第2章 Spring框架基础

广东财经大学信息学院
罗 东 俊 博士
ZSUJONE@126.COM

（内部资料，请勿外传）

目的和要求

- 了解Spring的体系结构。
- 理解Spring的软件构造核心思想—— IoC和AOP思想。
- 掌握Spring的软件构造基本方法—— IoC和AOP编程。

主要内容

- 2.1 Spring概述
- 2.2 Spring IoC
- 2.3 Spring AOP

2.1 Spring概述

- 2.1.1 什么是Spring
- 2.1.2 Spring的体系结构
- 2.1.3 Spring框架的优点

2.1.1 什么是Spring

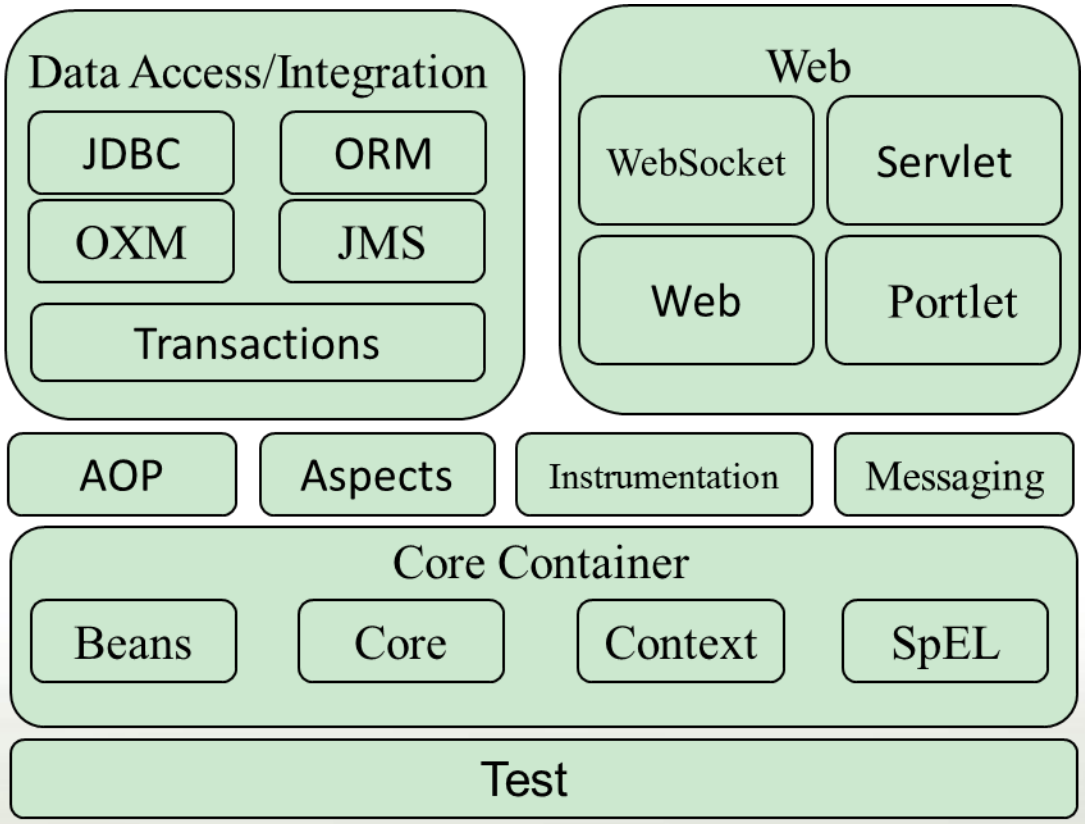
- Spring是分层的JavaSE/EE full-stack轻量级开源框架，以IoC（Inverse of Control，控制反转）和AOP（Aspect Oriented Programming，面向切面编程）为内核，使用基本的JavaBean来完成以前只可能由EJB完成的工作，取代了EJB的臃肿、低效的开发模式。

Spring的目标

- Spring致力于JavaEE应用各层的解决方案
  - ✧在表现层(Web)提供了SpingMVC以及与Struts等框架的整合功能；
  - ✧在业务逻辑层(Service)可以管理事务、记录日志等
  - ✧在持久层(Dao)可以整合MyBatis、Hibernate、JdbcTemplate等技术

2.1.2 Spring的体系结构

- Spring框架采用的是分层架构，它一系列的功能要素被分成20个模块。



1.Core Container(核心容器)

- **Beans：提供了BeanFactory，Spring将管理对象称为Bean。**
- **Core：提供了Spring框架的基本组成部分，包括IoC和DI功能。**
- **Context：建立在Core和Beans模块的基础之上，它是访问定义和配置的任何对象的媒介。**
- **SpEL：Spring3.0后新增的模块，是运行时查询和操作对象图的强大的表达式语言。**

2.Data Access/Integration

- **JDBC**：提供了一个**JDBC**的抽象层，大幅度的减少了在开发过程中对数据库操作的编码。
- **ORM**：对流行的对象关系映射**API**，包括**JPA**、**JDO**和**Hibernate**提供了集成层支持。
- **OXM**：提供了一个支持对象**/XML**映射的抽象层实现，如**JAXB**、**Castor**、**XMLBeans**、**JiBX**和**XStream**。
- **JMS**：**4.1**版本后支持与**Spring-message**模块的集成。
- **Transactions**：支持对实现特殊接口以及所有**POJO**类的编程和声明式的事务管理。

3.Web

- **WebSocket：Spring4.0以后新增的模块，它提供了WebSocket和SockJS的实现，以及对STOMP的支持。**
- **Servlet：也称Spring-webmvc模块，包含Spring模型—视图—控制器（MVC）和REST Web Services实现的Web程序。**
- **Web：提供了基本的Web开发集成特性，如：多文件上传、使用Servlet监听器来初始化IoC容器以及Web应用上下文。**
- **Portlet：提供了在portlet环境中使用MVC实现，类似Servlet模块的功能。**

4.其他模块

- **AOP**：提供了面向切面编程实现，允许定义方法拦截器和切入点，将代码按照功能进行分离，以降低耦合性。
- **Aspects**：提供了与**AspectJ**的集成功能，**AspectJ**是一个功能强大且成熟的面向切面编程（**AOP**）框架。
- **Instrumentation**：提供了类工具的支持和类加载器的实现，可以在特定的应用服务器中使用。
- **Messaging**：**Spring4.0**以后新增的模块，它提供了对消息传递体系结构和协议的支持。
- **Test**：提供了对单元测试和集成测试的支持。

2.1.3 Spring框架的优点

- Spring具有简单、可测试和松耦合等特点。Spring不仅可以用于服务器端开发，也可以应用于任何Java应用的开发中。

七大优点

- 1.非侵入式（non-invasive）设计
  - ✧非侵入式使应用程序代码对框架的依赖最小化。
- 2.方便解耦、简化开发
  - ✧Spring就是一个大工厂，可以将所有对象的创建和依赖关系的维护工作都交给Spring容器管理，大大降低了组件之间的耦合性。
- 3.支持AOP
  - ✧AOP允许将一些通用任务，如安全、事务、日志等进行集中式处理，从而提高了程序的复用性。
- 4.支持声明式事务处理
  - ✧只需通过配置就可以完成对事务的管理，而无需手动编程

