**Beetl源码分析**

**八两俊**

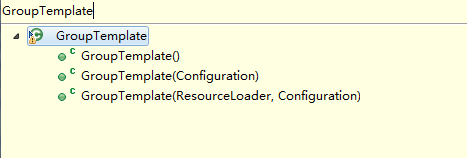
**2014.12**

Beetl是由[闲.大赋](http://my.oschina.net/u/567839)开发并开源的一个新一代的模板引。Beetl,是Bee Template Language的缩写，它绝不是简单的另外一种模板引擎，而是新一代的模板引擎，它功能强大，性能良好，秒杀当前流行的模板引擎。而且还易学易用。

下面我们对Beetl的源码进行剖析，以达到了解Beetl实现原理的目的。

## GroupTemplate

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。



GroupTemplate构造器

通常使用Beetl要实例化一个ResourceLoader和Configuration，然后通过这两个参数实例化GroupTemplate：

ClasspathResourceLoader loader = **new** ClasspathResourceLoader();

Configuration configuration = Configuration.*defaultConfiguration*();

GroupTemplate gt = **new** GroupTemplate(loader, configuration);

Template template = gt.getTemplate("/com/example/beetl/Page.btl");

template.binding("current", "1");

System.***out***.println(template.render());

ResourceLoader详细见[ResourceLoader](#_ResourceLoader)。

Configuration详细见[Configuration](#_Configuration)。

### GroupTemplate(ResourceLoader,Configuration)

**public** GroupTemplate(ResourceLoader loader, Configuration conf) {

**try** {

**this**.resourceLoader = loader;

**this**.conf = conf;

init();

initResourceLoader();

} **catch** (Exception ex) {

**throw** **new** RuntimeException("初始化失败", ex);

}

}

GroupTemplate实例化过程中主要分为两部分，一部分是初始化Configuration中的配置信息，另一部分就是初始化ResourceLoader的初始化。详细见[GroupTemplate.init()](#_GroupTemplate.init())和 [GroupTemplate.initResourceLoader()](#_GroupTemplate.initResourceLoader())。

### GroupTemplate.init()

protected void init() {

engine = TemplateEngineFactory.*getEngine*(conf.getEngine());

this.initFunction();

this.initFormatter();

this.initTag();

this.initVirtual();

classSearch = new ClassSearch(conf.getPkgList());

nativeSecurity = (NativeSecurityManager) ObjectUtil.*instance*(conf.getNativeSecurity());

if (conf.errorHandlerClass == null) {

errorHandler = null;

} else {

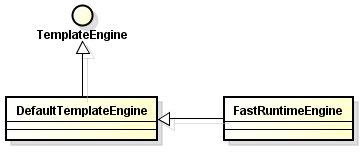
errorHandler = (ErrorHandler) ObjectUtil.*instance*(conf.errorHandlerClass);

}

}

这一步主要是主要是根据Configuration中的配置初始化TemplateEngine、NativeSecurityManager、ErrorHandler、ClassSearch和注册Function、Formatter、VirtualClassAttribute

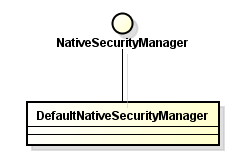
TemplateEngine：Beetl模版引擎，在默认配置文件beetl-default.properties中通过ENGINE指定，可以通过beetl.properties用自定义的TemplateEngine覆盖默认配置。



TemplateEngine类图

ClassSearch：在classloader下仅仅根据类名加载类，会尝试用系统内置的以及配置好的包名作为类的包名。beetl默认将java.util和java.lang包下的类加入搜索列表，同时beetl也支持通过beetl.properties自定义配置，对应的KEY为IMPORT\_PACKAGE，存在多个可以用分号[;]隔开。

NativeSecurityManager：判断是否允许调用本地方法，beetl默认提供DefaultNativeSecurityManager，同时也支持通过beetl.properties自定义配置，对应的KEY为NATIVE\_SECUARTY\_MANAGER。



