Spring核心技术

# Spring的概述

Spring框架，可以解决对象创建以及对象之间依赖关系的一种框架。

且可以和其他框架一起使用；Spring与Struts, Spring与hibernate

(起到整合（粘合）作用的一个框架)

Spring提供了一站式解决方案：

1） Spring Core spring的核心功能： IOC容器, 解决对象创建及依赖关系

2） Spring Web Spring对web模块的支持。

-🡪 可以与struts整合,让struts的action创建交给spring

-🡪 spring mvc模式

3） Spring DAO Spring 对jdbc操作的支持 【JdbcTemplate模板工具类】

4） Spring ORM spring对orm的支持：

🡪 既可以与hibernate整合，【session】

🡪 也可以使用spring的对hibernate操作的封装

5）Spring AOP 切面编程

6）SpringEE spring 对javaEE其他模块的支持

# Spring的环境搭建

## spring环境需要maven 坐标

|  |
| --- |
| **<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"***  **xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>**  **<modelVersion>4.0.0</modelVersion>**  **<groupId>com.itmayiedu</groupId>**  **<artifactId>itmayiedu-spring</artifactId>**  **<version>0.0.1-SNAPSHOT</version>**  **<dependencies>**  **<!-- 引入Spring-AOP等相关Jar -->**  **<dependency>**  **<groupId>org.springframework</groupId>**  **<artifactId>spring-core</artifactId>**  **<version>3.0.6.RELEASE</version>**  **</dependency>**  **<dependency>**  **<groupId>org.springframework</groupId>**  **<artifactId>spring-context</artifactId>**  **<version>3.0.6.RELEASE</version>**  **</dependency>**  **<dependency>**  **<groupId>org.springframework</groupId>**  **<artifactId>spring-aop</artifactId>**  **<version>3.0.6.RELEASE</version>**  **</dependency>**  **<dependency>**  **<groupId>org.springframework</groupId>**  **<artifactId>spring-orm</artifactId>**  **<version>3.0.6.RELEASE</version>**  **</dependency>**  **<dependency>**  **<groupId>org.aspectj</groupId>**  **<artifactId>aspectjrt</artifactId>**  **<version>1.6.1</version>**  **</dependency>**  **<dependency>**  **<groupId>aspectj</groupId>**  **<artifactId>aspectjweaver</artifactId>**  **<version>1.5.3</version>**  **</dependency>**  <dependency>  <groupId>cglib</groupId>  <artifactId>cglib</artifactId>  <version>2.1\_2</version>  </dependency>  **</dependencies>**  **</project>** |

## 需要交给Spring管理注入类

|  |
| --- |
| **public class UserEntity {**  **private String name;**  **private Integer age;**  **public String getName() {**  **return name;**  **}**  **public void setName(String name) {**  **this.name = name;**  **}**  **public Integer getAge() {**  **return age;**  **}**  **public void setAge(Integer age) {**  **this.age = age;**  **}**  **}** |

## spring配置文件

|  |
| --- |
| **<?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:p=*"http://www.springframework.org/schema/p"***  **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/context***  ***http://www.springframework.org/schema/context/spring-context.xsd***  ***http://www.springframework.org/schema/aop***  ***http://www.springframework.org/schema/aop/spring-aop.xsd"*>**  **<bean id=*"userEntity"* class=*"com.itmayiedu.entity.UserEntity"* />**  **</beans>** |

## 测试类

|  |
| --- |
| **public class SpringTest {**  **public static void main(String[] args) {**  **ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext(**  **"applicationContext.xml");**  **UserEntity userEntity = (UserEntity) applicationContext.getBean("userEntity");**  **System.*out*.println(userEntity);**  **}**  **}** |

## 验证问题，spring bean id重复会怎么办？

# Spring 加载过程

## spring是单例还是多例？

## spring作用域

### singleton 作用域

当一个bean的 作用域设置为singleton, 那么Spring IOC容器中只会存在一个共享的bean实例，并且所有对bean的请求，只要id与该bean定义相匹配，则只会返回bean的同一实例。换言之，当把 一个bean定义设置为singleton作用域时，Spring IOC容器只会创建该bean定义的唯一实例。这个单一实例会被存储到单例缓存（singleton cache）中，并且所有针对该bean的后续请求和引用都 将返回被缓存的对象实例，这里要注意的是singleton作用域和GOF设计模式中的单例是完全不同的，单例设计模式表示一个ClassLoader中 只有一个class存在，而这里的singleton则表示一个容器对应一个bean，也就是说当一个bean被标识为singleton时 候，spring的IOC容器中只会存在一个该bean。

### Prototype

prototype作用域部署的bean，每一次请求（将其注入到另一个bean中，或者以程序的方式调用容器的 getBean()方法）都会产生一个新的bean实例，相当与一个new的操作，对于prototype作用域的bean，有一点非常重要，那就是Spring不能对一个prototype bean的整个生命周期负责，容器在初始化、配置、装饰或者是装配完一个prototype实例后，将它交给客户端，随后就对该prototype实例不闻不问了。不管何种作用域，容器都会调用所有对象的初始化生命周期回调方法，而对prototype而言，任何配置好的析构生命周期回调方法都将不会被调用。 清除prototype作用域的对象并释放任何prototype bean所持有的昂贵资源，都是客户端代码的职责。（让Spring容器释放被singleton作用域bean占用资源的一种可行方式是，通过使用 bean的后置处理器，该处理器持有要被清除的bean的引用。）

### request

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，配置实例：

request、session、global session使用的时候首先要在初始化web的web.xml中做如下配置：

### 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

# SpringIOC 容器

SpringIOC容器，是spring核心内容。

作用： 创建对象 & 处理对象的依赖关系

## 容器创建对象：

创建对象, 有几种方式：

1） 调用无参数构造器

2） 带参数构造器

3） 工厂创建对象

工厂类，静态方法创建对象

工厂类，非静态方法创建对象

|  |
| --- |
| **<!-- 无参构造函数 -->**  **<bean id=*"user1"* class=*"com.itmayiedu.entity.UserEntity"* scope=*"prototype"* />**  **<!-- 有参构造函数 -->**  **<bean id=*"user2"* class=*"com.itmayiedu.entity.UserEntity"*>**  **<constructor-arg name=*"name"* type=*"java.lang.String"***  **value=*"张三"*></constructor-arg>**  **<constructor-arg name=*"age"* type=*"java.lang.Integer"***  **value=*"18"*></constructor-arg>**  **</bean>**  **<bean id=*"factory"* class=*"com.itmayiedu.entity.ObjectFactory"*></bean>**  **<!-- 通过实例工厂方法创建 -->**  **<bean id=*"user3"* factory-bean=*"factory"* factory-method=*"getInstance"*></bean>**  **<!-- 通过静态工厂方法创建 -->**  **<bean id=*"user4"* class=*"com.itmayiedu.entity.ObjectFactory"***  **factory-method=*"getStaticInstance"*></bean>** |

