# BeanInfo

实体信息，可以通过调用如下方法获取：

|  |
| --- |
| BeanInfo beanInfo = Introspector.*getBeanInfo*(BeanInfoTest.class); |

BeanInfoTest.class为我们指定的实体类

## BeanDescriptor

通过BeanInfo可以或取这个实体的描述信息：

|  |
| --- |
| BeanDescriptor beanDescriptor = beanInfo.getBeanDescriptor(); beanDescriptor.getBeanClass();beanDescriptor.getName(); |

## 1.2 PropertyDescriptor

获取实体下的属性信息，包含可读函数和可写函数

|  |
| --- |
| PropertyDescriptor[] propertyDescriptors = beanInfo.getPropertyDescriptors(); *//获取包装对象具体的属性* for (PropertyDescriptor propertyDescriptor:propertyDescriptors) {  if("class".equalsIgnoreCase(propertyDescriptor.getName()) || StringUtils.*isEmpty*(propertyDescriptor.getName())){  continue;  }  *//属性名* System.*out*.println(propertyDescriptor.getName());  *//属性类型* System.*out*.println(propertyDescriptor.getPropertyType());  *//属性对应的setter方法* System.*out*.println(propertyDescriptor.getReadMethod());  *////属性对应的getter方法* System.*out*.println(propertyDescriptor.getWriteMethod());  } |

## 1.3 TypeDescriptor

这个是spring内部定义的一个实体类，

# 二、属性编辑器PropertyEditor

属性编辑器就是对属性值进行转换，java默认的实现是PropertyEditorSupport

它只能把待转化的对象转化为String类型的，其他情况它都不支持了。

比如我们想把字符转化为Integer，则需要自定义一个PropertyEditor，并且实现PropertyEditorSuppor类，重写setAsText和getAsText方法即可。

Eg：

|  |
| --- |
| public class CharsetEditor extends PropertyEditorSupport {   @Override  public void setAsText(String text) throws IllegalArgumentException {  if (StringUtils.*hasText*(text)) {  setValue(Charset.*forName*(text));  }  else {  setValue(null);  }  }  @Override  public String getAsText() {  Charset value = (Charset) getValue();  return (value != null ? value.name() : "");  }  } |

以上demo可以把字符“utf-8”转换为Charset类型的。

**注意：在spring内部创建了大量的自定义编辑器，基本能满足我们绝大多数的类型转换工作，并且也都已经封装在PropertyEditorRegistrySupport类里了。**

## 2.1 PropertyEditorRegistrySupport

在这个类里默认创建了大量常使用的属性编辑器。当然也预留了客户自定义的类型编辑器。

Demo：

|  |
| --- |
| public static void main(String[] args) {  PropertyEditorRegistrySupport pe = new PropertyEditorRegistrySupport();  pe.registerDefaultEditors();  PropertyEditor defaultEditor = pe.getDefaultEditor(Integer.class);*//获取Integer类型的转换器* defaultEditor.setAsText("9"); *//转换前的文本* Object value = defaultEditor.getValue(); *//已经把文本子字符“9”转换为数字9了*  *}* |

## 2.2 TypeConverterDelegate（TypeConverterSupport）

关于属性的值，在做类型转换的时候，spring对此进一步做了封装，首选，使用者可自定义自己的转换规则，只要实现ConversionService接口即可，其次也可以自定义自己的属性编辑器，最后兜底的是内部定义的默认属性编辑器，

这个过程被进一步的封装在TypeConverterDelegate类了，

Demo：

|  |
| --- |
| TypeConverterDelegate de = new TypeConverterDelegate(new PropertyEditorRegistrySupport(),null); Integer integer = de.convertIfNecessary(null, null, "8", Integer.class, TypeDescriptor.*valueOf*(Integer.class)); System.*out*.println(integer); |

# 三、对象属性赋值

Spring ioc的核心点是创建对象，然后给对象里的属性一一赋值，这里涉及两点，一个是属性值的类型转换，二是把转化后的值赋值给指定对象的属性上。我们上面已经讲过了类型转化的事情，下面我们讲讲赋值的事。

