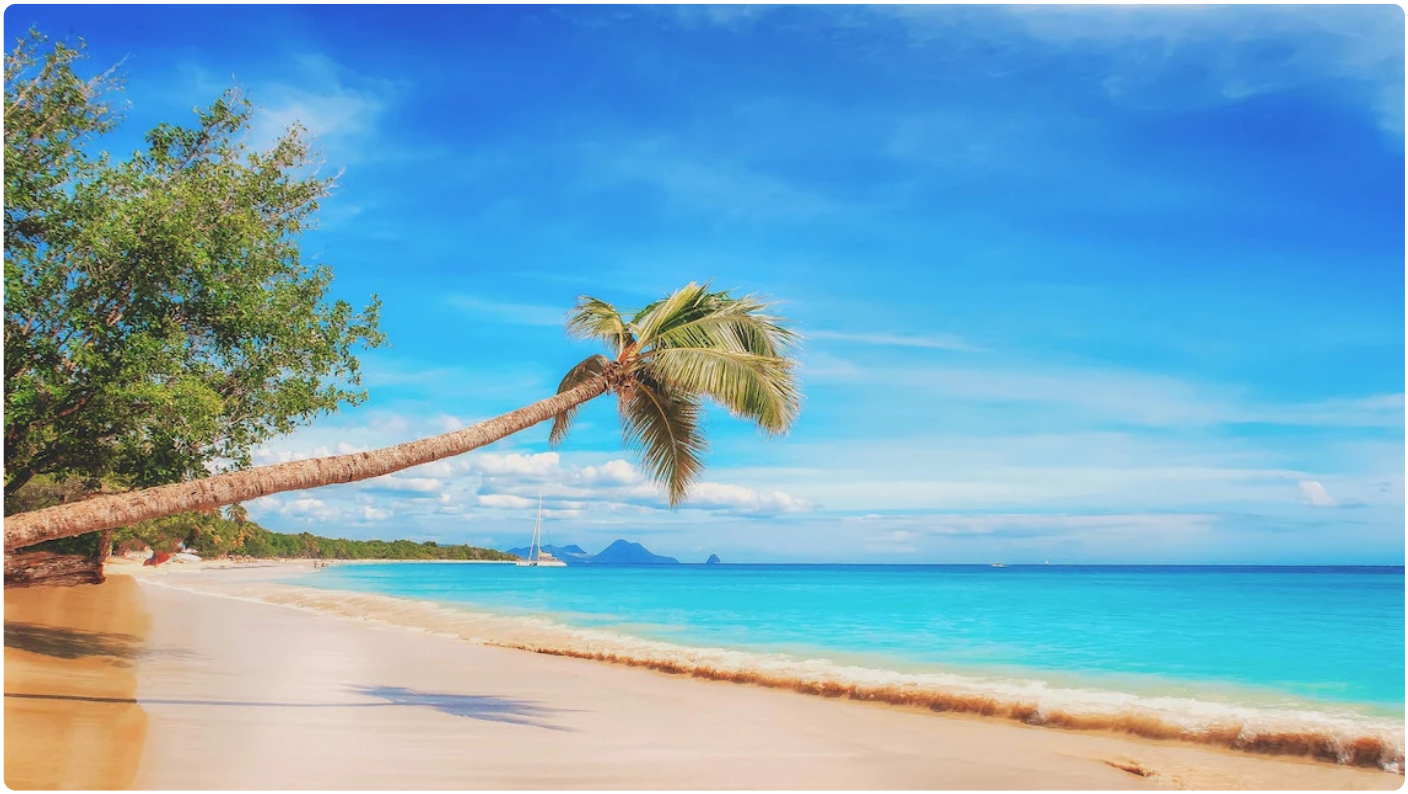
06｜再回首：如何实现一个IoC容器?

