

## 05 | 实现完整的IoC容器：构建工厂体系并添加容器事件

2023-03-22 郭屹 来自北京

《手把手带你写一个MiniSpring》



你好，我是郭屹。

前面我们已经实现了 IoC 的核心部分，骨架已经有了，那怎么让这个 IoC 丰满起来呢？这就需要实现更多的功能，让我们的 IoC 更加完备。所以这节课我们将通过建立 BeanFactory 体系，添加容器事件等一系列操作，进一步完善 IoC 的功能。

### 实现一个完整的 IoC 容器

为了让我们的 MiniSpring 更加专业一点，也更像 Spring 一点，我们将实现 3 个功能点。

1. 进一步增强扩展性，新增 4 个接口。

ListableBeanFactory

ConfigurableBeanFactory

ConfigurableListableBeanFactory

EnvironmentCapable


2. 实现 DefaultListableBeanFactory，该类就是 Spring IoC 的引擎。
3. 改造 ApplicationContext。

下面我们就一条条来看。

## 增强扩展性


首先我们来增强 BeanFactory 的扩展性，使它具有不同的特性。

我们以前定义的 AutowireCapableBeanFactory 就是在通用的 BeanFactory 的基础上添加了 Autowired 注解特性。比如可以将 Factory 内部管理的 Bean 作为一个集合来对待，获取 Bean 的数量，得到所有 Bean 的名字，按照某个类型获取 Bean 列表等等。这个特性就定义在 ListableBeanFactory 中。

 复制代码

```
1 public interface ListableBeanFactory extends BeanFactory {
2     boolean containsBeanDefinition(String beanName);
3     int getBeanDefinitionCount();
4     String[] getBeanDefinitionNames();
5     String[] getBeanNamesForType(Class<?> type);
6     <T> Map<String, T> getBeansOfType(Class<T> type) throws BeansException;
7 }
```


我们还可以将维护 Bean 之间的依赖关系以及支持 Bean 处理器也看作一个独立的特性，这个特性定义在 ConfigurableBeanFactory 接口中。

 复制代码

```
1 public interface ConfigurableBeanFactory extends
2     BeanFactory, SingletonBeanRegistry {
3     String SCOPE_SINGLETON = "singleton";
4     String SCOPE_PROTOTYPE = "prototype";
5     void addBeanPostProcessor(BeanPostProcessor beanPostProcessor);
```

```
6     int getBeanPostProcessorCount();
7     void registerDependentBean(String beanName, String dependentBeanName);
8     String[] getDependentBeans(String beanName);
9     String[] getDependenciesForBean(String beanName);
10 }
```

然后还可以集成，用一个 ConfigurableListableBeanFactory 接口把 AutowireCapableBeanFactory、ListableBeanFactory 和 ConfigurableBeanFactory 合并在一起。


 复制代码

```
1 package com.minis.beans.factory.config;
2 import com.minis.beans.factory.ListableBeanFactory;
3 public interface ConfigurableListableBeanFactory
4     extends ListableBeanFactory, AutowireCapableBeanFactory,
5     ConfigurableBeanFactory {
6 }
```

由上述接口定义的方法可以看出，这些接口都给通用的 BeanFactory 与 BeanDefinition 新增了众多处理方法，用来增强各种特性。

在 Java 语言的设计中，一个 Interface 代表的是一种特性或者能力，我们把这些特性或者能力一个个抽取出来，各自独立互不干扰。如果一个具体的类，想具备某些特性或者能力，就去实现这些 interface，随意组合。这是一种良好的设计原则，叫 **interface segregation**（接口隔离原则）。这条原则在 Spring 框架中用得很多，你可以注意一下。


由于 ConfigurableListableBeanFactory 继承了 AutowireCapableBeanFactory，所以我们需要调整之前定义的 AutowireCapableBeanFactory，由 class 改为 interface。

 复制代码

```
1 public interface AutowireCapableBeanFactory extends BeanFactory{
2     int AUTOWIRE_NO = 0;
3     int AUTOWIRE_BY_NAME = 1;
4     int AUTOWIRE_BY_TYPE = 2;
5     Object applyBeanPostProcessorsBeforeInitialization(Object existingBean,
6 String beanName) throws BeansException;
7     Object applyBeanPostProcessorsAfterInitialization(Object existingBean,
```

```
8 String beanName) throws BeansException;
9 }
```