## 依赖注入：

Spring中，如何给对象的属性赋值? 【DI, 依赖注入】

1) 通过构造函数

2) 通过set方法给属性注入值

3) p名称空间

4) 注解

#### # (常用)Set方法注入值

|  |
| --- |
| <!-- dao instance -->  <bean id=*"userDao"* class=*"cn.itmayiedu.UserDao"*></bean>  <!-- service instance -->  <bean id=*"userService"* class=*"cn.itmayiedu.UserService"*>  <property name=*"userDao"* ref=*"userDao"*></property>  </bean>    <!-- action instance -->  <bean id=*"userAction"* class=*"cn.itmayiedu.UserAction"*>  <property name=*"userService"* ref=*"userService"*></property>  </bean> |

**同学们 思考下，为什么需要生成set方法才能赋值？**

#### # p 名称空间注入属性值 (优化)

|  |
| --- |
| <?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:p=*"http://www.springframework.org/schema/p"*  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"*>    <!-- ###############对象属性赋值############### -->    <!--  给对象属性注入值：  # p 名称空间给对象的属性注入值  (spring3.0以上版本才支持)  -->  <bean id=*"userDao"* class=*"cn.itmayiedu.UserDao"*></bean>    <bean id=*"userService"* class=*"cn.itmayiedu.UserService"* p:userDao-ref=*"userDao"*></bean>    <bean id=*"userAction"* class=*"cn.itmayiedu.UserAction"* p:userService-ref=*"userService"*></bean>      <!-- 传统的注入：  <bean id="user" class="cn.itmayiedu.User" >  <property name="name" value="xxx"></property>  </bean>  -->  <!-- p名称空间优化后 -->  <bean id=*"user"* class=*"cn.itmayiedu.User"* p:name=*"Jack0001"*></bean>    </beans> |

#### # 注解版本使用

注解方式可以简化spring的IOC容器的配置!

使用注解步骤：

1）先引入context名称空间

xmlns:context="http://www.springframework.org/schema/context"

2）开启注解扫描

<context:component-scan base-package="com.itmayiedu02"></context:component-scan>

3）使用注解

通过注解的方式，把对象加入ioc容器。

创建对象以及处理对象依赖关系，相关的注解：

@Component 指定把一个对象加入IOC容器

@Repository 作用同@Component； 在持久层使用

@Service 作用同@Component； 在业务逻辑层使用

@Controller 作用同@Component； 在控制层使用

@Resource 属性注入

总结：

1） 使用注解，可以简化配置，且可以把对象加入IOC容器,及处理依赖关系(DI)  
 2） 注解可以和XML配置一起使用。

## @Resource与@Autowired区别

1、@Autowired与@Resource都可以用来装配bean. 都可以写在字段上,或写在setter方法上。   
2、@Autowired默认按类型装配（这个注解是属业spring的），默认情况下必须要求依赖对象必须存在，如果要允许null 值，可以设置它的required属性为false，如：@Autowired(required=false) ，如果我们想使用名称装配可以结合@Qualifier注解进行使用，如下：

**Java代码  [收藏代码](javascript:void())**

1. @Autowired() @Qualifier("baseDao")
2. **private** BaseDao baseDao;

 3、@Resource（这个注解属于J2EE的），默认安照名称进行装配，名称可以通过name属性进行指定，   
如果没有指定name属性，当注解写在字段上时，默认取字段名进行按照名称查找，如果注解写在setter方法上默认取属性名进行装配。 当找不到与名称匹配的bean时才按照类型进行装配。但是需要注意的是，如果name属性一旦指定，就只会按照名称进行装配。

**Java代码  [收藏代码](javascript:void())**

1. @Resource(name="baseDao")
2. **private** BaseDao baseDao;

我喜欢用 @Resource注解在字段上，且这个注解是属于J2EE的，减少了与spring的耦合。最重要的这样代码看起就比较优雅。

# 代理模式

## 概述

代理（Proxy）是一种设计模式， 提供了对目标对象另外的访问方式；即通过代理访问目标对象。 这样好处： 可以在目标对象实现的基础上，增强额外的功能操作。(扩展目标对象的功能)。

举例：明星（邓紫棋）🡨---经纪人<-------用户

目标 （代理）



代理模式的关键点： 代理对象与目标对象。

## 静态代理

静态代理，

1） 代理对象，要实现与目标对象一样的接口；

2） 举例:

保存用户(模拟)

Dao , 直接保存

DaoProxy, 给保存方法添加事务处理

总结静态代理：

1）可以做到在不修改目标对象的功能前提下，对目标对象功能扩展。

2）缺点：

--》 因为代理对象，需要与目标对象实现一样的接口。所以会有很多代理类，类太多。

--》 一旦接口增加方法，目标对象与代理对象都要维护。

解决：

代理工厂？ 可以使用动态代理。

## 动态代理

动态代理，

1）代理对象，不需要实现接口；

2）代理对象的生成，是利用JDKAPI， 动态的在内存中构建代理对象(需要我们指定创建 代理对象/目标对象 实现的接口的类型；);

3) 动态代理， JDK代理， 接口代理；

JDK中生成代理对象的API：

|-- Proxy

static Object newProxyInstance(

ClassLoader loader, 指定当前目标对象使用类加载器

Class<?>[] interfaces, 目标对象实现的接口的类型

InvocationHandler h 事件处理器

)

动态代理总结：

代理对象不需要实现接口，但是目标对象一定要实现接口；否则不能用动态代理！  
 (class $Proxy0 implements IuserDao)

思考：

有一个目标对象，想要功能扩展，但目标对象没有实现接口，怎样功能扩展？

Class UserDao{}

// 子类的方式

Class subclass extends UserDao{}

以子类的方式实现(cglib代理)

## Cglib代理

Cglib代理，也叫做子类代理。在内存中构建一个子类对象从而实现对目标对象功能的扩展。

* **JDK的动态代理有一个限制，就是使用动态代理的对象必须实现一个或多个接口。如果想代理没有实现接口的类，就可以使用CGLIB实现。**
* **CGLIB是一个强大的高性能的代码生成包，它可以在运行期扩展Java类与实现Java接口。它广泛的被许多AOP的框架使用，例如Spring AOP和dynaop，为他们提供方法的interception（拦截）。**
* **CGLIB包的底层是通过使用一个小而快的字节码处理框架ASM，来转换字节码并生成新的类。不鼓励直接使用ASM，因为它要求你必须对JVM内部结构包括class文件的格式和指令集都很熟悉。**

