

# *Thymeleaf*



## 教程： 使用 *Thymeleaf*

文件版本： 20180605 - 2018年6月5日

项目版本： 3.0.9.RELEASE

项目网站： [https : //www.thymeleaf.org](https://www.thymeleaf.org)

# 1介绍Thymeleaf

## 1.1什么是Thymeleaf?

Thymeleaf是一个适用于Web和独立环境的现代服务器端Java模板引擎，能够处理HTML，XML，JavaScript，CSS甚至纯文本。

Thymeleaf的主要目标是提供一种优雅且高度可维护的模板创建方式。为实现这一目标，它以自然模板的概念为基础，将其逻辑注入模板文件，其方式不会影响模板被用作设计原型。这改善了设计沟通，缩小了设计和开发团队之间的差距。

Thymeleaf也从一开始就设计了Web标准 - 特别是HTML5 - 允许您创建完全验证的模板，如果这是您的需要。

## 1.2 Thymeleaf过程可以使用哪种模板？

开箱即用，Thymeleaf允许您处理六种模板，每种模板称为模板模式：

- HTML
- XML
- 文本
- JAVASCRIPT
- CSS
- 生的

有两种标记模板模式（HTML 和 XML），三种文本模板模式（TEXT，JAVASCRIPT 和 CSS）和一种无操作模板模式（RAW）。

该HTML模板模式将允许任何类型的HTML的输入，包括HTML5，HTML4和XHTML。不会执行验证或格式良好检查，并且将在输出中尽可能地尊重模板代码/结构。

该XML模板模式将允许XML输入。在这种情况下，代码应该是格式良好的 - 没有未封闭的标签，没有不带引号的属性等 - 如果发现格式错误，解析器将抛出异常。请注意，不会执行任何验证（针对DTD或XML架构）。

该TEXT模板模式将允许非标记性质的模板使用特殊的语法。此类模板的示例可能是文本电子邮件或模板文档。请注意，HTML或XML模板也可以被处理TEXT，在这种情况下，它们不会被解析为标记，并且每个标记，DOCTYPE，注释等都将被视为纯文本。

该JAVASCRIPT模板模式将允许在Thymeleaf应用程序的JavaScript文件的处理。这意味着能够以与HTML文件相同的方式在JavaScript文件中使用模型数据，但是使用特定于JavaScript的集成，例如专门的转义或自然脚本。该JAVASCRIPT模板模式被认为是一种文本模式，因此使用相同的特殊语法的TEXT模板模式。

该CSS模板模式将允许参与Thymeleaf应用CSS文件的处理。与JAVASCRIPT模式类似，CSS模板模式也是文本模式，并使用TEXT模板模式中的特殊处理语法。

该RAW模板模式将根本不处理模板。它用于将未经处理的资源（文件，URL响应等）插入到正在处理的模板中。例如，HTML格式的外部非受控资源可以包含在应用程序模板中，安全地知道这些资源可能包含的任何Thymeleaf代码都不会被执行。

## 1.3方言：标准方言

标准方言：thymeleaf官方提供的一套标签库  
spring标准方言：spring为了整合thymeleaf而扩展thymeleaf标准方言

Thymeleaf是一个极易扩展的模板引擎（实际上它可以称为模板引擎框架），允许您定义和自定义模板处理的细节级别。

将一些逻辑应用于标记工件（标签，某些文本，注释或仅仅是占位符，如果模板不是标记）的对象称为处理器，这些处理器的集合 - 加上可能还有一些额外的工件 - 是什么一个方言通常是由。开箱即用，Thymeleaf的核心库提供了一种称为标准方言的方言，对大多数用户来说应该足够了。

请注意，方言实际上可以没有处理器，并且完全由其他类型的工件组成，但处理器绝对是最常见的用例。

本教程涵盖标准方言。您将在以下页面中了解的每个属性和语法功能都由此方言定义，即使没有明确提及。

当然，如果用户希望在利用库的高级功能的同时定义自己的处理逻辑，则可以创建自己的方言（甚至扩展标准方言）。

Thymeleaf也可以配置为一次使用多种方言。

官方的thymeleaf-spring3和thymeleaf-spring4集成包都定义了一种称为“SpringStandard Dialect”的方言，它与标准方言大致相同，但是为了更好地利用Spring Framework中的某些功能（例如），使用Spring Expression Language或SpringEL代替OGNL）。因此，如果您是Spring MVC用户，那么您不会浪费时间，因为您在此处学习的几乎所有内容都将在Spring应用程序中使用。

标准方言的大多数处理器都是属性处理器。这使得浏览器甚至可以在处理之前正确显示HTML模板文件，因为它们只会忽略其他属性。例如，虽然使用标记库的JSP可能包含不能由浏览器直接显示的代码片段，例如：

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

..... Thymeleaf标准方言将允许我们实现相同的功能：

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

这不仅可以被浏览器正确显示，而且还允许我们（可选地）在其中指定一个值属性（在这种情况下为“James Carrot”），当在浏览器中静态打开原型时将显示该属性，并且这将由 `${user.name}` 在处理模板期间评估得到的值代替。

这有助于您的设计人员和开发人员处理相同的模板文件，并减少将静态原型转换为工作模板文件所需的工作量。执行此操作的能力是称为自然模板的功能。

## 2 The Good Thymes虚拟杂货店

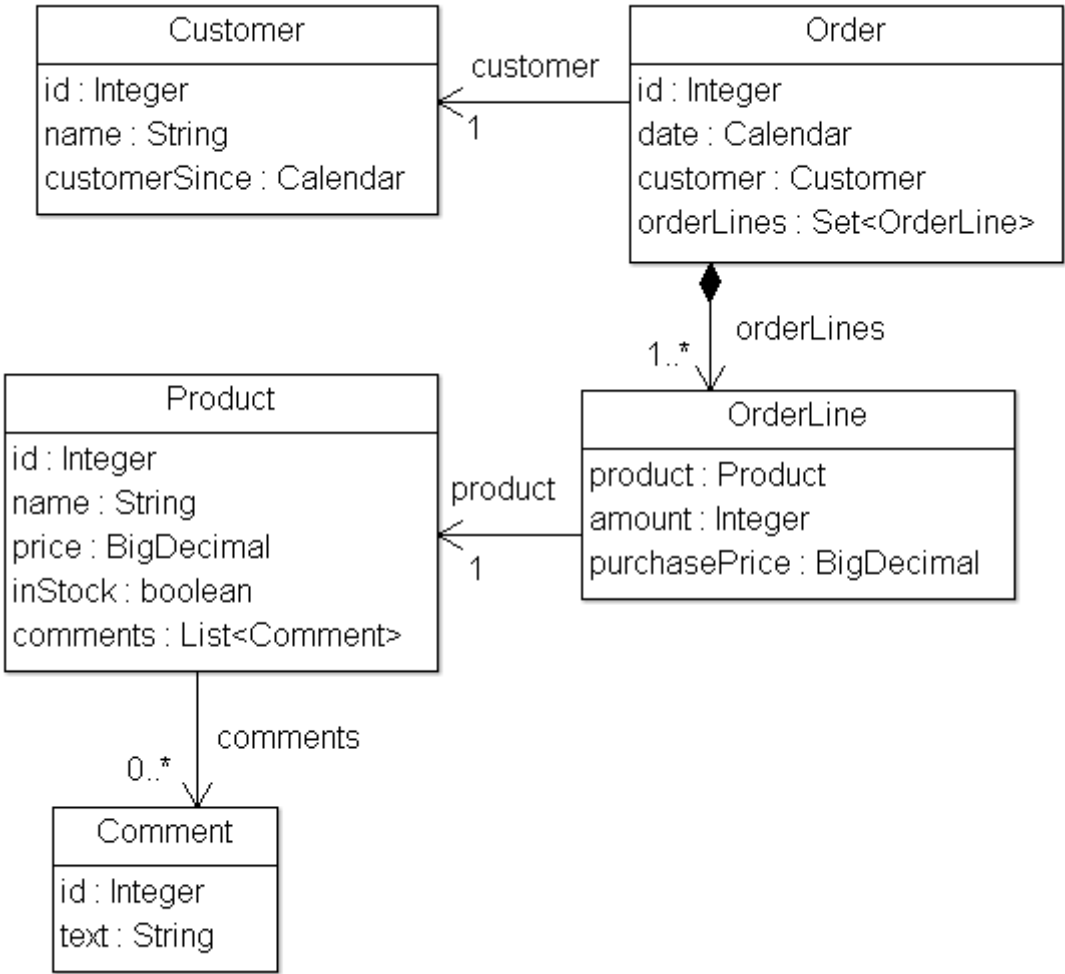
可以在[Good Thymes Virtual Grocery GitHub存储库](#)中找到本指南的本章和后续章节中显示的示例的源代码。

### 2.1 杂货店的网站

为了更好地解释使用Thymeleaf处理模板所涉及的概念，本教程将使用可从项目网站下载的演示应用程序。

这个应用程序是一个假想的虚拟杂货的网站，并将为我们提供许多场景来展示Thymeleaf的许多功能。

首先，我们需要一套简单的模型实体用于我们的应用程序：通过创建 **Products** 销售。我们还将管理这些： **Customers Orders Comments Products**



示例应用程序模型

我们的应用程序还有一个非常简单的服务层，由 **Service** 包含以下方法的对象组成：

```
public class ProductService {
    ...

    public List<Product> findAll() {
        return ProductRepository.getInstance().findAll();
    }

    public Product findById(Integer id) {
        return ProductRepository.getInstance().findById(id);
    }
}
```

}

}

在Web层，我们的应用程序将有一个过滤器，它将根据请求URL将执行委托给启用Thymeleaf的命令：

```
private boolean process(HttpServletRequest request, HttpServletResponse response)
    throws ServletException {

    try {

        // This prevents triggering engine executions for resource URLs
        if (request.getRequestURI().startsWith("/css") ||
            request.getRequestURI().startsWith("/images") ||
            request.getRequestURI().startsWith("/favicon")) {
            return false;
        }

        /*
         * Query controller/URL mapping and obtain the controller
         * that will process the request. If no controller is available,
         * return false and let other filters/servlets process the request.
         */
        IGTVGController controller = this.application.resolveControllerForRequest(request);
        if (controller == null) {
            return false;
        }

        /*
         * Obtain the TemplateEngine instance.
         */
        ITemplateEngine templateEngine = this.application.getTemplateEngine();

        /*
         * Write the response headers
         */
        response.setContentType("text/html;charset=UTF-8");
        response.setHeader("Pragma", "no-cache");
        response.setHeader("Cache-Control", "no-cache");
        response.setDateHeader("Expires", 0);

        /*
         * Execute the controller and process view template,
         * writing the results to the response writer.
         */
        controller.process(
            request, response, this.servletContext, templateEngine);

        return true;
    } catch (Exception e) {
        try {
            response.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
        } catch (final IOException ignored) {
            // Just ignore this
        }
        throw new ServletException(e);
    }
}
```

这是我们的 **IGTVGController** 界面：

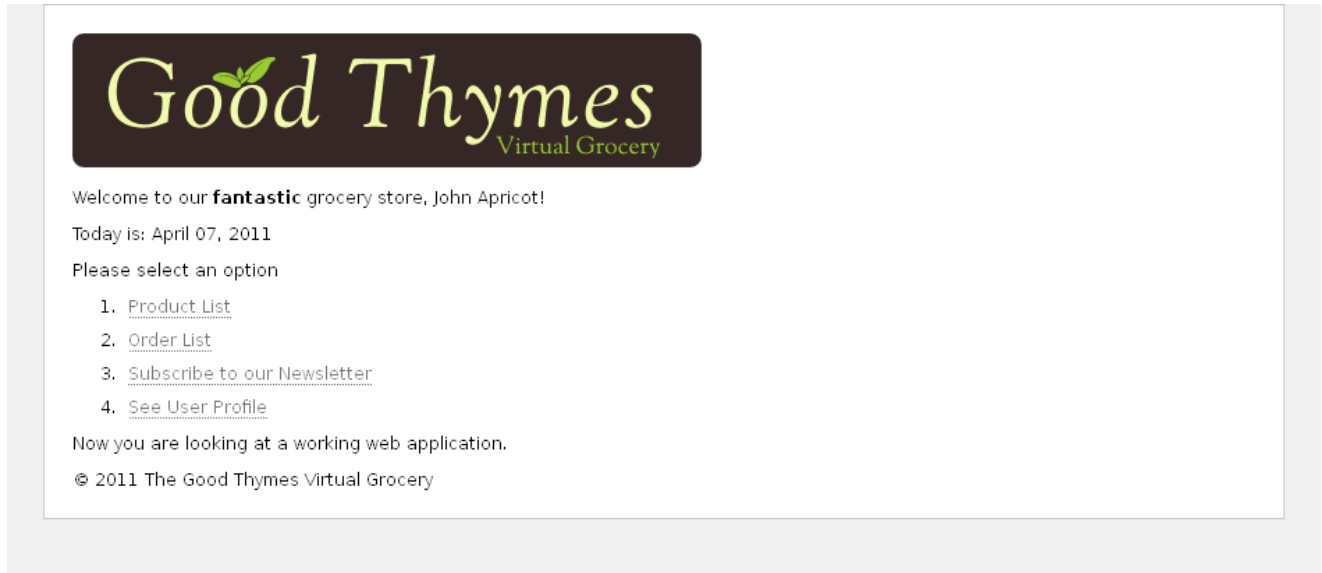
```
public interface IGTVGController {
```

```
public void process(
    HttpServletRequest request, HttpServletResponse response,
    ServletContext servletContext, ITemplateEngine templateEngine);

}
```

