Beetl2.0使用说明书20140527

李家智 <xiandafu@126.com>

1. 什么是Beetl

Beetl目前版本是2.0.6，大小650K，相对于其他java模板引擎，具有功能齐全，语法直观,性能超高，以及及其编写的模板容易维护等特点。使得开发和维护模板有很好的体验。是新一代的模板引擎。总得来说，它的特性如下：

- 功能完备：同主流的java模板引擎相比，Beetl具有绝大多数功能。适用于*各种应用场景*，从对响应速度有很高要求的大网站到功能繁多的CMS管理系统都适合。Beetl本身还具有很多独特功能来完成模板编写和维护，这是其他模板引擎所不具有的。
- 非常简单：类似Javascript语法和习俗，只要半小时就能通过半学半猜完全掌握用法。拒绝其他模板引擎那种非人性化的语法和习俗。
- 超高的性能：Beetl 远超过主流java模板引擎性能(引擎性能3-4倍与freemaker，参考附录），而且消耗较低的CPU
- 易于整合：Beetl能很容易的与各种web框架整合，如Spring MVC，JFinal,Struts,Nutz，Jodd，Servlet等。
- 支持模板单独开发和测试，即在MVC架构中，即使没有M和C部分，也能开发和测试模板。
- 扩展和个性化：Beetl支持自定义方法，格式化函数，虚拟属性，标签，和HTML标签. 同时Beetl也支持自定义占位符和控制语句起始符号也支持使用者完全可以打造适合自己的工具包.

> **关于性能**
>
> 通过与主流模板引擎Freemaker，Vecloity以及JSP对比，Beetl均远高于前面三个，这是因为宏观上，通过了优化的渲染引擎，IO的二进制输出，字节码属性访问增强，微观上，通过一维数组保存上下文Context,静态文本合并处理，通过重复使用字节数组来防止java频繁的创建和销毁数组，还使用模板缓存，运行时优化等方法。详情参考附录

独特功能

Beetl有些功能是发展了10多年的模板引擎所不具备的，这些功能非常利于模板的开发和维护，如下

1. 自定义占位符和控制语句起始符号，这有利于减小模板语法对模板的倾入性，比如在html模板中，如果定义控制语句符号是<!--:和 -->,那么，大部分模板文件都能同过浏览器打开。有的使用者仅仅采用了单个符号@ 以及回车换号作为控制语句起始符号，这又能提高开发效率
2. 可单独测试的模板。无需真正的控制层和模型层，Beetl的模板就可以单独开发和测试
3. 同时支持较为松散的MVC和严格的MVC，如果在模板语言里嵌入计算表达式，复杂条件表达式，以及函数调用有干涉业务逻辑嫌疑，你可以禁止使用这些语法。
4. 强大的安全输出，通过安全输出符号！，能在模板变量，变量属性引用，for循环，占位符输出，try-catch中等各个地方提供安全输出，保证渲染正常
5. 模板变量：运行将模板的某一部分输出像js那样赋值个一个变量，稍后再处理。利用模板变量能完成非常复杂的页面布局（简单的布局可使用layout标签函数）
6. 类型推测，能在运行的时候推测模板变量类型，从而优化性能，也可以通过注解的方法显示的说明模板变量属性（这是非必须的，但有助于IDE自动提示功能）
7. 可插拔的设计，错误信息提示，模板引擎缓存机制，模板资源管理，本地调用的安全管理器,严格MVC限制，模板引擎本身都有默认的实现，但又完全可以自定义以适合特定需求
8. 增强的语法，如for-elsefor, select-case，安全输出符号！等，这些语法特别适合模板开发
9. 性能超高,具有最快的模板解释引擎，同时，又有较低的CPU消耗。4倍于国内使用的Freemaker，。适合各类模板应用，如代码生成工具，CMS系统，普通网站，超高访问量的门户系统，和富客户端JS框架整合的后台管理应用

联系作者

作者：闲.大赋 （李家智）

QQ群：219324263

邮件：xiandafu@126.com [mailto:xiandafu@126.com]

Beetl主页：ibeetl.com

源码主页：https://github.com/javamonkey/beetl2.0

在线体验和代码分享 http://ibeetl.com:8080/beetlonline/

工作单位：中国电信

如果有任何疑问或者提交bug，可以直接联系我

2. 基本用法

从GroupTemplate开始

```
StringTemplateResourceLoader resourceLoader = new StringTemplateResourceLoader();
Configuration cfg = Configuration.defaultConfiguration();
GroupTemplate gt = new GroupTemplate(resourceLoader, cfg);
Template t = gt.getTemplate("hello,${name}");
t.binding("name", "beetl");
```

```
String str = t.render();
System.out.println(str);
```

Beetl的核心是GroupTemplate，创建GroupTemplate需要俩个参数，一个是模板资源加载器，一个是配置类，模板资源加载器Beetl内置了4种，分别是

- StringTemplateResourceLoader：字符串模板加载器，用于加载字符串模板，如本例所示
- FileResourceLoader：文件模板加载器，需要一个根目录作为参数构造，，传入getTemplate方法的String是模板文件相对于Root目录的相对路径
- ClasspathResourceLoader：文件模板加载器，模板文件位于Classpath里
- WebAppResourceLoader：用于webapp集成，假定模板跟目录就是WebRoot目录

代码第5行将变量name传入模板里，其值是“Beetl”。 代码第6行是渲染模板，得到输出，template提供了多种获得渲染输出的方法，如下

- tempalte.render() 返回渲染结果，如本例所示
- template.renderTo(Writer) 渲染结果输出到Writer里
- template.renderTo(OutputStream) 渲染结果输出到OutputStream里

1. 关于如何使用模板资源加载器，请参考下一节
2. 如何对模板进行配置，请参考下一节

模板基础配置

Beetl提供不但功能齐全，而且还有你很多独特功能，通过简单的配置文件，就可以定义众多的功能，默认情况下，Configuration类总是会先加载默认的配置文件（位于/org/beetl/core/beetl-default.properties）下，其内容片断如下：

```
#默认配置
ENGINE=org.beetl.core.engine.FastRuntimeEngine
DELIMITER_PLACEHOLDER_START=${
DELIMITER_PLACEHOLDER_END=}
DELIMITER_STATEMENT_START=<%
DELIMITER_STATEMENT_END=%>
DIRECT_BYTE_OUTPUT = FALSE
HTML_TAG_SUPPORT = true
HTML_TAG_FLAG = #
NATIVE_CALL = TRUE
TEMPLATE_CHARSET = UTF-8
ERROR_HANDLER = org.beetl.core.ConsoleErrorHandler
NATIVE_SECUARTY_MANAGER= org.beetl.core.DefaultNativeSecurityManager
RESOURCE_LOADER=org.beetl.core.resource.ClasspathResourceLoader
MVC_STRICT = FALSE

#资源配置，resource后的属性只限于特定ResourceLoader
#classpath 根路径
RESOURCE.root= /
#是否检测文件变化
RESOURCE.autoCheck= true
```

第2行配置引擎实现类，默认即可.

第3,4行指定了占位符号，默认是${}.也可以指定为其他占位符。

第4,5行指定了语句的定界符号，默认是<% %>,也可以指定为其他定界符号

第6行指定IO输出模式，默认是FALSE,即通常的字符输出，再考虑高性能情况下，可以设置成true。相信请参考高级用法

第8,9行指定了支持HTML标签，且符号为#（只能用一个符号标示），默认配置下，模板引擎识别<#tag ></#tag>这样的类似html标签，并能调用相应的标签函数。

第10行指定允许本地Class直接调用

第11行指定模板字符集是UTF-8

第12行指定异常的解析类，默认是ConsoleErrorHandler，他将在render发生异常的时候在后台打印出错误信息（System.out）。

第13行指定了本地Class调用的安全策略

第14行指定了默认使用的模板资源加载器

第15行配置了是否进行严格MVC，通常情况下，此处设置为false.

第20行和22行配置了模板资源加载器的一些属性，如设置根路径为/,即Classpath的顶级路径，并且总是检测模板是否更改

作为新手，通常只需要关注3,4,5,6行定界符的配置，以及11行模板字符集的配置就可以了，其他配置会在后面章节陆续提到

模板开发者可以创建一个beetl.properties的配置文件，此时，该配置文件将覆盖默认的配置文件属性，比如，你的定界符考虑是<!--: 和 -->,则在beetl.properties加入一行即可,并将此配置文件放入Classpath根目录下即可。Configuration.defaultConfiguration()总是先加载系统默认的，然后再加载Beetl.properties的配置属性，如果用重复，用后者代替前者的配置

```
#自定义配置
DELIMITER_STATEMENT_START=<!--:
DELIMITER_STATEMENT_END=-->
```

模板资源加载器

资源加载器是根据String值获取Resource实例的工场类，同时资源加载器还要负责响应模板引擎询问模板是否变化的调用。对于新手来说，无需考虑模板资源加载器如何实现，只需要根据自己场景选择系统提供的三类模板资源加载器即可

字符串模板加载器

在创建GroupTemplate过程中，如果传入的是StringTemplateResourceLoader，则允许通过调用gt.getTemplate(String template)来获取模板实例对象，如2.1所示

文件资源模板加载器

更通常情况下，模板资源是以文件形式管理的，集中放在某一个文件目录下（如webapp的模板根目录就可能是WEB-INF/template里），因此，可以使用FileResourceLoader来加载模板实例，如下代码：

```
String root = System.getProperty("user.dir")+File.separator+"template";
FileResourceLoader resourceLoader = new FileResourceLoader(root,"utf-8");
Configuration cfg = Configuration.defaultConfiguration();
GroupTemplate gt = new GroupTemplate(resourceLoader, cfg);
Template t = gt.getTemplate("/s01/hello.txt");
String str = t.render();
```

```
System.out.println(str);
```

第1行代码指定了模板根目录，即位于项目工程下的template目录　第2行构造了一个资源加载器，并指定字符集为UTF-8（也不许指定，因为配置文件默认就是UTF-8); 第5行通过模板的相对路径/s01/hello.txt来加载模板

Classpath资源模板加载器

还有种常情况下，模板资源是打包到jar文件或者同Class放在一起，因此，可以使用ClasspathResourceLoader来加载模板实例，如下代码：

```
ClasspathResourceLoader resourceLoader = new ClasspathResourceLoader();
Configuration cfg = Configuration.defaultConfiguration();
GroupTemplate gt = new GroupTemplate(resourceLoader, cfg);
Template t = gt.getTemplate("/org/beetl/sample/s01/hello.txt");
String str = t.render();
System.out.println(str);
```

第1行代码指定了模板根目录，即搜索模板的时候从根目录开始，如果new ClasspathResourceLoader("/template"), 则表示搜索/template下的模板。此处用空构造函数，表示搜索路径是根路径，且字符集默认字符集UTF-8.