NativeSecurityManager类图

ErrorHandler：错误处理类,beetl默认的错误处理类为ConsoleErrorHandler。beetl同时也支持通过beetl.properties自定义配置，对应的KEY为ERROR\_HANDLER。

initFunction：为Beetl注册自定义方法Function

详细见[GroupTemplate.initFunction](#_GroupTemplate.initFunction)

initFormatter：为Beetl注册自定义格式化函数Formatter

详细见[GroupTemplate.initFormatter](#_GroupTemplate.initFormatter)

initTag：为Beetl注册自定义标签Tag，

详细见[GroupTemplate.initTag](#_GroupTemplate.initTag)

initVirtual：为Beetl注册自定义虚拟VirtualClassAttribute

详细见[GroupTemplate.initVirtual](#_GroupTemplate.initVirtual)

#### GroupTemplate.initFunction

**protected** **void** initFunction() {

Map<String, String> fnMap = **this**.conf.fnMap;

**for** (Entry<String, String> entry : fnMap.entrySet()) {

String name = entry.getKey();

String clsName = entry.getValue();

**this**.registerFunction(name, (Function) ObjectUtil.*instance*(clsName));

}

Map<String, String> fnpMap = **this**.conf.fnPkgMap;

**for** (Entry<String, String> entry : fnpMap.entrySet()) {

String name = entry.getKey();

String clsName = entry.getValue();

**this**.registerFunctionPackage(name, ObjectUtil.*getClassByName*(clsName), ObjectUtil.*tryInstance*(clsName));

}

}

Beetl主持三种类型的自定义方法，如下：

1、实现Function接口，在beetl.properties中申明定义，KEY需要以”FN.”或”fn.”开头，如：

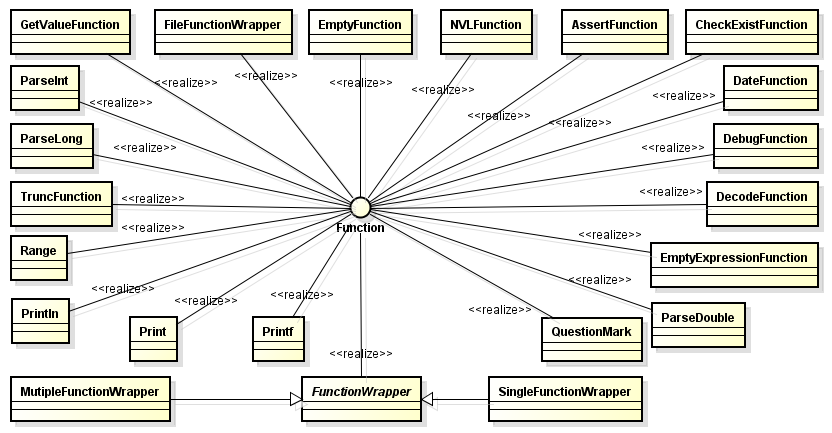
FN.has = org.beetl.ext.fn.CheckExistFunction

2、使用普通的Java类，在beetl.properties中声明，KEY需要以”FNP.”或”fnp.”开头，如：

FNP.array = org.beetl.ext.fn.ArrayUtil

3、使用模版文件作为方法。

从下面的Function类图中可以看出beetl所有的自定义函数都需要实现Function接口，在上面的”第2点”中，将fnp对应的类包装成了FunctionWrapper类型的对象；上面的”第3点”模版文件会被包装成FileFunctionWrapper类型的对象。



Function类图

#### GroupTemplate.initFormatter

**protected** **void** initFormatter() {

Map<String, String> formatMap = **this**.conf.formatMap;

**for** (Entry<String, String> entry : formatMap.entrySet()) {

String name = entry.getKey();

String clsName = entry.getValue();

**this**.registerFormat(name, (Format) ObjectUtil.*instance*(clsName));

}

Map<String, String> defaultFormatMap = **this**.conf.defaultFormatMap;

Map<String, Format> temp = **new** HashMap<String, Format>();

