课程主题

Spring核心接口和类的介绍&手写Spring IoC模块V3版本&IoC模块源码阅读

课程回顾

画图说明BeanDefinition的注册和Bean实例创建流程

课程目标

1. 搞清楚BeanFactory家族的接口和类的作用（接口隔离原则、抽象模板方法设计模式等）
2. 搞清楚ApplicationContext家族的接口和类的作用
3. 搞清楚BeanDefinitionRegistry和SingletonBeanRegistry的作用（OOA/D）
4. 搞清楚注册BeanDefinition流程中各个类的作用
5. 搞清楚创建Bean实例流程中各个类的作用
6. 通过以上接口和类的理解，我们写出IoC模块的V3版本
7. 可以自主完成阅读Spring框架中BeanDefinition注册流程的源码
8. 可以自主完成阅读Spring框架中Bean实例创建流程的源码
9. 可以自主完成阅读Spring框架中依赖注入流程的源码
10. 可以确定aop流程的源码阅读入口

课程内容

设计模式理解

七大设计原则

> 通过理解七大设计原则，来告诉程序员如何进行面向对象的设计与世界

- 开闭原则：对修改关闭，对扩展开放。一切都是为了保证代码的扩展性和复用性。而开闭原则是基础要求。
- 单一职责原则：单类应该如何定义
- 接口隔离原则：单接口应该如何定义
- 依赖倒置原则：面向接口/抽象编程思维，在方法的返回值、参数类型等都使用接口或者抽象类，而不是使用实现类。
- 里式替换原则：如何去编写继承类的代码，子类不要去覆盖父类已经实现的方法。（抽象模板方法）
- 迪米特法则：最少认知原则，不要和陌生人说话。类与类之间要高内聚，低耦合。
    - 项目经理不要直接去访问与他没有直接关系的测试人员。而是调用测试经理的相关功能。
- 合成复用原则：能用组合关系的情况下，不要使用继承关系。就比如说，如果你想拥有某个对象的功能，不要直接继承它，而是将它作为我的成员变量去使用。