Cglib子类代理：

1) 需要引入cglib – jar文件， 但是spring的核心包中已经包括了cglib功能，所以直接引入spring-core-3.2.5.jar即可。

2）引入功能包后，就可以在内存中动态构建子类

3）代理的类不能为final， 否则报错。

4） 目标对象的方法如果为final/static, 那么就不会被拦截，即不会执行目标对象额外的业务方法。

在Spring的AOP编程中，

如果加入容器的目标对象有实现接口，用JDK代理；

如果目标对象没有实现接口，用Cglib代理；

# 手动实现AOP编程

AOP 面向切面的编程，

AOP可以实现“业务代码”与“关注点代码”分离

|  |
| --- |
| **// 保存一个用户**  **public void add(User user) {**  **Session session = null;**  **Transaction trans = null;**  **try {**  **session = HibernateSessionFactoryUtils.getSession(); // 【关注点代码】**  **trans = session.beginTransaction(); // 【关注点代码】**    **session.save(user); // 核心业务代码**    **trans.commit(); //…【关注点代码】**  **} catch (Exception e) {**  **e.printStackTrace();**  **if(trans != null){**  **trans.rollback(); //..【关注点代码】**  **}**  **} finally{**  **HibernateSessionFactoryUtils.closeSession(session); ////..【关注点代码】**  **}**  **}** |

分析总结：

关注点代码，就是指重复执行的代码。

**业务代码与关注点代码分离，好处？**

--🡪 关注点代码写一次即可；

-🡪开发者只需要关注核心业务；

-🡪运行时期，执行核心业务代码时候动态植入关注点代码； 【代理】

如何分离？

过程式/对象式/代理模式分离

# AOP编程

## 概述：

AOP: Aspect Oriented Programming 面向切面编程。  
　　面向切面编程(也叫面向方面)：Aspect Oriented Programming(AOP),是目前软件开发中的一个热点。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。  
　　AOP是OOP的延续，是（Aspect Oriented Programming）的缩写，意思是面向切面（方面）编程。  
　　主要的功能是：**日志记录，性能统计，安全控制，事务处理，异常处理**等等。  
　　主要的意图是：将日志记录，性能统计，安全控制，事务处理，异常处理等代码从业务逻辑代码中划分出来，通过对这些行为的分离，我们希望可以将它们独立到非指导业务逻辑的方法中，进而改  变这些行为的时候不影响业务逻辑的代码。

　　可以通过预编译方式和运行期动态代理实现在不修改源代码的情况下给程序动态统一添加功能的一种技术。AOP实际是GoF设计模式的延续，设计模式孜孜不倦追求的是调用者和被调用者之间的解耦，AOP可以说也是这种目标的一种实现。

假设把应用程序想成一个立体结构的话，OOP的利刃是纵向切入系统，把系统划分为很多个模块（如：用户模块，文章模块等等），而AOP的利刃是横向切入系统，提取各个模块可能都要重复操作的部分（如：权限检查，日志记录等等）。由此可见，AOP是OOP的一个有效补充。

注意：AOP不是一种技术，实际上是编程思想。凡是符合AOP思想的技术，都可以看成是AOP的实现。

**Aop，** aspect object programming 面向切面编程

功能： 让关注点代码与业务代码分离！

**关注点,**

重复代码就叫做关注点；

**切面，**

关注点形成的类，就叫切面(类)！

面向切面编程，就是指 对很多功能都有的重复的代码抽取，再在运行的时候网业务方法上动态植入“切面类代码”。

**切入点，**

执行目标对象方法，动态植入切面代码。

可以通过切入点表达式，指定拦截哪些类的哪些方法； 给指定的类在运行的时候植入切面类代码。

## 注解方式实现AOP编程

<aop:aspectj-autoproxy></aop:aspectj-autoproxy> 开启事物注解权限

4) 使用注解

@Aspect 指定一个类为切面类

@Pointcut("execution(\* com.itmayiedu.service.UserService.add(..))") 指定切入点表达式

@Before("pointCut\_()") 前置通知: 目标方法之前执行

@After("pointCut\_()") 后置通知：目标方法之后执行（始终执行）

@AfterReturning("pointCut\_()") 返回后通知： 执行方法结束前执行(异常不执行)

@AfterThrowing("pointCut\_()") 异常通知: 出现异常时候执行

@Around("pointCut\_()") 环绕通知： 环绕目标方法执行

|  |
| --- |
| **@Component**  **@Aspect**  **public class Aop {**  **@Before("execution(\* com.itmayiedu.service.UserService.add(..))")**  **public void begin() {**  **System.*out*.println("前置通知");**  **}**  **@After("execution(\* com.itmayiedu.service.UserService.add(..))")**  **public void commit() {**  **System.*out*.println("后置通知");**  **}**  **@AfterReturning("execution(\* com.itmayiedu.service.UserService.add(..))")**  **public void afterReturning() {**  **System.*out*.println("运行通知");**  **}**  **@AfterThrowing("execution(\* com.itmayiedu.service.UserService.add(..))")**  **public void afterThrowing() {**  **System.*out*.println("异常通知");**  **}**  **@Around("execution(\* com.itmayiedu.service.UserService.add(..))")**  **public void around(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {**  **System.*out*.println("我是环绕通知-前");**  **proceedingJoinPoint.proceed();**  **System.*out*.println("我是环绕通知-后");**  **}**  **}** |

## XML方式实现AOP编程

Xml实现aop编程：

1） 引入jar文件 【aop 相关jar， 4个】

2） 引入aop名称空间

3）aop 配置

\* 配置切面类 （重复执行代码形成的类）

\* aop配置

拦截哪些方法 / 拦截到方法后应用通知代码

|  |
| --- |
| <?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:p=*"http://www.springframework.org/schema/p"*  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/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd"*>    <!-- dao 实例 -->  <bean id=*"userDao"* class=*"com.itmayiedu.UserDao"*></bean>  <bean id=*"orderDao"* class=*"com.itmayiedu.OrderDao"*></bean>    <!-- 切面类 -->  <bean id=*"aop"* class=*"com.itmayiedu.Aop"*></bean>    <!-- Aop配置 -->  <aop:config>  <!-- 定义一个切入点表达式： 拦截哪些方法 -->  <aop:pointcut expression=*"execution(\* com.itmayiedu.\*.\*(..))"* id=*"pt"*/>  <!-- 切面 -->  <aop:aspect ref=*"aop"*>  <!-- 环绕通知 -->  <aop:around method=*"around"* pointcut-ref=*"pt"*/>  <!-- 前置通知： 在目标方法调用前执行 -->  <aop:before method=*"begin"* pointcut-ref=*"pt"*/>  <!-- 后置通知： -->  <aop:after method=*"after"* pointcut-ref=*"pt"*/>  <!-- 返回后通知 -->  <aop:after-returning method=*"afterReturning"* pointcut-ref=*"pt"*/>  <!-- 异常通知 -->  <aop:after-throwing method=*"afterThrowing"* pointcut-ref=*"pt"*/>    </aop:aspect>  </aop:config>  </beans> |