我们现在要做的就是创建 **IGTVGController** 接口的实现，从服务中检索数据并使用 **ITemplateEngine** 对象处理模板。

最后，它看起来像这样：



示例应用程序主页

但首先让我们看看该模板引擎是如何初始化的。

## 2.2创建和配置模板引擎

我们的过滤器中的 *process* (...) 方法包含以下行：

```
ITemplateEngine templateEngine = this.application.getTemplateEngine();
```

这意味着 *GTVGApplication* 类负责创建和配置Thymeleaf应用程序中最重要的对象之一：**TemplateEngine** 实例（**ITemplateEngine** 接口的实现）。

我们的 **org.thymeleaf.TemplateEngine** 对象初始化如下：

```
public class GTVGApplication {

    ...
    private final TemplateEngine templateEngine;
    ...

    public GTVGApplication(final ServletContext servletContext) {

        super();

        ServletContextTemplateResolver templateResolver =
            new ServletContextTemplateResolver(servletContext);

        // HTML is the default mode, but we set it anyway for better understanding of code
        templateResolver.setTemplateMode(TemplateMode.HTML);
        // This will convert "home" to "/WEB-INF/templates/home.html"
        templateResolver.setPrefix("/WEB-INF/templates/");
    }
}
```

```

templateResolver.setSuffix(".html");
// Template cache TTL=1h. If not set, entries would be cached until expelled
templateResolver.setCacheTTLs(Long.valueOf(3600000L));

// Cache is set to true by default. Set to false if you want templates to
// be automatically updated when modified.
templateResolver.setCacheable(true);

this.templateEngine = new TemplateEngine();
this.templateEngine.setTemplateResolver(templateResolver);

...
}

}

```

配置 **TemplateEngine** 对象有很多种方法，但是现在这几行代码将足以告诉我们所需的步骤。

## 模板解析器

---

让我们从模板解析器开始：

```

ServletContextTemplateResolver templateResolver =
    new ServletContextTemplateResolver(servletContext);

```

模板解析器是实现Thymeleaf API接口的对象，称为 **org.thymeleaf.templateresolver.ITemplateResolver**：

```

public interface ITemplateResolver {

    ...

    /*
     * Templates are resolved by their name (or content) and also (optionally) their
     * owner template in case we are trying to resolve a fragment for another template.
     * Will return null if template cannot be handled by this template resolver.
     */
    public TemplateResolution resolveTemplate(
        final IEngineConfiguration configuration,
        final String ownerTemplate, final String template,
        final Map<String, Object> templateResolutionAttributes);
}

```

这些对象负责确定我们的模板的访问方式，在这个GTVG应用程序

中， **org.thymeleaf.templateresolver.ServletContextTemplateResolver** 我们将从 *Servlet* 上下文中检索模板文件作为资源的方式：**javax.servlet.ServletContext** 每个Java Web应用程序中都存在的应用程序范围的对象，并从Web应用程序根解析资源。

但这并不是我们可以说的关于模板解析器的全部内容，因为我们可以为其上设置一些配置参数。一，模板模式：

```

templateResolver.setTemplateMode(TemplateMode.HTML);

```

**HTML**是默认的模板模式 **ServletContextTemplateResolver**，但最好还是建立它，以便我们的代码清楚地记录正在发生的事情。

```

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

```

该前缀和后缀修改，我们将传递到发动机获得要使用的真实资源名称的模板名称。

使用此配置，模板名称 *“product / list”* 将对应于：

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

（可选）通过 `cacheTTLMs` 属性在模板解析器中配置解析模板可以在缓存中存在的时间量：

```
templateResolver.setCacheTTLMs(3600000L);
```

如果达到最大高速缓存大小并且它是当前高速缓存的最旧条目，则在达到该TTL之前，模板仍然可以从高速缓存中排出。

用户可以通过实现 **ICacheManager** 接口或修改 **StandardCacheManager** 对象来管理默认缓存来定义缓存行为和大小。

关于模板解析器还有很多东西需要学习，但是现在来看看**Template Engine**对象的创建。

## 模板引擎

---

**Template Engine**对象是 **org.thymeleaf.ITemplateEngine** 接口的实现。其中一个实现是由Thymeleaf核心提供的：**org.thymeleaf.TemplateEngine** 我们在这里创建一个实例：

```
templateEngine = new TemplateEngine();  
templateEngine.setTemplateResolver(templateResolver);
```

相当简单，不是吗？我们所需要的只是创建一个实例并将模板解析器设置为它。

模板解析器是唯一需要的参数 **TemplateEngine**，尽管稍后将介绍许多其他参数（消息解析器，缓存大小等）。现在，这就是我们所需要的。

我们的模板引擎现已准备就绪，我们可以使用Thymeleaf开始创建我们的页面。



## 3使用文本

### 3.1多语言欢迎

我们的第一个任务是为我们的杂货网站创建一个主页。

这个页面的第一个版本非常简单：只是标题和欢迎信息。这是我们的 `/WEB-INF/templates/home.html` 文件：

```
<!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/gtvvg.css" th:href="@{/css/gtvvg.css}" />
  </head>

  <body>

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

  </body>

</html>
```

您将注意到的第一件事是该文件是HTML5，任何浏览器都可以正确显示它，因为它不包含任何非HTML标记（浏览器会忽略它们不理解的所有属性，例如 `th:text`）。

但是您可能还注意到此模板实际上不是有效的HTML5文档，因为 `th:*` HTML5规范不允许我们在表单中使用这些非标准属性。事实上，我们甚至 `xmlns:th` 在我们的 `<html>` 标签中添加了一个属性，绝对不是HTML5-ish：

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

在ide中不规范的html还会报错

...它在模板处理中根本没有任何影响，但是作为一个咒语，阻止我们的IDE抱怨缺少所有这些 `th:*` 属性的命名空间定义。

那么如果我们想让这个模板HTML5有效呢？简单：切换到Thymeleaf的数据属性语法，使用 `data-` 属性名称和hyphen（-）分隔符的前缀而不是分号（:）：

```
<!DOCTYPE html>

<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/gtvvg.css" data-th:href="@{/css/gtvvg.css}" />
  </head>

  <body>

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

  </body>

</html>
```



**data-** HTML5规范允许使用自定义前缀属性，因此，使用上面的代码，我们的模板将是有效的HTML5文档。

两种表示法都是完全等效且可互换的，但为了代码示例的简单性和紧凑性，本教程将使用名称空间表示法（**th:\***）。此外，**th:\*** 符号更通用，并且在每个Thymeleaf模板模式（XML，TEXT...）中**data-** 都允许使用，而符号仅允许在HTML模式中使用。

**data-**只能在html中使用，**th:\***在全部模板模式中都可以使用

## 使用**th:** 文本和外化文本

外化文本是从模板文件中提取模板代码的片段，以便它们可以保存在单独的文件（通常是**.properties** 文件）中，并且可以使用其他语言编写的等效文本（称为国际化或简称*i18n*）轻松替换它们。外化的文本片段通常称为“消息”。

消息始终具有标识它们的键，而Thymeleaf允许您指定文本应与具有以下**#{...}** 语法的特定消息对应：

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

我们在这里看到的实际上是Thymeleaf标准方言的两个不同特征：

- **替换标签中的文本内**  
该**th:text** 属性评估其值表达式并将结果设置为主机标签的主体，有效地替换了我们在代码中看到的“欢迎使用我们的杂货店！”文本。**读取模板文件中的键值**
- 的**#{home.welcome}** 表达，在指定的标准表达式语法，指示要由所使用的文本**th:text** 属性应与该消息**home.welcome** 对应于哪个语言环境，我们正在处理与模板键。

现在，这个外化文本在哪里？

Thymeleaf中外化文本的位置是完全可配置的，它取决于**org.thymeleaf.messageresolver.IMessageResolver** 所使用的具体实现。通常，**.properties** 将使用基于文件的实现，但是如果我们想要，例如，从数据库获取消息，我们可以创建自己的实现。

但是，我们在初始化期间没有为模板引擎指定消息解析器，这意味着我们的应用程序正在使用标准消息解析器，由**org.thymeleaf.messageresolver.StandardMessageResolver** 实现。

标准消息解析程序期望 **/WEB-INF/templates/home.html** 在同一文件夹中找到属性文件中的消息，并使用与模板相同的名称，例如：

**标准的模板解析器的属性文件会和模板文件在用同一个目录下**

- **/WEB-INF/templates/home\_en.properties** 用于英文文本。
- **/WEB-INF/templates/home\_es.properties** 西班牙语文本。
- **/WEB-INF/templates/home\_pt\_BR.properties** 用于葡萄牙语（巴西）语言文本。
- **/WEB-INF/templates/home.properties** 对于默认文本（如果区域设置不匹配）。

我们来看看我们的**home\_es.properties** 文件：

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

这就是我们将Thymeleaf流程作为模板所需的全部内容。让我们创建我们的Home控制器。

## 上下文

为了处理我们的模板，我们将创建一个**HomeController** 实现 **IGTVGController** 我们之前看到的接口的类：

```
public class HomeController implements IGTVGController {

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

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

```

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

```

我们看到的第一件事是创建一个上下文。Thymeleaf上下文是实现 **org.thymeleaf.context.IContext** 接口的对象。上下文应包含在变量映射中执行模板引擎所需的所有数据，并且还引用必须用于外部化消息的语言环境。

```

public interface IContext {

    public Locale getLocale();
    public boolean containsVariable(final String name);
    public Set<String> getVariableNames();
    public Object getVariable(final String name);

}

```

这个接口有一个专门的扩展，**org.thymeleaf.context.IWebContext** 用于基于ServletAPI的Web应用程序（如SpringMVC）。

```

public interface IWebContext extends IContext {

    public HttpServletRequest getRequest();
    public HttpServletResponse getResponse();
    public HttpSession getSession();
    public ServletContext getServletContext();

}

```

Thymeleaf核心库提供了以下每个接口的实现：

- **org.thymeleaf.context.Context** 实现 **IContext**
- **org.thymeleaf.context.WebContext** 实现 **IWebContext**

正如您在控制器代码中看到的那样，**WebContext** 是我们使用的那个。实际上我们必须这样做，因为使用 **ServletContextTemplateResolver** 要求我们使用上下文实现 **IWebContext**。

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

这四个构造函数参数中只有三个是必需的，因为如果没有指定系统，将使用系统的默认语言环境（尽管在实际应用程序中不应该发生这种情况）。

我们可以使用一些专门的表达式来从 **WebContext** 模板中获取请求参数以及请求，会话和应用程序属性。例如：

- **\${x}** 将返回 **x** 存储在Thymeleaf上下文中的变量或作为请求属性。
- **\${param.x}** 将返回一个被调用的请求参数 **x**（可能是多值的）。
- **\${session.x}** 将返回一个名为的会话属性 **x**。
- **\${application.x}** 将返回一个名为的 *servlet* 上下文属性 **x**。

## 执行模板引擎

准备好上下文对象后，现在我们可以告诉模板引擎使用上下文处理模板（通过其名称），并将响应编写器传递给它，以便可以将响应写入它：

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

让我们使用西班牙语语言环境查看结果：

```
<!DOCTYPE html>

<html>

<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>
```

## 3.2有关文本和变量的更多信息

### 未转义的文字

---

我们主页的最简单版本现在似乎已经准备就绪，但有一些我们没有想过的.....**如果我们有这样的消息怎么办？**

```
home.welcome=Welcome to our <b>fantastic</b> grocery store!
```

如果我们像以前一样执行此模板，我们将获得：

