

TINY FRAMEWORK

参考手册 **Version 1.2**

TINY GROUP

THINK BIG,

START SMALL,

SCALE FAST.

1. Tiny模板引擎	4
1.1 Tiny模板引擎简介	4
1.1.1 Tiny模板引擎之来历	4
1.1.2 Tiny模板引擎概述	4
1.1.3 卓越的性能表现	5
1.1.4 友好的错误提示	5
1.1.5 易于集成	6
1.1.6 简单易用类同Velocity的指令	6
1.2 Tiny模板引擎下载与使用	7
1.2.1 Tiny模板引擎开源许可	7
1.2.2 Tiny模板引擎运行环境	7
1.2.3 Tiny模板引擎Maven依赖	7
1.2.4 Tiny模板引擎源码下载	7
1.3 Tiny模板引擎语法参考	7
1.3.1 取值表达式: \${}	8
1.3.2 Tiny模板引擎指令集	8
1.3.2.1 赋值语句: #set指令	8
1.3.2.2 条件语句: #if...#else...#elseif...#end	9
1.3.2.3 循环语句: #for...#else#end	10
1.3.2.3.1 循环中断: #break	12
1.3.2.3.2 循环继续: #continue	13
1.3.2.4 模板嵌套语句: #include	13
1.3.2.5 宏定义语句: #macro	14
1.3.2.6 停止执行: #stop	16
1.3.2.7 行结束指令	17
1.3.3 文本内容	17
1.3.4 注释Comment	18
1.3.5 表达式expression	19
1.3.5.1 MAP常量	19
1.3.5.2 变量名Variable	19
1.3.5.3 基本表达式	20
1.3.5.4 数组常量	22
1.3.5.5 方法调用	23
1.3.6 缩进排版支持	23
1.4 Tiny模板引擎开发指南	24
1.4.1 构建Tiny模板引擎实例	24
1.4.2 为模板引擎添加资源加载器	24
1.4.2.1 添加字符串资源加载器	25
1.4.2.2 添加文件对象资源加载器	26
1.4.2.3 添加ClassLoader资源加载器	26
1.4.3 构建Tiny模板上下文	27
1.5 Tiny模板引擎之二次开发手册	28
1.5.1 Tiny模板引擎概念详解	28
1.5.1.1 Tiny模板引擎之引擎	28
1.5.1.2 Tiny模板引擎之资源加载器	33
1.5.1.2.1 资源加载器扩展	33
1.5.1.2.2 资源加载器接口定义	34
1.5.1.3 Tiny模板引擎之上下文Context	36
1.5.1.4 Tiny模板引擎之模板	37
1.5.1.5 Tiny模板引擎之宏	37
1.5.1.5.1 Tiny模板引擎之宏接口定义	38
1.5.1.5.2 编程方式定义宏	40
1.5.1.5.3 模板方式定义宏	42
1.5.1.5.4 宏调用方式	44
1.5.1.6 Tiny模板引擎之函数	45
1.5.1.6.1 Tiny模板引擎之函数接口定义	45
1.5.1.6.2 Tiny模板引擎之函数扩展	46
1.5.1.6.3 Tiny模板引擎之类成员函数扩展	47
1.5.1.7 Tiny模板引擎之布局	48

1.5.1.7.1 Tiny模板引擎布局之入门示例	49
1.5.1.7.2 Tiny模板引擎布局之进阶示例	51
1.5.1.8 Tiny模板引擎之国际化	52
1.5.2 系统内嵌函数	54
1.5.2.1 宏调用方法call, callMacro	54
1.5.2.2 格式化函数fmt, format, formatter	54
1.5.2.2.1 java.util.Formatter用法	55
1.5.2.3 求值函数eval, evaluate	58
1.5.3 Tiny模板引擎扩展	59
1.6 Tiny模板引擎之示例	59
1.6.1 从HelloWorld了解Tiny模板引擎	59
1.6.2 从文件系统加载HelloWorld模板	59
1.6.3 递归调用示例	60
1.6.4 多层宏调用示例	60
1.6.5 宏定义中调用宏示例	61
1.6.6 多次循环调用示例	62
1.6.7 输出内容缩进示例	62
1.7 Tiny模板引擎之最佳实践	62
1.7.1 布局之实践	62
1.8 与其它模板引擎对比	63
1.8.1 Tiny模板引擎 vs Velocity1.7	63
1.8.1.1 与Velocity指令差异	63
1.8.1.2 与Velocity语法差异	65
1.8.1.3 与Velocity性能差异	66
1.8.2 Tiny模板引擎 vs JetbrickTemplate	66
1.8.2.1 与Jetbrick指令差异	66
1.8.2.2 与Jetbrick语法差异	68
1.8.2.3 与Jetbrick性能差异	68
1.9 Tiny模板引擎之常见问题解答	68
1.9.1 Tiny模板引擎中模板已经编译成类，发布时是否可以不发布模板源文件？	68
1.9.2 Tiny模板引擎中模板的执行过程是怎样的？	68
1.9.3 Tiny模板引擎可以和Spring集成么？	68
1.9.4 Tiny模板引擎支持多实例运行么？	68
1.9.5 Tiny模板引擎有配置文件么？	69
1.9.6 为什么明明我写的模板文件是存在的，却抛出找不到模板异常？	69
1.9.7 可不可以从宏里传数据给调用的模板？	69
1.9.8 在布局文件中可不可以访问当前模板的变量？	69

Tiny模板引擎

Tiny模板引擎简介

Tiny模板引擎之来历



序

本来是没有自己写一个模板引擎的计划的，因为按我的理解，一直认识这种“语言”级的引擎，难度是非常大的。总感觉自己的水平不够，因此不敢有这个念头。直到大量使用Velocity的时候，碰到velocity诸多尽如人意的地方，但是又无能为力，退回到JSP吧，又心不甘。于是就期望着寻找一种语法结构接近velocity，但是又没有Velocity这些不方便之处的模板语言。于是进到一个模板语言群，一群大佬们个个至少是一个模板语言的作者，于是作者在里面表达了自己的期望，大佬们都介绍了自己的模板引擎，于是作者一个接一个的看源码，看文档。说实际，看文档，感觉都非常不错，都有自己的特色，看语法也都不错，除了一部分自己特别关注的点没有之外，其他部分都非常不错了。但是距离自己的诉求还是有差距，怎么办呢？于是就准备找一个最接近的模板引擎来进行一定的扩展，挑来挑去就挑中了jetbrick这个模板语言。

之所以挑中这个是因为以下几个原因：

1. Antlr词法及语法文件编写非常清晰，对于我这种Antlr盲来说，也可以看得懂，甚至可以照葫芦画瓢修改修改，这个非常重要，在后期进行许当的语法改进，这个就体现出优点了
2. 代码质量较好，使用sonar进行分析，给的结果都还是相当不错的，在作者看过的所有的模板语言中，算上成之选
3. 语法结构与Velocity的非常接近，这点对我也非常重要，因为我的想法就是velocity语法有相当的接受度，与Velocity语法接近，velocity的一些使用者可以方便的进行切换
4. 测试用例比较完善，在Tiny模板引擎完成之后，利用其测试用例进行测试发现了好几个BUG，说明还是非常有效果的
5. 环境搭建容易，直接下载源码，就可以安装成功，可以跑测试用例

好的，挑也挑了，选也选了，就开始编写TinyTemplate了，let's GO。

Tiny模板引擎概述

Tiny模板引擎 是一个基于Java技术构建的模板引擎，它具有体量小、性能高和扩展易的特点。适合于所有通过文本模板生成文本类型内容的场景，如：XML、源文件、HTML等等，可以说，它的出现就是为了替换Velocity模板引擎而来，因此在指令集上在尽量与Velocity接近的同时，又扩展了一些Velocity不能很好解决问题的指令与功能，在表达多方面则尽量与java保持一致，所以非常的易学易用。

1. 体量小表现在总共不到4000行的代码，去掉解析器近1000行，核心引擎只有不行3000行代码
2. 性能高表现在与现在国内几款高性能模板引擎如：Jetbrick、webit等引擎的性能相比，近乎伯仲之间，但是比Velocity、Freemarker等则有长足的进步，效率大致是Velocity四倍
3. 扩展性表现在Tiny框架引擎的所有环境都可以自行扩展，并与原有体系进行良好统一
4. 易学习表现在Tiny框架概念清晰、模块划分科学、具有非常的高内聚及低耦合
5. 使用方式灵活表现在，可以多例方式、单例方式，并可以与Spring等有良好集成

简要特点介绍

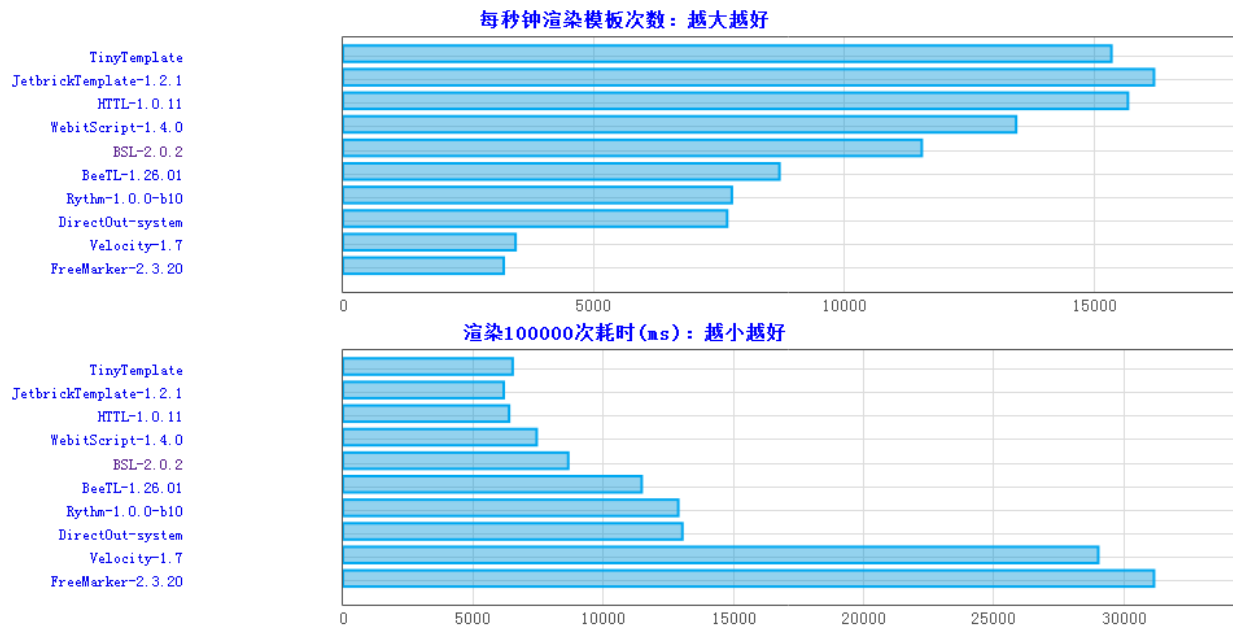
- 类似于 Velocity 的指令方式，相同或相似指令达90%左右
- 支持静态编译
- 支持编译缓存
- 支持热加载
- 支持可变参数方法调用
- 支持类成员方法重载

TinyFramework 参考手册

- 支持函数扩展
- 采用弱类型方式，对于模板层的代码编写约束更小，模型层怎样变化，模板层的代码调整都非常容易
- 支持宏定义 #macro
- 支持布局 Layout

卓越的性能表现

Tiny模板引擎采用编译方式，执行，因此比Velocity、FreeMarker等第一代模板引擎都快得多。但是由于使用了弱类型的方式，较强类型会稍慢一点，5%左右的性能差异，可以忽略不计。



上述数据在同一台计算机上测得。

友好的错误提示

TinyFramework 参考手册

```
line 10:8 token recognition error at: '#'
line 11:4 token recognition error at: '#'
Exception in thread "main" java.lang.RuntimeException:
org.tinygroup.template.parser.SyntaxErrorException: Template parse failed.
/template/tiny/test4.vm:11:11
message: missing ')' at 'end'
    7: #macro test1()
    8: <b>
    9:     #@test(
   10:         #bodyContent
   11:     #end
      ^^^

at
org.tinygroup.template.loader.FileObjectResourceLoader.loadTemplate(FileObjectResourceLoader.java:66)
)
at
org.tinygroup.template.loader.FileObjectResourceLoader.createTemplate(FileObjectResourceLoader.java:28)
at
org.tinygroup.template.loader.FileObjectResourceLoader.loadTemplateItem(FileObjectResourceLoader.java:34)
at
org.tinygroup.template.loader.AbstractResourceLoader.getTemplateItem(AbstractResourceLoader.java:81)
at org.tinygroup.template.loader.AbstractResourceLoader.getTemplate(AbstractResourceLoader.java:70)
at org.tinygroup.template.impl.TemplateEngineDefault.getTemplate(TemplateEngineDefault.java:138)
at org.tinygroup.template.impl.TemplateEngineDefault.renderTemplate(TemplateEngineDefault.java:184)
at org.tinygroup.template.impl.TemplateEngineDefault.renderTemplate(TemplateEngineDefault.java:259)
at org.tinygroup.template.TinyTemplateTestCase.main(TinyTemplateTestCase.java:14)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:597)
at com.intellij.rt.execution.application.AppMain.main(AppMain.java:134)
Caused by: org.tinygroup.template.parser.SyntaxErrorException: Template parse failed.
```

易于集成

由于Tiny模板引擎完全暴露的是Java接口，因此可以用在任何支持Java的层面上，因此可以用Java的地方，就可以使用Tiny模板引擎。

采用编码方式是这样子进行注入的：

```
public static void main(String[] args) throws TemplateException {
    TemplateEngine engine = new TemplateEngineDefault();
    engine.addTemplateLoader(new FileObjectResourceLoader("vm", "layout", null,
"src/test/resources"));
}
```

把它变成Spring配置方式，会有难度么，自己动手试一下？

简单易用类同Velocity的指令

TinyFramework 参考手册

```
<table>
  <tr>
    <td>序号</td>
    <td>姓名</td>
    <td>邮箱</td>
  </tr>
  #for (user in userlist)
  <tr>
    <td>${for.index}</td>
    <td>${user.name}</td>
    <td>${user.email}</td>
  </tr>
#end
</table>
```

Tiny模板引擎下载与使用

Tiny模板引擎开源许可

Copyright (c) 1997–2013, www.tinygroup.org (luo_guo@icloud.com).

