课堂主题

Spring源码专题

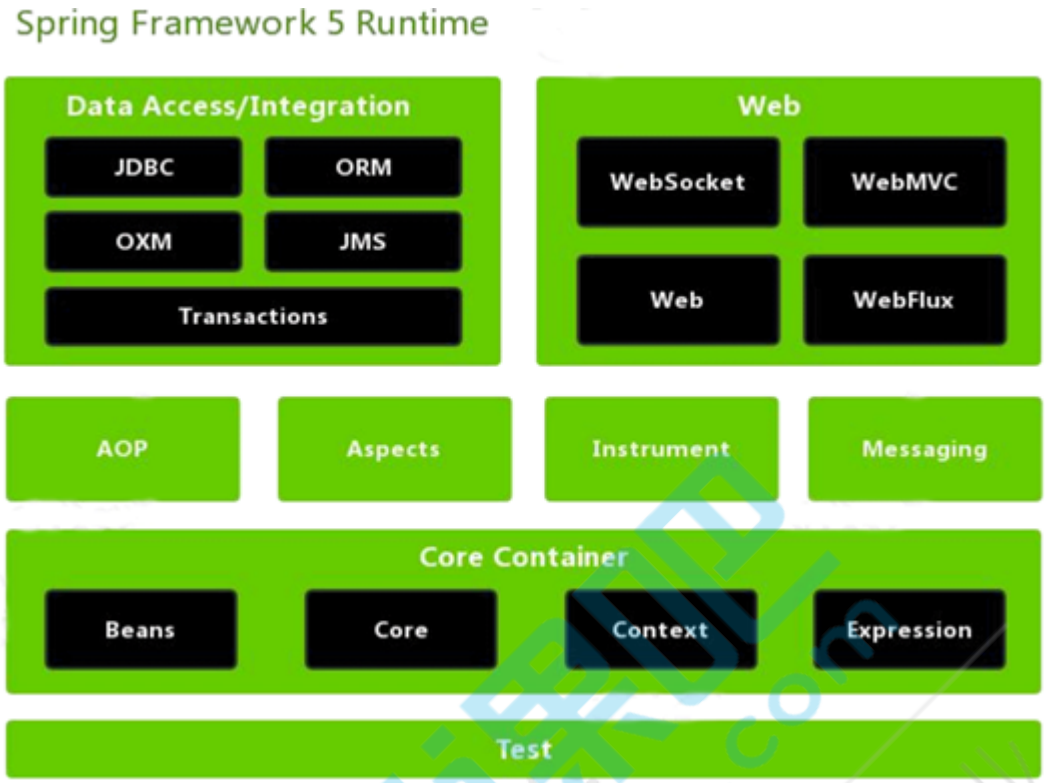
课堂目标

- 搞明白源码中IoC流程
- 搞明白源码中BeanDefinition对象的创建流程
- 搞明白源码中Bean对象的实例化流程
- 搞明白源码中Bean对象的依赖注入流程
- 搞明白源码中AOP是如何应用的
- 搞明白源码中声明式事务是如何实现的

知识要点

Spring体系结构

Spring总共大约有20个模块，由1300多个不同的文件构成。而这些组件被分别整合在核心容器（CoreContainer）、AOP（AspectOrientedProgramming）和设备支持（Instrmentation）、数据访问及集成（DataAccess/Integeration）、Web、报文发送（Messaging）、Test，6个模块集合中。以下是Spring5的模块结构图：



组成Spring框架的每个模块集合或者模块都可以单独存在，也可以一个或多个模块联合实现。每个模块的组成和功能如下：

1.核心容器

由spring-beans、spring-core、spring-context和spring-expression（Spring Expression Language,SpEL）4个模块组成。

spring-beans和spring-core模块是Spring框架的核心模块，包含了控制反转（InversionofControl,IOC）和依赖注入（DependencyInjection,DI）。BeanFactory接口是Spring框架中的核心接口，它是工厂模式的具体实现。BeanFactory使用控制反转对应用程序的配置和依赖性规范与实际的应用程序代码进行了分离。但BeanFactory容器实例化后并不会自动实例化Bean，只有当Bean被使用时BeanFactory容器才会对该Bean进行实例化与依赖关系的装配。

spring-context模块构架于核心模块之上，他扩展了BeanFactory，为她添加了Bean生命周期控制、框架事件体系以及资源加载透明化等功能。此外该模块还提供了许多企业级支持，如邮件访问、远程访问、任务调度等，ApplicationContext是该模块的核心接口，她是BeanFactory的超类，与BeanFactory不同，ApplicationContext容器实例化后会自动对所有的单实例Bean进行实例化与依赖关系的装配，使之处于待用状态。

spring-expression模块是统一表达式语言（EL）的扩展模块，可以查询、管理运行中的对象，同时也方便的可以调用对象方法、操作数组、集合等。它的语法类似于传统EL，但提供了额外的功能，最出色的要数函数调用和简单字符串的模板函数。这种语言的特性是基于Spring产品的需求而设计，他可以非常方便地同SpringIOC进行交互。

2.AOP和设备支持

由spring-aop、spring-aspects和spring-instrument3个模块组成。

spring-aop是Spring的另一个核心模块，是AOP主要的实现模块。作为继OOP后，对程序员影响最大的编程思想之一，AOP极大地开拓了人们对于编程的思路。在Spring中，他是以JVM的动态代理技术为基础，然后设计出了一系列的AOP横切实现，比如前置通知、返回通知、异常通知等，同时，Pointcut接口来匹配切入点，可以使用现有的切入点来设计横切面，也可以扩展相关方法根据需求进行切入。

spring-aspects模块集成自AspectJ框架，主要是为SpringAOP提供多种AOP实现方法。

spring-instrument模块是基于JAVASE中的"java.lang.instrument"进行设计的，应该算是AOP的一个支援模块，主要作用是**在JVM启用时，生成一个代理类，程序员通过代理类在运行时修改类的字节，从而改变一个类的功能，实现AOP的功能。**在分类里，我把他分在了AOP模块下，在Spring官方文档里对这个地方也有点含糊不清，这里是纯个人观点。

3.数据访问及集成

由spring-jdbc、spring-tx、spring-orm、spring-jms和spring-oxm5个模块组成。

spring-jdbc模块是Spring提供的JDBC抽象框架的主要实现模块，用于简化SpringJDBC。主要是提供JDBC模板方式、关系数据库对象化方式、SimpleJdbc方式、事务管理来简化JDBC编程，主要实现类是JdbcTemplate、SimpleJdbcTemplate以及NamedParameterJdbcTemplate。

spring-tx模块是SpringJDBC事务控制实现模块。使用Spring框架，它对事务做了很好的封装，通过它的AOP配置，可以灵活的配置在任何一层；但是在很多的需求和应用，直接使用JDBC事务控制还是有其优势的。其实，事务是以业务逻辑为基础的；一个完整的业务应该对应业务层里的一个方法；如果业务操作失败，则整个事务回滚；所以，事务控制是绝对应该放在业务层的；但是，持久层的设计则应该遵循一个很重要的原则：保证操作的原子性，即持久层里的每个方法都应该是不可以分割的。所以，在使用SpringJDBC事务控制时，应该注意其特殊性。

spring-orm模块是ORM框架支持模块，主要集成Hibernate,JavaPersistenceAPI(JPA)和JavaDataObjects(JDO)用于资源管理、数据访问对象(DAO)的实现和事务策略。

spring-jms模块（JavaMessagingService）能够发送和接受信息，自SpringFramework4.1以后，他还提供了对spring-messaging模块的支撑。

spring-oxm模块主要提供一个抽象层以支撑OXM（OXM是Object-to-XML-Mapping的缩写，它是一个O/M-mapper，将java对象映射成XML数据，或者将XML数据映射成java对象），例如：JAXB,Castor,XMLBeans,JiBX和XStream等。