新增抽象类 AbstractAutowireCapableBeanFactory 替代原有的实现类。

 复制代码

```
1 public abstract class AbstractAutowireCapableBeanFactory
2     extends AbstractBeanFactory implements
3     AutowireCapableBeanFactory{
4     private final List<BeanPostProcessor> beanPostProcessors = new
5     ArrayList<BeanPostProcessor>();
6
7     public void addBeanPostProcessor(BeenPostProcessor beanPostProcessor) {
8         this.beanPostProcessors.remove(beanPostProcessor);
9         this.beanPostProcessors.add(beanPostProcessor);
10    }
11    public int getBeanPostProcessorCount() {
12        return this.beanPostProcessors.size();
13    }
14    public List<BeanPostProcessor> getBeanPostProcessors() {
15        return this.beanPostProcessors;
16    }
17    public Object applyBeanPostProcessorsBeforeInitialization(Object
18    existingBean, String beanName)
19        throws BeansException {
20        Object result = existingBean;
21        for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
22            beanProcessor.setBeanFactory(this);
23            result = beanProcessor.postProcessBeforeInitialization(result,
24    beanName);
25            if (result == null) {
26                return result;
27            }
28        }
29        return result;
30    }
31    public Object applyBeanPostProcessorsAfterInitialization(Object
32    existingBean, String beanName)
33        throws BeansException {
34        Object result = existingBean;
35        for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
36            result = beanProcessor.postProcessAfterInitialization(result,
37    beanName);
38            if (result == null) {
39                return result;
40            }
41        }
42    }
```

```
41     }
42     return result;
43 }
44 }
```

上述代码与之前的实现类一致，在此不多赘述。

## 环境

除了扩充 BeanFactory 体系，我们还打算给容器增加一些环境因素，使一些容器整体所需要的属性有个地方存储访问。

在 core 目录下新建 env 目录，增加 PropertyResolver.java、EnvironmentCapable.java、Environment.java 三个接口类。EnvironmentCapable 主要用于获取 Environment 实例，Environment 则继承 PropertyResolver 接口，用于获取属性。所有的 ApplicationContext 都实现了 Environment 接口。

### Environment.java 接口

 复制代码

```
1 public interface Environment extends PropertyResolver {
2     String[] getActiveProfiles();
3     String[] getDefaultProfiles();
4     boolean acceptsProfiles(String... profiles);
5 }
```

### EnvironmentCapable.java 接口

 复制代码

```
1 public interface EnvironmentCapable {
2     Environment getEnvironment();
3 }
```

### PropertyResolver.java 接口

```
1 public interface PropertyResolver {
2     boolean containsProperty(String key);
3     String getProperty(String key);
4     String getProperty(String key, String defaultValue);
5     <T> T getProperty(String key, Class<T> targetType);
6     <T> T getProperty(String key, Class<T> targetType, T defaultValue);
7     <T> Class<T> getPropertyAsClass(String key, Class<T> targetType);
8     String getRequiredProperty(String key) throws IllegalStateException;
9     <T> T getRequiredProperty(String key, Class<T> targetType) throws
10    IllegalStateException;
11     String resolvePlaceholders(String text);
12     String resolveRequiredPlaceholders(String text) throws
13    IllegalArgumentException;
14 }
```

## IoC 引擎

接下来我们看看 IoC 引擎——DefaultListableBeanFactory 的实现。

```
1 public class DefaultListableBeanFactory extends
2    AbstractAutowireCapableBeanFactory
3        implements ConfigurableListableBeanFactory{
4     public int getBeanDefinitionCount() {
5         return this.beanDefinitionMap.size();
6     }
7     public String[] getBeanDefinitionNames() {
8         return (String[]) this.beanDefinitionNames.toArray();
9     }
10    public String[] getBeanNamesForType(Class<?> type) {
11        List<String> result = new ArrayList<>();
12        for (String beanName : this.beanDefinitionNames) {
13            boolean matchFound = false;
14            BeanDefinition mbd = this.getBeanDefinition(beanName);
15            Class<?> classToMatch = mbd.getClass();
16            if (type.isAssignableFrom(classToMatch)) {
17                matchFound = true;
18            }
19            else {
20                matchFound = false;
21            }
22            if (matchFound) {
23                result.add(beanName);
24            }
25        }
26        return result.toArray(new String[result.size()]);
27    }
28 }
```

