链滴

Dubbo 系列笔记之 XML 配置文件解析流程

作者：wangning1018

原文链接：https://ld246.com/article/1568868650705

来源网站：链滴

简单叨叨一下Dubbo是如何自定义标签给spring承载bean的。

Spring通过XML解析程序将其解析为DOM树，通过NamespaceHandler指定对应的Namespace的BeaDefinitionParser将其转换成BeanDefinition。再通过Spring自身的功能对BeanDefinition实例化对象。ubbo做的只是实现了NamespaceHandler解析成BeanDefinition。

好了，总结起来就这么简单，下面我们具体来看一下。

一、约束文件schema

下面是一个标准的schema文件头的格式point_down

采取基于 Schema 配置格式，文件头的声明会复杂一些，先看一个简单的示例：

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

xsi:schemaLocation=
    "http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
```

⑦默认命名空间

②xsi 标准命名空间，用于指定自定义命名空间的 Schema 文件

③-1 自定义命名空间，aop 是该命名空间的简称

③-2 名称空间全称，必须在 xsi 命名空间为其指定空间对应的 Schema 文件，参见④

④为每个命名空间指定具体的 Schema 文件

原文链接：Dubbo 系列笔记之 XML 配置文件解析流程

首先自定义的标签会有一些约束规范，比如我自定义的有哪几种标签，标签里面有哪些属性等等，在ML中每个命名空间都会有一个.xsd的约束文件。

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!--
  Licensed to the Apache Software Foundation (ASF) under one or more
  contributor license agreements.  See the NOTICE file distributed with
  this work for additional information regarding copyright ownership.
  The ASF licenses this file to You under the Apache License, Version 2.0
  (the "License"); you may not use this file except in compliance with
  the License.  You may obtain a copy of the License at

      http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
  -->
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
       http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd">

    <!-- provider's application name, used for tracing dependency relationship -->
    <dubbo:application name="demo-provider"/>

    <!-- use multicast registry center to export service -->
    <dubbo:registry address="zookeeper://127.0.0.1:2181"/>

    <!-- use dubbo protocol to export service on port 20880 -->
    <dubbo:protocol name="dubbo" port="20880"/>

    <!-- service implementation, as same as regular local bean -->
    <bean id="demoService" class="com.alibaba.dubbo.demo.provider.DemoServiceImpl"/>

    <!-- declare the service interface to be exported -->
    <dubbo:service interface="com.alibaba.dubbo.demo.DemoService" ref="demoService"/>

</beans>
```

一个约束文件.xsd长得像下面这样point_down

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:beans="http://www.springframework.org/schema/beans"
            xmlns:tool="http://www.springframework.org/schema/tool"
            xmlns="http://dubbo.apache.org/schema/dubbo"
            targetNamespace="http://dubbo.apache.org/schema/dubbo">

    <xsd:import namespace="http://www.w3.org/XML/1998/namespace"/>
    <xsd:import namespace="http://www.springframework.org/schema/beans"/>
    <xsd:import namespace="http://www.springframework.org/schema/tool"/>

    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[ Namespace support for the dubbo services provided by dubbo framework. ]]></xsd:documentation>
    </xsd:annotation>

    <xsd:complexType name="abstractMethodType">
        <xsd:attribute name="timeout" type="xsd:string" use="optional" default="0">
            <xsd:annotation>
                <xsd:documentation><![CDATA[ The method invoke timeout. ]]></xsd:documentation>
            </xsd:annotation>
        </xsd:attribute>
        <xsd:attribute name="retries" type="xsd:string" use="optional">
            <xsd:annotation>
                <xsd:documentation><![CDATA[ The method retry times. ]]></xsd:documentation>
            </xsd:annotation>
        </xsd:attribute>
        <xsd:attribute name="actives" type="xsd:string" use="optional">
            <xsd:annotation>
                <xsd:documentation><![CDATA[ The max active requests. ]]></xsd:documentation>
            </xsd:annotation>
        </xsd:attribute>
        <xsd:attribute name="connections" type="xsd:string" use="optional">
            <xsd:annotation>
                <xsd:documentation>
                    <![CDATA[ The exclusive connections. default share one connection. ]]></xsd:documentation>
            </xsd:annotation>
        </xsd:attribute>
        <xsd:attribute name="loadbalance" type="xsd:string" use="optional">
            <xsd:annotation>
                <xsd:documentation><![CDATA[ The method load balance. ]]></xsd:documentation>
            </xsd:annotation>
        </xsd:attribute>
        <xsd:attribute name="async" type="xsd:string" use="optional" default="false">
            <xsd:annotation>
                <xsd:documentation><![CDATA[ The method does async. ]]></xsd:documentation>
            </xsd:annotation>
        </xsd:attribute>
        <xsd:attribute name="sent" type="xsd:string" use="optional">
            <xsd:annotation>
                <xsd:documentation><![CDATA[ The async method return await message sent ]]></xsd:documentation>
            </xsd:annotation>
        </xsd:attribute>
```

原文链接：Dubbo 系列笔记之 XML 配置文件解析流程

里面限制自定义的标签里面有哪些属性，属性的类型是什么啊这种。

二、spring.handlers和spring.schemas

当spring解析xml时遇到自定义的标签时，spring会加载spring.handlers和spring.schemas这两个件，两个文件长下面这样point_down

 spring.schemas：里面指定了该标签的约束文件本地路径，在解析XML文件时将XSD重定向到本文件，避免在解析XML文件时需要上网下载XSD文件。通过实现org.xml.sax.EntityResolver接口来现该功能。

```
http\://dubbo.apache.org/schema/dubbo/dubbo.xsd=META-INF/dubbo.xsd
http\://code.alibabatech.com/schema/dubbo/dubbo.xsd=META-INF/compat/dubbo.xsd
```

 spring.handlers：里面指定了由那个handler去处理这些自定义的标签，实现一个handler需要实现org.springframework.beans.factory.xml.NamespaceHandler接口，或使用org.springframework.beans.factory.xml.NamespaceHandlerSupport的子类。

