# BeanDefinition

## BeanDefinition

BeanDefinition:描述bean实例，包括了bean的属性，构造函数；可以让BeanFactoryPostProcessor来修改BeanDefinition属性（**Spring容器启动的过程中，会将Bean解析成Spring内部的BeanDefinition结构**。）

## AbstractBeanDefinition

AbstractBeanDefinition：BeanDefinition的抽象基类，AbstractBeanDefinition实现了BeanDefinition定义的一系列操作，定义了描述Bean画像的一系列属性，在AbstractBeanDefinition的基础上，Spring衍生出了一系列具有特殊用途的BeanDefinition。

## GenericBeanDefinition

GenericBeanDefinition：标准bean definition，通用的

除了具有指定类、可选的构造参数值和属性参数这些其它bean definition一样的特性外，它还具有通过parenetName属性来灵活（动态）设置parent bean definition，而非硬编码作为root bean definition

GenericBeanDefinition源码实现非常的的简单，只增加了一个parentName的属性值，其余的实现都在父类AbstractBeanDefinition里

## ChildBeanDefinition

ChildBeanDefinition它可以继承它父类的设置，即ChildBeanDefinition对RootBeanDefinition有一定的依赖关系。（功能和GenericBeanDefinition），所以此处忽略~

## RootBeanDefinition

一个RootBeanDefinition定义表明它是一个可合并的bean definition：即在spring beanFactory运行期间，可以返回一个特定的bean。但在Spring2.5以后，我们绝大多数情况还是可以使用GenericBeanDefinition来做。

我们非常熟悉的final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);这句代码，就是去合并parent的属性进来，这样体现了继承的强大。属性也才完整。

## ScannedGenericBeanDefinition

存储@Component、@Service、@Controller等注解注释的类

它的源码很简单，就是多了一个属性：private final AnnotationMetadata metadata用来存储扫描进来的Bean的一些注解信息。

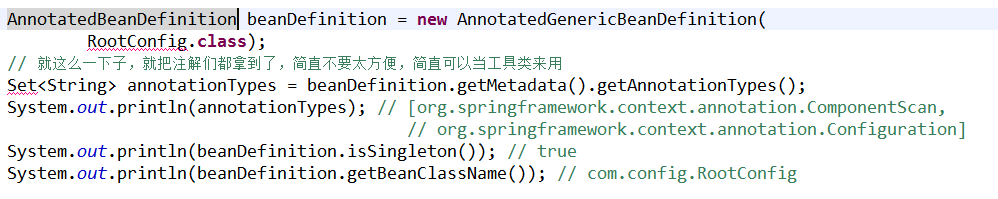
## AnnotatedGenericBeanDefinition

AnnotatedBeanDefinition

AnnotatedGenericBeanDefinition

在基于注解驱动的Spring应用着，它使用得非常的多。因为获取注解信息非常的方便~

AnnotatedGenericBeanDefinition只能用于已经被注册或被扫描到的类（否则你手动new一个，它就不在容器里了，那就脱离管理了）



AnnotatedBeanDefinition beanDefinition = new AnnotatedGenericBeanDefinition(RootConfig.class);

// 就这么一下子，就把注解们都拿到了，简直不要太方便，简直可以当工具类来用

Set<String> annotationTypes = beanDefinition.getMetadata().getAnnotationTypes();

System.out.println(annotationTypes); //[org.springframework.context.annotation.ComponentScan, org.springframework.context.annotation.Configuration]

System.out.println(beanDefinition.isSingleton()); //true

System.out.println(beanDefinition.getBeanClassName()); //com.config.RootConfig

## ConfigurationClassBeanDefinition

首先需要注意的是，它是ConfigurationClassBeanDefinitionReader的一个私有的静态内部类：**这个类负责将@Bean注解的方法转换为对应的ConfigurationClassBeanDefinition类（非常的重要）**

如果@Bean注解没有指定bean的名字，默认会用方法的名字命名bean

@Configuration注解的类会成为一个工厂类，而所有的@Bean注解的方法会成为工厂方法，通过工厂方法实例化Bean，而不是直接通过构造函数初始化（所以我们方法体里面可以很方便的书写逻辑。。。）

