

04 | 增强IoC容器：如何让我们的Spring支持注解？

2023-03-20 郭屹 来自北京

《手把手带你写一个MiniSpring》



你好，我是郭屹。

上节课我们通过一系列的操作使 XML 使配置文件生效，然后实现了 Spring 中 Bean 的构造器注入与 setter 注入，通过引入“早期毛胚 Bean”的概念解决了循环依赖的问题，我们还为容器增加了 Spring 中的一个核心方法 `refresh()`，作为整个容器启动的入口。现在我们的容器已经初具模型了，那如何让它变得更强大，从种子长成一株幼苗呢？

这节课我们就来实现一个增强版的 IoC 容器，支持通过注解的方式进行依赖注入。注解是我们在编程中常用的技术，可以减少配置文件的内容，便于管理的同时还能提高开发效率。所以这节课我们将**实现 Autowired 注解，并用这个方式进行依赖注入**。

目录结构

我们手写 MiniSpring 的目的是更好地学习 Spring。因此，我们会时不时回头来整理整个项目的目录结构，和 Spring 保持一致。

现在我们先参考 Spring 框架的结构，来调整我们的项目结构，在 beans 目录下新增 factory 目录，factory 目录中则新增 xml、support、config 与 annotation 四个目录。

 复制代码

```
1 |— beans
2 |   |— factory
3 |       |— xml
4 |       |— support
5 |       |— config
6 |       |— annotation
```

接下来将之前所写的类文件移动至新增目录下，你可以看一下移动后的结构。

 复制代码

```
1 factory — BeanFactory.java
2 factory.xml — XmlBeanDefinitionReader.java
3 factory.support — DefaultSingletonBeanRegistry.java、
4 BeanDefinitionRegistry.java、 SimpleBeanFactory.java
5 factory.config — SingletonBeanRegistry.java、 ConstructorArgumentValues.java、
6 ConstructorArgumentValue.java、 BeanDefinition.java
7
8 // 注：
9 // ConstructorArgumentValues由ArgumentValues改名而来
10 // ConstructorArgumentValue由ArgumentValue改名而来
```

熟悉了这个项目结构后，你再回头去看 Spring 框架的结构，会发现它们是一样的，不光目录一样，文件名也是一样的，类中的主要方法名和属性名也是一样的。我这么做的目的是便于你之后自己继续学习。

注解支持

如果你用过 Spring 的话，对 Autowired 注解想必不陌生，这也是常用的依赖注入的方式，在需要注入的对象上增加 @Autowired 注解就可以了，你可以参考下面这个例子。

```
1 public class Test {  
2     @Autowired  
3     private TestAutowired testAutowired;  
4 }
```

这种方式的好处在于，不再需要显式地在 XML 配置文件中使用 ref 属性，指定需要依赖的对象，直接在代码中加上这个注解，就能起到同样的依赖注入效果。但是你要知道，计算机运行程序是机械式的，并没有魔法，加的这一行注解不会自我解释，必须有另一个程序去解释它，否则注解就变成了注释。

那么，问题就来了，**我们要在哪一段程序、哪个时机去解释这个注解呢？**

简单分析一下，这个注解是作用在一个实例变量上的，为了生效，我们首先必须创建好这个对象，也就是在 createBean 时机之后。

回顾前面几节课的内容，我们通过一个 refresh() 方法包装了整个 Bean 的创建过程，我们能看到在创建 Bean 实例之后，要进行初始化工作，refresh() 方法内预留了 postProcessBeforeInitialization、init-method 与 postProcessAfterInitialization 的位置，根据它们的名称也能看出是在初始化前、中、后分别对 Bean 进行处理。这里就是很好的时机。

接下来我们一起来看看这些功能是如何实现的。


在这个预留的位置，我们可以考虑调用一个 Bean 处理器 Processor，由处理器来解释注解。我们首先来定义 BeanPostProcessor，它内部的两个方法分别用于 Bean 初始化之前和之后。

1. Bean 初始化之前

```
1 public interface BeanPostProcessor {  
2     Object postProcessBeforeInitialization(Object bean, String beanName) throws  
3     BeansException;
```


```
4 }
```

2. Bean 初始化之后

 复制代码

```
1 public interface BeanPostProcessor {  
2     Object postProcessAfterInitialization(Object bean, String beanName) throws  
3     BeansException;  
4 }
```

接下来我们定义 Autowired 注解，很简单，你可以参考一下。

 复制代码

```
1 @Target(ElementType.FIELD)  
2 @Retention(RetentionPolicy.RUNTIME)  
3 public @interface Autowired {  
4 }
```

根据这个定义可以知道，Autowired 修饰成员变量（属性），并且在运行时生效。

为了实现 @Autowired 这个注解，我们很自然地会想到，利用反射获取所有标注了 Autowired 注解的成员变量，把它初始化成一个 Bean，然后注入属性。结合前面我们定义的 BeanPostProcessor 接口，我们来定义 Autowired 的处理类 AutowiredAnnotationBeanPostProcessor。

