课堂主题

springmvc应用开发

课堂目标

- 熟悉springmvc六大组件的作用及使用方式
- 掌握搭建ssm开发框架
- 掌握springmvc对于返回值的处理
- 掌握springmvc对于参数的处理
- 理解REST和RESTful，掌握springmvc对于RESTful的支持
- 掌握springmvc拦截器的应用
- 掌握springmvc中基于cors的跨域解决方案
- 掌握spring中的父子容器
- 了解springmvc的mock测试
- 了解ControllerAdvice类的开发ModelAttribute、InitBinder、ExceptionHandler
- 了解springmvc异常处理器
- 了解springmvc中的get\post请求乱码和响应乱码处理
- 了解springmvc中非注解开发方式

知识要点

- applicationContext-dao.xml（核心）
- db.properties
- log4j.properties
- 整合Service
  - applicationContext-service.xml
- 整合Controller
  - web.xml
  - springmvc.xml
- web.xml加载spring父容器
- 整合测试（编写代码）
- 需求
- 需求分析
- 持久层代码
- 业务层代码
- 表现层代码
- 部署测试

应用掌握篇

- 返回值处理
  - 不使用注解修饰
    - ModelAndView
    - void
    - String（推荐）
      - 逻辑视图名
      - redirect重定向
      - forward转发
  - 使用注解修饰
    - ResponseBody注解介绍
      - 注解介绍
      - 常用的HttpMessageConverter
    - ResponseBody示例
      - JSP示例代码
      - Controller代码
- 参数绑定处理
  - 什么是参数绑定
  - 默认支持的参数类型
  - 参数绑定使用要求
    - 简单类型
      - 直接绑定
      - 注解绑定
      - RequestParam注解
    - 绑定POJO类型
    - 绑定集合或者数组类型
      - 简单类型数组
      - POJO类型集合或者数组
  - 参数绑定示例
    - JSP代码
    - Controller代码
    - PO代码
  - 自定义日期参数绑定
    - Converter代码
    - Converter配置
  - 文件类型参数绑定
    - 加入依赖包
    - 上传页面
    - 配置Multipart解析器
    - Controller类代码
- RequestMapping注解
  - value属性

- 请求URL映射
- 窄化请求映射
- method属性
- params属性

- RESTful支持
  - HTTP介绍
    - HTTP协议概述
      - 什么是HTTP协议?
      - 什么是URL/URI?
      - 什么是WEB资源?
      - HTTP的作用是什么?
    - HTTP协议版本
    - HTTP协议组成
      - 请求协议信息
      - 响应协议信息
    - GET请求和POST请求的区别
  - 什么是RESTful
  - RESTful的特性
  - 如何设计RESTful应用程序的API
  - SpringMVC对RESTful的支持
    - RESTful的URL路径变量
    - RESTful的CRUD
    - RESTful的资源表述
    - 静态资源访问
- 拦截器应用
  - 拦截器介绍
  - 定义拦截器
  - 配置拦截器
  - 多拦截器拦截规则
  - 拦截器应用（实现登录认证）
    - 需求
    - 登录页面
    - Controller类
    - HandlerInterceptor类
    - HandlerInterceptor配置
- CORS跨域解决方案
  - 什么是跨域
  - 为什么要有同源策略
  - 如何解决跨域
  - 什么是CORS
  - 客户端跨域处理
    - 请求分类标准
    - 简单请求
    - 非简单请求
  - 服务器端跨域处理
    - CORS实现1
      - 跨域不提交Cookie
      - 跨域提交Cookie
    - CORS实现2
- 父子容器

应用了解篇

- Mock测试（模拟测试）
  - 什么是mock测试
  - 为什么使用mock测试
  - MockMVC介绍
    - MockMvc
      - MockMVCBuilder
      - MockMVCBuilders

介绍篇

基础概念介绍

BS和CS开发架构

- 一种是 `C/S` 架构，也就是客户端/服务器；
- 一种是 `B/S` **架构**，也就是**浏览器/服务器架构。**

> **说明：**
>
> 我们现在使用Java开发的大多数都是**web应用**，这些应用几乎全都是基于 B/S 架构进行开发的。那么在 B/S 架构中，应用系统标准的三层架构分为：**表现层、业务层、持久层。**这种三层架构在我们的实际开发中使用的非常多，所以我们课程中的案例也都是基于三层架构设计的。

JavaEE制定了一套规范，去进行BS结构的处理。这套规范就是**Servlet。**

应用系统三层架构

- **表现层：**
  - 也就是我们常说的**web 层。**
  - 它负责**接收客户端请求，向客户端响应结果**，通常客户端使用http 协议请求web 层，web 层需要接收 http 请求，完成 http 响应。
  - **表现层包括展示层和控制层**：控制层负责接收请求，展示层负责结果的展示。
  - 表现层依赖业务层，接收到客户端请求一般会调用业务层进行业务处理，并将处理结果响应给客户端。
  - 表现层的设计一般都**使用 MVC 模型。**（MVC 是表现层的设计模型，和其他层没有关系）
- **业务层**：
  - 也就是我们常说的 service 层。
  - 它负责业务逻辑处理，和我们开发项目的需求息息相关。web 层依赖业务层，但是业务层不依赖 web 层。
  - 业务层在业务处理时可能会依赖持久层，如果要对数据持久化需要保证事务一致性。（也就是我们说的， 事务应该放到业务层来控制）
- **持久层**：
  - 也就是我们是常说的 dao 层。
  - 负责数据持久化，包括数据层即数据库和数据访问层，数据库是对数据进行持久化的载体，数据访问层是业务层和持久层交互的接口，业务层需要通过数据访问层将数据持久化到数据库中。
  - 通俗的讲，持久层就是和数据库交互，对数据库表进行曾删改查的。

MVC设计模式

MVC 是**模型(model) - 视图(view) - 控制器(controller)**的缩写， 是一种用于设计编写 Web 应用程序表现层的模式。

MVC 设计模式的三大角色：

- **Model（模型）：**

  模型包含业务模型和数据模型，数据模型用于封装数据，业务模型用于处理业务。

- **View（视图）：**

  通常指的就是我们的 jsp 或者 html。作用一般就是展示数据的。

  通常视图是依据数据模型创建的。

- **Controller（控制器）：**

  是应用程序中处理用户交互的部分。作用一般就是处理程序逻辑的。

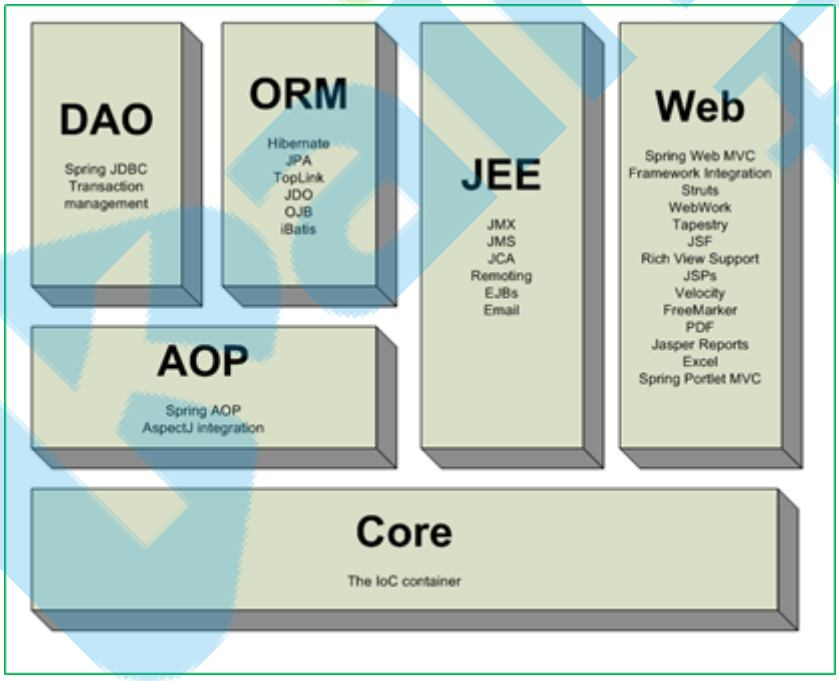
SpringMVC介绍

SpringMVC是什么

- SpringMVC 是一种基于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 里面。Spring 框架提供了构建 Web 应用程序的全功能 MVC 模块。
- 使用 Spring 可插入的 MVC 架构，从而在使用 Spring 进行 WEB 开发时，可以选择使用 Spring 的 Spring MVC 框架或集成其他 MVC 开发框架，如 Struts1(现在一般不用)，Struts2 等。
- 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。

SpringMVC与Spring的联系

`Spring MVC`全名叫`Spring Web MVC`，它是`Spring`家族`web`模块的一个重要成员。这一点我们可以从`Spring`的整体结构中看得出来：



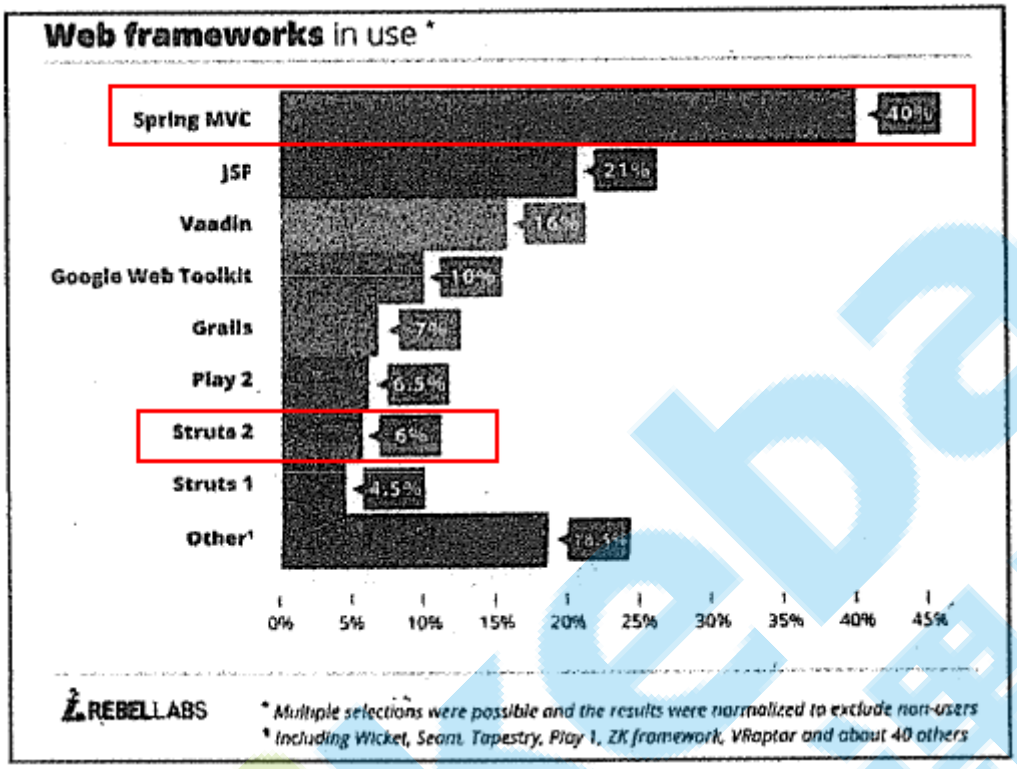
为什么学习SpringMVC

也许你要问，为什么要学习Spring MVC呢？Struts2不才是主流吗？看SSH的概念有多火？

其实很多初学者混淆了一个概念，**SSH**实际上指的是`Struts1.x+Spring+Hibernate`。这个概念已经有十几年的历史了。在Struts1.x时代，它是当之无愧的霸主，但是在新的MVC框架涌现的时代，形式已经不是这样了，Struts2.x借助了Struts1.x的好名声，让国内开发人员认为Struts2.x是霸主继任者（其实两者在技术上无任何关系），导致国内程序员大多数学习基于Struts2.x的框架，又一个貌似很火的概

念出来了**S2SH**（`Struts2+Spring+Hibernate`）整合开发。

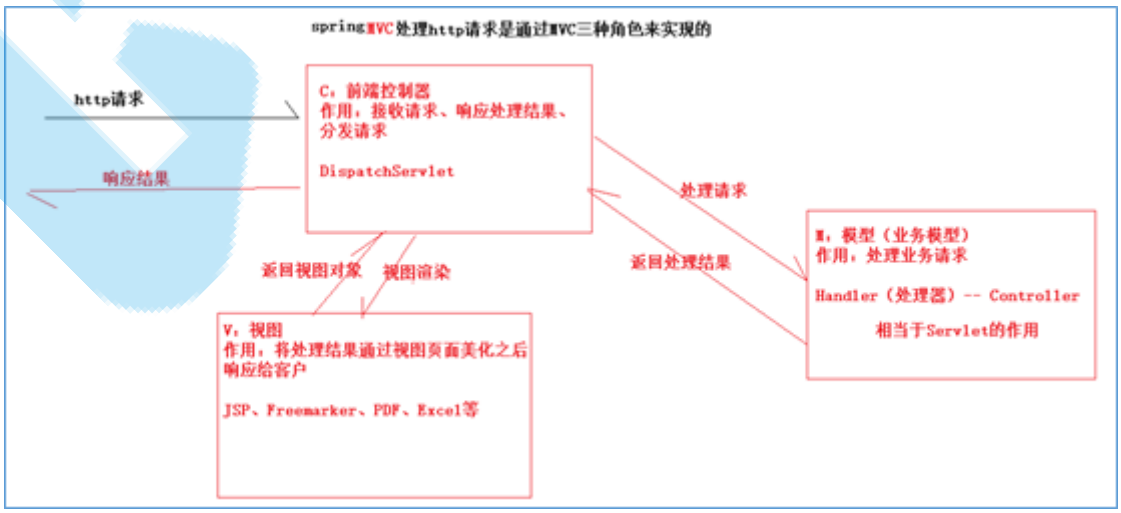
不要再被蒙蔽了，看看下面的调查统计吧：



SpringMVC的市场占有率是40%，而Struts2只有可怜的6%。这已然说明了学习SpringMVC的必要性了，再说了，SpringMVC本身就是spring家族的一员，与整合spring时，SpringMVC根本无需中间整合包，而struts2得需要。

既然已经知道了SpringMVC的重要性了，那么下面就跟着我一起看看它的神奇之处吧！

六大组件介绍



六大组件(MVC组件其他三大组件)说明：

> **说明：**

> 1. 在`springmvc`的各个组件中，**前端控制器、处理器、视图**称为`springmvc`的**MVC组件**
> 2. 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**
> 3. 需要开发的组件有：**处理器、视图**

