# {{title}}

{{author}} - {{email}} - {{version}}

{{abstract}}

“{{target}}

## 1. License

{{license}}

## 2. 源码

{{sourceLink}}

## 3. Why poi-tl

{{#solutionCompare}}

 **poi-tl**是一个基于Apache POI的Word模板引擎，也是一个免费开源的Java类库，你可以非常方便的加入到你的项目中，并且拥有着让人喜悦的特性。

{{#funcDesc}}

## 4. 版本

Apache POI已经进入5.0.0+时代，如果你仍希望使用低版本的Apache POI，请查阅历史版本。

* 当前版本 1.12.0 Documentation，Apache POI5.2.2+，JDK1.8+
* [1.11.x Documentation](http://deepoove.com/poi-tl/1.11.x/)，Apache POI5.1.0+，JDK1.8+
* [1.10.x Documentation](http://deepoove.com/poi-tl/1.10.x/)，Apache POI4.1.2，JDK1.8+
* [1.10.3 Documentation](http://deepoove.com/poi-tl/1.10.3/)，Apache POI4.1.2，JDK1.8+
* [1.9.x Documentation](http://deepoove.com/poi-tl/1.9.x/)，Apache POI4.1.2，JDK1.8+
* [1.8.x Documentation](http://deepoove.com/poi-tl/1.8.x/)，Apache POI4.1.2，JDK1.8+
* [1.7.x Documentation](http://deepoove.com/poi-tl/1.7.x/)，Apache POI4.0.0+，JDK1.8+
* [1.6.x Documentation](http://deepoove.com/poi-tl/1.6.x/)，Apache POI4.0.0+，JDK1.8+
* [1.5.x Documentation](http://deepoove.com/poi-tl/1.5.x/)，Apache POI3.16+，JDK1.6+

V1.12.0版本作了一个不兼容的改动，升级的时候需要注意：

* 重构了PictureRenderData，改为抽象类，建议使用Pictures工厂方法来创建图片数据

## 5. Getting Started

### 5.1. Maven

{{mavenImportCode}}

### 5.2. Gradle

implementation 'com.deepoove:poi-tl:1.12.0'

### 5.3. 2分钟快速入门

新建Word文档template.docx，包含标签 {{label1}}

*template.docx*

{{label1}}

*代码示例*

{{simpleCode}}

compile 编译模板、render 渲染数据、write 输出到流

**TDO模式：Template + data-model = output**

*output.docx*

Hi, poi-tl Word模板引擎

### 5.4. Template：模板

模板是Docx格式的Word文档，你可以使用Microsoft office、WPS Office、Pages等任何你喜欢的软件制作模板，也可以使用Apache POI代码来生成模板。

所有的标签都是以{{label2}}结尾，标签可以出现在任何位置，包括页眉，页脚，表格内部，文本框等，表格布局可以设计出很多优秀专业的文档，推荐使用表格布局。

poi-tl模板遵循**“所见即所得”**的设计，模板和标签的样式会被完全保留。

### 5.5. Data-model：数据

数据类似于哈希或者字典，可以是Map结构（key是标签名称）：

{{mapDataCode}}

可以是对象（属性名是标签名称）：

{{objectDataCode}}

数据可以是树结构，每级之间用点来分隔开，比如{{label3}}标签对应的数据是author对象的name属性值。

FreeMarker、Velocity文本模板中可以通过三个标签设置图片路径、宽和高：

<img src="{{label4}}" width="{{label5}}" height="{{label6}}">

但是Word模板不是由简单的文本表示，所以在渲染图片、表格等元素时提供了数据模型，它们都实现了接口RenderData，比如图片数据模型PictureRenderData包含图片路径、宽、高三个属性。

### 5.6. Output：输出

以流的方式进行输出：

template.write(OutputStream stream);

可以写到任意输出流中，比如文件流：

template.write(new FileOutputStream("output.docx"));

