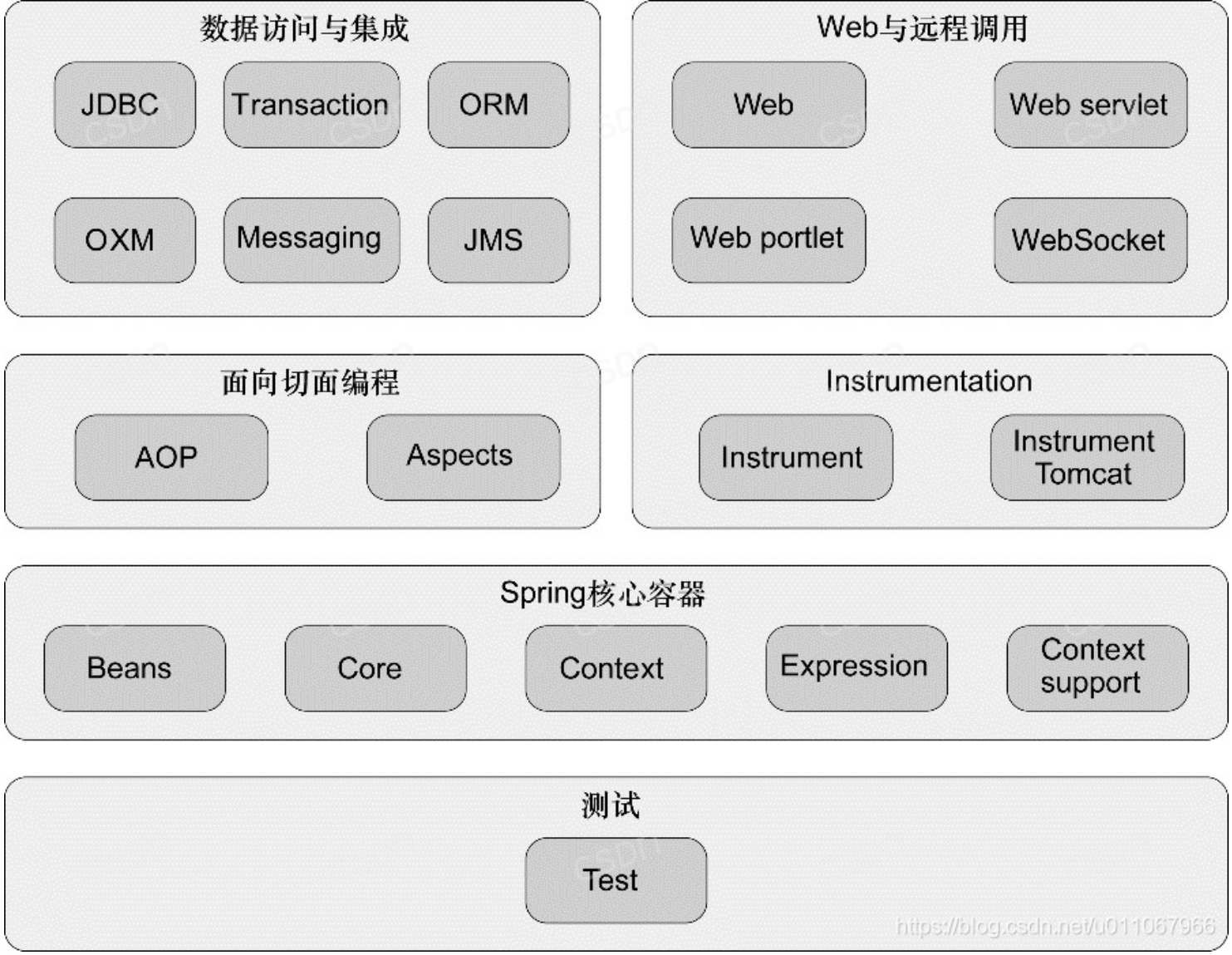
《Spring源码深度解析》

Byte空间 已于 2022-04-10 22:16:48 修改

Spring整体架构和环境搭建

Spring整体架构



Spring Core：框架的最基础部分，提供 IoC 容器，对 bean 进行管理。

Spring Context：继承BeanFactory，提供上下文信息，扩展出JNDI、EJB、电子邮件、国际化等功能。

Spring DAO：提供了JDBC的抽象层，还提供了声明性事务管理方法。

Spring ORM：提供了JPA、JDO、 Hibernate、MyBatis 等ORM映射层.

Spring AOP：集成了所有AOP功能

Spring Web：提供了基础的 Web 开发的上下文信息，现有的Web框架，如JSF、Tapestry、Structs等，提供了集成

Spring Web MVC：提供了 Web 应用的 Model-View-Controller 全功能实现。

环境搭建

参考网上的文章搭建。

容器的基本实现

2.1容器基本用法

```
public class Hello {
    public void sayHello() {
        System.out.println("Hello, spring");
    }
}
public static void main(String[] args) {
        BeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("applicationContext.xml"));
        Hello hello = (Hello)beanFactory.getBean("hello");
        hello.sayHello();
    }
```

xml配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="hello" class="Hello"></bean>

</beans>
```

2.4 Spring的结构构成

2.4.1 beans包的层级结构

整个beans工程的源码包的功能如下：
src/main/java：用于展示Spring的主逻辑；
src/main/resource：用于存放系统的配置文件；

src/test/java：用于主要逻辑进行单元测试；
src/test/resource：用于存放测试用的配置文件。

2.4.2 核心类介绍

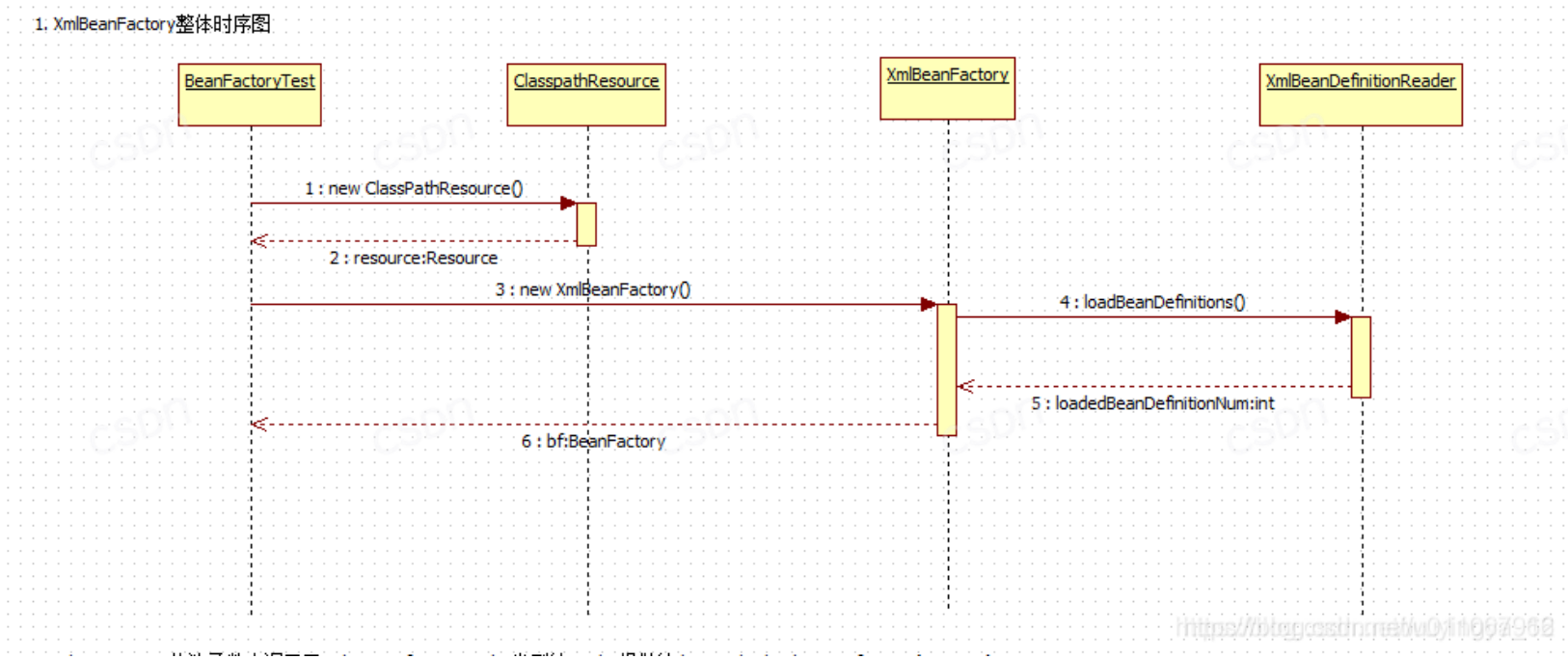
DefaultListableBeanFactory
XmlBeanFactory继承自DefaultListableBeanFactory，而DefaultListableBeanFactory是整个Bean加载的核心部分，是Spring注册以及加载bean的默认实现，而对于XmlBeanFactory与DefaultListableBeanFactory不同的地方其实是在XmlBeanFactory中使用了自定义的XML读取器XmlBeanDefinitionReader，实现了个性化的BeanDefinitionReader读取，DefaultListableBeanFactory继承了AbstractAutowireCapableBeanFactory并实现了ConfigurableListableBeanFactory以及BeanDefinitionRegistry接口。

XmlBeanDefinitionReader
XML配置文件的读取是Spring中重要的功能，因为Spring的大部分功能都是以配置文件作为切入点的XmlBeanDefinitionReader的主要功能就是资源文件读取、解析以及注册。

经过以上分析可以梳理出整个XML配置文件读取的大致力流程：

1. 通过继承自AbstractBeanDefinitionReader中的方法，来使用RourceLoader将资源文件路径转换为对应的Resource文件；
2. 通过DocumentLoader对Resource文件进行转换，将Resource文件转换为Document文件；
3. 通过实现接口BeanDefinitionDocumentReader的DefaultBeanDefinitionDocumentReader类对Document进行解析，并使用BeanDefinitionParserDelegate对Element进行解析。

2.5 容器的基础XmlBeanFactory



2.5.1 配置文件封装

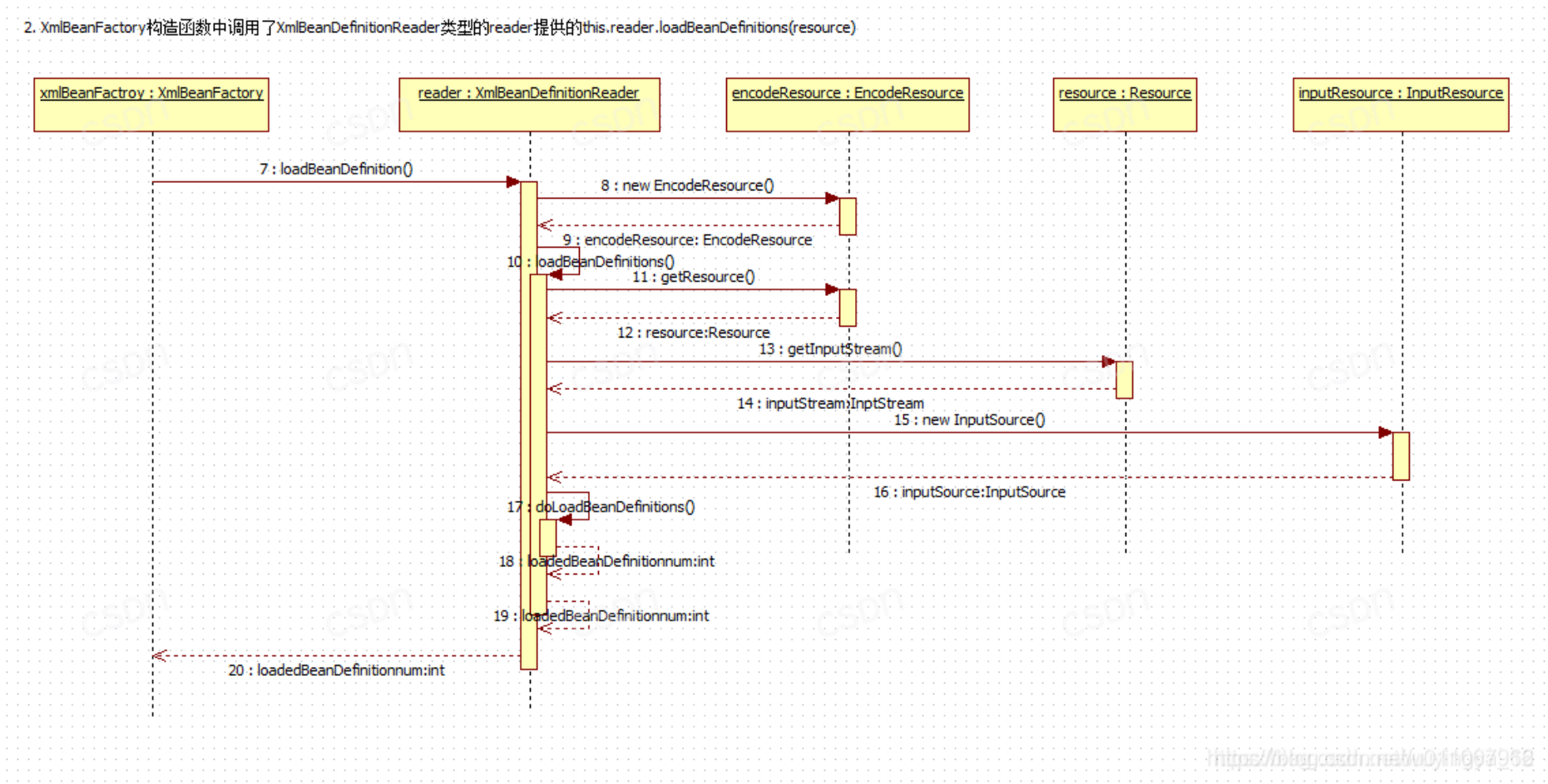
Spring对其内部使用到的资源实现了自己的抽象结构：Resource接口来封装底层资源。

对不同来源的资源文件都有相应的Resource实现：文件(FileSystemResource)、Classpath资源(ClasspathResource)、URL资源(UrlResource)、InputResource(InputStreamResource)、Byte数组(ByteArrayResource)等。

2.5.2 加载Bean

XmlBeanFactory的初始化有若干方法中，this.reader.loadBeanDefinitions(resource)才是资源加载的真正实现