- **DispatcherServlet**：前端控制器

  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。

- **Handler**：处理器

  Handler 是继DispatcherServlet前端控制器的后端控制器，在DispatcherServlet的控制下Handler对具体的用户请求进行处理。

  由于Handler涉及到具体的用户业务请求，所以一般情况需要程序员根据业务需求开发Handler。

- **View**：视图

  springmvc框架提供了很多的View视图类型的支持，包括：jstlView、freemarkerView、pdfView等。我们最常用的视图就是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。

- **HandlerMapping**：处理器映射器

  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。

- **HandlAdapter**：处理器适配器

  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。

- **View Resolver**：视图解析器

  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。

项目搭建篇

搭建入门工程

添加依赖

就是`mvc`依赖、`jstl`依赖还有`servlet-api`依赖

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.kkb</groupId>
    <artifactId>springmvc-demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>war</packaging>

    <dependencies>
        <!-- spring MVC依赖包 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>5.0.7.RELEASE</version>
        </dependency>

        <!-- jstl -->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>jstl</artifactId>
            <version>1.2</version>
        </dependency>

        <!-- servlet -->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.5</version>
            <scope>provided</scope>
        </dependency>

    </dependencies>
    <build>
        <plugins>
            <!-- 配置Maven的JDK编译级别 -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.2</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                    <encoding>UTF-8</encoding>
                </configuration>
            </plugin>

            <plugin>
                <groupId>org.apache.tomcat.maven</groupId>
                <artifactId>tomcat7-maven-plugin</artifactId>
                <version>2.2</version>
                <configuration>
                    <path>/</path>
                    <port>8080</port>
                </configuration>
            </plugin>
```

```
        </plugins>
    </build>
</project>
```

开发步骤

配置部分

web.xml

在`web.xml`中添加`前端控制器DispatcherServlet`的配置。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    version="2.5">

    <!-- 学习前置条件 -->
    <!-- 问题1：web.xml中servelet、filter、listener、context-param加载顺序 -->
    <!-- 问题2：load-on-startup标签的作用，影响了servlet对象创建的时机 -->
    <!-- 问题3：url-pattern标签的配置方式有四种：/dispatcherServlet、 /servlet/*  、*
、/ ,以上四种配置，优先级是如何的-->
    <!-- 问题4：url-pattern标签的配置为/*报错，原因是它拦截了JSP请求，但是又不能处理JSP请
求。为什么配置/就不拦截JSP请求，而配置/*，就会拦截JSP请求-->
    <!-- 问题5：配置了springmvc去读取spring配置文件之后，就产生了spring父子容器的问题 -->

    <!-- 配置前端控制器 -->
    <servlet>
        <servlet-name>springmvc</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!-- 设置spring配置文件路径 -->
        <!-- 如果不设置初始化参数，那么DispatcherServlet会读取默认路径下的配置文件 -->
        <!-- 默认配置文件路径：/WEB-INF/springmvc-servlet.xml -->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc.xml</param-value>
        </init-param>
        <!-- 指定初始化时机，设置为2，表示Tomcat启动时，DispatcherServlet会跟随着初始化
-->
        <!-- 如果不指定初始化时机，DispatcherServlet就会在第一次被请求的时候，才会初始化，
而且只会被初始化一次 -->
        <load-on-startup>2</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <!-- URL-PATTERN的设置 -->
        <!-- 不要配置为/*,否则报错 -->
        <!-- 通俗解释：/*，会拦截整个项目中的资源访问，包含JSP和静态资源的访问，对于静态资源
的访问springMVC提供了默认的Handler处理器 -->
        <!-- 但是对于JSP来讲，springmvc没有提供默认的处理器，我们也没有手动编写对于的处理
器，此时按照springmvc的处理流程分析得知，它短路了 -->
```

```
        <url-pattern>/</url-pattern>
    </servlet-mapping>
    <!-- <servlet> -->
    <!-- <servlet-name>sss</servlet-name> -->
    <!-- <servlet-class>sss</servlet-class> -->
    <!-- </servlet> -->
    <!-- <servlet-mapping> -->
    <!-- <servlet-name>sss</servlet-name> -->
    <!-- <url-pattern>/sss</url-pattern> -->
    <!-- </servlet-mapping> -->
</web-app>
```

springmvc.xml

`DispatcherServlet`启动时会去加载`springmvc.xml`文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        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">

    <!-- 配置处理器Bean的读取 -->
    <!-- 扫描controller注解,多个包中间使用半角逗号分隔 -->
    <context:component-scan base-package="com.kkb.springmvc.controller"/>

    <!-- 配置三大组件之处理器适配器和处理器映射器 -->
    <!-- 内置了RequestMappingHandlerMapping和RequestMappingHandlerAdapter等组件注
册-->
    <mvc:annotation-driven />

    <!-- 配置三大组件之视图解析器 -->
    <!-- InternalResourceViewResolver:默认支持JSP视图解析-->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

配置说明：

- **mvc:annotation-drivern**

```
- ContentNegotiationManagerFactoryBean
- RequestMappingHandlerMapping(重要)
- ConfigurableWebBindingInitializer
- RequestMappingHandlerAdapter(重要)
- CompositeUriComponentsContributorFactoryBean
```

```
- ConversionServiceExposingInterceptor
- MappedInterceptor
- ExceptionHandlerExceptionResolver
- ResponseStatusExceptionResolver
- DefaultHandlerExceptionResolver
- BeanNameUrlHandlerMapping
- HttpRequestHandlerAdapter
- SimpleControllerHandlerAdapter
- HandlerMappingIntrospector
```

编码部分

Controller

```
处理器开发方式有多种：
- 实现HttpRequestHandler接口
- 实现Controller接口
- 使用注解方式
```

不过企业开发中，推荐使用**注解方式**开发处理器。

```java
//@Controller：在类上添加该注解，指定该类为一个请求处理器，不需要实现任何接口或者继承任何类。
@Controller
public class HelloController {

    //@RequestMapping：在方法上或者类上添加该注解，指定请求的`url`由该方法处理。
    @RequestMapping("showView")
    public String showView() {
        return "hello";
    }
    @RequestMapping("showData")
    @ResponseBody
    public String showData() {
        return "showData";
    }
}
```

> **注意事项：**
>
> `@Controller`注解的类中每个`@RequestMapping`注解的方法，最终都会转为`HandlerMethod`类（而这个类才是`SpringMVC`注解开发方式中真正意义的处理器）

访问测试

```
访问地址：http://localhost:8080/showView
```

SSM框架整合

整合思路

将工程的三层结构中的`JavaBean`分别使用`Spring容器`（**通过XML方式**）进行管理。

1. 整合持久层`mapper`，包括`数据源`、`SqlSessionFactory`及`mapper`代理对象的整合；
2. 整合业务层`Service`，包括`事务Bean`及`service`的`bean`的配置；
3. 整合表现层`Controller`，直接使用`springmvc`的配置。
4. `Web.xml`加载`spring`容器（包含多个XML文件，还分为`父子容器`）

核心配置文件：

- `applicationContext-dao.xml`
- `applicationContext-service.xml`
- `springmvc.xml`
- `web.xml`

工程搭建

依赖包：

```
- spring（包括springmvc）
- mybatis
- mybatis-spring整合包
- 数据库驱动
- 第三方连接池
- JSTL
- servlet-api
```

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.kkb</groupId>
    <artifactId>ssm-5</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>war</packaging>

    <dependencies>
        <!-- 持久层依赖 开始 -->
        <!-- spring ioc组件需要的依赖包 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
            <version>5.0.7.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.0.7.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.0.7.RELEASE</version>
```

```
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-expression</artifactId>
            <version>5.0.7.RELEASE</version>
        </dependency>

        <!-- spring 事务管理和JDBC依赖包 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-tx</artifactId>
            <version>5.0.7.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>5.0.7.RELEASE</version>
        </dependency>

        <!-- mysql数据库驱动包 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.35</version>
        </dependency>

        <!-- dbcp连接池的依赖包 -->
        <dependency>
            <groupId>commons-dbcp</groupId>
            <artifactId>commons-dbcp</artifactId>
            <version>1.4</version>
        </dependency>

        <!-- mybatis依赖 -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.4.5</version>
        </dependency>

        <!-- mybatis和spring的整合依赖 -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis-spring</artifactId>
            <version>1.3.1</version>
        </dependency>

        <!-- 持久层依赖 结束 -->

        <!-- 业务层依赖 开始 -->
        <!-- 基于AspectJ的aop依赖 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aspects</artifactId>
            <version>5.0.7.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>aopalliance</groupId>
```

```
            <artifactId>aopalliance</artifactId>
            <version>1.0</version>
        </dependency>
        <!-- 业务层依赖 结束 -->


        <!-- 表现层依赖 开始 -->
        <!-- spring MVC依赖包 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>5.0.7.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>5.0.7.RELEASE</version>
        </dependency>

        <!-- jstl 取决于视图对象是否是JSP -->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>jstl</artifactId>
            <version>1.2</version>
        </dependency>

        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.1.0</version>
            <scope>provided</scope>
        </dependency>


        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context-support</artifactId>
            <version>5.0.7.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-oxm</artifactId>
            <version>5.0.7.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>com.thoughtworks.xstream</groupId>
            <artifactId>xstream</artifactId>
            <version>1.4.10</version>
        </dependency>


        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
            <version>2.9.6</version>
        </dependency>
```

```
        <!-- 表现层依赖 结束 -->

        <!-- spring 单元测试组件包 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>5.0.7.RELEASE</version>
        </dependency>

        <!-- 单元测试Junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>

        <!-- Mock测试使用的json-path依赖 -->
        <dependency>
            <groupId>com.jayway.jsonpath</groupId>
            <artifactId>json-path</artifactId>
            <version>2.2.0</version>
        </dependency>


        <!-- 文件上传 -->
        <dependency>
            <groupId>commons-fileupload</groupId>
            <artifactId>commons-fileupload</artifactId>
            <version>1.3.1</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <!-- 配置Maven的JDK编译级别 -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.2</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                    <encoding>UTF-8</encoding>
                </configuration>
            </plugin>

            <plugin>
                <groupId>org.apache.tomcat.maven</groupId>
                <artifactId>tomcat7-maven-plugin</artifactId>
                <version>2.2</version>
                <configuration>
                    <path>/</path>
                    <port>80</port>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

工程整合（配置文件）

整合Mapper

applicationContext-dao.xml（核心）

在`classpath`下，创建`spring`目录，然后创建`applicationContext-dao.xml`：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    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/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- 加载db.properties -->
    <context:property-placeholder location="classpath:db.properties" />
    <!-- 配置数据源 -->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <property name="driverClassName" value="${jdbc.driver}" />
        <property name="url" value="${jdbc.url}" />
        <property name="username" value="${jdbc.username}" />
        <property name="password" value="${jdbc.password}" />
        <property name="maxActive" value="30" />
        <property name="maxIdle" value="5" />
    </bean>

    <!-- 配置SqlSessionFacotory -->
    <bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">

        <!-- 配置数据源 -->
        <property name="dataSource" ref="dataSource" />
        <property name="typeAliasesPackage" value="com.kkb.ssm.po"></property>
    </bean>

    <!-- 配置mapper扫描器，SqlSessionConfig.xml中的mapper配置去掉 -->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <!-- 指定扫描的包 -->
        <property name="basePackage" value="com.kkb.ssm.mapper" />
    </bean>
</beans>
```

db.properties

在`classpath`下，创建`db.properties`：

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/ssm?characterEncoding=utf8
jdbc.username=root
jdbc.password=root
```

log4j.properties

在`classpath`下，创建`log4j.properties`：

```
#dev env [debug] product   env [info]
log4j.rootLogger=DEBUG,   stdout
# Console output...
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p   [%t] - %m%n
```

整合Service

applicationContext-service.xml

在`spring`文件夹下创建`applicationContext-service.xml`，文件中配置`service`。在这个配置文件中，需要配置`service`的`bean`和事务管理。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"

    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    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/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">

    <!-- 扫描Service -->
    <context:component-scan base-package="com.kkb.ssm.service" />

    <!-- 配置事务 -->
    <!-- 事务管理器，对mybatis操作数据库进行事务控制，此处使用jdbc的事务控制 -->
    <bean id="transactionManager"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <!-- 指定要进行事务管理的数据源 -->
        <property name="dataSource" ref="dataSource"></property>
    </bean>

    <!-- 通知 -->
```

```
    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <!-- 传播行为 -->
            <tx:method name="save*" propagation="REQUIRED" />
            <tx:method name="add*" propagation="REQUIRED" />
            <tx:method name="insert*" propagation="REQUIRED" />
            <tx:method name="delete*" propagation="REQUIRED" />
            <tx:method name="del*" propagation="REQUIRED" />
            <tx:method name="remove*" propagation="REQUIRED" />
            <tx:method name="update*" propagation="REQUIRED" />
            <tx:method name="modify*" propagation="REQUIRED" />
            <tx:method name="find*" read-only="true" />
            <tx:method name="query*" read-only="true" />
            <tx:method name="select*" read-only="true" />
            <tx:method name="get*" read-only="true" />
        </tx:attributes>
    </tx:advice>

    <!-- aop -->
    <aop:config>
        <aop:advisor advice-ref="txAdvice"
            pointcut="execution(* com.kkb.ssm.service.impl.*.*(..))" />
    </aop:config>
</beans>
```

整合Controller

`Spring`和`springmvc`之间无需整合，直接用springmvc的配置

web.xml

在`web.xml`中添加`springmvc`的配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    id="WebApp_ID" version="2.5">

    <!-- 配置springmvc的前端控制器 -->
    <servlet>
        <servlet-name>springmvc</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:spring/springmvc.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
```

```
</web-app>
```

springmvc.xml

在`spring`包下创建`springmvc.xml`文件，内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    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">

    <!-- 配置处理器映射器和处理器适配器 -->
    <mvc:annotation-driven />

    <!-- 配置视图解析器 -->
    <bean

class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>
    <!-- 使用注解的handler可以使用组件扫描器，加载handler -->
    <context:component-scan base-package="com.kkb.ssm.controller" />
</beans>
```

web.xml加载spring父容器

在web.xml中，使用监听器来对spring的配置文件进行加载：

```
<!-- 加载spring容器 -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:spring/applicationContext-*.xml
    </param-value>
</context-param>
<listener>
    <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