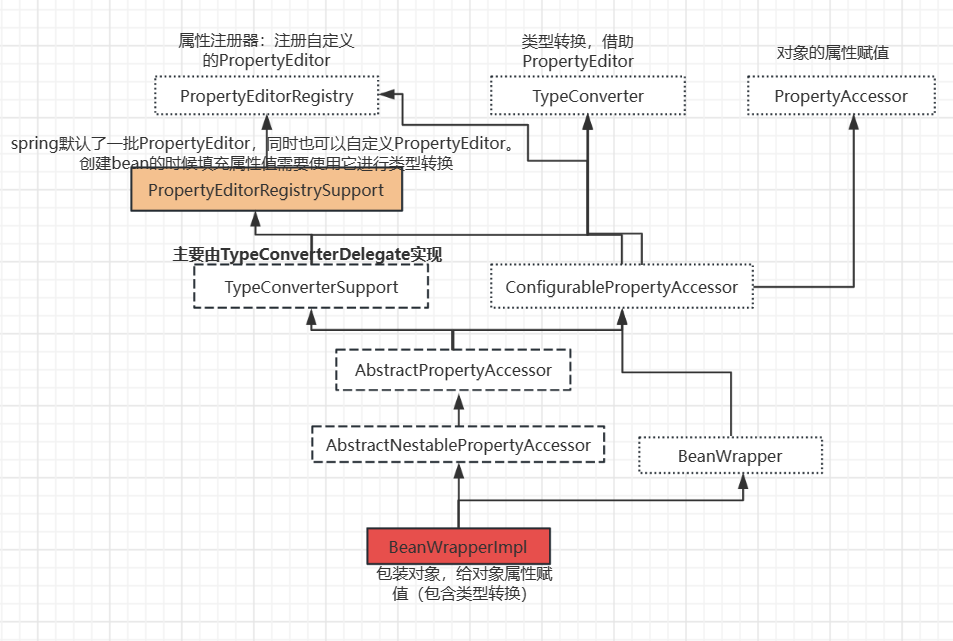
## 3.1 PropertyAccessor

这个是spring定义的父接口，核心实现方法在：

AbstractNestablePropertyAccessor#setPropertyValue

AbstractNestablePropertyAccessor是个抽象类，具体的真实实现类是BeanWrapperImp类：

# 四、BeanWrapperImp



通过上面的介绍，我们可以发现BeanWrapperImpl类包装对象的目的是，给对象属性值进行类型转换，其次是对被包装的对象属性赋值。

Demo：

|  |
| --- |
| public static void main(String[] args) throws Exception {BeanWrapper beanWrapper = new BeanWrapperImpl(new BeanInfoTest()); //包装对象  HashMap<String, List<Integer>> param = new HashMap<>();  List<Integer> tt = new ArrayList<>();  tt.add(1);tt.add(2);param.put("test",tt);  *//给包装对象的属性赋值，支持map  // beanWrapper.setPropertyValue("param",param);*  *//给包装对象的属性赋值，支持PropertyValue*  beanWrapper.setPropertyValue(new PropertyValue("age","18"));   Map map = new HashMap<>();  map.put("name", "zhangmm");  *//可以把所有的属性值放入map中，自动帮做类型转换并赋值* beanWrapper.setPropertyValues(map); *//给包装对象的属性赋值，支持MutablePropertyValues*  MutablePropertyValues ps = new MutablePropertyValues();  ps.addPropertyValue("param",param);  beanWrapper.setPropertyValues(ps,true);  //通过以上方式，就可以给被包装的对象属性赋值了，下面打印出赋值的结果。  BeanInfoTest wrappedInstance = (BeanInfoTest) beanWrapper.getWrappedInstance(); System.out.println(wrappedInstance.getName()); //已经赋值了 zhangmm System.out.println(wrappedInstance.getParam()); System.out.println(wrappedInstance.getAge());  *//做类型转换:spring内置了大量了常用的属性编辑器，可自动完成类型转换工作* BeanWrapper beanWrapper01 = new BeanWrapperImpl();  Integer integer = beanWrapper01.convertIfNecessary("8", Integer.class); } |

# 五、创建对象

## 5.1 InstantiationStrategy

创建实例对象的接口，不管对于有参还是无参构造方法，不管是普通类还是cglib创建的类，都是通过这个接口实现的，它有两个具体的实现类：

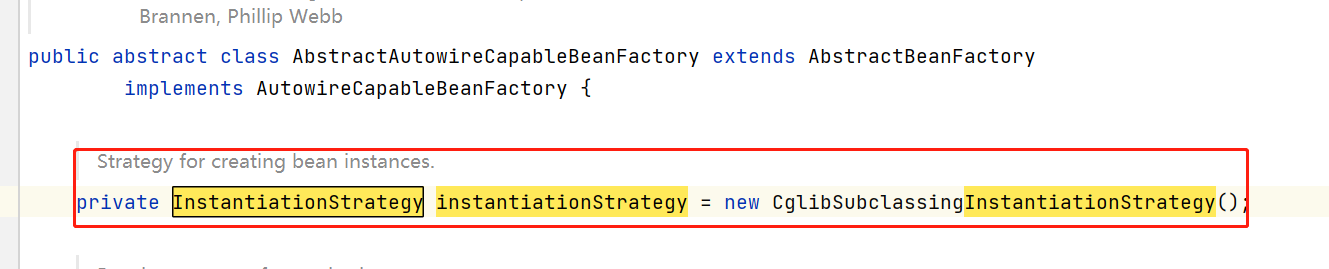
|  |  |
| --- | --- |
| CglibSubclassingInstantiationStrategy | Cglib创建的类的实例化 |
| SimpleInstantiationStrategy | 普通类的实例化 |

Demo：

|  |
| --- |
| public static void test3( Class<?> clazz ){  RootBeanDefinition bd = new RootBeanDefinition(null);  bd.setBeanClass(clazz);   SimpleInstantiationStrategy simpleInstantiationStrategy = new SimpleInstantiationStrategy();  Object instantiate = simpleInstantiationStrategy.instantiate(bd,null,null);  System.*out*.println(instantiate); } |

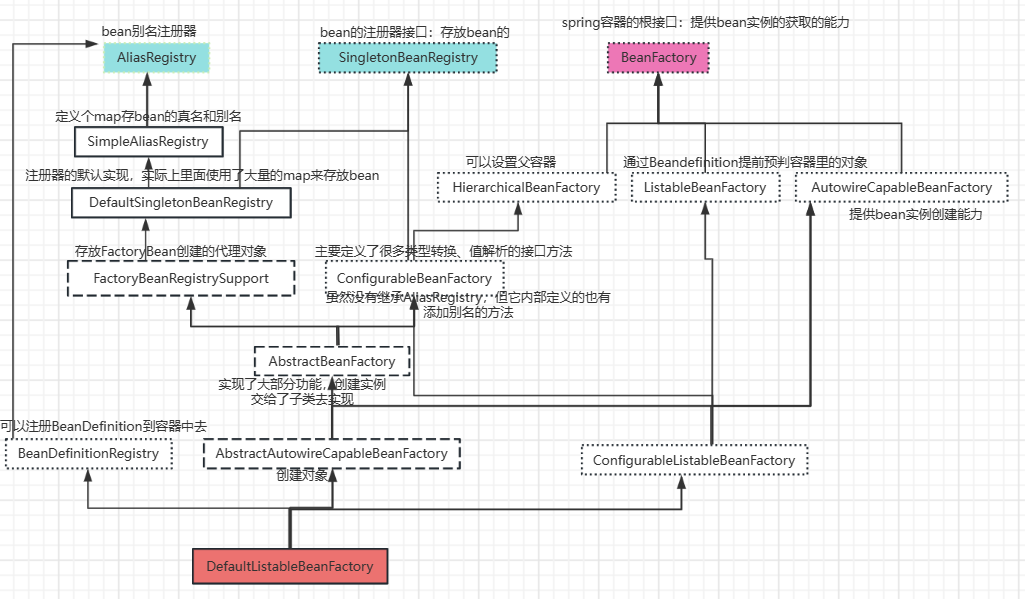
在spring的源码里，AbstractAutowireCapableBeanFactory类里定义了

