

Spring

王新阳

wxyyuppie@bjfu.edu.cn



Spring简介



Spring简介

Spring是一个开源框架，它2004年由Rod Johnson创建。

➤ **目的：解决企业应用开发的复杂性，提高开发效率**

◆它完成大量开发中的通用步骤，开发者仅需关心与特定应用相关的部分

➤ **范围：任何Java应用**

◆Spring的用途不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，适用于任何Java应用。

➤ **功能：使用基本的JavaBean代替EJB(Enterprise JavaBean)，提供了更多的企业应用功能**

◆使用基本的JavaBean代替Session Bean, Entity Bean 和Message Bean





Spring简介

- Spring是一个**控制反转** (Ioc) 和**面向切面**编程 (AOP) 的**轻量级**的容器，为软件开发提供全方位支持的**应用程序框架**。
- **控制反转** (*Inversion of Control, IoC*) 与 **依赖注入** (*Dependency Injection, DI*)。由**容器**来管理对象之间的**依赖关系** (**而不是对象本身来管理**)，就叫“控制反转”或“依赖注入”。



Spring简介

- **容器**是符合某种**规范**能够提供一系列**服务**的**管理器**，开发人员可以利用容器所提供的**服务**来方便地**实现某些特殊的功能**。
- 所谓的“**重量级**”容器是指那些**完全遵守J2EE的规范**，提供规范中**所有的服务**。**EJB**就是典型的例子。
- “**轻量级**”容器的也是**遵守J2EE的规范**，但其中的**服务可以自由配置**。

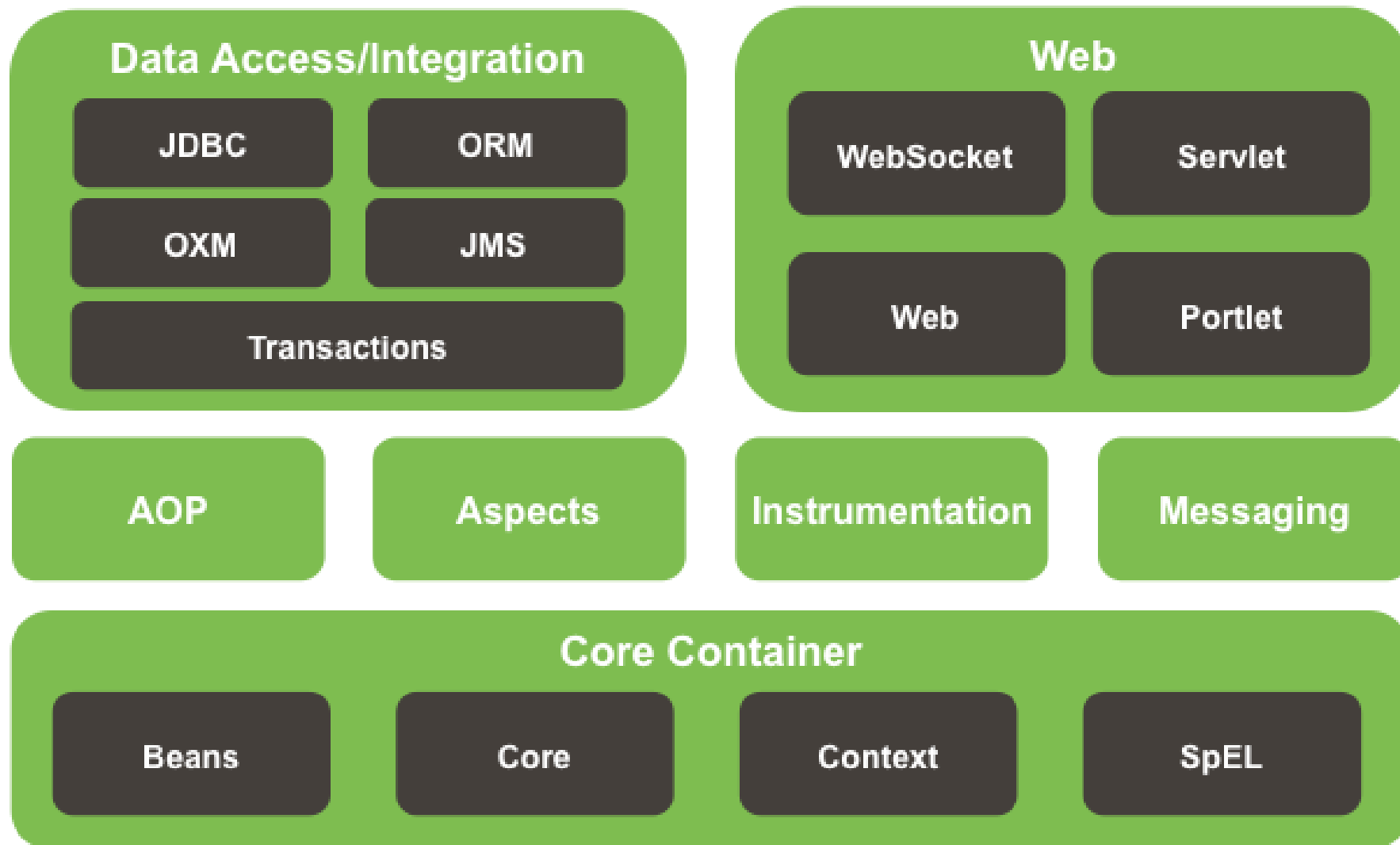


Spring特点

- **轻量级框架**: Spring 是非侵入性的 - 基于 Spring 开发的应用中的对象可以不依赖于 Spring 的 API
- **依赖注入**(DI --- Dependency Injection、IoC—Inverse of Control)降低了业务对象替换的复杂性, 提高了组件之间的解耦性
- **面向切面编程**(AOP --- Aspect Oriented Programming)支持将一些通用任务如安全, 事务和日志等进行集中式处理, 提高复用率
- Spring 是一个容器, 因为它**包含并且管理应用对象的生命周期**框架: Spring 实现了使用简单的组件配置组合成一个复杂的应用. 在 Spring 中可以使用 XML 和 Java 注解组合这些对象
- **一站式**: 在 IOC 和 AOP 的基础上可以整合各种企业应用的开源框架和优秀的第三方类库 (实际上 Spring 自身也提供了展现层的 SpringMVC 和 持久层的 Spring JDBC)



Spring组成











Spring的下载与安装——1、Spring的JAR包

Spring Framework jar官方下载路径:

<http://repo.springsource.org/libs-release-local/org/springframework/spring/>。

本课程版本: `spring-framework-5.0.2.RELEASE-dist.zip`。

 docs	2017/11/27 10:51	文件夹
 libs	2017/11/27 10:51	文件夹
 schema	2017/11/27 10:51	文件夹
 license.txt	2017/11/27 10:24	文本文档
 notice.txt	2017/11/27 10:24	文本文档
 readme.txt	2017/11/27 10:24	文本文档

三类JAR文件

- 以**RELEASE.jar**结尾的文件: Spring框架class的JAR包, 即开发Spring应用所需要的JAR包;
- 以**RELEASE-javadoc.jar**结尾的文件: Spring框架API文档的压缩包;
- 以**RELEASE-sources.jar**结尾的文件是Spring框架源文件的压缩包。

在libs目录中, 有四个基础包: `spring-core-5.0.2.RELEASE.jar`、`spring-beans-5.0.2.RELEASE.jar`、`spring-context-5.0.2.RELEASE.jar`和`spring-expression-5.0.2.RELEASE.jar`, 分别对应Spring核心容器的四个模块: **Spring-core模块**、**Spring-beans模块**、**Spring-context模块**和**Spring-expression模块**。



Spring的下载与安装

- Spring框架依赖于Apache Commons Logging组件，该组件的JAR包可以通过网址“http://commons.apache.org/proper/commons-logging/download_logging.cgi”下载，本课程版本为“commons-logging-1.2-bin.zip”。
- 开发Spring应用时，只需要将Spring的四个基础包和commons-logging-1.2.jar复制到Web应用的WEB-INF/lib目录下即可。不确定需要哪些JAR包时，可以将Spring的libs目录中spring-XXX-5.0.2.RELEASE.jar全部复制到WEB-INF/lib目录下即可。



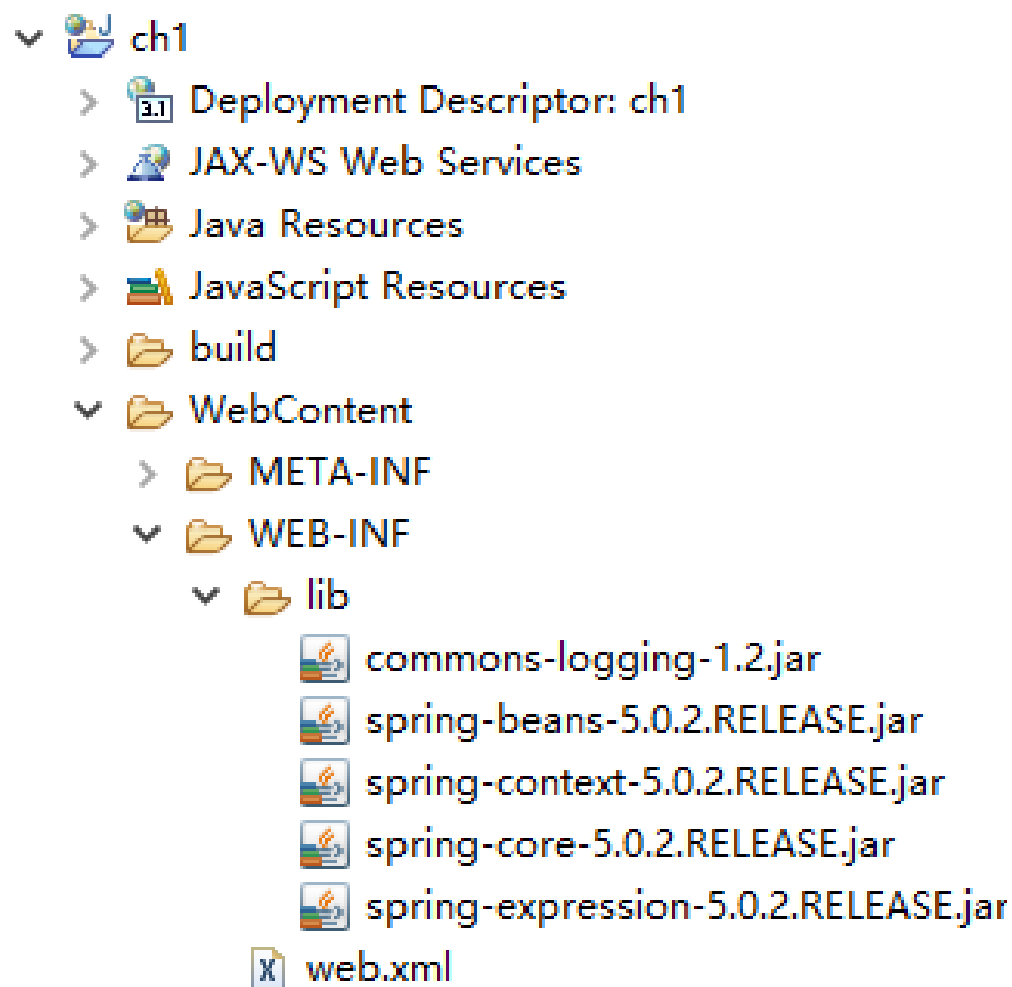
一个简单Spring示例



使用Eclipse开发Spring简单示例

1. 使用Eclipse创建Web应用并导入JAR包

见工程ch1





使用Eclipse开发Spring简单示例

2. 创建接口TestDao

在src目录下，创建一个dao包，并在dao包中创建接口TestDao，接口中定义一个sayHello()方法，代码如下：

```
package dao;  
public interface TestDao {  
    public void sayHello();  
}
```