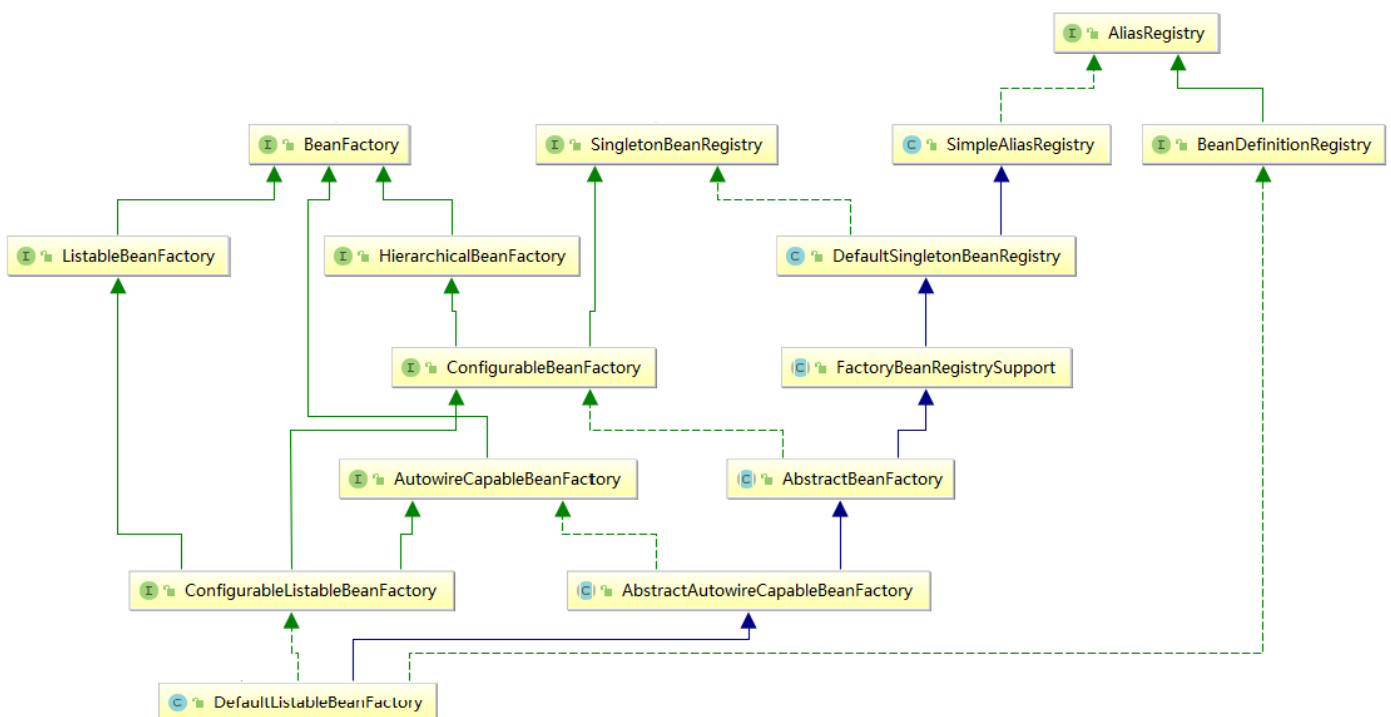
```

24         }
25     }
26     return (String[]) result.toArray();
27 }
28 @SuppressWarnings("unchecked")
29 @Override
30 public <T> Map<String, T> getBeansOfType(Class<T> type) throws BeansException
31 {
32     String[] beanNames = getBeanNamesForType(type);
33     Map<String, T> result = new LinkedHashMap<>(beanNames.length);
34     for (String beanName : beanNames) {
35         Object beanInstance = getBean(beanName);
36         result.put(beanName, (T) beanInstance);
37     }
38     return result;
39 }
40 }
41

```

从上述代码中，似乎看不出这个类是如何成为 IoC 引擎的，因为它的实现都是很简单地获取各种属性的方法。它成为引擎的秘诀在于**它继承了其他 BeanFactory 类来实现 Bean 的创建管理功能**。从代码可以看出它继承了 AbstractAutowireCapableBeanFactory 并实现了 ConfigurableListableBeanFactory 接口。

参看 Spring 框架的这一部分，整个继承体系图。






可以看出，我们的 MiniSpring 跟 Spring 框架设计得几乎是一模一样。当然，这是我们有意为之，我们手写 MiniSpring 就是为了深入理解 Spring。

当 ClassPathXmlApplicationContext 这个 Spring 核心启动类运行时，注入了 DefaultListableBeanFactory，为整个 Spring 框架做了默认实现，这样就完成了框架内部的逻辑闭环。

## 事件


接着我们来完善事件的发布与监听，包括 ApplicationEvent、ApplicationListener、ApplicationEventPublisher 以及 ContextRefreshEvent，事件一经发布就能让监听者监听到。

### ApplicationEvent

 复制代码

```
1 public class ApplicationEvent extends EventObject {
2     private static final long serialVersionUID = 1L;
3     protected String msg = null;
4     public ApplicationEvent(Object arg0) {
5         super(arg0);
6         this.msg = arg0.toString();
7     }
8 }
```


### ApplicationListener

 复制代码

```
1 public class ApplicationListener implements EventListener {
2     void onApplicationEvent(ApplicationEvent event) {
3         System.out.println(event.toString());
4     }
5 }
```




## ContextRefreshEvent

 复制代码

```
1 public class ContextRefreshEvent extends ApplicationEvent{
2     private static final long serialVersionUID = 1L;
3     public ContextRefreshEvent(Object arg0) {
4         super(arg0);
5     }
6
7     public String toString() {
8         return this.msg;
9     }
10 }
```

## ApplicationEventPublisher

 复制代码

```
1 public interface ApplicationEventPublisher {
2     void publishEvent(ApplicationEvent event);
3     void addApplicationListener(ApplicationListener listener);
4 }
```