2023-03-24 郭屹 来自北京

《手把手带你写一个MiniSpring》



你好，我是郭屹。

第一阶段的学习完成啦，你是不是自己也实现出了一个简单可用的 IoC 容器呢？如果已经完成了，欢迎你把你的实现代码放到评论区，我们一起交流讨论。

我们这一章学的 IoC（Inversion of Control）是我们整个 MiniSpring 框架的基石，也是框架中最核心的一个特性，为了让你更好地掌握这节课的内容，我们对这一整章的内容做一个重点回顾。

IoC 重点回顾

IoC 是面向对象编程里的一个重要原则，目的是从程序里移出原有的控制权，把控制权交给了容器。IoC 容器是一个中心化的地方，负责管理对象，也就是 Bean 的创建、销毁、依赖注入等操作，让程序变得更加灵活、可扩展、易于维护。

在使用 IoC 容器时，我们需要先配置容器，包括注册需要管理的对象、配置对象之间的依赖关系以及对象的生命周期等。然后，IoC 容器会根据这些配置来动态地创建对象，并把它们注入到需要它们的位置上。当我们使用 IoC 容器时，需要将对象的配置信息告诉 IoC 容器，这个过程叫做依赖注入（DI），而 IoC 容器就是实现依赖注入的工具。因此，理解 IoC 容器就是理解它是如何管理对象，如何实现 DI 的过程。

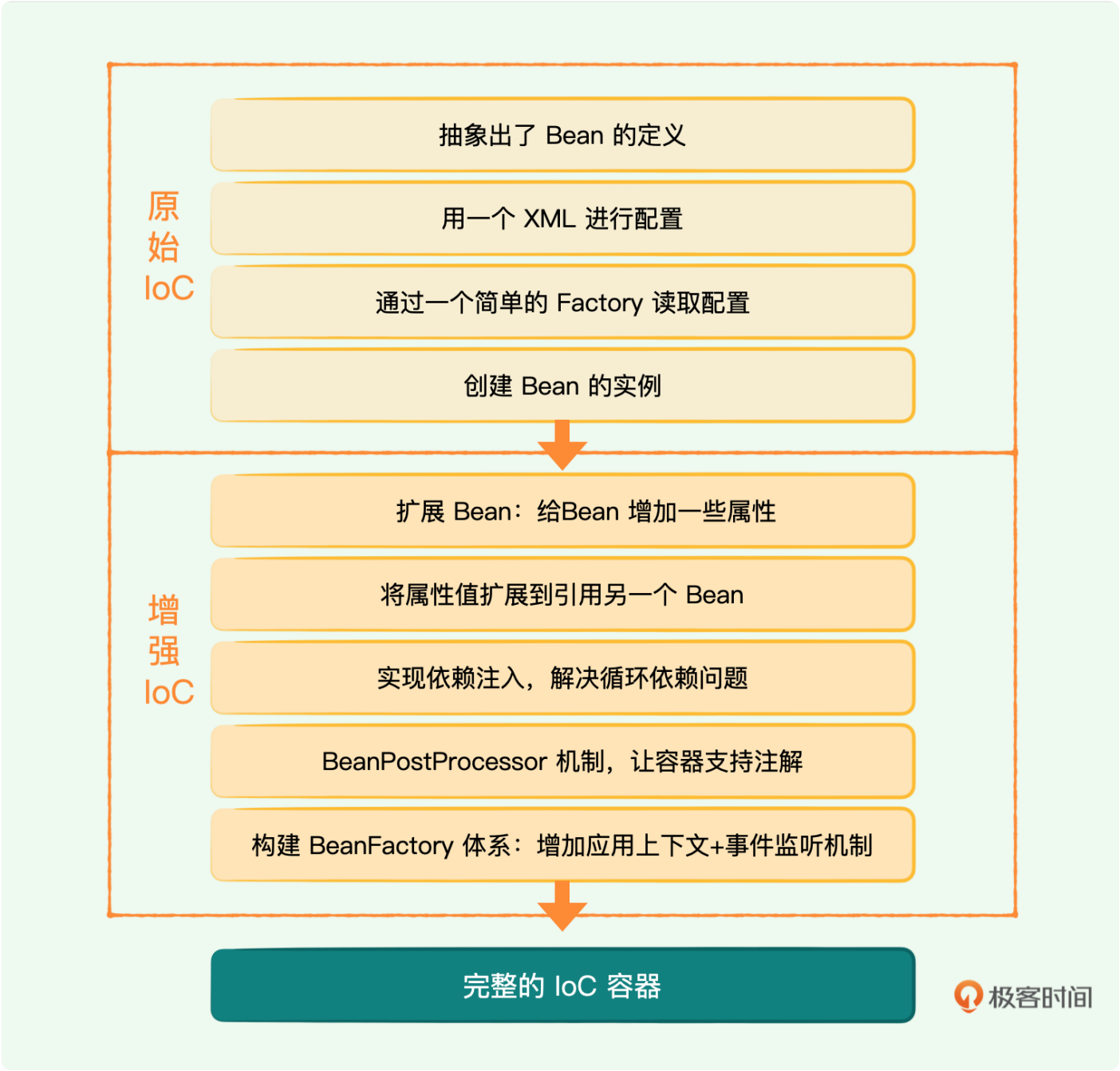
举个例子来说，我们有一个程序需要使用 A 对象，这个 A 对象依赖于一个 B 对象。我们可以把 A 对象和 B 对象的创建、配置工作都交给 IoC 容器来处理。这样，当程序需要使用 A 对象的时候，IoC 容器会自动创建 A 对象，并将依赖的 B 对象注入到 A 对象中，最后返回给程序使用。

