Spring 源码学习

2018年8月25日

《Spring 源码深度解析》学习
《Spring 框架的设计理念与设计模式分析》
面试问烂的 Spring IOC 过程
芋道源码 账号：yudaoyuanma 密码：ydym1024

idea导入源码

1. 安装 gradle，配置
2. 进入 spring-framework 目录下，使用 gradle 预编译 oxm 模块。因为该模块和 core 模块由于重新打包的依赖冲突。

   > gradlew :spring-oxm:compileTestJava

3. 导入 idea 中，提示错误时排除 `spring-aspects` 模块，因为该模块引用了 idea 未知的方面类型，无法预编译。

   > 导入 IDEA： File->import project->import from external model->Gradle
   > 设置 JDK 版本 (1.8+)
   > 排除 `spring-aspects` 模块 (Go to File->Project Structure->Modules)

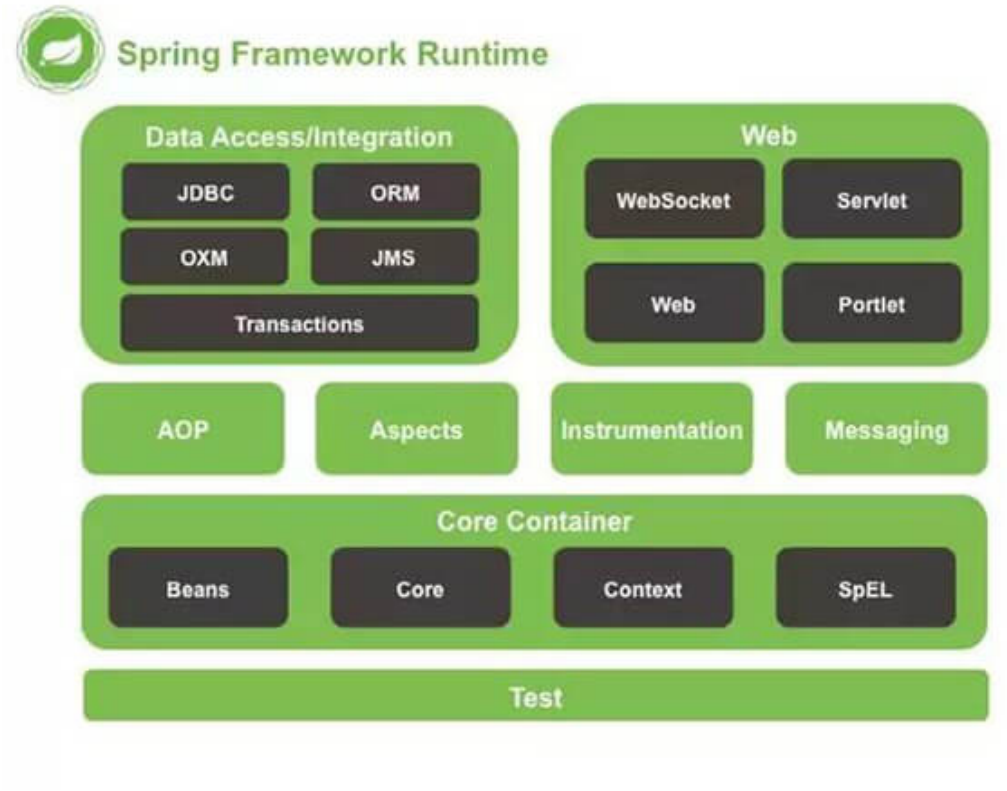
4. JUnit 测试时，若失败，可设置JVM选项

   > -XX:MaxPermSize=2048m -Xmx2048m -XX:MaxHeapSize=2048m

5. 若 idea 中调用 rebuild project ，则必须再次预编译 oxm 模块的测试。

Spring 模块

早期的 **Spring Framework**



Spring 核心容器

- Core 模块，包含 spring 的核心工具类。
- Beans 模块，包含访问配置文件、创建和管理 bean 以及进行 IOC / DI 操作相关的所有类。核心容器的主要组件是 BeanFactory 。
- Context 模块，建立在 core 和 beans 基础上，提供上下文信息给 Spring 框架。继承了 beans 的特性，添加了对 JNDI 、 EJB 、电子邮件、国际化、事件机制、校验和调度等功能的支持。ApplicationContext 接口是该模块关键。
- Expression 模块（ SpEL， Spring Expression Language ），提供了一个强大的语言表达式用于运行时查询和操纵对象。支持设置和获取属性的值，属性的分配，方法的调用，访问数组上下文、容器和索引器、逻辑和算术运算符、命名变量以及从 spring 的 IOC 容器中根据名称检索对象。支持 list 投影、选择和一般的list集合。

数据访问 **Data Access / Integration**

- JDBC (Java DataBase Connectivity) 模块，提供一个 jdbc 抽象层，可以消除冗长的 jdbc 编码和解析数据库厂商特有的错误代码。包含了 spring 对 jdbc 数据访问进行封装的所有类。
- ORM (Object Relational Mapping) 模块，提供了对 hibernate5 （ORM框架） 和 JPA （Java持久化API） 的集成。
  *OXM (Object XML Mappers) 模块，提供了一个对 Object/XML 映射实现的抽象层，该映射实现包括 JAXB 、 Castor 、 XMLBeans 、 JiBX 、 XStream 。
- TX （Transaction） 模块支持编程和声明性的事务管理，这些事务类遵循特定接口，并对所有 pojo 适用。
- JMS (Java Messaging Service) 模块，简化 JMS API 的使用，包含了制造和消费消息的特性。
- Messaging 模块，为 STOMP 提供支持，支持注解编程模型，用于从 WebSocket 客户端路由和处理 STOMP 消息。
- JCL 模块。

Web

web 上下文模块建立在应用程序上下文模块之上，简化了处理多部分请求以及将请求参数绑定到域对象的工作。

- Web 模块，提供了基础的面向 web 的集成特性。如多文件上传， servlet listeners 初始化 ioc 容器以及面向 web 应用的上下文。包含 spring 远程支持中 web 的部分。
- WebMVC 模块，包含 MVC 的实现，使得模型范围内的代码和 web forms 之间分离开来，并与 spring 的其他特性集成在一起，容纳了大量视图技术，其中包括 JSP 、 Velocity 、 Tiles 、 iText 和 POI 。
- WebSocket 模块，提供了一个在 Web 应用中实现高效、双向通讯，需要考虑客户端和服务端之间高频和低延时消息交换的机制。
- WebFlux 模块，基于 Reactive 库的响应式的 Web 开发框架。《使用 Spring 5 的 WebFlux 开发反应式 Web 应用》

AOP

AOP 提供了符合 AOP 联盟标准的面向切面编程的实现，可以定义方法拦截器和切点，将逻辑代码分隔开，降低耦合性。为基于 spring 的应用程序提供了事务管理服务，可以将声明式事务集成到应用程序。包含：aop、aspects、instrument。

- AOP 模块，将面向切面的编程集成到 Spring 中，提供了事务管理，可以将声明式事务管理集成到应用程序中。
- aspects 模块，提供了对AspectJ的支持。
- Instrument 模块，提供了 class instrumentation 支持和classloader实现，可以在特定应用服务器使用。

Test

test模块支持JUnit和TestNG对Spring组件测试。

核心组件协同工作

Bean 包装的是 Object，而 Object 必然有数据，如何给这些数据提供生存环境就是 Context 要解决的问题，对 Context 来说他就是要发现每个 Bean 之间的关系，为它们建立这种关系并且要维护好这种关系。所以 Context 就是一个 Bean 关系的集合，这个关系集合又叫 Ioc 容器，一旦建立起这个 Ioc 容器后 Spring 就可以为你工作了。那 Core 组件又有什么用武之地呢？其实 Core 就是发现、建立和维护每个 Bean 之间的关系所需要的一些列的工具，从这个角度看来， Core 这个组件叫 Util 更能让你理解。

Spring 设计理念

Spring 解决了一个非常关键的问题，可以把对象之间的依赖关系转而用配置文件来管理，也就是依赖注入机制。而这个注入关系在一个叫 Ioc 容器中管理，那 Ioc 容器就是被 Bean 包裹的对象。 Spring 正是通过把对象包装在 Bean 中而达到对这些对象的管理以及一些列额外操作的目的。

IoC

控制反转（ Inverse of Control ）：对于spring来说，就是由spring来负责控制对象的生命周期和对象间的关系。

Spring IoC 容器

容器创建 Bean 对象，将其组装在一起，配置它们并管理它们的完整生命周期。

- 容器使用依赖注入管理组成应用程序的Bean对象。
- 容器通过读取提供的配置元数据 Bean Definition 来接收对象进行实例化，配置和组装的指令。
- 配置元数据 Bean Definition 可以通过 XML 、 Java注解 或 Java Config 代码提供。

Spring 提供了两种 IoC 容器， BeanFactory 和 ApplicationContext。前者是低级容器，后者是高级容器。
BeanFactory 在 beans 项目中，包含 bean 集合的工厂类，在收到要求时实例化 Bean 对象。
ApplicationContext 在 context 项目中，拓展了 BeanFactory 接口，提供了一些管理 Bean 对象和深入的功能。

DI

依赖注入（ Dependency Injection ，简称 DI ）：在程序运行过程中，客户类不直接实例化具体服务类实例，而是客户类的运行上下文环境或专门组件负责实例化服务类，然后将其注入到客户类中，保证客户类的正常运行。简单的说，就是动态的向某个对象提供它所需要的其他对象。