4.Web

由spring-web、spring-webmvc、spring-websocket和spring-webflux4个模块组成。

spring-web模块为Spring提供了最基础Web支持，主要建立于核心容器之上，通过Servlet或者Listeners来初始化IOC容器，也包含一些与Web相关的支持。

spring-webmvc模块众所周知是一个的Web-Servlet模块，实现了SpringMVC（model-view-Controller）的Web应用。

spring-websocket模块主要是与Web前端的全双工通讯的协议。（资料缺乏，这是个人理解）

spring-webflux是一个新的非堵塞函数式ReactiveWeb框架，可以用来建立异步的，非阻塞，事件驱动的服务，并且扩展性非常好。

5.报文发送

即spring-messaging模块。

spring-messaging是从Spring4开始新加入的一个模块，主要职责是为Spring框架集成一些基础的报文传送应用。

6.Test

即spring-test模块。

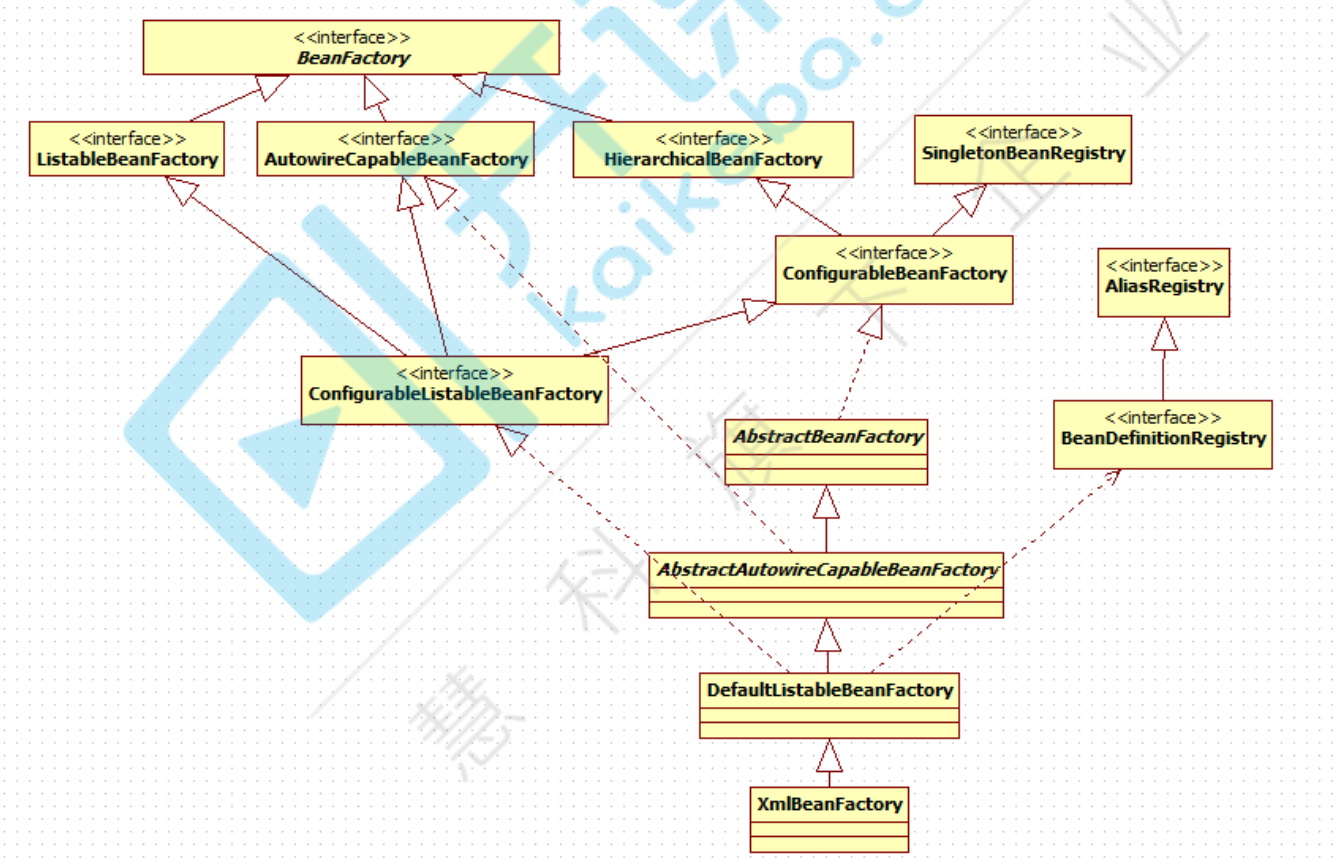
spring-test模块主要为测试提供支持的，毕竟在不需要发布（程序）到你的应用服务器或者连接到其他企业设施的情况下能够执行一些集成测试或者其他测试对于任何企业都是非常重要的。

源码阅读篇

Spring重要接口详解

BeanFactory继承体系

体系结构图



这是BeanFactory基本的类体系结构，这里没有包括强大的ApplicationContext体系，ApplicationContext单独搞一个。

四级接口继承体系：

1. `BeanFactory`作为一个主接口不继承任何接口，暂且称为一级接口。

2. `AutowireCapableBeanFactory`、`HierarchicalBeanFactory`、`ListableBeanFactory`3个子接口继承了它，进行功能上的增强。这3个子接口称为**二级接口。**
3. `ConfigurableBeanFactory`可以被称为**三级接口**，对二级接口`HierarchicalBeanFactory`进行了再次增强，它还继承了另一个外来的接口`SingletonBeanRegistry`
4. `ConfigurableListableBeanFactory`是一个更强大的接口，继承了上述的所有接口，无所不包，称为**四级接口。**

> 总结：
>
> |--`BeanFactory` 是Spring bean容器的根接口.
>
> ```
> 1 提供获取bean,是否包含bean,是否单例与原型,获取bean类型,bean 别名的api.
> ```
>
> |-- -- `AutowireCapableBeanFactory` 提供工厂的装配功能。
>
> |-- -- `HierarchicalBeanFactory` 提供父容器的访问功能
>
> |-- -- -- `ConfigurableBeanFactory` 如名,提供factory的配置功能,眼花缭乱好多api
>
> |-- -- -- -- `ConfigurableListableBeanFactory` 集大成者,提供解析,修改bean定义,并初始化单例.
>
> |-- -- `ListableBeanFactory` 提供容器内bean实例的枚举功能.这边不会考虑父容器内的实例.
>
> 看到这边,我们是不是想起了设计模式原则里的接口隔离原则。

下面是继承关系的2个抽象类和2个实现类：

1. `AbstractBeanFactory`作为一个抽象类，实现了三级接口`ConfigurableBeanFactory`大部分功能。
2. `AbstractAutowireCapableBeanFactory`同样是抽象类，继承自`AbstractBeanFactory`，并额外实现了二级接口`AutowireCapableBeanFactory`。
3. `DefaultListableBeanFactory`继承自`AbstractAutowireCapableBeanFactory`，实现了最强大的四级接口`ConfigurableListableBeanFactory`，并实现了一个外来接口`BeanDefinitionRegistry`，它并非抽象类。
4. 最后是最强大的`XmlBeanFactory`，继承自`DefaultListableBeanFactory`，重写了一些功能，使自己更强大。

那为何要定义这么多层次的接口呢？

查阅这些接口的源码和说明发现，每个接口都有他使用的场合，它主要是为了区分在Spring内部在操作过程中对象的传递和转化过程中，对对象的数据访问所做的限制。例如ListableBeanFactory接口表示这些Bean是可列表的，而HierarchicalBeanFactory表示的是这些Bean是有继承关系的，也就是每个Bean有可能有父Bean。AutowireCapableBeanFactory接口定义Bean的自动装配规则。这四个接口共同定义了Bean的集合、Bean之间的关系、以及Bean行为.

> 总结：
>
> `BeanFactory`的类体系结构看似繁杂混乱，实际上由上而下井井有条，非常容易理解。

BeanFactory

```java
package org.springframework.beans.factory;

public interface BeanFactory {

    //用来引用一个实例，或把它和工厂产生的Bean区分开
    //就是说，如果一个FactoryBean的名字为a，那么，&a会得到那个Factory
    String FACTORY_BEAN_PREFIX = "&";

    /*
     * 四个不同形式的getBean方法，获取实例
     */
    Object getBean(String name) throws BeansException;
    <T> T getBean(String name, Class<T> requiredType) throws BeansException;
    <T> T getBean(Class<T> requiredType) throws BeansException;
    Object getBean(String name, Object... args) throws BeansException;
    // 是否存在
    boolean containsBean(String name);
    // 是否为单实例
    boolean isSingleton(String name) throws NoSuchBeanDefinitionException;
    // 是否为原型（多实例）
    boolean isPrototype(String name) throws NoSuchBeanDefinitionException;
    // 名称、类型是否匹配
    boolean isTypeMatch(String name, Class<?> targetType)
            throws NoSuchBeanDefinitionException;
    // 获取类型
    Class<?> getType(String name) throws NoSuchBeanDefinitionException;
    // 根据实例的名字获取实例的别名
    String[] getAliases(String name);

}
```

在BeanFactory里只对IOC容器的基本行为作了定义，根本不关心你的Bean是如何定义怎样加载的。正如我们只关心工厂里得到什么的产品对象，至于工厂是怎么生产这些对象的，这个基本的接口不关心。

- 源码说明：
  - 4个获取实例的方法。getBean的重载方法。
  - 4个判断的方法。判断是否存在，是否为单例、原型，名称类型是否匹配。
  - 1个获取类型的方法、一个获取别名的方法。根据名称获取类型、根据名称获取别名。一目了然！
- 总结：
  - 这10个方法，很明显，这是一个典型的工厂模式的工厂接口。

