

01 | 原始IoC：如何通过BeanFactory实现原始版本的IoC容器？

2023-03-13 郭屹 来自北京

《手把手带你写一个MiniSpring》



你好，我是郭屹，从今天开始我们来学习手写 MiniSpring。

这一章，我们将从一个最简单的程序开始，一步步堆积演化，最后实现 Spring 这一庞大框架的核心部分。这节课，我们就来构造第一个程序，也是最简单的一个程序，将最原始的 IoC 概念融入我们的框架之中，**我们就用这个原始的 IoC 容器来管理一个 Bean**。不过要说的是，它虽然原始，却也是一个可以运行的 IoC 容器。

IoC 容器

如果你使用过 Spring 或者了解 Spring 框架，肯定会对 IoC 容器有所耳闻。它的意思是使用 Bean 容器管理一个个的 Bean，最简单的 Bean 就是一个 Java 的业务对象。在 Java 中，创建一个对象最简单的方法就是使用 new 关键字。IoC 容器，也就是 **BeanFactory**，**存在的意义就是将创建对象与使用对象的业务代码解耦**，让业务开发人员无需关注底层对象（Bean）的构建和生命周期管理，专注于业务开发。

那我们可以先想一想，怎样实现 Bean 的管理呢？我建议你不要直接去参考 Spring 的实现，那是大树长成之后的模样，复杂而庞大，令人生畏。


作为一颗种子，它其实可以非常原始、非常简单。实际上我们只需要几个简单的部件：我们用一个部件来对应 Bean 内存的映像，一个定义在外面的 Bean 在内存中总是需要有一个映像的；一个 XML reader 负责从外部 XML 文件获取 Bean 的配置，也就是说这些 Bean 是怎么声明的，我们可以写在一个外部文件里，然后我们用 XML reader 从外部文件中读取进来；我们还需要一个反射部件，负责加载 Bean Class 并且创建这个实例；创建实例之后，我们用一个 Map 来保存 Bean 的实例；最后我们提供一个 getBean() 方法供外部使用。我们这个 IoC 容器就做好了。



好，接下来我们一步步来构造。

实现一个原始版本的 IoC 容器

对于现在要着手实现的原始版本 Bean，我们先只管理两个属性：**id** 与 **class**。其中，class 表示要注入的类，而 id 则是给这个要注入的类一个别名，它可以简化记忆的成本。我们要做的是把 Bean 通过 XML 的方式注入到框架中，你可以看一下 XML 的配置。

 复制代码


```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <beans>
3     <bean id = "xxxid" class = "com.minis.xxxclass"></bean>
4 </beans>
```

接下来我们要做一些准备工作。首先，新建一个 Java 项目，导入 dom4j-1.6.1.jar 包。这里导入的 dom4j 包封装了许多操作 XML 文件的方法，有助于我们快速处理 XML 文件中的各种属性，这样就不需要我们自己再写一个 XML 的解析工具了，同时它也为我们后续处理依托于 XML 注入的 Bean 提供了便利。

另外要说明的是，我们写 MiniSpring 是为了学习 Spring 框架，所以我们会尽量少地去依赖第三方包，多自己动手，以原始社会刀耕火种的方式写程序，这可以让我们彻底地理解底层原理。希望你能够跟我一起动手，毕竟编程说到底是一个手艺活，要想提高编程水平，唯一的方法就是动手去写。只要不断学，不断想，不断做，就能大有成效。

构建 BeanDefinition

好了，在有了第一个 Java 项目后，我们创建 com.minis 包，我们所有的程序都是放在这个包下的。在这个包下构建第一个类，对应 Bean 的定义，命名为 BeanDefinition。我们在这个类里面定义两个最简单的域：id 与 className。你可以看一下相关代码。


 复制代码

```
1 public class BeanDefinition {
2     private String id;
3     private String className;
4     public BeanDefinition(String id, String className) {
5         this.id = id;
6         this.className = className;
7     }
8     //省略getter和setter
```

可以看到，这段代码为这样一个 Bean 提供了全参数的构造方法，也提供了基本的 getter 和 setter 方法，方便我们获取域的值以及对域里的值赋值。

实现 ClassPathXmlApplicationContext

接下来，我们假定已经存在一个用于注入 Bean 的 XML 文件。那我们要做的自然是，按照一定的规则将这个 XML 文件的内容解析出来，获取 Bean 的配置信息。我们的第二个类 ClassPathXmlApplicationContext 就可以做到这一点。通过这个类的名字也可以看出，它的作用是解析某个路径下的 XML 来构建应用上下文。让我们来看看如何初步实现这个类。

