

## 第 4 章

# Spring 核心

### 技能目标

- ❖ 理解 Spring IoC 的原理
- ❖ 掌握 Spring IoC 的配置
- ❖ 理解 Spring AOP 的原理
- ❖ 掌握 Spring AOP 的配置

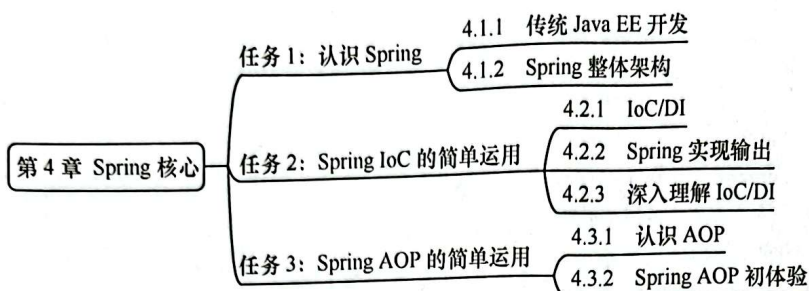
### 本章任务

学习本章，读者需要完成以下 3 个任务。记录学习过程中遇到的问题，并通过自己的努力或访问 [kgc.cn](http://kgc.cn) 解决。

任务 1：认识 Spring

任务 2：Spring IoC 的简单运用

任务 3：Spring AOP 的简单运用



## 任务 1 认识 Spring

关键步骤如下。

- 了解 Spring 的优点。
- 了解 Spring 的整体架构。

### 4.1.1 传统 Java EE 开发

在学习 Spring 之前，先了解一下企业级应用。企业级应用是指为商业组织、大型企业创建并部署的解决方案及应用。这些大型企业级应用的结构复杂，涉及的外部资源众多，事务密集、数据规模大、用户数量多，有较强的安全性考虑和较高的性能要求。

企业级应用绝不可能是一个个的独立系统。在企业中，一般都会部署多个交互的应用，同时这些应用又有可能与其他企业的相关应用连接，从而构成一个结构复杂的、跨越 Internet 的分布式企业应用集群。此外，作为企业级应用，不但要有强大的功能，还要能够满足未来业务需求的发展变化，易于扩展和维护。

传统 Java EE 在解决企业级应用问题时的“重量级”架构体系，使它的开发效率、开发难度和实际性能都令人失望。当人们苦苦寻找解决办法的时候，Spring 以一个“救世主”的形象出现在广大 Java 程序员面前。说到 Spring，就要提到 Rod Johnson，2002 年他编写了《Expert One-on-One Java EE Design and Development》一书。在书中，他对传统 Java EE 技术的日益臃肿和低效提出了质疑，他觉得应该有更便捷的做法，于是提出了 Interface 21，也就是 Spring 框架的雏形。他提出了技术应以实用为主的主张，引发了人们对“正统”Java EE 的反思。2003 年 2 月，Spring 框架正式成为一个开源项目，并发布于 SourceForge 中。



Spring 致力于 Java EE 应用的各种解决方案，而不仅仅专注于某一层的方案。可以说，Spring 是企业应用开发的“一站式”选择，贯穿表现层、业务层和持久层。并且 Spring 并不想取代那些已有的框架，而是以高度的开放性与它们无缝整合。

### 4.1.2 Spring 整体架构

Spring 确实给人一种格外清新的感觉，仿佛微雨后的绿草丛，蕴藏着勃勃生机。Spring 是一个轻量级框架，它大大简化了 Java 企业级开发，提供强大、稳定功能的同时并没有带来额外的负担。Spring 有两个主要目标：一是让现有技术更易于使用，二是养成良好的编程习惯（或者称为最佳实践）。

作为一个全面的解决方案，Spring 坚持一个原则：不重新发明轮子。已经有较好解决方案的领域，Spring 绝不做重复性的实现。例如，对象持久化和 ORM，Spring 只是对现有的 JDBC、MyBatis、Hibernate 等技术提供支持，使之更易用，而不是重新实现。

Spring 框架由大约 20 个功能模块组成。这些模块被分成六个部分，分别是 Core Container、Data Access/Integration、Web、AOP (Aspect Oriented Programming)、Instrumentation 及 Test，如图 4.1 所示。

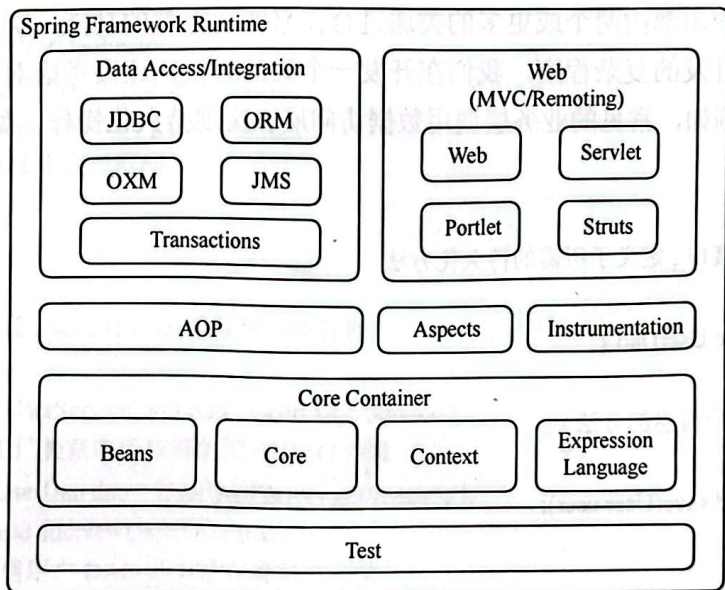


图 4.1 Spring 体系结构

Spring Core 是框架的最基础部分，提供了 IoC 特性。Spring Context 为企业级开发提供了便利的集成工具。Spring AOP 是基于 Spring Core 的符合规范的面向切面编程的实现。Spring JDBC 提供了 JDBC 的抽象层，简化了 JDBC 编码，同时使代码更健壮。Spring ORM 对市面上流行的 ORM 框架提供了支持。Spring Web 为 Spring 在 Web 应用程序中的使用提供了支持。关于 Spring 的其他功能模块在开发中的作用，可以查阅 Spring 的文档进行了解，这里不再赘述。

## 任务 2 Spring IoC 的简单运用

关键步骤如下。

- 掌握 IoC 的原理。
- 使用 IoC 的设值注入方式输出 “Hello, Spring !”。
- 使用 IoC 的设值注入方式实现动态组装的打印机。

### 4.2.1 IoC/DI

控制反转 (Inversion of Control, IoC) 也称为依赖注入 (Dependency Injection, DI), 是面向对象编程中的一种设计理念, 用来降低程序代码之间的耦合度。

依赖一般指通过局部变量、方法参数、返回值等建立的对于其他对象的调用关系。例如, 在 A 类的方法中, 实例化了 B 类的对象并调用其方法来完成特定的功能, 我们就说 A 类依赖于 B 类。

