基于XML的网关报文生成解析模板引擎算法

C-3 创建:汤新奇, 最后修改: 汤新奇 今天 23:10

目录

1、网关报文分析
2、基于XML的词法设计
3、基于XML的语法设计
序列化语法描述
反序列化语法描述
4、TTM框架

1、网关报文分析

目前网关使用的报文形式具有多样化的特性，主流包括XML、JSON，KEY-VALUE等。如何通过一套模板引擎来自动化的生成或者解析目标报文，接下来将详细阐述。以XML报文（实际可混合使用）为例，

代码块 XML

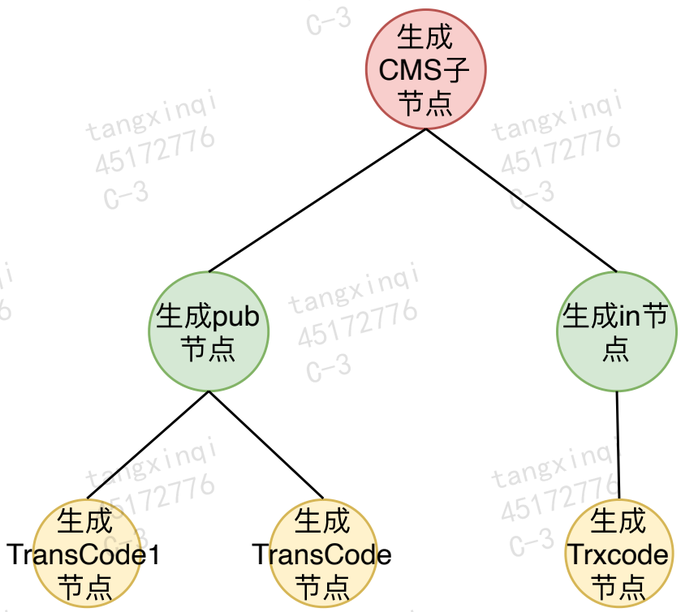
```xml
<?xml version="1.0" encoding="utf-8"?>
<CMS>
  <eb>
    <pub>
      <TransCode1>SZFH_AXZF1</TransCode1>
        <TransCode>SZFH_AXZF</TransCode>
    </pub>
    <in>
      <Trxcode>10004</Trxcode>
    </in>
  </eb>
</CMS>
```

分析报文的生成步骤如下：

- 步骤一：生成TransCode节点： <TransCode1>SZFH_AXZF1</TransCode1>和<TransCode>SZFH_AXZF</TransCode>
- 步骤二：生成pub节点： <pub><TransCode>SZFH_AXZF</TransCode></pub>
- 步骤三：生成Trxcode节点：<Trxcode>10004</Trxcode>
- 步骤四：生成in节点：<in> <Trxcode>10004</Trxcode></in>
- 步骤五：生成eb节点：<eb><pub><TransCode>SZFH_AXZF</TransCode> </pub><in> <Trxcode>10004</Trxcode></in></eb>
- 步骤六：生成CMS节点：<CMS><eb><pub><TransCode>SZFH_AXZF</TransCode></pub><in><Trxcode>10004</Trxcode></in></eb></CMS>

通过分析可以得出，整体报文生成需要 步骤六->步骤五->步骤四->步骤三->步骤二->步骤一，（其中->表示依赖关系），当然步骤N不限于节点生成，还可以包含加密、签名验签等。

如果将上面步骤进行语义转化为如下图：



报文生成- 语法树

问题点提出：如何定义一套语言及语法规则，通过语义分析产生**语法树**，最终翻译成可执行目标引擎，通过引擎生成报文

2、基于XML的词法设计

- 序列化关键字：**template-request、template-text、template-map、template-json、template-xml、template-object、template-array、foreach、key**
- 反序列化关键字：**template-response、template-text、template-xml、template-object、template-array、value、result-set、result-key**

3、基于XML的语法设计

- **序列化语法描述**

关键词	关键字指令	关键字可接受指令	属性	属性含义	属性值
template-request	序列化指令	template-text	-	-	-
		template-map			
		template-json			

