Jetbrick主页

jetbrick QQ 交流圈：310491655

文 档 下 载 源 码

概述-快速了解

1 概述 Overview
2 简单易用的指令
3 卓越性能 Performance
4 易于集成 Integrate
5 友好的错误提示

1 概述 Overview

jetbrick-template 是一个新一代 Java 模板引擎，具有高性能和高扩展性。 适合于动态 HTML 页面输出或者代码生成，可替代 JSP 页面或者 Velocity 等模板。 指令和 Velocity 相似，表达式和 Java 保持一致，易学易用。

- 支持类似于 Velocity 的多种指令
- 支持静态编译
- 支持编译缓存
- 支持热加载
- 支持类型推导
- 支持泛型
- 支持可变参数方法调用
- 支持方法重载
- 支持类似于 Groovy 的方法扩展
- 支持函数扩展
- 支持自定义标签 #tag
- 支持宏定义 #macro
- 支持布局 Layout

2 简单易用的指令

jetbrick-template 指令集和老牌的模板引擎 Velocity 非常相似，易学易用。

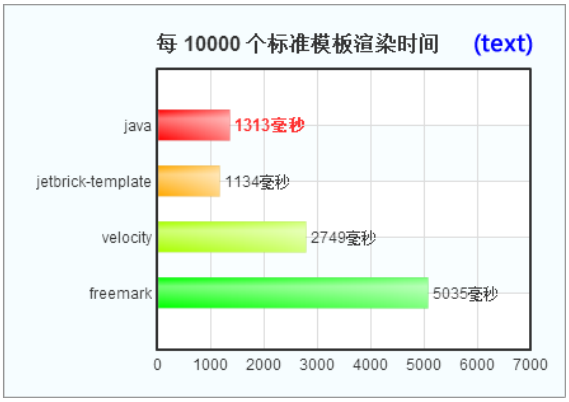
```
#define(List<UserInfo> userlist)
<table>
  <tr>
    <td>序号</td>
    <td>姓名</td>
    <td>邮箱</td>
  </tr>
  #for (UserInfo user : userlist)
  <tr>
    <td>${for.index}</td>
    <td>${user.name}</td>
    <td>${user.email}</td>
  </tr>
  #end
</table>
```

详细指令语法，请参考：语法指南。或者和 Velocity 的比较。

3 卓越性能 Performance

jetbrick-template 将模板编译成 Java ByteCode 运行，并采用强类型推导，无需反射和减少类型转换。渲染速度等价于 Java 硬编码。比 Velocity 等模板快一个数量级。 比 JSP 也快，因为 JSP 只有 Scriptlet 是编译的，Tag 和 EL 都是解

释执行的。 而 jetbrick-template 是全编译的。



在 Stream 模式中(Webapp 采用 OutputStream 将文本输出到浏览器)，由于 Java 硬编码输出字符串需要进行一次编码的转换。 而 jetbrick-template 却在第一次运行期间就缓存了编码转换结果，使得 jetbrick-template 的性能甚至优于 Java 硬编码。

具体测试用例，请参考：Template Engine Benchmark Test (platform: Window 7 x64, Intel i5, 16GB RAM, JDK 1.6.0_41 x64)

4 易于集成 Integrate

可以和市面上常见的 Web MVC framework 进行集成。

- HttpServlet
- Filter
- Struts 2.x
- Spring MVC
- JFinal
- Nutz
- Jodd

具体集成方法，请参考： Web 框架集成

也可以和 Spring Ioc 进行集成，请参考：如何在 Spring 中使用 JetEngine

5 友好的错误提示

具有详细的模板解析和编译错误提示，出错提示可以定位到原始模板所在的行号。

```
22:14:51.271 [main] INFO  (JetTemplate.java:68) - Loading template source file: D:\workspace\github\jetbrick-sche
22:14:51.406 [main] ERROR (JetTemplateErrorListener.java:27) - Template parse failed:
D:\workspace\github\jetbrick-schema-app\bin\config\report\schema.html.jetx:37
message: The method getColumnNam() or isColumnNam() is undefined for the type jetbrick.schema.app.model.TableColu
33:         </tr>
34:     #for(TableColumn c: t.columns)
35:         <tr style="background-color:white;">
36:             <td>${c.displayName}</td>
37:             <td>${c.columnNam}</td>
                        ^^^^^^^^^

Exception in thread "main" jetbrick.commons.exception.SystemException: java.lang.RuntimeException: The method get
  at jetbrick.commons.exception.SystemException.unchecked(SystemException.java:23)
  at jetbrick.commons.exception.SystemException.unchecked(SystemException.java:12)
  at jetbrick.schema.app.TemplateEngine.apply(TemplateEngine.java:44)
  at jetbrick.schema.app.Task.writeFile(Task.java:83)
  at jetbrick.schema.app.task.SqlReportTask.execute(SqlReportTask.java:19)
  at jetbrick.schema.app.SchemaGenerateApp.taskgen(SchemaGenerateApp.java:56)
  at jetbrick.schema.app.SchemaGenerateApp.main(SchemaGenerateApp.java:74)
Caused by: java.lang.RuntimeException: The method getColumnNam() or isColumnNam() is undefined for the type jetbr
  at jetbrick.template.parser.JetTemplateCodeVisitor.reportError(JetTemplateCodeVisitor.java:1388)
```

```
    at jetbrick.template.parser.JetTemplateCodeVisitor.visitExpr_field_access(JetTemplateCodeVisitor.java:665)
    at jetbrick.template.parser.JetTemplateCodeVisitor.visitExpr_field_access(JetTemplateCodeVisitor.java:1)
...
```

- 出错模板：D:\workspace\github\jetbrick-schema-app\bin\config\report\schema.html.jetx
- 出错行号：37
- 错误原因：The method getColumnNam() or isColumnNam() is undefined for the type jetbrick.schema.app.model.TableColumn.

点击这里查看：jetbrick-template 常见错误分析

2014-01-05：jetbrick-template-1.2.0 正式版已发布，新增预编译功能。

1 开源许可 License
2 第三方依赖包 Dependences
3 Maven 依赖 POM.xml
4 从源码安装 Sources
5 范例下载 Samples
6 离线文档下载 Documents
7 最新版本 Latest Version
8 更新历史 Release Notes

1 开源许可 License

2 第三方依赖包 Dependences

- JDK 1.6+
- ANTLR runtime 4.x
- Slf4j 1.7.x

> **注意**：
>
> jetbrick-template 需要使用 jdk6+ 里面的 tools.jar 来编译模板生成的 Java 源文件。如果只有 jre6+，那么可以 copy 一个对应的 tools.jar 到 classpath 下面。
> 从 1.1.2 开始，也支持 JDT（Eclipse Java Compiler）编译 java。只需要把 org.eclipse.jdt.core_xxx.xxx.jar (又名 ecj.jar) 放在 classpath 下面就可。（无需配置，在 jdk 自带的 Compiler 没法使用的情况下，会自动搜索 JDT）

3 Maven 依赖 POM.xml

已发布到 Maven 中央库： http://central.maven.org/maven2/com/github/subchen/

```
<project>
  <dependencies>
    <dependency>
      <groupId>com.github.subchen</groupId>
      <artifactId>jetbrick-template</artifactId>
      <version>1.2.0</version>
    </dependency>
    <dependency>
      <groupId>org.antlr</groupId>
      <artifactId>antlr4-runtime</artifactId>
```

```
      <version>4.1</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.7.5</version>
    </dependency>
  </dependencies>
<project>
```

pom.xml for JDT.

```
<dependency>
  <groupId>org.eclipse.tycho</groupId>
  <artifactId>org.eclipse.jdt.core</artifactId>
  <version>3.9.1.v20130905-0837</version>
</dependency>
```

4 从源码安装 Sources

github: https://github.com/subchen/jetbrick-template

编译方法：

1. 先安装 apache-ant 1.9.x

```
wget http://mirrors.cnnic.cn/apache//ant/binaries/apache-ant-1.9.2-bin.zip
```

2. 设置好 JDK, ANT 环境变量

```
set JAVA_HOME=/path/jdk_1.6.x
set ANT_HOME=/path/apache-ant_1.9.x
set PATH=%JAVA_HOME%/bin;%ANT_HOME%/bin;%PATH%
```

3. 编译

```
git clone https://github.com/subchen/jetbrick-template.git
cd jetbrick-template
ant dist
```

4. 编译后的文件存放在 build 目录中

```
jetbrick-template-x.x.x.jar
jetbrick-template-x.x.x.zip
jetbrick-template-x.x.x-all.zip
```

5 范例下载 Samples

5.0.1 官方范例：

- jetx-samples-servlet.zip
- jetx-samples-jfinal.zip
- jetx-samples-springmvc.zip
- jetx-samples-struts.zip
- jetx-samples-jodd.zip

下载的 zip 包中包含完整的源代码和可直接运行的 war 包。
更多代码可以前往： https://github.com/subchen/jetbrick-template-webmvc-samples/

5.0.2 网友提供的范例：

- jetx-samples-nutz-by-howe.zip
- jetx-samples-springmvc-by-yingzhuo.zip

5.0.3 自定义标签 Tag：

- 自定义标签 Tags for Spring Security
- 自定义标签 Tags for Shiro

感谢网友 应卓 提供相关 Tag 实现。

6 离线文档下载 Documents

jetbrick-template-1.1.3-documents.pdf

感谢网友 laughing 提供离线 PDF 文档。

7 最新版本 Latest Version

时间	版本	二进制包	完整包	更新历史
2014-01-05	1.2.0	jetbrick-template-1.2.0.zip	jetbrick-template-1.2.0-all.zip	更新历史
2013-12-22	1.1.3	jetbrick-template-1.1.3.zip	jetbrick-template-1.1.3-all.zip	更新历史
2013-12-15	1.1.2	jetbrick-template-1.1.2.zip	jetbrick-template-1.1.2-all.zip	更新历史
2013-12-08	1.1.1	jetbrick-template-1.1.1.zip	jetbrick-template-1.1.1-all.zip	更新历史
2013-12-02	1.1.0	jetbrick-template-1.1.0.zip	jetbrick-template-1.1.0-all.zip	更新历史
2013-11-22	1.0.2	jetbrick-template-1.0.2.zip	jetbrick-template-1.0.2-all.zip	更新历史
2013-11-20	1.0.1	jetbrick-template-1.0.1.zip	jetbrick-template-1.0.1-all.zip	更新历史
2013-11-18	1.0.0	jetbrick-template-1.0.0.zip	jetbrick-template-1.0.0-all.zip	更新历史

8 更新历史 Release Notes

Version 1.2.0 (2014-01-05)

- [新增] #38 增加默认的 #tag cache() 实现模板局部缓存功能
- [新增] #49 增加模板预编译工具/选项
- [新增] #54 增加安全管理器：黑白名单
- [新增] #62 在 Web 环境中使用 jetx 时候，建议增加一个隐藏变量
- [新增] #63 对 Array/List/Map 的 [] 访问，增加安全调用
- [新增] #64 Spell error in JetAnnoations Class name, Should be JetAnnotations.
- [新增] #65 给 #for 指令内部对象增加 for.odd 和 for.even 支持.

Version 1.1.3 (2013-12-22)

- [新增] #50 增加 Jodd Madvoc 的集成支持
- [新增] #56 增加 MultipathResourceLoader，支持多个模板路径
- [增强] #52 增强 asDate() 方法，默认支持更多的格式，比如 ISO8601, RFC 822
- [增强] #55 对#for指令的增强建议
- [增强] #57 增强 template.path 和 compile.path 的配置功能
- [增强] #58 为JetEngineFactoryBean提供构造注入方式的spring配置
- [修复] #53 engine.createTemplate("你好") 编译失败
- [修复] #59 JetUtils.asBoolean() 对 Collection 和 Map 的判断有误
- [修复] #60 #if (obj == null) 报错，invalid token null
- [修复] #61 import.variables 如果存在多个泛型类型定义会报错

Version 1.1.2 (2013-12-15)

- [新增] #32 增加 annotation 自动扫描查找 Methods / Functions / Tags
- [新增] #43 新增 JDT 编译方法，以应对没有 JDK 的环境
- [新增] #51 增加 #tag default_block(name) 默认 layout block 的实现
- [改进] #48 compile.path 配置路径移除 jetx_x_x_x 的路径后缀
- [修复] #44 启用 trim.directive.comments 的时候，出现 NullPointerException
- [修复] #45 #tag 中的 #include 输出的内容位置不正确
- [修复] #46 通过 classpath 加载 jar 中模板，出现 ResourceNotFoundException
- [修复] #47 tomcat 目录中带空格，template 编译失败

Version 1.1.1 (2013-12-08)

- [新增] #28 增加 asDefault() 方法扩展，支持设置默认值
- [新增] #30 增强 #put，一次支持多个变量的传递
- [新增] #31 增加 Spring FactoryBean 的集成支持
- [新增] #35 增加 #tag block(name) 默认实现，配合 #include 实现多个内容块的 layout
- [增强] #39 增强 #tag layout 功能，允许添加自定义参数给 layout 模板
- [修复] #20 The column of error line is wrong when the line contains '\t'
- [修复] #24 三元表达式如果使用 Interface 或者 Primitive Class 作为选项，会出现 NullPointerException
- [修复] #25 #if #else #end 语句后面貌似丢了一个换行
- [修复] #27 #set指令创建double型字面变量时，小数点后面跟0则不能通过编译
- [修复] #29 如果没有 #if 只有独立的 #else 或者 #end，没有报错，且剩余内容会被省略掉
- [修复] #33 属性安全调用问题？
- [修复] #34 拼写错误: #tag layout 中的实现用的是 bodyContext, 文档中描述的是 bodyContent，不一致
- [修复] #37 throw NullPointerException when method parameter is null.
- [修复] #40 #form 和 #for 指令冲突，编译失败
- [修复] #41 从 request uri 中获取模板路径存在问题，会出现404错误
- [修复] #42 include() 函数和 #tag layout() 传的 Map 参数出现编译错误

Version 1.1.0 (2013-12-02)

- [新增] #12 增加自定义 Tag 功能
- [新增] #13 增加 #macro 宏定义
- [新增] #15 增加对类的静态字段和静态方法的直接访问
- [新增] #18 增加默认的 layout Tag 实现
- [新增] #19 与Nutz集成，实现JetTemplateView (Thanks wendal1985@gmail.com)
- [修复] #14 如果运算符的操作数的返回值是 void, 那么就会出现编译错误
- [修复] #20 The column of error line is wrong when the line contains '\t'
- [修复] #21 NumberUtils.format(123) should be "123.00"
- [修复] #22 对于 ${bean.property}，优先使用 getXXX()
- [修复] #23 Fixed request uri in JetTemplateServlet/JetTemplateFilter

Version 1.0.2 (2013-11-22)

- [新增] #10 增加选项：compile.always 第一次访问模板强制编译
- [新增] #16 允许 import 一个单独的 Class, 避免出现冲突
- [新增] #17 增加 iterator(start,stop,step) 代替 range(...) 函数
- [增强] #9 如果 compile.path 对应的目录非法或者没有权限不可写, 应该启动Engine时就报错
- [修复] #8 jetx 模板生成的 java 文件名可能会产生冲突
- [修复] #11 模板的路径如有使用 "../../file.jetx" 那么就会访问到 template.path 路径的外面

Version 1.0.1 (2013-11-20)

- [新增] #1 支持 Servlet 2.x
- [新增] #4 增加指令注释支持，如： 增强对可视化编辑器的友好度
- [修复] #2 trim.directive.line 选项，如果指令两边为非空格，也会被 trim
- [修复] #3 compile.debug 默认应该为 false
- [修复] #6 JDK 6 can't load the template class compiled using JDK 7.
- [修复] #7 JetTemplateServlet 和 JetTemplateFilter 默认可能输出错误的 contentType.