```
1  public XmlBeanFactory(Resource resource, BeanFactory parentBeanFactory) throws BeansException {
2          super(parentBeanFactory);
3          this.reader.loadBeanDefinitions(resource);
4      }
```



整个处理过程如下：

1. 封装资源文件。当进入XmlBeanDefinitionReader后首先对参数Resource使用EncodeResource类进行封装；
2. 获取输入流。从Resource中获取对应的InputStream并构造InputSource；
3. 通过构造的InputSource实例和Resource实例继续调用函数doLoadBeanDefinitions。

2.6获取XML的验证模式

只要理解了XSD和DTD的使用方法，Spring检测验证模式办法就是判断是否包含DOCTYPE，如果包含就是DTD，否则就是XSD。

2.7 获取Document

经过了验证模式准备步骤就可以进行Document加载了，同样XmlBeanFactoryReader对于文档读取并没有亲力亲为，而是委托给了DocumentLoader去执行，这里DocumentLoader只是一个接口，而真正调用的是DefaultDocumentLoader。

2.8 解析及注册BeanDefinitions

当文件转换为Document后，接下来的提取以及注册bean就是重头戏，继续上面的分析，当程序已经拥有XML文档文件的Document实例对象时，就会引入registerBeanDefinition(Document doc,Resource resource)个方法，其中的参数doc是通过上一节loadDocument加载转换出来的。在这个方法中很好的应用了面向对象中单一职责的原则，将逻辑委托给单一的类进行处理，而这个逻辑处理类就是BeanDefinitionDocumentReader。BeanDefinitionDocumentReader是一个接口，而实例化的工作是在createBeanDefinitionDocumentReader()中完成，而通过此方法，BeanDefinitionDocumentReader真正类型其实已经是DefaultBeanDefinitionDocumentReader了，进入DefaultBeanDefinitionDocumentReader后，发行这个方法的重要目的之一就是提取root，以便于再次将root作为参数继续BeanDefinition的注册。即核心逻辑的底层doRegisterBeanDefinitions(root)。

```
        preProcessXml(root);
        parseBeanDefinitions(root, this.delegate);
        postProcessXml(root);
```

默认标签的解析

Spring中的标签包括默认标签和自定义标签，而这两种标签的用法以及解析方式存在着很大不同。
默认标签的解析是在parseDefaultElement函数进行的，函数中的功能逻辑一目了然，分别对4中不同标签(import、alisa、bean和beans)做了不同处理。

```
private void parseDefaultElement(Element ele,BeanDefinitionParserDetegate detegate){
    // 对import标签的处理
importBeanDefinitionResource(ele);
// 对alisa标签的处理
proccessAlisaRegistration(ele);
// 对bean标签的处理
proccessBeanDefinition(ele,detegate);
// 对beans标签的处理0
doRegisterDefinition(ele);
}
```

3.1bean标签的解析以及注册

```
  protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate delegate) {
    BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
    if(bdHolder != null) {
        bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);

        try {
```

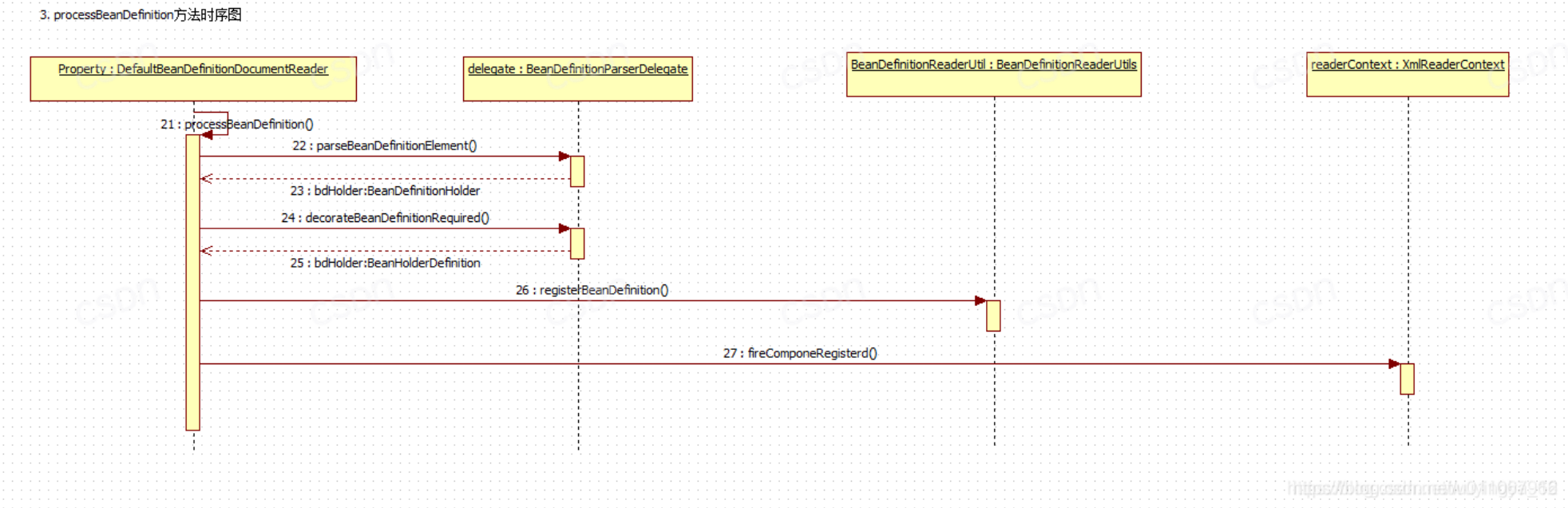
```
            BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder, this.getReaderContext().getRegistry());
        } catch (BeanDefinitionStoreException var5) {
            this.getReaderContext().error("Failed to register bean definition with name \'" + bdHolder.getBeanName() + "\'", ele, var5
        }

        this.getReaderContext().fireComponentRegistered(new BeanComponentDefinition(bdHolder));
    }

}
```

大致逻辑如下：

- 首先委托BeanDefinitionDelegate类的parseBeanDefinitionElement方法进行元素解析，返回BeanDefinitionHolder类型的实例bdHolder，经过这个方法后，bdHolder实例已经包含我们配置文件中配置的各种属性了，例如class、name、id、alias之类的属性
- 当返回的bdHolder不为空的情况下若存在默认标签的子节点下再有自定义属性，还需要再次对自定义标签进行解析
- 解析完成后，需要对解析后bdHolder进行注册，同样，注册操作委托给了BeanDefinitionReaderUtils的registerBeanDefinition方法
- 最后发出响应事件，通知相关的监听器，这个bean已经加载完成了



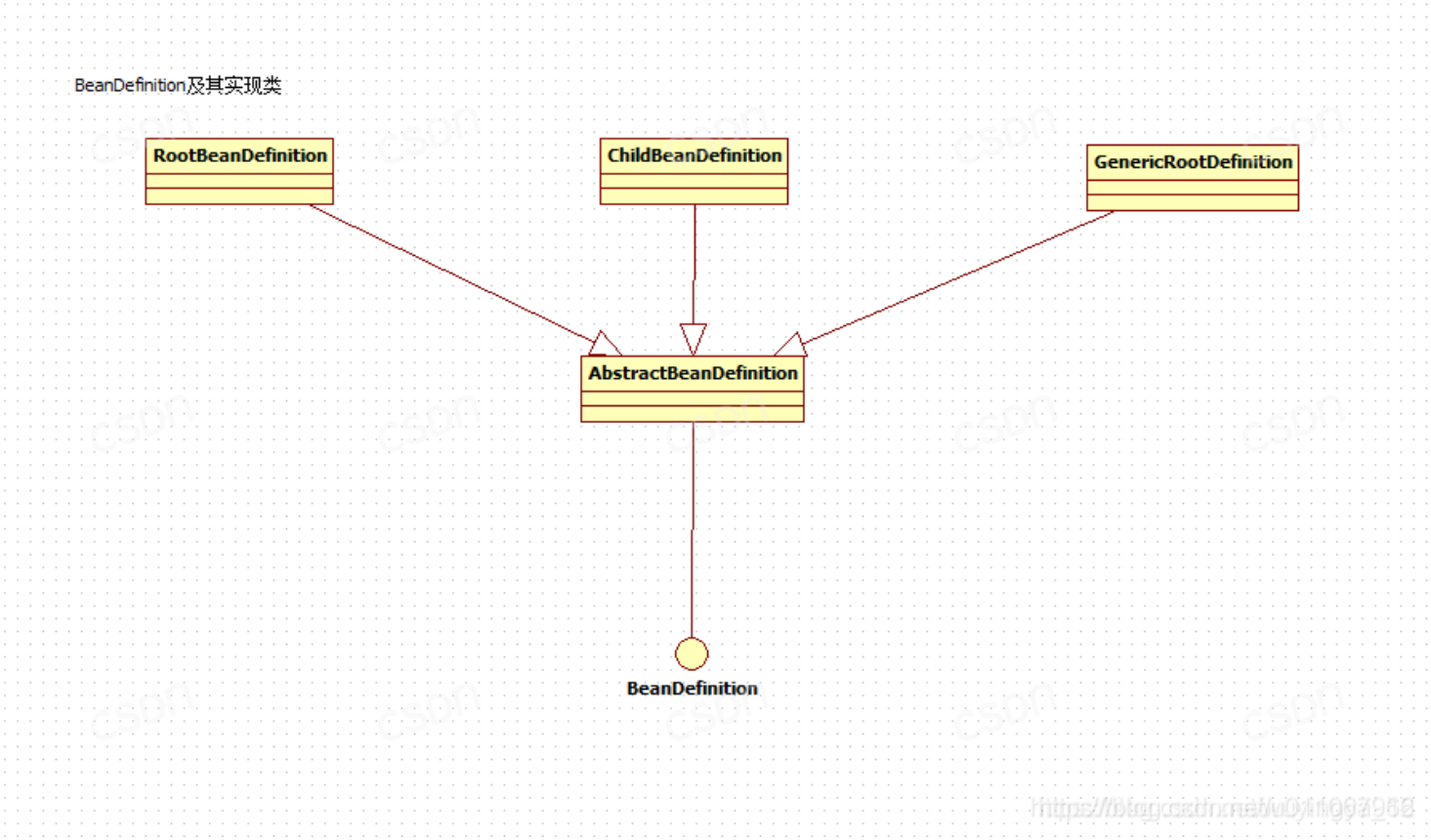
3.1.1bean标签的解析以及注册

首先从元素解析以及信息提取开始，也就是
BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele)，进入BeanDefinitionDelegate类的parseBeanDefinitionElement方法。

1. 创建用于属性承载的BeanDefinition

BeanDefinition是一个接口，在Spring中存在三种实现：RootBeanDefinition、ChildBeanDefinition以及GenericBeanDefinition三种均继承了AbstractBeanDefinition，其中BeanDefinition是配置文件< bean>元素标签在容器中的内部表示形式。< bean>元素标签拥有class、scope、lazy-init等配置属性，BeanDefinition则提供了相应的beanClass、scope、lazyInit属性，BeanDefinition和< bean>中的属性是一一对应的，其中RootBeanDefinition是最常用的实现类，它对应一般性的< bean>元素标签，GenericBeanDefinition是自2.5版本以后新加入到bean文件配置属性的定义类，是一站式服务类。

Spring通过BeanDefinition将配置信息转换为容器内部标识，并将这些BeanDefinition注册到BeanDefinitionRegistry中，Spring容器的BeanDefinitionRegistry就像Spring配置信息的内存数据库，主要以map的形式保存，后续的操作直接从BeanDefinitionRegistry中读取配置信息。



2. 解析各种属性

当创建bean信息的承载实例后，并可以进行bean信息的各种属性解析，首先进入parseBeanDefinitionAttributes方法。parseBeanDefinitionAttributes方法对于element所有元素属性进行解析：

