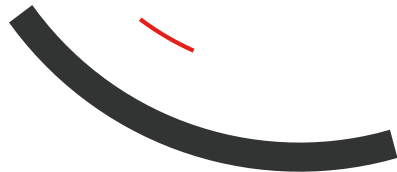




黑马程序员™  
[www.itheima.com](http://www.itheima.com)

传智播客旗下  
高端IT教育品牌



# SpringMVC入门

# 目录

# Contents

- ◆ Spring与Web环境集成
- ◆ SpringMVC的简介
- ◆ SpringMVC的组件解析

# ■ 1. Spring集成web环境

## 1.1 ApplicationContext应用上下文获取方式

应用上下文对象是通过`new ClasspathXmlApplicationContext(spring配置文件)`方式获取的，但是每次从容器中获得Bean时都要编写`new ClasspathXmlApplicationContext(spring配置文件)`，这样的弊端是配置文件加载多次，应用上下文对象创建多次。

在Web项目中，可以使用`ServletContextListener`监听Web应用的启动，我们可以在Web应用启动时，就加载Spring的配置文件，创建应用上下文对象`ApplicationContext`，在将其存储到最大的域`ServletContext`域中，这样就可以在任意位置从域中获得应用上下文`ApplicationContext`对象了。

# ■ 1. Spring集成web环境

## 1.2 Spring提供获取应用上下文的工具

上面的分析不用手动实现，Spring提供了一个监听器**ContextLoaderListener**就是对上述功能的封装，该监听器内部加载Spring配置文件，创建应用上下文对象，并存储到**ServletContext**域中，提供了一个客户端工具**WebApplicationContextUtils**供使用者获得应用上下文对象。

所以我们需要做的只有两件事：

- ① 在**web.xml**中配置**ContextLoaderListener**监听器（导入spring-web坐标）
- ② 使用**WebApplicationContextUtils**获得应用上下文对象**ApplicationContext**

# 1. Spring集成web环境

## 1.3 导入Spring集成web的坐标

```
<dependency>  
  <groupId>org.springframework</groupId>  
  <artifactId>spring-web</artifactId>  
  <version>5.0.5.RELEASE</version>  
</dependency>
```

# 1. Spring集成web环境

## 1.4 配置ContextLoaderListener监听器

```
<!--全局参数-->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
</context-param>
<!--Spring的监听器-->
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

## 纯注解方式

<!-- 基于纯注解方式: -->

<!-- 告诉ContextLoaderListener创建的Spring容器的类:

org.springframework.web.context.support.AnnotationConfigWebApplicationContext

-->

<context-param>

<param-name>contextClass</param-name>

<param-value>org.springframework.web.context.support.AnnotationConfigWebApplicationContext</param-value>

</context-param>

<!--告诉ContextLoaderListener的配置类是哪一个 -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>org.snbo.config.SpringConfiguration</param-value>

</context-param>

# 1. Spring集成web环境

## 1.5 通过工具获得应用上下文对象

```
ApplicationContext applicationContext =  
    WebApplicationContextUtils.getWebApplicationContext(servletContext);  
Object obj = applicationContext.getBean("id");
```



# ■ 1. Spring集成web环境

## 1.5 知识要点

### Spring集成web环境步骤

- ① 配置ContextLoaderListener监听器
- ② 使用WebApplicationContextUtils获得应用上下文

# 目录Contents

- ◆ Spring与Web环境集成
- ◆ SpringMVC的简介
- ◆ SpringMVC的组件解析

## ■ 2. SpringMVC 简介

### 2.1 SpringMVC概述

**SpringMVC** 是一种基于 Java 的实现 **MVC 设计模型**的请求驱动类型的轻量级 **Web 框架**，属于 **SpringFrameWork** 的后续产品，已经融合在 Spring Web Flow 中。

SpringMVC 已经成为目前最主流的MVC框架之一，并且随着Spring3.0 的发布，全面超越 Struts2，成为最优秀的 MVC 框架。它通过一套注解，让一个简单的 Java 类成为处理请求的控制器，而无须实现任何接口。同时它还支持 **RESTful** 编程风格的请求。

## ■ 2. SpringMVC 简介

### 2.3 SpringMVC快速入门

需求：客户端发起请求，服务器端接收请求，执行逻辑并进行视图跳转。

开发步骤：

- ① 导入SpringMVC相关坐标
- ② 配置SpringMVC核心控制器DispathcerServlet
- ③ 创建Controller类和视图页面
- ④ 使用注解配置Controller类中业务方法的映射地址
- ⑤ 配置SpringMVC核心文件 spring-mvc.xml
- ⑥ 客户端发起请求测试

## 2. SpringMVC 简介

### 2.2 SpringMVC快速入门

#### ① 导入Spring和SpringMVC的坐标

```
<!--Spring坐标-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.0.5.RELEASE</version>
</dependency>
<!--SpringMVC坐标-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.0.5.RELEASE</version>
</dependency>
```

## ■ 2. SpringMVC 简介

### 2.2 SpringMVC快速入门

#### ① 导入Servlet和Jsp的坐标

```
<!--Servlet坐标-->
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>servlet-api</artifactId>
  <version>2.5</version>
</dependency>
<!--Jsp坐标-->
<dependency>
  <groupId>javax.servlet.jsp</groupId>
  <artifactId>jsp-api</artifactId>
  <version>2.0</version>
</dependency>
```

## ■ 2. SpringMVC 简介

### 2.2 SpringMVC快速入门

② 在web.xml配置SpringMVC的核心控制器

```
<servlet>
    <servlet-name>DispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring-mvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>DispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

## ■ 2. SpringMVC 简介

### 2.2 SpringMVC快速入门

#### ③ 创建Controller和业务方法

```
public class QuickController {  
    public String quickMethod() {  
        System.out.println("quickMethod running....");  
        return "index";  
    }  
}
```



## 2. SpringMVC 简介

### 2.2 SpringMVC快速入门

#### ③ 创建视图页面index.jsp

```
<html>
<body>
    <h2>Hello SpringMVC!</h2>
</body>
</html>
```

## 2. SpringMVC 简介

### 2.2 SpringMVC快速入门

#### ④ 配置注解

```
@Controller
public class QuickController {
    @RequestMapping("/quick")
    public String quickMethod(){
        System.out.println("quickMethod running....");
        return "index";
    }
}
```

## ■ 2. SpringMVC 简介

### 2.2 SpringMVC快速入门

#### ⑤ 创建spring-mvc.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:mvc="http://www.springframework.org/schema/mvc"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <!--配置注解扫描-->

    <context:component-scan base-package="com.itheima"/>

</beans>
```

## 2. SpringMVC 简介

### 2.2 SpringMVC快速入门

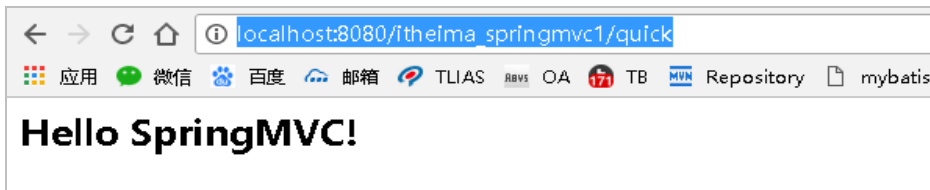
#### ⑥ 访问测试地址