Version 1.0.0 (2013-11-18)

- 支持类似于 Velocity 的多种指令
- 支持静态编译
- 支持编译缓存
- 支持热加载
- 支持类型推导
- 支持泛型
- 支持可变参数方法调用
- 支持方法重载
- 支持类似于 Groovy 的方法扩展
- 支持函数扩展
- First public release

---

1 从这里开始 Start
1.1 基本步骤 Steps
2 核心对象 Core
2.1 JetEngine
2.2 JetTemplate
2.3 JetContext
3 高级用法
3.1 方法扩展 Methods
3.2 函数扩展 Functions
3.3 自定义标签 Tags
4 错误处理 Finding Issue
4.1 语法错误 Syntax Error
4.2 编译错误 Compile Error
4.3 运行期错误 Runtime Error
4.4 如何调试模板 debug ?

1 从这里开始 Start

1.1 基本步骤 Steps

1. 创建自定义配置的 `JetEngine` 对象。推荐使用单例模式创建。
2. 根据模板路径，获取一个模板对象 `JetTemplate`。
3. 创建一个 `Map<String, Object>` 对象，并加入你的 data objects。
4. 准备一个待输出的对象，`OutputStream` 或者 `Writer`。
5. 根据你的 data objects 来渲染模板，并获得输出结果。

具体的 Java 代码，看上去是这样的：

```
// 创建一个默认的 JetEngine
JetEngine engine = JetEngine.create();

// 获取一个模板对象
JetTemplate template = engine.getTemplate("/sample.jetx");

// 创建 context 对象
Map<String, Object> context = new HashMap<String, Object>();
context.put("user", user);
context.put("books", books);

// 渲染模板
StringWriter writer = new StringWriter();
template.render(context, writer);

// 打印结果
System.out.println(writer.toString());
```

整个过程，是不是非常简单？

下面将介绍几个 API 的核心对象：`JetEngine`，`JetTemplate`，`JetContext`

2 核心对象 Core

2.1 JetEngine

整个模板引擎的由 `JetEngine` 驱动，不同的 `JetEngine` 对象可以使用不同的配置。一般在一个 Application 或者 Webapp 中，我们只需要一个 `JetEngine` 对象就可以了，我们推荐使用单例模式创建。

2.1.1 如何创建 JetEngine ?

1. `JetEngine.create()`

在 classpath 根目录下面自动查找 `jetbrick-template.properties` 文件。如果文件不存在，则使用默认配置。

2. `JetEngine.create(File)`

   从用户指定的 `File` 文件中加载系统配置，该文件必须是一个 `.properties` 文件。

3. `JetEngine.create(Properties)`

   从用户指定的 `Properties` 对象中加载系统配置。

有哪些配置？ 看这里所有的配置

2.1.2 获取 JetTemplate 对象

通过下面的方法获取 `JetTemplate` 对象。

```
public JetTemplate getTemplate(String name) throws ResourceNotFoundException;
```

我们也可以获得一个 `Resource` （模板文件或者非模板文件），或者判断一个 `Resource` 是否存在。

```
public boolean lookupResource(String name);
public Resource getResource(String name) throws ResourceNotFoundException;
```

> **注意**：对于一个 resource 或者 template 的 name，应该以 `/` 开头，并且以 `/` 作为分隔符，如：`/templates/index.jetx`。

2.1.3 从源码中直接创建模板

```
public JetTemplate createTemplate(String source);
```

比如：

```
JetTemplate template = engine.createTemplate("${1+2*3}");
UnsafeCharArrayWriter out = new UnsafeCharArrayWriter();
template.render(new JetContext(), out);
Assert.assertEquals("7", out.toString());
```

注意： createTemplate() 每次都会编译生成新的 JetTemplate 对象，如果需要缓存，请自行维护。

2.2 JetTemplate

对应于一个模板文件，通过 `JetEngine.getTemplate(name)` 获取。在第一次获取的时候，会先将模板生成对象的 `.java` 文件，然后在将 `.java` 文件编译成 `.class` 文件。

如果模板不存在，则抛出 `ResourceNotFoundException`。

然后通过下面几个方法可以对模板进行渲染：

```
public void render(Map<String, Object> context, Writer out);
public void render(Map<String, Object> context, OutputStream out);
public void render(JetContext context, Writer out);
public void render(JetContext context, OutputStream out);
```

我们可以使用 `Map<String, Object>` 或者 `JetContext` 存储我们的 data objects。`JetContext` 是对 `Map<String, Object>` 的简单封装。

> **注意**：
>
> - `context` 对象在模板运行期间，并不会受到模板污染，即数据不会被改变（保证数据的无侵入性）。

2.3 JetContext

用来存储和获取模板关联的 data objects。可以通过 `new JetContext()` 或者 `new JetContext(map)` 创建。

使用 `JetContext` 就像使用 Java 的 `HashMap` 一样。常用的方法如下：

```
public Object get(String name);
public void put(String name, Object value);
public void putAll(Map<String, Object> context);
```

> **注意**：
>
> - `JetContext` 会被 `#put` 指令修改
> - 用户提供的 `JetContext` 不会受到 `#set` 的影响，但是内部的使用的 `JetContext` 对象会受到 `#set` 指令的影响。
> - `JetContext` 会在父子模板调用的时候，形成一个 Context Chain，子模板可以自动获取父模板的变量，而父模板无法看到子模板的 `JetContext`。但是子模板可以通过 `#put(name, value)` 来修改父模板的 `JetContext`。具体查看：如何嵌入子模板？

3 高级用法

上面只是简单的介绍了一下 `jetbrick-template` 的基本用法，下面将介绍一些高级用法，也是 `jetbrick-template` 有别于其他模板引擎的特色。

3.1 方法扩展 Methods

我们知道一个 Java Class 的 所有 methods 都是定义在同一个 class 文件中的，不能在其他地方进行动态扩展。但是一些其他动态语言却支持在 Class 外部为这个 Class 增加一些方法。比如：

- JavaScript 的 prototype 机制
- Groovy 的 metaClass 机制
- JetBrains 的 Kotlin

jetbrick-template 也在这里带给大家强大的动态方法扩展机制。如：`"123".asInt()`，`new Date().format("yyyy-MM-dd")`。

> **注意**：如果 Class 已经定义了同名方法，则优先使用 Class 定义的方法。但是扩展方法支持方法重载(Overrload)。

方法扩展支持 2 种模式：

- 上下文无关方法：MethodTool.method(bean, ...)
- 上下文相关方法：MethodTool.method(bean, JetPageContext, ...)

3.1.1 上下文无关方法 MethodTool.method(bean, ...)

- 方法签名必须是 `public` 和 `static`
- 方法的第一个参数类型必须是要扩展的 Class
- 方法其余参数自定义

示例：对 `String` 进行扩展

```
public class StringMethods {
  public static String link(String text, String url) {
    return "<a href=\"" - url - "\">" - text - "</a>";
  }
}
```

然后需要把扩展的 `StringMethods` 注册到 `JetEngine`。

```
// 把 StringMethods 加入到 engine 中
Properties config = new Properties();
config.put(JetConfig.IMPORT_METHODS, StringMethods.class.getName());
JetEngine engine = JetEngine.create(config);
...
```

模板：

```
${"BAIDU".link("http://www.baidu.com/")}
```

输出结果：

```
<a href="http://www.baidu.com/">BAIDU</a>
```

3.1.2 上下文相关方法 MethodTool.method(bean, JetPageContext, ...)

如果扩展的方法需要用到 template 相关联的运行时信息 `JetPageContext`，那么我们就需要扩展一个上下文相关的 method。

和上下文无关的扩展方法相比，上下文相关的扩展方法多一个参数。

- 方法签名必须是 `public` 和 `static`
- 方法第一个参数类型是要扩展的 Class
- **方法第二个参数类型必须是 JetPageContext**
- 方法其余参数自定义

```
public class UserInfoMethods {
    public static String isOnline(UserInfo user, JetPageContext ctx) {
        HttpSession session = (HttpSession) ctx.getContext().get(JetContext.SESSION_NAME);
        return session.getAttribute("user_" - user.getName()) != null;
    }
}

// 把 UserInfoMethods 加入到 engine 中
Properties config = new Properties();
config.put(JetConfig.IMPORT_METHODS, UserInfoMethods.class.getName());
JetEngine engine = JetEngine.create(config);
...
```

模板：

```
#define(UserInfo user)
${user.isOnline()}
```

3.2 函数扩展 Functions

jetbrick-template 还支持函数扩展，如 `${now()}`，`${include("tag.jetx")}`。

- 上下文无关函数：任意参数
- 上下文相关函数：第一个参数必须是 JetPageContext

示例：

```
public class Functions {
  // 上下文无关函数
  public static String today(String format) {
    return new SimpleDateFormat(format).format(new Date());
  }
  // 上下文相关函数
```

```
  public static String hello(JetPageContext ctx) {
    return "Hello " - ctx.getContext().get("name");
  }
}

// 把 Functions 加入到 engine 中
Properties config = new Properties();
config.put(JetConfig.IMPORT_FUNCTIONS, Functions.class.getName());
JetEngine engine = JetEngine.create(config);
...
```

模板：

```

${today("yyyy-MM-dd")}
${hello()}
```

> **注意**：函数和扩展方法的唯一区别是少了第一个扩展类型的参数，其他的都一样。

3.3 自定义标签 Tags

jetbrick-template 自定义标签 Tag，类似于 JSP Taglib，但是要比 JSP Taglib 更简单更好用。

示例：

```

public class Tags {
  public static void cache(JetTagContext ctx, String name, int timeout) throws IOException {
    Cache cache = CacheManager.getCache(); // 请用自己的 Cache 代替
    Object value = cache.get(name);
    if (value == null) {
      value = ctx.getBodyContext();
      cache.put(name, value, timeout);
    }
    ctx.getWriter().print(value);
  }
}
```

对于每一个 Tag 的方法声明，有如下要求：

- 方法签名必须是 `public` `static`
- 方法返回值必须是 `void`
- 方法第一个参数必须是 `JetTagContext` ，其余参数自定义
- 允许 throws 任意的 `Throwable`
- 允许定义相同名字的 Tag，但是方法参数不一样（Overload）
- 支持可变参数 (VarArgs)

然后需要把自定义的 Tags 注册到 `JetEngine`。

```

// 把 Tags 加入到 engine 中
Properties config = new Properties();
config.put(JetConfig.IMPORT_TAGS, Tags.class.getName());
JetEngine engine = JetEngine.create(config);
...
```

模板：

```

#tag cache("sum", 10)
    计算结果将被缓存10秒： ${1+2+3+4+5+6+7+8+9}
#end
```

具体可以参考：jetbrick-template 中如何自定义 Tag？

4 错误处理 Finding Issue

`jetbrick-template` 提供了强大的错误定位功能，你再也不用担心找不到错误原因了。

4.1 语法错误 Syntax Error

模板示例：

```
#for (user in userlist)
<tr>
    <td>${for.index}</td>
    <td>${user.name}</td>
    <td>${user.roles.asHTML()}</td>
</tr>
#end
```

错误提示：(错误所在的行号和列号，错误模板路径，错误原因等)

```
22:14:51.406 [main] ERROR (JetTemplateErrorListener.java:27) - Template parse failed:
C:\Users\Sub\AppData\Local\Temp\jetx_1_0_0\template\sample.java:5
message: The method asHTML() is undefined for the type List.
1. #for (user: userlist)
2. <tr>
3.     <td>${for.index}</td>
4.     <td>${user.name}</td>
5.     <td>${user.roles.asHTML()}</td>
                      ^^^^^^
```

4.2 编译错误 Compile Error

这种错误正常情况下是不会发生的，如果发生这种情况，请到这里 open issues。

但是如果发生这样的错误，也可以得到下面的类似错误提示。

```
Exception in thread "main" java.lang.IllegalStateException: Compilation failed.
C:\Users\Sub\AppData\Local\Temp\jetx_1_0_0\template\debug_jetx.java:13: 'void' type not allowed here
  11:     JetWriter $out = $ctx.getWriter();
  12:     JetContext context = $ctx.getContext();
  13:     $out.print(("1"+JetFunctions.debug("aaa"))); // line: 1
                     ^
1 error(s)
```

我们可以从打印出来的编译错误中，可以看到大部分源代码后面都会有一个 `// line: XXX` 的注释，这个就是生成的 java 代码对应原始模板文件的行号映射。这样我们就能找到原始模板的错误行数了。

模板示例：

```
1: ${"1"+debug("aaa")}
```

生成的 Java 代码示例：

```
$out.print(("1"+JetFunctions.debug("aaa"))); // line: 1
```

4.3 运行期错误 Runtime Error

如果在模板运行期间发生错误，那么就可以得到类似下面的错误 Java Exception Stack。

错误例子模板如下：

```
```

```
#set (arraylist = ["a","b","c","d"])
#for (int x : arraylist)
    ${x}
#end
```

获得的运行期错误 Java Exception Stack 如下：

```

generateJavaClass: C:\Users\Sub\AppData\Local\Temp\jetx_1_0_0\template\for_loop_list_jetx.class
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer
    at template.for_loop_list_jetx.render(for_loop_list_jetx.java:14)
    at jetbrick.template.JetTemplate.render(JetTemplate.java:125)
    at jetbrick.template.JetTemplate.render(JetTemplate.java:115)
    at testcase.JetEngineTestCase.test(JetEngineTestCase.java:36)
    at testcase.JetEngineTestCase.main(JetEngineTestCase.java:64)
```

根据错误所在行(for_loop_list_jetx.java:14)，我们查看生成的 Java Source。

```

11:    List arraylist = (List) Arrays.asList("a","b","c","d"); // line: 1
12:    Iterator<?> $it_3 = JetUtils.asIterator(arraylist);
13:    while ($it_3.hasNext()) { // line: 2
14:      Integer x = (Integer) $it_3.next();
15:      $out.print($txt_4, $txt_4_bytes);
```

然后根据 Java Source 中对应的行数，知道这个是一个 `#for` 指令，查看生成的注释(`// line: 2`)，就能找到对应的原始模板所在的错误行号是第二行： `#for (int x : arraylist)` 。

至此，我们就能知道错误的原因是 `arraylist` 是一个 `List<Object>` ，里面的每个元素是 `String` ，强制类型转换成 `int` 失败导致的。正确的模板语句应该是 `#for (String x : arraylist)` 。

4.4 如何调试模板 debug ?

