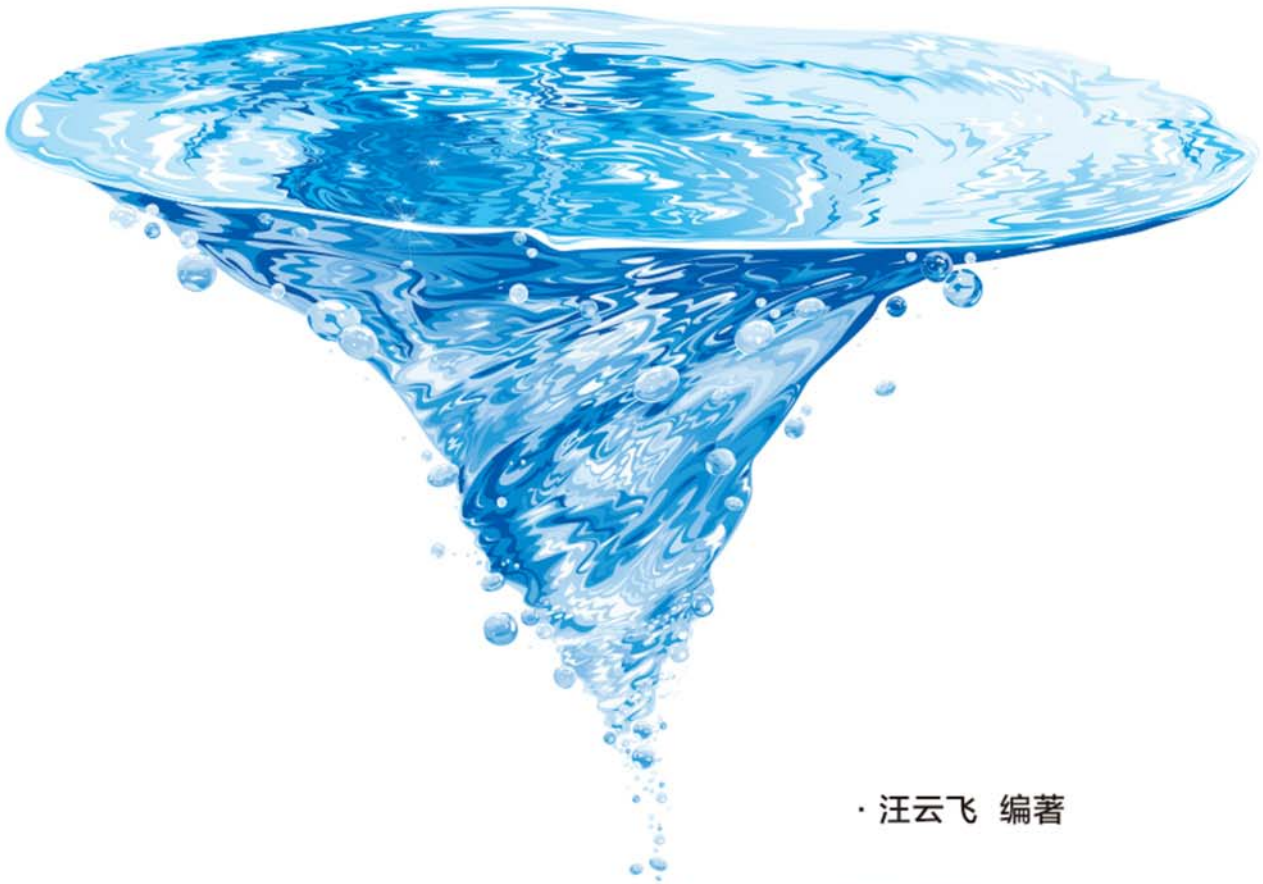
Broadview®
www.broadview.com.cn



· 汪云飞 编著

Java EE开发的颠覆者

Spring Boot 实战

本书每个章节的基本架构都是：点睛+实战

点睛：用最简练的语言去描述当前的技术
实战：对当前技术进行实战意义的代码演示

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

内 容 简 介

在当今Java EE 开发中，Spring 框架是当之无愧的王者。而Spring Boot 是Spring 主推的基于“习惯优于配置”的原则，让你能够快速搭建应用的框架，从而使得Java EE 开发变得异常简单。

本书从Spring 基础、Spring MVC 基础讲起，从而无难度地引入Spring Boot 的学习。涵盖使用Spring Boot 进行Java EE 开发的绝大数应用场景，包含：Web 开发、数据访问、安全控制、批处理、异步消息、系统集成、开发与部署、应用监控、分布式系统开发等。

当你学完本书后，你将能使用Spring Boot 解决Java EE 开发中所遇到的绝大多数问题。

图书在版编目（CIP）数据

Java EE 开发的颠覆者：Spring Boot 实战 / 汪云飞编著. —北京：电子工业出版社，2016.3
ISBN 978-7-121-28208-9

Ⅰ. ①J… Ⅱ. ①汪… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP 数据核字(2016)第037759 号

责任编辑：安 娜
印　　刷：北京中新伟业印刷有限公司
装　　订：河北省三河市路通装订厂
出版发行：电子工业出版社
　　北京市海淀区万寿路173 信箱　　邮编：100036
开　　本：787×980　1/16　印张：32.75　字数：675 千字
版　　次：2016 年3 月第1 版
印　　次：2016 年3 月第1 次印刷
印　　数：3000 册 定价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

我有将平时工作所悟写成博客以记录的习惯，随着逐渐的积累，终于可以形成目前这样一本实战性的手册。我平时在阅读大量的 Spring 相关书籍的时候发现：很多书籍对知识的讲解一味求全求深，导致读者很难快速掌握某一项技术，且因为求全求深而忽略了最佳实践，让读者云里雾里，甚至半途而废。

所以本书的每个章节的基本架构都是：点睛+实战。

点睛：用最简练的语言去描述当前的技术；

实战：对当前技术进行实战意义的代码演示。

本书代码的另一个特点是：技术相关，业务不相关。在本书的实战例子中不会假设一个业务需求，然后让读者既要理解技术，又要理解假设的业务，本书的目标是让读者 “学习时只关注技术，开发时只关注业务”。

本书涉及的技术比较广，尤其是第三部分：实战 Spring Boot，这让我很难在一本书中对每一项技术细节都详细说明；我希望本书能为读者在相关技术应用上抛砖引玉，读者在遇到特定技术的问题时可以去学习特定技术的相关书籍。

Spring 在 Java EE 开发中是实际意义上的标准，但我们在开发 Spring 的时候可能会遇到以下让人头疼的问题：

（1）大量配置文件的定义；

（2）与第三方软件整合的技术问题。

Spring 每个新版本的推出都以减少配置作为自己的主要目标，例如：

（1）推出@Component、@Service、@Repository、@Controller 注解在类上声明 Bean；

（2）推出@Configuration、@Bean 的 Java 配置来替代 xml 配置。

在脚本语言和敏捷开发大行其道的时代，Java EE 的开发显得尤为笨重，让人误解 Java EE 开发就该如此。Spring 在提升 Java EE 开发效率的脚步上从未停止过，而 Spring Boot 的推出是具有颠覆和划时代意义的。Spring Boot 具有以下特征：

（1）遵循“习惯优于配置”原则，使用 Spring Boot 只需很少的配置，大部分时候可以使用默认配置；

（2）项目快速搭建，可无配置整合第三方框架；

（3）可完全不使用 xml 配置，只使用自动配置和 Java Config；

（4）内嵌 Servlet（如 Tomcat）容器，应用可用 jar 包运行（java –jar）；

（5）运行中应用状态的监控。

虽然 Spring Boot 给我们带来了类似于脚本语言开发的效率，但 Spring Boot 里没有使用任何让你意外的技术，完全是一个单纯的基于 Spring 的应用。如 Spring_Boot 的自动配置是通过 Spring 4.x 的@Conditional 注解来实现的，所以在学习 Spring Boot 之前，我们需要快速学习 Spring 与 Spring MVC 的基础知识。

第一部分：点睛 Spring 4.x

快速学习 Spring 4.x 的各个知识点，包括基础配置、常用配置以及高级配置，以便熟悉常用配置，并体会使用 Java 语法配置所带来的便捷。

第二部分：点睛 Spring MVC 4.x

快速学习 Spring MVC 4.1 的各个知识点，MVC 的开发是我们日常开发工作中最常打交道的，所以学习 Spring MVC 对 Spring Boot 的使用极有帮助。

第三部分：实战 Spring Boot

这部分是整本书的核心部分，每个章节都会通过讲解和实战的例子来演示 Spring Boot 在实际项目中遇到的方方面面的情况，真正达到让 Spring Boot 成为 Java EE 开发的实际解决方案。

Spring Boot 发布于 2014 年 4 月，根据知名博主 Baeldung 的调查，截至 2014 年年底，使用 Spring Boot 作为 Spring 开发方案的已有 34.1%，这是多么惊人的速度。

希望读者在阅读完本书后，能够快速替代现有的开发方式，使用 Spring Boot 进行重构，和大量配置与整合开发说再见!

本书是我的第一本技术书籍，主要目的是让读者快速上手 Spring Boot 这项颠覆性的 Java EE 开发技术，由于作者水平有限，书中纰漏之处在所难免，敬请读者批评指正。

目　　录

第一部分　点睛 Spring 4.x

第二部分 点睛 Spring MVC 4.x

第三部分 实战 Spring Boot

第 7 章

Spring Boot 的 Web 开发

Web 开发是开发中至关重要的一部分，Web 开发的核心内容主要包括内嵌 Servlet 容器和 Spring MVC。

7.1 Spring Boot 的 Web 开发支持

Spring Boot 提供了 spring-boot-starter-web 为 Web 开发予以支持，spring-boot-starter- web 为我们提供了嵌入的 Tomcat 以及 Spring MVC 的依赖。而 Web 相关的自动配置存储在 spring-boot-autoconfigure.jar 的 org.springframework.boot. autoconfigure.web 下，如图 7-1 所示。

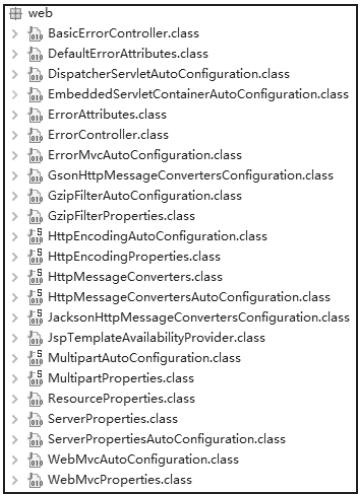


图 7-1 Web 相关的自动配置

从这些文件名可以看出：

- ServerPropertiesAutoConfiguration 和 ServerProperties 自动配置内嵌 Servlet 容器；
- HttpEncodingAutoConfiguration 和 HttpEncodingProperties 用来自动配置 http 的编码；
- MultipartAutoConfiguration 和 MultipartProperties 用来自动配置上传文件的属性；
- JacksonHttpMessageConvertersConfiguration 用来自动配置 mappingJackson2HttpMessageConverter 和 mappingJackson2XmlHttpMessage Converter；
- WebMvcAutoConfiguration 和 WebMvcProperties 配置 Spring MVC。

7.2　Thymeleaf 模板引擎

本书前面的内容很少用到页面模板引擎相关的内容，偶尔使用了 JSP 页面，但是尽可能少地涉及 JSP 相关知识，这是因为 JSP 在内嵌的 Servlet 的容器上运行有一些问题（内嵌 Tomcat、Jetty 不支持以 jar 形式运行 JSP，Undertow 不支持 JSP）。

Spring Boot 提供了大量模板引擎，包含括 FreeMarker、Groovy、Thymeleaf、Velocity 和 Mustache，Spring Boot 中推荐使用 Thymeleaf 作为模板引擎，因为 Thymeleaf 提供了完美的 Spring MVC 的支持。