整合测试（编写代码）

需求

实现商品查询列表，从`mysql`数据库查询商品信息。

需求分析

- **表现层**
  - 请求URL：/queryItem
  - 请求参数：无
  - 请求返回值：json格式数据
- **业务层**
  - 业务处理逻辑（需求分析）：实现商品列表的查询
- **持久层**
  - 只针对表进行增删改查操作

持久层代码

根据需求分析，持久层需要查询item表中的记录，使用逆向工程的代码即可。

通过逆向工程，把po类、mapper.xml和mapper.java类生成出来并拷贝到项目中。

业务层代码

根据需求开发service的接口以及实现类，**注意：使用注解 @Service开发service。**

```
@Service
public class ItemServiceImpl implements ItemService {

    @Autowired
    private ItemMapper mapper;

    public List<Item> queryItemList() {
        ItemExample example = new ItemExample();
        return mapper.selectByExample(example);
    }

}
```

表现层代码

在`Controller`类上添加 **@Controller** 注解

在`Controller`方法上添加 **@RequestMapping**注解进行url请求映射

```
@Controller
public class ItemController {
    @Autowired
```

```
    private ItemService service;

    @RequestMapping("/item")
    @ResponseBody
    public List<Item> queryItem() {
        // 根据查询条件去数据库中查询商品列表
        List<Item> itemList = service.queryItemList();

        return itemList;
    }
}
```

部署测试

http://localhost:8080/ssm/item

应用掌握篇

返回值处理

不使用注解修饰

ModelAndView

```
Controller方法中定义ModelAndView对象并返回，对象中可添加model数据、指定view。
```

void

在Controller**方法形参上可以定义request和response**，使用`request`或`response`指定响应结果：

```
void service(HttpServletRequest request,HttpServletResponse response){}
```

1、使用request转发向页面，如下：

```
request.getRequestDispatcher("页面路径").forward(request, response);
```

2、也可以通过response页面重定向：

```
response.sendRedirect("url")
```

3、也可以通过response指定响应结果，例如响应json数据如下：

```
response.setCharacterEncoding("utf-8");
response.setContentType("application/json;charset=utf-8");
response.getWriter().write("json串");
```

String（推荐）

逻辑视图名

```
return "item/item-list";
```

redirect重定向

```
return "redirect:testRedirect";
```

> **redirect:**
>
> - 相当于“`response.sendRedirect()`”
> - 浏览器URL发生改变
> - Request域不能共享

forward转发

```
return “forward:testForward";
```

> **forward:**
>
> - 相当于“`request.getRequestDispatcher().forward(request,response)`”
> - 浏览器URL不发送改变
> - `Request`域可以共享

使用注解修饰

ResponseBody注解介绍

注解介绍

- **ResponseBody注解的作用：**

```
一、ResponseBody注解可以针对Controller返回值类型，使用内置的9种HttpMessageConverter进行匹配，找到合适的HttpMessageConverter进行处理。
二、HttpMessageConverter处理逻辑分为三步：
    1、指定HttpServletResponse的ContentType值。
    2、将转换之后的数据放到HttpServletResponse对象的响应体返回到页面
```

- **@RequestBody**注解的作用和**@ResponseBody**注解正好相反，它是处理**请求参数**的http消息转换的。

常用的HttpMessageConverter

- **MappingJacksonHttpMessageConverter**

```
作用：处理POJO类型返回值
默认使用MappingJackson的JSON处理能力，将后台返回的Java对象（POJO类型），转为JSON格式输出到页面
将响应体的Content-Type设置为application/json；charset=utf-8
调用response.getWriter()方法将json格式的字符串写回给调用者。
```

- **StringHttpMessageConverter**

```
作用：处理String类型返回值
将响应体的Content-Type设置为text/plain；charset=utf-8
调用response.getWriter()方法将String类型的字符串写回给调用者。
```

ResponseBody示例

JSP示例代码

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSON格式数据参数绑定和返回值处理演示demo</title>
    </head>
    <body>
        <!-- 使用@ResponseBody注解-->
        <a href="${pageContext.request.contextPath}/responsebody/returnString">
测试String类型返回值</a>
        <a href="${pageContext.request.contextPath}/responsebody/returnPOJO">测试
POJO类型返回值</a>
        <!-- 使用@RestController注解-->
        <a
href="${pageContext.request.contextPath}/restcontroller/returnString">测试String
类型返回值</a>
        <a href="${pageContext.request.contextPath}/restcontroller/returnPOJO">
测试POJO类型返回值</a>
    </body>
</html>
```

Controller代码

- **方式1（使用@ResponseBody注解）：**

```
@RequestMapping("responsebody")
```

```
@Controller
public class MyResponseBodyController {

    // @RequestMapping注解中的consumes和produces分别是为请求头和响应头设置contentType
    @RequestMapping(value = "returnString", produces = "text/plain;charset=UTF-
8")
    @ResponseBody
    public String returnString() {
        // 如果在使用@ResponseBody注解的前提下，如果返回值是String类型，则返回值会由
StringHttpMessageConverter进行处理
        return "查询失败";
    }

    @RequestMapping("returnPOJO")
    @ResponseBody
    public User returnPOJO() {
        User user = new User();
        user.setId(1);
        user.setUsername("bingbing");
        user.setSex("女");

        // 如果在使用@ResponseBody注解的前提下，如果返回值是POJO类型，则返回值会由
MappingJacksonHttpMessageConverter（需要第三方jar包支持）进行处理
        return user;
    }
}
```

- **方式2（使用@RestController注解）：**

```
/**
 * 用来学习JSON交互
 * @author think
 *
 */
@RequestMapping("restcontroller")
@RestController //相当于Controller注解和ResponseBody注解的组合
public class MyRestcontroller {

    //@RequestMapping注解中的consumes和produces分别是为请求头和响应头设置contentType
    @RequestMapping(value="returnString",produces="text/plain;charset=UTF-8")
    public String returnString() {
        // 如果在使用@ResponseBody注解的前提下，如果返回值是String类型，则返回值会由
StringHttpMessageConverter进行处理
        return "查询失败";
    }
    @RequestMapping("returnPOJO")
    public User returnPOJO() throws CustomException{

        User user = new User();
        user.setId(1);
        user.setUsername("bingbing");
        user.setSex("女");

        // 如果在使用@ResponseBody注解的前提下，如果返回值是POJO类型，则返回值会由
MappingJacksonHttpMessageConverter（需要第三方jar包支持）进行处理
        return user;
    }
```

```
}
```

参数绑定处理

什么是参数绑定

- **什么是参数绑定?**

  ```
  就是将请求参数串中的value值获取到之后，再进行类型转换，然后将转换后的值赋值给Controller
  类中方法的形参，这个过程就是参数绑定。
  总结参数绑定需要两步：
      --类型转换（请求中的String类型值--->Controller各种数据类型的方法形参）
      --赋值操作，将转换之后的值赋值给Controller方法形参
  ```

- **请求参数格式**

  ```
  默认是key/value格式，比如：    http://XXXXX?id=1&type=301
  ```

- **请求参数值的数据类型**

  ```
  都是String类型的各种值
  ```

- **请求参数值要绑定的目标类型**

  ```
  Controller类中的方法参数，比如简单类型、POJO类型、集合类型等。
  ```

- **SpringMVC内置的参数解析组件**

  ```
  默认内置了24种参数解析组件（ArgumentResolver）
  ```

默认支持的参数类型

Controller方法形参中可以随时添加如下类型的参数（Servlet API支持），处理适配器会自动识别并进行赋值

- **HttpServletRequest**

  通过`request`对象获取请求信息

- **HttpServletResponse**

  通过`response`处理响应信息

- **HttpSession**

  通过`session`对象得到`session`中存放的对象

- **InputStream、OutputStream**
- **Reader、Writer**
- **Model/ModelMap**

ModelMap继承自LinkedHashMap，Model是一个接口，它们的底层实现都是同一个类（BindingAwareModelMap），作用就是向页面传递数据，相当于Request的作用，如下：

```
model.addAttribute("msg", “测试springmvc”);
```

参数绑定使用要求

简单类型

直接绑定

http请求参数的【key】和controller方法的【形参名称】一致

注解绑定

请求参数的【key】和controller方法的【形参名称】不一致时，需要使用【@RequestParam】注解才能将请求参数绑定成功。

RequestParam注解

- value：

参数名字，即入参的请求参数名字，如value=“itemid”表示请求的参数中的名字为itemid的参数的值将传入

- required：

是否必须，默认是true，表示请求中一定要有相应的参数，否则将报；

```
TTP Status 400 - Required Integer parameter 'XXXX' is not present
```

- defaultValue：

默认值，表示如果请求中没有同名参数时的默认值

绑定POJO类型

要求表单中【参数名称】和Controller方法中的【POJO形参的属性名称】保持一致。

绑定集合或者数组类型

简单类型数组

通过HTTP请求批量传递简单类型数据的情况，Controller方法中可以用String[]或者pojo的String[]属性接收（两种方式任选其一），但是不能使用List集合接收。

POJO类型集合或者数组

批量传递的请求参数，最终要使用List<POJO>来接收，那么这个List<POJO>必须放在另一个POJO类中。

参数绑定示例

JSP代码

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>参数绑定演示demo</title>
</head>
<body>
    <!-- request请求的内容类型主要分为：K/V类型、Multipart类型、JSON类型 -->

    <!-- 将request请求参数，绑定到简单类型（基本类型和String类型）方法参数 -->
    <!-- 直接绑定 -->
    <a href="${pageContext.request.contextPath}/user/findUserById?
id=1&name=bingbing">查询用户1</a>
    <!-- @RequestParam注解绑定 -->
    <a href="${pageContext.request.contextPath}/user/findUserById2?uid=1">查询用
户2</a>

    <!-- 将request请求参数，绑定到POJO类型(简单POJO和包装POJO的)方法参数 -->
    <form action="${pageContext.request.contextPath}/user/saveUser"
method="post">
        用户名称：<input type="text" name="username"><br />
        用户性别：<input type="text" name="sex"><br />
        所属省份：<input type="text" name="address.provinceName"><br />
        所属城市：<input type="text" name="address.cityName"><br />
        <input type="submit" value="保存">
    </form>

    <!-- 将request请求参数，绑定到[元素是简单类型的集合或数组]参数 -->
    <!-- 使用数组接收 -->
    <a href="${pageContext.request.contextPath}/user/findUserByIds?
id=1&id=2&id=3">根据ID批量删除用户</a>
    <!-- 使用List接收（错误示例） -->
    <a href="${pageContext.request.contextPath}/user/findUserByIds2?
id=1&id=2&id=3">根据ID批量删除用户</a>
    <!-- 使用Bean的List属性接收 -->
    <a href="${pageContext.request.contextPath}/user/findUserByIds3?
uid=1&uid=2&uid=3">根据ID批量删除用户</a>
```

```
    <!-- 将request请求参数，绑定到[元素是POJO类型的List集合或Map集合]参数 -->
    <form action="${pageContext.request.contextPath}/user/updateUser"
method="post">
        用户名称：<input type="text" name="username"><br />
        用户性别：<input type="text" name="sex"><br />
        <!-- itemList[集合下标]：集合下标必须从0开始 -->
        <!-- 辅助理解：先将name属性封装到一个Item对象中，再将该Item对象放入itemList集合的
指定下标处 -->
        购买商品1名称：<input type="text" name="itemList[0].name"><br />
        购买商品1价格：<input type="text" name="itemList[0].price"><br />
        购买商品2名称：<input type="text" name="itemList[1].name"><br />
        购买商品2价格：<input type="text" name="itemList[1].price"><br />
        <!-- itemMap['item3']：其中的item3、item4就是Map集合的key -->
        <!-- 辅助理解：先将name属性封装到一个Item对象中，再将该Item对象作为value放入
itemMap集合的指定key处 -->
        购买商品3名称：<input type="text" name="itemMap['item3'].name"><br />
        购买商品3价格：<input type="text" name="itemMap['item3'].price"><br />
        购买商品4名称：<input type="text" name="itemMap['item4'].name"><br />
        购买商品4价格：<input type="text" name="itemMap['item4'].price"><br />
        <input type="submit" value="保存">
    </form>

    <!-- 将request请求参数，绑定到Date类型方法参数 -->
    <!-- 请求参数是：【年月日】 格式 -->
    <a href="${pageContext.request.contextPath}/user/deleteUser?birthday=2018-
01-01">根据日期删除用户(String)</a>
    <!-- 请求参数是：【年月日 时分秒】 格式 -->
    <a href="${pageContext.request.contextPath}/user/deleteUser2?birthday=2018-
01-01 12:10:33">根据日期删除用户(Date)</a>

    <!-- 文件类型参数绑定 -->
    <form action="${pageContext.request.contextPath}/fileupload" method="post"
enctype="multipart/form-data">
        图片：<input type="file" name="uploadFile" /><br />
        <input type="submit" value="上传" />
    </form>

</body>
</html>
```

Controller代码

```
/**
 * 用来学习参数绑定
 * @author think
 *
 */
@RequestMapping("user")
@Controller
public class UserController {

    @RequestMapping("findUserById")
    public String findUserById(Integer id,Model model,HttpServletRequest
request) {
```

```
        model.addAttribute("msg", "直接参数绑定接收到的参数："+id);
        model.addAttribute("msg", "通过Request getParameter参数接收到的参
数："+request.getParameter("id"));
        return "success";
    }
    //  @RequestParam：可以理解为request.getParameter("参数key")
    @RequestMapping("findUserById2")
    public String findUserById2(@RequestParam("uid") Integer id,Model model) {
        model.addAttribute("msg", "接收到的参数："+id);
        return "success";
    }
    @RequestMapping("saveUser")
    public String saveUser(User user,Model model) {
        model.addAttribute("msg", "接收到的参数："+user.toString());
        return "success";
    }
    @RequestMapping("deleteUser")
    public String deleteUser(String birthday,Model model) {
        model.addAttribute("msg", "接收到的参数："+birthday);
        return "success";
    }
    @RequestMapping("deleteUser2")
    public String deleteUser2(Date birthday,Model model) {
        model.addAttribute("msg", "接收到的参数："+birthday);
        return "success";
    }
    @RequestMapping("findUserByIds")
    public String findUserByIds(Integer[] id,Model model) {
        model.addAttribute("msg", "接收到的参数："+id);
        return "success";
    }
    @RequestMapping("findUserByIds2")
    public String findUserByIds2(List<Integer> id,Model model) {
        model.addAttribute("msg", "接收到的参数："+id);
        return "success";
    }
    @RequestMapping("findUserByIds3")
    public String findUserByIds3(User user,Model model) {
        model.addAttribute("msg", "接收到的参数："+user.getUid());
        return "success";
    }
    @RequestMapping("updateUser")
    public String updateUser(User user,Model model) {
        model.addAttribute("msg", "接收到的参数："+user.getUid());
        return "success";
    }
}
```