ListableBeanFactory

可将Bean逐一列出的工厂

```java
public interface ListableBeanFactory extends BeanFactory {
    // 对于给定的名字是否含有
    boolean containsBeanDefinition(String beanName); BeanDefinition
```

```
    // 返回工厂的BeanDefinition总数
    int getBeanDefinitionCount();
    // 返回工厂中所有Bean的名字
    String[] getBeanDefinitionNames();
    // 返回对于指定类型Bean（包括子类）的所有名字
    String[] getBeanNamesForType(Class<?> type);

    /*
     * 返回指定类型的名字
     *      includeNonSingletons为false表示只取单例Bean，true则不是
     *      allowEagerInit为true表示立刻加载，false表示延迟加载。
     * 注意：FactoryBeans都是立刻加载的。
     */
    String[] getBeanNamesForType(Class<?> type, boolean includeNonSingletons,
            boolean allowEagerInit);
    // 根据类型（包括子类）返回指定Bean名和Bean的Map
    <T> Map<String, T> getBeansOfType(Class<T> type) throws BeansException;
    <T> Map<String, T> getBeansOfType(Class<T> type,
            boolean includeNonSingletons, boolean allowEagerInit)
            throws BeansException;

    // 根据注解类型，查找所有有这个注解的Bean名和Bean的Map
    Map<String, Object> getBeansWithAnnotation(
            Class<? extends Annotation> annotationType) throws BeansException;

    // 根据指定Bean名和注解类型查找指定的Bean
    <A extends Annotation> A findAnnotationOnBean(String beanName,
            Class<A> annotationType);

}
```

- 源码说明：
  - 3个跟BeanDefinition有关的总体操作。包括BeanDefinition的总数、名字的集合、指定类型的名字的集合。

    > 这里指出，BeanDefinition是Spring中非常重要的一个类，每个BeanDefinition实例都包含一个类在Spring工厂中所有属性。

  - 2个getBeanNamesForType重载方法。根据指定类型（包括子类）获取其对应的所有Bean名字。
  - 2个getBeansOfType重载方法。根据类型（包括子类）返回指定Bean名和Bean的Map。
  - 2个跟注解查找有关的方法。根据注解类型，查找Bean名和Bean的Map。以及根据指定Bean名和注解类型查找指定的Bean。
- 总结：

正如这个工厂接口的名字所示，这个工厂接口最大的特点就是可以列出工厂可以生产的所有实例。当然，工厂并没有直接提供返回所有实例的方法，也没这个必要。它可以返回指定类型的所有的实例。而且你可以通过getBeanDefinitionNames()得到工厂所有bean的名字，然后根据这些名字得到所有的Bean。这个工厂接口扩展了BeanFactory的功能，作为上文指出的BeanFactory二级接口，有9个独有的方法，扩展了跟BeanDefinition的功能，提供了BeanDefinition、BeanName、注解有关的各种操作。**它可以根据条件返回Bean的信息集合，这就是它名字的由来——ListableBeanFactory。**

HierarchicalBeanFactory

分层的Bean工厂

```
public interface HierarchicalBeanFactory extends BeanFactory {
    //  返回本Bean工厂的父工厂
    BeanFactory getParentBeanFactory();
    //  本地工厂是否包含这个Bean
    boolean containsLocalBean(String name);
}
```

- 参数说明：
  - 第一个方法返回本Bean工厂的父工厂。这个方法实现了工厂的分层。
  - 第二个方法判断本地工厂是否包含这个Bean（忽略其他所有父工厂）。这也是分层思想的体现。
- 总结：

  这个工厂接口非常简单，实现了Bean工厂的分层。这个工厂接口也是继承自BeanFacotory，也是一个二级接口，相对于父接口，它只扩展了一个重要的功能——工厂分层。

AutowireCapableBeanFactory

自动装配的Bean工厂

```
public interface AutowireCapableBeanFactory extends BeanFactory {
    // 这个常量表明工厂没有自动装配的Bean
    int AUTOWIRE_NO = 0;
    // 表明根据名称自动装配
    int AUTOWIRE_BY_NAME = 1;
    // 表明根据类型自动装配
    int AUTOWIRE_BY_TYPE = 2;
    // 表明根据构造方法快速装配
    int AUTOWIRE_CONSTRUCTOR = 3;
    //表明通过Bean的class的内部来自动装配（有没翻译错...）Spring3.0被弃用。
    @Deprecated
    int AUTOWIRE_AUTODETECT = 4;
    //  根据指定Class创建一个全新的Bean实例
    <T> T createBean(Class<T> beanClass) throws BeansException;
    //  给定对象，根据注释、后处理器等，进行自动装配
    void autowireBean(Object existingBean) throws BeansException;

    // 根据Bean名的BeanDefinition装配这个未加工的Object，执行回调和各种后处理器。
    Object configureBean(Object existingBean, String beanName) throws BeansException;

    // 分解Bean在工厂中定义的这个指定的依赖descriptor
    Object resolveDependency(DependencyDescriptor descriptor, String beanName) throws
BeansException;

    // 根据给定的类型和指定的装配策略，创建一个新的Bean实例
    Object createBean(Class<?> beanClass, int autowireMode, boolean dependencyCheck)
throws BeansException;

```

```
27      // 与上面类似，不过稍有不同。
28      Object autowire(Class<?> beanClass, int autowireMode, boolean dependencyCheck)
    throws BeansException;
29
30      /*
31       * 根据名称或类型自动装配
32       */
33      void autowireBeanProperties(Object existingBean, int autowireMode, boolean
    dependencyCheck)
34              throws BeansException;
35
36      /*
37       * 也是自动装配
38       */
39      void applyBeanPropertyValues(Object existingBean, String beanName) throws
    BeansException;
40
41      /*
42       * 初始化一个Bean...
43       */
44      Object initializeBean(Object existingBean, String beanName) throws
    BeansException;
45
46      /*
47       * 初始化之前执行BeanPostProcessors
48       */
49      Object applyBeanPostProcessorsBeforeInitialization(Object existingBean, String
    beanName)
50              throws BeansException;
51      /*
52       * 初始化之后执行BeanPostProcessors
53       */
54      Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String
    beanName)
55              throws BeansException;
56
57      /*
58       * 分解指定的依赖
59       */
60      Object resolveDependency(DependencyDescriptor descriptor, String beanName,
61              Set<String> autowiredBeanNames, TypeConverter typeConverter) throws
    BeansException;
62
63  }
```

源码说明：

1. 总共5个静态不可变常量来指明装配策略，其中一个常量被Spring3.0废弃、一个常量表示没有自动装配，另外3个常量指明不同的装配策略——根据名称、根据类型、根据构造方法。
2. 8个跟自动装配有关的方法，实在是繁杂，具体的意义我们研究类的时候再分辨吧。
3. 2个执行BeanPostProcessors的方法。
4. 2个分解指定依赖的方法

总结：

```
这个工厂接口继承自BeanFacotory，它扩展了自动装配的功能，根据类定义BeanDefinition装配Bean、执行前、后处理器等。
```

ConfigurableBeanFactory

复杂的配置Bean工厂

```
public interface ConfigurableBeanFactory extends HierarchicalBeanFactory, SingletonBeanRegistry {

    String SCOPE_SINGLETON = "singleton";  //  单例

    String SCOPE_PROTOTYPE = "prototype";  //  原型

    /*
     * 搭配HierarchicalBeanFactory接口的getParentBeanFactory方法
     */
    void setParentBeanFactory(BeanFactory parentBeanFactory) throws IllegalStateException;

    /*
     * 设置、返回工厂的类加载器
     */
    void setBeanClassLoader(ClassLoader beanClassLoader);

    ClassLoader getBeanClassLoader();

    /*
     * 设置、返回一个临时的类加载器
     */
    void setTempClassLoader(ClassLoader tempClassLoader);

    ClassLoader getTempClassLoader();

    /*
     * 设置、是否缓存元数据，如果false，那么每次请求实例，都会从类加载器重新加载（热加载）

     */
    void setCacheBeanMetadata(boolean cacheBeanMetadata);

    boolean isCacheBeanMetadata();//是否缓存元数据

    /*
     * Bean表达式分解器
     */
    void setBeanExpressionResolver(BeanExpressionResolver resolver);

    BeanExpressionResolver getBeanExpressionResolver();

```

```
    /*
     * 设置、返回一个转换服务
     */
    void setConversionService(ConversionService conversionService);

    ConversionService getConversionService();

    /*
     * 设置属性编辑登记员...
     */
    void addPropertyEditorRegistrar(PropertyEditorRegistrar registrar);

    /*
     * 注册常用属性编辑器
     */
    void registerCustomEditor(Class<?> requiredType, Class<? extends PropertyEditor>
propertyEditorClass);

    /*
     * 用工厂中注册的通用的编辑器初始化指定的属性编辑注册器
     */
    void copyRegisteredEditorsTo(PropertyEditorRegistry registry);

    /*
     * 设置、得到一个类型转换器
     */
    void setTypeConverter(TypeConverter typeConverter);

    TypeConverter getTypeConverter();

    /*
     * 增加一个嵌入式的StringValueResolver
     */
    void addEmbeddedValueResolver(StringValueResolver valueResolver);

    String resolveEmbeddedValue(String value);//分解指定的嵌入式的值

    void addBeanPostProcessor(BeanPostProcessor beanPostProcessor);//设置一个Bean后处
理器

    int getBeanPostProcessorCount();//返回Bean后处理器的数量

    void registerScope(String scopeName, Scope scope);//注册范围

    String[] getRegisteredScopeNames();//返回注册的范围名

    Scope getRegisteredScope(String scopeName);//返回指定的范围

    AccessControlContext getAccessControlContext();//返回本工厂的一个安全访问上下文

    void copyConfigurationFrom(ConfigurableBeanFactory otherFactory);//从其他的工厂复制
相关的所有配置