比如网络流：

{{outputCode}}

**最后不要忘记关闭这些流。**

## 6. 标签

### 6.1. 文本

文本标签：{{label7}}

代码示例

{{simpleTextRenderCode}}

链式代码示例

{{simpleChainTextRenderCode}}

效果展示

{{+showSimpleTextRenderData}}

### 6.2. 图片

图片标签：{{label8}}

代码示例

{{simplePicturesRenderCode}}

效果展示

{{+showSimplePicturesRenderData}}

### 6.3. 表格

表格标签：{{label9}}

基础表格代码示例

{{simpleTableCode1}}

表格样式代码示例

{{simpleTableCode2}}

表格合并代码示例

{{simpleMergeTableCode}}

效果展示

{{+showSimpleTableRenderData}}

### 6.4. 列表

列表标签：{{label10}}

代码示例

{{simpleListRenderCode}}

效果展示

{{+showSimpleListRenderData}}

### 6.5. 区块对

区块对由前后两个标签组成，开始标签以?标识，

结束标签以/标识：{{label11}}

**区块对开始和结束标签中间可以包含多个图片、表格、段落、列表、图表等**，开始和结束标签可以跨多个段落，也可以在同一个段落，但是如果在表格中使用区块对，开始和结束标签必须在同一个单元格内，因为跨多个单元格的渲染行为是未知的。

区块对在处理一系列文档元素的时候非常有用，位于区块对中的文档元素可以被渲染零次，一次或N次，这取决于区块对的取值。

**False或空集合**：隐藏区块中的所有文档元素

**非False且不是集合**：显示区块中的文档元素，渲染一次

**非空集合**：根据集合的大小，循环渲染区块中的文档元素

#### 6.5.1. False或空集合

如果区块对的值是 null 、false 或者空的集合，位于区块中的所有文档元素将不会显示，这就等同于if语句的条件为 false。

数据模型和模板

|  |
| --- |
| {{label12}} |

效果展示

|  |
| --- |
| Made it,Ma!{{?announce}}Top of the world!{{/announce}}  Made it,Ma!  {{?announce}}  Top of the world!🎋  {{/announce}} |

#### 6.5.2. 非False且不是集合

如果区块对的值不为 null 、 false ，且不是集合，位于区块中的所有文档元素会被渲染一次，这就等同于if语句的条件为 true。

数据模型和模板

|  |
| --- |
| {{label13}} |

效果展示

|  |
| --- |
| {{?person}}  Hi {{name}}!  {{/person}} |

区块对中标签的作用域会被限定在当前区块对内，当且仅当区块对的值是boolean类型且为true时，这些标签作用域才不会改变。

#### 6.5.3. 非空集合

如果区块对的值是一个非空集合，区块中的文档元素会被迭代渲染一次或者N次，这取决于集合的大小，类似于foreach语法。

数据模型和模板

|  |
| --- |
| {{label14}} |

效果展示

|  |
| --- |
| {{?songs}}  {{name}}  {{/songs}} |

循环内置变量

在循环中提供了一些内置变量，这些内置变量只能用于区块对中。

|  |  |  |
| --- | --- | --- |
| **{{loopInnerVars}}变量** | **类型** | **说明** |
| [varName] | [varType] | [varDesc] |

数据模型和模板（**注意：如果标签内要使用运算符，需要开启Spring表达式**）

|  |
| --- |
| {{label15}} |

效果展示

|  |
| --- |
| {{?produces}}  {{\_index + 1}}. {{=#this}}  {{/produces}} |

### 6.6. 嵌套

嵌套又称为导入、包含或者合并，以+标识：{{label16}}

代码示例

{{simpleNestRenderCode}}

主模板包含嵌套标签{{label17}}

simpleNestSubTemplate.docx是一个包含了{{label18}}的子模板，使用subData集合渲染后合并到主模板

效果展示

-----------------------------------------分界线-------------------------------------

{{+showSimpleNestRenderData}}

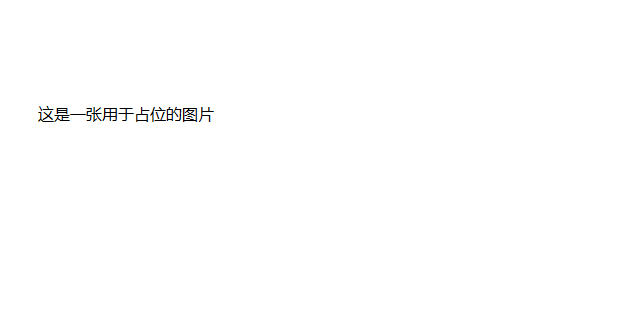
-----------------------------------------分界线-------------------------------------

## 7. 引用标签

引用标签是一种特殊位置的特殊标签，提供了直接引用文档中的元素句柄的能力，这个重要的特性在我们只想改变文档中某个元素极小一部分样式和属性的时候特别有用，因为其余样式和属性都可以在模板中预置好，真正的所见即所得。

### 7.1. 图片

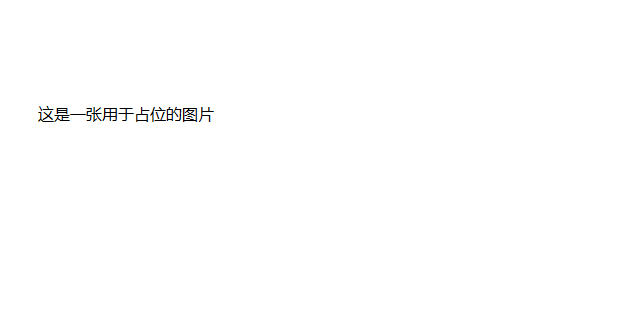
引用图片标签是一个文本：{{label19}}，标签位置在：设置图片格式—​可选文字—​标题或者说明（新版本Microsoft Office标签位置在：编辑替换文字-替换文字）。