PO代码

```
public class User {
    private int id;
    private String username;
    private Date birthday;
    private String sex;
```

```
    // 演示包装POJO参数绑定
    private Address address;

    // 演示批量简单类型参数接收
    private List<Integer> uid = new ArrayList<>();

    // 将request请求参数，绑定到[元素是POJO类型的List集合]参数
    private List<Item> itemList = new ArrayList<>();

    // 将request请求参数，绑定到[元素是POJO类型的Map集合]参数
    private Map<String, Item> itemMap = new HashMap<>();

    //setter/getter方法
}
```

自定义日期参数绑定

对于springmvc无法解析的参数绑定类型，比如[年月日时分秒格式的日期]绑定到Date类型会报错，此时需要自定义[参数转换器]进行参数绑定。

Converter代码

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

Converter配置

```
    <!-- 加载注解驱动 -->
    <mvc:annotation-driven conversion-service="conversionService"/>
    <!-- 转换器配置 -->
    <bean id="conversionService"
class="org.springframework.format.support.FormattingConversionServiceFactoryBean
">
        <property name="converters">
            <set>
                <bean class="com.kkb.ssm.controller.converter.DateConverter"/>
            </set>
        </property>
    </bean>
```

文件类型参数绑定

`SpringMVC`文件上传的实现，是由`commons-fileupload`这个第三方jar包实现的。

加入依赖包

```
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.3.1</version>
</dependency>
```

上传页面

```
JSP中的form表单需要指定enctype=”multipart/form-data”
```

配置Multipart解析器

在`springmvc.xml`中配置`multipart`类型解析器：

```
<!-- multipart类型解析器，文件上传 -->
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!-- 上传文件的最大尺寸 5M-->
    <property name="maxUploadSize" value="5242880"/>
</bean>
```

Controller类代码

```
@RequestMapping("fileupload")
    public String findUserById(MultipartFile uploadFile) throws Exception {
        // 编写文件上传逻辑（mvc模式和三层结构模式）
        // 三层模式：表现层（controller、action）、业务层（service、biz）、持久层（dao、
mapper）
```

```
        // MVC模式主要就是来解决表现层的问题的（原始的表现层是使用Servlet编写，即编写业务逻辑，又编写视图展示）

        if (uploadFile != null) {
            System.out.println(uploadFile.getOriginalFilename());
            // 原始图片名称
            String originalFilename = uploadFile.getOriginalFilename();
            // 如果没有图片名称，则上传不成功
            if (originalFilename != null && originalFilename.length() > 0) {
                // 存放图片的物理路径
                String picPath = "E:\\";
                // 获取上传文件的扩展名
                String extName = originalFilename.substring(originalFilename.lastIndexOf("."));
                // 新文件的名称
                String newFileName = UUID.randomUUID() + extName;
                // 新的文件
                File newFile = new File(picPath + newFileName);
                // 把上传的文件保存成一个新的文件
                uploadFile.transferTo(newFile);
                // 同时需要把新的文件名更新到数据库中
            }
        }

        return "文件上传成功";
    }
```

RequestMapping注解

value属性

请求URL映射

- **作用：用于映射URL和HandlerMethod方法。**
- **用法如下：**

```
@RequestMapping(value="/item")
@RequestMapping("/item“）
@RequestMapping(value={"/item","/queryItem”})
```

窄化请求映射

- **作用：**限制此类下的所有方法的访问请求url必须以**请求前缀**开头，对url进行**模块化分类**管理。
- **用法如下：**

  访问时的URL是`/item/findItem`

```
@RequestMapping("item")
@Controller
public class ItemController {

   @RequestMapping("findItem")
   public String findItem(Model model) {
      model.addAttribute("msg", "ItemController...findItem方法执行了");
      return "success";
   }
}
```

method属性

- **作用：限定请求URL只能通过指定的method请求方式去访问该 `HandlerMethod`**
- **用法如下**：

```
@RequestMapping(value="/findItem",method=RequestMethod.GET)
@RequestMapping(value="/findItem",method=RequestMethod.POST)
@RequestMapping(value="/findItem",method=
{RequestMethod.GET,RequestMethod.POST})
```

params属性

- **作用**：通过设置 `params` 参数条件，进行访问 `HandlerMethod` 的限制
- **用法如下**：
  - URL请求

```
<a href="item/removeItem?name=iphone6&price>5000">删除商品，金额大于
5000</a>
<br />
<a href="item/removeItem?name=iphoneXs&price>7000">删除商品，金额大于
7000</a>
```

  - Controller方法：

```
   @RequestMapping(value="removeItem",params= {"name","price>5000"})
    public String removeItem(Model model) {
        model.addAttribute("msg", "ItemController...removeItem方法执行
了");
        return "success";
    }
```

RESTful支持

理解什么是REST之前，先去理解一下什么是HTTP

HTTP介绍

HTTP协议概述

什么是HTTP协议?

> HTTP协议是建立在客户端和服务器之间的一个应用层协议，在客户端和服务器之间需要数据的传输，而传输数据的时候，我们要按照指定的规则或者叫协议去传输数据。

> * HTTP是建立在TCP/IP协议基础之上的一个网络协议。
> * HTTP协议属于网络七层结构中最上层（应用层）的协议。
> * HTTP协议是一个无状态协议（不会记录每次访问时的信息）
> * HTTP是一个客户端和服务器端请求和应答的标准（TCP）。客户端是终端用户，服务器端是网站。

什么是URL/URI?

- **URI**：Uniform Resource Identifier，**统一资源标识符。**

> 它相当于一个网络资源的名称，只是名称的表现形式是/开头的路径形式。

- **URL**：Uniform Resource Location，统一资源定位符。
- **URL和URI的区别**：URL是URI的子集。

什么是WEB资源?

通过浏览器可以访问到的所有资源都是`web资源`，web资源分为**静态资源**和**动态资源**：

- 动态资源是通过后台程序展示页面数据的，比如Servlet请求。
- 静态资源的数据是不变的，比如HTML、JPG、AVI。

HTTP的作用是什么?

就是为了约束客户端和服务器之间传输web资源时的格式。

HTTP协议版本

- HTTP协议现在有哪些版本？
- 1.0版本和1.1版本的区别是什么？

HTTP1.1和HTTP1.0版本之间最大的区别就是：*可以一个连接传输多个web资源。*

推荐使用HTTP1.1版本！！！

HTTP协议组成

HTTP协议由两部分组成：**请求协议信息**和**响应协议信息。**

请求协议信息

- 请求协议由哪几部分组成？
- 请求协议的请求行包含哪些信息？
- 请求协议的请求头如何理解？请求头中常用的一些配置的作用各自是什么？
- MIME是什么？常见的MIME类型有哪些？
- 请求协议的请求体有几种表现形式？

HTTP请求协议信息由三部分组成：请求行、请求头、请求体，简称行头体。

- 请求行

也叫请求首行，它包含四部分（请求方法、URI、协议/版本、回车换行）：

```
GET /user.html HTTP/1.1
```

请求方法：

GET、POST等8种
互联网中WEB资源操作也有增删改查方法，它们分别是POST、DELETE、PUT和GET。

根据HTTP标准，HTTP请求可以使用多种请求方法。

HTTP1.0定义了三种请求方法： GET, POST 和 HEAD方法。

HTTP1.1新增了五种请求方法：OPTIONS, PUT, DELETE, TRACE 和 CONNECT 方法。

序号	方法	描述
1	GET	请求指定的页面信息，并返回实体主体。
2	HEAD	类似于get请求，只不过返回的响应中没有具体的内容，用于获取报头
3	POST	向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的建立和/或已有资源的修改。
4	PUT	从客户端向服务器传送的数据取代指定的文档的内容。
5	DELETE	请求服务器删除指定的页面。
6	CONNECT	HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。
7	OPTIONS	允许客户端查看服务器的性能。
8	TRACE	回显服务器收到的请求，主要用于测试或诊断。

URI：

Uniform Resource Identifier，统一资源标识符。它相当于一个网络资源的名称，只是名称的表现形式是/开头的路径形式。

URL：Uniform Resource Location，统一资源定位符

URL和URI的区别：URL是URI的子集。

协议/版本：

表示这次请求是通过哪个协议发送的，比如HTTP协议、HTTPS协议等，使用的HTTP协议一般都是1.1版本的。

- 请求头

请求头的信息是以[key:value]形式展现的。

一般来说，大多数请求头的信息都不是必须的，我们只需要了解一些常见的请求头信息即可！

请求头说明：

Content-Type	**是请求消息中非常重要的内容，表示请求正文中的文档属于什么MIME类型。Content-Type: [type]/[subtype]; parameter。例如最常见的就是text/html，它的意思是说返回的内容是文本类型，这个文本又是HTML格式的。**
Host	指定请求资源的Intenet主机和端口号，必须表示请求url的原始服务器或网关的位置。HTTP/1.1请求必须包含主机头域，否则系统会以400状态码返回
Accept	浏览器可接受的MIME类型
Accept-Charset	浏览器可接受的字符集
Accept-Encoding	浏览器能够进行解码的数据编码方式，比如gzip。Servlet能够向支持gzip的浏览器返回经gzip编码的HTML页面。许多情形下这可以减少5到10倍的下载时间
Accept-Language	浏览器所希望的语言种类，当服务器能够提供一种以上的语言版本时要用到
Authorization	授权信息，通常出现在对服务器发送的WWW-Authenticate头的应答中
Connection	表示是否需要持久连接。如果Servlet看到这里的值为“Keep- Alive”，或者看到请求使用的是HTTP1.1（HTTP 1.1默认进行持久连接），它就可以利用持久连接的优点，当页面包含多个元素时（例如Applet，图片），显著地减少下载所需要的时间。要实现这一点，Servlet需要在应答中发送一个Content-Length头，最简单的实现方法是：先把内容写入 ByteArrayOutputStream，然后在正式写出内容之前计算它的大小
Content-Length	表示请求消息正文的长度
Cookie	这是最重要的请求头信息之一，可以在客户端记录访问状态。
From	请求发送者的email地址，由一些特殊的Web客户程序使用，浏览器不会用到它
If-Modified-Since	只有当所请求的内容在指定的日期之后又经过修改才返回它，否则返回304“Not Modified”应答
Pragma	指定“no-cache”值表示服务器必须返回一个刷新后的文档，即使它是代理服务器而且已经有了页面的本地拷贝
Referer	包含一个URL，用户从该URL代表的页面出发访问当前请求的页面，使用场景：防盗链、统计网站访问信息。
User-Agent	浏览器类型（**客户端类型**），如果Servlet返回的内容与浏览器类型有关则该值非常有用
UA-Pixels，UA-Color，UA-OS，UA-CPU	由某些版本的IE浏览器所发送的非标准的请求头，表示屏幕大小、颜色深度、操作系统和CPU类型

- **MIME概述：**多用途互联网邮件扩展类型，也叫媒体类型。

MIME（多用途互联网邮件扩展类型）

编辑

本词条由"科普中国"百科科学词条编写与应用工作项目 审核 。

MIME(Multipurpose Internet Mail Extensions)多用途互联网邮件扩展类型。是设定某种扩展名的文件用一种应用程序来打开的方式类型，当该扩展名文件被访问的时候，浏览器会自动使用指定应用程序来打开。多用于指定一些客户端自定义的文件名，以及一些媒体文件打开方式。

它是一个互联网标准，扩展了电子邮件标准，使其能够支持：

- **MIME格式**：大类型/小类型，阅读是反过来，比如text/html,读成html文本。

- **常见MIME类型如下**：

常见的MIME类型如下：

- text/html ： HTML格式
- text/plain ：纯文本格式
- text/xml ： XML格式
- image/gif ：gif图片格式
- image/jpeg ：jpg图片格式
- image/png：png图片格式

以application开头的媒体格式类型：

- application/xhtml+xml ：XHTML格式
- application/xml ： XML数据格式
- application/atom+xml ：Atom XML聚合格式
- application/json ： JSON数据格式
- application/pdf ：pdf格式
- application/msword ： Word文档格式
- application/octet-stream ： 二进制流数据（如常见的文件下载）
- application/x-www-form-urlencoded ： <form encType="" >中默认的encType，form表单数据被编码为key/value格式发送到服务器（表单默认的提交数据的格式）

另外一种常见的媒体格式是上传文件之时使用的：

- multipart/form-data ： 需要在表单中进行文件上传时，就需要使用该格式

- 请求体

也叫请求正文。

* GET请求的请求体是空的，请求参数都是通过请求行传给服务器端。
* POST请求的请求体可以承载数据，请求头和请求体之间有一个空行作为分割线。

- **通过表单POST提交的请求体的表现形式主要有三种**：

值	描述
application/x-www-form-urlencoded	在发送前编码所有字符（默认）
multipart/form-data	不对字符编码。 在使用包含文件上传控件的表单时，必须使用该值。
text/plain	空格转换为 "+" 加号，但不对特殊字符编码。

- application/x-www-form-urlencoded：会对中文进行URL编码，并且多个参数以&连接，上传文件只能上传文件名称。

- text/plain：纯文本方式，不会对中文进行URL编码，不会使用&连接多个key-value参数，上传文件只能上传文件名称。

- multipart/form-data：多部件表现形式，这种方式主要可以完成文件上传，可以将上传的文件名称和文件内容都传递给服务器端。


```
<h1>POST方式--application/x-www-form-urlencoded</h1>
<form action="#" method="post" enctype="application/x-www-form-urlencoded">
    姓名：<input type="text" name="name"/>
    年龄：<input type="text" name="age"/>
    帅照：<input type="file" name="file" />
    <input type="submit" value="POST">
</form>
<h1>POST方式--text/plain</h1>
<form action="#" method="post" enctype="text/plain">
    姓名：<input type="text" name="name"/>
    年龄：<input type="text" name="age"/>
    帅照：<input type="file" name="file" />
    <input type="submit" value="POST">
</form>
<h1>POST方式--multipart/form-data</h1>
<form action="#" method="post" enctype="multipart/form-data">
    姓名：<input type="text" name="name"/>
    年龄：<input type="text" name="age"/>
    帅照：<input type="file" name="file" />
    <input type="submit" value="POST">
</form>
```

