# Spring基于纯注解方式的使用

## @Configuration

* **介绍：**

\* 从Spring3.0开始可以使用@Configuration定义**配置类**，可**替换xml配置文件**

**\* 相当于<beans>根标签**

\* **配置类**内部包含有一个或多个被**@Bean注解**的方法，这些方法将会被AnnotationConfigApplicationContext或AnnotationConfigWebApplicationContext类进行扫描，并用于构建bean定义对象，初始化Spring容器。

* **属性：**

value:用于指定配置类的字节码

* **示例代码：**

**@Configuration**

public class SpringConfiguration {

**//spring容器初始化时，会调用配置类的无参构造函数**

public SpringConfiguration(){

System.out.println(“容器启动初始化。。。”);

}

}

## @Bean

* **介绍：**

\* @Bean标注在方法上(返回某个实例的方法)，等价于spring配置文件中的<bean>

\* 作用为：注册bean对象

\* 主要用来配置非自定义的bean，比如DruidDataSource、SqlSessionFactory

* **属性：**

\* name：给当前@Bean 注解方法创建的对象指定一个名称(即 bean 的 id）。

\* 如果不指定，默认与标注的方法名相同

\* @Bean注解默认作用域为单例singleton作用域，可通过@Scope(“prototype”)设置为原型作用域；

* **示例代码：**

**@Configuration**

public class SpringConfiguration {

**//spring容器初始化时，会调用配置类的无参构造函数**

public SpringConfiguration(){

System.out.println(“容器启动初始化。。。”);

}

**@Bean**

**@Scope(“prototype”)**

public **UserService**userService(){

**return new UserServiceImpl(1,“张三”);**

}

}

## @ComponentScan

* **介绍：**

\* 相当于**context:component-scan标签**

\* 组件扫描器，扫描@Component、@Controller、@Service、@Repository注解的类。

\* 该注解是编写在类上面的，一般配合@Configuration注解一起使用。

* **属性：**

\* basePackages：用于指定要扫描的包。

\* value：和basePackages作用一样。

* **示例代码：**

**Bean类（Service类）：**

|  |
| --- |
| @Service  **publicclass** UserServiceImpl **implements** UserService {  @Override  **publicvoid** saveUser() {  System.***out***.println("保存用户 Service实现");  }  } |

**配置类：**

|  |
| --- |
| @Configuration  @ComponentScan(basePackages="com.kkb.spring.service")  **publicclass** SpringConfiguration {  **public** SpringConfiguration() {  System.***out***.println("容器初始化...");  }    // @Bean  // @Scope("prototype")  // public UserService userService() {  // return new UserServiceImpl(1,"张三");  // }  } |

## @PropertySource

* **介绍**

\* 加载properties配置文件

\* 编写在类上面

\* 相当于context:property-placeholder标签

* **属性**

value[]：用于指定properties文件路径，如果在类路径下，需要写上classpath

* **示例代码**

**配置类：**

|  |
| --- |
| @Configuration  @PropertySource("classpath:jdbc.properties")  **publicclass**JdbcConfig {  @Value("${jdbc.driver}")  **private** String driver;  @Value("${jdbc.url}")  **private** String url;  @Value("${jdbc.username}")  **private** String username;  @Value("${jdbc.password}")  **private** String password;  /\*\*  \* 创建一个数据源，并存入 spring 容器中  \*  \* **@return**  \*/  @Bean(name = "dataSource")  **public** DataSource createDataSource() {  **try** {  ComboPooledDataSourceds = **new**ComboPooledDataSource();  ds.setDriverClass(driver);  ds.setJdbcUrl(url);  ds.setUser(username);  ds.setPassword(password);  **return**ds;  } **catch** (Exception e) {  **thrownew** RuntimeException(e);  }  }  } |

properties文件：

|  |
| --- |
| jdbc.driver=com.mysql.jdbc.Driver  jdbc.url=jdbc:mysql:///spring  jdbc.username=root  jdbc.password=root |

## @Import

* **介绍**

\* 用来组合多个配置类

\* 相当于spring配置文件中的import标签

\* 在引入其他配置类时，可以不用再写@Configuration 注解。当然，写上也没问

题。

* **属性**

value：用来指定其他配置类的字节码文件

* **示例代码：**

|  |
| --- |
| @Configuration  @ComponentScan(basePackages = "com.kkb.spring")  @Import({ JdbcConfig.**class** })  **publicclass**SpringConfiguration {  }  @Configuration  @PropertySource("classpath:jdbc.properties")  **publicclass**JdbcConfig {  } |

## 创建纯注解方式上下文容器

* Java应用（AnnotationConfigApplicationContext）

|  |
| --- |
| ApplicationContext context = **new**AnnotationConfigApplicationContext(SpringConfiguration.**class**);  UserService service = context.getBean(UserService.**class**);  service.saveUser(); |

**测试**

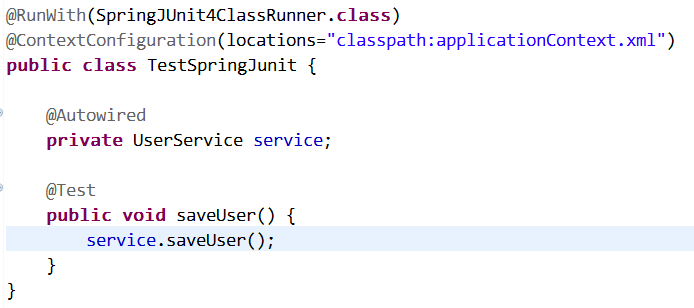
第一步：添加依赖

\* 添加spring-test包即可。

* 第二步：通过@RunWith注解，指定spring的运行器

\* Spring的运行器是SpringJunit4ClassRunner

* 第三步：通过@ContextConfiguration注解，指定spring运行器需要的配置文件路径
* 第四步：通过@Autowired注解给测试类中的变量注入数据



# Spring AOP原理分析

\* 通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术

* **横向抽取机制：**
* **Joinpoint(连接点)**

-- 所谓连接点是指那些被拦截到的点。在spring中,这些点指的是方法,因为spring只支持方法类型的连接点

* **Pointcut(切入点)**

-- 所谓切入点是指我们要对哪些Joinpoint进行拦截的定义

* **Advice(通知/增强)**

-- 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)

