

spring的SPEL表达式

原创

水岸齐天

已于 2023-06-08 22:03:03 修改

阅读量744

收藏 2

点赞数 1

版权


分类专栏:

Spring

文章标签:

spring

java

 Spring 专栏收录该内容

0 订阅 6 篇文章

订阅专栏

文章目录

Spel概述	
SPEL表达式形式汇总	
SpEL求表达式值步骤:	
步骤	
示例	
SpEL原理及接口	
工作原理	
SpEL的主要接口	
SpEL语法	
基本表达式	
字面量表达式	
算数运算表达式	
关系表达式	
逻辑表达式	
字符串连接及截取表达式	
三目运算	
Elivis运算符	
正则表达式	
括号优先级表达式	
类相关表达式	
类类型表达式	
类实例化	
instanceof表达式	
变量定义及引用	
自定义函数	
表达式赋值	
对象属性存取及安全导航表达式	
对象方法调用	
Bean引用	
集合相关表达式	
内联List	
在Bean定义中使用spel表达式	
xml风格的配置	
注解风格的配置	
总结	
实例	

原文链接：<https://blog.csdn.net/likun557/article/details/107853045>

Spel 概述

Spring表达式语言全称为“Spring Expression Language”，缩写为“SPEL”，类似于Struts2x中使用的 **OGNL** 表达式语言，能在 **运行时** 构建 **复杂表达式**、**存取对象图属性**、**对象方法调用**等等，并且能 **与Spring功能完美整合**，如能用来配置Bean定义。

表达式语言给静态Java语言增加了动态功能。

SpEL是单独模块，只依赖于core模块，不依赖于其他模块，可以单独使用。

SpEL表达式形式汇总

SpEL支持如下表达式：

一、 基本表达式： 字面量表达式、关系，逻辑与算数运算表达式、字符串连接及截取表达式、三目运算及Elivis表达式、正则表达式、括号优先级表达式；

二、 类相关表达式： 类类型表达式、类实例化、instanceof表达式、变量定义及引用、赋值表达式、自定义函数、对象属性存取及安全导航表达式、对象方法调用、Bean引用；

三、 集合相关表达式： 内联List、内联数组、集合，字典访问、列表，字典，数组修改、集合投影、集合选择；不支持多维内联数组初始化；不支持内联字典定义；

四、 其他表达式： 模板表达式:模板必须以“#{”开头，以“}”结尾，如"#{'Hello '}#{‘World!’}"。

注：SpEL表达式中的 关键字 是 不区分大小 写的。

支持SpEL的Jar包：“org.springframework.expression-3.0.5.RELEASE.jar”

SpEL求表达式值步骤：

步骤

一般分为四步，其中第三步可选：

- 1、 创建解析器： SpEL使用ExpressionParser接口表示解析器，提供 SpelExpressionParser 默认实现；
- 2、 解析表达式： 使用ExpressionParser的parseExpression来 解析相应的表达式为Expression对象。
- 3、 构造上下文： 准备比如变量定义等等表达式需要的上下文数据。
- 4、 求值： 通过Expression接口的getValue方法根据上下文获得表达式值。