7.2.1　Thymeleaf 基础知识

Thymeleaf 是一个 Java 类库，它是一个 xml/xhtml/html5 的模板引擎，可以作为 MVC 的 Web 应用的 View 层。

Thymeleaf 还提供了额外的模块与 Spring MVC 集成，所以我们可以使用 Thymeleaf 完全替代 JSP。

下面我们演示日常工作中常用的 Thymeleaf 用法，我们将把本节的内容在 7.2.4 节运行演示。

1. 引入 Thymeleaf

下面的代码是一个基本的 Thymeleaf 模板页面，在这里我们引入了 Bootstrap（作为样式控制）和 jQuery（DOM 操作），当然它们不是必需的：

```
<html xmlns:th="http://www.thymeleaf.org"><!-- 1 -->
```

```
  <head>
     <meta content="text/html;charset=UTF-8"/>
    <link th:src="@{bootstrap/css/bootstrap.min.css}" rel="stylesheet"/> <!-- 2
-->
    <link th:src="@{bootstrap/css/bootstrap-theme.min.css}"
rel="stylesheet"/><!-- 2 -->
  </head>
  <body>

  <script th:src="@{jquery-1.10.2.min.js}" type="text/javascript"></script><!--
2 -->
  <script th:src="@{bootstrap/js/bootstrap.min.js}"></script><!-- 2 -->
  </body>
 </html>
```

代码解释

① 通过 xmlns:th=http://www.thymeleaf.org 命名空间，将镜头页面转换为动态的视图。需要进行动态处理的元素将使用“th:”为前缀；

② 通过“@{}”引用 Web 静态资源，这在 JSP 下是极易出错的。

2. 访问 model 中的数据

通过“${}”访问 model 中的属性，这和 JSP 极为相似。

```
  <div class="panel panel-primary">
    <div class="panel-heading">
       <h3 class="panel-title">访问 model</h3>
    </div>
    <div class="panel-body">
        <span th:text="${singlePerson.name}"></span>
    </div>
  </div>
```

代码解释

使用访问 model 中的 singlePerson 的 name 属性。注意：需要处理的动态内容需要加上“th:”前缀。

3. model 中的数据迭代

Thymeleaf 的迭代和 JSP 的写法也很相似，代码如下：

```
<div class="panel panel-primary">
  <div class="panel-heading">
     <h3 class="panel-title">列表</h3>
  </div>
  <div class="panel-body">
     <ul class="list-group">
        <li class="list-group-item" th:each="person:${people}">
           <span th:text="${person.name}"></span>
             <span th:text="${person.age}"></span>
        </li>
     </ul>
  </div>
</div>
```

代码解释

使用 th:each 来做循环迭代（th:each=*"person:${people}"*），person 作为迭代元素来使用，然后像上面一样访问迭代元素中的属性。

4. 数据判断

代码如下：

```
<div th:if="${not #lists.isEmpty(people)}">
    <div class="panel panel-primary">
      <div class="panel-heading">
         <h3 class="panel-title">列表</h3>
      </div>
      <div class="panel-body">
         <ul class="list-group">
            <li class="list-group-item" th:each="person:${people}">
               <span th:text="${person.name}"></span>
                 <span th:text="${person.age}"></span>
            </li>
         </ul>
      </div>
    </div>
</div>
```

代码解释

通过${not #lists.isEmpty(people)}表达式判断 people 是否为空。Thymeleaf 支持 **>**、**<**、**>=**、**<=**　**==**、**!=** 作为比较条件，同时也支持将 SpringEL 表达式语言用于条件中。

5. 在 JavaScript 中访问 model

在项目中，我们经常需要在 JavaScript 访问 model 中的值，在 Thymeleaf 里实现代码如下：

```
<script th:inline="javascript">
    var single = [[${singlePerson}]];
    console.log(single.name+"/"+single.age)
</script>
```

代码解释

- 通过 th:inline="*javascript*"添加到 script 标签，这样 JavaScript 代码即可访问 model 中的属性；
- 通过“[[${}]]”格式获得实际的值。

还有一种是需要在 html 的代码里访问 model 中的属性，例如，我们需要在列表后单击每一行后面的按钮获得 model 中的值，可做如下处理：

```
<li class="list-group-item" th:each="person:${people}">
   <span th:text="${person.name}"></span>
    <span th:text="${person.age}"></span>
    <button class="btn" th:onclick="'getName(\'' + ${person.name} + '\');'">获得名字</button>
</li>
```

代码解释

注意格式 th:onclick="'getName(\'' + ${person.name} + '\');'"。

6. 其他知识

更多更完整的 Thymeleaf 的知识，请查看 http://www.thymeleaf.org 的官网。

7.2.2 与 Spring MVC 集成

在 Spring MVC 中，若我们需要集成一个模板引擎的话，需要定义 ViewResolver，而 ViewResolver 需要定义一个 View，如 4.2.2 节中我们为 JSP 定义的 ViewResolver 的代码：

```
@Bean
    public InternalResourceViewResolver viewResolver(){
        InternalResourceViewResolver viewResolver = new
InternalResourceViewResolver();
        viewResolver.setPrefix("/WEB-INF/classes/views/");
```

```
        viewResolver.setSuffix(".jsp");
        viewResolver.setViewClass(JstlView.class);
        return viewResolver;
    }
```

通过上面的代码可以看出，使用 JsltView 定义了一个 InternalResourceViewResolver，因而使用 Thymeleaf 作为我们的模板引擎也应该做类似的定义。庆幸的是，Thymeleaf 为我们定义好了 org.thymeleaf.spring4.view.ThymeleafView 和 org.thymeleaf.spring4.view.ThymeleafViewResolver（默认使用 ThymeleafView 作为 View）。Thymeleaf 给我们提供了一个 SpringTemplateEngine 类，用来驱动在 Spring MVC 下使用 Thymeleaf 模板引擎，另外还提供了一个 TemplateResolver 用来设置通用的模板引擎（包含前缀、后缀等），这使我们在 Spring MVC 中集成 Thymeleaf 引擎变得十分简单，代码如下：

```
   @Bean
   public  TemplateResolver templateResolver(){
    TemplateResolver templateResolver = new ServletContextTemplateResolver();
    templateResolver.setPrefix("/WEB-INF/templates");
    templateResolver.setSuffix(".html");
    templateResolver.setTemplateMode("HTML5");
    return templateResolver;

   }

   @Bean
   public SpringTemplateEngine templateEngine(){
    SpringTemplateEngine templateEngine = new SpringTemplateEngine();
    templateEngine.setTemplateResolver(templateResolver());
    return templateEngine;
   }

   @Bean
   public ThymeleafViewResolver thymeleafViewResolver(){
    ThymeleafViewResolver thymeleafViewResolver = new ThymeleafViewResolver();
    thymeleafViewResolver.setTemplateEngine(templateEngine());
//      thymeleafViewResolver.setViewClass(ThymeleafView.class);
    return thymeleafViewResolver;
   }
```

7.2.3　Spring Boot 的 Thymeleaf 支持

在上一节我们讲述了 Thymeleaf 与 Spring MVC 集成的配置，讲述的目的是为了方便大家

理解 Spring MVC 和 Thymeleaf 集成的原理。但在 Spring Boot 中这一切都是不需要的，Spring Boot 通过 org.springframework.boot.autoconfigure.thymeleaf 包对 Thymeleaf 进行了自动配置，如图 7-2 所示。

图 7-2　thymeleaf 包

通过 ThymeleafAutoConfiguration 类对集成所需要的 Bean 进行自动配置，包括 templateResolver、templateEngine 和 thymeleafViewResolver 的配置。

通过 ThymeleafProperties 来配置 Thymeleaf，在 application.properties 中，以 spring.thymeleaf 开头来配置，通过查看 ThymeleafProperties 的主要源码，我们可以看出如何设置属性以及默认配置：

```
@ConfigurationProperties("spring.thymeleaf")
public class ThymeleafProperties {

    public static final String DEFAULT_PREFIX = "classpath:/templates/";

    public static final String DEFAULT_SUFFIX = ".html";

    /**
     *前缀设置，Spring Boot 默认模板，防止在 classpath:/templates/ 目录下
     */
    private String prefix = DEFAULT_PREFIX;

    /**
     * 后缀设置，默认为 html
     */
    private String suffix = DEFAULT_SUFFIX;

    /**
     * 模板模式设置，默认为 HTML 5
     */
    private String mode = "HTML5";

    /**
     * 模板的编码设置，默认为 UTF-8
     */
```

```
    private String encoding = "UTF-8";

    /**
     * 模板的媒体类型设置，默认为 text/html.
     */
    private String contentType = "text/html";

    /**
     * 是否开启模板缓存，默认是开启，开发时请关闭
     */
    private boolean cache = true;
    //….
}
```

7.2.4　实战

1. 新建 Spring Boot 项目

选择 Thymeleaf 依赖，spring-boot-starter-thymeleaf 会自动包含 spring-boot-starter-web，如图 7-3 所示。

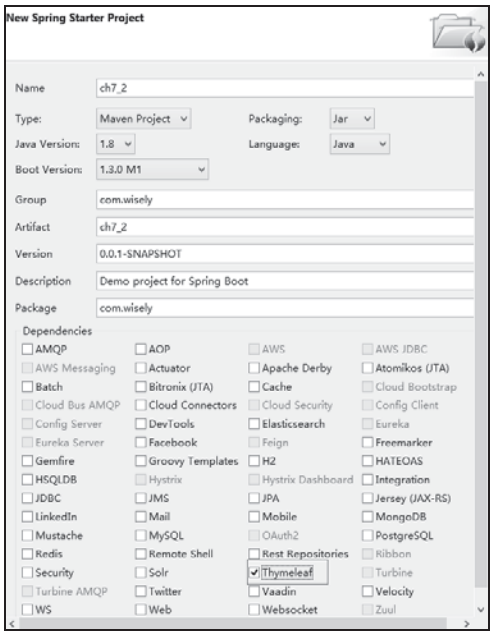


图 7-3　新建 Spring Boot 项目

2. 示例 JavaBean

此类用来在模板页面展示数据用，包含 name 属性和 age 属性：

```
package com.wisely;

public class Person {
    private String name;
    private Integer age;

    public Person() {
        super();
    }
    public Person(String name, Integer age) {
        super();
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Integer getAge() {
        return age;
    }
    public void setAge(Integer age) {
        this.age = age;
    }
}
```

3. 脚本样式静态文件

根据默认原则，脚本样式、图片等静态文件应放置在 src/main/resources/static 下，这里引入了 Bootstrap 和 jQuery，结构如图 7-4 所示。

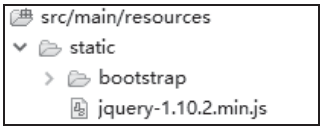


图 7-4　文件位置

4. 演示页面

根据默认原则，页面应放置在 src/main/resources/templates 下。在 src/main/resources/templates 下新建 index.html，如图 7-5 所示。

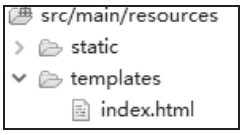


图 7-5　新建 index.html

代码如下：

```
<html xmlns:th="http://www.thymeleaf.org">
  <head>
     <meta content="text/html;charset=UTF-8"/>
     <meta http-equiv="X-UA-Compatible" content="IE=edge"/>
    <meta name="viewport" content="width=device-width, initial-scale=1"/>
   <link th:href="@{bootstrap/css/bootstrap.min.css}" rel="stylesheet"/>
   <link th:href="@{bootstrap/css/bootstrap-theme.min.css}" rel="stylesheet"/>
  </head>
  <body>

  <div class="panel panel-primary">
    <div class="panel-heading">
       <h3 class="panel-title">访问 model</h3>
    </div>
    <div class="panel-body">
        <span th:text="${singlePerson.name}"></span>
    </div>
  </div>

  <div th:if="${not #lists.isEmpty(people)}">
      <div class="panel panel-primary">
        <div class="panel-heading">
           <h3 class="panel-title">列表</h3>
        </div>
        <div class="panel-body">
           <ul class="list-group">
                <li class="list-group-item" th:each="person:${people}">
                   <span th:text="${person.name}"></span>
                    <span th:text="${person.age}"></span>
```

```
                    <button class="btn" th:onclick="'getName(\'' + ${person.name}
+ '\');'">获得名字</button>
                </li>
            </ul>
        </div>
    </div>
</div>

  <script th:src="@{jquery-1.10.2.min.js}" type="text/javascript"></script><!--
2 -->
  <script th:src="@{bootstrap/js/bootstrap.min.js}"></script><!-- 2 -->

  <script th:inline="javascript">
    var single = [[${singlePerson}]];
    console.log(single.name+"/"+single.age)

    function getName(name){
        console.log(name);
    }
  </script>

  </body>
</html>
```