* **Introduction(引介)**

-- 引介是一种特殊的通知在不修改类代码的前提下, Introduction可以在运行期为类动态地添加一些方法或Field

* **Target(目标对象)**

-- 代理的目标对象

* **Weaving(织入)**

-- 是指把增强应用到目标对象来创建新的代理对象的过程

* **Proxy（代理）**

-- 一个类被AOP织入增强后，就产生一个结果代理类

* **Aspect(切面)**

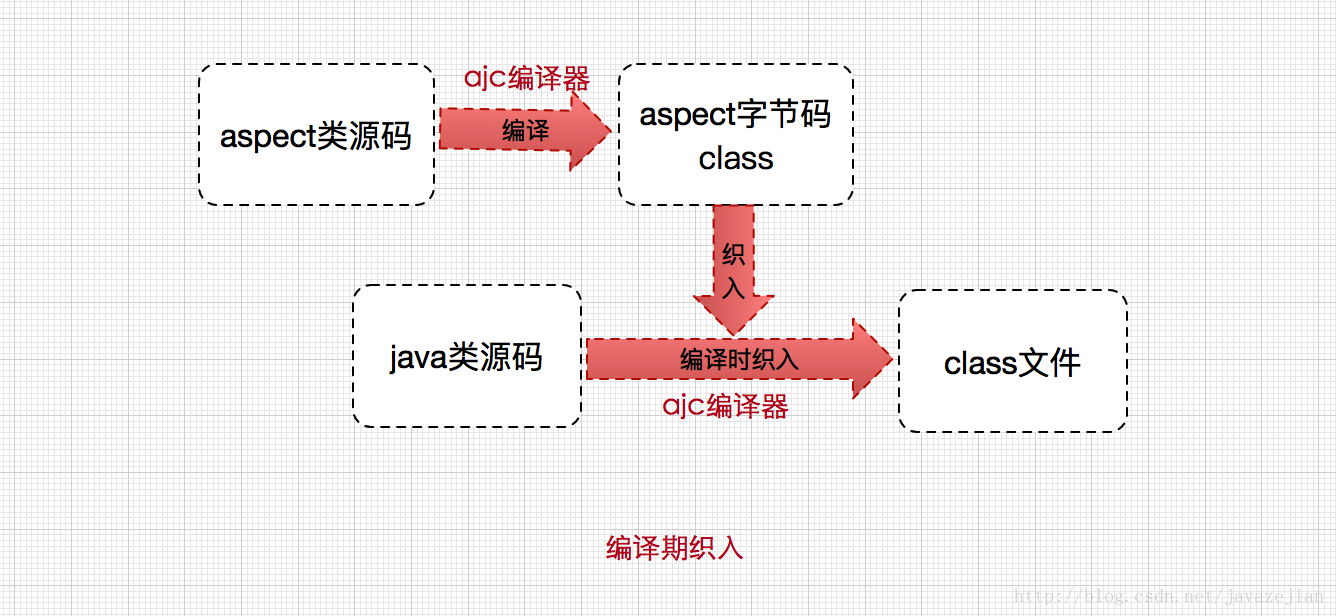
**-- 是切入点和通知的结合，以后咱们自己来编写和配置的**

* **Advisor（通知器、顾问）**

**--和Aspect很相似**

织入过程，一般分为**动态织入和静态织入**，**动态织入的方式是在运行时动态将要增强的代码织入到目标类中**，这样往往是**通过动态代理技术完成**的，如Java JDK的动态代理(Proxy，底层通过反射实现)或者CGLIB的动态代理(底层通过继承实现)，**Spring AOP采用的就是基于运行时增强的代理技术**

**ApectJ采用的就是静态织入的方式**。**ApectJ主要采用的是编译期织入**，在这个期间使用AspectJ的acj编译器(类似javac)把aspect类编译成class字节码后，在java目标类编译时织入，即先编译aspect类再编译目标类。

