

Thymeleaf使用手册

文档版本：20170418 18 April 2017

项目版本：3.0.5.RELEASE

项目网站：http://www.thymeleaf.org

英文原文：http://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html

中文翻译：http://www.cnblogs.com/jiangchao226/p/5900222.html

目录

[1. Thymeleaf简介 1](#_Toc25847)

[1.1. 什么是Thymeleaf 1](#_Toc22611)

[1.2. 什么样的模板可以使用Thymeleaf处理 1](#_Toc22567)

[1.3. 方言：标准方言 2](#_Toc22049)

[2. 古泰虚拟商店 3](#_Toc581)

[2.1. 商店站点 4](#_Toc28755)

[2.2. 创建并且配置一个模板引擎 7](#_Toc24742)

[2.2.1. 模板解释器(The Template Resolver) 8](#_Toc27239)

[2.2.2. 模板引擎(The Template Engine) 10](#_Toc24167)

[3. 使用文本(Using Texts) 10](#_Toc10831)

[3.1. 多语言欢迎页(A multi-language welcome) 10](#_Toc31699)

[3.1.1. 使用th:text和外部文本 12](#_Toc8385)

[3.1.2. 上下文 13](#_Toc31223)

[3.1.3. 执行模板引擎 15](#_Toc6402)

[3.2. 更多的文本和变量(More on texts and variables) 15](#_Toc29641)

[3.2.1. 非转义文本(Unescaped Text) 15](#_Toc21485)

[3.2.2. 使用和显示变量 16](#_Toc22922)

[4. 标准表达式语法(Standard Expression Syntax) 18](#_Toc12357)

[4.1. 信息(Messages) 19](#_Toc10324)

[4.2. 变量(Variables) 20](#_Toc29335)

[4.2.1. 基本对象表达式 22](#_Toc18157)

[4.2.2. 工具对象表达式 23](#_Toc4411)

[4.2.3. 在首页中使用格式化日期 24](#_Toc31317)

[4.3. 选定变量表达式 25](#_Toc19754)

[4.4. Url链接 26](#_Toc14254)

[4.4.1. 为首页制作一个简单菜单 28](#_Toc26315)

[4.4.2. 相对于服务器根的URL 28](#_Toc1640)

[4.5. 片段 29](#_Toc30181)

[4.6. 字面值 29](#_Toc31966)

[4.6.1. 文本值 29](#_Toc26195)

[4.6.2. 数值 29](#_Toc29381)

[4.6.3. 布尔值 30](#_Toc26417)

[4.6.4. null值 30](#_Toc8214)

[4.6.5. 标记符号token值 30](#_Toc18586)

[4.7. 追加文本 31](#_Toc21604)

[4.8. 字面值替换 31](#_Toc25650)

[4.9. 算数运算 31](#_Toc4536)

[4.10. 比较和判断 32](#_Toc20231)

[4.11. 条件表达式 32](#_Toc28517)

[4.12. 默认表达式（Elvis运算符） 33](#_Toc18332)

[4.13. 无操作标记 34](#_Toc3191)

[4.14. 日期的转换与格式化 35](#_Toc10513)

[4.15. 预处理(Preprocessing) 35](#_Toc22190)

[5. 设置属性值(Setting Attribute Values) 37](#_Toc13512)

[5.1. 属性值通用设置方式 37](#_Toc21018)

[5.2. 特定属性值设定的方式 38](#_Toc19959)

[5.3. 同时设置多个值 41](#_Toc11990)

[5.4. 追加和重写(Appending and prepending) 41](#_Toc24327)

[5.5. 拥有固定值的布尔属性 42](#_Toc9596)

[5.6. 设置自定义属性 43](#_Toc7857)

[5.7. HTML5自定义方式的支持 43](#_Toc6210)

[6. 迭代(Iteration) 44](#_Toc23350)

[6.1. 迭代基础 44](#_Toc9688)

[6.1.1. 使用 th:each 44](#_Toc25638)

[6.1.2. 可迭代的值 45](#_Toc1165)

[6.2. 保持迭代状态 46](#_Toc12178)

[6.3. 通过懒检索进行优化 47](#_Toc28483)

[7. 条件判断(Conditional Evaluation) 48](#_Toc32159)

[7.1. if和unless 49](#_Toc31533)

[7.2. switch 50](#_Toc5582)

[8. 模板的布局(Template Layout) 51](#_Toc9338)

[8.1. 引入模板片段 51](#_Toc21372)

[8.1.1. 定义和引用片段 51](#_Toc4436)

[8.1.2. 片段语法说明 52](#_Toc9059)

[8.1.3. 引用不包含th:fragment的片段 53](#_Toc17886)

[8.1.4. th:insert和th:replace的不同点(以及th:include) 54](#_Toc25126)

[8.2. 片段的参数 55](#_Toc6421)

[9. 本地变量(Local Variables) 71](#_Toc19496)

[10. Attribute Precedence 74](#_Toc31165)

[11. Comments and Blocks 75](#_Toc28996)

[11.1. Standard HTML/XML comments 75](#_Toc19400)

[11.2. Thymeleaf parser-level comment blocks 75](#_Toc29474)

[11.3. Thymeleaf prototype-only comment blocks 76](#_Toc10707)

[11.4. Synthetic th:block tag 77](#_Toc19250)

[12. Inlining 78](#_Toc29615)

[12.1. Expression inlining 78](#_Toc32157)

[12.2. Text inlining 80](#_Toc6673)

[12.3. JavaScript inlining 80](#_Toc2222)

[12.4. CSS inlining 84](#_Toc22746)

[13. Textual template modes 86](#_Toc7291)

[13.1. Textual syntax 86](#_Toc23709)

[14. Some more pages for our grocery 94](#_Toc17621)

[14.1. Order List 94](#_Toc30023)

[14.2. Order Details 95](#_Toc16333)

[15. 15 More on Configuration 97](#_Toc18201)

[15.1. Template Resolvers 98](#_Toc19791)

[16. Template Cache 105](#_Toc13390)

[17. Decoupled Template Logic 107](#_Toc26784)

[17.1. Decoupled logic: The concept 107](#_Toc19752)

[17.2. Configuring decoupled templates 110](#_Toc8902)

[17.3. The th:ref attribute 111](#_Toc16700)

[17.4. Performance impact of decoupled templates 112](#_Toc19737)

[17.5. Resolution of decoupled logic 113](#_Toc25646)

[18. Appendix A: Expression Basic Objects 113](#_Toc23964)

[19. Appendix B: Expression Utility Objects 118](#_Toc10559)

[20. Appendix C: Markup Selector Syntax 148](#_Toc709)

# Thymeleaf简介

## 什么是Thymeleaf

Thymeleaf是网站或者独立应用程序的新式的服务端java模板引擎，可以执行HTML，XML，JavaScript，CSS甚至纯文本模板。

Thymeleaf的主要目标是提供一个以优雅的高可维护型的方式创建模板，为了达到这个目的，他建立了自然模板的概念，以一种不影响模板设计原型的方式将逻辑注入到模板文件中，可以显著的减少设计和开发之间的沟通成本。

在需要的情况下，Thymeleaf可以创建一个各种校验模式的模板（尤其是基于HTML5）

## 什么样的模板可以使用Thymeleaf处理

Thymeleaf允许使用六种模板，每个被称为一种模板模式：

* HTML
* XML
* TEXT
* JAVASCRIPT
* CSS
* RAW

这其中有两种标记语言模板模式（HTML，XML），三种文本语言模板模式（TEXT，JAVASCRIPT，CSS），和一种无操作模板模式（RAW）

其中：

HTML模板模式可以使用任何HTML输入，包括HTML5，HTML4或XHTML，全验证或无验证都将可以被执行，模板中的代码和结构将被尽最大可能性的解析输出。

XML模板模式将被用来执行XML的输入，在这种模式下，如果被要求严格验证，则比如没有闭合标签，属性没有引号等，都将输出异常，注意一点是，无验证（通过DTD或XMLSchema）仍然是可以执行的。

文本模板模式将可以使用一种非标记性的特殊语法。它可能是一个电子邮件的模板或者文本模板等模板文件。注意：HTML模式或XML模式也可以使用文本模板模式处理，但在这种情况下，他们不会被解析为标记语言，每个标记，DOCTYPE，comment等都将视为单纯的文本。

JavaScript模板模式可以使用Thymeleaf处理Js文件。这意味着它能够在JS文件中以HTML文件同样的方式使用数据模型。但它还整合了一些JS的专有属性，如一些特定编码和脚本语言。JavaScript模板是一种支持特殊语法的文本模板模式。

CSS模板模式和JavaScript模板模式类似，即可以使用Thymeleaf处理CSS文件，它也是用支持特殊语法的文本模板模式。

RAW模板模式是一种非执行模板，他用了处理一些保持原样的资源（如文件，某Url的响应等），比如，在一些格式之外，可能会有一些信息需要原封不动的展现出来，这些应该明确的告知Thymeleaf不要讲它们进行执行。

## 方言：标准方言

Thymeleaf 是一种可扩展的模板引擎（其实应该称呼为模板引擎框架），它允许您自由的定义制作一个可以精确到任何程度的模板。

一个对象，运用一些逻辑（如标签，文本，注释，或仅仅是一个占位符），被称为处理器，这些处理器加一些额外的东西，被称为方言。其中Thymeleaf的核心库提供了一个一目了然的方言被称为标准方言，他应该能够满足大多数用户的需求。

应明确一点就是，方言实际上可以是没有处理器，完全有其他的类型的东西注册，但处理器是一种最常见的使用方式。

本教程即使用标准方言，你可以在下面页中了解这个方言的每一个属性和语法功能的定义，即使他没有明确提及。

当然，如果用户想要定义自己的处理逻辑，和使用库中的各种先进功能，那么，可以定义自己的方言（甚至扩展标准方言），而模板引擎可以一次配置多种方言。

官方thymeleaf-spring3和thymeleaf-spring4集成包都定义了另一种方言被称为“String标准方言”，大多数相当于标准方言，但有一小部分为了适应Spring框架的某些功能（如利用Spring的EL表达式而不是Thymeleaf标准的OGNL）所以，如果你是一个Spring mvc用户，请你不要在浪费时间抓紧学习，因为那你在这里学习的所有东西，都讲在你的Spring项目中使用

Thymeleaf标准方言可以在任何模式的模板中使用，尤其适合面向web的模板模式（如HTML5和XHTML），除了HTML5，他可以支持和验证一下的XHTML格式：XHTML 1.0 Transitional, XHTML 1.0 Strict, XHTML 1.0, 和 XHTML 1.1.

标准方言的大多数处理器都是属性处理器。他们在浏览器处理HTML模板文件之前，他会简单处理额外的属性，如，在jsp中使用标签库可以包括代码段，但代码段浏览器并不会直接显示一样：

<form:inputText name="userName" value="${user.name}" />

Thymeleaf中，可以实现同样的功能：

<input type="text" name="userName" value="niufennan" th:value="${user.name}" />

这个会在浏览器中被正确的显示，同时，也容许我们指定一个默认值（可选）（"niufennan"）,当静态文件在浏览器中打开的时候，显示的是Thymeleaf使用模板中${user.name}的值对value中的值进行了替换的结果。

这将可以使设计人员和开发人员使用几乎相同的模板文件，并且可以使用极少的工作，就使一个静态文件转换为开发模板文件。有这样能力的模板通常称之为自然模板

# 古泰虚拟商店

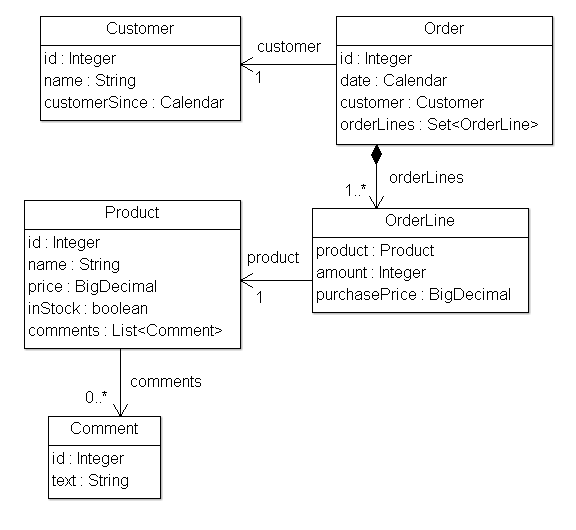
当前及未来示例代码下载地点

## 商店站点

为了更好的说明Thymeleaf的各种概念，教程所使用的Demo可以从项目网站下载。

这个项目是一个假想的购物网站，将提供足够的场景来演示Thymeleaf的各种不同的特点。

这个购物网站需要一个非常简单的实体模型，产品(Products)销售给客户(Customers),同时创建订单(Orders),并且，还需要我们管理用户对产品的评论(Coomments)



模型示意图

一个简单的服务层：

public class ProductService {

...

public List<Product> findAll() {

return ProductRepository.getInstance().findAll();

}

public Product findById(Integer id) {

return ProductRepository.getInstance().findById(id);

}

}

最后，web层将有一个过滤器，用来根据请求的url来执行Thymeleaf

private boolean process(HttpServletRequest request, HttpServletResponse response)

throws ServletException {

try {

//静态资源不使用框架

if(request.getRequestURI().startsWith("/css")||

request.getRequestURI().startsWith("/images")||

request.getRequestURI().startsWith("/favicon")

)

{

return false;

}

/\*

\* 根据controller/url的映射规则，获取将要执行的控制权的request

\* 如果没有控制权可用，返回false，让其他的filter或者servlet来执行

\*/

IGTVGController controller = application.getTemplateEngine().resolveControllerForRequest(request);

if (controller == null) {

return false;

}

/\*

\* 获取 TemplateEngine 实例.

\*/

ITemplateEngine templateEngine = application.getTemplateEngine();

/\*

\* 写 response headers

\*/

response.setContentType("text/html;charset=UTF-8");

response.setHeader("Pragma", "no-cache");

response.setHeader("Cache-Control", "no-cache");

response.setDateHeader("Expires", 0);

/\*

\* 执行控制权并处理模板

\* 将结果写入response

\*/

controller.process(

request, response, this.servletContext, templateEngine);

return true;

} catch (Exception e) {

throw new ServletException(e);

}

}

还有IGTVGController 接口

public interface IGTVGController {

public void process(

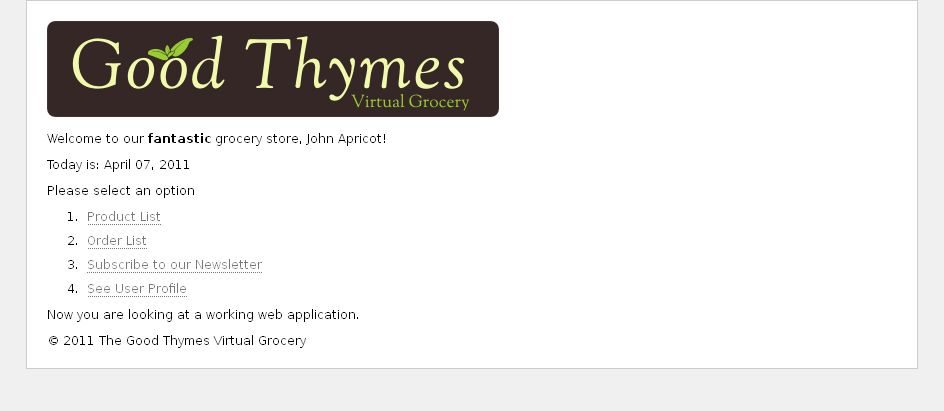
HttpServletRequest request, HttpServletResponse response,

ServletContext servletContext, ITemplateEngine templateEngine) throws Exception;

}

现在要做的就是IGTVGController接口的实现，主要实现了从服务中检索数据和处理模板使用的ITemplateEngine 对象

最后，他的效果图如下



效果图

但是，首先让我们看看模板引擎是如何初始化的。

## 创建并且配置一个模板引擎

在过滤器process方法中加入如下的代码：

ITemplateEngine templateEngine = application.getTemplateEngine();

这意味着GTVGApplication类中在加载的时候创建和配置了Thymeleaf启动应用的最重要对象：模板引擎实例(ITemplateEngine接口的实现)

我们的org.thymeleaf.TemplateEngine对象的初始化就像是这样：

public class GTVGApplication {

...

private static TemplateEngine templateEngine;

...

public GTVGApplication(ServletContext servletContext) {

ServletContextTemplateResolver templateResolver =

new ServletContextTemplateResolver(servletContext);

//有String类型重载

templateResolver.setTemplateMode(TemplateMode.HTML);

// 这段将把 "home" 转换为 "/WEB-INF/templates/home.html"

templateResolver.setPrefix("/WEB-INF/templates/");

templateResolver.setSuffix(".html");

//模板的缓存的生存时间默认为1小时，如果没有设置，将被缓存到LRU（缓存淘汰算法）中

templateResolver.setCacheTTLMs(3600000L);

//缓存默认为true，若希望修改模板后自动更改（如调试时），可设置为false

templateResolver.setCacheable(true);

templateEngine = new TemplateEngine();

templateEngine.setTemplateResolver(templateResolver);

}

...

}

当然，配置TemplateEngine对象的方式有很多，但这几行代码将让我们学会足够的必要步骤。

### 模板解释器(The Template Resolver)

从模板解释器开始：

ServletContextTemplateResolver templateResolver = new ServletContextTemplateResolver();

模板解释器对象从如下接口实现：

org.thymeleaf.templateresolver.ITemplateResolver:

public interface ITemplateResolver {

...

/\*

\* 模板通过名字或内容来解析。星期如果他有主模板的情况下还将尝试将此解析为另一个模板的片段

\* 如果这个模板不能通过解析，将返回null.