几乎所有的应用都由两个或更多的类通过合作来实现完整的功能。类与类之间的依赖关系增加了程序开发的复杂程度, 我们在开发一个类的时候, 还要考虑对正在使用该类的其他类的影响。例如, 常见的业务层调用数据访问层以实现持久化操作, 如示例 1 所示。

#### 示例 1

```
/**
 * 用户 DAO 接口, 定义了所需的持久化方法
 */
public interface UserDao {
    /**
     * 保存用户信息的方法
     */
    public void save(User user);
}

/**
 * 用户 DAO 实现类, 实现对 User 类的持久化操作
 */
public class UserDaoImpl implements UserDao {
    public void save(User user) {
        // 这里并未实现完整的数据库操作, 仅为说明问题
        System.out.println("保存用户信息到数据库");
    }
}

/**
 * 用户业务类, 实现对 User 功能的业务管理
```



IoC 的原理和定义



```

*/
public class UserServiceImpl implements UserService {
    // 实例化所依赖的 UserDao 对象
    private UserDao dao = new UserDaoImpl();
    public void addNewUser(User user) {
        // 调用 UserDao 的方法保存用户信息
        dao.save(user);
    }
}

```

如以上代码所示，UserServiceImpl 对 UserDaoImpl 存在依赖关系。这样的代码很常见，但是存在一个严重的问题，即 UserServiceImpl 和 UserDaoImpl 高度耦合，如果因为需求变化需要替换 UserDao 的实现类，将导致 UserServiceImpl 中的代码随之发生修改。如此，程序将不具备优良的可扩展性和可维护性，甚至在开发中难以测试。

我们可以利用简单工厂和工厂方法模式的思路解决此类问题，如示例 2 所示。

### 示例 2

```

/**
 * 增加用户 DAO 工厂类，负责用户 DAO 实例的创建工作
 */
public class UserDaoFactory {
    // 负责创建用户 DAO 实例的方法
    public static UserDao getInstance() {
        // 具体实现过程略
    }
}

/**
 * 用户业务类，实现对 User 功能的业务管理
 */
public class UserServiceImpl implements UserService {
    // 通过工厂类获取所依赖的用户 DAO 对象
    private UserDao dao = UserDaoFactory.getInstance();
    public void addNewUser(User user) {
        // 调用用户 DAO 的方法保存用户信息
        dao.save(user);
    }
}

```

示例 2 中的用户 DAO 工厂类 UserDaoFactory 体现了“控制反转”的思想：UserServiceImpl 不再依靠自身的代码去获得所依赖的具体 DAO 对象，而是把这一工作转交给了“第三方”UserDaoFactory，从而避免了和具体 UserDao 实现类之间的耦合。由此可见，在如何获取所依赖的对象上，“控制权”发生了“反转”，即从 UserServiceImpl 转移到了 UserDaoFactory，这就是“控制反转”。

问题虽然得到了解决，但是大量的工厂类被引入开发过程中，明显增加了开发的工

作量。而 Spring 能够分担这些额外的工作，其提供了完整的 IoC 实现，让我们得以专注于业务类和 DAO 类的设计。

## 4.2.2 Spring 实现输出



### 问题

我们已经了解了“控制反转”，那么在项目中如何使用 Spring 实现“控制反转”呢？

开发第一个 Spring 项目，输出“Hello, Spring!”。

具体要求如下。

- 编写 HelloSpring 类输出“Hello, Spring!”。
- 其中的字符串内容“Spring”是通过 Spring 框架赋值到 HelloSpring 类中的。

### 实现思路及关键代码

- (1) 下载 Spring 并添加到项目中。
- (2) 编写 Spring 配置文件。
- (3) 编写代码通过 Spring 获取 HelloSpring 实例。

程序最终运行结果如图 4.2 所示。

```
06-15 13:49:32[INFO]org.springframework.context.support.ClassPathXmlApplicationContext
-Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@e3865e:
06-15 13:49:32[INFO]org.springframework.beans.factory.xml.XmlBeanDefinitionReader
-Loading XML bean definitions from class path resource [applicationContext.xml]
06-15 13:49:33[INFO]org.springframework.beans.factory.support.DefaultListableBeanFactory
-Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory
Hello, Spring!
```

图 4.2 控制台输出“Hello, Spring!”

首先通过 Spring 官网 <http://repo.spring.io/release/org/springframework/spring/> 下载所需版本的 Spring 资源，这里以 Spring Framework 3.2.13 版本为例。下载的压缩包 spring-framework-3.2.13.RELEASE-dist.zip 解压后的文件夹目录结构如图 4.3 所示。

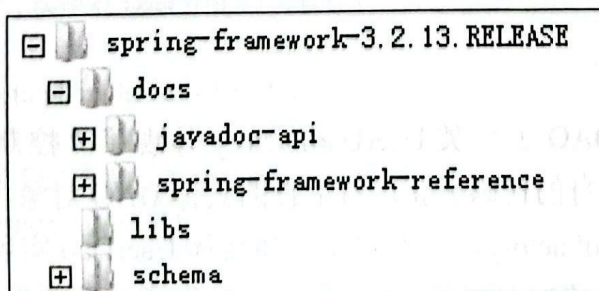


图 4.3 Spring Framework 3.2.13 目录结构

- docs: 该文件夹下包含 Spring 的相关文档，包括 API 参考文档、开发手册。



- **libs**: 该文件夹下存放 Spring 各个模块的 jar 文件, 每个模块均提供三项内容: 开发所需的 jar 文件、以“-javadoc”后缀表示的 API 和以“-sources”后缀表示的源文件。
- **schema**: 配置 Spring 的某些功能时需要用到的 schema 文件, 对于已经集成了 Spring 的 IDE 环境 (如 MyEclipse), 这些文件不需要专门导入。



### 经验

作为开源框架, Spring 提供了相关的源文件。在学习和开发过程中, 可以通过阅读源文件, 了解 Spring 的底层实现。这不仅有利于正确理解和运用 Spring 框架, 也有助于开拓思路, 提升自身的编程水平。

接下来介绍如何在 MyEclipse 中开发 HelloSpring 项目。

在 MyEclipse 中新建一个项目 HelloSpring, 将所需的 Spring 的 jar 文件添加到该项目中。需要注意的是, Spring 的运行依赖于 commons-logging 组件, 需要将相关 jar 文件一并导入。为了方便观察 Bean 实例化过程, 我们采用 log4j 作为日志输出, 所以也应该将 log4j 的 jar 文件添加到项目中。项目所需的 jar 文件如图 4.4 所示。