第4行通过模板的相对路径org/beetl/sample/s01/hello.txt来加载模板

WebApp资源模板加载器

WebAppResourceLoader 是用于web应用的资源模板加载器，默认根路径是WebRoot目录。也可以通过制定root属性来设置相对于WebRoot的的模板根路径，从安全角考虑，建议放到WEB-INF目录下

如下是Jfinal集成 里初始化GroupTemplate的方法

```
Configuration cfg = Configuration.defaultConfiguration();
WebAppResourceLoader resourceLoader = new WebAppResourceLoader();
groupTemplate = new GroupTemplate(resourceLoader, cfg);
```

> WebAppResourceLoader 假定 beetl.jar 是位于 WEB-INF/lib 目录下，因此，可以通过WebAppResourceLoader类的路径来推断出WebRoot路径从而指定模板根路径。所有线上环境一般都是如此，如果是开发环境或者其他环境部符合此假设，你需要调用resourceLoader.setRoot() 来指定模板更路径

定界符与占位符号

Beetl模板语言类似JS语言和习俗，只需要将Beetl语言放入定界符号里即可，如默认的是<% %>,占位符用于静态文本里嵌入占位符用于输出，如下是正确例子

```
<%
var a = 2;
var b = 3;
var result = a+b;
%>
hello 2+3=${result}
```

千万不要在定界符里使用占位符号，因为占位符仅仅嵌在静态文本里，如下例子是错误例子

```
<%
var a = "hi";
var c = ${a}+"beetl"; //应该是var c = a+"beetl"
```

```
%>
```

每次有人问我如上例子为啥不能运行的时候，我总是有点憎恶velocity 带来的这种非人性语法

定界符号里是表达式，如果表达式跟定界符有冲突，可以在表达式里用 “\” 符号，如

```
${[1,2,3]} //输出一个json列表
${ {key:1,value:2 \} } //输出一个json map，} 需要加上\


定界符很占位符 通常还有别的选择，如下定界符

- @ 和回车换行（此时,模板配置DELIMITER_STATEMENT_END= 或者 DELIMITER_STATEMENT_END=null 都可以）
- #: 和回车换行
- <!--： 和 -->
- <!--# 和 -->
- <? 和 ?>

占位符:
- ~ ~
- #{ }
- # #


你也可以与团队达成一致意见来选择团队喜爱择定界符号和占位符号。
```

注释

Beetl语法类似js语法，所以注释上也同js一样： 单行注释采用//

多行注视采用/**/

```
<%
/*此处是一个定义变量*/
var a = 3; //定义一个变量.

/* 以下内容都将被注释
%>

<% */ %>
```

第2行是一个多行注释

第3行是一个单行注释

第5行到第8行采用的是多行注释，因此里面有内容也是注释，模板将不予处理

临时变量定义

在模板中定义的变量成为临时变量，这类似js中采用var 定义的变量，如下例子

```
<%

var a = 3;
var b = 3,c = "abc",d=true,e=null;
var f = [1,2,3];
```

```
var g = {key1:a,key2:c};
var i = a+b;
%>
```

全局变量定义

全局变量是通过template.binding传入的变量, 这些变量能在模板的任何一个地方，包括子模板都能访问到。如java代码里

```
template.binding("list",service.getUserList());

//在模板里
<%
for(user in list){
%>
hello,${user.name};
<%}%>
```

共享变量

共享变量指在所有模板中都可以引用的变量，可过groupTemplate.setSharedVars(Map<String, Object> sharedVars)传入的变量, 这些变量能在 所有模板 的任何一个地方

```
.....
GroupTemplate gt = new GroupTemplate(resourceLoader, cfg);
Map<String,Object> shared = new HashMap<String,Object>();
shared.put("name", "beetl");
gt.setSharedVars(shared);
Template t = gt.getTemplate("/org/beetl/sample/s0208/t1.txt");
String str = t.render();
System.out.println(str);
t = gt.getTemplate("/org/beetl/sample/s0208/t2.txt");
str = t.render();
System.out.println(str);
```

```
//t1.txt
hi,${name}
//t2.txt
hello,${name}
```

模板变量

模板变量是一种特殊的变量，即可以将模板中任何一段的输出赋值到改变量，并允许稍后再其他地方使用，如下代码

```
<%
var content = {
    var c = "1234";
    print(c);
%>
模板其他内容:

<%}; %>
```

第2行定义了一个模板变量content = { ...} ；此变量跟临时变量一样，可以在其他地方使用，最常见的用户是用于复杂的布局。请参考高级用法布局

引用属性

属性引用是模板中的重要一部分，beetl支持属性引用如果javascript的支持方式一样，如下

1 Beetl支持通过”.”号来访问对象的的属性，如果javascript一样。如果User对象有个getName()方法，那么在模板中，可以通过${xxx.name}来访问

2 如果模板变量是数组或者List类，这可以通过[] 来访问，如${userList[0]}

3 如果模板变量是Map类，这可以通过[]来访问，如${map[“name”]},如果key值是字符串类型，也可以使用${map.name}.但不建议这么使用，因为会让模板阅读者误以为是一个Pojo对象

4 Beetl也支持Generic Get方式，即如果对象有一个public Object get(String key)方法，可以通过”.”号或者[]来访问，譬如 ${activityRecord.name}或者${activityRecord[“name”] }都将调用activityRecord的 get(String key)方法。如果对象既有具体属性，又有Generic get（这种模型设计方式是不值得鼓励），则以具体属性优先级高.

5 Beetl也可以通过[]来引用属性，如${user[“name”]} 相当于${user.name}.这跟javascript保持一致。但建议不这么做，因为容易让阅读模板的人误认为这是一个Map类型

6 Beetl 还可以定位额外的对象属性，而无需更改java对象，这叫着虚拟属性，如，对于所有集合，数组，都有共同的虚拟熟悉size.虚拟属性是 “.~” +虚拟1属性名

```
template.binding("list",service.getUserList());
template.binding("pageMap",service.getPage());

//在模板里
总共 ${list.~size}
<%
for(user in list){
%>
hello,${user.name};


<%}%>

当前页${pageMap['page']},总共${pageMap["total"]}
```

算数表达式

Beetl支持类似javascript的算术表达式和条件表达式，如+ - * / % 以及（），以及自增++，自减--

```
<%
var a = 1;
var b = "hi";
var c = a++;
var d = a+100.232;
var e = (d+12)*a;
var f = 122228833330322.1112h
%>
```

Beetl里定义的临时变量类型默认对应的java是Int型或者double类型，对于模板常用情况说，已经够了. 如果需要定义长精度类型（对应java的BigDecimal），则需要在数字末尾加上h以表示这是长精度BigDecimal，其后的计算和输出以及逻辑表达式都将按照长精度类型来考虑。

逻辑表达式

Beetl支持类似Javascript,java的条件表达式 如>，<， == ，!=，>= , <= 以及 !, 还有&&和 || ，还有三元表达式等，如下例子

```
<%
var a = 1;
var b=="good";
var c = null;

if(a!=1&&b=="good"&&c==null){
	.....
}
%>
```

循环语句

Beetl支持丰富的循环方式，如for-in, for(exp;exp;exp)，以及while循环，以及循环控制语句break;continue;　另外，如果没有进入for循环体，还可以执行elsefor指定的语句。

for-in

for-in循环支持遍历集合对象，对于List和数组来说以及Iterator，对象就是集合对象，对于Map来说，对象就是Map.entry,如下俩个例子

```
<%
for(user in userList){
	print(userLP.index);
	print(user.name);

}
%>
```

第三行代码userLP是Beetl隐含定义的变量，能在循环体内使用。其命名规范是item名称后加上LP，他提供了当前循环的信息，如

- userLP.index :当前的索引，从1开始
- userLP.size:集合的长度
- userLP.first 是否是第一个
- userLP.last 是否是最后一个
- userLP.even 是否是偶数个
- userPL.odd 是否是奇数过

> 如何记住后缀是LP，有俩个诀窍，英语棒的是Loop的缩写，拼音好的是老婆的拼音缩写，这可以让程序员每次写到这的时候都会想想老婆（不管有没有，哈哈）

如下是Map使用例子

```
<%
```

```
for(entry in map){
	var key = entry.key;
	var value = entry.value;
	print(value.name);

}
%>
```

for(exp;exp;exp)

对于渲染逻辑更为常见的是经典的for循环语句，如下例子

```
<%
var a = [1,2,3];
for(var i=0;i<a.~size;i++){
	print(a[i]);
}
%>
```

while

对于渲染逻辑同样常见的有的while循环语句，如下例子

```
<%
var i = 0;
while(i<5){
	print(i);
	i++;
}
%>
```

elsefor

不同于通常程序语言，如果没有进入循环体，则不需额外的处理，模板渲染逻辑更常见情况是如果没有进入循环体，还需要做点什么，因此，对于for循环来说，还有elsefor 用来表达如果循环体没有进入，则执行elsefor 后的语句

```
<%
var list = [];
for(item in list){

}elsefor{
    print("未有记录");
}
%>
```

条件语句

if else

同js一样，支持if else,如下例子

```
<%
var a =true;
```

```
var b = 1;
if(a&&b==1){

}else if(a){

}else{

}

%>
```

switch-case

同js一样，支持switch-case,如下例子

```
<%
var b = 1;
switch(b){
	case 0:
		print("it's 0");
		break;
	case 1:
		print("it's 1");
		break;
	default:
		print("error");

}
%>
```

switch变量可以支持任何类型，而不像js那样只能是整形

select-case

select-case 是switch case的增强版。他允许case 里有逻辑表达式，同时，也不需要每个case都break一下，默认遇到合乎条件的case执行后就退出。

```
<%
var b = 1;
select(b){
	case 0,1:
		print("it's small int");
	case 2,3:
		print("it's big int");
	default:
		print("error");

}
%>
```

select 后也不需要一个变量，这样case 后的逻辑表达式将决定执行哪个case.其格式是

```
select(){
```

```
    case boolExp,orBoolExp2:
        doSomething();

}
%>
```

```
<%
var b = 1;
select(){
    case b<1,b>10:
        print("it's out of range");
        break;
    case b==1:
        print("it's 1");
        break;
    default:
        print("error");

}
%>
```

try-catch

通常模板渲染逻辑很少用到try-catch 但考虑到渲染逻辑复杂性，以及模板也有不可控的地方，所以提供try `catch`，在渲染失败的时候仍然能保证输出正常

```
<%
try{
    callOtherSystemView()
}catch(error){
    print("暂时无数据");
}

%>
```

error代表了一个异常，你可以通过error.message 来获取可能的错误信息

也可以省略catch部分，这样出现异常，不做任何操作

虚拟属性

虚拟属性也是对象的属性，但是虚拟的，非模型对象的真实属性，这样的好处是当模板需要额外的用于显示的属性的时候但又不想更改模型，便可以采用这种办法 如beetl内置的虚拟属性. `~size` 针对了数组以及集合类型。

```
${user.gender}
${user.~genderShowName}
```

