Spring框架

认识spring

简介

Spring框架是由于软件开发的复杂性而创建的。Spring使用的是基本的**JavaBean**来完成以前只可能由EJB完成的事情。

◆目的：解决企业应用开发的复杂性

◆功能：使用基本的JavaBean代替EJB，并提供了更多的企业应用功能

◆范围：任何Java应用

Spring是一个轻量级控制反转(IoC)和面向切面(AOP)为内核的容器框架。

Spring致力于JavaEE应用各层的解决方案，是企业应用一站式开发很好的选择，在表现层它提供了SpringMVC以及整合Struts的功能，在业务逻辑层可以管理事务、记录日志等，在持久层可以整合Hibernate、Mybatis等框架。虽然Spring贯穿表现层、业务逻辑层、持久层，但**Spring并不是要取代那些已有的优秀框架，而是可以高度开放的与其它优秀框架无缝整合。**

发展历程

2001年10月 Rod Johnson写了一本书《Expert One-on-OneJ2EE》，为了实现这本书的想法，完成了开源框架interface21。

2003 年 Rod Johnson 和同伴在interface21的基础上开发了一个全新的框架命名为Spring.
Spring。2004 年 03 月，1.0 版发布。

- 基于xml schema的配置 @Deprecated

2006 年 10 月，2.0 版发布。

- 注解驱动的配置
- 新的Spring AOP使用方式：可以给POJO编写AspectJ注解，也可以用基于XML
  Schema的配置

  （1.x调用一套专有的Spring AOP API）。
- 更容易的事务声明
- JPA支持
- 表单标签库
- 异步的JMS支持
- 脚本语言的支持：脚本语言包括JRuby，Groovy和BeanShell

2007 年 11 月，更名为SpringSource，同时发布了 Spring 2.5。

- 注解驱动的配置：为简化Bean的配置，spring2.0增加了对一些注解的支持。spring2.5支持更多的注解，包括\@Autowired和JSR-250注解中的@Resource,@PostConstruct和@PreDestroy。
- 组件的自动搜索：引入了组件自动搜索的功能，可以从classpath里自动搜索带有特定注解的组件，从而免去了手功配置。
- 对AspectJ加载时织入的支持： 支持在加载时向springIOC容器里织入AspectJ切面，这将允许在SpringAOP所能支持的范围之外使用AspectJ切面。
- **基于注解的web控制器**：支持一种新的基于注解的方式来开发web控制器。会自动搜索使用@Controller注解的控制器类，同时，配置在@RequestMapping，@RequestParam和@ModelAttribute注解里的信息也会被自动搜索到。
- 增强的测试支持： spring2.5建立了新的测试框架，这个框架被称为springTestContext框架。该框架支持注解驱动测试，同时，该框架也对底层测试框架进行了抽象。

2009 年 12 月，Spring 3.0 发布。

- 支持了基于Java类的配置
- 功能升级：

  添加了引入环境profile功能

  添加了\@enable注解，使用特定功能

  添加了对声明式缓存的支持，能够使用简单的注解声明缓存边界和规则

  添加的用于构造器注入的c命名空间，类似与Spring2的p命名空间，用于对应属性注入

开始支持Servlet3.0，包括基于java配置中生命Servlet和Filter，不再只仅仅借助web.xml改善对于JPA的 支持，让JPA能在Spring中完整配置JPA，不必再使用persistence.xml文件

- SpringMVC功能增强：

  自动绑定路径变量到模型属性中

  提供了@RequestMappingProduces和consumes属性，用于匹配请求中的Accept和Content-Type头部信息提供@RequestPart注解，用于将Multipart请求中的而某些部分绑定到处理器的方法参数中支持Flash属性，在redirect请求后依然能够存活的属性，flash属性的RedirectAttribute类型可以使用Servlet3.0的异步请求，允许一个独立的线程中处理请求等等。

2013 年 12 月，Pivotal 宣布发布 Spring 框架 4.0。

- 全面支持Java 8.0：Lambda表达式支持，Java8.0提供的日期和时间API支持，重复注解支持，空指针终结者Optional的支持。
- 核心容器的增强：泛型依赖注入的支持，Map依赖注入的支持，List依赖注入的支持，Lazy注解延迟加载的支持，Condition条件注解的支持，CGLib动态代理增强。
- 支持基于Groovy DSL定义Bean
- Web增强：SpringMVC基于Servlet3.0开发，提供RestController注解，提供AsyncRestTemplate支持客户端的异步无阻塞请求。
- WebSocket的支持

2017 年 09 月，Spring 5.0 发布。

- JDK基线更新：Spring 5.0代码库运行于Java 8之上。

- 核心框架修订
- 核心容器更新
- 使用Kotlin进行函数式编程
- 响应式编程模型
- 测试方面的提升

优点

1、非侵入式设计

Spring是一种非侵入式（no-invasive）框架，它可以使应用程序代码对框架的依赖最小化。

2、方便解耦、简化开发

Spring是一个大工厂，可以将所有 对象的创建、依赖关系的维护交给Spring容器管理，大大降低了组建之间的耦合。

3、支持AOP

允许将一些通用任务，比如安全、事务、日志等，进行集中式管理，从而提高程序的复用性。

4、支持声明式事务处理

通过配置就可以完成对事务的管理，无需手动编程。

5、方便整合其它优秀框架

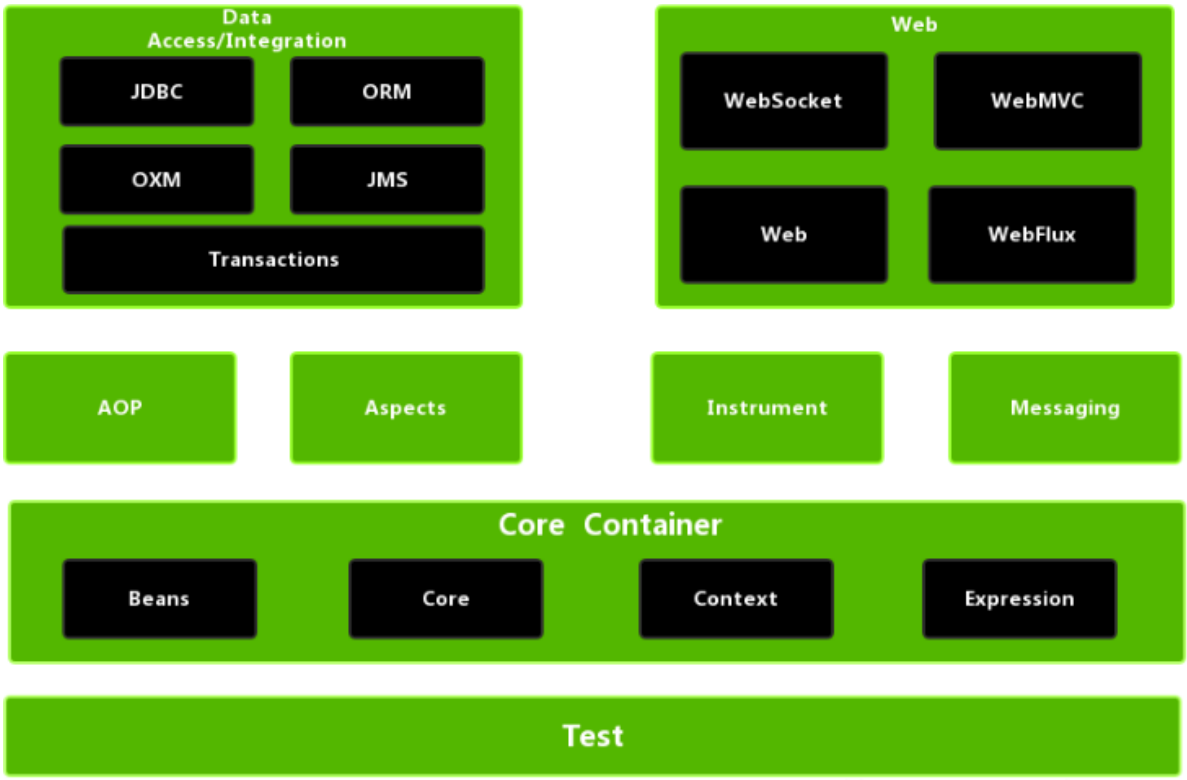
Spring可以与大多数框架无缝整合。

6、测试方便

Spring支持Junit4，可通过注解测试程序，很方便。

7、降低了使用JavaEE API的难度Spring对JavaEE开发中难用的一些API进行了封装，降低了这些API的使用难度。

体系结构



组成 Spring框架的每个模块（或组件）都可以单独存在，或者与其他一个或多个模块联合实现。每个模块的功能如下：

核心容器

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

spring-beans 和 spring-core 模块是 Spring框架的核心模块，包含了控制反转（Inversion of Control,IOC）和依赖注入（Dependency Injection, DI）。BeanFactory 接口是Spring框架中的核心接口，它是工厂模式的具体实现。BeanFactory

使用控制反转对应用程序的配置和依赖性规范与实际的应用程序代码进行了分离。但**BeanFactory 容器实例化后并不会自动实例化 Bean，只有当 Bean 被使用时 BeanFactory容器才会对该 Bean 进行实例化与依赖关系的装配。**

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

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

AOP 和设备支持

由 spring-aop、spring-aspects 和 spring-instrument 3 个模块组成。

spring-aop 是 Spring 的另一个核心模块，是 AOP 主要的实现模块。作为继 OOP后，对程序员影响最大的编程思想之一，AOP 极大地开拓了人们对于编程的思路。在 Spring 中，他是以JVM 的动态代理技术为基础，然后设计出了一系列的AOP横切实现，比如前置通知、返回通知、异常通知等，同

时， Pointcut接口来匹配切入点，可以使用现有的切入点来设计横切面，也可以扩展相关方法根据需求进行切入。

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

spring-instrument 模块是基于 JAVA SE中的”java.lang.instrument”进行设计的，应该算是AOP 的一个支援模块，**主要作用是在 JVM启用时，生成一个代理类，程序员通过代理类在运行时修改类的字节，从而改变一个类的功能，实现 AOP 的功能。**

数据访问及集成

由spring-jdbc、spring-tx、spring-orm、spring-jms 和 spring-oxm 5 个模块组成。

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

spring-tx 模块是 Spring JDBC 事务控制实现模块。使用Spring框架，它对事务做了很好的封装，通过它的 AOP 配置，可以灵活的配置在任何一层；但是在很多的需求和应用，直接使用JDBC事务控制还是有其优势的。其实，事务是以业务逻辑为基础的；一个完整的业务应该对应业务层里的一个方法；如果业务操作失败，则整个事务回滚；所以，事务控制是绝对应该放在业务层的；但是，持久层的设计则应