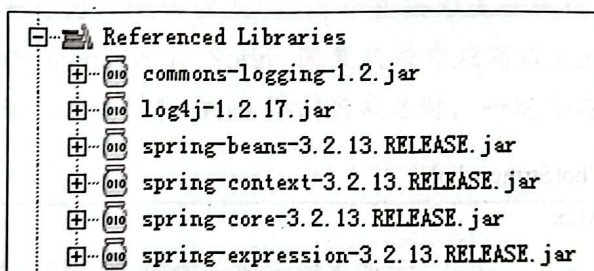


图 4.4 HelloSpring 需要的 jar 文件

为项目添加 log4j.properties 文件, 用来控制日志输出。log4j.properties 文件内容如示例 3 所示。

### 示例 3

```
# rootLogger 是所有日志的根日志, 修改该日志属性将对所有日志起作用
# 下面的属性配置中, 所有日志的输出级别是 info, 输出源是 con
log4j.rootLogger=info,con
# 定义输出源的输出位置是控制台
log4j.appender.con=org.apache.log4j.ConsoleAppender
# 定义输出日志的布局采用的类
log4j.appender.con.layout=org.apache.log4j.PatternLayout
# 定义日志输出布局
log4j.appender.con.layout.ConversionPattern=%d{MM-dd HH:mm:ss}[%p]%-5m%n
编写 HelloSpring 类, 代码如示例 4 所示。
```

## 示例 4

```

/**
 * 第一个 Spring, 输出 "Hello, Spring!"
 */
public class HelloSpring {
    // 定义 who 属性, 该属性的值将通过 Spring 框架进行设置
    private String who = null;
    /**
     * 定义打印方法, 输出一句完整的问候
     */
    public void print() {
        System.out.println("Hello," + this.getWho() + "I");
    }
    /**
     * 获得 who
     * @return who
     */
    public String getWho() {
        return who;
    }
    /**
     * 设置 who
     * @param who
     */
    public void setWho(String who) {
        this.who = who;
    }
}

```

接下来编写 Spring 配置文件, 在项目的 classpath 根路径下创建 applicationContext.xml 文件 (为便于管理框架的配置文件, 可在项目中创建专门的源文件夹, 如 resources 目录, 并将 Spring 配置文件创建在其根路径下)。在 Spring 配置文件中创建 HelloSpring 类的实例并为 who 属性注入属性值。Spring 配置文件内容如示例 5 所示。

## 示例 5

```

<?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-3.2.xsd">
    <!-- 通过 bean 元素声明需要 Spring 创建的实例。该实例的类型通过 class 属性指定,
        并通过 id 属性为该实例指定一个名称, 以便于访问 -->

```



```

<bean id="helloSpring" class="cn.springdemo.HelloSpring">
  <!-- property 元素用来为实例的属性赋值，
      此处实际是调用 setWho() 方法实现赋值操作 -->
  <property name="who">
    <!-- 此处将字符串 "Spring" 赋值给 who 属性 -->
    <value>Spring</value>
  </property>
</bean>
</beans>

```

在 Spring 配置文件中，使用 `<bean>` 元素来定义 Bean（也可称为组件）的实例。`<bean>` 元素有两个常用属性：一个是 `id`，表示定义的 Bean 实例的名称；另一个是 `class`，表示定义的 Bean 实例的类型。



### 经验

(1) 使用 `<bean>` 元素定义一个组件时，通常需要使用 `id` 属性为其指定一个用来访问的唯一名称。如果想为 Bean 指定更多的别名，可以通过 `name` 属性指定，名称之间使用逗号、分号或空格进行分隔。

(2) 在本例中，Spring 为 Bean 的属性赋值是通过调用属性的 setter 方法实现的，这种做法称为“设值注入”，而非直接为属性赋值。若属性名为 `who`，setter 方法名为 `setSomebody()`，Spring 配置文件中应写成 `name="somebody"` 而非 `name="who"`。所以在为属性和 setter 访问器命名时，一定要遵循 JavaBean 的命名规范。

在项目中添加测试方法。关键代码如下示例 6 所示。

### 示例 6

```

// 通过 ClassPathXmlApplicationContext 实例化 Spring 的上下文
ApplicationContext context = new ClassPathXmlApplicationContext(
    "applicationContext.xml");
// 通过 ApplicationContext 的 getBean() 方法，根据 id 来获取 Bean 的实例
HelloSpring helloSpring = (HelloSpring) context
    .getBean("helloSpring");
// 执行 print() 方法
helloSpring.print();

```

运行结果如图 4.2 所示。

在示例 6 中，`ApplicationContext` 是一个接口，负责读取 Spring 配置文件，管理对象的加载、生成，维护 Bean 对象之间的依赖关系，负责 Bean 的生命周期等。`ClassPathXmlApplicationContext` 是 `ApplicationContext` 接口的实现类，用于从 classpath 路径中读取 Spring 配置文件。



### 知识扩展

(1) 除了 `ClassPathXmlApplicationContext`, `ApplicationContext` 接口还有其他实现类, 例如, `FileSystemXmlApplicationContext` 也可以用于加载 Spring 配置文件, 有兴趣的读者可以查阅相关资料了解。

(2) 除了 `ApplicationContext` 及其实现类, 还可以通过 `BeanFactory` 接口及其实现类对 Bean 组件实施管理。事实上, `ApplicationContext` 就是建立在 `BeanFactory` 的基础之上。`BeanFactory` 接口是 Spring IoC 容器的核心, 负责管理组件和它们之间的依赖关系, 应用程序通过 `BeanFactory` 接口与 Spring IoC 容器交互。`ApplicationContext` 是 `BeanFactory` 的子接口, 可以对企业级开发提供更全面的支持。有兴趣的读者可以自行查阅相关资料了解 `BeanFactory` 与 `ApplicationContext` 的区别与联系。

通过“Hello.Spring!”的例子, 我们发现 Spring 会自动接管配置文件中 Bean 的创建和为属性赋值的工作。Spring 在创建 Bean 的实例后, 会调用相应的 setter 方法为实例设置属性值。实例的属性值将不再由程序中的代码来主动创建和管理, 改为被动接受 Spring 的注入, 使得组件之间以配置文件而不是硬编码的方式组织在一起。



### 提示

相对于“控制反转”, “依赖注入”的说法也许更容易理解一些, 即由容器 (如 Spring) 负责把组件所“依赖”的具体对象“注入” (赋值) 给组件, 从而避免组件之间以硬编码的方式耦合在一起。

### 技能训练

#### 上机练习 1——在控制台输出

##### 训练要点

- 使用 Spring 实现依赖注入。

##### 需求说明

- 输出:

张嘎说: “三天不打小鬼子, 手都痒痒!”