5. 数据准备

代码如下：

```
package com.wisely;

import java.util.ArrayList;
import java.util.List;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
@SpringBootApplication
public class Ch72Application {

    @RequestMapping("/")
    public String index(Model model){
```

```
        Person single = new Person("aa",11);

        List<Person> people = new ArrayList<Person>();
        Person p1 = new Person("xx",11);
        Person p2 = new Person("yy",22);
        Person p3 = new Person("zz",33);
        people.add(p1);
        people.add(p2);
        people.add(p3);

        model.addAttribute("singlePerson", single);
        model.addAttribute("people", people);

        return "index";
    }

    public static void main(String[] args) {
        SpringApplication.run(Ch72Application.class, args);
    }
}
```

6. 运行

访问 http://localhost:8080，效果如图 7-6 所示。

单击“获得名字”选项，效果如图 7-7 所示。

图 7-6　访问 http://localhost：8080

图 7-7　单击获得名字

7.3 Web 相关配置

7.3.1 Spring Boot 提供的自动配置

通过查看 WebMvcAutoConfiguration 及 WebMvcProperties 的源码，可以发现 Spring Boot 为我们提供了如下的自动配置。

1. 自动配置的 ViewResolver

（1）ContentNegotiatingViewResolver

这是 Spring MVC 提供的一个特殊的 ViewResolver，ContentNegotiatingViewResolver 不是自己处理 View，而是代理给不同的 ViewResolver 来处理不同的 View，所以它有最高的优先级。

（2）BeanNameViewResolver

在控制器（@Controller）中的一个方法的返回值的字符串（视图名）会根据 BeanNameViewResolver 去查找 Bean 的名称为返回字符串的 View 来渲染视图。是不是不好理解，下面举个小例子。

定义 BeanNameViewResolver 的 Bean：

```
@Bean
        public BeanNameViewResolver beanNameViewResolver() {
            BeanNameViewResolver resolver = new BeanNameViewResolver();
            return resolver;
        }
```

定义一个 View 的 Bean，名称为 jsonView：

```
@Bean
    public MappingJackson2JsonView jsonView(){
        MappingJackson2JsonView jsonView = new MappingJackson2JsonView();
        return jsonView;
    }
```

在控制器中，返回值为字符串 jsonView，它会找 Bean 的名称为 jsonView 的视图来渲染：

```
@RequestMapping(value = "/json",produces={MediaType.APPLICATION_JSON_VALUE})
    public String json(Model model) {
        Person single = new Person("aa",11);
        model.addAttribute("single", single);
```

```
        return "jsonView";
    }
```

（3）InternalResourceViewResolver

这个是一个极为常用的 ViewResolver，主要通过设置前缀、后缀，以及控制器中方法来返回视图名的字符串，以得到实际页面，Spring Boot 的源码如下：

```
       @Bean
        @ConditionalOnMissingBean(InternalResourceViewResolver.class)
        public InternalResourceViewResolver defaultViewResolver() {
            InternalResourceViewResolver resolver = new
InternalResourceViewResolver();
            resolver.setPrefix(this.prefix);
            resolver.setSuffix(this.suffix);
            return resolver;
        }
```

2. 自动配置的静态资源

在自动配置类的 addResourceHandlers 方法中定义了以下静态资源的自动配置。

（1）类路径文件

把类路径下的/static、/public、/resources 和 /META-INF/resources 文件夹下的静态文件直接映射为/**，可以通过 http://localhost:8080/**来访问。

（2）webjar

何谓 webjar，webjar 就是将我们常用的脚本框架封装在 jar 包中的 jar 包，更多关于 webjar 的内容请访问 http://www.webjars.org 网站。

把 webjar 的/META-INF/resources/webjars/下的静态文件映射为/webjar/**，可以通过 http://localhost:8080/webjar/**来访问。

3. 自动配置的 Formatter 和 Converter

关于自动配置 Formatter 和 Converter，我们可以看一下 WebMvcAutoConfiguration 类中的定义：

```
    @Override
        public void addFormatters(FormatterRegistry registry) {
            for (Converter<?, ?> converter : getBeansOfType(Converter.class)){
```

```
                registry.addConverter(converter);
            }

            for (GenericConverter converter :
getBeansOfType(GenericConverter.class)) {
                registry.addConverter(converter);
            }

            for (Formatter<?> formatter : getBeansOfType(Formatter.class)) {
                registry.addFormatter(formatter);
            }
        }

        private <T> Collection<T> getBeansOfType(Class<T> type) {
            return this.beanFactory.getBeansOfType(type).values();
        }
```

从代码中可以看出，只要我们定义了 Converter、GenericConverter 和 Formatter 接口的实现类的 Bean，这些 Bean 就会自动注册到 Spring MVC 中。

4. 自动配置的 HttpMessageConverters

在 WebMvcAutoConfiguration 中，我们注册了 messageConverters，代码如下：

```
@Autowired
        private HttpMessageConverters messageConverters;

        @Override
        public void configureMessageConverters(List<HttpMessageConverter<?>>
converters) {
            converters.addAll(this.messageConverters.getConverters());
        }
```

在这里直接注入了 HttpMessageConverters 的 Bean，而这个 Bean 是在 HttpMessageConvertersAutoConfiguration 类中定义的，我们自动注册的 HttpMessage Converter 除了 Spring MVC 默认的 ByteArrayHttpMessageConverter、StringHttpMessage Converter、Resource HttpMessageConverter、SourceHttpMessageConverter、AllEncompassing FormHttpMessageConverter 外，在我们的 HttpMessageConverters AutoConfiguration 的自动配置文件里还引入了 JacksonHttpMessageConverters Configuration 和 GsonHttpMessage ConverterConfiguration，使我们获得了额外的 HttpMessageConverter：

- 若 jackson 的 jar 包在类路径上，则 Spring Boot 通过 JacksonHttpMessage Converters Configuration 增加 MappingJackson2HttpMessage Converter 和 Mapping Jackson2 XmlHttpMessageConverter；
- 若 gson 的 jar 包在类路径上，则 Spring Boot 通过 GsonHttpMessageConverter Configuration 增加 GsonHttpMessageConverter。

在 Spring Boot 中，如果要新增自定义的 HttpMessageConverter，则只需定义一个你自己的 HttpMessageConverters 的 Bean，然后在此 Bean 中注册自定义 HttpMessageConverter 即可，例如：

```
@Bean
 public HttpMessageConverters customConverters() {
    HttpMessageConverter<?> customConverter1= new CustomConverter1();
    HttpMessageConverter<?> customConverter2= new CustomConverter2();
    return new HttpMessageConverters(customConverter1, customConverter2);
 }
```

5. 静态首页的支持

把静态 index.html 文件放置在如下目录。

- classpath:/META-INF/resources/index.html
- classpath:/resources/index.html
- classpath:/static/index.html
- classpath:/public/index.html

当我们访问应用根目录 http://localhost:8080/时，会直接映射。

7.3.2　接管 Spring Boot 的 Web 配置

如果 Spring Boot 提供的 Spring MVC 不符合要求，则可以通过一个配置类（注解有@Configuration 的类）加上@EnableWebMvc 注解来实现完全自己控制的 MVC 配置。

当然，通常情况下，Spring Boot 的自动配置是符合我们大多数需求的。在你既需要保留 Spring Boot 提供的便利，又需要增加自己的额外的配置的时候，可以定义一个配置类并继承 WebMvcConfigurerAdapter，无须使用@EnableWebMvc 注解，然后按照第 4 章讲解的 Spring MVC 的配置方法来添加 Spring Boot 为我们所做的其他配置，例如：

```
@Configuration
public class WebMvcConfig  extends WebMvcConfigurerAdapter{
```

```
    @Override
      public void addViewControllers(ViewControllerRegistry registry) {
         registry.addViewController("/xx").setViewName("/xx");
      }

}
```

值得指出的是，在这里重写的 addViewControllers 方法，并不会覆盖 WebMvcAutoConfiguration 中的 addViewControllers（在此方法中，Spring Boot 将“/”映射至 index.html），这也就意味着我们自己的配置和 Spring Boot 的自动配置同时有效，这也是我们推荐添加自己的 MVC 配置的方式。

7.3.3 注册 Servlet、Filter、Listener

当使用嵌入式的 Servlet 容器（Tomcat、Jetty 等）时，我们通过将 Servlet、Filter 和 Listener 声明为 Spring Bean 而达到注册的效果；或者注册 ServletRegistrationBean 、FilterRegistrationBean 和 ServletListenerRegistrationBean 的 Bean。

（1）直接注册 Bean 示例，代码如下：

```
  @Bean
public XxServlet xxServlet (){
       return new XxServlet();
   }
   @Bean
   public YyFilter yyFilter (){
       return new YyFilter();
   }

   @Bean
   public ZzListener zzListener (){
       return new ZzListener();
   }
```

（2）通过 RegistrationBean 示例：

```
  @Bean
    public ServletRegistrationBean servletRegistrationBean(){
       return new ServletRegistrationBean(new XxServlet(),"/xx/*");
   }
   @Bean
   public FilterRegistrationBean filterRegistrationBean(){
```

```
        FilterRegistrationBean registrationBean = new FilterRegistrationBean();
        registrationBean.setFilter( new YyFilter());
        registrationBean.setOrder(2);
        return registrationBean;
    }

    @Bean
    public ServletListenerRegistrationBean<ZzListener>
zzListenerServletRegistrationBean(){
        return new ServletListenerRegistrationBean<ZzListener>(new
ZzListener());
    }
```

7.4　Tomcat 配置

本节虽然叫 Tomcat 配置，但其实指的是 servlet 容器的配置，因为 Spring Boot 默认内嵌的 Tomcat 为 servlet 容器，所以本节只讲对 Tomcat 配置，其实本节的配置对 Tomcat、Jetty 和 Undertow 都是通用的。

7.4.1　配置 Tomcat

关于 Tomcat 的所有属性都在 org.springframework.boot.autoconfigure.web. ServerProperties 配置类中做了定义，我们只需在 application.properties 配置属性做配置即可。通用的 Servlet 容器配置都以“server”作为前缀，而 Tomcat 特有配置都以“server.tomcat”作为前缀。下面举一些常用的例子。

配置 Servlet 容器:

```
server.port= #配置程序端口，默认为 8080
server.session-timeout= #用户会话 session 过期时间，以秒为单位
server.context-path= #配置访问路径，默认为/
```

配置 Tomcat：

```
server.tomcat.uri-encoding = #配置 Tomcat 编码，默认为 UTF-8
server.tomcat.compression= # Tomcat 是否开启压缩，默认为关闭 off
```

更为详细的 Servlet 容器配置及 Tomcat 配置，请查看附录 A 中以“server”和“server. tomcat”为前缀的配置。

7.4.2 代码配置 Tomcat

如果你需要通过代码的方式配置 servlet 容器，则可以注册一个实现 EmbeddedServletContainerCustomizer 接口的 Bean；若想直接配置 Tomcat、Jetty、Undertow，则可以直接定义 TomcatEmbeddedServletContainerFactory、JettyEmbeddedServletContainerFactory、UndertowEmbeddedServletContainerFactory。

1. 通用配置

（1）新建类的配置：

```
package com.wisely.ch7_4;

import java.util.concurrent.TimeUnit;

import
org.springframework.boot.context.embedded.ConfigurableEmbeddedServletContaine
r;
import
org.springframework.boot.context.embedded.EmbeddedServletContainerCustomizer;
import org.springframework.boot.context.embedded.ErrorPage;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;

@Component
public class CustomServletContainer implements
EmbeddedServletContainerCustomizer{

    @Override
    public void customize(ConfigurableEmbeddedServletContainer container) {
        container.setPort(8888); //1
        container.addErrorPages(new ErrorPage(HttpStatus.NOT_FOUND,
"/404.html"));//2
        container.setSessionTimeout(10,TimeUnit.MINUTES); //3

    }
}
```

（2）当前配置文件内配置。若要在当前已有的配置文件内添加类的 Bean 的话，则在 Spring 配置中，注意当前类要声明为 static：

```
@SpringBootApplication
```

```
public class Ch74Application {

   public static void main(String[] args) {
      SpringApplication.run(Ch74Application.class, args);
   }

   @Component
   public static class CustomServletContainer implements
EmbeddedServletContainerCustomizer{

        @Override
        public void customize(ConfigurableEmbeddedServletContainer container) {
            container.setPort(8888); //1
            container.addErrorPages(new ErrorPage(HttpStatus.NOT_FOUND,
"/404.html"));//2
            container.setSessionTimeout(10,TimeUnit.MINUTES); //3
        }

   }

}
```

