SpringMVC+Thymeleaf
开发指南

文档编号	
编写人	hankChan
修改日期	2016-12-30
版本	V1.0

Thymeleaf 简述

Thymeleaf 是一款跟 Velocity、FreeMarker 类似的模板引擎，它可以完全替代 JSP。

有如下特点：

1. Thymeleaf 在有网络和无网络的环境下皆可运行，即它可以让美工在浏览器查看页面的静态效果，也可以让程序员在服务器查看带数据的动态页面效果。这是由于它支持 HTML 原型，然后在 HTML 标签里增加额外的属性来达到模板+数据的展示方式。浏览器解释 HTML 时会忽略未定义的标签属性，所以 Thymeleaf 的模板可以静态地运行；当有数据返回到页面时，Thymeleaf 标签会动态地替换掉静态内容，使页面动态显示。

2. Thymeleaf 开箱即用的特性。它提供标准和 Spring 标准两种方言，可以直接套用模板实现 JSTL、OGNL 表达式效果，避免每天套模板、改 JSTL、改标签的困扰。同时开发人员也可以扩展和创建自定义的方言。

3. Thymeleaf 提供 Spring 标准方言和一个与 SpringMVC 完美集成的可选模块，可以快速的实现表单绑定、属性编辑器、国际化等功能。

Demo-实现

Demo-基础环境搭建

开发环境基础

（由于一些 Thymeleaf 的版本依赖问题，建议使用以下版本信息）

JavaJDK 版本：1.8

Thymeleaf 版本：3.0.0.RELEASE

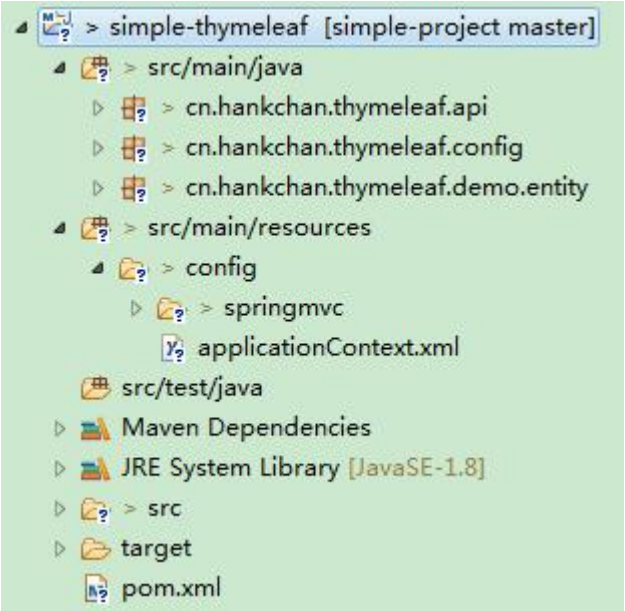
Spring-Framework 版本：4.2.5.RELEASE

Maven 版本：3.3.9

创建基于 Maven 的 Web 应用项目

（后补）

项目结构图：



添加 Thymeleaf 依赖

（确保 JavaJDK 版本设置正确）

在 pom.xml 文件中添加如下依赖：

```
        <spring.version>4.2.5.RELEASE</spring.version>
        <thymeleaf.version>3.0.0.RELEASE</thymeleaf.version>
    </properties>
    <dependencies>
        <!-- thymeleaf依赖 -->
        <dependency>
            <groupId>org.thymeleaf</groupId>
            <artifactId>thymeleaf</artifactId>
            <version>${thymeleaf.version}</version>
        </dependency>
        <dependency>
            <groupId>org.thymeleaf</groupId>
            <artifactId>thymeleaf-spring4</artifactId>
            <version>${thymeleaf.version}</version>
        </dependency>
        <dependency>
            <groupId>org.thymeleaf.extras</groupId>
            <artifactId>thymeleaf-extras-springsecurity4</artifactId>
            <version>${thymeleaf.version}</version>
        </dependency>
        <dependency>
            <groupId>org.thymeleaf.extras</groupId>
            <artifactId>thymeleaf-extras-java8time</artifactId>
            <version>3.0.0-SNAPSHOT</version>
        </dependency>
```

