

FreeMarker2.3.10

—Programmer's Guide (中文版)

前言.....	3
一、快速入门.....	4
1.1、创建配置实例.....	4
1.2、创建数据模型（Data Model）.....	4
1.3、获取模版（template）.....	5
1.4、把模版与数据模型合并.....	6
1.5、完整的代码.....	6
二、数据模型（Data Model）.....	7
2.1、基础类型.....	7
2.2、Scalar 类型（单值对应的 Data Model）.....	8
2.3、容器类型.....	8
2.4、方法变量.....	9
2.5、转换器变量.....	10
2.6、节点变量.....	12
2.7、对象包裹.....	12
三、配置（Configuration）.....	15
3.1、基础.....	15
3.2、共享变量.....	15
3.3、配置参数.....	16
3.4、加载模板.....	17
3.5、异常处理.....	19
四、其它说明.....	20
4.1、变量.....	20
4.2、字符编码.....	20
4.3、多线程.....	21
4.5、Bean 包裹.....	21

前言

FreeMarker 官方参考文档总共有四份，它们分别是

- [Designer's Guide](#) (网上已有翻译，主要从 FreeMarker 的概念上介绍)
- [Programmer's Guide](#) (本文档所以翻译的部分，主要从框架的设计方面介绍)
- [XML Processing Guide](#) (对 XML 数据模型处理的介绍)
- [Reference](#) (FreeMarker 的参考文档，语言使用介绍)

中文翻译之所以选择 [Programmer's Guide](#) 是因为个人觉得该部分对如何实现 FreeMarker 进行了比较深入的阐述。有助于读者很好的了解其运作机制，以及去理解其他模板引擎（如 Velocity）的工作机理。

注：由于原文档部分内容直译可能难于被读者理解，所以有些地方采用意译为主，因此在翻译用词上难免可能会有出入，大家对翻译的内容有任何意见都可以给我直接发邮件告知 motomagice@yahoo.com.cn

一、快速入门

1.1、创建配置实例

首先你需要创建一个 `Configuration` (`freemarker.template.Configuration`) 的实例，设置其中的某些属性。`Configuration` 是存放 FreeMarker 的 Application 级别配置信息的一个重要地方。同时，它还负责创建及预解析模版(template)。

在应用系统的生命周期中 (servlet) 你只需要初始化创建一次 `Configure` 实例 (因为它保存的是全局配置信息)

```
Configuration cfg = new Configuration();
// 指定一个加载模版的数据源
// 这里我设置模版的根目录
cfg.setDirectoryForTemplateLoading(new File("/where/you/store/templates"));
// 指定模版如何查看数据模型.这个话题是高级主题...
// 你目前只需要知道这么用就可以了:
cfg.setObjectWrapper(new DefaultObjectWrapper());
```

你仅仅只需要这么一个配置就可以了。注意：如果你的系统中有多个独立的模块都要使用 FreeMarker 那么你就需要多个 `Configuration` 实例 (也就是说你每一个组建都需要一个私有的配置)

1.2、创建数据模型 (Data Model)

如果配置简单的数据模型，你可能只需要 `java.lang` 和 `java.util` 以及一些 `Java Beans` 来构建 FreeMarker 的数据库模型。

- 字符串使用 `java.lang.String` .
- 数字使用 `java.lang.Number`
- 布尔使用 `java.lang.Boolean`
- 数组以及序列使用 `java.util.List`
- Hashes (一种容器类型可以包含的多种基本类型) 使用 `java.util.Map`
- 对于 hashes 类型的数据模型你也可以使用 `bean` 对象来存放, 而数据项必须和 `bean` 对象的属性项名称要一致。例如 `product` 有一个 `price` 属性, 那么 FreeMarker 则可以通过 `product.price` 来获取相应的值。

让我们来看一看怎么创建下面这个数据模型。

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

以下是创建该模型的 java 代码:

```
//创建一个 hash 类型作为数据模型的 root
Map root = new HashMap();
//把字符串 user 放置到 root 中
root.put("user", "Big Joe");
//再创建一个 hash 类型名字叫做 latestProduct
Map latest = new HashMap();
//同样把它放置到 root 中
root.put("latestProduct", latest);
//放置 url 和 name 属性到 latest
latest.put("url", "products/greenmouse.html");
latest.put("name", "green mouse");
```