示例

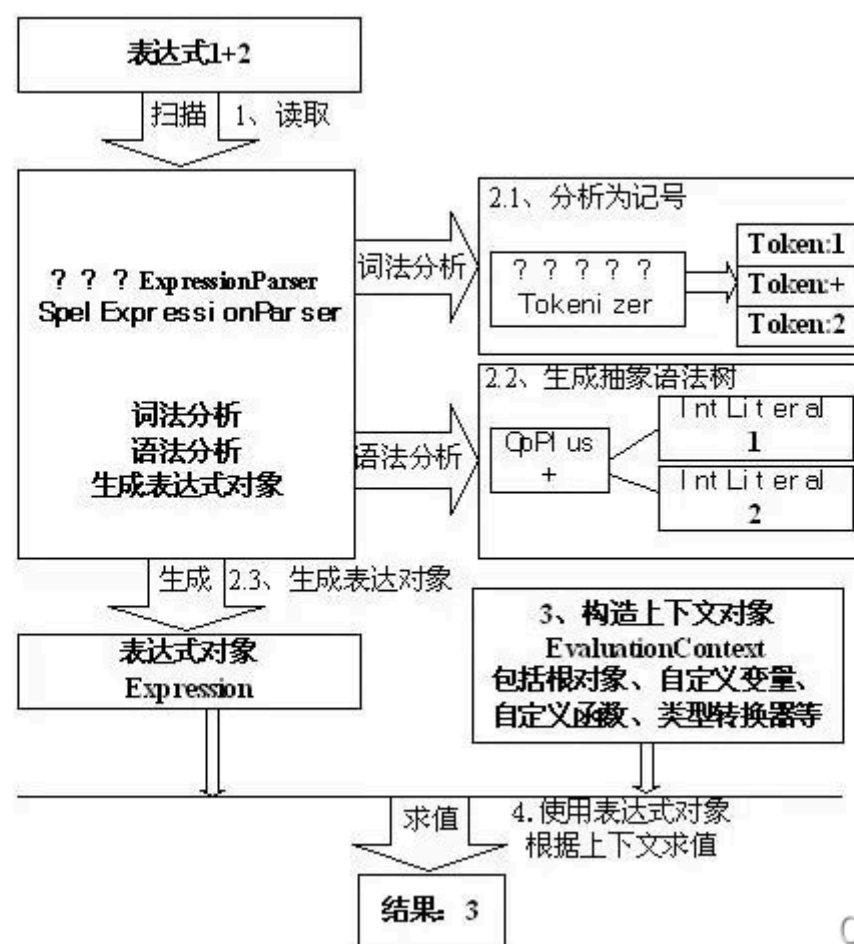
```
1 package com.javacode2018.spel;
2
3 import org.junit.Test;
4 import org.springframework.expression.EvaluationContext;
5 import org.springframework.expression.Expression;
6 import org.springframework.expression.ExpressionParser;
7 import org.springframework.expression.spel.standard.SpelExpressionParser;
8 import org.springframework.expression.spel.support.StandardEvaluationContext;
9
10 public class SpelTest {
11     @Test
12     public void test1() {
13         ExpressionParser parser = new SpelExpressionParser();
14         Expression expression = parser.parseExpression("('Hello' + ' World').concat(#end)");
15         EvaluationContext context = new StandardEvaluationContext();
16         context.setVariable("end", "!");
17         System.out.println(expression.getValue(context));
18     }
19 }
```

输出

Hello World!

SpEL原理及接口

工作原理



CSDN @哎呀呀别老偷看我

- 1.首先定义表达式：“1+2”；
- 2.定义解析器ExpressionParser实现，SpEL提供默认实现SpelExpressionParser；
 - 2.1.SpelExpressionParser解析器内部使用Tokenizer类进行词法分析，即把字符串流分析为记号流，记号在SpEL使用Token类来表示；
 - 2.2.有了记号流后，解析器便可根据记号流生成内部抽象语法树；在SpEL中语法树节点由SpelNode接口实现代表：如OpPlus表示加操作节点、IntLiteral表示int型字面量节点；使用SpelNode实现组成了抽象语法树；
 - 2.3.对外提供Expression接口来简化表示抽象语法树，从而隐藏内部实现细节，并提供getValue简单方法用于获取表达式值；SpEL提供默认实现为SpelExpression；
- 3.定义表达式上下文对象（可选），SpEL使用EvaluationContext接口表示上下文对象，用于设置根对象、自定义变量、自定义函数、类型转换器等，SpEL提供默认实现StandardEvaluationContext；
- 4.使用表达式对象根据上下文对象（可选）求值（调用表达式对象的getValue方法）获得结果。

SpEL的主要接口

ExpressionParser接口

表示解析器，默认实现是org.springframework.expression.spel.standard包中的SpelExpressionParser类，使用parseExpression方法将字符串表达式转换为Expression对象，对于ParserContext接口用于定义字符串表达式是不是模板，及模板开始与结束字符：

```

1 public interface ExpressionParser {
2     Expression parseExpression(String expressionString) throws ParseException;
3     Expression parseExpression(String expressionString, ParserContext context) throws ParseException;
4 }

```

示例：

```

1 @Test
2 public void testParserContext() {
3     ExpressionParser parser = new SpelExpressionParser();
4     ParserContext parserContext = new ParserContext() {
5         @Override
6         public boolean isTemplate() {
7             return true;
8         }
9
10        @Override
11        public String getExpressionPrefix() {
12            return "#{";
13        }
14
15        @Override
16        public String getExpressionSuffix() {
17            return "}";
18        }
19    }