\*/

public TemplateResolution resolveTemplate(

final IEngineConfiguration configuration,

final String ownerTemplate, final String template,

final Map<String, Object> templateResolutionAttributes);

}

有这些解释器对象决定我们的模板在GTVG应用中将被如何访问，并在org.thymeleaf.templateresolver.ServletContextTemplateResolver中实现，我们将在Servlet的上下文中检索制定的模板文件资源。Servlet上下文指的是javax.servlet.ServletContext对象，他广泛的在每一个java web应用中存在。并将使用web应用程序的根作为资源文件的根路径。

但这不能说所有的模板都如此解析，因为我们可以给他设置一些配置参数。首先，模板模式中，其中一个标准为：

templateResolver.setTemplateMode("HTML");

HTML是ServletContextTemplateResolver的默认模板模式，

但出于代码可读性还是显示设置。

templateResolver.setPrefix("/WEB-INF/templates/");

templateResolver.setSuffix(".html");

顾名思义，这是模板路径的前缀和后缀，通过模板的名称，可以将真实的路径传递到引擎来使用它。

使用这个配置：模板名称"product/list"的路径将为：

servletContext.getResourceAsStream("/WEB-INF/templates/product/list.html")

通过配置为cachtTTLMS属性配置一个总的时间量，可以配置模板解释器在缓存中存活的时间：

templateResolver.setCacheTTLMs(3600000L);

当然，如果缓存达到了最大缓存，并且此模板是最早的缓存条目，那么在达到TTL之前，他还是有可能被清理出缓存的。

>缓存的行为和大小，都可以由用户自己定义，既可以实现ICacheManager接口自己实现规则，也可以通过修改StandardCacheManager对象设置缓存默认管理规则。

我们将在以后学习模板解释器，现在让我们来看看模板引擎对象。

### 模板引擎(The Template Engine)

模板引擎对象是接口org.thymeleaf.ITemplateENgine的实现,Themeleaf的core包提供一个默认的实现：org.thymeleaf.TemplateEngine,下边是创建引擎的一个例子：

templateEngine = new TemplateEngine();

templateEngine.setTemplateResolver(templateResolver);

很简单，是吧，我们所需要的仅仅是创建一个实例并设置一个模板解释器。

一个模板解释器是一个TemplateEngine必须的参数，当然，还需要其他的参数，将在后边介绍（如消息解释器，缓存大小等），现在，这些已经满足我们需要了。

我们的模板引擎现在已经可以启动，并可以使用Thymeleaf创建我们的页面了。

# 使用文本(Using Texts)

## 多语言欢迎页(A multi-language welcome)

我们的第一个任务是为我们的商店网站创建一个首页。

第一个版本，将是一个非常简单的页面：仅仅有个title和一个欢迎信息。在 \*\*/WEB-INF/template/home.html\*\* 文件：

<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">

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

<head>

<title>Good Thymes Virtual Grocery</title>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />

<link rel="stylesheet" type="text/css" media="all" href="../../css/gtvg.css" th:href="@{/css/gtvg.css}" />

</head>

<body>

<p th:text="#{home.welcome}">Welcome to our grocery store!</p>

</body>

</html>

首先你会注意到，这个文件是HTML5标准的DOCTYPE格式，他可以在主流浏览器中正确的显示（浏览器会忽略掉一下他不认识的属性，如th:text）。

但是，你也可能注意到，这个模板并不是一个完全有效的HTML5文档，因为有HTML5规格中没有的非标准属性，如th:\*属性，事实上，我们添加了一个xmlns:th属性在html标签，用来分类非html5的标记或属性。

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

那么，如何使用一个完全符合HTML5标准的模板呢？也很容易，使用Thymeleaf数据属性语法即可，它使用data-前缀的方式来使用：

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html>

<head>

<title>Good Thymes Virtual Grocery</title>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />

<link rel="stylesheet" type="text/css" media="all"

href="../../css/gtvg.css" th:href="@{/css/gtvg.css}" />

</head>

<body>

<p data-th-text="#{home.welcome}">欢迎您光临本店!</p>

</body>

</html>

自定义的data-前缀属性是html5完全支持的语法，这样，上面这段代码就是一个完全符合HTML5定义的文档。

这两种语法是完全等价的，可以无缝替换，但为了使代码更为简洁，本教程使用命名空间符号(th:),另外，th:语法是更常用的语法，在每一种Thymeleaf模板模式都是可以使用，但data-这种方式只能在HTML模式中使用。

### 使用th:text和外部文本

外部文本通常是模板文件中需要替换的代码，通常保存在外部的独立文件（通常是特定的.properties文件）中。他们可以很容易的设置其他语言的等效文本（这一过程通常称为国际化）。这些外在的文本片段通常被称为"messages"

Messages需要一个唯一性的Key进行标识，Thymeleaf允许你通过指定一个 \*\*#{...}\*\* 语法格式的Message对应一个具体的文本。如：

<p th:text="#{home.welcome}">欢迎您光临本店!(Welcome to our grocery store!)</p>

我们可以看到Thymeleaf标准方言的两个主要的语言点：

* th：text属性，他声明设置表达式的值，并使表达式返回的值来填充标签内容，替换或设置标签内部的内容，当前例子中即替换“欢迎光临本店”这些字。
* #{home.welcome}表达式，一个标准的表达式语法，指出在模板中，th:text属性说对应Message的key，即使用home.welcome对应的value替换现有内容。

现在，对应的外部文本在哪？

在Thymeleaf中这些文件的位置都是可配的，他通过实现org.thymeleaf.messageresolver.IMessageResolver接口进行配置，通常会实现一个基于.properties文件的实现，当然也同样可以根据实际情况进行自定义实现，如将Message存储在数据库中。

如果我们在模板引擎初始化的时候没有配置任何消息解释器的实现，那就意味着我们使用的是默认的一个消息解析实现，实现类为：org.thymeleaf.messageresolver.StandardMessageResolver.

这个消息解释器会扫描/WEB-INF/template/home.html相同目录下的相同名称的.properties文件，比如：

* /WEB-INF/templates/home\_en.properties 英文
* /WEB-INF/templates/home\_fr.properties 法语
* /WEB-INF/templates/home\_jp.properties 日语
* /WEB-INF/templates/home.properties 默认

其中的一个properties文件如下(en):

home.welcome=Welcome to our grocery store!

我们已经完成了我们所需要的Thymeleaf框架的模板，接下来创建Home的Controller。

### 上下文

为了处理模板，我们将创建一个实现了之前看到的IGTVTController接口的HomeController类：

public class HomeController implements IGTVGController {

@Override

public void process(HttpServletRequest request,

HttpServletResponse response,

ServletContext servletContext,

TemplateEngine templateEngine) throws Exception {

// TODO Auto-generated method stub

WebContext ctx=new WebContext(request,response,servletContext,

request.getLocale());

templateEngine.process("home", ctx,response.getWriter());

}

}

可以看到，第一件事就是创建了一个context，一个实现了org.thymeleaf.context.IContext接口的Thymeleaf上下文。上下文应该包含了模板引擎所执行所有所需数据的变量的map，并且引用了Locale所需的外部文件。

public interface IContext {

public Set<String> getVariableNames();

public Locale getLocale();

public boolean containsVariable(final String name);

public Object getVariable(final String name);

}

并且扩展了一个专用接口，org.thymeleaf.context.IWebContext,用于基于ServletApi的应用使用，如SpringMVC:

public interface IWebContext extends IContext {

public HttpServletRequest getRequest();

public HttpServletResponse getResponse();

public HttpSession getSession();

public ServletContext getServletContext();

}

Thymeleaf核心库为每个接口给出了一个实现：

* org.thymeleaf.context.Context implements IContext
* org.thymeleaf.context.WebContext implements IWebContext

正如在controller的代码中看到的那样，我们使用了WebContext，事实上我们也必须使用它，因为这是ServletContextTemplateResolver必须使用一个IWebContext的实现类。

WebContext ctx = new WebContext(request, response，servletContext, request.getLocale());

这个构造函数的四个参数中有三个是必须的，第四个参数系统可以设置一个默认的本地区域，如果没有设置，则使用系统默认。

从WebContext还提供了一些专用的表达式，可以在模板中获得request的参数，request，session，application的属性等，比如：

${x}:返回一个Thymeleaf的context中存储的变量或request的属性。

${param.x} 返回request的参数x（可以为复合值）

${session.x} 返回session的属性x

${application.x} 返回一个servlet上下文的属性x

就在执行前，一个特殊的变量被设置为包含了Context和WebContext的全context对象（即所有实现了IContext的对象），被称为执行信息（execInfo）,这个变量有两个您可以从模板中使用的数据。

模板名：#{execInfo.templateName},一个引擎执行时的特定名称，对应正在执行的模板。

当前的日期时间：#{execInfo.now}，一个Calender对象，对应引擎开始执行的时刻值。

### 执行模板引擎

随着context对象已经准备好，剩下只需要指定模板名称和context，以执行模板引擎，兵传递给response的writer，以写入响应。

templateEngine.process("home", ctx, response.getWriter());

看看执行结果：

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

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

<head>

<title>Good Thymes Virtual Grocery</title>

<meta content="text/html; charset=UTF-8" http-equiv="Content-Type"/>

<link rel="stylesheet" type="text/css" media="all" href="/gtvg/css/gtvg.css" />

</head>

<body>

<p>¡Bienvenido a nuestra tienda de comestibles!</p>

</body>

</html>

## 更多的文本和变量(More on texts and variables)

### 非转义文本(Unescaped Text)

一个最简版的Home页面似乎已经准备好了，但是，如果有这样的需求怎么办呢：

home.welcome=欢迎光临这个<b>超级棒</b>的商店!

如果是这样的，输出结果为：

home.welcome=欢迎光临这个&lt;b&gt;超级棒 &lt;/b&gt;的商店!

很显热，这并不是我希望看见的，我们希望b作为一个标签来处理，而不是转义后显示在浏览器上。

这是浏览器对th:text的默认行为。如果我们希望Thymeleaf直接使用html标签，而不是转义他们，我们可以使用不同的属性th:utext("unescaped text")

<p th:utext="#{home.welcome}">欢迎您光临本店!</p>

这样输出信息就是所需要的了:

<p>欢迎光临这个<b>超级棒</b>的商店!</p>

### 使用和显示变量

如果我想在首页多现实一下信息，比如显示日期，就像下边这样：

欢迎这个超级棒的网店

当前日期为：2016-9-2

那么首先，我们应该回到首页，添加当前时间到context变量：

WebContext ctx=new WebContext(request,response,servletContext,request.getLocale());

//时间

SimpleDateFormat sdf =new SimpleDateFormat("yyyy-MM-dd");

Calendar cal=Calendar.getInstance();

ctx.setVariable("today",sdf.format(cal.getTime()));

templateEngine.process("home", ctx,response.getWriter());

public void process(

HttpServletRequest request, HttpServletResponse response,

ServletContext servletContext, TemplateEngine templateEngine) {

SimpleDateFormat dateFormat = new SimpleDateFormat("dd MMMM yyyy");

Calendar cal = Calendar.getInstance();

WebContext ctx =

new WebContext(request, response, servletContext, request.getLocale());

ctx.setVariable("today", dateFormat.format(cal.getTime()));

templateEngine.process("home", ctx, response.getWriter());

}

我们已经添加了一个String变量在context中，现在修改模板：

<p th:utext="#{home.welcome}">欢迎您光临本店!</p>

<p>当前日期为: <span th:text="${today}">2016年9月5日</span></p>

<body>

<p th:utext="#{home.welcome}">Welcome to our grocery store!</p>

<p>Today is: <span th:text="${today}">13 February 2011</span></p>

</body>

可以看到，我们仍然使用th:text属性，但是语法稍有不同，这次使用的是${...}而不是#{...},这是一个用来显示变量值的表达式。它使用OGNL语言来映射context中的变量。

\({today}表达式只是简单的表示为：获取一个叫today的变量，但是这个表达式也可以变得很复杂，如\){user.name}的意思是获取一个user变量，并调用他的getName()方法。

属性值有相当多的可能性，如：消息，变量表达式等等，下一章将展示这些都是什么.

# 标准表达式语法(Standard Expression Syntax)

在将这个小demo发展成一个真正的网店之前，先休息一下，熟悉一下Thymeleaf标准方言中最重要的组成部分：Thymeleaf标准表达式语法。

在之前，已经看到过两种有效的属性表达式：消息和变量表达式：

<p th:utext="#{home.welcome}">欢迎您光临本店!</p>

<p>当前日期为: <span th:text="${today}">2016年9月5日</span></p>

但还有更多的类型和有趣的细节，我们都还不知道，下面我们首先来一个标准表达式的快速总结：

简单表达式

变量表达式:${...}

选定变量表达式:\*{...}

信息表达式:#{...}

链接表达式:@{...}

片段表达式:~{...}

字面值

文本值:'one text','Another one!',...

数值:1,34,3.0,12.3...

布尔值:true,false

Null值：null

标记符号值（token）:one,sometext,main,...(?)

操作符

字符串连接:+

文本替换:|The name is ${name}|

算数运算符：

基本操作符:+,-,\*,/,%

取负值(一元运算符):-

boolean运算符:

基本二元操作符:and,or

一元操作符:!,not

比较和判断:

比较运算符:>,<,>=,<=,(gt,le,ge,le)

判断相等运算符:==,!=(eq,ne)

条件运算符

if-then:(if)?(then)

if-then-else:(if)?(then):(else)

default:(value)?:(defaultvalue)

特殊标记

无操作:\_

所有这些还可以合并嵌套使用：

'User is of type '+(${user.isAdmin()}?'Administrator':(${user.type}?:'Unknown'))

## 信息(Messages)

我们已经知道，消息表达式许可我们将：

<p th:utext="#{home.welcome}">Welcome to our grocery store!</p>

连接到资源，将内容转换为：

home.welcome=欢迎您光临本店!

home.welcome=¡Bienvenido a nuestra tienda de comestibles!

但是，如果文本内容不是完全静态的怎么办？例如，我们想实现在已知某位用访问本站的时候（登录后），在页面显示他的名字：

<p>张三，您好！欢迎您光临本店!</p>

<p>¡Bienvenido a nuestra tienda de comestibles, John Apricot!</p>

这意味着我们需要一个参数：

home.welcome={0}，您好！欢迎您光临本店!

home.welcome=¡Bienvenido a nuestra tienda de comestibles, {0}!

参数是根据java.text.MessageFormat标准语法指定，意味着你可以添加数字，日期等api指定的格式。

为了给参数赋值，可以给一个session的属性为用户：

<p th:utext="#{home.welcome(${session.user.name})}">

张三，您好！欢迎您光临本店！

</p>

如果需要，可以指定多个参数，用逗号分隔，事实上，消息的key本身也可以是一个变量：

<p th:utext="#{${welcomeMsgKey}(${session.user.name})}">

张三，您好！欢迎您光临本店！

</p>

## 变量(Variables)

我们已经知道${...}表达式实际上是OGNL表达式在执行context中映射的变量。

>更多关于OGNL语法和功能的详细信息，可以阅读http://commons.apache.org/ognl/

在SpringMVC的应用中，OGNL将被SpringEL替代，但两种语法非常相似，常用的语法完全相同。

通过OGNL定义，我们可以指定

<p>当前日期为：: <span th:text="${today}">2016-8-13</span>.</p>

事实上，是这样实现的：

ctx.getVariables("today");

OGNL也允许我们创建更强大的表达式，比如说这样：

<p th:utext="#{home.welcome(${session.user.name})}">

张三，您好！欢迎您光临本店！

</p>

实际上他是这样执行的：

((User)ctx.getVariables("session").get("user")).getName();

通过getter方法导航是OGNL语法的一大特点:下面演示更多用法：

//访问属性使用（.）进行访问，相当于调用getter方法

${person.father.name}

//访问属性也可以通过方括号[]进行访问，属性名作为一个变量通过单引号(')或双引号(")写在方括号中

\*\*${person['father']['name']}\*\*

//如果对象是一个map，可以用引号和方括号的语法将相当于直营一个调用它的key获取对象的get方法。

${countriesByCode.ES}

${personsByName['张三'].age}

//对于数组或集合的索引也用方括号来执行

${personsArray[0].name}

//可以调用各种方法，无论是否有参数

${person.createCompleteName()}

${person.createCompleteNameWithSeparator('-')}

### 基本对象表达式

当使用OGNL表达式的context变量的时候，可以使用更方便的表达方式。这些对象也是用来#符号:

#ctx: context对象

#vars:context属性

#locale:context本地化信息

#request:(仅限WebContext) HttpServletRequest对象

#response:(仅限WebContext) HttpServletResponse对象

#session:（仅限WebContext）HttpSession对象

#servletContext:（仅限WebContext）ServletContext对象

所以我们可以这样使用

这里是:<span th:text="${#locale.country}">中国</span>.

你可以在[[附录A](#_18 Appendix A: Expression Basic Objects)]中查看这些对象的全部参考

### 工具对象表达式

除了基本对象，Thymeleaf还为我们提供了一套实用对象，可以帮助我们在执行表达式中解决一下常见的任务:

#execInfo:正在处理的模板信息

#messages：获取外部信息的内部变量的一个实用方法，同时也可以用#{...}获取

#uris:针对URL或URI进行一些转码的方法

#conversions:根据配置执行一些转换方法

#dates:针对java.util.Date对象的实用方法：包括日期格式化，日期提取等。

#calendars:与#dates相似，但针对的是java.util.Calendar对象。

#numbers:针对numeric对象格式化的实用方法

#strings:针对String对象的实用方法，包括包含，判断起始，前/后追加等方法

#objects:针对object类的一般实用方法

#boolean：针对boolean运算的一些实用方法

#arrays:针对数组的实用方法

#lists：针对list的实用方法

#sets：针对set的实用方法

#map:针对map的实用方法

#aggregates:在数组或集合中创建聚合的一些实用方法

#ids：用于处理可能重复的标识属性的使用方法，例如，作为迭代的变量。

你可以在[附录B](http://www.thymeleaf.org/doc/tutorials/2.1/usingthymeleaf.html#appendix-b-expression-utility-objects)中查看这些工具对象的全部参考

### 在首页中使用格式化日期

现在，我们知道了这些工具对象表达式，就可以改变首页中显示日期的方式，首先，改变我们HomeController中的代码，将：

//SimpleDateFormat dateFormat = new SimpleDateFormat("dd MMMM yyyy");

//Calendar cal = Calendar.getInstance();

SimpleDateFormat sdf =new SimpleDateFormat("yyyy-MM-dd");

Calendar cal=Calendar.getInstance();

ctx.setVariable("today",sdf.format(cal.getTime()));

templateEngine.process("home", ctx, response.getWriter());

这三行可以合并为1行

ctx.setVariable("today",Calendar.getInstance());

然后，修改模板文件的相应行为：

<p>

当前日期为： <span th:text="${#calendars.format(today,'yyyy-MM-dd')}">2016年9月5日</span>

</p>

## 选定变量表达式

变量表达式不但可以使用 \*\*${...}\*\* 表达式，同时也可以使用 \*\* \*{...}\*\* 表达式。

他们有个最重要的区别：\*{...}表达式的值是在选定的对象而不是整个context的map。也就是说，如果没有选定的对象，\*{...}和${...}没有区别

那么问题来了：什么是选定对象？一个 \*\*th:object\*\* 对象属性，使用用户权限页面来演示一下：

<div th:object="${session.user}">

<p>姓名：<span th:text="\*{firstName}">张三</span></p>

<p>年龄：<span th:text="\*{age}">26</span></p>

<p>国籍：<span th:text="\*{nationlity}">中国</span></p>

</div>

这相当于：

<div>

<p>姓名：<span th:text="${session.user.firstName}">张三</span></p>

<p>年龄：<span th:text="${session.user.age}">26</span></p>

<p>国籍：<span th:text="${session.user.nationlity}">中国</span></p>

</div>

当然，也可以混合使用

<div th:object="${session.user}">

<p>姓名：<span th:text="\*{firstName}">张三</span></p>

<p>年龄：<span th:text="${session.user.age}">26</span></p>

<p>国籍：<span th:text="\*{nationlity}">中国</span></p>

</div>

当一个使用选定对象的地方，选定的对象其实就是使用了#object表达式的${...}表达式

<div th:object="${session.user}">

<p>姓名：<span th:text="${#object.firstName}">张三</span></p>

<p>年龄：<span th:text="${session.user.age}">26</span></p>

<p>国籍：<span th:text="${#object.nationlity}">中国</span></p>

</div>

也就是说，如果没有已经完成的选定对象，那么，\*{...}和${...}两种表达式是完全等价的。

<div>

<p>姓名：<span th:text="\*{session.user.firstName}">张三</span></p>

<p>年龄：<span th:text="\*{session.user.age}">26</span></p>

<p>国籍：<span th:text="\*{session.user.nationlity}">中国</span></p>

</div>

## Url链接

由于其重要性，URL是web应用程序模板的一等公民，Thymeleaf的标准方言都为它定义了特殊语法：@{...}

他有不同类型的网址：

绝对地址，比如：https://niufennan.github.io/

相对地址，可以有如下方式：

相对于页的，比如：user/login.html

相对于上下午的，比如：/itemdetails?d=3(将自动添加服务器的上下文名称)

相对于服务器的，比如：~/billing/processInvoice(可以在同一服务器中的不同上下文（即application）中使用)

相对于协议的，比如：//cdn.bootcss.com/jquery/2.2.3/jquery.min.js

真正将这些表达式转换并输出为URL的工具，是一个注册在ITemplateEngine中的一个org.thymeleaf.linkbuilder.ILinkBuilder接口的实现类。