```
http://localhost:8080/itheima_springmvc1/quick
```

控制台打印

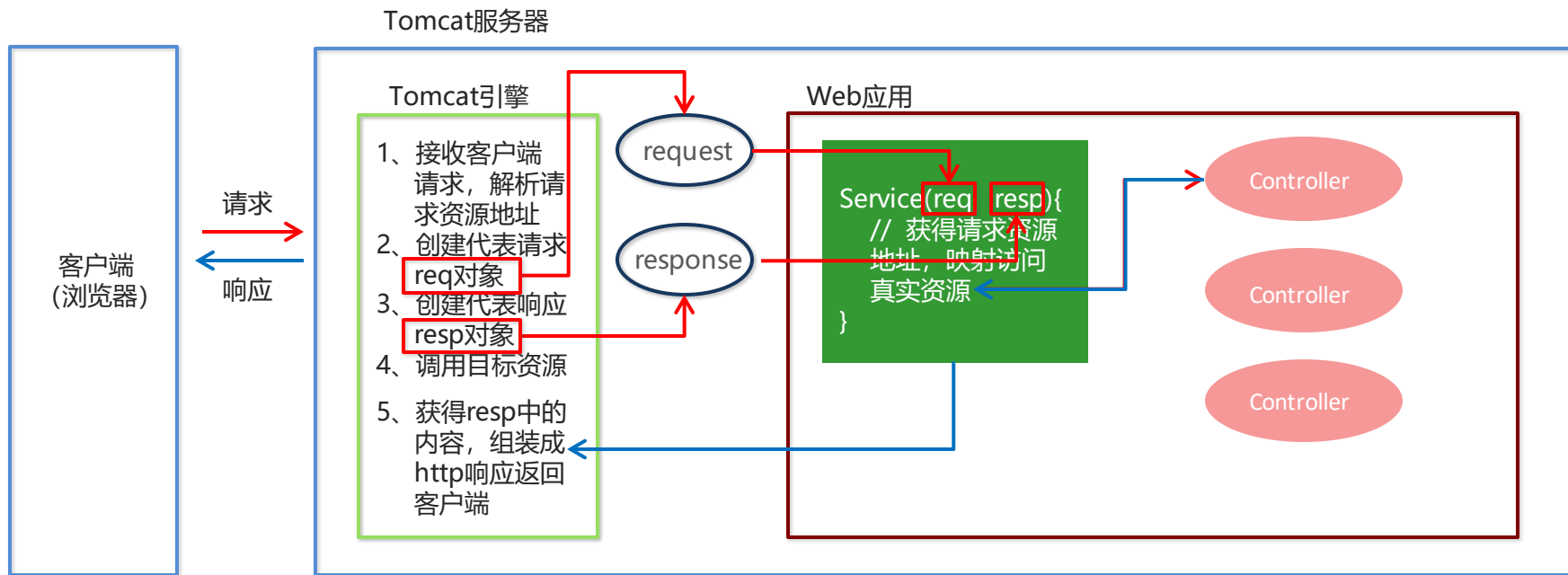
```
quickMethod running.....
```

页面显示



## 2. SpringMVC 简介

### 2.3 SpringMVC流程图示



## ■ 2. SpringMVC 简介

### 2.4 知识要点

#### SpringMVC的开发步骤

- ① 导入SpringMVC相关坐标
- ② 配置SpringMVC核心控制器DispathcerServlet
- ③ 创建Controller类和视图页面
- ④ 使用注解配置Controller类中业务方法的映射地址
- ⑤ 配置SpringMVC核心文件 spring-mvc.xml
- ⑥ 客户端发起请求测试

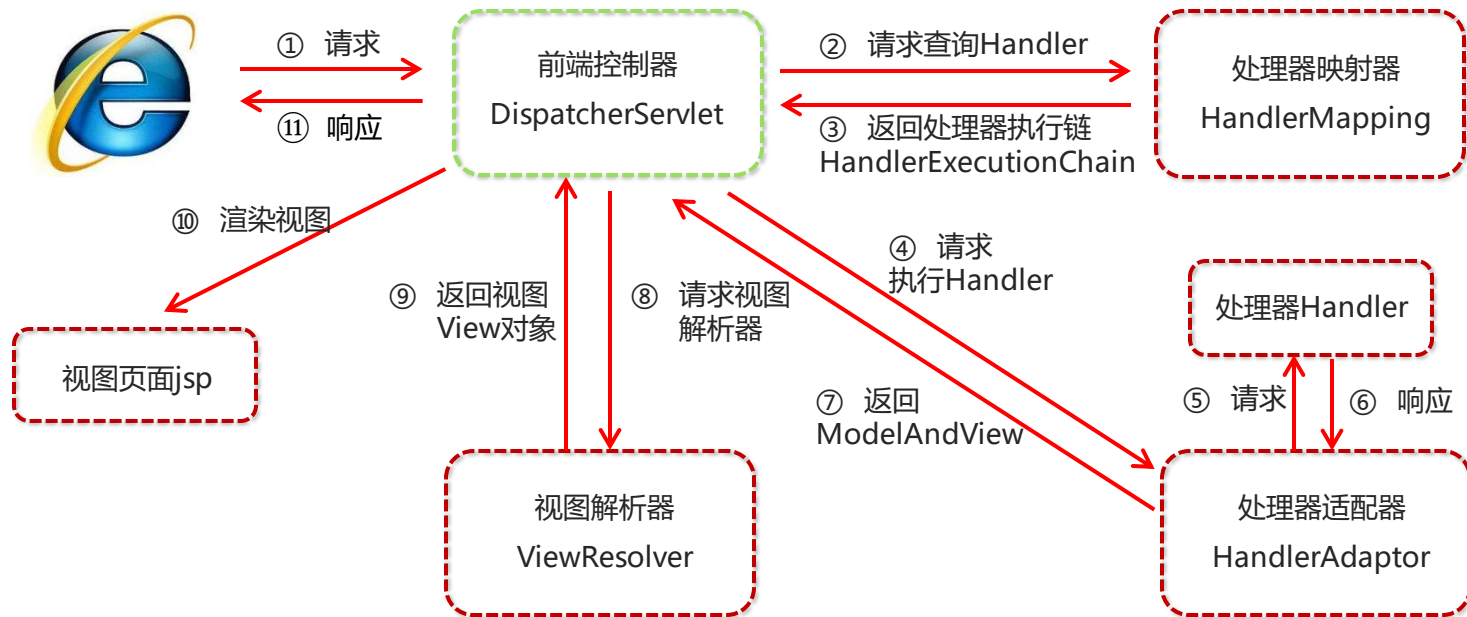
# 目录

# Contents

- ◆ Spring与Web环境集成
- ◆ SpringMVC的简介
- ◆ SpringMVC的组件解析

## 3. SpringMVC 组件解析

### 3.1 SpringMVC的执行流程





## ■ 3. SpringMVC 组件解析

### 3.1 SpringMVC的执行流程

- ① 用户发送请求至前端控制器DispatcherServlet。
- ② DispatcherServlet收到请求调用HandlerMapping处理器映射器。
- ③ 处理器映射器找到具体的处理器(可以根据xml配置、注解进行查找)，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。
- ④ DispatcherServlet调用HandlerAdapter处理器适配器。
- ⑤ HandlerAdapter经过适配调用具体的处理器(Controller，也叫后端控制器)。
- ⑥ Controller执行完成返回ModelAndView。
- ⑦ HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet。
- ⑧ DispatcherServlet将ModelAndView传给ViewReslover视图解析器。
- ⑨ ViewReslover解析后返回具体View。
- ⑩ DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）。DispatcherServlet响应用户。

## ■ 3. SpringMVC 组件解析

### 3.2 SpringMVC组件解析

#### 1. 前端控制器: DispatcherServlet

用户请求到达前端控制器，它就相当于 MVC 模式中的 C，DispatcherServlet 是整个流程控制的中心，由它调用其它组件处理用户的请求，DispatcherServlet 的存在降低了组件之间的耦合性。

#### 2. 处理器映射器: HandlerMapping

HandlerMapping 负责根据用户请求找到 Handler 即处理器，SpringMVC 提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

#### 3. 处理器适配器: HandlerAdapter

通过 HandlerAdapter 对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。

## ■ 3. SpringMVC 组件解析

### 3.2 SpringMVC组件解析

#### 4. 处理器: Handler

它就是我们开发中要编写的具体业务控制器。由 DispatcherServlet 把用户请求转发到 Handler。由 Handler 对具体的用户请求进行处理。

#### 5. 视图解析器: View Resolver

View Resolver 负责将处理结果生成 View 视图, View Resolver 首先根据逻辑视图名解析成物理视图名, 即具体的页面地址, 再生成 View 视图对象, 最后对 View 进行渲染将处理结果通过页面展示给用户。

#### 6. 视图: View

SpringMVC 框架提供了很多的 View 视图类型的支持, 包括: jstlView、freemarkerView、pdfView等。最常用的视图就是 jsp。一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户, 需要由程序员根据业务需求开发具体的页面

## ■ 3. SpringMVC 组件解析

### 3.3 SpringMVC注解解析

#### @RequestMapping

作用：用于建立请求 URL 和处理请求方法之间的对应关系

位置：

- 类上，请求URL 的第一级访问目录。此处不写的话，就相当于应用的根目录
- 方法上，请求 URL 的第二级访问目录，与类上的使用@RequestMapping标注的一级目录一起组成访问虚拟路径

属性：

- **value**：用于指定请求的URL。它和path属性的作用是一样的
- **method**：用于指定请求的方式
- **params**：用于指定限制请求参数的条件。它支持简单的表达式。要求请求参数的key和value必须和配置的一模一样

例如：

- **params = {"accountName"}**，表示请求参数必须有accountName
- **params = {"money!100"}**，表示请求参数中money不能是100

## 3. SpringMVC 组件解析

### 3.3 SpringMVC注解解析

#### 1. mvc命名空间引入

命名空间: `xmlns:context="http://www.springframework.org/schema/context"`

`xmlns:mvc="http://www.springframework.org/schema/mvc"`

约束地址: `http://www.springframework.org/schema/context`

`http://www.springframework.org/schema/context/spring-context.xsd`

`http://www.springframework.org/schema/mvc`

`http://www.springframework.org/schema/mvc/spring-mvc.xsd`

#### 2. 组件扫描

SpringMVC基于Spring容器,所以在进行SpringMVC操作时,需要将Controller存储到Spring容器中,如果使用@Controller注解标注的话,就需要使用`<context:component-scan base-package="com.itheima.controller"/>`进行组件扫描。

## 3. SpringMVC 组件解析

### 3.4 SpringMVC的XML配置解析

#### 1. 视图解析器

SpringMVC有默认组件配置，默认组件都是`DispatcherServlet.properties`配置文件中配置的，该配置文件地址 `org/springframework/web/servlet/DispatcherServlet.properties`，该文件中配置了默认的视图解析器，如下：

```
org.springframework.web.servlet.ViewResolver=org.springframework.web.servlet.view.InternalResourceViewResolver
```

翻看该解析器源码，可以看到该解析器的默认设置，如下：

```
REDIRECT_URL_PREFIX = "redirect:"    --重定向前缀  
FORWARD_URL_PREFIX  = "forward:"     --转发前缀（默认值）  
prefix = "";          --视图名称前缀  
suffix = "";          --视图名称后缀
```

## 3. SpringMVC 组件解析

### 3.4 SpringMVC的XML配置解析

#### 1. 视图解析器

我们可以通过属性注入的方式修改视图的前后缀

*<!--配置内部资源视图解析器-->*

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/"></property>
    <property name="suffix" value=".jsp"></property>
</bean>
```

## ■ 3. SpringMVC 组件解析

### 3.5 知识要点

#### SpringMVC的相关组件

- 前端控制器: DispatcherServlet
- 处理器映射器: HandlerMapping
- 处理器适配器: HandlerAdapter
- 处理器: Handler
- 视图解析器: View Resolver
- 视图: View

#### SpringMVC的注解和配置

- 请求映射注解: @RequestMapping
- 视图解析器配置:

```
REDIRECT_URL_PREFIX = "redirect:"
```

```
FORWARD_URL_PREFIX = "forward:"
```

```
prefix = "";
```

```
suffix = "";
```





传智播客旗下高端IT教育品牌



黑马程序员™  
www.itheima.com

传智播客旗下  
高端IT教育品牌



# SpringMVC的请求和响应

# 目录Contents

- ◆ SpringMVC的数据响应
- ◆ SpringMVC获得请求数据

# ■ 1. SpringMVC的数据响应

## 1.1 SpringMVC的数据响应方式

### 1) 页面跳转

- 直接返回字符串
- 通过ModelAndView对象返回

### 2) 回写数据

- 直接返回字符串
- 返回对象或集合

# 1. SpringMVC的数据响应

## 1.2 页面跳转

### 1. 返回字符串形式

直接返回字符串：此种方式会将返回的字符串与视图解析器的前后缀拼接后跳转。

```
@RequestMapping("/quick")  
public String quickMethod(){  
    return "index";  
}
```

```
<property name="prefix" value="/WEB-INF/views/" />  
<property name="suffix" value=".jsp" />
```

转发资源地址: `/WEB-INF/views/index.jsp`

返回带有前缀的字符串：

转发: `forward:/WEB-INF/views/index.jsp`

重定向: `redirect:/index.jsp`

# 1. SpringMVC的数据响应

## 1.2 页面跳转

### 2. 返回ModelAndView对象

```
@RequestMapping("/quick2")
public ModelAndView quickMethod2(){
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.setViewName("redirect:index.jsp");
    return modelAndView;
}

@RequestMapping("/quick3")
public ModelAndView quickMethod3(){
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.setViewName("forward:/WEB-INF/views/index.jsp");
    return modelAndView;
}
```

# 1. SpringMVC的数据响应

## 1.2 页面跳转

### 3. 向request域存储数据

在进行转发时，往往要向request域中存储数据，在jsp页面中显示，那么Controller中怎样向request域中存储数据呢？

① 通过SpringMVC框架注入的request对象setAttribute()方法设置

```
@RequestMapping("/quick")  
  
public String quickMethod(HttpServletRequest request){  
    request.setAttribute("name", "zhangsan");  
    return "index";  
}
```

# 1. SpringMVC的数据响应

## 1.2 页面跳转

### 3. 向request域存储数据

② 通过ModelAndView的addObject()方法设置

```
@RequestMapping("/quick3")  
  
public ModelAndView quickMethod3() {  
    ModelAndView modelAndView = new ModelAndView();  
    modelAndView.setViewName("forward:/WEB-INF/views/index.jsp");  
    modelAndView.addObject("name", "lisi");  
    return modelAndView;  
}
```



# 1. SpringMVC的数据响应

## 1.3 回写数据

### 1. 直接返回字符串

Web基础阶段，客户端访问服务器端，如果想直接回写字符串作为响应体返回的话，只需要使用 `response.getWriter().print( "hello world" )` 即可，那么在Controller中想直接回写字符串该怎样呢？

- ① 通过SpringMVC框架注入的response对象，使用 `response.getWriter().print( "hello world" )` 回写数据，此时不需要视图跳转，业务方法返回值为void。

```
@RequestMapping ( "/quick4" )  
  
public void quickMethod4 ( HttpServletResponse response ) throws  
IOException {  
    response.getWriter (). print ( "hello world" );  
}
```

# 1. SpringMVC的数据响应

## 1.3 回写数据

### 1. 直接返回字符串

- ② 将需要回写的字符串直接返回，但此时需要通过 `@ResponseBody` 注解告知SpringMVC框架，方法返回的字符串不是跳转是直接在http响应体中返回。

```
@RequestMapping("/quick5")  
@ResponseBody  
public String quickMethod5() throws IOException {  
    return "hello springMVC!!!";  
}
```

# 1. SpringMVC的数据响应

## 1.3 回写数据

### 1. 直接返回字符串

在异步项目中，客户端与服务器端往往要进行json格式字符串交互，此时我们可以手动拼接json字符串返回。

```
@RequestMapping("/quick6")
@ResponseBody
public String quickMethod6() throws IOException {
    return "{\"name\":\"zhangsan\",\"age\":18}";
}
```