```

```
20     };
21     String template = "#{ 'Hello ' }#{ 'World!' }";
22     Expression expression = parser.parseExpression(template, parserContext);
23     System.out.println(expression.getValue());
}
```

在此我们演示的是使用ParserContext的情况，此处定义了ParserContext实现：定义表达式是模板形式，表达式前缀为“#{”，后缀为“}”；使用parseExpression解析时传入的模板必须以“#{”开头，以“}”结尾，如“#{ 'Hello ' }#{ 'World!' }”。

默认传入的字符串表达式不是模板形式，如之前演示的Hello World。

EvaluationContext接口

表示上下文环境，默认实现是org.springframework.expression.spel.support包中的 StandardEvaluationContext 类，使用 setRootObject方法来设置根对象，使用 setVariable方法来注册自定义变量，使用 registerFunction来注册自定义函数 等等。

Expression接口

表示表达式对象，默认实现是org.springframework.expression.spel.standard包中的 SpelExpression，提供getValue方法用于获取表达式值，提供setValue方法用于设置对象值。

SpEL语法

基本表达式

字面量表达式

SpEL支持的字面量包括：字符串、数字类型（int、long、float、double）、布尔类型、null类型。

类型	示例
字符串	String str1 = parser.parseExpression("Hello World!").getValue(String.class);
数字类型	int int1 = parser.parseExpression("1").getValue(Integer.class); long long1 = parser.parseExpression("-1L").getValue(long.class); float float1 = parser.parseExpression("1.1").getValue(Float.class); double double1 = parser.parseExpression("1.1E+2").getValue(double.class); int hex1 = parser.parseExpression("0xa").getValue(Integer.class); long hex2 = parser.parseExpression("0xaL").getValue(long.class);
布尔类型	boolean true1 = parser.parseExpression("true").getValue(boolean.class); boolean false1 = parser.parseExpression("false").getValue(boolean.class);
null类型	Object null1 = parser.parseExpression("null").getValue(Object.class);

示例：
CSDN @#F

```
1  @Test
2  public void test2() {
3      ExpressionParser parser = new SpelExpressionParser();
4
5      String str1 = parser.parseExpression("'Hello World!'").getValue(String.class);
6      int int1 = parser.parseExpression("1").getValue(Integer.class);
7      long long1 = parser.parseExpression("-1L").getValue(long.class);
8      float float1 = parser.parseExpression("1.1").getValue(Float.class);
9      double double1 = parser.parseExpression("1.1E+2").getValue(double.class);
10     int hex1 = parser.parseExpression("0xa").getValue(Integer.class);
11     long hex2 = parser.parseExpression("0xaL").getValue(long.class);
12     boolean true1 = parser.parseExpression("true").getValue(boolean.class);
13     boolean false1 = parser.parseExpression("false").getValue(boolean.class);
14 }
```

```
15      Object null1 = parser.parseExpression("null").getValue(Object.class);
16
17      System.out.println("str1=" + str1);
18      System.out.println("int1=" + int1);
19      System.out.println("long1=" + long1);
20      System.out.println("float1=" + float1);
21      System.out.println("double1=" + double1);
22      System.out.println("hex1=" + hex1);
23      System.out.println("hex2=" + hex2);
24      System.out.println("true1=" + true1);
25      System.out.println("false1=" + false1);
26      System.out.println("null1=" + null1);
    }
```

输出

str1=Hello World!
int1=1
long1=-1
float1=1.1
double1=110.0
hex1=10
hex2=10
true1=true
false1=false
null1=null

算数运算表达式

SpEL支持加(+)、减(-)、乘(*)、除(/)、求余 (%) 、幂 (^) 运算。

类型	示例
加减乘除	int result1 = parser.parseExpression("1+2-3*4/2").getValue(Integer.class);//-3
求余	int result2 = parser.parseExpression("4%3").getValue(Integer.class);//1
幂运算	int result3 = parser.parseExpression("2^3").getValue(Integer.class);//8

