第 2 章 Chapter 2

饮水思源：回顾与探索 Spring 框架的本质

SpringBoot 框架的命名关键在“Boot”上，或许 Boot Spring 更能说明这个微框架设计的初衷，也就是快速启动一个 Spring 应用！

所以，自始至终，SpringBoot 框架都是为了能够帮助使用 Spring 框架的开发者快速高效地构建一个个基于 Spring 框架以及 Spring 生态体系的应用解决方案。要深刻理解 SpringBoot 框架，首先我们需要深刻理解 Spring 框架，所以让我们先来读读历史吧！

2.1 Spring 框架的起源

虽然笔者在自己的上一本著作《Spring 揭秘》中对 Spring 框架进行了十分详尽的介绍和剖析，但这里还是要再啰嗦几句。

Spring 框架诞生于“黑暗”的 EJB 1 的时代（如果你没有听说过，恭喜你，说明你还年轻），那是一个 J2EE 规范统治的时代，基于各种容器和 J2EE 规范的软件解决方案是唯一的“正道”，沉重的研发模式和生态让那个时代的开发者痛苦不堪。随着经典巨著《Expert One-on-One J2EE Design and Development》的诞生，重规范时代终于迎来了一线曙光，该书的作者 Rod Johnson 在书中阐

述了轻量级框架的研发理念，对原有笨重的规范进行了抨击，并基于书中的理念推出了最初版的 Spring 框架，并延续至今已达 10 多年之久。

Spring 框架是构建高效 Java 研发体系的一种最佳实践，它通过一系列统一而简洁的设计，为广大 Java 开发者开拓了一条光明的 Java 应用最佳实践之路。

大家熟知的 Spring IoC 与 AOP 自不必说，Spring 更是对 Java 应用开发中常用的技术进行了合理的设计和封装，使得 Java 应用开发者可以避免昔日因 API 和系统设计不当而易犯的错误，又能够高效地完成相应问题领域的研发工作，真可说是 Java 开发必备良器！

当然，因为这不是一本专门介绍 Spring 框架的书，所以，这里不会详细展开对 Spring 框架的细节回顾。不过，一些核心的实践以及与 SpringBoot 相关的概念，还是有必要说在前的，比如 Spring IoC！

2.2 Spring IoC 其实很简单

有部分 Java 开发者对 IoC（Inversion Of Control）和 DI（Dependency Injection）的概念有些混淆，认为二者是对等的，实际上我在之前的著作中已经说过了，IoC 其实有两种方式，一种就是 DI，而另一种是 DL，即 Dependency Lookup（依赖查找），前者是当前软件实体被动接受其依赖的其他组件被 IoC 容器注入，而后者则是当前软件实体主动去某个服务注册地查找其依赖的那些服务，概念之间的关系如图 2-1 所示可能更贴切些。

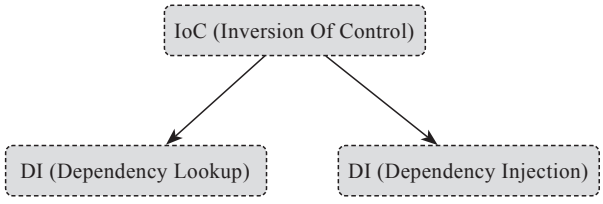

图 2-1 IoC 相关概念示意图

我们通常提到的 Spring IoC，实际上是指 Spring 框架提供的 IoC 容器实现（IoC Container），而使用 Spring IoC 容器的一个典型代码片段就是：

```
public class App {
    public static void main(String[] args) {
```

```
        ApplicationContext context = new FileSystemXmlApplication-
Context("...");
        // ...
        MockService service = context.getBean(MockService.class);
        service.doSomething();
    }
}
```

任何一个使用Spring框架构建的独立的Java应用（Standalone Java Application），通常都会存在一行类似于“context.getBean(..);”的代码，实际上，这行代码做的就是DL的工作，而构建的任何一种IoC容器背后（比如BeanFactory或者ApplicationContext）发生的事情，则更多是DI的过程（也可能有部分DL的逻辑用于对接遗留系统）。