# 1. SpringMVC的数据响应

## 1.3 回写数据

### 1. 直接返回字符串

上述方式手动拼接json格式字符串的方式很麻烦，开发中往往要将复杂的java对象转换成json格式的字符串，我们可以使用web阶段学习过的json转换工具jackson进行转换，导入jackson坐标。

```
<!--jackson-->  
<dependency>  
  <groupId>com.fasterxml.jackson.core</groupId>  
  <artifactId>jackson-core</artifactId>  
  <version>2.9.0</version>  
</dependency>
```

# 1. SpringMVC的数据响应

## 1.3 回写数据

### 1. 直接返回字符串

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.9.0</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-annotations</artifactId>
  <version>2.9.0</version>
</dependency>
```

# 1. SpringMVC的数据响应

## 1.3 回写数据

### 1. 直接返回字符串

通过jackson转换json格式字符串，回写字符串。

```
@RequestMapping("/quick7")
@ResponseBody
public String quickMethod7() throws IOException {
    User user = new User();
    user.setUsername("zhangsan");
    user.setAge(18);
    ObjectMapper objectMapper = new ObjectMapper();
    String s = objectMapper.writeValueAsString(user);
    return s;
}
```

# 1. SpringMVC的数据响应

## 1.3 回写数据

### 2. 返回对象或集合

通过SpringMVC帮助我们对对象或集合进行json字符串的转换并回写，为处理器适配器配置消息转换参数，指定使用jackson进行对象或集合的转换，因此需要在spring-mvc.xml中进行如下配置：

```
<bean class="org.springframework.web.servlet.mvc.method.annotation
    .RequestMappingHandlerAdapter">
    <property name="messageConverters">
        <list>
            <bean class="org.springframework.http.converter.json
                .MappingJackson2HttpMessageConverter">
            </bean>
        </list>
    </property>
</bean>
```

# 1. SpringMVC的数据响应

## 1.3 回写数据

### 2. 返回对象或集合

```
@RequestMapping("/quick8")
@ResponseBody
public User quickMethod8() throws IOException {
    User user = new User();
    user.setUsername("zhangsan");
    user.setAge(18);
    return user;
}
```



# 1. SpringMVC的数据响应

## 1.3 回写数据

### 2. 返回对象或集合

在方法上添加@**ResponseBody**就可以返回json格式的字符串，但是这样配置比较麻烦，配置的代码比较多，因此，我们可以使用mvc的注解驱动代替上述配置。

```
<!--mvc的注解驱动-->  
<mvc:annotation-driven/>
```

在 SpringMVC 的各个组件中，**处理器映射器**、**处理器适配器**、**视图解析器**称为 SpringMVC 的三大组件。使用<mvc:annotation-driven>自动加载 RequestMappingHandlerMapping（处理映射器）和 RequestMappingHandlerAdapter（处理适配器），可用在Spring-xml.xml配置文件中使使用<mvc:annotation-driven>替代注解处理器和适配器的配置。同时使用<mvc:annotation-driven>默认底层就会集成jackson进行对象或集合的json格式字符串的转换。

# ■ 1. SpringMVC的数据响应

## 1.4 知识要点

### SpringMVC的数据响应方式

#### 1) 页面跳转

- 直接返回字符串
- 通过ModelAndView对象返回

#### 2) 回写数据

- 直接返回字符串
- 返回对象或集合

# 目录Contents

- ◆ SpringMVC的数据响应
- ◆ SpringMVC获得请求数据

## ■ 2. SpringMVC 获得请求数据

### 2.1 获得请求参数

客户端请求参数的格式是：**name=value&name=value... ..**

服务器端要获得请求的参数，有时还需要进行数据的封装，SpringMVC可以接收如下类型的参数：

- 基本类型参数
- POJO类型参数
- 数组类型参数
- 集合类型参数

## ■ 2. SpringMVC 获得请求数据

### 2.2 获得基本类型参数

Controller中的业务方法的参数名称要与请求参数的name一致，参数值会自动映射匹配。

```
http://localhost:8080/itheima_springmvc1/quick9?username=zhangsan&age=12
```

```
@RequestMapping("/quick9")
@ResponseBody
public void quickMethod9(String username,int age) throws IOException {
    System.out.println(username);
    System.out.println(age);
}
```

## ■ 2. SpringMVC 获得请求数据

### 2.3 获得POJO类型参数

Controller中的业务方法的POJO参数的属性名与请求参数的name一致，参数值会自动映射匹配。

```
http://localhost:8080/itheima_springmvc1/quick9?username=zhangsan&age=12
```

```
public class User {  
    private String username;  
    private int age;  
    getter/setter...  
}  
  
@RequestMapping("/quick10")  
@ResponseBody  
public void quickMethod10(User user) throws IOException {  
    System.out.println(user);  
}
```

## ■ 2. SpringMVC 获得请求数据

### 2.4 获得数组类型参数

Controller中的业务方法数组名称与请求参数的name一致，参数值会自动映射匹配。

```
http://localhost:8080/itheima_springmvc1/quick11?strs=111&strs=222&strs=333
```

```
@RequestMapping("/quick11")
@ResponseBody
public void quickMethod11(String[] strs) throws IOException {
    System.out.println(Arrays.asList(strs));
}
```

## 2. SpringMVC 获得请求数据

### 2.5 获得集合类型参数

获得集合参数时，要将集合参数包装到一个POJO中才可以。

```
<form action="${pageContext.request.contextPath}/quick12" method="post">
    <input type="text" name="userList[0].username"><br>
    <input type="text" name="userList[0].age"><br>
    <input type="text" name="userList[1].username"><br>
    <input type="text" name="userList[1].age"><br>
    <input type="submit" value="提交"><br>
</form>
```

```
@RequestMapping("/quick12")
@ResponseBody
public void quickMethod12(Vo vo) throws IOException {
    System.out.println(vo.getUserList());
}
```



## ■ 2. SpringMVC 获得请求数据

### 2.5 获得集合类型参数

当使用ajax提交时，可以指定contentType为json形式，那么在方法参数位置使用@RequestBody可以直接接收集合数据而无需使用POJO进行包装。

```
<script>
    //模拟数据
    var userList = new Array();
    userList.push({username: "zhangsan", age: "20"});
    userList.push({username: "lisi", age: "20"});
    $.ajax({
        type: "POST",
        url: "/itheima_springmvc1/quick13",
        data: JSON.stringify(userList),
        contentType : 'application/json;charset=utf-8'
    });
</script>
```

## ■ 2. SpringMVC 获得请求数据

### 2.5 获得集合类型参数

当使用ajax提交时，可以指定contentType为json形式，那么在方法参数位置使用@RequestBody可以直接接收集合数据而无需使用POJO进行包装。

```
@RequestMapping("/quick13")
@ResponseBody
public void quickMethod13(@RequestBody List<User> userList) throws
IOException {
    System.out.println(userList);
}
```

## ■ 2. SpringMVC 获得请求数据

### 2.5 获得集合类型参数

注意：通过谷歌开发者工具抓包发现，没有加载到jquery文件，原因是SpringMVC的前端控制器DispatcherServlet的url-pattern配置的是/,代表对所有的资源都进行过滤操作，我们可以通过以下两种方式指定放行静态资源：

- 在spring-mvc.xml配置文件中指定放行的资源

```
<mvc:resources mapping="/js/**" location="/js/" />
```
- 使用<mvc:default-servlet-handler/>标签

## ■ 2. SpringMVC 获得请求数据

### 2.6 请求数据乱码问题

当post请求时，数据会出现乱码，我们可以设置一个过滤器来进行编码的过滤。

```
<filter>
  <filter-name>CharacterEncodingFilter</filter-name>
  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>CharacterEncodingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

## ■ 2. SpringMVC 获得请求数据

### 2.7 参数绑定注解@RequestParam

当请求的参数名称与Controller的业务方法参数名称不一致时，就需要通过@RequestParam注解显示的绑定。

```
<form action="${pageContext.request.contextPath}/quick14" method="post">
    <input type="text" name="name"><br>
    <input type="submit" value="提交"><br>
</form>
```

```
@RequestMapping("/quick14")
@ResponseBody
public void quickMethod14(@RequestParam("name") String username) throws
IOException {
    System.out.println(username);
}
```

## ■ 2. SpringMVC 获得请求数据

### 2.7 参数绑定注解@RequestParam

注解@RequestParam还有如下参数可以使用：

- **value**：与请求参数名称
- **required**：此在指定的请求参数是否必须包括，默认是true，提交时如果没有此参数则报错
- **defaultValue**：当没有指定请求参数时，则使用指定的默认值赋值

```
@RequestMapping("/quick14")
@ResponseBody
public void quickMethod14(@RequestParam(value="name",required =
false,defaultValue = "itcast") String username) throws IOException {
    System.out.println(username);
}
```

## ■ 2. SpringMVC 获得请求数据

### 2.8 获得Restful风格的参数

**Restful**是一种软件**架构风格、设计风格**，而不是标准，只是提供了一组设计原则和约束条件。主要用于客户端和服务  
器交互类的软件，基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存机制等。

**Restful**风格的请求是使用 **“url+请求方式”** 表示一次请求目的的，HTTP 协议里面四个表示操作方式的动词如下：

- GET：用于获取资源
- POST：用于新建资源
- PUT：用于更新资源
- DELETE：用于删除资源

例如：

- /user/1 GET： 得到 id = 1 的 user
- /user/1 DELETE： 删除 id = 1 的 user
- /user/1 PUT： 更新 id = 1 的 user
- /user POST： 新增 user

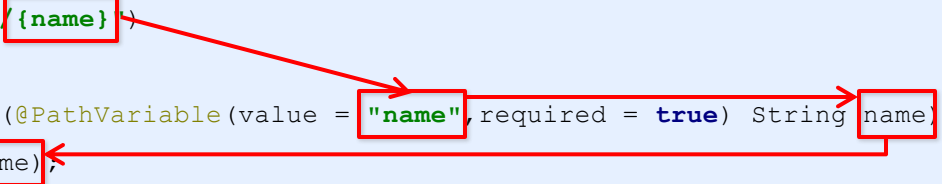
## 2. SpringMVC 获得请求数据

### 2.8 获得Restful风格的参数

上述url地址/user/1中的1就是要获得的请求参数，在SpringMVC中可以使用占位符进行参数绑定。地址/user/1可以写成/user/{id}，占位符{id}对应的就是1的值。在业务方法中我们可以使用@PathVariable注解进行占位符的匹配获取工作。

```
http://localhost:8080/itheima_springmvc1/quick19/zhangsan
```

```
@RequestMapping("/quick19/{name}")
@ResponseBody
public void quickMethod19(@PathVariable(value = "name", required = true) String name) {
    System.out.println(name);
}
```





## ■ 2. SpringMVC 获得请求数据

### 2.9 自定义类型转换器

- SpringMVC 默认已经提供了一些常用的类型转换器，例如客户端提交的字符串转换成int型进行参数设置。
- 但是不是所有的数据类型都提供了转换器，没有提供的就需要自定义转换器，例如：日期类型的数据就需要自定义转换器。

自定义类型转换器的开发步骤：

- ① 定义转换器类实现Converter接口
- ② 在配置文件中声明转换器
- ③ 在<annotation-driven>中引用转换器

## ■ 2. SpringMVC 获得请求数据

### 2.9 自定义类型转换器

#### ① 定义转换器类实现Converter接口

```
public class DateConverter implements Converter<String, Date>{  
    @Override  
    public Date convert(String source) {  
        SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd");  
        try {  
            Date date = format.parse(source);  
            return date;  
        } catch (ParseException e) {  
            e.printStackTrace();  
        }  
        return null;  
    }  
}
```