Thymeleaf可以在任何情况下处理绝对地址，但相应的，也会要求你给予一个实现了IWebContext接口的context对象，他包含了一些创建链接需要的来自Http请求的相关信息。

在默认情况下，注册的实现是类org.thymeleaf.linkbuilder.StandardLinkBuilder，它对于基于Servlet API的网络或离线应用已经足够了，而其他情况（如非ServletAPI的web项目）则可能需要自己创建接口的实现。

举例说明一下，需使用th:href属性：

<!-输出： 'http://localhost:8080/gtvg/order/details?orderId=3' -->

<a href="details.html" th:href="@{http://localhost:8080/gtvg/order/details(orderId=${o.id})}">view</a>

<!-输出： '/gtvg/order/details?orderId=3' -->

<a href="details.html" th:href="@{/order/details(orderId=${o.id})}">view</a>

<!-输出： '/gtvg/order/3/details' -->

<a href="details.html" th:href="@{/order/{orderId}/details(orderId=${o.id})}">view</a>

注意：

th:href是一个修饰属性，一旦设置这个属性，它会计算链接，并设置标签内的href属性的url值。

可以对url的参数使用表达式(比如orderId=${o.id})，url上所需的编码工作，也会自动执行。

如果需要多个参数，用逗号(,)分开即可如：(@{/order/process(execId=${execId},execType='FAST')})

网络路径中也可以使用变量模板，如：@{/order/{orderId}/details(orderId=${orderId})}

相对URL使用/开始，如/order/details，将自动适应上下文的名词前缀。

如果不清楚cookie是否被启用，一个jsessionid=...的后缀可能被加入到url中，用于回话保存，这就是所谓的URL重写。Thymeleaf允许利用Servlet的api为每一个url，使用response.encoding来扩展重写过滤器。

th:href允许有一个静态工作的url配置在模板中，这样，我们的模板即使在不工作时，仍然可以通过浏览器互相连接。

就像#{...}一样，URL表达式中的值也可以是另一个表达式的结果：

<a th:href="@{${url}(orderId=${o.id})}">view</a>

<a th:href="@{'/details/'+${user.login}(orderId=${o.id})}">view</a>

### 为首页制作一个简单菜单

现在已经知道了如何创建一个连接，是时候给首页增加一个小菜单了，以告诉大家这个站点还有些什么内容。

<p>请选择：</p>

<ol>

<li><a href="product/list.html" th:href="@{/product/list}">产品列表</a></li>

<li><a href="order/list.html" th:href="@{/order/list}">订单列表</a></li>

<li><a href="subscribe.html" th:href="@{/subscribe}">订阅新闻</a></li>

<li><a href="userprofile.html" th:href="@{/userprofile}">用户权限</a></li>

</ol>

### 相对于服务器根的URL

还有一个附加的语法，可用于指向一个服务器的根地址（而不是上下午的根地址），以便于指向同一服务器中的不同上下文。他的语法为：\*\*@{~/path/to/something}\*\*

## 片段

片段表达式是用一种简单的方式来标识一个片段的标记，并可以将他移动的其余模板，它允许我们执行重写模板，传递给其他模板参数等操作。

做常用的片段使用方式是插入使用，属性值为th:insert或th:replace（更多内容见后边部分）

<div th:insert="~{commons :: main}">...</div>

它可以在任何地方使用，就像其他变量一样：

<div th:with="frag=~{footer :: #main/text()}">

<p th:insert="${frag}">

</div>

本教程稍后的部分有一篇完整的关于模板布局的介绍，其中包含变量表达式更深层次的介绍。

## 字面值

### 文本值

字面值指的是包含在单引号之间字符，它可以使任何字符，但应该尽量避免使用\'

<p>

Now you are looking at a <span th:text="'working web application'">template file</span>.

</p>

现在你看到的是模板文件.

### 数值

顾名思义，数值显示的就是一个数字：

<p>今年是<span th:text="2016">1942</span>.</p>

<p>两年后，将是<span th:text="2013 + 2">1944</span>.</p>

### 布尔值

布尔值只有true和false两个值，举例：

<div th:if="${user.isAdmin()} == false"> ...

注意，在这个例子中，==false是写在了\({...}的外边，所以使Thymeleaf本身在支持它，如果写在了\){...}的里边，则变为由OGNL或SpringEL库来支持它。

<div th:if="${user.isAdmin() == false}"> ...

### null值

null值可以这样使用

<div th:if="${variable.something} == null"> ...

### 标记符号token值

实际上，数值，布尔值，null值都是一种特殊的token值。

这些token值允许为标准表达式进行一点点的简化，他们的工作方式和文本值完全一样，但是只能使用字符(a-z,A-Z),数值(0-9),括号即[ ]和( ),

所以不能有空格，括号等。

并且，他可以不被包含在单引号中：

<div th:class="content">...</div>

而不是这样(这样为文本值)：

<div th:class="'content'">...</div>

## 追加文本

一段文本，无论是一段字面值，变量的返回值，还是信息表达式，都可以使用+来追加一段文本

<div th:text="'这个人的名字为： ' + ${user.name}">

## 字面值替换

还可以直接替换一段字面值中包含的字符串信息，而无需通过操作符 \*\*'...'+'...'\*\* .

这些替换必须被竖线（|）包围，如：

<span th:text="|欢迎光临这个应用, ${user.name}!|">

即相当于：

<span th:text="'欢迎光临这个应用, ' + ${user.name} + '!'">

格式替换也可以和其他表达式相结合使用：

<span th:text="${onevar} + ' ' + |${twovar}, ${threevar}|">

注意：只有变量表达式${...}可以出现在|...|中，其他的如布尔，数值或文字的token，条件表达式等都不可以使用。

## 算数运算

在表达式中还可以使用一些算数运算，如：+,-,\*,/,%

<div th:with="isEven=(${prodStat.count} % 2 == 0)" >

注意：同布尔值的例子一样，这个同样可以写成使用OGNL或SpringEL支持的方式：

<div th:with="isEven=${prodStat.count % 2 == 0}">

注意：有两个运算符存在别名：div(/),mod(%)

## 比较和判断

表达式中的值可以通过>, <, >=, <=进行比较，也可以通过 == 和 != 操作符进行相等的判断。注意在xml中使用的时候，不应该直接使用<, >，应当使用 \*\*&lt;\*\* 和 \*\*&gt;\*\* 来代替

<div th:if="${prodStat.count} &gt; 1">

<div th:text="'执行模式为 ' + ( (${execMode} == 'dev')? 'Development' : 'Production')">

注意，这些都存在着别名：gt(>),lt(<),ge(>=),le(<=),not(!),eq(==),neq/ne(!=)

## 条件表达式

条件表达式为根据条件（另一个表达式）来选择两个表达式中的一个。

看一个示例：

<tr th:class="${row.even}? 'even' : 'odd'">

...

</tr>

条件表达式的所有的三个部分均在一个自表达式中，意味着他可以用在变量${...},\*{...}，信息#{...},URL@{...}或字面量('...')中.

他还可以使用()来进行嵌套

<tr th:class="${row.even}? (${row.first}? 'first' : 'even') : 'odd'">

...

</tr>

else部分可以省略，如果条件为否，则为一个null值

<tr th:class="${row.even}? 'alt'">

...

</tr>

## 默认表达式（Elvis运算符）

默认表达式是一种特殊的没有then部分的条件表达式。在一些语言，如Groovy中，它相当于Elvis运算符，它允许有两个表达式，第二个只有在第一个返回null的时候才执行。

修改一下用户权限页，如：

<div th:object="${session.user}">

...

<p>年龄: <span th:text="\*{age}?: '(没有输入)'">27</span>.</p>

</div>

正如看到的那样，使用?:操作符，我们在指定当年龄无效的时候，给定一个默认值（使用字面值），这条代码相当于：

<p>年龄: <span th:text="\*{age != null}? \*{age} : '(没有输入)'">27</span>.</p>

和条件值一样，他也可以使用括号来实现嵌套：

<p>

姓名:

<span th:text="\*{name}?: (\*{admin}? 'Admin' : #{default.username})">张三</span>

</p>

## 无操作标记

无操作标记是使用下划线表示(\_)

这背后是想指定一个标记来表达期望的结果是什么也不做，也就是说，如果处理的属性（如th:text）没有值。

这将允许开发人员使用原型文本作为默认值，如：

<span th:text="${user.name} ?: 'no user authenticated'">...</span>

将可以改为：

<span th:text="${user.name} ?: \_">no user authenticated</span>

直接使用原型，可以使使代码更加灵活

## 日期的转换与格式化

Thymeleaf可以使用双大括号的方式，为变量${...}和选择\*{...}表达式通过配置转换服务进行数据转换。

它基本上是这样的:

<td th:text="${{user.createTime}}">...</td>

这个双大括号${{}}通知Thymeleaf对user.createTime表达式的结果进行转换服务，并在输出结果之前进行格式化操作。

转换服务器默认使用IStandardConversionService的实现类（StandardConversionService类），它只能够简单的执行一个对象的toString()，即将一个对象转为字符串服务，有关注册一个转换服务的更多信息，可以查看稍后更多配置部分。

在thymeleaf-spring3和thymeleaf-spring4有一整套基于Spring转换服务的Thymeelaf转换服务配置，因此他可以自动实现${{}}和\*{{}}的服务。

## 预处理(Preprocessing)

Thymeleaf表达式除了这些特征，还为我们提供了预处理的功能。

预处理是指在正常的完成一个表达式之前执行的工作。他允许对最终执行的实际表达式进行修改.

一个预处理表达式和一个正常表达式几乎一样，但他在双下划线之间(\*\*\_\_$(表达式)\_\_\*\*）

比如，国际化的配置文件message\_jp.properties的一个条目包含了一个OGNL表达式用于调用一个静态方法:

article.text=@myapp.translator.Translator@translateToJapaese({0})

以及另一个等效的Message\_en.properties:

article.text=@myapp.translator.Translator@translateToEnglish({0})

对此，可以先创建一个用于标记的表达式，这个表达式的值取决于区域设置，为此，使用预处理，然后让Thymeleaf去执行它：

<p th:text="${\_\_#{article.text('textVar')}\_\_}">一些文字...</p>

例如日本，上边预处理与如下内容等效：

<p th:text="${@myapp.translator.Translator@translateToJapaese(textVar)}">Some text here...</p>

预处理中如有字符串\_\_可以使用\\_\\_来进行转义。

# 设置属性值(Setting Attribute Values)

本章将介绍如何设置或修改标签属性的值，下一章将讲解如何设置内容的值。

## 属性值通用设置方式

我们的网站会不定期发布一些新闻，我们希望我们的用户能够订阅他，所以我们创建一个/WEB-INF/templates/subscribe.html的表单模板：

<form action="subscribe.html">

<fieldset>

<input type="text" name="email" />

<input type="submit" value="订阅" />

</fieldset>

</form>

这个看起来不错，但是，这个更像是一个普通的html页面，首先，这个表单的action是指向了模板文件本身，并没有重写URL，第二，提交按钮的值，他是一个普通的文本，而我们希望他是一个国际化的。

使用th:attr属性，它具有设置或修改一个标签属性值的能力。

<form action="subscribe.html" th:attr="action=@{/subscribe}">

<fieldset>

<input type="text" name="email" />

<input type="submit" value="订阅!" th:attr="value=#{subscribe.submit}"/>

</fieldset>

</form>

用法很简单:th:attr将是一个值对应一个属性的表达式，在转换处理后，将会返回如下结果：

<form action="/gtvg/subscribe">

<fieldset>

<input type="text" name="email" />

<input type="submit" value="subscribe me!"/>

</fieldset>

</form>

除了更新了属性值，还可以看到，应用的已经自动将url更新为context前缀的url。

如果，我们想在同时更新多个属性呢？xml的规则不允许在一个标签内设置两个同名的属性，所以，可以用逗号来分割th:attr的值，比如：

<img src="../../images/gtvglogo.png" th:attr="src=@{/images/gtvglogo.png},title=#{logo},alt=#{logo}" />

将转换为：

<img src="/gtgv/images/gtvglogo.png" title="这里是logo" alt="这里是logo" />

## 特定属性值设定的方式

到了现在，你可能会发现，像：

<input type="submit" value="订阅" th:attr="value=#{subscribe.submit}"/>

这种，看起来是一个非常难看的模板。这种指定属性值的赋值方式很明显不是最优雅的一种创建方式模板。

事实也的确如此，这也就是为什么th:attr很少出现在实际应用的模板中，实际中经常使用的是th:\*来实现标签中特定的属性。

在Thymeleaf的标准方言中，通常都是很直观的方式，如设置一个按钮的value值，使用th:value表达式，如：

<input type="submit" value="订阅" th:value="#{subscribe.submit}"/>

看起来感觉好多了，然后在看一下表单中的action属性：

<form action="subscribe.html" th:action="@{/subscribe}">

在之前home.html模板中的th:href，其实就在使用的这个语法：

<li><a href="product/list.html" th:href="@{/product/list}">产品列表</a></li>

Thymeleaf针对每一个属性都有一个特定的设置语法：

|  |  |  |
| --- | --- | --- |
| th:abbr | th:accept | th:accept-charset |
| th:accesskey | th:action | th:align |
| th:alt | th:archive | th:audio |
| th:autocomplete | th:axis | th:background |
| th:bgcolor | th:border | th:cellpadding |
| th:cellspacing | th:challenge | th:charset |
| th:cite | th:class | th:classid |
| th:codebase | th:codetype | th:cols |
| th:colspan | th:compact | th:content |
| th:contenteditable | th:contextmenu | th:data |
| th:datetime | th:dir | th:draggable |
| th:dropzone | th:enctype | th:for |
| th:form | th:formaction | th:formenctype |
| th:formmethod | th:formtarget | th:fragment |
| th:frame | th:frameborder | th:headers |
| th:height | th:high | th:href |
| th:hreflang | th:hspace | th:http-equiv |
| th:icon | th:id | th:inline |
| th:keytype | th:kind | th:label |
| th:lang | th:list | th:longdesc |
| th:low | th:manifest | th:marginheight |
| th:marginwidth | th:max | th:maxlength |
| th:media | th:method | th:min |
| th:name | th:onabort | th:onafterprint |
| th:onbeforeprint | th:onbeforeunload | th:onblur |
| th:oncanplay | th:oncanplaythrough | th:onchange |
| th:onclick | th:oncontextmenu | th:ondblclick |
| th:ondrag | th:ondragend | th:ondragenter |
| th:ondragleave | th:ondragover | th:ondragstart |
| th:ondrop | th:ondurationchange | th:onemptied |
| th:onended | th:onerror | th:onfocus |
| th:onformchange | th:onforminput | th:onhashchange |
| th:oninput | th:oninvalid | th:onkeydown |
| th:onkeypress | th:onkeyup | th:onload |
| th:onloadeddata | th:onloadedmetadata | th:onloadstart |
| th:onmessage | th:onmousedown | th:onmousemove |
| th:onmouseout | th:onmouseover | th:onmouseup |
| th:onmousewheel | th:onoffline | th:ononline |
| th:onpause | th:onplay | th:onplaying |
| th:onpopstate | th:onprogress | th:onratechange |
| th:onreadystatechange | th:onredo | th:onreset |
| th:onresize | th:onscroll | th:onseeked |
| th:onseeking | th:onselect | th:onshow |
| th:onstalled | th:onstorage | th:onsubmit |
| th:onsuspend | th:ontimeupdate | th:onundo |
| th:onunload | th:onvolumechange | th:onwaiting |
| th:optimum | th:pattern | th:placeholder |
| th:poster | th:preload | th:radiogroup |
| th:rel | th:rev | th:rows |
| th:rowspan | th:rules | th:sandbox |
| th:scheme | th:scope | th:scrolling |
| th:size | th:sizes | th:span |
| th:spellcheck | th:src | th:srclang |
| th:standby | th:start | th:step |
| th:style | th:summary | th:tabindex |
| th:target | th:title | th:type |
| th:usemap | th:value | th:valuetype |
| th:vspace | th:width | th:wrap |
| th:xmlbase | th:xmllang | th:xmlspace |

## 同时设置多个值

还有两个比较特殊的属性即 \*\*th:alt-title\*\* 和 \*\*th:lang-xmllang\*\* ,他们能同时设置两个属性:

* th:alt-title 同时设置alt和title
* th:lang-xmllang 同时设置lang和xml:lang

不然修改首页中刚刚的logo图标签：

<img src="../../images/gtvglogo.png" th:attr="src=@{/images/gtvglogo.png},title=#{logo},alt=#{logo}" />

修改为:

<img src="../../images/gtvglogo.png" th:src="@{/images/gtvglogo.png}" th:title="#{logo}" th:alt="#{logo}" />

或者：

<img src="../../images/gtvglogo.png" th:src="@{/images/gtvglogo.png}" th:alt-title="#{logo}" />

## 追加和重写(Appending and prepending)

Thymeleaf还提供了th:attrappend和th:attrprepend属性，用于为属性之前或之后增加值

比如，你可能想在一个按钮的现有css类的基础上在新增一个css类，将会非常容易：

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

如果你对cssStyle变量的值为warning，那么输出将为：

<input type="button" value="点击" class="btn warning" />

因为样式表使用的如此频繁，所以标准方言中还有两个附加属性，th:classappend和th:styleappend属性，用于追加一个class类或者一段样式表而不改变现有内容：

<tr th:each="prod : ${prods}" class="row" th:classappend="${prodStat.odd}? 'odd'">

each为迭代属性，稍后介绍。

## 拥有固定值的布尔属性

在XHTML/HTML5中有些属性是特殊的，它们要么是一个固定值，要么根本就不存在，比如：

<input type="checkbox" name="option1" checked="checked" />

<input type="checkbox" name="option2" />

在严格模式下，checked不能有其他的值，类似的还有disabled,muliple,readonly,selected等。

在标准方言中，允许你通过一个条件值来设置这些属性，如果值为真，这些属性将设置为它的固定值，如果为假，则不会设置此属性。如：

<input type="checkbox" name="active" th:checked="${user.active}" />

在标准方言中，类似的属性如下：

|  |  |  |
| --- | --- | --- |
| th:async | th:autofocus | th:autoplay |
| th:checked | th:controls | th:declare |
| th:default | th:defer | th:disabled |
| th:formnovalidate | th:hidden | th:ismap |
| th:loop | th:multiple | th:novalidate |
| th:nowrap | th:open | th:pubdate |
| th:readonly | th:required | th:reversed |
| th:scoped | th:seamless | th:selected |

## 设置自定义属性

除了刚刚看到的对于特定属性的处理外，Thymeleaf还在标准方言中提供了一个默认属性处理器，可以设置任意自定义属性的值，甚至可以没有具体的th:\*处理器。

比如：

<span th:whatever="${user.name}">...</span>

将返回

<span whatever="张三">...</span>

## HTML5自定义方式的支持

还可以使用一些完全不同的语法来应用到模板之中，它对html5更加友好。

<table>

<tr data-th-each="user : ${users}">

<td data-th-text="${user.login}">...</td>

<td data-th-text="${user.name}">...</td>

</tr>

</table>

例子中 data-{前缀}-{名字}这种语法是一个html5的自定义属性的标准方式。这种方式下，开发者无需导入命名空间，如th:,Thymeleaf将自动提供执行（所有方言模式，不仅是标准方言）。

还有一种语法来指定自定义标签：{前缀}-{名字}，这里遵循的是W3C标准的自定义标签。这也是可以使用，举个例子，比如th:block就可以使用th-block,这个将在稍后解释。