Rod 说: “世界上有 10 种人, 认识二进制的和不认识二进制的。”

- 说话人和说话内容都通过 Spring 注入。



## 实现思路及关键代码

- (1) 将 Spring 添加到项目。
- (2) 编写程序代码和配置文件（同时配置张嘎和 Rod 两个 Bean）。
- (3) 获取 Bean 实例，调用功能方法。

### 参考解决方案

程序代码如下。

```
/**
 * 依赖注入范例 :Greeting
 */
public class Greeting {
    // 说话的人
    private String person = "Nobody";
    // 说话的内容
    private String words = "nothing";
    // 省略 setter、getter 方法
    .....

    /**
     * 定义说话方法
     */
    public void sayGreeting() {
        System.out.println(person + " 说 : " + words + " ");
    }
}
```

Spring 配置文件内容如下。

```
<?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-3.2.xsd">
    <!-- 定义 bean, 该 bean 的 id 是 zhangGa(张嘎), class 指定该 bean 实例的实现类 -->
    <bean id="zhangGa" class="cn.service.Greeting">
        <!-- property 元素用来指定需要容器注入的属性, person 属性需要容器注入,
            Greeting 类必须拥有 setPerson() 方法 -->
        <property name="person">
            <!-- 为 person 属性注入值 -->
            <value> 张嘎 </value>
        </property>
        <!-- words 属性需要容器注入, Greeting 类必须拥有 setWords() 方法 -->
        <property name="words">
            <!-- 为 words 属性注入值 -->
```

```

        <value> 三天不打小鬼子,手都痒痒!</value>
    </property>
</bean>
<bean id="rod" class="cn.service.Greeting">
    <property name="person">
        <value>Rod</value>
    </property>
    <property name="words">
        <value>世界上有 10 种人,认识二进制的和不认识二进制的。</value>
    </property>
</bean>
</beans>

```

测试方法的关键代码如下。

```

// 通过 ClassPathXmlApplicationContext 显式地实例化 Spring 的上下文
ApplicationContext context = new ClassPathXmlApplicationContext(
    "applicationContext.xml");
// 通过 id 获取 Greeting Bean 的实例
Greeting zhangGa = (Greeting) context.getBean("zhangGa");
Greeting rod = (Greeting) context.getBean("rod");
// 执行 sayGreeting() 方法
zhangGa.sayGreeting();
rod.sayGreeting();

```

### 4.2.3 深入理解 IoC/DI

通过 HelloSpring 示例的学习基本了解了 Spring 的配置及 Spring 的依赖注入,接下来开发一个打印机程序,以便更深入地理解 Spring 的“依赖注入”。



#### 问题

如何开发一个打印机模拟程序,使其符合以下条件。

- 可以灵活地配置使用彩色墨盒或灰色墨盒。
- 可以灵活地配置打印页面的大小。



#### 分析

程序中包括打印机(Printer)、墨盒(Ink)和纸张(Paper)三类组件,如图 4.5 所示。打印机依赖墨盒和纸张。







采取如下的步骤开发这个程序。

- (1) 定义 Ink 和 Paper 接口。
- (2) 使用 Ink 接口和 Paper 接口开发 Printer 程序。在开发 Printer 程序时并不依赖 Ink 和 Paper 的具体实现类。
- (3) 开发 Ink 接口和 Paper 接口的实现类：ColorInk、GreyInk 和 TextPaper。
- (4) 组装打印机，运行调试。

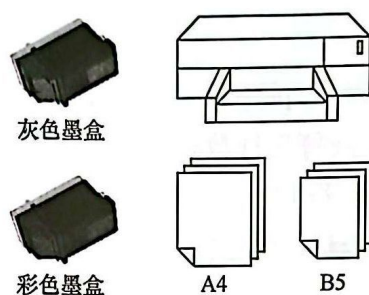


图 4.5 打印机的组件

## 1. 定义 Ink 和 Paper 接口

### 示例 7

```
/**
 * 墨盒接口
 */
public interface Ink {
    /**
     * 定义打印采用的颜色的方法
     * @param r 红色值
     * @param g 绿色值
     * @param b 蓝色值
     * @return 返回打印采用的颜色
     */
    public String getColor(int r, int g, int b);
}
```

Ink 接口只定义一个 `getColor()` 方法，传入红、绿、蓝三种颜色的值，表示逻辑颜色；返回一个形如 `#ffc800` 的颜色字符串，表示打印采用的颜色。

### 示例 8

```
/**
 * 纸张接口
 */
```

```

public interface Paper {
    public static final String newline = "\r\n";
    /**
     * 输出一个字符到纸张
     */
    public void putInChar(char c);
    /**
     * 得到输出到纸张上的内容
     */
    public String getContent();
}

```

Paper 接口中定义了两个方法：putInChar() 用于向纸张中输出一个字符，向纸张输出字符后，纸张会根据自身大小（每页行数和每行字数的限制）在输入流中插入换行符、分页符和页码；getContent() 用于得到纸张中的所有内容。

## 2. 使用 Ink 接口和 Paper 接口开发 Printer 程序

### 示例 9

```

/**
 * 打印机程序
 */
public class Printer {
    // 面向接口编程，而不是具体的实现类
    private Ink ink = null;
    private Paper paper = null;
    /**
     * 设置注入所需的 setter 方法
     * @param ink 传入墨盒参数
     */
    public void setInk(Ink ink) {
        this.ink = ink;
    }
    /**
     * 设置注入所需的 setter 方法
     * @param paper 传入纸张参数
     */
    public void setPaper(Paper paper) {
        this.paper = paper;
    }
    /**
     * 打印机打印方法
     * @param str 传入打印内容
     */
    public void print(String str){

```



```

// 输出颜色标记
System.out.println(" 使用 "+
    ink.getColor(255, 200, 0)+" 颜色打印 :\n");
// 逐字符输出到纸张
for(int i=0;i<str.length();++i){
    paper.putInChar(str.charAt(i));
}
// 将纸张的内容输出
System.out.print(paper.getContent());
}
}

```

Printer 类中只有一个 print() 方法，输入参数是一个即将被打印的字符串，打印机将这个字符串逐个字符输出到纸张，然后将纸张中的内容输出。

在开发 Printer 程序的时候，只需要了解 Ink 接口和 Paper 接口即可，完全不需要依赖这些接口的某个具体实现类，这是符合实际情况的。在设计真实的打印机时也是这样，设计师只是针对纸张和墨盒的接口规范进行设计。在使用时，只要符合相应的规范，打印机就可以根据需求更换不同的墨盒和纸张。

软件设计与此类似，由于明确地定义了接口，在编写代码的时候，完全不用考虑和某个具体实现类的依赖关系，从而可以构建更复杂的系统。组件间的依赖关系和接口的重要性在将各个组件组装在一起的时候得以体现。通过这种开发模式，还可以根据需要方便地更换接口的实现，就像为打印机更换不同的墨盒和纸张一样。Spring 提倡面向接口编程也是基于这样的考虑。

print() 方法运行的时候是从哪里获得 Ink 和 Paper 的实例呢？

这时就需要提供“插槽”，以便组装的时候可以将 Ink 和 Paper 的实例“注入”进来，对 Java 代码来说就是定义 setter 方法。至此，Printer 类的开发工作就完成了。

### 3. 开发 Ink 接口和 Paper 接口的实现类：ColorInk、GreyInk 和 TextPaper

#### 示例 10

```

/**
 * 彩色墨盒。ColorInk 实现 Ink 接口
 */
public class ColorInk implements Ink {
    // 打印采用彩色
    public String getColor(int r, int g, int b) {
        Color color = new Color(r,g,b);
        return "#" + Integer.toHexString(color.getRGB()).substring(2);
    }
}