- **总结**

请求协议由三部分组成：行头体

* 请求首行：请求方法 web资源URI http/1.1

* 请求头：key value方式，不同的请求头配置会告诉服务器端不同的辅助信息。

* 请求体：承载传输的具体数据，不过请求体中的数据表现形式有三种，这三种都是通过form表单的enctype属性来决定的。

响应协议信息

- 响应协议由哪几部分组成？
- 响应协议的状态行包含哪些信息？
- 状态行中的状态码的含义分别是如何表示的？列出常见的几个状态码及说明？
- 响应协议的响应头包含哪些头信息？
- 响应协议的响应体如何理解？

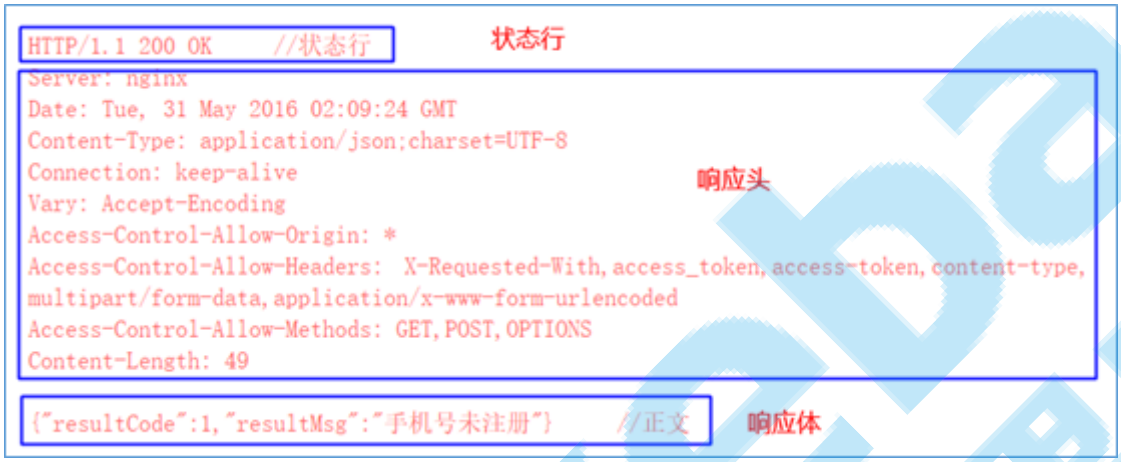
响应协议由哪几部分组成?

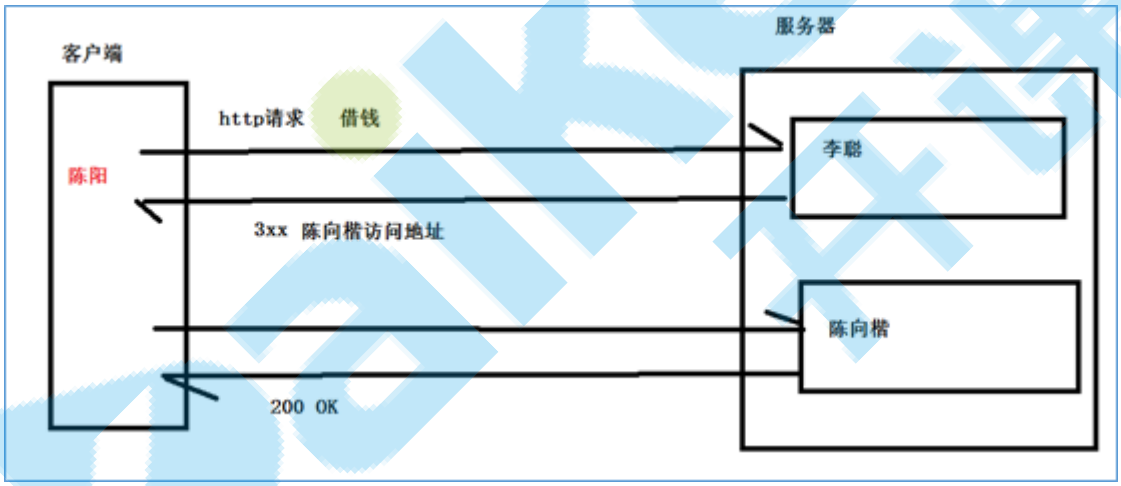
响应协议信息，也由三部分组成：状态行、响应头、响应体（响应正文）。

- 状态行

```
HTTP/1.1 200 OK
```

状态行由**协议/版本**、**数字形式的状态码**、**状态描述**三部分组成。





状态码说明：

状态代码有三位数字组成，第一个数字定义了响应的类别，共分五种类别:

1xx：指示信息--表示请求已接收，继续处理

2xx：成功--表示请求已被成功接收、理解、接受

3xx：重定向--要完成请求必须进行更进一步的操作

4xx：客户端错误--请求有语法错误或请求无法实现

5xx：服务器端错误--服务器未能实现合法的请求

常见状态码：

```
200 OK                      //客户端请求成功
400 Bad Request             //客户端请求有语法错误，不能被服务器所理解
401 Unauthorized            //请求未经授权，这个状态代码必须和WWW-Authenticate报头域一起使用
403 Forbidden               //服务器收到请求，但是拒绝提供服务
404 Not Found               //请求资源不存在，eg：输入了错误的URL
500 Internal Server Error   //服务器发生不可预期的错误
503 Server Unavailable      //服务器当前不能处理客户端的请求，一段时间后可能恢复正常
```

- 响应头

响应头中的信息也是key value方式展现的。

Content-Type	**是返回消息中非常重要的内容，表示后面的文档属于什么MIME类型。Content-Type: [type]/[subtype]; parameter。例如最常见的就是text/html，它的意思是说返回的内容是文本类型，这个文本又是HTML格式的。原则上浏览器会根据Content-Type来决定如何显示返回的消息体内容**
Location	Location响应报头域用于重定向接受者到一个新的位置。例如：客户端所请求的页面已不存在原先的位置，为了让客户端重定向到这个页面新的位置，服务 器端可以发回Location响应报头后使用重定向语句，让客户端去访问新的域名所对应的服务器上的资源。当我们在JSP中使用重定向语句的时候，服务器 端向客户端发回的响应报头中，就会有Location响应报头域。
Server	Server响应报头域包含了服务器用来处理请求的软件信息。它和User-Agent请求报头域是相对应的，前者发送服务器端软件的信息，后者发送客户 端软件(浏览器)和操作系统的信息。下面是Server响应报头域的一个例子：Server: Apache-Coyote/1.1

Content-Type	**是返回消息中非常重要的内容，表示后面的文档属于什么MIME类型。Content-Type: [type]/[subtype]; parameter。例如最常见的就是text/html，它的意思是说返回的内容是文本类型，这个文本又是HTML格式的。原则上浏览器会根据Content-Type来决定如何显示返回的消息体内容**
WWW-Authenticate	WWW-Authenticate响应报头域必须被包含在401(未授权的)响应消息中，这个报头域和前面讲到的Authorization请求报头域是 相关的，当客户端收到401响应消息，就要决定是否请求服务器对其进行验证。如果要求服务器对其进行验证，就可以发送一个包含了 Authorization报头域的请求，下面是WWW-Authenticate响应报头域的一个例子：WWW-Authenticate: Basic realm="Basic Auth Test!" 从这个响应报头域，可以知道服务器端对我们所请求的资源采用的是基本验证机制。
Content-Length	表示响应消息正文的长度
Expires	Expires实体报头域给出响应过期的日期和时间。通常，代理服务器或浏览器会缓存一些页面。当用户再次访问这些页面时，直接从缓存中加载并显示给用户，这样缩短了响应的时间，减少服务器的负载。为了让代理服务器或浏览器在一段时间后更新页面，我们可以使用Expires实体报头域指定页面过期的时 间。当用户又一次访问页面时，如果Expires报头域给出的日期和时间比Date普通报头域给出的日期和时间要早(或相同)，那么代理服务器或浏览器就 不会再使用缓存的页面而是从服务器上请求更新的页面。不过要注意，即使页面过期了，也并不意味着服务器上的原始资源在此时间之前或之后发生了改变。
Last-Modified	Last-Modified实体报头域用于指示资源最后的修改日期及时间。
Set-Cookie	设置和页面关联的Cookie。Servlet不应使用response.setHeader("Set-Cookie", ...)，而是应使用HttpServletResponse提供的专用方法addCookie。参见下文有关Cookie设置的讨论。
Allow	服务器支持哪些请求方法（如GET、POST等）。
Content-Encoding	文档的编码（Encode）方法。只有在解码之后才可以得到Content-Type头指定的内容类型。利用gzip压缩文档能够显著地减少HTML文档的下载时间。Java的GZIPOutputStream可以很方便地进行gzip压缩，但只有Unix上的Netscape和Windows上的IE 4、IE 5才支持它。因此，Servlet应该通过查看Accept-Encoding头（即request.getHeader("Accept-Encoding")）检查浏览器是否支持gzip，为支持gzip的浏览器返回经gzip压缩的HTML页面，为其他浏览器返回普通页面。

- **响应体**

响应体，也叫响应正文，里面包含服务器发给客户端的web资源信息。

响应正文信息返回到浏览器时，浏览器需要根据响应头中**Content-type**设置的MIME类型来打开响应正文信息。

GET请求和POST请求的区别

- 提交数据的方式不同

> GET是通过请求行提交请求参数的。
>
> POST是通过请求体提交请求参数的。

- 使用场景不同

> GET请求的目的是获取到数据，简单点说，就是客户端向服务器端要东西
>
> POST请求的目的是给服务器提交数据。就是客户端向服务器端给东西。

- 传递参数的大小不同

> GET请求是通过请求行中的请求URL传递给客户端的。HTTP协议对请求URL的长度没有限制，但是不同的浏览器对请求URL长度是由限制的。
>
> POST请求是通过请求体传递请求参数的。

- 总之POST传递的请求参数大小比GET方式要大，要多。

什么是RESTful

- **什么是REST**

> REST（英文：Representational State Transfer，简称 REST，意思是：（资源）表述性状态转化）描述了一个架构样式的网络系统， 比如 web 应用程序。
>
> 它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。它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，更易于实现缓存等机制。
>
> 它本身并没有什么实用性，其核心价值在于如何设计出符合 REST 风格的网络接口。

- **什么是RESTful**

> REST指的是一组架构约束条件和原则。满足这些约束条件和原则的应用程序或设计就是 RESTful。

RESTful的特性

- 资源（Resources）：

> 网络上的一个实体，或者说是网络上的一个具体信息。
> 它可以是一段文本、一张图片、一首歌曲、一种服务，总之就是一个具体的存在。
> 可以用一个 URI（统一资源定位符）指向它，每种资源对应一个特定的 URI 。要获取这个资源，访问它的 URI 就可以，因此 URI 即为每一个资源的独一无二的识别符。

- 表现层（Representation）：

> 把资源具体呈现出来的形式，叫做它的表现层 （Representation）。
> 比如，文本可以用 txt 格式表现，也可以用 HTML 格式、XML 格式、JSON 格式表现，甚至可以采用二进制格式。

- 状态转化（State Transfer）：

> 每发出一个HTTP请求，就代表了客户端和服务器的一次交互过程。HTTP协议，是一个无状态协议，即所有的【状态】都保存在服务器端。因此，如果客户端想要操作服务器， 必须通过某种手段，让服务器端发生“状态转化”（State Transfer）。而这种转化是建立在表现层之上的，所以就是 “ 表现层状态转化” 。
>
> 具体说， 就是 HTTP 协议里面，四个表示操作方式的动词：GET 、POST 、PUT 、DELETE 。它们分别对应四种基本操作：GET 用来获取资源，POST 用来新建资源，PUT 用来更新资源，DELETE 用来删除资源。

如何设计RESTful应用程序的API

> 路径设计：数据库设计完毕之后，基本上就可以确定有哪些资源要进行操作，相对于的路径也可以设计出来
> 动词设计：也就是针对资源的具体操作类型，由HTTP动词表示，常用的HTTP动词如下：POST、DELETE、PUT、GET

- **RESTful 的示例：**

```
/account/1  HTTP GET ：   得到 id = 1 的 account

/account/1  HTTP DELETE:  删除 id = 1 的 account

/account/1  HTTP PUT:     更新 id = 1 的 account
```

SpringMVC对RESTful的支持

RESTful的URL路径变量

- **URL-PATTERN**：设置为**/**，方便拦截`RESTful`请求。

```
<servlet-mapping>
    <servlet-name>DispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

- **@PathVariable**：可以解析出来URL中的**模板变量（{id}）**
  - URL:

```
http://localhost:8080/ssm/item/1/zhangsan
```

- Controller:

```
    @RequestMapping(“{id}/{name}”)
    @ResponseBody
    public Item queryItemById(@PathVariable Integer id, @PathVariable String
name){}
```

RESTful的CRUD

```
- @RequestMapping：通过设置method属性值，可以将同一个URL映射到不同的`HandlerMethod`方法上

- @GetMapping、@PostMapping、@PutMapping、@DeleteMapping注解同`@RequestMapping`注解的`method`属性设置。
```

RESTful的资源表述

```
RESTful服务中一个重要的特性就是一种资源可以有多种表现形式，在SpringMVC中可以使用ContentNegotiatingManager这个内容协商管理器来实现这种方式。
```

内容协商的方式有三种：

- **扩展名**：比如.json表示我要JSON格式数据、.xml表示我要XML格式数据
- **请求参数**：默认是”format”
- **请求头设置Accept参数**：比如设置Accept为application/json表示要JSON格式数据

```
不过现在RESTful响应的数据一般都是JSON格式，所以一般也不使用内容协商管理器，直接使用@ResponseBody注解将数据按照JSON格式返回。
```

静态资源访问

如果在DispatcherServlet中设置url-pattern为 /则必须对静态资源进行访问处理。

在**springmvc.xml**文件中，使用`mvc:resources`标签，具体如下：