对于数据模型中的 `latestProduct` 属性来说,你也可以使用普通 `Java Bean` 存储,但是 `bean` 必须要拥有 `url` 和 `name` 两个属性(也就是它有 `getURL()` 这些方法),对于 `FreeMarker` 模版来说两种定义数据库模型(`Map`, `bean` 方式)的方式实质上是一样的。

1.3、获取模版 (template)

模版对象一般就是指代 `freemarker.template.Template` 实例。通常你可以从一个 `Configuration` 实例中获取模版实例。你可以通过 `getTemplate` 这个方法来获得。

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

当你调用如上代码的时候,它将会创建一个与 `test.ftl` 对应的模版实例,也就是读取 `/where/you/store/templates/test.ftl` 路径下的文件然后解析(编译)。模版对象一般存储的是经过解析过的模版内容。

如果 `Configuration` 实例配置缓存 `Template` 实例策略的话,那么当你再次要获取 `test.ftl` 的时候,那么就可以从缓存中获得,而不会重新生成一个新的模版实例。

1.4、把模版与数据模型合并

我们都知道，**数据模型+模版=输出**，而我们一旦拥有数据模型（root）和一个模版（template）那么我们就可以把他们合并获得输出。

以上这个过程是通过 `template` 类的 `process` 方法来实现的，该方法需要两个参数一个是表示数据模型的 `root` 一个表示输出的 `Writer`。它把解析过的文件输出到 `Writer` 上。简单起见，我把输出指定到了控制台（标准输出）

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

由于 `Template` 实例是无状态的，一旦你获取了一个模版实例，那么你可以与不同的数据库模型多次合并。另外 `test.ftl` 文件是在 `Template` 实例被创建的时候就读取的，而不是等到调用 `process` 的时候才读取。

1.5、完整的代码

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

public class Test {

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

        /* 一般在应用的整个生命周期中你仅需要执行一下代码一次*/

        /* 创建一个合适的 configuration */
        Configuration cfg = new Configuration();
        cfg.setDirectoryForTemplateLoading(new
            File("/where/you/store/templates"));
        cfg.setObjectWrapper(new DefaultObjectWrapper());
        /* 而以下代码你通常会在一个应用生命周期中执行多次*/
        /*获取或创建一个模版*/
        Template temp = cfg.getTemplate("test.ftl");

        /*创建一个数据模型 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");

