

1.Spring MVC 简介

Spring MVC 是 Spring 基于 MVC 设计理念提供的一个表现层的 Web 框架。是目前主流的 MVC 框架之一。

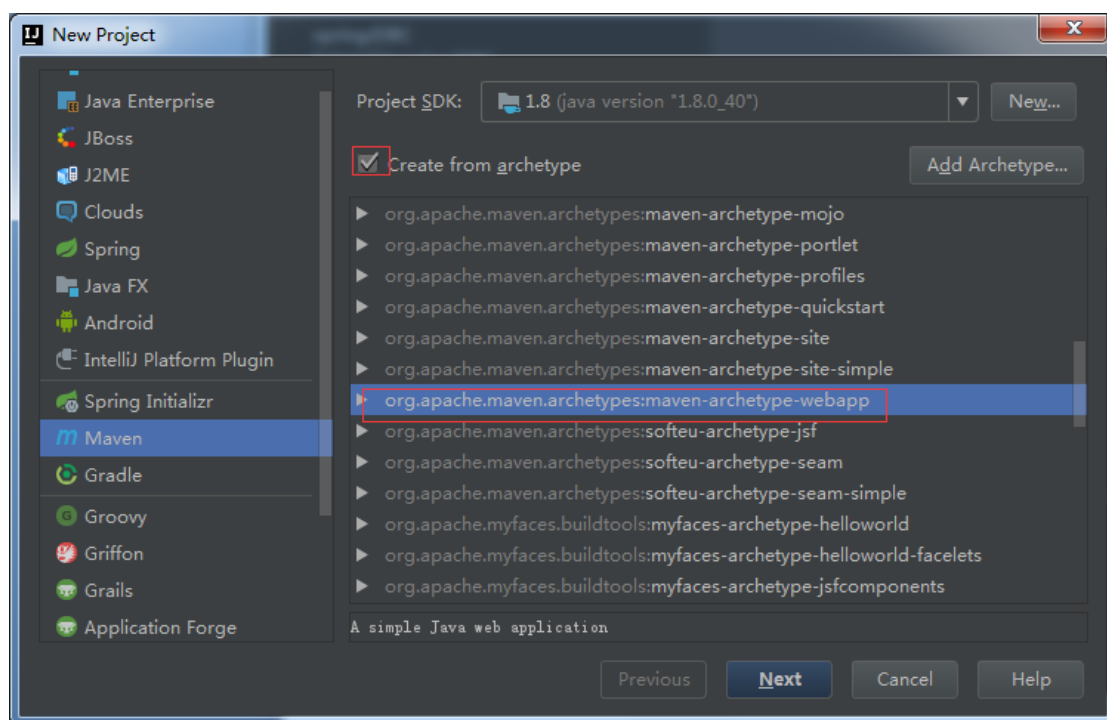
Spring MVC 通过一套 MVC 注解，让 pojo 成为处理请求的控制器，无需实现任何接口。比其他 MVC 框架更具扩展性和灵活性。

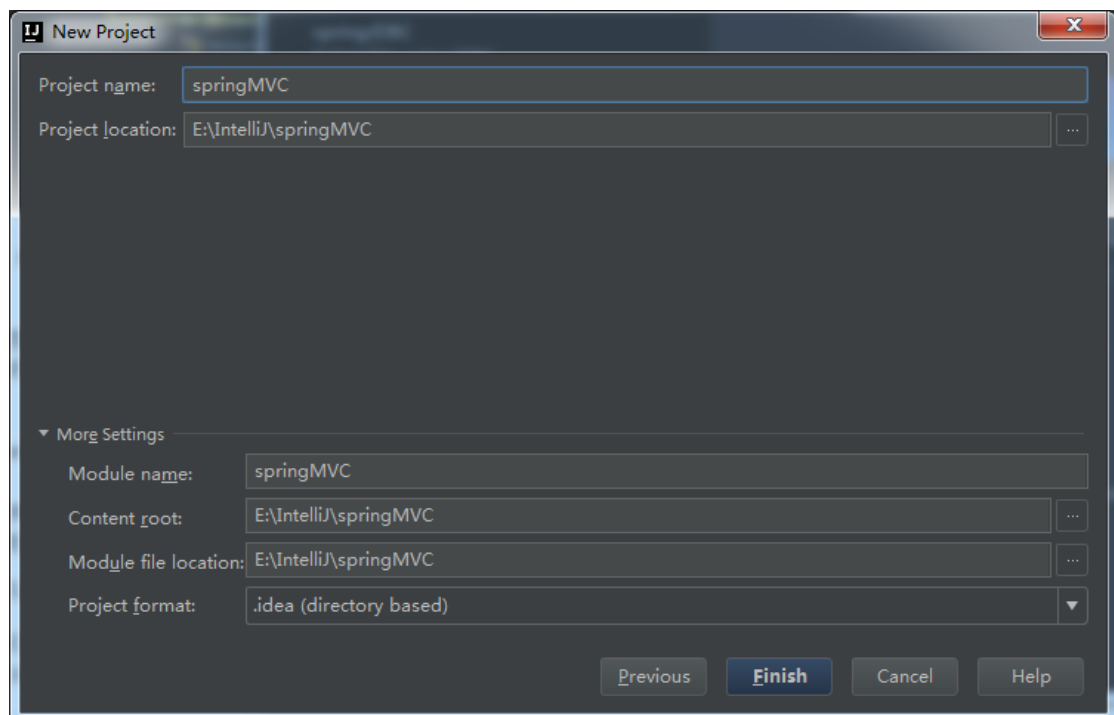
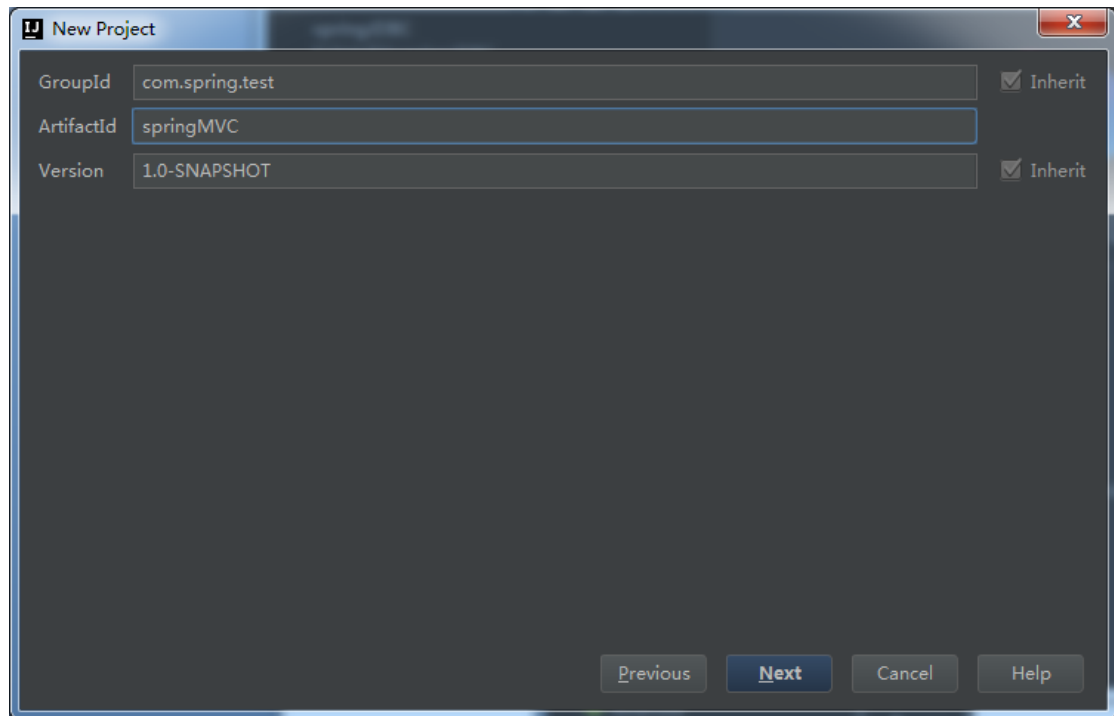
2.环境搭建

本节介绍一个简单的 springMVC 环境搭建，输出一个 Hello World。

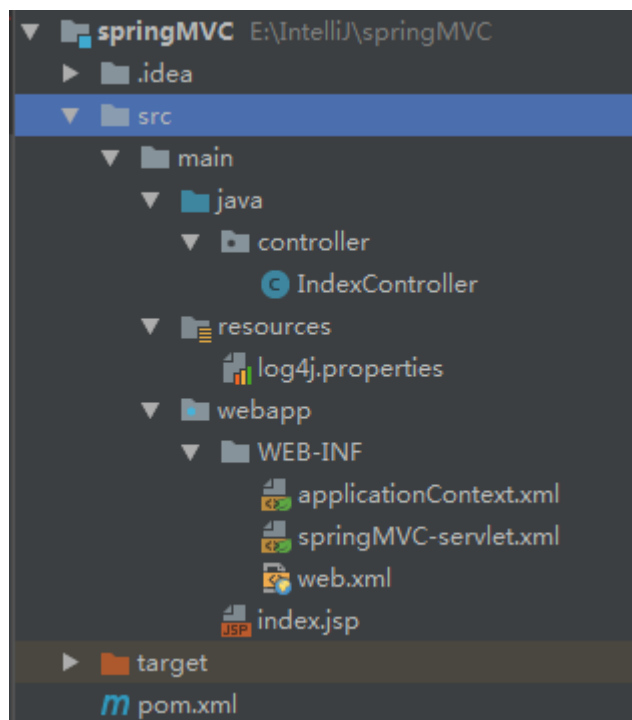
2.1 创建工程

选用一个 webapp 模板，这个模板会帮我们创建好 webapp/WEB-INF 目录和 web.xml 文件。如果不使用模板，也可以自己创建目录和文件。





本次工程的完整工程目录如下：



2.2 导入 jar 包

在 pom.xml 中加入配置：

```
<dependencies>
  <!--日志文件-->
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.6.1</version>
  </dependency>
  <!--spring 相关包-->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>4.3.11.RELEASE</version>
  </dependency>
</dependencies>
```

在 resources 目录下创建 log4j.properties 文件，配置如下：

```
log4j.rootLogger=DEBUG, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - %m%n
```

2.3 编写 Spring 配置文件

在 WEB-INF 中创建 springMVC-servlet.xml 和 applicationContext.xml 两个文件。IDEA 可以使用模板自动创建文件，过程参照文档《Spring 基础入门.docx》，Add Framework Support 的时候选择 SpringMVC 即可。

springMVC-servlet.xml 中配置如下：

```
<?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-4.2.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-4.2.xsd
           http://www.springframework.org/schema/mvc
           http://www.springframework.org/schema/mvc/spring-mvc-4.2.xsd">
    <!-- 启动注解，注册服务 -->
    <mvc:annotation-driven/>

    <!-- 启动自动扫描 -->
    <context:component-scan base-package="controller">
        <!-- 制定扫描规则，只扫描使用@Controller 注解的 JAVA 类 -->
        <context:include-filter type="annotation"
            expression="org.springframework.stereotype.Controller"/>
    </context:component-scan>
</beans>
```

applicationContext.xml 暂时先不用写东西。

2.4 配置 web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
             http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">
    <!--配置 listener，在启动 Web 容器的时候加载 Spring 的配置-->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
```

```

<!--将欢迎页设置成 index.html-->
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
</welcome-file-list>

