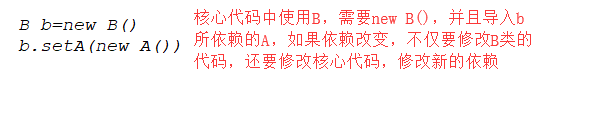
Spring IOC：

在传统的项目中，如果在核心代码中如果需要调用一个对象，那么得在核心代码中new Object()，同时还要引入对象的依赖，如果对象的依赖改变，核心代码也会修改，耦合



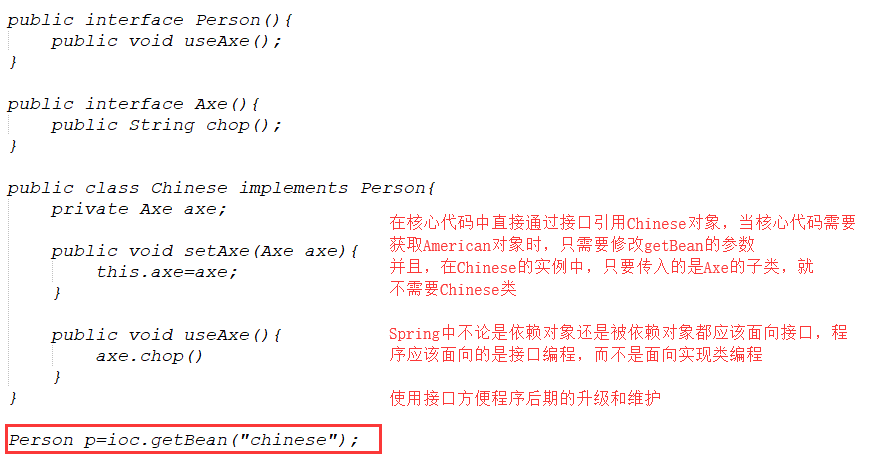
简单工厂：工厂创建对象，将对象之间返回，用户需要知道创建对象的过程，只要告诉指定的工厂，让其返回指定的对象，但是如果对象改变，不仅要在核心代码中修改引用的工厂，还要修改工厂方法

IOC：核心代码只需声明对象的引用，Spring容器会自动创建，并将对象b的所依赖的对象a也引入到对象b中，直接返回给核心代码一个创建好的对象b，核心代码只需引用即可

B b=Ioc.getBean(“b”)

面向接口编程：

Spring推荐面向接口编程，在核心代码中，引用对象时，直接可以声明对象的父类作为引用即可(程序应该面向他们的接口，而不是面向实现类编程)



Spring是通过配置文件，利用反射来创建实例的

BeanFactory和ApplicationContext：

ApplicationContext实现了BeanFactory接口，ApplicationContext包含了BeanFactory的所有功能，所以一般使用ApplicationContext来作为IOC容器，ApplicationContext在加载完xml配置文件后就会将Singleton的Bean实例化并加载到IOC容器中，BeanFactory只会加载xml，不会对bean预加载，只有使用的时候才会去初始化

ApplicationContext还实现了MessageSource(国际化)接口

ApplicationContext ioc = new ClassPathXmlApplicationContext("application.xml");

ClassPathXmlApplicationContext：加载类路径下的资源(ClassPath)

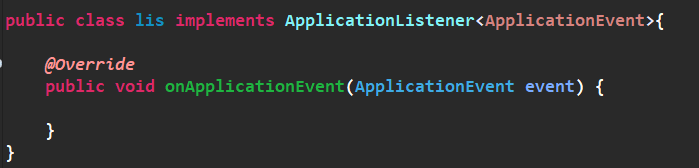
FileSystemXmlApplicationContext：加载系统路径下的资源

XmlWebApplicationContext：使用ServletContextResource进行资源访问

ApplicationContext的事件机制：

ApplicationListener：监听器，ApplicationListener会监听IOC容器的创建，当IOC容器创建时，会执行onApplicationEvent方法

ApplicationEvent：容器事件





Ioc.publishEvent(new ApplicationEvent)

当向IOC容器中发布容器事件(ApplicationEvent的实现类)时，会调用onApplicationEvent方法，并将ApplicationEvent实现类的对象作为参数传递给onApplicationEvent

Spring的内置事件：

ContextRefreshedEvent：当ApplicationContext初始化时候调用

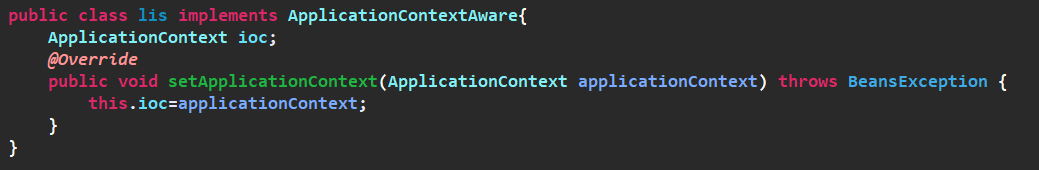
ContextStartEvent：当试用ConfigurableApplicationContext(ApplicationContext的子类)的start()启动IOC容器时触发改事件

ContextCloseEvent：当试用ConfigurableApplicationContext(ApplicationContext的子类)的close()启动IOC容器时触发改事件

ContextStoppedEvent：当试用ConfigurableApplicationContext(ApplicationContext的子类)的stop()启动IOC容器时触发改事件

RequestHandleEvent：只适用于DispatcherServlet的web应用，当Spring处理用户请求结束后，系统会自动触发该事件

Bean获取IOC：



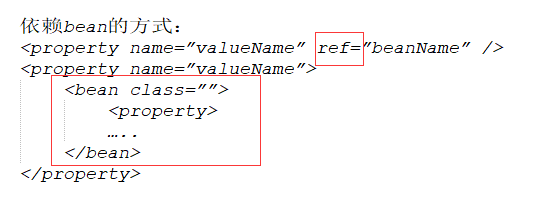
当bean实现BeanFactory/ApplicationContextAware接口后，重写set方法，就可以在当前bean中使用IOC容器了

Spring依赖的方式：

设值注入：通过setter方法注入对象的依赖(其他对象，属性)

构造注入：通过构造函数注入对象的依赖(其他对象，属性)

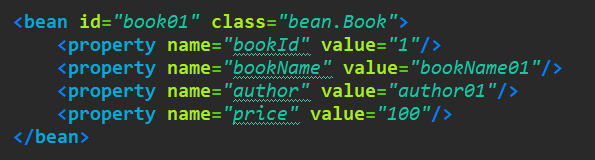
区别：如果对象的依赖不发生改变，则可以使用构造注入，但是一般建议采用设值注入



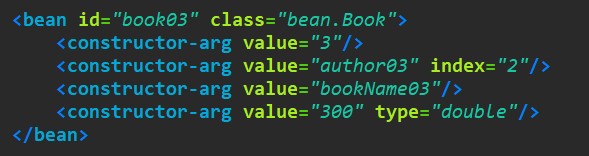
Spring构造时的依赖：



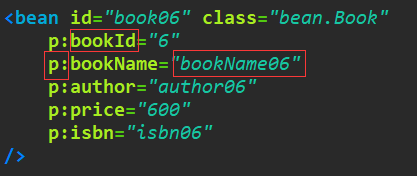
Bean



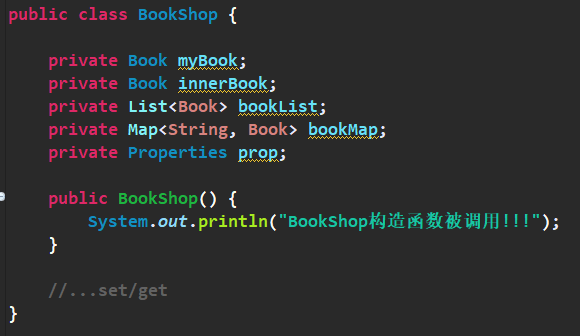
方式一



通过构造注入，会根据传入参数的个数自动匹配到构造器，可以通过index指定参数的位子，type指定参数的类型，并且Spring支持自动类型装换，在constructor-arg中传入的是字符串3，在构造函数中会被转化为int类型(constructor构造器)

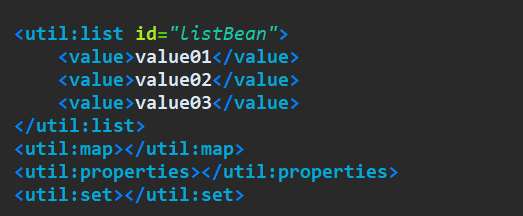


通过p标签简化传参：先调用无参的构造函数，再调用set方法，当属性命名时以-ref结尾的，p标签发生冲突

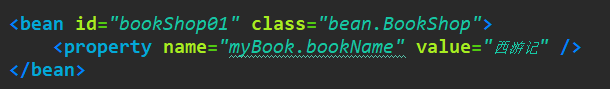


Bean



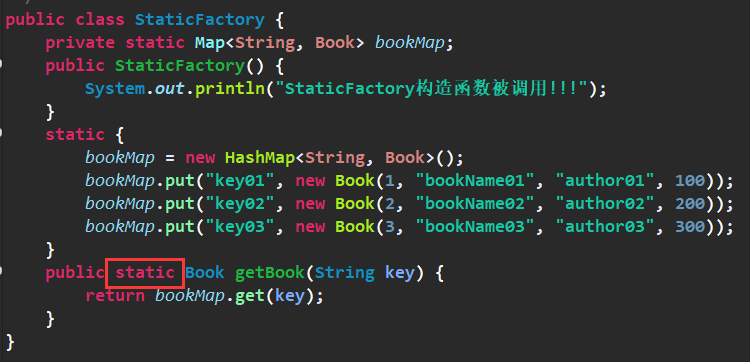


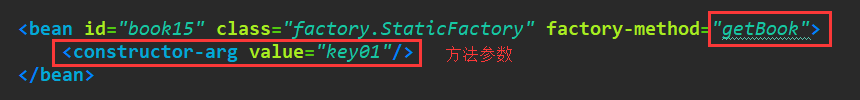
集合属性 通过util标签创建集合更加方便，快捷



联级属性

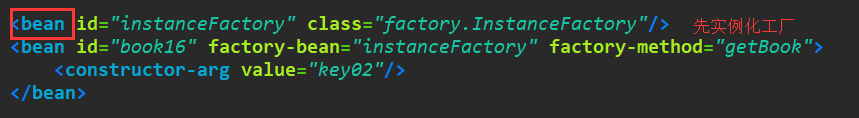
通过工厂创建bean



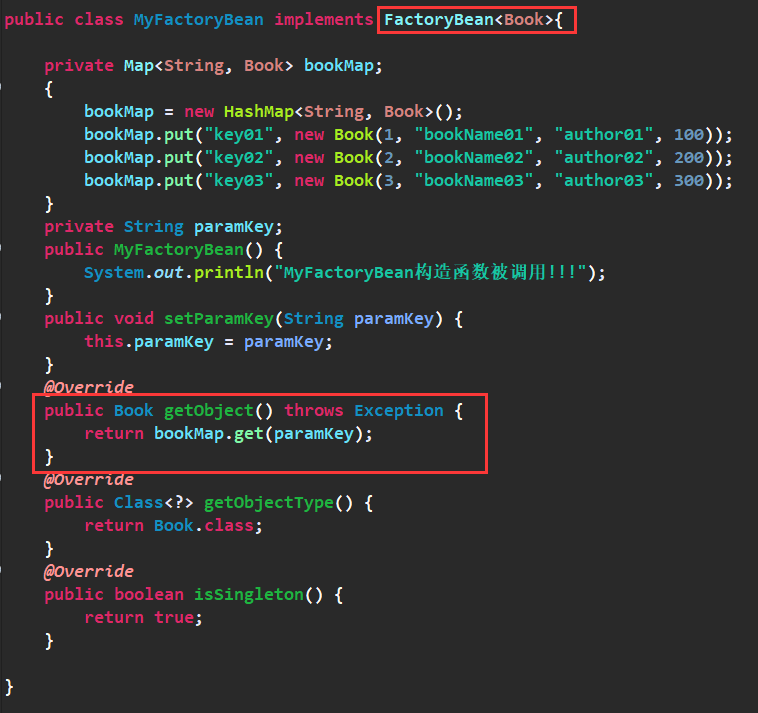


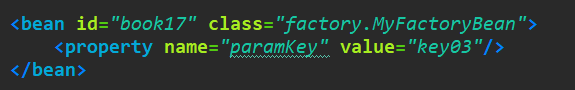
静态工厂方法获取Bean





实例工厂方法创建Bean，先创建工厂实例，在获取对象

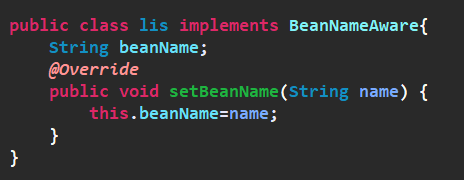




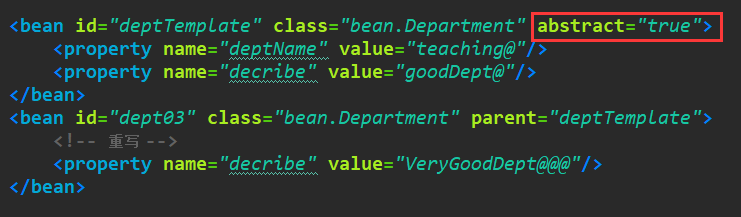
FactoryBean：通过实现FactoryBean接口重写getObject方法获取Bean：getBean(“book17”)

如果要获取FactoryBean的实例 ：getBean(“&book17”)

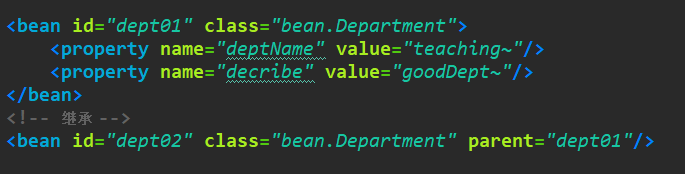
Bean获取本身的id：(获取ApplicationContext：ApplicationContextAware)



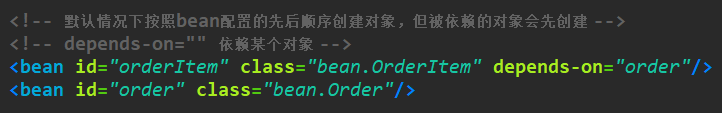
当实例化当前bean后，会回调setBeanName方法，将当前bean的name赋值给当前Bean



实现abstract=”true”的Bean作为抽象Bean，不能被实例，只能被其他bean继承



Bean与Bean之间也可以继承和依赖







通过scope指定bean的作用域

Spring会跟踪singleton的bean从创建一直到销毁，而其他创建好后就不管了，singleton的bean会被Spring放入到一个Map容器中管理

在bean标签可以指定destroy-method属性，当bean销毁之前需要做的处理，也可以让bean实现DisposableBean接口，重写destroy方法(只对Singleton的Bean有效)

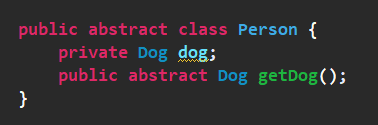
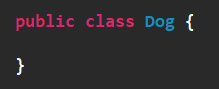
在bean标签可以指定init-method属性，当bean创建只会进行初始化操作，也可以让bean实现InitializingBean接口，重写afterPropertiesSet方法，@PostCut

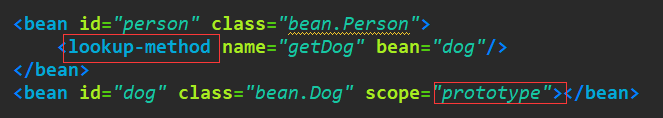
Singleton的bean依赖property的bean

当singleton的bean依赖property的bean时，先会去创建singleton的bean，然后再创建property的bean，那么singleton的bean就会依赖第一次注入的property的bean，下次就不会在注入property的bean，但是每次又需要依赖不同的property的bean，解决：

方式一：放弃依赖，获取singleton后，主动获取property的bean，但是这样就会在singleton中引入IOC容器，污染了代码(bean获取ApplicationContext，BeanFactory/ApplicationContextAware)

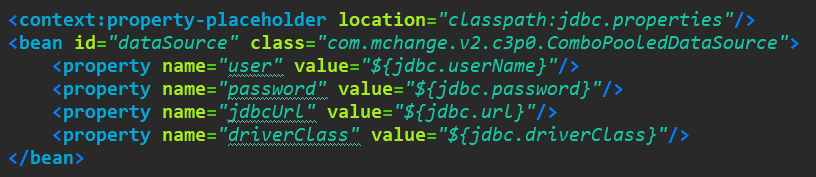
方式二：利用方法注入，使用lookup方法注入



当Spring检查到bean标签有lookup-method属性时(@lookup注解)，会为当前bean创建动态代理类(CGLIB)，并且重写getDog方法，在容器中获取dog，其实底层实现和方式一类似，在重写的getDog方法中获取IOC，再在IOC中获取dog

这里为什么abstract类可以实例化，这里实例化的bean不是Person，而Person的代理类(bean.Person$$EnhancerBySpringCGLIB)，而且这里是不是abstract都无所谓，因为根本执行的不是Person的getDog方法，而是代理类的。