# BeanDefinitionBuilder

BeanDefinition实例对象的构造器

# BeanDefinitionReader

该接口的作用就是加载 Bean

在 Spring 中，Bean 一般来说都在配置文件中定义。而在配置的路径由在 web.xml 中定义（还有全注解的方式）。所以加载 Bean 的步骤大致就是：

加载资源，通过配置文件的路径（Location）加载配置文件（Resource）

解析资源，通过解析配置文件的内容得到 Bean。

## AbstractBeanDefinitionReader

它实现了一些基本的方法，但是核心方法loadBeanDefinitions肯定是交给子类实现了

## XmlBeanDefinitionReader

**从xml中加载Bean定义信息**

## ConfigurationClassBeanDefinitionReader

loadBeanDefinitionsFromImportedResources处理，内部借助的就是XmlBeanDefinitionReader去解析它的

# DefaultListableBeanFactory

Spring注册和加载bean的默认实现

默认实现了ListableBeanFactory和BeanDefinitionRegistry接口，基于BeanDefinition对象，是一个成熟的bean Factroy。

Spring容器加载时，最常见的应用是：在访问bean前，需要注册所有的definition（可能从bean definition配置文件中）。使用预先建立的bean定义元数据对象，从本地的BeanDefinition表中查询BeanDefinition因而将不会花费太多成本。

DefaultListableBeanFactory既可以作为一个单独的beanFactory，也可以作为自定义beanFactory的父类。

BeanFactory, BeanDefinitionRegistry

## DeafultListableBeanFactory所实现的接口及接口作用：

AliasRegistry:AliasRegistry是制定别名的管理规则，定义了对alias的简单的简单增改删等操作。

SimpleAlliasRegistry:主要是用map作为alias的缓存，并对接口AliasRegistry进行实现。

SingletonBeanRegistry:定义对单例的注册及获取。

BeanFactory:定义获取bean及bean的各种属性

DefaultSingletonBeanRegistry:对接口SingletonBeanRegistry各函数的实现。

HierarchicalBeanFactory:继承BeanFactory,也就是在BeanFactory的基础上定义了对parentFactory的支持。

BeanDefinitionRegistry:定义对BeanDefinition的各种增改删操作。BeanDefinition中定义的属性有诸如类名、sccope、属性、构造函数参数列表、依赖的bean、是否单例类、是否懒加载等，其实就是将Bean的定义信息存储到这个BeanDefinition相应的属性中，之后对Bean的操作就是直接对BeanDefinition进行的。

FactoryBeanRegistrySupport:在DefaultSingletonBeanRegistry基础上增加了对FactoryBean的特殊处理功能。

ConfigurableBeanFactory:提供配置Factory的各种方法。

ListableBeanFactory:根据各种条件获得bean的配置清单。

AbstractBeanFactory:综合FactoryBeanRegistrySupport和ConfigurableBeanFactory的功能。

AutowireCapableBeanFactory:提供创建bean自动注入，初始化以及应用bean的后置处理器

AbstractAutowireCapableBeanFactory:综合AbstractBeanFactory并对接口Autowire,CapableBeanFactory进行实现。

ConfigurableListableBeanFactory:BeanFactory配置清单，指定忽略类型及接口等。

# 后置处理器

接口BeanDefinitionRegistryPostProcessor：继承BeanFactoryPostProcessor接口，定义了方法postProcessBeanDefinitionRegistry，可以在BeanDefinition初始化之后修改BeanDefinition，也可以注册新的BeanDefinition

BeanFactoryPostProcessor：接口方法postProcessBeanFactory，在bean实例化之前，读取配置元数据，并可以修改它，如果定义了多个BeanFactoryPostProcessor，可以通过设置order来控制执行顺序

典型应用：PropertyPlaceholderConfigurer用来读取properties中的配置信息，然后替换${}表达式

BeanPostProcessor：有postProcessBeforeInitialization和postProcessAfterInitialization方法

在bean实例化之前和之后执行（典型应用AOP）；所有的bean定义已经保存加载到BeanFactory，但是bean的实例还未创建

# Spring容器启动

org.springframework.context.support.AbstractApplicationContext.refresh()方法,spring容器的启动和刷新都是在这个方法中完成的