4.4.1 使用 debug(format, args...) 函数

范例：

```
${debug("id = {}, users.size = {}.", id, users.size()}
```

> 注意：
>
> 1. 要使用 debug 函数，需要 Slf4j 配合，在对应的 log 实现中打开 debug.
> 2. 具体的 format 参数格式请查看 Slf4j Logger。

开启 debug 日志：

- Log4j:

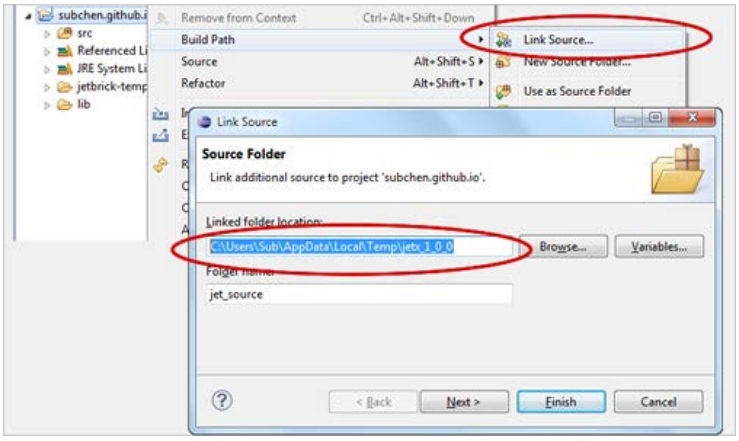
```
log4j.logger.jetbrick.template.runtime.JetUtils = DEBUG
```

- Logback

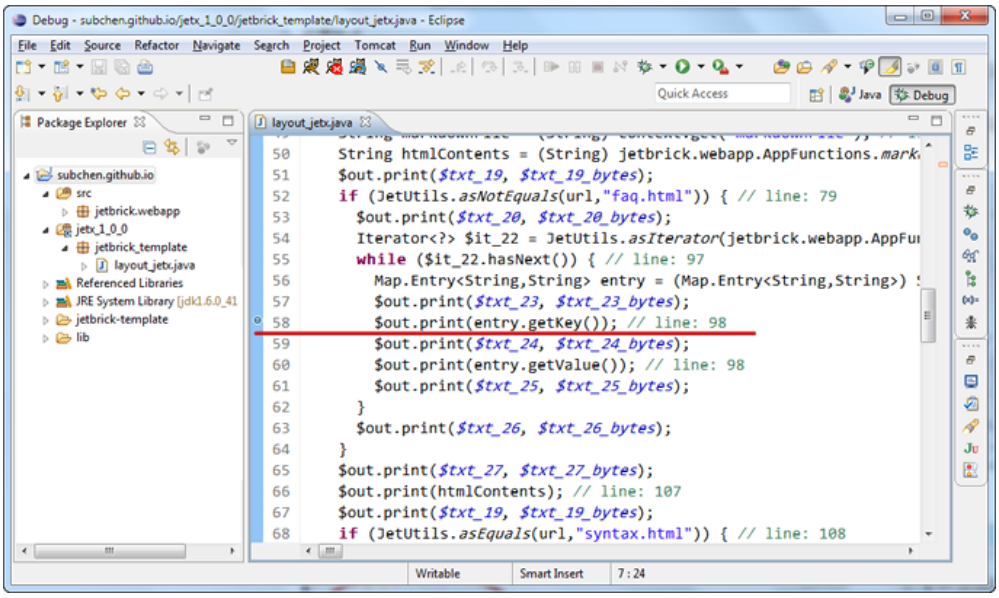
```
<logger name="jetbrick.template.runtime.JetUtils" level="DEBUG" />
```

4.4.2 用 Eclipse 进行调试

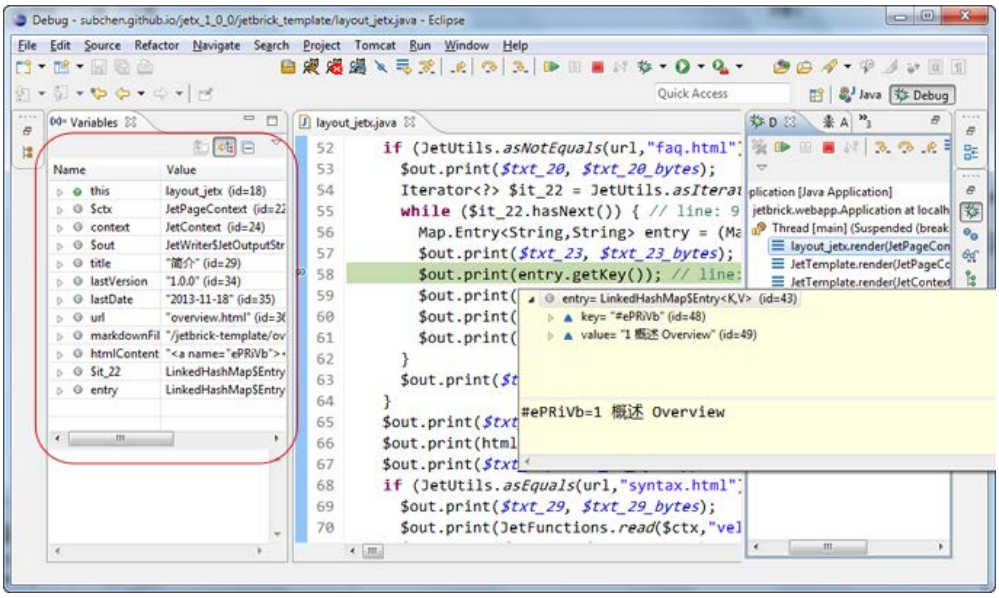
1. 将模板编译路径连接到 Project 的 source path



2. 设置断点



3. 开始 debug



调试秘诀-如何在Jetbrick Template中使用debug函数及配置细节

原 如何在 jetbrick-template 中使用 debug函数？

目录[-]

- debug 函数格式
- 开启 debug 日志
- 模板中输出 debug
- 查看 Tomcat 控制台日志

debug 函数格式

jetbrick-template 已经内置了一个 debug 函数： 格式如下：

```
1 void debug(String format, Object... args)
```

> 注意：
>
> 1. 要使用 debug 函数，需要 Slf4j 配合，在对应的 log 实现中打开 debug.
> 2. 具体的 format 参数格式请查看 Slf4j Logger。

开启 debug 日志

- Log4j:

```
1 log4j.logger.jetbrick.template.runtime.JetUtils = DEBUG
```

- Logback

```
1 <logger name="jetbrick.template.runtime.JetUtils" level="DEBUG" />
```

模板中输出 debug

然后，我们就可以在模板中这么用：

```
1 #define(String id, List<UserInfo> users)
2 ${debug("id = {}, users.size = {}.", id, users.size()}
```

查看 Tomcat 控制台日志

```
1 11:41:00.332 [main] DEBUG (JetUtils.java:187) - template debug: id = 16, users.size = 210.
```

配置指南-内功心法

1 全局配置选项
1.1 全局包/类/变量
1.2 扩展方法/函数/标签
1.3 模板路径和编码格式
1.4 编译选项
1.5 安全管理器
1.6 注释指令
1.7 其他选项
1.8 推荐配置

1 全局配置选项

名称	说明	默认值
import.packages	默认导入的 java 包	
import.classes	默认导入的 java 类	
import.variables	默认导入的 java 对象	
import.methods	默认导入的扩展方法	
import.functions	默认导入的扩展函数	
import.tags	默认导入的自定义标签 tags	
import.autoscan	是否自动扫描用户自定义扩展 Class	false
import.autoscan.packages	在指定的包中进行自动扫描	
input.encoding	模板源文件的编码格式	utf-8
output.encoding	模板输出编码格式	utf-8
template.loader	模板资源载入Class	jetbrick.template.resource.loader.FileSystemResourceLoader
template.path	模板资源的根目录	当前目录
template.suffix	默认模板文件扩展名	.jetx
template.reloadable	是否允许热加载	false
compile.strategy	编译策略	always
compile.always (已过时)	是否总是重新编译	true
compile.debug	是否允许输出 debug 信息	false
compile.path	默认编译输出路径	系统TEMP目录下面的 jetx 目录
security.manager	安全管理器实现类	
security.manager.file	安全管理器黑白名单文件	
security.manager.namelist	安全管理器黑白名单列表	
trim.directive.line	是否要删除指令行两边的空白	true
trim.directive.comments	是否支持指令两边增加注释对	false
trim.directive.comments.prefix	指令注释的开始部分	<!--
trim.directive.comments.suffix	指令注释的结束部分	-->

> **注意**：
>
> 1. 所有配置选项都必须在 `JetEngine` 初始化的时候指定，不允许动态修改。
> 2. 所有配置选项都支持变量啦，具体参考 `template.path` 或者 [`compile.path`][17] 中的例子。

1.1 全局包/类/变量

在模板中，如果要用到一些其他的 Class, 那么可以先 import 进来，这样就可以在模板中使用短名字，比如 `Date` 而不是 `java.util.Date`。

1.1.1 import.packages

用来配置包名，会自动导入包下面的所有类。允许配置多个包名，用逗号分隔。

1.1.2 import.classes

用来配置单个类名，优先级比 `import.packages` 高。允许配置多个类名，用逗号分隔。

示例如下：

```
import.packages = jetbrick.schema.app.model, jetbrick.schema.app.methods.*
import.classes = java.io.File, java.util.List
```

> **注意**：`jetbrick-template` 会自动引入 `java.lang.*` 和 `java.util.*`。

1.1.3 import.variables

在一个 webapp 中，我们希望每个模板都自动引入一些变量，比如 `HttpServletRequest request`，那么我们就可以在这里定义。

允许配置多个变量定义，用逗号分隔。示例如下：

```
import.variables = HttpServletRequest request
import.variables = jetbrick.orm.Pagelist pagelist, List<Entity> entites
```

变量的类型可以使用泛型定义，并且会自动在 `import.packages` 和 `import.classes` 里面查找 Class。

> **注意**：全局定义的变量如果在模板中被重新定义成其他类型(`#define`, `#set`)，则以模板定义优先。

1.2 扩展方法/函数/标签

1.2.1 import.methods

我们知道一个 Java Class 的 所有 methods 都是定义在同一个 class 文件中的，不能在其他地方进行动态扩展。但是一些其他动态语言却支持在 Class 外部为这个 Class 增加一些方法。比如：

- JavaScript 的 prototype 机制
- Groovy 的 metaClass 机制
- Kotlin

jetbrick-template 也在这里带给大家强大的动态方法扩展机制。

具体参考： jetbrick-template 动态方法扩展

这里就是把实现了动态扩展的 Method Class 注册到 JetEngine 中。允许配置多个 Class 定义，用逗号分隔。示例如下：

```
import.methods = StringMethods, app.project.methods.UserAuthMethods
```

定义的类名会自动在 `import.packages` 里面查找 Class。

`jetbrick-template` 默认会注册 `jetbrick.template.runtime.JetMethods` ，
具体参考：默认的方法扩展 Methods。

1.2.2 import.functions

和 `import.methods` 类似，我们还支持在模板中使用函数。

允许配置多个 Function Class 定义，用逗号分隔。示例如下：

```
import.functions = app.project.methods.UserAuthFunctions
```

`jetbrick-template` 默认会注册 `jetbrick.template.runtime.JetFunctions` ，
具体参考：默认的函数扩展 Functions。

1.2.3 import.tags

我们支持在模板中自定义标签 #tag。

允许配置多个 Tag Class 定义，用逗号分隔。示例如下：

```
import.tags = app.project.tags.UserTags
```

`jetbrick-template` 默认会注册 `jetbrick.template.runtime.JetTags` ，
具体参考：默认的自定义标签 Tags。

1.2.4 import.autoscan

是否自动扫描用户自定义的扩展 Class，扫描的内容是： 扩展方法，扩展函数，自定义标签。

默认 `false` ，不启用。

1.2.5 import.autoscan.packages

在指定的包下面进行自动扫描，如果为空，那么扫描整个 classpath。支持定义多个包。

```
import.autoscan = true
import.autoscan.packages = app.methods, app.functions, app.tags
```

> **注意**：
>
> 1. 扫描整个 classpath 需要花费一定的时间（大约每秒10000个类），建议配置 `import.autoscan.packages` 以加快速度。
> 2. 由于不会对扫描的 class 加载到 jvm 中，所以不会产生 OOM。

更多详细内容请参考： 如何让自动扫描发现用户自定义的扩展方法/函数/标签 Class

1.3 模板路径和编码格式

1.3.1 input.encoding

模板源文件的编码格式，默认为 `utf-8`。

1.3.2 output.encoding

模板输出内容的编码格式，默认为 `utf-8`。

> **注意**：一般在 web 中，`output.encoding` 应该和 html 页面的 `contentType` 中的编码，以及 `response` 的 `characterEncoding` 完全一致。

1.3.3 template.loader

如何找到我们自己的模板文件呢？这里就是定义我们要使用的查找类。我们支持下面几种 Class

```
template.loader = jetbrick.template.resource.loader.FileSystemResourceLoader
template.loader = jetbrick.template.resource.loader.ClasspathResourceLoader
template.loader = jetbrick.template.resource.loader.JarResourceLoader
template.loader = jetbrick.template.web.WebResourceLoader
template.loader = jetbrick.template.resource.loader.MultipathResourceLoader
```

默认为 `jetbrick.template.resource.loader.FileSystemResourceLoader`。

注意：如果是 Web 集成模式，默认值为 `jetbrick.template.web.WebResourceLoader`。

1.3.4 template.path

除了要定义 `template.loader`，我们还需要定义模板存放的根目录。

默认为系统当前目录：`System.getProperty("user.dir")`。

注意：如果是 web 集成模式，默认为 webapp 的根目录。具体请参考：JetEngine 自动加载方式 中注意事项。

- 从文件系统加载

```
template.loader = jetbrick.template.resource.loader.FileSystemResourceLoader
template.path = /opt/app/templates/
```

- 从Classpath下加载

```
template.loader = jetbrick.template.resource.loader.ClasspathResourceLoader
template.path = /META-INF/templates/
```

- 从jar包中加载

```
template.loader = jetbrick.template.resource.loader.JarResourceLoader
template.path = /opt/app/templates.jar
```

- 从webapp目录中加载(仅在Web框架集成中有效，并且已经被设置为默认项)

```
template.loader = jetbrick.template.web.WebResourceLoader
template.path = /WEB-INF/templates
```

- 从多个目录中加载

```
template.loader = jetbrick.template.resource.loader.MultipathResourceLoader
template.path = file:/path/to, classpath:/, jar:/path/to/sample.jar, webapp:/WEB-INF/templates
```

注意：`template.path` 支持多种路径，由逗号分隔。每个路径由一个前缀开头，代表相应的 ResouceLoader。具体如下：

前缀	代表的 ResourceLoader
file:	FileSystemResourceLoader
classpath:	ClasspathResourceLoader
jar:	JarResourceLoader
webapp:	WebResourceLoader
<MyClassLoader>:	用户自定义的 ResourceLoader (完整类名)

现在 `template.path` 支持变量了，如：

```
template.path = ${user.dir}/templates
template.path = ${webapp.dir}/WEB-INF/templates
```

那么我们支持哪些变量呢？其实这些变量都来自于 `System.getProperty(name)`，只要 `System` 里有的，都支持。
其中 `webapp.dir` 是个特殊变量，由 Web 集成框架在系统启动的时候，通过 `System.setProperty("webapp.dir", servletContext.getRealPath("/"))` 设置的。

1.3.5 template.suffix

默认的模板文件扩展名 `.jetx`，主要用于 Web 框架集成中，用于查找和过滤模板用。

1.3.6 template.reloadable

在开发模式下面，我们一般需要频繁的修改模板内容来进行调试。那么我们需要打开这个功能来支持模板的热部署。（类似于 `JSP`）

是否需要重新编译和加载模板，取决于模板源文件的最后修改时间。

默认为 `false`，建议只在开发模式中启用。

1.4 编译选项

jetbrick-template 采用编译成 Java ByteCode 来提高性能。

1.4.1 compile.strategy

模板从 1.2.0 开始，提供更加灵活的编译策略。由下面 4 中情况

```
```

```
compile.strategy = precompile
compile.strategy = always
compile.strategy = auto
compile.strategy = none
```

- `precompile`
  在 JetEngine 初始化的时候，自动获取所有的模板(根据 `template.suffix` 过滤)，然后启动一个独立的线程进行编译。
  这样虽然启动时间会增加，但是后面的模板访问将会非常的快。并且在预编译没有完成期间，应用可以正常访问，不冲突。

- `always` （默认值）
  和原先的 `compile.always = true` 等价。
  就是在模板被首次访问的时候，进行编译。

- `auto`
  和原先的 `compile.always = false` 等价。
  就是在模板被首次访问的时候，如果磁盘中已经存在编译好的 Class 文件（并且源文件没有改变），那么直接加载该 Class 文件，否则进行编译。

- `none`
  改模式下，将不在对模板进行编译。（发布的时候，用户无需发布任何模板源文件）
  用户必须通过 `JetxGenerateApp` 预编译工具，事先将模板全部编译成 class 文件，并将所有的 class 文件放在 classpath 下面。
  注意： class 文件放在 classpath 下面，而不是 `compile.path` 对应的目录。

> 注意：
> 不管采用什么模式，对于使用 `JetEngine.createTemplate(source)` 直接由源码创建的模板，仍然需要进行编译。

1.4.2 compile.always

已过时，已经被 `compile.strategy` 代替。

1.4.3 compile.debug

是否在日志中打印输出模板生成的 Java Source 源代码。

默认 `false` ，建议在开发模式中启用。

> **注意**：同时需要 slf4j 的配合才能输出日志。默认已经开启了 `INFO` 级别的日志。

1.4.4 compile.path

在模板编译的时候，会先生成对应的 `.java` 文件，然后在把 `.java` 文件编译成 `.class` 文件。我们生成的这 2 种文件就放在这个目录下面。

在用 Eclipse 进行 debug 的时候，可以 link 这个目录为 sourcepath 来进行 debug。
具体参考：如何调试模板 debug ?

