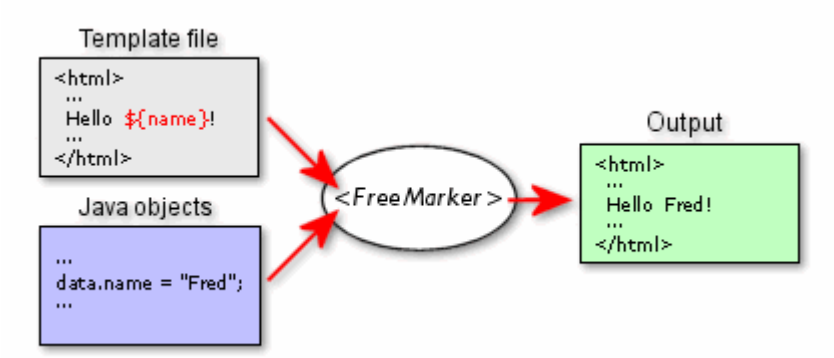
FreeMarker 概述

FreeMarker 是一个模板引擎，一个基于模板生成文本输出的通用工具，使用纯 Java 编写

FreeMarker 被设计用来生成 HTML Web 页面，特别是基于 MVC 模式的应用程序

虽然 FreeMarker 具有一些编程的能力，但通常由 Java 程序准备要显示的数据，由 FreeMarker 生成页面，通过模板显示准备的数据（如下图）



FreeMarker 不是一个 Web 应用框架，而适合作为 Web 应用框架一个组件

FreeMarker 与容器无关，因为它并不知道 HTTP 或 Servlet；FreeMarker 同样可以应用于非 Web 应用程序环境

FreeMarker 更适合作为 Model2 框架（如 Struts）的视图组件，你也可以在模板中使用 JSP 标记库

FreeMarker 是免费的

FreeMarker 特性

1、通用目标

能够生成各种文本：HTML、XML、RTF、Java 源代码等等

易于嵌入到你的产品中：轻量级；不需要 Servlet 环境

插件式模板载入器：可以从任何源载入模板，如本地文件、数据库等等

你可以按你所需生成文本：保存到本地文件；作为 Email 发送；从 Web 应用程序发送它返回给 Web 浏览器

2、强大的模板语言

所有常用的指令：include、if/elseif/else、循环结构

在模板中创建和改变变量

几乎在任何地方都可以使用复杂表达式来指定值

命名的宏，可以具有位置参数和嵌套内容

名字空间有助于建立和维护可重用的宏库，或者将一个大工程分成模块，而不必担心名字冲突

输出转换块：在嵌套模板片段生成输出时，转换 HTML 转义、压缩、语法高亮等等；你可以定义自己的转换

3、通用数据模型

FreeMarker 不是直接反射到 Java 对象，Java 对象通过插件式对象封装，以变量方式在模板中显示
你可以使用抽象（接口）方式表示对象（JavaBean、XML 文档、SQL 查询结果集等等），告诉模板开发者使用方法，使其不受技术细节的打扰

4、为 Web 准备

在模板语言中内建处理典型 Web 相关任务（如 HTML 转义）的结构
能够集成到 Model2 Web 应用框架中作为 JSP 的替代
支持 JSP 标记库
为 MVC 模式设计：分离可视化设计和应用程序逻辑；分离页面设计员和程序员

5、智能的国际化和本地化

字符集智能化（内部使用 UNICODE）
数字格式本地化敏感
日期和时间格式本地化敏感
非 US 字符集可以用作标识（如变量名）
多种不同语言的相同模板

6、强大的 XML 处理能力
<#recurse> 和<#visit>指令（2.3 版本）用于递归遍历 XML 树
在模板中清楚和直觉的访问 XML 对象模型

FreeMarker vs. Velocity

1、概述

Velocity 是一个简单而且更加轻量级的工具，但是它没有达到 FreeMarker 能够做的许多任务，而且它的模板语言不是很强大
我们认为对于大多数应用程序，FreeMarker 比 Velocity 工作更简单，因为：
Ø使用 Velocity，你需要寻找特定工具或各种工作环境来一次次的解决典型的模板创作任务，结果会浪费更多时间
Ø工作环境经常意味着在 Velocity 模板中直接使用 Java 对象的方法，这违反了简单、无编程 HTML 设计的观念
Ø或者将表示任务移到控制器代码中，这违反了 MVC 模式
使用 FreeMarker，可以以 out-of-the-box（如何翻译确切？）的方式实现 Velocity 所能做的