## prepareRefresh()

initPropertySources():初始化属性设置，默认是为空的，让子类自定义个性化属性设置的方法

getEnvironment().validateRequiredProperties();校验相关属性的合法性

this.earlyApplicationEvents = new LinkedHashSet<ApplicationEvent>();保存容器中的早期时间

## ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory()

创建BeanFactory（DefaultListableBeanFactory）：

GenericApplicationContext的构造函数中创建BeanFactory，创建后返回

## prepareBeanFactory(beanFactory)

准备BeanFactory，设置BeanFactory的一些属性，比如类加载器，后置处理器，设置忽略自动装配的接口

beanFactory.setBeanClassLoader(getClassLoader());

beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));

ApplicationContextAwareProcessor：应用上下文的处理器，主要在postProcessBeforeInitialization方法执行实现Aware接口的对应方法

## postProcessBeanFactory(beanFactory)

空方法，留给子类来实现，在 beanfactory创建并准备完成后做进一步的设置

上面将BeanFactory的初始化工作全部做完了，之后就直接使用

## invokeBeanFactoryPostProcessors(beanFactory)

执行BeanDefinitionRegistryPostProcessor和BeanFactoryPostProcessor2个接口的方法，在执行之前通过getBeanFactoryPostProcessors()拿到容器中所有的BeanFactory后置处理器的实例化对象集合

首先执行BeanDefinitionRegistryPostProcessor

1. 判断BeanFactory后置处理器的类型是否是BeanDefinitionRegistryPostProcessor类型的，如果是就是执行postProcessBeanDefinitionRegistry方法，因为还没有实例化，所以此时还没有执行
2. 执行beanFactory.getBeanNamesForType，拿到BeanDefinitionRegistryPostProcessor的所有beanname
3. 判断是否实现优先级接口PriorityOrdered，实现了就调用getBean方法创建BeanDefinitionRegistryPostProces实例，将对象放到优先集合中（priorityOrderedPostProcessors）
4. 对PriorityOrdered的bean排序
5. 执行postProcessBeanDefinitionRegistry方法
6. 判断是否实现了Ordered接口，实现了就调用getBean方法创建BeanDefinitionRegistryPostProces实例，将对象放到优先集合中（orderedPostProcessors）
7. 对Ordered进行排序
8. 执行postProcessBeanDefinitionRegistry方法
9. 创建没有优先级的BeanDefinitionRegistryPostProcessor对象，执行postProcessBeanDefinitionRegistry方法

执行BeanFactoryPostProcessor

1. 执行beanFactory.getBeanNamesForType拿到所有的beanfactorypostProcessor的beanName
2. 将实现了PriorityOrdered接口的方法优先执行的集合中（priorityOrderedPostProcessors）
3. 排序，然后执行postProcessBeanFactory方法
4. 将实现了Ordered接口，放到orderedPostProcessorNames集合中
5. 排序，然后执行postProcessBeanFactory方法
6. 将没有实现上面2个接口的放到另外一个集合中
7. 执行postProcessBeanFactory方法

## registerBeanPostProcessors(beanFactory)

注册bean的后置处理器

1. 执行beanFactory.getBeanNamesForType(BeanPostProcessor.class, true, false);拿到所有BeanPostProcessor的beanName
2. 创建实现了PriorityOrdered接口BeanPostProcessor的bean实例，然后把实例放到集合（priorityOrderedPostProcessors）中
3. 创建实现了Ordered接口BeanPostProcessor的bean实例，然后把实例放到集合（orderedPostProcessorNames）中
4. 创建普通的BeanPostProcessor的bean实例，然后将实例放到集合（nonOrderedPostProcessorNames）中
5. 排序，然后将BeanPostProcessor实例注册到beanFactorty容器中

## initMessageSource()

给容器初始化消息资源，初始化MessageSource组件（DelegatingMessageSource）

## initApplicationEventMulticaster()

注册事件派发器, SimpleApplicationEventMulticaster

## onRefresh()

空方法子类重写，在容器刷新的时候自定义一些逻辑

## registerListeners()

注册ApplicationContextListener

## finishBeanFactoryInitialization(beanFactory)

实例化非延迟的单例bean

// Instantiate all remaining (non-lazy-init) singletons.

beanFactory.preInstantiateSingletons();

1. 拿到所有的beanName,然后遍历这些beanName拿到对应的BeanDefinition，执行方法getMergedLocalBeanDefinition(beanName)来拿到bean定义
2. 判断BeanDefinition的属性，
3. 如果不是抽象类，单例，非延迟加载进行后续处理
4. 判断是否是FactoryBean，
5. 调用getBean创建实例