<!--配置 DispatcherServlet -->
<servlet>
    <servlet-name>springMVC</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>springMVC</servlet-name>
    <url-pattern>*.html</url-pattern>
</servlet-mapping>
</web-app>

```

web 项目启动的时候，加载 spring 的默认路径是"/WEB-INF/applicationContext.xml，在 WEB-INF 目录下创建的 xml 文件的名称必须是 applicationContext.xml。如果是要自定义文件名可以在 web.xml 里加入 contextConfigLocation 这个 context 参数，如下：

```

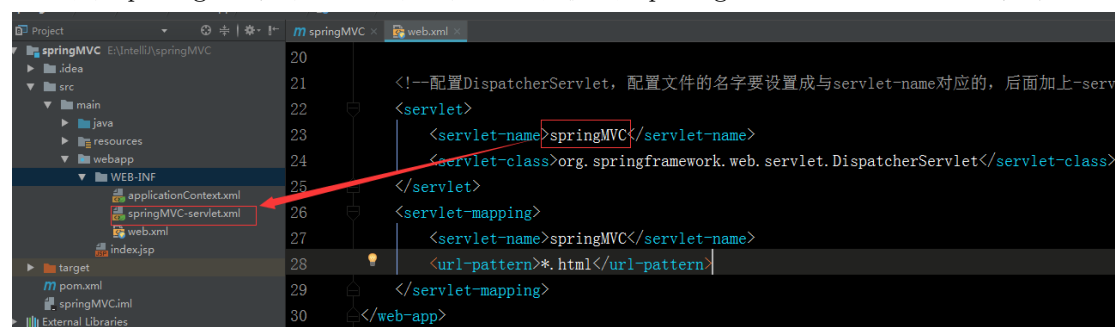
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:spring-*.xml</param-value>
</context-param>

```

上面的配置可以加载 resources 文件夹下 spring-开头的 xml 文件。

DispatcherServlet 在 SpringMVC 中负责流程控制和职责分派，如文件上传、请求映射等。

其中<servlet-name>中的值是和配置文件相对应的，如此处<servlet-name>的值是 springMVC，对应配置文件的名字就叫 springMVC-servlet.xml，如图：



如果想要使用其它文件名，可以通过配置 contextConfigLocation，如下：

```

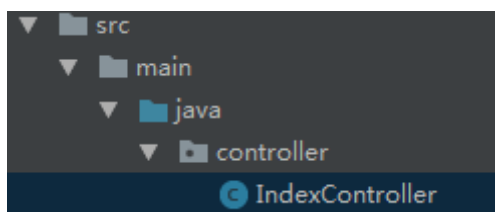
<servlet>
    <servlet-name>springMVC</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!--指定其他文件名-->
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>WEB-INF/applicationContext.xml</param-value>
    </init-param>

```

```
</init-param>  
</servlet>
```

2.5 编写 controller

在 java 文件夹下新建 controller 文件夹，创建 IndexController.java



```
package controller;  
  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;  
  
@Controller  
public class IndexController {  
    //RequestMapping 设置访问路径是 index.html  
    @RequestMapping("index.html")  
    public String showIndex() {  
        //返回 index.jsp 页面  
        return "index.jsp";  
    }  
}
```

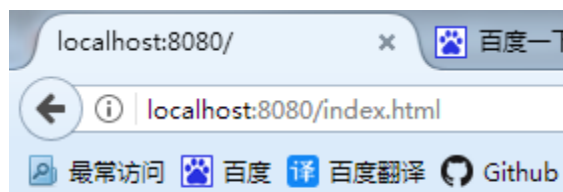
SpringMVC 中使用@Controller 注解声明控制器，所有接受请求的方法都要写在@Controller 注解的类中。@RequestMapping 用来设置请求的 url 地址。

2.6 页面 index.jsp

默认工程创建的时候在 webapp 下就包含 index.jsp:

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

运行项目。访问 <http://localhost:8080/index.html>



Hello World!

注意访问的是 index.html 不是 index.jsp。

Controller 类

```
@Controller
public class IndexController {
    //RequestMapping设置访问路径是index.html
    @RequestMapping(value = "/index.html")
    public String showIndex() {
        //返回index.jsp页面
        return "/index.jsp";
    }
}
```

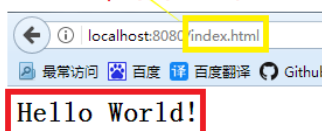
页面index.jsp

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

web.xml中的配置，
匹配所有.html结尾的url

```
<servlet-mapping>
    <servlet-name>springMVC</servlet-name>
    <url-pattern>*.html</url-pattern>
</servlet-mapping>
```

浏览器中看到的结果



3.使用@RequestMapping 映射请求

3.1 在类和方法上添加@RequestMapping

SpringMVC 使用@RequestMapping 为控制器指定可以处理哪些 URL 请求。可以在类上和方法上都添加@RequestMapping 注解。

如在 IndexController 中添加注解：

```
@Controller
//加上这个注解，本类方法访问路径前都要加上/mvc
@RequestMapping("/mvc")
public class IndexController {
    //RequestMapping 设置访问路径是 index.html
    //因为类名上面已经加了/mvc，所以进入这个方法的 url 是/mvc/index.html
    @RequestMapping("/index.html")
    public String showIndex() {
        //因为添加了父路径，而 index.jsp 在根路径，所以前面加/代表相对于根路径
        return "/index.jsp";
    }
}
```

```

    }
}

```

IndexController 类上添加了 `@RequestMapping("/mvc")`，相当于一个父路径，即 IndexController 类中所有 `@RequestMapping` 注解的方法访问路径前都要添加上这个路径。想进入 `showIndex()` 方法，路径是 `/mvc/index.html`。

`@RequestMapping("/mvc")` 相当于 `@RequestMapping(value="/mvc")`

DispatcherServlet 截获请求后，就通过 `@RequestMapping` 提供的信息，将请求进行分发。

应用场景示例：

部门Controller

```

@Controller
@RequestMapping("/dept")
public class DeptController {
    //访问路径是/dept/list.html
    @RequestMapping("/list.html")
    public String list() {
        //省略查询部门列表的代码
        return "/dept_list.jsp";
    }

    //访问路径是/dept/update.html
    @RequestMapping("/update.html")
    public String update(Department dept) {
        //省略修改部门代码
        return "/dept_list.jsp";
    }
}

```

员工Controller

```

@Controller
@RequestMapping("/employee")
public class EmployeeController {
    //访问路径是/employee/list.html
    @RequestMapping("/list.html")
    public String list() {
        //省略查询员工列表的代码
        return "/emp_list.jsp";
    }

    //访问路径是/employee/update.html
    @RequestMapping("/update.html")
    public String update(Employee emp) {
        //省略修改员工代码
        return "/emp_list.jsp";
    }
}

```

父路径不同

子路径相同

DeptController 用于处理部门相关的业务。EmployeeController 用于处理员工相关的业务。部门列表叫 `list.html`，员工列表也叫 `list.html`。可以通过父路径来区分请求。不需要程序员给每个请求单独起名字，也能让项目中的 url 更规范。

3.2 通过 method 指定 POST 或 GET

`@RequestMapping` 中有一个 `method` 属性，用于指定当前的方法是用 POST 还是 GET 访问：

```

//此方法只支持 POST 访问方式
@RequestMapping(value = "/post.html", method = RequestMethod.POST)
public String showPost() {
    return "/index.jsp";
}

```

如果在浏览器中通过 GET 访问，会出现如下错误：



3.3 其它参数

其它参数需要对应到 http 请求头：

- 标准的 HTTP 请求报头



1. consumes:

```
@RequestMapping(value = "xx.html", consumes = "application/json")
```

方法仅处理 request Content-Type 为“application/json”类型的请求。

2. produces:

```
@RequestMapping(value = "xx.html", produces = "application/json")
```

方法仅处理 request 请求中 Accept 头中包含了“application/json”的请求，同时返回的内容类型为 application/json;

3. params:

```
@RequestMapping(value = "xx.html", params = "act=list")
```

只处理请求中包含 act 参数并且值为 list 的请求。

params="param1"请求中必须包含参数名为 param1 的参数。

params="!param1"请求中不能包含参数名为 param1 的参数。

params="param1!=value1"请求中包含名为 param1 的参数，但参数值不能为 value1。

params={"p1=v1",p2}请求中必须包含名称为 p1 和 p2 两个参数，且 p1 的值为 v1。

4. headers:

```
@RequestMapping(value = "xx.html", headers = "Referer=http://www.xx.com/")
```

仅处理 request 的 header 中包含了指定“Refer”请求头和对应值为“http://www.xx.com/”的请求；

3.4 REST 风格 URL

如/delete/123 和/delete/456 这两个 url，其中/delete/是公用的，后面的 123 和 456 是动态的参数。使用@PathVariable 获取参数：

```
//url 中包含参数
@RequestMapping(value = "/rest1/{id}")
public String testRest1(@PathVariable Integer id) {
    System.out.println(id);
    return "/index.jsp";
}

//占位的参数和方法中参数名不同的时候
@RequestMapping(value = "/rest2/{id}")
public String testRest2(@PathVariable("id") Integer someId) {
    System.out.println(someId);
    return "/index.jsp";
}
```

3.5 支持正则表达式

```
//支持正则表达式匹配
@RequestMapping("/reg/{param:[\\d]+.html}")
public String testRest3(@PathVariable Integer param) {
    System.out.println(param);
    return "/index.jsp";
}
```

{参数名:正则表达式}

应用场景示例：

比如有一些静态页面展示，如果每一个页面都写一个访问的方法，会增加代码量。可以使用动态参数，在路径中提取出页面名称，合并到一个方法。(ps：在项目开发中，有时需要对权限进行限制，比如有些页面必须登录后才能访问。这时就需要限制用户不能直接通过.jsp 直接访问.jsp 页面文件，而是通过.html 经过 Spring 过滤器进行处理。会在后面的章节讲到。)

The diagram illustrates how two separate methods can be simplified into one. On the left, two methods are shown: `showRegistPage()` for `/page/regist.html` and `showLoginPage()` for `/page/login.html`. Red arrows point from these to a simplified method on the right: `showPage(@PathVariable String page)`, which uses `@RequestMapping("/page/{page}.html")` and returns `page + ".jsp"`. A red label "简化成一个方法" (Simplify into one method) points to the simplified method.

