

# FreeMarker 手册

用于 FreeMarker 2.3.16

Translated By Nan Lei

南磊 译

Copyright:

The Chinese translation of the FreeMarker Manual by Nan Lei is licensed under a Creative Commons Attribution 3.0 Unported License (see <http://creativecommons.org/licenses/by/3.0/> ).

This licence only applies to the Chinese translation, not to the original (English) FreeMarker Manual.



版权说明:

FreeMarker 中文版手册由南磊翻译，本文档基于 Creative Commons Attribution 3.0 Unported 授权许可（参见 <http://creativecommons.org/licenses/by/3.0/deed.zh> ）

本许可仅应用于中文版，不对原版英文手册。

（译者联系方式为： [nanlei1987@gmail.com](mailto:nanlei1987@gmail.com) ）

# 目录

目录.....	2
前言.....	5
什么是 FreeMarker? .....	5
我们应该阅读什么内容? .....	5
文档规约.....	6
联系我们.....	6
几点说明.....	6
第一部分 模板开发指南.....	7
第一章 模板开发入门 .....	7
1.1 简介 .....	7
1.2 模板 + 数据模型 = 输出.....	7
1.3 数据模型一览 .....	9
1.4 模板一览 .....	11
第二章 数值和类型.....	17
2.1 基本内容 .....	17
2.2 类型 .....	20
第三章 模板 .....	24
3.1 总体结构 .....	24
3.2 指令 .....	25
3.3 表达式.....	26
3.4 插值 .....	41
第四章 其它 .....	44
4.1 自定义指令.....	44
4.2 在模板中定义变量 .....	50
4.3 命名空间 .....	52
4.4 空白处理 .....	56
4.5 替换（方括号）语法.....	59
第二部分 程序开发指南.....	61
第一章 程序开发入门 .....	61
1.1 创建配置实例 .....	61
1.2 创建数据模型 .....	61
1.3 获得模板 .....	62
1.4 合并模板和数据模型.....	63
1.5 将代码放在一起.....	63
第二章 数据模型 .....	65
2.1 基本内容 .....	65
2.2 标量 .....	66
2.3 容器 .....	67
2.4 方法 .....	68
2.5 指令 .....	69
2.6 节点变量 .....	76

2.7 对象包装 .....	76
第三章 配置 .....	80
3.1 基本内容 .....	80
3.2 共享变量 .....	81
3.3 配置信息 .....	81
3.4 模板加载 .....	83
3.5 错误控制 .....	86
第四章 其它 .....	90
4.1 变量 .....	90
4.2 字符集问题 .....	91
4.3 多线程 .....	92
4.4 Bean 的包装 .....	93
4.5 日志 .....	99
4.6 在 servlets 中使用 FreeMarker .....	99
4.7 为 FreeMarker 配置安全策略 .....	106
4.8 遗留的 XML 包装实现 .....	107
4.9 和 Ant 一起使用 FreeMarker .....	110
4.10 Jython 包装器 .....	111
第三部分 XML 处理指南 .....	113
前言 .....	113
第一章 揭示 XML 文档 .....	113
1.1 节点树 .....	113
1.2 将 XML 放到数据模型中 .....	115
第二章 必要的 XML 处理 .....	116
2.1 通过例子来学习 .....	116
2.2 形式化描述 .....	124
第三章 声明的 XML 处理 .....	127
3.1 基础内容 .....	127
3.2 详细内容 .....	130
第四部分 参考文档 .....	133
第一章 内建函数参考文档 .....	133
1.1 处理字符串的内建函数 .....	133
1.2 处理数字的内建函数 .....	146
1.3 处理日期的内建函数 .....	150
1.4 处理布尔值的内建函数 .....	153
1.5 处理序列的内建函数 .....	153
1.6 处理哈希表的内建函数 .....	159
1.7 处理节点 (XML) 的内建函数 .....	159
1.8 很少使用的和专家级的内建函数 .....	161
第二章 指令参考文档 .....	164
2.1 if, else, elseif 指令 .....	164
2.2 switch, case, default, break 指令 .....	166
2.3 list, break 指令 .....	167
2.4 include 指令 .....	168

2.5 import 指令 .....	171
2.6 noparse 指令 .....	172
2.7 compress 指令 .....	173
2.8 escape, noescape 指令 .....	174
2.9 assign 指令 .....	176
2.10 global 指令 .....	178
2.11 local 指令 .....	179
2.12 setting 指令 .....	179
2.13 用户自定义指令 (<@...>) .....	181
2.14 macro, nested, return 指令 .....	183
2.15 function, return 指令 .....	188
2.16 flush 指令 .....	189
2.17 stop 指令 .....	190
2.18 ftl 指令 .....	190
2.19 t, lt, rt 指令 .....	192
2.20 nt 指令 .....	193
2.21 attempt, recover 指令 .....	193
2.22 visit, recurse, fallback 指令 .....	195
第三章 特殊变量参考文档 .....	199
第四章 FTL 中的保留名称 .....	200
第五章 废弃的 FTL 结构 .....	200
5.1 废弃的指令列表 .....	200
5.2 废弃的内建函数列表 .....	200
5.3 老式的 macro 和 call 指令 .....	201
5.4 转换指令 .....	202
5.5 老式 FTL 语法 .....	204
5.6 #{...}式的数字插值 .....	205

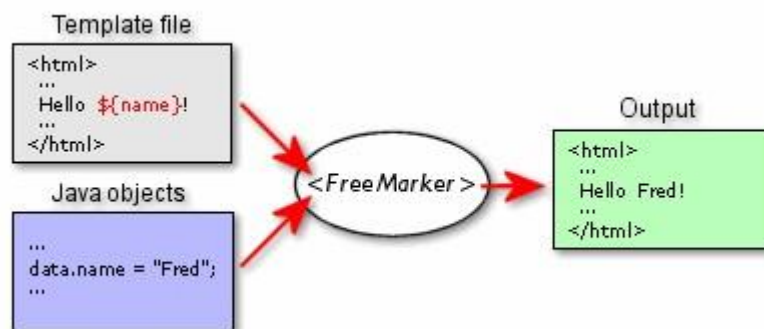
# 前言

## 什么是 FreeMarker?

FreeMarker 是一款模板引擎：一种基于模板的、用来生成输出文本(任何来自于 HTML 格式的文本来自动生成源代码)的通用工具。它是为 Java 程序员提供的一个开发包或者说是类库。它不是面向最终用户，而是为程序员提供的嵌入他们开发产品的一款应用程序。

FreeMarker 实际上是被设计用来生成 HTML 网页，尤其是通过基于实现了 MVC(Model View Controller, 模型-视图-控制器)模式的 Servlet 应用程序。使用 MVC 模式的动态网页的构思是使得你可以将前端设计者(编写 HTML)从程序员中分离出来。所有人各司其职，发挥其擅长的一面。网页设计师可以改写页面的显示效果而不受程序员编译代码的影响，因为应用程序的逻辑(Java 程序)和页面设计(FreeMarker 模板)已经分开了。页面模板代码不会受到复杂的程序代码的影响。这种分离的思想即便对一个程序员和页面设计师是同一个人来说的项目都是非常有用的，因为分离使得代码保持简洁而且便于维护。

尽管 FreeMarker 也有编程的能力，但它也不是像 PHP 那样的一种全面的编程语言。反而，Java 程序准备数据来显示(比如 SQL 查询)，FreeMarker 仅仅使用模板生成文本页面来呈现已经准备好的数据。



FreeMarker 不是 Web 应用框架。它是 Web 应用框架中的一个适用的组件，但是 FreeMarker 引擎本身并不知道 HTTP 协议或 Servlet。它仅仅来生成文本。即便这样，它也非常适用于非 Web 应用环境。要注意的是，我们使用 FreeMarker 作为视图层组件为如 Struts 这样的 Model 2 框架提供现成的解决方案。

FreeMarker 是免费的，基于 BSD 风格的许可。它是 OSI 认证的开源软件。OSI 认证是开源倡议的认证标识。

## 我们应该阅读什么内容?

如果你是一名...

前端设计师，那么你应该阅读**模板开发指南**，然后如果需要的话可以参阅参考手册来获取更多技术细节。

程序员，那么你应该先阅读**模板开发指南**，然后是**程序开发指南**，最后如果需要的话可以参阅参考手册来获取更多技术细节。

## 文档规约

变量名，模板代码段，Java 类名等用如下格式书写，如：foo。

如果需要具体值来代替某些内容，那么用斜体书写，如：Hello *yourName*!

模板示例如下书写

Something

数据对象示例如下书写:

Something

输出数据示例如下书写:

Something

程序示例如下书写:

Something

在面向页面设计师和程序员所编写的章节中代码段给程序员这样写：这只是对程序员而言的。

这样来强调新名词：一些新名词

## 联系我们

获取最新版本的 FreeMarker，订阅邮件请访问 FreeMarker 主页：<http://freemarker.org>

如果你需要帮助或者有好的建议，可以使用邮件(邮件档案可以免费搜索)或者 Web 论坛。如果你想报告一个 Bug，请使用 Web 的 Bug 跟踪器或者是邮件。查阅这些内容请访问 <http://freemarker.org>。同时，要注意我们有一个 FAQ 和索引，你可以使用它们。

## 几点说明

因为英文版文档的作者是匈牙利人，其母语非英语，那么在这种情况的翻译过程，可能会有错误存在，作者结合自身的实践力求准确，但因个人才疏学浅，水平有限，恳请读者批评指正。

手册的更新根据大家的反馈随时进行，但只在有阶段性成果时公开发布修正版本，并在 FreeMarker 2.4 版本研发完整后，会即时联系原作者获取新特性以便修改。

本翻译是免费的，您可以自由下载和传播，不可用于任何商业行为。但文档版权归译者所有，原版归 FreeMarker 项目组所有，您可以引用其中的描述，但必须指明出处。如需用于商业行为，您必须和原作者联系。

如果你发现英文原版任何错误（包括语法错误，错别字）或者是在文档中找到一些误导或混淆错误，也可以是其他的建议，或是咨询 FreeMarker 中的问题，您可以联系原作者。

E-mail: [ddekany@freemail.hu](mailto:ddekany@freemail.hu)

注：关于本文档的翻译错误（包括语法错误，错别字）或中文技术交流，可以发送邮件至译者邮箱：[nanlei1987@gmail.com](mailto:nanlei1987@gmail.com)，我们共同研究，共同进步，不胜感激。

# 第一部分 模板开发指南

## 第一章 模板开发入门

章节内容如下：

简介

模板+数据模型=输出

数据模型一览

模板一览

### 1.1 简介

本章是关于 **FreeMarker** 的非常简略的介绍，后续章节中将会详细介绍它们。不过没关系，只要你阅读了本章节的内容，你就能够编写简单的却很有用的 **FreeMarker** 模板程序。

### 1.2 模板 + 数据模型 = 输出

假设你在线商店应用中需要一个 **HTML** 页面，和下面这个页面相似：

```
<html>
<head>
  <title>Welcome!</title>
</head>
<body>
  <h1>Welcome Big Joe!</h1>
  <p>Our latest product:
  <a href="products/greenmouse.html">green mouse</a>!
</body>
</html>
```

比方说，用户名（所有的“**Big Joe**”）应该是登录这个网页的访问者的名字，最新产品的数据应该来自于数据库，这样它才可以随时变更。在这样的情况下你不能在 **HTML** 页面中直接输入登录用户名，最新产品的 **URL** 和名称，你不能使用静态的 **HTML** 代码，那样是不能及时改变的。

对于这个问题，**FreeMarker** 的解决方案是使用模板来代替静态 **HTML** 文本。模板文件同样是静态的 **HTML** 代码，但是除了这些 **HTML** 代码外，代码中还包括了一些 **FreeMarker** 指令，这些指令就能做到动态效果。

```
<html>
<head>
  <title>Welcome!</title>
</head>
<body>
  <h1>Welcome ${user}!</h1>
  <p>Our latest product:
  <a href="${latestProduct.url}">${latestProduct.name}</a>!
</body>
</html>
```

这个模板存放在 Web 服务器上，看上去像是静态 HTML 页面。但是不管何时，只要有人访问这个页面时，FreeMarker 将会介入执行，然后动态转换模板，用最新的数据内容替换 `${...}` 中的部分（例如：用 Big Joe 代替 `${user}` 或者其他的访问者的用户名），生成普通的 HTML 文本并发送结果到访问者的 Web 浏览器端去显示。所以访问者的 Web 浏览器会接收到类似于第一个 HTML 示例中的内容（也就是说，显示普通的 HTML 文本而没有 FreeMarker 的指令），浏览器也不会感知到 FreeMarker 在服务器端被使用了。模板文件本身（存储在 Web 服务器端的文件）在这个过程内也不会改变什么，所以这个转换过程发生在一次又一次的访问中。这就保证了显示的信息总是即时的。

现在，你也许已经注意到，该模板并没有包含关于如何找出当前的访问者是谁，或者是如何去查询数据库查找最新的产品的指令。它似乎已经知道这些数据了。确实是这样，FreeMarker 背后（确切的说是 MVC 模式的背后）的重要思想就是表现逻辑和业务逻辑相分离。在模板只是处理显示问题，也就是视觉设计问题和格式问题。所准备的显示数据（如用户名等）与 FreeMarker 无关，通常是使用 Java 语言或其他目的语言来编写的。所以模板开发者不需要关心这些数值是如何计算出来的。事实上，在模板保持不变的同时，这些数值的计算方式可以完全发生变化。而且，除了模板外，页面外观发生的变化可以完全不触碰其他任何东西。当模板开发者和程序员是不同一个人的时候，分离带来的好处更是显而易见的。

FreeMarker（还有模板开发者）它并不关心数据是如何计算的，FreeMarker 只是知道真实的数据是什么。模板能用的所有数据被包装成 **data-model 数据模型**。数据模型的创建是通过已经存在的程序来计算得到的。至于模板开发者，数据模型像是树状结构（比如硬盘上的文件夹和文件），正如是这样，数据模型就可以如下形式来描述：

```
(root)
|
+- user = "Big Joe"
|
+- latestProduct
  |
  +- url = "products/greenmouse.html"
  |
  +- name = "green mouse"
```

（为了避免误解：数据模型并不是文本文件，上面所描述的只是一种数据模型的表现形式。它来自于 Java 对象，但这会成为 Java 程序员要面对的问题。）

比较之前模板中你看到的 `${user}` 和 `${latestProduct.name}`。作为一种比



喻：数据模型就像计算机文件系统上的内容：根 `root` 和 `latestProduct` 对于目录（文件夹），`user`，`url` 和 `name` 对应文件。`url` 和 `name` 在 `latestProduct` 目录中，所以 `latestProduct.name` 就像是说 `latestProduct` 目录的 `name` 一样。但是我所说的，这仅仅是个比喻，这里并没有真实的文件和目录。

概括地讲，模板和数据模型是 `FreeMarker` 所需，并用来生成输出内容的（比如之前展示的 `HTML`）：模板+数据模型=输出

## 1.3 数据模型一览

正如你看到的，数据模型基本是树状结构的。这棵树可以复杂而且深度很大，比如：

```
(root)
|
+- animals
|  |
|  +- mouse
|  |  |
|  |  +- size = "small"
|  |  |
|  |  +- price = 50
|  |
|  +- elephant
|  |  |
|  |  +- size = "large"
|  |  |
|  |  +- price = 5000
|  |
|  +- python
|  |  |
|  |  +- size = "medium"
|  |  |
|  |  +- price = 4999
|
+- test = "It is a test"
|
+- whatnot
|
+- because = "don't know"
```

上图中变量扮演目录的角色（根 `root`，`animal`，`mouse`，`elephant`，`python`，`whatnot`）被称为 **hash 哈希表**。哈希表通过可查找的名称（例如：“`animal`”，“`mouse`”，“`price`”）来访问存储的其他变量（如子变量）。

如果仅存储单值的变量（`size`，`price`，`text` 和 `because`）则它们被称为 **scalars 标量**。

如果要在模板中使用子变量，那应该从根 `root` 开始指定它的路径，每级之间用点来分

隔。要访问 `price` 和 `mouse` 的话，应该从根开始，先是 `animals`，然后是 `mouse`，最后是 `price`，所以应该这样写：`animals.mouse.price`。当放置`${...}`这种特定代码在表达式前后时，我们就告诉 `FreeMarker` 在那个位置上要来输出对应的文本。

**sequences 序列**也是一种非常重要的变量，它们和哈希表变量相似，但是它们不存储所包含变量的名称，而是按顺序存储子变量。这样，就可以使用数字索引来访问这些子变量。在这种数据模型中，`animal` 和 `whatnot.fruits` 就是序列。

```
(root)
|
+- animals
|  |
|  +- (1st)
|  |  |
|  |  +- name = "mouse"
|  |  |
|  |  +- size = "small"
|  |  |
|  |  +- price = 50
|  |
|  +- (2nd)
|  |  |
|  |  +- name = "elephant"
|  |  |
|  |  +- size = "large"
|  |  |
|  |  +- price = 5000
|  |
|  +- (3rd)
|  |  |
|  |  +- name = "python"
|  |  |
|  |  +- size = "medium"
|  |  |
|  |  +- price = 4999
|
+- whatnot
|
+- fruits
|
+- (1st) = "orange"
|
+- (2nd) = "banana"
```

可以使用数组的方括号方式来访问一个序列的子变量。索引从零开始（从零开始是程序员写代码的传统风格），那么就意味着序列第一项的索引是 `0`，第二项的索引是 `1`，并以此类

推。要得到第一个动物的名称的话，那么就应该这么写代码：`animals[0].name`。要得到 `whatnot.fruits`（就是“banana”这个字符串）的第二项，那么就应该这么来写：`whatnot.fruits[1]`。

标量可以分为如下类别：

字符串：这是文本类型，字符的任意序列，比如“m”，“o”，“u”，“s”，“e”这些，而且 `name-S` 和 `size-S` 也是字符串范畴。

数字：这是数值类型，比如 `price-S` 这些。在 `FreeMarker` 中字符串“50”和数字 50 是两种完全不同的类型。前者只是两个字符的序列（这恰好是我们可以读的一个数字），而后者是一个可以在算数运算中直接使用的数值。

日期/时间：这是时间日期类型。例如动物被捕捉的日期，或商店开始营业的时间。

布尔值：对应对/错（是/否，开/关等）这样仅代表正反的值。比如动物可以有一个受保护（`protected`，译者注）的子变量，这个变量存储这个动物是否被保护起来。

总结：

数据模型可以被看做是树状结构。

标量存储单一的值，这种类型的值可以是字符串，数字，日期/时间或者是布尔值。

哈希表是存储变量和与其相关且有唯一标识名称变量的容器。

序列是存储有序变量的容器。存储的变量可以通过数字索引来检索，索引通常从零开始。

## 1.4 模板一览

简介

指令示例

if 指令

list 指令

include 指令

联合使用指令

处理不存在的变量

### 1.4.1 简介

最简单的模板是普通 HTML 文件（或者是其他任何文本文件—`FreeMarker` 本身不属于 HTML）。当客户端访问页面时，`FreeMarker` 要发送 HTML 代码至客户端浏览器端显示。如果想要页面动起来，就要在 HTML 中放置能被 `FreeMarker` 所解析的特殊部分。

`${...}`：`FreeMarker` 将会输出真实的值来替换花括号内的表达式，这样被称为 **interpolations 插值**，可以参考第上面一个示例的内容。

**FTL tags 标签**（`FreeMarker` 模板的语言标签）：FTL 标签和 HTML 标签有一点相似，但是它们是 `FreeMarker` 的指令而且是不会直接输出出来的东西。这些标签的使用一般以符号 `#` 开头。（用户自定义的 FTL 标签使用 `@` 符号来代替 `#`，但这是更高级的主题了，后面会详细讨论）

**Comments 注释**：`FreeMarker` 的注释和 HTML 的注释相似，但是它用 `<#--` 和 `-->` 来分隔。任何介于这两个分隔符（包含分隔符本身）之间内容会被 `FreeMarker` 忽略，就不会输出出来了。

其他任何不是 FTL 标签，插值或注释的内容将被视为静态文本，这些东西就不会被

FreeMarker 所解析，会被按照原样输出出来。

**directives 指令** 就是所指的 FTL 标签。这些指令在 HTML 的标签（如 `<table>` 和 `</table>`）元素（如 `table` 元素）中的关系是相同的。（如果现在你还不能区分它们，把“FTL 标签”和“指令”看做是同义词即可。）

## 1.4.2 指令示例

尽管 FreeMarker 有很多指令，作为入门，在快速了解中我们仅仅来看三个最为常用的指令。

### 1.4.2.1 if 指令

使用 if 指令可以有条件地跳过模板的一部分，这和程序语言中 if 是相似的。假设在第一个示例中，你只想向你的老板 **Big Joe**（而不是其他人）问好，就可以这样做：

```
<html>
<head>
  <title>Welcome!</title>
</head>
<body>
  <h1>
    Welcome ${user}<#if user == "Big Joe">, our beloved
    leader</#if>!
  </h1>
  <p>Our latest product:
  <a href="${latestProduct.url}">${latestProduct.name}</a>!
</body>
</html>
```

在这里，我们告诉了 FreeMarker，我们尊敬的领导才是 if 条件中那唯一的 `user` 变量值，当它和“Big Joe”相同时才显示出来。那么，当 `condition` 的判断结果为 `false`（布尔值）时，在 `<#if condition>` 和 `</#if>` 标签之间的内容将会被略过。

我们来详细说说 `condition` 的使用：`==` 是用来判断在它两侧的值相等的操作符，比较的结果是布尔值，`true` 或者 `false`。在 `==` 的左侧，是引用的变量，我们很熟悉这样的语法，它会被变量的值来替代。右侧是指定的字符串，在模板中的字符串必须放在引号内。

当 `price` 是 0 的时候，下面的代码将会打印：“Pythons are free today!”

```
<#if animals.python.price == 0>
  Pythons are free today!
</#if>
```

和之前的示例相似，字符串被直接指定，但是这里则是数字（0）被直接指定。注意到数字是不用放在引号内的。如果将 0 放在引号内（“0”），FreeMarker 就会将其误判为字符串了。

当 `price` 不是 0 的时候，下面的代码将会打印：“Pythons are not free today!”

```
<#if animals.python.price != 0>
  Pythons are free today!
</#if>
```

你也许会猜测了，**!=**就是不等于。

你也可以这样来写代码（使用数据模型来描述哈希表变量）：

```
<#if animals.python.price < animals.elephant.price>
  Pythons are cheaper than elephants today.
</#if>
```

使用**<#else>**标签可以指定当条件为假时程序可以执行的内容。例如：

```
<#if animals.python.price < animals.elephant.price>
  Pythons are cheaper than elephants today.
<#else>
  Pythons are not cheaper than elephants today.
</#if>
```

如果蟒蛇的价格比大象的价格低，将会打印“Python are cheaper than elephants today.”，否则打印“Pythons are not cheaper than elephants today.”

如果变量是布尔值(true 或者 false)，那么就可以直接让其作为 **if** 的条件 **condition**：

```
<#if animals.python.protected>
  Warning! Pythons are protected animals!
</#if>
```

### 1.4.2.2 list 指令

当需要用列表来遍历集合的内容时，**list** 指令是相当好用的。例如，如果用之前示例描述序列的数据模型来合并模板。

```
<#if animals.python.protected>
  Warning! Pythons are protected animals!
</#if>
```

那么输出将会是这样的：

```
<p>We have these animals:
<table border=1>
  <tr><th>Name<th>Price
  <tr><td>mouse<td>50 Euros
  <tr><td>elephant<td>5000 Euros
  <tr><td>python<td>4999 Euros
</table>
```

`list` 指令的一般格式为:

```
<#list sequence as loopVariable>repeatThis</#list>
```

`repeatThis` 部分将会在给定的 `sequence` 遍历时的每项中重复, 从第一项开始, 一个接着一个。所有的重复中 `loopVariable` 将持有当前项的值。这个循环变量仅存在于 `<#list ...>` 和 `</#list>` 标签之间。

再看一个示例, 遍历示例数据模型 `fruits`。

```
<p>And BTW we have these fruits:
<ul>
<#list whatnot.fruits as fruit>
  <li>${fruit}
</#list>
</ul>
```

`whatnot.fruits` 表达式应该很熟悉了, 我们引用了数据模型章节中展示的变量。

### 1.4.2.3 include 指令

使用 `include` 指令, 我们可以插入其他文件的内容到当前的模板中。

假设要在一些页面中显示版权声明的信息。那么可以创建一个文件来单独包含版权声明, 之后在需要它的地方插入它即可。比方说, 我们可以将版权信息单独存放在 `copyright_footer.html` 文件中。

```
<hr>
<i>
Copyright (c) 2000 <a href="http://www.acmee.com">Acmee
Inc</a>,
<br>
All Rights Reserved.
</i>
```

当需要用到这个文件时, 可以使用 `include` 指令来实现插入。

```
<html>
<head>
  <title>Test page</title>
</head>
<body>
  <h1>Test page</h1>
  <p>Blah blah...
<#include "/copyright_footer.html">
</body>
</html>
```

输出的内容为:

```

<html>
<head>
  <title>Test page</title>
</head>
<body>
  <h1>Test page</h1>
  <p>Blah blah...
<hr>
<i>
Copyright (c) 2000 <a href="http://www.acmee.com">Acmee
Inc</a>,
<br>
All Rights Reserved.
</i>
</body>
</html>

```

如果改变了 `copyright_footer.html` 中的内容, 那么访问者就会在所有页面中看到新的版权声明信息了。

#### 1.4.2.4 联合使用指令

在页面也可以多次使用指令, 而且指令间可以相互嵌套, 正如在 HTML 元素中嵌套使用标签一样。下面的代码会遍历动物, 用大号字体来打印大型动物的名字。

```

<p>We have these animals:
<table border=1>
  <tr><th>Name<th>Price
  <#list animals as being>
  <tr>
    <td>
      <#if being.size == "large"><font size="+1"></#if>
      ${being.name}
      <#if being.size == "large"></font></#if>
    <td>${being.price} Euros
  </#list>
</table>

```

注意到 FreeMarker 并不解析 FTL 标签外的文本, 插值和注释, 条件不满足时它也会忽略所有嵌套的 `font` 标签。

#### 1.4.2.5 处理不存在的变量

在实际应用中数据模型经常会有可选的变量 (也就是说有时可能不存在实际值)。除了

一些典型的人为原因的失误，FreeMarker 不能容忍引用不存在的变量，除非明确地告诉它当变量不存在时如何处理。这里介绍两种典型的处理方法。

*对程序员而言：一个不能存在的变量和一个是 `null` 的变量，对于 FreeMarker 来说是一样的，所以这里丢失所指的包含这两种情况。*

不论在哪里引用变量，都可以指定一个默认值来避免变量丢失这种情况，通过在变量名后面跟着一个 `!` 和默认值。就像下面的例子，当 `user` 从数据模型中丢失时，模板将会将 `user's` 的值表现为字符串 `"Anonymous"`。（若 `user` 并没有丢失，那么模板就表现出 `"Anonymous"` 不存在一样）：

```
<h1>Welcome ${user!"Anonymous"}!</h1>
```

当然也可以通过放置 `??` 在变量名后面来询问 FreeMarker 一个变量是否存在。将它和 `if` 指令合并，那么如果 `user` 变量不存在的话将会忽略整个问候代码段：

```
<#if user??><h1>Welcome ${user}!</h1></#if>
```

关于多级访问的变量，比如 `animals.python.price`，书写代码：`animals.python.price!0`，仅当 `animals.python` 存在而仅仅最后一个子变量 `price` 可能不存在（这种情况下我们假设价格是 0）。如果 `animals` 或者 `python` 不存在，那么模板处理将会以“未定义的变量”错误停止。为了防止这种情况的发生，可以这样来书写代码 `(animals.python.price)!0`。这种情况下当 `animals` 或 `python` 不存在时表达式的结果仍然是 0。对于 `??` 也是同样用来处理这种逻辑的：`animals.python.price??` 对比 `(animals.python.price)??`



## 第二章 数值和类型

章节内容如下：

基本内容  
类型

### 2.1 基本内容

简介  
什么是数值？  
什么是类型？  
数据模型是哈希表

#### 2.1.1 简介

**注意：**

这里假设你已经阅读完入门章节了。

理解数值和类型的概念是理解数据模型的关键所在。然而，数值和类型的概念并不局限于数据模型，下面你就会看到。

#### 2.1.2 什么是数值？

*对于程序员来说可以直接跳过这部分，它和程序语言中的数值类型是相似的。*

你所知道的来自于每天所使用的数字，比如 **16**，**0.5** 等这些用语就是数值的示例，也就是数字。在计算机语言中，这些用语有着更广泛的含义，比如数值并不一定是数字值，看下面这个数据模型：

```

(root)
|
+- user = "Big Joe"
|
+- today = Jul 6, 2007
|
+- todayHoliday = false
|
+- lotteryNumbers
|   |
|   +- (1st) = 20
|   |
|   +- (2st) = 14
|   |
|   +- (3rd) = 42
|   |
|   +- (4th) = 8
|   |
|   +- (5th) = 15
|
+- cargo
  |
  +- name = "coal"
  |
  +- weight = 40

```

我们说变量 `user` 的数值是“Big Joe”（字符串），`today` 的数值是 Jul 6,2007（日期），`todayHoliday` 的数值是 `false`（布尔值，是/否，这样的值）。`lotteryNumbers` 的数值是包含 20, 14, 42, 8, 15 的序列。在这种意义上，`lotteryNumbers` 是多值的，它包含多个数值（如其中的第二项是 14），但是 `lotteryNumbers` 本身还是单值。它像一个装有很多东西的盒子，整个盒子被看做是独立的。最后有一个数值 `cargo`，它是一个哈希表（也可以看做是盒子）。所以数值就是存储在变量中的（在 `user`，`cargo` 或 `cargo.name` 中）东西。但是不需要存储的数值也可以称之为数值，比如这里的数字 100：

```
<#if cargo.weight < 100>Light cargo</#if>
```

当模板被执行时，计算的临时结果也称为数值，比如 20, 120（它会打印 120）

```
${cargo.weight / 2 + 100}
```

这最后一种的解释：两个数 40（货物的重量）和 2 相除的结果是 20，这是一个新计算出的数值。把它和 100 相加，那么 120 就出来了，接着就打印出来了（`${...}`），接着模板继续向下执行直到所有结果都计算出来。

现在你应该能体会到数值这个词的含义了，不仅仅是数字值。

### 2.1.3 什么是类型？

数值中非常重要的一个概念就是类型。比方说，变量 `user` 的数值类型是字符串，`lotteryNumbers` 的数值类型是序列。数值的类型非常重要，因为它决定了这些数值可以在哪里使用的最大限度。比如 `${user/2}` 就是错误的，但是 `${cargo.weight/2}` 就能计算出结果 20，除法仅对数字值有效，而不能作用于字符串。仅当 `cargo` 是一个哈希表时 `cargo.name` 可以使用。也可以用 `<#list ...>` 仅仅来遍历序列。`<#if ...>` 指令的条件 `condition` 只能是布尔值等。

#### 注意：

这里说一点点术语：称“布尔”或“布尔值”或“布尔类型”都是相同的含义。

数值同时也可以含有多种类型，尽管这样很少使用。看下面这个数据模型 `mouse`，就又是字符串又是哈希表。

```
(root)
|
+- mouse = "Yerri"
   |
   +- age = 12
      |
      +- color = "brown"
```

如果用上面的数据模型合并到模板中，就该这么来写：

```
${mouse}      <#-- 用 mouse 作为字符串 -->
${mouse.age}   <#-- 用 mouse 作为哈希表 -->
${mouse.color} <#-- 用 mouse 作为哈希表 -->
```

它的输出内容为：

```
Yerri
12
brown
```

### 2.1.4 数据模型是哈希表

注意观察每个你已经知道的数据模型：被“(root)”标识的内容就是哈希表类型的数值。当书写如 `user` 这样的代码，那就意味着想要把“`user`”变量存储在哈希表的根上。而如果代码是：`root.user`，也没有名为“`root`”的变量，那么这就没有任何作用。

某些人也许会被这种数据模型的例子所困惑，也就是说，根哈希表包含更多的哈希表或序列（如 `lotteryNumbers` 和 `cargo`）。其他就没有更特殊的了。哈希表包含其他变量，那些变量包含数值，数值可以是字符串，数字等，当然也可以是哈希表或序列。最初我们解释过，就像字符串和数字，序列或哈希表也是数值。

## 2.2 类型

简介

标量

容器

子程序

方法和函数

用户自定义指令

函数/方法和用户自定义指令的比较

其它

节点

### 2.2.1 简介

支持的类型有：

- 标量：
  - ◆ 字符串
  - ◆ 数字
  - ◆ 布尔值
  - ◆ 日期
- 容器：
  - ◆ 哈希表
  - ◆ 序列
  - ◆ 集
- 子程序：
  - ◆ 方法和函数
  - ◆ 用户自定义指令
- 其它/很少使用：
  - ◆ 节点

### 2.2.2 标量

标量是最基本，最简单的数值类型，它们可以是：

- 字符串：简单的文本，例如：产品名称。  
如果想在模板中直接给出字符串的值，而不是使用数据模型中的变量，那么将文本写在引号内即可，比如“green mouse”或者‘green mouse’。（关于语法的更多细节请看后续章节）
- 数字：例如：产品的价格。*整数和非整数是不区分的，只有单一的数字类型。比如使用了计算器，计算 3/2 的结果是 1.5 而不是 1。*  
如果要在模板中直接给出数字的值，可以这么来写：150，-90.05，或者 0.001。  
（关于语法的更多细节请看后续章节）
- 布尔值：布尔值代表了逻辑上的对或错（是或否）。例如：用户到底是否登录了。

典型的应用是使用布尔值作为 `if` 指令的条件, 比如 `<#if loggedIn>...</#if>` 或者 `<#if price==0>...</#if>`, 后面这个 `price==0` 部分的结果就是布尔值。

在模板中可以使用保留字 `true` 和 `false` 来指定布尔值。

- 日期: 日期变量可以存储和日期/时间相关的数据。有三种变化。
  - ◆ 精确到天的日期 (通常指的是“日期”), 比如 `April 4, 2003`
  - ◆ 每天的时间 (不包括日期部分), 比如 `10:19:18 PM`。时间的存储精确到毫秒。
  - ◆ 日期-时间 (也称作“时间戳”), 比如 `April 4, 2003 10:19:18 PM`。时间部分的存储精确到毫秒。

不幸的是, 受到 `Java` 平台的限制, `FreeMarker` 又是不能决定日期的那部分来使用 (也就是说, 是日期-时间, 每天的时间, 等)。这个问题的解决方法是高级的主题, 后面的章节将会讨论到。

在模板中直接定义日期数值是可以的, 但这也是高级主题, 后面的章节将会讨论到。

要记住, `FreeMarker` 区别字符串, 数字和布尔值, 所以字符串“150”和数字 150 是完全不同的。数字持有的是数字的值, 布尔值表达的是逻辑上的对或错。字符串可以是任意字符的序列。

## 2.2.3 容器

这些值存在的目的是为了包含其他变量, 它们仅仅作为容器。被包含的变量通常是子变量。容器的类型有:

- 哈希表: 每个子变量都可以通过一个唯一的名称来查找, 这个名称是不受限制的字符串。哈希表并不确定其中子变量的顺序, 也就是说没有第一个变量, 第二个变量这样的说法, 变量仅仅是通过名称来访问的。
- 序列: 每个子变量通过一个整数来标识。第一个子变量的标识符是 0, 第二个是 1, 第三个是 2, 这样来类推, 而且子变量是有顺序的。这些数字通常被称为是子变量的索引。序列通常比较密集, 也就是所有索引, 包括最后一个子变量的都和子变量是相关联的, 但不是绝对必要的。子变量的数值类型也并不需要一致。
- 集: 从模板设计者角度来看, 集是有限制的序列。不能获取集的大小, 也不能通过索引取出集中的子变量, 但是它们仍然可以通过 `list` 指令来遍历。

要注意一个数值也可有多种类型, 对于一个数值可能存在哈希表和序列这两种类型, 这时, 该变量就支持索引和名称两种访问方式。不过容器基本是当作哈希表或者序列来使用的, 而不是两者同时使用。

尽管存储在哈希表, 序列 (集) 中的变量可以是任意类型的, 这些变量也可以是哈希表, 序列 (集)。这样就可以构建任意深度的数据结构。

数据模型本身 (最好说为它的根) 也是哈希表。

## 2.2.4 子程序

### 2.2.4.1 方法和函数

一个值是方法或函数的时候那么它就可以计算其他的值, 结果取决于传递给它的参数。

对于程序员来说：方法/函数是第一类值，就像函数化的编程语言。也就是说方函数/方法也可以是其他函数或方法的参数或者返回值，并可以把它定义为变量。

假设程序员在数据模型中放置了一个方法变量 `avg`，那么它就可以被用来计算数字的平均值。给定 3 和 5 作为参数，访问 `avg` 时就能得到结果 4。

方法的使用后续章节会有解释，下面这个示例会帮助我们理解方法的使用：

```
The average of 3 and 5 is: ${avg(3, 5)}  
The average of 6 and 10 and 20 is: ${avg(6, 10, 20)}  
The average of the price of a python and an elephant is:  
${avg(animals.python.price, animals.elephant.price)}
```

可以得到如下的输出：

```
The average of 3 and 5 is: 4  
The average of 6 and 10 and 20 is: 12  
The average of the price of a python and an elephant is:  
4999.5
```

那么方法和函数有什么区别呢？这是模板作者所关心的，它们没有关系，但不是一点没有。方法是来自于数据模型（它们反射了 **Java** 对象的方法），而函数是定义在模板内的（使用了函数指令-这也是高级主题），但二者可以用同一种方式来使用。

## 2.2.4.2 用户自定义指令

用户自定义指令（用自定义名称作为 **FreeMarker** 的标签）这种类型的值也是一种子程序，一种可以复用的模板代码段。但这也是高级主题，我们在后续章节中会详细解释。

对于程序员来说：用户自定义指令（比如宏），也是第一类值，就像函数/方法一样。

这里仅仅对用户自定义指令有一个认识即可（如果现在还不能理解可以先忽略它）。假设现在有一个变量，`box`，它的值是用户自定义的指令，用来打印一些特定的 **HTML** 信息，这个指令定义了一个标题和其中的信息。

```
<@box title="Attention!">  
  Too much copy-pasting may leads to  
  maintenance headaches.  
</@box>
```

## 2.2.4.3 函数/方法和用户自定义指令的比较

这个内容也是对高级用户来说的（如果你还不能理解可以先忽略这部分）。如果要使用函数/方法或自定义指令去实现一些东西的时候，二者之间的选择是两难的。按经验来说，如果能够实现，请用自定义指令而不用函数/方法。

- 输出（返回值）的是标记（**HTML**,**XML** 等）。主要原因是函数的返回结果可以自动进行 **XML** 转义（这是因为 `${...}` 的特性），而用户自定义指令的输出则不是（这是因为 `<@...>` 的特性所致，它的输出假定为是标记，因此就不再转义）。
- 副作用是重要的，而不是返回值。例如一个指令的目的是往服务器日志中添加一个

条目。（事实上你不能得到自定义指令的返回值，但有些反馈的类型是有可能设置非本地变量的）

- 会进行流程的控制（就像 `list` 或 `if` 指令那样），但是不能在函数/方法上这么做。

FreeMarker 不知道的 Java 对象的方法通常在模板中是可以作为方法来使用的，而不用考虑 Java 对象方法本身的特性，因为在这里没有其他的选项。

## 2.2.5 其它

### 2.2.5.1 节点

节点变量代表了树状结构中的一个节点，而且通常是配合 XML 来处理的，这是专业且高级的主题。

这里我们仅对高级用户进行一个概览：节点和存储在其他节点中的序列很相似，通常也被当作为子节点。节点存储它所在容器节点的引用，也就是父节点。节点的主要作用是拓扑信息。其他数据必须通过使用多类型的值来存储。就像一个值可以同时是一个节点和一个数字，这样它存储的数字可以作为支付额来使用。除了拓扑信息，节点也可以存储一些元信息：如节点名称，类型（字符串），命名空间（字符串）。若一个节点象征 XHTML 文档中的 `h1` 元素，那么它的名字可以是 `"h1"`，类型可以是 `"element"`，命名空间可以是 `"http://www.w3.org/1999/xhtml"`。但对于数据模型设计者来说，如果这些元信息还有如何来使用有什么意义呢。检索拓扑信息和元信息的方法将会在后续章节中来说明（这里你可以先不用理解）。

## 第三章 模板

章节内容如下：

总体结构

指令

表达式

插值

注意：

这里假设你已经阅读完入门章节和数值，类型章节了。

### 3.1 总体结构

实际上你用程序语言编写的程序就是模板，模板也被称为 **FTL**（代表 FreeMarker 模板语言）。这是为编写模板设计的非常简单的编程语言。

模板（FTL 编程）是由如下部分混合而成的：

**Text 文本：**文本会照着原样来输出。

**Interpolation 插值：**这部分的输出会被计算的值来替换。插值由 `${` 和 `}` 所分隔（或者 `#{` 和 `}`，这种已经不建议再使用了）。

**FTL tags 标签：**FTL 标签和 HTML 标签很相似，但是它们却是给 FreeMarker 的指示而且不会打印在输出内容中。

**Comments 注释：**注释和 HTML 的注释也很相似，但它们是由 `<#--` 和 `-->` 来分隔的。注释会被 FreeMarker 所忽略，更不会在输出内容中显示。

我们来看一个具体的模板，其中的内容已经用颜色来标记了：**文本**，**插值**，**FTL 标签**，**注释**，为了可视地看到换行符而使用了 `[BR]`。

```
<html> [BR]
<head> [BR]
  <title>Welcome!</title> [BR]
</head> [BR]
<body> [BR]
  <#-- Greet the user with his/her name --> [BR]
  <h1>Welcome ${user}!</h1> [BR]
  <p>We have these animals: [BR]
  <ul> [BR]
    <#list animals as being> [BR]
      <li>${being.name} for ${being.price} Euros [BR]
    </#list> [BR]
  </ul> [BR]
</body> [BR]
</html>
```

FTL 是区分大小写的。`list` 是指令的名称而 `List` 就不是，类似地 `${name}` 和



`${Name}` 或者 `${NAME}` 也是不同的。

应该意识到非常重要的一点：**插值**仅仅可以在**文本**中间使用（也可以在字符串表达式中，后续介绍）。

**FTL 标签**不可以在其他 **FTL 标签**和**插值**中使用。下面这样写就是**错的**：

```
<#if <#include 'foo'>='bar'>...</#if>
```

**注释**可以放在 **FTL 标签**和**插值**中间。比如：

```
<h1>Welcome ${user <!-- The name of user -->}}!</h1> [BR]
<p>We have these animals: [BR]
<ul> [BR]
<#list <!-- some comment... --> animals as <!-- again... -->
being> [BR]
...