```
<!-- 当DispatcherServlet配置为/来拦截请求的时候，需要配置静态资源的访问映射 -->
<mvc:resources location="/js/" mapping="/js/"/>
<mvc:resources location="/css/" mapping="/css/"/>
```

`SpringMVC`会把`mapping`映射到**ResourceHttpRequestHandler**，这样静态资源在经过`DispatcherServlet`转发时就可以找到对应的`Handler`了。

拦截器应用

SpringMVC的**拦截器**主要是针对**特定处理器**进行拦截的。

拦截器介绍

- SpringMVC拦截器（Interceptor）实现对每一个请求处理前后进行相关的业务处理，类似与servlet中的**Filter**。
- SpringMVC 中的Interceptor 拦截请求是通过HandlerInterceptor **接口**来实现的。

在SpringMVC中定义一个Interceptor非常简单，主要有4种方式：

1. 实现SpringMVC的HandlerInterceptor接口；
2. 继承实现了HandlerInterceptor接口的类，比如SpringMVC已经提供的实现了HandlerInterceptor 接口的抽象类HandlerInterceptorAdapter；
3. 实现SpringMVC的WebRequestInterceptor接口；
4. 继承实现了WebRequestInterceptor的类；

定义拦截器

实现HandlerIntercepter接口：

```
public class MyHandlerIntercepter implements HandlerInterceptor{

    //Handler执行前调用
    //应用场景：登录认证、身份授权
    //返回值为true则是放行，为false是不放行
    @Override
    public boolean preHandle(HttpServletRequest request,
            HttpServletResponse response, Object handler) throws Exception {

        return false;
    }

    //进入Handler开始执行，并且在返回ModelAndView之前调用
    //应用场景：对ModelAndView对象操作，可以把公共模型数据传到前台，可以统一指定视图
    @Override
    public void postHandle(HttpServletRequest request,
            HttpServletResponse response, Object handler,
            ModelAndView modelAndView) throws Exception {

    }
    //执行完Handler之后调用
    //应用场景：统一异常处理、统一日志处理
    @Override
    public void afterCompletion(HttpServletRequest request,
            HttpServletResponse response, Object handler, Exception ex)
            throws Exception {

    }
```

```
}
```

配置拦截器

SpringMVC 拦截器是绑定在 HandlerMapping 中的，即：如果某个 HandlerMapping 中配置拦截，则该 HandlerMapping 映射成功的 Handler 会使用该拦截器。

SpringMVC 的全局拦截器配置，其实是把配置的拦截器注入到每个已初始化的 HandlerMapping 中了。

```
<!-- 配置全局mapping的拦截器 -->
<mvc:interceptors>
    <!-- 公共拦截器可以拦截所有请求，而且可以有多个 -->
    <bean class="com.kkb.ssm.interceptor.MyHandlerInterceptor" />
    <bean class="com.kkb.ssm.interceptor.MyHandlerInterceptor2" />
    <!-- 如果有针对特定URL的拦截器，则进行以下配置 -->
    <mvc:interceptor>
        <!-- /**表示所有URL和子URL路径 -->
        <mvc:mapping path="/orders/**" />
         <!-- 特定请求的拦截器只能有一个 -->
        <bean class="com.kkb.ssm.interceptor.MyHandlerInterceptor3" />
    </mvc:interceptor>
</mvc:interceptors>
```

多拦截器拦截规则

如果有多个拦截器，那么配置到`springmvc.xml`中最上面的拦截器，拦截优先级最高。

拦截器应用（实现登录认证）

需求

拦截器对访问的请求 URL 进行登录拦截校验。

分析如下：

1、 如果请求的URL是公开地址（无需登录就可以访问的URL,具体指的就是保护login字段的请求URL），采取放行。

2、 如果用户session存在，则放行。

3、 不放行，都要跳转到登录页面。

登录页面

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>登录页面</title>
</head>
<body>
    <form action="${pageContext.request.contextPath }/login" method="post">
        <table align="center" border="1" cellspacing="0" >
            <tr>
                <td>用户名：<input type="text" name="username"/></td>
            </tr>
            <tr>
                <td>密　　码：<input type="text" name="password"/></td>
            </tr>
            <tr>
                <td><input type="submit" value="登录"/></td>
            </tr>
        </table>
    </form>
</body>
</html>
```

Controller类

```
@Controller
public class LoginController {
    // 登录
    @RequestMapping("/login")
    public String login(HttpSession session, String username, String password) {
        // Service进行用户身份验证

        // 把用户信息保存到session中
        session.setAttribute("username", username);

        // 重定向到商品列表页面
        return "redirect:/item/findItem";
    }

    // 退出
    @RequestMapping("/logout")
    public String logout(HttpSession session) {

        //清空session
        session.invalidate();
        // 重定向到登录页面
        return "redirect:/login.jsp";
    }
}
```

HandlerInterceptor类

```
public class LoginInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request,
            HttpServletResponse response, Object handler) throws Exception {

        //获取请求的URI
        String requestURI = request.getRequestURI();

        System.out.println(requestURI);
        //1、     如果请求的URL是公开地址（无需登录就可以访问的URL），采取放行。
        if(requestURI.indexOf("login")>-1) return true;
        //2、     如果用户session存在，则放行。
        String username = (String)
request.getSession().getAttribute("username");
        if(username !=null && !username.equals("")) return true;
        //3、     如果用户session中不存在，则跳转到登录页面。
        response.sendRedirect("/springmvc-demo/login.jsp");
        return false;
    }

    //其他代码略
}
```

HandlerInterceptor配置

```
<!-- 配置全局mapping的拦截器 -->
<mvc:interceptors>
    <!-- 拦截所有请求URL-->
    <bean class="com.kkb.ssm.interceptor.LoginInterceptor" />
</mvc:interceptors>
```

CORS跨域解决方案

什么是跨域

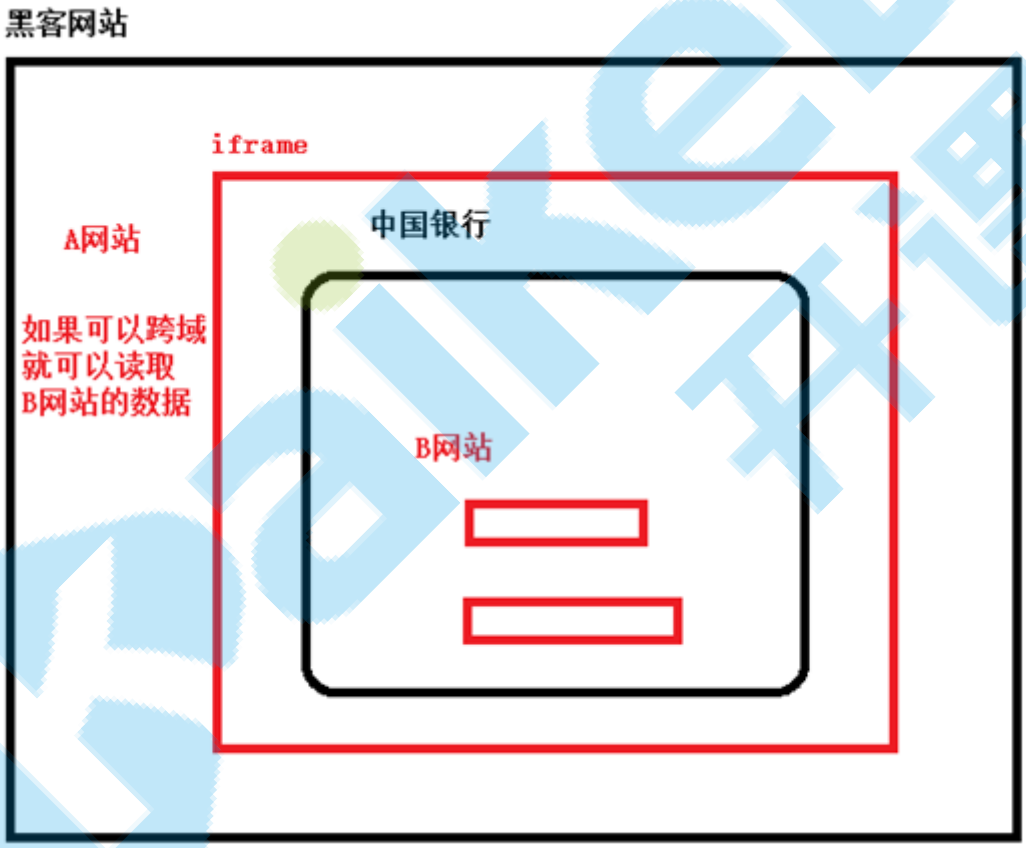
浏览器因为安全考虑，所以设置了同源策略。同源策略简单理解就是**DNS域名，端口号，协议**完全相同就称为同源。同源下的页面之间才能进行js的dom操作，如果不在同一个源下任何跨文档dom访问都是被阻止的。**不同源下的访问可以称之为跨域访问。**

下面表格里的a.js是无法获取b.js的内容的。

情况	举例
端口号不同	http://www.baidu.com/a.js vs. http://www.baidu.com:8080/b.js
主域相同子域不同	http://www.baidu.com/a.js vs. http://blog.baidu.com/b.js
协议不同	http://www.baidu.com/a.js vs. https://www.baidu.com/b.js
域名不同	http://www.baidu.com/a.js vs. http://www.qq.com/b.js
域名和对应ip	http://www.baidu.com/a.js vs. http://192.168.2.2/b.js

> 当然在实际应用中，多数出现在ajax请求时，在不同域下请求数据会遇到禁止跨域的问题。

为什么要有同源策略



如何解决跨域

解决跨域主要考虑两方面：

- 一个是避开`Ajax`请求方式；
- 一个是解决同源限制的问题。

解决跨域的方式有多种：

- 基于`JavaScript标签`的`src`方式

- 基于 `Jquery` 的 `JSONP` 方式
- 基于 `CORS` 的方式（解决同源的问题）

`JSONP` 和 `CORS` 的区别：

- `JSONP` 只能解决 `GET` 方式提交
- `CORS` 不仅支持 `GET` 方式，同时也支持 `POST` 提交方式。

我们重点就来讲解 `CORS` **跨域**方式。

什么是CORS

- `CORS` 是一个 `W3C` 标准，全称是"**跨域资源共享**"（`Cross-origin resource sharing`）。
- 它允许浏览器向跨源服务器，发出 `XMLHttpRequest` 请求，从而克服了 `AJAX` 只能同源使用的限制。
- `CORS` 需要**浏览器和服务器同时支持。**目前，所有浏览器都支持该功能，`IE` **浏览器不能低于** `IE10`。
- `CORS` 原理：
  - 客户端自动向**请求头** `header` 中注入 `Origin`。
  - 服务器端需要向**响应头** `header` 中注入 `Access-Control-Allow-Origin`
  - 浏览器检测到 `header` 中的 `Access-Control-Allow-Origin`，则就可以跨域操作了。

客户端跨域处理

请求分类标准

浏览器将 `CORS` 请求分成两类：**简单请求**（`simple request`）和**非简单请求**（`not-so-simple request`）。

只要同时满足以下两大条件，就属于**简单请求。**

凡是不同时满足上面两个条件，就属于**非简单请求。**

> 结论：浏览器对这两种请求的处理，是不一样的。

简单请求

对于简单请求，浏览器直接发出CORS请求。具体来说，就是在头信息之中，增加一个Origin字段。

- 请求信息：

```
GET /cors HTTP/1.1
Origin:
Host:
Accept-Language: en-US
Connection: keep-alive
User-Agent: Mozilla/5.0...
```

- 响应信息：

```
Access-Control-Allow-Origin:
Access-Control-Allow-Credentials: true
Access-Control-Expose-Headers: FooBar
Content-Type: text/html; charset=utf-8
```

- 字段说明

（1）Access-Control-Allow-Origin
该字段是必须的。它的值要么是请求时Origin字段的值，要么是一个*，表示接受任意域名的请求。

（2）Access-Control-Allow-Credentials
该字段可选。它的值是一个布尔值，表示是否允许发送Cookie。
默认情况下，Cookie不包括在CORS请求之中。如果
设为true，即表示服务器明确许可，Cookie可以包含在请求中，一起发给服务器。
这个值也只能设为true，如果服务器不要浏览器发送Cookie，删除该字段即可。

非简单请求

非简单请求是那种对服务器有特殊要求的请求，比如请求方法是PUT或DELETE，或者Content-Type字段的类型是application/json。

非简单请求的CORS请求，会在正式通信之前，增加一次HTTP查询请求，称为**"预检"请求（preflight）**。

浏览器先询问服务器，当前网页所在的域名是否在服务器的许可名单之中，以及可以使用哪些HTTP动词和头信息字段。只有得到肯定答复，浏览器才会发出正式的XMLHttpRequest请求，否则就报错。

- **请求信息：**

HTTP请求的方法是PUT，并且发送一个自定义头信息X-Custom-Header。

浏览器发现，这是一个非简单请求，就自动发出一个"预检"请求，要求服务器确认可以这样请求。下面是这个"预检"请求的HTTP头信息。

```
OPTIONS /cors HTTP/1.1
Origin: [redacted]
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: X-Custom-Header
Host: [redacted]
Accept-Language: en-US
Connection: keep-alive
User-Agent: Mozilla/5.0...
```

"预检"请求用的请求方法是`OPTIONS`，表示这个请求是用来询问的。头信息里面，关键字段是`Origin`，表示请求来自哪个源。

除了`Origin`字段，"预检"请求的头信息包括两个特殊字段。

```
（1）Access-Control-Request-Method
    该字段是必须的，用来列出浏览器的CORS请求会用到哪些HTTP方法，上例是PUT。

（2）Access-Control-Request-Headers
    该字段是一个逗号分隔的字符串，指定浏览器CORS请求会额外发送的头信息字段，上例是X-Custom-Header。
```

一旦服务器通过了"预检"请求，以后每次浏览器正常的`CORS`请求，就都跟简单请求一样，会有一个`Origin`头信息字段。服务器的回应，也都会有一个`Access-Control-Allow-Origin`头信息字段。

服务器端跨域处理

CORS实现1

`springmvc4.x`以下，使用`springmvc`**的拦截器**实现

跨域不提交Cookie

```
public class AllowOriginInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object arg2) throws Exception {
        // 有跨域行为时参考网址 http://namezhou.iteye.com/blog/2384434
        if (request.getHeader("Origin") != null) {
            response.setContentType("text/html;charset=UTF-8");
            // 允许哪一个URL
            response.setHeader("Access-Control-Allow-Origin", "*");
            // 允许那种请求方法
            response.setHeader("Access-Control-Allow-Methods", "POST, GET, OPTIONS, DELETE");
            response.setHeader("XDomainRequestAllowed", "1");
            System.out.println("正在跨域");
        }
```

```
        return true;
    }
}
```