 复制代码

```
1 public class ClassPathXmlApplicationContext {
2     private List<BeanDefinition> beanDefinitions = new ArrayList<>();
3     private Map<String, Object> singletons = new HashMap<>();
4     //构造器获取外部配置，解析出Bean的定义，形成内存映像
5     public ClassPathXmlApplicationContext(String fileName) {
6         this.readXml(fileName);
7         this.instanceBeans();
8     }
9     private void readXml(String fileName) {
10         SAXReader saxReader = new SAXReader();
11         try {
12             URL xmlPath =
13 this.getClass().getClassLoader().getResource(fileName);
14             Document document = saxReader.read(xmlPath);
15             Element rootElement = document.getRootElement();
16             //对配置文件中的每一个<bean>，进行处理
17             for (Element element : (List<Element>) rootElement.elements()) {
18                 //获取Bean的基本信息
19                 String beanID = element.attributeValue("id");
20                 String beanClassName = element.attributeValue("class");
21                 BeanDefinition beanDefinition = new BeanDefinition(beanID,
22 beanClassName);
23                 //将Bean的定义存放到beanDefinitions
24                 beanDefinitions.add(beanDefinition);
25             }
26         }
27     }
28     //利用反射创建Bean实例，并存储在singletons中
29     private void instanceBeans() {
30         for (BeanDefinition beanDefinition : beanDefinitions) {
31             try {
32                 singletons.put(beanDefinition.getId(),
33 Class.forName(beanDefinition.getClassName()).newInstance());
```

```
34         }
35     }
36 }
37 //这是对外面的一个方法，让外部程序从容器中获取Bean实例，会逐步演化成核心方法
38 public Object getBean(String beanName) {
39     return singletons.get(beanName);
40 }
41 }
```

由上面这一段代码可以看出，ClassPathXmlApplicationContext 定义了唯一的构造函数，构造函数里会做两件事：一是提供一个 readXml() 方法，通过传入的文件路径，也就是 XML 文件的全路径名，来获取 XML 内的信息，二是提供一个 instanceBeans() 方法，根据读取到的信息实例化 Bean。接下来让我们看看，readXml 和 instanceBeans 这两个方法分别做了什么。

首先来看 readXML，这也是我们解析 Bean 的核心方法，因为配置在 XML 内的 Bean 信息都是文本信息，需要解析之后变成内存结构才能注入到容器中。该方法最开始创建了 SAXReader 对象，这个对象是 dom4j 包内提供的。随后，它通过传入的 fileName，也就是定义的 XML 名字，获取根元素，也就是 XML 里最外层的标签。然后它循环遍历标签中的属性，通过 element.attributeValue("id") 和 element.attributeValue("class") 拿到配置信息，接着用这些配置信息构建 BeanDefinition 对象，然后把 BeanDefinition 对象加入到 BeanDefinitions 列表中，这个地方就保存了所有 Bean 的定义。

接下来，我们看看 instanceBeans 方法实现的功能：实例化一个 Bean。因为 BeanDefinitions 存储的 BeanDefinition 的 class 只是一个类的全名，所以我们现在需要将这个名字转换成一个具体的类。我们可以通过 Java 里的反射机制，也就是 Class.forName 将一个类的名字转化成一个实际存在的类，转成这个类之后，我们把它放到 singletons 这个 Map 里，构建 ID 与实际类的映射关系。

到这里，我们就把 XML 文件中的 Bean 信息注入到了容器中。你可能会问，我到现在都没看到 BeanFactory 呀，是不是还没实现完？

其实不是的，目前的 `ClassPathXmlApplicationContext` 兼具了 `BeanFactory` 的功能，它通过 `singletons` 和 `beanDefinitions` 初步实现了 `Bean` 的管理，其实这也是 Spring 本身的做法。后面我会进一步扩展的时候，会分离这两部分功能，来剥离出一个独立的 `BeanFactory`。


验证功能

现在，我们已经实现了第一个管理 `Bean` 的容器，但还要验证一下我们的功能是不是真的实现了。下面我们就编写一下测试代码。在 `com.minis` 目录下，新增 `test` 包。你可以看一下相关的测试代码。

 复制代码


```
1 public interface AService {  
2     void sayHello();  
3 }
```

这里，我们定义了一个 `sayHello` 接口，该接口的实现是在控制台打印出 “a service 1 say hello” 这句话。

 复制代码

```
1 public class AServiceImpl implements AService {  
2     public void sayHello() {  
3         System.out.println("a service 1 say hello");  
4     }  
5 }
```