```
http\://dubbo.apache.org/schema/dubbo=com.alibaba.dubbo.config.spring.schema.DubboNamespaceHandler
http\://code.alibabatech.com/schema/dubbo=com.alibaba.dubbo.config.spring.schema.DubboNamespaceHandler
```

三、DubboNamespaceHandler

在spring加载完spring.handlers后就知道要通过DubboNamespaceHandler去解析dubbo:xxx这种dubbo的标签。

下面我们来看一下，它长这个样子point_down

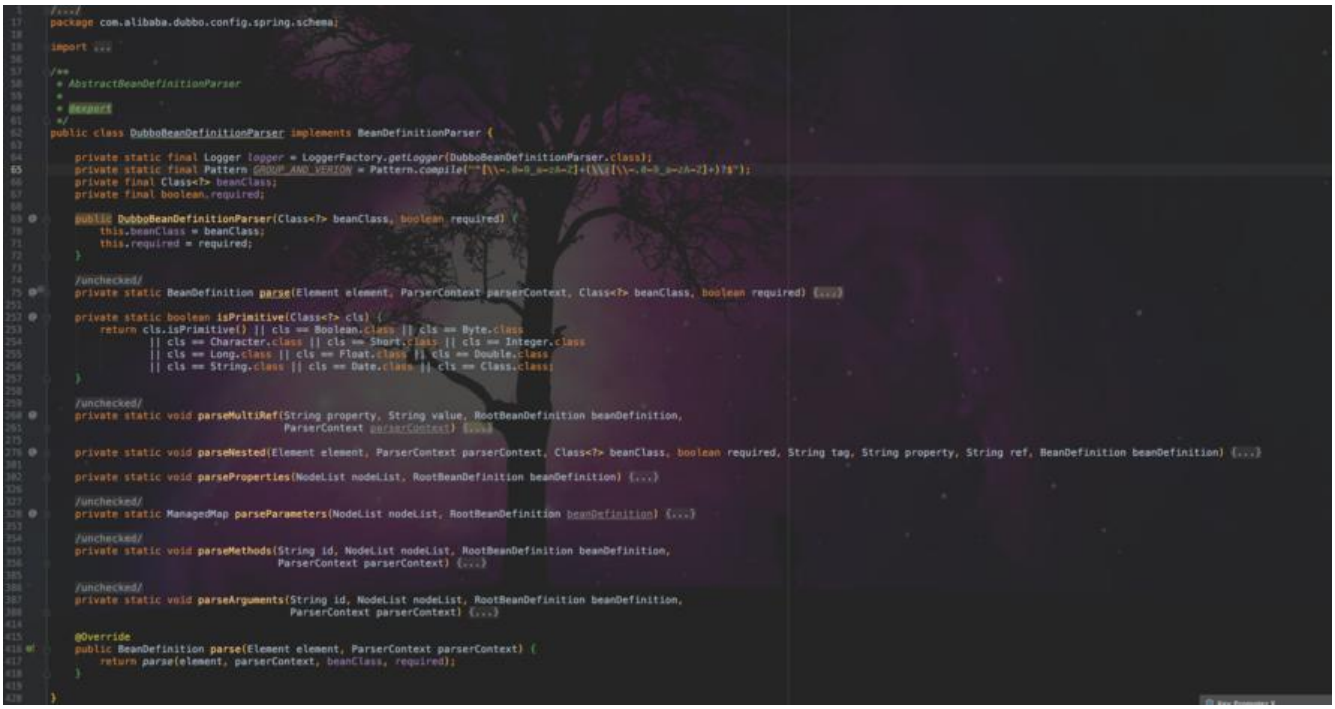
```
/.../
package com.alibaba.dubbo.config.spring.schema;

import ...

/**
 * DubboNamespaceHandler
 *
 * @export
 */
public class DubboNamespaceHandler extends NamespaceHandlerSupport {