# 事物的概述

## ⑴ 原子性（Atomicity）

　　原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

## ⑵ 一致性（Consistency）

　　一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

　　拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

## ⑶ 隔离性（Isolation）

　　隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

　　即要达到这么一种效果：对于任意两个并发的事务T1和T2，在事务T1看来，T2要么在T1开始之前就已经结束，要么在T1结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

　　关于事务的隔离性数据库提供了多种隔离级别，稍后会介绍到。

## ⑷ 持久性（Durability）

　　持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

　　例如我们在使用JDBC操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务以及正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成，否则就会造成我们看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。

# 程序中事务控制

## 环境准备

用户访问—》C--》 Service---》Dao

一个业务的成功： 调用的service是执行成功的，意味着service中调用的所有的dao是执行成功的。  **事务应该在Service层统一控制。**

1）没有应用事务的代码：

2）模拟：

在service中调用2次dao， 希望其中一个dao执行失败，整个操作要回滚。

开发步骤：

1. 后台环境准备

数据库、表/entity/dao/service

2. dao 的实现用JdbcTemplate

3. 对象创建都有Spring容器完成

|  |
| --- |
|  |

# 程序中事务控制

## 事务控制概述

**编程式事务控制**

自己手动控制事务，就叫做编程式事务控制。

Jdbc代码：

Conn.setAutoCommite(false); // 设置手动控制事务

Hibernate代码：

Session.beginTransaction(); // 开启一个事务

【细粒度的事务控制： 可以对指定的方法、指定的方法的某几行添加事务控制】

(比较灵活，但开发起来比较繁琐： 每次都要开启、提交、回滚.)

**声明式事务控制**

Spring提供了对事务的管理, 这个就叫声明式事务管理。

Spring提供了对事务控制的实现。用户如果想用Spring的声明式事务管理，只需要在配置文件中配置即可； 不想使用时直接移除配置。这个实现了对事务控制的最大程度的解耦。

Spring声明式事务管理，核心实现就是基于Aop。

【粗粒度的事务控制： 只能给整个方法应用事务，不可以对方法的某几行应用事务。】

(因为aop拦截的是方法。)

Spring声明式事务管理器类：

Jdbc技术：**DataSourceTransactionManager**

Hibernate技术：**HibernateTransactionManager**

## 编程事物管理

### 手动管理事物

|  |
| --- |
| 1. UserDao.java |
| **@Repository**  **public class UserDao {**  **@Autowired**  **private JdbcTemplate jdbcTemplate;**  **public void add(String name, Integer age) {**  **String sql = "INSERT INTO users(NAME, age) VALUES(?,?);";**  **int update = jdbcTemplate.update(sql, name, age);**  **System.*out*.println("updateResult:" + update);**  **}**  **}** |
| 2. UserService |
| **@Service**  **public class UserService {**  **@Autowired**  **private UserDao userDao;**  **public void add() {**  **userDao.add("lisi", 18);**  **int i=1/0;//可能会发生异常**  **userDao.add("yushengjun", 19);**  **}**  **}** |
| 3. App 测试类 |
| **public** **class** UserTest {  **public** **static** **void** main(String[] args) {  ClassPathXmlApplicationContext applicationContext = **new** ClassPathXmlApplicationContext("bean.xml");  UserService userService = (UserService) applicationContext.getBean("userService");  userService.add();  }  } |
| 4.手动事物管理类 |
| @Component  **public** **class** TransactionUtils {  // 事物管理器  @Autowired  **private** DataSourceTransactionManager dataSourceTransactionManager;  **public** TransactionStatus begin() {  TransactionStatus transaction = dataSourceTransactionManager.getTransaction(**new** DefaultTransactionDefinition());  **return** transaction;  }  **public** **void** commit(TransactionStatus transaction) {  dataSourceTransactionManager.commit(transaction);  }  **public** **void** rollback(TransactionStatus transaction) {  dataSourceTransactionManager.rollback(transaction);  }  } |
| 5. bean.xml (Spring务管理配置) |
| **<?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:p=*"http://www.springframework.org/schema/p"***  **xmlns:context=*"http://www.springframework.org/schema/context"***  **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/context***  ***http://www.springframework.org/schema/context/spring-context.xsd***  ***http://www.springframework.org/schema/aop***  ***http://www.springframework.org/schema/aop/spring-aop.xsd***  ***http://www.springframework.org/schema/tx***  ***http://www.springframework.org/schema/tx/spring-tx.xsd"*>**  **<!-- 开启注解 -->**  **<context:component-scan base-package=*"com.itmayiedu"*></context:component-scan>**  **<!-- 1. 数据源对象: C3P0连接池 -->**  **<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>**  **<property name=*"driverClass"* value=*"com.mysql.jdbc.Driver"*></property>**  **<property name=*"jdbcUrl"* value=*"jdbc:mysql://localhost:3306/test"*></property>**  **<property name=*"user"* value=*"root"*></property>**  **<property name=*"password"* value=*"root"*></property>**  **</bean>**  **<!-- 2. JdbcTemplate工具类实例 -->**  **<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>**  **<property name=*"dataSource"* ref=*"dataSource"*></property>**  **</bean>**  **<!-- 配置事物 -->**  **<bean id=*"DataSourceTransactionManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>**  **<property name=*"dataSource"* ref=*"dataSource"*></property>**  **</bean>**  **</beans>** |