引用图片标签只会替换图片而不会改变图片尺寸和布局，数据模型和图片标签一致：PictureRenderData。

代码示例

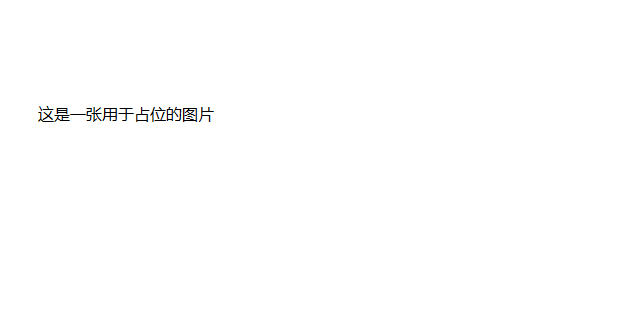
put("refPic", Pictures.ofLocal("sayi.png").create());



### 7.2. 多系列图表

多系列图表指的是条形图（3D条形图）、柱形图（3D柱形图）、面积图（3D面积图）、折线图（3D折线图）、雷达图、散点图等。

多系列图表的标签是一个文本：{{label19}}，标签位置在：图表区格式—​可选文字—​标题（新版本Microsoft Office标签位置在：编辑替换文字-替换文字）。



代码示例

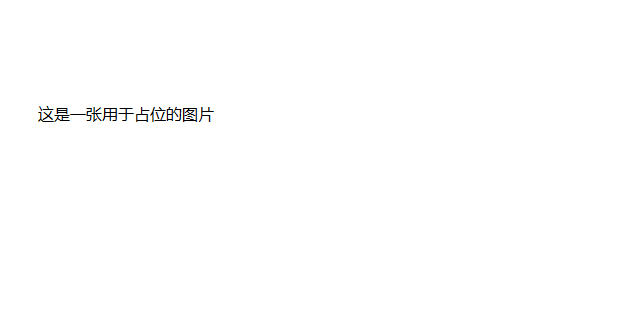
|  |
| --- |
| ChartMultiSeriesRenderData chart = Charts  .ofMultiSeries("ChartTitle", new String[] { "中文", "English" })  .addSeries("countries", new Double[] { 15.0, 6.0 })  .addSeries("speakers", new Double[] { 223.0, 119.0 })  .create();  put("barChart", chart); |

效果展示 (新的图表系列数据会完全替换原有图表数据，而原有图表的样式都会被保留)

### 7.3. 单系列图表

单系列图表指的是饼图（3D饼图）、圆环图等。

单系列图表的标签是一个文本：{{label19}}，标签位置在：图表区格式—​可选文字—​标题（新版本Microsoft Office标签位置在：编辑替换文字-替换文字）。



代码示例

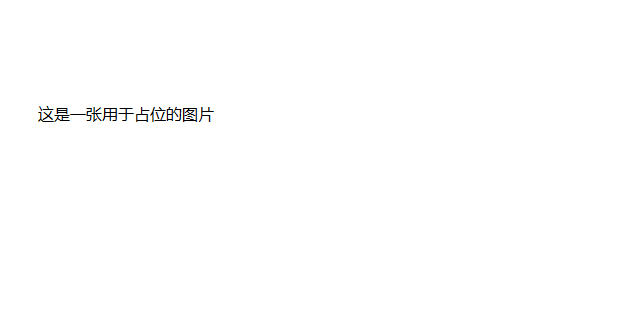
|  |
| --- |
| ChartSingleSeriesRenderData pie = Charts  .ofSingleSeries("ChartTitle", new String[] { "美国", "中国" })  .series("countries", new Integer[] { 9826675, 9596961 })  .create();  put("pieChart", pie); |

效果展示

### 7.4. 组合图表

组合图表指的是由多系列图表（柱形图、折线图、面积图）组合而成的图表。

组合图表的标签是一个文本：{{label19}}，标签位置在：图表区格式—​可选文字—​标题（新版本Microsoft Office标签位置在：编辑替换文字-替换文字）。



代码示例

|  |
| --- |
| ChartSingleSeriesRenderData comb = Charts  .ofComboSeries("MyChart", new String[] { "中文", "English" })  .addBarSeries("countries", new Double[] { 15.0, 6.0 })  .addBarSeries("speakers", new Double[] { 223.0, 119.0 })  .addBarSeries("NewBar", new Double[] { 223.0, 119.0 })  .addLineSeries("youngs", new Double[] { 323.0, 89.0 })  .addLineSeries("NewLine", new Double[] { 123.0, 59.0 }).create();  put("combChart", comb); |

效果展示

## 8. 配置

poi-tl提供了类 Configure 来配置常用的设置，使用方式如下：

|  |
| --- |
| ConfigureBuilder builder = Configure.builder();  XWPFTemplate.compile("template.docx", builder.buid()); |