```
public AbstractBeanDefinition parseBeanDefinitionAttributes(Element ele, String beanName,
            @Nullable BeanDefinition containingBean, AbstractBeanDefinition bd) {
    // 解析scope属性
    // 解析singleton属性
    // 解析abstract属性
    // 解析lazy-init属性
    // 解析autowire属性
    // 解析dependency-check属性
    // 解析denpends-on属性
    // 解析autowire-candidate属性
    // 解析primary属性
    // 解析init-method属性
```

```
    // 解析destory-method属性
    // 解析factory-method属性
    // 解析factory-bean属性
}
```

3. 解析子元素meta

```
public void parseMetaElement(Element ele,BeanMetadataAttributeAccessor attributeAccessor)
// 获取当前节点的所有元素
// 提取meta
// 使用key、value构造BeanMetadataAttribute
// 记录信息
}
```

4. 解析子元素lookup-method
   子元素lookup-method似乎并不是很常用，但是在某些时候它的确是非常有用的，通常称它为获取器注入，引用《Spring in Action》中的一句话：获取器注入是一种特殊的方法注入，它是一个方法声明为放回某种类型的bean，但是实际要返回的bean是在配置文件里面配置的，此方法在涉及有些可插拔的功能上，解除程序依赖。

5. 解析子元素replaced-method
   这个方法主要是针对bean中replaced-method子元素的提取，方法替换：可以在运行时用新的方法替换现有的方法，与之前look-up不同的是，replace-method不但可以动态的替换返回实体bean，而且还能动态的更改原有方法的逻辑。

6. 解析子元素constructor-arg
   对于construtor-arg子元素的解析，Spring通过parseConstructorArgElements函数来实现的

7. 解析子元素property
   parsePropertyElement函数完成了对property属性的提取，具体的解析过程如下：

8. 解析子元素qualifier
   对于qualifier元素的获取，接触更多的是注解的形式，在使用Spring框架中进行自动注入时，Spring容器中匹配的候选Bean数目必须有且仅有一个，当找找不到一个匹配的Bean时，Spring容器将会抛出BeanCreationException异常，并指出必须至少拥有一个匹配的Bean。

Spring允许通过Qualifier指定注入Bean的名称，这样就消除歧义了。

3.1.2 AbstractBeanDefinition属性

至此便完成了对XML文档到GenericBeanDefinition的转换，也就是到这里，XML中所有配置都可以在GenericBeanDefinition的实例中找到对应的配置。

GenericBeanDefinition只是子类实现，而大部分通用属性都保存在了AbstractBeanDefinition中。

```
public abstract class AbstractDeanDefinition extends BeanMetadateAttributeAccessor implements BeanDefinition,Cloneable{
// bean的作用范围，对应bean属性的scope
// 是否是单例，来自bean属性scope
// 是否是原型，来自bean属性scope
// 是否是抽象，来自bean属性abstract
// 是否延迟加载，来自lazy-init
// 自动注入模式，对应bean属性autowire
// 依赖检查，Spring 3.0后弃用这个属性
// 用来表示一个bean的实例化靠另一个bean先实例化，对应bean属性depend-on
// autowire-candidate属性设置为false，这样容器在查找自动装配对象时，将不考虑该bean，即它不会被考虑作为其他bean自动装配的候选者，但是该bean还是可以使用自动
// 自动装配时当出现多个bean候选者时，将作为首选者，对应bean属性primary
// 用于记录Qualifier，对应子元素Qualifier
// 允许访问非公开的构造器和方法，程序设置
// 是否以一种宽松的模式解析构造函数
// 记录构造函数注入属性，对应bean属性constructor-arg
// 普通属性集合
// 方法重写的持有者，记录lookup-method、replace-method元素
// 对应bean属性factory-bean
// 对应bean属性factory-method
// 初始化方法，对应bean属性init-mothod方法
// 销毁方法，对应bean属性destory-method方法
// 是否执行init-mothod方法，程序设置
// 是否执行destory-method方法，程序设置
// 是否是用户定义的而不是应用程序本身定义的，创建AOP时候为true，程序设置
// 定义bean的应用，APPLICATION：用户；INFRASRUCTRUE：完全内部使用，与用户无关；SUPPORT：某些复杂配置的一部分，程序设置
// bean的描述信息
// 这个bean定义的资源
}

```

3.1.3 解析默认标签中的自定义标签元素

```
public BeanDefinitionHolder decorateBeanDefinitionIfRequire(Element ele,BeanDefinitionHolder definitionHolder,null)
```

可以看到函数分别对元素的所有属性以及子元素进行了decorateIfReuired函数的调用

3.1.4 注册解析的BeanDefinition

对于配置文件，解析是解析完了，装饰也装饰完了，对于得到的beanDefinition已经可以满足后续的使用要求了，为剩下的工作就是注册了，也就是processBeanDefinition函数中的BeanDefinitionReaderUtils.registryBeanDefinition(bdHolder,getReaderContext().getRegistry())代码的解析了

```
public static void registryBeanDefinition(BeanDefinitionHolder definitionHolder,BeanDefinitionRegistry registry)
// 使用beanName做唯一标识注册
// 注册所有的别名
```

1. 通过beanName注册BeanDefinition
   对于beanName的注册，或许许多人认为的方式就是将beanDefinition直接放入map中就好了，使用beanName作为key，确实，Spring就是这么做的，只不过除此之外，它还做了点别的事

```
public void registryBeanDefinition(String beanName,Definition beanDefinition) throws BeanDefinitionStoreException
// 注册前的最后一次校验，这里的校验不同于之前的XML文件校验，主要是对于AbstractBeanDefinition属性中的methodOverrides校验，校验methodOverrides是否与工厂
// 因为beanDefinitionMap是全局变量，这里肯定会存在并发访问的情况
// 处理注册已经注册的beanName情况
// 如果对应的BeanName已经注册且在配置中配置了bean不允许被覆盖，则抛出异常
// 记录beanName
// 注册beanDefinition
// 重置所有beanName对应的缓存
}
```

上面的代码可以看出，对于bean的注册处理方法上，主要进行了几个步骤：

- 对AbstractBeanDefinition的校验。在解析XML文件的时候，是针对XML格式的校验，而此时的校验时是对于AbstractBeanDefinition的methodOverrides属性的；
- 对beanName已经注册情况的处理。如果没有设置不允许bean的覆盖，则需要抛出异常。 加入map缓存
- 清除解析之前留下的对应beanName的缓存

2. 通过别名注册BeanDefinition
   在理解了注册bean的原理之后，理解别名注册的原理就容易多了

```
public void registryAlias(String name,String alisa)
// 如果beanName与alisa相同的话不记录alisa，并删除对应的alisa
// 如果alisa不允许被覆盖则抛异常
}
```

由以上代码中可以得知注册alisa的步骤如下：

- alisa与beanName相同情况处理，若alisa与beanName并名称相同则不要处理并删除原有alisa；
- alisa覆盖处理，若alisaName已经使用并已经指向了另一beanName则需要用户的设置进行处理；
- alisa循环检查，单A->B存在时，若再次出现A->C->B的时候则会抛出异常； 注册alisa。

3.2 alisa标签的解析

在对bean进行定义时，除了使用id属性来指定名称之外，为了提供对个名称，可以使用alisa标签来指定，而所有的这些名称都指向同一个bean，在某些情况下提供别名非常有用，比如为了让应用的每一个组件能更容易的对公共组件进行引用。

```
processAlisaRegistation(Elemet ele)
// 获取beanName
// 获取alisa
// 注册alisa
// 别名注册后通知监听器做相应处理
}
```

3.3 import标签的解析

对于Spring配置文件的编写，分模块是大多数人能想到的方法，使用import是个好办法，applicationContext.xml文件中使用import的方法导入有模块配置文件，以后如果有新的模块的加入，那就可以简单修改这个文件了，这样可以大大简化了配置后期维护的复杂度，并使配置模块化，易于管理，import标签的解析方法：

```
protected void importBeanDefinitionResource(Element ele)
// 获取resource属性
// 如果不存在resource属性在不做处理
// 解析系统属性，格式如：“${user.dir}”
// 判定location是绝对URI还是先对URI
// 如果是绝对URI则直接根据地址加载对应的配置文件
// 如果是相对地址则根据相对地址计算出绝对地址
// Resource存在多个子类实现，而每个resource的createRelative方法都不一样，所以先使用子类的方法进行解析
// 如果解析不成功，则使用默认解析器ResourcePatternResolver进行解析
}
```

自定义标签的解析

bean的加载

经过前面的分析，终于结束了对XML配置文件的解析， 接下来就是对bean加载的探索，bean加载功能的实现远比bean的解析复杂的多，对于加载bean的功能，在Spring中的调用方法为：