## finishRefresh()

# 创建bean

创建bean实例的方法是getBean,getBean中完成创建的就是doGetBean这个方法

org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(String, Class<Object>, Object[], boolean)

1. String beanName = transformedBeanName(name)，判断当前name是否是bean的别名，如果是就从别名容器中拿到bean的实际name
2. Object sharedInstance = getSingleton(beanName);判断容器中是否存在当前beanName对应的实例

1从单例容器中拿出当前beanName对应的实例对象，判断是否存在改实例对象

2如果存在直接返回该对象

3不存在，判断当前对象是否是正在创建（主要是用来处理循环依赖）

4不是之前正在创建的bean并且允许循环依赖，判断是否存在对象工厂，有对象工厂就拿到单例对象，放到earlySingletonObjects中，清除singletonFactories中的对象工厂

1. 如果getSingleton返回的对象不为空，直接返回该对象
2. 如果getSingleton返回的对象为空，
3. 判断是否循环引用的bean，是的话就抛出异常BeanCurrentlyInCreationException
4. 标记当前bean为正在创建的bean，也就是放到alreadyCreated（set集合）中
5. 获取BeanDefinition
6. org.springframework.beans.factory.support.DefaultSingletonBeanRegistry.getSingleton(String, ObjectFactory<?>)

getSingleton方法

1. //从单例容器singletonObjects获取对象，如果存在直接返回，不存在就创建
2. beforeSingletonCreation()判断beanName是否创建已经在之前正在创建的bean,就抛出异常
3. 执行ObjectFactory的getObject方法创建对象，在getObject方法中实际上执行createBean来创建对象
4. createBean(beanName, mbd, args)

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createBean(String, RootBeanDefinition, Object[])

1. 拿到bean的class对象
2. 执行resolveBeforeInstantiation方法：在创建对象之前执行InstantiationAwareBeanPostProcessor的postProcessBeforeInstantiation方法，这里好像都是返回的null，返回不为null的情况暂不清楚
3. 执行doCreateBean方法创建对象，

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean(String, RootBeanDefinition, Object[])

在createBeanInstance方法中创建对象

createBeanInstance—》instantiateBean（通过反射创建对象），对象创建完成后，就将对象包装在BeanWrapperImpl实例中

（处理循环依赖）

单例&&容器支持循环依赖&&不在singletonsCurrentlyInCreation容器中（执行add返回true）

执行addSingletonFactory方法，保存一些bean信息，在getSingleton方法中要用到这些信息

单例容器中不包含当前的bean(首次创建)

将单例工厂保存到单例工厂容器中（单例工厂主要是判断bean是否需要创建代理对象，也就是执行SmartInstantiationAwareBeanPostProcessor的相关方法），清除前期单例容器对象中的bean

将beanName注册到注册单例的容器中

1. 方法populateBean设置bean属性（依赖注入）

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.populateBean(String, RootBeanDefinition, BeanWrapper)方法

在设置属性之前，会执行InstantiationAwareBeanPostProcessor的postProcessAfterInstantiation方法，看到的实现都是直接返回true，没有做什么操作

实现依赖注入（by\_name,by\_type）

设置属性：执行applyPropertyValues

1. 执行initializeBean方法来执行init-method方法
2. invokeAwareMethods（）：执行aware接口对应的方法
3. applyBeanPostProcessorsBeforeInitialization，执行BeanPostProcessor的postProcessBeforeInitialization方法
4. invokeInitMethods：bean实现了InitializingBean，就是执行afterPropertiesSet方法，执行init-method方法
5. applyBeanPostProcessorsAfterInitialization，执行BeanPostProcessor的postProcessAfterInitialization方法（这里可以用来创建当前bean实例的代理对象，AspectJAwareAdvisorAutoProxyCreator）
6. registerDisposableBeanIfNecessary：如果实现了DisposeableBean接口，就将bean方法private final Map<String, Object> disposableBeans = new LinkedHashMap<String, Object>();容器中，在关闭容器的时候执行disable方法
7. bean就创建完成
8. 返回创建好的bean