InstantiationStrategy instantiationStrategy的实现类：



实例创建的工作就是它完成了的。

# 六、DefaultListableBeanFactory



DefaultListableBeanFactory是整个spring体系的中心，一切的操作都离不开它，上图说明了它的继承关系，下面依次从上往下介绍，最后总结说明DefaultListableBeanFactory具有的功能。

## 6.1 AliasRegistry

AliasRegistry：注册别名的接口，key是别名，value是真实的bean名字

|  |  |
| --- | --- |
| 核心方法 |  |
| void registerAlias(String name, String alias); | 注册一个bean的别名 |
| void removeAlias(String alias); | 移除所有的别名 |
| boolean isAlias(String name); | 判断一个名字是否是一个bean的别名 |
| String[] getAliases(String name) | 获取所有bean的别名 |

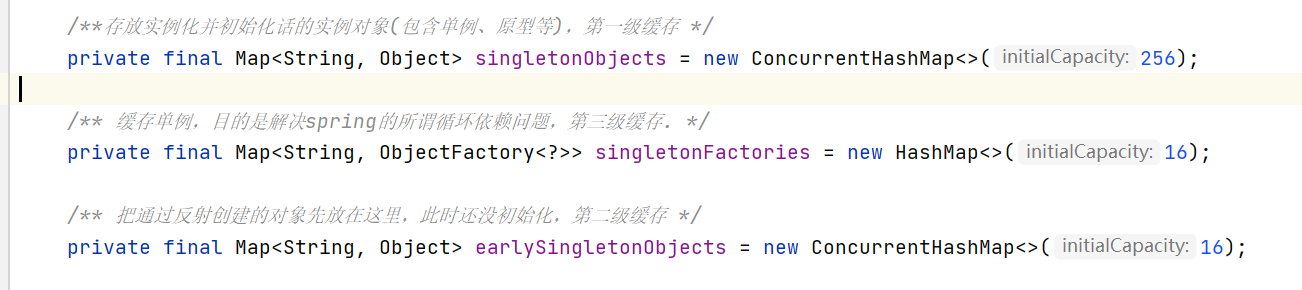
### SimpleAliasRegistry

这个是是Aliasregistry接口的简单实现，定义了一个map属性去存key=别名，value=bean的真实名，进而封装到上面的Aliasregistry接口方法的实现。

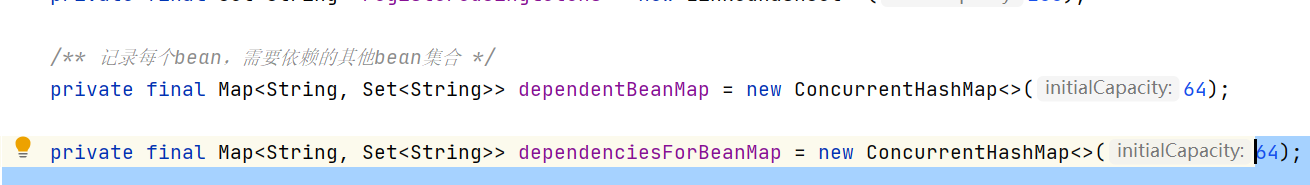
### DefaultSingletonBeanRegistry

主要功能是存贮需要使用的bean，作为spring ioc的大脑存在。

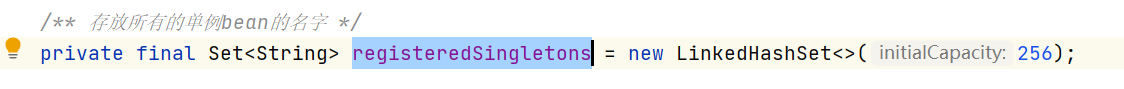
这里定义了几个map，著名的spring ioc的所谓三级缓存，都是定义在这个集合中的，



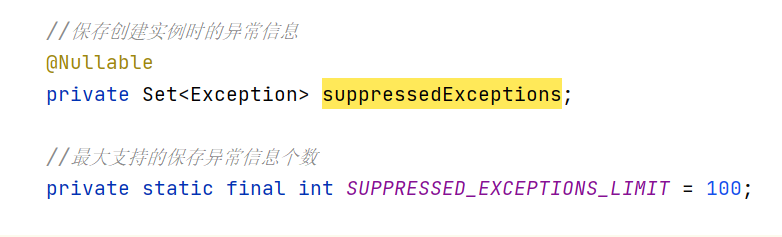
除此外还定义了一个bean依赖其他对象的几个map



记录单例bean的名字集合：



存放创建实例过程中碰到的异常信息：



需要被销毁的bean集合：



### FactoryBeanRegistrySupport

存放通过FactoryBean创建的代理的真实对象的bean---对象



## 6.2 SingletonBeanRegistry

SingletonBeanRegistry：单例bean注册，通常是BeanFactory的实现类实现此接口，目的是管理单例的注册功能。

核心方法：

|  |  |
| --- | --- |
| registerSingleton(String beanName, Object singletonObject) | 给定name和对象，注册单例 |
| Object getSingleton(String beanName) | 通过名字获取单例对象 |
| boolean containsSingleton(String beanName) | 判断是否包含单例对象 |
| String[] getSingletonNames() | 获取所有单例bean的名字 |
| int getSingletonCount() | 获取容器中单例bean的总个数 |
| Object getSingletonMutex() | 获取所有单例对象，实际返回的是个集合对象。 |