使用Eclipse开发Spring简单示例

3. 创建接口TestDao的实现类TestDaoImpl

在包dao下创建TestDao的实现类TestDaoImpl，代码如下：

```
package dao;  
public class TestDaoImpl implements TestDao{  
    @Override  
    public void sayHello() {  
        System.out.println("Hello, Study hard!");  
    }  
}
```



使用Eclipse开发Spring简单示例

4. 创建配置文件applicationContext.xml

在src目录下，创建Spring的配置文件applicationContext.xml，并在该文件中使用实现类TestDaoImpl创建一个id为test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">
    <!-- 将指定类TestDaoImpl配置给Spring，让Spring创建其实例 -->
    <bean id="test" class="dao.TestDaoImpl" />
</beans>
```



使用Eclipse开发Spring简单示例

5. 创建测试类

```
package test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import dao.TestDao;
public class Test {
    public static void main(String[] args) {
        //初始化Spring容器ApplicationContext, 加载配置文件
        ApplicationContext appCon = new
            ClassPathXmlApplicationContext("applicationContext.xml");
        //通过容器获取test实例
        TestDao tt = (TestDao) appCon.getBean("test");//test为配置文件中的id
        tt.sayHello();
    }
}
```



依赖与依赖注入

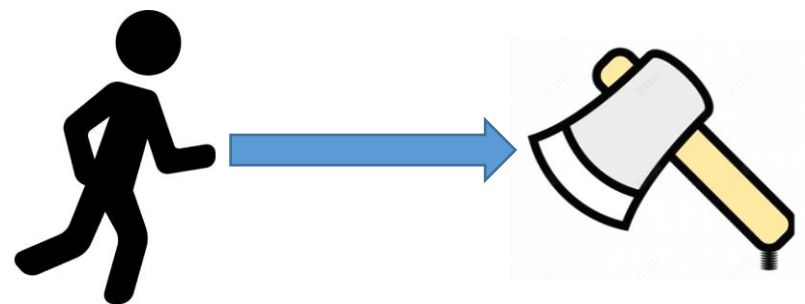


- 什么是依赖？
- 什么是依赖注入？



对象之间的依赖关系

```
public class Person{  
    private Axe axe;  
    // 设值注入所需的setter方法  
    public void setAxe(Axe axe){  
        this.axe = axe;  
    }  
    public void useAxe(){  
        System.out.println("我打算去砍点柴火! ");  
        // 调用axe的chop()方法,  
        // 表明Person对象依赖于axe对象  
        System.out.println(axe.chop());  
    }  
}
```



Person 调用 Axe

产生依赖关系

```
public class Axe{  
    public String chop(){  
        return "使用斧头砍柴";  
    }  
}
```



控制反转（依赖注入）

如果想吃面包，分为没有面包店和有面包店两种情况：

- **没有面包店：**按照自己的口味制作面包，需要购买原材料、调味、加工、烘烤等环节；
- **有面包店：**各种实体商店、网店，没有必要自己制作。直接去实体店或网店告诉店家自己的口味，由厨师制作适合自己口味的面包。

该例子中，顾客并没有自己制作面包，而是由店家制作，但包含了控制反转的思想：把制作面包的主动权交给店家。



少吃外卖！！





控制反转（依赖注入）

当某个Java对象（调用者，例如吃面包的人）需要调用另一个Java对象（被调用者，即被依赖对象，例如面包）时

- **传统编程模式**：调用者采用“new 被调用者”的代码方式来创建对象（例如自己制作面包）。

缺点：增加调用者与被调用者之间的耦合性，不利于后期代码的维护。

- **在Spring框架**：对象的实例不再由调用者来创建，而是由Spring容器（例如面包店）来创建。Spring容器负责控制程序之间的关系（例如面包店负责控制吃面包的人与面包的关系），而不是由调用者的程序直接控制。这样控制权由调用者转移到Spring容器，控制权发生了反转，这就是**控制反转**。

从Spring容器角度来看，Spring容器负责将被依赖对象赋值给调用者的成员变量，相当于为调用者注入它所依赖的实例，这就是**依赖注入**。



控制反转（依赖注入）

IoC(Inversion of Control) == DI(Dependency Injection)

基本思路：所有的Java Bean对象都交给Spring核心容器配置和管理

- **实现IoC和DI的基础：**Java的反射机制
- **IoC(Inversion of Control)：**其思想是反转资源获取的方向。传统的资源查找方式要求组件向容器发起请求查找资源。作为回应，容器适时的返回资源。而应用了 IOC 之后，则是容器主动地将资源推送给它所管理的组件，组件所要做的仅是选择一种合适的方式来接受资源。这种行为也被称为查找的被动形式
- **DI(Dependency Injection)** — IOC 的另一种表述方式：即组件以一些预先定义好的方式(例如：setter 方法)接受来自容器的资源注入。相对于 IOC 而言，这种表述更直接



依赖注入的最大特点

- 所有Java对象的创建都交给Spring容器，他们之间的关系也由Spring容器掌控
 - 程序中不再主动使用new来显式调用构造器
- 当调用者需要调用被依赖对象的方法时，只需等待Spring容器注入即可
 - 调用者无需主动获取被依赖的对象





控制反转举例

通过一个实例来了解什么是控制反转：

```
//假设有一个联想电脑的对象
public class ComputerLenovo{
    Intel intelCPU = new Intel(); //电脑的cpu
    public int calculate () {
        return intel.cal();
    }
}
```

固定死了，不能换！！

可以发现，在上面的业务中，联想的电脑将自己的cpu局限于intel的cpu，使得联想电脑和intel直接捆绑（耦合）在一起（**华为!!!**），这对于联想来说是致命的，因为联想电脑的cpu必须有多样的选择。



优化设计

对上面的代码进行优化：

为cpu定义一个接口，如下：

```
public class ComputerLenovo{  
    Cpu cpu = new Intel(); //电脑的cpu  
    public int calculate () {  
        return cpu.cal ();  
    }  
}
```

接口，
类似于国家
或行业标准

实现CPU接
口的Java类

虽然可以换了，但是耦合度太高！！

通过引入cpu接口，我们现在可以随意的更换cpu了，而不会将联想电脑和intel的cpu紧紧的耦合在一起。

上面的业务仍然还存在着耦合，即电脑同时依赖于cpu接口，和Intel对象。



耦合导致的问题

如果还有其他的电脑如：

```
public class ComputerAcer {  
    Cpu cpu = new Intel(); // 电脑的cpu  
    public int calculate () {  
        return cpu.cal ();  
    }  
}
```

很多其他厂商电脑，现在用的都是intel的cpu，假如有一天intel倒闭了，没有intel的cpu了，现在需要把cpu都换成AMD的cpu，但是如果现在有很多（成千上万）的电脑对象，就需要改成千上万次。所以以上的代码耦合度、依赖度太高。



进一步优化

把装配电脑cpu的权利交给第三方，姑且称为Factory,由Factory来充当装配工厂，装配电脑的各个组件，而不是把组件耦合到各个电脑中。改写代码如下：

```
public class ComputerLenovo{
    Cpu cpu;
    public void setCpu(Cpu cpu) {
        this.cpu=cpu;
    }
    public int calculate () {
        return cpu.cal();
    }
}
public class ComputerAcer {
    Cpu cpu;
    public void setCpu(Cpu cpu) {
        this.cpu=cpu;
    }
    public int calculate () {
        return cpu.cal();
    }
}
```



```
public class Factory{  
    ComputerLenovo computerLenovo = new ComputerLenovo();  
    computerLenovo.setCpu(new Intel());  
    computerLenovo.calculate();  
  