~genderShowName 是虚拟属性，其内部实现根据boolean变量gender来显示性别

如何完成虚拟属性，请参考高级用法

函数调用

Beetl内置了少量实用函数，可以在Beetl任何地方调用。如下例子是调用date 函数，不传参数情况下，返回当前日期

```
<%
var date = date();
var len = strutil.len("cbd");
println("len="+len);

%>
```

注意函数名支持namespace方式，因此代码第3行调用的函数是strutil.len

定义beetl的方法非常容易，有三种方法

- 实现Function类的call方法，并添加到配置文件里，或者显示的通过代码注册 registerFunction(name,yourFunction)
- 可以直接调用registerFunctionPackage(namespace,yourJavaObject),这时候yourJavaObject里的所有public方法都将注册为Beetl方法，方法名是namespace+".”+方法名
- 可以直接写模板文件并且以html作为后缀，放到root/functions目录下，这样此模板文件自动注册为一个函数，其函数名是该模板文件名。

详情请参考高级用法

Beetl内置函数请参考附录，以下列出了常用的函数

- date 返回一个java.util.Date类型的变量，如 date() 返回一个当前时间; ${date(*2011-1-1,yyyy-MM-dd*)} 返回指定日期
- print 打印一个对象 print(user.name);
- println 打印一个对象以及回车换行符号，回车换号符号使用的是模板本身的，而不是本地系统的. 如果仅仅打印一个换行符，则直接调用println() 即可
- nvl 函数nvl，如果对象为null，则返回第二个参数，否则，返回自己 nvl(user,"不存在")
- isEmpty 判断变量或者表达式是否为空，变量不存在，变量为null，变量是空字符串，变量是空集合，变量是空数组，此函数都将返回true
- has 变量名为参数，判断是否存在此全局变量，如 has(userList),类似于1.x版本的exist("userList"),但不需要输入引号了
- assert 如果表达式为false，则抛出异常
- trunc 截取数字，保留指定的小数位，如trunc(12.456,2) 输出是12.45
- decode 一个简化的if else 结构，如 decode(a,1,"a=1",2,"a=2","不知道了")},如果a是1，这decode输出"a=1"，如果a是2，则输出"a==2"，如果是其他值，则输出"不知道了"
- debug 在控制台输出debug指定的对象以及所在模板文件以及模板中的行数，如debug(1),则输出1 [在3行@/org/beetl/core/lab/hello.txt]
- parseInt 将数字或者字符解析为整形 如 parseInt("123");
- parseDouble 将数字或者字符解析为浮点类型 如parseDouble("1.23")
- range 接收三个参数，初始值，结束值，还有步增（可以不需要，则默认为1），返回一个Iterator，常用于循环中，如for(var i in range(1,5)) {print(i)},将依次打印1234.

安全输出

安全输出是任何一个模板引擎必须重视的问题，否则，将极大困扰模板开发者。Beetl中，如果要输出的模板变量为null，则beetl将不做输出，这点不同于JSP，JSP输出null，也不同于Feemarker，如果没有用!,它会报错.

模板中还有俩中情况会导致模板输出异常

- 有时候模板变量并不存在（譬如子模板里）
- 模板变量为null，但输出的是此变量的一个属性，如${user.wife.name}

针对前俩种种情况，可以在变量引用后加上！以提醒beetl这是一个安全输出的变量。

如${{user.wife.name! },即使user不存在，或者user为null，或者user.wife为null，或者user.wife.name为null beetl都不将输出

可以在!后增加一个常量（字符串，数字类型等），或者另外一个变量，方法，本地调用，作为默认输出，譬如：

${user.wife.name!”单身”}，如果user为null，或者user.wife为null，或者user.wife.name为null，输出”单身”

譬如

${user. birthday!@System.constants.DefaultBir}， 表示如果user为null，或者user. birthday为null，输出System.constants.DefaultBir

还有一种情况很少发生，但也有可能，输出模板变量发生的任何异常，如变量内部抛出的一个异常

这需要使用格式${!(变量)},这样，在变量引用发生任何异常情况下，都不作输出，譬如

${!(user.name)},，beetl将会调用user.getName()方法，如果发生异常，beetl将不会忽略此异常，继续渲染

值得注意的是，在变量后加上!不仅仅可以应用于占位符输出(但主要是应用于占位符输出)，也可以用于表达式中，如：

```
<%

<%
var k = user.name!'N/A'+user.age!;
%>
${k}

%>
```

如果user为null，则k值将为N/A

在有些模板里，可能整个模板都需要安全输出，也可能模板的部分需要安全输出，使用者不必为每一个表达式使用！，可以使用beetl的安全指示符号来完成安全输出 如：

```
<%
DIRECTIVE SAFE_OUTPUT_OPEN;
%>
${user.wife.name}
模板其他内容，均能安全输出……
<%
//关闭安全输出。
DIRECTIVE SAFE_OUTPUT_CLOSE;
```

```
%>
```

Beetl不建议每一个页面都使用DIRECTIVE SAFE_OUTPUT_OPEN，这样，如果如果真有不期望的错误，不容易及时发现，其次，安全输出意味着beetl会有额外的代码检测值是否存在或者是否为null，性能会略差点。所以建议及时关闭安全输出（这不是必须的，但页面所有地方是安全输出，可能不容易发现错误）

在for-in 循环中 ，也可以为集合变量增加安全输出指示符号，这样，如果集合变量为null，也可以不进入循环体，如：

```
<%
var list = null;
for(item in list!){

}eslefor{
    print("no data");
}
%>
```

变量是否存在

判断变量是否存在，可以采用内置的has或者isEmpty方法来判断，参数是变量，如

```
<%
if(has(flag)){
    print("not exit")
}
%>
```

如果需要判断变量是否存在，如果存在，还有其他判断条件，通常都这么写

```
<%
if(has(flag)||flag==0){
    //code
}
%>
```

如果flag不存在，或者flag存在，但值是0，都将执行if语句

但是，有更为简便的方法是直接用安全输出，如

```
<%
if(flag!0==0){
    //code
}
%>
```

flag!0 取值是这样的，如果flag不存在，则为0，如果存在，则取值flag的值，类似三元表达式 has(flag)?falg:0

安全输出表达式

安全输出表达式可以包括

- 字符串常量, 如 ${user.count!"无结果"}
- boolean常量 ${user.count!false}

- 数字常量，仅限于正数，因为如果是负数，则类似减号，容易误用，因此，如果需要表示负数，请用括号，如${user.count!(-1)}
- class直接调用，如${user.count!@User.DEFAULT_NUM}
- 方法调用，如 ${user.count!getDefault() }
- 属性引用，如 ${user.count!user.maxCount }
- 任何表达式，需要用括号

格式化

几乎所有的模板语言都支持格式化，Beetl也不列外，如下例子Beetl提供的内置日期格式

```
<% var date = date(); %>
Today is ${date,dateFormat="yyyy-MM-dd"}.
Today is $date,dateFormat$
```

格式化函数只需要一个字符串作为参数放在=号后面，如果没有为格式化函数输入参数，则使用默认值，dateFormat格式化函数默认值是local

Beetl也允许为指定的java class设定格式化函数，譬如已经内置了对java.util.Date,java.sql.Date 设置了了格式化函数，因此上面的例子可以简化为

```
${date, “yyyy-MM-dd”}.
```

Beetl针对日期好数字类型提供的默认的格式化函数，在org/beetl/core/beetl-default.properties里，注册了

```
##内置的格式化函数
FT.dateFormat =  org.beetl.ext.format.DateFormat
FT.numberFormat =  org.beetl.ext.format.NumberFormat
##内置的默认格式化函数
FTC.java.util.Date = org.beetl.ext.format.DateFormat
FTC.java.sql.Date = org.beetl.ext.format.DateFormat
FTC.java.sql.Time = org.beetl.ext.format.DateFormat
FTC.java.sql.Timestamp = org.beetl.ext.format.DateFormat
FTC.java.lang.Short = org.beetl.ext.format.NumberFormat
FTC.java.lang.Long = org.beetl.ext.format.NumberFormat
FTC.java.lang.Integer = org.beetl.ext.format.NumberFormat
FTC.java.lang.Float = org.beetl.ext.format.NumberFormat
FTC.java.lang.Double = org.beetl.ext.format.NumberFormat
FTC.java.math.BigInteger = org.beetl.ext.format.NumberFormat
FTC.java.math.BigDecimal = org.beetl.ext.format.NumberFormat
FTC.java.util.concurrent.atomic.AtomicLong = org.beetl.ext.format.NumberFormat
FTC.java.util.concurrent.atomic.AtomicInteger = org.beetl.ext.format.NumberFormat
```

标签

所谓标签，即允许处理模板文件里的一块内容，功能等于同jsp tag。如Beetl内置的layout标签

index.html

```
<%
layout("/inc/layout.html",{title:'主题'}){
```

```
%>
Hello,this is main part
<%} %>
```

layout.html

```
title is ${title}
body content ${bodyContent}
footer
```

第1行变量title来自于layout标签函数的参数

第2行bodyContent 是layout标签体{}渲染后的结果

关于layout标签，参考高级主体布局

Beetl内置了另外一个标签是include,允许 include 另外一个模板文件

```
<%
include("/inc/header.html"){}
%>
```

在标签中，{} 内容将依据标签的实现而执行，layout标签将执行{}中的内容，而include标签则忽略标签体内容。

关于如何实现标签函数，请参考高级主体

HTML标签

Beetl 也支持HTML tag形式的标签， 区分beetl的html tag 与 标准html tag。如设定HTML_TAG_FLAG=#，则如下html tag将被beetl解析

```
<#footer style=”simple”/>
<#richeditor id=”rid” name=”rname”  maxlength=”${maxlength}”> ${html} ...其他模板内容  </#richdeitor>
<#html:input  id=’aaaa’ />
```

```
<%if(style==’simple’){%>
 请联系我 ${session.user.name}
<%}else{%>
请联系我 ${session.user.name},phone:${session.user.phone}
<%}%>
```

如下还包含了自定义html标签一些一些规则

-
- 可以在自定义标签里引用标签体的内容，标签体可以是普通文本，beetl模板，以及嵌套的自定义标签等。如上<#richeditor 标签体里，可用“tagBody”来引用
- HTML自定义标签 的属性值均为字符串 如<#input value=”123” />,在input.tag文件里 变量value的类型是字符串
- 可以在属性标签里引用beetl变量，如<#input value=”${user.age}” />，此时在input.tag里，value的类型取决于user.age
- 在属性里引用beetl变量，不支持格式化，如<#input value=”${user.date,‘yyyy-MM-dd’ }” />,如果需要格式化，需要在input.tag文件里自行格式化