### 8.1. 前后缀

我一直使用 {{label20}} 的方式来致敬Google CTemplate，如果你更偏爱freemarker${}的方式：

|  |
| --- |
| builder.buildGramer("${", "}"); |

### 8.2. 标签类型

默认的图片标签是以@开始，如果你希望使用%开始作为图片标签：

|  |
| --- |
| builder.addPlugin('%', new PictureRenderPolicy()); |

如果你不是很喜欢默认的标签标识类型，你也可以自由更改：

|  |
| --- |
| builder.addPlugin('@', new TableRenderPolicy());  builder.addPlugin('#', new PictureRenderPolicy()); |

这样{{label8}}就变成了表格标签，{{label9}}变成了图片标签，虽然不建议改变默认标签标识，但是从中可以看到poi-tl插件的灵活度，在插件章节中我们将会看到如何自定义自己的标签。

### 8.3. 标签正则

标签默认支持中文、字母、数字、下划线的组合，我们可以通过正则表达式来配置标签的规则，比如不允许中文：

|  |
| --- |
| builder.buildGrammerRegex("[\\w]+(\\.[\\w]+)\*"); |

比如允许除了标签前后缀外的任意字符：

|  |
| --- |
| builder.buildGrammerRegex(RegexUtils.createGeneral("{{", "}}")); |

### 8.4. 计算标签值

计算标签值是指如何在数据模型中索引标签Key的值，可以完全自定义获取标签值的方式。

|  |
| --- |
| builder.setRenderDataComputeFactory(new RenderDataComputeFactory()); |

RenderDataComputeFactory是一个抽象工厂，你可以定义自己的工厂提供标签表达式计算接口 RenderDataCompute 的实现。

我们可以通过此方式支持任何的表达式引擎，Spring表达式正是通过 SpELRenderDataCompute 实现。

### 8.5. Spring表达式

Spring Expression Language 是一个强大的表达式语言，支持在运行时查询和操作对象图，可作为独立组件使用，需要引入相应的依赖：

|  |
| --- |
| <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-expression</artifactId>  <version>5.3.18</version>  </dependency> |

为了在模板标签中使用SpringEL表达式，需要将标签配置为SpringEL模式：

|  |
| --- |
| builder.useSpringEL(); |

#### 8.5.1. 基本使用

关于SpringEL的写法可以参见[官方文档](https://docs.spring.io/spring-framework/docs/5.3.18/reference/html/core.html" \l "expressions)，下面给出一些典型的示例。

|  |
| --- |
| {{label21}} |

效果展示

|  |
| --- |
| {{name}}  类方法调用，转大写：{{name.toUpperCase()}}  判断条件：{{name == 'poi-tl'}}  {{empty?:'这个字段为空'}}  三目运算符：{{sex ? '男' : '女'}}  类方法调用，时间格式化：{{new java.text.SimpleDateFormat('yyyy-MM-dd HH:mm:ss').format(time)}}  运算符：{{price/10000 + '万元'}}  数组列表使用下标访问：{{dogs[0].name}}  使用静态类方法：{{localDate.format(T(java.time.format.DateTimeFormatter).ofPattern('yyyy年MM月dd日'))}} |

#### 8.5.2. SpringEL作为区块对的条件

Spring表达式与区块对结合可以实现更强大的功能，示例如下：

|  |
| --- |
| {{label22}} |

效果展示（**使用SpringEL时区块对的结束标签可以是：{{label23}}**）

|  |
| --- |
| {{?desc == null or desc == ''}}{{summary}}{{/}}  {{?produces == null or produces.size() == 0}}无{{/}} |

### 8.6. 数据模型序列化

数据模型支持JSON字符串序列化，可以方便的构造远程HTTP或者RPC服务，需要引入相应依赖：

|  |
| --- |
| <dependency>  <groupId>com.deepoove</groupId>  <artifactId>poi-tl-jsonmodel-support</artifactId>  <version>1.0.0</version>  </dependency> |

然后配置数据模型前置转化器即可：

|  |
| --- |
| builder.addPreRenderDataCastor(new GsonPreRenderDataCastor()); |

### 8.7. 错误处理

poi-tl支持在发生错误的时候定制引擎的行为。

#### 8.7.1. 标签无法被计算

标签无法被计算的场景有几种，比如模板中引用了一个不存在的变量，或者级联的前置结果不是一个哈希，如 {{label3}} 中author的值为null，此时就无法计算name的值。

