

# Spring

## Spring 简介

---

Spring 是一个开源框架，它由Rod Johnson创建。他是为了解决企业应用开发的复杂性而创建的。Spring 使用基本的JavaBean来完成以前只可能由EJB完成的事情。然而，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，任何Java应用都可以从Spring中受益。

Spring是一个轻量级的控制反转（IOC）和面向切面编程（AOP）的容器框架。

轻量：从大小与开销两方面而言Spring都是轻量的。完整的Spring框架可以在一个大小只有1MB多的JAR文件里发布。并且Spring所需的开销也是微不足道的。此外，Spring是非侵入式的：典型地：Spring应用中的对象不依赖于Spring的特定类。

控制反转：Spring通过一种控制反转（IOC）的技术促进了松耦合。当应用了IOC，一个对象依赖的其他对象会通过被动的方式传递进来（DI），而不是找个对象自己创建或查找依赖对象。你可以认为。IOC和JNDI相反——不是对象从容器中查找依赖，而是容器在对象初始化时不等对象请求就主动将依赖传递给它。

面向切面编程：Spring提供了面向切面编程（AOP）的丰富支持，允许通过分离应用的业务逻辑与系统级服务（例如：审计、事务管理）进行内聚性的开发。应用对象只实现它们应该做的——完成业务逻辑，仅此而已。它们并不负责其他的系统级关注点，例如日志、事务支持。

容器：Spring包含并管理应用对象的配置和生命周期，在这个意义上它是一种容器，你可以配置你的每个Bean如何被创建——基于一个可配置原型（prototype），你的Bean可以创建一个单独的实例或者每次需要时都生成一个实例——以及它们之间是如何关联的。然而，Spring不应该被混同于传统的重量级EJB容器，它们经常是庞大与笨重的，难以使用。

框架：Spring可以将简单的组件配置、组合成为复杂的应用。在Spring中应用对象被声明式地组合，典型地就在一个XML文件里，Spring也提供了很多基础的功能（事务管理、持久化框架集成等等），将应用逻辑的开发留给了你。

所有Spring的这些特征是你能够编写更干净、更可管理、并且更易测试的代码。它们也为Spring中的各种模块提供了基础支持。

**简单来说，Spring框架是一个分层的JAVA SE/EE full stack（一站式）轻量级的开源框架。一站式是指Spring在MVC分层的每一层都提供了解决方案（spring mvc ,spring data,spring标签）。**

**Spring版本：**

官网：[www.spring.io](http://www.spring.io)

- Spring 3.x
- Spring 4.x
- Spring 5.x

# 第一章 Spring Framework

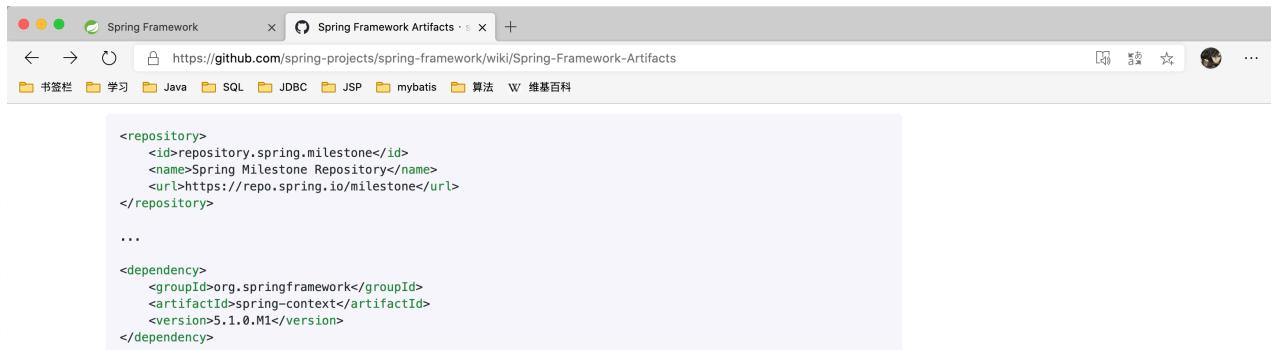
## 1.1 下载和解压

早期Spring作为开源框架，它的依赖和源码都在官网。Spring将源码托管到了GitHub上。

The screenshot shows the official Spring Framework website at <https://spring.io/projects/spring-framework>. The page features a navigation bar with links like 'Why Spring', 'Learn', 'Projects', 'Training', 'Support', and 'Community'. A sidebar on the left lists various Spring projects such as Spring Boot, Spring Data, Spring Cloud, etc. The main content area displays the 'Spring Framework' section, which includes an 'OVERVIEW' tab (highlighted with a red arrow labeled '1') and a 'LEARN' tab. It provides a brief introduction to the framework and its support policy.

This screenshot is identical to the one above, showing the Spring Framework homepage at <https://spring.io/projects/spring-framework>. The 'OVERVIEW' tab is highlighted with a red arrow labeled '1'.

The screenshot shows the GitHub repository for the Spring Framework at <https://github.com/spring-projects/spring-framework>. The repository's README.md file is displayed. A red oval highlights the 'Access to Binaries' section, which contains a link to the 'Spring Framework Artifacts wiki page'. A red arrow labeled '2' points to this section.



```
<repository>
    <id>repository.spring.milestone</id>
    <name>Spring Milestone Repository</name>
    <url>https://repo.spring.io/milestone</url>
</repository>

...
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.1.0.M1</version>
</dependency>
```

## Releases

You can also resolve GA versions of Spring Framework artifacts against <https://repo.spring.io/release>.

For more in-depth information about Spring repositories, see the [Spring Artifactory](#) page.

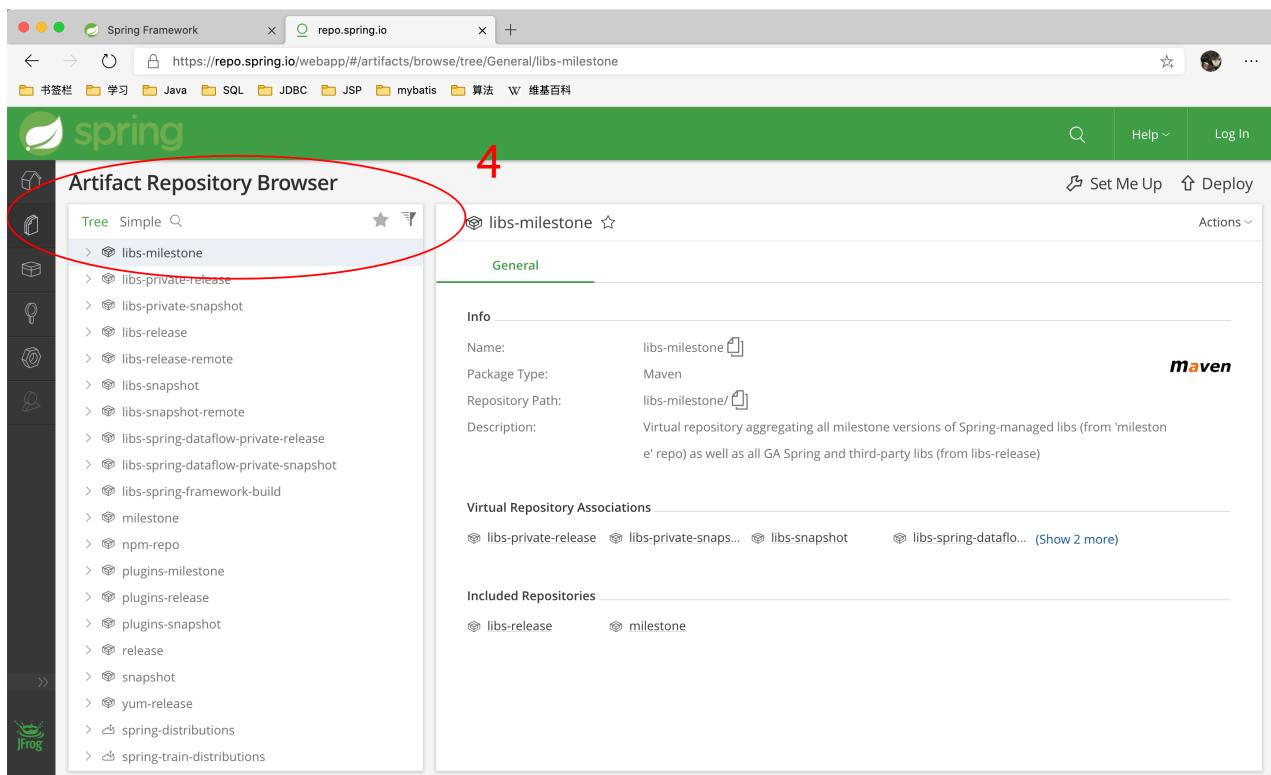
## Downloading a Distribution

3

If for whatever reason you are not using a build system with dependency management capabilities, you can download Spring Framework *distribution* zips from the Spring repository at <https://repo.spring.io/>. These distributions contain all source and binary jar files, as well as Javadoc and reference documentation, but *do not contain external dependencies*!

To create a distribution with all dependencies locally you can build from source, see [Build Zip with Dependencies](#) for details.

© 2020 GitHub, Inc. Terms Privacy Security Status Help Contact GitHub Pricing API Training Blog About



4

The screenshot shows the JFrog Artifactory interface. On the left, there's a sidebar with various icons and a tree view of repositories. The main area is titled "Artifact Repository Browser" and shows the "libs-milestone" repository under "General". The "Info" section contains the following details:

- Name: libs-milestone
- Package Type: Maven
- Repository Path: libs-milestone/
- Description: Virtual repository aggregating all milestone versions of Spring-managed libs (from 'milestone' repo) as well as all GA Spring and third-party libs (from libs-release)

Below the "Info" section, there are sections for "Virtual Repository Associations" and "Included Repositories".

Spring Framework repo.spring.io https://repo.spring.io/webapp/#/artifacts/browse/tree/General/

书签栏 学习 Java SQL JDBC JSP mybatis 算法 W 维基百科

## Artifact Repository Browser

Tree Simple  Actions ▾

libs-release-local

- .index
- com
- io
- org
  - apache/geode/geode-core
  - cloudfoundry
  - projectreactor
  - springframework
    - amqp
    - analytics/spring-analytics
    - android
    - batch
    - boot
    - build/aws-maven
    - cloud
    - credhub
    - data
    - hateoas/spring-hateoas
    - integration

### libs-release

General

**Info**

Name: libs-release

Package Type: Maven

Repository Path: libs-release/

Description: Virtual repository aggregating all GA versions of Spring-managed artifacts (from libs-release-local) as well as all third-party GA libs (from libs-release-remote and other repositories)

**Virtual Repository Associations**

libs-milestone libs-private-release libs-private-snapshots libs-snapshot (Show 4 more)

**Included Repositories**

release ext-release-local libs-release-remote

Spring Framework repo.spring.io https://repo.spring.io/webapp/#/artifacts/browse/tree/General/

书签栏 学习 Java SQL JDBC JSP mybatis 算法 W 维基百科

## Artifact Repository Browser

Tree Simple  Actions ▾

social

spring

- 3.2.0.RELEASE
- 3.2.1.RELEASE
- 3.2.10.RELEASE
- 3.2.11.RELEASE
- 3.2.12.RELEASE
- 3.2.13.RELEASE
- 3.2.14.RELEASE
- 3.2.15.RELEASE
- 3.2.16.RELEASE
- 3.2.17.RELEASE
- 3.2.18.RELEASE
- 3.2.2.RELEASE
- 3.2.3.RELEASE
- 3.2.4.RELEASE
- 3.2.5.RELEASE
- 3.2.6.RELEASE
- 3.2.7.RELEASE
- 3.2.8.RELEASE

### libs-release

General

**Info**

Name: libs-release

Package Type: Maven

Repository Path: libs-release/

Description: Virtual repository aggregating all GA versions of Spring-managed artifacts (from libs-release-local) as well as all third-party GA libs (from libs-release-remote and other repositories)

**Virtual Repository Associations**

libs-milestone libs-private-release libs-private-snapshots libs-snapshot (Show 4 more)

**Included Repositories**

release ext-release-local libs-release-remote

Spring Framework | repo.spring.io | Spring 官网 下载 SpringFramework | 新标签页

书签栏 学习 Java SQL JDBC JSP mybatis 算法 维基百科

## Artifact Repository Browser

**libs-release**

**General**

**Info**

Name: libs-release  
Package Type: Maven  
Repository Type: 完整版  
Description: 文档  
Virtual Repository Associations: libs-milestone, libs-private-release, libs-private-snapshots, libs-snapshot (Show 4 more)  
Included Repositories: release, ext-release-local, libs-release-remote

**maven pom文件**

**Virtual Repository Associations**

**Included Repositories**

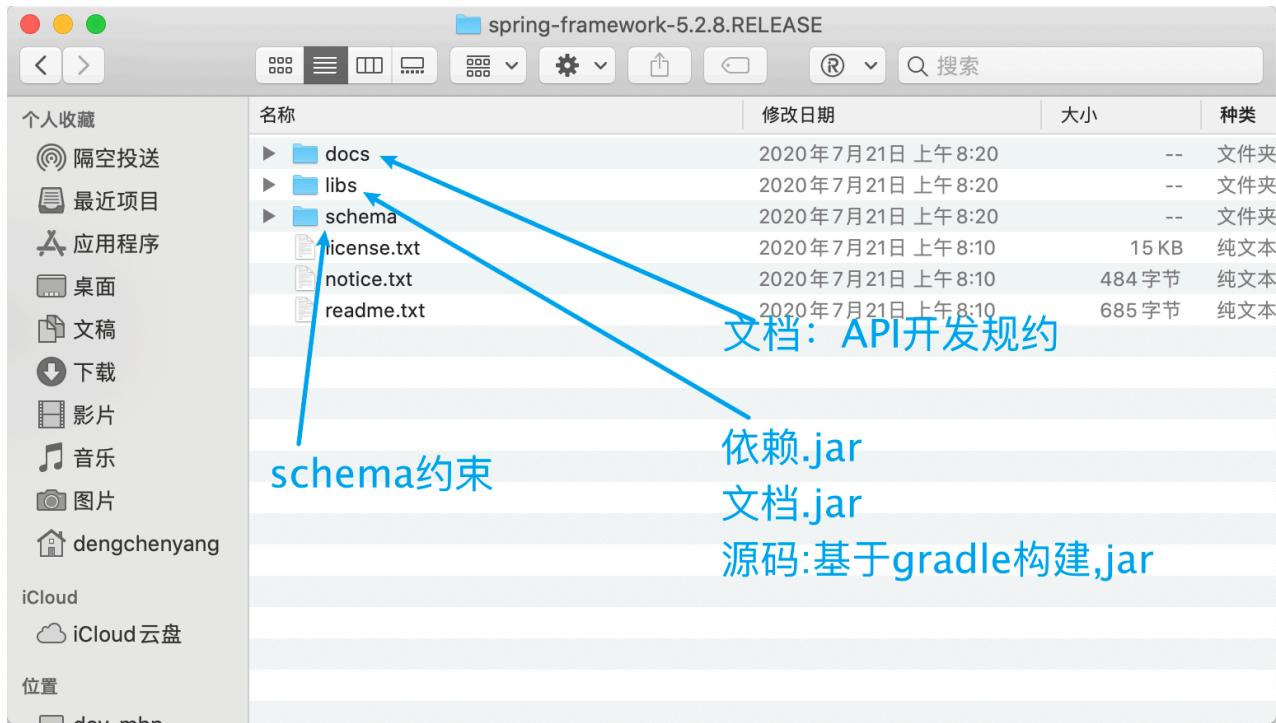
Spring Framework | repo.spring.io | Index of libs-release-local/org/springframework/spring/5.2.8.RELEASE/ | +

书签栏 学习 Java SQL JDBC JSP mybatis 算法 维基百科

## Index of libs-release-local/org/springframework/spring/5.2.8.RELEASE/

Name	Last modified	Size
<a href="#">spring-5.2.8.RELEASE-dist.zip</a>	21-Jul-2020 08:20	82.35 MB
<a href="#">spring-5.2.8.RELEASE-docs.zip</a>	21-Jul-2020 08:20	39.83 KB
<a href="#">spring-5.2.8.RELEASE-schema.zip</a>	21-Jul-2020 08:20	60.89 KB
<a href="#">spring-5.2.8.RELEASE.pom</a>	21-Jul-2020 08:20	1.45 KB

Artifactory Online Server at repo.spring.io Port 443



## 1.2 控制反转原理 (Inversion of Control)

控制反转 (Inversion of control) 是一种软件设计模式。那到底什么被反转了？获得依赖对象的过程被反转了。控制反转 (IoC) 把传统模式中需要自己通过 new 实例化构造函数，或者通过工厂模式实例化的任务交给容器。通俗的来理解，就是本来当需要某个类（构造函数）的某个方法时，自己需要主动实例化变为被动，不需要再考虑如何实例化其他依赖的类，只需要依赖注入 (Dependency Injection, DI), DI 是 IoC 的一种实现方式。所谓依赖注入就是由 IoC 容器在运行期间，动态地将某种依赖关系注入到对象之中。所以 IoC 和 DI 是从不同的角度的描述的同一件事情，就是通过引入 IoC 容器，利用依赖注入的方式，实现对象之间的解耦。

控制反转的底层使用工厂模式+配置文件+反射

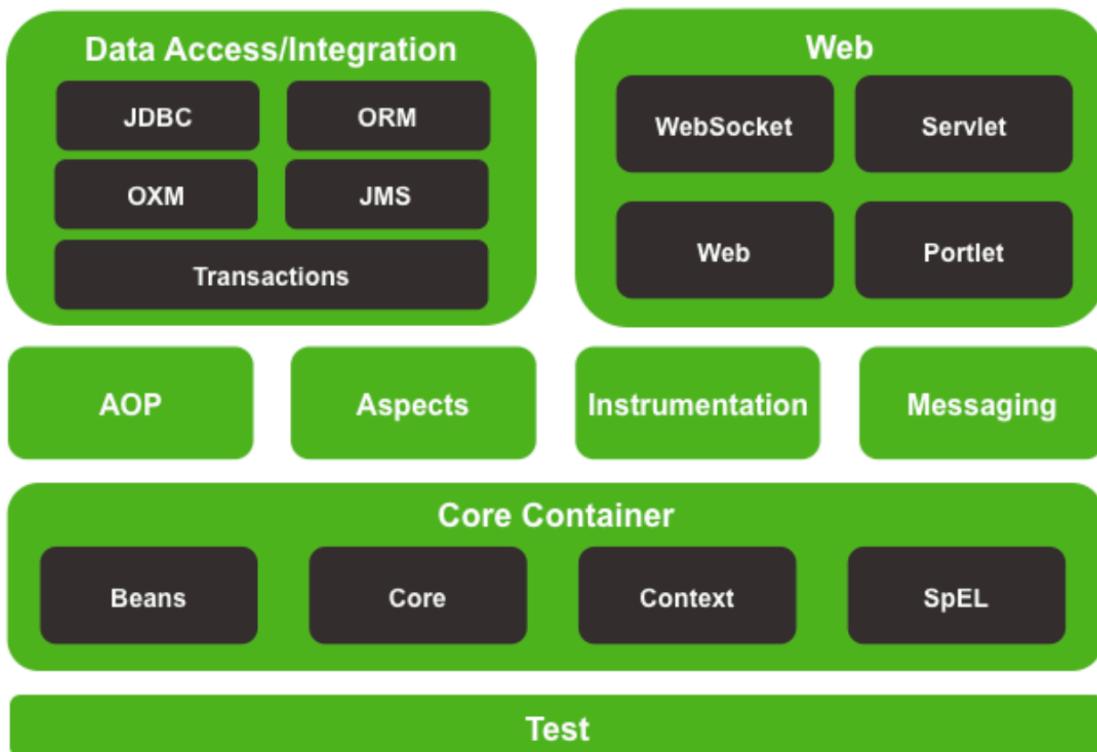
## 1.3 IOC入门程序

### 1.3.1 创建Java Project

### 1.3.2 加入Spring 核心依赖包



## Spring Framework Runtime



- spring-core.jar\*
  - spring-context.jar\*
  - spring-beans.jar\*
  - spring-expression.jar\*
  - spring-jcl.jar
  - log4j.jar
- 增加log4j.properties配置文件

### 1.3.3 创建spring 配置文件

applicationContext.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      xsi:schemaLocation="
5          http://www.springframework.org/schema/beans
6          http://www.springframework.org/schema/beans/spring-beans.xsd">
7
8      <!-- 用于配置交给spring管理的对象
9          id:唯一标识符, 使用schema唯一约束, 不允许出现特殊字符/
10         class: 用于配置全限定类名</pre>
```

```
10      name: 名称, 一般可以替代id, 理论上name是可以重复的, 实际开发过程中要么使用id, 要么使用
11      name
12          scope: 声明JavaBean的作用域:
13              singleten 单例 (默认Bean都是单例模式)
14              prototype 多例 (在和struts2整合时)
15              request 每次请求生成对象
16              session 每次会话生成对象
17              globalSession 只有在prolet项目中有效
18      -->
19      <bean id="userDao" class="com.sofwin.dao.impl.UserDaoImpl"></bean>
19  </beans>
```

### 1.3.4 创建需要控制反转的类

```
1  package com.sofwin.dao.impl;
2
3  import com.sofwin.dao.UserDao;
4
5  public class UserDaoImpl implements UserDao{
6
7      @Override
8      public void save() {
9          System.out.println("jdbc.....");
10     }
11
12 }
```

### 1.3.5 测试控制反转