Spring依赖注入的方式

构造函数注入	**setter 注入**
没有部分注入	有部分注入
不会覆盖 setter 属性	会覆盖 setter 属性
任意修改都会创建一个新实例	任意修改不会创建一个新实例
适用于设置很多属性	适用于设置少量属性

还有一种接口方式注入，需要被依赖的对象实现不必要的接口，带有侵入性。
spring还有一种注解注入装配，其实也是依赖前两种注入方式实现的。

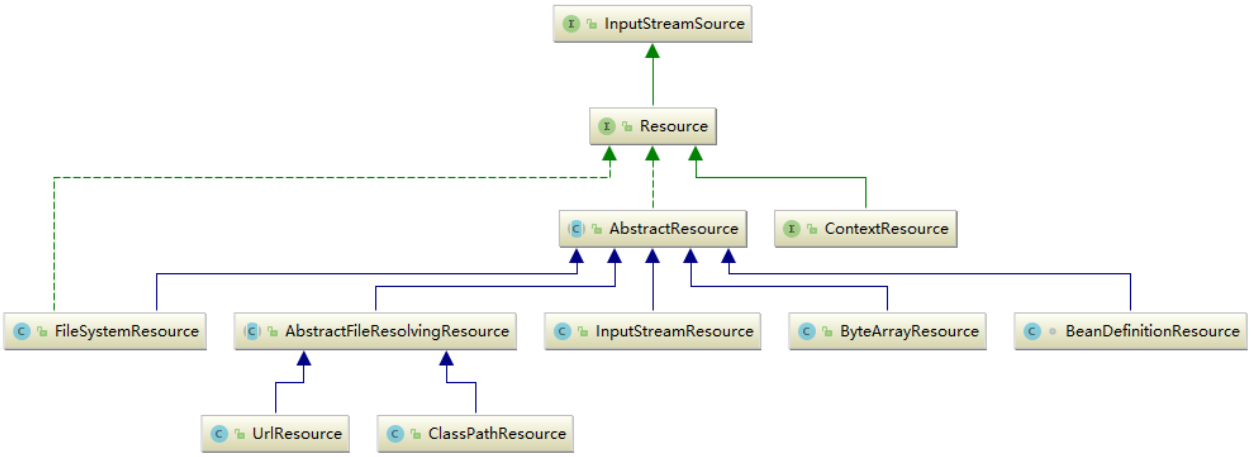
可以通过以下博客加深理解：
[谈谈对Spring IOC的理解]

core组件

Core 组件作为 Spring 的核心组件，他其中包含了很多的关键类，其中一个重要组成部分就是定义了资源的访问方式。这种把所有资源都抽象成一个接口的方式很值得在以后的设计中拿来学习。

Spring 提供了 Resource 和 ResourceLoader 来统一抽象整个资源及其定位，下面个将分别讲解。

Resource

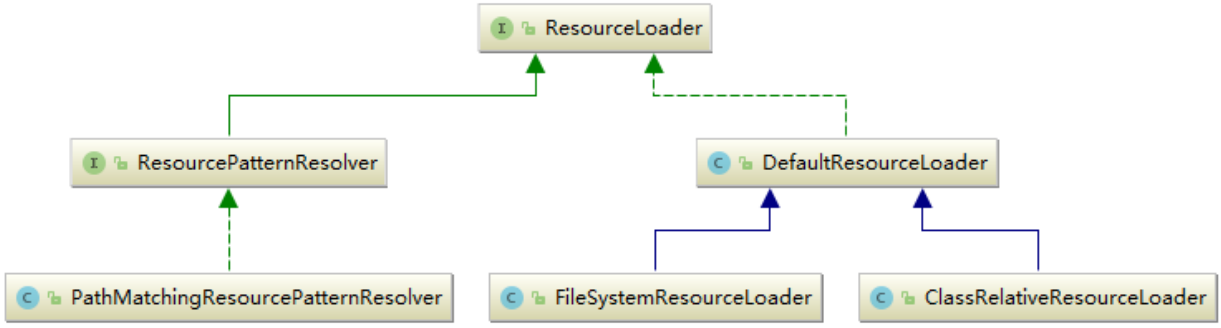


Resource 接口封装了各种可能的资源类型，也就是对使用者来说屏蔽了文件类型的不同。继承了 InputStreamSource 接口，这个接口中有个 getInputStream 方法，返回的是 InputStream 类。这样所有的资源都被可以通过 InputStream 这个类来获取，所以也屏蔽了资源的提供者。
AbstractResource 作为 Resource 接口的默认抽象实现，实现了 Resource 接口的大部分公共方法。如果想要实现自定义 Resource，继承该抽象类，根据资源特性覆盖相应的方法。

有以下几个Resource：

- FileSystemResource ： 对 `java.io.File` 类型资源的封装，只要是跟 File 打交道的，基本上与 FileSystemResource 也可以打交道。支持文件和 URL 的形式，实现 WritableResource 接口，且从 Spring Framework 5.0 开始，使用 `NIO2 API` 进行读/写交互。
- UrlResource ： 对 java.net.URL类型资源的封装。内部委派 URL 进行具体的资源操作。
- ByteArrayResource ： 对字节数组提供的数据的封装。如果通过 InputStream 形式访问该类型的资源，该实现会根据字节数组的数据构造一个相应的 ByteArrayInputStream。
- ClassPathResource ： class path 类型资源的实现。使用给定的 ClassLoader 或者给定的 Class 来加载资源。
- InputStreamResource ： 将给定的 InputStream 作为一种资源的 Resource 的实现类。

ResourceLoader



ResourceLoader，spring 资源加载的统一抽象，屏蔽了所有的资源加载者的差异，根据给定的资源文件地址，返回对应的Resource。默认实现是 DefaultResourceLoader。

ProtocolResolver，用户自定义资源解决策略。通过继承上面提到过的 AbstractResource ，来实现自定义资源，通过继承本接口，来解析对应的 `location` （资源路径），返回相应的资源句柄。DefaultResourceLoader 中会优先查看 ProtocolResolver是否配置，如果配置过，优先使用该资源解析策略来解析。

- FileSystemResourceLoader，文件系统的资源加载，用来加载类似 `D:/test/test.txt` 的资源路径。
- ClassRelativeResourceLoader，用于给定的类的所在包或所在包的子包下的资源加载。具体可以参考博客Spring5：就这一次，搞定资源加载器之ClassRelativeResourceLoader。

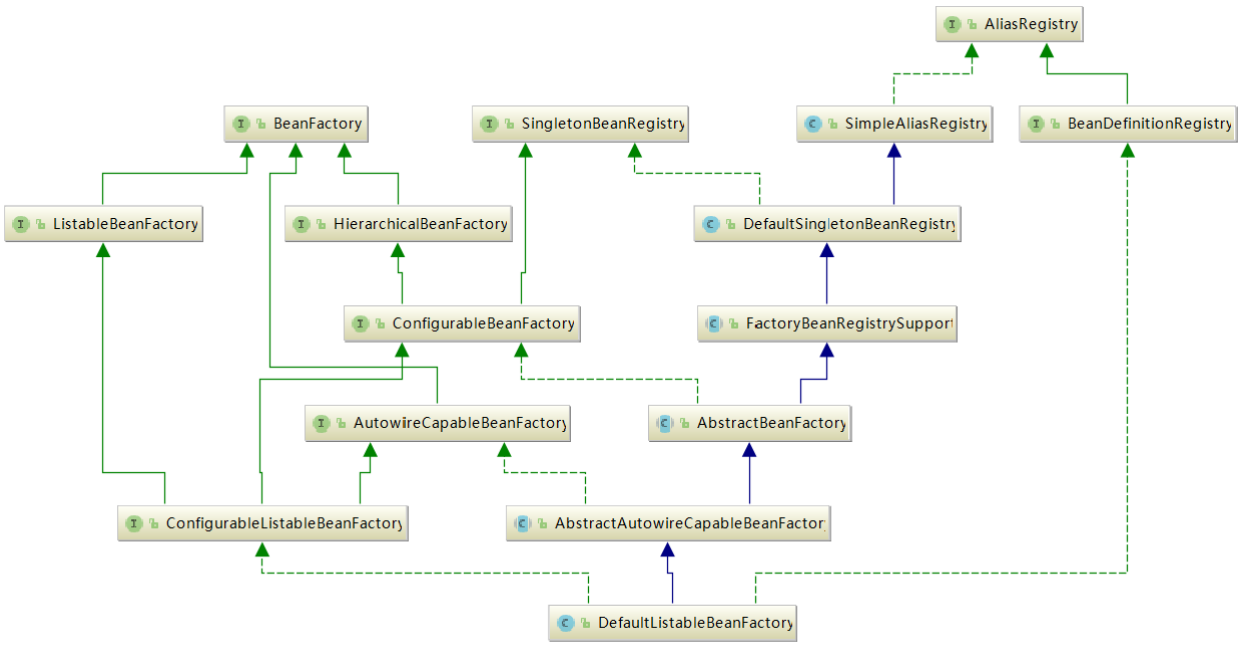
DefaultResourceLoader#getResource 每次只能根据 location 返回一个 Resource，加载多个资源只能多次调用。 ResourcePatternResolver，支持根据指定的资源路径匹配模式，每次返回多个 Resource 实例。添加了一种资源模式： `classpath*:` ，用于匹配路径下的所有资源。