```
public Object getBean(String name) throws BeanException
return doGetBean(name,null,null,false)

protected < T> T doGetBean(final String beanName,final Class< T> requireType,final Object [] args,boolean typeCheckOnly) throws BeanEx
// 提取对应的beanName
// 检查缓存中或者实例工厂中是否有对应的实例，为什么首先使用这段代码呢，因为在创建单例bean的时候会存在依赖注入的情况，而在创建依赖的时候为了避免循环依赖，Spring
// 直接尝试从缓存或singletonFactories中的ObjectFactory中获取
// 返回对应的示例，有时候存在BeanFactory的情况并不是直接返回实例本身而是返回指定方法返回实例
// 当在单例情况下才会尝试解决循环依赖，原型模式情况下，如果存在A中有B属性，B中有A属性，那么依赖注入的时候，就会产生当A还未创建完的时候因为对于B的创建再次返回创
// 如果beanDefinitionMap中也就是在所有已加载的类中不包括beanName则尝试从parentBeanFactory中检测
// 递归到BeanFactory中寻找
// 如果不是仅仅做类型检查而是创建bean，需要进行记录
// 将储存在GenericBeanDefinition转换为RootBeanDefinition，如果指定BeanName是子Bean的话同时会合并父类的相关属性
// 若存在依赖则需要递归实例化依赖的bean，并缓存依赖调用
// 实例化依赖的bean后便可以实例化mbd本身了
// singleton模式的创建
// prototype模式的创建
// 指定scope上实例化bean
// 检查需要的类型是否符合bean的实际类型
}
```

1. 转换对应的beanName
   这里传入的参数有可能是别名，也有可能是FactoryBean，所以需要进行一系列的解析，这些解析内容包括如下内容。
   1.1 去除FactroyBean的修饰符，也就是如果那么=“&aa”，那么会首先去除&而使name=“aa”。
   1.2 去除alisa所表示的最终beanName，例如别名A指向名称为B的bean则返回B；若别名A指向别名B，别名B又指向名称为C的bean则返回C。
2. 尝试从缓存加载单例
   单例在Spring的统一个容器中只会被创建一次，后续在获取bean，就直接从单例缓存中获取了。当然这里也尝试加载，如果不成功则再次尝试从singletonFactories中加载。因为在创建单例bean的时候会在存在依赖注入的情况，而在创建依赖的时候为了避免循环依赖，在Spring创建bean的原则是不等bean创建完成就会将创建bean的ObjectFactory提早曝光加入到缓存中，一旦下一个bean创建时候需要依赖上一个bean直接使用ObjectFactory。
3. bean的实例化
   如果从缓存中得到了bean的原始状态，则需要对bean进行实例化。这里必须强调一下，缓存中记录的只是最原始的bean状态，并不一定是最终想要的bean。所有使用getObjectForBeanInstance完成这个工作。

4. 原型模式的依赖检查
   只用在单例情况下才会尝试解决循环依赖，如果存在A中有B的属性，B中有A的属性，那么当依赖注入的时候，就是产生当A还未创建完的时候因为对于B的创建再次返回创建A，造成循环依赖，也就是情况：isPrototypeCurrentlyInCreation(beanName)判断为true。

5. 检测parentBeanFactory
   从代码上看，如果缓存没有数据的话直接转到父类工厂上去加载了，这是为什么呢？它是检测如果加载的XML配置文件中不包含beanName所对应的配置，就只能到parentBeanFactory去尝试加载了，然后再去递归的调用getBean方法。

6. 将储存XML配置文件的GenericBeanDefinition转换为RootBeanDefinition。
   因为从XML配置文件读取到的Bean的信息是存储在GenericBeanDefinition中的，但是所有的Bean后续处理都是针对RootBeanDefinition的，所以这里需要进行一个转换，转换的同时如果父类bean不为空的话，则会一并合并父类的属性。

7. 寻找依赖
   因为bean的初始化过程中可能会用到某些属性，而某些属性可能是动态配置且配置成依赖于其他的bean，那么这个时候就有必要先加载依赖的bean，所以，在Spring加载顺序中在初始化某一个bean的时候首先会初始化这个bean所对应的依赖。

8. 针对不同的scope进行bean的创建，在Spring中存在不同的scope，其中默认的是singleton，但是还有注入prototype，request之类的，在这个步骤中，Spring会根据不同的配置进行不同的初始化策略。

9. 类型转换
   程序到这里返回bean后基本结束了，通常对该方法的调用参数requireType是为空的，但是可能会存在这样的情况，返回的bean其实是一个String，但是requireType却传入Integer类型，那么这时候本步骤就会起作用了，它的功能就是将返回的bean转换为requireType所指定的类型。当然，String转换为Integer是最简单的一种转换，在Spring中提供了各种各样的转换器，用户也可以自己扩展转换器来满足自己的需求。

5.1 FactoryBean的使用

一般情况下，Spring通过反射机制利用bean的class属性指定实现类来实例化bean。在某些情况下，实例化bean过程比较复杂，如果按照传统方式，则需要在< bean>中提取大量的配置信息，配置方式的灵活性是受限的，这时候采用编码的方式可能会得到一个简单方案。Spring为此提供了一个org.SpringFramework.bean.factory.FactoryBean工厂类接口，用户可以通过实现该接口定制实现实例化bean的逻辑。

```
public interface FactoryBean<T> {
    T getObject：返回有FactoryBean创建的bean实例，如果isSingleton()返回true，则该实例还会放到Spring容器中单实例缓存池中。
    boolean isSingleton()：返回有FactoryBean创建的bean实例的作用域是singleton还是prototype。
    Class< T> getObjectType() ：返回FactoryBean创建的bean类型。
}
```

当配置文件中< bean>class属性配饰的实现类是FactoryBean时，通过getBean()方法返回的不是FactoryBean本身，而是FactoryBean#getObject()方法所返回的对象，相当于FactoryBean#getObject()方法代理了getBean()方法。

5.2 缓存获取单例bean

介绍过FactoryBean的用法后，就可以了解bean的加载过程了。前面提到过，单例在Spring的同一个容器内只会被创建一次，后续再获取bean直接从单例缓存中获取，当然这里只是尝试加载，首先尝试从缓存中加载，然后再次尝试从singletonFactories中加载。因为在创建单例bean的时候会存在依赖注入的情况，而在创建依赖的时候为了避免循环依赖，Spring创建bean的原则是不等bean创建完成就会将bean的ObjectFactory提早曝光加入到缓存中，一旦下一个bean创建时需要依赖上一个bean吗，则直接使用ObjectFactory。

```
protected Object getSingleton(String beanName,boolean allowEarlyReference)
// 检查缓存中是否存在实例
// 如果为空，则锁定全局变量并进行处理
// 如果此bean正在加载则不处理
// 当某些方法需要提前初始化的时候则会调用addSingletonFactory方法将对应的ObjectFactory初始化策略存储在singletonFactories
// 调用预先设定的getObject方法
// 记录在缓存中，earlySingletonObjects和singletonFactories互斥
}
```

- singletonObjects：用于保存BeanName和创建bean实例之间的关系，bean name --> bean instance
- singletonFactories：用于保存BeanName和创建bean的工厂之间的关系，bean name --> ObjectFactory
- earlySingletonObjects：也是保存BeanName和创建bean实例之间的关系，与singletonObjects不同之处是，当一个单例bean被放到这里时，那么当bean还在创建过程中，就可以通过getBean方法获取到了，其目的是用来检测循环引用的。
- registeredSingleton：用于保存当前所用已注册的bean。

5.3 从bean的实例中获取对象

在getBean中，getObjectForBeanInstance是个高频率使用的方法，无论是从缓存获得bean还是根据不同的scope策略加载bean。总之，我们得到bean的实例后第一步就是调用这个方法来检验正确性，其实就是用于检测当前bean是否是FactoryBean类型的bean，如果是，那么需要调用该bean对应的FactoryBean实例的getObject()作为返回值。

5.4 获取单例

之前讲解了从缓存获取单例的过程，那么，如果缓存中不存在已经加载的单例bean就需要从头开始bean的加载过程了，而Spring中使用getSingleton的重载方法实现bean的加载过程。

```
public Object getSingleton(String beanName, ObjectFactory singletonFactory)
// 全局变量需要同步
// 首先检查对应的bean是否已经加载过，因为singleton模式其实就是复用已创建的bean，所以这一步是必须的
// 如果为空才可以进行singleton的bean的初始化
// 初始化bean
```

```
// 加入缓存
}
```

上述代码中其实是使用回调方法，使得程序可以在单例创建的前后做一些准备及处理操作，而真正的获取单例bean方法其实并不是在此方法中实现的，其实现逻辑是在ObjectFactory类型的实例singletonFactory中实现的，而这些准备及处理操作包括如下内容：

- 检查缓存是否已经加载过；
- 如没有加载，则记录beanName的正在加载状态；
- 加载单例前记录状态。

ObjectFactory的核心部分其实只是调用了createBean的方法，所以还需要到createBean方法中寻求真理。

```
getSingleton(beanName,new ObjectFactory< Object>()
// return createBean(beanName,mbd,args)
```

5.5 准备创建bean

我们不可能指望在一个函数中完成一个复杂的逻辑，而且我们跟踪了这么多Spring的代码。经历的这么多函数，或多或少也发现一些规律：一个真正干活的函数其实是以do开头的；而给我们错觉的函数，其实只是从全局的角度去做一些统筹的工作。这个规律对于createBean也不例外，那么createBean函数中做了哪些准备工作：

```
protected Object createBean(final String beanName,final RootBeanDefinition mbd,final Object[] args) throws BeanCreationException
// 锁定class，根据设置的class属性或者根据className来解析Class
// 验证及准备覆盖的方法
// 给BeanPostProcessors一个机会返回代理来替代真正的实例
// 调用doCreateBean()方法
}
```

从代码中可以总结出函数完成的具体步骤及功能

- 根据设置的class属性或者根据className来解析Class；
- 对override属性进行标记及验证。

5.5.1 处理override属性

```
protected void prepareMethodOverride(MethodOverride mo) throws BeanDefinitionValidationException
// 获取对应类中方法名的个数
// 如果个数为1时，标记MethodOverride暂未被覆盖，避免参数类型检查的开销
}
```

通过以上两个函数的代码实现了之前反复提到过的，在Spring配置中存在lookup-method和replace-method两个配置功能，而这个配置的加载其实就是将配置统一存放在BeanDefinition中的methodOverrides属性中，这两个功能的实现原理其实是在bean实例化的时候如果检测到存在methodOverrides属性，会动态地为当前bean生成代理并使用对应的拦截器为bean做增强处理。

5.5.2 实例化的前置处理

在真正调用doCreate方法创建bean的实例前使用了这样一个方法resolveBeforeInstantiation(beanName,mbd)对BeanDefinition中的属性做前置处理。当然，无论其中是否有相应的逻辑实现我们都可以理解，因为真正的逻辑实现前后留有处理函数也是可扩展的一种体现。但是这并不是最重要的，在函数中还提供了一个短路判断，这才是最关键的部分。

当经过前置处理后返回的结果如果不为空，那么会直接略过后续的Bean的创建而直接返回结果。这个特性判断虽然很容易被忽略，但是却起着至关重要的作用，我们熟知的AOP功能就是基于这里判断的。