```
1  package com.sofwin.test;
2
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  import com.sofwin.dao.UserDao;
7
8  public class Test1 {
9
10     public static void main(String[] args) {
11         //创建工厂对象
12         ApplicationContext context = new
13             ClassPathXmlApplicationContext("applicationContext.xml");
14
15         UserDao dao=(UserDao)context.getBean("userDao");
16         UserDao dao2=(UserDao)context.getBean("userDao");
17         dao.save();
18         System.out.println(dao==dao2);
19     }
20 }
```

ApplicationContext是接口

- ClassPathXmlApplicationContext() 创建工厂对象，加载的是类路径下的配置文件
- FileSystemXmlApplicationContext() 创建工厂对象，加载的是文件系统中的配置文件

### 1.3.6 配置schema 约束

略。。。

## 1.4 IOC详解（基于XML方式）

### 1.4.1 普通Bean

同入门程序。简单的Java类被称为普通bean。应该有默认构造方法

### 1.4.2 静态工厂 Bean

使用了工厂模式的Java类，并且获取对象的方法使用的是静态方法

使用步骤：

1. 创建静态工厂类

```

1 package com.sofwin.util;
2
3 import com.sofwin.dao.UserDao;
4 import com.sofwin.dao.impl.UserDaoImpl;
5
6 public class DaoFactory {
7
8     public static UserDao createUserDao() {
9         return new UserDaoImpl();
10    }
11 }
```

2. 在配置文件中配置

```

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   xsi:schemaLocation="
5       http://www.springframework.org/schema/beans
6       http://www.springframework.org/schema/beans/spring-beans.xsd">
7
8     <!-- factor-method: 用于指定当前静态工厂类中用于生成对象的静态方法的方法名--&gt;
9     &lt;bean id="userDao" class="com.sofwin.util.DaoFactory" factory-
method="createUserDao"&gt;&lt;/bean&gt;
9 &lt;/beans&gt;</pre>

```

## 使用场景：

在整合第三方框架时，有提供的静态工厂类，我们想要获取工厂类生成的对象时。

### 1.4.3 实例工厂 Bean

使用了工厂模式的Java类，并且获取对象的方法是普通方法。这种类型的Java类称之为实例工厂bean

#### 使用步骤：

##### 1. 创建实例工厂类

```
1 package com.sofwin.util;
2
3 import com.sofwin.dao.UserDao;
4 import com.sofwin.dao.impl.UserDaoImpl;
5
6 public class DaoFactory {
7
8     public UserDao createUserDao() {
9         return new UserDaoImpl();
10    }
11 }
```

##### 2. 在配置文件中进行配置

```
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   xsi:schemaLocation="
5       http://www.springframework.org/schema/beans
6       http://www.springframework.org/schema/beans/spring-beans.xsd">
7     <!--配置实例工厂bean  -->
8     <bean id="factory" class="com.sofwin.util.DaoFactory2"></bean>
9     <!-- 调用实例工厂的普通方法
10        factory-id: 用于指定工厂bean的id, 注意该工厂bean必须为实例工厂
11        factory-method: 用于配置实例工厂中生成对象的方法名
12        -->
13     <bean id="userDao" factory-bean="factory" factory-method="createUserDao">
14     </bean>
15 </beans>
```

## 1.4.4 后处理Bean

在自定义的bean的初始化方法执行前和执行后进行监控，进行扩展

使用步骤：

1. 定义类实现接口 `BeanPostProcessor`，并实现方法

```
postProcessBeforeInitialization(), postProcessAfterInitialization
```

```
1 package com.sofwin.util;
2
3 import org.springframework.beans.BeansException;
4 import org.springframework.beans.factory.config.BeanPostProcessor;
5
6 /**
7 * 自定义后处理bean
8 * @author dengchenyang
9 *
10 */
11 public class MyProcessor implements BeanPostProcessor {
12
13     @Override
14     public Object postProcessAfterInitialization(Object bean, String
15         beanName) throws BeansException {
16         System.out.println("当前初始化后的beanName" + beanName + ":" + bean);
17         return BeanPostProcessor.super.postProcessAfterInitialization(bean,
18             beanName);
19     }
20
21     @Override
22     public Object postProcessBeforeInitialization(Object bean, String
23         beanName) throws BeansException {
24         System.out.println("当前初始化前的beanName" + beanName + ":" + bean);
25         return BeanPostProcessor.super.postProcessBeforeInitialization(bean,
26             beanName);
27     }
28 }
```

2. 将自定义的后处理bean交给spring容器管理，只有交给spring容器管理之后，他才能监控到容器中配置bean的初始化方法

```
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   xsi:schemaLocation="
5       http://www.springframework.org/schema/beans
6       http://www.springframework.org/schema/beans/spring-beans.xsd">
7     <!--配置实例工厂bean -->
8     <bean id="factory" class="com.sofwin.util.DaoFactory"></bean>
9     <!-- 调用实例工厂的普通方法
```

```
9         factory-id: 用于指定工厂bean的id, 注意该工厂bean必须为实例工厂
10        factory-method: 用于配置实例工厂中生成对象的方法名
11        -->
12        <bean id="userDao" factory-bean="factory" factory-
13          method="createUserDao"></bean>
14        <bean class="com.sofwin.util.MyProcessor"></bean>
15      </beans>
```

**注意：**后处理bean由容器自动调用，所以在配置时不需要id属性

**应用场景：**容器中所有需要做功能扩展，或在初始化之前需要增强功能的bean，可以使用后处理bean

## 1.4.5 Bean的生命周期

### 1. 构造方法

构造方法最先被执行

### 2. 初始化方法

在实例创建后可以进行一些参数的初始化设置

```
1   __init__ //在Python中代表的就是初始化方法
```

### 3. 销毁方法

在容器销毁时触发，当容器关闭时容器就销毁了。在实例销毁后可以释放资源

```
1  package com.sofwin.dao.impl;
2
3  import com.sofwin.dao.UserDao;
4
5  public class UserDaoImpl implements UserDao{
6
7      @Override
8      public void save() {
9          System.out.println("jdbc.....");
10     }
11
12     public void init() {
13         System.out.println("init.....");
14     }
15
16     public void destroy() {
17         System.out.println("destory.....");
18     }
19
20 }
```

```

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      xsi:schemaLocation="
5          http://www.springframework.org/schema/beans
6          http://www.springframework.org/schema/beans/spring-beans.xsd">
7
8      <!--
9          init-method: 用于指定当前JavaBean的初始化方法的方法名
10         destroy-method: 用于指定当前JavaBean的销毁方法的方法名
11     -->
12     <bean id="userDao" class="com.sofwin.dao.impl.UserDaoImpl" init-method="init"
13         destroy-method="destroy"></bean>
14 </beans>
```

```

1 package com.sofwin.test;
2
3 import org.springframework.context.support.ClassPathXmlApplicationContext;
4
5 import com.sofwin.dao.UserDao;
6 /**
7 * IOC入门程序
8 * @author dengchenyang
9 *
10 */
11 public class Test3 {
12
13     private static ClassPathXmlApplicationContext context;
14
15     public static void main(String[] args) {
16         context = new ClassPathXmlApplicationContext("applicationContext.xml");
17
18         UserDao dao=(UserDao)context.getBean("userDao");
19         dao.save();
20         //容器销毁
21         context.close();
22     }
23 }
```

## 总结：

- 在spring中常用的两个工厂 `BeanFactory` 和 `ApplicationContext`，都是接口，并且 `ApplicationContext` 是 `BeanFactory` 的子接口，同时 `BeanFactory` 的实例已经过时，不推荐使用。
- 对于 `ApplicationContext`，单例的bean会在容器创建时全部实例化。而对于 `BeanFactory` 在容器创建时不会实例化bean，在调用`getBean()`时才会去实例化

## 1.5 IOC详解（基于注解方式）

### 1.5.1 分类

- 类级别的注解

```
@Component
```

组件，是类级别的注解，只能放在类的定义的上面。spring提供的IOC的基础注解

```
1 @Component("userDao")
2 //等价于
3 @Component(value="userDao")
```

可以替换掉XML文件中的 `<bean>` 标签，其中value值代替id或name属性，class属性是通过注解所在的类名获取到的

衍生注解：语义化的衍生注解，在spring MVC中使用

```
@Repository 用于定义dao层的bean
@Service 用于定义service层的bean
@Controller 用于定义控制层的注解
```

- 属性和方法级别的注解

```
@PostConstructor 在构造方法后执行，定义初始化方法 (init-method)
@PreDestroy 在容器销毁前执行，定义销毁方法 (destroy-method)
```

### 1.5.2 实例

注意：使用注解首先需要开启组件扫描，并增加AOP依赖

#### 1. 在XML文件中加入context约束，并开启组件扫描

```
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="
6     http://www.springframework.org/schema/beans
7     http://www.springframework.org/schema/beans/spring-beans.xsd
8     http://www.springframework.org/schema/context
9     http://www.springframework.org/schema/context/spring-context.xsd">
10
11    <!-- 开启组件扫描 base-package: 用于定义需要扫描的包 -->
12    <context:component-scan base-package="com.sofwin"></context:component-
13      scan>
14  </beans>
```

## 2. 在需要控制反转的类上增加注解

```
1 package com.sofwin.dao.impl;
2
3 import javax.annotation.PostConstruct;
4 import javax.annotation.PreDestroy;
5
6 import org.springframework.stereotype.Component;
7
8 import com.sofwin.dao.UserDao;
9
10 @Component("userDao")
11 public class UserDaoImpl implements UserDao{
12
13     @Override
14     public void save() {
15         System.out.println("jdbc.....");
16     }
17
18     @PostConstruct
19     public void init() {
20         System.out.println("init.....");
21     }
22
23     @PreDestroy
24     public void destroy() {
25         System.out.println("destory.....");
26     }
27
28 }
```

## 1.6 依赖注入（基于XML方式）

Dependency Injection简称DI。依赖注入会将所依赖的对象自动交给目标对象，而不是让对象自己去获取依赖

**依赖注入的前提：** 依赖注入必须使用IOC的环境，并且依赖的双方都必须交给spring容器来管理

### 1.6.1 简单类型的依赖注入

注入的属性是简单类型，包括：基本数据类型+包装数据类型

- setter方法注入

**在JavaBean中包含有公有的setter方法的属性，可以通过setter方法注入，当前类中必须有默认构造**

```
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="
```

```

6          http://www.springframework.org/schema/beans
7          http://www.springframework.org/schema/beans/spring-beans.xsd
8          http://www.springframework.org/schema/context
9          http://www.springframework.org/schema/context/spring-context.xsd">
10
11      <!-- property标签用于进行setter方法注入
12          name: 有公有setter方法的属性名
13          value: 简单类型的属性需要的值
14      -->
15      <bean id="user" class="com.sofwin.pojo.User" >
16          <property name="id" value="10"></property>
17          <property name="username" value="admin"></property>
18          <property name="age" value="10"></property>
19      </bean>
20  </beans>

```

- 构造方法注入

我们需要重载构造方法，并且使用 **<constructor-arg>** 进行配置，底层使用反射获取 Constructor对象，new Instance(...)

```

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="
6          http://www.springframework.org/schema/beans
7          http://www.springframework.org/schema/beans/spring-beans.xsd
8          http://www.springframework.org/schema/context
9          http://www.springframework.org/schema/context/spring-context.xsd">
10
11      <!-- constructor-arg标签用于指定user中构造方法中参数的值
12          index: 参数的索引值
13          value: 当前参数的实际值
14          type: 用于指定形参中的类型。一般用于有多个相同参数的构造时，通过定义参数的类
15          型来选择对应的构造方法
16          name: 形参名称
17      -->
18      <bean id="user" class="com.sofwin.pojo.User" >
19          <constructor-arg index="0" value="10"></constructor-arg>
20          <constructor-arg index="1" value="admin"></constructor-arg>
21          <constructor-arg index="2" value="100"></constructor-arg>
22      </bean>
23  </beans>

```

```

1  <bean id="user" class="com.sofwin.pojo.User" >
2      <constructor-arg name="id" value="10"></constructor-arg>
3      <constructor-arg name="username" value="admin"></constructor-arg>
4      <constructor-arg name="age" value="100"></constructor-arg>
5  </bean>

```

```
1      <bean id="user" class="com.sofwin.pojo.User" >
2          <constructor-arg name="id" value="10" type="java.lang.Integer">
3              </constructor-arg>
4          <constructor-arg name="username" value="admin"></constructor-arg>
5          <constructor-arg name="age" value="100"></constructor-arg>
6      </bean>
```

- P命名空间注入

需要配置P命名空间。为了简化setter方法注入的方式

```
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      xmlns:t="http://www.springframework.org/schema/p"
6      xsi:schemaLocation="
7          http://www.springframework.org/schema/beans
8          http://www.springframework.org/schema/beans/spring-beans.xsd
9          http://www.springframework.org/schema/context
10         http://www.springframework.org/schema/context/spring-context.xsd">
11
12     <!-- p命名空间只要求schema后必须是p, xmlns后可以是任意字符
13     在bean标签中使用: p:有setter方法的属性的属性名="简单属性对应的值"
14     -->
15     <bean id="user" class="com.sofwin.pojo.User" t:id="1" t:username="admin"
16           t:age="10"></bean>
17 </beans>
```

- SPEL表达式注入

在spring3.0 之后才引入了SPEL表达式注入， SpEL表达式注入可以结合setter方法注入和P命名空间注入使用

```
1 //语法
2 #{}
3 #{常量}
4 #{表达式}
5 #{Java方法和函数}
```

SpEL表达式注入在简单类型的注入使用比较少，用在引用类型的依赖注入较多（因为可以获取到对象属性）

```

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      xmlns:t="http://www.springframework.org/schema/p"
6      xsi:schemaLocation="
7          http://www.springframework.org/schema/beans
8          http://www.springframework.org/schema/beans/spring-beans.xsd
9          http://www.springframework.org/schema/context
10         http://www.springframework.org/schema/context/spring-context.xsd">
11     <bean id="user" class="com.sofwin.pojo.User" t:id="#${(java.lang.Math).random()*100}" t:username="#{'admin'}" t:age="#{20+1}"></bean>
12   </beans>

```

## 1.6.2 引用类型的依赖注入

引用类型的依赖注入，注入的属性是引用类型（地址）。被注入的引用对象也应该交给spring容器管理

- setter方法注入

在JavaBean中包含有公有的setter方法的属性，可以通过setter方法注入，当前类中必须有默认构造

```

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      xmlns:t="http://www.springframework.org/schema/p"
6      xsi:schemaLocation="
7          http://www.springframework.org/schema/beans
8          http://www.springframework.org/schema/beans/spring-beans.xsd
9          http://www.springframework.org/schema/context
10         http://www.springframework.org/schema/context/spring-context.xsd">
11     <!-- 把被注入的对象交给spring容器管理 -->
12     <bean id="role1" class="com.sofwin.pojo.Role" t:id="1" t:rolename="用户管理员"></bean>
13     <!-- ref: reference引用地址，spring容器中定义的bean的id -->
14     <bean id="user" class="com.sofwin.pojo.User">
15         <property name="id" value="1"></property>
16         <property name="username" value="admin"></property>
17         <property name="age" value="10"></property>
18         <property name="role" ref="role1"></property>
19     </bean>
20   </beans>

```

- 构造方法注入

```

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      xmlns:t="http://www.springframework.org/schema/p"
6      xsi:schemaLocation="
7          http://www.springframework.org/schema/beans
8          http://www.springframework.org/schema/beans/spring-beans.xsd
9          http://www.springframework.org/schema/context
10         http://www.springframework.org/schema/context/spring-context.xsd">
11
12     <!-- 把被注入的对象交给spring容器管理 -->
13     <bean id="role1" class="com.sofwin.pojo.Role" t:id="1" t:rolename="用户管
14     理员"></bean>
15
16     <!-- ref: reference引用地址, spring容器中定义的bean的id -->
17     <bean id="user" class="com.sofwin.pojo.User">
18         <constructor-arg index="0" value="1"></constructor-arg>
19         <constructor-arg index="1" value="admin"></constructor-arg>
20         <constructor-arg index="2" value="10"></constructor-arg>
21         <constructor-arg index="3" ref="role1"></constructor-arg>
22     </bean>
23 </beans>

```

- P命名空间注入

```

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      xmlns:t="http://www.springframework.org/schema/p"
6      xsi:schemaLocation="
7          http://www.springframework.org/schema/beans
8          http://www.springframework.org/schema/beans/spring-beans.xsd
9          http://www.springframework.org/schema/context
10         http://www.springframework.org/schema/context/spring-context.xsd">
11
12     <!-- 把被注入的对象交给spring容器管理 -->
13     <bean id="role1" class="com.sofwin.pojo.Role" t:id="1" t:rolename="用户管
14     理员"></bean>
15     <!-- p:role-ref="注入的bean的id"引用类型的属性赋值 -->
16     <bean id="user" class="com.sofwin.pojo.User" t:id="1" t:username="admin"
17     t:age="10" t:role-ref="role1"></bean>
18 </beans>

```

- SpEL表达式注入

```

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      xmlns:t="http://www.springframework.org/schema/p"
6      xsi:schemaLocation="
7          http://www.springframework.org/schema/beans
8          http://www.springframework.org/schema/beans/spring-beans.xsd

```

```
8         http://www.springframework.org/schema/context
9         http://www.springframework.org/schema/context/spring-context.xsd">
10
11     <!-- 把被注入的对象交给spring容器管理 -->
12     <bean id="role1" class="com.sofwin.pojo.Role" t:id="1" t:rolename="用户管
13 理员"></bean>
14     <!-- p:role-ref="注入的bean的id"引用类型的属性赋值 -->
15     <bean id="user" class="com.sofwin.pojo.User" t:id="1" t:username="admin"
16     t:age="#{role.id}" t:role-ref="role1"></bean>
17
18 </beans>
```

### 1.6.3 其他类型的依赖注入

- List

```
1      <!-- 把被注入的对象交给spring容器管理 -->
2      <bean id="role1" class="com.sofwin.pojo.Role" t:id="1" t:rolename="用户管
3          理员"></bean>
4      <!-- List, Set, 数组类型的属性注入, 如果注入的值只有一个, 可以直接使用value或ref注入
5          -->
6      <bean id="user" class="com.sofwin.pojo.User" t:id="1" t:username="admin"
7          t:age="10" t:role-ref="role1">
8          <property name="names">
9              <!-- 用于注入List类型的属性 -->
10             <list>
11                 <!-- 集合泛型为简单类型时使用value标签 type指定集合的泛型 标签中间放
12                 注入的值 -->
13                 <value type="int">1</value>
14                 <value type="int">2</value>
15                 <value type="int">3</value>
16             </list>
17         </property>
18         <property name="roles">
19             <list>
20                 <!-- 集合泛型为引用类型时使用ref标签 bean指定被注入的bean的id -->
21                 <ref bean="role1"/>
22                 <ref bean="role1"/>
23             </list>
24         </property>
25     </bean>
```

- Map

```
1 <!-- 略 -->
2         <property name="score">
3             <!-- 用于进行Map类型的依赖注入 -->
4             <map>
5                 <!-- entry标签用于设置map中的key和value
6                     key: 当map中的key为简单类型时
7                     key-ref: 当map中的key为引用类型时, 放配置的bean的id-->
```

```
8                     value: 当map中的value为简单类型时
9                     value-ref: 当map中的value为引用类型时, 放配置的bean的id
10                    -->
11                    <entry key="id" value="100"></entry>
12                    <entry key-ref="role1" value-ref="role1"></entry>
13                </map>
14            </property>
15        <!-- 略 -->
```

- Set

```
1    <!-- 略 -->
2    <property name="sets">
3        <!-- 对set类型的属性进行依赖注入 -->
4        <set>
5            <!-- 注入简单类型 -->
6            <value>1</value>
7            <!-- 注入引用类型 -->
8            <ref bean="role1" />
9        </set>
10   </property>
11   <!-- 略 -->
```

- Properties

```
1    <!-- 略 -->
2    <property name="pro">
3        <!-- 对properties类型的属性进行依赖注入 -->
4        <props>
5            <!-- properties中定义的key只能是简单类型 -->
6            <prop key="key1">value1</prop>
7        </props>
8    </property>
9    <!-- 略 -->
```

- 数组类型

```
1    <!-- 略 -->
2    <property name="usernames">
3        <!-- 对数组类型的属性进行依赖注入 -->
4        <array>
5            <!-- 当数组为简单类型时使用value标签 -->
6            <value>1</value>
7            <value>2</value>
8        </array>
9    </property>
10   <!-- 略 -->
```

```
1 <!-- 略 -->
2 <property name="usernames_>
3     <!-- 对数组类型的属性进行依赖注入 -->
4     <array>
5         <!-- 当数组为引用类型时使用ref标签 -->
6         <ref bean="role1"></ref>
7         <ref bean="role1"></ref>
8     </array>
9 </property>
10 <!-- 略 -->
```

## 总结：

依赖注入是指当获取某个bean时，它所依赖的bean会被注入到要获取的bean中。触发依赖注入有两种情况：

- 当用户第一次通过getBean()的方式向IOC容器索要Bean的实例时，容器检测（判断）是否和其他Bean有依赖关系，如果有触发依赖注入
- 当用户在bean中定义有 `lazy-init` 属性时，让容器在解析注册的bean时进行预实例化，触发依赖注入

## 1.7 依赖注入（基于注解方式）

### 注意：

- 需要开启扫描
- 在开发过程中如果需要使用注解方式的注入依赖，依赖的bean的IOC也必须使用注解

```
1 <context:component-scan base-package="com.sofwin"></context:component-scan>
```

举例：User 依赖Role，如果User中的Role类型的属性注入时使用的是注解，那么User的IOC必须使用注解方式，Role随意。

### 1.7.1 按名称注入

```
@Resource
```

按名称注入，行业内认为注入时尽量使用@Resource来进行，原因是该注解是JDK的注解，和spring容器解耦合。但在实际开发中，为了开发速度，大家选用的通常是@AutoWired

该注解属于类级别、属性级别、方法级别。注解可以写在需要注入的属性上，也可以写在该属性的set()方法上。