		template-xml			
		template-object			
		template-array			
template-text	生成文本指令： • 指令处理文本 • 指令返回文本	文本、${param}	id	指令返回值ID, 可通过$ID获取元素	需全局唯一
		template-text	t-if	判断语句，支持 ==、!=、>=、<=、>、<、&&、||、	param != null
		template-json	ignore	忽略指令结果值传递给父指令，其他指令可通过ID获取	false, true
		template-xml	type-handler	指令返回值后置处理器	handler的name，需全局唯一
		template-array			
		foreach			
template-map	生成map指令 • 指令处理key-value对象 • 指令返回Map	key	id	指令返回值ID, 可通过$ID获取元素	需全局唯一
			t-if	判断语句，支持 ==、!=、>=、<=、>、<、&&、||、	param != null
			ignore	忽略指令结果值传递给父指令，其他指令可通过ID获取	false, true
template-json	生成json指令 • 指令处理key-value对象 • 指令返回json字符串	key	id	指令返回值ID, 可通过$ID获取元素	需全局唯一
			t-if	判断语句，支持 ==、!=、>=、<=、>、<、&&、||、	param != null
			ignore	忽略指令结果值传递给父指令，其他指令可通过ID获取	false, true
			type-handler	指令返回值后置处理器	handler的name，需全局唯一
template-xml	生成xml指令 • 指令处理key-value对象 • 指令返回xml字符串	文本、${param}	id	指令返回值ID, 可通过$ID获取元素	需全局唯一
		key	t-if	判断语句，支持 ==、!=、>=、<=、>、<、&&、||、	param != null
		foreach	ignore	忽略指令结果值传递给父指令，其他指令可通过ID获取	false, true
			type-handler	指令返回值后置处理器	handler的name，需全局唯一
template-object	生成bean指令 • 指令处理key-value对象 • 指令返回bean对象	key	id	指令返回值ID, 可通过$ID获取元素	需全局唯一
			t-if	判断语句，支持 ==、!=、>=、<=、>、<、&&、||、	param != null
			type	对象类型	类全路径名
			ignore	忽略指令结果值传递给父指令，其他指令可通过ID获取	false, true

template-array	生成array指令 • 指令处理数组对象 • 指令返回数组对象 或者数组字符串	template-text	id	指令返回值ID, 可通过$ID获取元素	需全局唯一
		template-map	t-if	判断语句，支持 ==、!=、>=、<=、>、<、&&、||、	param != null
		template-json	depend-on	指令入参	ID或者key
		template-xml	ignore	忽略指令结果值传递给父指令，其他指令可通过ID获取	false, true
		template-object	type-handler	指令返回值后置处理器	handler的name，需全局唯一
		template-array	separator	数组字符串的分隔符	字符串
		key	open	数组字符串的开头	字符串
		foreach	close	数组字符串的结尾	字符串
key	key-value指令 • 构造key-value对象	文本、${param}	name	key的具体值	字符串
		template-text			
		template-map			
		template-json			
		template-xml			
		template-object	t-if	判断语句，支持 ==、!=、>=、<=、>、<、&&、||、	param != null
		template-array			
		key			
		foreach			
foreach	循环指令	文本、${param}	collection	集合源	param
		template-text			
		template-map			
		template-json	t-if	判断语句，支持 ==、!=、>=、<=、>、<、&&、||、	param != null
		template-xml			
		template-object			
		template-array	item	集合元素	

