# ====模拟Spring的Bean工厂====

## 模拟BeanFactory和ApplicationContext

Spring的BeanFactory和ApplicationContext想必大家经常听，下面来模拟一下这两者的关系和实现。

Spring的BeanFactory ，就是一个容器，是用一个map实现的，就是从配置文件读取

|  |
| --- |
| <**bean id="v" class="com.bjsxt.spring.factory.Train"**/> |

这样的配置，遍历解析这样的xml配置，以id为key，class后的类全限定名用反射生成的对象为value，放到这个map中去。当用的时候，直接map.get(id); 获取这个Bean对象。

ClassPathXmlApplicationContext是BeanFactory的一种实现。这里模拟这种实现。

这里模拟Spring的Bean工厂：

|  |
| --- |
| **public interface** BeanFactory {  Object getBean(String id); } |

ClassPathXmlApplicationContext :

|  |
| --- |
| **public class** ClassPathXmlApplicationContext **implements** BeanFactory{  *//存放一个个Bean对象的容器，* **private** Map<String, Object> **container** = **new** HashMap<String, Object>();   *// 构造方法找到配置文件，读取xml配置文件* **public** ClassPathXmlApplicationContext(String fileName) **throws** Exception{  SAXBuilder sb = **new** SAXBuilder();  Document doc = sb.build(**this**.getClass().getClassLoader()  .getResourceAsStream(fileName));  Element root = doc.getRootElement();  List list = XPath.selectNodes(root, **"/beans/bean"**);  System.***out***.println(list.size());   **for** (**int** i = 0; i < list.size(); i++) {  Element bean = (Element) list.get(i);  String id = bean.getAttributeValue(**"id"**);  String clazz = bean.getAttributeValue(**"class"**);  Object o = Class.*forName*(clazz).newInstance();  **container**.put(id, o);  System.***out***.println(id + **" "** + clazz);  }  }   *//读取配置文件，读取id为传进来的id的Bean，实例化* @Override  **public** Object getBean(String id) {  **return container**.get(id);  } } |

xml配置：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans**>  <**bean id="v" class="com.lhy.springfactory.Car"**/>  <**bean id="trian" class="com.lhy.springfactory.Train"**/>  *<!-- //v=com.bjsxt.spring.factory.Car -->* </**beans**> |

Train.java

|  |
| --- |
| **public class** Train **implements** Moveable{  @Override  **public void** run() {  System.***out***.println(**"小火车呜呜呜..."**);  } } |

Car.java

|  |
| --- |
| **public class** Car **implements** Moveable{  @Override  **public void** run() {  System.***out***.println(**"冒着烟奔跑中..."**);  } } |

测试程序：

|  |
| --- |
| **public static void** main(String[] args) **throws** Exception {   BeanFactory f =  **new** ClassPathXmlApplicationContext(  **"com/lhy/springfactory/applicationContext.xml"**);  Object o = f.getBean(**"v"**);  Moveable m = (Moveable) o;  m.run();   Train trian = (Train) f.getBean(**"trian"**);  trian.run(); } |

打印：

v com.lhy.springfactory.Car

trian com.lhy.springfactory.Train

冒着烟奔跑中...

小火车呜呜呜...

结论：

这样，就把类配置在了配置文件里。进一步，还可以在配置文件描述类与类之间的关系。

[**详细代码**]  https://github.com/lhy1234/DesignPattern\_Factory

粗糙版BeanFactory和ApplicationContext已经体验完了，下面开始详细学习。

# ===BeanFactory和ApplicationContext=

Spring本质其实就是通过一个配置文件描述Bean与Bean之间的依赖关系，利用java反射功能实例化Bean并建立Bean之间的依赖关系。Spring的IoC容器在完成这些底层工作的基础上，还提供了Bean实例缓存，生命周期管理，Bean实例代理，事件发布，资源装载等高级服务。