 复制代码

```
1 public class AutowiredAnnotationBeanPostProcessor implements BeanPostProcessor {  
2     private AutowireCapableBeanFactory beanFactory;  
3  
4     @Override  
5     public Object postProcessBeforeInitialization(Object bean, String beanName)  
6     throws BeansException {  
7         Object result = bean;  
8  
9         Class<?> clazz = bean.getClass();  
10        Field[] fields = clazz.getDeclaredFields();
```

```

11         if(fields!=null){
12             //对每一个属性进行判断, 如果带有@Autowired注解则进行处理
13             for(Field field : fields){
14                 boolean isAutowired =
15 field.isAnnotationPresent(Autowired.class);
16                 if(isAutowired){
17                     //根据属性名查找同名的bean
18                     String fieldName = field.getName();
19                     Object autowiredObj =
20 this.getBeanFactory().getBean(fieldName);
21                     //设置属性值, 完成注入
22                     try {
23                         field.setAccessible(true);
24                         field.set(bean, autowiredObj);
25                         System.out.println("autowire " + fieldName + " for bean
26 " + beanName);
27                     }
28                 }
29             }
30         }
31         return result;
32     }
33     @Override
34     public Object postProcessAfterInitialization(Object bean, String beanName)
35 throws BeansException {
36         return null;
37     }
38     public AutowireCapableBeanFactory getBeanFactory() {
39         return beanFactory;
40     }
41     public void setBeanFactory(AutowireCapableBeanFactory beanFactory) {
42         this.beanFactory = beanFactory;
43     }
44 }
45

```

其实, 核心代码就只有几行。

```

1 boolean isAutowired = field.isAnnotationPresent(Autowired.class);
2 if(isAutowired){
3     String fieldName = field.getName();
4     Object autowiredObj = this.getBeanFactory().getBean(fieldName);
5     field.setAccessible(true);
6     field.set(bean, autowiredObj);

```

判断类里面的每一个属性是不是带有 Autowired 注解，如果有，就根据属性名获取 Bean。从这里我们可以看出，属性名字很关键，我们就是靠它来获取和创建的 Bean。有了 Bean 之后，我们通过反射设置属性值，完成依赖注入。

新的 BeanFactory

在这里我们引入了 AutowireCapableBeanFactory，这个 BeanFactory 就是专为 Autowired 注入的 Bean 准备的。

在此之前我们已经定义了 BeanFactory 接口，以及一个 SimpleBeanFactory 的实现类。现在我们又需要引入另外一个 BeanFactory——**AutowireCapableBeanFactory**。基于代码复用、解耦的原则，我们可以对通用部分代码进行抽象，抽象出一个 AbstractBeanFactory 类。

目前，我们可以把 refresh()、getBean()、registerBeanDefinition() 等方法提取到抽象类，因为我们提供了默认实现，确保这些方法即使不再被其他 BeanFactory 实现也能正常生效。改动比较大，所以这里我贴出完整的类代码，下面就是 AbstractBeanFactory 的完整实现。

```

1 public abstract class AbstractBeanFactory extends DefaultSingletonBeanRegistry
2 implements BeanFactory, BeanDefinitionRegistry {
3     private Map<String, BeanDefinition> beanDefinitionMap = new
4 ConcurrentHashMap<>(256);
5     private List<String> beanDefinitionNames = new ArrayList<>();
6     private final Map<String, Object> earlySingletonObjects = new HashMap<>(16);
7     public AbstractBeanFactory() {
8     }
9     public void refresh() {
10         for (String beanName : beanDefinitionNames) {

```

```

11         try {
12             getBean(beanName);
13         }
14     }
15 }
16 @Override
17 public Object getBean(String beanName) throws BeansException {
18     //先尝试直接从容器中获取bean实例
19     Object singleton = this.getSingleton(beanName);
20     if (singleton == null) {
21         //如果没有实例，则尝试从毛胚实例中获取
22         singleton = this.earlySingletonObjects.get(beanName);
23         if (singleton == null) {
24             //如果连毛胚都没有，则创建bean实例并注册
25             System.out.println("get bean null ----- " + beanName);
26             BeanDefinition beanDefinition = beanDefinitionMap.get(beanName);
27             singleton = createBean(beanDefinition);
28             this.registerBean(beanName, singleton);
29             // 进行beanpostprocessor处理
30             // step 1: postProcessBeforeInitialization
31             applyBeanPostProcessorBeforeInitialization(singleton, beanName);
32             // step 2: init-method
33             if (beanDefinition.getInitMethodName() != null &&
34 !beanDefinition.equals("")) {
35                 invokeInitMethod(beanDefinition, singleton);
36             }
37             // step 3: postProcessAfterInitialization
38             applyBeanPostProcessorAfterInitialization(singleton, beanName);
39         }
40     }
41
42     return singleton;
43 }
44 private void invokeInitMethod(BeanDefinition beanDefinition, Object obj) {
45     Class<?> clz = beanDefinition.getClass();
46     Method method = null;
47     try {
48         method = clz.getMethod(beanDefinition.getInitMethodName());
49     }
50     try {
51         method.invoke(obj);
52     }
53 }
54 @Override
55 public Boolean containsBean(String name) {
56     return containsSingleton(name);
57 }
58 public void registerBean(String beanName, Object obj) {
59     this.registerSingleton(beanName, obj);

```

```

60     }
61     @Override
62     public void registerBeanDefinition(String name, BeanDefinition
63 beanDefinition) {
64         this.beanDefinitionMap.put(name, beanDefinition);
65         this.beanDefinitionNames.add(name);
66         if (!beanDefinition.isLazyInit()) {
67             try {
68                 getBean(name);
69             }
70         }
71     }
72     @Override
73     public void removeBeanDefinition(String name) {
74         this.beanDefinitionMap.remove(name);
75         this.beanDefinitionNames.remove(name);
76         this.removeSingleton(name);
77     }
78     @Override
79     public BeanDefinition getBeanDefinition(String name) {
80         return this.beanDefinitionMap.get(name);
81     }
82     @Override
83     public boolean containsBeanDefinition(String name) {
84         return this.beanDefinitionMap.containsKey(name);
85     }
86     @Override
87     public boolean isSingleton(String name) {
88         return this.beanDefinitionMap.get(name).isSingleton();
89     }
90     @Override
91     public boolean isPrototype(String name) {
92         return this.beanDefinitionMap.get(name).isPrototype();
93     }
94     @Override
95     public Class<?> getType(String name) {
96         return this.beanDefinitionMap.get(name).getClass();
97     }
98     private Object createBean(BeanDefinition beanDefinition) {
99         Class<?> clz = null;
100         //创建毛胚bean实例
101         Object obj = doCreateBean(beanDefinition);
102         //存放到毛胚实例缓存中
103         this.earlySingletonObjects.put(beanDefinition.getId(), obj);
104         try {
105             clz = Class.forName(beanDefinition.getClassName());
106         }
107         //完善bean, 主要是处理属性
108         populateBean(beanDefinition, clz, obj);

