第 3 章 Spring Bean

学习目的与要求

本章主要介绍 Spring Bean 的配置、实例化、作用域、生命周期以及装配方式等内容。通过本章的学习，要求读者了解 Spring Bean 的生命周期，掌握 Spring Bean 的配置、实例化、作用域以及装配方式等内容。

本章主要内容

- Bean 的配置
- Bean 的实例化
- Bean 的作用域
- Bean 的生命周期
- Bean 的装配方式

在 Spring 的应用中，使用 Spring IoC 容器可以创建、装配和配置应用组件对象，这里的组件对象称为 Bean。本章重点学习如何将 Bean 装配注入 Spring IoC 容器中。

3.1 Bean 的配置

Spring 可以看作一个大型工厂，用于生产和管理 Spring 容器中的 Bean。如果要使用这个工厂生产和管理 Bean，需要开发者将 Bean 配置到 Spring 的配置文件中。Spring 框架支持 XML 和 Properties 两种格式的配置文件，在实际开发中常用 XML 格式的配置文件。

XML 配置文件的根元素是 <beans>，在 <beans> 中包含了多个 <bean> 元素，每个 <bean> 元素定义一个 Bean，并描述 Bean 如何被装配到 Spring 容器中。<bean> 元素的常用属性及其子元素如表 3.1 所示。

表 3.1 <bean> 元素的常用属性及其子元素

属性或子元素名称	描　　述
id	Bean 在 BeanFactory 中的唯一标识，在代码中通过 BeanFactory 获取 Bean 实例时需要以此作为索引名称
class	Bean 的具体实现类，使用类名（例如 dao.TestDIDaoImpl）
scope	指定 Bean 实例的作用域，具体属性值及含义参见 3.3 节
<constructor-arg>	<bean> 元素的子元素，使用构造方法注入，指定构造方法的参数。该元素的 index 属性指定参数的序号，ref 属性指定对 BeanFactory 中其他 Bean 的引用关系，type 属性指定参数的类型，value 属性指定参数的常量值

续表

属性或子元素名称	描　　述
<property>	<bean> 元素的子元素，用于设置一个属性。该元素的 name 属性指定 Bean 实例中的相应属性名称，value 属性指定 Bean 的属性值，ref 属性指定属性对 BeanFactory 中其他 Bean 的引用关系
<list>	<property> 元素的子元素，用于封装 List 或数组类型的依赖注入，具体用法参见 3.5 节
<map>	<property> 元素的子元素，用于封装 Map 类型的依赖注入，具体用法参见 3.5 节
<set>	<property> 元素的子元素，用于封装 Set 类型的依赖注入，具体用法参见 3.5 节
<entry>	<map> 元素的子元素，用于设置一个键值对，具体用法参见 3.5 节

Bean 的配置示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- 使用 id 属性定义 myTestDIDao，其对应的实现类为 dao.TestDIDaoImpl -->
    <bean id="myTestDIDao" class="dao.TestDIDaoImpl"/>
    <!-- 使用构造方法注入 -->
    <bean id="testDIService" class="service.TestDIServiceImpl">
        <!-- 给构造方法传引用类型的参数值 myTestDIDao -->
        <constructor-arg index="0" ref="myTestDIDao"/>
    </bean>
</beans>
```

扫一扫

视频讲解

3.2　Bean 的实例化

在面向对象编程中，如果想使用某个对象，需要先实例化该对象。同样，在 Spring 框架中，如果想使用 Spring 容器中的 Bean，也需要先实例化 Bean。Spring 框架实例化 Bean 有 3 种方式，即构造方法实例化、静态工厂方法实例化和实例工厂方法实例化（其中，最常用的实例化方法是构造方法实例化）。

▶ 3.2.1　构造方法实例化

在 Spring 框架中，Spring 容器可以调用 Bean 对应类中的无参数构造方法来实例化 Bean，这种方式称为构造方法实例化。下面通过一个实例来演示构造方法实例化 Bean 的过程。

【例 3-1】 构造方法实例化 Bean。

❶ 创建模块 ch3_1

参考 1.3 节创建名为 ch3 的项目，在 ch3 项目中创建一个名为 ch3_1 的模块，同时给 ch3_1 模块添加 Web Application，并将 Spring 的 4 个基础包和 Spring Commons Logging Bridge 对应的 JAR 包 spring-jcl-6.0.0.jar 复制到 ch3_1 的 WEB-INF/lib 目录中，添加为模块

依赖。

❷ 创建 BeanClass 类

在 ch3_1 模块的 src 目录下创建 instance 包，并在该包中创建 BeanClass 类，代码如下：