它的直接实现类是DefaultSingletonBeanRegistry，所以它同时也具有注册单例的功能。

子接口是ConfigurableBeanFactory

## 6.3 BeanFactory

BeanFactory：spring容器的最顶级接口，主要功能是获取实例

核心方法：

|  |  |
| --- | --- |
| Object getBean(String name) | 给定名字获取对象 |
| <T> T getBean(String name, Class<T> requiredType) | 给定名字和类型，获取对象 |
| <T> T getBean(Class<T> requiredType) | 给定类型，获取对象 |
| boolean containsBean(String name) | 判断spring容器中是否包含给定类型的对象 |
| boolean isSingleton(String name) | 给定名字判断是否是单例 |
| boolean isPrototype(String name) | 给定名字判断是否是原型 |
| Class<?> getType(String name) | 给定名字，判断对象的类型 |
| Class<?> getType(String name, boolean allowFactoryBeanInit) |  |
| String[] getAliases(String name) | 获取所有别名 |
| <T> ObjectProvider<T> getBeanProvider(Class<T> requiredType) | 给定类类型，获取ObjectProvider对象 |
| <T> ObjectProvider<T> getBeanProvider(ResolvableType requiredType) | 给定类型，获取ObjectProvider对象 |

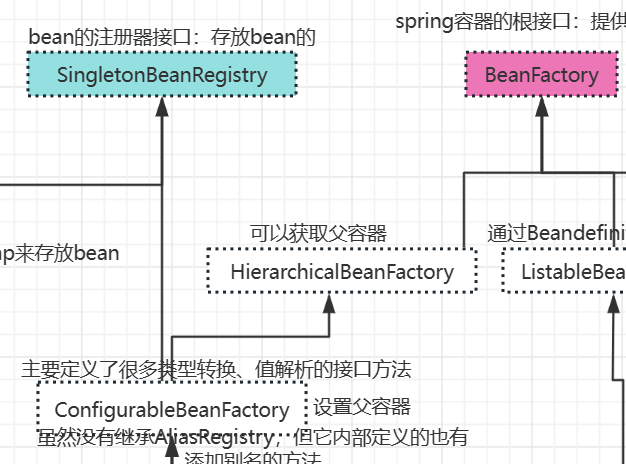
### HierarchicalBeanFactory

它的核心功能是获取父容器，在feign中，最最核心的原理就是一个主容器，多个服务，对个子容器，就使用了这里的知识点。

核心方法：

|  |  |
| --- | --- |
| BeanFactory getParentBeanFactory() | 获取父容器 |

#### ConfigurableBeanFactory



它继承了SingletonBeanRegistry和HierarchicalBeanFactory，

所以它也具有单例注册器的功能，此外，它还能获取父容器。

核心方法有：

|  |  |
| --- | --- |
| BeanExpressionResolver getBeanExpressionResolver() | 获取解析el表达式的解析类 |
| void setBeanExpressionResolver(@Nullable BeanExpressionResolver resolver); | 设置解析el表达式的解析类 |
| void setTypeConverter(TypeConverter typeConverter); | 设置类型转换类 |
| TypeConverter getTypeConverter(); | 获取类型转换类 |
| ConversionService getConversionService(); | 1、给定值和类型做类型转换，2、判断是否可以做类型转换 |
| void setConversionService(@Nullable ConversionService conversionService); | 1、给定值和类型做类型转换，2、判断是否可以做类型转换 |
| void registerCustomEditor(Class<?> requiredType, Class<? extends PropertyEditor> propertyEditorClass); | 设置客户自定义类型解析器 |
| void addPropertyEditorRegistrar(PropertyEditorRegistrar registrar); | 添加属性解析器 |
| void setParentBeanFactory(BeanFactory parentBeanFactory) | 设置父容器，和父接口对应 |
| void addEmbeddedValueResolver(StringValueResolver valueResolver); | 解析@value等注解里的值，并做类型转化 |
| boolean hasEmbeddedValueResolver(); |  |
| String resolveEmbeddedValue(String value); |  |
| void addBeanPostProcessor(BeanPostProcessor beanPostProcessor); | 添加BeanPostProcessor实现类 |
| int getBeanPostProcessorCount(); | 返回容器中BeanPostProcessor实现类个数 |

##### BeanExpressionResolver

##### TypeConverter

##### ConversionService

##### PropertyEditorRegistrar

##### BeanPostProcessor

##### StringValueResolver

### ListableBeanFactory

通过Beandefinition提前预判容器里的对象

核心方法：

|  |  |
| --- | --- |
| boolean containsBeanDefinition(String beanName); | 给定一个名字，判断是否容器中国有BeanDefinition |
| int getBeanDefinitionCount() | 返回容器中所有的BeanDefinition个数 |
| String[] getBeanDefinitionNames() | 返回容器中所有的BeanDefinition集合 |
| <T> Map<String, T> getBeansOfType(@Nullable Class<T> type | 给定类型，返回所有匹配的实例集合 |
| String[] getBeanNamesForAnnotation(Class<? extends Annotation> annotationType) | 给定注解，返回所有使用此注解的类的beanName集合 |
| Map<String, Object> getBeansWithAnnotation(Class<? extends Annotation> annotationType) | 给定注解，返回所有使用此注解的类的集合，key是beanName，value是对应的实例 |
| <A extends Annotation> A findAnnotationOnBean(String beanName, Class<A> annotationType)  throws NoSuchBeanDefinitionException; | 给定beanName、类类型，返回所有匹配的注解 |