```

#### 示例 11

```
/**
```

```

* 灰色墨盒，GreyInk 实现 Ink 接口
*/
public class GreyInk implements Ink {
    // 打印采用灰色
    public String getColor(int r, int g, int b) {
        int c = (r+g+b)/3;
        Color color = new Color(c,c,c);
        return "#" + Integer.toHexString(color.getRGB()).substring(2);
    }
}

```

彩色墨盒的 `getColor()` 方法对传入的颜色参数做了简单的格式转换；灰色墨盒则对传入的颜色值进行计算，先转换成灰度颜色，再进行格式转换。这不是需要关注的重点，了解其功能即可。

### 示例 12

```

/**
 * 文本打印纸张实现。TextPaper 实现 Paper 接口
 */
public class TextPaper implements Paper {
    // 每行字符数
    private int charPerLine = 16;
    // 每页行数
    private int linePerPage = 5;
    // 纸张中内容
    private String content = "";
    // 当前横向位置，从 0 到 charPerLine-1
    private int posX = 0;
    // 当前行数，从 0 到 linePerPage-1
    private int posY = 0;
    // 当前页数
    private int posP = 1;

    public String getContent() {
        String ret = this.content;
        // 补齐本页空行，并显示页码
        if (!(posX==0 && posY==0)){
            int count = linePerPage - posY;
            for (int i=0;i<count;++i){
                ret += Paper.newline;
            }
            ret += "== 第 " + posP + " 页 ==";
        }
        return ret;
    }
}

```



```

public void putInChar(char c) {
    content += c;
    ++posX;
    // 判断是否换行
    if (posX==charPerLine){
        content += Paper.newline;
        posX = 0;
        ++posY;
    }
    // 判断是否翻页
    if (posY==linePerPage){
        content += "== 第 " + posP + " 页 ==";
        content += Paper.newline + Paper.newline;
        posY = 0;
        ++posP;
    }
}

// setter 方法, 用于注入每行的字符数
public void setCharPerLine(int charPerLine) {
    this.charPerLine = charPerLine;
}

// setter 方法, 用于注入每页的行数
public void setLinePerPage(int linePerPage) {
    this.linePerPage = linePerPage;
}
}

```

在 TextPaper 实现类的代码中, 我们不用关心具体的逻辑实现, 只需理解其功能即可。其中 content 用于保存当前纸张的内容。charPerLine 和 linePerPage 用于限定每行可以打印多少个字符和每页可以打印多少行。需要注意的是, setCharPerLine() 和 setLinePerPage() 这两个 setter 方法, 与示例 9 中的 setter 方法类似, 也是为了组装时“注入”数据留下的“插槽”。我们不仅可以注入某个类的实例, 还可以注入基本数据类型、字符串等类型的数据。

#### 4. 组装打印机, 运行调试

组装打印机的工作在 Spring 的配置文件 (applicationContext.xml) 中完成。首先, 创建几个待组装零件的实例, 如示例 13 所示。

##### 示例 13

```

<!-- 定义彩色墨盒 Bean, id 是 colorInk -->
<bean id="colorInk" class="cn.ink.ColorInk" />
<!-- 定义灰色墨盒 Bean, id 是 greyInk -->
<bean id="greyInk" class="cn.ink.GreyInk" />
<!-- 定义 A4 纸张 Bean, id 是 a4Paper -->

```

```

<!-- 通过 setCharPerLine() 方法为 charPerLine 属性注入每行字符数 -->
<!-- 通过 setLinePerPage() 方法为 linePerPage 属性注入每页行数 -->
<bean id="a4Paper" class="cn.paper.TextPaper">
    <property name="charPerLine" value="10" />
    <property name="linePerPage" value="8" />
</bean>
<!-- 定义 B5 纸张 Bean,id 是 b5Paper -->
<bean id="b5Paper" class="cn.paper.TextPaper">
    <property name="charPerLine" value="6" />
    <property name="linePerPage" value="5" />
</bean>

```

各“零件”都定义好后，下面来完成打印机的组装，如示例 14 所示。

#### 示例 14

```

<!-- 组装打印机。定义打印机 Bean, 该 Bean 的 id 是 printer,class 指定该 Bean 实例的实现类 -->
<bean id="printer" class="cn.printer.Printer">
    <!-- 通过 ref 属性注入已经定义好的 bean -->
    <!-- 注入彩色墨盒 -->
    <property name="ink" ref="colorInk" />
    <!-- 注入 B5 打印纸张 -->
    <property name="paper" ref="b5Paper" />
</bean>

```

示例 14 的代码组装了一台彩色的、使用 B5 打印纸的打印机。需要注意的是，这里没有使用 <property> 的 value 属性，而是使用了 ref 属性。value 属性用于注入基本数据类型以及字符串类型的值。ref 属性用于注入已经定义好的 Bean，如刚刚定义好的 colorInk、greyInk、a4Paper 和 b5Paper。由于 Printer 的 setInk(Ink ink) 方法要求传入的参数是 Ink(接口) 类型，所以任何 Ink 接口的实现类都可以注入。

完整的 Spring 配置文件代码如示例 15 所示。

#### 示例 15

```

<?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-3.2.xsd">
    <!-- 定义墨盒 -->
    <bean id="colorInk" class="cn.ink.ColorInk" />
    <bean id="greyInk" class="cn.ink.GreyInk" />
    <!-- 定义纸张 -->
    <bean id="a4Paper" class="cn.paper.TextPaper">
        <property name="charPerLine" value="10" />
        <property name="linePerPage" value="8" />
    </bean>
    <bean id="b5Paper" class="cn.paper.TextPaper">

```



```

    <property name="charPerLine" value="6" />
    <property name="linePerPage" value="5" />
</bean>
<!-- 组装打印机 -->
<bean id="printer" class="cn.printer.Printer">
    <property name="ink" ref="colorInk" />
    <property name="paper" ref="b5Paper" />
</bean>
</beans>

```

从配置文件中可以看到 Spring 管理 Bean 的灵活性。Bean 与 Bean 之间的依赖关系放在配置文件里组织，而不是写在代码里。通过对配置文件的指定，Spring 能够精确地为每个 Bean 注入属性。

每个 Bean 的 id 属性是该 Bean 的唯一标识。程序通过 id 属性访问 Bean，Bean 与 Bean 的依赖关系也通过 id 属性完成。

打印机组装好之后如何工作呢？测试方法的关键代码如下例 16 所示。

#### 示例 16

```

/**
 * 测试打印机
 */
ApplicationContext context = new ClassPathXmlApplicationContext(
    "applicationContext.xml");
// 通过 Printer bean 的 id 来获取 Printer 实例
Printer printer = (Printer) context.getBean("printer");
String content = "几位轻量级容器的作者曾骄傲地对我说：这些容器非常有 " +
    "用，因为它们实现了“控制反转”。这样的说辞让我深感迷惑：控 " +
    "制反转是框架所共有的特征，如果仅仅因为使用了控制反转就认为 " +
    "这些轻量级容器与众不同，就好像在说“我的轿车是与众不同的， " +
    "因为它有 4 个轮子。”";
printer.print(content);