默认会在系统TEMP目录 `System.getProperty("java.io.tmpdir")` 下面新建一个 `jetx` 目录。如果这个目录非法或者没有写的权限，那么就会抛出 Exception。

> **注意**：
>
> - 如果一个应用中使用多个 `JetEngine` 实例，请配置不同的 `compile.path` 防止出现冲突。我们建议用户每次都重定义这个路径。

现在 `compile.path` 支持变量了，如：

```
compile.path = ${java.io.tmp}/jetx
compile.path = ${webapp.dir}/WEB-INF/jetx_classes
```

那么我们支持哪些变量呢？其实这些变量都来自于 `System.getProperty(name)` ，只要 `System` 里有的，都支持。
其中 `webapp.dir` 是个特殊变量，由 Web 集成框架在系统启动的时候，通过 `System.setProperty("webapp.dir", servletContext.getRealPath("/"))` 设置的。

1.5 安全管理器

从 1.2.0 开始，模块新增了安全管理器，特别适合于 CMS 软件，允许用户自定义模板的场景。

1.5.1 security.manager

配置安全管理器的实现类，默认为空，表示禁用安全管理器。

启用方式(使用默认的安全管理器)：

```
security.manager = jetbrick.template.parser.JetSecurityManagerImpl
```

用户也可以实现自己的安全管理器，只要实现接口： `jetbrick.template.JetSecurityManager` 即可。

安全管理器只在对模板进行解析编译的时候进行，运行期不会影响任何性能。

1.5.2 security.manager.file

给默认的安全管理器，配置黑白名单，将该名单放在独立的外部文件中。（每行一个名单）

```
security.manager.file = ${webapp.dir}/WEB-INF/jetx-white-black-list.txt
```

1.5.3 security.manager.namelist

给默认的安全管理器，配置黑白名单，多个名单以逗号分隔。

```
security.manager.namelist = -java.lang.System.exit \
                            -java.lang.reflect \
                            -java.sql \
                            -javax.tools \
                            -java.io \
                            +java.io.File.getName \
                            +java.io.File.getPath \
                            -sun \
```

`security.manager.file` 和 `security.manager.namelist` 二选一配置即可。

黑白名单的格式如下：

1. 前缀符号：
    - `+` 开头代表白名单
    - `-` 开头代表黑名单

- 没有开始符号，则默认为白名单

2. 名单格式：

   - 包名： `pkg`
   - 类名名： `pkg.class`
   - 方法名： `pkg.class.method`
   - 字段名： `pkg.class.field`

实例：

```
-java.sql                          // 禁止访问 java.sql 下面的任何 Class，包括所有孙子包下面的 Class
-java.lang.System.exit             // 禁止调用 System.exit() 方法
+java.util.Collections.EMPTY_LIST  // 允许访问 Collections.EMPTY_LIST 字段
```

1.6 注释指令

由于目前的指令一般直接嵌入在 HTML，对于一些使用可视化编辑器的用户来说，可能会造成一些干扰。
模板从 1.0.1 开始增加对指令注释支持，如： `<!-- #if (...) -->`; 增强对可视化编辑器的友好度。

1.6.1 trim.directive.comments

是否开启对注释指令的支持，默认为 `false` ，表示不启用。

1.6.2 trim.directive.comments.prefix

设置注释开始格式，默认为 `<!--`

1.6.3 trim.directive.comments.suffix

设置注释开始格式，默认为 `-->`

> **注意**： 如果开启注释指令的支持，系统并没有强制要求 `trim.directive.comments.prefix` 和 `trim.directive.comments.suffix` 必须配对出现。也就是说如果使用 `<!-- #end` 也是可以的。当然我们还是建议你配对使用。

范例：

```
<table>
<!-- #for (User user: userlist) -->
  <tr>
    <td>${user.name}</td>
    <td>${user.email}</td>
  </tr>
<!-- #end -->
</table>
```

1.7 其他选项

1.7.1 trim.directive.line

由于指令之间存在很多的空白内容，而空白内容也会被作为原始文本原封不动的输出，这样会造成很多输出的内容参差不

齐。这个就是用来优化输出格式的，对于用模板来进行代码生成时候特别有用。不建议关闭。

模板示例：

```
#for (int n: [1,2,3])
${n}
#end
```

禁用后效果：`false`

```
1

2

3
```

启用后的效果：`true` (默认启用)

```
1
2
3
```

1.8 推荐配置

1.8.1 开发环境

```
import.packages = pkg1, pkg2
import.autoscan = true
import.autoscan.packages = pkg1, pgk2

template.path = /path/to/templates/
template.reloadable = true

compile.strategy = always
compile.path = /path/to/temp/
compile.debug = true
```

1.8.2 生产环境

```
import.packages = pkg1, pkg2
import.autoscan = true
import.autoscan.packages = pkg1, pgk2

template.path = /path/to/templates/
template.reloadable = false

compile.strategy = precompile
compile.path = /path/to/temp/
compile.debug = false
```

语法指南-招式秘诀

1 值 Value
2 指令 Directive
3 文本 Text
4 注释 Comment
5 表达式 Expression
6 默认的方法扩展 Methods
7 默认的函数扩展 Functions
8 默认的自定义标签 Tags
9 和 Velocity 的比较

这个是 `jetbrick-template` 模板语法参考手册。我们推荐的模板文件扩展名为 `.jetx`，嵌入式子模板的扩展名为 `.inc.jetx`。

1 值 Value

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` 为任何合法的 Java 表达式，参考: 表达式。

示例：

```
${user.name}
${user.book.available()}
$!{user.description}
```

> **注意**：
>
> - 如果 `expression` 为 `null`，则不会输出任何东西，如果需要输出 `null`，可以使用如下的方法扩展：`${expression.asString()}`。
> - 如果 `expression` 的返回类型为 `void`，那么也不会做任何输出动作。

2 指令 Directive

2.1 变量类型声明 #define

jetbrick-template 为了提高性能，采用了强类型来编译模板，所以需要为每一个用到的变量定义变量类型。如下：

语法：

- `#define(type name, ...)`

示例：

```
#define(String name)
#define(UserInfo user, List<UserInfo> userlist)
```

> **注意**：
>
> - 在相同作用域下面，不允许重复定义变量类型，变量只在当前作用域有效。
> - 对于没有定义变量类型，默认作用域为全局有效(Global)。
> - 对于没有定义变量类型，那么优先从上下文表达式中进行类型推导, 否则默认类型为 `Object`。

2.2 赋值语句 #set

语法：

- `#set(type name = value, ...)`
- `#set(name = value, ...)`

示例：

```
#set(int num = 1+2*3)
#set(color1 = "#ff0000", color2 = "#00ff00")
```

> **注意**：
>
> - 在相同作用域下面，不允许重复定义变量类型，变量只在当前作用域有效。
> - 影响当前模板，以及子模板的 `JetContext` 变量。
> - 不影响父模板的 `JetContext` 变量。

2.3 赋值语句 #put

将变量内容保存到当前模板以及所有父模板的 `JetContext` 中，方便父子模板间进行变量传递。

语法：

- `#put(name, value)`
- `#put(name1, value1, name2, value2, ...)`

示例：

```
#put("num", 1 + 2 * 3)
#put("user", user, "name", "jetbrick")
```

> **注意**：
>
> - 可以传递多个 key-value 对
> - 参数 `name` 必须是 `String` 类型

2.4 条件语句 #if, #elseif, #else

如果 `#if` 条件表达式计算结果为 `true` 或非空，则输出指令所包含的块, 否则输出 `#else` 指令块。

语法：

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

示例：

```
#if (user.role == "admin")
    ...
#elseif (user.role == "vip")
    ...
#elseif (user.role == "guest")
    ...
#else
    ...
#end
```

> **注意**：

> - 对于 `expression` 为非 `Boolean` 值：非零数字，非空字串，非空集合，非 `null` 对象，即为 `true`。
> - `#elseif` 允许出现多次。
> - 如果 `#else` 后面紧跟着其他文本，比如 `#elseABC`，那么可以通过添加一对空括号来分割，修改成 `#else()ABC`。这样可读性就能加强，模板解析也不会出现问题。所有的无参数指令，比如 `#end`，`#break`，`#stop` 都支持这样操作。`()` 前面和之间请不要插入任何空格。

2.5 循环语句 #for

循环重复输出指令所包含的块，如果是空的集合对象，那么输出 `#else` 块。

语法：

- `#for(id: expression)` ... `#end`
- `#for(id: expression)` ... `#else` ... `#end`
- `#for(type id: expression)` ... `#end`
- `#for(type id: expression)` ... `#else` ... `#end`

`#for` 支持以下类型的 `expression`

- Iterator
- Iterable (Collection, ...)
- Map
- Enumeration
- Array
- null (空循环)
- Object (只循环一次)

示例：

```
#for (book : user.books)
    ${for.index} // 内部循环计数器，从 1 开始计数
    ...
#end
```

循环变量 id 类型声明，用作强制转型，避免类型推导失败。

```
#for (Book book : user.books)
    ...
#end
```

指令 `#else` 可用于循环为空时的内容输出。

```
#for (Book book : user.books)
    ...
#else
    No books are found in here.
#end
```

2.5.1 for 内部对象

- `for.index` 可用于内部循环计数，从 1 开始计数。
- `for.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。
- `for.first` 是否第一个元素。
- `for.last` 是否最后一个元素。
- `for.odd` 是否第奇数个元素。
- `for.even` 是否第偶数个元素。

2.6 循环中断或继续语句 #break, #continue

当 `expression` 为 `true` ，`#break` 中断当前循环，而 `#continue` 跳过余下的内容，跳到下一个循环。

语法：

- `#break`
- `#break(expression)`
- `#continue`
- `#continue(expression)`

示例：

```
#for (book : user.books)
    ...
    #break(book.price > 100)
    ...
#end
```

```
#for (book : user.books)
    ...
    #continue(book.price > 100)
    ...
#end
```

> **注意**：
>
> - 无参数格式代表 `expression` 永远为 `true`。

2.7 停止解析语句 #stop

当 `expression` 为真或非空时，停止模板运行。

语法：

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

示例：

```
#stop(error != null)
```

> **注意**：
>
> - 无参数格式代表 `expression` 永远为 `true`。

2.8 嵌套模板语句 #include

嵌入一个子模板，将子模板内容输出到当前位置。

语法：

- `#include(file)`
- `#include(file, parameters)`

示例：

```
#include("/include/header.jetx") // 绝对路径
#include("../userinfo.jetx") // 相对路径
#include(file) // 动态路径
#include("/include/header.jetx", {role: "admin"}) // 传递参数
```

> **注意**：
>
> - 子模板自动共享父模板 `JetContext` 变量，同时还可以另外传递一些参数给子模板。
> - 子模板可以用 `#put` 指令修改父模板的 `JetContext`，然后在父模板中用 `context["name"]` 获取变量值。

具体用法请查考： 在 jetbrick-template 中如何使用 #include？

2.9 宏定义 #macro

定义一个代码片段，然后可以重复使用。(New from 1.1.0)

语法：

- `#macro name(type name, ...)` ... `#end`

> **注意**： 每个宏可以定义 0~N 个参数。

示例：

```
#macro header(String name)
    Hello ${name}!
#end

${header("张三")}
${header("李四")}
```

输出结果：

```
Hello 张三!
Hello 李四!
```

> **注意**：
>
> - 宏的调用就和函数调用一样。(如果和函数存在名称冲突，那么宏定义优先)
> - 宏的调用返回值是 Void，所以不支持对返回值进行计算。

2.10 自定义标签 #tag

jetbrick-template 支持自定义 Tag，类似于 JSP Taglib，但是要比 JSP Taglib 更简单更好用。(New from 1.1.0)

语法：

- `#tag name(args ...)` ... `#end`

> **注意**：
>
> - 需要在 Java 端先定义 Tag 标签的实现。
> - Tag 调用的参数必须和定义的一致。

示例：

```
#tag layout("layout.jetx")
    Hello ${name}!
#end
```

具体用法请查考： 在 jetbrick-template 中如何自定义 Tag？

3 文本 Text

普通文本内容将会被直接进行输出。

3.1 不解析文本块

原样输出模板内容，用于输出纯文本内容，或批量转义块中的特殊字符。类似于 XML 中的 CDATA。

语法：

- `#[[` ... `]]#`

示例：

```
#[[
    Source code will be displayed in here.
    Hello ${user.name}
]]#
```

3.2 特殊字符转义

原样输出指令特殊字符，转义字符由 `\` 进行转义。

语法：

- `\\`
- `\#`
- `\$`

示例：

```
\#if
\${user.name}
\\${user.name}
```

> **注意**：
>
> - 如果遇到类似 `#ff0000` 类似于指令的内容，但又不是系统定义的指令，那么也会原样输出，并不需要进行转义。
> - `\` 后面跟的字符不是 `#` 和 `$`，也不需要进行转义，直接输出。

4 注释 Comment

4.1 行注释

隐藏行注释的内容，以换行符结束，用于注解过程，或屏蔽指令内容。

语法：

- `##` ...