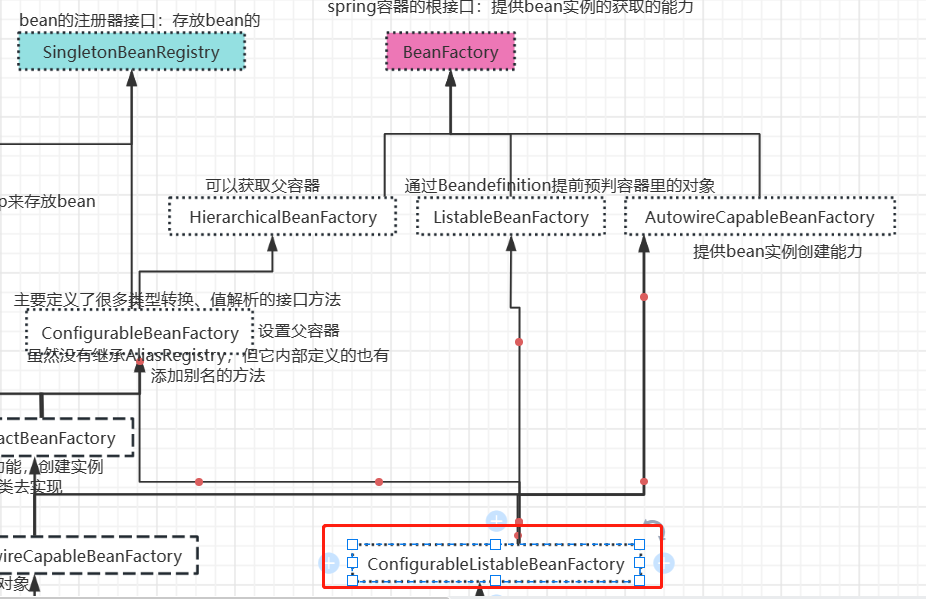
### AutowireCapableBeanFactory

提供bean实例创建能力

核心方法：

|  |  |
| --- | --- |
| <T> T createBean(Class<T> beanClass) throws BeansException; | 给定类类型，创建实例 |

### ConfigurableListableBeanFactory



ConfigurableListableBeanFactory这个接口继承了ListableBeanFactory、AutowireCapableBeanFactory、ConfigurableBeanFactory接口，所以这三个接口的功能它都继承了，如ListableBeanFactory具有通过Beandefinition提前预判容器里的对象；AutowireCapableBeanFactory提供bean实例创建能力；ConfigurableBeanFactory具有单例注册器的功能，获取父容器，类型转换，值解析等能力。此外，它还定义了一个核心能力：确定所有非懒加载的对象都已经实例化，对应的方法是preInstantiateSingletons，具体的实现是在DefaultListableBeanFactory类完成的，后续我门在讲DefaultListableBeanFactory类的时候重点讲解此方法。核心方法：

|  |  |
| --- | --- |
| void ignoreDependencyType(Class<?> type) | 依赖注入的时候，忽略指定的类 |
| void ignoreDependencyInterface(Class<?> ifc) | 依赖注入的时候，忽略指定的接口 |
| BeanDefinition getBeanDefinition(String beanName) | 根据beanName，获取BeanDefinition |
| Iterator<String> getBeanNamesIterator() | 获取所有的beanName |
| void freezeConfiguration() | 一个信号变量，表示不在更改实例化的对象 |
| boolean isConfigurationFrozen() | 判断是否已经加了上面的信号量 |
| void preInstantiateSingletons() throws BeansException | 确保所有非拦截在的对象都已经实例化 |

### AbstractBeanFactory

它实现了大部分的功能，包含AliasRegistry（别名注册器）、SingletonBeanRegistry（单例注册器）、BeanFactory（获取单例）、HierarchicalBeanFactory（获取父容器）、ConfigurableBeanFactory（设置父容器、类型转换、el表达式解析等）

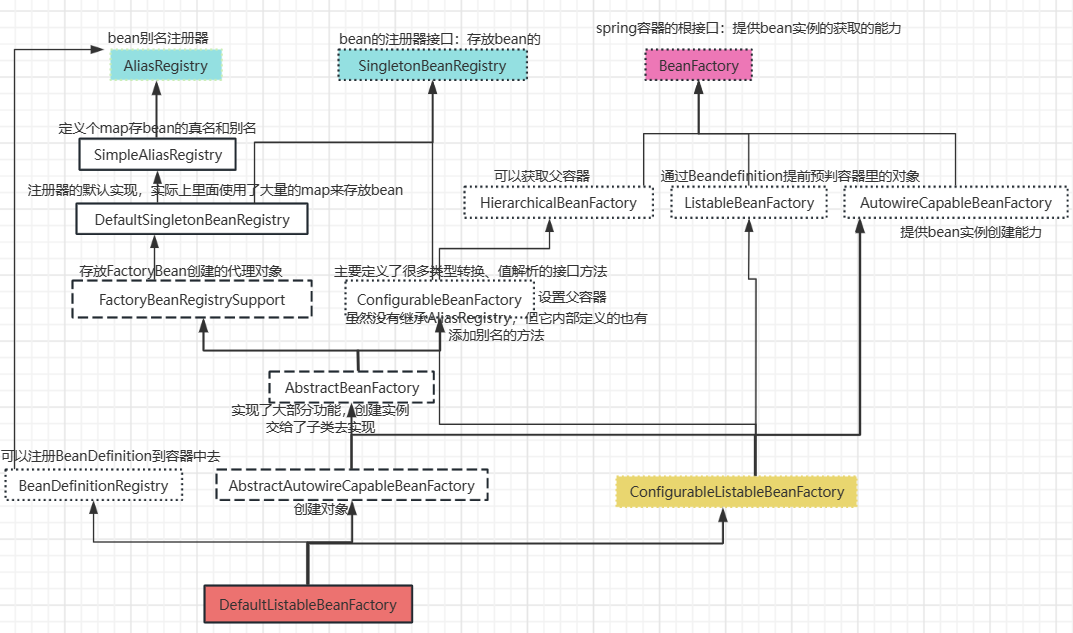
但是创建实例的功能，还是交给子类AbstractAutowireCapableBeanFactory去实现了