- html tag 属性名将作为 其对应模板的变量名。
- 默认机制下，全局变量都将传给html tag对应的模板文件，这个跟include一样。当然，这机制也可以改变，对于标签来说，通常是作为一个组件存在，也不一定需要完全传送所有全局变量，而只传送（request, session, 这样变量），因此需要重新继承org.beetl.ext.tag.HTMLTagSupportWrapper.并重载callHtmlTag方法。并注册为htmltag标签。具体请参考https://github.com/javamonkey/beetl2.0/blob/master/beetl-core/src/test/java/org/beetl/core/tag/HtmlTagTest.java
- 如果在htmltag目录下找不到同名的模板文件，则beetl再次寻找是否有同名注册的标签函数（关于如何写标签函数，参考高级用法），如果存在，则调用标签函数

直接调用java方法和属性

```
${@user.getMaxFriend("lucy")}
${@user.maxFriend[0].getName()}
${@com.xxxx.constants.Order.getMaxNum()}
<%
var max = @com.xxxx.constants.Order.MAX_NUM;
%>
```

可以调用instance的public方法和属性，也可以调用静态类的属性和方法 ,需要加一个/@指示此调用是直接调用class，其后的表达式是java风格的。

- GroupTemplate可以配置为不允许直接调用Class，具体请参考配置文件
- 也可以通过安全管理器配置到底哪些类Beetl不允许调用，具体请参考高级用法。默认情况， java. lang. Runtime, 和 java.lang.Process不允许在模板里调用
- 请按照java规范写类名和方法名，属性名。这样便于beetl识别到底调用的是哪个类，哪个方法
- 可以省略包名，只用类名。beetl将搜索包路径找到合适的类（需要设置配置IMPORT_PACKAGE=包名;包名，或者调用Configuration.addPkg)方法具体请参考附件配置文件说明

严格MVC控制

如果在配置文件中设置了严格MVC，则以下语法将不在模板文件里允许，否则将报出STRICK_MVC 错误

- 定义变量，为变量赋值, 如var a = 12是非法的
- 算术表达式 如${user.age+12}是非法的
- 除了只允许布尔以外，不允许逻辑表达式和方法调用 如if(user.gender==1)是非法的
- 方法调用，如${subString(string,1)}是非法的
- Class方法和属性调用，如${@user.getName()}是非法的
- 严格的MVC，非常有助于逻辑与视图的分离，特别当逻辑与视图是由俩个团队来完成的。如果你嗜好严格MVC，可以调用groupTemplate.enableStrict()

通过重载AntlrProgramBuilder，可以按照自己的方法控制到底哪些语法是不允许在模板引擎中出现的，但这已经超出了Beetl模板的基础使用

指令

指令格式为： DIRECTIVE 指令名 指令参数（可选） Beetl目前支持安全输出指令，分别是

- DIRECTIVE SAFE_OUTPUT_OPEN ; 打开安全输出功能，此指令后的所有表达式都具有安全输出功能，
- DIRECTIVE SAFE_OUTPUT_CLOSE ; 关闭安全输出功能。详情参考安全输出
- DIRECTIVE DYNAMIC varName1,varName2 ...指示后面的变量是动态类型，Beetl应该考虑为Object. 也可以省略后面的变量名，则表示模板里所有变量都是Object

```
<% DIRECTIVE DYNAMIC idList;
for(value in idList) .....
```

DYNAMIC　通常用在组件模板里，因为组件模板可以接收任何类型的对象。如列表控件，可以接收任何含有id和value属性的对象。

1 注意 DYNAMIC 后的变量名也允许用引号，这主要是兼容Beetl1.x版本

2 Beetl1.x 指令都是大写，当前版本也允许小写，如 directive dynamic idList

类型声明

Beetl 本质上还是强类型的模板引擎，即模板每个变量类型是特定的，在模板运行过程中，beetl 会根据全局变量自动推测出模板中各种变量和表达式类型。 也可以通过类型申明来说明beetl全局变量的类型，如下格式

```
<%
/**
*@type (List<User> idList,User user)
*/
for(value in idList) .....
```

类型申明必须放到多行注释里，格式是@type(...),里面的申明类似java方法的参数申明。正如你看到的类型申明是在注释里，也就表明了这在Beetl模板引擎中不是必须的，或者你只需要申明一部分即可，之所以提供可选的类型说明，是因为

- 提高一点性能
- 最重要的是，提高了模板的可维护性。可以让模板维护者知道变量类型，也可以让未来的ide插件根据类型声明来提供属性提示，重构等高级功能

需要注意的是，如果在类型声明里提供的是类名，而不是类全路径，这样必须在配置文件里申明类的搜索路径（（需要设置配置IMPORT_PACKAGE=包名;包名，或者调用Configuration.addPkg)），默认的搜索路径有java.util.* 和 java.lang.*

错误处理

Beetl能较为详细的显示错误原因，包括错误行数，错误符号，错误内容附近的模板内容，以及错误原因，如果有异常，还包括异常和异常信息。 默认情况下，仅仅在控制台显示，如下代码：

```
<%
var a = 1;
var b = a/0;
%>
```

运行此模板后，错误提示如下：

```
>>DIV_ZERO_ERROR:0 位于3行 资源:/org/beetl/sample/s0125/error1.txt
1|<%
2|var a = 1;
3|var b = a/0;
4|%>
```

```
<%
var a = 1;
var b = a
var c = a+2;
%>
```

运行此模板后

```
>>缺少符号(PARSER_MISS_ERROR):缺少输入 ';' 在 'var' 位于4行 资源:/org/beetl/sample/s0125/error2.txt
1|<%
2|var a = 1;
3|var b = a
4|var c = a+2;
5|%>
```

1　默认的错误处理器仅仅像后台打印错误，并没有抛出异常，如果需要在render错误时候抛出异常到控制层，则可以使用org.beetl.core.ReThrowConsoleErrorHandler。不仅打印异常，还抛出BeetlException，

2 可以自定义异常处理器，比如把错误输出到 作为渲染结果一部分输出，或者输出更美观的html内容等，具体参考高级用法

Beetl小工具

BeetlKit 提供了一些便利的方法让你立刻能使用Beetl模板引擎。提供了如下方法

- public static String render(String template, Map<String, Object> paras) 渲染模板，使用paras参数，渲染结果作为字符串返回
- public static void renderTo(String template, Writer writer, Map<String, Object> paras) 渲染模板，使用paras参数，渲染结果作为字符串返回
- public static void execute(String script, Map<String, Object> paras) 执行某个脚本
- public static Map execute(String script, Map<String, Object> paras, String[] locals) 执行某个脚本，将locals指定的变量名和模板执行后相应值放入到返回的Map里
- public static Map executeAndReturnRootScopeVars(String script) 执行某个脚本，返回所有顶级scope的所有变量和值
- public static String testTemplate(String template, String initValue) 渲染模板template，其变量来源于intValue脚本运行的结果，其所有顶级Scope的变量都将作为template的变量

```
String template = "var a=1,c=2+1;";
Map result = executeAndReturnRootScopeVars(template);
System.out.println(result);
//输出结果是{c=3, a=1}
```

琐碎功能

- 对齐：　　我发现别的模板语言要是做到对齐，非常困难,Beetl你完全不用担心，比如velocty，stringtemlate，freemarker例子都出现了不对齐的情况，影响了美观，Beetl完全无需担心输出对齐
- Escape：可以使用\ 做escape 符号，如\$monkey\$ 将作为一个普通的文本，输出为$monkey$.再如为了在钱后加上美元符号（占位符恰好又是美元符号）可以用这俩种方式hello,it's $money$\$，或者Hello,it's $money+"\$"$ 。如果要输出\符号本生，则需要用俩个\\,这点与javascript，java 语义一致.

3. 高级用法

配置GroupTemplate

Beetl建议通过配置文件配置配置GroupTemplate，主要考虑到未来可能IDE插件会支持Beetl模板，模板的属性，和函数等如果能通过配置文件获取，将有助于IDE插件识别。 配置GroupTemplate有俩种方法

- 配置文件：　默认配置在/org/beetl/core/beetl-default.properties　里，Beetl首先加载此配置文件，然后再加载classpath里的beetl.properties,并用后者覆盖前者。配置文件通过Configuration类加载，因此加载完成后，也可以通过此类API来修改配置信息
- 通过调用GroupTemplate提供的方法来注册函数，格式化函数，标签函数等

配置文件分为三部分，第一部分是基本配置，在第一节讲到过。第二部分是资源类配置，可以在指定资源加载类，以及资源加载器的属性，如下

```
RESOURCE_LOADER=org.beetl.core.resource.ClasspathResourceLoader
#资源配置，resource后的属性只限于特定ResourceLoader
#classpath 根路径
RESOURCE.root= /
#是否检测文件变化
RESOURCE.autouCheck= true
```

第一行指定了类加载器，第二行指定了模板根目录的路径，此处/ 表示位于classpath 根路径下，第三行是否自动检测模板变化，默认为true，开发环境下自动检测模板是否更改。关于如何如何自定义ResouceLoader，请参考下一章

配置文件第三部分是扩展部分，如方法，格式化函数等

```
##### 扩展 ###############
## 内置的方法
FN.date = org.beetl.ext.fn.DateFunction
FN.nvl = org.beetl.ext.fn.NVLFunction
.................
##内置的功能包
FNP.strutil = org.beetl.ext.fn.StringUtil

##内置的格式化函数
FT.dateFormat =  org.beetl.ext.format.DateFormat
FT.numberFormat =  org.beetl.ext.format.NumberFormat
.................

##内置的默认格式化函数
FTC.java.util.Date = org.beetl.ext.format.DateFormat
FTC.java.sql.Date = org.beetl.ext.format.DateFormat

## 标签类
```

```
TAG.include= org.beetl.ext.tag.IncludeTag
TAG.includeFileTemplate= org.beetl.ext.tag.IncludeTag
TAG.layout= org.beetl.ext.tag.LayoutTag
TAG.htmltag= org.beetl.ext.tag.HTMLTagSupportWrapper
```

fn前缀表示Function，fnp前缀表示FunctionPackage，FT表示format函数，FTC表示类的默认Format函数，TAG表示标签类。Beetl强烈建议通过配置文件加载扩展。以便随后IDE插件能识别这些注册函数