/* 合并数据模型和模版*/
Writer out = new OutputStreamWriter(System.out);
temp.process(root, out);
out.flush();
}
}
```

注意：简单起见，以上代码是没有考虑异常处理的。

二、数据模型（Data Model）

2.1、基础类型

FreeMarker 的数据模型（root 状模型）可以使用标准的 java 类（Map，String 等等）来表示。在其内部，FreeMarker 使用的数据模型其实是———实现了 `freemarker.template.TemplateModel` 接口的对象。但是你却可以使用诸如 java collections 对象作为你的数据模型，这是因为这些数据模型通常会被（FreeMarker）在内部转换成 `TemplateModel` 类型的对象。这种功能呢就叫做 **object wrapping**。它可以透明地把任意对象转化成 `TemplateModel` 接口类型的实例。例如，它可以转换 `java.sql.ResultSet` 对象，也可以转变 `javax.servlet.ServletRequest` 对象，甚至可以转化 XML 文档表示的数据模型。

但是你在转换这些对象的时候，必须选取适合的包裹类（Wrapper，这些包裹类来实现普通对象到 `TemplateModel` 对象的转化工作，但你也可以自己定制一些 Wrapper）。既然模版可以访问的数据模型必须都是转化成 `TemplateModel` 后的对象，所以你首先应该熟悉各个实现 `TemplateModel` 接口的子类。

`TemplateModel` 主要有三个子类，每一个子类分别表示一种 FreeMarker 的基本类型，比如 `hashes` 类型用 `TemplateHashModel` 表示，`sequence` 类型实现用 `TemplateSequenceModel` 来表示，`numbers` 类型用 `TemplateNumberModel` 来表示，另外，自己也可以定制其他的数据模型的转化。

举个例子，如果你想转化 `java.sql.ResultSet` 为 `sequence`，那么你只需要写一个实现了 `TemplateSequenceModel` 接口的对象，并且该对象可以读取 `ResultSet` 然后并将其转换成相应的 `TemplateModel`，其实你要做的仅仅是生成一个 `ResultSet` 的包装类，并且继承 `TemplateSequenceModel` 接口。注意：一个类可以实现多个 `TemplateModel` 类型的接口，这也就是为什么 FreeMarker 的变量会有多种的原因。

注意：在包 `freemarker.template` 中提供了以上这些接口的默认实现。例如，你要转换一个 `String` 类型那么你可以直接使用 `SimpleScalar`，同样要转换一个 `Map` 类型你可以使用 `SimpleScalar`，诸如此类等等。

如果你放置到 `root` 数据模型中的对象本身就是实现了 `TemplateModel` 接口的实例，那么 `Object Wrapper` 并不会去对该对象进行任何转换。

2.2、Scalar 类型（单值对应的 Data Model）

FreeMarker 中的 scalar 类型可以表示如下的四种基本类型

- Boolean
- Number
- String
- Date

每种类型都是 `TemplateTypeModel` 接口的实现，`Type` 是以上类型名称的替换。这些接口只有一个方法 `getAsType()`，它返回以上 java 基本类型所表示的值（也就是 `boolean`，`Number`，`String` 类型的值）。

注意：由于历史版本的原因，`String` 所对应的类型是 `TemplateScalarModel` 而不是 `TemplateStringModel`。

以上接口的实现都可以在 `freemarker.template` 包中找到。但是 `boolean` 类型的值是用 `TemplateBooleanModel.TRUE` 和 `TemplateBooleanModel.FALSE` 来表示的。`Scalars` 类型在 FTL（FreeMarker 模版）中都是不可变类。也就是该类型在赋值之后是不能改变的，除非生成一个新的实例。

数据类型中存在的难点

FreeMarker 数据类型方面还有些许复杂，这是因为 java API 通常不区分 `java.util.Date-s`（也就是说表示日期的，表示时间的都可以用 `Date` 类来存储）。为了能够用文本准确的表示 `Date` 的值，FreeMarker 必须准确的知道 `Date` 对象里面存放的到底是哪一种日期格式。不幸的是 Java API 只能区分数据库中定义的日期格式，因为数据库语言会明确的使用 `date`，`time` 以及 `timestamp` 这三种类型来对日期进行存储，而相应的 `java.sql` 包中有 3 种类型与之对应。

而在 FreeMarker 中时这么处理的，`TemplateDateModel` 接口有 `getAsDate()`，`int getDateType()` 两个方法。你可以通过其 `getAsDate()` 获得一个 `java.util.Date` 对象，另外可以通过 `getDateType()` 再获得一个整数，而该整数用以指明究竟使用日期的哪种表达。该整型变量的值分别是 `DATE`，`TIME`，`DATETIME` 和 `UNKNOWN`，并且都是常量。

那么什么又是 `UNKNOWN` 类型呢？在 FreeMarker 中当 `TemplateDateModel` 不知道存放在其中的日期是什么类型（`date,time,timestamp`）时将会用 `UNKNOWN` 表示。

2.3、容器类型

在 FreeMarker 中一般有三种类型可以充当容器。

● Hashes

`Hashes` 类型是实现了 `TemplateHashModel` 接口的实例。`TemplateHashModel` 容器类型包含两个方法 `get(String key)` 它可以根据给定的 `key` 返回容器包含的子变量 `boolean isEmpty()` 它可以判定容器是否包含有子变量。如果容器中不包含任何变量的话，那么 `get` 方法会返回 `null`。

我们通常会使用实现该接口且名字叫 `SimpleHash` 的类来表示 `Hashes` 类型。其内部实现机制其实是使用 `java.util.Hash` 来存储子变量。你也可以通过该实例提供的方法来加入或者去除子变量。另外该类型的容器是不变类型(`immutable`)。

● sequences

`sequences` 通常是实现了 `TemplateSequenceModel` 接口的 `java` 对象。该类型包含两个方法 `TemplateModel get(int index)` 和 `int size()`。我们通常会使用实现该接口且名字叫 `SimpleSequence` 的类来表示 `sequence` 类型，其内部机制其实是使用 `java.util.List` 来存储子变量。`SimpleSequence` 有新增子变量的方法。

● Collections

`collections` 通常是实现了 `TemplateCollectionModel` 接口的 `java` 对象。该类型包含一个方法 `TemplateModelIterator iterator()`。该接口类似 `java.util.Iterator`。但是它返回的是 `TemplateModels` 而不是 `Object-s`。异常的时候会抛出 `TemplateModelExceptions`。我们通常会使用实现该接口且名字叫 `SimpleCollection` 的类来表示 `collections` 类型。

2.4、方法变量

方法变量通常是实现了 `TemplateMethodModel` 接口的类，该接口有一个方法 `TemplateModel exec(java.util.List arguments)`。当你使用方法表达式 (`method call expression`) 调用一个方法 (`exec`) 的时候，实际上是在执行 `exec`。方法表达式的参数其实就是方法参数的变形。方法的返回值也就是方法表达式的返回值。

由于方法接口 `TemplateMethodModelEx` 继承自 `TemplateMethodModel` 接口，所以方法也可以当作变量一样被放置到 `root` 中。而显然方法接口是没有默认实现的，因为这些实现都要你亲自书写。

举个例子，有一个方法它返回第一个字符串首次出现在第二个字符串中的位置。如果在第二个字符串中找不到，那么则返回 `-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)));
    }
}
```