## 创建代理对象

代理对象的创建是由AspectJAwareAdvisorAutoProxyCreator来完成的，

AspectJAwareAdvisorAutoProxyCreator

继承AbstractAdvisorAutoProxyCreator

继承AbstractAutoProxyCreator，postProcessAfterInitialization的具体实现就在这个父类中

org.springframework.aop.framework.autoproxy.AbstractAutoProxyCreator.postProcessAfterInitialization(Object, String)

1. 判断是否创建过了当前bean的代理对象，创建了就直接返回创建过了的代理对象
2. 没有创建，执行wrapIfNecessary方法
3. getAdvicesAndAdvisorsForBean()，返回bean方法的增强拦截器
4. createProxy创建代理对象：创建ProxyFactory代理工厂，初始化DefaultAopProxyFactory（在ProxyFactory的构造函数中创建），将方法增强器设置到代理工厂中

调用getProxy来创建代理对象：通过DefaultAopProxyFactory的createAopProxy方法来决定使用哪种代理方法：

isOptimize:用来控制通过CGLIB创建的代理是否使用激进的优化策略

isProxyTargetClass:为true时，目标类本身被代理而不是目标类的接口，即使用CGLIB创建代理

hasNoUserSuppliedProxyInterfaces:是否存在代理接口

这几个条件如果都不满足就使用动态代理

这几个条件满足一个，就判断当前类是否实现了接口，如果实现了就用jdk,没有就用cglib

根据对应的方式创建代理对象

代理对象执行业务方法的过程

org.aopalliance.intercept.MethodInterceptor

在调用目标对象的方法时，就可以实现在调用方法之前、调用方法过程中、调用方法之后对其进行控制。

在目标方法上嵌套

CglibMethodInvocation---》ReflectiveMethodInvocation

实现了MethodInvocation接口，用来控制目标方法和增强方法的执行，具体的实现是在ReflectiveMethodInvocation的proceed方法中实现的

获取目标对象，目标方法

method目标方法

targetclass:目标类对象

获取拦截器链（拦截器就是实现MethodInterceptor接口的类）

List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);

创建CglibMethodInvocation对象，然后调用proceed方法来执行目标方法和增强的拦截器链

ExposeInvocationInterceptor --0

ExposeInvocationInterceptor就是用来传递MethodInvocation的。在后续的任何下调用链环节，只要需要用到当前的MethodInvocation就通过

ExposeInvocationInterceptor.currentInvocation()静态方法获得。

ExposeInvocationInterceptor的invoke中执行MethodInvocation的proceed方法，然后从拦截器链中拿到

AspectJAfterThrowingAdvice

AspectJAfterThrowingAdvice --1

当执行的业务方法异常时，调用invokeAdviceMethod方法来执行异常方法

在invoke中执行MethodInvocation的proceed方法，然后从拦截器链中拿到AfterReturningAdviceInterceptor

AfterReturningAdviceInterceptor --2

在invoke中执行MethodInvocation的proceed方法，然后从拦截器链中拿到AspectJAfterAdvice

AspectJAfterAdvice --3

在invoke中执行MethodInvocation的proceed方法，然后从拦截器链中拿到MethodBeforeAdviceInterceptor

MethodBeforeAdviceInterceptor --4

执行before方法，然后执行业务方法

然后回到AspectJAfterAdvice方法，在finally中执行invokeAdviceMethod方法，也就是执行after方法，返回业务方法

的结果

回到AfterReturningAdviceInterceptor，业务方法没有异常，拿到返回的结果，然后执行afterreturn方法，有异常不执行afterreturn

回到AspectJAfterThrowingAdvice，业务方法有异常执行异常afterthrowing方法，没有异常不执行，

然后返回到ExposeInvocationInterceptor，执行完成invoke方法，整个流程也就执行完毕

总结

CglibMethodInvocation—ReflectiveMethodInvocation

创建CglibMethodInvocation对象，然后调用

org.springframework.aop.framework.ReflectiveMethodInvocation.proceed()

在proceed方法中完成执行MethodInterceptor拦截器链，实现功能增强（将通知方法和业务方法按照指定的顺序执行）

XML中解析aop时

1. before

解析before标签，before对应的beandefinition的class就是AspectJMethodBeforeAdvice，创建beandefinition来封装before标签的信息