```



```

109         return obj;
110     }
111     //doCreateBean创建毛胚实例, 仅仅调用构造方法, 没有进行属性处理
112     private Object doCreateBean(BeanDefinition beanDefinition) {
113         Class<?> clz = null;
114         Object obj = null;
115         Constructor<?> con = null;
116         try {
117             clz = Class.forName(beanDefinition.getClassName());
118             // handle constructor
119             ConstructorArgumentValues constructorArgumentValues =
120 beanDefinition.getConstructorArgumentValues();
121             if (!constructorArgumentValues.isEmpty()) {
122                 Class<?>[] paramTypes = new Class<?>
123 [constructorArgumentValues.getArgumentCount()];
124                 Object[] paramValues = new
125 Object[constructorArgumentValues.getArgumentCount()];
126                 for (int i = 0; i <
127 constructorArgumentValues.getArgumentCount(); i++) {
128                     ConstructorArgumentValue constructorArgumentValue =
129 constructorArgumentValues.getIndexedArgumentValue(i);
130                     if ("String".equals(constructorArgumentValue.getType()) ||
131 "java.lang.String".equals(constructorArgumentValue.getType())) {
132                         paramTypes[i] = String.class;
133                         paramValues[i] = constructorArgumentValue.getValue();
134                     } else if
135 ("Integer".equals(constructorArgumentValue.getType()) ||
136 "java.lang.Integer".equals(constructorArgumentValue.getType())) {
137                         paramTypes[i] = Integer.class;
138                         paramValues[i] = Integer.valueOf((String)
139 constructorArgumentValue.getValue());
140                     } else if ("int".equals(constructorArgumentValue.getType()))
141 {
142                         paramTypes[i] = int.class;
143                         paramValues[i] = Integer.valueOf((String)
144 constructorArgumentValue.getValue());
145                     } else {
146                         paramTypes[i] = String.class;
147                         paramValues[i] = constructorArgumentValue.getValue();
148                     }
149                 }
150                 try {
151                     con = clz.getConstructor(paramTypes);
152                     obj = con.newInstance(paramValues);
153                 }
154             }
155         }
156         System.out.println(beanDefinition.getId() + " bean created. " +
157 beanDefinition.getClassName() + " : " + obj.toString());

```

```

158         return obj;
159     }
160     private void populateBean(BeanDefinition beanDefinition, Class<?> clz,
161 Object obj) {
162         handleProperties(beanDefinition, clz, obj);
163     }
164     private void handleProperties(BeanDefinition beanDefinition, Class<?> clz,
165 Object obj) {
166         // handle properties
167         System.out.println("handle properties for bean : " +
168 beanDefinition.getId());
169         PropertyValues propertyValues = beanDefinition.getPropertyValues();
170         //如果有属性
171         if (!propertyValues.isEmpty()) {
172             for (int i = 0; i < propertyValues.size(); i++) {
173                 PropertyValue propertyValue =
174 propertyValues.getPropertyValueList().get(i);
175                 String pType = propertyValue.getType();
176                 String pName = propertyValue.getName();
177                 Object pValue = propertyValue.getValue();
178                 boolean isRef = propertyValue.getIsRef();
179                 Class<?>[] paramTypes = new Class<?>[1];
180                 Object[] paramValues = new Object[1];
181                 if (!isRef) { //如果不是ref, 只是普通属性
182                     //对每一个属性, 分数据类型分别处理
183                     if ("String".equals(pType) ||
184 "java.lang.String".equals(pType)) {
185                         paramTypes[0] = String.class;
186                     } else if ("Integer".equals(pType) ||
187 "java.lang.Integer".equals(pType)) {
188                         paramTypes[i] = Integer.class;
189                     } else if ("int".equals(pType)) {
190                         paramTypes[i] = int.class;
191                     } else {
192                         paramTypes[i] = String.class;
193                     }
194                     paramValues[0] = pValue;
195                 } else { //is ref, create the dependent beans
196                     try {
197                         paramTypes[0] = Class.forName(pType);
198                     }
199                     try { //再次调用getBean创建ref的bean实例
200                         paramValues[0] = getBean((String) pValue);
201                     }
202                 }
203                 //按照setXxxx规范查找setter方法, 调用setter方法设置属性
204                 String methodName = "set" + pName.substring(0, 1).toUpperCase()
205 + pName.substring(1);
206                 Method method = null;


```

```

207         try {
208             method = clz.getMethod(methodName, paramTypes);
209         }
210         try {
211             method.invoke(obj, paramValues);
212         }
213     }
214 }
215 }
216 abstract public Object applyBeanPostProcessorBeforeInitialization(Object
217 existingBean, String beanName) throws BeansException;
218 abstract public Object applyBeanPostProcessorAfterInitialization(Object
219 existingBean, String beanName) throws BeansException;
220 }

```

上面的代码较长，但仔细一看可以发现绝大多数是我们原本已经实现的方法，只是移动到了 `AbstractBeanFactory` 这个抽象类之中。最关键的代码是 `getBean()` 中的这一段。