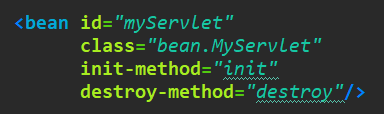


通过context:property-placeholder引入外部资源文件

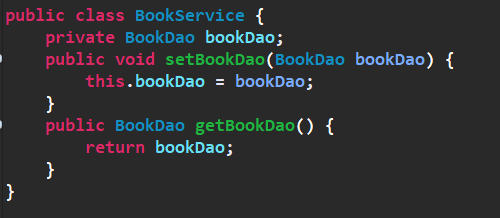


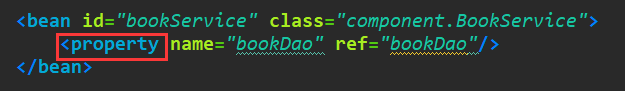
获取ioc容器中的bean

spEL和EL，JSTL语言的用法类型，属于查询语句(SpEL基于ApplicationContext，XMLBeanFactory中无效)



创建好Bean之后，会调用init方法，当容器销毁时，会调用destroy方法(destroy和实现DisposableBean接口类似)





手动装配(ref)



自动装配：autowire指定根据byType/byName

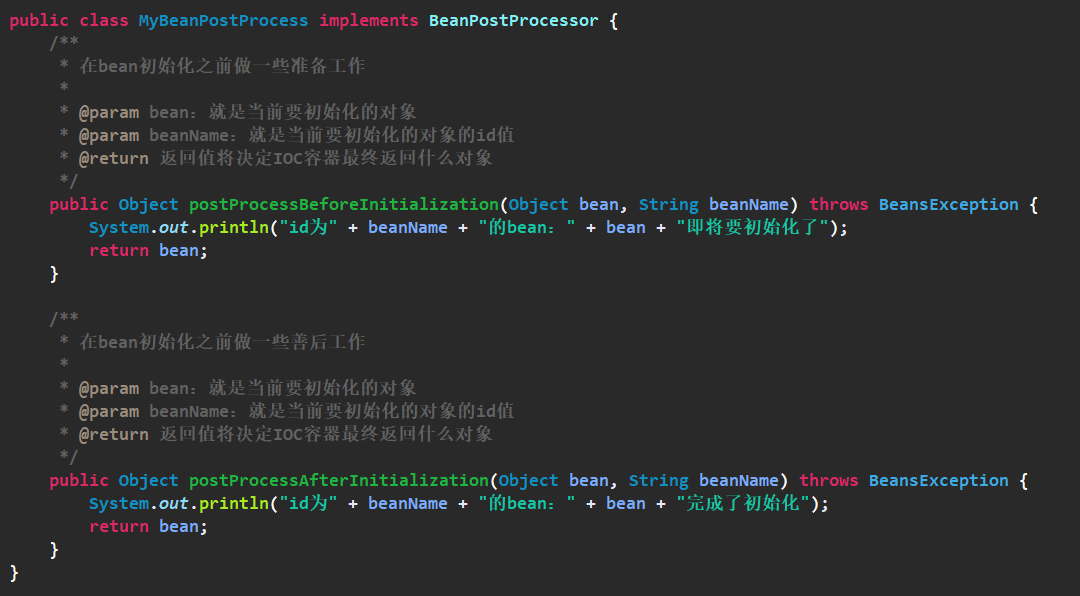
byType：要求IOC中唯一

但是一般很少用自动装配，不透明，可以在bean标签中使用autowire-candidate属性设置bean是否作为自动装配选项

后置处理器：后置处理器基于ApplicationContext，在XMLBeanFactory中无效

后置处理器分为Bean后置处理器和BeanFactory后置处理器

Bean后置处理器：实现BeanPostProcessor接口，并重写它的方法



创建完成后需要注册(有无id都无所谓)



注册完后，每个bean在实例化之前和实例化完成之后都会调用Bean后处理器相应的方法

Bean后处理器的使用：

Spring提供了两个常用的后处理器：

BeanNameAutoProxyCreator：根据Bean实例的name属性创建Bean实例的代理

DefaultAdvisorAutoProxyCreator：根据Advisor对容器中所有的Bean实例创建代理

BeanFactory后置处理器：

BeanFactoryPostProcessor，

void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)和Bean后处理器差不多，Spring容器加载完后执行

propertyPlaceholderConfigurer：

容器后置处理器，负责读取Properties属性和<context:property-placeholder location=*"路径"*/>差不多

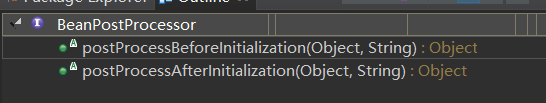
当bean标签中有$的时候，就会去读取properties文件对应的值

PropertyOverrideConfigurer：

容器后处理器，比propertyPlaceholderConfigurer的功能更强大，properties属性文件指定的信息可以直接覆盖Spring配置文件中的原数据，<context:property-override location=*"路径"*/>的简化配置

后置处理器：

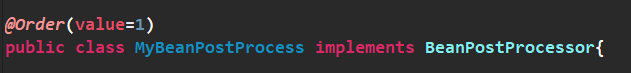
BeanPostProcessor



postProcessorBeforeInitailization：在Bean实例化，依赖注入之后，自定义初始化方法之前执行

bean实例化，依赖注入，postProcessorBeforeInitailization，自定义初始化方法(init-method指定的方法)

postProcessorAfterInitailization：bean实例化，依赖注入以及自定义初始化方法之后调用

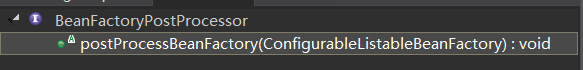


在配置文件声明BeanPostProcessor后，所有的Bean在创建完后都会执行所有后置处理器，可以在定义的后置处理器声明Order注解，值越小越先执行

**Bean初始化过程：new，依赖注入，postProcessorBeforeInitailization，init方法，postProcessorAfterInitailization**

**Bean的销毁：desroty-method(只针对Singleton)，等待GC，finalize**

BeanFactoryPostProcessor：



IOC允许BeanFactoryPostProcessor在容器实例化任何bean之前读取bean的配置信息，并且可以修改它的配置信息，每个Bean在实例化之前都会被BeanFactoryPostProcessor执行，同样BeanFactoryPostProcessor也可以有多个，通过Order属性声明优先级

使用注解：

@Component

普通组件，当前Bean会被Spring容器管理

@ Respository

持久化层

@Controller

控制器层

@Service

业务逻辑层

Spring没有标识组件的能力，使用不同组件只是方便开发人员区分



扫描当前包下的所有类，可以使用其他属性限制扫描的类

Base-package：指定扫描的包

<context:component-scan> 元素会自动注册一个bean的后置处理器：AutowiredAnnotationBeanPostProcessor的实例。该后处理器可以自动装配标记了@Autowired、@Resource或@Inject注解的属性。

@Autowired注解用于自动装配，未找到依赖的bean时会抛出异常，但是添加属性required=false时，当找不到就不会装配，但是在使用的时候可能会抛空指针异常。

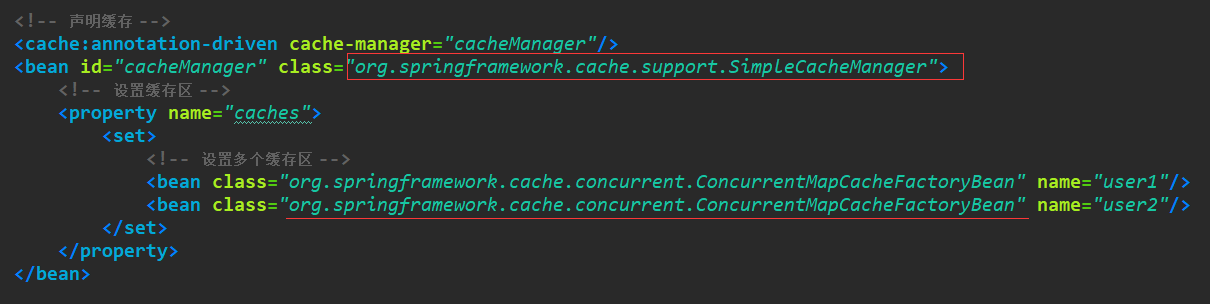
@Qualifiter：为注入的bean指定name，不过也可以在在@ Component等组件中命名

Spring的缓存：

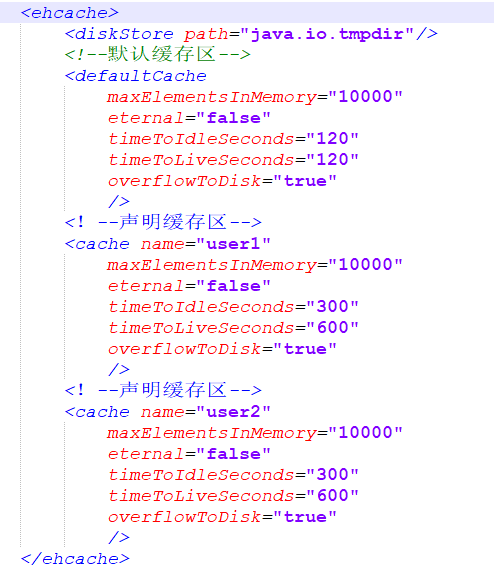
Spring3.1之后可以对Bean和Bean中的方法进行缓存

实现：

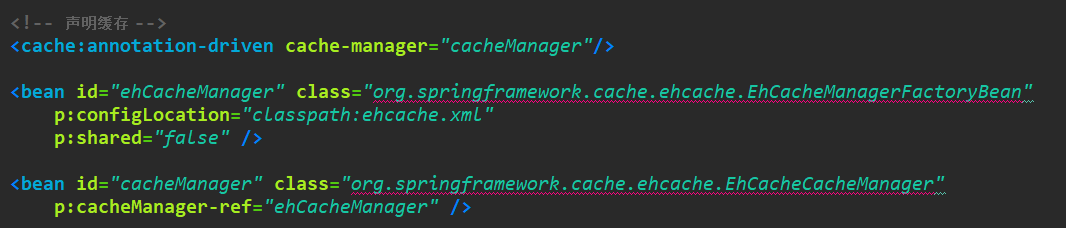
Spring内置缓存是一种内置缓存，底层直接使用JDK的ConcurrentMap缓存，但是内置不好



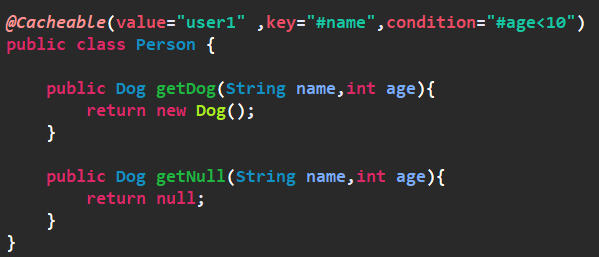
EhCache



配置缓存文件



Spring使用*EhCacheCacheManager*作为EhCache缓存管理器

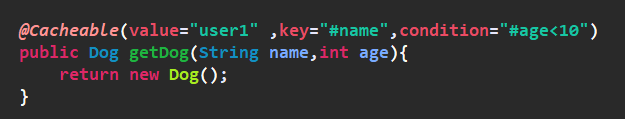


缓存类：当第一次执行getDog方法后，返回一个Dog值，会将方法的参数，返回值缓存到user1缓存区，当下次再调用Person的任意方法时，只要参数一样，直接返回缓存中的Dog，无论是调用getDog还getNull，只要参数一样就直接返回Dog，方法都不执行。

Key：当方法传入的name数一样，也直接返回缓存中实例

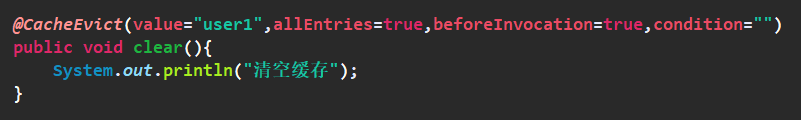
Condition：当Condition表示的SpEL表达式为真时，才执行缓存的方法

缓存方法：



差不多，只是缩小了范围，现在是方法名必须一样

清空缓存



Value：指定清空的缓存区域

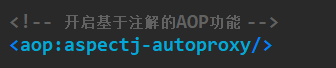
allEntries：是否清空整缓存区

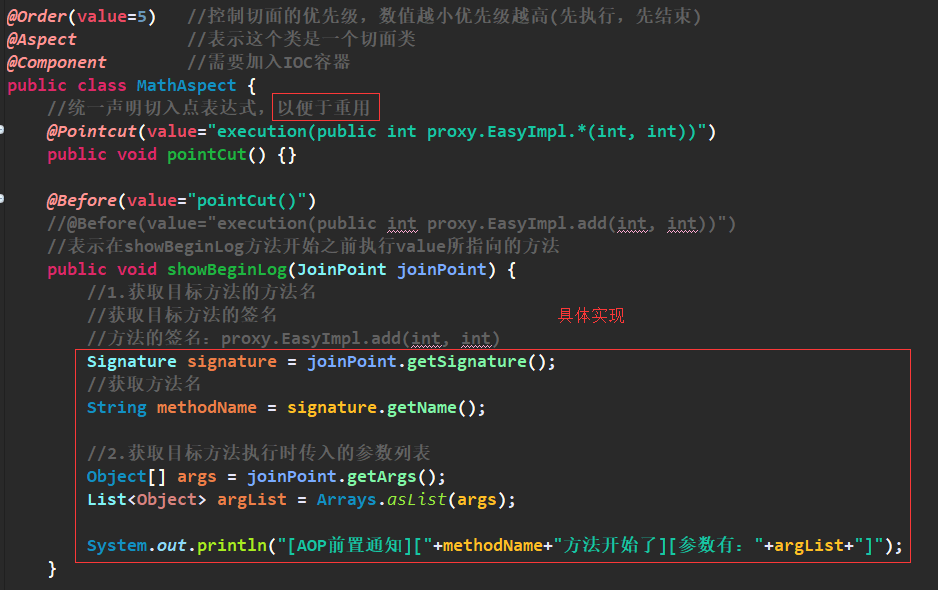
beforeInvocation：是否在在方法执行之前清除缓存

condition：SpEL表达式，清除缓存的条件

AOP：动态代理的实现

AOP是基于Aspect





当IOC容器初始化Aspect切面之后，就会为那些bean创建代理，这里就会为proxy.下的EasyImpl创建代理类

@Aspect：当前类是一个切面类

@Component：交由Spring管理

@Order：切面的优先级，一个方法被有多个切面时，指定切面调用的优先级，值越小优先级越高

@Pointcut：声明切面的切点，value是Aspect表达式，这里使用@Pointcut中作用是重用切点，可以在每个切面上声明Aspect表达式，但是这样不方便复用

Aspect支持的5中通知注解：

@Before：前置通知，在方法调用之前执行

@After：后置通知，在方法执行之后执行

@AfterRunning：返回通知，在方法返回结果之后执行

@AfterThrowing：异常通知，在方法抛出异常之后执行

@Around：环绕通知，围绕方法执行(执行以上所有)

Aspect表达式：

execution(Public int proxy.EasyImpl . \*(int, int))

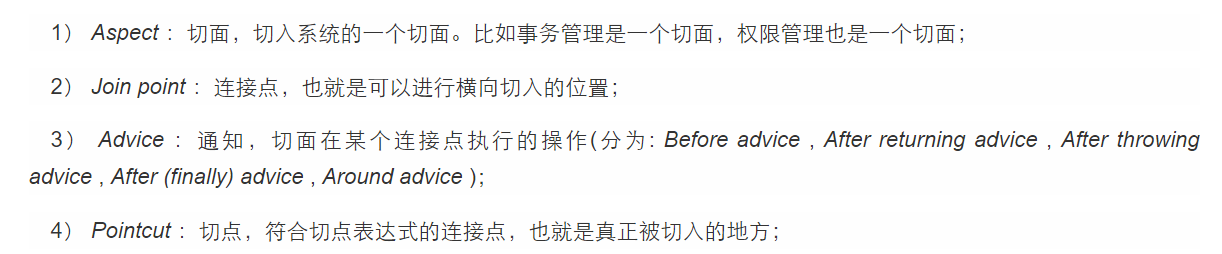
execution(权限修饰符 返回值类型 全类名.方法(参数列表))

execution(\* \* proxy.EasyImpl.\*(int,..))：proxy.EasyImpl下的任意修饰符，任意返回值，方法参数第一个是int类型，其他参数，参数个数任意，类型任意的方法

execution(\* \* \*.\*(..))：所有方法

前置通知：方法执行之前通知

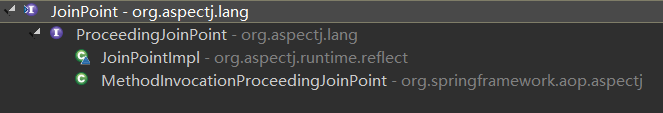




**记住这几个概念**

Join point就是目标方法，可以获取目标方法的一些信息

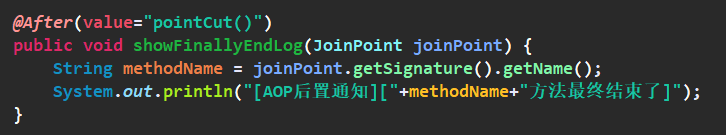
方法名，参数等信息都是封装到JoinPoint接口中的



Signature signature = joinPoint.getSignature()：获取方法的签名，可以获取方法名String methodName = signature.getName()

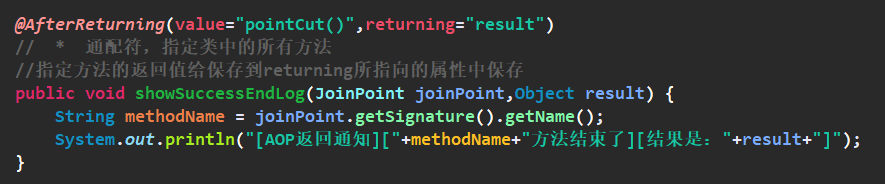
Object[] args = joinPoint.getArgs()：获取方法参数