## 声明式事务管理

### XML方式实现

|  |
| --- |
| 1. UserDao.java |
| **@Repository**  **public class UserDao {**  **@Autowired**  **private JdbcTemplate jdbcTemplate;**  **public void add(String name, Integer age) {**  **String sql = "INSERT INTO users(NAME, age) VALUES(?,?);";**  **int update = jdbcTemplate.update(sql, name, age);**  **System.*out*.println("updateResult:" + update);**  **}**  **}** |
| 2. UserService |
| **@Service**  **public class UserService {**  **@Autowired**  **private UserDao userDao;**  **public void add() {**  **userDao.add("lisi", 18);**  **int i=1/0;//可能会发生异常**  **userDao.add("yushengjun", 19);**  **}**  **}** |
| 3. App 测试类 |
| **public** **class** UserTest {  **public** **static** **void** main(String[] args) {  ClassPathXmlApplicationContext applicationContext = **new** ClassPathXmlApplicationContext("bean.xml");  UserService userService = (UserService) applicationContext.getBean("userService");  userService.add();  }  } |
| 4.手动事物管理类 |
| @Component  **public** **class** TransactionUtils {  // 事物管理器  @Autowired  **private** DataSourceTransactionManager dataSourceTransactionManager;  **public** TransactionStatus begin() {  TransactionStatus transaction = dataSourceTransactionManager.getTransaction(**new** DefaultTransactionDefinition());  **return** transaction;  }  **public** **void** commit(TransactionStatus transaction) {  dataSourceTransactionManager.commit(transaction);  }  **public** **void** rollback(TransactionStatus transaction) {  dataSourceTransactionManager.rollback(transaction);  }  } |
| 5. bean.xml (Spring务管理配置) |
| **<!-- 开启注解 -->**  **<context:component-scan base-package=*"com.itmayiedu"*></context:component-scan>**  **<!-- 1. 数据源对象: C3P0连接池 -->**  **<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>**  **<property name=*"driverClass"* value=*"com.mysql.jdbc.Driver"*></property>**  **<property name=*"jdbcUrl"* value=*"jdbc:mysql://localhost:3306/test"*></property>**  **<property name=*"user"* value=*"root"*></property>**  **<property name=*"password"* value=*"root"*></property>**  **</bean>**  **<!-- 2. JdbcTemplate工具类实例 -->**  **<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>**  **<property name=*"dataSource"* ref=*"dataSource"*></property>**  **</bean>**  **<!-- 配置事物 -->**  **<bean id=*"dataSourceTransactionManager"***  **class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>**  **<property name=*"dataSource"* ref=*"dataSource"*></property>**  **</bean>**  **<!—配置事物增强-->**  **<tx:advice id=*"txAdvice"* transaction-manager=*"dataSourceTransactionManager"*>**  **<tx:attributes>**  **<tx:method name=*"get\*"* read-only=*"true"* />**  **<tx:method name=*"find\*"* read-only=*"true"* />**  **<tx:method name=*"\*"* read-only=*"false"* />**  **</tx:attributes>**  **</tx:advice>**  **<!-- Aop配置： 拦截哪些方法(切入点表表达式) + 应用上面的事务增强配置 -->**  **<aop:config>**  **<aop:pointcut expression=*"execution(\* com.itmayiedu.service.\*.\*(..))"***  **id=*"pt"* />**  **<aop:advisor advice-ref=*"txAdvice"* pointcut-ref=*"pt"* />**  **</aop:config>** |

#### 使用事物注意事项

事物是程序运行如果没有错误,会自动提交事物,如果程序运行发生异常,则会自动回滚。

如果使用了try捕获异常时.一定要在catch里面手动回滚。

事物手动回滚代码

TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();

### 注解方式实现

使用注解实现Spring的声明式事务管理，更加简单！

步骤：

1） 必须引入Aop相关的jar文件

2） bean.xml中指定注解方式实现声明式事务管理以及应用的事务管理器类

3）在需要添加事务控制的地方，写上: @Transactional

@Transactional注解：

1）应用事务的注解

2）定义到方法上： 当前方法应用spring的声明式事务

3）定义到类上： 当前类的所有的方法都应用Spring声明式事务管理;

4）定义到父类上： 当执行父类的方法时候应用事务。

|  |
| --- |
| Bean.xm |
| **<?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:p=*"http://www.springframework.org/schema/p"***  **xmlns:context=*"http://www.springframework.org/schema/context"***  **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/context***  ***http://www.springframework.org/schema/context/spring-context.xsd***  ***http://www.springframework.org/schema/aop***  ***http://www.springframework.org/schema/aop/spring-aop.xsd***  ***http://www.springframework.org/schema/tx***  ***http://www.springframework.org/schema/tx/spring-tx.xsd"*>**  **<!-- 开启注解 -->**  **<context:component-scan base-package=*"com.itmayiedu"*></context:component-scan>**  **<!-- 1. 数据源对象: C3P0连接池 -->**  **<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>**  **<property name=*"driverClass"* value=*"com.mysql.jdbc.Driver"*></property>**  **<property name=*"jdbcUrl"* value=*"jdbc:mysql://localhost:3306/test"*></property>**  **<property name=*"user"* value=*"root"*></property>**  **<property name=*"password"* value=*"root"*></property>**  **</bean>**  **<!-- 2. JdbcTemplate工具类实例 -->**  **<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>**  **<property name=*"dataSource"* ref=*"dataSource"*></property>**  **</bean>**  **<!-- 配置事物 -->**  **<bean id=*"dataSourceTransactionManager"***  **class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>**  **<property name=*"dataSource"* ref=*"dataSource"*></property>**  **</bean>**  **<!-- 开启注解事物 -->**  **<tx:annotation-driven transaction-manager=*"dataSourceTransactionManager"*/>**  **</beans>** |
| UserService |
| **@Transactional**  **public void add() {**  **try {**  **userDao.add("lisi", 18);**  **int i = 1 / 0;**  **userDao.add("yushengjun", 19);**  **} catch (Exception e) {**  **TransactionAspectSupport.*currentTransactionStatus*().setRollbackOnly();**  **}**    **}** |
|  |

# 传播七种行为

Spring中事务的定义：

Propagation（key属性确定代理应该给哪个方法增加事务行为。这样的属性最重要的部份是传播行为。）有以下选项可供使用：

* PROPAGATION\_REQUIRED--支持当前事务，如果当前没有事务，就新建一个事务。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--支持当前事务，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。

|  |
| --- |
| **@Transactional(**  **readOnly = false, // 读写事务**  **timeout = -1, // 事务的超时时间不限制**  **noRollbackFor = ArithmeticException.class, // 遇到数学异常不回滚**  **isolation = Isolation.*DEFAULT*, // 事务的隔离级别，数据库的默认**  **propagation = Propagation.*REQUIRED* // 事务的传播行为**  **)** |

事务传播行为:

Propagation.REQUIRED

指定当前的方法必须在事务的环境下执行；

如果当前运行的方法，已经存在事务， 就会加入当前的事务；

Propagation.REQUIRED\_NEW

指定当前的方法必须在事务的环境下执行；

如果当前运行的方法，已经存在事务： 事务会挂起； 会始终开启一个新的事务，执行完后； 刚才挂起的事务才继续运行。

举例：

Class Log{

Propagation.REQUIRED

insertLog();

}

Propagation.REQUIRED

Void saveDept(){

insertLog(); // 加入当前事务

**.. 异常, 会回滚**

saveDept();

}

Class Log{

Propagation.REQUIRED\_NEW

insertLog();

}

Propagation.REQUIRED

Void saveDept(){

insertLog(); // 始终开启事务

.. 异常, 日志不会回滚

saveDept();