 复制代码

```


1 BeanDefinition beanDefinition = beanDefinitionMap.get(beanName);
2 singleton = createBean(beanDefinition);
3 this.registerBean(beanName, singleton);
4
5 // beanpostprocessor
6 // step 1: postProcessBeforeInitialization
7 applyBeanPostProcessorBeforeInitialization(singleton, beanName);
8 // step 2: init-method
9 if (beanDefinition.getInitMethodName() != null &&
10 !beanDefinition.equals("")) {
11     invokeInitMethod(beanDefinition, singleton);
12 }
13 // step 3: postProcessAfterInitialization
14 applyBeanPostProcessorAfterInitialization(singleton, beanName);

```

先获取 Bean 的定义，然后创建 Bean 实例，再进行 Bean 的后处理并初始化。在这个抽象类里，我们需要关注两个核心的改动。


1. 定义了抽象方法 `applyBeanPostProcessorBeforeInitialization` 与 `applyBeanPostProcessorAfterInitialization`，由名字可以看出，分别是在 Bean 处理类初始化之前和之后执行的方法。这两个方法交给具体的继承类去实现。

2. 在 `getBean()` 方法中，在以前预留的位置，实现了对 Bean 初始化前、初始化和初始化后的处理。

 复制代码

```
1 // step 1: postProcessBeforeInitialization
2 applyBeanPostProcessorBeforeInitialization(singleton, beanName);
3 // step 2: init-method
4 if (beanDefinition.getInitMethodName() != null && !beanDefinition.equals("")) {
5     invokeInitMethod(beanDefinition, singleton);
6 }
7 // step 3: postProcessAfterInitialization
8 applyBeanPostProcessorAfterInitialization(singleton, beanName);
```

现在已经抽象出了一个 `AbstractBeanFactory`，接下来我们看看具体的 `AutowireCapableBeanFactory` 是如何实现的。

 复制代码

```
1 public class AutowireCapableBeanFactory extends AbstractBeanFactory{
2     private final List<AutowiredAnnotationBeanPostProcessor> beanPostProcessors =
3     new ArrayList<>();
4     public void addBeanPostProcessor(AutowiredAnnotationBeanPostProcessor
5 beanPostProcessor) {
6         this.beanPostProcessors.remove(beanPostProcessor);
7         this.beanPostProcessors.add(beanPostProcessor);
8     }
9     public int getBeanPostProcessorCount() {
10         return this.beanPostProcessors.size();
11     }
12     public List<AutowiredAnnotationBeanPostProcessor> getBeanPostProcessors() {
13         return this.beanPostProcessors;
14     }
15     public Object applyBeanPostProcessorsBeforeInitialization(Object
16 existingBean, String beanName) throws BeansException {
17         Object result = existingBean;
18         for (AutowiredAnnotationBeanPostProcessor beanProcessor :
19 getBeanPostProcessors()) {
20             beanProcessor.setBeanFactory(this);
21             result = beanProcessor.postProcessBeforeInitialization(result,
22 beanName);
23             if (result == null) {
24                 return result;
25             }
26         }
```

```

27         return result;
28     }
29     public Object applyBeanPostProcessorsAfterInitialization(Object existingBean,
30 String beanName) throws BeansException {
31         Object result = existingBean;
32         for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
33             result = beanProcessor.postProcessAfterInitialization(result,
34 beanName);
35             if (result == null) {
36                 return result;
37             }
38         }
39         return result;
40     }
41 }

```

从代码里也可以看出，它实现起来并不复杂，用一个列表 beanPostProcessors 记录所有的 Bean 处理器，这样可以按照需求注册若干个不同用途的处理器，然后调用处理器。

 复制代码


```

1 for (AutowiredAnnotationBeanPostProcessor beanProcessor :
2 getBeanPostProcessors()) {
3     beanProcessor.setBeanFactory(this);
4     result = beanProcessor.postProcessBeforeInitialization(result,
5 beanName);
6 }

```

代码一目了然，就是对每个 Bean 处理器，调用方法 postProcessBeforeInitialization。

最后则是调整 ClassPathXmlApplicationContext，引入的成员变量由 SimpleBeanFactory 改为新建的 AutowireCapableBeanFactory，并在构造函数里增加上下文刷新逻辑。

 复制代码

```

1 public ClassPathXmlApplicationContext(String fileName, boolean isRefresh) {
2     Resource resource = new ClassPathXmlResource(fileName);
3     AutowireCapableBeanFactory beanFactory = new
4 AutowireCapableBeanFactory();
5     XmlBeanDefinitionReader reader = new
6 XmlBeanDefinitionReader(beanFactory);
7     reader.loadBeanDefinitions(resource);

```

```

8         this.beanFactory = beanFactory;
9         if (isRefresh) {
10             try {
11                 refresh();
12             }
13         }
14     }
15
16     public List<BeanFactoryPostProcessor> getBeanFactoryPostProcessors() {
17         return this.beanFactoryPostProcessors;
18     }
19     public void addBeanFactoryPostProcessor(BeanFactoryPostProcessor
20 postProcessor) {
21         this.beanFactoryPostProcessors.add(postProcessor);
22     }
23     public void refresh() throws BeansException, IllegalStateException {
24         // Register bean processors that intercept bean creation.
25         registerBeanPostProcessors(this.beanFactory);
26         // Initialize other special beans in specific context subclasses.
27         onRefresh();
28     }
29     private void registerBeanPostProcessors(AutowireCapableBeanFactory
30 beanFactory) {
31         beanFactory.addBeanPostProcessor(new
32 AutowiredAnnotationBeanPostProcessor());
33     }
34     private void onRefresh() {
35         this.beanFactory.refresh();
36     }