```
package instance;
public class BeanClass {
    public String message;
    public BeanClass() {
        message = " 构造方法实例化 Bean";
    }
    public BeanClass(String s) {
        message = s;
    }
}
```

❸ 创建配置文件

在 ch3_1 模块的 src 目录下创建 Spring 的配置文件 applicationContext.xml，在配置文件中定义一个 id 为 constructorInstance 的 Bean，代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- 构造方法实例化 Bean -->
    <bean id="constructorInstance" class="instance.BeanClass"/>
</beans>
```

❹ 创建测试类

在 ch3_1 模块的 src 目录下创建 test 包，并在该包中创建测试类 TestInstance，代码如下：

```
package test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import instance.BeanClass;
public class TestInstance {
    private static ApplicationContext appCon;
    public static void main(String[] args) {
        appCon = new ClassPathXmlApplicationContext
            ("applicationContext.xml");
        // 测试构造方法实例化 Bean
        BeanClass b1 = (BeanClass)appCon.getBean("constructorInstance");
        System.out.println(b1+ b1.message);
    }
}
```

运行上述测试类，控制台的输出结果如图 3.1 所示。

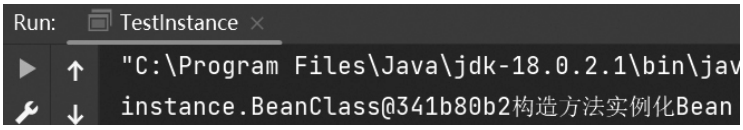

图 3.1　构造方法实例化 Bean 的运行结果

▶ 3.2.2　静态工厂方法实例化

在使用静态工厂方法实例化 Bean 时，要求开发者在工厂类中创建一个静态方法来创

建 Bean 的实例。在配置 Bean 时，需要使用 class 属性指定静态工厂类，同时还需要使用 factory-method 属性指定工厂类中的静态方法。下面通过一个实例来演示静态工厂方法实例化 Bean 的过程。

【例 3-2】 静态工厂方法实例化 Bean。

该实例在例 3-1 的基础上实现，具体过程如下。

❶ 创建工厂类 BeanStaticFactory

在 instance 包中创建工厂类 BeanStaticFactory，在该类中有一个静态方法来实例化对象，具体代码如下：

```
package instance;
public class BeanStaticFactory {
    private static BeanClass beanInstance = new BeanClass("调用静态工厂方法实例化 Bean");
    public static BeanClass createInstance() {
        return beanInstance;
    }
}
```

❷ 编辑配置文件

在配置文件 applicationContext.xml 中添加如下配置代码：

```
<!-- 静态工厂方法实例化 Bean，createInstance 为静态工厂类 BeanStaticFactory 中的静态方法 -->
<bean id="staticFactoryInstance" class="instance.BeanStaticFactory"
    factory-method="createInstance"/>
```

❸ 添加测试代码

在测试类 TestInstance 中添加如下代码：

```
// 测试静态工厂方法实例化 Bean
BeanClass b2 = (BeanClass)appCon.getBean("staticFactoryInstance");
System.out.println(b2  + b2.message);
```

此时测试类的运行结果如图 3.2 所示。

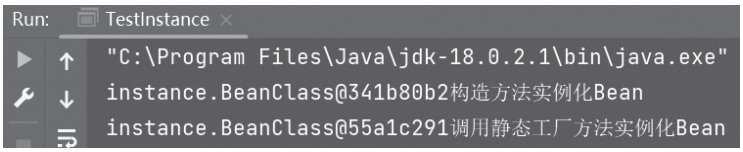

图 3.2　实例化 Bean 的运行结果

▶ 3.2.3　实例工厂方法实例化

在使用实例工厂方法实例化 Bean 时，要求开发者在工厂类中创建一个实例方法来创建 Bean 的实例。在配置 Bean 时，需要使用 factory-bean 属性指定配置的实例工厂，同时还需要使用 factory-method 属性指定实例工厂中的实例方法。下面通过一个实例来演示实例工厂方法实例化 Bean 的过程。