该遵循一个很重要的原则：保证操作的原子性，即持久层里的每个方法都应该是不可以分割的。所
以，在使用 Spring JDBC 事务控制时，应该注意其特殊性。

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

**spring-jms 模块（Java Messaging Service）能够发送和接受信息，自 Spring
Framework 4.1 以后，他还提供了对 spring-messaging 模块的支撑。**

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

Web

由 spring-web、spring-webmvc、spring-websocket 和 spring-webflux 4 个模块组成。

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

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

spring-websocket 模块主要是与 Web 前端的全双工通讯的协议。

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

报文发送

即spring-messaging模块。

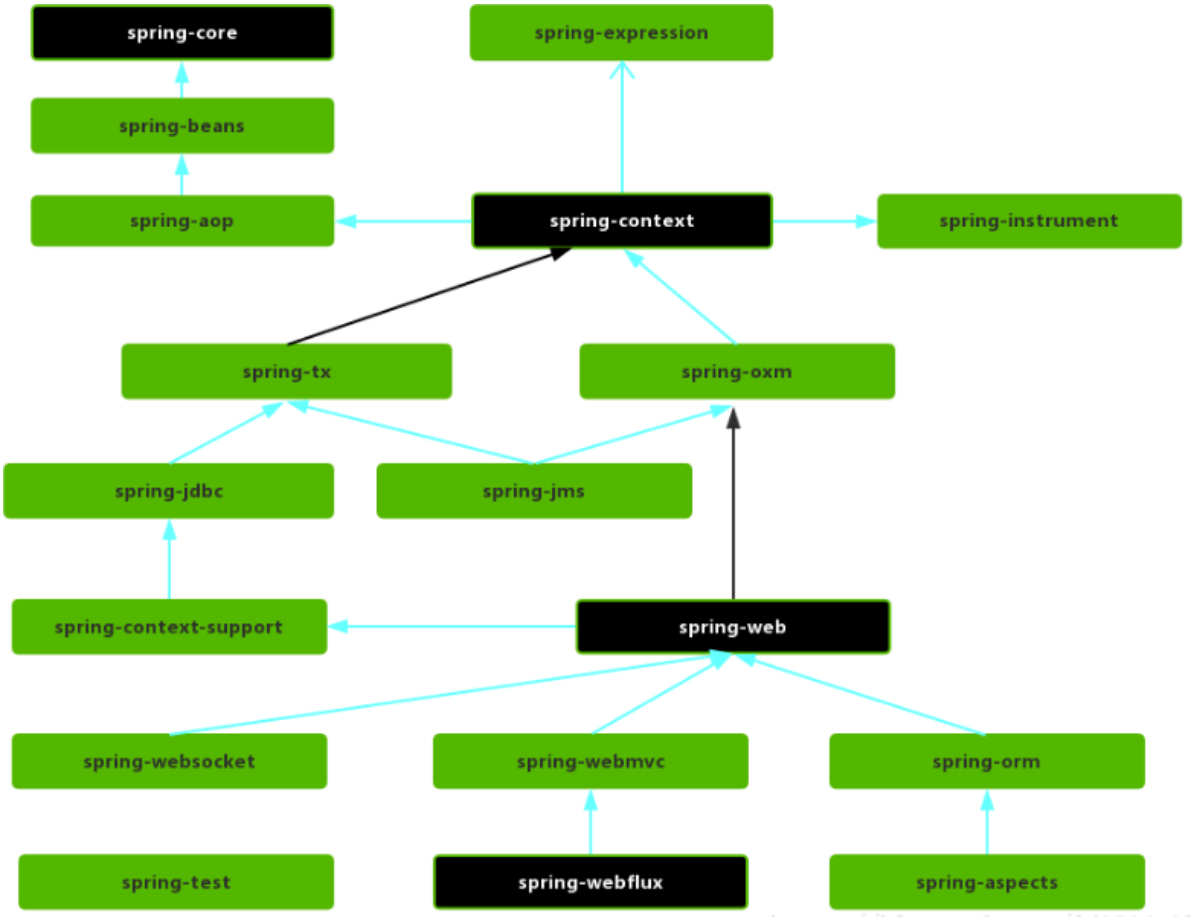
spring-messaging是从Spring4开始新加入的一个模块，主要职责是为Spring框架集成一些基础的报文
传送应用。

Test

即spring-test模块。

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

各模块之间的依赖关系



源码阅读

IOC核心功能

IoC

IoC 全称为 Inversion of Control，翻译为“控制反转”，它还有一个别名为 DI（Dependency Injection ）,即依赖注入。

IoC，就是由 Spring IoC 容器来负责对象的生命周期和对象之间的关系。

在传统的开发模式下，程序员都是采用直接 new一个对象的方式来创建对象，也就是说依赖的对象直接由程序员自己控制，但是有了 IoC容器后，则直接由 IoC 容器来控制对象的创建。没有 IoC 的时候程序员都是在自己对象中主动去创建被

依赖的对象，这是正转。但是有了IoC 后，所依赖的对象直接由 IoC容器创建后注入到被注入的对象中，依赖的对象由原来的主动获取变成被动接受，所以是反转。

DI

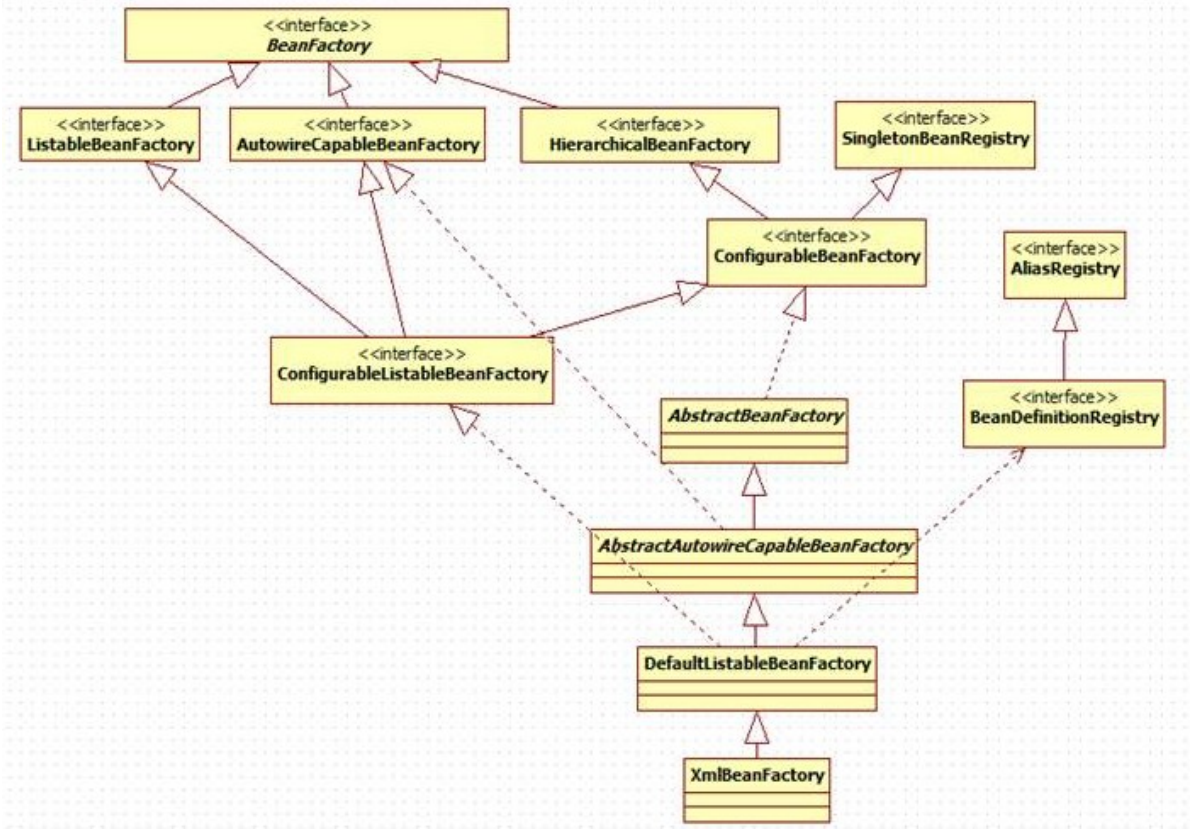
IoC的一个重点是在系统运行中，动态的向某个对象提供它所需要的其他对象。这一点是通过DI（Dependency Injection，依赖注入）来实现的。

IoC和DI由什么**关系**呢？其实它们**是同一个概念的不同角度描述**，由于控制反转概念比较含糊（可能只是理解为容器控制对象这一个层面，很难让人想到谁来维护对象关系），所以2004年大师级人物MartinFowler又给出了一个新的名字：“依赖注入”，相对IoC
而言，**“依赖注入”明确描述了“被注入对象依赖IoC 容器配置依赖对象”。**

Spring重要接口详解

BeanFactory继承体系

体系结构图



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

四级接口继承体系：

1. **BeanFactory 作为一个主接口不继承任何接口，暂且称为一级接口。**
2. AutowireCapableBeanFactory、HierarchicalBeanFactory、ListableBeanFactory3个子接口继承了它，进行功能上的增强。这3个子接口称为**二级接口**。
3. ConfigurableBeanFactory 可以被称为**三级接口**，对二级接口HierarchicalBeanFactory 进行了再次增强，它还继承了另一个外来的接口 SingletonBeanRegistry
4. ConfigurableListableBeanFactory是一个更强大的接口，继承了上述的所有接口，无所不包，称为**四级接口**。

总结：

| -- BeanFactory 是Spring bean容器的**根接口。**

最主要的方法就是getBean(StringbeanName),提供获取bean,是否包含bean,是否单例与原型,获取bean类型 ,bean 别名的 api.

| -- -- AutowireCapableBeanFactory 提供工厂的装配功能。

| -- -- HierarchicalBeanFactory 提供父容器的访问功能。

| -- -- -- ConfigurableBeanFactory 如名,提供factory的配置功能,好多api。

| -- -- -- -- ConfigurableListableBeanFactory集大成者,提供解析,修改bean定义,并初始化单例。

| -- -- ListableBeanFactory提供容器内bean实例的枚举功能. 这边不会考虑父容器内的实例.