SpEL还提供求余（MOD）和除（DIV）而外两个运算符，与“%”和“/”等价，不区分大小写。

关系表达式

等于 (==) 、不等于(!=)、大于(>)、大于等于(>=)、小于(<)、小于等于(<=)，区间（between）运算。

如parser.parseExpression(“1>2”).getValue(boolean.class);将返回false;
而parser.parseExpression(“1 between {1, 2}”).getValue(boolean.class);将返回true。

between运算符右边操作数必须是列表类型，且只能包含2个元素。第一个元素为开始，第二个元素为结束，区间运算是包含边界值的，即 xxx>=list.get(0) && xxx<=list.get(1)。

SpEL同样提供了等价的“EQ”、“NE”、“GT”、“GE”、“LT”、“LE”来表示等于、不等于、大于、大于等于、小于、小于等于，不区分大小写。

```
1      @Test
2      public void test3() {
```



```
3      ExpressionParser parser = new SpelExpressionParser();
4      boolean v1 = parser.parseExpression("1>2").getValue(boolean.class);
5      boolean between1 = parser.parseExpression("1 between {1,2}").getValue(boolean.class);
6      System.out.println("v1=" + v1);
7      System.out.println("between1=" + between1);
8  }
```

输出

v1=false
between1=true

逻辑表达式

且 (and或者&&) 、或(or或者||)、非(!或NOT)。

```
1  @Test
2  public void test4() {
3      ExpressionParser parser = new SpelExpressionParser();
4
5      boolean result1 = parser.parseExpression("2>1 and (!true or !false)").getValue(boolean.class);
6      boolean result2 = parser.parseExpression("2>1 && (!true || !false)").getValue(boolean.class);
7
8      boolean result3 = parser.parseExpression("2>1 and (NOT true or NOT false)").getValue(boolean.class);
9      boolean result4 = parser.parseExpression("2>1 && (NOT true || NOT false)").getValue(boolean.class);
10
11     System.out.println("result1=" + result1);
12     System.out.println("result2=" + result2);
13     System.out.println("result3=" + result3);
14     System.out.println("result4=" + result4);
15 }
```

输出

result1=true
result2=true
result3=true
result4=false

字符串连接及截取表达式

使用“+”进行字符串连接，使用“String'[0] [index]”来截取一个字符，目前只支持截取一个，如“'Hello ' + 'World!'”得到“Hello World!”；而“'Hello World!'[0]”将返回“H”。

三目运算

三目运算符 “表达式1?表达式2:表达式3”用于构造三目运算表达式，如“2>1?true:false”将返回true；

Elivis运算符

Elivis运算符“表达式1?:表达式2”从Groovy语言引入用于简化三目运算符的，当表达式1为非null时则返回表达式1，当表达式1为null时则返回表达式2，简化了三目运算符方式“表达式1? 表达式1:表达式2”，如“null?:false”将返回false，而“true?:false”将返回true；

正则表达式

使用“str matches regex，如“'123' matches 'd{3}'”将返回true；

括号优先级表达式

使用“(表达式)”构造，括号里的具有高优先级。

类相关表达式

类类型表达式

使用“T(Type)”来表示java.lang.Class实例，“Type”必须是 类全限定名， “java.lang”包除外， 即该包下的类可以不指定包名；使用类类型表达式还可以进行访问类静态方法及类静态字段。

具体使用方法如下：

```
1  @Test
2  public void testClassTypeExpression() {
3      ExpressionParser parser = new SpelExpressionParser();
4      //java.lang包类访问
5      Class<String> result1 = parser.parseExpression("T(String)").getValue(Class.class);
6      System.out.println(result1);
7
8      //其他包类访问
9      String expression2 = "T(com.javacode2018.spel.SpelTest)";
10     Class<SpelTest> value = parser.parseExpression(expression2).getValue(Class.class);
11     System.out.println(value == SpelTest.class);
12
13     //类静态字段访问
14     int result3 = parser.parseExpression("T(Integer).MAX_VALUE").getValue(int.class);
15     System.out.println(result3 == Integer.MAX_VALUE);
16
17     //类静态方法调用
18     int result4 = parser.parseExpression("T(Integer).parseInt('1')").getValue(int.class);
19     System.out.println(result4);
20 }
```

输出

```
class java.lang.String
true
true
1
```

对于java.lang包里的可以直接使用“T(String)”访问；其他包必须是类全限定名；可以进行静态字段访问如“T(Integer).MAX_VALUE”；也可以进行静态方法访问如“T(Integer).parseInt(‘1’)”。