```

新的 refresh() 方法，会先注册 BeanPostProcessor，这样 BeanFactory 里就有解释注解的处理器了，然后在 getBean() 的过程中使用它。

最后，我们来回顾一下完整的过程。


1. 启动 ClassPathXmlApplicationContext 容器，执行 refresh()。
2. 在 refresh 执行过程中，调用 registerBeanPostProcessors()，往 BeanFactory 里注册 Bean 处理器，如 AutowiredAnnotationBeanPostProcessor。
3. 执行 onRefresh()，执行 AbstractBeanFactory 的 refresh() 方法。

4. AbstractBeanFactory 的 refresh() 获取所有 Bean 的定义，执行 getBean() 创建 Bean 实例。
5. getBean() 创建完 Bean 实例后，调用 Bean 处理器并初始化。

 复制代码

```
1 applyBeanPostProcessorBeforeInitialization(singleton, beanName);
2 invokeInitMethod(beanDefinition, singleton);
3 applyBeanPostProcessorAfterInitialization(singleton, beanName);
```

6. applyBeanPostProcessorBeforeInitialization 由具体的 BeanFactory，如 AutowireCapableBeanFactory，来实现，这个实现也很简单，就是对 BeanFactory 里已经注册好的所有 Bean 处理器调用相关方法。

 复制代码

```
1 beanProcessor.postProcessBeforeInitialization(result, beanName);
2 beanProcessor.postProcessAfterInitialization(result, beanName);
```

7. 我们事先准备好的 AutowiredAnnotationBeanPostProcessor 方法里面会解释 Bean 中的 Autowired 注解。

测试注解

到这里，支持注解的工作就完成了，接下来就是测试 Autowired 注解了。在这里我们做两个改动。

1. 在测试类中增加 Autowired 注解。

 复制代码

```
1 package com.minis.test;
2 import com.minis.beans.factory.annotation.Autowired;
3 public class BaseService {
4     @Autowired
5     private BaseService bbs;
6     public BaseService getBbs() {
```

```

7         return bbs;
8     }
9     public void setBbs(BaseBaseService bbs) {
10         this.bbs = bbs;
11     }
12     public BaseService() {
13     }
14     public void sayHello() {
15         System.out.println("Base Service says Hello");
16         bbs.sayHello();
17     }
18 }