```
1 //按名称注入，name属性的值为spring容器中需要注入的bean的id
2 @Resource(name="role1")
```

```
@Autowired
```

```
@Qualifier(value="")
```

```
1 @Component("user")
```

```
2  public class User {  
3  
4      private Integer id;  
5      private String username;  
6      private Integer age;  
7      private Role role;  
8      private List<Integer> names;  
9      @Autowired  
10     @Qualifier(value="role1")  
11     private List<Role> roles;  
12     private Map<?, ?> score;  
13     private Set sets;  
14     private Properties pro;  
15     private String[] usernames;  
16  
17     //set()get()方法略  
18 }  
19
```

## 1.7.2 按类型注入

@Autowired

该注解属于属性和方法级别。按类型注入，要注意被注入的bean这种类型在容器中只能被定义一次。在实际开发中通常使用按类型注入。

## 1.7.3 简单属性的注入

@Value

## 1.8 Spring 整合JUnit

### 1.8.1 JUnit简介

JUnit是一个Java语言的单元测试框架。它由Kent Beck和Erich Gamma建立，逐渐成为源于Kent Beck的sUnit的xUnit家族中最为成功的一个。JUnit有它自己的JUnit扩展生态圈。多数Java的开发环境都已经集成了JUnit作为单元测试的工具。

JUnit是由 Erich Gamma 和 Kent Beck 编写的一个回归测试框架（regression testing framework）。Junit测试是程序员测试，即所谓白盒测试，因为程序员知道被测试的软件如何（How）完成功能和完成什么样（What）的功能。Junit是一套框架，继承TestCase类，就可以用Junit进行自动测试了

使用实例：

1. 添加JUnit依赖： `junit-4.13.jar` , `hamcrest-core-1.3.jar`
2. 编写单元测试

```
1  package com.sofwin.test;  
2  
3  import org.junit.After;
```

```
4 import org.junit.Assert;
5 import org.junit.Before;
6 import org.junit.Test;
7
8 import com.sofwin.service.UserService;
9
10 /**
11 * 测试-spring整合JUnit
12 * @author dengchenyang
13 *
14 */
15 public class Test1 {
16
17     /**
18      * 用于加载资源
19      */
20     @Before
21     public void init() {
22         System.out.println("init.....");
23     }
24
25     /**
26      * 释放资源
27      */
28     @After
29     public void after() {
30         System.out.println("after.....");
31     }
32
33     /**
34      * 无返回值无参数的方法
35      * 方法上增加@Test注解，该方法就可以使用JUnit去独立运行
36      */
37     @Test
38     public void test01() {
39         System.out.println("test01.....");
40         UserService service=new UserService();
41         int id=service.getId();
42         Assert.assertEquals(10, id);
43     }
44 }
```

## 1.8.2 Spring整合JUnit

同样spring对JUnit单元测试也提供了支持。spring-test.jar就是对JUnit的支持。

使用实例：

1. 导入依赖： `junit-4.13.jar` , `hamcrest-core-1.3.jar` , `spring-test.jar`
2. 编写单元测试

```
1 package com.sofwin.test;
```

```

2
3 import org.junit.Assert;
4 import org.junit.Test;
5 import org.junit.runner.RunWith;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.test.context.ContextConfiguration;
8 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
9
10 import com.sofwin.service.UserService;
11
12 /**
13 * 测试-spring整合JUnit
14 * @author dengchenyang
15 * 构建包含有spring环境的单元测试，环境中包含容器（配置文件）
16 * @ContextConfiguration用于指定spring环境需要的配置文件
17 */
18 @RunWith(SpringJUnit4ClassRunner.class)
19 @ContextConfiguration(locations = "classpath:applicationContext.xml")
20 public class Test2 {
21
22     /**
23     * 有了spring环境，说明工厂对象也存在
24     * 可以直接在当前类中去使用IOC、DI、AOP等技术
25     */
26     @Autowired
27     private UserService service;
28
29     @Test
30     public void test01() {
31         int id=service.getId();
32         Assert.assertEquals(10, id);
33     }
34 }
```

## 1.9 AOP 面向切面编程

AOP 是Aspect Oriented Programming的缩写，意为面向切面编程 或面向方面编程。通过预编译方式和运行期间动态代理实现程序功能的统一维护的一种技术。AOP是OOP的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数编程的一种衍生泛型。利用AOP可以对业务逻辑的各个部分之间耦合度降低，提高程序的可重用性，提高了开发的效率，提高了程序的可扩展性。

**总结：**

AOP是通过创建目标对象的代理对象，在代理对象调用方法时，进行功能增强，从而实现了功能的扩展。

JDK的动态代理方式：只能代理生成实现过接口的实例。

针对JDK动态代理的缺陷：spring提供了2种代理方式：

1. 基于JDK的动态代理：当目标对象实现过接口时，可以使用JDK的动态代理来生成代理对象从而实

现功能扩展

2. 基于字节码的增强：当目标类没有实现过接口或实现过接口时，都可以使用基于cglib的字节码增强从而生成代理对象（通过给目标对象创建一个子类对象，从而生成代理对象）

## 19.1 AOP 常用概念

目标类 (target)

需要进行功能扩展的类

连接点 (joinPoint)

目标类中的所有方法都称为连接点

切入点 (pointCut)

连接点中需要增强的方法

增强/通知 (advice)

需要进行的功能扩展

织入 (weaving)

将增强应用到切入点的过程

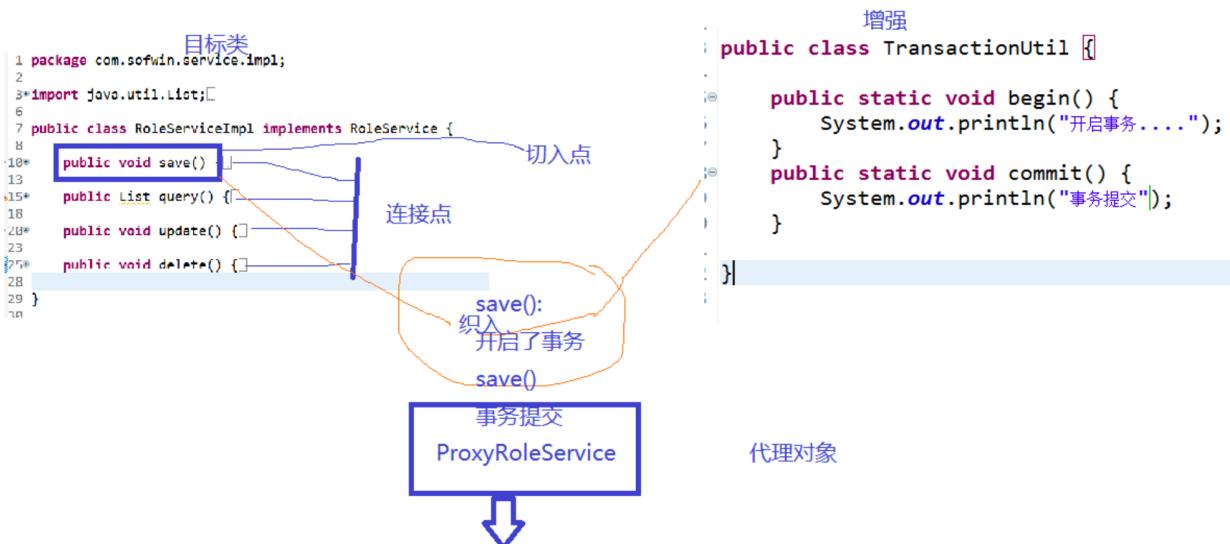
代理对象 (proxy)

织入完成后生成的代理对象

切面 (aspect)

多个切入点组成切面

## RoleService中save方法增加开启事务和事务提交



## 1.9.2 AOP 底层原理

基于JDK动态代理

### 1. 确定目标类 UserServiceImpl 实现了接口 UserService

```
1 package com.sofwin.service.impl;
2
3 import java.util.List;
4
5 import com.sofwin.service.UserService;
6
7 public class UserServiceImpl implements UserService{
8
9     @Override
10    public void save() {
11        System.out.println("save.....");
12    }
13
14    @Override
15    public List query() {
16        return null;
17    }
18
19    @Override
20    public void update() {
21        System.out.println("update.....");
22    }
23
24    @Override
25    public void delete() {
26        System.out.println("delete.....");
27    }
28}
```

```
29 }
```

2. 确定连接点 (4个)
3. 确定切入点 (save()方法)
4. 编写增强类 (TransactionUtil)

```
1 package com.sofwin.util  
2  
3 public class TransactionUtil(){  
4  
5     public static void begin(){  
6         System.out.println("开启事务。 . . . . ");  
7     }  
8  
9     public static void commit(){  
10        System.out.println("事务提交。 . . . . ");  
11    }  
12}
```

5. 创建一个工厂类，在方法中生成目标对象的代理对象，并且在代理对象中去调用增强类中的方法

```
1 package com.sofwin.util;  
2  
3 import java.lang.reflect.InvocationHandler;  
4 import java.lang.reflect.Method;  
5 import java.lang.reflect.Proxy;  
6  
7 import com.sofwin.service.UserService;  
8 import com.sofwin.service.impl.UserServiceImpl;  
9  
10 public class ProxyBeanFactory {  
11  
12     /**  
13      * 基于JDK的动态代理方式获取目标类的代理对象，从而实现目标类的增强  
14      * @return  
15      */  
16     public static UserService getUserServiceProxy() {  
17         //创建目标对象  
18         UserService service = new UserServiceImpl();  
19         UserService proxy =  
20             (UserService)Proxy.newProxyInstance(service.getClass().getClassLoader(),  
21             service.getClass().getInterfaces(), new InvocationHandler() {  
22                 @Override  
23                 public Object invoke(Object proxy, Method method, Object[] args)  
24                     throws Throwable {  
25                     //确定切入点  
26                     String name=method.getName();  
27                     if("save".equals(name)) {  
28                         TransactionUtil.begin();  
29                     }  
30                 }  
31             });  
32         return proxy;  
33     }  
34 }
```

```

28         //执行目标方法并获取返回值
29         Object obj=method.invoke(service, args);
30         if("save".equals(name)) {
31             TransactionUtil.commit();
32         }
33         return obj;
34     }
35 );
36
37     return proxy;
38 }
39
40 }

```

6. 测试：比较创建 `UserServiceImpl` 对象，调用`save()`方法；通过工厂对象，调用静态方法获取 `UserServiceImpl` 对象，再去调用`save()`方法的执行结果

```

1 /**
2  * 测试基于JDK动态代理方式的AOP
3 */
4 @Test
5 public void testSave() {
6     UserService userServiceProxy =
7     ProxyBeanFactory.getUserServiceProxy();
8     userServiceProxy.save();
9 }

```

注意：此时通过工厂对象生成的UserService对象已经变为\$Proxy类型，不再是UserServiceImpl。

### 基于cglib字节码增强

cglib作为字节码增强的框架，很流行，早期的Hibernate使用了cglib。一般使用cglib我们需要下载cglib的依赖，但是spring将cglib的依赖直接打包在了spring依赖中。

1. 确定目标类 `RoleService`
2. 确定连接点（4个）
3. 确定切入点（`save()`方法）
4. 确定增强类（`TransactionUtil`）
5. 在工厂类中创建静态方法，去创建代理对象，进行织入

```

1 package com.sofwin.util;
2
3 import java.lang.reflect.InvocationHandler;
4 import java.lang.reflect.Method;
5 import java.lang.reflect.Proxy;
6
7 import org.springframework.cglib.proxy.Enhancer;
8 import org.springframework.cglib.proxy.MethodInterceptor;
9 import org.springframework.cglib.proxy.MethodProxy;
10

```

```
11 import com.sofwin.service.UserService;
12 import com.sofwin.service.impl.RoleServiceImpl;
13 import com.sofwin.service.impl.UserServiceImpl;
14
15 public class ProxyBeanFactory {
16
17     /**
18      * 基于cglib字节码增强的方式获取代理对象，，从而实现目标类的增强
19      * @return
20      */
21     public static RoleServiceImpl getRoleServiceImplProxy() {
22         //创建目标对象
23         RoleServiceImpl service=new RoleServiceImpl();
24         //创建cglib核心对象
25         Enhancer enhancer = new Enhancer();
26         //设置父类
27         enhancer.setSuperclass(service.getClass());
28         //设置目标方法的回调
29         enhancer.setCallback(new MethodInterceptor() {
30
31             /**
32              * 拦截目标方法
33              * 目标对象中的所有方法都会进入到该方法中
34              */
35             @Override
36             public Object intercept(Object proxy, Method method, Object[]
37             args, MethodProxy methodProxy) throws Throwable {
38                 //确定切入点
39                 String name=method.getName();
40                 if("save".equals(name)) {
41                     TransactionUtil.begin();
42                 }
43                 //执行目标方法并获取返回值
44                 Object invoke = method.invoke(service, args);
45                 if("save".equals(name)) {
46                     TransactionUtil.commit();
47                 }
48                 return invoke;
49             });
50             //在确定了父类和回调方法后，生成了目标对象的代理对象
51             RoleServiceImpl proxy = (RoleServiceImpl)enhancer.create();
52             return proxy;
53     }
54 }
```

## 6. 测试：通过cglib的方式AOP是否成功

```

1  /**
2   * 测试基于cglib字节码增强方式AOP
3   */
4  @Test
5  public void testSave2() {
6      RoleServiceImpl roleServiceImplProxy =
7          ProxyBeanFactory.getRoleServiceImplProxy();
8      roleServiceImplProxy.save();
}

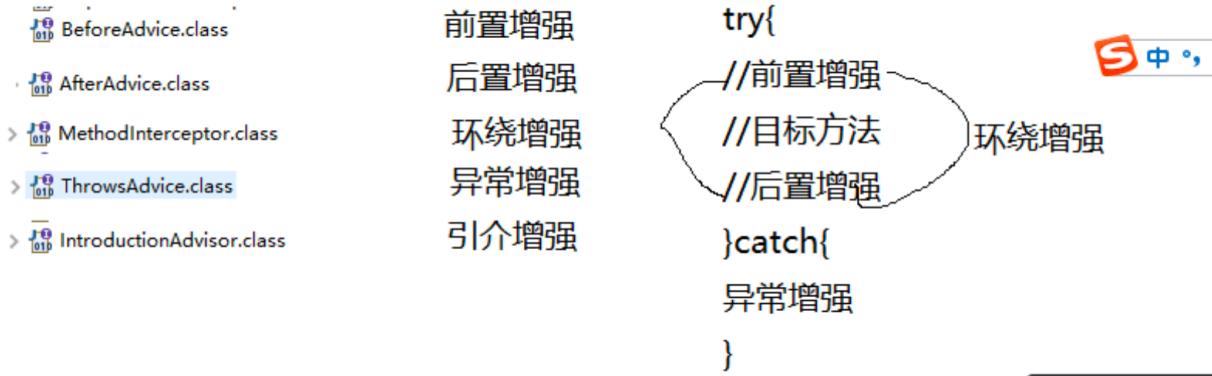
```

### 1.9.3 半自动AOP

AOP联盟制定了一套用于规范AOP实现的底层API，通过这些统一的底层API，可以使得各个AOP实现及工具产品之间实现相互移植。这些API主要以接口的形式提供，是AOP编程思想所要解决的横切面交叉关注点问题各部件的最高抽象。spring的AOP框架中也直接以这些API为基础构建

#### AOP联盟规定的5种增强类型

- 前置增强：在目标方法执行前对功能进行增强
- 后置增强：在目标方法执行后对功能进行扩展
- 环绕增强：在目标方法执行前后对功能进行扩展
- 异常增强：当抛出异常时实现的功能扩展
- 引介增强（了解）：给目标对象增加属性或方法，破坏了面向对象编程的思想



#### 搭建半自动AOP环境

spring的4个核心依赖：

- spring-core.jar
- spring-beans.jar
- spring-context.jar
- spring-expression.jar

其他依赖：

- spring-jcl.jar
- spring-test.jar

- junit-4.13.jar
- spring-aop.jar

半自动AOP依赖：

- aopaliance.jar： AOP联盟所提出的AOP的所有规范（接口）

实例：

### 1. 创建增强类，实现接口 `MethodInterceptor`

```

1 package com.sofwin.util;
2
3 import org.aopaliance.intercept.MethodInterceptor;
4 import org.aopaliance.intercept.MethodInvocation;
5
6 /**
7 * 测试半自动AOP的实现
8 * 创建增强类
9 * 环绕增强
10 * @author dengchenyang
11 *
12 */
13 public class MyAdvise implements MethodInterceptor{
14
15     /**
16      * 目标对象的方法回调
17      */
18     @Override
19     public Object invoke(MethodInvocation methodInvocation) throws Throwable
20     {
21         System.out.println("开启事务。。。。。。");
22         //手动调用目标方法
23         Object obj = methodInvocation.proceed();
24         System.out.println("提交事务。。。。。。");
25         return obj;
26     }
27 }
28

```

### 2. 配置代理对象的工厂类，由spring提供

```

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       xmlns:t="http://www.springframework.org/schema/p"
6       xmlns:aop="http://www.springframework.org/schema/aop"
7       xsi:schemaLocation="
8           http://www.springframework.org/schema/beans
9           http://www.springframework.org/schema/beans/spring-beans.xsd

```

```

9         http://www.springframework.org/schema/context
10        http://www.springframework.org/schema/context/spring-context.xsd
11        http://www.springframework.org/schema/aop
12        http://www.springframework.org/schema/aop/spring-aop.xsd">
13
14        <!-- 将目标类交给spring容器管理 -->
15        <bean id="userService" class="com.sofwin.service.impl.UserServiceImpl">
16        </bean>
17
18        <!-- 将增强类交给spring容器管理 -->
19        <bean id="myAdvise" class="com.sofwin.util.MyAdvise"></bean>
20
21        <!-- 配置spring提供的生成代理对象的工厂类 -->
22        <bean id="userServiceProxy"
23            class="org.springframework.aop.framework.ProxyFactoryBean">
24
25            <!-- 注入目标对象 -->
26            <property name="target" ref="userService"></property>
27
28            <!-- 注入的是Class[], Class类型, 直接用value="全限定类名" -->
29            <property name="interfaces" value="com.sofwin.service.UserService">
30
31        </property>
32
33            <!-- 注入增强对象 -->
34            <property name="interceptorNames" value="myAdvise"></property>
35
36            <!-- boolean:true, false 让spring强制使用cglib springAOP规则:
37                如果目标类实现了接口, 默认使用jdk动态代理来实现AOP
38                如果目标类没有实现任何接口, 默认使用cglib的字节码增强来实现AOP
39                如果设置optimize=true, 强制使用cglib
40
41            -->
42            <property name="optimize" value="false"></property>
43
44        </bean>
45    </beans>

```

### 3. 测试：

```

1 package com.sofwin.test;
2
3 /**
4 * AOP 测试类
5 * @author dengchenyang
6 *
7 */
8 @RunWith(SpringJUnit4ClassRunner.class)
9 @ContextConfiguration(locations = {"classpath:applicationContext.xml"})
10 public class Test1 {
11
12     @Resource(name="userServiceProxy")
13     private UserService service;
14
15     /**
16      * 测试半自动AOP
17      */
18     @Test
19     public void testSave3() {
20         service.save();
21     }

```

```
22  
23 }
```

**总结：** 使用半自动AOP不需要我们手动去创建工厂对象，直接使用spring提供的 **ProxyFactoryBean** 即可

**存在的问题：**

1. AOP的增强存在强耦合关系， 和aopaliance
2. 每个代理对象的创建都需要去配置工厂，并且工厂的配置比较繁琐

#### 1.9.4 传统AOP

传统AOP的配置方式在半自动AOP配置的基础上进行了简化，去掉了工厂对象的配置，提供了 **<aop:>** 标签，使用简单的标签就可以去配置织入和生成代理对象的过程。

1. 搭建环境
  - spring-core.jar
  - spring-beans.jar
  - spring-context.jar
  - spring-expression.jar
  - spring-jcl.jar
  - spring-test.jar
  - spring-aop.jar
  - aopaliance.jar
  - aspectjweaver.jar
2. 创建目标类（UserServiceImpl）
3. 增强需要使用环绕增强（MyAdvise）
4. 配置织入的过程

```
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   xmlns:t="http://www.springframework.org/schema/p"  
6   xmlns:aop="http://www.springframework.org/schema/aop"  
7   xsi:schemaLocation="  
8       http://www.springframework.org/schema/beans  
     http://www.springframework.org/schema/beans/spring-beans.xsd  
9       http://www.springframework.org/schema/context  
     http://www.springframework.org/schema/context/spring-context.xsd  
10      http://www.springframework.org/schema/aop  
     http://www.springframework.org/schema/aop/spring-aop.xsd">  
11  
12      <!-- 将目标类交给spring容器管理 -->  
13      <bean id="userService" class="com.sofwin.service.impl.UserServiceImpl">  
14      </bean>  
15      <!-- 将增强类交给spring容器管理 -->  
16      <bean id="myAdvise" class="com.sofwin.util.MyAdvise"></bean>
```

```
17      <aop:config proxy-target-class="false">
18          <aop:advisor advice-ref="myAdvise" pointcut="execution(*
com.sofwin.service.impl.UserServiceImpl.save())" />
19      </aop:config>
20  </beans>
```

## 1.9.5 切入点表达式

切入点表达式的作用是在整个项目中，通过特定的规则来寻找切入点。切入点表达式是基于 `execution()` 函数的。

括号中的表达式：

```
1  返回值 包名.类名.方法名(参数类型) throws
```

- 返回值不可以省略：int,String,Integer,User,\*代表任意类型返回值
- 包名不能省略，可以使用通配符\*
- 类名不能省略，可以使用通配符\*代表任意类
- 方法名不能省略，可以使用通配符\*代表任意方法
- 方法中的参数类型：不能省略()代表无参的方法、(int),(int,String),(..)代表任意个任意类型的参数
- throws可以省略
- 可以在切入点表达式中使用&&和||对多个表达式进行与或运算