如果你把方法变量按照如下的方式放置到 `root` 中

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

那么你就可以按照如下的方式在模版中使用

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

以下是输出:

```
2
-1
```

如果你想访问运行时的 `FTL` 环境，对变量进行读写那么你需要获取 `Environment.getCurrentEnvironment()`。

2.5、转换器变量

转换器是实现 `TemplateTransformModel` 接口的类。你可以在 `freemarker.template.utility` 包下面找到一些有用的转换器实现。

- 扩展你自己的转换器

转换器接口有一个方法 `Writer getWriter(Writer out, Map args)`。该方法将会转换标签之间的内容，首先把标签之间的内容读取到 `Writer` 对象中，再由 `Writer` 对象对其中的内容施行转换处理，转换后的内容会再次存储到 `Writer` 中。调用 `flush` 方法后会把内容输出。不需要你去调用 `out.close()`，当到达结束标签的时候 `close` 会自动被调用。

以下是一个转换标签之间内容为大写的例子

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

class UpperCaseTransform implements TemplateTransformModel {

    public Writer getWriter(Writer out, Map args) {
        return new UpperCaseWriter(out);
    }

    private class UpperCaseWriter extends Writer {

        private Writer out;
```

```

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

    public void write(char[] cbuf, int off, int len)
        throws IOException {
        out.write(new String(cbuf, off, len).toUpperCase());
    }