    static {
        Version.checkDuplicate(DubboNamespaceHandler.class);
    }

    @Override
    public void init() {
        registerBeanDefinitionParser( elementName: "application", new DubboBeanDefinitionParser(ApplicationConfig.class, required: true));
        registerBeanDefinitionParser( elementName: "module", new DubboBeanDefinitionParser(ModuleConfig.class, required: true));
        registerBeanDefinitionParser( elementName: "registry", new DubboBeanDefinitionParser(RegistryConfig.class, required: true));
        registerBeanDefinitionParser( elementName: "monitor", new DubboBeanDefinitionParser(MonitorConfig.class, required: true));
        registerBeanDefinitionParser( elementName: "provider", new DubboBeanDefinitionParser(ProviderConfig.class, required: true));
        registerBeanDefinitionParser( elementName: "consumer", new DubboBeanDefinitionParser(ConsumerConfig.class, required: true));
        registerBeanDefinitionParser( elementName: "protocol", new DubboBeanDefinitionParser(ProtocolConfig.class, required: true));
        registerBeanDefinitionParser( elementName: "service", new DubboBeanDefinitionParser(ServiceBean.class, required: true));
        registerBeanDefinitionParser( elementName: "reference", new DubboBeanDefinitionParser(ReferenceBean.class, required: false));
        registerBeanDefinitionParser( elementName: "annotation", new AnnotationBeanDefinitionParser());
    }
}
```

通过上面代码我们可以知道，解析标签的工作并不是namespaceHandler去做的，它做的只是为每个签注册BeanDefinitionParser，告诉spring哪个BeanDefinitionParser去真正处理这个标签。

四、DubboBeanDefinitionParser

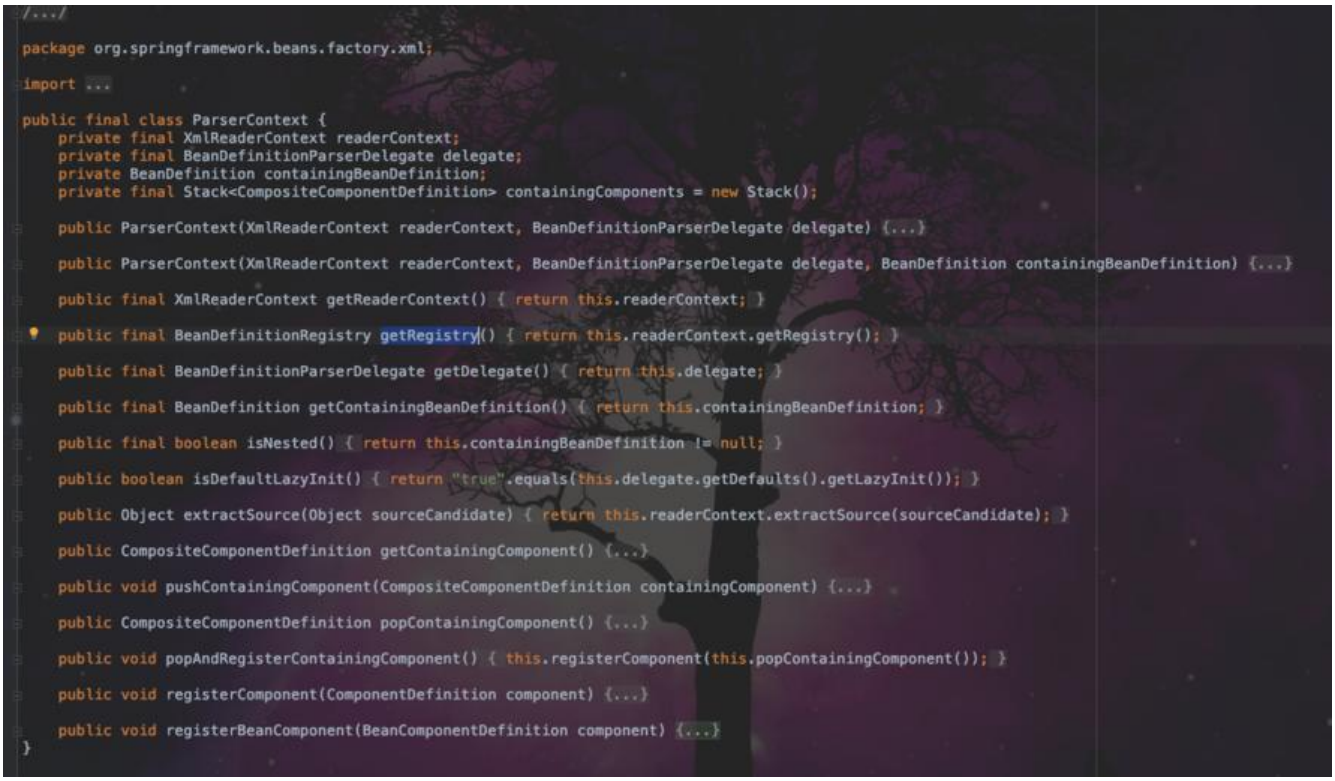
下面我们简单来看一下DubboBeanDefinitionParser长什么样，嗯，大概就下面这个样子吧

oint_down



在解析标签时spring会调用BeanDefinitionParser的parse()方法去生成一个BeanDefinition。

我们注意到，parse()方法有两个参数Element element和ParserContext parserContext，element xml解析器在解析完xml标签后将其组装成一个这样的对象，而第二个参数parserContext，我们通过的getRegistry()方法获取BeanDefinitionRegistry对象。他长下面这个样
oint_down

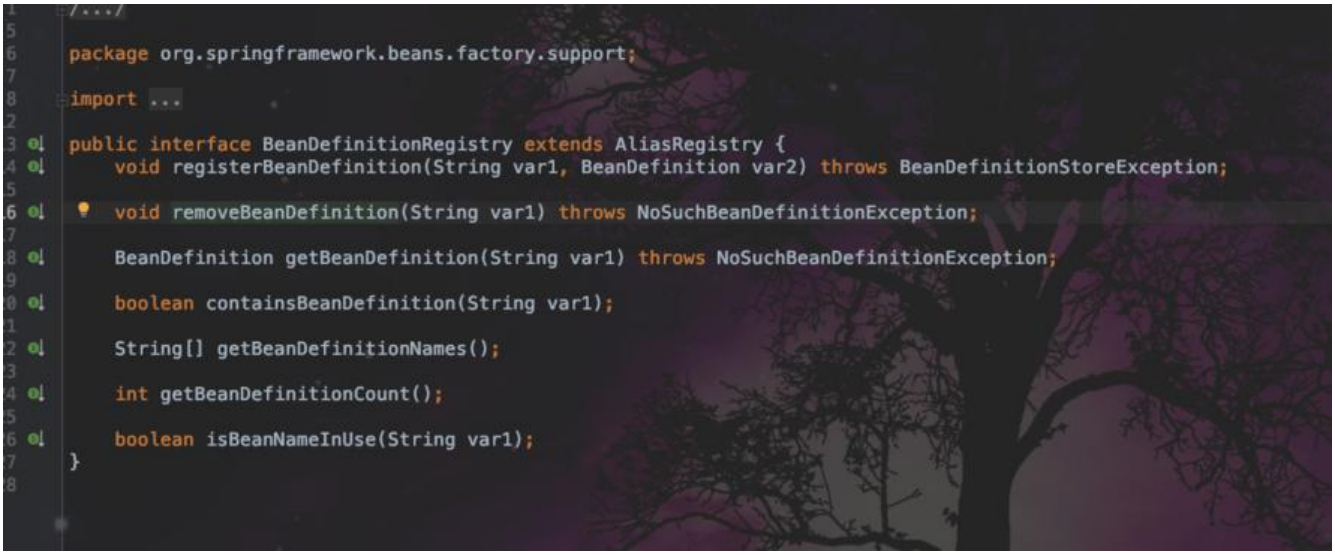


说到这里，那么我们解析配置的初衷是什么呢?

没错，我们为了把配置解析成bean去交给spring托管。

而BeanDefinitionRegistry的作用主要是向spring注册表中注册 BeanDefinition 实例，通过调用其registerBeanDefinition()方法完成注册的过程。

简单看一下BeanDefinitionRegistry长什么样子point
down



可以看到BeanDefinitionRegistry是一个接口，它定义了一些注册BeanDefinition的一些必要方法。

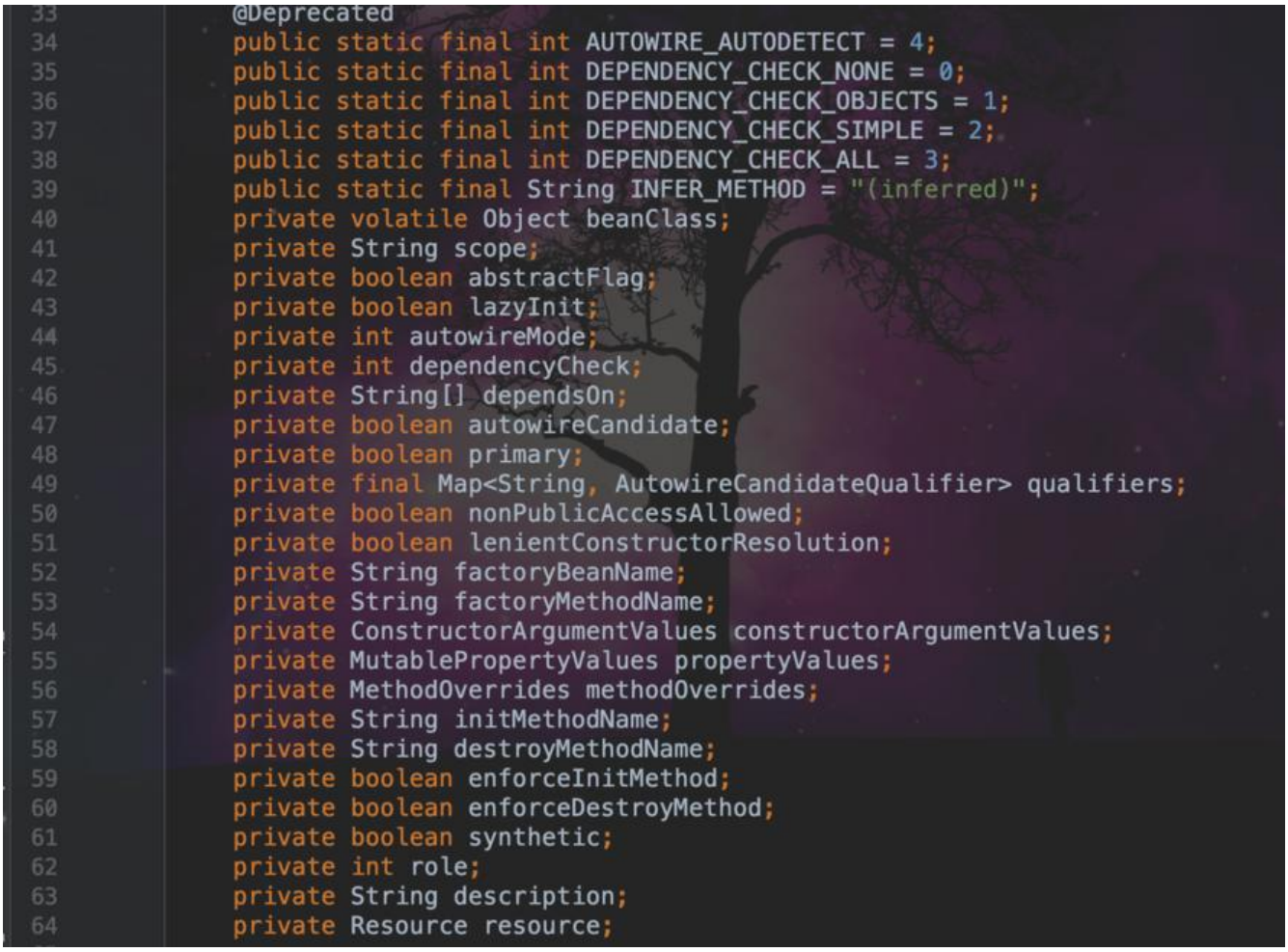
那么具体BeanDefinitionRegistry是如何注册bean的呢?

我们看一下它的registerBeanDefinition(String var1, BeanDefinition var2)，第一个参数是bean的i，第二个就是BeanDefinition，BeanDefinition描述了一个bean的画像，他是基础的bean定义接口由他衍生出AbstractBeanDefinition和RootBeanDefinition。

五、BeanDefinition

- AbstractBeanDefinition

他长得挺长的，主要是在BeanDefinition的基础上定义了一些属性，基本囊括了Bean实例化需要的有信息。如下，point_down



