



下载APP



## 13 | 在 Spring 项目中如何进行单元测试？

2021-09-01 郑晔

《程序员的测试课》

课程介绍 >



讲述：郑晔

时长 10:33 大小 9.68M



你好，我是郑晔！

上一讲，我们将 TODO 应用从命令行扩展为了 REST 服务。在这个应用里，我们用到了 Spring 这个在 Java 世界中广泛使用的框架。大多数人对于 Spring 这个框架的认知都停留在如何使用它完成各种功能特性上，而 Spring 更大的价值其实在对于开发效率的巨大提升上，其中就包含了对测试的支持。

在接下来的两讲，我们就把注意力从一个具体的项目上挪开，放到 Spring 框架本身，看看它对开发效率提升的支持。



### 轻量级开发的 Spring

很多人对于 Spring 的理解是从依赖注入容器开始的，但是，Spring 真正对行业的影响却是从它对原有开发模式的颠覆开始。

在 21 世纪初的时候，Java 世界的主流开发方式是 J2EE，也就是 Java 的企业版。在那个时候，企业版代表软件开发的最高水准。在这个企业版的构想中，所有的复杂都应该隐藏起来，写代码的程序员不需要知道各种细节，需要的东西拿过来用就好了。

这种想法本身是没有问题的，时至今日，很多平台和框架也是这么想的。到了具体的做法上，J2EE 提供了一个应用服务器，我把这些复杂性都放在这个应用服务器里实现，你写好的程序部署到这个应用服务器上就万事大吉了。但正是因为应用服务器的存在，使用 J2EE 进行开发变成了一件无比复杂的事情。

将程序打包部署这件事说起来很简单，但在实际的工作中，如果一个团队没有做好自动化，打包部署会非常麻烦。再者，除了自己的业务代码，所有相关的第三方 JAR 包都需要打到最终的发布包中，造成的结果就是发布包非常大。在那个网络带宽还不是特别大的年代，传输这个发布包也要花很长的时间。

更关键的是，一旦出了问题怎么去定位也是个令人头疼的问题。

程序员最熟悉定位问题的方式就是调试代码。之前所有的代码都是在本地，调试起来还比较容易，现在代码运行在应用服务器上，我们必须连接到远程应用服务器上进行调试，而要连接应用服务器进行调试，还需要一些配置，总之，这件事真的是非常麻烦。

对于麻烦的事情，人们倾向于少做或不做，但是 J2EE 让这件麻烦事成了必选项。所以，那个年代的 Java 程序员处于一种痛苦不堪的状态，开发效率极其低下。

就在整个 Java 社区饱受折磨之际，Spring 横空出世。对于 J2EE 提出的问题，Spring 是承认的，但对其给出的解决方案，它却是不认的。因为应用服务器太重了，Spring 给社区带来了轻量级开发。

Spring 的逻辑很简单，这些东西通过程序库的方式就可以完成，为什么非要弄一个应用服务器呢？采用程序库的方式，最大的优势就在于可以在本地开发环境中进行开发和调试，这就极大地降低开发的难度。于是，对于同样的问题，Spring 抛弃了 J2EE 中的大部分内容，给出了自己的程序库解决方案，应用服务器变得可有可无了。

事实证明，人们更喜爱简单的解决方案，即便 J2EE 有强大的官方背书，程序员们还是义无反顾地抛弃了它。Spring 从此成了 Java 社区的主流，也成了轻量级开发的代名词。

Spring 不仅是恰当地把握了时机，占据了 Java 世界中的关键位置，更重要的是，在随后的发展中，一直凭借对于轻量级开发的追求以及良好的品位，使得它在 Java 程序员心目中占据着无可替代的位置。即便中间有部分地方其它的程序库做得稍微好一些，它也能很快地学习过来。

前面我说过，虽然 Spring 抛弃了 J2EE 中的大部分内容，基于 Web 服务器的开发还是得到了保留。因为当时确实没有什么更好的选择，虽然大部分代码可以在本地测试，但很多时候我们还是要打成一个 WAR 包部署到像 Tomcat 这样的 Web 服务器上。不过，随着 Tomcat 和一众 Web 服务器提供了可嵌入的 API，打包部署这种 J2EE 残留方式就彻底成为了过去，也就诞生今天很多 Java 程序员熟悉的 Spring Boot，可以说 Spring Boot 是 Spring 多年努力的集大成者！