2、特性比较清单

下面是一个使用 FreeMarker 能够实现，而 Velocity 不能实现的不太全面的特性清单：
（1）数字和日期支持
　　可以对任何数字类型进行算术运算和比较，包括精度类型
　　可以比较和显示（格式化）日期/时间值
（2）国际化
　　根据各种内建和定制的数字格式，格式化本地敏感的数字

根据各种内建和定制的日期格式，格式化本地敏感和时区敏感的日期

标识（变量名）可以包含非英语字符，如重音字符、阿拉伯字符、中文字符等

（3）循环处理

可以跳出循环

可以在循环外访问循环体内的控制变量

可以测试是否达到最后一次循环

（4）模板级别的数组处理

可以使用类似[i]语法的索引方式访问数组元素

可以查询数组长度

（5）宏

宏可以有局部变量

可以递归调用宏，同样可以在模板的后面定义要调用的宏

调用宏时，可以按位置或名字的方式传递参数

宏参数可以有缺省值，使得在调用时忽略参数也有效

调用的宏可以有嵌套的体内容（<@myMacro>body</@myMacro>），能够在宏被调用时进行处理

宏是纯变量的，可以基于表达式来执行宏，或者作为参数传递给另一个宏

（6）命名空间

可以对变量使用多命名空间，这对创建宏库很重要，因为这可以避免应用程序中指定的变量和宏库中变量的名字冲突

（7）使用内建的函数/操作符维护 Java 无关的 string、list 和 map

可以将字符串转换成大/小写、首字符大/小写，对 HTML、XML 或 RTF 进行转义处理，substring、split、查询字符串长度、find/replace 子串等等

通过索引访问 list 元素，获得子 list，合并 list，查询 list 长度，对 list 排序

通过 key 变量访问 map 元素，检查 map 是否为空，获得 key 或值的 list

（8）揭示模板中的错误

当访问一个未定义的变量，FreeMarker 不会沉默；你可以配置 FreeMarker 来停止 render 模板显示错误信息，或者跳过错误部分；无论哪种，FreeMarker 会记录问题（日志）

在写错指令名时，FreeMarker 会抛出异常

（9）高级 render 控制

可以使用一组标记来封装模板的一块区域，以便在块区中所有要修改的地方应用 HTML 或 XML 转义（或其它使用 FreeMarker 表达式表示的转换）

FreeMarker 有转换器，它们是模板的一块区域，在 render 时，通过转换过滤；内建的转换器包括空白字符压缩、HTML 和 XML 转义；你可以实现自己的转换器；当然转换器可以嵌套

可以使用 flush 指令显式的 flush 输出

可以使用 stop 指令停止 render

（10）文字

除了通常的字符串、数字和布尔值文字，也可以在模板中定义 list 和 map 文字

支持所有的 Java 转义文字：\b、\t、\n、\f、\r、\"、\'、\\，也支持\xXXXX 使用 UNICODE 指定字符

（11）高级空白字符移除

FreeMarker 坚持移除各行只包含不输出 FreeMarker 标记的空白字符

对于明显要整修掉不需要的空白字符的指令来说，空白字符是个大问题

（12）集成其它技术

可以在模板中使用 JSP 标记库

可以直接在 Python 工程中使用

（13）强大的 XML 转换能力

在 2.3 版本中，FreeMarker 具有强大的新 XML 转换能力，使得替代 XSLT 成为可能

Velocity 在这方面是无法真正竞争的，除非改进核心引擎，如支持宏库映射到名字空间，宏中支持局部变量

（14）高级模板元程序

可以捕获输出的任何部分到 context 变量中

可以解释任何 context 变量，如果它是一个模板定义

上述两者的结合使用

FreeMarker 设计指南(1)