【例 3-3】 实例工厂方法实例化 Bean。

该实例在例 3-2 的基础上实现，具体过程如下。

❶ 创建工厂类 BeanInstanceFactory

在 instance 包中创建工厂类 BeanInstanceFactory，在该类中有一个实例工厂方法来实

例化对象，具体代码如下：

```
package instance;
public class BeanInstanceFactory {
    public BeanClass createBeanClassInstance() {
        return new BeanClass("调用实例工厂方法实例化 Bean");
    }
}
```

❷ 编辑配置文件

在配置文件 applicationContext.xml 中添加如下配置代码：

```
<!-- 配置实例工厂 -->
<bean id="myFactory" class="instance.BeanInstanceFactory"/>
<!-- 使用 factory-bean 属性指定配置工厂，
使用 factory-method 属性指定使用工厂中的哪个方法实例化 Bean -->
<bean id="instanceFactoryInstance" factory-bean="myFactory"
    factorymethod="createBeanClassInstance"/>
```

❸ 添加测试代码

在测试类 TestInstance 中添加如下代码：

```
// 测试实例工厂方法实例化 Bean
BeanClass b3 = (BeanClass)appCon.getBean("instanceFactoryInstance");
System.out.println(b3  + b3.message);
```

此时测试类的运行结果如图 3.3 所示。

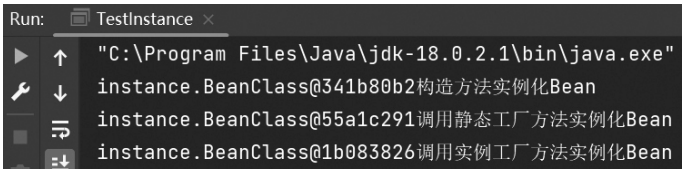

图 3.3　实例化 Bean 的运行结果

扫一扫

视频讲解

3.3　Bean 的作用域

在 Spring 中不仅可以完成 Bean 的实例化，还可以为 Bean 指定作用域。在 Spring 6.0 中为 Bean 的实例定义了如表 3.2 所示的作用域。

表 3.2　Bean 的作用域

作用域名称	描　　述
singleton	默认的作用域，使用 singleton 定义的 Bean 在 Spring 容器中只有一个 Bean 实例
prototype	Spring 容器每次获取 prototype 定义的 Bean 都将创建一个新的 Bean 实例
request	在一次 HTTP 请求中容器将返回一个 Bean 实例，不同的 HTTP 请求返回不同的 Bean 实例。该作用域仅在 Web Spring 应用程序上下文中使用
session	在一个 HTTP Session 中容器将返回同一个 Bean 实例。该作用域仅在 Web Spring 应用程序上下文中使用
application	为每个 ServletContext 对象创建一个实例，即同一个应用共享一个 Bean 实例。该作用域仅在 Web Spring 应用程序上下文中使用

续表

作用域名称	描　　述
websocket	为每个 WebSocket 对象创建一个 Bean 实例。该作用域仅在 Web Spring 应用程序上下文中使用

在表 3.2 所示的 6 种作用域中，singleton 和 prototype 是最常用的两种作用域，后面 4 种作用域仅用在 Web Spring 应用程序上下文中，在本节将会对 singleton 和 prototype 作用域进行详细的讲解。

▶ 3.3.1　singleton 作用域

当将 Bean 的 scope 设置为 singleton 时，Spring IoC 容器仅生成和管理一个 Bean 实例。在使用 id 或 name 获取 Bean 实例时，IoC 容器将返回共享的 Bean 实例。

由于 singleton 是 scope 的默认方式，所以有两种方式将 Bean 的 scope 设置为 singleton。配置文件的示例代码如下：

```
<bean id="constructorInstance" class="instance.BeanClass"/>
```

或

```
<bean id="constructorInstance" class="instance.BeanClass" scope=
    "singleton"/>
```

下面通过一个实例来测试 singleton 作用域。

【例 3-4】 测试 singleton 作用域。

该实例在例 3-3 的基础上实现，仅需要在测试类中添加如下代码：

```
BeanClass b4 = (BeanClass)appCon.getBean("instanceFactoryInstance");
System.out.println(b4  + b4.message);
```