```
<p>Welcome to our &lt;b>fantastic</b> grocery store!</p>
```

这不完全符合我们的预期，因为我们的 **<b>** 标签已被转义，因此它将在浏览器中显示。

这是 **th:text** 属性的默认行为。如果我们希望Thymeleaf尊重我们的HTML标签而不是逃避它们，我们将不得不使用不同的属性：**th:utext** 对于“非转义文本”：

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

这将输出我们的消息，就像我们想要的那样：

```
<p>Welcome to our <b>fantastic</b> grocery store!</p>
```

### 使用和显示变量

---

现在让我们在主页上添加更多内容。例如，我们可能希望在欢迎消息下方显示日期，如下所示：

```
Welcome to our fantastic grocery store!
```

```
Today is: 12 july 2010
```

首先，我们必须修改控制器，以便将该日期添加为上下文变量：

```
public void process(
    final HttpServletRequest request, final HttpServletResponse response,
    final ServletContext servletContext, final ITemplateEngine templateEngine)
```

```

throws Exception {

    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());

}

```

我们添加了一个 **String** 调用 **today** 我们上下文的变量，现在我们可以显示它在模板中：

```

<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>

```

正如您所看到的，我们仍在使用 **th:text** 属性的属性（这是正确的，因为我们想要替换标签的主体），但这次语法有点不同而不是 **#{...}** 表达式值，我们使用的是 **\${...}** ← 这是一个变量表达式，它包含一个名为 **OGNL**（对象图导航语言）的语言表达式，该表达式将在我们之前讨论过的上下文变量映射上执行。

该 **\${today}** 表达式只是表示“今天拿到称为变量”，但这些表述可能更加复杂（如 **\${user.name}** “获取被叫用户的变量，并调用它的 **getName()** 方法”）。

属性值有很多可能性：消息，变量表达式.....还有很多。下一章将向我们展示所有这些可能性。

## 4标准表达式语法

我们将在杂货虚拟商店的开发中稍作休息，以了解Thymeleaf标准方言中最重要的部分之一：Thymeleaf标准表达式语法。

我们已经看到在这种语法中表达两种类型的有效属性值：消息和变量表达式：

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

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

但是有更多类型的表达式，以及更多有趣的细节来了解我们已经知道的。首先，让我们看一下标准表达式功能的快速摘要：

- 简单表达：

- 变量表达式： `${...}`
- 选择变量表达式： `*{...}`
- 消息表达式： `#{...}`
- 链接网址表达式： `@{...}`
- 片段表达式： `~{...}`

- 字面

- 文本文字： `'one text'` , `'Another one!'` , ...
- 号码文字： `0` , `34` , `3.0` , `12.3` , ...
- 布尔文字： `true` , `false`
- 空字面： `null`
- 文字标记： `one` , `sometext` , `main` , ...

- 文字操作：

- 字符串连接： `+`
- 文字替换： `|The name is ${name}|`

- 算术运算：

- 二元运算符： `+` , `-` , `*` , `/` , `%`
- 减号（一元运算符）： `-`

- 布尔运算：

- 二元运算符： `and` , `or`
- 布尔否定（一元运算符）： `!` , `not`

- 比较和相等：

- 比较： `>` , `<` , `>=` , `<=` ( `gt` , `lt` , `ge` , `le` )
- 平等运营商： `==` , `!=` ( `eq` , `ne` )

- 有条件的运营商：

- IF-THEN： `(if) ? (then)`
- IF-THEN-ELSE： `(if) ? (then) : (else)`
- 默认： `(value) ? : (defaultvalue)`

- 特殊代币：

- 无操作： `_`

所有这些功能都可以组合和嵌套：

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

## 4.1消息

我们已经知道，`#{...}` 消息表达式允许我们链接：

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

.....对此：

前面消息表达式是静态的，比如下面这个表达式，如果要使用动态的内容可以通过添加参数的方式

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

但是有一个方面我们还没有想到：如果消息文本不是完全静态会发生什么？例如，如果我们的应用程序知道谁是随时访问该网站的用户并且我们想通过名字问候他们怎么办？

静态文本

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

这意味着我们需要在消息中添加一个参数。像这样：

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

参数根据 java.text.MessageFormat 标准语法指定，这意味着您可以格式化为 `java.text.*` 包的类API文档中指定的数字和日期。

为了为我们的参数指定一个值，并给定一个被调用的HTTP会话属性 `user`，我们可以：

```
<p th:utext="#{home.welcome(${session.user.name})}">
  Welcome to our grocery store, Sebastian Pepper!
</p>
```

给参数传值

请注意，使用 `th:utext` 此处意味着格式化的消息不会被转义。此示例假定 `user.name` 已经转义。

可以指定几个参数，以逗号分隔。

消息密钥本身可以来自变量：

```
<p th:utext="#{${welcomeMsgKey}(${session.user.name})}">
  Welcome to our grocery store, Sebastian Pepper!
</p>
```

## 4.2变量

我们已经提到 `${...}` 表达式实际上是在上下文中包含的变量映射上执行的OGNL（对象 - 图形导航语言）表达式。

有关OGNL语法和功能的详细信息，请阅读 OGNL语言指南

在Spring支持MVC的应用程序中，OGNL将被SpringEL取代，但其语法与OGNL的语法非常相似（实际上，对于大多数常见情况来说，完全相同）。

从OGNL的语法，我们知道表达式：

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

.....实际上相当于：

```
ctx.getVariable("today");
```

但是OGNL允许我们创建更强大的表达式，这就是：

```
<p th:utext="#{home.welcome(${session.user.name})}">
  Welcome to our grocery store, Sebastian Pepper!
</p>
```

...通过执行以下命令获取用户名：

```
((User) ctx.getVariable("session").get("user")).getName();
```

但是getter方法导航只是OGNL的功能之一。让我们看看更多：

```
/*
 * Access to properties using the point (.). Equivalent to calling property getters.
 */
${person.father.name}

/*
 * Access to properties can also be made by using brackets ([]) and writing
 * the name of the property as a variable or between single quotes.
 */
${person['father']['name']}

/*
 * If the object is a map, both dot and bracket syntax will be equivalent to
 * executing a call on its get(...) method.
 */
${countriesByCode.ES}
${personsByName['Stephen Zucchini'].age}

/*
 * Indexed access to arrays or collections is also performed with brackets,
 * writing the index without quotes.
 */
${personsArray[0].name}

/*
 * Methods can be called, even with arguments.
 */
${person.createCompleteName()}
${person.createCompleteNameWithSeparator('-')}
```

## 表达式基本对象

在上下文变量上评估OGNL表达式时，某些对象可用于表达式以获得更高的灵活性。将从 # 符号开始引用这些对象（根据OGNL标准）：

- **#ctx**：上下文对象。
- **#vars**：上下文变量。
- **#locale**：上下文区域设置。
- **#request**：(仅限Web Contexts) **HttpServletRequest** 对象。
- **#response**：(仅限Web Contexts) **HttpServletResponse** 对象。



- **#session**:(仅限Web Contexts) **HttpSession** 对象。
- **#servletContext**:(仅限Web Contexts) **ServletContext** 对象。

所以我们可以这样做：

```
Established locale country: <span th:text="${#locale.country}">US</span>.
```

您可以在[附录A](#)中阅读这些对象的完整参考。

## Expression Utility对象 说白了就是一些工具类对象

除了这些基本对象，Thymeleaf还将为我们提供一组实用程序对象，帮助我们在表达式中执行常见任务。

- **#execInfo**：有关正在处理的模板的信息。
- **#messages**：在变量表达式中获取外化消息的方法，与使用 **#{...}** 语法获取的方法相同。
- **#uris**：转义部分URL / URI的方法
- **#conversions**：用于执行已配置的转换服务的方法（如果有）。
- **#dates**：**java.util.Date** 对象的方法：格式化，组件提取等。
- **#calendars**：类似于 **#dates**，但 **java.util.Calendar** 对象。
- **#numbers**：格式化数字对象的方法。
- **#strings**：**String** 对象的方法：contains, startsWith, prepending / appending等。
- **#objects**：一般的对象的方法。
- **#bools**：布尔评估的方法。
- **#arrays**：数组的方法。
- **#lists**：列表方法。
- **#sets**：集合的方法。
- **#maps**：地图的方法。
- **#aggregates**：在数组或集合上创建聚合的方法。
- **#ids**：处理可能重复的id属性的方法（例如，作为迭代的结果）。

您可以在[附录B](#)中查看每个实用程序对象提供的功能。

## 在我们的主页中重新格式化日期

现在我们了解这些实用程序对象，我们可以使用它们来改变我们在主页中显示日期的方式。而不是在我们这样做 **HomeController**：

```
SimpleDateFormat dateFormat = new SimpleDateFormat("dd MMMM yyyy");
Calendar cal = Calendar.getInstance();

WebContext ctx = new WebContext(request, servletContext, request.getLocale());
ctx.setVariable("today", dateFormat.format(cal.getTime()));

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

.....我们可以做到这一点：

```
WebContext ctx =
    new WebContext(request, response, servletContext, request.getLocale());
ctx.setVariable("today", Calendar.getInstance());

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

...然后在视图层本身中执行日期格式设置：

```
<p>
  Today is: <span th:text="{#calendars.format(today,'dd MMMM yyyy')}">13 May 2011</span>
</p>
```

### 4.3 选择表达式（星号语法）

变量表达式不仅可以写成 `${...}`，也可以作为 `*{...}`。

但是有一个重要的区别：星号语法评估所选对象上的表达式而不是整个上下文。也就是说，只要没有选定的对象，美元和星号语法就会完全相同。**如果没有选择对象，选择表达式和变量表达式完全相同**

**什么是选定的对象？使用 `th:object` 属性的表达式的结果。**我们在用户个人资料（`userprofile.html`）页面中使用一个：

```
<div th:object="{session.user}">
  <p>Name: <span th:text="{firstName}">Sebastian</span>.</p>
  <p>Surname: <span th:text="{lastName}">Pepper</span>.</p>
  <p>Nationality: <span th:text="{nationality}">Saturn</span>.</p>
</div>
```

这完全等同于：



```
<div>
  <p>Name: <span th:text="{session.user.firstName}">Sebastian</span>.</p>
  <p>Surname: <span th:text="{session.user.lastName}">Pepper</span>.</p>
  <p>Nationality: <span th:text="{session.user.nationality}">Saturn</span>.</p>
</div>
```

当然，美元和星号语法可以混合使用：

```
<div th:object="{session.user}">
  <p>Name: <span th:text="{*{firstName}">Sebastian</span>.</p>
  <p>Surname: <span th:text="{session.user.lastName}">Pepper</span>.</p>
  <p>Nationality: <span th:text="{*{nationality}">Saturn</span>.</p>
</div>
```

当对象选择到位时，所选对象也可用作美元表达式作为 `#object` 表达式变量：

```
<div th:object="{session.user}">
  <p>Name: <span th:text="{#{#object.firstName}">Sebastian</span>.</p>
  <p>Surname: <span th:text="{session.user.lastName}">Pepper</span>.</p>
  <p>Nationality: <span th:text="{*{nationality}">Saturn</span>.</p>
</div>
```

如上所述，如果没有执行任何对象选择，则美元和星号语法是等效的。

```
<div>
  <p>Name: <span th:text="{*{session.user.name}">Sebastian</span>.</p>
  <p>Surname: <span th:text="{*{session.user.surname}">Pepper</span>.</p>
  <p>Nationality: <span th:text="{*{session.user.nationality}">Saturn</span>.</p>
</div>
```

### 4.4 链接 URL

由于它们的重要性，URL 是 Web 应用程序模板中的一等公民，而 *Thymeleaf Standard Dialect* 具有特殊的语法，`@` 语法：`@{...}`

有不同类型的网址：

- 绝对网址：`http://www.thymeleaf.org`
- 相对URL，可以是：
  - 页面相对：`user/login.html` 比如：我的小说项目web根目录位localhost/novel，那么就
  - 上下文相关：(`/itemdetails?id=3` 服务器中的上下文名称将自动添加) 相对与它
  - 服务器相对：(`~/billing/processInvoice` 允许在同一服务器中的另一个上下文 (=应用程序) 中调用URL。 服务器根目录：localhost/
  - 协议相对URL：`//code.jquery.com/jquery-2.0.3.min.js` 省略了协议的路径，如：`www.baidu.com`，而不是`https://www.baidu.com`

这些表达式的实际处理及其转换为将要输出的URL是通过 `org.thymeleaf.linkbuilder.ILinkBuilder` 注册到 `ITemplateEngine` 正在使用的对象中的接口的实现来完成的。

默认情况下，此接口的单个实现是在类中注册的 `org.thymeleaf.linkbuilder.StandardLinkBuilder`，这对于基于Servlet API的脱机（非Web）和Web方案都是足够的。其他方案（例如与非Servlet API Web框架的集成）可能需要链接构建器接口的特定实现。

让我们使用这种新语法。符合 `th:href` 属性：