1、快速入门

（1）模板 + 数据模型 = 输出

- FreeMarker 基于设计者和程序员是具有不同专业技能的不同个体的观念
- 他们是分工劳动的：设计者专注于表示——创建 HTML 文件、图片、Web 页面的其它可视化方面；程序员创建系统，生成设计页面要显示的数据
- 经常会遇到的问题是：在 Web 页面（或其它类型的文档）中显示的信息在设计页面时是无效的，是基于动态数据的
- 在这里，你可以在 HTML（或其它要输出的文本）中加入一些特定指令，FreeMarker 会在输出页面给最终用户时，用适当的数据替代这些代码
- 下面是一个例子：

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

- 这个例子是在简单的 HTML 中加入了一些由**${…}**包围的特定代码，这些特定代码是 FreeMarker 的指令，而包含 FreeMarker 的指令的文件就称为模板（Template）
- 至于 user、latestProduct.url 和 latestProduct.name 来自于数据模型（data model）

- 数据模型由程序员编程来创建，向模板提供变化的信息，这些信息来自于数据库、文件，甚至于在程序中直接生成
- 模板设计者不关心数据从那儿来，只知道使用已经建立的数据模型
- 下面是一个可能的数据模型：

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

- 数据模型类似于计算机的文件系统，latestProduct 可以看作是目录，而 user、url 和 name 看作是文件，url 和 name 文件位于 latestProduct 目录中（这只是一个比喻，实际并不存在）
- 当 FreeMarker 将上面的数据模型合并到模板中，就创建了下面的输出：

```
<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>
```

（2）数据模型

- 典型的数据模型是树型结构，可以任意复杂和深层次，如下面的例子：

```
(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"
```

- 类似于目录的变量称为 hashes，包含保存下级变量的唯一的查询名字
- 类似于文件的变量称为 scalars，保存单值
- scalars 保存的值有两种类型：字符串（用引号括起，可以是单引号或双引号）和数字（不要用引号将数字括起，这会作为字符串处理）
- 对 scalars 的访问从 root 开始，各部分用“."分隔，如 animals.mouse.price
- 另外一种变量是 sequences，和 hashes 类似，只是不使用变量名字，而使用数字索引，如下面的例子：

```
(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"
```

- 这种对 scalars 的访问使用索引，如 animals[0].name

（3）模板

- 在 FreeMarker 模板中可以包括下面三种特定部分：
  - ${…}：称为 interpolations，FreeMarker 会在输出时用实际值进行替代
  - FTL 标记（FreeMarker 模板语言标记）：类似于 HTML 标记，为了与 HTML 标记区分，用#开始（有些以@开始，在后面叙述）
  - 注释：包含在<#--和-->（而不是<!--和-->）之间
- 下面是一些使用指令的例子：
  - if 指令

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

  - list 指令

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

输出为：

```
<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>
```

- include 指令

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

- 一起使用指令

```
<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>
```

2、数据模型

（1）基础

- 在快速入门中介绍了在模板中使用的三种基本对象类型：scalars、hashes 和 sequences，其实还可以有其它更多的能力：
  - scalars：存储单值
  - hashes：充当其它对象的容器，每个都关联一个唯一的查询名字
  - sequences：充当其它对象的容器，按次序访问
  - 方法：通过传递的参数进行计算，以新对象返回结果

- 用户自定义 FTL 标记：宏和变换器

- 通常每个变量只具有上述的一种能力，但一个变量可以具有多个上述能力，如下面的例子：

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

- mouse 既是 scalars 又是 hashes，将上面的数据模型合并到下面的模板：

```
${mouse}       <#-- use mouse as scalar -->
${mouse.age}   <#-- use mouse as hash -->
${mouse.color} <#-- use mouse as hash -->
```