此时测试类的代码具体如下：

```
package test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import instance.BeanClass;
public class TestInstance {
    private static ApplicationContext appCon;
    public static void main(String[] args) {
        appCon = new ClassPathXmlApplicationContext
            ("applicationContext.xml");
        // 测试构造方法实例化 Bean
        BeanClass b1 = (BeanClass)appCon.getBean("constructorInstance");
        System.out.println(b1 + b1.message);
        // 测试静态工厂方法实例化 Bean
        BeanClass b2 = (BeanClass)appCon.getBean("staticFactoryInstance");
        System.out.println(b2  + b2.message);
        // 测试实例工厂方法实例化 Bean
        BeanClass b3 = (BeanClass)appCon.getBean("instanceFactoryInstance");
        System.out.println(b3  + b3.message);
        BeanClass b4 = (BeanClass)appCon.getBean("instanceFactoryInstance");
        System.out.println(b4  + b4.message);
    }
}
```

测试类的运行结果如图 3.4 所示。

图 3.4　singleton 作用域的运行结果

从图 3.4 所示的运行结果可知，在获取多个作用域为 singleton 的同名 Bean 实例时，IoC 容器仅返回同一个 Bean 实例。

▶ 3.3.2　prototype 作用域

当将 Bean 的 scope 设置为 prototype 时，Spring IoC 容器将为每次请求创建一个新的实例。如果将 3.2.3 节中 id 为 instanceFactoryInstance 的 Bean 定义修改如下：

```
<bean id="instanceFactoryInstance" factory-bean="myFactory"
    factory-method="createBeanClassInstance" scope="prototype"/>
```

此时测试类的运行结果如图 3.5 所示。

图 3.5　prototype 作用域的运行结果

从图 3.5 所示的运行结果可知，在获取多个作用域为 prototype 的同名 Bean 实例时，IoC 容器将返回多个不同的 Bean 实例。

扫一扫

视频讲解

3.4　Bean 的生命周期

一个对象的生命周期包括创建（实例化与初始化）、使用以及销毁等阶段，在 Spring 中，Bean 对象的生命周期也遵循这一过程，但是 Spring 提供了许多对外接口，允许开发者对 3 个过程（实例化、初始化、销毁）的前后做一些操作。在 Spring 中，实例化是为 Bean 对象开辟空间，初始化则是对属性的初始化。

Spring 容器可以管理 singleton 作用域的 Bean 的生命周期，在此作用域下，Spring 能够精确地知道 Bean 何时被创建，何时初始化完成，以及何时被销毁。对于 prototype 作用域的 Bean，Spring 只负责创建，当容器创建了 Bean 的实例后，Bean 实例就交给了客户端的代码管理，Spring 容器将不再跟踪其生命周期，并且不会管理那些被配置成 prototype 作用域的 Bean。在 Spring 中 Bean 的生命周期的执行是一个很复杂的过程，用户可借鉴 Servlet 的生命周期（实例化、初始化 init、接收请求 service、销毁 destroy）理解 Bean 的生命周期。

Bean 的生命周期的整个过程如下：

（1）根据 Bean 的配置情况实例化一个 Bean。

（2）根据 Spring 上下文对实例化的 Bean 进行依赖注入，即对 Bean 的属性进行初始化。

（3）如果 Bean 实现了 BeanNameAware 接口，将调用它实现的 setBeanName (String beanId) 方法设置 Bean 的名字，此处参数传递的是 Spring 配置文件中 Bean 的 ID。

（4）如果 Bean 实现了 BeanFactoryAware 接口，将调用它实现的 setBeanFactory (BeanFactory beanFactory) 方法设置 Bean 工厂，此处参数传递的是当前 Spring 工厂实例的引用。

（5）如果 Bean 实现了 ApplicationContextAware 接口，将调用它实现的 setApplication Context(ApplicationContext ac) 方法设置应用的上下文，此处参数传递的是 Spring 上下文实例的引用。

（6）如果 Bean 关联了 BeanPostProcessor 接口，将调用预初始化方法 postProcessBefore Initialization(Object obj, String s) 对 Bean 进行初始化前的操作。