**for** (Entry<String, String> entry : defaultFormatMap.entrySet()) {

String defaultType = entry.getKey();

String formatClass = entry.getValue();

Format format = temp.get(formatClass);

**if** (format == **null**) {

format = (Format) ObjectUtil.*instance*(formatClass);

temp.put(formatClass, format);

}

**this**.registerDefaultFormat(ObjectUtil.*getClassByName*(defaultType), format);

}

//原始类型无法通过反射获取，因此不再配置文件里

Format numberFormat = temp.get("org.beetl.ext.format.NumberFormat");

**this**.registerDefaultFormat(**short**.**class**, numberFormat);

**this**.registerDefaultFormat(**long**.**class**, numberFormat);

**this**.registerDefaultFormat(**int**.**class**, numberFormat);

**this**.registerDefaultFormat(**float**.**class**, numberFormat);

**this**.registerDefaultFormat(**double**.**class**, numberFormat);

}

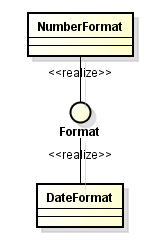
beetl支持自定义两种类型的Formatter，一种是定义formatter方法，另一种是为指定类注册默认的formatter。

方式一：自定义formatter方法，可以在beetl.properties配置文件下声明，KEY需要以”FT.”或”ft.”开头，如：

FT.numberFormat = org.beetl.ext.format.NumberFormat

方式二：为指定类注册默认的formatter，可以在beetl.properties配置文件下声明，KEY为需要以”FTC.”或”ftc.”开头，如：

FTC.java.util.concurrent.atomic.AtomicLong = org.beetl.ext.format.NumberFormat



Format类图

#### GroupTemplate.initTag

protected void initTag() {

Map<String, String> tagMap = this.conf.tagMap;

for (Entry<String, String> entry : tagMap.entrySet()) {

String name = entry.getKey();

String clsName = entry.getValue();

this.registerTag(name, ObjectUtil.*getClassByName*(clsName));

}

Map<String, String> tagFactoryMap = this.conf.tagFactoryMap;

for (Entry<String, String> entry : tagFactoryMap.entrySet()) {

String name = entry.getKey();

String clsName = entry.getValue();

this.registerTagFactory(name, (TagFactory) ObjectUtil.*instance*(clsName));

}

}

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。

beetl提供两种形式来获取自定义的建标签函数,这两种方式都需要在beetl.properties配置文件中声明

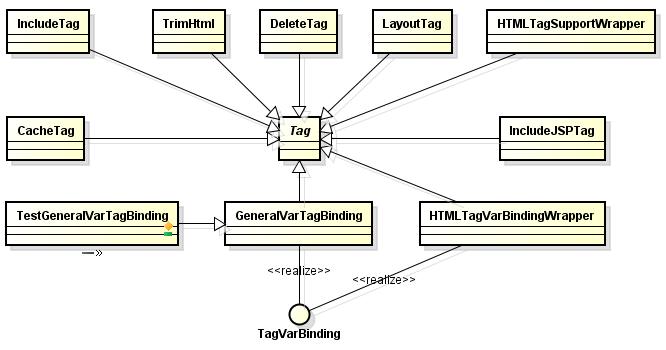
1、KEY以”TAG.”或”tag.”开头，value需要继承Tag抽象类。

TAG.includeFileTemplate= org.beetl.ext.tag.IncludeTag

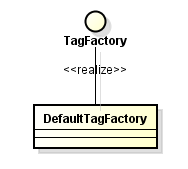
2、KEY以”TAGF.”或”tagf.”开头，value需要实现TagFactory接口。

