常执行完成，还是抛出异常，都会执行返回通知中的内容
环绕通知 [Around advice]	环绕通知围绕在连接点前后，比如一个方法调用的前后。 这是最强大的通知类型，能在方法调用前后自定义一些操作。 环绕通知还需要负责决定目标方法的执行

4.2.4.2 通知使用细节

1. 前置通知: 每种通知都能够添加连接点参数, 可以获取连接点信息

```

/*
    前置通知
    1. 在目标方法执行之前执行
    2. 获取不到目标方法返回值
*/
public void beginTransaction(JoinPoint joinpoint){
    System.out.println("连接点名称: "+joinpoint.getSignature().getName());
;
    System.out.println("目标类" + joinpoint.getTarget().getClass());

    System.out.println("Begin Transaction");
}

```

2. 后置通知可以获取返回值类型, 但当目标方法产生异常, 后置通知将不再执行

```

public class UserDaoImpl implements UserDao {
    @Override
    public String saveUser() {
        /* 制造异常
           int a = 1/0;
        */
        System.out.println("save User");
        return "11111111";
    }
}
...
...
<!-- 正常返回通知
    1. 可以获取目标方法的返回值
    2. 当目标方法产生异常，后置通知将不再执行

```

```
-->
<aop:after-returning method="commit" pointcut-ref="pointcut" return
ing="returnVal"/>
</aop:aspect>
```

```
/*
    后置通知，在目标方法执行之后执行，返回值参数的名称与 xml 中保护一致
*/
public void commit(JoinPoint joinpoint, Object returnVal){
    System.out.println("目标方法返回值: " + returnVal);
    System.out.println("Commit Transaction");
}
```

3.最终通知

```
/*
    最终通知
    无论目标方法是否发出异常都将执行
*/
public void finallyMethod(JoinPoint joinpoint){
    System.out.println("Finally Method");
}
```

```
<!-- 最终通知 -->
<aop:after method="finallyMethod" pointcut-ref="pointcut"/>
```

4.异常通知

```
/*
    异常通知
    接受目标方法抛出的异常
*/
public void throwingMethod(JoinPoint joinPoint, Throwable throwable){
    System.out.println("异常: " + throwable.getMessage());
}
```

```
<!--异常通知-->
<aop:after-throwing method="throwingMethod" pointcut-ref="pointcut" thr
owing="throwable"/>
```

5.环绕通知

```
/*
    环绕通知
    ProceedingJoinPoint: 子接口
    控制目标方法的执行
    前置通知和后置通知也能在目标方法的前后添加内容,但是不能控制目标方法的执行
*/
public void around(ProceedingJoinPoint joinPoint) throws Throwable {
    System.out.println("before..before..before");
    joinPoint.proceed(); // 调用目标方法
    System.out.println("after..after..after");
}
```

```
<!-- 环绕通知 -->
    <aop:around method="around" pointcut-ref="pointcut"/>
</aop:aspect>
```

4.2.5 SpringAOP 细节

1. 如果目标类实现了接口,则采用 JDKProxy; 如果没有实现接口,采用 CGLIBProxy [Spring 内部做的]
2. 目标类实现了接口,但还想要采用 CGLIBProxy,作如下更改:

```
<aop:config proxy-target-class="true">...</aop:config>
```

5. Spring 与 Struts2 整合

先写一个经典的三层架构

```
public interface UserDao {
    public void add();
}

-----

public class UserDaoImpl implements UserDao {
    @Override
```

```

        public void add() {
            System.out.println("添加成功~");
        }
    }

    -----

    public interface UserService {
        public void register();
    }

    -----

    public class UserServiceImpl implements UserService {
        private UserDao dao;
        public void setDao(UserDao dao) {
            this.dao = dao;
        }

        @Override
        public void register() {
            dao.add();
        }
    }

```

applicationContext.xml:

```

<bean name="userDao" class="com.lanou.dao.impl.UserDaoImpl"/>

<bean name="userService" class="com.lanou.service.impl.UserServiceImpl"
>
    <property name="dao" ref="userDao"/>
</bean>

```

在应用启动时默认加载 spring 的 applicationContext 配置文件,所以在 web.xml 中进行注册监听器

```

<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListen
er</listener-class>
</listener>

```


这个 listener 在 spring-web jar 包中,如果没有需要导入 jar 包

5.1 方式一

动作类还是 Struts2 负责管理,只向 spring 容器要service 实例

```
public class UserAction extends ActionSupport {
    // 需要和 spring 中的 bean 名字保持一致!
    private UserService userService;

    public void setUserService(UserService userService) {
        this.userService = userService;
    }

    public String register() {
        userService.register();
        return SUCCESS;
    }
}
```

struts.xml:

```
<package name="p1" extends="struts-default">
    <action name="register" class="com.lanou.web.action.UserAction" method="register">
        <result name="success">/success.jsp</result>
    </action>
</package>
```

但是这时候访问 register ,会出现空指针,因为 userService 不存在,它是由 spring 的 BeanFactory 管理的,而 userAction 是由 ObjectFactory 管理的,ObjectFactory目前不能够从 Spring 容器获取userService.