```
//展示注册页面
@RequestMapping("/page/regist.html")
public String showRegistPage() {
    return "regist.jsp";
}

//展示登录页面
@RequestMapping("/page/login.html")
public String showLoginPage() {
    return "login.jsp";
}

//使用动态参数匹配，简化代码
@RequestMapping("/page/{page}.html")
public String showPage(@PathVariable String page) {
    return page + ".jsp";
}
```

上面的代码返回的页面路径前没有加/, 所以是相对于当前访问路径的, 页面需要放在 webapp 的 page 文件夹下才能显示。

3.6 ANT 风格 URL(了解)

Ant 风格支持三种匹配符:

1. `?`: 匹配一个字符。

```
@RequestMapping("/ant/test?.html")
```

可以匹配 `/ant/testaa.html` 或 `/ant/testbb.html`

2. `*`: 匹配任意字符。

```
@RequestMapping("/ant/*/test.html")
```

可以匹配 `/ant/aaa/test.html` 或 `/ant/bcd/test.html` 等

3. `**`: 匹配多层路径。

```
@RequestMapping("/ant/**/test.html")
```

可以匹配 `/ant/test.html` 或 `/ant/aa/test.html` 或 `/ant/aa/bb/test.html` 等

ANT 风格的 url 通常用在资源路径的加载中。

4. 参数绑定

4.1 @RequestParam

使用 `@RequestParam` 可以实现把请求中的参数传递给被请求的方法。

本小节需要新建一个 `ParamController.java`。

4.1.1 value

```
package controller;

import org.springframework.stereotype.Controller;
```

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class ParamController {
    @RequestMapping("param1.html")
    public String testParam1(@RequestParam(value = "name") String name,
                             @RequestParam(value = "id") Integer id) {
        System.out.println(name);
        System.out.println(id);
        return "index.jsp";
    }
}
```

在浏览器中访问如下地址：<http://localhost:8080/param1.html?name=abc&id=1>
可以看到控制台上输出：

```
abc
```

```
1
```

```
@RequestParam(value = "name") String name
```

相当于 servlet 中的：

```
String name=request.getParameter("name");
```

代码：

```
@RequestParam(value = "id") Integer id
```

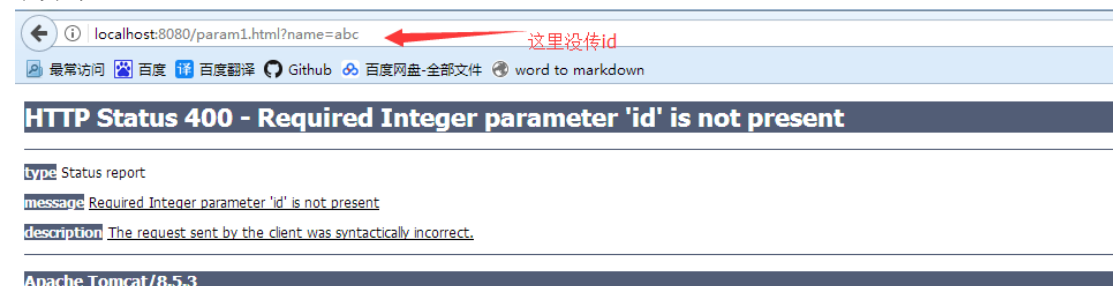
相当于 servlet 中的：

```
String idStr= request.getParameter("id");
Integer id=null;
if(idStr!=null&& idStr.trim()!="") {
    id=Integer.parseInt(idStr)
}
```

可见当参数类型不是 String 的时候，@RequestParam 可以大大简化代码。

4.1.2 required

默认的使用 @RequestParam 注解的参数都是必传的。上面的例子中，使用 @RequestParam 注解了 name 和 id 两个参数。如果在访问的时候少传一个参数，会出现如下异常：



如果某个参数不是必须的，可以使用 required 属性设置：

```
@RequestParam(value = "id",required = false) Integer id
```

required 默认是 true，代表参数必传；设置为 false，则代表参数可以为 null。

需要注意的是如果注入的是基本类型的数据，如 int 型，参数为空就会抛出异常，因为 null 不能赋值给基本数据类型，只能是对象类型。

4.1.3 defaultValue

如果没传某个参数时，想给参数一个默认值，可以使用 defaultValue 属性设置：

```
@RequestParam(value = "name",defaultValue = "123") String name
```

如果访问时没传 name 参数，将会默认给 name 赋值为 123。相当于 servlet 中的：

```
String name = request.getParameter("name");
if (name == null) {
    name = "123";
}
```

4.1.4 简化写法

```
//当 url 中的参数名与方法接收时参数名一致，且参数都是必传的，可以省略@RequestParam
@RequestMapping("param3.html")
public String testParam3(String name, Integer id) {
    System.out.println(name);
    System.out.println(id);
    return "index.jsp";
}
```

4.1.5 映射 POJO 类型参数

SpringMVC 支持 POJO 类型参数映射，即将多个参数直接封装成实体类。

如我们要将参数封装到 Employee 和 Dept 对象中，先创建实体类：

```
package pojo;

public class Dept {
    private Integer id;
    private String name;
    private List<Employee> employees;
    //getter/setter 方法略
}
```

```
package pojo;

public class Employee {
    private Integer id;
```

```

private String name;
private Float salary;
private Dept dept;
//getter/setter 方法略
}

```

controller 中的方法:

```

//映射 pojo 类
@RequestMapping("param5.html")
public String testParam5(Employee employee) {
    System.out.println("员工名: " + employee.getName());
    System.out.println("员工 id:" + employee.getId());
    //支持级联形式的映射
    System.out.println("员工部门名: " + employee.getDept().getName());
    System.out.println("员工部 id: " + employee.getDept().getId());
    return "index.jsp";
}

```

为方便测试, jsp 页面中以表单形式提交请求:

```

<form action="/param5.html" method="post">
    员工 id:<input name="id" type="text"/><br/>
    员工名: <input name="name" type="text"/><br/>
    <!--级联形式的映射-->
    员工部门 id:<input name="dept.id" type="text"/><br/>
    员工部门名: <input name="dept.name" type="text"/><br/>
    <input type="submit" value="提交"/>
</form>

```

运行结果:

员工id:1

员工名: emp

员工部门id:2

员工部门名: dept

提交

提交后控制台输出:

```

员工名: emp
员工id:1
员工部门名: dept
员工部id: 2

```

4.1.6 基本类型的数组

基本数据类型的数组如 `Integer[]`，`String[]`等

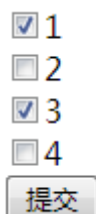
比如批量删除数据的时候，需要传递多个数据的 id，这时用一个数组去接收：

```
//映射数组
@RequestMapping("param6.html")
public String testParam6(Integer[] ids) {
    for (Integer id : ids) {
        System.out.println(id);
    }
    return "index.jsp";
}
```

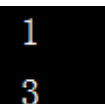
页面：

```
<form action="/param6.html" method="post">
    <input name="ids" value="1" type="checkbox"/>1<br/>
    <input name="ids" value="2" type="checkbox"/>2<br/>
    <input name="ids" value="3" type="checkbox"/>3<br/>
    <input name="ids" value="4" type="checkbox"/>4<br/>
    <input type="submit" value="提交"/>
</form>
```

运行结果：



控制台输出：



4.2 @RequestBody(选学)

对于复杂的数据类型，如 `Dept[]`、`List<Integer>`、`List<Dept>`、`List<Map<String,Object>>` 以及 `Dept` 里包含 `List<Employee>` 的映射，不能再使用简单的 form 表单提交请求了，需要使用 ajax 模拟提交 json 数据，并指定请求的 `contentType` 是 `application/json`。在 Controller 中用 `@RequestBody` 接收参数。

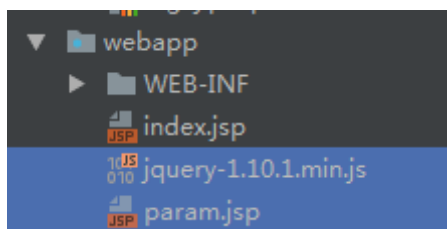
SpringMVC 解析 json 需要一个 json 适配器，json 转化使用的是 `jackson`，需要在 `pom.xml` 中添加 `jackson` 的 jar 包：

```
<!-- Jackson Json 处理工具包 -->
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.7.4</version>
</dependency>
```

在 springMVC-servlet.xml 中配置 json 适配器:

```
<bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
  <property name="messageConverters">
    <list>
      <bean
class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">
        <property name="supportedMediaTypes">
          <list>
            <value>application/json;charset=utf-8</value>
          </list>
        </property>
      </bean>
    </list>
  </property>
</bean>
```

在工程中加入 jQuery 类库，如 jquery-1.10.1.min.js:



param.jsp 是我们写页面代码的文件。

在页面中引入 jQuery:

```
<script type="text/javascript" src="jquery-1.10.1.min.js"></script>
```

4.2.1 List<Object>类型的映射

List 是基本数据类型的集合，如 List<String>类型的映射:

```
//List 集合中包含基本数据类型
@RequestMapping("param7.html")
public String testParam7(@RequestBody List<String> names) {
  for (String name : names) {
    System.out.println(name);
  }
}
```



```
    return "index.jsp";
}
```

页面中发送请求的方法：

```
<input type="button" onclick="testParam7()" value="测试 List<Object>"/>
<script type="text/javascript">
    function testParam7() {
        var nameList = ["张三", "李四 "]; //String 类型的数组
        $.ajax({
            type: "POST",
            url: "/param7.html",
            data: JSON.stringify(nameList), //将对象序列化成 JSON 字符串
            contentType: "application/json;charset=utf-8",
            success: function(data) {
                alert(data);
            }
        });
    }
</script>
```

4.2.2 List<POJO> 和 POJO[]类型的映射

如将请求数据封装成 List<Dept>:

```
//List 集合中包含对象类型
@RequestMapping("param8.html")
//如果是数组，就写(@RequestBody Dept[] depts)
public String testParam8(@RequestBody List<Dept> depts) {
    //public String testParam8(@RequestBody Dept[] depts) {
        for (Dept dept : depts) {
            System.out.println(dept.getId() + dept.getName());
        }
        return "index.jsp";
    }
}
```

如果要封装成数组，就写

页面提交请求方法：

```
<input type="button" onclick="testParam8()" value="测试 List<POJO>"/>
<script type="text/javascript">
    function testParam8() {
        var deptList = new Array(); //集合中存放的是实体类
        deptList.push({"id":1, "name":"技术部"});
        deptList.push({"id":2, "name":"测试部"});
        $.ajax({
            type: "POST",
```

```

        url: "/param8.html",
        data: JSON.stringify(deptList), //将对象序列化成 JSON 字符串
        contentType : "application/json;charset=utf-8",
        success: function(data) {
            alert(data);
        }
    });
}
</script>

```

4.2.3 POJO 中包含 List 的映射

```

//POJO 中包含 List。Dept 中包含 List<Employee>
@RequestMapping("param9.html")
public String testParam9(@RequestBody Dept dept) {
    System.out.println(dept.getName());
    for (Employee employee : dept.getEmployees()) {
        System.out.println(employee.getName());
    }
    return "index.jsp";
}

```

页面提交请求的代码:

```

<input type="button" onclick="testParam9()" value="POJO 中包含 List"/>
<script type="text/javascript">
    function testParam9() {
        var empList = new Array(); //集合中存放的是实体类
        empList.push({"id":1, "name":"张三"});
        empList.push({"id":2, "name":"李四"});
        var dept={"id":1, "name":"技术部", "employees":empList};
        $.ajax({
            type: "POST",
            url: "/param9.html",
            data: JSON.stringify(dept), //将对象序列化成 JSON 字符串
            contentType : "application/json;charset=utf-8",
            success: function(data) {
                alert(data);
            }
        });
    }
</script>

```

4.2.4 List<Map<String,Object>>的映射

List 中封装 Map 的操作和封装 POJO 类似：

```
//List<Map<String, Object>>的映射
@RequestMapping("param10.html")
public String testParam10(@RequestBody List<Map<String, Object>> map) {
    for (Map<String, Object> stringObjectMap : map) { //遍历集合
        for (String key : stringObjectMap.keySet()) { //遍历当前 Map 中的 key
            //根据 key 拿到对应的值
            System.out.println(key + "=" + stringObjectMap.get(key));
        }
        System.out.println("-----");
    }
    return "index.jsp";
}
```