这个设计采用了设计模式原则里的**接口隔离原则。**

下面是继承关系的2个抽象类和2个实现类：

1. AbstractBeanFactory 作为一个抽象类，实现了三级接口ConfigurableBeanFactory大部分功能。
2. AbstractAutowireCapableBeanFactory 同样是抽象类，继承自 AbstractBeanFactory，并额外实现了二级接口AutowireCapableBeanFactory 。
3. DefaultListableBeanFactory 继承自AbstractAutowireCapableBeanFactory，实现了最强大的四级接口 ConfigurableListableBeanFactory，并实现了一个外来接口BeanDefinitionRegistry ， 它并非抽象类。

4. 最后是最强大的 XmlBeanFactory ，继承自 DefaultListableBeanFactory，重写了一些功能， 使自己更强大。

定义这么多层次的接口的原因

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

BeanFactory

```
1  package org.springframework.beans.factory;
2
3
4  public interface BeanFactory {
5
6      //用来引用一个实例，或把它和工厂产生的Bean区分开
7      //就是说，如果一个FactoryBean的名字为a，那么，&a会得到那个Factory
8      String FACTORY_BEAN_PREFIX = "&";
9
10
11     /**
12      * 通过bean 去容器中获取一个bean对象
```

```
13      * @param name 名称(有可能是bean的真正的名称 也有可能是工厂bean的名称 也有可能是bean的别名)
14      * @return 返回bean实例
15      * @throws NoSuchBeanDefinitionException 抛出的异常，容器中没有该bean的定义
16      * @throws BeansException if the bean could not be obtained
17      */
18     Object getBean(String name) throws BeansException;
19
20     /**
21      * 通过bean 去容器中获取一个bean对象
22      * @param name 名称(有可能是bean的真正的名称 也有可能是工厂bean的名称 也有可能是bean的别名)
23      * @param requiredType 去容器中获取Bean的class类型 可以是实现的接口和父类
24      * @return 返回Bean对应的实例
25      * @throws NoSuchBeanDefinitionException 抛出的异常，容器中没有该bean的定义
26      * @throws BeanNotOfRequiredTypeException 容器中没有指定class类型的bean
27      * @throws BeansException bean还没有被创建
28      */
29     <T> T getBean(String name, @Nullable Class<T> requiredType) throws BeansException;
30
31     /**
32      *通过bean 去容器中获取一个bean对象
33      * @param name 名称(有可能是bean的真正的名称 也有可能是工厂bean的名称 也有可能是bean的别名)
```

```
34      * @param args 这个是用于指定传入获取bean的构造函数的参数,通过传入该参数，那么就可以明确的知道去调用哪个构造函数是实例化对象
35      * @return 返回Bean对应的实例
36      * @throws NoSuchBeanDefinitionException 抛出的异常，容器中没有该bean的定义
37      * @throws BeanDefinitionStoreException 主要用于多例模式下的构造器创建？？？？暂时没理解
38      * @throws BeansException bean还没有被创建
39      */
40     Object getBean(String name, Object... args) throws BeansException;
41
42     /**
43      *通过指定的bean的类型去容器中获取对象   若容器中有多个想同类型的bean
44      * 我们通过ctx.getBean(beanType.class) 就会抛出异常
45      * @param requiredType class对象需要的类型
46      * @return 返回的bean对象
47      * @throws NoSuchBeanDefinitionException 没有对应class的bean定义
48      * @throws NoUniqueBeanDefinitionException 找到多个匹配的
49      * @throws BeansException 创建bean的实例
50      * @since 3.0
51      * @see ListableBeanFactory
52      */
53     <T> T getBean(Class<T> requiredType) throws BeansException;
54
55     /**
56      *获取bean实例
```

```
57        * @param requiredType bean的类型
58        * @param args 构造函数的参数的类型，在实例化的过程中不需要去推断构造函数了
59        * @return bean实例
60        */
61       <T> T getBean(Class<T> requiredType, Object... args) throws BeansException;
62
63
64       /**
65        * 在获取之前需要判断是否已经存在
66        *判断我们的容器中是否包含了当前的bean对象
67        */
68       boolean containsBean(String name);
69
70       /**
71        *判断当前的bean是不是单例的
72        */
73       boolean isSingleton(String name) throws NoSuchBeanDefinitionException;
74
75       /**
76        *是不是原型的
77        */
78       boolean isPrototype(String name) throws NoSuchBeanDefinitionException;
79
80       /**
81        *是不是匹配的类型
82        */
83       boolean isTypeMatch(String name, ResolvableType typeToMatch) throws NoSuchBeanDefinitionException;
```

```
84
85      /**
86       *是不是匹配的类型
87       */
88      boolean isTypeMatch(String name,
    @Nullable Class<?> typeToMatch) throws
    NoSuchBeanDefinitionException;
89
90      /**
91       *通过beanName获取对应bean的class类型
92       */
93      @Nullable
94      Class<?> getType(String name) throws
    NoSuchBeanDefinitionException;
95
96      /**
97       * 根据实例的名字获取实例的别名
98       */
99      String[] getAliases(String name);
```

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

源码说明：

- 4个获取实例的方法。getBean的重载方法。
- 4个判断的方法。判断是否存在，是否为单例、原型，名称类型是否匹配。
- 1个获取类型的方法、一个获取别名的方法。根据名称获取类型、根据名称获取别名。一目了然！

总结:

- 这10个方法，很明显，这是一个典型的工厂模式的工厂接口。

ListableBeanFactory

可将Bean逐一列出的工厂

```
 public interface ListableBeanFactory extends BeanFactory {

    // 对于给定的名字是否含有
    boolean containsBeanDefinition(String beanName); BeanDefinition
    // 返回工厂的BeanDefinition总数
    int getBeanDefinitionCount();
    // 返回工厂中所有Bean的名字
    String[] getBeanDefinitionNames();

    String[] getBeanNamesForType(ResolvableType type);

   String[] getBeanNamesForType(ResolvableType type, boolean includeNonSingletons, boolean allowEagerInit);
     // 返回对于指定类型Bean（包括子类）的所有名字
    String[] getBeanNamesForType(@Nullable Class<?> type);
     /*
    *返回指定类型的名字
    *includeNonSingletons为false表示只取单例Bean，true则不是
```

```
18      *allowEagerInit为true表示立刻加载，false表示延迟加载。
19      * 注意：FactoryBeans都是立刻加载的。
20      */
21      String[] getBeanNamesForType(@Nullable Class<?> type, boolean includeNonSingletons, boolean allowEagerInit);
22
23
24
25      // 根据类型（包括子类）返回指定Bean名和Bean的Map
26      <T> Map<String, T> getBeansOfType(@Nullable Class<T> type) throws BeansException;
27
28
29      <T> Map<String, T> getBeansOfType(@Nullable Class<T> type, boolean includeNonSingletons, boolean allowEagerInit)
30              throws BeansException;
31
32          //查找使用注解的类
33      String[] getBeanNamesForAnnotation(Class<? extends Annotation> annotationType);
34
35      // 根据注解类型，查找所有有这个注解的Bean名和Bean的Map
```

```
36      Map<String, Object>
getBeansWithAnnotation(Class<? extends
Annotation> annotationType) throws
BeansException;
37
38      // 根据指定Bean名和注解类型查找指定的Bean
39      @Nullable
40      <A extends Annotation> A
findAnnotationOnBean(String beanName,
Class<A> annotationType)
41              throws
NoSuchBeanDefinitionException;
42
43  }
44
```

源码说明：

- 3个跟BeanDefifinition有关的总体操作。包括BeanDefifinition的总数、名字的集合、指定类型的名字的集合。

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

- 2个getBeanNamesForType重载方法。根据指定类型（包括子类）获取其对应的所有Bean名字。
- 2个getBeansOfType重载方法。根据类型（包括子类）返回指定Bean名和Bean的Map。

- 2个跟注解查找有关的方法。根据注解类型，查找Bean名和Bean的Map。以及根据指定Bean名和
  注解类型查找指定的Bean。

总结：

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

HierarchicalBeanFactory

分层的Bean工厂

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

参数说明：

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

总结:

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

AutowireCapableBeanFactory

自动装配的Bean工厂

```
1  public interface AutowireCapableBeanFactory
   extends BeanFactory {
2      // 这个常量表明工厂没有自动装配的Bean
3      int AUTOWIRE_NO = 0;
4      // 表明根据名称自动装配
5      int AUTOWIRE_BY_NAME = 1;
6      // 表明根据类型自动装配
7      int AUTOWIRE_BY_TYPE = 2;
8      // 表明根据构造方法快速装配
9      int AUTOWIRE_CONSTRUCTOR = 3;
10     //表明通过Bean的class的内部来自动装配（有没翻
   译错...）Spring3.0被弃用。
11     @Deprecated
12     int AUTOWIRE_AUTODETECT = 4;
13     // 根据指定Class创建一个全新的Bean实例
14     <T> T createBean(Class<T> beanClass)
   throws BeansException;
```

```
15      // 给定对象，根据注释、后处理器等，进行自动装配
16      void autowireBean(Object existingBean)
throws BeansException; 17
17        // 根据Bean名的BeanDefinition装配这个未加
工的Object，执行回调和各种后处理器。
18          Object configureBean(Object
existingBean, String beanName) throws
BeansException;
19
20          // 分解Bean在工厂中定义的这个指定的依赖
descriptor
21          Object
resolveDependency(DependencyDescriptor
descriptor, String beanName) throws
BeansException;
22
23          // 根据给定的类型和指定的装配策略，创建一个
新的Bean实例
24          Object createBean(Class<?>
beanClass, int autowireMode, boolean
dependencyCheck) throws BeansException;
25
26      // 与上面类似，不过稍有不同。
27          Object autowire(Class<?> beanClass,
int autowireMode, boolean dependencyCheck)
throws BeansException;
28
29      /*
30      * 根据名称或类型自动装配
31      */
32          void autowireBeanProperties(Object
existingBean, int autowireMode, boolean
dependencyCheck) throws BeansException;
```

```
33
34      /*
35      * 也是自动装配
36      */
37      void applyBeanPropertyValues(Object
existingBean, String beanName) throws
BeansException;
38      /*
39      * 初始化一个Bean...
40      */
41          Object initializeBean(Object
existingBean, String beanName) throws
BeansException;
42      /*
43      * 初始化之前执行BeanPostProcessors
44          */
45      Object
applyBeanPostProcessorsBeforeInitialization(
Object existingBean,String beanName) throws
BeansException;
46     /*
47      * 初始化之后执行BeanPostProcessors
48      */
49      Object
applyBeanPostProcessorsAfterInitialization(O
bject existingBean, String beanName)
50 throws BeansException;
51  /**
52       * 销毁参数中指定的Bean，同时调用此Bean上的
DisposableBean和
DestructionAwareBeanPostProcessors方法
53       * 在销毁途中，任何的异常情况都只应该被直接捕获
和记录，而不应该向外抛出。
```

```
54          */
55         void destroyBean(Object existingBean);
56 
57 
58         /**
59          * 查找唯一符合指定类的实例，如果有，则返回实例的名字和实例本身
60          * 和BeanFactory中的getBean(Class)方法类似，只不过多加了一个bean的名字
61          */
62         <T> NamedBeanHolder<T> resolveNamedBean(Class<T> requiredType) throws BeansException;
63         Object resolveBeanByName(String name, DependencyDescriptor descriptor) throws BeansException;
64         Object resolveDependency(DependencyDescriptor descriptor, @Nullable String requestingBeanName) throws BeansException;
65 
66         /*
67         * 分解指定的依赖
68         */
69         Object resolveDependency(DependencyDescriptor descriptor, String beanName, Set<String> autowiredBeanNames, TypeConverter typeConverter) throws BeansException;
70 }
71 
```

源码说明：

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

总结:

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

ConfigurableBeanFactory

复杂的配置Bean工厂

```
public interface ConfigurableBeanFactory extends HierarchicalBeanFactory, SingletonBeanRegistry {
         String SCOPE_SINGLETON = "singleton"; // 单例
        String SCOPE_PROTOTYPE = "prototype"; // 原型
        /*
         * 搭配HierarchicalBeanFactory接口的getParentBeanFactory方法
         */
       void setParentBeanFactory(BeanFactory parentBeanFactory) throws IllegalStateException;
```

```
8         /*
9          * 设置、返回工厂的类加载器
10         */
11         void setBeanClassLoader(ClassLoader
beanClassLoader);
12         ClassLoader getBeanClassLoader();
13         /*
14         * 设置、返回一个临时的类加载器
15         */
16         void setTempClassLoader(ClassLoader
tempClassLoader);
17         ClassLoader getTempClassLoader();
18         /*
19         * 设置、是否缓存元数据，如果false，那么每
次请求实例，都会从类加载器重新加载（热加
20         载）
21         */
22         void setCacheBeanMetadata(boolean
cacheBeanMetadata);
23         boolean isCacheBeanMetadata();//是否
缓存元数据
24         /*
25         * Bean表达式分解器
26         */
27         void
setBeanExpressionResolver(BeanExpressionReso
lver resolver);
28         BeanExpressionResolver
getBeanExpressionResolver();
29         /*
30         * 设置、返回一个转换服务
31         */
```

```
32          void
setConversionService(ConversionService
conversionService);
33          ConversionService
getConversionService();
34          /*
35          * 设置属性编辑登记员...
36          */
37          void
addPropertyEditorRegistrar(PropertyEditorReg
istrar registrar);
38          /*
39          * 注册常用属性编辑器
40          */
41          void registerCustomEditor(Class<?>
requiredType, Class<? extends
42          PropertyEditor>
propertyEditorClass);
43          /*
44          * 用工厂中注册的通用的编辑器初始化指定的属
性编辑注册器
45          */
46          void
copyRegisteredEditorsTo(PropertyEditorRegist
ry registry);
47          /*
48          * 设置、得到一个类型转换器
49          */
50          void setTypeConverter(TypeConverter
typeConverter);
51          TypeConverter getTypeConverter();
52          /*
53          * 增加一个嵌入式的StringValueResolver
```

```
54          */
55          void
addEmbeddedValueResolver(StringValueResolver
valueResolver);
56          String resolveEmbeddedValue(String
value);//分解指定的嵌入式的值
57          void
addBeanPostProcessor(BeanPostProcessor
beanPostProcessor);//设置一个Bean后处理器
58          int getBeanPostProcessorCount();//返
回Bean后处理器的数量
59          void registerScope(String scopeName,
Scope scope);//注册范围
60          String[]
getRegisteredScopeNames();//返回注册的范围名
61          Scope getRegisteredScope(String
scopeName);//返回指定的范围
62          AccessControlContext
getAccessControlContext();//返回本工厂的一个安全
访问上下文
63          void
copyConfigurationFrom(ConfigurableBeanFactor
y otherFactory);//从其他的工厂复制相关的所有配置
64          /*
65          * 给指定的Bean注册别名
66          */
67          void registerAlias(String beanName,
String alias) throws
BeanDefinitionStoreException;
68          void
resolveAliases(StringValueResolver
valueResolver);//根据指定的StringValueResolver
移除所有的别名
```

```
69
70          /*
71          * 返回指定Bean合并后的Bean定义
72          */
73          BeanDefinition
getMergedBeanDefinition(String beanName)
throws NoSuchBeanDefinitionException;
74          boolean isFactoryBean(String name)
throws NoSuchBeanDefinitionException;//判断指
定Bean是否为一个工厂Bean
75          void setCurrentlyInCreation(String
beanName, boolean inCreation);//设置一个Bean是
否正在创建
76          boolean isCurrentlyInCreation(String
beanName);//返回指定Bean是否已经成功创建
77          void registerDependentBean(String
beanName, String dependentBeanName);//注册一
个依赖于指定bean的Bean
78          String[] getDependentBeans(String
beanName);//返回依赖于指定Bean的所欲Bean名
79          String[]
getDependenciesForBean(String beanName);//返
回指定Bean依赖的所有Bean名
80          void destroyBean(String beanName,
Object beanInstance);//销毁指定的Bean
81          void destroyScopedBean(String
beanName);//销毁指定的范围Bean
82          void destroySingletons(); //销毁所有的
单例类83
83  }
```

ConfigurableListableBeanFactory

BeanFactory的集大成者

```
1  public interface
   ConfigurableListableBeanFactory extends
   ListableBeanFactory,
   AutowireCapableBeanFactory,
   ConfigurableBeanFactory {
2
3      void ignoreDependencyType(Class<?>
   type);//忽略自动装配的依赖类型
4      void ignoreDependencyInterface(Class<?>
   ifc);//忽略自动装配的接口
5      /*
6      * 注册一个可分解的依赖
7      */
8      void
   registerResolvableDependency(Class<?>
   dependencyType, Object autowiredValue);
9      /*
10     * 判断指定的Bean是否有资格作为自动装配的候选者
11     */
12     boolean isAutowireCandidate(String
   beanName, DependencyDescriptor descriptor)
   throws NoSuchBeanDefinitionException;
13     // 返回注册的Bean定义
14     BeanDefinition getBeanDefinition(String
   beanName) throws
   NoSuchBeanDefinitionException;
15     // 暂时冻结所有的Bean配置
16     void freezeConfiguration();
17     // 判断本工厂配置是否被冻结
```

```
18      boolean isConfigurationFrozen();
19      // 使所有的非延迟加载的单例类都实例化。
20     void preInstantiateSingletons() throws
BeansException;
21  }
```

源码说明：

1、2个忽略自动装配的的方法。

2、1个注册一个可分解依赖的方法。

3、1个判断指定的Bean是否有资格作为自动装配的候选者的方法。

4、1个根据指定bean名，返回注册的Bean定义的方法。

5、2个冻结所有的Bean配置相关的方法。

6、1个使所有的非延迟加载的单例类都实例化的方法。

总结：

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

BeanDefinitionRegistry

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

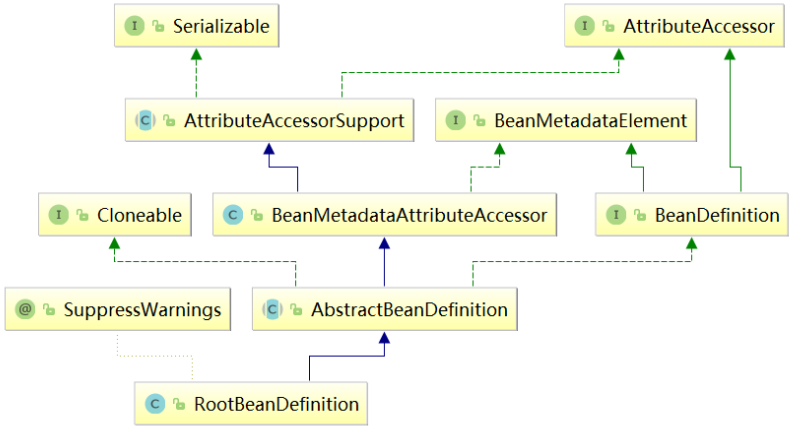
```
1  public interface BeanDefinitionRegistry extends AliasRegistry {
2      // 给定bean名称，注册一个新的bean定义
3      void registerBeanDefinition(String beanName, BeanDefinition beanDefinition) throws BeanDefinitionStoreException;
4      /*
5      * 根据指定Bean名移除对应的Bean定义
6      */
7      void removeBeanDefinition(String beanName) throws NoSuchBeanDefinitionException;
8      /*
9      * 根据指定bean名得到对应的Bean定义
10     */
11     BeanDefinition getBeanDefinition(String beanName) throws NoSuchBeanDefinitionException;
12     /*
13     * 查找，指定的Bean名是否包含Bean定义
14     */
15     boolean containsBeanDefinition(String beanName);
16     String[] getBeanDefinitionNames();//返回本容器内所有注册的Bean定义名称int getBeanDefinitionCount();//返回本容器内注册的Bean定义数目
```

```
17      boolean isBeanNameInUse(String
    beanName);//指定Bean名是否被注册过。
18  }
19
```

BeanDefinition继承体系

体系结构图

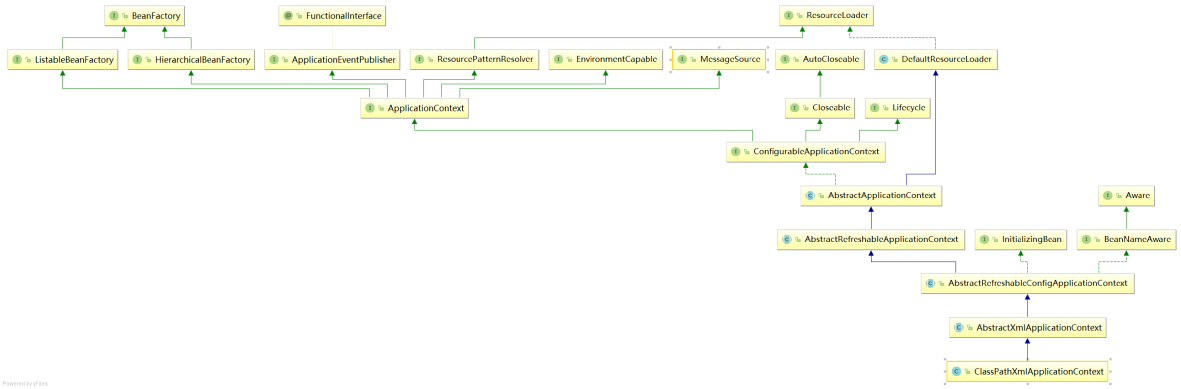
SpringIOC容器管理了我们定义的各种Bean对象及其相互的关系，Bean对象在Spring实现中是以BeanDefifinition来描述的，其继承体系如下：



Bean的解析过程非常复杂，功能被分的很细，因为这里需要被扩展的地方很多，必须保证有足够的灵活性，以应对可能的变化。Bean的解析主要就是对 Spring 配置文件的解析。这个解析过程主要通过下图中的类完成：

ApplicationContext继承体系

体系结构图



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

BeanFactory和ApplicationContext区别：

联系：BeanFactory和ApplicationContext都是Spring的顶级接口 、 **ApplicationContext**是继承了BeanFactory接口

区别： BeanFactory它产生对象是通过懒汉模式来产生 **ApplicationContext**产生对象是在应用启动的时候，一次性将（非懒加载的单例bean）全部创建出来