```
<!-- Will produce 'http://localhost:8080/gtvg/order/details?orderId=3' (plus rewriting) -->
<a href="details.html"
  th:href="@{http://localhost:8080/gtvg/order/details(orderId=${o.id})}">view</a>

<!-- Will produce '/gtvg/order/details?orderId=3' (plus rewriting) -->
<a href="details.html" th:href="@{/order/details(orderId=${o.id})}">view</a>

<!-- Will produce '/gtvg/order/3/details' (plus rewriting) -->
<a href="details.html" th:href="@{/order/{orderId}/details(orderId=${o.id})}">view</a>
```

这里要注意的一些事情：

- `th:href` 是一个修饰符属性：一旦处理，它将计算要使用的链接URL并将该值设置 `href` 为 `<a>` 标记的属性。
- 我们被允许对URL参数使用表达式（如您所见 `orderId=${o.id}`）。还将自动执行所需的URL参数编码操作。
- 如果需要几个参数，这些参数将用逗号分隔：`@{/order/process(execId=${execId},execType='FAST')}`
- URL路径中也允许使用变量模板：`@{/order/{orderId}/details(orderId=${orderId})}` 使用了变量模板参数不会最加到后面了
- 以 `/`（例如：）开头的相对URL `/order/details` 将自动以应用程序上下文名称为前缀。
- 如果未启用cookie或尚未知道cookie，则 `";jsessionid=..."` 可能会在相对URL中添加后缀，以便保留会话。这称为URL重写，Thymeleaf允许您通过 `response.encodeURL(...)` Servlet API为每个URL插入自己的重写过滤器。
- 该 `th:href` 属性允许我们（可选）`href` 在我们的模板中具有工作静态属性，以便我们的模板链接在直接打开以进行原型设计时仍可由浏览器导航。

与消息语法（`#{...}`）的情况一样，URL基数也可以是评估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>
```

## 我们主页的菜单

既然我们知道如何创建链接URL，那么在我们的主页中为网站中的其他一些页面添加一个小菜单呢？

```
<p>Please select an option</p>
<ol>
  <li><a href="product/list.html" th:href="@{/product/list}">Product List</a></li>
  <li><a href="order/list.html" th:href="@{/order/list}">Order List</a></li>
  <li><a href="subscribe.html" th:href="@{/subscribe}">Subscribe to our Newsletter</a></li>
  <li><a href="userprofile.html" th:href="@{/userprofile}">See User Profile</a></li>
</ol>
```

## 服务器根目录相对 *URL*

---

可以使用其他语法来创建服务器根相对（而不是上下文根相对）URL，以便链接到同一服务器中的不同上下文。这些URL将被指定为 `@{~/path/to/something}`

## 4.5 片段

片段表达式是表示标记片段并将其移动到模板周围的简单方法。这允许我们复制它们，将它们作为参数传递给其他模板，等等。

最常见的用途是使用 **th:insert** 或进行片段插入 **th:replace**（在后面的部分中将详细介绍）：

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

但它们可以在任何地方使用，就像任何其他变量一样：

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

本教程后面有一整节专门介绍模板布局，包括对片段表达式的更深入解释。

## 4.6 文字

### 文字文字

---

文本文字只是在单引号之间指定的字符串。它们可以包含任何字符，但您应该使用它们中的任何单引号 `\'`。

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

### 数字文字

---

数字文字只是：数字。

```
<p>The year is <span th:text="2013">1492</span>.</p>
<p>In two years, it will be <span th:text="2013 + 2">1494</span>.</p>
```

### 布尔文字

---

布尔文字是 **true** 和 **false**。例如：

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

在这个例子中，`== false` 被写在大括号外面，因此是Thymeleaf来处理它。如果它是在大括号内写的，那么它将由OGNL / SpringEL引擎负责：

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

## *null*文字

---

该 `null` 文本也可用于：

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

## 文字代币

---

Numeric, boolean and null literals are in fact a particular case of *literal tokens*.

These tokens allow a little bit of simplification in Standard Expressions. They work exactly the same as text literals ( `'...'` ), but they only allow letters ( `A-Z` and `a-z` ), numbers ( `0-9` ), brackets ( `[` and `]` ), dots ( `.` ), hyphens ( `-` ) and underscores ( `_` ). So no whitespaces, no commas, etc.

The nice part? Tokens don't need any quotes surrounding them. So we can do this:

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

instead of:

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

## 4.7 Appending texts 字符串连接

Texts, no matter whether they are literals or the result of evaluating variable or message expressions, can be easily appended using the `+` operator:

```
<span th:text="'The name of the user is ' + ${user.name}">>
```

## 4.8 Literal substitutions 文字替换

Literal substitutions allow for an easy formatting of strings containing values from variables without the need to append literals with `'...' + '...'`.

These substitutions must be surrounded by vertical bars ( `|` ), like:

```
<span th:text="|Welcome to our application, ${user.name}!|">
```

Which is equivalent to: 文字替换：字符串中在两个竖线中的表达式可以直接解析，类似php中的双引号解析变量一样

```
<span th:text="'Welcome to our application, ' + ${user.name} + '!'>
```

Literal substitutions can be combined with other types of expressions:

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

Only variable/message expressions ( `${...}`, `*{...}`, `#{...}` ) are allowed inside `[...]` literal substitutions. No other literals ( `'...'` ), boolean/numeric tokens, conditional expressions etc. are.

## 4.9 Arithmetic operations 算术运算

Some arithmetic operations are also available: `+`, `-`, `*`, `/` and `%`.

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

请注意，这些运算符也可以应用于OGNL变量表达式本身（在这种情况下，将由OGNL而不是Thymeleaf标准表达式引擎执行）：

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

请注意，其中一些运算符存在文本别名：`div`（`/`），`mod`（`%`）。

## 4.10 比较器和平等

在表达式中的值可以与进行比较 `>`，`<`，`>=` 和 `<=` 符号，以及 `==` 和 `!=` 运营商可以被用来检查是否相等（或缺乏）。请注意，XML确定不应在属性值中使用 `<` 和 `>` 符号，因此应将它们替换为 `&lt;` 和 `&gt;`。

```
<div th:if="${prodStat.count} &gt; 1">
<span th:text="'Execution mode is ' + ( (${execMode} == 'dev')? 'Development' : 'Production')">
```

更简单的替代方法可能是使用某些运算符存在的文本别名：`gt`（`>`），`lt`（`<`），`ge`（`>=`），`le`（`<=`），`not`（`!`）。还 `eq`（`==`），`neq` / `ne`（`!=`）。

## 4.11 条件表达式 就是 ? :

条件表达式仅用于评估两个表达式中的一个，具体取决于评估条件的结果（它本身就是另一个表达式）。

让我们来看一个例子片段（引入另一个属性修改器，`th:class`）：

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

条件表达式（`condition`，`then` 和 `else`）的所有三个部分本身都是表达式，这意味着它们可以是变量（`${...}`，`*{...}`），消息（`#{...}`），URL（`@{...}`）或文字（`'...'`）。

条件表达式也可以使用括号嵌套：

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

其他表达式也可以省略，在这种情况下，如果条件为`false`，则返回`null`值：

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

## 4.12 默认表达式（*Elvis*运算符）

默认表达式就是，表达式?:默认值

表达式为null，使用默认值，这里表达式只判断是否为null而不是布尔值

一个默认的表情是一种特殊的条件值的没有那么一部分。它等同于某些语言（如Groovy）中存在的*Elvis*运算符，允许您指定两个表达式：如果它不计算为null，则使用第一个表达式，但如果确实如此，则使用第二个表达式。

让我们用户在用户个人资料页面中看到它：

```
<div th:object="${session.user}">
  ...
  <p>Age: <span th:text="*{age}?: '(no age specified)'">27</span>.</p>
</div>
```

正如您所看到的，运算符是 **?:**，并且我们在此处使用它来指定名称的默认值（在这种情况下为文字值），仅当评估结果 **\*{age}** 为null时。因此，这相当于：

```
<p>Age: <span th:text="*{age != null}? *{age} : '(no age specified)'">27</span>.</p>
```

与条件值一样，它们可以包含括号之间的嵌套表达式：

```
<p>
  Name:
  <span th:text="*{firstName}?: (*{admin}? 'Admin' : #{default.username})">Sebastian</span>
</p>
```

## 4.13 无操作令牌

等价于，什么都不做的意思

No-Operation标记由下划线符号（**\_**）表示。

这个标记背后的想法是指定表达式的期望结果是什么都不做，即完全就像可处理属性（例如 **th:text**）根本不存在一样。

除了其他可能性之外，这允许开发人员将原型文本用作默认值。例如，而不是：

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

...我们可以直接使用“无用户身份验证”作为原型文本，从设计的角度来看，这会使代码更简洁，更通用：

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

## 4.14 数据转换/格式化

Thymeleaf 为 **variable()** 和 **selection()** 表达式定义了一个双括号语法，允许我们通过配置的转换服务应用数据转换。 **#{...} \*{...}**

它基本上是这样的：

```
<td th:text="${#{user.lastAccessDate}}">...</td>
```

注意到那里的双支撑？：**#{...}**。这指示Thymeleaf将 **user.lastAccessDate** 表达式的结果传递给转换服务，并要求它在写入结果之前执行格式化操作（转换为 **String**）。

假设它 `user.lastAccessDate` 是类型 `java.util.Calendar`，如果已经注册了转换服务（实现 `IStandardConversionService`）并且包含有效的转换 `Calendar -> String`，则将应用它。

`IStandardConversionService`（`StandardConversionService` 类）的默认实现只是 `.toString()` 在转换为的任何对象上执行 `String`。有关如何注册自定义转换服务实现的更多信息，请查看“[更多配置](#)”部分。

官方 `thymeleaf-spring3` 和 `thymeleaf-spring4` 集成软件包的透明集成了 Spring 自己 Thymeleaf 的转换服务机制转换服务的基础设施，所以在 Spring 配置宣称，转换服务和格式化将进行自动获得 `${...}` 和 `*{...}` 表达。

## 4.15 预处理

除了用于表达式处理的所有这些功能外，Thymeleaf 还具有预处理表达式的功能。

预处理是在正常表达式之前完成的表达式的执行，它允许修改最终将被执行的表达式。

预处理表达式与普通表达式完全相同，但显示为双下划线符号（如 `__${expression}__`）。

让我们假设我们有一个 `Messages_fr.properties` 包含 OGNL 表达式的 18 条目，该表达式调用特定于语言的静态方法，如：

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

.....和 `Messages_es.properties` equivalent :

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

我们可以创建一个标记片段，根据语言环境评估一个表达式或另一个表达式。为此，我们将首先选择表达式（通过预处理），然后让 Thymeleaf 执行它：

```
<p th:text="${__#{article.text('textVar')}}__">Some text here...</p>
```

请注意，法语区域设置的预处理步骤将创建以下等效项：

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

`__` 可以使用在属性中对预处理字符串进行转义 `\\_\\_`。



## 5 设置属性值

本章将解释我们在标记中设置（或修改）属性值的方式。

### 5.1 设置任何属性的值

Say our website publishes a newsletter, and we want our users to be able to subscribe to it, so we create a `/WEB-INF/templates/subscribe.html` template with a form:

```
<form action="subscribe.html">
  <fieldset>
    <input type="text" name="email" />
    <input type="submit" value="Subscribe!" />
  </fieldset>
</form>
```

As with Thymeleaf, this template starts off more like a static prototype than it does a template for a web application. First, the **action** attribute in our form statically links to the template file itself, so that there is no place for useful URL rewriting. Second, the **value** attribute in the submit button makes it display a text in English, but we'd like it to be internationalized.