    ComputerAcer computerAcer= new ComputerAcer();  
    computerAcer.setCpu(new AMD());  
    computerAcer.calculate();  
}
```

可以根据需要灵活组装任意一个实例！！

上面代码把cpu的具体品牌从电脑中移除，只剩下了cpu的接口，可以通过Factory随意**装配任何cpu**。

这里的Factory就相当于spring容器：**它通过配置文件或者注解描述类和类之间的的依赖关系，自动完成类的初始化和依赖注入的工作，让开发者从类的初始化和依赖关系装配中解脱出来。**



进一步理解

一个疑问：到底是什么东西的控制被反转？

对应上面例子：

控制指的是联想电脑选择CPU组件的控制权；那么反转指的就是这一控制权被从联想电脑中移除，转交到第三方工厂中。

对于软件来说，就是某一接口具体实现类的选择控制权从调用类中移除，转交给第三方（如Spring IOC容器）决定。



Spring IOC容器



Spring IoC容器

- **Spring容器**

- Spring容器是生成Bean实例的工厂，并管理容器中的Bean。
- 在基于Spring的JavaEE应用中，所有组件都被当成Bean处理，包括数据源，Hibernate的SessionFactory和事务管理器。
- 在 Spring 容器读取 Bean 配置创建 Bean 实例前，必须对它进行**实例化**。只有在容器实例化后，才可以从 IOC 容器里获取 Bean 实例并使用。

- **Spring 提供了两种类型的 IOC 容器实现**

- BeanFactory: IOC 容器的基本实现，是 Spring 框架的基础设施，面向 Spring 本身；
- ApplicationContext: 提供了更多的高级特性。是 BeanFactory 的子接口。ApplicationContext 面向使用 Spring 框架的开发者，几乎所有的应用场合都直接使用 ApplicationContext 而非底层的 BeanFactory
- 无论使用何种方式，配置文件是相同的。



IOC容器——BeanFactory

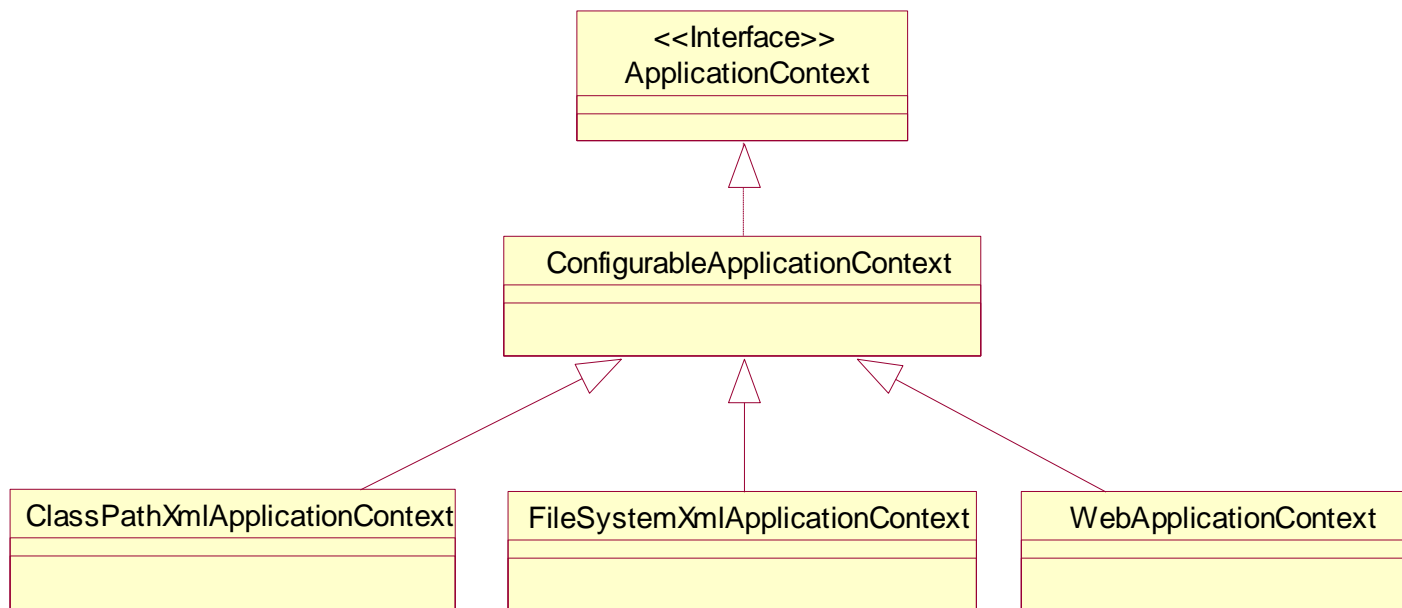
创建BeanFactory实例时，需要提供XML文件的绝对路径。例如：

```
public static void main(String[] args) {  
    //初始化Spring容器，加载配置文件  
    BeanFactory beanFac = new XmlBeanFactory(  
        new FileSystemResource("D:\\eclipse-  
        workspace\\ch1\\src\\applicationContext.xml"));  
    //通过容器获取test实例  
    TestDao tt = (TestDao)beanFac.getBean("test");  
    tt.sayHello();  
}
```



IOC容器——ApplicationContext

- **ConfigurableApplicationContext** 扩展于 **ApplicationContext**，新增加两个主要方法：**refresh()** 和 **close()**，让 **ApplicationContext** 具有启动、刷新和关闭上下文的能力。
- **ApplicationContext** 在初始化上下文时就实例化所有**单例**的 **Bean**。
- **WebApplicationContext** 是专门为 **WEB** 应用而准备的，它允许从相对于 **WEB** 根目录的路径中完成初始化工作。





IOC容器——ApplicationContext

创建ApplicationContext接口实例通常有三种方法：

- **ClassPathXmlApplicationContext:**
—从 类路径下加载配置文件
- **FileSystemXmlApplicationContext:**
— 从文件系统中加载配置文件
- **通过Web服务器实例化ApplicationContext容器**



ClassPathXmlApplicationContext从类路径classPath目录（src根目录）寻找指定的XML配置文件

```
public static void main(String[] args) {  
    //初始化Spring容器ApplicationContext, 加载配置文件  
    ApplicationContext appCon = new  
        ClassPathXmlApplicationContext("applicationContext.xml");  
    //通过容器获取test实例  
    TestDao tt = (TestDao)appCon.getBean("test");  
    tt.sayHello();  
}
```



FileSystemXmlApplicationContext从指定文件的绝对路径中寻找XML配置文件，找到并装载完成ApplicationContext的实例化工作。

```
public static void main(String[] args) {  
    //初始化Spring容器ApplicationContext，加载配置文件  
    ApplicationContext appCon = new FileSystemXmlApplicationContext("D:\\eclipse-  
workspace\\ch1\\src\\applicationContext.xml");  
  
    //通过容器获取test实例  
    TestDao tt = (TestDao)appCon.getBean("test");  
    tt.sayHello();  
}
```



Web服务器实例化ApplicationContext容器时，一般使用基于org.springframework.web.context.ContextLoaderListener的实现方式（需要将spring-web-5.0.2.RELEASE.jar复制到WEB-INF/lib目录中），此方法只需在web.xml中添加如下代码：

```
<context-param>
    <!-- 加载src目录下的applicationContext.xml文件 -->
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:applicationContext.xml
    </param-value>
</context-param>
<!-- 指定以ContextLoaderListener方式启动Spring容器 -->
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```



Spring依赖注入实现



依赖注入的方式

- **Spring 支持 3 种依赖注入的方式**
 - 构造方法注入
 - 属性 (setter) 注入
 - 工厂方法注入 (很少使用, 不推荐)

注：构造方法注入和属性注入都是基于XML的Bean装配方式。具体见Bean的装配方式。

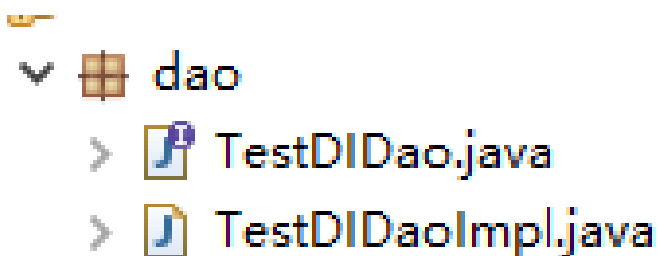


构造方法注入——步骤演示1

1. 创建dao

参见工程ch2

创建dao包，并在该包中创建TestDIDao接口和接口实现类TestDIDaoImpl。
创建dao的目的是在service中使用构造方法依赖注入TestDIDao接口对象。



```
package dao;
public interface TestDIDao {
    public void sayHello();
}
```

```
package dao;
import org.springframework.stereotype.Service;
@Service
public class TestDIDaoImpl implements
TestDIDao{
    @Override
    public void sayHello() {
        System.out.println("TestDIDao say: Hello,
Study hard!");
    }
}
```



构造方法注入——步骤演示2

2. 创建service

创建 service 包，并在该包中创建 TestDIService 接口和接口实现类 TestDIServiceImpl。**向** TestDIServiceImpl 中使用构造方法依赖注入 TestDIDao 接口对象。

▼ service

- > TestDIService.java
- > TestDIServiceImpl.java

```
package service;

public interface TestDIService {
    public void sayHello();
}
```

```
package service;
import dao.TestDIDao;
public class TestDIServiceImpl implements TestDIService{
    private TestDIDao testDIDao;
    //构造方法，用于实现依赖注入
    public TestDIServiceImpl(TestDIDao testDIDao) {
        super();
        this.testDIDao = testDIDao;
    }
    @Override
    public void sayHello() {
        //调用testDIDao中的sayHello方法
        testDIDao.sayHello();
        System.out.println("TestDIService 构造方法 注入 say: Hello, Study hard!");
    }
}
```

注入TestDIDao
接口对象



构造方法注入——步骤演示3

3. 编写配置文件

在src根目录下，创建Spring配置文件applicationContext.xml。在配置文件中，首先，将dao.TestDIDaoImpl类托管给Spring，让Spring创建其对象。其次，将service.TestDIServiceImpl类托管给Spring，让Spring创建其对象，同时给构造方法传递实参。

```
<?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">
    <!-- 将指定类TestDIDaoImpl配置给Spring，让Spring创建其实例 -->
    <bean id="myTestDIDao" class="dao.TestDIDaoImpl" />
    <!-- 使用构造方法注入 -->
    <bean id="testDIService" class="service.TestDIServiceImpl">
        <!-- 将myTestDIDao注入到TestDIServiceImpl类的属性 testDIDao上 -->
        <constructor-arg index="0" ref="myTestDIDao"/>
    </bean>
</beans>
```



构造方法注入——步骤演示4

4. 创建test

创建test包，并在该包中创建测试类TestDI，具体代码如下：

```
package test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import service.TestDIService;
public class TestDI {
    public static void main(String[] args) {
        //初始化Spring容器ApplicationContext，加载配置文件
        ApplicationContext appCon = new ClassPathXmlApplicationContext("applicationContext.xml");
        //通过容器获取testDIService实例，测试构造方法 注入
        TestDIService ts = (TestDIService)appCon.getBean("testDIService");
        ts.sayHello();
    }
}
```



构造方法注入——代码分析

以上代码到底哪里体现了依赖注入？与传统调用方式的区别是什么？

```
package service;
import dao.TestDIDao;
public class TestDIServiceImpl implements TestDIService{
    private TestDIDao testDIDao;
    //构造方法，用于实现依赖注入
    public TestDIServiceImpl(TestDIDao testDIDao) {
        super();
        this.testDIDao = testDIDao;
    }
    @Override
    public void sayHello() {
        //调用testDIDao中的sayHello方法
        testDIDao.sayHello();
        System.out.println("TestDIService 构造方法 注入 say: Hello, Stud");
    }
}
```