- 输出结果是：

```
Yerri
12
brown
```

（2）Scalar 变量

- Scalar 变量存储单值，可以是：
  - 字符串：简单文本，在模板中使用引号（单引号或双引号）括起
  - 数字：在模板中直接使用数字值
  - 日期：存储日期/时间相关的数据，可以是日期、时间或日期-时间（Timestamp）；通常情况，日期值由程序员加到数据模型中，设计者只需要显示它们
  - 布尔值：true 或 false，通常在<#if …>标记中使用

（3）hashes 、sequences 和集合

- 有些变量不包含任何可显示的内容，而是作为容器包含其它变量，者有两种类型：
  - hashes：具有一个唯一的查询名字和它包含的每个变量相关联
  - sequences：使用数字和它包含的每个变量相关联，索引值从 0 开始
- 集合变量通常类似 sequences，除非无法访问它的大小和不能使用索引来获得它的子变量；集合可以看作只能由<#list …>指令使用的受限 sequences

（4）方法

- 方法变量通常是基于给出的参数计算值
- 下面的例子假设程序员已经将方法变量 avg 放到数据模型中，用来计算数字平均值：

```
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 python and elephant is: ${avg(animals.python.price,
 animals.elephant.price)}
```

（5）宏和变换器

- 宏和变换器变量是用户自定义指令（自定义 FTL 标记），会在后面讲述这些高级特性

（6）节点

- 节点变量表示为树型结构中的一个节点，通常在 XML 处理中使用，会在后面的专门章节中讲述

3、模板

（1）整体结构

- 模板使用 FTL（FreeMarker 模板语言）编写，是下面各部分的一个组合：
  - 文本：直接输出
  - Interpolation：由${和}，或#{和}来限定，计算值替代输出
  - FTL 标记：FreeMarker 指令，和 HTML 标记类似，名字前加#予以区分，不会输出
  - 注释：由<#--和-->限定，不会输出
- 下面是以一个具体模板例子：

```
<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>
```

- [BR]是用于换行的特殊字符序列
- 注意事项：

- FTL 区分大小写，所以 list 是正确的 FTL 指令，而 List 不是；${name} 和${NAME}是不同的
- Interpolation 只能在文本中使用
- FTL 标记不能位于另一个 FTL 标记内部，例如：

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

- 注释可以位于 FTL 标记和 Interpolation 内部，如下面的例子：

```
<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]

...
```

- 多余的空白字符会在模板输出时移除

（2）指令

- 在 FreeMarker 中，使用 FTL 标记引用指令
- 有三种 FTL 标记，这和 HTML 标记是类似的：
  - 开始标记：<#directivename parameters>
  - 结束标记：</#directivename>
  - 空内容指令标记：<#directivename parameters/>
- 有两种类型的指令：预定义指令和用户定义指令
- 用户定义指令要使用@替换#，如<@mydirective>...</@mydirective>（会在后面讲述）
- FTL 标记不能够交叉，而应该正确的嵌套，如下面的代码是错误的：

```
<ul>

<#list animals as being>

  <li>${being.name} for ${being.price} Euros

  <#if use = "Big Joe">

     (except for you)

</#list>

</#if> <#-- WRONG! -->

</ul>
```

- 如果使用不存在的指令，FreeMarker 不会使用模板输出，而是产生一个错误消息
- FreeMarker 会忽略 FTL 标记中的空白字符，如下面的例子：

```
<#list[BR]

  animals       as[BR]

     being[BR]
```

```
>[BR]

${being.name} for ${being.price} Euros[BR]

</#list    >
```

- 但是，<、</和指令之间不允许有空白字符

（3）表达式

- 直接指定值
  - 字符串
    - 使用单引号或双引号限定
    - 如果包含特殊字符需要转义，如下面的例子：

```
${"It's \"quoted\" and

this is a backslash: \\"}



${'It\'s "quoted" and

this is a backslash: \\'}
```