后置通知：方法执行完之后或者抛出异常时通知



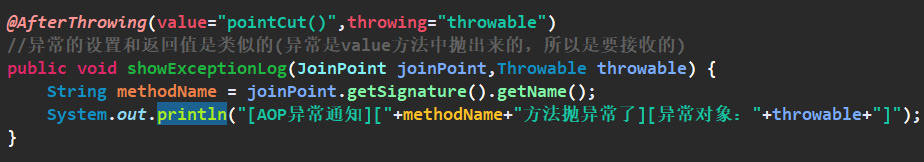
返回通知：后置通知是方法正常返回和抛出异常都会执行，如果只想正常返回时通知，则可以使用返回通知

返回通知在方法执行完之后通知，在后置通知执行完后执行



result是方法的返回结果，returning的名称必须方法的返回值名称一致

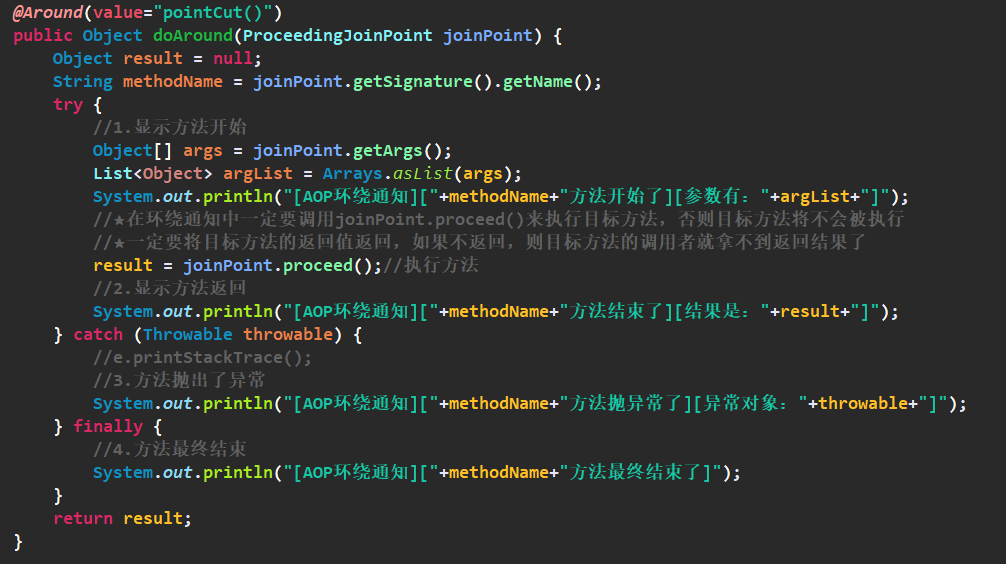
异常通知：抛出异常的时候执行，在后置通知之后执行



环绕通知：

正常执行：环绕通知—前置通知---方法内部----环绕通知(返回结果)----环绕通知(方法结束)---后置通知----返回通知

抛出异常：环绕通知---前置通知---环绕通知(异常)---环绕通知(方法结束)---后置通知---返回通知



参数是ProceedingJoinPoint

//在为EasyImpl声明切面的时候，只能使用接口获取(采用JDK动态代理时，如果使用CgLib代理则不存在)

public interface MathCalculator {}

public class EasyImpl implements MathCalculator{}

MathCalculator calculator = ioc.getBean(MathCalculator.class)：在使用Class获取类的时候，只能通过接口获取，而不能使用EasyImpl.class，在IOC中实际上不是EasyImpl的实现对象，而是代理后的，所以通过EasyImpl.class是不能获取到的，但是代理的对象任然实现了接口，所以通过接口获取到，但是要求唯一，也可以通过name的方式获取

MathCalculator calculator = (MathCalculator)ioc.getBean("easyImpl")

AOP动态代理的方式：JDK动态代理和GLIB，默认采用jdk动态代理，使用cgli时，导包+配置，在开启AOP的时候，设置

proxy-target-class为true

<aop:aspectj-autoproxy proxy-target-class="true"/>

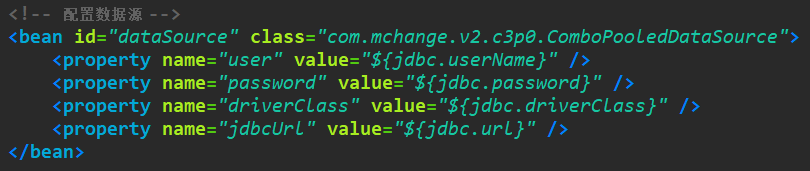
同样也可以使用XML代替注解

jdbcTemplate：

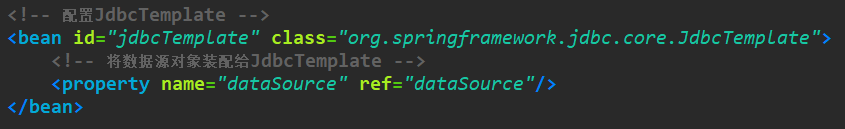
jdbcTemplate是小型的ORM框架和DBUtils类似

使用：

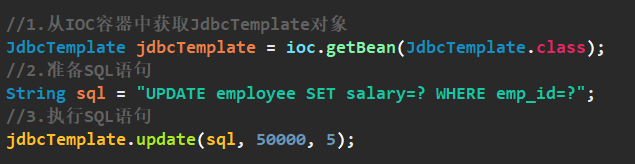
创建DataSource



配置jdbcTemplate



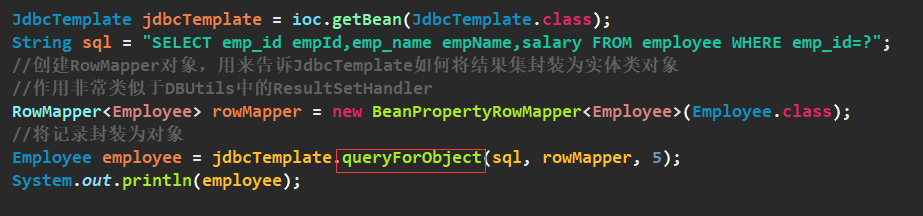
update(增，删，改)



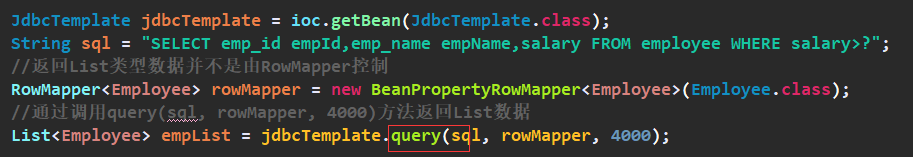
批量增删改：



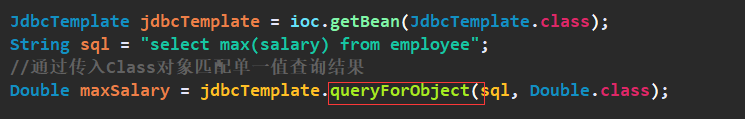
返回实体类



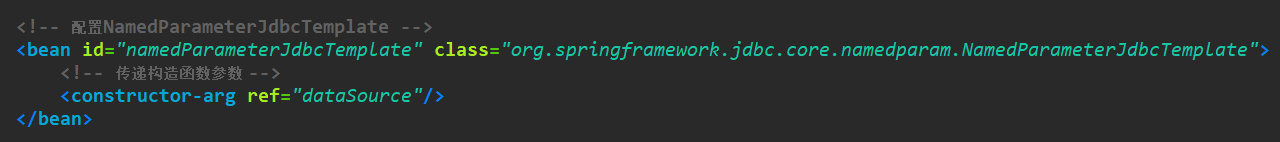
返回实体类集合

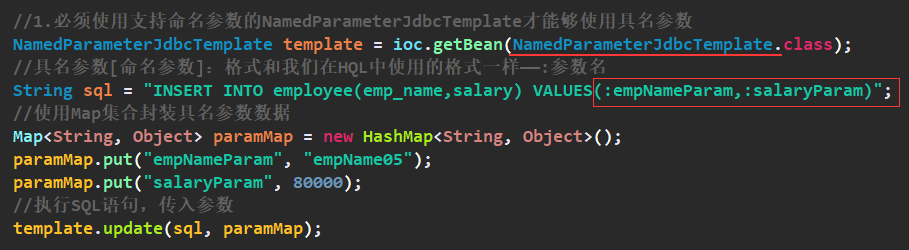


queryForObject(默认返回Object)



使用命名参数：





SqlParameterSource形式传入参数值



声明式事务：

Spring提供了数据库的事务

Spring的声明式事务基于AOP的方式实现

开启事务----前置通知

提交事务----后置通知

回滚事务----异常通知

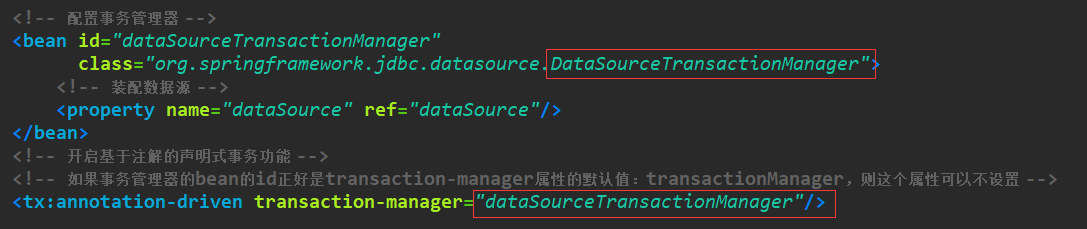
Spring的核心事务管理抽象是PlatformTransactionManager

PlatformTransactionManager的主要实现类：

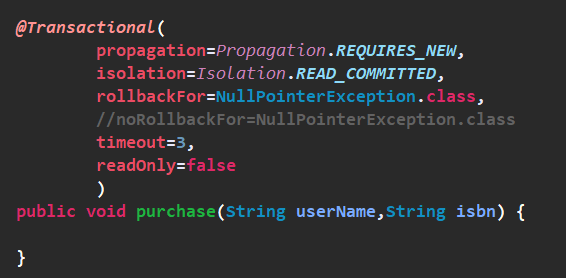
DataSourceTransactionManager：通过JDBC存取

JtaTransactionManager：服务器端进行事务管理

HibernateTransactionManager：Hibernate进行事务管理



在Spring的配置文件声明事务，并且开启基于注解的事务



在方法上添加@Transactional注解，声明当前方法是一个事务

Propagation：指定事务的传播行为

Isolation：设置数据库的隔离级别(RU，RC，RR，串行化)

rollbackFor：当捕获到指定异常时，事务回滚(可以多个)

NorollbackFor：捕获到指定异常时，不回滚(可以有多个)

timeout：事务因某些原因，在一定时间内为执行完，则事务回滚

readOnly：是否只读，只读操作的话，优化数据的效率(读不用开启事务)

事务是对表/行上锁

事务的传播行为：

当一个事务中调用了另外一个事务，那么这两者之间是什么关系

事务A(方法A)，事务B(方法B)

REQUIRED(默认)：方法A声明为REQUIRED，当方法B调用方法A时，如果方法B是一个事务(无论任何事务类型)，则A不创建事务，直接在事务B中执行，如果B只是一个方法，则A开启一个事务

REQUIRED\_NEW：方法A声明为REQUIRED\_NEW，当方法B调用方法A时，无论B是不是一个事务，A都创建事务，如果B也是一个事务，执行过程是：方法B执行到方法A开始时，事务B挂起，事务A执行，当事务A执行完时，事务B继续执行

SUPPORTS：方法A声明为SUPPORTS，当方法B调用方法A时，方法B是一个事务，则A在事务B中运行，如果B不是事务，则A也只是一个普通方法(这也是和REQUIRED的区别)

NOT\_SUPPORTED：方法A声明为NOT\_SUPPORTED，当方法B调用方法A时，如果B是事务，则B挂起，A执行，B不是事务，方法A执行

MANDATORY：方法A声明为MANDATORY，当方法B调用A时，如果B是事务，则方法A在事务B中运行，如果B不是事务，则抛出异常(A必须被事务调用)

NEVER：方法A不能运行在事务中，如果事务调用A，则抛出异常

NESTED：方法A声明为NESTED，当方法调用方法A时，如果B不是事务，A新建一个事务，B是事务，则方法A嵌套在事务B中(嵌套事务)

常用REQUIRED，NOT\_SUPPORTED,SUPPORTS

REQUIRED，REQUIRED\_NEW，NESTED的区别：

REQUIRED：无则生，有则入，方法B是事务，若方法A,B中出现异常，则都回滚，若B不是事务，回滚只发生在A中

NESTED：B是事务，A回滚不会影响B的事务，事务B回滚，事务A回滚

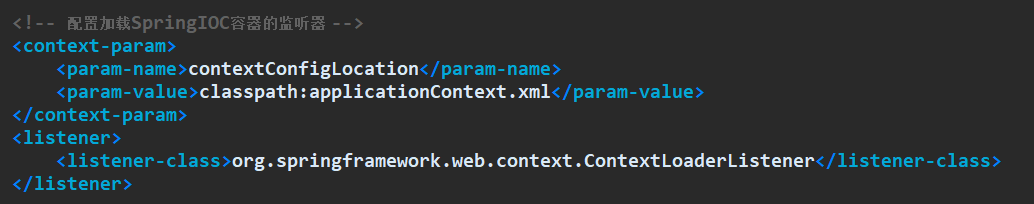
REQUIRED\_NEW：事务A执行时，事务B挂起，所以事务A回滚，不会响应事务B，事务B回滚，也不会影响事务A

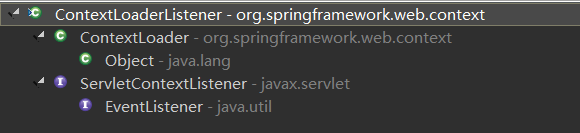
NESTED 开始一个 "嵌套的" 事务,  它是已经存在事务的一个真正的子事务. 嵌套事务开始执行时,  它将取得一个 savepoint. 如果这个嵌套事务失败, 我们将回滚到此 savepoint. 嵌套事务是外部事务的一部分, 只有外部事务结束后它才会被提交.

REQUIRES\_NEW 和 NESTED 的最大区别在于, REQUIRES\_NEW 完全是一个新的事务, 而NESTED 则是外部事务的子事务, 如果外部事务 commit, 嵌套事务也会被 commit, 这个规则同样适用于 roll back.

同样也可以通过XML来配置声明式事务

Spring整合web框架：





Spring是基于web的listener创建的，ContextLoaderListener实现ServletContextListener，当web服务启动时，就会去加载Spring的配置文件，将Spring融入到WEB应用中

Spring加载流程：

BeanFactory bf = new XmlBeanFactory(new ClassPathResource("application.xml"));

ClassPathResource读取配置文件

XmlBeanFactory通过XMLBeanDefinitionReader解析XML，然后通过SAX解析将Resources转化为一个Document(Node的实现类)然后将解析的工作交给DefaultBeanDefinitionDocumentReader

然后调用DefaultBeanDefinitionDocumentReader的parseDefaultElement方法对Document中的每个Node进行遍历解析

根据不同的标签(beans,bean,import,alias)调用不通过的方法解析

解析完后将每个标签封装到BeanDefinition中，然后注册到DefaultListableBeanFactory类的

beanDefinitionMap中(Map<String, BeanDefinition> beanDefinitionMap = new ConcurrentHashMap<String, BeanDefinition>(256);)

加载配置文件完毕后就可以获取了

先看看创建过程

先从beanDefinitionMap根据beanName获取BeanDefinition，然后依次解析标签的属性，调用构造函数，

创建好对象后，检查依赖，如果有依赖，再次调用getBean获取依赖的对象(递归)，最后将创建好的单例bean

存放到一个ConcurrentHashMap中

获取：首先在singleton的ConcurrentHashMap中获取，如果没有，再去创建，如果获取到，要去检测当前bean是不

是一个FactoryBean(getObjectForBeanInstance)，如果是，还要调用他的getBean方法获取真正的bean

Spring解决循环依赖：

A--->B----->C---->A

在创建A之前，会告知Spring，当前bean正在被创建(beforeSingletonCreation会向map中添加一条创建bean的记录)

调用了A的构造方法后，寻找A的依赖(此时A还没被创建完)，然后将获取A的Factory放入到map类型的singletonFactories中

然后创建B，告知Spring B在创建中，调用B的构造方法，并将暴露一个获取B的ObjectFactory，并放入singletonFactories中

然后创建C，告知Spring C在创建中，调用C的构造方法，并暴露一个获取C的ObjectFactory，并让入singletonFactories中

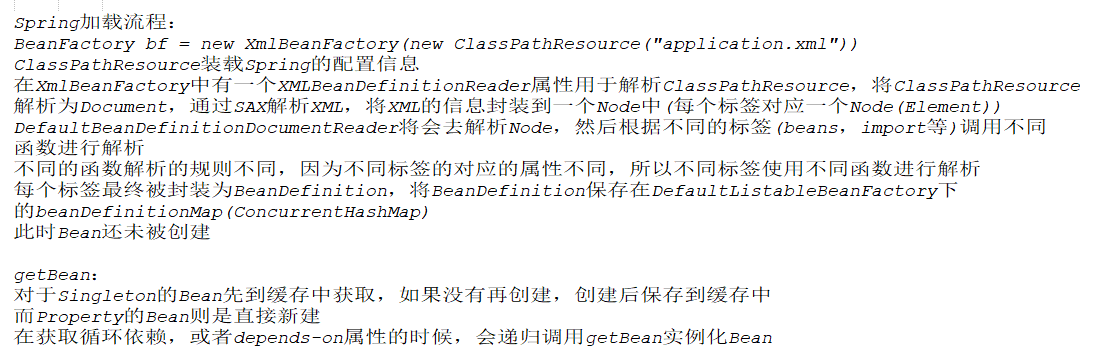
C获取依赖A，发现A在创建中，直接调用singletonFactories中的获取A的ObjectFactory获取A，此致C创建完，然后倒过去创建B，再创建A(获取依赖本身就是一个递归的过程)