Licensed under the GPL, Version 3.0 (the "License");

you may not use this file except in compliance with the License.

You may obtain a copy of the License at

<http://www.gnu.org/licenses/gpl.html>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Tiny模板引擎运行环境

- JDK1.6+
- ANTLR runtime 4.x
- Eclipse JDT

Tiny模板引擎Maven依赖

```
<dependency>
  <groupId>org.tinygroup</groupId>
  <artifactId>org.tinygroup.template</artifactId>
  <version>会不断变化</version>
</dependency>
```

Tiny模板引擎源码下载

TODO

TinyFramework 参考手册

一 Tiny模板引擎语法参考

阅读约定：

符号	说明
[...]	表示中间的内容可以省略

取值表达式：`{}`

语法

`{expression}`：输出表达式的计算结果
`#{expression}`：输出表达式的计算结果，并 escape 其中的 HTML 标签。

其中expression必须是一个合法的TinyTemplate表达式，具体参考表达式小节。

示例

`{user.name}`或`#{user.name}` 表示读取user对象的name属性
`{user.books.size()}`或`#{user.books.size()}` 表示调用用户的books属性的size()方法
`{user.description}`或`#{user.description}` 表示显示用户的description属性



小贴士

如果表达式执行结果为null或void，则不会输出任何内容。

Tiny模板引擎指令集

赋值语句：`#set`指令

语法

`#set(name1=expression, name2=expression, [...])`用于向当前上下文赋值
`#{set(name1=expression, name2=expression, [...])}`用于向当前模板的上下文赋值

赋值语句的格式必须以`#set(...)`的格式进行定义，如果要一次定义多个赋值语句，可以用半角的“，”进行分隔。

示例

`#set(tips="hello, World")`
`#set(numberArray=[1, 2, 3, 4, 5])`
`#set(dictMap={"male":1, "female":2})`

TinyFramework 参考手册



小贴士

如果对一个一个变量多次进行赋值，将对其值进行替换，比如：

```
#set(name="abc", name="def")
```

最终name变量中赋的值为字符串“def”

在Tiny模板语言中，不论是定义的变量还是循环变量，在宏模板执行过程中将全程有效，直到被修改。

在Tiny语言中，引入了变量作用域近者优先的概念，当前区块有，则取当前区块，如果当前区块没有，则取离得最近的变量。

比如：

示例

```
#for(i:[1, 2, 3, 4, 5])
#for(i:[6, 7, 8, 9])
  ${i}
#end
${i}
#end
```

在上面的代码中，两个区块的`${i}`不会任何影响。



小贴士

关于上下文，如果调用宏，会产生一个上下文；如果进入循环语句，也会创建一个上下文。这些创建的上下文，在其生命周期是有限的，出了生命周期以后，设置在他上面的变量就不能被访问了。如果在宏里或循环里，想把值设到自己的生命周期结束之后还可以被继续使用，就要设置到模板的上下文上。

设置到当前上下文用`#set`，设置到模板的上下文上，则用`#!set`，如果当前位置就在模板中，使用`#set`和`#!set`没有任何区别。

条件语句：`#if...#else...#elseif...#end`

语法

```
#if(expression)
...
#elseif(expression)
...
#else
...
#end
```

其中expression必须是一个合法的TinyTemplate表达式，具体参考表达式小节。

TinyFramework 参考手册

示例

```
#if(user)
...
#end
表示如果user对象不为空时，执行里面的代码块
#if(user.age<7)
... 小学前
#elif(user.age<13)
... 小学
#elif(user.age<16)
... 中学
#elif(user.age<19)
... 高中
#else
... 高中以后
#end
```



小贴士

其中`#if`指令及`#end`指令必须包含，`#elseif`及`#else`指令可以省略，`#elseif`可以多次出现，而`#else`最多只能出现一次。

多个条件之间可以用`&&`、`||`等进行连接。

有时候`#else`或`#end`会后后面的字符内容连起来，从而导致模板引擎无法正确识别，这时就需要用`#{else}`或`#{end}`方式，避免干扰。

循环语句：`#for...#else#end`

语法

```
#for(var:expression)
...
#else
...
#end
表示对expression进行循环处理，当expression不可以循环时，执行#else指令部分的内容。
```

其中`expression`必须是一个合法的TinyTemplate表达式，具体参考表达式小节。

示例

```
#for(number:[1, 2, 3, 4, 5])
  value:${number}
#end

#for(book:user.books)
  书名: ${book.title}, 标题: ${book.author.name}
#else
  No books found.
#end
```

Tiny模板引擎中的循环语句，`expression`可以支持任意的表达式。



逻辑表达式判定方法

Tiny模板引擎对于表达式进行了多种强力支持，不仅仅只有集合类型才可以参与运算，它的执行规则如下：

- 如果是`null`，则不执行循环体。
- 如果是`Map`，则循环变量存放其`entry`，可以用`循环变量.key`、`循环变量.value`的方式读取其中的值
- 如果是`Collection`或`Array`则循环变量放其中的元素
- 如果是`Enumeration`、`Iterator`，则循环变量存放其下一个变量。
- 如果是`enum`类，则循环变量存放其枚举值
- 否则，则把对象作为循环对象，但是只循环一次



循环状态变量

每个`#for`语句，会在循环体内产生两个变量，一个是变量本身，一个是变量名+“`For`”，比如：

```
#for(num:[1, 2, 3, 4, 5])
```

```
...
```

```
#end
```

这时在循环体内可以有两个变量可以访问一个是“`num`”，一个是“`numFor`”。

其中`numFor`是其状态变量，用于查看`for`循环中的一些内部状态下面用`numFor`来说明：

- `numFor.index` 可用于内部循环计数，从 1 开始计数。
- `numFor.size` 获取循环总数。如果对 `Iterator` 进行循环，或者对非 `Collection` 的 `Iterable` 进行循环，则返回 `-1`。
- `numFor.first` 是否第一个元素。
- `numFor.last` 是否最后一个元素。
- `numFor.odd` 是否第奇数个元素。
- `numFor.even` 是否第偶数个元素。



小贴士

TinyFramework 参考手册

#end指令在有歧义的时候可以用\${end}代替。

#循环变量及循环状态变量只在循环体内可以使用，出了循环体则不再可用。



注意

虽然Tiny模板引擎支持同名循环变量嵌套使用，但是强烈不建议这样做，会导致程序难于阅读与理解。

建议的方式

```
#for(i:[1, 2, 3]
#for(j:[4, 5, 6])
...
#end
#end
```

不建议的方式

```
#for(i:[1, 2, 3]
#for(i:[4, 5, 6])
...
#end
#end
```

循环中断：#break

循环中断语句，只能用在循环体内，表示跳出当前循环体。

语法

```
#break(expression)
#break
```

其中expression必须是一个合法的TinyTemplate表达式，具体参考表达式小节。

TinyFramework 参考手册

示例

```
#for (num: [1, 2, 3])  
  #break (num==2)
```

```
#end
```

表示当num的值为2的时候，跳出循环体。

它等价于：

```
#for (num: [1, 2, 3])  
  #if (num==2) #break #end
```

```
#end
```

可以看出上面的写法更方便。



小贴士

`#break`只能跳出一层循环，不能跳出多层循环。

`#break`只能放在循环体中，放在循环体外，会导致编译失败。

循环继续：`#continue`

循环继续语句，只能用在循环体内，表示不再执行下面的内容，继续下一次循环。

语法

```
#continue (expression)
```

```
#continue
```

其中expression必须是一个合法的TinyTemplate表达式，具体参考表达式小节。

示例

```
#for (num: [1, 2, 3])  
  #continue (num==2)
```

```
#end
```

表示当num的值为2的时候，执行下一次循环。

它等价于：

```
#for (num: [1, 2, 3])  
  #if (num==2) #continue #end
```

```
#end
```

可以看出上面的写法更方便。



小贴士

`#continue`只能继续当前循环，不能继续外层循环。

`#continue`只能放在循环体中，放在循环体外，会导致编译失败。

模板嵌套语句：`#include`

模板嵌套语句用于将另外一个模板的内容在当前位置进行输出

TinyFramework 参考手册

语法

```
#include(expression)
#include(expression, {key:value, key:value})
```

其中expression必须是一个合法的TinyTemplate表达式，具体参考表达式小节。

这个表达式的执行结果应该是一个字符串，其标示了要嵌套的子模板的路径。

它后可以跟参数，也可以不跟参数，如果跟参数的话，只能跟一个map类型的值。

示例

```
#include("/a/b/aa.page")    ##表示绝对路径
#include("../a/b/aa.page")  ##表示相对路径
#include(format("file%s-%s.page", 1, 2))
##表示采用格式化函数执行结果作为路径，这个例子中为：与当前访问路径相同路径中的“file1-2.page”文件

#include("/a/b/aa.page", {aa:user, bb:book})##表示带参数访问，会带过去两个参数aa和bb。
```



小贴士

子模板可以访问所有祖先模板中的变量。

出于封装性方面的考虑，不允许子模板修改父模板中变量的值，以避免不可预知的问题。

宏定义语句：#macro

在Tiny模板引擎中，宏是一个非常强大灵活的对象，使用它可以避免在模板中编写重复的代码。

语法

```
#macro macroName([varName[, varName]])
    #bodyContent
#end
```

宏的名字和变量的名字必须符合Tiny变量的定义规范，宏的参数的个数可以为0~N个。

#bodyContent表示中间可以包含任意的符合Tiny模板规范的内容。

模板调用

模板的调用方式有两种：

一种是单行调用方式，格式如下：

```
#macroName([expression|varName:expression[, expression|varName:expression]*])
```

另外一种带内容调用方式，格式如下：

```
#@macroName([expression|varName:expression[, expression|varName:expression]*])
```

.....

```
#end
```

参数支持按顺序赋值方式，也支持命名赋值（varName:值）方式。

TinyFramework 参考手册

示例

```
#macro header(subTitle)
  <h1>Tiny框架: ${title}</h1>
#end
调用方式:
#header("homepage")
#header("about")
运行结果:
<h1>Tiny框架: homepage</h1>
<h1>Tiny框架: about</h1>
```

高级示例

```
#macro div()
  <div>
    #bodyContent
  </div>
#end
#macro p()
  <p>
    #bodyContent
  </p>
#end
调用方式:

#@div()
  #@p()
    <em>一些信息</em><b>一些内容</b>
  #end
#end

运行结果:
<div>
  <p>
    <em>一些信息</em><b>一些内容</b>
  </p>
</div>
```



小贴士

macro的访问有两种方式：一种是包含内容的，一种是不包含内容的。

#bodyContent的出现次数可以出现0~N次。

如果参数变量与外部变量的名称完全相同，可以不在调用时传参，Tiny模板引擎会自动读取对应的值。

通过命名传值可以避免复杂的传值指令及不必再费心考虑参数顺序。



注意

宏的定义支持嵌套定义，也就是说支持下面的形式：

TinyFramework 参考手册

```
#macro aa()  
aa-Content  
#macro(bb)  
bb-Content  
#end  
#end
```

它等价于下面的方式:

```
#macro aa()  
aa-Content  
#end  
#macro(bb)  
bb-Content  
#end
```

考虑到代码的易读性, 建议还是采用第二种的定义方式。



高级技巧

在Velocity中, 在定义宏的时候是不可以调用带内容的宏的, 而在Tiny模板引擎中, 则可以无限定义。



Velocity不支持下面的方式

```
#macro(macroName)  
#@subMacroName()  
$bodyContent  
#end  
#end
```

在Tiny模板引擎中, 则支持下面的方式:



Tiny框架引擎支持良好

```
#macro macroName()  
#@subMacroName1()  
#@subMacroName1()  
$bodyContent  
#end  
#end  
#end
```


TinyFramework 参考手册

一 停止执行：#stop

停止执行语句，表示不再执行下面的内容，直接结束当前处理。

语法

```
#stop(expression)
#stop
```

其中expression必须是一个合法的TinyTemplate表达式，具体参考表达式小节。

示例

```
#continue(error)
表示当error执行布尔运行为真的时候，直接退出。
它等价于：
#if(num==2)#stop#end
可以看出上面的写法更方便。
```



小贴士

#stop只能继续当前当前处理，而不表示中断所有处理，比如：在模板文件里执行，只会停止模板文件的执行；在宏里执行，只能停止宏的执行。

行结束指令

语法：

```
#eol
#{eol}
```

表示显式输出一个“\r\t”。



小贴士

在Tiny模板引擎中，默认会把文本输出内容进行trim操作，因此，默认是没有回车换行符的。因此，如果想额外增加一个回车换行符，就需要增加#eol指令。

示例：

文本内容

语法

```
非指令、非注释、非表达式的所有内容
#[[...]]#，不解析文本块的所有内容
```

示例

```
This ia test info.
<a href="#">link</a>
#[[
${abc}
#macro aa()info #end
]]#
它的运行结果为:
This ia test info.
<a href="#">link</a>
${abc}
#macro aa()info #end
```