### AbstractAutowireCapableBeanFactory

核心是实现了创建对象的操作。

### DefaultListableBeanFactory

集大成者，这个家伙实现了以上介绍的所有类，所以它具有他们每一个类的功能，下面依次介绍。



下面在分别介绍下它主要实现的接口。

**AliasRegistry**：注册别名的接口，key是别名，value是真实的bean名字

|  |  |
| --- | --- |
| 核心方法 |  |
| void registerAlias(String name, String alias); | 注册一个bean的别名 |
| void removeAlias(String alias); | 移除所有的别名 |
| boolean isAlias(String name); | 判断一个名字是否是一个bean的别名 |
| String[] getAliases(String name) | 获取所有bean的别名 |

**SingletonBeanRegistry**：单例bean注册，通常是BeanFactory的实现类实现此接口，目的是管理单例的注册功能。

核心方法：

|  |  |
| --- | --- |
| registerSingleton(String beanName, Object singletonObject) | 给定name和对象，注册单例 |
| Object getSingleton(String beanName) | 通过名字获取单例对象 |
| boolean containsSingleton(String beanName) | 判断是否包含单例对象 |
| String[] getSingletonNames() | 获取所有单例bean的名字 |
| int getSingletonCount() | 获取容器中单例bean的总个数 |
| Object getSingletonMutex() | 获取所有单例对象，实际返回的是个集合对象。 |

**BeanFactory**：spring容器的最顶级接口，主要功能是获取实例

核心方法：

|  |  |
| --- | --- |
| Object getBean(String name) | 给定名字获取对象 |
| <T> T getBean(String name, Class<T> requiredType) | 给定名字和类型，获取对象 |
| <T> T getBean(Class<T> requiredType) | 给定类型，获取对象 |
| boolean containsBean(String name) | 判断spring容器中是否包含给定类型的对象 |
| boolean isSingleton(String name) | 给定名字判断是否是单例 |
| boolean isPrototype(String name) | 给定名字判断是否是原型 |
| Class<?> getType(String name) | 给定名字，判断对象的类型 |
| Class<?> getType(String name, boolean allowFactoryBeanInit) |  |
| String[] getAliases(String name) | 获取所有别名 |
| <T> ObjectProvider<T> getBeanProvider(Class<T> requiredType) | 给定类类型，获取ObjectProvider对象 |
| <T> ObjectProvider<T> getBeanProvider(ResolvableType requiredType) | 给定类型，获取ObjectProvider对象 |

**HierarchicalBeanFactory**它的核心功能是获取父容器，在feign中，最最核心的原理就是一个主容器，多个服务，对个子容器，就使用了这里的知识点。

核心方法：

|  |  |
| --- | --- |
| BeanFactory getParentBeanFactory() | 获取父容器 |

**ConfigurableBeanFactory**它继承了SingletonBeanRegistry和HierarchicalBeanFactory，

所以它也具有单例注册器的功能，此外，它还能获取父容器。

核心方法有：

|  |  |
| --- | --- |
| BeanExpressionResolver getBeanExpressionResolver() | 获取解析el表达式的解析类 |
| void setBeanExpressionResolver(@Nullable BeanExpressionResolver resolver); | 设置解析el表达式的解析类 |
| void setTypeConverter(TypeConverter typeConverter); | 设置类型转换类 |
| TypeConverter getTypeConverter(); | 获取类型转换类 |
| ConversionService getConversionService(); | 1、给定值和类型做类型转换，2、判断是否可以做类型转换 |
| void setConversionService(@Nullable ConversionService conversionService); | 1、给定值和类型做类型转换，2、判断是否可以做类型转换 |
| void registerCustomEditor(Class<?> requiredType, Class<? extends PropertyEditor> propertyEditorClass); | 设置客户自定义类型解析器 |
| void addPropertyEditorRegistrar(PropertyEditorRegistrar registrar); | 添加属性解析器 |
| void setParentBeanFactory(BeanFactory parentBeanFactory) | 设置父容器，和父接口对应 |
| void addEmbeddedValueResolver(StringValueResolver valueResolver); | 解析@value等注解里的值，并做类型转化 |
| boolean hasEmbeddedValueResolver(); |  |
| String resolveEmbeddedValue(String value); |  |
| void addBeanPostProcessor(BeanPostProcessor beanPostProcessor); | 添加BeanPostProcessor实现类 |
| int getBeanPostProcessorCount(); | 返回容器中BeanPostProcessor实现类个数 |

**ListableBeanFactory**通过Beandefinition提前预判容器里的对象

核心方法：

|  |  |
| --- | --- |
| boolean containsBeanDefinition(String beanName); | 给定一个名字，判断是否容器中国有BeanDefinition |
| int getBeanDefinitionCount() | 返回容器中所有的BeanDefinition个数 |
| String[] getBeanDefinitionNames() | 返回容器中所有的BeanDefinition集合 |
| <T> Map<String, T> getBeansOfType(@Nullable Class<T> type | 给定类型，返回所有匹配的实例集合 |
| String[] getBeanNamesForAnnotation(Class<? extends Annotation> annotationType) | 给定注解，返回所有使用此注解的类的beanName集合 |
| Map<String, Object> getBeansWithAnnotation(Class<? extends Annotation> annotationType) | 给定注解，返回所有使用此注解的类的集合，key是beanName，value是对应的实例 |
| <A extends Annotation> A findAnnotationOnBean(String beanName, Class<A> annotationType)  throws NoSuchBeanDefinitionException; | 给定beanName、类类型，返回所有匹配的注解 |

**AutowireCapableBeanFactory**提供bean实例创建能力

核心方法：

|  |  |
| --- | --- |
| <T> T createBean(Class<T> beanClass) throws BeansException; | 给定类类型，创建实例 |

ConfigurableListableBeanFactory这个接口继承了ListableBeanFactory、AutowireCapableBeanFactory、ConfigurableBeanFactory接口，所以这三个接口的功能它都继承了，如ListableBeanFactory具有通过Beandefinition提前预判容器里的对象；AutowireCapableBeanFactory提供bean实例创建能力；ConfigurableBeanFactory具有单例注册器的功能，获取父容器，类型转换，值解析等能力。此外，它还定义了一个核心能力：确定所有非懒加载的对象都已经实例化，对应的方法是preInstantiateSingletons，具体的实现是在DefaultListableBeanFactory类完成的，后续我门在讲DefaultListableBeanFactory类的时候重点讲解此方法。核心方法：