TAGF.includeFileTemplateFactory = org.beetl.ext.tag.IncludeTagFactory

两种形式，最后都会被包装成TagFactory，存放在GroupTemplate的tagFactoryMap全局变量中。



Tag类图



TagFactory类图

#### GroupTemplate.initVirtual

**protected** **void** initVirtual() {

// 可以根据类型做一定优化

**this**.registerVirtualAttributeEval(**new** VirtualAttributeEval() {

**public** Integer eval(Object o, String attributeName, Context ctx) {

**if** (attributeName.equals("size")) {

**if** (o **instanceof** Collection) {

**return** ((Collection) o).size();

} **else** **if** (o **instanceof** Map) {

**return** ((Map) o).size();

} **else** **if** (o.getClass().isArray()) {

**return** ((Object[]) o).length;

} **else** {

**throw** **new** IllegalArgumentException();

}

} **else** {

**throw** **new** IllegalArgumentException();

}

}

**public** **boolean** isSupport(Class c, String attributeName) {

**if** ((Map.**class**.isAssignableFrom(c) || Collection.**class**.isAssignableFrom(c) || c.isArray()) && attributeName.equals("size")) {

**return** **true**;

} **else** {

**return** **false**;

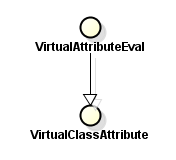
}

}

});

}

如果需要自定义属性，可以调用GroupTemplate的registerVirtualAttributeEval方法注册，所有自定义的属性都需要实现VirtualAttributeEval接口，类图如下：



VirtualAttributeEval类图

### GroupTemplate.initResourceLoader()

**protected** **void** initResourceLoader() {

**if** (**this**.resourceLoader == **null**) {

**this**.resourceLoader = (ResourceLoader) ObjectUtil.*instance*(conf.resourceLoader);

}

resourceLoader.init(**this**);

}

beetl默认提供的ResourceLoader为ClasspathResourceLoader，各应用提供在beetl.properties中声明自定义的ResourceLoader，对应的KEY为”RESOURCE\_LOADER”。

在initResourceLoader中会调用ResourceLoader的init(GroupTemplate)方法初始化资源加载器的相关信息，详细见[ResourceLoader](#_ResourceLoader)。

### GroupTemplate.getTemplate(String)

**public** Template getTemplate(String key) {

key = key.intern();

Program program = (Program) **this**.programCache.get(key);

**if** (program == **null**) {

**synchronized** (key) {

**if** (program == **null**) {

Resource resource = resourceLoader.getResource(key);

program = **this**.loadTemplate(resource);

**this**.programCache.set(key, program);

**return** **new** Template(**this**, program, **this**.conf);

}

}

}

**if** (resourceLoader.isModified(program.rs)) {

**synchronized** (key) {

Resource resource = resourceLoader.getResource(key);

program = **this**.loadTemplate(resource);

**this**.programCache.set(key, program);

}

}

**return** **new** Template(**this**, program, **this**.conf);

}

1、programCache是GroupTemplate中的一个全局变量，其对应的类型为Cache，Beetl默认采用LocalCache，LocalCache的实现原理是采用一个ConcurrentHashMap来完成**模版KEY域**与**脚本运行程序(Program)**之间的关系缓存。

2、如果没有在programCache中无法获得缓存数据，则通过ResourceLoader的getResource获取到模版资源Resource信息，然后调用loadTemplate加载出脚本运行程序(Programe)信息，详细见[GroupTemplate.loadTemplate(Resource)](#_GroupTemplate.loadTemplate(Resource)。然后根据GroupTemplate、Program和Configuration创建Template(详细见[Template](#_Template))。

3、如果在programCache中获取到了缓存数据，则调用ResourceLoader的isModified(Resource)方法判断模版文件资源是否修改，如果修改则从新加载模版。然后根据GroupTemplate、Program和Configuration创建Template(详细见[Template](#_Template))。

上述2、3步都需要将生成的Programe数据加入到programCache缓存中。

### GroupTemplate.loadTemplate(Resource)

**private** Program loadTemplate(Resource res) {

Transformator sf = **null**;

**try** {

Reader reader = res.openReader();

sf = **new** Transformator(conf.placeholderStart, conf.placeholderEnd, conf.statementStart, conf.statementEnd);

**if** (**this**.conf.isHtmlTagSupport()) {

sf.enableHtmlTagSupport(conf.getHtmlTagStart(), conf.getHtmlTagEnd());