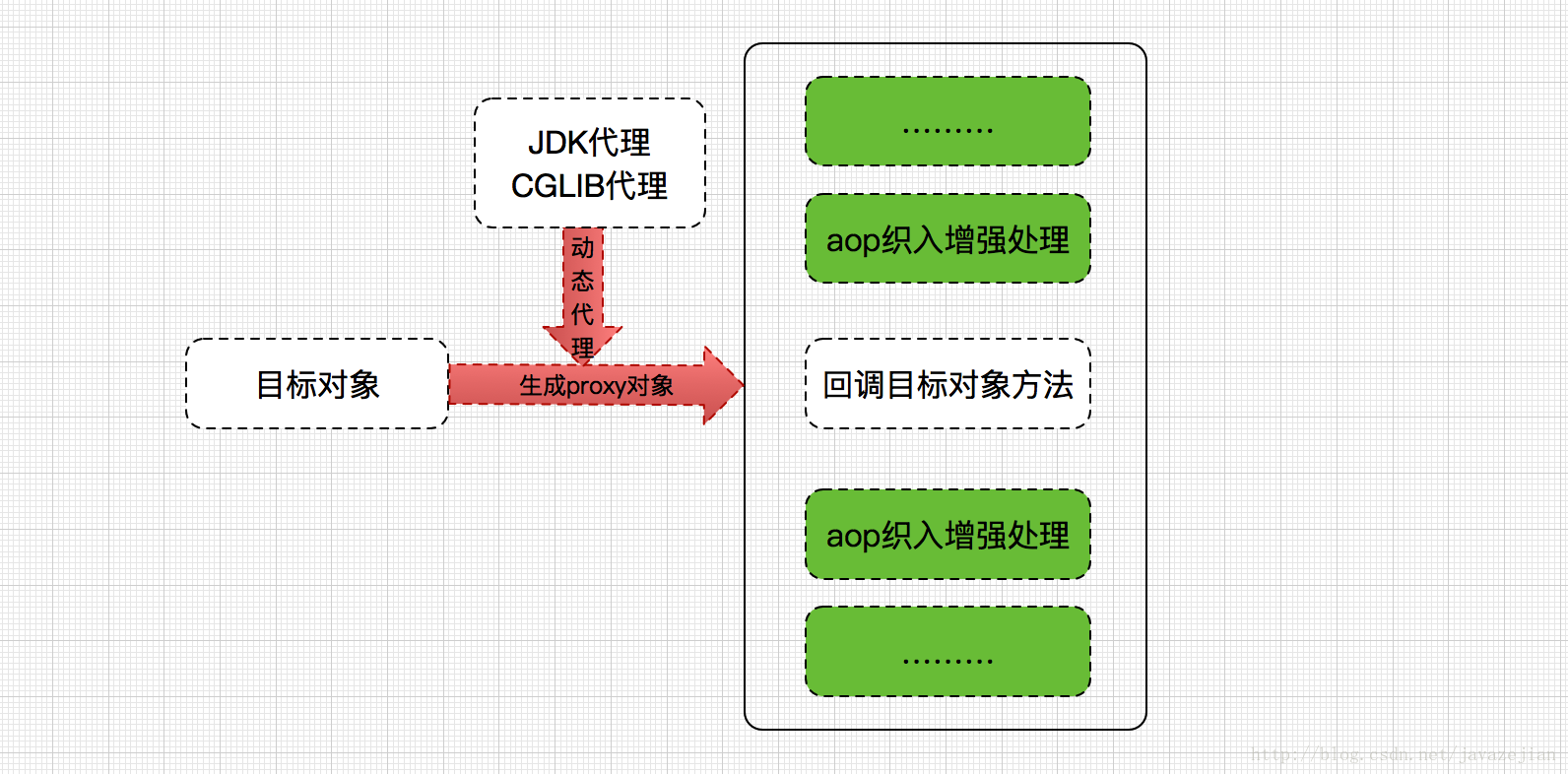


SpringAOP

\* Spring AOP是通过**动态代理技术**实现的

\* 而动态代理是基于**反射**设计的。（关于反射的知识，请自行学习）

\* 动态代理技术的实现方式有两种：基于接口的**JDK动态代理**和基于继承的**CGLib动态代理**。



* **JDK动态代理**

\* 目标对象必须实现接口

|  |
| --- |
| 1. 使用Proxy类来生成代理对象的一些代码如下：  /\*\*  \* 使用JDK的方式生成代理对象  \* @author Administrator  \*/  public class MyProxyUtils {  public static UserService getProxy(final UserServiceservice) {  // 使用Proxy类生成代理对象  UserService proxy =  (UserService) Proxy.newProxyInstance(  **service.getClass().getClassLoader(),**  **service.getClass().getInterfaces(),**  **new InvocationHandler()** {  // 代理对象方法一执行，invoke方法就会执行一次  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  if("save".equals(method.getName())){  System.out.println("记录日志...");  // 开启事务  }  // 提交事务  // 让service类的save或者update方法正常的执行下去  return **method.invoke(service, args);**  }  });  // 返回代理对象  return proxy;  }  } |

* **CGLib动态代理**

\* 目标对象不需要实现接口

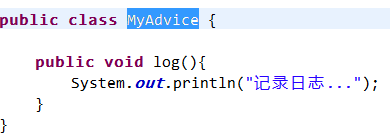
\* 底层是通过继承目标对象产生代理子对象（代理子对象中继承了目标对象的方法，并可以对该方法进行增强）

|  |
| --- |
| 2. 编写相关的代码  public static UserService getProxy(){  // 创建CGLIB核心的类  **Enhancer**enhancer = new Enhancer();  // 设置父类  enhancer.setSuperclass(**UserServiceImpl**.class);  // 设置回调函数  enhancer.setCallback(new **MethodInterceptor**() {  @Override  public Object intercept(Object obj, Method method, Object[] args,  MethodProxy methodProxy) throws Throwable {  if("save".equals(method.getName())){  // 记录日志  System.out.println("记录日志了...");  }  return methodProxy.invokeSuper(obj, args);  }  });  // 生成代理对象  UserService proxy = (UserService) enhancer.create();  return proxy;  } |

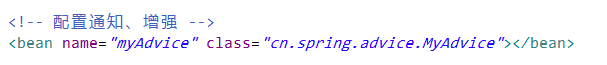
# AspectJ

Spring + AspectJ整合

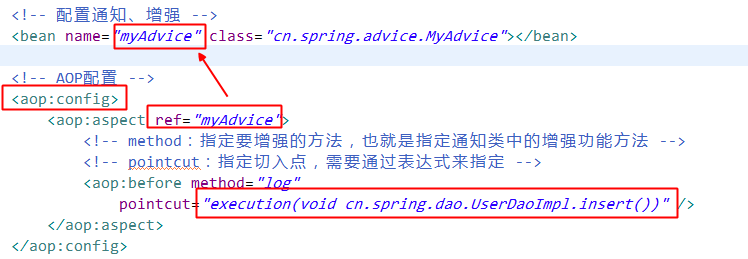
**写通知（增强类，一个普通的类）**



* **配置通知，将通知类交给spring IoC容器管理**



* **配置AOP 切面**



### 切入点表达式



### 通知类型

\* 通知类型（五种）：前置通知、后置通知、最终通知、环绕通知、异常抛出通知。

\* 前置通知：

\* 执行时机：目标对象方法之前执行通知

\* 配置文件：<aop:before method="before" pointcut-ref="myPointcut"/>

\* 应用场景：方法开始时可以进行校验

\* 后置通知：

\* 执行时机：目标对象方法之后执行通知，**有异常则不执行了**

\* 配置文件：<aop:after-returning method="afterReturning" pointcut-ref="myPointcut"/>

\* 应用场景：可以修改方法的返回值

\* 最终通知：

\* 执行时机：目标对象方法之后执行通知，**有没有异常都会执行**

\* 配置文件：<aop:after method="after" pointcut-ref="myPointcut"/>

\* 应用场景：例如像释放资源

\* 环绕通知：

\* 执行时机：目标对象方法之前和之后都会执行。

\* 配置文件：<aop:around method="around" pointcut-ref="myPointcut"/>

\* 应用场景：事务、统计代码执行时机

\* 异常抛出通知：

\* 执行时机：在抛出异常后通知

\* 配置文件：<aop:after-throwing method=" afterThrowing " pointcut- ref="myPointcut"/>

\* 应用场景：包装异常

## 使用注解实现

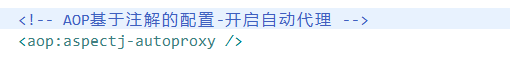
编写**切面类**



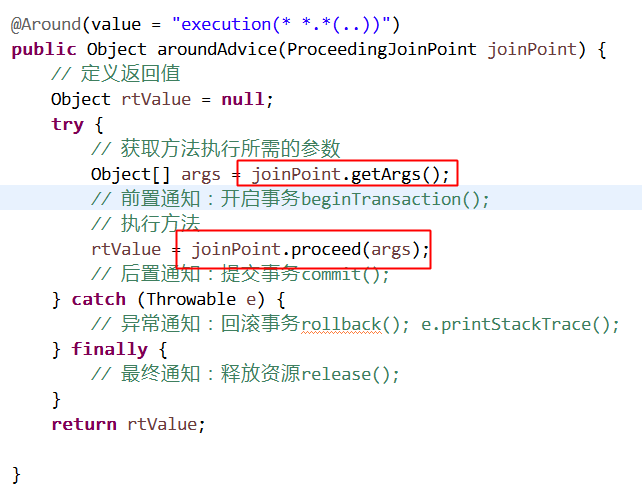
* **配置切面类**

<context:component-scan base-package=*"com.kkb.spring"/*>

* 开启AOP自动代理



### 环绕通知注解配置



### 定义通用切入点

\* 使用@PointCut注解在切面类中定义一个通用的切入点，其他通知可以引用该切入点



## 纯注解的Spring AOP配置方式

@Configuration

@ComponentScan(basePackages="com.kkb")