容器初始化流程源码分析

IoC容器的初始化包括BeanDefinition的Resource定位、载入和注册这三个基本的过程。

主流程源码分析

找入口

- java程序入口

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

- web程序入口

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

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

流程解析

AbstractApplicationContext的 refresh 方法

```

```

```
public void refresh() throws BeansException, IllegalStateException {


    synchronized (this.startupShutdownMonitor) {
    // Prepare this context for refreshing.
    // STEP 1:  刷新预处理
    prepareRefresh();
    // Tell the subclass to refresh the internal bean factory.
    // STEP 2:
    // a） 创建IoC容器（DefaultListableBeanFactory）
    // b） 加载解析XML文件（最终存储到Document对象中）
    // c） 读取Document对象，并完成BeanDefinition的加载和注册工作
    ConfigurableListableBeanFactory beanFactory =
    obtainFreshBeanFactory();
    // Prepare the bean factory for use in this context.
    // STEP 3:  对IoC容器进行一些预处理（设置一些公共属性）
    prepareBeanFactory(beanFactory);
    try {
    // Allows post-processing of the bean factory in context
    subclasses.
    // STEP 4:
    postProcessBeanFactory(beanFactory);
```

```
23      // Invoke factory processors registered
    as beans in the context.
24      // STEP 5:  调用BeanFactoryPostProcessor
    后置处理器对 BeanDefinition
25      处理
26
     invokeBeanFactoryPostProcessors(beanFactory
    );
27      // Register bean processors that
    intercept bean creation.
28      // STEP 6:  注册BeanPostProcessor后置处理器
29      registerBeanPostProcessors(beanFactory);
30      // Initialize message source for this
    context.
31      // STEP 7:  初始化一些消息源（比如处理国际化的
    i18n等消息源）
32      initMessageSource();
33      // Initialize event multicaster for this
    context.
34      // STEP 8:  初始化应用事件广播器
35      initApplicationEventMulticaster();
36      // Initialize other special beans in
    specific context
37      subclasses.
38      // STEP 9:  初始化一些特殊的bean
39      onRefresh();
40      // Check for listener beans and register
    them.
41      // STEP 10:  注册一些监听器
42      registerListeners();
43      // Instantiate all remaining (non-lazy-
    init) singletons.
```

```
44      // STEP 11:  实例化剩余的单例bean（非懒加载方式）
45      // 注意事项：Bean的IoC、DI和AOP都是发生在此步骤
46
 finishBeanFactoryInitialization(beanFactory);
47      // Last step: publish corresponding event.
48      // STEP 12:  完成刷新时，需要发布对应的事件
49      finishRefresh();
50      }catch (BeansException ex) {
51      if (logger.isWarnEnabled()) {
52      logger.warn("Exception encountered during context
53      initialization - " +
54      "cancelling refresh attempt: " + ex);
55      }
56      // Destroy already created singletons to avoid dangling
57      resources.
58      destroyBeans();
59      // Reset 'active' flag.
60      cancelRefresh(ex);
61      // Propagate exception to caller.
62      throw ex;
63      }
64      finally {
65      // Reset common introspection caches in Spring's core, since we
66      // might not ever need metadata for singleton beans anymore...
67          resetCommonCaches();
```

```
68      }
69     }
70 }
```

创建BeanFactory流程源码分析

找入口

AbstractApplicationContext类的 refresh 方法：

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

流程解析

进入**AbstractApplicationContext**的 obtainFreshBeanFactory 方法：用于创建一个新的 IoC容器，这个 IoC容器
就是**DefaultListableBeanFactory**对象。

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

进入**AbstractRefreshableApplicationContext**的 refreshBeanFactory 方法：

- 销毁以前的容器
- 创建新的 IoC容器
- 加载 BeanDefinition 对象注册到IoC容器中

```
protected final void refreshBeanFactory() throws BeansException {

    // 如果之前有IoC容器，则销毁
    if (hasBeanFactory()) {
        destroyBeans(); closeBeanFactory();
    }
    try {
    // 创 建 IoC 容 器 ， 也 就 是 DefaultListableBeanFactory
```

```
9           DefaultListableBeanFactory beanFactory = createBeanFactory();
10
 beanFactory.setSerializationId(getId());
11          customizeBeanFactory(beanFactory);
12      // 加载BeanDefinition对象，并注册到IoC容器中（重点）
13          loadBeanDefinitions(beanFactory);
14          synchronized (this.beanFactoryMonitor) {
15              this.beanFactory = beanFactory;
16          }
17      }
18      catch (IOException ex) {
19          throw new ApplicationContextException("I/O error parsing bean definition source for " + getDisplayName(), ex);
20      }
21  }
```

进入**AbstractRefreshableApplicationContext**的createBeanFactory 方法

```
protected DefaultListableBeanFactory createBeanFactory() {

    return new DefaultListableBeanFactory(getInternalParentBeanFactory());

}
```

加载BeanDefinition流程分析

找入口

AbstractRefreshableApplicationContext类的 refreshBeanFactory 方法中第13行代码：

```
protected final void refreshBeanFactory() throws BeansException {
        // 如果之前有IoC容器，则销毁
    if (hasBeanFactory()) { destroyBeans(); closeBeanFactory();
    }
    try {
         // 创 建 IoC 容 器 ， 也 就 是 DefaultListableBeanFactory
        DefaultListableBeanFactory beanFactory = createBeanFactory();

 beanFactory.setSerializationId(getId());
```

```
        customizeBeanFactory(beanFactory);
         // 加载BeanDefinition对象，并注册到IoC容器中（重点）
        loadBeanDefinitions(beanFactory);
        synchronized (this.beanFactoryMonitor) {
          this.beanFactory = beanFactory;
        }
    }
    catch (IOException ex) {
         throw new ApplicationContextException("I/O error parsing bean definition source for " + getDisplayName(), ex);
    }
}
```

BeanDefifinition流程

```
|--AbstractRefreshableApplicationContext#refreshBeanFactory
    |--AbstractXmlApplicationContext#loadBeanDefinitions：走了多个重载方法

        |--AbstractBeanDefinitionReader#loadBeanDefinitions：走了多个重载方法

```

```
6              |--
XmlBeanDefinitionReader#loadBeanDefinitions
：走了多个重载方法
7
8                  |--
XmlBeanDefinitionReader#doLoadBeanDefinition
s
9
10                     |--
XmlBeanDefinitionReader#registerBeanDefiniti
ons
11
12                         |--
DefaultBeanDefinitionDocumentReader#register
BeanDefinitions
13                             |--
#doRegisterBeanDefinitions
14                                 |--
#parseBeanDefinitions
15                                   |--
#parseDefaultElement
16                                       |--
#processBeanDefinition
17
|--
BeanDefinitionParserDelegate#parseBeanDefini
tionElement
18
 |--#parseBeanDefinitionElement
19
```

流程相关类的说明

AbstractRefreshableApplicationContext

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

AbstractXmlApplicationContext

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

AbstractBeanDefinitionReader

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

XmlBeanDefinitionReader

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

DefaultBeanDefinitionDocumentReader

BeanDefinitionParserDelegate

流程解析

进入**AbstractXmlApplicationContext**的loadBeanDefifinitions方法：

- 创建一个**XmlBeanDefinitionReader**，通过阅读XML文件，真正完成BeanDefifinition的加载和注册。
- 配置**XmlBeanDefinitionReader**并进行初始化。

- 委托给**XmlBeanDefinitionReader**去加载BeanDefifinition。

```
protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws BeansException, IOException {
        // Create a new XmlBeanDefinitionReader for the given BeanFactory.
        // 给指定的工厂创建一个BeanDefinition阅读器
        // 作用：通过阅读XML文件，真正完成BeanDefinition的加载和注册
            XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);
            // Configure the bean definition reader with this context's
        // resource loading environment.

 beanDefinitionReader.setEnvironment(this.getEnvironment());

 beanDefinitionReader.setResourceLoader(this);

 beanDefinitionReader.setEntityResolver(new
        ResourceEntityResolver(this)); 13
        // Allow a subclass to provide custom initialization of the reader,
```

```
13          // then proceed with actually
loading the bean definitions.
14
 initBeanDefinitionReader(beanDefinitionRe
ader); 17
15          // 委托给BeanDefinition阅读器去加载
BeanDefinition
16
 loadBeanDefinitions(beanDefinitionReader)
; 20    }
17
18          protected void
loadBeanDefinitions(XmlBeanDefinitionReade
r reader) throws
19          BeansException, IOException {
20          // 获取资源的定位
21          // 这里getConfigResources是一个空实
现，真正实现是调用子类的获取资源定位的方法
22          // 比如:
ClassPathXmlApplicationContext中进行了实现
23          // 而
FileSystemXmlApplicationContext没有使用该方
法
24          Resource[] configResources =
getConfigResources();
25          if (configResources != null) {
26          // XML Bean读取器调用其父类
AbstractBeanDefinitionReader读取定位的资源
27
 reader.loadBeanDefinitions(configResource
s); 32 }
```

```
            //如果子类中获取的资源定位为空，则获取FileSystemXmlApplicationContext构造方法中
            setConfigLocations方法设置的资源

            String[] configLocations = getConfigLocations();
            if (configLocations != null) {
            // XML Bean读取器调用其父类AbstractBeanDefinitionReader读取定位的资源

            reader.loadBeanDefinitions(configLocations);
            }
        }
```

loadBeanDefinitions 方法经过一路的兜兜转转，最终来到了**XmlBeanDefifinitionReader**的

doLoadBeanDefinitions 方法：

- 一个是对XML文件进行DOM解析；
- 一个是完成BeanDefinition对象的加载与注册 。

```
protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)

 throws BeanDefinitionStoreException {
    try {
    // 通过DOM4J加载解析XML文件，最终形成Document对象
```

```
6
7       Document doc = doLoadDocument(inputSource, resource);
8
9       // 通过对Document对象的操作，完成BeanDefinition的加载和注册工作
10
11      return registerBeanDefinitions(doc, resource); 8 }
12
13      //省略一些catch语句
14
15      catch (Throwable ex) {
16          ......
17
18      }
19  }
```

此处我们暂不处理**DOM4J**加载解析XML的流程，我们重点分析**BeanDefifinition**的加载注册流程进入

XmlBeanDefifinitionReader的 registerBeanDefinitions 方法:

- 创建**DefaultBeanDefifinitionDocumentReader**用来解析Document对象。
- 获得容器中已注册的BeanDefifinition数量
- 委托给**DefaultBeanDefifinitionDocumentReader**来完成BeanDefifinition的加载、注册工

  作。

- 统计新注册的BeanDefifinition数量

```
1  public int
registerBeanDefinitions(Document doc,
Resource resource) throws
2  BeanDefinitionStoreException {
3      // 创建
DefaultBeanDefinitionDocumentReader用来解析
Document对象
4       BeanDefinitionDocumentReader
documentReader =
createBeanDefinitionDocumentReader();
5      // 获得容器中注册的Bean数量
6      int countBefore =
getRegistry().getBeanDefinitionCount();
7      //解析过程入口，
BeanDefinitionDocumentReader只是个接口
8      //具体的实现过程在
DefaultBeanDefinitionDocumentReader完成
9
 documentReader.registerBeanDefinitions(do
c, createReaderContext(resource));
10     // 统计注册的Bean数量
11     return
getRegistry().getBeanDefinitionCount() -
countBefore;
12 }
```

进入**DefaultBeanDefifinitionDocumentReader**的registerBeanDefinitions方法：

- 获得Document的根元素标签

- 真正实现BeanDefifinition解析和注册工作

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

进入**DefaultBeanDefifinitionDocumentReader**
doRegisterBeanDefinitions
方法:

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

```
protected void doRegisterBeanDefinitions(Element root) {
```

```
    // Any nested <beans> elements will cause recursion in this method.In
    // order to propagate and preserve <beans> default-* attributes correctly,
    // keep track of the current (parent) delegate, which may be null.Create
    // the new (child) delegate with a reference to the parent for fallback purposes,
    // then ultimately reset this.delegate back to its original (parent) reference.
    // this behavior emulates a stack of delegates without actually necessitating one.
    // 这里使用了委托模式，将具体的BeanDefinition解析工作交给了BeanDefinitionParserDelegate去完成
    BeanDefinitionParserDelegate parent = this.delegate;
    this.delegate = createDelegate(getReaderContext(), root, parent);
    if (this.delegate.isDefaultNamespace(root)) {
        String profileSpec = root.getAttribute(PROFILE_ATTRIBUTE);
        if (StringUtils.hasText(profileSpec)) {
            String[] specifiedProfiles =

 StringUtils.tokenizeToStringArray(profileSpec,BeanDefinitionParserDelegate.MULTI_VALUE_ATTRIBUTE_DELIMITERS);
```

```
16
 if(!getReaderContext().getEnvironment().acceptsProfiles(specifiedProfiles){
17                 if (logger.isInfoEnabled()){
18                     logger.info("Skipped XML bean definition file due to specified profiles[" +profileSpec + "] not matching: " +
19 
 getReaderContext().getResource());
20                 }
21                 return;
22             }
23                 }
24                 }
25     // 在解析Bean定义之前，进行自定义的解析，增强解析过程的可扩展性
26     preProcessXml(root);
27     // 委托给BeanDefinitionParserDelegate,从Document的根元素开始进行
28     BeanDefinition的解析
29     parseBeanDefinitions(root, this.delegate);
30     // 在解析Bean定义之后，进行自定义的解析，增加解析过程的可扩展性
31     postProcessXml(root);
32     this.delegate = parent;
33 }
34 
```

Bean实例化流程分析

找入口

AbstractApplicationContext类的 refresh 方法

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

预实例化所有的非懒加载的单例bean

```
protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory){
    // Stop using the temporary ClassLoader for type matching.
    beanFactory.setTempClassLoader(null);
    // Allow for caching all bean definition metadata, not expecting further changes.
    beanFactory.freezeConfiguration();
    // Instantiate all remaining (non-lazy-init) singletons.
    // 预实例化所有的非懒加载的单例bean
    beanFactory.preInstantiateSingletons();
}
```

preInstantiateSingletons方法是DefaultListableBeanFactory中实现,预实例化所有的单例bean

```
  public void preInstantiateSingletons() throws BeansException {
        if (logger.isTraceEnabled()) {

            logger.trace("Pre-instantiating singletons in " + this);

        }

        // Iterate over a copy to allow for init methods which in turn register newbean definitions.

        // While this may not be part of the regular factory bootstrap, it doesotherwise work fine.

        //拿到beanDefinitionNames集合中所有的beanName，之前在解析xml或者注解的时候注册进去的

        List<String> beanNames = new ArrayList\<\(this.beanDefinitionNames);

        // Trigger initialization of all non-lazy singleton beans...
        for (String beanName : beanNames) {
            // 获取已经merge的RootBeanDefinition对象
```

```
19              RootBeanDefinition bd =
getMergedLocalBeanDefinition(beanName);
20              // 如果beanDefinition不是抽象bean
是单例 且不是懒加载，那么就会实例化
21              if (!bd.isAbstract() &&
bd.isSingleton() && !bd.isLazyInit()) {
22                  // 判断是否FactoryBean类
23                  if (isFactoryBean(beanName))
{
24                      // 如果是FactoryBean类型，
那么获取bean就需要 &+beanName的形式获取，否则获取
的是  getObject的实例
25                      Object bean =
getBean(FACTORY_BEAN_PREFIX + beanName);
26                      if (bean instanceof
FactoryBean) {
27                          final FactoryBean<?>
factory = (FactoryBean<?>) bean;
28                          boolean isEagerInit;
29                          if
(System.getSecurityManager() != null &&
factory instanceof SmartFactoryBean) {
30                              isEagerInit =
AccessController.doPrivileged((PrivilegedAct
ion<Boolean>) ((SmartFactoryBean<?>)
factory)::isEagerInit,
31
32
 getAccessControlContext());
33                          } else {
```

```
34                                  isEagerInit =
(factory instanceof SmartFactoryBean &&
((SmartFactoryBean<?>)
factory).isEagerInit());
35                              }
36                              if (isEagerInit) {
37
 getBean(beanName);
38                              }
39                          }
40                      }
41                      // 非factoryBean类
42                      else {
43                          getBean(beanName);
44                      }
45                  }
46              }
47
48              // Trigger post-initialization
callback for all applicable beans...
49              for (String beanName : beanNames) {
50                  Object singletonInstance =
getSingleton(beanName);
51                  if (singletonInstance instanceof
SmartInitializingSingleton) {
52                      final
SmartInitializingSingleton smartSingleton =
(SmartInitializingSingleton)
singletonInstance;
53                      if
(System.getSecurityManager() != null) {
```

```
54
 AccessController.doPrivileged((PrivilegedAction<Object>) () -> {
55
 smartSingleton.afterSingletonsInstantiated();
56                         return null;
57                     }, getAccessControlContext());
58                 } else {
59
 smartSingleton.afterSingletonsInstantiated();
60                 }
61             }
62         }
63   }
```

通过getBean方法获取实例

```
1 @Override
2 public Object getBean(String name) throws BeansException {
3     return doGetBean(name, null, null, false);
4 }
```