配置拦截器的代码，请自行补充！！

跨域提交Cookie

- 注意事项

  `Access-Control-Allow-Credentials` 为 `true` 的时候，`Access-Control-Allow-Origin` 一定不能设置为”*”，否则报错。

  如果有多个拦截器，一定要把处理跨域请求的拦截器放到首位。

- JS代码
  - Jquery Ajax

```
$.ajax({
    url: '自己要请求的url',
    method:'请求方式',  //GET POST PUT DELETE
    xhrFields:{
        withCredentials:true
    },
    success:function(data){
       //自定义请求成功做什么
    },
    error:function(){
    //自定义请求失败做什么
    }
})
```

- angularJS

1. 全局 在模块配置中添加

```
app.config(['$[Math Processing Error]httpProvider',function($httpProvider) {

  $httpProvider.defaults.withCredentials = true;

}

]);
```

1. 单个请求

```
$http.get(url, {withCredentials: true});

$http.post(url,data, {withCredentials: true});

$httpProvider.defaults.withCredentials = true;
```

- Java代码

```
public class AllowOriginInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object arg2) throws Exception {
       // 有跨域行为时参考网址 http://namezhou.iteye.com/blog/2384434
       if (request.getHeader("Origin") != null) {
            response.setContentType("text/html;charset=UTF-8");
            // 允许哪一个URL 访问 request.getHeader("Origin") 根据请求来的url动态允许
           response.setHeader("Access-Control-Allow-Origin",
request.getHeader("Origin"));
            // 允许那种请求方法
           response.setHeader("Access-Control-Allow-Methods", "POST, GET,
OPTIONS, DELETE,HEAD");
            response.setHeader("Access-Control-Max-Age", "0");
            // 允许请求头里的参数列表
            response.setHeader("Access-Control-Allow-Headers",
                    "Origin, No-Cache, X-Requested-With, If-Modified-Since,
Pragma, Last-Modified, Cache-Control, Expires, Content-Type, X-E4M-
With,userId,token");
            // 允许对方带cookie访问
     response.setHeader("Access-Control-Allow-Credentials", "true");
            response.setHeader("XDomainRequestAllowed", "1");
            System.out.println("正在跨域");
        }
        return true;
    }
}
```

CORS实现2

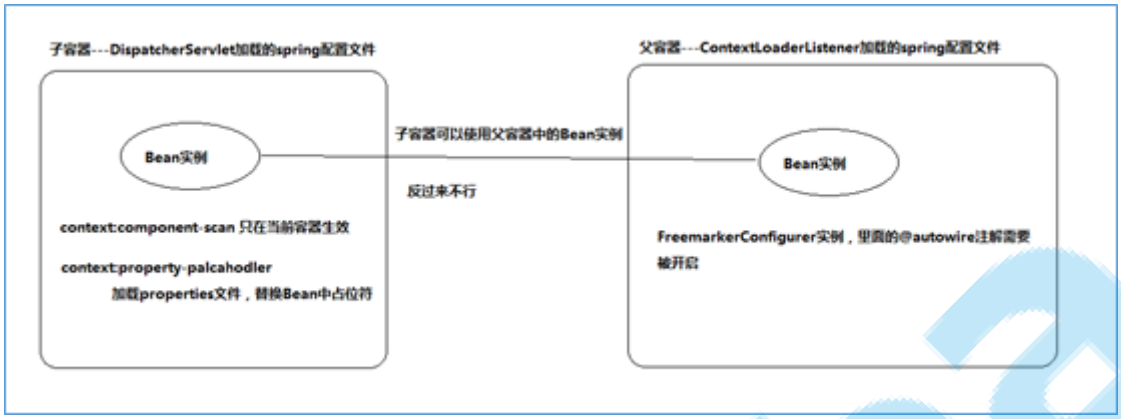
`SpringMVC4.x` 以上处理 `CORS` 跨域方式。

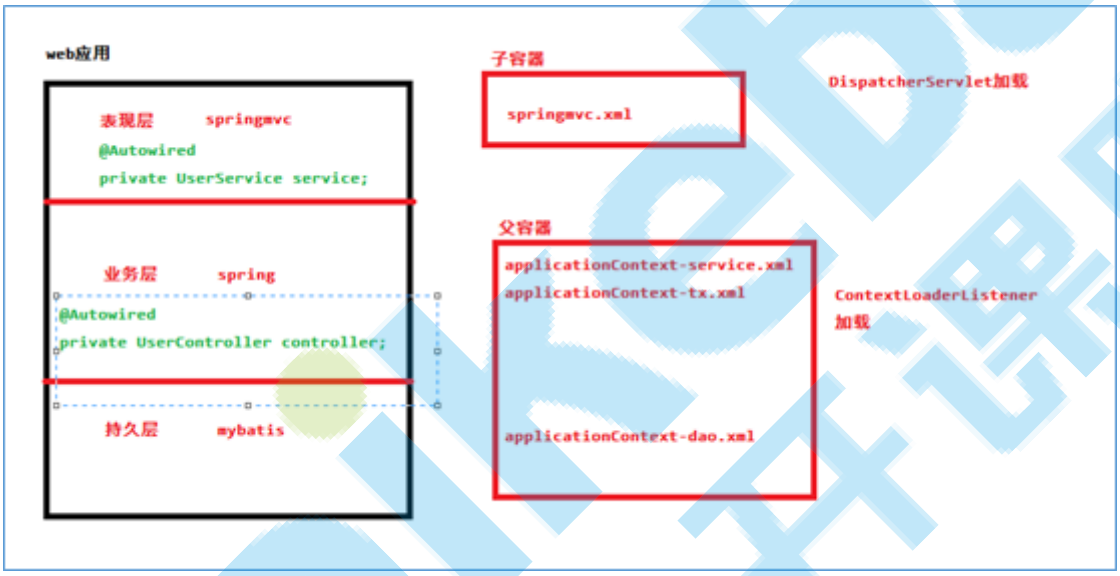
- 某个方法可以跨域访问，在某个方法上使用 `@CrossOrigin`
- 某个 `Controller` 类都可以跨域访问，在类上上使用 `@CrossOrigin`

全局访问，在 `springmvc.xml` 中配置

```
<mvc:cors>
    <mvc:mapping path="/**"/>
</mvc:cors>
```

父子容器





- **问题1**：在子容器声明一个Bean，然后父容器中使用`@Autowired`注解注入，验证注入是否成功。
- **问题2**：`@Autowired`注解的开启方式有哪些？如果父容器中不开启`@Autowired`注解，是如何操作?

应用了解篇

Mock测试（模拟测试）

什么是mock测试

> 在单元测试过程中，对于某些不容易构造或者不容易获取的对象，用一个**虚拟的对象**来创建以便测试的测试方法，就是`mock`**测试**。

> 这个**虚拟的对象**就是`mock`**对象**。`mock`**对象**就是真实对象在调试期间的**代替品**。

> 使用Mock Object进行测试，主要是用来模拟那些在应用中不容易构造（如HttpServletRequest必须在Servlet容器中才能构造出来）或者比较复杂的对象（如JDBC中的ResultSet对象）从而使测试顺利进行的工具。

> 目前，在Java阵营中主要的Mock测试工具有JMock，MockCreator，Mockrunner，EasyMock，MockMaker等，在微软的.Net阵营中主要是Nmock，.NetMock等。

为什么使用mock测试

- 避免开发模块之间的耦合
- 轻量、简单、灵活

MockMVC介绍

基于RESTful风格的SpringMVC单元测试，我们可以测试完整的Spring MVC流程，即**从URL请求到控制器处理，再到视图渲染都可以测试。**

MockMvc

- 对于服务器端的Spring MVC测试支持**主入口点。**
- 通过MockMVCBuilder构造
- MockMVCBuilder由MockMVCBuilders建造者的静态方法去建造。
- 核心方法：**perform**(RequestBuilder rb)--- 执行一个RequestBuilder请求，会自动执行SpringMVC的流程并映射到相应的控制器执行处理，该方法的返回值是一个ResultActions；

MockMVCBuilder

- MockMvcBuilder是使用构建者模式来构造MockMvc的构造器
- 其主要有两个实现：**StandaloneMockMvcBuilder**和**DefaultMockMvcBuilder**，分别对应之前的两种测试方式。
- 不过我们可以直接使用**静态工厂MockMvcBuilders创建**即可，不需要直接使用上面两个实现类。

MockMVCBuilders

- 负责创建MockMVCBuilder对象
- 有两种创建方式
  - standaloneSetup(Object... controllers):

    通过参数指定一组控制器，这样就不需要从上下文获取了。
  - **webAppContextSetup**(WebApplicationContext wac)

    指定WebApplicationContext，将会从该上下文获取相应的控制器并得到相应的MockMvc

MockMvcRequestBuilders

- 用来构建Request请求的
- 其主要有两个子类**MockHttpServletRequestBuilder**和**MockMultipartHttpServletRequestBuilder**（如文件上传使用），即用来Mock客户端请求需要的所有数据。

ResultActions

- **andExpect**：添加ResultMatcher验证规则，验证控制器执行完成后结果是否正确；
- **andDo**：添加ResultHandler结果处理器，比如调试时打印结果到控制台；
- **andReturn**：最后返回相应的**MvcResult**；然后进行自定义验证/进行下一步的异步处理；

MockMvcResultMatchers

- 用来匹配执行完请求后的**结果验证**
- 如果匹配失败将抛出相应的异常
- 包含了很多验证API方法

MockMvcResultHandlers

- 结果处理器，表示要对结果做点什么事情
- 比如此处使用MockMvcResultHandlers.print()输出整个响应结果信息。

MvcResult

- 单元测试执行结果，可以针对执行结果进行**自定义验证逻辑。**

MockMVC使用

添加依赖

```
<!-- spring 单元测试组件包 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>5.0.7.RELEASE</version>
        </dependency>

        <!-- 单元测试Junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>

        <!-- Mock测试使用的json-path依赖 -->
        <dependency>
            <groupId>com.jayway.jsonpath</groupId>
            <artifactId>json-path</artifactId>
            <version>2.2.0</version>
        </dependency>
```

测试类

```
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
```

```
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.MediaType;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.MvcResult;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;
import org.springframework.test.web.servlet.result.MockMvcResultHandlers;
import org.springframework.test.web.servlet.result.MockMvcResultMatchers;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;
//@WebAppConfiguration：可以在单元测试的时候，不用启动Servlet容器，就可以获取一个Web应用上
下文

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:spring/*.xml")
@WebAppConfiguration
public class TestMockMVC {

    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        // 初始化一个MockMVC对象的方式有两种：单独设置、web应用上下文设置
        // 建议使用Web应用上下文设置
        mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();
    }

    @Test
    public void test() throws Exception {
        // 通过perform去发送一个HTTP请求
        // andExpect：通过该方法，判断请求执行是否成功
        // andDo ：对请求之后的结果进行输出
        MvcResult result =
mockMvc.perform(MockMvcRequestBuilders.get("/item/showEdit").param("id", "1"))
                .andExpect(MockMvcResultMatchers.view().name("item/item-edit"))
                .andExpect(MockMvcResultMatchers.status().isOk())
                .andDo(MockMvcResultHandlers.print())
                .andReturn();

        System.out.println("==============================");
        System.out.println(result.getHandler());
    }
```

```
    @Test
    public void test2() throws Exception {
        // 通过perform去发送一个HTTP请求
        // andExpect：通过该方法，判断请求执行是否成功
        // andDo ：对请求之后的结果进行输出
        MvcResult result = mockMvc.perform(get("/item/findItem").param("id",
"1").accept(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())

.andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$.id").value(1))
                .andExpect(jsonPath("$.name").value("台式机123"))
                .andDo(print())
                .andReturn();

        System.out.println("================================");
        System.out.println(result.getHandler());
    }
}
```

- **@WebAppConfiguration**

用于声明一个ApplicationContext集成测试加载WebApplicationContext。

ControllerAdvice

@ControllerAdvice

介绍

- 该注解顾名思义是一个增强器，是对注解了@Controller注解的类进行增强。

- 该注解使用@Component注解，这样的话当我们使用<context:component-scan>扫描时也能扫描到。

- 该注解内部使用@ExceptionHandler、@InitBinder、@ModelAttribute注解的方法会应用到所有的Controller类中 @RequestMapping注解的方法。

使用

```
@ControllerAdvice
public class MyControllerAdvice {

    //应用到所有@RequestMapping注解方法，在其执行之前把返回值放入ModelMap中
    @ModelAttribute
```

```
    public Map<String, Object> ma(){
        Map<String, Object> map = new HashMap<>();
        map.put("name", "tom");
        return map;
    }

    //应用到所有【带参数】的@RequestMapping注解方法，在其执行之前初始化数据绑定器
    @InitBinder
    public void initBinder(WebDataBinder dataBinder) {
         DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dataBinder.registerCustomEditor(Date.class,
                                    new CustomDateEditor(dateFormat,true));
        System.out.println("...initBinder...");
    }

    //应用到所有@RequestMapping注解的方法，在其抛出指定异常时执行
    @ExceptionHandler(Exception.class)
    @ResponseBody
    public String handleException(Exception e) {
        return e.getMessage();
    }
}
```

@ModelAttribute

就是将数据放入Model中，然后可以在页面中展示该数据。

介绍

- **该注解特点**：主要作用于ModelMap这个模型对象的，用于在视图中显示数据。
- **该注解注意事项**：和@ResponseBody注解的使用是互斥的。

作用于方法(常用)

有没有`@RequestMapping`注解的方法都可以。

无@RequestMapping

执行时机：在本类内所有`@RequestMapping`注解方法之前执行。

```
@Controller
public class ModelAttributeController {

    // 如果方法有返回值，则直接将该返回值放入`ModelMap`中，`key`可以指定
    // @ModelAttribute
    @ModelAttribute(value="myUser")
    public User populateModel() {
       User user=new User();
       user.setAccount("ray");
       return user;
    }
```

```
    // 如果方法没有返回值，则可以利用它的执行时机这一特点，做一些预处理。
    @ModelAttribute
    public void populateModel(@RequestParam String abc, Model model) {
       model.addAttribute("attributeName", abc);
    }

    @RequestMapping(value = "/hello")
    public String hello() {
        // 逻辑视图名称
        return "hello";
    }
}
```

有@RequestMapping

```
@Controller
public class ModelAttributeController {

    // 逻辑视图名称根据请求`URL`生成，比如URL：`item/findItem`，那么这个URL就是逻辑视图名称
    @RequestMapping(value = "/hello")
    @ModelAttribute
    public String hello() {
        //返回值会放入ModelMap中,key为返回值类型的首字母小写
        return "hello";
    }

}
```