2. 特定配置

下面以 Tomcat 为例（Jetty 使用 JettyEmbeddedServletContainerFactory，Undertow 使用 UndertowEmbeddedServletContainerFactory）：

```
@Bean
   public EmbeddedServletContainerFactory servletContainer() {
   TomcatEmbeddedServletContainerFactory factory = new
TomcatEmbeddedServletContainerFactory();
factory.setPort(8888); //1
factory.addErrorPages(new ErrorPage(HttpStatus.NOT_FOUND, "/404.html"));//2
   factory.setSessionTimeout(10, TimeUnit.MINUTES); //3
   return factory;
}
```

代码解释

上面两个例子的代码都实现了这些功能：

① 配置端口号；

② 配置错误页面，根据 HttpStatus 中的错误状态信息，直接转向错误页面，其中 404.html

放置在 src/main/resources/static 下即可；

③ 配置 Servlet 容器用户会话（session）过期时间。

7.4.3 替换 Tomcat

Spring Boot 默认使用 Tomcat 作为内嵌 Servlet 容器，查看 spring-boot-starter-web 依赖，如图 7-8 所示。

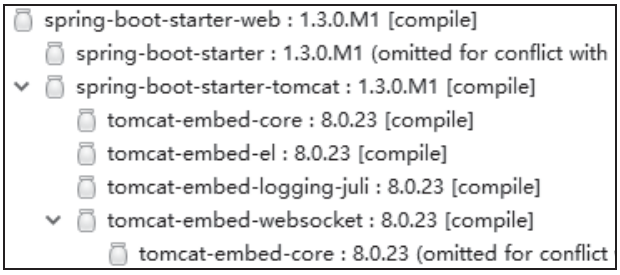


图 7-8 查看 Spring-boot-starter-web 依赖

如果要使用 Jetty 或者 Undertow 为 sevvlet 容器，只需修改 spring-boot-starter-web 的依赖即可。

1. 替换为 Jetty

在 pom.xml 中，将 spring-boot-starter-web 的依赖由 spring-boot-starter-tomcat 替换为 spring-boot-starter-Jetty：

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <exclusions>
            <exclusion>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-tomcat</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jetty</artifactId>
    </dependency>
```

此时启动 Spring Boot，控制台输出效果如图 7-9 所示。

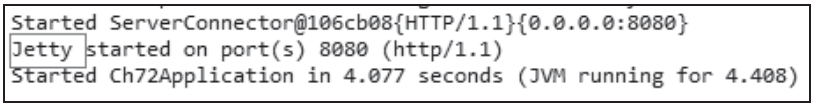


图 7-9　控制台输出效果

2. 替换为 Undertow

在 pom.xml 中，将 spring-boot-starter-web 的依赖由 spring-boot-starter-tomcat 替换为 spring-boot-starter-undertow：

```
<dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
            <exclusions>
                <exclusion>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-starter-tomcat</artifactId>
                </exclusion>
            </exclusions>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-undertow</artifactId>
    </dependency>
```

此时启动 Spring Boot，控制台输出效果如图 7-10 所示。

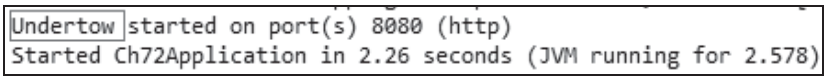


图 7-10　控制台输出效果

7.4.4　SSL 配置

SSL 的配置也是我们在实际应用中经常遇到的场景。

SSL（Secure Sockets Layer，安全套接层）是为网络通信提供安全及数据完整性的一种安全协议，SSL 在网络传输层对网络连接进行加密。SSL 协议位于 TCP/IP 协议与各种应用层协议之间，为数据通信提供安全支持。SSL 协议可分为两层： SSL 记录协议（SSL Record Protocol），它建立在可靠的传输协议（如 TCP）之上，为高层协议提供数据封装、压缩、加密等基本功能的支持。SSL 握手协议（SSL Handshake Protocol），它建立在 SSL 记录协议之上，

用于在实际数据传输开始前，通信双方进行身份认证、协商加密算法、交换加密密钥等。

而在基于 B/S 的 Web 应用中，是通过 HTTPS 来实现 SSL 的。HTTPS 是以安全为目标的 HTTP 通道，简单讲是 HTTP 的安全版，即在 HTTP 下加入 SSL 层，HTTPS 的安全基础是 SSL。

因为 Spring Boot 用的是内嵌的 Tomcat，因而我们做 SSL 配置的时候需要做如下的操作。

1. 生成证书

使用 SSL 首先需要一个证书，这个证书既可以是自签名的，也可以是从 SSL 证书授权中心获得的。本例为了演示方便，演示自授权证书的生成。

每一个 JDK 或者 JRE 里都有一个工具叫 keytool，它是一个证书管理工具，可以用来生成自签名的证书，如图 7-11 所示。

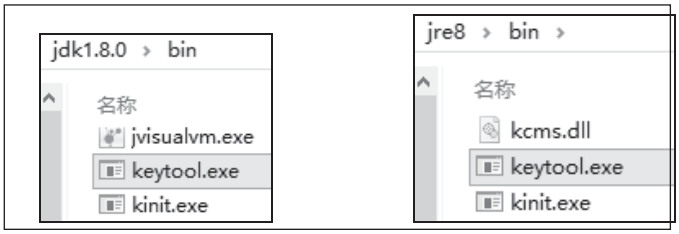


图 7-11　keytool

在配置了 JAVA_HOME，并将 JAVA_HOME 的 bin 目录加入到 Path 后，即可在控制台调用该命令，如图 7-12 所示。

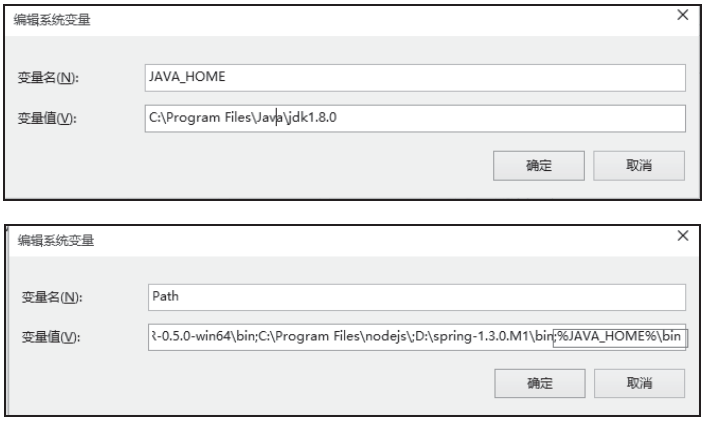


图 7-12　将 bin 目录加入到 Path

在控制台输入如下命令，然后按照提示操作，如图 7-13 所示。

```
keytool -genkey  -alias tomcat
```

图 7-13　按照提示操作

这时候我们在当前目录下生成了一个.keystore 文件，这就是我们要用的证书文件，如图 7-14 所示。

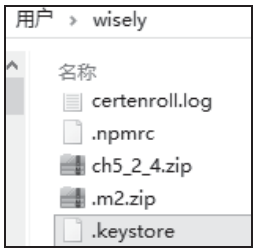


图 7-14　keystore 文件

2. Spring Boot 配置 SSL

添加一个 index.html 到 src/main/resources/static 下，作为测试。

将.keystore 文件复制到项目的根目录，然后在 application.properties 中做如下 SSL 的配置：

```
server.port = 8443
server.ssl.key-store = .keystore
server.ssl.key-store-password= 111111
server.ssl.keyStoreType= JKS
```

```
server.ssl.keyAlias: tomcat
```

此时启动 Spring Boot，控制台输出效果如图 7-15 所示。

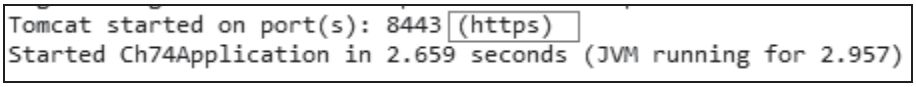


图 7-15　控制台输出效果

此时访问 https://localhost:8443，效果如图 7-16 所示。

图 7-16　访问 localhost:8443

3. http 转向 https

很多时候我们在地址栏输入的是 http，但是会自动转向到 https，例如我们访问百度的时候，如图 7-17 所示。

图 7-17　http 自动转向 https

要实现这个功能，我们需配置 TomcatEmbeddedServletContainerFactory，并且添加 Tomcat 的 connector 来实现。

这时我们需要在配置文件里增加如下配置：

```
import org.apache.catalina.Context;
import org.apache.catalina.connector.Connector;
import org.apache.tomcat.util.descriptor.web.SecurityCollection;
```

```
import org.apache.tomcat.util.descriptor.web.SecurityConstraint;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.boot.context.embedded.EmbeddedServletContainerFactory;
import
org.springframework.boot.context.embedded.tomcat.TomcatEmbeddedServletContain
erFactory;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class Ch74Application {

   public static void main(String[] args) {
      SpringApplication.run(Ch74Application.class, args);
   }

   @Bean
   public EmbeddedServletContainerFactory servletContainer() {
     TomcatEmbeddedServletContainerFactory tomcat = new
TomcatEmbeddedServletContainerFactory() {
        @Override
        protected void postProcessContext(Context context) {
          SecurityConstraint securityConstraint = new SecurityConstraint();
          securityConstraint.setUserConstraint("CONFIDENTIAL");
          SecurityCollection collection = new SecurityCollection();
          collection.addPattern("/*");
          securityConstraint.addCollection(collection);
          context.addConstraint(securityConstraint);
        }
       };

     tomcat.addAdditionalTomcatConnectors(httpConnector());
     return tomcat;
   }
   @Bean
   public Connector httpConnector() {
     Connector connector = new
Connector("org.apache.coyote.http11.Http11NioProtocol");
     connector.setScheme("http");
     connector.setPort(8080);
     connector.setSecure(false);
     connector.setRedirectPort(8443);
     return connector;
```

```
    }
}
```

此时启动 Spring Boot，控制台输出效果如图 7-18 所示。

```
Tomcat started on port(s): 8443 (https) 8080 (http)
Started Ch74Application in 2.464 seconds (JVM running for 2.786)
```

图 7-18　启动 Spring Boot

此时我们访问:http://localhost:8080，会自动转到 https://localhost:8443，如图 7-19 所示。

图 7-19　自动转到 https://localhost:8443

7.5　Favicon 配置

7.5.1　默认的 Favicon

Spring Boot 提供了一个默认的 Favicon，每次访问应用的时候都能看到，如图 7-20 所示。

7.5.2　关闭 Favicon

我们可以在 application.properties 中设置关闭 Favicon，默认为开启，如图 7-21 所示。