自定义方法

实现Function

```
public class Print implements Function
{

    public String call(Object[] paras, Context ctx)
    {
        Object o = paras[0];

        if (o != null)
        {
            try
            {
                ctx.byteWriter.write(o.toString());
            }
            catch (IOException e)
            {
                throw new RuntimeException(e);
            }
        }
        return "";

    }
```

call方法有俩个参数，第一个是数组，这是由模板传入的，对应着模板的参数，第二个是Context，包含了模板的上下文，主要提供了如下属性

- byteWriter 输出流
- template 模板本身
- gt GroupTemplate
- globalVar 该模板对应的全局变量
- byteOutputMode 模板的输出模式，是字节还是字符
- safeOutput 模板当前是否处于安全输出模式
- 其他属性建议不熟悉的开发人员不要乱动

> 1 call方法要求返回一个Object，如果无返回，返回null即可
>
> 2 为了便于类型判断，call方法最好返回一个具体的类，如date函数返回的就是java.util.Date
>
> 3 call方法里的任何异常应该抛出成Runtime异常

使用普通的java类

尽管实现Function对于模板引擎来说，是效率最高的方式，但考虑到很多系统只有util类，这些类里的方法仍然可以注册为模板函数。其规则很简单，就是该类的所有public方法。如果需还要Contrext 变量，则需要在方法最后一个参数加上Context即可，如

```
public class util
{

    public String print(Object a, Context ctx)
    {
        ..............

    }
```

注意

> 1 从beetl效率角度来讲，采用的普通java类尽量少同名方法。
>
> 2 如果确实有同名方法，尽量少的具有同样参数长度的同名方法
>
> 3 如果以上俩种情况避免不了，则方法调用效率比较低，beetl将根据参数类型在运行时刻判断是调用哪个方法。找到第一个能调用的方法为此次调用的function。这和java是一样的。

使用模板文件作为方法

可以不用写java代码，模板文件也能作为一个方法。默认情况下，需要将模板文件放到Root的functions目录下，且扩展名为.html(可以通过ResourceLoader配置文件属性来修改此俩默认值) 方法参数分别是para1,para2.....

如下root/functions/page.html

```
<%
//para0,para1 由函数调用传入
var current = para0,total = para1,style=para2!'simple'
%>
当前页面 ${current},总共${total}
```

则在模板中

```
<%
page(current,total);
%>
```

也可以在functions建立子目录，这样function则具有namespace，其值就是文件夹名

自定义格式化函数

需要实现Format接口

```
public class DateFormat implements Format
{

    public Object format(Object data, String pattern)
```

```
    {
        if (data == null)
            return null;
        if (Date.class.isAssignableFrom(data.getClass()))
        {
            SimpleDateFormat sdf = null;
            if (pattern == null)
            {
                sdf = new SimpleDateFormat();
            }
            else
            {
                sdf = new SimpleDateFormat(pattern);
            }
            return sdf.format((Date) data);

        }
        else
        {
            throw new RuntimeException("Arg Error:Type should be Date");
        }
    }
```

data 参数表示需要格式化的对象，pattern表示格式化模式，开发时候需要考虑pattern为null的情况

自定义标签

标签形式有俩种，一种是标签函数，第二种是html tag。第二种实际上在语法解析的时候会转化成第一种，其实现是HTMLTagSupportWrapper，此类将会寻找root/htmltag目录下同名的标签文件作为模板来执行。类似普通模板一样，在此就不详细说了

标签函数

标签函数类似jsp2.0的实现方式，需要实现Tag类的render方法即可

```
public class DeleteTag extends Tag
{

    @Override
    public void render()
    {
        // do nothing,just ignore body
        ctx.byteWriter.write("被删除了，付费可以看")

    }

}
```

如上一个最简单的Tag，将忽略tag体，并输出内容

```
public class XianDeDantengTag extends Tag
{

    @Override
    public void render()
```

```
    {
        doBodyRender();

    }

}
```

此类将调用父类方法doBodyRender，渲染tag body体

```
public class CompressTag extends Tag
{

    @Override
    public void render()
    {
        BodyContent  content = getBodyContent();
        String content = content.getBody();
        String zip = compress(conent);
        ctx.byteWriter.write(zip);

    }

}
```

此类将调用父类方法getBodyContent ， 获得tag body后压缩输出

tag类提供了如下属性和方法供使用

- args 传入标签的参数
- gt GroupTemplate
- ctx Context
- bw 当前的输出流
- bs 标签体对应的语法树，不熟悉勿动

自定义虚拟属性

可以为特定类注册一个虚拟属性，也可以为一些类注册虚拟属性

- public void registerVirtualAttributeClass(Class cls, VirtualClassAttribute virtual) 实现VirtualClassAttribute方法可以为特定类注册一个需要属性，如下代码：

```
gt.registerVirtualAttributeClass(User.class, new VirtualClassAttribute() {

    @Override
    public String eval(Object o, String attributeName, Context ctx)
    {

        User user = (User) o;
        if(attributeName.equals("ageDescritpion")){
            if (user.getAge() < 10)
            {
                return "young";
```

```
            }
            else
            {
                return "old";
            }
        }


    }

});
```

User类的所有虚拟属性将执行eval方法，此方法根据年纪属性来输出对应的描述。

- public void registerVirtualAttributeEval(VirtualAttributeEval e) 为一些类注册需要属性，VirtualAttributeEval.isSupport方法将判断是否应用虚拟属性到此类

如下是虚拟属性类的定义

```
public interface VirtualClassAttribute
{
    public Object eval(Object o, String attributeName, Context ctx);

}


public interface VirtualAttributeEval extends VirtualClassAttribute
{

    public boolean isSupport(Class c, String attributeName);
}
```

自定义资源加载器

如果模板资源来自其他地方，如数据库，或者混合了数据库和物理文件，则需要自定义一个资源加载器。资源加载器需要实现ResourceLoader类。如下：

```
public interface ResourceLoader
{

    /**
     * 根据key获取Resource
     *
     * @param key
     * @return
     */
    public Resource getResource(String key);

    /** 检测模板是否更改，每次渲染模板前，都需要调用此方法，所以此方法不能占用太多时间，否则会影响渲染功能
     * @param key
     * @return
     */
    public boolean isModified(Resource key);

    /**
```

```
     * 关闭ResouceLoader，通常是GroupTemplate关闭的时候也关闭对应的ResourceLoader
     */
    public void close();

    /** 一些初始化方法
     * @param gt
     */
    public void init(GroupTemplate gt);

}
```

init方法可以初始化GroupTemplate，比如读取配置文件的root属性，autoCheck属性，字符集属性，以及加载functions目录下的所有模板方法 如FileResourceLoader 的 init方法

```
@Override
    public void init(GroupTemplate gt)
    {
        Map<String, String> resourceMap = gt.getConf().getResourceMap();
        if (this.root == null)
        {
            this.root = resourceMap.get("root");

        }
        if (this.charset == null)
        {
            this.charset = resourceMap.get("charset");

        }
        if (this.functionSuffix == null)
        {
            this.functionSuffix = resourceMap.get("functionSuffix");
        }

        this.autoCheck = Boolean.parseBoolean(resourceMap.get("autoCheck"));

        File root = new File(this.root, this.functionRoot);
        this.gt = gt;
        if (root.exists())
        {
            readFuntionFile(root, "", "/".concat(functionRoot).concat("/"));
        }

    }
```

readFuntionFile 方法将读取functions下的所有模板，并注册为方法

```
protected void readFuntionFile(File funtionRoot, String ns, String path)
    {
        String expected = ".".concat(this.functionSuffix);
        File[] files = funtionRoot.listFiles();
        for (File f : files)
        {
            if (f.isDirectory())
            {
                //读取子目录
```

```
                readFuntionFile(f, f.getName().concat("."), path.concat(f.getName()).concat("/"));
            }
            else if (f.getName().endsWith(functionSuffix))
            {
                String resourceId = path + f.getName();
                String fileName = f.getName();
                fileName = fileName.substring(0, (fileName.length() - functionSuffix.length() - 1));
                String functionName = ns.concat(fileName);
                FileFunctionWrapper fun = new FileFunctionWrapper(resourceId);
                gt.registerFunction(functionName, fun);
            }
        }
    }
```

Resource类需要实现OpenReader方法，以及isModified方法。对于模板内容存储在数据库中，openReader返回一个Clob，isModified 则需要根据改模板内容对应的lastUpdate（通常数据库应该这么设计）来判断模板是否更改

```
public abstract class Resource
{


    /**
     * 打开一个新的Reader
     *
     * @return
     */
    public abstract Reader openReader();

    /**
     * 检测资源是否改变
     *
     * @return
     */
    public abstract boolean isModified();



』
```

参考例子可以参考beetl自带的ResourceLoader

自定义错误处理器

错误处理器需要实现ErrorHandler接口的processExcption(BeetlException beeExceptionos, Writer writer);

- beeExceptionos，模板各种异常
- writer 模板使用的输出流。系统自带的并未采用此Writer，而是直接输出到控制台

自定义错误处理可能是有多个原因，比如

1 想将错误输出到页面而不是控制台

2 错误输出美化一下，而不是自带的格式

3 错误输出的内容做调整，如不输出错误行的模板内容，而仅仅是错误提示

4 错误输出到日志系统里

5 不仅仅输出日志，还抛出异常。默认自带的不会抛出异常，ReThrowConsoleErrorHandler 继承了ConsoleErrorHandler方法，打印异常后抛出

```
public class ReThrowConsoleErrorHandler extends ConsoleErrorHandler
{

	@Override
	public void processExcption(BeetlException ex, Writer writer)
	{

		super.processExcption(ex, writer);
		throw ex;

	}

}
```

beetl 提供 ErrorInfo类来wrap BeetlException，转化为较为详细的提示信息，他具有如下信息

- type 一个简单的中文描述
- errorCode 内部使用的错误类型标识
- errorTokenText 错误发生的节点文本
- errorTokenLine 错误行
- msg 错误消息，有可能没有，因为有时候errorCode描述的已经很清楚了
- cause 错误的root 异常，也可能没有。

BeetlException 也包含了一个关键信息就是 resourceId，即出错所在的模板文件

自定义安全管理器

所有模板的本地调用都需要通过安全管理器校验，默认需要实现NativeSecurityManager 的public boolean permit(String resourceId, Class c, Object target, String method) 方法

如下是默认管理器的实现方法

```
public class DefaultNativeSecurityManager implements NativeSecurityManager
{

	@Override
	public boolean permit(String resourceId, Class c, Object target, String method)
	{
		String name = c.getSimpleName();
		String pkg = c.getPackage().getName();
		if (pkg.startsWith("java.lang"))
		{
			if (name.equals("Runtime") || name.equals("Process") || name.equals("ProcessBuilder")
					|| name.equals("System"))
			{
				return false;
			}
```

```
        }

        return true;
    }

}
```