字符转义

如果有些内容本来是文本内容，但是由于与模板引擎的指令或表达式产生冲突，此时可以采用转义方式进行处理。
比如：
要输出：`${abc}` 而不是要读取变量`abc`的值，则可以用这样的写法：`\${abc}`
支持的转义方式有：

```
\\
\#
\$
比如下面的示例：
\#macro abc()===>#macro abc()
\${abc}===>${abc}
\\\$ {abc}===>\${abc}
```



小贴士

非模板引擎支持的类似的指令，比如：`#aabbff`，不会被识别为指令，而会原样输出，可以不进行转义。

`"\"` 后面跟的字符不是 `"#"` 和 `"$"`，也不需要进行转义，直接输出。

注释Comment

语法

```
##这里是行注释内容
#--这里是块注释内容 --#
## 这里是块注释内容 *#
```

示例

```
## ${abc} #abc() 或者其它内容
#--
这里可以出现任意内容
--#
##
这里可以出现任意内容
*#
```



小贴士

之所以支持两种块注释方式，是为了在兼容性及方便性方面提供更大的便捷。

`## *#`方式是为了与Velocity兼容，这样熟悉Velocity的人员更容易上手。

`#-- --#`是为了便于把Html中的注释改成Tiny模板注释。

表达式expression

MAP常量

语法

```
{expression:expression,...}
```

示例

```
{} ##表示空Map
{aa:"aaValue",bb:"bbValue"} ##纯字符串Map
{aa:1,bb:"bbValue"} ##数字及字符串混合Map
{"aa",1,bb:"2",cc:1+2} ##数字，变量，表达式混合Map
```

调用示例

```
${map.key}
${map["key"]}
${map.get("key")}
```



小贴士

如果不能确认，前面的变量是否为空，可以加一个安全调用方式：

```
${map?.key}
${map?.get("key")}
```

变量名Variable

TinyFramework 参考手册

变量名

语法

`[_a-zA-Z][_a-zA-Z$0-9]*`
上面的正则表达式表示，在Tiny框架引擎中，变量必须是以下划线或大小写字母开头的，后续可以跟下划线或大小写字母和数字的字符串，才可以作为变量名。
实际上宏的名字，也是遵守同样的规则。



小贴士

只要符合上面的规范的字符串都可以作为Tiny框架引擎中的变量，即使是java的关键字也可以。

由于在进行循环时，Tiny模板引擎会在循环变量名后附加“`For`”作为状态变量，因此，需要注意避免冲突，以影响使用。

基本表达式

类型	表达式	说明
数字常量	<code>123</code> <code>123L</code> <code>0.01D</code> <code>99.99E</code> <code>-10D</code>	
字符串常量	<code>"abc\r\n"</code> <code>'abc\u00A0\r\n'</code>	不论是单引号框起来的字符串还是双引号框起来的字符串，都
布尔值常量	<code>true</code> <code>false</code>	
空值常量	<code>null</code>	
算术计算	<code>+</code> <code>-</code> <code>*</code> <code>/</code> <code>%</code>	

TinyFramework 参考手册

自增/自减	<div>++</div> <div>--</div>	不论放在变量前面与后面，没有区别
位运算	<div>~</div> <div>&</div> <div> </div>	
移位运算	<div>>></div> <div>>>></div> <div><<</div>	
比较运算	<div>==</div> <div>!=</div> <div>></div> <div>>=</div> <div><</div> <div><=</div>	<div>==</div> 的执行逻辑是对两边的表达式执行equals运算。但是为所有的对象都可以用比较符进行比较运算，只不过，实现不支持操作异常。
逻辑运算	<div>!</div> <div>&&</div> <div> </div>	
三元表达式	<div>exp?a:b</div> <div>exp?:b</div>	a?:b等价于a?a:b，当表达式比较复杂的时候，这种简写形式会更方便。
数组读取	array[i]	
成员属性访问	object.fieldName	
成员方法调用	object.methodName([...])	可以通过框架为某种类型增加新的方法或覆盖原有方法。
方法调用	functionName([...])	调用框架中的内嵌或扩展方法。



逻辑表达式判定方法

Tiny模板引擎对于布尔表达式进行子多种强力支持，不仅仅只有布尔值才可以参与运算，它的运行规则如下：

- 如果是null，则返回false
- 如果是布尔值，则返回布尔值

TinyFramework 参考手册

- 如果是字符串且为空串“”，则返回false
- 如果是集合，且集合中没有元素，则返回false
- 如果是数组，且数组长度为0，则返回false
- 如果是Iterator类型，则如果没有后续元素，则返回false
- 如果是Enumerator类型，则如果没有后续元素，则返回false
- 如果是Map类型且其里面没有KV对，则返回false
- 否则返回true



小贴士

浮点数比较，不推荐采用==方式进行比较，因为这样会由于精度原因导致出现误差，而导致看似相同的结果在执行equals的时候，返回false.

在访问属性或成员变量的时候，普通的方式是用“.”运算符，也可以使用“?.”运算符，表示如果前置变量非空，都继续执行取属性值或调用成员函数，避免空指针异常的发生。



警告

浮点数比较，不推荐采用==方式进行比较，因为这样会由于精度原因导致出现误差，而导致看似相同的结果在执行equals的时候，返回false.

数组常量

语法

```
[expression,...]
```

示例

```
[] ##表示空数组
[1..5]##等价于[1,2,3,4,5]
[5..1]##等价于[5,4,3,2,1]
[(1+4)..1]##等价于[5,4,3,2,1]
[1,2,3,4,5]##纯数字数组
[1,"aa",2,"cc",3]##数字及字符串混合数组
[1,aa,2,"cc",3]##数字，变量，字符串混合数组
```

调用示例

```
${list[1]}
${list.get(1)}
```



小贴士

如果不能确认，前面的变量是否为空，可以加一个安全调用方式：`${list?.[index]}`或`${list?.get(1)}`

TinyFramework 参考手册

注意：

方法调用

语法

```
functionName(...)  
varName.functionName(...)
```

示例

```
${format("a:%10s b:%10s c:%10s", 1, 2, 3)}  
${"info".length()}
```



小贴士

Tiny模板框架中内嵌有一些函数，用户可以方便的进行函数扩展，具体请查阅函数扩展章节。

缩进排版支持

对于Velocity的排版方面，是让我对其无法忍受的重要原因，看看他生成出来的内容，乱成一锅粥的样子，我就知道这种内容只能适合机器读取，而不适合于人类读取。所以，在编写TinyTemplate之前我就决定，必须解决这个问题。

为此，我增加了几个特殊的标记用来解决这个问题。

指令	说明
#[, #{[]	缩进一格
#, #[]	取消缩进一格
#t, #{t}	添加当前缩进空格
#eol, #{eol}	换行

示例：

```
#for(i:[1..3])  
  ${i}#eol  
  #  
  #for(j:[1..3])  
    #t${i}*${j}=${i*j}#eol  
  #end  
#]  
#end
```

运行结果：

TinyFramework 参考手册

```
1
    1*1=1
    1*2=2
    1*3=3
2
    2*1=2
    2*2=4
    2*3=6
3
    3*1=3
    3*2=6
    3*3=9
```



小贴士

如果不加缩进排版指令，会采用紧凑模式，把指令间的空字符全部去掉的。

模板引擎可以设置是否启用紧凑模式，默认是未启用状态，如果启用了紧凑模式，则上面的四个缩进排版指令会全部失效，同时系统的性能也会有显著提升。

在调试模式，建议关闭紧凑模式；在运行期建议打开紧凑模式以提升性能。

Tiny模板引擎开发指南

Tiny模板引擎，以简单易用、易于扩展为设计目标，既可以满足初级用户使用的需求，也可以满足高用户的保用需求。

初级用户只要查看本章内容即可，高级用户可以通过查看二次开发手册继续深造。

构建Tiny模板引擎实例

Tiny模板引擎的获取非常简单：

```
TemplateEngine engine = new TemplateEngineDefault();
```



小贴士

Tiny模板引擎支持多实例运行，也就是说在一个环境中，你完全可以构建多个模板引擎实例来执行不同的任务，比如一个用来做Web页面渲染，一个进行文档生成，一个用来代码生成。请放心，多个引擎之间不会有任何影响。

从引擎的角度，他不关心模板是由哪个加载器加载的。因此在引擎里没有获取模板对象的方法。

为模板引擎添加资源加载器

Tiny模板引擎虽然上自己不能对模板进行任何处理，因此必须通过资源加载器来完成对模板相关的处理，为模板引擎添加资源加载器的过程非常简单：

TinyFramework 参考手册

```
TemplateEngine engine = new TemplateEngineDefault();
engine.addTemplateLoader(new
StringResourceLoader()); //添加内串资源加载器--用于对内存中的字符串对象进行处理
engine.addTemplateLoader(new
ClassLoaderResourceLoader(...)); //添加类加载器资源加载器--用于对编译成类的class文件进行处理
engine.addTemplateLoader(new
FileObjectResourceLoader(...)); //添加文件对象资源加载器--用于对各种文件系统中的文件进行处理
```

FileObjectResourceLoader的构造函数说明:

```
FileObjectResourceLoader(String templateExtName, String layoutExtName, String
macroLibraryExtName, String root)
templateExtName: 模板文件的扩展名
layoutExtName: 布局文件的扩展名
macroLibraryExtName: 宏仓库文件的扩展名
root: 要加载模板资源的根目录
```

ClassLoaderResourceLoader的构造函数说明:

```
public ClassLoaderResourceLoader(String templateExtName, String layoutExtName, String
macroLibraryExtName)
templateExtName: 模板文件的扩展名
layoutExtName: 布局文件的扩展名
macroLibraryExtName: 宏仓库文件的扩展名
```



小贴士

templateExtName必须输入, 另外两个如果用不到, 可以设置为null。

扩展名不用带“.”, 只要输入小数点后面的部分就可以了。

文件名及扩展名都是大小写敏感的。因此, 如果文件明明存在, 但是又提示找不到, 请确认文件的大小写是匹配的。

在Tiny模板引擎中, 对于扩展名, 没有任何要求, 完全由使用自己自己确定就可以了。



小贴士

一个引擎可以添加任意多个资源加载器, 即使是同种类型的也没有关系。

三种类加载器兼容了字符串、文件、类, 所以, 大多数情况下都足够了, 如果需要从数据库加载模板对象, 则要自行扩展。

添加字符串资源加载器

Tiny模板引擎的通过字符串创建模板对象是非常容易的:

TinyFramework 参考手册

```
TemplateEngine engine = new TemplateEngineDefault();
    ResourceLoader<String> resourceLoader=new StringResourceLoader();
    engine.addTemplateLoader(resourceLoader);
    Template template=resourceLoader.createTemplate("HelloWorld");
    template.render();
```



小贴士

注意：要利用字符串创建模板对象，一定要显式的利用字符串的createTemplate方法创建，因此一定要有一个StringResourceLoader的实例对象才行。

原因是通守字符串变量创建的模板对象，不存在路径，因此，无法通过引擎渲染，只能通过Template的对象进行渲染。

添加文件对象资源加载器

Tiny模板引擎通过文件创建模板对象是很容易的：

```
public static void main(String[] args) throws TemplateException {
    TemplateEngine engine = new TemplateEngineDefault();
    engine.addTemplateLoader(new FileObjectResourceLoader("html", null, null,
"src\\main\\resources\\templates"));
    engine.renderTemplate("/helloworld.html");
}
```



小贴士

注意模板路径与root目录之间的关系，上面的示例中：

root是在src\main\resources\templates，也就表示在访问模板时，路径“/”就表示目录“src\main\resources\templates”，所以/helloworld.html实际上就是指：src\main\resources\templates\helloworld.html文件。

在Tiny模板引擎中，所有的对资源的引用，其路径都采用linux文件规范，也就是用“/”来作为不同层次目录的分隔符。



小贴士

如果一定要获取模板对象，可以保留模板加载器的对象，通过它来获取Template对象。

添加ClassLoader资源加载器

Tiny模板引擎通过类加载器创建模板对象是很容易的：

TinyFramework 参考手册

```
public static void main(String[] args) throws TemplateException {
    TemplateEngine engine = new TemplateEngineDefault();
    engine.addTemplateLoader(new ClassLoaderResourceLoader("html", null, null);
    engine.renderTemplate("/helloworld.html");
}
```



提示

路径“/helloworld.html”模板编译完成后对应的类在当前ClassLoader中必须是存在的，否则会报找不到模板异常。

当然，有时候，这些类并不在当前运行环境的当中，它可能在某个目录或者jar包文件中，尤其是采用类型于Bundle这种解决方案的时候，也很简单，只要在构造函数后面再添加一个参数即可。

```
ClassLoaderResourceLoader(String templateExtName, String layoutExtName, String
macroLibraryExtName, URL[] urls)
```

你就是你可以用urls来指定这个资源加载器负责到哪些外部文件中查找这些jar或目录。



小贴士

在Tiny模板引擎中，所有的对资源的引用，其路径都采用linux文件规范，也就是用“/”来作为不同层次目录的分隔符。

ClassLoaderResourceLoader使得把模板编译成的类，打成jar包发布而真正的模板文件不再在运行环境中存在成为可能。

通过ClassLoaderResourceLoader，可以方便的实现即使是编译成class文件，依然可以进行热加载，只要再新建一个Loader就可以了。



小贴士

如果一定要获取模板对象，可以保留模板加载器的对象，通过它来获取Template对象。

构建Tiny模板上下文

Tiny模板引擎的在进行赏析渲染时，需要有上下文存在。当然，为了方便开发调试，如果没有参数传递，也可以不传入，但是如果要和现有的环境对接，那么就一定需要构建上下文，否则，你的模板就成为静态模板了。

```
TemplateContextDefault()
TemplateContextDefault(Map dataMap);
默认的模板上下文有两个构造函数，你可以直接创建一个空的，也可以利用一个Map把map中的值直接放入上下文。
TemplateContext context=new TemplateContextDefault();
或
TemplateContext context=new TemplateContextDefault(map);
```

