通用解析验证工具使用手册

|  |  |
| --- | --- |
| 系统名称 |  |
| 项目负责人 |  |
| 作者 |  |
| 文档提交日期 |  |

(版权所有,翻版必究)

修改记录

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No | 修改后  版本号 | 修改内容简介 | 修改日期 | 修改人 |
| 1 | V1.0 | 全文 |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

目 录

[1 简介 1](#_Toc402196653)

[2 使用接口 1](#_Toc402196654)

[2.1 解析接口 1](#_Toc402196655)

[2.2 验证接口 1](#_Toc402196656)

[3 JSON解析 1](#_Toc402196657)

[4 XML解析 2](#_Toc402196658)

[5 Excel解析 4](#_Toc402196659)

[6 Bean验证 5](#_Toc402196660)

[7 参考资料 6](#_Toc402196661)

# 简介

通用解析工具提供源数据到Java对象的解析、以及Java对象的验证功能。输入源可以为输入流或文件，支持的格式为JSON、XML和Excel。

工具采用注解的方式声明解析和验证的规则，方便使用和扩展。

# 使用接口

## 解析接口

|  |
| --- |
| **public** **interface** BeanParser {  **public** <T> List<T> parse(InputStream in, Class<T> clazz) **throws** BeanParseException;  **public** <T> List<T> parse(File file, Class<T> clazz) **throws** BeanParseException;  } |

调用BeanParserBuilder的build静态方法，将返回一个新的BeanParser实现。BeanParser包括parse的两个重载方法，分别对应输入流和文件两种输入方式。Parse方法的第二个参数指定解析结果集的Java对象类型。

BeanParser是线程安全的，可以使用同一个BeanParser实例，不用每次都创建一个。

## 验证接口

|  |
| --- |
| **public** **interface** BeanValidator {  **public** <T> BeanWrapper<T> validate(T t);  **public** <T> BeanResult<T> validate(List<T> list);  } |

调用BeanValidatorBuilder的build静态方法，将返回一个新的BeanValidator实现。BeanValidator包括对对象和List两种验证方法，返回的BeanWrapper中包含一个错误信息的List，BeanResult中包含成功和失败两个List，List中内容为BeanWrapper。

BeanValidator是线程安全的，可以使用同一个BeanValidator实例，不用每次都创建一个。

# JSON解析

工具内部使用jackson实现JSON的解析，解析规则的注解来自于jackson，具体可参考jackson文档。下面以一个简单的例子说明如何使用。

首先，根据源数据中的数据格式定义Java对象。例如，对如下的JSON数据，

|  |
| --- |
| {  "name": "alan",  "age": 25  } |

定义Java对象Person，

|  |
| --- |
| @Source(SourceType.*JSON*)  **public** **static** **class** Person {  @JsonProperty("name")  **private** String name;  @JsonProperty("age")  **private** **int** age;  ……  } |

接下来，在Java对象上定义解析规则。在类上使用Source注解标注输入源的数据类型，这里是JSON；然后在属性上使用JsonProperty注解标注属性在JSON中对应的字段名，这里字段名可以与Java中的属性名不同，比如Java中使用驼峰命名规则，而在JSON数据中可能使用下划线的方式命名字段名。如果不使用JsonProperty注解，那么必须保证Java对象的属性名与JSON数据的字段名一致。

然后，使用BeanParserBuilder创建BeanParser实例，

|  |
| --- |
| BeanParser parser = BeanParserBuilder.*build*(); |

最后，调用BeanParser的parse方法进行解析。

|  |
| --- |
| List<Person> persons = parser.parse(in, Person.**class**); |

# XML解析

工具内部使用JAXB实现XML的解析，解析规则的注解来自JAXB，具体可参考JAXB文档。下面同样以一个简单的例子说明如何使用。

首先，根据源数据中的数据格式定义Java对象。例如，对如下的XML数据

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <persons>  <person>  <name>alan</name>  <age>25</age>  </person>  <person>  <name>G.E.M.</name>  <age>20</age>  </person>  </persons> |

定义Java对象Persons和Person，

|  |
| --- |
| @XmlAccessorType(XmlAccessType.*FIELD*)  @XmlRootElement(name = "persons")  @Source(SourceType.*XML*)  **public** **static** **class** Persons {  @XmlElement(required = **true**)  **private** List<Person> person;  **public** List<Person> getPerson() {  **return** person;  }  **public** **void** setPersons(List<Person> person) {  **this**.person = person;  }  ……  }  @XmlAccessorType(XmlAccessType.*FIELD*)  @XmlRootElement(name = "person")  **public** **static** **class** Person {  @XmlElement(required = **true**, name = "name")  **private** String name;  @XmlElement(required = **true**, name = "age")  **private** **int** age;  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  ……  } |

由于XML数据的嵌套特性，定义的Java对象中必须含有一个容器对象，对应XML的根节点，这里的Persons即为容器对象。

在Java对象上定义解析规则。在类上使用Source注解标注源数据类型，这里为XML；在类上使用XmlRootElement注解标注类对应的XML元素名；在类上使用XmlAccessType注解指定访问Java对象属性的方式，有4个枚举值，其中FIELD表示通过成员变量的方式访问；在属性上使用XmlElement注解标注属性在XML数据中对应的字段名。