示例：

```
${user.name} ## This is a line comment.
```

4.2 块注释

隐藏块注释内容，可包含换行符，用于注解过程，或屏蔽指令内容。

语法：

- `#--` ... `--#`

示例：

```
#--
    This is a block comment.
--#
```

5 表达式 Expression

支持所有 Java 表达式，并对其进行了一些有用的扩展。

5.0.1 与 Java 相同的地方 (运算符优先级和 Java 保存一致)

- 数字常量： `123` `123L` `0.01D` `99.99E-10D`
- 字符串常量： `"abc\r\n"` `'abc\u00A0\r\n'`
- BOOLEAN 常量： `true` `false`
- NULL 常量: `null`
- 算术运行： `+` `-` `*` `/` `%`
- 自增/自减： `++` `--`
- 位运算： `~` `&` `|`
- 移位运算： `>>` `>>>` `<<`
- 比较运算： `==` `!=` `>` `>=` `<` `<=`
- 逻辑运算： `!` `&&` `||`
- 三元表达式： `? :`
- 实例对象判断： `instanceof`
- NEW 对象： `new Object(...)`
- 强制类型转换： `(java.lang.String)obj`
- 数组存取： `array[i]`
- 字段访问： `obj.field`
- 方法调用： `obj.method(...)`
- 方法调用：支持可变参数方法(Varargs)和重载方法(Overload)
- 静态字段调用： `@Long.MAX_VALUE`
- 静态方法调用： `@Long.valueOf("123")`
- 支持数组定义： `String[]`
- 支持泛型定义： `List<String>` `Map<String[], List<?>>`

5.0.2 与 Java 不同的地方

- 双等号 `==` 会被解析成 `equals()` 方法比较，而不是比内存地址。
- 单双引号都将生成字符串： `'a'` 或 `"a"` 都是 `String` 类型。
- Bean 属性会解析成 getter 方法调用， `${user.name}` 等价于 `${user.getName()}`
- 所有实现 `Comparable` 的对象都支持比较运算符，比如： `#if(date1 < date2)` ，可以比较日期的先后。
- 所有对象都条件测试，并返回 `true` 或者 `false` 。对于非 `Boolean` 对象，所有非零数字，非空字串，非空集合，非 `null` 对象，即为 `true` 。
- `List` 和 `Map` 可以方括号取值，比如： `list[0]` 等价于 `list.get(0)` ， `map["abc"]` 等价于 `map.get("abc")` 。
- `Map` 和 `JetContext` 对象支持 `.` 访问内部的对象（属性调用），如： `map.key` ， `context.key` 。
- 支持 Groovy 的 `?.` 安全调用，以避免 `NullPointerException` 。
- 静态字段/方法调用，需要用 `@` 前缀，比如： `@Long.MAX_VALUE` ，如果类带包名，需要这么用： `@(java.lang.Long).valueOf("123")`

5.1 变量名 Variable

可以是任意合法的 Java 变量名：

- 其中 `$` 开头的变量为模板内部变量，不允许直接使用。
- 不允许使用 Java 关键字。

如：`user`, `user_name`, `userName`

2 个内置的特殊变量：

- `context` ： 当前模板的 `JetContext` 对象。
- `for` ： 用于 `#for` 指令内部状态对象。具体参考 `#for` 指令用法。

5.2 List 常量

语法：

- `[item, ...]`

示例：

```
[] // empty List
[1, 2, 3, 4, 5]
["abc", 123, new Date(), 1+2*3]
```

取值：

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

// 安全调用
${list?[index]}
${list?.get(index)}
```

5.3 Map 常量

语法：

- `{key: value, ...}`

示例：

```
{} // empty Map
{"name" : "Jason", "statue" : 0}
```

取值：

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

// 安全调用
${map?.key}
${map?["key"]}
${map?.get("key")}
```

5.4 Bean 属性调用 bean.property

Bean 属性会解析成 getter 方法调用。

属性查找顺序，以 `${obj.foo}` 为例：

1. 查找 obj.getFoo() 方法
2. 查找 obj.isFoo() 方法
3. 查找 obj class 的 foo 字段
4. 查找 obj.get(name) 方法 (如果是 `Map` 或者 `JetContext`)

以上查找过程会在第一次编译的时候完成，不影响性能。
支持对属性返回值的类型推导。

> **注意**： `jetbrick-template` 支持属性的安全调用，和 Groovy 相同，你可以使用 `?.` 来代替 `.`，以避免出现 `NullPointerException`。

5.5 Bean 方法调用 bean.method(...)

- 支持普通方法调用
- 支持不定长参数方法调用 Varargs
- 支持方法重载 Overload
- 支持扩展方法调用。参考：扩展方法调用
- 支持对方法返回值的类型推导

示例：

```
## 方法重载 Overload
${"Hello".substring(2)}
${"Hello".substring(2, 3)}
```

> **注意**： `jetbrick-template` 支持方法的安全调用，和 Groovy 相同，你可以使用 `?.` 来代替 `.`，以避免出现 `NullPointerException`。

5.6 函数调用 function

jetbrick-template 还支持函数调用，如 `${now()}`，`${include("tag.jetx")}`。

具体参考：扩展函数调用

5.7 静态字段调用 @Class.Field

(New from 1.1.0)

语法：

- `@Class.Field`
- `@(package.Class).Field`

示例：

```
${@Long.MAX_VALUE}
${@(java.lang.Long).MAX_VALUE}
```

5.8 静态方法调用 @Class.method

(New from 1.1.0)

语法：

- `@Class.method(...)`

- `@(package.Class).method(...)`

示例：

```
${@Collections.emptyList()}
${@(java.lang.Long).valueOf("123")}
```

6 默认的方法扩展 Methods

所有方法扩展定义在 `jetbrick.template.runtime.JetMethods`

6.1 基本数据类型转换 Cast

- String.asBoolean()
- String.asChar()
- String.asByte()
- String.asShort()
- String.asInt()
- String.asLong()
- String.asFloat()
- String.asDouble()
- String.asDate()
- String.asDate(String format)
- Object.asString()

6.2 集合类型转换 Cast

- Collection.asList()
- boolean[].asList()
- char[].asList()
- byte[].asList()
- short[].asList()
- int[].asList()
- long[].asList()
- float[].asList()
- doubl[].asList()
- Object[].asList()

6.3 数据格式化 Format

- Number.format()
- Number.format(String format)
- Date.format()
- Date.format(String format)

6.4 数据 Escape/Unescape

- String.escapeJava()
- String.unescapeJava()
- String.escapeJavaScript()

String.unescapeJavaScript()
- String.escapeXml()
- String.unescapeXml()
- String.escapeUrl()
- String.escapeUrl(String encoding)
- String.unescapeUrl()
- String.unescapeUrl(String encoding)

6.5 默认值输出

- Object.asDefault(Object defaultValue)

6.6 JSON 输出

- Object.asJSON()

6.7 字符串转换

- String.toUnderlineName()
- String.toCamelCase()
- String.toCapitalizeCamelCase()
- String.repeat(int count)

6.8 算术计算

- int[].sum()
- int[].avg()
- int[].max()
- int[].min()

7 默认的函数扩展 Functions

所有函数扩展定义在 `jetbrick.template.runtime.JetFunctions`

7.1 常用函数

- `Date now()`
  获取当前时间
- `int random()`
  获取一个随机数
- `UUID uuid()`
  获取一个 UUID

7.2 循环计数生成器

生成一个用于循环的数组，主要用于 `#for` 的循环迭代。

@Deprecated from 1.0.2

- `int[] range(int start, int stop)`
- `int[] range(int start, int stop, int step)`

New added from 1.0.2

- `Iterator<Integer> iterator(int start, int stop)`
- `Iterator<Integer> iterator(int start, int stop, int step)`

范例：

```
#for (int i : iterator(1,100))
    ${i}
#end
```

7.3 嵌入子模板 include(...)

嵌入一个子模板。和 `#include` 指令的区别，此函数对子模板的输出进行了缓存，可以处理返回的内容，但是效率没有 `#include` 指令高。

- `String include(String relativeName)`
- `String include(String relativeName, Map<String, Object> parameters)`

7.4 嵌入纯文本文件 read(...)

- `String read(String relativeName)`
- `String read(String relativeName, String encoding)`

7.5 调试专用函数 debug(...)

- `void debug(String format, Object... args)`
  输出调试信息，需要配合 Slf4j 使用。 参数格式请查看 Slf4j Logger。

7.6 Web 路径获取 ctxpath() / webroot()

- `String ctxpath()`
  返回相对于 web 根目录的绝对路径，如 /myapp
- `String ctxpath(String url)`
  将 url 转换为相对于 web 根目录的绝对路径，如 /myapp/path/file
- `String webroot()`
  返回完整的 web 站点路径，如 http://127.0.0.1:8080/myapp
- `String webroot(String url)`
  将 url 转换为完整的 web 站点路径，如 http://127.0.0.1:8080/myapp/path/file

8 默认的自定义标签 Tags

所有 Tags 定义在 `jetbrick.template.runtime.JetTags`

- `#tag layout(String file)` ... `#end`

应用页面布局。
参考：jetbrick-template 中如何实现 layout 功能？

- `#tag layout(String file, Map<String, Object> parameters)` ... `#end`
  应用页面布局(支持传递传输)。
  参考：jetbrick-template 中如何实现 layout 功能？

- `#tag block(String name)` ... `#end`
  将块内容保存到变量名为 name 的 JetContext 中。
  参考：jetbrick-template 中如何实现 layout 功能？

- `#tag default_block(String name)` ... `#end`
  如果不存在指定的 JetContext 变量，那么输出 default_block 块内容，否则输出指定的 JetContext 变量。
  参考：jetbrick-template 中如何实现 layout 功能？

- `#tag application_cache(String name, long timeout)` ... `#end`
  将内存缓存到 ServletContext 中，在 timeout 秒之后自动超时。

- `#tag session_cache(String name, long timeout)` ... `#end`
  将内存缓存到 HttpSession 中，在 timeout 秒之后自动超时。

9 和 Velocity 的比较

9.1 语法差异

- jetbrick-template 指令中的变量不加 `$` 符，只支持 `#if(x == y)`，不支持 `#if($x == $y)`，因为指令中没有加引号的字符串就是变量，和常规语言的语法一样，加$有点废话，而且容易忘写。
- jetbrick-template 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，也防止多人开发时，有人加大括号，有人不加，干脆没得选，都加，保持一致。
- jetbrick-template 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}`，以免开发人员忘写感叹号，泄漏表达式源码，如需原样输出，可使用转义 `\${foo}`，在 jetbrick-template 中，`$!{foo}` 表示对内容进行 HTML 过滤，用于原样输出 HTML 片段。
- jetbrick-template 支持在所有使用变量的地方，进行表达式计算，也就是你不需要像 Velocity 那样，先 `#set($j = $i + 1)` 到一个临时变量，再输出临时变量 `${j}`，jetbrick-template 可以直接输出 `${i + 1}`，其它指令也一样，比如：`#if(i + 1 == n)`。
- jetbrick-template 采用扩展 Class 原生方法的方式，如：`${"str".toChar()}`，而不像 Velocity 的 Tool 工具方法那样：`$(StringTool.toChar("a"))`，这样的调用方式更直观，更符合代码书写习惯。
- jetbrick-template 支持属性和方法的安全调用。如 `${user?.name}`，`${user?.hasRole("vip")}`。如果 `user` 对象为 `null`，那么返回结果就是 `null`，不会出现烦人的 `NullPointerException`。
- jetbrick-template 还支持静态字段/方法调用，函数扩展，上下文相关的方法/函数扩展。

9.2 指令差异

velocity	jetbrick-template	异同	功能	变化
${foo.bar} $foo.bar	${foo.bar}	相同	输出占位符	jetbrick-template 大括号必需
$!{foo.bar} $!foo.bar	$!{foo.bar}	不同	空值不显示源码	velocity 为空值不显示源码 jetbrick-template 改为HTML过滤输出
## ... #* ... *#	## ... #-- ... --#	相似	注释	块注释格式不一样

#[[...]]#	#[[...]]#	相同	不解析文本块	
\# \$ \\	\# \$ \\	相同	特殊字符转义	
n/a	#define(Type foo = bar)	新增	给变量声明类型	
#set($foo = $bar)	#set(foo = bar) #set(Type foo = bar)	相同	给变量赋值	可带类型声明
n/a	#put(name, value)	新增	保存变量到全局	支持父子模板参数的全局传递
#if($foo == $bar)	#if(foo == bar)	相同	条件判断	
#elseif($foo == $bar)	#elseif(foo == bar)	相同	否定条件判断	
#else	#else	相同	否定判断	
#end	#end	相同	结束指令	
#foreach($item in $list)	#for(item : list) #for(Type item : list)	相似	循环指令	改为Java格式，可以带类型声明
#break	#break #break(foo == bar)	相同	中断循环	可以直接带条件
n/a	#continue #continue(foo == bar)	新增	继续下一个循环	可以直接带条件
#stop	#stop #stop(foo == bar)	相同	停止模板解析	可以直接带条件
#macro($foo)	#macro foo(...)	相似	可复用模板片段宏	velocity 将宏作为指令执行 jetbrick-template 作为函数执行
n/a	#tag foo(...)	新增	自定义标签	jetbrick-template 允许自定义标签
#include("foo.txt")	read("foo.txt")	相似	读取文本文件内容	jetbrick-template 用扩展函数实现
#parse("foo.vm")	#include("foo.jetx") #include("foo.jetx", args)	相同	包含另一模板输出	jetbrick-template 支持私有参数传递
#evaluate("${1 + 2}")	n/a	不同	模板求值	jetbrick-tempate 暂不支持

1 简述
2 Web 中的默认隐含对象
3 JetEngine 自动加载方式
4 各种集成方式介绍
4.1 直接使用 HttpServlet
4.2 直接使用 Filter 模式
4.3 Struts 2.x
4.4 Spring MVC
4.5 JFinal
4.6 Nutz
4.7 Jodd

1 简述

jetbrick-template 除了作为普通的模板引擎嵌入在 Application 中外，大部分情况下还会和各种 WebMVC 框架整合作为前端的 View，来代替过时的 JSP 或者 Velocity。

目前已近集成了几种流行的 Web 框架：

- HttpServlet
- Filter
- Struts 2.x
- Spring MVC
- JFinal
- Nutz
- Jodd

点击这里下载各种集成方式的演示 demo

2 Web 中的默认隐含对象

当 jetbrick-template 被用作 Web 应用中时候，会自动引入下面的对象，这些对象在所有的模板中全局可访问。

隐含对象	类 型	说 明
context	JetConext	
servletContext	ServletContext	
session	HttpSession	
request	HttpServletRequest	
response	HttpServletResponse	
applicationScope	Map<String,Object>	快捷访问 servletContext.getAttribute(name)
sessionScope	Map<String,Object>	快捷访问 session.getAttribute(name)
requestScope	Map<String,Object>	快捷访问 request.getAttribute(name)
parameter	Map<String,String>	快捷访问 request.getParameter(name)
parameterValues	Map<String,String[]>	快捷访问 request.getParameterValues(name)
ctxpath	String	快捷访问 request.getContextPath()
webroot	String	返回完整的webapp路径: http://127.0.0.1:8080/myapp

下面的例子演示了如何使用这些隐含变量：

模板如下：

```
1
```

```
request.requestURI == ${request.requestURI}
request.getParameter("name") == ${parameter.name}
request.getAttribute("items") == ${requestScope.items}
session.getAttribute("user") == ${sessionScope.user}
```

特别需要说明的一点是：模板中使用或者声明的全局变量不光会从 `JetConext context` 中获取，在 Web 应用中，还会从 `requestScope` ，`sessionScope` ，`applicationScope` 中查找对应的内容。

默认的查顺序如下：

1. context
2. requestScope
3. sessionScope
4. applicationScope

也就是说，如果存在 `request.getAttribute("user")` 的情况下 `${user}` 等价于 `${requestScope.user}` 。

3 JetEngine 自动加载方式

需要在 web.xml 中进行配置，下面两个配置项都是可选项。

```
<context-param>
  <param-name>jetbrick-template-config-location</param-name>
  <param-value>/WEB-INF/jetbrick-template.properties</param-value>
</context-param>

<listener>
  <listener-class>jetbrick.template.web.JetWebEngineLoader</listener-class>
</listener>
```

如果没有配置 `jetbrick-template-config-location` 参数，那么配置文件默认从 classpath 根目录下面查找 jetbrick-template.properties。

如果没有配置 JetWebEngineLoader 启动监听器，那么 JetEngine 也会在模板第一次请求的时候自动初始化。配置成 Listener，可以在 Webapp 启动的时候马上进行初始化。

注意：

1. 在 Web 集成模式中，采用以下的默认值：

   ```
   template.loader = jetbrick.template.web.WebResourceLoader
   template.path = /
   ```

2. 对于 `WebResourceLoader` 的来说，`template.path` 的路径相对于 webapp 的根目录。

   ```
   / == servletContext.getRealPath("/")
   /WEB-INF/jetx == servletContext.getRealPath("/WEB-INF/jetx")
   ```