## Spring 的测试

不过在 Spring Boot 出现之前，正是因为无法摆脱打包部署的这样的模式，基于这条路走下去开发难度依然不小，可以说并没有从根本上改变问题。但 Spring 的轻量级开发理念是支撑它一路向前的动力，既然那个时候 Web 服务器不能舍弃，索性 Spring 就选择了另外一条路：从测试支持入手。

所以 Spring 提供了一条测试之路，让我们在最终打包之前，能够让自己编写的代码在本地得到完整验证。你在实战环节中已经见识过如何使用 Spring 做测试了。简单来说就是**使用单元测试构建稳定的业务核心，使用 Spring 提供的基础设施进行集成测试。**

严格地说，构建稳定的业务核心其实并不依赖于 Spring，但 Spring 提供了一个将组件组装到一起基础设施，也就是依赖注入（Dependency Injection，简称 DI）容器。通常会利用 DI 容器完成我们的工作，也正是因为 DI 容器用起来很容易，所以常常会造成 **DI 容器的误用，反而会阻碍测试。**

在 [第 6 讲](#) 中，我们讨论过要编写能够组合的代码。依赖注入的风格会引导我们编写能够组合的代码，也就是不要在类的内部创建组件，而是通过依赖注入的方式将组件注入到对象之中。

所以，在一个使用 Spring 项目进行单元测试的关键就是，**保证代码可以组合的，也就是通过依赖注入的**。你可能会说，我们都用了 Spring，那代码肯定是组合的。这还真不一定，有些错误的做法就会造成对依赖注入的破坏，进而造成单元测试的困难。

## 不使用基于字段的注入

有一种典型的错误就是基于字段的注入，比如像下面这样。

[复制代码](#)

```
1 @Service
2 public class TodoItemService {
3     @Autowired
4     private TodoItemRepository repository;
5
6 }
```

@Autowired 是一个很好用的特性，它会告诉 Spring 自动帮我们注入相应的组件。在字段上加 Autowired 是一个容易写的代码，但它对单元测试却很不友好，因为你需要很繁琐地去设置这个字段的值，比如通过反射。


如果不使用基于字段的注入该怎么做呢？其实很简单，提供一个构造函数就好，把 @Autowired 放在构造函数上，像下面这样子。

[复制代码](#)

```
1 @Service
2 public class TodoItemService {
3     private final TodoItemRepository repository;
4
5     @Autowired
6     public TodoItemService(final TodoItemRepository repository) {
7         this.repository = repository;
8     }
9     ...
10 }
```

这样一来，编写测试的时候我们只要像普通对象一样去测试就好了，具体的做法你要是记不清了，可以去回顾一下实战环节。


这种构造函数一般我们都可以利用 IDE 的快捷键生成，所以这段代码对我们来说也不是很重的负担。如果你还嫌弃这种代码的冗余，也可以用 Lombok ( Lombok 是一个帮助我们生成代码的程序库 ) 的 Annotation 来简化代码，像下面这样。

 复制代码

```
1 @Service
2 @RequiredArgsConstructor
3 public class TodoItemService {
4     private final TodoItemRepository repository;
5
6     ...
7 }
```

## 不依赖于 ApplicationContext

使用 Spring 还有一种典型的错误，就是通过 ApplicationContext 获取依赖的对象，比如像下面这样。

 复制代码

```
1 @Service
2 public class TodoItemService {
3     @Autowired
4     private ApplicationContext context;
5
6     private TodoItemRepository repository;
7
8     public TodoItemService() {
9         this.repository = context.getBean(TodoItemRepository.class);
10    }
11    ...
12 }
```

我们可以把 ApplicationContext 理解成 DI 容器，原本使用 DI 容器的优点就是可以不知晓依赖是怎么产生的，而在这段代码里，却知晓了 DI 容器，这就完全打破了 DI 容器设计的初衷（关于 Spring 的设计初衷，我在《软件设计之美》中专门 [有一讲](#) 分析过，如果你有兴趣可以去了解一下）。