poi-tl可以在发生这种错误时对计算结果进行配置，默认会认为标签值为null。当我们需要严格校验模板是否有人为失误时，可以抛出异常：

|  |
| --- |
| builder.useDefaultEL(true); |

注意的是，如果使用SpringEL表达式，可以通过参数来配置是否抛出异常：

|  |
| --- |
| builder.useSpringEL(true); |

#### 8.7.2. 标签数据类型不合法

我们知道渲染图片、表格等标签时对数据模型是有要求的，如果数据不合法（为NULL或者是一个错误的数据类型），可以配置模板标签的渲染行为。

poi-tl默认的行为会清空标签，如果希望对标签不作任何处理：

|  |
| --- |
| builder.setValidErrorHandler(new DiscardHandler()); |

如果希望执行严格的校验，直接抛出异常：

|  |
| --- |
| builder.setValidErrorHandler(new AbortHandler()); |

### 8.8. 模板生成模板

模板引擎不仅仅可以生成文档，也可以生成新的模板，比如我们把原先的一个文本标签分成一个文本标签和一个表格标签：

|  |
| --- |
| Configure config = Configure.builder().bind("title", new DocumentRenderPolicy()).build();  Map<String, Object> data = new HashMap<>();  DocumentRenderData document = Documents.of()  .addParagraph(Paragraphs.of("{ {title} }").create())  .addParagraph(Paragraphs.of("{ {#table} }").create())  .create();  data.put("title", document); |

### 8.9. 无模板创建文档

使用 XWPFTemplate.create 在无需模板的情况下创建文档，可以充分利用poi-tl友好的API来生成文档元素。

|  |
| --- |
| String text = "this a paragraph";  DocumentRenderData data = Documents.of().addParagraph(Paragraphs.of(text).create()).create();  XWPFTemplate template = XWPFTemplate.create(data); |

### 8.10. 日志

poi-tl使用slf4j作为日志门面，你可以自由选择日志实现，比如logback、log4j等。我们以logback为例：

首先在项目中添加logback依赖：

|  |
| --- |
| <dependency>  <groupId>ch.qos.logback</groupId>  <artifactId>logback-core</artifactId>  <version>1.2.3</version>  </dependency>  <dependency>  <groupId>ch.qos.logback</groupId>  <artifactId>logback-classic</artifactId>  <version>1.2.3</version>  </dependency> |

然后配置logback.xml文件，可以配置日志级别和格式：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <configuration>  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  <encoder>  <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>  </encoder>  </appender>  <logger name="com.deepoove.poi" level="debug" additivity="false">  <appender-ref ref="STDOUT" />  </logger>  <root level="info">  <appender-ref ref="STDOUT" />  </root>  </configuration> |

debug级别的日志会打印解析渲染过程中的信息，有利于程序调试，另外在模板引擎执行结束后会打印耗时信息：

Successfully Render the template file in 13 millis

## 9. 插件

插件，又称为**自定义函数**，它允许用户在模板标签位置处执行预先定义好的函数。由于插件机制的存在，我们几乎可以在模板的任何位置执行任何操作。

**插件是poi-tl的核心**，默认的标签和引用标签都是通过插件加载。

### 9.1. 默认插件

poi-tl默认提供了八个策略插件，用来处理文本、图片、列表、表格、文档嵌套、引用图片、引用多系列图表、引用单系列图表等：

{{\*pluginListDefault}}

由于这八个插件如此通用，因此将这些插件注册为不同的标签类型，从而搭建了poi-tl的标签体系，也构筑了poi-tl高度自由的插件机制。

### 9.2. 开发一个插件

实现一个插件就是要告诉我们在模板的某个地方用某些数据做某些事情，我们可以通过实现RenderPolicy接口开发自己的插件：

|  |
| --- |
| public interface RenderPolicy {  void render(ElementTemplate eleTemplate, Object data, XWPFTemplate template);  } |

ElementTemplate是当前标签位置、data是数据模型、XWPFTemplate代表整个模板

接下来我们写一个将标签替换为Hello, world的插件：

|  |
| --- |
| public class HelloWorldRenderPolicy implements RenderPolicy {  @Override  public void render(ElementTemplate eleTemplate, Object data, XWPFTemplate template) {  XWPFRun run = ((RunTemplate) eleTemplate).getRun();  String thing = "Hello, world";  run.setText(thing, 0);  }  } |

XWPFRun是Apache POI的类，表示当前位置、渲染文本hello, world

poi-tl提供了抽象模板类 AbstractRenderPolicy ，它定义了一些骨架步骤并且将数据模型的校验和渲染逻辑分开，使用泛型约束数据类型，让插件开发起来更简单，接下来我们再写一个更复杂的插件，在模板标签位置完完全全使用代码创建一个表格，这样我们就可以随心所欲的操作表格：

|  |
| --- |
| public class CustomTableRenderPolicy extends AbstractRenderPolicy<Object> {  @Override  protected void afterRender(RenderContext<Object> context) {  // 清空标签  clearPlaceholder(context, true);  }  @Override  public void doRender(RenderContext<Object> context) throws Exception {  XWPFRun run = context.getRun();  BodyContainer bodyContainer = BodyContainerFactory.getBodyContainer(run);  // 定义行列  int row = 10, col = 8;  // 插入表格  XWPFTable table = bodyContainer.insertNewTable(run, row, col);  // 表格宽度  TableTools.setWidth(table, UnitUtils.cm2Twips(14.63f) + "", null);  // 边框和样式  TableTools.borderTable(table, BorderStyle.DEFAULT);  // 1) 调用XWPFTable API操作表格  // 2) 调用TableRenderPolicy.Helper.renderRow方法快速方便的渲染一行数据  // 3) 调用TableTools类方法操作表格，比如合并单元格  // ......  TableTools.mergeCellsHorizonal(table, 0, 0, 7);  TableTools.mergeCellsVertically(table, 0, 1, 9);  }  } |