页面发送请求的代码：

```
<input type="button" onclick="testParam10()" value="测试 List<Map<String,Object>>"/>
<script type="text/javascript">
    function testParam10() {
        var data = new Array(); //
        data.push({"id":1, "name": "技术部"});
        data.push({"id":3, "addr": "三楼"});
        $.ajax({
            type: "POST",
            url: "/param10.html",
            data: JSON.stringify(data), //将对象序列化成 JSON 字符串
            contentType: "application/json;charset=utf-8",
            success: function(data) {
                alert(data);
            }
        });
    }
</script>
```

4.3 使用 Servlet API

如果我们在方法中还想用 Servlet 的 request 和 response，可以在方法上添加参数：

```
//得到 HttpServletRequest 和 HttpServletResponse
@RequestMapping("param4.html")
public String testParam4(HttpServletRequest request, HttpServletResponse response) {
    System.out.println(request.getParameter("name"));
}
```

```
return "index.jsp";
}
```

需要在 pom.xml 中添加 servlet 的 jar 包：

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

5. 处理模型数据

SpringMVC 中的模型数据是非常重要的，因为 MVC 中的控制（C）请求处理业务逻辑来生成数据模型（M），而视图（V）就是为了渲染数据模型的数据。当有一个查询的请求，控制器（C）会把请求拦截下来，然后把根据请求的内容对它进行分配适合的处理方法，在处理方法上进行处理查询的业务逻辑，得到了数据，再把数据封装成数据模型对象，最后把数据模型（M）对象传给了视图（V），让视图去渲染数据模型。

SpringMVC 提供了以下几种途径输出模型数据：

- ModelAndView：处理方法返回值类型为 ModelAndView 时，方法体即可通过该对象添加模型数据。
- @ModelAttribute：方法入参标注该注解后，入参的对象就会放到数据模型中。
- Map 及 Model：入参为 org.springframework.ui.Model、org.springframework.ui.ModelMap 或 java.util.Map 时，处理方法返回时，Map 中的数据会自动添加到模型中。
- @SessionAttributes：将模型中的某个属性暂存到 HttpSession 中，以便多个请求之间可以共享这个属性。

5.1 Map 和 Model 入参

```
/**
 * 当参数为 Map 时
 * SpringMVC 会传入 一个 BindingAwareModelMap
 * 往 BindingAwareModelMap 里面存入的值 会在后面存入 request 域中
 * 相当于在方法返回前执行了一个 request.setAttribute 的操作
 */
@RequestMapping("/map.html")
public String map(Map<String, Object> map) {
    System.out.println(map.getClass().getName());
    map.put("name", "aaa");
    map.put("id", 123);
}
```

```
        return "/model.jsp";
    }

    /**
     * 参数为 Model 类型的，作用和 Map 一样
     */
    @RequestMapping("/model.html")
    public String model(Model model) {
        model.addAttribute("id", 123);
        model.addAttribute("name", "aaa");
        return "/model.jsp";
    }
}
```

测试页面：

```
name=${name}<br/>
id=${id}<br/>
```

运行结果：访问 map.html 或 model.html 的时候，页面上显示：

```
name=aaa
id=123
```

5.2 ModelAndView

ModelAndView 既包含数据模型，又包含视图信息

```
@RequestMapping("/modelandview.html")
public ModelAndView testModeAndView() {
    ModelAndView modelAndView = new ModelAndView();
    //将 Model 数据作为 request.attribute Foward 到下一个页面。
    modelAndView.addObject("id", 123);
    modelAndView.addObject("name", "abc");
    modelAndView.setViewName("/model.jsp");//设置要返回的页面
    return modelAndView;
}
```

5.3 @SessionAttributes

若希望在多个请求之间共用某个模型属性数据,则可以在控制器类上标注一个 `@SessionAttributes`,SpringMVC 将在模型中对应的属性暂存到 `HttpSession` 中。`@SessionAttributes` 只能标注在类上。

`@SessionAttributes` 除了可以通过属性名指定需要放到会话中的属性外，还可以通过模型属性的对象类型指定哪些模型属性需要放到会话中

- `@SessionAttributes(types=Dept.class)` 会将隐含模型中所有类型为 `Dept.class` 的属性添加到 session 中。
- `@SessionAttributes(value={"user","admin"})` 会将模型中名为 `user` 和 `admin` 的属性添加到 session 中
- `@SessionAttributes(types={Dept.class, Employee.class})` 会将模型中所有类型为 `Dept` 和 `Employee` 的属性添加到 session 中
- `@SessionAttributes(value={"user","admin"}, types={Dept.class})` 会将模型中名为 `user` 和 `admin` 和类型为 `Dept` 的对象放到 session 中。

在类上添加 `@SessionAttributes` 注解

```
@SessionAttributes(types = Dept.class, value = {"user", "admin"})
@Controller
public class ModelController {
```

测试方法：

```
@RequestMapping("/session.html")
public ModelAndView testSettion(Map<String, Object> map) {
    map.put("admin", "I am admin");
    map.put("user", "I am user");
    Dept dept = new Dept();
    dept.setName("session name");
    map.put("dept", dept);
    // @SessionAttributes 注解里没有声明 other 这个属性，所以不会在 session 中
    map.put("other", "I'm other");
    return new ModelAndView("/model.jsp", "result", map);
}
```

测试页面：

```
request 中的属性:<br/>
admin:${requestScope.admin}<br/>
user:${requestScope.user}<br/>
dept.name:${requestScope.dept.name}<br/>
other:${requestScope.other}<br/>
session 中的属性:<br/>
admin:${sessionScope.admin}<br/>
user:${sessionScope.user}<br/>
dept.name:${sessionScope.dept.name}<br/>
other:${sessionScope.other}<br/>
```

运行效果：

```
request中的属性:
admin:I am admin
user:I am user
dept.name:session name
other:I'm other
session中的属性:
admin:I am admin
user:I am user
dept.name:session name
other:
```

可以看到模型中的属性都放到了 request 的域中。@SessionAttributes 中没有声明 other，所以 session 中的 other 是空的。

5.4 @ModelAttribute

5.4.1 方法参数上使用@ModelAttribute

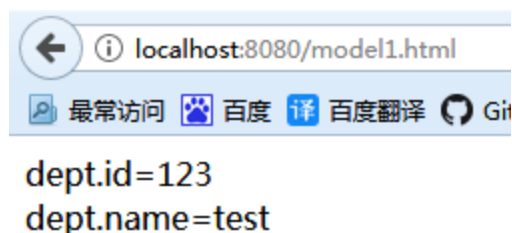
在参数前使用@ModelAttribute，在进去方法时可以通过参数给对象赋值，如下面的代码，当请求/model1.html?id=1 的时候，会给 dept 的 id 属性赋值。在方法中可以对 dept 做进一步的处理。@ModelAttribute 可以自动将被注解的对象作为数据模型返回给页面。

```
@RequestMapping("/model1.html")
public String testModelAttribute(@ModelAttribute Dept dept) {
    dept.setId(123);
    dept.setName("test");
    //使用@ModelAttribute 注解 dept
    //相当于执行了 request.setAttribute("dept",dept);
    //页面上可以直接取数据
    return "/model.jsp";
}
```

测试页面 model.jsp，使用 EL 表达式取值：

```
<body>
dept.id=${dept.id}<br/>
dept.name=${dept.name}
</body>
```

运行结果



5.4.2 定义方法时使用@ModelAttribute

在方法上使用@ModelAttribute 后，执行这个 Controller 的任意一个方法之前，都会调用这个方法给对象赋值。

```
/**
 * 在方法上使用@ModelAttribute，调用这个 Controller 任意一个方法之前
 * 都会执行这个方法给模型赋值
 */
@ModelAttribute("dept")
public Dept getDept() {
    Dept dept = new Dept();
    dept.setId(456);
    dept.setName("name");
    return dept;
}

/**
 * 在调用这个方法前，会执行 getDept()
 * 如果请求中有参数，会覆盖掉 getDept() 的值
 * dept 会作为数据模型返回到页面上
 */
@RequestMapping("/model2.html")
public String testModelAttribute2(@ModelAttribute Dept dept) {
    System.out.println(dept.getId());
    System.out.println(dept.getName());
    return "/model.jsp";
}
```

6. 视图和视图解析器

对于 Controller 的目标方法，无论其返回值是 String、View、ModelMap 或是 ModelAndView，SpringMVC 都会在内部将它们封装为一个 ModelAndView 对象进行返回。

Spring MVC 借助视图解析器（ViewResolver）得到最终的视图对象（View），最终的视图可以是 JSP 也可能是 Excell、JFreeChart 等各种表现形式的视图。

View ---View 接口表示一个响应给用户的视图，例如 jsp 文件，pdf 文件，html 文件等。视图的作用是渲染模型数据，将模型里的数据以某种形式呈现给客户。

为了实现视图模型和具体实现技术的解耦，Spring 在 `org.springframework.web.servlet` 包中定义了一个高度抽象的 **View** 接口。

视图对象由视图解析器负责实例化。由于视图是无状态的，所以他们不会有线程安全的问题。所谓视图是无状态的，是指对于每一个请求，都会创建一个 **View** 对象。

JSP 是最常见的视图技术。

6.1 ViewResolver

ViewResolver 的主要作用是把一个逻辑上的视图名称解析为一个真正的视图，SpringMVC 中用于把 **View** 对象呈现给客户端的是 **View** 对象本身，而 **ViewResolver** 只是把逻辑视图名称解析为对象的 **View** 对象。

6.1.1 InternalResourceViewResolver

InternalResourceViewResolver 可以在视图名称前自动加前缀或后缀：

```
<!-- 配置视图解析器 -->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/"></property><!-- 前缀 -->
    <property name="suffix" value=".jsp"></property><!-- 后缀 -->
</bean>
```

如果配置了上面的解析器，Controller 中返回字符串就不需要写 `/index.jsp` 了，直接返回 `"index"`，就会按照 `/index.jsp` 去解析

6.1.2 MappingJackson2JsonView

用于返回 json 类型 的数据，用法见 11.2。

6.1.3 FreeMarkViewResolver

FreeMaker 介绍请见文档《FreeMaker 入门.docx》

FreeMarker 与 spring 整合需要导入 jar：

```
<dependency>
    <groupId>org.freemarker</groupId>
    <artifactId>freemarker</artifactId>
    <version>2.3.23</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-support</artifactId>
```

```
<version>4.3.11.RELEASE</version>
</dependency>
```

使用 FreeMarker 模板生成静态网页，需要在 springMVC-servlet.xml 中配置：

```
<bean id="freemarkerConfig"
      class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
    <!--模板存放路径-->
    <property name="templateLoaderPath" value="/WEB-INF/ftl/" />
    <property name="defaultEncoding" value="UTF-8" />
</bean>
```

在 WEB-INF/ftl 下创建模板文件 hello.ftl：



```
<html>
<head>
    <title>Title</title>
</head>
<body>
<h1>${hello}</h1>
</body>
```