```
@Deprecated
public static final int AUTOWIRE_AUTODETECT = 4;
public static final int DEPENDENCY_CHECK_NONE = 0;
public static final int DEPENDENCY_CHECK_OBJECTS = 1;
public static final int DEPENDENCY_CHECK_SIMPLE = 2;
public static final int DEPENDENCY_CHECK_ALL = 3;
public static final String INFER_METHOD = "(inferred)";
private volatile Object beanClass;
private String scope;
private boolean abstractFlag;
private boolean lazyInit;
private int autowireMode;
private int dependencyCheck;
private String[] dependsOn;
private boolean autowireCandidate;
private boolean primary;
private final Map<String, AutowireCandidateQualifier> qualifiers;
private boolean nonPublicAccessAllowed;
private boolean lenientConstructorResolution;
private String factoryBeanName;
private String factoryMethodName;
private ConstructorArgumentValues constructorArgumentValues;
private MutablePropertyValues propertyValues;
private MethodOverrides methodOverrides;
private String initMethodName;
private String destroyMethodName;
private boolean enforceInitMethod;
private boolean enforceDestroyMethod;
private boolean synthetic;
private int role;
private String description;
private Resource resource;
```

总体来看这个抽象类长了这些东西：

1. Bean的描述信息（例如是否是抽象类、是否单例）
2. depends-on属性（String类型，不是Class类型）
3. 自动装配的相关信息
4. init函数、destroy函数的名字（String类型）
5. 工厂方法名、工厂类名（String类型，不是Class类型）
6. 构造函数形参的值
7. 被IOC容器覆盖的方法
8. Bean的属性以及对应的值（在初始化后会进行填充）

● RootBeanDefinition

从spring2.5开始，spring一开始都是使用GenericBeanDefinition类保存Bean的相关信息，在需要
，在将其转换为其他的BeanDefinition类型。

这里两个BeanDefinition可以说是互补的关系，poin
_down

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

import ...

public class RootBeanDefinition extends AbstractBeanDefinition {
    private BeanDefinitionHolder decoratedDefinition;
    private AnnotatedElement qualifiedElement;
    boolean allowCaching = true;
    boolean isFactoryMethodUnique = false;
    volatile ResolvableType targetType;
    volatile Class<?> resolvedTargetType;
    volatile ResolvableType factoryMethodReturnType;
    final Object constructorArgumentLock = new Object();
    Object resolvedConstructorOrFactoryMethod;
    boolean constructorArgumentsResolved = false;
    Object[] resolvedConstructorArguments;
    Object[] preparedConstructorArguments;
    final Object postProcessingLock = new Object();
    boolean postProcessed = false;
    volatile Boolean beforeInstantiationResolved;
    private Set<Member> externallyManagedConfigMembers;
    private Set<String> externallyManagedInitMethods;
    private Set<String> externallyManagedDestroyMethods;

    public RootBeanDefinition() {
    }
```

我们可以在源码中看到，RootBeanDefinition继承了AbstractBeanDefinition，在其基础上面定义更多属性。从上图可以看到第一个属性BeanDefinitionHolder，那么这个BeanDefinitionHolder是么东西呢，point_down

```java
/.../

package org.springframework.beans.factory.config;

import ...

public class BeanDefinitionHolder implements BeanMetadataElement {
    private final BeanDefinition beanDefinition;
    private final String beanName;
    private final String[] aliases;

    public BeanDefinitionHolder(BeanDefinition beanDefinition, String beanName) {...}

    public BeanDefinitionHolder(BeanDefinition beanDefinition, String beanName, String[] aliases) {...}

    public BeanDefinitionHolder(BeanDefinitionHolder beanDefinitionHolder) {...}

    public BeanDefinition getBeanDefinition() { return this.beanDefinition; }

    public String getBeanName() { return this.beanName; }

    public String[] getAliases() { return this.aliases; }

    public Object getSource() { return this.beanDefinition.getSource(); }

    public boolean matchesName(String candidateName) {...}

    public String getShortDescription() {...}

    public String getLongDescription() {...}

    public String toString() { return this.getLongDescription(); }

    public boolean equals(Object other) {...}

    public int hashCode() {...}
}
```