```

2. 注释 XML 配置文件中关于循环依赖的配置。

 复制代码

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <beans>
3     <bean id="bbs" class="com.minis.test.BaseBaseService">
4         <property type="com.minis.test.AServiceImpl" name="as" ref="aservice" />
5     </bean>
6     <bean id="aservice" class="com.minis.test.AServiceImpl">
7         <constructor-arg type="String" name="name" value="abc"/>
8         <constructor-arg type="int" name="level" value="3"/>
9         <property type="String" name="property1" value="Someone says"/>
10        <property type="String" name="property2" value="Hello World!"/>
11        <property type="com.minis.test.BaseService" name="ref1"
12        ref="baseservice"/>
13    </bean>
14    <bean id="baseservice" class="com.minis.test.BaseService">
15    <!--        <property type="com.minis.test.BaseBaseService" name="bbs"
16    ref="basebaseservice" />-->
17    </bean>
18 </beans>

```

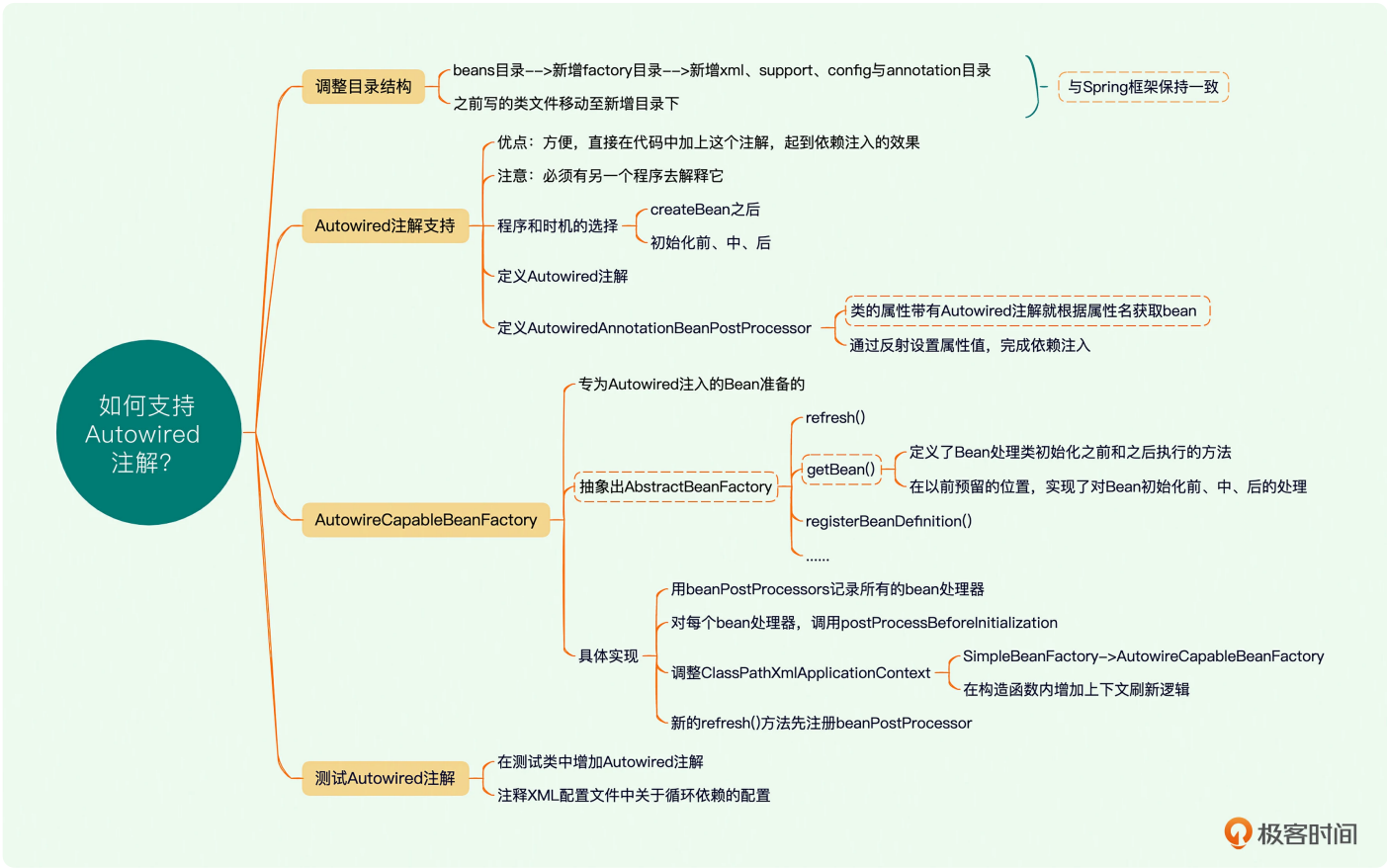
小结

这节课我们丰富了原来的框架，支持了注解，让它更有模有样了。

注解是现代最受程序员欢迎的特性，我们通过 Autowired 这个注解实现了 Bean 的注入，这样程序员不用再在 XML 配置文件中手动配置 property，而是在类中声明 property 的时候直

接加上注解即可，框架使用的机制是名称匹配，这也是 Spring 所支持的一种匹配方式。

接着我们提取了 BeanFactory 接口，定义了一个抽象的 AbstractBeanFactory。通过这个抽象类，将 Bean 工厂需要做的事情的框架搭建出来，然后在具体实现类中完善细节。这种程序结构称为 interface-abstract class-class（接口抽象类），是一种做框架时常用的设计模式。



我们自己手写 MiniSpring，不仅仅是要学习一个功能如何实现，还要学习大师的做法，模仿他们的代码和设计，练习得多了就能像专业程序员一样地写代码了。

完整源代码参见 <https://github.com/YaleGuo/minispring>

课后题

学完这节课，我也给你留一道思考题。我们实现了 Autowired 注解，在现有框架中能否支持多个注解？欢迎你在留言区与我交流讨论，也欢迎你把这节课分享给需要的朋友。我们下节课见！

精选留言 (24)



睿智的仓鼠

2023-03-21 来自湖北