使用这个来设置动态的属性

Enter then the **th:attr** attribute, and its ability to change the value of attributes of the tags it is set in:

```
<form action="subscribe.html" th:attr="action=@{/subscribe}">
  <fieldset>
    <input type="text" name="email" />
    <input type="submit" value="Subscribe!" th:attr="value=#{subscribe.submit}"/>
  </fieldset>
</form>
```

这个概念非常简单：**th:attr** 只需要一个为属性赋值的表达式。创建了相应的控制器和消息文件后，处理该文件的结果将是：

```
<form action="/gtvg/subscribe">
  <fieldset>
    <input type="text" name="email" />
    <input type="submit" value="¡Suscríbete!" />
  </fieldset>
</form>
```

除了新的属性值之外，您还可以看到 `applicacion` 上下文名称已自动添加到 URL 基础中 `/gtvg/subscribe`，如前一章所述。

但是，如果我们想一次设置多个属性呢？XML 规则不允许您在标记中设置两次属性，因此 **th:attr** 将采用以逗号分隔的分配列表，例如：

```

```

给定所需的消息文件，这将输出：

```

```

### 5.2 为特定属性设置值

到现在为止，您可能会想到以下内容：

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

.....是一个非常丑陋的标记。在属性值中指定赋值可能非常实用，但如果您必须始终执行此操作，则它不是创建模板的最佳方式。

Thymeleaf同意你的意见，这就是 **th:attr** 模板中几乎没有使用的原因。通常，您将使用 **th:\*** 其任务设置特定标记属性的其他属性（而不仅仅是任何属性 **th:attr**）。

例如，要设置 **value** 属性，请使用 **th:value**：

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

这看起来好多了！让我们尝试 **action** 对 **form** 标记中的属性执行相同操作：

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

你还记得 **th:href** 我们 **home.html** 之前放过的东西吗？它们正是同样的属性：

```
<li><a href="product/list.html" th:href="@{/product/list}">Product List</a></li>
```

有很多这样的属性，每个属性都针对特定的HTML5属性：

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:online
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:xml:lang	th:xmlspace

### 5.3 一次设置多个值

有两个叫比较特殊的属性 **th:alt-title** 和 **th:lang-xml:lang** 可用于同时设置两个属性相同的值。特别：

- **th:alt-title** 将设置 **alt** 和 **title**。
- **th:lang-xml:lang** 将设置 **lang** 和 **xml:lang**。

对于我们的GTVG主页，这将允许我们替换：

```

```

.....或者这个，相当于：

```

```

...有了这个：

```

```

## 5.4附加和预先

Thymeleaf还提供了 **th:attrappend** 和 **th:attrprepend** 属性，它们将评估结果附加（后缀）或前置（前缀）到现有属性值。

例如，您可能希望将要添加的CSS类的名称（未设置，仅添加）存储到上下文变量中的某个按钮，因为要使用的特定CSS类将取决于用户执行的操作。之前：

对属性的值进行前，后最加内容

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

如果您在 **cssStyle** 变量设置为的情况下处理此模板 **"warning"**，您将获得：

```
<input type="button" value="Do it!" class="btn warning" />
```

标准方言中还有两个特定的附加属性：**th:classappend** 和 **th:styleappend** 属性，用于向元素添加CSS类或样式片段而不覆盖现有元素：

类追加和样式片段追加

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

（不要担心该 **th:each** 属性。它是一个迭代属性，我们稍后会讨论它。）

## 5.5固定值布尔属性

HTML has the concept of *boolean attributes*, attributes that have no value and the prescence of one means that value is "true". In XHTML, these attributes take just 1 value, which is itself.

For example, **checked** :

html中的布尔属性，可以使用专门的标签指定，只需要传入布尔值表示就可以

```
<input type="checkbox" name="option2" checked /> <!-- HTML -->
<input type="checkbox" name="option1" checked="checked" /> <!-- XHTML -->
<input type="radio" name="boy" th:checked="true"> girl
```

The Standard Dialect includes attributes that allow you to set these attributes by evaluating a condition, so that if evaluated to true, the attribute will be set to its fixed value, and if evaluated to false, the attribute will not be set:

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

The following fixed-value boolean attributes exist in the Standard Dialect:

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

## 5.6 Setting the value of any attribute (default attribute processor)

### 设置任意属性的值

Thymeleaf提供了一个默认属性处理器，允许我们设置任何属性的值，即使 **th:\*** 在标准方言中没有为它定义特定的处理器。

所以类似于：

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

将导致：

```
<span whatever="John Apricot">...</span>
```

## 5.7支持HTML5友好的属性和元素名称

也可以使用完全不同的语法以更加HTML5友好的方式将处理器应用于模板。

```
<table>
  <tr data-th-each="user : ${users}">
    <td data-th-text="${user.login}">...</td>
    <td data-th-text="${user.name}">...</td>
  </tr>
</table>
```

该 **data-{prefix}-{name}** 语法编写自定义属性在HTML5中，而无需开发人员使用任何命名空间的名称，如标准的方式 **th:\***。Thymeleaf使所有方言（不仅是标准方言）自动使用此语法。

还有一种语法来指定自定义标签：**{prefix}-{name}**，它遵循W3C自定义元素规范（较大的W3C Web组件规范的一部分）。例如，这可以用于 **th:block** 元素（或者也可以 **th-block**），这将在后面的部分中解释。

重要提示：此语法是对命名空间语法的补充 **th:\***，它不会替换它。完全没有意图在将来弃用命名空间语法。

# 6迭代

到目前为止，我们已经创建了一个主页，一个用户个人资料页面以及一个允许用户订阅我们的新闻通讯的页面.....但是我们的产品呢？为此，我们需要一种方法来迭代集合中的项目以构建我们的产品页面。

## 6.1迭代基础知识

要在我们的 `/WEB-INF/templates/product/list.html` 页面中显示产品，我们将使用表格。我们的每个产品都会连续显示（一个 `<tr>` 元素），因此对于我们的模板，我们需要创建一个模板行 - 一个可以说明我们希望如何显示每个产品的模板行 - 然后指示 Thymeleaf 重复它，每个产品一次。

标准方言为我们提供了一个属性：`th:each`。

### 使用 `th: each`

对于我们的产品列表页面，我们需要一个控制器方法，从服务层检索产品列表并将其添加到模板上下文中：

```
public void process(
    final HttpServletRequest request, final HttpServletResponse response,
    final ServletContext servletContext, final 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());
}
```

然后我们将 `th:each` 在我们的模板中使用迭代产品列表：

```
<!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/gtvvg.css" th:href="@{/css/gtvvg.css}" />
</head>

<body>

<h1>Product list</h1>

<table>
<tr>
<th>NAME</th>
<th>PRICE</th>
<th>IN STOCK</th>
</tr>
<tr th:each="prod : ${prods}">
<td th:text="${prod.name}">Onions</td>
<td th:text="${prod.price}">2.41</td>
<td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
```

```

    </tr>
</table>

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

</body>

</html>

```

That **prod : \${prods}** attribute value you see above means “for each element in the result of evaluating **\${prods}**, repeat this fragment of template, using the current element in a variable called **prod**”. Let’s give a name each of the things we see:

- We will call **\${prods}** the *iterated expression* or *iterated variable*.
- We will call **prod** the *iteration variable* or simply *iter variable*.

Note that the **prod** iter variable is scoped to the **<tr>** element, which means it is available to inner tags like **<td>**.

## Iterable values

The **java.util.List** class isn’t the only value that can be used for iteration in Thymeleaf. There is a quite complete set of objects that are considered *iterable* by a **th:each** attribute:

- Any object implementing **java.util.Iterable**
- Any object implementing **java.util.Enumeration**.
- Any object implementing **java.util.Iterator**, whose values will be used as they are returned by the iterator, without the need to cache all values in memory.
- Any object implementing **java.util.Map**. When iterating maps, iter variables will be of class **java.util.Map.Entry**.
- Any array.
- Any other object will be treated as if it were a single-valued list containing the object itself.

除了list还有以下对象可以迭代

## 6.2 Keeping iteration status

When using **th:each**, Thymeleaf offers a mechanism useful for keeping track of the status of your iteration: the *status variable*.

Status variables are defined within a **th:each** attribute and contain the following data:

- The current *iteration index*, starting with 0. This is the **index** property. 当前索引 从0开始
- The current *iteration index*, starting with 1. This is the **count** property. 当前索引从1开始
- The total amount of elements in the iterated variable. This is the **size** property. 元素总个数
- The *iter variable* for each iteration. This is the **current** property. 当前迭代的对象
- Whether the current iteration is even or odd. These are the **even/odd** boolean properties. 是奇数还是偶数
- Whether the current iteration is the first one. This is the **first** boolean property. 是第一个
- Whether the current iteration is the last one. This is the **last** boolean property. 是最后一个

让我们看看我们如何在前面的例子中使用它：

```

<table th:each="u,s : ${list}">
  <p th:text="| index:${s.index}|"></p>
  <p th:text="| current:${s.current.name} -- ${s.current.age}|"></p>
  <p th:text="| even:${s.even}|"></p>
  <p th:text="| odd:${s.odd}|"></p>
  <p th:text="| first:${s.first}|"></p>
  <p th:text="| last:${s.last}|"></p>
  <p th:text="| size:${s.size}|"></p>
  <tr>
    <td th:text="${u.name}"></td>
    <td th:text="${u.age}"></td>
  </tr>
</table>

<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
  </tr>
  <tr th:each="prod,iterStat : ${prods}" th:class="${iterStat.odd} - odd">
    <td th:text="${prod.name}">Onions</td>
  </tr>
</table>

```

```

<td th:text="${prod.price}">2.41</td>
<td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
</tr>
</table>

```

状态变量（**iterStat** 在此示例中）在 **th:each** 属性中通过在**iter**变量本身之后写入其名称来定义，用逗号分隔。就像**iter**变量一样，状态变量的范围也限定为由包含该 **th:each** 属性的标记定义的代码片段。

我们来看看处理模板的结果：

```

<!DOCTYPE html>

<html>

<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>

<h1>Product list</h1>

<table>
<tr>
<th>NAME</th>
<th>PRICE</th>
<th>IN STOCK</th>
</tr>
<tr class="odd">
<td>Fresh Sweet Basil</td>
<td>4.99</td>
<td>yes</td>
</tr>
<tr>
<td>Italian Tomato</td>
<td>1.25</td>
<td>no</td>
</tr>
<tr class="odd">
<td>Yellow Bell Pepper</td>
<td>2.50</td>
<td>yes</td>
</tr>
<tr>
<td>Old Cheddar</td>
<td>18.75</td>
<td>yes</td>
</tr>
</table>

<p>
<a href="/gtvg/" shape="rect">Return to home</a>
</p>

</body>

</html>

```

请注意，我们的迭代状态变量已经完美地工作，**odd** 仅将CSS类建立到奇数行。

如果您没有显式设置状态变量，Thymeleaf将始终通过后缀 **Stat** 为迭代变量的名称为您创建一个：

```

<table>
<tr>

```



```

<th>NAME</th>
<th>PRICE</th>
<th>IN STOCK</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>

```

## 6.3通过延迟检索数据进行优化

有时我们可能希望优化数据集合的检索（例如，从数据库中），以便只有在真正使用它们时才会检索这些集合。

实际上，这可以应用于任何数据片段，但考虑到内存中集合可能具有的大小，检索要迭代的集合是此方案的最常见情况。

为了支持这一点，Thymeleaf提供了一种懒惰加载上下文变量的机制。实现 **ILazyContextVariable** 接口的上下文变量- 最有可能通过扩展其 **LazyContextVariable** 默认实现 - 将在执行时解决。例如：

```

context.setVariable(
    "users",
    new LazyContextVariable<List<User>>() {
        @Override
        protected List<User> loadValue() {
            return databaseRepository.findAllUsers();
        }
    });

```

可以在不知道其惰性的情况下使用此变量，例如：

```

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

```

但与此同时，**loadValue()** 如果在代码中 **condition** 进行求值，则永远不会被初始化（它的方法永远不会被调用） **false**：

```

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

```

## 7 条件评估

### 7.1 简单条件：“if”和“除非”

有时，如果满足某个条件，您将需要模板的片段才会出现在结果中。

例如，假设我们希望在产品表中显示一行，其中包含每个产品的评论数量，如果有任何评论，则指向该产品的评论详细信息页面的链接。

为此，我们将使用以下 **th:if** 属性：

```
<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=${prod.id})}"
      th:if="${not #lists.isEmpty(prod.comments)}">view</a>
  </td>
</tr>
</table>
```

这里有很多东西要看，所以让我们关注重要的一点：

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

这将创建指向评论页面（带有URL `/product/comments`）的链接，其 `prodId` 参数设置为 `id` 产品的参数，但仅限于产品有任何评论。

我们来看看生成的标记：

```
<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>

```

完善！这正是我们想要的。

请注意，该 **th:if** 属性不仅会评估布尔条件。它的功能稍微超出了它，它将按照 **true** 以下规则评估指定的表达式：

- 如果value不为null:
  - 如果value是布尔值，则为 **true**。
  - 如果value是数字且不为零
  - 如果value是一个字符且不为零
  - 如果value是String并且不是“false”，“off”或“no”
  - 如果value不是布尔值，数字，字符或字符串。
- （如果value为null，则th: if将计算为false）。

Also, **th:if** has an inverse attribute, **th:unless**, which we could have used in the previous example instead of using a **not** inside the OGNL expression:

```

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

```

## 7.2 Switch statements

There is also a way to display content conditionally using the equivalent of a *switch* structure in Java: the **th:switch** / **th:case** attribute set.

```

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

```

Note that as soon as one **th:case** attribute is evaluated as **true**, every other **th:case** attribute in the same switch context is evaluated as **false**.