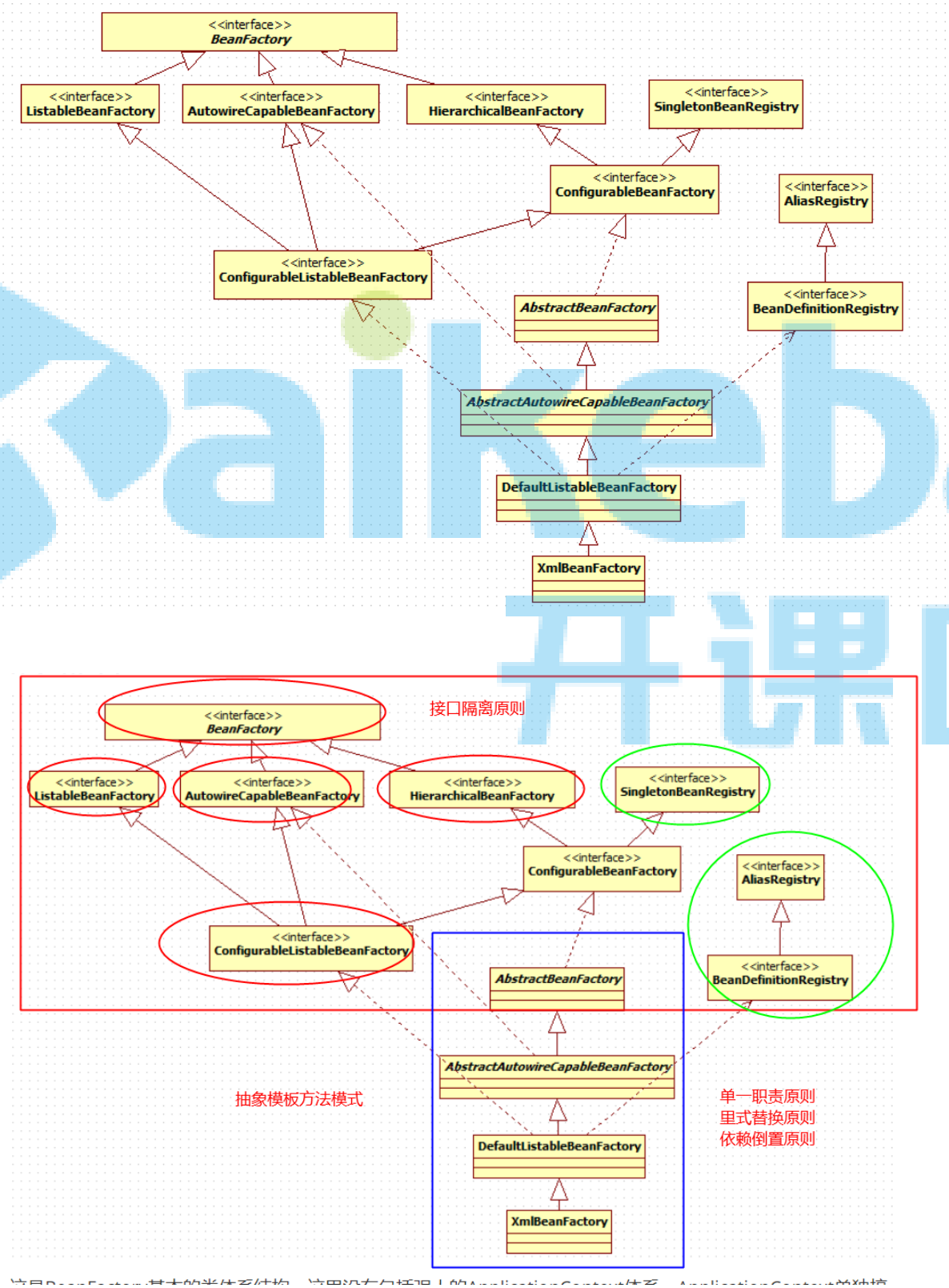
二十三种设计模式

- 创建型设计模式：简单工厂模式、工厂方法模式、抽象工厂模式、单例模式、原型模式、构建者模式。
- 行为型设计模式：责任链模式、观察者模式、门面模式、策略模式、适配器模式等
- 结构型设计模式：组合模式、代理模式、装饰模式等

一、Spring重要接口详解

1.1 BeanFactory继承体系

1.1.1 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.
>
> 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。

> 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。

1.1.2 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);

}
```

- 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。

1.1.3 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的集合，这就是它名字的由来——ListableBeanFactory。**

1.1.4 HierarchicalBeanFactory

分层的Bean工厂

```java
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。

1.1.5 AutowireCapableBeanFactory

自动装配的Bean工厂

```java
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;

    // 与上面类似，不过稍有不同。
    Object autowire(Class<?> beanClass, int autowireMode, boolean
dependencyCheck) throws BeansException;

    /*
     * 根据名称或类型自动装配
     */
```

```
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。

1.1.6 ConfigurableBeanFactory

复杂的配置Bean工厂