通过 bodyContainer.insertNewTable 在当前标签位置插入表格，使用XWPFTable API来操作表格。

随心所欲的意思是原则上Apache POI支持的操作，都可以在当前标签位置进行渲染，Apache POI不支持的操作也可以通过直接操纵底层XML来实现。

### 9.3. 使用插件

插件开发好后，为了让插件在某个标签处执行，我们需要将插件与标签绑定。

#### 9.3.1. 将插件应用到标签

当我们有个模板标签为{{label24}}，默认是文本标签，如果希望在这个位置做些不一样或者更复杂的事情，我们可以将插件应用到这个模板标签：

|  |
| --- |
| ConfigureBuilder builder = Configure.builder();  builder.bind("report", new CustomTableRenderPolicy()); |

此时，{ {report} } 将不再是一个文本标签，而是一个自定义标签。

ConfigureBuilder采用了链式调用的方式，可以一次性设置多个标签的插件：

|  |
| --- |
| builder.bind("report", new CustomTableRenderPolicy()).bind("name", new MyRenderPolicy()); |

#### 9.3.2. 将插件注册为新标签类型

当开发的插件具有一定的通用能力就可以将其注册为新的标签类型。比如增加%标识：{{label25}}，对应自定义的渲染策略 HelloWorldRenderPolicy：

|  |
| --- |
| builder.addPlugin('%', new HelloWorldRenderPolicy()); |

此时，{{label25}}将成为一种新的标签类型，它的执行函数是 HelloWorldRenderPolicy。

### 9.4. Plugin Example

接下来用一个完整的代码示例向你展示 Do Anything Anywhere 的想法，它不使用任何poi-tl的默认插件，完全使用自定义函数完成。

插件是一个函数，它的入参是anywhere和anything，函数体就是do something。

|  |
| --- |
| // where绑定policy  Configure config = Configure.builder().bind("sea", new AbstractRenderPolicy<String>() {  @Override  public void doRender(RenderContext<String> context) throws Exception {  // anywhere  XWPFRun where = context.getWhere();  // anything  String thing = context.getThing();  // do 文本  where.setText(thing, 0);  }  }).bind("sea\_img", new AbstractRenderPolicy<String>() {  @Override  public void doRender(RenderContext<String> context) throws Exception {  // anywhere delegate  WhereDelegate where = context.getWhereDelegate();  // any thing  String thing = context.getThing();  // do 图片  FileInputStream stream = null;  try {  stream = new FileInputStream(thing);  where.addPicture(stream, XWPFDocument.PICTURE\_TYPE\_JPEG, 400, 450);  } finally {  IOUtils.closeQuietly(stream);  }  // clear  clearPlaceholder(context, false);  }  }).bind("sea\_feature", new AbstractRenderPolicy<List<String>>() {  @Override  public void doRender(RenderContext<List<String>> context) throws Exception {  // anywhere delegate  WhereDelegate where = context.getWhereDelegate();  // anything  List<String> thing = context.getThing();  // do 列表  where.renderNumbering(Numberings.of(thing.toArray(new String[] {})).create());  // clear  clearPlaceholder(context, true);  }  }).build();  // 初始化where的数据  HashMap<String, Object> args = new HashMap<String, Object>();  args.put("sea", "Hello, world!");  args.put("sea\_img", "sea.jpg");  args.put("sea\_feature", Arrays.asList("面朝大海春暖花开", "今朝有酒今朝醉"));  args.put("sea\_location", Arrays.asList("日落：日落山花红四海", "花海：你想要的都在这里"));  // 一行代码  XWPFTemplate.compile("sea.docx", config).render(args).writeToFile("out\_sea.docx"); |

效果展示

-----------------------------------------分界线-------------------------------------------

{{sea}}

{{sea\_img}}

{{sea\_feature}}

{{sea\_location}}

-----------------------------------------分界线-------------------------------------------