通过模板生成静态网页：

```
package controller;

import freemarker.template.Configuration;
import freemarker.template.Template;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer;

import java.io.File;
import java.io.FileWriter;
import java.io.Writer;
import java.util.HashMap;
import java.util.Map;

@Controller
public class FreeMarkerController {
    @Autowired
    private FreeMarkerConfigurer freeMarkerConfigurer;


    @RequestMapping("/freemarker.html")
```

```

@ResponseBody
public String genHtml() throws Exception {
    // 1、从 spring 容器中获得 FreeMarkerConfigurer 对象。
    // 2、从 FreeMarkerConfigurer 对象中获得 Configuration 对象。
    Configuration configuration = freeMarkerConfigurer.getConfiguration();
    // 3、使用 Configuration 对象获得 Template 对象。
    Template template = configuration.getTemplate("hello.ftl");
    // 4、创建数据集
    Map dataModel = new HashMap<>();
    dataModel.put("hello", "1000");
    // 5、创建输出文件的 Writer 对象。
    Writer out = new FileWriter(new File("F:/spring-freemarker.html"));
    // 6、调用模板对象的 process 方法，生成文件。
    template.process(dataModel, out);
    // 7、关闭流。
    out.close();
    return "OK";
}
}

```

在 F 盘下就能看到 spring-freemarker.html 文件了

 spring-freemarker.html

以上是生成静态网页的配置。

如果想像读取 jsp 一样动态展示 freeMarker 的页面，可以配置视图解析器：

```

<bean id="viewResolverFtl"
class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver">
    <property name="viewClass"
value="org.springframework.web.servlet.view.freemarker.FreeMarkerView"/>
    <property name="contentType" value="text/html; charset=utf-8"/>
    <property name="cache" value="true" />
    <property name="suffix" value=".ftl" />
    <property name="order" value="0"/>
</bean>

```

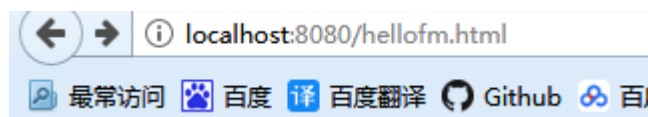
order 越小，视图解析器的优先级就越高。

```

@RequestMapping("/hellofm.html")
public String sayHello(ModelMap map) {
    //传递属性到页面
    map.addAttribute("hello", " Hello FreeMarker!");
    return "/hello";//去找 hello.ftl
}

```

运行结果：



Hello FreeMarker!

6.1.4 BeanNameViewResolver

引入 servlet 的 jar:

```
<!--servlet 依赖 jar 包-->
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
</dependency>
```

自定义一个视图，然后声明成 bean，Controller 中返回这个 bean 的名字，就可以显示当前的视图：

```
package view;

import org.springframework.web.servlet.View;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.Map;

public class HelloView implements View {

    public String getContentType() {
        return "text/html";
    }

    public void render(Map<String, ?> model, HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        //向相应中写入数据
        response.getWriter().print("Welcome to View:Hello");
    }
}
```

springMVC-servlet.xml 中配置：

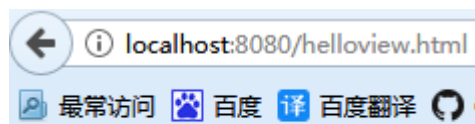
```
<bean class="org.springframework.web.servlet.view.BeanNameViewResolver">
  <property name="order" value="10" /><!--优先级靠后-->
</bean>
```

```
<bean id="helloView" class="view.HelloView"/>
```

Controller

```
@RequestMapping("helloview.html")
public String hello() {
    //因为当前没有 helloView.jsp
    //所以视图解析器依次执行，找到 id=helloView 的视图并显示
    return "helloView";
}
```

运行结果：



Welcome to View:Hello

6.2 自定义 View

处理 json 数据需要 json 的 jar 包

```
<!-- Jackson Json 处理工具包 -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.7.4</version>
</dependency>
```

```
package view;

import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.web.servlet.view.AbstractView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import java.text.SimpleDateFormat;
import java.util.Map;

public class JsonView extends AbstractView {

    /**
     * 该 View 对应的输出类型
     */
}
```

```

@Override
public String getContentType() {
    return "application/json; charset=UTF-8";
}

//向响应中写入数据
@Override
protected void renderMergedOutputModel(Map<String, Object> model,
                                       HttpServletRequest request, HttpServletResponse response)
                                       throws Exception {
    ObjectMapper mapper = new ObjectMapper();
    //设置 Date 类型的格式，默认是显示毫秒数
    mapper.setDateFormat(new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"));
    //需要注意的是放入 model 的对象一定要实现 Serializable 接口才能转化成 json
    String jsonStr = mapper.writeValueAsString(model);
    response.setContentType(getContentType());
    response.setHeader("Cache-Control", "no-cache");
    response.setCharacterEncoding("UTF-8");
    PrintWriter out = null;
    try {
        out = response.getWriter();
        out.print(jsonStr);
        out.flush();
    } catch (IOException e) {
    } finally {
        if (out != null) {
            out.close();
            out = null;
        }
    }
}
}

```

测试代码：

```

@RequestMapping("myview.html")
public ModelAndView myView() {
    Map<String, Object> result = new HashMap<>();
    result.put("key1", "123");
    result.put("key2", new String[]{"a", "b"});
    result.put("key3", new Date());
    return new ModelAndView(new JsonView(), result);
}

```

如果是 map 中的值是其它对象类型的，传给 ModelAndView 的数据必须有一个 modelName

6.3 转发和重定向

```
public String showView2() {
    //转发前面加 forward:
    return "index.html";
}

@RequestMapping("/redirect.html")
public String showView3() {
    //重定向前面加 redirect:
    return "redirect:index.html";
}
```

7. @ResponseBody

该注解用于将 Controller 的方法返回的对象，通过适当的 `HttpMessageConverter` 转换为指定格式后，写入到 `Response` 对象的 `body` 数据区。使用时机：返回的数据不是 `html` 标签的页面，而是其他某种格式的数据时（如 `json`、`xml` 等）使用。

7.1 返回 json 数据

处理 json 数据需要 json 的 jar 包

```
<!-- Jackson Json 处理工具包 -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.7.4</version>
</dependency>
```

返回 json 类型的数据，需要在 spring 配置文件中加入如下配置：

```
<!--配置返回值转换器-->
<bean id="contentNegotiationManagerFactoryBean"
    class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean">
    <!--是否支持后缀匹配-->
    <property name="favorPathExtension" value="true"/>
    <!--是否支持参数匹配-->
    <property name="favorParameter" value="true"/>
    <!--是否 accept-header 匹配-->
    <property name="ignoreAcceptHeader" value="false"/>
    <property name="mediaTypes">
```

```

        <map>
            <!--表示. json 结尾的请求返回 json-->
            <entry key="json" value="application/json"/>
            <!--表示.xml 结尾的返回 xml-->
            <entry key="xml" value="application/xml"/>
        </map>
    </property>
</bean>

```

测试 favorPathExtension 请求后缀分别是.xml 和.json

测试 favorParameter 请求中参数 format=json 和 format=xml

测试 ignoreAcceptHeader,请求的 Header 中 Accept=application/json 或 Accept=application/xml

如果要返回 Xml, 需要将要转换为 xml 的实体类上添加注解, 如:

```

@XmlRootElement
public class Dept {

```

在<mvc:annotation-driven/>标签中指定 content-negotiation-manager

```

<mvc:annotation-driven
content-negotiation-manager="contentNegotiationManagerFactoryBean"/>

```

测试类:

```

package controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import pojo.Dept;

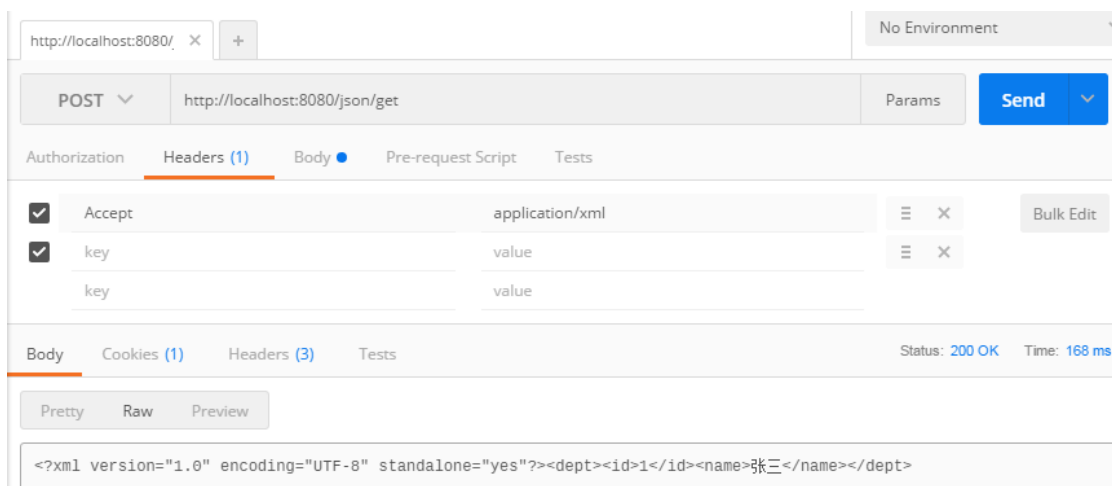
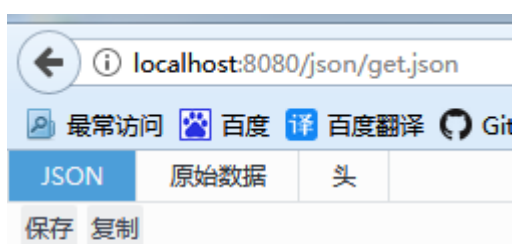
@Controller
@RequestMapping("/json")
public class JsonController {
    @RequestMapping("/get ")
    @ResponseBody//会自动将返回值转换成 json
    public Dept getJson() {
        Dept dept = new Dept();
        dept.setId(1);
        dept.setName("张三");
        return dept;
    }
}

```

测试结果:



```
- <dept>
  <id>1</id>
  <name>张三</name>
</dept>
```



7.2 实现 RESTFUL

```
package controller;

import org.springframework.web.bind.annotation.*;
import pojo.Dept;

@RestController//相当于本类中所有的方法都加了@ResponseBody
```

```
@RequestMapping("/rest")
public class RestTestController {
    //通过 method 限制请求的方式
    @RequestMapping(value =("/{id}", method = RequestMethod.GET)
    public Dept getDept(@PathVariable Integer id) {
        //模拟从数据库中查出一条数据
        Dept dept = new Dept();
        dept.setId(id);
        dept.setName("张三");
        return dept;
    }

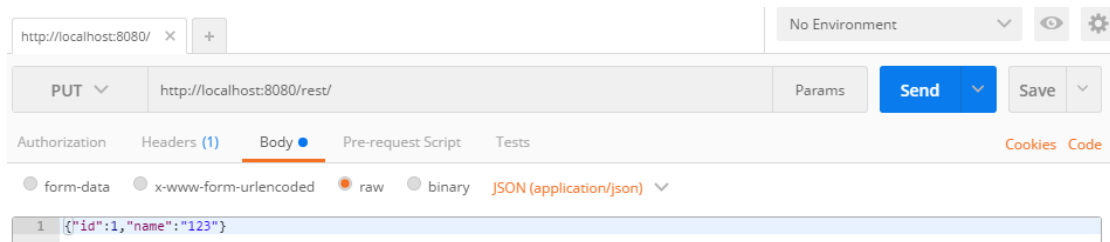
    @RequestMapping(method = RequestMethod.POST)
    public Dept addDept(Dept dept) {
        //模拟插入数据后生成主键
        dept.setId(1);
        System.out.println(dept.getName());
        return dept;
    }

    @RequestMapping(method = RequestMethod.PUT, consumes = "application/json")
    public Dept updateDept(@RequestBody Dept dept) {
        System.out.println(dept.getName());
        //执行修改的业务略
        dept.setName("修改");//模拟修改名字
        return dept;
    }

    @RequestMapping(value =("/{id}", method = RequestMethod.DELETE)
    public String deleteDept(@PathVariable Integer id) {
        //执行删除的业务略
        System.out.println(id);
        return "删除成功";
    }
}
```