解决办法:

添加 struts2和spring 的插件,在 struts2的包中: **struts2/lib/struts2-spring-plugin.jar**

在struts.xml 配置文件中替换类

```
<!-- 替换默认的产生动作实例的工厂为 spring -->
<constant name="struts.objectFactory" value="spring"/>
```

但是不需要做. 拷贝的 jar 包中有 struts-plugin.xml, 里面有这项配置

5.2 方式二

动作类也交给Spring 管理, 这时不需要 struts2-spring-plugin.jar

applicationContext.xml:

```
<!-- 实例化动作类 -->
<bean name="registAction" class="com.lanou.web.action.UserRegistAction"
      scope="prototype">
    <property name="userService" ref="userService"/>
</bean>
```

struts.xml:

将class修改成applicationContext.xml中配置 bean 的name

```
<package name="p1" extends="struts-default">
    <action name="register" class="registerAction" method="register">
        <result name="success">/success.jsp</result>
    </action>
</package>
```

6. Spring 与 Hibernate 整合

Spring整合Hibernate有什么好处?

- 1、由IoC容器来管理Hibernate的SessionFactory
- 2、让Hibernate使用Spring的声明式事务

事务: 作为切面 spring 容器

通知: 关于事务的操作

开启事务, 事务提交, 事务回滚

目标类: service 类

目标方法: service 的方法

代理对象代理方法: 目标方法+事务

步骤

1. 创建持久化类, 映射文件
2. 在 spring 中引入 sessionFactory[其中有 datasource]
3. 创建 dao/daoImpl , service/serviceImpl, 将 dao,service 写入 spring 中
4. spring 事务配置 aop

[1] 创建持久化类, 映射文件

[2] 在 spring 中引入 sessionFactory

[其中 datasource 使用c3p0 (别忘了添加 c3p0 和 mySql jar 包)]

将四大参数配置到 properties 中

```
jdbc.driver_class=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/java1
jdbc.username=root
jdbc.password=123456
```

配置 datasource:

```
<!-- 引入 properties 的配置文件 -->
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations" value="classpath:jdbc.properties"/>
</bean>

<!-- 配置 c3p0 连接 -->
<bean id="datasource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
    destroy-method="close">
    <property name="driverClass" value="${jdbc.driver_class}"/>
    <property name="jdbcUrl" value="${jdbc.url}"/>
    <property name="user" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

配置 sessionFactory:

第一种方式添加 hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- 连接数据库的驱动 [datasource 中c3p0配置过了] -->
        <!--<property name="connection.driver_class">com.mysql.jdbc.Dri
```



```

ver</property>-->
    <!--<property name="connection.url">jdbc:mysql://localhost:3306
/hibernate</property>-->
    <!--<property name="connection.username">root</property>-->
    <!--<property name="connection.password">123456</property>-->

    <property name="dialect">org.hibernate.dialect.MySQLDialect</pr
operty>

    <property name="show_sql">true</property>

    <property name="hbm2ddl.auto">update</property>

    <!-- 格式化 sql 语句 -->
    <property name="format_sql">true</property>

    <mapping resource="com/lanou/domain/Person.hbm.xml"/>

</session-factory>
</hibernate-configuration>

```

```

<bean id="sessionFactory" class="org.springframework.orm.hibernate5.Loc
alSessionFactoryBean">
    <property name="dataSource" ref="datasource"/>
    <property name="configLocation">
<value>classpath:hibernate.cfg.xml</value>
    </property>
</bean>

```

第二种方式: 不需要hibernate.cfg.xml, 在sessionFactory中直接配置:

```

<!-- 配置SessionFactory -->
<bean id="sessionFactory2" class="org.springframework.orm.hibernate5.Lo
calSessionFactoryBean">
    <property name="dataSource" ref="datasource"/>
    <property name="hibernateProperties">
        <props>
            <prop key="show_sql">true</prop>
            <prop key="format_sql">true</prop>
            <prop key="dialect">org.hibernate.dialect.MySQLDialect</pro
p>
            <prop key="hibernate.current_session_context_class">org.spr
ingframework.orm.hibernate5.SpringSessionContext</prop>
        </props>
    </property>
    <!-- //加载实体类的映射文件位置及名称 -->
    <property name="mappingLocations" value="classpath:com/lanou/domain