- PathMatchingResourcePatternResolver，支持 `classpath*:` 和ant风格（ `**/*.xml` ）的匹配模式。

beans组件

org.springframework.beans 包下。这个包下的所有类主要解决了三件事：对 Bean 的解析，形成 Bean 的定义，Bean 的实例化。

bean工厂继承关系



BeanFactory 有三个子类：ListableBeanFactory、HierarchicalBeanFactory 和 AutowireCapableBeanFactory。
每个接口都有使用的场合，它主要是为了区分在 Spring 内部对象的传递和转化过程中，对对象的数据访问所做的限制。

- ListableBeanFactory 接口表示这些 Bean 是可列表的，实现 Bean 的list集合操作功能。

- HierarchicalBeanFactory 表示的这些 Bean 是有继承关系的，也就是每个 Bean 有可能有父 Bean。
- AutowireCapableBeanFactory 接口定义 Bean 的自动装配规则。

左侧继承树：

- ConfigurableBeanFactory 接口是在继承 HierarchicalBeanFactory 的基础上，实现 BeanFactory 的全部配置管理功能， SingletonBeanRegistry 是单例 bean 的注册接口。
- ConfigurableListableBeanFactory 接口是继承 AutowireCapableBeanFactory、ListableBeanFactory、ConfigurableBeanFactory 三个接口的一个综合接口。

右侧继承树：

- AliasRegistry 接口是别名注册接口，SimpleAliasRegistry 类是简单的实现别名注册接口的类。
- DefaultSingletonBeanRegistry 是默认的实现 SingletonBeanRegistry 接口的类，同时，继承类 SimpleAliasRegistry。
- FactoryBeanRegistrySupport 是实现 FactoryBean 注册的功能实现。继承类 DefaultSingletonBeanRegistry 。负责 FactoryBean 相关的操作，并缓存 FactoryBean 的 getObject 实例化的 bean 。判断 factory 是单例，同时已经 new 好了单例时，先尝试去缓存找；如果找不到或者不是单例，委托 doGetObjectFromFactoryBean 实例化一个。

AbstractBeanFactory 是部分实现接口ConfigurableBeanFactory，并继承类 FactoryBeanRegistrySupport 。这个是最顶层的抽象 IOC 容器空构造器，主要用来具体实现了 BeanFactory 接口。

AbstractAutowireCapableBeanFactory 是实现接口 AutowireCapableBeanFactory ，并继承类 AbstractBeanFactory 。主要的功能就是实现了默认的 bean 创建方法 createBean() 。而在这个创建过程中，提供了诸如 bean 的属性注入，初始化方法的调用，自动装配的实现，bean 处理器的调用。

DefaultListableBeanFactory 实现接口 ConfigurableListableBeanFactory、BeanDefinitionRegistry（bean定义的注册接口）， 并继承 AbstractAutowireCapableBeanFactory，实现全部类管理的功能。

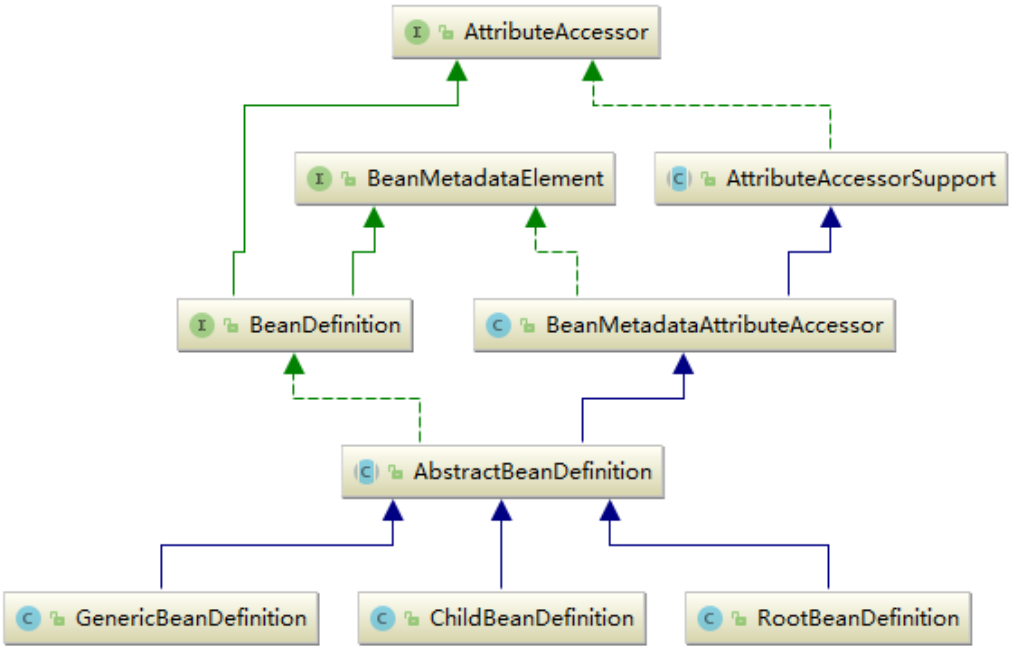
FactoryBean

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 。某些情况下，实例化 bean 过程比较复杂，如果按照传统的方式，则需要在 xml 中提供大量的配置信息，配置方式的灵活性是受限的，这时采用编码的方式可能会得到一个简单的方案。Spring 为此提供了一个 FactoryBean 的工厂类接口，用户可以通过实现该接口定制实例化 bean 的逻辑。

FactoryBean 接口对于 Spring 框架来说是重要的地址，Spring 自身就提供了 70 多个 FactoryBean 的实现。它们隐藏了实例化一些复杂 bean 的细节，给上层应用带来了便利。

FactoryBean，一个工厂 Bean，可以产生 Bean 的 Bean，这里的产生 Bean 是指 Bean 的实例。
如果一个类继承 FactoryBean ，只要实现他的 getObject 方法，就可以自己定义产生实例对象的方法。
在 Spring 内部这个 Bean 的实例对象是 FactoryBean，通过调用这个对象的 getObject 方法就能获取用户自定义产生的对象，从而为 Spring 提供了很好的扩展性。
Spring 获取 FactoryBean 本身的对象是在前面加上 & 来完成的。

Bean定义的类层次关系



Bean 的定义就是完整的描述了在 Spring 的配置文件中定义的 `<bean>` 节点中所有的信息，包括各种子节点。当 Spring 成功解析一个 `<bean>` 节点后，在 Spring 的内部就被转化成 BeanDefinition 对象，以后所有的操作都是对这个对象完成的。

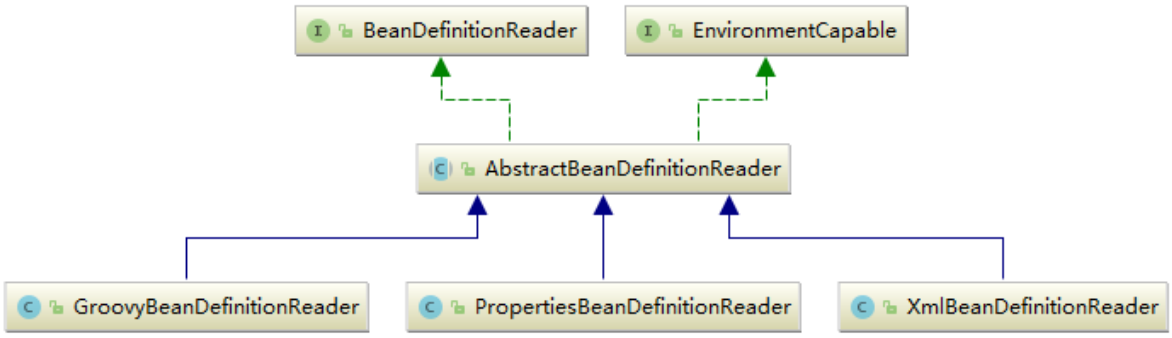
BeanDefinition 继承了两个接口:

- AttributeAccessor，定义了与其它对象的（元数据）进行连接和访问的约定，即对属性的修改，包括获取、设置、删除。
- BeanMetadataElement，Bean 元对象持有的配置元素可以通过 getSource() 方法来获取

BeanDefinition 是一个接口，在 Spring 中常用的有三种实现： RootBeanDefinition 、ChildBeanDefinition 、 GenericBeanDefinition。均继承了 AbstractBeanDefinition 抽象类。

- RootBeanDefinition，对应一般的 `<bean>` 元素标签。
- GenericBeanDefinition，2.5版本后加入的 bean 文件配置属性定义类，是一站式服务类。
- ChildBeanDefinition，子标签的含义，父 `<bean>` 用 RootBeanDefinition， 子 `<bean` 用 ChildBeanDefinition。