```
spring.mvc.favicon.enabled=false
```

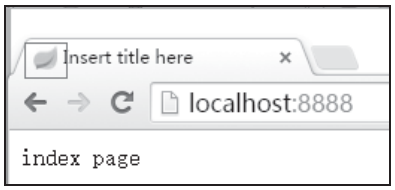


图 7-20　默认的 Favicon

图 7-21　关闭 Favicon

7.5.3　设置自己的 Favicon

若需要设置自己的 Favicon，则只需将自己的 favicon.ico（文件名不能变动）文件放置在类路径根目录、类路径 META-INF/resources/下、类路径 resources/下、类路径 static/下或类路径 public/下。这里将 favicon.ico 放置在 src/main/resources/static 下，运行效果如图 7-22 所示。

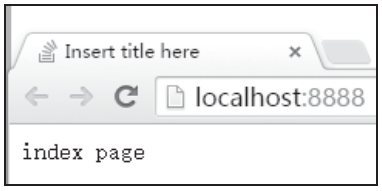


图 7-22　运行效果

7.6　WebSocket

7.6.1　什么是 WebSocket

WebSocket 为浏览器和服务端提供了双工异步通信的功能，即浏览器可以向服务端发送消息，服务端也可以向浏览器发送消息。WebSocket 需浏览器的支持，如 IE 10+、Chrome 13+、Firefox 6+，这对我们现在的浏览器来说都不是问题。

WebSocket 是通过一个 socket 来实现双工异步通信能力的。但是直接使用 WebSocket（或者 SockJS：WebSocket 协议的模拟，增加了当浏览器不支持 WebSocket 的时候的兼容支持）协议开发程序显得特别烦琐，我们会使用它的子协议 STOMP，它是一个更高级别的协议，STOMP 协议使用一个基于帧（frame）的格式来定义消息，与 HTTP 的 request 和 response 类似（具有类似于@RequestMapping 的@MessageMapping），我们会在后面实战内容中观察 STOMP 的帧。

7.6.2　Spring Boot 提供的自动配置

Spring Boot 对内嵌的 Tomcat（7 或者 8）、Jetty9 和 Undertow 使用 WebSocket 提供了支持。配置源码存于 org.springframework.boot.autoconfigure.websocket 下，如图 7-23 所示。

图 7-23　源码存放位置

Spring Boot 为 WebSocket 提供的 stater pom 是 spring-boot-starter-websocket。

7.6.3　实战

1. 准备

新建 Spring Boot 项目，选择 Thymeleaf 和 Websocket 依赖，如图 7-24 所示。

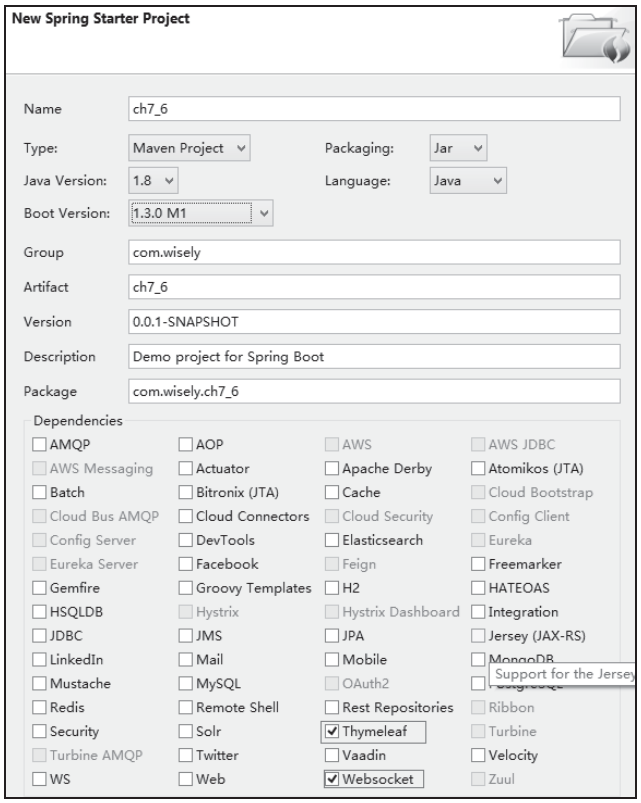


图 7-24　选择 Thymeleaf 和 Websocket

2. 广播式

广播式即服务端有消息时，会将消息发送给所有连接了当前 endpoint 的浏览器。

（1）配置 WebSocket，需要在配置类上使用@EnableWebSocketMessageBroker 开启 WebSocket 支持，并通过继承 AbstractWebSocketMessageBrokerConfigurer 类，重写其方法来配置 WebSocket。

代码如下：

```
package com.wisely.ch7_6;

import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.simp.config.MessageBrokerRegistry;
import
org.springframework.web.socket.config.annotation.AbstractWebSocketMessageBrok
erConfigurer;
import
org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker
;
import org.springframework.web.socket.config.annotation.StompEndpointRegistry;

@Configuration
@EnableWebSocketMessageBroker//1
public class WebSocketConfig extends AbstractWebSocketMessageBrokerConfigurer{

    @Override
   public void registerStompEndpoints(StompEndpointRegistry registry) { //2
      registry.addEndpoint("/endpointWisely").withSockJS(); //3
   }


   @Override
   public void configureMessageBroker(MessageBrokerRegistry registry) {//4
      registry.enableSimpleBroker("/topic"); //5
   }

}
```

代码解释

① 通过@EnableWebSocketMessageBroker 注解开启使用 STOMP 协议来传输基于代理（message broker）的消息，这时控制器支持使用@MessageMapping，就像使用@RequestMapping

一样。

② 注册 STOMP 协议的节点（endpoint），并映射的指定的 URL。

③ 注册一个 STOMP 的 endpoint，并指定使用 SockJS 协议。

④ 配置消息代理（Message Broker）。

⑤ 广播式应配置一个/topic 消息代理。

（2）浏览器向服务端发送的消息用此类接受：

```
package com.wisely.ch7_6.domain;

public class WiselyMessage {
    private String name;

    public String getName(){
        return name;
    }
}
```

（3）服务端向浏览器发送的此类的消息：

```
package com.wisely.ch7_6.domain;

public class WiselyResponse {
    private String responseMessage;
    public WiselyResponse(String responseMessage){
        this.responseMessage = responseMessage;
    }
    public String getResponseMessage(){
        return responseMessage;
    }
}
```

（4）演示控制器，代码如下：

```
package com.wisely.ch7_6.web;

import org.springframework.messaging.handler.annotation.MessageMapping;
import org.springframework.messaging.handler.annotation.SendTo;
import org.springframework.stereotype.Controller;

import com.wisely.ch7_6.domain.WiselyMessage;
```

```
import com.wisely.ch7_6.domain.WiselyResponse;

@Controller
public class WsController {
@MessageMapping("/welcome") //1
@SendTo("/topic/getResponse") //2
    public WiselyResponse say(WiselyMessage message) throws Exception {
            Thread.sleep(3000);
            return new WiselyResponse("Welcome, " + message.getName() + "!");
    }
}
```

代码解释

① 当浏览器向服务端发送请求时，通过@MessageMapping 映射/welcome 这个地址，类似于@RequestMapping。

② 当服务端有消息时，会对订阅了@SendTo 中的路径的浏览器发送消息。

（5）添加脚本。将 stomp.min.js（STOMP 协议的客户端脚本）、sockjs.min.js（SockJS 的客户端脚本）以及 jQuery 放置在 src/main/resources/static 下。读者可在这一章的源码里找到这几个脚本，或者自行下载。

（6）演示页面。在 src/main/resources/templates 下新建 ws.html，代码如下：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8" />
    <title>Spring Boot+WebSocket+广播式</title>

</head>
<body onload="disconnect()">
<noscript><h2 style="color: #ff0000">貌似你的浏览器不支持 websocket</h2></noscript>
<div>
    <div>
        <button id="connect" onclick="connect();">连接</button>
        <button id="disconnect" disabled="disabled" onclick="disconnect();">断开
连接</button>
    </div>
    <div id="conversationDiv">
        <label>输入你的名字</label><input type="text" id="name" />
        <button id="sendName" onclick="sendName();">发送</button>
```

```
      <p id="response"></p>
   </div>
</div>
<script th:src="@{sockjs.min.js}"></script>
<script th:src="@{stomp.min.js}"></script>
<script th:src="@{jquery.js}"></script>
<script type="text/javascript">
   var stompClient = null;

   function setConnected(connected) {
      document.getElementById('connect').disabled = connected;
      document.getElementById('disconnect').disabled = !connected;
      document.getElementById('conversationDiv').style.visibility = connected ?
'visible' : 'hidden';
      $('#response').html();
   }

   function connect() {
      var socket = new SockJS('/endpointWisely'); //1
      stompClient = Stomp.over(socket); //2
      stompClient.connect({}, function(frame) { //3
         setConnected(true);
         console.log('Connected: ' + frame);
         stompClient.subscribe('/topic/getResponse', function(respnose){ //4
            showResponse(JSON.parse(respnose.body).responseMessage);
         });
      });
   }
   function disconnect() {
      if (stompClient != null) {
         stompClient.disconnect();
      }
      setConnected(false);
      console.log("Disconnected");
   }

   function sendName() {
      var name = $('#name').val();
       //5
      stompClient.send("/welcome", {}, JSON.stringify({ 'name': name }));
   }

   function showResponse(message) {
        var response = $("#response");
```

```
        response.html(message);
    }
</script>
</body>
</html>
```

代码解释

① 连接 SockJS 的 endpoint 名称为“/endpointWisely”。

② 使用 STOMP 子协议的 WebSocket 客户端。

③ 连接 WebSocket 服务端。

④ 通过 stompClient.subscribe 订阅/topic/getResponse 目标（destination）发送的消息，这个是在控制器的@SendTo 中定义的。

⑤ 通过 stompClient.send 向/welcome 目标（destination）发送消息，这个是在控制器的@MessageMapping 中定义的。

（7）配置 viewController，为 ws.html 提供便捷的路径映射：

```
@Configuration
public class WebMvcConfig  extends WebMvcConfigurerAdapter{

     @Override
       public void addViewControllers(ViewControllerRegistry registry) {
          registry.addViewController("/ws").setViewName("/ws");
       }

}
```

（8）运行。我们预期的效果是：当一个浏览器发送一个消息到服务端时，其他浏览器也能接收到从服务端发送来的这个消息。

开启三个浏览器窗口，并访问 http://localhost:8080/ws，分别连接服务器。然后在一个浏览器中发送一条消息，其他浏览器接收消息。

连接服务端，如图 7-25 所示。

图 7-25　连接服务器

一个浏览器发送消息，如图 7-26 所示。

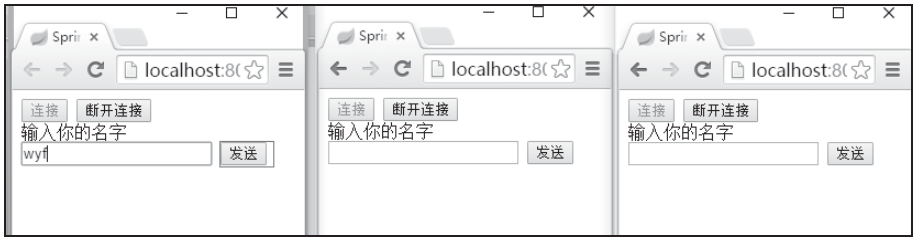


图 7-26　发送消息

所有浏览器接收服务端发送的消息，如图 7-27 所示。

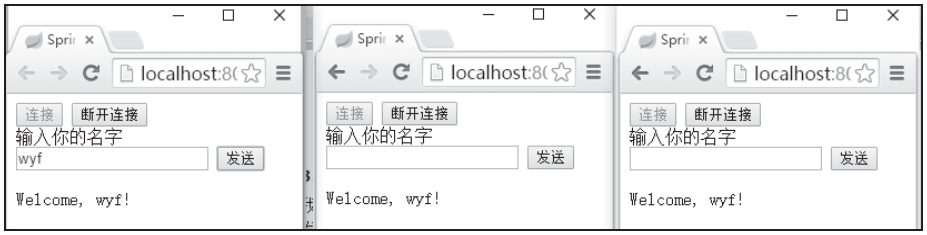


图 7-27　所有浏览器接收信息

我们在 Chrome 浏览器（在 Chrome 下按 F12 调出）下观察一下 STOMP 的帧，如图 7-28 所示。

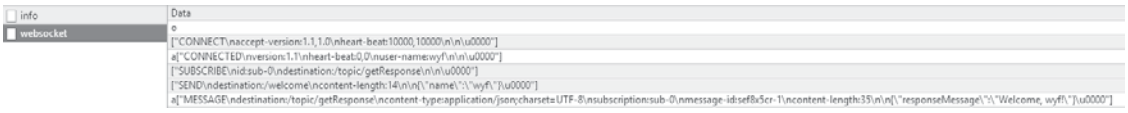


图 7-28　观察 STOMP 的帧

从上述截图可以观察得出，连接服务端的格式为：

```
CONNECT
accept-version:1.1,1.0
```

```
heart-beat:10000,10000
```

连接成功的返回为：

```
CONNECTED
version:1.1
heart-beat:0,0
```

订阅目标（destination）/topic/getResponse：

```
SUBSCRIBE
id:sub-0
destination:/topic/getResponse
```

向目标（destination）/welcome 发送消息的格式为：

```
SEND
destination:/welcome
content-length:14
{\"name\":\"wyf\"}
```

从目标（destination）/topic/getResponse 接收的格式为：

```
MESSAGE
destination:/topic/getResponse
content-type:application/json;charset=UTF-8
subscription:sub-0
message-id:zxj4wyau-0
content-length:35
{\"responseMessage\":\"Welcome, wyf!\"}
```

3. 点对点式

广播式有自己的应用场景，但是广播式不能解决我们一个常见的场景，即消息由谁发送、由谁接收的问题。

本例中演示了一个简单的聊天室程序。例子中只有两个用户，互相发送消息给彼此，因需要用户相关的内容，所以先在这里引入最简单的 Spring Security 相关内容。