注意：这个语法是th:\*命名空间的一个补充，并不是他的替换，在为了跟本没有弃用命名空间语法的想法。

# 迭代(Iteration)

到目前为止，这个网络商店已经有了一个主页，有了一个用户配置的页面，还有了一个让用户订阅我们的通讯页，但是，我们的产品呢？我们是不是首先应该有一个产品列表，让客户知道我们在卖什么东东么？嗯，显然是的，让我们现在就走吧！

## 迭代基础

在模板/WEB-INF/templates/product/list.html中要显示产品列表，需要一个table。然后每个产品在显示上是table的一行，所以在我们的模板中，就需要Thymeleaf来迭代每一个产品。

在标准方言中，提供了一个迭代属性,为th:each

### 使用 th:each

对于产品列表页，首先需要更改产品列表的控制器，让其从服务层中获取产品数据，然添加到模板的context中。

public void process(HttpServletRequest request, HttpServletResponse response, ServletContext servletContext, ITemplateEngine templateEngine) throws Exception {

ProductService productService=new ProductService();

List<Product> allProducts=productService.findAll();

WebContext ctx=new WebContext(request,response,servletContext,request.getLocale());

ctx.setVariable("prods", allProducts);

templateEngine.process("product/list", ctx,response.getWriter());

}

然后，在模板中通过th:each来迭代产品：

<h1>产品列表</h1>

<table>

<tr>

<th>产品名称</th>

<th>产品价格</th>

<th>有现货</th>

</tr>

<tr th:each="prod:${prods}">

<td th:text="${prod.name}">土豆</td>

<td th:text="${prod.price}">2.41</td>

<td th:text="${prod.inStock}?#{true}:#{false}">yes</td>

</tr>

</table>

<p>

<a href="../home.html" th:href="@{/}">返回首页</a>

</p>

\*\*prod：\({prods}\*\*属性值的意思是，迭代\){prods}的每个元素并重复这个模板的这个片段。然后解释一下这两部分分别的意思：

* ${prods}被称为迭代表达式或迭代变量
* prod被称为重复变量或迭代值

注意：迭代值止只可以用在tr节点上面(包括迭代里边包含的td标签)。

### 可迭代的值

不是只有java.util.List对象可以使用Thymeleaf进行迭代。事实上，任何一个完整的对象集都可以使用th:each属性：

所有实现了java.util.Iterable接口的对象

所有实现了java.util.Map接口的对象（此时的迭代值是java.util.Map.Entry类）

所有数组

任何对象都视为一个只包含了它本身的单值的列表。

## 保持迭代状态

当使用th:each的时候，Thymeleaf会提供一个跟着迭代状态的机制：状态变量。

状态定义被封装在th:each的属性中。并包含以下数据：

获取当前迭代的从0开始的下标，使用index属性

获取当前迭代的从1开始的下标，使用count属性

获取当前迭代元素的总量，使用size属性

获取迭代变量中的迭代值，使用current属性

当前迭代值是奇数还是偶数，使用even/odd的布尔值属性

当前的迭代值是不是第一个元素，使用first布尔值属性

当前迭代值是不是最后一个元素，使用last布尔值属性。

现在修改一下前一个例子：

<h1>产品列表</h1>

<table>

<tr>

<th>产品名称</th>

<th>产品价格</th>

<th>有现货</th>

</tr>

<tr th:each="prod,iterStat:${prods}" th:class="${iterStat.odd}?'odd'">

<td th:text="${prod.name}">土豆</td>

<td th:text="${prod.price}">2.41</td>

<td th:text="${prod.inStock}?#{true}:#{false}">yes</td>

</tr>

</table>

<p>

<a href="../home.html" th:href="@{/}">返回首页</a>

</p>

可以看到，状态变量(即iterStat)的定义:将这个变量的名字作为属性写在迭代值之后，用逗号于迭代值隔开。产生了迭代值之后，他的状态值就可以也仅仅可以在th:each包含的代码段中使用。

这段代码的执行结果为：

可以看到，这端代码执行的非常完美。并且只在奇数行添加了odd的css(行数从0开始)

如果没有显示的设置一个状态变量，Thymeleaf会默认的创建一个使用迭代值+Stat后缀的迭代变量，如：

<table>

<tr>

<th>产品名称</th>

<th>产品价格</th>

<th>有现货</th>

</tr>

<tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">

<td th:text="${prod.name}">Onions</td>

<td th:text="${prod.price}">2.41</td>

<td th:text="${prod.inStock}? #{true} : #{false}">yes</td>

</tr>

</table>

## 通过懒检索进行优化

有时我们需要对只有检索操作的数据集的检索（如从数据库查询）进行优化。

事实上，他可以应用于任何的数据块，而检索内存中可能要重复使用的集合可能是最常见的情况。

为了能够支持这一点，Thymeleaf提供了懒加载上下文变量的机制，上下文的变量内容通过时实现ILazyContextVariable接口（最有可可能使用它默认实现：LazyContextVariable），它将在模板解析的时候来执行，如:

context.setVariable(

"users",

new LazyContextVariable<List<User>>() {

@Override

protected List<User> loadValue() {

return databaseRepository.findAllUsers();

}

});

这个变量就可以使用懒加载，，使用方式为：

<ul>

<li th:each="u : ${users}" th:text="${u.name}">user name</li>

</ul>

但如果为下边的代码，条件为false的时候也将不会被初始化(它的loadValue()方法将不会被调用)：

<ul th:if="${condition}">

<li th:each="u : ${users}" th:text="${u.name}">user name</li>

</ul>

# 条件判断(Conditional Evaluation)

## if和unless

有时你可能想要，模板的某个片段只有在条件被满足的时候才出现在结果中。

比如，假设我们要在产品表中显示一个列，该列针对每个产品的评论，如果有评论，则链接到评论页面。

要实现这个功能，就要用到前面说到过的th:if属性：

<table>

<tr>

<th>产品名称</th>

<th>产品价格</th>

<th>有现货</th>

<th>用户评价</th>

</tr>

<tr th:each="prod:${prods}">

<td th:text="${prod.name}">土豆</td>

<td th:text="${#numbers.formatDecimal(prod.price,0,2)}">2.41</td>

<td th:text="${prod.isStock}?#{true}:#{false}">yes</td>

<td>

<span th:text=${#lists.size(prod.comments)}>2</span>个评价

<a href="comments.html" th:href="@{/product/comments(prodId=${prod.id})}"

th:if="${not #lists.isEmpty(prod.comments)}">查看</a>

</td>

</tr>

</table>

重点注意下面这一行：

<a href="comments.html" th:href="@{/product/comments(prodId=${prod.id})}" th:if="${not #lists.isEmpty(prod.comments)}">查看</a>

事实上，这段代码没什么好解释的，如果产品有评论，那么我们就创建一个跳转到评论页面的超链接，并且使用产品ID作为参数。

查看生成结果：

Perfect！这正是我们想要的！

th:if不光可以使用布尔值，一下规则都可以：

如果值不为空：

如果值为布尔型并且为true

如果值为数值型并且不为0

如果值为character并且不为0

如果值为String，并且不为"false","off"和"no"

如果值不为布尔型，数值型，character或String的任意类型

如果值为null，th:if将为false

th:if还有一个互逆的表达式为th:unless,还继续用之前的例子作一个演示：

<a href="comments.html"

th:href="@{/comments(prodId=${prod.id})}"

th:unless="${#lists.isEmpty(prod.comments)}">查看</a>

## switch

条件选择还可以像java一样，使用switch来声明：在Thymeleaf中使用th:switch和th:case来实现。

他的工作方式会和你想的一样：

<div th:switch="${user.role}">

<p th:case="'admin'">超级管理员用户</p>

<p th:case="#{roles.manager}">管理员用户</p>

</div>

注意：一旦一个th:case被判断为真，那么其他的同等级的th:case都将被判断为假

default的写法为th:case="\*"

<div th:switch="${user.role}">

<p th:case="'admin'">超级管理员用户</p>

<p th:case="#{roles.manager}">管理员用户</p>

<p th:case="\*">其他用户</p>

</div>

# 模板的布局(Template Layout)

## 引入模板片段

### 定义和引用片段

我们经常会想让我们的模板包含一些其他模板，比较常见的用途如页眉，页脚，菜单等。

为了做到这一点，Thymeleaf需要我们定义一些可用片段，我们能通过 th:fragment 属性来实现这一点。

现在，我们要添加一个带标准版权声明的页脚文件( \*\*/WEB-INF/templates/footer.html\*\* )：

<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">

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

<body>

<div th:fragment="copy">

&copy; 网络商店

</div>

</body>

</html>

上面定义了一个叫copy的代码片段，我们能通过th:insert和th:replace属性（还有th:include,但Thymeleaf3.\*版本不推荐使用），很容易的将他导入到首页中：

<body>

...

<div th:insert="~{footer :: copy}"></div>

</body>

注意一点，th:insert使用的是一个片段表达式(~{...}),或片段中一个更具体的表达式。但在前者的情况下(简单片段表达式)，比如就像上边的代码，~{...}是可选的，所以如下班的代码是等价的：

<body>

...

<div th:insert="footer :: copy"></div>

</body>

### 片段语法说明

判断表达式语法非常简单，有三种不同的格式：

模板名::dom选择将导入模板名所指定的代码片段到dom选择器中。

注意:dom选择器可以仅仅是一个片段的名字，所以可以指定一下非常简单的名字，如：footer：copy或更简单的。

>dom选择器语法类似于XPath或css选择器，更多内容见附录C。

直接使用模板名将此模板对应的完整的代码导入。

>注意：此时由th:insert和th:replace标签导入的模板必须在当前的模板引擎下的模板解释器可以分辨。

使用::dom选择器或this:dom选择器导入与之相同的模板。

在上面的格式中，模板名和dom选择器都可以使用任何表达式的结果来表示,比如：

<div th:insert="footer :: (${user.isAdmin}? #{footer.admin} : #{footer.normaluser})"></div>

片段中可以包含任何th:\*属性。一点判断被包含到目标模板(即使用th:insert/th:replace的文件)中，这些属性将被执行。然后他们将能使用目标模板中的任何context变量。

>这种方法有个很大的优势：你的任何片段的代码，完整的代码均可以显示在浏览器中，同时，仍保留了使用Thymeleaf把他导入到其他模板中的能力。

### 引用不包含th:fragment的片段

此外，由于强大的dom选择器，使我们可以导入不含有th:fragment属性的代码片段，他甚至可以标记所有来自Thymeleaf所不知道的应用，如：

...

<div id="copy-section">

&copy; 2011 网络商店

</div>

...

这里可以就像css一样的使用它的id属性：

<body>

...

<div th:insert="footer :: #copy-section"></div>

</body>

### th:insert和th:replace的不同点(以及th:include)

好了，现在让我们看看这几个属性有什么区别(th:insert,th:replace和th:include(3.\*版本不推荐使用)):

th:insert是将th:fragment标签的内容纳入宿主标签

th:replace是使用th:fragment标签替换宿主标签

th:include与th:insert类似，但是他插入的是片段的内容，而不是片段

不够直观？举个例子：

<div th:fragment="copy">

&copy; 网络商店

</div>

<footer th:fragment="copy">

&copy; 2011 The Good Thymes Virtual Grocery

</footer>

导入到两个div标签中：

<body>

...

<div th:insert="footer :: copy"></div>

<div th:replace="footer :: copy"></div>

<div th:include="footer :: copy"></div>

</body>

执行结果：

<body>

...

<div>

<footer>

&copy; 网络商店

</footer>

</div>

<footer>

&copy; 网络商店

</footer>

<div>

&copy; 网络商店

</div>

</body>

## 片段的参数

为了像一个函数一样的使用一个模板片段，在片段定义的时候th:fragment可以定义一组参数：

<div th:fragment="frag (onevar,twovar)">

<p th:text="${onevar} + ' ' + ${twovar}">...</p>

</div>

th:insert和th:replace都用同一种语法来使用参数片段：

<div th:replace="::frag (${value1},${value2})">...</div>

<div th:replace="::frag (onevar=${value1},twovar=${value2})">...</div>

注意在第二种键值对的方式中，参数不关心顺序：

<div th:replace="::frag (twovar=${value2},onevar=${value1})">...</div>

###在没有参数签名的片段使用参数(Fragment local variables without fragment arguments)

即使一个片段没有定义参数，就像这样：

<div th:fragment="frag">

...

</div>

我们可以使用上面键值对的方式来赋予参数，并且也只能使用键值对的方法。

<div th:replace="::frag (onevar=${value1},twovar=${value2})">

事实上，在目标也使用th:replace和th:with的组合属性来接收：

<div th:replace="::frag" th:with="onevar=${value1},twovar=${value2}">

注意，在这里，无论参数是否有签名，都定义的是一个片段的局部变量，所以不会导致它覆盖或清空之前的context变量，片段仍然可以正常访问调用它的模板的每个context变量。

###模板断言(th:assert for in-template assertions)

th:assert属性可以定义一个用逗号分隔的表达式，用来为每一个条件做出评估，以判断是否产生异常。

<div th:assert="${onevar},(${twovar} != 43)">...</div>

这是一个在片段签名时就验证参数的方便方式：

<header th:fragment="contentheader(title)" th:assert="${!#strings.isEmpty(title)}">...</header>

##8.3更灵活的模板:超越单纯的插入(Flexible layouts: beyond mere fragment insertion)

基于片段表达式，我们可以为片段指定一个非文本，数字，javabean的参数，以代替标记。

这样我们就能使用这样一种方式，它可以使用丰富的标记来调用模板，以实现一个非常灵活的 \*\*模板布局机制\*\* (template layout mechanism)。

注意 \*\*title\*\* 和 \*\*links\*\* 变量在片段中的使用：

<head th:fragment="common\_header(title,links)">

<title th:replace="${title}">这个超帅应用</title>

<!-常用样式脚本 -->

<link rel="stylesheet" type="text/css" media="all" th:href="@{/css/myapp.css}">

<link rel="shortcut icon" th:href="@{/images/favicon.ico}">

<script type="text/javascript" th:src="@{/sh/scripts/codebase.js}"></script>

<!--/\* 每页占位符链接 \*/-->

<th:block th:replace="${links}" />

</head>

我们使用的片段

...

<head th:replace="base :: common\_header(~{::title},~{::link})">

<title>超帅应用 Main</title>

<link rel="stylesheet" th:href="@{/css/bootstrap.min.css}">

<link rel="stylesheet" th:href="@{/themes/smoothness/jquery-ui.css}">

</head>

...

结果将我们的调用的模板的实际的

...

<head>

<title>Awesome Main</title>

<!-Common styles and scripts -->

<link rel="stylesheet" type="text/css" media="all" href="/awe/css/awesomeapp.css">

<link rel="shortcut icon" href="/awe/images/favicon.ico">

<script type="text/javascript" src="/awe/sh/scripts/codebase.js"></script>

<link rel="stylesheet" href="/awe/css/bootstrap.min.css">

<link rel="stylesheet" href="/awe/themes/smoothness/jquery-ui.css">

</head>

...

###使用空白片段(Using the empty fragment)

A special fragment expression, the empty fragment (~{}), can be used for specifying no markup. Using the previous example:

<head th:replace="base :: common\_header(~{::title},~{})">

<title>Awesome Main</title>

</head>

...

Note how the second parameter of the fragment (links) is set to the empty fragment and therefore nothing is written for the <th:block th:replace="${links}" /> block:

...

<head>

<title>Awesome Main</title>

<!-Common styles and scripts -->

<link rel="stylesheet" type="text/css" media="all" href="/awe/css/awesomeapp.css">

<link rel="shortcut icon" href="/awe/images/favicon.ico">

<script type="text/javascript" src="/awe/sh/scripts/codebase.js"></script>

</head>

...

###使用无属性标记(Using the no-operation token)

The no-op can be also used as a parameter to a fragment if we just want to let our fragment use its current markup as a default value. Again, using the common\_header example:

...

<head th:replace="base :: common\_header(\_,~{::link})">

<title>Awesome Main</title>

<link rel="stylesheet" th:href="@{/css/bootstrap.min.css}">

<link rel="stylesheet" th:href="@{/themes/smoothness/jquery-ui.css}">

</head>

...

See how the title argument (first argument of the common\_header fragment) is set to no-op (\_), which results in this part of the fragment not being executed at all (title = no-operation):

<title th:replace="${title}">The awesome application</title>

So the result is:

...

<head>

<title>The awesome application</title>

<!-Common styles and scripts -->

<link rel="stylesheet" type="text/css" media="all" href="/awe/css/awesomeapp.css">

<link rel="shortcut icon" href="/awe/images/favicon.ico">

<script type="text/javascript" src="/awe/sh/scripts/codebase.js"></script>

<link rel="stylesheet" href="/awe/css/bootstrap.min.css">

<link rel="stylesheet" href="/awe/themes/smoothness/jquery-ui.css">

</head>

...

###Advanced conditional insertion of fragments

The availability of both the emtpy fragment and no-operation token allows us to perform conditional insertion of fragments in a very easy and elegant way.

For example, we could do this in order to insert our common :: adminhead fragment only if the user is an administrator, and insert nothing (emtpy fragment) if not:

...

<div th:insert="${user.isAdmin()} ? ~{common :: adminhead} : ~{}">...</div>

...

Also, we can use the no-operation token in order to insert a fragment only if the specified condition is met, but leave the markup without modifications if the condition is not met:

...

<div th:insert="${user.isAdmin()} ? ~{common :: adminhead} : \_">

Welcome [[${user.name}]], click <a th:href="@{/support}">here</a> for help-desk support.

</div>

...

Additionally, if we have configured our template resolvers to check for existence of the template resources –by means of their checkExistence flag -– we can use the existence of the fragment itself as the condition in a default operation:

...

<!-The body of the <div> will be used if the "common :: salutation" fragment -->

<!-does not exist (or is empty). -->

<div th:insert="~{common :: salutation} ?: \_">

Welcome [[${user.name}]], click <a th:href="@{/support}">here</a> for help-desk support.

</div>

...

##8.4 移除模板片段(Removing template fragments)

Back to the example application, let’s revisit the last version of our product list template:

<table>

<tr>

<th>NAME</th>

<th>PRICE</th>

<th>IN STOCK</th>

<th>COMMENTS</th>

</tr>

<tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">

<td th:text="${prod.name}">Onions</td>

<td th:text="${prod.price}">2.41</td>

<td th:text="${prod.inStock}? #{true} : #{false}">yes</td>

<td>

<span th:text="${#lists.size(prod.comments)}">2</span> comment/s

<a href="comments.html"

th:href="@{/product/comments(prodId=${prod.id})}"

th:unless="${#lists.isEmpty(prod.comments)}">view</a>

</td>

</tr>

</table>

This code is just fine as a template, but as a static page (when directly open by a browser without Thymeleaf processing it) it would not make a nice prototype.

Why? Because, although perfectly displayable by browsers, that table only has a row, and this row has mock data. As a prototype, it simply wouldn’t look realistic enough… we should have more than one product, we need more rows.