七大优点

- **5.方便程序测试**
  - ✧提供了对**Junit**的支持，可以通过注解方便地测试**Spring**程序
- **6.方便集成各种优秀框架**
  - ✧不排斥各种优秀的开源框架，其内部提供了对各种优秀框架的直接支持
- **7.降低Java EE API的使用难度**
  - ✧对**Java EE**开发中非常难用的一些**API**（如**JDBC**、**JavaMail**等）都提供了封装，使这些**API**应用难度大大降低。

2.2 Spring IoC

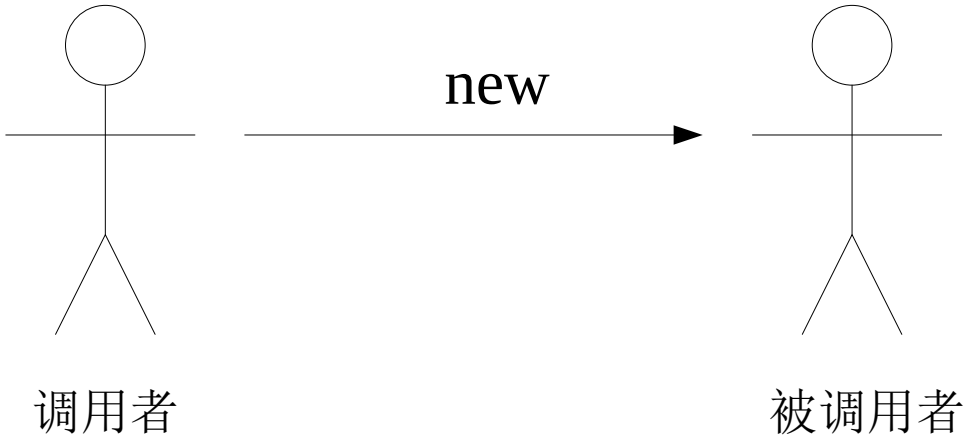
- 2.2.1 IoC的基本概念
- 2.2.2 Bean的装配方式
- 2.2.3 Bean的作用域

2.2.1 IoC的基本概念

- IoC（Inversion of Control）和DI（Dependency Injection）是从两个不同角度描述的同一个概念。
  - IoC：在使用Spring框架之后，对象的实例不再由调用者来创建，而是由Spring容器来创建，Spring容器会负责控制程序之间的关系，而不是由调用者的程序代码直接控制。这样，控制权由应用代码转移到了Spring容器，控制权发生了反转，这就是控制反转。
  - DI：从Spring容器的角度来看，Spring容器负责将被依赖对象赋值给调用者的成员变量，这相当于为调用者注入了它依赖的实例，这就是Spring的依赖注入。

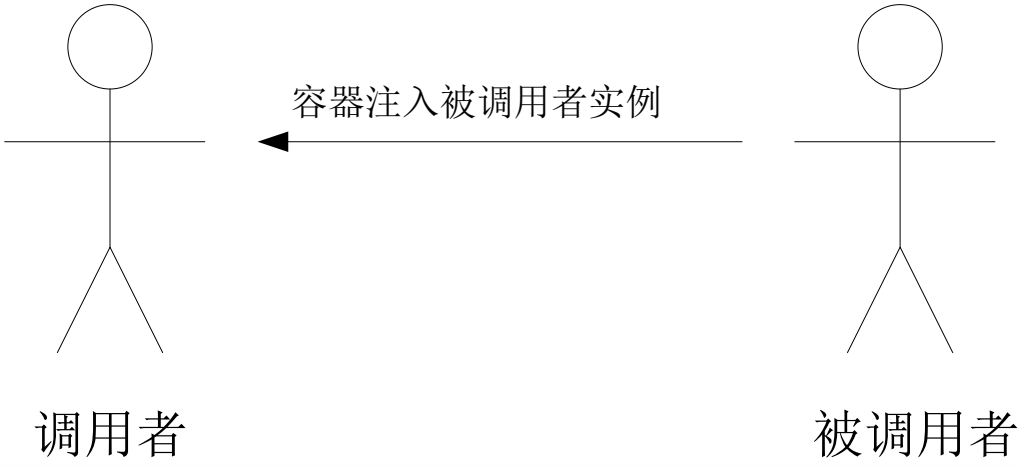
传统模式下的DI

- 传统模式下，通过调用者创建被调用者对象来实现DI。



Spring框架下的DI

- Spring框架下，被调用者对象由Spring容器创建，然后再注入给调用者对象。
  - ✧由Spring容器创建和管理的对象称为Bean。



2.2.2 Bean的装配方式

- Bean的装配可以理解为Bean依赖关系的注入过程。
- Spring容器支持多种形式的Bean的装配方式：
  - 1.基于XML配置文件的装配
  - 2.基于Annotation的装配

1.基于XML配置文件的装配

- Spring提供了两种基于XML配置文件的装配方式：
  - ✧①属性setter方法注入（设值注入）
  - ✧②构造方法注入（构造注入）

①属性setter方法注入

- 指Spring容器使用setter方法注入被依赖的实例。
  - ✧通过调用无参构造器或无参静态工厂方法实例化Bean后，调用该Bean的setter方法注入被依赖的实例。
  - ✧依赖者必须提供默认空参构造方法，并为所有属性提供setter方法。