输出结果是：

```
It's "quoted" and

this is a backslash: \



It's "quoted" and

this is a backslash: \
```

- 下面是支持的转义序列：

转义序列	含义
\"	双引号(u0022)
\'	单引号(u0027)
\\	反斜杠(u005C)
\n	换行(u000A)
\r	Return (u000D)
\t	Tab (u0009)
\b	Backspace (u0008)
\f	Form feed (u000C)
\l	<
\g	>
\a	&

转义序列	含义
\{	{
\x*Code*	4 位 16 进制 Unicode 代码

- 有一类特殊的字符串称为 raw 字符串，被认为是纯文本，其中的¥和{等不具有特殊含义，该类字符串在引号前面加 r，下面是一个例子：

```
${r"${foo}"}

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

输出的结果是：

```
${foo}

C:\foo\bar
```

- 数字
  - 直接输入，不需要引号
  - 精度数字使用“.”分隔，不能使用分组符号
  - 目前版本不支持科学计数法，所以“1E3”是错误的
  - 不能省略小数点前面的 0，所以“.5”是错误的
  - 数字 8、+8、08 和 8.00 都是相同的
- 布尔值
  - true 和 false，不使用引号
- 序列
  - 由逗号分隔的子变量列表，由方括号限定，下面是一个例子：

```
<#list ["winter", "spring", "summer", "autumn"] as x>

${x}

</#list>
```

输出的结果是：

```
winter

spring

summer

autumn
```

  - 列表的项目是表达式，所以可以有下面的例子：

```
[2 + 2, [1, 2, 3, 4], "whatnot"]
```

  - 可以使用数字范围定义数字序列，例如 2..5 等同于[2, 3, 4, 5]，但是更有效率，注意数字范围没有方括号
  - 可以定义反递增的数字范围，如 5..2
- 散列（hash）
  - 由逗号分隔的键/值列表，由大括号限定，键和值之间用冒号分隔，下面是一个例子：

```
{"name":"green mouse", "price":150}
```

- 键和值都是表达式，但是键必须是字符串

- 获取变量
  - 顶层变量： ${variable}，变量名只能是字母、数字、下划线、$、@和#的组合，且不能以数字开头
  - 从散列中获取数据
    - 可以使用点语法或方括号语法，假设有下面的数据模型：

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

下面都是等价的：

```
book.author.name

book["author"].name

book.author.["name"]

book["author"]["name"]
```

- 使用点语法，变量名字有顶层变量一样的限制，但方括号语法没有该限制，因为名字是任意表达式的结果

  - 从序列获得数据：和散列的方括号语法语法一样，只是方括号中的表达式值必须是数字；注意：第一个项目的索引是 0
  - 序列片断：使用[startIndex..endIndex]语法，从序列中获得序列片断（也是序列）；startIndex 和 endIndex 是结果为数字的表达式
  - 特殊变量：FreeMarker 内定义变量，使用.variablename 语法访问

- 字符串操作
  - Interpolation（或连接操作）
    - 可以使用${..}（或#{..}）在文本部分插入表达式的值，例如：

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

```
${"${user}${user}${user}${user}"}
```

- 可以使用+操作符获得同样的结果

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

${user + user + user + user}
```

- ${..}只能用于文本部分，下面的代码是错误的：

```
<#if ${isBig}>Wow!</#if>

<#if "${isBig}">Wow!</#if>
```

应该写成：

```
<#if isBig>Wow!</#if>
```

- 子串
  - 例子（假设 user 的值为“Big Joe”）：

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

${user[1..4]}
```

结果是（注意第一个字符的索引是 0）：

```
BJ

ig J
```

- 序列操作
  - 连接操作：和字符串一样，使用+，下面是一个例子：

```
<#list ["Joe", "Fred"] + ["Julia", "Kate"] as user>

- ${user}

</#list>
```

输出结果是：

```
- Joe

- Fred

- Julia

- Kate
```

- 散列操作
  - 连接操作：和字符串一样，使用+，如果具有相同的 key，右边的值替代左边的值，例如：

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

- Joe is ${ages.Joe}

- Fred is ${ages.Fred}

- Julia is ${ages.Julia}
```