它保存了bean的名字、别名、以及BeanDefinition持有的bean的一些基础信息。

总结一下：

1. 定义了id、别名与Bean的对应关系（BeanDefinitionHolder）
2. Bean的注解（AnnotatedElement）
3. 具体的工厂方法（Class类型），包括工厂方法的返回类型，工厂方法的Method对象
4. 构造函数、构造函数形参类型
5. Bean的class对象

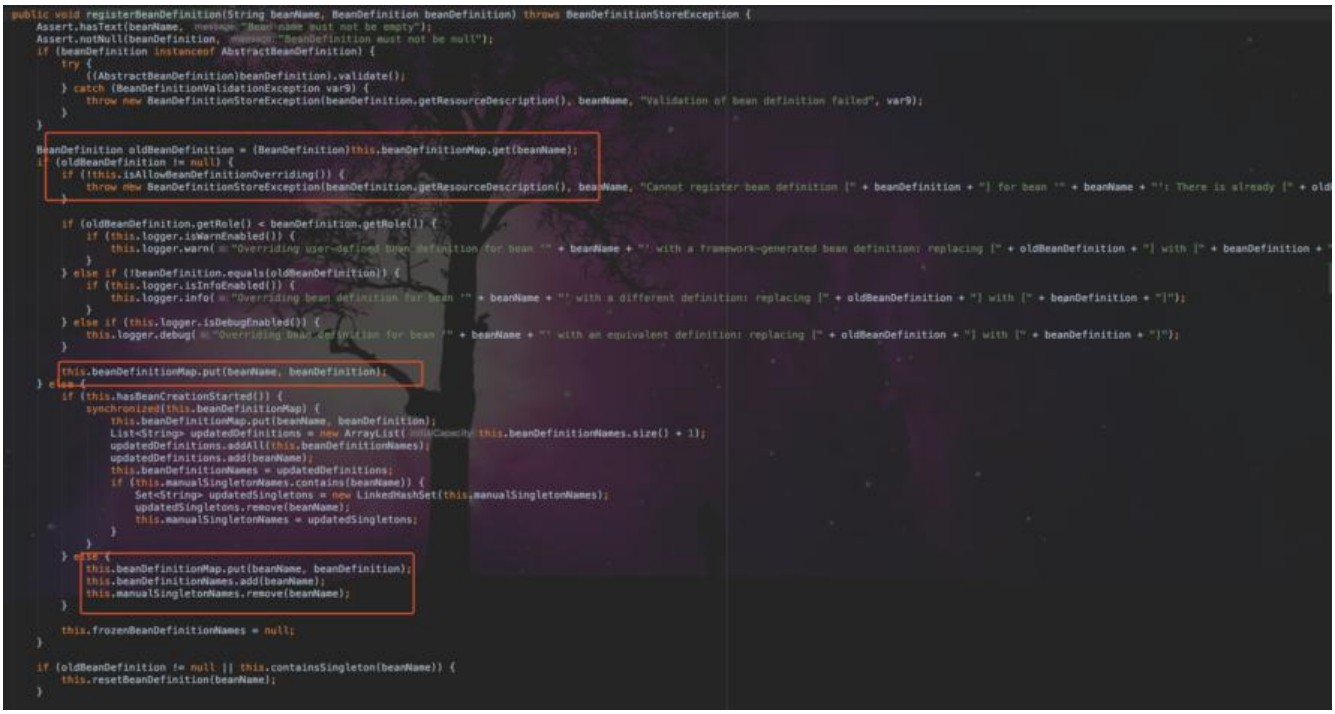
那么我这里我们就大概了解了一些BeanDefinition里面有什么东西，那回到上面问题，bean具体是么注册的呢？

六、BeanDefinitionRegistry

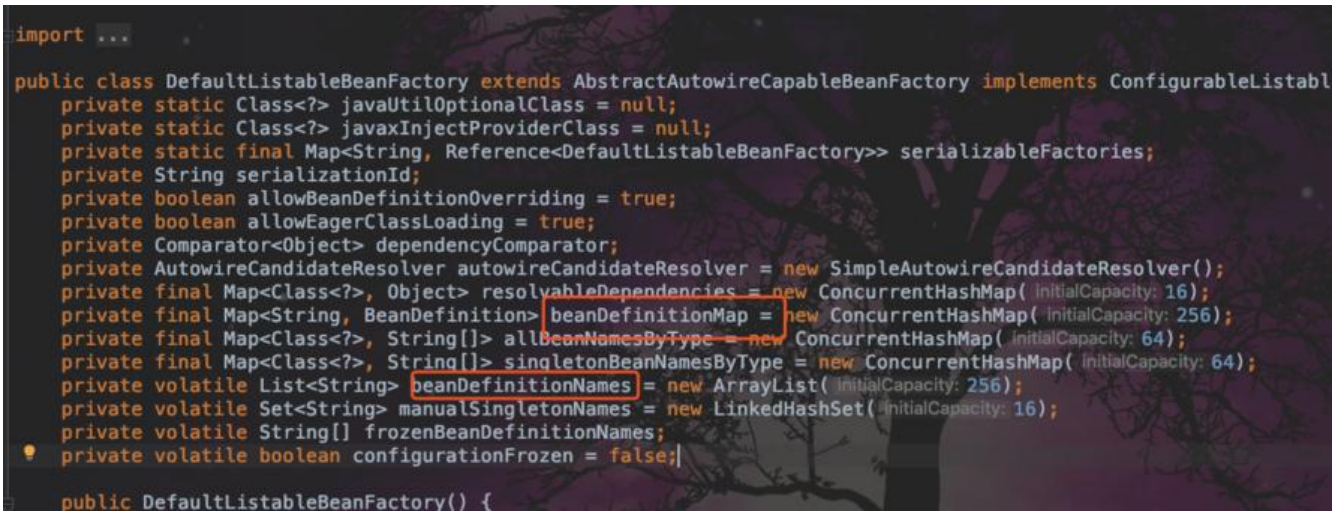
在上面我们简单看了一下BeanDefinitionRegistry这个接口中有一些方法。下面我们来着重看一下注方法是怎么实现的void registerBeanDefinition(String var1, BeanDefinition var2) throws BeanDefinitionStoreException;。点开这个方法的实现，我们可以看到spring中它有三个实现，如下图，