}

Reader scriptReader;

scriptReader = sf.transform(reader);

Program program = engine.createProgram(res, scriptReader, sf.textMap, sf.lineSeparator, **this**);

**return** program;

} **catch** (HTMLTagParserException e) {

ErrorGrammarProgram ep = **new** ErrorGrammarProgram(res, **this**, sf.lineSeparator);

ep.setException(e);

e.pushResource(res.id);

**return** ep;

} **catch** (IOException e) {

ErrorGrammarProgram ep = **new** ErrorGrammarProgram(res, **this**, sf.lineSeparator);

BeetlException ex = **new** BeetlException(BeetlException.***TEMPLATE\_LOAD\_ERROR***);

ex.pushResource(res.id);

ep.setException(ex);

**return** ep;

} **catch** (BeetlException ex) {

ErrorGrammarProgram ep = **new** ErrorGrammarProgram(res, **this**, sf != **null** ? sf.lineSeparator : **null**);

ex.pushResource(res.id);

ep.setException(ex);

**return** ep;

}

}

此方法的目的在于根据模版文件资源(Resource)，创建脚本运行程序(Program)。上述逻辑处理步骤为：

1、Reader reader = res.openReader();

调用Resource的openReader()获取Reader读入流对象，目前beetl默认提供三种Resource，分别是ClasspathResource([openReader()](#_openReader()))、FileResource([openReader()](#_openReader()_1))和StringTemplateResource([openReader()](#_openReader()_2))。

2、根据占位符和定界符创建Transformator对象，此对象将模版转化为beetl script的代码，此为核心代码之一。目前beetl默认支持的占位符和定界符分别为${}和<%%> ,可以从beetl-default.properties中找到此声明，如下：

DELIMITER\_PLACEHOLDER\_START=${

DELIMITER\_PLACEHOLDER\_END=}

DELIMITER\_STATEMENT\_START=<%

DELIMITER\_STATEMENT\_END=%>

当然，beetl也支持各应用自定义占位符和定界符，只需要在beetl.properties文件中覆盖响应的key和value即可。

3、获取配置文件beetl-default.properties或beetl.properties中KEY为”HTML\_TAG\_SUPPORT”和”HTML\_TAG\_FLAG”声明数据，两者分别代表是否允许html标签、html标签符号(默认为”#”)。如果允许则将html标签开始符号”<#”和结束符号”</#”分别设置到Transformator中的htmlTagStart和htmlTagEnd字段，并将其isSupportHtmlTag字段设置为true。

4、调用Transformator的transform(Reader)读入流Reader转换为beetl所需的脚本读入流Reader。详细见[Transformator.transform(Reader)](#_Transformator.transform(Reader))方法

5、调用TemplateEngine的createProgram方法将脚本读入流Reader等信息转换为脚本运行程序(Program)，详细见[FastRuntimeEngine.createProgram(Resource, Reader,Map<Integer,String>,String,GroupTemplate)](#_FastRuntimeEngine.createProgram(Res)。

6、beetl在这一步将所有的异常都包装成ErrorGrammarProgram。

## Transformator

### Transformator.transform(Reader)

**public** Reader transform(Reader orginal) **throws** IOException, HTMLTagParserException {

StringBuilder temp = **new** StringBuilder();

**int** bufSzie = 1024;

cs = **new** **char**[bufSzie];

**int** len = -1;

**while** ((len = orginal.read(cs)) != -1) {

temp.append(cs, 0, len);

}

cs = temp.toString().toCharArray();

// 找到回车换行符号

findCR();

**if** (**this**.endStatement == **null**) {

**this**.endStatement = lineSeparator;

}

checkAppendCR();

parser();

**if** (**this**.isSupportHtmlTag && **this**.htmlTagStack.size() != 0) {

String tagName = (String) htmlTagStack.peek();

GrammarToken token = GrammarToken.*createToken*(tagName, **this**.totalLineCount + 1);