可以看出，框架的 EventPublisher，本质是对 JDK 事件类的封装。接口已经定义好了，接下来我们实现一个最简单的事件发布者 SimpleApplicationEventPublisher。

```
1 public class SimpleApplicationEventPublisher implements
2 ApplicationEventPublisher{
3     List<ApplicationListener> listeners = new ArrayList<>();
4     @Override
5     public void publishEvent(ApplicationEvent event) {
6         for (ApplicationListener listener : listeners) {
7             listener.onApplicationEvent(event);
8         }
9     }
10    @Override
11    public void addApplicationListener(ApplicationListener listener) {
12        this.listeners.add(listener);
13    }
14 }
```

这个事件发布监听机制就可以为后面 ApplicationContext 的使用服务了。

## 完整的 ApplicationContext

最后，我们来完善 ApplicationContext，并把它作为公共接口，所有的上下文都实现自

ApplicationContext，支持上下文环境和事件发布。


```
1 public interface ApplicationContext
2     extends EnvironmentCapable, ListableBeanFactory, ConfigurableBeanFactory,
3 ApplicationEventPublisher{
4 }
```

我们计划做 4 件事。

1. 抽取 ApplicationContext 接口，实现更多有关上下文的内容。
2. 支持事件的发布与监听。
3. 新增 AbstractApplicationContext，规范刷新上下文 refresh 方法的步骤规范，且将每一步骤进行抽象，提供默认实现类，同时支持自定义。

#### 4. 完成刷新之后发布事件。

首先我们来增加 ApplicationContext 接口的内容，丰富它的功能。

 复制代码

```
1 public interface ApplicationContext
2     extends EnvironmentCapable, ListableBeanFactory,
3     ConfigurableBeanFactory, ApplicationEventPublisher{
4     String getApplicationName();
5     long getStartupDate();
6     ConfigurableListableBeanFactory getBeanFactory() throws
7     IllegalStateException;
8     void setEnvironment(Environment environment);
9     Environment getEnvironment();
10    void addBeanFactoryPostProcessor(BeanFactoryPostProcessor postProcessor);
11    void refresh() throws BeansException, IllegalStateException;
12    void close();
13    boolean isActive();
14 }
```

还是按照以前的模式，先定义接口，然后用一个抽象类搭建框架，最后提供一个具体实现类进行默认实现。Spring 的这个 interface-abstract-class 模式是值得我们学习的，它极大地增强了框架的扩展性。

我们重点看看 AbstractApplicationContext 的实现。因为现在我们只做到了从 XML 里读取配置，用来获取应用的上下文信息，但实际 Spring 框架里不只支持这一种方式。但无论哪种方式，究其本质都是对应用上下文的处理，所以我们来抽象 ApplicationContext 的公共部分。

 复制代码


```
1 public abstract class AbstractApplicationContext implements ApplicationContext{
2     private Environment environment;
3     private final List<BeanFactoryPostProcessor> beanFactoryPostProcessors = new
4     ArrayList<>();
5     private long startupDate;
6     private final AtomicBoolean active = new AtomicBoolean();
7     private final AtomicBoolean closed = new AtomicBoolean();
8     private ApplicationEventPublisher applicationEventPublisher;
9     @Override
```

```

10     public Object getBean(String beanName) throws BeansException {
11         return getBeanFactory().getBean(beanName);
12     }
13     public List<BeanFactoryPostProcessor> getBeanFactoryPostProcessors() {
14         return this.beanFactoryPostProcessors;
15     }
16     public void refresh() throws BeansException, IllegalStateException {
17         postProcessBeanFactory(getBeanFactory());
18         registerBeanPostProcessors(getBeanFactory());
19         initApplicationEventPublisher();
20         onRefresh();
21         registerListeners();
22         finishRefresh();
23     }
24     abstract void registerListeners();
25     abstract void initApplicationEventPublisher();
26     abstract void postProcessBeanFactory(ConfigurableListableBeanFactory
27 beanFactory);
28     abstract void registerBeanPostProcessors(ConfigurableListableBeanFactory
29 beanFactory);
30     abstract void onRefresh();
31     abstract void finishRefresh();
32     @Override
33     public String getApplicationName() {
34         return "";
35     }
36     @Override
37     public long getStartupDate() {
38         return this.startupDate;
39     }
40     @Override
41     public abstract ConfigurableListableBeanFactory getBeanFactory() throws
42 IllegalStateException;
43     @Override
44     public void addBeanFactoryPostProcessor(BeansException postProcessor) {
45         this.beanFactoryPostProcessors.add(postProcessor);
46     }
47     @Override
48     public void close() {
49     }
50     @Override
51     public boolean isActive(){
52         return true;
53     }
54     //省略包装beanfactory的方法
55 }