**@EnableAspectJAutoProxy**

public class SpringConfiguration {

}

# 连接池

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql:///spring"/>

<property name="username" value="root"/>

<property name="password" value="root"/>

</bean>

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">

<property name="driverClass" value="com.mysql.jdbc.Driver"/>

<property name="jdbcUrl" value="jdbc:mysql:///spring"/>

<property name="user" value="root"/>

<property name="password" value="root"/>

</bean>

# Spring应用之事务支持

## Spring框架的事务管理相关的类和API

|  |
| --- |
| **Spring并不直接管理事务，而是提供了多种事务管理器，他们将事务管理的职责委托给Hibernate或者JTA等持久化机制所提供的相关平台框架的事务来实现。 Spring事务管理器的接口是PlatformTransactionManager，通过这个接口，Spring为各个平台如JDBC、Hibernate等都提供了对应的事务管理器，但是具体的实现就是各个平台自己的事情了。** |

**1. PlatformTransactionManager接口** -- 平台事务管理器.(真正管理事务的类)。该接口有具体的实现类，根据不同的持久层框架，需要选择不同的实现类！

2. TransactionDefinition接口 -- 事务定义信息.(事务的隔离级别,传播行为,超时,只读)

3. TransactionStatus接口 -- 事务的状态（是否新事务、是否已提交、是否有保存点、是否回滚）

4. 总结：上述对象之间的关系：平台事务管理器真正管理事务对象.根据事务定义的信息TransactionDefinition 进行事务管理，在管理事务中产生一些状态.将状态记录到TransactionStatus中

5. PlatformTransactionManager接口中实现类和常用的方法

1. 接口的实现类

\* 如果使用的Spring的JDBC模板或者MyBatis（IBatis）框架，需要选择**DataSourceTransactionManager**实现类

\* 如果使用的是Hibernate的框架，需要选择**HibernateTransactionManager**实现类

2. 该接口的常用方法

\* void commit(TransactionStatus status)

\* TransactionStatus getTransaction(TransactionDefinition definition)

\* void rollback(TransactionStatus status)

6. TransactionDefinition

1. 事务隔离级别的常量

\* static int ISOLATION\_DEFAULT -- 采用数据库的默认隔离级别

\* static int ISOLATION\_READ\_UNCOMMITTED

\* static int ISOLATION\_READ\_COMMITTED

\* static int ISOLATION\_REPEATABLE\_READ

\* static int ISOLATION\_SERIALIZABLE

2. 事务的传播行为常量（不用设置，使用默认值）

事务的传播行为：解决的是业务层之间的方法调用

\* **PROPAGATION\_REQUIRED**（默认值） -- A中有事务,使用A中的事务.如果没有，B就会开启一个新的事务,将A包含进来.(保证A,B在同一个事务中)，默认值！！

\* PROPAGATION\_SUPPORTS -- A中有事务,使用A中的事务.如果A中没有事务.那么B也不使用事务.

\* PROPAGATION\_MANDATORY -- A中有事务,使用A中的事务.如果A没有事务.抛出异常.

\* PROPAGATION\_REQUIRES\_NEW -- A中有事务,将A中的事务挂起.B创建一个新的事务.(保证A,B没有在一个事务中)

\* PROPAGATION\_NOT\_SUPPORTED -- A中有事务,将A中的事务挂起.

\* PROPAGATION\_NEVER -- A中有事务,抛出异常.

\* PROPAGATION\_NESTED -- 嵌套事务.当A执行之后,就会在这个位置设置一个保存点.如果B没有问题.执行通过.如果B出现异常,运行客户根据需求回滚(选择回滚到保存点或者是最初始状态)

## spring框架事务管理的分类

1. Spring的编程式事务管理（不推荐使用）

\* 通过手动编写代码的方式完成事务的管理（不推荐）

2. Spring的**声明式事务管理**（**底层采用AOP的技术**）

\* 通过一段配置的方式完成事务的管理

### 编程式事务管理（了解）

1. 说明：Spring为了简化事务管理的代码:提供了模板类 TransactionTemplate，所以手动编程的方式来管理事务，只需要使用该模板类即可！！

2. 手动编程方式的具体步骤如下：

1. 步骤一:配置一个事务管理器，Spring使用PlatformTransactionManager接口来管理事务，所以咱们需要使用到他的实现类！！

<!-- 配置事务管理器 -->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"/>

</bean>

2. 步骤二:配置事务管理的模板

<!-- 配置事务管理的模板 -->

<bean id="transactionTemplate" class="org.springframework.transaction.support.TransactionTemplate">

<property name="transactionManager" ref="transactionManager"/>

</bean>

3. 步骤三:在需要进行事务管理的类中,注入事务管理的模板

<bean id="accountService" class="com.itheima.demo1.AccountServiceImpl">

<property name="accountDao" ref="accountDao"/>

<property name="transactionTemplate" ref="transactionTemplate"/>

</bean>

4. 步骤四:在业务层使用模板管理事务:

// 注入事务模板对象

private TransactionTemplate transactionTemplate;

public void setTransactionTemplate(TransactionTemplate transactionTemplate) {

this.transactionTemplate = transactionTemplate;

}

public void pay(final String out, final String in, final double money) {

transactionTemplate.execute(new TransactionCallbackWithoutResult() {

protected void doInTransactionWithoutResult(TransactionStatus status) {

// 扣钱

accountDao.outMoney(out, money);

int a = 10/0;

// 加钱

accountDao.inMoney(in, money);

}

});