		key			

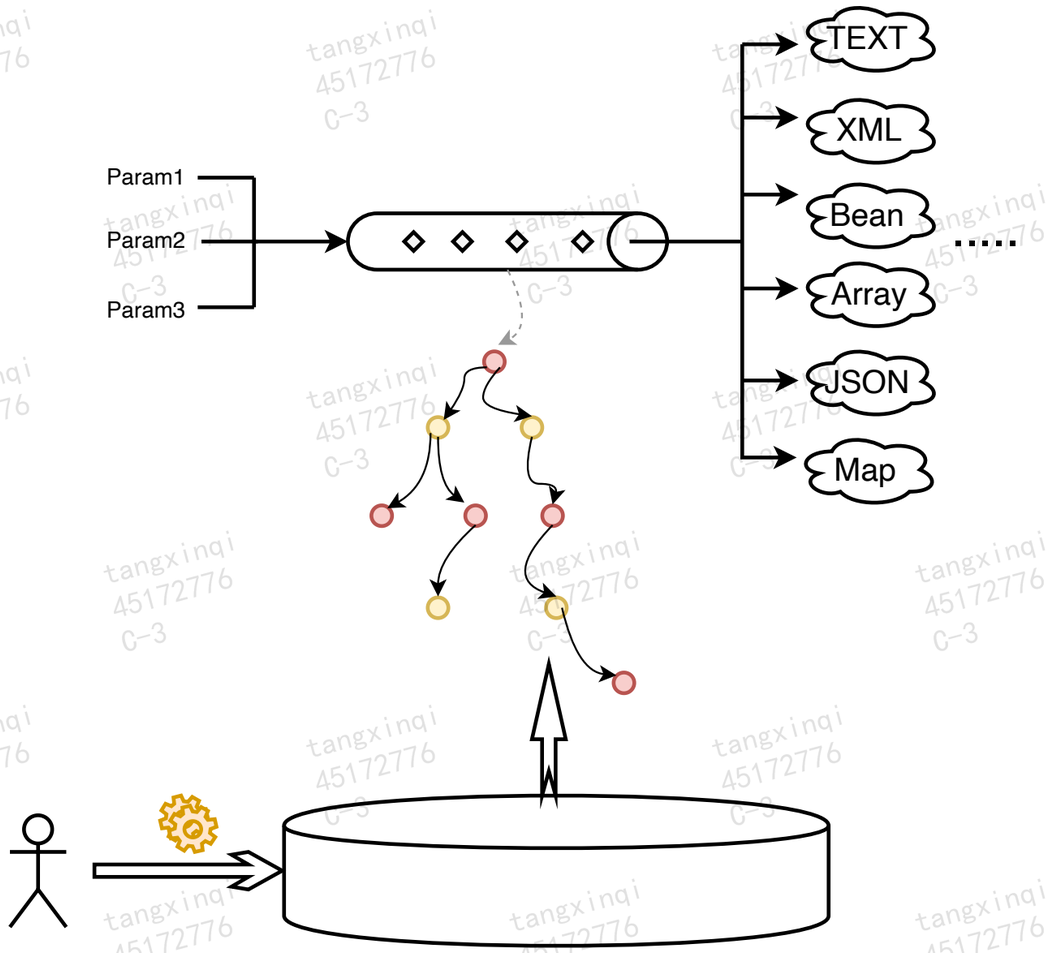
- **反序列化语法描述**

关键词	关键字指令	关键字可接受指令	属性	属性含义	属性值
template-response	反序列化指令	template-text	-	-	-
		template-xml			
		template-object			
		template-array			
		result-set			
template-text	解析文本指令 • 指令受理字符串 • 指令返回字符串	template-text	id	指令返回值ID, 可通过$ID获取元素	需全局唯一
		template-xml	t-if	判断语句，支持 ==、!=、>=、<=、>、<、&&、||、	param != null
			depend-on	指令入参，可接受多个，“,"分隔 open+$key+separator+$key+close拼接	需全局唯一
		template-array	type-handler	指令入参前置处理器	handler的name，需全局唯一
			separator	数组字符串的分隔符	字符
		template-object	open	数组字符串的开头	字符串
			close	数组字符串的结尾	字符串
template-xml	解析xml指令 • 指令受理xml字符串 • 指令返回Document对象	value	id	指令返回值ID, 可通过$ID获取元素	需全局唯一
			t-if	判断语句，支持 ==、!=、>=、<=、>、<、&&、||、	
			depend-on	指令入参，不支持多个	节点ID或者参数key
			type-handler	指令入参前置处理器	handler的name，需全局唯一
template-object	解析Bean指令 • 指令接受json字符串或者Map • 指令返回对象	value	id	指令返回值ID, 可通过$ID获取元素	需全局唯一
			t-if	判断语句，支持 ==、!=、>=、<=、>、<、&&、||、	param != null
			depend-on	指令入参，不支持多个	节点ID或者参数key
			type-handler	指令入参前置处理器	handler的name，需全局唯一

			type	对象类型，默认为HashMap	类全路径名
template-array	解析Array指令 • 指令接受数组或者数组字符串 • 指令返回数组	value	id	指令返回值ID, 可通过$ID获取元素	需全局唯一
			t-if	判断语句，支持 ==、!=、>=、<=、>、<、&&、||、	param != null
			depend-on	指令入参，不支持多个	节点ID或者参数key,
			type-handler	指令入参前置处理器	handler的name，需全局唯一
			separator	数组字符串的分隔符	字符
			open	数组字符串的开头	字符串
			close	数组字符串的结尾	字符串
value	取值指令	空	id	指令返回值ID, 可通过$ID获取元素	需全局唯一
			path	取值路径	path1.path2.path3
result-set	返回数据指令 • 将数据放入目标对象中	result-key	type	对象类型，默认为HashMap	类全名
result-key	返回数据映射指令 • 将数据放入目标对象中的属性中	文本、${param}	name	对应的result-set的属性名称	

4、TTM框架

- 简介：**TTM**(template to message) 框架是基于上述设计的xml模板语言，通过XML模板配置，翻译成JAVA可执行的目标代码，通过目标程序自动化生成或者解析目标数据，目标数据涵盖：TEXT、XML、JSON、Bean、Array、Map等类型。
- 框架原理：