 小贴士

Tiny模板引擎之二次开发手册

Tiny模板引擎基础概念介绍

中文概念	英文概念	解释
引擎	Engine	用于协调整个模板引擎的动作，属于协调者的角色
模板	Template	具体的一个模板，它采用代码编写或利用模板语言方式定义的方式构建，用
宏	Macro	一个模板片断，有用于访问的名字，调用时可以传入参数，在调用时可以包的#macro指令创建。
布局	Layout	布局从结构上与Template基本相同，只是在它里面有一个\${pageContent}占
函数	Function	用于对引擎进行扩展增加新的函数，或者为某种类型的类增加成员函数。
资源加载器	ResourceLoader	用于加载各种资源，默认引擎只能访问资源加载器中的内容，而不能访问资
宏库文件	MacroLibrary	用于定义各种公用的宏。

 小贴士

本章节的内容，针对于想对Tiny模板引擎有深入了解或期望进行二次扩展的人员，对于普通开发人员，可以不看本章内容。

模板、宏库文件、布局有结构上完全相同，只是在使用限制上有些许差别。

1. 模板文件中定义\${pageContent}，会永远找不到此变量，而不会产生任何效果。
2. 布局文件中可以必须定义\${pageContent}，以便在此位置上渲染模板内容。（从语义上不定义也不会报错，但是会导致只会渲染布局文件本身）
3. 宏库文件中可以定义许多的macro，不建议定义注释、和#macro指令以外的内容。（实际上你也可以定义其它的模板语言片断，如文本内容，但是这些内容会不起任何作用，仅仅是占用一定存储空间）。

Tiny模板引擎概念详解

Tiny模板引擎之引擎

 小贴士

所谓引擎，说得好听就是Tiny模板引擎的核心，说得不好听，就是一个打酱油的，他实际不做什么处理，他只是起组织协调作用。

TinyFramework 参考手册

理论上，Tiny模板引擎也是可以进行扩展或者自行实现的，但是实际当中，并不建议这么做。

```
/**
 * 模板引擎
 * Created by luoguo on 2014/6/6.
 */
public interface TemplateEngine extends TemplateContextOperator {
    /**
     * 设置模板加载器列表
     *
     * @param templateLoaderList
     */
    void setTemplateLoaderList(List<ResourceLoader> templateLoaderList);
    /**
     * 设置编码
     *
     * @param encode
     * @return
     */
    TemplateEngine setEncode(String encode);
    /**
     * 设置
     *
     * @param i18nVistor
     */
    TemplateEngine setI18nVistor(I18nVisitor i18nVistor);
    /**
     * 返回国际化访问接口实现类
     *
     * @return
     */
    I18nVisitor getI18nVisitor();
    /**
     * 添加函数
     *
     * @param function
     */
    TemplateEngine addTemplateFunction(TemplateFunction function);
    /**
     * 返回注册的方法
     *
     * @param methodName 注册的方法名
     * @return
     */
    TemplateFunction getTemplateFunction(String methodName);
    /**
     * 返回注册的方法
     *
     * @param className 绑定的类名
     * @param methodName 注册的方法名
     * @return
     */
    TemplateFunction getTemplateFunction(String className, String methodName);
    /**
     * 返回编码
     *
     * @return
     */
    String getEncode();
}
```

TinyFramework 参考手册

```

    * 添加类型加载器
    *
    * @param templateLoader
    */
TemplateEngine addTemplateLoader(ResourceLoader templateLoader);
/**
    * 返回所有的 Loader
    *
    * @return
    */
List<ResourceLoader> getTemplateLoaderList();
/**
    * 渲染宏
    *
    * @param macroName 要执行的宏名称
    * @param template 调用宏的模板
    * @param context 上下文
    * @param writer 输出器
    */
void renderMacro(String macroName, Template template, TemplateContext context, Writer writer)
throws IOException, TemplateException;
/**
    * 直接渲染指定的模板
    *
    * @param macro 要执行的宏
    * @param template 调用宏的模板
    * @param context 上下文
    * @param writer 输出器
    * @throws IOException
    * @throws TemplateException
    */
void renderMacro(Macro macro, Template template, TemplateContext context, Writer writer) throws
IOException, TemplateException;

/**
    * 根据路径渲染一个模板文件
    *
    * @param path 模板对应的路径
    * @param context 上下文
    * @param writer 输出器
    */
void renderTemplate(String path, TemplateContext context, Writer writer) throws
TemplateException;
/**
    * 采用没有上下文，控制台输出方式进行渲染
    *
    * @param path
    * @throws TemplateException
    */
void renderTemplate(String path) throws TemplateException;
/**
    * 采用没有上下文，控制台输出方式进行渲染
    *
    * @param template
    * @throws TemplateException
    */
void renderTemplate(Template template) throws TemplateException;
/**
    * 直接渲染一个模板
    *
    * @param template 要渲染的模板
    * @param context 上下文

```

TinyFramework 参考手册

```
* @param writer    输出器
* @throws TemplateException
*/
void renderTemplate(Template template, TemplateContext context, Writer writer) throws
TemplateException;
/**
 * 根据宏名查找要调用的宏
 *
 * @param macroName
 * @param template
 * @param context
 * @return
 * @throws TemplateException
 */
Macro findMacro(Object macroName, Template template, TemplateContext context) throws
TemplateException;
/**
 * 执行方法
 *
 * @param functionName
 * @param parameters
 * @return
 * @throws TemplateException
 */
Object executeFunction(Template template, TemplateContext context, String functionName, Object...
parameters) throws TemplateException;
/**
 * 获取资源对应的文本
 *
 * @param path
 * @return
 */
String getResourceContent(String path, String encode) throws TemplateException;
/**
 * 获取指定路径资源的内容
 *
 * @param path
 * @return
 * @throws TemplateException
 */
String getResourceContent(String path) throws TemplateException;
/**
 * 返回是否允许缓冲
 *
 * @return
 */
boolean isCacheEnabled();
/**
 * 设置是否允许缓冲
 *
 * @param cacheEnabled
 * @return
 */
TemplateEngine setCacheEnabled(boolean cacheEnabled);
/**
 * 注册库文件中所有的宏
 *
 * @param path
 */
void registerMacroLibrary(String path) throws TemplateException;
/**
 * 注册单个宏
```

TinyFramework 参考手册

```

    *
    * @param macro
    * @throws TemplateException
    */
void registerMacro(Macro macro) throws TemplateException;
/**
 * 注册模板文件中所有的宏
 *
 * @param template
 * @throws TemplateException
 */
void registerMacroLibrary(Template template) throws TemplateException;
```


TinyFramework 参考手册

```
}
```

模板引擎提供了大量的set、get方法，方便对象注入，也方便对象读取。



小贴士

公共宏是唯一由模板引擎实际进行管控的对象，它只支持添加，不支持删除。

如果添加了重名的宏，先添加的宏将被后添加的宏所取代。

Tiny模板引擎之资源加载器

默认状态下，模板引擎不再任何的资源加载器，也就是说默认状态下，它里面不会有任何资源，所以要想模板引擎可以干活，必须为其添加资源加载器ResourceLoader才可以。

框架默认提供了如下资源加载器：

资源加载器	说明
StringResourceLoader	用于进行字符串方式的模板加载。它唯一的作用是把一段字符串构建成-
FileObjectResourceLoader	用于对各种文件系统中的模板进行加载，比如：文件、FTP、ZIP包、JAR包，
ClassLoaderResourceLoader	用于对运行环境中的已经编译后的class文件进行加载的加载器，它的存在为



提示

采用StringResourceLoader及Tiny模板引擎中的函数功能，可以方便的构建出类似Velocity的#evaluate的功能。但是请注意，这样会导致在运行时的CPU急剧升级，处理能力急剧下降。所以不建议在实际运行环境中，大批量运行的页面中利用StringResourceLoader对字符串进行处理后执行。



小贴士

资源加载器是Tiny模板引擎的核心接口，被模板引擎接口委托来进行资源加载方面的一系列处理。

当然，您也可以添加自己的资源加载器，比如：从数据库加载模板，就是个不错的好主意。

资源加载器扩展

从接口上看，资源加载器的处理还是比较复杂的。确实，凡是冠以“核心”的，往往等同于它的处理会比较复杂。为了避免开发人员扩展资源加载器的难度，还专门编写了资源加载器抽象类AbstractResourceLoader，这个时候只要继承AbstractResourceLoader类来编写资源加载器，难度就会显著降低。



小贴士

Tiny模板引擎中，内建了三种资源加载器，去看看它们就会对扩展资源加载器有了更进一步的了解。

资源加载器接口定义

```
/**
 * 用于载入模板
 * Created by luoguo on 2014/6/9.
 */
public interface ResourceLoader<T> {
    /**
     * 是否检查模板是否被修改过
     * @param checkModified
     */
    void setCheckModified(boolean checkModified);
    /**
     * 确定某个路径对应的文件是否被修改
     *
     * @param path
     * @return
     */
    boolean isModified(String path);
    /**
     * 返回路径
     *
     * @param templatePath
     * @return
     * @throws TemplateException
     */
    String getLayoutPath(String templatePath);
    /**
     * 返回模板对象，如果不存在则返回null
     *
     * @param path
     * @return
     */
    Template getTemplate(String path) throws TemplateException;
    /**
     * 返回布局对象
     *
     * @param path
     * @return
     * @throws TemplateException
     */
    Template getLayout(String path) throws TemplateException;
    /**
     * 返回宏库文件
     *
     * @param path
     * @return
     * @throws TemplateException
     */
    Template getMacroLibrary(String path) throws TemplateException;
    /**
     * 获取资源对应的文本
     *
     * @param path
     * @return
     */
}
```

TinyFramework 参考手册

```
String getResourceContent(String path, String encode) throws TemplateException;
/**
 * 添加模板对象
 *
 * @param template
 * @return
 */
ResourceLoader addTemplate(Template template) throws TemplateException;
/**
 * 创建并注册模板
 *
 * @param templateMaterial
 * @return
 */
Template createTemplate(T templateMaterial) throws TemplateException;
/**
 * 返回注入模板引擎
 *
 * @param templateEngine
 */
void setTemplateEngine(TemplateEngine templateEngine);
/**
 * 返回类加载器
 * @return
 */
ClassLoader getClassLoader();
/**
 * 设置类加载器
 * @param classLoader
 */
void setClassLoader(ClassLoader classLoader);
/**
 * 获取流程引擎
 *
 * @return
 */
TemplateEngine getTemplateEngine();

/**
 * 返回模板文件的扩展
 *
 * @return
 */
String getTemplateExtName();
/**
 * 返回布局文件的扩展名
 *
 * @return
 */
```

TinyFramework 参考手册

```
String getLayoutExtName();  
}
```



小贴士

这里之所以还有ClassLoader的注入口子，是因为要考虑模块化的因素。

Tiny模板引擎之上下文Context

上下文接口TemplateContext，它的声明如下：

```
public interface TemplateContext extends Context {  
}
```

是的，它看起来就是这么简单，但是实际上东西都在Context接口中。本来是可以不加这一层的，但是考虑到可能有些用户会做深入扩展，因此还是加了这么一层。

它的实现类也是同样的，看起来非常简单，但是实际上已经都在其父类中实现了，这里看到的的就是两个构造函数了。

```
public class TemplateContextDefault extends ContextImpl implements TemplateContext {  
    public TemplateContextDefault() {  
    }  
    public TemplateContextDefault(Map dataMap) {  
        super(dataMap);  
    }  
}
```

对于大多数用户来说，只要知道下面的两个方法就可以了：

```
put(String name, T object); // 用于向上下文中放入变量  
T get(String name); // 用于从上下文中读出变量
```



小贴士

在实际开发当中，参数都是通过外部传入的，一般来说，外部的参数不会直接是TemplateContext类型，这个时候就会涉及到如果把外部参数转变成TemplateContext对象的问题。不推荐采用创建一个TemplateContextDefault对象，然后一条一条put进去的方案，强烈建议自己实现一个类，继承TemplateContextDefault类，然后把旧的对象做为构造函数传入，然后覆盖TemplateContextDefault类中的put、get方法。这种方式，会避免从旧的数据get数据，再put到TemplateContext，从而节省时间。同时，通过这种处理，也为后续的处理增加一种可能性，他可以拿得到原生的对象。

举例来说，如果与Web集成，没有必要把Request中的所有值都读出来，put到TemplateContext中，只要做一个扩展TemplateContext，改写get方法，从request和requestAttribute读数据；而put方法，则改写成把数据写入到requestAttribute即可。

TinyFramework 参考手册

Tiny模板引擎之模板

Tiny模板引擎之宏

在Tiny模板引擎中，宏是个狠角色，用得好宏，将无往不胜、无坚不摧，用不好宏，将模板脚本繁复、难于维护。

首先看看它的庐山真面目：

```
/**
 * 宏，就可以理解为一个对方法，它有输入参数，但没有输出参数，而是直接针对Writer对象进行内容输出
 * Created by luoguo on 2014/6/6.
 */
public interface Macro {
    /**
     * 返回宏的名字
     *
     * @return
     */
    String getName();
    /**
     * 返回宏的参数名称
     *
     * @return
     */
    String[] getParameterNames();
    /**
     * 设置模板引擎
     * @param template
     */
    void setTemplate(Template template);
    /**
     * 获得模板引擎
     * @return
     */
    Template getTemplate();
    /**
     * 进行渲染
     *
     * @param context
     * @param writer
     */
    void render(Template template, TemplateContext context, Writer writer) throws TemplateException;
}
```