我们在课程中是如何一步步实现 IoC 容器的呢?

我们先是抽象出了 Bean 的定义，用一个 XML 进行配置，然后通过一个简单的 Factory 读取配置，创建 bean 的实例。这个极简容器只有一两个类，但是实现了 bean 的读取，这是原始的种子。

然后再扩展 Bean，给 Bean 增加一些属性，如 constructor、property 和 init-method。此时的属性值还是普通数据类型，没有对象。然后我们将属性值扩展到引用另一个 Bean，实现依赖注入，同时解决了循环依赖问题。之后通过 BeanPostProcessor 机制让容器支持注解。

最后我们将 BeanFactory 扩展成一个体系，并增加应用上下文和容器事件侦听机制，完成一个完整的 IoC 容器。



你可以根据这些内容再好好回顾一下这个过程。另外每节课后我都留了一道思考题，你是不是认真做了呢？如果没做的话，我建议你做完再来看答案。

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

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?

参考答案

在传统的程序设计中，当需要一个对象时，我们通常使用 new 操作符手动创建一个对象，并且在创建对象时需要手动管理对象所依赖的其他对象。但是，在 IoC 控制反转中，这个过程被翻转了过来，对象的创建和依赖关系的管理被转移到了 IoC 容器中。

具体来说，在 IoC 容器中，对象的创建和依赖关系的管理大体分为两个过程。

1. 对象实例化：IoC 容器负责创建需要使用的对象实例。这意味着如果一个对象需要使用其他对象，IoC 容器会自动处理这些对象的创建，并且将它们注入到需要它们的对象中。
2. 依赖注入：IoC 容器负责把其他对象注入到需要使用这些依赖对象的对象中。这意味着我们不需要显式地在代码中声明依赖关系，IoC 容器会自动将依赖注入到对象中，从而解耦了对象之间的关系。

这些过程大大简化了对象创建和依赖关系的管理，使代码更加易于维护和扩展。下面是一个简单的 Java 代码示例，展示了 IoC 控制反转在代码中的体现。

复制代码

```
public class UserController {
    private UserService userService; // 对象不用手动创建，由容器负责创建

    public void setUserService(UserService userService) { // 不用手动管理依赖关系，由
        this.userService = userService;
    }

    public void getUser() {
        userService.getUser();
    }
}
```

在上面的示例中，UserController 依赖于 UserService，但是它并没有手动创建 UserService 对象，而是通过 IoC 容器自动注入。这种方式使得代码更加简洁，同时也简化了对象之间的依赖关系。

02 | 扩展 Bean：如何配置 constructor、property 和 init-method?

思考题

你认为通过构造器注入和通过 Setter 注入有什么异同？它们各自的优缺点是什么？

参考答案

先来说说它们之间的相同之处吧，首先它们都是为了把依赖的对象传递给类或对象，从而在运行时减少或消除对象之间的依赖关系，它们都可以用来注入复杂对象和多个依赖对象。此外 Setter 注入和构造器注入都可以用来缩小类与依赖对象之间的耦合度，让代码更加灵活、易于维护。