②构造方法注入

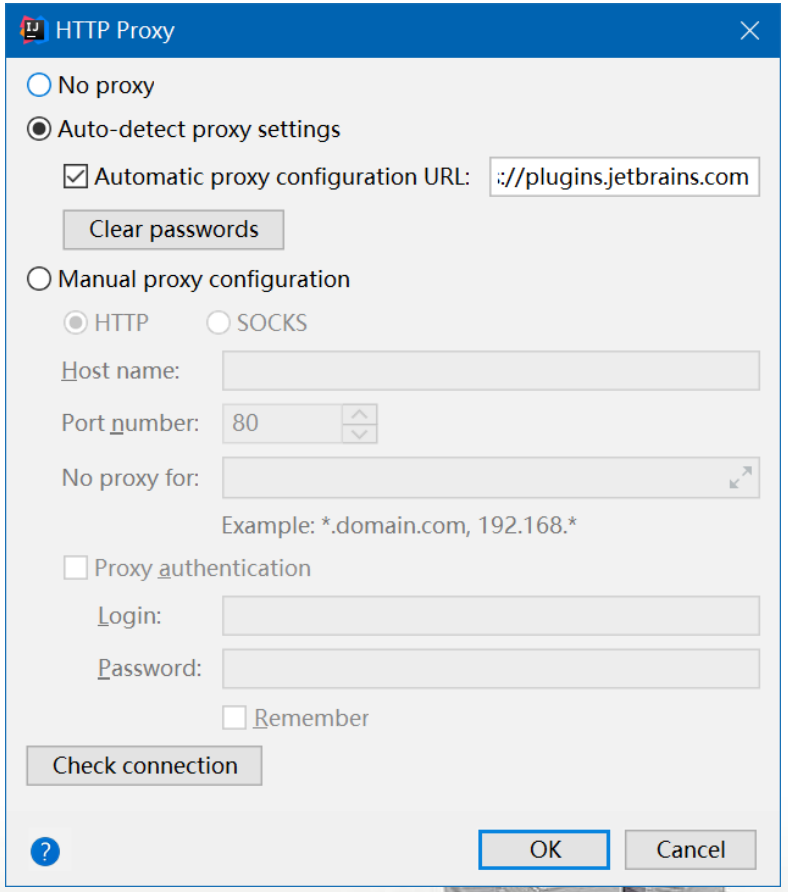
- 指Spring容器使用构造方法注入被依赖的实例。
  - ✧通过调用带参的构造器实例化Bean，每个参数代表着一个依赖。
  - ✧依赖者必须提供带有所有属性的有参构造方法。

基于XML配置文件的装配案例

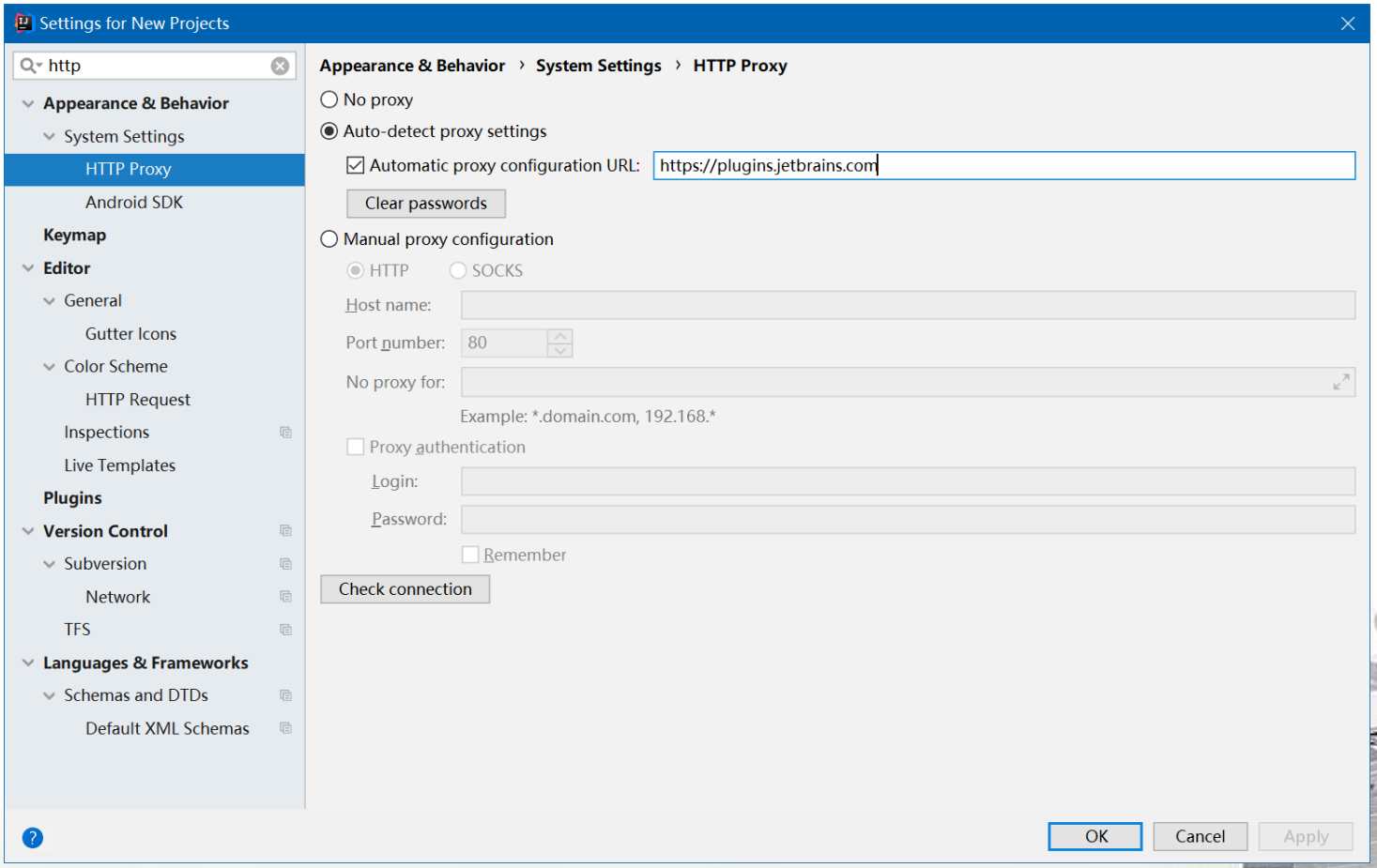
- 搭建步骤：
  - ①环境准备
  - ②创建Spring基础项目
  - ③创建DAO类
  - ④创建业务类
  - ⑤创建控制器类
  - ⑥创建XML配置文件
  - ⑦创建测试类
  - ⑧查看测试结果

①环境准备

- 保证安装好软件如下：
  - ✧JDK 1.8.0_201（及以上版本）
  - ✧IntelliJ IDEA Ultimate 2018旗舰版
- HTTP Proxy setting：https://plugins.jetbrains.com