Bean的解析



读取 spring 配置文件中内容，并将其装换成 IoC 内部的数据结构 BeanDefinition。

Bean作用域

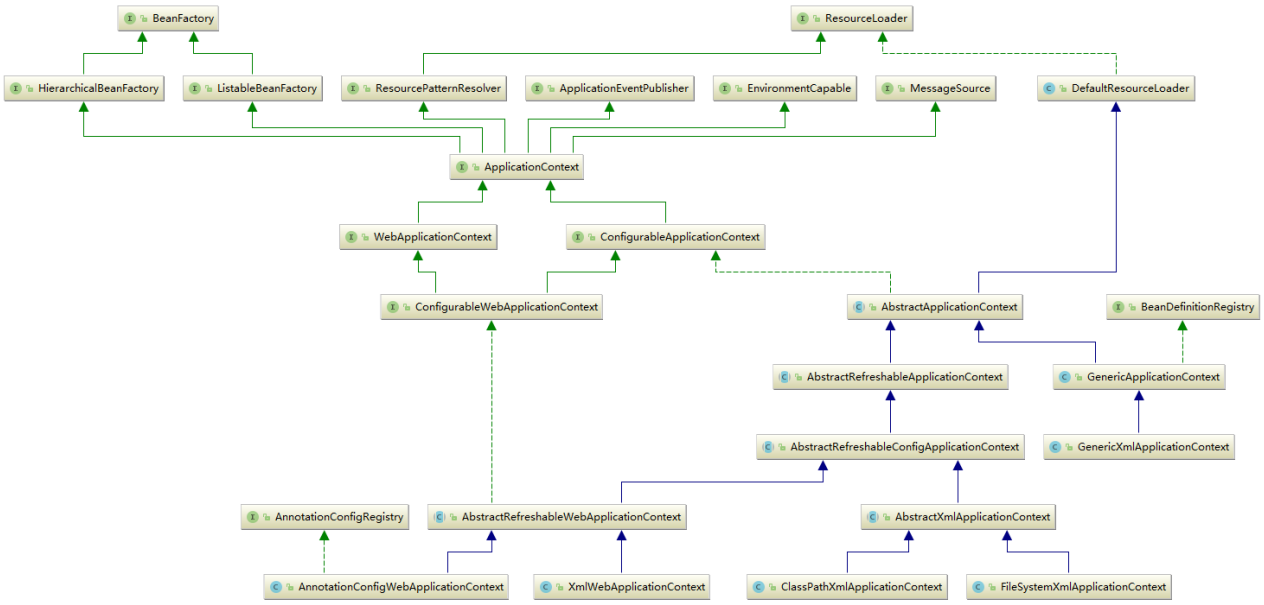
类别	说明
singleton	每个 Spring IOC 容器中仅存在一份 Bean 实例，默认
prototype	每次调用都创建新的 bean 实例
request	一次 HTTP 请求创建一个新的 Bean 实例，该 Bean 仅在 当前 HTTP 请求内有效
session	一个 session 对应一个 Bean 实例，同时该 Bean 仅在当前 HTTP Session 内有效
application	一个 Web Application 产生一个新的 Bean，仅在当前 Web Application 内有效

> 仅当用户使用支持 `Web` 的 `ApplicationContext` 时，最后三个才可用。

自定义scope，有需要的可以了解。

context组件

Context 相关的类结构图



ApplicationContext 继承了 BeanFactory，是它的拓展升级版。

- 继承 ResourcePatternResolver 接口，可以加载多种 Resource，灵活访问不同资源。
- 继承 MessageSource 接口，提供国际化的标准访问策略。
- 继承 ApplicationEventPublisher 接口，提供事件机制。

ApplicationContext 的子类主要包含两个方面：

- ConfigurableApplicationContext 表示该 Context 是可修改的，也就是在构建 Context 中用户可以动态添加或修改已有的配置信息。
- WebApplicationContext，就是为 web 准备的 Context， 可以直接访问到 ServletContext。

再往下分就是按照构建 Context 的文件类型，接着就是访问 Context 的方式。这样一级一级构成了完整的 Context 等级层次。
常用的 ApplicationContext 容器：

- ClassPathXmlApplicationContext，从 ClassPath 的 XML 配置文件中读取上下文，并生成上下文定义。程序应用上下文从环境变量中获取。

```
ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");
```

- FileSystemXmlApplicationContext，由文件系统中的XML配置文件读取上下文。

```
ApplicationContext context = new FileSystemXmlApplicationContext("bean.xml");
```

- XmlWebApplicationContext，由 Web 应用的 XML 文件读取上下文。
- AnnotationConfigWebApplicationContext，注解配置的 web 应用上下文。
- ConfigServletWebServerApplicationContext，Spring Boot的应用上下文。

总体来说 ApplicationContext 必须要完成以下几件事：

- 标识一个应用环境
- 利用 BeanFactory 创建 Bean 对象
- 保存对象关系表

- 能够捕获各种事件

Context 和 Resource 的关系

Context 是把资源的加载、解析和描述工作委托给了 ResourcePatternResolver 类来完成，他相当于一个接头人，他把资源的加载、解析和资源的定义整合在一起便于其他组件使用。

Spring 的事件类型

ApplicationContext 中提供了支持事件和代码中监听器的功能。
创建 Bean 用来监听在 ApplicationContext 中发布的时间，实现了 ApplicationListener 接口 ，当一个 ApplicationEvent 被发布后， Bean 自动被通知。

Spring 提供了5种标准事件类型

- 上下文更新事件（ ContextRefreshedEvent ）：该事件会在 ApplicationContext 被初始化或者更新时发布。也可以在调用 ConfigurableApplicationContext 接口中的 refresh() 方法时被触发。
- 上下文开始事件（ ContextStartedEvent ）：当容器调用 ConfigurableApplicationContext 的 start() 方法开始/重新开始容器时触发该事件。
- 上下文停止事件（ ContextStoppedEvent ）：当容器调用 ConfigurableApplicationContext 的 stop() 方法停止容器时触发该事件。
- 上下文关闭事件（ ContextClosedEvent ）：当 ApplicationContext 被关闭时触发该事件。容器被关闭时，其管理的所有单例 Bean 都被销毁。
- 请求处理事件（ RequestHandledEvent ）：在 Web 应用中，当一个 HTTP 请求（ request ）结束触发该事件。

继承 ApplicationEvent 自定义事件

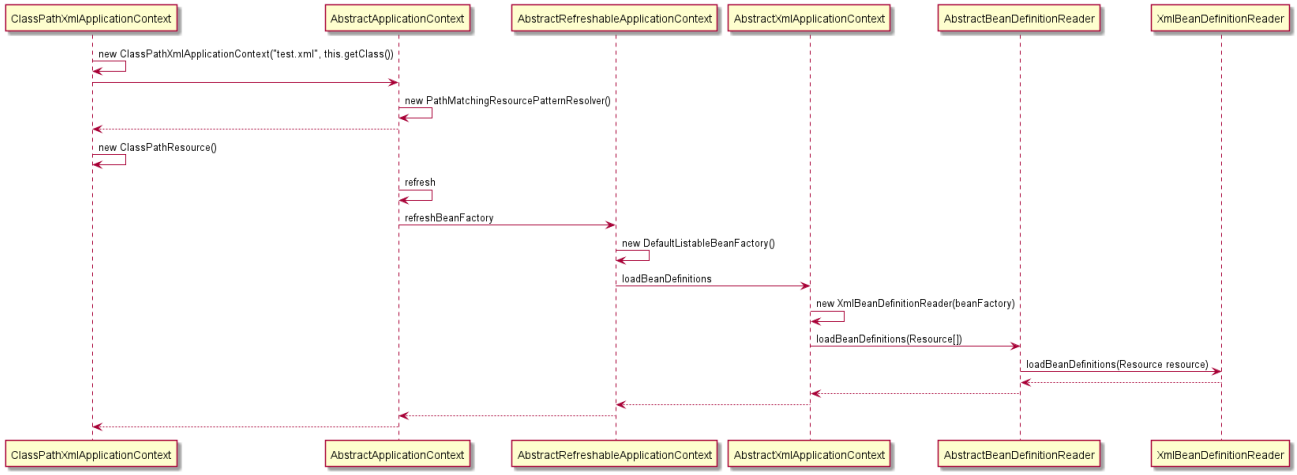
```
// 自定义事件
public class MyApplicationEvent extends ApplicationEvent {
    public MyApplicationEvent(Object source, final String msg) {
        super(source);
    }
}

// 自定义监听器
public class MyApplicationEventListener implements ApplicationListener<MyApplicationEvent>
{
    @Override
    public void onApplicationEvent(ApplicationEvent applicationEvent) {
        // 处理事件
    }
}

// 发布事件
@Autowired
private ApplicationContext applicationContext;
public void publishMyApplicationEvent(String msg) {
    MyApplicationEvent event = new MyApplicationEvent(applicationContext, msg);
    applicationContext.publishEvent(event);
}
```

应用上下文

IoC 容器实际上就是 Context 组件结合其他两个组件共同构建了一个 Bean 关系网，关键点就在 AbstractApplicationContext 类的 refresh 方法中。

具体分析可以参考：Spring IOC 容器源码分析

资源定位



ClassPathXmlApplicationContext，创建上下文，使用xml文件加载bean定义。

AbstractApplicationContext，刷新应用上下文。

AbstractApplicationContext，委托给子类 AbstractRefreshableApplicationContext，创建Bean工厂。

AbstractRefreshableApplicationContext，实现了 AbstractApplicationContext 的 refreshBeanFactory 抽象方法，这里默认创建的bean工厂是 DefaultListableBeanFactory。