我们将 XML 文件命名为 `beans.xml`，注入 `AServiceImpl` 类，起个别名，为 `aservice`。

 复制代码

```
1 <?xml version="1.0" encoding="UTF-8" ?>  
2 <beans>  
3     <bean id = "aservice" class = "com.minis.test.AServiceImpl"></bean>  
4 </beans>
```

除了测试代码，我们还需要启动类，定义 `main` 函数。


```
1 public class Test1 {  
2     public static void main(String[] args) {  
3         ClassPathXmlApplicationContext ctx = new  
4 ClassPathXmlApplicationContext("beans.xml");  
5         AService aService = (AService)ctx.getBean("aservice");  
6         aService.sayHello();  
7     }  
8 }
```

在启动函数中可以看到，我们构建了 `ClassPathXmlApplicationContext`，传入文件名为 “beans.xml”，也就是我们在测试代码中定义的 XML 文件名。随后我们通过 `getBean` 方法，获取注入到 singletons 里的这个类 `AService`。`aService` 在这儿是 `AService` 接口类型，其底层实现是 `AServicImpl`，这样再调用 `AServicImpl` 类中的 `sayHello` 方法，就可以在控制台打印出 “a service 1 say hello” 这一句话。

到这里，我们已经成功地构造了一个最简单的程序：**最原始的 IoC 容器**。在这个过程中我们引入了 `BeanDefinition` 的概念，也实现了一个应用的上下文 `ClassPathXmlApplicationContext`，从外部的 XML 文件中获取文件信息。只用了很少的步骤就实现了 IoC 容器对 Bean 的管理，后续就不再需要我们手动地初始化这些 Java 对象了。

解耦 `ClassPathXmlApplicationContext`

但是我们也可以看到，这时的 `ClassPathXmlApplicationContext` 承担了太多的功能，这并不符合我们常说的对象单一功能的原则。因此，我们需要做的优化扩展工作也就呼之欲出了：分解这个类，主要工作就是两个部分，一是提出一个最基础的核心容器，二是把 XML 这些外部配置信息的访问单独剥离出去，现在我们只有 XML 这种方式，但是之后还有可能配置到 Web 或数据库文件里，拆解出去之后也便于扩展。


为了看起来更像 Spring，我们以 Spring 的目录结构为范本，重新构造一下我们的项目代码结构。

```
1 com.minis.beans;  
2 com.minis.context;
```

```
3 com.minis.core;
4 com.minis.test;
```

定义 BeansException

在正式开始解耦工作之前，我们先定义属于我们自己的异常处理类：BeansException。我们来看看异常处理类该如何定义。


 复制代码

```
1 public class BeansException extends Exception {
2     public BeansException(String msg) {
3         super(msg);
4     }
5 }
```

可以看到，现在的异常处理类比较简单，它是直接调用父类（Exception）处理并抛出异常。有了这个基础的 BeansException 之后，后续我们可以根据实际情况对这个类进行拓展。

定义 BeanFactory

首先要拆出一个基础的容器来，刚才我们反复提到了 BeanFactory 这个词，现在我们正式引入 BeanFactory 这个接口，先让这个接口拥有两个特性：一是获取一个 Bean（getBean），二是注册一个 BeanDefinition（registerBeanDefinition）。你可以看一下它们的定义。

 复制代码

```
1 public interface BeanFactory {
2     Object getBean(String beanName) throws BeansException;
3     void registerBeanDefinition(BeansDefinition beanDefinition);
4 }
```

定义 Resource

刚刚我们将 BeanFactory 的概念进行了抽象定义。接下来我们要定义 Resource 这个概念，我们把外部的配置信息都当成 Resource（资源）来进行抽象，你可以看下相关接口。


```
1 public interface Resource extends Iterator<Object> {
2 }
```

定义 ClassPathXmlResource

目前我们的数据来源比较单一，读取的都是 XML 文件配置，但是有了 Resource 这个接口后面我们就可以扩展，从数据库还有 Web 网络上面拿信息。现在有 BeanFactory 了，有 Resource 接口了，拆解这两部分的接口也都有了。接下来就可以来实现了。