```java
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

}
```

1.1.7 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。

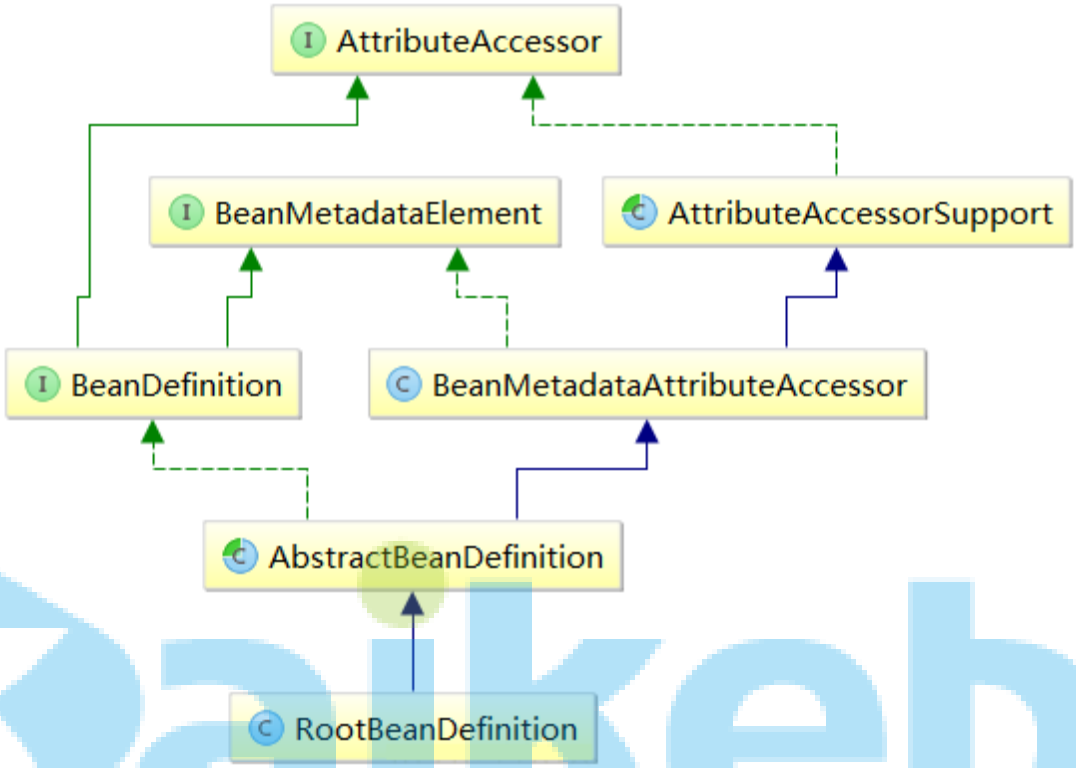
1.1.8 BeanDefinitionRegistry

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;

    /*
     * 根据指定bean名得到对应的Bean定义
     */
    BeanDefinition getBeanDefinition(String beanName) throws
NoSuchBeanDefinitionException;

    /*
     * 查找，指定的Bean名是否包含Bean定义
     */
    boolean containsBeanDefinition(String beanName);

    String[] getBeanDefinitionNames();//返回本容器内所有注册的Bean定义名称

    int getBeanDefinitionCount();//返回本容器内注册的Bean定义数目

    boolean isBeanNameInUse(String beanName);//指定Bean名是否被注册过。

}