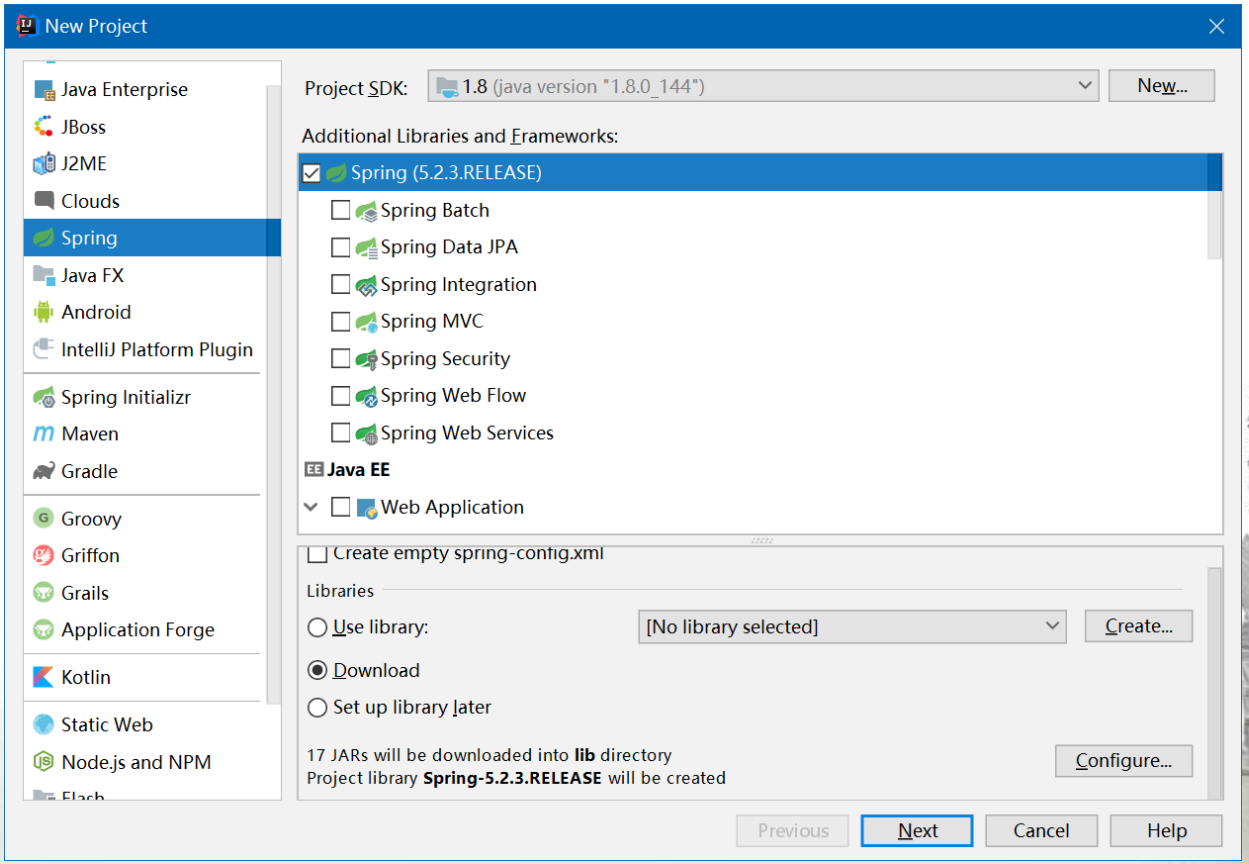


HTTP Proxy setting



②创建Spring基础项目

- 在IDEA中，创建一个名为chapter02的Spring基础项目，IDEA会自动下载和加载必要的jar包。



③创建DAO类

- 在chapter02项目的src目录下，创建一个com.itheima.assemble包，并在包中创建接口UserDao及其实现类UserDaoImpl，要求实现类中实现接口中定义的save()方法。

DAO类

```
public interface UserDao {
        public void save();
}
```

```
public class UserDaoImpl implements UserDao {
        public void save() {
                System.out.println("userDao…save…");
        }
}
```

④创建业务类

- 在com.itheima.assemble包中创建接口UserService及其实现类UserServiceImpl。
- 在实现类中实现接口中定义的save()方法和属性userDao的设值注入，save()方法调用UserDao的save()方法，并输出提示。

业务类

```
public interface UserService {
        public void save();
}
```

```
public class UserServiceImpl implements UserService {
        private UserDao userDao;
        public void save() {
                this.userDao.save();
                System.out.println("userService…save…");
        }
        public void setUserDao(UserDaoImpl userDao) {
                this.userDao=userDao;
        }
}
```

⑤创建控制器类

- 在com.itheima.assemble包中创建控制器类UserController，要求在类中定义save()方法和实现属性userService的构造注入和设值注入， save()方法调用UserService的save()方法，并输出提示。

控制器类

```
public class UserController {
        private UserService userService;
        public UserController(){
            super();
        }
        public UserController(UserService userService){
            super();
            this.userService=userService;
        }
        public void save() {
            this.userService.save();
            System.out.println("userController…save…");
        }
        public void setUserService(UserServiceImpl userService) {
            this.userService=userService;
        }
}
```

⑥创建XML配置文件

- 在com.itheima.assemble包中创建XML配置文件beans.xml，在配置文件中通过设值注入或（和）构造注入的方式装配UserDao类、UserService类和UserController类的实例。

XML配置文件

```
<?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-4.3.xsd">
   <!-- 将指定类配置给Spring，让Spring创建其对象的实例 -->
   <bean id="userDao" class="com.itheima.assemble.UserDaoImpl" />
   <!--使用属性设值注入方式添加一个id为userService的Bean实例 -->
   <bean id="userService" class="com.itheima.assemble.UserServiceImpl">
      <!-- 将id为userDao的Bean实例注入到userService实例中 -->
      <property name="userDao" ref="userDao" />
   </bean>
   <!--使用属性设值注入方式添加一个id为userController1的Bean实例 -->
   <bean id="userController1" class="com.itheima.assemble.UserController">
      <!-- 将id为userService的Bean实例注入到userController实例中 -->
      <property name="userService" ref="userService" />
   </bean>

   <!--使用属性构造注入方式添加一个id为userController2的Bean实例 -->
   <bean id="userController2" class="com.itheima.assemble.UserController">
      <!-- 将id为userService的Bean实例注入到userController实例中 -->
      <constructor-arg index="0" ref="userService"/>
   </bean>
</beans>
```

XML配置文件简化形式

- XML配置文件可以基于自动装配形式进行简化。
- 所谓自动装配，就是将一个Bean自动地注入到其他Bean的Property中。
- 实现方法：
  - ✧使用<bean>元素中的autowire属性值。

autowire属性值及说明

属性值	说明
default（默认值）	由<bean>的上级标签<beans>的 default-autowire 属性值确定。例如<beans default-autowire="byName">，则该<bean>元素中的 autowire 属性对应的属性值就为 byName。
byName	根据属性的名称自动装配。容器将根据名称查找与属性完全一致的 Bean，并将其属性自动装配。
byType	根据属性的数据类型（Type）自动装配，如果一个 Bean 的数据类型，兼容另一个 Bean 中属性的数据类型，则自动装配。
constructor	根据构造函数参数的数据类型，进行 byType 模式的自动装配。
no	默认情况下，不使用自动装配，Bean 依赖必须通过 ref 元素定义。

简化的XML配置文件

```
<?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-4.3.xsd">
    <!-- 将指定类配置给Spring，让Spring创建其对象的实例 -->
    <bean id="userDao" class="com.itheima.assemble.UserDaoImpl" />
    <!--使用属性设值注入方式添加一个id为userService的Bean实例 -->
    <bean id="userService" class="com.itheima.assemble.UserServiceImpl" autowire="byName"/>
    <!--使用属性设值注入方式添加一个id为userController1的Bean实例 -->
    <bean id="userController1" class="com.itheima.assemble.UserController" autowire="byName"/>
    <!--使用属性构造注入方式添加一个id为userController2的Bean实例 -->
    <bean id="userController2" class="com.itheima.assemble.UserController" autowire="constructor"/>
</beans>
```