```
1  execution(* com.sofwin.*.impl.*.saveUser(..))||execution(*
com.sofwin.*.impl.*.updateUser(..)) //切入点为com.sofwin包下的所有子包中的impl子包下
的saveUser方法和updateUser方法，参数不限定
```

常见错误：

- 没有使用 `execution()`
- 丢失空格，表达式不正确

## 1.9.6 基于Aspect J的AOP

Aspect J是一个AOP框架，它扩展了Java语言。AspectJ定义了AOP语法，他有一个专门的编译器用来生成Java字节码编码规范的class文件。

- AspectJ是一个Java的AOP框架
- Spring2.0开始才引入了基于AspectJ的AOP
- `@Aspect`是AspectJ的新注解，基于JDK5的注解

基于AspectJ的AOP优点：

1. 增强不再与任何类有强耦合关系（不再实现接口，可以是普通类）
2. spring提供了基于AspectJ的配置方式
3. 在企业开发中自定义的增强一般都使用基于AspectJ的配置方式

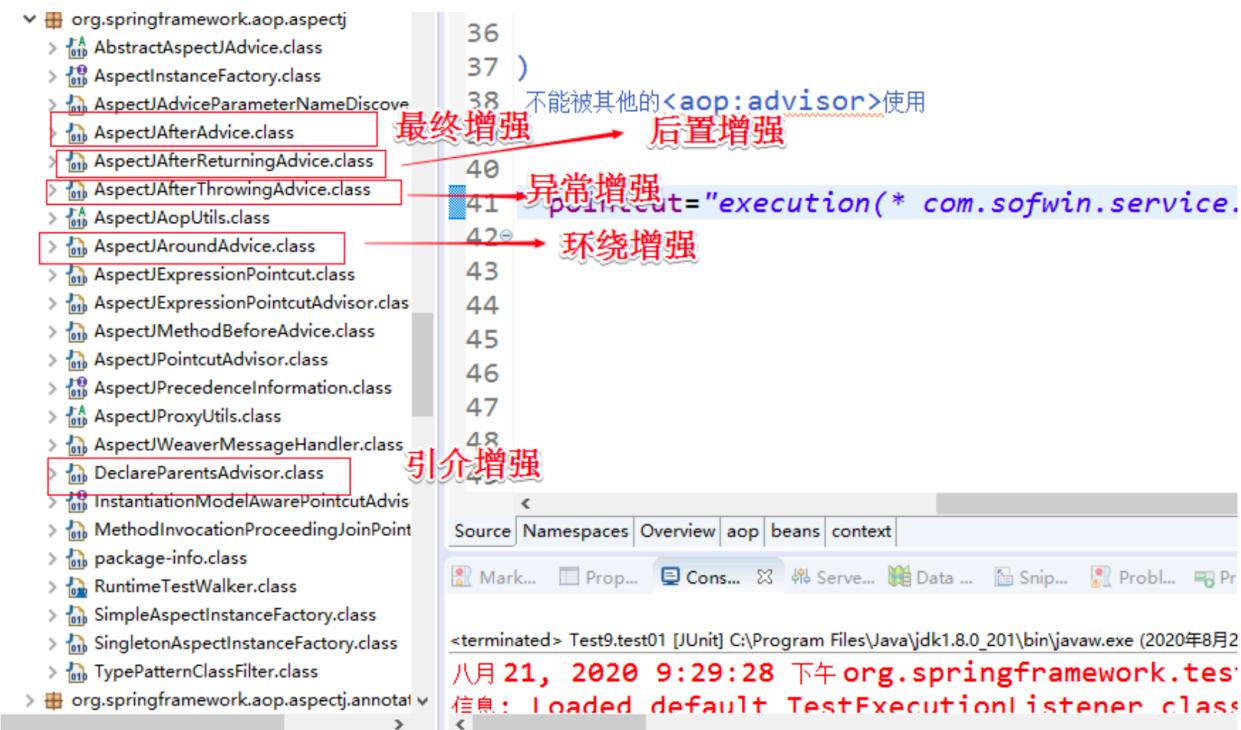
## 环境搭建

在传统AOP的环境基础上增加：spring-aspects.jar

### 基于AspectJ的增强类型

- 前置增强(beforeAdvisor)
- 后置增强(afterReturnning)
- 环绕增强(around)
- 最终增强(after)
- 异常增强(throws)
- 引介增强(declareParentsAdvisor)

```
1 try{
2     前置增强(准备资源)
3     目标方法
4     后置增强(功能扩展)
5 }catch(){
6     异常增强(异常处理)
7 }finally{
8     最终增强(释放资源)
9 }
```



### 基于XML的AspectJ

1. 确定目标类 (UserServiceImpl)
2. 确定连接点
3. 确定切入点 (save()方法)

#### 4. 编写增强类（普通的Java类）

```
1 package com.sofwin.util;
2
3 import org.aspectj.lang.JoinPoint;
4 import org.aspectj.lang.ProceedingJoinPoint;
5
6 /**
7 * 增强类
8 * @author dengchenyang
9 *
10 */
11 public class MyAspectJ {
12
13     /**
14      * 前置增强
15      * @param join
16      */
17     public void before(JoinPoint join) {
18         String name = join.getSignature().getName();
19         int modifiers = join.getSignature().getModifiers();
20         System.out.println("before....."+name+", "+modifiers);
21     }
22
23     /**
24      * 后置增强
25      * @param join: 获取目标方法的相关信息
26      * @param obj: 获取目标方法的返回值
27      */
28     public void afterReturnning(JoinPoint join, Object obj) {
29         System.out.println("after....."+obj);
30     }
31
32     /**
33      * 环绕增强
34      * 必须手动调用目标方法
35      * @param point: 用于手动调用目标方法
36      * @return : 返回值为Object类型
37      * @throws Throwable
38      */
39     public Object around(ProceedingJoinPoint point) throws Throwable {
40         //目标方法执行前
41         System.out.println("目标方法执行前1");
42         System.out.println("目标方法执行前2");
43         //手动调用目标方法
44         Object obj = point.proceed();
45         //目标方法执行后
46         System.out.println("目标方法执行后");
47         if(obj instanceof Integer) {
48             return Integer.parseInt(obj.toString());
49         }
50         if(obj instanceof String) {
```

```

51             return "username"+obj;
52         }
53     return null;
54 }
55
56 /**
57 * 异常增强
58 * @param ex
59 */
60 public void throwing(Throwable ex) {
61     if(ex.getMessage().contains("by zero")) {
62         System.out.println("除数不能为零！");
63     }
64 }
65
66 /**
67 * 最终增强
68 */
69 public void after() {
70     System.out.println("after.....");
71 }
72 }

```

## 5. 在spring配置文件中，织入生成代理对象

```

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      xmlns:t="http://www.springframework.org/schema/p"
6      xmlns:aop="http://www.springframework.org/schema/aop"
7      xsi:schemaLocation="
8          http://www.springframework.org/schema/beans
9          http://www.springframework.org/schema/beans/spring-beans.xsd
10         http://www.springframework.org/schema/context
11         http://www.springframework.org/schema/context/spring-context.xsd
12         http://www.springframework.org/schema/aop
13         http://www.springframework.org/schema/aop/spring-aop.xsd">
14
15     <!-- 将目标类交给spring容器管理 -->
16     <bean id="userService" class="com.sofwin.service.impl.UserServiceImpl">
17         </bean>
18         <!-- 将增强类交给spring容器管理 -->
19         <bean id="myAspectJ" class="com.sofwin.util.MyAspectJ"></bean>
20         <!-- 基于AspectJ方式的AOP织入过程依然需要使用<aop:config>标签 -->
21         <aop:config proxy-target-class="false">
22             <!-- 用于指定增强类 -->
23             <aop:aspect id="myAspectJ" ref="myAspectJ">
24                 <!--配置前置增强method:指定增强类中的方法的方法名 pointcut:指定前置增强的
25                 切入点 pointcut-ref:指定前置增强的切入点，用全局的切入点 arg-names:用于定义前置增强中
26                 获取到的目标方法的相关信息，值:前置增强的方法中的参数的参数名称，并且该参数只能是
27                 JoinPoint类型的参数-->

```

```

21             <aop:before method="before" pointcut="execution(*
22 com.sofwin.service.impl.*.save(..))" arg-names="join"/>
23             <aop:pointcut expression="execution(*
24 com.sofwin.service.impl.*.get*(..))" id="p1"/>
25         <!--用于配置后置增强 method:增强类中的后置增强的方法名 pointcut
26 pointcut-ref:使用全局的pointcut returnning:用于接受目标方法的返回值值:后置增强的方法
27 中的参数的参数名称(Object类型的参数)
28             <aop:after-returning method="afterReturnning" pointcut-
29 ref="p1" returning="obj"/>
30             -->
31             <!--用于配置环绕增强
32             <aop:around method="around" pointcut-ref="p1"/>
33             -->
34             <!--配置异常增强 throwing:用于配置获取目标方法中的异常信息值:异常增强的方
35 法中的参数(Throwable类型参数的参数名称)-->
36             <aop:after-throwing method="throwing" pointcut-ref="p1"
37 throwing="ex"/>
38             <aop:after method="after" pointcut-ref="p1"/>
39         </aop:aspect>
40     </aop:config>
41 </beans>

```

## 6. 测试

### 基于注解的AspectJ

1. 开启组件扫描
2. 打开AspectJ代理驱动

```

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      xmlns:t="http://www.springframework.org/schema/p"
6      xmlns:aop="http://www.springframework.org/schema/aop"
7      xsi:schemaLocation="
8          http://www.springframework.org/schema/beans
9          http://www.springframework.org/schema/beans/spring-beans.xsd
10         http://www.springframework.org/schema/context
11         http://www.springframework.org/schema/context/spring-context.xsd
12         http://www.springframework.org/schema/aop
13         http://www.springframework.org/schema/aop/spring-aop.xsd">
14
15     <!-- 1.打开组件扫描 -->
16     <context:component-scan
17         base-package="com.sofwin"></context:component-scan>
18     <!-- 将目标类UserService交给spring管理 -->
19     <bean id="roleService" class="com.sofwin.service.impl.RoleServiceImpl">
20     </bean>
21     <!-- 打开aspectj的代理驱动 proxy-target-class:是否强制使用cglib -->
22 
```

```
18     <aop:aspectj-autoproxy proxy-target-class="true"></aop:aspectj-
19     autoproxy>
20 </beans>
```

### 3. 编写增强，在增强中使用AspectJ的注解

```
1 package com.sofwin.util;
2
3 import org.aspectj.lang.JoinPoint;
4 import org.aspectj.lang.ProceedingJoinPoint;
5 import org.aspectj.lang.annotation.After;
6 import org.aspectj.lang.annotation.AfterReturning;
7 import org.aspectj.lang.annotation.AfterThrowing;
8 import org.aspectj.lang.annotation.Around;
9 import org.aspectj.lang.annotation.Aspect;
10 import org.aspectj.lang.annotation.Before;
11 import org.aspectj.lang.annotation.Pointcut;
12 import org.springframework.stereotype.Component;
13
14 /**
15  * 佛祖保佑 * 包名:com.sofwin.util autor:andyLiu *增强类
16  */
17 @Component
18 //用于声明该类是一个基于aspectj的增强类 //<aop:aspect>
19 @Aspect
20 public class MyAspectJ {
21     // 前置增强
22     // value="切入点表达式|嵌入点名称"
23     // @Before(value = "execution(* com.sofwin.service.impl.*.*(..))")
24     public void before(JoinPoint join) {
25         String methodName = join.getSignature().getName();
26         int modifiers = join.getSignature().getModifiers();
27         System.out.println("before....." + methodName + "," + modifiers);
28     }
29
30     /**
31      * JoinPoint:获取目标方法的相关信息 * Object:用于获取目标方法的返回值的
32      */
33     // @AfterReturning(value = "execution(* com.sofwin.service.impl.*.*(..))", returning = "obj")
34     public void afterReturnning(JoinPoint join, Object obj) {
35         System.out.println("afterReturnning....." + obj);
36     }
37
38     /**
39      * 环绕增强的方法:返回值为Object类型,
40      * 必须:手动去调用目标方法
41      * ProceedingJoinPoint:用于手动调用目标方法 * @return
42      * @throws Throwable
43      */
44     // @Around(value="execution(* com.sofwin.service.impl.*.*(..))")
```

```

45  public Object around(ProceedingJoinPoint point) throws Throwable { //目标方法
46      执行前执行
47      System.out.println("目标方法前1"); System.out.println("目标方法前2"); //手动掉用
48      目标方法, 为目标方法的返回值
49      Object obj = point.proceed();
50      //目标方法后执行 System.out.println("目标方法后"); if(obj instanceof Integer) {
51      return Integer.parseInt(obj.toString())+100; }if(obj instanceof String)
52      {
53          return "userName:" + obj;
54      }
55
56      /** 异常增强(对异常进行加工处理)
57      */
58      @AfterThrowing(value = "aa()", throwing = "ex")
59      public void throwing(Throwable ex) {
60          if(ex.getMessage().contains("by zero")) { System.out.println("除数不能为0");
61      }
62      }
63
64      @After(value="aa()")
65      public void after() {
66          System.out.println("after.....");
67
68      //用于定义切入点的
69      //将注解写在一个无返回值, 无参数的方法中
70      //在原来使用切入点的位置直接使用方法名就可以代表切入点
71      @Pointcut("execution(* com.sofwin.service.impl.*.*(..))")
72      public void aa() {
73      }
74  }

```

#### 4. 测试

**总结:** 基于注解的AspectJ 的AOP

- @Aspect: 用于声明增强
- @Before: 前置增强
- @AfterReturn: 后置增强
- @Around: 环绕增强
- @After: 最终增强
- @AfterThrowing: 异常增强
- @Pointcut: 定义切入点

## 1.10 Spring JDBC

- 配置数据库连接池
- 读取配置文件
- 查询结果的封装

搭建spring JDBC的环境：

- spring-beans.jar
- spring-context.jar
- spring-expression.jar
- spring-core.jar
- spring-jcl.jar
- **spring-aop.jar**: 会使用jdbc.properties配置文件
- mysql-connector-java.jar
- druid.jar
- spring-test.jar
- **spring-tx.jar**
- **spring-jdbc.jar**
- junit.jar
- hamcrest.jar

### 1.10.1 JdbcTemplate

```
1 package com.sofwin.test;
2
3 import org.junit.Test;
4 import org.springframework.jdbc.core.JdbcTemplate;
5
6 import com.alibaba.druid.pool.DruidDataSource;
7
8 /**
9  * 测试Spring JDBC
10 * @author dengchenyang
11 *
12 */
13 public class Test1 {
14
15     @Test
16     public void testSpringJdbc() {
17         //创建模板对象
18         JdbcTemplate template = new JdbcTemplate();
19         //创建数据源
20         DruidDataSource druidDataSource = new DruidDataSource();
21         druidDataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
22         druidDataSource.setUrl("jdbc:mysql://localhost/java1908z");
23         druidDataSource.setUsername("root");
24         druidDataSource.setPassword("12345678");
25         //为模板对象指定数据源
26         template.setDataSource(druidDataSource);
27         //测试spring JDBC中的方法
```

```
28         Integer count=template.queryForObject("select count(*) from s_user",
29             Integer.class);
30         System.out.println(count);
31     }
```

## 总结：

1. 对象的创建耦合度较高，可以将对象交给spring管理
2. DataSource也应该交给spring管理，并且DataSource和Template之间有依赖注入DI
3. DataSource中的数据库连接信息有需要依赖注入DI

将Template交给spring管理

```
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      xmlns:t="http://www.springframework.org/schema/p"
6      xmlns:aop="http://www.springframework.org/schema/aop"
7      xsi:schemaLocation="
8          http://www.springframework.org/schema/beans
9          http://www.springframework.org/schema/beans/spring-beans.xsd
10         http://www.springframework.org/schema/context
11         http://www.springframework.org/schema/context/spring-context.xsd
12         http://www.springframework.org/schema/aop
13         http://www.springframework.org/schema/aop/spring-aop.xsd">
14
15     <!-- 开启组件扫描 -->
16     <context:component-scan base-package="com.sofwin"></context:component-scan>
17
18     <!-- 创建 JDBCTemplate -->
19     <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
20         <constructor-arg index="0" ref="dataSource"></constructor-arg>
21     </bean>
22
23     <!-- 创建数据源 -->
24     <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" >
25         <property name="driverClassName" value="com.mysql.cj.jdbc.Driver">
26             </property>
27             <property name="url" value="jdbc:mysql://localhost/java1908z"></property>
28             <property name="username" value="root"></property>
29             <property name="password" value="12345678"></property>
30         </bean>
31
32     </beans>
```

```
1  package com.sofwin.test;
2
3  import java.util.List;
4  import java.util.Map;
```

```
5
6 import org.junit.Test;
7 import org.junit.runner.RunWith;
8 import org.springframework.beans.factory.annotation.Autowired;
9 import org.springframework.test.context.ContextConfiguration;
10 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
11
12 import com.sofwin.pojo.Suser;
13 import com.sofwin.service.impl.UserServiceImpl;
14
15 @RunWith(SpringJUnit4ClassRunner.class)
16 @ContextConfiguration("classpath:applicationContext.xml")
17 public class Test2 {
18
19     @Autowired
20     private UserServiceImpl service;
21
22     /**
23      * 测试C
24      */
25     @Test
26     public void test01() {
27         Suser user=new Suser();
28         user.setUsername("test111");
29         user.setPwd("test111");
30         user.setSex("男");
31         service.save(user);
32     }
33
34     /**
35      * 测试R
36      */
37     @Test
38     public void test02() {
39         Suser user=new Suser();
40         user.setUsername("test111");
41         List<Map<String, Object>> result = service.query(user);
42         for (Map<String, Object> map : result) {
43             System.out.println(map);
44         }
45     }
46
47     /**
48      * 测试U
49      */
50     @Test
51     public void test03() {
52         Suser user=new Suser();
53         user.setId(116);
54         user.setUsername("test111");
55         user.setPwd("test111");
56         user.setSex("女");
```

```

57         service.updateById(user);
58     }
59
60     /**
61      * 测试D
62      */
63     @Test
64     public void test04() {
65         service.deleteById(117);
66     }
67 }
```

方法名	描述
<code>execute(String sql)</code>	执行DDL语句，一般执行insert和delete语句
<code>int update(String)</code>	执行update语句，int代表影响行数
<code>int update(String sql, Object... obj)</code>	执行update语句（带占位符），可变参数为动态SQL的赋值
<code>T queryForObject(String sql, Class type)</code>	执行返回结果为一行一列的查询语句，参数1为静态SQL,参数2为返回值的类对象
<code>List&lt;Map&lt;String, Object&gt;&gt; queryForList(String sql)</code>	将查询结果封装为泛型为map的集合，map中的key为查询的columnLabel，Value是该列对应的值
<code>List&lt;Map&lt;String, Object&gt;&gt; queryForList&lt;String&gt;(String sql, Object..obj)</code>	同上，SQL语句为动态SQL,参数为占位符对应的值
<code>List&lt;T&gt; queryForList&lt;String&gt;(String sql, Class type)</code>	执行查询结果为多行一列的查询。参数1为静态SQL，参数2为集合中泛型的类对象
<code>T queryForObject(String sql, RowMapper&lt;T&gt;)</code>	参数2为定义的结果封装
<code>List&lt;T&gt; query(String sql, RowMapper&lt;T&gt;)</code>	参数1：静态SQL,参数2为定义结果封装
<code>List&lt;T&gt; query(String sql, Object[] args, RowMapper&lt;T&gt;)</code>	参数1：动态SQL，参数2：动态SQL中需要的值；参数3:定义结果封装
<code>List&lt;T&gt; query(String sql, RowMapper&lt;T&gt; mapper, Object...args)</code>	参数1：动态SQL;参数2：定义结果封装；参数3：动态SQL中需要的值

## 总结：

存在问题：所有的DAO都需要显式地去注入JdbcTemplate，并且在配置文件中需要将JdbcTemplate交给spring管理同时依赖注入DataSource

## 1.10.2 JdbcDaoSupport

这是spring提供的spring JDBC的支持。`JdbcDaoSupport` 是spring提供的使用spring JDBC的一个父类，在这个父类中需要设置一个`DataSource`，子类可以通过`getJdbcTemplate()` 来获取`JdbcTemplate`。（说明 使用了`JdbcDaoSupport`的父类不需要关注`jdbcTemplate`的生成过程，不需要在配置文件中去配置`jdbcTemplate`）

注意：因为父类需要注入一个`DataSource`，并且`DataSource`存在于spring的源码中，所以`DataSource`的注入不能使用注解的方式进行依赖注入，所以`JdbcDaoSupport`的所有子类，都不允许使用注解方式控制反转

```
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      xmlns:t="http://www.springframework.org/schema/p"
6      xmlns:aop="http://www.springframework.org/schema/aop"
7      xsi:schemaLocation="
8          http://www.springframework.org/schema/beans
9          http://www.springframework.org/schema/beans/spring-beans.xsd
10         http://www.springframework.org/schema/context
11         http://www.springframework.org/schema/context/spring-context.xsd
12         http://www.springframework.org/schema/aop
13         http://www.springframework.org/schema/aop/spring-aop.xsd">
14
15
16     <!-- 创建数据源 -->
17     <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" >
18         <property name="driverClassName" value="com.mysql.cj.jdbc.Driver">
19         <property name="url" value="jdbc:mysql://localhost/java1908z"></property>
20         <property name="username" value="root"></property>
21         <property name="password" value="12345678"></property>
22     </bean>
23
24 </beans>
```

```
1  public class UserDaoImpl extends JdbcDaoSupport implements UserDao {
2      @Override
3      public User selectById(Integer id) {
4          String sql = "select * from s_user where id=?";
5          User user = super.getJdbcTemplate().queryForObject(sql, new
6          RowMapper<User>() {
7              @Override
8              public User mapRow(ResultSet rs, int rowNum) throws SQLException {
9                  User user = new User();
10                 user.setId(rs.getInt("id"));
11                 user.setUserName(rs.getString("username"));
```