## ■ 2. SpringMVC 获得请求数据

### 2.9 自定义类型转换器

#### ② 在配置文件中声明转换器

```
<bean id="converterService"
      class="org.springframework.context.support.ConversionServiceFactoryBean">
  <property name="converters">
    <list>
      <bean class="com.itheima.converter.DateConverter"/>
    </list>
  </property>
</bean>
```

#### ③ 在<annotation-driven>中引用转换器

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

## ■ 2. SpringMVC 获得请求数据

### 2.10 获得Servlet相关API

SpringMVC支持使用原始ServletAPI对象作为控制器方法的参数进行注入，常用的对象如下：

- HttpServletRequest
- HttpServletResponse
- HttpSession

```
@RequestMapping("/quick16")
@ResponseBody
public void quickMethod16(HttpServletRequest request, HttpServletResponse
response,
    HttpSession session){
    System.out.println(request);
    System.out.println(response);
    System.out.println(session);
}
```

## ■ 2. SpringMVC 获得请求数据

### 2.11 获得请求头

#### 1. @RequestHeader

使用@RequestHeader可以获得请求头信息，相当于web阶段学习的request.getHeader(name)

@RequestHeader注解的属性如下：

- **value**：请求头的名称
- **required**：是否必须携带此请求头

```
@RequestMapping("/quick17")
@ResponseBody
public void quickMethod17(
    @RequestHeader(value = "User-Agent", required = false) String
    headerValue) {
    System.out.println(headerValue);
}
```

## ■ 2. SpringMVC 获得请求数据

### 2.11 获得请求头

#### 2. @CookieValue

使用@CookieValue可以获得指定Cookie的值

@CookieValue注解的属性如下：

- **value**：指定cookie的名称
- **required**：是否必须携带此cookie

```
@RequestMapping("/quick18")
@ResponseBody
public void quickMethod18(
    @CookieValue(value = "JSESSIONID", required = false) String jsessionId){
    System.out.println(jsessionId);
}
```

## 2. SpringMVC 获得请求数据

### 2.12 文件上传

#### 1. 文件上传客户端三要素

- 表单项type= "file"
- 表单的提交方式是post
- 表单的enctype属性是多部分表单形式，及enctype= "multipart/form-data"

```
<form action="${pageContext.request.contextPath}/quick20" method="post"
    enctype="multipart/form-data">
    名称: <input type="text" name="name"><br>
    文件: <input type="file" name="file"><br>
    <input type="submit" value="提交"><br>
</form>
```

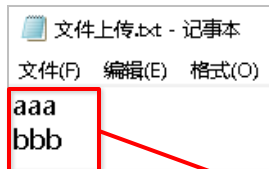
## 2. SpringMVC 获得请求数据

### 2.12 文件上传

#### 2. 文件上传原理

- 当form表单修改为多部分表单时, request.getParameter()将失效。
- enctype= "application/x-www-form-urlencoded" 时, form表单的正文内容格式是:  
**key=value&key=value&key=value**
- 当form表单的enctype取值为Multipart/form-data时, 请求正文内容就变成多部分形式:

```
<input type="text" name="name"/>  
<input type="file" name="file"/>
```



```
-----7de1a433602ac  
Content-Disposition: form-data; name="name"  
  
zhangsan  
  
-----7de1a433602ac  
Content-Disposition: form-data; name="file";  
filename="C:\Users\muzimoo\Desktop\文件上传.txt"  
Content-Type: text/plain  
  
aaa  
bbb  
  
-----7de1a433602ac--
```



## ■ 2. SpringMVC 获得请求数据

### 2.13 单文件上传步骤

- ① 导入fileupload和io坐标
- ② 配置文件上传解析器
- ③ 编写文件上传代码

## ■ 2. SpringMVC 获得请求数据

### 2.14 单文件上传实现

① 导入fileupload和io坐标

```
<dependency>
  <groupId>commons-fileupload</groupId>
  <artifactId>commons-fileupload</artifactId>
  <version>1.2.2</version>
</dependency>
<dependency>
  <groupId>commons-io</groupId>
  <artifactId>commons-io</artifactId>
  <version>2.4</version>
</dependency>
```

## ■ 2. SpringMVC 获得请求数据

### 2.14 单文件上传实现

#### ② 配置文件上传解析器

```
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!--上传文件总大小-->
    <property name="maxUploadSize" value="5242800"/>
    <!--上传单个文件的大小-->
    <property name="maxUploadSizePerFile" value="5242800"/>
    <!--上传文件的编码类型-->
    <property name="defaultEncoding" value="UTF-8"/>
</bean>
```

## ■ 2. SpringMVC 获得请求数据

### 2.14 单文件上传实现

#### ③ 编写文件上传代码

```
@RequestMapping("/quick20")
@ResponseBody
public void quickMethod20(String name, MultipartFile uploadFile) throws
IOException {
    // 获得文件名称
    String originalFilename = uploadFile.getOriginalFilename();
    // 保存文件
    uploadFile.transferTo(new File("C:\\\\upload\\"+originalFilename));
}
```

## ■ 2. SpringMVC 获得请求数据

### 2.15 多文件上传实现

多文件上传，只需要将页面修改为多个文件上传项，将方法参数MultipartFile类型修改为MultipartFile[]即可

```
<h1>多文件上传测试</h1>
<form action="${pageContext.request.contextPath}/quick21" method="post"
enctype="multipart/form-data">
    名称: <input type="text" name="name"><br>
    文件1: <input type="file" name="uploadFiles"><br>
    文件2: <input type="file" name="uploadFiles"><br>
    文件3: <input type="file" name="uploadFiles"><br>
    <input type="submit" value="提交"><br>
</form>
```

## ■ 2. SpringMVC 获得请求数据

### 2.15 多文件上传实现

```
@RequestMapping("/quick21")
@ResponseBody
public void quickMethod21(String name,MultipartFile[] uploadFiles) throws
IOException {
    for (MultipartFile uploadFile : uploadFiles){
        String originalFilename = uploadFile.getOriginalFilename();
        uploadFile.transferTo(new File("C:\\upload\\"+originalFilename));
    }
}
```

## ■ 2. SpringMVC 获得请求数据

### 2.14 知识要点

#### MVC实现数据请求方式

- 基本类型参数
- POJO类型参数
- 数组类型参数
- 集合类型参数

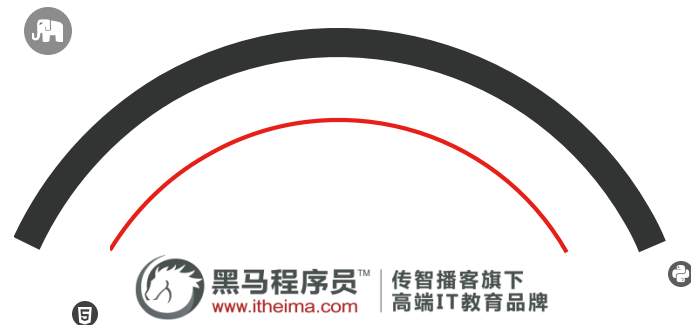
#### MVC获取数据细节

- 中文乱码问题
- @RequestParam 和 @PathVariable
- 自定义类型转换器
- 获得Servlet相关API
- @RequestHeader 和 @CookieValue
- 文件上传

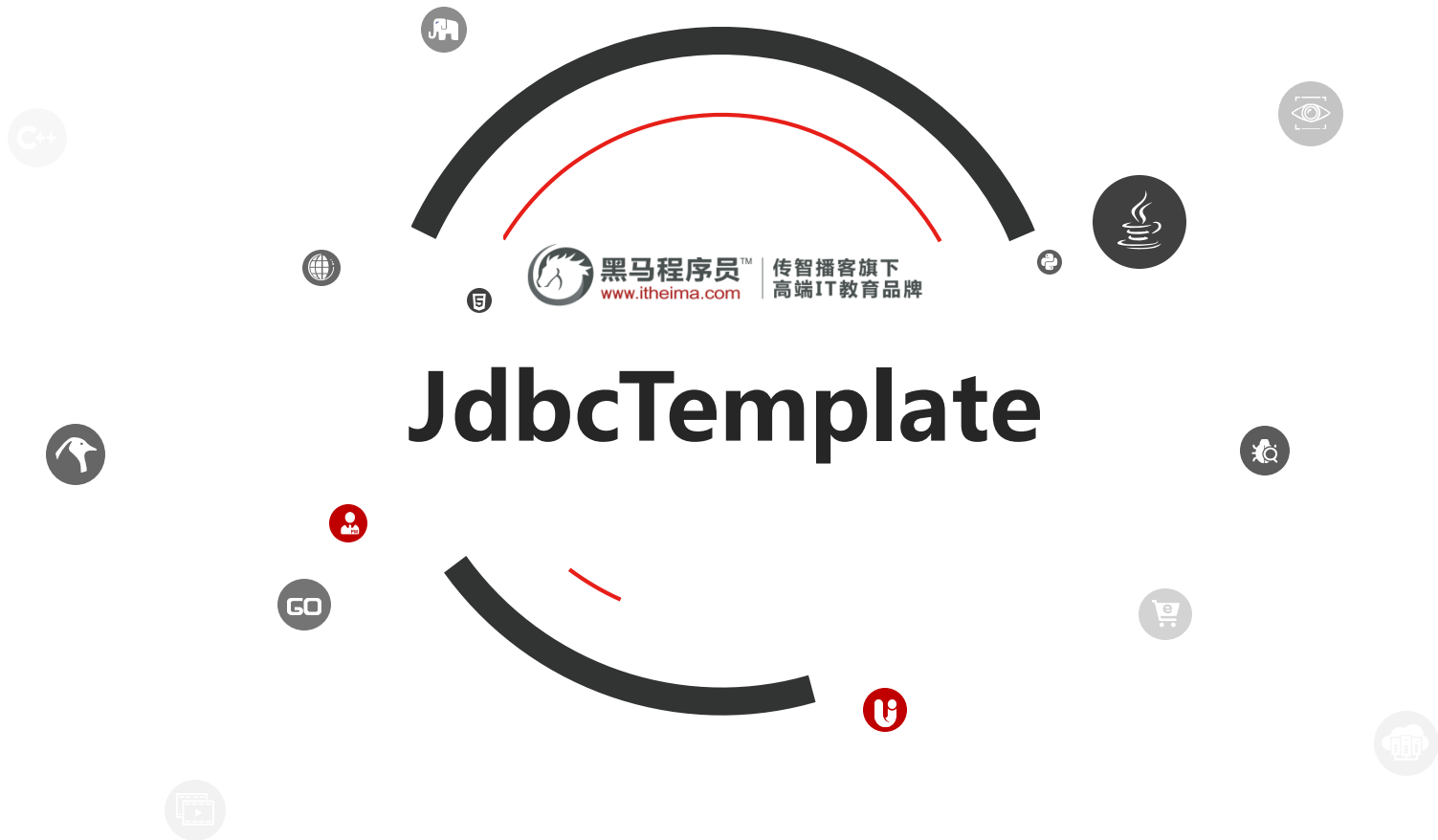


传智播客旗下高端IT教育品牌





# JdbcTemplate



# 目录 Contents

## ◆ Spring JdbcTemplate基本使用

# ■ 1. Spring JdbcTemplate基本使用

## 1.1 JdbcTemplate概述

它是spring框架中提供的一个对象，是对原始繁琐的Jdbc API对象的简单封装。spring框架为我们提供了很多的操作模板类。例如：操作关系型数据的JdbcTemplate和HibernateTemplate，操作nosql数据库的RedisTemplate，操作消息队列的JmsTemplate等等。