3. jetbrick-template 内置和其他 Web 框架的集成方式都可以用这两个配置想进行全局初始化。

4. 在 web 集成模式中，`JetEngine` 是单例的，可以通过 `JetWebEngineLoader.getJetEngine()` 获取。

4 各种集成方式介绍

4.1 直接使用 HttpServlet

jetbrick-template 可以直接作为 HttpServlet 使用。需要在 web.xml 中作如下配置。

```
<servlet>
  <servlet-name>jetbrick-template</servlet-name>
  <servlet-class>jetbrick.template.web.servlet.JetTemplateServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>jetbrick-template</servlet-name>
  <url-pattern>*.jetx</url-pattern>
</servlet-mapping>
```

最简单，也是最直接的方式。打开浏览器访问 http://127.0.0.1:8080/index.jetx 看看效果吧。

具体例子代码参考： https://github.com/subchen/jetbrick-template-webmvc-samples/

4.2 直接使用 Filter 模式

jetbrick-template 可以直接作为 Filter 使用。需要在 web.xml 中作如下配置。

```
<filter>
  <filter-name>jetbrick-template</filter-name>
  <filter-class>jetbrick.template.web.servlet.JetTemplateFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>jetbrick-template</filter-name>
  <url-pattern>*.jetx</url-pattern>
</filter-mapping>
```

4.3 Struts 2.x

jetbrick-template 可以和 Struts 2.x 进行集成。

首先需要对 Struts 进行如下配置（ `struts.xml` ），这个配置是告诉 Struts 使用 `jetbrick.template.web.struts.JetTemplateResult` 这个类来处理采用 jetbrick-template 格式的模板：

```
<result-types>
  <result-type name="jetx" class="jetbrick.template.web.struts.JetTemplateResult" />
</result-types>
```

接下来配置你的 action 如下：

```
<action name="index" class="jetbrick.template.samples.struts.action.IndexAction">
  <result type="jetx">/index.jetx</result>
</action>
```

打开浏览器访问 http://127.0.0.1:8080/index.do 看看效果吧。

具体例子代码参考： https://github.com/subchen/jetbrick-template-webmvc-samples/

4.4 Spring MVC

jetbrick-template 可以和 Spring MVC 进行集成。

配置方式如下：

```
<bean id="viewResolver" class="jetbrick.template.web.springmvc.JetTemplateViewResolver">
  <property name="suffix" value=".jetx" />
  <property name="contentType" value="text/html; charset=utf-8" />
  <property name="order" value="9" />
</bean>
```

具体例子代码参考： https://github.com/subchen/jetbrick-template-webmvc-samples/

4.5 JFinal

jetbrick-template 可以和 JFinal 进行集成。

1. 修改 JFinal 主配置文件

```
public class JetxConfig extends JFinalConfig {
    @Override
    public void configConstant(Constants me) {
        me.setMainRenderFactory(new JetTemplateRenderFactory());
        ...
    }
    ...
}
```

2. 新建一个 Controller

```
public class UsersController extends Controller {
    public void index() {
        setAttr("userlist", DaoUtils.getUserList());
        render("/users.jetx");
    }
}
```

可以了，就这么简单!

具体例子代码参考： https://github.com/subchen/jetbrick-template-webmvc-samples/

4.6 Nutz

感谢 wendal (wendal1985@gmail.com) 提供相关代码。

View： `jetbrick.template.web.nutz.JetTemplateView`
ViewMaker： `jetbrick.template.web.nutz.JetTemplateViewMaker`

1. 将视图工厂整合进应用中：

   在主模块上，加： `@Views({JetTemplateViewMaker.class})` 注解

```
@Views({JetTemplateViewMaker.class})
@...
public class MainModule {
}
```

2. 使用 jetx 视图：

```
@At
@Ok(".jetx:/WEB-INF/html/user_info.jetx")
public String list(@Param("name") String name, HttpServletRequest request){
  if (name != null) return name;
```

```
    return "测试";
}
```

3. 模板中使用:

```
#define(String obj)
${obj}
```

4. 获得输出：

```
测试
```

具体例子代码参考： https://github.com/subchen/jetbrick-template-webmvc-samples/

4.7 Jodd

1. 首先需要配置 Jodd 的配置文件：madvoc.props

```
[jetbrick.template.web.jodd.JetTemplateResult]
contentType=text/html; charset=UTF-8

[automagicMadvocConfigurator]
includedEntries=jodd.*,jetbrick.template.web.jodd.*,yourapp.jodd.action.*
```

2. Action 例子

```
@MadvocAction
public class UsersAction {
    @Out
    Collection<UserInfo> userlist;

    @Action(extension = Action.NONE)
    public Object view() {
        userlist = DaoUtils.getUserList();
        return "jetx:/users.jetx";
    }
}
```

3. jetx 例子

```
<table border="1" width="600">
  <tr>
    <td>ID</td>
    <td>姓名</td>
    <td>邮箱</td>
    <td>书籍</td>
  </tr>
  #for(UserInfo user: userlist)
  <tr>
    <td>${user.id}</td>
    <td>${user.name}</td>
    <td>${user.email}</td>
    <td><a href="books?author=${user.id}">书籍列表</a></td>
  </tr>
  #end
</table>
```

具体例子代码参考： https://github.com/subchen/jetbrick-template-webmvc-samples/

关于Jetbrick Template

2014-01-05：jetbrick-template-1.2.0 正式版已发布，新增预编译功能。

1. github 源码

   https://github.com/subchen/jetbrick-template

2. 官方主页

   http://subchen.github.io/jetbrick-template

3. 提交 bug 或者 requirements：

   https://github.com/subchen/jetbrick-template/issues

4. QQ 交流圈

   310491655

5. Email 联系

   subchen@gmail.com

6. 作者

   Guoqiang Chen, Shanghai, China

7. 博客

   http://my.oschina.net/sub/blog

8. 贡献者

   laughing (4680381@qq.com)

   - 提供离线 PDF 文档制作

   刘兵 (284520459@qq.com)

   应卓 (yingzhor@gmail.com)

   - 自定义标签 Tags for Spring Security
   - 自定义标签 Tags for Shiro
   - #31 增加 Spring FactoryBean 的集成支持

   Wendal Chen (wendal1985@gmail.com)

   - #19 与Nutz集成

9. 编码规范

   JAVA 编码规范 1.0 (jetbrick 版)

1 JAVA 编码规范 1.0 (jetbrick 版)
1.1 Java 文件格式
1.2 包名
1.3 类名
1.4 Imports
1.5 方法
1.6 常量
1.7 变量
1.8 注释
1.9 异常
1.10 日志
1.11 单元测试

1 JAVA 编码规范 1.0 (jetbrick 版)

1.1 Java 文件格式

1. 文件格式必须是 `UTF-8`，无 `BOM` 格式
2. 文件回车换行符必须是 `Unix` 风格
3. 每个文件结尾必须有一个空白行
4. 行尾空白内容应该被 trim 掉
5. 每个文件开头必须写上项目的标准 LICENSE 注释，如下：

```
/**
 * jetbrick-template
 * http://subchen.github.io/jetbrick-template/
 *
 * Copyright 2010-2013 Guoqiang Chen. All rights reserved.
 * Email: subchen@gmail.com
 *
 * Licensed under the Apache License, Version 2.0 (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.apache.org/licenses/LICENSE-2.0
 *
 * 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.
 */
```

6. 代码必须是格式化的，请使用统一的 Eclipse 的代码格式文件：eclipse-jetbrick-style-formatter.xml
7. 不想被自动格式化的代码请用 `@formatter` 包裹，如：

```
//@formatter:off
private static final String[] DATE_PATTERNS = new String[] {
    "yyyy-MM-dd HH:mm:ss.SSS",
    "yyyy-MM-dd HH:mm:ss",
    "yyyy-MM-dd",
    "HH:mm:ss"
};
//@formatter:on
```

8. Java 文件必须是可编译的，不应该有任何的 `warning` 存在

1.2 包名

1. 包名必须是全部小写的，最好用一个单词表示
2. 包名必须以 `jetbrick` 开头

3. 接口或者抽象类的多种实现，推荐以 `spi`，`support` 包命名

1.3 类名

1. 类名必须首字母大写，驼峰命名法： 如 `UserInfo` ，`ClassUtils`
2. 类名尽量不要缩写，如果缩写，必须为特别常用的缩写
3. 接口的命名不要以 `I` 开头
4. 抽象类推荐以 `Abstract` 开头
5. 接口的默认实现推荐以 `Default` 开头或者 `Impl` 结尾
6. 每个 Class 都需要标注 `@auther`，`@since`
7. 每个 Class 都应该有简短的注释

1.4 Imports

1. Imports 间不要有空行
2. 超过 3 个相同包下面的 Class 需要使用 `.*` 代替
3. 不要使用 `import static`，除了 `JUnit` / `TestNG` 的 `assertXXX` 方法

1.5 方法

1. 方法名称应该采用首字母小写，驼峰命名法： 如 `getUser` ，`lookupClass`
2. 对于一个 `public` 的方法，都应该对参数进行基本的校验，比如 `null` 检测
3. 对外开放 API 的 `public` 方法都需要标注 `@since`
4. 每个 `public` 方法都应该有简短的注释

1.6 常量

1. 常量必须是全大写，并用 `_` 分隔，如 `MAX_INTEGER`
2. 常量必须是 `static` `final`

1.7 变量

1. 变量名称必须首字母小写，驼峰命名法
2. 变量名尽量使用缩写，以简短为主
3. 不要用拼音，要用英文表示
4. 如果是集合或数组，用复数名词，或者添加 `List`，`Map` 等后缀

1.8 注释

1. 注释必须和代码保持一致，中文/英文均可
2. 注释中的第一个句子要以（英文）句号、问号或者感叹号结束。 `javadoc` 工具会将注释中的第一个句子放在方法汇总表和索引中。
3. 如果注释中有超过一个段落，用 `<p>` 标签分隔
4. 如果注释中有多个章节，用 `<h2>` 标签声明每个章节的标题
5. 示例代码以 `<pre>` 包裹

1.9 异常

1. 异常类名必须以 `Exception` 结尾
2. 所有自定义异常都必须继承自 `RuntimeException`

3. 方法尽量不要抛出非 `RuntimeException` 异常
4. 异常应该和主要的 Class 放在一起，而不是所有的异常类放在一个包下面
5. 异常描述应该使用英文句子，尽量不要用中文。
6. 被 `catch` 住的 `Exception` ，必须要处理，或者重新抛出

1.10 日志

1. 日志框架使用 `slf4j`
2. 实例不多的对象类，不要使用 `static` 声明 `log`
3. 尽量使用 `debug` 而不是 `info` 级别
4. 启动时候需要输出的重要日志，请用 `info` 级别
5. 被 `catch` 住的 `Exception` ，应该被打印出来 `log.error(e)`

1.11 单元测试

1. 单元测试框架用 `TestNG`
2. 单元测试覆盖率工具用 `EclEmma`
3. Mock 框架使用 `Mockito`
4. 尽可能为每个方法提供单元测试
5. 覆盖率应该不低于 70%

FAQ合集（此页面需要联网）

jetbrick

QQ 交流圈：31 1655

Fork me on GitHub

jetbrick template

1. 全新一代 Java 模板引擎
2. 具有高性能、高扩展性
3. 完美替代 JSP, Velocity 等引擎模板

jetbrick-template-1.2.0.zip
2014-01-05

2014-01-05：jetbrick-template-1.2.0 正式版已发布，新增预编译功能。

- jetbrick-template 将模板编译成 Java Class 有什么好处？
- jetbrick-template 中为什么需要 #define 声明变量类型？
- jetbrick-template 常见错误分析
- jetbrick-template 如何让自动扫描发现用户自定义的扩展方法/函数/标签 Class
- jetbrick-template 中如何嵌入子模板？（父子间参数传递）
- jetbrick-template 中如何自定义 Tag？
- jetbrick-template 中如何实现 layout 功能？
- jetbrick-template 在 Spring 中的集成方法

FAQ-为什么要将Jetbrick Template编译成Java Classs？

- jetbrick-template 将模板编译成 Java Class 有什么好处？
- jetbrick-template 中为什么需要 #define 声明变量类型？
- jetbrick-template 常见错误分析
- jetbrick-template 如何让自动扫描发现用户自定义的扩展方法/函数/标签 Class
- jetbrick-template 中如何嵌入子模板？（父子间参数传递）
- jetbrick-template 中如何自定义 Tag？
- jetbrick-template 中如何实现 layout 功能？
- jetbrick-template 在 Spring 中的集成方法

1 jetbrick-template 将模板编译成 Java Class 有什么好处？

1. 性能卓越

   消除反射，极少类型转换，减少内存消耗，比解释性的模板性能高一个数量级。

2. 方便调试

   解释性的模板无法调试，先生成 Java 源代码，在编译成 Class 文件，就可以方便进行调试。

3. 支持代码重构时的预先提示

   如果需要重构 Java 源代码，那么在重构的时候，立即可以知道哪些模板会受到影响。

   - 将模板生成的 Java Source 连接到项目的 sourcepath，然后借助于 Eclipse 等 IDE 的重构预览模式就可以发现哪些模板受到重构影响）。
   - 将模板全部重新编译一下，就能知道那些模板存在编译错误，可以提前发现问题，而不是等到模板实际运行的时候才发现问题。

FAQ-为什么必须使用#define声明变量类型？

- jetbrick-template 将模板编译成 Java Class 有什么好处？
- jetbrick-template 中为什么需要 #define 声明变量类型？
- jetbrick-template 常见错误分析
- jetbrick-template 如何让自动扫描发现用户自定义的扩展方法/函数/标签 Class
- jetbrick-template 中如何嵌入子模板？（父子间参数传递）
- jetbrick-template 中如何自定义 Tag？
- jetbrick-template 中如何实现 layout 功能？
- jetbrick-template 在 Spring 中的集成方法

1 jetbrick-template 中为什么需要 #define 声明变量类型？

由于 jetbrick-template 将模板编译成 Java Class 来提高模板的运行性能。所以模板在运行之前需要像 JSP 一样，先进行编译。

jetbrick-template 编译的时候，需要确定变量的类型来消除反射和类型转换。(注意： JSP 的 EL 表达式是通过反射解释执行的，不需要在编译期间确定变量的类型)

jetbrick-template 支持类型推导，但是如果无法进行类型推导的时候，就会默认对象的类型是 Object，那么如果需要调用非 Object 对象的属性或者方法的时候，就需要借助 #define 或者 #set 指令来进行变量类型的声明。

如何获取变量的类型：

1. 全局默认导入的变量类型，如 `context` ， `request` ， `session` 等
2. 全局 `import.variables` 中定义的变量类型
3. `obj.foo` 对应的属性类型
4. `obj.method()` 对应的方法返回值
5. 扩展函数、扩展方法的返回值
6. 其他运算结果

注意： 由于 Java 泛型采用的是类型消除（伪泛型），所以对于 Java 泛型的类型推导在一些情况下是不工作的，这时就需要用 `#set(type name = expression)` 将中间结果定义为一个类型。也可以用强制类型转换： `((String)(obj.items.get(0)).toUpperCase()` 。

一般在下列情况需要 #define 类型声明

- `${bar.foo}` 读取 foo 属性
- `${bar.foo()}` 调用 foo() 方法
- `${fnCall(bar)}` 调用 fnCall() 函数扩展

下列情况不需要 `#define` 类型声明

- `${bar}` 直接输出对象
- `#for (type var : items)` #for 指令的循环对象 items

我们建议每个模板中所有的 `#define` 语句都统一定义在文件开头位置，这样方便我们知道模板依赖的各种变量。

`#define` 指令是声明哪些全局 `contex` t 的变量类型，如果需要指定中间运算结果的变量类型，请用 `#set` 指令。

- jetbrick-template 将模板编译成 Java Class 有什么好处？
- jetbrick-template 中为什么需要 #define 声明变量类型？
- jetbrick-template 常见错误分析
- jetbrick-template 如何让自动扫描发现用户自定义的扩展方法/函数/标签 Class
- jetbrick-template 中如何嵌入子模板？（父子间参数传递）
- jetbrick-template 中如何自定义 Tag？
- jetbrick-template 中如何实现 layout 功能？
- jetbrick-template 在 Spring 中的集成方法