```

注意：

如果您已经自己尝试了上面所有的示例的话，那么你也许会注意一些空格、制表符和换行符从模板输出中都不见了，尽管我们之前已经说了文本是按照原样输出的。现在不用为此而计较，这是由于“空格剥离”特性在起作用，它当然会自动去除一些多余的空格，制表符和换行符了。这个特性后续也会解释到。

## 3.2 指令

使用 FTL 标签来调用 **directives 指令**，比如调用 `list` 指令。在语法上我们使用了两个标签：`<#list animals as being>`和`</#list>`。

标签分为两种：

- 开始标签：`<#directivename parameters>`
- 结束标签：`</#directivename>`

除了标签以`#`开头外，其他都和 HTML，XML 的语法很相似。如果标签没有嵌套内容（在开始标签和结束标签之间的内容），那么可以只使用开始标签。例如`<#if something>...</#if>`，但是 FreeMarker 知道`<#include something>`中 `include` 指令没有可嵌套的内容。

`parameters` 的格式由 `directivename` 来决定。

事实上，指令有两种类型：预定义指令和用户自定义指令。对于用户自定义的指令使用 `@`来代替`#`，比如`<@mydirective parameters>...</@mydirective>`。更深的区别在于如果指令没有嵌套内容，那么必须这么使用`<@mydirective parameters />`，这和 XML 语法很相似（例如`<img ... />`）。但是用户自定义指令是后面要讨论的高级主题。

像 HTML 标签一样，FTL 标签必须正确的嵌套使用。下面这段示例代码就是错的，因为 `if` 指令在 `list` 指令嵌套内容的内外都有：

```

<ul>
<#list animals as being>
  <li>${being.name} for ${being.price} Euros
  <#if user == "Big Joe">
    (except for you)
</#list> <!-- WRONG! The "if" has to be closed first. -->
</#if>
</ul>

```

注意一下 FreeMarker 仅仅关心 FTL 标签的嵌套而不关心 HTML 标签的嵌套，它只会把 HTML 看做是相同的文本，不会来解释 HTML。

如果你尝试使用一个不存在的指令（比如你输错了指令的名称），FreeMarker 就会拒绝执行模板，同时抛出错误信息。

FreeMarker 会忽略 FTL 标签中的多余空白标记，所以你也可以这么来写代码：

```

<#list [BR]
  animals      as [BR]
    being [BR]
> [BR]
${being.name} for ${being.price} Euros [BR]
</#list >

```

当然，也不能在<，</和指令名中间插入空白标记。

指令列表和详细介绍可以参照指令参考部分（但是我建议先看表达章节）。

注意：

通过配置后，FreeMarker 可以在 FTL 标签和 FTL 注释中，使用 [ 和 ] 来代替<和>，就像 [#if user == "Big Joe"] ... [/#if]。然而我们不建议这样来使用。要获取更多信息，请参考：废弃 deepFTL 结构/老实 FTL 语法。

## 3.3 表达式

### 3.3.1 简介

当需要给插值或者指令参数提供值时，可以使用变量或其他复杂的表达式。例如，我们设 x 为 8，y 为 5，那么  $(x+y)/2$  的值就会被处理成数字类型的值 6.5

在我们展开细节之前，先来看一些具体的例子：

- 当给插值提供值时：插值的使用方式为  $\${expression}$ ，把它放到你想输出文本的位置上然后给值就可以打印了。即  $\${(5+8)/2}$  会打印“6.5”出来（如果输出的语言不是英语，也可能打印出“6,5”）。
- 当给指令参数提供值时：在入门章节我们已经看到 if 指令的使用了。这个指令的语法是：<#if expression>...</#if>。这里的表达式计算结果必须是布尔类型的。比如<#if 2 < 3>中的 2 < 3（2 小于 3）是结果为 true 的布尔表达式。

### 3.3.2 快速浏览（备忘单）

这里是给已经了解 FreeMarker 的人或有经验的程序员的一个提醒：

- 直接指定值
  - ◆ 字符串: `"Foo"` 或者 `'Foo'` 或者 `"It's \"quoted\""` 或者 `r"C:\raw\string"`
  - ◆ 数字: `123.45`
  - ◆ 布尔值: `true, false`
  - ◆ 序列: `["foo", "bar", 123.45], 1..100`
  - ◆ 哈希表: `{"name": "green mouse", "price": 150}`
- 检索变量
  - ◆ 顶层变量: `user`
  - ◆ 从哈希表中检索数据: `user.name, user["name"]`
  - ◆ 从序列中检索: `products[5]`
  - ◆ 特殊变量: `.main`
- 字符串操作
  - ◆ 插值（或连接）: `"Hello ${user}!"`（或 `"Free" + "Marker"`）
  - ◆ 获取一个字符: `name[0]`
- 序列操作
  - ◆ 连接: `users + ["guest"]`
  - ◆ 序列切分: `products[10..19]` 或 `products[5..]`
- 哈希表操作
  - ◆ 连接: `passwords + {"joe": "secret42"}`
- 算数计算: `(x * 1.5 + 10) / 2 - y % 100`
- 比较: `x == y, x != y, x < y, x > y, x >= y, x <= y, x < y,` 等等
- 逻辑操作: `!registered && (firstVisit || fromEurope)`
- 内建函数: `name?upper_case`
- 方法调用: `repeat("What", 3)`
- 处理不存在的值
  - ◆ 默认值: `name! "unknown"` 或者 `(user.name)! "unknown"` 或者 `name! 或者 (user.name)!`
  - ◆ 检测不存在的值: `name??` 或者 `(user.name)??`

参考：运算符的优先级

### 3.3.3 直接确定值

通常我们喜欢是使用直接确定的值而不是计算的结果。

#### 3.3.3.1 字符串

在文本中确定字符串值的方法是看引号和单引号，比如 `"some text"` 或 `'some`

`text'`，这两种形式是相等的。如果文本本身包含用于字符引用的引号（双引号`"`或单引号`'`）或反斜杠时，应该在它们的前面再加一个反斜杠，这就是转义。转义允许你直接在文本中输入任何字符，包括反斜杠。例如：

```
${"It's \"quoted\" and  
this is a backslash: \\"}  
  
${'It\'s "quoted" and  
this is a backslash: \\'}
```

输出为：

```
It's "quoted" and  
this is a backslash: \  
  
It's "quoted" and  
this is a backslash: \
```

注意：

这里当然可以直接在模板中输入文本而不需要`${...}`。但是我们在这里用它只是为了示例来说明表达式的使用。

下面的表格是 **FreeMarker** 支持的所有转义字符。在字符串使用反斜杠的其他所有情况都是错误的，运行这样的模板都会失败。

转义序列	含义
<code>\</code>	引号（u0022）
<code>\'</code>	单引号（又称为撇号）（u0027）
<code>\\</code>	反斜杠（u005C）
<code>\n</code>	换行符（u000A）
<code>\r</code>	回车（u000D）
<code>\t</code>	水平制表符（又称为标签）（u0009）
<code>\b</code>	退格（u0008）
<code>\f</code>	换页（u000C）
<code>\l</code>	小于号：<
<code>\g</code>	大于号：>
<code>\a</code>	和号：&
<code>\xCode</code>	字符的 16 进制 Unicode 码（UCS 码）

在`\x`之后的 *Code* 是 1-4 位的 16 进制码。下面这个示例中都是在字符串中放置版权符号`"\xA9 1999-2001"`，`"\x0A9 1999-2001"`，`"\x00A9 1999-2001"`：如果紧跟 16 进制码后一位的字符也能解释成 16 进制码时，就必须把 4 位补全，否则 **FreeMarker** 解析就会出现错误。

注意字符序列`${(#)}`有特殊的含义，它们被用做插入表达式的数值（典型的应用是：`"Hello ${user}!"`）。这将在后续章节中解释。如果想要打印`${}`，就要使用下面所说的原生字符串。

一种特殊的字符串就是原生字符串。在原生字符串中，反斜杠和`${}`没有特殊的含义，它们被视为普通的字符。为了表明字符串是原生字符串，在开始的引号或单引号之前放置字母 `r`，例如：

```
${r"${foo}"}  
${r"C:\foo\bar"}
```

将会打印：

```
${foo}  
C:\foo\bar
```

### 3.3.3.2 数字

输入不带引号的数字就可以直接指定一个数字，必须使用点作为小数的分隔符而不能是其他的分组分隔符。可以使用`-`或`+`来表明符号（`+`是多余的）。科学记数法暂不支持使用（`1E3`就是错误的），而且也不能在小数点之前不写`0`（`.5`也是错误的）。

下面的数字都是合法的：`0.08`, `-5.013`, `8`, `008`, `11`, `+11`。

数值文字`08`, `+8`, `8.00`和`8`是完全相等的，它们都是数字`8`。因此`${08}`, `${+8}`, `${8.00}`和`${8}`打印的都是相同的。

### 3.3.3.3 布尔值

直接写`true`或`false`就表征一个布尔值了，不需使用引号。

### 3.3.3.4 序列

指定一个文字的序列，使用逗号来遍历子变量，然后把整个列表放到方括号内。例如：

```
<#list ["winter", "spring", "summer", "autumn"] as x>  
${x}  
</#list>
```

将会打印出：

```
winter  
spring  
summer  
autumn
```

列表中的项目是表达式，那么也可以这样做：`[2 + 2, [1, 2, 3, 4], "whatnot"]`，其中第一个子变量是数字`4`，第二个子变量是一个序列，第三个子变量是字符串`"whatnot"`。

也可以用`start..end`定义存储数字范围的序列，这里的`start`和`end`是处理数值表达式，比如`2..5`和`[2, 3, 4, 5]`是相同的，但是使用前者会更有效率（小内存占用而且速度快）。可以看出前者也没有使用方括号，这样也可以用来定义递减的数字范围，比如`5..2`。（此外，还可以省略`end`，只需`5..`即可，但这样序列默认包含`5,6,7,8`等递增量直到无穷大）

### 3.3.3.5 哈希表

在模板中指定一个哈希表，就可以遍历用逗号来分隔的“键/值”对，把列表放到花括号内。键和值成对出现并以冒号分隔。看这个例子：`{"name":"green mouse", "price":150}`。注意到名字和值都是表达式，但是用来检索的名字就必须是字符串类型的。

### 3.3.4 检索变量

#### 3.3.4.1 顶层变量

为了访问顶层的变量，可以简单地使用变量名。例如，用表达式 `user` 就可以在根上获取以“user”为名存储的变量值。然后就可以打印出所存储在里面的内容。

```
${user}
```

如果没有顶层变量，那么 **FreeMarker** 在处理表达式时就会发生错误的结果，进而终止模板的执行（除非程序员事先配置了 **FreeMarker**）。

在这个表达式中变量名可以包含字母（也可以是非拉丁文），数字（也可以是非拉丁数字），下划线(\_)，美元符号(\$)，at 符号(@)和哈希表(#)，此外要注意变量名不能以数字开头。

#### 3.3.4.2 从哈希表中检索数据

如果有一个表达式的结果是哈希表，那么我们可以使用点和子变量的名字得到它的值，假设我们有如下的数据模型：

```
(root)
|
+- book
|  |
|  +- title = "Breeding green mice"
|  |
|  +- author
|      |
|      +- name = "Julia Smith"
|      |
|      +- info = "Biologist, 1923-1985, Canada"
|
+- test = "title"
```

现在，就可以通过 `book.title` 来读取 `title` 表达式 `book` 将返回一个哈希表（就像上一章中解释的那样）。按这种逻辑进一步来说，我们可以使用表达式 `book.author.name` 来读取到 `author` 的 `name`。

如果我们想指定同一个表达式的子变量，那么还有另外一种语法格式：`book["title"]`。在方括号中可以给出任意长度字符串的表达式。在上面这个数据模型示例中你还可以这么来获取 `title: book[test]`，下面这些示例它们含义都是相等的：`book.author.name`, `book["author"].name`, `book.author["name"]`, `book["author"]["name"]`。

当使用点式语法时，顶层变量名的命名也有相同的限制（命名时智能使用字母，数字，下划线，`$`，`@`等），而使用方括号语法形式时命名有没有这样的限制，它可以是任意的表达式。（为了 FreeMarker 支持 XML，如果变量名是\*（星号）或者\*\*，那么就应该使用方括号语法格式。）

对于顶层变量来说，如果尝试访问一个不存在的变量也会引起错误导致模板解析执行的中断（除非程序员事先配置过 FreeMarker）。

### 3.3.4.3 从序列中检索数据

这和从哈希表中检索是相同的，但是你能只能使用方括号语法形式来进行，而且方括号内的表达式最终必须是一个数字而不是字符串。在第一章的数据模型示例中，为了获取第一个动物的名字（记住第一项数字索引是 0 而不是 1）可以这么来写：`animals[0].name`。

### 3.3.4.4 特殊变量

特殊变量是由 FreeMarker 引擎本身定义的，为了使用它们，可以按照如下语法形式来进行：`.variable_name`。

通常情况下是不需使用特殊变量，而对专业用户来说可能用到。所有特殊变量的说明可以参见参考手册。

## 3.3.5 字符串操作

### 3.3.5.1 插值（或连接）

如果要在字符串中插入表达式的值，可以在字符串的文字中使用 `${...}`（`#{...}`）。`${...}` 的作用和在文本区的是相同的。假设用户是“Big Joe”，看下面的代码：

```
${"Hello ${user}!"}  
${"${user}${user}${user}${user}"}
```

将会打印如下内容：

```
Hello Big Joe!  
Big JoeBig JoeBig JoeBig Joe
```

另外，也可以使用`+`号来达到类似的效果，这是比较老的方法，也叫做字符串连接。示例如下：

```
${"Hello " + user + "!"}  
${user + user + user + user}
```

这样打印的效果和多次使用`${...}`是一样的。

警告：

使用者在使用插值时经常犯的一个错误是：在不能使用插值的地方使用了它。插值只能在文本区段（`<h1>Hello ${name}!</h1>`）和字符串文字（`<#include "/footer/${company}.html">`）中使用。一个典型的错误使用是`<#if ${isBig}>Wow!</#if>`，这是语法上的错误。只能这么来写：`<#if isBig>Wow!</#if>`，`<#if "${isBig}">Wow!</#if>`来写也是错误的。因为`if`指令的参数需要的是布尔值，而这里是字符串，那么就会引起运行时错误。

### 3.3.5.2 获取一个字符

在给定索引值时可以获取字符串中的一个字符，这和 3.3.4.3 中从序列中检索数据是相似的，比如 `user[0]`。这个操作执行的结果是一个长度为 1 的字符串，FTL 并没有独立的字符类型。和序列中的子变量一样，这个索引也必须是数字，范围是从 0 到字符串的长度，否则模板的执行将会发生错误并终止。

由于序列的子变量语法和字符的 `getter` 语法冲突，那么只能在变量不是序列时使用字符的 `getter` 语法（因为 FTL 支持多类型值，所以它是可能的），这种情况下使用序列方式就比较多。（为了变通，可以使用内建函数 `string`，比如 `user?string[0]`。不必担心你不理解这个含义，内建函数将会在后续章节中讨论。）

看一个例子（假设 `user` 是“Big Joe”）

```
${user[0]}  
${user[4]}
```

将会打印出（注意第一个字符的索引是 0）：

```
B  
J
```

注意：

可以按照获切分序列的方式来获取一定范围内的字符，比如`${user[1..4]}`和`${user[4..]}`。然而现在这种使用方法已经被废弃了，作为它的替代，可以使用内建函数 `substring`，内建函数将会在后续章节中讨论。

## 3.3.6 序列操作

### 3.3.6.1 连接

序列的连接可以使用`+`号来进行，例如：



```
<#list ["Joe", "Fred"] + ["Julia", "Kate"] as user>
- ${user}
</#list>
```

将会打印出:

```
- Joe
- Fred
- Julia
- Kate
```

要注意不要在很多重复连接时使用序列连接操作, 比如在循环中往序列上追加项目, 而这样的使用是可以的: `<#list users + admins as person>`。尽管序列连接的很快, 而且速度是和被连接序列的大小相独立的, 但是最终的结果序列的读取却比原先的两个序列慢那么一点。通过这种方式进行的许多重复连接最终产生的序列读取的速度会慢。

### 3.3.6.2 序列切分

使用 `[firstindex..lastindex]` 可以获取序列的一份, 这里的 `firstindex` 和 `lastindex` 的表达式结果是数字。如果 `seq` 存储序列 "a", "b", "c", "d", "e", "f", 那么表达式 `seq[1..4]` 将会是含有 "b", "c", "d", "e" 的序列 (索引为 1 的项是 "b", 索引为 4 的项是 "e")。

`lastindex` 可以被省略, 那么这样将会读取到序列的末尾。如果 `seq` 存储序列 "a", "b", "c", "d", "e", "f", 那么 `seq[3..]` 将是含有 "d", "e", "f" 的序列。

注意:

从 FreeMarker 2.3.3 版本以后 `lastindex` 才能省略。

如果试图访问一个序列首变量之前的项或末变量之后的项将会引起错误, 模板的执行也会中断。

### 3.3.7 哈希表操作

#### 3.3.7.1 连接

像连接字符串那样, 也可以使用 `+` 号的方式来连接哈希表。如果两个哈希表含有键相同的项, 那么在 `+` 号右侧的哈希表中的项目优先。例如:

```
<#assign ages = {"Joe":23, "Fred":25} + {"Joe":30, "Julia":18}>
- Joe is ${ages.Joe}
- Fred is ${ages.Fred}
- Julia is ${ages.Julia}
```

将会打印出:

```
- Joe is 30
- Fred is 25
- Julia is 18
```

注意很多项目连接时不要使用哈希表连接，比如在循环时往哈希表中添加新项。这和序列连接的情况是一致的。

### 3.3.8 算数计算

算数计算包含基本的四则运算和求模运算，运算符有：

- 加法：`+`
- 减法：`-`
- 乘法：`*`
- 除法：`/`
- 求模（求余）：`%`

示例如下：

```
${100 - x * x}
${x / 2}
${12 % 10}
```

假设 `x` 是 5，就会打印出：

```
75
2.5
2
```

要保证两个操作数都是结果值为数字的表达式。下面的这个例子在运行时，FreeMarker 就会发生错误，因为字符串 `"5"` 不是数字 5。

```
${3 * "5"} <#-- WRONG! -->
```

但这种情况也有一个例外，就是 `+` 号，它是用来连接字符串的，如果 `+` 号的一端是字符串，另外一端是数字，那么数字就会自动转换为字符串类型（使用适当的格式）。示例如下：

```
${3 + "5"}
```

将会输出：

```
35
```

通常来说，FreeMarker 不会自动字符串转换为数字，反之会自动进行。

有时我们只想获取计算结果的整数部分，这可以使用内建函数 `int` 来解决。（关于内建函数后续章节会来解释）

```

${(x/2)?int}
${1.1?int}
${1.999?int}
${-1.1?int}
${-1.999?int}

```

仍然假设 `x` 的值是 5，那么将会输出：

```

2
1
1
-1
-1

```

### 3.3.9 比较

有时我们需要知道两个值是否相等，或者那个值更大一点。

为了演示具体的例子，我们在这里使用 `if` 指令。`if` 指令的用法是：`<#if expression>...</#if>`，其中的表达式的值必须是布尔类型，否则将会出错，模板执行中断。如果表达式的结果是 `true`，那么在开始和结束标记内的内容将会被执行，否则就会被跳过。

测试两个值相等使用 `=`（或者采用 Java 和 C 语言中的 `==`，二者是完全等同的。）

测试两个值不等使用 `!=`。例子中假设 `user` 是 "Big Joe"。

```

<#if user = "Big Joe">
  It is Big Joe
</#if>
<#if user != "Big Joe">
  It is not Big Joe
</#if>

```

`<#if ...>` 中的表达式 `user = "Big Joe"` 结果是布尔值 `true`，上面的代码将会输出 "It is Big Joe"。

`=` 或 `!=` 两边的表达式的结果都必须是标量，而且两个标量都必须是相同类型（也就是说字符串智能和字符串来比较，数字只能和数字来比较等）。否则将会出错，模板执行中断。例如 `<#if 1 = "1">` 就会导致错误。注意 `FreeMarker` 进行的是精确的比较，所以字符串在比较时要注意大小写和空格：`"x"`，`"x "` 和 `"X"` 是不同的值。

对数字和日期值的比较，也可以使用 `<`，`<=`，`>=` 和 `>`。不能把它们当作字符串来比较。比如：

```

<#if x <= 12>
  x is less or equivalent with 12
</#if>

```

使用 `>=` 和 `>` 的时候有一小问题。`FreeMarker` 解释 `>` 的时候可以把它当作 FTL 标签的结

束符。为了避免这种问题，不得不将表达式放到括号内：`<#if (x > y)>`，或者可以在比较关系处使用`&gt;`和`&lt;`：`<#if x &gt; y>`。（通常在 `FLT` 标签中不支持实体引用（比如`&...;`这些），否则就会抛出算数比较异常）。另外，可以使用 `lt` 代替`<`，`lte` 代替`<=`，`gt` 代替`>`，`gte` 代替`>=`，由于历史遗留的原因，`FTL` 也支持`\lt`，`\lte`，`\gt` 和`\gte`，他们和使用不带反斜杠的效果一样。

### 3.3.10 逻辑操作

常用的逻辑操作符：

- 逻辑或：`||`
- 逻辑与：`&&`
- 逻辑非：`!`

逻辑操作符仅仅在布尔值之间有效，若用在其他类型将会产生错误导致模板执行中止。

例如：

```
<#if x < 12 && color = "green">
    We have less than 12 things, and they are green.
</#if>
<#if !hot> <#-- here hot must be a boolean -->
    It's not hot.
</#if>
```

### 3.3.11 内建函数

正如其名，内建函数提供始终可用的内置功能。内建函数以`?`形式提供变量的不同形式或者其他信息。使用内建函数的语法和访问哈希表变量的语法很像，除了使用`?`号代替点，其他都一样。例如得到字符串的大写形式：`user?upper_case`。

在参考文档中可以查到所有内建函数的资料。现在，我们只需了解一些重要的内建函数就行了。

- 字符串使用的内建函数：
  - ◆ `html`: 字符串中所有的特殊 `HTML` 字符都需要用实体引用来代替（比如`<`代替`&lt;`）
  - ◆ `cap_first`: 字符串的第一个字母变为大写形式
  - ◆ `lower_case`: 字符串的小写形式
  - ◆ `upper_case`: 字符串的大写形式
  - ◆ `trim`: 去掉字符串首尾的空格
- 序列使用的内建函数：
  - ◆ `size`: 序列中元素的个数
- 数字使用的内建函数：
  - ◆ `int`: 数字的整数部分（比如`-1.9?int` 就是`-1`）

示例：

```
${test?html}
${test?upper_case?html}
```

假设字符串 `test` 存储“Tom & Jerry”，那么输出为：

```
Tom & Jerry
TOM & JERRY
```

注意 `test?upper_case?html`，内嵌函数双重使用，`test?upper_case` 的结果是字符串了，但也还可以继续在其后使用 `html` 内建函数。

另外一个例子：

```
${seasons?size}
${seasons[1]?cap_first} <!-- left side can by any expression -->
${"horse"?cap_first}
```

假设 `seasons` 存储了序列“winter”, “spring”, “summer”, “autumn”，那么上面的输出将会是：

```
4
Spring
Horse
```

### 3.3.12 方法调用

可以使用方法调用操作来使用一个已经定义的方法。方法调用的语法形式是使用逗号来分割在括号内的表达式而形成的参数列表，这些值就是参数。方法调用操作将这些值传递给方法，然后返回一个结果，这个结果就是整个方法调用表达式的值。

假设程序员定义了一个可供调用的方法 `repeat`。第一个参数字符串类型，第二个参数是数字类型。方法的返回值是字符串类型，而方法要完成是将第一个参数重复显示，显示的次数是第二个参数的值。

```
${repeat("What", 3)}
```

将会打印出：

```
WhatWhatWhat
```

### 3.3.13 处理不存在的值

要注意这个操作是用 FreeMarker 2.3.7 版本以后才有的(用来代替内建函数 `default`，`exists` 和 `if_exists`)。

正如我们前面解释的那样，当访问一个不存在的变量时 FreeMarker 将会报错导致模板执行中断。通常我们可以使用两个操作符来压制这个错误，控制错误的发生。被控制的变量可以是顶层变量，哈希表或序列的子变量。此外这些操作符还能处理方法调用的返回值不存在的情况(这点对 Java 程序员来说，返回值是 `null` 而不是返回值为 `void` 类型的方法)，

通常来说，我们应该使用这些操作符来控制可能不存在的变量。

对于知道 Java 中 `null` 的人来说，FreeMarker 2.3.x 版本把它们视为不存在的变量。简单地说，模板语言中没有 `null` 这个概念。比如有一个 `bean`，`bean` 中有一个 `maidenName` 属性，对于模板而言（假设你没有配置 FreeMarker 来使用一些极端的对象包装），这个属性的值是 `null`，和不存在这个属性的情况是一致的。调用方法的返回值如果是 `null` 的话 FreeMarker 也会把它当作不存在的变量来处理（假定你只使用了普通的对象包装）。了解更多可以参考 FAQ。

**注意：**

如果你想知道为什么 FreeMarker 对不存在的变量如此挑剔，请阅读 FAQ 部分。

### 3.3.13.1 默认值

使用形式概览：`unsafe_expr!default_expr` 或 `unsafe_expr!` 或 `(unsafe_expr)!default_expr` 或 `(unsafe_expr)!`

这个操作符允许你为可能不存在的变量指定一个默认值。

例如，假设下面展示的代码中没有名为 `mouse` 的变量：

```
{mouse!"No mouse."}
<#assign mouse="Jerry">
{mouse!"No mouse."}
```

将会输出

```
No mouse.
Jerry
```

默认值可以是任何类型的表达式，也可以不必是字符串。你也可以这么写：`hits!0` 或 `colors!["red", "green", "blue"]`。默认值表达式的复杂程度没有严格限制，你还可以这么来写：`cargo.weight!(item.weight * itemCount + 10)`。

**警告：**

如果在 `!` 后面有复合表达式，如 `1 + x`，通常使用括号，像 `{x!(1 + y)}` 或 `{(x!1) + y}`，这样就根据你的意思来确定优先级。由于 FreeMarker 2.3.x 版本的源码中的小失误所以必须这么来做。`!`（作为默认值操作）的优先级非常低。这就意味着 `{x!1 + y}` 会被 FreeMarker 误解为 `{x!(1 + y)}`，而真实的意义是 `{(x!1) + y}`。这个源码的错误在 FreeMarker 2.4 中会得到修正。在编程中注意这个错误，要么就使用 FreeMarker 2.4！

如果默认值被省略了，那么结果将会是空串，空序列或空哈希表。（这是 FreeMarker 允许许多类型值的体现）如果想让默认值为 `0` 或 `false`，则注意不能省略它。例如：

```
{mouse!})
<#assign mouse = "Jerry">
{mouse!})
```

输出为：

```
()
(Jerry)
```

### 警告：

因为语法的含糊`<@something a=x! b=y />`将会解释为`<@something a=x! (b=y) />`，那就是说`b=y`将会视为是比较运算，然后结果作为`x`的默认值。而不是想要的参数`b`是`x`的默认值，为了避免这种情况，如下编写代码：`<@something a=(x!) b=y />`。

在不是顶层变量时，默认值操作符可以有两种使用方式：

```
product.color!"red"
```

如果这样的写法，那么`product`中当`color`不存在时(返回`"red"`)将会被处理，但是如果连`produce`都不存在时将不会处理。也就是说这样写时变量`product`必须存在，否则模板就会报错。

```
(product.color)"red"
```

这时，如果当不存在时也会被处理，那就是说如果`product`不存在或者`product`存在而`color`不存在，都能显示默认值`"red"`而不会报错。本例和上例写法的重要区别在于用括号时，就允许其中表达式的任意部分可以未定义。

当然，默认值操作也可以作用于序列，比如：

```
<#assign seq = ['a', 'b']>
${seq[0]!"-"}
${seq[1]!"-"}
${seq[2]!"-"}
${seq[3]!"-"}

```

输出为：

```
a
b
-
-
```

如果序列索引是负数（比如`seq[-1]!"-"`）也会发生错误，这样默认值和其他操作也就不起作用了。

## 3.3.13.2 检测不存在的值

使用形式概览：`unsafe_expr??`或`(unsafe_expr)??`

这个操作符告诉我们一个值是否存在。基于这种情况，结果是`true`或`false`。

示例如下，假设并没有名为`mouse`的变量：

```

<#if mouse??>
    Mouse found
<#else>
    No mouse found
</#if>
Creating mouse...
<#assign mouse = "Jerry">
<#if mouse??>
    Mouse found
<#else>
    No mouse found
</#if>

```

输出为:

```

No mouse found
Creating mouse...
Mouse found

```

访问非顶层变量的使用规则和默认值操作符也是一样的，即 `product.color??` 和 `(product.color)??`。

### 3.3.14 括号

括号可以用来给表达式分组。示例如下：

<code>\${3 * 2 + 2}</code>	<code>&lt;!-- 输出是: --&gt;</code>
<code>\${3 * (2 + 2)}</code>	<code>&lt;!-- 8 --&gt;</code>
<code>\${3 * ((2 + 2) * (1 / 2))}</code>	<code>&lt;!-- 12 --&gt;</code>
<code>\$("#green " + "mouse"?upper_case}</code>	<code>&lt;!-- 6 --&gt;</code>
<code>\${("green " + "mouse")?upper_case}</code>	<code>&lt;!-- green MOUSE --&gt;</code>
<code>&lt;#if !( color = "red"    color = "green")&gt;</code>	<code>&lt;!-- GREEN MOUSE --&gt;</code>
<code>    The color is nor red nor green</code>	
<code>&lt;/#if&gt;</code>	

别忘了方法调用时使用的括号和给表达式分组的括号含义是完全不同的。

### 3.3.15 表达式中的空格

FTL 忽略表达式中的多余空格，下面两种表示是相同的：

```

${x + ":" + book.title?upper_case}

```

和



```
${x+"."+book.title?upper_case}
```

还有

```
${
    x
+ ":" + book . title
? upper_case
}
```

### 3.3.16 操作符的优先级

下面的表格显示了已定义操作符的优先级。表格中的运算符按照优先程度降序排列：上面的操作符优先级高于它下面的。高优先级的运算符执行要先于优先级比它低的。表格同一行上的两个操作符优先级相同。当有相同优先级的二元运算符（运算符有两个参数，比如+和-）挨着出现时，它们按照从左到右的原则运算。

运算符组	运算符
最高优先级运算符	<code>[subvarName] [subStringRange] .? (methodParams) expr! expr??</code>
一元前缀运算符	<code>+expr -expr !expr</code>
乘除法，求模	<code>* / %</code>
加减法	<code>+ -</code>
关系运算符	<code>&lt; &gt; &lt;= &gt;=</code> (相当于: <code>gt, lt</code> , 等)
相等，不等	<code>==</code> (也可以是: <code>=</code> ) <code>!=</code>
逻辑与	<code>&amp;&amp;</code>
逻辑或	<code>  </code>
数字范围	<code>..</code>

如果你熟悉 C 语言，Java 语言或 JavaScript 语言，注意 FreeMarker 中的优先级规则和它们是相同的，除了那些只有 FTL 本身含有的操作符。

因为编程的失误，默认值操作符（`exp!exp`）不在表格中，按照向后兼容的原则，在 FreeMarker 2.4 版本中将会修正它。而且它将是最高优先级的运算符，但是在 FreeMarker 2.3.x 版本中它右边的优先级由于失误就非常低。所以在默认值操作符的右边中使用复杂表达式时可以使用括号，可以是 `x!(y + 1)` 或 `(x!y) + 1`，而不能是 `x!y + 1`。

## 3.4 插值

插值的使用语法是：`${expression}`，`expression` 可以是所有种类的表达式（比如 `${100 + x}`）。

插值是用来给插入具体值然后转换为文本（字符串）。插值仅仅可以在两种位置使用：文本区（如 `<h1>Hello ${name}!</h1>`）和字符串表达式（如 `<#include "/footer/${company}.html">`）

**警告：**

一个常犯的错误是在不能使用插值的地方使用了它。典型的错误就是`<#if ${isBig}>Wow!</#if>`，这是语法上的错误。只要写为`<#if isBig>Wow!</#if>`就对了，而且`<#if "${isBig}">Wow!</#if>`也是错误的，因为这样参数就是字符串类型了，但是 `if` 指令的参数要求是布尔值，所以就会发生运行时错误。

插值表达式的结果必须是字符串，数字或日期类型的，因为只有数字和日期类型可以自动转换为字符串类型，其他类型的值（如布尔，序列）只能手动转换为字符串类型，否则就会发生错误导致模板执行中止。

如果插值在文本区（不是在字符串表达式内），那么如果 `escape` 指令起作用的话，插入的字符串将会自动消失。

#### 数字插入指南

如果表达式是数字类型的，那么根据默认数字格式，数值将会转换成字符串类型。这也许会包含最大小数问题，数字分组问题和相似处理问题。通常程序员应该设置默认的数字格式，而模板设计者不需要处理它（但是可以使用 `number_format` 设置来进行，详情参考 `setting` 指令部分的文档）。你可以使用内建函数 `string` 为一个插值来重写默认数值格式。

小数的分隔符通常（其他类似的符号也是这样，如分组符号）是根据所在地的标准（语言，国家）来确定的，这也需要程序员来设置。例如这个模板：

```
${1.5}
```

如果当前地区设置为英国时，将会打印：

```
1.5
```

而如果当前的地区设置为匈牙利时，将会打印：

```
1,5
```

因为匈牙利人使用逗号作为小数点。

可以使用内建函数 `string` 为单独的插值来修改设置。

#### 警告：

可以看出，插值的打印都是给人们看的（），而不是给计算机。有时候这样并不好，比如你要打印数据库记录的主键，用来作为 URL 的一部分或 HTML 表单的隐藏域，或者要打印 CSS/JavaScript 数字。这些值都是给计算机程序去识别的而不是人，很多程序对数字格式的要求非常严格，只能理解一部分简单的美式数字格式。那样的话，可以使用内建函数 `c`（代表计算机）来解决这个问题，比如：

```
<a href="/shop/productdetails?id=${product.id?c}">Details...</a>
```

#### 日期/时间插入指南

如果表达式的值是日期类型，那么日期中的数字将会按照默认格式来转换成文本。通常程序员应该设置默认格式，而页面设计者无需处理这点（如果需要，可以参考 `date_format`, `time_format` 和 `datetime_format` 的 `setting` 指令的设置）

可以使用内建函数 `string` 为单独的插值重写默认格式。

#### 警告：

为了将日期展示成文本，FreeMarker 必须知道日期中的哪一部分在使用，也就是说，如果仅仅日期部分（年，月，日）或仅仅时间部分（时，分，秒，毫秒）或两部分。不幸的是，由于 Java 平台技术的限制，自动探测一些变量是不现实的。可以找程序员对数据模型中可

能出问题的变量进行处理。如果找出日期变量的哪部分在使用是不可能的话，就必须帮助 FreeMarker 使用内建函数 `date`，`time` 和 `datetime` 来识别，否则就会出现错误停止。

#### *布尔值插入指南*

若要使用插值方式来打印布尔值会引起错误，中止模板的执行。例如：`${a == 2}` 就会引起错误，不会打印“true”或其他内容。

然而，我们可以使用内建函数 `string` 来将布尔值转换为字符串形式。比如打印变量“married”（假设它是布尔值），那么可以这么来写：`${married?string("yes", "no")}`。

#### *精确的转换规则*

对于有兴趣研究的人，表达式的值转换为字符（仍有不存在的可能）串精确的规则就是下面这些，以这个顺序进行：

1. 如果这个值是数字，那么它会按照指定的 `number_format` 设置规则来转换为字符串。所以这些转换通常是对用户进行的，而不是对计算机。
2. 如果这个值是日期，时间或时间日期类型的一种，那么它们会按照指定的 `time_format`，`date_format` 和 `datetime_format` 设置规则来转换为字符串，这要看日期信息中是只包含时间，日期还是全包括了。如果它不能被探测出来是哪种日期类型（日期 VS 时间 VS 日期时间），就会发生错误了。
3. 如果值本来就是字符串类型的，不需要转换。
4. 如果 FreeMarker 引擎在传统兼容模式下：
  1. 如果值是布尔类型，那么就转换成“true”，false 值将会转换为空串。
  2. 如果表达式未被定义（`null` 或者变量未定义），那么就转换为空串。
  3. 否则就会发生错误中止模板执行。
5. 否则就会发生错误中止模板执行。

## 第四章 其它

章节内容如下：

- 自定义指令
- 在模板中定义变量
- 命名空间
- 空白处理
- 替换（方括号）语法

### 4.1 自定义指令

- 简介
- 基本内容
- 参数
- 嵌套内容
- 宏和循环变量
- 自定义指令和宏进阶

#### 4.1.1 简介

自定义指令可以使用 `macro` 指令来定义，这是模板设计者所关心的内容。Java 程序员若不想在模板中实现指令，而在 Java 语言中实现指令，可以使用 `freemarker.template.TemplateDirectiveModel` 类来扩展（后续章节将会说明）。

#### 4.1.2 基本内容

宏是有一个变量名的模板片段。你可以在模板中使用宏作为自定义指令，这样就能进行重复性的工作。例如，创建一个宏变量来打印大号的“Hello Joe!”。

```
<#macro greet>
  <font size="+2">Hello Joe!</font>
</#macro>
```

`macro` 指令自身不打印任何内容，它只是用来创建宏变量，所以就会有一个名为 `greet` 的变量。在 `<#macro greet>` 和 `</#macro>` 之间的内容（称为**宏定义体**）当使用它作为指令时将会被执行。你可以在 FTL 标记中通过 `@` 代替 `#` 来使用自定义指令。使用变量名作为指令名。而且，自定义指令的结束标记也是需要的。那么，就可以这样来使用 `greet` 了：

```
<@greet></@greet>
```

因为`<anything></anything>`和`<anything/>`是相同的,你也可以使用单标记形式(如果你了解 XML,那么就很容易理解了,它们是相似的):

```
<@greet/>
```

将会打印:

```
<font size="+2">Hello Joe!</font>
```

宏能做的还有很多,因为在`<#macro ...>`和`</#macro>`之间的东西是模板片段,也就是说它可以包含插值(`${...}`)和 FTL 标签(如`<#if ...>...</#if>`)。

注意:

程序员通常将使用`<@...>`称为宏调用。

### 4.1.3 参数

我们来改进 `greet` 宏使之可以使用任意名字,而不仅仅是“Joe”。为了实现这个目标,就要使用到参数。在 `macro` 指令中,宏名称的后面位置是用来定义变量的。这里我们仅在 `greet` 宏中定义一个变量, `person`:

```
<#macro greet person>
  <font size="+2">Hello ${person}!</font>
</#macro>
```

那么就可以这样来使用这个宏:

```
<@greet person="Fred"/> and <@greet person="Batman"/>
```

这和 HTML 的语法是很相似的,它会打印出:

```
<font size="+2">Hello Fred!</font>
and <font size="+2">Hello Batman!</font>
```

那么就看到了,宏参数的真实值是可以作为变量(`person`)放在宏定义体中的。使用预定义指令,参数的值(=号后边的值)可以是 FTL 表达式。这样,不像 HTML,“Fred”和“Batman”的引号就可以不用要了。`<@greet person=Fred/>`也意味着使用变量的值 `Fred` 作为 `person` 参数,而不是字符串“Fred”。当然参数值并不一定是字符串类型,也可以是数字,布尔值,哈希表,序列等...也可以在=号左边使用复杂表达式(比如 `someParam=(price + 50)*1.25`)。

自定义指令可以有多个参数。如下所示,再添加一个新参数 `color`:

```
<#macro greet person color>
  <font size="+2" color="${color}">Hello ${person}!</font>
</#macro>
```

那么,这个宏就可以这样来使用:

```
<@greet person="Fred" color="black"/>
```

参数的顺序不重要，下面这个和上面的也是相同的。

```
<@greet color="black" person="Fred"/>
```

当调用这个宏的时候，你仅仅可以使用在 `macro` 指令中定义的参数（这个例子中是：`person` 和 `color`）。那么当你尝试 `<@greet person="Fred" color="black" background="green"/>` 的时候就会发生错误，因为并没有在 `<#macro ...>` 中定义参数 `background`。

同时也必须给出在宏中定义所有参数的值。如果你尝试 `<@greet person="Fred"/>` 时也会发生错误，因为忘记指定 `color` 的值了。很多情况下需要给一个参数指定一个相同的值，所以我们仅仅想在这个值发生变化后重新赋给变量。那么要达到这个目的，在 `macro` 指令中必须这么来指定变量：`param_name=usual_value`。例如，当没有特定值的时候，我们想要给 `color` 赋值为 `"black"`，那么 `greet` 指令就要这么来写：

```
<#macro greet person color="black">
  <font size="+2" color="{color}">Hello ${person}!</font>
</#macro>
```

现在，我们这么使用宏就可以了：`<@greet person="Fred"/>`，因为它和 `<@greet person="Fred" color="black"/>` 是相同的，这样参数 `color` 的值就是已知的了。如果想给 `color` 设置为 `"red"`，那么就写成：，这时 `macro` 指令就会使用这个值来覆盖之前设置的通用值，参数 `color` 的值就会是 `"red"` 了。

根据 FTL 表达式规则，明白这一点是至关重要的，`someParam=foo` 和 `someParam="{foo}"` 是不同的。第一种情况，是把变量 `foo` 的值作为参数的值来使用。第二种情况则是使用插值形式的字符串，那么参数值就是字符串了，这个时候，`foo` 的值呈现为文本，而不管 `foo` 是什么类型（数字，日期等）的。看这个例子：`someParam=3/4` 和 `someParam="{3/4}"` 是不同的，如果指令需要 `someParam` 是一个数字值，那么就不要用第二种方式。切记不要改变这些。

宏参数的另外一个重要的方面是它们是局部变量。更多局部变量的信息可以阅读 4.2 节内容：在模板中定义变量。

## 4.1.4 嵌套内容

自定义指令可以嵌套内容，和预定义指令相似：`<#if ...>nested content</#if>`。比如，下面这个例子中是创建了一个可以为嵌套的内容画出边框：

```
<#macro border>
  <table border=4 cellpadding=4><tr><td>
    <#nested>
  </tr></td></table>
</#macro>
```

`<#nested>` 指令执行位于开始和结束标记指令之间的模板代码段。如果这样写：

```
<@border>The bordered text</@border>
```

那么就会输出：

```
<table border=4 cellspacing=0 cellpadding=4><tr><td>
  The bordered text
</td></tr></table>
```

`nested` 指令也可以多次被调用，例如：

```
<#macro do_thrice>
  <#nested>
  <#nested>
  <#nested>
</#macro>
<@do_thrice>
  Anything.
</@do_thrice>
```

就会输出：

```
Anything.
Anything.
Anything.
```

如果不使用 `nested` 指令，那么嵌套的内容就会被执行，如果不小心将 `greet` 指令写成了这样：

```
<@greet person="Joe">
  Anything.
</@greet>
```

FreeMarker 不会把它视为错误，只是打印：

```
<font size="+2">Hello Joe!</font>
```

嵌套的内容被忽略了，因为 `greet` 宏没有使用 `nested` 指令。

嵌套的内容可以是任意有效的 FTL，包含其他的用户自定义指令，这样也是对的：

```
<@border>
  <ul>
    <@do_thrice>
      <li><@greet person="Joe"/>
    </@do_thrice>
  </ul>
</@border>
```

将会输出：

```
<table border=4 cellspacing=0 cellpadding=4><tr><td>
  <ul>
    <li><font size="+2">Hello Joe!</font>
    <li><font size="+2">Hello Joe!</font>
    <li><font size="+2">Hello Joe!</font>
  </ul>
</tr></td></table>
```

在嵌套的内容中，宏的局部变量是不可见的。为了说明这点，我们来看：

```
<#macro repeat count>
  <#local y = "test">
  <#list 1..count as x>
    ${y} ${count}/${x}: <#nested>
  </#list>
</#macro>
<@repeat count=3>${y!"?"} ${x!"?"} ${count!"?"}</@repeat>
```

将会打印：

```
test 3/1: ? ? ?
test 3/2: ? ? ?
test 3/3: ? ? ?
```

因为 `y`，`x` 和 `count` 是宏的局部（私有）变量，从宏外部定义是不可见的。此外不同的局部变量的设置是为每个宏自己调用的，所以不会导致混乱：

```
<#macro test foo>${foo} (<#nested>) ${foo}</#macro>
<@test foo="A"><@test foo="B"><@test foo="C"/></@test></@test>
```

将会打印出：

```
A (B (C () C) B) A
```

## 4.1.5 宏和循环变量

像 `list` 这样的预定义指令可以使用循环变量，你可以通过阅读 4.2 节：在模板中定义变量来理解循环变量。

自定义指令也可以有循环变量。比如我们来扩展先前例子中的 `do_thrice` 指令，就可以暴露出当前的循环变量的值。而对于预定义指令（如 `list`）当使用指令（就像中的 `foo`）时，循环变量的名字是已经给定的，变量值的设置是由指令本身完成的。



```

<#macro do_thrice>
  <#nested 1>
  <#nested 2>
  <#nested 3>
</#macro>
<@do_thrice ; x> <!-- user-defined directive uses ";" instead
of "as" -->
  ${x} Anything.
</@do_thrice>

```

将会输出：

```

1 Anything.
2 Anything.
3 Anything.

```

语法规则是为特定的循环（也就是嵌套内容的重复）传递循环变量的真实值来作为 `nested` 指令（当然参数可以是任意的表达式）的参数。循环变量的名称是在自定义指令的开始标记（`<@...>`）的参数后面通过分号确定的。

一个宏可以使用多个循环变量（变量的顺序是很重要的）：

```

<#macro repeat count>
  <#list 1..count as x>
    <#nested x, x/2, x==count>
  </#list>
</#macro>
<@repeat count=4 ; c, halfc, last>
  ${c}. ${halfc}<#if last> Last!</#if>
</@repeat>

```

那么，将会输出：

```

1. 0.5
2. 1
3. 1.5
4. 2 Last!

```

在自定义指令的开始标签（分号之后）为循环变量指定不同的数字是没有问题的，而不能在 `nested` 指令上使用。如果在分号之后指定的循环变量少，那么就看不到 `nested` 指令提供的最后的值，因为没有循环变量来存储这些值，下面的这些都是可以的：

```
<@repeat count=4 ; c, halfc, last>
  ${c}. ${halfc}<#if last> Last!</#if>
</@repeat>
<@repeat count=4 ; c, halfc>
  ${c}. ${halfc}
</@repeat>
<@repeat count=4>
  Just repeat it...
</@repeat>
```

如果在分号后面指定了比 `nested` 指令还多的变量，那么最后的循环变量将不会被创建（在嵌套内容中不会被定义）。

### 4.1.6 自定义指令和宏进阶

现在你也许已经阅读过 `FreeMarker` 参考手册的相关部分了：

- 调用自定义指令
- 宏指令

你也可以在 `FTL` 中定义方法，参见 `function` 指令。

也许你对命名空间感兴趣。命名空间可以帮助你组织和重用你经常使用的宏。

## 4.2 在模板中定义变量

正如我们已经描述过的，模板可以使用在数据模型中定义的变量。在数据模型之外，模板本身也可以定义变量来使用。这些临时变量可以适应 `FTL` 指令来创建和替换。要注意每一次模板执行时都维护它自己的这些变量的私有设置，这些变量是在页面用以呈现信息的。变量的初始值是空，当模板执行结束这些变量便被销毁了。

你可以访问一个在模板里定义的变量就像是访问数据模型根上的变量一样。这个变量比定义在数据模型中的同名参数有更高的优先级，那就是说，如果你恰巧定义了一个名为“foo”的变量，而在数据模型中也有一个名为“foo”的变量，那么模板中的变量就会将数据模型根上的变量隐藏（而不是覆盖！）。例如 `${foo}` 将会打印在模板中定义的变量。

在模板中可以定义三种类型的变量：

- 简单变量：它能从模板中的任何位置来访问，或者从使用 `include` 指令引入的模板访问。可以使用 `assign` 或 `macro` 指令来创建或替换这些变量。
- 局部变量：它们只能被设置在宏定义体内，而且只在宏内可见。一个局部变量的生存周期只是宏的调用过程。可以使用 `local` 指令在宏定义体内来创建或替换局部变量。
- 循环变量：循环变量是由指令（如 `list`）自动创建的，而且它们只在指令的开始和结束标记内有效。宏的参数是局部变量而不是循环变量。

示例：使用 `assign` 创建和替换变量

```

<#assign x = 1> <!-- create variable x -->
${x}
<#assign x = x + 3> <!-- replace variable x -->
${x}

```

输出为:

```

1
4

```

局部变量也会隐藏（不是覆盖）同名的简单变量。循环变量也会隐藏（不是覆盖）同名的局部变量和简单变量。例如:

```

<#assign x = "plain">
1. ${x} <!-- we see the plain var. here -->
<@test/>
6. ${x} <!-- the value of plain var. was not changed -->
<#list ["loop"] as x>
    7. ${x} <!-- now the loop var. hides the plain var. -->
    <#assign x = "plain2"> <!-- replace the plain var, hiding does
not mater here -->
    8. ${x} <!-- it still hides the plain var. -->
</#list>
9. ${x} <!-- the new value of plain var. -->

<#macro test>
    2. ${x} <!-- we still see the plain var. here -->
    <#local x = "local">
    3. ${x} <!-- now the local var. hides it -->
    <#list ["loop"] as x>
        4. ${x} <!-- now the loop var. hides the local var. -->
    </#list>
    5. ${x} <!-- now we see the local var. again -->
</#macro>

```

输出为:

```

1. plain
2. plain
3. local
4. loop
5. local
6. plain
7. loop
8. loop
9. plain2

```

内部循环变量可以隐藏外部循环变量：

```
<#list ["loop 1"] as x>
  ${x}
  <#list ["loop 2"] as x>
    ${x}
    <#list ["loop 3"] as x>
      ${x}
    </#list>
  ${x}
</#list>
${x}
</#list>
```

输出为：

```
loop 1
  loop 2
    loop 3
  loop 2
loop 1
```

注意到循环变量的设置是通过指令调用时创建的（本例中的`<list ...>`标签）。没有其他方式去改变循环变量的值（也就是说你不能使用定义指令来改变它的值。）。从上面来看，尽管也可以使用一个循环变量来隐藏另外一个。

有时会发生一个变量隐藏数据模型中的同名变量，但是如果想访问数据模型中的变量，就可以使用特殊变量 `globals`。例如，假设我们在数据模型中有一个名为 `user`，值为“Big Joe”的变量。

```
<#assign user = "Joe Hider">
${user}          <!-- prints: Joe Hider -->
${.globals.user} <!-- prints: Big Joe -->
```

想了解更多的变量使用语法，请阅读 3.3 节：表达式

## 4.3 命名空间

简介

创建一个库

在引入的命名空间上编写变量

命名空间和数据模型

命名空间的生命周期

为他人编写库

### 4.3.1 简介

当运行 FTL 模板时，就会有使用 `assign` 和 `macro` 指令创建的变量的集合（可能是空的），可以从前一章节来看如何使用它们。像这样的变量集合被成为 **namespace 命名空间**。在简单的情况下可以只使用一个命名空间，称之为 **main namespace 主命名空间**。因为通常只使用本页上的命名空间，所以就没有意识到这点。

如果想创建可以重复使用的宏，函数和其他变量的集，通常用术语来说就是引用的 **library 库**。使用多个命名空间是必然的。只要考虑你在一些项目中，或者想和他人共享使用的时候，你是否有一个很大的宏的集合。但要确保库中没有宏（或其他变量）名和数据模型中变量同名，而且也不能和模板中引用其他库中的变量同名。通常来说，变量因为名称冲突也会相互碰撞。所以要为每个库中的变量使用不同的命名空间。

### 4.3.2 创建一个库

我们来建立一个简单的库。假设你需要一个通用的变量和（在你疑问之前，宏当作是变量）：

```
<#macro copyright date>
  <p>Copyright (C) ${date} Julia Smith. All rights reserved.</p>
</#macro>
<#assign mail = "jsmith@acme.com">
```

把上面的这些定义存储在文件 `lib/my_test.ftl` 中（目录是你存放模板的位置）。假设想在 `aWebPage.ftl` 中使用这个模板。如果在 `aWebPage.ftl` 使用 `<#include "/lib/my_test.ftl">`，那么就会在主命名空间中创建两个变量，这样就不是很好，因为想让它们只在同一个命名空间“**My Test Library**”中。所以就不得不使用 `import` 指令来代替 `include` 了。乍一看，这个指令和 `include` 很相似，但是它会为 `lib/my_test.ftl` 创建一个空的命名空间，然后在那里执行。`lib/my_test.ftl` 会发现它自己在一个新的环境中，那里只有数据模型的变量可以来呈现（因为它们在那里都是可见的），然后会在这个环境中创建两个变量。现在来看这很不错，但是如果访问 `aWebPage.ftl` 中的两个变量，而它们使用的是主命名空间，就不能看到其他命名空间中的变量。解决方法是 `import` 指令不仅仅创建命名空间，而且要通过 `import` 的调用者（本例中的主命名空间）创建一个新的哈希表变量，这就成为进入新的命名空间的大门。那么 `aWebPage.ftl` 就像下面这样：

```
<#import "/lib/my_test.ftl" as my> <!-- the hash called "my" will
be the "gate" -->
<@my.copyright date="1999-2002"/>
${my.mail}
```

要注意它是怎么访问为 `lib/my_test.ftl` 创建的命名空间中的变量的，通过新创建的哈希表，`my`。那么将会打印出：

```
<p>Copyright (C) 1999-2002 Julia Smith. All rights reserved.</p>
jsmith@acme.com
```

如果在主命名空间中有一个变量，名为 `mail` 或 `copyright`，那么就不会引起混乱了，因为两个模板使用了不同的命名空间。例如，在 `lib/my_test.ftl` 中修改 `copyright` 成如下这样：

```
<#macro copyright date>
  <p>Copyright (C) ${date} Julia Smith. All rights reserved.
  <br>Email: ${mail}</p>
</#macro>
```

然后修改 `aWebPage.ftl` 中的内容：

```
<#import "/lib/my_test.ftl" as my>
<#assign mail="fred@acme.com">
<@my.copyright date="1999-2002"/>
${my.mail}
${mail}
```

那么将会输出：

```
<p>Copyright (C) 1999-2002 Julia Smith. All rights reserved.
<br>Email: jsmith@acme.com</p>
jsmith@acme.com
fred@acme.com
```

当调用了 `copyright` 宏之后，输出和上面的是相似的，因为 `FreeMarker` 已经暂时转向由 `import` 指令为 `/lib/my_test.ftl` 生成的命名空间了。因此，`copyright` 宏看到在主命名空间中变量 `mail` 存在，而且其他的 `mail` 不存在。

### 4.3.3 在引入的命名空间上编写变量

偶尔想要在一个被包含的命名空间上创建或替换一个变量。那么可以使用 `assign` 指令在完成，如果用到了它的 `namespace` 变量，例如下面这样：

```
<#import "/lib/my_test.ftl" as my>
${my.mail}
<#assign mail="jsmith@other.com" in my>
${my.mail}
```

将会输出：

```
jsmith@acme.com
jsmith@other.com
```

### 4.3.4 命名空间和数据模型

数据模型中的变量在任何位置都是可见的。如果在数据模型中有一个名为 `user` 的变量，那么 `lib/my_test.ftl` 也能访问它，`aWebPage.ftl` 当然也能。

```
<#macro copyright date>
  <p>Copyright (C) ${date} ${user}. All rights reserved.</p>
</#macro>
<#assign mail = "${user}@acme.com">
```

如果 `user` 是“Fred”的话，下面这个例子：

```
<#import "/lib/my_test.ftl" as my>
<@my.copyright date="1999-2002"/>
${my.mail}
```

将会输出：

```
<p>Copyright (C) 1999-2002 Fred. All rights reserved.</p>
Fred@acme.com
```

不要忘了在模板中命名空间（可以使用 `assign` 或 `macro` 指令来创建的变量）中的变量有着比数据模型中的变量更高的优先级。因此，数据模型的内容不会干涉到由库创建的变量。

#### 注意：

在通常一些应用中，你也许想在模板中创建所有命名空间都可见的变量，就像数据模型中的变量一样。但是你不能在模板中改变数据模型，却可以通过 `global` 指令来达到相似的效果，可以阅读参考手册来获得更多信息。

### 4.3.5 命名空间的生命周期

命名空间由使用的 `import` 指令中所写的路径来识别。如果想多次 `import` 这个路径，那么只会为第一次的 `import` 引用创建命名空间执行模板。后面相同路径的 `import` 只是创建一个哈希表当作访问相同命名空间的“门”。例如，在 `aWebPage.ftl` 中：

```
<#import "/lib/my_test.ftl" as my>
<#import "/lib/my_test.ftl" as foo>
<#import "/lib/my_test.ftl" as bar>
${my.mail}, ${foo.mail}, ${bar.mail}
<#assign mail="jsmith@other.com" in my>
${my.mail}, ${foo.mail}, ${bar.mail}
```

将会输出：

```
jsmith@acme.com, jsmith@acme.com, jsmith@acme.com
jsmith@other.com, jsmith@other.com, jsmith@other.com
```

这里可以看到通过 `my`，`foo` 和 `bar` 访问相同的命名空间。

还要注意命名空间是不分层次的，它们相互之间是独立存在的。那么，如果在命名空间 `N1` 中 `import` 命名空间 `N2`，那 `N2` 也不在 `N1` 中，`N1` 只是可以通过哈希表来访问 `N2`。这和在主命名空间中 `import N2`，然后直接访问命名空间 `N2` 是一样的过程。

每一次模板的执行过程，它都有一个私有的命名空间的集合。每一次模板执行工作都是

一个分离且有序的过程，它们仅仅存在一段很短的时间同时页面用以呈现内容，然后就和所有填充过的命名空间一起消失了。因此，无论何时我们说第一次调用 `import`，一个单一模板执行工作的内容都是这样。

### 4.3.6 为他人编写库

如果你已经为其他人员编写一个有用的，高质量的库，你也许想把它放在网络上（就像 <http://freemarker.org/libraries.html>）。为了防止和其他作者使用库的命名相冲突，而且引入其他库时要书写简单，这有一个事实上的标准，那就是指定库路径的格式。这个标准是：库的路径必须对模板和其他库可用（可引用），就像这样：

```
/lib/yourcompany.com/your_library.ftl
```

如果你为 **Example** 公司工作，它们拥有 **www.example.com** 网的主页，你的工作是开发一个部件库，那么要引入你所写的 **FTL** 的路径应该是：

```
/lib/example.com/widget.ftl
```

注意到 **www** 已经被省略了。第三次路径分割后的部分可以包含子目录，可以像下面这样写：

```
/lib/example.com/commons/string.ftl
```

一个重要的规则就是路径不应该包含大写字母，为了分隔词语，使用下划线 `_`，就像 `wml_form`（而不是 `wmlForm`）。

如果你的工作不是为公司或组织开发库，也要注意，你应该使用项目主页的 **URL**，比如 `/lib/example.sourceforge.net/example.ftl` 或 `/lib/geocities.com/jsmith/example.ftl`。

## 4.4 空白处理

简介

剥离空白

使用 `compress` 指令

### 4.4.1 简介

在运行中，模板中的空白在某种程度上来说是纠缠所有模板引擎的一个问题。

我们来看这个模板。我已经用颜色标记了模板中的组件：**文本**，**插值**，**FTL 标签**。使用 `[BR]-s` 来想象换行。



```

<p>List of users: [BR]
<#assign users = [{"name":"Joe",      "hidden":false}, [BR]
                  {"name":"James Bond", "hidden":true}, [BR]
                  {"name":"Julia",     "hidden":false}]>[BR]
<ul>[BR]
<#list users as user>[BR]
  <#if !user.hidden>[BR]
    <li>${user.name}[BR]
  </#if>[BR]
</#list>[BR]
</ul>[BR]
<p>That's all.