⑦创建测试类

- 在com.itheima.assemble包中创建测试类XmlBeanAssembleTest，在类中分别获取配置文件的userContonroller1和userContonroller2实例，并调用实例中的save()方法。

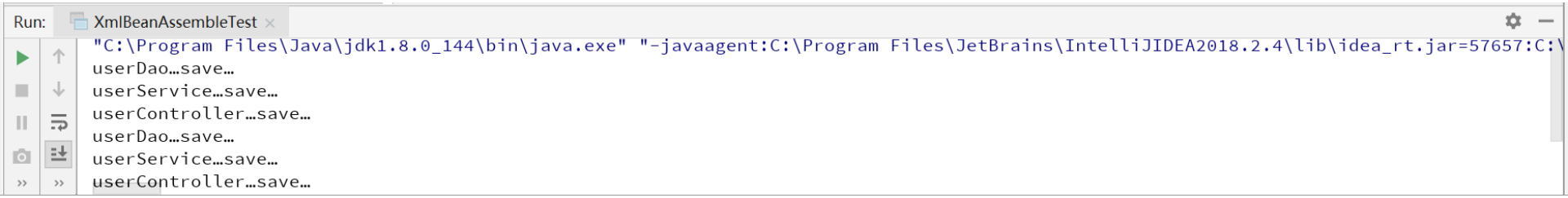
测试类

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class XmlBeanAssembleTest {
    public static void main(String[] args) {
        // 定义配置文件路径
        String xmlPath = "com/itheima/assemble/beans.xml";
        // 加载配置文件
        ApplicationContext applicationContext =
                new ClassPathXmlApplicationContext(xmlPath);
        // 获取UserController1实例
        UserController userController1 =
                (UserController) applicationContext.getBean("userController1");
        userController1.save();
        // 获取userController2实例
        UserController userController2 =
                (UserController) applicationContext.getBean("userController2");
        userController2.save();
    }
}
```

⑧查看测试结果

- 测试结果如图所示。



2.基于Annotation的装配

- 基于XML的装配可能会导致XML配置文件过于臃肿，给后续的维护和升级带来一定的困难。为此，Spring提供了对Annotation（注解）技术的全面支持。

常用注解

- **@Component：将一个自定义类作为Bean添加到Spring容器中。**
- **@Repository：用于将数据访问层（DAO）的类标识为Spring中的Bean 。**
- **@Service：用于将业务层（Service）的类标识为Spring中的Bean。**
- **@Controller：用于将控制层（Controller）的类标识为Spring中的Bean 。**

常用注解

- **@Autowired：**该注解可以对类成员变量、方法及构造方法进行标注，完成自动装配的工作。
  - ✧通过@Autowired的使用来消除setter 和getter方法。
  - ✧默认按Bean类型装配。如果按Bean实例名称装配需结合使用@Qualifier，由@Qualifier的参数指定。

常用注解

- **@Resource：同@Autowired，按Bean实例名称或类型进行装配，由@Resource的name或type参数指定。**
  - ✧**如果按Bean名称装配，需在使用@Repository、@Service、 @Controller等添加一个Bean实例时指定Bean实例的id，如：**
    - **@Repository(“userDao”)：添加一个id为userDao的DAO类Bean实例；**
    - **@Service(“userService”)：添加一个id为userService的业务类Bean实例；**
    - **@Controller(“userController”)：添加一个id为userController的控制器类Bean实例。**

基于Annotation的装配案例

- 搭建步骤：
  - ✧①创建DAO类
  - ✧②创建业务类
  - ✧③创建控制器类
  - ✧④创建配置类
  - ✧⑤创建测试类
  - ✧⑥查看测试结果

①创建DAO类

- 在项目chapter02的src目录下，创建一个com.itheima.annotation包，并在包中创建接口UserDao及其实现类UserDaoImpl，要求实现类中实现接口中定义的save()方法。

DAO类

```
public interface UserDao {
        public void save();
}
```

```
import org.springframework.stereotype.Repository;

//添加一个DAO类的Bean实例
@Repository
public class UserDaoImpl implements UserDao {
        public void save() {
                System.out.println("userDao…save…");
        }
}
```

②创建业务类

- 在com.itheima.annotation包中创建接口UserService及其实现类UserServiceImpl，要求在实现类中实现接口中定义的save()方法和属性userDao的设值自动注入，save()方法调用UserDao的save()方法，并输出提示。

业务类

```
public interface UserService {
     public void save();
}
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
//添加一个业务类的Bean实例
@Service
public class UserServiceImpl implements UserService {
   //使用属性设值自动注入方式
   @Autowired
   private UserDao userDao;
   public void save() {
      this.userDao.save();
      System.out.println("userservice....save...");
   }
}
```

③创建控制器类

- 在com.itheima.annotation包中创建控制器类UserController，要求在类中定义save()方法和实现属性userService的构造自动注入，save()方法调用UserService的save()方法，并输出提示。

控制器类

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
//添加一个控制器类的Bean实例
@Controller
public class UserController {
    private UserService userService;
    //使用属性构造自动注入方式
    @Autowired
    public UserController(UserService userService) {
        this.userService = userService;
    }
    public void save() {
        this.userService.save();
        System.out.println("userController...save...");
    }
}
```

④创建配置类

- 在com.itheima.annotation包中创建配置类AnnotationAssembleConfig，由该类通知Spring扫描指定包com.itheima.annotation下所有类的Bean实例。

配置类

```
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
//定义一个配置类
@Configuration
//通知Spring扫描指定包下所有Bean实例
@ComponentScan("com.itheima.annotation")
public class AnnotationAssembleConfig {
}
```

等价的XML配置文件

```
<?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-4.3.xsd
    http://www.springframework.org/schema/context
 http://www.springframework.org/schema/context/spring-context-4.3.xsd">
   <!--使用 context命名空间 ,通知Spring扫描指定包下所有类的Bean实例，进行注解解析-->
   <context:component-scan base-package="com.itheima.annotation" />
</beans>
```

⑤创建测试类

- 在com.itheima.assemble包中创建测试类AnnotationAssembleTest，在类中获取UserContonroller类的Bean实例，并调用实例中的save()方法。

测试类

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
public class AnnotationAssembleTest {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext applicationContext=
                new AnnotationConfigApplicationContext(AnnotationAssembleConfig.class);
        //获取UserController类的Bean实例
        System.out.println("获取UserController类的Bean实例");
        UserController userController =applicationContext.getBean(UserController.class);
        userController.save();
        //获取UserService类的Bean实例
        System.out.println("获取UserService类的Bean实例");
        UserService userService =applicationContext.getBean(UserService.class);
        userService.save();
    }
}
```

⑥查看测试结果

- 测试结果如图所示。