```
protected Object resolveBeforeInstantiation(String beanName,RootBeanDefinition mbd)


```

此方法中最吸引人的无疑是两个方法：applyBeanPostProcessBeforeInstantiation以及applyBeanPostProcessAfterInitialization。这两个方法实现非常简单，无非是对后处理器中所有InstantiationAwareBeanPostProcessor类型的后处理器进行postProcessBeforeInstantiation方法和BeanPostProcessAfterInitialization方法的调用。

1. 实例化前的后处理器的应用
   bean的实例化前调用，就是将AbstractBeanDefinition转换为BeanWrapper前的处理，给子类一个修改BeanDefinition的机会，也就是说当程序经过这个方法后，bean可能已经不是我们认识的bean了，而是或许成为了一个经过处理的代理bean，可能通过cglib生成的，也可以是其他技术生成的。

2. 实例化后的后处理器的应用
   在讲解从缓存中获取单例bean的时候提到过，Spring中的规则是在bean的初始化后尽可能保证将注册的后处理器的postProcessAfterInitialization方法应用到该bean中，因为如果返回的bean不为空，那么便不会再次经历普通bean的创建过程，所以只能在这里应用后处理器的postProcessAfterInitialization方法。

5.6 循环依赖

实例化bean是一个非常复杂的过程，而其中最难以理解的就是对循环依赖的解决。

5.6.1 什么是循环依赖

循环依赖就是循环引用，就是两个或多个bean之间的持有对方，比如ClassA引用ClassB，ClassB引用ClassC，ClassC引用ClassA，则最终反应为一个环。此处不是循环调用，循环调用是方法之间的环调用。

循环调用是无法解决的，除非有终结条件，否则就是死循环，最终导致内存溢出错误。

5.6.2 Spring如何解决循环依赖

Spring容器循环依赖包括构造器循环依赖和setter循环依赖，那么Spring容器是如何解决循环依赖的呢?

1. 构造器循环依赖
   表示通过构造器注入构造的循环依赖，此依赖是无法解决的，只能抛出BeanCurrentlyInCreationException异常表示循环依赖。

如何创建TestA类时，构造器需要TestB类，那么将去创建TestB，在创建TestB类时又发现需要TestC类，则去创建TestC，最终在创建TestC时发现又需要TestA，从而形成一个环，没有办法创建。

Spring容器将每一个正在创建的bean标识符放在一个“当前创建bean池”中，bean标识符在创建过程中将一直保持在这个池中，因此在创建bean过程中发现自己已经在“当前创建bean池”里时，将抛出BeanCurrentlyInCreationException异常表示循环依赖，而对于创建完毕的bean将从“当前创建bean池”中清除掉。

2. setter循环依赖
   表示通过setter注入方式构成的循环依赖，对于setter注入方式构成的循环依赖是通过Spring容器提前暴露刚创建完成构造器注入单位完成其他步骤的bean来完成的，而且只能解决单例作用域下的bean循环依赖。通过提前暴露一个单例工厂方法，从而使其他bean能引用到该bean。

3. prototype范围的依赖处理
   对于prototype作用域bean，Spring容器无法完成依赖注入，因为Spring容器不进行缓存prototype作用域的bean，因为无法提前暴露一个创建的bean。

对于singleton作用域bean，通过“setAllowCirularReferences(false)”来禁用循环引用。

5.7 创建bean(AbstractAutowireCapableBeanFactory)

介绍了循环依赖以及Spring中的循环依赖的处理方法后。当经过resolveBeforeInstantitation方法后，程序有两个选择，如果创建了代理或者重写可InstantiationAwareBeanPostProcessor的postProcessBeforeInstantiation方法并在方法postProcessBeforeInstantiation中改变bean，则直接方法就可以了，否则进行常规bean的创建，而常规bean的创建就是在doCreateBean中完成。

```
protected Object doCreateBean(final String beanName,final RootBeanDefinition mbd,final Object[] args)
// 根据指定bean使用对应的策略创建新的实例，如：工厂模式、构造函数自动注入、简单初始化等
// 是否提早曝光：单例&允许循环依赖&当前bean是否在创建中，检测循环依赖
// 为了避免后期循环依赖，可以在bean初始化完成钱将创建实例的ObjectFactory加入工厂
// 对bean在一次依赖引用，主要应用SmartInstantiationAwareBeanPostProcessor，其中我们熟知的AOP就是在这里将advice动态织入bean中，如没有则直接返回
// 对bean属性填充，将各个属性值注入，其中，可能存在依赖于其他bean的属性，则会递归初始化依赖bean
// 调用初始化方法，比如init-method
// earlySingletonReference只检测到有循环依赖的情况下才不会为空
// 如果exposeObject没有在初始化方法中被改变，也就是没有增强
// 检测依赖
// 因为bean创建其所依赖的bean一定是已经创建了，actualDependentBeans不为空则表示当前bean却没有没全部创建完，也就是说存在循环依赖

```

```
13  // 根据scope注册bean
    }
```

我们来看看整个函数的概要思路：

- 如果是单例则需要首先清除缓存；
- 实例化bean，将BeanDefinition转换为BeanWrapper。
- MergedBeanDefinitionPostProcessor的应用，bean合并后的处理，Autowire正是通过此方法实现诸如类型的解析。
- 依赖处理，在Spring中会有依赖循环的情况，例如，当A中含有B的属性，而B中又含有A的属性是就会构成一个循环依赖，此时如果A和B都是单例，那么在Spring中的处理方式就是当创建B的时候，涉及自动注入A的步骤时，并不是直接去再次创建A，而是通过放入缓存中的ObjectFactory来创建实例，这样就解决了循环依赖的问题了。
- 属性填充，将所有属性填充至bean的实例中。
- 循环依赖检查，之前提到过，在Spring中解决循环依赖只对单例有效，而对于prototype的bean，Spring没有好的解决办法，唯一做的就是抛出异常，在这个步骤里面会检测已经加载的bean是否已经出现了依赖循环，并判断是否需要抛出异常。
- 注册DisposeableBean，如果配置了destroy-method，这里需要注册以便于在销毁时候调用。
- 完成创建并返回。

可以看出上面的步骤非常的繁琐，每一步骤使用大量的代码来完成其功能，最复杂也是最难理解的当属循环依赖的处理，在真正进入doCreateBean前有必要先了解下循环依赖。

5.7.1 创建bean的实例

当了解了循环依赖以后就可以深入分析创建bean的每一个步骤了，首先从createBeanInstance开始。

```
1  protected BeanWapper createBeanInstance(String beanName,RootBeanDefinition mbd,Object [] args)
2  // 解析class
3  // 如果工厂方法不为空则使用工厂方法初始化策略
4  // 一个类有多个构造函数，每个构造函数都有不同的参数，所有调用前需要先根据参数锁定构造函数或对应的工厂方法
5  // 如果已经解析过则使用解析好的构造函数方法不需要再次锁定
6  // 构造函数自动注入
7  // 使用默认构造函数构造
8  // 需要根据参数解析构造函数
9  // 构造函数自动注入
10 // 使用默认构造函数构造
11 }
```

1. autowireConstructor

   对于实例的创建Spring中分为两种情况，一种是通用的实例化，另一种是带有参数的实例化。带有参数的实例化过程相当复杂，因为存在着不确定行，所以在判断对应参数上做了大量的工作。

```
public BeanWrapper autowireConstructor(final String beanName,final RootBeanDefinition mbd,Constructor [] chosenCtors,final Object [] e
// explicitArgs通过getBean方法传入，如果getBean方法调用的时候指定方法参数那么直接使用
// 如果getBean方法时候没有指定则尝试从配置文件中解析
// 尝试从缓存中获取
// 配置的构造函数参数
// 如果缓存中存在
// 解析参数类型，缓存中的值可能是原始值也可能是最终值
// 如果没有缓存
// 提取配置文件中的配置的构造函数参数
// 用于承载解析后的构造函数参数的值
// 能解析到的参数个数
// 排序给定的构造函数，public构造函数优先参数数量降序，非public构造函数参数数量降序
// 如果已找到选用的构造函数或者需要参数个数小于当前构造函数参数给则终止，以为已经按照参数个数降序排列
// 有参数则根据构造对应参数类型的参数
// 注释上获取参数名称
// 获取参数名探索器
// 获取指定构造函数的参数名称
// 根据名称和数据类型创建参数持有者
// 构造函数没有参数的情况
// 探测是否有不确定性的构造函数存在，例如不同构造函数的参数为父子关系
// 将解析的构造函数加入缓存
// 将构造的实例加入BeanWrapper
}
```

总览一下整个函数，其实现的功能考虑了一下几个方面：

- 构造函数参数确定。

  1.1 根据explicitArgs参数判断，如果传入的explicitArgs参数不为空，那便可以直接确认参数，因为explicitArgs参数是在调用Bean的时候用户指定的，在BeanFactory类中存在这样的方法：

  Object getBean(String name,Object … args) throws BeanException

  在获取bean的时候，用户不但可以指定bean的名称还可以指定bean所对应类的构造函数或者工厂方法的方法参数，主要用于静态工厂方法的调用，而这里需要给定完全匹配的参数的，所以，便可以判断，如果传入参数explicitArgs不为空，则可以确定构造函数参数就是它。

1.2 缓存中获取，除此之外，确定参数的办法如果之前已经分析过，也就是说构造函数已经记录在缓存中，那么便可以直接拿来使用。而且，这里要提到的是，在缓存中缓存的可能是参数的最终类型也可能是参数的初始类型。

1.3 配置文件获取，如果不能根据传入参数explicitArgs确认构造函数的参数也无法在缓存中得到相关信息，那么只能进行新一轮的分析了。分析从配置文件的构造韩式信息开始，经过之前的分析，Spring配置文件中的信息经过转换会通过BeanDefinition实例承载，也就是参数mbd中包含，那么可以通过调用mbd.getConstructorArgumentValues()来获取配置的构造函数信息。有了配置中的信息便可以获取对应的参数值信息了，获取参数值的信息包括指定值，如：直接指定构造函数中某个值为原始类型String类型，或者是一个对其他bean的引用，而这一处理委托给了resolveConstructorArgments方法，并返回能解析到的参数的个数。

- 构造函数的确定，经过第一步后已经确定了构造函数的参数，接下来的任务就是根据构造函数参数在所有的构造函数中锁定对应的构造函数，而且匹配的方法就是根据参数个数匹配，所以在匹配之前需要对构造函数按照public构造函数优先参数数量降序，非public构造函数参数数量降序。这样可以在遍历的情况下迅速判断排在后面的构造函数参数个数是否符合条件，由于在配置文件中并不是唯一限制使用参数位置索引的方式去创建，同样还支持指定参数名称进行设定参数值的情况，那么这种情况就需要首先确定构造函数中的参数名称，获取参数名称可以有两种方式，一种是通过注解的方式直接获取，另一种方式就是使用Spring提供的工具类ParameterNameDiscoverer来获取，构造函数、参数名称、参数类型、参数值、确定后就可以锁定构造函数以及转换对应的参数类型了。
- 根据确定的构造函数转换对应的参数类型，主要是使用Spring中提供的类型转换器或者用户自定义的类型转换器进行转换。
- 构造函数不确定性的验证，当然，有时候及时构造函数、参数名称、参数类型、参数值都确定后也不一定会直接锁定构造函数，不同构造函数的参数为父子关系，所以Spring在最后又做了一次验证。
- 根据实例化策略以及得到的构造函数及构造函数参数实例化Bean。

2. instantiateBean

   经历了带有参数的构造函数的实例构造，对于不带参数的构造函数的实例化过程中并没有什么实质性逻辑，带有参数的实例构造中，Spring把精力都放在了构造函数以及参数的匹配上，所以如果没有参数的话那将是非常简单的一件事，直接代用实例化策略进行实例化就可以了。

3. 实例化策略

   实例化过程中反复提到过实例化策略，那这又是做什么用的呢？其实，经过前面的分析，已经得到了足以实例化的所有相关信息，完全可以使用最简单的反射方法只讲反射来构造实例对象，但是Spring却没有这么做。

程序中，首先判断beanDefinition.getOverrides()为空也就是用户没有使用replace或者lookup的配置方法，呢么直接使用反射的方法，简单快捷，但是如果使用了这两特性，在直接使用反射的方式创建实例就不妥了，因为需要将这两个配置提供的功能切入进去，所以必须要使用动态代理的方式将包含两个特性所对应的逻辑的拦截增强器设置进去，这样才可以保证在调用方法的时候会被对应的拦截器增强，返回值为包含拦截器的代理实例。

5.7.2 记录创建bean的ObjectFactory

在doCreate函数中有这样一段代码：

```
boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCirularReference && isSingletonCurrentlyInCreation(beanName));
if(earlySingletonExposure)
// 为避免后期循环依赖，可以在bean初始化完成前创建实例的ObjectFactory加入工厂

// 对bean再一次依赖引用，主要应用SmartInstantiationAwareBeanPostaProcessor，其中AOP就是在这里将advice动态织入bean中，如没有则直接返回bean不做任何处理
```

```
6
7
```

这段代码不是很复杂，但是很多人不是太理解这段代码的作用，而且，这段代码仅从此函数中去理解也很难理解其中的含义，需要从全局的接到去思考Spring的依赖解决办法。

> earlySingletonExposure：从字面的意思就是提早曝光的单例。需要确认有哪些条件影响这个值。
> mbd.isSingleton()：没有太多可以解释的，此RootBeanDefinition代表是否是单例。
> this.allowCirularReference：是否允许循环依赖，在配置文件中并没有找到如何配置，但是在AbstractRefreshableApplicationContext中提供了设置函数，可以通过硬编码的方法进行设置或者通过自定义名称空间进行设置
> isSingletonCurrentlyInCreation(beanName)：该bean是否在创建中，在Spring中，会有个专门的属性默认为DefaultSingletonBeanRegistry的isSingletonsCurrentlyInCreation来记录bean的加载状态，在bean开始创建前会将beanName记录在属性中，在bean创建结束后会将beanName从属性中移除，不同scope的记录位置并不一样。

5.7.3 属性注入

在了解循环依赖的时候，populateBean这个函数的主要功能就是属性填充，那么究竟是如何实现填充的呢?