1 jetbrick-template 常见错误分析

1.1 The method getXXX() or isXXX() is undefined for the type Object

```
${obj.foo}
```

1. 确定 obj 存在 getFoo() or isFoo() 方法，并且是 `public` 的
2. obj 对象是否已经声明变量类型，否则请用 `#define(TYPE obj)` 声明变量类型。

1.2 The method foo(xxx, ...) is undefined for the type Object

```
${obj.foo(...)}
```

1. 确定 obj 存在 foo(...) 方法，并且是 `public` 的，参数类型是否匹配。
2. obj 对象是否已经声明变量类型，否则请用 `#define(TYPE obj)` 声明变量类型。
3. 如果 foo 是扩展方法，那么请确认扩展函数 XXX 是否已经注册到 `JetEngine` 中，或者参数类型是否匹配。

1.3 Operator [] is not applicable for the object (Object)

```
${obj[foo]}
```

1. obj 对象是否已经声明变量类型，否则请用 `#define(TYPE obj)` 声明变量类型。
2. 只有 `List` ，`Map` ，`JetContext` 对象支持 "[]" 操作

1.4 Duplicate local variable xxx

变量 xxx 定义的两次（相同作用域只能定义一次），请查找 `#define` 和 `#set` 指令是否对 变量 xxx 进行多次定义

1.5 Type mismatch: cannot convert from XXX to YYY

```
#define(String str)
#set(int a = str)
```

变量类型部不兼容，比如 `String` 对象复制给 `int`。

1.6 Undefined function XXX

扩展函数 XXX 没有找到，请确认扩展函数 XXX 是否已经注册到 `JetEngine` 中，或者参数类型是否匹配。

1.7 Undefined tag definition: XXX(...)

没有找到对应的 Tag 定义，请确认 Tag 是否已经注册到 `JetEngine` 中，或者参数类型是否匹配。

1.8 line xxx: Implicit definition for context variable: XXX

变量 XXX 没有声明变量类型，而直接使用。（这个仅仅是一个 Warning，不是 Error）

FAQ-如何自动扫描发现用户自定义的扩展方法/函数/标签/Class?

2014-01-05：jetbrick-template-1.2.0 正式版已发布，新增预编译功能。

- jetbrick-template 将模板编译成 Java Class 有什么好处？
- jetbrick-template 中为什么需要 #define 声明变量类型？
- jetbrick-template 常见错误分析
- jetbrick-template 如何让自动扫描发现用户自定义的扩展方法/函数/标签 Class
- jetbrick-template 中如何嵌入子模板？（父子间参数传递）
- jetbrick-template 中如何自定义 Tag？
- jetbrick-template 中如何实现 layout 功能？
- jetbrick-template 在 Spring 中的集成方法

1 如何自动发现用户自定义的扩展方法/函数/标签 Class

jetbrick-template 主要的扩展点有下列几个：

1. 扩展方法
2. 扩展函数
3. 自定义标签 #tag

常规的配置方法如下：

```
# 扩展方法
import.methods = app.methods.StringMethods, app.methods.DateMethods

# 扩展函数
import.functions = app.functions.UserFunctions

# 自定义标签
import.tags = app.tags.UserTags, app.tags.CacheTags
```

如果需要增加或者调整 Class，需要同时维护这个配置文件，比较麻烦。

从 1.1.2 开始，增加 annotation 自动扫描查找 Methods / Functions / Tags Class 的功能

具体的 annotation 如下：

```
@JetAnnotations.Methods
@JetAnnotations.Functions
@JetAnnotations.Tags
```

只要在对应的 Class 中，增加对应的 annotation 即可。

例如：

```
@JetAnnotations.Methods
public class StringMethods {
    ...
}
```

然后开启 `import.autoscan = true` 就可以自动发现了。

当然，为了加快发现的速度，建议同时配置 `import.autoscan.packages`。

Good luck.

- jetbrick-template 将模板编译成 Java Class 有什么好处？
- jetbrick-template 中为什么需要 #define 声明变量类型？
- jetbrick-template 常见错误分析
- jetbrick-template 如何让自动扫描发现用户自定义的扩展方法/函数/标签 Class
- jetbrick-template 中如何嵌入子模板？（父子间参数传递）
- jetbrick-template 中如何自定义 Tag？
- jetbrick-template 中如何实现 layout 功能？
- jetbrick-template 在 Spring 中的集成方法

1 jetbrick-template 中如何嵌入子模板？（父子间参数传递）

在 jetbrick-template 中我们有:

- `#include(file, ...)` 指令
- `include(file, ...)` 函数
- `read(file, ...)` 函数 - 用于嵌入纯文本文件

1.1 最常规的做法是

1.1.1 include 静态文件名

```
#include("/include/header.jetx")
${include("/include/header.jetx")}
```

1.1.2 include 动态文件名

```
#define(String url)
#include(url)
${include(url)}
```

> **注意**： `#include` 指令和 `include()` 函数的区别是：
>
> - `#include` 指令直接将内容输出到原始的 Stream/Writer 里面，效率高。
> - `include()` 函数将内容缓存到一个 String 中返回，可以对返回值进行进一步处理。
> - `#include` 指令如果包含的是静态文件名，那么会检查文件是否存在。 `include()` 函数不会做任何检查。

1.2 对 include() 函数返回值处理的应用

```

```

```
下面内容全部转为大写字母
${include("ascii.jetx").toUpperCase()}
```

输出内容

```

ABCDEFG...
```

1.3 父子模板间传递参数

我们有以下几种方式可以在父子模板间传递参数

- 父模板的 `JetContext` 自动传递给子模板。
- `#set` 指令会修改当前模板的 `JetContext`，同时也会影响子模板。
- `#include` 指令和 `include()` 函数的第二个参数可以传递一个单独的 Map 对象给子模板。
- 子模板通过 `#put` 指令向父模板返回数据。

1.3.1 include 父传子参数例子：

子模板 sub.inc.jetx

```

Hello from sub, parent name is ${parentName}.
```

父模板 parent.jetx

```

This is parent.
#include("sub.inc.jetx", {"parentName", "PARENT_NAME"})
```

显示结果如下：

```

This is parent.
Hello from sub, parent name is PARENT_NAME.
```

1.3.2 #put 子传父参数例子：

子模板 sub.inc.jetx

```

Hello from sub.
#put("age", 1234567890);
```

父模板 parent.jetx

```

This is parent.
#include("sub.inc.jetx")
sub.age = ${context.age}
```

显示结果如下：

```

This is parent.
Hello from sub.
sub.age = 1234567890
```

FAQ-Jetbrick Template中如何自定义标签(Tag)

- jetbrick-template 将模板编译成 Java Class 有什么好处？
- jetbrick-template 中为什么需要 #define 声明变量类型？
- jetbrick-template 常见错误分析
- jetbrick-template 如何让自动扫描发现用户自定义的扩展方法/函数/标签 Class
- jetbrick-template 中如何嵌入子模板？（父子间参数传递）
- jetbrick-template 中如何自定义 Tag？
- jetbrick-template 中如何实现 layout 功能？
- jetbrick-template 在 Spring 中的集成方法

1 如何自定义 Tag

jetbrick-template 支持自定义 Tag，类似于 JSP Taglib，但是要比 JSP Taglib 更简单更好用。

这里以建立一个支持 Cache 的 Tag 为例。

1.1 Cache Tag 代码范例：

1.1.1 首先定义一个 Tag 实现方法

每一个自定义的 Tag 对应一个 Java 的 `public` `static` 的方法。比如我们这里定义的 cache tag，如下:

Tag 方法定义：

```
public class MyTags {
    public static void cache(JetTagContext ctx, String name, int timeout) throws IOException {
        Cache cache = CacheManager.getCache(); // 请用自己的 Cache 代替
        Object value = cache.get(name);
        if (value == null) {
            value = ctx.getBodyContext();
            cache.put(name, value, timeout);
        }
        ctx.getWriter().print(value);
    }
}
```

1.1.2 必须要注册到全局的 JetEngine 中，我们使用如下的配置：

```
import.tags = MyTags, ...
```

1.1.3 然后写对应的例子模板：

```
#tag cache("sum", 10)
    计算结果将被缓存10秒： ${1+2+3+4+5+6+7+8+9}
#end
```

1.2 Cache Tag 代码说明

对于每一个 Tag 的方法声明，有如下要求：

- 方法签名必须是 `public` `static`
- 方法返回值必须是 `void`
- 方法第一个参数必须是 `JetTagContext` ， 其余参数自定义
- 允许 throws 任意的 Throwable
- 允许定义相同名字的 Tag，但是方法参数不一样 （ Overload ）
- 支持可变参数 (VarArgs)

在 Tag 方法中，我们可以通过 `JetTagContext` 来获取相关内容。主要 API 如下：

- String JetTagContext.getBodyContext()
  获取 `#tag ... #end` 之间的内容
- void JetTagContext.writeBodyContext()
  将内容原封不动的在原地进行输出
- void JetTagContext.getWriter().print(...)
  自定义输出内容
- JetContext JetTagContext.getContext()
  获取模板管理的 `JetContext` 对象，在 Web 环境中，可以通过 `JetContext` 对象进一步获取 `request`，`response` 等对象。
- JetEngine JetTagContext.getEngine()
  获取模板全局 Engine。

详细了解 `JetTagContext` 具体 API，请看 apidocs。

关于模板的使用：

- Tag 的参数必须和定义的一致。
  如果定义为: `cache(JetTagContext ctx, String name, int timeout)` ，那么在调用的时候，必须传2个参数，一个 String， 一个 int，比如：
  `#tag cache("abc", 123) ... #end`

FAQ-如何实现layout布局功能以重复利用模板?

2014-01-05：jetbrick-template-1.2.0 正式版已发布，新增预编译功能。

- jetbrick-template 将模板编译成 Java Class 有什么好处？
- jetbrick-template 中为什么需要 #define 声明变量类型？
- jetbrick-template 常见错误分析
- jetbrick-template 如何让自动扫描发现用户自定义的扩展方法/函数/标签 Class
- jetbrick-template 中如何嵌入子模板？（父子间参数传递）
- jetbrick-template 中如何自定义 Tag？
- jetbrick-template 中如何实现 layout 功能？
- jetbrick-template 在 Spring 中的集成方法

1 在 jetbrick-template 中如何实现 layout 功能

通常页面都有一个布局，大体上有页面头部，尾部，以及正文三部分。头部和尾部内容基本上固定，只有正文是变化的。

jetbrick-template 可以使用 3 种方法实现模板的 layout 功能。

1. `#include(file, ...)` 指令
2. `#tag layout(file, ...)` 标签
3. `#tag block(name)` + `#inlcude(file, ...)` 组合

1.1 常规模式

我们将页面公共部分放在 include 文件中，然后在每个页面中应用 include 文件来实现内容共享。

main.jetx

```
#include("/include/header.jetx")
<table>
  这是正文内容
</table>
#include("/include/footer.jetx")
```

这个不算 layout?
呵呵，继续放下看。

1.2 变种模式

常规模式中，只是共享了一些公共内容，但是并没有实现真正的 layout，因为 layout 一变，include 的方式可能就会发生变化，这样所有文件都要重新 include，不是很方便。

下面就是 #include 模式的变种（动态 include），来实现真正的 layout。

layout.jetx

```
<html>
  <head>
```

```
    <title>${title}</title>
  </head>
  <body>
    <div class="header">This is a logo image.</div>

    #include(parameter.pageUrl)

    <div class="footer">Copyright @2000-2010, All Rights Reserved.</div>
  </body>
</html>
```

main.jetx

```

<table>
  这是正文内容
</table>
```

Action.java

```

String url = "/layout.jetx?pageUrl=" + URLEncoder.encode("/main.jetx", "utf-8");
RequestDispatcher rd = getServletContext().getRequestDispatcher(url);
rd.forward(request, response);
```

怎么样？通过一个 pageUrl 参数来达到动态 layout 的目的，这样我们以后只要修改layout.jetx 文件就能达到修改所有页面布局了。

URL 访问方法： http://127.0.0.1:8080/layout.jetx?pageUrl=main.jetx

1.3 #tag layout(file, ...)

这里再介绍一种采用 layout 自定义标签来实现。

layout.jetx

```

<html>
  <head>
    <title>${title}</title>
  </head>
  <body>
    <div class="header">This is a logo image.</div>

    ${bodyContent}

    <div class="footer">Copyright @2000-2010, All Rights Reserved.</div>
  </body>
</html>
```

main.jetx

```

#tag layout("layout.jetx")
<table>
  这是正文内容
</table>
#end
```

layout 标签允许指定一个 layout 模板文件，在渲染页面的时候，会将layout标签体的渲染内容作为一个 bodyContent 变量插入到 layout 指定模板文件里。

这样，我们就可以通过直接访问 http://127.0.0.1:8080/main.jetx

1.4 组合 #tag block() + #inlcude

上面的 #tag layout() 标签只能嵌入一块自定义的内容 `${bodyContent}` ，那么如果想要嵌入多个自定义的变量块呢？

main.jetx

```

#tag block("bodyContent1")
```

```
    BODY 1111
#end
#tag block("bodyContent2")
    BODY 2222
#end
#include ("layout.jetx")
```

layout.jetx

```

This is a header.
<div>
    ${bodyContent1}
</div>
<div>
    ${bodyContent2}
</div>
This is a footer.
```

输出结果：

```

This is a header.
<div>
    BODY 1111
</div>
<div>
    BODY 2222
</div>
This is a footer.
```

怎么样？现在就可以随意的进行组合了。

1.5 block 的默认值和重载

在 layout.jetx 中，我们也可以定义默认内容，在 main.jetx 中对默认内容进行重载，如下：

layout.jetx

```

This is a header.
<div>
#tag default_block("bodyContent1")
    This is a default BODY 1111
#end
</div>
<div>
#tag default_block("bodyContent2")
    This is a default BODY 2222
#end
</div>
This is a footer.
```

main.jetx

```

#tag block("bodyContent1")
    Override block 111
#end
#include ("layout.jetx")
```

输出结果：

```

This is a header.
<div>
    Override block 111
</div>
<div>
    This is a default BODY 2222
</div>
This is a footer.
```

1.6 总结

不管采用哪种方式，jetbrick-template 都提供很大的灵活性来实现页面布局功能。

其中自定义 #tag 标签机制提供了非常强大功能，很容易实现其他模板引擎难以实现的功能，比如像 JSP Taglib 一样实现自定义标签，如 Cache Tag 将内容缓存到 memecached/redis 的标签等等。

FAQ-Jetbrick Template在Spring中的集成方法

2014-01-05：jetbrick-template-1.2.0 正式版已发布，新增预编译功能。

- jetbrick-template 将模板编译成 Java Class 有什么好处？
- jetbrick-template 中为什么需要 #define 声明变量类型？
- jetbrick-template 常见错误分析
- jetbrick-template 如何让自动扫描发现用户自定义的扩展方法/函数/标签 Class
- jetbrick-template 中如何嵌入子模板？（父子间参数传递）
- jetbrick-template 中如何自定义 Tag？
- jetbrick-template 中如何实现 layout 功能？
- jetbrick-template 在 Spring 中的集成方法