类实例化

类实例化同样使用java关键字“new”，类名必须是全限定名，但java.lang包内的类型除外，如String、Integer。

```
1  @Test
2  public void testConstructorExpression() {
3      ExpressionParser parser = new SpelExpressionParser();
4      String result1 = parser.parseExpression("new String('路人甲java')").getValue(String.class);
5      System.out.println(result1);
6
7      Date result2 = parser.parseExpression("new java.util.Date()").getValue(Date.class);
8      System.out.println(result2);
9  }
```

实例化完全跟Java内方式一样，运行输出

```
路人甲java
Tue Aug 03 20:22:43 CST 2020
```

instanceof表达式

SpEL支持instanceof运算符，跟Java内使用同义；如“haha instanceof T(String)”将返回true。

```
1  @Test
2  public void testInstanceOfExpression() {
3
```

```
1
4 ExpressionParser parser = new SpelExpressionParser();
5 Boolean value = parser.parseExpression("'路人甲' instanceof T(String)").getValue(Boolean.class);
6 System.out.println(value);
}
```

输出

true

变量定义及引用

变量定义 通过EvaluationContext接口的 `setVariable(variableName, value)`方法定义；

在 表达式中使用`"#variableName"`引用；

除了引用自定义变量，SpE还允许引用根对象及当前上下文对象，使用`"#root"`引用根对象，使用`"#this"`引用当前上下文对象；

```
1 @Test
2 public void testVariableExpression() {
3     ExpressionParser parser = new SpelExpressionParser();
4     EvaluationContext context = new StandardEvaluationContext();
5     context.setVariable("name", "路人甲java");
6     context.setVariable("lesson", "Spring系列");
7
8     //获取name变量, Lesson变量
9     String name = parser.parseExpression("#name").getValue(context, String.class);
10    System.out.println(name);
11    String lesson = parser.parseExpression("#lesson").getValue(context, String.class);
12    System.out.println(lesson);
13
14    //StandardEvaluationContext构造器传入root对象, 可以通过#root来访问root对象
15    context = new StandardEvaluationContext("我是root对象");
16    String rootObj = parser.parseExpression("#root").getValue(context, String.class);
17    System.out.println(rootObj);
18
19    //this用来访问当前上下文中的对象
20    String thisObj = parser.parseExpression("#this").getValue(context, String.class);
21    System.out.println(thisObj);
22 }
```

输出

路人甲java

Spring系列

我是root对象

我是root对象

使用`"#variable"`来引用在EvaluationContext定义的变量；除了可以引用自定义变量，还可以使用`"#root"`引用根对象，`"#this"`引用当前上下文对象，此处`"#this"`即根对象。

自定义函数

目前只支持类静态方法注册为自定义函数；SpEL使用StandardEvaluationContext的registerFunction方法进行注册自定义函数，其实完全可以使用setVariable代替，两者其实本质是一样的；

```
1 @Test
2 public void testFunctionExpression() throws SecurityException, NoSuchMethodException {
3     //定义2个函数, registerFunction和setVariable都可以, 不过从语义上面来看用registerFunction更恰当
4     StandardEvaluationContext context = new StandardEvaluationContext();
5     Method parseInt = Integer.class.getDeclaredMethod("parseInt", String.class);
6     context.registerFunction("parseInt1", parseInt);
7     context.setVariable("parseInt2", parseInt);
8
9     ExpressionParser parser = new SpelExpressionParser();
10    System.out.println(parser.parseExpression("#parseInt1('3')").getValue(context, int.class));
11    System.out.println(parser.parseExpression("#parseInt2('3')").getValue(context, int.class));
12 }
```



```
12
13     String expression1 = "#parseInt1('3') == #parseInt2('3')";
14     boolean result1 = parser.parseExpression(expression1).getValue(context, boolean.class);
15     System.out.println(result1);
16 }
```

此处可以看出“registerFunction”和“setVariable”都可以注册自定义函数，但是两个方法的含义不一样，推荐使用“registerFunction”方法注册自定义函数。