```


上面这段代码的核心是 refresh() 方法的定义，而这个方法又由下面这几个步骤组成。

 复制代码

```
1    abstract void registerListeners();
2    abstract void initApplicationEventPublisher();
3    abstract void postProcessBeanFactory(ConfigurableListableBeanFactory
4    beanFactory);
5    abstract void registerBeanPostProcessors(ConfigurableListableBeanFactory
6    beanFactory);
7    abstract void onRefresh();
8    abstract void finishRefresh();
```

看名字就比较容易理解，首先是注册监听者，接下来初始化事件发布者，随后处理 Bean 以及对 Bean 的状态进行一些操作，最后是将初始化完毕的 Bean 进行应用上下文刷新以及完成刷新后进行自定义操作。因为这些方法都有 abstract 修饰，允许把这些步骤交给用户自定义处理，因此极大地增强了扩展性。

我们现在已经拥有了一个 ClassPathXmlApplicationContext，我们以这个类为例，看看如何实现上面的几个步骤。ClassPathXmlApplicationContext 代码改造如下：

 复制代码

```
1    public class ClassPathXmlApplicationContext extends AbstractApplicationContext{
2        DefaultListableBeanFactory beanFactory;
3        private final List<BeanFactoryPostProcessor> beanFactoryPostProcessors = new
4        ArrayList<>();
5        public ClassPathXmlApplicationContext(String fileName) {
6            this(fileName, true);
7        }
8        public ClassPathXmlApplicationContext(String fileName, boolean isRefresh) {
9            Resource resource = new ClassPathXmlResource(fileName);
10           DefaultListableBeanFactory beanFactory = new
11           DefaultListableBeanFactory();
12           XmlBeanDefinitionReader reader = new
13           XmlBeanDefinitionReader(beanFactory);
14           reader.loadBeanDefinitions(resource);
15           this.beanFactory = beanFactory;
16           if (isRefresh) {
17               try {
18                   refresh();
19               }
20           }
```

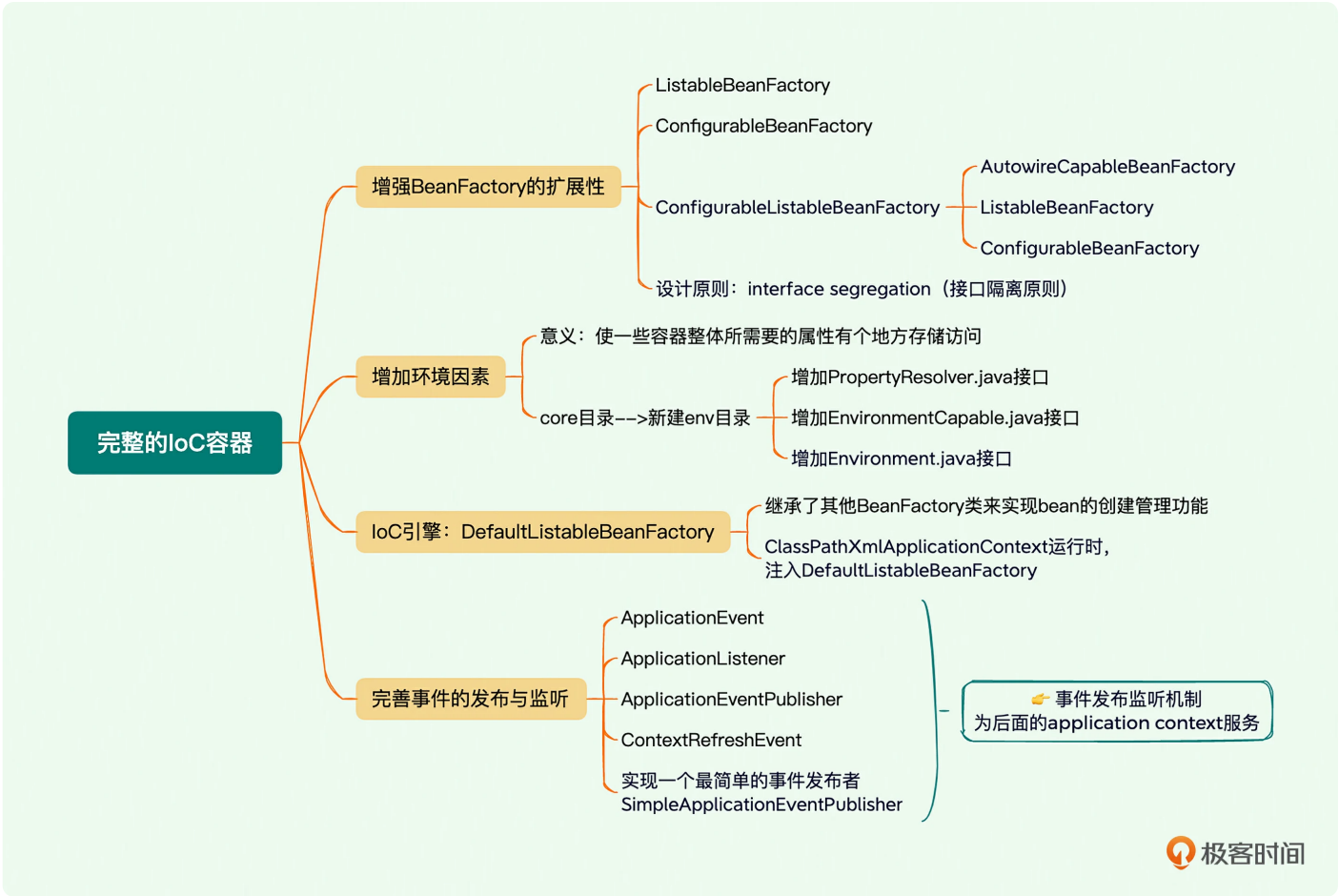
```

21     }
22     @Override
23     void registerListeners() {
24         ApplicationListener listener = new ApplicationListener();
25         this.getApplicationEventPublisher().addApplicationListener(listener);
26     }
27     @Override
28     void initApplicationEventPublisher() {
29         ApplicationEventPublisher aep = new SimpleApplicationEventPublisher();
30         this.setApplicationEventPublisher(aep);
31     }
32     @Override
33     void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) {
34     }
35     @Override
36     public void publishEvent(ApplicationEvent event) {
37         this.getApplicationEventPublisher().publishEvent(event);
38     }
39     @Override
40     public void addApplicationListener(ApplicationListener listener) {
41         this.getApplicationEventPublisher().addApplicationListener(listener);
42     }
43     public void addBeanFactoryPostProcessor(BeanFactoryPostProcessor
44 postProcessor) {
45         this.beanFactoryPostProcessors.add(postProcessor);
46     }
47     @Override
48     void registerBeanPostProcessors(ConfigurableListableBeanFactory beanFactory)
49 {
50         this.beanFactory.addBeanPostProcessor(new
51 AutowiredAnnotationBeanPostProcessor());
52     }
53     @Override
54     void onRefresh() {
55         this.beanFactory.refresh();
56     }
57     @Override
58     public ConfigurableListableBeanFactory getBeanFactory() throws
59 IllegalStateException {
60         return this.beanFactory;
61     }
62     @Override
63     void finishRefresh() {
64         publishEvent(new ContextRefreshEvent("Context Refreshed..."));
65     }
66 }