输出结果是：

```
- Joe is 30

- Fred is 25

- Julia is 18
```

- 算术运算
  - 十、一、×、/、%，下面是一个例子：

```
${x * x - 100}

${x / 2}

${12 % 10}
```

输出结果是（假设 x 为 5）：

```
-75

2.5

2
```

  - 操作符两边必须是数字，因此下面的代码是错误的：

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

  - 使用+操作符时，如果一边是数字，一边是字符串，就会自动将数字转换为字符串，例如：

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

输出结果是：

```
35
```

  - 使用内建的 int（后面讲述）获得整数部分，例如：

```
${(x/2)?int}

${1.1?int}

${1.999?int}

${-1.1?int}

${-1.999?int}
```

输出结果是（假设 x 为 5）：

```
2

1

1

-1

-1
```

- 比较操作符
  - 使用=（或==，完全相等）测试两个值是否相等，使用!= 测试两个值是否不相等

- =和!=两边必须是相同类型的值，否则会产生错误，例如<#if 1 = "1">会引起错误
- Freemarker 是精确比较，所以对"x"、"x "和"X"是不相等的
- 对数字和日期可以使用<、<=、>和>=，但不能用于字符串
- 由于 Freemarker 会将>解释成 FTL 标记的结束字符，所以对于>和>=可以使用括号来避免这种情况，例如<#if (x > y)>
- 另一种替代的方法是，使用 lt、lte、gt 和 gte 来替代<、<=、>和>=

- 逻辑操作符
  - &&（and）、||（or）、!（not），只能用于布尔值，否则会产生错误
  - 例子：

```
<#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>
```

- 内建函数
  - 内建函数的用法类似访问散列的子变量，只是使用“?”替代“.”，下面列出常用的一些函数
  - 字符串使用的：
    - html：对字符串进行 HTML 编码
    - cap_first：使字符串第一个字母大写
    - lower_case：将字符串转换成小写
    - upper_case：将字符串转换成大写
    - trim：去掉字符串前后的空白字符
  - 序列使用的：
    - size：获得序列中元素的数目
  - 数字使用的：
    - int：取得数字的整数部分（如-1.9?int 的结果是-1）
  - 例子（假设 test 保存字符串"Tom & Jerry"）：

```
${test?html}

${test?upper_case?html}
```

输出结果是：

```
Tom &amp;amp; Jerry

TOM &amp;amp; JERRY
```

- 操作符优先顺序

操作符组	操作符
后缀	[subvarName] [subStringRange] . (methodParams)

操作符组	操作符
一元	+expr、-expr、!
内建	?
乘法	*、 / 、%
加法	+、-
关系	<、>、<=、>=（lt、lte、gt、gte）
相等	==（=）、!=
逻辑 and	&&
逻辑 or	||
数字范围	..

（4）Interpolation

- Interpolation 有两种类型：
  - 通用 Interpolation：${expr}
  - 数字 Interpolation：#{expr}或#{expr; format}
- 注意：Interpolation 只能用于文本部分
- 通用 Interpolation
  - 插入字符串值：直接输出表达式结果
  - 插入数字值：根据缺省格式（由#setting 指令设置）将表达式结果转换成文本输出；可以使用内建函数 string 格式化单个 Interpolation，下面是一个例子：

```
<#setting number_format="currency"/>

<#assign answer=42/>

${answer}

${answer?string}  <#-- the same as ${answer} -->

${answer?string.number}

${answer?string.currency}

${answer?string.percent}
```

输出结果是：

```
$42.00

$42.00

42

$42.00

4,200%
```

- 插入日期值：根据缺省格式（由#setting 指令设置）将表达式结果转换成文本输出；可以使用内建函数 string 格式化单个 Interpolation，下面是一个使用格式模式的例子：

```
${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')'")}
```