The default option is specified as **th:case="\*"**：默认default值

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

## 8 Template Layout

### 8.1 Including template fragments

#### Defining and referencing fragments

---

In our templates, we will often want to include parts from other templates, parts like footers, headers, menus...

在我们的模板中，我们经常需要包含其他模板中的部分，页脚，标题，菜单等部分.....

为了做到这一点，Thymeleaf需要我们定义这些部分，“片段”，以便包含，这可以使用 **th:fragment** 属性来完成。

假设我们要在所有杂货页面上添加标准版权页脚，因此我们创建一个 **/WEB-INF/templates/footer.html** 包含以下代码的文件：

```
<!DOCTYPE html>

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

  <body>

    <div th:fragment="copy">
      &copy; 2011 The Good Thymes Virtual Grocery
    </div>

  </body>

</html>
```

上面的代码定义了一个名为的片段 **copy**，我们可以使用其中一个 **th:insert** 或 **th:replace** 属性轻松地我们的主页中包含这些片段（并且 **th:include**，尽管自Thymeleaf 3.0以来不再推荐使用它）：

```
<body>

  ...

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

</body>
```

请注意，**th:insert** 需要一个片段表达式（**~{...}**），它是一个导致片段的表达式。在上面的例子中，这是一个非复杂的片段表达式，（**~{, }**）封闭是完全可选的，所以上面的代码相当于：

```
<body>

  ...

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

</body>
```

#### 片段规范语法

---

片段表达式的语法非常简单。选择器，可以选择片段，也可以当css选择器用来进行选择  
有三种不同的格式：

- **"~{templatename::selector}"** 包括在名为的模板上应用指定标记选择器而产生的片段 **templatename**。请注意，**selector** 可以仅仅是一个片段的名称，所以你可以指定为简单的东西 **~{templatename::fragmentname}** 就像在 **~{footer :: copy}** 上面。**引入模板中的某个片段**

标记选择器语法由底层的AttoParser解析库定义，类似于XPath表达式或CSS选择器。有关详细信息，请参阅[附录C](#)。

- `~{templatename}` 包含名为的完整模板 `templatename` 。 **引入整个模板**

请注意，您在 `th:insert` / `th:replace` tags中使用的模板名称必须由模板引擎当前使用的模板解析器解析。

- `~{::selector}` 或 `~{this::selector}` 插入来自同一模板的片段，进行匹配 `selector` 。如果在表达式出现的模板上找不到，则模板调用（插入）的堆栈将遍历最初处理的模板（根），直到 `selector` 在某个级别匹配。 **引入当前模板中的某个片段**

双方 `templatename` 并 `selector` 在上面的例子可以是全功能的表达式（甚至条件语句！），如：

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

再次注意周围的 `~{...}` 包络在 `th:insert` / 中是如何可选的 `th:replace` 。

片段可以包含任何 `th:*` 属性。一旦将片段包含在目标模板（具有 `th:insert` / `th:replace` attribute的模板）中，就会评估这些属性，并且它们将能够引用此目标模板中定义的任何上下文变量。

这种片段方法的一大优点是，您可以在浏览器完全可显示的页面中编写片段，具有完整且有效的标记结构，同时仍保留使Thymeleaf将其包含在其他模板中的能力。

## 没有引用片段 `th:fragment`

由于标记选择器的强大功能，我们可以包含不使用任何 `th:fragment` 属性的片段。它甚至可以是来自不同应用程序的标记代码，完全不了解Thymeleaf：

```
...
<div id="copy-section">
  &copy; 2011 The Good Thymes Virtual Grocery
</div>
...
```

我们可以使用上面的片段，通过其 `id` 属性简单地引用它，与CSS选择器类似：

```
<body>

...

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

</body>
```

## `th:insert`和`th:replace`（和`th:include`）之间的区别

和之间有什么区别 `th:insert` 和 `th:replace` （和 `th:include` ，因为3.0不推荐）？

- **`th:insert`** 是最简单的：它只是插入指定的片段作为其主机标签的主体。
- **`th:replace`** 实际上用指定的片段替换它的主机标签。
- **`th:include`** 类似于 `th:insert` ，但不是插入片段，它只插入此片段的内容。

所以像这样的HTML片段：

```
<footer th:fragment="copy">
  &copy; 2011 The Good Thymes Virtual Grocery
</footer>
```

...在主机 **<div>** 标签中包含三次，如下所示：

```
<body>

...

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

</body>
```

.....将导致：

```
<body>

...

<div>
  <footer>
    &copy; 2011 The Good Thymes Virtual Grocery
  </footer>
</div>

<div>
  &copy; 2011 The Good Thymes Virtual Grocery
</div>

</body>
```

把片段插入进入

把原来的元素替换为片段

把片段中的内容插入

## 8.2可参数化的片段签名

为了为模板片段创建更像函数的机制，使用定义的片段 **th:fragment** 可以指定一组参数：

```
<div th:fragment="frag (onevar,twovar)">
  <p th:text="${onevar} + ' - ' + ${twovar}">...</p>
</div>
```

这需要使用这两种语法之一来从 **th:insert** 或调用片段 **th:replace**：

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

给片段传入参数

两种传递参数的语法

请注意，在最后一个选项中，顺序并不重要：

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

## 片段局部变量没有片段参数

即使片段定义没有这样的参数：

```
<div th:fragment="frag">
    ...
</div>
```

我们可以使用上面指定的第二种语法来调用它们（只有第二种语法）：

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

这将相当于组合 **th:replace** 和 **th:with**：

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

请注意，片段的局部变量的这种规范 - 无论它是否具有参数签名 - 都不会导致在执行之前清空上下文。片段仍然可以访问调用模板中使用的每个上下文变量，就像它们当前一样。

## th: 断言 *in-template*断言

该 **th:assert** 属性可以指定一个以逗号分隔的表达式列表，这些表达式应该被评估并为每次评估生成true，否则会引发异常。

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

这对于验证片段签名的参数非常方便：

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

## 8.3灵活的布局：仅仅是片段插入

由于片段表达式，我们可以为不是文本，数字，**bean**对象的片段指定参数.....而是指定标记片段。 **把片段作为参数进行传递**

这允许我们以一种方式创建我们的片段，使得它们可以通过来自调用模板的标记来丰富，从而产生非常灵活的模板布局机制。

注意在下面的片段中使用 **title** 和 **links** 变量：

```
<head th:fragment="common_header(title,links)">

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

    <!-- Common styles and scripts -->
    <link rel="stylesheet" type="text/css" media="all" th:href="@{/css/awesomeapp.css}">
    <link rel="shortcut icon" th:href="@{/images/favicon.ico}">
    <script type="text/javascript" th:src="@{/sh/scripts/codebase.js}"></script>

    <!--/* Per-page placeholder for additional links */-->
    <th:block th:replace="${links}" />

</head>
```

我们现在可以将这个片段称为：



```
...
<head th:replace="base :: common_header(~{::title},~{::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>
...
```

...结果将使用我们的调用模板中的实际 `<title>` 和 `<link>` 标签作为 `title` 和 `links` 变量的值，从而导致我们的片段在插入过程中自定义：

```
...
<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>
...
```

## 使用空片段

一个特殊的片段表达式，即空片段（`~{ }`），可用于指定无标记。使用前面的示例：

```
<head th:replace="base :: common_header(~{::title},~{ })">

    <title>Awesome - Main</title>

</head>
...
```

注意fragment（`links`）的第二个参数是如何设置为空片段的，因此没有为 `<th:block th:replace="${links}" />` 块写入任何内容：

↓  
整个元素给使用空片段进行替换

```
...
<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>
...
```

## 使用无操作令牌

如果我们只想让我们的片段使用其当前标记作为默认值，则`no-op`也可以用作片段的参数。再次，使用 `common_header` 示例：

```
...
<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>
...
```

无操作标记，传递过去后，片段的标记不执行，会使用，元素默认值  
比如title参数为无标记，那么会使用片段的title默认值

和签名空片段区别在于空片段还是片段，会用片段的title元素还是会用空片段来替换

看看如何将 `title` 参数（`common_header` 片段的第一个参数）设置为`no-op`（`_`），这导致片段的这一部分根本不被执行（`title` = 无操作）：

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

使用了元素的内容作为默认值

结果是：

```
...
<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>
...
```

## 高级条件插入片段

`empty`片段和无操作令牌的可用性允许我们以非常简单和优雅的方式执行片段的条件插入。

例如，我们可以这样做，以便仅在用户是管理员时插入我们的 `common :: adminhead` 片段，并且如果不是，则不插入任何内容（`empty`片段）：

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

此外，我们可以使用无操作令牌，以便仅在满足指定条件时插入片段，但如果不满足条件则保留标记而不进行修改：

```
...
<div th:insert="${user.isAdmin()} ? ~{common :: adminhead} : _">
    Welcome [[${user.name}]], click <a th:href="@{/support}">here</a> for help-desk support.
</div>
...
```

另外，如果我们已经配置了模板解析器来检查模板资源是否存在 - 通过它们的 `checkExistence` 标志 - 我们可以使用片段本身的存在作为默认操作中的条件：

```

...
<!-- 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 删除模板片段

回到示例应用程序，让我们重新访问我们的产品列表模板的最新版本：

```

<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>

```

这段代码作为一个模板很好，但作为一个静态页面（当浏览器直接打开而没有Thymeleaf处理它时）它就不会成为一个好的原型。

为什么？因为虽然浏览器可以完全显示，但该表只有一行，而且这行包含模拟数据。作为原型，它看起来不够逼真.....我们应该有多个产品，我们需要更多行。

所以让我们添加一些：

```

<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>

```

好的，现在我们有三个，对原型来说肯定更好。但是.....当我们用Thymeleaf处理它时会发生什么？：

```

<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>

```

最后两行是模拟行！嗯，当然它们是：迭代仅适用于第一行，所以没有理由为什么Thymeleaf应该删除其他两个。

我们需要一种在模板处理过程中删除这两行的方法。让我们 **th:remove** 在第二个和第三个 **<tr>** 标签上使用该属性：

```

<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>

```

处理完毕后，所有内容都将按原样重复：

```

<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>

```

**all** 属性中的这个值是什么意思？**th:remove** 可以根据其价值以五种不同的方式表现：

- **all**：删除包含标记及其所有子标记。
- **body**：不要删除包含标记，但删除其所有子标记。
- **tag**：删除包含标记，但不删除其子项。
- **all-but-first**：除第一个子项外，删除包含标记的所有子项。
- **none**：没做什么。此值对于动态评估很有用。

这个 **all-but-first** 价值有什么用呢？它将让我们 **th:remove="all"** 在原型设计时节省一些：

```

<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>
  </tbody>
</table>

```

```

<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>

```

该 **th:remove** 属性可采取任何 *Thymeleaf* 标准表示，因为它返回允许字符串值中的一个，只要（**all**，**tag**，**body**，**all-but-first** 或 **none**）。

这意味着删除可能是有条件的，例如：

```

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

```

另请注意，**th:remove** 考虑 **null** 到同义词 **none**，因此以下工作方式与上面的示例相同：

```

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

```

在这种情况下，如果 **condition** 为 **false**，**null** 将返回，因此不会执行删除。

## 8.5 布局继承

为了能够将单个文件作为布局，可以使用片段。具有 **title** 和 **content** 使用 **th:fragment** 和的简单布局示例 **th:replace**：

```

<!DOCTYPE html>
<html th:fragment="layout (title, content)" xmlns:th="http://www.thymeleaf.org">
<head>
  <title th:replace="${title}">Layout Title</title>
</head>
<body>
  <h1>Layout H1</h1>
  <div th:replace="${content}">
    <p>Layout content</p>
  </div>
  <footer>
    Layout footer
  </footer>
</body>
</html>

```

此示例声明了一个名为 **layout** 的片段，其中 **title** 和 **content** 作为参数。在下面的示例中，两者都将在页面上替换，并通过提供的片段表达式继承它。