```

11             user.setPwd(rs.getString("pwd"));
12             user.setRoleId(rs.getInt("roleId"));
13         return user;
14     }
15 }, id);
16 return user;
17 }
18 }
```

## 1.11 Spring 事务管理

需求：转账业务

```

1 create table b_aaccount(
2     id int not null primary key auto_increment,
3     userid varchar(20),-- 用户卡号
4     account int-- 操作金额
5 );
6 insert into b_account(userid,account) values('001',100),('001',-50);-- 001用户存入
100, 取出50
```

```

1 package com.sofwin.service.impl;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Service;
5 import com.sofwin.dao.UserDao;
6 import com.sofwin.pojo.User;
7 import com.sofwin.service.UserService;
8 // 如果注解不增加id的话, spring默认会将类名首字母小写作为bean 的id
9 @Service
10 public class UserServiceImpl implements UserService {
11     @Autowired
12     private UserDao dao ;
13     @Override
14     public User queryUserById(Integer id) {
15         return dao.selectById(id);
16     }
17     @Override
18     public void transfer(String fromUserId, String toUserId, Integer account) {
19         // 转出的人插入 -account
20         dao.saveAccount(fromUserId, account*-1);
21         // 发送短信
22         int i=1/0;
23         // 转入的人出入 account
24         dao.saveAccount(toUserId, account);
25     }
26 }
```

```
1 package com.sofwin.test;
```

```

2 import org.junit.Test;
3 import org.junit.runner.RunWith;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.test.context.ContextConfiguration;
6 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
7 import com.sofwin.dao.UserDao;
8 import com.sofwin.service.UserService;
9 // 构建测试的容器
10 @RunWith(SpringJUnit4ClassRunner.class)
11 @ContextConfiguration("classpath:applicationContext.xml")
12 public class Test4 {
13     @Autowired
14     private UserDao dao;
15     @Autowired
16     private UserService service;
17     @Test
18     public void test01() {
19         // 001给002转50
20         // 001余额为50 002余额为50
21         service.transfer("001", "002", 50);
22     }

```

```

mysql> select sum(account),use
+-----+-----+
| sum(account) | userid |
+-----+-----+
|      50      |    001   |
|      50      |    002   |
+-----+-----+

```

sum(account)	userid
0	001
50	002

100

50

**总结：**如果没有事务001的账户不可能插入到数据库中，插入到数据库中了说明这时候有事务，那为什么数据一致性出现错误呢？原因就是每个save()方法都有一个自动提交的事务。

### 1.11.1 事务回顾

**事务的定义：** 事务是一组逻辑操作，这组操作中的所有语句要么全部执行成功，要么全部执行失败。

**事务的特性（ACID）：**

- 原子性（Atomicity）：事务是一个原子操作，由一系列动作组成。事务的原子性确保动作要么全部完成，要么完全不起作用。
- 一致性（Consistency）：一旦事务完成（不管成功还是失败），系统必须确保它所建模的业务处于一致的状态，而不会是部分完成部分失败。在现实中的数据不应该被破坏。
- 隔离性（Isolation）：可能有许多事务会同时处理相同的数据，因此每个事务都应该与其他事务隔离开来，防止数据损坏。
- 持久性（Durability）：一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响，这样

就能从任何系统崩溃中恢复过来。通常情况下，事务的结果被写到持久化存储器中。

### 事务的并发读问题：

- 脏读：有两个事务T1,T2， T1读取了T2更新但没有被提交的数据。如果T2回滚后， T1读取的内容是无效的。
- 不可重复读：有两个事务T1,T2， T1读取一个字段，然后T2更新该字段。之后T1再去读取同一字段， T1事务两次读取同一字段读到的数据是不一样的。
- 幻读：有两个事务T1和T2， T1从一个表中读取了一个字段，然后T2在该表中插入了一些数据。之后T1再次读取同一个表，就会出现不同行数。

### 数据库的四种隔离级别：

隔离级别	描述
READ UNCOMMITTED (读未提交数据)	允许事务读取其他事务未提交的数据，可能出现脏读、不可重复读、幻读
READ COMMITTED (读已提交数据)	允许事务读取其他事务已经提交的数据，解决了脏读问题但是可能出现不可重复读、幻读的问题
REPEATABLE READ (可重复读)	确保一个事务可以多次从一个字段中读到相同的值，在事务执行期间内禁止其他事务对该字段的更新。解决了脏读和不可重复读的问题，但是可能出现幻读的问题
SERIALIZABLE (串行化)	确保一个事务可以从一个表中读取相同的行。在事务执行期间禁止其他事务对该表进行插入、更新、删除操作。解决了所有事务并发时读的问题，但是性能十分低下。

- MySQL支持4种隔离级别， 默认使用的是REPEATABLE READ（可重复读）
- Oracle支持2种隔离级别， READ COMMITTED ,SERIALIZABLE， 默认使用READ COMMITTED
- Sql Sever支持4种隔离级别， 默认使用READ COMMITTED（读已提交）

### 事务的传播行为，spring提供了7种 Propagation：

传播规则回答了这样一个问题：一个新的事务应该被启动还是被挂起，或者是一个方法是否应该在事务性上下文中运行。

- 保证所有的操作在同一个事务中（3种）
  - PROPAGATION\_REQUIRED（默认）  
Spring默认的传播机制，能满足绝大部分业务需求，如果外层有事务，则当前事务加入到外层事务，一块提交，一块回滚。如果外层没有事务，新建一个事务执行
  - PROPAGATION\_SUPPORT  
如果外层有事务，则加入外层事务，如果外层没有事务，则直接使用非事务方式执行。完全依赖外层的事务
  - PROPAGATION\_MANDATORY

与 NEVER 相反，如果外层没有事务，则抛出异常

- 保证所有的操作不再同一个事务中（3种）

- PROPAGATION\_REQUIRED

该事务传播机制是每次都会新开启一个事务，同时把外层事务挂起，当当前事务执行完毕，恢复上层事务的执行。如果外层没有事务，执行当前新开启的事务即可

- PROPAGATION\_NOT\_SUPPORTED

该传播机制不支持事务，如果外层存在事务则挂起，执行完当前代码，则恢复外层事务，无论是否异常都不会回滚当前的代码

- PROPAGATION\_NEVER

该传播机制不支持外层事务，即如果外层有事务就抛出异常

- 嵌套事务（1种）

- PROPAGATION\_NESTED

该传播机制的特点是可以保存状态保存点，当前事务回滚到某一个点，从而避免所有的嵌套事务都回滚，即各自回滚各自的，如果子事务没有把异常吃掉，基本还是会引发全部回滚的。

## 1.11.2 Spring 提供的事务管理

Spring 提供了一个平台事务管理器 `PlatformTransactionManager`（接口）用于管理事务，它有两个实现：

- `DataSourceTransactionManager`：用于管理 JDBC 的事务（Spring JDBC、MyBatis）
- `HibernateTransactionManager`：用于管理 Hibernate 的事务

`TransactionDefinition`（事务的定义）+ `TransactionStatus`（事务的状态）：

定义了事务的相关属性包含（事务的隔离级别，事务的传播行为，事务是否只读，事务的名称等）；

定义了事务的相关状态，事务是否开始，事务是否完成，是否回滚等

```
/* IS ONLY AVAILABLE IF SUPPORTED BY THE UNDERLYING TRANSACTION MANAGER.
 * @author Juergen Hoeller
 * @since 27.03.2003
 * @see #setRollbackOnly()
 * @see PlatformTransactionManager#getTransaction
 * @see org.springframework.transaction.support.TransactionCallback#doInTransaction
 * @see org.springframework.transaction.interceptor.TransactionInterceptor#currentTransaction
 */
public interface TransactionDefinition extends TransactionExecution, SavepointManager, Flushable {
    /**
     * Return whether this transaction internally carries a savepoint,
     * that is, has been created as nested transaction based on a savepoint.
     * This method is mainly here for diagnostic purposes, alongside
     * {@link #isNewTransaction()}. For programmatic handling of custom
     * savepoints, use the operations provided by {@link SavepointManager}.
     * @see #isNewTransaction()
     * @see #createSavepoint()
     * @see #rollbackToSavepoint(Object)
     * @see #releaseSavepoint(Object)
     */
    boolean hasSavepoint();
    /**
     * Flush the underlying session to the datastore, if applicable:
     * for example, all affected Hibernate/JPA sessions.
     * <p>This is effectively just a hint and may be a no-op if the underlying
     * transaction manager does not have a flush concept. A flush signal may
     * get applied to the primary resource or to transaction synchronizations,
     * depending on the underlying resource.
     */
    void flush();
}
```

总结：

平台事务管理器创建事务，将事务的属性保存在TransactionDefinition中，事务会被不断调用，在调用的过程中事务的状态会发生改变(比如创建保存点，或者提交事务)，将这些改变记录到 TransactionStatus 中。

### 1.11.3 编程性事务（了解）

可以参考JdbcTemplate，在Spring中提供了一个TransactionTemplate(事务管理模板)，我们可以将事务管理模板注入到需要进行事务管理的Java类中，execute()中保证了事务的一致性。

#### 1. 将事务管理模板交给spring容器管理

```
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      xmlns:t="http://www.springframework.org/schema/p"
6      xmlns:aop="http://www.springframework.org/schema/aop"
7      xsi:schemaLocation="
8          http://www.springframework.org/schema/beans
9          http://www.springframework.org/schema/beans/spring-beans.xsd
10         http://www.springframework.org/schema/context
11         http://www.springframework.org/schema/context/spring-context.xsd
12         http://www.springframework.org/schema/aop
13         http://www.springframework.org/schema/aop/spring-aop.xsd">
14
15
16     <!-- 开启组件扫描 -->
17     <context:component-scan
18         base-package="com.sofwin"></context:component-scan>
19
20
21     <!-- 创建 JDBCTemplate -->
22     <bean id="jdbcTemplate"
23         class="org.springframework.jdbc.core.JdbcTemplate">
24         <constructor-arg index="0" ref="dataSource"></constructor-arg>
25     </bean>
26
27     <!-- 创建数据源 -->
28     <bean id="dataSource"
29         class="com.alibaba.druid.pool.DruidDataSource">
30         <property name="driverClassName"
31             value="com.mysql.cj.jdbc.Driver"></property>
32         <property name="url" value="jdbc:mysql://localhost/java1908z">
33             </property>
34             <property name="username" value="root"></property>
35             <property name="password" value="12345678"></property>
36         </bean>
37
38
39     <!-- 定义平台事务管理器 在平台事务管理器中需要注入数据源 -->
40     <bean id="transactionManager"
41
42         class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
43             <property name="dataSource" ref="dataSource"></property>
```

```

37      </bean>
38      <!-- 创建事务管理模板 事务管理模板需要注入数据源 数据源需要通过
39      transactionManager:PlatformTransatcionManager平台事务管理器 -->
40      <bean
41          class="org.springframework.transaction.support.TransactionTemplate">
42          <property name="transactionManager" ref="transactionManager">
43      </property>
44      </bean>

```

2. 将事务管理模板注入到service中去

3. 在需要事务管理的方法中去调用execute()方法，保证了在execute()方法中的所有操作使用了相同的事务

```

// 如果注解不增加id的话，spring默认会将类名首字母小写作为bean 的id
@Service
public class UserServiceImpl implements UserService {
    // 注入事务管理模板
    @Autowired
    private TransactionTemplate template;

    @Autowired
    private UserDao dao ;
    @Override
    public User queryUserById(Integer id) {
        return dao.selectById(id);
    }
    @Override
    public void transfer(String fromUserId, String toUserId, Integer account) {

        template.execute(new TransactionCallback() {
            // 在该方法中运行的所有代码保证在同一个事务中
            @Override
            public Object doInTransaction(TransactionStatus status) {
                // 转出的人插入 -account
                dao.saveAccount(fromUserId, account*-1);
                // 数据库中插入 001 -50
                // 发送短信
                int i=1/0;// 代码抛异常，事务回滚，将001 -50 这条记录回滚掉
                // 转入的人出入 account
                dao.saveAccount(toUserId, account);
                return null;
            }
        });
    }

    public void test01() {
        dao.saveUser(null);
    }
}

```

## 1.11.4 工厂事务管理（了解）

Spring提供了一个TransactionProxyFactoryBean，代理工厂bean(生成有事务的代理对象，原来userService中transfer方法没有事务，通过AOP注入事务增强，生成代理对象)，通过代理工厂bean的方式对指定的方法进行事务管理。使用AOP的思想对切入点进行事务的增强

对比编程性事务,不需要破坏原有的业务逻辑

总结： 工厂事务管理我们需要将每个需要进行事务管理的service全部配置生成代理对象。

## 1.11.5 声明式事务（掌握）

Spring内置了事务管理的AOP,我们只需要指定切入点，指定增强内容，Spring就可以通过声明式事务对指定切入点的方法进行指定内容的增强，不需要手动去配置代理对象的生成了。

### 基于XML方式的声明式事务

1. 事务的AOP需要使用 `<tx>` 标签，需要导入tx约束
2. 配置平台事务管理器
3. 配置 `<tx:advice>` 事务增强（传播行为、隔离级别）
4. 配置AOP将增强应用到切入点上

```
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      xmlns:t="http://www.springframework.org/schema/p"
6      xmlns:aop="http://www.springframework.org/schema/aop"
7      xmlns:tx="http://www.springframework.org/schema/tx"
8      xsi:schemaLocation="
9          http://www.springframework.org/schema/beans
10         http://www.springframework.org/schema/beans/spring-beans.xsd
10         http://www.springframework.org/schema/context
11         http://www.springframework.org/schema/context/spring-context.xsd
11         http://www.springframework.org/schema/aop
12         http://www.springframework.org/schema/aop/spring-aop.xsd
12         http://www.springframework.org/schema/tx
13         http://www.springframework.org/schema/tx/spring-tx.xsd">
13
14     <!-- 开启组件扫描 -->
15     <context:component-scan base-package="com.sofwin"></context:component-scan>
16
17     <!-- 创建 JDBCTemplate -->
18     <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
19         <constructor-arg index="0" ref="dataSource"></constructor-arg>
20     </bean>
21
22     <!-- 创建数据源 -->
23     <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
24         <property name="driverClassName" value="com.mysql.cj.jdbc.Driver">
25     </property>
```

```

25      <property name="url" value="jdbc:mysql://localhost/java1908z"></property>
26      <property name="username" value="root"></property>
27      <property name="password" value="12345678"></property>
28  </bean>
29
30
31      <!-- 定义平台事务管理器 在平台事务管理器中需要注入数据源 -->
32      <bean id="transactionManager"
33          class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
34          <property name="dataSource" ref="dataSource"></property>
35      </bean>
36      <!-- 配置事务增强 transactionManager:指定当前使用的平台事务管理器 id:增强的唯一标识 -->
37      >
38      <tx:advice transaction-manager="transactionManager" id="tx1">
39          <!-- 用于配置增强内容 -->
40          <tx:attributes>
41              <!-- 对指定方法进行增强 name:指定的方法名称 isolation:指定隔离级别
42 propagation:指定传播行为 -->
43              <tx:method name="transfer" propagation="REQUIRED" />
44          </tx:attributes>
45      </tx:advice>
46      <!-- 配置AOP -->
47      <aop:config>
48          <!-- 配置切入点 -->
49          <aop:pointcut expression="execution(*
com.sofwin.service.impl.*.transfer(..))" id="p1" />
50          <!-- 将增强应用到切入点 -->
51          <aop:advisor advice-ref="tx1" pointcut-ref="p1" />
52      </aop:config>
53
54  </beans>

```

## 企业开发的常用配置：

```

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      xmlns:t="http://www.springframework.org/schema/p"
6      xmlns:aop="http://www.springframework.org/schema/aop"
7      xmlns:tx="http://www.springframework.org/schema/tx"
8      xsi:schemaLocation="
9          http://www.springframework.org/schema/beans
10         http://www.springframework.org/schema/beans/spring-beans.xsd
11         http://www.springframework.org/schema/context
12         http://www.springframework.org/schema/context/spring-context.xsd
13         http://www.springframework.org/schema/aop
14         http://www.springframework.org/schema/aop/spring-aop.xsd
15         http://www.springframework.org/schema/tx
16         http://www.springframework.org/schema/tx/spring-tx.xsd">
17
18      <!-- 开启组件扫描 -->

```

```

15     <context:component-scan base-package="com.sofwin"></context:component-scan>
16
17     <!-- 创建 JDBCTemplate -->
18     <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
19         <constructor-arg index="0" ref="dataSource"></constructor-arg>
20     </bean>
21
22     <!-- 创建数据源 -->
23     <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
24         <property name="driverClassName" value="com.mysql.cj.jdbc.Driver">
25             <property name="url" value="jdbc:mysql://localhost/java1908z"></property>
26             <property name="username" value="root"></property>
27             <property name="password" value="12345678"></property>
28     </bean>
29
30
31     <!-- 定义平台事务管理器 在平台事务管理器中需要注入数据源 -->
32     <bean id="transactionManager"
33         class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
34         <property name="dataSource" ref="dataSource"></property>
35     </bean>
36     <!-- 配置事务增强 transactionManager:指定当前使用的平台事务管理器 id:增强的唯一标识 -->
37     <tx:advice transaction-manager="transactionManager" id="tx1">
38         <!-- 用于配置增强内容 -->
39         <tx:attributes>
40             <!-- 对指定方法进行增强 name:指定的方法名称, 可以使用通配符 isolation:指定隔离
41                 级别 propagation:指定传播行为 -->
42             <tx:method name="query*" propagation="REQUIRED" read-only="true" />
43         </tx:attributes>
44     </tx:advice>
45     <!-- 配置AOP -->
46     <aop:config>
47         <!-- 配置切入点 -->
48         <aop:pointcut expression="execution(* com.sofwin.service.impl.*.*(..))"
49             id="p1" />
49         <!-- 将增强应用到切入点 -->
50         <aop:advisor advice-ref="tx1" pointcut-ref="p1" />
51     </aop:config>
51 </beans>

```

## 基于注解方式的声明式事务

1. 开启组件扫描
2. 开启事务注解驱动
3. 在需要使用事务的类或方法上加入 `@Transactional` 注解即可

```

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      xmlns:t="http://www.springframework.org/schema/p"
6      xmlns:aop="http://www.springframework.org/schema/aop"
7      xmlns:tx="http://www.springframework.org/schema/tx"
8      xsi:schemaLocation="
9          http://www.springframework.org/schema/beans
10         http://www.springframework.org/schema/beans/spring-beans.xsd
11         http://www.springframework.org/schema/context
12         http://www.springframework.org/schema/context/spring-context.xsd
13         http://www.springframework.org/schema/aop
14         http://www.springframework.org/schema/aop/spring-aop.xsd
15         http://www.springframework.org/schema/tx
16         http://www.springframework.org/schema/tx/spring-tx.xsd">
17
18     <!-- 开启组件扫描 -->
19     <context:component-scan base-package="com.sofwin"></context:component-scan>
20
21     <!-- 创建 JDBCTemplate -->
22     <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
23         <constructor-arg index="0" ref="dataSource"></constructor-arg>
24     </bean>
25
26     <!-- 创建数据源 -->
27     <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
28         <property name="driverClassName" value="com.mysql.cj.jdbc.Driver">
29             </property>
30             <property name="url" value="jdbc:mysql://localhost/java1908z"></property>
31             <property name="username" value="root"></property>
32             <property name="password" value="12345678"></property>
33     </bean>
34
35     <!-- 定义平台事务管理器 在平台事务管理器中需要注入数据源 -->
36     <bean id="transactionManager"
37         class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
38         <property name="dataSource" ref="dataSource"></property>
39     </bean>
40     <!-- 开启事务注解驱动 transactionManager指定凭条事务管理器 proxy-target-class:是否强制使用cglib -->
41     <tx:annotation-driven proxy-target-class="true" transaction-
42         manager="transactionManager" />
43
44 </beans>

```

```

1 package com.sofwin.service.impl;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Service;
5 import org.springframework.transaction.TransactionStatus;
6 import org.springframework.transaction.annotation.Propagation;
7 import org.springframework.transaction.annotation.Transactional;
8 import org.springframework.transaction.support.TransactionCallback;

```