}

测试步骤：

1）日志表Log\_

2）LogService.java

insertLog();

## 如何查看源码

Spring源码下载https://github.com/spring-projects/spring-framework/tags?after=v3.1.0.RC1

## 源代码结构组织

Build-spring-framework是整个Spring源代码的构建目录，里面是项目的构建脚本，如果要自己动手构建Spring，可以进入这个目录使用ANT进行构建。

l  org.springframework.context是IoC容器的源代码目录

l  org.springframework.aop是AOP实现的源代码目录

l  org.springframework.jdbc是JDBC的源代码部分

l  org.springframework.orm是O/R Mapping对应的源代码实现部分

# SpringIOC源码分析

## IOC初始化

1、 XmlBeanFactory(屌丝IOC)的整个流程

2、 FileSystemXmlApplicationContext 的IOC容器流程

1、高富帅IOC解剖

2、 设置资源加载器和资源定位

3、AbstractApplicationContext的refresh函数载入Bean定义过程：

4、AbstractApplicationContext子类的refreshBeanFactory()方法：

5、AbstractRefreshableApplicationContext子类的loadBeanDefinitions方法：

6、AbstractBeanDefinitionReader读取Bean定义资源：

7、资源加载器获取要读入的资源：

8、XmlBeanDefinitionReader加载Bean定义资源：

9、DocumentLoader将Bean定义资源转换为Document对象：

10、XmlBeanDefinitionReader解析载入的Bean定义资源文件：

11、DefaultBeanDefinitionDocumentReader对Bean定义的Document对象解析：

12、BeanDefinitionParserDelegate解析Bean定义资源文件中的<Bean>元素：

13、BeanDefinitionParserDelegate解析<property>元素：

14、解析<property>元素的子元素：

15、解析<list>子元素：

16、解析过后的BeanDefinition在IoC容器中的注册：

17、DefaultListableBeanFactory向IoC容器注册解析后的BeanDefinition：

## IOC体系

BeanFactory

         Spring Bean的创建是典型的工厂模式，这一系列的Bean工厂，也即IOC容器为开发者管理对象间的依赖关系提供了很多便利和基础服务，在Spring中有许多的IOC容器的实现供用户选择和使用，其相互关系如下：



## BeanFactory

 BeanFactory定义了 IOC 容器的最基本形式，并提供了 IOC 容器应遵守的的最基本的接口，也就是Spring IOC 所遵守的最底层和最基本的编程规范。在  Spring 代码中， BeanFactory 只是个接口，并不是 IOC容器的具体实现，但是 Spring 容器给出了很多种实现，如 DefaultListableBeanFactory 、 XmlBeanFactory 、ApplicationContext 等，都是附加了某种功能的实现。

|  |
| --- |
| **public** **interface** BeanFactory {  //这里是对FactoryBean的转义定义，因为如果使用bean的名字检索FactoryBean得到的对象是工厂生成的对象，    //如果需要得到工厂本身，需要转义  //转义符“&”用来获取FactoryBean本身  String ***FACTORY\_BEAN\_PREFIX*** = "&";  //根据bean的名字进行获取bean的实例，这是IOC最大的抽象方法  Object getBean(String name) **throws** BeansException;  //根据bean的名字和Class类型进行获取Bean的实例,和上面方法不同的是，bean名字和Bean 的class类型不同时候会爆出异常  <T> T getBean(String name, Class<T> requiredType) **throws** BeansException;  <T> T getBean(Class<T> requiredType) **throws** BeansException;  Object getBean(String name, Object... args) **throws** BeansException;  //检测这个IOC容器中是否含有这个Bean  **boolean** containsBean(String name);  //判断这个Bean是不是单利  **boolean** isSingleton(String name) **throws** NoSuchBeanDefinitionException;  //判断这个Bean是不是原型  **boolean** isPrototype(String name) **throws** NoSuchBeanDefinitionException;  //查询指定的bean的名字和Class类型是不是指定的Class类型  **boolean** isTypeMatch(String name, Class targetType) **throws** NoSuchBeanDefinitionException;  //这里对得到bean实例的Class类型  Class<?> getType(String name) **throws** NoSuchBeanDefinitionException;  //这里得到bean的别名，如果根据别名检索，那么其原名也会被检索出来  String[] getAliases(String name);  } |

## BeanDefinition

|  |
| --- |
| 这个接口，可以理解为xml bean元素的数据载体。通过对比xml bean标签的属性列表和BeanDefinition的属性列表一看便知。  我的理解，是解析XML的过程，就是 xml <bean>元素内容 转换为BeanDefinition对象的过程。而且这个接口，支持层级，对应对象的继承。  有一个类BeanDefinitionHolder,BeanDefinitionHolder，根据名称或者别名持有beanDefinition，它承载了name和BeanDefinition的映射信息。  BeanWarpper:  提供对标准javabean的分析和操作方法：单个或者批量获取和设置属性值，获取属性描述符，查询属性的可读性和可写性等。支持属性的嵌套设置，深度没有限制。  AbstractRefreshableApplicationContext的refreshBeanFactory()这个方法 |