看起来还是比较简单的，比较值得关注的方法就三个：getName、getParameterNames、render

实际编写一个宏的实现，可以继承AbstractMacro和AbstractBlockMacro 类，这样就可以直接写几个必须的方法即可。



提示

如果需要从上下文传递参数，请用getParameterNames返回参数名列表，否则就无法进行参数传递。

TinyFramework 参考手册



小贴士

在宏里，可以通过上下文主动获取数据。

在定义宏的参数的时候，把经常用到的放在前面，把命名用概率比较低的放在后面，这样在调用时，调用者传参会比较方便。



警告

请不要起与Tiny模板引擎指令同名的宏的名字，否则在调用时会出现歧义。

如果必须起与指令同名的宏，就只能通过`#call("macroName")`或`#@call("macroName")`的方式来调用了。

虽然即使不声明，宏里也可以外部的变量，但是推荐采用参数方式传递数据。

Tiny模板引擎之宏接口定义

在Tiny模板引擎中，宏是个狠角色，用得好宏，将无往不胜、无坚不摧，用不好宏，将模板脚本繁复、难于维护。

首先看看它的庐山真面目：

TinyFramework 参考手册

```
/**
 * 宏，就可以理解为一个对方法，它有输入参数，但没有输出参数，而是直接针对Writer对象进行内容输出
 * Created by luoguo on 2014/6/6.
 */
public interface Macro {
    /**
     * 返回宏的名字
     *
     * @return
     */
    String getName();
    /**
     * 返回宏的参数名称
     *
     * @return
     */
    String[] getParameterNames();
    /**
     * 设置模板引擎
     * @param template
     */
    void setTemplate(Template template);
    /**
     * 获得模板引擎
     * @return
     */
    Template getTemplate();
    /**
     * 进行渲染
     *
     * @param context
     * @param writer
     */
    void render(Template template, TemplateContext context, Writer writer) throws TemplateException;
}
```

看起来还是比较简单的，比较值得关注的方法就三个：getName、getParameterNames、render

实际编写一个宏的实现，可以继承AbstractMacro和AbstractBlockMacro 类，这样就可以直接写几个必须的方法即可。



提示

如果需要从上下文传递参数，请用getParameterNames返回参数名列表，否则就无法进行参数传递。



小贴士

在宏里，可以通过上下文主动获取数据。



警告

请不要起与Tiny模板引擎指令同名的宏的名字，否则在调用时会出现歧义。

TinyFramework 参考手册

如果必须起与指令同名的宏，就只能通过`#call("macroName")`或`#@call("macroName")`的方式来调用了。

虽然即使不声明，宏里也可以外部的变量，但是推荐采用参数方式传递数据。

编程方式定义宏

编程方式定义HelloWorld宏

实际编写一个宏的实现，可以继承`AbstractMacro`和`AbstractBlockMacro` 类，这样就可以直接写几个必须的方法即可，下面实际演练一下编写宏，我们还是做一个**bold**的宏，用于对宏包含的内容前后增加**...**标签：

```
static class HelloWorldMacro extends AbstractMacro {
    public BoldMacro() {
        super("helloWorld");
    }
    protected void renderMacro(Template template, TemplateContext context, Writer writer) throws
IOException, TemplateException {
        writer.write("Hello, World.");
    }
}
```

简单解释：构造函数中指明，宏的名字叫：helloWorld，然后在renderMacro中输出helloWorld：

```
public static void main(String[] args) throws TemplateException {
    TemplateEngine engine = new TemplateEngineDefault();
    engine.registerMacro(new HelloWorldMacro ());
    Template template=engine.getDefaultTemplateLoader().createTemplate("#helloWorld()");
    template.render();
}
```

构建一个模板引擎，然后注册我们刚写好的宏实例，然后执行模板脚本：

```
#helloWorld()
```

下面是运行结果：

```
Hello, World.
```



小贴士

宏有两种，一种是可以嵌套内容的，一种是不可以嵌套内容的。

`AbstractMacro`用于不需要嵌套内容的宏继承，调用方式：`#macroName()`

TinyFramework 参考手册

AbstractBlockMacro 类用于需要嵌套内容的宏继承，调用方式：

```
@macroName()  
.....  
#end
```

编程方式定义bold宏

下面实际演练一下编写宏，我们还是做一个bold的宏，用于对宏包含的内容前后增加...标签：

```
static class BoldMacro extends AbstractBlockMacro {  
    public BoldMacro() {  
        super("bold");  
    }  
    protected void renderHeader(Template template, TemplateContext context, Writer writer)  
throws IOException, TemplateException {  
        writer.write("<b>");  
    }  
    protected void renderFooter(Template template, TemplateContext context, Writer writer)  
throws IOException, TemplateException {  
        writer.write("</b>");  
    }  
}
```

简单解释：构造函数中指明，宏的名字叫：bold，然后在renderHeader中输出，在renderFooter中，输出，下面看调用示例：

```
public static void main(String[] args) throws TemplateException {  
    TemplateEngine engine = new TemplateEngineDefault();  
    engine.registerMacro(new BoldMacro());  
    Template  
template=engine.getDefaultTemplateLoader().createTemplate("#@bold()HelloWorld.#end");  
    template.render();  
}
```

构建一个模板引擎，然后注册我们刚写好的宏实例，然后执行模板脚本：

```
@bold()HelloWorld.#end
```

下面是运行结果：

TinyFramework 参考手册

```
<b>HelloWorld.</b>
```



小贴士

宏有两种，一种是可以嵌套内容的，一种是不可以嵌套内容的。

AbstractMacro用于不需要嵌套内容的宏继承，调用方式：`#macroName()`

AbstractBlockMacro 类用于需要嵌套内容的宏继承，调用方式：

```
#@macroName()  
.....  
#end
```

模板方式定义宏

模板方式定义bold宏

实际编写一个bold的宏，用于对宏包含的内容前后增加**...**标签：

```
#macro bold()  
<b>#bodyContent</b>  
#end
```

简单解释：定义宏bold，然后在调用bold宏时，在其包含内容前面放****，后面放****：

构建一个模板引擎，然后执行模板脚本：

```
#@bold()HelloWorld.#end
```

下面是运行结果：

```
<b>HelloWorld.</b>
```



小贴士

宏有两种，一种是可以嵌套内容的，一种是不可以嵌套内容的。

TinyFramework 参考手册

AbstractMacro用于不需要嵌套内容的宏继承，调用方式：`#macroName()`

AbstractBlockMacro 类用于需要嵌套内容的宏继承，调用方式：

```
#@macroName()  
.....  
#end
```

相比起通过编程实现，用模板的的方式添加，相当简单。

模板方式定义helloWorld宏

实际编写一个helloWorld的宏，用于输出“Hello, World.”：

```
#macro helloWorld()  
Hello, World.  
#end
```

简单解释：定义宏bold，然后在调用bold宏时，输出“Hello, World.”：

下面是运行结果：

```
<b>HelloWorld.</b>
```



小贴士

宏有两种，一种是可以嵌套内容的，一种是不可以嵌套内容的。

AbstractMacro用于不需要嵌套内容的宏继承，调用方式：`#macroName()`

AbstractBlockMacro 类用于需要嵌套内容的宏继承，调用方式：

```
#@macroName()  
.....  
#end
```

相比起通过编程实现，用模板的的方式添加，相当简单。

TinyFramework 参考手册

宏调用方式

一个宏，不管是编程的方式创建的还是通过`#macro`指令来创建的，其作用与使用方式完全相同，没有任何区别，因此其调用方式也是一致的。

在Tiny模板引擎中，可以通过宏的名字直接调用，也可以通过`#call`指令的方式进行调用。

下面假设定义的宏的名字是`hello`，来说明调用方式：

	无包含内容调用	有包含内容调用
#call指令方式	<code>#call("hello",...)</code>	<code>#@call("hello",...)</code> ... <code>#end</code>
#macroName方式	<code>#hello(...)</code>	<code>#hello(...)</code> ... <code>#end</code>

调用宏的时候，可以带参，也可以不带参数。

在带参数的时候，Tiny模板引擎有两种传参形式，一种是按顺序方式，一种是命名方式：

	按顺序方式	命名方式	说明
规范	<code>expression</code>	<code>identify:expression</code>	<code>expression</code> 是指Tiny的表达式
示例	<code>1</code> <code>"abc"</code> <code>name</code> <code>user.name</code> <code>user.getName()</code>	<code>value:1</code> <code>value:"abc"</code> <code>value:name</code> <code>value:user.name</code> <code>value:user.getName()</code>	

顺序参数传递和命名传递方式可以结合使用，但是推荐采用前面是顺序传递的参数，后面是命名传递的方式。


设置定义下面的`hello`宏，包含了5个函数

TinyFramework 参考手册


```
#macro hello(a, b, c, d, e)
...
#end
```

调用示例：

调用方式	解释	备注
#hello(1, 2, 3, 4, 5)	表示调用#hello的时候，a=1, b=2, c=3, d=4, e=5	
#hello(1, 2)	表示调用#hello的时候，a=1, b=2其它值未设定	
#hello(e:5)	表示调用#hello的时候，e=5	
#hello(e:5, d:4)	表示调用#hello的时候，d=4, e=5其它值未设定	
#hello(e:5, 2, 3)	表示调用#hello的时候，b=2, c=3, e=5其它值未设定	

警告

一定要分清楚命名传参和顺序传参上的差异，否则会出现逻辑问题。

小贴士

命名传参用于宏的参数比较多，但是在实际使用的时候，有可能不是全部有用，这个时候就可以采用命名传参的方式。


#macroName方式适合于名字确切的场景。

#call(expression)方式，适合于名字需要动态计算的场景。

Tiny模板引擎之函数

对于Tiny模板引擎来说，函数是非常有用且易于扩展的。

Tiny模板引擎中函数有两种，一种是表达式函数，一种是类型扩展函数。

名称解释

表达式函数：就是可以在表达中使用的函数，使用方式：functionName(...)

类型扩展函数：就是可以为某种类型扩展出的成员函数，使用方式object.functionName()....

Tiny模板引擎之函数接口定义

TinyFramework 参考手册

模板函数接口

```
/**
 * 模板函数扩展
 * Created by luoguo on 2014/6/9.
 */
public interface TemplateFunction {
    /**
     * 绑定某种类上，使之成为这些类型的成员函数
     *
     * @return
     */
    String getBindingTypes();
    /**
     * 返回函数名，如果有多个名字，则用逗号分隔
     *
     * @return
     */
    String getNames();
    /**
     * 设置模板引擎
     *
     * @param templateEngine
     */
    void setTemplateEngine(TemplateEngine templateEngine);
    /**
     * 返回模板引擎
     * @return
     */
    TemplateEngine getTemplateEngine();
    /**
     * 执行函数体
     *
     * @param parameters
     * @return
     */
    Object execute(TemplateContext context, Object... parameters) throws TemplateException;
}
```

Tiny模板引擎之函数扩展

看上面的接口声明，方法还是比较多的，幸运的是框架在其之上，实现了AbstractTemplateFunction类，把常用的方法已经实现，如果继承AbstractTemplateFunction类来进行函数扩展，就非常容易了。

下面看一个实类，来添加一个format函数：

TinyFramework 参考手册

```
public class FormatterTemplateFunction extends AbstractTemplateFunction{
    private Formatter formatter=new Formatter();
    public FormatterTemplateFunction() {
        super("fmt,format,formatter");
    }
    public Object execute(TemplateContext context, Object... parameters) throws TemplateException {
        if(parameters.length==0||!(parameters[0] instanceof String)){
            notSupported(parameters);
        }
        String formatString = parameters[0].toString();
        Object[] objects = Arrays.copyOfRange(parameters, 1, parameters.length);
        return formatter.format(formatString, objects);
    }
}
```

一共有两个方法，构造方法调用父类构造方法，传入要注册的函数名，如果要注册多个名字，可以用半角逗号“,”进行分隔。

execute方法就是真正的执行方法了，它传入一个可变参数，需要实现者读取参数并进行处理，最终返回结果。

Tiny模板引擎之类成员函数扩展

下面看一个实类，来添加一个format函数：

```
public class FormatterTemplateFunction extends AbstractTemplateFunction{
    private Formatter formatter=new Formatter();
    public FormatterTemplateFunction() {
        super("fmt,format,formatter");
    }
    public Object execute(TemplateContext context, Object... parameters) throws TemplateException {
        if(parameters.length==0||!(parameters[0] instanceof String)){
            notSupported(parameters);
        }
        String formatString = parameters[0].toString();
        Object[] objects = Arrays.copyOfRange(parameters, 1, parameters.length);
        return formatter.format(formatString, objects);
    }
}
```

一共有两个方法，构造方法调用父类构造方法，传入要注册的函数名，如果要注册多个名字，可以用半角逗号“,”进行分隔。

execute方法就是真正的执行方法了，它传入一个可变参数，需要实现者读取参数并进行处理，最终返回结果。

下面也展示一个给字符串函数扩展一个加粗效果的函数：

TinyFramework 参考手册

```
static class BoldFunction extends AbstractTemplateFunction{
    public BoldFunction() {
        super("bold");
    }
    public String getBindingTypes() {
        return "java.lang.String";
    }
    @Override
    public Object execute(Object... parameters) throws TemplateException {
        return "<b>" + parameters[0].toString() + "</b>";
    }
}
```