## BeanFactory：

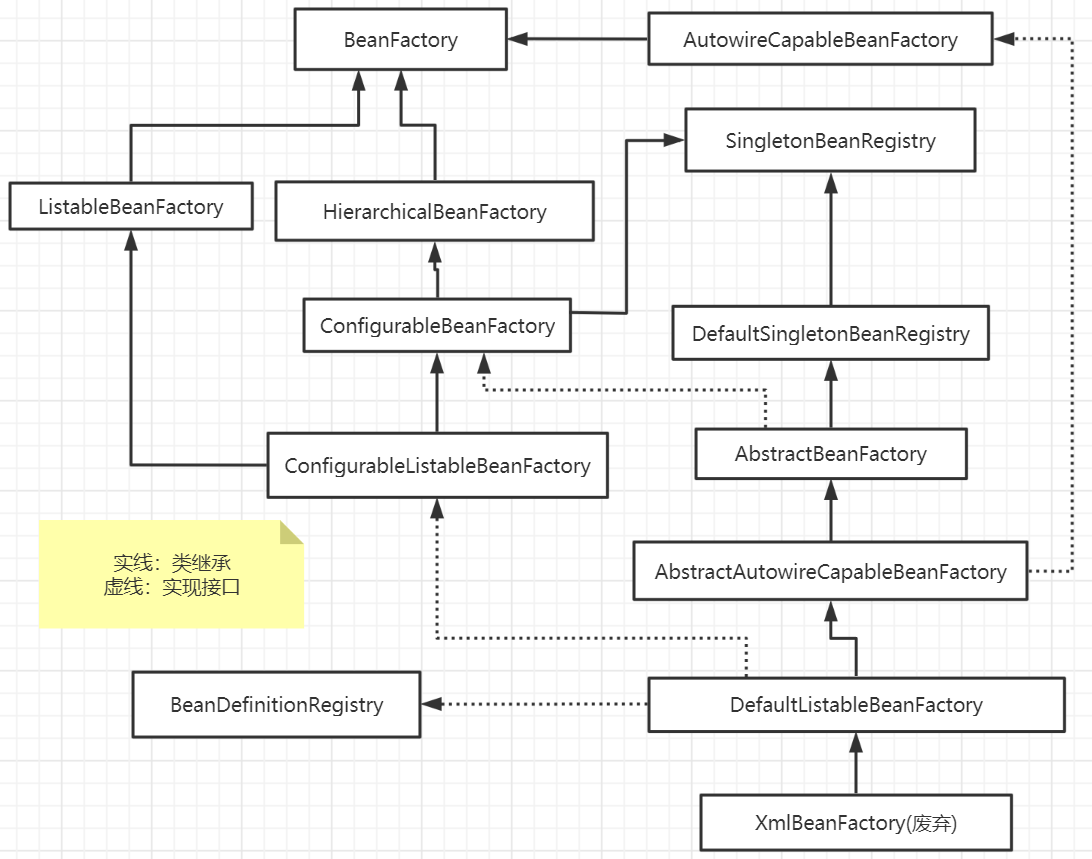
### 定义

俗称：bean工厂 或 IoC容器，是Spring最核心的接口，Spring框架的基础设施，面向Spring本身。

BeanFactory是一个通用的类工厂，可以创建并管理各种的pojo，不同于传统意义的JavaBean。

### 体系结构

Spring为BeanFactory提供了多种实现，最常见的是XmlBeanFactory，但在Spring3.2中已经被废弃，建议使用XmlBeanDefinitionReader+DefaultListableBeanFactory 替代。



### BeanFactory的方法

|  |
| --- |
| **public interface** BeanFactory {  String ***FACTORY\_BEAN\_PREFIX*** = **"&"**;  Object getBean(String name) **throws** BeansException;  <T> T getBean(String name, Class<T> requiredType) **throws** BeansException;  Object getBean(String name, Object... args) **throws** BeansException;  <T> T getBean(Class<T> requiredType) **throws** BeansException;  <T> T getBean(Class<T> requiredType, Object... args) **throws** BeansException;  <T> ObjectProvider<T> getBeanProvider(Class<T> requiredType);  <T> ObjectProvider<T> getBeanProvider(ResolvableType requiredType);  **boolean** containsBean(String name);  **boolean** isSingleton(String name) **throws** NoSuchBeanDefinitionException;  **boolean** isPrototype(String name) **throws** NoSuchBeanDefinitionException;  **boolean** isTypeMatch(String name, ResolvableType typeToMatch) **throws** NoSuchBeanDefinitionException;  **boolean** isTypeMatch(String name, Class<?> typeToMatch) **throws** NoSuchBeanDefinitionException;  @Nullable  Class<?> getType(String name) **throws** NoSuchBeanDefinitionException;  @Nullable  Class<?> getType(String name, **boolean** allowFactoryBeanInit) **throws** NoSuchBeanDefinitionException;  String[] getAliases(String name); } |

#### FACTORY\_BEAN\_PREFIX

这是定义的 FactoryBean 前缀。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 如果你在容器中有一个bean的类型是org.springframework.beans.factory.FactoryBean,例如：   |  |  | | --- | --- | | **public class** CarFactoryBean **implements** FactoryBean<Car> {  @Override  **public** Car getObject() **throws** Exception {  **return new** Car();  }   @Override  **public** Class<?> getObjectType() {  **return** Car.**class**;  } } |  |   并且将该bean注册到Ioc容器中:   |  | | --- | | <**bean id="carBean" class="com.nb.nbbase2.beans.CarFactoryBean"**/> |   那么通常从容器根据bean id获取bean时:   |  | | --- | | @Test **public void** test2(){  ClassPathXmlApplicationContext ctx = **new** ClassPathXmlApplicationContext(**"beans.xml"**);  Object carBean = ctx.getBean(**"carBean"**);  System.***err***.println(carBean.getClass().toString()); } |   //打印  class com.nb.nbbase2.beans.Car  （如果换成Object carBean = ctx.getBean(**"&carBean"**);则打印：  class com.nb.nbbase2.beans.CarFactoryBean）  获取到的是BeanFactory创建的类，通过FactoryBean<CarFactoryBean>定义的 getObject()返回工厂创建的实例。（class com.nb.nbbase2.beans.Car）  那么我想获取FactoryBean本身就需要使用BeanFactory定义的"&"了（注意这里BeanFactory与FactoryBean的区别，BeanFactory可以理解为Ioc的实现定义，而FactoryBean可以理解为BeanFactory中注册bean的底层工厂定义）。  可见BeanFactory定义的"&"的用处，用于区分获取是由BeanFactory创建的bean实例还是FactoryBean本身实例是通过FactoryBean定义的getObject行为来设定的。 |