So let’s add some:

<table>

<tr>

<th>NAME</th>

<th>PRICE</th>

<th>IN STOCK</th>

<th>COMMENTS</th>

</tr>

<tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">

<td th:text="${prod.name}">Onions</td>

<td th:text="${prod.price}">2.41</td>

<td th:text="${prod.inStock}? #{true} : #{false}">yes</td>

<td>

<span th:text="${#lists.size(prod.comments)}">2</span> comment/s

<a href="comments.html"

th:href="@{/product/comments(prodId=${prod.id})}"

th:unless="${#lists.isEmpty(prod.comments)}">view</a>

</td>

</tr>

<tr class="odd">

<td>Blue Lettuce</td>

<td>9.55</td>

<td>no</td>

<td>

<span>0</span> comment/s

</td>

</tr>

<tr>

<td>Mild Cinnamon</td>

<td>1.99</td>

<td>yes</td>

<td>

<span>3</span> comment/s

<a href="comments.html">view</a>

</td>

</tr>

</table>

Ok, now we have three, definitely better for a prototype. But… what will happen when we process it with Thymeleaf?:

<table>

<tr>

<th>NAME</th>

<th>PRICE</th>

<th>IN STOCK</th>

<th>COMMENTS</th>

</tr>

<tr>

<td>Fresh Sweet Basil</td>

<td>4.99</td>

<td>yes</td>

<td>

<span>0</span> comment/s

</td>

</tr>

<tr class="odd">

<td>Italian Tomato</td>

<td>1.25</td>

<td>no</td>

<td>

<span>2</span> comment/s

<a href="/gtvg/product/comments?prodId=2">view</a>

</td>

</tr>

<tr>

<td>Yellow Bell Pepper</td>

<td>2.50</td>

<td>yes</td>

<td>

<span>0</span> comment/s

</td>

</tr>

<tr class="odd">

<td>Old Cheddar</td>

<td>18.75</td>

<td>yes</td>

<td>

<span>1</span> comment/s

<a href="/gtvg/product/comments?prodId=4">view</a>

</td>

</tr>

<tr class="odd">

<td>Blue Lettuce</td>

<td>9.55</td>

<td>no</td>

<td>

<span>0</span> comment/s

</td>

</tr>

<tr>

<td>Mild Cinnamon</td>

<td>1.99</td>

<td>yes</td>

<td>

<span>3</span> comment/s

<a href="comments.html">view</a>

</td>

</tr>

</table>

The last two rows are mock rows! Well, of course they are: iteration was only applied to the first row, so there is no reason why Thymeleaf should have removed the other two.

We need a way to remove those two rows during template processing. Let’s use the th:remove attribute on the second and third <tr> tags:

<table>

<tr>

<th>NAME</th>

<th>PRICE</th>

<th>IN STOCK</th>

<th>COMMENTS</th>

</tr>

<tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">

<td th:text="${prod.name}">Onions</td>

<td th:text="${prod.price}">2.41</td>

<td th:text="${prod.inStock}? #{true} : #{false}">yes</td>

<td>

<span th:text="${#lists.size(prod.comments)}">2</span> comment/s

<a href="comments.html"

th:href="@{/product/comments(prodId=${prod.id})}"

th:unless="${#lists.isEmpty(prod.comments)}">view</a>

</td>

</tr>

<tr class="odd" th:remove="all">

<td>Blue Lettuce</td>

<td>9.55</td>

<td>no</td>

<td>

<span>0</span> comment/s

</td>

</tr>

<tr th:remove="all">

<td>Mild Cinnamon</td>

<td>1.99</td>

<td>yes</td>

<td>

<span>3</span> comment/s

<a href="comments.html">view</a>

</td>

</tr>

</table>

Once processed, everything will look again as it should:

<table>

<tr>

<th>NAME</th>

<th>PRICE</th>

<th>IN STOCK</th>

<th>COMMENTS</th>

</tr>

<tr>

<td>Fresh Sweet Basil</td>

<td>4.99</td>

<td>yes</td>

<td>

<span>0</span> comment/s

</td>

</tr>

<tr class="odd">

<td>Italian Tomato</td>

<td>1.25</td>

<td>no</td>

<td>

<span>2</span> comment/s

<a href="/gtvg/product/comments?prodId=2">view</a>

</td>

</tr>

<tr>

<td>Yellow Bell Pepper</td>

<td>2.50</td>

<td>yes</td>

<td>

<span>0</span> comment/s

</td>

</tr>

<tr class="odd">

<td>Old Cheddar</td>

<td>18.75</td>

<td>yes</td>

<td>

<span>1</span> comment/s

<a href="/gtvg/product/comments?prodId=4">view</a>

</td>

</tr>

</table>

And what does that all value in the attribute, mean? th:remove can behave in five different ways, depending on its value:

all: Remove both the containing tag and all its children.

body: Do not remove the containing tag, but remove all its children.

tag: Remove the containing tag, but do not remove its children.

all-but-first: Remove all children of the containing tag except the first one.

none : Do nothing. This value is useful for dynamic evaluation.

What can that all-but-first value be useful for? It will let us save some th:remove="all" when prototyping:

<table>

<thead>

<tr>

<th>NAME</th>

<th>PRICE</th>

<th>IN STOCK</th>

<th>COMMENTS</th>

</tr>

</thead>

<tbody th:remove="all-but-first">

<tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">

<td th:text="${prod.name}">Onions</td>

<td th:text="${prod.price}">2.41</td>

<td th:text="${prod.inStock}? #{true} : #{false}">yes</td>

<td>

<span th:text="${#lists.size(prod.comments)}">2</span> comment/s

<a href="comments.html"

th:href="@{/product/comments(prodId=${prod.id})}"

th:unless="${#lists.isEmpty(prod.comments)}">view</a>

</td>

</tr>

<tr class="odd">

<td>Blue Lettuce</td>

<td>9.55</td>

<td>no</td>

<td>

<span>0</span> comment/s

</td>

</tr>

<tr>

<td>Mild Cinnamon</td>

<td>1.99</td>

<td>yes</td>

<td>

<span>3</span> comment/s

<a href="comments.html">view</a>

</td>

</tr>

</tbody>

</table>

The th:remove attribute can take any Thymeleaf Standard Expression, as long as it returns one of the allowed String values (all, tag, body, all-but-first or none).

This means removals could be conditional, like:

<a href="/something" th:remove="${condition}? tag : none">Link text not to be removed</a>

Also note that th:remove considers null a synonym to none, so the following works the same as the example above:

<a href="/something" th:remove="${condition}? tag">Link text not to be removed</a>

In this case, if ${condition} is false, null will be returned, and thus no removal will be performed.

# 本地变量(Local Variables)

Thymeleaf calls local variables the variables that are defined for a specific fragment of a template, and are only available for evaluation inside that fragment.

An example we have already seen is the prod iter variable in our product list page:

<tr th:each="prod : ${prods}">

...

</tr>

That prod variable will be available only within the bounds of the <tr> tag. Specifically:

It will be available for any other th:\* attributes executing in that tag with less precedence than th:each (which means they will execute after th:each).

It will be available for any child element of the <tr> tag, such as any <td> elements.

Thymeleaf offers you a way to declare local variables without iteration, using the th:with attribute, and its syntax is like that of attribute value assignments:

<div th:with="firstPer=${persons[0]}">

<p>

The name of the first person is <span th:text="${firstPer.name}">Julius Caesar</span>.

</p>

</div>

When th:with is processed, that firstPer variable is created as a local variable and added to the variables map coming from the context, so that it is available for evaluation along with any other variables declared in the context, but only within the bounds of the containing <div> tag.

You can define several variables at the same time using the usual multiple assignment syntax:

<div th:with="firstPer=${persons[0]},secondPer=${persons[1]}">

<p>

The name of the first person is <span th:text="${firstPer.name}">Julius Caesar</span>.

</p>

<p>

But the name of the second person is

<span th:text="${secondPer.name}">Marcus Antonius</span>.

</p>

</div>

The th:with attribute allows reusing variables defined in the same attribute:

<div th:with="company=${user.company + ' Co.'},account=${accounts[company]}">...</div>

Let’s use this in our Grocery’s home page! Remember the code we wrote for outputting a formatted date?

<p>

Today is:

<span th:text="${#calendars.format(today,'dd MMMM yyyy')}">13 february 2011</span>

</p>

Well, what if we wanted that "dd MMMM yyyy" to actually depend on the locale? For example, we might want to add the following message to our home\_en.properties:

date.format=MMMM dd'','' yyyy

…and an equivalent one to our home\_es.properties:

date.format=dd ''de'' MMMM'','' yyyy

Now, let’s use th:with to get the localized date format into a variable, and then use it in our th:text expression:

<p th:with="df=#{date.format}">

Today is: <span th:text="${#calendars.format(today,df)}">13 February 2011</span>

</p>

That was clean and easy. In fact, given the fact that th:with has a higher precedence than th:text, we could have solved this all in the span tag:

<p>

Today is:

<span th:with="df=#{date.format}"

th:text="${#calendars.format(today,df)}">13 February 2011</span>

</p>

You might be thinking: Precedence? We haven’t talked about that yet! Well, don’t worry because that is exactly what the next chapter is about.

# Attribute Precedence

What happens when you write more than one th:\* attribute in the same tag? For example:

<ul>

<li th:each="item : ${items}" th:text="${item.description}">Item description here...</li>

</ul>

We would expect that th:each attribute to execute before the th:text so that we get the results we want, but given the fact that the HTML/XML standards do not give any kind of meaning to the order in which the attributes in a tag are written, a precedence mechanism had to be established in the attributes themselves in order to be sure that this will work as expected.

So, all Thymeleaf attributes define a numeric precedence, which establishes the order in which they are executed in the tag. This order is:

|  |  |  |
| --- | --- | --- |
| Order | Feature | Attributes |
| 1 | Fragment inclusion | th:insert  th:replace |
| 2 | Fragment iteration | th:each |
| 3 | Conditional evaluation | th:if  th:unless  th:switch  th:case |
| 4 | Local variable definition | th:object  th:with |
| 5 | General attribute modification | th:attr  th:attrprepend  th:attrappend |
| 6 | Specific attribute modification | th:value  th:href  th:src  ... |
| 7 | Text (tag body modification) | th:text  th:utext |
| 8 | Fragment specification | th:fragment |
| 9 | Fragment removal | th:remove |

This precedence mechanism means that the above iteration fragment will give exactly the same results if the attribute position is inverted (although it would be slightly less readable):

<ul>

<li th:text="${item.description}" th:each="item : ${items}">Item description here...</li>

</ul>

# Comments and Blocks

## 标准 HTML/XML 注释

Standard HTML/XML comments <!-... --> can be used anywhere in Thymeleaf templates. Anything inside these comments won’t be processed by Thymeleaf, and will be copied verbatim to the result:

<!-User info follows -->

<div th:text="${...}">

...

</div>

## Thymeleaf parser-level comment blocks

Parser-level comment blocks are code that will be simply removed from the template when Thymeleaf parses it. They look like this:

<!--/\* This code will be removed at Thymeleaf parsing time! \*/-->

Thymeleaf will remove everything between <!--/\* and \*/-->, so these comment blocks can also be used for displaying code when a template is statically open, knowing that it will be removed when Thymeleaf processes it:

<!--/\*-->

<div>

you can see me only before Thymeleaf processes me!

</div>

<!--\*/-->

This might come very handy for prototyping tables with a lot of <tr>’s, for example:

<table>

<tr th:each="x : ${xs}">

...

</tr>

<!--/\*-->

<tr>

...

</tr>

<tr>

...

</tr>

<!--\*/-->

</table>

## Thymeleaf prototype-only comment blocks

Thymeleaf allows the definition of special comment blocks marked to be comments when the template is open statically (i.e. as a prototype), but considered normal markup by Thymeleaf when executing the template.

<span>hello!</span>

<!--/\*/

<div th:text="${...}">

...

</div>

/\*/-->

<span>goodbye!</span>

Thymeleaf’s parsing system will simply remove the <!--/\*/ and /\*/--> markers, but not its contents, which will be left therefore uncommented. So when executing the template, Thymeleaf will actually see this:

<span>hello!</span>

<div th:text="${...}">

...

</div>

<span>goodbye!</span>

As with parser-level comment blocks, this feature is dialect-independent.

## Synthetic th:block tag

Thymeleaf’s only element processor (not an attribute) included in the Standard Dialects is th:block.

th:block is a mere attribute container that allows template developers to specify whichever attributes they want. Thymeleaf will execute these attributes and then simply make the block, but not its contents, disappear.

So it could be useful, for example, when creating iterated tables that require more than one <tr> for each element:

<table>

<th:block th:each="user : ${users}">

<tr>

<td th:text="${user.login}">...</td>

<td th:text="${user.name}">...</td>

</tr>

<tr>

<td colspan="2" th:text="${user.address}">...</td>

</tr>

</th:block>

</table>

And especially useful when used in combination with prototype-only comment blocks:

<table>

<!--/\*/ <th:block th:each="user : ${users}"> /\*/-->

<tr>

<td th:text="${user.login}">...</td>

<td th:text="${user.name}">...</td>

</tr>

<tr>

<td colspan="2" th:text="${user.address}">...</td>

</tr>

<!--/\*/ </th:block> /\*/-->

</table>

Note how this solution allows templates to be valid HTML (no need to add forbidden <div> blocks inside <table>), and still works OK when open statically in browsers as prototypes!

# Inlining

## Expression inlining

Although the Standard Dialect allows us to do almost everything using tag attributes, there are situations in which we could prefer writing expressions directly into our HTML texts. For example, we could prefer writing this:

<p>Hello, [[${session.user.name}]]!</p>

…instead of this:

<p>Hello, <span th:text="${session.user.name}">Sebastian</span>!</p>

Expressions between [[...]] or [(...)] are considered inlined expressions in Thymeleaf, and inside them we can use any kind of expression that would also be valid in a th:text or th:utext attribute.

Note that, while [[...]] corresponds to th:text (i.e. result will be HTML-escaped), [(...)] corresponds to th:utext and will not perform any HTML-escaping. So with a variable such as msg = 'This is <b>great!</b>', given this fragment:

<p>The message is "[(${msg})]"</p>

The result will have those <b> tags unescaped, so:

<p>The message is "This is <b>great!</b>"</p>

Whereas if escaped like:

<p>The message is "[[${msg}]]"</p>

The result will be HTML-escaped:

<p>The message is "This is &lt;b&gt;great!&lt;/b&gt;"</p>

Note that text inlining is active by default in the body of every tag in our markup –not the tags themselves -–, so there is nothing we need to do to to enable it.

Inlining vs natural templates

If you come from other template engines in which this way of outputting text is the norm, you might be asking: Why aren’t we doing this from the beginning? It’s less code than all those th:text attributes!

Well, be careful there, because although you might find inlining quite interesting, you should always remember that inlined expressions will be displayed verbatim in your HTML files when you open them statically, so you probably won’t be able to use them as design prototypes anymore!

The difference between how a browser would statically display our fragment of code without using inlining…

Hello, Sebastian!

…and using it…

Hello, [[${session.user.name}]]!

…is quite clear in terms of design usefulness.

Disabling inlining

This mechanism can be disabled though, because there might actually be occasions in which we do want to output the [[...]] or [(...)] sequences without its contents being processed as an expression. For that, we will use th:inline="none":

<p th:inline="none">A double array looks like this: [[1, 2, 3], [4, 5]]!</p>

This will result in:

<p>A double array looks like this: [[1, 2, 3], [4, 5]]!</p>

## Text inlining

Text inlining is very similar to the expression inlining capability we have just seen, but it actually adds more power. It has to be enabled explicitly with th:inline="text".

Text inlining not only allows us to use the same inlined expressions we just saw, but in fact processes tag bodies as if they were templates processed in the TEXT template mode, which allows us to perform text-based template logic (not only output expressions).

We will see more about this in the next chapter about the textual template modes.

## JavaScript inlining

JavaScript inlining allows for a better integration of JavaScript <script> blocks in templates being processed in the HTML template mode.

As with text inlining, this is actually equivalent to processing the scripts contents as if they were templates in the JAVASCRIPT template mode, and therefore all the power of the textual template modes (see next chapter) will be at hand. However, in this section we will focus on how we can use it for adding the output of our Thymeleaf expressions into our JavaScript blocks.

This mode has to be explicitly enabled using th:inline="javascript":

<script th:inline="javascript">

...

var username = [[${session.user.name}]];

...

</script>

This will result in:

<script th:inline="javascript">

...

var username = "Sebastian \"Fruity\" Applejuice";

...

</script>

Two important things to note in the code above:

First, that JavaScript inlining will not only output the required text, but also enclose it with quotes and JavaScript-escape its contents, so that the expression results are output as a well-formed JavaScript literal.

Second, that this is happening because we are outputting the ${session.user.name} expression as escaped, i.e. using a double-bracket expression: [[${session.user.name}]]. If instead we used unescaped like:

<script th:inline="javascript">

...

var username = [(${session.user.name})];

...

</script>

The result would look like:

<script th:inline="javascript">

...

var username = Sebastian "Fruity" Applejuice;

...

</script>

…which is malformed JavaScript code. But outputting something unescaped might be what we need if we are building parts of our script by means of appending inlined expressions, so it’s good to have this tool at hand.

JavaScript natural templates

The mentioned intelligence of the JavaScript inlining mechanism goes much further than just applying JavaScript-specific escaping and outputting expression results as valid literals.

For example, we can wrap our (escaped) inlined expressions in JavaScript comments like:

<script th:inline="javascript">

...

var username = /\*[[${session.user.name}]]\*/ "Gertrud Kiwifruit";

...

</script>

And Thymeleaf will ignore everything we have written after the comment and before the semicolon (in this case 'Gertrud Kiwifruit'), so the result of executing this will look exactly like when we were not using the wrapping comments:

<script th:inline="javascript">

...

var username = "Sebastian \"Fruity\" Applejuice";

...

</script>

But have another careful look at the original template code:

<script th:inline="javascript">

...

var username = /\*[[${session.user.name}]]\*/ "Gertrud Kiwifruit";

...

</script>

Note how this is valid JavaScript code. And it will perfectly execute when you open your template file in a static manner (without executing it at a server).

So what we have here is a way to do JavaScript natural templates!

Advanced inlined evaluation and JavaScript serialization

An important thing to note regarding JavaScript inlining is that this expression evaluation is intelligent and not limited to Strings. Thymeleaf will correctly write in JavaScript syntax the following kinds of objects:

Strings

Numbers

Booleans

Arrays

Collections

Maps

Beans (objects with getter and setter methods)

For example, if we had the following code:

<script th:inline="javascript">

...

var user = /\*[[${session.user}]]\*/ null;

...

</script>

That ${session.user} expression will evaluate to a User object, and Thymeleaf will correctly convert it to Javascript syntax:

<script th:inline="javascript">

...

var user = {"age":null,"firstName":"John","lastName":"Apricot",

"name":"John Apricot","nationality":"Antarctica"};

...

</script>

The way this JavaScript serialization is done is by means of an implementation of the org.thymeleaf.standard.serializer.IStandardJavaScriptSerializer interface, which can be configured at the instance of the StandardDialect being used at the template engine.

The default implementation of this JS serialization mechanism will look for the Jackson library in the classpath and, if present, will use it. If not, it will apply a built-in serialization mechanism that covers the needs of most scenarios and produces similar results (but is less flexible).

## CSS inlining

Thymeleaf also allows the use of inlining in CSS <style> tags, such as:

<style th:inline="css">

...

</style>

For example, say we have two variables set to two different String values:

classname = 'main elems'

align = 'center'

We could use them just like:

<style th:inline="css">

.[[${classname}]] {

text-align: [[${align}]];

}

</style>

And the result would be:

<style th:inline="css">

.main\ elems {

text-align: center;

}

</style>

Note how CSS inlining also bears some intelligence, just like JavaScript’s. Specifically, expressions output via escaped expressions like [[${classname}]] will be escaped as CSS identifiers. That is why our classname = 'main elems' has turned into main\ elems in the fragment of code above.

Advanced features: CSS natural templates, etc.

In an equivalent way to what was explained before for JavaScript, CSS inlining also allows for our <style> tags to work both statically and dynamically, i.e. as CSS natural templates by means of wrapping inlined expressions in comments. See:

<style th:inline="css">

.main\ elems {

text-align: /\*[[${align}]]\*/ left;

}

</style>

# Textual template modes

## Textual syntax

Three of the Thymeleaf template modes are considered textual: TEXT, JAVASCRIPT and CSS. This differentiates them from the markup template modes: HTML and XML.

