



下载APP



导读 | 5分钟轻松了解Spring基础知识

2021-04-19 傅健

Spring编程常见错误50例

[进入课程 >](#)



讲述：傅健

时长 07:00 大小 6.42M



你好，我是傅健。

在开始我们第一章的学习之前，我想为你总结下有关 Spring 最基础的知识，这可以帮助我们后面的学习进展更加顺利一些。


就第一章来说，我们关注的是 Spring 核心功能使用中的各类错误案例。针对问题的讲解，我们大多都是直奔主题，这也是这个专栏的内容定位。所以对于**很多基础的知识**和**流程**，我们不会在解析过程中反复介绍，但它们依然是重要的，是我们解决问题的前提。借助这篇导读，我带你梳理下。



回顾 Spring 本身，什么是 Spring 最基础的知识呢？

其实就是那些 **Spring 最本质的实现和思想**。当你最开始学习的时候，你可能困惑于为什么要用 Spring，而随着对 Spring 原理的深入探究和应用，你慢慢会发现，最大的收获其实还是对于这个困惑的理解。接下来我就给你讲讲。

在进行“传统的”Java 编程时，对象与对象之间的关系都是紧密耦合的，例如服务类 Service 使用组件 ComponentA，则可能写出这样的代码：

 复制代码


```
1 public class Service {  
2     private ComponentA component = new ComponentA("first component");  
3 }
```

在没有 Spring 之前，你应该会觉得这段代码并没有多大问题，毕竟大家都这么写，而且也没有什么更好的方式。就像只有一条大路可走时，大家都朝一个方向走，你大概率不会反思是不是有捷径。

而随着项目的开发推进，你会发现检验一个方式好不好的硬性标准之一，就是看它**有没有拥抱变化的能力**。假设有一天，我们的 ComponentA 类的构造器需要更多的参数了，你会发现，上述代码到处充斥着这行需要改进的代码：

```
private ComponentA component = new ComponentA("first component");
```

此时你可能会想了，那我用下面这种方式来构造 Service 就可以了吧？


 复制代码

```
1 public class Service {  
2     private ComponentA component;  
3     public Service(ComponentA component){  
4         this.component = component;  
5     }  
6 }
```

当然不行，你忽略了一点，你在构建 Service 对象的时候，不还得使用 new 关键字来构建 Component？需要修改的调用处并不少！

很明显，这是一个噩梦。那么，除了这点，还有没有别的不好的地方呢？上面说的是非单例的情况，如果 ComponentA 本身是一个单例，会不会好些？毕竟我们可能找一个地方 new 一次 ComponentA 实例就足够了，但是你可能会发现另外一些问题。

下面是一段用“双重检验锁”实现的 ComponentA 类：

 复制代码

```
1 public class ComponentA{
2     private volatile static ComponentA INSTANCE;
3
4     private ComponentA() {}
5
6     public static ComponentA getInstance(){
7         if (INSTANCE== null) {
8             synchronized (ComponentA.class) {
9                 if (INSTANCE== null) {
10                     INSTANCE= new ComponentA();
11                 }
12             }
13         }
14         return INSTANCE;
15     }
16 }
```

其实写了这么多代码，最终我们只是要一个单例而已。而且假设我们有 ComponentB、ComponentC、ComponentD 等，那上面的重复性代码不都得写一遍？也是烦的不行，不是么？

除了上述两个典型问题，还有不易于测试、不易扩展功能（例如支持 AOP）等缺点。说白了，所有问题的根源（之一）就是**对象与对象之间耦合性太强了**。

所以 Spring 的引入，解决了上面这些零零种种的问题。那么它是怎么解决的呢？

这里套用一个租房的场景。我们为什么喜欢通过中介来租房子呢？因为省事呀，只要花点小钱就不用与房东产生直接的“纠缠”了。

Spring 就是这个思路，它就像一个“中介”公司。当你需要一个依赖的对象（房子）时，你直接把你的需求告诉 Spring（中介）就好了，它会帮你搞定这些依赖对象，按需创建它们，而无需你的任何额外操作。

不过，在 Spring 中，房东和租房者都是对象实例，只不过换了一个名字叫 Bean 而已。

可以说，通过一套稳定的生产流程，作为“中介”的 Spring 完成了生产和预装（牵线搭桥）这些 Bean 的任务。此时，你可能想了解更多。例如，如果一个 Bean（租房者）需要用到另外一个 Bean（房子）时，具体是怎么操作呢？

本质上只能从 Spring “中介” 里去找，有时候我们直接根据名称（小区名）去找，有时候则根据类型（户型），各种方式不尽相同。你就把 **Spring 理解成一个 Map 型的公司** 即可，实现如下：