上面的例子中，首先定义一个BoldFunction，使得注意的是，相比较于上次的format，它增加了这个方法：

```
public String getBindingTypes() {
    return "java.lang.String";
}
```

这表示这个方法会绑定到String类上，可以同时为多个类绑定方法，只要用半角的“,”对类型名进行分隔即可。

在执行逻辑处，可以看到，只是简单的在第一个参数周两边加上了“”和“”。

```
public static void main(String[] args) throws TemplateException {
    TemplateEngine engine = new TemplateEngineDefault();
    engine.addTemplateFunction(new BoldFunction());
    Template
    template=engine.getDefaultTemplateLoader().createTemplate("${format('Hello,%s','world').bold()}");
    template.render();
}
```

上面构建引擎，注册方法，然后执行下面的模板脚本：

```
${format('Hello,%s','world').bold()}
```

它的涵义是先用format函数对字符串“hello,%s”用参数“world”进行格式化，格式化来的结果调用其成员函数bold()进行处理，于是就获得了下面的运行结果：

```
<b>Hello,world</b>
```


TinyFramework 参考手册

布局在Tiny模板引擎中也是一个非常重要的概念，通过Tiny模板引擎，可以快速进行页面构建与渲染，同时可以为程序员做减少，越到底层的程序员，所要完成的工作越少。同常见的模板引擎中的布局实现相比，Tiny模板引擎中的布局，具有一定的强制性，同时也会减少程序员的开发工作量，程序员不再关心布局文件的引入，你放入有布局文件的目录，就会应用布局；你把它拿出来，它也就没有了。

Tiny模板引擎的布局，其方法结构与普通模板文件完全相同，只是它的扩展名与模板文件名不同，用以进行区别，至于起个什么名字，就是模板引擎使用者的自由了。

布局文件规范

布局文件与模板文件完全相同，只是在布局文件中需要放置一个`{pageContent}`标签，用以标示插入点。



小贴士

Tiny模板引擎支持多重布局，也就是说，如果在从当前模板所在文件夹到根目录“/”的路径上，不管有多少个符合条件的布局文件，都会进行渲染。

Tiny模板引擎布局文件的选择方式为：如果有与模板路径的同名的布局文件(仅扩展名不同)，则优先选取同名布局文件；如果没有同名布局文件就选取default布局文件。

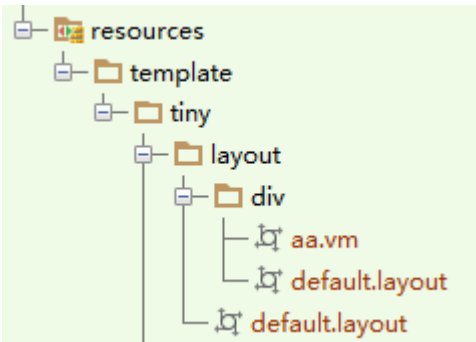


提示

需要注意，布局文件的标示是`{pageContent}`，而宏中插入点的标示为`#bodyContent`或`{bodyContent}`。

Tiny模板引擎布局之入门示例

为了更好的理解Tiny模板引擎中的布局，直接用示例说明，下面是目录结构：



/template/tiny/layout/div目录中aa. vm的内容如下：

```
Hello, World
```

/template/tiny/layout/div目录中default. layout的内容如下：

TinyFramework 参考手册

```
<b>
    ${pageContent}
</b>
```

/template/tiny/layout目录中的default.layout的内容如下:

```
<div>
    ${pageContent}
</div>
```

这个时候, 如下编写测试代码:

```
public class TinyTemplateTestCase {
    public static void main(String[] args) throws TemplateException {
        final TemplateEngine engine = new TemplateEngineDefault();
        FileObjectResourceLoader tinySample = new FileObjectResourceLoader("vm", "layout", null,
"src/test/resources");
        engine.addTemplateLoader(tinySample);
        engine.renderTemplate("/template/tiny/layout/div/aa.vm");
    }
}
```

下面是执行结果:

```
<div>
    <b>
        Hello, World
    </b>
</div>
```

也就是说, 虽然aa.vm文件里只有Hello, World, 但是由于在与它同名路径中有一个布局文件, 因此就用布局文件对其进行了渲染, 从而变成下面的内容:

```
<b>
    Hello, World
</b>
```

结果他上层目录中另外还有一个布局文件/template/tiny/layout/default.layout, 于是他就被渲染成最终的结果:

```
<div>
    <b>
        Hello, World
    </b>
</div>
```



小贴士

TinyFramework 参考手册

Tiny模板引擎中的布局文件是不必由模板文件进行显式声明的。

Tiny模板引擎中的布局文件的命名规范为：模板基本文件名+“.”+布局文件扩展名，比如：

i 模板文件名为：aa.vm，则模板基本文件名为aa，其布局文件扩展名为layout，这个时候它的布局文件名就是：aa.layout

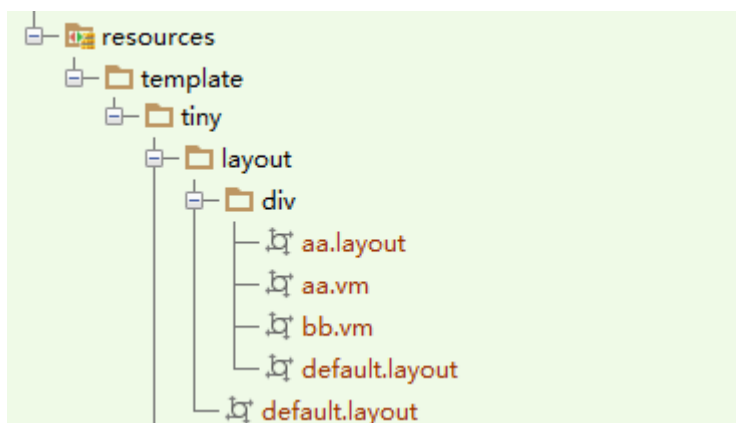
Tiny模板引擎中有一个特殊的布局文件名，它就是default+“.”+布局文件扩展名，比如：

i 布局文件扩展名为layout，这个时候它的布局文件名就是：default.layout

Tiny模板引擎布局之进阶示例

上一节演示一布局文件的使用，这一节展示一种高级的用法，详细请看下面的示例：

下面是文件目录结构



/template/tiny/layout/div目录中aa.vm的内容如下：

```
Hello, World
```

/template/tiny/layout/div目录中bb.vm的内容如下：

```
Hello, 悠然
```

/template/tiny/layout/div目录中default.layout的内容如下：

```
<b>
    ${pageContent}
</b>
```

/template/tiny/layout/div目录中aa.layout的内容如下：

TinyFramework 参考手册

```
<h3>
    ${pageContent}
</h3>
```

/template/tiny/layout目录中的default.layout的内容如下：

```
<div>
    ${pageContent}
</div>
```

这个时候，如下编写测试代码：

```
public class TinyTemplateTestCase {
    public static void main(String[] args) throws TemplateException {
        final TemplateEngine engine = new TemplateEngineDefault();
        FileObjectResourceLoader tinySample = new FileObjectResourceLoader("vm", "layout", null,
"src/test/resources");
        engine.addTemplateLoader(tinySample);
        engine.renderTemplate("/template/tiny/layout/div/aa.vm");
        System.out.println("=====");
        engine.renderTemplate("/template/tiny/layout/div/bb.vm");
    }
}
```

下面是执行结果：

```
<div>
    <h3>
        Hello, World
    </h3>
</div>
=====
<div>
    <b>
        Hello, 悠然
    </b>
</div>
```

聪明的你一定明白了，布局文件的渲染过程是这样的，优先渲染与模板文件同名的布局文件，只有不存在与模板文件同名的布局文件的时候，才会渲染default.layout布局文件。



小贴士

从Tiny模板引擎的设计者来说，一般是不推荐采用与模板同名的布局文件的，也就是应该尽量避免这种情况的出现。但是从另外一个方面考虑，在某种特殊情况下，可能需要对于某个特殊页面开开绿灯，做一点与众不同的事情，所以可以通过增加同名布局的方式来做个个性化的事情。

Tiny模板引擎之国际化

Tiny模板引擎对于国际化在设计时就给予了考虑。

TinyFramework 参考手册

— Tiny的模板国际化，考虑了两种国际化方式，一种是采用国际化资源进行替换的方式，这种方式适合于改动内容不大的方式，只要把一些标题内容进行替换即可；另外一种整页替换的方式，也就是同一个页面用不同的语言写多次，不同的语种的人来的时候，用不同的页面进行渲染。

但是不管采用哪种方式，都要实现下面的接口。

I18nVisitor

```
public interface I18nVisitor {
    /**
     * 获取当前位置
     * @param context
     * @return
     */
    Locale getLocale(TemplateContext context);
    /**
     * 返回国际化资源
     * @param context
     * @param key
     * @return
     */
    String getI18nMessage(TemplateContext context,String key);
}
```

如果需要进行国际化，只要实现上面的接口，并设置给TemplateEngine即可。



小贴士

如果是整页替换方式getI18nMessage方法可以不做处理。

如果是整页替换的方式，页面的命名规则如下：

页面类型	页面扩展名	页面名	文件名	说明
模板	page	about	about.page	在访问时，如果检测到对应Locale的模板页面则访问默认的about.page页面。
			about_zh_CN.page	
			about_zh_TW.page	
布局	layout	index	index.layout	在访问时，如果检测到对应Locale的页面则访问默认index.layout页面。
			index_zh_CN.page	
			index_zh_TW.page	

TinyFramework 参考手册

宏库文件	component	common	common.component common_zh_CN.component common_zh_TW.component	在编写宏时，需要在宏的名字 比如有个宏是 <code>#macro hello() Hello #end !</code> 当在调用hello时，就会调用宏 hello_zh_CN
------	-----------	--------	--	--



小贴士

不带语言编码的文件，必须存在，它将作为默认的渲染文件存在。

系统内嵌函数

系统内嵌函数是指在模板引擎中，默认就会注册的函数，直接可以拿来使用的函数。

宏调用方法call, callMacro

宏调用方法(call, callMacro)，用于执行一个宏，并把执行完成的结果作为字符串返回。

函数调用方式：

宏调用方法

```
call("macroName")
call("macroName", {aaa:1, bbb:2})
callMacro("macroName")
callMacro("macroName", {aaa:1, bbb:2})
函数的返回值为宏的运行结果。
```

示例：

```
${call("macroName")}
等价于#call("macroName")
造价于#macroName()
```

格式化函数fmt, format, formatter

格式化函数(fmt, format, formatter)，用于对数据进行格式化，并返回执行结果。

函数调用方式：

宏调用方法

三个函数的作用完全相同。

```
format(formatString,...)
```

返回值：格式化后结果，字符串

示例：

```
#set(result=format("hello,%s",name))
${format("hello,%s",name)}
```



小贴士

format函数的底层实现是调用了java.util.Formatter实现的，因此具体如何填写，格式化串，请参考：[java.util.Formatter用法](#) 章节。

java.util.Formatter用法

有时会想把数字，日期，字符串按照给定规则给格式化。SUN

JDK 为我们提供了这个API，它是java.util.Formatter。此类提供了对布局对齐和排列的支持，以及

对数值、字符串和日期/时间数据的常规格式和特定于语言环境的输出的支持。

如何格式化？

给定规则：

要想按照自己的想法格式化必须事先编写一个规则。那这个规则要怎么编写？

1. 常规类型、字符类型和数值类型的格式说明符的语法如下：

```
%[argument_index$][flags][width][.precision]conversion
```

2. 用来表示日期和时间类型的格式说明符的语法如下：

```
%[argument_index$][flags][width]conversion
```

3. 与参数不对应的格式说明符的语法如下：

```
%[flags][width]conversion
```

API中有这样三种规则，很显然第一个规则的内容是最全面的。其它规则的内容和第一规则的内容有重复，那单说第一规则内容，其它规则依次类推。

注意：规则一中的precision前面要加英文句号“.”