```

```
    /*
     * 给指定的Bean注册别名
     */
    void registerAlias(String beanName, String alias) throws
BeanDefinitionStoreException;

    void resolveAliases(StringValueResolver valueResolver);//根据指定的
StringValueResolver移除所有的别名

    /*
     * 返回指定Bean合并后的Bean定义
     */
    BeanDefinition getMergedBeanDefinition(String beanName) throws
NoSuchBeanDefinitionException;

    boolean isFactoryBean(String name) throws NoSuchBeanDefinitionException;//判断指
定Bean是否为一个工厂Bean

    void setCurrentlyInCreation(String beanName, boolean inCreation);//设置一个Bean是
否正在创建

    boolean isCurrentlyInCreation(String beanName);//返回指定Bean是否已经成功创建

    void registerDependentBean(String beanName, String dependentBeanName);//注册一个
依赖于指定bean的Bean

    String[] getDependentBeans(String beanName);//返回依赖于指定Bean的所欲Bean名

    String[] getDependenciesForBean(String beanName);//返回指定Bean依赖的所有Bean名

    void destroyBean(String beanName, Object beanInstance);//销毁指定的Bean

    void destroyScopedBean(String beanName);//销毁指定的范围Bean

    void destroySingletons();  //销毁所有的单例类

}
```

ConfigurableListableBeanFactory

BeanFactory的集大成者

```
public interface ConfigurableListableBeanFactory
        extends ListableBeanFactory, AutowireCapableBeanFactory,
ConfigurableBeanFactory {

    void ignoreDependencyType(Class<?> type);//忽略自动装配的依赖类型

    void ignoreDependencyInterface(Class<?> ifc);//忽略自动装配的接口

    /*
     * 注册一个可分解的依赖
     */
```

```
    void registerResolvableDependency(Class<?> dependencyType, Object
autowiredValue);

    /*
     * 判断指定的Bean是否有资格作为自动装配的候选者
     */
    boolean isAutowireCandidate(String beanName, DependencyDescriptor descriptor)
 throws NoSuchBeanDefinitionException;

    // 返回注册的Bean定义
    BeanDefinition getBeanDefinition(String beanName) throws
NoSuchBeanDefinitionException;
    // 暂时冻结所有的Bean配置
    void freezeConfiguration();
    // 判断本工厂配置是否被冻结
    boolean isConfigurationFrozen();
    // 使所有的非延迟加载的单例类都实例化。
    void preInstantiateSingletons() throws BeansException;

}
```

- 源码说明：

  1、2个忽略自动装配的的方法。

  2、1个注册一个可分解依赖的方法。

  3、1个判断指定的Bean是否有资格作为自动装配的候选者的方法。

  4、1个根据指定bean名，返回注册的Bean定义的方法。

  5、2个冻结所有的Bean配置相关的方法。

  6、1个使所有的非延迟加载的单例类都实例化的方法。

- 总结：

  工厂接口 `ConfigurableListableBeanFactory` 同时继承了3个接口，`ListableBeanFactory`、`AutowireCapableBeanFactory` 和 `ConfigurableBeanFactory`，扩展之后，加上自有的这8个方法，这个工厂接口总共有83个方法，实在是巨大到不行了。这个工厂接口的自有方法总体上只是对父类接口功能的补充，包含了 `BeanFactory` 体系目前的所有方法，可以说是接口的集大成者。

BeanDefinitionRegistry

额外的接口,这个接口基本用来操作定义在工厂内部的BeanDefinition的。

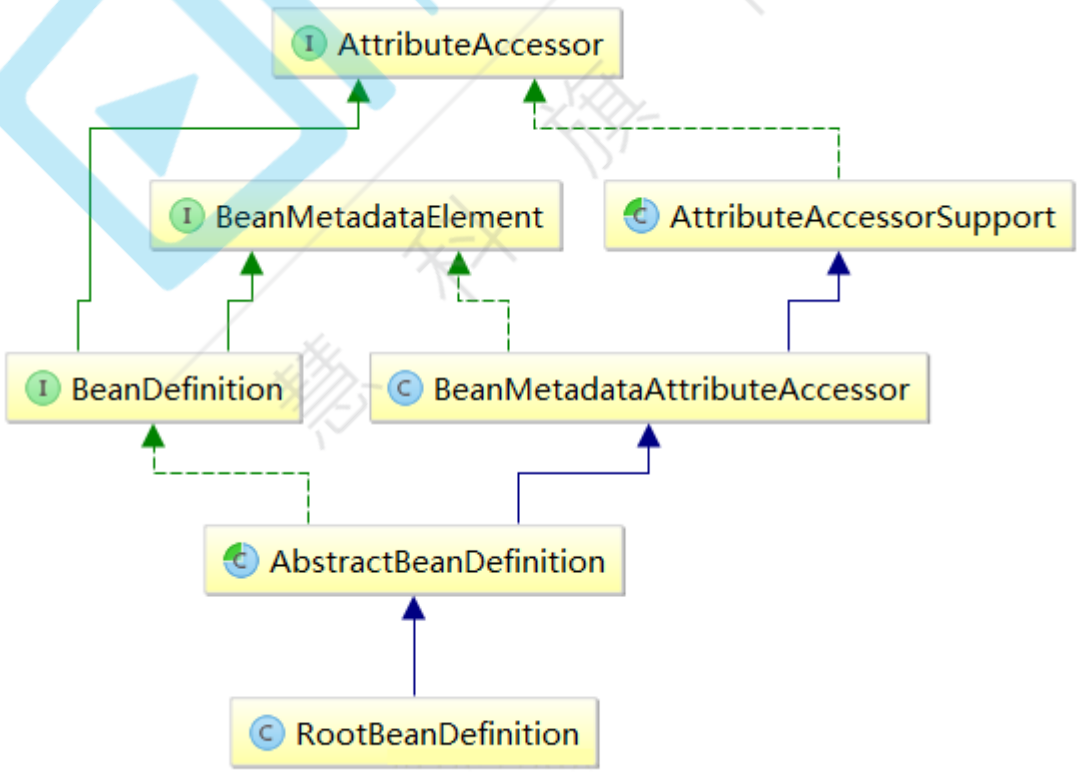
```
public interface BeanDefinitionRegistry extends AliasRegistry {
    // 给定bean名称，注册一个新的bean定义
    void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)
throws BeanDefinitionStoreException;

    /*
     * 根据指定Bean名移除对应的Bean定义
     */
    void removeBeanDefinition(String beanName) throws NoSuchBeanDefinitionException;
```

```
9
10      /*
11       * 根据指定bean名得到对应的Bean定义
12       */
13      BeanDefinition getBeanDefinition(String beanName) throws
    NoSuchBeanDefinitionException;
14
15      /*
16       * 查找，指定的Bean名是否包含Bean定义
17       */
18      boolean containsBeanDefinition(String beanName);
19
20      String[] getBeanDefinitionNames();//返回本容器内所有注册的Bean定义名称
21
22      int getBeanDefinitionCount();//返回本容器内注册的Bean定义数目
23
24      boolean isBeanNameInUse(String beanName);//指定Bean名是否被注册过。
25
26  }
```