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

    public void close() {
    }
}
}

```

如果你把该变量放置到 **root** 数据模型中

```

root.put("upperCase", new UpperCaseTransform());

```

那么在模板文件中，你可以如下使用：

```

blah1
<@upperCase>
blah2
blah3
</@upperCase>
blah4

```

输出如下：

```

blah1
BLAH2
BLAH3
blah4

```

通常更好的做法是把一些可能公用的变量放置到 **Configure** 对象中当作为共享变量。另外一点你需要认识到，如果转换器是有状态的话，那么它必须存储在 **Writer** 实例中，而不是 **TemplateTransformModel**。那么 **TemplateTransformModel** 应当是无状态的，而且仅仅是创建 **Writer** 的工厂类。

考虑一下标签嵌套的情况和共享变量被多个线程同时访问的情况。**FreeMarker** 是这么处理标签嵌套的，其返回的 **Writer** 可能是实现了一个 **freemarker.template.TransformControl** 接口。该方法可以回调，那么就给了 **Writer** 对象一个机会是否执行嵌套在其中的标签（如 **<@myTransform>** 和 **</@myTransform>** 之间还有标签）请参考 **API** 获得更加详细的介绍。

2.6、节点变量

节点变量是对树状数据结构中节点的表述。节点变量可以用来处理 XML 文档，但是也可以用来处理具有树状组织接口的数据结构。一个节点变量具有以下一些属性，这些属性都是由 `TemplateNodeModel` 接口提供的。

- 基本属性：

`TemplateSequenceModel getChildNodes()`：一个节点具有一系列的孩子节点（除了那些叶子节点外）而孩子节点属于节点变量。

`TemplateNodeModel getParentNode()`：一个节点很显然有且只有一个父节点，除了根节点外。

- 可选属性：（具体情况下，没有意义的属性一般返回 `null`）

`String getNodeName()`：节点的名字也就是宏的名字，当你使用 `recurse` 和 `visit` 指令的时候需要首先知道节点的名字。

`String getNodeNameType()`：标识节点的类型 `element`, `text`, `comment`

`String getNamespaceURI()`：返回该节点的所处的名字空间（与 `FreeMarker` 的名字空间没有任何联系）

对于 `FreeMarker`，直接使用内建的节点类型（`node built-ins`），大多数情况下，节点变量仅提供节点访问的基本功能，更详细的例子在“`FreeMarker deals with XML`”

2.7、对象包裹

你在 API 中可以看到，`FreeMarker` 数据容器(`root`)可以放置任意的对象，而不一定就是实现了 `TemplateModel` 接口的对象。这是为什么呢？！因为 `FreeMarker` 提供的容器实例会在其内部把放置在其中的对象自动转换成实现了 `TemplateModel` 接口的对象。比如说，如果你放置一个 `String` 对象在容器中，它就会把 `String` 对象在内部自动转换成 `SimpleScalar`。

至于何时发生转换，这是容器自身逻辑的问题。但是最晚也会在获取子变量的时候进行转换，因为获取子变量方法会返回 `TemplateModel` 对象而不是 `Object` 对象。例如，`SimpleHash`, `SimpleSequence` 和 `SimpleCollection` 使用延迟转换策略（`laziest strategy`）；它们会在第一次获取子变量的时候把其他类型的对象转换成 `TemplateModel` 类型。

至于什么类型的对象可以被转换，以及具体转换成何种类型，一方面容器自身可以处理，另一方面也可以把它委托给 `ObjectWrapper` 实例去处理。`ObjectWrapper` 是一个接口具有一个方法 `TemplateModel wrap(java.lang.Object obj)`。用户可以传递一个 `Object` 对象，它就会返回一个与之对应的 `TemplateModel` 对象，或者抛出异常。这些转换规则是写死在 `ObjectWrapper` 实现里面的。

`FreeMarker` 提供的 `ObjectWrapper` 重要的实现有：

- `ObjectWrapper.DEFAULT_WRAPPER`：它可以把 `String` 转换成 `SimpleScalar`，`Number` 转换成 `SimpleNumber`，`List` 和 `array` 转换成 `SimpleSequence`，`Map` 转换成 `SimpleHash`，`Boolean` 转换成 `TemplaeBooleanModel.TRUE/FALSE` 等等。（对于其他的类型对象的转换就要调用 `BEANS_WRAPPER`）
- `ObjectWrapper.BEANS_WRAPPER`：它可以使用反射访问任意 `JavaBean` 的属性（后面有单独的一章专门介绍该对象）

我们来看一个具体的例子，来观察 `SimpleXxx` 究竟是如何工作的，`SimpleHash`，`SimpleSequence` 和 `SimpleCollection` 使用 `DEFAULT_WRAPPER` 来包裹子变量，所以下面这个例子是讲解的 `DEFAULT_WRAPPER` 处理原理。

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

假如 `root` 是数据模型的根，那么以下的数据模型可以表述如下：

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

注意在 `theMap` 和 `theList` 中的对象也可被当作子变量访问，这是因为当你试图以 `theMap.anotherString` 访问数据的时候，`SimpleHash` 会悄悄的把其替换成 `SimpleScalar` 对象。当你把任意的对象放置到 `root` 中时，`DEFAULT_WRAPPER` 将会调用 `BEANS_WRAPPER` 来对其进行转换。

```
SimpleHash root = new SimpleHash();
//放置一个简单的String 对象
root.put("theString", "wombat");
//放置任意的一个 java objects:
root.put("theObject", new TestObject("green mouse",
1200));
```

假定 TestObject 类如下:

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

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

    // JavaBean 属性
    // 注意这些属性你不能直接访问
    // 你必须给它们写 getter 方法
    public String getName() {return name;}
    public int getPrice() {return price;}

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

以上所表示的数据模型是:

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

因此我们在模板中合并他们

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

输出结果是:

```
green mouse  
1200  
-0,45990349068959124
```

你或许在之前的手册中已经看到，我们是用 `java.util.HashMap` 作为数据模型的 `root`，而不是 `SimpleHash` 或其它对象。这是因为当 `Template.process(...)` 处理的时候会自动依据 `Configuration` 级别指定的 `object_wrapper` 对其进行转换。

`object_wrapper` 默认的配置是 `ObjectWrapper.DEFAULT_WRAPPER`，如果你想改变包裹策略，如 `ObjectWrapper.BEANS_WRAPPER`，你可以做以下修改：

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

注意，你可以在这里设置任何实现了 `ObjectWrapper` 接口的对象，也就是说你可以定制你自己的包裹类。

三、配置（Configuration）

3.1、基础

`Configuration` 是一个存放应用级别（`application level`）公共配置信息，以及模版（`Template`）可使用的全局共享变量的一个对象。同时它还负责模版（`Template`）实例的创建以及缓存。`Configuration` 实际上是 `freemarker.template.Configuration` 对象的实例，使用其构造函数创建。通常应用使用一个共享的单实例 `Configuration` 对象。

`Configuration` 对象可被 `Template` 对象的方法使用，每一个模版实例都关联与一个 `Configuration` 实例，它是通过 `Template` 的构造函数被关联进去的，通常是你使用这个方法 `Configuration.getTemplate` 获得模版对象的。

3.2、共享变量

共享变量是那些定义给所有模版（`Template`）使用的变量。你可以通过 `configuration` 对象的 `setSharedVariable` 方法来添加共享变量。

```

Configuration cfg = new Configuration();
...
cfg.setSharedVariable("to_upper", new UpperCaseTransform());
cfg.setSharedVariable("company", "FooInc.");
ObjectWrapper.DEFAULT_WRAPPER

```

所有与该 `configuration` 对象关联的模版实例都可以通过获得 `to_upper` 转换器, `company` 来获得字符串, 因此你不需要再一次次的往 `root` 中添加这些变量了。如果你往 `root` 添加同名的变量, 那么你新添加的变量将会覆盖之前的共享变量。

警告!

如果 `configuration` 对象被多线程调用, 那么不要使用 `TemplateModel` 实现类作为共享变量, 因为他们是非线程安全的。例如基于 `servlet` 的 `web` 站点就是这种情况。

`Configuration` 对象初始化时已经包含一些共享转换器变量:

名字	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、配置参数

配置参数是那些可以影响 `FreeMarker` 运行行为的那些命名参数。例如 `locale`, `number_format`

配置参数存储在 `Configuration` 实例中, 它可以被模版实例 (`Template`) 修改。例如, 你在 `Configuration` 中设置了 `locale` 等于 `"en_US"`, 那么所有的模版对象都会使用, `"en_US"` 除非你在单个模版实例中利用 `setLocale` 方法修改了默认配置。因此 `configuration` 设置的参数可以当作是默认参数, 它可以被 `Template` 一级设置的参数覆盖, 而它们两者设置的参数信息又可以被环境中设置的参数所覆盖 (也就是模版文件指令设置的) 如下:

```

${1.2}
<#setting locale="en_US">
${1.2}

```

这种调用方式你可以想象成 3 个层 (配置对象层, 模版层, 运行环境层) 下面表格中显示了每一层对于参数的设置:

	Setting A	Setting B	Setting C	Setting D	Setting E	Setting F
Layer 3: Runtime environment	1	-	-	1	-	-
Layer 2: Template	2	2	-	-	2	-
Layer 1: Configuration	3	3	3	3	-	-

那么配置参数的最终结果分别是：A = 1, B = 2, C = 3, D = 1, E = 2.而F参数很可能就是null。如果要查询可设置的参数列表，你可以查阅 [FreeMarker API 文档](#) 的以下两个部分：

`freemarker.core.Configurable.setSetting(String, String):`

所有层的配置

`freemarker.template.Configuration.setSetting(String,String):`

Configuration 层的配置

3.4、加载模板

模版加载器

模版加载器是基于抽象路径（"`index.ftl`"或"`products/catalog.ftl`"）加载原始数据的那些对象。而究竟加载何种资源（目录中的文件数据还是数据库中的数据）取决于具体的加载器实现。当你调用 `cfg.getTemplate` 时，FreeMarker 将会询问你之前配置给 Configuration 对象的模版加载器，有该模版加载器负责文件的载入。

内建的模版加载器

你可以用以下三个方法来设置模版加载的三种方式

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

或者

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

或者

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

以上第一种方式显示的指定了一个文件系统中的目录，FreeMarker 将会在此目录记载模版，不用说，此目录必须存在，否在会抛出异常。

第二种方式以一个 Class 作为一个输入参数，当你想使用 `ClassLoader` 的方式来加载模版的时候，你就可以使用这种方式，这种方式将会调用来寻找模版文件，同时这种模版加载的方式要比前一种稳定一些尤其是在生产系统中。你可以很容易的把资源文件，以及图标等打包到jar文件中。

第三种方式把 web 应用的上下文以及基路径（相对与 `WEB-INF` 的父路进来说）作为

参数。该种方式的模版加载器将会从 web 应用上下文种加载模版。

从多个位置加载模版

如果你想从多个位置加载模版的话，你可以分别创建与不同位置对应的单个模版加载器，然后把它们包裹到一个名叫 `MultiTemplateLoader` 模版加载器中，最终通过方法 `setTemplateLoader(TemplateLoader loader)` 把其设置给 `Configuration` 对象，以下有一个从两个不同位置加载模版的例子：