}

### 声明式事务管理（重点）

声明式事务管理又分成两种方式

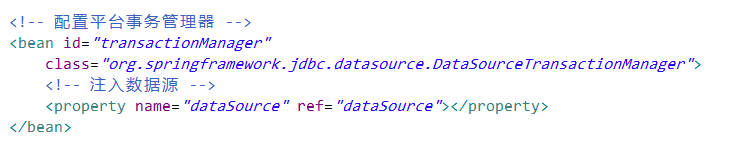
\* 基于AspectJ的XML方式（重点掌握）

\* 基于AspectJ的注解方式（重点掌握）

## 事务管理之基于AspectJ的XML方式（重点掌握）

\* 配置事务管理的AOP

\* 平台事务管理器：DataSourceTransactionManager



\* 事务通知：<tx:advice id=”” transaction-manager=””/>

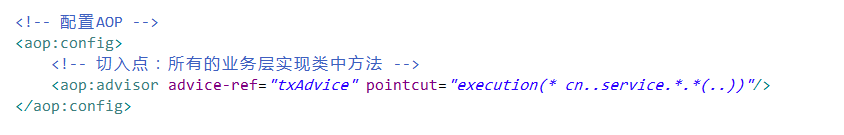


\* AOP配置：

<aop:config>

<aop:advisor advice-ref=”” pointcut=””/>

</aop:config>



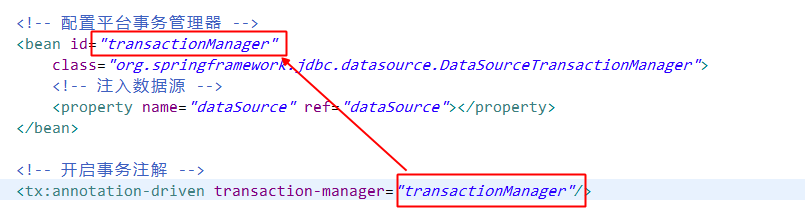
## 事务管理之基于AspectJ的注解方式（重点掌握）

\* service类上或者方法上加注解：

\* 类上加@Transactional：表示该类中所有的方法都被事务管理

\* 方法上加@Transactional：表示只有该方法被事务管理

\* 开启事务注解：



* ApplicationContext和BeanFactory的区别？

\* 创建Bean对象的时机不同:

\* BeanFactory采取**延迟加载**，第一次getBean时才会初始化Bean。

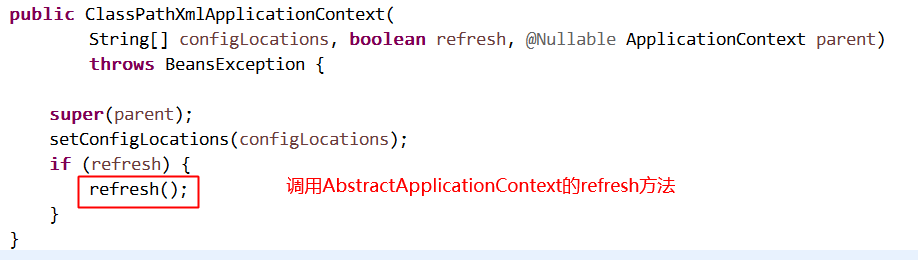
\* ApplicationContext是加载完applicationContext.xml时，就创建具体的Bean对象的实例。（**只对BeanDefition中描述为是单例的bean，才进行饿汉式加载**）

1. web服务器（tomcat）启动会加载web.xml（启动**ContextLoaderListener**监听器）:
2. 调用ContextLoaderListener类的contextInitializd方法，创建Web环境中的Spring上下文对象
3. 调用ContextLoader类的initWebApplicationContext方法
4. 继续调用ContextLoader类的configureAndRefreshWebApplicationContext方法，该方法中调用最终初始化Bean的**refresh**方法

* **主流程入口：**

ApplicationContext context = new ClassPathXmlApplicationContext（“spring.xml”）

* **ClassPathXmlApplicationContext类：重载的构造方法依次调用，进入下面代码**



* **AbstractApplicationContext的refresh方法：初始化spring容器的核心代码**

|  |
| --- |
| @Override  **publicvoid**refresh() **throws** BeansException, IllegalStateException {  **synchronized** (**this**.startupShutdownMonitor) {  //**1、** Prepare this context for refreshing.  prepareRefresh();  **//创建DefaultListableBeanFactory（真正生产和管理bean的容器）**  **//BeanDefinition处理**  **// 1 定位XML文件(Resource、ResourceLoader)**  **//2 加载XML（Document、Dom4j解析）**  **// 3 从Document对象中解析BeanDefinition然后注册到BeanDefinitionRegistry（Map）**  **//**  **//通过NamespaceHandler解析自定义标签的功能（比如:context标签、aop标签、tx标签）**  //**2、** Tell the subclass to refresh the internal bean factory.  ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();  //**3、** Prepare the bean factory for use in this context.  prepareBeanFactory(beanFactory);  **try** {  //**4、** Allows post-processing of the bean factory in context subclasses.  postProcessBeanFactory(beanFactory);  **//实例化并调用实现了BeanFactoryPostProcessor接口的Bean**  **//比如：PropertyPlaceHolderConfigurer（context:property-placeholer）**  **//就是此处被调用的，作用是替换掉BeanDefinition中的占位符（${}）中的内容**  //**5、** Invoke factory processors registered as beans in the context.  invokeBeanFactoryPostProcessors(beanFactory);  **//创建并注册BeanPostProcessor到BeanFactory中（Bean的后置处理器）**  **//比如：AutowiredAnnotationBeanPostProcessor（实现@Autowired注解功能）**  **// RequiredAnnotationBeanPostProcessor（实现@Required注解功能）**  **//这些注册的BeanPostProcessor**  //**6、** Register bean processors that intercept bean creation.  registerBeanPostProcessors(beanFactory);  //**7、** Initialize message source for this context.  initMessageSource();  //**8、** Initialize event multicaster for this context.  initApplicationEventMulticaster();  //**9、** Initialize other special beans in specific context subclasses.  onRefresh();  //**10、** Check for listener beans and register them.  registerListeners();  **//1 通过构造方法，创建Bean的实例。此处的Bean是非懒加载方式的单例Bean（未设置属性）**  **//2 填充属性（DI，依赖注入）---循环依赖（A中有B的依赖、B中有A的依赖）**  **//3 初始化实例（比如调用init-method方法）**  **//调用BeanPostProcessor（后置处理器）对实例bean进行后置处理（AOP功能）**  //**11、** Instantiate all remaining (non-lazy-init) singletons.  finishBeanFactoryInitialization(beanFactory);  //**12、** Last step: publish corresponding event.  finishRefresh();  }  **catch** (BeansException ex) {  **if** (logger.isWarnEnabled()) {  logger.warn("Exception encountered during context initialization - " +  "cancelling refresh attempt: " + ex);  }  // Destroy already created singletons to avoid dangling resources.  destroyBeans();  // Reset 'active' flag.  cancelRefresh(ex);  // Propagate exception to caller.  **throw**ex;  }  **finally** {  // Reset common introspection caches in Spring's core, since we  // might not ever need metadata for singleton beans anymore...  resetCommonCaches();  }  }  } |