## 1.2 JdbcTemplate开发步骤

- ① 导入spring-jdbc和spring-tx坐标
- ② 创建数据库表和实体
- ③ 创建JdbcTemplate对象
- ④ 执行数据库操作

# 1. Spring JdbcTemplate基本使用

## 1.3 JdbcTemplate快速入门

### ① 导入坐标

```
<!--导入spring的jdbc坐标-->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>5.0.5.RELEASE</version>
</dependency>
<!--导入spring的tx坐标-->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-tx</artifactId>
  <version>5.0.5.RELEASE</version>
</dependency>
```

# 1. Spring JdbcTemplate基本使用

## 1.3 JdbcTemplate快速入门

② 创建accout表和Account实体

栏位	索引	外键	触发器	选项	注释	SQL 预览
名	类型		长度	小数点	允许空值 (	
▶ name	varchar		50	0	<input type="checkbox"/>	 1
money	double		0	0	<input checked="" type="checkbox"/>	

```
public class Account {  
    private String name;  
    private double money;  
    //省略get和set方法  
}
```

# 1. Spring JdbcTemplate基本使用

## 1.3 JdbcTemplate快速入门

- ③ 创建JdbcTemplate对象
- ④ 执行数据库操作

//1、创建数据源对象

```
ComboPooledDataSource dataSource = new ComboPooledDataSource();  
dataSource.setDriverClass("com.mysql.jdbc.Driver");  
dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/test");  
dataSource.setUser("root");  
dataSource.setPassword("root");
```

//2、创建JdbcTemplate对象

```
JdbcTemplate jdbcTemplate = new JdbcTemplate();
```

//3、设置数据源给JdbcTemplate

```
jdbcTemplate.setDataSource(dataSource);
```

//4、执行操作

```
jdbcTemplate.update("insert into account values(?,?)","tom",5000);
```

# 1. Spring JdbcTemplate基本使用

## 1.4 Spring产生JdbcTemplate对象

我们可以将JdbcTemplate的创建权交给Spring，将数据源DataSource的创建权也交给Spring，在Spring容器内部将数据源DataSource注入到JdbcTemplate模版对象中，配置如下：

```
<!--数据源DataSource-->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="driverClass" value="com.mysql.jdbc.Driver"></property>
    <property name="jdbcUrl" value="jdbc:mysql:///test"></property>
    <property name="user" value="root"></property>
    <property name="password" value="root"></property>
</bean>
<!--JdbcTemplate-->
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"></property>
</bean>
```

# ■ 1. Spring JdbcTemplate基本使用

## 1.4 Spring产生JdbcTemplate对象

从容器中获得JdbcTemplate进行添加操作

```
@Test
public void testSpringJdbcTemplate() throws PropertyVetoException {
    ApplicationContext applicationContext = new
    ClassPathXmlApplicationContext("applicationContext.xml");
    JdbcTemplate jdbcTemplate = applicationContext.getBean(JdbcTemplate.class);
    jdbcTemplate.update("insert into account values(?,?)", "lucy", 5000);
}
```



# 1. Spring JdbcTemplate基本使用

## 1.5 JdbcTemplate的常用操作

修改操作

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class JdbcTemplateCRUDTest {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Test
    //测试修改操作
    public void testUpdate() {
        jdbcTemplate.update("update account set money=? where
name=?", 1000, "tom");
    }
}
```

# 1. Spring JdbcTemplate基本使用

## 1.5 JdbcTemplate的常用操作

删除和查询全部操作

```
@Test
public void testDelete() {
    jdbcTemplate.update("delete from account where name=?", "tom");
}

@Test
public void testQueryAll() {
    List<Account> accounts = jdbcTemplate.query("select * from account", new
    BeanPropertyRowMapper<Account>(Account.class));
    for (Account account : accounts) {
        System.out.println(account.getName());
    }
}
```

# 1. Spring JdbcTemplate基本使用

## 1.5 JdbcTemplate的常用操作

查询单个数据操作

```
@Test
//测试查询单个对象操作
public void testQueryOne() {
    Account account = jdbcTemplate.queryForObject("select * from account where
name=?", new BeanPropertyRowMapper<Account>(Account.class), "tom");
    System.out.println(account.getName());
}

@Test
//测试查询单个简单数据操作(聚合查询)
public void testQueryCount() {
    Long aLong = jdbcTemplate.queryForObject("select count(*) from account",
Long.class);
    System.out.println(aLong);
}
```

# 1. Spring JdbcTemplate基本使用

## 1.6 知识要点

- ① 导入spring-jdbc和spring-tx坐标
- ② 创建数据库表和实体
- ③ 创建JdbcTemplate对象

```
JdbcTemplate jdbcTemplate = new JdbcTemplate();  
  
jdbcTemplate.setDataSource(dataSource);
```

- ④ 执行数据库操作

更新操作:

```
jdbcTemplate.update (sql,params)
```

查询操作:

```
jdbcTemplate.query (sql,Mapper,params)  
  
jdbcTemplate.queryForObject (sql,Mapper,params)
```



传智播客旗下高端IT教育品牌



黑马程序员™  
[www.itheima.com](http://www.itheima.com)

传智播客旗下  
高端IT教育品牌

# SpringMVC拦截器

# ■ 1. SpringMVC拦截器

## 1.1 拦截器 (interceptor) 的作用

Spring MVC 的**拦截器**类似于 Servlet 开发中的过滤器 Filter，用于对处理器进行**预处理**和**后处理**。

将拦截器按一定的顺序联结成一条链，这条链称为**拦截器链 (Interceptor Chain)**。在访问被拦截的方法或字段时，拦截器链中的拦截器就会按其之前定义的顺序被调用。拦截器也是AOP思想的具体实现。

# 1. SpringMVC拦截器

## 1.2 拦截器和过滤器区别

区别	过滤器 (Filter)	拦截器 (Interceptor)
使用范围	是 servlet 规范中的一部分，任何 Java Web 工程都可以使用	是 SpringMVC 框架自己的，只有使用了 SpringMVC 框架的工程才能用
拦截范围	在 url-pattern 中配置了/*之后，可以对所有要访问的资源拦截	在<mvc:mapping path= "" />中配置了/*之后，也可以多所有资源进行拦截，但是可以通过<mvc:exclude-mapping path= "" />标签排除不需要拦截的资源



# 1. SpringMVC拦截器

## 1.3 拦截器是快速入门

自定义拦截器很简单，只有如下三步：

- ① 创建拦截器类实现HandlerInterceptor接口
- ② 配置拦截器
- ③ 测试拦截器的拦截效果

# 1. SpringMVC拦截器

## 1.3 拦截器是快速入门

### ① 创建拦截器类实现HandlerInterceptor接口

```
public class MyHandlerInterceptor1 implements HandlerInterceptor {  
    public boolean preHandle(HttpServletRequest request, HttpServletResponse  
        response, Object handler) {  
        System.out.println("preHandle running...");  
        return true;  
    }  
  
    public void postHandle(HttpServletRequest request, HttpServletResponse  
        response, Object handler, ModelAndView modelAndView) {  
        System.out.println("postHandle running...");  
    }  
  
    public void afterCompletion(HttpServletRequest request, HttpServletResponse  
        response, Object handler, Exception ex) {  
        System.out.println("afterCompletion running...");  
    }  
}
```

# 1. SpringMVC拦截器

## 1.3 拦截器是快速入门

### ② 配置拦截器

```
<!--配置拦截器-->
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/**" />
        <bean class="com.ithema.interceptor.MyHandlerInterceptor1" />
    </mvc:interceptor>
</mvc:interceptors>
```

# 1. SpringMVC拦截器

## 1.3 拦截器是快速入门

### ③ 测试拦截器的拦截效果（编写目标方法）

```
@RequestMapping("/quick23")
@ResponseBody
public ModelAndView quickMethod23() throws IOException, ParseException {
    System.out.println("目标方法执行....");
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.addObject("name", "itcast");
    modelAndView.setViewName("index");
    return modelAndView;
}
```

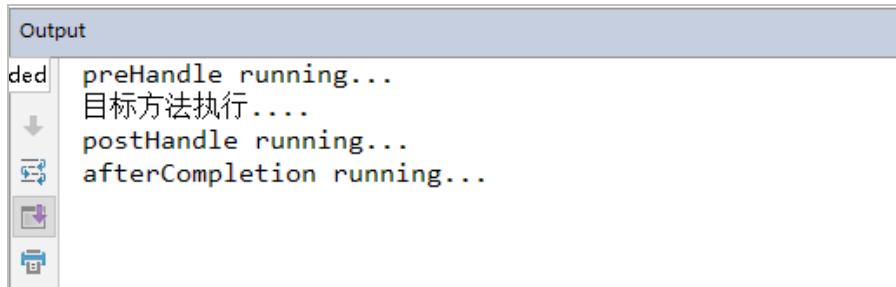
# 1. SpringMVC拦截器

## 1.3 拦截器是快速入门

③ 测试拦截器的拦截效果（访问网址）

```
http://localhost:8080/itheima_springmvc1/quick23
```

控制台打印结果



The screenshot shows an IDE's Output window with the following text:

```
Output
ded preHandle running...
    目标方法执行....
    postHandle running...
    afterCompletion running...
```

# 1. SpringMVC拦截器

## 1.4 多拦截器操作

同上，在编写一个MyHandlerInterceptor2操作，测试执行顺序