然后，使用BeanParserBuilder创建BeanParser实例，

|  |
| --- |
| BeanParser parser = BeanParserBuilder.*build*(); |

最后，调用BeanParser的parse方法进行解析。

|  |
| --- |
| List<Persons> list = parser.parse(in, Persons.**class**); |

由于Persons是容器对象，所以返回的结果集是一个size为1的list。需要编码取出容器对象中的真实数据集合，如下所示。

|  |
| --- |
| List<Person> persons = list.get(0).getPerson(); |

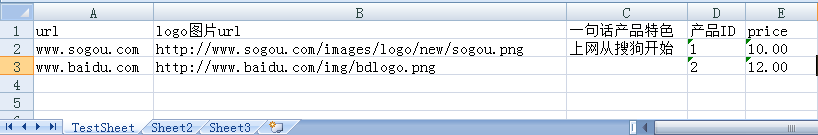
# Excel解析

工具内部使用POI实现Excel的解析，解析规则基于自定义的注解。由于Excel的内容十分灵活和多变，为了简化Excel的解析过程，这里对Excel内容作了一些约束，只有满足以下条件时，才能够完成解析过程。

* Excel的内容是简单的行列结构，即每行数据结构相同，一行表示一条数据，行中的一列表示一条数据中的一个字段；
* 每个sheet的第一行是标题行；
* 各个sheet可定义不同的结果集对象，但每次调用parse方法，只会解析其中一个sheet；

下面同样以一个简单的例子说明如何使用。

首先，根据Excel的数据格式定义Java对象，例如，对如下的Excel，



定义Java对象Item，

|  |
| --- |
| @Source(SourceType.*EXCEL*)  @ExcelSheet("TestSheet")  **public** **class** Item {  @ExcelTitle(names = { "price" })  **private** String price;  @ExcelTitle(names = { "url" })  **private** String url;  @ExcelTitle(names = { "logo图片url" })  **private** String logoUrl;  @ExcelTitle(names = { "一句话产品特色" })  **private** String description;  @ExcelTitle(names = { "产品ID" })  **private** String productId;  ……  } |

在Java对象上定义解析规则。在类上使用Source注解标注输入源的数据类型，这里是EXCEL；在类上使用ExcelSheet注解标注数据在Excel中的哪个sheet上，这里是TestSheet；在属性上用注解ExcelTitle标注属性对应Excel中的哪一列，注解的值必须是对应列的标题。

然后，使用BeanParserBuilder创建BeanParser实例，

|  |
| --- |
| BeanParser parser = BeanParserBuilder.*build*(); |

最后，调用BeanParser的parse方法进行解析。

|  |
| --- |
| List<Item> list = parser.parse(in, Item.**class**); |

这样就得到了TestSheet中的所有数据。

如果有多个sheet需要解析，重复上述过程即可。

# Bean验证

工具内部使用Oval框架实现Bean的验证，验证规则的注解来自Oval，具体可参考Oval的文档；同时，Oval支持自定义验证规则，工具也提供了一些常用的自定义规则，如字符串字节长度的验证等。

首先，在Java对象上结合业务细节定义验证规则，如下例所示。

|  |
| --- |
| @Source(SourceType.*EXCEL*)  @ExcelSheet("TestSheet")  **public** **class** Item {  @ExcelTitle(names = { "price" })  @NotNullEmpty(message = "price必填项，不能为空")  @Numeric(when = "js:\_value != null && \_value != ''", message = "price必须是数字")  @StringByteLength(when = "js:\_value != null && \_value !=''", max = 100, message = "price要求<={max}字节")  @Digits(when = "js:\_value != null && \_value != '' && !isNaN(\_value)", maxFraction = 2, message = "price  小数点后不能超过{maxFraction}位")  @Max(when = "js:\_value != null && \_value != '' && !isNaN(\_value)", value = 999.99, message = "price价格不能高于{max}元")  @Min(when = "js:\_value != null && \_value != '' && !isNaN(\_value)", value = 1, message = "price价格不能低于底价(底价={min})")  **private** String price;    ……  } |

其中，

* NotNullEmpty注解为自定义注解，验证字符串不能为null或长度不能0；
* Numberic注解为自定义注解，验证字符串是否为数字；
* StringByteLength注解为自定义注解，验证字符串的字节是否符合要求；
* Digits注解为Oval提供的注解，验证小数点前后数字的位数；
* Max注解为Oval提供的注解，验证最大值；
* Min注解为Oval提供的注解，验证最小值；

Oval提供了大量的注解，可以灵活组合出符合业务需求的验证规则。

注解中可以用脚本语言指定验证规则启用的条件，如上例注解中的when属性，用一段js脚本指定了启用条件为值不为Null、不为空串、是数值。Oval支持脚本语言有BeanShell、Groovy、JEXL、JavaScript、MVEL、OGNL、Ruby，使用脚本语言需要额外的Jar依赖，具体可参考Oval官方文档。

当使用JavaScript时，可以不指定外部依赖Jar（Mozilla Rhino），这是由于Oval会使用JDK的javax.script包，而默认支持了JavaScript脚本。其他脚本语言就必须添加外部依赖了。

然后，使用BeanValidatorrBuilder创建BeanValidator实例，

|  |
| --- |
| BeanValidator validator = BeanValidatorBuilder.*build*(); |

最后，调用BeanParser的parse方法进行验证。

|  |
| --- |
| BeanResult<Item> result = validator.validate(List<Item> list); |

返回的result包含成功和失败两个List，失败的结果集中包含了具体的异常信息。

# 参考资料