```
Run:  AnnotationAssembleTest ×
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
获取UserController类的Bean实例
userdao...save...
userservice....save...
userController...save...
获取UserService类的Bean实例
userdao...save...
userservice....save...
```

2.2.3 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中为Bean的实例定义了7种作用域，通过@Scope注解来实现。

7种作用域

作用域名称	说明
singleton（单例）	使用 singleton 定义的 Bean 在 Spring 容器中将只有一个实例，也就是说，无论有多少个 Bean 引用它，始终将指向同一个对象。这也是 Spring 容器默认的作用域。
prototype（原型）	每次通过 Spring 容器获取的 prototype 定义的 Bean 时，容器都将创建一个新的 Bean 实例。
request	在一次 HTTP 请求中，容器会返回该 Bean 的同一个实例。对不同的 HTTP 请求则会产生一个新的 Bean，而且该 Bean 仅在当前 HTTP Request 内有效。
session	在一次 HTTP Session 中，容器会返回该 Bean 的同一个实例。对不同的 HTTP 请求则会产生一个新的 Bean，而且该 Bean 仅在当前 HTTP Session 内有效。
globalSession	在一个全局的 HTTP Session 中，容器会返回该 Bean 的同一个实例。仅在使用 portlet 上下文时有效。
application	为每个 ServletContext 对象创建一个实例。仅在 Web 相关的 ApplicationContext 中生效。**即同一个应用共享一个Bean实例。**
websocket	为每个 websocket 对象创建一个实例。仅在 Web 相关的 ApplicationContext 中生效。

示例代码

```
import org.springframework.stereotype.Service;
//添加一个业务类的singleton类型Bean实例
@Service
public class SingleService {
}
```

```
import org.springframework.stereotype.Service;
import org.springframework.context.annotation.Scope;
//添加一个业务类的prototype类型Bean实例
@Service
@Scope(“prototype”)
public class PrototypeService {
}
```

2.3 Spring AOP

- 2.3.1 Spring AOP简介
- 2.3.2 基于AspectJ的AOP实现

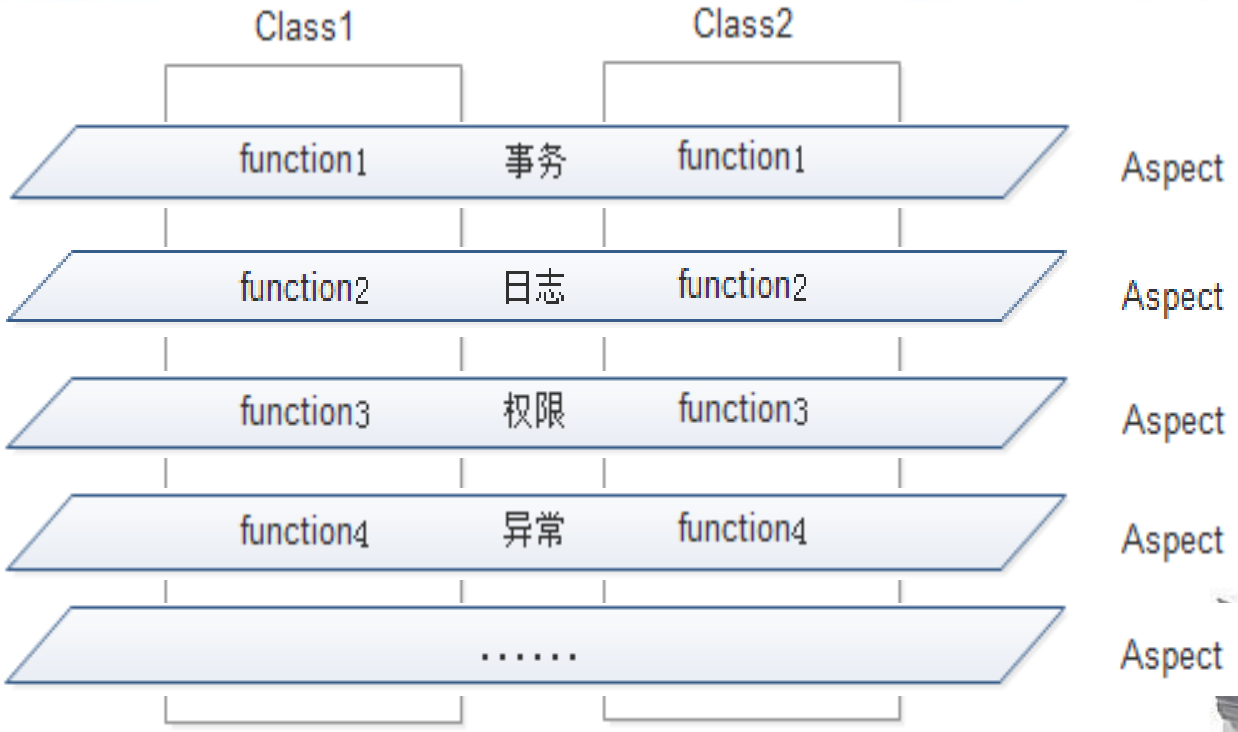
2.3.1 Spring AOP简介

- AOP的全称是Aspect-Oriented Programming，即面向切面编程（也称面向方面编程）。它是面向对象编程（OOP）的一种补充，目前已成为一种比较成熟的编程方式。

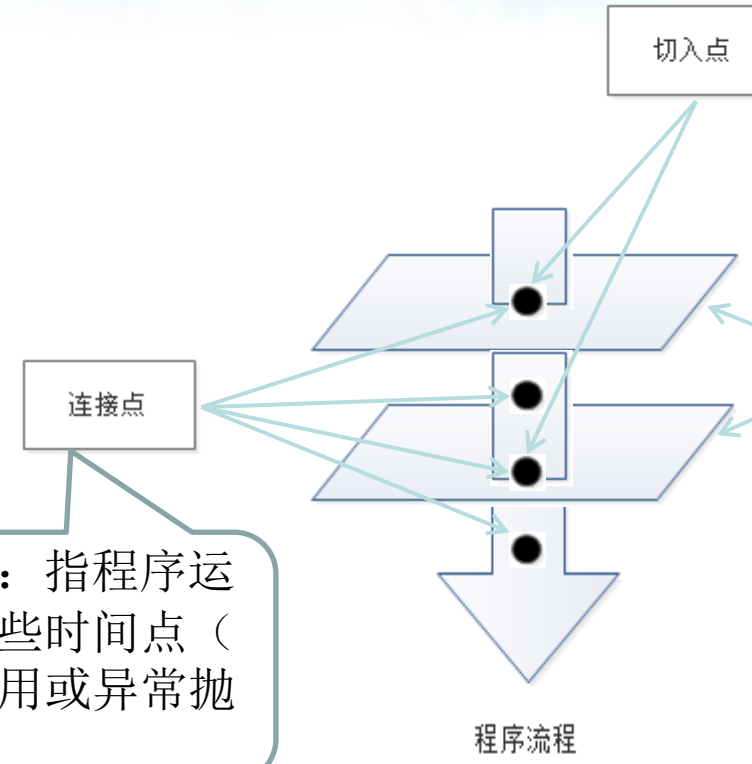
AOP思想

- AOP采取横向抽取机制，将分散在各个方法中的重复代码提取出来，然后在程序编译或运行时，再将这些提取出来的代码应用到需要执行的地方。
  - ✧传统的OOP思想显然无法办到，因为OOP只能实现父子关系的纵向的重用。

类与切面的关系


Class1
Class2
function1
事务
function1
Aspect
function2
日志
function2
Aspect
function3
权限
function3
Aspect
function4
异常
function4
Aspect
.....
Aspect

AOP术语



Pointcut：被标识为需要增强处理的连接点

Joinpoint：指程序运行中的一些时间点（如方法调用或异常抛出）

Aspect：封装的用于横向插入系统功能（如事务、日志等）的类

- Advice（通知/增强处理）：指在切入点执行的增强处理代码，可以理解为切面类中的方法，是切面的具体实现。

Spring的通知类型

- **Spring按照通知在目标类方法的连接点位置，可以分为5种类型。**
  - **org.springframework.aop.MethodBeforeAdvice（前置通知）**
    在目标方法执行前实施增强，可以应用于权限管理等功能。
  - **org.springframework.aop.AfterReturningAdvice（后置通知）**
    在目标方法执行后实施增强，可以应用于关闭流、上传文件、删除临时文件等功能。
  - **org.aopalliance.intercept.MethodInterceptor（环绕通知）**
    在目标方法执行前后实施增强，可以应用于日志、事务管理等功能。
  - **org.springframework.aop.ThrowsAdvice（异常抛出通知）**
    在方法抛出异常后实施增强，可以应用于处理异常记录日志等功能。
  - **org.springframework.aop.IntroductionInterceptor（引介通知）**
    在目标类中添加一些新的方法和属性，可以应用于修改老版本程序。

2.3.2 基于AspectJ的AOP实现

- AspectJ是一个基于Java语言的第三方AOP框架，它提供了一套具有强大AOP功能的注解。

AspectJ的注解及其描述

注解名称	描述
@Aspect	用于定义一个切面。
@Pointcut	用于定义切入点表达式。在使用时还需定义一个包含名字和任意参数的方法签名来表示切入点名称。实际上，这个方法签名就是一个返回值为 void，且方法体为空的普通的方法。
@Before	用于定义前置通知，相当于 BeforeAdvice。在使用时，通常需要指定一个 value 属性值，该属性值用于指定一个切入点表达式（可以是已有的切入点，也可以直接定义切入点表达式）。
@AfterReturning	用于定义后置通知，相当于 AfterReturningAdvice。在使用时可以指定 pointcut/value 和 returning 属性，其中 pointcut/value 这两个属性的作用一样，都用于指定切入点表达式。returning 属性值用于表示 Advice 方法中可定义与此同名的形参，该形参可用于访问目标方法的返回值。
@Around	用于定义环绕通知，相当于 MethodInterceptor。在使用时需要指定一个 value 属性，该属性用于指定该通知被织入的切入点。
@AfterThrowing	用于定义异常通知来处理程序中未处理的异常，相当于 ThrowAdvice。在使用时可指定 pointcut/value 和 throwing 属性。其中 pointcut/value 用于指定切入点表达式，而 throwing 属性值用于指定一个形参名来表示 Advice 方法中可定义与此同名的形参，该形参可用于访问目标方法抛出的异常。
@After	用于定义最终 final 通知，不管是否异常，该通知都会执行。使用时需要指定一个 value 属性，该属性用于指定该通知被织入的切入点。
@DeclareParents	用于定义引介通知，相当于 IntroductionInterceptor（不要求掌握）。

AspectJ的切入点标识机制

- AspectJ标识切入点是通过利用注解@Pointcut指明切入点表达式和切入点方法签名来实现的。
  - ✧切入点表达式指明了目标类中哪些方法需要增强处理，其语法格式为：execution(…)，它是注解@Pointcut的属性。
  - ✧切入点方法签名是一个由注解@Pointcut声明的返回值为void、方法体为空的普通方法。
    - 通过该方法签名，各类通知就能清楚其要加在目标类的哪些方法中。

切入点标识示例