作用于方法参数

作用于有`@RequestMapping`注解的方法：

- `STEP1`：获取`ModelMap`中指定的数据（由`@ModelAttribute`注解的方法产生）
- `STEP2`：将使用该注解的参数对象放到`ModelMap`中。
- 如果`STEP1`和`STEP2`中的对象在`ModelMap`中`key`相同，则合并这两部分对象的数据。

```
//对象合并使用方式，比如一个用户的大部分信息都是从数据库获取出来的，而表单过来的数据只有一个，此时使用对象合并就会方便很多
@Controller
public class ModelAttributeController {

    @ModelAttribute("myUser")
    public User populateModel() {
       User user=new User();
       user.setAccount("ray");
       return user;
    }

    @RequestMapping(value = "/hello")
    public String hello(@ModelAttribute("myUser") User user) {
       user.setName("老王");
```

```
        return "hello";
    }
}
```

@InitBinder

介绍

由 `@InitBinder` 标识的方法，可以通过 `PropertyEditor` 解决数据类型转换问题，比如 `String-->Date` 类型。

不过 `PropertyEditor` 只能完成 `String-->任意类型` 的转换，这一点不如 `Converter` 灵活，`Converter` 可以完成 `任意类型-->任意类型` 的转换。

它可以对 `WebDataBinder` 对象进行初始化，`WebDataBinder` 是 `DataBinder` 的子类,用于完成由**表单字段到 `JavaBean` 属性的绑定。**

注意事项：

- `@InitBinder` 方法**不能有返回值**,它必须声明为 `void`。
- `@InitBinder` 方法的**参数通常是 `WebDataBinder`**。

使用

课堂补充

@ExceptionHandler

介绍

这个注解表示 `Controller` 中任何一个 `@RequestMapping` 方法发生异常，则会被注解了 `@ExceptionHandler` 的方法拦截到。

拦截到对应的异常类则执行对应的方法，如果都没有匹配到异常类，则采用近亲匹配的方式。

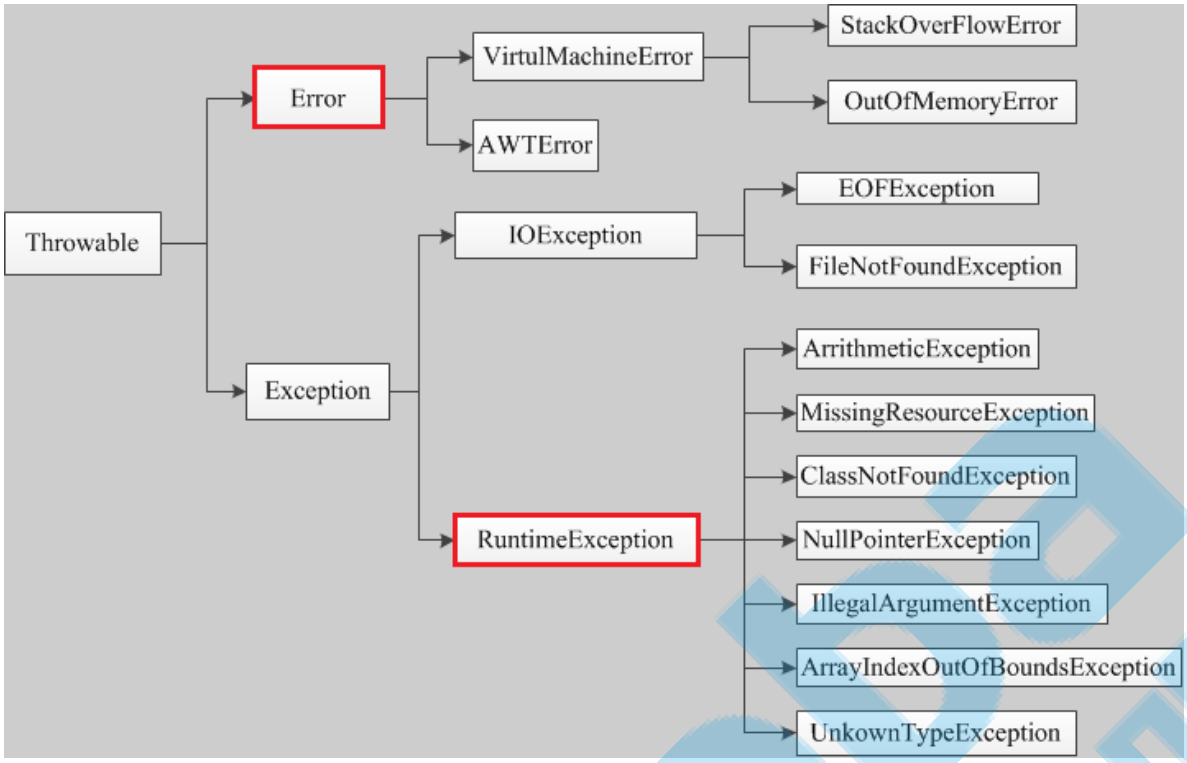
使用

课堂补充

异常处理器

异常处理器（`HandlerExceptionResolver`）：`SpringMVC` 在处理请求过程中出现的异常信息交由**异常处理器**进行处理，异常处理器可以实现针对一个应用系统中出现的异常进行相应的处理。

异常理解



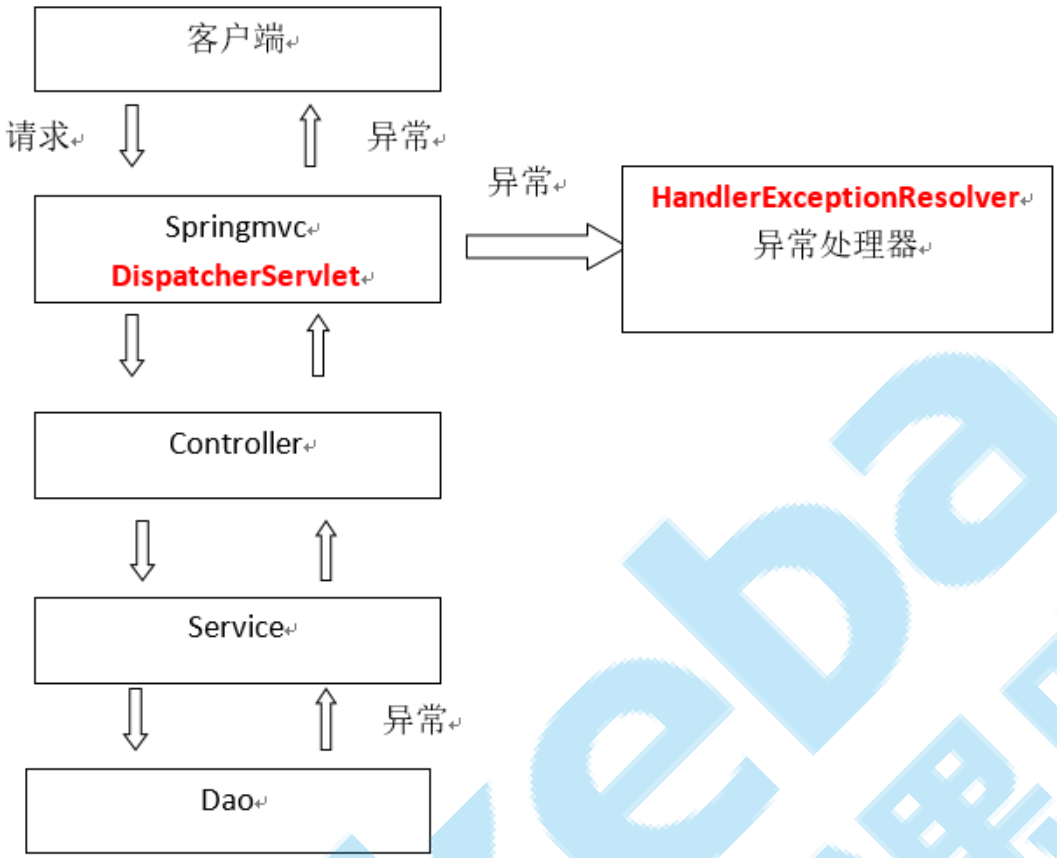
我们通常说的异常包含**编译时异常（已检查异常、预期异常）**和**运行时异常（未检查异常）**。

- 继承 `RuntimeException` 类的异常类，就是**运行时异常。**
- 继承 `Exception` 类的异常类，没有继承 `RuntimeException` 类的异常类，就是**编译时异常。**

常见异常举例：

- **运行时异常**，比如：**空指针异常、数组越界异常。**对于这样的异常，只能通过程序员丰富的经验来解决和测试人员不断的严格测试来解决。
- **编译时异常**，比如：**数据库异常、文件读取异常、自定义异常等。**对于这样的异常，必须使用 `try catch` 代码块或者 `throws` 关键字来处理异常。

异常处理思路



异常处理器演示

自定义异常类

```
public class BusinessException extends Exception {

    private static final long serialVersionUID = 1L;

    //异常信息
    private String message;

    public BusinessException(String message) {
        super(message);
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

异常处理器实现

```java
public class BusinessExceptionResolver implements HandlerExceptionResolver {

    @Override
    public ModelAndView resolveException(HttpServletRequest request,
            HttpServletResponse response, Object handler, Exception ex) {

        //自定义预期异常
        BusinessException businessException = null;
        //如果抛出的是系统自定义的异常
        if(ex instanceof BusinessException){
            businessException = (BusinessException) ex;
        }else{
            businessException = new BusinessException("未知错误");
        }

        ModelAndView modelAndView = new ModelAndView();
        //把错误信息传递到页面
        modelAndView.addObject("message", businessException.getMessage());
        //指向错误页面
        modelAndView.setViewName("error");
        return modelAndView;
    }
}
```

异常处理器配置

在 `springmvc.xml` 中加入以下代码:

```xml
<!-- 自定义异常处理器（全局） -->
<bean class="com.kkb.ssm.resolver.BusinessExceptionResolver"/>
```

错误页面

```jsp
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>错误信息</title>
</head>
<body>
${message }
</body>
</html>
```

异常测试

```
@RequestMapping(value = "/showItemEdit")
@ResponseBody
public String showItemEdit(Integer id) throws Exception{

    // 查询要显示的商品内容
    Item item = itemService.queryItemById(id);

    if(item == null) throw new BusinessException("查询不到商品无法修改");

    return item;
}
```

乱码解决

GET请求乱码

原因分析

- `GET`**请求参数是通过请求首行中的** `URI` 发送给Web服务器（`Tomcat`）的。
- `Tomcat` 服务器会对 `URI` 进行编码操作（**此时使用的是** `Tomcat` **设置的字符集，默认是** `iso8859-1`）
- 到了我们的应用程序中的请求参数，已经是被 `Tomcat` 使用 `ISO8859-1` 字符集进行编码之后的了。

解决方式1

修改tomcat配置文件，指定UTF-8编码，如下：

```
<Connector URIEncoding="utf-8" connectionTimeout="20000" port="8080"
protocol="HTTP/1.1" redirectPort="8443"/>
```

解决方式2

对请求参数进行重新编码，如下：

```
String username = request.getParamter("userName");
username = new String(username.getBytes("ISO8859-1"),"utf-8") ;
```

解决方式3

过滤器+请求装饰器统一解决请求乱码

- MyRequestWrapper
- MyCharacterEncodingFilter

POST请求乱码

在`web.xml`中加入：

```
<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>
```

响应乱码

使用`@RequestMapping`注解中的`produces`属性，指定响应体中的编码格式。

```
    // @RequestMapping注解中的consumes和produces分别是为请求头和响应头设置contentType
    @RequestMapping(value = "findUserById", produces = "text/plain;charset=UTF-
8")
    @ResponseBody
    public String findUserById(Integer id) {
        // 在使用@ResponseBody注解的前提下
        // 如果返回值是String类型，则返回值会由StringHttpMessageConverter进行处理
        return "查询失败";
    }
```

非注解开发方式

处理器开发

实现`HttpRequestHandler`接口

```
//第二种Handler处理器的编写方式：实现HttpRequestHandler接口
public class DemoController implements HttpRequestHandler {

    @Override
    public void handleRequest(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

        //数据填充
        request.setAttribute("msg","实现HttpRequestHandler接口的开发方式" );

        request.getRequestDispatcher("/WEB-INF/jsp/msg.jsp").forward(request,
response);
    }
}
```

处理器配置

```
<!-- 配置处理器 -->
<bean class="com.kkb.springmvc.controller.DemoController"/>
```

配置处理器映射器

如果在DispatcherServlet.properties文件中出现的处理器适配器和处理器映射器，如果没有特殊要求（比如在适配器中配置转换器、在映射器中配置拦截器），那么就不需要再单独进行配置了。

BeanNameUrlHandlerMapping

该处理器映射器的映射思路：

- 解析 bean 标签中的 name 属性值（**请求 URL**），作为 key。
- 解析 bean 标签中的 class 属性值，作为 value。

根据映射器的映射规则，在 springmvc.xml 中**配置请求 URL 和处理器之间的映射关系**

```
<!-- 配置处理器 -->
<bean name="/helloHandler class="com.kkb.springmvc.controller.DemoController"/>
```

配置处理器适配器

HttpRequestHandlerAdapter

```
public class HttpRequestHandlerAdapter implements HandlerAdapter {

    @Override
    public boolean supports(Object handler) {
        return (handler instanceof HttpRequestHandler);
    }
```

```
    @Override
    @Nullable
    public ModelAndView handle(HttpServletRequest request, HttpServletResponse
response, Object handler)
            throws Exception {

        ((HttpRequestHandler) handler).handleRequest(request, response);
        return null;
    }
}
```

```
<!-- 配置http请求处理器适配器 -->
<bean class="org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter">
</bean>
```

SimpleControllerHandlerAdapter

```
public class SimpleControllerHandlerAdapter implements HandlerAdapter {

    @Override
    public boolean supports(Object handler) {
        return (handler instanceof Controller);
    }

    @Override
    @Nullable
    public ModelAndView handle(HttpServletRequest request, HttpServletResponse
response, Object handler)
            throws Exception {

        return ((Controller) handler).handleRequest(request, response);
    }
}
```

```
<!-- 配置处理器适配器 ，所有适配器都实现HandlerAdapter接口 -->
<bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"
/>
```

注意事项

- 处理器映射器和处理器适配器可以同时存在多个。
- 非注解方式开发的处理器只能使用非注解的映射器和适配器进行处理。