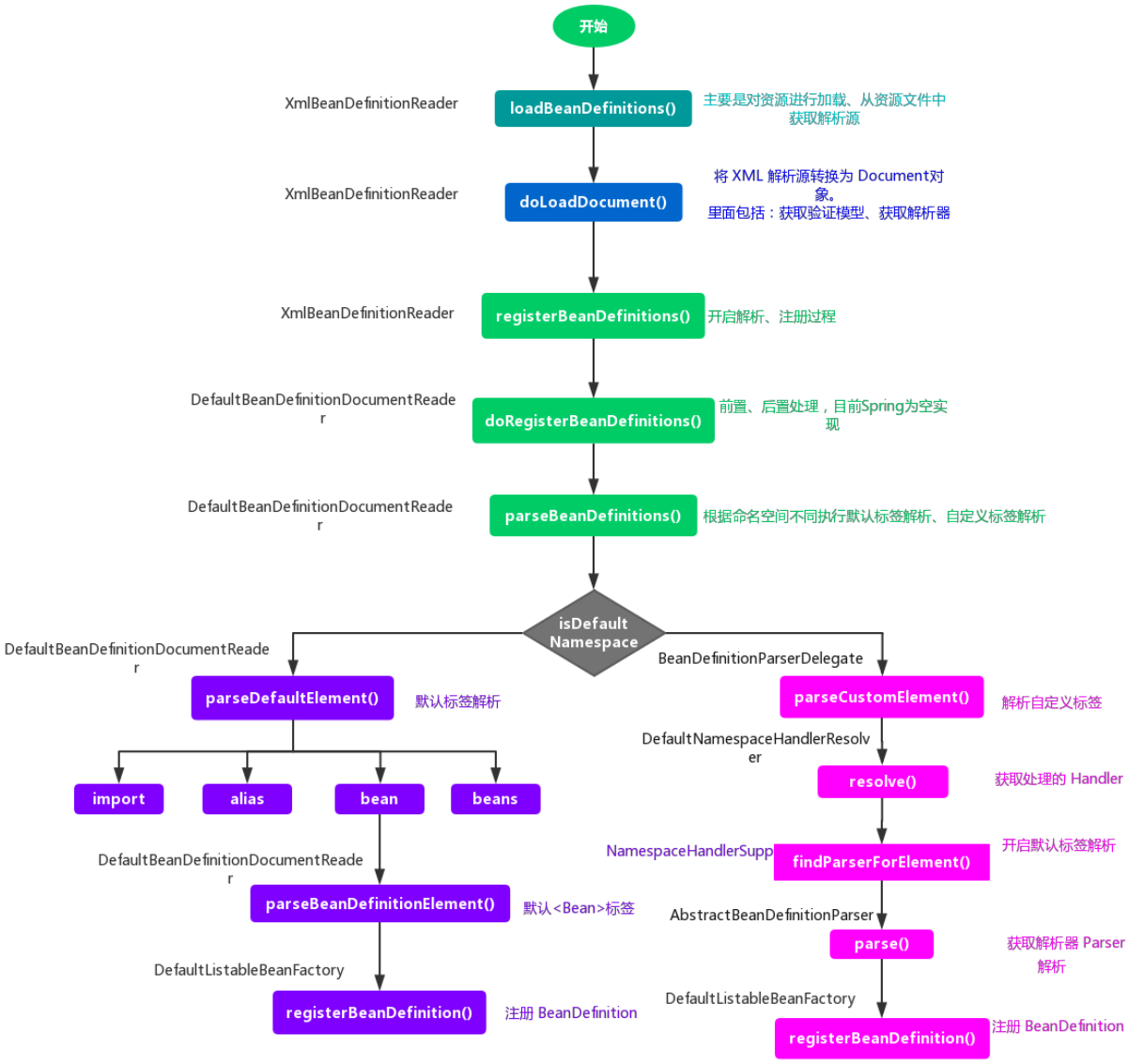
AbstractRefreshableApplicationContext，完成此上下文的bean工厂的初始化，初始化所有剩余的单例 bean。

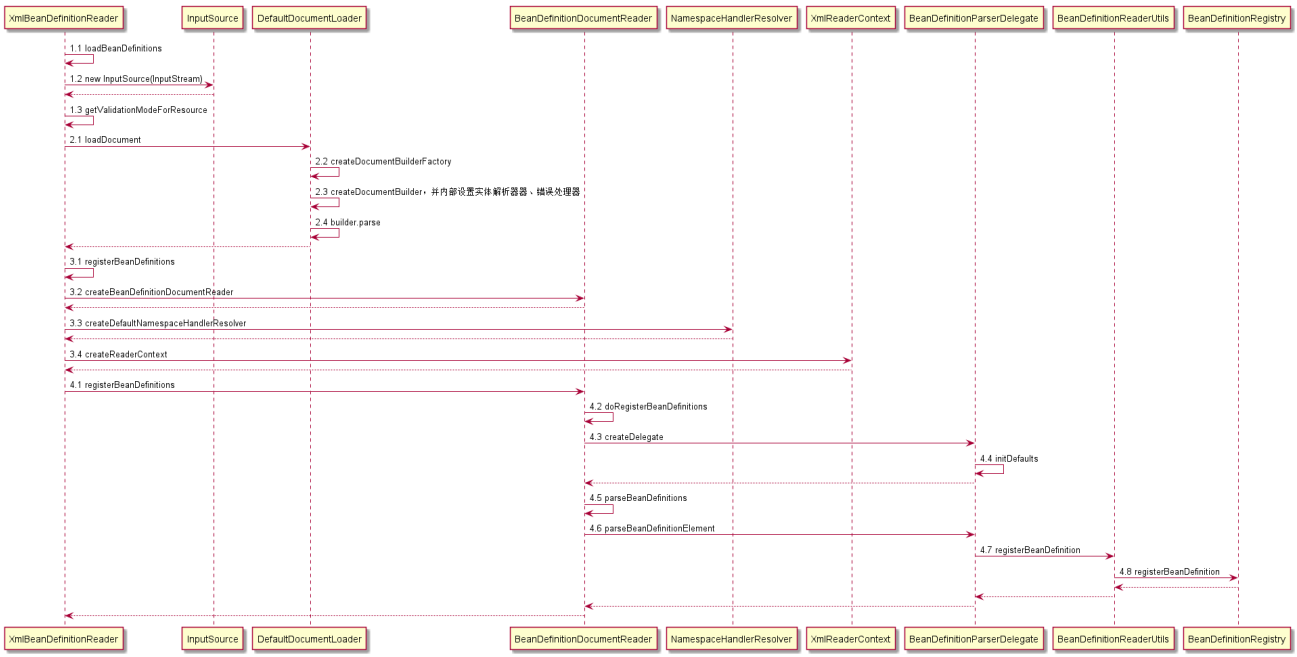
资源加载

有必要区分一下，不同类中的 loadBeanDefinitions 解析的职责：

- 在 AbstractXmlApplicationContext 类中，职责为：对 applicationContext.xml 的解析操作，就是解析工厂的那个xml。
- 在 AbstractBeanDefinitionReader 类中，职责为：从指定的资源加载 bean 定义，真正实现在其子类，这里是做了些兼容性错误处理。
- XmlBeanDefinitionReader 类，是 AbstractBeanDefinitionReader 的子类，而且是一个真正的实现类，是实现 BeanDefinitionReader 接口的 loadBeanDefinitions(Resource var1) 等方法的关键解析类。职责为：读取并真正解析 xml 文件。
- AbstractRefreshableApplicationContext 中只定义了抽象的 loadBeanDefinitions 方法，容器真正调用的是其子类 AbstractXmlApplicationContext 对该方法的实现。

Bean资源解析、生成并注册Bean定义





- 加载资源
- 资源包装
- 获取xml文件的验证模式，保证xml文件的正确性（dtd、xsd）
- 文档读取

- 创建DocumentBuilderFactory，设置命名空间和校验模式
- 创建DocumentBuilder，设置实体解析和错误处理
- 获取文档对象
- 注册bean定义，准备对象
- 创建默认的文档解析 DefaultBeanDefinitionDocumentReader
- 创建命名空间解析器，比如p标签、c标签
- 创建解析器的上下文对象，包含命名空间和被解析的资源
- 开始实际上文档解析，bean注册
- 创建bean定义解析委托类
- 使用根元素和父类委托进行初始化
- 解析bean定义，包含前置、后置处理；对import、alias、bean、beans的标签解析；对自定义标签的解析，如tx
- 解析属性以及标签
- 注册bean定义
- 将解析的bean定义放入 registry中