```

如果 FreeMarker 能按照规则输出所有的问题，那将会输出：

```

<p>List of users: [BR]
[BR]
<ul>[BR]
[BR]
[BR]
[BR]
<li>Joe[BR]
[BR]
[BR]
[BR]
[BR]
[BR]
[BR]
<li>Julia[BR]
[BR]
[BR]
</ul>[BR]
<p>That's all.

```

这里有太多的不想要的空格和换行了。幸运的是，HTML 和 XML 都不是对空白敏感的，但是这么多多余的空白是很令人头疼的，而且增加处理后的 HTML 文件大小也是没必要的。当然，对于空白敏感的方式的输出这依旧是个大问题。

FreeMarker 提供下面的工具来处理这个问题：

- 忽略某些模板文件的空白的工具（解析阶段空白就被移除了）：
  - ◆ 剥离空白：这个特性会自动忽略在 FTL 标签周围多余的空白。这个特性可以通过模板来随时使用和禁用。
  - ◆ 微调指令：t，rt 和 lt，使用这些指令可以明确地告诉 FreeMarker 去忽略某些空白。可以阅读参考手册来获取更多信息。
  - ◆ FTL 参数 strip\_text：这将从模板中删除所有顶级文本。对模板来说这很有用，它只包含某些定义的宏（还有以他一些没有输出的指令），因为它可以移除宏定义和其他顶级指令中的换行符，这样可以提高模板的可读性。
- 从输出中移除空白的工具（移除临近的空白）：

## ◆ compress 指令

### 4.4.2 剥离空白

如果对于模板来说使这个特性成为可能，那么它就会自动忽略（也就是不在输出中打印出来）两种典型的多余空白：

- 缩进空白和在行末尾的尾部空白（包括换行符）将会被忽略，只会留下 FTL 标签（比如 `<@myMacro/>`，`<#if ...>`）和 FTL 注释（如 `<#-- blah -->`），除了被忽略的空白本身。例如，如果一行只包含一个 `<#if ...>`，那么在标签前面的缩进和标签后面的换行符将会被忽略。然而，如果这行上包含 `<#if ...>x`，那么空白就不会被忽略，因为这个 `x` 不是 FTL 标签。注意，根据这些规则，一行上包含 `<#if ...><#list ...>`，空白就会被忽略，而一行上 `<#if ...> <#list ...>` 有就不会，因为在两个 FTL 标签之间的空白是嵌入的空白，而不是缩进的或尾部空白。
- 加在下面这些指令之间的空白会被忽略：`macro`，`function`，`assign`，`global`，`local`，`ftl`，`import`，但也是仅仅指令之间只有一个空白或 FTL 注释。实际中，它意味着你可以在宏定义和参数定义之间放置空行，因为行间距是为了更好的可读性，不包括打印不必要的空行（换行符）被输出。

使用了剥离空白后，上面那个例子中的输出就会是：

```
<p>List of users: [BR]
<ul> [BR]
  <li>Joe [BR]
  <li>Julia [BR]
</ul> [BR]
<p>That's all.
```

这是因为在剥离之后，模板就编程这样了，被忽略的文本没有被标色：

```
<p>List of users: [BR]
<#assign users = [{"name":"Joe",          "hidden":false}, [BR]
                  {"name":"James Bond", "hidden":true}, [BR]
                  {"name":"Julia",       "hidden":false}]> [BR]
<ul> [BR]
<#list users as user> [BR]
  <#if !user.hidden> [BR]
    <li>${user.name} [BR]
  </#if> [BR]
</#list> [BR]
</ul> [BR]
<p>That's all.
```

剥离空白功能可以通过 `ftl` 指令在模板中开启或关闭。如果你没有通过 `ftl` 指令来指定，那么剥离空白功能是开启还是关闭就依据程序员是如何设置 FreeMarker 的了。默认的情况下剥离空白是开启的，程序员可以留着不管（建议这样做）。注意开启剥离空白时不

会降低模板执行的效率，剥离空白的操作在模板加载时就已经完成了。

剥离空白可以为单独的一行关闭，就是使用 `nt` 指令（对没有去掉空白的行）。

### 4.4.3 使用 `compress` 指令

另外一种方法就是使用 `compress` 指令，和剥离空白相反，这个工作是直接基于生成的输出内容的，而不是对于模板进行。也就是说，它会动态地检查输出内容，而不会检查生成输出 FTL 的程序。它会很强势地移除缩进，空行和重复的空格/制表符（可以阅读参考手册来获取更多信息）。所以对于下面这段代码：

```
<#compress>
<#assign users = [{"name":"Joe",      "hidden":false},
                  {"name":"James Bond", "hidden":true},
                  {"name":"Julia",     "hidden":false}]>
List of users:
<#list users as user>
  <#if !user.hidden>
    - ${user.name}
  </#if>
</#list>
That's all.
</#compress>
```

将会输出：

```
List of users:
- Joe
- Julia
That's all.
```

注意 `compress` 是完全独立于剥离空白的。所以它剥离模板中的空白是可能的。那么之后输出的内容就是被压缩过的。

此外，在默认情况下，名为 `compress` 的用户自定义指令是可以在数据模型中存在的（由于向下兼容特性）。这和指令是相同的，除了可以选择设置 `single_line` 属性，这将会移除所有的介于其中的换行符。在最后那个例子中，如果使用 `<@compress single_line=true>...</@compress>` 来代替 `<#compress>...</#compress>`，那么就会得到如下输出：

```
List of users: - Joe - Julia That's all.
```

## 4.5 替换（方括号）语法

注意：

这个特性自 FreeMarker 2.3.4 版本后可用。

FreeMarker 支持一个替换的语法。就是在 FreeMarker 指令和注释中用 `[` 和 `]` 来代替和，

例如下面这个例子：

- 调用预定义指令：`[#list animals as being]...[/#list]`
- 调用用户自定义指令：`[@myMacro /]`
- 注释：`[#-- the comment --]`

为了使用这种语法从而代替默认的，从模板开始，使用 `ftl` 指令都要使用这用语法。如果你不知道什么是 `ftl` 指令，那么就用 `[#ftl]` 来开始模板，要记住这个要放在文件的最前面（除了它前面的空格）。例如，下面的示例入门章节的最后一个例子使用这种替换语法的样子（假设这是一个完整的模板，而不是一个片段）。

```
[#ftl]
<p>We have these animals:
<table border=1>
  <tr><th>Name<th>Price
  [#list animals as being]
  <tr>
    <td>
      [#if being.size = "large"]<b>[/#if]
      ${being.name}
      [#if being.size = "large"]</b>[/#if]
    <td>${being.price} Euros
  [/#list]
</table>
```

这种替换语法（方括号）和默认语法（尖括号）在一个模板中是相互排斥的。那就是说，整个模板要么全使用替换语法，要么全使用默认语法。如果模板使用了代替语法，那么如 `<#if ...>` 这样的东西就会被算作是静态文本，而不是 `FTL` 标签。类似地，如果模板使用默认语法，那么如 `[#if ...]` 这样的也会被算作是静态文本，而不是 `FTL` 标签。

如果你以 `[#ftl ...]`（...代表可选的参数列表，当然仅用 `[#ftl]s` 也行）来开始文件，那文件就会使用替换（方括号）语法。如果使用 `<#ftl ...>` 来开始，那么文件就会使用正常（尖括号）语法。如果文件中没有 `ftl` 指令，那么程序员可以通过配置 `FreeMarker`（程序员参看 `API` 文档的 `Configuration.setTagSyntax(int)` 来使用）来决定使用哪种语法。但是程序员可能使用默认配置。`FreeMarker 2.3.x` 版本默认配置使用常规语法。而 `2.4` 版本中的默认配置将会自动检测，也就是说第一个 `FreeMarker` 标签决定了语法形式（它可以是任意的，而不仅仅是 `ftl`）。

# 第二部分 程序开发指南

## 第一章 程序开发入门

章节内容如下：

创建配置实例

创建数据模型

获得模板

合并模板和数据模型

将代码放在一起

注意，如果你还是 **FreeMarker** 的新手，你应该先阅读模板开发指南部分的入门章节。

### 1.1 创建配置实例

首先，你应该创建一个 `freemarker.template.Configuration` 实例，然后调整它的设置。`Configuration` 实例是存储 **FreeMarker** 应用级设置的中央区域。同时，它也处理创建和缓存预解析模板的工作。

也许你只在应用（可能是 `servlet`）生命周期的开始执行它一次：

```
Configuration cfg = new Configuration();
// Specify the data source where the template files come from.
// Here I set a file directory for it:
cfg.setDirectoryForTemplateLoading(
    new File("/where/you/store/templates"));
// Specify how templates will see the data-model. This is an
// advanced topic...
// but just use this:
cfg.setObjectWrapper(new DefaultObjectWrapper());
```

现在开始，应该使用单实例配置。注意不管一个系统有多少独立的组件来使用 **FreeMarker**，它们都会使用他们自己私有的 `Configuration` 实例。

### 1.2 创建数据模型

在简单的示例中你可以使用 `java.lang` 和 `java.util` 下的类，用户自定义的 **Java Beans** 来构建数据对象。

- 使用 `java.lang.String` 来构建字符串。
- 使用 `java.lang.Number` 来派生数字。
- 使用 `java.lang.Boolean` 来构建布尔值。
- 使用 `java.util.List` 或 **Java** 数组来构建序列。

- 使用 `java.util.Map` 来构建哈希表。
- 使用你自己定义的 `bean` 类来构建哈希表，`bean` 中的项和 `bean` 的属性对应。例如 `product` 中的 `price` 属性可以用 `product.price` 来获取。(bean 的 `action` 也可以通过这种方式暴露出来，要了解更多信息可以参看：`bean` 的包装)。

让我们为模板开发指南的第一个例子来构建数据模型，为了方便，这里再展示一次：

```
(root)
|
+- user = "Big Joe"
|
+- latestProduct
  |
  +- url = "products/greenmouse.html"
  |
  +- name = "green mouse"
```

下面是构建这个数据模型的 Java 代码片段：

```
// Create the root hash
Map root = new HashMap();
// Put string "user" into the root
root.put("user", "Big Joe");
// Create the hash for "latestProduct"
Map latest = new HashMap();
// and put it into the root
root.put("latestProduct", latest);
// put "url" and "name" into latest
latest.put("url", "products/greenmouse.html");
latest.put("name", "green mouse");
```

对于 `latestProduct` 你也可以使用有 `url` 和 `name` 属性的 Java Bean(也就是说，对象有公共的 `String getURL()` 和 `String getName()` 方法)；它和模板的观点相同。

## 1.3 获得模板

模板代表了 `freemarker.template.Template` 的实例。典型的做法是从 `Configuration` 实例中获取一个 `Template` 实例。无论什么时候你需要一个模板实例，都可以使用它的 `getTemplate` 方法来获取。在之前设置的目录中，用 `test.ftl` 存储示例模板，那么就可以这样做：

```
Template temp = cfg.getTemplate("test.ftl");
```

当调用这个方法的时候，将会创建一个 `test.ftl` 的 `Template` 实例，通过读取然后解析（编译）它。`Template` 实例以解析后的形式存储模板，而不是以源文件的文本形式。

`Configuration` 获取 `Template` 实例，当再次获得 `test.ftl` 时，它可能不

会创建新的 `Template` 实例（因此不会读取和解析文件），而是返回第一次创建的实例。

## 1.4 合并模板和数据模型

我们都知道的，数据模型+模板=输出，我们已经有了一个数据模型（`root`）和一个模板（`temp`）了，所以为了得到输出就要合并它们。这是由模板的 `process` 方法完成的。它用数据模型的根和 `Writer` 对象作为参数，然后向 `Writer` 对象写入产生的内容。为简单起见，这里我们只做标准的输出：

```
Writer out = new OutputStreamWriter(System.out);
temp.process(root, out);
out.flush();
```

这会向你的终端输出你在模板开发指南部分的第一个示例中看到的内容。

一旦获得了 `Template` 实例，就能将它和不同的数据模型进行不限次数（`Template` 实例是无状态的）的合并。此外，当 `Template` 实例创建之后 `test.ftl` 文件才能访问，而不是调用处理方法时。

## 1.5 将代码放在一起

这是一个由之前的片段组合在一起的源程序文件。不要忘了将 `freemarker.jar` 放到 `CLASSPATH` 中。

```

import freemarker.template.*;
import java.util.*;
import java.io.*;

public class Test {

    public static void main(String[] args) throws Exception {

        /* You should do this ONLY ONCE in the whole application
        life-cycle:  */

        /* Create and adjust the configuration */
        Configuration cfg = new Configuration();
        cfg.setDirectoryForTemplateLoading(
            new File("/where/you/store/templates"));
        cfg.setObjectWrapper(new DefaultObjectWrapper());

        /* You usually do these for many times in the application
        life-cycle: */

        /* Get or create a template */
        Template temp = cfg.getTemplate("test.ftl");

        /* Create a data-model */
        Map root = new HashMap();
        root.put("user", "Big Joe");
        Map latest = new HashMap();
        root.put("latestProduct", latest);
        latest.put("url", "products/greenmouse.html");
        latest.put("name", "green mouse");

        /* Merge data-model with template */
        Writer out = new OutputStreamWriter(System.out);
        temp.process(root, out);
        out.flush();
    }
}

```

注意：

为了简单这里压制了异常，而在实际产品中不要这样做。



## 第二章 数据模型

章节内容如下：

基本内容

标量

容器

方法

指令

节点变量

对象包装

这只是一个介绍说明，查看 [FreeMarker Java API 文档](#) 获取更多信息。

### 2.1 基本内容

在程序开发入门章节中，我们已经知道如何使用基本的 Java 类 (`Map`, `String` 等) 构建一个数据模型了。在内部，模板中可用的变量都是实现了 `freemarker.template.TemplateModel` 接口的 Java 对象。但在你自己的数据模型中，可以使用基本的 Java 集合作为变量，因为这些变量会在内部被替换为适当的 `TemplateModel` 类型。这种功能被称作是 **object wrapping** 对象包装。对象包装功能可以透明地把任何类型的对象转换为实现了 `TemplateModel` 接口类型的实例。这使得下面的转换成为可能，如在模板中把 `java.sql.ResultSet` 转换为序列变量，把 `javax.servlet.ServletRequest` 对象转换成包含请求属性的哈希表变量，甚至可以遍历 XML 文档作为 FTL 变量。包装（转换）这些对象，需要使用合适的，也就是所谓的对象包装器实现（可能是自定义的实现）；这将在后面讨论。现在的要点是想从模板访问的任何对象，早晚都要转换为实现了 `TemplateModel` 接口的对象。那么首先你应该熟悉来写 `TemplateModel` 接口的实现类。

有一个 `freemarker.template.TemplateModel` 粗略的子接口对应每种基本变量类型（`TemplateHashModel` 对应哈希表，`TemplateSequenceModel` 对应序列，`TemplateNumberModel` 对应数字等等）。例如，想为使用 `java.sql.ResultSet` 为模板作为一个序列，那么就需要编写一个 `TemplateSequenceModel` 实现类，这个类能够读取 `java.sql.ResultSet-s`。我们常这么说，你使用 `TemplateModel` 实现类包装了 `java.sql.ResultSet`，基本上只是封装 `java.sql.ResultSet`，来提供使用普通的 `TemplateSequenceModel` 接口访问它。要注意一个类可以实现多个 `TemplateModel` 接口，这就是为什么 FTL 变量可以有多种类型（参考：模板开发指南/数值和类型/基本内容）。

注意这些接口的一个不重要实现是提供了 `freemarker.template` 包。例如，将一个 `String` 转换成 FTL 的字符串变量，可以使用 `SimpleScalar`，将 `java.util.Map` 转换成 FTL 的哈希表变量，可以使用 `SimpleHash` 等等。

如果想尝试自己的 `TemplateModel` 实现，一个简单的方式是创建它的实例，然后将这个实例放入数据模型中（也就是把它放在哈希表的根上）。对象包装器将会给模板暴露

它的原状，因为它已经实现了 `TemplateModel` 接口，所以没有转换（包装）的需要。（这个手法当你不想用对象包装器来包装（转换）某些对象时仍然有用。）

## 2.2 标量

有 4 种标量的类型：

- 布尔值
- 数字
- 字符串
- 日期

每一种标量类型都是 `TemplateTypeModel` 接口的实现，这里的 *Type* 就是类型的名称。这些接口只定义了一个方法 `type getAsType()`；它返回变量的 Java 类型（`boolean`，`Number`，`String` 和 `Date` 各自代表的值）的值。

**注意：**

由于历史的原因字符串标量的接口是 `TemplateScalarModel`，而不是 `TemplateStringModel`。

这些接口的一个不重要实现和 `SimpleType` 类名在 `freemarker.template` 包中是可用的。但是却没有 `SimpleBooleanModel` 类型；为了代表布尔值，可以使用 `TemplateBooleanModel.TRUE` 和 `TemplateBooleanModel.FALSE` 来单独使用。

**注意：**

由于历史的原因字符串标量的实现类是 `SimpleScalar`，而不是 `SimpleString`。

在 FTL 中标量是一成不变的。当在模板中设置变量的值时，使用其他的实例来替换 `TemplateTypeModel` 实例时，是不用改变原来实例中存储的值的。

### 2.2.1 数据类型的难点

数据类型上还有一些复杂，因为 Java API 通常不区别 `java.util.Date`s，只存储日期部分（April 4, 2003），时间部分（10:19: 18 PM），或两者都存（April 4, 2003 10:19:18 PM）。为了用本文正确显示一个日期变量，FreeMarker 必须知道 `java.util.Date` 的哪个部分存储了有意义上的信息，那部分没有被使用。不幸的是，Java API 在这里明确的说，由数据库控制（SQL），因为数据库通常有分离的日期，时间和时间戳（又叫做日期-时间）类型，`java.sql` 有 3 个对应的 `java.util.Date` 子类 and 它们相匹配。

`TemplateDateModel` 接口有两个方法：分别是 `java.util.Date getAsDate()` 和 `int getDateType()`。这个接口典型的实现是存储一个 `java.util.Date` 对象，加上一个整数来辨别“数据库存储的类型”。这个整数的值必须是 `TemplateDateModel` 接口中的常量：`DATE`，`TIME`，`DATETIME` 和 `UNKNOWN`。

什么是 `UNKNOWN`？我们之前说过，`java.lang` 和 `java.util` 下的类通常被自动转换成 `TemplateModel` 的实现类，就是所说的对象包装器。当对象转换器面对一个 `java.util.Date` 对象时，而不是 `java.sql` 日期类的实例，它就不能确定“数据库存储的类型”是什么，所以就使用 `UNKNOWN`。往后执行，如果模板需要使用这个变量，操作也需要使用“数据存储的类型”，那就会停止执行并抛出错误。为了避免这种情况，对于那些可能有问题的变量，模板开发人员需要帮助 FreeMarker 决定“数据库存储的类型”，使

用内建函数 `date`, `time` 或 `datetime` 就可以了。注意一下, 如果对要格式化参数使用了内建函数 `string`, 比如 `foo?string("MM/dd/yyyy")`, 那么 FreeMarker 就不必知道“数据库存储的类型”了。

## 2.3 容器

容器包括哈希表, 序列和集合三种。

### 2.3.1 哈希表

哈希表是实现了 `TemplateHashModel` 接口的 Java 对象。`TemplateHashModel` 接口有两个方法: `TemplateModel get(String key)`, 这个方法根据给定的名称返回子变量, `boolean isEmpty()` 这个方法表明哈希表是否含有子变量。`get` 方法当在给定的名称没有找到子变量时返回 `null`。

`TemplateHashModelEx` 接口扩展了 `TemplateHashModel`。它增加了更多的方法, 使得可以使用内建函数 `values` 和 `keys` 来枚举哈希表中的子变量。

经常使用的实现类是 `SimpleHash`, 这个类实现了 `TemplateHashModelEx` 接口。从内部来说, 它使用一个 `java.util.Hash` 类型的对象存储子变量。`SimpleHash` 的方法可以添加和移除子变量。这些方法应该用来在变量被创建之后直接初始化。

在 FTL 中容器是一成不变的。那就是说你不能添加, 替换和移除容器中的子变量。

### 2.3.2 序列

序列是实现了 `TemplateSequenceModel` 的 Java 对象。它包含两个方法: `TemplateModel get(int index)` 和 `int size()`。

经常使用的实现类是 `SimpleSequence`, 这个类内部使用一个 `java.util.List` 类型的对象存储它的子变量。`SimpleSequence` 有添加子元素的方法。在序列创建之后应该使用这些方法来填充序列。

### 2.3.3 集合

集合是实现了 `TemplateCollectionModel` 接口的 Java 对象。这个接口只有一个方法: `TemplateModelIterator iterator()`。`TemplateModelIterator` 接口和 `java.util.Iterator` 相似, 但是它返回 `TemplateModels` 而不是 `Object-s`, 而且它能抛出 `TemplateModelExceptions`。

通常使用的实现类是 `SimpleCollection`。

## 2.4 方法

方法变量在于实现了 `TemplateMethodModel` 接口的模板内。这个接口包含一个方法：`TemplateModel exec(java.util.List arguments)`。当使用方法调用表达式调用方法时，`exec` 方法将会被调用。形参将会包含 FTL 方法调用形参的值。`exec` 方法的返回值给出了 FTL 方法调用表达式的返回值。

`TemplateMethodModelEx` 接口扩展了。它没有任何新增的方法。事实上这个对象实现这个标记接口暗示给 FTL 引擎，形式参数应该直接以 `TemplateModel-s` 形式放进 `java.util.List`。否则将会以 `String-s` 形式放入 `List`。

明显的原因是这些接口没有默认的实现。

例如这个方法，返回第一个字符串在第二个字符串第一次出现时的索引位置，如果第二个字符串中不包含第一个字符串，则返回“-1”：

```
public class IndexOfMethod implements TemplateMethodModel {

    public TemplateModel exec(List args) throws
TemplateModelException {
        if (args.size() != 2) {
            throw new TemplateModelException("Wrong
arguments");
        }
        return new SimpleNumber(
            ((String) args.get(1)).indexOf((String)
args.get(0)));
    }
}
```

如果将一个实例放入根，像这样：

```
root.put("indexOf", new IndexOfMethod());
```

那么就可以在模板中调用：

```
<#assign x = "something">
${indexOf("met", x)}
${indexOf("foo", x)}
```

那么输出为：

```
2
-1
```

如果需要访问 FTL 运行时环境（读/写变量，获取本地信息等），可以使用 `Environment.getCurrentEnvironment()` 来获取。

## 2.5 指令

Java 程序员可以使用 `TemplateDirectiveModel` 接口在 Java 代码中实现自定义指令。详情可以参加 [API 文档](#)。

注意：

`TemplateDirectiveModel` 在 FreeMarker 2.3.11 版本时才加入。用来代替快被废弃的 `TemplateTransformModel`。

### 2.5.1 第一个示例

我们要实现一个指令，这个指令可以将在它开始标签和结束标签之内的东西都转换为大写形式。就像这个模板：

```
foo
<@upper>
  bar
  <!-- All kind of FTL is allowed here -->
  <#list ["red", "green", "blue"] as color>
    ${color}
  </#list>
  baaz
</@upper>
wombat
```

将会输出：

```
foo
  BAR
    RED
    GREEN
    BLUE
  BAAZ
wombat
```

下面是指令类的源代码：

```
package com.example;

import java.io.IOException;
import java.io.Writer;
import java.util.Map;

import freemarker.core.Environment;
import freemarker.template.TemplateDirectiveBody;
import freemarker.template.TemplateDirectiveModel;
import freemarker.template.TemplateException;
```

```

import freemarker.template.TemplateModel;
import freemarker.template.TemplateModelException;

/**
 * FreeMarker user-defined directive that progressively
transforms
 * the output of its nested content to upper-case.
 *
 * <p><b>Directive info</b></p>
 *
 * <p>Directive parameters: None
 * <p>Loop variables: None
 * <p>Directive nested content: Yes
 */
public class UpperDirective implements TemplateDirectiveModel
{

    public void execute(Environment env,
        Map params, TemplateModel[] loopVars,
        TemplateDirectiveBody body)
        throws TemplateException, IOException {
        // Check if no parameters were given:
        if (!params.isEmpty()) {
            throw new TemplateModelException(
                "This directive doesn't allow parameters.");
        }
        if (loopVars.length != 0) {
            throw new TemplateModelException(
                "This directive doesn't allow loop
variables.");
        }

        // If there is non-empty nested content:
        if (body != null) {
            // Executes the nested body. Same as <#nested> in FTL,
except
            // that we use our own writer instead of the current
output writer.
            body.render(new
UpperCaseFilterWriter(env.getOut()));
        } else {
            throw new RuntimeException("missing body");
        }
    }
}

```

```

/**
 * A {@link Writer} that transforms the character stream to
upper case
 * and forwards it to another {@link Writer}.
 */
private static class UpperCaseFilterWriter extends Writer
{

    private final Writer out;

    UpperCaseFilterWriter (Writer out) {
        this.out = out;
    }

    public void write(char[] cbuf, int off, int len)
        throws IOException {
        char[] transformedCbuf = new char[len];
        for (int i = 0; i < len; i++) {
            transformedCbuf[i] =
Character.toUpperCase(cbuf[i + off]);
        }
        out.write(transformedCbuf);
    }

    public void flush() throws IOException {
        out.flush();
    }

    public void close() throws IOException {
        out.close();
    }
}
}

```

现在我们需要创建这个类的实例，然后让这个指令在模板中可以通过名称“upper”来访问（或者其他我们想用的名字）。一个可行的方案是把这个指令放到数据模型中：

```
root.put("upper", new com.example.UpperDirective());
```

但更好的实践是将常用的指令作为共享变量放到 `Configuration` 中。

当然也可以使用内建函数 `new` 将指令放到一个 `FTL` 库（宏的集，就像在模板中，使用 `include` 或 `import`）中。

```

<#-- Maybe you have directives that you have implemented in FTL
-->
<#macro something>
    ...
</#macro>

<#-- Now you can't use <#macro upper>, but instead you can: -->
<#assign upper = "com.example.UpperDirective"?new()>

```

## 2.5.2 第二个示例

我们来创建一个指令，这个指令可以一次一次地执行其中的嵌套内容，这个次数由指定的数字确定（就像 `list` 指令），可以使用 `<hr>-s` 将输出的重复内容分开。这个指令我们命名为“repeat”。示例模板如下：

```

<#assign x = 1>

<@repeat count=4>
    Test ${x}
    <#assign x = x + 1>
</@repeat>

<@repeat count=3 hr=true>
    Test
</@repeat>

<@repeat count=3; cnt>
    ${cnt}. Test
</@repeat>

```

输出为：

```

Test 1
Test 2
Test 3
Test 4

Test
<hr> Test
<hr> Test

1. Test
2. Test
3. Test

```



指令的实现类为:

```
package com.example;
import java.io.IOException;
import java.io.Writer;
import java.util.Iterator;
import java.util.Map;

import freemarker.core.Environment;
import freemarker.template.SimpleNumber;
import freemarker.template.TemplateBooleanModel;
import freemarker.template.TemplateDirectiveBody;
import freemarker.template.TemplateDirectiveModel;
import freemarker.template.TemplateException;
import freemarker.template.TemplateModel;
import freemarker.template.TemplateModelException;
import freemarker.template.TemplateNumberModel;

/**
 * FreeMarker user-defined directive for repeating a section
 * of a template,
 * optionally with separating the output of the repetitions with
 * <tt>&lt;hr>&/tt>-s.
 *
 * <p><b>Directive info</b></p>
 *
 * <p>Parameters:
 * <ul>
 * <li><code>count</code>: The number of repetitions.
 * Required!
 * <ul>
 * <li>Must be a non-negative number. If it is not a whole number
 * then it will
 * <li>be rounded <em>down</em>.
 * <li><code>hr</code>: Tells if a HTML "hr" element could be
 * printed between
 * <li>repetations. Boolean. Optional, defaults to
 * <code>>false</code>.
 * </ul>
 *
 * <p>Loop variables: One, optional. It gives the number of the
 * current
 * <li>repetition, starting from 1.
 *
 * <p>Nested content: Yes
 */
```

```

public class RepeatDirective implements TemplateDirectiveModel
{
    private static final String PARAM_NAME_COUNT = "count";
    private static final String PARAM_NAME_HR = "hr";
    public void execute(Environment env, Map params,
        TemplateModel[] loopVars, TemplateDirectiveBody body)
        throws TemplateException, IOException {
        // Processing the parameters:
        int countParam = 0;
        boolean countParamSet = false;
        boolean hrParam = false;
        Iterator paramIter = params.entrySet().iterator();
        while (paramIter.hasNext()) {
            Map.Entry ent = (Map.Entry) paramIter.next();
            String paramName = (String) ent.getKey();
            TemplateModel paramValue = (TemplateModel)
ent.getValue();
            if (paramName.equals(PARAM_NAME_COUNT)) {
                if (!(paramValue instanceof
TemplateNumberModel)) {
                    throw new TemplateModelException("The \"" + PARAM_NAME_HR +
"\\" parameter " + "must be a number.");
                }
                countParam = ((TemplateNumberModel) paramValue)
                    .getAsNumber().intValue();
                countParamSet = true;
                if (countParam < 0) {
                    throw new TemplateModelException(
                        "The \"" + PARAM_NAME_HR + "\\"
parameter " + "can't be negative.");
                }
            } else if (paramName.equals(PARAM_NAME_HR)) {
                if (!(paramValue instanceof
TemplateBooleanModel)) {
                    throw new TemplateModelException(
                        "The \"" + PARAM_NAME_HR + "\\"
parameter " + "must be a boolean.");
                }
                hrParam = ((TemplateBooleanModel) paramValue)
                    .getAsBoolean();
            } else {
                throw new TemplateModelException(
                    "Unsupported parameter: " + paramName);
            }
        }
    }
}

```

```

if (!countParamSet) {
    throw new TemplateModelException(
        "The required \"" + PARAM_NAME_COUNT + "\" paramter"
+ "is missing.");
}
if (loopVars.length > 1) {
    throw new TemplateModelException(
        "At most one loop variable is allowed.");
}
// Yeah, it was long and boring...
// Do the actual directive execution:
Writer out = env.getOut();
if (body != null) {
    for (int i = 0; i < countParam; i++) {
// Prints a <hr> between all repetations if the "hr" parameter
// was true:
        if (hrParam && i != 0) {
            out.write("<hr>");
        }
        // Set the loop variable, if there is one:
        if (loopVars.length > 0) {
            loopVars[0] = new SimpleNumber(i + 1);
        }
        // Executes the nested body (same as <#nested> in
FTL). In this
        // case we don't provide a special writer as the
parameter:
        body.render(env.getOut());
    }
}
}
}

```

### 2.5.3 提示

`TemplateDirectiveModel` 对象通常是有状态的，这一点非常重要。经常犯的错误是存储指令的状态然后在对象的属性中调用执行。想一下相同指令的嵌入调用，或者指令对象被用作共享变量，并通过多线程同时访问。

不幸的是，不支持传递参数的位置（而不是参数名称）。从 **FreeMarker** 的 2.4 版本开始，它将被修正。

## 2.6 节点变量

节点变量体现了树形结构中的节点。节点变量的引入是为了帮助在数据模型中处理 XML 文档，但是它们也可以用于构建树状模型。如需有关模板语言角度中的节点信息，可以阅读之前模板开发指南中 2.2.5.1 节内容。

节点变量有下列属性，这些都由 `TemplateNodeModel` 接口的方法提供。

- 基本属性：
  - ◆ `TemplateSequenceModel getChildNodes()`：一个节点有子节点序列（除非这个节点是叶子节点，这时方法返回一个空序列或者是 `null`）。子节点应该也是节点变量。
  - ◆ `TemplateNodeModel getParentNode()`：一个节点只有一个父节点（除非这个节点是树的根节点，这时方法返回 `null`）。
- 可选属性。如果一个属性在具体的使用中没有意义，那对应的方法应该返回 `null`：
  - ◆ `String getNodeName()`：节点名称也是宏的名称，当使用 `recurse` 和 `visit` 指令时，它用来控制节点。因此如果想通过节点使用这些指令，那么节点名称是必须的。
  - ◆ `String getNodeType()`：在 XML 中：`"element"`，`"text"`，`"comment"`等。如果这些信息可用，就是通过 `recurse` 和 `visit` 指令来查找节点的默认处理宏。而且，它对其他有具体用途的应用程序也是有用的。
  - ◆ `String getNamespaceURI()`：这个节点所属的命名空间（和用于库的 FTL 命名空间无关）。例如，在 XML 中，这就是元素和属性所属的 XML 命名空间的 URI。信息这个如果可用，就是通过 `recurse` 和 `visit` 指令来查找存储控制宏的 FTL 命名空间。

在 FTL 这里，节点属性的直接使用可以通过内建函数 `node` 完成，还有 `visit` 和 `recurse` 宏。

在很多使用情况下，实现了 `TemplateNodeModel` 接口和其他接口的变量，因为节点变量属性仅提供基本的节点间导航的方法。需要具体的例子，请参考 FreeMarker 如何处理 XML。

## 2.7 对象包装

当往容器中添加一些东西时，正如在 FreeMarker API 文档中看到的那样，它可以收到任何 java 对象类型的参数，而不一定是 `TemplateModel`。这是因为模板实现时会默认地用合适的 `TemplateModel` 对象来替换原有对象。比如向容器中加入一个 `String`，也许它将被替换为一个 `SimpleScalar` 实例来存储相同的文本。

至于替换什么时候发生，这就是容器有问题（类的业务实现了容器接口）的业务处理所在，但是它在获取子变量时必须会发生，因为 `getter` 方法（依据接口而定）会返回 `TemplateModel`，而不是 `Object`。`SimpleHash`，`SimpleSequence` 和 `SimpleCollection` 使用最懒的策略，当第一次获取子变量时，它们用一个适合的 `TemplateModel` 来替换一个非 `TemplateModel` 子变量。

至于什么类型的 Java 对象可以被替换，又使用什么样的 `TemplateModel` 来实现，它可以被实现的容器自身来控制，也可以委派给 `ObjectWrapper` 的一个实例。`ObjectWrapper` 是一个接口，其中只有一个方法：`TemplateModel`

`wrap(java.lang.Object obj)`。可以传递一个 `Object` 类型的参数，它会返回对应的 `TemplateModel` 对象，如果不行则抛出 `TemplateModelException` 异常。替换原则是在 `ObjectWrapper` 的实现类中编码实现的。

最重要的 `ObjectWrapper` 实现类是 `FreeMarker` 核心包提供的：

- `ObjectWrapper.DEFAULT_WRAPPER`：它用 `SimpleScalar` 来替换 `String`，`SimpleNumber` 来替换 `Number`，`SimpleSequence` 来替换 `List` 和 数组，`SimpleHash` 来替换 `Map`，`TemplateBooleanModel.TRUE` 或 `TemplateBooleanModel.FALSE` 来替换 `Boolean`，`freemarker.ext.dom.NodeModel` 来替换 W3C 的 DOM 模型节点。对于 `Jython` 对象，包装器会调用 `freemarker.ext.jython.JythonWrapper`。而对于其他对象，则会调用 `BEAN_WRAPPER`。
- `ObjectWrapper.BEANS_WRAPPER`：它可以通过 `Java` 反射机制来获取到 `Java Bean` 的属性和其他任意对象类型的成员变量。最新的 `FreeMarker 2.3` 版本中，它是 `freemarker.ext.beans.BeansWrapper` 的实例，有单独的一章将会来介绍它。

做一个具体的例子，让我们来看看 `SimpleXxx` 类型都是怎么工作的。`SimpleHash`，`SimpleSequence` 和 `SimpleCollection` 使用 `DEFAULT_WRAPPER` 来包装子变量（除非在构造方法中传递另外一个包装器）。这个例子在 `Action` 中来展示 `DEFAULT_WRAPPER`。

```
Map map = new HashMap();
map.put("anotherString", "blah");
map.put("anotherNumber", new Double(3.14));
List list = new ArrayList();
list.add("red");
list.add("green");
list.add("blue");

SimpleHash root = new SimpleHash(); // will use the default
wrapper
root.put("theString", "wombat");
root.put("theNumber", new Integer(8));
root.put("theMap", map);
root.put("theList", list);
```

假设 `root` 是数据模型的 `root`，那么得到的数据模型将是：

```
(root)
|
+- theString = "wombat"
|
+- theNumber = 8
|
+- theMap
|  |
|  +- anotherString = "blah"
|  |
|  +- anotherNumber = 3.14
|
+- theList
|
|  +- (1st) = "red"
|
|  +- (2nd) = "green"
|
|  +- (3rd) = "blue"
```

注意在 `theMap` 和 `theList` 中的 `Object-s` 也可以作为子变量来访问。这是因为，当要访问 `theMap.anotherString` 时，`SimpleHash`（这里作为根哈希表）会静静地用 `SimpleHash` 实例替换 `Map` (`theMap`)，这个实例使用了和根哈希表相同的包装器。所以当访问其中的子变量 `anotherString` 时，就会使用 `SimpleScalar` 来替换它。

如果在数据模型中放了任意的对象，那么 `DEFAULT_WRAPPER` 就会调用 `BEANS_WRAPPER` 来包装这个对象：

```
SimpleHash root = new SimpleHash();
// expose a "simple" java objects:
root.put("theString", "wombat");
// expose an "arbitrary" java objects:
root.put("theObject", new TestObject("green mouse", 1200));
```

假设 `TestObject` 是这样的：

```

public class TestObject {
    private String name;
    private int price;

    public TestObject(String name, int price) {
        this.name = name;
        this.price = price;
    }

    // JavaBean properties
    // Note that public fields are not visible directly;
    // you must write a getter method for them.
    public String getName() {return name;}
    public int getPrice() {return price;}

    // A method
    public double sin(double x) {
        return Math.sin(x);
    }
}

```

数据模型就会是这样：

```

(root)
|
+- theString = "wombat"
|
+- theObject
  |
  +- name = "green mouse"
  |
  +- price = 1200
  |
  +- number sin(number)

```

我们可以这样把它和模板合并：

```

${theObject.name}
${theObject.price}
${theObject.sin(123)}

```

将会输出：

```

green mouse
1200
-0,45990349068959124

```

之前我们已经看到了，我们使用 `java.util.HashMap` 作为根哈希表，而不是 `SimpleHash` 或其他特定的 `FreeMarker` 类。因为 `Template.process(...)` 自动包装了给定的数据模型参数的对象，所以它才会起作用。它使用受 `Configuration` 级设置的对象包装器，`object_wrapper`（除非明确指定一个 `ObjectWrapper` 作为它的参数）。因此，编写简单的 `FreeMarker` 应用程序就不需要知道 `TemplateModel-s` 了。注意根的类型不需要一定是 `java.util.Map`。它也可以是实现了 `TemplateHashModel` 接口的被包装的对象。

`object_wrapper` 设置的默认值是 `ObjectWrapper.DEFAULT_WRAPPER`。如果想改变它，比如换成 `ObjectWrapper.BEANS_WRAPPER`，那么可以这样来配置 `FreeMarker` 引擎（在其它线程开始使用它之前）：

```
cfg.setObjectWrapper(ObjectWrapper.BEANS_WRAPPER);
```

要注意我们可以在这里设置任何对象实现接口 `ObjectWrapper`，当然也可以用来设置你自己实现的实现类。

对于包装了基本 Java 容器类型（比如 `java.util.Map-s` 和 `java.util.List-s`）的 `TemplateModel` 实现类，常规是它们使用像它们父容器那样的相同对象包装器来包装它们的子变量。从技术上讲，它们是被父容器（它对所创建的子类有全部的控制权）实例化的，因为父容器创建了它们，所以它们使用和父容器一样的对象包装器。如果 `BEANS_WRAPPER` 用来包装根哈希表，那么它也会被用来包装子变量（子变量的子变量也是如此，以此类推）。这个之前看到的 `theMap.anotherString` 是同样的现象。

## 第三章 配置

章节内容如下：

- 基本内容
- 共享变量
- 配置信息
- 模板加载
- 错误控制

这仅仅是一个概览，了解更多请阅读 `FreeMarker` 的 API 文档。

### 3.1 基本内容

配置就是在对象中存储你常用（应用级别）的设置和定义某些想在所有模板中可用的变量。它们也会处理 `Template` 实例的创建和缓存。配置对象是 `freemarker.template.Configuration` 的实例，可以通过构造方法来创建它。一个应用程序通常只使用一个共享的 `Configuration` 实例。

配置对象通过 `Template` 的方法来使用，特别是通过 `process` 方法。每个实例都有一个确切的且和它相关的 `Configuration` 实例，这个实例由 `Template` 构造方法分配给 `Template` 实例。可以指定一个 `Configuration` 实例作为它的参数。通常情况下，使用 `Configuration.getTemplate`（而不是直接调用 `Template` 的构造



方法) 来获得 `Template` 实例, 这种情况下, 相关的 `Configuration` 实例就会是 `getTemplate` 方法被调用时的那个实例。

### 3.2 共享变量

**Shared variables 共享变量** 是为所有模板所定义的变量。可以使用 `setSharedVariable` 方法向配置实例中添加共享变量:

```
Configuration cfg = new Configuration();
...
cfg.setSharedVariable("wrap", new WrapDirective());
cfg.setSharedVariable("company", "Foo Inc."); // Using
ObjectWrapper.DEFAULT_WRAPPER
```

在所有使用这个配置的模板中, 名为 `wrap` 的用户自定义指令和一个名为 `company` 的字符串将会在数据模型的根上可见, 那就不用再在根哈希表上一次又一次地添加它们。在传递给 `Template.process` 根对象里的变量将会隐藏同名的共享变量。

**警告!**

如果配置对象被多线程使用, 不要使用 `TemplateModel` 实现类来作为共享变量, 因为它是线程不安全的。这是基于 `Servlet` 的 `Web` 站点的典型情况。

出于向后兼容的特性, 共享变量的集合初始化时 (就是对于新的 `Configuration` 实例来说) 不能为空。它包含下列用户自定义指令 (用户自定义指令使用时需要用 `@` 来代替 `#`):

名称	类
<code>capture_output</code>	<code>freemarker.template.utility.CaptureOutput</code>
<code>compress</code>	<code>freemarker.template.utility.StandardCompress</code>
<code>html_escape</code>	<code>freemarker.template.utility.HtmlEscape</code>
<code>normalize_newlines</code>	<code>freemarker.template.utility.NormalizeNewlines</code>
<code>xml_escape</code>	<code>freemarker.template.utility.XmlEscape</code>

### 3.3 配置信息

**Settings 配置信息** 是影响 `FreeMarker` 行为的被命名的值。配置信息的例子有: `locale`, `number_format`。

配置信息信息存储在 `Configuration` 实例中, 可以在 `Template` 实例中被覆盖。例如在配置对象中给 `locale` 设置为 `"en_US"`, 那么 `locale` 在所有模板中都使用 `"en_US"` 的配置, 除非在模板中 `locale` 被明确地设置成不同的 (参见本地化设置)。因此, 在 `Configuration` 中的值充当默认值, 这些值在每个模板中也可以被覆盖。在 `Configuration` 和 `Template` 实例中的值也可以在单独调用 `Template.process` 方法后被覆盖。对于每个调用了

`freemarker.core.Environment` 对象的值在内部创建时就持有模板执行的运行时环境，也包括了那个级别被覆盖了的设置信息。在模板执行时，那里存储的值也可以被改变，所以模板本身也可以设置配置信息，比如在输出中途变换 `locale` 设置。

配置信息可以被想象成 3 层 (`Configuration`, `Template`, `Environment`)，最高层包含特定的值，它为设置信息提供最有效的值。比如（设置信息 A 到 F 仅仅是为这个示例而构想的）：

	Setting A	Setting B	Setting C	Setting D	Setting E	Setting F
Layer 3: <code>Environment</code>	1	-	-	1	-	-
Layer 2: <code>Template</code>	2	2	-	-	2	-
Layer 1: <code>Configuration</code>	3	3	3	3	-	-