```

9   import org.springframework.transaction.support.TransactionTemplate;
10  import com.sofwin.dao.UserDao;
11  import com.sofwin.pojo.User;
12  import com.sofwin.service.UserService;
13  // 如果注解不增加id的话, spring默认会将类名首字母小写作为bean 的id
14  @Service
15  // 在类上增加注解, 该类的所有方法都使用这种事务
16  //@Transactional(propagation = Propagation.REQUIRED)
17  public class UserServiceImpl implements UserService {
18      @Autowired
19      private UserDao dao ;
20
21      @Override
22      public User queryUserById(Integer id) {
23          return dao.selectById(id);
24      }
25
26      @Override
27      // 注解写在方法上代表只有该方法使用这种事务
28      @Transactional(propagation = Propagation.REQUIRED)
29      public void transfer(String fromUserId, String toUserId, Integer account) {
30          // 转出的人插入 -account
31          dao.saveAccount(fromUserId, account*-1);
32          // 数据库中插入 001 -50
33          // 发送短信
34          int i=1/0;// 代码抛异常, 事务回滚, 将001 -50 这条记录回滚掉
35          // 转入的人出入 account
36          dao.saveAccount(toUserId, account);
37      }
38
39  }

```

总结：

企业开发过程中推荐使用声明式事务，对于通用的事务推荐使用基于xml方式的声明式事务。对于个别特殊方法的事务可以使用注解。并且在企业开发过程中，事务需要配置在业务层service层。不能配置在dao层。业务层的业务逻辑中可能调用n多个dao的方法。

## 第二章 Spring MVC

Spring MVC是一种基于Java的实现了WEB MVC设计模式的请求驱动类型的轻量级的web框架(MVC架构模式)，使用MVC架构模式，将web层进行解耦合的。请求驱动就是请求-响应模型。目的:帮助我们简化web开发。

主流的MVC框架:

- Struts2:采用多例模式(原型模式)
- Spring mvc采用单例模式

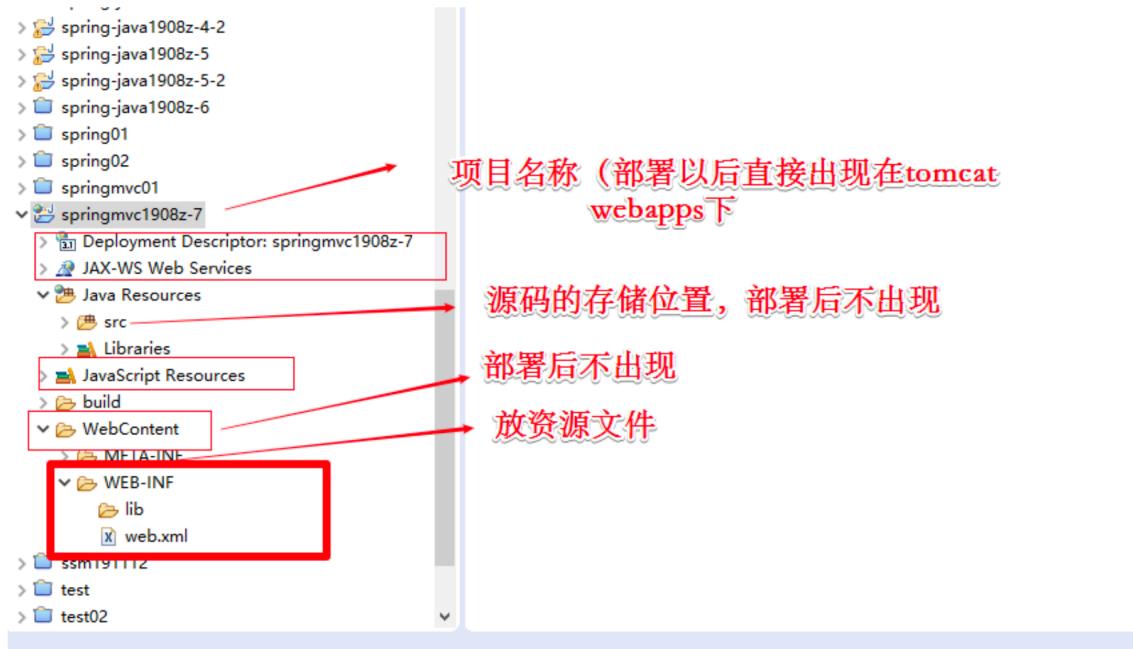
## 2.1 Spring MVC 框架搭建

通过搭建Spring MVC框架了解它是如何帮助我们简化WEB开发的

### 1. 创建Java web project

回忆：文件夹中包含WEB-INF/web.xml、WEB-INF/lib

第三方依赖：可以存放在tomcat/lib目录下，也可以存放在项目/WEB-INF/lib



### 2. 加入spring MVC核心依赖

- 核心依赖  
spring-core.jar,spring-context.jar,spring-expression.jar,spring-beans.jar
- 日志依赖  
spring-jcl.jar,log4j.jar
- 需要使用注解  
spring-aop.jar
- JUnit依赖  
junit.jar,hamcrest.jar,spring-test.jar
- spring MVC依赖  
spring-web.jar,spring-webmvc.jar

### 3. 在 `web.xml` 中配置前端控制器

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
   http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" id="WebApp_ID"
   version="3.1">
3      <display-name>springmvc-01</display-name>
4      <welcome-file-list>
```

```

5      <welcome-file>index.html</welcome-file>
6      <welcome-file>index.htm</welcome-file>
7      <welcome-file>index.jsp</welcome-file>
8      <welcome-file>default.html</welcome-file>
9      <welcome-file>default.htm</welcome-file>
10     <welcome-file>default.jsp</welcome-file>
11     </welcome-file-list>
12     <servlet>
13         <servlet-name>springmvc-01</servlet-name>
14         <servlet-
15             class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
16             <init-param>
17                 <param-name>contextConfigLocation</param-name>
18                 <param-value>classpath:applicationContext.xml</param-value>
19             </init-param>
20         </servlet>
21     <servlet-mapping>
22         <servlet-name>springmvc-01</servlet-name>
23         <url-pattern>*.do</url-pattern>
24     </servlet-mapping>
25 </web-app>

```

#### 4. 配置spring的配置文件 applicationContext.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      xmlns:t="http://www.springframework.org/schema/p"
6      xmlns:aop="http://www.springframework.org/schema/aop"
7      xsi:schemaLocation="
8          http://www.springframework.org/schema/beans
9          http://www.springframework.org/schema/beans/spring-beans.xsd
10         http://www.springframework.org/schema/context
11         http://www.springframework.org/schema/context/spring-context.xsd
12         http://www.springframework.org/schema/aop
13         http://www.springframework.org/schema/aop/spring-aop.xsd">
14
15     <!-- 开启组件扫描 -->
16     <context:component-scan base-package="com.sofwin"></context:component-
17     scan>
18 </beans>

```

#### 5. 编写控制层

```

1 package com.sofwin.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.servlet.ModelAndView;
6

```

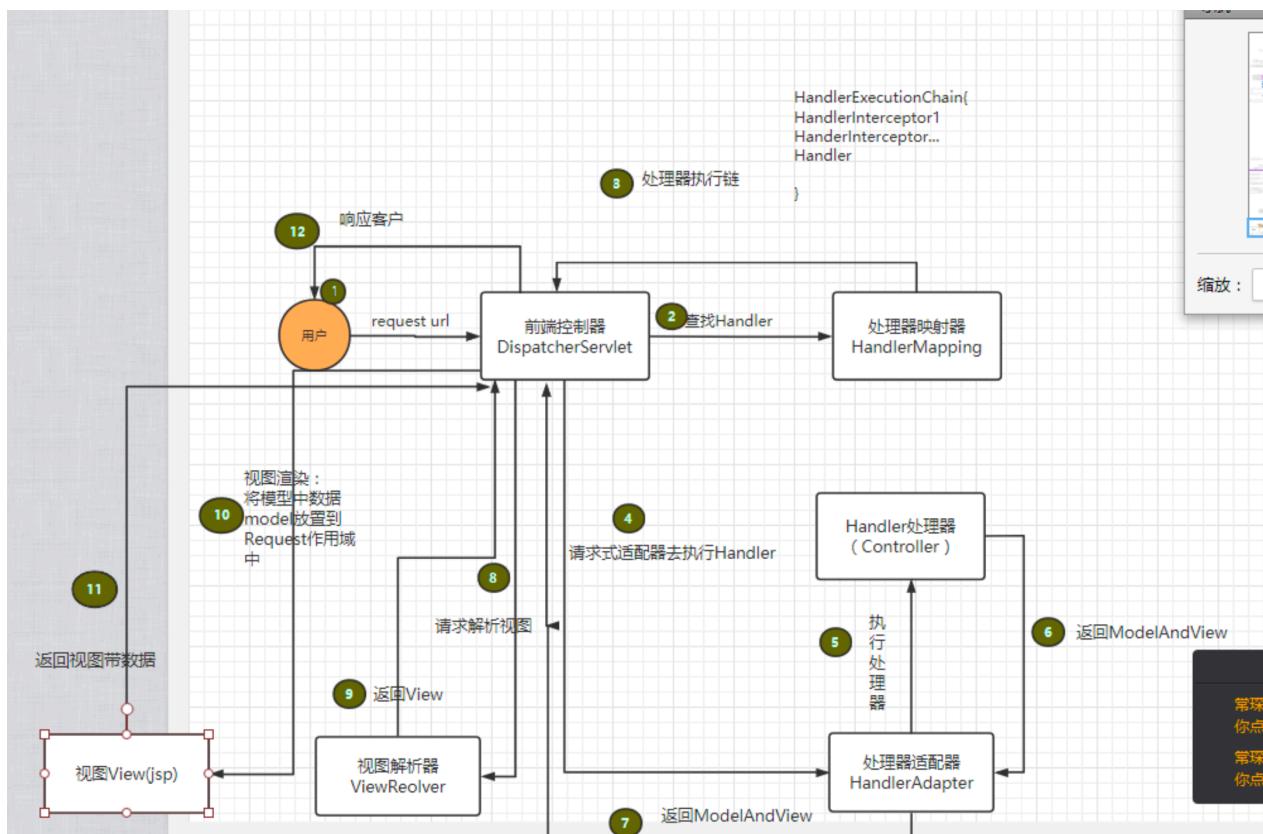
```

7  /**
8  * 测试 第一次spring MVC框架的搭建
9  * 注解方式
10 * @author dengchenyang
11 *
12 */
13 @Controller
14 public class TestController {
15
16     @RequestMapping("/test")
17     public ModelAndView test(Integer id) {
18         ModelAndView modelAndView = new ModelAndView("index.jsp");
19         modelAndView.addObject("id"+id);
20         return modelAndView;
21     }
22 }

```

## 2.2 Spring MVC运行原理（掌握）

- HandlerMapping: 处理器映射器
- HandlerAdaptor: 处理器适配器
- ViewResolver: 视图解析器
- DispatcherServlet: 前端控制器



1. 客户端发起 <https://localhost:8080/springmvc-01/test.do> 请求，请求经过Tomcat后进入到当前项目的前端控制器 (url-pattern:\*.do)
2. 前端控制器请求HandlerMapping去查找Hadler (Handler可以是XML方式配置或注解方式配置)

3. HandlerMapping返回处理器执行器链（Handler、HandlerInterceptor）给前端控制器
4. 前端控制器去调用HandlerAdaptor去执行Handler
5. HandlerAdaptor将根据适配的结果去执行Handler
6. Handler执行完成后将ModelAndView返回给HandlerAdaptor
7. HandlerAdaptor将 ModelAndView 返回给前端控制器
8. 前端控制器在拿到 ModelAndView 后去请求 ViewResolver 进行视图解析
9. ViewResolver 将解析后的视图交给前端控制器
10. 前端控制器使用model去填充或渲染视图
11. 将渲染结果返回给前端控制器
12. 前端控制器将结果响应给客户端

从代码看运行原理：

The diagram illustrates the Spring MVC request handling process. At the top, a browser screenshot shows a request to `localhost:8080/springmvc1908z-7/test.do`. Below it, the Java code for the DispatcherServlet and a TestController are shown.

**DispatcherServlet Configuration (Top):**

```

<!--
    <servlet>
        <servlet-name>springmvc1908z-7</servlet-name>
        <!-- servlet的全限定类名
            指spring提供的前端控制器
        -->
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <!-- 名称只能为contextConfigLocation -->
            <param-name>contextConfigLocation</param-name>
            <!-- 配置文件的位置 -->
            <param-value>classpath:applicationContext.xml</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>springmvc1908z-7</servlet-name>
        <!--
            路径映射
            类型映射
            详细映射
        -->
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
</web-app>

```

**TestController.java (Bottom):**

```

4 org.springframework.web.servlet.LocaleResolver=org.springframework.web.servl
5 org.springframework.web.servlet.ThemeResolver=org.springframework.web.servle
6 处理器映射器 BeanNameUrlHandlerMapping
7 org.springframework.web.servlet.HandlerMapping=org.springframework.web.servl
8 org.springframework.web.method.annotation.RequestMappingHand
9 org.springframework.web.method.annotation.ResponseStatusExceptionRe
10 org.springframework.web.servlet.function.support.RouterFunctionMapping
11 org.springframework.web.servlet.mvc.method.annotation.RequestMappingHand
12 org.springframework.web.servlet.HandlerAdapter=org.springframework.web.servl
13 org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,
14 org.springframework.web.servlet.mvc.method.annotation.RequestMappingHand
15 org.springframework.web.servlet.mvc.method.annotation.RequestMappingHand
16 org.springframework.web.servlet.function.support.HandlerFunctionAdapter
17
18
19 org.springframework.web.servlet.HandlerExceptionResolver=org.springfram
20 org.springframework.web.servlet.annotation.ResponseStatusExceptionRe
21 org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResol
22
23 org.springframework.web.servlet.RequestToViewNameTranslator=
;
```

**Annotations:**

- ①: Browser URL bar showing `localhost:8080/springmvc1908z-7/test.do`.
- ②: URL pattern `*.do` in the DispatcherServlet configuration.
- ③: DispatcherServlet class name `DispatcherServlet` in the configuration.
- ④: Internationalization annotation `@Controller` on the TestController.
- ⑤: HandlerAdapter class name `SimpleControllerHandlerAdapter`.
- ⑥: Request mapping annotation `@RequestMapping("/test")` on the `test` method.
- ⑦: ViewResolver class name `RequestToViewNameTranslator`.
- ⑧: Request mapping annotation `@RequestMapping("/test")` on the `test` method.
- ⑨: Implementation of the `test` method, creating a `ModelAndView` object and adding an attribute `"id": id`.

总结：

- DispatcherServlet: 前端控制器

不需要开发人员开发，由spring提供。作用：servlet（接受请求，响应结果）

- HandlerMapping: 处理器映射器

不需要开发人员开发，spring内置处理器映射器，默认有三种。

```
1 org.springframework.web.servlet.HandlerMapping=org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,\n2 org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping,\n3 org.springframework.web.servlet.function.support.RouterFunctionMapping
```

作用：根据请求的URL查找对应的Handler。

在spring MVC中可能有这三种处理器映射器，有至少三种控制层的配置方式，eg:BeanName,注解

- HandlerAdaptor: 处理器适配器

不需要开发人员编写，处理器适配器是为了解决处理器映射器的调用Handler的方式不同而设计的适配器模式（每一个处理器映射器适配一种执行方式）

- Handler: 处理器

需要开发人员自己编写

Handler应该按照HandlerAdapter的规则去编写： BeanNameUrlHandlerAdaptor方式，类必须实现 Controller 接口，处理器直接执行 `handlerRequest()` 方法即可；

RequestMappingHandlerAdapter:类上必须有 Controller 接口，方法上必须有 `@RequestMapping():@xxMapping()`

- ViewResolver: 视图解析器

不需要开发人员编写，开发人员可以做响应的设置（将逻辑视图转换为真正的视图）

- View: 视图

需要开发人员编写， JSP、HTML、Farmework、velocity、themleaf

## 2.3 BeanNameUrlHandlerMapping

也是spring MVC默认支持的处理器映射器

注意： Spring 可以对项目进行解耦合，同样提供的处理器映射器和处理器适配器也是为了解耦合。可以自定义处理器映射器和处理器适配器，不包含在Spring MVC默认支持的处理器映射器和处理器适配器的，需要我们手动去声明。默认的不需要再去手动声明了

1. 创建控制层：编写Java类实现Controller接口，实现`handlerRequest()`方法

```
1 package com.sofwin.controller;\n2\n3 import javax.servlet.http.HttpServletRequest;\n4 import javax.servlet.http.HttpServletResponse;\n5\n6 import org.springframework.web.servlet.ModelAndView;\n7 import org.springframework.web.servlet.mvc.Controller;\n8\n9 /**\n10  * 测试 BeanNameUrlHandlerMapping方式搭建spring MVC框架\n11  * @author dengchenyang\n12  *\n13  */\n14 public class TestController2 implements Controller{\n15 }
```

```

16     /**
17      * 类似servlet中的service()方法
18      * doGet(),doPost() 接受请求, 处理请求, 响应结果
19      */
20     @Override
21     public ModelAndView handleRequest(HttpServletRequest arg0,
22                                     HttpServletResponse arg1) throws Exception {
23         ModelAndView modelAndView = new ModelAndView("index2.jsp");
24         return modelAndView;
25     }
26 }

```

## 2. 配置控制层：将 <bean> 标签的name属性作为接受请求的url

```

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         xmlns:t="http://www.springframework.org/schema/p"
6         xmlns:aop="http://www.springframework.org/schema/aop"
7         xsi:schemaLocation="
8             http://www.springframework.org/schema/beans
9             http://www.springframework.org/schema/beans/spring-beans.xsd
10            http://www.springframework.org/schema/context
11            http://www.springframework.org/schema/context/spring-context.xsd
12            http://www.springframework.org/schema/aop
13            http://www.springframework.org/schema/aop/spring-aop.xsd">
14
15     <!-- 开启组件扫描 -->
16     <context:component-scan base-package="com.sofwin"></context:component-
17     scan>
18     <bean name="/test2.do" class="com.sofwin.controller.TestController2">
19     </bean>
20 </beans>

```

## 2.4 SimpleUrlHandlerMapping

SimpleUrlHandlerMapping并没有出现在默认的处理器映射器中，在使用时，需要独立去配置

### 1. 创建处理器:普通Java类, implements Controller

```

1 package com.sofwin.controller;
2
3 import javax.servlet.http.HttpServletRequest;
4 import javax.servlet.http.HttpServletResponse;
5
6 import org.springframework.web.servlet.ModelAndView;
7 import org.springframework.web.servlet.mvc.Controller;
8
9 /**
10  * 测试 SimpleUrlHandlerMapping的方式去搭建spring MVC框架

```

```

11     * @author dengchenyang
12     *
13     */
14 public class TestController3 implements Controller{
15
16     /**
17      * 类似servlet中的service()方法
18      * doGet(),doPost() 接受请求, 处理请求, 响应结果
19      */
20     @Override
21     public ModelAndView handleRequest(HttpServletRequest arg0,
22                                     HttpServletResponse arg1) throws Exception {
23         ModelAndView modelAndView = new ModelAndView("index3.jsp");
24         return modelAndView;
25     }
26 }
27

```

## 2. 将控制层IoC

### 3. 配置处理器映射器

```

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   xmlns:t="http://www.springframework.org/schema/p"
6   xmlns:aop="http://www.springframework.org/schema/aop"
7   xsi:schemaLocation="
8       http://www.springframework.org/schema/beans
9       http://www.springframework.org/schema/beans/spring-beans.xsd
9       http://www.springframework.org/schema/context
10      http://www.springframework.org/schema/context/spring-context.xsd
10      http://www.springframework.org/schema/aop
11      http://www.springframework.org/schema/aop/spring-aop.xsd">
12
13     <!-- 开启组件扫描 -->
14     <context:component-scan base-package="com.sofwin"></context:component-
15     scan>
16
17     <!-- 将控制层IOC -->
18     <bean id="testController3"
19       class="com.sofwin.controller.TestController3"></bean>
20
21     <!-- 配置SimpleUrlHandlerMapping, 该bean用于配置处理器映射器, 所以不需要id属性
22     -->
23     <bean
24       class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
25         <!-- 可以使用mappings来注入对应的URL -->
26         <property name="mappings">
27             <props>

```

```
23             <!-- key: 请求的URL; value: 对应的实现了Controller接口的Bean的id
24             -->
25         <prop key="/test3.do">testController3</prop>
26     </props>
27 </property>
28 </bean>
29 </beans>
```

## 2.5 配置最新的处理器映射器、处理器适配器

用于4.x,3.x

### 方式1

```
1 <!-- 兼容4.X和3.X, 需要注入RequestMappingHandlerMapping和RequestMappingHandlerAdapter--
2 >
3 <bean
4     class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerM
5         apping"></bean>
6 <bean
7     class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerA
8         dapter"></bean>
```

### 方式2

```
1 <!-- 使用最新的处理器映射器和处理器适配器 -->
2 <mvc:annotation-driven></mvc:annotation-driven>
```

## 2.6 同步请求的创建方式

@Controller 声明了该类是一个控制层的类，类中有很多方法，请求该进入那个方法呢？由方法上的注解决定！

- @RequestMapping

用于处理请求URL映射的注解，属于类和方法级别。

**注意：**当该注解放在类上时，代表命名空间。该类中的所有请求必须有前提。

```
1 /**
2 *
3 * value:请求路径, String[](可以放单个字符串, 代表某个请求) * 已数组形式放值代表 多个请求都
4 * 可以进入该方法
5 * method:RequestMethod[]
6 * 设置该请求的请求方式
7 */
8 @RequestMapping(value = {"test", "test2"}, method = {RequestMethod.POST})
```

- GetMapping

属于方法级别的注解，用于处理GET请求

```
1 @RequestMapping(Method=RequestMethod.GET)
```

- **PostMapping**

属于方法级别的注解，用于处理POST请求

```
1 @RequestMapping(Method=RequestMethod.POST)
```

- **PutMapping**

属于方法级别的注解，功能等同于PostMapping用于向服务器提交更新信息的请求

```
1 @RequestMapping(Method=RequestMethod.PUT)
```

- **DeleteMapping**

属于方法级别的注解，用于处理删除请求

```
1 @RequestMapping(Method=RequestMethod.DELETE)
```

- **PatchMapping**

属于方法级别的注解，用于处理Patch请求,显示层使用的是spring标签

```
1 @RequestMapping(Method=RequestMethod.PATCH)
```

## 2.6.1 定义返回值为ModelAndView的方法

```
1 /**
2      * ModelAndView有2个构造方法：当使用无参构造时，可以使用setViewName()来设置视图的路径；  
3      * 当使用参数为String的构造方法时，该参数即为视图的路径。  
4      * ModelAndView中的addObject()可以用来设置数据，相当于Request.setAttribute()  
5      * ModelAndView使用转发，视图默认使用的是相对定位  
6      * @return  
7      */
8      @RequestMapping("/test")
9      public ModelAndView test01() {
10         ModelAndView modelAndView = new ModelAndView("/index.jsp");
11         modelAndView.addObject("id", 100);
12         return modelAndView;
13     }
```

## 2.6.2 定义返回值为String的方法

```
1 /**
2      * @return "forward:视图名称"----转发至视图相当于return "视图名称"
3      *         "redirect:视图名称"----重定向至视图
4      */
5      @GetMapping("test02")
6      public String test02(){
7          return "redirect:/index.jsp";
8      }
```

## 2.6.3 定义无返回值void的方法