现在我们读取并解析 XML 文件配置是在 ClassPathXmlApplicationContext 类中完成的，所以我们下一步的解耦工作就是定义 ClassPathXmlResource，将解析 XML 的工作交给它完成。


```
1 public class ClassPathXmlResource implements Resource{
2     Document document;
3     Element rootElement;
4     Iterator<Element> elementIterator;
5     public ClassPathXmlResource(String fileName) {
6         SAXReader saxReader = new SAXReader();
7         URL xmlPath = this.getClass().getClassLoader().getResource(fileName);
8         //将配置文件装载进来，生成一个迭代器，可以用于遍历
9         try {
10             this.document = saxReader.read(xmlPath);
11             this.rootElement = document.getRootElement();
12             this.elementIterator = this.rootElement.elementIterator();
13         }
14     }
15     public boolean hasNext() {
16         return this.elementIterator.hasNext();
17     }
18     public Object next() {
19         return this.elementIterator.next();
20     }
21 }
```

操作 XML 文件格式都是 dom4j 帮我们做的。

注：dom4j 这个外部 jar 包方便我们读取并解析 XML 文件内容，将 XML 的标签以及参数转换成 Java 的对象。当然我们也可以自行写代码来解析文件，但是为了简化代码，避免重复造轮子，这里我们选择直接引用第三方包。

XmlBeanDefinitionReader

现在我们已经解析好了 XML 文件，但解析好的 XML 如何转换成我们需要的 BeanDefinition 呢？这时 XmlBeanDefinitionReader 就派上用场了。


 复制代码

```
1 public class XmlBeanDefinitionReader {
2     BeanFactory beanFactory;
3     public XmlBeanDefinitionReader(BeanFactory beanFactory) {
4         this.beanFactory = beanFactory;
5     }
6     public void loadBeanDefinitions(Resource resource) {
7         while (resource.hasNext()) {
8             Element element = (Element) resource.next();
9             String beanID = element.attributeValue("id");
10            String beanClassName = element.attributeValue("class");
11            BeanDefinition beanDefinition = new BeanDefinition(beanID, beanClassName);
12            this.beanFactory.registerBeanDefinition(beanDefinition);
13        }
14    }
15 }
```

可以看到，在 XmlBeanDefinitionReader 中，有一个 loadBeanDefinitions 方法会把解析的 XML 内容转换成 BeanDefinition，并加载到 BeanFactory 中。

BeanFactory 功能扩展

首先，定义一个简单的 BeanFactory 实现类 SimpleBeanFactory。

 复制代码

```
1 public class SimpleBeanFactory implements BeanFactory{
2     private List<BeanDefinition> beanDefinitions = new ArrayList<>();
3     private List<String> beanNames = new ArrayList<>();
4     private Map<String, Object> singletons = new HashMap<>();
5     public SimpleBeanFactory() {
```

```

6     }
7
8     //getBean, 容器的核心方法
9     public Object getBean(String beanName) throws BeansException {
10        //先尝试直接拿Bean实例
11        Object singleton = singletons.get(beanName);
12        //如果此时还没有这个Bean的实例, 则获取它的定义来创建实例
13        if (singleton == null) {
14            int i = beanNames.indexOf(beanName);
15            if (i == -1) {
16                throw new BeansException();
17            }
18            else {
19                //获取Bean的定义
20                BeanDefinition beanDefinition = beanDefinitions.get(i);
21                try {
22                    singleton = Class.forName(beanDefinition.getClassName()).newInstance();
23                }
24                //注册Bean实例
25                singletons.put(beanDefinition.getId(), singleton);
26            }
27        }
28        return singleton;
29    }
30
31    public void registerBeanDefinition(BeanDefinition beanDefinition) {
32        this.beanDefinitions.add(beanDefinition);
33        this.beanNames.add(beanDefinition.getId());
34    }
35 }

```

由 SimpleBeanFactory 的实现不难看出, 这就是把 ClassPathXmlApplicationContext 中有关 BeanDefinition 实例化以及加载到内存中的相关内容提取出来了。提取完之后 ClassPathXmlApplicationContext 就是一个“空壳子”了, 一部分交给了 BeanFactory, 一部分又交给了 Resource 和 Reader。这时候它又该如何发挥“集成者”的功能呢? 我们看看它现在是什么样子的。

 复制代码

```

1 public class ClassPathXmlApplicationContext implements BeanFactory{
2     BeanFactory beanFactory;
3     //context负责整合容器的启动过程, 读外部配置, 解析Bean定义, 创建BeanFactory
4     public ClassPathXmlApplicationContext(String fileName) {
5         Resource resource = new ClassPathXmlResource(fileName);
6         BeanFactory beanFactory = new SimpleBeanFactory();