```
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);
```

`FreeMarker` 将会首先在路径 `/tmp/templates` 中搜索模版文件，如果没有找到那么回到路径 `/usr/data/templates` 中搜索，如果还没有找到，那么则会尝试用 `class-loader` 的方式加载。

从其他资源中获取模版文件

如果在这些内建的模版加载器中没有一个符合你的要求，那么你可以自己定制一个模版加载器，只需要实现 `freemarker.cache.TemplateLoader` 接口就可以了，然后通过方法 `setTemplateLoader(TemplateLoader loader)` 把其传递给 `Configuration` 对象。

缓存模版

`FreeMarker` 缓存模版的意思就是，当你通过 `getTemplate` 方法获取一个模版的时候，`FreeMarker` 不仅会返回一个 `Template` 对象，而且会缓存该对象，当你下一次以相同的路径请求模版的时候，它就会返回缓存中的模版对象。如果你改变了模版文件，那么当你下一次获取模版的时候，`FreeMarker` 会自动重新加载，重新解析模版。虽然如此，但是如果直接判断一个文件是否修改过是一个耗时的操作，那么 `FreeMarker` 在 `Configuration` 对象级别提供了一个配置参数“`update delay`”。该参数的意思是 `FreeMarker` 多长时间去判断一次模版的版本，默认设置是 5 秒钟，也就是每个 5 秒就会判断模版是否经过修改，如果你想实时的判断，那么设置该参数为 0。另外一点需要注意，并不是所有的加载器都支持这种

判断方式，举例来说基于 `class-loader` 的模版加载器就不会发现你修改过模版文件。

对于删除缓存中的模版 `FreeMarker` 是这么做的，你可以使用 `Configuration` 对象方法 `clearTemplateCache` 以手工的方式清楚缓存中的模版对象。而实际上缓存部分可以作为一个组建加入到 `FreeMarker` 中（也就是它可以使用第三方缓存方案）你可以通过设置 `cache_storage` 这个参数来实现。对大多数开发者来 `FreeMarker` 自带的 `freemarker.cache.MruCacheStorage` 实现已经足够了。这个缓存使用 2 个级别的 `Most Recently Used`（最近最多用）策略。在第一个级别，所有的缓存条目都是使用强引用（`strongly referenced`：条目并不会被 JVM 所清楚，与其相对的弱引用 `softly reference`）直到达到最大时间，那些最近最少使用的条目就会被迁移到二级缓存。在这个级别条目都是使用弱引用直到达到过期。若引用与强引用的区域的大小是在构造函数中设置的，例如你想把强引用区域设置为 20，弱引用区域设置为 250，那你可以使用以下代码：

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

由于 `MruCacheStorage` 是默认的缓存实现，那么你也可以这样设置：

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

当你创建一个新的 `Configuration` 时，其默认使用 `MruCacheStorage` 缓存实现且默认的值 `maxStrongSize` 等于 0，`maxSoftSize` 等于 `Integer.MAX_VALUE`（也就是理论最大值）。但是对于高负荷的系统来说，我们建议 `maxStrongSize` 设置成一个非 0 的数值，不然会导致频繁的重新加载，重新解析模版。

3.5、异常处理

可能产生的异常

`FreeMarker` 产生的异常一般可归以下几类：

- `FreeMarker` 初始化阶段产生的异常：通常在你的应用中仅需要初始化 `FreeMarker` 一次，而当在这个时间段类产生的异常就叫做初始化异常。
- 加载解析模版期的异常：当你通过 `Configuration.getTemplate()` 方法获取模版的时候（如果模版之前没有被缓存），将会产生两类异常。

`IOException`：由于模版没有找到，或在读取模版的时候发生其他的 IO 异常，比如你没有读取该文件的权限等等。

`freemarker.core.ParseException` 由于模版文件的语法使用不正确。

- 执行期间的异常。：当你调用 `Template.process(...)` 方法的时候，会抛出两类异常。

`IOException` 往输出写数据时候发生的错误。