```
@Pointcut("execution(* com.itheima.aspectj.*.*(..))")
private void myPointCut(){}

@Before("myPointCut()")
 public void myBefore(JoinPoint joinPoint) {
       System.out.print("前置通知 ： 模拟执行权限检查...,");
}
```

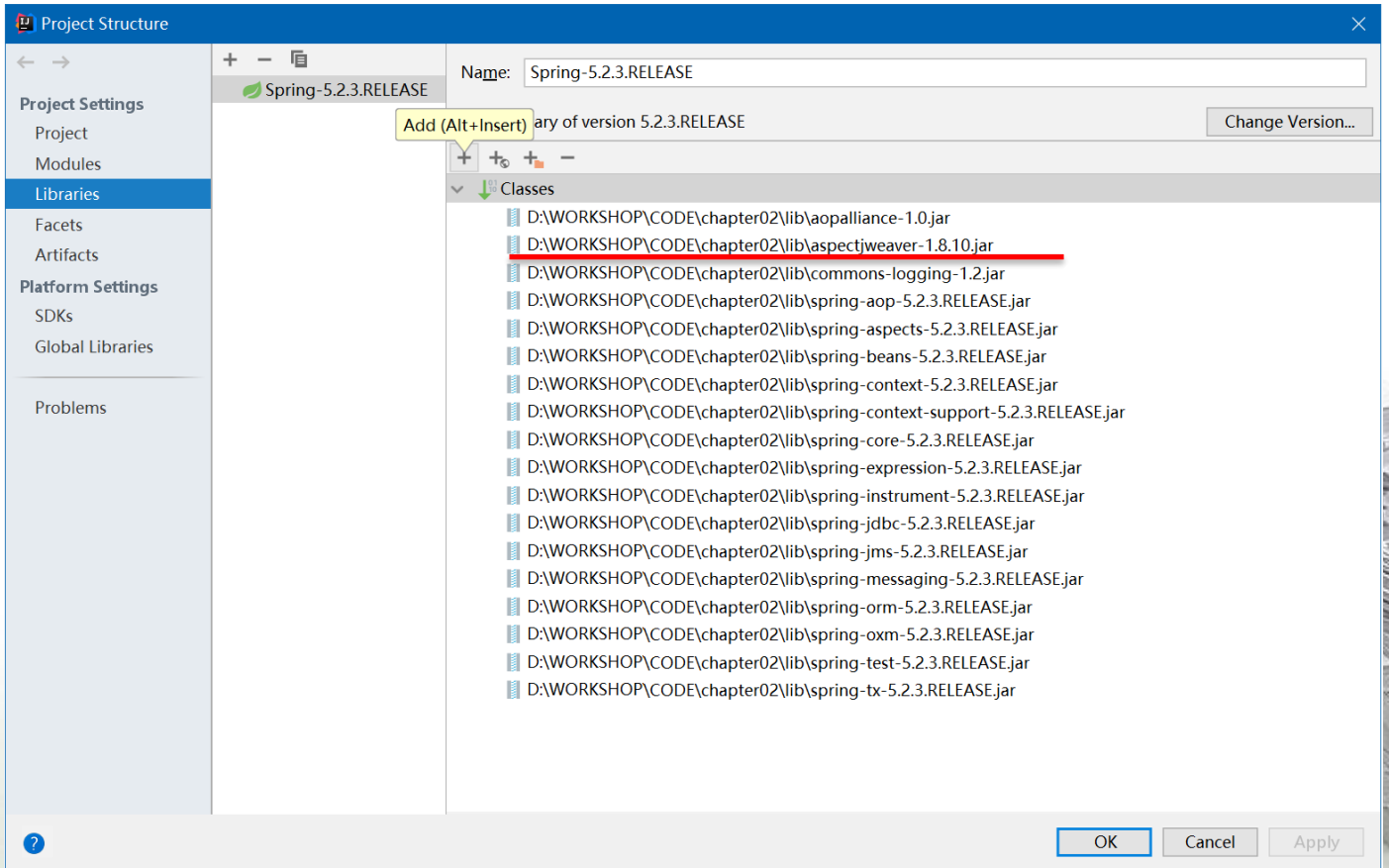
- execution(* com.itheima.aspectj.*.*(..))是切入点表达式，第一个*表示任意返回类型，com.itheima.aspectj表示的是需要匹配的包名，第二个*表示任意类名，第三个*表示任意方法名，后面(..)表示方法的任意参数。第一个*与包名之间有一个空格。
- myPointCut()是切入点方法签名，myBefore()是上述切入点的前置增强。

基于AspectJ的AOP实现案例

- 搭建步骤：
  - ①导入AspectJ框架
  - ②创建目标类
  - ③创建切面类
  - ④创建配置类
  - ⑤创建测试类
  - ⑥查看测试结果

①导入AspectJ框架

- 导入AspectJ框架的JAR包：
  - ✧aspectjweaver-1.8.10.jar



②创建目标类

- 在项目**chapter02**的**src**目录下，创建一个**com.itheima.aspectj**包，并在包中创建接口**UserDao**及其实现类**UserDaoImpl**，要求实现类中实现接口中定义的**addUser()**方法和**deleteUser()**方法。
- 本案例将实现类**UserDaoImpl**作为目标类，对其中的方法进行增强处理。

目标类