```

<!DOCTYPE html>
<html th:replace="~{layoutFile :: layout(~{::title}, ~{::section})}">
<head>
  <title>Page Title</title>
</head>
<body>
<section>
  <p>Page content</p>
  <div>Included on page</div>
</section>
</body>
</html>

```

在这个文件中，该 **html** 标签将被替换的布局，但在布局 **title** 和 **content** 将已被替换 **title**，并 **section** 分别块。

如果需要，布局可以由多个片段组成页眉和页脚。



## 9 局部变量

Thymeleaf将局部变量称为模板的特定片段定义的变量，并且仅可用于在该片段内进行评估。

我们已经看到的一个例子是 **prod** 我们的产品列表页面中的**iter**变量：

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

该 **prod** 变量仅在 **<tr>** 标记的范围内可用。特别：

- 它将可用于 **th:\*** 在该标记中执行的任何其他属性，其优先级低于 **th:each**（这意味着它们将在之后执行 **th:each**）。
- 它将可用于 **<tr>** 标记的任何子 **<td>** 元素，例如何元素。

Thymeleaf为您提供了一种使用 **th:with** 属性声明局部变量而无需迭代的方法，其语法类似于属性值赋值：

```
<div th:with="firstPer=${persons[0]}">
    <p>
        The name of the first person is <span th:text="${firstPer.name}">Julius Caesar</span>.
    </p>
</div>
```

当 **th:with** 被处理时，该 **firstPer** 变量被创建为一个局部变量，并加入到变量映射从上下文来，使得它可用于评估与在上下文中声明的任何其它变量一起，但仅在含有的边界 **<div>** 标记。

您可以使用通常的多重赋值语法同时定义多个变量：

```
<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>
```

该 **th:with** 属性允许重用在同一属性中定义的变量：

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

我们在Grocery的主页上使用它！还记得我们为输出格式化日期而编写的代码吗？

```
<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** :

那么，如果我们想要**"dd MMMM yyyy"**实际依赖于语言环境怎么办？例如，我们可能希望将以下消息添加到我们的**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:

现在，让我们使用`th:with`将本地化的日期格式转换为变量，然后在`th:text`表达式中使用它：

```
<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:

那简洁干净。事实上，鉴于这一事实`th:with`具有较高的`precedence`比`th:text`，我们可以解决这一切的`span`标签

```
<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.

你可能在想：优先权？我们还没有谈过这个！好吧，不要担心，因为这正是下一章的内容

# 10 Attribute Precedence

## 属性优先级

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

**th:\***在同一个标 签中写入多个属性会发生什么？例如

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

我们希望该 **th:each** 属性在之前执行， **th:text** 以便我们得到我们想要的结果，但是考虑到HTML / XML标准没有给标签中的属性写入的顺序赋予任何意义，优先级必须在属性本身中建立机制，以确保这将按预期工作。

因此，所有Thymeleaf属性都定义了一个数字优先级，它确定了它们在标记中执行的顺序。这个顺序是：

订购	特征	属性
1	片段包含	th:insert th:replace
2	片段迭代	th:each
3	有条件的评估	th:if th:unless th:switch th:case
4	局部变量定义	th:object th:with
五	一般属性修改	th:attr th:attrprepend th:attrappend
6	具体属性修改	th:value th:href th:src ...
7	文字（标签正文修改）	th:text th:utext
8	片段规范	th:fragment
9	片段删除	th:remove

这个优先级机制意味着如果属性位置被反转，上面的迭代片段将给出完全相同的结果（尽管它的可读性稍差）：

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

# 11 评论和块

## 11.1. 标准HTML/XML注释

标准HTML/XML注释 `<!-- ... -->` 可以在Thymeleaf模板中的任何位置使用。Thymeleaf将不会处理这些评论中的任何内容，并将逐字复制到结果中：

```
<!-- User info follows -->
<div th:text="${...}">
  ...
</div>
```

## 11.2. Thymeleaf解析器级注释块

解析器级注释块是在Thymeleaf解析它时将简单地从模板中删除的代码。它们看起来像这样：

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

Thymeleaf将删除一切与 `<!--/*` 和 `*/-->`，所以这些注释块也可以用于显示当模板是静态开放代码，知道当Thymeleaf处理它，它都将被删除：

```
<!--/*-->
<div>
  you can see me only before Thymeleaf processes me!
</div>
<!--/*-->
```

注释代码块，最后生成的html代码中，注释的代码会删除

在html静态文件中不会生效，变成thymeleaf模板解析后，变成注释

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>
```

## 11.3. 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="${...}">
```

在html静态文件中它是注释代码块，在thymeleaf模板解析后，会删除注释，即进行反注释

```

    ...
  </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.

## 11.4. 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:

就是一个元素块，

```

<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!

## 12 Inlining

### 12.1 Expression inlining

虽然标准方言允许我们使用标签属性几乎完成所有操作，但在某些情况下我们可能更喜欢将表达式直接写入HTML文本。例如，我们更喜欢这样写：

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

.....而不是这个：

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

在Thymeleaf 之间表达 `[[...]]` 或被 `[(...)]` 认为是内联表达式，在其中我们可以使用任何类型的表达式，这些表达式在一个 `th:text` 或 `th:utext` 属性中也是有效的。

请注意，虽然 `[[...]]` 对应于 `th:text`（即结果将被HTML转义），但是 `[(...)]` 对应于 `th:utext` 并且不会执行任何HTML转义。所以对于一个变量，如 `msg = 'This is <b>great!</b>'` 给定这个片段：

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

结果将使这些 `<b>` 标签不转义，因此：

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

而如果像以下一样逃脱：

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

结果将被HTML转义：

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

请注意，默认情况下，文本内联在标记中的每个标记的主体中都是活动的 - 而不是标记本身 - 因此我们无需执行任何操作即可启用它。

#### 内联 vs 自然模板

---

如果你来自其他模板引擎，其中这种输出文本的方式是常态，你可能会问：为什么我们从一开始就不这样做？它的代码少于所有这些 `th:text` 属性！

好吧，小心那里，因为虽然你可能会发现内联非常有趣，但你应该永远记住，当你静态打开它们时，内联表达式将逐字显示在你的HTML文件中，所以你可能无法将它们用作设计原型了！

浏览器静态显示我们的代码片段而不使用内联的区别...

```
Hello, Sebastian!
```

.....并使用它.....

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

.....在设计实用性方面非常清楚。

## 禁用内联

但是，可以禁用此机制，因为实际上可能存在我们确实希望输出 `[[...]]` 或 `[...]` 序列而不将其内容作为表达式处理的情况。为此，我们将使用 `th:inline="none"`：

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

这将导致：

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

## 12.2 文字内联

文本内联与我们刚刚看到的表达内联功能非常相似，但它实际上增加了更多功能。它必须明确启用 `th:inline="text"`。

文本内联不仅允许我们使用我们刚才看到的相同内联表达式，而且实际上处理标签主体就好像它们是在 **TEXT** 模板模式下处理的模板一样，这允许我们执行基于文本的模板逻辑（不仅仅是输出表达式）。

我们将在下一章中看到有关文本模板模式的更多信息。


## 12.3 JavaScript内联

JavaScript内联允许 `<script>` 在 **HTML** 模板模式下处理的模板中更好地集成JavaScript 块。

与文本内联一样，这实际上相当于处理脚本内容，就好像它们是 **JAVASCRIPT** 模板模式中的模板一样，因此文本模板模式的所有功能（见下一章）都将在眼前。但是，在本节中，我们将重点介绍如何使用它将Thymeleaf表达式的输出添加到JavaScript块中。

必须使用 `th:inline="javascript"` 以下方式明确启用此模式：

```
<script th:inline="javascript">
...
var username = [[${session.user.name}]];
...
</script>
```



会进行转义

这将导致：

```
<script th:inline="javascript">
...
var username = "Sebastian \"Fruity\" Applejuice";
...
</script>
```

上面代码中需要注意的两件重要事项：

首先，JavaScript内联不仅会输出所需的文本，而且还会用引号和JavaScript来包含它 - 转义其内容，以便将表达式结果输出为格式良好的JavaScript文字。

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>
```

不进行转义

The result would look like:

```
<script th:inline="javascript">
...
var username = Sebastian "Fruity" Applejuice;
...
</script>
```

可以使用不转义的方式输出js代码，而非字符串

...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 JavaScript自然模板

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>
```

在模板中会执行这里进行解析

在静态文件中会只用这里

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



关于JavaScript内联的一个重要注意事项是，这种表达式评估是智能的，不仅限于字符串。Thymeleaf将在JavaScript语法中正确编写以下类型的对象：

- 字符串
- 数字
- 布尔
- 数组
- 集合
- 地图
- Bean（具有`getter`和`setter`方法的对象）

例如，如果我们有以下代码：

```
<script th:inline="javascript">
    ...
    var user = /*[[${session.user}]]*/ null;
    ...
</script>
```

该`${session.user}`表达式将评估为一个 **User** 对象，Thymeleaf将正确地将其转换为JavaScript语法：

```
<script th:inline="javascript">
    ...
    var user = {"age":null,"firstName":"John","lastName":"Apricot",
                "name":"John Apricot","nationality":"Antarctica"};
    ...
</script>
```

这种JavaScript序列化的方式是通过 `org.thymeleaf.standard.serializer.IStandardJavaScriptSerializer` 接口的实现，可以 **StandardDialect** 在模板引擎使用的实例上配置。

此JS序列化机制的默认实现将在类路径中查找[Jackson库](#)，如果存在，将使用它。如果没有，它将应用内置的序列化机制，涵盖大多数场景的需求并产生类似的结果（但不太灵活）。

## 12.4 CSS内联

Thymeleaf还允许在CSS `<style>` 标签中使用内联，例如：

```
<style th:inline="css">
    ...
</style>
```

例如，假设我们将两个变量设置为两个不同的 **String** 值：

```
classname = 'main elems'
align = 'center'
```

我们可以像以下一样使用它们：

```
<style th:inline="css">
    .[[${classname}]] {
        text-align: [[${align}]];
    }
</style>
```

结果将是：

```
<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>
```

## 13 Textual template modes

### 13.1 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\u00E9" + " " + 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-unesescaped*.
- Attributes in **JAVASCRIPT** template mode will be *JavaScript-unesescaped*.
- Attributes in **CSS** template mode will be *CSS-unesescaped*.

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

```
[# th:if="${120<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*.

## 14 Some more pages for our grocery

现在我们对使用Thymeleaf了解很多，我们可以在我们的网站上添加一些新页面来进行订单管理。

请注意，我们将重点关注HTML代码，但如果要查看相应的控制器，可以查看捆绑的源代码。

### 14.1 订单清单

让我们从创建订单列表页面开始 `/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/gtvvg.css" th:href="@{/css/gtvvg.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>
```

这里没有什么可以让我们感到惊讶，除了这一点OGNL魔法：

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

这样做，对于顺序中的每个订单行（`OrderLine` 对象），将其 `purchasePrice` 和 `amount` 属性相乘（通过调用相应的 `getPurchasePrice()` 和 `getAmount()` 方法）并将结果返回到数字列表中，稍后由 `#aggregates.sum(...)` 函数聚合以获得订单总数价钱。



你必须喜欢OGNL的力量。

## 14.2 订单详情

现在，对于订单详细信息页面，我们将在其中大量使用星号语法：

```
<!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/gtvvg.css" th:href="@{/css/gtvvg.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>
```

除了这个嵌套对象选择之外，这里没什么新东西：

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

    ...