（1）添加 Spring Security 的 starter pom：

```
<dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
```

（2）Spring Security 的简单配置。这里不对 Spring Security 做过多解释，只解释对本项目有帮助的部分：

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter{
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                .authorizeRequests()
                .antMatchers("/","/login").permitAll()//1
                .anyRequest().authenticated()
                .and()
                .formLogin()
                .loginPage("/login") //2
                .defaultSuccessUrl("/chat") //3
                .permitAll()
                .and()
                .logout()
                .permitAll();
    }

    //4
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
         auth
                .inMemoryAuthentication()
                .withUser("wyf").password("wyf").roles("USER")
                .and()
                .withUser("wisely").password("wisely").roles("USER");
    }
    //5
    @Override
    public void configure(WebSecurity web) throws Exception {
        web.ignoring().antMatchers("/resources/static/**");
    }

}
```

代码解释

① 设置 Spring Security 对/和/“login”路径不拦截。

② 设置 Spring Security 的登录页面访问的路径为/login。

③ 登录成功后转向/chat 路径。

④ 在内存中分别配置两个用户 wyf 和 wisely，密码和用户名一致，角色是 USER。

⑤ /resources/static/目录下的静态资源，Spring Security 不拦截。

（3）配置 WebSocket：

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig extends AbstractWebSocketMessageBrokerConfigurer{

    @Override
   public void registerStompEndpoints(StompEndpointRegistry registry) {
      registry.addEndpoint("/endpointWisely").withSockJS();
      registry.addEndpoint("/endpointChat").withSockJS();//1
   }


   @Override
   public void configureMessageBroker(MessageBrokerRegistry registry) {
      registry.enableSimpleBroker("/queue","/topic"); //2
   }

}
```

代码解释

① 注册一个名为/endpointChat 的 endpoint。

② 点对点式应增加一个/queue 消息代理。

（4）控制器。在 WsController 内添加如下代码：

```
    @Autowired
    private SimpMessagingTemplate messagingTemplate;//1

    @MessageMapping("/chat")
    public void handleChat(Principal principal, String msg) { //2
        if (principal.getName().equals("wyf")) {//3
            messagingTemplate.convertAndSendToUser("wisely",
                    "/queue/notifications", principal.getName() + "-send:"
```

```
                        + msg); //4
        } else {
            messagingTemplate.convertAndSendToUser("wyf",
                    "/queue/notifications", principal.getName() + "-send:"
                        + msg);
        }
    }
```

代码解释

① 通过 SimpMessagingTemplate 向浏览器发送消息。

② 在 Spring MVC 中，可以直接在参数中获得 principal，pinciple 中包含当前用户的信息。

③ 这里是一段硬编码，如果发送人是 wyf，则发送给 wisely；如果发送人是 wisely，则发送给 wyf，读者可以根据项目实际需要改写此处代码。

④ 通过 messagingTemplate.convertAndSendToUser 向用户发送消息，第一个参数是接收消息的用户，第二个是浏览器订阅的地址，第三个是消息本身。

（5）登录页面。在 src/main/resources/templates 下新建 login.html，代码如下：

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
     xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity3">
<meta charset="UTF-8" />
<head>
    <title>登录页面</title>
</head>
<body>
<div th:if="${param.error}">
    无效的账号和密码
</div>
<div th:if="${param.logout}">
    你已注销
</div>
<form th:action="@{/login}" method="post">
    <div><label> 账号 : <input type="text" name="username"/> </label></div>
    <div><label> 密码: <input type="password" name="password"/> </label></div>
    <div><input type="submit" value="登陆"/></div>
</form>
</body>
</html>
```

（6）聊天页面。在 src/main/resources/templates 下新建 chat.html，代码如下：

```
<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">
<meta charset="UTF-8" />
<head>
    <title>Home</title>
    <script th:src="@{sockjs.min.js}"></script>
    <script th:src="@{stomp.min.js}"></script>
    <script th:src="@{jquery.js}"></script>
</head>
<body>
<p>
    聊天室
</p>

<form id="wiselyForm">
    <textarea rows="4" cols="60" name="text"></textarea>
    <input type="submit"/>
</form>

<script th:inline="javascript">
    $('#wiselyForm').submit(function(e){
        e.preventDefault();
        var text = $('#wiselyForm').find('textarea[name="text"]').val();
        sendSpittle(text);
    });

    var sock = new SockJS("/endpointChat"); //1
    var stomp = Stomp.over(sock);
    stomp.connect('guest', 'guest', function(frame) {
        stomp.subscribe("/user/queue/notifications", handleNotification);//2
    });


    function handleNotification(message) {
        $('#output').append("<b>Received: " + message.body + "</b><br/>")
    }

    function sendSpittle(text) {
        stomp.send("/chat", {}, text);//3
    }
    $('#stop').click(function() {sock.close()});
```

```
</script>

<div id="output"></div>
</body>
</html>
```

代码解释

① 连接 endpoint 名称为“/endpointChat”的 endpoint。

② 订阅/user/queue/notifications 发送的消息，这里与在控制器的 messagingTemplate.convertAndSendToUser 中定义的订阅地址保持一致。这里多了一个/user，并且这个/user 是必须的，使用了/user 才会发送消息到指定的用户。

（7）增加页面的 viewController：

```
@Configuration
publicclass WebMvcConfig  extends WebMvcConfigurerAdapter{

    @Override
    publicvoid addViewControllers(ViewControllerRegistry registry) {
    registry.addViewController("/ws").setViewName("/ws");
    registry.addViewController("/login").setViewName("/login");
    registry.addViewController("/chat").setViewName("/chat");
      }

}
```

（8）运行。我们预期的效果是：两个用户登录系统，可以互发消息。但是一个浏览器的用户会话 session 是共享的，我们可以在谷歌浏览器设置两个独立的用户，从而实现用户会话 session 隔离，如图 7-29 所示。

现在分别在两个用户下的浏览器访问：http://localhost:8080/login，并登录，如图 7-30 所示。

wyf 用户向 wisely 用户发送消息，如图 7-31 所示。

wisely 用户向 wyf 用户发送消息如图 7-32 所示。

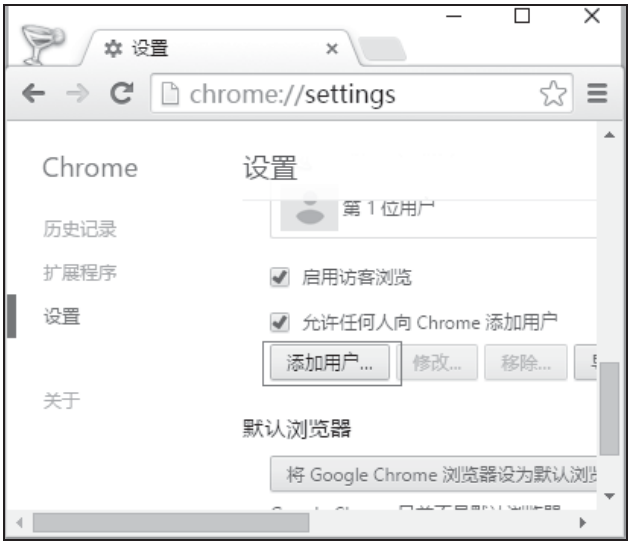


图 7-29　两个独立的用户

图 7-30　分别登录

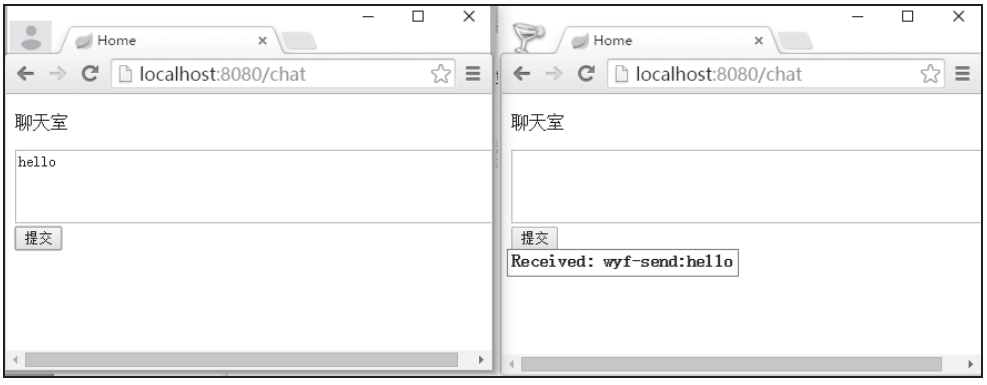


图 7-31　wyf 用户向 wisely 用户发送消息

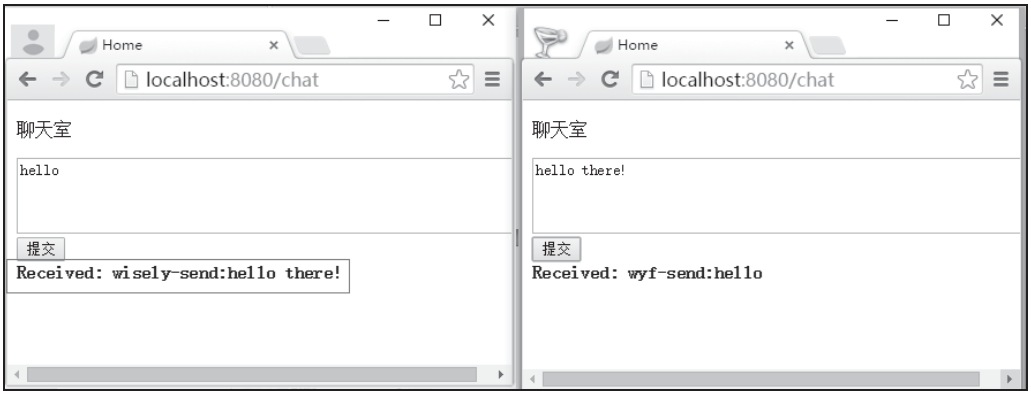


图 7-32　wisely 用户向 wyf 用户发送消息

7.7　基于 Bootstrap 和 AngularJS 的现代 Web 应用

现代的 B/S 系统软件有下面几个特色。

1. 单页面应用

单页面应用（single-page application，简称 SPA）指的是一种类似于原生客户端软件的更流畅的用户体验的页面。在单页面应用中，所有的资源（HTML、JavaScript、CSS）都是按需动态加载到页面上的，且不需要服务端控制页面的转向。

2. 响应式设计

响应式设计（Responsive web design，简称 RWD）指的是不同的设备（电脑、平板、手机）访问相同的页面的时候，得到不同的页面视图，而得到的视图是适应当前屏幕的。当然就算在电脑上，我们通过拖动浏览器窗口的大小，也能得到合适的视图。

3. 数据导向

数据导向是对于页面导向而言的，页面上的数据获得是通过消费后台的 REST 服务来实现的，而不是通过服务器渲染的动态页面（如 JSP）来实现的，一般数据交换使用的格式是 JSON。

本节将针对 Bootstrap 和 AngularJS 进行快速入门式的引导，如需深入学习，请参考官网或相关专题书籍。

7.7.1　Bootstrap

1. 什么是 Bootstrap

Bootstrap 官方定义：Bootstrap 是开发响应式和移动优先的 Web 应用的最流行的 HTML、CSS、JavaScript 框架。

Boostrap 实现了只使用一套代码就可以在不同的设备显示你想要的视图的功能。Bootstrap 还为我们提供了大量美观的 HTML 元素前端组件和 jQuery 插件。

2. 下载并引入 Bootstrap

下载地址：http://getbootstrap.com/getting-started/，如图 7-33 所示。

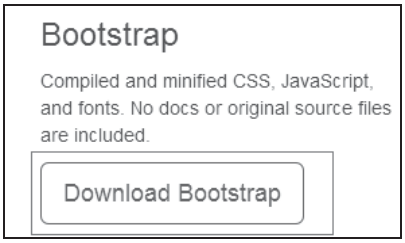


图 7-33　下载页面

下载的压缩包的目录结构如图 7-34 所示。

图 7-34　目录结构

最简单的 Bootstrap 页面模板如下：