但同时它们之间之间也存在很多的差异。我把它们各自的优缺点整理成了一张表格放到了下面，你可以参考。

注入方式	优缺点	
构造器注入 👇 通过构造函数参数 把依赖项传递给类	优点	• 能确保类的依赖完整，因为依赖对象需要在构造器中传入，如果没有传入完整的依赖对象，类的实例化就会失败。 • 构造器注入代码结构简洁，易于理解和维护。 • 适用于生命周期较短的对象，对象在实例化时会获得所需的依赖，对象销毁时也会自动释放。
	缺点	• 如果依赖对象较多，构造函数的参数列表可能会很长，这会让代码变得复杂难懂。 • 不够灵活：使用构造器注入，类的依赖对象需要在构造器中进行声明和初始化，这会导致类与依赖对象之间的耦合度比较高，而且在运行时无法更改依赖对象。 • 如果对象的生命周期较长，构造器注入就不太适用了，因为依赖对象可能会在对象使用过程中发生变化，但是构造器注入无法满足这类需求。
Setter注入 👇 通过定义类的Setter方法 来完成对类的属性注入	优点	• 比较灵活，Setter注入能够让开发者更灵活地修改类的依赖关系。 • Setter注入使得代码易于测试，因为它可以很容易地mock或替换依赖对象，帮助开发者做单元测试。 • 代码简洁：Setter注入避免了构造函数中出现过多参数的情况，使得代码更加易于理解和维护。 • 容错性强：Setter注入使得依赖对象和被依赖对象之间的耦合度降低，即使某些依赖对象无法使用，也不会导致整个系统崩溃。
	缺点	• 为每个依赖属性定义对应的setter方法，需要使用大量的setter方法，这可能会让代码显得冗余。 • 对象状态不一致，由于Setter注入是通过调用一系列方法来完成依赖注入的，依赖对象可能被注入了一部分但未完成全部注入的情况，从而导致对象状态不一致。 • 可能会增加耦合度，通过Setter注入，类的实例可能需要访问依赖类的属性或方法，增加依赖对象之间的耦合度。

极客时间

两者之间的优劣，人们有不同的观点，存在持久的争议。Spring 团队本身的观点也在变，早期版本他们推荐使用 Setter 注入，Spring5 之后推荐使用构造器注入。当然，我们跟随 Spring 团队，现在也是建议用构造器注入。

03 | 依赖注入：如何给 Bean 注入值并解决循环依赖问题?

思考题

你认为能不能在一个 Bean 的构造器中注入另一个 Bean?

参考答案

可以在一个 Bean 的构造器中注入另一个 Bean。具体的做法就是通过构造器注入或者通过构造器注解方式注入。

方式一：构造器注入

在一个 Bean 的构造器中注入另一个 Bean，可以使用构造器注入的方式。例如：

复制代码

```
public class ABean {
    private final BBean Bbean;

    public ABean(BeanB Bbean) {
        this.Bbean = Bbean;
    }

    // ...
}

public class BBean {
    // ...
}
```

可以看到，上述代码中的 ABean 类的构造器使用了 BBean 类的实例作为参数进行构造的方式，通过这样的方式可以将 BBean 实例注入到 ABean 中。

方式二：构造器注解方式注入

在 Spring 中，我们也可以通过在 Bean 的构造器上增加注解来注入另一个 Bean，例如：

```
复制代码
1  public class ABean {
2      private final BBean Bbean;
3
4      @Autowired
5      public ABean(BBean Bbean) {
6          this.Bbean = Bbean;
7      }
8
9      // ...
10 }
11
12 public class BBean {
13     // ...
14 }
```

在上述代码中，ABean 中的构造器使用了 @Autowired 注解，这个注解可以将 BBean 注入到 ABean 中。

通过这两种方式，我们都可以在一个 Bean 的构造器中注入另一个 Bean，需要根据具体情况来选择合适的方式。通常情况下，通过构造器注入是更优的选择，可以确保依赖项的完全初始化，避免对象状态的污染。

对 MiniSpring 来讲，只需要做一点改造，在用反射调用 Constructor 的过程中处理参数的时候增加 Bean 类型的判断，然后对这个构造器参数再调用一次 getBean() 就可以了。

当然，我们要注意了。构造器注入是在 Bean 实例化过程中起作用的，一个 Bean 没有实例化完成的时候就去实例化另一个 Bean，这个时候连“早期的毛胚 Bean”都没有，因此解决不了循环依赖的问题。

04 | 增强 IoC 容器：如何让我们的 Spring 支持注解?

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?