## 10. 更多插件

### 10.1. 插件列表

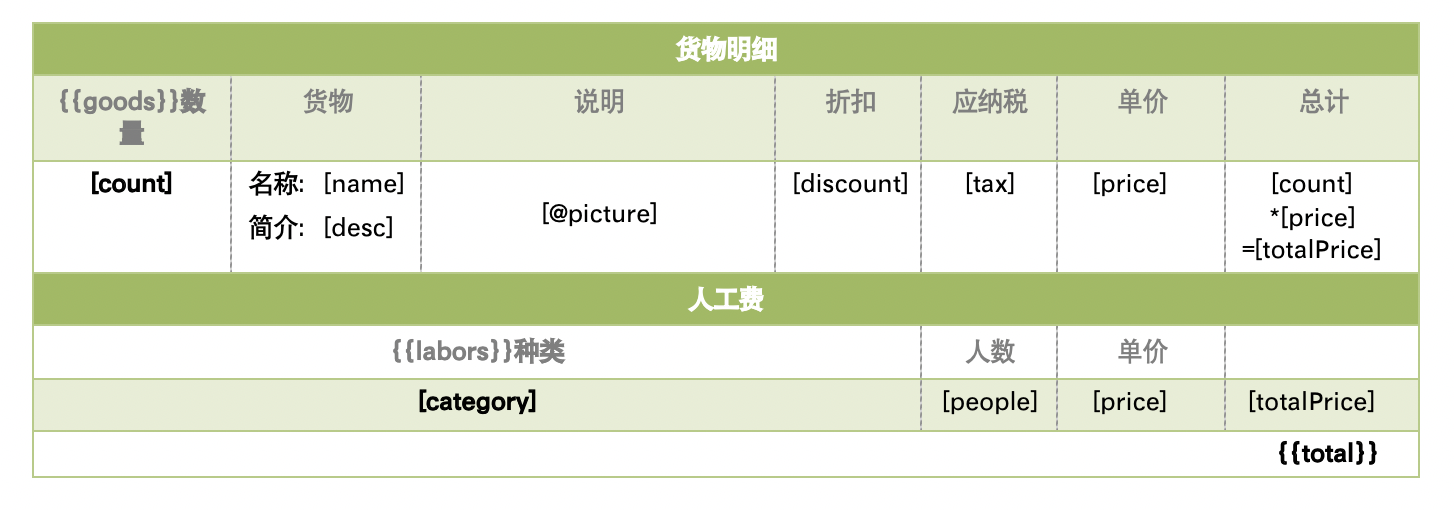
除了八个通用的策略插件外，还内置了一些非常有用的插件。

|  |  |  |
| --- | --- | --- |
| {{loopPlugins}} |  |  |
| [pluginName] | [pluginDesc] | [remark] |

### 10.2. 表格行循环

LoopRowTableRenderPolicy 是一个特定场景的插件，根据集合数据循环表格行。

货物明细和人工费在同一个表格中，货物明细需要展示所有货物，人工费需要展示所有费用。{{label26}}是个标准的标签，将{{label26}}置于循环行的上一行，循环行设置要循环的标签和内容，注意此时的标签应该使用[] ，以此来区别poi-tl的默认标签语法。同理，{{label27}}也置于循环行的上一行。



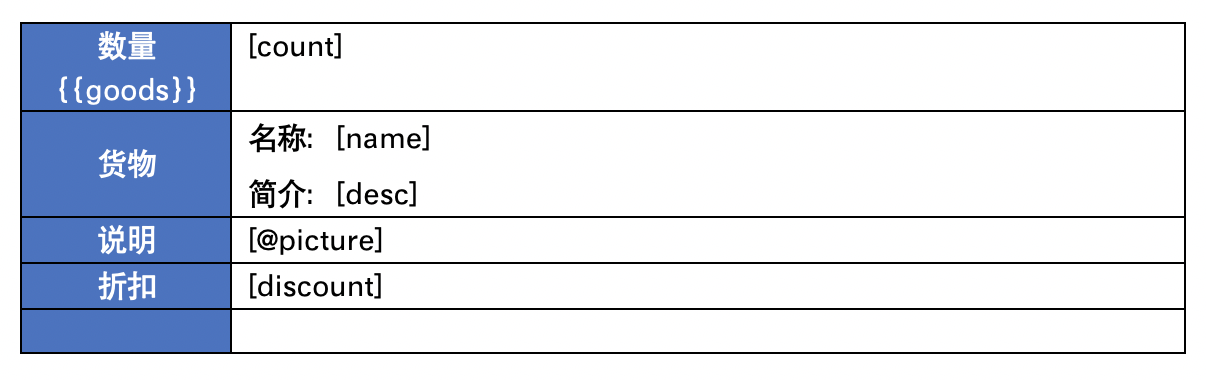
效果展示

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 货物明细 | | | | | | |
| {{goods}}数量 | 货物 | 说明 | 折扣 | 应纳税 | 单价 | 总计 |
| [count] | 名称：[name]  简介：[desc] | [@picture] | [discount] | [tax] | [price] | [count]\*[price]=[totalPrice] |
| 人工费 | | | | | | |
| {{labors}}种类 | | | | 人数 | 单价 |  |
| [category] | | | | [people] | [price] | [totalPrice] |
| {{total}} | | | | | | |

### 10.3. 表格列循环

LoopColumnTableRenderPolicy 是一个特定场景的插件，根据集合数据循环表格列。**要注意的是，由于文档宽度有限，因此模板列必须设置宽度，所有循环列将平分模板列的宽度。**