```
<!DOCTYPE html>
<html lang="zh-cn">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <!-- 上面 3 个 meta 标签必须是 head 的头三个标签，其余的 head 内标签在此 3 个之后 The above
3 meta tags *must* come first in the head; any other head content must come *after*
these tags -->
    <title>Bootstrap 基本模板</title>

    <!-- Bootstrap 的 CSS -->
    <link href="bootstrap/css/bootstrap.min.css" rel="stylesheet">

    <!-- HTML5 shim and Respond.js 用来让 IE 8 支持 HTML 5 元素和媒体查询 -->

    <!--[if lt IE 9]>
      <script src="js/html5shiv.min.js"></script>
      <script src="js/respond.min.js"></script>
    <![endif]-->
  </head>
  <body>
    <h1>你好, Bootstrap!</h1>

    <!-- jQuery 是 Bootstrap 脚本插件必需的 -->
    <script src="js/jquery.min.js"></script>
    <!-- 包含所有编译的插件 -->
    <script src="bootstrap/js/bootstrap.min.js"></script>
  </body>
</html>
```

3. CSS 支持

Bootstrap 的 CSS 样式为基础的 HTML 元素提供了美观的样式，此外还提供了一个高级的网格系统用来做页面布局。

（1）布局网格

在 Bootstrap 里，行使用的样式为 row，列使用 col-md-数字，此数字范围为 1～12，所有列加起来的和也是 12，代码如下：

```
    <div class="row">
        <div class="col-md-1">.col-md-1</div>
        <div class="col-md-1">.col-md-1</div>
        <div class="col-md-1">.col-md-1</div>
        <div class="col-md-1">.col-md-1</div>
        <div class="col-md-1">.col-md-1</div>
        <div class="col-md-1">.col-md-1</div>
```

```
        <div class="col-md-1">.col-md-1</div>
        <div class="col-md-1">.col-md-1</div>
        <div class="col-md-1">.col-md-1</div>
        <div class="col-md-1">.col-md-1</div>
        <div class="col-md-1">.col-md-1</div>
        <div class="col-md-1">.col-md-1</div>
    </div>
    <div class="row">
        <div class="col-md-8">.col-md-8</div>
        <div class="col-md-4">.col-md-4</div>
    </div>
    <div class="row">
        <div class="col-md-4">.col-md-4</div>
        <div class="col-md-4">.col-md-4</div>
        <div class="col-md-4">.col-md-4</div>
    </div>
    <div class="row">
        <div class="col-md-6">.col-md-6</div>
        <div class="col-md-6">.col-md-6</div>
    </div>
```

布局效果如图 7-35 所示。

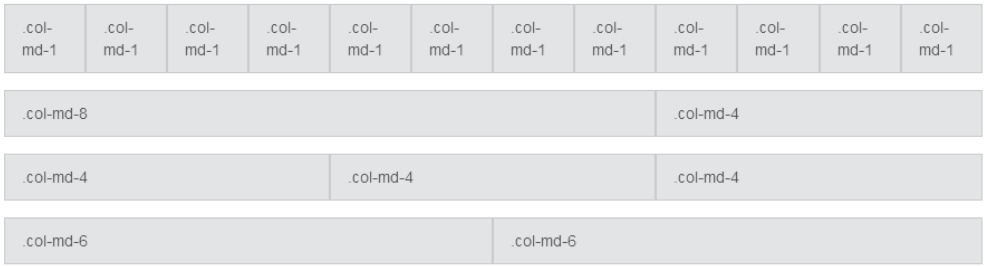


图 7-35　布局效果

（2）html 元素

Bootstrap 为 html 元素提供了大量的样式，如表单元素、按钮、图标等。更多内容请查看：http://getbootstrap.com/css/。

4. 页面组件支持

Bootstrap 为我们提供了大量的页面组件，包括字体图标、下拉框、导航条、进度条、缩略图等，更多请阅读 http://getbootstrap.com/components/。

5. Javascript 支持

Bootstrap 为我们提供了大量的 JavaScript 插件，包含模式对话框、标签页、提示、警告等，更多内容请查看 http://getbootstrap.com/javascript/。

7.7.2 AngularJS

1. 什么是 AngularJS

AngularJS 官方定义：AngularJS 是 HTML 开发本应该的样子，它是用来设计开发 Web 应用的。

AngularJS 使用声名式模板+数据绑定（类似于 JSP、Thymeleaf）、MVW（Model-View-Whatever）、MVVM（Model-View-ViewModel）、MVC（Model-View-Controller）、依赖注入和测试，但是这一切的实现却只借助纯客户端的 JavaScript。

HTML 一般是用来声明静态页面的，但是通常情况下我们希望页面是基于数据动态生成的，这也是我们很多服务端模板引擎出现的原因；而 AngularJS 可以只通过前端技术就实现动态的页面。

2. 下载并引入 AngularJS

AngularJS 下载地址：https://angularjs.org/，如图 7-36 所示。

图 7-36 下载页面

最简单的 AngularJS 页面：

```
<!doctype html>
<html ng-app><!-- 1 -->
  <head>
    <script src="js/angular.min.js"></script><!-- 2 -->
  </head>
  <body>
    <div>
      <label>名字:</label>
      <input type="text" ng-model="yourName" placeholder="輸入你的名字"><!-- 3 -->
      <hr>
      <h1>你好 {{yourName}}!</h1><!-- 4 -->
    </div>
  </body>
</html>
```

代码解释

① ng-app 所作用的范围是 AngularJS 起效的范围，本例是整个页面有效。

② 载入 AngularJS 的脚本。

③ ng-model 定义整个 AngularJS 的前端数据模型，模型的名称为 yourName，模型的值来自你输入的值若输入的值改变，则数据模型值也会改变。

④ 使用{{模型名}}来读取模型中的值。

效果如图 7-37 所示。

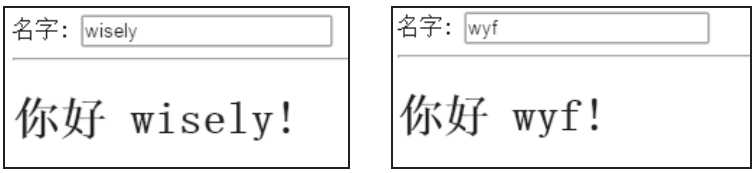


图 7-37　运行效果

3. 模块、控制器和数据绑定

我们对 MVC 的概念已经烂熟于心了，但是平时的 MVC 都是服务端的 MVC，这里用 AngularJS 实现了纯页面端的 MVC，即实现了视图模板、数据模型、代码控制的分离。

再来看看数据绑定，数据绑定是将视图和数据模型绑定在一起。如果视图变了，则模型的

值就变了；如果模型值变了，则视图也会跟着改变。

AngularJS 为了分离代码达到复用的效果，提供了一个 module（模块）。定义一个模块需使用下面的代码。

无依赖模块：

```
angular.module('firstModule',[]);
```

有依赖模块：

```
angular.module('firstModule',['moduleA','moduleB']);
```

我们看到了 V 就是我们的页面元素，M 就是我们的 ng-model，那 C 呢？我们可以通过下面的代码来定义控制器，页面使用 ng-controller 来和其关联：

```
angular.module('firstModule',[])
    .controller('firstController',function(){
    ...
};
);


<div ng-controller="firstController">
...
</div>
```

4. Scope 和 Event

（1）Scope

Scope 是 AngularJS 的内置对象，用$Scope 来获得。在 Scope 中定义的数据是数据模型，可以通过{{模型名}}在视图上获得。Scope 主要是在编码中需要对数据模型进行处理的时候使用，Scope 的作用范围与在页面声明的范围一致（如在 controller 内使用，scope 的作用范围是页面声明 ng-controller 标签元素的作用范围）。

定义：

```
$scope.greeting='Hello'
```

获取：

```
{{greeting}}
```

（2）Event

因为 Scope 的作用范围不同，所以不同的 Scope 之间若要交互的话需要通过事件（Event）来完成。

1）冒泡事件（Emit）冒泡事件负责从子 Scope 向上发送事件，示例如下。

子 Scope 发送：

```
$scope.$emit('EVENT_NAME_EMIT ', 'message');
```

父 Scope 接受：

```
$scope.$on(''EVENT_NAME_EMIT', function(event, data) {
...
})
```

2）广播事件（Broadcast）。广播事件负责从父 Scope 向下发送事件，示例如下。

父 Scope 发送：

```
$scope.$broadcast('EVENT_NAME_BROAD ', 'message');
```

子 scope 接受

```
$scope.$on(''EVENT_NAME_BROAD', function(event, data) {
...
})
```

5. 多视图和路由

多视图和路由是 AngularJS 实现单页面应用的技术关键，AngularJS 内置了一个 $routeProvider 对象来负责页面加载和页面路由转向。

需要注意的是，1.2.0 之后的 AngularJS 将路由功能移出，所以使用路由功能要另外引入 angular-route.js

例如：

```
angular.module('firstModule').config(function($routeProvider) {
$routeProvider.when('/view1', { //1
    controller: 'Controller1', //2
    templateUrl: 'view1.html', //3
}).when('/view2', {
    controller: 'Controller2',
```

```
    templateUrl: 'view2.html',
});
})
```

代码解释

① 此处定义的是某个页面的路由名称。

② 此处定义的是当前页面使用的控制器。

③ 此处定义的要加载的真实页面。

在页面上可以用下面代码来使用我们定义的路由：

```
<ul>
    <li><a href="#/view1">view1</a></li>
    <li><a href="#/view2">view2</a></li>
</ul>
<ng-view></ng-view> <!-- 此处为加载进来的页面显示的位置-->
```

6. 依赖注入

依赖注入是 AngularJS 的一大酷炫功能。可以实现对代码的解耦，在代码里可以注入 AngularJS 的对象或者我们自定义的对象。下面示例是在控制器中注入$scope，注意使用依赖注入的代码格式。

```
  angular.module('firstModule')
    .controller("diController", ['$scope',
           function ($scope) {
              ...
       }]);
```

7. Service 和 Factory

AngularJS 为我们内置了一些服务，如$location、$timeout、$rootScope（请读者自行学习相关的知识）。很多时候我们需要自己定制一些服务，AngularJS 为我们提供了 Service 和 Factory。

Service 和 Factory 的区别是：使用 Service 的话，AngularJS 会使用 new 来初始化对象；而使用 Factory 会直接获得对象。

（1）Service

定义：

```
angular.module('firstModule').service('helloService',function(){
    this.sayHello=function(name){
        alert('Hello '+name);
    }
});
```

注入调用：

```
  angular.module('firstModule')
    .controller("diController", ['$scope', 'helloService',
           function ($scope,helloService) {
              helloService.sayHello('wyf');
       }]);
```

（2）Factory

定义：

```
angular.module('firstModule').service('helloFactory',function(){
    return{
    sayHello:function(name){
            alert('Hello '+name);
        }
    }
});
```

注入调用：

```
  angular.module('firstModule')
    .controller("diController", ['$scope', 'helloFactory',
           function ($scope, helloFactory) {
              helloFactory.sayHello('wyf');
       }]);
```

8. http 操作

AngularJS 内置了$http 对象用来进行 Ajax 的操作：

```
$http.get(url)
$http.post(url,data)
$http.put(url,data)
```

```
$http.delete(url)
$http.head(url)
```

9. 自定义指令

AngularJS 内置了大量的指令（directive），如 ng-repeat、ng-show、ng-model 等。即使用一个简短的指令可实现一个前端组件。

比方说，有一个日期的 js/jQuery 插件，使用 AngularJS 封装后，在页面上调用此插件可以通过指令来实现，例如：

```
元素指令: <date-picker></date-picker>
属性指令: <input type="text" date-picker/>
样式指令: <input type="text" class="date-picker"/>
注释指令 : <!--directive:date-picker-->
```

定义指令：

```
angular.module('myApp',[]).directive('helloWorld', function() {
return {
    restrict: 'AE',//支持使用属性、元素
    replace: true,
    template: '<h3>Hello, World!</h3>'
};
});
```

调用指令，元素标签：

```
<hello-world/>

<hello:world/>
```

或者属性方式：

```
<div hello-world />
```

7.7.3 实战

在前面两节，我们快速介绍了 Bootstrap 和 AngularJS，本节我们将它们和 Spring Boot 串起来做个例子。

在例子中，我们使用 Bootstrap 制作导航，使用 AngularJS 实现导航切换页面的路由功能，并演示 AngularJS 通过$http 服务和 Spring Boot 提供的 REST 服务，最后演示用指令封装 jQuery

UI 的日期选择器。

1. 新建 Spring Boot 项目

初始化一个 Spring Boot 项目，依赖只需选择 Web（spring-boot-starter-web）。

项目信息：