The key difference between textual template modes and the markup ones is that in a textual template there are no tags into which to insert logic in the form of attributes, so we have to rely on other mechanisms.

The first and most basic of these mechanisms is inlining, which we have already detailed in the previous chapter. Inlining syntax is the most simple way to output results of expressions in textual template mode, so this is a perfectly valid template for a text email.

Dear [(${name})],

Please find attached the results of the report you requested

with name "[(${report.name})]".

Sincerely,

The Reporter.

Even without tags, the example above is a complete and valid Thymeleaf template that can be executed in the TEXT template mode.

But in order to include more complex logic than mere output expressions, we need a new non-tag-based syntax:

[# th:each="item : ${items}"]

[(${item})]

[/]

Which is actually the condensed version of the more verbose:

[#th:block th:each="item : ${items}"]

[#th:block th:utext="${item}" /]

[/th:block]

Note how this new syntax is based on elements (i.e. processable tags) that are declared as [#element ...] instead of <element ...>. Elements are open like [#element ...] and closed like [/element], and standalone tags can be declared by minimizing the open element with a / in a way almost equivalent to XML tags: [#element ... /].

The Standard Dialect only contains a processor for one of these elements: the already-known th:block, though we could extend this in our dialects and create new elements in the usual way. Also, the th:block element ([#th:block ...] ... [/th:block]) is allowed to be abbreviated as the empty string ([# ...] ... [/]), so the above block is actually equivalent to:

[# th:each="item : ${items}"]

[# th:utext="${item}" /]

[/]

And given [# th:utext="${item}" /] is equivalent to an inlined unescaped expression, we could just use it in order to have less code. Thus we end up with the first fragment of code we saw above:

[# th:each="item : ${items}"]

[(${item})]

[/]

Note that the textual syntax requires full element balance (no unclosed tags) and quoted attributes – it’s more XML-style than HTML-style.

Let’s have a look at a more complete example of a TEXT template, a plain text email template:

Dear [(${customer.name})],

This is the list of our products:

[# th:each="prod : ${products}"]

[(${prod.name})]. Price: [(${prod.price})] EUR/kg

[/]

Thanks,

The Thymeleaf Shop

After executing, the result of this could be something like:

Dear Mary Ann Blueberry,

This is the list of our products:

Apricots. Price: 1.12 EUR/kg

Bananas. Price: 1.78 EUR/kg

Apples. Price: 0.85 EUR/kg

Watermelon. Price: 1.91 EUR/kg

Thanks,

The Thymeleaf Shop

And another example in JAVASCRIPT template mode, a greeter.js file, we process as a textual template and which result we call from our HTML pages. Note this is not a <script> block in an HTML template, but a .js file being processed as a template on its own:

var greeter = function() {

var username = [[${session.user.name}]];

[# th:each="salut : ${salutations}"]

alert([[${salut}]] + " " + username);

[/]

};

After executing, the result of this could be something like:

var greeter = function() {

var username = "Bertrand \"Crunchy\" Pear";

alert("Hello" + " " + username);

alert("Ol\u00E1" + " " + username);

alert("Hola" + " " + username);

};

Escaped element attributes

In order to avoid interactions with parts of the template that might be processed in other modes (e.g. text-mode inlining inside an HTML template), Thymeleaf 3.0 allows the attributes in elements in its textual syntax to be escaped. So:

Attributes in TEXT template mode will be HTML-unescaped.

Attributes in JAVASCRIPT template mode will be JavaScript-unescaped.

Attributes in CSS template mode will be CSS-unescaped.

So this would be perfectly OK in a TEXT-mode template (note the &gt;):

[# th:if="${120&lt;user.age}"]

Congratulations!

[/]

Of course that &lt; would make no sense in a real text template, but it is a good idea if we are processing an HTML template with a th:inline="text" block containing the code above and we want to make sure our browser doesn’t take that <user.age for the name of an open tag when statically opening the file as a prototype.

13.2 Extensibility

One of the advantages of this syntax is that it is just as extensible as the markup one. Developers can still define their own dialects with custom elements and attributes, apply a prefix to them (optionally), and then use them in textual template modes:

[#myorg:dosomething myorg:importantattr="211"]some text[/myorg:dosomething]

13.3 Textual prototype-only comment blocks: adding code

The JAVASCRIPT and CSS template modes (not available for TEXT) allow including code between a special comment syntax /\*[+...+]\*/ so that Thymeleaf will automatically uncomment such code when processing the template:

var x = 23;

/\*[+

var msg = "This is a working application";

+]\*/

var f = function() {

...

Will be executed as:

var x = 23;

var msg = "This is a working application";

var f = function() {

...

You can include expressions inside these comments, and they will be evaluated:

var x = 23;

/\*[+

var msg = "Hello, " + [[${session.user.name}]];

+]\*/

var f = function() {

...

13.4 Textual parser-level comment blocks: removing code

In a way similar to that of prototype-only comment blocks, all the three textual template modes (TEXT, JAVASCRIPT and CSS) make it possible to instruct Thymeleaf to remove code between special /\*[\*/ and /\* -]\*/ marks, like this:

var x = 23;

/\*[\*/

var msg = "This is shown only when executed statically!";

/\* -]\*/

var f = function() {

...

Or this, in TEXT mode:

...

/\*[Note the user is obtained from the session, which must exist -]\*/

Welcome [(${session.user.name})]!

...

13.5 Natural JavaScript and CSS templates

As seen in the previous chapter, JavaScript and CSS inlining offer the possibility to include inlined expressions inside JavaScript/CSS comments, like:

...

var username = /\*[[${session.user.name}]]\*/ "Sebastian Lychee";

...

…which is valid JavaScript, and once executed could look like:

...

var username = "John Apricot";

...

This same trick of enclosing inlined expressions inside comments can in fact be used for the entire textual mode syntax:

/\*[# th:if="${user.admin}"]\*/

alert('Welcome admin');

/\*[/]\*/

That alert in the code above will be shown when the template is open statically – because it is 100% valid JavaScript –, and also when the template is run if the user is an admin. It is equivalent to:

[# th:if="${user.admin}"]

alert('Welcome admin');

[/]

…which is actually the code to which the initial version is converted during template parsing.

Note however that wrapping elements in comments does not clean the lines they live in (to the right until a ; is found) as inlined output expressions do. That behaviour is reserved for inlined output expressions only.

So Thymeleaf 3.0 allows the development of complex JavaScript scripts and CSS style sheets in the form of natural templates, valid both as a prototype and as a working template.

# Some more pages for our grocery

Now we know a lot about using Thymeleaf, we can add some new pages to our website for order management.

Note that we will focus on HTML code, but you can have a look at the bundled source code if you want to see the corresponding controllers.

## Order List

Let’s start by creating an order list page, /WEB-INF/templates/order/list.html:

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<title>Good Thymes Virtual Grocery</title>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />

<link rel="stylesheet" type="text/css" media="all"

href="../../../css/gtvg.css" th:href="@{/css/gtvg.css}" />

</head>

<body>

<h1>Order list</h1>

<table>

<tr>

<th>DATE</th>

<th>CUSTOMER</th>

<th>TOTAL</th>

<th></th>

</tr>

<tr th:each="o : ${orders}" th:class="${oStat.odd}? 'odd'">

<td th:text="${#calendars.format(o.date,'dd/MMM/yyyy')}">13 jan 2011</td>

<td th:text="${o.customer.name}">Frederic Tomato</td>

<td th:text="${#aggregates.sum(o.orderLines.{purchasePrice \* amount})}">23.32</td>

<td>

<a href="details.html" th:href="@{/order/details(orderId=${o.id})}">view</a>

</td>

</tr>

</table>

<p>

<a href="../home.html" th:href="@{/}">Return to home</a>

</p>

</body>

</html>

There’s nothing here that should surprise us, except for this little bit of OGNL magic:

<td th:text="${#aggregates.sum(o.orderLines.{purchasePrice \* amount})}">23.32</td>

What that does is, for each order line (OrderLine object) in the order, multiply its purchasePrice and amount properties (by calling the corresponding getPurchasePrice() and getAmount() methods) and return the result into a list of numbers, later aggregated by the #aggregates.sum(...) function in order to obtain the order total price.

You’ve got to love the power of OGNL.

## Order Details

Now for the order details page, in which we will make a heavy use of asterisk syntax:

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<title>Good Thymes Virtual Grocery</title>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />

<link rel="stylesheet" type="text/css" media="all"

href="../../../css/gtvg.css" th:href="@{/css/gtvg.css}" />

</head>

<body th:object="${order}">

<h1>Order details</h1>

<div>

<p><b>Code:</b> <span th:text="\*{id}">99</span></p>

<p>

<b>Date:</b>

<span th:text="\*{#calendars.format(date,'dd MMM yyyy')}">13 jan 2011</span>

</p>

</div>

<h2>Customer</h2>

<div th:object="\*{customer}">

<p><b>Name:</b> <span th:text="\*{name}">Frederic Tomato</span></p>

<p>

<b>Since:</b>

<span th:text="\*{#calendars.format(customerSince,'dd MMM yyyy')}">1 jan 2011</span>

</p>

</div>

<h2>Products</h2>

<table>

<tr>

<th>PRODUCT</th>

<th>AMOUNT</th>

<th>PURCHASE PRICE</th>

</tr>

<tr th:each="ol,row : \*{orderLines}" th:class="${row.odd}? 'odd'">

<td th:text="${ol.product.name}">Strawberries</td>

<td th:text="${ol.amount}" class="number">3</td>

<td th:text="${ol.purchasePrice}" class="number">23.32</td>

</tr>

</table>

<div>

<b>TOTAL:</b>

<span th:text="\*{#aggregates.sum(orderLines.{purchasePrice \* amount})}">35.23</span>

</div>

<p>

<a href="list.html" th:href="@{/order/list}">Return to order list</a>

</p>

</body>

</html>

Not much really new here, except for this nested object selection:

<body th:object="${order}">

...

<div th:object="\*{customer}">

<p><b>Name:</b> <span th:text="\*{name}">Frederic Tomato</span></p>

...

</div>

...

</body>

…which makes that \*{name} equivalent to:

<p><b>Name:</b> <span th:text="${order.customer.name}">Frederic Tomato</span></p>

# 15 More on Configuration

## Template Resolvers

For our Good Thymes Virtual Grocery, we chose an ITemplateResolver implementation called ServletContextTemplateResolver that allowed us to obtain templates as resources from the Servlet Context.

Besides giving us the ability to create our own template resolver by implementing ITemplateResolver, Thymeleaf includes four implementations out of the box:

org.thymeleaf.templateresolver.ClassLoaderTemplateResolver, which resolves templates as classloader resources, like:

return Thread.currentThread().getContextClassLoader().getResourceAsStream(template);

org.thymeleaf.templateresolver.FileTemplateResolver, which resolves templates as files from the file system, like:

return new FileInputStream(new File(template));

org.thymeleaf.templateresolver.UrlTemplateResolver, which resolves templates as URLs (even non-local ones), like:

return (new URL(template)).openStream();

org.thymeleaf.templateresolver.StringTemplateResolver, which resolves templates directly as the String being specified as template (or template name, which in this case is obviously much more than a mere name):

return new StringReader(templateName);

All of the pre-bundled implementations of ITemplateResolver allow the same set of configuration parameters, which include:

Prefix and suffix (as already seen):

templateResolver.setPrefix("/WEB-INF/templates/");

templateResolver.setSuffix(".html");

Template aliases that allow the use of template names that do not directly correspond to file names. If both suffix/prefix and alias exist, alias will be applied before prefix/suffix:

templateResolver.addTemplateAlias("adminHome","profiles/admin/home");

templateResolver.setTemplateAliases(aliasesMap);

Encoding to be applied when reading templates:

templateResolver.setEncoding("UTF-8");

Template mode to be used:

// Default is HTML

templateResolver.setTemplateMode("XML");

Default mode for template cache, and patterns for defining whether specific templates are cacheable or not:

// Default is true

templateResolver.setCacheable(false);

templateResolver.getCacheablePatternSpec().addPattern("/users/\*");

TTL in milliseconds for parsed template cache entries originated in this template resolver. If not set, the only way to remove an entry from the cache will be LRU (cache max size exceeded and the entry is the oldest).

// Default is no TTL (only LRU would remove entries)

templateResolver.setCacheTTLMs(60000L);

The Thymeleaf + Spring integration packages offer a SpringResourceTemplateResolver implementation which uses all the Spring infrastructure for accessing and reading resources in applications, and which is the recommended implementation in Spring-enabled applications.

Chaining Template Resolvers

Also, a Template Engine can specify several template resolvers, in which case an order can be established between them for template resolution so that, if the first one is not able to resolve the template, the second one is asked, and so on:

ClassLoaderTemplateResolver classLoaderTemplateResolver = new ClassLoaderTemplateResolver();

classLoaderTemplateResolver.setOrder(Integer.valueOf(1));

ServletContextTemplateResolver servletContextTemplateResolver =

new ServletContextTemplateResolver(servletContext);

servletContextTemplateResolver.setOrder(Integer.valueOf(2));

templateEngine.addTemplateResolver(classLoaderTemplateResolver);

templateEngine.addTemplateResolver(servletContextTemplateResolver);

When several template resolvers are applied, it is recommended to specify patterns for each template resolver so that Thymeleaf can quickly discard those template resolvers that are not meant to resolve the template, enhancing performance. Doing this is not a requirement, but a recommendation:

ClassLoaderTemplateResolver classLoaderTemplateResolver = new ClassLoaderTemplateResolver();

classLoaderTemplateResolver.setOrder(Integer.valueOf(1));

// This classloader will not be even asked for any templates not matching these patterns

classLoaderTemplateResolver.getResolvablePatternSpec().addPattern("/layout/\*.html");

classLoaderTemplateResolver.getResolvablePatternSpec().addPattern("/menu/\*.html");

ServletContextTemplateResolver servletContextTemplateResolver =

new ServletContextTemplateResolver(servletContext);

servletContextTemplateResolver.setOrder(Integer.valueOf(2));

If these resolvable patterns are not specified, we will be relying on the specific capabilities of each of the ITemplateResolver implementations we are using. Note that not all implementations might be able to determine the existence of a template before resolving, and thus could always consider a template as resolvable and break the resolution chain (not allowing other resolvers to check for the same template), but then be unable to read the real resource.

All the ITemplateResolver implementations that are included with core Thymeleaf include a mechanism that will allow us to make the resolvers really check if a resource exists before considering it resolvable. It is the checkExistence flag, which works like:

ClassLoaderTemplateResolver classLoaderTemplateResolver = new ClassLoaderTemplateResolver();

classLoaderTemplateResolver.setOrder(Integer.valueOf(1));

classLoaderTempalteResolver.setCheckExistence(true);

This checkExistence flag forces the resolver perform a real check for resource existence during the resolution phase (and let the following resolver in the chain be called if existence check returns false). While this might sound good in every case, in most cases this will mean a double access to the resource itself (once for checking existence, another time for reading it), and could be a performance issue in some scenarios, e.g. remote URL-based template resources – a potential performance issue that might anyway get largely mitigated by the use of the template cache (in which case templates will only be resolved the first time they are accessed).

15.2 Message Resolvers

We did not explicitly specify a Message Resolver implementation for our Grocery application, and as it was explained before, this meant that the implementation being used was an org.thymeleaf.messageresolver.StandardMessageResolver object.

StandardMessageResolver is the standard implementation of the IMessageResolver interface, but we could create our own if we wanted, adapted to the specific needs of our application.

The Thymeleaf + Spring integration packages offer by default an IMessageResolver implementation which uses the standard Spring way of retrieving externalized messages, by using MessageSource beans declared at the Spring Application Context.

Standard Message Resolver

So how does StandardMessageResolver look for the messages requested at a specific template?

If the template name is home and it is located in /WEB-INF/templates/home.html, and the requested locale is gl\_ES then this resolver will look for messages in the following files, in this order:

/WEB-INF/templates/home\_gl\_ES.properties

/WEB-INF/templates/home\_gl.properties

/WEB-INF/templates/home.properties

Refer to the JavaDoc documentation of the StandardMessageResolver class for more detail on how the complete message resolution mechanism works.

Configuring message resolvers

What if we wanted to add a message resolver (or more) to the Template Engine? Easy:

// For setting only one

templateEngine.setMessageResolver(messageResolver);

// For setting more than one

templateEngine.addMessageResolver(messageResolver);

And why would we want to have more than one message resolver? For the same reason as template resolvers: message resolvers are ordered and if the first one cannot resolve a specific message, the second one will be asked, then the third, etc.

15.3 Conversion Services

The conversion service that enables us to perform data conversion and formatting operations by means of the double-brace syntax (${{...}}) is actually a feature of the Standard Dialect, not of the Thymeleaf Template Engine itself.

As such, the way to configure it is by setting our custom implementation of the IStandardConversionService interface directly into the instance of StandardDialect that is being configured into the template engine. Like:

IStandardConversionService customConversionService = ...

StandardDialect dialect = new StandardDialect();

dialect.setConversionService(customConversionService);

templateEngine.setDialect(dialect);

Note that the thymeleaf-spring3 and thymeleaf-spring4 packages contain the SpringStandardDialect, and this dialect already comes pre-configured with an implementation of IStandardConversionService that integrates Spring’s own Conversion Service infrastructure into Thymeleaf.

15.4 Logging

Thymeleaf pays quite a lot of attention to logging, and always tries to offer the maximum amount of useful information through its logging interface.

The logging library used is slf4j, which in fact acts as a bridge to whichever logging implementation we might want to use in our application (for example, log4j).

Thymeleaf classes will log TRACE, DEBUG and INFO-level information, depending on the level of detail we desire, and besides general logging it will use three special loggers associated with the TemplateEngine class which we can configure separately for different purposes:

org.thymeleaf.TemplateEngine.CONFIG will output detailed configuration of the library during initialization.

org.thymeleaf.TemplateEngine.TIMER will output information about the amount of time taken to process each template (useful for benchmarking!)

org.thymeleaf.TemplateEngine.cache is the prefix for a set of loggers that output specific information about the caches. Although the names of the cache loggers are configurable by the user and thus could change, by default they are:

org.thymeleaf.TemplateEngine.cache.TEMPLATE\_CACHE

org.thymeleaf.TemplateEngine.cache.EXPRESSION\_CACHE

An example configuration for Thymeleaf’s logging infrastructure, using log4j, could be:

log4j.logger.org.thymeleaf=DEBUG

log4j.logger.org.thymeleaf.TemplateEngine.CONFIG=TRACE

log4j.logger.org.thymeleaf.TemplateEngine.TIMER=TRACE

log4j.logger.org.thymeleaf.TemplateEngine.cache.TEMPLATE\_CACHE=TRACE

# Template Cache

Thymeleaf works thanks to a set of parsers – for markup and text – that parse templates into sequences of events (open tag, text, close tag, comment, etc.) and a series of processors – one for each type of behaviour that needs to be applied – that modify the template parsed event sequence in order to create the results we expect by combining the original template with our data.

It also includes – by default – a cache that stores parsed templates; the sequence of events resulting from reading and parsing template files before processing them. This is especially useful when working in a web application, and builds on the following concepts:

Input/Output is almost always the slowest part of any application. In-memory processing is extremely quick by comparison.

Cloning an existing in-memory event sequence is always much quicker than reading a template file, parsing it and creating a new event sequence for it.

Web applications usually have only a few dozen templates.

Template files are small-to-medium size, and they are not modified while the application is running.