```
1  /**
2   * void 类型的方法既可以是同步方法，也可以是异步方法
3   * @throws IOException
4   * @throws ServletException
5   */
6   @GetMapping("test03")
7   public void test03(HttpServletRequest request, HttpServletResponse response)
8   throws ServletException, IOException {
9       System.out.println("testesttest");
10      request.setAttribute("id", 100);
11      request.getRequestDispatcher("/index.jsp").forward(request, response);
12 }
```

## 2.6.4 Spring MVC的内置对象

- HttpServletRequest
- HttpServletResponse
- HttpSession
- Model：等价于request@
- ModelMap：等价于request

总结：

1. 在所有控制层中不允许出现相同的路径映射
2. Model和ModelMap是spring提供的mvc框架中数据传递的对象。spring mvc中modelAndView虽然是转发，但是他并没有依赖HttpServletRequest

## 2.7 请求参数的获取方式

### 2.7.1 使用传统方式

HttpServletRequest对象，request.getParameter和request.getParameterValues来获取请求参数

### 2.7.2 简单类型的请求参数的获取

- 同名参数的获取

直接在请求方法中增加一个和请求参数同名的参数即可获取，并且可以进行自动类型转换。

多个同名参数的获取需要使用： `@RequestParam(name="test")` 注解

```

1  /**
2   * 如果请求参数为id springmvc在请求参数获取时会进行自动类型转换
3   * @param id
4   * @return
5   */
6  @GetMapping("test04")
7  public ModelAndView test04(String id) {
8      ModelAndView modelAndView = new ModelAndView("/index.jsp");
9      modelAndView.addObject("id", id);
10     return modelAndView;
11 }

```

- 不同名参数的获取

使用@RequestParam注解

```

1  /**
2   * 如果请求参数为ids, 而形参必须为id
3   * @RequestParam()用于解决请求参数和方法形参不一致的请求参数的获取, 是参数级别的方
4   * 法, 只能放在参数的声明前
5   *          name: 用于指定请求参数的名字
6   *          required: 用于指定该请求参数是否必须存在。true代表该请求参数不能为空,
7   *          反之false允许为空
8   *          defaultValue: 当请求参数为空时设置默认值
9   *          springmvc在请求参数获取时会进行自动类型转换
10  * @param id
11  * @return
12  */
13 @GetMapping("test05")
14 public ModelAndView test05(@RequestParam(name = "ids", required
15 =true, defaultValue = "800") Integer id) {
16     ModelAndView modelAndView = new ModelAndView("/index.jsp");
17     modelAndView.addObject("id", id);
18     return modelAndView;
19 }

```

### 2.7.3 pojo类型的请求参数的获取

```

1 /**
2  * 需要在方法的参数中增加Pojo类型的参数, 形参名称随意, 请求参数的名称如果能和方法中某个pojo
3  * 类型中的属性一样, 自动会将该请求参数封装到方法的参数中。
4  * @return
5  */
6 @GetMapping("test06")
7 public ModelAndView test06(User user, Role role) {
8     ModelAndView modelAndView = new ModelAndView("/index.jsp");
9     mav.addObject("user", user);
10    mav.addObject("role", role);
11    return modelAndView;
12 }

```

## 2.7.4 数组类型的请求参数的获取

```
1  /**
2   * http://localhost:8080/springmvc1908z-8/test/test5.do?
3   * userIds=1&userIds=2&userIds=3
4   */
5   @GetMapping("test05")
6   public ModelAndView test05(String[] names) {
7       ModelAndView modelAndView = new ModelAndView("/index.jsp");
8       modelAndView.addObject("names", names);
9       return modelAndView;
10 }
```

## 2.7.5 集合类型的请求参数的获取

```
1 // 获取该角色中的所有用户
2 // id userName id userName
3 @GetMapping("test7")
4 public ModelAndView test7(RoleDto role) {
5     ModelAndView mav = new ModelAndView("/index.jsp");
6     mav.addObject("users", role.getUsers());
7     return mav;
8 }
```

```
1 <form action="test/test7.do">
2     用户1:<input type="text" name="users[0].id"/><input type="text"
3     name="users[0].userName"/><br>
4     用户2:<input type="text" name="users[1].id"/><input type="text"
5     name="users[1].userName"/><br>
6     用户3:<input type="text" name="users[2].id"/><input type="text"
7     name="users[2].userName"/><br>
8     <input type="submit" value="保存"/>
9 </form>
```

## 2.8 参数类型转换

### 2.8.1 自动类型转换

spring mvc提供了主流的类型转换(String 转int ,String转float,String转double,String 日期) String转日期时， SimpleDateFormat 需要日期格式化字符串。

前提是 `<mvc:annotation-driven>` 开启mvc注解驱动，在需要转换的成员变量或参数上增加`@DateTimeFormat`注解

```
1 // http://localhost:8080/springmvc1908z-8/test/test8.do?
2     // a=1&b=2.1&c=3.1415&d=2020-01-01
3     @GetMapping("test8")
4     public ModelAndView test8(int a, float b, double c, @DateTimeFormat(pattern =
"yyyy-MM-dd") Date d) {
5         ModelAndView mav = new ModelAndView("/index.jsp");
6         return mav;
7     }
```

## 2.8.2 自定义类型转换器

1. 创建类型转换器的类，实现 Convert 接口

```
1 package com.sofwin.convert;
2
3 import java.text.ParseException;
4 import java.text.SimpleDateFormat;
5 import java.util.Date;
6 import org.springframework.core.convert.converter.Converter;
7
8 /**
9  * 创建自定义类型转换器 S:source 原类型 T:target 目标类型
10 *
11 */
12 public class StringToDateConvert implements Converter<String, Date> {
13     @Override
14     public Date convert(String source) {
15         SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
16         Date target = null;
17         try {
18             target = sdf.parse(source);
19         } catch (ParseException e) {
20             e.printStackTrace();
21         }
22         return target;
23     }
24 }
```

2. 将自定义类型转换器注入到spring提供的类型转换工厂中

```
1 <!-- 将类型转换器交给spring管理 -->
2 <bean id="stringToDate" class="com.sofwin.convert.StringToDateConvert">
</bean>
3 <!-- 注入自定义类型转换器到类型转换工厂 -->
4 <bean id="convertFactory"
class="org.springframework.format.support.FormattingConversionServiceFactory
Bean ">
5     <!-- 注入类型转换 -->
6     <property name="converters">
7         <set>
8             <ref bean="stringToDate"/>
9         </set>
10    </property>
11 </bean>
```

### 3. 将新的工厂对象注入到注解驱动中

```
1 <mvc:annotation-driven conversion-service="convertFactory"></mvc:annotation-
driven>
```

## 2.9 异步请求的创建方式

spring MVC异步请求支持JSON格式，需要加入JSON依赖

### 2.9.1 异步请求框架的搭建

- 加入Jackson的依赖
  - jackson-core.jar
  - jackson-databind.jar
  - jackson-annotation.jar
- 开启springMVC注解驱动

```
1 <!-- 开启springMVC注解驱动 -->
2 <mvc:annotation-driven></mvc:annotation-driven>
```

### 2.9.2 异步请求的定义方式

异步请求的返回值可以使任意格式，如List、Map、String

```
1 @RequestMapping("test02")
2 @ResponseBody
3 public Map test02() {
4     Map map=new HashMap();
5     map.put("id", "1");
6     map.put("username", "username");
7     return map;
8 }
```

@RequestBody

属于参数级别的注解，只能写在方法的参数上。方法的参数用来获取请求参数的，`@RequestBody`就是将异步请求的请求参数封装为pojo的。

注意：

- 请求方式必须是POST请求
- 异步请求的contentType必须是application/json
- 请求参数必须是JSON格式的字符串

```
1  <%@ page language="java" contentType="text/html; charset=UTF-8"
2      pageEncoding="UTF-8"%>
3  <!DOCTYPE html>
4  <html>
5  <head>
6  <meta charset="UTF-8">
7  <title>Insert title here</title>
8  <script type="text/javascript" src="jquery-3.5.0.min.js"></script>
9  </head>
10 <body>
11 Hello spring MVC!
12 <hr>
13 <input type="button" value="点击获取数据" onclick="data() ;" />
14 </body>
15 <script type="text/javascript">
16     /* 来一个POST请求 */
17     function data(){
18         var user={
19             id:1000,
20             username:'admin'
21         };
22         $.ajax({
23             url:'test/test04.do',
24             type:'post',
25             contentType:'application/json',
26             dataType:'json',
27             data:JSON.stringify(user),
28             success:function(ret){
29                 alert(JSON.stringify(ret))
30             }
31         });
32     }
33 </script>
34 </html>
```

```
1  /**
2   * 测试异步请求中的@RequestBody注解
3   * @param user
4   * @return
5   */
6  @RequestMapping("test04")
7  @ResponseBody
8  public User test04(@RequestBody User user) {
9      return user;
10 }
```

@ResponseBody

属于类和方法级别的注解。声明该请求返回的不是视图，而是异步请求返回的数据。并且会将List类型的返回值自动转换为JSON对象数组，将pojo类型或Map类型的返回值自动转换为JSON对象

```
1  <%@ page language="java" contentType="text/html; charset=UTF-8"
2      pageEncoding="UTF-8"%>
3  <!DOCTYPE html>
4  <html>
5  <head>
6  <meta charset="UTF-8">
7  <title>Insert title here</title>
8  <script type="text/javascript" src="jquery-3.5.0.min.js"></script>
9  </head>
10 <body>
11 Hello spring MVC!
12 <hr>
13 <input type="button" value="点击获取数据" onclick="data()"/>
14 </body>
15 <script type="text/javascript">
16     function data(){
17         /* 属于GET请求 */
18         $.getJSON("test/test03.do", function(ret){
19             alert(JSON.stringify(ret));
20         })
21     }
22 </script>
23 </html>
```

```
1      @RequestMapping("test03")
2      @ResponseBody
3      public List test03() {
4          List list =new ArrayList();
5          for (int i = 0; i < 10; i++) {
6              User user = new User();
7              user.setId(i+1);
8              user.setUsername("username"+i+1);
9              list.add(user);
10         }
11     return list;
12 }
```

```
@RestController
```

组合注解等价于 `@Controller` 加 `@ResponseBody`。该注解生命控制层中的所有方法都自动增加了 `@ResponseBody` 注解。

## 2.10 文件上传

spring提供了对 `commons-fileupload` 的支持，简化了文件上传的开发步骤

### 2.10.1 框架搭建

1. 加入依赖：

- commons-fileupload.jar
- commons-io.jar

2. 开启springMVC注解驱动：

```
1  <!-- 开启springMVC注解驱动 -->
2  <mvc:annotation-driven></mvc:annotation-driven>
```

3. 配置多媒体视图解析器：

```
1  <!-- 配置多媒体视图解析器 bean的id固定：multipartResolver -->
2  <bean id="multipartResolver"
3    class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
4      <!-- 设置文件上传的编码格式 -->
5      <property name="defaultEncoding" value="UTF-8"></property>
6      <!-- 设置文件在内存中的最大存储大小 -->
7      <property name="maxInMemorySize" value="102400"></property>
8  </bean>
```

4. 编写form表单：

```

1 <!-- 测试文件上传 -->
2 <form action="test/upload.do" method="POST" enctype="multipart/form-data" >
3 图片1: <input type="file" name="file"/>
4 <input type="submit" value="上传">
5 </form>
6
7 <!-- 测试多文件上传 -->
8 <form action="test/upload.do" method="POST" enctype="multipart/form-data" >
9 图片1: <input type="file" name="file"/>
10 图片2: <input type="file" name="file"/>
11 <input type="submit" value="上传">
12 </form>

```

## 5. 编写文件上传的请求：包括单文件上传、多文件上传

```

1 /**
2      * 处理文件上传请求，将文件上传至项目中的/upload路径中
3      * MultipartFile[]: 用于接收文件上传的请求参数, file中封装了请求参数中
4      * type="file" name="file"的信息
5      * 注意: 同名的多文件上传有需要增加@RequestParam()注解
6      * @param request
7      * @param file
8      * @return
9      * @throws IOException
10     * @throws IllegalStateException
11 */
12 @RequestMapping("upload")
13 public ModelAndView test02(HttpServletRequest request, @RequestParam(name
14 = "file") MultipartFile[] file) throws IllegalStateException, IOException {
15     ModelAndView modelAndView = new ModelAndView("/display.jsp");
16     //获取项目的绝对路径
17     String path=request.getRealPath("/")+"upload";
18     //用于保存新的文件名称
19     List<String> list = new ArrayList<String>();
20     //遍历处理文件上传参数file
21     for (MultipartFile perFile : file) {
22         //获取唯一ID
23         String fileNameTemp=UUID.randomUUID().toString();
24         //原始文件名称
25         String fileNameOld=perFile.getOriginalFilename();
26         //新的文件名称
27         String
28         fileName=fileNameTemp+fileNameOld.substring(fileNameOld.indexOf("."), fileNameOld.length());
29         File newFile = new File(path+"/"+fileName);
30         list.add(fileName);
31         //将请求参数中的文件写入到服务器中
32         perFile.transferTo(newFile);
33     }
34     modelAndView.addObject("filenames",list);
35     return modelAndView;
36 }

```

## 2.11 POST请求编码格式处理

在spring MVC中POST请求会出现中文乱码，在高版本的Tomcat中不会出现GET请求中文乱码，低版本中如果出现可以在Tomcat的配置文件中设置urlEncoding="utf-8"

POST请求中文乱码：`request.setCharacterEncoding("UTF-8")`，该代码必须放在第一行。

spring提供了一种处理POST请求中文乱码的方案：使用spring自带的过滤器-

### CharacterEncodingFilter

```
1  <!-- 配置过滤器解决post中文乱码问题 -->
2  <filter>
3      <filter-name>characterEncoding</filter-name>
4      <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-
    class>
5      <init-param>
6          <param-name>encoding</param-name>
7          <param-value>UTF-8</param-value>
8      </init-param>
9  </filter>
10 <filter-mapping>
11     <filter-name>characterEncoding</filter-name>
12     <url-pattern>/*</url-pattern>
13 </filter-mapping>
```

## 2.12 Restful风格请求

REST，即Representational State Transfer的缩写。直接翻译的意思是"表现层状态转化"。一般使用场景：电商中的商品详情(restful)

```
1  https://item.jd.com/100007218425.html
2  https://item.jd.com/100006487373.html
3  请求名称:商品编号(变化)
4  将请求参数作为*请求名称*
```

`@PathVariable`：路径变量，属于参数级别的注解，该请求参数可以作为变量出现在`@RequestMapping()`中、方法参数前

```
1  @RequestMapping("/goodsInfo/{spNo}")
2  public ModelAndView goodsInfo(@PathVariable("spNo") String spNo) {
3      ModelAndView mav = new ModelAndView("/display.jsp");
4      mav.addObject("spno", spNo);
5      return mav;
6  }
```

注意：请求名称必须唯一，考虑变量的取值范围。为了保证和其他请求不重复，一般restful风格的请求需要独立去定义namespace。

## 2.13 拦截器

Spring MVC中的拦截器（Interceptor）类似于Servlet中的过滤器（Filter），它主要用于拦截用户请求（Handler）并作相应的处理。主要完成请求参数的解析、将页面表单参数赋值给值栈中的相应属性、执行功能校验、程序异常调试等工作。

拦截器和过滤器的比较：

1. 拦截器是基于Java反射机制的，过滤器是基于函数回调
2. 拦截器不依赖于servlet容器，过滤器依赖于servlet容器
3. 拦截器只能对Handler起作用，而过滤器可以过滤所有的请求（.png/.css/.mp4）
4. 拦截器可以访问action上下文、值栈里的对象，而过滤器不能访问
5. 在action的生命周期中，拦截器可以多次被调用，而过滤器只能在容器初始化时被调用一次
6. 拦截器可以获取IOC容器中的各个bean，而过滤器就不行，这点很重要，在拦截器里注入一个service，可以调用业务逻辑

### 2.13.1 拦截器的使用

1. 创建拦截器，创建类实现接口 **HandlerInterceptor**

```
1 package com.sofwin.interceptor;
2
3 import javax.servlet.http.HttpServletRequest;
4 import javax.servlet.http.HttpServletResponse;
5
6 import org.springframework.web.servlet.HandlerInterceptor;
7 import org.springframework.web.servlet.ModelAndView;
8
9 /**
10 * 测试拦截器的使用
11 * @author dengchenyang
12 *
13 */
14 public class TestInterceptor implements HandlerInterceptor{
15
16     /**
17      * 在响应完成之后执行
18      * request:
19      * response:
20      * handler:
21      * ex:
22      */
23     @Override
24     public void afterCompletion(HttpServletRequest request,
25         HttpServletResponse response, Object handler, Exception ex)
26             throws Exception {
27         System.out.println("afterCompletion.....");
28         HandlerInterceptor.super.afterCompletion(request, response, handler,
29             ex);
30     }
31 }
```

```

31     * 在请求执行完成之后，响应之前执行
32     * request:
33     * response:
34     * ModelAndView:
35     * 获取到请求执行完成后返回的 ModelAndView 在请求执行完成后可以重新设置响应结果
36     */
37     @Override
38     public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler,
39                         ModelAndView modelAndView) throws Exception {
40         System.out.println("postHanler.....");
41         HandlerInterceptor.super.postHandle(request, response, handler,
modelAndView);
42     }
43
44     /**
45      * 在请求进入到Handler之前执行的
46      * request:
47      * response:
48      * handler:
49      * 返回值: true放行请求到Handler中去
50      */
51     @Override
52     public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler)
53             throws Exception {
54         System.out.println("proHanler.....");
55         return HandlerInterceptor.super.preHandle(request, response,
handler);
56     }
57
58 }
59

```

## 2. 配置拦截器，决定哪些请求应该进入到拦截器中

```

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         xmlns:t="http://www.springframework.org/schema/p"
6         xmlns:aop="http://www.springframework.org/schema/aop"
7         xmlns:mvc="http://www.springframework.org/schema/mvc"
8         xsi:schemaLocation="
9             http://www.springframework.org/schema/beans
10            http://www.springframework.org/schema/beans/spring-beans.xsd
11            http://www.springframework.org/schema/context
12            http://www.springframework.org/schema/context/spring-context.xsd
13            http://www.springframework.org/schema/aop
14            http://www.springframework.org/schema/aop/spring-aop.xsd
15            http://www.springframework.org/schema/mvc
16            http://www.springframework.org/schema/mvc/spring-mvc.xsd">

```

```

13
14      <!-- 开启组件扫描 -->
15      <context:component-scan base-package="com.sofwin"></context:component-
16      scan>
17          <!-- 开启springMVC注解驱动 -->
18          <mvc:annotation-driven></mvc:annotation-driven>
19          <!-- 配置多媒体视图解析器 bean的id固定: multipartResolver -->
20          <bean id="multipartResolver"
21              class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
22              <!-- 设置文件上传的编码格式 -->
23              <property name="defaultEncoding" value="UTF-8"></property>
24              <!-- 设置文件在内存中的最大存储大小 -->
25              <property name="maxInMemorySize" value="102400"></property>
26          </bean>
27          <!-- 配置多个拦截器 -->
28          <mvc:interceptors>
29              <!-- 单个拦截器的配置 -->
30              <mvc:interceptor>
31                  <!-- 配置拦截器映射: /*: 代表拦截所有一级请求; /**: 代表拦截所有请求 -->
32                  <mvc:mapping path="/**" />
33                  <!-- 在以上拦截的基础上放行的系列请求 -->
34                  <mvc:exclude-mapping path="/test/test01.do" />
35                  <!-- 配置拦截器 -->
36                  <bean class="com.sofwin.interceptor.TestInterceptor"></bean>
37                  <!-- 如果拦截器Ioc在外部的话 <ref bean="" />使用外部定义的Interceptor ->
38          </mvc:interceptor>
39      </mvc:interceptors>
40  </beans>

```

## 2.13.2 总结

**拦截器的使用场景：**

- 身份验证
- 权限验证
- 日志框架

**过滤器链：** 多个过滤器去过滤同一个请求，形成了过滤器链。过滤器链的顺序xml中是由过滤器的定义顺序去决定，annotation中是根据过滤器的名称编码排序。

**拦截器链：** 多个拦截器去拦截同一个请求，形成了拦截器链。按照 `<mvc:interceptors>` 中 `<mvc:interceptor>` 定义的先后顺序去执行，谁先定义就先去拦截。

**执行顺序：** 过滤器》拦截器》Handler。请求会先进入过滤器(过滤器链)，过滤以后doFilter进入interceptor拦截器(链)，在放行拦截后请求才进入到handle中

## 2.14 视图解析器的配置

在一个项目通常使用的是同一种视图。企业开发过程中为了保证页面不能直接被客户端访问，通常会将jsp页面放置在WEB-INF中。

```
1      <!-- 配置视图解析器 -->
2      <bean
3          class="org.springframework.web.servlet.view.InternalResourceViewResolver">
4              <!-- 注入视图前缀 视图的访问路径 会自动添加在 ModelAndView 中设置的 viewName 前面 -->
5              <property name="prefix" value="/WEB-INF/"></property>
6              <!-- 注入视图后缀 视图的类型 会自动添加在 ModelAndView 中设置的 viewName 后面 -->
7              <property name="suffix" value=".jsp"></property>
8      </bean>
```

## 2.15 全局异常处理器

当ssm项目在运行期间发生异常时，需要进行异常处理。(jsp可以使用iserrorPage来配置异常页面进行异常处理)

全局异常处理器，当请求过程中发生异常，请求会进入到全局异常处理器中(将 exception 传入)，可以在全局异常处理器中去定义如何处理异常。

### 2.15.1 全局异常处理器的使用

1. 创建异常处理器，创建类实现接口 HandlerExceptionResolver