HTMLTagParserException ex = **new** HTMLTagParserException("解析html tag 标签出错,未找到匹配结束标签 " + tagName);

ex.pushToken(token);

ex.line = totalLineCount + 1;

**this**.clear();

**throw** ex;

}

orginal.close();

**return** **new** StringReader(sb.toString());

}

说实话，这段方法我也没怎么看懂，主要的逻辑是将入参Reader转换成char[]数组，然后逐个解析char[]数组中的字符，解析的内容主要有占位符、html 标签和转义字符等。最后将解析后的数据包装成StringReader对象返回。

## TemplateEngine

## FastRuntimeEngine.createProgram(Resource, Reader,Map<Integer,String>,String,GroupTemplate)

**public** Program createProgram(Resource rs, Reader reader, Map<Integer, String> textMap, String cr, GroupTemplate gt) {

Program program = **super**.createProgram(rs, reader, textMap, cr, gt);

Probe nextFilter = **new** BasicProgramOptProbe(program);

Probe filter = **new** TypeBindingProbe(program, nextFilter);

FilterProgram filterProgram = **new** FilterProgram(program, filter);

**return** filterProgram;

}

## Template

## ResourceLoader

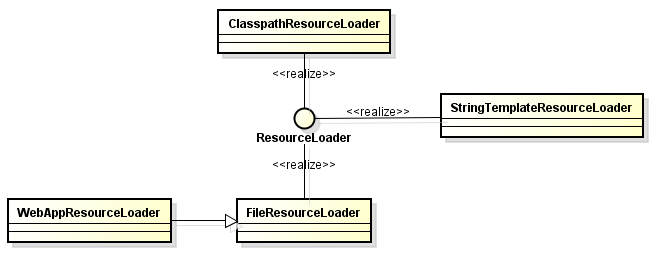
ResourceLoader主要用于加载Beetl的模版信息，我们可以从下面的类图中可以看出Beetl提供了ClasspathResourceLoader、StringTemplateResourceLoader、FileResourceLoader和WebAppResourceLoader的具体实现。

StringTemplateResourceLoader：字符串模板加载器，用于加载字符串模板。

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。



ResourceLoader类图

ResourceLoader定义了如下接口：

/\*\* 根据key获取Resource\*/

**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\*/

**public** **boolean** isModified(Resource key);

**public** **boolean** exist(String key);

/\*\* 关闭ResouceLoader，通常是GroupTemplate关闭的时候也关闭对应的ResourceLoader\*/

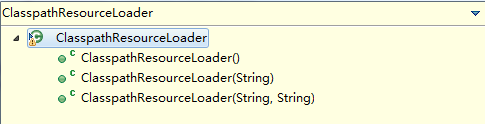
**public** **void** close();

/\*\* 一些初始化方法\*/

**public** **void** init(GroupTemplate gt);

### ClasspathResourceLoader

ClasspathResourceLoader提供了三种不同的构造器可用于对ClasspathResourceLoader实例化工作。不同的构造器的不同之处在于对某些全局变量的初始化，如root、charset和classloader变量。



ClasspathResourceLoader构造器

#### ClasspathResourceLoader.getResource()方法实现

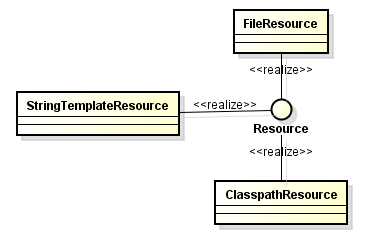
**public** Resource getResource(String key) {

Resource resource = **new** ClasspathResource(key, root + key, **this**);

**return** resource;

}

相当于创建对应的Resource，Beetl中已定义的Resource类图如下：



Resource类图

#### ClasspathResourceLoader.isModified()方法实现

**public** **boolean** isModified(Resource key) {

**if** (**this**.autoCheck) {

**return** key.isModified();

} **else** {

**return** **false**;

}

}

通过判断全局变量autoCheck[默认在beetl-default.properties配置，也可以通过beetl.properties覆盖此配置]判断是否需要自动检查Resource的变化，对于ClasspathResourceLoader来说，是通过调用ClasspathResource.isModified()方法判断ClassPathResouce是否变化，最终还是通过调用File.lastModified()来进行判断的。