|  |
| --- |
| **protected** **final** **void** refreshBeanFactory() **throws** BeansException {  **if** (hasBeanFactory()) {  destroyBeans();  closeBeanFactory();  }  **try** {  DefaultListableBeanFactory beanFactory = createBeanFactory();//创建IOC容器  beanFactory.setSerializationId(getId());  customizeBeanFactory(beanFactory);  loadBeanDefinitions(beanFactory);//载入loadBeanDefinitions  **synchronized** (**this**.beanFactoryMonitor) {  **this**.beanFactory = beanFactory;  }  }  **catch** (IOException ex) {  **throw** **new** ApplicationContextException("I/O error parsing bean definition source for " + getDisplayName(), ex);  }  }  **public abstract class AbstractXmlApplicationContext extends AbstractRefreshableConfigApplicationContext { 实现**  /\*\*  \* Loads the bean definitions via an XmlBeanDefinitionReader.  \* **@see** org.springframework.beans.factory.xml.XmlBeanDefinitionReader  \* **@see** #initBeanDefinitionReader  \* **@see** #loadBeanDefinitions  \*/  @Override  **protected** **void** loadBeanDefinitions(DefaultListableBeanFactory beanFactory) **throws** BeansException, IOException {  // Create a new XmlBeanDefinitionReader for the given BeanFactory.  XmlBeanDefinitionReader beanDefinitionReader = **new** XmlBeanDefinitionReader(beanFactory);  // Configure the bean definition reader with this context's  // resource loading environment.  beanDefinitionReader.setResourceLoader(**this**);  beanDefinitionReader.setEntityResolver(**new** ResourceEntityResolver(**this**));  // Allow a subclass to provide custom initialization of the reader,  // then proceed with actually loading the bean definitions.  initBeanDefinitionReader(beanDefinitionReader);  loadBeanDefinitions(beanDefinitionReader);  }  先调用本类里面的loadBeanDefinitions  **protected** **void** loadBeanDefinitions(XmlBeanDefinitionReader reader) **throws** BeansException, IOException {  Resource[] configResources = getConfigResources();  **if** (configResources != **null**) {  reader.loadBeanDefinitions(configResources);  }  String[] configLocations = getConfigLocations();  **if** (configLocations != **null**) {  reader.loadBeanDefinitions(configLocations);  }  }  委托给reader.loadBeanDefinitions(configLocation);    XmlBeanDefinitionReader 通过XmlBeanDefinitionReader来读取。下面看一下XmlBeanDefinitionReader这个方法，但其实并不在这个类实现这个方法，而是在它的基类里面AbstractBeanDefinitionReader  **public** **int** loadBeanDefinitions(String... locations) **throws** BeanDefinitionStoreException {  Assert.*notNull*(locations, "Location array must not be null");  **int** counter = 0;  **for** (String location : locations) {  counter += loadBeanDefinitions(location);  }  **return** counter;  }  进入到loadBeanDefinitions  **public** **int** loadBeanDefinitions(EncodedResource encodedResource) **throws** BeanDefinitionStoreException {  Assert.*notNull*(encodedResource, "EncodedResource must not be null");  **if** (logger.isInfoEnabled()) {  logger.info("Loading XML bean definitions from " + encodedResource.getResource());  }  Set<EncodedResource> currentResources = **this**.resourcesCurrentlyBeingLoaded.get();  **if** (currentResources == **null**) {  currentResources = **new** HashSet<EncodedResource>(4);  **this**.resourcesCurrentlyBeingLoaded.set(currentResources);  }  **if** (!currentResources.add(encodedResource)) {  **throw** **new** BeanDefinitionStoreException(  "Detected cyclic loading of " + encodedResource + " - check your import definitions!");  }  **try** {  InputStream inputStream = encodedResource.getResource().getInputStream();//获取IO  **try** {  InputSource inputSource = **new** InputSource(inputStream);  **if** (encodedResource.getEncoding() != **null**) {  inputSource.setEncoding(encodedResource.getEncoding());  }  **return** doLoadBeanDefinitions(inputSource, encodedResource.getResource());//这个方法从流中读取  }  **finally** {  inputStream.close();  }  }  **catch** (IOException ex) {  **throw** **new** BeanDefinitionStoreException(  "IOException parsing XML document from " + encodedResource.getResource(), ex);  }  **finally** {  currentResources.remove(encodedResource);  **if** (currentResources.isEmpty()) {  **this**.resourcesCurrentlyBeingLoaded.remove();  }  }  }  进入到doLoadBeanDefinitions Resource IO封装  **protected** **int** doLoadBeanDefinitions(InputSource inputSource, Resource resource)  **throws** BeanDefinitionStoreException {  **try** {  **int** validationMode = getValidationModeForResource(resource);  Document doc = **this**.documentLoader.loadDocument(  inputSource, getEntityResolver(), **this**.errorHandler, validationMode, isNamespaceAware());  **return** registerBeanDefinitions(doc, resource); //解析XML  }  **catch** (BeanDefinitionStoreException ex) {  **throw** ex;  }  **catch** (SAXParseException ex) {  **throw** **new** XmlBeanDefinitionStoreException(resource.getDescription(),  "Line " + ex.getLineNumber() + " in XML document from " + resource + " is invalid", ex);  }  **catch** (SAXException ex) {  **throw** **new** XmlBeanDefinitionStoreException(resource.getDescription(),  "XML document from " + resource + " is invalid", ex);  }  **catch** (ParserConfigurationException ex) {  **throw** **new** BeanDefinitionStoreException(resource.getDescription(),  "Parser configuration exception parsing XML from " + resource, ex);  }  **catch** (IOException ex) {  **throw** **new** BeanDefinitionStoreException(resource.getDescription(),  "IOException parsing XML document from " + resource, ex);  }  **catch** (Throwable ex) {  **throw** **new** BeanDefinitionStoreException(resource.getDescription(),  "Unexpected exception parsing XML document from " + resource, ex);  }  }  进入到registerBeanDefinitions  /\*\*  \* Register the bean definitions contained in the given DOM document.  \* Called by <code>loadBeanDefinitions</code>.  \* <p>Creates a new instance of the parser class and invokes  \* <code>registerBeanDefinitions</code> on it.  \* **@param** doc the DOM document  \* **@param** resource the resource descriptor (for context information)  \* **@return** the number of bean definitions found  \* **@throws** BeanDefinitionStoreException in case of parsing errors  \* **@see** #loadBeanDefinitions  \* **@see** #setDocumentReaderClass  \* **@see** BeanDefinitionDocumentReader#registerBeanDefinitions  \*/  **public** **int** registerBeanDefinitions(Document doc, Resource resource) **throws** BeanDefinitionStoreException {  // Read document based on new BeanDefinitionDocumentReader SPI.  BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();  **int** countBefore = getRegistry().getBeanDefinitionCount();  documentReader.registerBeanDefinitions(doc, createReaderContext(resource));  **return** getRegistry().getBeanDefinitionCount() - countBefore;  }  documentReader.registerBeanDefinitionsXML解析  /\*\*  \* Parses bean definitions according to the "spring-beans" DTD.  \* <p>Opens a DOM Document; then initializes the default settings  \* specified at <code>&lt;beans&gt;</code> level; then parses  \* the contained bean definitions.  \*/  **public** **void** registerBeanDefinitions(Document doc, XmlReaderContext readerContext) {  **this**.readerContext = readerContext;  logger.debug("Loading bean definitions");  Element root = doc.getDocumentElement();  BeanDefinitionParserDelegate delegate = createHelper(readerContext, root);  preProcessXml(root);  parseBeanDefinitions(root, delegate);  postProcessXml(root);  }  **-----遍历节点**  **protected** **void** parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {  **if** (delegate.isDefaultNamespace(root)) {  NodeList nl = root.getChildNodes();  **for** (**int** i = 0; i < nl.getLength(); i++) {  Node node = nl.item(i);  **if** (node **instanceof** Element) {  Element ele = (Element) node;  **if** (delegate.isDefaultNamespace(ele)) {  parseDefaultElement(ele, delegate); //默认解析  }  **else** {  delegate.parseCustomElement(ele);  }  }  }  }  **else** {  delegate.parseCustomElement(root);  }  }  ---判断解析类  **private** **void** parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate) {  **if** (delegate.nodeNameEquals(ele, ***IMPORT\_ELEMENT***)) {  importBeanDefinitionResource(ele);//import类型  }  **else** **if** (delegate.nodeNameEquals(ele, ***ALIAS\_ELEMENT***)) {  processAliasRegistration(ele);//别名方式  }  **else** **if** (delegate.nodeNameEquals(ele, ***BEAN\_ELEMENT***)) {  processBeanDefinition(ele, delegate);//bean解析方式  }  } |