#### getBean方法

**前两个方法：**在获取一个 Bean 的时候，可以指定 Bean 的名字，也可以指定 Bean 的类型，也可以同时指定 。

**带args参数的两个方法：**args 是获取 Bean 时所需要的参数（用此方法需要将 Bean 的 scope 设置为 prototype），每次获取 Bean 的时候，才进行 Bean 的初始化（否则配置的参数无法生效）。

说白了就是new对象的时候，传给构造方法的参数，可以传多个。

|  |  |  |  |
| --- | --- | --- | --- |
| 例子：  User类   |  | | --- | | **package** com.nb.nbbase2.beans;  */\*\*  \* create by lihaoyang on 2020/9/8  \*/* **public class** User {  **private** String **username**;  **private** Integer **age**;  **public User(){}**  **public** User(String username, Integer age) {  **this**.**username** = username;  **this**.**age** = age;  }   **public** String getUsername() {  **return username**;  }   **public void** setUsername(String username) {  **this**.**username** = username;  }   **public** Integer getAge() {  **return age**;  }   **public void** setAge(Integer age) {  **this**.**age** = age;  }   @Override  **public** String toString() {  **return "User{"** +  **"username='"** + **username** + **'\''** +  **", age="** + **age** +  **'}'**;  } } | |  |   Xml里面不给设置有参构造，注意设置 scope 属性为 prototype，否则不起效  <**bean id="user" class="com.nb.nbbase2.beans.User" scope="prototype"**/>  测试：   |  | | --- | | @Test **public void** testArgs(){  ClassPathXmlApplicationContext ctx = **new** ClassPathXmlApplicationContext(**"beans.xml"**);  User user = ctx.getBean(User.**class**,**new** Object[]{**"niubei"**,18});  System.***err***.println(**"user = "** + user); } |   打印：  user = User{username='niubei', age=18}  结论：  带有Args的BeanFactory#getBean方法，可以在getBean的时候给构造器传参，来实例化对象。 |

#### getBeanProvider

先不讨论 <https://mp.weixin.qq.com/s/PfLZn-Ir6ejKg1LeH5s2PQ>

#### containsBean

判断容器中是否包含某个 Bean

#### isSingleton

判断某个 Bean 是否是单例的。

#### isPrototype

判断某个 Bean 是否是 prototype

#### isTypeMatch

返回指定名称的 Bean 是否匹配指定的类型。

#### getType

返回指定名称的 Bean 对应的数据类型。

#### getAliases

返回 Bean 的别名

### 初始化BeanFactory

#### xml配置文件启动IoC容器

(这里new的是springboot项目)使用配置文件为Car提供配置信息，通过BeanFactory装载配置文件，启动SpringIoC容器。

Car类

|  |
| --- |
| **package** com.nb.nbbase2.beans;  */\*\*  \* create by lihaoyang on 2020/9/8  \*/* **public class** Car {   **private** String **brand**;  **private** String **color**;  **private** Integer **maxSpeed**;   **public** String getBrand() {  **return brand**;  }   **public void** setBrand(String brand) {  **this**.**brand** = brand;  }   **public** String getColor() {  **return color**;  }   **public void** setColor(String color) {  **this**.**color** = color;  }   **public** Integer getMaxSpeed() {  **return maxSpeed**;  }   **public void** setMaxSpeed(Integer maxSpeed) {  **this**.**maxSpeed** = maxSpeed;  }   @Override  **public** String toString() {  **return "Car{"** +  **"brand='"** + **brand** + **'\''** +  **", color='"** + **color** + **'\''** +  **", maxSpeed="** + **maxSpeed** +  **'}'**;  } } |

XML配置文件

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>   <**bean id="car" class="com.nb.nbbase2.beans.Car"** >  <**property name="brand" value="长安"**></**property**>  <**property name="color" value="红色"**></**property**>  <**property name="maxSpeed" value="200"**></**property**>  </**bean**>  </**beans**> |

启动容器

|  |
| --- |
| @Test **public void** test() {    ResourcePatternResolver resolver = **new** PathMatchingResourcePatternResolver();  Resource resource = resolver.getResource(**"beans.xml"**);  System.***err***.println(**"UrL--"**+resolver);  *//被废弃  //BeanFactory factory = new XmlBeanFactory(resource);* DefaultListableBeanFactory factory = **new** DefaultListableBeanFactory();  XmlBeanDefinitionReader reader = **new** XmlBeanDefinitionReader(factory);  reader.loadBeanDefinitions(resource);   System.***err***.println(**"----> "**+factory.getBean(**"car"**)); } |

**打印**

----> Car{brand='长安', color='红色', maxSpeed=200}

**总结**

XmlBeanDefinitionReader通过Resource装载Spring配置文件并启动IoC容器，然后就可以通过BeanFactory#getBean(beanName)从容器获取Bean。

1. 启动容器并不初始化bean,初始化发生在第一次调用。
2. 对于单例Bean，BeanFactory会缓存bean。在DefaultSingletonBeanRegistry类中有一个ConcurrentHashMap

|  |
| --- |
| */\*\* Cache of singleton objects: bean name to bean instance. \*/* **private final** Map<String, Object> **singletonObjects** = **new** ConcurrentHashMap<>(256); |