```
↑ preHandle running...  
↓ preHandle running222...  
目标方法执行....  
postHandle running222...  
postHandle running...  
afterCompletion running222...  
afterCompletion running...
```

# 1. SpringMVC拦截器

## 1.5 拦截器方法说明

方法名	说明
preHandle()	方法将在请求处理之前进行调用，该方法的返回值是布尔值Boolean类型的，当它返回为false 时，表示请求结束，后续的Interceptor 和Controller 都不会再执行；当返回值为true 时就会继续调用下一个Interceptor 的preHandle 方法
postHandle()	该方法是在当前请求进行处理之后被调用，前提是preHandle 方法的返回值为true 时才能被调用，且它会在DispatcherServlet 进行视图返回渲染之前被调用，所以我们可以在这个方法中对Controller 处理之后的ModelAndView 对象进行操作
afterCompletion()	该方法将在整个请求结束之后，也就是在DispatcherServlet 渲染了对应的视图之后执行，前提是preHandle 方法的返回值为true 时才能被调用

# ■ 1. SpringMVC拦截器

## 1.6 知识要点

自定义拦截器步骤

- ① 创建拦截器类实现HandlerInterceptor接口
- ② 配置拦截器
- ③ 测试拦截器的拦截效果



# 1. SpringMVC拦截器

## 1.7 案例-用户登录权限控制

需求：用户没有登录的情况下，不能对后台菜单进行访问操作，点击菜单跳转到登录页面，只有用户登录成功后才能进行后台功能的操作

The diagram illustrates the user login and role management process in the ITCAST system. It consists of three main components: the login page, the system menu, and the role management table.

**ITCAST后台管理系统 (ITCAST Backend Management System)**

The login page shows a "登录系统" (Login System) form. The username field contains "zhangsan" and the password field is empty. The "登录" (Login) button is highlighted. Below the login form, there are links for "记住下次自动登录" (Remember me) and "忘记密码" (Forgot password).

**数据后台管理 (Data Backend Management)**

The system menu shows the user is logged in as "zhangsan" (在线). The menu items are:

- 菜单 (Menu)
- 首页 (Home)
- 系统管理 (System Management) - expanded
  - 用户管理 (User Management)
  - 角色管理 (Role Management) - highlighted with a red box
  - 访问日志 (Access Log)
- 基础数据 (Basic Data)

**列表 (List)**

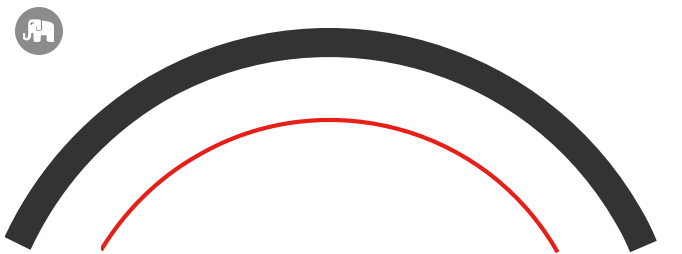
The role management table displays a list of roles. The table has columns for ID, 角色名称 (Role Name), and 角色描述 (Role Description). The roles are:

ID	角色名称	角色描述
1	院长	负责全面工作
2	研究员	课程研发工作
3	讲师	授课工作
4	助教	协助解决学生的问题
5	班主任	负责学生的生活
7	就业指导	负责学生的就业工作

Red arrows indicate the flow of the process: from the login page to the system menu, and from the role management table back to the system menu.

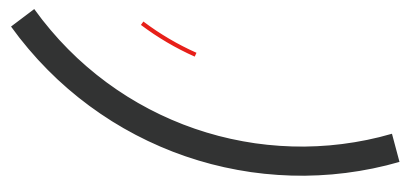


传智播客旗下高端IT教育品牌



黑马程序员™  
[www.itheima.com](http://www.itheima.com)

传智播客旗下  
高端IT教育品牌



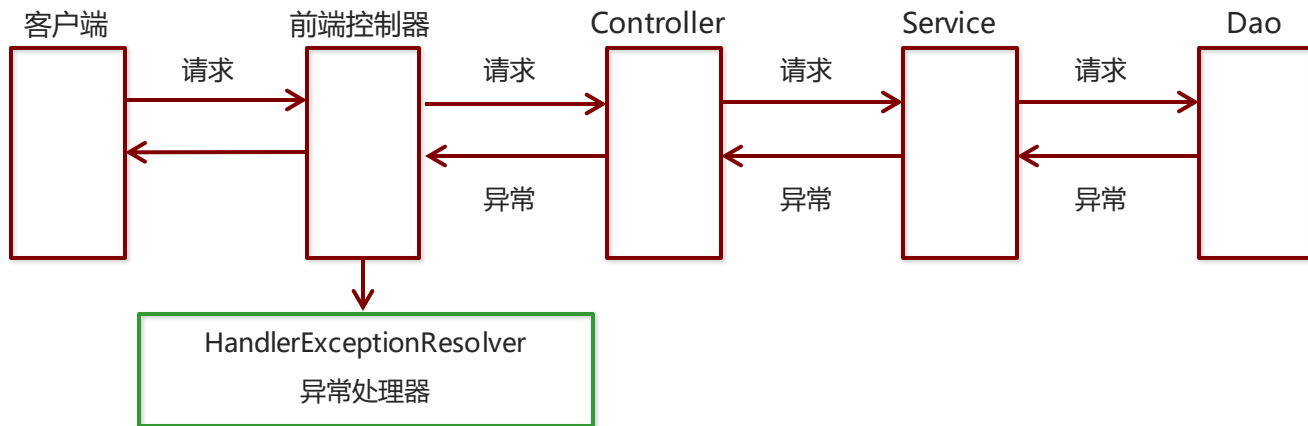
# SpringMVC异常处理机制

# 1. SpringMVC异常处理

## 1.1 异常处理的思路

系统中异常包括两类：**预期异常**和**运行时异常RuntimeException**，前者通过捕获异常从而获取异常信息，后者主要通过规范代码开发、测试等手段减少运行时异常的发生。

系统的**Dao**、**Service**、**Controller**出现都通过throws Exception向上抛出，最后由SpringMVC前端控制器交由异常处理器进行异常处理，如下图：



# 1. SpringMVC异常处理

## 1.2 异常处理两种方式

- 使用Spring MVC提供的简单异常处理器SimpleMappingExceptionHandler
- 实现Spring的异常处理接口HandlerExceptionHandler 自定义自己的异常处理器

# 1. SpringMVC异常处理

## 1.3 简单异常处理器SimpleMappingExceptionHandler

SpringMVC已经定义好了该类型转换器，在使用时可以根据项目情况进行相应异常与视图的映射配置

```
<!--配置简单映射异常处理器-->
```

```
<bean
```

```
class="org.springframework.web.servlet.handler.SimpleMappingExceptionHandler">
```

```
  <property name="defaultErrorView" value="error"/> 默认错误视图
```

```
  <property name="exceptionMappings">
```

```
    <map>
```

异常类型

错误视图

```
      <entry key="com.itheima.exception.MyException" value="error"/>
```

```
      <entry key="java.lang.ClassCastException" value="error"/>
```

```
    </map>
```

```
  </property>
```

```
</bean>
```

# 1. SpringMVC异常处理

## 1.4 自定义异常处理步骤

- ① 创建异常处理器类实现HandlerExceptionResolver
- ② 配置异常处理器
- ③ 编写异常页面
- ④ 测试异常跳转

# 1. SpringMVC异常处理

## 1.4 自定义异常处理步骤

### ① 创建异常处理器类实现HandlerExceptionResolver

```
public class MyExceptionHandler implements HandlerExceptionResolver {  
    @Override  
    public ModelAndView resolveException(HttpServletRequest request,  
        HttpServletResponse response, Object handler, Exception ex) {  
        //处理异常的代码实现  
        //创建ModelAndView对象  
        ModelAndView modelAndView = new ModelAndView();  
        modelAndView.setViewName("exceptionPage");  
        return modelAndView;  
    }  
}
```



# 1. SpringMVC异常处理

## 1.4 自定义异常处理步骤

### ② 配置异常处理器

```
<bean id="exceptionResolver"
      class="com.itheima.exception.MyExceptionHandler" />
```

### ③ 编写异常页面

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    这是一个最终异常的显示页面
</body>
</html>
```

# 1. SpringMVC异常处理

## 1.4 自定义异常处理步骤

### ④ 测试异常跳转

```
@RequestMapping("/quick22")
@ResponseBody
public void quickMethod22() throws IOException, ParseException {
    SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd");
    simpleDateFormat.parse("abcde");
}
```

# 1. SpringMVC异常处理

## 1.5 知识要点

### 异常处理方式

- 配置简单异常处理器SimpleMappingExceptionHandler
- 自定义异常处理器

### 自定义异常处理步骤

- ① 创建异常处理器类实现HandlerExceptionHandler
- ② 配置异常处理器
- ③ 编写异常页面
- ④ 测试异常跳转

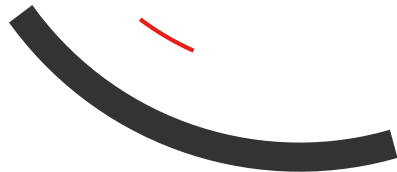


传智播客旗下高端IT教育品牌



黑马程序员™  
[www.itheima.com](http://www.itheima.com)

传智播客旗下  
高端IT教育品牌



# 面向切面编程AOP

# 目录Contents

- ◆ Spring 的 AOP 简介
- ◆ 基于 XML 的 AOP 开发
- ◆ 基于注解的 AOP 开发

# ■ 1. Spring 的 AOP 简介

## 1.1 什么是 AOP

**AOP** 为 **A**spect **O**riented **P**rogramming 的缩写，意思为**面向切面编程**，是通过预编译方式和运行期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，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

# ■ 1. Spring 的 AOP 简介

## 1.2 AOP 的作用及其优势

- 作用：在程序运行期间，在不修改源码的情况下对方法进行功能增强
- 优势：减少重复代码，提高开发效率，并且便于维护



# ■ 1. Spring 的 AOP 简介

## 1.3 AOP 的底层实现

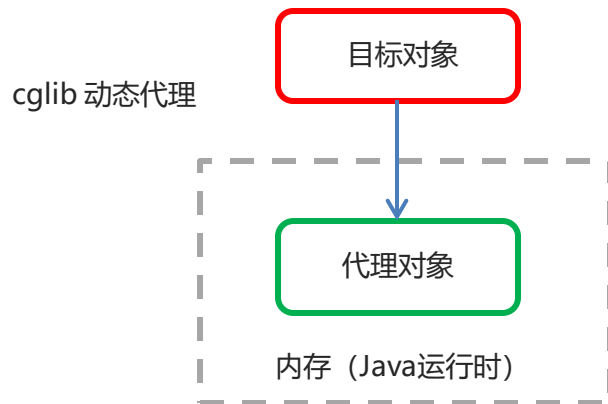
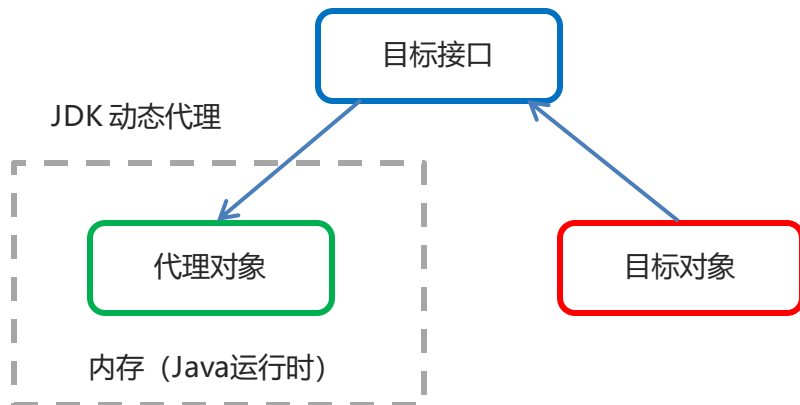
实际上，AOP 的底层是通过 Spring 提供的动态代理技术实现的。在运行期间，Spring通过动态代理技术动态的生成代理对象，代理对象方法执行时进行增强功能的介入，在去调用目标对象的方法，从而完成功能的增强。

# 1. Spring 的 AOP 简介

## 1.4 AOP 的动态代理技术

常用的动态代理技术

- JDK 代理：基于接口的动态代理技术
- cglib 代理：基于父类的动态代理技术



# 1. Spring 的 AOP 简介

## 1.5 JDK 的动态代理

### ① 目标类接口

```
public interface TargetInterface {  
    public void method();  
}
```

### ② 目标类

```
public class Target implements TargetInterface {  
    @Override  
    public void method() {  
        System.out.println("Target running...");  
    }  
}
```

# 1. Spring 的 AOP 简介

## 1.5 JDK 的动态代理

### ③ 动态代理代码

```
Target target = new Target(); //创建目标对象
//创建代理对象

TargetInterface proxy = (TargetInterface) Proxy.newProxyInstance(target.getClass()
    .getClassLoader(), target.getClass().getInterfaces(), new InvocationHandler() {

    @Override

    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {

        System.out.println("前置增强代码...");

        Object invoke = method.invoke(target, args);

        System.out.println("后置增强代码...");

        return invoke;

    }

});
```

# 1. Spring 的 AOP 简介

## 1.5 JDK 的动态代理

### ④ 调用代理对象的方法测试

```
// 测试,当调用接口的任何方法时,代理对象的代码都无序修改  
proxy.method();
```

```
"C:\Program Files\Java\jdk1.8.0_162\bin\java" ...
```

```
前置增强代码...
```

```
Target running....
```

```
后置增强代码...
```

```
Process finished with exit code 0
```

# 1. Spring 的 AOP 简介

## 1.6 cglib 的动态代理

### ① 目标类

```
public class Target {  
    public void method() {  
        System.out.println("Target running...");  
    }  
}
```