    <div th:object="*{customer}">
        <p><b>Name:</b> <span th:text="*{name}">Frederic Tomato</span></p>
        ...
    </div>

    ...
</body>
```

.....这 `*{name}` 相当于：

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

# 15有关配置的更多信息

## 15.1模板解析器

对于Good Thymes Virtual Grocery，我们选择了一个 **ITemplateResolver** 名为的实现 **ServletContextTemplateResolver**，它允许我们从Servlet Context获取模板作为资源。

除了让我们能够通过实现 **ITemplateResolver**，Thymeleaf 创建我们自己的模板解析器包括四个开箱即用的实现：

- **org.thymeleaf.templateresolver.ClassLoaderTemplateResolver**，将模板解析为类加载器资源，如：

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

- **org.thymeleaf.templateresolver.FileTemplateResolver**，将模板解析为文件系统中的文件，如：

```
return new FileInputStream(new File(template));
```

- **org.thymeleaf.templateresolver.UrlTemplateResolver**，将模板解析为URL（甚至是非本地的），如：

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

- **org.thymeleaf.templateresolver.StringTemplateResolver**，它可以直接解析模板，因为它 **String** 被指定为 **template**（或模板名称，在这种情况下显然不仅仅是一个名称）：

```
return new StringReader(templateName);
```

所有预绑定的实现都 **ITemplateResolver** 允许相同的配置参数集，包括：

- 前缀和后缀（已经看到）：

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

- 允许使用与文件名不直接对应的模板名称的模板别名。如果同时存在后缀/前缀和别名，则将在前缀/后缀之前应用别名：

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

- 读取模板时要应用的编码：

```
templateResolver.setEncoding("UTF-8");
```

- 要使用的模板模式：

```
// Default is HTML
templateResolver.setTemplateMode("XML");
```

- 模板缓存的默认模式，以及用于定义特定模板是否可缓存的模式：

```
// 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 to exceed the cache max size (oldest entry will be removed).

```
// Default is no TTL (only cache size exceeded 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);
```

当应用多个模板解析器时，建议为每个模板解析器指定模式，以便Thymeleaf可以快速丢弃那些不打算解析模板的模板解析器，从而提高性能。这样做不是必要条件，而是建议：

```
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));
```

如果未指定这些可解析的模式，我们将依赖于 **ITemplateResolver** 我们正在使用的每个实现的特定功能。请注意，并非所有实现都可以在解析之前确定模板的存在，因此可以始终将模板视为可解析并破坏解析链（不允许其他解析器检查相同的模板），但随后无法阅读真实的资源。

**ITemplateResolver** 核心Thymeleaf中包含的所有实现都包含一种机制，允许我们在解析可解析之前让解析器真正检查资源是否存在。这是旗帜，其作用如下：**checkExistence**

```
ClassLoaderTemplateResolver classLoaderTemplateResolver = new ClassLoaderTemplateResolver();
classLoaderTemplateResolver.setOrder(Integer.valueOf(1));
classLoaderTemplateResolver.setCheckExistence(true);
```

此 **checkExistence** 标志强制解析器在解析阶段执行资源存在的实际检查（如果存在检查返回false，则调用链中的后续解析器）。虽然这在每种情况下听起来都不错，但在大多数情况下，这意味着对资源本身的双重访问（一次用于检查存在，另一次用于读取它），并且在某些情况下可能是性能问题，例如基于远程URL模板资源 - 一个潜在的性能问题，无论如何都可以通过使用模板缓存来大大减轻（在这种情况下，模板只会在第一次访问时解析）。

## 15.2消息解析器

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
```

## 16 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.

这一切都导致了这样的想法：在不浪费大量内存的情况下缓存Web应用程序中最常用的模板是可行的，并且它还将节省大量时间，这些时间将花费在一小组文件上的输入/输出操作上事实上，这永远不会改变。

我们如何控制这个缓存？首先，我们之前已经了解到，我们可以在模板解析器中启用或禁用它，甚至只对特定模板执行操作：

```
// Default is true
templateResolver.setCacheable(false);
templateResolver.getCacheablePatternSpec().addPattern("/users/*");
```

此外，我们可以通过建立自己的缓存管理器对象来修改其配置，该对象可以是默认 **StandardCacheManager** 实现的实例：

```
// Default is 200
StandardCacheManager cacheManager = new StandardCacheManager();
cacheManager.setTemplateCacheMaxSize(100);
...
templateEngine.setCacheManager(cacheManager);
```

**org.thymeleaf.cache.StandardCacheManager** 有关配置缓存的更多信息，请参阅javadoc API。

可以从模板缓存中手动删除条目：

```
// Clear the cache completely
templateEngine.clearTemplateCache();

// Clear a specific template from the cache
templateEngine.clearTemplateCacheFor("/users/userList");
```

# 17解耦模板逻辑

## 17.1解耦逻辑：概念

到目前为止，我们已经为我们的Grocery Store工作，模板以通常的方式完成，逻辑以属性的形式插入到我们的模板中。

但Thymeleaf也让我们彻底脱钩从逻辑模板标记，允许创建完全逻辑较少标记模板在 HTML 和 XML 模板模式。

主要思想是模板逻辑将在单独的逻辑文件中定义（更确切地说是逻辑资源，因为它不需要是文件）。默认情况下，该逻辑资源将是与模板文件位于同一位置（例如文件夹）的附加文件，具有相同的名称但具有 **.th.xml** 扩展名：

```
/templates
+-->/home.html
+-->/home.th.xml
```

因此该 **home.html** 文件可以完全无逻辑。它可能看起来像这样：

```
<!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>
```

绝对没有Thymeleaf代码。这是一个模板文件，没有Thymeleaf或模板知识的设计师可以创建，编辑和/或理解。或者由某些外部系统提供的HTML片段，根本没有Thymeleaf挂钩。

现在让我们 **home.html** 通过创建我们的附加 **home.th.xml** 文件将该模板转换为Thymeleaf模板：

```
<?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>
```

在这里，我们可以 **<attr>** 在 **thlogic** 块内看到很多标签。这些 **<attr>** 标签通过其属性选择在原始模板的节点上执行属性注入，这些 **sel** 属性包含Thymeleaf 标记选择器（实际上是`AttoParser`标记选择器）。

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.

## 17.2 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.

## 17.3 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 ( `~{...}` ).

## 17.4 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 sequence 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*.

## 17.5 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?

- 首先，它将 `a prefix` 和 `a` 应用于模板资源 `suffix` 的基本名称（通过其 `ITemplateResource#getBaseName()` 方法获得）。前缀和后缀都可以配置，默认情况下，前缀为空，后缀为 `.th.xml`。
- 其次，它要求模板资源通过其方法解析具有计算名称的相对资源 `ITemplateResource#relative(String relativeLocation)`。

`IDecoupledTemplateLogicResolver` 可以 `TemplateEngine` 轻松配置要使用的具体实现：

```
final StandardDecoupledTemplateLogicResolver decoupledresolver =
    new StandardDecoupledTemplateLogicResolver();
decoupledresolver.setPrefix("../viewlogic/");
```

```
...  
templateEngine.setDecoupledTemplateLogicResolver(decoupledResolver);
```

## 18附录A：表达式基本对象

始终可以调用某些对象和变量映射。我们来看看他们：

### 基础对象

- **#ctx**: 上下文对象。实施 `org.thymeleaf.context.IContext` 或 `org.thymeleaf.context.IWebContext` 依赖于我们的环境（独立或Web）。

注意 **#vars** 并且 **#root** 是同一对象的同义词，但 **#ctx** 建议使用。

```
/*
 * =====
 * 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**: 直接访问 `java.util.Locale` 与当前请求关联的。

```
${#locale}
```

请求/会话属性的 **Web** 上下文命名空间等。

在Web环境中使用Thymeleaf时，我们可以使用一系列快捷方式来访问请求参数，会话属性和应用程序属性：

请注意，这些不是上下文对象，而是作为变量添加到上下文中的映射，因此我们不使用它们 **#**。在某种程度上，它们充当命名空间。

- **param**: 用于检索请求参数。 `${param.foo}` 是一个 `String[]` 带有 `foo` 请求参数值的，因此 `${param.foo[0]}` 通常用于获取第一个值。

```
/*
 * =====
 * 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 attribute '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 attribute '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** : 直接访问 `javax.servlet.ServletContext` 与当前请求关联的对象。

```
${#servletContext.getAttribute('foo')}\n${#servletContext.contextPath}\n...
```

## 19附录B：表达式实用程序对象

### 执行信息

---

- **#execInfo**: 表达式对象，提供有关在Thymeleaf标准表达式中处理的模板的有用信息。

```

/*
 * =====
 * 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}

```

### 消息

---

- **#messages**: 用于在变量表达式中获取外部化消息的实用程序方法，与使用 **#{...}** 语法获取它们的方式相同。

```

/*
 * =====
 * 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)}

```

## 的 **URI**/网址

---

- **#uris**: 用于在Thymeleaf标准表达式中执行URI / URL操作（尤其是转义/转义）的实用程序对象。

```

/*
* =====
* 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)}

```

## 转换

---

- **#conversions:** 允许在模板的任何位置执行转换服务的实用程序对象：

```

/*
 * =====
 * 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)}

```

## 日期

---

- **#dates:** `java.util.Date` 对象的实用方法：

```

/*
 * =====
 * 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()}

```

## 日历

- **#calendars**: 类似于 **#dates**，但对于 **java.util.Calendar** 对象:

```

/*
 * =====
 * 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()}

```

## 数字

- **#numbers:** 数字对象的实用方法:

```

/*
 * =====
 * 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)}

```

## 字符串

---

- **#strings:** `String` 对象的实用方法:

```

/*
 * =====
 * 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)}

```

## 对象

---

- **#objects:** 一般对象的实用程序方法

```

    /*
    * =====
    * 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)}

```

## 布尔

---

- **#bools:** 布尔评估的实用程序方法

```

    /*
    * =====
    * 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)}

```

## 数组

---

- **#arrays:** 数组的实用程序方法

```

/*
 * =====
 * 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)}

```

## 清单

---

- **#lists:** 列表的实用程序方法

```

/*
 * =====
 * 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)}

```

## 集

---

- **#sets:** 集合的实用程序方法

```

/*
 * =====
 * 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)}

```

## 地图

---

- **#maps:** 地图的实用程序方法

```

    /*
    * =====
    * 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)}

```

## 骨料

---

- **#aggregates:** 用于在数组或集合上创建聚合的实用程序方法

```

    /*
    * =====
    * 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)}

```

## 标识

---

- **#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 of 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')}

```

## 20 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` 表示当前节点的名为x的直接子节点。
- `//x` 表示任意深度的名为x的当前节点的子节点。
- `x[@z="v"]` 表示名称为x的元素和名为z且值为“v”的属性。
- `x[@z1="v1" and @z2="v2"]` 表示名称为x的元素，属性z1和z2分别为值“v1”和“v2”。
- `x[i]` 表示名称为x的元素，位于其兄弟姐妹中的数字i中。
- `x[@z="v"][i]` 表示名称为x的元素，属性z的值为“v”，并且在其兄弟姐妹中的数字i中也与该条件匹配。

但也可以使用更简洁的语法：

- `x` 完全等同于 `//x`（`x` 在任何深度级别搜索具有名称或引用的元素，引用是属性 `th:ref` 或 `th:fragment` 属性）。
- 选择器也允许没有元素名称/引用，只要它们包含参数规范。因此 `[@class='oneclass']` 是一个有效的选择器，它使用带有值的 `class` 属性查找任何元素（标记）`"oneclass"`。

高级属性选择功能：

- 除了 `=`（相等）之外，其他比较运算符也是有效的：`!=`（不等于），`^=`（以...开头）和 `$=`（以...结尾）。例如：`x[@class^='section']` 表示具有名称的元素 `x` 和以... `class` 开头的属性值 `section`。
- 可以从 `@`（XPath样式）和不带（jQuery样式）开始指定属性。所以 `x[z='v']` 相当于 `x[@z='v']`。
- 多属性修饰符既可以与 `and`（XPath样式）连接，也可以通过链接多个修饰符（jQuery样式）来连接。所以 `x[@z1='v1' and @z2='v2']` 实际上相当于 `x[@z1='v1'][@z2='v2']`（也是 `x[z1='v1'][z2='v2']`）。

类似jQuery的直接选择器：

- `x.oneclass` 相当于 `x[class='oneclass']`。
- `.oneclass` 相当于 `[class='oneclass']`。
- `x#oneid` 相当于 `x[id='oneid']`。
- `#oneid` 相当于 `[id='oneid']`。
- `x%oneref` 表示 `<x>` 具有 `th:ref="oneref"` 或 `th:fragment="oneref"` 属性的标记。
- `%oneref` 表示具有 `th:ref="oneref"` 或 `th:fragment="oneref"` 属性的任何标记。请注意，这实际上相当于 `oneref` 因为可以使用引用而不是元素名称。
- 直接选择器和属性选择器可以混合使用：`a.external[@href^='https']`。

所以上面的Markup Selector表达式：

```
<div th:insert="mytemplate :: //div[@class='content']">...</div>
```

可以写成：

```
<div th:insert="mytemplate :: div.content">...</div>
```

检查一个不同的例子，这个：

```
<div th:replace="mytemplate :: myfrag">...</div>
```

将寻找 **th:fragment="myfrag"** 片段签名（或 **th:ref** 引用）。但是 **myfrag** 如果它们存在的话，它们也会寻找带有名称的标签（它们不是HTML格式的标签）。注意区别：

```
<div th:replace="mytemplate :: .myfrag">...</div>
```

...实际上会查找任何元素 **class="myfrag"**，而不关心 **th:fragment** 签名（或 **th:ref** 引用）。

## 多值类匹配

---

标记选择器将类属性理解为多值，因此即使元素具有多个类值，也允许在此属性上应用选择器。

例如，**div.two** 将匹配 **<div class="one two three" />**