（7）如果 Bean 实现了 InitializingBean 接口，将调用 afterPropertiesSet() 方法对 Bean 进行初始化。

（8）如果 Bean 在 Spring 配置文件中配置了 init-method 属性，将自动调用其配置的初始化方法进行初始化操作。

（9）如果 Bean 关联了 BeanPostProcessor 接口，将调用 postProcessAfterInitialization (Object obj, String s) 方法，由于是在 Bean 初始化结束时调用 postProcessAfterInitialization () 方法，也可用于内存或缓存技术。

以上工作（1~9）完成后就可以使用该 Bean，由于该 Bean 的作用域是 singleton，所以调用的是同一个 Bean 实例。

（10）当 Bean 不再需要时将经过销毁阶段，如果 Bean 实现了 DisposableBean 接口，将调用其实现的 destroy () 方法将 Spring 中的 Bean 销毁。

（11）如果在配置文件中通过 destroy-method 属性指定了 Bean 的销毁方法，将调用其配置的销毁方法进行销毁。

在 Spring 中，通过实现特定的接口或通过 <bean> 元素的属性设置可以对 Bean 的生命周期过程产生影响。开发者可以随意地配置 <bean> 元素的属性，但不建议过多地使用 Bean 实现接口，因为这样将使代码和 Spring 的聚合过于紧密。下面通过一个实例演示 Bean 的生命周期。

【例 3-5】 演示 Bean 的生命周期。

❶ 创建模块 ch3_5

在 ch3 项目中创建一个名为 ch3_5 的模块，同时给 ch3_5 模块添加 Web Application，并将 Spring 的 4 个基础包和 Spring Commons Logging Bridge 对应的 JAR 包 spring-jcl-6.0.0.jar 复制到 ch3_5 的 WEB-INF/lib 目录中，添加为模块依赖。

❷ 创建 Bean 的实现类

在 ch3_5 模块的 src 目录下创建名为 life 的包，并在 life 包下创建 BeanLife 类。在 BeanLife 类中有两个方法：一个用于演示初始化过程；另一个用于演示销毁过程。其具体代码如下：

```
package life;
public class BeanLife {
```

```
    BeanLife(){
        System.out.println(" 执行构造方法，创建对象。");
    }
    public void initMyself() {
        System.out.println(this.getClass().getName() + " 执行自定义的初始化方法 ");
    }
    public void destroyMyself() {
        System.out.println(this.getClass().getName() +" 执行自定义的销毁方法 ");
    }
}
```

❸ 创建配置文件

在 ch3_5 模块的 src 目录下创建 Spring 的配置文件 applicationContext.xml，在配置文件中定义一个 id 为 beanLife 的 Bean，使用 init-method 属性指定初始化方法，使用 destroy-method 属性指定销毁方法，具体配置代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- 配置 Bean，使用 init-method 属性指定初始化方法，使用 destroy-method 属性指定
        销毁方法 -->
<bean id="beanLife" class="life.BeanLife" init-method="initMyself"
    destroy-method="destroyMyself"/>
</beans>
```

❹ 测试生命周期

在 ch3_5 模块的 src 目录下创建一个名为 test 的包，并在该包中创建测试类 TestLife，具体代码如下：