```
1  package com.sofwin.ex;
2
3  import javax.servlet.http.HttpServletRequest;
4  import javax.servlet.http.HttpServletResponse;
5
6  import org.springframework.web.servlet.HandlerExceptionResolver;
7  import org.springframework.web.servlet.ModelAndView;
8
9  /**
10   * 创建全局异常处理器
11   * @author dengchenyang
12   *
13   */
14  public class MyExceptionResolver implements HandlerExceptionResolver {
15
16      /**
17       * 当请求发生异常后调用的方法
18       * request: 可以通过 request 获取当前 Handler 中值栈的信息
19       * response: 可以通过 response 对异常进行重定向
20       * handler: 可以获取当前请求的相关信息
21       * ex: 当前发生的异常信息
22       * 返回值: ModelAndView 可以将当前异常转发到一个自定义异常处理视图中
23       */
24      @Override
25      public ModelAndView resolveException(HttpServletRequest arg0,
HttpServletResponse arg1, Object arg2,
```

```
26             Exception arg3) {
27         ModelAndView modelAndView = new ModelAndView("ex");
28         return modelAndView;
29     }
30 }
31 }
```

## 2. 配置全局异常处理器

```
1      <!-- 配置全局异常处理器 -->
2      <bean class="com.sofwin.ex.MyExceptionResolver"></bean>
```

## 2.15.2 企业级异常处理

### 1. 使用自定义异常

```
1 package com.sofwin.ex;
2
3 public class MyException extends Exception {
4
5     /**
6      *
7      */
8     private static final long serialVersionUID = 1L;
9     private String errorCode;
10    private String errorInfo;
11
12    public MyException(String errorCode, String errorInfo) {
13        super();
14        this.errorCode = errorCode;
15        this.errorInfo = errorInfo;
16    }
17    public MyException() {
18        super();
19    }
20    public String getErrorCode() {
21        return errorCode;
22    }
23    public void setErrorCode(String errorCode) {
24        this.errorCode = errorCode;
25    }
26    public String getErrorInfo() {
27        return errorInfo;
28    }
29    public void setErrorInfo(String errorInfo) {
30        this.errorInfo = errorInfo;
31    }
32
33 }
```

### 2. 使用全局异常处理器来接受异常情况（请求）

```
1 package com.sofwin.ex;
2
3 import javax.servlet.http.HttpServletRequest;
4 import javax.servlet.http.HttpServletResponse;
5
6 import org.springframework.web.servlet.HandlerExceptionResolver;
7 import org.springframework.web.servlet.ModelAndView;
8
9 /**
10 * 创建全局异常处理器
11 *
12 * @author dengchenyang
13 *
14 */
15 public class MyExceptionResolver implements HandlerExceptionResolver {
16
17     /**
18      * 当请求发生异常后调用的方法 request: 可以通过request获取当前Handler中值栈的信息
19      * response: 可以通过response对异常进行重定向 handler: 可以获取当前请求的相关信息
20      * ex: 当前发生的异常信息
21      *      * 返回值: ModelAndView 可以将当前异常转发到一个自定义异常处理视图中
22      */
23     @Override
24     public ModelAndView resolveException(HttpServletRequest arg0,
25                                         HttpServletResponse arg1, Object arg2,
26                                         Exception ex) {
27         if (ex instanceof MyException) {
28             ModelAndView mav = new ModelAndView("ex");
29             MyException ex2 = (MyException) ex;
30             mav.addObject("code", ex2.getErrorCode());
31             mav.addObject("info", ex2.getErrorInfo());
32             return mav;
33         } else {
34             ModelAndView mav = new ModelAndView("ex");
35             return mav;
36         }
37     }
38 }
```

```
1 package com.sofwin.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.servlet.ModelAndView;
6
7 import com.sofwin.ex.MyException;
8
9 @Controller
10 @RequestMapping("test")
11 public class TestController {
```

```
12
13     /**
14      * 测试全局异常处理器
15      * @return
16      * @throws MyException
17      */
18     @RequestMapping("test03")
19     public ModelAndView test03(Integer b) throws MyException {
20         ModelAndView modelAndView = new ModelAndView("index");
21         int a=10;
22         if(b==0) throw new MyException("0000","除数不能为零");
23         if(b==100) throw new MyException("0001","除数不能为100");
24         int i=a/b;
25         return modelAndView;
26     }
27 }
```

## 第三章 SSM整合

---

### 3.1 Spring基础框架整合MyBatis

### 3.2 Spring基础框架整合MyBatis分页插件

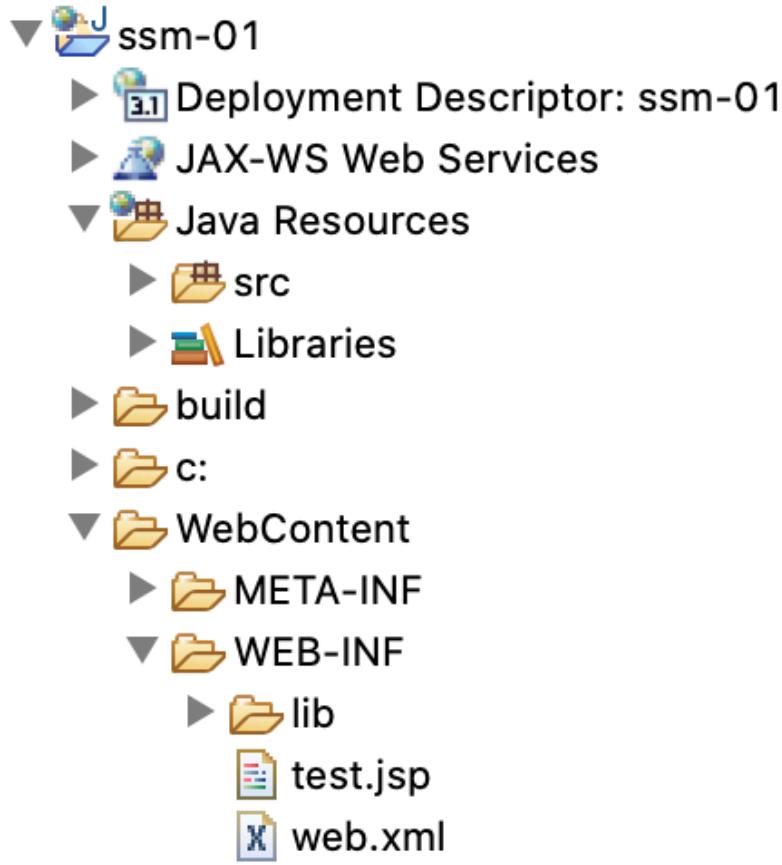
### 3.3 SSM项目

## 第四章 SSM项目

---

### 4.1 创建项目

创建Java WEB project



## 4.2 导入依赖

- Spring Framework基础框架
  - 核心依赖:
    - spring-beans.jar,spring-context.jar,spring-expression.jar,spring-core.jar
  - 日志依赖:
    - spring-jcl.jar,log4j.jar,log4j.properties
  - AOP依赖:
    - spring-aop.jar,spring-aspects.jar,aspectjweaver.jar,aopalliance.jar
  - 测试依赖:
    - spring-test.jar,junit.jar,hamcrest-core.jar
- MyBatis整合spring Framework
  - MyBatis依赖:
    - mybatis.jar,mysql-connector-java.jar,mybatis-spring.jar,druid.jar
    - pagehelper.jar,sqlparser.jar
  - Spring依赖:
    - Spring-tx.jar
    - Spring-jdbc.jar
- Spring MVC依赖
  - spring-web.jar,spring-webmvc.jar
  - jackson-core.jar,jackson-annotations.jar,jackson-databind.jar
  - commons-fileupload.jar,commons-io.jar

- jstl.jar

## 4.3 搭建Spring Framework基础框架

### 1. 创建 TestService() 接口

```
1 package com.sofwin.service;
2
3 import java.util.List;
4
5 import com.sofwin.pojo.CUser;
6
7 public interface TestService {
8
9     /**
10      * 测试方法
11      */
12     void test();
13
14     /**
15      * 测试springMVC的方法
16      */
17     List<CUser> selectCUsers();
18 }
19
```

### 2. 创建 TestServiceImpl 实现

```
1 package com.sofwin.service.impl;
2
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Service;
7
8 import com.sofwin.mapper.CUserMapper;
9 import com.sofwin.pojo.CUser;
10 import com.sofwin.service.TestService;
11
12 @Service
13 public class TestServiceImpl implements TestService {
14
15     @Autowired
16     private CUserMapper mapper;
17
18     @Override
19     public void test() {
20         System.out.println("test.....");
21     }
22
23     @Override
24     public List<CUser> selectCUsers() {
```

```
25         return mapper.selectByExample(null);
26     }
27
28 }
29
```

### 3. 创建 applicationContext.xml 配置文件，开启组件扫描

```
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      xmlns:aop="http://www.springframework.org/schema/aop"
6      xmlns:tx="http://www.springframework.org/schema/tx"
7      xsi:schemaLocation="
8          http://www.springframework.org/schema/beans
9          http://www.springframework.org/schema/beans/spring-beans.xsd
10         http://www.springframework.org/schema/context
11         http://www.springframework.org/schema/context/spring-context.xsd
12         http://www.springframework.org/schema/aop
13         http://www.springframework.org/schema/aop/spring-aop.xsd
14         http://www.springframework.org/schema/tx
15         http://www.springframework.org/schema/tx/spring-tx.xsd">
16
17     <!-- 开启组件扫描 -->
18     <context:component-scan base-package="com.sofwin"></context:component-
19     scan>
20 </beans>
```

### 4. 创建测试类 TestBaseFramework

```
1  package com.sofwin.test;
2
3  import org.junit.Test;
4  import org.junit.runner.RunWith;
5  import org.springframework.beans.factory.annotation.Autowired;
6  import org.springframework.test.context.ContextConfiguration;
7  import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
8
9  import com.sofwin.service.TestService;
10
11 /**
12  * 测试spring Framework基础框架
13  * @author dengchenyang
14  *
15  */
16 @RunWith(SpringJUnit4ClassRunner.class)
17 @ContextConfiguration(locations = "classpath:applicationContext.xml")
18 public class TestBaseFramework {
19
20     @Autowired
21     TestService service;
```

```
22
23     @Test
24     public void test01() {
25         service.test();
26     }
27 }
```

## 4.4 搭建Mybatis基础框架

1. 创建Mapper接口

使用逆向工程得到

2. 创建映射文件

使用逆向工程得到

3. 创建 `mybatis-config.xml` 全局配置文件

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE configuration
3      PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-config.dtd">
5  <configuration>
6      <!-- 加载配置文件 -->
7      <properties resource="jdbc.properties"></properties>
8      <environments default="env1">
9          <environment id="env1">
10             <transactionManager type="JDBC"></transactionManager>
11             <dataSource type="POOLED">
12                 <property name="driver" value="${jdbc.driver}" />
13                 <property name="url" value="${jdbc.url}" />
14                 <property name="username" value="${jdbc.username}" />
15                 <property name="password" value="${jdbc.pwd}" />
16             </dataSource>
17         </environment>
18     </environments>
19     <!-- 加载映射文件 -->
20     <mappers>
21         <package name="com.sofwin.mapper" />
22     </mappers>
23 </configuration>
```

4. 创建测试类 `TestMybatis`

```
1  package com.sofwin.test;
2
3  import java.io.IOException;
4  import java.io.InputStream;
5  import java.util.List;
6
7  import org.apache.ibatis.io.Resources;
8  import org.apache.ibatis.session.SqlSession;
9  import org.apache.ibatis.session.SqlSessionFactory;
```

```

10 import org.apache.ibatis.session.SqlSessionFactoryBuilder;
11 import org.junit.Test;
12
13 import com.sofwin.mapper.CUserMapper;
14 import com.sofwin.pojo.CUser;
15
16 /**
17 * 测试MyBatis框架
18 * @author dengchenyang
19 *
20 */
21 public class TestMybatis {
22
23     @Test
24     public void test01() throws IOException {
25         SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
26         InputStream is = Resources.getResourceAsStream("mybatis-
config.xml");
27         SqlSessionFactory factory = builder.build(is);
28         SqlSession session = factory.openSession();
29         CUserMapper mapper = session.getMapper(CUserMapper.class);
30         List<CUser> result = mapper.selectByExample(null);
31         for (CUser cUser : result) {
32             System.out.println(cUser.getName()+"："+cUser.getSex());
33         }
34         session.close();
35     }
36 }

```

## 4.5 整合MyBatis和Spring基础框架

### 1. 创建 applicationContext-dao.xml 配置文件

```

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   xmlns:aop="http://www.springframework.org/schema/aop"
6   xmlns:tx="http://www.springframework.org/schema/tx"
7   xsi:schemaLocation="
8       http://www.springframework.org/schema/beans
9       http://www.springframework.org/schema/beans/spring-beans.xsd
10      http://www.springframework.org/schema/context
11      http://www.springframework.org/schema/context/spring-context.xsd
12      http://www.springframework.org/schema/aop
13      http://www.springframework.org/schema/aop/spring-aop.xsd
14      http://www.springframework.org/schema/tx
15      http://www.springframework.org/schema/tx/spring-tx.xsd">
16      <!-- 加载外部配置文件 -->
17      <context:property-placeholder location="classpath:jdbc.properties"/>
18      <!-- 配置数据源 -->

```

```

15 <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
16   <property name="driverClassName" value="${jdbc.driver}"></property>
17   <property name="url" value="${jdbc.url}"></property>
18   <property name="username" value="${jdbc.username}"></property>
19   <property name="password" value="${jdbc.pwd}"></property>
20 </bean>
21 <!-- 配置SqlSessionFactoryBean -->
22 <bean class="org.mybatis.spring.SqlSessionFactoryBean">
23   <property name="dataSource" ref="dataSource"></property>
24   <!-- 设置别名 -->
25   <property name="typeAliasesPackage" value="com.sofwin.pojo"></property>
26   <!-- 加载外部的全局配置文件 -->
27   <property name="configLocation" value="classpath:mybatis-config.xml">
28 </property>
29 </bean>
30 <!-- mapper扫描 -->
31 <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
32   <property name="basePackage" value="com.sofwin.mapper"></property>
33 </bean>
</beans>

```

## 2. 测试

```

1 package com.sofwin.test;
2
3 import java.util.List;
4
5 import org.junit.Test;
6 import org.junit.runner.RunWith;
7 import org.springframework.beans.factory.annotation.Autowired;
8 import org.springframework.test.context.ContextConfiguration;
9 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
10
11 import com.sofwin.mapper.CUserMapper;
12 import com.sofwin.pojo.CUser;
13
14 /**
15  * 整合springFarmework基础框架和mybatis并测试
16  * @author dengchenyang
17  *
18  */
19 @RunWith(SpringJUnit4ClassRunner.class)
20 @ContextConfiguration(locations = "classpath:applicationContext.xml")
21 public class TestSpringFrameworkMybatis {
22
23     @Autowired
24     CUserMapper mapper;
25
26     @Test
27     public void test01() {
28         List<CUser> result = mapper.selectByExample(null);
29         for (CUser cUser : result) {

```

```
30             System.out.println(cUser.getName()+"："+cUser.getSex());
31         }
32     }
33 }
```

## 4.6 搭建spring MVC框架

### 1. 创建 `applicationContext-mvc.xml` 配置文件

```
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:mvc="http://www.springframework.org/schema/mvc"
5   xsi:schemaLocation="
6       http://www.springframework.org/schema/beans
7       http://www.springframework.org/schema/beans/spring-beans.xsd
8       http://www.springframework.org/schema/mvc
9       http://www.springframework.org/schema/mvc/spring-mvc.xsd">
10
11 <!-- 使用最新的处理器映射器和处理器适配器 -->
12 <mvc:annotation-driven></mvc:annotation-driven>
13 <!-- 配置视图解析器 -->
14 <bean
15   class="org.springframework.web.servlet.view.InternalResourceViewResolver">
16   <property name="prefix" value="/WEB-INF/"></property>
17   <property name="suffix" value=".jsp"></property>
18 </bean>
19 </beans>
```

### 2. 在 `web.xml` 中配置前端控制器

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xmlns="http://xmlns.jcp.org/xml/ns/javaee"
4   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
5   http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" id="WebApp_ID"
6   version="3.1">
7   <display-name>
8     ssm-01</display-name>
9   <welcome-file-list>
10    <welcome-file>index.html</welcome-file>
11    <welcome-file>index.htm</welcome-file>
12    <welcome-file>index.jsp</welcome-file>
13    <welcome-file>default.html</welcome-file>
14    <welcome-file>default.htm</welcome-file>
15    <welcome-file>default.jsp</welcome-file>
16  </welcome-file-list>
17
18 <!-- 配置前端控制器 -->
19 <servlet>
20   <servlet-name>ssm-01</servlet-name>
```

```

17      <servlet-
18          class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
19          <init-param>
20              <param-name>contextConfigLocation</param-name>
21              <param-value>classpath:applicationContext-
22                  dao.xml,classpath:applicationContext-
23                  mvc.xml,classpath:applicationContext.xml</param-value>
24          </init-param>
25      </servlet>
26      <servlet-mapping>
27          <servlet-name>ssm-01</servlet-name>
28          <url-pattern>*.do</url-pattern>
29      </servlet-mapping>
30      <!-- 配置过滤器，解决POST请求中文乱码 -->
31      <filter>
32          <filter-name>charactorEncoding</filter-name>
33          <filter-
34              class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
35          <init-param>
36              <param-name>encoding</param-name>
37              <param-value>UTF-8</param-value>
38          </init-param>
39      </filter>
40      <filter-mapping>
41          <filter-name>charactorEncoding</filter-name>
42          <url-pattern>/*</url-pattern>
43      </filter-mapping>
44  </web-app>

```

### 3. 创建控制层 TestSpringmvc

```

1 package com.sofwin.controller;
2
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Controller;
7 import org.springframework.web.bind.annotation.RequestMapping;
8 import org.springframework.web.servlet.ModelAndView;
9
10 import com.sofwin.pojo.CUser;
11 import com.sofwin.service.TestService;
12
13 @Controller
14 public class TestSpringmvc {
15
16     @Autowired
17     private TestService service;
18
19     @RequestMapping("test")
20     public ModelAndView test01() {
21         ModelAndView modelAndView = new ModelAndView("test");

```

```
22         List<CUser> users = service.selectCUsers();
23         modelAndView.addObject("users", users);
24         return modelAndView;
25     }
26 }
```

#### 4. 创建视图 test.jsp

```
1  <%@ page language="java" contentType="text/html; charset=UTF-8"
2      pageEncoding="UTF-8"%>
3  <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
4  <!DOCTYPE html>
5  <html>
6  <head>
7  <meta charset="UTF-8">
8  <title>Insert title here</title>
9  </head>
10 <body>
11 <h1>Test .....</h1>
12 <hr>
13 <c:forEach items="${users}" var="user">
14 ${user.username}<br>${user.pwd}<br>${user.name}<br>${user.sex}
15 <hr>
16 </c:forEach>
17 </body>
18 </html>
```

#### 5. 访问 <http://localhost:8080/ssm-01/test.do>



Test .....

---

```
admin01
admin01
张三
男
```

---

## 4.7 整合Spring+Spring MVC+Mybatis

### 1. Service 层中注入 Mapper

```
1 package com.sofwin.service.impl;
2
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Service;
7
8 import com.sofwin.mapper.CUserMapper;
9 import com.sofwin.pojo.CUser;
10 import com.sofwin.service.TestService;
11
12 @Service
13 public class TestServiceImpl implements TestService {
14
15     @Autowired
16     private CUserMapper mapper;
17
18     @Override
19     public void test() {
20         System.out.println("test.....");
21     }
22
23     @Override
24     public List<CUser> selectCUsers() {
25         return mapper.selectByExample(null);
26     }
27
28 }
```

### 2. Controller 层中注入 Service

```
1 package com.sofwin.controller;
2
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Controller;
7 import org.springframework.web.bind.annotation.RequestMapping;
8 import org.springframework.web.servlet.ModelAndView;
9
10 import com.sofwin.pojo.CUser;
11 import com.sofwin.service.TestService;
12
13 @Controller
14 public class TestSpringmvc {
15
16     @Autowired
17     private TestService service;
```

```
18
19     @RequestMapping("test")
20     public ModelAndView test01() {
21         ModelAndView modelAndView = new ModelAndView("test");
22         List<CUser> users = service.selectCUsers();
23         modelAndView.addObject("users", users);
24         return modelAndView;
25     }
26 }
```

### 3. 在前端控制器中注入配置文件

```
1 <!-- 配置前端控制器 -->
2 <servlet>
3     <servlet-name>ssm-01</servlet-name>
4     <servlet-
5         class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
6         <init-param>
7             <param-name>contextConfigLocation</param-name>
8             <param-value>classpath:applicationContext-
9                 dao.xml,classpath:applicationContext-
10                mvc.xml,classpath:applicationContext.xml</param-value>
11         </init-param>
12     </servlet>
13     <servlet-mapping>
14         <servlet-name>ssm-01</servlet-name>
15         <url-pattern>*.do</url-pattern>
16     </servlet-mapping>
```

### 4. 在视图中加入jstl

```
1 <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

## 4.8 完善

- 加入异步请求
  - 加入Jackson的依赖
  - 开启SpringMVC注解驱动，配置最新的处理器映射器、处理器适配器
  - 定义异步请求
- 加入文件上传
  - 加入commons-fileuoliad.jar,commons-io.jar
  - 开启SpringMVC注解驱动
  - 配置多媒体视图解析器
  - 编写form表单
  - 编写文件上传的请求：包括单文件上传、多文件上传
- 加入自定义类型转换器
  - 创建类型转换器的类，实现 Convert 接口
  - 将自定义类型转换器注入到spring提供的类型转换工厂中
  - 将新的工厂对象注入到注解驱动中

- 加入拦截器
  - 创建拦截器，创建类实现接口 `HandlerInterceptor`
  - 配置拦截器，决定哪些请求进入到拦截器中
- 加入全局异常处理器
  - 使用自定义异常
  - 使用全局异常处理器来接受异常情况（请求）
- spring事务的配置及测试
  - 事务的AOP需要使用 `<tx>` 标签，需要导入tx约束
  - 配置平台事务管理器
  - 配置 `<tx:advice>` 事务增强（传播行为、隔离级别）
  - 配置AOP将增强应用到切入点上
- AOP的应用