为什么使用构造注入不行：在调用构造函数的时候，参数必须已经存在，只能采用设置注入

prototype的bean为什么不行：因为Spring不会对prototype的bean进行缓存，C在获取A的时候，又会重新创建，而不是读取已创建的，这样就会导致循环

Bean 的配置方式: 通过全类名（反射）、通过工厂方法（静态工厂方法 & 实 例工厂方法）、FactoryBean

IOC容器对Bean的声明周期：创建Bean-->为bean注入依赖-->bean的后置处理器-->init方法-->后置处理器-->使用bean-->destroy



BeanFactory和ApplicationContext的区别：

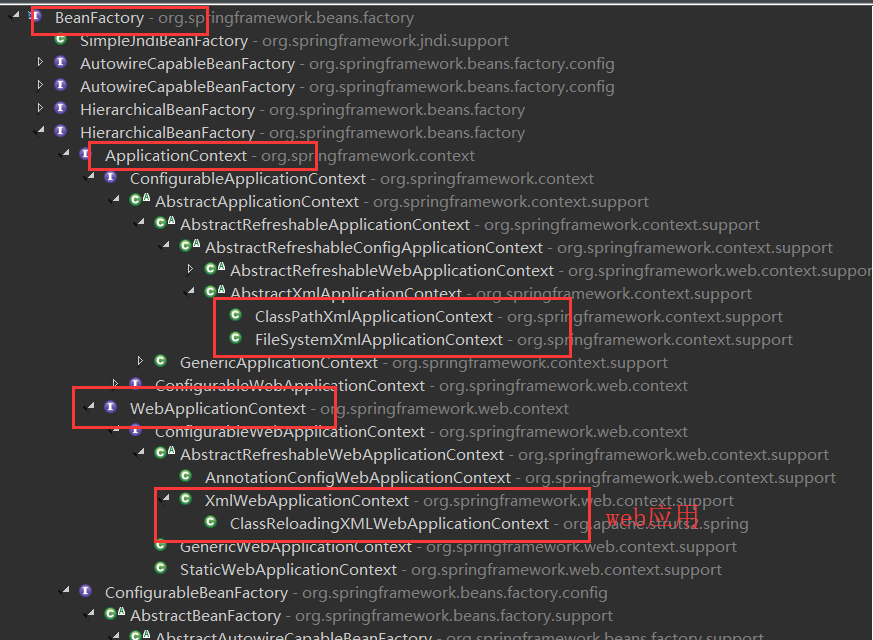
BeanFactory和ApplicationContext都是接口

Spring使用BeanFactory来实例化，配置和管理Bean，BeanFactory是IOC的核心接口，BeanFactory提供了框架的配置及基本功能，但是不支持web功能和Spring的aop功能

ApplicationContext接口是BeanFactory接口的子接口，在BeanFactory的基础上实现了：

国际化信息访问，事件的传播特性(AOP的特性)，适用于web应用

BeanFactory采用延迟加载机制，即调用getBean的时候才会去实例化Bean(这样会导致只有在使用的时候才会发现Spring的配置问题)，而ApplicationContext则是在启动容器的时候一次性就创建所有Bean(可以在启动的时候就发现一些Spring配置的错误)，所以ApplicationContext在启动的时候会比较慢，占用一定的内存空间

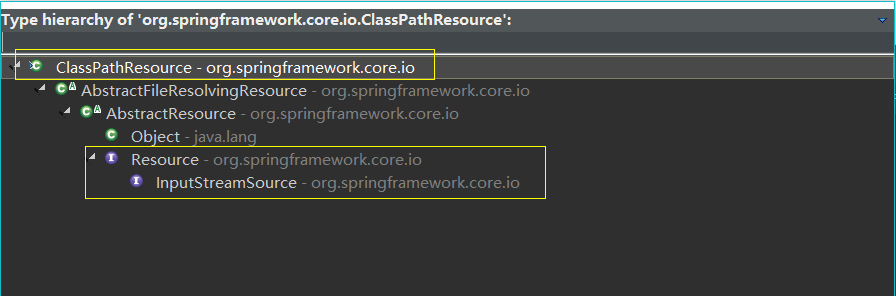


# 源码分析

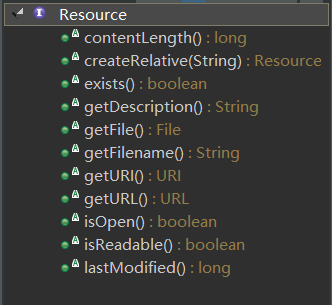
**Spring加载xml文档的过程：**

BeanFactory bf = new XmlBeanFactory(new ClassPathResource("application.xml"));

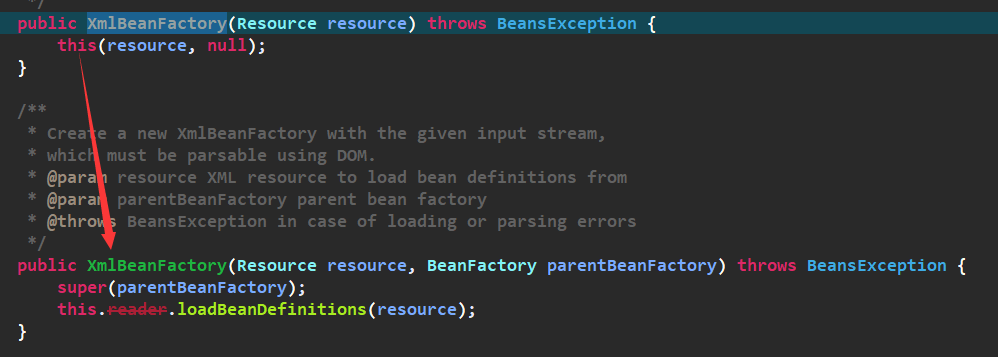
ClassPathResource：Spring中专门读取文件的操作类



在Resource接口中定义了很多文件操作的方法



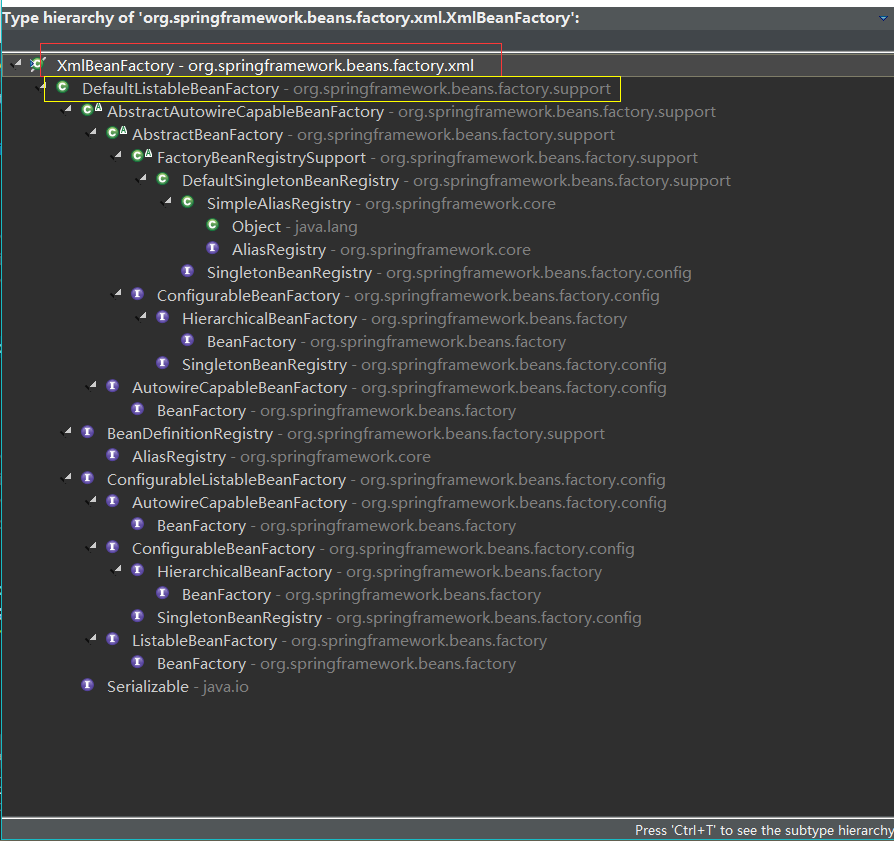
实例化完ClassPathResource后，调用XmlBeanFactory的构造函数



super(parentBeanFactory)：创建父容器(后面会讲到)

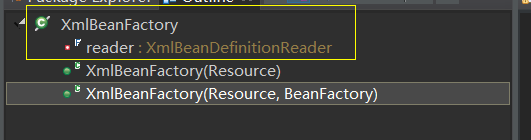
关键代码是**this**.reader.loadBeanDefinitions(resource)，这里的reader是XMLBeanDefinitionReader

XMLBeanDefinitionReader的详解：



XmlBeanFactory继承于DefaultListableBeanFactory

两者之间的区别在于：XMLBeanFactory自定义了一个XML解析器XmlBeanDefinitionReader



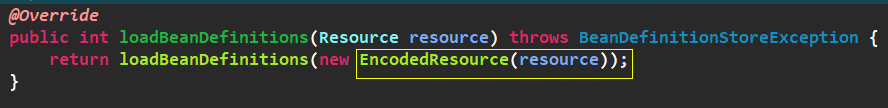
所有读取XML的操作都是通过XMLBeanDefinitionReader操作的



所以在XMLBeanFactory中通过XMLBeanDefinitionReader来读取xml配置文件

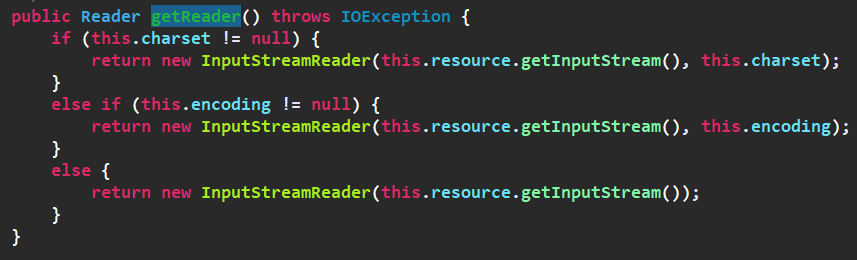
这里的Resource就是子类ClassPathResource对象

加载Bean的过程在loadBeanDefinitions中进行

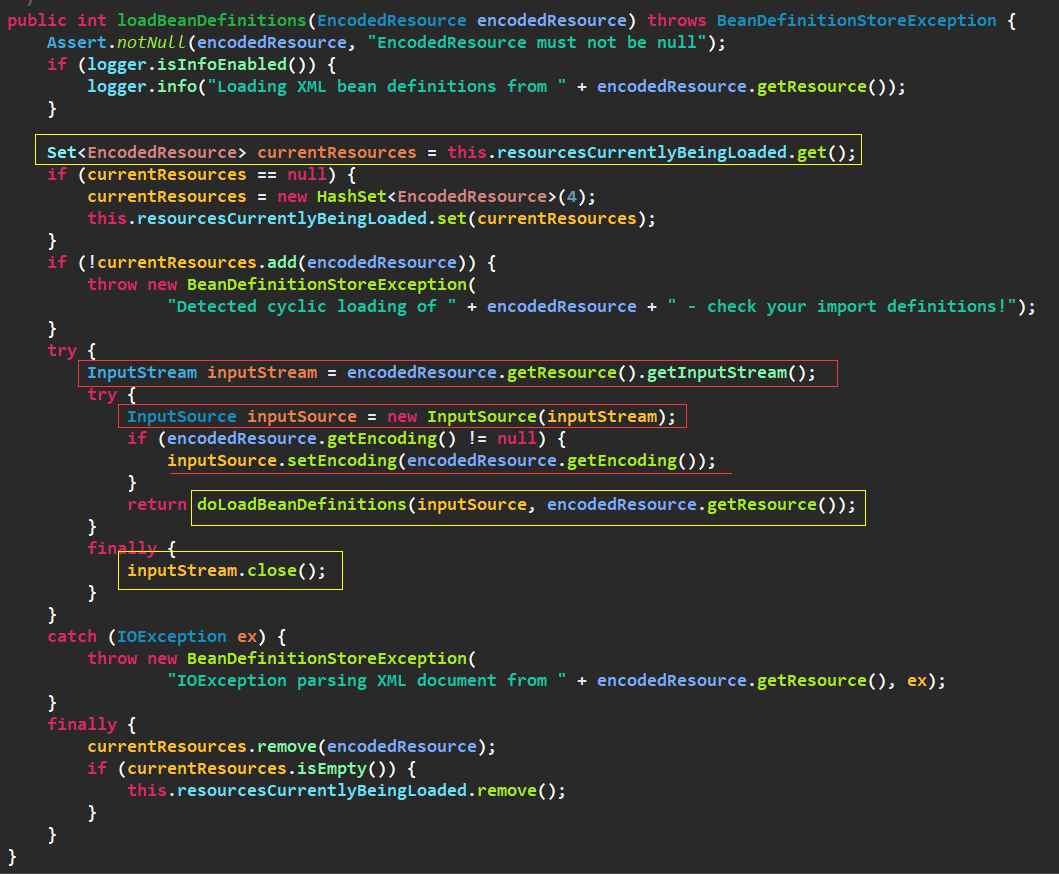


这里将Resource转化为EncodedResource主要是在获取Reader的时候对流进行编码转换





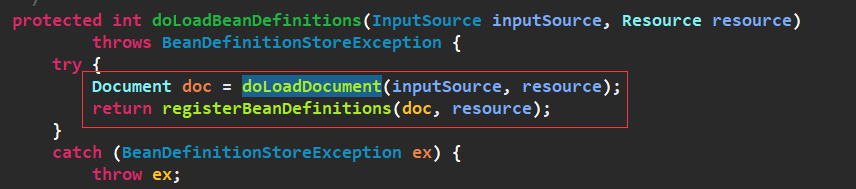
核心loadBeanDefinitions



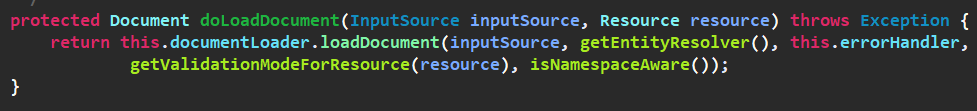
**InputStream** **inputStream** = encodedResource.getResource().getInputStream()：这里实际还是从Resource中获取的InputStream

**InputSource** **inputSource** = **new** InputSource(inputStream)：InputSource是SAX下的一个类，用于SAX解析

核心doLoadBeanDefinitions：



doLoadDocument：将Resource转化为Document，doLoadDocument主要的工作就是通过SAX解析application.xml，并将解析结果封装为Document(Node的实现类)



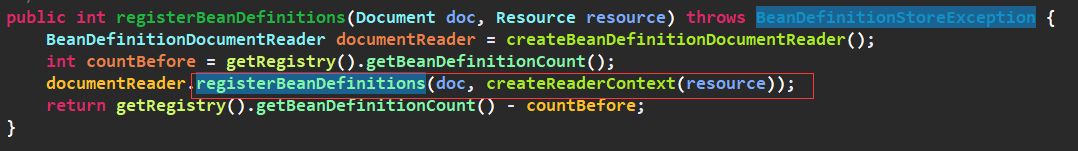
getValidationModeForResource：获取application.xml配置文件的验证规格

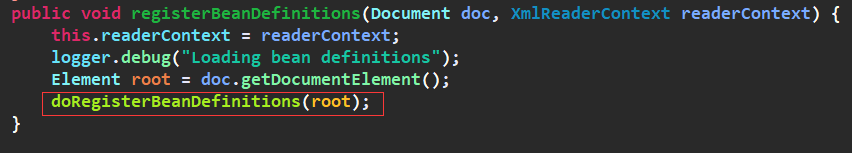
XML的标签是通过DTD和XSD约束进行合法性校验的

EntityResolver：获取DTD/XSD约束文件的路径

loadDocument：通过SAX解析XML文件，返回Document对象(Node)

registerBeanDefinitions：提取及注册Bean(解析Document的Node)