配置信息的有效值为：A=1，B=2，C=3，D=1，F=2。而 F 的设置则是 `null`，或者在你获取它的时候将抛出异常。

让我们看看如何准确设置配置信息：

`Configuration` 层：原则上设置配置信息时使用 `Configuration` 对象的 `setter` 方法，例如：

```
Configuration myCfg = new Configuration();
myCfg.setLocale(java.util.Locale.ITALY);
myCfg.setNumberFormat("0.####");
```

在真正使用 `Configuration` 对象（通常在初始化应用程序时）之前来配置它，后面必须将其视为只读的对象。

在实践中，比如很多 Web 应用框架中，就应该使用这种框架特定的配置方式来进行配置，如使用成对字符串来配置（像在 `.properties` 文件中那样）。这种情况下，框架的作者大多数使用 `Configuration` 对象的 `setSetting(String name, String value)` 方法。可以参加 API 文档的 `setSetting` 部分来获取可用的设置名和参数的格式。在 Spring 框架中，我们可以这样进行：

```
<bean id="freemarkerConfig"
class="org.springframework.web.servlet.view.freemarker.Free
MarkerConfigurer">
  <property name="freemarkerSettings">
    <props>
      <prop key="locale">it_IT</prop>
      <prop key="number_format">0.####</prop>
    </props>
  </property>
</bean>
```

这种形式的配置（`String` 键-值对）和直接使用 `Configuration` 的 API 相比，很不幸的被限制了。

`Template` 层：这里不需要设置配置信息，除非想替代 `freemarker.cache.TemplateCache` 来管理 `Template` 对象，这样的话，应该在 `Template` 对象第一次被使用前就设置配置信息，然后就将 `Template` 对象视为是只

读的。

`Environment` 层：有两种配置方法：

- 使用 Java API：使用 `Environment` 对象的 `setter` 方法。当然想要在模板执行之前来做，然后当调用 `myTemplate.process(...)` 时会遇到问题，因为在内部创建 `Environment` 对象后立即就执行模板了，导致没有机会来进行设置。这个问题的解决可以用下面两个步骤进行：

```
Environment env =
myTemplate.createProcessingEnvironment(root, out);
env.setLocale(java.util.Locale.ITALY);
env.setNumberFormat("0.####");
env.process(); // process the template
```

- 在模板中直接使用指令，例如：

```
<#setting locale="it_IT">
<#setting number_format="0.####">
```

在这层，当什么时候改变配置信息，没有限制。

要知道 `FreeMarker` 支持什么样的配置信息，先看看 `FreeMarker Java API` 文档中的下面这部分内容：

- 在三层中 `freemarker.core.Configurable` 的 `setter` 方法来配置。
- 只在 `Configuration` 层可用用的 `freemarker.template.Configuration` 的 `setter` 方法来配置。
- 在三层中可用 `String` 键 - 值对书写的 `freemarker.core.Configurable.setSetting(String, String)` 配置。
- 只在 `Configuration` 层中可用用 `String` 键 - 值对书写的 `freemarker.template.Configuration.setSetting(String, String)` 配置。

## 3.4 模板加载

### 3.4.1 模板加载器

模板加载器是加载基于抽象模板路径如 `"index.ftl"` 或 `"products/catalog.ftl"` 的原生文本数据对象。这由具体的模板加载器对象来确定，它们取得请求数据时使用了什么样的数据来源（文件夹中的文件，数据，等等）。当调用 `cfg.getTemplate`（这里的 `cfg` 就是 `Configuration` 实例）时，`FreeMarker` 询问模板加载器是否已经为 `cfg` 建立返回给定模板路径的文本，之后 `FreeMarker` 解析文本生成模板。

### 3.4.1.1 内建模板加载器

在 `Configuration` 中可以使用下面方便的方法来建立三种模板加载方法。(每种方法都会在内部新建一个模板加载器对象，然后建立 `Configuration` 实例来使用它。)

```
void setDirectoryForTemplateLoading(File dir);
```

或

```
void setClassForTemplateLoading(Class cl, String prefix);
```

或

```
void setServletContextForTemplateLoading(Object  
servletContext, String path);
```

上述的第一种方法在文件系统上设置了一个明确的目录，它确定了从哪里加载模板。不要说可能，`File` 参数肯定是一个存在的目录。否则，将会抛出异常。

第二种调用方法使用了一个 `Class` 类型的参数和一个前缀。这是让你来指定什么时候通过相同的机制来加载模板，不过是用 Java 的 `ClassLoader` 来加载类。这就意味着传入的 `Class` 参数会被用来调用 `Class.getResource()` 方法来找到模板。参数 `prefix` 是给模板的名称来加前缀的。在生产环境中，类加载机制是首选用来加载模板的方法，因为通常情况下，从类路径下加载这种机制，要比从文件系统的特定目录位置加载安全且简单。在最终的应用程序中，所有代码都使用 `.jar` 文件也是不错的，这样用户就可以直接执行包含所有资源的 `.jar` 文件了。

第三种调用方式需要 Web 应用的上下文和一个基路径作为参数，这个基路径是 Web 应用根路径（`WEB-INF` 目录的上级目录）的相对路径。那么加载器将会从 Web 应用目录开始加载模板。尽管加载方法对没有打包的 `.war` 文件起作用，因为它使用了 `ServletContext.getResource()` 方法来访问模板，注意这里我们指的是“目录”。如果忽略了第二个参数（或使用了“”），那么就可以混合存储静态文件（`.html`，`.jpg` 等）和 `.ftl` 文件，只是 `.ftl` 文件可以被送到客户端执行。当然必须在 `WEB-INF/web.xml` 中指定一个 `Servlet` 来处理 URI 格式为 `*.ftl` 的请求，否则客户端无法获取到模板，因此你将会看到 Web 服务器给出的秘密提示内容。在站点中不能使用空路径，这将成为一个问题，你应该在 `WEB-INF` 目录下的某个位置存储模板文件，这样模板源文件就会偶然地被执行到，而且模板可以自动更新而不需重启 Web 应用程序，但是对于类加载机制，这样就行不通了。

### 3.4.1.2 从多个位置加载模板

如果需要从多个位置加载模板，那就不得不为每个位置都实例化模板加载器对象，将它们包装到一个被成为 `MultiTemplateLoader` 的特殊模板加载器，最终将之传递给 `Configuration` 对象的 `setTemplateLoader(TemplateLoader loader)` 方法。下面给出一个使用类加载器从两个不同位置加载模板的示例：

```
import freemarker.cache.*; // template loaders live in this
package
...
FileTemplateLoader ftl1 = new FileTemplateLoader(new
File("/tmp/templates"));
FileTemplateLoader ftl2 = new FileTemplateLoader(new
File("/usr/data/templates"));
ClassTemplateLoader ctl = new ClassTemplateLoader(getClass(),
"" );
TemplateLoader[] loaders = new TemplateLoader[] { ftl1, ftl2,
ctl };
MultiTemplateLoader mtl = new MultiTemplateLoader(loaders);
cfg.setTemplateLoader(mtl);
```

现在，FreeMarker 将会尝试从 `/tmp/templates` 目录加载模板，如果在这个目录下不能发现请求的模板，它就会继续尝试从 `/usr/data/templates` 目录下加载，如果还是不能发现请求的模板，那么它就会使用类加载器来加载。

### 3.4.1.3 从其他资源加载模板

如果内建的类加载器都不适合你，那么就需要来写自己的类加载器了，这个类需要实现 `freemarker.cache.TemplateLoader` 接口，然后将它传递给 `Configuration` 对象的 `setTemplateLoader(TemplateLoader loader)` 方法。可以阅读 FreeMarker API 文档获取更多信息。

如果你的模板需要通过 URL 访问其他模板，那么就不需要实现 `TemplateLoader` 接口了，可以选择子接口 `freemarker.cache.URLTemplateLoader` 来替代，只需实现 `URL getURL(String templateName)` 方法即可。

### 3.4.1.4 模板路径

解析模板路径是由模板解析器来决定的。但是要和其他对路径的格式要求很严格的组件一起工作。通常来说，强烈建议模板加载器使用 URL 风格的路径。在 URL 路径（或在 `UNIX` 路径）中有其他含义时，那么路径中不要使用 `/`，`./`，`../` 和 `://`。字符 `*` 和 `?` 是被保留的。而且，模板加载器也不想模板以 `/` 开始；FreeMarker 从来不会用这样的路径来调用模板加载器。FreeMarker 再将路径传递给模板加载器之前通常会将路径进行正常化操作，所以路径中不会含有 `../` 这些，相对于假想的模板根目录。

注意 FreeMarker 模板加载时经常使用斜线（而不是反斜线），不管运行的主机操作系统是什么。

## 3.4.2 模板缓存

FreeMarker 缓存模板（假设使用 `Configuration` 对象的方法来创建 `Template`

对象)。这意味着当调用 `getTemplate` 时，FreeMarker 不但返回 `Template` 对象的结果，而且将它存储在缓存中，当下一次再以相同（或相等）路径调用 `getTemplate` 时，那么它只返回缓存的 `Template` 实例，而不会再次加载和解析模板文件了。

如果更改了模板文件，当下次调用模板时，FreeMarker 将会自动重新载入和解析模板。然而，要检查模板文件是否改变是需要时间的，有一个 `Configuration` 级别的设置称为“更新延迟”。这个时间就是从上对某个模板检查更新后 FreeMarker 再次检查所要消耗的时间。这个时间的默认值是 5 秒。如果想要看到模板立即更新，那么就设置它为 0。要注意某些模板加载器也许在模板更新时会有问题。例如，典型的基于类加载器的模板加载器就不会注意到模板文件的改变。

当调用了 `getTemplate` 方法，与此同时 FreeMarker 意识到这个模板文件已经被移除了，所以这个模板也会从缓存中移除。如果 Java 虚拟机认为会有内存溢出时，默认情况它会将任意的模板从缓存中移除。此外，你还可以使用 `Configuration` 对象的 `clearTemplateCache` 方法手动清空缓存。

何时将一个被缓存了的模板清除的实际策略是由配置的属性 `cache_storage` 来确定的，通过这个属性可以配置任何 `CacheStorage` 实现。对于大多数用户来说，使用 `freemarker.cache.MruCacheStorage` 就足够了。这个缓存存储实现了二级最近使用的缓存。在第一级缓存中，组件都被强烈引用到特定的最大数目（强烈引用的组件可以被 Java 虚拟机抛弃，而轻微引用的组件则想法）。当超过最大数量时，最近被使用的组件将被送至二级缓存中，那里它们被轻微的引用，直到达到另一个很高的数目。强烈和轻微的大小可以由构造方法来指定。例如，设置强烈部分为 20，轻微部分为 250：

```
cfg.setCacheStorage(new freemarker.cache.MruCacheStorage(20,
250))
```

或者，使用 `MruCacheStorage`，它是默认的缓存存储实现。

```
cfg.setSetting(Configuration.CACHE_STORAGE_KEY, "strong:20,
soft:250");
```

当创建了一个新的 `Configuration` 对象时，它使用一个 `maxStrongSize` 值为 0 的 `MruCacheStorage` 来初始化，`maxSoftSize` 的值是 `Integer.MAX_VALUE`（也就是说在实际中，是无限大的）。但是使用非 0 的 `maxStrongSize` 对于高负载的服务器来说也许是一个更好的策略，对于轻微引用的组件来说，如果资源消耗已经很高的话，Java 虚拟机往往会引起更高的资源消耗，因为它不断从缓存中抛出经常使用的模板，这些模板还不得不再次加载和解析。

## 3.5 错误控制

### 3.5.1 可能的异常

关于 FreeMarker 发生的异常，可以分为如下几类：

当配置 FreeMarker 时发生异常：典型地，就是在应用程序初始化时，仅仅配置了一次 FreeMarker。在这个过程中，异常就会发生，从 FreeMarker 的 API 中，我们可以很清楚的看到这一点。

当加载和解析模板时发生异常：当调用了 `Configuration.getTemplate(...)`



方法，FreeMarker 就要把模板文件加载到内容中然后来解析它（除非模板已经在 `Configuration` 对象中被缓存了）。在这期间，有两种异常可以发生：

- 因模板文件没有找到而发生的 `IOException`，或在读取文件时发生其他的 I/O 问题。比如没有读取文件的权限，或者是磁盘错误。这些错误的发出者是 `TemplateLoader` 对象，可以将它设置到 `Configuration` 对象中。（为了正确起见：这里所说的“文件”，是简化形式。例如，模板也可以存储在关系型数据库的表中。这是 `TemplateLoader` 所做的事。）
- 根据 FTL 的语言规则，模板文件发生语法不正确而导致的 `freemarker.core.ParseException` 异常。当获得 `Template` 对象（`Configuration.getTemplate(...)`）时，这种错误就会发生，而不是当执行（`Template.process(...)`）模板的时候。这种异常是 `IOException` 的一个子类。

当执行（处理）模板时发生的异常，也就是当调用了 `Template.process(...)` 时会发生的两种异常：

当试图写入输出对象时发生错误而导致的 `IOException`。

当执行模板时发生的其他问题而导致的 `freemarker.template.TemplateException`。比如，一个频繁发生的错误是当模板引用一个不存在的变量。默认情况下，当 `TemplateException` 发生，FreeMarker 会用普通文本格式打印出 FTL 的错误信息和堆栈跟踪信息到输出中。然后通过再次抛出 `TemplateException` 而中止模板的执行，然后就可以捕捉到 `Template.process(...)` 方法抛出的异常。但是这种行为可以来定制。FreeMarker 也会经常写 `TemplateException-s` 的日志。

### 3.5.2 根据 `TemplateException-s` 来制定处理行为

`TemplateException-s` 在模板处理期间的抛出由 `freemarker.template.TemplateExceptionHandler` 对象控制，这个对象可以使用 `setTemplateExceptionHandler(...)` 方法配置到 `Configuration` 对象中。`TemplateExceptionHandler` 对象只包含一个方法：

```
void handleTemplateException(TemplateException te, Environment env, Writer out) throws TemplateException;
```

无论 `TemplateException` 什么时候发生，这个方法都会被调用。异常处理是传递的 `te` 参数控制的，模板处理的运行时环境可以访问 `env` 变量，处理器可以使用 `out` 变量来打印输出信息。如果方法抛出异常（通常是重复抛出 `te`），那么模板执行就会中止，而且 `Template.process(...)` 也会抛出同样的异常。如果 `handleTemplateException` 对象不抛出异常，那么模板执行将会继续，就好像什么也没有发生一样，但是引发异常的语句将会跳过（后面详细说）。当然，控制器仍然可以在输出中打印错误指示信息。

任何一种情况下，当 `TemplateExceptionHandler` 被调用前，FreeMarker 将会记录异常日志。

让我们用实例来看一下当错误控制器不抛出异常时，FreeMarker 是如何跳过出错语句的。假设我们已经使用了如下模板异常控制器：

```

class MyTemplateExceptionHandler implements
TemplateExceptionHandler {
    public void handleTemplateException(TemplateException te,
Environment env, java.io.Writer out)
        throws TemplateException {
        try {
            out.write("[ERROR: " + te.getMessage() + "]);
        } catch (IOException e) {
            throw new TemplateException("Failed to print error
message. Cause: " + e, env);
        }
    }
}
...
cfg.setTemplateExceptionHandler(new
MyTemplateExceptionHandler());

```

如果错误发生在非 `FTL` 标记（没有被包含在`<#...>`或`<@...>`之间）的插值中，那么整个插值将会被跳过。那么下面这个模板（假设 `badVar` 在数据模型中不存在）：

```
a${badVar}b
```

如果我们使用了 `MyTemplateExceptionHandler`，就会打印：

```
a[ERROR: Expression badVar is undefined on line 1, column 4 in
test.ftl.]b
```

下面这个模板也会打印相同信息（除了报错的列数会不同）：

```
a{"moo" + badVar}b
```

因为像这样来写，只要插值内发生任何错误，整个插值都会被跳过。

如果错误发生在指令调用中参数的计算时，或者是指令参数列表发生问题，或在 `<@exp ...>` 中计算 `exp` 时发生错误，或者 `exp` 是用户自定义的指令，那么整个指令调用都会被跳过。例如：

```
a<#if badVar>Foo</if>b
```

会打印：

```
a[ERROR: Expression badVar is undefined on line 1, column 7 in
test.ftl.]b
```

要注意在错误发生在 `if` 指令的开始标签（`<#if badVar>`）中，但是整个指令的调用都被跳过了。从逻辑上说，嵌套的内（`Foo`）容被跳过了，因为嵌套的内容是受被包含的指令（`if`）控制（打印）的。

下面这个的输出也是相同的（除了报错的列数会不同）：

```
a<#if "foo${badVar}" == "foobar">Foo</if>b
```



因为正如这样来写的，如果在参数处理时发生任何一个错误，整个指令的调用都将会被跳过。

如果错误发生在已经开的指令执行的之后，那么指令调用将不会被跳过。也就是说，如果在嵌套的内容中发生任何错误：

```
a
<#if true>
  Foo
  ${badVar}
  Bar
</#if>
c
```

或者在一个宏定义体内：

```
a
<@test />
b
<#macro test>
  Foo
  ${badVar}
  Bar
</#macro>
```

那么输出将会是：

```
a
  Foo
  [ERROR: Expression badVar is undefined on line 4, column 5 in
test.ftl.]
  Bar
c
```

FreeMarker 带有这些预先编写的错误控制器：

- `TemplateExceptionHandler.DEBUG_HANDLER`：打印堆栈跟踪信息（包括 `FTL` 错误信息和 `FTL` 堆栈跟踪信息）和重新抛出的异常。这是默认的控制器（也就是说，在所有新的 `Configuration` 对象中，它是初始配置的）。
- `TemplateExceptionHandler.HTML_DEBUG_HANDLER`：和 `DEBUG_HANDLER` 相同，但是它可以格式化堆栈跟踪信息，那么就可以在 `Web` 浏览器中来阅读。当你制作 `HTML` 页面时，建议它使用而不是 `DEBUG_HANDLER`。
- `TemplateExceptionHandler.IGNORE_HANDLER`：简单抑制所有异常（但是要记住，`FreeMarker` 仍然会写日志）。它对处理没有任何作用，也不会重新抛出异常。
- `TemplateExceptionHandler.RETHROW_HANDLER`：简单重新抛出所有异常而不会做其他的事情。这个控制器对 `Web` 应用程序（假设你在发生异常之后不想继续执行模板）来说非常好，因为它在生成页面发生错误的情况下，给你了很多对 `Web` 应用程序的控制权。要获得更多在 `Web` 应用程序中处理错误，可以参

见 FAQ。

### 3.5.3 在模板中明确地处理错误

尽管它和 **FreeMarker** 的配置（本章的主题）无关，为了完整性，在这里提及一下，你可以在模板中直接控制错误。这通常是一个不好的习惯（尽量保持模板简单，技术含量不要太高），但有时仍然需要：

- 控制不存在/为空的变量：模板开发指南/模板/表达式/处理不存在的值
- 在发生障碍的“porlets”中留存下来，还有这样可阅读的部分：参考手册/指令参考/尝试恢复

## 第四章 其它

章节内容如下：

变量

字符集问题

多线程

Bean 的包装

日志记录

以 **Serlvets** 来使用 **FreeMarker**

为 **FreeMarker** 配置安全策略

遗留的 **XML** 包装实现

以 **Ant** 来使用 **FreeMarker**

**Jython** 包装

这只是一个介绍性的导引，可以查看 **FreeMarker** 的 **API** 文档获取详细信息。

### 4.1 变量

本章介绍当模板在访问变量时发生了什么，还有变量是如何存储的。

当调用 **Template.process** 时，它会在方法内部创建一个 **Environment** 对象，在 **process** 返回之前一直使用。**Environment** 对象存储模板执行时的运行状态。除了这些，它还存储由模板中指令，如 **assign**，**macro**，**local** 或 **global** 创建的变量。它从来不会改变你传递给 **process** 的数据模型对象，也不会创建或替换存储在配置中的共享变量。

当你想要读取一个变量时，**FreeMarker** 将会以这种顺序来查找，直到发现了完全匹配的变量名称才会停下来：

1. 在 **Environment** 对象中：

1. 如果在循环中，在循环变量的集合中。循环变量是由如 **list** 指令创建的变量。
2. 如果在宏中，在宏的局部变量集合中。局部变量可以由 **local** 指令创建。而且，宏的参数也是局部变量。

3. 在当前的命名空间中。可以使用 `assign` 指令将变量放到一个命名空间中。
4. 由 `global` 指令创建的变量集合中。FTL 将它们视为数据模型的普通成员变量一样来控制这些变量。也就是说，它们在所有的命名空间中都可见，你也可以如访问一个数据模型中的数据一样来访问它们。
2. 在传递给 `process` 方法的数据模型对象中。
3. 在 `Configuration` 对象存储的共享变量集合中。

在实践中，来自模板设计者的观点这 6 种情况应该只是 4 种，因为从那种观点来看，后面 3 种（由 `global` 创建的变量，真实的数据模型对象，共享变量）共同构成了全局变量的集合。

要注意在 FTL 中从确定层面获取确定的变量是可以的。

## 4.2 字符集问题

FreeMarker，像其他大多数 Java 应用程序一样，使用“UNICODE 文本”（UTF-16）来工作。不过，也有必须处理字符集的情况，因为它不得和外界叫唤数据，这就会使用到很多字符集。

### 4.2.1 输入的字符集

当 FreeMarker 要加载模板文件（或没有解析的文本文件），那就必须要知道文件使用的字符集，因为文件的存储是原生的字节数组形式。可以使用配置项 `encoding` 来确定字符集。这个配置项只在 FreeMarker 使用 `Configuration` 对象的 `getTemplate` 方法加载模板（解析过的或没解析过的）起作用。要注意 `include` 指令在内部也使用了这个方法，所以 `encoding` 的值对一个已经加载的模板（如果这个模板包含 `include` 指令的调用）来说很重要。

`encoding` 配置的 `getter` 和 `setter` 方法在第一个（配置）层面很特殊。`getter` 方法猜想返回值是基于 `Locale` 传递的参数；它在地图区域编码表（称为编码地图）中查询编码，如果区域没有找到，就返回默认编码。可以使用配置对象的 `setEncoding(Locale locale, String encoding)` 方法来填充编码表；编码表初始化时是空的。默认的初始编码是系统属性 `file.encoding` 的值，但是可以通过 `setDefaultEncoding` 方法来设置一个不同的默认值。

你也可以在模板层或运行环境层（当指定编码值作为 `getTemplate` 方法的参数时，应该在模板层覆盖 `encoding` 设置）直接给定值来覆盖 `encoding` 设置。如果不覆盖它，那么 `locale` 设置的有效值将会是 `configuration.getEncoding(Locale)` 方法的返回值。

而且，代替这种基于字符猜测的机制，你也可以在模板文件中使用 `ftl` 指令来指定特定的字符集。

也许你想知道为模板选择什么样的字符集。这主要是依赖于你用来创建和修改模板的工具（如文本编辑器）。原则上，使用 UTF-8 是最好的，但是 2004 年时只有很少的一部分工具支持 UTF-8，很多都不支持这种字符集。所以那种情况下就要使用本土语言中使用最广泛的字符集，这也许是你工作环境中默认使用的字符集。

要注意模板使用的字符集和模板生成的输出内容的字符集是独立的（除非包含 FreeMarker 软件故意将设置输出内容的字符集设置成和模板字符集相同）。

## 4.2.2 输出的字符集

注意：

`output_encoding` 设置/参数和内建函数 `url` 从 FreeMarker 2.3.1 开始可用，而在 2.3 以前版本中不存在。

原则上，FreeMarker 不处理输出内容的字符集问题，因为 FreeMarker 将输出内容都写入了 `java.io.Writer` 对象。而 `Writer` 对象是由封装了 FreeMarker（比如 Web 应用框架）的软件生成的，那么输出内容的字符集就是由封装软件来控制的。而 FreeMarker 有一个称为 `output_encoding`（开始于 FreeMarker 2.3.1 版本）的设置。封装软件应该使用这个设置（`Writer` 对象使用的字符集）来通知 FreeMarker 在输出中（否则 FreeMarker 不能找到它）使用哪种字符集。有一些新特性，如内建函数 `url`，特殊变量 `output_encoding` 也利用这个信息。因此，如果封装软件没有设置字符集这个信息，那么 FreeMarker 需要知道输出字符集的特性就不能被使用了。

如果你使用 FreeMarker 来编写软件，你也许想知道输出内容中选择哪种字符集。当然这取决于 FreeMarker 输出内容的计算机，但是如果用户对这个问题可以变通，那么通用的实践是使用模板文件的字符集作为输出的字符集，或者使用 UTF-8。通常使用 UTF-8 是最佳实践，因为任意的文本可能来自数据模型，那就可能包含不能被模板字符集所编码的字符。

如果使用了 `Template.createProcessingEnvironment(...)` 和 `Environment.process(...)` 来代替 `Template.process(...)`，FreeMarker 的设置可以对任意独立执行的模板进行。因此，你可以对每个独立执行的模板设置 `output_encoding` 信息：

```
Writer w = new OutputStreamWriter(out, outputCharset);
Environment env =
    template.createProcessingEnvironment(dataModel, w);
env.setOutputEncoding(outputCharset);
env.process();
```

## 4.3 多线程

在多线程环境中，`Configuration` 实例，`Template` 实例和数据模型应该是永远不能改变（只读）的对象。也就是说创建和初始化它们（如使用 `set...` 方法），之后就不能修改它们（如不能再次调用 `set...` 方法）。这就允许我们在多线程环境中避免代价很大的同步锁。要小心 `Template` 实例；当你使用 `Configuration.getTemplate` 方法获得一个实例时，也许得到的是从模板缓存中的实例，这些实例都已经被其他线程使用，所以不要调用它们的 `set...` 方法（调用 `process` 当然还是不错的）。

如果你只从同一个线程中访问所有对象，那么上面所述的限制将不会起作用。

使用 FTL 来修改数据模型对象或者共享变量是不太可能的，除非将方法（或其他对象）放到数据模型中来做。我们不鼓励你编写修改数据模型对象或共享变量的方法。多试试使用存储在环境对象（这个对象是为独立的 `Template.process` 调用而创建的，用来存储模板处理的运行状态）中的变量，所以最好不要修改那些由多线程使用的数据。要获取更多信息，请阅读：4.1 章节（变量）。

## 4.4 Bean 的包装

`freemarker.ext.beans.BeansWrapper` 是一个对象包装器，最初加到 FreeMarker 中是为了将任意的 POJO-s (Plain Old Java Objects 普通 Java 对象) 包装成 `TemplateModel` 接口类型。这样它就可以以正常的方式来进行处理，事实上 `DefaultObjectWrapper` 本身是 `BeansWrapper` 的扩展类。这里描述的所有东西对 `DefaultObjectWrapper` 都适用，除了 `DefaultObjectWrapper` 会用 `freemarker.template.SimpleXxx` 类包装 `String`, `Number`, `Date`, `array`, `Collection` (如 `List`), `Map`, `Boolean` 和 `Iterator` 对象，会用 `with freemarker.ext.dom.NodeModel` 来包装 W3C 的 DOM 节点，所以上述这些描述的规则不适用。

当下面这些情况下，你会想使用 `BeansWrapper` 来代替 `DefaultObjectWrapper`：

在模板执行期间，模型中的 `Collection-s` 和 `Map-s` 应该被允许修改。（`DefaultObjectWrapper` 会阻止这样做，因为当它包装对象时创建了数据集合的拷贝，而这些拷贝都是只读的。）

如果 `array`, `Collection` 和 `Map` 对象的标识符当在模板中被传递到被包装对象的方法时，必须被保持下来。也就是说，那些方法必须得到之前包装好的同类对象。

如果之前列出的 Java API 中的类 (`String`, `Map`, `List` 等) 应该在模板中可见时。还有，默认情况下它们是不可见的，但是可以设置暴露水平 (后面介绍)。要注意这不是一个好的实践，尽量使用内建函数 (如 `foo?size`, `foo?upper`, `foo?replace('_', '-')` 等) 来代替 Java API 的使用。

下面是对 `BeansWrapper` 创建的 `TemplateModel-s` 进行总结。为了后续的讨论，这里我们假设在包装之前对象都称为 `obj`，而包装后称为 `model`。

### 4.4.1 TemplateHashModel functionality 模板哈希表模型

所有的对象都将被包装成 `TemplateHashModel` 类型，进而可以暴露出 JavaBeans 对象中的属性和方法。这样，就可以在模板中使用 `model.foo` 来调用 `obj.getFoo()` 或 `obj.isFoo()` 方法。（要注意公有的属性直接是不可见的，必须为它们写 `getter` 方法才行）公有方法通过哈希表模型来取得，就像模板方法模型那样，因此可以使用 `model.doBar()` 来调用 `object.doBar()`。下面我们来更多讨论一下方法模型功能。

如果请求的键值不能映射到一个 bean 的属性或方法时，那么框架将试图定位到“通用的 `get` 方法”，这个方法的签名是 `public any-return-type get(String)` 或 `public any-return-type get(Object)`，使用请求键值来调用它们。这样就使得访问 `java.util.Map` 和其他类似类型的键值对非常便利。只要 `map` 的键是 `Strings`，属性和方法名可以在映射中查到。（有一种解决方法可以用来避免在映射中遮挡名称，继续阅读。）要注意 `java.util.ResourceBundle` 对象的方法使用 `getObject(String)` 作为通用 `get` 方法。

如果在 `BeansWrapper` 实例中调用了 `setExposeFields(true)` 方法，那么它仍然会暴露出类的公有的，非静态变量，用它们作为哈希表的键和值。即如果 `foo` 是类 `Bar` 的一个公有的，非静态的变量，而 `bar` 是一个包装了 `Bar` 实例模板变量，那么表达式 `bar.foo` 的值将会作为 `bar` 对象中 `foo` 变量的值。所有这个类的超类中公有变量都

会被暴露出来。

#### 4.4.2 说一点安全性

默认情况下，不能访问模板制作时认为不安全的一些方法。比如，不能使用同步方法（`wait`, `notify`, `notifyAll`），线程和线程组管理方法（`stop`, `suspend`, `resume`, `setDaemon`, `setPriority`），反射（`Field` `setXxx`, `Method`.`invoke`, `Constructor`.`newInstance` , `Class`.`newInstance` , `Class`.`getClassLoader` 等），`System`和`Runtime`类中各种危险的方法（`exec`, `exit`, `halt`, `load` 等）。`BeansWrapper`也有一些安全级别（被称作“方法暴露级别”），默认的级别被称作 `EXPOSE_SAFE`，它可能对大多数应用程序来说是适用的。没有安全保证的级别称作是，它允许你调用上述的不安全的方法。一个严格的级别 `EXPOSE_PROPERTIES_ONLY`，它只会暴露出 `bean` 属性的 `getters` 方法。最后，一个称作是 `EXPOSE_NOTHING` 的级别，它不会暴露任何属性和方法。这种情况下，你可以通过哈希表模型接口访问的那些数据只是 `map` 和资源包中的项，还有从通用 `get (Object)` 和 `get (String)` 方法调用返回的对象，所提供的受影响的对象有这样的方法。

#### 4.4.3 TemplateScalarModel functionality 模板标量模型

对于 `java.lang.String` 对象的模型会实现接口 `TemplateScalarModel`，这个接口中方法 `getAsString()` 简单代替了方法 `toString()`。要注意把 `String` 对象包装到 `Bean` 包装器中，要提供比它们作为标量时更多的功能：因为哈希表接口描述了上述所需功能，那么包装 `Strings` 的模型也会提供访问所有 `String` 的方法（`indexOf`, `substring` 等）。

#### 4.4.4 TemplateNumberModel functionality 模板数字模型

对于是 `java.lang.Number` 的实例对象的模型包装器，它们实现了 `TemplateNumberModel` 接口，接口中的 `getAsNumber()` 方法返回被包装的数字对象。要注意把 `Number` 对象包装到 `Bean` 包装器中，要提供比它们作为数字时更多的功能：因为哈希表接口描述了上述所需功能，那么包装 `Numbers` 的模型也会提供访问所有 `Number` 的方法。

#### 4.4.5 TemplateCollectionModel functionality 模板集合模型

对于本地的 `Java` 数组和其他所有实现 `java.util.Collection` 接口的类的模型包装器，都实现了 `TemplateCollectionModel` 接口，因此增强了使用 `list` 指令的附加功能。



#### 4.4.6 TemplateSequenceModel functionality 模板序列模型

对于本地的 Java 数组和其他所有实现 `java.util.List` 接口的类的模型包装器，都实现了 `TemplateSequenceModel` 接口，这样它们之中的元素就可以使用 `model[i]` 这样的语法通过索引来访问。你也可以使用内建函数 `model?size` 来查询数组的长度和列表的大小。

而且，所有方法用可指定的一个单独的参数，从 `java.lang.Integer`（即 `int`, `long`, `float`, `double`, `java.lang.Object`, `java.lang.Number`, `java.lang.Integer`）中通过反射方法调用，这些方法也实现了这个接口。这就意味着你可以通过很方便的方式来访问被索引的 bean 属性：`model.foo[i]` 将会翻译为 `obj.getFoo(i)`。

#### 4.4.7 TemplateMethodModel functionality 模板方法模型

一个对象的所有方法作为访问 `TemplateMethodModelEx` 对象的代表，它们在对象模型的方法名中使用哈希表的键。当你使用 `model.method(arg1, arg2, ...)` 来调用方法时，形式参数被作为模板模型传递给方法。方法首先不会包装它们，后面我们会说到展开包装的详细内容。这些不被包装的参数之后被实际方法来调用。以防方法被重载，许多特定的方法将会被选择使用相同的规则，也就是 Java 编译器从一些重载的方法中选择一个方法。以防没有方法签名匹配传递的参数，或者没有方法可以被无歧义地选择，`TemplateModelException` 异常将会抛出。

返回值类型为 `void` 的方法返回 `TemplateModel.NOTHING`，那么它们就可以使用 `${obj.method(args)}` 语法被安全地调用。

`java.util.Map` 实例的模型仍然实现了 `TemplateMethodModelEx` 接口，作为调用它们 `get()` 方法的一种方式。正如前面所讨论的那样，你可以使用哈希表功能来访问“get”方法，但是它有一些缺点：因为第一个属性和方法名被键来检查，所以执行过慢；和属性，方法名冲突的键将会被隐藏；最终这种方法中你只可使用 `String` 类型的键。对比一下，调用 `model(key)`，将直接翻译为 `model.get(key)`：因为没有属性和方法名的查找，速度会很快；不容易被隐藏；最终对非字符串的键也能正常工作，因为参数没有被包装，只是被普通的方法调用。实际上，`Map` 中的 `model(key)` 和 `model.get(key)` 是相等的，只是写起来很短。

`java.util.ResourceBundle` 类的模型也实现了 `TemplateMethodModelEx` 接口，作为一种访问资源和信息格式化的方便形式。对资源束的单参数调用，将会取回名称和未包装参数的 `toString()` 方法值一致的资源。对资源束的多参数调用的情况和单参数一样，但是它会将参数作为格式化的模式传递给 `java.text.MessageFormat`，在第二个和后面的作为格式化的参数中使用未包装的值。`MessageFormat` 对象将会使用它们本来的本地化资源束来初始化。

#### 4.4.8 展开规则

当从模板中调用 Java 方法时，它的参数需要从模板模型转换回 Java 对象。假设目标类型（方法常规参数被声明的类型）是用来 `T` 代表的，下面的规则将会按下述的顺序进行依

次尝试:

- 对包装器来说, 如果模型是空模型, 就返回 Java 中的 `null`。
- 如果模型实现了 `AdapterTemplateModel` 接口, 如果它是 `T` 的实例, 或者它是一个数字而且可以使用数字强制转换成 `T`, 那么 `model.getAdaptedObject(T)` 的结果会返回。由 `BeansWrapper` 创建的所有方法是 `AdapterTemplateModel` 的实现, 所以由 `BeansWrapper` 为基本的 Java 对象创建的展开模型通常不如原本的 Java 对象。
- 如果模型实现了已经废弃的 `WrapperTemplateModel` 接口, 如果它是 `T` 的实例, 或者它是一个数字而且可以使用数字强制转换成 `T`, 那么 `model.getWrappedObject()` 方法的结果会返回。
- 如果 `T` 是 `java.lang.String` 类型, 那么如果模型实现了 `TemplateScalarModel` 接口, 它的字符串值将会返回。注意如果模型没有实现接口, 我们不能尝试使用 `String.valueOf(model)` 方法自动转换模型到 `String` 类型。这里不得不使用内建函数 `string` 明确地用字符串来处理非标量。
- 如果 `T` 是原始的数字类型或者是可由 `T` 指定的 `java.lang.Number` 类型, 还有模型实现了 `TemplateNumberModel` 接口, 如果它是 `T` 的实例或者是它的装箱类型 (如果 `T` 是原始类型), 那么它的数字值会返回。否则, 如果 `T` 是一个 Java 内建的数字类型 (原始类型或是 `java.lang.Number` 的标准子类, 包括 `BigInteger` 和 `BigDecimal`), 类型 `T` 的一个新对象或是它的装箱类型会由数字模型的适当强制的值来生成。
- 如果 `T` 是 `boolean` 值或 `java.lang.Boolean` 类型, 模型实现了 `TemplateHashModel` 接口, 那么布尔值将会返回。
- 如果 `T` 是 `java.util.Map` 类型, 模型实现了 `TemplateHashModel` 接口, 那么一个哈希表模型的特殊 `Map` 表示对象将会返回。
- 如果 `T` 是 `java.util.List` 类型, 模型实现了 `TemplateSequenceModel` 接口, 那么一个序列模型的特殊 `List` 表示对象将会返回。
- 如果 `T` 是 `java.util.Set` 类型, 模型实现了 `TemplateCollectionModel` 接口, 那么集合模型的一个特殊 `Set` 表示对象将会返回。
- 如果 `T` 是 `java.util.Collection` 或 `java.lang.Iterable` 类型, 模型实现了 `TemplateCollectionModel` 或 `TemplateSequenceModel` 接口, 那么集合或序列模型 (各自地) 一个特殊的 `Set` 或 `List` 表示对象将会返回。
- 如果 `T` 是 Java 数组类型, 模型实现了 `TemplateSequenceModel` 接口, 那么一个新的指定类型的数组将会创建, 它其中的元素使用数组的组件类型作为 `T`, 递归展开到数组中。
- 如果 `T` 是 `char` 或者 `java.lang.Character` 类型, 模型实现了 `TemplateScalarModel` 接口, 它的字符串表示中包含精确的一个字符, 那么一个 `java.lang.Character` 类型的值将会返回。
- 如果 `T` 定义的是 `java.util.Date` 类型, 模型实现了 `TemplateDateModel` 接口, 而且它的日期值是 `T` 的实例, 那么这个日期值将会返回。
- 如果模型是数字模型, 而且它的数字值是 `T` 的实例, 那么数字值就会返回。你可



以得到一个实现了自定义接口的 `java.lang.Number` 类型的自定义子类，也许 `T` 就是那个接口。

- 如果模型是日期类型，而且它的日期值是 `T` 的实例，那么日期值将会返回。类似的考虑为\*。
- 如果模型是标量类型，而且 `T` 可以从 `java.lang.String` 类型来定义，那么字符串值将会返回。这种情况涵盖 `T` 是 `java.lang.Object`, `java.lang.Comparable` 和 `java.io.Serializable` 类型。
- 如果模型是布尔类型，而且 `T` 可以从 `java.lang.Boolean` 类型来定义，那么布尔值将会返回。和\*\*是相同的。
- 如果模型是哈希表类型，而且 `T` 可以从 `freemarker.ext.beans.HashAdapter` 类型来定义，那么一个哈希表适配器将会返回。和\*\*是形同的。
- 如果模型是序列类型，而且 `T` 可以从 `freemarker.ext.beans.SequenceAdapter` 类型来定义，那么一个序列适配器将会返回。和\*\*是形同的。
- 如果模型是集合类型，而且 `T` 可以从 `freemarker.ext.beans.SetAdapter` 类型来定义，那么集合的 `set` 适配器将会返回。和\*\*是形同的。
- 如果模型是 `T` 的实例，那么模型本身将会返回。这种情况涵盖方法明确地声明一个 `FreeMarker` 特定模型接口，而且允许返回指令，当 `java.lang.Object` 被请求时允许返回方法和转换的模型。
- 意味着没有可能转换的异常被抛出。

#### 4.4.9 访问静态方法

从 `BeansWrapper.getStaticModels()` 方法返回的 `TemplateHashModel` 可以被用于创建哈希表模型来访问静态方法和任意类型的字段。

```
BeansWrapper wrapper = BeansWrapper.getDefaultInstance();
TemplateHashModel staticModels = wrapper.getStaticModels();
TemplateHashModel fileStatics =
    (TemplateHashModel) staticModels.get("java.io.File");
```

之后你就可以得到模板的哈希表模型，它会暴露所有 `java.lang.System` 类的静态方法和静态字段（`final` 类型和非 `final` 类型）作为哈希表的键。设想你已经将之前的模型放到根模型中了：

```
root.put("File", fileStatics);
```

从现在开始，你可以在模板中使用 `${File.SEPARATOR}` 来插入文件分隔符，或者你可以列出所有文件系统根元素，通过：

```
<#list File.listRoots() as fileSystemRoot>...</#list>
```

来进行。

当然，你必须小心这个模型带来的潜在的安全问题。

你可以给模板作者完全的自由，不管它们通过将静态方法的哈希表放到模板的根模型中，来使用哪种类的静态方法，如用如下方式：

```
root.put("statics", BeansWrapper.getDefaultInstance().getStaticModels());
```

如果它被用作是以类名为键的哈希表，这个对象暴露的只是任意类的静态方法。那么你可以在模板中使用如 `${statics["java.lang.System"].currentTimeMillis() }` 这样的表达式。注意，这样会有更多的安全隐患，比如，如果方法暴露级别对 `EXPOSE_ALL` 是很弱的，那么某些人可以使用这个模型调用 `System.exit()`。

注意在上述的示例中，我们通常使用默认的 `BeansWrapper` 实例。这是一个方便使用的静态包装器实例，你可以在很多情况下使用。特别是你想修改一些属性（比如模型缓存，安全级别，或者是空模型对象表示）时，你也可以自由地来创建自己的 `BeansWrapper` 实例，然后用它们来代替默认包装器。

#### 4.4.10 访问枚举类型

在 JRE 1.5 版本之后，从方法 `BeansWrapper.getEnumModels()` 返回的 `TemplateHashModel` 可以被用作创建访问枚举类型值的哈希表模型。（试图在之前 JRE 中调用这个方法会导致 `UnsupportedOperationException` 异常。）

```
BeansWrapper wrapper = BeansWrapper.getDefaultInstance();
TemplateHashModel enumModels = wrapper.getEnumModels();
TemplateHashModel roundingModeEnums =
    (TemplateHashModel) enumModels.get("java.math.RoundingMode");
```

这样你就可以得到模板哈希表模型，它暴露了 `java.math.RoundingMode` 类所有枚举类型的值，并把它们作为哈希表的键。设想你将之前的模型已经放入根模型中了：

```
root.put("RoundingMode", roundingModeEnums);
```

从现在开始，你可以在模板中使用表达式 `RoundingMode.UP` 来引用枚举值 `UP`。

你可以给模板作者完全的自由，不管它们使用哪种枚举类，将枚举模型的哈希表放到模板的根模型中，可以这样做：

```
root.put("enums", BeansWrapper.getDefaultInstance().getEnumModels());
```

如果它被用作是类名作为键的哈希表，这个对象暴露了任意的枚举类。那么你可以在模板中使用如 `${enums["java.math.RoundingMode"].UP}` 的表达式。

被暴露的枚举值可以被用作是标量（它们会委派它们的 `toString()` 方法），也可以用在相同或不同的比较中。

注意在上述的例子中，我们通常使用默认的 `BeansWrapper` 实例。这是一个方便使用的静态包装器实例，你可以在很多情况下使用。特别是你想修改一些属性（比如模型缓存，安全级别，或者是空模型对象表示）时，你也可以自由地来创建自己的 `BeansWrapper` 实例，然后用它们来代替默认包装器。

## 4.5 日志

FreeMarker 整合了如下的日志包：Log4J，Avalon LogKit 和 java.util.logging（Java2 平台 1.4 版本之后）。为了让 FreeMarker 使用这些日志记录器，你不需要做什么事情；如果它们在加载 FreeMarker 类库时被类加载器发现，那它们将会自动被使用。所有 FreeMarker 产生的日志信息，将会被日志的顶级记录器是以 `freemarker` 命名的来记录。现在被使用的记录器是：

记录器名称	目标
<code>freemarker.beans</code>	记录 Beans 包装器模块的日志信息。
<code>freemarker.cache</code>	记录模板加载和缓存相关的日志信息。
<code>freemarker.runtime</code>	记录在模板执行期间抛出的模板异常日志信息。
<code>freemarker.runtime.attempt</code>	记录在模板执行期间抛出的模板异常日志信息，但是是由 <code>attempt/recover</code> 指令捕捉的。开启 DEBUG 严重级别来查看异常。
<code>freemarker.servlet</code>	记录由 <code>FreeMarkerServlet</code> 类产生的日志信息。

FreeMarker 会按照如下顺序来查找日志包，而且会使用第一个发现的日志包：Log4J，Avalon，java.util.logging。然而，如果你使用适当的参数在 `freemarker.log.Logger` 类中调用静态方法 `selectLoggerLibrary`，你可以明确地来选择一种日志包，或者干脆关闭日志记录功能。

你也可以调用 `freemarker.log.Logger` 类中的静态方法 `selectLoggerLibrary`，传递一个字符串来用作是前缀，这是上述提到的日志包名称。如果你想给每个应用使用不同的日志包，这个基本操作会是很有用的。

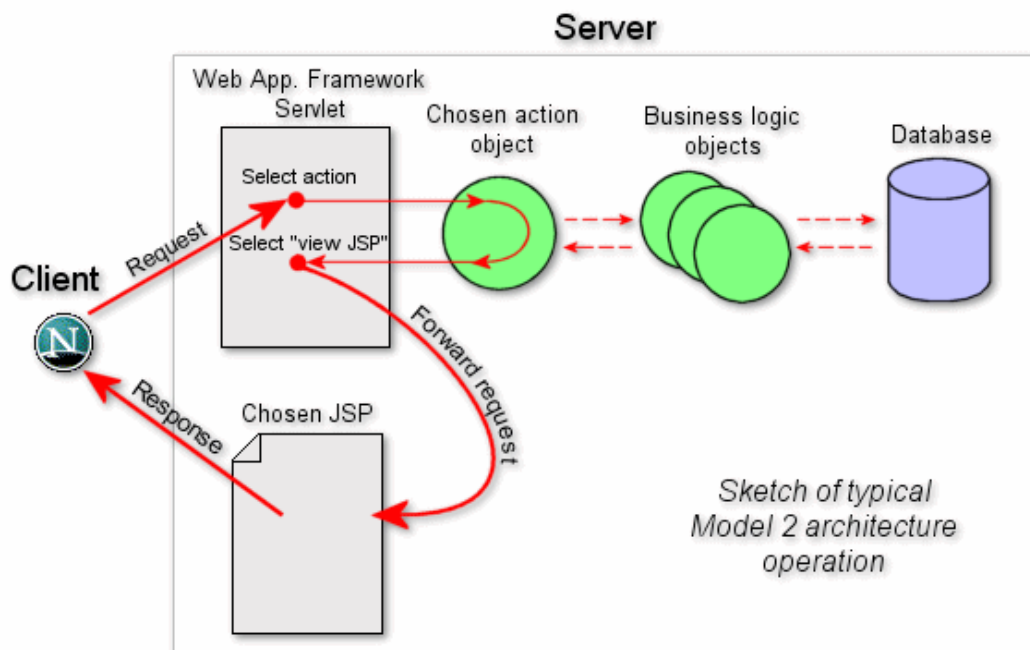
## 4.6 在 servlets 中使用 FreeMarker

作为基础的了解，在 Web 应用领域中使用 FreeMarker 和其他没有什么不同。FreeMarker 将输出内容写到你传递给 `Template.process` 方法的 `Writer` 对象中，它并不关心 `Writer` 对象将输出内容打印到控制台或是一个文件中，或是 `HttpServletResponse` 对象的输出流中。FreeMarker 并不知道 servlets 和 Web；它仅仅是使用模板文件来合并 Java 对象，之后从它们中间生成输出文本。从这里可知，如何创建一个 Web 应用程序都随你的习惯来。

但是，你可能想在已经存在的 Web 应用框架中使用 FreeMarker。许多框架都是基于“Model 2”架构的，JSP 页面来控制显示。如果你使用了这样的框架（不如 Apache Struts），那么继续阅读本文。对于其他框架请参考它们的文档。

### 4.6.1 在“Model 2”中使用 FreeMarker

许多框架依照 HTTP 请求转发给用户自定义的“action”类，将数据作为属性放在 `ServletContext`，`HttpSession` 和 `HttpServletRequest` 对象中，之后请求被框架派发到一个 JSP 页面中（视图层），使用属性传递过来的数据来生成 HTML 页面，这样的策略通常就是所指的 Model 2 模型。