我们分别看一下这三个类怎么实现的：point_down

1. DefaultListableBeanFactory



首先，是做了一下校验，判断，看要注册的bean是否已经存在了等等。注意到它有两个关键的成员量：

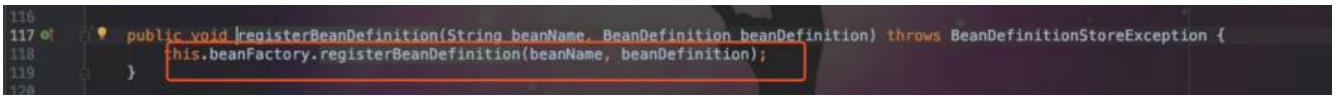


其中beanDefinitionMap这个ConcurrentHashMap注册表，而beanDefinitionNames显而易见是beanName的集合。

观察到，注册bean的最重要步骤就是this.beanDefinitionMap.put(beanName, beanDefinition);。

2. GenericApplicationContext

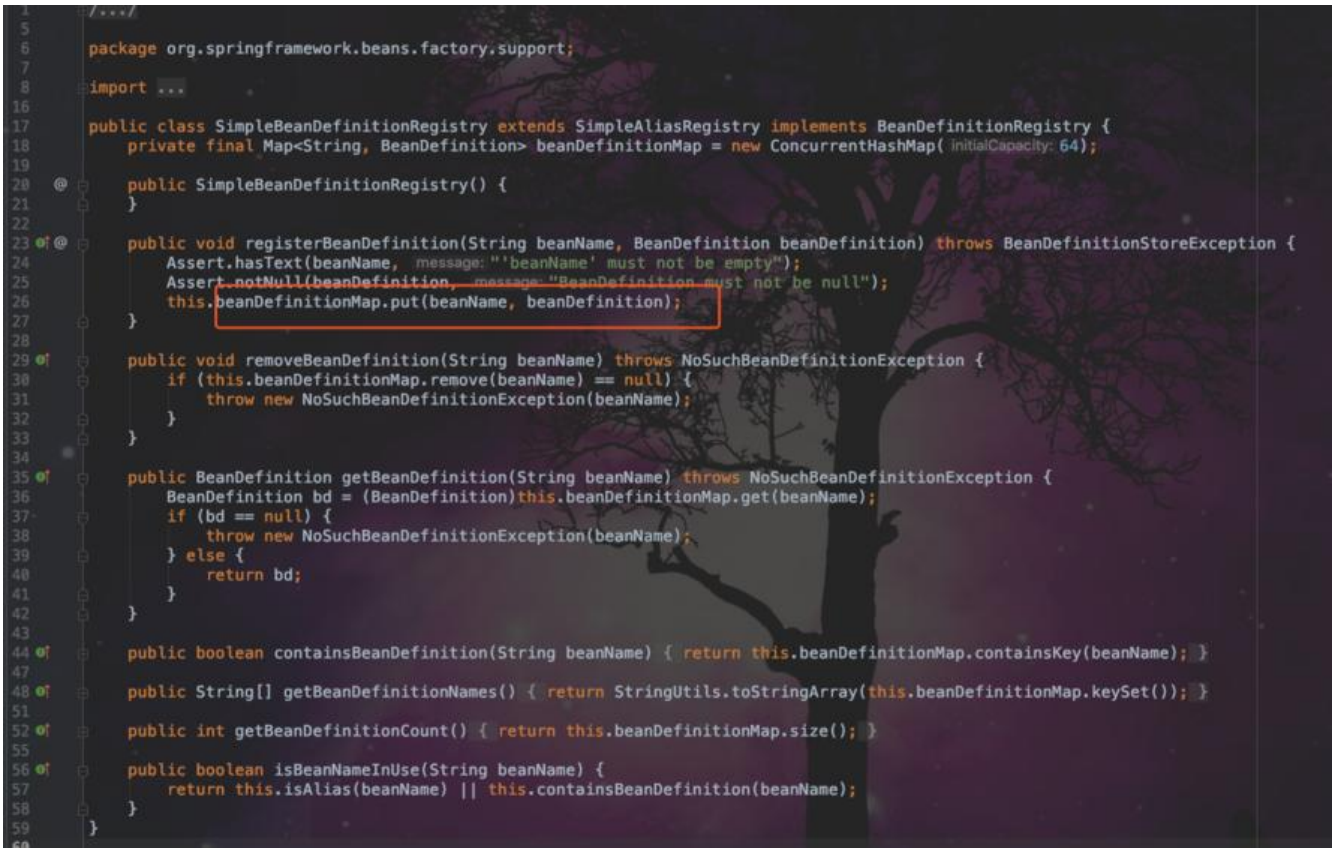
再来看一下BeanDefinitionRegistry的第二个实现类GenericApplicationContext：