鼠标选定项目，更新依赖：右键=>Maven=>Update Project

配置 Spring+SpringMVC

首先搭建最基础的 Spring+SpringMVC 的开发环境，包括基础的 web 应用文件、Spring 配置文件、SpringMVC 配置文件的配置。

配置 web.xml 文件

在应用根目录下的 src/main/webapp/WEB-INF/web.xml 文件中，配置 Spring 监听器和 SpringMVC 前端控制器信息。具体如下：

```
<display-name>Archetype Created Web Application</display-name>
<!-- Spring 监听器 -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:config/applicationContext.xml</param-value>
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

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

配置 Spring 配置文件

根据我们在 web.xml 的定义可以知道，Spring 的配置文件放在 classpath 下（此处就是在在 Maven 项目中的 src/main/resources 路径）的 config 目录下，在此命名为：applicationContext.xml

加入启动注解配置的节点，如下图：

```
<context:annotation-config />
```

配置 SpringMVC 配置文件

加入启动相关注解驱动的节点及默认 ServletHandler，如下图：

```
<mvc:annotation-driven />
<mvc:default-servlet-handler />
```

到此，基础的 Spring+SpringMVC 基础环境搭建完成！接下来集成整合 Thymeleaf 到 Spring 中。

配置管理 Thymeleaf 必须的 Bean

为了在 Spring 中使用 Thymeleaf，需要配置三个启动 Thymeleaf 与 Spring 集成的 Bean：

ThymeleafViewResolver：视图解析器，将逻辑视图名称解析为 Thymeleaf 模板视图

SpringTemplateEngine：处理模板并渲染结果

TemplateResolver：模板解析器，加载 Thymeleaf 模板

推荐使用 XML 配置方式或者 JavaConfig 方式配置管理，在此暂时采用 XML 配置方式管理 Thymeleaf 相关的 Bean。（JavaConfig 方式在 ThymeleafConfig 类中也实现了，但是没有启用）

配置 Thymeleaf 的模板解析器

在 springmvc.xml 中添加管理 Thymeleaf 的模板解析器的 Bean，具体如下图所示：

```
<!-- 配置模板解析器 -->
<!--
    TemplateEngine从/WEB-INF/templates/目录中读取文件夹，默认的后缀名是.html，
    所以在渲染模板时只需要提供模板的名字（例如index.html可以省略为index）
 -->
<bean id="templateResolver"
        class="org.thymeleaf.spring4.templateresolver.SpringResourceTemplateResolver">
    <property name="prefix" value="/WEB-INF/templates/"/>
    <property name="suffix" value=".html"/>
    <property name="templateMode" value="HTML"/>
</bean>
```

其中，prefix 属性为生成模板文件完整路径前缀，suffix 属性为生成模板文件的后缀。templateMode 为模板模型，值为 HTML。老的版本属性值可以设定为 HTML5，但是新版本已经将 HTML5 标记为弃用（@Deprecated），不建议使用。

值得注意的是，老版本有使用 `ServletContextTemplateResolver` 作为该 Bean 的 class 的情况，但是由于新版本不太提供该抽象类的构造器，所以使用其子类 `SpringResourceTemplateResolver` 来作为 class 的值。

配置 Thymeleaf 的模板引擎

Thymeleaf 模板引擎用于渲染页面，如下图所示：

```
<!-- 当在applicationContext.xml中指定的ThymeleafConfig配置类中初始化，这里就都需要屏蔽 -->
<!-- 配置TemplateEngine -->
<bean id="templateEngine" class="org.thymeleaf.spring4.SpringTemplateEngine">
    <property name="templateResolver" ref="templateResolver" />
</bean>
```

配置 Spring 的视图解析器

类似于 JSP 页面，在 Spring 上需要配置解析 Thymeleaf 的视图解析器，如下图：