使用这样的框架，你就可以非常容易的用 FTL 文件来代替 JSP 文件。但是，因为你的 Servlet 容器（Web 应用程序服务器），不像 JSP 文件，它可能并不知道如何处理 FTL 文件，那么就需要对 Web 应用程序进行一些额外的配置。

1. 复制 `freemarker.jar` 到（从 FreeMarker 发布包的 `lib` 目录中）你的 Web 应用程序的 `WEB-INF/lib` 目录下。
2. 将下面的部分添加到 Web 应用程序的 `WEB-INF/web.xml` 文件中（调整它是否需要）。

```

<servlet>
  <servlet-name>freemarker</servlet-name>
  <servlet-class>
    freemarker.ext.servlet.FreemarkerServlet
  </servlet-class>
  <!-- FreemarkerServlet 设置: -->
  <init-param>
    <param-name>TemplatePath</param-name>
    <param-value>/</param-value>
  </init-param>
  <init-param>
    <param-name>NoCache</param-name>
    <param-value>true</param-value>
  </init-param>
  <init-param>
    <param-name>ContentType</param-name>
    <param-value>text/html; charset=UTF-8</param-value>
    <!-- 强制使用 UTF-8 作为输出编码格式! -->
  </init-param>
</servlet>
  
```

```

<!-- FreeMarker 设置: -->
<init-param>
  <param-name>template_update_delay</param-name>
  <param-value>0</param-value>
  <!-- 0 只对开发使用! 否则使用大一点的值. -->
</init-param>
<init-param>
  <param-name>default_encoding</param-name>
  <param-value>ISO-8859-1</param-value>
  <!-- 模板文件的编码方式. -->
</init-param>
<init-param>
  <param-name>number_format</param-name>
  <param-value>0.#####</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>freemarker</servlet-name>
  <url-pattern>*.ftl</url-pattern>
</servlet-mapping>

```

这就可以了。配置完后，你可以像使用 JSP (\*.jsp) 文件那样使用 FTL 文件 (\*.ftl) 了。（当然你可以选择除 ftl 之外的扩展名；这只是惯例）

### 注意

它是怎么工作的？让我们来看看 JSP 是怎么工作的。许多 servlet 容器处理 JSP 时使用一个映射为 \*.jsp 的 servlet 请求 URL 格式。这样 servlet 就会接收所有 URL 是以 .jsp 结尾的请求，查找请求 URL 地址中的 JSP 文件，内部编译完后交给 Servlet，然后调用生成信息的 servlet 来生成页面。这里为 URL 类型是 \*.ftl 映射的 FreemarkerServlet 也是相同功能，只是 FTL 文件不会编译给 Servlet-s，而是给 Template 对象，之后 Template 对象的 process 方法就会被调用来生成页面。

比如，代替这个 JSP 页面（注意它使用了 Struts 标签库来保存设计，而不是嵌入可怕的 Java 代码）：

```

<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<html>
<head><title>Acme Products International</title>
<body>
  <h1>Hello <bean:write name="user"/>!</h1>
  <p>These are our latest offers:
  <ul>
    <logic:iterate name="latestProducts" id="prod">

```

```

        <li><bean:write name="prod" property="name"/>
            for <bean:write name="prod" property="price"/> Credits.
        </logic:iterate>
    </ul>
</body>
</html>

```

你可以使用这个 FTL 文件（使用 `ftl` 扩展名而不是 `jsp`）：

```

<html>
<head><title>Acme Products International</title>
<body>
    <h1>Hello ${user}!</h1>
    <p>These are our latest offers:
    <ul>
        <#list latestProducts as prod>
            <li>${prod.name} for ${prod.price} Credits.
        </#list>
    </ul>
</body>
</html>

```

### 警告！

在 FreeMarker 中，`<html:form action="/query">...</html:form>` 仅仅被视为是静态文本，所以它会按照原本输出出来了，就像其他 XML 或 HTML 标记一样。JSP 标签也仅仅是 FreeMarker 的指令，没有什么特殊之处，所以你可以使用 *FreeMarker* 语法形式来调用它们，而不是 JSP 语法：`<@html.form action="/query">...</@html.form>`。注意在 FreeMarker 语法中你不能像 JSP 那样在参数中使用 `${...}`，而且不能给参数值加引号。所以这样是不对的：

```

<!-- WRONG: -->
<@my.jspTag color="${aVariable}" name="aStringLiteral"
    width="100" height=${a+b} />

```

但下面这样是正确的：

```

<!-- Good: -->
<@my.jspTag color=aVariable name="aStringLiteral"
    width=100 height=a+b />

```

在这两个模板中，当你要引用 `user` 和 `latestProduct` 时，它会首先试图去查找一个名字已经在模板中创建的变量（比如 `prod`；如果你使用 JSP：这是一个 `page` 范围内的属性）。如果那样做不行，它会尝试在对 `HttpServletRequest` 象中查找那个名字的属性，如果没有找到就在 `HttpSession` 中找，如果还没有找到那就在 `ServletContext` 中找。FTL 按这种情况工作是因为 `FreemarkerServlet` 创建数据模型由上面提到的 3 个对象中的属性而来。那也就是说，这种情况下根哈希表不是 `java.util.Map`（正如本手册中的一些例子那样），而是



`ServletContext+HttpSession+HttpServletRequest`；FreeMarker 在处理数据模型类型的时候非常灵活。所以如果你想将变量“name”放到数据模型中，那么你可以调用 `servletRequest.setAttribute("name", "Fred")`；这是模型 2 的逻辑，而 FreeMarker 将会适应它。

`FreemarkerServlet` 也会在数据模型中放置 3 个哈希表，这样你就可以直接访问 3 个对象中的属性了。这些哈希表变量是：`Request`，`Session`，`Application`（和 `ServletContext` 对应）。它还会暴露另外一个名为 `RequestParameters` 的哈希表，这个哈希表提供访问 HTTP 请求中的参数。

`FreemarkerServlet` 也有很多初始参数。它可以被设置从任意路径来加载模板，从类路径下，或相对于 Web 应用程序的目录。你可以设置模板使用的字符集。你还可以设置想使用的对象包装器等。

通过子类别，`FreemarkerServlet` 易于定制特殊需要。那就是说，你需要对所有模板添加一个额外的可用变量，使用 `servlet` 的子类，覆盖 `preTemplateProcess()` 方法，在模板被执行前，将你需要的额外数据放到模型中。或者在 `servlet` 的子类中，在 `Configuration` 中设置这些全局的变量作为共享变量。

要获取更多信息，可以阅读该类的 Java API 文档。

## 4.6.2 包含其他 Web 应用程序资源中的内容

你可以使用由 `FreemarkerServlet`（2.3.15 版本之后）提供的客户化标签 `<@include_page path="..." />` 来包含另一个 Web 应用资源的内容到输出内容中；这对于整合 JSP 页面（在同一 Web 服务器中生活在 FreeMarker 模板旁边）的输出到 FreeMarker 模板的输出中非常有用。使用：

```
<@include_page path="path/to/some.jsp"/>
```

和使用 JSP 指令是相同的：

```
<jsp:include page="path/to/some.jsp">
```

**注意：**

`<@include_page ...>` 不能和 `<#include ...>` 搞混，后者是为了包含 FreeMarker 模板而不会牵涉到 Servlet 容器。使用 `<#include ...>` 包含的模板和包含它的模板共享模板处理状态，比如数据模型和模板语言变量，而 `<@include_page ...>` 开始一个独立的 HTTP 请求处理。

**注意：**

一些 Web 应用框架为此提供它们自己的解决方案，这种情况下你就可以使用它们来替代。而一些 Web 应用框架不使用 `FreemarkerServlet`，所以 `include_page` 是没有作用的。

路径可以是相对的，也可以是绝对的。相对路径被解释成相对于当前 HTTP 请求（一个可以触发模板执行的请求）的 URL，而绝对路径在当前的 `servlet` 上下文（当前的 Web 应用）中是绝对的。你不能从当前 Web 应用的外部包含页面。注意你可以包含任意页面，而不仅仅是 JSP 页面；我们仅仅使用以 `.jsp` 结尾的页面作为说明。

除了参数 `path` 之外，你也可以用布尔值（当不指定时默认是 `true`）指定一个名为 `inherit_params` 可选的参数来指定被包含的页面对当前的请求是否可见 HTTP 请求中

的参数。

最后，你可以指定一个名为 `params` 的可选参数，来指定被包含页面可见的新请求参数。如果也传递继承的参数，那么指定参数的值将会得到前缀名称相同的继承参数的值。`params` 的值必须是一个哈希表类型，它其中的每个值可以是字符串，或者是字符串序列（如果你需要多值参数）。这里给出一个完整的示例：

```
<@include_page path="path/to/some.jsp" inherit_params=true
params={"foo": "99", "bar": ["a", "b"]} />
```

这会包含 `path/to/some.jsp` 页面，传递它的所有的当前请求的参数，除了“foo”和“bar”，这两个会被分别设置为“99”和多值序列“a”，“b”。如果原来请求中已经有这些参数的值了，那么新值会添加到原来存在的值中。那就是说，如果“foo”有值“111”和“123”，那么现在它会有“99”，“111”，“123”。

事实上使用 `params` 给参数传递非字符串值是可能的。这样的值首先会被转换为适合的 *Java* 对象（数字，布尔值，日期等），之后调用它们 *Java* 对象的 `toString()` 方法来得到字符串值。最好不要依赖这种机制，作为替代，明确参数值在模板级别不能转换成字符串类型之后，在使用到它的地方可以使用内建函数 `?string` 和 `?c`。

### 4.6.3 在 FTL 中使用 JSP 客户化标签

`FreemarkerServlet` 将一个哈希表类型的 `JspTaglibs` 放到数据模型中，就可以使用它来访问 JSP 标签库了。JSP 客户化标签库将被视为普通用户自定义指令来访问。例如，这是使用了 `Struts` 标签库的 JSP 文件：

```
<%@page contentType="text/html; charset=ISO-8859-2"
language="java"%>
<%@taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<%@taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>

<html>
  <body>
    <h1><bean:message key="welcome.title"/></h1>
    <html:errors/>
    <html:form action="/query">
      Keyword: <html:text property="keyword"/><br>
      Exclude: <html:text property="exclude"/><br>
      <html:submit value="Send"/>
    </html:form>
  </body>
</html>
```

这是一个（近似）等价的 FTL：



```

<#assign html=JspTaglibs["/WEB-INF/struts-html.tld"]>
<#assign bean=JspTaglibs["/WEB-INF/struts-bean.tld"]>

<html>
  <body>
    <h1><@bean.message key="welcome.title"/></h1>
    <@html.errors/>
    <@html.form action="/query">
      Keyword: <@html.text property="keyword"/><br>
      Exclude: <@html.text property="exclude"/><br>
      <@html.submit value="Send"/>
    </@html.form>
  </body>
</html>

```

因为 JSP 客户化标签是在 JSP 环境中来书写操作的，它们假设变量（在 JSP 中常被指代“beans”）被存储在 4 个范围中：page 范围，request 范围，session 范围和 application 范围。FTL 没有这样的表示法（4 种范围），但是 `FreemarkerServlet` 给客户化标签提供仿真的环境，这样就可以维持 JSP 范围中的“beans”和 FTL 变量之间的对应关系。对于自定义的 JSP 标签，请求，会话和应用范围是和真实 JSP 相同的：`javax.servlet.ServletContext`，`HttpSession` 和 `ServerRequest` 对象中的属性。从 FTL 的角度来看，这三种范围都在数据模型中，这点前面已经解释了。page 范围和 FTL 全局变量（参见 `global` 指令）是对应的。那也就是，如果你使用 `global` 指令创建一个变量，通过仿真的 JSP 环境，它会作为 page 范围变量对自定义标签可见。而且，如果一个 JSP 标签创建了一个新的 page 范围变量，那么结果和用 `global` 指令创建的是相同的。要注意在数据模型中的变量作为 page 范围的属性对 JSP 标签是不可见的，尽管它们在全局是可见的，因为数据模型和请求，会话，应用范围是对应的，而不是 page 范围。

在 JSP 页面中，你可以对所有属性值加引号，这和参数类型是字符串，布尔值或数字没有关系，但是因为在 FTL 模板中自定义标签可以被用户自定义 FTL 指令访问到，你将不得不在自定义标签中使用 FTL 语法规则，而不是 JSP 语法。所以当你指定一个属性的值时，那么在等号的右边是一个 FTL 表达式。因此，你不能对布尔值和数字值的参数加引号（比如：`<@tiles.insert page="/layout.ftl" flush=true/>`），否则它们将被解释为字符串值，当 `FreeMarker` 试图传递值到期望非字符串值的自定义标记中时，这就会引起类型不匹配错误。而且还要注意，这很自然，你可以使用任意 FTL 表达式作为属性的值，比如变量，计算的结果值等。（比如：`<@tiles.insert page=layoutName flush=foo && bar/>`）

`servlet` 容器运行过程中，因为它实现了自身的轻量级 JSP 运行时环境，它用到 JSP 标签库，而 `FreeMarker` 并不依赖于 JSP 支持。这是一个很小但值得注意的地方：在它们的 TLD 文件中，开启 `FreeMarker` 的 JSP 运行时环境来分发事件到 JSP 标签库中注册时间监听器，你应该将下面的内容添加到 Web 应用下的 `WEB-INF/web.xml` 文件中：

```

<listener>
<listener-class>freemarker.ext.jsp.EventForwarding</listener-
-class>
</listener>

```

注意尽管 `Servlet` 容器没有本地的 JSP 支持,你也可以在 `FreeMarker` 中使用 JSP 标签库。只是确保对 JSP 1.2 版本(或更新)的 `javax.servlet.jsp.*` 包在 Web 应用程序中可用就行。如果你的 `Servlet` 容器只对 JSP 1.1 支持,那么你不得不将下面六个类(比如你可以从 Tomcat 5.x 或 Tomcat 4.x 的 jar 包中提取)复制到 Web 应用的 `WEB-INF/classes/...` 目录下:

```
javax.servlet.jsp.tagext.IterationTag,  
javax.servlet.jsp.tagext.TryCatchFinally,  
javax.servlet.ServletContextListener,  
javax.servlet.ServletContextAttributeListener,  
javax.servlet.http.HttpSessionAttributeListener,  
javax.servlet.http.HttpSessionListener.
```

但是要注意,因为容器只支持 JSP 1.1,通常是使用较早的 `Servlet 2.3` 之前的版本,事件监听器可能就不支持,因此 JSP 1.2 标签库来注册事件监听器会正常工作。

在撰写本文档时, JSP 已经升级到 2.1 了,许多特性也已经实现了,除了 JSP 2(也就是说 JSP 自定义标记在 JSP 语言中实现了)的“标签文件”特性。标签文件需要被编译成 Java 类文件,在 `FreeMarker` 下才会有用。

#### 4.6.4 在 JSP 页面中嵌入 FTL

有一个标签库允许你将 FTL 片段放到 JSP 页面中。嵌入的 FTL 片段可以访问 JSP 的 4 种范围内的属性(Beans)。你可以在 `FreeMarker` 发布包中找到一个可用的示例和这个标签库。

### 4.7 为 FreeMarker 配置安全策略

当 `FreeMarker` 运行在装有安全管理器的 Java 虚拟机中时,你不得不再授与一些权限,确保运行良好。最值得注意的是,你需要为对 `freemarker.jar` 的安全策略文件添加这些条目:

```
grant codeBase "file:/path/to/freemarker.jar"  
{  
    permission java.util.PropertyPermission "file.encoding",  
    "read";  
    permission java.util.PropertyPermission "freemarker.*",  
    "read";  
}
```

另外,如果从一个目录中加载模板,你还需要给 `FreeMarker` 授权来从那个目录下读取文件,使用如下的授权:

```
grant codeBase "file:/path/to/freemarker.jar"  
{  
    ...  
    permission java.io.FilePermission "/path/to/templates/-",  
    "read";  
}
```

最终，如果你使用默认的模板加载机制，也就是从当前文件夹下加载模板，那么需要指定这些授权内容：（主要表达式`${user.dir}`将会在运行时被策略解释器处理，几乎它就是一个 FreeMarker 模板）

```
grant codeBase "file:/path/to/freemarker.jar"
{
    ...
    permission java.util.PropertyPermission "user.dir", "read";
    permission java.io.FilePermission "${user.dir}/-", "read";
}
```

很自然地，如果你在 Windows 下运行，使用两个反斜杠来代替一个斜杠来分隔路径中的目录间隔。

## 4.8 遗留的 XML 包装实现

### 注意：

遗留的 XML 包装已经废弃了。FreeMarker 2.3 已经引入了对新的 XML 处理模型的支持。新的 XML 包装包已经引入了，`freemarker.ext.dom`。对于新用法，我们鼓励你使用。它会在 XML 处理指南中来说明。

`freemarker.ext.xml.NodeListModel` 类提供了来包装 XML 文档展示为节点树模板模型。每个节点列表可以包含零个或多个 XML 节点（文档类型，元素类型，文本类型，处理指令，注释，实体引用，CDATA 段等）。节点列表实现了下面模板的语义模型接口：

### 4.8.1 TemplateScalarModel 模板标量模型

当使用一个标量时，节点列表将会呈现 XML 片段，表示其包含的节点。这使得使用 XML 到 XML 转换模板很方便。

### 4.8.2 TemplateCollectionModel 模板集合模型

当用 `list` 指令来使用一个集合时，它会简单枚举它的节点。每个节点将会被当作一个新的单一节点组的节点列表返回。

### 4.8.3 TemplateSequenceModel 模板序列模型

当被用作是序列时，它会返回第 *i* 个节点作为一个新的节点列表，包含单独的被请求的节点。也就是说，要返回`<book>`元素的第三个`<chapter>`元素，你可以使用下面的代码（节点索引是从零开始的）：

```
<#assign 3rdChapter = xmlDoc.book.chapter[2]>
```

## 4.8.4 TemplateHashModel 模板哈希表模型

当被用作是哈希表时，它基本上是用来遍历子节点。也就是，如果你有个名为 `book` 的节点列表，并包装了一个有很多 `chapter` 的元素节点，那么 `book.chapter` 将会产生一个 `book` 元素的所有 `chapter` 元素的节点列表。`@`符号常被用来指代属性：`book.@title` 产生一个有单独属性的节点列表，也就是 `book` 元素的 `title` 属性。

意识到下面这样的结果是很重要的，比如，如果 `book` 没有 `chapter-s`，那么 `book.chapter` 就是一个空序列，所以 `xmlDoc.book.chapter??` 就不会是 `false`，它会一直是 `true`！相似地，`xmlDoc.book.somethingTotallyNonsense??` 也不会是 `false`。为了检查是否发现子节点，可以使用 `xmlDoc.book.chapter?size == 0`。

哈希表定义了一些“魔力的键”。所有的这些键以下划线开头。最值得注意的是 `_text`，可以得到节点的文本内容：`${book.@title._text}` 将会给模板交出属性的值。相似地，`_name` 将会取得元素或属性的名字。`*`或 `_allChildren` 返回所有节点列表元素中的直接子元素，而 `@*`或 `_allAttributes` 返回节点列表中元素的所有属性。还有很多这样的键；下面给出哈希表键的详细总结：

键名	结果为
<code>*或_children</code>	所有当前节点（非递归）的直接子元素。 适用于元素和文档节点
<code>@*或_attributes</code>	当前节点的所有属性。仅适用于元素。
<code>@attributeName</code>	当前节点的命名属性。适用于元素，声明和处理指令。在声明中它支持属性 <code>publicId</code> ， <code>systemId</code> 和 <code>elementName</code> 。在处理指令中，它支持属性 <code>target</code> 和 <code>data</code> ，还有数据中以 <code>name="value"</code> 对出现的其他属性名。对于声明和处理指令的属性节点是合成的，因此它们没有父节点。要注意， <code>@*</code> 不能在声明或处理指令上进行操作。
<code>_ancestor</code>	当前节点的所有祖先，直到根元素（递归）。 适用于类型和 <code>_parent</code> 相同的节点类型。
<code>_ancestorOrSelf</code>	当前节点和它的所有祖先节点。 适用于和 <code>_parent</code> 相同的节点类型。
<code>_cname</code>	当前节点（命名空间 <code>URI</code> +本地名称）的标准名称，每个节点（非递归）一个字符串值。适用于元素和属性。
<code>_content</code>	当前节点的全部内容，包括子元素，文本，实体引用和处理指令（非递归）。适用于元素和文档。
<code>_descendant</code>	当前节点的所有递归的子孙元素。 适用于文档和元素节点。
<code>_descendantOrSelf</code>	当前节点和它的所有递归的子孙元素。 适用于文档和元素节点。
<code>_document</code>	当前节点所属的所有文档类型。 适用于所有除文本的节点。
<code>_doctype</code>	当前节点的声明。仅仅适用于文档类型节点。
<code>_filterType</code>	这是一种按类型过滤的模板方法模型。当被调用时，它会产生一个节点列表，仅仅包含它们当前节点，这些节点的类型

	和传递给它们参数的一种类型相匹配。你应该传递任意数量的字符串给这个方法，其中包含来保持类型的名字。合法的类型名称是：“attribute”，“cdata”，“comment”，“document”，“documentType”，“element”，“entity”，“entityReference”，“processingInstruction”，“text”。
<code>_name</code>	当前节点的名称。每个节点（非递归）一个字符串值。适用于元素和属性（返回它们的本地名称），实体，处理指令（返回它的目标），声明（返回它的公共 ID）。
<code>_nsprefix</code>	当前节点的命名空间前缀，每个节点（非递归）一个字符串值。 适用于元素和属性。
<code>_nsuri</code>	当前节点的命名空间 URI，每个节点（非递归）一个字符串值。 适用于元素和属性。
<code>_parent</code>	当前节点的父节点。 适用于元素，属性，注释，实体，处理指令。
<code>_qname</code>	当前节点在 <code>[namespacePrefix:]localName</code> 形式的限定名，每个节点（非递归）一个字符串值。 适用于元素和属性。
<code>_registerNamespace(prefix, uri)</code>	注册一个对当前节点列表和从当前节点列表派生出的所有节点列表有指定前缀和 URI 的 XML 命名空间。注册之后，你可以使用 <code>nodelist["prefix:localname"]</code> 或 <code>nodelist["@prefix:localname"]</code> 语法来访问元素和属性，它们的名称是命名空间范围内的。注意命名空间的前缀需要不能和当前 XML 文档它自己使用的前缀相匹配，因为命名空间纯粹是由 URI 来比较的。
<code>_text</code>	当前节点的文本内容，每个节点（非递归）一个字符串值。 适用于元素，属性，注释，处理指令（返回它的数据）和 CDATA 段。保留的 XML 字符（‘<’ 和 ‘&’）不能被转义。
<code>_type</code>	返回描述节点类型的节点列表，每个节点包含一个字符串值。可能的节点名称是：合法的节点名称是：“attribute”，“cdata”，“comment”，“document”，“documentType”，“element”，“entity”，“entityReference”，“processingInstruction”，“text”。如果节点类型是未知的，就返回“unknown”。
<code>_unique</code>	当前节点的一个拷贝，仅仅保留每个节点第一次的出现，消除重复。重复可以通过应用对树的向上遍历出现在节点列表中，如 <code>_parent</code> ， <code>_ancestor</code> ， <code>_ancestorOrSelf</code> 和 <code>_document</code> ，也就是说会返回一个节点列表，它包含 <code>foo</code> 中重复的节点，每个节点会包含出现的次数，和它子节点数目相等。这些情况下，使用 <code>foo._children._parent._unique</code> 来消除重复。 适用于所有节点类型。
其他的键	当前节点的子元素的名称和键相匹配。这允许以 <code>book.chapter.title</code> 这种风格语法进行方便的子元

	素遍历。注意在技术上 <code>nodeset.childname</code> 和 <code>nodeset("childname")</code> 相同，但是两者写法都很短，处理也很迅速。适用于文档和元素节点。
--	---

## 4.8.5 TemplateMethodModel 模板方法模型

当被用作方法模型，它返回一个节点列表，这个列表是处理节点列表中当前内容的 XPath 表达式的结果。为了使这种特性能够工作，你必须将 `Jaxen` 类库放到类路径下。比如：

```
<#assign firstChapter=xmlDoc("//chapter[first()]")>
```

## 4.8.6 命名空间处理

为了遍历有命名空间范围内名称的子元素这个目的，你可以使用节点列表注册命名空间前缀。你可以在 Java 代码中来做，调用：

```
public void registerNamespace(String prefix, String uri);
```

方法，或者在模板中使用

```
${odelist._registerNamespace(prefix, uri)}
```

语法。从那里开始，你可以在命名空间通过特定的 URI 来标记引用子元素，用这种语法

`odelist["prefix:localName"]`

和

`odelist["@prefix:localName"]`

和在 XPath 表达式中使用这些命名空间前缀一样。命名空间使用一个节点列表来注册并传播到所有节点列表，这些节点列表来自于原来的节点列表。要注意命名空间只可使用 URI 来进行匹配，所以你可以在你的模板中安全地使用命名空间的前缀，这和在现实 XML 中的不同，在模板和 XML 文档中，一个前缀只是一个对 URI 的本地别名。

## 4.9 和 Ant 一起使用 FreeMarker

我们现在知道有两种“FreeMarker Ant tasks”：

- **FreemarkerXmlTask**：它来自于 FreeMarker 的发布包，打包到 `freemarker.jar` 中。这是使用 FreeMarker 模板转换 XML 文档的轻量级的，易于使用的 Ant 任务。它的入口源文件（输入文件）是 XML 文件，和生成的输出文件对应，这是通过单独模板实现的。也就是说，对于每个 XML 文件，模板会被执行（在数据模型中的 XML 文档），模板的输出会被写入到一个和原 XML 文件名相似名称的文件中。因此，模板文件扮演了一个和 XSLT 样式表相似的角色，但它是 FTL，而不是 XSLT。
- **FMPP**：这是一个重量级的，以很少的 XML 为中心，第三方 Ant 任务（和独立的命令行工具）。它主要的目的是用作为模板文件的源文件（输入文件）生成它们自己对应的输出文件，但它也对以 XML-s 为源文件的 **FreemarkerXmlTask** 进行



支持。而且，相比于 `FreemarkerXmlTask`，它还有额外的特性。那么它的缺点是什么？它太复杂太一般化了，不容易掌握和使用。

这一部分介绍了 `FreemarkerXmlTask`，要了解 FMPP 更多的信息，可以访问它的主页：<http://fmpp.sourceforge.net/>。

为了使用 `FreemarkerXmlTask`，你必须首先在你的 `Ant` 构建文件中定义 `freemarker.ext.ant.FreemarkerXmlTask`，然后调用任务。假设你想使用假定的“`xml2html.ftl`”模板转换一些 XML 文档到 HTML，XML 文档在目录“`xml`”中而 HTML 文档生成到目录“`html`”中，你应该这样来写：

```
<taskdef name="freemarker"
  classname="freemarker.ext.ant.FreemarkerXmlTask">
  <classpath>
    <pathelement location="freemarker.jar" />
  </classpath>
</taskdef>
<mkdir dir="html" />
<freemarker basedir="xml" destdir="html" includes="**/*.xml"
  template="xml2html.ftl" />
```

这个任务将会对每个 XML 文档调用模板。每个文档将会被解析成 DOM 树，然后包装成 `FreeMarker` 节点变量。当模板开始执行时，特殊变量 `.node` 被设置成 XML 文档节点的根。

注意，如果你正使用遗留的（`FreeMarker 2.2.x` 和以前版本）XML 适配器实现，也同样可以进行，而且 XML 树的根节点被放置在数据模型中，作为变量 `document`。它包含了遗留的 `freemarker.ext.xml.NodeListModel` 类的实例。

注意所有通过构建文件定义的属性将会作为名为“`properties`”的哈希表模型来用。一些其他方法也会可用；对模板中什么样的可用变量的详细描述，还有什么样的属性可以被任务接受，参见 `freemarker.ext.ant.FreemarkerXmlTask` 的 JavaDoc 文档。

## 4.10 Jython 包装器

`freemarker.ext.jython` 包包含了启用任意 Jython 对象的模型，并被用作是 `TemplateModel`。在一个很基础的示例中，你可以使用如下调用：

```
public TemplateModel wrap(Object obj);
```

`freemarker.ext.jython.JythonWrapper` 类的方法。这个方法会包装传递的对象，包装成合适的 `TemplateModel`。下面是一个对返回对象包装器的属性的总结。为了下面的讨论，我们假设在模板模型根中，对对象 `obj` 调用 `JythonWrapper` 后模型名为 `model`。

### 4.10.1 TemplateHashModel functionality 模板哈希表模型功能

`PyDictionary` 和 `PyStringMap` 将会被包装成一个哈希表模型。键的查找映射到 `__finditem__` 方法；如果一个项没有被找到，那么就返回一个为 `None` 的模型。

## 4.10.2 TemplateScalarModel functionality 模板标量模型功能

每一个 python 对象会实现 `TemplateScalarModel` 接口，其中的 `getAsString()` 方法会委派给 `toString()` 方法。

## 4.10.3 TemplateBooleanModel functionality 模板布尔值模型功能

每一个 python 对象会实现 `TemplateBooleanModel` 接口，其中的 `getAsBoolean()` 方法会指派给 `__nonzero__()` 方法，符合 Python 语义的 true/false。

## 4.10.4 TemplateNumberModel functionality 模板数字模型功能

对 `PyInteger`，`PyLong` 和 `PyFloat` 对象的模型包装器实现了 `TemplateNumberModel` 接口，其中的 `getAsNumber()` 方法返回 `__tojava__(java.lang.Number.class)`。

## 4.10.5 TemplateSequenceModel functionality 模板序列模型功能

对所有扩展了 `PySequence` 的类的模型包装器会实现 `TemplateSequenceModel` 接口，因此它们中的元素可以通过使用 `model[i]` 语法形式的序列来访问，这会指派给 `__finditem__(i)`。你也可以使用内建函数 `model?size` 查询数组的长度或者 list 的大小，它会指派给 `__len__()`。



# 第三部分 XML 处理指南

## 前言

尽管 FreeMarker 最初被设计用作 Web 页面的模板引擎，对于 2.3 版本来说，它的另外一个应用领域目标是：转换 XML 到任意的文本输出（比如 HTML）。因此，在很多情况下，FreeMarker 也是一个可选的 XSLT。

从技术上来说，在转换 XML 文档上没有什么特别之处。它和你使用 FreeMarker 做其他事情都是一样的：你将 XML 文档丢到数据模型中（和其他可能的变量），然后你将 FTL 模板和数据模型合并来生成输出文本。对于更好的 XML 处理的额外特性是节点 FTL 变量类型（在通用的树形结构中象征一个节点，不仅仅是对 XML 有用）和用内建函数，指令处理它们，你使用的 XML 包装器会暴露 XML 文档，并将作为模板的 FTL 变量。

使用 FreeMarker 和 XSLT 有什么不同？FTL 语言有常规的命令式/过程式的逻辑。另一方面，XSLT 是声明式的语言，由很聪明的人设计出来，所以它并不能轻易吸收它的逻辑，也不会很多情况下使用。而且它的语法也非常繁琐。然而，当你处理 XML 文档时，XSLT 的“应用模板”方法可以非常方便，因此 FreeMarker 支持称作“访问者模式”的相似事情。所以在很多应用程序中，写 FTL 的样式表要比写 XSLT 的样式表容易很多。另外一个根本的不同是 FTL 转换节点树到文本，而 XSLT 转换一棵树到另一棵树。所以你不能经常在使用 XSLT 的地方使用 FreeMarker。

## 第一章 揭示 XML 文档

### 1.1 节点树

我们使用如下示例的 XML 文档：

```
<book>
  <title>Test Book</title>
  <chapter>
    <title>Ch1</title>
    <para>p1.1</para>
    <para>p1.2</para>
    <para>p1.3</para>
  </chapter>
  <chapter>
    <title>Ch2</title>
    <para>p2.1</para>
    <para>p2.2</para>
  </chapter>
</book>
```

W3C 的 DOM 定义 XML 文档模型为节点树。上面 XML 的节点树可以被视为：

```

document
|
+- element book
|
|   +- text "\n "
|   |
|   +- element title
|   | |
|   | +- text "Test Book"
|   |
|   +- text "\n "
|   |
|   +- element chapter
|   | |
|   | +- text "\n  "
|   | |
|   | +- element title
|   | | |
|   | | +- text "Ch1"
|   | |
|   | +- text "\n  "
|   | |
|   | +- element para
|   | | |
|   | | +- text "p1.1"
|   | |
|   | +- text "\n  "
|   | |
|   | +- element para
|   | | |
|   | | +- text "p1.2"
|   | |
|   | +- text "\n  "
|   | |
|   | +- element para
|   | | |
|   | | +- text "p1.3"
|   |
+- element
|
|   +- text "\n  "
|   |
|   +- element title
|   | |

```

```

|   +- text "Ch2"
|
+- text "\n   "
|
+- element para
|   |
|   +- text "p2.1"
|
+- text "\n   "
|
+- element para
|
|   +- text "p2.2"

```

要注意，烦扰的“\n”是行的中断（这里用\n指示，在 FTL 字符串中使用转义序列）和标记直接的缩进空格。

注意和 DOM 相关的术语：

- 一棵树最上面的节点称为 **root** 根，在 XML 文档中，它通常是“文档”节点，而不是最顶层元素（本例中的 **book**）。
- 如果 B 是 A 的直接后继，我们说 B 节点是 A 节点的 **child** 子节点。比如，两个 **chapter** 元素是 **book** 元素的子节点，但是 **para** 元素就不是。
- 如果 A 是 B 的直接前驱，也就是说，如果 B 是 A 的子节点，我们说节点 A 是节点 B 的 **parent** 父节点。比如，**book** 元素是两个 **chapter** 元素的父节点，但是它不是 **para** 元素的父节点。
- XML 文档中可以出现几种成分，比如元素，文本，注释，处理指令等。所有这些成分都是 DOM 树的节点，所以就有元素节点，文本节点，注释节点等。原则上，元素的属性也是树的节点——它们是元素的子节点——，但是，通常我们（还有其他 XML 相关的技术）不包含元素的子节点。所以基本上它们不被记为子节点。

程序员将 DOM 树的文档节点方法 FreeMarker 的数据模型中，那么模板开发人员可以以变量为出发点来使用 DOM 树。

FTL 中的 DOM 节点和 **node variable** 节点变量对应。这是变量类型，和字符串，数字，哈希表等类型相似。节点变量类型使得 FreeMarker 来获取一个节点的父节点和子节点成为可能。这是技术上需要允许模板开发人员在节点间操作，也就是，使用节点内建函数或者 **visit** 和 **recurse** 指令；我们会在后续章节中展示它们的使用。

## 1.2 将 XML 放到数据模型中

**注意：**

这个部分是对程序员来说的：

创建一个简单的程序来运行下面的示例是非常容易的。仅仅用下面这个例子来替换程序开发指南中快速入门示例中的“Create a data-model”部分：

```
/* Create a data-model */
Map root = new HashMap();
root.put(
    "doc",
    freemarker.ext.dom.NodeModel.parse(new
File("the/path/of/the.xml")););
```

然后你可以在基本的输出（通常是终端屏幕）中得到一个程序可以输出 XML 转换的结果。

注意：

- `parse` 方法默认移除注释和处理指令节点。参见 API 文档获取详细信息。
- `NodeModel` 也允许你直接包装 `org.w3c.dom.Node-s`。首先你也许想用静态的实用方法清空 DOM 树，比如 `NodeModel.simplify` 或你自定义的清空规则。

注意有可用的工具，你可以使用它们从 XML 文档中来生成文件，所以你不需对通用任务来写自己的代码。参加“和 Ant 一起使用 FreeMarker”部分。

## 第二章 必要的 XML 处理

### 2.1 通过例子来学习

注意：

这部分我们使用的 DOM 树和变量都是前一章做的那个。

假设程序员在数据模型中放置了一个 XML 文档，就是名为 `doc` 的变量。这个变量和 DOM 树的根节点“document”对应。真实的变量 `doc` 之后结构是非常复杂的，大约类似 DOM 树。所以为了避免钻牛角尖，我们通过例子来看看如何使用。

#### 2.1.1 通过名称来访问元素

这个 FTL 打印 `book` 的 `title`：

```
<h1>${doc.book.title}</h1>
```

输出是：

```
<h1>Test Book</h1>
```

正如你所看到的，`doc` 和 `book` 都可以当作哈希表来使用。你可以按照子变量的形式来获得它们的子节点。基本上，你用描述路径的方法来访问在 DOM 树中的目标（元素 `title`）。你也许注意到了上面有一些是假象：使用 `${doc.book.title}`，就好像我们指示 FreeMarker 打印 `title` 元素本身，但是我们应该打印它的子元素文本（看看 DOM 树）。那也可以办到，因为元素不仅仅是哈希表变量，也是字符串变量。元素节点的标量是从它的文本子节点级联中获取的字符串结果。然而，如果元素有子元素，尝试使用一个元素作为标量会引起错误。比如 `${doc.book}` 将会以错误而终止。

FTL 打印 2 个 chapter 的 title:

```
<h2>${doc.book.chapter[0].title}</h2>
<h2>${doc.book.chapter[1].title}</h2>
```

这里, `book` 有两个 `chapter` 子元素, `doc.book.chapter` 是存储两个元素节点的序列。因此, 我们可以概括上面的 FTL, 所以它以任意 `chapter` 的数量起作用:

```
<#list doc.book.chapter as ch>
  <h2>${ch.title}</h2>
</#list>
```

但是如果只有一个 `chapter` 会怎么样呢? 实际上, 当你访问一个作为哈希表子变量的元素时, 通常也可以是序列 (不仅仅是哈希表和字符串), 但如果序列只包含一个项, 那么变量也作为项目自身。所以, 回到第一个示例中, 它也会打印 `book` 的 `title`:

```
<h1>${doc.book[0].title}</h1>
```

但是你知道那里就只有一个 `book` 元素, 而且 `book` 也就只有一个 `title`, 所以你可以忽略那些 `[0]`。如果 `book` 恰好有一个 `chapter-s` (否则它就是模糊的: 它怎么知道你想要的是哪个 `chapter` 的 `title`? 所以它就会以错误而停止), `${doc.book.chapter.title}` 也可以正常进行。但是因为一个 `book` 可以有很多 `chapter`, 你不能使用这种形式。如果元素 `book` 没有子元素 `chapter`, 那么 `doc.book.chapter` 将是一个长度为零的序列, 所以用 FTL `<#list ...>` 也可以进行。

知道这样一个结果是很重要的, 比如, 如果 `book` 没有 `chapter-s`, 那么 `book.chapter` 就是一个空序列, 所以 `doc.book.chapter` 就不会是 `false`, 它就一直是 `true`! 类似地, `doc.book.somethingTotallyNonsense??` 也不会是 `false`。来检查是否有子节点, 可以使用 `doc.book.chapter[0]??` (或 `doc.book.chapter?.size == 0`)。当然你可以使用类似所有的控制处理操作符 (比如 `doc.book.author[0]!"Anonymous"`), 只是不要忘了那个 `[0]`。

#### 注意:

序列的大小是 1 的规则是方便 XML 包装的方便特性 (通过多类型的 FTL 变量来实现)。其他普通序列将不会起作用。

现在我们完成了打印每个 `chapter` 所有的 `para-s` 示例:

```
<h1>${doc.book.title}</h1>
<#list doc.book.chapter as ch>
  <h2>${ch.title}</h2>
  <#list ch.para as p>
    <p>${p}</p>
  </#list>
</#list>
```

这将打印出:

```
<h1>Test</h1>
  <h2>Ch1</h2>
    <p>p1.1
    <p>p1.2
    <p>p1.3
  <h2>Ch2</h2>
    <p>p2.1
    <p>p2.2
```

上面的 FTL 可以书写的更加漂亮：

```
<#assign book = doc.book>
<h1>${book.title}</h1>
<#list book.chapter as ch>
  <h2>${ch.title}</h2>
  <#list ch.para as p>
    <p>${p}
  </#list>
</#list>
```

最终，一个广义的子节点选择机制是：模板列出所有示例 XML 文档的 `para-s`。

```
<#list doc.book.chapter.para as p>
  <p>${p}
</#list>
```

输出将会是：

```
<p>p1.1
<p>p1.2
<p>p1.3
<p>p2.1
<p>p2.2
```

这个示例说明了哈希表变量选择序列子节点的做法（在前面示例中的序列大小为 1）。在这个具体的例子中，子变量 `chapter` 返回一个大小为 2 的序列（因为有两个 `chapter-s`），之后子变量 `para` 在那个序列中选择 `para` 所有的子节点。

这种机制的一个负面结果是类似于 `doc.somethingNonsense.otherNonsesne.totalNonsense` 这样的东西会被算作是空序列，而且你也不会得到任何错误信息。

## 2.1.2 访问属性

这个 XML 和原来的那个是相同的，除了它使用 `title` 属性，而不是元素：

```

<!-- THIS XML IS USED FOR THE "Accessing attributes" CHAPTER ONLY!
-->
<!-- Outside this chapter examples use the XML from earlier.
-->

<book title="Test">
  <chapter title="Ch1">
    <para>p1.1</para>
    <para>p1.2</para>
    <para>p1.3</para>
  </chapter>
  <chapter title="Ch2">
    <para>p2.1</para>
    <para>p2.2</para>
  </chapter>
</book>

```

一个元素的属性可以通过和元素的子元素一样的方式来访问,除了你在属性名的前面放置一个@符号:

```

<#assign book = doc.book>
<h1>${book.@title}</h1>
<#list book.chapter as ch>
  <h2>${ch.@title}</h2>
  <#list ch.param as p>
    <p>${p}
  </#list>
</#list>

```

这会打印出和前面示例相同的结果。

按照和获取子节点一样的逻辑来获得属性,所以上面的 `ch.@title` 结果就是大小为 1 的序列。如果没有 `title` 属性,那么结果就是一个大小为 0 的序列。所以要注意,这里使用内建函数也是有问题的:如果你很好奇是否 `foo` 含有属性 `bar`,那么你不得不写 `foo.@bar[0]??` 来验证。( `foo.@bar??` 是不对的,因为它总是返回 `true` )。类似地,如果你想要一个 `bar` 属性的默认值,那么你就不得不写 `foo.@bar[0]!"theDefaultValue"`。

正如子元素那样,你可以选择多节点的属性。例如,这个模板将打印所有 `chapter` 的 `title` 属性。

```

<#list doc.book.chapter.@title as t>
  ${t}
</#list>

```

## 2.1.3 探索 DOM 树

这个 FTL 将会枚举所有 `book` 元素的子节点：

```
<#list doc.book?children as c>
- ${c?node_type} <#if c?node_type =
'element'>${c?node_name}</#if>
</#list>
```

这会打印出：

```
- text
- element title
- text
- element chapter
- text
- element chapter
- text
```

`?node_type` 的意思可能没有解释清除。有一些在 DOM 树中存在的节点类型，比如 `"element"`，`"text"`，`"comment"`，`"pi"` 等。

`?node_name` 返回节点的节点名称。对于其他的节点类型，也会返回一些东西，但是它对声明的 XML 处理更有用，这会在后面章节中讨论。

如果 `book` 元素有属性，由于实际的原因它可能不会在上面的列表中出现。但是你可以获得包含元素所有属性的列表，使用变量元素的子变量 `@@`。如果你将 XML 的第一行修改为这样：

```
<book foo="Foo" bar="Bar" baaz="Baaz">
```

然后运行这个 FTL：

```
<#list doc.book.@@ as attr>
- ${attr?node_name} = ${attr}
</#list>
```

然后得到这个输出（或者其他相似的结果）：

```
- baaz = Baaz
- bar = Bar
- foo = Foo
```

要返回子节点的列表，有一个方便的子变量来仅仅列出元素的子元素：

```
<#list doc.book.* as c>
- ${c?node_name}
</#list>
```

将会打印：



```
- title
- chapter
- chapter
```

你可以使用内建函数 `parent` 来获得元素的父节点：

```
<#assign e = doc.book.chapter[0].para[0]>
<!-- Now e is the first para of the first chapter -->
${e?node_name}
${e?parent?node_name}
${e?parent?parent?node_name}
${e?parent?parent?parent?node_name}
```

这会打印出：

```
para
chapter
book
@document
```

在最后一行你访问到了 DOM 树的根节点，文档节点。它不是一个元素，这就是为什么得到了一个奇怪的名字；现在我们来处理它，文档节点没有父节点。

你可以使用内建函数 `root` 来快速返回到文档节点：

```
<#assign e = doc.book.chapter[0].para[0]>
${e?root?node_name}
${e?root.book.title}
```

这会打印：

```
@document
Test Book
```

在内建函数完整的列表中你可以用来在 DOM 树中导航，可以阅读参考文档中的内建函数-节点部分。

## 2.1.4 使用 XPath 表达式

**注意：**

XPath 表达式仅在 Jaxen（推荐使用，但是使用至少 Jaxen 1.1-beta-8 版本，不能再老了）或 Apache Xalan 库可用时有效。（Apache Xalan 库在 Sun J2SE 1.4，1.5 和 1.6（也许在后续版本）中已经包含了；不需要独立的 Xalan 的 jar 包）

如果哈希表的键使用了节点变量而不能被解释（下一部分对此精确定义），那么它就会被当作 XPath 表达式被解释。要获得 XPath 的更多信息，可以访问 <http://www.w3.org/TR/xpath>。

例如，这里我们列出标题为“Ch1”的 chapter 的 `para` 元素：

```
<#list doc["book/chapter[title='Ch1']/para"] as p>
  <p>${p}
</#list>
```

它会打印：

```
<p>p1.1
<p>p1.2
<p>p1.3
```

长度为 1（在前面部分解释过了）的序列的规则也代表 XPath 的结果。也就是说，如果结果序列仅仅包含 1 个节点，它也会当作节点自身。例如，打印 chapter 元素“Ch1”第一段：

```
${doc["book/chapter[title='Ch1']/para[1]" ]}
```

这也会打印相同内容：

```
${doc["book/chapter[title='Ch1']/para[1]" ][0]}
```

XPath 表达式的内容节点（或者是节点序列）是哈希表子变量被用作发布 XPath 表达式的节点。因此，这将打印和上面例子相同的内容：

```
${doc.book["chapter[title='Ch1']/para[1]" ]}
```

注意现在你可以使用 0 序列或多（比 1 多）节点作为内容，这只在程序员已经建立 FreeMarker 使用 Jaxen 而不是 Xalan 时才可以。

也要注意 XPath 序列的项索引从 1 开始，而 FTL 的序列项索引是用 0 开始的。因此，要选择第一个 chapter，XPath 表达式是 `"/book/chapter[1]"`，而 TL 表达式是 `book.chapter[0]`。

如果程序员建立使用 Jaxen 而不是 Xalan，那么 FreeMarker 的变量在使用 XPath 变量引用时是可见的：

```
<#assign currentTitle = "Ch1">
<#list doc["book/chapter[title=$currentTitle]/para"] as p>
  ...
```

注意 `$currentTitle` 不是 FreeMarker 的插值，因为那里没有 `{` 和 `}`。那是 XPath 表达式。

一些 XPath 表达式的结果不是节点集，而是字符串，数字或者布尔值。对于那些 XPath 表达式，结果分别是 FTL 字符串，数字或布尔值变量。例如，下面的例子将会计算 XML 文档中 `para` 元素的总数，所以结果是一个数字：

```
${x["count(//para)" ]}
```

输出是：

```
5
```

## 2.1.5 XML 命名空间

默认来说，当你编写如 `doc.book` 这样的东西时，那么它会选择属于任何 XML 命名空间（和 XPath 相似）名字为 `book` 的元素。如果你想在 XML 命名空间中选择一个元素，你必须注册一个前缀，然后使用它。比如，如果元素 `book` 是命名空间 `http://example.com/ebook`，那么你不得不关联一个前缀，要在模板的顶部使用 `ftl` 指令的 `ns_prefixes` 参数：

```
<#ftl ns_prefixes={"e":"http://example.com/ebook"}>
```

现在你可以编写如 `doc["e:book"]` 的表达式。（因为冒号会混淆 FreeMarker，方括号语法的使用是需要的）

`ns_prefixes` 的值作为哈希表，你可以注册多个前缀：

```
<#ftl ns_prefixes={
    "e":"http://example.com/ebook",
    "f":"http://example.com/form",
    "vg":"http://example.com/vectorGraphics"}
>
```

`ns_prefixes` 参数影响整个 FTL 命名空间。这就意味着实际中，你在主页面模板中注册的前缀必须在所有的 `<#include ...>-d` 模板中可见，而不是 `<#imported ...>-d` 模板（经常用来引用 FTL 库）。从另外一种观点来说，一个 FTL 库可以注册 XML 命名空间前缀来为自己使用，而前缀注册不会干扰主模板和其他库。

要注意，如果一个输入模板是给定 XML 命名空间域中的，为了方便你可以设置它为默认命名空间。这就意味着如果你不使用前缀，如在 `doc.book` 中，那么它会选择属于默认命名空间的元素。这个默认命名空间的设置使用保留前缀 `D`，例如：

```
<#ftl ns_prefixes={"D":"http://example.com/ebook"}>
```

现在表达式 `doc.book` 选择属于 XML 命名空间 `http://example.com/ebook` 的 `book` 元素。不幸的是，XPath 不支持默认命名空间。因此，在 XPath 表达式中，没有前缀的元素名称通常选择不输入任何 XML 命名空间的元素。然而，在默认命名空间中访问元素你可以直接使用前缀 `D`，比如：`doc["D:book/D:chapter[title='Ch1']"]`。

注意当你使用默认命名空间时，那么你可以使用保留前缀 `N` 来选择不属于任意节点空间的元素。比如 `doc.book["N:foo"]`。这对 XPath 表达式不起作用，上述的都可以写作 `doc["D:book/foo"]`。

## 2.1.6 不用忘了转义！

我们在所有的示例中都犯了大错。我们生成 HTML 格式的输出，HTML 格式保留如 `<`，`&` 等的字符。所以当我们打印普通文本（比如标题和段落）时，我们不得不转义它，因此，示例的正确版本是：

```

<#escape x as x?html>
<#assign book = doc.book>
<h1>${book.title}</h1>
<#list book.chapter as ch>
  <h2>${ch.title}</h2>
  <#list ch.para as p>
    <p>${p}
  </#list>
</#list>
</#escape>

```

所以如果 book 的标题是 “Romeo & Julia”，那么 HTML 输出的结果就是正确的：

```

...
<h1>Romeo & Julia</h1>
...