```



```
/*.hbm.xml"></property>
</bean>
```

[4] 创建 dao/daoImpl , service/serviceImpl, 将 dao,service 写入 spring 中

```
public class UserDaoImpl extends HibernateDaoSupport implements UserDao
{
    @Override
    public void addUser() {
        System.out.println("add");
        this.getHibernateTemplate().save(new User("林教练", "炸死你!"));
        // this.getSessionFactory()
    }
}
```

```
<bean id="userDao" class="com.lanou.spring.dao.impl.UserDaoImpl">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
<bean id="userService" class="com.lanou.spring.service.impl.UserServiceImpl">
    <property name="userDao" ref="userDao"/>
</bean>
```

[5] spring 事务配置 aop

事务切面类由 spring 提供, 我们不需要配置

PlatformTransactionManager 最顶级接口

有一个子类: AbstractPlatformTransactionManager

实现类: HibernateTransactionManager / DataSourceTransactionManager

在 AbstractPlatformTransactionManager 的 commit 中

有异常就回滚 processRollback 具体处理回滚的消息要交给具体的事务管理器, 比如
HibernateTransactionManager

没有异常就提交 processCommit 具体提交交给具体的事务管理器

```
<!-- 事务的声明 -->
<!-- 事务管理器 -->
<bean id="transactionManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

```

<!-- 事务属性进行配置 -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <tx:attributes>
    <!--
      isolation: 事务隔离级别, 处理事务并发问题
      DEFAULT: 当前数据库默认的隔离级别
      propagation: 事务的传播属性.
      REQUIRED: 当前的方法必须要在事务中进行.如果有实物环境就加入, 如
      果没有的话,就新建事务. 默认值
      read-only: 是否只读
    -->
    <tx:method name="add*" isolation="DEFAULT" propagation="REQ
UIRED" read-only="false"/>
    <tx:method name="update*"/>
    <tx:method name="*" read-only="true"/>
  </tx:attributes>
</tx:advice>
<!-- 配置事务切点, 并把切点和事务属性关联起来 -->
<!-- aop 设置-->
<!-- 目标类: service -->
<aop:config>
  <aop:pointcut id="txPointcut" expression="execution(* com.lanou
.*.service.impl.*(..))"/>

  <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut"/>
</aop:config>

```

7. S2SH 框架整合

7.1. 各个框架扮演的角色

Struts2: MVC

Spring :

1.使用 IoC 和 DI 实现完全的面向接口编程,在 Action 层为了做到完全的面向接口编程,必须让 action 的实例由 spring 容器产生

2.声明式事务处理: 不需要再管事务了.

Hibernate: 完成数据库操作

7.2. 整合的原则

谁能先测试,先写谁. 写一步对一步

[1] 持久化类,映射文件

- [2] 引入 sessionFactory
- [3] dao, service 层
- [4] spring 的声明式事务处理
- [5] 写 action和前台

7.3. 步骤

- [1] 创建 web 工程, 选择 Spring, Struts2, Hibernate . 导入其他 jar 包
- [2] 三个资源文件夹: src / config / test
- [3] 创建持久化类和映射文件 com.lanou.domain : Person + Person.hbm.xml
- [4] 写 spring 配置文件 . 配置文件不能够只有一个 ac-db.xml
- [5] 测试 test/com.lanou.s2sh.test/ SessionFactoryTest , SpringUtil
- [6] 创建 dao 层和 service 层类和接口
- [7] 写 spring 声明式事务处理的配置, 并把dao, service 放入 spring 中 , ac-person.xml
- [8] 声明式事务测试
- [9] 创建 action, 把 action 放进 spring 容器中. scope=prototype, 测试 action 创建
- [10] action的 struts 配置, struts.xml(src) / struts-person.xml
- [11] web.xml

7.4 启动流程

1. Tomcat 启动时候

Struts2 容器:

- 加载default.properties 配置文件
- struts-default.xml
- struts-plugin.xml
 - 存在 struts2-spring-plugin.jar
 - struts-plugin.xml: struts2的 action 产生, buildBean
- struts.xml
- web.xml
 - contextLoaderListener: 初始化spring容器
 - contextConfigLocation: 配置文件路径, 没有会去默认路径查找 WEB-INF

spring容器启动后

- 创建了 dao和service层的对象和代理对象, 但是 action 没有创建