BeanDefinition继承体系

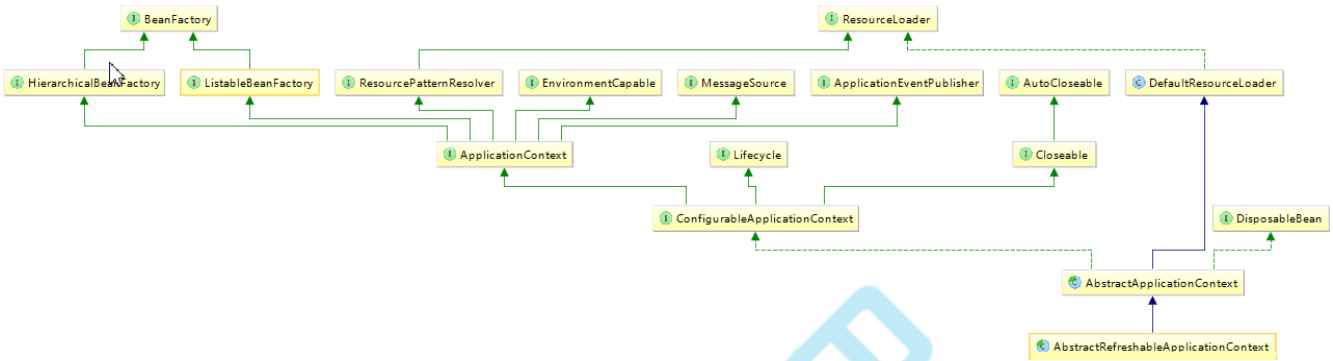
体系结构图

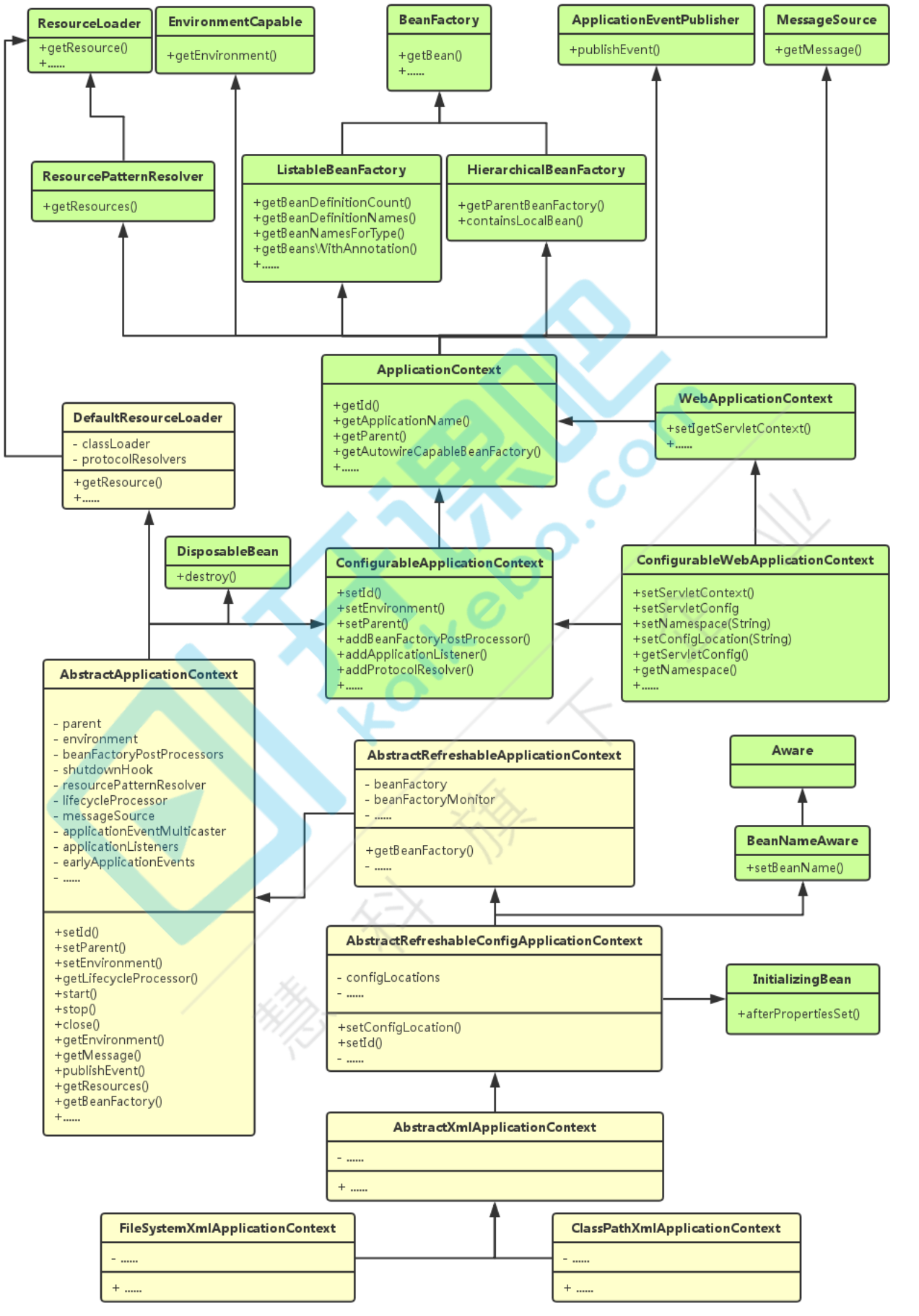
SpringIOC容器管理了我们定义的各种Bean对象及其相互的关系，Bean对象在Spring实现中是以BeanDefinition来描述的，其继承体系如下：