通过这节课真的是感受到Spring设计的巧妙之处了，我目前的理解是，解耦分为两种：设计上的解耦、实现类上的解耦。通过抽取AbstractBeanFactory，把BeanPostProcessor的设计与BeanFactory本身解耦，AutowiredCapableBeanFactory再通过定义BeanPostProcessor接口类型的属性，向外提供属性设置的方法，做到了和BeanPostProcessor实现类的解耦，最后在ClassPathXmlApplicationContext中统一注册BeanPostProcessor，再抽取成一个启动方法，非常优雅。AbstractBeanFactory的“接口抽象类”思想也很巧妙。这些思想在学Spring源码时早已听说，当时只觉得这样设计是灵活的，但不知道具体灵活在何处，通过自己手写实现下来真的是越来越清晰了，实属好课！

作者回复：赞！



👍 2



陈小远

2023-03-20 来自四川

跟到第四节了，单纯看文章逻辑来说还能理解一些实现，但是结合github上的源码（对照的是ioc4分支，不知道是否找对），发现源码和文章逻辑叙述的时候同一个类贴的代码实现是不一样的，导致对照学习的时候产生混乱。比如源码中AutowiredCapableBeanFactory是直接实现的BeanFactory，但在文章的表述中是继承的AbstractBeanFactory，由此就在此节无法对照源码和文章表述自我参照着来完成注解的功能。看了四节后说说自己的一些看法或观点：

- 1、源码在github上，因为不是整个分支全克隆下来学习，单独的点击查看某个类的代码比较慢，影响学习效率，如果码云上有学习地址可能会好很多；
- 2、文章代码和源码不一致的问题不知道是跳跃太大还是个人没找对位置，很多人可能会渐渐的迷乱从而无法继续跟进学习；
- 3、源码相对于文章来说跳跃性比较大，如果正文中没法完全交代清楚，建议在源码的readme文件尽可能详细的给出一些突然出现的类的说明和设计意图

总的来说，通过老师的引导，再结合Spring的源码，还是有那么点感觉的，不过今天这节课实在没跟下来，可能还需要多花点时间自己琢磨琢磨

作者回复: Github上的分支跟讲稿差不多对应的, 但是不是完全一一对应上的, 这是历史原因, 几年前就放到Github上了, 但是讲稿为了篇幅的规定又略微有些调整。你看的这部分是在ioc4中。每个分支里有一个docs/readme.txt文件, 是当时边写的时候边手工做的记录。

你很用心, 希望能跟下来, 对自己大有益处的。

共 3 条评论 >

👍 2



杨松

2023-04-14 来自辽宁

老师, 请教下关于AutowiredAnnotationBeanPostProcessor的时机问题, 为什么不是在处理属性handleProperties的位置, 这个方法正好是设置bean实例的属性啊, 我一直没弄懂为啥不放在这里?

作者回复: 这个不是功能实现的问题, 是时序规定的事情。第一步处理是根据配置的构造函数进行实例化, 第二步是根据配置的properties给属性赋值, 然后才到了后期修饰, 而Spring统一用了一个beanpostprocessor机制来处理。三步一遍遍做完的。学习框架, 功能实现不是最主要的, 而是结构和时序, 光说功能, 不要框架也一样的。

共 3 条评论 >

👍 1



马儿

2023-03-22 来自四川

ClassPathXmlApplicationContext增加了一个BeanFactoryPostProcessor属性, 本文中没有给出定义, 看了GitHub源码把这个类拷贝出来。这个类的作用是什么呢? 是像对BeanPostProcessor一样对BeanFactory进行特殊处理的吗?

作者回复: 对的对的。这里没有用上, 只是写了一个占位在这里, 说明从时序上这儿可以进行对beanfactory的后期处理。这是当时留给学生的扩展练习。



👍 2



怕什么, 抱紧我

2023-05-26 来自湖南

老师:

1. Autowire注入对象B, 此时B的beanDefinnition还没有加载进来,会报错! 设置lazy = ture 也不行, 只能把if(!beanDefinition.isLazyInit())注释掉, 还有其它方法吗!

2.AutowiredAnnotationBeanPostProcessor#postProcessBeforeInitialization()方法:

```
String fieldName = field.getName();
```

```
Object autowiredObj = this.getBeanFactory().getBean(fieldName);
```

这段代码，类的属性字段名要和xml配置的beanId相同，否则找不到！

作者回复: 1.你看别的问题回复，这个问题讨论过了。

2.是的。因为这是简版，主要为了说明原理。

共 2 条评论 >



Michael



2023-04-17 来自陕西

想请教一下老师，那现在是不是SimpleBeanFactory完全没用了？

作者回复: 对。演变过程中会丢掉一些东西。



杨松



2023-04-13 来自辽宁

老师请教下，包factory.support和包factory.config划分上有什么依据，不太明白为什么这么划分

作者回复: 原则都是一样的，物以类聚，紧密的一堆类放一起。但是具体划分又因人而异。我是按照Spring框架来的。MiniSpring最后的结构跟Spring一样，这样继续学习Spring就自然而然了。



啊良梓是我



2023-04-03 来自广东

...

```
@Override public void registerBeanDefinition(String name, BeanDefinition beanDefinition) { this.beanDefinitionMap.put(name, beanDefinition); this.beanDefinitionNames.add(name); if (!beanDefinition.isLazyInit()) { try { getBean(name); } } }
```

...

这里我有个问题，就是这个如果在设置BeanDefinition的时候，执行getBean方法，但是如果Bean里面注入了第三方Bean，这时候，有可能这个第三方Bean的BeanDefinition还没有实例化

的，很明显不能循环注入Bean的啊？这个应该怎么处理的呢？延迟等待？或者说主动去遍历Resource接口获取资源，当也拿不到的时候，直接报编译器异常？

作者回复: 说得太对了，用心了。这个代码有你说的这个问题。原因是minispring是为学习spring框架的原理而构建的，主要体现内部结构。这个地方是线下课时给学生扩展练习用的。这个地方，为鲁棒性，要先把全部definition加载完，再getbean。

共 2 条评论 >



追梦

2023-03-31 来自广东

老师好，您说在beans.xml配置的bean别名和被@Autowired修饰的属性名必须一样，但是我觉得这样不够完整，我使用了@Autowired自然是希望所有该类型都被注入无论其属性名是什么，我想记录一个属性值映射singleton的hashMap对象来存储，老师觉得这个思路怎么样？或者老师有什么其他看法嘛对于文件中bean的别名和属性名必须相同

作者回复: 我这么处理只是为了简单。Spring框架本身还可以按照Type来注入。



努力呼吸

2023-03-29 来自广东

老师，为什么beanDefinitionMap是线程安全的容器；beanDefinitionNames和earlySingletonObject不需要呢？

作者回复: beanDefinitionNames应该加上。earlySingletonObject是纯粹内部使用的，使用的时候都是放在synchronized代码块中的，你可以看一下Spring框架本身的源代码。



梦某人

2023-03-29 来自日本

个人实现地址如下: <https://github.com/DDream/myMiniSpring>

欢迎其他同学参考以及 Star，根据 Commit history 查看我根据课程内容分割的实现过程。

提问1: applyBeanPostProcessorAfterInitialization 方法在 AbstractFactory 中存在于返回值，有什么存在的必要性，但是，obj 作为一个引用对象，需要返回吗？在 AbstractFactory 中也并没有使用到其返回值。（另外方法如果直接为 void 依然有效，我已经测试过。）

提问2: AbstractFactory 这个似乎过于单例了？可能是因为 mini 实现的原因？因为目前的 Ab

stractFactory 基本等于修改了 SimpleSingletonFactory。

提问3: 关于 Factory 的 Refres 问题, 当我在 ClassPathXmlApplicationContext 的 refresh() 方法中, 在 注册前后处理器之前进行了 Refresh, 就会导致 Autowired 无法成功注入, 这是因为实例已经建立好了吗? 那如果想修复这个问题, 老师能否提供一定的思路? 比如, 在 getBean 的时候做一遍属性检查吗? (当然有可能是我个人实现的问题)

由于没看源码, 是按照老师给出的代码和讲解实现的, 所以可能存在一定理解偏差, 希望老师指正一下。

关于课后题:

可以实现, 这部分在 Processor 部分进行了解耦合, 只要实现相关注解的 Processor, 并在 Context 中进行加载, 即可。

作者回复: 1, 这里是不可以不要返回值的。但是到了FactoryBean的实现的时候, 需要。

2, AbstractFactory主要的目的是抽取一个模板, 让别的beanFactory来继承。

3, 这个refresh是留下的一个口子, 用来扩展练习的。要解决你说的问题, 我觉得是要先将beandefinition完全加载完之后再 getbean。

另外, 文稿的代码时不全的, 只是主要的部分。完整代码, 还是要看Github上的, 有不同分支。

共 2 条评论 >



一念之间

2023-03-26 来自上海

有一个简单的问题 为什么处理器要叫做PostProcessor呢? 这里的post到底是对于什么动作而言的呢?

作者回复: post的字面含义是“什么什么之后”, 这里是指create bean之后, 用postProcessor进行修饰。



三太子

2023-03-25 来自重庆

<https://github.com/yx-Yaoxaing/minispring/wiki/%E5%85%B3%E4%BA%8Emini-spring>
自己写的代码提到github上 每日打卡! 遇到的基础问题 都写在了wiki

作者回复: 赞



1

**Geek_320730**

2023-03-22 来自北京

1.invokeInitMethod(BeanDefinition bd, Object obj) 中 master分支是用的obj.getObject(),ioc 4分支中是bd.getObject(),应该是obj.getObject()
2.循环调用postProcessBeforeInitialization或postProcessAfterInitialization那一块,好像是返回的bean为null的时候,中断这个循环调用,返回上次调用结果?是不是应该在调用之前先用另一个变量暂存一下,返回值为null的时候,return变量?按代码逻辑会return null,这样的话,后续应该会空指针异常吧。。。

作者回复: 你看得仔细, 赞!

1, 我再检查一下啊, 看看哪里出错了, 怎么不一致了。

2, 你说的对。代码是原理性的, 能正常运行, 不是工业级的。

**聪聪不匆匆**

2023-03-22 来自上海

第四节github上面ioc4 branch分支代码 和 文档中相差还是较大的。比如DefaultListableBeanFactory 类文档中并没有提到。再比如ClassPathXmlApplicationContext类中构造函数使用的是AutowireCapableBeanFactory类, 但是在github中确是DefaultListableBeanFactory类, 这个类怎么来的、什么用处、怎么演进的并没有地方提现到。诸如此类, github中在本次分支中多出来了很多类都在文档中未曾提到。

作者回复: 感谢指出问题, 这是历史原因造成的。后面我来调整。

**风轻扬**

2023-03-22 来自北京

刚学了这么几节课, 感觉就收获很大, 代码手敲一遍, 体会更深。有一个体会是: Spring的代码都是低耦合的, 感觉低耦合的优势就是复用性强, 扩展性强。扩展性方面, 业务逻辑千变万化, Spring不能想到所有的业务情况, 所以它就搞抽象, 定标准。业务方可以根据自己的实际情况来实现标准, 从而完成自己的个性化需求。这个思想很重要, 工作中要注意用起来之前也经常看Spring源码, 但因为Spring源码太庞大了, 只能先基于一个点看, 这就导致无法从整体上了解Spring的运行机制。这门课正好就是从整体理解Spring。把之前学习的点串起来, 不错。好课一枚, 赞!

作者回复: 感谢。希望你学下来收获大, 这也是我带学生的初衷。



风轻扬

2023-03-22 来自北京

老师, 有几个问题请教一下。

- 1、AutowireCapableBeanFactory类中的addBeanPostProcessor为什么要先remove再add呢?
- 2、代码里出现异常的时候, 老师都是只写一个try, 这貌似不行吧, 我是java8, 这是高版本jdk的新特性吗?
- 3、AutowireCapableBeanFactory类中的invokeInitMethod方法, 您的代码逻辑中, 获取Class对象用的是BeanDefinition.getClass(), 我理解, 应该用Class.forName(BeanDefinition.getClassName()), init方法应该是我们业务类上定义的init方法吧?

作者回复: 1, 是不想重复

2, 文稿中的代码不全, 编辑当时建议只列关键代码。你要上Github找不同的分支, 下载完整代码。

3, 我用的是obj.getClass()。init方法是业务类定义的初始化方法, 名字在XML文件中配置。

共 2 条评论 >



Unknown

2023-03-22 来自福建

与spring的autowired byType注入不同, 老师实现的autowired是根据beanName注入, 所以AbstractBeanFactory里面的两个缓存Map key为小写类名。

测试用例那边是不是有点问题 不应该直接用bbs 应该用basebaseservice 这样才能创建的到吧

作者回复: Spring支持byType和byName, MiniSpring写的时候选择了一种作为示例, byType是作为学生扩展练习的。测试用例你自己试一下, 这些配置可能讲课过程中改去改来过, 你试一下能运行就行。



C.

2023-03-21 来自江苏

我有个疑问, 就是为什么BeanDefinition的lazyInit都设置为true, 默认懒加载, Spring中不都是默认false吗?

作者回复: 你的思考很好, 动脑筋了。这是当时写了一个架子, 让学生知道可以支持这样的属性, 当时留给学生当扩展作业的。主要是弄清楚内部结构知晓原理, 莫纠结于某一个点。



C.

2023-03-21 来自江苏

BeanDefinition这个类的懒加载lazyInit从ioc3版本开始变成了true,之前的false会导致导致AbstractBeanFactory这个类里

```
public void registerBeanDefinition(String name, BeanDefinition beanDefinition) {  
    this.beanDefinitionMap.put(name, beanDefinition);  
    this.beanDefinitionNames.add(name);  
    if (!beanDefinition.isLazyInit()) {  
        try {  
            getBean(name);  
        } catch (BeansException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

getBean执行, 导致beanDefinitionMap中bean的定义信息没加载完, 导致的找不到bean的定义信息导致的No bean.

作者回复: 是的是的, 你发现了, 赞! 这是当时写了一个架子, 留给学生当扩展作业的。

共 2 条评论 >