# 1. Spring 的 AOP 简介

## 1.6 cglib 的动态代理

### ② 动态代理代码

```
Target target = new Target(); //创建目标对象

Enhancer enhancer = new Enhancer(); //创建增强器
enhancer.setSuperclass(Target.class); //设置父类
enhancer.setCallback(new MethodInterceptor() { //设置回调
    @Override
    public Object intercept(Object o, Method method, Object[] objects,
        MethodProxy methodProxy) throws Throwable {
        System.out.println("前置代码增强....");
        Object invoke = method.invoke(target, objects);
        System.out.println("后置代码增强....");
        return invoke;
    }
});

Target proxy = (Target) enhancer.create(); //创建代理对象
```

# ■ 1. Spring 的 AOP 简介

## 1.6 cglib 的动态代理

### ③ 调用代理对象的方法测试

```
//测试,当调用接口的任何方法时,代理对象的代码都无序修改  
proxy.method();
```

```
"C:\Program Files\Java\jdk1.8.0_162\bin\java" ...  
前置增强代码...  
Target running....  
后置增强代码...  
  
Process finished with exit code 0
```



# 1. Spring 的 AOP 简介

## 1.7 AOP 相关概念

Spring 的 AOP 实现底层就是对上面的动态代理的代码进行了封装，封装后我们只需要对需要关注的部分进行代码编写，并通过配置的方式完成指定目标的方法增强。

在正式讲解 AOP 的操作之前，我们必须理解 AOP 的相关术语，常用的术语如下：

- Target（目标对象）：代理的目标对象
- Proxy（代理）：一个类被 AOP 织入增强后，就产生一个结果代理类
- Joinpoint（连接点）：所谓连接点是指那些被拦截到的点。在spring中,这些点指的是方法，因为spring只支持方法类型的连接点
- Pointcut（切入点）：所谓切入点是指我们要对哪些 Joinpoint 进行拦截的定义
- Advice（通知/增强）：所谓通知是指拦截到 Joinpoint 之后所要做的事情就是通知
- Aspect（切面）：是切入点和通知（引介）的结合
- Weaving（织入）：是指把增强应用到目标对象来创建新的代理对象的过程。spring采用动态代理织入，而AspectJ采用编译期织入和类装载期织入

# ■ 1. Spring 的 AOP 简介

## 1.8 AOP 开发明确的事项

### 1. 需要编写的内容

- 编写核心业务代码（目标类的目标方法）
- 编写切面类，切面类中有通知(增强功能方法)
- 在配置文件中，配置织入关系，即将哪些通知与哪些连接点进行结合

### 2. AOP 技术实现的内容

Spring 框架监控切入点方法的执行。一旦监控到切入点方法被运行，使用代理机制，动态创建目标对象的代理对象，根据通知类别，在代理对象的对应位置，将通知对应的功能织入，完成完整的代码逻辑运行。

### 3. AOP 底层使用哪种代理方式

在 spring 中，框架会根据目标类是否实现了接口来决定采用哪种动态代理的方式。

# ■ 1. Spring 的 AOP 简介

## 1.9 知识要点

- aop: 面向切面编程
- aop底层实现: 基于JDK的动态代理 和 基于Cglib的动态代理
- aop的重点概念:
  - Pointcut (切入点) : 被增强的方法
  - Advice (通知/ 增强) : 封装增强业务逻辑的方法
  - Aspect (切面) : 切点+通知
  - Weaving (织入) : 将切点与通知结合的过程
- 开发明确事项:
  - 谁是切点 (切点表达式配置)
  - 谁是通知 (切面类中的增强方法)
  - 将切点和通知进行织入配置

# 目录 Contents

- ◆ Spring 的 AOP 简介
- ◆ 基于 XML 的 AOP 开发
- ◆ 基于注解的 AOP 开发

## ■ 2. 基于 XML 的 AOP 开发

### 2.1 快速入门

- ① 导入 AOP 相关坐标
- ② 创建目标接口和目标类（内部有切点）
- ③ 创建切面类（内部有增强方法）
- ④ 将目标类和切面类的对象创建权交给 spring
- ⑤ 在 applicationContext.xml 中配置织入关系
- ⑥ 测试代码

## ■ 2. 基于 XML 的 AOP 开发

### 2.1 快速入门

#### ① 导入 AOP 相关坐标

```
<!--导入spring的context坐标, context依赖aop-->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>5.0.5.RELEASE</version>
</dependency>
<!-- aspectj的织入 -->
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.8.13</version>
</dependency>
```

## 2. 基于 XML 的 AOP 开发

### 2.1 快速入门

② 创建目标接口和目标类（内部有切点）

```
public interface TargetInterface {  
    public void method();  
}
```

```
public class Target implements TargetInterface {  
    @Override  
    public void method() {  
        System.out.println("Target running...");  
    }  
}
```

## ■ 2. 基于 XML 的 AOP 开发

### 2.1 快速入门

③ 创建切面类（内部有增强方法）

```
public class MyAspect {  
    //前置增强方法  
    public void before() {  
        System.out.println("前置代码增强.....");  
    }  
}
```



## 2. 基于 XML 的 AOP 开发

### 2.1 快速入门

④ 将目标类和切面类的对象创建权交给 spring

```
<!--配置目标类-->  
<bean id="target" class="com.itheima.aop.Target"></bean>  
<!--配置切面类-->  
<bean id="myAspect" class="com.itheima.aop.MyAspect"></bean>
```

## ■ 2. 基于 XML 的 AOP 开发

### 2.1 快速入门

⑤ 在 applicationContext.xml 中配置织入关系

导入aop命名空间

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xsi:schemaLocation="
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context.xsd
            http://www.springframework.org/schema/aop
            http://www.springframework.org/schema/aop/spring-aop.xsd
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd">
```

## ■ 2. 基于 XML 的 AOP 开发

### 2.1 快速入门

⑤ 在 applicationContext.xml 中配置织入关系

配置切点表达式和前置增强的织入关系

```
<aop:config>
    <!-- 引用myAspect的Bean为切面对象-->
    <aop:aspect ref="myAspect">
        <!-- 配置Target的method方法执行时要进行myAspect的before方法前置增强-->
        <aop:before method="before" pointcut="execution(public void
com.itheima.aop.Target.method())"></aop:before>
    </aop:aspect>
</aop:config>
```

## ■ 2. 基于 XML 的 AOP 开发

### 2.1 快速入门

#### ⑥ 测试代码

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class AopTest {
    @Autowired
    private TargetInterface target;

    @Test
    public void test1(){
        target.method();
    }
}
```

## ■ 2. 基于 XML 的 AOP 开发

### 2.1 快速入门

#### ⑦ 测试结果

```
信息: Loading XML bean definitions from class path resource [applicationContext.xml]  
十月 23, 2018 5:55:49 下午 org.springframework.context.support.AbstractApplicationCor  
信息: Refreshing org.springframework.context.support.GenericApplicationContext@6f7fd  
前置代码增强.....  
Target running....  
  
Process finished with exit code 0
```

## ■ 2. 基于 XML 的 AOP 开发

### 2.2 XML 配置 AOP 详解

#### 1. 切点表达式的写法

表达式语法：

```
execution([修饰符] 返回值类型 包名.类名.方法名(参数))
```

- 访问修饰符可以省略
- 返回值类型、包名、类名、方法名可以使用星号\* 代表任意
- 包名与类名之间一个点. 代表当前包下的类，两个点.. 表示当前包及其子包下的类
- 参数列表可以使用两个点.. 表示任意个数，任意类型的参数列表

例如：

```
execution(public void com.itheima.aop.Target.method())  
execution(void com.itheima.aop.Target.*(..))  
execution(* com.itheima.aop.*.*(..))  
execution(* com.itheima.aop..*.*(..))  
execution(* *.*.*.*(..))
```

## 2. 基于 XML 的 AOP 开发

### 2.2 XML 配置 AOP 详解

#### 2. 通知的类型

通知的配置语法：

```
<aop:通知类型 method= “切面类中方法名” pointcut= “切点表达式”> </aop:通知类型>
```

名称	标签	说明
前置通知	<aop:before>	用于配置前置通知。指定增强的方法在切入点方法之前执行
后置通知	<aop:after-returning>	用于配置后置通知。指定增强的方法在切入点方法之后执行
环绕通知	<aop:around>	用于配置环绕通知。指定增强的方法在切入点方法之前和之后都执行
异常抛出通知	<aop:throwing>	用于配置异常抛出通知。指定增强的方法在出现异常时执行
最终通知	<aop:after>	用于配置最终通知。无论增强方式执行是否有异常都会执行

## 2. 基于 XML 的 AOP 开发

### 2.2 XML 配置 AOP 详解

#### 3. 切点表达式的抽取

当多个增强的切点表达式相同时，可以将切点表达式进行抽取，在增强中使用 pointcut-ref 属性代替 pointcut 属性来引用抽取后的切点表达式。

```
<aop:config>
    <!-- 引用myAspect的Bean为切面对象-->
    <aop:aspect ref="myAspect">
        <aop:pointcut id="myPointcut" expression="execution(* com.itheima.aop.*.*(..))"/>
        <aop:before method="before" pointcut-ref="myPointcut"></aop:before>
    </aop:aspect>
</aop:config>
```



## ■ 2. 基于 XML 的 AOP 开发

### 2.3 知识要点

- aop织入的配置

```
<aop:config>
    <aop:aspect ref="切面类">
        <aop:before method="通知方法名称" pointcut="切点表达式"></aop:before>
    </aop:aspect>
</aop:config>
```

- 通知的类型：前置通知、后置通知、环绕通知、异常抛出通知、最终通知
- 切点表达式的写法：

```
execution([修饰符] 返回值类型 包名.类名.方法名(参数))
```

# 目录Contents

- ◆ Spring 的 AOP 简介
- ◆ 基于 XML 的 AOP 开发
- ◆ 基于注解的 AOP 开发

## 3. 基于注解的 AOP 开发

### 3.1 快速入门

基于注解的aop开发步骤：

- ① 创建目标接口和目标类（内部有切点）
- ② 创建切面类（内部有增强方法）
- ③ 将目标类和切面类的对象创建权交给 spring
- ④ 在切面类中使用注解配置织入关系
- ⑤ 在配置文件中开启组件扫描和 AOP 的自动代理
- ⑥ 测试

## 3. 基于注解的 AOP 开发

### 3.1 快速入门

① 创建目标接口和目标类（内部有切点）

```
public interface TargetInterface {  
    public void method();  
}
```

```
public class Target implements TargetInterface {  
    @Override  
    public void method() {  
        System.out.println("Target running...");  
    }  
}
```

## 3. 基于注解的 AOP 开发

### 3.1 快速入门

② 创建切面类（内部有增强方法）

```
public class MyAspect {  
    //前置增强方法  
    public void before() {  
        System.out.println("前置代码增强.....");  
    }  
}
```

## 3. 基于注解的 AOP 开发

### 3.1 快速入门

③ 将目标类和切面类的对象创建权交给 spring

```
@Component("target")
public class Target implements TargetInterface {
    @Override
    public void method() {
        System.out.println("Target running...");
    }
}

@Component("myAspect")
public class MyAspect {
    public void before() {
        System.out.println("前置代码增强.....");
    }
}
```

## 3. 基于注解的 AOP 开发

### 3.1 快速入门

④ 在切面类中使用注解配置织入关系

```
@Component("myAspect")
@Aspect
public class MyAspect {
    @Before("execution(* com.itheima.aop.*.*(..))")
    public void before() {
        System.out.println("前置代码增强.....");
    }
}
```

## 3. 基于注解的 AOP 开发

### 3.1 快速入门

⑤ 在配置文件中开启组件扫描和 AOP 的自动代理

```
<!--组件扫描-->  
<context:component-scan base-package="com.ithema.aop"/>  
  
<!--aop的自动代理-->  
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```



## 3. 基于注解的 AOP 开发

### 3.1 快速入门

#### ⑥ 测试代码