```

```
7         XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(beanFactory)
8         reader.loadBeanDefinitions(resource);
9         this.beanFactory = beanFactory;
10    }
11    //context再对外提供一个getBean, 底下就是调用的BeanFactory对应的方法
12    public Object getBean(String beanName) throws BeansException {
13        return this.beanFactory.getBean(beanName);
14    }
15    public void registerBeanDefinition(BeanDefinition beanDefinition) {
16        this.beanFactory.registerBeanDefinition(beanDefinition);
17    }
18 }
19
```

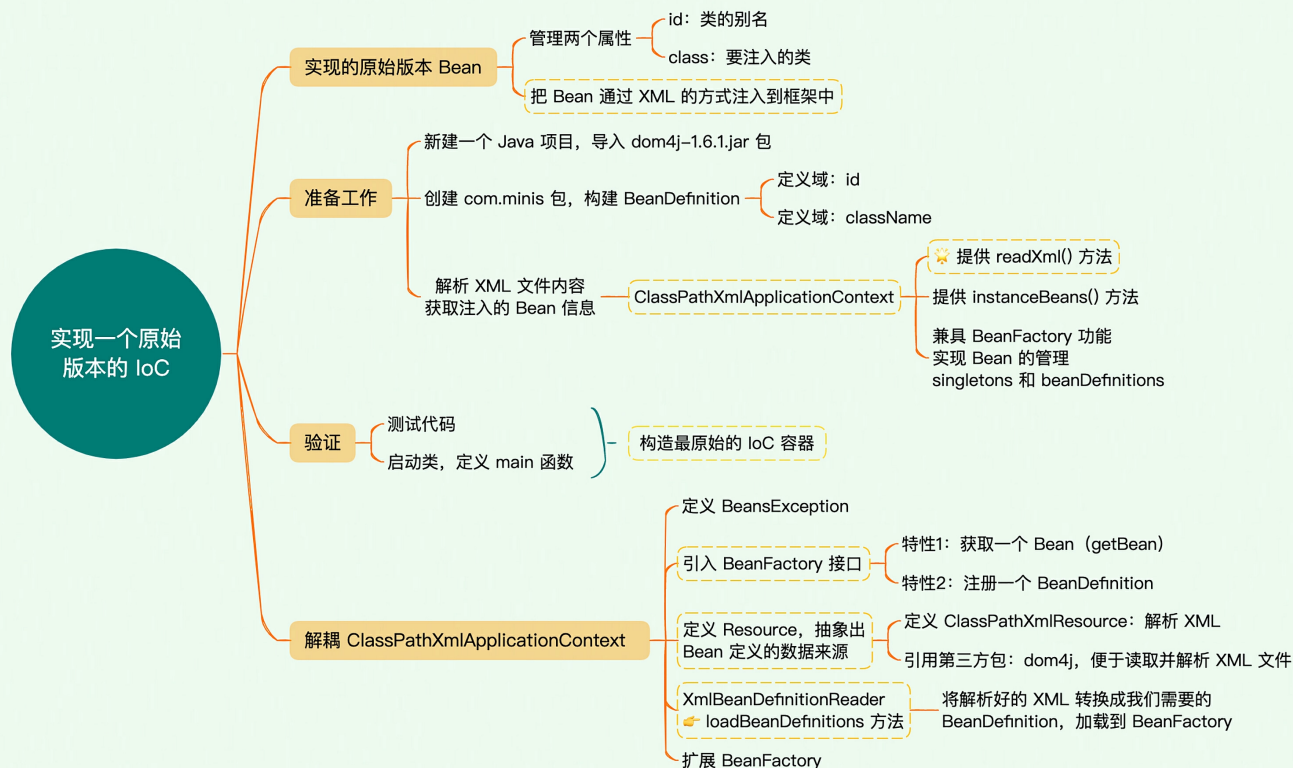
可以看到，当前的 ClassPathXmlApplicationContext 在实例化的过程中做了三件事。

1. 解析 XML 文件中的内容。
2. 加载解析的内容，构建 BeanDefinition。
3. 读取 BeanDefinition 的配置信息，实例化 Bean，然后把它注入到 BeanFactory 容器中。

通过上面几个步骤，我们把 XML 中的配置转换成 Bean 对象，并把它交由 BeanFactory 容器去管理，这些功能都实现了。虽然功能与原始版本相比没有发生任何变化，但这种**一个类只做一件事的思想**是值得我们编写代码的过程中借鉴的。

小结

好了，这节课就讲到这里。通过简简单单几个类，我们就初步构建起了 MiniSpring 的核心部分：Bean 和 IoC。



可以看到，通过这节课的构建，我们在业务程序中不需要再手动 new 一个业务类，只要把它交由框架容器去管理就可以获取我们所需的对象。另外还支持了 Resource 和 BeanFactory，用 Resource 定义 Bean 的数据来源，让 BeanFactory 负责 Bean 的容器化管理。通过功能解耦，容器的结构会更加清晰明了，我们阅读起来也更加方便。当然最重要的是，这可以方便我们今后对容器进行扩展，适配更多的场景。