通过 postman 可以测试请求(ajax 方式无法测试 PUT 和 DELETE)。

测试 put 的时候，请求的 body 设置为 raw，Headers 的 ContentType=application/json，否则会报 415:



注意类名上方的@Controller，相当于在类中每个方法上都添加了@RequestBody。
deleteDept 方法返回了一句 String 类型的提示信息，默认的 String 类型的返回值，编码是 ISO-8859-1,中文会乱码，解决方案是在配置文件中修改编码：

修改<mvc:annotation-driven>节点，添加<mvc:message-converters>

```
<mvc:annotation-driven
content-negotiation-manager="contentNegotiationManagerFactoryBean">
    <!--String 返回值默认编码是 ISO-8859-1，需要-->
    <mvc:message-converters>
        <bean class="org.springframework.http.converter.StringHttpMessageConverter"
            <constructor-arg value="UTF-8" />
        </bean>
    </mvc:message-converters>
</mvc:annotation-driven>
```

8.HttpEntity

HttpEntity 和@RequestBody 和@ResponseBody 类似，除了可以得到 request 和 response 的 body 以外，还可以操作 header。

```
@RequestMapping("/entity.html")
public ResponseEntity<Dept> getEntity(RequestEntity<Dept> requestEntity) {
    //获取请求头
    String requestHeader = requestEntity.getHeaders().getFirst("MyRequestHeader");
    System.out.println(requestHeader);
    Dept dept = new Dept();
    dept.setId(1);
    dept.setName("张三");
    HttpHeaders responseHeaders = new HttpHeaders();//创建响应头
    responseHeaders.set("MyResponseHeader", "MyValue");//自定义响应头
    //响应对象
    ResponseEntity<Dept> responseEntity =
        new ResponseEntity<>(dept, responseHeaders, HttpStatus.OK);
    return responseEntity;
}
```

测试的页面：

```
<input type="button" onclick="testEntity()" value="测试HttpEntity"/>
<script type="text/javascript">
    function testEntity() {
        $.ajax({
            type: "POST",
            url: "/json/entity.html",
            headers: {"MyRequestHeader": "abc"},
            success: function(data, status, xhr) { //xhr 可以看到响应的头
                alert(data.id);
                alert(status);
                alert("Header="+xhr.getResponseHeader("MyResponseHeader"));
            },
            error: function(data, status, xhr) {
                alert(data.id);
                alert(status);
                alert("Header="+xhr.getResponseHeader("MyResponseHeader"));
            }
        });
    }
}
</script>
```

9. 文件上传

SpringMVC 的文件上传非常简便，首先导入文件上传依赖的 jar：

```
<!-- 文件上传所依赖的 jar 包 -->
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.3.1</version>
</dependency>
```

在 springMVC-servlet.xml 配置文件中配置文件解析器：

```
<!--1*1024*1024 即 1M resolveLazily 属性启用是为了推迟文件解析，以便捕获文件大小异常-->
<!--文件上传解析器-->
<bean id="multipartResolver"
    class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <property name="maxUploadSize" value="1048576"/>
</bean>
```

```
<property name="defaultEncoding" value="UTF-8"/>
<property name="resolveLazily" value="true"/>
</bean>
```

注意解析器的 id 必须等于 multipartResolver, 否则上传会出现异常:

```
root cause
org.springframework.web.multipart.MultipartException: Could not parse multipart servlet request; nested exception is java.lang.IllegalStateException: Unable to process parts as no multi-part configuration has been provided
org.springframework.web.multipart.support.StandardMultipartHttpServletRequest.parseRequest(StandardMultipartHttpServletRequest.java:112)
org.springframework.web.multipart.support.StandardMultipartHttpServletRequest.<init>(StandardMultipartHttpServletRequest.java:86)
org.springframework.web.multipart.support.StandardMultipartHttpServletRequest.<init>(StandardMultipartHttpServletRequest.java:73)
org.springframework.web.multipart.support.MultipartResolutionDelegate.adaptToMultipartHttpServletRequest(MultipartResolutionDelegate.java:80)
org.springframework.web.multipart.support.MultipartResolutionDelegate.resolveMultipartArgument(MultipartResolutionDelegate.java:103)
org.springframework.web.method.annotation.RequestParameterMethodArgumentResolver.resolveValue(RequestParameterMethodArgumentResolver.java:162)
org.springframework.web.method.annotation.AbstractHandlerMethodArgumentResolver.resolveArgument(AbstractHandlerMethodArgumentResolver.java:103)
org.springframework.web.method.support.HandlerMethodArgumentResolverComposite.resolveArgument(HandlerMethodArgumentResolverComposite.java:112)
org.springframework.web.method.support.InvocableHandlerMethod.getMethodArgumentValues(InvocableHandlerMethod.java:158)
org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:128)
org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:97)
org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandlerMethod(RequestMappingHandlerAdapter.java:827)
org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.java:738)
org.springframework.web.servlet.mvc.method.annotation.AbstractHandlerMethodAdapter.handle(AbstractHandlerMethodAdapter.java:85)
org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:867)
org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:901)
org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:970)
org.springframework.web.servlet.FrameworkServlet.doPost(FrameworkServlet.java:872)
javax.servlet.http.HttpServlet.service(HttpServlet.java:643)
org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:846)
javax.servlet.http.HttpServlet.service(HttpServlet.java:729)
org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:52)
org.springframework.web.filter.CharacterEncodingFilter.doFilterInternal(CharacterEncodingFilter.java:197)
org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:107)

root cause
java.lang.IllegalStateException: Unable to process parts as no multi-part configuration has been provided
org.apache.catalina.connector.Request.parseParts(Request.java:2742)
org.apache.catalina.connector.Request.getParts(Request.java:2709)
org.apache.catalina.connector.RequestFacade.getParts(RequestFacade.java:1094)
org.springframework.web.multipart.support.StandardMultipartHttpServletRequest.parseRequest(StandardMultipartHttpServletRequest.java:93)
```

9.1 单个文件上传

```
package controller;

import org.apache.commons.io.FileUtils;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.web.multipart.commons.CommonsMultipartFile;

import java.io.File;

@Controller
public class FileController {

    /**
     * 上传单个文件操作
     * MultipartFile file 就是上传的文件
     * @return
     */
    @RequestMapping(value = "/upload1.html")
    public String fileUpload1(@RequestParam("file") MultipartFile file) {
        try {
            //将上传的文件存在 E:/upload/下
            FileUtils.copyInputStreamToFile(file.getInputStream(),
                new File("E:/upload/",
                    file.getOriginalFilename()));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

        //上传成功返回原来页面
        return "/file.jsp";
    }
}

```

上传文件时，Controller 的方法中参数类型是 `MultipartFile` 即可将文件映射到参数上。

页面：

file.jsp:

```

<form method="post" action="/upload1.html" enctype="multipart/form-data">
    <input type="file" name="file"/>
    <button type="submit">提交</button>
</form>

```

另外上传的文件还可以映射成 `CommonsMultipartFile`，它是 `MultipartFile` 的子类：

```

/**
 * CommonsMultipartFile 是 MultipartFile 的子类
 * @return
 */
@RequestMapping("/upload2.html")
public String fileUpload2(@RequestParam("file") CommonsMultipartFile file) {
    try {
        System.out.println("fileName: " + file.getOriginalFilename());
        String path = "E:/upload/" + file.getOriginalFilename();
        File newFile = new File(path);
        //通过 CommonsMultipartFile 的方法直接写文件
        file.transferTo(newFile);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return "/file.jsp";
}

```

页面：

```

<form method="post" action="/upload2.html" enctype="multipart/form-data">
    <input type="file" name="file"/>
    <button type="submit">提交</button>
</form>

```

9.2 批量上传

批量上传文件的时候，把方法中的参数类型写成数组即可：

```

/**
 * 批量上传的时候参数是数组
 * @return
 */
@RequestMapping("/upload3.html")

```

```

public String fileUpload3(@RequestParam("file") CommonsMultipartFile[] file) {
    try {
        //批量上传时遍历文件数组
        for (CommonsMultipartFile f : file) {
            System.out.println("fileName: " + f.getOriginalFilename());
            String path = "E:/upload/" + f.getOriginalFilename();
            File newFile = new File(path);
            f.transferTo(newFile);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return "/file.jsp";
}

```

页面：

```

<form method="post" action="/upload3.html" enctype="multipart/form-data">
    <!--注意 name 都是 file, 与 Controller 中方法的参数名对应-->
    <input type="file" name="file"/>
    <input type="file" name="file"/>
    <input type="file" name="file"/>
    <button type="submit" >提交</button>
</form>

```

10. 使用拦截器

模拟登陆拦截器：

```

package interceptor;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.handler.HandlerInterceptorAdapter;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class LoginInterceptor extends HandlerInterceptorAdapter {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
        System.out.println("开始拦截");
        // 其他情况判断 session 中是否有 key, 有的话继续用户的操作
        if (request.getSession().getAttribute("user") != null) {

```

```

        return true;
    }
    // 最后的情况就是进入登录页面
    response.sendRedirect(request.getContextPath() + "/login.jsp");
    return false;
}

@Override
public void postHandle(HttpServletRequest request, HttpServletResponse response,
Object handler, ModelAndView modelAndView) throws Exception {
    System.out.println("视图解析前 postHandle");
}

@Override
public void afterConcurrentHandlingStarted(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
    System.out.println("处理异步请求");
}
}

```

配置拦截器:

```

<mvc:interceptors>
    <mvc:interceptor><!--配置局部拦截器，需要满足下列路径条件-->
        <mvc:mapping path="/*" />
        <mvc:exclude-mapping path="/login.html" />
        <bean class="interceptor.LoginInterceptor"/><!--自定义拦截器注册-->
    </mvc:interceptor>
</mvc:interceptors>

```

登陆的 Controller

```

package controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

import javax.servlet.http.HttpServletRequest;

@Controller
public class LoginController {
    @RequestMapping("login.html")
    public String login(String username, HttpServletRequest request) {
        //模拟登陆
        request.getSession().setAttribute("user", username);
        return "redirect:index.html";
    }
}

```


登陆页面:

```
<form action="/login.html">
    <input name="username"/><input type="submit">
</form>
```

11. 异常处理

11.1 集成异常处理

对于一些通用的，不需要特殊处理的异常，可以使用统一的异常处理器，在 springMVC-servlet.xml 中加入配置：

```
<!--集成异常处理-->
<bean
class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <!-- 定义默认的异常处理页面-->
    <property name="defaultErrorView" value="error.jsp"></property>
    <!-- 定义异常处理页面用来获取异常信息的变量名，默认名为 exception -->
    <property name="exceptionAttribute" value="ex"></property>
    <!-- 定义需要特殊处理的异常，用类名或完全路径名作为 key，异常也页名作为值 -->
    <property name="exceptionMappings">
        <props>
            <prop key="java.lang.NullPointerException">error.jsp</prop>
            <prop key="java.lang.ClassCastException">error.jsp</prop>
            <prop key="java.lang.IndexOutOfBoundsException">error.jsp</prop>
            <!-- 这里还可以继续扩展对不同异常类型的处理 -->
        </props>
    </property>
</bean>
```