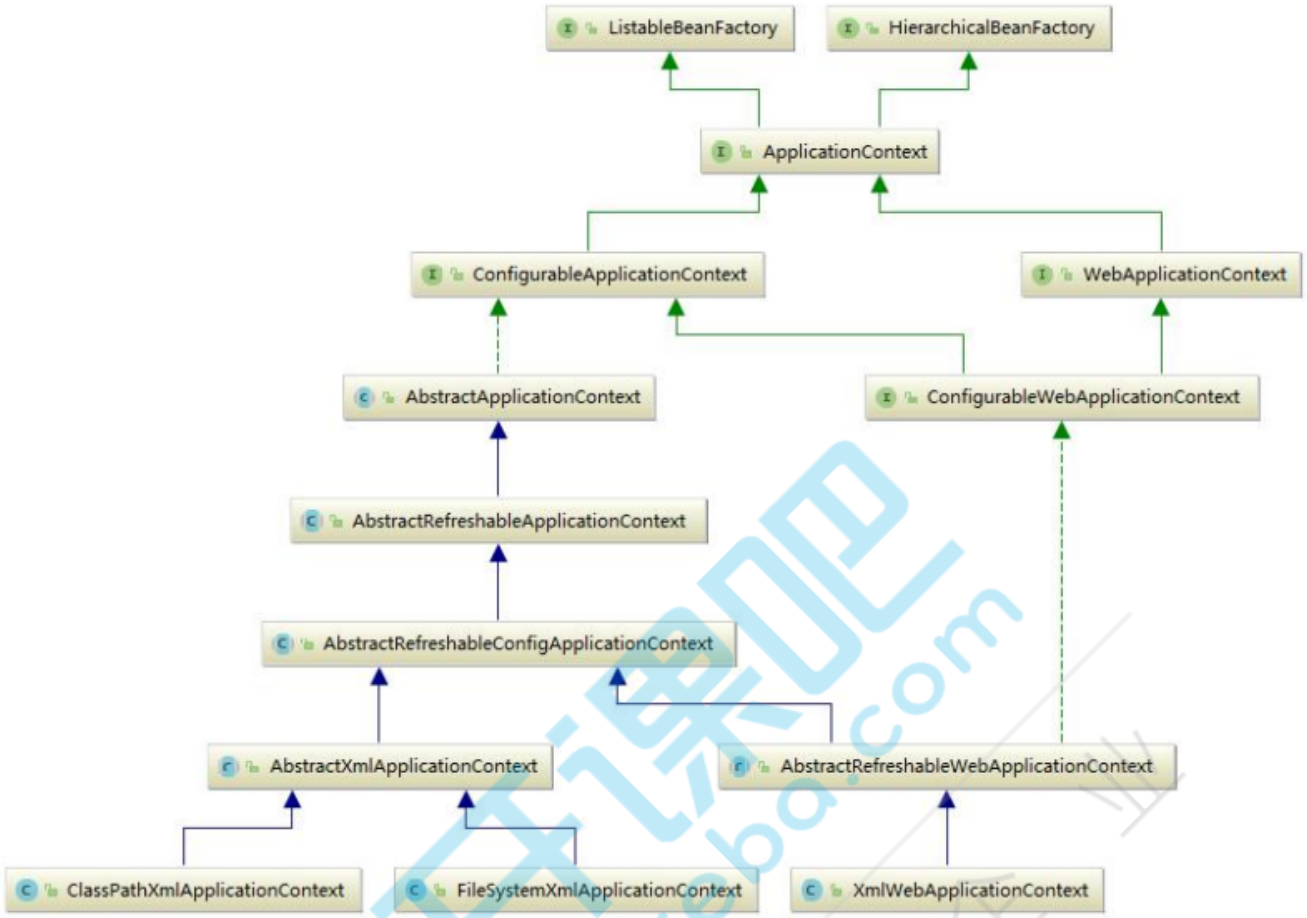


ApplicationContext继承体系

体系结构图




ResourceLoader
+getResource()
+.....
EnvironmentCapable
+getEnvironment()
BeanFactory
+getBean()
+.....
ApplicationEventPublisher
+publishEvent()
MessageSource
+getMessage()
ResourcePatternResolver
+getResources()
ListableBeanFactory
+getBeanDefinitionCount()
+getBeanDefinitionNames()
+getBeanNamesForType()
+getBeans\WithAnnotation()
+.....
HierarchicalBeanFactory
+getParentBeanFactory()
+containsLocalBean()
ApplicationContext
+getId()
+getApplicationName()
+getParent()
+getAutowireCapableBeanFactory()
+.....
WebApplicationContext
+setIgetServletContext()
+.....
DefaultResourceLoader
- classLoader
- protocolResolvers
+getResource()
+.....
DisposableBean
+destroy()
ConfigurableApplicationContext
+setId()
+setEnvironment()
+setParent()
+addBeanFactoryPostProcessor()
+addApplicationListener()
+addProtocolResolver()
+.....
ConfigurableWebApplicationContext
+setServletContext()
+setServletConfig
+setNamespace(String)
+setConfigLocation(String)
+getServletConfig()
+getNamespace()
+.....
AbstractApplicationContext
- parent
- environment
- beanFactoryPostProcessors
- shutdownHook
- resourcePatternResolver
- lifecycleProcessor
- messageSource
- applicationEventMulticaster
- applicationListeners
- earlyApplicationEvents
- .....
+setId()
+setParent()
+setEnvironment()
+getLifecycleProcessor()
+start()
+stop()
+close()
+getEnvironment()
+getMessage()
+publishEvent()
+getResources()
+getBeanFactory()
+.....
AbstractRefreshableApplicationContext
- beanFactory
- beanFactoryMonitor
- .....
+getBeanFactory()
- .....
Aware
BeanNameAware
+setBeanName()
AbstractRefreshableConfigApplicationContext
- configLocations
- .....
+setConfigLocation()
+setId()
- .....
InitializingBean
+afterPropertiesSet()
AbstractXmlApplicationContext
- .....
+ .....
FileSystemXmlApplicationContext
- .....
+ .....
ClassPathXmlApplicationContext
- .....
+ .....
http://cnblogs.com/zffenger



ApplicationContext允许上下文嵌套，通过保持父上下文可以维持一个上下文体系。对于Bean的查找可以在这个上下文体系中发生，首先检查当前上下文，其次是父上下文，逐级向上，这样为不同的Spring应用提供了一个共享的Bean定义环境。

容器初始化流程源码分析

主流程源码分析

找入口

- java程序入口

```
BeanFactory bf = new XMLBeanFactory("spring.xml");

ApplicationContext ctx = new ClassPathXmlApplicationContext("spring.xml");
```

- web程序入口

```xml
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:spring.xml</param-value>
</context-param>
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

> 注意：不管上面哪种方式，最终都会调`AbstractApplicationContext的refresh方法`，而这个方法才是我们真正的入口。

流程解析

- AbstractApplicationContext的`refresh`方法

```java
public void refresh() throws BeansException, IllegalStateException {
        synchronized (this.startupShutdownMonitor) {
            // Prepare this context for refreshing.
            // STEP 1： 刷新预处理
            prepareRefresh();

            // Tell the subclass to refresh the internal bean factory.
            // STEP 2：
            //      a） 创建IoC容器（DefaultListableBeanFactory）
            //      b） 加载解析XML文件（最终存储到Document对象中）
            //      c） 读取Document对象，并完成BeanDefinition的加载和注册工作
            ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

            // Prepare the bean factory for use in this context.
            // STEP 3： 对IoC容器进行一些预处理（设置一些公共属性）
            prepareBeanFactory(beanFactory);

            try {
                // Allows post-processing of the bean factory in context subclasses.
                // STEP 4：
                postProcessBeanFactory(beanFactory);

                // Invoke factory processors registered as beans in the context.
                // STEP 5： 调用BeanFactoryPostProcessor后置处理器对BeanDefinition处理
                invokeBeanFactoryPostProcessors(beanFactory);

                // Register bean processors that intercept bean creation.
                // STEP 6： 注册BeanPostProcessor后置处理器
                registerBeanPostProcessors(beanFactory);

                // Initialize message source for this context.
                // STEP 7： 初始化一些消息源（比如处理国际化的i18n等消息源）
                initMessageSource();

```

```
                // Initialize event multicaster for this context.
                // STEP 8： 初始化应用事件广播器
                initApplicationEventMulticaster();

                // Initialize other special beans in specific context subclasses.
                // STEP 9： 初始化一些特殊的bean
                onRefresh();

                // Check for listener beans and register them.
                // STEP 10： 注册一些监听器
                registerListeners();

                // Instantiate all remaining (non-lazy-init) singletons.
                // STEP 11： 实例化剩余的单例bean（非懒加载方式）
                // 注意事项：Bean的IoC、DI和AOP都是发生在此步骤
                finishBeanFactoryInitialization(beanFactory);

                // Last step: publish corresponding event.
                // STEP 12： 完成刷新时，需要发布对应的事件
                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流程源码分析

找入口

AbstractApplicationContext类的`refresh`方法：

```
// Tell the subclass to refresh the internal bean factory.
// STEP 2：
//      a） 创建IoC容器（DefaultListableBeanFactory）
//      b） 加载解析XML文件（最终存储到Document对象中）
//      c） 读取Document对象，并完成BeanDefinition的加载和注册工作
ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
```

流程解析

- 进入AbstractApplication的`obtainFreshBeanFactory`方法：

  用于创建一个新的`IoC容器`，这个`IoC容器`就是DefaultListableBeanFactory对象。

```
    protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {
        // 主要是通过该方法完成IoC容器的刷新
        refreshBeanFactory();
        ConfigurableListableBeanFactory beanFactory = getBeanFactory();
        if (logger.isDebugEnabled()) {
            logger.debug("Bean factory for " + getDisplayName() + ": " + beanFactory);
        }
        return beanFactory;
    }
```

- 进入AbstractRefreshableApplicationContext的`refreshBeanFactory`方法：
  - 销毁以前的容器
  - 创建新的`IoC容器`
  - 加载`BeanDefinition`对象注册到IoC容器中

```
protected final void refreshBeanFactory() throws BeansException {
        // 如果之前有IoC容器，则销毁
        if (hasBeanFactory()) {
            destroyBeans();
            closeBeanFactory();
        }
        try {
            // 创建IoC容器，也就是DefaultListableBeanFactory
            DefaultListableBeanFactory beanFactory = createBeanFactory();
            beanFactory.setSerializationId(getId());
            customizeBeanFactory(beanFactory);
            // 加载BeanDefinition对象，并注册到IoC容器中（重点）
            loadBeanDefinitions(beanFactory);
            synchronized (this.beanFactoryMonitor) {
                this.beanFactory = beanFactory;
            }
        }
        catch (IOException ex) {
```

```
            throw new ApplicationContextException("I/O error parsing bean definition source for " + getDisplayName(), ex);
        }
    }
```

- 进入AbstractRefreshableApplicationContext的`createBeanFactory`方法

```
    protected DefaultListableBeanFactory createBeanFactory() {
        return new DefaultListableBeanFactory(getInternalParentBeanFactory());
    }
```

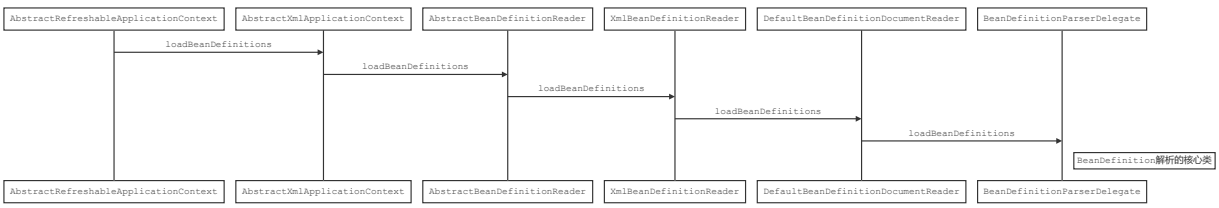
加载BeanDefinition流程分析

找入口

AbstractRefreshableApplicationContext类的`refreshBeanFactory`方法中第13行代码：

```
protected final void refreshBeanFactory() throws BeansException {
        // 如果之前有IoC容器，则销毁
        if (hasBeanFactory()) {
            destroyBeans();
            closeBeanFactory();
        }
        try {
            // 创建IoC容器，也就是DefaultListableBeanFactory
            DefaultListableBeanFactory beanFactory = createBeanFactory();
            beanFactory.setSerializationId(getId());
            customizeBeanFactory(beanFactory);
            // 加载BeanDefinition对象，并注册到IoC容器中（重点）
            loadBeanDefinitions(beanFactory);
            synchronized (this.beanFactoryMonitor) {
                this.beanFactory = beanFactory;
            }
        }
        catch (IOException ex) {
            throw new ApplicationContextException("I/O error parsing bean definition source for " + getDisplayName(), ex);
        }
    }
```