Spring的IoC容器中发生的事情其实也很简单，总结下来即两个阶段：

（1）采摘和收集“咖啡豆”（bean）

（2）研磨和烹饪咖啡

哦，不对，这是一本技术书，差点儿写成咖啡文化杂志。

那我们还是回过头来继续说Spring IoC容器的依赖注入流程吧！ Spring IoC容器的依赖注入工作可以分为两个阶段：

阶段一：收集和注册

第一个阶段可以认为是构建和收集bean定义的阶段，在这个阶段中，我们可以通过XML或者Java代码的方式定义一些bean，然后通过手动组装或者让容器基于某些机制自动扫描的形式，将这些bean定义收集到IoC容器中。

假设我们以XML配置的形式来收集并注册单一bean，一般形式如下：

```
<bean id="mockService" class="..MockServiceImpl">
    ...
</bean>
```

如果嫌逐个收集bean定义麻烦，想批量地收集并注册到IoC容器中，我们也可以通过XML Schema形式的配置进行批量扫描并采集和注册：

```
<context:component-scan base-package="com.keevol">
```

基于JavaConfig形式的收集和注册，不管是单一还是批量，后面我们都会单独提及。

阶段二：分析和组装

当第一阶段工作完成后，我们可以先暂且认为 IoC 容器中充斥着一个个独立的 bean，它们之间没有任何关系。但实际上，它们之间是有依赖关系的，所以，IoC 容器在第二阶段要干的事情就是分析这些已经在 IoC 容器之中的 bean，然后根据它们之间的依赖关系先后组装它们。如果 IoC 容器发现某个 bean 依赖另一个 bean，它就会将这另一个 bean 注入给依赖它的那个 bean，直到所有 bean 的依赖都注入完成，所有 bean 都“整装待发”，整个 IoC 容器的工作即算完成。

至于分析和组装的依据，Spring 框架最早是通过 XML 配置文件的形式来描述 bean 与 bean 之间的关系的，随着 Java 业界研发技术和理念的转变，基于 Java 代码和 Annotation 元信息的描述方式也日渐兴盛（比如 @Autowired 和 @Inject），但不管使用哪种方式，都只是为了简化绑定逻辑描述的各种“表象”，最终都是为本阶段的最终目的服务。

很多 Java 开发者一定认为 spring 的 XML 配置文件是一种配置（Configuration），但本质上，这些配置文件更应该是一种代码形式，XML 在这里其实可以看作一种 DSL，它用来表述的是 bean 与 bean 之间的依赖绑定关系，诸君还记得没有 IoC 容器的年代要自己写代码新建（new）对象并配置（set）依赖的吧？

2.3 了解一点儿 JavaConfig

Java 5 的推出，加上当年基于纯 Java Annotation 的依赖注入框架 Guice 的出现，使得 Spring 框架及其社区也“顺应民意”，推出并持续完善了基于 Java 代码和 Annotation 元信息的依赖关系绑定描述方式，即 JavaConfig 项目。

基于 JavaConfig 方式的依赖关系绑定描述基本上映射了最早的基于 XML 的配置方式，比如：

（1）表达形式层面

基于 XML 的配置方式是这样的：

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.
xsd http://www.springframework.org/schema/context http://www.
springframework.org/schema/context/spring-context.xsd">
    <!-- bean 定义 -->
</beans>
```

而基于 JavaConfig 的配置方式是这样的：

```
@Configuration
public class MockConfiguration{
    // bean 定义
}
```

任何一个标注了 @Configuration 的 Java 类定义都是一个 JavaConfig 配置类。

（2）注册 bean 定义层面

基于 XML 的配置形式是这样的：

```
<bean id="mockService" class="..MockServiceImpl">
    ...
</bean>
```

而基于 JavaConfig 的配置形式是这样的：

```
@Configuration
public class MockConfiguration {
    @Bean
    public MockService mockService() {
        return new MockServiceImpl();
    }
}
```

任何一个标注了 @Bean 的方法，其返回值将作为一个 bean 定义注册到 Spring 的 IoC 容器，方法名将默认成为该 bean 定义的 id。

（3）表达依赖注入关系层面

为了表达 bean 与 bean 之间的依赖关系，在 XML 形式中一般是这样的：

```
<bean id="mockService" class="..MockServiceImpl">
    <property name="dependencyService" ref="dependencyService"/>
```

```
</bean>

<bean id="dependencyService" class="DependencyServiceImpl"/>
```

而在 JavaConfig 中则是这样的：

```
@Configuration
public class MockConfiguration {

    @Bean
    public MockService mockService() {
        return new MockServiceImpl(dependencyService());
    }