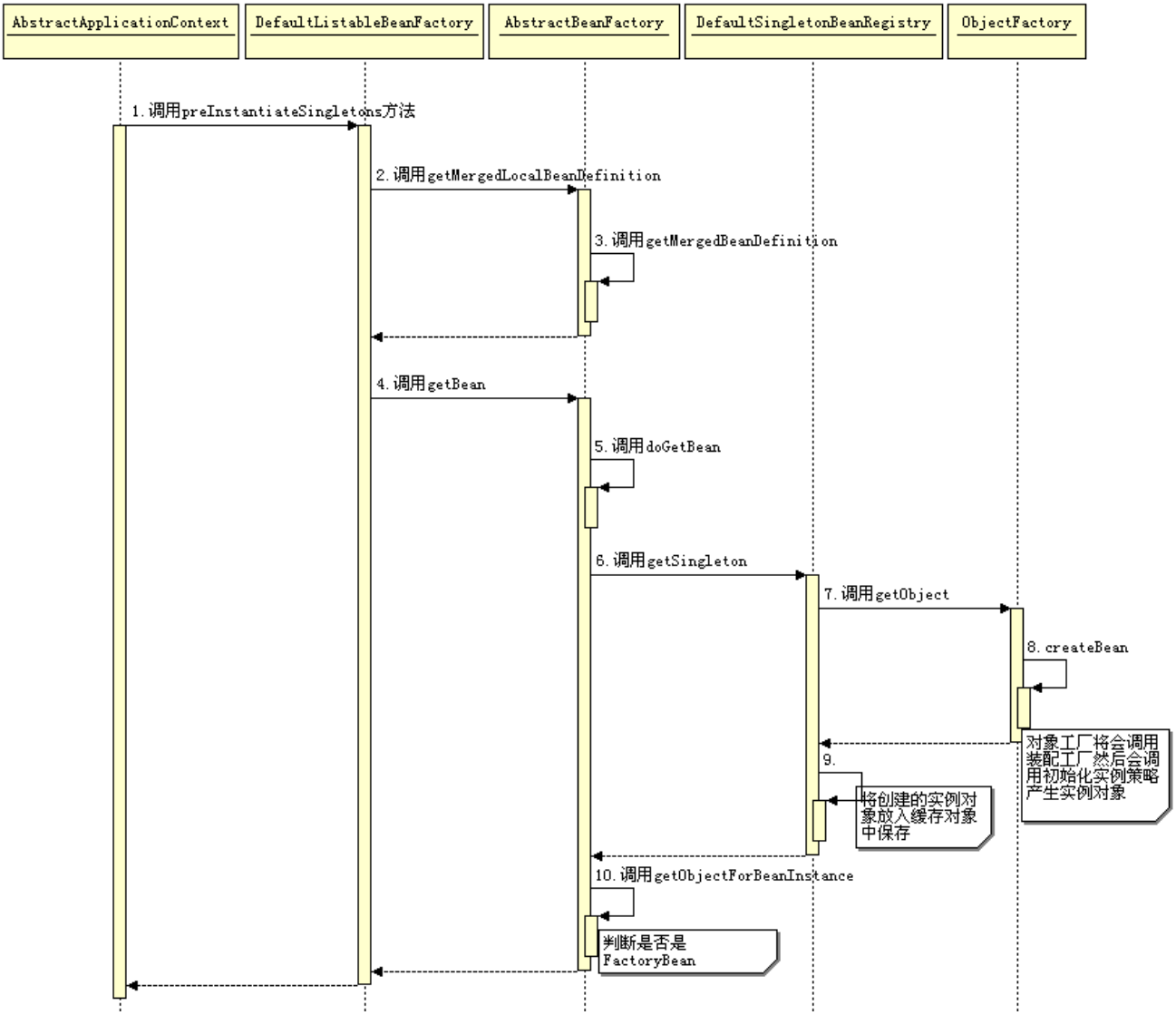
自定义标签的解析，不做过多介绍。

- 【死磕 Spring】—— IOC 之解析自定义标签

Bean的实例化/加载

显示或隐式调用 BeanFactory#getBean()，会触发 Bean 加载。
AbstractBeanFactory 为 getBean 方法进行了实现，然后实际上委托给了 doGetBean 方法。

Bean 实例创建时序图



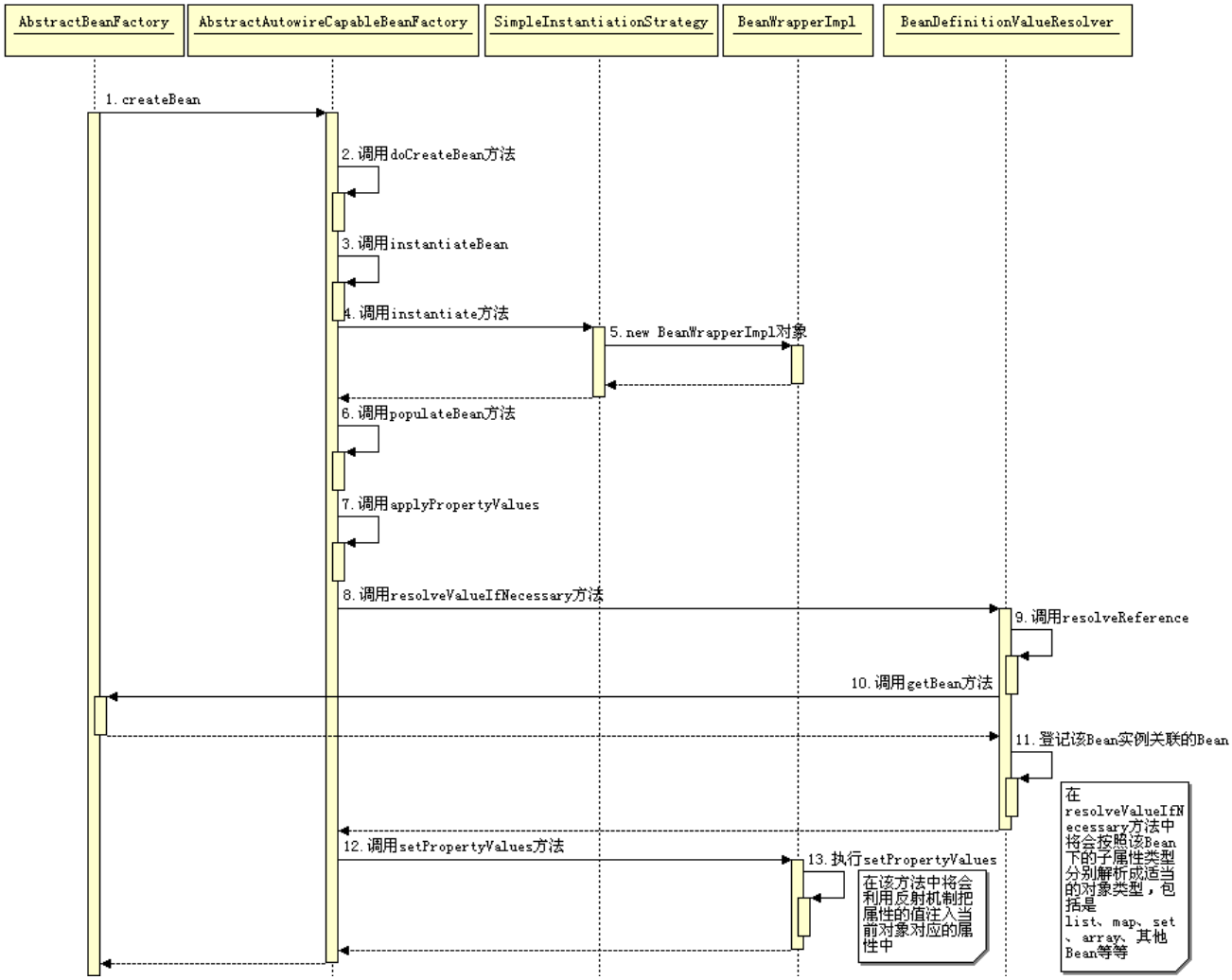
简单说一下，这里根据bean的作用域进行实例化：

- 如果Bean定义的单例模式(Singleton)，则容器在创建之前先从缓存中查找，以确保整个容器中只存在一个实例对象。
- 如果Bean定义的是原型模式(Prototype)，则容器每次都会创建一个新的实例对象。
- 两者都不是，则根据Bean定义资源中配置的生命周期范围，选择实例化 Bean 的合适方法，这种在 Web 应用程序中 比较常用，如：request、session、application等作用域。

这里定义了根据 Bean 定义的模式，采取的不同创建 Bean 实例对象的策略，具体的Bean实例对象的创建过程由实现了 ObejctFactory 接口的匿名内部类的 createBean 方法完成。

ObejctFactory 使用委派模式，具体的 Bean 实例创建过程交给了其 AbstractAutowireCapableBeanFactory 的 createBean 方法完成。

Bean 对象关系建立



AbstractAutowireCapableBeanFactory 的 createBean 具体实现 是由 doCreateBean 方法来执行的。总结起来包括四个过程：

- createBeanInstance(String beanName, RootBeanDefinition mbd, Object[] args) 方法，实例化 bean 。
- populateBean(String beanName, RootBeanDefinition mbd, BeanWrapper bw) 方法，进行属性填充。
- initializeBean(final String beanName, final Object bean, RootBeanDefinition mbd) 方法，初始化 bean 对象。
- 循环依赖的处理。

下面将一一进行讲解。

AbstractAutowireCapableBeanFactory#createBeanInstance，创建bean实例

AbstractAutowireCapableBeanFactory 的 createBeanInstance 方法（2-3步之间），使用 工厂方法（静态工厂、工厂实例）、自动装配特性 或者 默认无参构造函数 生成 Bean 实例对象。

这里还要讲一下 BeanWrapper， `org.springframework.beans.BeanWrapper` 相当于一个代理器，有两个顶级类接口，分别是 PropertyAccessor 和 PropertyEditorRegistry 。

- PropertyAccessor 接口定义了各种访问Bean属性的方法，如 setPropertyValue(String,Object)，setPropertyValues(PropertyValues pvs) 等。
- PropertyEditorRegistry 是属性编辑器的注册表。

Spring通过BeanWrapper完成Bean属性的填充工作。在Bean实例被 InstantiationStrategy 创建出来之后，容器将 Bean 实例通过 BeanWrapper 包装起来。由于它是接口，必然有个实现类，实现依赖注入的具体实现。那就是 BeanWrapperImpl ，它的作用是：