```
@RunWith(SpringJUnit4ClassRunner.class)
@Configuration("classpath:applicationContext.xml")
public class AopTest {
    @Autowired
    private TargetInterface target;

    @Test
    public void test1(){
        target.method();
    }
}
```

## 3. 基于注解的 AOP 开发

### 3.1 快速入门

#### ⑦ 测试结果

```
信息: Closing org.springframework.context.support.GenericApplicationContext@6f7fd0e6  
前置代码增强.....  
Target running....
```

```
Process finished with exit code 0
```

## 3. 基于注解的 AOP 开发

### 3.2 注解配置 AOP 详解

#### 1. 注解通知的类型

通知的配置语法: @通知注解("切点表达式")

名称	注解	说明
前置通知	@Before	用于配置前置通知。指定增强的方法在切入点方法之前执行
后置通知	@AfterReturning	用于配置后置通知。指定增强的方法在切入点方法之后执行
环绕通知	@Around	用于配置环绕通知。指定增强的方法在切入点方法之前和之后都执行
异常抛出通知	@AfterThrowing	用于配置异常抛出通知。指定增强的方法在出现异常时执行
最终通知	@After	用于配置最终通知。无论增强方式执行是否有异常都会执行

## 3. 基于注解的 AOP 开发

### 3.2 注解配置 AOP 详解

#### 2. 切点表达式的抽取

同 xml 配置 aop 一样，我们可以将切点表达式抽取。抽取方式是在切面内定义方法，在该方法上使用@Pointcut 注解定义切点表达式，然后在在增强注解中进行引用。具体如下：

```
@Component("myAspect")
@Aspect
public class MyAspect {
    @Before("MyAspect.myPoint()")
    public void before() {
        System.out.println("前置代码增强.....");
    }
    @Pointcut("execution(* com.itheima.aop.*.*(..))")
    public void myPoint() {}
}
```

## 3. 基于注解的 AOP 开发

### 3.3 知识要点

- 注解aop开发步骤
  - ① 使用@Aspect标注切面类
  - ② 使用@通知注解标注通知方法
  - ③ 在配置文件中配置aop自动代理<aop:aspectj-autoproxy/>
- 通知注解类型

前置通知	@Before	用于配置前置通知。指定增强的方法在切入点方法之前执行
后置通知	@AfterReturning	用于配置后置通知。指定增强的方法在切入点方法之后执行
环绕通知	@Around	用于配置环绕通知。指定增强的方法在切入点方法之前和之后都执行
异常抛出通知	@AfterThrowing	用于配置异常抛出通知。指定增强的方法在出现异常时执行
最终通知	@After	用于配置最终通知。无论增强方式执行是否有异常都会执行



传智播客旗下高端IT教育品牌



黑马程序员™  
[www.itheima.com](http://www.itheima.com)

传智播客旗下  
高端IT教育品牌

# 声明式事务控制

# 目录 Contents

- ◆ 编程式事务控制相关对象
- ◆ 基于 XML 的声明式事务控制
- ◆ 基于注解的声明式事务控制



# 1. 程式事务控制相关对象

## 1.1 PlatformTransactionManager

PlatformTransactionManager 接口是 spring 的事务管理器，它里面提供了我们常用的操作事务的方法。

方法	说明
TransactionStatus getTransaction(TransactionDefination defination)	获取事务的状态信息
void commit(TransactionStatus status)	提交事务
void rollback(TransactionStatus status)	回滚事务

### 注意：

PlatformTransactionManager 是接口类型，不同的 Dao 层技术则有不同的实现类，例如：Dao 层技术是jdbc 或 mybatis 时：org.springframework.jdbc.datasource.DataSourceTransactionManager  
Dao 层技术是hibernate时：org.springframework.orm.hibernate5.HibernateTransactionManager

# 1. 程式事务控制相关对象

## 1.2 TransactionDefinition

TransactionDefinition 是事务的定义信息对象，里面有如下方法：

方法	说明
<code>int getIsolationLevel()</code>	获得事务的隔离级别
<code>int getPropagationBehavior()</code>	获得事务的传播行为
<code>int getTimeout()</code>	获得超时时间
<code>boolean isReadOnly()</code>	是否只读

# ■ 1. 程式事务控制相关对象

## 1.2 TransactionDefinition

### 1. 事务隔离级别

设置隔离级别，可以解决事务并发产生的问题，如脏读、不可重复读和虚读。

- ISOLATION\_DEFAULT
- ISOLATION\_READ\_UNCOMMITTED
- ISOLATION\_READ\_COMMITTED
- ISOLATION\_REPEATABLE\_READ
- ISOLATION\_SERIALIZABLE

# ■ 1. 程式事务控制相关对象

## 1.2 TransactionDefinition

### 2. 事务传播行为

- REQUIRED: 如果当前没有事务, 就新建一个事务, 如果已经存在一个事务中, 加入到这个事务中。一般的选择 (默认值)
- SUPPORTS: 支持当前事务, 如果当前没有事务, 就以非事务方式执行 (没有事务)
- MANDATORY: 使用当前的事务, 如果当前没有事务, 就抛出异常
- REQUIRES\_NEW: 新建事务, 如果当前在事务中, 把当前事务挂起。
- NOT\_SUPPORTED: 以非事务方式执行操作, 如果当前存在事务, 就把当前事务挂起
- NEVER: 以非事务方式运行, 如果当前存在事务, 抛出异常
- NESTED: 如果当前存在事务, 则在嵌套事务内执行。如果当前没有事务, 则执行 REQUIRED 类似的操作
- 超时时间: 默认值是-1, 没有超时限制。如果有, 以秒为单位进行设置
- 是否只读: 建议查询时设置为只读

# 1. 程式事务控制相关对象

## 1.3 TransactionStatus

TransactionStatus 接口提供的是事务具体的运行状态，方法介绍如下。

方法	说明
<code>boolean hasSavepoint()</code>	是否存储回滚点
<code>boolean isCompleted()</code>	事务是否完成
<code>boolean isNewTransaction()</code>	是否是新事务
<code>boolean isRollbackOnly()</code>	事务是否回滚

# ■ 1. 程式事务控制相关对象

## 1.4 知识要点

### 程式事务控制三大对象

- PlatformTransactionManager
- TransactionDefinition
- TransactionStatus

# 目录 Contents

- ◆ 编程式事务控制相关对象
- ◆ 基于 XML 的声明式事务控制
- ◆ 基于注解的声明式事务控制

## ■ 2. 基于 XML 的声明式事务控制

### 2.1 什么是声明式事务控制

Spring 的声明式事务顾名思义就是**采用声明的方式来处理事务**。这里所说的声明，就是指在配置文件中声明，用在 Spring 配置文件中声明式的处理事务来代替代码式的处理事务。

#### 声明式事务处理的作用

- 事务管理不侵入开发的组件。具体来说，业务逻辑对象就不会意识到正在事务管理之中，事实上也应该如此，因为事务管理是属于系统层面的服务，而不是业务逻辑的一部分，如果想要改变事务管理策划的话，也只需要在定义文件中重新配置即可
- 在不需要事务管理的时候，只要在设定文件上修改一下，即可移去事务管理服务，无需改变代码重新编译，这样维护起来极其方便

**注意：**Spring 声明式事务控制底层就是AOP。



## ■ 2. 基于 XML 的声明式事务控制

### 2.2 声明式事务控制的实现

声明式事务控制明确事项：

- 谁是切点？
- 谁是通知？
- 配置切面？

## ■ 2. 基于 XML 的声明式事务控制

### 2.2 声明式事务控制的实现

#### ① 引入tx命名空间

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xmlns:tx="http://www.springframework.org/schema/tx"
        xsi:schemaLocation="

        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
```

## 2. 基于 XML 的声明式事务控制

### 2.2 声明式事务控制的实现

#### ② 配置事务增强

```
<!--平台事务管理器-->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"></property>
</bean>

<!--事务增强配置-->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="*" />
    </tx:attributes>
</tx:advice>
```

## ■ 2. 基于 XML 的声明式事务控制

### 2.2 声明式事务控制的实现

#### ③ 配置事务 AOP 织入

```
<!--事务的aop增强-->
<aop:config>
    <aop:pointcut id="myPointcut" expression="execution(*
com.itheima.service.impl.*.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="myPointcut"></aop:advisor>
</aop:config>
```

## 2. 基于 XML 的声明式事务控制

### 2.2 声明式事务控制的实现

#### ④ 测试事务控制转账业务代码

```
@Override
public void transfer(String outMan, String inMan, double money) {
    accountDao.out(outMan,money);
    int i = 1/0;
    accountDao.in(inMan,money);
}
```

## 2. 基于 XML 的声明式事务控制

### 2.3 切点方法的事务参数的配置

```
<!--事务增强配置-->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="*" />
    </tx:attributes>
</tx:advice>
```

其中，<tx:method> 代表切点方法的事务参数的配置，例如：

```
<tx:method name="transfer" isolation="REPEATABLE_READ" propagation="REQUIRED" timeout="-1"
read-only="false"/>
```

- name: 切点方法名称
- isolation: 事务的隔离级别
- propagation: 事务的传播行为
- timeout: 超时时间
- read-only: 是否只读

## ■ 2. 基于 XML 的声明式事务控制

### 2.4 知识要点

#### 声明式事务控制的配置要点

- 平台事务管理器配置
- 事务通知的配置
- 事务aop织入的配置

# 目录 Contents

- ◆ 编程式事务控制相关对象
- ◆ 基于 XML 的声明式事务控制
- ◆ 基于注解的声明式事务控制



## 3. 基于注解的声明式事务控制

### 3.1 使用注解配置声明式事务控制

#### 1. 编写 AccountDao

```
@Repository("accountDao")
public class AccountDaoImpl implements AccountDao {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    public void out(String outMan, double money) {
        jdbcTemplate.update("update account set money=money-? where
name=?", money, outMan);
    }

    public void in(String inMan, double money) {
        jdbcTemplate.update("update account set money=money+? where
name=?", money, inMan);
    }
}
```

## 3. 基于注解的声明式事务控制

### 3.1 使用注解配置声明式事务控制

#### 2. 编写 AccountService

```
@Service("accountService")
@Transactional
public class AccountServiceImpl implements AccountService {
    @Autowired
    private AccountDao accountDao;

    @Transactional(isolation = Isolation.READ_COMMITTED, propagation =
    Propagation.REQUIRED)
    public void transfer(String outMan, String inMan, double money) {
        accountDao.out(outMan, money);
        int i = 1/0;
        accountDao.in(inMan, money);
    }
}
```

## 3. 基于注解的声明式事务控制

### 3.1 使用注解配置声明式事务控制

#### 3. 编写 applicationContext.xml 配置文件

```
<!--之前省略dataSource、jdbcTemplate、平台事务管理器的配置-->  
<!--组件扫描-->  
<context:component-scan base-package="com.itheima"/>  
<!--事务的注解驱动-->  
<tx:annotation-driven/>
```

## ■ 3. 基于注解的声明式事务控制

### 3.2 注解配置声明式事务控制解析

- ① 使用 @Transactional 在需要进行事务控制的类或是方法上修饰，注解可用的属性同 xml 配置方式，例如隔离级别、传播行为等。
- ② 注解使用在类上，那么该类下的所有方法都使用同一套注解参数配置。
- ③ 使用在方法上，不同的方法可以采用不同的事务参数配置。
- ④ Xml配置文件中要开启事务的注解驱动<tx:annotation-driven />

## 3. 基于注解的声明式事务控制

### 3.3 知识要点

#### 注解声明式事务控制的配置要点

- 平台事务管理器配置 (xml方式)
- 事务通知的配置 (@Transactional注解配置)
- 事务注解驱动的配置 `<tx:annotation-driven/>`



传智播客旗下高端IT教育品牌