    @Bean
    public DependencyService dependencyService() {
        return new DependencyServiceImpl();
    }
}
```

如果一个 bean 的定义依赖其他 bean，则直接调用对应 JavaConfig 类中依赖 bean 的创建方法就可以了。

在 JavaConfig 形式的依赖注入过程中，我们使用方法调用的形式注入依赖，如果这个方法返回的对象实例只被一个 bean 依赖注入，那也还好，如果多于一个 bean 需要依赖这个方法调用返回的对象实例，那是不是意味着我们就会创建多个同一类型的对象实例？

从代码表述的逻辑来看，直觉上应该是会创建多个同一类型的对象实例，但实际上最终结果却不是这样，依赖注入的都是同一个 Singleton 的对象实例，那这是如何做到的？

笔者一开始以为 Spring 框架会通过解析 JavaConfig 的代码结构，然后通过解析器转换加上反射等方式完成这一目的，但实际上 Spring 框架的设计和实现者采用了另一种更通用的方式，这在 Spring 的参考文档中有说明，即通过拦截配置类的方法调用来避免多次初始化同一类型对象的问题，一旦拥有拦截逻辑的子类发现当前方法没有对应的类型实例时才会去请求父类的同一方法来初始化对象实例，否则直接返回之前的对象实例。

所以，原来 Spring IoC 容器中有的特性（features）在 JavaConfig 中都可以表述，只是换了一种形式而已，而且，通过声明相应的 Java Annotation 反而“内聚”一处，变得更加简洁明了了。

2.3.1　那些高曝光率的 Annotation

至于 @Configuration，我想前面已经提及过了，这里不再赘述，下面我们看几个其他比较常见的 Annotation，便于为后面更好地理解 SpringBoot 框架的奥秘做准备。

1. @ComponentScan

@ComponentScan 对应 XML 配置形式中的 <context:component-scan> 元素，用于配合一些元信息 Java Annotation，比如 @Component 和 @Repository 等，将标注了这些元信息 Annotation 的 bean 定义类批量采集到 Spring 的 IoC 容器中。

我们可以通过 basePackages 等属性来细粒度地定制 @ComponentScan 自动扫描的范围，如果不指定，则默认 Spring 框架实现会从声明 @ComponentScan 所在类的 package 进行扫描。

@ComponentScan 是 SpringBoot 框架魔法得以实现的一个关键组件，大家可以重点关注，我们后面还会遇到它。

2. @PropertySource 与 @PropertySources

@PropertySource 用于从某些地方加载 *.properties 文件内容，并将其中的属性加载到 IoC 容器中，便于填充一些 bean 定义属性的占位符（placeholder），当然，这需要 PropertySourcesPlaceholderConfigurer 的配合。

如果我们使用 Java 8 或者更高版本开发（本书写作期间 Java 9 还没发布），那么，我们可以并行声明多个 @PropertySource：

```
@Configuration
@PropertySource("classpath:1.properties")
@PropertySource("classpath:2.properties")
@PropertySource("...")
public class XConfiguration{
    ...
}
```

如果我们使用低于 Java 8 版本的 Java 开发 Spring 应用，又想声明多个 @PropertySource，则需要借助 @PropertySources 的帮助了：

```
@PropertySources({
        @PropertySource("classpath:1.properties"),
        @PropertySource("classpath:2.properties"),
        ...
})
public class XConfiguration{
    ...
}
```

3. @Import 与 @ImportResource

在 XML 形式的配置中，我们通过 <import resource="XXX.xml"/> 的形式将多个分开的容器配置合到一个配置中，在 JavaConfig 形式的配置中，我们则使用 @Import 这个 Annotation 完成同样目的：

```
@Configuration
@Import(MockConfiguration.class)
public class XConfiguration {
    ...
}
```

@Import 只负责引入 JavaConfig 形式定义的 IoC 容器配置，如果有一些遗留的配置或者遗留系统需要以 XML 形式来配置（比如 dubbo 框架），我们依然可以通过 @ImportResource 将它们一起合并到当前 JavaConfig 配置的容器中：

```
@Configuration
@Import(MockConfiguration.class)
@ImportResource("...")
public class XConfiguration {
    ...
}
```

2.4 本章小结

“磨刀不误砍柴工”，本章我们主要回顾了一下 Spring 框架的历史，并对 Spring 框架的一些核心功能和特性进行了精炼的剖析，在把我们的思维之刀磨砺快了之后，让我们开始解一下 SpringBoot 这头小牛儿吧！