```
package test;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import life.BeanLife;
public class TestLife {
    public static void main(String[] args) {
        // 初始化 Spring 容器，加载配置文件
        // 为了方便演示销毁方法的过行，这里使用 ClassPathXmlApplicationContext
        ClassPathXmlApplicationContext ctx =
            new ClassPathXmlApplicationContext("applicationContext.xml");
        System.out.println(" 获得对象前 ");
        BeanLife blife = (BeanLife)ctx.getBean("beanLife");
        System.out.println(" 获得对象后 " + blife);
        ctx.close();     // 关闭容器，销毁 Bean 对象
    }
}
```

上述测试类运行结果如图 3.6 所示。

图 3.6　Bean 的生命周期演示效果

从图 3.6 可以看出，在加载配置文件时创建 Bean 对象，执行了 Bean 的构造方法和初始化方法 initMyself()；在获得对象后，关闭容器时，执行了 Bean 的销毁方法 destroyMyself。

扫一扫

视频讲解

3.5　Bean 的装配方式

Bean 的装配可以理解为将 Bean 依赖注入 Spring 容器中，Bean 的装配方式即 Bean 依赖注入的方式。Spring 容器支持基于 XML 配置的装配、基于注解的装配以及自动装配等多种装配方式，其中最受人们青睐的装配方式是基于注解的装配（在本书后续章节中采用基于注解的装配方式装配 Bean）。本节将主要讲解基于 XML 配置的装配和基于注解的装配。

▶ 3.5.1　基于 XML 配置的装配

基于 XML 配置的装配方式已经有很久的历史了，曾经是主要的装配方式。在通过 2.3 节的学习后，大家知道 Spring 提供了两种基于 XML 配置的装配方式，即使用构造方法注入和使用属性的 Setter 方法注入。

在使用构造方法注入方式装配 Bean 时，Bean 的实现类需要提供带参数的构造方法，并在配置文件中使用 <bean> 元素的子元素 <constructor-arg> 来定义构造方法的参数；在使用属性的 Setter 方法注入方式装配 Bean 时，Bean 的实现类需要提供一个默认无参数的构造方法，并为需要注入的属性提供对应的 Setter 方法，另外还需要使用 <bean> 元素的子元素 <property> 为每个属性注入值。

下面通过一个实例来讲解基于 XML 配置的装配方式。

【例 3-6】 基于 XML 配置的装配方式。

❶ 创建模块 ch3_6

在 ch3 项目中创建一个名为 ch3_6 的模块，同时给 ch3_6 模块添加 Web Application，并将 Spring 的 4 个基础包和 Spring Commons Logging Bridge 对应的 JAR 包 spring-jcl-6.0.0.jar 复制到 ch3_6 的 WEB-INF/lib 目录中，添加为模块依赖。

❷ 创建 Bean 的实现类

在 ch3_6 模块的 src 目录下创建名为 assemble 的包，并在该包中创建 ComplexUser 类，在 ComplexUser 类中分别使用构造方法注入和使用属性的 Setter 方法注入，具体代码如下：

```
package assemble;
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.Set;
public class ComplexUser {
    private String uname;
    private List<String> hobbyList;
    private Map<String,String> residenceMap;
    private Set<String> aliasSet;
    private String[] array;
    /*
     * 使用构造方法注入，需要提供带参数的构造方法
     */
    public ComplexUser(String uname, List<String> hobbyList, Map<String,
        String> residenceMap, Set<String> aliasSet, String[] array) {
```

```
        super();
        this.uname = uname;
        this.hobbyList = hobbyList;
        this.residenceMap = residenceMap;
        this.aliasSet = aliasSet;
        this.array = array;
    }
    /**
     * 使用属性的 Setter 方法注入，提供默认无参数的构造方法，并为注入的属性提供 Setter 方法
     */
    public ComplexUser() {
        super();
    }
    /****** 此处省略所有属性的 Setter 方法 ******/
    @Override
    public String toString() {
        return "ComplexUser [uname=" + uname + ", hobbyList=" + hobbyList +
            ", residenceMap=" + residenceMap + ", aliasSet=" + aliasSet +
            ", array=" + Arrays.toString(array) + "]";
    }
}
```

❸ 创建配置文件

在 ch3_6 模块的 src 目录下创建 Spring 的配置文件 applicationContext.xml，在配置文件中使用实现类 ComplexUser 配置 Bean 的两个实例，具体代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- 使用构造方法注入方式装配 ComplexUser 实例 user1 -->
    <bean id="user1" class="assemble.ComplexUser">
    <constructor-arg index="0" value="chenheng1"/>
    <constructor-arg index="1">
        <list>
        <value> 唱歌 </value>
        <value> 跳舞 </value>
        <value> 爬山 </value>
    </list>
    </constructor-arg>
    <constructor-arg index="2">
        <map>
            <entry key="dalian" value=" 大连 "/>
            <entry key="beijing" value=" 北京 "/>
            <entry key="shanghai" value=" 上海 "/>
        </map>
    </constructor-arg>
    <constructor-arg index="3">
        <set>
            <value> 陈恒 100</value>
            <value> 陈恒 101</value>
            <value> 陈恒 102</value>
        </set>
    </constructor-arg>
    <constructor-arg index="4">
        <array>
            <value>aaaaa</value>
            <value>bbbbb</value>
        </array>
    </constructor-arg>
    </bean>
```