This all leads to the idea that caching the most used templates in a web application is feasible without wasting large amounts of memory, and also that it will save a lot of time that would be spent on input/output operations on a small set of files that, in fact, never change.

And how can we take control of this cache? First, we’ve learned before that we can enable or disable it at the Template Resolver, even acting only on specific templates:

// Default is true

templateResolver.setCacheable(false);

templateResolver.getCacheablePatternSpec().addPattern("/users/\*");

Also, we could modify its configuration by establishing our own Cache Manager object, which could be an instance of the default StandardCacheManager implementation:

// Default is 50

StandardCacheManager cacheManager = new StandardCacheManager();

cacheManager.setTemplateCacheMaxSize(100);

...

templateEngine.setCacheManager(cacheManager);

Refer to the javadoc API of org.thymeleaf.cache.StandardCacheManager for more info on configuring the caches.

Entries can be manually removed from the template cache:

// Clear the cache completely

templateEngine.clearTemplateCache();

// Clear a specific template from the cache

templateEngine.clearTemplateCacheFor("/users/userList");

# Decoupled Template Logic

## Decoupled logic: The concept

So far we have worked for our Grocery Store with templates done the usual way, with logic being inserted into our templates in the form of attributes.

But Thymeleaf also allows us to completely decouple the template markup from its logic, allowing the creation of completely logic-less markup templates in the HTML and XML template modes.

The main idea is that template logic will be defined in a separate logic file (more exactly a logic resource, as it doesn’t need to be a file). By default, that logic resource will be an additional file living in the same place (e.g. folder) as the template file, with the same name but with .th.xml extension:

/templates

+->/home.html

+->/home.th.xml

So the home.html file can be completely logic-less. It might look like this:

<!DOCTYPE html>

<html>

<body>

<table id="usersTable">

<tr>

<td class="username">Jeremy Grapefruit</td>

<td class="usertype">Normal User</td>

</tr>

<tr>

<td class="username">Alice Watermelon</td>

<td class="usertype">Administrator</td>

</tr>

</table>

</body>

</html>

Absolutely no Thymeleaf code there. This is a template file that a designer with no Thymeleaf or templating knowledge could have created, edited and/or understood. Or a fragment of HTML provided by some external system with no Thymeleaf hooks at all.

Let’s now turn that home.html template into a Thymeleaf template by creating our additional home.th.xml file like this:

<?xml version="1.0"?>

<thlogic>

<attr sel="#usersTable" th:remove="all-but-first">

<attr sel="/tr[0]" th:each="user : ${users}">

<attr sel="td.username" th:text="${user.name}" />

<attr sel="td.usertype" th:text="#{|user.type.${user.type}|}" />

</attr>

</attr>

</thlogic>

Here we can see a lot of <attr> tags inside a thlogic block. Those <attr> tags perform attribute injection on nodes of the original template selected by means of their sel attributes, which contain Thymeleaf markup selectors (actually AttoParser markup selectors).

Also note that <attr> tags can be nested so that their selectors are appended. That sel="/tr[0]" above, for example, will be processed as sel="#usersTable/tr[0]". And the selector for the user name <td> will be processed as sel="#usersTable/tr[0]//td.username".

So once merged, both files seen above will be the same as:

<!DOCTYPE html>

<html>

<body>

<table id="usersTable" th:remove="all-but-first">

<tr th:each="user : ${users}">

<td class="username" th:text="${user.name}">Jeremy Grapefruit</td>

<td class="usertype" th:text="#{|user.type.${user.type}|}">Normal User</td>

</tr>

<tr>

<td class="username">Alice Watermelon</td>

<td class="usertype">Administrator</td>

</tr>

</table>

</body>

</html>

This looks more familiar, and is indeed less verbose than creating two separate files. But the advantage of decoupled templates is that we can give for our templates total independence from Thymeleaf, and therefore better maintainability from the design standpoint.

Of course some contracts between designers or developers will still be needed – e.g. the fact that the users <table> will need an id="usersTable" –, but in many scenarios a pure-HTML template will be a much better communication artifact between design and development teams.

## Configuring decoupled templates

Enabling decoupled templates

Decoupled logic will not be expected for every template by default. Instead, the configured template resolvers (implementations of ITemplateResolver) will need to specifically mark the templates they resolve as using decoupled logic.

Except for StringTemplateResolver (which does not allow decoupled logic), all other out-of-the-box implementations of ITemplateResolver will provide a flag called useDecoupledLogic that will mark all templates resolved by that resolver as potentially having all or part of its logic living in a separate resource:

final ServletContextTemplateResolver templateResolver =

new ServletContextTemplateResolver(servletContext);

...

templateResolver.setUseDecoupledLogic(true);

Mixing coupled and decoupled logic

Decoupled template logic, when enabled, is not a requirement. When enabled, it means that the engine will look for a resource containing decoupled logic, parsing and merging it with the original template if it exists. No error will be thrown if the decoupled logic resource does not exist.

Also, in the same template we can mix both coupled and decoupled logic, for example by adding some Thymeleaf attributes at the original template file but leaving others for the separate decoupled logic file. The most common case for this is using the new (in v3.0) th:ref attribute.

## The th:ref attribute

th:ref is only a marker attribute. It does nothing from the processing standpoint and simply disappears when the template is processed, but its usefulness lies in the fact that it acts as a markup reference, i.e. it can be resolved by name from a markup selector just like a tag name or a fragment (th:fragment).

So if we have a selector like:

<attr sel="whatever" .../>

This will match:

Any <whatever> tags.

Any tags with a th:fragment="whatever" attribute.

Any tags with a th:ref="whatever" attribute.

What is the advantage of th:ref against, for example, using a pure-HTML id attribute? Merely the fact that we might not want to add so many id and class attributes to our tags to act as logic anchors, which might end up polluting our output.

And in the same sense, what is the disadvantage of th:ref? Well, obviously that we’d be adding a bit of Thymeleaf logic (“logic”) to our templates.

Note this applicability of the th:ref attribute does not only apply to decoupled logic template files: it works the same in other types of scenarios, like in fragment expressions (~{...}).

## Performance impact of decoupled templates

The impact is extremely small. When a resolved template is marked to use decoupled logic and it is not cached, the template logic resource will be resolved first, parsed and processed into a secuence of instructions in-memory: basically a list of attributes to be injected to each markup selector.

But this is the only additional step required because, after this, the real template will be parsed, and while it is parsed these attributes will be injected on-the-fly by the parser itself, thanks to the advanced capabilities for node selection in AttoParser. So parsed nodes will come out of the parser as if they had their injected attributes written in the original template file.

The biggest advantage of this? When a template is configured to be cached, it will be cached already containing the injected attributes. So the overhead of using decoupled templates for cacheable templates, once they are cached, will be absolutely zero.

## Resolution of decoupled logic

The way Thymeleaf resolves the decoupled logic resources corresponding to each template is configurable by the user. It is determined by an extension point, the org.thymeleaf.templateparser.markup.decoupled.IDecoupledTemplateLogicResolver, for which a default implementation is provided: StandardDecoupledTemplateLogicResolver.

What does this standard implementation do?

First, it applies a prefix and a suffix to the base name of the template resource (obtained by means of its ITemplateResource#getBaseName() method). Both prefix and suffix can be configured and, by default, the prefix will be empty and the suffix will be .th.xml.

Second, it asks the template resource to resolve a relative resource with the computed name by means of its ITemplateResource#relative(String relativeLocation) method.

The specific implementation of IDecoupledTemplateLogicResolver to be used can be configured at the TemplateEngine easily:

final StandardDecoupledTemplateLogicResolver decoupledresolver =

new StandardDecoupledTemplateLogicResolver();

decoupledResolver.setPrefix("../viewlogic/");

...

templateEngine.setDecoupledTemplateLogicResolver(decoupledResolver);

# Appendix A: Expression Basic Objects

Some objects and variable maps are always available to be invoked. Let’s see them:

Base objects

#ctx : the context object. An implementation of org.thymeleaf.context.IContext or org.thymeleaf.context.IWebContext depending on our environment (standalone or web).

Note #vars and #root are synomyns for the same object, but using #ctx is recommended.

/\*

\* ======================================================================

\* See javadoc API for class org.thymeleaf.context.IContext

\* ======================================================================

\*/

${#ctx.locale}

${#ctx.variableNames}

/\*

\* ======================================================================

\* See javadoc API for class org.thymeleaf.context.IWebContext

\* ======================================================================

\*/

${#ctx.request}

${#ctx.response}

${#ctx.session}

${#ctx.servletContext}

#locale : direct access to the java.util.Locale associated with current request.

${#locale}

Web context namespaces for request/session attributes, etc.

When using Thymeleaf in a web environment, we can use a series of shortcuts for accessing request parameters, session attributes and application attributes:

Note these are not context objects, but maps added to the context as variables, so we access them without #. In some way, they act as namespaces.

param : for retrieving request parameters. ${param.foo} is a String[] with the values of the foo request parameter, so ${param.foo[0]} will normally be used for getting the first value.

/\*

\* ============================================================================

\* See javadoc API for class org.thymeleaf.context.WebRequestParamsVariablesMap

\* ============================================================================

\*/

${param.foo} // Retrieves a String[] with the values of request parameter 'foo'

${param.size()}

${param.isEmpty()}

${param.containsKey('foo')}

...

session : for retrieving session attributes.

/\*

\* ======================================================================

\* See javadoc API for class org.thymeleaf.context.WebSessionVariablesMap

\* ======================================================================

\*/

${session.foo} // Retrieves the session atttribute 'foo'

${session.size()}

${session.isEmpty()}

${session.containsKey('foo')}

...

application : for retrieving application/servlet context attributes.

/\*

\* =============================================================================

\* See javadoc API for class org.thymeleaf.context.WebServletContextVariablesMap

\* =============================================================================

\*/

${application.foo} // Retrieves the ServletContext atttribute 'foo'

${application.size()}

${application.isEmpty()}

${application.containsKey('foo')}

...

Note there is no need to specify a namespace for accessing request attributes (as opposed to request parameters) because all request attributes are automatically added to the context as variables in the context root:

${myRequestAttribute}

Web context objects

Inside a web environment there is also direct access to the following objects (note these are objects, not maps/namespaces):

#request : direct access to the javax.servlet.http.HttpServletRequest object associated with the current request.

${#request.getAttribute('foo')}

${#request.getParameter('foo')}

${#request.getContextPath()}

${#request.getRequestName()}

...

#session : direct access to the javax.servlet.http.HttpSession object associated with the current request.

${#session.getAttribute('foo')}

${#session.id}

${#session.lastAccessedTime}

...

#servletContext : direct access to the javax.servlet.ServletContext object associated with the current request.

${#servletContext.getAttribute('foo')}

${#servletContext.contextPath}

...

# Appendix B: Expression Utility Objects

Execution Info

#execInfo : expression object providing useful information about the template being processed inside Thymeleaf Standard Expressions.

/\*

\* ======================================================================

\* See javadoc API for class org.thymeleaf.expression.ExecutionInfo

\* ======================================================================

\*/

/\*

\* Return the name and mode of the 'leaf' template. This means the template

\* from where the events being processed were parsed. So if this piece of

\* code is not in the root template "A" but on a fragment being inserted

\* into "A" from another template called "B", this will return "B" as a

\* name, and B's mode as template mode.

\*/

${#execInfo.templateName}

${#execInfo.templateMode}

/\*

\* Return the name and mode of the 'root' template. This means the template

\* that the template engine was originally asked to process. So if this

\* piece of code is not in the root template "A" but on a fragment being

\* inserted into "A" from another template called "B", this will still

\* return "A" and A's template mode.

\*/

${#execInfo.processedTemplateName}

${#execInfo.processedTemplateMode}

/\*

\* Return the stacks (actually, List<String> or List<TemplateMode>) of

\* templates being processed. The first element will be the

\* 'processedTemplate' (the root one), the last one will be the 'leaf'

\* template, and in the middle all the fragments inserted in nested

\* manner to reach the leaf from the root will appear.

\*/

${#execInfo.templateNames}

${#execInfo.templateModes}

/\*

\* Return the stack of templates being processed similarly (and in the

\* same order) to 'templateNames' and 'templateModes', but returning

\* a List<TemplateData> with the full template metadata.

\*/

${#execInfo.templateStack}

Messages

#messages : utility methods for obtaining externalized messages inside variables expressions, in the same way as they would be obtained using #{...} syntax.

/\*

\* ======================================================================

\* See javadoc API for class org.thymeleaf.expression.Messages

\* ======================================================================

\*/

/\*

\* Obtain externalized messages. Can receive a single key, a key plus arguments,

\* or an array/list/set of keys (in which case it will return an array/list/set of

\* externalized messages).

\* If a message is not found, a default message (like '??msgKey??') is returned.

\*/

${#messages.msg('msgKey')}

${#messages.msg('msgKey', param1)}

${#messages.msg('msgKey', param1, param2)}

${#messages.msg('msgKey', param1, param2, param3)}

${#messages.msgWithParams('msgKey', new Object[] {param1, param2, param3, param4})}

${#messages.arrayMsg(messageKeyArray)}

${#messages.listMsg(messageKeyList)}

${#messages.setMsg(messageKeySet)}

/\*

\* Obtain externalized messages or null. Null is returned instead of a default

\* message if a message for the specified key is not found.

\*/

${#messages.msgOrNull('msgKey')}

${#messages.msgOrNull('msgKey', param1)}

${#messages.msgOrNull('msgKey', param1, param2)}

${#messages.msgOrNull('msgKey', param1, param2, param3)}

${#messages.msgOrNullWithParams('msgKey', new Object[] {param1, param2, param3, param4})}

${#messages.arrayMsgOrNull(messageKeyArray)}

${#messages.listMsgOrNull(messageKeyList)}

${#messages.setMsgOrNull(messageKeySet)}

URIs/URLs

#uris : utility object for performing URI/URL operations (esp. escaping/unescaping) inside Thymeleaf Standard Expressions.

/\*

\* ======================================================================

\* See javadoc API for class org.thymeleaf.expression.Uris

\* ======================================================================

\*/

/\*

\* Escape/Unescape as a URI/URL path

\*/

${#uris.escapePath(uri)}

${#uris.escapePath(uri, encoding)}

${#uris.unescapePath(uri)}

${#uris.unescapePath(uri, encoding)}

/\*

\* Escape/Unescape as a URI/URL path segment (between '/' symbols)

\*/

${#uris.escapePathSegment(uri)}

${#uris.escapePathSegment(uri, encoding)}

${#uris.unescapePathSegment(uri)}

${#uris.unescapePathSegment(uri, encoding)}

/\*

\* Escape/Unescape as a Fragment Identifier (#frag)

\*/

${#uris.escapeFragmentId(uri)}

${#uris.escapeFragmentId(uri, encoding)}

${#uris.unescapeFragmentId(uri)}

${#uris.unescapeFragmentId(uri, encoding)}

/\*

\* Escape/Unescape as a Query Parameter (?var=value)

\*/

${#uris.escapeQueryParam(uri)}

${#uris.escapeQueryParam(uri, encoding)}

${#uris.unescapeQueryParam(uri)}

${#uris.unescapeQueryParam(uri, encoding)}

Conversions

#conversions : utility object that allows the execution of the Conversion Service at any point of a template:

/\*

\* ======================================================================

\* See javadoc API for class org.thymeleaf.expression.Conversions

\* ======================================================================

\*/

/\*

\* Execute the desired conversion of the 'object' value into the

\* specified class.

\*/

${#conversions.convert(object, 'java.util.TimeZone')}

${#conversions.convert(object, targetClass)}

Dates

#dates : utility methods for java.util.Date objects:

/\*

\* ======================================================================

\* See javadoc API for class org.thymeleaf.expression.Dates

\* ======================================================================

\*/

/\*

\* Format date with the standard locale format

\* Also works with arrays, lists or sets

\*/

${#dates.format(date)}

${#dates.arrayFormat(datesArray)}

${#dates.listFormat(datesList)}

${#dates.setFormat(datesSet)}

/\*

\* Format date with the ISO8601 format

\* Also works with arrays, lists or sets

\*/

${#dates.formatISO(date)}

${#dates.arrayFormatISO(datesArray)}

${#dates.listFormatISO(datesList)}

${#dates.setFormatISO(datesSet)}

/\*

\* Format date with the specified pattern

\* Also works with arrays, lists or sets

\*/

${#dates.format(date, 'dd/MMM/yyyy HH:mm')}

${#dates.arrayFormat(datesArray, 'dd/MMM/yyyy HH:mm')}

${#dates.listFormat(datesList, 'dd/MMM/yyyy HH:mm')}

${#dates.setFormat(datesSet, 'dd/MMM/yyyy HH:mm')}

/\*

\* Obtain date properties

\* Also works with arrays, lists or sets

\*/

${#dates.day(date)} // also arrayDay(...), listDay(...), etc.

${#dates.month(date)} // also arrayMonth(...), listMonth(...), etc.

${#dates.monthName(date)} // also arrayMonthName(...), listMonthName(...), etc.

${#dates.monthNameShort(date)} // also arrayMonthNameShort(...), listMonthNameShort(...), etc.

${#dates.year(date)} // also arrayYear(...), listYear(...), etc.

${#dates.dayOfWeek(date)} // also arrayDayOfWeek(...), listDayOfWeek(...), etc.

${#dates.dayOfWeekName(date)} // also arrayDayOfWeekName(...), listDayOfWeekName(...), etc.

${#dates.dayOfWeekNameShort(date)} // also arrayDayOfWeekNameShort(...), listDayOfWeekNameShort(...), etc.

${#dates.hour(date)} // also arrayHour(...), listHour(...), etc.

${#dates.minute(date)} // also arrayMinute(...), listMinute(...), etc.

${#dates.second(date)} // also arraySecond(...), listSecond(...), etc.

${#dates.millisecond(date)} // also arrayMillisecond(...), listMillisecond(...), etc.

/\*

\* Create date (java.util.Date) objects from its components

\*/

${#dates.create(year,month,day)}

${#dates.create(year,month,day,hour,minute)}

${#dates.create(year,month,day,hour,minute,second)}

${#dates.create(year,month,day,hour,minute,second,millisecond)}

/\*

\* Create a date (java.util.Date) object for the current date and time

\*/

${#dates.createNow()}

${#dates.createNowForTimeZone()}

/\*

\* Create a date (java.util.Date) object for the current date (time set to 00:00)

\*/

${#dates.createToday()}

${#dates.createTodayForTimeZone()}

Calendars

#calendars : analogous to #dates, but for java.util.Calendar objects:

/\*

\* ======================================================================

\* See javadoc API for class org.thymeleaf.expression.Calendars

\* ======================================================================

\*/

/\*

\* Format calendar with the standard locale format

\* Also works with arrays, lists or sets

\*/

${#calendars.format(cal)}

${#calendars.arrayFormat(calArray)}

${#calendars.listFormat(calList)}

${#calendars.setFormat(calSet)}

/\*

\* Format calendar with the ISO8601 format

\* Also works with arrays, lists or sets

\*/

${#calendars.formatISO(cal)}

${#calendars.arrayFormatISO(calArray)}

${#calendars.listFormatISO(calList)}

${#calendars.setFormatISO(calSet)}

/\*

\* Format calendar with the specified pattern

\* Also works with arrays, lists or sets

\*/

${#calendars.format(cal, 'dd/MMM/yyyy HH:mm')}

${#calendars.arrayFormat(calArray, 'dd/MMM/yyyy HH:mm')}

${#calendars.listFormat(calList, 'dd/MMM/yyyy HH:mm')}

${#calendars.setFormat(calSet, 'dd/MMM/yyyy HH:mm')}

/\*

\* Obtain calendar properties

\* Also works with arrays, lists or sets

\*/

${#calendars.day(date)} // also arrayDay(...), listDay(...), etc.