- 应用实例

template-text

序列化	反序列化

代码块 XML

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE template-request PUBLIC "TTM"
"http://localhost:9528/static/template.request.dtd" >
<template-request>
    <template-text>test.content</template-text>
</template-request>
```

代码块 XML

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE template-response PUBLIC "TTM"
"http://localhost:9528/static/template.response.dtd" >
<template-response>
    <!-- 将返回值中key1,key2,key3进行拼接  -->
    <template-text id="text1" depend-on="key1,key2,key3" separator=
open="(" close=")"/>
    <result-set>
        <!--不能动，呼死你-->
        <result-key name="result">${text1}</result-key>
```

```
    </result-set>
</template-response>
```

template-xml

序列化	反序列化

代码块 XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE template-request PUBLIC "TTM"
"http://localhost:9528/static/template.request.dtd" >
<template-request>
    <template-xml>
        <![CDATA[<?xml version="1.0" encoding="UTF-8" ?>]]>
        <key name="xml">
            <key name="test1">
                <key name="test1_test1">content1</key>
                <key name="test1_test2"/>
            </key>
            <key name="test2">content2</key>
            <key name="test3">content3</key>
            <key name="test4">content4</key>
        </key>
    </template-xml>
</template-request>
```

代码块 XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE template-response PUBLIC "TTM"
"http://localhost:9528/static/template.response.dtd" >
<template-response>
    <template-xml>
        <value id="test1" path="xml.test1.test1_test1"/>
        <value id="test2" path="xml.test2"/>
        <value id="test3" path="xml.test3"/>
    </template-xml>
    <result-set>
        <result-key name="test1">${test1}</result-key>
        <result-key name="test2">${test2}</result-key>
        <result-key name="test3">${test3}</result-key>
    </result-set>
</template-response>
```

template-array

序列化	反序列化

代码块 XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE template-request PUBLIC "TTM"
"http://localhost:9528/static/template.request.dtd" >
<template-request>
    <template-array separator="|" open="(" close=")">
        <template-text>${param1}</template-text>
        <template-text>${param2}</template-text>
        <template-text>${param3}</template-text>
        <template-text>${param4}</template-text>
    </template-array>
</template-request>
```

代码块 XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE template-response PUBLIC "TTM"
"http://localhost:9528/static/template.response.dtd" >
<template-response>
    <template-array separator="\|" open="(" close=")">
        <value id="value1" path="$[0]"/>
        <value id="value2" path="$[1]"/>
        <value id="value3" path="$[2]"/>
        <value id="value4" path="$[3]"/>
    </template-array>
    <result-set>
```

```
        <result-key name="key1">${value1}</result-key>
        <result-key name="key2">${value2}</result-key>
        <result-key name="key3">${value3}</result-key>
        <result-key name="key4">${value4}</result-key>
    </result-set>
</template-response>
```

template-object

序列化

代码块 XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE template-request PUBLIC "TTM"
"http://localhost:9528/static/template.request.dtd" >
<template-request>
    <template-object
type="com.txq.pay.supergw.message.boot.starter.test.TestBean">
        <key name="key1">mapValue1</key>
        <key name="key2">mapValue2</key>
        <key name="key3">mapValue3</key>
        <key name="key4">${param1}</key>
    </template-object>
</template-request>
```

反序列化

代码块 XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE template-response PUBLIC "TTM"
"http://localhost:9528/static/template.response.dtd" >
<template-response>
    <template-object
type="com.txq.pay.supergw.message.boot.starter.test.TestBean">
        <value id="test1" path="key1"/>
        <value id="test2" path="key2"/>
        <value id="test3" path="key3"/>
        <value id="test4" path="key4"/>
    </template-object>
    <result-set
type="com.txq.pay.supergw.message.boot.starter.test.TestBean">
        <result-key name="key1">${test1}</result-key>
        <result-key name="key2">${test2}</result-key>
        <result-key name="key3">${test3}</result-key>
        <result-key name="key4">${test4}</result-key>
</result-set>
</template-response>
```

template-map

序列化

代码块 XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE template-request PUBLIC "TTM" "http://localhost:9528/static/template.request.dtd" >
<template-request>
    <template-map>
        <key name="key1">mapValue1</key>
```

```
        <key name="key2">mapValue2</key>
        <key name="key3">mapValue3</key>
        <key name="key4">${param1}</key>
    </template-map>
</template-request>
```

template-json

序列化

代码块 XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE template-request PUBLIC "TTM" "http://localhost:9528/static/template.request.dtd" >
<template-request>
    <template-json>
        <key name="key1">mapValue1</key>
        <key name="key2">mapValue2</key>
        <key name="key3">mapValue3</key>
        <key name="key4">${param1}</key>
    </template-json>
</template-request>
```