- Bean包裹器
- 属性访问器
- 属性编辑器注册表。

AbstractAutowireCapableBeanFactory#populateBean，bena属性的依赖注入

AbstractAutowireCapableBeanFactory 中还不得不讲的是 populateBean 方法：用来对Bean属性的依赖注入进行处理。这部分包括两部分：对属性值的解析，依赖的注入。

AbstractAutowireCapableBeanFactory # applyPropertyValues 方法（完成属性转换）：

- 属性值类型不需要转换时，不需要解析属性值，直接准备进行依赖注入。
- 属性值需要进行类型转换时，如对其他对象的引用等，首先需要解析属性值，然后对解析后的属性值进行依赖注入。

调用 BeanDefinitionValueResolver # resolveValueIfNecessary 方法对属性值的解析：

> 当容器在对属性进行依赖注入时，如果发现属性值需要进行类型转换，如属性值是容器中另一个Bean实例对象的引用，则容器首先需要根据属性值解析出所引用的对象，然后才能将该引用对象注入到目标实例对象的属性上去，对属性进行解析的由 resolveValueIfNecessary 方法实现。

这里的解析的创建和注入是一个递归的过程，不断的解析内部的 Bean 属性值。

AbstractAutowireCapableBeanFactory # setPropertyValues 方法解析：前面也说到 BeanWrapper 接口的实现类就是 BeanWrapperImpl。一系列的依赖注入都在这个实现类里面， BeanWrapperImpl 使用的抽象类就是 AbstractPropertyAccessor ，里面有 setPropertyValues 方法的实现，有个模板模式，就是调用在 AbstractPropertyAccessor 中的抽象方法 setPropertyValue（在 BeanWrapperImpl 实现）。简单总结一下：

- 对于集合类型的属性，将其属性值解析为目标类型的集合后直接赋值给属性。
- 对于非集合类型的属性，大量使用了JDK的反射和内省机制，通过属性的 getter 方法(reader method)获取指定属性注入以前的值，同时调用属性的 setter 方法(writer method)为属性设置注入后的值。

AbstractAutowireCapableBeanFactory#initializeBean，bena初始化

经过了bean创建、属性注入、依赖处理，bean的加载还剩下最后一步：bean初始化，这里的操作其实和 bean 的生命周期有关，后面会详细涉及。

- 执行 Aware 方法，特殊的bean进行处理，包括 BeanNameAware、BeanClassLoaderAware、BeanFactoryAware 三个aware。
- 执行所有 bean 后置处理器（BeanPostProcessor）的 postProcessBeforeInitialization 方法。
- 执行用户自定义的 init 方法，包括： InitializingBean的afterPropertiesSet 方法， `init-method` 属性指定的方法。

- 执行所有 bean 后置处理器（BeanPostProcessor）的 postProcessAfterInitialization 方法。

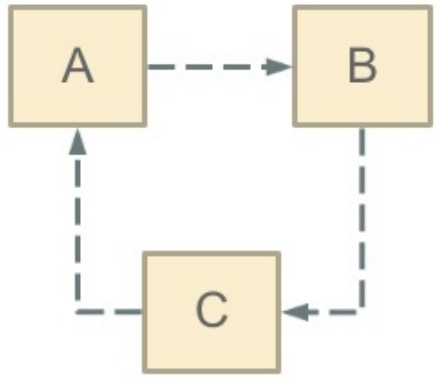
至此，Spring 的 bean 加载就讲完了，如果后面还想回顾，可以参考 【死磕 Spring】—— IOC 之加载 bean：总结。

循环依赖的处理

AbstractAutowireCapableBeanFactory 的 doCreateBean 的 循环依赖处理，其实不仅仅在这个方法中存在循环依赖，在整个 bean 加载过程中都有涉及。

循环依赖

简单普及一下循环依赖：



循环依赖，其实就是循环引用，就是两个或者两个以上的 bean 互相引用对方，最终形成一个闭环，如 A 依赖 B，B 依赖 C，C 依赖 A 。即一个死循环的过程，除非存在终结条件，不然始终无法退出循环。

Spring 循环依赖的场景有两种：

- 构造器的循环依赖。
- field 属性的循环依赖。

对于构造器的循环依赖，Spring 是无法解决的，只能抛出 BeanCurrentlyInCreationException 异常表示循环依赖，所以分析的都是基于 field 属性的循环依赖。而且 Spring 只解决了 作用域 为 singleton 的循环依赖，其他模式下的循环依赖， Spring 无法解决，直接抛出 BeanCurrentlyInCreationException 异常。

AbstractBeanFactory # doGetBean()，从三层缓存中获取单例对象

doGetBean 方法中，首先会调用 DefaultSingletonBeanRegistry # getSingleton 方法，根据 beanName 从单例 bean 缓存中获取 单例 bean 对象，如果不为空则直接返回。

getSingleton 方法会从三个缓存中获取bean对象，可以理解为三层缓存:

- singletonObjects ：单例对象的缓存，存储最终形态的 bean 对象。
- singletonFactories ： 创建单例对象的工厂的缓存。
- earlySingletonObjects ：早期的单例对象的缓存，还没经过属性注入、bean 处理器处理的对象。

还存在两个变量：singletonsCurrentlyInCreation、allowEarlyReference，是 Spring 解决单例循环依赖的核心变量。

- singletonsCurrentlyInCreation，存储正在创建中的 bean 的名称，即 bean 正在初始化但是没有完成初始化的过程。
- allowEarlyReference，允许早期暴露单例引用，即允许从三级缓存中通过 getObject() 方法，拿到对象。

getSingleton 方法流程如下：

- 从一级缓存 singletonObjects 获取单例对象。
- 如果一级缓存中不存在 且 当前指定的 beanName 正在创建中，就再从二级缓存中获取。
- 如果仍然没有获取到 且 允许通过三级缓存获取，则从三级缓存中获取 单例对象工厂。
- 如果获取到 单例对象工厂，则通过其 getObject() 方法，获取单例对象，并将其加入到二级缓存中，并从三级缓存中移除 对象工厂。

这样，就从**三级缓存升级到二级缓存**了。所以，二级缓存存在的意义，就是缓存三级缓存中的 ObjectFactory 的 getObject() 方法的执行结果，提早曝光的单例 Bean 对象，这也就是 Spring 解决单例循环依赖的 核心。

AbstractAutowireCapableBeanFactory # doCreateBean()，三级缓存的添加

回到创建bean对象的方法中来，当一个 Bean 满足以下三个条件时，则调用 addSingletonFactory 方法，将对象工厂添加到三级缓存中：

- bean 定义是单例的。
- allowEarlyReference = true，允许添加到三级缓存中，提前暴露 bean。
- singletonsCurrentlyInCreation 中有值，当前 bean 正在创建中。

同时，addSingletonFactory 这个方法发生在 createBeanInstance 方法之后，即这个 bean 对象已经被创建出来了，但没有进行 初始化 和 属性填充，但对于需要它的人来说已经足够了，所以将此对象的引用提供给三级缓存的 getObject 方法。

AbstractBeanFactory # doGetBean()，一级缓存的添加

在从缓存中获取不到单例对象的时候，就只好手动创建了，调用 AbstractBeanFactory 的 getSingleton() 方法。这个方法首先里会将这个单例 bean 标记为 正在创建中的状态，然后开始创建单例 bean ，这里的创建流程其实就是上面的 bean 加载的流程，不做过多讲解。
在创建完毕以后，将最终完美的单例 bean 添加到一级缓存中。