```

上述代码分别实现了几个抽象方法，就很高效地把 ClassPathXmlApplicationContext 类融入到了 ApplicationContext 框架里了。Spring 的这个设计模式值得我们学习，采用抽象类的方式来解耦，为用户提供了极大的扩展性的便利，这也是 Spring 框架强大的原因之一。Spring 能集成 MyBatis、MySQL、Redis 等框架，少不了设计模式在背后支持。

至此，我们的 IoC 容器就完成了，它很简单，但是这个容器麻雀虽小五脏俱全，关键是我们深入理解 Spring 框架提供了很好的解剖样本。



## 小结

经过这节课的学习，我们初步构造了一个完整的 IoC 容器，目前它的功能包括 4 项。

1. 识别配置文件中的 Bean 定义，创建 Bean，并放入容器中进行管理。
2. 支持配置方式或者注解方式进行 Bean 的依赖注入。
3. 构建了 BeanFactory 体系。



#### 4. 容器应用上下文和事件发布。

对照 Spring 框架，上述几点就是 Spring IoC 的核心。通过这个容器，我们构建应用程序的时候，将业务逻辑封装在 Bean 中，把对 Bean 的创建管理交给框架，即所谓的“控制反转”，应用程序与框架程序互动，共同运行完整程序。

实现这些概念和特性的手段和具体代码，我们都有意模仿了 Spring，它们的结构和名字都是一样的，所以你回头阅读 Spring 框架本身代码的时候，会觉得很熟悉，学习曲线平滑。我们沿着大师的脚步往前走，不断参照大师的作品，吸收大师的养分培育自己，让我们的 MiniSpring 一步步成长为一棵大树。

完整源代码参见 [🔗 https://github.com/YaleGuo/minis](https://github.com/YaleGuo/minis)

## 课后题

学完这节课，我也给你留一道思考题。我们的容器以单例模式管理所有的 Bean，那么怎么应对多线程环境？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！

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

## 精选留言 (15)



马儿

2023-03-23 来自四川

请教老师一下，

1. ClassPathXmlApplicationContext 和 AbstractApplicationContext 都有 beanFactoryPostProcessors 属性，是不是重复了呢？感觉直接复用父类的这个属性和相关方法也是可以的。
2. AbstractAutowireCapableBeanFactory 这个类中的 beanPostProcessors 属性写死了是 AutowiredAnnotationBeanPostProcessor，不符合面向接口编程的风格。另外由于没有面向 BeanPostProcessor 导致 DefaultListableBeanFactory 需要再实现一遍 SingletonBeanRegistry
3. AbstractBeanFactory 实现了 BeanFactory 又写了两个抽象方法 applyBeanPostProcessorsBeforeInitialization 和 applyBeanPostProcessorAfterInitialization，这里为什么不直接实现 AutowireCapableBeanFactory 呢？