注册全局共享变量

GroupTemplate.setSharedVars(Map<String, Object> sharedVars)

布局

布局可以通过Beetl提供的include,layout 以及模板变量来完成。模板变量能完成复杂的布局

- 采用layout include

```
<%
//content.html内容如下：
layout("/inc/layout.html"){%>
this is 正文
.....
<%}%>
```

如上一个子页面将使用layout布局页面，layout 页面内容如下

```
<%include("/inc/header.html"){} %>
this is content:${layoutContent}
this is footer:
```

layoutContent 是默认变量，也可以改成其他名字，具体请参考layout标签函数

- 采用模板变量和include

```
<%
    var jsPart = {
%>
 web页面js部分

<%};%>

<%
    var htmlPart = {
%>
 web页面html部分

<%};
 include("/inc/layout.html",{jsSecition:jsPart,htmlSection:htmlPart})
%>
```

layout.html页面如下：

```
<body>
<head>
```

```
${jsSection}
</head>
<body>
.......
${htmlSection}
</body>
```

模板测试

模板允许被单独测试，即使缺少模型。beetl可以通过BeetlKit类来单独测试模板，BeetlKi的方法 public static String testTemplate(String template, String initValue)。要求输入俩个String，第一个是模板本身，第二个是模板的变量，BeetlKit会计算initValue,并得出所有顶层的所有变量周围模型，来渲染Template，比如

```
String initValue = "var a=1;var session = {user:{id,1,name:'xiandafu'}}";
String template = "${session["user"].name}";
String result = BeetlKit.testTemplate(template,initValue);
```

运行testTemplate，可以返回"xiandafu";

Beetl在线体验就是采用BeetlKit来完成的

性能优化

Beetl性能已经很快了，有些策略能更好提高性能

- 使用二进制输出，此策略可以使模板在语法分析的时候将静态文本转化为二进制，省去了运行时刻编码时间，这是主要性能提高方式。但需要注意，此时需要提供一个二进制输出流，而不是字符流，否则性能反而下降
- 使用FastRuntimeEngine，默认配置。 此引擎能对语法树做很多优化，从而提高运行性能，如生成字节码来访问属性而不是传统的反射访问。关于引擎，可能在新的版本推出更好的引擎，请随时关注。
- 通过@type 来申明全局变量类型，这不能提高运行性能，但有助于模板维护
- 自定义ResourceLoader的isModified必须尽快返回，因此每次渲染模板的时候都会调用此方法

为什么Beetl性能这么好……(待续)

分布式缓存模板

Beetl模板引擎模板在同一个虚拟机里缓存Beetl 脚本。也可以将缓存脚本到其他地方，只要实现Cache接口，并设置ProgramCacheFactory.cache即可，这样GroupTemplate将从你提供的Cache中存取Beetl脚本

> 此功能未被很好测试

定制模板引擎

Beetl在线体验（http://ibeetl.com:8080/beetlonline/）面临一个挑战，允许用户输入任何脚本做练习或者分享代码。但又需要防止用户输入恶意的代码，如

```
<%
for(var i=0;i<10000000;i++){
    //其他代码
}
```

```
%>
```

此时，需要定制模板引擎，遇到for循环的时候，应该限制循环次数，譬如，在线体验限制最多循5次，这是通过定义替换GeneralForStatement类来完成的，这个类对应了for(exp;exp;exp)，我们需要改成如下样子：

```
class RestrictForStatement extends GeneralForStatement
{
    public RestrictForStatement(GeneralForStatement gf)
    {
        super(gf.varAssignSeq, gf.expInit, gf.condtion, gf.expUpdate, gf.forPart, gf.elseforPart, gf.token);
    }

    public void execute(Context ctx)
    {
        if (expInit != null)
        {
            for (Expression exp : expInit)
            {
                exp.evaluate(ctx);
            }
        }
        if (varAssignSeq != null)
        {
            varAssignSeq.execute(ctx);
        }


        boolean hasLooped = false;
        int i = 0;
        for (; i < 5; i++)
        {
            boolean bool = (Boolean) condtion.evaluate(ctx);
            if (bool)
            {
                hasLooped = true;
                forPart.execute(ctx);
                switch (ctx.gotoFlag)
                {
                    case IGoto.NORMAL:
                        break;
                    case IGoto.CONTINUE:
                        ctx.gotoFlag = IGoto.NORMAL;
                        continue;
                    case IGoto.RETURN:
                        return;
                    case IGoto.BREAK:
                        ctx.gotoFlag = IGoto.NORMAL;
                        return;
                }

            }
            else
            {
                break;
            }
```

```
            if (this.expUpdate != null)
            {
                for (Expression exp : expUpdate)
                {
                    exp.evaluate(ctx);
                }
            }

        }

        if (i >= 5)
        {
            try
            {
                ctx.byteWriter.write("--Too may Data in loop, Ignore the left Data  for Online Engine--");
                ctx.byteWriter.flush();


            }
            catch (IOException e)
            {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }

    }

    @Override
    public void infer(InferContext inferCtx)
    {
        super.infer(inferCtx);

    }

}
```

尽管上面代码很复杂，但实际上是改写了原来的GeneralForStatement，将原来的24行while(true) 替换成for (; i < 5; i++) 用来控制最大循环，并且62行检测如果循环退出后，i等于5，则提示Too Many Data in Loop.

现在需要将此类替换原有的GeneralForStatement，

```
public class OnlineTemplateEngine extends DefaultTemplateEngine
{
    public Program createProgram(Resource resource, Reader reader, Map<Integer, String> textMap, String cr,
            GroupTemplate gt)
    {
        Program program = super.createProgram(resource, reader, textMap, cr, gt);
        Statement[] sts = program.metaData.statements;
        StatementParser parser = new StatementParser(sts, gt, resource.getId());
        //parser.addListener(WhileStatement.class, new RestrictLoopNodeListener());
        parser.addListener(GeneralForStatement.class, new RestrictLoopNodeListener());

        parser.parse();
```

```
        return program;
    }
}
```

- 因为是在线体验，所以继承了DefaultTemplateEngine，在其他场景，也可以集成FastRuntimeTemplateEngine
- StatementParser 是关键类，他允许对模板的Program进行解析，并替换其中的Statement。parser.addListener方法接受俩个参数，第一个是需要找的类，第二个是执行的监听器。

```
class RestrictLoopNodeListener implements Listener
{

    @Override
    public Object onEvent(Event e)
    {
        Stack stack = (Stack) e.getEventTaget();
        Object o = stack.peek();
        if (o instanceof GeneralForStatement)
        {
            GeneralForStatement gf = (GeneralForStatement) o;
            RestrictForStatement rf = new RestrictForStatement(gf);
            return rf;
        }

        else
        {
            return null;
        }
    }
```

该监听器返回一个新的RestrictForStatement 类，用来替换来的GeneralForStatement。如果返回null，则不需替换。这通常发生在你仅仅通过修改该类的某些属性就可以的场景

完成这些代码后，在配置文件中申明使用新的引擎

```
ENGINE=org.bee.tl.online.OnlineTemplateEngine
```

这样就完成了模板引擎定制。

4. Web集成

集成技术开发指南

Web集成模块需要向模板提供web标准的变量，做如下说明

- session 提供了session会话，模板通过session["name"],或者session.name 引用session里的变量
- request 标准的HTTPSerlvetRequest,可以在模板里引用request属性（getter），如${request.url}。
- ctxPath Web应用ContextPath
- servlet 是WebVariable的实例，包含了HTTPSession,HTTPSerlvetRequest,HTTPSerlvetResponse.三个属性，模板中可以通过request.response,sessin 来引用，如 ${serlvet.request.url};
- request 中的所有attribute.在模板中可以直接通过attribute name 来引用，如在controller层request.setAttribute("user",user),则在模板中可以直接用${user.name}

- 所有的GroupTemplate的共享变量

Web集成模块还需要根据是否采用二进制输出模板，来决定是否采用OutputStream 还是Writer。

Beetl默认提供了WebRender用于帮助web集成开发，所有内置的集成均基于此方法。如果你认为Beetl内置的各个web框架集成功能不够，你可以继承此类，或者参考此类源码重新写，其代码如下

```
package org.beetl.ext.web;

import java.io.IOException;
import java.io.OutputStream;
import java.io.Writer;
import java.util.Enumeration;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.beetl.core.GroupTemplate;
import org.beetl.core.Template;
import org.beetl.core.exception.BeetlException;

/**
 * 通常web渲染的类，将request变量赋值给模板，同时赋值的还有session,request,ctxPath
 * 其他框架可以继承此类做更多的定制
 * @author joelli
 *
 */
public class WebRender
{
	GroupTemplate gt = null;

	public WebRender(GroupTemplate gt)
	{
		this.gt = gt;
	}

	/**
	 * @param key 模板资源id
	 * @param request
	 * @param response
	 * @param args 其他参数，将会传给modifyTemplate方法
	 */
	public void render(String key, HttpServletRequest request, HttpServletResponse response, Object... args)
	{
		Writer writer = null;
		OutputStream os = null;
		try

		{
			//				response.setContentType(contentType);
			Template template = gt.getTemplate(key);
			Enumeration<String> attrs = request.getAttributeNames();

			while (attrs.hasMoreElements())
			{
				String attrName = attrs.nextElement();
```

```
                template.binding(attrName, request.getAttribute(attrName));

            }
            WebVariable webVariable = new WebVariable();
            webVariable.setRequest(request);
            webVariable.setResponse(response);
            webVariable.setSession(request.getSession());

            template.binding("session", new SessionWrapper(webVariable.getSession()));

            template.binding("servlet", webVariable);
            template.binding("request", request);
            template.binding("ctxPath", request.getContextPath());

            modifyTemplate(template, key, request, response, args);

            if (gt.getConf().isDirectByteOutput())
            {
                os = response.getOutputStream();
                template.renderTo(os);
            }
            else
            {
                writer = response.getWriter();
                template.renderTo(writer);
            }

        }
        catch (IOException e)
        {
            handleClientError(e);
        }
        catch (BeetlException e)
        {
            handleBeetlException(e);
        }

        finally
        {
            try
            {
                if (writer != null)
                    writer.flush();
                if (os != null)
                {
                    os.flush();
                }
            }
            catch (IOException e)
            {
                handleClientError(e);
            }
        }
    }
```

```
    /**
     * 可以添加更多的绑定
     * @param template 模板
     * @param key 模板的资源id
     * @param request
     * @param response
     * @param args  调用render的时候传的参数
     */
    protected void modifyTemplate(Template template, String key, HttpServletRequest request,
            HttpServletResponse response, Object... args)
    {

    }

    /**处理客户端抛出的IO异常
     * @param ex
     */
    protected void handleClientError(IOException ex)
    {
        //do nothing
    }

    /**处理客户端抛出的IO异常
     * @param ex
     */
    protected void handleBeetlException(BeetlException ex)
    {
        throw ex;
    }
}
```