我们写一个 controller 专门抛出异常，用来模拟程序中可能出现的异常信息：

```
package controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class ExceptionController {
    @RequestMapping("/ex.html")
    public String exceptionTest(Integer type) throws Exception {
        //手动抛出几个异常，模拟程序中可能出现的异常
        switch (type) {
            case 1:
```

```

        throw new NullPointerException("测试空指针异常");
    case 2:
        throw new ClassCastException("测试类型转换异常");
    case 3:
        throw new IndexOutOfBoundsException("测试越界异常");
    }
    return "index.jsp";
}
}

```

测试页面：

```

<body>
<a href="/ex.html?type=1">空指针</a><br/>
<a href="/ex.html?type=2">类型转换</a><br/>
<a href="/ex.html?type=3">越界</a><br/>
${ex}<!--就是 spring 异常处理器中配置的 exceptionAttribute-->
</body>

```

运行结果：

[空指针](#)

[类型转换](#)

[越界](#)

java.lang.IndexOutOfBoundsException: 测试越界异常

点击不同的链接可以看大不同的提示信息。

11.2 自定义异常处理

对于需要特殊处理的异常，可以自定义异常处理器

自定义异常处理的类需要继承 HandlerExceptionResolver

```

package exception;

import org.springframework.web.servlet.HandlerExceptionResolver;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class MyExceptionHandler implements HandlerExceptionResolver {

    @Override
    public ModelAndView resolveException(HttpServletRequest httpServletRequest,
                                         HttpServletResponse httpServletResponse, Object o, Exception e) {
        String msg = e.getMessage();
        httpServletRequest.setAttribute("ex", msg);
    }
}

```

```

        return new ModelAndView("/error.jsp");
    }
}

```

在 spring 配置文件中定义 bean:

<!--自定义异常处理-->

```
<bean id="exceptionHandler" class="exception.MyExceptionHandler"/>
```

注：自定义异常处理和 11.1 中的集成异常处理不能一起使用，需要将 11.1 中的 bean 注释掉。

如果需要处理 ajax 发送的请求，出现异常时返回 json 数据，可以增加处理 json 的代码。

修改 MyExceptionHandler 中的代码：

```

package exception;

import org.springframework.web.servlet.HandlerExceptionResolver;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.view.json.MappingJackson2JsonView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.HashMap;
import java.util.Map;

public class MyExceptionHandler implements HandlerExceptionResolver {

    @Override
    public ModelAndView resolveException(HttpServletRequest httpServletRequest,
        HttpServletResponse httpServletResponse, Object o, Exception e) {
        String msg = e.getMessage();
        if (httpServletResponse.isCommitted()) {
            return null;
        }
        //如果是 ajax 请求就返回 json 数据
        if (isAjax(httpServletRequest)) {
            Map<String, String> result = new HashMap<>();
            result.put("ex", msg);
            MappingJackson2JsonView view = new MappingJackson2JsonView();
            return new ModelAndView(view, "result", result);
        } else { //不是 ajax 就返回错误页面
            httpServletRequest.setAttribute("ex", msg);
            return new ModelAndView("/error.jsp");
        }
    }

    public static boolean isAjax(HttpServletRequest request) {

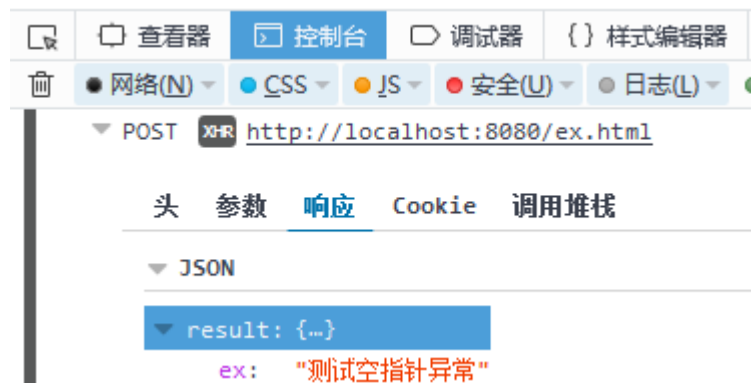
```

```
        return
    "XMLHttpRequest".equalsIgnoreCase(request.getHeader("X-Requested-With")) ||
    request.getParameter("ajax") != null;
    }
}
```

测试页面：

```
<script type="text/javascript" src="jquery-1.10.1.min.js"></script>
<input type="button" onclick="testEx()" value="测试自定义异常"/>
<script type="text/javascript">
    function testEx() {
        $.ajax({
            type: "POST",
            url: "/ex.html?type=1",
            success: function(data) {
                alert(data.result.ex); // 读取 json 数据
            }
        });
    }
</script>
```

测试结果：



12. 数据校验、数据格式化



参考 博客 <http://www.importnew.com/19477.html>

12.1 数据校验

使用 spring 数据校验，先要导入校验器的 jar:

```

<!--数据校验-->
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.2.Final</version>
</dependency>
  
```

此处使用的 hibernate 校验器

JSR 规范:

在实体类的属性上添加注解，可以完成数据校验:

@Null 被注释的元素必须为 null

@NotNull 被注释的元素必须不为 null

@AssertTrue 被注释的元素必须为 true

@AssertFalse 被注释的元素必须为 false

@Min(value) 被注释的元素必须是一个数字，其值必须大于等于指定的最小值

@Max(value) 被注释的元素必须是一个数字，其值必须小于等于指定的最大值

@DecimalMin(value) 被注释的元素必须是一个数字，其值必须大于等于指定的最小值

@DecimalMax(value) 被注释的元素必须是一个数字，其值必须小于等于指定的最大值

@Size(max=, min=) 被注释的元素的大小必须在指定的范围内

@Digits(integer, fraction) 被注释的元素必须是一个数字，其值必须在可接受的范围内
 @Past 被注释的元素必须是一个过去的日期
 @Future 被注释的元素必须是一个将来的日期
 @Pattern(regex=, flag=) 被注释的元素必须符合指定的正则表达式

 Hibernate Validator 附加的注解

@NotBlank(message =) 验证字符串非 null，且长度必须大于 0
 @Email 被注释的元素必须是电子邮箱地址
 @Length(min=, max=) 被注释的字符串的大小必须在指定的范围内
 @NotEmpty 被注释的字符串的必须非空
 @Range(min=, max=, message=) 被注释的元素必须在合适的范围内

如 Employee 实体类中：

```
public class Employee {
    private Integer id;
    @NotEmpty(message = "用户名不能为空")
    @Size(min = 3, max = 6, message = "姓名长度应在{min}-{max}")
    private String name;

    @Min(value = 2700, message = "工资不能少于{value}")
    @Max(value = 10000, message = "工资不能超过{value}")
    private Float salary;
```

在 springMVC-servlet.xml 中配置校验器：

```
<!-- 配置校验器 -->
<bean id="validator"
    class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">
    <!-- 校验器，使用 hibernate 校验器 -->
    <property name="providerClass"
        value="org.hibernate.validator.HibernateValidator"/>
</bean>
```

12.1.1 简单的数据校验

新建 ValidateController

```
package controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.validation.FieldError;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
```

```

import org.springframework.web.servlet.ModelAndView;
import pojo.Employee;

import java.security.NoSuchAlgorithmException;

@Controller
public class ValidateController {
    @RequestMapping("/val1.html")
    public ModelAndView validate1(@Validated Employee emp, BindingResult result) {
        if (result.hasErrors()) { //如果验证错误
            FieldError nameError = result.getFieldError("name");
            FieldError salaryError = result.getFieldError("salary");
            ModelAndView view = new ModelAndView();
            view.setViewName("/validate.jsp"); //如果有错就返回原页面
            if (nameError != null) {
                view.addObject("nameError", nameError.getDefaultMessage());
            }
            if (salaryError != null) {
                view.addObject("salaryError", salaryError.getDefaultMessage());
            }
            return view;
        }
        //验证成功去首页
        return new ModelAndView("/index.jsp");
    }
}

```

@Validated 修饰的参数会被按照规则校验。BindingResult 会存放校验信息。

在需要校验的 pojo 前边添加 @Validated，在需要校验的 pojo 后边添加 BindingResult
bindingResult 接收校验出错信息

注意：@Validated 和 BindingResult bindingResult 是配对出现，并且形参顺序是固定的（一前一后）

页面 validate.jsp

```

<form action="/val1.html" method="post">
    name:<input type="text" name="name"/>${nameError}<br/>
    salary:<input type="text" name="salary"/>${salaryError}<br/>
    <input type="submit" value="提交"/>
</form>

```

运行结果：

name:

salary:

提交后:

name: 姓名长度应在3-6

salary: 工资不能少于2700

12.1.2 使用 @ModelAttribute 和 <form:>

spring 有自定义的表单标签，<form:>冒号后面的是生成 html 的标签名，如<form:input>就会生成一个<input>:

```
<form:form modelAttribute="empModel" method="post" action="/val2.html">
    name: <form:input path="name" /><br/>
    <!--输出 name 的校验信息-->
    <form:errors path="name"></form:errors><br/>
    salary: <form:input path="salary" /><br/>
    <form:errors path="salary"></form:errors><br/>
    <input type="submit" value="Submit" /><br/>
    <!--输出所有错误信息-->
    <form:errors path="*"></form:errors>
</form:form>
```

使用这个标签需要在 jsp 页面中引入头文件:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

<form:>标签将的 modelAttribute 与 action 对应的方法上 @ModelAttribute 修饰的对象对应。要想进入这个页面，首先要经过一个 Controller 的方法，在 Model 中添加对应的属性:

```
@RequestMapping("/goVal2.html")
public String goVal2(Model model) {
    if (!model.containsAttribute("empModel")) {
        //empModel 与页面中的<form:form modelAttribute="empModel">对应
        model.addAttribute("empModel", new Employee());
    }
    return "/validate.jsp";
}
```

如果不经 Controller 或者 Controller 中没有放 empModel 这个属性，那么页面就会报错:

```
root cause
java.lang.IllegalStateException: Neither BindingResult nor plain target object for bean name 'empModel' available as request attribute
    org.springframework.web.servlet.support.BindStatus.<init> (BindStatus.java:144)
    org.springframework.web.servlet.tags.form.AbstractDataBoundFormElementTag.getBindStatus (AbstractDataBoundFormElementTag.java:168)
    org.springframework.web.servlet.tags.form.AbstractDataBoundFormElementTag.getPropertyPath (AbstractDataBoundFormElementTag.java:188)
    org.springframework.web.servlet.tags.form.AbstractDataBoundFormElementTag.getName (AbstractDataBoundFormElementTag.java:154)
    org.springframework.web.servlet.tags.form.AbstractDataBoundFormElementTag autogeneratedId (AbstractDataBoundFormElementTag.java:141)
    org.springframework.web.servlet.tags.form.AbstractDataBoundFormElementTag.resolveId (AbstractDataBoundFormElementTag.java:132)
    org.springframework.web.servlet.tags.form.AbstractDataBoundFormElementTag.writeDefaultAttributes (AbstractDataBoundFormElementTag.java:116)
    org.springframework.web.servlet.tags.form.AbstractHtmlElementTag.writeDefaultAttributes (AbstractHtmlElementTag.java:422)
```