```

## 2.2 形式化描述

每一个和 DOM 树中单独节点对应的变量都是节点和哈希表类型的多类型的变量（对于程序员：实现 `TemplateNodeModel` 和 `TemplateHashModel` 两个接口）。因此，你可以使用和节点有关的内建函数来处理它们。哈希表的键被解释为 XPath 表达式，除了在下面表格中显示的特殊键。一些节点变量也有字符串类型，所以你可以使用它们作为字符串变量（对于程序员：他们需要实现 `TemplateScalarModel` 接口）。

节点类型 ( <code>?node_type</code> )	节点名称 ( <code>?node_name</code> )	字符串值 (比如 <code>&lt;p&gt;\${node}&lt;/p&gt;</code> )	特殊的哈希表的键
"document"	"@document"	没有字符串值。 (当你尝试用字符串来使用时发生错误。)	"elementName", "prefix:elementName", "***", "@@markup", "@@nested_markup", "@@text"
"element"	"name": 元素的名称。这是本地命名空间前缀的名字。)	如果它没有子元素，所有子节点的文本串联起来。当你尝试用它当字符串时会发生错误。	"elementName", "prefix:elementName", "***", "@attrName", "@prefix:attrName", "@@", "@*", "@@start_tag", "@@end_tag", "@@attributes_markup", "@@markup", "@@nested_markup",

			"@@text", "@@qname"
"text"	"@text"	文本本身。	"@@markup", "@@nested_markup", "@@text"
"pi"	"@pi\$target"	在目标名称和?>之间的部分	"@@markup", "@@nested_markup", "@@text"
"comment"	"@comment"	注释的文本, 不包括分隔符 <!-- 和 -->。	"@@markup", "@@nested_markup", "@@text"
"attribute"	"name": 属性的名字。这是本地命名(也就是没有命名空间前缀的名字。)	属性的值。	"@@markup", "@@nested_markup", "@@text", "@@qname"
"document_type"	"@document_type\$name": name 是文档元素的名字。	没有字符串值。(当你尝试用字符串来使用时发生错误。)	"@@markup", "@@nested_markup", "@@text"

注意:

- 没有 CDATA 类型, CDATA 节点被透明地认为是文本节点。
- 变量不支持?keys 和?values。
- 元素和属性节点的名字是本地命名, 也就是, 它们不包含命名空间前缀。节点所属的命名空间的 URI 可以使用内建函数?node\_namespace 来获得。
- XPath 表达式需要 Jaxen (推荐使用, 但是请使用 1.1-beta-8 之后的版本; 可以从 <http://jaxen.org/> 下载。) 或者 Apache 的 Xalan 库, 否则会有错误并且终止模板的执行。要注意, 隐藏在 XPath 表达式中一些特殊哈希表键有相同的意思, 尽管没有 XPath 可用的实现, 但那些 XPath 表达式仍然会起作用。如果 Xalan 和 Jaxen 两者都可用, FreeMarker 将会使用 Xalan, 除非你在 Java 代码中通过调用 `freemarker.ext.dom.NodeModel.useJaxenXPathSupport()` 方法, 才会使用 Jaxen。
- 如果 Jaxen 被用来支持 XPath (而不是 Xalan), 那么 FreeMarker 变量通过 XPath 变量的引用是可见的 (比如 `doc["book/chapter[title=$currentTitle]"]`)。

特殊哈希表键的含义:

- "`elementName`", "`prefix:elementName`": 返回元素名为 `elementName` 的子节点的序列。(注意这里的子节点就是直接后继)选择是 XML 命名空间的标识, 除非 XML 文档用不再命名空间标识节点中的 XML 解析器解析。在 XML 命名空间标识节点中, 不含前缀的名字(元素名)仅仅选择不属于任何 XML 命名空间的元素(除非你已经注册了默认的 XML 命名空间), 有前缀的名字(前缀: 元素名)选择属于前缀代表命名空间的元素。前缀的注册和默认 XML 命名空间的设置是由 `ftl` 指令的 `ns_prefixes` 参数完成的。
- "`*`": 返回所有子元素(直接后继)节点的序列。这个序列会按“文档顺序”包含元素, 也就是说, 按照每个 XML 节点的表现形式的第一个字符的顺序出现(在

一般实体的扩展之后)。

- `"**"`: 返回所有后继节点的序列。这个序列按文档顺序包含元素。
- `"@attName"`, `"@prefix:attrName"`: 作为一个大小为 1, 包含属性节点的序列的形式, 返回元素的属性名 `attName`, 如果属性不存在时, 作为一个空序列返回 (所以来检查属性是否存在, 可以使用 `foo.@attName[0]??`, 而不是 `foo.@attName??`)。正如 `"elementName"` 这个特殊的键, 如果序列的长度为 1, 那么它也当作是第一个子变量。如果没有使用 `prefix`, 那么它只返回属性, 而不使用 XML 命名空间 (尽管你已经设置了以恶搞默认的 XML 命名空间)。如果使用了 `prefix`, 它返回仅仅属于关联的 `prefix` 的 XML 命名空间的属性。前缀的注册是由 `ftl` 指令的 `ns_prefixes` 参数完成的。
- `"@"` 或 `"@*"`: 返回属于父节点的节点的属性序列, 这和 XPath 中的 `@*` 是相同。
- `"@qname"`: 返回元素的完全限定名 (比如 `e:book`, 和由 `?node_name` 返回的本地名 `book` 形成对比)。使用的前缀 (如 `e`) 是基于在当前命名空间用 `ftl` 指令的 `ns_prefixes` 参数注册的前缀而选择的, 而不受源 XML 文档中使用的前缀影响。如果你已经设置了一个默认的 XML 命名空间, 那么节点会使用默认的, 前缀 `D` 就会被使用了。不属于任何 XML 命名空间的节点, 不使用前缀 (尽管你设置过默认的命名空间)。如果节点没有为命名空间注册的前缀, 那结果是不存在的变量 (`node.@qname??` 是 `false`)。
- `"@markup"`: 这会以字符串形式返回一个节点的完整 XML 标记。(完整的 XML 标记表示它也包含子节点的标记, 而且包含子节点的子节点的标记, 以此类推) 你得到的标记和在源 XML 文档中的标记相比不是必须的, 它只是语义上的相同。特别地, 注意 `CDATA` 段将会解释成普通文本。也要注意基于你是怎么用包装原生的 XML 文档的, 注释或处理指令节点可能被移除, 而且它们将会从源文件的输出中消失。对于在输出 XML 段的每个 XML 命名空间, 第一个被输出的开始标记将会包含 `xmlns:prefix` 属性, 而且那些前缀将会在输出的元素和属性名中使用。这些前缀和使用 `ftl` 指令的 `ns_prefixes` 参数注册的前缀相同 (没有前缀使用 `D`, 因为它会和 `xmlns` 属性被用来注册默认的命名空间), 如果 XML 命名空间没有定义前缀, 那么会使用一个任意未被选择使用的前缀。
- `"@nested_markup"`: 这个和 `"@markup"` 相似, 但是它返回不包括开放和封闭标记元素的 XML 标记。对于文档节点, 它返回和 `"@markup"` 相同的内容。对于其他类型节点 (文本, 处理指令等), 它返回一个空字符串。不像 `"@markup"`, 在输出中没有 `xmlns:prefix` 属性, 但是关于在元素和属性名中使用前缀规则是相同的。
- `"@text"`: 它返回文本节点 (所有后继文本节点, 而不是直接子节点) 的值, 连接成一个单独的字符串。如果节点没有子文本节点, 那么返回的是空字符串。
- `"@start_tag"`: 返回元素节点开始标记的标记。正如 `@markup`, 输出和源 XML 文档相比不是必须的, 但是在语义上是相同的。关于 XML 命名空间 (输出中的 `xmlns:prefix` 属性等), 规则和 `@markup` 是相同的。
- `"@end_tag"`: 返回元素节点结束标记的标记, 正如 `@markup`, 输出和源 XML 文档相比不是必须的, 但是在语义上是相同的。
- `@attributes_markup`: 返回元素节点属性的标记, 正如 `@markup`, 输出和源 XML 文档相比不是必须的, 但是在语义上是相同的。



## 2.2.1 节点序列

许多特殊哈希表的键（指的示上面所有的列表），和 XPath 表达式的结果是节点集（参考 XPath 介绍 <http://www.w3.org/TR/xpath/>），返回节点的序列。

这些节点序列，如果它们只存储一个子变量，也会当作子变量本身。例如，如果 `book` 元素只有一个 `title` 子节点，`${book.title[0]}` 和 `${book.title}` 是相同的。

返回一个空节点序列是普通的情形。例如，如果一个确定的 XML 文档，`book` 元素没有子元素 `chapter`，那么 `book.chapter` 的结果就是一个空的节点序列。要小心！这也意味着，`book.chapter`（注意类型）也会返回空的节点序列，而且会抛出错误而停止执行。同时，`book.chapter??`（注意类型）将会返回 `true`，因为空的序列存在，所以你不得不使用 `book.chapter[0]??` 来检查。

节点序列存储的不是 1 个节点（但是 0 或者比 1 多的节点）也会支持上面所描述的哈希表的键。那就是，下面这些特殊的键都是支持的：

- `"elementName", "prefix:elementName"`
- `"@attrName", "@prefix:attrName"`
- `"@markup", "@nested_markup"`
- `"@text"`
- `"*", "**"`
- `"@" , "@*"`

当你在一个包含多于 1 个节点或 0 个节点序列里应用上面这些特殊的键中之一的时候，那么对于序列（就是特殊的键可以起作用，比如文本节点对于键 `*` 或 `@foo` 将会被略过）中的每个节点，这些特殊的键将会被应用于单独的节点，结果将会从最终的结果中被连接。结果会被以和节点在序列中出现的对应顺序来连接。连接就意味着字符串或序列的连接是基于结果类型之上的。如果特殊的键会得到单独节点的字符串结果，那么对于多个节点的结果就是一个单独的字符串（对单独节点的结果进行连接），而如果特殊的键返回单独节点的序列，那么对于多个节点，结果就是一个单独的序列。如果你应用特殊键的序列中没有节点，那么字符串结果就是空串，而序列结果就是空序列。

XPath 表达式可以和节点序列一同使用。然而，对于 0 或者大于 1 的节点，因为 Xalan 对 XPath 实现的限制，仅仅你使用 Jaxen 代替 Xalan 时它才会起作用。

## 第三章 声明的 XML 处理

### 3.1 基础内容

**注意：**

这部分使用的 DOM 树和变量是之前章节中做好的。

因为 XML 处理的方法非常必要——这在前面章节中已经展示了一你编写一个 FTL 程序来遍历树，为了找到不同种类的节点。而使用声明的方法，你宁愿定义如何控制不同种类的节点，之后让 FreeMarker 遍历那个树，调用你定义的处理程序。这个方法对于复杂的 XML 模式非常有用，相同元素可以作为其他元素的子元素出现。这样的模式的示例就是 XHTML 和 XDocBook。

你最经常使用来处理声明方式的指令就是 `recurse` 指令，这个指令获取节点变量，并把它作为是参数，从第一个子元素开始，一个接一个地“访问”所有它的子元素。“访问”

一个节点意味着它调用了用户自定义的指令(比如宏),它的名字和子节点(`?node_name`)的名字相同。我们这么说,用户自定义指令操作节点。使用用户自定义指令处理的节点作为特殊变量`.node`是可用的。例如,这个 FTL:

```
<#recurse doc>

<#macro book>
  I'm the book element handler, and the title is: ${.node.title}
</#macro>
```

会打印(这里已经移除了输出内容中一些烦扰的空白):

```
I'm the book element handler, and the title is: Test Book
```

如果你调用 `recurse` 而不用参数,那么它使用`.node`,也就是说,它访问现在处理这个节点的所有子节点。所以这个 FTL:

```
<#recurse doc>

<#macro book>
  Book element with title ${.node.title}
  <#recurse>
  End book
</#macro>

<#macro title>
  Title element
</#macro>

<#macro chapter>
  Chapter element with title: ${.node.title}
</#macro>
```

会打印(这里已经移除了输出内容中一些烦扰的空白):

```
Book element with title Test Book
Title element
Chapter element with title: Ch1
Chapter element with title: Ch2
End book
```

你已经看到了如何来为元素节点定义处理器,但不是为文本节点定义处理器。因为处理器的名字是和它处理的节点名字相同的,作为所有文本节点的节点名字是`@text`(参考格式化描述的表格中内容),你为文本节点定义处理器,可以是这样的:

```
<#macro @text>${.node?html}</#macro>
```

注意`?html`。你不得不转义 HTML 文本,因为你生成的是 HTML 格式的输出。这个模板就是转换 XML 到完整的 HTML:



```

<#recurse doc>
<#macro book>
  <html>
    <head>
      <title><#recurse .node.title></title>
    </head>
    <body>
      <h1><#recurse .node.title></h1>
      <#recurse>
    </body>
  </html>
</#macro>
<#macro chapter>
  <h2><#recurse .node.title></h2>
  <#recurse>
</#macro>
<#macro para>
  <p><#recurse>
</#macro>
<#macro title>
  <#--
    We have handled this element imperatively,
    so we do nothing here.
  -->
</#macro>
<#macro @text>${.node?html}</#macro>

```

那么输出是（这里已经移除了输出内容中一些烦扰的空白）：

```

<html>
  <head>
    <title>Test Book</title>
  </head>
  <body>
    <h1>Test Book</h1>
    <h2>Ch1</h2>
    <p>p1.1
    <p>p1.2
    <p>p1.3
    <h2>Ch2</h2>
    <p>p2.1
    <p>p2.2
  </body>
</html>

```

注意你可以在输出中使用 `trim` 指令，如`<#t>`来大幅减少多余的空白。参考：模板开发指南/杂项/空白处理。

你也许会说 `FTL` 处理它的这些必要方法可以更短些。那是对的，但是示例 `XML` 使用了非常简单的模式，正如之前说过的，声明方法带和 `XML` 模式一同来了它的格式，而这个模式关于这里可以出现什么元素是不固定的。所以，介绍元素 `mark`，应该把文本标记为红色，而和你在哪儿使用它无关；在 `title` 或在 `para` 中。对于这点，使用声明的方法，你可以增加一个宏：

```
<#macro mark><font color=red><#recurse></font></#macro>
```

那么`<mark>...</mark>`将会自动起作用。所以对于命令式的 `XML` 模式，声明的 `XML` 处理确实将会很短，而且更重要的是，对于必要的 `XML` 处理，`FTL-s` 会更清晰。但这都依赖于你的决定，什么时候使用哪种方法；不要忘记你可以自由混合两种方法。也就是说，在一个元素处理器中，你可以使用命令式的方法来处理元素的内容。

## 3.2 详细内容

### 3.2.1 默认处理器

对于一些 `XML` 节点类型，有默认的处理程序，这会处理你不给这些节点定义处理器的节点（也就是说，如果没有可用的，和节点名称相同的用户自定义指令）。这里的节点类型，默认的处理程序会做：

- 文本节点：打印其中的文本。要注意，在很多应用程序中，这对你来说并不好，因为你应该在你发送它们到输出（使用`?html` 或`?xml` 或`?rtf` 等，这基于输出的格式）前转义这些文本。
- 处理指令节点：如果你定义了自定义指令，可以通过调用处理器调用`@pi`，否则将什么都不做（忽略这些节点）。
- 注释节点，文档类型节点：什么都不做（忽略这些节点）。
- 文档节点：调用 `recurse`，也就是说，访问文档节点的所有子节点。

元素和属性节点通常将会被 `XML` 独立机制处理。也就是，`@node_type` 将会被调用作为处理器，如果它没有被定义，那么错误会阻止模板的处理。

元素节点的情形，这意味着如果你定义了一个称为`@element` 的宏（或其他种类的用户自定义指令），没有其他特定的处理器时，那么它会捕捉所有元素节点。如果你没有`@element` 处理器，那么你必须为所有可能的元素定义处理器。

属性节点在 `recurse` 指令中不可见，所以不需要为它们编写处理器。

### 3.2.2 访问单独节点

使用 `visit` 指令你可以访问单独的节点，而不是节点的子节点：`<#visit nodeToVisit>`。有时这会很有用。

### 3.2.3 XML 命名空间

我们说过对于一个元素的处理器，用户自定义指令（比如宏）的名字就是元素的名字。事实上，它是元素的完全限定名：`prefix:elementName`。这个关于 `prefix-es` 的使用规则和命令式处理是相同的。因此，用户自定义指令 `book` 仅仅处理不属于任何 XML 命名空间（除非你已经定义了默认的 XML 命名空间）的 `book` 元素。所以示例 XML 将会使用 XML 命名空间 `http://example.com/ebook`：

```
<book xmlns="http://example.com/ebook">
...
```

那么 FTL 就会像这样：

```
<#ftl ns_prefixes={"e":"http://example.com/ebook"}>
<#recurse doc>
<#macro "e:book">
  <html>
    <head>
      <title><#recurse .node["e:title"]></title>
    </head>
    <body>
      <h1><#recurse .node["e:title"]></h1>
      <#recurse>
    </body>
  </html>
</#macro>
<#macro "e:chapter">
  <h2><#recurse .node["e:title"]></h2>
  <#recurse>
</#macro>
<#macro "e:para">
  <p><#recurse>
</#macro>
<#macro "e:title">
  <!--
    We have handled this element imperatively,
    so we do nothing here.
  -->
</#macro>
<#macro @text>${.node?html}</#macro>
```

或者你可以定义一个默认的 XML 命名空间，那后面部分的模板保持和源 XML 命名空间相同，比如：

```
<#ftl ns_prefixes={"D":"http://example.com/ebook"}>

<#recurse doc>

<#macro book>
...

```

但是这种情形下不要忘了在 XPath 表达式（我们在默认中没有使用）中，默认的 XML 命名空间必须通过明确的 **D:** 来访问，因为在 XPath 中没有前缀的名称通常指代没有 XML 命名空间的节点。而且注意到命令式的 XML 处理也是相同的逻辑，如果（当且仅当）没有默认 XML 命名空间时，元素处理器的名字没有 XML 命名空间是 **N:elementName**。然而，对于不是元素类型的节点（比如文本节点），你不能在处理器名称中使用前缀 **N**，因为这些节点在 XML 命名空间中是没有这些概念的。所以对于示例，文本节点的处理器通常就是 **@text**。

对于更详细的内容，请阅读参考文档中的 **recurse** 和 **visit** 指令。

# 第四部分 参考文档

章节内容如下：

内建函数参考文档

指令参考文档

特殊变量参考文档

FTL 中的保留名称

废弃的 FTL 结构

## 第一章 内建函数参考文档

### 1.1 处理字符串的内建函数

#### 1.1.1 substring 取子串

**注意：**

这个内建函数从 FreeMarker 2.3.7 开始可用。

概要：`exp?substring(from, toExclusive)`，也可以作为 `exp?substring(from)` 调用

一个字符串的子串。*from* 是第一个字符开始的索引。它必须是一个数字而且至少是 0，而且要小于或等于 *toExclusive*，否则错误就会中断模板的处理。*toExclusive* 是子串中最后一个字符之后的位置索引，换句话说，它比最后一个字符的索引大 1。它必须是数字，至少是 0，要小于或等于字符串的长度，否则错误就会中止模板的处理。如果 *toExclusive* 被忽略了，那么它默认就是字符串的长度。如果参数不是整型的数字，那么数值中只有整型的部分会被使用。

例如：

```
- ${'abc'?substring(0)}  
- ${'abc'?substring(1)}  
- ${'abc'?substring(2)}  
- ${'abc'?substring(3)}  
  
- ${'abc'?substring(0, 0)}  
- ${'abc'?substring(0, 1)}  
- ${'abc'?substring(0, 2)}  
- ${'abc'?substring(0, 3)}  
  
- ${'abc'?substring(0, 1)}  
- ${'abc'?substring(1, 2)}  
- ${'abc'?substring(2, 3)}
```

输出为:

```
- abc
- bc
- c
-

-
- a
- ab
- abc

- a
- b
- c
```

### 1.1.2 cap\_first 首字母大写

字符串中的第一个单词的首字母大写。更精确的“单词”的意思可以查看内建函数 `word_list`。比如:

```
${" green mouse"?cap_first}
${"GreEN mouse"?cap_first}
${"- green mouse"?cap_first}
```

输出为:

```
Green mouse
GreEN mouse
- green mouse
```

在 `"- green mouse"` 的情形下, 第一个单词是 `-`。

### 1.1.3 uncap\_first 首字母小写

这和 `cap_first` 是相反的。字符串第一个单词的首字母小写。

### 1.1.4 capitalize 首字母大写

字符串的所有单词都大写。更精确的“单词”的意思可以查看内建函数 `word_list`。比如:

```
${" green mouse"?capitalize}
${"GreEN mouse"?capitalize}
```

输出为:

```
Green Mouse  
Green Mouse
```

### 1.1.5 chop\_linebreak 切断换行符

如果在末尾没有换行符的字符串，那么可以换行，否则不改变字符串。

### 1.1.6 date, time, datetime 日期，时间，时间日期

字符串转换成日期值。建议指定一个确定格式个参数。比如：

```
<#assign test1 = "10/25/1995"?date("MM/dd/yyyy")>  
<#assign test2 = "15:05:30"?time("HH:mm:ss")>  
<#assign test3 = "1995-10-25 03:05 PM"?datetime("yyyy-MM-dd  
hh:mm a")>  
${test1}  
${test2}  
${test3}
```

将会打印出（基于当地（语言）和其他设置决定输出）如下内容：

```
Oct 25, 1995  
3:05:30 PM  
Oct 25, 1995 3:05:00 PM
```

要注意日期根据 `date_format`，`time_format` 和 `datetime_format` 的设置转换回字符串（对于日期转换成字符的更多内容请阅读：日期内建函数，日期插值）。这和你转换字符串到日期类型时使用什么格式没有关系。

如果你了解模板处理时默认日期/时间/时间日期格式，你可以不用格式化参数：

```
<#assign test1 = "Oct 25, 1995"?date>  
<#assign test2 = "3:05:30 PM"?time>  
<#assign test3 = "Oct 25, 1995 03:05:00 PM"?datetime>  
${test1}  
${test2}  
${test3}
```

如果字符串不在适当的格式，当你尝试使用这些内建函数时，错误将会中断模板执行。

### 1.1.7 ends\_with 以...结尾

返回是否这个字符串以指定的子串结尾。比如 `"redhead"?ends_with("head")` 返回布尔值 `true`。而且 `"head"?ends_with("head")` 也返回 `true`。

## 1.1.8 html HTML 格式的转义文本

字符串按照 HTML 标记输出。也就是说，下面字符串将会被替代：

- `<`用`&lt` 替换；
- `>`用`&gt` 替换；
- `&`用`&amp;` 替换；
- `"`用`&quot` 替换；

注意如果你想安全地插入一个属性，你必须在 HTML 模板中使用引号标记（是`"`，而不是`'`）为属性值加引号：

```
<input type=text name=user value="{user?html}">
```

注意在 HTML 页面中，通常你想对所有插值使用这个内建函数。所以你可以使用 `escape` 指令来节约很多输入，减少偶然错误的机会。

## 1.1.9 group 分组

这个函数只和内建函数 `matches` 的结果使用。请参考 `matches` 函数。

## 1.1.10 index\_of 索引所在位置

返回第一次字符串中出现子串时的索引位置。例如`"abcabc"?index_of("bc")`将会返回 1（不要忘了第一个字符的索引是 0）。而且，你可以指定开始搜索的索引位置：将`"abcabc"?index_of("bc", 2)`会返回 4。这对第二个参数的数值没有限制：如果它是负数，那就和是 0 是 `ige` 效果了，如果它比字符串的长度还大，那么就和它是字符串长度那个数值是一个效果。小数会被切成整数。

如果第一个参数作为子串没有在该字符串中出现时（如果你使用了第二个参数，那么就从给定的序列开始。），那么就返回-1。

## 1.1.11 j\_string Java 语言规则的字符串转义

根据 Java 语言字符串转义规则来转义字符串，所以它很安全的将值插入到字符串类型中。此外，所有 UCS 编码下的字符在 Java 语言中并没有专门的转义序列将指向 `0x20`，会被用 UNICODE 转义替换（`\uXXXX`）。

例如：

```
<#assign beanName = 'The "foo" bean.'>
String BEAN_NAME = "${beanName?j_string}";
```

将会打印：

```
String BEAN_NAME = "The \"foo\" bean.";
```



### 1.1.12 js\_string JavaScript 语言规则的字符串转义

根据 JavaScript 语言字符串转义规则来转义字符串，所以它很安全的将值插入到字符串类型中。引号 (") 和单引号 (') 要被转义。从 FreeMarker 2.3.1 开始，也要将 > 转义为 \> (为了避免 </script>)。另外，所有 UCS 编码下的字符在 Java 语言中并没有专门的转义序列将指向 0x20，会被用十六进制转义替换 (\xXX)。(当然，根据 JavaScript 语言字符串语法，反斜杠 (\) 也要被转义，换行符要被转义为 \n 等)

例如：

```
<#assign user = "Big Joe's \"right hand\"">
<script>
  alert("Welcome ${user?js_string}!");
</script>
```

将会打印：

```
<script>
  alert("Welcome Big Joe\'s \"right hand\"!");
</script>
```

### 1.1.13 last\_index\_of 最后的索引所在位置

返回最后一次(最右边)字符串中出现子串时的索引位置。它返回子串第一个(最左边)字符所在位置的索引。例如 "abcabc"?last\_index\_of("ab")：将会返回 3。而且你可以指定开始搜索的索引。例如："abcabc"?last\_index\_of("ab", 2)，将会返回 0。要注意第二个参数暗示了子串开始的最大索引。对第二个参数的数值没有限制：如果它是负数，那么效果和是零的一样，如果它比字符串的长度还大，那么就和它是字符串长度那个数值是一个效果。小数会被切成整数。

如果第一个参数作为子串没有在该字符串中出现时(如果你使用了第二个参数，那么就从给定的序列开始。)，那么就返回-1。

### 1.1.14 length 字符串长度

字符串中字符的数量

### 1.1.15 lower\_case 小写形式

字符串的小写形式。比如 "GrEeN MoUsE?lower\_case" 将会是 "green mouse"。

## 1.1.16 left\_pad 距左边

### 注意：

这个内建函数从 FreeMarker 2.3.1 版本开始可用。在 2.3 版本中是没有的。

如果它仅仅用 1 个参数，那么它将在字符串的开始插入空白，直到整个串的长度达到参数指定的值。如果字符串的长度达到指定数值或者比指定的长度还长，那么就什么都不做了。比如这样：

```
[${""?left_pad(5)}]
[${"a"?left_pad(5)}]
[${"ab"?left_pad(5)}]
[${"abc"?left_pad(5)}]
[${"abcd"?left_pad(5)}]
[${"abcde"?left_pad(5)}]
[${"abcdef"?left_pad(5)}]
[${"abcdefg"?left_pad(5)}]
[${"abcdefgh"?left_pad(5)}]
```

将会打印：

```
[    ]
[   a]
[  ab]
[ abc]
[abcd]
[abcde]
[abcdef]
[abcdefg]
[abcdefgh]
```

如果使用了两个参数，那么第一个参数表示的含义和你使用一个参数时的相同，第二个参数指定用什么东西来代替空白字符。比如：

```
[${""?left_pad(5, "-")}]
[${"a"?left_pad(5, "-")}]
[${"ab"?left_pad(5, "-")}]
[${"abc"?left_pad(5, "-")}]
[${"abcd"?left_pad(5, "-")}]
[${"abcde"?left_pad(5, "-")}]
```

将会打印：

```
[-----]
[----a]
[---ab]
[--abc]
[-abcd]
[abcde]
```

第二个参数也可以是个长度比 1 大的字符串。那么这个字符串会周期性的插入，比如：

```
[${""?left_pad(8, ".oO")}]  
[${"a"?left_pad(8, ".oO")}]  
[${"ab"?left_pad(8, ".oO")}]  
[${"abc"?left_pad(8, ".oO")}]  
[${"abcd"?left_pad(8, ".oO")}]
```

将会打印：

```
[.oO.oO.o]  
[.oO.oO.a]  
[.oO.oOab]  
[.oO.oabc]  
[.oO.abcd]
```

第二个参数必须是个字符串值，而且至少有一个字符。

### 1.1.17 right\_pad 距右边

**注意：**

这个内建函数从 FreeMarker 2.3.1 版本开始可用。在 2.3 版本中是没有的。

这个和 `left_pad` 相同，但是它从末尾开始插入字符而不是从开头。

比如：

```
[${""?right_pad(5)}]  
[${"a"?right_pad(5)}]  
[${"ab"?right_pad(5)}]  
[${"abc"?right_pad(5)}]  
[${"abcd"?right_pad(5)}]  
[${"abcde"?right_pad(5)}]  
[${"abcdef"?right_pad(5)}]  
[${"abcdefg"?right_pad(5)}]  
[${"abcdefgh"?right_pad(5)}]  
  
[${""?right_pad(8, ".oO")}]  
[${"a"?right_pad(8, ".oO")}]  
[${"ab"?right_pad(8, ".oO")}]  
[${"abc"?right_pad(8, ".oO")}]  
[${"abcd"?right_pad(8, ".oO")}]
```

这将输出：

```
[ ]
[a ]
[ab ]
[abc ]
[abcd ]
[abcde]
[abcdef]
[abcdefg]
[abcdefgh]

[.oO.oO.o]
[aoO.oO.o]
[abO.oO.o]
[abc.oO.o]
[abcdoO.o]
```

### 1.1.18 contains 包含

**注意：**

这个内建函数从 FreeMarker 2.3.1 版本开始可用。在 2.3 版本中是没有的。

如果函数中的参数可以作为源字符串的子串，那么返回 `true`。比如：

```
<#if "piceous"?contains("ice")>It contains "ice"</#if>
```

将会输出：

```
It contains "ice"
```

### 1.1.19 matches 匹配

这是一个“超级用户”函数。不管你是否懂正则表达式。

**注意：**

这个函数仅仅对使用 Java2 平台的 1.4 版本之后起作用。否则它会发生错误并中止模板的处理。

这个函数决定了字符串是否精确匹配上模式。而且，它返回匹配的子串列表。返回值是一个多类型的值：

- 布尔值：如果字符串精确匹配上模式返回 `true`，否则返回 `false`。例如，`"foo"?matches('fo*')` 是 `true`，但是 `"foo bar"?matches('fo*')` 是 `false`。
- 序列：字符串匹配子串的列表。可能是一个长度为 0 的序列。

比如：

```
<#if "fxo"?matches("f.?o")>Matches.<#else>Does not
match.</#if>

<#assign res = "foo bar fyo"?matches("f.?o")>
<#if res>Matches.<#else>Does not match.</#if>
Matching sub-strings:
<#list res as m>
- ${m}
</#list>
```

将会打印:

```
Matches.

Does not match.
Matching sub-strings:
- foo
- fyo
```

如果正则表达式包含分组（括号），那么你可以使用内建函数 `groups` 来访问它们：

```
<#assign res = "aa/rx; ab/r;"?matches("(\\w[^/]+)/([^;]+);")>
<#list res as m>
- ${m} is ${m?groups[1]} per ${m?groups[2]}
</#list>
```

这会打印:

```
- aa/rx; is aa per rx
- ab/r; is ab per r
```

`matches` 接受两个可选的标记参数，要注意它不支持标记 `r`，因为它通常使用正则表达式。

### 1.1.20 number 数字格式

字符串转化为数字格式。这个数字必须是你 `FTL` 中直接指定数值的格式。也就是说，它必须以本地独立的形式出现，小数的分隔符就是一个点。此外这个函数认识科学记数法。（比如 `"1.23E6"`，`"1.5e-8"`）。

如果这恶搞字符串不在恰当的格式，那么在你尝试访问这个函数时，错误会抛出并中止模板的处理。

已知的问题：如果你使用比 `Java2` 平台 1.3 版本还早的版本，这个函数将不会被识别，前缀和科学记数法都不会起作用。

### 1.1.21 replace 替换

在源字符串中,用另外一个字符穿来替换原字符串中出现它的部分。它不处理词的边界。比如:

```
${"this is a car acarus"?replace("car", "bulldozer")}
```

将会打印:

```
this is a bulldozer abulldozerus
```

替换是从左向右执行的。这就意味着:

```
${"aaaaa"?replace("aaa", "X")}
```

将会打印:

```
Xaa
```

如果第一个参数是空字符串,那么所有的空字符串将会被替换,比如 `"foo"?replace("", "|")`, 就会得到 `"|f|o|o|"`。

`replace` 接受一个可选的标记参数,作为第三个参数。

### 1.1.22 rtf 富文本

字符串作为富文本 (RTF 文本), 也就是说, 下列字符串:

- `\` 替换为 `\\`
- `{` 替换为 `\{`
- `}` 替换为 `\}`

### 1.1.23 url URL 转义

**注意:**

这个内建函数从 FreeMarker 2.3.1 版本开始可用。在 2.3 版本中是没有的。

在 URL 之后的字符串进行转义。这意味着, 所有非 US-ASCII 的字符和保留的 URL 字符将会被 `%XX` 形式来转义。例如:

```
<#assign x = 'a/b c'>
${x?url}
```

输出将是 (假设用来转义的字符集是 US-ASCII 兼容的字符集):

```
a%2Fb%20c
```

注意它会转义所有保留的 URL 字符 (`/`, `=`, `&` 等), 所以编码可以被用来对查询参数的值进行, 比如:

```
<a href="foo.cgi?x=${x?url}&y=${y?url}">Click here...</a>
```

注意：

上面的没有 HTML 编码（`?htm`）是必要的，因为 URL 转义所有保留的 HTML 编码。但是要小心：通常引用的属性值，用普通引号（`"`）包括，而不是单引号（`'`），因为单引号是不被 URL 转义的。

为了进行 URL 转义，必须要选择字符集，它被用来计算被转义的部分（`%XX`）。如果你是 HTML 页面设计者，而且你不懂这个，不要担心：程序员应该配置 **FreeMarker**，则它默认使用恰当的字符集（程序员应该多看看下面的内容）。如果你是一个比较热衷于技术的人，那么你也也许想知道被 `url_escaping_charset` 设置的指定字符集，它可以在模板的执行时间设置（或者，更好的是，由程序员之前设置好）。例如：

```
<#--
  This will use the charset specified by the programmers
  before the template execution has started.
-->
<a href="foo.cgi?x=${x?url}">foo</a>

<#-- Use UTF-8 charset for URL escaping from now: -->
<#setting url_escaping_charset="UTF-8">

<#-- This will surely use UTF-8 charset -->
<a href="bar.cgi?x=${x?url}">bar</a>
```

此外，你可以明确地指定一个为单独 URL 转义的字符集，作为内建函数的参数：

```
<a href="foo.cgi?x=${x?url('ISO-8895-2')}">foo</a>
```

如果内建函数 `url` 没有参数，那么它会使用由 `url_escaping_charset` 设置的字符集。这个设置应该被软件设置，包括 **FreeMarker**（比如一个 Web 应用框架），因为它不会被默认设置为 `null`。如果它没有被设置，那么 **FreeMarker** 退回使用 `output_encoding` 的设置，这个也会被默认设置，所以它也是又软件设置的。如果 `output_encoding` 也没有被设置，那么没有参数的内建函数 `url` 将不会被执行，二期它会引起运行时错误。当然，有参数的 `url` 函数将会执行。

用 `setting` 指令在模板中设置 `url_escaping_charset` 是可能的。至少在真实的 MVC 应用中，这是一个不好的实践行为。`output_encoding` 不能由 `setting` 指令来设置，所以它应该是软件的工作。你可以阅读程序开发指南/其它/字符集问题来获取更多信息。

## 1.1.24 split 分割

它被用来根据另外一个字符串的出现将原字符串分割成字符串序列。比如：

```
<#list "someMOOtestMOOtext"?split("MOO") as x>
- ${x}
</#list>
```

将会打印：

```
- some
- test
- text
```

既然已经假设所有分隔符的出现是在新项之前，因此：

```
<#list "some,,test,text,"?split(",") as x>
- "${x}"
</#list>
```

将会打印：

```
- "some"
- ""
- "test"
- "text"
- ""
```

`split` 函数接受一个可选的标记参数作为第二个参数。

### 1.1.25 starts\_with 以...开头

如果字符串以指定的子字符串开头，那么返回 `true`。比如 `"redhead"?starts_with("red")` 返回布尔值 `true`，而且 `"red"?starts_with("red")` 也返回 `true`。

### 1.1.26 string（当被用作是字符串值时）

什么也不做，仅仅返回和其内容一致的字符串。例外的是，如果值是一个多类型的值（比如同时有字符串和序列两种），那么结果就只是一个简单的字符串，而不是多类型的值。这可以被用来防止多种人为输入。

### 1.1.27 trim 修整字符串

去掉字符串首尾的空格。例如：

```
(${ " green mouse " ?trim })
```

输出是：

```
(green mouse)
```



### 1.1.28 upper\_case 大写形式

字符串的大写形式。比如"GrEeN MoUsE"将会是"GREEN MOUSE"。

### 1.1.29 word\_list 词列表

包含字符串词的列表，并按它们在字符串中的顺序出现。词是连续的字符序列，包含任意字符，但是不包括空格。比如：

```
<#assign words = "  a bcd, .  1-2-3"?word_list>
<#list words as word>[${word}]/</#list>
```

将会输出：

```
[a][bcd],[.][1-2-3]
```

### 1.1.30 xhtml XHTML 格式

字符串作为 XHTML 格式文本输出，下面这些：

- < 替换为 &lt;
- > 替换为 &gt;
- & 替换为 &amp;
- " 替换为 &quot;
- ' 替换为 &#39;

这个函数和 `xml` 函数的唯一不同是 `xhtml` 函数转义 ' 为 &#39;，而不是 &apos;，但是一些老版本的浏览器不正确解释 &apos;。

### 1.1.31 xml XML 格式

字符串作为 XML 格式文本输出，下面这些：

- < 替换为 &lt;
- > 替换为 &gt;
- & 替换为 &amp;
- " 替换为 &quot;
- ' 替换为 &apos;

### 1.1.32 通用标记

很多字符串函数接受一个可选的被称为“标记”的字符串参数。在这些字符串中，每一个字符都影响着内建函数行为的一个特定方面。比如，字母 `i` 表示内建函数不应该在同一个字母的大小写上有所差异。标记中字母的顺序并不重要。

下面是标记字母的完整列表：

- **i**：大小写不敏感：不区分同一个字母大小写之间的差异。
- **f**：仅仅是第一。也就是说，替换/查找等，只是第一次出现的东西。
- **r**：查找的子串是正则表达式。**FreeMarker** 使用变化的正则表达式，在 <http://java.sun.com/j2se/1.4.1/docs/api/java/util/regex/Pattern.html> 中描述。只有你使用 **Java2** 平台的 1.4 版本以后，标记才会起作用。否则它会发生错误导致模板处理停止。
- **m**：正则表达式多行模式。在多行模式下，表达式 **^** 和 **\$** 仅仅匹配前后，分别是一行结尾或者是字符串的结束。默认这些表达式仅仅匹配整个字符串的开头和结尾。
- **s**：启用正则表达式的 **dotall** 模式（和 **Perl** 的单行模式一样）。在 **dotall** 模式下，表达式 **.** 匹配任意字符串，包括行结束符。默认这个表达式不匹配行结束符。
- **c**：在正则表达式中许可空白和注释。

示例：

```
<#assign s = 'foo bAr baar'>
${s?replace('ba', 'XY')}
i: ${s?replace('ba', 'XY', 'i')}
if: ${s?replace('ba', 'XY', 'if')}
r: ${s?replace('ba*', 'XY', 'r')}
ri: ${s?replace('ba*', 'XY', 'ri')}
rif: ${s?replace('ba*', 'XY', 'rif')}
```

这会输出：

```
foo bAr XYar
i: foo XYr XYar
if: foo XYr baar
r: foo XYAr XYr
ri: foo XYr XYr
rif: foo XYr baar
```

这是内建函数使用通用标记的表格，哪些支持什么样的标记。

内建函数	<b>i</b>	<b>r</b>	<b>m</b>	<b>s</b>	<b>c</b>	<b>f</b>
<b>replace</b>	是	是	只和 <b>r</b>	只和 <b>r</b>	只和 <b>r</b>	是
<b>split</b>	是	是	只和 <b>r</b>	只和 <b>r</b>	只和 <b>r</b>	否
<b>match</b>	是	否	是	是	是	否

## 1.2 处理数字的内建函数

相关的 **FAQs**：如果你有和 1,000,000 或 1 000 000 而不是 1000000 类似的东西，或者是 3.14 而不是 3,14 的东西，反之亦然，请参考 **FAQ** 中相关内容，也要注意内建函数 **c** 的所有内容。

### 1.2.1 c 数字转字符

#### 注意：

这个内建函数从 FreeMarker 2.3.3 以后开始存在。

这个函数将数字转换成字符串，这都是对计算机来说的，而不是对用户。也就是说，它根据程序语言的用法来进行格式化，这对于 FreeMarker 的所有本地数字格式化设置来说是独立的。它通常使用点来作为小数分隔符，而且它从来不用分组分隔符（像 3,000,000），指数形式（比如 5E20），多余的在开头或结尾的 0（比如 03 或 1.0），还有+号（比如+1）。它最多在小数点后打印 16 位，因此数值的绝对值小于 1E-16 将会显示为 0。这个函数非常严格，因为作为默认（像 `${x}` 这样）数字被本地（语言，国家）特定的数字格式转换为字符串，这是让用户来看的（比如 3000000 可能会被打印为 3,000,000）。当数字不对用户打印时（比如，对于一个数据库记录 ID，用作是 URL 的一部分，或者是 HTML 表单中的隐藏域，或者打印 CSS/JavaScript 的数值文本），这个函数必须被用来打印数字（也就是使用 `${x?c}` 来代替 `${x}`），否则输出可能由于当前数字格式设置，本地化（比如在一些国家中，小数点不是点，而是逗号）和数值（像大数可能被分组分隔符“损坏”）而损坏。

### 1.2.2 string（当用作是数值类型时） 数字转字符串

将一个数字转换成字符串。它使用程序员已经指定的默认格式。你也可以明确地用这个函数再指定一个数字格式，这在后面会展示。

有四种预定义的数字格式：computer, currency, number 和 percent。这些格式的明确含义是本地化(国家)指定的，受 Java 平台安装环境所控制，而不是 FreeMarker，除了 computer，用作和函数 c 是相同的格式。你可以这样来使用预定义的格式：

```
<#assign x=42>
${x}
${x?string} <!-- the same as ${x} -->
${x?string.number}
${x?string.currency}
${x?string.percent}
${x?string.computer}
```

如果你本地是 US English，那么就会生成：

```
42
42
42
$42.00
4,200%
42
```

前三个表达式的输出是相同的，因为前两个表达式是默认格式，这里是数字。你可以使用一个设置来改变默认设置：

```
<#setting number_format="currency">
<#assign x=42>
${x}
${x?string} <!-- the same as ${x} -->
${x?string.number}
${x?string.currency}
${x?string.percent}
```

现在会输出:

```
$42.00
$42.00
42
$42.00
4,200%
```

因为默认的数字格式被设置成了“货币”。

除了这三种预定义格式,你可以使用 Java 中数字格式语法写的任意的数字格式(<http://java.sun.com/j2se/1.4/docs/api/java/text/DecimalFormat.html>):

```
<#assign x = 1.234>
${x?string("0")}
${x?string("0.#")}
${x?string("0.##")}
${x?string("0.###")}
${x?string("0.####")}

${1?string("000.00")}
${12.1?string("000.00")}
${123.456?string("000.00")}

${1.2?string("0")}
${1.8?string("0")}
${1.5?string("0")} <-- 1.5, rounded towards even neighbor
${2.5?string("0")} <-- 2.5, rounded towards even neighbor

${12345?string("0.##E0")}
```

输出这些:

```

1
1.2
1.23
1.234
1.234

001.00
012.10
123.46

1
2
2 <-- 1.5, rounded towards even neighbor
2 <-- 2.5, rounded towards even neighbor

1.23E4

```

在金融和统计学中，四舍五入都是根据所谓的一半原则，这就意味着对最近的“邻居”进行四舍五入，除非离两个邻居距离相等，这种情况下，它四舍五入到偶数的邻居。如果你注意看 1.5 和 2.5 的四舍五入的话，这在上面的示例中是可以看到的，两个都被四舍五入到 2，因为 2 是偶数，但 1 和 3 是奇数。

除了 Java 小数语法模式之外，你可以编写如 `${aNumber?string("currency")}` 这样的代码，它会做和 `${aNumber?string.currency}` 一样的事情。

正如之前展示的预定义格式，默认的数字格式也可以在模板中进行设置：

```

<#setting number_format="0.##">
${1.234}

```

输出这个：

```
1.23
```

要注意数字格式是本地化敏感的：

```

<#setting locale="en_US">
US people write:      ${12345678?string(",##0.00")}
<#setting locale="hu">
Hungarian people write: ${12345678?string(",##0.00")}

```

输出这个：

```

US people write:      12,345,678.00
Hungarian people write: 12 345 678,00

```

### 1.2.3 round,floor,ceiling 数字的舍入处理

注意：

内建函数 `round` 从 FreeMarker 2.3.13 版本之后才存在。

使用确定的舍入法则，转换一个数字到整数：

- `round`：返回最近的整数。如果数字以.5 结尾，那么它将进位（也就是说向正无穷方向进位）
- `floor`：返回数字的舍掉小数后的整数（也就是说向服务穷舍弃）
- `ceiling`：返回数字小数进位后的整数（也就是说向正无穷进位）

示例：

```
<#assign testlist=[
  0, 1, -1, 0.5, 1.5, -0.5,
  -1.5, 0.25, -0.25, 1.75, -1.75]>
<#list testlist as result>
  ${result} ?floor=${result?floor} ?ceiling=${result?ceiling}
  ?round=${result?round}
</#list>
```

打印：

```
0 ?floor=0 ?ceiling=0 ?round=0
1 ?floor=1 ?ceiling=1 ?round=1
-1 ?floor=-1 ?ceiling=-1 ?round=-1
0.5 ?floor=0 ?ceiling=1 ?round=1
1.5 ?floor=1 ?ceiling=2 ?round=2
-0.5 ?floor=-1 ?ceiling=0 ?round=0
-1.5 ?floor=-2 ?ceiling=-1 ?round=-1
0.25 ?floor=0 ?ceiling=1 ?round=0
-0.25 ?floor=-1 ?ceiling=0 ?round=0
1.75 ?floor=1 ?ceiling=2 ?round=2
-1.75 ?floor=-2 ?ceiling=-1 ?round=-2
```

这些内建函数在分页处理时也许有用。如果你仅仅想展示数字的舍入形式，那么你应该使用内建函数 `string` 和 `numer_format` 设置。

## 1.3 处理日期的内建函数

### 1.3.1 string（当用作日期值时）日期转字符串

这个内建函数以指定的格式转换日期类型到字符串类型。（当默认格式由 FreeMarker 的 `date_format`, `time_format` 和 `datetime_format` 设置来支配时是很好的选择，那么你就不需要使用这个内建函数了。）