### 二级接口

BeanFactory是顶级接口，直接的二级接口有三个：

**ListableBeanFactory、**HierarchicalBeanFactory、AutowireCapableBeanFactory

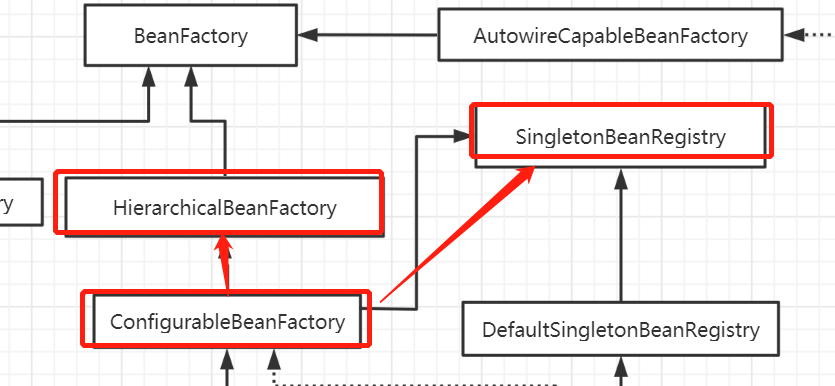
#### HierarchicalBeanFactory

父子级联的IoC容器的接口，继承自 BeanFactory，定义了工厂分层，子容器可以通过接口方法访问父容器。在其基础上又扩展了两个方法：

|  |
| --- |
| **public interface** HierarchicalBeanFactory **extends** BeanFactory {   @Nullable  BeanFactory getParentBeanFactory();  **boolean** containsLocalBean(String name); } |

* getParentBeanFactory 方法返回 Bean 工厂的父工厂，实现了工厂分层。
* containsLocalBean 方法则判断本地工厂是否包含这个 Bean。

HierarchicalBeanFactory 有一个子接口 ConfigurableBeanFactory，ConfigurableBeanFactory 接口继承自 HierarchicalBeanFactory 和 SingletonBeanRegistry，其中 SingletonBeanRegistry 这个接口定义了对单例 Bean 的定义以及获取方法。也就是说，ConfigurableBeanFactory 中兼具工厂分层和单例处理功能。同时，在 HierarchicalBeanFactory 中 getParentBeanFactory 所 get 到的对象，也是在 ConfigurableBeanFactory 中完成配置。



#### AutowireCapableBeanFactory

AutowireCapableBeanFactory 继承自 BeanFacotory，它扩展了自动装配的功能。

|  |
| --- |
| **public interface** AutowireCapableBeanFactory **extends** BeanFactory {  **int *AUTOWIRE\_NO*** = 0;  **int *AUTOWIRE\_BY\_NAME*** = 1;  **int *AUTOWIRE\_BY\_TYPE*** = 2;  **int *AUTOWIRE\_CONSTRUCTOR*** = 3;  @Deprecated  **int *AUTOWIRE\_AUTODETECT*** = 4;  String ***ORIGINAL\_INSTANCE\_SUFFIX*** = **".ORIGINAL"**;  <T> T createBean(Class<T> beanClass) **throws** BeansException;  Object createBean(Class<?> beanClass, **int** autowireMode, **boolean** dependencyCheck) **throws** BeansException;  **void** autowireBean(Object existingBean) **throws** BeansException;  Object autowire(Class<?> beanClass, **int** autowireMode, **boolean** dependencyCheck) **throws** BeansException;  **void** autowireBeanProperties(Object existingBean, **int** autowireMode, **boolean** dependencyCheck) **throws** BeansException;  Object configureBean(Object existingBean, String beanName) **throws** BeansException;  Object initializeBean(Object existingBean, String beanName) **throws** BeansException;  **void** applyBeanPropertyValues(Object existingBean, String beanName) **throws** BeansException;  Object applyBeanPostProcessorsBeforeInitialization(Object existingBean, String beanName) **throws** BeansException;  Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName) **throws** BeansException;  **void** destroyBean(Object existingBean);  <T> NamedBeanHolder<T> resolveNamedBean(Class<T> requiredType) **throws** BeansException;  Object resolveBeanByName(String name, DependencyDescriptor descriptor) **throws** BeansException;  @Nullable  Object resolveDependency(DependencyDescriptor descriptor, @Nullable String requestingBeanName) **throws** BeansException;  @Nullable  Object resolveDependency(DependencyDescriptor descriptor, @Nullable String requestingBeanName,  @Nullable Set<String> autowiredBeanNames, @Nullable TypeConverter typeConverter) **throws** BeansException; } |