注册bean的代码实现只有一行，可以看到他只是调用了上面DefaultListableBeanFactory的注册方法。

3. SimpleBeanDefinitionRegistry

最后一个，SimpleBeanDefinitionRegistry。它是这样的一个东西：
oint_down



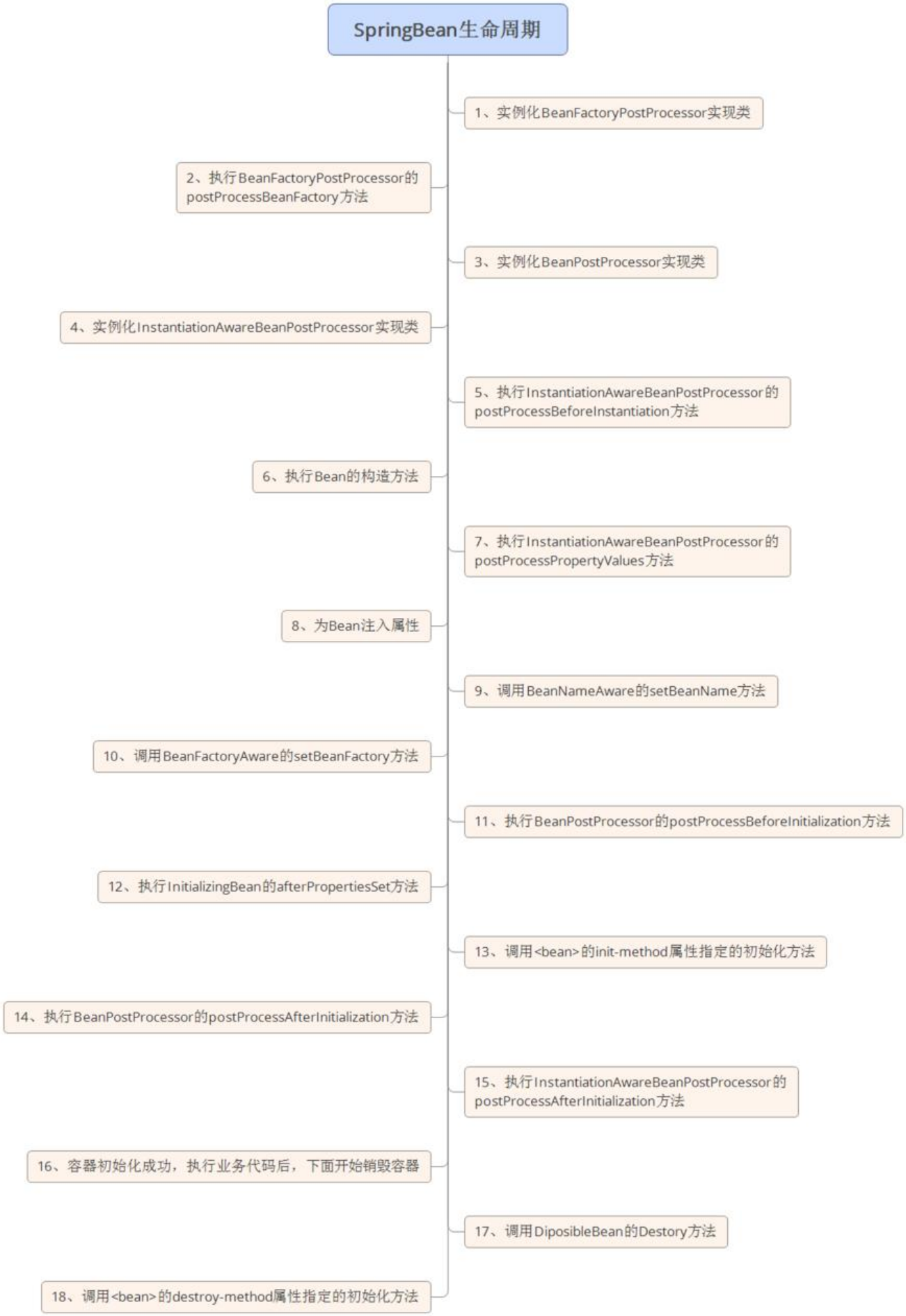
```java
/.../

package org.springframework.beans.factory.support;

import ...

public class SimpleBeanDefinitionRegistry extends SimpleAliasRegistry implements BeanDefinitionRegistry {
    private final Map<String, BeanDefinition> beanDefinitionMap = new ConcurrentHashMap( initialCapacity: 64);

    public SimpleBeanDefinitionRegistry() {
    }

    public void registerBeanDefinition(String beanName, BeanDefinition beanDefinition) throws BeanDefinitionStoreException {
        Assert.hasText(beanName,  message: "'beanName' must not be empty");
        Assert.notNull(beanDefinition,  message: "BeanDefinition must not be null");
        this.beanDefinitionMap.put(beanName, beanDefinition);
    }

    public void removeBeanDefinition(String beanName) throws NoSuchBeanDefinitionException {
        if (this.beanDefinitionMap.remove(beanName) == null) {
            throw new NoSuchBeanDefinitionException(beanName);
        }
    }

    public BeanDefinition getBeanDefinition(String beanName) throws NoSuchBeanDefinitionException {
        BeanDefinition bd = (BeanDefinition)this.beanDefinitionMap.get(beanName);
        if (bd == null) {
            throw new NoSuchBeanDefinitionException(beanName);
        } else {
            return bd;
        }
    }

    public boolean containsBeanDefinition(String beanName) { return this.beanDefinitionMap.containsKey(beanName); }

    public String[] getBeanDefinitionNames() { return StringUtils.toStringArray(this.beanDefinitionMap.keySet()); }

    public int getBeanDefinitionCount() { return this.beanDefinitionMap.size(); }

    public boolean isBeanNameInUse(String beanName) {
        return this.isAlias(beanName) || this.containsBeanDefinition(beanName);
    }
}
```

比较关键的一句就是红框所示。

这样下来，我们知道，注册bean最关键的就是往注册表的ConcurrentHashMap中put进去bean的name和BeanDefinition。

七、bean的实例化

到这里为止，我们由Dubbo的xml配置文件解析，延伸到了spring如何注册bean。那么纵观bean的个生命周期，bean的初始化可以说是为bean的一生埋下了种子，那么最后我们再看看bean初始化的一个动作---bean注册后是如何被实例化的。附一张bean生命周期全图，顺便看看bean宝宝长大了要做一些什么:



原文链接：Dubbo 系列笔记之 XML 配置文件解析流程

其实在spring源码的BeanFactory注释的头伊始，就已经说明了bean的生命周期：
oint_down

```
* <p>Bean factory implementations should support the standard bean lifecycle interfaces
* as far as possible. The full set of initialization methods and their standard order is:
* <ol>
* <li>BeanNameAware's {@code setBeanName}
* <li>BeanClassLoaderAware's {@code setBeanClassLoader}
* <li>BeanFactoryAware's {@code setBeanFactory}
* <li>EnvironmentAware's {@code setEnvironment}
* <li>EmbeddedValueResolverAware's {@code setEmbeddedValueResolver}
* <li>ResourceLoaderAware's {@code setResourceLoader}
* (only applicable when running in an application context)
* <li>ApplicationEventPublisherAware's {@code setApplicationEventPublisher}
* (only applicable when running in an application context)
* <li>MessageSourceAware's {@code setMessageSource}
* (only applicable when running in an application context)
* <li>ApplicationContextAware's {@code setApplicationContext}
* (only applicable when running in an application context)
* <li>ServletContextAware's {@code setServletContext}
* (only applicable when running in a web application context)
* <li>{@code postProcessBeforeInitialization} methods of BeanPostProcessors
* <li>InitializingBean's {@code afterPropertiesSet}
* <li>a custom init-method definition
* <li>{@code postProcessAfterInitialization} methods of BeanPostProcessors
* </ol>
*
* <p>On shutdown of a bean factory, the following lifecycle methods apply:
* <ol>
* <li>{@code postProcessBeforeDestruction} methods of DestructionAwareBeanPostProcessors
* <li>DisposableBean's {@code destroy}
* <li>a custom destroy-method definition
* </ol>
*
```

好了，言归正传，这次写的篇幅可能有点长，最后我们快点叨叨一下bean的实例化过程吧。

在此之前，我们和bean的注册结合起来，其实bean的初始化就相当于一个造书的过程。解析配置信时相当于我们要写一本书时，先有书的内容，这些配置信息就是bean的内容。有了书的内容后，我要把内容写在一页一页的纸上，相当于一个个bean的一个个属性，而这些“纸”合起来就是BeanDefiition。然后我们要把这些稿纸给到工厂，那就要有一个人去保存只写纸了，这个人就是bean的注册。在这个人手里每一堆稿纸都对应一个书名，就是bean的名字，这样我们就完成了bean的注册过程接下来就是要把这些稿纸交给工厂去装订成一本真正的书，那这个过程就是bean的实例化。

- bean什么时候会实例化?