输出的结果类似下面的格式：

```
2003-04-08 21:24:44 Pacific Daylight Time

Tue, Apr 8, '03

Tuesday, April 08, 2003, 09:24:44 PM (PDT)
```

- 插入布尔值：根据缺省格式（由#setting 指令设置）将表达式结果转换成文本输出；可以使用内建函数 string 格式化单个 Interpolation，下面是一个例子：

```
<#assign foo=true/>

${foo?string("yes", "no")}
```

输出结果是：

```
yes
```

- 数字 Interpolation 的#{expr; format}形式可以用来格式化数字，format 可以是：
  - mX：小数部分最小 X 位
  - MX：小数部分最大 X 位
  - 例子：

```
          <#-- If the language is US English the output is: -->

<#assign x=2.582/>

<#assign 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  -->
```

4、杂项

（1）用户定义指令

- 宏和变换器变量是两种不同类型的用户定义指令，它们之间的区别是宏是在模板中使用 macro 指令定义，而变换器是在模板外由程序定义，这里只介绍宏
- 基本用法
  - 宏是和某个变量关联的模板片断，以便在模板中通过用户定义指令使用该变量，下面是一个例子：

```
<#macro greet>

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

</#macro>
```

  - 作为用户定义指令使用宏变量时，使用@替代 FTL 标记中的#

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

  - 如果没有体内容，也可以使用：

```
<@greet/>
```

- 参数
  - 在 macro 指令中可以在宏变量之后定义参数，如：

```
<#macro greet person>

  <font size="+2">Hello ${person}!</font>

</#macro>
```

  - 可以这样使用这个宏变量：

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

  输出结果是：

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

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

  - 宏的参数是 FTL 表达式，所以下面的代码具有不同的意思：

```
<@greet person=Fred/>
```

  - 这意味着将 Fred 变量的值传给 person 参数，该值不仅是字符串，还可以是其它类型，甚至是复杂的表达式
  - 宏可以有多参数，下面是一个例子：

```
<#macro greet person color>

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

</#macro>
```

  - 可以这样使用该宏变量：

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

  - 其中参数的次序是无关的，因此下面是等价的：

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

- 只能使用在 macro 指令中定义的参数，并且对所有参数赋值，所以下面的代码是错误的：

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

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

- 可以在定义参数时指定缺省值，如：

```
<#macro greet person color="black">

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

</#macro>
```

- 这样<@greet person="Fred"/>就正确了
- 宏的参数是局部变量，只能在宏定义中有效

● 嵌套内容

- 用户定义指令可以有嵌套内容，使用<#nested>指令执行指令开始和结束标记之间的模板片断
- 例子：

```
<#macro border>

  <table border=4 cellspacing=0 cellpadding=4><tr><td>

    <#nested>

  </tr></td></table>

</#macro>
```

这样使用该宏变量：

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

输出结果：

```
  <table border=4 cellspacing=0 cellpadding=4><tr><td>

    The bordered text

  </tr></td></table>
```

- <#nested>指令可以被多次调用，例如：

```
<#macro do_thrice>

  <#nested>

  <#nested>

  <#nested>

</#macro>

<@do_thrice>

  Anything.
```

```
</@do_thrice>
```

输出结果：

```
Anything.

Anything.

Anything.
```

- 嵌套内容可以是有效的 FTL，下面是一个有些复杂的例子：

```
<@border>

  <ul>

  <@do_thrice>

    <li><@greet person="Joe"/>

  </@do_thrice>

  </ul>

</@border>
```

输出结果：

```
  <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>
```

- 宏定义中的局部变量对嵌套内容是不可见的，例如：

```
<#macro repeat count>

  <#local y = "test">

  <#list 1..count as x>

    ${y} ${count}/${x}: <#nested>

  </#list>
```

```
</#macro>

<@repeat count=3>${y?default("?")} ${x?default("?")}
${count?default("?")}</@repeat>
```