Serlvet集成

只需要在Servlet代码里引用ServletGroupTemplate就能集成Beetl，他提供了一个render(String child, HttpServletRequest request, HttpServletResponse response)方法。例子如下：

```
protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {

        response.setContentType("text/html;charset=UTF-8");
        //request.setAttribute ....
        ServletGroupTemplate.instance().render("/index.html", request, response);


    }
```

ServletGroupTemplate同其他web集成一样，将读取配置文件来配置，如果需要通过代码配置，可以在Serlvet listener里 ServletGroupTemplate.instance().getGroupTemplate()方法获取GroupTemplate

SpringMVC集成

需要做如下配置即可

```
<bean id="beetlConfig" class="org.beetl.ext.spring.BeetlGroupUtilConfiguration" init-method="init"/>
```

```
<bean id="viewResolver" class="org.beetl.ext.spring.BeetlSpringViewResolver">
    <property name="contentType" value="text/html;charset=UTF-8">
</bean>
```

同其他集成方式一样，模板的配置将放在beetl.properties中。

如果想获取GroupTemplate,可以调用如下代码

```
BeetlGroupUtilConfiguration config = (BeetlGroupUtilConfiguration) this.getApplicationContext().getBean(
                    "beetlConfig");
GroupTemplate group = config.getGroupTemplate();
```

Jodd集成

只需要在Action里返回"beetl:…..."就可以了，冒号后面是模板路径，请以/开头，如下例子：

```
@MadvocAction
public class IndexAction {

   @Out
   String value;

   @Action("/index.html")
   public String world() {

      value = "Hello World!";
      return "beetl:/ok.html";
   }
}
```

这样，将访问ROOT/ok.html的模板。如果你想配置GroupTemplate，正如其他集成框架一样，只需要写一个beetl.properties 即可。

如下是集成的类实现，你也可以定制自己的集成方式

```
package org.beetl.ext.jodd;

import java.io.IOException;

import jodd.madvoc.ActionRequest;
import jodd.madvoc.result.ActionResult;

import org.beetl.core.Configuration;
import org.beetl.core.GroupTemplate;
import org.beetl.core.resource.WebAppResourceLoader;
import org.beetl.ext.web.WebRender;

public class BeetlActionResult implements ActionResult
{

    public static final String NAME = "beetl";
    GroupTemplate groupTemplate = null;
```

```
    public BeetlActionResult()
    {

        try
        {

            Configuration cfg = Configuration.defaultConfiguration();
            WebAppResourceLoader resourceLoader = new WebAppResourceLoader();
            groupTemplate = new GroupTemplate(resourceLoader, cfg);

        }
        catch (IOException e)
        {
            throw new RuntimeException("加载GroupTemplate失败", e);
        }

    }

    @Override
    public void render(ActionRequest actionRequest, Object resultValue) throws Exception
    {

        WebRender render = new WebRender(groupTemplate);
        render.render((String) resultValue, actionRequest.getHttpServletRequest(),
                actionRequest.getHttpServletResponse());

    }

    @Override
    public String getResultType()
    {
        return NAME;
    }

    @Override
    public Class getResultValueType()
    {

        return String.class;
    }

    @Override
    public void init()
    {

    }

}
```

JFinal集成

Beetl提供 JFinal 集成，使用BeetlRenderFactory ，通过如下注册即可使用beetl模板引擎

```
import org.beetl.ext.jfinal.BeetlRenderFactory
public class DemoConfig extends JFinalConfig
```

```
{
    public void configConstant(Constants me)
    {

        me.setMainRenderFactory(new BeetlRenderFactory());
        // 获取GroupTemplate ,可以设置共享变量等操作
        GroupTemplate groupTemplate = BeetlRenderFactory.groupTemplate ;

}
```

BeetlRenderFactory默认使用FileResourceLoader，其根目录位于WebRoot目录下，如果你需要修改到别的目录，可以设置配置文件，如

```
RESOURCE.root= /WEB-INF/template/
```

如下是 BeetlRenderFactory 源码

```
package org.beetl.ext.jfinal;

import java.io.File;
import java.io.IOException;

import org.beetl.core.Configuration;
import org.beetl.core.GroupTemplate;
import org.beetl.core.resource.FileResourceLoader;

import com.jfinal.render.IMainRenderFactory;
import com.jfinal.render.Render;

public class BeetlRenderFactory implements IMainRenderFactory
{

    public static String viewExtension = ".html";
    public static GroupTemplate groupTemplate = null;

    static
    {
        try
        {

            Configuration cfg = Configuration.defaultConfiguration();
            WebAppResourceLoader resourceLoader = new WebAppResourceLoader();
            groupTemplate = new GroupTemplate(resourceLoader, cfg);

        }
        catch (IOException e)
        {
            throw new RuntimeException("加载GroupTemplate失败", e);
        }
    }

    public Render getRender(String view)
    {
        return new BeetlRender(groupTemplate, view);
    }
```

```
    public String getViewExtension()
    {
        return viewExtension;
    }

}
```

BeetlRender 继承了Jfinal Render类，内容如下：

```
package org.beetl.ext.jfinal;

import org.beetl.core.GroupTemplate;
import org.beetl.core.exception.BeetlException;
import org.beetl.ext.web.WebRender;

import com.jfinal.render.Render;
import com.jfinal.render.RenderException;

public class BeetlRender extends Render
{
    GroupTemplate gt = null;
    private transient static final String encoding = getEncoding();
    private transient static final String contentType = "text/html; charset=" + encoding;

    public BeetlRender(GroupTemplate gt, String view)
    {
        this.gt = gt;
        this.view = view;
    }

    @Override
    public void render()
    {

        try

        {
            response.setContentType(contentType);
            WebRender webRender = new WebRender(gt);
            webRender.render(view, request, response);

        }
        catch (BeetlException e)
        {
            throw new RenderException(e);
        }

    }

}
```

Nuts集成

Nutz集成提供了 BeetlViewMaker ，实现了 ViewMaker方法，如下代码

```
public class HelloBeetl {
	@At("/hello")
	@Ok("beetl:hello.html")
	public String doHello() {
		// do something ....
		return "";
	}
}
```

代码第三行使用前缀beetl，这表示使用BeetlViewMaker.如下椒BeetlViewMaker实现

```
package org.beetl.ext.nutz;

import java.io.IOException;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.beetl.core.Configuration;
import org.beetl.core.GroupTemplate;
import org.beetl.core.resource.WebAppResourceLoader;
import org.beetl.ext.web.WebRender;
import org.nutz.ioc.Ioc;
import org.nutz.mvc.View;
import org.nutz.mvc.ViewMaker;
import org.nutz.mvc.view.AbstractPathView;

/**
 * Beelt for Nutz
 *
 * @author wendal,joelli
 *
 */
public class BeetlViewMaker implements ViewMaker
{

	public GroupTemplate groupTemplate;
	private boolean inited;

	public void init()
	{
		if (inited)
			return;

		Configuration cfg;
		try
		{
			cfg = Configuration.defaultConfiguration();
			WebAppResourceLoader resourceLoader = new WebAppResourceLoader();
			groupTemplate = new GroupTemplate(resourceLoader, cfg);
			inited = true;
		}
		catch (IOException e)
		{
			throw new RuntimeException("加载GroupTemplate失败", e);
```

```
        }

    }

    public void depose()
    {
        if (groupTemplate != null)
            groupTemplate.close();
    }

    public View make(Ioc ioc, String type, String value)
    {
        if (!inited)
            init();
        if ("beetl".equals(type))
            return new AbstractPathView(value) {
                @SuppressWarnings("unchecked")
                public void render(HttpServletRequest req, HttpServletResponse resp, Object obj) throws Throwable
                {

                    String child = evalPath(req, obj);
                    WebRender render = new WebRender(groupTemplate);
                    render.render(child, req, resp);

                }
            };
        return null;
    }
}
```

Struts2集成

需要在struts2配置文件里添加result-types做如下配置

```
<package name="default" namespace="/" extends="struts-default">
.......
<result-types>
    <result-type name="beetl"
        class="org.beetl.ext.struts2.Struts2BeetlActionResult" default="true" />
</result-types>

<action name="HelloWorld" class="example.HelloWorld">
  <result>/hello.html</result>
</action>
........
</package>
```

该类会根据struts配置文件获取模板，如上例的hello.html,并将formbean的属性，以及request属性作为全局变量传递给模板 Struts2BeetlActionResult 集成代码如下：

```
public class Struts2BeetlActionResult extends StrutsResultSupport
{


    ReflectionProvider reflectionProvider = null;
```

```
public static GroupTemplate groupTemplate;
static {
    Configuration cfg;
    try
    {
        cfg = Configuration.defaultConfiguration();
        WebAppResourceLoader resourceLoader = new WebAppResourceLoader();
        groupTemplate = new GroupTemplate(resourceLoader, cfg);

    }
    catch (IOException e)
    {
        throw new RuntimeException("加载GroupTemplate失败", e);
    }
}

@Inject
public void setReflectionProvider(ReflectionProvider prov)
{
    this.reflectionProvider = prov;
}

protected void doExecute(String locationArg, ActionInvocation invocation) throws Exception
{

    ActionContext ctx = invocation.getInvocationContext();

    HttpServletRequest req = (HttpServletRequest) ctx.get(ServletActionContext.HTTP_REQUEST);
    HttpServletResponse rsp = (HttpServletResponse) ctx.get(ServletActionContext.HTTP_RESPONSE);

    if (!locationArg.startsWith("/"))
    {
        String base = ResourceUtil.getResourceBase(req);
        locationArg = base + "/" + locationArg;
    }

    Template template = groupTemplate.getTemplate(locationArg);

    Object action =     invocation.getAction();
    Map<String, Object> values = reflectionProvider.getBeanMap(action);

    WebRender render = new WebRender(groupTemplate){
        protected void modifyTemplate(Template template, String key, HttpServletRequest request,
                HttpServletResponse response, Object... args)
        {
            Map model = (Map) args[0];

            for (Object o : model.entrySet())
            {
                Entry entry = (Entry) o;
                String name = (String) entry.getKey();
                Object value = entry.getValue();
                template.binding(name, value);
            }
        }
```

```
        };
        render.render(locationArg, req, rsp,values);


    }


}
```

郑重申明

鉴于struts2有安全漏洞，而官方补丁打法很消极，所以请谨慎使用Struts2，Beetl的安全性已经通过在线体验，多个使用网站得以体现 一旦你的struts2网站被攻破，请先确定是否是struts2 的问题