首先通过五个常量定义了五种不同的装配策略，分别是：不自动装配、通过名称自动装配、通过类型自动装配、通过构造方法自动装配，还有一个过期的常量。

ORIGINAL\_INSTANCE\_SUFFIX 则是初始化实例给定名称时约定的后缀，该后缀会添加到类的全路径后面，例如：com.mypackage.MyClass.ORIGINAL。

createBean 方法用来创建 Bean 实例；

autowire\* 方法则用来完成自动装配；

configureBean 用来配置 Bean；

initializeBean 用来初始化Bean；

applyBeanPropertyValues 将指定 bean 的 BeanDefinition 应用到一个已经存在的 Bean上；applyBeanPostProcessorsBeforeInitialization/applyBeanPostProcessorsAfterInitialization 调用 Bean 的后置处理器；

destroyBean 方法用来销毁 Bean；

resolve\* 方法用来解析 Bean。

#### ListableBeanFactory

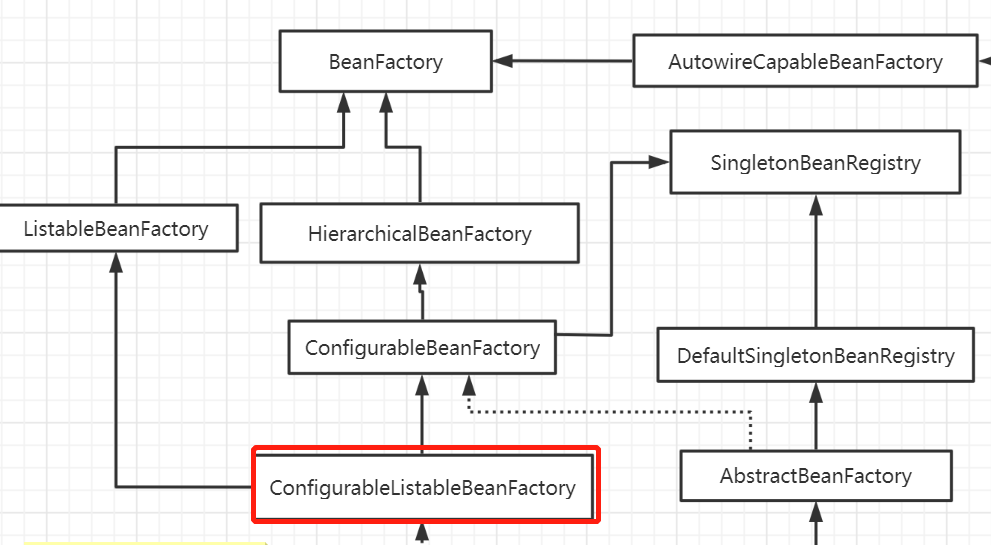
定义了访问容器中Bean基本信息的若干方法，如查看Bean的个数，获取某一类型Bean的配置名、查看容器中是否包括某一Bean等。

ListableBeanFactory 继承自 BeanFacotory，该接口可以列出工厂可以生产的所有实例。

|  |
| --- |
| **public interface** ListableBeanFactory **extends** BeanFactory {  **boolean** containsBeanDefinition(String beanName);  **int** getBeanDefinitionCount();  String[] getBeanDefinitionNames();  String[] getBeanNamesForType(ResolvableType type);  String[] getBeanNamesForType(ResolvableType type, **boolean** includeNonSingletons, **boolean** allowEagerInit);  String[] getBeanNamesForType(@Nullable Class<?> type);  String[] getBeanNamesForType(@Nullable Class<?> type, **boolean** includeNonSingletons, **boolean** allowEagerInit);  <T> Map<String, T> getBeansOfType(@Nullable Class<T> type) **throws** BeansException;  <T> Map<String, T> getBeansOfType(@Nullable Class<T> type, **boolean** includeNonSingletons, **boolean** allowEagerInit) **throws** BeansException;  String[] getBeanNamesForAnnotation(Class<? **extends** Annotation> annotationType);  Map<String, Object> getBeansWithAnnotation(Class<? **extends** Annotation> annotationType) **throws** BeansException;  @Nullable  <A **extends** Annotation> A findAnnotationOnBean(String beanName, Class<A> annotationType) **throws** NoSuchBeanDefinitionException; } |

* containsBeanDefinition：判断容器是否包含某一个 Bean 的定义。
* getBeanDefinitionCount：获取 BeanDefinition 数量。getBeanDefinitionCount 和 containsBeanDefinition 执行效率较低，慎用。
* getBeanDefinitionNames：获取所有 Bean 的名字。
* getBeanNamesForType：返回指定类型的 BeanName。
* getBeansOfType：返回指定类的名称和 Bean Map（相同类型的 Bean 可能存在多种实例）。
* getBeanNamesForAnnotation：根据注解获取指定的 BeanName。
* getBeansWithAnnotation：根据注解获取指定的 BeanName 以及 Bean Map。
* findAnnotationOnBean：根据指定 beanName 和注解类型查找指定的 Bean。

ListableBeanFactory 有一个实现接口 ，这是以上所介绍接口的集大成者，ConfigurableListableBeanFactory 继承自 ListableBeanFactory、AutowireCapableBeanFactory 以及 ConfigurableBeanFactory，可以看到，三个二级接口功能集于一身，它包含了 BeanFactory 体系目前的所有方法。这些方法都在 DefaultListableBeanFactory 中进行实现



**核心的类**

**BeanDefinition：**

注释， 一个 BeanDefinition 描述了一个 Bean 实例，实例包含属性值、构造方法参数值以及更多实现信息。该 BeanDefinition 只是是一个最小的接口，主要目的是允许修改属性值和其他 Bean 元数据 。