以前看似高深的 Spring 核心概念之一的 IoC，就这样被我们拆解成了最简单的概念。它虽然原始，但已经具备了基本的功能，是一颗可以生长发育的种子。我们后面把其他功能一步步添加上去，这个可用的小种子就能发育成一棵大树。

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

课后题

学完这节课，我也给你留一道思考题。IoC 的字面含义是“控制反转”，那么它究竟“反转”了什么？又是怎么体现在代码中的？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 (60)



请叫我和尚

2023-03-13 来自北京

内容讲的很清晰，点赞。

这里有几个建议：

1. 每次代码设计之前，能否通过一个 UML 类图来表示，通过类图感觉更容易懂整个类与类之前的关系

2. 发现课程的内容还是有点偏向 Spring 那种感觉，就是有点类似：啊，Spring 是这样拆分几个类功能，那我们就这样拆分，这个方式感觉还是有点死板

感觉应该是：从一个设计者角度来讲，这样拆分更灵活，扩展性更强，叭叭叭等等，这样才能把读者带入进入课程，产生共鸣。

个人想法，欢迎交流，不知道是不是就我自己有这样感受，还是其他....

作者回复：好建议，我会考虑考虑。这个课程其实最早并不是教程，就是我自己手写Spring框架时的步骤，我当年用了29天时间，一天天搭建出来的。所以现在这个课程就是按照这个次序讲解的，可能是自己写的，觉得很自然。感谢你的建议。



👍 25



adelyn

2023-03-13 来自北京

请问会不会穿插讲一下用到的设计模式，单独学设计模式总是学不扎实，如果能讲到就太好了

作者回复：会穿插讲到。确实如你所说，单独学设计模式总是会隔。

共 4 条评论 >

👍 15



姐姐

2023-03-22 来自浙江

从最初的简单ApplicationContext拆解成后面的复杂ApplicationContext，我理解起来还是有困难的，努力理解如下，大神勿喷：1 readxml方法从资源文件读取内容并存入beanDefinitions，这件事情有两个地方不确定，资源的来源不同、资源的格式不同，抽象的Resource的接口，它的不同子类从不同的来源读取，但是最终都是以Resource接口的形式提供给外部访问的，这样解决了第一个不确定来源的问题；但是resource接口中被迭代的object又是根据不同格式不同而不同的，element只是xml格式的，所以又定义了BeanDefinitionReader接口，它的不同子类可以读取不同格式的资源来形成beanDefinition。2. instanceBeans方法取消了。3. getBean方法功能增强了，不仅是获得bean，对于未创建的bean还要创建bean 4 新的applicationContext负责组装，可以根据它的名字来体现它的组装功能，例如ClassPathXmlApplicationContext 它组装的Resource的实现类是ClassPathXmlResource，然后因为是xml的，所以需要BeanDefinitionReader的实现类XmlBeanDefinitionReader来读取并注册进beanFactory，同时ApplicationContext也提供了getBean底层调用beanfactory的实现，提供了registerBeanDefinition 来向底层的beanFactory注册bean。5 beanFactory 提供了registerBeanDefinition和getBean接口，这样无论是applicationContext还是beanDefinitionReader都可以向它注册beanDefinition，只要向它注册了，就可以调用它的getBean方法，我一直很纠结为什么不是beanfactory调用不同的beanDefinitionReader，写完这些，好像有点理解了，这样beanfactory就很专注自己的getBean方法，别的组件要怎么注入，它都不管了。

作者回复：你这个总结，是我见到的最详细的了，真用心。学完一遍必定大有收获。

共 3 条评论 >

👍 12



风轻扬

2023-03-15 来自北京

说一下自己对思考题的理解。

控制反转。

控制：java对象创建的控制权。

反转：将java对象创建的控制权从程序员手中反转到IOC容器手中。

另外，说一下学完这一讲的感觉。直白点说，很激动。我看过Spring这部分的源码，当时感觉挺简单的，并没有往深处想，其实忽略了“Spring为什么要这样写”的问题，现在感觉这才是源码的核心所在，突然有一点融会贯通的感觉，感觉很好。一直知道Spring的扩展性好，今天实实在在看到了。感谢老师传道解惑