```

运行结果如图 4.6 所示。

至此，打印机已经全部组装完成并可以正常使用了。现在我们来总结一下：和 Spring 有关的只有组装和运行两部分代码。仅这两部分代码就让我们获得了像更换打印机的墨盒和打印纸一样更换程序组件的能力。这就是 Spring 依赖注入的魔力。

通过 Spring 的强大组装能力，我们在开发每个程序组件的时候，只要明确关联组件的接口定义，而不需要关心具体实现，这就是所谓的“面向接口编程”。

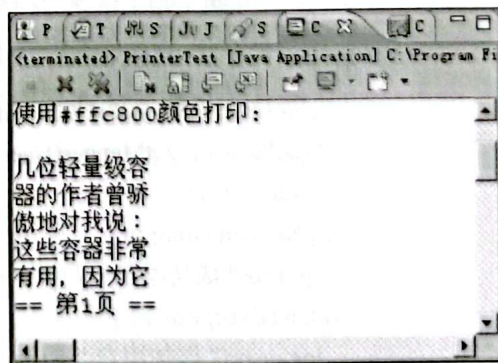


图 4.6 打印机打印的结果

## 技能训练

## 上机练习 2——模仿实现打印机功能

## 需求说明

模仿示例 7 至示例 16 的内容，自己动手实现打印机功能，并使用 Spring IoC 实现墨盒和纸张的灵活替换。

## 任务 3 Spring AOP 的简单运用

关键步骤如下。

- 掌握 Spring AOP 的原理。
- 使用 Spring AOP 实现自动的系统日志功能。

## 4.3.1 认识 AOP

面向切面编程（Aspect Oriented Programming, AOP）是软件编程思想发展到一定阶段的产物，是对面向对象编程（Object Oriented Programming, OOP）的有益补充。AOP 一般适用于具有横切逻辑的场合，如访问控制、事务管理、性能监测等。

什么是横切逻辑呢？我们先来看下面这段程序代码。

```
/**
 * 用户业务类，实现对 User 功能的业务管理
 */
public class UserServiceImpl implements UserService {
    private static final Logger log = Logger.getLogger(UserServiceImpl.class);

    public boolean addNewUser(User user) {
        log.info("添加用户" + user.getUsername());
        SqlSession sqlSession = null;
        boolean flag = false;
        try {
            sqlSession = MyBatisUtil.createSqlSession();
            if (sqlSession.getMapper(UserMapper.class).add(user) > 0)
                flag = true;
            sqlSession.commit();
            log.info("成功添加用户" + user.getUsername());
        } catch (Exception e) {
            log.error("添加用户" + user.getUsername() + "失败", e);
            sqlSession.rollback();
            flag = false;
        } finally {
```



AOP 的定义和原理



```

        MyBatisUtil.closeSqlSession(sqlSession);
    }
    return flag;
}
}

```

在该段代码中，UserService 的 addNewUser() 方法根据需求增加了日志和事务功能。

这是一个再典型不过的业务处理方法。日志、异常处理、事务控制等，都是一个健壮的业务系统所必需的。但为了保证系统健壮可用，就要在众多的业务方法中反复编写类似的代码，使得原本就很复杂的业务处理代码变得更加复杂。业务功能的开发者还要关注这些“额外”的代码是否处理正确，是否有遗漏。如果需要修改日志信息的格式或者安全验证的规则，或者再增加新的辅助功能，都会导致业务代码频繁而大量的修改。

在业务系统中，总有一些散落、渗透到系统各处且不得不处理的事情，这些穿插在既定业务中的操作就是所谓的“横切逻辑”，也称为切面。怎样才能不受这些附加要求的干扰，专注于真正的业务逻辑呢？我们很容易想到的就是将这些重复性的代码抽取出来，放在专门的类和方法中处理，这样就便于管理和维护了。但即便如此，依然无法实现既定业务和横切逻辑的彻底解耦合，因为业务代码中还要保留对这些方法的调用代码，当需要增加或减少横切逻辑的时候，还是要修改业务方法中的调用代码才能实现。我们希望的是无须编写显式的调用，在需要的时候，系统能够“自动”调用所需的功能，这正是 AOP 要解决的主要问题。

面向切面编程，简单地说就是在不改变原有程序的基础上为代码段增加新的功能，对其进行增强处理。它的设计思想来源于代理设计模式，下面以图示的方式进行简单的说明。通常情况下调用对象的方法如图 4.7 所示。

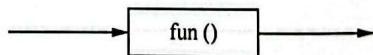


图 4.7 直接调用对象的方法

在代理模式中可以为对象设置一个代理对象，代理对象为 fun() 提供一个代理方法，当通过代理对象的 fun() 方法调用原对象的 fun() 方法时，就可以在代理方法中添加新的功能，这就是所谓的增强处理。增强的功能既可以插到原对象的 fun() 方法前面，也可以插到其后面，如图 4.8 所示。

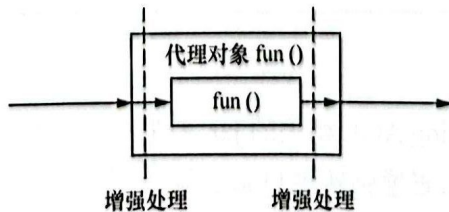


图 4.8 通过代理对象调用方法

在这种模式下,给编程人员的感觉就是在原有代码乃至原业务流程都不改变的情况下,直接在业务流程中切入新代码,增加新功能,这就是所谓的面向切面编程。对面向切面编程有了感性认识以后,还需要了解它的一些基本概念。

- 切面 (Aspect): 一个模块化的横切逻辑 (或称横切关注点), 可能会横切多个对象。
- 连接点 (Join Point): 程序执行中的某个具体的执行点。图 4.8 中原对象的 fun() 方法就是一个连接点。
- 增强处理 (Advice): 切面在某个特定连接点上执行的代码逻辑。
- 切入点 (Pointcut): 对连接点的特征进行描述, 可以使用正则表达式。增强处理和一个切入点表达式相关联, 并在与这个切入点匹配的某个连接点上运行。
- 目标对象 (Target object): 被一个或多个切面增强的对象。
- AOP 代理 (AOP proxy): 由 AOP 框架所创建的对象, 实现执行增强处理方法等功能。
- 织入 (Weaving): 将增强处理连接到应用程序中的类型或对象上的过程。
- 增强处理类型: 如图 4.8 所示, 在原对象的 fun() 方法之前插入的增强处理为前置增强, 在该方法正常执行完以后插入的增强处理为后置增强, 此外还有环绕增强、异常抛出增强、最终增强等类型。



#### 说明

切面可以理解为由增强处理和切入点组成, 既包含了横切逻辑的定义, 也包含了连接点的定义。面向切面编程主要关心两个问题, 即在什么位置执行什么功能。Spring AOP 是负责实施切面的框架, 即由 Spring AOP 完成织入工作。

Advice 直译为“通知”, 但这种叫法并不确切, 在此处翻译成“增强处理”, 更便于理解。

### 4.3.2 Spring AOP 初体验



#### 问题

日志输出的代码直接嵌入在业务流程的代码中, 不利于系统的扩展和维护。如何使用 Spring AOP 来实现日志输出, 以解决这个问题呢?