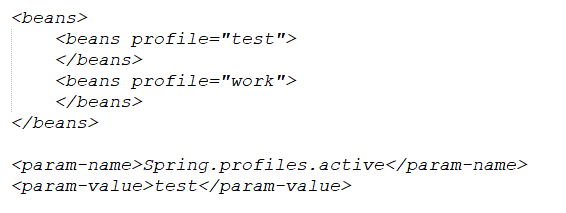




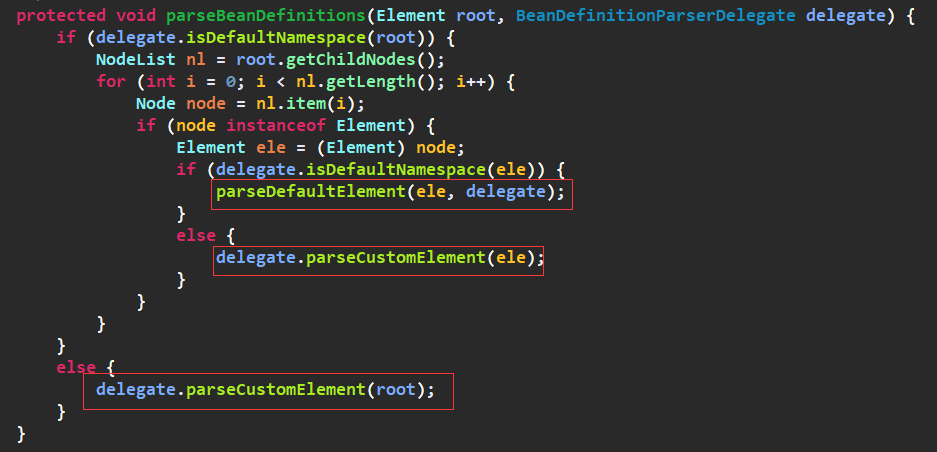
将解析Document的工作交给DefaultBeanDefinitionDocumentReader(这也是为什么要创建documentReader的原因)，至此XMLBeanDefinitionReader的工作完成，其他的操作交给DefaultBeanDefinitionDocumentReader



String profileSpec = root.getAttribute(*PROFILE\_ATTRIBUTE*)：在Spring的配置文件的beans标签下可以再创建一个beans标签，并且有一个profile属性，在实际开发工程中，我们可以定义两套Spring配置文件，用于测试和部署，在集成web项目的时候，可以指定使用哪一套



核心：parseBeanDefinitions



parseDefaultElement和parseCustomElement都是对Bean进行处理获取标签中的属性，详见bean标签的解析及注册。

Spring加载文件流程：

BeanFactory bf = new XmlBeanFactory(new ClassPathResource("application.xml"));ClassPathResource是Resource接口的实现类，通过Resource可以对配置文件进行流的相关操作。

XmlBeanFactory中有一个专门用于解析Resource的类XMLBeanDefinitionReader(大部分操作都是此类中进行的)，然后通过XMLBeanDefinitionReader的loadBeanDefinitions(Resource)解析Resource

XMLBeanDefinitionReader会将Resource解析成为一个Document：通过SAX解析XML，将XML的信息封装到Node类型的Document中

随后将Document交给DefaultBeanDefinitionDocumentReader，解析Document中的每个Node(Bean标签)，之前的操作都是XMLBeanDefinitionReader，解析Node(Bean标签)都是在DefaultBeanDefinitionDocumentReader中

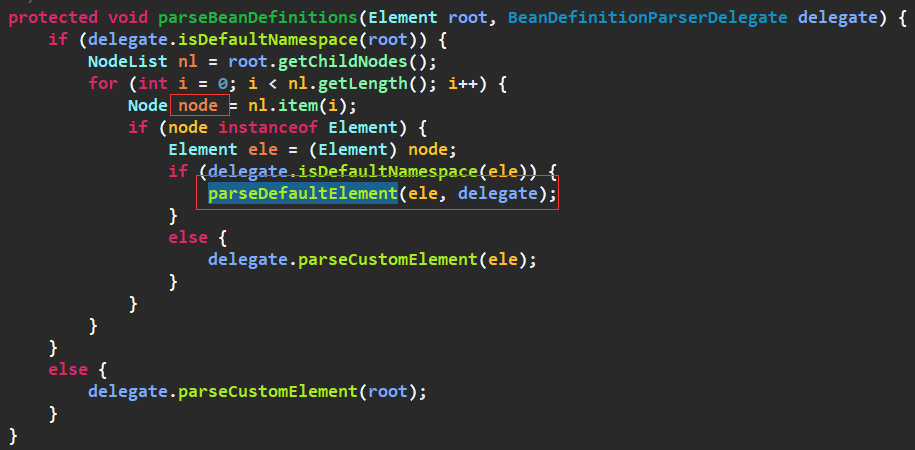
DefaultBeanDefinitionDocumentReader会解析每个Node(bean标签)，及其对应的属性(标签的属性)

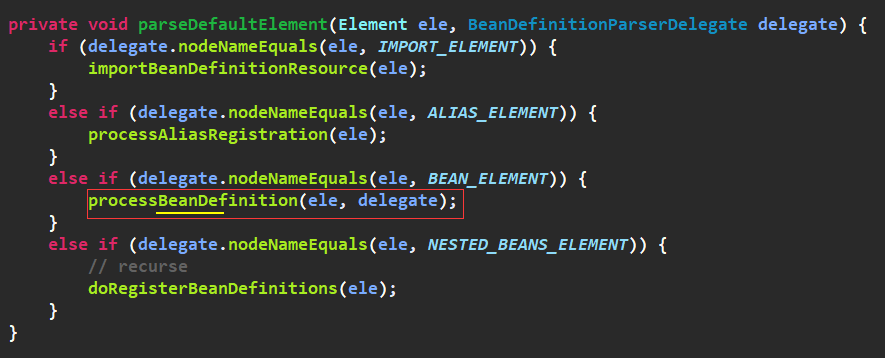
XMLBeanDefinitionReader的作用是将application.xml中的标签封装为Document(Node)，对标签(Node)的解析工作交给DefaultBeanDefinitionDocumentReader

注意：这里还未实例化Bean。

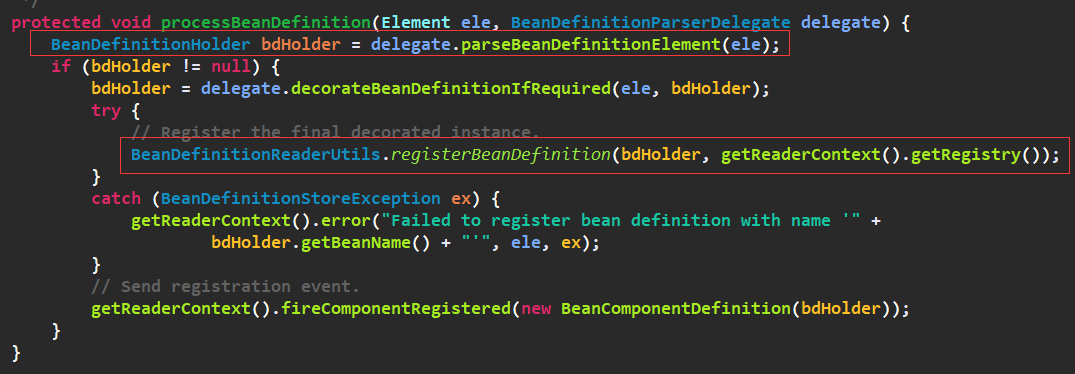
**bean标签的解析及注册：**

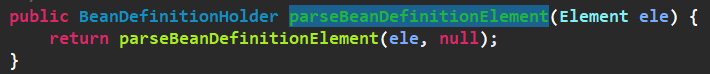
XMLBeanDefinitionReader是将xml封装为Node类型的Document，DefaultBeanDefinitionDocumentReader读取Document并注册BeanDefinition，Node(ELement)的解析交给BeanDefinitionParserDelegate完成

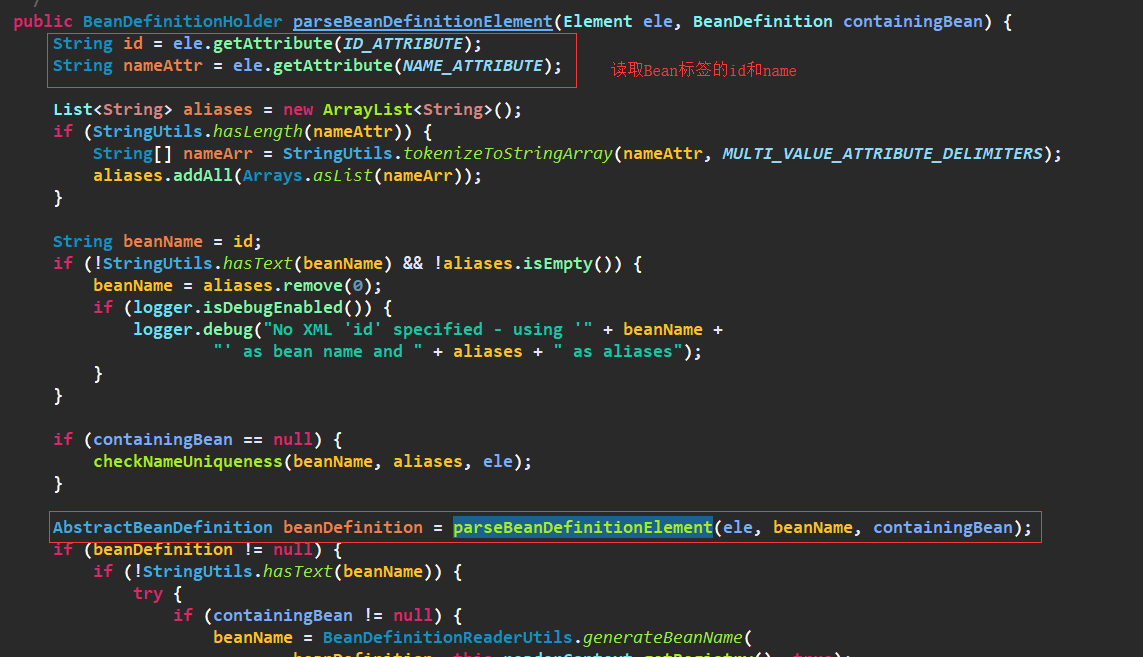




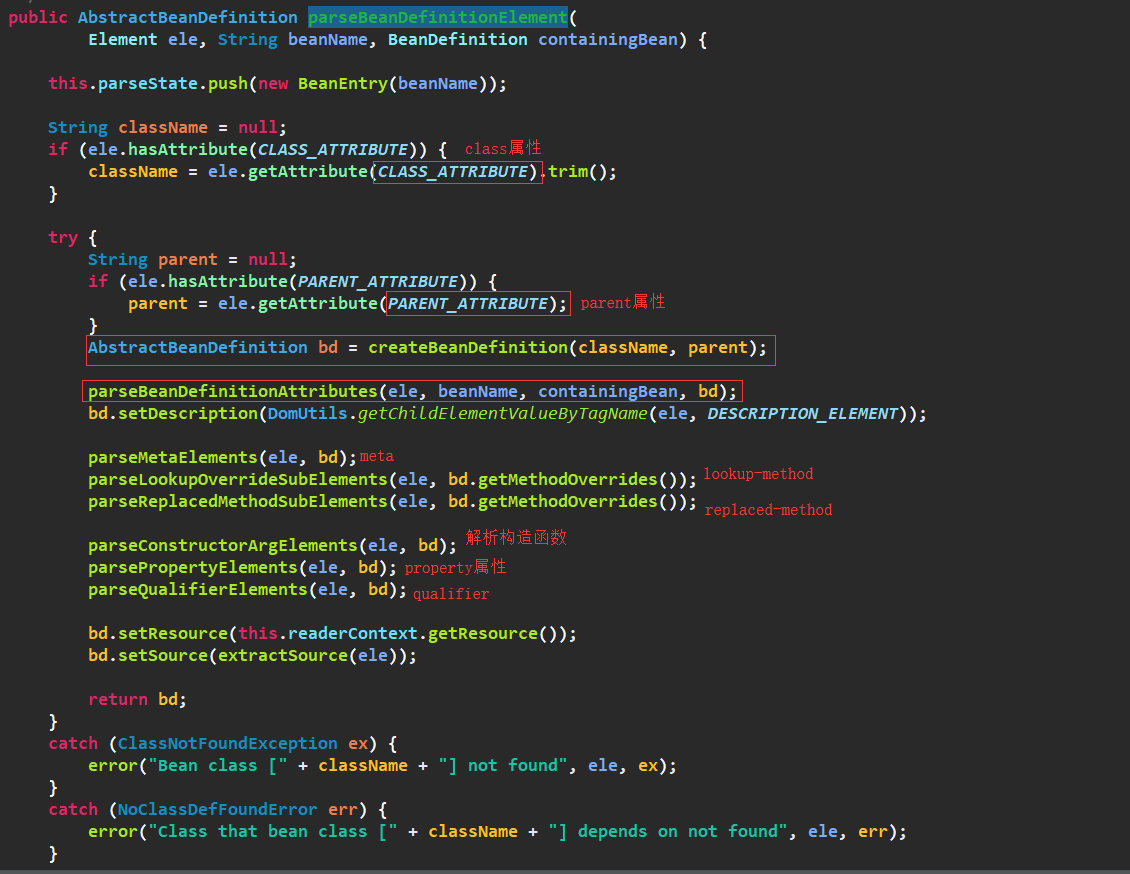
根据不同类型的标签(import，alias，bean，beans)分别进行解析，这里以bean标签的解析过程为例





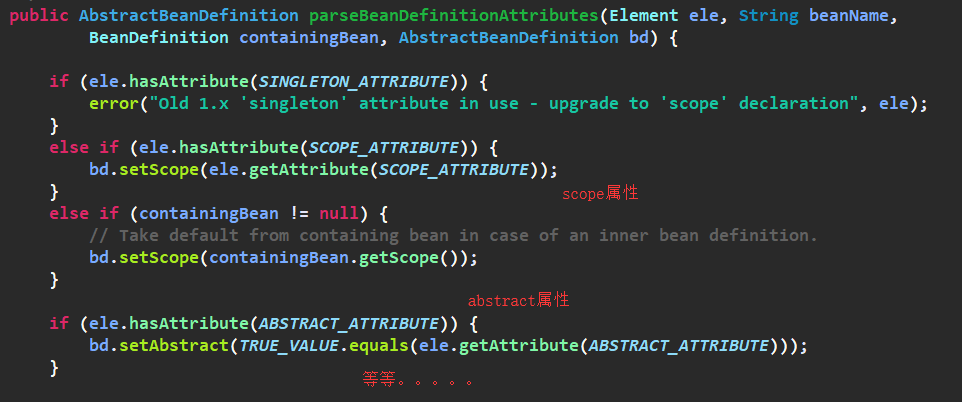


parseBeanDefinitionElement：用于读取其他属性

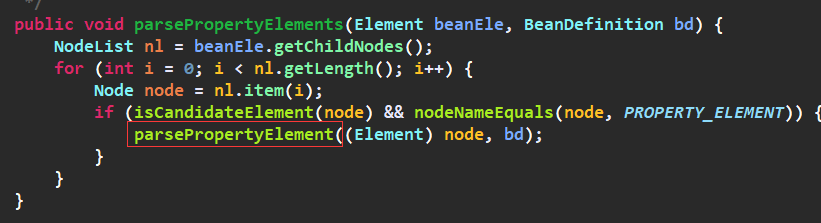


createBeanDefinition：创建 AbstractBeanDefinition，将解析的属性保存到bd中

parseBeanDefinitionAttributes：获取Element中的属性，并将属性封装到bd中(BeanDefinition重要，解释下)

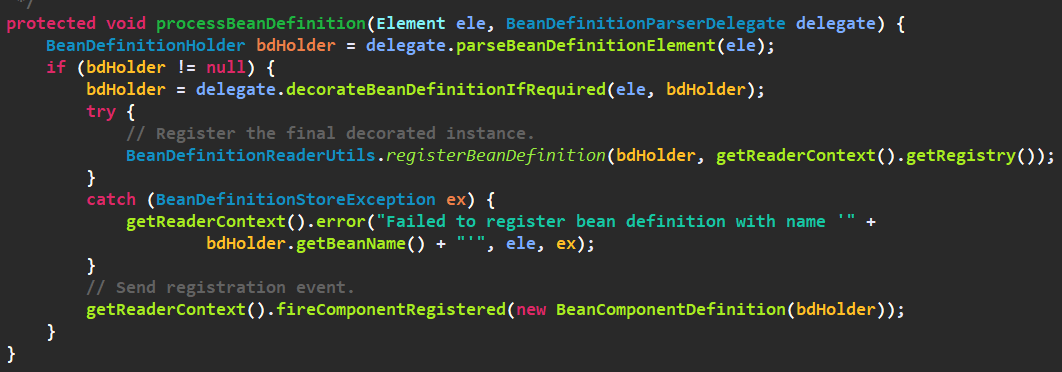


parseXxxELements：用于解析bd中的属性，这里以parsePropertyElements(ele, bd)为例

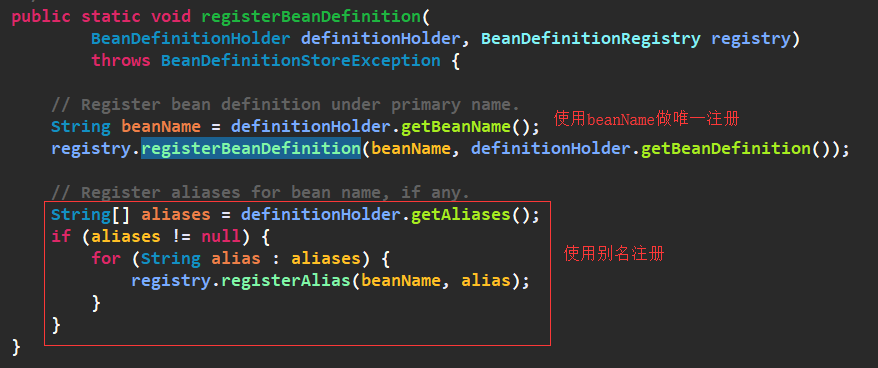


parsePropertyValue：解析property对应的value值

bd.getPropertyValues().addPropertyValue(pv)：将property的value值记录在bd中，到这里就完成了将每个Bean标签的属性的XML到GenericBeanDefinition的转换，在AbstractBeanDefinition(GenericBeanDefinition子类)中都可以找到对应的属性