```
<!-- 配置Spring中的ViewResolver -->
<bean class="org.thymeleaf.spring4.view.ThymeleafViewResolver">
    <property name="templateEngine" ref="templateEngine" />
</bean>
```

Demo-创建测试用例

编写前端控制器类

在 src/main/java 下新建一个包，创建一个 IndexController.java 类作为前端控制器，在 SpringMVC 配置文件中设置指定扫描 Controller 的包路径，该路径为 Controller 所在的包名，如下图所示：

```
<context:component-scan base-package="cn.hankchan.thymeleaf.api" />
```

在该*.api 包下创建一个测试类：IndexController.java

该类使用@Controller 注解修饰作为一个前端控制器类。

编写一个处理请求的方法：根据请求 URL，做业务数据的转发，存入 Model 对象，返回到模板页面，由 Thymeleaf 模板引擎渲染 如下所示：

```
@RequestMapping("/second")
public String second(Model model) {  // model用于存放数据模型
    Map<String, User> map = new HashMap<>();
    // 处理数据
    User user1 = new User();
    user1.setLogin("chenjm");
    user1.setFirstName("Chen");
    user1.setLastName("Jiaming");
    user1.setGravatar("nothing");
    User user2 = new User("zhangt", "Zhang", "ting", "nothing");
    // 封装数据
    map.put("chenjm", user1);
    map.put("zhangt", user2);
    // 放入Model。在页面中可以通过${...}获取相应的数据
    //如: ${users.chenjm.firstName}的值为: Chen
    model.addAttribute("users", map);
    // 返回一个模板引擎渲染的页面结果,
    //这里为: /WEB-INF/templates/second.html
    return "second";
}
```

定义模板页面

Thymeleaf 很大程度上说就是 HTML 页面，但是需要在模板页面中，引入 Thymeleaf 标签：

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
```

在/WEB-INF/templates/目录下创建该模板页面 second.html，如下图：

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<body>
    <h1>Success Page</h1>
    <th th:text="${users}">UsersMap</th><br>

    firstName:<th th:text="${users.chenjm.firstName}">User1-Chenjm</th><br>
    User2:<th th:text="${users.zhangt}">User2-Zhangt</th><br>

</body>
</html>
```

然后就可以根据 Thymeleaf 提供的表达式语言配置生成的模板引擎。

SpringMVC-Thymeleaf 常用功能

SpringMVC 常用组件

@RequestMapping(“/url”)

用于修饰前端控制器类或者前端控制器类的响应请求方法。主要是用来匹配请求 URL，客户端发起请求，SpringMVC 通过/url 匹配到该注解修饰的方法中，通过该方法中定义的代码，响应客户端请求。

@ResponseBody

用于修饰前端控制器方法的返回值类型，如果方法声明了注解@ResponseBody ，则会直接将返回值输出到页面。

@ModelAttribute

用于修饰前端控制器类中的方法，被该注解修饰的方法会在该前端控制器类中的响应请求方法被调用前首先调用，可以用来处理一些与该前端控制器响应匹配的公共数据的操作。

当返回值类型为 void 时，可以通过 Model、ModelAndView 等模型添加数据，如下图：

```
/**
 * ModelAttribute标签修饰的方法会在该Controller中的
 * 所有RequestMapping修饰的方法执行前执行。
 * 可以用于在Model域中存放公共的数据
 */
@ModelAttribute
public void beforeRequstAction(Model model) {
    model.addAttribute("myContextUrl", "http://localhost:8083");
}
```

当@ModelAttribute 修饰方法返回值类型不为 void 时，通过设置@ModelAttribute 的 value 属性值，可以在模板页面中通过${key}方式获取到该返回类型的数据，如下图所示：

```
/**
 * 返回值类型不为void，通过指定ModelAttribute的value属性值
 * 可以在模板页面通过${users}方式获取到该返回类型的数据
 */
@ModelAttribute("users")
public List<User> beforeFourth() {
    List<User> users = new ArrayList<>();
    users.add(new User("1", "first-1", "last-1", "grava-1"));
    return users;
}
```

在模板页面中，可以通过如下方式获取数据（使用了 th:each 标签，后面会提到）：