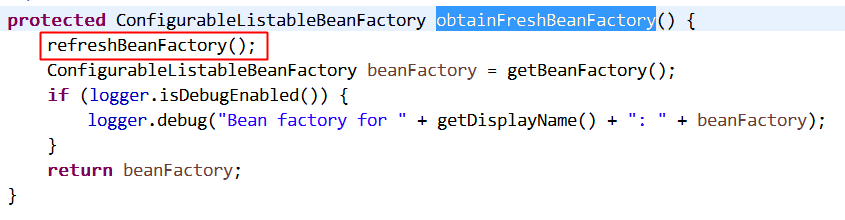
### 创建BeanFactory流程分析

#### 获取新的BeanFactory子流程

* 子流程入口（从主流程refresh方法中的第二步开始）



* 调用AbstractApplicationContext中的obtainFreshBeanFactory方法



* 调用AbstractRefreshableApplicationContext的refreshBeanFactory方法



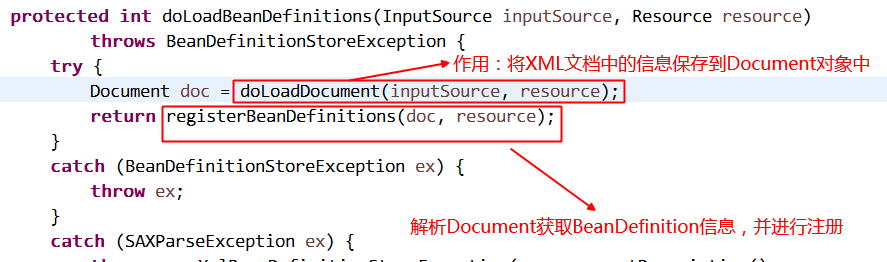
#### 加载解析BeanDefinition子流程（loadDefinitions方法）

##### 源码分析

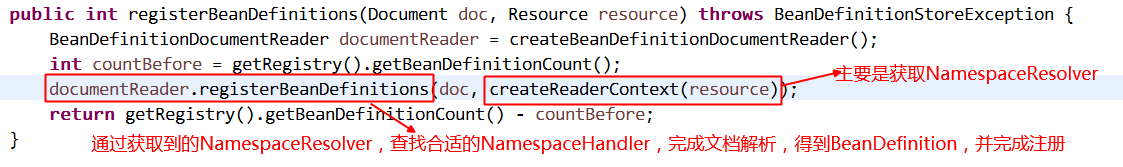
* 子流程入口（AbstractRefreshableApplicationContext类的方法）



* 此处依次调用多个类的loadBeanDefinitions方法（AbstractXmlApplicationContext🡪 AbstractBeanDefinitionReader🡪 XmlBeanDefinitionReader），一直调用到XmlBeanDefinitionReader 类的doLoadBeanDefinitions方法



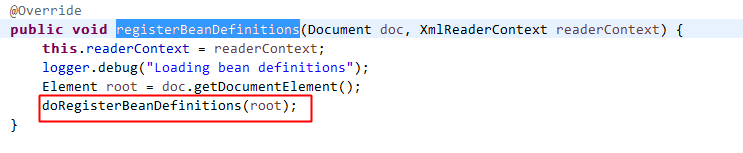
* 对于doLoadDocument方法不是我们关注的重点，我们进入到该类的registerBeanDefinitions方法看看



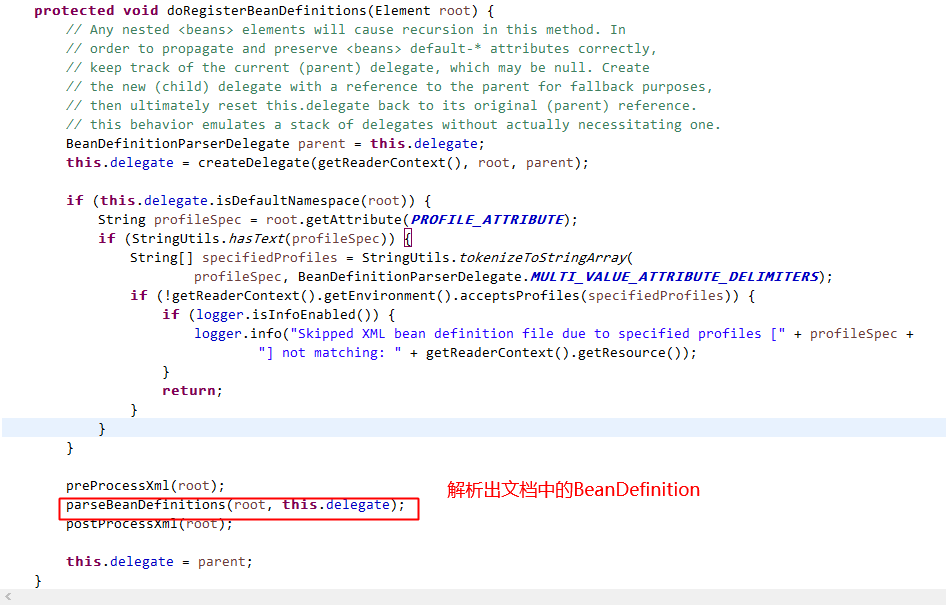
* 此处有两个地方是我们关注的：一个createRederContext方法，一个是DefaultBeanDefinitionDocumentReader类的registerBeanDefinitions方法，先进入createRederContext方法看看



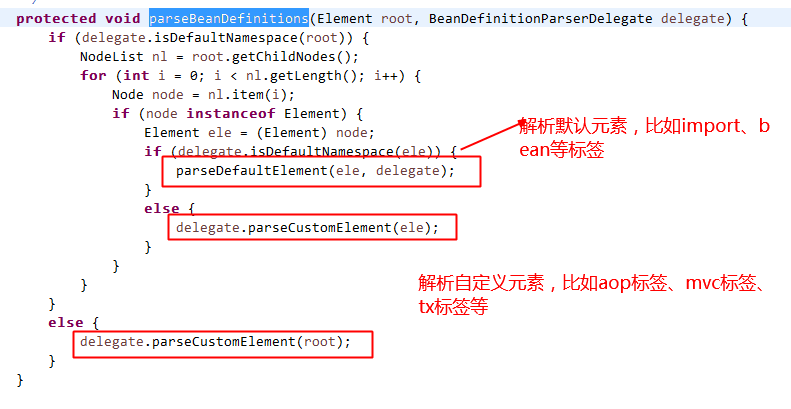
* 至此，14个NamespaceHandlerResolver初始化成功。然后我们再进入DefaultBeanDefinitionDocumentReader类的registerBeanDefinitions方法