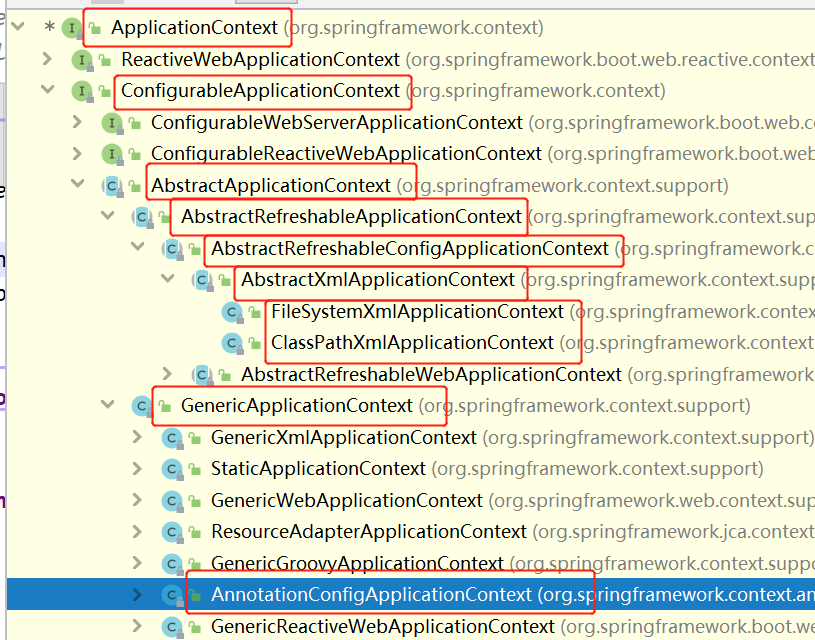
说白了 就是解析xml后的bean标签后，创建的bean对象的包装类。管理Spring中的各种对象，以及它们之间依赖关系。控制反转都是围绕对BeanDefinition 的处理来完成的。

**ConfigurableBeanFactory ：**

增强了IoC容器的可定制性。定义了设置类装载器、属性编辑器、容器初始化后置处理器等方法。

## ApplicationContext

俗称：应用上下文 或 Spring容器，是建立在BeanFactory基础之上，提供了更多面向应用的功能，提供了国际化支持和框架事件体系，面向程序员，都使用ApplicationContext而不使用BeanFactory。



### 类体系结构

ApplicationContext的主要实现类：

ClassPathXmlApplicationContext：默认从类路径加载配置文件

FileSystemXmlApplicationContext：默认从文件系统加载配置文

#### 扩展的接口

下图可以看出，ApplicationContext继承了ListableBeanFactory,HierarchicalBeanFactory接口，使其拥有了BeanFactory的功能，还继承了其它接口扩展了BeanFactory的功能：