参考答案

如果这些注解是不同作用的，那么在现有架构中是可以支持多个注解并存的。比如要给某个属性上添加一个 @Require 注解，表示这个属性不能为空，我们来看下实现的思路。

MiniSpring 中，对注解的解释是通过 BeanPostProcessor 来完成的。我们增加一个 RequireAnnotationBeanPostProcessor 类，在它的 postProcessAfterInitialization() 方法中解释这个注解，判断是不是为空，如果为空则抛出 BeanException。

然后改写 ClassPathXmlApplicationContext 类中的 registerBeanPostProcessors() 方法，将这个新定义的 beanpostprocessor 注册进去。

复制代码

```
beanFactory.addBeanPostProcessor(new
RequireAnnotationBeanPostProcessor());
```

这样，在 getBean() 方法中就会在 init-method 被调用后用到这个 RequireAnnotationBeanPostProcessor。

05 | 实现完整的 IoC 容器：构建工厂体系并添加容器事件

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？

参考答案

第二节课我们曾经提到过这个问题。这里我们来概括一下。

> 我们将 singletons 定义为了一个 ConcurrentHashMap，而且在实现 registrySingleton 时前面加了一个关键字 synchronized。这一切都是为了确保在多线程并发的情况下，我们仍然能安全地实现对单例 Bean 的管理，无论是单线程还是多线程，我们整个系统里面这个 Bean 总是唯一的、单例的。——内容来自第 2 课

在单例模式下，容器管理所有的 Bean 时，多线程环境下可能存在线程安全问题。为了避免这种问题，我们可以采取一些措施。

1. 避免共享数据

在单例模式下，所有的 Bean 都是单例的，如果 Bean 中维护了共享数据，那么就可能出现线程安全问题。为了避免共享数据带来的问题，我们可以采用一些方法来避免数据共享。例如，在 Bean 中尽量使用方法局部变量而不是成员变量，并且保证方法中不修改成员变量。

2. 使用线程安全的数据结构

在单例模式下，如果需要使用一些共享数据的数据结构，建议使用线程安全的数据结构，比如 ConcurrentHashMap 代替 HashMap，使用 CopyOnWriteArrayList 代替 ArrayList 等。这些线程安全的数据结构能够确保在多线程环境下安全地进行并发读写操作。

3. 同步

在单例模式下，如果需要操作共享数据，并且不能使用线程安全的数据结构，那么就需要使用同步机制。可以通过 synchronized 关键字来实现同步，也可以使用一些更高级的同步机制，例如 ReentrantLock、ReadWriteLock 等。

需要注意的是，使用同步机制可能会影响系统性能，并且容易出现死锁等问题，所以需要合理使用。

4. 使用 ThreadLocal

如果我们需要在多线程环境下共享某些数据，但是又想保证数据的线程安全性，可以使用 ThreadLocal 来实现。ThreadLocal 可以保证每个线程都拥有自己独立的数据副本，从而避免多个线程对同一数据进行竞争。

综上所述，在单例模式下，为了避免多线程环境下的线程安全问题，我们需要做好线程安全的设计工作，避免共享数据，选用线程安全的数据结构，正确使用同步机制，以及使用

ThreadLocal 等方法保证数据的线程安全性。

精选留言 (3)

郭硕

2023-03-24 来自浙江

老师多讲一些在SUN公司能了解到的关于Spring的内幕吗，比如出了什么特性，当时SUN内部的反应。目前老师讲的这些在网上基本都能搜到，没有比一般的博主深入太多。感觉不能体现出来老师的在SUN任职过的特殊经历，更想听听当时的历史背景和演化的过程。

作者回复: 真没什么内幕。

4

Jay

2023-03-24 来自湖北

有两个问题需要请教一下老师：
1. 构造器注入和setter注入对比中构造器优点第三条：“适用于生命周期较短的对象，对象在实例化时会获得所需要的的依赖，对象销毁时也会自动释放”。
目前我们所实现的IoC容器还没有对象销毁功能，所有的示例都还保存在容器中，那也就不会被JVM回收吧？后续是否需要实现bean的销毁功能呢？

2. 没想明白怎么实现“构造器注解autowired注入”：autowired processor是在bean实例化之后，也就是构造函数完成之后，那这个时候还如何通过autowired注入呢?这个时候构造函数要么调用完成了，要么调用失败了呀？

作者回复: 1，没有实现销毁功能。
2，实例化时通过反射调用构造函数进行的，在准备构造函数参数的时候遇到bean就再次调用getBean()就可以了。

1

peter

2023-03-25 来自山西

请问：ThreadLocal方法有数据一致性问题吗？假设两个线程共享数据A，线程1在自己的ThreadLocal中有副本A1，线程2在自己的ThreadLocal中有副本A2，那么，A1、A2和A之间会存在数据一致性问题吗？ A1、A2需要更新到A吗？

> 作者回复: ThreadLocal设计思路不是为了共享和同步而是为了隔离，多个线程隔离开的。你的例子中A1和A2是没有关系的，不存在共享A的问题。