```
<h2 th:each="user : ${users}">
    FirstName:<p th:text="${user.firstName}"></p>
    LastName:<p th:text="${user.lastName}"></p>
</h2>
```

Model

Model 是一个接口， 其实现类为 ExtendedModelMap，继承了 ModelMap 类。ModelMap 对象主要用于传递控制方法处理数据到结果页面，也就是说我们把结果页面上需要的数据放到 ModelMap 对象中即可，他的作用类似于 request 对象的 setAttribute 方法的作用，用来在一个请求过程中传递处理的数据。通过以下方法向页面传递参数：addAttribute(String key, Object value);

在页面上可以通过 EL 变量方式${key}等一系列数据展示标签获取并展示 ModelMap 中的数据。ModelMap 本身不能设置页面跳转的 URL 地址别名或者物理跳转地址，我们可以通过控制器方法的返回值来设置跳转 URL 地址别名或者物理跳转地址。

ModelAndView

ModelAndView 对象有两个作用：

作用一：设置转向地址，如下图所示（这也是 ModelAndView 和 ModelMap 的主要区别）

```
@RequestMapping("/user")
public ModelAndView userPage() {
    ModelAndView modelAndView = new ModelAndView();
    // 指定返回的模板视图名
    modelAndView.setViewName("user");
    return modelAndView;
}
```

作用二：用于传递控制方法处理结果数据到结果页面，也就是说我们把需要在结果页面上需要的数据放到 ModelAndView 对象中即可，他的作用类似于 request 对象的 setAttribute 方法的作用，用来在一个请求过程中传递处理的数据。通过以下方法向页面传递参数：

```java
@RequestMapping("/user")
public ModelAndView userPage() {
    ModelAndView modelAndView = new ModelAndView();

    // 将数据传递到模板页面，只需要放入该对象即可
    modelAndView.addObject("key", Arrays.asList("hello", "hey", "hi"));

    // 指定返回的模板视图名
    modelAndView.setViewName("user");
    return modelAndView;
}
```

在页面上可以通过 EL 变量方式${key}等一系列数据展示标签获取并展示 ModelAndView 中的数据。

Th 标签

th 标签灵活度很高，所有的 th 标签都能够嵌套使用。

简单表达式

变量表达式

使用${...}属性占位符，可以引用对象的某个属性，如下是引用了 user 对象的 name 属性：

```
<!-- 变量表达式 -->
firstName: <th th:text="${users.chenjm.firstName}">User1-Chenjm</th><br><br>
User2: <th th:text="${users.zhangt}">User2-Zhangt</th><br><br>
```

如上所示，当占位符的值存在时，将会自动填充覆盖文本节点的内容。

选择/星号表达式

选择表达式很像变量表达式，不过它们用一个预先选择的对象来代替上下文变量容器(map)来执行：*{customer.name}。两者的区别是：*{...} 会从选定的对象中匹配，而 ${...} 是从整个 context 中去匹配。

被指定的对象由 th:object 属性定义。如下所示，th:object 指定了一个对象，而使用*{...}表达式可以获取该对象下的属性值：

```
<div th:object="${user}">
    <p>FirstName: <span th:text="*{firstName}">Hello!!</span>.</p>
    <p>LastName: <span th:text="*{lastName}">World!!</span>.</p>
</div>
```

文字国际化表达式

Thymeleaf 支持文字国际化，使用#{...}语法定义，如下：

Welcome to our grocery store!

调用国际化的 welcome 语句,国际化资源文件如下：

```
resource_en_US.properties:

home.welcome=Welcome to here!


resource_zh_CN.properties:

home.welcome=欢迎您的到来!
```

URL 表达式

URL 表达式使用@{...}语法定义，@{...}支持绝对路径和相对路径。

其中相对路径又支持跨上下文调用 URL 和协议的引用（如：//code.jquery.com/jquery-2.0.3.min.js）

在 URL 表达式语法中，可以通过在 URL 的最后添加小括号“(...)”，并且在括号中定义动态生成的占位符、请求参数等内容（下面会举例说明）。

URL 绝对路径示例：