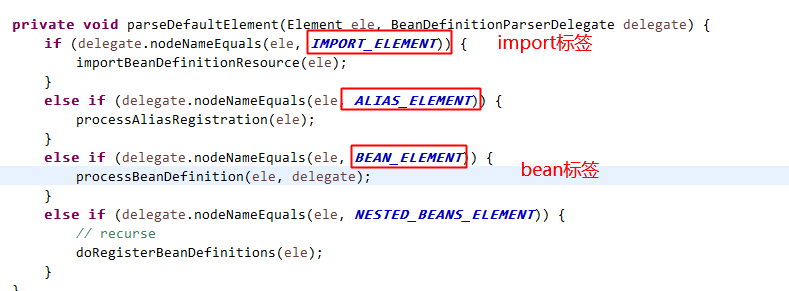
* 继续进入到该类的doRegisterBeanDefinitions方法看看，这是真正干活的方法



* 继续进入parseBeanDefinitions方法



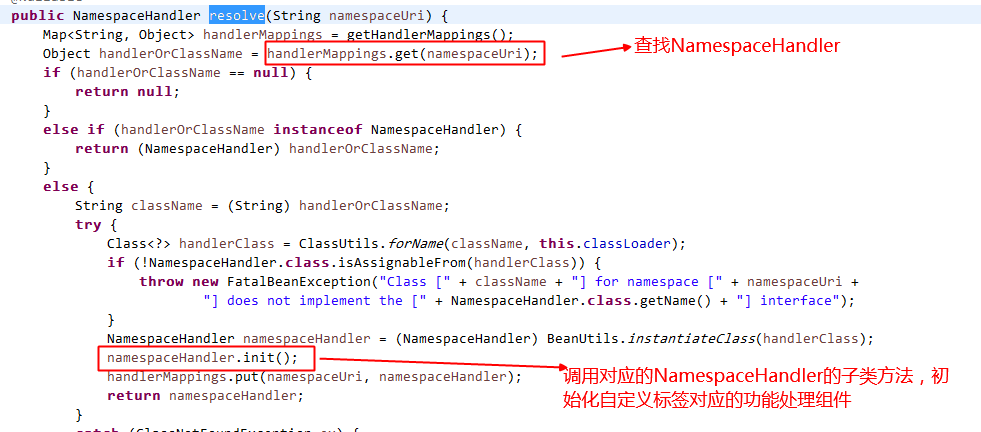
* 我们看到有两种解析方案，先看看parseDefaultElement方法



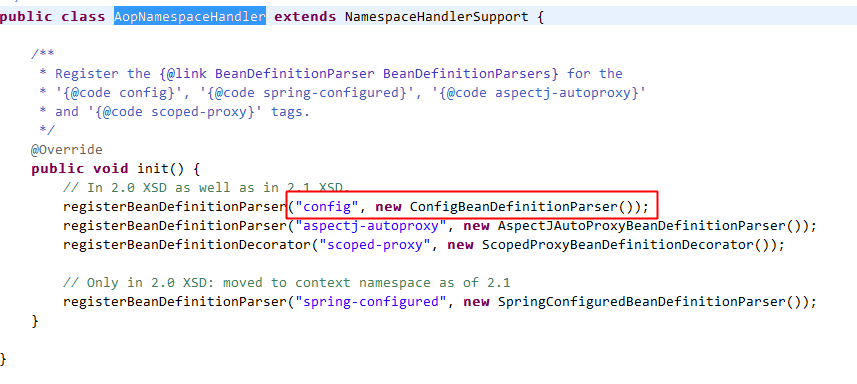
* 不过我们重点看看BeanDefinitionParserDelegate类的parseCustomElement方法（AOP标签、tx标签的解析都是在该步骤中完成的）



* getNamespaceURI方法的作用一目了然，我们就不去追踪了，接下来我们进入DefaultNamespaceHandlerResolver类的resolve方法看看：



* 在上面代码中，我们看到了一行代码：namespaceHandler.init();这个方法是很重要的。它实现了自定义标签到处理类的注册工作，不过NamespaceHandler是一个接口，具体的init方法需要不同的实现类进行实现，我们通过**AopNamespaceHandler**了解一下init的作用，其中**aop:config标签是由ConfigBeanDefinitionParser类进行处理**：



* 至此，我们了解到了xml中的aop标签都是由哪些类进行处理的了。不过init方法只是注册了标签和处理类的对应关系，那么什么时候调用处理类进行解析的呢？我们再回到BeanDefinitionParserDelegate类的parseCustomElement方法看看



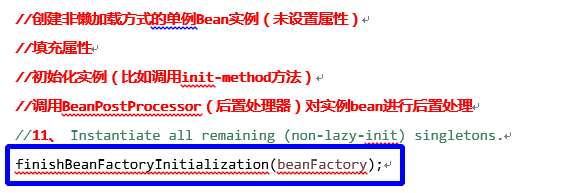
* 我们看到，最后一行执行了parse方法，那么parse方法，在哪呢？我们需要到NamespaceHandlerSupport类中去看看，它是实现NamespaceHandler接口的，并且AopNamespaceHandler是继承了NamespaceHandlerSupport类，那么该方法也会继承到AopNamespaceHandler类中。



* 至此，整个XML文档的解析工作，包括bean标签以及自定义标签如何解析为BeanDefinition信息的过程，我们已经了解了。
* **后续具体想了解哪个自定义标签的处理逻辑，可以自行去查找xxxNamespaceHandler类进行分析。**