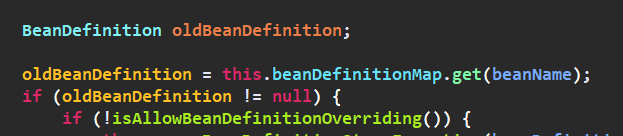


解析完Element后，将属性封装到BeanDefinition中，然后bd又被封装为BeanDefinitionHolder，所以，当前标签的属性最终是封装到了BeanDefinitionHolder中，完成了标签的解析后应该将标签的内容进行注册(就是保存):*registerBeanDefinition*

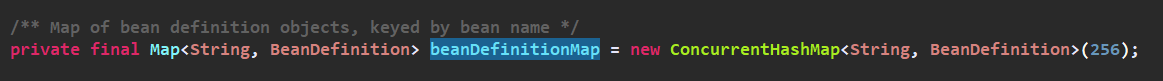


registry.registerBeanDefinition(beanName, definitionHolder.getBeanDefinition()):通过BeanDefinitionRegistry的registerBeanDefinition方法进行注册， registerBeanDefinition调用的是BeanDefinitionRegistry的实现类DefaultListableBeanFactory而这个类和BeanFactory又有莫大的联系。

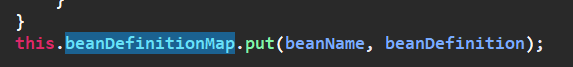
在来看看registerBeanDefinition方法，首先通过beanName查询DefaultListableBeanFactory中是否存在



beanDefinitionMap是一个Map类型，用来保存BeanDefinition



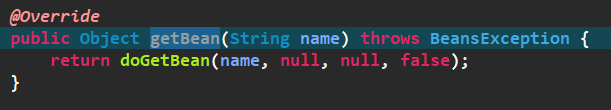
如果oldBeanDefinition为空，则



此致，XML中的一个标签被解析为BeanDefinition，并且保存在ConcurrentHashMap中，然后有去解析下一个标签，并将结果保存在BeanDefinitionRegistry的子类DefaultListableBeanFactory的ConcurrentHashMap中。一个标签对应一个BeanDefinition，XML中的所有标签保存在ConcurrentHashMap中。

**getBean：**

加载完XML就可以获取了



AbstractBeanFactory的getBean

补充：

FactoryBean接口：

|  |
| --- |
| public interface FactoryBean<T> {  获取工厂返回的对象  T getObject() throws Exception;  Class<?> getObjectType();  boolean isSingleton();  } |

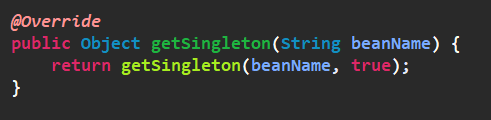
|  |
| --- |
| <bean id=*"book17"* class=*"factory.MyFactoryBean"*>  <property name=*"paramKey"* value=*"key03"*/>  </bean> |

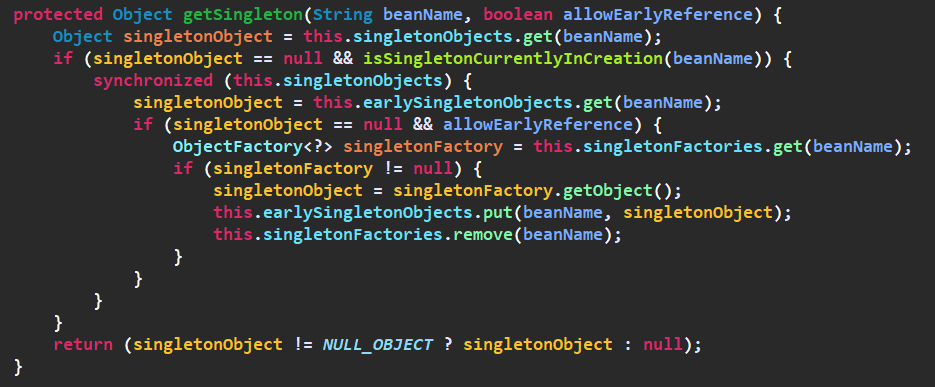
一个类实现FactoryBean并重写getObject方法后，可以通过bean标签直接实例FactoryBean，在获取BeanFactory的getObject返回的对象时，可以直接使用getBean(“book17”)即可，如果想获取FactoryBean的实例。使用getBean(“&book17”)

doGetBean

|  |
| --- |
| protected <T> T doGetBean(  final String name, final Class<T> requiredType, final Object[] args, boolean typeCheckOnly)  throws BeansException {  /\*  **获取对应的beanName：**  **别名**  **去除FactoryBean的&**  \*/  final String beanName = transformedBeanName(name);  Object bean;    /\*  **从缓存中获取，如果Bean的类型是singleton，在创建成功后会被放到ConcurrentHashMap中**  \*/  // Eagerly check singleton cache for manually registered singletons.  Object sharedInstance = getSingleton(beanName);  if (sharedInstance != null && args == null) {  if (logger.isDebugEnabled()) {  if (isSingletonCurrentlyInCreation(beanName)) {  logger.debug("Returning eagerly cached instance of singleton bean '" + beanName +  "' that is not fully initialized yet - a consequence of a circular reference");  }  else {  logger.debug("Returning cached instance of singleton bean '" + beanName + "'");  }  }  /\*  **返回的Bean可能使BeanFactory的实例，但是我们要获取的是BeanFactory的getBean返回的实例**  **获取到Bean后，都需要验证获取的Bean是不是BeanFactory**  \*/  bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);  }  else {  /\*  **判断当前的bean是否是循环依赖**  \*/  // Fail if we're already creating this bean instance:  // We're assumably within a circular reference.  if (isPrototypeCurrentlyInCreation(beanName)) {  throw new BeanCurrentlyInCreationException(beanName);  }    /\*  **containsBeanDefinition(beanName)：从BeanDefinitionMap中检测是否存在beanName，如果不存在，则尝试从parentBeanFactory中检测**  \*/  // Check if bean definition exists in this factory.  BeanFactory parentBeanFactory = getParentBeanFactory();  if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {  // Not found -> check parent.  String nameToLookup = originalBeanName(name);  if (args != null) {  // Delegation to parent with explicit args.  return (T) parentBeanFactory.getBean(nameToLookup, args);  }  else {  // No args -> delegate to standard getBean method.  return parentBeanFactory.getBean(nameToLookup, requiredType);  }  }  if (!typeCheckOnly) {  markBeanAsCreated(beanName);  }  try {  /\*  **根据beanName获取BeanDefinition**  \*/  final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);  checkMergedBeanDefinition(mbd, beanName, args);    /\*  **当前bean标签是否有depends-on属性，先创建依赖的Bean**  \*/  // Guarantee initialization of beans that the current bean depends on.  String[] dependsOn = mbd.getDependsOn();  if (dependsOn != null) {  for (String dependsOnBean : dependsOn) {  if (isDependent(beanName, dependsOnBean)) {  throw new BeanCreationException(mbd.getResourceDescription(), beanName,  "Circular depends-on relationship between '" + beanName + "' and '" + dependsOnBean + "'");  }  registerDependentBean(dependsOnBean, beanName);  /\*  **递归创建依赖的对象**  \*/  getBean(dependsOnBean);  }  }    /\*  **如果当前要创建的bean是singleton的**  \*/  // Create bean instance.  if (mbd.isSingleton()) {  /\*  **注意，这里是内部类**  \*/  sharedInstance = getSingleton(beanName, new ObjectFactory<Object>() {  @Override  public Object getObject() throws BeansException {  try {  return createBean(beanName, mbd, args);  }  catch (BeansException ex) {  // Explicitly remove instance from singleton cache: It might have been put there  // eagerly by the creation process, to allow for circular reference resolution.  // Also remove any beans that received a temporary reference to the bean.  destroySingleton(beanName);  throw ex;  }  }  });  /\*  **检验bean是否是FactoryBean**  \*/  bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);  }    /\*  **当前bean是prototype**  \*/  else if (mbd.isPrototype()) {  // It's a prototype -> create a new instance.  Object prototypeInstance = null;  try {  beforePrototypeCreation(beanName);  prototypeInstance = createBean(beanName, mbd, args);  }  finally {  afterPrototypeCreation(beanName);  }  bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);  }  else {    /\*  **指定scope上实例化bean**  \*/  String scopeName = mbd.getScope();  final Scope scope = this.scopes.get(scopeName);  if (scope == null) {  throw new IllegalStateException("No Scope registered for scope name '" + scopeName + "'");  }  try {  Object scopedInstance = scope.get(beanName, new ObjectFactory<Object>() {  @Override  public Object getObject() throws BeansException {  beforePrototypeCreation(beanName);  try {  return createBean(beanName, mbd, args);  }  finally {  afterPrototypeCreation(beanName);  }  }  });  bean = getObjectForBeanInstance(scopedInstance, name, beanName, mbd);  }  catch (IllegalStateException ex) {  throw new BeanCreationException(beanName,  "Scope '" + scopeName + "' is not active for the current thread; consider " +  "defining a scoped proxy for this bean if you intend to refer to it from a singleton",  ex);  }  }  }  catch (BeansException ex) {  cleanupAfterBeanCreationFailure(beanName);  throw ex;  }  }  // Check if required type matches the type of the actual bean instance.  if (requiredType != null && bean != null && !requiredType.isAssignableFrom(bean.getClass())) {  try {  return getTypeConverter().convertIfNecessary(bean, requiredType);  }  catch (TypeMismatchException ex) {  if (logger.isDebugEnabled()) {  logger.debug("Failed to convert bean '" + name + "' to required type [" +  ClassUtils.getQualifiedName(requiredType) + "]", ex);  }  throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());  }  }  return (T) bean;  } |

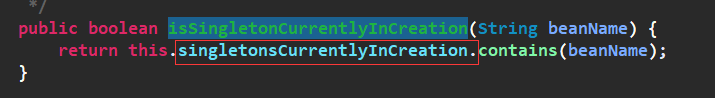
一 从缓存中获取getSingleton(String beanName)：这里的参数只有一个





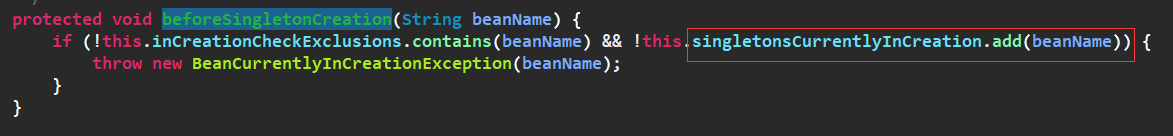
① Object singletonObject = this.singletonObjects.get(beanName)：当singleton的bean创建好后，都会保存到ConcurrentHashMap中，而singletonObjects就是保存singleton的ConcurrentHashMap

② isSingletonCurrentlyInCreation：循环依赖的时候会使用，判断当前beanName的bean是否处于创建状态，在循环依赖的时候会详细讲解



singletonsCurrentlyInCreation是一个集合类型

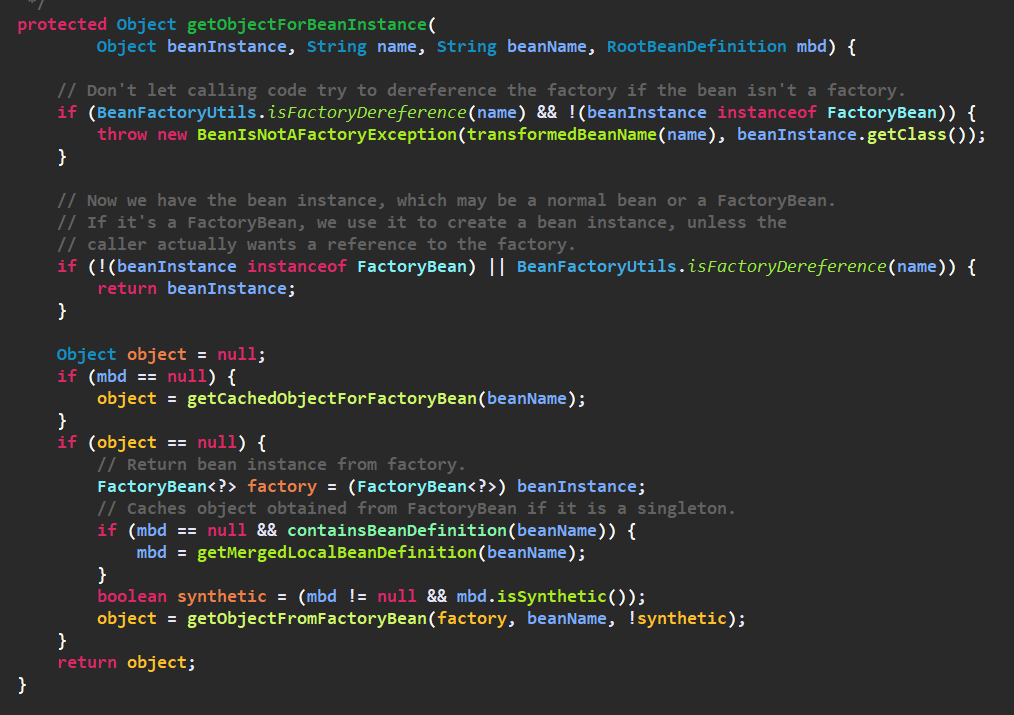
当单例的bean在创建之前有一个beforeSingletonCreation方法，告知Spring当前bean正在被创建



getSingleton主要的作用就是获取ConcurrentHashMap中以实例的singleton的bean

二 getObjectForBeanInstance：

获取到bean后的第一件事就是判断是否是FactoryBean，或者指定了factory-method属性的bean



|  |
| --- |
| if (!(beanInstance instanceof FactoryBean) || BeanFactoryUtils.*isFactoryDereference*(name)) {  return beanInstance;  } |

beanInstance instanceof FactoryBean：判断是否是FactoryBean的实例

BeanFactoryUtils.*isFactoryDereference*(name)：判断name是否是以&开头(以&开头获取的就是FactoryBean或对应的Bean)

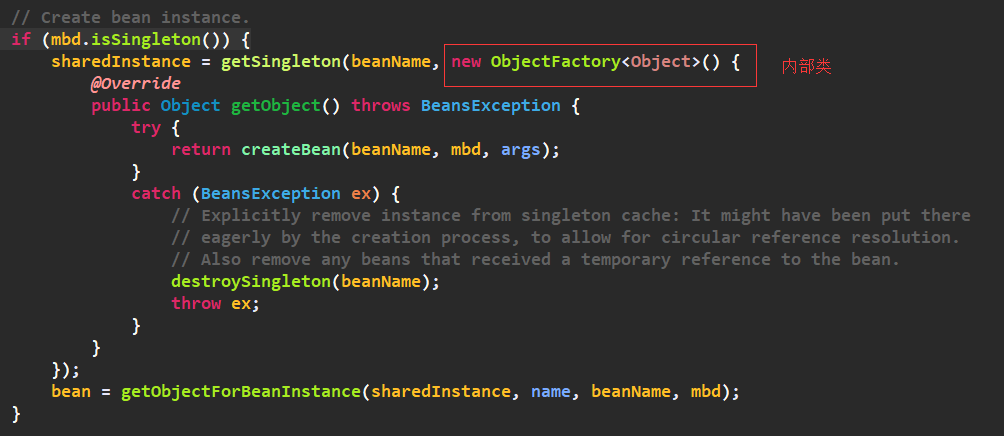
如果不是FactoryBean的实例或者是以&开头，直接返回，如果是FactoryBean的实例并且不是以&开头，则调用getObject方法

1. getCachedObjectForFactoryBean：

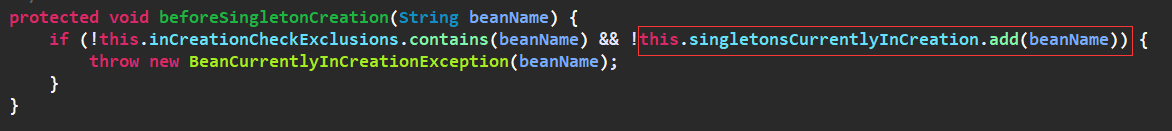
从缓存中获取(后面看)