```
public interface UserDao {
    public void addUser();
    public void deleteUser();
}
```

```
import org.springframework.stereotype.Repository;
//添加一个DAO类的Bean实例
@Repository
public class UserDaoImpl implements UserDao {
    public void addUser(){
        System.out.println("添加用户");
    }
    public void deleteUser(){
        System.out.println("删除用户");
    }
}
```

③创建切面类

- 在**com.itheima.aspectj**包中创建切面类**MyAspect**，在此类中编写通知。

切面类

```
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.*;
import org.springframework.stereotype.Component;

//声明一个切面类，并作为一个组件使用时才有效
@Aspect
@Component
public class MyAspect {
    // 定义切入点，通知增强目标类中的哪些方法
    @Pointcut("execution(* com.itheima.aspectj.*.*(..))")
    private void myPointCut(){}
```

切面类

```
// 前置通知，使用JoinPoint接口作为参数获得目标对象信息
@Before("myPointCut()")
public void myBefore(JoinPoint joinPoint) {
    System.out.print("前置通知 ： 模拟执行权限检查...,");
    System.out.print("目标类是："+joinPoint.getTarget() );
    System.out.println(",被织入增强处理的目标方法为：" +joinPoint.getSignature().getName());
}
// 环绕通知
@Around("myPointCut()")
public Object myAround(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
    // 开始
    System.out.println("环绕开始：执行目标方法之前，模拟开启事务...");
    // 执行当前目标方法
    Object obj = proceedingJoinPoint.proceed();
    // 结束
    System.out.println("环绕结束：执行目标方法之后，模拟关闭事务...");
    return obj;
}
```

切面类

```
    // 异常通知
    @AfterThrowing(value="myPointCut()",throwing="e")
    public void myAfterThrowing(JoinPoint joinPoint, Throwable e) {
        System.out.println("异常通知：" + "出错了" + e.getMessage());
    }
    // 最终通知
    @After("myPointCut()")
    public void myAfter() {
        System.out.println("最终通知：模拟方法结束后的释放资源...");
    }
    // 后置通知
    @AfterReturning(value="myPointCut()")
    public void myAfterReturning(JoinPoint joinPoint) {
        System.out.print("后置通知：模拟记录日志...," );
        System.out.println("被织入增强处理的目标方法为：" + joinPoint.getSignature().getName());
    }
}
```

④创建配置类

- 在com.itheima.aspectj包中创建配置类AspectjAopConfig，由该类通知Spring扫描指定包com.itheima.aspectj下所有类使用的注解，并开启Spring对AspectJ的支持。

配置类

```
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

//声明一个配置类
@Configuration
//通知Spring扫描指定包下所有类使用的注解
@ComponentScan("com.itheima.aspectj")
//开启Spring对AspectJ的支持
@EnableAspectJAutoProxy
public class AspectjAopConfig {
}
```

等价的XML配置文件

```
<?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"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
  http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-4.3.xsd">
     <!-- 指定需要扫描的包，使注解生效 -->
     <context:component-scan base-package="com.itheima.aspectj" />
     <!--开启Spring对AspectJ的支持-->
     <aop:aspectj-autoproxy />
</beans>
```

⑤创建测试类

- 在com.itheima.aspectj包中创建测试类AspectjAopTest，在类中获取UserDao类的Bean实例，并调用实例中的方法。

测试类

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;


public class AspectjAopTest {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext applicationContext =
                new AnnotationConfigApplicationContext(AspectjAopConfig.class);
        System.out.println("获取UserDao类的Bean实例");
        UserDao userDao = applicationContext.getBean(UserDao.class);
        userDao.addUser();
        applicationContext.close();
    }
}
```

⑥查看测试结果

- 测试结果如图所示。

```
Run:  AspectjAopTest
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
获取UserDao类的Bean实例
环绕开始：执行目标方法之前，模拟开启事务...
前置通知 ：模拟执行权限检查...,目标类是：com.itheima.aspectj.UserDaoImpl@2b4bac49,被织入增强处理的目标方法为：addUser
添加用户
环绕结束：执行目标方法之后，模拟关闭事务...
最终通知：模拟方法结束后的释放资源...
后置通知：模拟记录日志...,被织入增强处理的目标方法为：addUser
```

本章小结

- 本章具体讲解了：
  - Spring概述
  - Spring IoC
  - Spring AOP


聪明等于勤奋
劳力就是幸福