#### 实现思路及关键代码

- (1) 在项目中添加 Spring AOP 相关的 jar 文件。
- (2) 编写前置增强和后置增强实现日志功能。
- (3) 编写 Spring 配置文件, 对业务方法进行增强处理。
- (4) 编写代码, 获取带有增强处理的业务对象。



程序最终运行结果如图 4.9 所示。

```
08-31 14:21:45[INFO]aop.UserServiceLogger
-调用 service.impl.UserServiceImpl@19e67d4 的 addNewUser 方法, 方法入参: [entity.User@1ab7a89]
保存用户信息到数据库
08-31 14:21:45[INFO]aop.UserServiceLogger
-调用 service.impl.UserServiceImpl@19e67d4 的 addNewUser 方法, 方法返回值: null
```

图 4.9 使用 Spring AOP 实现日志功能

首先在项目中添加所需的 jar 文件, jar 文件清单如图 4.10 所示。spring-aop-3.2.13.RELEASE.jar 提供了 Spring AOP 的实现。同时, Spring AOP 还依赖 AOP Alliance 和 AspectJ 项目中的组件, 相关版本的下载链接分别为 <https://sourceforge.net/projects/aopalliance/files/aopalliance/1.0/> 和 <http://mvnrepository.com/artifact/org.aspectj/aspectjweaver>。

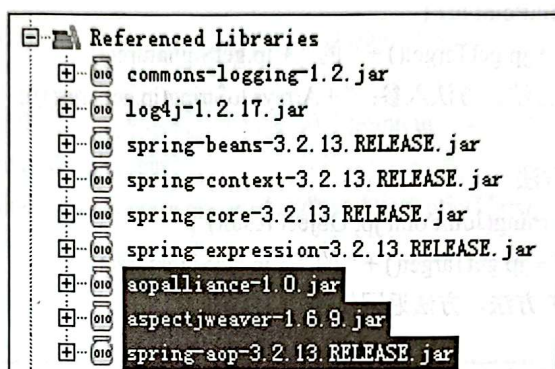


图 4.10 使用 Spring AOP 所需的 jar 文件

接下来, 编写业务类 UserServiceImpl, 代码如下例 17 所示。

#### 示例 17

```
/**
 * 用户业务类, 实现对 User 功能的业务管理
 */
public class UserServiceImpl implements UserService {
    // 声明接口类型的引用和具体实现类解耦合
    private UserDao dao;

    // dao 属性的 setter 访问器
    public void setDao(UserDao dao) {
        this.dao = dao;
    }

    public void addNewUser(User user) {
        // 调用用户 DAO 的方法保存用户信息
        dao.save(user);
    }
}
```

UserServiceImpl 业务类中有一个 addNewUser() 方法，实现用户业务的添加。可以发现，在该方法中并没有实现日志输出功能，接下来就以 AOP 的方式为该方法添加日志功能。编写增强类，代码如下示例 18 所示。

#### 示例 18

```
import java.util.Arrays;
import org.apache.log4j.Logger;
import org.aspectj.lang.JoinPoint;
/**
 * 定义包含增强方法的 JavaBean
 */
public class UserServiceLogger {
    private static final Logger log=Logger.getLogger(UserServiceLogger.class);
    // 代表前置增强的方法
    public void before(JoinPoint jp) {
        log.info(" 调用 " + jp.getTarget() + " 的 " + jp.getSignature().
            getName() + " 方法。方法入参: " + Arrays.toString(jp.getArgs()));
    }
    // 代表后置增强的方法
    public void afterReturning(JoinPoint jp, Object result) {
        log.info(" 调用 " + jp.getTarget() + " 的 " + jp.getSignature().
            getName() + " 方法。方法返回值: " + result);
    }
}
```

UserServiceLogger 类中定义了 before() 和 afterReturning() 两个方法。我们希望把 before() 方法作为前置增强使用，即将该方法添加到目标方法之前执行；把 afterReturning() 方法作为后置增强使用，即将该方法添加到目标方法正常返回之后执行。这里先以前置增强和后置增强为例，其他增强类型会在后续章节中介绍。

为了能够在增强方法中获得当前连接点的信息，以便实施相关的判断和处理，可以在增强方法中声明一个 JoinPoint 类型的参数，Spring 会自动注入实例。通过实例的 getTarget() 方法得到被代理的目标对象，通过 getSignature() 方法返回被代理的目标方法，通过 getArgs() 方法返回传递给目标方法的参数数组。对于实现后置增强的 afterReturning() 方法，还可以定义一个参数用于接收目标方法的返回值。

在 Spring 配置文件中对相关组件进行声明。配置如示例 19 所示。

#### 示例 19

```
<bean id="dao" class="dao.impl.UserDaoImpl"></bean>
<bean id="service" class="service.impl.UserServiceImpl">
    <property name="dao" ref="dao"></property>
</bean>
```

```
<bean id="theLogger" class="aop.UserServiceLogger"></bean>
```

接下来在 Spring 配置文件中进行 AOP 相关的配置，首先定义切入点，代码如下示例



20 所示。

### 示例 20

```
<?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:aop="http://www.springframework.org/schema/aop"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.2.xsd">
  <bean id="dao" class="dao.impl.UserDaoImpl"></bean>
  <bean id="service" class="service.impl.UserServiceImpl">
    <property name="dao" ref="dao"></property>
  </bean>
  <bean id="theLogger" class="aop.UserServiceLogger"></bean>
  <aop:config>
    <!-- 定义一个切入点表达式，并命名为“pointcut” -->
    <aop:pointcut id="pointcut"
      expression="execution(public void addNewUser(entity.User))"/>
  </aop:config>
</beans>
```



### 注意

在 <beans> 元素中需要添加 aop 的名称空间，以导入与 AOP 相关的标签。

与 AOP 相关的配置都放在 <aop:config> 标签中，如配置切入点的标签 <aop:pointcut>。<aop:pointcut> 的 expression 属性可以配置切入点表达式，示例 20 中它的值为：

```
execution(public void addNewUser(entity.User))
```

execution 是切入点指示符，括号中是一个切入点表达式，用于配置需要切入增强处理的方法的特征。切入点表达式支持模糊匹配，下面介绍几种常用的模糊匹配。

- public \* addNewUser(entity.User): “\*” 表示匹配所有类型的返回值。
- public void \*(entity.User): “\*” 表示匹配所有方法名。
- public void addNewUser(..): “..” 表示匹配所有参数个数和类型。
- \* com.service.\*(..): 这个表达式匹配 com.service 包下所有类的所有方法。
- \* com.service..\*(..): 这个表达式匹配 com.service 包及其子包下所有类的所有方法。