```

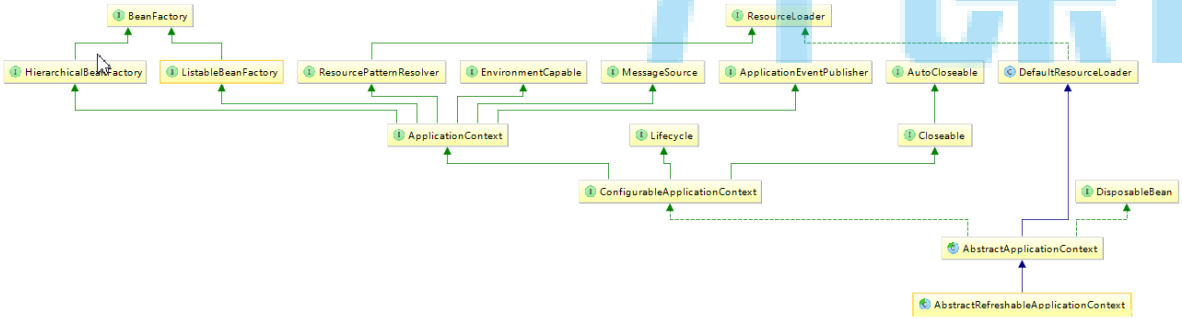
1.2 BeanDefinition继承体系

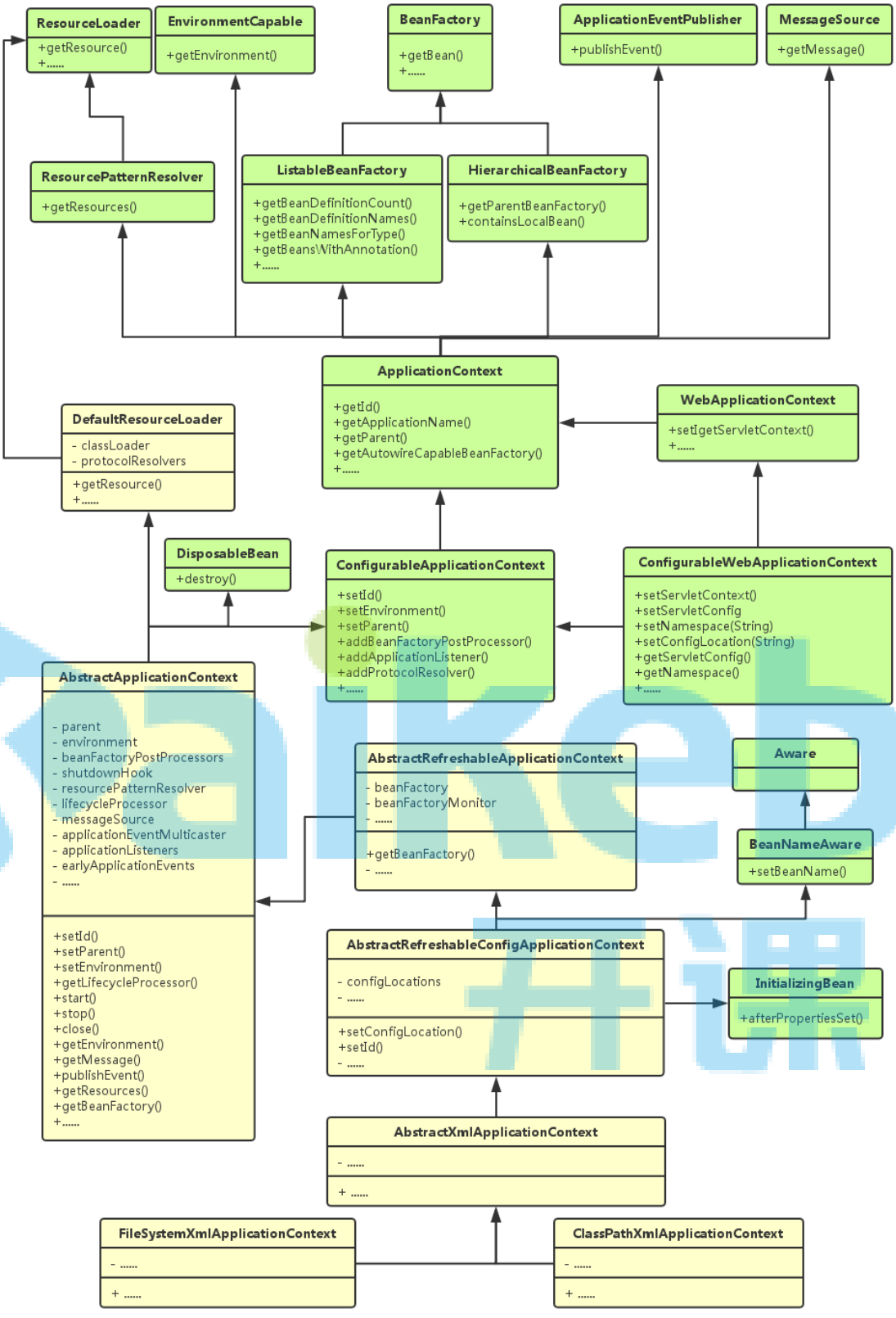
1.2.1 体系结构图



1.3 ApplicationContext继承体系

1.3.1 体系结构图





http://cnblogs.com/zffenger

二、Spring容器初始化流程源码分析

```
1 找入口：一般就是调用第三方框架的时候，这个地方就是入口
```

2.1 主流程源码分析

2.1.1 找入口

- java程序入口

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

```

- web程序入口

```
<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。

2.1.2 流程解析

- AbstractApplicationContext的 refresh 方法

```
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();
            }
        }
    }

```

2.2 创建BeanFactory流程源码分析

2.2.1 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();

```

2.2.2 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());
    }

```

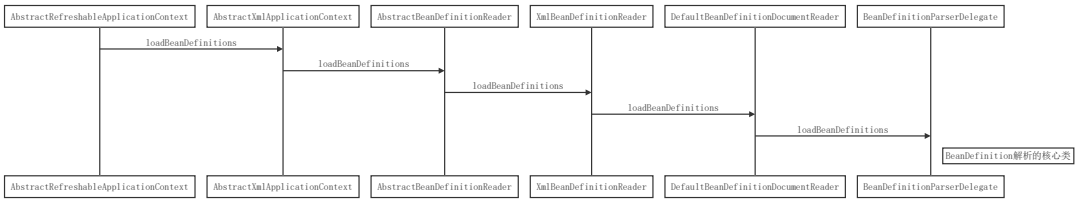
2.3 加载BeanDefinition流程分析

2.3.1 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);
        }
    }

```

2.3.2 流程图



2.3.3 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

2.3.4 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;
    }

```

2.4 Bean实例化流程分析

2.4.1 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);

```

2.4.2 流程解析