```
<a
th:href="@{http://localhost:8083/simple-thymeleaf/test/get-{num}.jso
n(num=${id})}">getUserById</a>
```

如上所示：可以直接在@{...}中定义一个绝对路径，并且在该 URL 中，存在一个路径请求参数占位符{num}，我们可以在 URL 的结尾，通过“(...)”形式和${...}语法定义该{num}占位符的动态值。这样一来该请求 URL 其实就跟 SpringMVC 中的前端控制器的@RequestMapping 定义的值一致。

包含请求参数的 URL 处理：

```
<a
th:href="@{http://localhost:8083/simple-thymeleaf/test/getUserByPara
m.json(userName=${userName},lastName=${lastName},gravatar=${gravatar}
)}">ReqByParam</a>
```

如上所示：当需要发送一个包含请求参数的 URL 请求时（形如：http://localhost:8080/webapp/detail?id=123&name=abc），使用连接符操作会较为复杂。Thymeleaf 可以通过上述方式直接发起包含请求参数的 URL 请求。

即：直接在完整的 URL 最后通过“(...)”设定请求参数名和值（多个参数用逗号分隔，参数值支持${...}语法），Thymeleaf 会自动将其作为请求参数处理。

常用 Th 标签表达式

集合遍历

在 Thymeleaf 标签中提供了遍历集合数据的标签，语法为 th:each。用法如下：

```
<!-- th:each标签：根据集合遍历，并且能够自动生成输出多个子节点 -->
<li th:each="user : ${userList}">
    FirstName: <ul th:text="${user.firstName}"></ul>
    LastName: <ul th:text="${user.lastName}"></ul>
</li>
```

#标签的常用表达式对象

在上下文变量评估 OGNL 表达式时，一些对象表达式可获得更高的灵活性。这些对象将由#号开始引用。Thymeleaf 提供了许多内置对象，提供了很多方便的如下功能，日期格式化，字符串处理，数字格式化，数组、集合操作等：

#dates	java.util.Date 对象的实用方法。
#calendars	和 dates 类似，但它是 java.util.Calendar 对象。
#numbers	格式化数字对象的实用方法。
#strigns, contains,startsWith, prepending/appending,etc	字符串对象的实用方法： contains, startsWith, prepending/appending 等。
#bools	对布尔值求值的实用方法。

#arrays	数组的实用方法。
#lists	List 的实用方法。
#sets	Set 的实用方法。
#maps	Map 的实用方法。
#messages, equal to using #{}	在表达式中获取外部信息的实用方法。
#ids	处理可能重复的 id 属性的实用方法。
#ctx 等等，还有很多。	略

处理数字和日期

如：#numbers 和#dates 标签用于处理数字和日期，如下图所示：

```
<!-- 简单数据转换（数字，日期）标签方法 -->
<dt>Price</dt>
<!-- 使用#numbers方法，将动态生成的product.price的值转为小数点后两位 -->
<dd th:text="${#numbers.formatDecimal(product.price, 1, 2)}">***</dd>

<dt>Date to yyyy-MM-dd</dt>
<!-- 使用#dates方法，将日期转为yyyy-MM-dd格式 -->
<dd th:text="${#dates.format(product.date, 'yyyy-MM-dd')}">today</dd>
```

拼接字符串

有时候需要动态的改变文本的某一段内容，这就需要用到字符串处理，可以使用双竖线||拼接字符串，如下所示：

```
<!-- 字符串操作 -->
<span th:text="'The name of the user is ' + ${user.firstName}" /><br>
<span th:text="'Welcome to our application, ' + ${user.firstName} + '!'" /><br>
<!-- 双竖线||符号内的字符做拼接处理，效果同上 -->
<span th:text="|Welcome to our application, ${user.firstName}!|" /><br>
<span th:text="${onevar} + ' ' + |${twovar}, ${threevar}|" /><br>
```

运算比较操作符

基础运算

基础的运算符包括加减乘除取余，比较操作符中如果存在 > 和 <，就需要转义处理，需要进行转义：

>	>
<	<
>=	≥或&gte;
<=	≤或 lte;
==	&eq;
!=	≠或&neq;

三元运算符

支持使用（X?A:B）形式的语法进行逻辑判断，如下图：