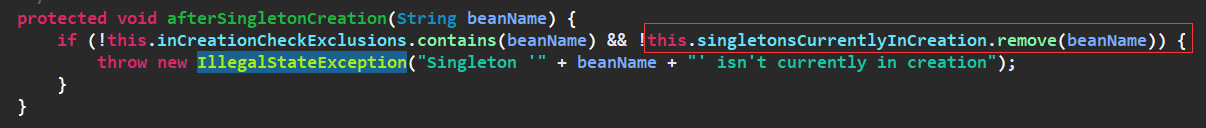
等下倒过来看看

三 从singleton的缓存中没有获取到bean，那么就创建，首先判断的bean的周期，再根据不同的周期实例化bean

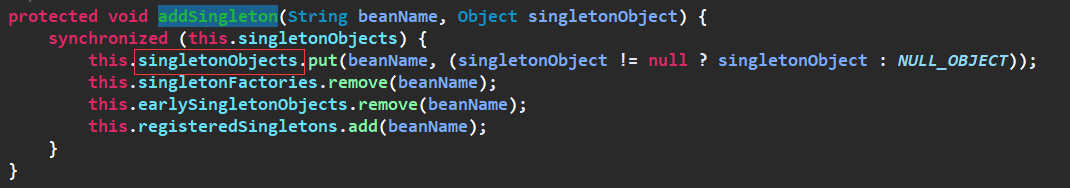


|  |
| --- |
| public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {  Assert.notNull(beanName, "'beanName' must not be null");  synchronized (this.singletonObjects) {  /\*  **任然是先从缓存中获取**  \*/  Object singletonObject = this.singletonObjects.get(beanName);  if (singletonObject == null) {  if (this.singletonsCurrentlyInDestruction) {  throw new BeanCreationNotAllowedException(beanName,  "Singleton bean creation not allowed while the singletons of this factory are in destruction " +  "(Do not request a bean from a BeanFactory in a destroy method implementation!)");  }  if (logger.isDebugEnabled()) {  logger.debug("Creating shared instance of singleton bean '" + beanName + "'");  }  /\*  **当前bean正在创建的记录加入缓存**  \*/  beforeSingletonCreation(beanName);  boolean newSingleton = false;  boolean recordSuppressedExceptions = (this.suppressedExceptions == null);  if (recordSuppressedExceptions) {  this.suppressedExceptions = new LinkedHashSet<Exception>();  }  try {  /\*  **实例化bean的关键**  \*/  singletonObject = singletonFactory.getObject();  newSingleton = true;  }  catch (IllegalStateException ex) {  // Has the singleton object implicitly appeared in the meantime ->  // if yes, proceed with it since the exception indicates that state.  singletonObject = this.singletonObjects.get(beanName);  if (singletonObject == null) {  throw ex;  }  }  catch (BeanCreationException ex) {  if (recordSuppressedExceptions) {  for (Exception suppressedException : this.suppressedExceptions) {  ex.addRelatedCause(suppressedException);  }  }  throw ex;  }  finally {  if (recordSuppressedExceptions) {  this.suppressedExceptions = null;  }  /\*  **移出创建bean的记录**  \*/  afterSingletonCreation(beanName);  }  if (newSingleton) {  /\*  **将创建的bean加入到ConcurrentHashMap中**  \*/  addSingleton(beanName, singletonObject);  }  }  return (singletonObject != NULL\_OBJECT ? singletonObject : null);  }  } |



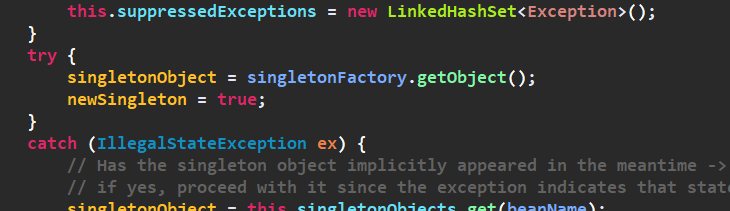


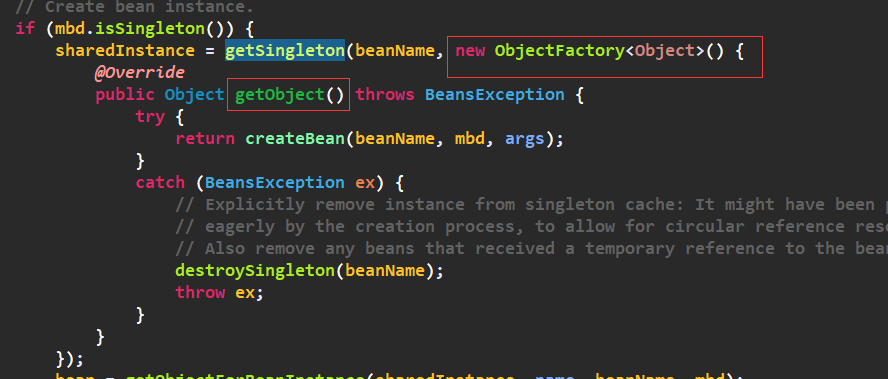
singletonsCurrentlyInCreation：Map结构，用于记录beanName是否正在被创建，用于判断循环依赖



将创建好的singleton的bean放入到singleton缓存的singletonObjects(ConcurrentHashMap)中

createBean



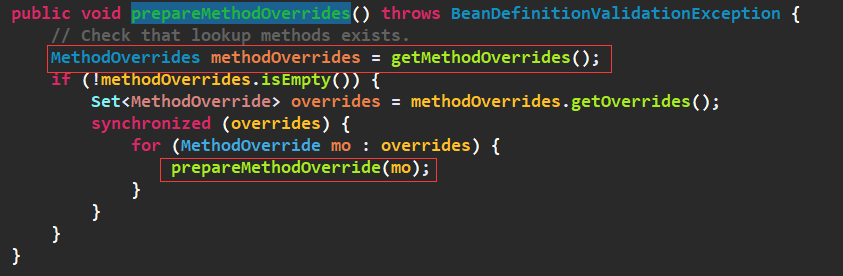


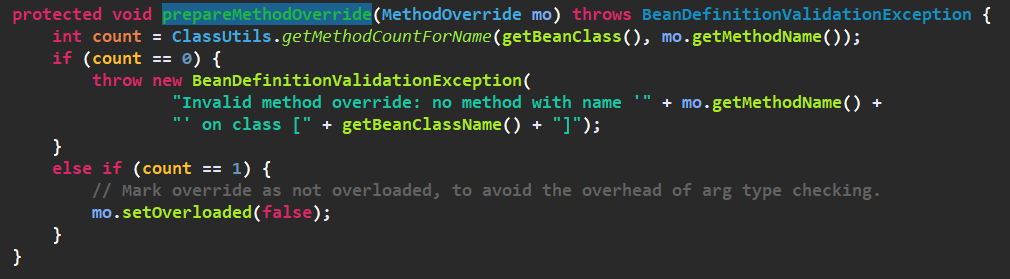
内部类，所以实际上还是调用的是内部类中的getObject方法，createBean是关键方法

|  |
| --- |
| protected Object createBean(String beanName, RootBeanDefinition mbd, Object[] args)  throws BeanCreationException {  if (logger.isDebugEnabled()) {  logger.debug("Creating instance of bean '" + beanName + "'");  }  RootBeanDefinition mbdToUse = mbd;  // Make sure bean class is actually resolved at this point, and  // clone the bean definition in case of a dynamically resolved Class  // which cannot be stored in the shared merged bean definition.  /\*  **加载Class**  \*/  Class<?> resolvedClass = resolveBeanClass(mbd, beanName);  if (resolvedClass != null && !mbd.hasBeanClass() && mbd.getBeanClassName() != null) {  mbdToUse = new RootBeanDefinition(mbd);  mbdToUse.setBeanClass(resolvedClass);  }  // Prepare method overrides.  try {  /\*  **解析lookup-method和replace-method标签**  \*/  mbdToUse.prepareMethodOverrides();  }  catch (BeanDefinitionValidationException ex) {  throw new BeanDefinitionStoreException(mbdToUse.getResourceDescription(),  beanName, "Validation of method overrides failed", ex);  }  try {  // Give BeanPostProcessors a chance to return a proxy instead of the target bean instance.  /\*  **后置处理器操作**  \*/  Object bean = resolveBeforeInstantiation(beanName, mbdToUse);  if (bean != null) {  return bean;  }  }  catch (Throwable ex) {  throw new BeanCreationException(mbdToUse.getResourceDescription(), beanName,  "BeanPostProcessor before instantiation of bean failed", ex);  }    /\*  **创建bean**  \*/  Object beanInstance = doCreateBean(beanName, mbdToUse, args);  if (logger.isDebugEnabled()) {  logger.debug("Finished creating instance of bean '" + beanName + "'");  }  return beanInstance;  } |

mbdToUse.prepareMethodOverrides()：

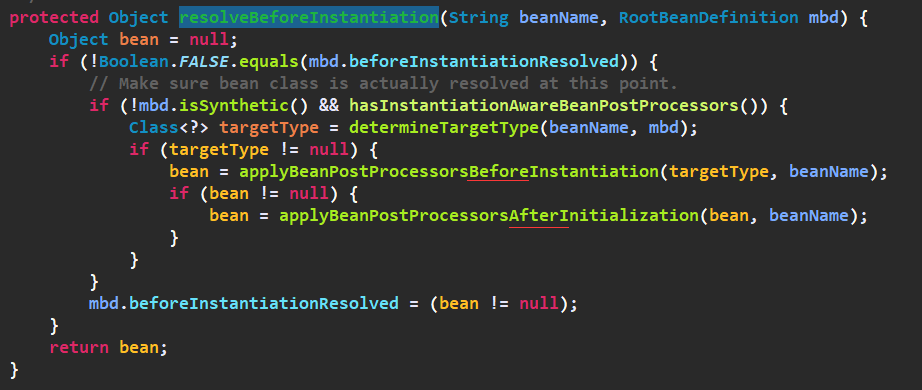
解析bean标签中的lookup-method和replace-method属性





这里先判断重载方法的个数，实际的处理在bean创建的时候处理，大于1默认为true

resolveBeforeInstantiation(beanName, mbdToUse)：主要是后处理器的应用



在创建bean之前做一些操作(后置处理器有关)

doCreateBean(beanName, mbdToUse, args)：

|  |
| --- |
| protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final Object[] args) {  // Instantiate the bean.  BeanWrapper instanceWrapper = null;  if (mbd.isSingleton()) {  instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);  }  if (instanceWrapper == null) {  /\*  **实例化bean，返回一个bean的包装**  \*/  instanceWrapper = createBeanInstance(beanName, mbd, args);  }  final Object bean = (instanceWrapper != null ? instanceWrapper.getWrappedInstance() : null);  Class<?> beanType = (instanceWrapper != null ? instanceWrapper.getWrappedClass() : null);  // Allow post-processors to modify the merged bean definition.  synchronized (mbd.postProcessingLock) {  if (!mbd.postProcessed) {  applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);  mbd.postProcessed = true;  }  }  // Eagerly cache singletons to be able to resolve circular references  // even when triggered by lifecycle interfaces like BeanFactoryAware.  boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&  isSingletonCurrentlyInCreation(beanName));  if (earlySingletonExposure) {  if (logger.isDebugEnabled()) {  logger.debug("Eagerly caching bean '" + beanName +  "' to allow for resolving potential circular references");  }  /\*  **循环依赖**  \*/  addSingletonFactory(beanName, new ObjectFactory<Object>() {  @Override  public Object getObject() throws BeansException {  return getEarlyBeanReference(beanName, mbd, bean);  }  });  }  // Initialize the bean instance.  Object exposedObject = bean;  try {  /\*  **注入bean的属性值(setter)**  \*/  populateBean(beanName, mbd, instanceWrapper);  if (exposedObject != null) {  /\*  **bean标签的init-method属性**  \*/  exposedObject = initializeBean(beanName, exposedObject, mbd);  }  }  catch (Throwable ex) {  if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException) ex).getBeanName())) {  throw (BeanCreationException) ex;  }  else {  throw new BeanCreationException(mbd.getResourceDescription(), beanName, "Initialization of bean failed", ex);  }  }  if (earlySingletonExposure) {  Object earlySingletonReference = getSingleton(beanName, false);  if (earlySingletonReference != null) {  if (exposedObject == bean) {  exposedObject = earlySingletonReference;  }  else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) {  String[] dependentBeans = getDependentBeans(beanName);  Set<String> actualDependentBeans = new LinkedHashSet<String>(dependentBeans.length);  /\*  **检查依赖**  \*/  for (String dependentBean : dependentBeans) {  if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {  actualDependentBeans.add(dependentBean);  }  }  if (!actualDependentBeans.isEmpty()) {  throw new BeanCurrentlyInCreationException(beanName,  "Bean with name '" + beanName + "' has been injected into other beans [" +  StringUtils.collectionToCommaDelimitedString(actualDependentBeans) +  "] in its raw version as part of a circular reference, but has eventually been " +  "wrapped. This means that said other beans do not use the final version of the " +  "bean. This is often the result of over-eager type matching - consider using " +  "'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for example.");  }  }  }  }  // Register bean as disposable.  try {  /\*  **根据scope注册bean(销毁)**  \*/  registerDisposableBeanIfNecessary(beanName, bean, mbd);  }  catch (BeanDefinitionValidationException ex) {  throw new BeanCreationException(mbd.getResourceDescription(), beanName, "Invalid destruction signature", ex);  }  return exposedObject;  } |

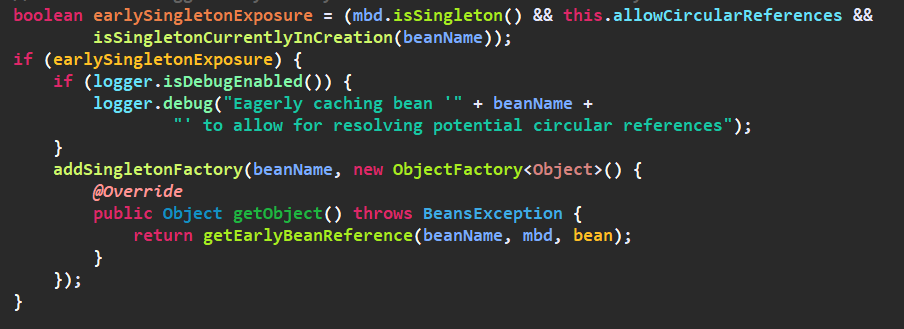
createBeanInstance：创建bean

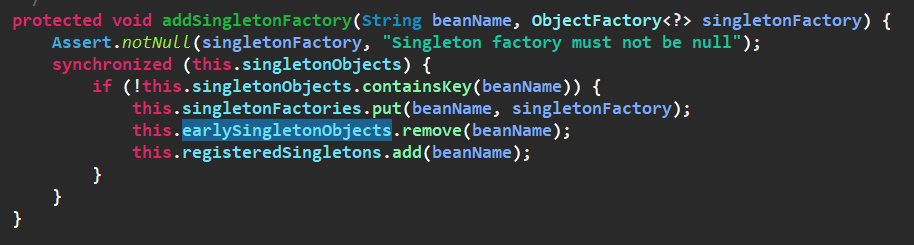
|  |
| --- |
| protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, Object[] args) {  // Make sure bean class is actually resolved at this point.  Class<?> beanClass = resolveBeanClass(mbd, beanName);  if (beanClass != null && !Modifier.isPublic(beanClass.getModifiers()) && !mbd.isNonPublicAccessAllowed()) {  throw new BeanCreationException(mbd.getResourceDescription(), beanName,  "Bean class isn't public, and non-public access not allowed: " + beanClass.getName());  }  /\*  **如果bean标签指定了构造函数，则使用指定的**  \*/  if (mbd.getFactoryMethodName() != null) {  return instantiateUsingFactoryMethod(beanName, mbd, args);  }    /\*  **构造函数可能有很多，在寻找的过程中会非常消耗内存，所以将解析过的构造函数直接**  **放入到beanDefinition中的resolvedConstructorOrFacMethod属性中，如果存在，直接**  **取，否则重新加载**  \*/  // Shortcut when re-creating the same bean...  boolean resolved = false;  boolean autowireNecessary = false;  if (args == null) {  synchronized (mbd.constructorArgumentLock) {  /\*  **判断resolvedConstructorOrFactoryMethod是否为空**  \*/  if (mbd.resolvedConstructorOrFactoryMethod != null) {  resolved = true;  autowireNecessary = mbd.constructorArgumentsResolved;  }  }  }  if (resolved) {  if (autowireNecessary) {  /\*  **构造函数自动注入**  \*/  return autowireConstructor(beanName, mbd, null, null);  }  else {  /\*  **默认构造函数**  \*/  return instantiateBean(beanName, mbd);  }  }  // Need to determine the constructor...  /\*  **根据参数解析构造函数**  \*/  Constructor<?>[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName);  if (ctors != null ||  mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_CONSTRUCTOR ||  mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args)) {  return autowireConstructor(beanName, mbd, ctors, args);  }  // No special handling: simply use no-arg constructor.  return instantiateBean(beanName, mbd);  } |

这里只是调用了bean的构造函数实例化了bean，返回一个BeanWrapper(bean的包装类)