API有以下解释：

可选的 argument_index 是一个十进制整数，用于表明参数在参数列表中的位置。第一个参数由“1\$” 引用，第二个参数由“2\$” 引用，依此类推。

可选 flags 是修改输出格式的字符集。有效标志集取决于转换类型。

TinyFramework 参考手册

- 可选 `width` 是一个非负十进制整数，表明要向输出中写入的最少字符数。
- 可选 `precision` 是一个非负十进制整数，通常用来限制字符数。特定行为取决于转换类型。
- 所需 `conversion` 是一个表明应该如何格式化参数的字符。给定参数的有效转换集取决于参数的数据类型。
- `argument_index`很好理解，就是一参数占位符，用来表示要被格式化的参数。
- `flags`

标志	常规	字符	整数	浮点	日期/时间	说明
'-'	y	y	y	y	y	结果将是左对齐的。
'#'	y1	-	y3	y	-	结果应该使用依赖于转换类型的替换形式
'+'	-	-	y4	y	-	结果总是包括一个符号
' '	-	-	y4	y	-	对于正值，结果中将包括一个前导空格
'0'	-	-	y	y	-	结果将用零来填充
','	-	-	y2	y5	-	结果将包括特定于语言环境的 组分分隔符
'('	-	-	y4	y5	-	结果将是用圆括号括起来的负数

- 1 取决于 [Formattable](#) 的定义。
 - 2 只适用于 `'d'` 转换。
 - 3 只适用于 `'o'`、`'x'` 和 `'X'` 转换。
 - 4 对 [BigInteger](#) 应用 `'d'`、`'o'`、`'x'` 和 `'X'` 转换时，或者对 `byte` 及 [Byte](#)、`short` 及 [Short](#)、`int` 及 [Integer](#)、`long` 及 [Long](#) 分别应用 `'d'` 转换时适用。
 - 5 只适用于 `'e'`、`'E'`、`'f'`、`'g'` 和 `'G'` 转换。
- 任何未显式定义为标志的字符都是非法字符，并且都被保留，以供扩展使用。
- `width` 就表示一最少字符数，被格式化参数用`precision` 截取器截取后与`width` 相比，被格式化参数

TinyFramework 参考手册

— 字符数如果小于width，则加字符则到等于width。如果大于则width不起作用。所以可以叫width为少加多过器。

precision 在上面也提到了，precision 是一个截取器，用于截取被格式化参数。

conversion 转换可分为以下几类：

常规- 可应用于任何参数类型

字符- 可应用于表示 Unicode 字符的基本类型：char、[Character](#)、byte、[Byte](#)、short 和 [Short](#)。当 [Character.isValidCodePoint\(int\)](#) 返回 true 时，可将此转换应用于 int 和 [Integer](#) 类型

数值

1 整数- 可应用于 Java 的整数类型：byte、[Byte](#)、short、[Short](#)、int、[Integer](#)、long、[Long](#) 和 [BigInteger](#)

2 浮点- 可用于 Java 的浮点类型：float、[Float](#)、double、[Double](#) 和 [BigDecimal](#)

日期/时间- 可应用于 Java 的、能够对日期或时间进行编码的类型：long、[Long](#)、[Calendar](#) 和 [Date](#)。

百分比- 产生字面值 '%' (' \u0025')

行分隔符- 产生特定于平台的行分隔符

常规	B b	如果参数 arg 为 null，则结果为 "false"。如果 arg 是一个 boolean，则结果为 String.valueOf() 返回的字符串。否则结果为 "true"。
	H h	如果参数 arg 为 null，则结果为 "null"。否则，结果为调用Integer arg.hashCode()) 得到的结果。
	S s	如果参数 arg 为 null，则结果为 "null"。如果 arg 实现 Format arg.formatTo。否则，结果为调用 arg.toString() 得到的结果。
字符	C c	结果是一个 Unicode 字符
整数	d	结果被格式化为十进制整数
	o	结果被格式化为八进制整数
	X x	结果被格式化为十六进制整数
浮点	E e	结果被格式化为用计算机科学记数法表示的十进制数

TinyFramework 参考手册

	f	结果被格式化为十进制数
	G g	根据精度和舍入运算后的值，使用计算机科学记数形式或十进制格式对结果
	A a	结果被格式化为带有效位数和指数的十六进制浮点数
日期，日间	T t	日期和时间转换字符的前缀
百分比	%	结果为字面值 ' % '
行分隔符	n	结果为特定于平台的行分隔符

常出现的异常

java.util.IllegalFormatConversionException: d != java.lang.Double
：被格式化参数类型与规则转换类型不对应。

java.util.FormatFlagsConversionMismatchException: Conversion = d, Flags = #
#：flag不适用于规则转换类型。

Formatter类是用正则表达式验证给定规则的

正则表达式如下：

```
private static final String formatSpecifier = "%(\\d+\\$)?([-#+0,(\\<]*)?(\\d+)?(\\.|\\d+)?([tT])?([a-zA-Z%])";
```

总结：

- 最重要的是它可以格式化日期/时间, 数值和字符。
- 可以把日期转换成年，月，日，星期等。可以为数值填充空格或0，添加分组字符，正负号，及小括号。我们可以在System.out.format(),String.format()方法中直接应用Formatter类。

求值函数eval, evaluate

求值 函数(eval，evaluate)，用于执行一段宏代码，并返回执行结果。

函数调用方式：

宏调用方法

eval(~模板内容~)
evaluage(templateContentVarName)
返回值：执行后的结果，字符串

示例：

TinyFramework 参考手册

```
#set(result=eval("hello, ${name}"))
${eval("hello, ${name}")}
```

Tiny模板引擎扩展

Tiny模板引擎之示例

从HelloWorld了解Tiny模板引擎

要用Tiny模板引擎做一个HelloWorld是非常简单的：

```
public final class TinyTemplateHelloWorld {
    public static void main(String[] args) throws TemplateException {
        TemplateEngine engine = new TemplateEngineDefault();
        ResourceLoader<String> resourceLoader=new StringResourceLoader();
        engine.addTemplateLoader(resourceLoader);
        Template template=resourceLoader.createTemplate("HelloWorld");
        template.render();
    }
}
```

说明：

第一步新建一个模板引擎，第二步，创建一个字符串模板加载器并添加到模板引擎中，然后利用“Hello World”字符串来创建一个模板，然后进行渲染，下面是运行结果：

```
HelloWorld
```

但是这个引擎还还初级，只能执行普通字符串形式的模板，这当然不够，我们实际运行当中，当然也需要从文件加载模板。

从文件系统加载HelloWorld模板

当然了，实际应用当中，不可能所有的模板都来自字符串，来自文件是一种再自然不过的情况。

这个时候，Tiny模板引擎就不够用了，初始化的时候就需要多加一行代码：

```
public final class TinyTemplateHello1World {
    public static void main(String[] args) throws TemplateException {
        TemplateEngine engine = new TemplateEngineDefault();
        engine.addTemplateLoader(new FileObjectResourceLoader("html", null, null,
"src\\main\\resources\\templates"));
        engine.renderTemplate("/helloworld.html");
    }
}
```

说明：

第一步新建一个模板引擎，第二步，添加一个文件类型模板加载器到引擎，然后对此文件系统中的模板

TinyFramework 参考手册

进行渲染了，下面是运行结果：

```
HelloWorld
```



小贴士

```
new FileObjectResourceLoader("html", null, null,  
"D:\\git\\ebm\\src\\main\\resources\\templates")
```

这里的参数分别是：模板文件扩展名，布局文件扩展名，宏公共库文件扩展名，宏加载根目录。

由于这里不需要展现而已及宏公共库文件扩展名，因此给了两个空值。

宏加载根目录的设置非常重要，它表示了宏文件的相对根所在的位置。

递归调用示例

```
#macro printNumber(number)  
    ${number}#eol  
    #if(number<10)  
        #printNumber(number+1)  
    #end  
#end  
#printNumber(1)
```

运行结果：

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

多层宏调用示例

TinyFramework 参考手册

```
#macro firstMacro()
<div>
    #bodyContent
</div>
#end
#macro secondMacro()
<b>
    #bodyContent
</b>
#end
#@firstMacro()
    #@secondMacro()
    Information
    #end
#end
#@secondMacro()
    #@firstMacro()
    Information
    #end
#end
```

运行结果:

```
<div><b>Information</b></div><b><div>Information</div></b>
```

宏定义中调用宏示例

```
#macro firstMacro()
<div>
    #@secondMacro()
    #bodyContent
    #end
</div>
#end
#macro secondMacro()
<b>
    #bodyContent
</b>
#end
#@firstMacro()
    Information
#end
```

运行结果:

```
<div><b>Information</b></div>
```



小贴士

这个功能是非常必要的，但是在

TinyFramework 参考手册

Velocity及许多宏语言中都是无法支持的，自己动手丰衣足食，现在终于可以这样子玩了。

多次循环调用示例

```
#for(i:[1..2])
  #for(j:[1..2])
    ${i}*${j}=${i*j}#eol
  #end
#end
```

运行结果：

```
1*1=1
1*2=2
2*1=2
2*2=4
```

输出内容缩进示例

```
#for(i:[1..2])
${i}#eol
#[
  #for(j:[1..2])
#t${i}*${j}=${i*j}#eol
  #end
#]
#end
```

运行结果：

```
1
  1*1=1
  1*2=2
2
  2*1=2
  2*2=4
```



小贴士

在许多模板语言中，要进行严格的格式化控制都几乎是不可能的，而在Tiny模板引擎中对此则有强有力支持，可以做到随心所欲生成格式。

Tiny模板引擎之最佳实践 布局之实践

TinyFramework 参考手册

我们现在的布局，只有一个标准的页面占位符，就是\$pageContent或\${pageContent}，这样是比较适合于整页进行布局的方式。但是很多的时候，在上层而已文件中，有时需要有多多个渲染点需要在后续页面中才能确定展现内容，这个时候，我们默认的一个标准的页面点位符就不够了。

与其它模板引擎对比

Tiny模板引擎 vs Velocity1.7

与Velocity指令差异

velocity	Tiny 模板引擎	异同	功能	变化
<code>\${foo.bar}</code> <code>\$foo.bar</code>	<code>\${foo.bar}</code>	相同	输出占位符	Tiny 模板引擎 大括号必需，不支持 <code>\$foo.bar</code> 格式。里面表达式计算中，变量不必再加\$前缀
<code>!\${foo.bar}</code> <code>!\$foo.bar</code>	<code>!\${foo.bar}</code>	不同	空值不显示源码	velocity 为空值不显示源码 Tiny 模板引擎 改为 HTML 过滤输出
<code>## ...</code> <code>#* ... *#</code>	<code>## ...</code> <code>#-- ... --#</code> <code>#* ... *#</code>	相同	注释	增加一种新的块注释格式： <code>#-- --#</code>
<code>#[[...]]</code>	<code>#[[...]]</code>	相同	不解析文本块	
<code>\# \\${\$}</code>	<code>\# \\${\$}</code>	相同	特殊字符转义	
<code>#set(\$foo = \$bar)</code>	<code>#set(foo = bar)</code>	相同	给变量赋值	
<code>#if(\$foo == \$bar)</code>	<code>#if(foo == bar)</code>	相同	条件判断	
<code>#elseif(\$foo == \$bar)</code>	<code>#elseif(foo == bar)</code>	相同	否定条件判断	
<code>#else, #{else}</code>	<code>#else, #{else}</code>	相同	否定判断	
<code>#end, #{end}</code>	<code>#end, #{end}</code>	相同	结束指令	
<code>#foreach(\$item in \$list)</code>	<code>#foreach(item : list)</code> <code>#for(item in list)</code> <code>#foreach(item in list)</code> <code>#for(item : list)</code>	相同	循环指令	扩展与 java 兼容的方式 可以用 for，也可以用 foreach 可以用 ":" 也可以用 "in"
<code>#break</code>	<code>#break</code> <code>#break(foo == bar)</code>	相同	中断循环	可以直接带条件
n/a	<code>#continue</code> <code>#continue(foo == bar)</code>	新增	继续下一个循环	可以直接带条件
<code>#stop</code>	<code>#stop</code> <code>#stop(foo == bar)</code>	相同	停止模板解析	可以直接带条件
<code>#macro(foo)</code>	<code>#macro foo(...)</code>	相似	可复用模板片段宏	区别 1: Velocity 的宏名字在括号里面，Tiny 模板引擎的宏名字在括号外面。 区别 2: Velocity 的嵌入内容占位符为： <code>\$bodyContent</code> ，Tiny 模板引擎的嵌入内容占位符为 <code>#bodyContent</code>
<code>#include("foo.txt")</code>	<code>read("foo.txt")</code> <code>read("foo.txt", "UTF-8")</code>	相似	读取文本文件内容	Tiny 模板引擎 用扩展函数实现，如果不指定编码，则采用模板引擎中指定的编码格式。
<code>#parse("foo.vm")</code>	<code>#include("foo.jetx")</code> <code>#include("foo.jetx", {aa:1,bb:2})</code>	相同	包含另一模板输出	Tiny 模板引擎 支持参数传递，传方式为一个 Map，key 对应 Value 为一个具体的参数

TinyFramework 参考手册

velocity	Tiny模板引擎	异同	功能	
<code>\${foo.bar}</code> <code>\$foo.bar</code>	<code>\${foo.bar}</code>	相同	输出占位符	Tiny模板
<code>!\${foo.bar}</code> <code>!\$foo.bar</code>	<code>!\${foo.bar}</code>	不同	空值不显示源码	velocity Tiny模板
<code>## ...</code> <code>*# ... *#</code>	<code>## ...</code> <code>#-- ... --#</code> <code>*# ... *#</code>	相同	注释	增加一行
<code>#[[...]]</code>	<code>#[[...]]</code>	相同	不解析文本块	
<code>\# \\$ \\\</code>	<code>\# \\$ \\\</code>	相同	特殊字符转义	
<code>#set(\$foo = \$bar)</code>	<code>#set(foo = bar)</code>	相同	给变量赋值	
<code>#if(\$foo == \$bar)</code>	<code>#if(foo == bar)</code>	相同	条件判断	
<code>#elseif(\$foo == \$bar)</code>	<code>#elseif(foo == bar)</code>	相同	否定条件判断	
<code>#else, #{else}</code>	<code>#else, #{else}</code>	相同	否定判断	
<code>#end, #{end}</code>	<code>#end, #{end}</code>	相同	结束指令	
<code>#foreach(\$item in \$list)</code>	<code>#foreach(item : list)</code> <code>#for(item in list)</code> <code>#foreach(item in list)</code> <code>#for(item : list)</code>	相同	循环指令	扩展与 可以用 可以用
<code>#break</code>	<code>#break</code> <code>#break(foo == bar)</code>	相同	中断循环	可以直接
n/a	<code>#continue</code> <code>#continue(foo == bar)</code>	新增	继续下一个循环	可以直接

TinyFramework 参考手册

#stop	#stop #stop(foo == bar)	相同	停止模板解析	可以直
#macro(foo)	#macro foo(...)	相似	可复用模板片段宏	区别1: 区别2:
#include("foo.txt")	read("foo.txt") read("foo.txt" , "UTF-8")	相似	读取文本文件内容	Tiny模
#parse("foo.vm")	#include("foo.jetx") #include("foo.jetx", {aa:1,bb:2})	相同	包含另一模板输出	Tiny模
#evaluate("\${1 + 2}")	eval或evaluate函数。	相同	支持模板内容	Veloc: Tiny模
n/a	#call(expression, args)	扩展	动态调用宏	动态调 module 则可以 #call #call