- 框架特性（VS"友商"）

1. **去中心化**

TTM提供统一模板源（TemplateSource）接口，开发者可在不同场景下实现不同的子类，从而在不同的场景下加载不同的模板。TTM默认提供了ClassPathResourceTemplateSource子类实现，可将本地resources文件下作为数据源。

2. **灵活性强**

TTM提供标准指令切入点type-handler，开发者可通过实现TypeHandler接口，完成特定指令序列操作（例如先加密后Base64、先Base64后加密）

3. **三重防错机制**

第一重机制：DTD文件-TTM提供标准的XML模板DTD校验文件，可以很好的控制模板格式，同时可以提示开发者具体指令详情，从而能够快速准确的配置模板

第二重机制：jar包版本控制-TTM通过严格的RELEASE版本控制，各个服务之间可以控制pom版本依赖来规避修改带来的问题，同时独立的TypeHandler子类实现，满足设计模式的“开闭原则”和“里氏替换原则”

第三重机制：由于“去中心化”特性，QA可通过引入jar包方式进行简单的二次开发，从而可以依次回归序列化和反序列化的各种case，保证上线前的序列化和反序列化的准确性

4. **具有可视化**

基于XML的模板有大量成熟编辑器编辑（例如idea），提升了开发者的方便性和灵活性；另外统一格式且一体化的模板文件，可以很直观的看出目标结果的具体结构，这有利于问题排查和后期维护。于此同时，不需要投入大量人力开发前端工具，可节省大量开发和维护人力。

- TTM接入-SpringBoot方式

1、pom依赖引入

代码块 XML

```xml
<dependency>
    <groupId>com.txq.pay</groupId>
    <artifactId>message-boot-starter</artifactId>
    <version>1.0.0-SNAPSHOT</version>
</dependency>
```

2、yml配置

代码块 XML

```yaml
ttm:
  # 开启ttm
  enable: true
  # 初始化加载模板列表
  templates: anxin.send,anxin.send.response
```

3、java引用

代码块 Java

```java
@RunWith(SpringRunner.class)
@SpringBootTest
public class MessageTest {

    private MessageSerializer serializer;

    @Autowired
    public void setSerializer(MessageSerializer serializer) {
        this.serializer = serializer;
    }

    @Test
    public void testMessageSerialize() {
        ChannelContext context = new ChannelContext();
        context.setGwOutNo("1234567890");
        context.setRecAccountNo("222222222222222");
        context.setRecAccountName("QDB");
        context.setRecBankNo("102100099996");
        context.setAccountNo("987654321");
        context.setAmount(12345L);
        context.setComment("TEST_COMMENT");
        context.setTradeDate(new Date());
        Map<String, String> result = serializer.serialize("anxin.send", context);
```

```
        log.info("result is {}", result);
    }

    @Test
    public void testMessageDeserialize() {
        String value = "reqData=PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0idXRmLTgiPz4KCjxDTVM+CiAgPGViPgogICAg\n" +
                "PHB1Yj4KICAgICAgPFRyYW5zQ29kZT5YWFg8L1RyYW5zQ29kZT4KICAgICAgPENNJUz5YWFg8L0NJ\n" +
                "Uz4KICAgICAgPEJhbmtDb2RlPlhYWDwvQmFua0NvZGU+CiAgICAgIDxJRD5YWFg8L0lEPgogICAg\n" +
                "ICA8VHJhbkRhdGU+WFhYPC9UaUaW1lPlhYWDwvVHJhblRpbWU+\n" +
                "CiAgICAgIDxmU2Vxbm8+WFhYPC9mU2Vxbm8+CiAgICAgIDxSZXRDb2RlPjAwMDA8L1JldENvZGU+\n" +
                "CiAgICAgIDxSZXRNc2c+vbvS17PJuaY8L1JldE1zZz4KICAgICAgPC91cHVibGljPgogICAgPG91\n" +
                "ICAgPENvbW1zZXE+WFhYPgogICAgICA8VHJhbnNlcT4xMjM0NTY3ODkwOTg3NjU0\n" +
                "MzI8L1RyYW5zZXE+CiAgICAgIDxNc2dzdGF1cy8+CiAgICA8L2ViPgo8L0NNUz4=";
        Map<String, String> result = serializer.deserialize("anxin.send.response", value);
        log.info("response is {}", result);

        value = "errorCode=FT#O999999";
        result = serializer.deserialize("anxin.send.response", value);
        log.info("response is {}", result);
    }
}
```