循环依赖：

在对bean注入属性之前，会检测循环bean的依赖关系，循环依赖只会存在singleton中



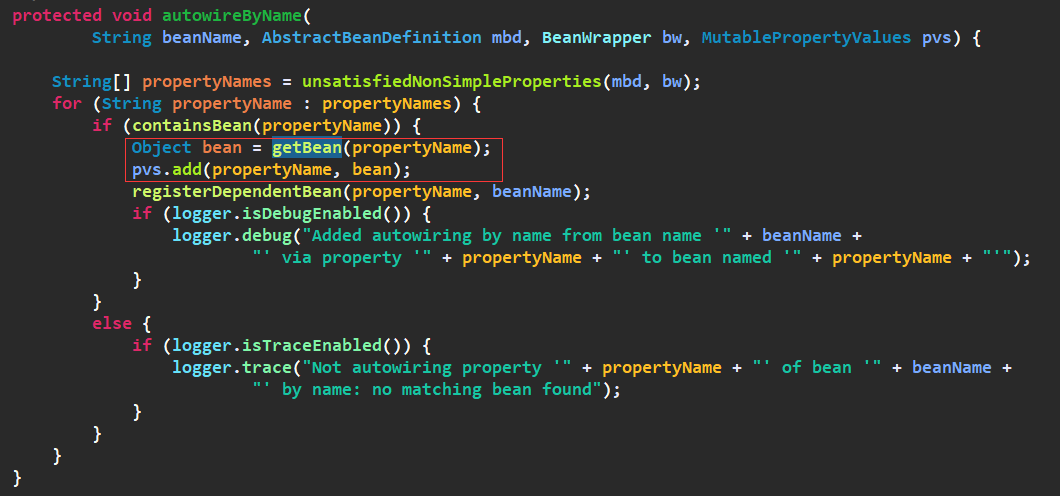


将当前bean的Factory放入到singletonFactories中，循环依赖的时候调用

populateBean(beanName, mbd, instanceWrapper)：

|  |
| --- |
| protected void populateBean(String beanName, RootBeanDefinition mbd, BeanWrapper bw) {  /\*  **将属性封装到PropertyValues中**  \*/  PropertyValues pvs = mbd.getPropertyValues();  if (bw == null) {  if (!pvs.isEmpty()) {  throw new BeanCreationException(  mbd.getResourceDescription(), beanName, "Cannot apply property values to null instance");  }  else {  // Skip property population phase for null instance.  return;  }  }  // Give any InstantiationAwareBeanPostProcessors the opportunity to modify the  // state of the bean before properties are set. This can be used, for example,  // to support styles of field injection.  boolean continueWithPropertyPopulation = true;  /\*  **后置处理器**  \*/  if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {  for (BeanPostProcessor bp : getBeanPostProcessors()) {  if (bp instanceof InstantiationAwareBeanPostProcessor) {  InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;  if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {  continueWithPropertyPopulation = false;  break;  }  }  }  }  if (!continueWithPropertyPopulation) {  return;  }  if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_BY\_NAME ||  mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_BY\_TYPE) {  MutablePropertyValues newPvs = new MutablePropertyValues(pvs);  // Add property values based on autowire by name if applicable.  if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_BY\_NAME) {  /\*  **根据名称自动注入**  \*/  autowireByName(beanName, mbd, bw, newPvs);  }  // Add property values based on autowire by type if applicable.  if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_BY\_TYPE) {  /\*  **根据类型自动注入**  \*/  autowireByType(beanName, mbd, bw, newPvs);  }  pvs = newPvs;  }    /\*  **后处理器**  \*/  boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();  boolean needsDepCheck = (mbd.getDependencyCheck() != RootBeanDefinition.DEPENDENCY\_CHECK\_NONE);  if (hasInstAwareBpps || needsDepCheck) {  PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);  if (hasInstAwareBpps) {  for (BeanPostProcessor bp : getBeanPostProcessors()) {  if (bp instanceof InstantiationAwareBeanPostProcessor) {  InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;  pvs = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), beanName);  if (pvs == null) {  return;  }  }  }  }  if (needsDepCheck) {  checkDependencies(beanName, mbd, filteredPds, pvs);  }  }  /\*  **将属性应用到bean中，因为属性都是封装到pvs中的**  \*/  applyPropertyValues(beanName, mbd, bw, pvs);  } |

autowireByName/ autowireByType主要是注册依赖的bean



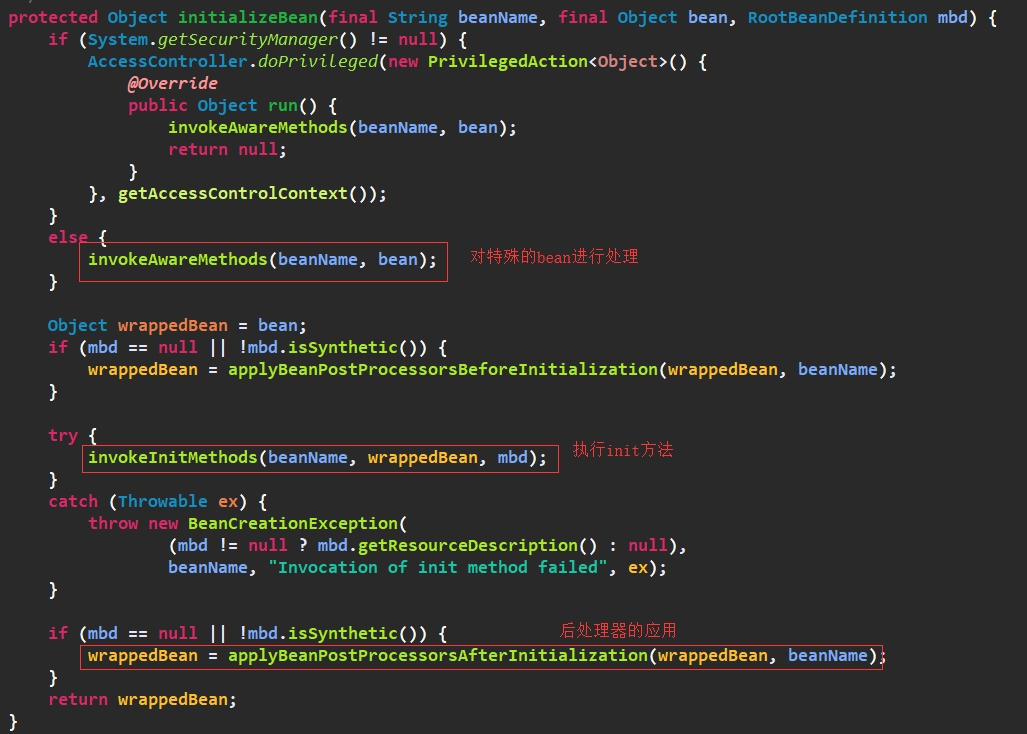
重新调用getBean，注册依赖的bean

applyPropertyValues(beanName, mbd, bw, pvs)：

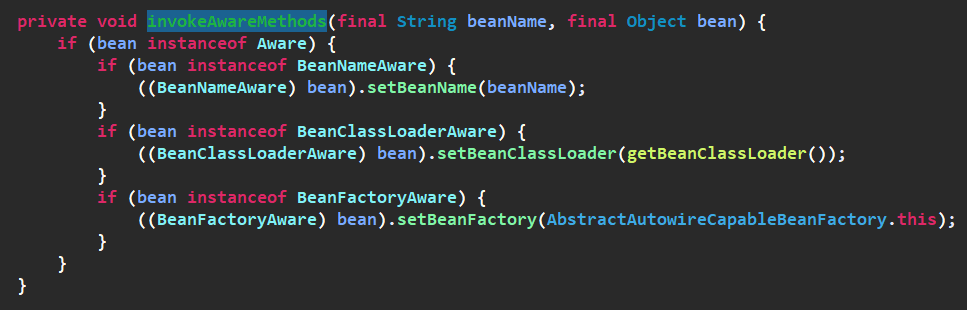
将pvs中的属性封装到bean中

bean标签的init-method属性：

当bean实例化后，会调用init-method指定的方法，进行一些初始化操作



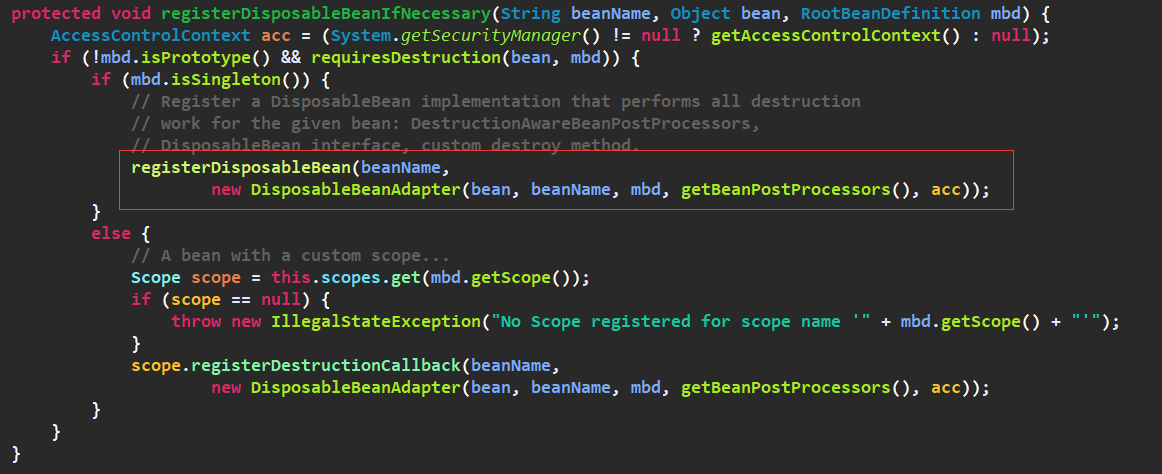
invokeAwareMethods：



当bean实现这些接口后，在被初始化后，会获取到相应的资源，实现BeanFactoryAware接口的bean在初始化后，Spring容器将会注入到BeanFactory中。

applyBeanPostProcessorsAfterInitialization：后处理器相关操作

registerDisposableBeanIfNecessary：注册DisposableBean

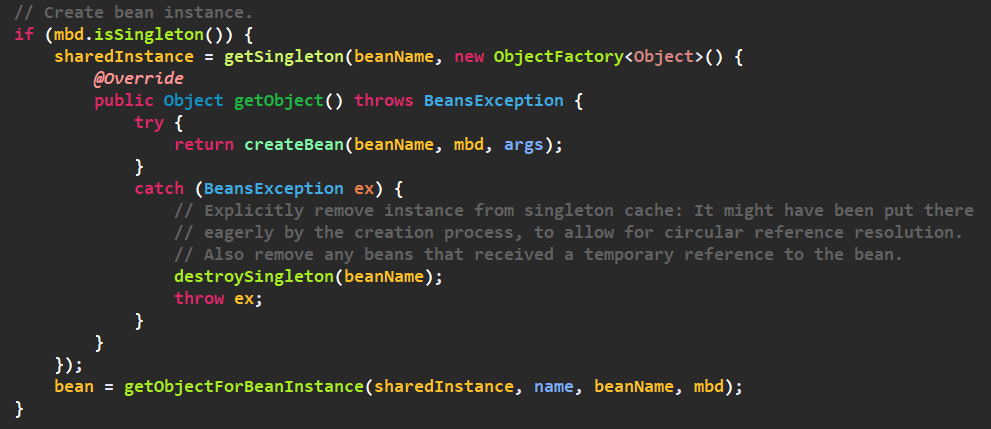


销毁bean除了设置destroy-method方法外，还可以通过注册DestructionAwareBeanPostProcesser后置处理器来同一销毁bean

registerDisposableBean：在单例模式下注册需呀销毁的bean，该方法会处理实现DisposableBean的bean，并且对所有的bean使用DestructionAwareBeanPostProcesser处理。

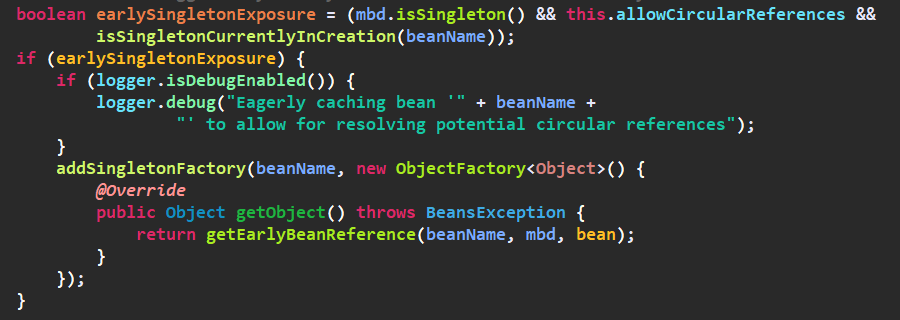
循环依赖的处理：

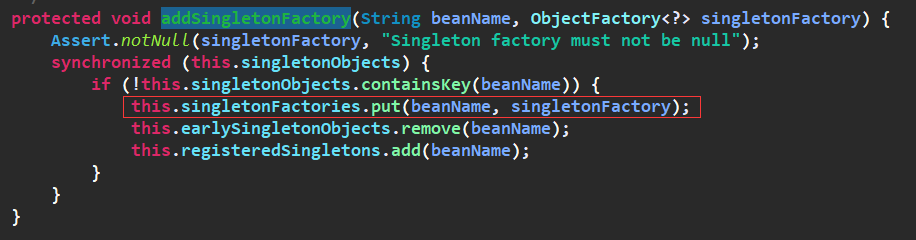
第一次创建一个Bean的时候，直接是执行



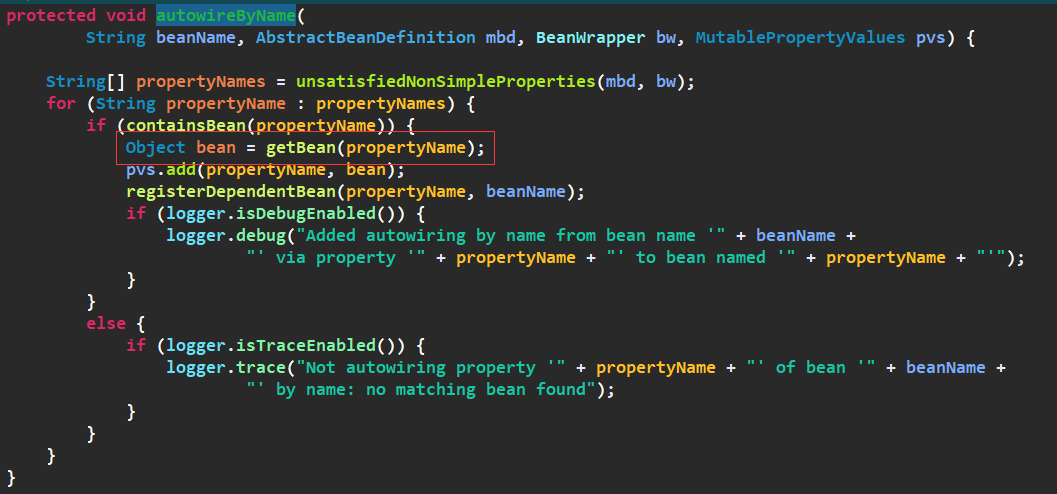


在创建bean之前，有一个beforeSingletonCreation操作，会向集合类型的singletonsCurrentlyInCreation添加当前beanName，标志当前本处于创建状态。然后会去执行bean的构造函数，实例化bean，实例化bean后，会见当前bean的ObjectFactory方法放入到singletonFactories中





然后执行为bean赋值属性的操作(populateBean)，在populateBean方法中通过autowireByName/autowireByType注入属性

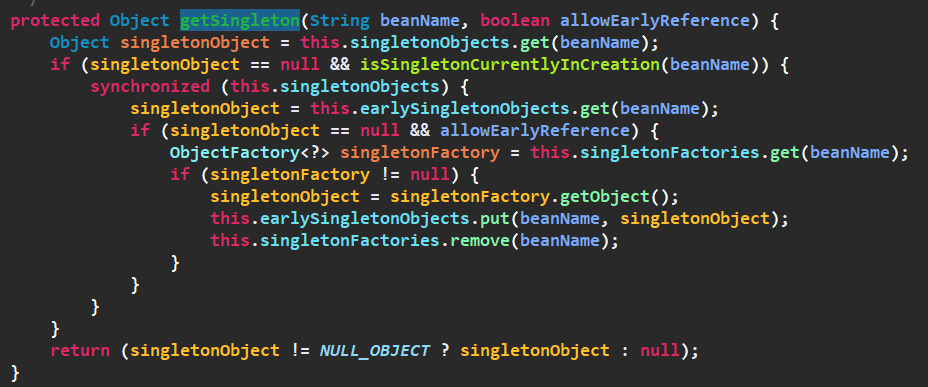


getBean获取的bean

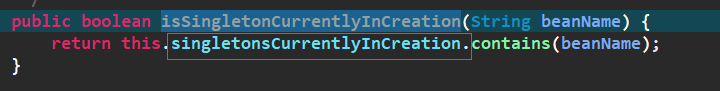
注意，当前bean还没有被创建完，只是实例化了，依赖还没注入完

这里我们假设当前bean自身依赖自己

执行getBean，先从singleton的缓存中获取：

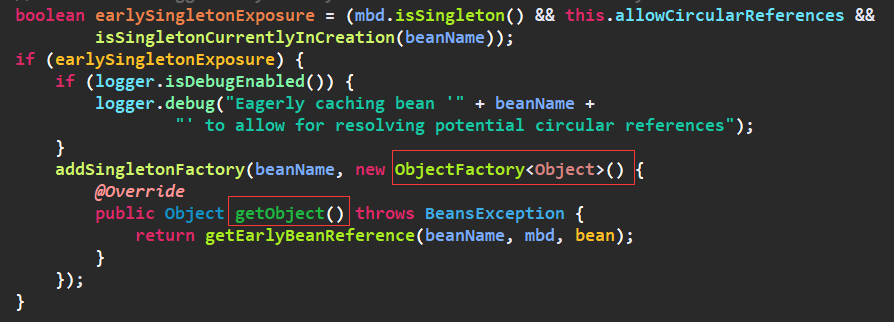


isSingletonCurrentlyInCreation：



isSingletonCurrentlyInCreation判断的是SingletonCurrentlyInCreation中是否有beanName，而在创建bean之前有一个beforeSingletonCreation操作就是向SingletonCurrentlyInCreation中添加beanName

然后就可以在singletonFactories中获取依赖的bean的ObjectFactory，调用getObject方法获取依赖的bean



Spring解决循环依赖的方案：A🡪B-🡪A

先创建A对象只是实例化，依赖关系还没完成，创建B，然后去依赖A，然后A再去依赖B

Spring解决循环依赖只能通过set注入解决，不能通过构造函数依赖，因为通过构造函数依赖，对象创建的同时依赖关系就已经确定了

ApplicationContext