创建AspectJPointcutAdvisor的beandefinition，class为AspectJPointcutAdvisor，将before的beandefinition封装到这个beandefinition中，将advisor注册到bean factory中

2. after

解析after标签，after对应的beandefinition就是AspectJAfterAdvice，创建beandefinition来封装after标签的信息

创建AspectJPointcutAdvisor的beandefinition，将after的beandefinition封装到这个beandefinition中，将advisor注册到bean factory中

解析after-returning，after-throwing,around

都是如上

3.after-returning对应的beandefinition就是AspectJAfterReturningAdvice

4 after-throwing对应的beandefinition就是AspectJAfterThrowingAdvice

around对应的beandefinition就是AspectJAroundAdvice

解析完后，创建对象时就会创建相应实例的代理对象

切面标签对应的AspectJPointcutAdvisor类注册到spring容器中，id为全量类名加上#数字 ，数字用来区分id 比如：org.springframework.aop.aspectj.AspectJPointcutAdvisor#0

执行的时候是通过MethodInvocation接口来实现的，AspectJMethodBeforeAdvice和AspectJAfterReturningAdvice

没有实现MethodInvocation，就创建适配器AfterReturningAdviceInterceptor和MethodBeforeAdviceInterceptor，这2个类实现了invoke方法

将对应对象封装到这2个拦截器中来实现目标方法的增强

# Spring声明式事务的实现

1. 注解@ EnableTransactionManagement

在EnableTransactionManagement 上使用了@Import，引用了ImportSelector的实现类TransactionManagementConfigurationSelector，

添加了ProxyTransactionManagementConfiguration和AutoProxyRegistrar到容器中

**AutoProxyRegistrar**是一个ImportBeanDefinitionRegistrar,它会检测导入者类上的某个注解是否带有属性mode和proxyTargetClass,

如果检测到这些属性，在mode为PROXY时，它会向容器注册一个自动代理创建器auto proxy creator。

AutoProxyRegistrar给spring容器中注册一个InfrastructureAdvisorAutoProxyCreator，

InfrastructureAdvisorAutoProxyCreator实现了

InstantiationAwareBeanPostProcessor,InstantiationAwareBeanPostProcessor是一个BeanPostProcessor。它可以拦截spring的Bean初始化(Initialization)前后和实例化(Initialization)前后。利用后置处理器机制在被拦截的bean创建以后包装该bean并返回一个代理对象代理对象执行方法利用拦截器链进行调用（同的原理）

用来创建目标对象的代理对象

**ProxyTransactionManagementConfiguration：**

ProxyTransactionManagementConfiguration：是一个spring的配置类,它为spring容器注册了一个BeanFactoryTransactionAttributeSourceAdvisor,是一个事务增强器。（类似before标签的AspectJMethodBeforeAdvice）它有两个重要的字段：AnnotationTransactionAttributeSource和TransactionInterceptor。

TransactionAttributeSource：AnnotationTransactionAttributeSource（注解事务属性解析器），

TransactionInterceptor：事务拦截器，实现了MethodInvocation接口，在他的invoke方法中实现了事务管理

目标方法和事务执行的流程

代理对象执行ReflectiveMethodInvocation的proceed方法，有拦截器TransactionIntercepor

TransactionIntercepor，就执行TransactionIntercepor的invoke方法，

在TransactionIntercepor的invoke方法中根据目标方式的注解属性判断是否需要开启事务，开启事务后，再次执行ReflectiveMethodInvocation的proceed方法，这是已经没有拦截器的就是执行目标方法，执行完成后，回到TransactionIntercepor的invoke方法中关闭或者提交事务

TransactionIntercepor

在invoke方法中获取到目标对象和方法，执行父类TransactionAspectSupport的invokeWithinTransaction方法，在这个方法中实现事务的管理；

首先@Transactional注解的相关信息（TransactionAttributeSource），如果获取到了TransactionAttributeSource，那么当前方法就使用了@Transactional注解，获取注解配置的属性信息，然后拿到事务管理器，开启事务，执行业务方法，提交或者回滚事务

当一个类中使用了@Transactional注解时，这个类就会创建代理对象，以CGLIB为了，执行带有 @Transactional的方法就会拿到拦截器链（只有一个TransactionInterceptor）来实现功能增强，执行没有@Transactional的方法时，获取不到拦截器链，就直接执行目标方法

1. XML配置