提交表单验证的方法:


```

@RequestMapping("/val2.html")
public String test(@Validated @ModelAttribute("empModel") Employee emp,
                  BindingResult result, Model model) {
    //如果有验证错误 返回到 form 页面
    if (result.hasErrors()) {
        //经过 goVal2 方法的目的是为了设置 empModel 属性
        //如果 empModel 属性没有设置，页面就报错了
        return goVal2(model);
    }
    return "/index.jsp";
}

```

12.1.3 数据校验信息国际化

国际化就是根据浏览器默认语言的不同，显示不同的提示信息：

在 Employee 中，给 id 字段添加验证信息：

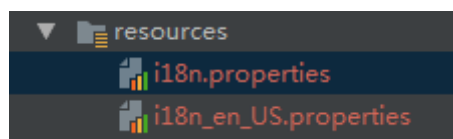
```

public class Employee {
    @NotNull(message="{NotNull.emp.id}")
    private Integer id;
}

```

{NotNull.emp.id}是从 properties 文件中读取属性值

在 resources 目录下创建两个文件，一个用来存放中文提示信息，一个存放英文提示信息：



注意文件的命名，xx.properties 对应的英文配置文件是 xx_en_US.properties

i18n.properties

```
NotNull.emp.id=id 不能为空
```

i18n_en_US.properties

```
NotNull.emp.id=userId can not be null
```

两个文件的 key 是对应的，值是不同的语言

springMVC-servlet.xml 中配置，其中 id=validator 的 bean 前面已经配置过了，这里再添加一个属性即可

```

<!-- 配置校验器 -->
<bean id="validator"
class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">
    <!-- 校验器，使用 hibernate 校验器 -->
    <property name="providerClass"
value="org.hibernate.validator.HibernateValidator"/>
    <!--这里添加一个校验信息的数据源-->
    <property name="validationMessageSource" ref="messageSource" />
</bean>
<!--自动装配校验器-->
<mvc:annotation-driven validator="validator"/>

```

```
<bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
  <property name="basenames">
    <list>
      <!-- 在 web 环境中一定要定位到 classpath 否则默认到当前 web 应用下找 -->
      <value>classpath:il8n</value>
      <value>classpath:org/hibernate/validator/ValidationMessages</value>
    </list>
  </property>
</bean>
```

使用 12.1.2 节中的测试代码，运行结果：

name :

salary :

id不能为空

将浏览器语言切换成英文，刷新页面：

name :

salary :

userId can not be null

12.2 数据格式化

12.2.1 使用注解格式化

springMVC 在映射 Date 类型的属性时会报错：

如果属性是封装在实体类中的，可以使用@DateTimeFormat 注解，如 Employee 中的 hireDate 属性。

```
@DateTimeFormat(pattern = "yyyy-MM-dd")
private Date hireDate;
```

12.2.2 initBinder 实现格式化

@DateTimeFormat 是在实体类中格式化日期类型的属性，所有的 Controller 中用到该实体类都会自动使用注解定义的格式是格式化数据。除此之外，还可以使用 @InitBinder 在 Controller 中自定义格式化：

```
package controller;

import org.springframework.beans.propertyeditors.CustomDateEditor;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.ServletRequestDataBinder;
import org.springframework.web.bind.annotation.InitBinder;
import org.springframework.web.bind.annotation.RequestMapping;

import java.text.SimpleDateFormat;
import java.util.Date;

@Controller
public class DateFormatController {
    //自定义格式化
    @InitBinder
    public void initBinder(ServletRequestDataBinder binder) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        binder.registerCustomEditor(Date.class,
            new CustomDateEditor(dateFormat, true));
    }

    @RequestMapping("/date.html")
    public String date(Date date) {
        System.out.println(date);
        return "/format.jsp";
    }
}
```

@InitBinder 定义的格式化规则对当前 Controller 有效。

12.2.3 自定义格式化

springMVC 有多种方式实现自定义数据格式化，假设现在输入一个电话号码，格式是 010-12345678，我们有一个实体类，把区号和电话号码分开：

```
package pojo;

public class PhoneNumModel {
    private String areaCode;//区号
    private String phoneNumber;//电话号码
}
```

```
//getter/setter 方法略
}
```

当请求中传入一个 String 类型的参数 “010-12345678”，通过 springMVC 的数据格式化，可以把 String 转换成实体类 PhoneNumModel

第一种方式：定义一个转换工具类，继承 PropertyEditorSupport

```
package util;

import org.springframework.core.convert.converter.Converter;
import org.springframework.util.StringUtils;
import pojo.PhoneNumModel;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PhoneNumConverter implements Converter<String, PhoneNumModel> {
    //正则表达式，定义数据规则
    Pattern pattern = Pattern.compile("^((\\d{3,4})-(\\d{7,8}))$");

    @Override
    public PhoneNumModel convert(String s) {
        if (s == null || !StringUtils.hasLength(s)) {
            return null; //如果没值，设值为 null
        }
        Matcher matcher = pattern.matcher(s);
        if (matcher.matches()) {
            PhoneNumModel phoneNumber = new PhoneNumModel();
            phoneNumber.setAreaCode(matcher.group(1));
            phoneNumber.setPhoneNumber(matcher.group(2));
            return phoneNumber;
        } else {
            throw new IllegalArgumentException(String.format("类型转换失败，需要格式[010-12345678]，但格式是[%s]", s));
        }
    }
}
```

Controller 中使用 @InitBinder 注册自定义转换器：

```
package controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.InitBinder;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
```

```

import org.springframework.web.servlet.ModelAndView;
import pojo.PhoneNumModel;
import util.PhoneNumEditor;

@Controller
public class MyBinderController {

    //自定义格式化
    @InitBinder
    public void initBinder(WebDataBinder binder) {
        binder.registerCustomEditor(PhoneNumModel.class, new PhoneNumEditor());
    }

    @RequestMapping("/phone.html")//注意一定要写@RequestParam
    public ModelAndView phone(@RequestParam("phone") PhoneNumModel phone) {
        //绑定成功时可以看到输出
        System.out.println(phone.getAreaCode());
        System.out.println(phone.getPhoneNumber());
        return new ModelAndView("/format.jsp", "phone", phone);
    }
}

```

测试分 url:

<http://localhost:8080/phone.html?phone=010-1234567>

通过@InitBinder 的方式，数据绑定规则只对当前 Controller 有效

在 Spring4.2 之后提出了一种新的转换方式，工具类实现 Converter 接口：

```

package util;

import org.springframework.core.convert.converter.Converter;
import org.springframework.util.StringUtils;
import pojo.PhoneNumModel;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PhoneNumConverter implements Converter<String, PhoneNumModel> {
    //正则表达式，定义数据规则
    Pattern pattern = Pattern.compile("^((\\d{3,4})-(\\d{7,8}))$");

    @Override
    public PhoneNumModel convert(String s) {
        if (s == null || !StringUtils.hasLength(s)) {
            return null; //如果没值，设值为 null
        }
        Matcher matcher = pattern.matcher(s);
    }
}

```

```

    if (matcher.matches()) {
        PhoneNumModel phoneNumber = new PhoneNumModel();
        phoneNumber.setAreaCode(matcher.group(1));
        phoneNumber.setPhoneNumber(matcher.group(2));
        return phoneNumber;
    } else {
        throw new IllegalArgumentException(String.format("类型转换失败，需要格式[010-12345678]，但格式是[%s]", s));
    }
}
}

```

在 springMVC 配置文件中注册自定义转换器：

```

<!-- 需要将转换器设置给注解驱动 -->
<mvc:annotation-driven conversion-service="conversionServiceFactoryBean"/>
<bean id="conversionServiceFactoryBean"
    class="org.springframework.context.support.ConversionServiceFactoryBean">
    <property name="converters">
        <set>
            <bean class="util.PhoneNumConverter"/>
        </set>
    </property>
</bean>

```

Controller 中不需要写 @InitBinder 了。这个配置对所有 Controller 都生效。

13. 其它注解

13.1 @CookieValue

```

@RequestMapping("cookie.html")
public String cookie(@CookieValue(value = "JSESSIONID", defaultValue = "mysession")
String jsessionId) {
    System.out.println(jsessionId);
    return "/index.jsp";
}

```

@CookieValue 用于获取 cookie 信息。value 用于指定 cookie 的名字，defaultValue 是当对应的 cookie 为空时系统设置的默认值。required 设置为 true 表示必须。

上面的代码如果浏览器中没有 cookie，会输出 mysession。再次刷新页面，就会打印出当前的 jsessionid，如：

名称	域名	路径	过期时间	最后访问	值
JSESSIONID	localhost	/	会话	Sun, 15 Oct 2017 03:2...	0846E8413299CB69A80492F5FB86CB19

13.2 @Value

@Value 可以实现从配置文件中读取数据并注册给 Controller

在 springMVC 配置文件种加载 properties 文件：

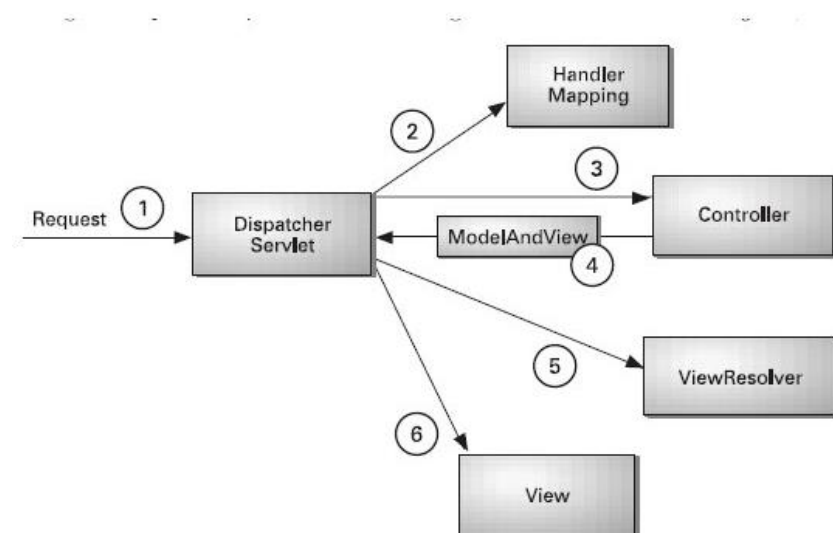
```
<context:property-placeholder location="classpath:*.properties"/>
```

测试代码，这里读取的是 12.1.3 中配置文件中的 key

```
//从配置文件中读取属性
@Value("${NotNull.emp.id}")
private String NOT_NULL_ID;

@RequestMapping("value.html")
public String value() {
    System.out.println(NOT_NULL_ID);
    return "/index.jsp";
}
```

14. 数据绑定流程



1) ApplicationContext 初始化时建立所有 url 和 controller 类的对应关系 (用 Map 保存)；

2) 根据请求 url 找到对应的 controller, 并从 controller 中找到处理请求的方法

3)Spring MVC 主框架将 ServletRequest 对象及目标方法的入参实例传递给 WebDataBinderFactory 实例，以创建 DataBinder 实例对象

4)DataBinder 是数据绑定的核心部件，调用装配在 Spring MVC 上下文中的 ConversionService 组件进行数据类型转换、数据格式化工作。将 Servlet 中的请求信息填充到入参对象中

5)调用 Validator 组件对已经绑定了请求消息的入参对象进行数据合法性校验，并最终生成数据绑定结果 BindingData 对象

6)Spring MVC 抽取 BindingResult 中的入参对象和校验错误对象，将它们赋给处理方法的响应入参。