作者回复: 你很用心。

- 1, 是重复了。
- 2, 程序最后变成了BeanPostProcessor接口, 你接着看。
- 3, 后面有个beanfactory的继承关系图, 你可以看看。

共 2 条评论 >

👍 1



**KernelStone**

2023-06-02 来自广东

这一小结其实新增的内容不算多! 只是对之前已有的代码进行结构调整。在项目中对DefaultListableBeanFactory生成UML结构图, 再进行从上到下的梳理, 这样会舒服一些。

0、【接口】BF, Bean工厂

1、【接口】SingletonBeanRegistry, 单例Bean仓库

2、DefaultSingletonBeanRegistry, 单例Bean仓库默认实现。提供了 1 注册列表 2 单例容器 3 依赖注入管理信息 (两个Map, 应该是依赖 & 被依赖)

3、【接口】BeanDefinitionRegistry 【接口】ListableBF, 这两个对照看差异。前者强调对BeanDefinition进行操作, 后者强调是对List集合进行操作。

4、【接口】ConfigurableBF, Bean处理器 (add & get, 没有apply), 以及管理依赖信息。

5、【接口】AutowireCapableBF, 提供自动装配选项 (No、byName、byType), 并在初始化前后应用 (apply) Bean处理器。

6、【集成接口】ConfigurableListableBF, 无内容。

7、【抽象类】AbstractBF, 主要是refresh(), invokeInitMethod(), createBean(), 构造器注入和属性注入。

8、AbstractAutowireCapableBF, 提供成员List<BeanPostProcessor>! 也因此它可以通过该成员进行更多的bean处理器操作, 即add、get、apply在此有了具体实现。

9、DefaultListableBF, 其实没有啥, 打开一看只Override了【接口】ListableBF中的4个方法, 其余是默认继承。(即沿着类结构往上一堆, 上面也说过)

因此, 这节课真没什么新东西, 不过梳理这个新的工厂体系, 倒是很麻烦。。

作者回复: 你这个总结真好!



**怕什么，抱紧我**

2023-05-30 来自中国香港

ConfigurableBeanFactory定义了getDependentBeans()方法;  
ConfigurableBeanFactory的实现类是DefaultListableBeanFactory, 但是  
DefaultListableBeanFactory没有实现getDependentBeans()方法, 居然没有报错!  
要是极客时间能发图, 我肯定发一个图上来!  
我到底错哪儿了!

作者回复: 看一下DefaultSingletonBeanRegistry



**怕什么，抱紧我**

2023-05-30 来自中国香港

原谅我实在没有看明白

ConfigurableBeanFactory接口, 有一个方法getDependentBeans();  
DefaultListableBeanFactory是它的实现类, 大师并没有实现getDependentBeans这个方法,  
表示看的很懵b

作者回复: `public class DefaultListableBeanFactory extends AbstractAutowireCapableBeanFactory  
implements ConfigurableListableBeanFactory`

你按照这个继承体系一层层往上找



**梦幻之梦想**

2023-04-25 来自陕西

我想问下DefaultListableBeanFactory中的beanDefinitionMap是怎么来的

作者回复: AbstractBeanFactory中继承下来的。你看一下Github上的全代码。



**CSY.**

2023-04-07 来自河南

老师我有个问题

ConfigurableBeanFactory 中的 dependentBeanMap 等几个方法为什么要使用同级继承在DefaultSingletonBeanRegistry实现, 而不在AbstractBeanFactory等中实现?

作者回复: 并没有特别的理由, 就是参考的Spring的做法。



**啊良梓是我**

2023-04-03 来自广东

```
String className = beanDefinition.getClassName();
Class<?> aClass = null;
try {
    aClass = Class.forName(className);
} catch (ClassNotFoundException e) {
    throw new RuntimeException(e);
}
```

应该是这样子获取BeanDefinition定义的Bean类型才对?

作者回复: 你这么做也是可以的, 不过就是重新加载了类。



**啊良梓是我**

2023-04-03 来自广东

```
BeanDefinition mbd = this.getBeanDefinition(beanName); Class classToMatch = mbd.getClass();
```

这里为什么是拿BeanDefinition的Class的?这样子没意义吧?或者我漏掉什么了?

前面存储Bean class 是 BeanDefinition的BeanName 才对.

作者回复: 你再仔细看代码, 这个class代表的是哪个?