### 创建Bean流程分析

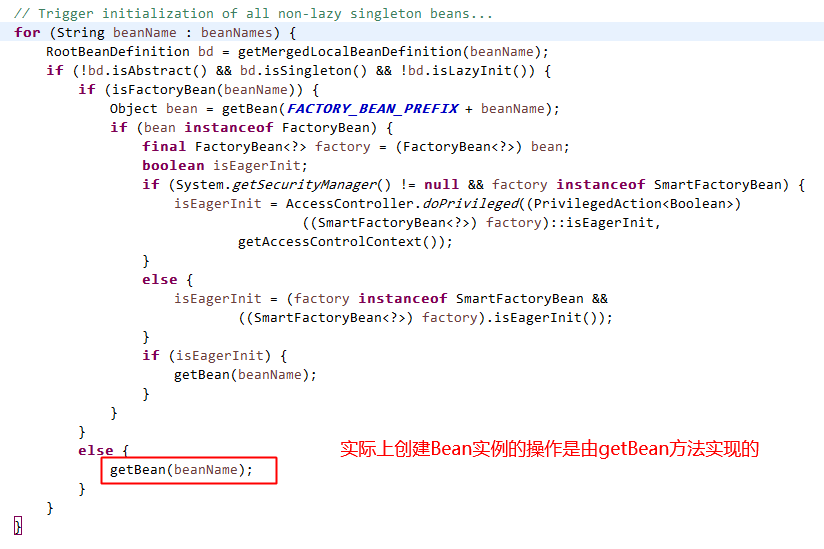
* 子流程入口



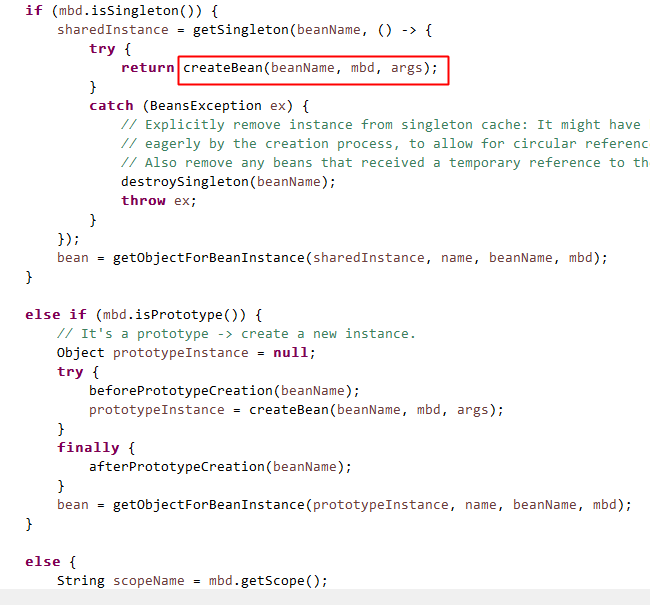
* 我们进入finishBeanFactoryInitialization方法看看：



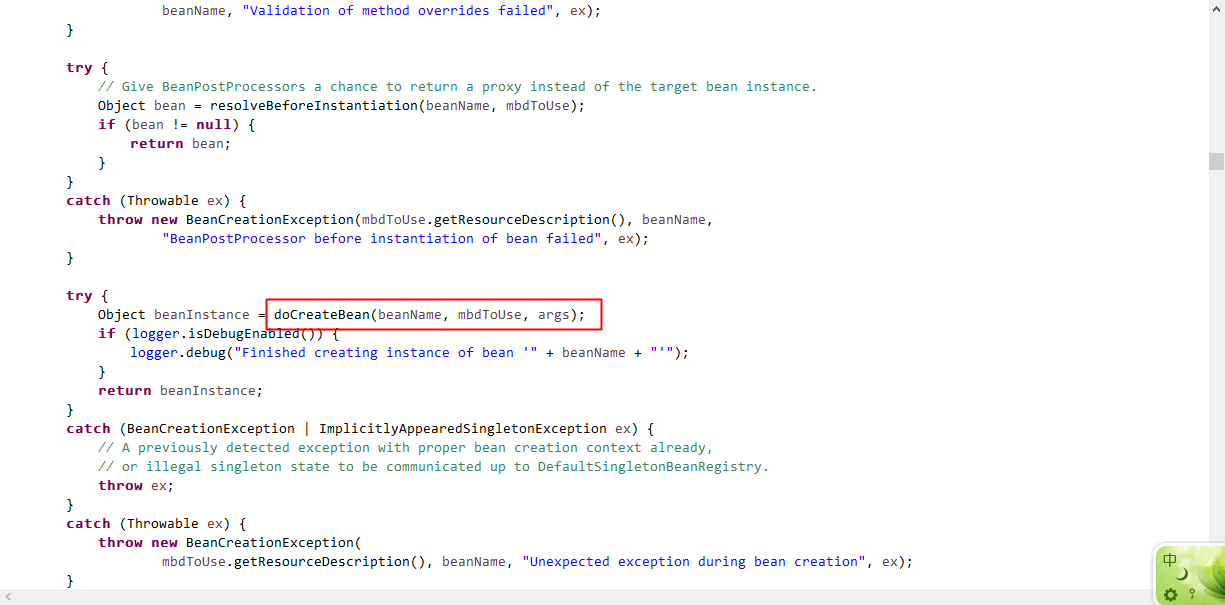
* 继续进入DefaultListableBeanFactory类的preInstantiateSingletons方法，我们找到下面部分的代码，看到工厂Bean或者普通Bean，最终都是通过getBean的方法获取实例的。



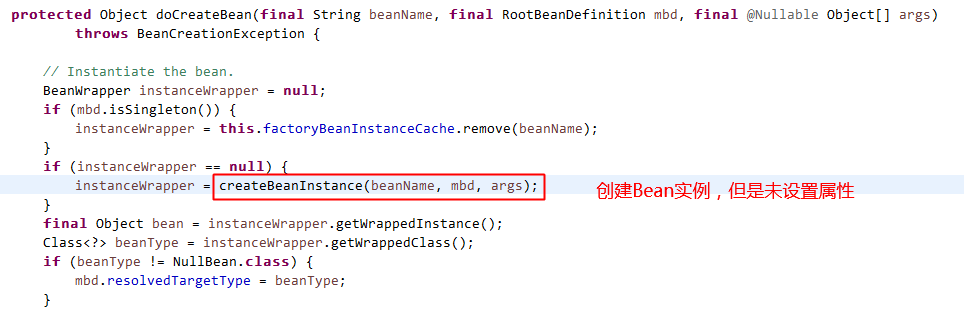
* 继续跟踪下去，我们进入到了AbstractBeanFactory类的doGetBean方法，这个方法中的代码很多，我们直接找到核心部分：



* 接着进入到AbstractAutowireCapableBeanFactory类的方法，找到以下代码部分



* 我们终于找到核心的地方了，进入doCreateBean方法看看，该方法我们关注两块重点区域：





* 对于如何创建Bean的实例，和填充属性，暂时先不去追踪了，我们先去看看initializeBean方法是如何调用BeanPostProcessor的，因为这个牵扯到我们对于AOP动态代理的理解。



* 至此，如何创建Bean我们已经明白了。