1 jetbrick-template 在 Spring 中的集成方法

你可以按照以下几种方式之一来置 `JetEngine` 在 `Spring` 上下文的实例 (单例模式)。

> **注意**：当同时指定 `configFile` 和 `configProperties` 时，`configProperties` 中的配置会覆盖 `configFile` 中的配置。

```
<!-- 使用 classpath 下面的默认配置文件 -->
<bean id="jetEngine" class="jetbrick.template.JetEngineFactoryBean" />

<!-- 指定配置文件 -->
<bean id="jetEngine" class="jetbrick.template.JetEngineFactoryBean">
  <property name="configFile" value="classpath:META-INF/jetbrick-template.properties" />
</bean>

<!-- 指定配置文件 -->
<bean id="jetEngine" class="jetbrick.template.JetEngineFactoryBean">
  <property name="configFile" value="file:/path/to/jetbrick-template.properties" />
</bean>

<!-- 直接配置属性 -->
<bean id="jetEngine" class="jetbrick.template.JetEngineFactoryBean">
  <property name="configProperties">
    <props>
      <prop key="compile.debug">true</prop>
      ...
    </props>
 </property>
</bean>
```

感谢 应卓 (yingzhor@gmail.com) 提供相关代码。

Jetbrick Template 1.1.3 新功能介绍

原 jetbrick-template 1.1.3 新功能介绍

目录[-]

- Jodd Madvoc 的集成支持
- #for指令的增强
- 增强 asDate() 方法
- 增强 template.path 和 compile.path 的配置功能
- Web 集成模式的默认 Loader 更改为 WebResourceLoader
- 增加 MultipathResourceLoader，支持多个模板路径

更新历史：

- [新增] #50 增加 Jodd Madvoc 的集成支持
- [新增] #56 增加 MultipathResourceLoader，支持多个模板路径
- [增强] #52 增强 asDate() 方法，默认支持更多的格式，比如 ISO8601, RFC 822
- [增强] #55 对#for指令的增强建议
- [增强] #57 增强 template.path 和 compile.path 的配置功能
- [增强] #58 为JetEngineFactoryBean提供构造注入方式的spring配置

Jodd Madvoc 的集成支持

文档看这个： http://subchen.github.io/jetbrick-template/integrate.html#x2314405
例子看这里： https://github.com/subchen/jetbrick-template-webmvc-samples/
例子下载： jetx-samples-jodd.zip

#for指令的增强

原来在 #for 指令循环的内部，只能通过 `${for.index}` 获取当前的索引，现在新增了 3 个属性，如下：

```
for.index   - 循环索引，从 1 开始
for.size    - 循环大小，如果无法获取，返回-1（如果对一个iterator进行循环，则无法获知大小）
for.first   - 是否是第一个
for.last    - 是否是最后一个
```

增强 asDate() 方法

原来的 String.asDate() 扩展方法，只能识别下面的格式：

```
yyyy-MM-dd HH:mm:ss.SSS
yyyy-MM-dd HH:mm:ss
yyyy-MM-dd
HH:mm:ss
```

现在新增如下的日期格式识别：

```
yyyy/MM/dd HH:mm:ss.SSS
yyyy/MM/dd HH:mm:ss
yyyy/MM/dd
yyyy-MM-dd'T'HH:mm:ss:SSSZZ
EEE, dd MMM yyyy HH:mm:ss z
EEE, dd MMM yyyy HH:mm z
EEE, dd MMM yy HH:mm:ss z
EEE, dd MMM yy HH:mm z
dd MMM yyyy HH:mm:ss z
dd MMM yyyy HH:mm z
dd MMM yy HH:mm:ss z
dd MMM yy HH:mm z
```

增强 template.path 和 compile.path 的配置功能

现在 jetbrick-template.properties 配置文件里面可以使用变量了，如下：

```
template.path = ${user.dir}/templates
```

```
template.path = ${webapp.dir}/WEB-INF/templates
```

那么我们支持哪些变量呢？、
其实这些变量都来自于 System.getProperty(name)，只要 System 里有的，都支持。
其中 `webapp.dir` 是个特殊变量，由 Web 集成框架在系统启动的时候，通过 `System.setProperty("webapp.dir", servletContext.getRealPath("/"))` 设置的。

Web 集成模式的默认 Loader 更改为 WebResourceLoader

原来的版本，在 Web 集成模式中， `ResourceLoader` 默认为 `FileSystemResourceLoader`，从 1.1.3 开始，新增 `WebResourceLoader` 作为默认的 Web 模板加载器。

从webapp目录中加载配置如下：

```
template.loader = jetbrick.template.web.WebResourceLoader
template.path = /WEB-INF/templates
```

注意，Web 模式如果还在用 `FileSystemResourceLoader` 的同学，记得把 `template.path` 修改为 `${webapp.dir}/WEB-INF/xxx` 这样的路径，否则就找不到 jetx 文件啦！

增加 MultipathResourceLoader，支持多个模板路径

现在我们支持从多个 path 中载入模板拉！

需要配置如下：

```
template.loader = jetbrick.template.resource.loader.MultipathResourceLoader
template.path = file:/path/to, classpath:/, jar:/path/to/sample.jar, webapp:/WEB-INF/templates
```

注意： `template.path` 支持多种路径，由逗号分隔。每个路径由一个前缀开头，代表相应的 ResouceLoader。具体如下：

- `file:` FileSystemResourceLoader
- `classpath:` ClasspathResourceLoader
- `jar:` JarResourceLoader
- `webapp:` WebResourceLoader
- `<MyClassLoader>:` 用户自定义的 ResourceLoader (完整类名)

Jetbrick Template 1.2.0 新功能介绍

原 jetbrick-template 1.2.0 新功能介绍

目录[-]

- 功能更新
- 模板的预编译选项
- JetxGenerateApp 预编译工具
- 安全管理器：黑白名单
- 新增 ctxpath/webroot 全部变量和函数
- 增加 #tag cache 实现缓存模板片段
- 新增 for.odd, for.even
- 对 Array/List/Map 的 [] 访问，增加安全调用

功能更新

[新增] #38 增加默认的 #tag cache() 实现模板局部缓存功能
[新增] #49 增加模板预编译工具/选项
[新增] #54 增加安全管理器：黑白名单
[新增] #62 在 Web 环境中使用 jetx 时候，建议增加一个隐藏变量
[新增] #63 对 Array/List/Map 的 [] 访问，增加安全调用
[新增] #64 Spell error in JetAnnoations Class name, Should be JetAnnotations.
[新增] #65 给 #for 指令内部对象增加 for.odd 和 for.even 支持.

模板的预编译选项

原来有个编译参数 `compile.always` 是用来控制是否在首次访问的时候，强制进行编译还是从上次编译的结果直接load编译好的class。
而新版本则更进一步，提供了更加灵活的编译策略，目前有 4 种可以选择：

```
compile.strategy = precompile
compile.strategy = always
compile.strategy = auto
compile.strategy = none
```

`compile.always` 已作废。

- `precompile`
  在 JetEngine 初始化的时候，自动获取所有的模板(根据 `template.suffix` 过滤)，然后启动一个独立的线程进行编译。
  这样虽然启动时间会增加，但是后面的模板访问将会非常的快。并且在预编译没有完成期间，应用可以正常访问，不冲突。
- `always` （默认值）
  和原先的 `compile.always = true` 等价。
  就是在模板被首次访问的时候，进行编译。
- `auto`
  和原先的 `compile.always = false` 等价。
  就是在模板被首次访问的时候，如果磁盘中已经存在编译好的 Class 文件（并且源文件没有改变），那么直接加载该 Class 文件,否则进行编译。
- `none`
  改模式下，将不在对模板进行编译。（发布的时候，用户无需发布任何模板源文件）
  用户必须通过 JetxGenerateApp 预编译工具，事先将模板全部编译成 class 文件，并将所有的 class 文件放在 classpath 下面。
  注意： class 文件放在 classpath 下面，而不是 `compile.path` 对应的目录。

> 注意：
> 不管采用什么模式，对于使用 `JetEngine.createTemplate(source)` 直接由源码创建的模板，仍然需要进行编译。

”

JetxGenerateApp 预编译工具

用户可以在外部通过命令行调用，将所有的模板进行预编译，这样生成环境可以直接使用预编译好的 Class 文件，减少首次访问时候的延迟。

```
shell # jar -jar jetbrick-template-xxx.jar \
            -all \
            -src /webapp/templates \
            -d /webapp/generatedClasses \
            -config /webapp/WEB-INF/jetbrick-template.properties

options:
  -all: compile all templates even if errors
  -src <path>: template root directory
  -d <path>: output directory
  -config <file>: config file
```

预编译完成后，请设置 `compile.strategy = none` 。

安全管理器：黑白名单

安全管理器主要用于对模板中调用类，方法和访问字段进行安全管理。主要用于类似于 CMS 系统中，需要用户自定义模板的时候时候，防止用户访问未经授权的内容。

安全管理器默认通过一个黑白名单列表来实现。黑白名单可以放在一个单独的文本文件中，也可以直接放在配置文件中。用户也可以实现自定义的安全管理器。

安全管理器默认禁用，如需使用，需要如下的配置：

```
security.manager = jetbrick.template.parser.JetSecurityManagerImpl
security.manager.file = ${webapp.dir}/WEB-INF/jetx-white-black-list.txt
```

`jetx-white-black-list.txt` 范例内容如下：

```
-java.lang.System.exit
-java.lang.reflect
-java.sql
-javax.tools
-java.io
+java.io.File.getName
+java.io.File.getPath
-sun
```

> “
> 提示：
> 安全管理器只在对模板进行解析编译的时候进行，运行期不会影响任何性能。
> ”

新增 ctxpath/webroot 全部变量和函数

其中 2 个变量：
ctxpath == request.getContextPath()
webroot == http:// : + request.getContextPath()

同时支持函数方式调用：

如：

```
${ctxpath("/index.jetx")}
${webroot("/index.jetx")}
```

增加 #tag cache 实现缓存模板片段

```
#tag application_cache("NEWS_LIST", 60 * 60)
    <ul>
        #for(News news: @NewsMgmt.getAll())
            <li><a href="news/${news.id}">$!{news.title}</a></li>
        #end
    </ul>
#end

#tag session_cache("LOGIN_INFO", 5 * 60)
    Welcome ${loginedUser.name},
    Logined at ${loginedUser.lastdate}
```

```
#end
```

以上新闻信息会被自动缓存 1 小时, 而用户的登录信息会缓存 5 分钟。

如果需要临时清楚缓存，可以通过如下代码：

```
// application_cache
TimedSizeCache cache = (TimedSizeCache) servletContext.getAttribute(TimedSizeCache.CACHE_KEY);
cache.remove("NEWS_LIST");

// session_cache
TimedSizeCache cache = (TimedSizeCache) session.getAttribute(TimedSizeCache.CACHE_KEY);
cache.remove("LOGIN_INFO");
```

新增 for.odd, for.even

据需增强 for 循环状态对象。

```
#for (User user: userlist)
    #if (for.odd)
        ...
    #end
    #if (for.even)
        ...
    #end
#end
```

对 Array/List/Map 的 [] 访问，增加安全调用

现在支持如下调用拉

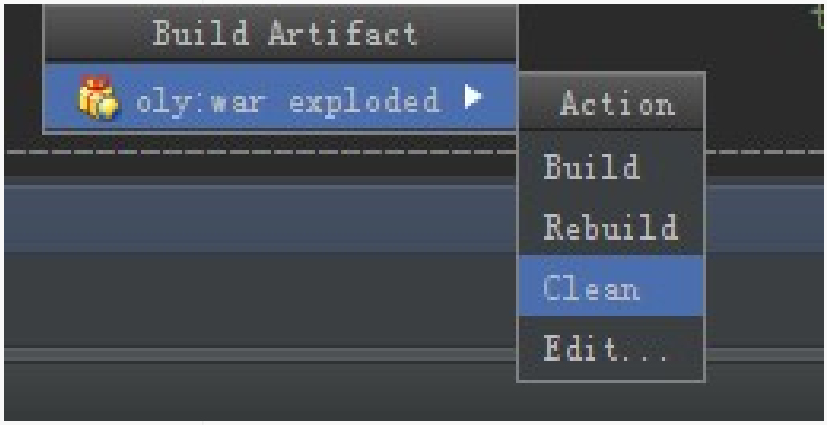
```
array?[index]
list?[index]
map?["key"]
```

JetbrickTemplate升级注意事项

1、如果使用了Maven，记得修改依赖、版本号。特别要注意Clean工程和部署目录！！！否则会报错%>_<%

```
<dependency>
    <groupId>com.github.subchen</groupId>
    <artifactId>jetbrick-template</artifactId>
    <version>1.1.3</version>
</dependency>
```

Intellij Idea clean如图：一定要clean才可以，Rebuild不会删的。。。



如果你没有Clean会导致WEB-IF\lib（部署目录）下老版本和新版本的jar同时存在，报错可能如下：

HTTP Status 500 - Filter execution threw an exception

type Exception report

message Filter execution threw an exception

description The server encountered an internal error that prevented it from fulfilling this request.

exception

```
javax.servlet.ServletException: Filter execution threw an exception
```

root cause

```
java.lang.AbstractMethodError: jetbrick.template.web.WebResourceLoader.initialize(Ljava/lang/String;Ljava/lang/String;)V
        jetbrick.template.JetEngine.createResourceLoader(JetEngine.java:207)
        jetbrick.template.JetEngine.<init>(JetEngine.java:66)
        jetbrick.template.web.JetWebEngine.<init>(JetWebEngine.java:29)
        jetbrick.template.web.JetWebEngineLoader.initJetWebEngine(JetWebEngineLoader.java:86)
        jetbrick.template.web.JetWebEngineLoader.setServletContext(JetWebEngineLoader.java:65)
        jetbrick.template.web.jfinal.JetTemplateRender.render(JetTemplateRender.java:40)
        com.jfinal.core.ActionHandler.handle(ActionHandler.java:92)
        com.jfinal.plugin.druid.DruidStatViewHandler.handle(DruidStatViewHandler.java:74)
        com.jfinal.ext.handler.ContextPathHandler.handle(ContextPathHandler.java:47)
        com.jfinal.core.JFinalFilter.doFilter(JFinalFilter.java:72)
```

note The full stack trace of the root cause is available in the Apache Tomcat/7.0.47 logs.

1.1.3新版本值得注意的一点

```
# 1.3新特性 Web 模式如果还在用 FileSystemResourceLoader 的同学，记得把 template.path 修改为 ${webapp.dir}
# /WEB-INF/xxx 这样的路径，否则就找不到 jetx 文件啦！
template.path                                   =               /WEB-INF/view/themes
template.loader                                 =               jetbrick.template.web.WebResourceLoader
```

调试秘诀-如何在Jetbrick Template中使用debug函数及配置细节

原 如何在 jetbrick-template 中使用 debug函数？

目录[-]

- debug 函数格式
- 开启 debug 日志
- 模板中输出 debug
- 查看 Tomcat 控制台日志

debug 函数格式

jetbrick-template 已经内置了一个 debug 函数： 格式如下：

```
1 void debug(String format, Object... args)
```

> 注意：
>
> 1. 要使用 debug 函数，需要 Slf4j 配合，在对应的 log 实现中打开 debug.
> 2. 具体的 format 参数格式请查看 Slf4j Logger。

开启 debug 日志

- Log4j:

```
1 log4j.logger.jetbrick.template.runtime.JetUtils = DEBUG
```

- Logback

```
1 <logger name="jetbrick.template.runtime.JetUtils" level="DEBUG" />
```

模板中输出 debug

然后，我们就可以在模板中这么用：

```
1 #define(String id, List<UserInfo> users)
2 ${debug("id = {}, users.size = {}.", id, users.size()}
```

查看 Tomcat 控制台日志

```
1 11:41:00.332 [main] DEBUG (JetUtils.java:187) - template debug: id = 16, users.size = 210.
```