* ApplicationEventPublisher

让容器拥有发布应用上下文事件的功能，包括容器启动、容器关闭等事件。（后面再议）

* MessageSource

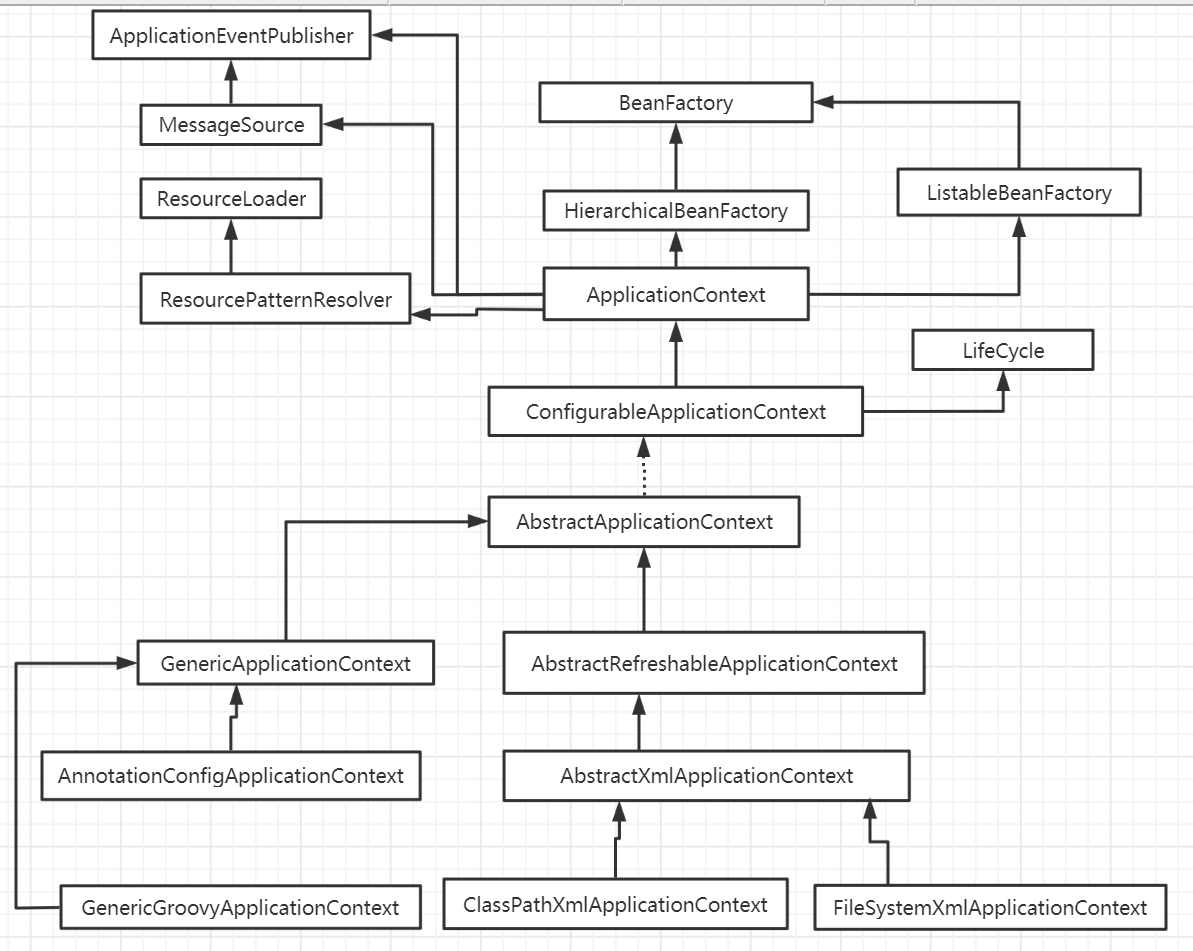
国际化相关功能

* ResourcePatternResolver

可以通过带前缀的Ant风格的资源文件路径装载Spring配置文件

* LifeCycle

xxx



#### 子接口

* **ConfigurableApplicationContext**

新增了refresh()方法、close()方法，使ApplicationContext具有启动、刷新、关闭应用上下文能力。

在应用上下文关闭时->调用refresh()方法可启动应用上下文，

在应用上下文已经启动时->调用refresh()方法清空缓存并重新装载配置信息，

Close()关闭应用上下文

### 初始化ApplicationContext

1，配置文件在类路径下

|  |
| --- |
| ApplicationContext ctx = **new** ClassPathXmlApplicationContext(**"classpath:beans.xml"**);  其中构造方法的**classpath:可省略** |

2，配置文件在文件系统

|  |
| --- |
| ApplicationContext ctx = **new** FileSystemXmlApplicationContext(**"file:D:\\Z\_lhy\\beans.xml"**);  其中构造方法的**file:可省略** |

1. 还可以传多个配置文件，spring会自动整合成一个

|  |
| --- |
| ApplicationContext ctx = **new** ClassPathXmlApplicationContext(**"beans.xml"**,**"beans2.xml"**); |

1. java config初始化

|  |
| --- |
| @Configuration **public class** BeansConfig {   *//默认方法名就是bean id* @Bean  **public** Car car(){  Car car = **new** Car();  car.setBrand(**"昌河"**);  car.setColor(**"黄色"**);  car.setMaxSpeed(90);  **return** car;  } }  @Test **public void** testJavaConfig() {  ApplicationContext ctx = **new** AnnotationConfigApplicationContext(BeansConfig.**class**);  Car car = ctx.getBean(**"car"**,Car.**class**);  String[] names = ctx.getBeanDefinitionNames();  **for**(String name : names){  System.***err***.println(**"name --- "**+name);  }  System.***err***.println(car.toString()); }  name --- org.springframework.context.annotation.internalCommonAnnotationProcessor  name --- org.springframework.context.event.internalEventListenerProcessor  name --- org.springframework.context.event.internalEventListenerFactory  name --- beansConfig  name --- car  Car{brand='昌河', color='黄色', maxSpeed=90} |