#### ClasspathResourceLoader.init()方法实现

**public** **void** init(GroupTemplate gt) {

Map<String, String> resourceMap = gt.getConf().getResourceMap();

**if** (resourceMap.get("root") != **null**) {

String temp = resourceMap.get("root");

**if** (temp.equals("/") || temp.length() == 0) {

} **else** {

**if** (**this**.root.endsWith("/")) {

**this**.root = **this**.root + resourceMap.get("root");

} **else** {

**this**.root = **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"));

//初始化functions

URL url = classLoader.getResource("");

**this**.gt = gt;

**if** (url.getProtocol().equals("file")) {

File root = **new** File(url.getFile(), **this**.functionRoot);

**if** (root.exists()) {

String ns = "";

String path = "/".concat(**this**.functionRoot).concat("/");

readFuntionFile(root, ns, path);

}

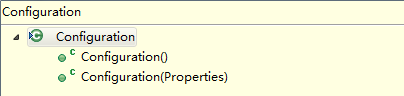
}

}

## Configuration

Configuration用于存储Beetl的配置信息，这些配置信息可来源于beetl-default.properties文件提供的beetl默认配置信息，各应用通过beetl.properties覆盖默认配置信息。

Configuration提供了两个构造器用于实例化，如下：



Configuration构造器

### Configuration.Configuration()

**public** Configuration() **throws** IOException {

// 为beetl增加java.util和java.lang包搜索路径

pkgList.add("java.util.");

pkgList.add("java.lang.");

ps = **new** Properties();

// 读取beetl提供的默认属性，详细见<beetl-default.properties>，稍后解析

ps.load(Configuration.**class**.getResourceAsStream("/org/beetl/core/beetl-default.properties"));

parseProperties(ps);

// 读取各应用可以复写beetl-default.properties中已定义的属性。

InputStream ins = Configuration.**class**.getResourceAsStream("/beetl.properties");

**if** (ins != **null**) {

ps.clear();

ps.load(ins);

parseProperties(ps);

}

}

### Configuration.Configuration(Properties)

**public** Configuration(Properties ps) **throws** IOException{

**this**();// 详细见[Configuration.Configuration()](#_Configuration.Configuration())

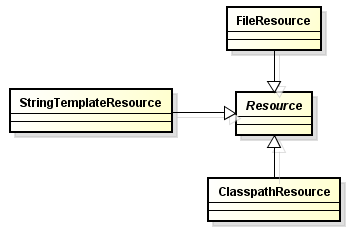
parseProperties(ps);

}

除了默认的读取默认的配置文件外，还可以动态的通过Properties来覆盖配置文件中的属性信息。

## Resource

Resource代表Beetl的模版文件资源，Beetl提供了FileResource、ClasspathResource和StringTemplateResource三种Resource的实现，类图如下：



Resource类图

### FileResource

对应的ResourceLoader为FileResourceLoader

#### openReader()

**public** Reader openReader() {

Reader br;

**try** {

br = **new** BufferedReader(**new** InputStreamReader(**new** FileInputStream(file), ((FileResourceLoader) **this**.getResourceLoader()).charset));

} **catch** (UnsupportedEncodingException e) {

**return** **null**;

} **catch** (FileNotFoundException e) {

FileResourceLoader loader = (FileResourceLoader) **this**.resourceLoader;

BeetlException be = **new** BeetlException(BeetlException.***TEMPLATE\_LOAD\_ERROR***, " 模板根目录为 " + loader.getRoot());

be.resourceId = **this**.id;

**throw** be;

}

**return** br;

}

### ClasspathResource

对应的ResourceLoader为ClasspathResourceLoader。

#### openReader()

### StringTemplateResource

对应的ResourceLoader为StringTemplateResourceLoader。

#### openReader()