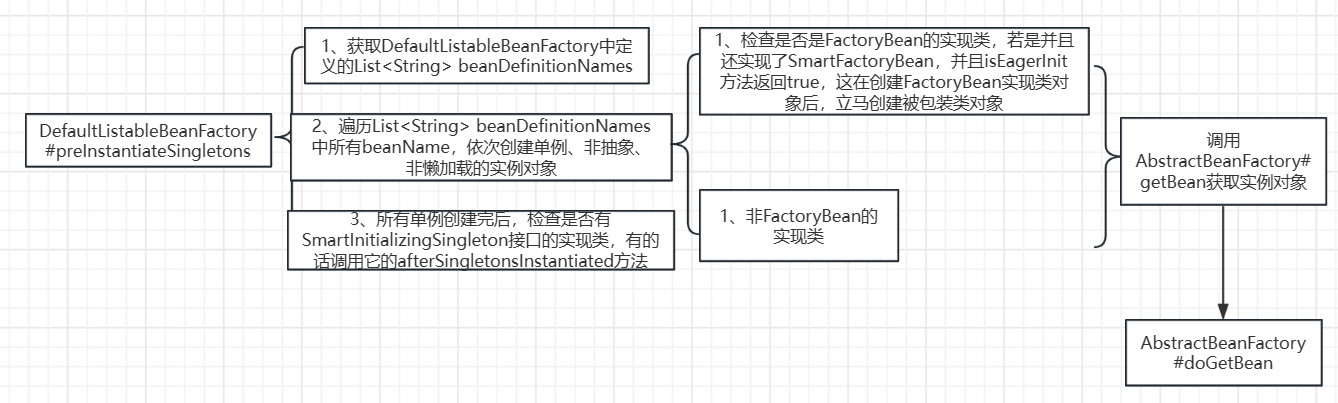
|  |  |
| --- | --- |
| void ignoreDependencyType(Class<?> type) | 依赖注入的时候，忽略指定的类 |
| void ignoreDependencyInterface(Class<?> ifc) | 依赖注入的时候，忽略指定的接口 |
| BeanDefinition getBeanDefinition(String beanName) | 根据beanName，获取BeanDefinition |
| Iterator<String> getBeanNamesIterator() | 获取所有的beanName |
| void freezeConfiguration() | 一个信号变量，表示不在更改实例化的对象 |
| boolean isConfigurationFrozen() | 判断是否已经加了上面的信号量 |
| **void preInstantiateSingletons() throws BeansException** | 确保所有非拦截在的对象都已经实例化 |

下面重点讲解 DefaultListableBeanFactory#preInstantiateSingletons方法，看看它是如何确保所有非懒加载的对象完成实例化的。

#### DefaultListableBeanFactory#preInstantiateSingletons

它做了三件事情：

1. 获取本类中收集到的List<String> beanDefinitionNames，copye到一个局部变量中；
2. 循环上面的集合，对非赖加载、非抽象类并且是单例的类进行实例化；
3. 循环实例化后的所有对象，对应SmartInitializingSingleton接口的实现类，调用afterSingletonsInstantiated方法：



重点说两个接口：

##### SmartFactoryBean

它是FactoryBean接口的子类，我们知道我们在创建实例的时候，经常可能需要先对它进行一些预处理，然后在创建该实例，此时就需要定义FactoryBean接口的实现类即可，但是FactoryBean的实现类不是在spring 容器初始化的时候就创建实现类的对象，是在调用它的getObject方法的时候再去创建的，

如果我们需要在spring ioc容器初始化的时候就去创建它的实现类对象，那么我们就要实现SmartFactoryBean接口，并且设置isEagerInit方法返回为true，这样在spring 容器初始化的时候就去创建实现类的实例对象。

举例说明：

##### SmartInitializingSingleton

所有单例实例化完后调用的方法，比如我们想在spring 容器对所有实例都初始化完之后，干些事情，此时就可以写一个SmartInitializingSingleton接口的实现类，在它的afterSingletonsInstantiated方法里写自己的实现即可。