分析一下doGetBean方法

```
1
2       //真正实现向IOC容器获取Bean的功能，也是触发依赖注入 功能的地方    核心步骤是createBean
3       @SuppressWarnings("unchecked")
4       protected <T> T doGetBean(final String name, @Nullable final Class<T> requiredType,
5                   @Nullable final Object[] args, boolean typeCheckOnly) throws BeansException {
6
7           //根据指定的名称获取被管理Bean的名称，剥离指定名称中对容器的相关依赖
8           //如果指定的是别名，将别名 转换为规范的Bean名称
9           /*
10              1. 转换为对应的beanName
11              这里传入的参数可能是别名，也可能是FactoryBean，所以需要进行解析
12              -去除FactoryBean 的修饰符，也就是如果是name="&aa"，那么会首先去除&而使name="aa"。
13              -取指定alias 所表示的最终beanName
14           */
15          final String beanName = transformedBeanName(name);
16          Object bean;
17
18          // Eagerly check singleton cache for manually registered singletons. 急切地检查单例缓存中手动注册的单例。
19          /*
20              2. 尝试从缓存中加载单例
```

```
21                检查缓存中 或者 实例工厂中是否有对应的实例
22                为什么首先会使用这段代码呢？
23                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，
24                Spring创建bean的原则 是不等bean创建完成就 将创建bean的 ObjectFactory提早曝光
25                也就是将ObjectFactory加入到缓存中，（引入三级缓存  三个map，）
26                一旦下个bean创建时候需要依赖上个bean，则直接使用ObjectFactory
27            */
28            // 先直接尝试 从缓存或者 singletonFactories 中的ObjectFactory中获取
29            Object sharedInstance = getSingleton(beanName);
30            //IOC容器创建单例模式Bean实例对象 ，整个IOC容器中只创建一次
31            if (sharedInstance != null && args == null) {
32                //如果指定名称的Bean在容器中已有单例模式的Bean被创建
33                //直接返回已经创建的Bean
34                if (logger.isTraceEnabled()) {
35                    if (isSingletonCurrentlyInCreation(beanName)) {
36                        logger.trace("Returning eagerly cached instance of singleton bean '" + beanName +
37                                "' that is not fully initialized yet - a consequence of a circular reference");
```

```
38                  }
39                  else {
40                      logger.trace("Returning
cached instance of singleton bean '" +
beanName + "'");
41                  }
42              }
43              /*
44                  3. bean的实例化
45                  返回对应的实例，有时候存在诸如
BeanFactory的情况，
46                      并不是直接返回实例本身，而是
返回指定方法返回的实例
47               */
48              //获取给定Bean的实例对象，主要是完成
FactoryBean的相关处理
49              /*
50                  注意：BeanFactory是管理容器中
Bean的工厂，
51                  而FactoryBean是创建创建对象的工
厂Bean，两者之间有区别
52               */
53              bean =
getObjectForBeanInstance(sharedInstance,
name, beanName, null);
54          }
55          else {
56              // Fail if we're already
creating this bean instance:
57              // We're assumably within a
circular reference.
58          //缓存没有   正在创建的单例模式Bean
59              //缓存中已经有创建的原型模式Bean
```

```
60                //但是由于循环引用的问题导致实例化对象失败
61                /*
62                    4. 原型模式的依赖检查
63                    只有在单例情况才会尝试解决 循环依赖,
64                    原型模式情况下，如果存在A中有B的属性，B中有A的属性，
65                                那么在依赖注入的时候就会产生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。
66                                则抛出异常
67                 */
68                if (isPrototypeCurrentlyInCreation(beanName)) {
69                    throw new BeanCurrentlyInCreationException(beanName);
70                }
71
72                // Check if bean definition exists in this factory.
73                /*
74                    5. 检测parentBeanFactory
75                    对IOC容器中是否存在指定名称的BeanDefinition进行检查,
76                    首先检查是否能在当前的BeanFactory中获取的所需要的Bean,
77                    如果不能则 委托当前容器的父级容器去查找,
78                    如果还是找不到则沿着容器的继承体系向父级容器查找
79                 */
```

```
80              BeanFactory parentBeanFactory =
getParentBeanFactory();
81
82              //当前容器的父级容器存在，且当前容器
中不存在指定名称的Bean
83              // 如果在beanDefinitionMap中 也就
是在所有已经加载的类中不包括beanName，则尝试从
parentBeanFactory中检测
84              if (parentBeanFactory != null
&& !containsBeanDefinition(beanName)) {
85                  // Not found -> check
parent.
86                  //解析指定Bean名称的原始名称
87                  String nameToLookup =
originalBeanName(name);
88                  if (parentBeanFactory
instanceof AbstractBeanFactory) {
89                      // 递归到BeanFactory中寻
找
90                      return
((AbstractBeanFactory)
parentBeanFactory).doGetBean(
91                              nameToLookup,
requiredType, args, typeCheckOnly);
92                  }
93                  else if (args != null) {
94                      // Delegation to parent
with explicit args.
95                      // 委派父级容器 根据指定名
称和显式的参数查找
96                      return (T)
parentBeanFactory.getBean(nameToLookup,
args);
```

```
97                  }
98                  else if (requiredType !=
    null) {
99                      // No args -> delegate
    to standard getBean method.
100                     // 委派父级容器 根据指定名
    称和类型查找
101                     return
    parentBeanFactory.getBean(nameToLookup,
    requiredType);
102                 }
103                 else {
104                     // 委派父级容器根据指定名称
    查找
105                     return (T)
    parentBeanFactory.getBean(nameToLookup);
106                 }
107             }
108             //创建的Bean是否需要进行类型验证，一
    般不需要
109             if (!typeCheckOnly) {
110                 //向容器标记指定的Bean 为已经被
    创建 ；添加到alreadyCreated Set集合中
111
    markBeanAsCreated(beanName);
112             }
113
114             try {
115                 /*
116                     6. 将存储XML配置文件的
    GenericBeanDefinition 转换为
    RootBeanDefinition
```

```
117                         将存储XML配置文件的
GenericBeanDefinition 转换为
RootBeanDefinition，
118                         如果指定BeanName是子Bean
的话 ，会合并父类的相关属性
119                          */
120                        //根据指定Bean名称获取其父级的
Bean定义
121                        //主要解决Bean继承时子类合并父类
公共属性问题
122                        final RootBeanDefinition
mbd =
getMergedLocalBeanDefinition(beanName);
123
checkMergedBeanDefinition(mbd, beanName,
args);
124
125                        /*
126                            7.寻找依赖
127                            确保当前bean 依赖的bean的
初始化。
128                            首先获取当前Bean依赖关系
mbd.getDependsOn()
129                            接着根据依赖的BeanName递归
调用getBean()方法
130                            直到调用getSingleton()返
回依赖
131                          */
132                        // Guarantee initialization
of beans that the current bean depends on.
bean的依赖次序 属性
133                        String[] dependsOn =
mbd.getDependsOn();
```

```
134                     // 若存在依赖 则需要递归实例化依
    赖的bean
135                     if (dependsOn != null) {
136                         for (String dep :
    dependsOn) {
137                             if
    (isDependent(beanName, dep)) {
138                                 throw new
    BeanCreationException(mbd.getResourceDescri
    ption(), beanName,
139 
    "Circular depends-on relationship between
    '" + beanName + "' and '" + dep + "'");
140                             }
141                             // 缓存依赖调用
142 
    registerDependentBean(dep, beanName);
143                             try {
144                                 //递归调用
    getBean方法，获取当前Bean的依赖Bean
145                                 getBean(dep);
146                             }
147                             catch
    (NoSuchBeanDefinitionException ex) {
148                                 throw new
    BeanCreationException(mbd.getResourceDescri
    ption(), beanName,
149                                         "'" +
    beanName + "' depends on missing bean '" +
    dep + "'", ex);
150                             }
151                         }
152                     }
```

```
153
154                 // Create bean instance.
155                 /*
156                     8. 针对不同的scope 进行 bean 的创建  默认的是singleton
157                     实例化依赖的bean后 便可以实例化mbd本身了
158                  */
159                 // 8.1 创建单例模式Bean的实例对象
160                 if (mbd.isSingleton()) {
161                     //这里使用了一个匿名内部类，创建Bean实例对象，并且注册给所依赖的对象
                    ObjectFactory
162                     sharedInstance = getSingleton(beanName, () -> {
163                         try {
164                             // 直到调用getSingleton()返回依赖
165                             // 创建一个指定Bean实例对象，如果有父级继承，则合并子类和父类的定义
166                             // 一个真正干活的函数其实是以do 开头的
167                             return createBean(beanName, mbd, args);
168                         }
169                         catch (BeansException ex) {
170                             // Explicitly remove instance from singleton cache: It might have been put there
```

```
171                                 // eagerly by
the creation process, to allow for circular
reference resolution.
172                                 // Also remove
any beans that received a temporary
reference to the bean.
173
destroySingleton(beanName);
174                                 throw ex;
175                             }
176                         });
177                         //获取给定Bean的实例对象
178                         bean =
getObjectForBeanInstance(sharedInstance,
name, beanName, mbd);
179                     }
180                     // 8.2 IOC容器创建原型模式Bean
实例对象
181                     else if (mbd.isPrototype())
{
182                         // It's a prototype ->
create a new instance.
183                         //原型模式(Prototype)是每
次都会创建一个新的对象
184                         Object
prototypeInstance = null;
185                         try {
186                             //回调
beforePrototypeCreation方法，默认的功能是注册当
前创建的原型对象
187
beforePrototypeCreation(beanName);
188                             //创建指定Bean对象实例
```

```
189                             prototypeInstance =
createBean(beanName, mbd, args);
190                         }
191                         finally {
192                             //回调
afterPrototypeCreation方法，默认的功能告诉IOC容
器指定Bean的原型对象不再创建
193
afterPrototypeCreation(beanName);
194                         }
195                         //获取给定Bean的实例对象
196                         bean =
getObjectForBeanInstance(prototypeInstance,
name, beanName, mbd);
197                     }
198                     /*
199                         8.3 其他scope类型
200                         要创建的Bean既不是单例模
式，也不是原型模式，则根据Bean定义资源中      配置的
生命周期范围，
201                         选择实例化Bean的合适方法，
这种在Web应用程序中
202                         比较常用，如：request、
session、application等生命周期
203                      */
204                     else {
205                         String scopeName =
mbd.getScope();
206                         final Scope scope =
this.scopes.get(scopeName);
207                         //Bean定义资源中没有配置生
命周期范围，则Bean定义不合法
208                         if (scope == null) {
```

```
209                         throw new
IllegalStateException("No Scope registered
for scope name '" + scopeName + "'");
210                     }
211                     try {
212                         //这里又使用了一个匿名
内部类，获取一个指定生命周期范围的实例
213                         Object
scopedInstance = scope.get(beanName, () ->
{
214
beforePrototypeCreation(beanName);
215                             try {
216                                 return
createBean(beanName, mbd, args);
217                             }
218                             finally {
219
afterPrototypeCreation(beanName);
220                             }
221                         });
222                         //获取给定Bean的实例对
象
223                         bean =
getObjectForBeanInstance(scopedInstance,
name, beanName, mbd);
224                     }
225                     catch
(IllegalStateException ex) {
226                         throw new
BeanCreationException(beanName,
```

```
227                                     "Scope '" +
scopeName + "' is not active for the
current thread; consider " +
228                                     "defining a
scoped proxy for this bean if you intend to
refer to it from a singleton",
229                                     ex);
230                         }
231                     }
232                 }
233                 catch (BeansException ex) {
234
cleanupAfterBeanCreationFailure(beanName);
235                     throw ex;
236                 }
237             }
238
239             // Check if required type matches
the type of the actual bean instance.
240             /*
241                 9. 类型转换、将返回的bean的类型进行
检查
242                 并可以是将返回的bean 转换为
requiredType 所指定的类型
243                 对创建的Bean实例对象进行类型检查，是
否符合实际类型
244              */
245             if (requiredType != null &&
!requiredType.isInstance(bean)) {
246                 try {
247                     T convertedBean =
getTypeConverter().convertIfNecessary(bean,
requiredType);
```

```
248                 if (convertedBean == null)
{
249                     throw new
BeanNotOfRequiredTypeException(name,
requiredType, bean.getClass());
250                 }
251                 return convertedBean;
252             }
253             catch (TypeMismatchException
ex) {
254                 if
(logger.isTraceEnabled()) {
255                     logger.trace("Failed to
convert bean '" + name + "' to required
type '" +
256
ClassUtils.getQualifiedName(requiredType) +
"'", ex);
257                 }
258                 throw new
BeanNotOfRequiredTypeException(name,
requiredType, bean.getClass());
259             }
260         }
261         return (T) bean;
262     }
```

根据beanName到缓存中取找 Object sharedInstance = getSingleton(beanName);

首先到一级缓存中取对象，如果没有且该bean不是正常创建，则返回如果是正在创建（循环依赖），则到二级缓存中取，如果为空，且允许提前引用，则到三级缓存中取取到后放入二级缓存，删除三级缓存

构造方法依赖： 死循环，只能更改代码，无法解决，因为对象还没有new

属性依赖（set）:对象已经创建好了，只是属性的填充的时候，也就是DI依赖注入的时候，遇到循环问题

1、创建User对象

正在创建中的对象比方说放到一个set集合

User对象实例化【只要实例化了，将User对象的引用会提前暴露到三级缓存中，并将user对象封装到ObjectFactory中】

注意： ObjectFactory不只是可以获取User对象，还可以获取到User对象产生的代理对象，也就是说可能是原对象也可能是代理对象

三级缓存产生对象之后，如果再次获取User就会产生User对象（这里的缓存可能是原对象也可能是代理对象），放到二级缓存中

User对象依赖注入（setRole）

setRole(Role role) --> 创建Role对象

Role对象的实例化

------》 依赖注入（调用我们的setUser方法）

--------》setUser(User user) ---->创建User对象

一级缓存中查找？ 【找不到】

然后到二级缓存找不到，然后再到三级缓存中找到User【一定能找到】

Role对象初始化完成

将Role对象放到一级缓存中

User对象初始化

将User对象放到一级缓存中。

```
@Nullable
    protected Object getSingleton(String beanName, boolean allowEarlyReference) {
        // 检查一级缓存 中是否存在实例
        Object singletonObject = this.singletonObjects.get(beanName);
        // isSingletonCurrentlyInCreation(beanName)返回指定的单例bean当前是否在创建中(在整个工厂中)。
```

```
6           if (singletonObject == null &&
isSingletonCurrentlyInCreation(beanName)) {
7               // 如果为空，则锁定全局变量并进行处理
8               synchronized
(this.singletonObjects) {
9                   // 二级缓存中查找，如果这个bean正
在加载，则不处理。半成品
10                  singletonObject =
this.earlySingletonObjects.get(beanName);
11                  // allowEarlyReference 是否允
许早期依赖（默认为true）
12                  if (singletonObject == null
&& allowEarlyReference) {
13                      // 三级缓存中查找
14                      /*
15                          当某些方法需要提前初始化
的时候
16                          则会调用
addSingletonFactory方法对应的ObjectFactory初始
化策略
17                          并存储在
singletonFactories中
18                       */
19                      ObjectFactory<?>
singletonFactory =
this.singletonFactories.get(beanName);
20                      if (singletonFactory !=
null) {
21                          // 调用预先设定的
getObject方法
22                          singletonObject =
singletonFactory.getObject();
```

```
                            // 将生成的singletonObject放入二级缓存中
                            this.earlySingletonObjects.put(beanName, singletonObject);
                            // 从singletonFactories中移除已处理的beanName
                            // earlySingletonObjects 和 singletonFactories互斥
                            this.singletonFactories.remove(beanName);
                        }
                    }
                }
            }
            return singletonObject;
        }
```

具体创建单例bean的方法：

```
if (mbd.isSingleton()) {
                        //这里使用了一个匿名内部类，创建Bean实例对象，并且注册给所依赖的对象ObjectFactory
                        sharedInstance = getSingleton(beanName, () -> {
                            try {
                                // 直到调用getSingleton()返回依赖
```

```
6                                   // 创建一个指定
Bean实例对象，如果有父级继承，则合并子类和父类的定义
7                                   // 一个真正干活的函
数其实是以do 开头的
8                                   return
createBean(beanName, mbd, args);
9                               }
10                              catch
(BeansException ex) {
11                                  // Explicitly
remove instance from singleton cache: It
might have been put there
12                                  // eagerly by
the creation process, to allow for circular
reference resolution.
13                                  // Also remove
any beans that received a temporary
reference to the bean.
14
destroySingleton(beanName);
15                                  throw ex;
16                              }
17                          });
18                          //获取给定Bean的实例对象
19                          bean =
getObjectForBeanInstance(sharedInstance,
name, beanName, mbd);
20                  }
```