接口，可以传入任何实现该接口的具体类

通过 applicationContext.xml 文件配置指定，由Spring容器创建相应bean并建立依赖关系，可以根据需要灵活组配

传统调用通过代码定义两个类之间的依赖关系，每次只能指定一种依赖关系

```
package service;
import dao.TestDIDao;
import dao.TestDIDaoImpl;

public class TestDIServiceImpl2 implements TestDIService {

    private TestDIDaoImpl testDIDaoImpl;
    //构造方法，用于实现依赖注入
    public TestDIServiceImpl2(TestDIDaoImpl testDIDaoImpl) {
        super();
        this.testDIDaoImpl = testDIDaoImpl;
    }
    @Override
    public void sayHello() {
        //调用testDIDao中的sayHello方法
        testDIDaoImpl.sayHello();
        System.out.println("TestDIService 构造方法 注入 say: Hello, Study hard!");
    }
}
```

具体实现类，非接口无法兼容其他类型



构造方法注入——构造方法注入方式

- 通过构造方法注入Bean 的属性值或依赖的对象，它保证了 Bean 实例在实例化后就可以使用。
- 构造器注入在 `<constructor-arg>` 元素里声明属性, `<constructor-arg>` 中没有 name 属性
 - 按位置索引匹配入参
 - 按类型匹配入参
 - 位置和类型混合入参

参见工程SpringTest
constructionbeans.xml



依赖注入——属性setter方法注入方式

setter方法注入是Spring框架中最主流的注入方式，它利用Java Bean规范所定义的setter方法来完成注入，灵活且可读性高。setter方法注入，Spring框架也是使用Java的反射机制实现的。

- ▶ 属性注入时，先调用无参构造函数，再调用setter函数进行注入
- ▶ 属性注入：通过 setter 方法注入Bean 的属性值或依赖的对象
- ▶ 属性注入使用 `<property>` 元素, 使用 `name` 属性指定 Bean 的属性名称, `value` 属性或 `<value>` 子节点指定属性值，或者`ref`属性指定bean实例id;
- ▶ 属性注入是实际应用中最常用的注入方式

参见SpringTest工程
helloworldbeans.xml以及
Helloworld.HelloWorld.java
helloworld.main.java



属性setter方法注入——步骤演示

参见工程ch2

1. 创建dao

过程同构造方法注入

2. 创建接口实现类TestDIServiceImpl1

改写接口实现类TestDIServiceImpl1,
创建接口实现类TestDIServiceImpl1,
在TestDIServiceImpl1中使用属性
setter方法依赖注入TestDIDao接口对
象。

```
package service;

import dao.TestDIDao;

public class TestDIServiceImpl1 implements TestDIService{
    private TestDIDao testDIDao;
    //添加testDIDao属性的setter方法，用于实现依赖注入
    public void setTestDIDao(TestDIDao testDIDao) {
        this.testDIDao = testDIDao;
    }
    @Override
    public void sayHello() {
        //调用testDIDao中的sayHello方法
        testDIDao.sayHello();
        System.out.println("TestDIService setter方法注入 say: Hello, Study hard!");
    }
}
```



属性setter方法注入——步骤演示

3. 将TestDIServiceImpl1类托管给Spring

将service.TestDIServiceImpl1类托管给Spring，让Spring创建其对象。同时，调用TestDIServiceImpl1类的setter方法完成依赖注入。在配置文件添加如下代码：

```
<!-- 使用setter方法注入 -->
```

```
<bean id="testDIService1" class="service.TestDIServiceImpl1">
```

```
<!-- 调用TestDIServiceImpl1类的setter方法，将myTestDIDao注入到  
TestDIServiceImpl1类的属性testDIDao上-->
```

```
<property name="testDIDao" ref="myTestDIDao"/>
```

```
</bean>
```



属性setter方法注入——步骤演示

4. 在test中测试setter方法注入

在主类中，添加如下代码测试setter方法注入：

```
//通过容器获取testDIService实例，测试setter方法注入  
TestDIService ts1 = (TestDIService)appCon.getBean("testDIService1");  
ts1.sayHello();
```




使用属性setter注入的参数设置

- 字面值
- 引用其它Bean
- 内部Bean
- 集合值

参见SpringTest工程
helloworld.Main.java及
personaxebeans.xml、 teachers.xml、
helloworldbeans.xml



字面值

- 字面值：可用字符串表示的值，可以通过 `<value>` 元素标签或 `value` 属性进行注入。
- 基本数据类型及其封装类、`String` 等类型都可以采取字面值注入的方式
- 若字面值中包含特殊字符，可以使用 `<![CDATA[]]>` 把字面值包裹起来。

参见SpringTest工程
helloworldbeans.xml



引用其它 Bean

- 组成应用程序的 Bean 经常需要相互协作以完成应用程序的功能. 要使 Bean 能够相互访问, 就必须在 Bean 配置文件中指定对 Bean 的引用
- 在 Bean 的配置文件中, 可以通过 `<ref>` 元素或 `ref` 属性为 Bean 的属性或构造器参数指定对 Bean 的引用.
- 也可以在属性或构造器里包含 Bean 的声明, 这样的 Bean 称为内部 Bean

示例: SpringTest工程personaxebeans.xml



内部 Bean(嵌套Bean , 匿名Bean)

- 当 Bean 实例**仅仅**给一个特定的属性使用时, 可以将其声明为内部 Bean. 内部 Bean 声明直接包含在 `<property>` 或 `<constructor-arg>` 元素里, 不需要设置任何 id 或 name 属性
- 内部 Bean 不能使用在任何其他地方

示例: SpringTest工程personaxebeans.xml



集合值

- 类属性为集合类型，如List,Map,Set,Properties
- 注入时需要对应使用 **<list.../>**, **<map.../>**, **<set.../>**, **<props.../>**
- **<list.../>**, **<set.../>**
 - 元素可以是字面值value,引用ref,嵌套bean以及其它集合元素
- **<props.../>**
 - Key和value只能是字符串
- **<map.../>**
 - Key或者key-ref
 - Value或者value-ref

**示例：SpringTest工程
impl.UniversityTeacher.java和teachers.xml**



Spring中Bean和普通JavaBean的对比

- **传统Java应用中的JavaBean的作用**
 - 用于封装对象，需要setter和getter方法
 - 用于在各层之间传递数据，不受容器和框架管理
- **Spring中Bean**
 - 可以是任何java组件或者java对象，最多需要setter方法
 - 被Spring容器创建和管理

示例：SpringTest工程

helloworld.DataSourceBean.java和datasourcebean.xml



小结

- Spring的本质是通过XML配置文件来驱动Java代码创建对象
- 创建类对象时
 - 无参数→调用无参构造函数
 - 有参数→①调用无参构造函数②调用setter函数进行参数注入
 - 或者有参数→调用有参构造器
- 参数类型
 - 所有类型都可以传入一个Java对象，使用ref引用或者使用内部Bean
 - 基本类型如String和Date等，还可以使用value指定字面值



小结

- **Setter方法与构造方法注入的对比：**
 - **setter注入的缺点是无法准确表达哪些属性是必须的，哪些是可选的；**
 - **构造方法注入的优势是通过构造强制依赖关系，不可能实例化不完全的或无法使用的bean**



Spring Bean的装配方式



Spring Bean的装配方式

——bean注入到容器的方式

Bean的装配可以理解为将Bean依赖注入到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。

基于XML的装配通常采用两种实现方式，即前面讲的构造方法注入和属性（setter）注入。



Spring Bean的装配方式

1 基于XML配置的装配

由前面可以知道Spring提供了两种基于XML配置的装配方式：
构造方法注入和属性setter方法注入。

通过以下步骤来实现基于XML配置的装配方式。

1. 创建Bean的实现类
2. 配置Bean
3. 测试基于XML配置的装配方式

见ch3工程代码applicationContext.xml、assemble.ComplexUser



Spring Bean的装配方式

2 基于注解的装配

在Spring框架中定义了一系列的注解，常用注解如下所示。

1. **@Component**
2. **@Repository**
3. **@Service**
4. **@Controller**
5. **@Autowired**
6. **@Resource**

下面首先通过一个实例讲解@Component()，直观理解注解的装配过程

ch3工程

annotationContext.xml、 annotation.AnnotationUser



@Component

该注解是一个泛化的概念，仅仅表示一个组件对象（Bean），可以作用在任何层次上。

(1) 创建Bean的实现类

@Component()

/** 相当于 @Component("annotationUser") 或 @Component(value = "annotationUser"), annotationUser为Bean的id，默认为首字母小写的类名**/

```
public class AnnotationUser {
```

```
    @Value("chenheng")//只注入了简单的值，复杂值的注入目前使用该方式还解决不了
```

```
    private String uname;
```

```
    /**省略setter和getter方法**/
```

```
}
```



(2) 配置注解

现在有了Bean的实现类，但还不能进行测试，因为Spring容器并不知道去哪里扫描Bean对象。需要在配置文件中配置注解，注解配置方式如下：

```
<context:component-scan base-package="Bean所在的包路径"/>
```

<!-- 使用context命名空间，通过Spring扫描指定包annotation及其子包下所有Bean的实现类，进行注解解析,也称为组件扫描 -->

```
<context:component-scan base-package="annotation"/>
```



(3) 测试Bean实例

```
ApplicationContext appCon = new  
ClassPathXmlApplicationContext("annotationContext.xml");  
AnnotationUser au = (AnnotationUser)appCon.getBean("annotationUser");  
System.out.println(au.getUname());
```

注：

1、在Spring 4.0以上版本，配置注解指定包中的注解进行扫描前，需要事先导入Spring AOP的JAR包spring-aop-5.0.2.RELEASE.jar。

2、context:annotation-config与context:component-scan都可以用于配置注解，但前者对于@Component、@Controller、@Service等注解不能激活，只能使用后者。所以更推荐使用后者。具体区别可见https://blog.csdn.net/fox_bert/article/details/80793030



test

- Sum.java
- TestAnnotation.java**
- TestAssemble.java
- TestInstance.java
- TestLife.java
- TestMoreAnnotation.java

```
4 xmlns:context="http://www.springframework.org/schema/context"
5 xsi:schemaLocation="http://www.springframework.org/schema/beans
6 http://www.springframework.org/schema/beans/spring-beans.xsd
7 http://www.springframework.org/schema/context
8 http://www.springframework.org/schema/context/spring-context.xsd">
9 <!-- 使用context命名空间，通过Spring扫描指定包下所有Bean的实现类，进行注解解析 -->
10 <context:component-scan base-package="annotation"/>
11 </beans>
```

根据注解默认装配以驼峰
规则命名的同名java 类

```
1 package annotation;
2 import org.springframework.beans.factory.annotation.Value;
3
4 @Component()//相当于@Component("annotationUser")或@Component
5 public class AnnotationUser {
6     @Value("chenheng")
7     private String uname;
8
9     public String getUname() {
10         return uname;
11     }
12     public void setUname(String uname) {
13         this.uname = uname;
14     }
15     @Override
16     public String toString() {
17         return "uname=" + uname;
18     }
19 }
20
```