流程图



BeanDefinition流程

```
|--AbstractRefreshableApplicationContext#refreshBeanFactory
    |--AbstractXmlApplicationContext#loadBeanDefinitions：走了多个重载方法
        |--AbstractBeanDefinitionReader#loadBeanDefinitions：走了多个重载方法
            |--XmlBeanDefinitionReader#loadBeanDefinitions：走了多个重载方法
                |--XmlBeanDefinitionReader#doLoadBeanDefinitions
                    |--XmlBeanDefinitionReader#registerBeanDefinitions
                        |--
DefaultBeanDefinitionDocumentReader#registerBeanDefinitions
                            |--#doRegisterBeanDefinitions
                                |--#parseBeanDefinitions
                                    |--#parseDefaultElement
                                        |--#processBeanDefinition
                                            |--
BeanDefinitionParserDelegate#parseBeanDefinitionElement
                                                |--#parseBeanDefinitionElement

```

流程相关类的说明

- AbstractRefreshableApplicationContext

  主要用来对BeanFactory提供 `refresh` 功能。包括BeanFactory的创建和 `BeanDefinition` 的定义、解析、注册操作。

- AbstractXmlApplicationContext

  主要提供对于 `XML资源` 的加载功能。包括从Resource资源对象和资源路径中加载XML文件。

- AbstractBeanDefinitionReader

  主要提供对于 `BeanDefinition` 对象的读取功能。具体读取工作交给子类实现。

- XmlBeanDefinitionReader

  主要通过 `DOM4J` 对于 `XML资源` 的读取、解析功能，并提供对于 `BeanDefinition` 的注册功能。

- DefaultBeanDefinitionDocumentReader
- BeanDefinitionParserDelegate

流程解析

- 进入AbstractXmlApplicationContext的loadBeanDefinitions方法：
  - 创建一个XmlBeanDefinitionReader，通过阅读XML文件，真正完成BeanDefinition的加载和注册。
  - 配置XmlBeanDefinitionReader并进行初始化。
  - 委托给XmlBeanDefinitionReader去加载BeanDefinition。

```
    protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory)
throws BeansException, IOException {
        // Create a new XmlBeanDefinitionReader for the given BeanFactory.
        // 给指定的工厂创建一个BeanDefinition阅读器
        // 作用：通过阅读XML文件，真正完成BeanDefinition的加载和注册

```

```
        XmlBeanDefinitionReader beanDefinitionReader = new
XmlBeanDefinitionReader(beanFactory);

        // Configure the bean definition reader with this context's
        // resource loading environment.
        beanDefinitionReader.setEnvironment(this.getEnvironment());
        beanDefinitionReader.setResourceLoader(this);
        beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));

        // Allow a subclass to provide custom initialization of the reader,
        // then proceed with actually loading the bean definitions.
        initBeanDefinitionReader(beanDefinitionReader);

        // 委托给BeanDefinition阅读器去加载BeanDefinition
        loadBeanDefinitions(beanDefinitionReader);
    }

    protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws
                        BeansException, IOException {
        // 获取资源的定位
        // 这里getConfigResources是一个空实现，真正实现是调用子类的获取资源定位的方法
        // 比如：ClassPathXmlApplicationContext中进行了实现
        //      而FileSystemXmlApplicationContext没有使用该方法
        Resource[] configResources = getConfigResources();
        if (configResources != null) {
            // XML Bean读取器调用其父类AbstractBeanDefinitionReader读取定位的资源
            reader.loadBeanDefinitions(configResources);
        }
        // 如果子类中获取的资源定位为空，则获取FileSystemXmlApplicationContext构造方法中
setConfigLocations方法设置的资源
        String[] configLocations = getConfigLocations();
        if (configLocations != null) {
            // XML Bean读取器调用其父类AbstractBeanDefinitionReader读取定位的资源
            reader.loadBeanDefinitions(configLocations);
        }
    }
```

- `loadBeanDefinitions`方法经过一路的兜兜转转，最终来到了`XmlBeanDefinitionReader`的`doLoadBeanDefinitions`方法：
  - 一个是对`XML`文件进行`DOM`解析；
  - 一个是完成`BeanDefinition`对象的加载与注册。

```
protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)
            throws BeanDefinitionStoreException {
        try {
            // 通过DOM4J加载解析XML文件，最终形成Document对象
            Document doc = doLoadDocument(inputSource, resource);
            // 通过对Document对象的操作，完成BeanDefinition的加载和注册工作
            return registerBeanDefinitions(doc, resource);
        }
        //省略一些catch语句
        catch (Throwable ex) {
            ......
        }
    }
```

- 此处我们暂不处理DOM4J加载解析XML的流程，我们重点分析BeanDefinition的加载注册流程
- 进入XmlBeanDefinitionReader的 `registerBeanDefinitions` 方法：
  - 创建DefaultBeanDefinitionDocumentReader用来解析Document对象。
  - 获得容器中已注册的BeanDefinition数量
  - 委托给DefaultBeanDefinitionDocumentReader来完成BeanDefinition的加载、注册工作。
  - 统计新注册的BeanDefinition数量

```
    public int registerBeanDefinitions(Document doc, Resource resource) throws
                                        BeanDefinitionStoreException {
        // 创建DefaultBeanDefinitionDocumentReader用来解析Document对象
        BeanDefinitionDocumentReader documentReader =
                                        createBeanDefinitionDocumentReader();
        // 获得容器中注册的Bean数量
        int countBefore = getRegistry().getBeanDefinitionCount();
        //解析过程入口，BeanDefinitionDocumentReader只是个接口
        //具体的实现过程在DefaultBeanDefinitionDocumentReader完成
        documentReader.registerBeanDefinitions(doc, createReaderContext(resource));
        // 统计注册的Bean数量
        return getRegistry().getBeanDefinitionCount() - countBefore;
    }
```

- 进入DefaultBeanDefinitionDocumentReader的 `registerBeanDefinitions` 方法：
  - 获得Document的根元素标签
  - 真正实现BeanDefinition解析和注册工作

```
    public void registerBeanDefinitions(Document doc, XmlReaderContext readerContext
    {
        this.readerContext = readerContext;
        logger.debug("Loading bean definitions");
        // 获得Document的根元素<beans>标签
        Element root = doc.getDocumentElement();
        // 真正实现BeanDefinition解析和注册工作
        doRegisterBeanDefinitions(root);
    }
```

- 进入DefaultBeanDefinitionDocumentReader `doRegisterBeanDefinitions` 方法：
  - 这里使用了委托模式，将具体的BeanDefinition解析工作交给了BeanDefinitionParserDelegate去完成
  - 在解析Bean定义之前，进行自定义的解析，增强解析过程的可扩展性
  - 委托给BeanDefinitionParserDelegate,从Document的根元素开始进行BeanDefinition的解析
  - 在解析Bean定义之后，进行自定义的解析，增加解析过程的可扩展性

```
    protected void doRegisterBeanDefinitions(Element root) {
        // Any nested <beans> elements will cause recursion in this method. In
        // order to propagate and preserve <beans> default-* attributes correctly,
        // keep track of the current (parent) delegate, which may be null. Create
        // the new (child) delegate with a reference to the parent for fallback
purposes,
        // then ultimately reset this.delegate back to its original (parent)
reference.
        // this behavior emulates a stack of delegates without actually necessitating
one.

        // 这里使用了委托模式，将具体的BeanDefinition解析工作交给了
BeanDefinitionParserDelegate去完成
        BeanDefinitionParserDelegate parent = this.delegate;
        this.delegate = createDelegate(getReaderContext(), root, parent);

        if (this.delegate.isDefaultNamespace(root)) {
            String profileSpec = root.getAttribute(PROFILE_ATTRIBUTE);
            if (StringUtils.hasText(profileSpec)) {
                String[] specifiedProfiles = StringUtils.tokenizeToStringArray(
                        profileSpec,
BeanDefinitionParserDelegate.MULTI_VALUE_ATTRIBUTE_DELIMITERS);
                if
(!getReaderContext().getEnvironment().acceptsProfiles(specifiedProfiles)) {
                    if (logger.isInfoEnabled()) {
                        logger.info("Skipped XML bean definition file due to
specified profiles [" + profileSpec +
                                "] not matching: " +
getReaderContext().getResource());
                    }
                    return;
                }
            }
        }
        // 在解析Bean定义之前，进行自定义的解析，增强解析过程的可扩展性
        preProcessXml(root);
        // 委托给BeanDefinitionParserDelegate,从Document的根元素开始进行BeanDefinition的解
析
        parseBeanDefinitions(root, this.delegate);
        // 在解析Bean定义之后，进行自定义的解析，增加解析过程的可扩展性
        postProcessXml(root);

        this.delegate = parent;
```

```
    }
```

Bean实例化流程分析

找入口

AbstractApplicationContext类的`refresh`方法：

```
    // Instantiate all remaining (non-lazy-init) singletons.
    // STEP 11： 实例化剩余的单例bean（非懒加载方式）
    // 注意事项：Bean的IoC、DI和AOP都是发生在此步骤
    finishBeanFactoryInitialization(beanFactory);
```