如果缓存中不存在，创建单例getSingleton(String beanName, ObjectFactory<?> singletonFactory)方法：

```
public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {
        Assert.notNull(beanName, "Bean name must not be null");
        // 全局变量需要同步
        synchronized (this.singletonObjects) {
            /*
                首先检查对应的bean是否已经加载过，
                因为singleton模式 其实就是复用创建的bean
             */
            Object singletonObject = this.singletonObjects.get(beanName);
            // 如果为空，才可以进行singleton bean的初始化
            if (singletonObject == null) {
                if (this.singletonsCurrentlyInDestruction) {
                    throw new BeanCreationNotAllowedException(beanName,
                            "Singleton bean creation not allowed while singletons of this factory are in destruction " +
                            "(Do not request a bean from a BeanFactory in a destroy method implementation!)");
                }
                if (logger.isDebugEnabled()) {
```

```
18                      logger.debug("Creating
shared instance of singleton bean '" +
beanName + "'");
19                  }
20                  // 单例创建前的回调  add
21
beforeSingletonCreation(beanName);
22                  boolean newSingleton =
false;
23                  boolean
recordSuppressedExceptions =
(this.suppressedExceptions == null);
24                  if
(recordSuppressedExceptions) {
25
this.suppressedExceptions = new
LinkedHashSet<>();
26                  }
27                  try {
28                      // 初始化bean  通过调用参数
传入的ObjectFactory 的个体Object方法实例化bean
。
29                      singletonObject =
singletonFactory.getObject();
30                      newSingleton = true;
31                  }
32                  catch (IllegalStateException
ex) {
33                      // Has the singleton
object implicitly appeared in the meantime -
>
34                      // if yes, proceed with
it since the exception indicates that state.
```

```
35                      singletonObject =
this.singletonObjects.get(beanName);
36                      if (singletonObject ==
null) {
37                          throw ex;
38                      }
39                  }
40                  catch (BeanCreationException
ex) {
41                      if
(recordSuppressedExceptions) {
42                          for (Exception
suppressedException :
this.suppressedExceptions) {
43
ex.addRelatedCause(suppressedException);
44                          }
45                      }
46                      throw ex;
47                  }
48                  finally {
49                      if
(recordSuppressedExceptions) {
50
this.suppressedExceptions = null;
51                      }
52                      // 加载单例后的处理方法调用
remove
53
afterSingletonCreation(beanName);
54                  }
55                  if (newSingleton) {
```

```
                        // 将结果记录至缓存（一级缓存）并删除加载bean 过程中所记录的各种辅助状态。
                        addSingleton(beanName, singletonObject);
                    }
                }
                return singletonObject;
            }
        }
```

创建完成后放入一级缓存中，然后把bean对象返回

```
finally {
    if (recordSuppressedExceptions) {
        this.suppressedExceptions = null;
    }
    // 加载单例后的处理方法调用   remove
    afterSingletonCreation(beanName);
}
if (newSingleton) {
    // 将结果记录至缓存（一级缓存）并删除加载bean过程中所记录的各种辅助状态。
    addSingleton(beanName, singletonObject);
}
```

创建的bean流程

```
// Create bean instance.
if (mbd.isSingleton()) {
    sharedInstance = getSingleton(beanName, () -> {
        try {
            return createBean(beanName, mbd, args);
        }
        catch (BeansException ex) {
            // Explicitly remove instance from singleton cache: It might ha
            // eagerly by the creation process, to allow for circular refer
            // Also remove any beans that received a temporary reference to
            destroySingleton(beanName);
            throw ex;
        }
    });
    bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
}
```

调用AbstractAutowireCapableBeanFactory中的createBean方法

```
@Override
protected Object createBean(String beanName, RootBeanDefinition mbd, @Nullable Object[] args) throws BeanCreationException {
        ..........
        try {

            Object beanInstance = doCreateBean(beanName, mbdToUse, args);
            if (logger.isTraceEnabled()) {

                logger.trace("Finished creating instance of bean '" + beanName +"'");
            }
            return beanInstance;
        }
        ........
}
```

对象创建的具体流程在**doCreateBean**这个方法中：

```
/*
        1. 如果是单例则需要首先清除缓存。
        2. 实例化bean ，将BeanDefinition 转换为BeanWrapper 。
                转换是一个复杂的过程，但是我们可以尝试概括大致的功能，如下所示。
                - 如果存在工厂方法则使用工厂方法进行初始化。
                - 一个类有多个构造函数，每个构造函数都有不同的参数，所以需要根据参数锁定构造
                  函数并进行初始化。
                - 如果既不存在工厂方法也不存在带有参数的构造函数，则使用默认的构造函数进行
                  bean的实例化。
        3. MergedBeanDefinitionPostProcessor 的应用。
                bean合并后的处理，Autowired注解正是通过此方法实现诸如类型的预解析。
        4. 依赖处理。
                在Spring 中会有循环依赖的情况，
                例如，当A 中含有B 的属性，而B 中又含有A 的属性时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。
        5. 属性填充。将所有属性填充至bean 的实例中。
```

```
19              6. 循环依赖检查。
20                  之前有提到过，在Spring中解决循环依赖只对单例有效，而对于prototype 的bean,
21                  Spring没有好的解决办法，唯一要做的就是抛出异常。
22                  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，并判断是再需要抛出异常。
23              7. 注册DisposableBean
24                  如果配置了destroy-method ，这里需要注册以便于在销毁时候调用。
25              8. 完成创建并返回。
26                  可以看到上面的步骤非常的繁琐，每一步骤都使用了大量的代码来完成其功能，
27                  最复杂也是最难以理解的当属循环依赖的处理，
28                  在真正进入doCreateBean 前我们有必要先了解下循环依赖。
29          */
30      protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final @Nullable Object[] args)
31                  throws BeanCreationException {
32              /*
33                  在该方法中，首先调用createBeanInstance方法，
34                  创建bean实例对象（这个时候执行bean的构造方法），
35                  然后调用populateBean方法，对bean进行填充，注入相关依赖，
36                  之后再调用方法initializeBean，进行相关初始化工作
37               */
38
```

```
39          // 【实例化bean】 Instantiate the
bean.
40          BeanWrapper instanceWrapper = null;
41          if (mbd.isSingleton()) {
42              instanceWrapper =
this.factoryBeanInstanceCache.remove(beanNa
me);
43          }
44          if (instanceWrapper == null) {
45              //用来创建bean实例 （这个时候执行
bean的构造方法）;
46              // 这步执行，就完成了bean的实例化操
作
47              // 根据指定bean 使用对应的策略创建新
的实例， 如· 工厂方法、构造函数自动注入、简单初始化
48              instanceWrapper =
createBeanInstance(beanName, mbd, args);
49          }
50          final Object bean =
instanceWrapper.getWrappedInstance();
51          Class<?> beanType =
instanceWrapper.getWrappedClass();
52          if (beanType != NullBean.class) {
53              mbd.resolvedTargetType =
beanType;
54          }
55
56          // Allow post-processors to modify
the merged bean definition.
57          // 允许后处理程序修改合并后的bean定义。
58          synchronized
(mbd.postProcessingLock) {
59              if (!mbd.postProcessed) {
```

```
60                  try {
61                      // 将
MergedBeanDefinitionPostProcessors应用到指定
的bean定义，
62                      // 调用它们的
postProcessMergedBeanDefinition方法
63
applyMergedBeanDefinitionPostProcessors(mbd
, beanType, beanName);
64                  }
65                  catch (Throwable ex) {
66                      throw new
BeanCreationException(mbd.getResourceDescri
ption(), beanName,
67                              "Post-
processing of merged bean definition
failed", ex);
68                  }
69                  mbd.postProcessed = true;
70              }
71          }
72
73          // Eagerly cache singletons to be
able to resolve circular references
74          // even when triggered by lifecycle
interfaces like BeanFactoryAware.
75          // 快速缓存单例 以便能够解决循环引用
76          // 即使被BeanFactoryAware这样的生命周期
接口触发。
77          // 是否需要提早曝光（循环依赖） 条件（单例
&允许循环依赖&当前bean正在创建）
```

```
78          boolean earlySingletonExposure =
(mbd.isSingleton() &&
this.allowCircularReferences &&
79
isSingletonCurrentlyInCreation(beanName));
80          if (earlySingletonExposure) {
81              if (logger.isTraceEnabled()) {
82                  logger.trace("Eagerly
caching bean '" + beanName +
83                          "' to allow for
resolving potential circular references");
84              }
85              //上面完成bean的实例化（初始化之
前）， 向三级缓存中放入一个实例化对象（匿名处理类）
86              //aop就是在这里动态的将advice织入
bean中
87              addSingletonFactory(beanName,
() -> getEarlyBeanReference(beanName, mbd,
bean));
88          }
89
90          /*
91              初始化bean实例
92              下面调用populateBean方法，对bean进
行填充，注入相关依赖
93           */
94          // 【初始化bean】 Initialize the bean
instance.
95          Object exposedObject = bean;
96          try {
97              // 【填充】
98              // 填充Bean，该方法就是 发生依赖注入
的地方
```

```
99              populateBean(beanName, mbd,
instanceWrapper);
100             // 再调用方法initializeBean（如
init-method），进行相关初始化工作
101             exposedObject =
initializeBean(beanName, exposedObject,
mbd);
102         }
103         catch (Throwable ex) {
104             if (ex instanceof
BeanCreationException &&
beanName.equals(((BeanCreationException)
ex).getBeanName())) {
105                 throw
(BeanCreationException) ex;
106             }
107             else {
108                 throw new
BeanCreationException(
109
mbd.getResourceDescription(), beanName,
"Initialization of bean failed", ex);
110             }
111         }
112
113         if (earlySingletonExposure) {
114             Object earlySingletonReference
= getSingleton(beanName, false);
115             if (earlySingletonReference !=
null) {
116                 if (exposedObject == bean)
{
```

```
117                     exposedObject =
earlySingletonReference;
118                 }
119                 else if
(!this.allowRawInjectionDespiteWrapping &&
hasDependentBean(beanName)) {
120                     String[] dependentBeans
= getDependentBeans(beanName);
121                     Set<String>
actualDependentBeans = new LinkedHashSet<>
(dependentBeans.length);
122                     for (String
dependentBean : dependentBeans) {
123                         if
(!removeSingletonIfCreatedForTypeCheckOnly(
dependentBean)) {
124
actualDependentBeans.add(dependentBean);
125                         }
126                     }
127                     if
(!actualDependentBeans.isEmpty()) {
128                         throw new
BeanCurrentlyInCreationException(beanName,
129                                 "Bean with
name '" + beanName + "' has been injected
into other beans [" +
130
StringUtils.collectionToCommaDelimitedStrin
g(actualDependentBeans) +
131                                 "] in its
raw version as part of a circular
reference, but has eventually been " +
```

```
132                                     "wrapped.
This means that said other beans do not use
the final version of the " +
133                                     "bean. This
is often the result of over-eager type
matching - consider using " +
134
"'getBeanNamesOfType' with the
'allowEagerInit' flag turned off, for
example.");
135                         }
136                     }
137                 }
138             }
139
140             // Register bean as disposable.
141             try {
142
registerDisposableBeanIfNecessary(beanName,
bean, mbd);
143             }
144             catch
(BeanDefinitionValidationException ex) {
145                 throw new
BeanCreationException(
146
mbd.getResourceDescription(), beanName,
"Invalid destruction signature", ex);
147             }
148
149             return exposedObject;
150         }
151
```