Spring Bean的装配方式

2 基于注解的装配——①常见注解

(1) **@Component** : 描述Spring中的Bean, 是一个泛化概念, 仅仅表示一个组件, 可以作用在任何层次;

(2) **@Repository**: 描述数据访问层的Bean, 功能与@Component相同;

(3) **@Service**: 描述业务层的Bean, 功能与@Component相同;

(4) **@Controller**: 描述控制层的Bean, 功能与@Component相同;

(5) **@Autowired**: 用于**装配属性变量**, 属性的set方法以及构造方法, 配合对应的注解处理器完成Bean的自动配置, 默认按照 Bean 的类型进行装配;

(6) **@Resource**: 作用与@Autowired一样。其区别在于 **@Autowired默认按照 Bean的类型装配**, **@Resource默认按照 Bean 的实例名称进行装配**。@Resource中有两个重要属性: name和type, 如果指定name, 则按实例名称进行装配;如果指定type属性, 则按 Bean类型进行装配; 如果都不指定, 则先按实例名称装配, 若不能匹配, 再按照 Bean 类型进行装配;如果都无法匹配, 则抛出异常NoSuchBeanDefinitionException;

(7) **@Qualifier**: 与@Autowired注解配合使用, 会将默认的按类型装配修改为按 Bean 的实例名称装配, Bean 的实例名称由@Qualifier注解的参数指定。



Spring Bean的装配方式

2 基于注解的装配——①常见注解

(1) @Component

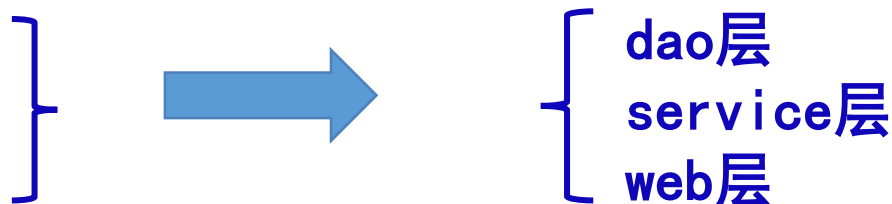
@Component等价于<bean class="">

@Component("id") 等价于 <bean id="" class="">

(2) @Repository

(3) @Service

(4) @Controller



以上三个注释用于web开发，是@Component的衍生注解，具有和@Component相同的用法，只是表示不同的用途，用于区分MVC层。

(5) @Autowired

(6) @Resource

(7) @Qualifier

用于属性装配



Spring Bean的装配方式

2 基于注解的装配——②给属性注入值

普通值: @Value("")

引用值: 方式1: 按照【类型】注入

 @Autowired

 方式2: 按照【名称】注入1

 @Autowired

 @Qualifier("名称")

 方式3: 按照【名称】注入2

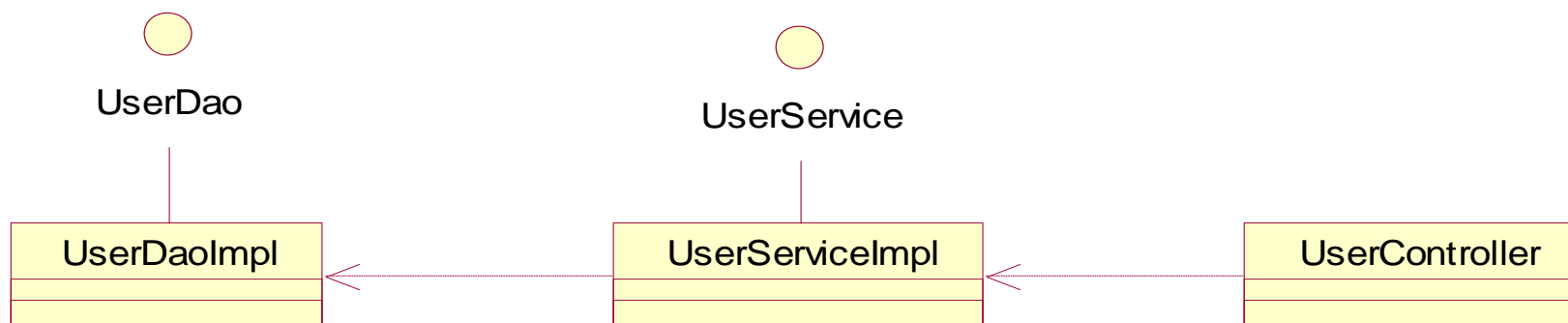
 @Resource("名称")



Spring Bean的装配方式

2 基于注解的装配——③综合实例

见SpringTest工程下annotation包



(1) 分别创建接口UserDao和实现类UserDaoImpl

使用@Repository注解将在一个UserDaoImpl类标识为 Spring 中的Bean, 相当于配置文件中
<bean id = "userDao" class = "com.annotation.UserDaoImpl" />

(2) 分别创建接口UserService和实现类UserServiceImpl

首先使用@Service注解将UserServiceImpl类标识为Spring中的Bean, 相当于配置文件中
<bean id = "userService" class = "com.annotation.UserServiceImpl" />;然后使用
@Resource注解标注在属性userDao上, 相当于配置文件中
< property name = "userDao" ref = "userDao" />。



Spring Bean的装配方式

2 基于注解的装配——③综合实例

见SpringTest工程下annotation包

```
package annotation;
import org.springframework.stereotype.Repository;
@Repository("userDao")
public class UserDaoImpl implements UserDao {
    public void save() {
        System.out.println("user... save");
    }
}
```

注入

```
package annotation;
import javax.annotation.Resource;
@Service("userService")
public class UserServiceImpl implements UserService {
    @Resource(name="userDao")
    private UserDao userDao;
    @Override
    public void save() {
        //调用userDao中的save方法
        this.userDao.save();
        System.out.println("userservice...save...");
    }
}
```



Spring Bean的装配方式

2 基于注解的装配——③综合实例

(3) 创建控制器类UserController

首先使用@Controller注解标注UserController类，这相当于在配置文件中编写

```
<bean id = "userController" class = "com.annotation.UserController" />;
```

然后使用@Resource注解标注在userService属性中，这相当于在配置文件中编写

```
<property name = "userService" ref = "userService" />;
```



Spring Bean的装配方式

2 基于注解的装配——③综合实例

(4) 创建配置文件beans.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xmlns:context="http://www.springframework.org/schema/context"
5       xsi:schemaLocation="http://www.springframework.org/schema/beans
6       http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
7       http://www.springframework.org/schema/context
8       http://www.springframework.org/schema/context/spring-context.xsd
9       ">
10
11   <!-- 使用context命名空间，在配置文件中开启相应的注解处理器 -->
12   <context:annotation-config />
13   <!-- 分别定义3个bean实例 -->
14   <bean id="userDao" class="annotation.UserDaoImpl" />
15   <bean id="userService" class="annotation.UserServiceImpl" />
16   <bean id="userController" class="annotation.UserController" />
17
```

- 增加了第4行，第7行和第8行中包含有上下文的约束信息；
- 配置<context: annotation-config />来开启注解处理器；
- 分别定义3个Bean对应所编写的3个实例。与XML装备方式有所不同的是，这里不再需要配置子元素<property>。



Spring Bean的装配方式

2 基于注解的装配——③综合实例

(4) 创建配置文件beans.xml

上述 Spring 配置文件中高端注解方式虽然较大程度简化了XML文件中的Bean的配置，但仍需要在 Spring 配置文件中——配置相应的 Bean，为此 Spring 注释提供了另外一种高效的注解配置方式（对包路径下的所有 Bean 文件进行扫描）。

```
<context:component-scan base-package="Bean 所在的包路径" />
```

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:context="http://www.springframework.org/schema/context"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
7     http://www.springframework.org/schema/context
8     http://www.springframework.org/schema/context/spring-context.xsd
9     ">
10
11     <!-- 使用context:annotation-config -->
12     <context:component-scan base-package="annotation" />
13
14 </beans>
```




Spring Bean的装配方式

2 基于注解的装配——③综合实例

(5) 创建测试类AnnotationAssembleTest

```
6 public class AnnotationAssembleTest {
7     public static void main(String[] args) {
8         //加载配置文件
9         ApplicationContext ac=new ClassPathXmlApplicationContext("annotation/beans.xml");
10        //获取UserController实例
11        UserController userController=(UserController) ac.getBean("userController");
12        //调用UserController中的save()方法
13        userController.save();
14    }
}
```

```
1 package annotation;
2 import org.springframework.stereotype.Repository;
3 @Repository("userDao")
4 public class UserDaoImpl implements UserDao {
5     public void save() {
6         System.out.println("user... save");
7     }
8 }
```

```
1 package annotation;
2 import javax.annotation.Resource;
3
4 @Controller("userController")
5 public class UserController {
6     @Resource(name="userService")
7     private UserService userService;
8     public void save() {
9         this.userService.save();
10        System.out.println("usercontroller... save...");
11    }
12 }
13
```

```
1 package annotation;
2 import javax.annotation.Resource;
3
4 @Service("userService")
5 public class UserServiceImpl implements UserService {
6     @Resource(name="userDao")
7     private UserDao userDao;
8     @Override
9     public void save() {
10        //调用userDao中的save方法
11        this.userDao.save();
12        System.out.println("userservice...save...");
13    }
14 }
```

注意： Spring 4.0以上版本使用上面的代码对指定包中的注解进行扫描前，需要先向项目中导入Spring AOP的JAR包spring-aop-4.3.6.RELEASE.jar，否则程序在运行时会报出“java.lang.NoClassDefFoundError: org/springframework/aop/TargetSource”错误



Spring Bean的装配方式

3 自动装配

<bean id= "... " class= "... " autowire= "autowire type" >

有以下几种自动装配类型：

- 1.**byName**:寻找和属性名相同的bean,若找不到，则装不上。
- 2.**byType**:寻找和属性类型相同的bean,找不到,装不上,找到多个抛异常。
- 3.**constructor**:查找和bean的构造**参数**一致的一个或多个bean，若找不到或找到多个，抛异常。按照参数的类型装配
- 4.**autodetect**:(3)和(2)之间选一个方式。不确定性的处理与(3)和(2)一致。
- 5.**default**:这个需要在<beans default-autowire= “指定” />
- 6.**no**:不自动装配，这是autowrite的默认值.