```
protected void populateBean(String beanName,AbstractBeanDefinition mbd,BeanWrapper bw)
// 没有属性填充则直接返回
// 给InstantiationAwareBeanPostProcessors最后一次机会在属性设置前改变bean，如：可以用来支持属性注入的类型
// 如果后处理器发出停止填充命令则终止后续操作
// 根据名称自动注入
// 根据类型自动注入
// 后处理器已经初始化
// 需要依赖检查
// 对所有需要依赖检查的属性进行后处理
// 将属性应用到bean中
}
```

在populateBean函数中提供了这样的处理流程。

- InstantiationAwareBeanPostProcessors处理器的postProcessAfterInstantiation函数的应用，此函数可以控制程序是否继续进行属性填充。
- 根据注入类型(byName/byType)，提取依赖的bean，并统一存到PropertyValues中。
- 应用InstantiationAwareBeanPostProcessors处理器的postProcessPropertyValues方法，对属性获取完毕填充前对属性再次处理，典型应用是RequiredAnnotationBeanPostProcessor类中对属性的验证。
- 将所有PropertyValues中的属性天充值BeanWrapper中。

1. autowireByName

   上面提到根据注入类型(byName/byType)，提取依赖的bean，并统一存入PropertyValues中，那么首先了解下byName功能是如何实现的。

```
prototype void autowireByName(String beanName,AbstractBeanDefinition mbd,BeanWrapper bw,MutablePropertyValues pvs)
// 寻找bw中需要依赖注入的属性
// 递归初始化相关的bean
// 注册依赖
}
```

2. autowireByType

   autowireByType与autowireByName对于我们理解和使用来说复杂程度都很相似，但是实现功能的复杂度却完全不一样。

```
protected void autowireByType(String beanName,AbstractBeanDefinition mbd,BeanWrapper bw,MutablePropertyValues pvs)
// 寻找bw中需要依赖注入的属性
// 探索指定属性的set方法
// 解析指定beanName的属性匹配的值，并把解析到的属性名称存储在autowireBeanName中，当属性存在多个封装bean时，将会找到所有的bean的类型将其注入
// 注册依赖
}
```

5.7.4 初始化bean

在配置bean是bean中有一个init-method属性，这个属性的作用是在bean实例化之前调用init-method指定的方法来根据业务进行相应的实例化。我们现在就已经进入到这个方法，首先看一下这个方法的执行位置，Spring程序已经执行过bean的实例化了，并且进行了属性填充，而就在这时将会调用用户设定的初始化方法。

```
protected Object initialzeBean(final String beanName,final Object bean,RootBeanDefinition mbd)
invokeAwaeMethod(beanName,bean)
对特殊bean的处理：Aware、BeanClassLoaderAware、BeanFactoryAware
// 应用后处理器
// 激活用户自定义的init方法
// 后处理器应用
}
```

1. 激活Aware方法

   在分析原理之前，我们先了解一下Aware的使用。Spring中提供一些Aware相关接口，比如：BeanFactoryAware、ApplicationContextAware、ResourceLoaderAware、ServletContextAware等，实现这些Aware接口的bean在初始化之后，可以取得一些相应的资源，例如实现BeanFactoryAware的bean在初始化后，Spring容器将会注入BeanFactory的实例，而实现ApplicationContextAware的bean，在bean被初始化后，将会被注入ApplicationContext的实例等。

2. 处理器的应用
   BeanPostProcess是Spring中开放式架构中一个必不可少的亮点，给用户充足的权限更改或者扩展Spring，而除了BeanPostProcessor还有其他的PostProcessor，当然大部分都是以此为基础的，继承自PostPostProcessor。BeanPostProcessor的使用位置就是这里，在调用客户自定义初始化方法之前以及调用自定义初始化方法后分别会调用BeanPostProcessor的postProcessBeforeInitialization和postProcessAfterInitialization方法，使客户可以根据自己的业务需求进行响应的处理。
3. 激活自定义的init方法
   客户定制的初始化方法除了使用配置init-method外，还用使用自定义bean实现InitializingBean接口，并在afterPropertiesSet中实现自己的初始化业务逻辑。

init-method和afterPropertiesSet都是在初始化bean时执行，执行顺序是afterPropertiesSet先执行，而init-method后执行。

在invokeInitMethods方法中就实现了这两个步骤的初始化方法调用。

5.7.5 注册DisponsableBean

Spring中不但提供了对于初始化方法的扩展入口，同样也提供了销毁方法的扩展入口，对于销毁方法的扩展，除了熟知的destory-method方法外，用户还可以注册后处理器DestructionAwareBeanPostProcessor来统一处理bean的销毁方法。

// 单例模式下注册需要销毁的bean，此方法中会处理实现DisposableBean的bean，并对所有的bean使用DestructionAwareBeanPostProcessor处理DisposableBean DestructionAwareBeanPostProcessors。
// 自定义scope的处理

容器的功能扩展

经过前几章的分析，对Spring中的容器功能有了简单的了解，在前面几章中一直以BeanFactory接口以及它的默认实现类XmlBeanFacotory为例进行分析，但是Spring中还提供了另一个接口ApplicationContext，用于扩展BeanFactory中现有的功能。

ApplicationContext和BeanFactory两者都是用于加载Bean的，但是相比之下，ApplicationContext提供更多的扩展功能，简单点说，ApplicationContext包含BeanFactory的所有功能。通常建议比BeanFactory优先，除非在一些限制的场合，比如字节长度对内存影响有很大的影响时(Applet)。绝大多数“典型的”企业应用和系统，ApplicationContext就是你需要的。

那么究竟ApplicationContext比BeanFactory多出了哪些功能呢？首先是使用不同类去加载配置文件在写法上的不同。

```
使用BeanFactory方法加载XML
BeanFactory bf = new XmlBeanFactory(new ClassPathResource(“beanFactoryTest.xml”))
使用ApplicationContext方法加载XML
ApplicationContext bf = new ClassPathXmlApplicationContext(“beanFactoryTest.xml”);
```

同样，还是以ClassPathXmlApplicationContext作为切入点，开始对整体功能进行分析，设置路径是必不可少的步骤，ClassPathXmlApplicationContext中可以配置文件路径数组的方法传入，ClassPathXmlApplicationContext可以对数组进行解析并加载。而对于解析及功能实现都在refresh()方法中实现。

6.1 设置配置路径

在ClassPathXmlApplicationContext中支持多个配置文件以数组方法进行传入：

```
public void setConfigLocations(@Nullable String... locations) {
        if (locations != null) {
            Assert.noNullElements(locations, "Config locations must not be null");
            this.configLocations = new String[locations.length];
            for (int i = 0; i < locations.length; i++) {
                this.configLocations[i] = resolvePath(locations[i]).trim();
            }
        }
        else {
            this.configLocations = null;
        }
    }
```

此函数主要用于解析给定的路径数组，当然，如果数组中包含特殊符号，那么在resolvePath中会搜索匹配的系统变量并替换。

6.2 扩展功能

设置路径之后，便可以根据路径做配置文件的解析以及各种功能的实现了。可以说refresh函数包含了几乎ApplicationContext中提供的全部功能，而且此函数中逻辑非常清晰明了，很容易分析对应的层次及逻辑。

```
public void refresh throw BeanException,IllegalStateException
// 准备刷新的上下文环境
// 初始化BeanFactory，并进行XML文件读取
// 对BeanFactory进行各种功能的填充
// 子类覆盖方法做额外的处理
// 激活各种BeanFactory处理器
// 注册拦截Bean创建的Bean拦截器，这里只是注册真正调用是在getBean时候
// 为上下文初始化Message源，即不同语言的消息体，国际化处理
// 初始化应用消息广播器，并放入“applicationEventMulticaster”bean中
// 留给子类来初始化其它的bean
// 在所有注册的bean中查找Listener bean，注册到消息广播器中
// 初始化剩下的单实例
// 完成刷新过程，通知声明周期处理器lifecycleProcessor刷新过程，同时发出ContextRefreshEvent通知别人
}
```

下面概括一下ClassPathXmlApplicationContext初始化的步骤，并解释它提供了哪些额外功能：

1. 初始化前的准备工作，例如对系统属性或者环境变量进行准备及验证，在某种情况下项目的使用需要读取某些系统变量，而这个变量的设置很可能会影响这系统的正确性，那么ClassPathXmlApplicationContext提供这个准备函数就显得非常必要了，它可以在Spring启动的时候提前对必须的变量进行存在性验证。
2. 初始化BeanFactory，并进行XML文件读取，之前有提到ClassPathXmlApplicationContext包含着BeanFactory所提供的一切特征，在这一步骤中将会复用BeanFactory中的配置文件读取解析及其他功能，这一步之后，ClassPathXmlApplicationContext实际上包含BeanFactory所提供的功能，也就是可以进行Bean的提取等基础操作。
3. 对BeanFactory进行各种功能填充。@Qualifer与@AutoWired这两个注解正式在这一步骤中增加的支持。
4. 子类覆盖方法做额外的处理，Spring之所以强大，为世人所推崇，除了它功能为大家提供了便利之外，还有一方面它的完美架构，开发式的架构让使用它的人很容易根据业务需求扩展已经存在的功能，这种开发式的设计在Spring中随处可见，例如在本例中提供了一个空的函数实现postProcessBeanFactory来方便程序员在业务上做进一步扩展。
5. 激活各种BeanFactory处理器。
6. 注册拦截bean创建的bean处理器，这只是注册，真正调用是在getBean时候。
7. 为上下文初始化Message源，即对不同语言的消息体进行国际化处理。
8. 初始化应用消息广播器，并放入“applicationEventMulticaster”bean中。
9. 留给子类来初始化其他bean。
10. 在所有的bean中查找listener bean，注册到消息广播器中。
11. 初始化剩下的单实例(非惰性的)。
12. 完成刷新过程，通知生命周期处理器lifecycleProcess刷新过程，同时发出ContextRefreshEvnet通知别人。

6.3 环境准备 prepareRefresh主要是做准备工作，例如对系统属性及环境变量的初始化及验证。

```
protected void prepareRefresh()
// 留给子类覆盖
// 验证需要的属性文件是否放入环境中
}
```

6.4 加载BeanFactory

obtainFreshBeanFactory方法从字面的理解是获取BeanFactory。之前有说过，ApplicationContext是对BeanFactory的功能上的扩展，不但包含BeanFactory的全部功能更是在其基础上增加了大量的扩展应用，那么obtainFreshBeanFactory正式实现BeanFactory的地方，也就是经过这个函数后ApplicationContext就已经拥有了BeanFactory的全部功能。

```
protected ConfigurableBeanFactory obtainFreshBeanFactory()
// 初始化BeanFactory，并进行XML文件读取，并得到BeanFactory记录在当前实体的属性中
// 返回当前实体的beanFactory属性
}
```

方法中将核心实现委托给了refreshBeanFactory：