举例说明：

下面看下DefaultListableBeanFactory#preInstantiateSingletons方法的第二个核心步骤，依次创建实例的过程：

##### AbstractBeanFactory#getBean 创建实例

###### AbstractBeanFactory#doGetBean

它转调的**是AbstractBeanFactory#doGetBean**方法：

1、若beanName前有&符号，这全部去掉，这是针对FactoryBean创建的实例，会在前面加上&

注意：若自己定义的beanName，就会出问题了。

2、先去缓存中查看是否已经创建好这个实例，若创建好，就直接取出来；

3、创建实例

3.1 检查：循环依赖不支持原型模式；

3.2 先检查父工厂是否已经有了，有的话就直接取值，取到看就返回，没有的话就继续往下走；

3.3标记这个bean正在创建；

3.4 合并BeanDefinition：被实例化的类若也有父实现类，这这里把从父类继承的属性定义全部继承过来；

3.5获取创建该bean的所有依赖实例，注意这里是mbd.getDependsOn()是通过@DependOn显示强加的，这种强依赖，spring是不支持的。

**所以平时听到的Spring解决循环依赖问题只是针对@Autowire注解。**

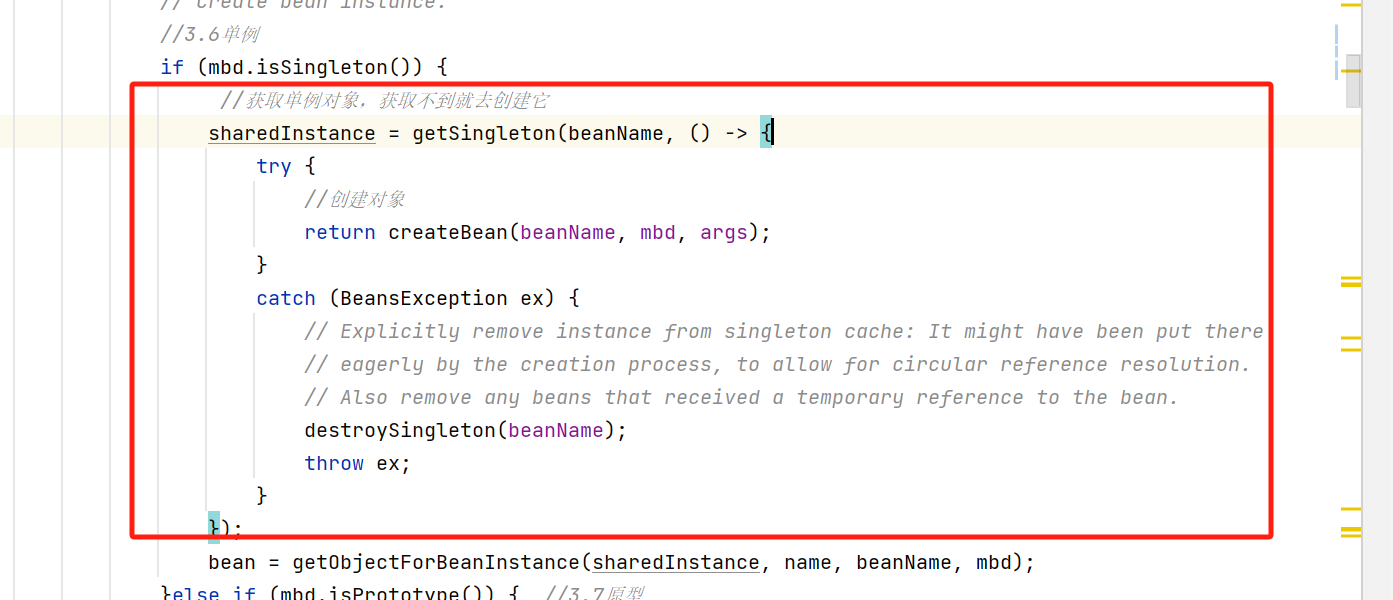
**对于@DependOn注解，存在循环依赖还是会报错的（抛出BeanCreationException异常）**

3.6单例的创建实例过程；

3.7原型的创建实例过程；

3.8其他scope的创建实例过程；

我们重点看下单例的创建过程：



核心是调用**DefaultSingletonBeanRegistry#getSingleton**（String，ObjectFactory）方法，

上面的代码入参是beanName，ObjectFactory对象实际上是一个Lambda表达式；

查看DefaultSingletonBeanRegistry#getSingleton源码：



它会调用传入的Lambda表达式的getObject方法，从而创建实例，然后把创建好的实例放入一级缓存中。

下面我们就看下传入的Lambda表达式被调用的代码，



调用的是AbstractBeanFactory#createBean方法：转调的是AbstractAutowireCapableBeanFactory#createBean方法

AbstractAutowireCapableBeanFactory#createBean

1、给用户留了一个口子，可以自定义类的实例过程。

若RootBeanDefinition类的beforeInstantiationResolved属性设置为true，并且有InstantiationAwareBeanPostProcessor实现类，这调用

**InstantiationAwareBeanPostProcessor.postProcessBeforeInstantiation**(Class<?> beanClass, String beanName)创建实例

InstantiationAwareBeanPostProcessor.postProcessAfterInstantiation(Object bean, String beanName)若上面创建的实例不为空，则继续调用这里的方法

后续也不用走ioc的方法去创建实例了，这里相当于留了一个口子，可以用户自定义类的实例过程。

2、若没有RootBeanDefinition类的beforeInstantiationResolved属性设置为true，这继续走ioc的核心代码去创建实例。调用的是AbstractAutowireCapableBeanFactory#doCreateBean方法

AbstractAutowireCapableBeanFactory#doCreateBean

1、创建BeanWrapper实现类实例BeanWrapperImpl对象，一个beanName，对应一个BeanWrapperImpl对象。保存在

AbstractAutowireCapableBeanFactory类的ConcurrentMap<String, BeanWrapper> factoryBeanInstanceCache属性中；

2、通过反射创建实例，被包装在BeanWrapperImpl对象中，

以上这两个步骤对应的是AbstractAutowireCapableBeanFactory#createBeanInstance方法；

3、为了解决使用@Autoiwed注解导致的循环依赖问题，先把创建的对象，放入三级缓存中，此时存入到ObjectFactory实现类是Lambda表达式：



4、给对象属性填充值

4.1 调用InstantiationAwareBeanPostProcessor的postProcessAfterInstantiation方法:

4.2 判断依赖注入模式，依据名字还是类型

4.3 调用InstantiationAwareBeanPostProcessor 的postProcessProperties或postProcessPropertyValues方法

典型案例：

//AutowiredAnnotationBeanPostProcessor --->解析@Autowired、@Value注解

//CommonAnnotationBeanPostProcessor --->解析@Resource注解

//ImportAwareBeanPostProcessor 处理ImportAware实现类的bean

//AbstractAutoProxyCreator //代理使用

4.4 对属性赋值 applyPropertyValues方法：

包括对属性值类型的转换，特别是@Value里的值，需要做解析及类型转换；

最后调用

4.5 BeanWrapperImpl对象的setPropertyValues方法，对被包装类的属性赋值。

5、给对象初始化

5.1 调用BeanNameAware、BeanClassLoaderAware、BeanFactoryAware实现类的方法

5.2 调用BeanPostProcessor实现类的BeforeInitialization方法

5.3 InitializingBean#afterPropertiesSet方法及 执行用户自定义的init方法

5.4 执行BeanPostProcessor#postProcessAfterInitialization方法

6、

7、

# Spring ioc的循环依赖问题