Spring Bean的装配方式

3 自动装配

参见SpringTest工程中的autowire包实例

注：如果原bean下相应的属性已经被property配置，则优先读取property下的值，只有当property不存在时才会根据autowire的设置来读取值



Spring Bean的装配方式

3 自动装配——byName

```
public class Master {  
    private String masterName;  
    private Dog myDog;
```

```
public class Dog {  
    private String dogName;  
    private int age;
```

自动为master1的myDog属性根据名称与配置文件中id为myDog的实例进行匹配

```
<bean id="master1" class="autowire.Master" autowire="byName">  
    <property name="masterName" value="韩梅梅" />  
</bean>
```

```
<!-- 配置dog对象 -->  
<bean id="myDog" class="autowire.Dog">  
    <property name="dogName" value="汪汪" />  
    <property name="age" value="3" />  
</bean>
```

```
Master master=(Master)ctx.getBean("master1");  
System.out.println(master.getMasterName()+" 养的狗叫 "+master.getMyDog().getDogName()+"， 今年"+master.getMyDog().getAge()+" 岁!");
```



Spring Bean的装配方式

3 自动装配——byType

```
public class Master {  
    private String masterName;  
    private Dog myDog;
```

```
public class Dog {  
    private String dogName;  
    private int age;
```

自动为master2的myDog属性根据类型与配置文件中类型为autowire.Dog的实例进行匹配

```
<bean id="master2" class="autowire.Master" autowire="byType">  
    <property name="masterName" value="李雷" />  
</bean>
```

```
<!-- 配置dog对象 -->  
<bean id="myDog" class="autowire.Dog">  
    <property name="dogName" value="汪汪" />  
    <property name="age" value="3" />  
</bean>
```

```
Master master2=(Master)ctx.getBean("master2");  
System.out.println(master2.getMasterName()+" 养的狗叫 "+master2.getMyDog().getDogName()+"， 今年"+master2.getMyDog().getAge()+" 岁!");
```



Spring Bean的装配方式

3 自动装配——constructor

```
public class Master {  
    private String masterName;  
    private Dog myDog;
```

* 专门为constructor自动匹配类型创建的构造函数

```
* @param dog  
*/  
public Master(Dog dog) {  
    this.myDog=dog;  
}
```

```
public class Dog {  
    private String dogName;  
    private int age;
```

自动为master3的构造函数中类型与配置文件中类型为autowire.Dog一致的实例进行匹配

```
<bean id="master3" class="autowire.Master" autowire="constructor">  
    <property name="masterName" value="小明" />  
</bean>
```

```
<!-- 配置dog对象 -->  
<bean id="myDog" class="autowire.Dog">  
    <property name="dogName" value="汪汪" />  
    <property name="age" value="3" />  
</bean>
```

```
Master master3=(Master)ctx.getBean("master3");  
System.out.println(master3.getMasterName()+" 养的狗叫 "+master3.getMyDog().getDogName()+"， 今年"+master3.getMyDog().getAge()+" 岁!");
```



Bean的作用域与生命周期



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，容器都将创建一个新的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。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。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。仅在Web Spring应用程序上下文中使用。
websocket	为每个WebSocket对象创建一个Bean实例。仅在Web Spring应用程序上下文中使用。



Spring Bean的作用域

- **Singleton**。在Spring中取得的实例被默认为Singleton(单例)

```
<bean id= "sample" class= "com.service.impl.SampleImpl " scope= "singleton" />
```

```
<bean id= "sample" class= "com.service.impl.SampleImpl " singleton= "true" />
```

- **Prototype**。在每次对该bean请求时创建出一个新的bean对象(原型)

```
<bean id= "sample" class= "com.service.impl.SampleImpl " scope= "prototype" />
```

- 其他作用域:

request

session

global session



主要用于web开发

SpringTest工程

helloworld.Main.java及scopebeans.xml.



Bean的生命周期

Bean的生命周期整个过程如下：

- 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，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()方法，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 setApplicationContext(ApplicationContext) 方法，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 postProcessBeforeInitialization(Object obj, String s)对Bean进行操作。**



7. 如果Bean实现了InitializingBean接口，将调用afterPropertiesSet()方法。

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。

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After方法，也可用于内存或缓存技术。

以上工作（1至9）完成以后就可以使用该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。



下面通过一个实例演示Bean的生命周期。

ch3工程life包

1. 创建Bean的实现类

在life包下创建类BeanLife。在类BeanLife中有两个方法，一个演示初始化过程，一个演示销毁过程。

```
package life;  
public class BeanLife {  
    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");  
    }  
}
```



2. 配置Bean

在Spring配置文件中，使用实现类BeanLife配置一个id为beanLife的Bean。
具体代码如下：

<!-- 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"/>**



3. 测试生命周期

ch3工程test.TestLife.java

```
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执行，这里使用ClassPathXmlApplicationContext实现类声明容器  
        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  
    }  
}
```

```
<terminated> TestLife [Java Application] F:\java\jdk\bin\javaw.exe (2020年11月22日 下午7:53:30)  
十一月 22, 2020 7:53:30 下午 org.springframework.context.support.ClassPathXmlAp  
信息: Refreshing org.springframework.context.support.ClassPathXmlAppl  
十一月 22, 2020 7:53:30 下午 org.springframework.beans.factory.xml.XmlBeanDefin  
信息: Loading XML bean definitions from class path resource [application  
life.BeanLife执行自定义的初始化方法  
获得对象前  
获得对象后life.BeanLife@167fdd33  
十一月 22, 2020 7:53:30 下午 org.springframework.context.support.ClassPathXmlAp  
信息: Closing org.springframework.context.support.ClassPathXmlAppl  
life.BeanLife执行自定义的销毁方法
```



Spring AOP



Spring AOP (面向切面编程)

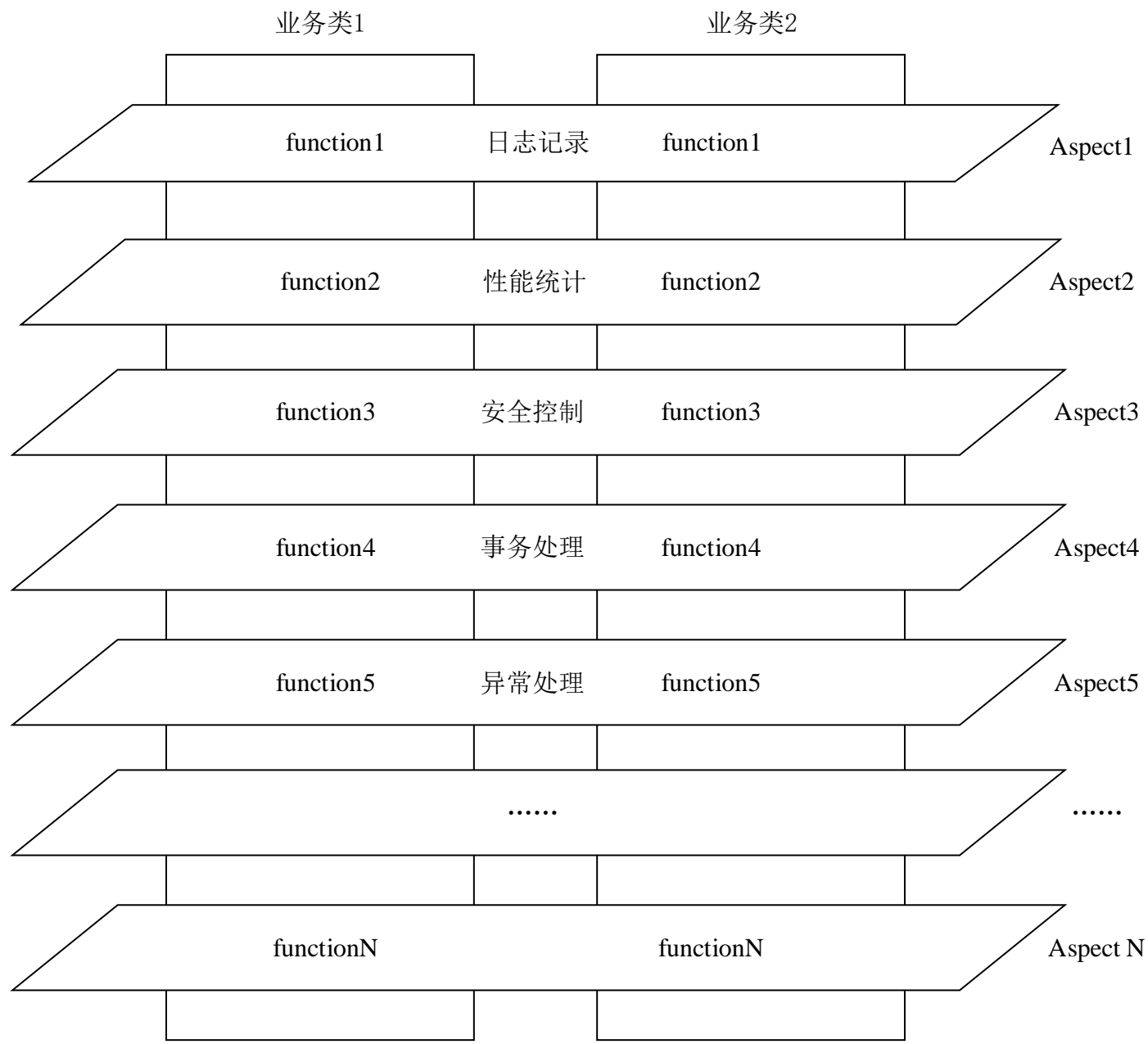
AOP的概念

要理解切面编程，就需要先理解什么是切面：

- ✓ **用刀把一个西瓜分成两瓣，切开的切口就是切面；**
- ✓ **炒菜，锅与炉子共同来完成炒菜，锅与炉子就是切面；**
- ✓ **web层级设计中，web层->网关层->服务层->数据层，每一层之间也是一个切面；**
- ✓ **编程中，对象与对象之间，方法与方法之间，模块与模块之间都是一个个切面。**