具体使用时可以根据自己的需求来设置切入点的匹配规则。当然，匹配的规则和关

键字还有很多，可以参考 Spring 的开发手册学习。

最后还需要在切入点处插入增强处理，这个过程的专业叫法是“织入”。实现织入的配置代码如下示例 21 所示。

### 示例 21

```
<?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:aop="http://www.springframework.org/schema/aop"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.2.xsd">
  <bean id="dao" class="dao.impl.UserDaoImpl"></bean>
  <bean id="service" class="service.impl.UserServiceImpl">
    <property name="dao" ref="dao"></property>
  </bean>
  <bean id="theLogger" class="aop.UserServiceLogger"></bean>
  <aop:config>
    <aop:pointcut id="pointcut"
      expression="execution(public void addNewUser(entity.User))" />
    <!-- 引用包含增强方法的 Bean -->
    <aop:aspect ref="theLogger">
      <!-- 将 before() 方法定义为前置增强并引用 pointcut 切入点 -->
      <aop:before method="before"
        pointcut-ref="pointcut"></aop:before>
      <!-- 将 afterReturning() 方法定义为后置增强并引用 pointcut 切入点 -->
      <!-- 通过 returning 属性指定为名为 result 的参数注入返回值 -->
      <aop:after-returning method="afterReturning"
        pointcut-ref="pointcut" returning="result"/>
    </aop:aspect>
  </aop:config>
</beans>
```

如示例 21 的配置所示，在 `<aop:config>` 中使用 `<aop:aspect>` 引用包含增强方法的 Bean，然后分别通过 `<aop:before>` 和 `<aop:after-returning>` 将方法声明为前置增强和后置增强，在 `<aop:after-returning>` 中通过 `returning` 属性指定需要注入返回值的属性名。方法的 `JoinPoint` 类型参数无须特殊处理，Spring 会自动为其注入连接点实例。

很明显，`UserService` 的 `addNewUser()` 方法可以和切入点 `pointcut` 相匹配，Spring 会生成代理对象，在它执行前后分别调用 `before()` 和 `afterReturning()` 方法，这样就完成了日志输出。

编写测试代码，如示例 22 所示。



**示例 22**

```

ApplicationContext ctx = new ClassPathXmlApplicationContext(
    "applicationContext.xml");
UserService service = (UserService) ctx.getBean("service");

User user = new User();
user.setId(1);
user.setUsername("test");
user.setPassword("123456");
user.setEmail("test@xxx.com");

service.addNewUser(user);

```

运行结果如图 4.9 所示。

从以上示例可以看出，业务代码和日志代码是完全分离的，经过 AOP 的配置以后，不做任何代码上的修改就在 addNewUser() 方法前后实现了日志输出。其实，只需稍稍修改切入点的指示符，不仅可以为 UserService 的 addNewUser() 方法增强日志功能，也可以为所有业务方法进行增强；并且可以增强日志功能，如实现访问控制、事务管理、性能监测等实用功能。

**技能训练****上机练习 3——使用 Spring AOP 实现日志功能****需求说明**

模仿示例 17 至示例 22，使用前置增强和后置增强对业务方法的执行过程进行日志记录。

**提示**

- (1) 在项目中添加 Spring AOP 相关的 jar 文件。
- (2) 编写前置增强和后置增强实现日志功能。
- (3) 编写 Spring 配置文件，定义切入点并对业务方法进行增强处理。
- (4) 编写代码，获取带有增强处理的业务对象。
- (5) 关键代码参考示例 17 至示例 22。

**本章总结**

- Spring 是一个轻量级的企业级框架，提供了 IoC 容器、AOP 实现、DAO/ORM 支持、Web 集成等功能，目标是使现有的 Java EE 技术更易用，并形成良好的编程习惯。



- 依赖注入让组件之间以配置文件的形式组织在一起，而不是以硬编码的方式耦合在一起。
- Spring 配置文件是完成组装的主要场所，常用节点包括 <bean> 及其子节点 <property>。
- AOP 的目的是从系统中分离出切面，将其独立于业务逻辑实现，并在程序执行时织入程序中运行。
- 面向切面编程主要关心两个问题：在什么位置，执行什么功能。
- 配置 AOP 主要使用 aop 命名空间下的元素完成，可以完成定义切入点和织入增强等操作。

## 本章练习

1. 根据你的理解，讲讲什么是依赖注入，以及依赖注入给我们的项目开发带来了什么好处。
2. 根据你的理解，讲讲什么是 AOP，以及使用 AOP 有什么好处。
3. 某网络游戏程序中有如下类：

```

public class Equip{                                // 装备
    private String name;                            // 装备名称
    private String type;                            // 装备类型, 头盔、铠甲等
    private Long speedPlus;                         // 速度增效
    private Long attackPlus;                        // 攻击增效
    private Long defencePlus;                       // 防御增效
    // 省略 getters & setters
    .....
}

public class Player {                               // 玩家
    private Equip armet;                             // 头盔
    private Equip loricae;                           // 铠甲
    private Equip boot;                              // 靴子
    private Equip ring;                             // 指环
    // 省略 getters & setters
    .....

    // 升级装备
    public updateEquip(Equip equip){
        if(" 头盔 ".equals(equip.getType())){
            System.out.println(armet.getName() + " 升级为 "
                + equip.getName());
            this.armet = equip;
        }
    }
}

```



```

    }
    // 省略其他装备判断
    .....
}
}

```

根据以上信息，使用 Spring DI 配置一个拥有如表 4-1 所示装备的玩家。

表4-1 玩家的装备

装备	战神头盔	连环锁子甲	波斯追风靴	蓝魔指环
速度增效	2	6	8	8
攻击增效	4	4	2	12
防御增效	6	15	3	2

### 提示

```

<bean id="zhanShenArmet" class="Equip">
  <property name="name" value=" 战神头盔 " />
  <property name="type" value=" 头盔 " />
  <property name="speedPlus" value="2" />
  .....
</bean>
.....
<bean id="zhangsan" class="Player">
  <property name="armet" ref="zhanShenArmet" />
  .....
</bean>

```

4. 在练习 3 的基础上编写程序代码，以如下格式输出蓝魔指环的属性：

蓝魔指环 [ 速度增效：8； 攻击增效：12； 防御增效：2]

### 提示

```

Equip lanMoRing = (Equip)context.getBean("lanMoRing");
System.out.println(.....);

```

5. 在练习 4 的基础上使用 AOP 实现如下功能。

现举行免费升级指环的活动，可以免费将任意指环升级为“紫色梦幻”指环，新的装备名称为“紫色梦幻+原指环名”，而且将在原指环基础上再加 6 点攻击、6 点防御。

### 提示

使用前置增强，判断要升级的装备是否为指环，如果是，则按需求修改传入的参数的名称，以及攻击增效和防御增效属性。