与Velocity语法差异

- Tiny模板引擎 指令中的变量不加 \$ 符，只支持 #if(x == y)，不支持 #if(\$x == \$y)，因为指令中没有加引号的标识就是变量，和常规语言的语法一样，加"\$"有点重复，不注意还容易引起错误。
- Tiny模板引擎 占位符必需加大括号，只支持 \${foo}，不支持 \$foo，因为 \$ 在 JavaScript 中也是合法变量名符号，而 \${} 不是，减少混淆，也防止意义上的二义性，比如\$aa.bb+1，可能有的多种解释\${aa}.bb+1、\${aa.bb}+1、\${aa.bb+1}。
- Tiny模板引擎 占位符当变量为 null 时输出空白串，而不像 Velocity 那样原样输出指令原文，即\${foo}，等价于 Velocity 的 \${!{foo}}。在 Tiny模板引擎 中，\${!{foo}} 表示对内容进行 HTML 过滤，用于原样输出 HTML 片段。
- Tiny模板引擎 支持在所有使用变量的地方，进行表达式计算，也就是你不需要像 Velocity 那样，先 #set(\$j = \$i + 1) 到一个临时变量，再输出临时变量 \${j}，Tiny模板引擎 可以直接输出\${i + 1}，其它指令也一样，比如：#if(i + 1 == n)。
- Tiny模板引擎支持采用扩展 Class 原生方法的方式，如：\${"str".toChar()}，而不像 Velocity 的 Tool 工具方法那样：\$(StringTool.toChar("a"))，这样的调用方式更直观，更符合代码书

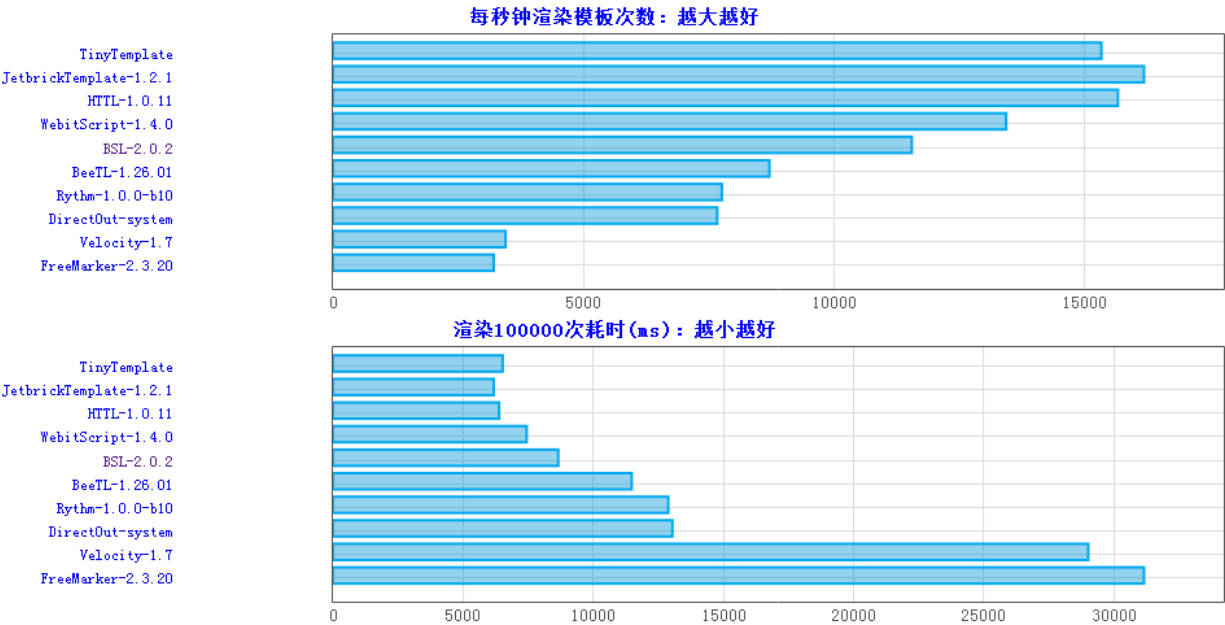
TinyFramework 参考手册

写习惯。

- Tiny模板引擎 支持属性和方法的安全调用。如 `${user?.name}`，`${user?.hasRole("vip")}`。如果user 对象为 null，那么返回结果就是 null，不会出现NullPointerException。

与Velocity性能差异

Tiny模板引擎的执行效率大致是Velocity的4~5倍，可谓完胜。



Tiny模板引擎 vs JetbrickTemplate

与Jetbrick指令差异

相同+：表示Tiny框架不仅完全兼容，且有所扩展。

相同-：表示基本相同，但又有所区别。

jetbrick	Tiny模板引擎	异同	功能	变化
<code>\${foo.bar}</code> <code>\$foo.bar</code>	<code>\${foo.bar}</code>	相同	输出占位符	
<code>\${!foo.bar}</code> <code>!foo.bar</code>	<code>!{foo.bar}</code>	相同	空值不显示源码	均为HTML转码输出
<code>## ...</code> <code>## ... ##</code>	<code>## ...</code> <code>## ... ##</code> <code>* ... *</code>	相同+	注释	增加一种新注释 以便于与Velocity兼容
<code>#[[...]]</code>	<code>#[[...]]</code>	相同	不解析文本块	
<code>\# \\$ \</code>	<code>\# \\$ \</code>	相同	特殊字符转义	
<code>#set(\$foo = \$bar)</code>	<code>#set(foo = bar)</code> <code>!set(foo = bar)</code>	相同+	给变量赋值	<code>!set</code> 用于向调用者赋值
<code>#if(\$foo == \$bar)</code>	<code>#if(foo == bar)</code>	相同	条件判断	
<code>#elseif(\$foo == \$bar)</code>	<code>#elseif(foo == bar)</code>	相同	否定条件判断	
<code>#else, #else()</code>	<code>#else, #{else}</code>	相同-	否定判断	<code>#else</code> 的分隔方式

TinyFramework 参考手册

<code>#end, #end()</code>	<code>#end, #{end}</code>	相同-	结束指令	#end的分隔方式:
<code>#foreach(\$item : \$list)</code>	<code>#foreach(item : list)</code> <code>#for(item in list)</code> <code>#foreach(item in list)</code> <code>#for(item : list)</code>	相同+	循环指令	Tiny多出一
<code>#break</code>	<code>#break</code> <code>#break(foo == bar)</code>	相同	中断循环	可以直接带条件
<code>#continue</code>	<code>#continue</code> <code>#continue(foo == bar)</code>	相同	继续下一个循环	可以直接带条件
<code>#stop</code>	<code>#stop</code> <code>#stop(foo == bar)</code>	相同	停止模板解析	可以直接带条件
<code>#macro foo()</code>	<code>#macro foo(...)</code>	相同+	可复用模板片段宏	区别: Jetbr
<code>#include("foo.txt", parameters)</code>	<code>#include("foo.txt", parameter)</code>	相同	包含另一模板输出	
<code>read("foo.txt")</code>	<code>#read("foo.txt")</code> <code>#read("foo.txt", "UTF-8")</code>	相同+	读取资源内容	支持指定编辑格:
n/a	eval或evaluate函数。	扩展	支持模板内容	
n/a	<code>#call(expression, args)</code>	扩展	动态调用宏	动态调用宏 module为模: 则可以写: <code>#call(forma</code> <code>#call(modu.</code>
<code>\${macroName(parameters)}</code>	<code>#macroName(parameters)</code> <code>@macroName(parameters)</code> <code>#end</code>	相同+	调用宏方式	jetbrick有点像 Tiny模板引擎则:
<code>#define(Type foo = bar)</code>		不同	给变量声明类型	Jetbrick为强类:
<code>#put(name, value)</code>	<code>#!set(name1=value1, name2=value2)</code>	相同	保存变量到全局	Tiny模板没: Jetbrick采

TinyFramework 参考手册

<code>#tag foo(...)</code> <code>#end</code>	<code>##macroName(parameters)</code> <code>#end</code>	相同+	自定义标签	采用编码的方式:
---	---	-----	-------	----------

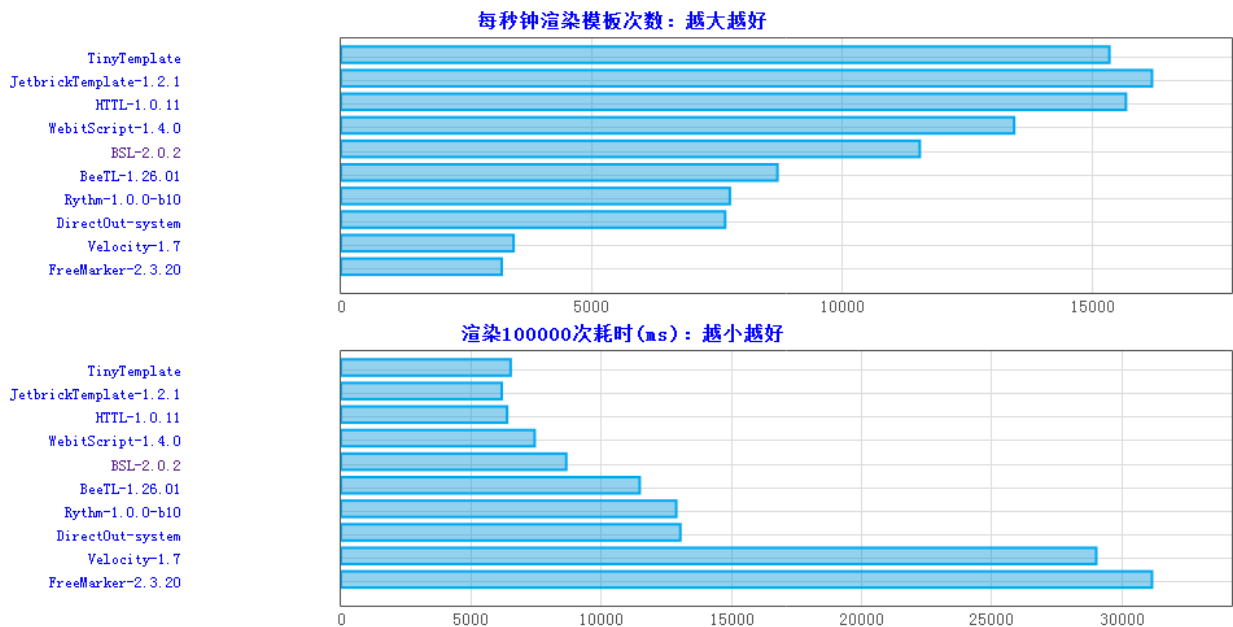
与Jetbrick语法差异

由于Tiny模板引擎的语法方面，大量借鉴自Jetbrick，因此指令语法完全相同。

与Jetbrick性能差异

Jetbrick采用强类型，因此在编译时，变量的类型都是明确的，因此编译后的代码更接近Java语言，因此效率会更高，Tiny框架模板引擎采用弱类型，因此只有在运行时，才知道具体的类型是什么，因此变量只能放在上下文中，因此效率会稍低。

从下面的图中也可以看到，JetBrick及HTTL这种强类型方式的性能较弱类型的性能是高的,但上从图中也可以看出，Tiny模板引擎的性能差异距离强类型方式的，也差距不大。



Tiny模板引擎之常见问题解答

Tiny模板引擎中模板已经编译成类，发布时是否可以不发布模板源文件？

当然可以！只要把生成的类文件打成jar包，并放在运行环境中，然后添加一个ClassLoaderResource即可！

Tiny模板引擎中模板的执行过程是怎样的？

一般来说模板引擎的执行方式有两种，一种是解释，一种是编译。

Tiny模板引擎采用了编译方式，即先把模板文件预编译为java文件，然后再编译Java文件为class文件，最后执行class文件。

Tiny模板引擎可以和Spring集成么？

Tiny模板引擎的构建完全可以通过Spring的Bean容器进行管理，实际上也推荐这么做。

Tiny模板引擎支持多实例运行么？

实际上Tiny模板引擎在设计之初考虑了必须可以支持在多实例模式下运行，因此Tiny模板引擎完全可以在多

TinyFramework 参考手册

—实例模式下运行。

Tiny模板引擎有配置文件么？

Tiny模板引擎在设计之时就抱着大道至简的思想，摒弃了配置文件的做法。同时也是为了避免大多实例模式下，一些配置上的冲突。

但是对于使用Tiny模板引擎的用户来说，完全可以按照自己的需求来设计自己的配置文件。

为什么明明我写的模板文件是存在的，却抛出找不到模板异常？

1. 查看是否有添加模板资源加载器？
2. 查看模板资源加载器构造函数中的模板文件扩展名与你的模板文件扩展名是否一致？（小数点后的部分）
3. 查看模板资源加载器的根目录与模板文件所在目录是否是包含关系？
4. 查看模板资源路径是否以"/"开始的路径？
5. 查看模板模板加载器Root路径+资源路径是否与文件路径相一致？

可不可以从宏里传数据给调用的模板？

宏没有返回值机制，因此没有返回值可供返回。一般情况下，在宏里用#set变量设置，只会影响到当前宏内部。

但是Tiny模板引擎考虑到应用实际需要，可以通过#!set指令把值设置到模板的上下文里，就可以被模板的后续处理使用了。

在布局文件中可不可以访问当前模板的变量？

当然可以，在所有层次的布局文件中，都可以访问当前模板上的变量，这也就为一些渲染效率提供了方便。

比如在布局文件中编写下面的内容：

```
XX系统- ${pageTitle}
```

在具体的页面中编写：

```
#set(pageTitle="首页")
```