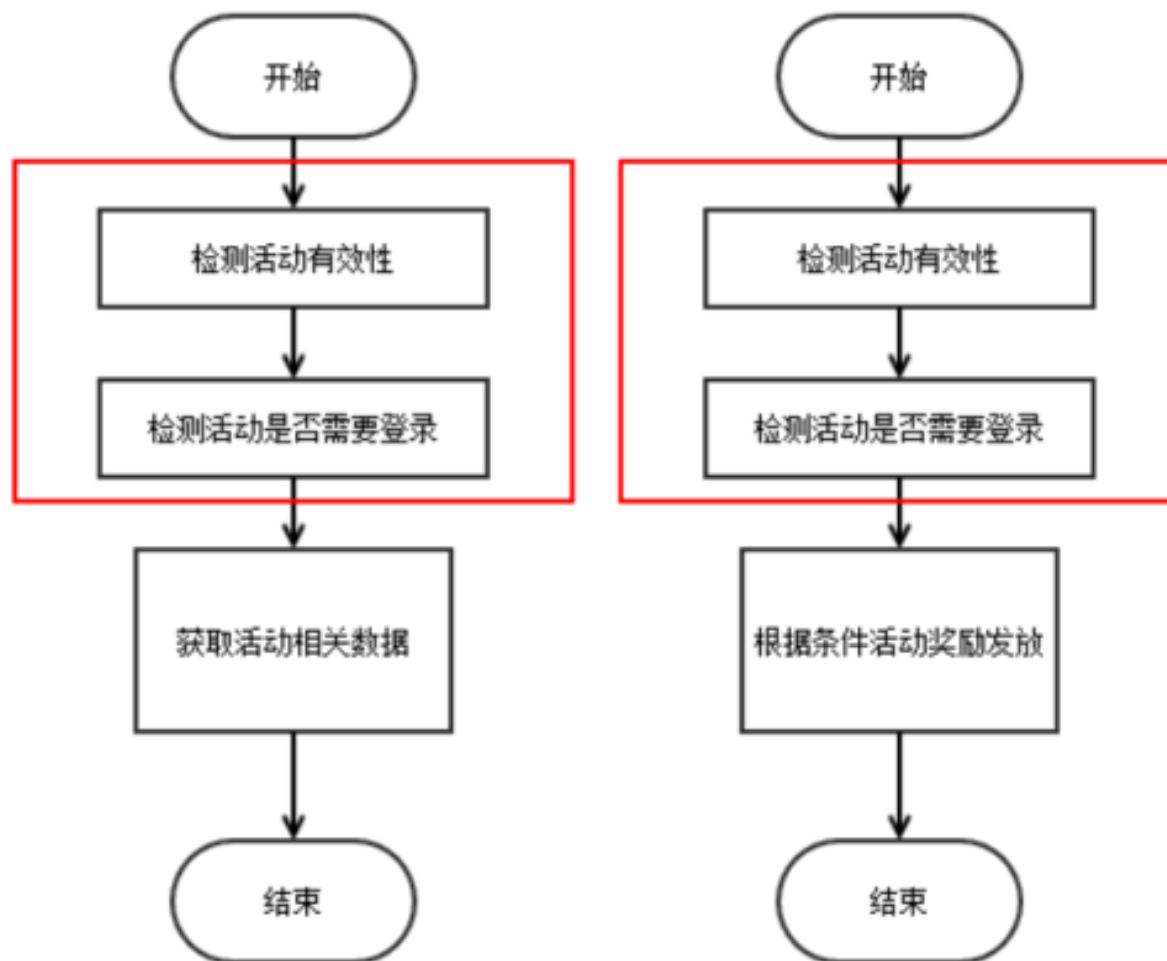
AOP的概念





AOP的概念

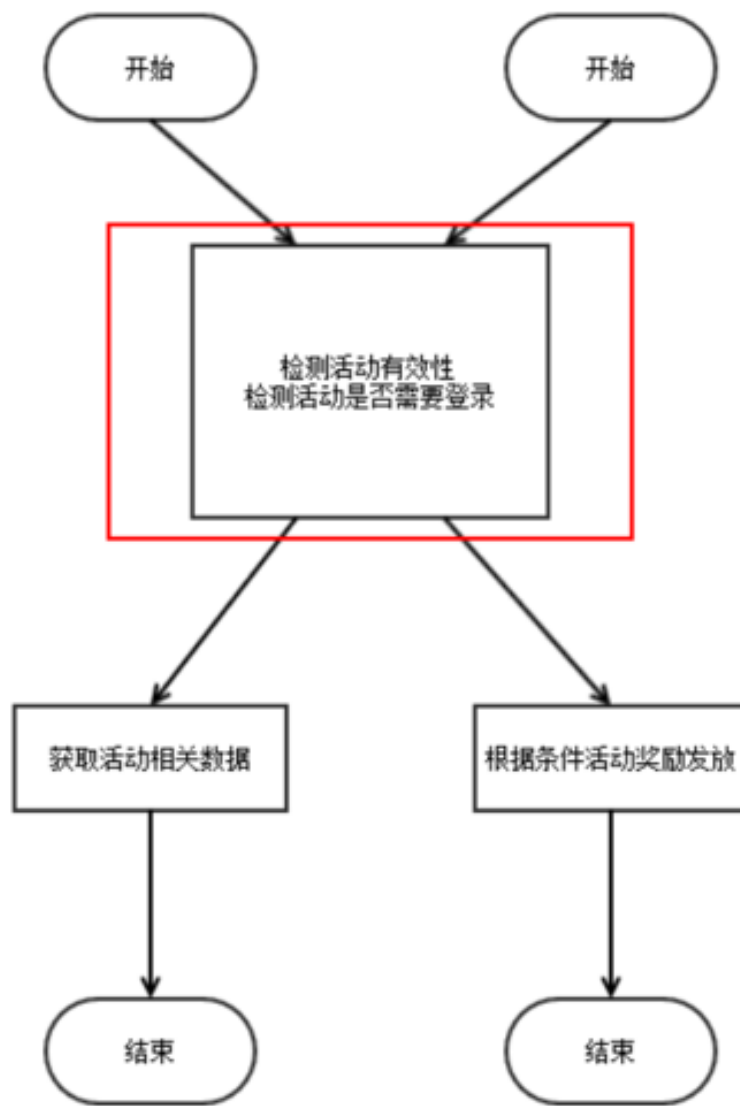
一般做活动的时候，一般对每一个接口都会做活动的有效性校验（是否开始、是否结束等等）、以及这个接口是不是需要用户登录。按照正常的逻辑，可以这么做。