作者回复：赞！用心的程序员会爬上高峰。



👍 9



未聞花名

2023-03-18 来自北京

给老师个建议，可以点一下为什么类中要放这些属性和方法，突然抽到几个新类感觉过渡有点快，这样对之后自己去设计类也能举一反三，感觉自己平时设计不太好，如果自己实现起来的话比Spring的优雅可读性要差很多。

最后附上dom4j的maven依赖，希望帮助到其他人

```
```java
```

```
<!-- https://mvnrepository.com/artifact/dom4j/dom4j -->
```

```
 <dependency>
```

```
 <groupId>dom4j</groupId>
```

```
 <artifactId>dom4j</artifactId>
```

```
 <version>1.6.1</version>
```

```
 </dependency>
```

```
```
```

作者回复: 感谢感谢!



👍 7



周润发

2023-03-15 来自浙江

首先很感谢老师的课程，内容值得细品。

想请教一个问题，为什么在代码中更多使用全局变量而不在方法中使用成员变量呢？有些全局变量只在某个方法会用到，有什么特别的考虑吗？

作者回复: 想得细致。并没有特别考虑。这个课程的特点是逐步演化，开头只是个胚胎，不断重构发育，最后一步步变成Spring的样子，所以后面的程序跟前面的都会有变化的，稍安勿躁，你一点点看到最后。我希望带给大家的是Spring这个框架的变化定型的过程，而不是一开头就讲现在的Spring是什么样子。

共 2 条评论 >

👍 5



杨松

2023-03-14 来自辽宁

老师这句话没理解呢：“我们用一个部件来对应 Bean 内存的映像”，文中5边型图片对应的其他4个都能理解

作者回复: 看的很细, 感谢。这是书面语表达的理解问题, 用我们程序员的口语来说就是“对一个Bean的定义, 需要有一个类来对应。bean的定义可能是写在外部XML文件中的, 类是运行时在内存中的, 所以表达成Bean内存的映像”

共 3 条评论 >

👍 4



C.

2023-03-24 来自江苏

ioc容器5天的版本手敲的, 对应章节的代码: <https://github.com/caozhenyuan/mini-spring.git>
点击分支查看对应章节的代码。帮大家跟敲不迷路。

作者回复: 善事。记得给出原始出处。

共 2 条评论 >

👍 3



Unknown

2023-03-15 来自福建

`getClass().getClassLoader().getResource(filepath)`

类加载器获取资源时 此处的filepath 需要放在resource目录里面(手动创建并标识类型为Resource root)

作者回复: 对的对的。你从Github上的ioc分支可以看到。

共 3 条评论 >

👍 4



马儿

2023-03-13 来自四川

以前的spring的源码都是散的, 希望能通过这个课把这些只是串联起来。希望老师讲的时候可以讲一些设计模式, 把创建类所在的包也希望能够说明一下, 这样的话更方便大家学习了解类的用途和作用。

作者回复: 完整源码参见 <https://github.com/YaleGuo/Minis>。按照编辑的意见, 在文稿中列出的代码是关键代码, 不完整。这一部分可以看Github上的ioc分支。

后面会讲到设计模式, 有好几位提到这个建议。



👍 3



浅行

2023-03-14 来自山东

有两个问题想请教一下：

1. 请问SimpleBeanFactory 为什么是在getBean时才注册实例，而不是registerBeanDefinition就先注册呢？
2. 请问ClassPathXmlApplicationContext 为什么也要实现BeanFactory呢？

作者回复：1，分开成两阶段，registerBeanDefinition仅仅只是处理bean的定义，不处理实例。

2，原始的容器时极简的，部件没有分拆。

一步步来，这个课程就是在演示怎么从一个原始的极简容器怎么逐步拆解成Spring的。这个过程的演示正是设计本课程的目的，也是程序员受益最大的做法。

共 2 条评论 >



2



Geek_b7449f

2023-03-27 来自四川

真的很详细！从零手敲，然后反反复复看几遍终于把我之前自学所学的散的知识串联起来，真的学有收获，比较期待能把设计模式也能给一起串进来呢。

本人是一名在校的学生，这里我将每一步步骤拆分为提交的方式，希望这样可以更适合所学在校的应届生

不洗勿喷哦~：<https://github.com/HHXiaoFu/mini-spring>

作者回复：善事。注明出处就可以。



1



小马哥

2023-03-26 来自北京

回答"浅行"同学的提问：

有两个问题想请教一下：

1. 请问SimpleBeanFactory 为什么是在getBean时才注册实例，而不是registerBeanDefinition就先注册呢？
2. 请问ClassPathXmlApplicationContext 为什么也要实现BeanFactory呢？

问题1：这是设计模式中的单一职责原则，对于注册BeanDefinition这个函数(registerBeanDefinition)来说，就只管注册BeanDefinition这个单一的功能，不负责实例化，所以不在其中插入实例化Bean的代码逻辑；

问题2：ClassPathXmlApplicationContext整合了读取Resource，注册Bean和获取Bean三个功能，

既然作为整合方对外提供获取bean和注册bean功能, 那需要遵守接口约束呀, 所以实现了Bean Factory.