```
protected final void refreshBeanFactory() throws BeanException
// 创建DefaultListBeanFactory
// 为了序列化指定id，如果需要的话，让这个BeanFactory从id反序列化到BeanFactory对象
// 定义beanFactory，设置相关属性，包括是否允许覆盖同名称的不同定义的对象以及循环依赖以及设置@Autowired和@Qualifer注解解析器QualifeAnnotationAutowireC
// 初始化DocumentReader，并进行XML文件读取及解析
}
```

6.4.2 加载BeanDefinition

在第一步中提到了将ClassPathXmlApplicationContext与XmlBeanFactory创建的对比，在实现配置文件的功能加载中除了第一步中已经初始化的DefaultListableBeanFactory外，还需要XmlBeanDefinitionReader来读取XML，那么在这个步骤中首先要做的就是初始化XmlBeanDefinitionReader。

```
protected void loadBeanDefinition(DefaultListableBeanFactory beanFactory) throws BeanException,IOException
// 为指定beanFactory创建XmlBeanDefinitionReader
// 对beanDefinitionReader进行环境变量的设置
// 对BeanDefinitionReader进行设置，可以覆盖
}
```

6.5 功能扩展

进入prepareBeanFactory前，Spring已经完成了对配置的解析，而ApplicationContext在功能上的扩展也由此展开。

```
protected void prepareBeanFactory(ConfigurableListableBeanFactory beanFactory)
// 设置beanFactory的classLoader为当前context的classLoader
// 设置beanFactory的表达式语言处理器，Spring3增加了语言表达式的支持，默认可以使用“#{bean.xxx}”的形式来调用相关属性值。
// 为beanFactory增加一个默认properEditor，这个主要是对bean的属性等设置管理的一个工具
// 添加BeanPostProcessor
// 设置几个忽略自动装配的接口
// 设置几个自动装配的特殊规则
// 增加对AspectJ的支持
// 添加默认的系统环境bean
}
```

上面的函数主要进行了几个方面的扩展：

- 增加SPEL语言的支持。
- 增加对属性编辑器的支持。
- 增加对一些内置类，比如EnvironmentAware、MessageSourceAware的信息注入。

- 设置了依赖功能可忽略接口。
- 注册一些固定依赖的属性。
- 增加AspectJ的支持
- 将相关环境变量及属性注册以单例模式注册

6.6 BeanFactory的后处理

BeanFactory作为容器功能的基础，用于存放已经加载的bean，为了保证程序上的高扩展性，Spring针对BeanFactory做了大量扩展，比如熟知的PostProcessor等都是在这里实现的。

6.6.1 激活注册的BeanFactoryPostProcessor

正式开始介绍之前先来了解一下BeanFactoryPostProcessor的用法。
BeanFactoryPostProcessor接口和BeanPostProcessor类似，可以对bean的定义（配置元数据）进行处理。也就是说，Spring IoC容器允许BeanFactoryPostProcessor在容器实际实例化任何其他的bean之前读取配置元数据，并有可能修改它。如果你愿意，你可以配置多个BeanFactoryPostProcessor。你还可以通过设置“order”属性来控制BeanFactoryPostProcessor的执行次序（仅当BeanFactoryPostProcessor实现了Ordered接口时你才可以设置此属性，因此在实现BeanFactoryPostProcessor时，就应当考虑实现Ordered接口）。

AOP

Spring 2.0采用@AspectJ注解对POJO进行标注，从而定义一个包含切点信息和增强横切面逻辑的切面。Spring 2.0可以将这个切面植入到匹配的目标Bean中。@AspectJ注解使用AspectJ切点表达式语法进行切点定义，可以通过切点函数、运算符、通配符等高级功能进行切点定义，拥有强大的连接点描述能力。

7.1 动态AOP使用示例

```
@Aspect
public class AspectTest {
    @Pointcut("execution(* *.test(..))")
    public void test() {

    }

    @Before("test()")
    public void beforeTest() {
        System.out.println("beforeTest");
    }

    @After("test()")
    public void afterTest() {
        System.out.println("afterTest");
```

```
    }

    @Around("test()")
    public Object aroundTest(ProceedingJoinPoint p) {
        System.out.println("before1");
        Object o = null;
        try {
            o = p.proceed();
        } catch (Throwable e) {
            e.printStackTrace();
        }
        System.out.println("after1");
        return o;
    }
}

public class TestBean {
    private String testStr = "testStr";

    public String getTestStr() {
        return testStr;
    }

    public void setTestStr(String testStr) {
        this.testStr = testStr;
    }
    public void test(){
        System.out.println("test");
    }
}
public class MyApplication {
        public static void main(String[] args) {
            ApplicationContext applicationContext = new ClassPathXmlApplicationContext(("applicationContext.xml"));
            TestBean testBean = (TestBean) applicationContext.getBean("testBean");
            testBean.test();
        }
}
```

那么，Spring究竟是如何实现AOP的呢，首先我们知道，Spring是否支持注解的AOP由一个配置文件控制的，也就是<aop:aspectj-autoproxy />，当配置文件中声明了这句配置的时候，Spring就会支持注解的AOP，那么分析就是从这句注解开始的。

7.2 动态AOP自定义标签

之前讲过Spring中的自定义注解，如果声明了自定义的注解，那么就一定会在程序中的某个地方注册了对应的解析器。搜索整个代码后发现了在AopNamespaceHandler中对应这一段函数

```
public void init() {
        // In 2.0 XSD as well as in 2.5+ XSDs
        registerBeanDefinitionParser("config", new ConfigBeanDefinitionParser());
        registerBeanDefinitionParser("aspectj-autoproxy", new AspectJAutoProxyBeanDefinitionParser());
        registerBeanDefinitionDecorator("scoped-proxy", new ScopedProxyBeanDefinitionDecorator());

        // Only in 2.0 XSD: moved to context namespace in 2.5+
        registerBeanDefinitionParser("spring-configured", new SpringConfiguredBeanDefinitionParser());
    }

```

在解析文件的时候，一旦遇到aspectj-autoproxy注解时就会使用解析器AspectJAutoProxyBeanDefinitionParser进行解析，所以来看看AspectJAutoProxyBeanDefinitionParser 的内部实现。

7.2.1 注册AnnotationAwareAspectJAutoProxyCreator

所有解析器，因为是对BeanDefinitionParser接口的统一实现，入口都是从parse函数开始的AspectJAutoProxyBeanDefinitionParser的parse函数如下

```
// 注册AnnotationAwareAspectJAutoProxyCreator
AopNamespaceUtils.registerAspectJAnnotationAwareAspectJAutoProxyCreatorIfNecessary(parserContext,element);
// 对于注解中子类的处理
extendBeanDefinition(element, parserContext);
```

其中registerAspectJAnnotationAutoProxyCreatorIfNecessary的逻辑实现如下：

```
// 注册或升级AutoProxyCreator定义beanName为org.Springframework.aop.config.internalAutoProxyCreator的beanDefinition
AopConfig.registerAspectJAnnotationAutoProxyCreatorIfNecessary(...);
// 对于proxy-target-class以及expose-proxy属性的处理
useClassProxyIfNecessary(...);
// 注册组件并通知，便于监听器做进一步处理，其中beanDefinition的className为AnnotationAwareAspectJAutoProxyCreator
registerComponentIfNecessary(beanDefinition,parserContext);

```

registerAspectJAnnotationAutoProxyCreatorIfNecessary方法中主要完成了3件事，基本上每一行就是一个完整的逻辑。

1. 注册或者升级AnnotationAwareAspectJAutoProxyCreator，对于AOP的实现，基本上都是靠AnnotationAwareAspectJAutoProxyCreator去完成的，它可以根据@Point注解定义的切面来自动代理相匹配的bean，但是为了配置简便，Spring使用了自定义配置来帮助自动注册AnnotationAwareAspectJAutoProxyCreator，其注册过程就是在这里实现的。
   registerOrEscalateApcAsRequired(…) //
   如果已经存在了自动代理创建器且存在的自动代理创建器与现在的不一致那么需要根据优先级来判断到底使用哪个 //
   如果已存在自动代理创建器并且与将要创建的一致，那么无需再次创建
   以上的逻辑实现了注册AnnotationAwareAspectJAutoProxyCreator类的功能，同时在这里还涉及到一个优先级的问题，如果已经存在了自动代理创建器，而且存在的自动代理创建器与现在的不一致，那么需要根据优先级来判断到底需要使用哪个。
2. 处理proxy-target-class以及expose-proxy属性，
   useClassProxyIfNecessary实现了proxy-target-class属性以及expose-proxy属性的处理
   // 对于proxy-target-class属性的处理
   AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(…) //
   对于expose-proxy属性的处理
   AopConfigUtils.forceAutoProxyCreatorToExposeProxying(…) //
   强制使用的过程其实是一个属性设置的过程

> JDK动态代理和CGLIB动态代理在实际的使用过程中会有一些细微差别：
> JDK动态代理：其代理对象必须是某个接口的实现，它通过在运行期间创建一个接口的实现类来完成对目标对象的代理。
> CGLIB代理：实现原理类似于JDK动态代理，只是它在运行期间生成的代理对象是针对目标类扩展的子类。CGLIB是高效的代码生成包，底层依靠ASM操作字节码实现的，性能比JDK强。

7.3 创建AOP代理

以上讲解了通过自动配置完成了对AnnotationAwareAspectJAutoProxyCreator类型的自动注册，那么这个类到底做了什么工作来完成AOP的操作呢?

从AnnotationAwareAspectJAutoProxyCreator类的层次结构可知，AnnotationAwareAspectJAutoProxyCreator实现了BeanPostProcessor接口，而实现BeanPostProcessor后，当Spring加载这个Bean时会在实例化前调用其postProcessAfterInitialization方法，而我们对AOP逻辑的分析也就此开始。

在父类AbstractAutoProxyCreator的postProcessAfterInitialization中的代码中：

```
// 根据给定的bean的class和name构建出个key，格式为：beanClassName_beanName
// 如果它适合被代理，则需要封装指定bean
return wrapIfNecessary(bean,beanName,cacheKey);
// wrapIfNecessary(bean,beanName,cacheKey)
// 如果已经处理过，则直接返回bean实例
```

```
// 如果无需增强则直接返回bean实例
// 判断给定的bean类是否代表一个基础设施类，如果是基础设施类则不应该被代理，或者配置了指定bean不需要代理
// 如果存在增强方法则创建代理
Object[] specifcInterceptors = getAdviceAndAdcisorForBean(bean.getClass());
// 如果获取到了增强则需要针对增强创建代理
Object proxy = createProxy(bean.getClass(),beanName,specifcInterceptors ,new SingletonTargetSource(bean));
```

7.5 创建AOP静态代理

AOP静态代理主要是在虚拟机启动时通过改变目标对象字节码的方式来完成对目标对象的增强，它与动态代理相比具有更高的效率，因为在动态代理调用过程中，还需要一个动态创建代理类并代理目标对象的步骤，而静态代理则是在启动时完成了字节码增强，当系统再次调用目标类时与调用正常类并无差别，所以效率上会相对高一些。

7.5.1 Instrumentation使用

Java在1.5引入java.lang.Instrument，你可以由此实现一个Java agent，通过此agent来修改类的字节码来改变一个类，通过Java Instrument实现一个简单的profiler，当然Instrument并不限于profiler，Instrument可以做很多事情，它类似于一个更低级、更松耦合的AOP，从底层改变一个类的行为。

以计算一个方法的运行时间为例，在方法的开头和结尾不得不在所有需要监控的方法的开头和结尾写入重复的代码，而Java Instrument使得这一切更干净。