```
groupId: com.wisely
arctifactId:ch7_7
package: com.wisely.ch7_7
```

准备 Bootstrap、AngularJS、jQuery、jQueryUI 相关的资源到 src/main/resources/static 下，结构如图 7-38 所示。

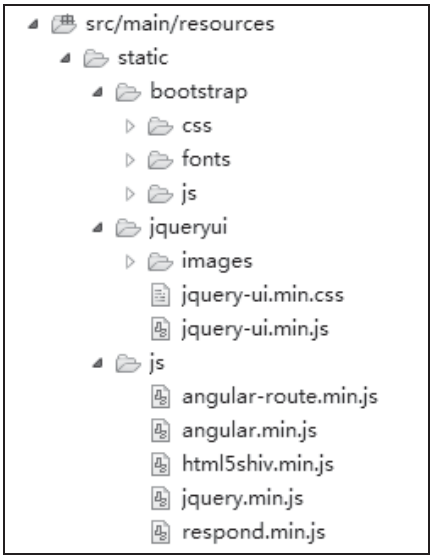


图 7-38　结构

另外要说明的是，本例的页面都是静态页面，所以全部放置在/static 目录下。

2. 制作导航

页面位置：src/main/resources/static/action.html：

```
<!DOCTYPE html>
<html lang="zh-cn" ng-app="actionApp">
<head>
```

```
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>实战</title>

<link href="bootstrap/css/bootstrap.min.css" rel="stylesheet">
<link href="jqueryui/jquery-ui.min.css" rel="stylesheet">
<style type="text/css">

.content {
  padding: 100px 15px;
  text-align: center;
}
</style>

<!--[if lt IE 9]>
     <script src="js/html5shiv.min.js"></script>
     <script src="js/respond.min.js"></script>
   <![endif]-->
</head>
<body>
<!-- 1 -->
    <nav class="navbar navbar-inverse navbar-fixed-top">
     <div class="container">

       <div id="navbar" class="collapse navbar-collapse">
         <ul class="nav navbar-nav">
           <li><a href="#/oper">后台交互</a></li>
           <li><a href="#/directive">自定义指令</a></li>
         </ul>
       </div>
     </div>
   </nav>

    <!-- 2 -->
    <div class="content">
       <ng-view></ng-view>
</div>
<!-- 3 -->
    <script src="js/jquery.min.js"></script>
    <script src="jqueryui/jquery-ui.min.js"></script>
    <script src="bootstrap/js/bootstrap.min.js"></script>
    <script src="js/angular.min.js"></script>
    <script src="js/angular-route.min.js"></script>
```

```
    <script src="js-action/app.js"></script>
    <script src="js-action/directives.js"></script>
    <script src="js-action/controllers.js"></script>
</body>
</html>
```

代码解释

① 使用 Bootstrap 定义的导航，并配合 AngularJS 的路由，通过路由名称#/oper 和#/directive 切换视图；

② 通过<ng-view></ng-view>展示载入的页面。

③ 加载本例所需的脚本，其中 jquery-ui.min.js 的脚本是为我们定制指令所用；app.js 定义 AngularJS 的模块和路由；directives.js 为自定义的指令；controllers.js 是控制器定义之处。

3. 模块和路由定义

页面位置：src/main/resources/static/js-action/app.js：

```
var actionApp = angular.module('actionApp',['ngRoute']); //1

actionApp.config(['$routeProvider' , function($routeProvider) {//2
    $routeProvider.when('/oper', { //3
        controller: 'View1Controller', //4
        templateUrl: 'views/view1.html', //5
    }).when('/directive', {
        controller: 'View2Controller',
        templateUrl: 'views/view2.html',
    });
}]);
```

代码解释

① 定义模块 actionApp，并依赖于路由模块 ngRout。

② 配置路由，并注入$routeProvider 来配置。

③ /oper 为路由名称。

④ controller 定义的是路由的控制器名称。

⑤ templateUrl 定义的是视图的真正地址。

4. 控制器定义

脚本位置：src/main/resources/static/js-action/ controllers.js：

```
//1
actionApp.controller('View1Controller', ['$rootScope', '$scope', '$http',
function($rootScope, $scope,$http) {
    //2
    $scope.$on('$viewContentLoaded', function() {
        console.log('页面加载完成');
    });
    //3
    $scope.search = function(){//3.1
      personName = $scope.personName; //3.2
      $http.get('search',{ //3.3
                    params:{personName:personName}//3.4
        }).success(function(data){ //3.5
            $scope.person=data; //3.6
        });

    };
}]);

actionApp.controller('View2Controller', ['$rootScope', '$scope',
function($rootScope, $scope) {
    $scope.$on('$viewContentLoaded', function() {
        console.log('页面加载完成');
    });
}]);
```

代码解释

① 定义控制器 View1Controller，并注入$rootScope、$scope 和$http。

② 使用$scope.$on 监听$viewContentLoaded 事件，可以在页面内容加载完成后进行一些操作。

③ 这段代码是这个演示的核心代码，请结合下面的 Viewl 的界面一起理解：

- 在 scope 内定义一个方法 search，在页面上通过 ng-click 调用。
- 通过$scope.personName 获取页面定义的 ng-model=“personName”的值。
- 使用$http.get 向服务端地址 search 发送 get 请求。
- 使用 params 增加请求参数。

- 用 success 方法作为请求成功后的回调。
- 将服务端返回的数据 data 通过$scope.person 赋给模型 person，这样页面视图上可以通过{{person.name}}、{{person.age}}、{{person.address}}来调用，且模型 person 值改变后，视图是自动更新的。

5. View1 的界面（演示与服务端交互）

页面位置：src/main/resources/static/views/view1.html。

```
 <div class="row">
   <label for="attr" class="col-md-2 control-label">名称</label>
<div class="col-md-2">
     <!-- 1 -->
    <input type="text" class="form-control" ng-model="personName">
   </div>
<div class="col-md-1">
<!-- 2 -->
    <button class="btn btn-primary" ng-click="search()">查询</button>
   </div>
 </div>

  <div class="row">
    <div class="col-md-4">
         <ul class="list-group">
           <!-- 3 -->
           <li class="list-group-item">名字:  {{person.name}}</li>
           <li class="list-group-item">年龄:   {{person.age}}</li>
           <li class="list-group-item">地址:   {{person.address}}</li>
         </ul>
    </div>
 </div>
```

代码解释

① 定义数据模型 ng-model=“personName”。

② 通过 ng-click=“search()”调用控制器中定义的方法。

③ 通过{{person.name}}、{{person.age}}、{{person.address}}访问控制器的 scope 里定义的 person 模型，模型和视图是绑定的。

6. 服务端代码

传值对象 Javabean：

```
package com.wisely.ch7_7;

public class Person {
    private String name;
    private Integer age;
    private String address;


    public Person() {
        super();
    }
    public Person(String name, Integer age, String address) {
        super();
        this.name = name;
        this.age = age;
        this.address = address;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Integer getAge() {
        return age;
    }
    public void setAge(Integer age) {
        this.age = age;
    }
    public String getAddress() {
        return address;
    }
    public void setAddress(String address) {
        this.address = address;
    }



}
```

控制器：

```
package com.wisely.ch7_7;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@SpringBootApplication
public class Ch77Application {

    @RequestMapping(value="/search",produces={MediaType.APPLICATION_JSON_VALU
E})
    public Person search(String personName){

        return new Person(personName, 32, "hefei");

    }

   public static void main(String[] args) {
      SpringApplication.run(Ch77Application.class, args);
   }
}
```

代码解释

这里我们只是模拟一个查询，即接受前台传入的 personName，然后返回 Person 类，因为我们使用的是@RestController，且返回值类型是 Person，所以 Spring MVC 会自动将对象输出为 JSON。

7. 自定义指令

脚本位置：src/main/resources/static/js-action/directives.js：

```
actionApp.directive('datePicker',function(){//1
   return {
      restrict: 'AC', //2
      link:function(scope,elem,attrs) { //3
         elem.datepicker();//4
      }
```

```
    };
});
```

代码解释

① 定义一个指令名为 datePicker。

② 限制为属性指令和样式指令。

③ 使用 link 方法来定义指令，在 link 方法内可使用当前 scope、当前元素及元素属性。

④ 初始化 jqueryui 的 datePicker（jquery 的写法是$('#id').datePicker()）。

通过上面的代码我们就定制了一个封装 jqueryui 的 datePicker 的指令，本例只是为了演示的目的，主流的脚本框架已经被很多人封装过了，有兴趣的读者可以访问 http://ngmodules.org/ 网站，这个网站包含了大量 AngularJS 的第三方模块、插件和指令。

8. View2 的页面（演示自定义指令）

页面地址：src/main/resources/static/views/view2.html：

```
  <div class="row">
    <label for="attr" class="col-md-2 control-label">属性形式</label>
<div class="col-md-2">
      <!-- 1 -->
     <input type="text" class="form-control" date-picker>
    </div>
  </div>

  <div class="row">
    <label for="style" class="col-md-2 control-label">样式形式</label>
<div class="col-md-2">
      <!-- 2 -->
     <input type="text" class="form-control date-picker" >
    </div>
  </div>
```

代码解释

① 使用属性形式调用指令。

② 使用样式形式调用指令。

9. 运行

菜单及路由切换如图 7-39 所示。

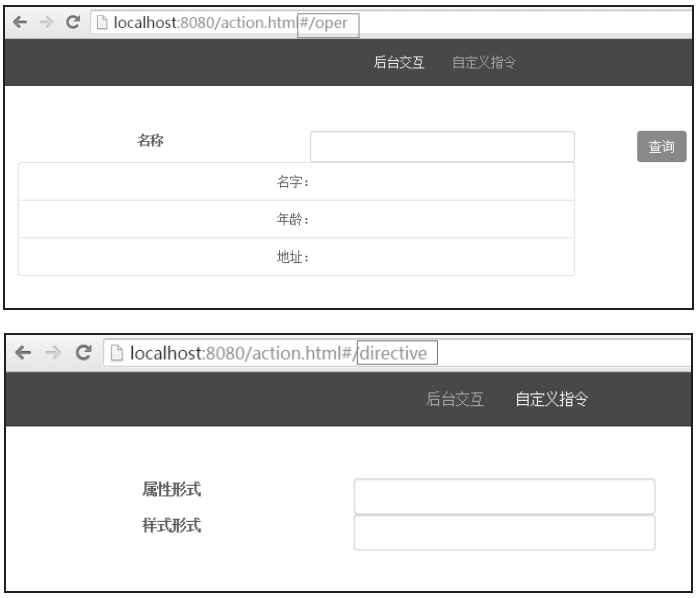


图 7-39　菜单及路由交换

与后台交互如图 7-40 所示。

图 7-40　与后台交互

自定义指令如图 7-41 所示。

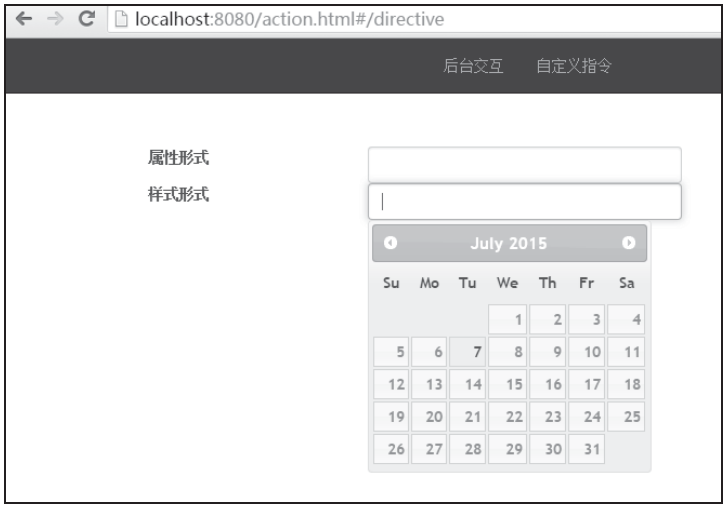

localhost:8080/action.html#/directive
后台交互
自定义指令
属性形式
样式形式
July 2015
Su Mo Tu We Th Fr Sa
1 2 3 4
5 6 7 8 9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31

图 7-41　自定义指令