1



小马哥

2023-03-19 来自北京

回答课后题:

- 1, 反转: 反转的是对Bean的控制权, 使用"new"的方式是由程序员在代码中主动控制; 使用IOC的方式是由容器来主动控制Bean的创建以及后面的DI属性注入;
- 2, 反转在代码中的体现: 因为容器框架并不知道未来业务中需要注入哪个Bean, 于是通过配置文件等方式告诉容器, 容器使用反射技术管理Bean的创建, 属性注入, 生命周期等.

作者回复: 赞



1



陈小远

2023-03-18 来自四川

Spring在Java编程中是事实的标准, 这个无需多言, 所以对老师的这个专栏也有比较大的兴趣。看了第一节课, 也跟着动手撸了些代码, 对有些疑惑的点做一简单阐述:

- 1、老师给的源码地址感觉应该是最终的成品的地址了, 不是按照课程一节一节的区分开来, 从学习跟随者的角度看, 可能不太友好, 无法从源码中找到项目从0到1的那种获得感;
- 2、针对本节中源码的讲述个人感觉还是有点跳跃了, 不过本身Spring体系太过庞杂, 无法面面俱到, 也无法真的能从0去反推出Spring中相关类设计的原因, 所以也表示理解。但有些设计意图还是希望老师能尽量多言一些, 而不是因为Spring包或类那样设计的就强行往Spring身上靠, 感觉有些突然。比如本节中抽象出了BeanFactory后, 为什么又突然来一个SimpleBean Factory, 而ClassPathXmlApplicationContext同样也是实现自该接口, 那通过SimpleBean Factory在组合到ClassPathXmlApplicationContext中实现功能是出于什么考虑呢? 虽然在评论区实际有看到小伙伴的讨论, 大概能明白这样设计的意思。

作者回复: 感谢你的细致, 很用心。

- 1, Github上的master分支是最终的源码, 还有十几个分支是中间阶段的, 基本对应一节一节。比你现在看的这个章节, 对应的是ioc分支。 Github上是有分支的, 大部分学员没有找到, 我很意外。
- 2, 跳跃的感觉, 好多人都有, 这也是本课程要打磨完善的地方, 感谢你的意见。具体到你的这个问题, BeanFactory只是抽象出来的一个接口, SimpleBeanFactory是一个实现, 而applicationContext从字面上就能看出它是一个上下文, 算是集大成者, 里面包含了一个factory。

共 3 条评论 >

👍 1



Ben Guo

2023-03-16 来自中国香港

谢谢老师用心的准备和讲解！关于思考题，控制反转中“反转”的是“object生成/销毁的控制对象”，从代码hardcode 指定对象的类型和创建过程，变成 从配置文件动态指定，然后IOC帮忙管理对象的生命周期。一个小小的建议，文中说了几个包名，但没有说明哪些类该归属哪个包及为什么，大家可能要参考老师的开源项目才能明白，如果这里做个简单的说明，大家可能就不用去看，且国内访问github很多时候还是麻烦一点！ 谢谢！

作者回复: 感谢感谢，我跟平台编辑去反映你的建议。我原本是这么安排的，不过编辑们根据他们的经验，建议我不用贴出完整代码，只留骨架。其实我个人更喜欢贴完整代码。



👍 1



我不懂技术

2023-03-15 来自福建

老师，有miniotomcat的开源地址吗？

作者回复: 会跟平台协商后再公布。

共 2 条评论 >

👍 1



郭英旭

2023-03-14 来自山东

期待下一章，在线等。挺急的



👍 1



C.

2023-03-14 来自江苏

真的是构造函数用到了极致。



👍 1



咕噜咕噜

2023-03-14 来自上海

控制反转：反转的是对对象的控制权，将对象的管理由业务类交给了spring容器
spring代码中体现：通过反射机制去创建一个类的对象，spring会通过Class.newInstance()或

者Constructor.newInstance()去创建一个对象。（具体可以参考org.springframework.beans.BeanUtils里面instantiate相关的方法）

作者回复: 赞



1