AbstractBeanFactory # addSingleton() 方法：

- 添加一级缓存
- 移除二级缓存
- 移除三级缓存

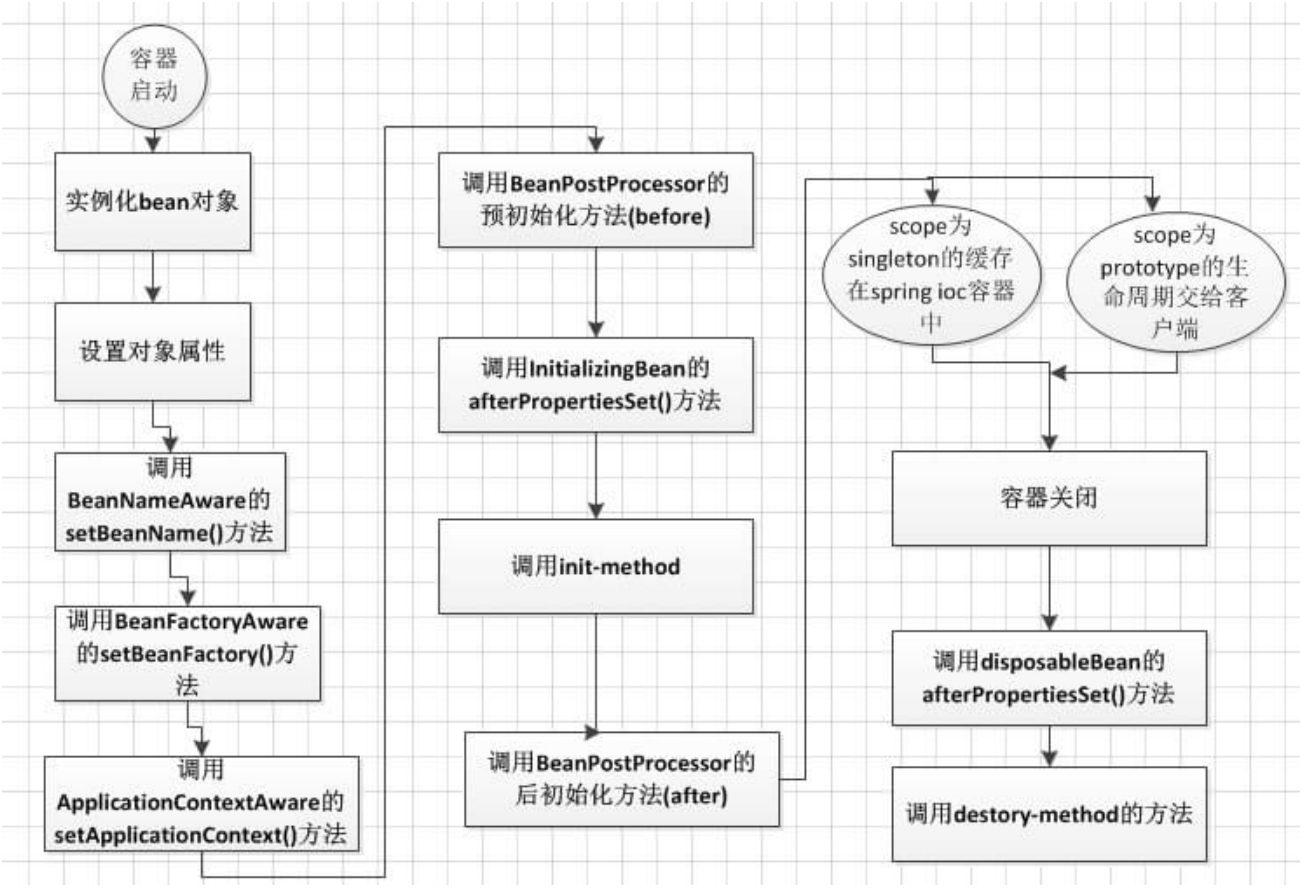
最后来描述下一开始那张循环依赖的解决流程：

- A 完成初始化，并将自己提前曝光出来（通过 ObjectFactory 即三级缓存将自己提前曝光），在初始化的时候，发现自己依赖对象 B，此时就会去尝试 getBean(B)，这个时候发现 B 还没有被创建出来。
- B 开始走创建流程，在 B 初始化的时候，同样发现自己依赖 C，C 也没有被创建出来。
- C 又开始初始化流程，但是在初始化的过程中发现自己依赖 A，于是尝试 getBean(A)，这个时候由于 A 已经添加至三级缓存中，通过 ObjectFactory 提前曝光过了，所以可以通过 ObjectFactory

getObject() 方法来拿到 A 对象，C 拿到 A 对象后顺利完成初始化，然后将自己添加到一级缓存中。

- 回到 B ，B 也可以拿到 C 对象，完成初始化，将自己添加到一级缓存中。
- A 可以顺利拿到 B 完成初始化，将自己添加到一级缓存中。
- 单例的循环依赖解决完成。

Bean 的生命周期



Bean 初始化

实例化 Bean 对象

- Spring 容器根据配置中的 Bean Definition 中实例化 Bean 对象。
- Bean Definition 可以通过 XML，Java 注解或 Java Config 代码提供。
- Spring 使用依赖注入填充所有属性，如 Bean 中所定义的配置。

Aware 相关的属性，注入到 Bean 对象

- 如果 Bean 实现 BeanNameAware 接口，则工厂通过传递 Bean 的 beanName 来调用 setBeanName(String name) 方法。
- 如果 Bean 实现 BeanFactoryAware 接口，工厂通过传递自身的实例来调用 setBeanFactory(BeanFactory beanFactory) 方法。

调用相应的方法，进一步初始化 Bean 对象

- 如果存在与 Bean 关联的任何 BeanPostProcessor ，则调用 preProcessBeforeInitialization(Object bean, String beanName) 方法。

- 如果 Bean 实现 InitializingBean 接口，则会调用 afterPropertiesSet() 方法。
- 如果为 Bean 指定了 init 方法（例如 的 init-method 属性），那么将调用该方法。
- 如果存在与 Bean 关联的任何 BeanPostProcessor ，则将调用 postProcessAfterInitialization(Object bean, String beanName) 方法。

Bean 的销毁

如果 Bean 实现 DisposableBean 接口，当 spring 容器关闭时，会调用 destroy() 方法。
如果为 Bean 指定了 destroy 方法（例如 的 destroy-method 属性），那么将调用该方法。

Spring IOC 的拓展

深入理解Spring 之 Spring 进阶开发必知必会 之 Spring 扩展接口

让 bean 对象有拓展能力，可以通过几个类来实现，通过继承接口中定义的方法，spring 会在适当时刻进行调用。

- FactoryBean，确定Bean的形状，XML 方式的 AOP 就是通过该接口实现的。
- BeanFactoryPostProcessor，在构建 BeanFactory 时调用，在Bean创建之前，读取Bean的元属性，并根据自己的需求对元属性进行改变，比如将Bean的scope从singleton改变为prototype。
- InstantiationAwareBeanPostProcessor ，在 Bean 实例化前后做一些操作。
- BeanPostProcessor，在 Spring 容器完成 Bean 的实例化、配置和其他的初始化后，添加一些自己的逻辑处理。
- InitializingBean，在 Bean 实例创建时调用（属性设置完毕后）。
- DisposableBean，在 Bean 实例销毁时被调用。

BeanPostProcessor

BeanPostProcessor 的作用：在 Spring 容器完成 Bean 的实例化、配置和其他初始化后，使用该接口对其进行一些配置、增加一些自己的处理逻辑。

BeanPostProcessor 是允许自定义 bean 实例修改的 Spring 的一个工厂钩子，是 Spring 提供的对象实例化阶段强有力的扩展点，允许 Spring 在实例化 bean 阶段对其进行定制化修改。常见的使用场景是处理标记接口实现类 或者 为当前对象提供代理实现（例如 AOP）。

普通的 BeanFactory 是不支持自动注册 BeanPostProcessor 的，需要手动调用 addBeanPostProcessor 方法添加。添加后的 BeanPostProcessor 适用于该 BeanFactory 创建的所有 bean。而高级容器 ApplicationContext 不同，它可以在其 bean 定义中自动检测所有的 BeanPostProcessor 并自动完成注册，同时将他们应用到随后创建的任何 Bean 中，这段代码在 `AbstractApplicationContext#registerBeanPostProcessors()` 中，也就是被 refresh() 方法调用了。

使用示例：

```
public class MyBeanPostProcessor implements BeanPostProcessor{

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws
BeansException {
        System.out.println("Bean [" + beanName + "] 初始化之前");
```

```
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException {
        System.out.println("Bean [" + beanName + "] 初始化之后");
        return bean;
    }

    public void run(){
        System.out.println("调用MyBeanPostProcessor");
    }
}
```

Aware

让 bean 继承 Aware 接口，可以让 bean 具有获取相应资源的能力。
Aware 是一个具有标识作用的超级接口，实现了该接口的 bean 是具有被 Spring 容器通知的能力，通知的方式是采用回调的方式。 Spring 容器在初始化时主动检测当前 bean 是否实现了相应的 Aware 接口，如果实现了则回调其 set 方法将相应的参数设置给该 bean ，这个时候该 bean 就能从 Spring 容器中取得相应的资源。

列举一些常用的 Aware：

名字	作用
LoadTimeWeaverAware	加载Spring Bean时织入第三方模块，如AspectJ
BeanClassLoaderAware	加载Spring Bean的类加载器
BootstrapContextAware	资源适配器BootstrapContext，如JCA,CCI
ResourceLoaderAware	底层访问资源的加载器
BeanFactoryAware	声明BeanFactory
PortletConfigAware	PortletConfig
PortletContextAware	PortletContext
ServletConfigAware	ServletConfig
ServletContextAware	ServletContext
MessageSourceAware	国际化
ApplicationEventPublisherAware	应用事件
NotificationPublisherAware	JMX通知
BeanNameAware	声明Spring Bean的名字

使用示例：

```java
@Component
public class MyApplicationContextAware implements ApplicationContextAware {
    private ApplicationContext applicationContext;

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) {
        this.applicationContext = applicationContext;
        System.out.println("为 bean 设置了 ApplicationContext");
    }
}
```

一个通俗的例子

把 Ioc 容器比作一个箱子，这个箱子里有若干个球的模子，可以用这些模子来造很多种不同的球，还有一个造这些球模的机器，这个机器可以产生球模。
那么他们的对应关系就是：BeanFactory 是那个造球模的机器，球模就是 Bean，而球模造出来的球就是 Bean 的实例。
那前面所说的几个扩展点又在什么地方呢？
BeanFactoryPostProcessor 对应到当造球模被造出来时，你将有机会可以对其做出适当的修正，也就是他可以帮你修改球模。
而 InitializingBean 和 DisposableBean 是在球模造球的开始和结束阶段，你可以完成一些预备和扫尾工作。
BeanPostProcessor 就可以让你对球模造出来的球做出适当的修正。
最后还有一个 FactoryBean，它可是一个神奇的球模。这个球模不是预先就定型了，而是由你来给他确定它的形状，既然你可以确定这个球模型的形状，当然他造出来的球肯定就是你想要的球了，这样在这个箱子里你可以发现所有你想要的球。