格式可以是预定义格式中的一种，或者你可以指定明确的格式化模式。

预定义的格式是 `short`, `medium`, `long` 和 `full`, 它们定义了冗长的结果文本输出。例如, 如果输出的本地化是 `U.S. English`, 而且时区是 `U.S. Pacific`, 那么下面的代码:

```
${openingTime?string.short}
${openingTime?string.medium}
${openingTime?string.long}
${openingTime?string.full}

${nextDiscountDay?string.short}
${nextDiscountDay?string.medium}
${nextDiscountDay?string.long}
${nextDiscountDay?string.full}

${lastUpdated?string.short}
${lastUpdated?string.medium}
${lastUpdated?string.long}
${lastUpdated?string.full}
```

将会打印这样的内容:

```
12:45 PM
12:45:09 PM
12:45:09 PM CEST
12:45:09 PM CEST

4/20/07
Apr 20, 2007
April 20, 2007
Friday, April 20, 2007

4/20/07 12:45 PM
Apr 20, 2007 12:45:09 PM
April 20, 2007 12:45:09 PM CEST
Friday, April 20, 2007 12:45:09 PM CEST
```

`short`, `medium`, `long` 和 `full` 的精确含义是以当前本地 (语言) 设置为主的。此外, 它不是由 `FreeMarker` 确定的, 而是由你运行 `FreeMarker` 的 `Java` 平台实现的。

对于包含日期和时间两部分的日期类型, 你可以分别指定日期和时间部分的长度:

```
${lastUpdated?string.short_long} <!-- short date, long time -->
${lastUpdated?string.medium_short} <!-- medium date, short time -->
```

将会输出:

```
4/8/03 9:24:44 PM PDT
Apr 8, 2003 9:24 PM
```

注意 `?string.short` 和 `?string.short_short` 是相同的，`?string.medium` 和 `?string.medium_medium` 是相同的，等等。

#### 警告！

不幸的是，由于 Java 平台的限制，你可以在数据模型中保存日期变量，那里 **FreeMarker** 不能决定变量是否存储的是日期部分（年，月，日），还是时间部分（时，分，秒，毫秒），还是两者都有。这种情况下，当你编写如 `${lastUpdated?string.short}`，或简单的 `${lastUpdated}` 时 **FreeMarker** 不知道如何来显示日期，因此模板会中止执行并抛出错误。要阻止这些发生，你可以使用内建函数 `?date`，`?time` 和 `?datetime` 来帮助 **FreeMarker**。比如 `${lastUpdated?datetime?string.short}`。要询问程序员数据模型中确定的变量是否有这个问题，或通常使用内建函数 `?date`，`?time` 和 `?datetime` 来处理。

要代替使用预定义的格式，你可以使用 `?string(pattern_string)` 来精确指定格式的模式。这个模式使用 Java 日期格式的语法。比如：

```
${lastUpdated?string("yyyy-MM-dd HH:mm:ss zzzz")}  
${lastUpdated?string("EEE, MMM d, 'yy")}  
${lastUpdated?string("EEEE, MMMM dd, yyyy, hh:mm:ss a  
'('zzz')'")}
```

将会输出：

```
2003-04-08 21:24:44 Pacific Daylight Time  
Tue, Apr 8, '03  
Tuesday, April 08, 2003, 09:24:44 PM (PDT)
```

#### 注意：

不像预定义格式，你不能和精确的给定模式使用内建函数 `?date`，`?time` 和 `?datetime`，因为使用这个模式你就告诉 **FreeMarker** 来显示哪部分的日期。然而，**FreeMarker** 将盲目地信任你，如果你显示的部分不存在于变量中，所以你可以显示“干扰”。比如 `${openingTime?string("yyyy-MM-dd hh:mm:ss a")}`，而 `openingTime` 中只存储了时间，将会显示 `1970-01-01 09:24:44 PM`。

定义模式的字符串也可以是 `"short"`，`"medium"`，`"short_medium"` 等。这和你使用预定义格式语法：`someDate?string("short")` 和 `someDate?string.short` 是相同的。

也可以参考：模板开发指南/模板/插值中的日期部分。

## 1.3.2 date, time, datetime 日期，时间，时间日期类型

这些内建函数用来指定日期变量中的哪些部分被使用：

- `date`：仅仅年，月和日的部分被使用。
- `ime`：仅仅时，分，秒和毫秒的部分被使用。
- `datetime`：日期和时间量部分都使用。

在最佳情况下，你不需要使用这些内建函数。不幸的是，由于 Java 平台上的技术限制，**FreeMarker** 有时不能发现日期中的哪一部分在使用（也就是说，仅仅年+月+日在使用，或仅仅时+分+秒+毫秒在使用，或两种都用）；询问程序员哪些变量会有这个问题。如果 **FreeMarker**



不得不执行需要这些信息的操作-比如用文本显示日期-但是它不知道哪一部分在使用，它会以错误来中止运行。这就是你不得不使用内建函数的时候了。比如，假设 `openingTime` 是一个有这样问题的变量：

```
<#assign x = openingTime> <!-- no problem can occur here -->
${openingTime?time} <!-- without ?time it would fail -->
<!-- For the sake of better understanding, consider this: -->
<#assign openingTime = openingTime?time>
${openingTime} <!-- this will work now -->
```

这些函数的另外一种用法：来截短日期。比如：

```
Last updated: ${lastUpdated} <!-- assume that lastUpdated is a
date-time value -->
Last updated date: ${lastUpdated?date}
Last updated time: ${lastUpdated?time}
```

将会输出这样的东西：

```
Last updated: 04/25/2003 08:00:54 PM
Last updated date: 04/25/2003
Last updated time: 08:00:54 PM
```

如果`?`的左边是字符串，那么这些内建函数会将字符串转换为日期变量。

## 1.4 处理布尔值的内建函数

### 1.4.1 string（当被用作是布尔值时） 转换布尔值为字符串

转换布尔值到字符串。你也以两种方式使用：

以 `foo?string`：这样会使用代表 `true` 和 `false` 值的默认字符串来转换布尔值为字符串。默认情况，`true` 被翻译为 `"true"`，而 `false` 被翻译为 `"false"`。如果你用 `FreeMarker` 来生成源代码，这是很有用的，因为这个值不是对本地化（语言，国家）敏感的。为了改变这些默认的字符串，你可以使用 `boolean_format` 设置。注意，如果变量是多类型的变量，也就是有布尔值和字符串，那么变量的字符串值会被返回。

以 `foo?string("yes", "no")`：如果布尔值是 `true`，这会返回第一个参数（这里是：`"yes"`），否则就返回第二个参数（这里是：`"no"`）。注意返回的值是一个字符串；如果参数是数字类型，首先它会被转换成字符串。

## 1.5 处理序列的内建函数

### 1.5.1 first 第一个子变量

序列的第一个子变量。如果序列为空，那么模板处理将会中止。

## 1.5.2 last 最后一个子变量

序列的最后一个子变量。如果序列为空，那么模板处理将会中止。

## 1.5.3 seq\_contains 序列包含...

**注意：**

这个内建函数从 FreeMarker 2.3.1 版本开始可用。而在 2.3 版本中不存在。

**注意：**

`seq_` 前缀在这个内建函数中是需要的，用来和 `contains` 区分开。`contains` 函数用来在字符串中查找子串（因为变量可以同时当作字符串和序列）。

辨别序列中是否包含指定值。它包含一个参数，就是来查找的值。比如：

```
<#assign x = ["red", 16, "blue", "cyan"]>
"blue": ${x?seq_contains("blue"?string("yes", "no"))}
"yellow": ${x?seq_contains("yellow"?string("yes", "no"))}
16: ${x?seq_contains(16)?string("yes", "no")}
"16": ${x?seq_contains("16"?string("yes", "no"))}
```

输出是：

```
"blue": yes
"yellow": no
16: yes
"16": no
```

为了查找值，这个函数使用了 FreeMarker 的比较规则（就像你使用的`==`运算符），除了比较两个不同类型的值，或 FreeMarker 不支持的类型来比较，其他都不会引起错误，只是为认为两个值不相等。因此，你可以使用它来查找标量值（也就是字符串，数字，布尔值，或日期/时间类型）。对于其他类型结果通常都是 `false`。

对于容错性，这个函数还对 `collections` 起作用。

## 1.5.4 seq\_index\_of 第一次出现...时的位置

**注意：**

这个内建函数从 FreeMarker 2.3.1 版本开始可用。而在 2.3 版本中不存在。

**注意：**

`seq_` 前缀在这个内建函数中是需要的，用来和 `index_of` 区分开。`index_of` 函数用来在字符串中查找子串（因为变量可以同时当作字符串和序列）。

返回序列中第一次出现该值时的索引位置，如果序列不包含指定的值时返回 `-1`。要查找的值作为第一个参数。比如这个模板：

```
<#assign colors = ["red", "green", "blue"]>
${colors?seq_index_of("blue")}
${colors?seq_index_of("red")}
${colors?seq_index_of("purple")}
```

将会输出：

```
2
0
-1
```

为了查找值，这个函数使用了 **FreeMarker** 的比较规则（就像你使用的`==`运算符），除了比较两个不同类型的值，或 **FreeMarker** 不支持的类型来比较，其他都不会引起错误，只是为认为两个值不相等。因此，你可以使用它来查找标量值（也就是字符串，数字，布尔值，或日期/时间类型）。对于其他类型结果通常是`-1`。

搜索开始的地方可以由第二个可选的参数来确定。如果在同一个序列中相同的项可以多次出现时，这是很有用的。第二个参数的数值没有什么限制：如果它是负数，那么就与它是零的效果一样，而如果它是比序列长度还大的数，那么就与它是序列长度值的效果一样。小数值会被切成整数。比如：

```
<#assign names = ["Joe", "Fred", "Joe", "Susan"]>
No 2nd param: ${names?seq_index_of("Joe")}
-2: ${names?seq_index_of("Joe", -2)}
-1: ${names?seq_index_of("Joe", -1)}
0: ${names?seq_index_of("Joe", 0)}
1: ${names?seq_index_of("Joe", 1)}
2: ${names?seq_index_of("Joe", 2)}
3: ${names?seq_index_of("Joe", 3)}
4: ${names?seq_index_of("Joe", 4)}
```

将会输出：

```
No 2nd param: 0
-2: 0
-1: 0
0: 0
1: 2
2: 2
3: -1
4: -1
```

### 1.5.5 seq\_last\_index\_of 最后一次出现..的位置

**注意：**

这个内建函数从 **FreeMarker 2.3.1** 版本开始可用。而在 **2.3** 版本中不存在。

**注意：**

`seq` 前缀在这个内建函数中是需要的，用来和 `last_index_of` 区分开。  
`last_index_of` 用于在字符串中搜索子串（因为一个变量可以同时是字符串和序列）。  
返回序列中最后一次出现值的索引位置，如果序列不包含指定的值时返回 `-1`。也就是说，和 `seq_index_of` 相同，只是在序列中从最后一项开始向前搜索。它也支持可选的第二个参数来确定从哪里开始搜索的索引位置。比如：

```
<#assign names = ["Joe", "Fred", "Joe", "Susan"]>
No 2nd param: ${names?seq_last_index_of("Joe")}
-2: ${names?seq_last_index_of("Joe", -2)}
-1: ${names?seq_last_index_of("Joe", -1)}
0: ${names?seq_last_index_of("Joe", 0)}
1: ${names?seq_last_index_of("Joe", 1)}
2: ${names?seq_last_index_of("Joe", 2)}
3: ${names?seq_last_index_of("Joe", 3)}
4: ${names?seq_last_index_of("Joe", 4)}
```

将会输出这个：

```
No 2nd param: 2
-2: -1
-1: -1
0: 0
1: 0
2: 2
3: 2
4: 2
```

## 1.5.6 reverse 反转序列

序列的反序形式。

## 1.5.7 size 序列大小

序列中子变量的数量（作为一个数值）。假设序列中至少有一个子变量，那么序列 `s` 中最大的索引是 `s?size - 1`（因为第一个子变量的序列是 `0`）。

## 1.5.8 sort 排序

以升序方式存储序列并返回。这仅在子变量都是字符串时有效，或者子变量都是数字，从 FreeMarker 2.3.1 以后，子变量是日期值（日期，时间，或日期+时间）也可以使用。如果子变量是字符串，它使用本地化（语言）的具体单词排序（通常不是大小写敏感的）。比如：

```
<#assign ls = ["whale", "Barbara", "zeppelin", "aardvark",  
"beetroot"]?sort>  
<#list ls as i>${i} </#list>
```

将会打印（至少是 US 区域设置）：

```
aardvark Barbara beetroot whale zeppelin
```

## 1.5.9 sort\_by 以...来排序

返回由给定的哈希表子变量来升序排序的哈希表序列，。这个规则和内建函数 `sort` 是一样的。除了序列中的子变量必须是哈希表类型，而且你不得不给顶哈希变量的名字，那会用来决定排序顺序。比如：

```
<#assign ls = [  
  {"name":"whale", "weight":2000},  
  {"name":"Barbara", "weight":53},  
  {"name":"zeppelin", "weight":-200},  
  {"name":"aardvark", "weight":30},  
  {"name":"beetroot", "weight":0.3}  
Order by name:  
<#list ls?sort_by("name") as i>  
- ${i.name}: ${i.weight}  
</#list>  
Order by weight:  
<#list ls?sort_by("weight") as i>  
- ${i.name}: ${i.weight}  
</#list>
```

将会打印（至少是 US 区域设置）：

```
Order by name:  
- aardvark: 30  
- Barbara: 53  
- beetroot: 0.3  
- whale: 2000  
- zeppelin: -200  
Order by weight:  
- zeppelin: -200  
- beetroot: 0.3  
- aardvark: 30  
- Barbara: 53  
- whale: 2000
```

从 FreeMarker 2.3.1 版本开始,如果你想使用来排序的子变量的层次比较深(也就是说,如果它是子变量中的子变量),那么你可以使用序列作为参数,它指定了子变量的名称,来向下引导所需的子变量。比如:

```
<#assign members = [
  {"name": {"first": "Joe", "last": "Smith"}, "age": 40},
  {"name": {"first": "Fred", "last": "Crooger"}, "age": 35},
  {"name": {"first": "Amanda", "last": "Fox"}, "age": 25}]>
Sorted by name.last:
<#list members?sort_by(['name', 'last']) as m>
- ${m.name.last}, ${m.name.first}: ${m.age} years old
</#list>
```

将会打印 (至少是 US 区域设置):

```
Sorted by name.last:
- Crooger, Fred: 35 years old
- Fox, Amanda: 25 years old
- Smith, Joe: 40 years old
```

## 1.5.10 chunk 区块

**注意:**

这个内建函数从 FreeMarker 2.3.3 版本以后可用。

这个内建函数分割序列到多个大小为函数的第一个参数给定的序列 (就像 `mySeq?chunk(3)`)。结果是包含这些序列的一个序列。最后一个序列可能比给定的大小要小,处分第二个参数也给出了 (比如 `mySeq?chunk(3, '-')`), 那个就是用来填充最后一个序列, 以达到给定的大小。比如:

```
<#assign seq = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']>

<#list seq?chunk(4) as row>
  <#list row as cell>${cell} </#list>
</#list>

<#list seq?chunk(4, '-') as row>
  <#list row as cell>${cell} </#list>
</#list>
```

这会输出:

```
a b c d
e f g h
i j

a b c d
e f g h
i j - -
```

这个函数通常在输出的序列中使用表格/柱状的格式。当被用于 HTML 表格时，第二个参数通常是 `"\xA0"`（也就是不换行的空格代码，也就是我们所知的“nbsp”），所以空 TD 的边界就不会不显示。

第一个参数必须是一个数字，而且至少是 1。如果这个数字不是整数，那么它会被静默地去掉小数部分（也就是说 3.1 和 3.9 都会被规整为 3）。第二个参数可以是任意类型的值。

## 1.6 处理哈希表的内建函数

### 1.6.1 keys 键的集合

一个包含哈希表中查找到的键的序列。注意并不是所有的哈希表都支持这个（询问程序员一个指定的哈希表是否允许这么操作）。

```
<#assign h = {"name":"mouse", "price":50}>
<#assign keys = h?keys>
<#list keys as key>${key} = ${h[key]}; </#list>
```

输出：

```
name = mouse; price = 50;
```

因为哈希表通常没有定义子变量的顺序，那么键名称的返回顺序就是任意的。然而，一些哈希表维持一个有意义的顺序（询问程序员指定的哈希表是否是这样）。比如，由上述 `{...}` 语法创建的哈希表保存了和你指定子变量相同的顺序。

### 1.6.2 值的集合

一个包含哈希表中子变量的序列。注意并不是所有的哈希表都支持这个（询问程序员一个指定的哈希表是否允许这么操作）。

至于返回的值的顺序，和函数 `keys` 的应用是一样的；看看上面的叙述就行了。

## 1.7 处理节点（XML）的内建函数

注意由这些内建函数返回的变量是由用于节点变量的实现生成的。意思就是返回的变量可以有更多的特性，附加于它这里的状态。比如，由内建函数 `children` 返回的序列和

XML DOM 节点也可以被用作是哈希表或字符串，这在第三部分 XML 处理指南中有解释。

### 1.7.1 children 子节点序列

一个包含该节点所有子节点（也就是直接后继节点）的序列。

XML: 这和特殊的哈希表的键\*几乎是一样的。除了它返回所有节点，而不但是元素。所以可能的子节点是元素节点，文本节点，注释节点，处理指令节点等，而且还可能是属性节点。属性节点排除在序列之外。

### 1.7.2 parent 父节点

在节点树中，返回该节点的直接父节点。根节点没有父节点，所以对于根节点，表达式 `node?parent??` 的值就是 `false`。

XML: 注意通过这个函数返回的值也是一个序列（当你编写 `someNode[".."]` 时，和 XPath 表达式 `..` 的结果是一样的）。也要注意属性节点，它返回属性所属的元素节点，尽管属性节点不被算作是元素的子节点。

### 1.7.3 root 根节点

该节点所属节点树的根节点。

XML: 根据 W3C，XML 文档的根节点不是最顶层的元素节点，而是文档本身，是最高元素的父节点。例如，如果你想得到被称为是 `foo` 的 XML（所谓的“文档元素”，不要和“文档”搞混了）的最高元素，那么你必须编写 `someNode?root.foo`。如果你仅仅写了 `someNode?root`，那么你得到的是文档本身，而不是文档元素。

### 1.7.4 ancestors 祖先节点

一个包含所有节点祖先节点的序列，以直接父节点开始，以根节点结束。这个函数的结果也是一个方法，你可以用它和元素的完全限定名来过滤结果。比如以名称 `section` 用 `node?ancestors("section")` 来获得所有祖先节点的序列，

### 1.7.5 node\_name 节点名称

当被“访问”时，返回用来决定哪个用户自定义指令来调用控制这个节点的字符串。可以参见 `visit` 和 `recurse` 指令。

XML: 如果节点是元素或属性，那么字符串就会是元素或属性的本地（没有前缀）名字。否则，名称通常在节点类型之后以 `@` 开始。可以参见 XML 处理指南中 2.2 节的形式化描述。要注意这个节点名称与在 DOM API 中返回的节点名称不同；FreeMarker 节点名称的目标是给要处理节点的用户自定义指令命名。



## 1.7.6 node\_type 节点类型

描述节点类型的字符串。**FreeMarker** 没有对节点类型定义准确的含义；它依赖于变量是怎么建模的。也可能节点并不支持节点类型。在这种情形下，该函数就返回未定义值，所以你就不能使用返回值。（你可以用 `node?node_type??` 继续检查一个节点是否是支持类型属性）

**XML**：可能的值是 `"attribute"`，`"text"`，`"comment"`，`"document_fragment"`，`"document"`，`"document_type"`，`"element"`，`"entity"`，`"entity_reference"`，`"notation"`，`"pi"`。注意没有 `"cdata"` 类型，因为 CDATA 被认为是普通文本元素。

## 1.7.7 node\_namespace 节点命名空间

返回节点命名空间的字符串。**FreeMarker** 没有为节点命名空间定义准确的含义；它依赖于变量是怎么建模的。也可能节点没有定义任何节点命名空间。这种情形下，该函数应该返回未定义的变量（也就是 `node?node_namespace??` 的值是 `false`），所以你不能使用这个返回值。

**XML**：这种情况下的 XML，就是 XML 命名空间的 URI（比如 `"http://www.w3.org/1999/xhtml"`）。如果一个元素或属性节点没有使用 XML 命名空间，那么这个函数就返回一个空字符串。对于其他 XML 节点这个函数返回未定义的变量。

## 1.8 很少使用的和专家级的内建函数

这些是你通常情况下不应该使用的内建函数，但是在特殊情况下（调试，高级宏）它们会有用。如果你需要在普通页面模板中使用这些函数，你可能会重新访问数据模型，所以你不要使用它们。

### 1.8.1 byte, double, float, int, long, short

返回一个包含原变量中相同值的 `SimpleNumber`，但是在内部表示值中使用了 `java.lang.Type`。如果方法被重载了，这是有用的，或者一个 `TemplateModel` 解包器在自动选择适合的 `java.lang.*` 类型有问题时。注意从 2.39 版本开始，解包器有本质上改进，所以你将基本不会使用到这些内建函数了，除非在重载方法调用中来解决一些含糊的东西。

### 1.8.2 eval 求值

这个函数求一个作为 FTL 表达式的字符串的值。比如 `"1+2"?eval` 返回数字 3。

### 1.8.3 has\_content 是否有内容

如果变量（不是 Java 的 `null`）存在而且不是“空”就返回 `true`，否则返回 `false`。“空”的含义靠具体的情形来决定。它是直观的常识性概念。下面这些都是空：长度为 0 的字符串，没有子变量的序列或哈希表，一个已经超过最后一项元素的集合。如果值不是字符串，序列，哈希表或集合，如果它是数字，日期或布尔值（比如 0 和 `false` 是非空的），那么它被认为是非空的，否则就是空的。注意当你的数据模型实现了多个模板模型接口，你可能会得到不是预期的结果。然而，当你有疑问时你通常可以使用 `expr!?.size > 0` 或 `expr!?.length > 0` 来代替 `expr?.has_content`。

这个函数是个特殊的函数，你可以使用像默认值操作符那样的圆括号手法。也就是说，你可以编写 `product.color?.has_content` 和 `(product.color)?.has_content` 样的代码。第一个没有控制当 `product` 为空的情形，而第二个控制了。

### 1.8.4 interpret 将字符串解释为 FTL 模板

这个函数解释字符串作为 FTL 模板，而且返回一个用户自定义指令，也就是当应用于任意代码块中时，执行模板就像它当时被包含一样。例如：

```
<#assign x=["a", "b", "c"]>
<#assign templateSource = r"<#list x as y>${y}</#list>">
<#-- Note: That r was needed so that the ${y} is not interpreted
above -->
<#assign inlineTemplate = templateSource?.interpret>
<@inlineTemplate />
```

输出为：

```
abc
```

正如你看到的，`inlineTemplate` 是用户自定义指令，也就是当被执行时，运行当时使用 `interpret` 函数生成的模板。

你也可以在两个元素的序列中应用这个函数。这种情况下序列的第一个元素是模板源代码，第二个元素是内联模板的名字。在调试时它可以给内联模板一个确定的名字，这是很有用的。所以你可以这么来写代码：

```
<#assign inlineTemplate = [templateSource, "myInlineTemplate"]?.interpret>
```

在上述的模板中，要注意给内联模板一个名字没有及时的效果，它仅仅在你得到错误报告时可以得到额外的信息。

### 1.8.5 is\_... 判断函数族

这些内建函数用来检查变量的类型，然后根据类型返回或。下面是 `is_...` 判断函数族的列表：

内建函数	如果值是...时返回 <b>true</b>
<code>is_string</code>	字符串
<code>is_number</code>	数字
<code>is_boolean</code>	布尔值
<code>is_date</code>	日期（所有类型：仅日期，仅时间和时间日期）
<code>is_method</code>	方法
<code>is_transform</code>	变换
<code>is_macro</code>	宏
<code>is_hash</code>	哈希表
<code>is_hash_ex</code>	扩展的哈希表（也就是支持 <code>?keys</code> 和 <code>?values</code> ）
<code>is_sequence</code>	序列
<code>is_collection</code>	集合
<code>is_enumerable</code>	序列或集合
<code>is_indexable</code>	序列
<code>is_directive</code>	指令的类型（比如宏，或 <code>TemplateDirectiveModel</code> ， <code>TemplateTransformModel</code> 等）
<code>is_node</code>	节点

## 1.8.6 namespace 命名空间

这个函数返回和宏变量关联的命名空间（也就是命名空间的入口哈希表）。你只能和宏一起来用它。

## 1.8.7 new 创建 TemplateModel 实现

这是用来创建一个确定 `TemplateModel` 实现的变量的内建函数。

在`?`的左边你可以指定一个字符串，是 `TemplateModel` 实现类的完全限定名。结果是调用构造方法生成一个方法变量，然后将新变量返回。

比如：

```
<!-- Creates an user-defined directive be calling the
parameterless constructor of the class -->
<#assign word_wrapp =
"com.acmee.freemarker.WordWrapperDirective"?new()>
<!-- Creates an user-defined directive be calling the
constructor with one numerical argument -->
<#assign word_wrapp_narrow =
"com.acmee.freemarker.WordWrapperDirective"?new(40)>
```

对于更多的关于构造方法参数被包装和如何选择重载的构造方法信息，请阅读：程序开发指南/其它/Bean 包装器部分内容。

## 第二章 指令参考文档

如果你没有在这里发现模板中的指令，可能你需要在废弃的 FTL 结构中来查找它了。

### 2.1 if, else, elseif 指令

#### 2.1.1 概要

```
<#if condition>
...
<#elseif condition2>
...
<#elseif condition3>
...
...
<#else>
...
</#if>
```

这里：

- `condition`, `condition2` 等：表达式将被计算成布尔值。

#### 2.1.2 描述

你可以使用 `if`, `elseif` 和 `else` 指令来条件判断是否越过模板的一个部分。这些 `condition`-s 必须计算成布尔值，否则错误将会中止模板处理。`elseif`-s 和 `else`-s 必须出现在 `if` 的内部（也就是，在 `if` 的开始标签和技术标签之间）。`if` 中可以包含任意数量的 `elseif`-s（包括 0 个）而且结束时 `else` 是可选的。

比如：

只有 `if`，没有 `elseif` 和 `else`：

```
<#if x == 1>
  x is 1
</#if>
```

只有 `if` 和 `else`，没有 `elseif`：

```
<#if x == 1>
  x is 1
<#else>
  x is not 1
</#if>
```

`if` 和两个 `elseif`，没有 `else`：

```
<#if x == 1>
  x is 1
<#elseif x == 2>
  x is 2
<#elseif x == 3>
  x is 3
</#if>
```

`if` 和 3 个 `elseif`, 还有 `else`:

```
<#if x == 1>
  x is 1
<#elseif x == 2>
  x is 2
<#elseif x == 3>
  x is 3
<#elseif x == 4>
  x is 4
<#else>
  x is not 1 nor 2 nor 3 nor 4
</#if>
```

要了解更多布尔表达式, 可以参考: 模板开发指南/模板/表达式部分内容。

你(当然)也可以嵌套 `if` 指令:

```
<#if x == 1>
  x is 1
  <#if y == 1>
    and y is 1 too
  <#else>
    but y is not
  </#if>
<#else>
  x is not 1
  <#if y < 0>
    and y is less than 0
  </#if>
</#if>
```

**注意:**

如何测试 `x` 比 1 大? `<#if x > 1>` 是不对的, 因为 `FreeMarker` 将会解释第一个 `>` 作为结束标记。因此, 编写 `<#if (x > 1)>` 或 `<#if x &gt; 1>` 是正确的。

## 2.2 switch, case, default, break 指令

### 2.2.1 概要

```
<#switch value>
  <#case refValue1>
    ...
    <#break>
  <#case refValue2>
    ...
    <#break>
  ...
  <#case refValueN>
    ...
    <#break>
  <#default>
    ...
</#switch>
```

这里：

- `value`, `refValue1` 等：表达式将会计算成相同类型的标量。

### 2.2.2 描述

这个指令的用法是不推荐的，因为向下通过的行为容易出错。使用 `elseif-s` 来代替，除非你想利用向下通过这种行为。

Switch 被用来选择模板中的一个片段，如何选择依赖于表达式的值：

```
<#switch being.size>
  <#case "small">
    This will be processed if it is small
    <#break>
  <#case "medium">
    This will be processed if it is medium
    <#break>
  <#case "large">
    This will be processed if it is large
    <#break>
  <#default>
    This will be processed if it is neither
</#switch>
```

在 `switch` 中间必须有一个或多个 `<#case value>`，在所有 `case` 标签之后，有一个可选的 `<#default>`。当 FreeMarker 到达指令时，它会选择一个 `refValue` 和

`value` 相等的 `case` 指令来继续处理模板。如果没有和合适的值匹配的 `case` 指令，如果 `default` 指令存在，那么就会处理 `default` 指令，否则就会继续处理 `switch` 结束标签之后的内容。现在有一个混乱的事情：当它选择一个 `case` 指令后，它就会继续处理 `case` 指令中的内容，直到遇到 `break` 指令。也就是它遇到另外一个 `case` 指令或 `<#default>` 标记时也不会自动离开 `switch` 指令。比如：

```
<#switch x>
  <#case x = 1>
    1
  <#case x = 2>
    2
  <#default>
    d
</#switch>
```

如果 `x` 是 1，那么它会打印 1 2 d；如果 `x` 是 2，那么就会打印 2 d；如果 `x` 是 3，那么它会打印 d。这就是前面提到的向下通过行为。`break` 标记指示 FreeMarker 直接略过剩下的 `switch` 代码段。

## 2.3 list, break 指令

### 2.3.1 概要

```
<#list sequence as item>
  ...
</#list>
```

这里：

- `sequence`：表达式将被算作序列或集合
- `item`：循环变量（不是表达式）的名称

### 2.3.2 描述

你可以使用 `list` 指令来处理模板的一个部分中的一个序列中包含的各个变量。在开始标签和结束标签中的代码将会被处理，首先是第一个子变量，然后是第二个子变量，接着是第三个子变量，等等，直到超过最后一个。对于每个变量，这样的迭代中循环变量将会包含当前的子变量。

在 `list` 循环中，有两个特殊的循环变量可用：

- `item_index`：这是一个包含当前项在循环中的步进索引的数值。
- `item_has_next`：来辨别当前项是否是序列的最后一项的布尔值。

示例 1：

```
<#assign seq = ["winter", "spring", "summer", "autumn"]>
<#list seq as x>
  ${x_index + 1}. ${x}<#if x_has_next>,</#if>
</#list>
```

将会打印:

```
1. winter,
2. spring,
3. summer,
4. autumn
```

示例 2: 你可以使用 `list` 在两个数字中来计数, 使用一个数字范围序列表达式:

```
<#assign x=3>
<#list 1..x as i>
  ${i}
</#list>
```

输出是:

```
1
2
3
```

注意上面的示例在你希望 `x` 是 0 的时候不会有作用, 那么它打印 0 和 -1。

你可以使用 `break` 指令在它通过最后一个序列的子变量之前离开 `list` 循环。比如这会仅仅打印 “winter” 和 “spring”。

```
<#list seq as x>
  ${x}
  <#if x = "spring"><#break></#if>
</#list>
```

注意如果你开启经典的兼容模式, 那么 `list` 也接受一个标量而且将它视为单元素的序列。

通常来说, 避免 `list` 中使用无论何时可能包装了 `Iterator` 作为参数的集合和使用包装了 `java.util.Collection` 或序列的集合是最好的。但是在某些情况, 当你处理时仅仅有一个 `Iterator`。要注意如果你传递了一个包装了 `Iterator` 的集合给 `list`, 你仅仅可以迭代一次元素, 因为 `Iterators` 是由它们一次性对象的特性决定的。当你尝试第二次列出这样一个集合变量时, 错误会中止模板的处理。

## 2.4 include 指令

### 2.4.1 概要

```
<#include path>
```



or

```
<#include path options>
```

这里：

- **path**：要包含文件的路径；一个算作是字符串的表达式。（用其他话说，它不用是一个固定的字符串，它也可以是像 `profile.baseDir + "/menu.ftl"` 这样的东西。）
- **options**：一个或多个这样的选项：`encoding=encoding, parse=parse`
  - ◆ **encoding**：算作是字符串的表达式
  - ◆ **parse**：算作是布尔值的表达式（为了向下兼容，也接受一部分字符串值）

## 2.4.2 描述

你可以使用它在你的模板中插入另外一个 **FreeMarker** 模板文件（由 **path** 参数指定）。被包含模板的输出格式是在 **include** 标签出现的位置插入的。被包含的文件和包含它的模板共享变量，就像是被复制粘贴进去的一样。**include** 指令不能由被包含文件的内容所替代，它只是当 **FreeMarker** 每次在模板处理期间到达 **include** 指令时处理被包含的文件。所以对于如果 **include** 在 **list** 循环之中的例子，你可以为每个循环周期内指定不同的文件名。

**注意：**

这个指令不能和 **JSP** (**Servlet**) 的 **include** 搞混，因为它不涉及到 **Servlet** 容器中，只是处理应外一个 **FreeMarker** 模板，不能“离开”**FreeMarker**。关于如何处理“**JSP include**”，可以参考 **FAQ** 中的内容。

**path** 参数可以是如 `"foo.ftl"` 和 `"../foo.ftl"` 一样的相对路径，或者是如 `"/foo.ftl"` 这样的绝对路径。相对路径是相对于使用 **import** 指令的模板文件夹。绝对路径是相对于程序员在配置 **FreeMarker** 时定义的基路径（通常指代“模板的根路径”）。

**注意：**

这和 **FreeMarker 2.1** 版本之前的处理方式不同，之前的路径通常是绝对路径。为了保留原来的行为，要在 **Configuration** 对象中开启经典的兼容模式。

通常使用/（斜杠）来分隔路径成分，而不是\（反斜杠）。如果你从你本地的文件系统加载模板，要使用反斜杠（像 **Windows** 操作系统）。**FreeMarker** 会自动转换它们。

比如：

假设 `/common/copyright.ftl` 包含：

```
Copyright 2001-2002 ${me}<br>
All rights reserved.
```

那么这个：

```
<#assign me = "Juila Smith">
<h1>Some test</h1>
<p>Yeah.
<hr>
<#include "/common/copyright.ftl">
```

会打印出：

```
<h1>Some test</h1>
<p>Yeah.
<hr>
Copyright 2001-2002 Juila Smith
All rights reserved.
```

支持的 *options* 选项有：

- **parse**: 如果它为真，那么被包含的文件将会当作 FTL 来解析，否则整个文件将被视为简单文本（也就是说不会在其中查找 FreeMarker 的结构）。如果你忽略了这个选项，那么它默认是 true。
- **encoding**: 被包含文件从包含它的文件继承的编码方式（实际就是字符集），除非你用这个选项来指定编码方式。编码名称要和 `java.io.InputStreamReader` 中支持的那些一致（对于 *Java API 1.3* 版本: *MIME* 希望的字符集是从 *IANA* 字符集注册处得到的）。合法的名字有: ISO-8859-2, UTF-8, Shift\_JIS, Big5, EUC-KR, GB2312。

比如：

```
<#include "/common/navbar.html" parse=false encoding="Shift_JIS">
```

注意，对于所有模板可能会用 *Configuration* 的“自动包含”设置自动处理通用的包含物。

### 2.4.2.1 使用获得机制

有一个特殊的路径组成，是用一个星号（\*）来代表的。它被解释为“当前目录或其他任意它的父目录”。因此，如果模板在 `/foo/bar/template.ftl` 位置上，有下面这行：

```
<#include "*/footer.ftl">
```

那么引擎就会在下面的位置上寻找模板，并按这个顺序：

- `/foo/bar/footer.ftl`
- `/foo/footer.ftl`
- `/footer.ftl`

这种机制被称为 **acquisition 获得** 并允许设计者在父目录中放置通用的被包含的文件，而且当需要时在每个子路径基础上重新定义它们。我们说包含它们的模板获得了从包含它的第一个父目录中的模板。注意你不但可以在星号的右面指定一个模板的名字，也可以指定一个子路径。也就是说，如果前面的模板由下面这个所替代：

```
<#include "*/commons/footer.ftl">
```

那么引擎将会从下面的路径开始寻找模板，并按这个顺序：

- `/foo/bar/commons/footer.ftl`
- `/foo/commons/footer.ftl`
- `/commons/footer.ftl`

然而，在路径中最大只能有一个星号。指定多余一个星号会导致模板不能被发现。

### 2.4.2.2 本地化查找

无论何时模板被加载，它都被分配了一个本地化环境。本地化环境是语言和可选的国家或方言标识。模板一般是由程序员编写一些代码来加载的，出于一些方面的考虑，程序员为模板选择一种本地化环境。比如：当 `FreemarkerServlet` 加载模板时，它经常用本地化环境匹配浏览器请求 Web 页面的语言偏好来请求模板。

当一个模板包含另一个模板时，它试图加载以相同的本地化环境加载模板。假定你的模板以本地化 `en_US` 来加载，那就意味着是 U.S. English。当你包含另外一个模板：

```
<include "footer.ftl">
```

那么引擎实际上就会寻找一些模板，并按照这个顺序：

- `footer_en_US.ftl`
- `footer_en.ftl`
- `footer.ftl`

要注意你可以使用 *Configuration* 的 *setLocalizedLookup* 方法关闭本地化查找特性。

当你同时使用获得机制和本地化查找时，在父目录中有指定本地化的模板优先于在子目录中有很少本地化的模板。假设你使用下面的代码来包含 `/foo/bar/template.ftl`：

```
<include "*/footer.ftl">
```

引擎将会查找这些模板，并按照这个顺序：

- `/foo/bar/footer_en_US.ftl`
- `/foo/footer_en_US.ftl`
- `/footer_en_US.ftl`
- `/foo/bar/footer_en.ftl`
- `/foo/footer_en.ftl`
- `/footer_en.ftl`
- `/foo/bar/footer.ftl`
- `/foo/footer.ftl`
- `/footer.ftl`

## 2.5 import 指令

### 2.5.1 概要

```
<#import path as hash>
```

这里：

- `path`：模板的路径。这是一个算作是字符串的表达式。（换句话说，它不是一个固定的字符串，它可以是这样的一些东西，比如，`profile.baseDir + "/menu.ftl"`。）
- `hash`：哈希表变量的结束名称，你可以由它来访问命名空间。这不是表达式。

## 2.5.2 描述

引入一个库。也就是说，它创建一个新的命名空间，然后在那个命名空间中执行给定 `path` 参数中的模板，所以模板用变量（宏，函数等）填充命名空间。然后使得新创建的命名空间对哈希表的调用者可用。这个哈希表变量将会在命名空间中，由 `import`（就像你可以用 `assign` 指令来创建一样。）的调用者被创建成一个普通变量，名字就是 `hash` 参数给定的。

如果你用同一个 `path` 多次调用 `import`，它会创建命名空间，但是只运行第一次 `import` 的调用。后面的调用仅仅创建一个哈希表变量，你只是通过它来访问同一个命名空间。

由引入的模板打印的输出内容将会被忽略（不会在包含它的地方被插入）。模板的执行是用变量填充命名空间，而不是写到输出中。

例如：

```
<#import "/libs/mylib.ftl" as my>

<@my.copyright date="1999-2002"/>
```

`path` 参数可以是一个相对路径，比如 `"foo.ftl"` 和 `"../foo.ftl"`，或者是像 `"/foo.ftl"` 一样的绝对路径。相对路径是相对于使用 `import` 指令模板的目录。绝对路径是程序员配置 **FreeMarker** 时定义的相对于根路径（通常指代“模板的根目录”）的路径。

通常使用 `/`（斜杠）来分隔路径组成，而不是 `\`（反斜杠）。如果你从你本地的文件系统中加载模板，那么它使用反斜杠（比如在 **Windows** 环境下），**FreeMarker** 将会自动转换它们。

像 `include` 指令一样，获得机制和本地化擦找也可以用来解决路径问题。

注意，对于所有模板来说，它可以自动做通用的引入操作，使用 *Configuration* 的“自动引入”设置就行了。

如果你命名空间不是很了解，你应该阅读：模板开发指南/其他/命名空间部分的内容。

## 2.6 noparse 指令

### 2.6.1 概要

```
<#noparse>
...
</#noparse>
```

### 2.6.2 描述

**FreeMarker** 不会在这个指令体中间寻找 **FTL** 标签，插值和其他特殊的字符序列，除了 `noparse` 的结束标记。

例如：

Example:

-----

**<#noparse>**

<#list animals as being>

<tr><td>\${being.name}<td>\${being.price} Euros

</#list>

**</#noparse>**

将会输出:

Example:

-----

<#list animals as being>

<tr><td>\${being.name}<td>\${being.price} Euros

</#list>

## 2.7 compress 指令

### 2.7.1 概要

<#compress>

...

</#compress>

### 2.7.2 描述

当你使用了对空白不敏感的格式（比如 HTML 或 XML）时压缩指令对于移除多余的空白是很有用的。它捕捉在指令体（也就是在开始标签和结束标签中）中生成的内容，然后缩小所有不间断的空白序列到一个单独的空白字符。如果被替代的序列包含换行符或是一段空间，那么被插入的字符也会是一个换行符。开头和结尾的不间断的空白序列将会完全被移除。

```
<#assign x = "    moo  \n\n  ">
```

```
(<#compress>
```

```
  1 2 3  4    5
```

```
  ${moo}
```

```
  test only
```

```
  I said, test only
```

```
</#compress>)
```

会输出：

```
(1 2 3 4 5
moo
test only
I said, test only)
```

## 2.8 escape, noescape 指令

### 2.8.1 概要

```
<#escape identifier as expression>
...
<#noescape>...</#noescape>
...
</#escape>
```

### 2.8.2 描述

当你使用 `escape` 指令包围模板中的一部分时，在块中出现的插值 (`${...}`) 会和转义表达式自动结合。这是一个避免编写相似表达式的很方便的方法。它不会影响在字符串形式的插值（比如在 `<#assign x = "Hello ${user}!">`）。而且，它也不会影响数值插值 (`#{...}`)。

例如：

```
<#escape x as x?html>
  First name: ${firstName}
  Last name: ${lastName}
  Maiden name: ${maidenName}
</#escape>
```

事实上它等同于：

```
First name: ${firstName?html}
Last name: ${lastName?html}
Maiden name: ${maidenName?html}
```

注意它和你在指令中用什么样的标识符无关 - 它仅仅是作为一个转义表达式的正式参数。

当你在 `include` 指令中调用宏时，理解在模板文本中转义仅仅对出现在 `<#escape ...>` 和 `</#escape>` 中的插值起作用是很重要的。也就是说，它不会转义文本中 `<#escape ...>` 之前的东西或 `</#escape>` 之后的东西，也不会从 `escape-d` 部分中来调用。

```

<#assign x = "<test>">
<#macro m1>
  m1: ${x}
</#macro>
<#escape x as x?html>
  <#macro m2>m2: ${x}</#macro>
  ${x}
  <@m1/>
</#escape>
${x}
<@m2/>

```

输出将是：

```

&lt;test&gt;
m1: <test>
<test>
m2: &lt;test&gt;

```

从更深的技术上说，*escape* 指令的作用是用在模板解析的时间而不是模板处理的时间。这就表示如果你调用一个宏或从一个转义块中包含另外一个模板，它不会影响到宏/被包含模板中的插值，因为宏调用和模板包含被算在模板处理时间。另外一方面，如果你用一个转义区块包围一个或多个宏声明（算在模板解析时间，和宏调用想法），那么那些宏中的插值将会和转义表达式合并。

有时需要暂时为一个或两个在转义区块中的插值关闭转义。你可以通过关闭，过后再重新开启转义区块来达到这个功能，但是那么你不得不编写两遍转义表达式。你可以使用非转义指令来替代：

```

<#escape x as x?html>
  From: ${mailMessage.From}
  Subject: ${mailMessage.Subject}
  <#noescape>Message:
  ${mailMessage.htmlFormattedBody}</#noescape>
  ...
</#escape>

```

和这个是等同的：

```

From: ${mailMessage.From?html}
Subject: ${mailMessage.Subject?html}
Message: ${mailMessage.htmlFormattedBody}
...

```

转义可以被嵌套（尽管你不会在罕见的情况下来做）。因此，你可以编写如下面代码（这个例子固然是有点伸展的，正如你可能会使用 *list* 来迭代序列中的每一项，但是我们现在所做的是阐述这个观点）的东西：

```
<#escape x as x?html>
  Customer Name: ${customerName}
  Items to ship:
  <#escape x as itemCodeToNameMap[x]>
    ${itemCode1}
    ${itemCode2}
    ${itemCode3}
    ${itemCode4}
  </#escape>
</#escape>
```

实际上和下面是等同的:

```
Customer Name: ${customerName?html}
Items to ship:
  ${itemCodeToNameMap[itemCode1]?html}
  ${itemCodeToNameMap[itemCode2]?html}
  ${itemCodeToNameMap[itemCode3]?html}
  ${itemCodeToNameMap[itemCode4]?html}
```

当你在嵌入的转义区块内使用非转义指令时,它仅仅不处理一个单独层级的转义。因此,为了在两级深的转义区块内完全关闭转义,你需要使用两个嵌套的非转义指令。

## 2.9 assign 指令

### 2.9.1 概要

```
<#assign name=value>
or
<#assign name1=value1 name2=value2 ... nameN=valueN>
or
<#assign same as above... in namespacehash>
or
<#assign name>
  capture this
</#assign>
or
<#assign name in namespacehash>
  capture this
</#assign>
```

这里:

- **name**: 变量的名字。不是表达式。而它可以本写作是字符串,如果变量名包含保留字符这是很有用的,比如<#assign "foo-bar" = 1>。注意这个字符串没有展开插值(如"\${foo}")。



- **value**: 存储的值。是表达式。
- **namespacehash**: (通过 **import**) 为命名空间创建的哈希表。是表达式。

## 2.9.2 描述

使用这个指令你可以创建一个新的变量，或者替换一个已经存在的变量。注意仅仅顶级变量可以被创建/替换（也就是说你不能创建/替换 `some_hash.subvar`，除了 `some_hash`）。

关于变量的更多内容，请阅读：模板开发指南/其他/在模板中定义变量

例如： `seasons` 变量可以存储一个序列：

```
<#assign seasons = ["winter", "spring", "summer", "autumn"]>
```

比如：变量 `test` 中存储增长的数字：

```
<#assign test = test + 1>
```

作为一个方便的特性，你可以使用一个 **assign** 标记来进行多次定义。比如这个会做上面两个例子中相同的事情：

```
<#assign
  seasons = ["winter", "spring", "summer", "autumn"]
  test = test + 1
>
```

如果你知道什么是命名空间：**assign** 指令在命名空间中创建变量。通常它在当前的命名空间（也就是和标签所在模板关联的命名空间）中创建变量。但如果你用了 **in namespacehash**，那么你可以用另外一个命名空间来创建/替换变量。比如，这里你在命名空间中 `/mylib.ftl` 创建/替换了变量 `bgColor`。

```
<#import "/mylib.ftl" as my>
<#assign bgColor="red" in my>
```

**assign** 的极端使用是当它捕捉它的开始标记和结束标记中间生成的输出时。也就是说，在标记之间打印的东西将不会在页面上显示，但是会存储在变量中。比如：

```
<#macro myMacro>foo</#macro>
<#assign x>
  <#list 1..3 as n>
    ${n} <@myMacro />
  </#list>
</#assign>
Number of words: ${x?word_list?size}
${x}
```

将会打印：

```
Number of words: 6
  1 foo
  2 foo
  3 foo
```

请注意你不应该使用它来往字符串中插入变量：

```
<#assign x>Hello ${user}!</#assign> <!-- BAD PRACTICE! -->
```

你可以这么来写：

```
<#assign x="Hello ${user}!">
```

## 2.10 global 指令

### 2.10.1 概要

```
<#global name=value>
or
<#global name1=value1 name2=value2 ... nameN=valueN>
or
<#global name>
  capture this
</#global>
```

这里：

- **name**：变量的名称。它不是表达式。但它可以被写作是字符串形式，如果变量名包含保留字符这是很有用的，比如`<#global "foo-bar" = 1>`。注意这个字符串没有扩展插值（如`"${foo}"`）。
- **value**：存储的值，是表达式。

### 2.10.2 描述

这个指令和 `assign` 相似，但是被创建的变量在所有的命名空间中都可见，但又不会存在于任何一个命名空间之中。精确地说，正如你会创建（或替换）一个数据模型变量。因此，这个变量是全局的。如果在数据模型中，一个相同名称的变量存在的话，它会被使用这个指令创建的变量隐藏。如果在当前的命名空间中，一个相同名称的变量存在的话，那么会隐藏由 `global` 指令创建的变量。

比如`<#global x = 1>`，用创建了一个变量，那么在所有命名空间中 `x` 都可见，除非另外一个称为 `x` 的变量隐藏了它（比如你已经用`<#assign x = 2>`创建了一个变量）。这种情形下，你可以使用特殊变量 `globals`，比如`${.globals.x}`。注意使用 `globals` 你看到所有全局可访问的变量；不但由 `global` 指令创建的变量，而且是数据模型中的变量。

自定义 JSP 标记的用户请注意：用这个指令创建的变量集合和 JSP 页面范围对应。这就

意味着，如果自定义 JSP 标记想获得一个页面范围的属性（page-scope bean），在当前命名空间中一个相同名称的变量，从 JSP 标记的观点出发，将不会隐藏。

## 2.11 local 指令

### 2.11.1 概要

```
<#local name=value>
or
<#local name1=value1 name2=value2 ... nameN=valueN>
or
<#local name>
    capture this
</#local>
```

这里：

- **name**：在 root 中局部对象的名称。它不是一个表达式。但它可以被写作是字符串形式，如果变量名包含保留字符这是很有用的，比如<#global "foo-bar" = 1>。注意这个字符串没有扩展插值（如"**`${foo}`**"）。
- **value**：存储的值，是表达式。

### 2.11.2 描述

它和 **assign** 指令类似，但是它创建或替换局部变量。这仅仅在宏和方法的内部定义才会有作用。

要获得更多关于变量的信息，可以阅读：模板开发指南/其他/在模板中定义变量部分内容。

## 2.12 setting 指令

### 2.12.1 概要

```
<#setting name=value>
```

这里：

- **name**：设置的名称。不是表达式！
- **value**：设置的值，是表达式。

### 2.12.2 描述

为进一步的处理而设置。设置是影响 FreeMarker 行为的值。新值仅仅在被设置的模板

处理时出现，而且不触碰模板本身。设置的初始值是由程序员设定的（参加：程序开发指南/配置/设置信息部分的内容）。

支持的设置有：

- **locale**: 输出的本地化（语言）。它可以影响数字，日期等显示格式。它的值是由语言编码（小写两个字母的 ISO-639 编码）和可选的国家码（大写的两个字母 ISO-3166 编码）组成的字符串，它们以下划线相分隔，如果我们已经指定了国家那么一个可选的不同编码（不是标准的）会以下划线分隔开国家。合法的值示例：`en`, `en_US`, `en_US_MAC`。FreeMarker 会尝试使用最特定可用的本地化设置，所以如果你指定了 `en_US_MAC`，但是它不被知道，那么它会尝试 `en_US`，然后尝试 `en`，然后是计算机（可能是由程序员设置的）默认的本地化设置。
- **number\_format**: 当没有指定确定的格式化形式时，用来转化数字到字符串形式的数字格式化设置。可以是下列中的一个预定义值 `number`（默认的），`computer`，`currency`，或 `percent`。此外，以 Java 小数数字格式化语法书写的任意的格式化形式也可以被指定。更多的格式形式请参考处理数字的内建函数 `string`。
- **boolean\_format**: 以逗号分隔的一对字符串来分别展示 `true` 和 `false` 值，当没有指定确定的格式时，转换布尔值到字符串。默认值是 `"true,false"`。也可以参考处理布尔值的内建函数 `string`。
- **date\_format**, **time\_format**, **datetime\_format**: 当没有指定确定的格式时，用来转换日期到字符串的日期/时间格式形式。如 `${someDate}`。 `date_format` 这个情形，它只影响和日期相关的日期（年，月，日）， `time_format` 只影响和时间相关的日期（时，分，秒，毫秒）， `datetime_format` 只影响时间日期类型的日期（年，月，日，时，分，秒，毫秒）。这个设置的可能值和处理日期的内建函数 `string` 的参数相似；可以在那部分参考更多内容。比如 `"short"`, `"long_medium"`, `"MM/dd/yyyy"`。
- **time\_zone**: 时区的名称来显示并格式化时间。默认情况下，使用系统的时区。也可以是 Java 时区 API 中的任何值。比如: `"GMT"`, `"GMT+2"`, `"GMT-1:30"`, `"CET"`, `"PST"`, `"America/Los_Angeles"`。
- **url\_escaping\_charset**: 用来 URL 转义（比如 `${foo?url}`）的字符集，来计算转义（`%XX`）的部分。通常包含 FreeMarker 的框架应该设置它，所以你不应该在模板中来设置。（程序员可以阅读程序开发指南/其他/字符集问题部分来获取更多内容）
- **classic\_compatible**: 这是对专业人员来说的，它的值应该是一个布尔值。参见 `freemarker.template.Configurable` 的文档来获取更多信息。

示例：假设模板的初始本地化是 `hu`（Hungarian，匈牙利），那么这个：

```
${1.2}
<#setting locale="en_US">
${1.2}
```

将会输出：

```
1,2
1.2
```

因为匈牙利人使用逗号作为小数的分隔符，而美国人使用点。

## 2.13 用户自定义指令（<@...>）

### 2.13.1 概要

```
<@user_def_dir_exp param1=val1 param2=val2 ... paramN=valN/>
```

（注意 XML 风格，>之前的/）

如果你需要循环变量，请参考 2.13.2.2 节内容。

```
<@user_def_dir_exp param1=val1 param2=val2 ... paramN=valN ;  
lv1, lv2, ..., lvN/>
```

或者和上面两个相同但是使用结束标签，请参考 2.13.2.1 节内容。

```
<@user_def_dir_exp ...>
```

...

```
</@user_def_dir_exp>
```

或

```
<@user_def_dir_exp ...>
```

...

```
</@>
```

或和上面的相同但是使用位置参数传递，请参考 2.13.2.3 节内容

```
<@user val1, val2, ..., valN/>
```

等...