输出结果：

```
  test 3/1: ? ? ?

  test 3/2: ? ? ?

  test 3/3: ? ? ?
```

- 在宏定义中使用循环变量
  - 用户定义指令可以有循环变量，通常用于重复嵌套内容，基本用法是：作为 nested 指令的参数传递循环变量的实际值，而在调用用户定义指令时，在<@…>开始标记的参数后面指定循环变量的名字
  - 例子：

```
<#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>
```

输出结果：

```
1. 0.5

2. 1

3. 1.5

4. 2 Last!
```

  - 指定的循环变量的数目和用户定义指令开始标记指定的不同不会有问题
    - 调用时少指定循环变量，则多指定的值不可见
    - 调用时多指定循环变量，多余的循环变量不会被创建

（2）在模板中定义变量

- 在模板中定义的变量有三种类型：
  - plain 变量：可以在模板的任何地方访问，包括使用 include 指令插入的模板，使用 assign 指令创建和替换
  - 局部变量：在宏定义体中有效，使用 local 指令创建和替换

- 循环变量：只能存在于指令的嵌套内容，由指令（如 list）自动创建; 宏的参数是局部变量，而不是循环变量

- 局部变量隐藏（而不是覆盖）同名的 plain 变量；循环变量隐藏同名的局部变量和 plain 变量，下面是一个例子：

```
<#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>
```

输出结果：

```
1. plain
  2. plain
  3. local
    4. loop
  5. local
6. plain
```

```
    7. loop
    8. loop
  9. plain2
```

- 内部循环变量隐藏同名的外部循环变量，如：

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

输出结果：

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

- 模板中的变量会隐藏（而不是覆盖）数据模型中同名变量，如果需要访问数据模型中的同名变量，使用特殊变量 global，下面的例子假设数据模型中的 user 的值是 Big Joe：

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

（3）名字空间

- 通常情况，只使用一个名字空间，称为主名字空间
- 为了创建可重用的宏、变换器或其它变量的集合（通常称库），必须使用多名字空间，其目的是防止同名冲突
- 创建库
  - 下面是一个创建库的例子（假设保存在 lib/my_test.ftl 中）：

```
<#macro copyright date>

  <p>Copyright (C) ${date} Julia Smith. All rights reserved.

  <br>Email: ${mail}</p>

</#macro>

<#assign mail = "jsmith@acme.com">
```

- 使用 import 指令导入库到模板中，Freemarker 会为导入的库创建新的名字空间，并可以通过 import 指令中指定的散列变量访问库中的变量：

```
<#import "/lib/my_test.ftl" as my>

<#assign mail="fred@acme.com">

<@my.copyright date="1999-2002"/>

${my.mail}

${mail}
```

输出结果：

```
  <p>Copyright (C) 1999-2002 Julia Smith. All rights reserved.

  <br>Email: jsmith@acme.com</p>

jsmith@acme.com

fred@acme.com
```

可以看到例子中使用的两个同名变量并没有冲突，因为它们位于不同的名字空间

- 可以使用 assign 指令在导入的名字空间中创建或替代变量，下面是一个例子：

```
<#import "/lib/my_test.ftl" as my>

${my.mail}

<#assign mail="jsmith@other.com" in my>

${my.mail}
```

- 输出结果：

```
jsmith@acme.com

jsmith@other.com
```

- 数据模型中的变量任何地方都可见，也包括不同的名字空间，下面是修改的库：

```
<#macro copyright date>

  <p>Copyright (C) ${date} ${user}. All rights reserved.</p>

</#macro>
```

```
<#assign mail = "${user}@acme.com">
```

- 假设数据模型中的 user 变量的值是 Fred，则下面的代码：

```
<#import "/lib/my_test.ftl" as my>

<@my.copyright date="1999-2002"/>

${my.mail}
```

- 输出结果：

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

Fred@acme.com
```