```
<!-- 运算操作符 -->
<div th:with="isEven=(${count} % 2 == 0)" />
<div th:if="${count} &gt; 1">
    <div
        th:text="'Execution mode is ' +
            ( (${execMode} == 'dev')? 'Development' : 'Production')" />
</div>
```

属性标签

前面提到了一些不属于 HTML 规范的属性标签，这都是 Thymeleaf 自定义的，通过这些属性标签来设置 HTML 标签的属性。

设置任意属性

Thymeleaf 的 th:attr 属性标签可以对 HTML 中任意属性值进行设置。例如 th:attr="action=abc"是对 action 属性的设置；th:attr="value=xyz”是对 value 属性的设置。示例代码如下：

```
<!--
    设置任意属性：如下，action=@{/subscribe}是对action属性
    设定为一个/subscribe的URL
 -->
<form action="subscribe.html" th:attr="action=@{/subscribe}">
    <fieldset>
        <input type="text" name="firstName" />
        <input type="text" name="lastName" />
        <input type="submit"
            value="Subscribe me!"
            th:attr="value=#{subscribe.submit}" />
    </fieldset>
</form>
```

还有许多其他的 th:* 自定义标签，都是和 HTML 标签一一对应的。

同时设置多个属性

如果需要 Thymeleaf 动态设置多个属性，可以像下面一样依次指定，也可以同时指定，不同属性通过“-”连接：th:alt-title 同时设置 alt 和 title 属性；th:lang-xmllang 同时设置 lang 和 xml:lang 属性。

```
<!-- 依次指定 -->
<img src="../../images/gtvglogo.png"
    th:attr="src=@{/images/gtvglogo.png},title=#{logo},alt=#{logo}" />
<img src="../../images/gtvglogo.png" th:src="@{/images/gtvglogo.png}"
    th:title="#{logo}" th:alt="#{logo}" />
<!-- 同时指定 -->
<img src="../../images/gtvglogo.png" th:src="@{/images/gtvglogo.png}"
    th:alt-title="#{logo}" />
```

前置和后置

有些情况下,可能需要改变属性的一部分值,比如 DOM 节点样式属性中的某个样式，这就需要在已有属性的基础上前置或后置新的属性。对此，Thymeleaf 提供了 th:attrappend 和 th:attrprepend 属性标签，如下所示：

```
<input type="button" value="Do it!" class="btn"
    th:attrappend="class=${' ' + cssStyle}" />
```

条件

Thymeleaf 的条件表达式包括 if，unless 和 switch。

th:if 语句

如果条件标签 th:if 的结果为真，则显示标签的内容，否则不显示，如下图：

```
<!-- 如果条件标签 th:if 的结果为真，则显示 a 标签的内容，否则不显示。 -->
<a href="test.html"
    th:href="@{/test/getUserByParam(userId=${id})}"
    th:if="${not #lists.isEmpty(users)}">view</a>
```

th:if 判 true 的情况有如下几种：

1	布尔值 true
2	非零数字
3	非零字符
4	除“false”“off”“no”以外的字符串
5	布尔值、数字、字符、字符串之外的变量形式

th:unless 语句

th:unless 的用法正好和 th:if 相反，要实现和上面代码一样的作用，th:unless 会这样写：

```
<a href="test.html"
    th:href="@{/test/getUserByParam(userId=${id})}"
    th:unless="${#lists.isEmpty(users)}">view</a>
```

th:switch 语句

th:switch 和多数编程语言的 switch-case 用法是很类似的，其中 default 情况的表达式是 th:case="*"，示例如下图：

```
<div th:switch="${user.role}">
  <p th:case="'admin'">User is an administrator</p>
  <p th:case="#{roles.manager}">User is a manager</p>
  <p th:case="*">User is some other thing</p>
</div>
```