流程解析

AOP流程源码分析

查找BeanDefinitionParser流程分析

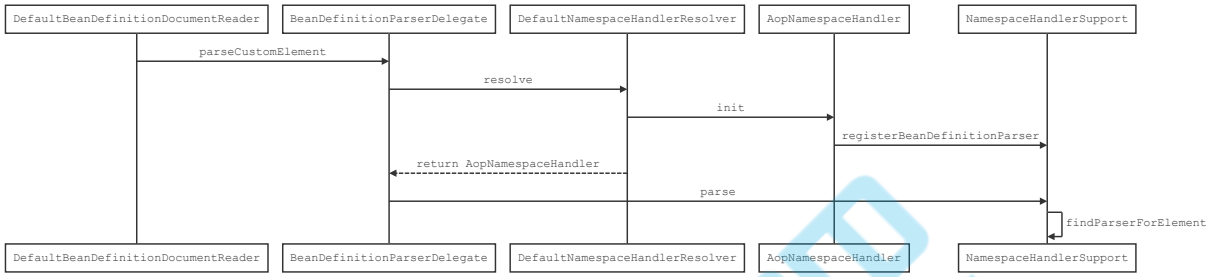
找入口

`DefaultBeanDefinitionDocumentReader#parseBeanDefinitions`方法的*第16行或者23行*：

```
    protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {
        // 加载的Document对象是否使用了Spring默认的XML命名空间（beans命名空间）
        if (delegate.isDefaultNamespace(root)) {
            // 获取Document对象根元素的所有子节点（bean标签、import标签、alias标签和其他自定义标签context、aop等）
            NodeList nl = root.getChildNodes();
            for (int i = 0; i < nl.getLength(); i++) {
                Node node = nl.item(i);
                if (node instanceof Element) {
                    Element ele = (Element) node;
                    // bean标签、import标签、alias标签，则使用默认解析规则
                    if (delegate.isDefaultNamespace(ele)) {
                        parseDefaultElement(ele, delegate);
                    }
                    //像context标签、aop标签、tx标签，则使用用户自定义的解析规则解析元素节点
                    else {
                        delegate.parseCustomElement(ele);
                    }
                }
            }
        }
        else {
```

```
22              // 如果不是默认的命名空间，则使用用户自定义的解析规则解析元素节点
23              delegate.parseCustomElement(root);
24          }
25      }
```

流程图



流程相关类的说明

流程解析

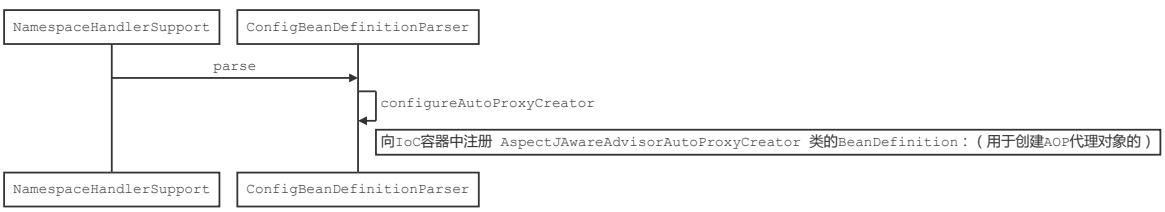
执行BeanDefinitionParser流程分析

找入口

NamespaceHandlerSupport类的`parse`方法*第6行代码*：

```
1       public BeanDefinition parse(Element element, ParserContext parserContext) {
2           // NamespaceHandler里面初始化了大量的BeanDefinitionParser来分别处理不同的自定义标签
3           // 从指定的NamespaceHandler中，匹配到指定的BeanDefinitionParser
4           BeanDefinitionParser parser = findParserForElement(element, parserContext);
5           // 调用指定自定义标签的解析器，完成具体解析工作
6           return (parser != null ? parser.parse(element, parserContext) : null);
7       }
```

流程图



流程相关类的说明

流程解析

产生AOP代理流程分析

AspectJAwareAdvisorAutoProxyCreator的继承体系

```
|-BeanPostProcessor
    postProcessBeforeInitialization---初始化之前调用
    postProcessAfterInitialization---初始化之后调用

|--InstantiationAwareBeanPostProcessor
    postProcessBeforeInstantiation---实例化之前调用
    postProcessAfterInstantiation---实例化之后调用
    postProcessPropertyValues---后置处理属性值

|---SmartInstantiationAwareBeanPostProcessor
    predictBeanType
    determineCandidateConstructors
    getEarlyBeanReference

|----AbstractAutoProxyCreator
    postProcessBeforeInitialization
    postProcessAfterInitialization----AOP功能入口
    postProcessBeforeInstantiation
    postProcessAfterInstantiation
    postProcessPropertyValues
    ...
|-----AbstractAdvisorAutoProxyCreator
    getAdvicesAndAdvisorsForBean
    findEligibleAdvisors
    findCandidateAdvisors
    findAdvisorsThatCanApply

|------AspectJAwareAdvisorAutoProxyCreator
    extendAdvisors
    sortAdvisors
```

找入口

AbstractAutoProxyCreator类的 `postProcessAfterInitialization` 方法*第6行代码*：

```
    public Object postProcessAfterInitialization(@Nullable Object bean, String beanName) throws BeansException {
        if (bean != null) {
            Object cacheKey = getCacheKey(bean.getClass(), beanName);
            if (!this.earlyProxyReferences.contains(cacheKey)) {
                // 使用动态代理技术，产生代理对象
                return wrapIfNecessary(bean, beanName, cacheKey);
            }
        }
        return bean;
    }
```

流程图

事务流程源码分析

获取TransactionInterceptor的BeanDefinition

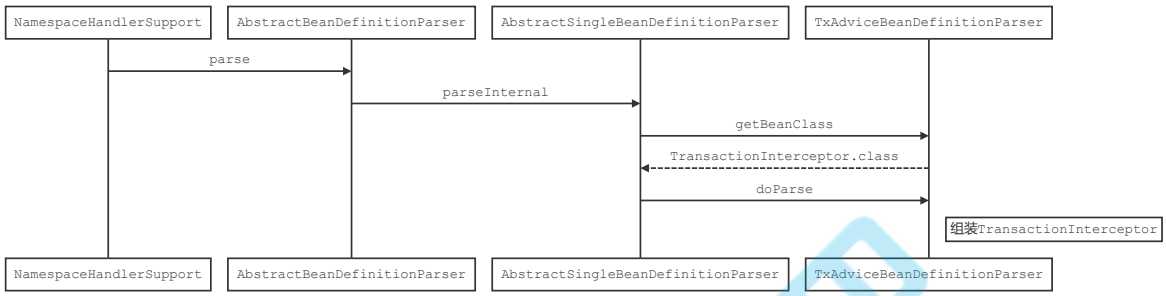
找入口

`AbstractBeanDefinitionParser#parse`方法：

```
public final BeanDefinition parse(Element element, ParserContext parserContext) {
        // 调用子类的parseInternal获取BeanDefinition对象
        AbstractBeanDefinition definition = parseInternal(element, parserContext);

        if (definition != null && !parserContext.isNested()) {
            try {
                String id = resolveId(element, definition, parserContext);
                if (!StringUtils.hasText(id)) {
                    parserContext.getReaderContext().error(
                            "Id is required for element '" +
parserContext.getDelegate().getLocalName(element)
                                    + "' when used as a top-level tag", element);
                }
                String[] aliases = null;
                if (shouldParseNameAsAliases()) {
                    String name = element.getAttribute(NAME_ATTRIBUTE);
                    if (StringUtils.hasLength(name)) {
                        aliases =
StringUtils.trimArrayElements(StringUtils.commaDelimitedListToStringArray(name));
                    }
                }
                BeanDefinitionHolder holder = new BeanDefinitionHolder(definition,
id, aliases);
                // 将处理<tx:advice>标签的类BeanDefinition对象，注册到IoC容器中
                registerBeanDefinition(holder, parserContext.getRegistry());
                if (shouldFireEvents()) {
                    BeanComponentDefinition componentDefinition = new
BeanComponentDefinition(holder);
                    postProcessComponentDefinition(componentDefinition);
                    parserContext.registerComponent(componentDefinition);
                }
            }
            catch (BeanDefinitionStoreException ex) {
                String msg = ex.getMessage();
                parserContext.getReaderContext().error((msg != null ? msg :
ex.toString()), element);
                return null;
            }
        }
        return definition;
```

```
36      }
```

流程图



流程解析

执行TransactionInterceptor流程分析

找入口

TransactionInterceptor类实现了MethodInterceptor接口，所以入口方法是 invoke 方法：

```
1  public Object invoke(final MethodInvocation invocation) throws Throwable {
2
3          Class<?> targetClass = (invocation.getThis() != null ?
   AopUtils.getTargetClass(invocation.getThis()) : null);
4
5          // 调用TransactionAspectSupport类的invokeWithinTransaction方法去实现事务支持
6          return invokeWithinTransaction(invocation.getMethod(), targetClass,
   invocation::proceed);
7      }
```

流程图

手写框架篇

框架分析

- 工程名称：spring-custom
- 工程目的：手写spring框架，实现spring框架核心功能：IoC和DI。