这里：

- `user_def_dir_exp`：表达式算作是自定义指令（比如宏），将会被调用。
- `param1`, `param2` 等：参数的名称，它们不是表达式。
- `val1`, `val2` 等：参数的值，它们是表达式。
- `lv1`, `lv2` 等：循环变量的名称，它们不是表达式。

参数的数量可以是 0（也就是没有参数）。

参数的顺序并不重要（除非你使用了位置参数传递）。参数名称必须唯一。在参数名中小写和大写的字母被认为是不同的字母（也就是 `Color` 和 `color` 是不同的）。

### 2.13.2 描述

这将调用用户自定义指令，比如宏。参数的含义，支持和需要的参数的设置依赖于具体的自定义指令。

你可以阅读模板开发指南/其他/定义你自己的指令部分。

示例 1：调用存储在变量 `html_escape` 中的指令：

```
<@html_escape>  
a < b  
Romeo & Juliet  
</@html_escape>
```

输出：

```
a &lt; b
Romeo & Juliet
```

示例 2: 调用有参数的宏

```
<@list items=["mouse", "elephant", "python"] title="Animals"/>
...
<#macro list title items>
  <p>${title?cap_first}:
  <ul>
    <#list items as x>
      <li>${x?cap_first}
    </#list>
  </ul>
</#macro>
```

输出:

```
<p>Animals:
<ul>
  <li>Mouse
  <li>Elephant
  <li>Python
</ul>
...
```

### 2.13.2.1 结束标签

你可以在结束标签中忽略 `user_def_dir_exp`。也就是说,你可以写`</@>`来替代`</@anything>`。这个规则当表达式 `user_def_dir_exp` 太复杂时非常有用,因为你不需要在结束标签中重复表达式。此外,如果表达式包含比简单变量名和点还多的表达式,你就不能再重复它们了。比如`<@a_hash[a_method()]>...</@a_hash[a_method()]>`就是错的,你必须写为`<@a_hash[a_method()]>...</@>`。但是`<@a_hash.foo>...</@a_hash.foo>`是可以的。

### 2.13.2.2 循环变量

一些自定义指令创建循环变量(和 `list` 指令相似)。正如预定义指令(如 `list`)一样,当你调用这个指令(如`<#list foos as foo>...</#list>`中的 `foo`)时循环变量的名称就给定了,而变量的值是由指令本身设置的。在自定义指令的情形下,语法是循环变量的名称在分号之后给定。比如:

```
<@myRepeatMacro count=4 ; x, last>
  ${x}. Something... <#if last> This was the last!</#if>
</@myRepeatMacro>
```

注意由自定义指令创建的循环变量数量和分号之后指定的循环变量数量需要不匹配。也就是说，如果你对重复是否是最后一个不感兴趣，你可以简单来写：

```
<@myRepeatMacro count=4 ; x>
  ${x}. Something...
</@myRepeatMacro>
```

或者你可以：

```
<@myRepeatMacro count=4>
  Something...
</@myRepeatMacro>
```

此外，如果你在分号之后指定更多循环变量而不是自定义指令创建的，也不会引起错误，只是最后的循环变量不能被创建（也就是在嵌套内容中那些将是未定义的）。尝试使用未定义的循环变量，就会引起错误（除非你使用如`?default` 这样的内建函数），因为你尝试访问了一个不存在的变量。

请参考模板开发指南/其他/定义你自己的指令部分来获取更多内容。

### 2.13.2.3 位置参数传递

位置参数传递（如`<@heading "Preface", 1/>`）是正常命名参数传递（如`<@heading title="Preface" level=1/>`）的速记形式，这里忽略了参数的名称。如果自定义指令只有一个参数，或者对于经常使用的自定义指令它参数的顺序很好记忆，速记形式应该被应用。为了应用这种形式，你不得不了解声明的命名参数的顺序（如果指令只有一个参数这是很琐碎的）。也就是，如果 `heading` 被创建为`<#macro heading title level>...`，那么`<@heading "Preface", 1/>` 和 `<@heading title="Preface" level=1/>`（或 `<@heading level=1 title="Preface"/>`；如果你使用参数名称，那顺序就不重要了）是相等的。要注意位置参数传递现在仅仅支持宏定义。

## 2.14 macro, nested, return 指令

### 2.14.1 概要

```
<#macro name param1 param2 ... paramN>
...
<#nested loopvar1, loopvar2, ..., loopvarN>
...
<#return>
```

```
...
</#macro>
```

这里：

- **name**：宏变量的名称，它不是表达式。然而，它可以被写成字符串的形式，如果宏名称中包含保留字符时这是很有用的，比如`<#macro "foo-bar">...`。注意这个字符串没有扩展插值（如`"${foo}"`）。
- **param1**, **param2** 等：局部变量的名称，存储参数的值（不是表达式），在`=`号后面和默认值（是表达式）是可选的。默认值也可以是另外一个参数，比如`<#macro section title label=title>`。
- **paramN**，最后一个参数，可以可选的包含一个尾部省略（`...`），这就意味着宏接受可变的参数数量。如果使用命名参数来调用，**paramN** 将会是包含给宏的所有未声明的键/值对的哈希表。如果使用位置参数来调用，**paramN** 将是额外参数的序列。
- **loopvar1**, **loopvar2** 等：可选的循环变量的值，是 **nested** 指令想为嵌套内容创建的。这些都是表达式。

**return** 和 **nested** 指令是可选的，而且可以在`<#macro>`和`</#macro>`之间被用在任意位置和任意次数。

没有默认值的参数必须在有默认值参数（`paramName=defaultValue`）之前。

## 2.14.2 描述

创建一个宏变量（在当前命名空间中，如果你知道命名空间的特性）。如果你对宏和自定义指令不了解，你应该阅读模板开发指南/其他/定义你自己的指令部分。

宏变量存储模板片段（称为宏定义体）可以被用作自定义指令。这个变量也存储自定义指令的被允许的参数名。当你将这个变量作为指令时，你必须给所有参数赋值，除了有默认值的参数。默认值当且仅当你调用宏而不给参数赋值时起作用。

变量会在模板开始时被创建；而不管 **macro** 指令放置在模板的什么位置。因此，这样也可以：

```
<#-- call the macro; the macro variable is already created: -->
<@test/>
...

<#-- create the macro variable: -->
<#macro test>
  Test text
</#macro>
```

然而，如果宏定义被插在 **include** 指令中，它们直到 **FreeMarker** 执行 **include** 指令时才会可用。

例如：没有参数的宏：

```
<#macro test>
  Test text
</#macro>
<#-- call the macro: -->
<@test/>
```



输出:

```
Test text
```

示例: 有参数的宏:

```
<#macro test foo bar baaz>
  Test text, and the params: ${foo}, ${bar}, ${baaz}
</#macro>
<#-- call the macro: -->
<@test foo="a" bar="b" baaz=5*5-2/>
```

输出:

```
Test text, and the params: a, b, 23
```

示例: 有参数和默认值参数的宏:

```
<#macro test foo bar="Bar" baaz=-1>
  Test text, and the params: ${foo}, ${bar}, ${baaz}
</#macro>
<@test foo="a" bar="b" baaz=5*5-2/>
<@test foo="a" bar="b"/>
<@test foo="a" baaz=5*5-2/>
<@test foo="a"/>
```

输出:

```
Test text, and the params: a, b, 23
Test text, and the params: a, b, -1
Test text, and the params: a, Bar, 23
Test text, and the params: a, Bar, -1
```

示例: 一个复杂的宏。

```
<#macro list title items>
  <p>${title?cap_first}:
  <ul>
    <#list items as x>
      <li>${x?cap_first}
    </#list>
  </ul>
</#macro>
<@list items=["mouse", "elephant", "python"] title="Animals"/>
```

输出:

```
<p>Animals:
<ul>
  <li>Mouse
  <li>Elephant
  <li>Python
</ul>
```

示例：一个支持多个参数和命名参数的宏：

```
<#macro img src extra...>
  
    ${attr}="${extra[attr]?html}"
  </#list>
  >
</#macro>
<@img src="/images/test.png" width=100 height=50 alt="Test"/>
```

输出：

```

```

### 2.14.2.1 nested

`nested` 指令执行自定义指令开始和结束标签中间的模板片段。嵌套的片段可以包含模板中合法的任意内容：插值，指令...等。它在上下文环境中被执行，也就是宏被调用的地方，而不是宏定义体的上下文中。因此，比如，你不能看到嵌套部分的宏的局部变量。如果你没有调用 `nested` 指令，自定义指令开始和结束标记中的部分将会被忽略。

示例：

```
<#macro do_twice>
  1. <#nested>
  2. <#nested>
</#macro>
<@do_twice>something</@do_twice>
```

输出：

```
1. something
2. something
```

嵌套指令可以对嵌套内容创建循环变量。比如：

```
<#macro do_thrice>
  <#nested 1>
  <#nested 2>
  <#nested 3>
</#macro>
<@do_thrice ; x>
  ${x} Anything.
</@do_thrice>
```

这会打印：

```
1 Anything.
2 Anything.
3 Anything.
```

一个更为复杂的示例：

```
<#macro repeat count>
  <#list 1..count as x>
    <#nested x, x/2, x==count>
  </#list>
</#macro>
<@repeat count=4 ; c, halfc, last>
  ${c}. ${halfc}<#if last> Last!</#if>
</@repeat>
```

输出将是：

```
1. 0.5
2. 1
3. 1.5
4. 2 Last!
```

### 2.14.2.2 return

使用 `return` 指令，你可以在任意位置留下一个宏或函数定义。比如：

```
<#macro test>
  Test text
  <#return>
  Will not be printed.
</#macro>
<@test/>
```

输出：

Test text

## 2.15 function, return 指令

### 2.15.1 概要

```
<#function name param1 param2 ... paramN>
...
<#return returnValue>
...
</#function>
```

这里：

- **name**：方法变量的名称（不是表达式）
- **param1, param2** 等：局部变量的名称，存储参数的值（不是表达式），在=号后面和默认值（是表达式）是可选的。
- **paramN**，最后一个参数，可以可选的包含一个尾部省略（...），这就意味着宏接受可变的参数数量。局部变量 **paramN** 将是额外参数的序列。
- **returnValue**：计算方法调用值的表达式。

**return** 指令可以在`<#function ...>`和`</#function>`之间被用在任意位置和任意次数。

没有默认值的参数必须在有默认值参数（`paramName=defaultValue`）之前。

### 2.15.2 描述

创建一个方法变量（在当前命名空间中，如果你知道命名空间的特性）。这个指令和 **macro** 指令的工作方式一样，除了 **return** 指令必须有一个参数来指定方法的返回值，而且视图写入输出的将会被忽略。如果到达`</#function>`（也就是说没有 **return returnValue**），那么方法的返回值就是未定义变量。

示例 1：创建一个方法来计算两个树的平均值：

```
<#function avg x y>
  <#return (x + y) / 2>
</#function>
${avg(10, 20)}
```

将会打印：

15

示例 2：创建一个方法来计算多个数字的平均值：

```
<#function avg nums...>
  <#local sum = 0>
  <#list nums as num>
    <#local sum = sum + num>
  </#list>
  <#if nums?size != 0>
    <#return sum / nums?size>
  </#if>
</#function>
${avg(10, 20)}
${avg(10, 20, 30, 40)}
${avg()}! "N/A"
```

会打印：

```
15
25
N/A
```

## 2.16 flush 指令

### 2.16.1 概要

```
<#flush>
```

### 2.16.2 描述

当 FreeMarker 生成输出时，他/她通常存储生成的输出内容然后以一个或几个大片段送到客户端。这种发送的行为被称为冲洗（就像冲厕所）。尽管冲洗是自动发生的，有时你想在模板处理时的一点强制执行，这就是 `flush` 指令要做的。如果需要它，那么程序员会告诉你；通常你不必使用这个指令。自动冲洗的机制和 `flush` 指令的明确效果是在程序员的控制之下的。

冲洗简单调用当前使用 `java.io.Writer` 实例的 `flush` 方法。整体的缓冲和冲洗机制在 `writer`（就是传递给 `Template.process` 方法的参数）中已经实现了；FreeMarker 不用来处理它。

## 2.17 stop 指令

### 2.17.1 概要

`<#stop>`

或

`<#stop reason>`

这里：

- `reason`：关于终端原因的信息化消息。表达式被算做是字符串。

### 2.17.2 描述

中止模板的处理。这是一种像紧急中断的机制：不要在普通情况下使用。抛出 `StopException` 异常会发生中止，而且 `StopException` 会持有 `reason` 参数的值。

## 2.18 ftl 指令

### 2.18.1 概要

`<#ftl param1=value1 param2=value2 ... paramN=valueN>`

这里：

- `param1`, `param2` 等：参数的名字，不是表达式。允许的参数有 `encoding`, `strip_whitespace`, `strip_text` 等。参加下面。
- `value1`, `value2` 等：参数的值。必须是一个常量表达式（如 `true`，或 `"ISO-8859-5"`，或 `{x:1, y:2}`）。它不能用变量。

### 2.18.2 描述

告诉 `FreeMarker` 和其他工具关于模板的信息，而且帮助程序员来自动检测一个文本文件是否是 `FTL` 文件。这个指令，如果存在，必须是模板的第一句代码。该指令前的任何空白将被忽略。这个指令的老式语法（`#less`）格式是不被支持的。

一些设置（编码方式，空白剥离等）在这里给定的话就有最高的优先级，也就是说，它们直接作用于模板而不管其他任何 `FreeMarker` 配置的设置。

参数：

- `encoding`：使用这个你可以在模板文件中为模板指定编码方式（字符集）（也就是说，这是新创建 `Template` 的 `encoding` 设置，而且 `Configuration.getTemplate` 中的 `encoding` 参数不能覆盖它。）。要注意，`FreeMarker` 会尝试会和自动猜测的编码方式（这依赖于程序员对 `FreeMarker` 的配置）找到 `ftl` 指令并解释它，然后就会发现 `ftl` 指令会让一些东西有所不

同，之后以新的编码方式来读取模板。因此，直到 `ftl` 标记使用第一个编码方式读取到结尾，模板必须是合法的 FTL。这个参数的合法值是从 IANA 字符集注册表中参考 MIME 中的字符集名称。比如 ISO-8859-5，UTF-8 或 Shift\_JIS。

- `strip_whitespace`: 这将开启/关闭空白剥离。合法的值是布尔值常量 `true` 和 `false` (为了向下兼容，字符串 `"yes"`, `"no"`, `"true"`, `"false"` 也是可以的)。默认值 (也就是当你不使用这个参数时) 是依赖于程序员对 FreeMarker 的配置，但是对新的项目还应该是 `true`。
- `strip_text`: 当开启它时，当模板被解析时模板中所有顶级文本被移除。这个不会影响宏，指令，或插值中的文本。合法值是布尔值常量 `true` 和 `false` (为了向下兼容，字符串 `"yes"`, `"no"`, `"true"`, `"false"` 也是可以的)。默认值 (也就是当你不使用这个参数时) 是 `false`。
- `strict_syntax`: 这会开启/关闭“严格的语法”。合法值是布尔值常量 `true` 和 `false` (为了向下兼容，字符串 `"yes"`, `"no"`, `"true"`, `"false"` 也是可以的)。默认值 (也就是当你不使用这个参数时) 是依赖于程序员对 FreeMarker 的配置，但是对新的项目还应该是 `true` (程序员：对于 `config.setStrictSyntaxMode(true)`；你应该明确设置它为 `true`)。要获取更多信息，可以参考：废弃的 FTL 结构/老式 FTL 语法部分。
- `ns_prefixes`: 这是关联节点命名空间前缀的哈希表。比如：  
`{"e": "http://example.com/ebook",  
"vg": "http://example.com/vektorGraphics"}`。这通常是用于 XML 处理的，前缀可以用于 XML 查询，但是它也影响 `visit` 和 `recurse` 指令的工作。相同节点的命名空间只能注册一个前缀 (否则会发生错误)，所以在前缀和节点命名空间中是一对一的关系。前缀 `D` 和 `N` 是保留的。如果你注册前缀 `D`，那么除了你可以使用前缀 `D` 来关联节点命名空间，你也可以设置默认的前缀命名空间。前缀 `N` 就不能被注册；当且仅当前缀 `D` 被注册时，`N` 被用来表示在特定位置没有节点命名空间的节点。(要参考默认节点命名空间的用法，`N`，一般的前缀，可以看 XML 处理中 `visit` 和 `recurse` 指令。) `ns_prefixes` 的作用限制在单独的 FTL 命名空间内，换句话说，就是为模板创建的 FTL 命名空间内。这也意味着 `ns_prefixes` 当一个 FTL 命名空间为包含它的模板所创建时才有作用，否则 `ns_prefixes` 参数没有效果。FTL 命名空间当下列情况下为模板创建：(a) 模板是“主”模板，也就是说它不是被 `<#include ...>` 来调用的模板，但是直接被调用的 (和 `process` 一起的 `Template` 或 `Environment` 类的方法)；(b) 模板直接被 `<#import ...>` 调用。
- `attributes`: 这是关联模板任意属性 (名-值对) 的哈希表。属性的值可以是任意类型 (字符串，数字，序列等)。FreeMarker 不会尝试去理解属性的含义。它是由封装 FreeMarker (比如一个 Web 应用框架) 的应用程序决定的。因此，允许的属性的设置是它们依赖的应用 (Web 应用框架) 的语义。程序员：你可以通过关联 `Template` 对象的 `getCustomAttributeNames` 和 `getCustomAttribute` 方法 (从 `freemarker.core.Configurable` 继承而来) 获得属性。如当模板被解析时，关联 `Template` 对象的模板属性，属性可以在任意时间被读取，而模板不需要被执行。上面提到的方法返回未包装的属性值，也就是说，使用 FreeMarker 独立的类型，如 `java.util.List`。

这个指令也决定模板是否使用尖括号语法 (比如 `<#include 'foo.ftl'>`) 或方括号语法 (如 `[#include 'foo.ftl']`)。简单而言，这个指令使用的语法将会是整

个模板使用的语法，而不管 FreeMarker 是如何配置的。

## 2.19 t, lt, rt 指令

### 2.19.1 概要

<#t>

<#lt>

<#rt>

<#nt>

### 2.19.2 描述

这些指令，指示 FreeMarker 去忽略标记中行的特定的空白

- **t**（整体削减）：忽略本行中首和尾的所有空白。
- **lt**（左侧削减）：忽略本行中首部所有的空白。
- **rt**（右侧削减）：忽略本行中尾部所有的空白。

这里：

- “首部空白”表示本行所有空格和制表符（和其他根据 **UNICODE** 中的空白字符，除了换行符）在第一个非空白字符之前。
- “尾部空白”表示本行所有的空格和制表符（和其他根据 **UNICODE** 中的空白字符，除了换行符）在最后一个非空白字符之后，还有行末尾的换行符。

理解这些检查模板本身的指令是很重要的，而不是当你合并数据模型时，模板生成的输出。（也就是说，空白的移除发生在解析阶段）

比如这个：

```
--  
1 <#t>  
2<#t>  
3<#lt>  
4  
5<#rt>  
6  
--
```

将会输出：

```
--  
1 23  
4  
5 6  
--
```



这些指令在行内的放置不重要。也就是说，不管你是将它们放在行的开头，或是行的末尾，或是在行的中间，效果都是一样的。

## 2.20 nt 指令

### 2.20.1 概要

```
<#nt>
```

### 2.20.2 描述

“不要削减”。这个指令关闭行中出现的空白削减。它也关闭其他同一行中出现的削减指令（`t`，`lt`，`rt` 的效果）。

## 2.21 attempt, recover 指令

### 2.21.1 概要

```
<#attempt>
  attempt block
<#recover>
  recover block
</#attempt>
```

这里：

- `attempt block`：任意内容的模板块。这是会被执行的，但是如果期间发生了错误，那么这块内容的输出将会回滚，之后 `recover block` 就会被执行。
- `recover block`：任意内容的模板块。这个仅在 `attempt block` 执行期间发生错误时被执行。你可以在这里打印错误信息或其他操作。

`recover` 是命令的。`attempt/recover` 可以嵌套在其他 `attempt` s 或 `recover` s 中。

**注意：**

上面的格式是从 2.3.3 版本开始支持的，之前它是 `<#attempt>...<#recover>...</#recover>`，也支持向下兼容。此外，这些指令在 FreeMarker 2.3.1 版本时引入的，在 2.3 版本中是不存在的。

### 2.12.2 描述

如果你想让页面成功输出内容，尽管它在页面特定位置发生错误也这样，那么这些指令就是有用的。如果一个错误在 `attempt block` 执行期间发生，那么模板执行就会中止，

但是 *recover block* 会代替 *attempt block* 执行。如果在 *attempt block* 执行期间没有发生错误，那么 *recover block* 就会忽略。一个简单的示例如下：

```
Primary content
<#attempt>
  Optional content: ${thisMayFails}
<#recover>
  Ops! The optional content is not available.
</#attempt>
Primary content continued
```

如果 `thisMayFails` 变量不存在，那么输出：

```
Primary content
  Ops! The optional content is not available.
Primary content continued
```

如果 `thisMayFails` 变量存在而且值为 `123`，那么输出：

```
Primary content
  Optional content: 123
Primary content continued
```

*attempt block* 块有多或没有的语义：不管 *attempt block* 块的完整内容是否输出（没有发生错误），或者在 *attempt block*（没有发生错误）块执行时没有输出结果。比如，上面的示例，发生在“Optional content”之后的失败被打印出来了，而没有在“Ops!”之前输出。（这是在 *attempt block* 块内，侵入的输出缓冲的实现，就连 *flush* 指令也会送输出到客户端。）

为了阻止来自上面示例的误解：*attempt/recover* 不仅仅是处理未定义变量（对于那个可以使用不存在变量控制符来处理）。它可以处理发生在块执行期间的各种类型的错误（而不是语法错误，这会在执行之前被检测到）。它的目的是包围更大的模板段，错误可能发生在很多地方。比如，你在模板中有一个部分，来处理打印广告，但是它不是页面的主要内容，所以你不希望你的页面因为一些打印广告（也可能是短暂的数据库故障）的错误而挂掉。所以你将整个广告区域放在 *attempt block* 块中。

在一些环境下，程序员配置 **FreeMarker**，所以对于特定的错误，它不会中止模板的执行，在打印一些错误提示信息到输出（请参考：程序开发指南/配置/错误处理部分）中之后，而是继续执行。*attempt* 指令不会将这些抑制的错误视为错误。

在 *recover block* 块中，错误的信息存在特殊变量 `error` 中。不要忘了以点开始引用特殊变量（比如：`${.error}`）

在模板执行期间发生的错误通常被被日志记录，不管是否发生在 *attempt block* 块中。

## 2.22 visit, recurse, fallback 指令

### 2.22.1 概要

```
<#visit node using namespace>
```

或

```
<#visit node>
```

```
<#recurse node using namespace>
```

或

```
<#recurse node>
```

或

```
<#recurse using namespace>
```

或

```
<#recurse>
```

```
<#fallback>
```

这里：

- **node**：算作节点变量的表达式。
- **namespace**：一个命名空间，或者是命名空间的序列。命名空间可以以命名空间哈希表（又称为根哈希表）给定，或者可以引入一个存储模板路径的字符串。代替命名空间哈希表，你也可以使用普通哈希表。

### 2.22.2 描述

`visit` 和 `recurse` 指令是用来递归处理树的。在实践中，这通常被用来处理 XML。

#### 2.22.2.1 visit 指令

当你调用了`<#visit node>`时，它看上去像用户自定义指令（如宏）来调用从节点名称（`node?node_name`）和命名空间（`node?node_namespace`）中有名称扣除的节点。名称扣除的规则：

- 如果节点不支持节点命名空间（如 XML 中的文本节点），那么这个指令名仅仅是节点的名称（`node?node_name`）。如果 `getNodeNamespace` 方法返回 `null` 时节点就不支持节点命名空间了。
- 如果节点支持节点命名空间（如 XML 中的元素节点），那么从节点命名空间中的前缀扣除可能在节点名称前和一个做为分隔符（比如 `e:book`）的冒号追加上去。前缀，以及是否使用前缀，依赖于何种前缀在 FTL 命名空间中用 `ftl` 指令的 `ns_prefixes` 参数注册的，那里 `visit` 寻找控制器指令（`visit` 调用的相同 FTL 命名空间不是重要的，后面你将会看到）。具体来说，如果没有用 `ns_prefixes` 注册默认的命名空间，那么对于不属于任何命名空间（当

`getNodeNamespace` 返回"" 的节点来说就不使用前缀。如果使用 `ns_prefixes` 给不属于任意命名空间的节点注册了默认命名空间, 那么就使用前缀 `N`, 而对于属于默认节点命名空间的节点就不使用前缀了。否则, 这两种情况下, 用 `ns_prefixes` 关联节点命名空间的前缀已经被使用了。如果没有关联节点命名空间的节点前缀, 那么 `visit` 仅仅就好像没有以合适的名称发现指令。

自定义指令调用的节点对于特殊变量 `.node` 是可用的。比如:

```
<#-- Assume that nodeWithNameX?node_name is "x" -->
<#visit nodeWithNameX>
Done.
<#macro x>
  Now I'm handling a node that has the name "x".
  Just to show how to access this node: this node has
  ${.node?children?size} children.
</#macro>
```

输出就像:

```
Now I'm handling a node that has the name "x".
Just to show how to access this node: this node has 3 children.
Done.
```

如果使用可选的 `using` 从句来指定一个或多个命名空间, 那么 `visit` 就会在那么命名空间中寻找指令, 和先前列表中指定的命名空间都获得优先级。如果指定 `using` 从句, 对最后一个未完成的 `visit` 调用的用 `using` 从句指定命名空间的命名空间或序列被重用了。如果没有这样挂起的 `visit` 调用, 那么当前的命名空间就被使用。比如, 如果你执行这个模板:

```
<#import "n1.ftl" as n1>
<#import "n2.ftl" as n2>

<#-- This will call n2.x (because there is no n1.x): -->
<#visit nodeWithNameX using [n1, n2]>

<#-- This will call the x of the current namespace: -->
<#visit nodeWithNameX>

<#macro x>
  Simply x
</#macro>
```

这是 `n1.ftl`:

```
<#macro y>
  n1.y
</#macro>
```

这是 `n2.ftl`:

```

<#macro x>
  n2.x
  <!-- This will call n1.y, because it inherits the "using [n1,
n2]" from the pending visit call: -->
  <#visit nodeWithNameY>
  <!-- This will call n2.y: -->
  <#visit nodeWithNameY using .namespace>
</#macro>

<#macro y>
  n2.y
</#macro>

```

这会打印:

```

n2.x
n1.y
n2.y

Simply x

```

如果 `visit` 既没有在和之前描述规则的名称扣除相同名字的 FTL 命名空间发现自定义指令，那么它会尝试用名称 `@node_type` 来查找，又如果节点不支持节点类型属性（也就是 `node?node_type` 返回未定义变量），那么使用名称 `@default`。对于查找来说，它使用和之前描述相同的机制。如果仍然没有找到处理节点的自定义指令，那么 `visit` 停止模板执行，并抛出错误。一些 XML 特定的节点类型在这方面有特殊的处理；参考：XML 处理指南/声明的 XML 处理/详细内容部分。示例：

```

<!-- Assume that nodeWithNameX?node_name is "x" -->
<#visit nodeWithNameX>

<!-- Assume that nodeWithNameY?node_type is "foo" -->
<#visit nodeWithNameY>

<#macro x>
  Handling node x
</#macro>

<#macro @foo>
  There was no specific handler for node ${node?node_name}
</#macro>

```

这会打印:

```

Handling node x

There was no specific handler for node y

```

### 2.22.2.2 recurse 指令

`<#recurse>`指令是真正纯语义上的指令。它访问节点的所有子节点（而没有节点本身）。所以来写：

```
<#recurse someNode using someLib>
```

和这个是相等的：

```
<#list someNode?children as child><#visit child using someLib></#list>
```

而目标节点在 `recurse` 指令中是可选的。如果目标节点没有指定，那就仅仅使用 `.node`。因此，`<#recurse>`这个精炼的指令和下面这个是相同的：

```
<#list .node?children as child><#visit child></#list>
```

对于熟悉 XSLT 的用户的评论，`<#recurse>`是和 XSLT 中 `<xsl:apply-templates/>`指令相当类似的。

### 2.22.2.3 fallback 指令

正如前面所学的，在 `visit` 指令的文档中，自定义指令控制的节点也许在多个 FTL 命名空间中被搜索。`fallback` 指令可以被用在自定义指令中被调用处理节点。它指挥 FreeMarker 在更多的命名空间（也就是，在当前调用列表中自定义指令命名空间之后的命名空间）中来继续搜索自定义指令。如果节点处理器被发现，那么就被调用，否则 `fallback` 不会做任何事情。

这个指令的典型用法是在处理程序库之上写定制层，有时传递控制到定制的库中：

```
<#import "/lib/docbook.ftl" as docbook>
<#--
  We use the docbook library, but we override some handlers
  in this namespace.
-->
<#visit document using [.namespace, docbook]>
<#--
  Override the "programlisting" handler, but only in the case
  if its "role" attribute is "java"
-->
<#macro programlisting>
  <#if .node.@role[0]!"" == "java">
    <#-- Do something special here... -->
    ...
  <#else>
    <#-- Just use the original (overridden) handler -->
    <#fallback>
  </#if>
</#macro>
```

## 第三章 特殊变量参考文档

特殊变量是由 FreeMarker 引擎自己定义的变量。要访问它们，你可以使用 `.variable_name` 语法。比如，你不能仅仅写 `version`，而必须写 `.version`。

支持的特殊变量有：

- `data_model`：你可以使用来直接访问数据模型的哈希表。也就是，你使用 `global` 指令定义在这里不可见的变量。
- `error`（从 FreeMarker 2.3.1 版本开始可用）：这个变量在 `recover` 指令体中可以访问，它存储了我们恢复错的错误信息。
- `globals`：你可以使用来访问全局可访问的变量的哈希表：数据模型和由 `global` 指令创建的变量。注意用 `assign` 或 `macro` 创建的变量不是全局的。因此当你使用 `globals` 时你不能隐藏变量。
- `language`：返回当前本地设置的语言部分的值。比如 `.locale` 是 `en_US`，那么 `.lang` 是 `en`。
- `locale`：返回当前本地设置的值。这是一个字符串，比如 `en_US`。要获取关于本地化字符串值的更多内容，请参考 `setting` 指令。
- `locales`：你可以访问本地化变量的哈希表（由 `local` 指令创建的变量，还有宏的参数）。
- `main`：你可以用来访问主命名空间的哈希表。注意像数据模型中的全局变量通过这个哈希表是不可见的。
- `namespace`：你可以用来访问当前命名空间的哈希表。注意像数据模型中的全局变量通过这个哈希表是不可见的。
- `node`（由于历史原因重命名为 `current_node`）：你可以用访问者模式（也就是用 `visit`, `recurse` 等指令）处理的当前节点。而且，当你使用 FreeMarker XML 的 Ant 任务时，它初始存储根节点。
- `output_encoding`（从 FreeMarker 2.3.1 版本开始可用）：返回当前输出字符集的名称。如果框架封装 FreeMarker 却没有为 FreeMarker 指定输出字符集时这个特殊变量是不存在的。（程序员可以阅读关于字符集问题的更多内容，在：程序开发指南/其他/字符集问题部分。）
- `template_name`：当前模板的名称（从 FreeMarker 2.3.14 版本开始可用）。
- `url_escaping_charset`（从 FreeMarker 2.3.1 版本开始可用）：如果存在，它存储了应该用于 URL 转义的字符集的名称。如果这个变量不存在就意味着没有人指定 URL 编码应该使用什么样的字符集。这种情况下，`url` 内建函数使用特殊变量 `output_encoding` 指定的字符集来进行 URL 编码。处理机制和它是相同的。（程序员可以阅读关于字符集问题的更多内容，在：程序开发指南/其他/字符集问题部分。）
- `vars`：表达式 `.vars.foo` 返回和表达式 `foo` 相同的变量。出于某些原因你不得不使用方括号语法时这是有用的，因为它只对哈希表子变量有用，所以你需要一个人工的父哈希表。比如，要读取有特殊名称的顶层变量可能会把 FreeMarker 弄糊涂，你可以写 `.vars["A strange name!"]`。或者，使用和变量 `varName` 给定的动态名称访问顶层变量你可以写 `.vars[varName]`。注意这个哈希表由 `.vars` 返回，并不支持 `?keys` 和 `?values`。
- `version`：返回 FreeMarker 版本号的字符串形式，比如 `2.2.8`。这可以用来检

查你的应用程序使用的是哪个版本的 FreeMarker，但是要注意这个特殊变量在 2.3-final 或 2.2.8 版本之前不存在。非最终发行版本号包含缩写形式的“preview”，是“pre”（比如 2.3pre6），或缩写形式的“release candidate”，是“rc”。

## 第四章 FTL 中的保留名称

下面的这些名称不能在方括号语法中被用作顶层变量（比如 `vars["in"]`），因为这是 FTL 中的关键字。

- `true`: 布尔值“true”
- `false`: 布尔值“false”
- `gt`: 比较运算符“大于”
- `gte`: 比较运算符“大于或等于”
- `lt`: 比较运算符“小于”
- `lte`: 比较运算符“小于或等于”
- `as`: 由少数指令使用
- `in`: 由少数指令使用
- `using`: 由少数指令使用

## 第五章 废弃的 FTL 结构

### 5.1 废弃的指令列表

下面这些指令是废弃的，但是仍然可以运行：

- `call`: 使用自定义指令来代替调用
- `comment`: 这是 `<#-- ...-->` 的老式格式。在 `<#comment>` 和 `</#comment>` 之间的任何东西都会被忽略。
- `foreach`: 它是 `list` 指令的代名词，有着轻微不同的参数语法。它的语法结构是 `<#foreach item in sequence>`，和 `<#list sequence as item>` 是相同的。
- `transform`: 使用自定义指令来代替调用

下面这些指令不在可以运行：

- 遗留的 `function`: 起初 `function` 是被用作定义宏，由 `macro` 指令的支持，它就被废弃了。对于 FreeMarker 2.3 版本来说，这个指令由不同的意义而再次引入：它被用来定义方法。

### 5.2 废弃的内建函数列表

下面这些内建函数是被废弃的，但是仍可以运行：

`default`: 由默认值运算符的引入，它被废弃了。`exp1?default(exp2)` 和 `exp1!exp2` 是相同的，`(exp1)?default(exp2)` 和 `(exp1)!exp2` 是相同的。唯一的不同是在 FreeMarker 2.4 版本之前，内建函数 `default` 通常算作是 `exp2`，而默认



值运算符仅仅当默认值真的需要时才算。从 FreeMarker 2.4 版本之后，内建函数 `default` 被改进了，和默认值运算符的行为非常像了。

`exists`: 由空值测试运算符的引入，它被废弃了。`exp1?exists` 和 `exp1??` 是一样的，`(exp1)?exists` 和 `(exp1)??` 也是一样的。

`if_exists`: 由默认值运算符的引入，它被废弃了。`exp1?if_exists` 和 `exp1!` 相似，`(exp1)?if_exists` 和 `(exp1)!` 相似。不同之处在于，用 `if_exists` 的默认值不仅仅同时是空字符串，空序列和空哈希表，而且布尔值 `false` 和不做任何事情的变换，还有忽略所有参数。

`web_safe`: 和 `html` 相同。

## 5.3 老式的 macro 和 call 指令

### 5.3.1 概要

```
<#macro name(argName1, argName2, ... argNameN)>
...
</#macro>
```

```
<#call name(argValue1, argValue2, ... argValueN)>
```

这里：

- `name`: 宏的名称（不是表达式）
- `argName1`, `argName2` 等: 存储参数值局部变量的名称（不是表达式）
- `argValue1`, `argValue2` 等: 表达式，参数的值

### 5.3.2 描述

**注意：**

这是 FreeMarker 2.1 版本的文档中宏还有宏它相关的指令。这仍然可以用，但是已经被废弃了。你也许想阅读 FreeMarker 2.2+ 版本的参考：就是指令参考中的 `macro`, `return` 部分和用户自定义指令部分。

宏是关联名称的模板段。你可以在你的模板中的很多位置使用命名的代码段，所以它可以在重复的任务中帮助你。宏可以有参数，这会在你调用它的时候影响生成的输出。

你可以使用 `macro` 指令来定义宏，之后你可以在整个模板中定义宏。`macro` 指令本身不往输出中写任何东西，它只是用来定义宏。例如这会定义一个称为 `warning` 的宏：

```
<#macro warning(message)>
  <div align=center>
    <table border=1 bgcolor=yellow width="80%"><tr><td
align=center>
      <b>Warning!</b>
      <p>${message}
    </td></tr></table>
  </div>
</#macro>
```

无论何时你使用 `call` 指令来调用这个宏时，宏定义体（在宏的开始标签和结束标签之间的部分）将会被处理。比如这个调用了名为 `warning` 的宏：

```
<div align=center>
  <table border=1 bgcolor=yellow width="80%"><tr><td
align=center>
  <b>Warning!</b>
  <p>Unplug the machine before opening the cover!
</td></tr></table>
</div>
```

作为 `call` 指令参数传递的参数将会在宏定义体中可以作为局部变量来访问。

当你调用一个宏，你必须指定和在宏定义时参数数量相同的参数。比如如果这个宏这么来定义：

```
<#macro test(a, b, c)>Nothing...</#macro>
```

那么这些是合法的调用：

```
<#call test(1, 2, 3)>
<#call test("one", 2 + x, [1234, 2341, 3412, 4123])>
```

如果宏没有定义参数，那么你可以忽略圆括号中的内容：

```
<#macro test>moor</#macro>
<#call test>
```

当你定义宏时，那么它在模板中就是可用的，你也只能在模板中来定义宏。但是你可能想在更多模板中使用相同的宏。这种情况下你可以在公共文件中存储你定义的宏，之后在所有你需要这些宏的模板中包含那个文件。

调用定义在模板下部的宏是不错的（因为宏在解析时间定义，而不是执行时间）。然而，如果宏定义被插入到 `include` 指令中了，它们知道 `FreeMarker` 执行 `include` 指令时才会可用。

你可以用 `return` 指令在 `</#macro>` 标签之前留下宏定义体。

## 5.4 转换指令

### 5.4.1 概要

```
<transform transVar>
  ...
</transform>
or
<transform transVar name1=value1 name2=value2 ...
nameN=valueN>
  ...
</transform>
```

这里：

- `transVar`：要来改变的表达式
- `name1`, `name2`, ...`nameN`：参数的名称。文字值，不是表达式
- `value1`, `value2`, ...`valueN`：算作参数值的表达式

## 5.4.2 描述

**注意：**

这个指令仍然可用，但是已经被废弃了。你也许想阅读用户自定义指令部分来查看它的替代物。

捕捉生成在它体内（也就是开始标签和结束标签之间）的输出，之后让给定的转换物在写入最终的输出之前改变。

示例：

```
<p>A very simple HTML file:
<pre>
<transform html_escape>
<html>
  <body>
    <p>Hello word!
  </body>
</html>
</transform>
</pre>
```

输出为：

```
<p>A very simple HTML file:
<pre>
&lt;html&gt;
  &lt;body&gt;
    &lt;p&gt;Hello word!
  &lt;/body&gt;
&lt;/html&gt;
</pre>
```

一些转换可能需要参数。参数的名称和意义依赖于转换的问题。比如这里我们给出一个名为“`var`”的参数：

```
<!-- This transform stores the output in the variable x,
      rather than sending it to the output -->
<transform capture_output var="x">
some test
</transform>
```

这是程序员在数据模型中放置必要转换的任务。对于可访问转换的名称和用法请问程序

员。最初对在 `freemarker.template.utility` 包中的大多数转换来说有共享变量。要获取更多信息，请阅读：程序开发指南/配置/共享变量部分的内容。

## 5.5 老式 FTL 语法

在 FTL 标签中使用 `#` 形式的 FTL 语法已经是不要求（在 2.1 版本之前是不允许的）的了。比如，你可以这样写代码：

```
<html>
<head>
  <title>Welcome!</title>
</head>
<body>
  <h1>Welcome ${user}!</h1>
  <p>We have there animals:
  <ul>
    <list animals as being>
      <li>${being.name} for ${being.price} Euros
    </list>
  </ul>
  <include "common_footer.html">
</body>
</html>
```

而没有 `#` 样式语法的代码对于 HTML 作者来说更加自然，它有很多的缺点，所以最终我们决定废弃它。使用新式语法（又称为“严格的语法”），`#` 是严格要求的。也就是说，像 `<include "common_footer.html">` 这样的东西将会原样出现在输出中，因为它们不被认为是 FTL 标签。注意用户自定义指令使用 `@` 代替 `#`。

然而，为了给用户时间来准备这种改变，在 FreeMarker 2.1 和 2.2 版本中，`#` 的用法是可选的，除非程序员调用 `Configuration` 的 `setStrictSyntaxMode(true)` 在 FreeMarker 配置中开启严格语法模式。事实上，我们把这个强烈建议给程序员。从后续释出版本开始，这个设置将会初始设置为 `true`。而且，如果你在模板文件中想使用严格语法或老式语法，你可以用 `ftl` 指令来指定。

“严格语法”比遗留的 FTL 语法的好处是：

- 由于对于 FTL 来说，所有 `<#...>` 和 `</#...>` 都是保留的。
  - 我们可以引入新的指令而不破坏向后兼容。
  - 我们可以检测你是否创建了一个类型，也就是 `<#inculde ...>` 被视为解析时的错误，而不是被静默地视为简单文本。
  - 对于第三方工具来处理模板（比如高亮语法显示）来说是简单的，特别是因为它们不需要知道新释出版本中被引入的新指令。
  - 模板更易于阅读，因为很容易辨认嵌入在 HTML 或其他标记中的 `<#...>` 标签。
- `<#` 和 `</#` 是合法的 XML（除了在 CDATA 段中），而且其他大多数 SGML 应用中也是合法的，所以它们不能妨碍用在静态文本部分（比如你在生成的 XML 中有 `include` 元素）的标签。

## 5.6 #{...}式的数字插值

已经被废弃了：使用 `number_format` 设置和内建函数 `string` 来代替。对于计算机使用（也就非本地的格式化）的格式化，使用内建函数 `c`（像 `number?c`）。

### 5.6.1 概要

```
#{expression}  
or  
#{expression; format}
```

这里：

- `expression`：可以算作是数字的表达式
- `format`：可选的格式说明符

### 5.6.2 描述

数字插值被用来输出数值。如果表达式不能算成数字，那么计算过程就会以抛出错误而终止。

可选的格式说明符指定了使用 `mminMmax` 语法显示的小数位数的最小和最大值。比如，`m2M5` 表示“最少两位，最多 5 位小数位”。最小值和最大值说明符部分可以被忽略。如果仅指定最小值，那么最大值和最小值相等。如果仅指定最大值，那么最小值是 0。

输出的小数点字符是国际化的（根据当前本地设置），这表示它可能不是一个点。

不像 `${...}`，`#{...}` 忽略 `number_format` 设置。实际上这是向后兼容的一个怪点，但是当你在如 `<a href="quertyDatabase?id=#{id}">` 这些情况打印数字时它可能是有用的，这里你肯定不想分组分隔符或像那些幻想的东西。然而，从 **FreeMarker 2.3.3** 版本开始，而使用内建函数 `c` 来达到这个目的，比如 `<a href="quertyDatabase?id=${id?c}">`。

示例：假设 `x` 是 2.582 而 `y` 是 4：

```
<!-- If the language is US English the output is: -->  
#{x}           <!-- 2.582 -->  
#{y}           <!-- 4 -->  
#{x; M2}       <!-- 2.58 -->  
#{y; M2}       <!-- 4 -->  
#{x; m1}       <!-- 2.6 -->  
#{y; m1}       <!-- 4.0 -->  
#{x; m1M2}     <!-- 2.58 -->  
#{y; m1M2}     <!-- 4.0 -->
```