**在业务核心代码中出现 ApplicationContext 是一种完全错误的做法。**一方面，它打破了 DI 容器原本的设计，另一方面，还让业务核心代码对第三方代码（也就是



ApplicationContext ) 产生了依赖。

我们再从设计的角度看一下，ApplicationContext 的出现使得我们在测试这段代码时，必须引入 ApplicationContext。要想在代码里获取到相应的组件，需要在测试中向 ApplicationContext 里添加相应的组件，这会让一个原本很简单的测试变得复杂起来。

你看，一个正常的测试是如此简单，但正是因为引入了 Spring，许多人反而会做错。Spring 最大的优点是可以在代码层面上不依赖于 Spring，而错误的做法反而是深深地依赖于 Spring。

我们前面讨论了这么多，其实并没有针对 Spring 对单元测试的支持进行讲解，但 Spring 其实还真提供了一个对单元测试的支持，也就是 @MockBean，也就是帮我们进行 Mock 对象的初始化，像对于下面这行代码来说：

[复制代码](#)

```
1 @MockBean
2 private TodoItemRepository repository;
```

它就等同于下面这段。

[复制代码](#)

```
1 @BeforeEach
2 public void setUp() {
3     this.repository = mock(TodoItemRepository.class);
4     ...
5 }
```

但是我并不想特意强调这种做法。一方面，这种初始化的代码清晰且不复杂，另一方面，即便我们真的打算节省这两行的代码，更好的做法是根据你使用的 Mock 框架采用其对应的做法。比如使用 Mockito，我们可以像下面这么写。

[复制代码](#)

```
1 @ExtendWith(MockitoExtension.class)
2 public class TodoItemServiceTest {
3     @Mock
4     private TodoItemRepository repository;
5 }
```

不过 @MockBean 并非一无是处，我们在集成测试中会用到它，让它参与到依赖注入的过程中去。下一讲，我们就来讨论一下如何使用 Spring 进行集成测试。

## 总结时刻

这一讲我们讲到了 Spring 这个 Java 世界使用最广泛的框架，它最大的贡献是对开发模式的颠覆：由原来 J2EE 依赖于部署的重量级开发模式，到可以在本地开发环境完成主要工作的轻量级开发方式。

轻量级的开发方式是 Spring 一以贯之的追求，采用 Spring 开发可以在部署到容器之前就完成所有代码的验证，其中对测试的支持是非常重要的的一环。

虽然我们今天的主题是如何使用 Spring 进行单元测试，但实际上真正做好的业务测试和普通代码的测试是没有区别的，所以，我们更多地是在谈如何规避过度使用 Spring 框架犯下的错误。比如不要使用基于字段的注入，也不要依赖于 ApplicationContext 获取相应的依赖，这些做法都会让原本简单的测试变得复杂。

如果今天的内容你只能记住一件事，那请记住：**业务代码不要过度依赖于框架。**

## 思考题

今天我们的重点是错误使用了框架，你在实际的工作中，遇到过度使用框架特性，反而让代码陷入难以调整的困境吗？欢迎在留言区分享你的经验。

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 赞 2     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   12 | 实战：将 ToDo 应用扩展为一个 REST 服务

下一篇   14 | 在 Spring 项目如何进行集成测试？

## 精选留言 (3)

[写留言](#)**邓志国**

2021-09-04

构造函数不写autowire也能依赖注入

展开 ∨

作者回复: 多谢补充

**蔡奎**

2021-09-01

老师,spring 应用中依赖太多,每次启动都需要几分钟,如何保证测试。为了减少启动时间,步子都会迈大了,导致一些逻辑都不会写测试。最后就放弃测试。

展开 ∨

作者回复: 你说得对,从单元测试的角度,不依赖于Spring是最好的测试选择。依赖于Spring属于集成测试,是下一讲的内容。

**我的康康**

2021-09-01

老师,那平常开发过程中,也是不推荐用基于字段注入 而是推荐用基于构造方法注入吗?

作者回复: 是的