5. 附录

内置方法

常用内置方法

- date 返回一个java.util.Date类型的变量，如 date() 返回一个当前时间; ${date(*2011-1-1,yyyy-MM-dd*)} 返回指定日期
- print 打印一个对象 print(user.name);
- println 打印一个对象以及回车换行符号，回车换号符号使用的是模板本身的，而不是本地系统的. 如果仅仅打印一个换行符，则直接调用println() 即可
- nvl 函数nvl，如果对象为null，则返回第二个参数，否则，返回自己 nvl(user,"不存在")
- isEmpty 判断变量或者表达式是否为空，变量不存在，变量为null，变量是空字符串，变量是空集合，变量是空数组，此函数都将返回true
- has 变量名为参数，判断是否存在此全局变量，如 has(userList),类似于1.x版本的exist("userList"),但不需要输入引号了
- assert 如果表达式为false，则抛出异常
- trunc 截取数字，保留指定的小数位，如trunc(12.456,2) 输出是12.45
- decode 一个简化的if else 结构，如 decode(a,1,"a=1",2,"a=2","不知道了")},如果a是1，这decode输出"a=1"，如果a是2，则输出"a==2"，如果是其他值，则输出"不知道了"
- debug 在控制台输出debug指定的对象以及所在模板文件以及模板中的行数，如debug(1),则输出1 [在3行@/org/beetl/core/lab/hello.txt]
- parseInt 将数字或者字符解析为整形 如 parseInt("123");
- parseDouble 将数字或者字符解析为浮点类型 如parseDouble("1.23")
- range 接收三个参数，初始值，结束值，还有步增（可以不需要，则默认为1），返回一个Iterator，常用于循环中，如for(var i in range(1,5)) {print(i)},将依次打印1234.

字符串相关方法

- strutil.startWith ${ strutil.startWith(“hello”,”he”) 输出是true
- strutil.endWith ${ strutil.endWith(“hello”,”o”) 输出是true
- strutil.length ${ strutil. length (“hello”),输出是5

- strutil.subString ${ strutil.subString (“hello”,1),输出是“ello”
- strutil.subStringTo ${ strutil.subStringTo (“hello”,1,2),输出是“el”
- strutil.split ${ strutil.split (“hello,joeli”,”,”),输出是数组，有俩个元素，第一个是hello，第二个是joelli”
- strutil.contain ${ strutil.contain (“hello,”el”),输出是true
- strutil.toUpperCase ${ strutil.toUpperCase (“hello”),输出是HELLO
- strutil.toLowerCase ${ strutil.toLowerCase (“Hello”),输出是hello
- strutil.replace ${ strutil.replace (“Hello”,”lo”,”loooo”),输出是helloooo
- 具体请参考http://docs.oracle.com/javase/6/docs/api/java/text/MessageFormat.html
- strutil.trim 去掉字符串的尾部空格
- strutil.formatDate Var a = strutil.formatDate(user.bir,’yyyy-MM-dd’);

数组相关方法

- array.range 返回数组或者Collection一部分，接受三个参数，第一个是数组或者Collection子类，第二，三个参数分别是起始位置
- array.remove 删除某个数组或者Collection的一个元素，并返回该数组或者Collection.第一个是数组或者Collection子类, 第二个参数是元素
- array.add 向数组或者Collection添加一个元素，并返回该数组或者Collection。第一个是数组或者Collection子类, 第二个参数是元素
- array.contain 判断数组或者元素是否包含元素，如果包含，返回true。否则false。第一个是数组或者Collection子类, 第二个参数是元素

内置格式化方法

- dateFormat 日期格式化函数，如 $*yyyy-Mm-dd*，等于符号后的参数也可以没有，则使用本地默认来做格式化如${date,dateFormat}
- numberFormat ${0.345,numberFormat="#.%"} 输出是 34.5%,具体请参考文档 http://docs.oracle.com/javase/6/docs/api/java/text/DecimalFormat.html

内置标签函数

- include include一个模板，如：

```
<%include("/header.html"){}%>
```

如果想往子模板中传入参数，则可以后面跟一个json变量

```
<%includeFileTemplate("/header.html",{'user',user:'id',user.id}){}%>
```

这样user，和id 可以在header.html被引用，并成为header.html的全局变量

> (beetl1.2 也叫includeFileTemplate，2.0仍然支持，但不再文档里体现了)

- layout 提供一个布局功能，每个页面总是由一定布局，如页面头，菜单，页面脚，以及正文。 layout标签允许为正文指定一个布局，如下使用方式

content.html内容如下：

```
<%
//content.html内容如下：
layout("/inc/layout.html"){%>
this is 正文
.....
<%%}%>
```

layout.html 是布局文件，内容如下•

```
<%
<%include("/inc/header.html"){} %>
this is content:${layoutContent}
this is footer:

<%%}%>
```

运行content.html模板文件后，，正文文件的内容将被替换到layoutContent的地方，变成如下内容

```
this is header
this is content:this is 正文
.....
this is footer:
```

如果想往layout页面传入参数，则传入一个json变量，如下往layout.html页面传入一个用户登录时间

```
<%
 layout("/inc/header.html",{'date':user.loginDate,'title':"内容页面"}){%>
this is 正文
.....

<%%}%>
```

如果layoutContent 命名有冲突，可以在layout第三个参数指定，如

```
<%

layout("/inc/header.html",{'date':user.loginDate,'title':"内容页面"},"myLayoutContent"){%>
this is 正文
.....

<%}%>
```

- cache 能Cache标签的内容，并指定多长时间刷新，如

```
<%:cache('key2',10,false){ %>
内容体
<%}%>
```

需要指定三个参数，第一个是cache的Key值，第二个是缓存存在的时间，秒为单位，第三个表示是否强制刷新•，false表示不，true表示强制刷新 Cache默认实现org.beetl.ext.tag.cache.SimpleCacheManager. 你可以设置你自己的Cache实现，通过调用CacheTag. cacheManager= new YourCacheImplementation();

可以在程序里调用如下方法手工删除Cache：

```
public void clearAll();

public void clearAll(String key);

public void clearAll(String... keys);
```

- includeJSP,可以在模板里包括一个jsp文件，如：

```
<%
includeJSP("/xxxx.jsp",{"key":"value"})
%>
```

key value 都是字符串，将以paramenter的形式提供给jsp，因此jsp可以通过request.getParameter("key")来获取参数

主要注意的是，这个标签并非内置，需要手工注册一下

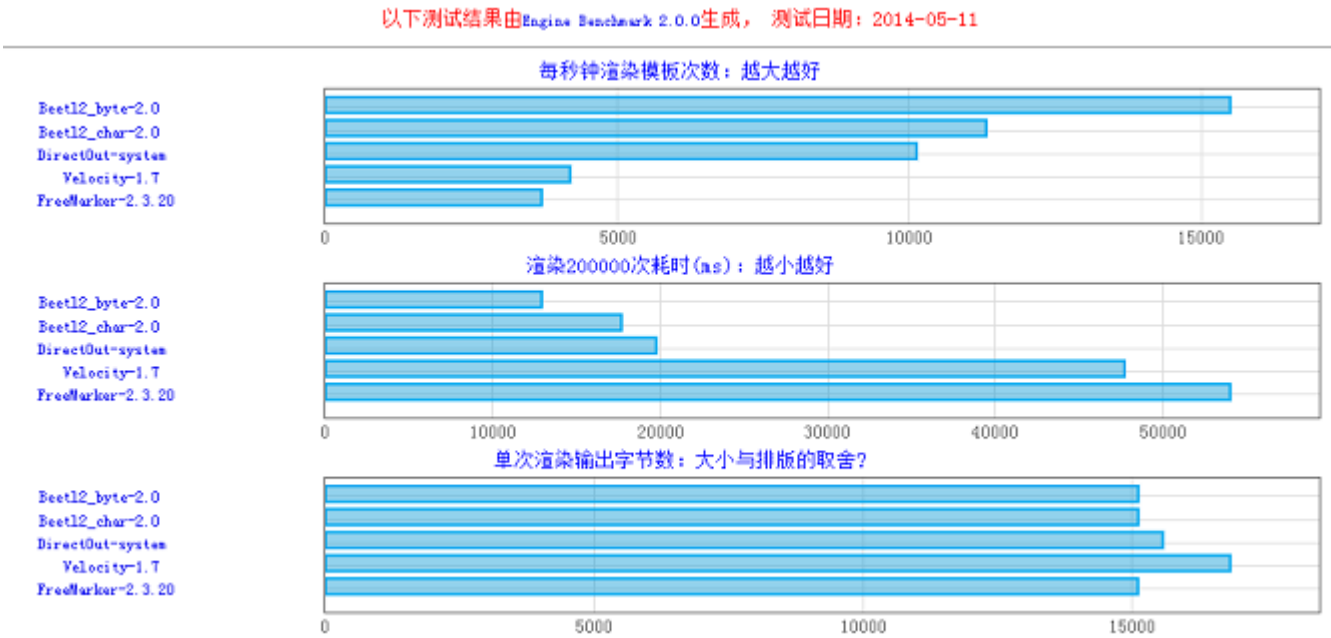
```
groupTemplate.registerTag("incdlueJSP",org.beetl.ext.jsp.IncludeJSPTag.class);
```

性能优化的秘密

Beetl2.0目前只完成了解释引擎，使用解释引擎好处是可以适用于各种场景，性能测试表明，Beetl2.0引擎是Freemaker的4倍，跟最好的 的编译引擎性能相比，也相差只有几个百分点。为什么Beetl能跑的如此之快呢，简单的说，有如下策略

- 优化IO输出，允许使用字节直接输出，模板中的静态文本事先转化为字节
- encode优化，对于number类型，输出通常是.toString 转化成String，然后encode输出，这中间浪费了大量的资源，Beetl实现了encode，输出一步到位
- Context 采用一维数组，语言里的Context通常采用Map实现，每次进入{} ，就新增一个child Map，尽管map很快，但不够快。也有其他模板语言采用二位数组提高性能，Beetl是通过固定大小一维数组来维护模板的Context，因此访问更快，也避免了Map和二维素组的频繁创建。其实除了此处，beetl很多地方都不采用Map来维护key-value,而都采用数组索引，以追求性能极限
- 字节码访问属性，通过反射获取性能比较慢，就算JVM有优化，但优化效果也不确定。Beetl通过字节码生成了属性访问类，从而将属性访问速度提高了一个数量级
- 类型推测：Beetl 是强制类型的，因此预先知道类型，可以对模板做一些优化而省去了动态判断类型的时间
- 使用数组Buffer，避免频繁创建和销毁数组
- 编译引擎将模板编译成类，会产生大量的类，虚拟机很难对这些做优化。而解释引擎只有几十个固定的类，虚拟机容易优化

性能测试对比



测试用例 https://github.com/javamonkey/ebm