[复制代码](#)

```
1 public class BeanFactory {
2
3     private Map<String, Bean> beanMap = new HashMap<>();
4
5     public Bean getBean(String key){
6         return beanMap.get(key) ;
7     }
8
9 }
```

如上述代码所示，Bean 所属公司提供了对于 Map 的操作来完成查找，找到 Bean 后装配给其它对象，这就是依赖查找、自动注入的过程。

那么回过头看，这些 Bean 又是怎么被创建的呢？

对于一个项目而言，不可避免会出现两种情况：一些对象是需要 Spring 来管理的，另外一些（例如项目中其它的类和依赖的 Jar 中的类）又不需要。所以我们得有一个办法去标识哪些是需要成为 Spring Bean，因此各式各样的注解才应运而生，例如 Component 注解等。

那有了这些注解后，谁又来做“发现”它们的工作呢？直接配置指定自然不成问题，但是很明显“自动发现”更让人省心。此时，我们往往需要一个扫描器，可以模拟写下这样一个扫描器：

[复制代码](#)

```
1 public class AnnotationScan {
```

```
2
3    //通过扫描包名来找到Bean
4    void scan(String packages) {
5        //
6    }
7
8 }
```

有了扫描器，我们就知道哪些类是需要成为 Bean。

那怎么实例化为 Bean（也就是一个对象实例而已）呢？很明显，只能通过**反射**来做了。不过这里面的方式可能有多种：

`java.lang.Class.newInstance()`

`java.lang.reflect.Constructor.newInstance()`


`ReflectionFactory.newConstructorForSerialization()`

有了创建，有了装配，一个 Bean 才能成为自己想要的样子。

而需求总是源源不断的，我们有时候想记录一个方法调用的性能，有时候我们又想在方法调用时输出统一的调用日志。诸如此类，我们肯定不想频繁再来个散弹式的修改。所以我们有了 AOP，帮忙拦截方法调用，进行功能扩展。拦截谁呢？在 Spring 中自然就是 Bean 了。

其实 AOP 并不神奇，结合刚才的 Bean（中介）公司来讲，假设我们判断出一个 Bean 需要“增强”了，我们直接让它从公司返回的时候，就使用一个代理对象作为返回不就可以了么？示例如下：

```
1 public class BeanFactory {
2
3     private Map<String, Bean> beanMap = new HashMap<>();
4
5     public Bean getBean(String key){
6         //查找是否创建过
7         Bean bean = beanMap.get(key);
8         if(bean != null){
9             return bean;
10        }
11    }
```

 复制代码


```
11         //创建一个Bean
12         Bean bean = createBean();
13         //判断要不要AOP
14         boolean needAop = judgeIfNeedAop(bean);
15         try{
16             if(needAop)
17                 //创建代理对象
18                 bean = createProxyObject(bean);
19             return bean;
20         }else:
21             return bean
22     }finally{
23         beanMap.put(key, bean);
24     }
25 }
26 }
```

那么怎么知道一个对象要不要 AOP? 既然一个对象要 AOP, 它肯定被标记了一些“规则”, 例如拦截某个类的某某方法, 示例如下:

[复制代码](#)

```
1 @Aspect
2 @Service
3 public class AopConfig {
4     @Around("execution(* com.spring.puzzle.ComponentA.execute()) ")
5     public void recordPayPerformance(ProceedingJoinPoint joinPoint) throws Thr
6         //
7     }
8 }
```

这个时候, 很明显了, 假设你的 Bean 名字是 ComponentA, 那么就应该返回 ComponentA 类型的代理对象了。至于这些规则是怎么建立起来的呢? 你看到它上面使用的各种注解大概就能明白其中的规则了, 无非就是**扫描注解, 根据注解创建规则**。

以上即为 Spring 的一些核心思想, 包括 **Bean 的构建、自动注入和 AOP**, 这中间还会掺杂无数的细节, 不过这不重要, 抓住这个核心思想对你接下来理解各种类型的错误案例才是大有裨益的!

你好, 我是傅健, 这节课我们来聊一聊 Spring Bean 的初始化过程及销毁过程中的一些问题。

虽然说 Spring 容器上手简单，可以仅仅通过学习一些有限的注解，即可达到快速使用的目的。但在工程实践中，我们依然会从中发现一些常见的错误。尤其当你对 Spring 的生命周期还没有深入了解时，类初始化及销毁过程中潜在的约定就不会很清楚。

21 人觉得很赞 | 提建议

© 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。

上一篇 开篇词 | 贴心“保姆” Spring罢工了怎么办？

下一篇 01 | Spring Bean定义常见错误

精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。