LoopColumnTableRenderPolicy 循环列的使用方式和插件 LoopRowTableRenderPolicy 是一样的，需要将占位标签放在循环列的前一列。



效果展示

|  |  |
| --- | --- |
| **数量{{goods2}}** | **[count]** |
| **货物** | **名称：[name]**  **简介：[desc]** |
| **说明** | **[@picture]** |
| **折扣** | **[discount]** |
|  |  |

### 10.4. 动态表格

当需求中的表格更加复杂的时候，我们完全可以设计好那些固定的部分，将需要动态渲染的部分单元格交给自定义模板渲染策略。poi-tl提供了抽象表格策略 DynamicTableRenderPolicy 来实现这样的功能。

|  |
| --- |
| public abstract class DynamicTableRenderPolicy implements RenderPolicy {  public abstract void render(XWPFTable table, Object data);  } |

{{label28}}标签可以在表格内的任意单元格内，DynamicTableRenderPolicy会获取XWPFTable对象进而获得操作整个表格的能力。



效果展示

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| {{detail\_table}}货物明细 | | | | | | |
| 数量 | 货物 | 说明 | 折扣 | 应纳税 | 单价 | 总计 |
|  |  |  |  |  |  |  |
| 人工费 | | | | | | |
| 种类 | | | | 人数 | 单价 |  |
|  | | | |  |  |  |
| 总共：{{d\_total}} | | | | | | |

### 10.5. 批注

CommentRenderPolicy 是内置插件，提供了对批注完整功能的支持。

代码示例

|  |
| --- |
| CommentRenderData comment = Comments.of("鹅")  .signature("Sayi", "s", LocaleUtil.getLocaleCalendar())  .comment("鹅，是一种动物")  .create();  Map<String, Object> data = new HashMap<>();  data.put("comment", comment);  Configure config = Configure.builder().bind("comment", new CommentRenderPolicy()).build();  XWPFTemplate.compile("comment\_template.docx", config).render(data); |

效果展示

**咏鹅**

骆宾王

鹅，鹅，鹅，曲项向天歌。

白毛浮绿水，红掌拨清波。

### 10.6. 插入附件

AttachmentRenderPolicy 是内置插件，提供了插入附件功能的支持。

代码示例

|  |
| --- |
| AttachmentRenderData attach = Attachments.ofLocal("attachment.xlsx", AttachmentType.XLSX).create();  Map<String, Object> data = new HashMap<>();  data.put("attachment", attach);  Configure config = Configure.builder().bind("attachment", new AttachmentRenderPolicy()).build();  XWPFTemplate.compile("attachment\_template.docx", config).render(data); |

效果展示

{{attachment}}

### 10.7. 代码高亮

HighlightRenderPolicy 插件对Word代码块进行高亮展示。

引入依赖：

|  |
| --- |
| <dependency>  <groupId>com.deepoove</groupId>  <artifactId>poi-tl-plugin-highlight</artifactId>  <version>1.0.0</version>  </dependency> |

快速开始

代码示例

|  |
| --- |
| HighlightRenderData code = new HighlightRenderData();  code.setCode("/\*\*\n"  + " \* @author John Smith <john.smith@example.com>\n"  + "\*/\n"  + "package l2f.gameserver.model;\n"  + "\n"  + "public abstract strictfp class L2Char extends L2Object {\n"  + " public static final Short ERROR = 0x0001;\n"  + "\n"  + " public void moveTo(int x, int y, int z) {\n"  + " \_ai = null;\n"  + " log(\"Should not be called\");\n"  + " if (1 > 5) { // wtf!?\n"  + " return;\n"  + " }\n"  + " }\n"  + "}");  code.setLanguage("java");  code.setStyle(HighlightStyle.builder().withShowLine(true).withTheme("zenburn").build());  Map<String, Object> data = new HashMap<>();  data.put("code", code);  Configure config = Configure.builder().bind("code", new HighlightRenderPolicy()).build();  XWPFTemplate.compile("highlight\_template.docx", config).render(data); |

效果展示

{{code}}

### 10.8. Markdown

MarkdownRenderPolicy 插件支持通过Markdown生成word文档。

引入依赖：

|  |
| --- |
| <dependency>  <groupId>com.deepoove</groupId>  <artifactId>poi-tl-plugin-markdown</artifactId>  <version>1.0.3</version>  </dependency> |

快速开始

代码示例

|  |
| --- |
| MarkdownRenderData code = new MarkdownRenderData();  code.setMarkdown(new String(Files.readAllBytes(Paths.get("README.md"))));  code.setStyle(MarkdownStyle.newStyle());  Map<String, Object> data = new HashMap<>();  data.put("md", code);  Configure config = Configure.builder().bind("md", new MarkdownRenderPolicy()).build();  XWPFTemplate.compile("markdown\_template.docx", config).render(data); |

效果展示 （通过Markdown插件将poi-tl根目录下的README.md内容转为word文档的结果示例）

|  |
| --- |
| {{md}} |