这里我们再做一个延伸，spring bean在什么时候会进行实例化呢？这里引用一下各大博客的标准话术

第一：如果你使用BeanFactory作为Spring Bean的工厂类，则所有的bean都是在第一次使用该Bea的时候实例化

第二：如果你使用ApplicationContext作为Spring Bean的工厂类，则又分为以下几种情况：

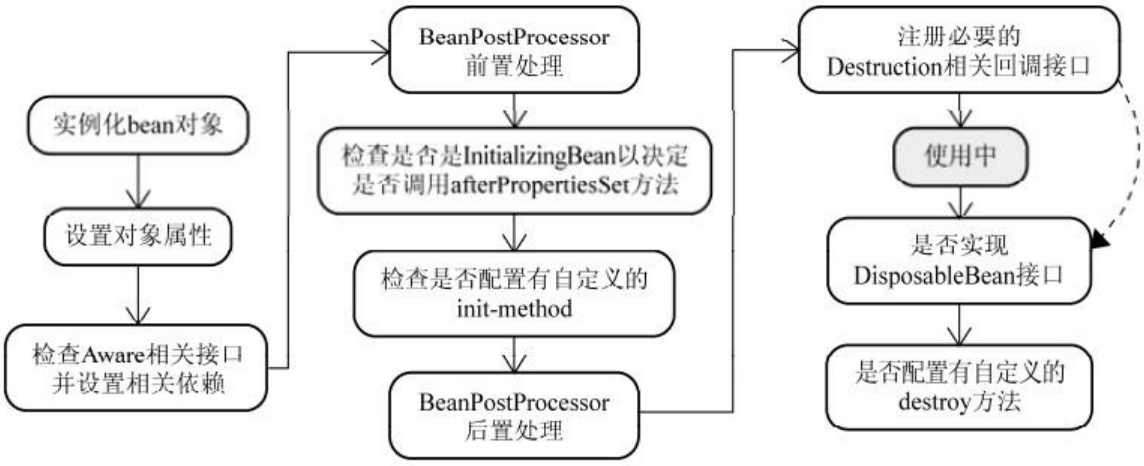
（1）：如果bean的scope是singleton的，并且lazy-init为false（默认是false，所以可以不用设置，则ApplicationContext启动的时候就实例化该Bean，并且将实例化的Bean放在一个map结构的缓中，下次再使用该Bean的时候，直接从这个缓存中取

（2）：如果bean的scope是singleton的，并且lazy-init为true，则该Bean的实例化是在第一次使该Bean的时候进行实例化

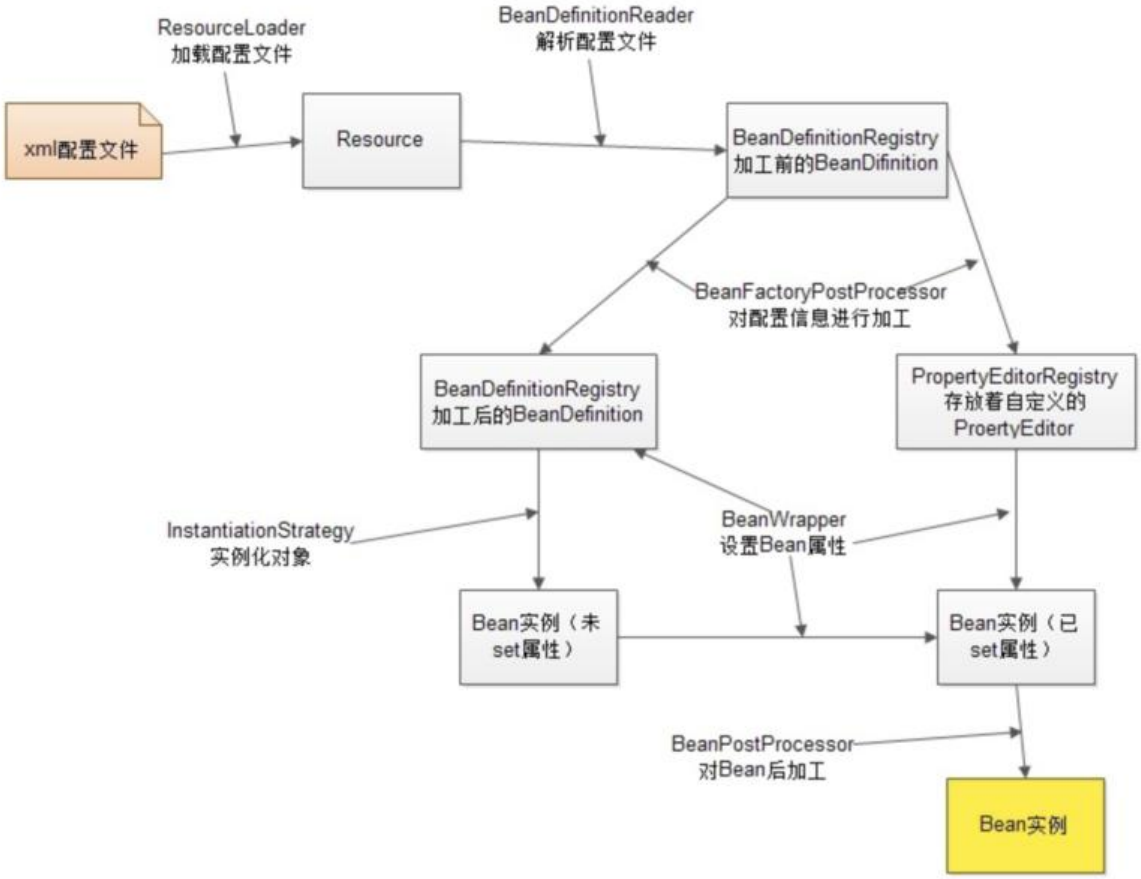
（3）：如果bean的scope是prototype的，则该Bean的实例化是在第一次使用该Bean的时候进行例化

● bean的实例化过程

一张图说明，point_down



在本文的最后，放一张Spring容器从加载配置文件到创建出一个完整Bean的作业流程：



通则达济天下，谋则远虑古今。