${#calendars.month(date)} // also arrayMonth(...), listMonth(...), etc.

${#calendars.monthName(date)} // also arrayMonthName(...), listMonthName(...), etc.

${#calendars.monthNameShort(date)} // also arrayMonthNameShort(...), listMonthNameShort(...), etc.

${#calendars.year(date)} // also arrayYear(...), listYear(...), etc.

${#calendars.dayOfWeek(date)} // also arrayDayOfWeek(...), listDayOfWeek(...), etc.

${#calendars.dayOfWeekName(date)} // also arrayDayOfWeekName(...), listDayOfWeekName(...), etc.

${#calendars.dayOfWeekNameShort(date)} // also arrayDayOfWeekNameShort(...), listDayOfWeekNameShort(...), etc.

${#calendars.hour(date)} // also arrayHour(...), listHour(...), etc.

${#calendars.minute(date)} // also arrayMinute(...), listMinute(...), etc.

${#calendars.second(date)} // also arraySecond(...), listSecond(...), etc.

${#calendars.millisecond(date)} // also arrayMillisecond(...), listMillisecond(...), etc.

/\*

\* Create calendar (java.util.Calendar) objects from its components

\*/

${#calendars.create(year,month,day)}

${#calendars.create(year,month,day,hour,minute)}

${#calendars.create(year,month,day,hour,minute,second)}

${#calendars.create(year,month,day,hour,minute,second,millisecond)}

${#calendars.createForTimeZone(year,month,day,timeZone)}

${#calendars.createForTimeZone(year,month,day,hour,minute,timeZone)}

${#calendars.createForTimeZone(year,month,day,hour,minute,second,timeZone)}

${#calendars.createForTimeZone(year,month,day,hour,minute,second,millisecond,timeZone)}

/\*

\* Create a calendar (java.util.Calendar) object for the current date and time

\*/

${#calendars.createNow()}

${#calendars.createNowForTimeZone()}

/\*

\* Create a calendar (java.util.Calendar) object for the current date (time set to 00:00)

\*/

${#calendars.createToday()}

${#calendars.createTodayForTimeZone()}

Numbers

#numbers : utility methods for number objects:

/\*

\* ======================================================================

\* See javadoc API for class org.thymeleaf.expression.Numbers

\* ======================================================================

\*/

/\*

\* ==========================

\* Formatting integer numbers

\* ==========================

\*/

/\*

\* Set minimum integer digits.

\* Also works with arrays, lists or sets

\*/

${#numbers.formatInteger(num,3)}

${#numbers.arrayFormatInteger(numArray,3)}

${#numbers.listFormatInteger(numList,3)}

${#numbers.setFormatInteger(numSet,3)}

/\*

\* Set minimum integer digits and thousands separator:

\* 'POINT', 'COMMA', 'WHITESPACE', 'NONE' or 'DEFAULT' (by locale).

\* Also works with arrays, lists or sets

\*/

${#numbers.formatInteger(num,3,'POINT')}

${#numbers.arrayFormatInteger(numArray,3,'POINT')}

${#numbers.listFormatInteger(numList,3,'POINT')}

${#numbers.setFormatInteger(numSet,3,'POINT')}

/\*

\* ==========================

\* Formatting decimal numbers

\* ==========================

\*/

/\*

\* Set minimum integer digits and (exact) decimal digits.

\* Also works with arrays, lists or sets

\*/

${#numbers.formatDecimal(num,3,2)}

${#numbers.arrayFormatDecimal(numArray,3,2)}

${#numbers.listFormatDecimal(numList,3,2)}

${#numbers.setFormatDecimal(numSet,3,2)}

/\*

\* Set minimum integer digits and (exact) decimal digits, and also decimal separator.

\* Also works with arrays, lists or sets

\*/

${#numbers.formatDecimal(num,3,2,'COMMA')}

${#numbers.arrayFormatDecimal(numArray,3,2,'COMMA')}

${#numbers.listFormatDecimal(numList,3,2,'COMMA')}

${#numbers.setFormatDecimal(numSet,3,2,'COMMA')}

/\*

\* Set minimum integer digits and (exact) decimal digits, and also thousands and

\* decimal separator.

\* Also works with arrays, lists or sets

\*/

${#numbers.formatDecimal(num,3,'POINT',2,'COMMA')}

${#numbers.arrayFormatDecimal(numArray,3,'POINT',2,'COMMA')}

${#numbers.listFormatDecimal(numList,3,'POINT',2,'COMMA')}

${#numbers.setFormatDecimal(numSet,3,'POINT',2,'COMMA')}

/\*

\* =====================

\* Formatting currencies

\* =====================

\*/

${#numbers.formatCurrency(num)}

${#numbers.arrayFormatCurrency(numArray)}

${#numbers.listFormatCurrency(numList)}

${#numbers.setFormatCurrency(numSet)}

/\*

\* ======================

\* Formatting percentages

\* ======================

\*/

${#numbers.formatPercent(num)}

${#numbers.arrayFormatPercent(numArray)}

${#numbers.listFormatPercent(numList)}

${#numbers.setFormatPercent(numSet)}

/\*

\* Set minimum integer digits and (exact) decimal digits.

\*/

${#numbers.formatPercent(num, 3, 2)}

${#numbers.arrayFormatPercent(numArray, 3, 2)}

${#numbers.listFormatPercent(numList, 3, 2)}

${#numbers.setFormatPercent(numSet, 3, 2)}

/\*

\* ===============

\* Utility methods

\* ===============

\*/

/\*

\* Create a sequence (array) of integer numbers going

\* from x to y

\*/

${#numbers.sequence(from,to)}

${#numbers.sequence(from,to,step)}

Strings

#strings : utility methods for String objects:

/\*

\* ======================================================================

\* See javadoc API for class org.thymeleaf.expression.Strings

\* ======================================================================

\*/

/\*

\* Null-safe toString()

\*/

${#strings.toString(obj)} // also array\*, list\* and set\*

/\*

\* Check whether a String is empty (or null). Performs a trim() operation before check

\* Also works with arrays, lists or sets

\*/

${#strings.isEmpty(name)}

${#strings.arrayIsEmpty(nameArr)}

${#strings.listIsEmpty(nameList)}

${#strings.setIsEmpty(nameSet)}

/\*

\* Perform an 'isEmpty()' check on a string and return it if false, defaulting to

\* another specified string if true.

\* Also works with arrays, lists or sets

\*/

${#strings.defaultString(text,default)}

${#strings.arrayDefaultString(textArr,default)}

${#strings.listDefaultString(textList,default)}

${#strings.setDefaultString(textSet,default)}

/\*

\* Check whether a fragment is contained in a String

\* Also works with arrays, lists or sets

\*/

${#strings.contains(name,'ez')} // also array\*, list\* and set\*

${#strings.containsIgnoreCase(name,'ez')} // also array\*, list\* and set\*

/\*

\* Check whether a String starts or ends with a fragment

\* Also works with arrays, lists or sets

\*/

${#strings.startsWith(name,'Don')} // also array\*, list\* and set\*

${#strings.endsWith(name,endingFragment)} // also array\*, list\* and set\*

/\*

\* Substring-related operations

\* Also works with arrays, lists or sets

\*/

${#strings.indexOf(name,frag)} // also array\*, list\* and set\*

${#strings.substring(name,3,5)} // also array\*, list\* and set\*

${#strings.substringAfter(name,prefix)} // also array\*, list\* and set\*

${#strings.substringBefore(name,suffix)} // also array\*, list\* and set\*

${#strings.replace(name,'las','ler')} // also array\*, list\* and set\*

/\*

\* Append and prepend

\* Also works with arrays, lists or sets

\*/

${#strings.prepend(str,prefix)} // also array\*, list\* and set\*

${#strings.append(str,suffix)} // also array\*, list\* and set\*

/\*

\* Change case

\* Also works with arrays, lists or sets

\*/

${#strings.toUpperCase(name)} // also array\*, list\* and set\*

${#strings.toLowerCase(name)} // also array\*, list\* and set\*

/\*

\* Split and join

\*/

${#strings.arrayJoin(namesArray,',')}

${#strings.listJoin(namesList,',')}

${#strings.setJoin(namesSet,',')}

${#strings.arraySplit(namesStr,',')} // returns String[]

${#strings.listSplit(namesStr,',')} // returns List<String>

${#strings.setSplit(namesStr,',')} // returns Set<String>

/\*

\* Trim

\* Also works with arrays, lists or sets

\*/

${#strings.trim(str)} // also array\*, list\* and set\*

/\*

\* Compute length

\* Also works with arrays, lists or sets

\*/

${#strings.length(str)} // also array\*, list\* and set\*

/\*

\* Abbreviate text making it have a maximum size of n. If text is bigger, it

\* will be clipped and finished in "..."

\* Also works with arrays, lists or sets

\*/

${#strings.abbreviate(str,10)} // also array\*, list\* and set\*

/\*

\* Convert the first character to upper-case (and vice-versa)

\*/

${#strings.capitalize(str)} // also array\*, list\* and set\*

${#strings.unCapitalize(str)} // also array\*, list\* and set\*

/\*

\* Convert the first character of every word to upper-case

\*/

${#strings.capitalizeWords(str)} // also array\*, list\* and set\*

${#strings.capitalizeWords(str,delimiters)} // also array\*, list\* and set\*

/\*

\* Escape the string

\*/

${#strings.escapeXml(str)} // also array\*, list\* and set\*

${#strings.escapeJava(str)} // also array\*, list\* and set\*

${#strings.escapeJavaScript(str)} // also array\*, list\* and set\*

${#strings.unescapeJava(str)} // also array\*, list\* and set\*

${#strings.unescapeJavaScript(str)} // also array\*, list\* and set\*

/\*

\* Null-safe comparison and concatenation

\*/

${#strings.equals(first, second)}

${#strings.equalsIgnoreCase(first, second)}

${#strings.concat(values...)}

${#strings.concatReplaceNulls(nullValue, values...)}

/\*

\* Random

\*/

${#strings.randomAlphanumeric(count)}

Objects

#objects : utility methods for objects in general

/\*

\* ======================================================================

\* See javadoc API for class org.thymeleaf.expression.Objects

\* ======================================================================

\*/

/\*

\* Return obj if it is not null, and default otherwise

\* Also works with arrays, lists or sets

\*/

${#objects.nullSafe(obj,default)}

${#objects.arrayNullSafe(objArray,default)}

${#objects.listNullSafe(objList,default)}

${#objects.setNullSafe(objSet,default)}

Booleans

#bools : utility methods for boolean evaluation

/\*

\* ======================================================================

\* See javadoc API for class org.thymeleaf.expression.Bools

\* ======================================================================

\*/

/\*

\* Evaluate a condition in the same way that it would be evaluated in a th:if tag

\* (see conditional evaluation chapter afterwards).

\* Also works with arrays, lists or sets

\*/

${#bools.isTrue(obj)}

${#bools.arrayIsTrue(objArray)}

${#bools.listIsTrue(objList)}

${#bools.setIsTrue(objSet)}

/\*

\* Evaluate with negation

\* Also works with arrays, lists or sets

\*/

${#bools.isFalse(cond)}

${#bools.arrayIsFalse(condArray)}

${#bools.listIsFalse(condList)}

${#bools.setIsFalse(condSet)}

/\*

\* Evaluate and apply AND operator

\* Receive an array, a list or a set as parameter

\*/

${#bools.arrayAnd(condArray)}

${#bools.listAnd(condList)}

${#bools.setAnd(condSet)}

/\*

\* Evaluate and apply OR operator

\* Receive an array, a list or a set as parameter

\*/

${#bools.arrayOr(condArray)}

${#bools.listOr(condList)}

${#bools.setOr(condSet)}

Arrays

#arrays : utility methods for arrays

/\*

\* ======================================================================

\* See javadoc API for class org.thymeleaf.expression.Arrays

\* ======================================================================

\*/

/\*

\* Converts to array, trying to infer array component class.

\* Note that if resulting array is empty, or if the elements

\* of the target object are not all of the same class,

\* this method will return Object[].

\*/

${#arrays.toArray(object)}

/\*

\* Convert to arrays of the specified component class.

\*/

${#arrays.toStringArray(object)}

${#arrays.toIntegerArray(object)}

${#arrays.toLongArray(object)}

${#arrays.toDoubleArray(object)}

${#arrays.toFloatArray(object)}

${#arrays.toBooleanArray(object)}

/\*

\* Compute length

\*/

${#arrays.length(array)}

/\*

\* Check whether array is empty

\*/

${#arrays.isEmpty(array)}

/\*

\* Check if element or elements are contained in array

\*/

${#arrays.contains(array, element)}

${#arrays.containsAll(array, elements)}

Lists

#lists : utility methods for lists

/\*

\* ======================================================================

\* See javadoc API for class org.thymeleaf.expression.Lists

\* ======================================================================

\*/

/\*

\* Converts to list

\*/

${#lists.toList(object)}

/\*

\* Compute size

\*/

${#lists.size(list)}

/\*

\* Check whether list is empty

\*/

${#lists.isEmpty(list)}

/\*

\* Check if element or elements are contained in list

\*/

${#lists.contains(list, element)}

${#lists.containsAll(list, elements)}

/\*

\* Sort a copy of the given list. The members of the list must implement

\* comparable or you must define a comparator.

\*/

${#lists.sort(list)}

${#lists.sort(list, comparator)}

Sets

#sets : utility methods for sets

/\*

\* ======================================================================

\* See javadoc API for class org.thymeleaf.expression.Sets

\* ======================================================================

\*/

/\*

\* Converts to set

\*/

${#sets.toSet(object)}

/\*

\* Compute size

\*/

${#sets.size(set)}

/\*

\* Check whether set is empty

\*/

${#sets.isEmpty(set)}

/\*

\* Check if element or elements are contained in set

\*/

${#sets.contains(set, element)}

${#sets.containsAll(set, elements)}

Maps

#maps : utility methods for maps

/\*

\* ======================================================================

\* See javadoc API for class org.thymeleaf.expression.Maps

\* ======================================================================

\*/

/\*

\* Compute size

\*/

${#maps.size(map)}

/\*

\* Check whether map is empty

\*/

${#maps.isEmpty(map)}

/\*

\* Check if key/s or value/s are contained in maps

\*/

${#maps.containsKey(map, key)}

${#maps.containsAllKeys(map, keys)}

${#maps.containsValue(map, value)}

${#maps.containsAllValues(map, value)}

Aggregates

#aggregates : utility methods for creating aggregates on arrays or collections

/\*

\* ======================================================================

\* See javadoc API for class org.thymeleaf.expression.Aggregates

\* ======================================================================

\*/

/\*

\* Compute sum. Returns null if array or collection is empty

\*/

${#aggregates.sum(array)}

${#aggregates.sum(collection)}

/\*

\* Compute average. Returns null if array or collection is empty

\*/

${#aggregates.avg(array)}

${#aggregates.avg(collection)}

IDs

#ids : utility methods for dealing with id attributes that might be repeated (for example, as a result of an iteration).

/\*

\* ======================================================================

\* See javadoc API for class org.thymeleaf.expression.Ids

\* ======================================================================

\*/

/\*

\* Normally used in th:id attributes, for appending a counter to the id attribute value

\* so that it remains unique even when involved in an iteration process.

\*/

${#ids.seq('someId')}

/\*

\* Normally used in th:for attributes in <label> tags, so that these labels can refer to Ids

\* generated by means if the #ids.seq(...) function.

\*

\* Depending on whether the <label> goes before or after the element with the #ids.seq(...)

\* function, the "next" (label goes before "seq") or the "prev" function (label goes after

\* "seq") function should be called.

\*/

${#ids.next('someId')}

${#ids.prev('someId')}

# Appendix C: Markup Selector Syntax

Thymeleaf’s Markup Selectors are directly borrowed from Thymeleaf’s parsing library: AttoParser.

The syntax for this selectors has large similarities with that of selectors in XPath, CSS and jQuery, which makes them easy to use for most users. You can have a look at the complete syntax reference at the AttoParser documentation.

For example, the following selector will select every <div> with the class content, in every position inside the markup (note this is not as concise as it could be, read on to know why):

<div th:insert="mytemplate :: //div[@class='content']">...</div>

The basic syntax includes:

/x means direct children of the current node with name x.

//x means children of the current node with name x, at any depth.

x[@z="v"] means elements with name x and an attribute called z with value “v”.

x[@z1="v1" and @z2="v2"] means elements with name x and attributes z1 and z2 with values “v1” and “v2”, respectively.

x[i] means element with name x positioned in number i among its siblings.

x[@z="v"][i] means elements with name x, attribute z with value “v” and positioned in number i among its siblings that also match this condition.

But more concise syntax can also be used:

x is exactly equivalent to //x (search an element with name or reference x at any depth level, a reference being a th:ref or a th:fragment attribute).

Selectors are also allowed without element name/reference, as long as they include a specification of arguments. So [@class='oneclass'] is a valid selector that looks for any elements (tags) with a class attribute with value "oneclass".

Advanced attribute selection features:

Besides = (equal), other comparison operators are also valid: != (not equal), ^= (starts with) and $= (ends with). For example: x[@class^='section'] means elements with name x and a value for attribute class that starts with section.

Attributes can be specified both starting with @ (XPath-style) and without (jQuery-style). So x[z='v'] is equivalent to x[@z='v'].

Multiple-attribute modifiers can be joined both with and (XPath-style) and also by chaining multiple modifiers (jQuery-style). So x[@z1='v1' and @z2='v2'] is actually equivalent to x[@z1='v1'][@z2='v2'] (and also to x[z1='v1'][z2='v2']).

Direct jQuery-like selectors:

x.oneclass is equivalent to x[class='oneclass'].

.oneclass is equivalent to [class='oneclass'].

x#oneid is equivalent to x[id='oneid'].

#oneid is equivalent to [id='oneid'].

x%oneref means <x> tags that have a th:ref="oneref" or th:fragment="oneref" attribute.

%oneref means any tags that have a th:ref="oneref" or th:fragment="oneref" attribute. Note this is actually equivalent to simply oneref because references can be used instead of element names.

Direct selectors and attribute selectors can be mixed: a.external[@href^='https'].

So the above Markup Selector expression:

<div th:insert="mytemplate :: //div[@class='content']">...</div>

Could be written as:

<div th:insert="mytemplate :: div.content">...</div>

Examining a different example, this:

<div th:replace="mytemplate :: myfrag">...</div>

Will look for a th:fragment="myfrag" fragment signature (or th:ref references). But would also look for tags with name myfrag if they existed (which they don’t, in HTML). Note the difference with:

<div th:replace="mytemplate :: .myfrag">...</div>

…which will actually look for any elements with class="myfrag", without caring about th:fragment signatures (or th:ref references).

Multivalued class matching

Markup Selectors understand the class attribute to be multivalued, and therefore allow the application of selectors on this attribute even if the element has several class values.

For example, div.two will match <div class="one two three" />

#说明:

翻译版本信息:

\*\*Document version\*\* : 20170418 18 April 2017

\*\*Project version\*\* : 3.0.5.RELEASE

最新版本信息:

\*\*Document version\*\* : 20170418 18 April 2017

\*\*Project version\*\* : 3.0.5.RELEASE

#参考资料:

[英文原文 Tutorial: Using Thymeleaf](http://www.thymeleaf.org/doc/tutorials/2.1/usingthymeleaf.html" \l "using-texts)

[简体译文 Thymeleaf3.0内容](http://www.cnblogs.com/jiangchao226/p/5900222.html)

[Thymeleaf源码项目](https://github.com/thymeleaf)