```
写ClassFileTransformer类，实现ClassFileTransformer，编写修改字节码方法；
编写agent类，JVM启动时在应用加载前会调用PerfMonAgent.permain，然后PerfMonAgentPerfMonAgent.permain实例化一个定制的ClassFileTransformer类，即PerMonXformer并通过inst.addTransformer(trans)把PerfMonXformer的实例加入Instrumentation实例(有JVM传入)，这就使得应用中的类加载时，PerfMonXformer.transform都会被调用，你在此方法中就可以改变加载的类，还可以使用JBoss的Javaassist轻易的改变类的字节码；
打包agent，引入相关的Jar包；
打包应用。
```

由执行结果可以看出，执行顺序以及通过改变字节码加入监控代码确实生效了，而且通过Instrument实现agent使得监控代码和应用代码完全隔离了。

SpringBoot体系原理

14.2第一个Starter

Spring Boot之所以流行，是因为Spring starter模式的提出。Spring starter的出现，可以让模块开发更加独立化，相互间依赖更加松散以及可以更加方便地集成。从前言中介绍的例子来看，正是由于在pom文件中引入了下述代码：

```
<dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-web</artifactId>
</dependency>
```

新建maven工程，引入pom：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.study</groupId>
    <artifactId>study-client-starter</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>1.8</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-autoconfigure</artifactId>
        </dependency>
    </dependencies>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-dependencies</artifactId>
                <version>2.0.1.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
</project>

```

新建接口并实现：

```java
public interface HelloService {
     String sayHello();
}
```

```
@Component
public class HelloServiceImpl implements HelloService {
    @Override
    public String sayHello() {
        return "hello";
    }
}

@Configuration
@ComponentScan({"com.study.module"})
public class HelloServiceAutoConfiguration {
}
```

14.3探索SpringApplication启动Spring

我们找到主函数入口SpringBootDemo1Application, 发现这个入口的启动还是比较奇怪的，这也是Spring Boot启动的必要做法，那么，这也可以作为我们分析Spring Boot的入口：

```
public ConfigurableApplicationContext run(String... args) {
...
 context = this.createApplicationContext();
            exceptionReporters = this.getSpringFactoriesInstances(SpringBootExceptionReporter.class, new Class[]{ConfigurableApplicati
            this.prepareContext(context, environment, listeners, applicationArguments, printedBanner);
            this.refreshContext(context);
....
}
```

SpringContext创建

```
protected ConfigurableApplicationContext createApplicationContext() {
        Class<?> contextClass = this.applicationContextClass;
        if (contextClass == null) {
            try {
                switch(this.webApplicationType) {
                case SERVLET:
                    contextClass = Class.forName("org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplica
```

```
                    break;
                case REACTIVE:
                    contextClass = Class.forName("org.springframework.boot.web.reactive.context.AnnotationConfigReactiveWebServerAppli
                    break;
                default:
                    contextClass = Class.forName("org.springframework.context.annotation.AnnotationConfigApplicationContext");
                }
            } catch (ClassNotFoundException var3) {
                throw new IllegalStateException("Unable create a default ApplicationContext, please specify an ApplicationContextClass
            }
        }

        return (ConfigurableApplicationContext)BeanUtils.instantiateClass(contextClass);
    }
```

bean的加载

```
protected void load(ApplicationContext context, Object[] sources) {
        ...
        loader.load();
    }
```

相信当读者看到BeanDefinitionLoader这个类的时候基本上就已经知道后续的逻辑了， bean的加载作为本书中最核心的部分早在第1章就已经开始分析了。

Spring扩展属性的加载

```
    this.refreshContext(context);
```

对于Spring的扩展属性加载则更为简单，因为这些都是Spring本身原有的东西，Spring Boot仅仅是使用refresh激活下而已，如果读者想回顾refresh的详细逻辑，可以回到第5章进一步查看。

14.4Starter自动化配置原理

我们已经知道了Spring Boot如何启动Spring, 但是目前为止我们并没有揭开Spring Boot的面纱，究竟Starter是如何生效的呢？这些逻辑现在看来只能体现在注解SpringBootApplication 本身了。

```java
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(
    excludeFilters = {@Filter(
    type = FilterType.CUSTOM,
    classes = {TypeExcludeFilter.class}
), @Filter(
    type = FilterType.CUSTOM,
    classes = {AutoConfigurationExcludeFilter.class}
)}
)
public @interface SpringBootApplication {
}
```

这其中我们更关注SpringBootApplication上的注解内容，因为注解具有传递性， EnableAutoConfiguration是个非常特别的注解，它是Spring Boot的全局开关，如果把这个注解去掉，则一切Starter都会失效，这就是约定大于配置的潜规则，那么，Spring Boot的核心很可能就藏在这个注解里面：

```java
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import({AutoConfigurationImportSelector.class})
public @interface EnableAutoConfiguration {
    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";

    Class<?>[] exclude() default {};

    String[] excludeName() default {};
}

```

EnableAutoConfigurationImportSelector作为Starter自动化导入的关键选项终于浮现出来，那么Spring是怎么识别并让这个注解起作用的呢？我们看到这个类中只有一个方法，那么只要看一看到底是哪个方法调用了它，就可以顺藤摸瓜找到最终的调用点。

Spring.factories的加载

顺着思路反向查找，看一看究竟是谁在哪里调用了isEnabled函数，强大的编译器很容器帮我们定位到了AutoConfigurationImportSelector类的方法

在上面的函数中，有一个是我们比较关注的getCandidateConfigurations函数：

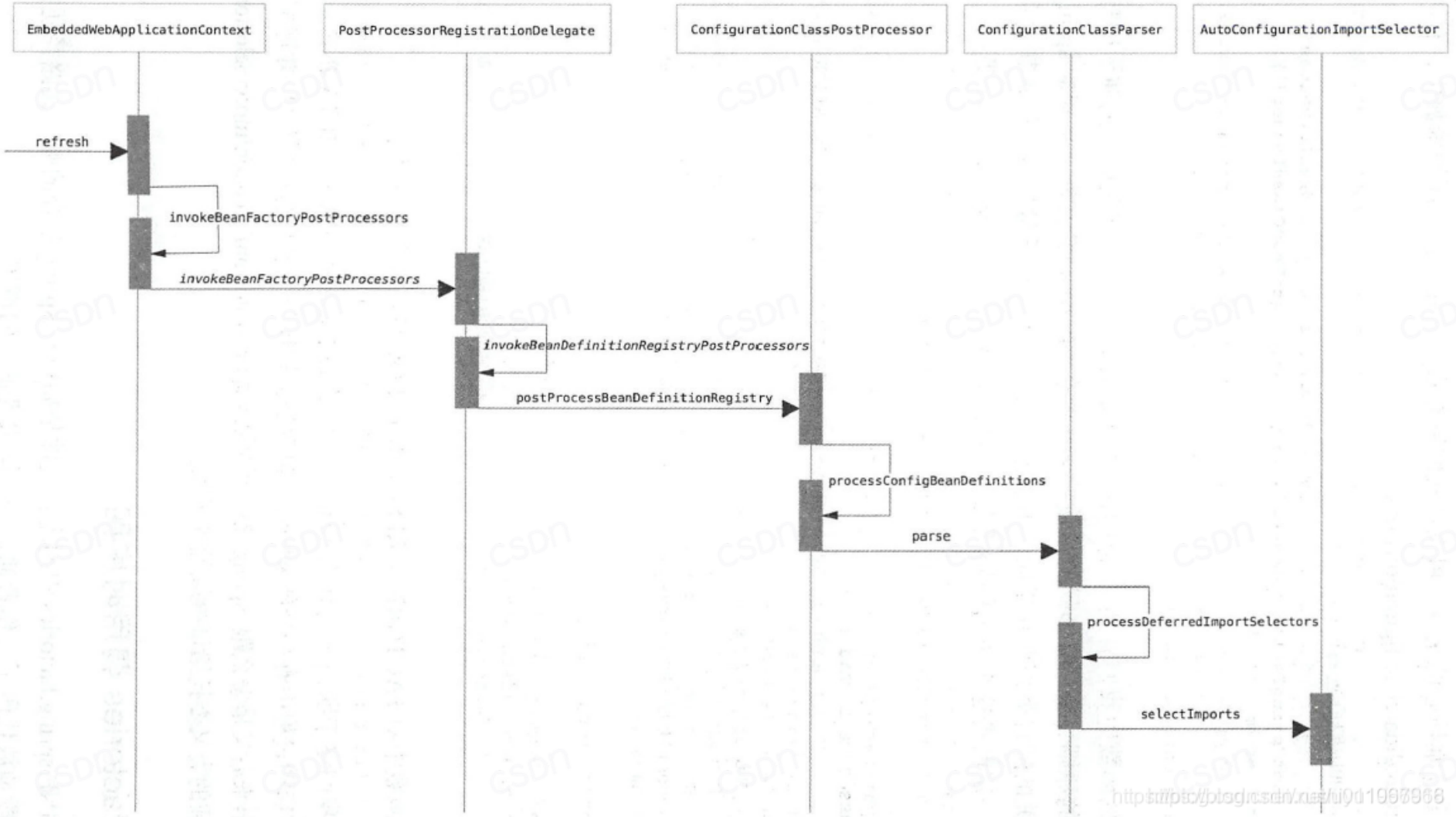
```
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {
        List<String> configurations = SpringFactoriesLoader.loadFactoryNames(this.getSpringFactoriesLoaderFactoryClass(), this.getBean
        Assert.notEmpty(configurations, "No auto configuration classes found in META-INF/spring.factories. If you are using a custom p
        return configurations;
    }
```

从上面的函数中我们看到了META-INF/Spring.factories, 在我们之前演示的环节，按照约定大于配置的原则，Starter如果要生效则必须要在META-INF文件下下建立Spring.factories文件，并把相关的配置类声明在里面，虽然这仅仅是一个报错异常提示，但是其实我们已经可以推断出来这一定就是这个逻辑的处理之处，继续进入SpringFactoriesLoader类：

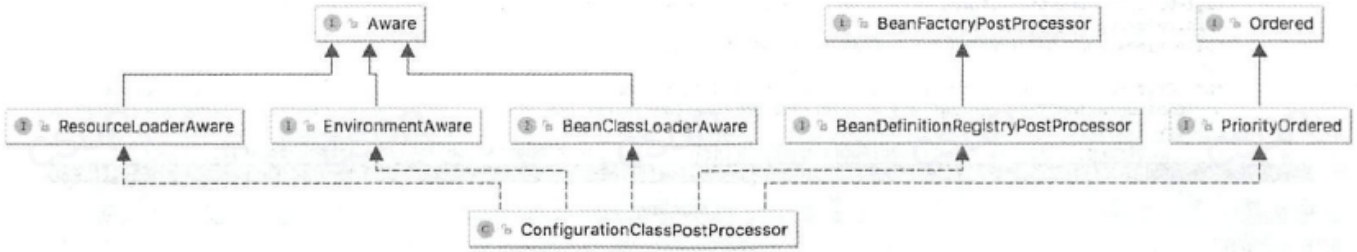
```
private static Map<String, List<String>> loadSpringFactories(@Nullable ClassLoader classLoader) {
        MultiValueMap<String, String> result = (MultiValueMap)cache.get(classLoader);
        if (result != null) {
            return result;
        } else {
            try {
                Enumeration<URL> urls = classLoader != null ? classLoader.getResources("META-INF/spring.factories") : ClassLoader.getS
}
```

至此，我们终于明白了为什么Starter的生效必须要依赖于配置META-INF /Spring. factories 文件，因为在启动过程中有一个硬编码的逻辑就是会扫描各个包中的对应文件，并把配置捞取出来。

factories调用时序图



图中梳理了从EmbeddedWebApplicationContext到AutoConfigurationimportSelector的调用链路，当然这个链路还有非常多的额外分支被忽略。不过至少从上图中我们可以很清晰地看到 AutoConfigurationimportSelector与Spring的整合过程，在这个调用链中最核心的就是Spring Boot使用了Spring提供的BeanDefinitionRegistry FostProcessor扩展点并实现了ConfigurationClassPostProcessor 类，从而实现了Spring之上的一系列逻辑扩展，让我们看一下ConfigurationClassPostProcessor的继承关系，如图所示：



当然Spring还提供了非常多不同阶段的扩展点，可以通过前几章的内容获取详细的扩展点以及实现原理。

配置类的解析