## Bean的解析方式

进入到 AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(ele, beanName, containingBean); 使用反射初始化类

|  |
| --- |
| **public** AbstractBeanDefinition parseBeanDefinitionElement(  Element ele, String beanName, BeanDefinition containingBean) {  **this**.parseState.push(**new** BeanEntry(beanName));  String className = **null**;  **if** (ele.hasAttribute(***CLASS\_ATTRIBUTE***)) {  className = ele.getAttribute(***CLASS\_ATTRIBUTE***).trim();  }  **try** {  String parent = **null**;  **if** (ele.hasAttribute(***PARENT\_ATTRIBUTE***)) {  parent = ele.getAttribute(***PARENT\_ATTRIBUTE***);  }  AbstractBeanDefinition bd = createBeanDefinition(className, parent);  parseBeanDefinitionAttributes(ele, beanName, containingBean, bd);  bd.setDescription(DomUtils.*getChildElementValueByTagName*(ele, ***DESCRIPTION\_ELEMENT***));  parseMetaElements(ele, bd);  parseLookupOverrideSubElements(ele, bd.getMethodOverrides());  parseReplacedMethodSubElements(ele, bd.getMethodOverrides());  parseConstructorArgElements(ele, bd);  parsePropertyElements(ele, bd);  parseQualifierElements(ele, bd);  bd.setResource(**this**.readerContext.getResource());  bd.setSource(extractSource(ele));  **return** bd;  }  **catch** (ClassNotFoundException ex) {  error("Bean class [" + className + "] not found", ele, ex);  }  **catch** (NoClassDefFoundError err) {  error("Class that bean class [" + className + "] depends on not found", ele, err);  }  **catch** (Throwable ex) {  error("Unexpected failure during bean definition parsing", ele, ex);  }  **finally** {  **this**.parseState.pop();  }  **return** **null**;  }  进入到AbstractBeanDefinition bd = createBeanDefinition(className, parent);  **protected** AbstractBeanDefinition createBeanDefinition(String className, String parentName)  **throws** ClassNotFoundException {  **return** BeanDefinitionReaderUtils.*createBeanDefinition*(  parentName, className, **this**.readerContext.getBeanClassLoader());  }  进入到BeanDefinitionReaderUtils.*createBeanDefinition*  **public** **static** AbstractBeanDefinition createBeanDefinition(  String parentName, String className, ClassLoader classLoader) **throws** ClassNotFoundException {  GenericBeanDefinition bd = **new** GenericBeanDefinition();  bd.setParentName(parentName);  **if** (className != **null**) {  **if** (classLoader != **null**) {  bd.setBeanClass(ClassUtils.*forName*(className, classLoader));//使用java反射机制初始化  }  **else** {  bd.setBeanClassName(className);  }  }  **return** bd;  } |

## Bean生命周期分析

1. spring对bean进行实例化,默认bean是单例  
   2)spring对bean进行依赖注入  
   3)如果bean实现了BeanNameAware接口,spring将bean的id传给setBeanName()方法  
   4)如果bean实现了BeanFactoryAware接口,spring将调用setBeanFactory方法,将BeanFactory实例传进来  
   5)如果bean实现了ApplicationContextAware()接口,spring将调用setApplicationContext()方法将应用上下文的引用传入  
   6) 如果bean实现了BeanPostProcessor接口,spring将调用它们的postProcessBeforeInitialization接口方法  
   7) 如果bean实现了InitializingBean接口,spring将调用它们的afterPropertiesSet接口方法,类似的如果bean使用了init-method属性声明了初始化方法,改方法也会被调用  
   8)如果bean实现了BeanPostProcessor接口,spring将调用它们的postProcessAfterInitialization接口方法  
   9)此时bean已经准备就绪,可以被应用程序使用了,他们将一直驻留在应用上下文中,直到该应用上下文被销毁  
   10)若bean实现了DisposableBean接口,spring将调用它的distroy()接口方法。同样的,如果bean使用了destroy-method属性声明了销毁方法,则该方法被调用

|  |
| --- |
| 1. public class UserEntity 2. implements BeanFactoryAware, BeanNameAware, InitializingBean, DisposableBean, ApplicationContextAware { 3. private String userName; 4. private Integer age = null; 5. public UserEntity() { 6. System.*out*.println("无惨构造函数....."); 7. } 8. public UserEntity(String userName, Integer age) { 9. System.*out*.println("我是有参构造函数 userName:" + userName + ",age:" + age); 10. this.userName = userName; 11. this.age = age; 12. } 13. public String getUserName() { 14. return userName; 15. } 16. public void setUserName(String userName) { 17. this.userName = userName; 18. } 19. public Integer getAge() { 20. return age; 21. } 22. public void setAge(Integer age) { 23. this.age = age; 24. } 25. @Override 26. public String toString() { 27. return "UserEntity [userName=" + userName + ", age=" + age + "]"; 28. } 29. // bean容器销毁 30. public void destroy() throws Exception { 31. System.*out*.println("destroy() 销毁bean"); 32. } 33. // 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 34. public void afterPropertiesSet() throws Exception { 35. System.*out*.println("afterPropertiesSet"); 36. } 37. // 如果bean实现了BeanNameAware接口,spring将bean的id传给setBeanName()方法 38. public void setBeanName(String name) { 39. System.*out*.println("setBeanName() set name:" + name); 40. } 41. // 如果bean实现了BeanNameAware接口,spring将bean的id传给setBeanName()方法 42. public void setBeanFactory(BeanFactory beanFactory) throws BeansException { 43. System.*out*.println("setBeanFactory()"); 44. } 45. // 获取Spring容器上下文 46. public void setApplicationContext(ApplicationContext applicationContext) throws BeansException { 47. System.*out*.println("获取上下文...."); 48. } 49. } 50. --------------- |
| System.***out***.println("开始初始化容器....");  ClassPathXmlApplicationContext app = **new** ClassPathXmlApplicationContext("spring.xml");  System.***out***.println("容器初始化成功....");  UserEntity userEntity = (UserEntity) app.getBean("userEntity01");  System.***out***.println("关闭容器...");  app.registerShutdownHook(); |

# SpringAop源码分析

常用单词

BeanFactory (兵fa ke te rui)

BeanDefinition(兵 fu 来xin)