AOP的概念

有个问题就是，有多少接口，就要多少次代码copy。对于一个“懒人”，这是不可容忍的。好，提出一个公共方法，每个接口都来调用这个接口。这里有点切面的味道了。

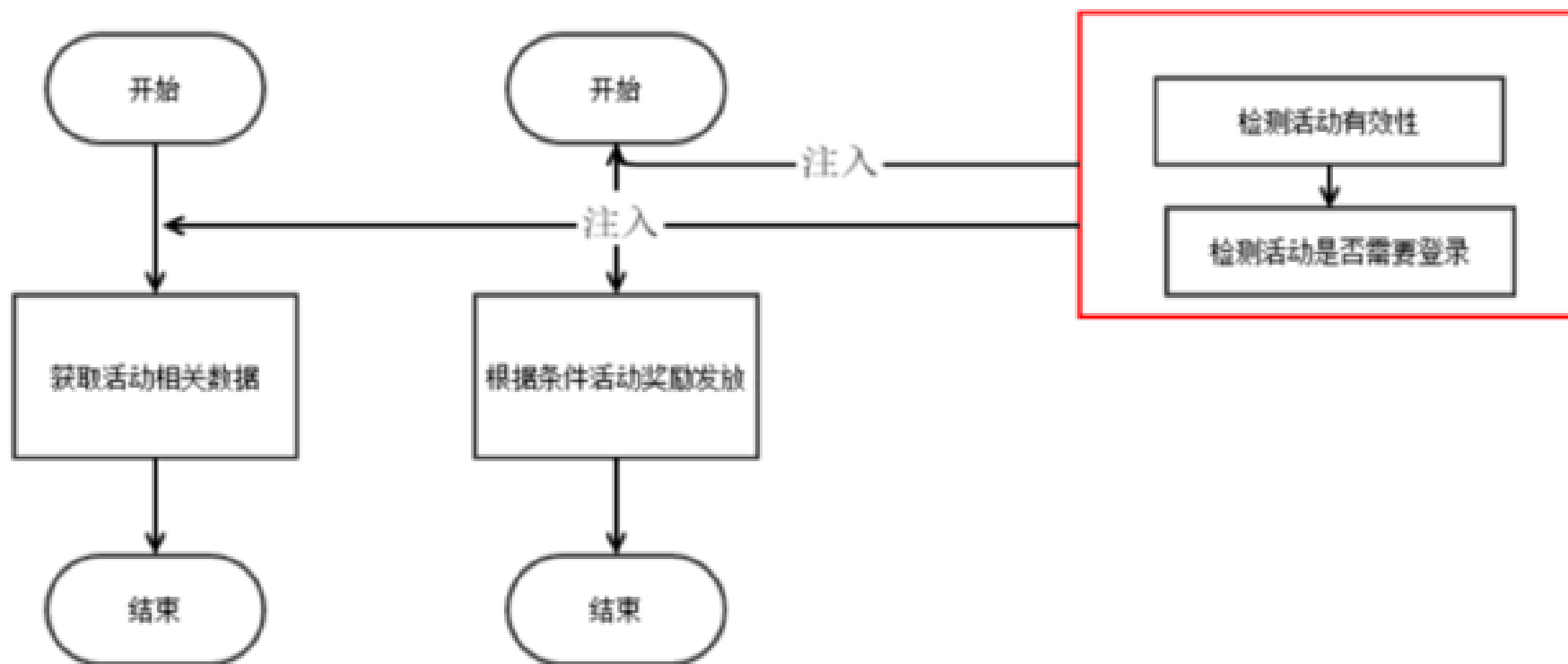




AOP的概念

引入新的问题：虽然不用每次都copy代码了，但是，每个接口都要调用这个方法，增加了代码的耦合性。于是就有了切面的概念，即将方法注入到接口调用的某个地方（切点）。

获取某个活动的数据，根据条件发放奖励





AOP的术语

1. 切面

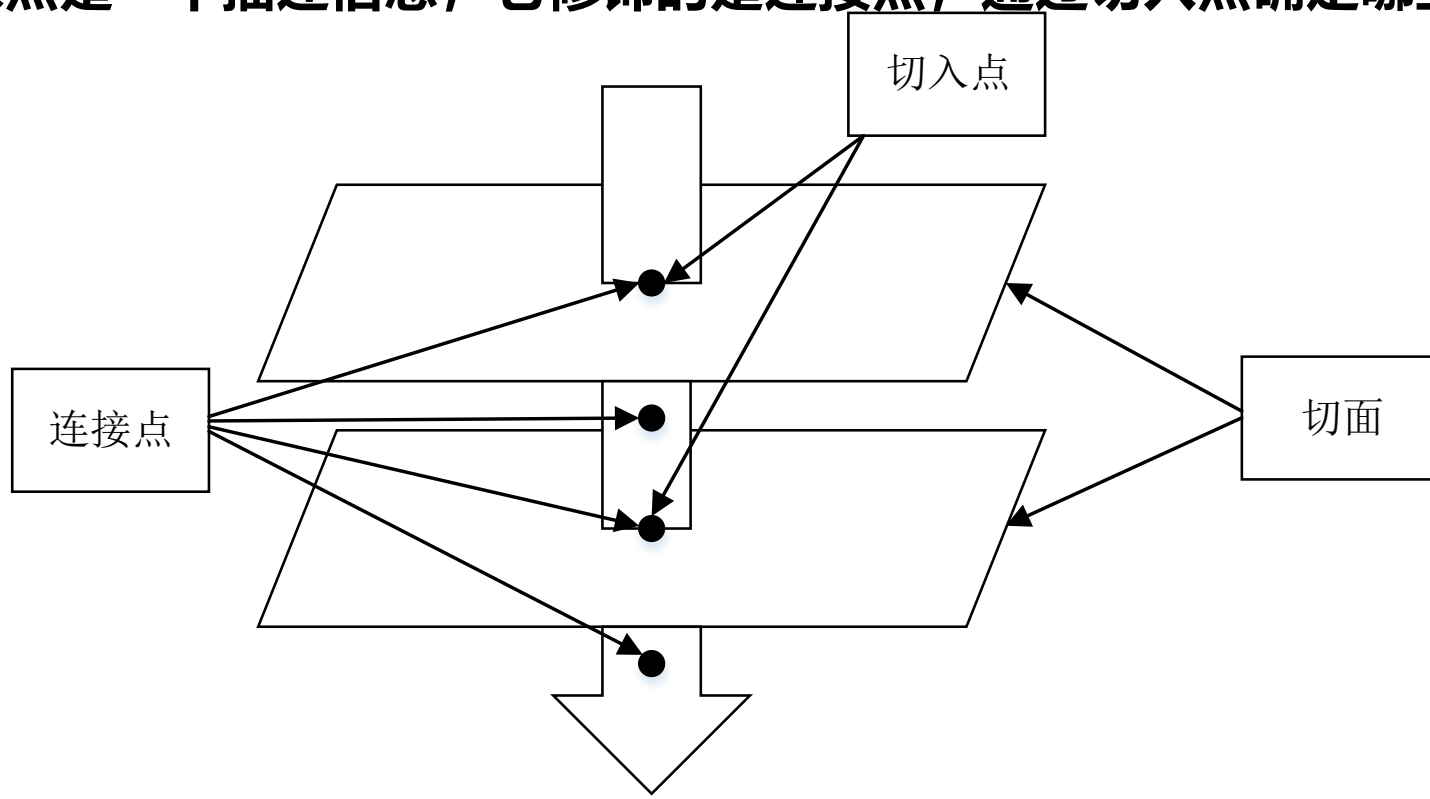
切面 (Aspect) 是指封装横切到系统功能 (如事务处理) 的类。

2. 连接点

连接点 (Joinpoint) 是指程序运行中的一些时间点, 如方法的调用或异常的抛出。

3. 切入点

切入点 (Pointcut) 是指那些需要处理的连接点。在Spring AOP 中, 所有的方法执行都是连接点, 而切入点是一个描述信息, 它修饰的是连接点, 通过切入点确定哪些连接点需要被处理。





AOP的术语

4. 通知 (增强处理)

由切面添加到特定的连接点（满足切入点规则）的一段代码，即在定义好的切入点处所要执行的程序代码。可以将其理解为切面开启后，切面的方法。因此，通知是切面的具体实现。

5. 引入

引入 (Introduction) 允许在现有的实现类中添加自定义的方法和属性。

6. 目标对象

目标对象 (Target Object) 是指所有被通知的对象。如果AOP 框架使用运行时代理的方式（动态的AOP）来实现切面，那么通知对象总是一个代理对象。

7. 代理

代理 (Proxy) 是通知应用到目标对象之后，被动态创建的对象。

8. 组入

组入 (Weaving) 是将切面代码插入到目标对象上，从而生成代理对象的过程。根据不同的实现技术，AOP织入有三种方式：编译器织入，需要有特殊的Java编译器；类装载期织入，需要有特殊的类装载器；动态代理织入，在运行期为目标类添加通知生成子类的方式。Spring AOP框架默认采用动态代理织入，而AspectJ（基于Java语言的AOP框架）采用编译器织入和类装载期织入。



AOP基本原则：

在不增加代码的基础上增加新的功能



动态代理实现AOP

使用JDK动态代理实现Spring AOP

见ch4工程

1. 创建应用

创建一个名为ch4的Web应用，并导入所需的JAR包。

2. 创建接口及实现类

在ch4的src目录下，创建一个dynamic.jdk包，在该包中创建接口TestDao和接口实现类TestDaoImpl（**目标对象**）。该实现类作为目标类，在代理类中对其方法进行增强处理。

3. 创建切面类

在ch4的src目录下，创建一个aspect包，在该包中创建切面类MyAspect，在该类中可以定义多个通知（增强处理的功能方法）。



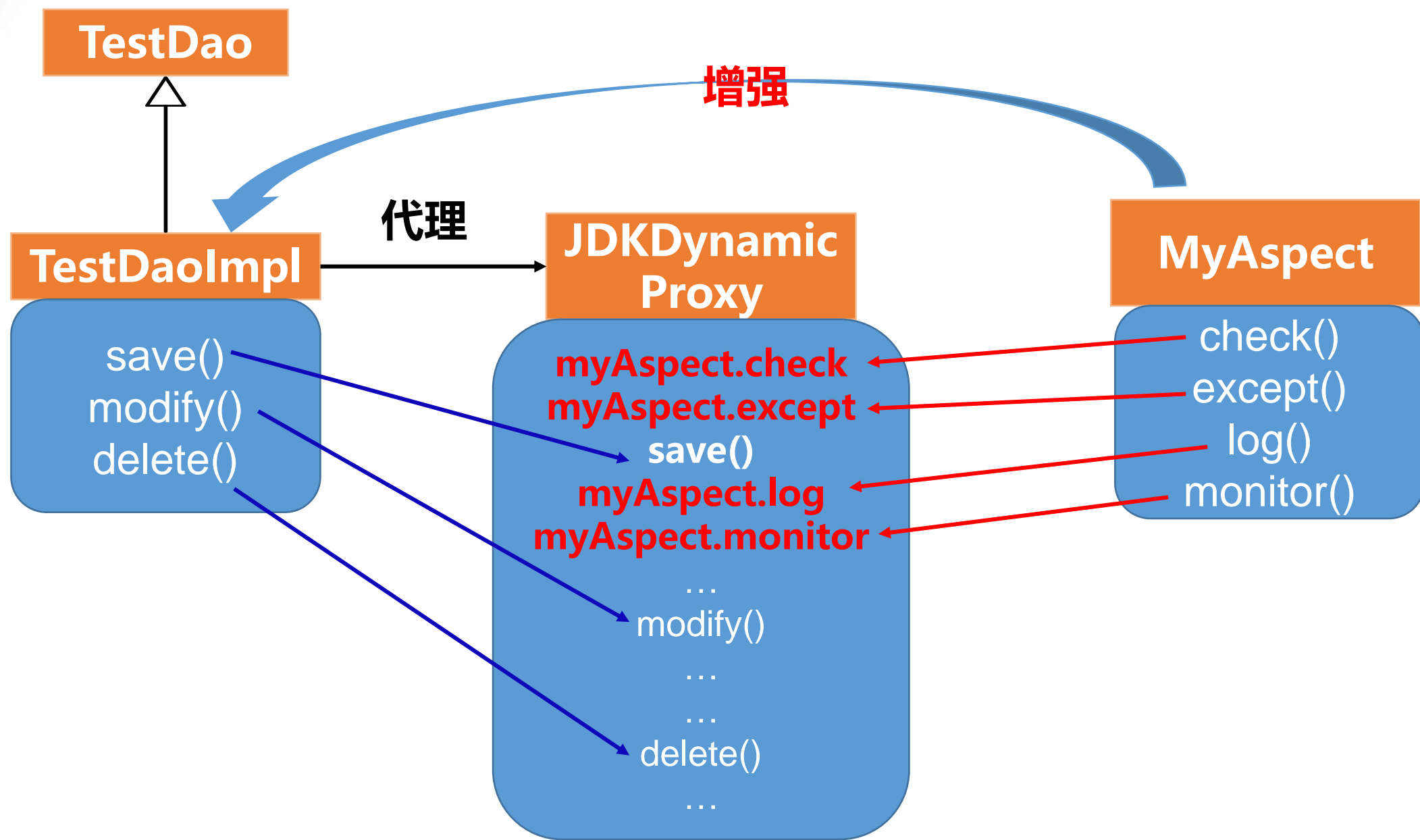
动态代理实现AOP

4. 创建代理类

在dynamic.jdk包中，创建代理类JDKDynamicProxy。在JDK动态代理中，代理类必须实现java.lang.reflect.InvocationHandler接口，并编写代理方法。在代理方法中，需要通过Proxy实现动态代理。

5. 创建测试类

在dynamic.jdk包中，创建测试类JDKDynamicTest。在主方法中创建代理对象和目标对象，然后从代理对象中获取对目标对象增强后的对象，最后调用该对象的添加、修改和删除方法。





通过Spring ProxyFactoryBean实现AOP

(1) 导入相关JAR包

见ch4工程

spring.proxyfactorybean包

在核心JAR包基础上，需要再向ch4应用的/WEB-INF/lib目录下导入JAR包spring-aop-5.0.2.RELEASE.jar和aopalliance-1.0.jar。

aopalliance-1.0.jar 是 AOP 联盟提供的规范包，可以通过地址
“ <http://mvnrepository.com/artifact/aopalliance/aopalliance/1.0> ”
下载。



(2) 创建切面类

由于该实例实现环绕通知，所以切面类需要实现 `org.aopalliance.intercept.MethodInterceptor` 接口。在 `src` 目录下，创建一个 `spring.proxyfactorybean` 包，并在该包中创建切面类 `MyAspect`。



(3) 配置切面并指定代理

切面类需要配置为Bean实例，Spring容器才能识别为切面对象。
在 `spring.proxyfactorybean` 包中，创建配置文件 `applicationContext.xml`，并在文件中配置切面和指定代理对象。



(4) 创建测试类

在spring.proxyfactorybean包中，创建测试类ProxyFactoryBeanTest，在主方法中使用Spring容器获取代理对象，并执行目标方法。



Spring AOP主要用于框架开发



感谢聆听

Thanks For Your Listening!