`freemarker.template.TemplateException` 其他运行期产生的异常。比如一个最常见的错误就是模版引用了一个不存在的变量。

异常的定制

由于客户一般不会直接使用该章节的内容去进行 `FreeMarker` 的配置（`FreeMarker` 官方也不赞成自己定义异常），所以以下的内容将略去。

四、其它说明

4.1、变量

在这一章中主要介绍模版是怎么样访问变量的，以及如何存储变量。

当你调用 `Template.process` 方法时，FreeMarker 内部会创建一个 `Environment` 对象直到 `process` 方法调用结束。这个变量存放着 FreeMarker 运行期的状态以及经由 `assign, macro, local global` 等创建的变量。当你需要读取某个变量的时候，FreeMarker 就会按照次序去查找，如果找到匹配的变量，那么就会返回。

- 1、在 `Environment` 对象中：
 - 1、循环变量是经由 `list` 指令创建的。
 - 2、在宏变量中可以创建 `local` 变量。
 - 3、在当前的命名空间中，你可以使用 `assign` 来给变量取值。
 - 4、使用 `global` 指令创建的变量，可以像使用数据模型中的数据一样在所有的命令空间都可以使用。
- 2、在数据库模型对象中：

这其中的变量是通过 `process` 方法被传递进去的。
- 3、在共享变量中：

这些是设置在 `Configuration` 对象中的变量。

4.2、字符编码

像大多数 java 应用一样，FreeMarker 也是使用“UNICODE”编码，虽然如此，但是有些情况下还是要面对字符编码问题（charsets），因为它必须与外界使用的多种多样的字符集进行交互。

输入的字符编码

当 FreeMarker 加载模版的时候，它必须得知道被加载文件的字符编码，你可以使用 `encoding` 设置来指定编码。这个设置仅仅在你调用 `Configuration` 对象的 `getTemplate` 方法的时候起作用。对于 `encoding` 这个参数提供了 `getter` 以及 `setter` 方法，当你使用 `getter` 方法获取编码的时候，它会基于你传递的 `locale` 来查找一张表（这张表是由 `locale, encoding` 对应组成的）来确定具体的 `encoding`，如果找不到的话，将会使用默认的 `encoding`（默认的 `encoding` 通常用 `setDefaultEncoding` 来设置）

同时你也可以使用 `ftl` 指令来指定页面的编码方式。

总之，把你所有的页面编码设置成为 `UTF-8` 将会是最好的办法。

输出字符编码

原则上 FreeMarker 将不对输出字符编码进行处理，因为输出是通过对象来实现的。因此这部分的翻译也就不在赘述，最好的办法依然是设置所有的字符编码为 `UTF-8`。

4.3、多线程

在多线程的环境下 `Configuration` 对象, `Template`, 以及数据模型都应该按照不变类去处理 (也就是只读对象)。即就是, 你只管创建它们而不能做修改。这可以帮助我们避免使用开销巨大的同步块。同时, 你也要清楚你使用 `getTemplate` 获得模版对象一般是从缓存中获取的, 所有你如果修改它的话, 那么其他线程对它的使用就会出错 (除非你是在单线程下访问的)

同样的道理, 我们也不建议你书写修改数据模型, 共享变量值的方法。

总之, 不要去修改可能在多线程环境下使用的变量。

4.5、Bean 包裹

`freemarker.ext.beans.BeansWrapper` 这个类实际上也是一个 `object wrapper` 对象, 最初被用来包裹任意的 POJO 成为 `TemplateModel`。而对象 `DefaultObjectWrapper` 是其的一个实现。所以这里讲的大部分理论都适合于对象 `DefaultObjectWrapper`, 但是除了这一点: `DefaultObjectWrapper` 可以包裹 `String, Number, Date, array, Collection (like List), Map, Boolean` 等类型, 而 `BeansWrapper` 不可以。

在以下的几种情况之下你需要使用 `BeansWrapper`:

- 在模板执行期间要可以修改 `Collections, Maps` (`DefaultObjectWrapper` 对象不允许这么做, 这是因为使用该对象包裹的时候仅仅是创建了一个副本)
- 如果想让等 `array, Collection and Map` 对象的标识标识保存直到被传递给包裹类的方法之后, 也就是说, 包裹类的方法可以调用未包裹之前的对象。
- 如果模板可以使用未包裹之前对象的方法。