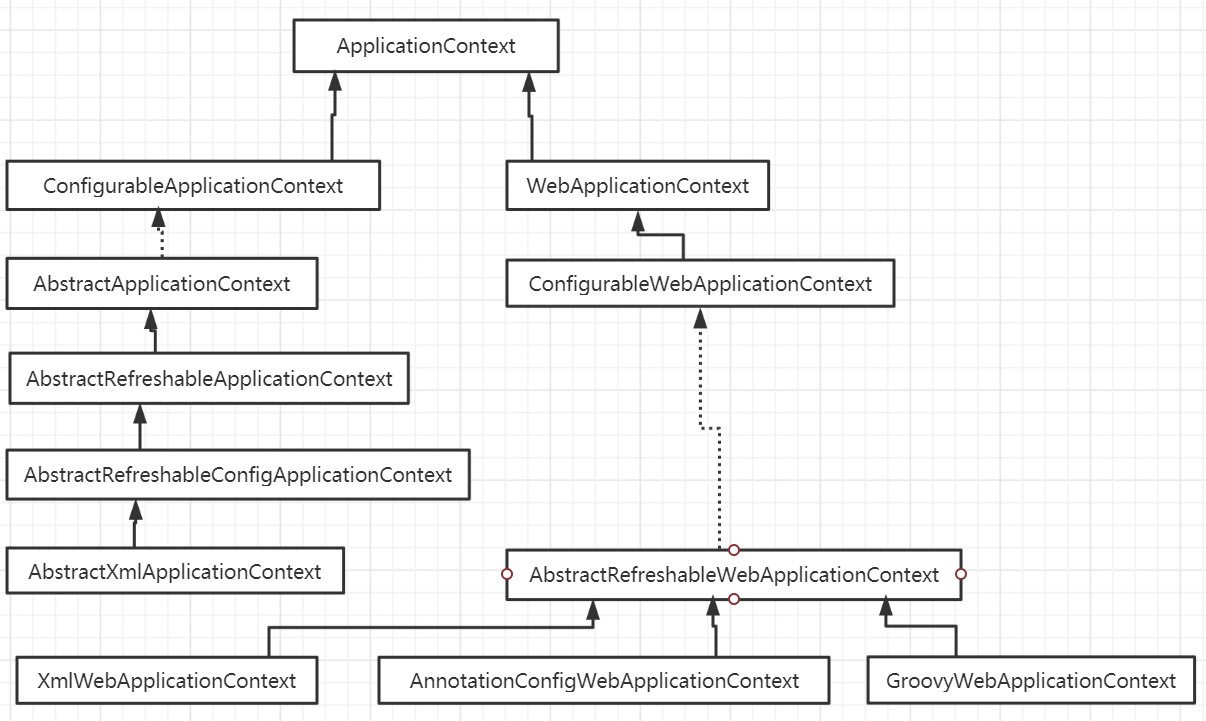
然后就可以ctx.getBean了

ApplicationContext和BeanFactory的一个重大区别：

BeanFactory在初始化容器时并不实例化bean，第一次访问时才初始化。

ApplicationContext初始化应用上下文时就初始化所有的单实例Bean。

### WebApplicationContext



### 父子容器

Xxxx

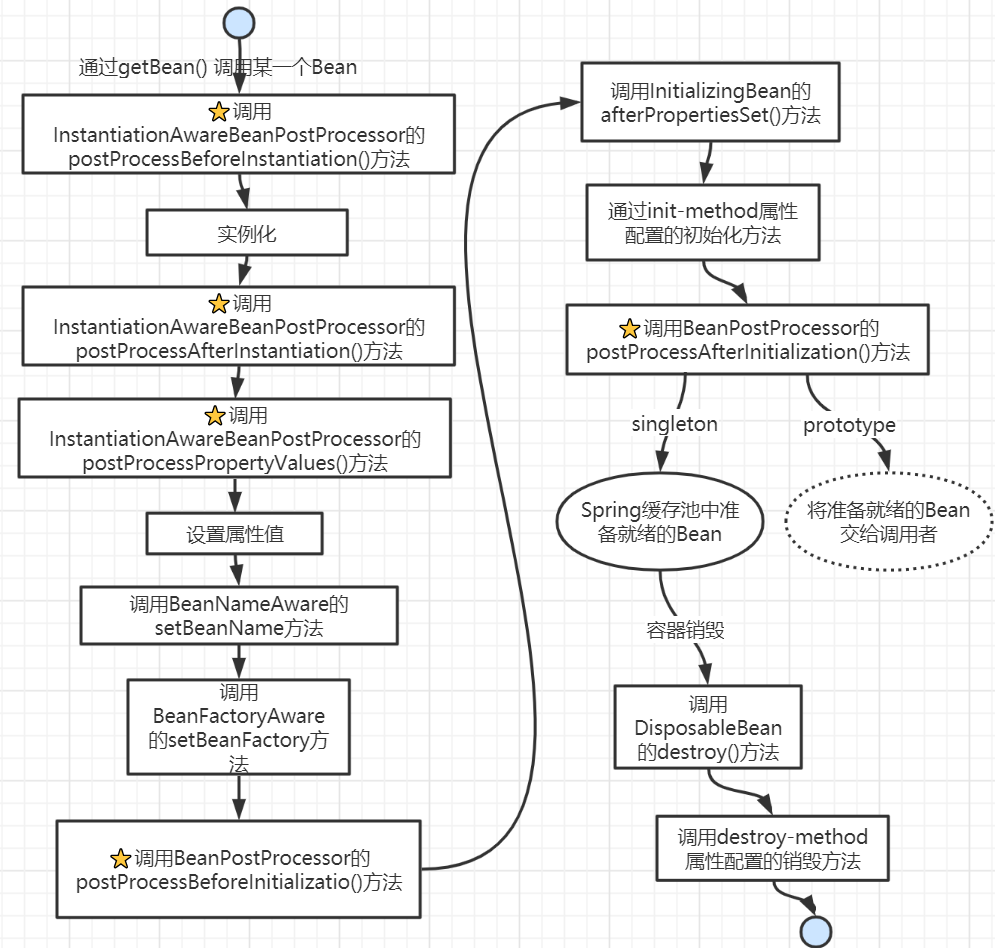
## Bean的生命周期

Spring中，可以从两个层面定义Bean的生命周期：

1. Bean的作用范围
2. 实例化Bean时所经历的一系列阶段

### BeanFactory中Bean的生命周期

#### 生命周期图解



#### 具体过程

1. 当调用者通过getBean(beanName)向容器请求某一个Bean时，如果容器注册了InstantiationAwareBeanPostProcessor接口，则在实例化Bean之前，将调用接口的postProcessBeforeInstantiation()方法。
2. 根据配置情况调用Bean构造函数或者工厂方法实例化Bean。
3. 如果容器注册了InstantiationAwareBeanPostProcessor接口,则实例化Bean之后，调用该接口的postProcessAfterInstantiation()方法，可以在这里对已经实例化的对象进行一些“梳妆打扮”。
4. 如果Bean配置了属性信息，q