共 4 条评论 >



**啊良梓是我**

2023-04-03 来自广东

```
package com.minis.beans.factory.config;
import com.minis.beans.factory.ListableBeanFactory;
public interface ConfigurableListableBeanFactory
```

```
extends ListableBeanFactory, AutowireCapableBeanFactory,  
ConfigurableBeanFactory {  
}
```

这里是伪代码？ AutowireCapableBeanFactory按照流程下来，这里是一个Class的来哦。。。怎么可以用interface继承他的呢

作者回复: 从github上下载完整代码



**宋健**

2023-04-03 来自广东

老师好，我想问几个小问题：

1. 请问postProcessBeanFactory这个抽象方法的作用是什么呢？
2. 我是不是可以在 registerBeanPostProcessors 中添加自己额外自定义的 BeanPostProcessor 来实现其他的注解解释器？

作者回复: 1. 你看时序，先有容器的启动，然后才到加载各个bean.。这里的postprocessbeanfactory就是一个节点，让程序员在容器启动后可以进行自己的处理。

2. 可以的，你可以自己增加对bean的额外修饰代码进行后期处理。



**Geek\_83a70c**

2023-04-02 来自广东

老师好，为什么ListableBeanFactory和ConfigurableBeanFactory、AutowireCapableBeanFactory都要继承BeanFactory接口，如果按照接口隔离思想，不是越隔离越好吗？例如以上3个接口根本其实无需涉及BeanFactory中的getBean()这个最主要的方法

作者回复: 它们都是beanfactory啊，只是有不同的特性，它们本身构成一个工厂体系。接口隔离不是指这个地方，是说实现一个类，需要什么能力，就加一个什么接口，互相隔开，独立使用。



**Geek\_513706**

2023-03-28 来自内蒙古

老师，想提个建议，以后添加代码的时候能不能把添加到哪个包里面说清楚

作者回复: 这个建议已经接纳了, 后面的文稿都带上了。完整代码要看Github上的。



**摩诃不思議**

2023-03-25 来自浙江

这节代码变化的太快了...

作者回复: 昨天分成了两个分支。geek\_ioc4和geek\_ioc5.你再看看。



**风轻扬**

2023-03-24 来自北京

思考题: Spring的bean作用域默认是单例的, 就是我们的DefaultSingletonBeanRegistry类中持有的那个那个singletons的ConcurrentHashMap, 每次获取bean之前, 都会先从这个单例map中获取, 获取不到才创建。

如果是多线程场景, 有竞态条件存在的情况下, 可以考虑将bean的作用域改为Prototype类型, 对于Prototype类型的bean, Spring会为每次get请求都新建bean, 所以每个请求获取到的bean是不一样的, 这样就没有并发问题了

除了这两种作用域, 还有另外四种作用域, 我没怎么接触过, 看了一下官方文档了解了一下。文档地址: <https://docs.spring.io/spring-framework/docs/5.3.27-SNAPSHOT/reference/html/core.html#beans-factory-scopes>

遇到Spring的问题, 可以多看看他们的文档, 比搜索引擎强多了, 写的很清晰

另外, 我有一个问题, 请教一下老师, ClassPathXmlApplicationContext为啥要实现BeanFactory? 感觉他们两个不是一个体系里的吧, 一个是上下文, 一个是bean工厂

作者回复: 实现BeanFactory接口的原因是为了对外提供同样的API。



**peter**

2023-03-22 来自北京

请教老师几个问题:

Q1: ApplicationEvent 类中定义的"serialVersionUID = 1L"有什么用? 如果有用, 为什么有的类没有定义serialVersionUID? (对于serialVersionUID, 以前好像有一篇博客讲过, 但一直没在意过, 没有用过, 好像也没有什么问题)

Q2: 文中提到的“设计模式”属于23种设计模式吗?

文中提到“Spring 的这个设计模式值得我们学习, 采用抽象类的方式来解耦, 为用户提供了极大的扩展性的便利”, 这里提到的“设计模式”, 应该不是常说的23种设计模式吧。

Q3: beans.xml文件必须放在Resource目录下面吗? 老师的工程, Resource目录与src目录是平级, 但我以前建的工程, resource目录在src/main目录下面,main下面有两个目录, java和resource, 这两个平级。这两种目录结构都可以吗?

Q4: 系统有自己缺省的处理类, 系统启动过程也是固定的。用户怎么利用扩展性? 比如, 用户想修改或增加某个功能, 怎么实现?

作者回复: ApplicationEvent实现Serializable接口, 按照约定, 要有这个serialVersionUID.

我说的“设计模式”, 并不一定指23种之一, 那23种只是GoF书中列出的而已, 设计模式就是一种解决问题的固定方案, 我们在编程中都要自己用心提取设计模式。

不同的目录结构都可以的, 能找到就行。

扩展性有不同程度, 单独提炼出接口, 然后给了默认实现, 有源码就可以改, 如果提供了注入机制, 也可以通过配置进行扩展。后面MVC有例子。