运行输出

3
3
true

表达式赋值

使用Expression#setValue方法可以给表达式赋值

```
1  @Test
2  public void testAssignExpression1() {
3      Object user = new Object() {
4          private String name;
5
6          public String getName() {
7              return name;
8          }
9
10         public void setName(String name) {
11             this.name = name;
12         }
13
14         @Override
15         public String toString() {
16             return "$classname{" +
17                 "name='" + name + '\'' +
18                 '}';
19         }
20     };
21     {
22         //user为root对象
23         ExpressionParser parser = new SpelExpressionParser();
24         EvaluationContext context = new StandardEvaluationContext(user);
25         parser.parseExpression("#root.name").setValue(context, "路人甲java");
26         System.out.println(parser.parseExpression("#root").getValue(context, user.getClass()));
27     }
28     {
29         //user为变量
30         ExpressionParser parser = new SpelExpressionParser();
31         EvaluationContext context = new StandardEvaluationContext();
32         context.setVariable("user", user);
33         parser.parseExpression("#user.name").setValue(context, "路人甲java");
34         System.out.println(parser.parseExpression("#user").getValue(context, user.getClass()));
35     }
36 }
```

输出

\$classname{name='路人甲java'}
\$classname{name='路人甲java'}

对象属性存取及安全导航表达式

对象属性获取非常简单，即使用如“a.property.property”这种点缀式获取，SpEL对于属性名首字母是不区分大小写的；SpEL还引入了Groovy语言中的安全导航运算符“(对象|属性)?.属性”，用来避免“?”前边的表达式为null时抛出空指针异常，而是返回null；修改对象属性值则可以通过赋值表达式或Expression接口的setValue

方法修改。

```
1 public static class Car {
2     private String name;
3
4     public String getName() {
5         return name;
6     }
7
8     public void setName(String name) {
9         this.name = name;
10    }
11
12    @Override
13    public String toString() {
14        return "Car{" +
15            "name='" + name + '\'' +
16            '}';
17    }
18 }
19
20 public static class User {
21     private Car car;
22
23     public Car getCar() {
24         return car;
25     }
26
27     public void setCar(Car car) {
28         this.car = car;
29     }
30
31    @Override
32    public String toString() {
33        return "User{" +
34            "car=" + car +
35            '}';
36    }
37 }
38
39 @Test
40 public void test5() {
41     User user = new User();
42     EvaluationContext context = new StandardEvaluationContext();
43     context.setVariable("user", user);
44
45     ExpressionParser parser = new SpelExpressionParser();
46     //使用.符号, 访问user.car.name会报错, 原因: user.car为空
47     try {
48         System.out.println(parser.parseExpression("#user.car.name").getValue(context, String.class));
49     } catch (EvaluationException | ParseException e) {
50         System.out.println("出错了: " + e.getMessage());
51     }
52     //使用安全访问符号?. , 可以规避null错误
53     System.out.println(parser.parseExpression("#user?.car?.name").getValue(context, String.class));
54
55     Car car = new Car();
56     car.setName("保时捷");
57     user.setCar(car);
58
59     System.out.println(parser.parseExpression("#user?.car?.toString()").getValue(context, String.class));
60 }
```

运行输出

出错了：EL1007E: Property or field 'name' cannot be found on null

null

Car{name='保时捷'}

对象方法调用

对象方法调用更简单，跟Java语法一样；如“haha.substring(2,4)”将返回“ha”；而对于根对象可以直接调用方法；

Bean引用

SpEL支持使用“@”符号来引用Bean，在引用Bean时需要使用BeanResolver接口实现来查找Bean，Spring提供BeanFactoryResolver实现。

`@PreAuthorize("@ss.hasPermi('system:config:list')")` 若依权限控制的注解实现

```
1  @Test
2  public void test6() {
3      DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
4      User user = new User();
5      Car car = new Car();
6      car.setName("保时捷");
7      user.setCar(car);
8      factory.registerSingleton("user", user);
9
10     StandardEvaluationContext context = new StandardEvaluationContext();
11     context.setBeanResolver(new BeanFactoryResolver(factory));
12
13     ExpressionParser parser = new SpelExpressionParser();
14     User userBean = parser.parseExpression("@user").getValue(context, User.class);
15     System.out.println(userBean);
16     System.out.println(userBean == factory.getBean("user"));
17 }
```

运行输出

User{car=Car{name='保时捷'}}

true

集合相关表达式

内联List

从Spring3.0.4开始支持内联List，使用{表达式，.....}定义内联List，如“{1,2,3}”将返回一个整型的ArrayList，而“{}”将返回空的List，对于字面量表达式列表，SpEL会使用java.util.Collections.unmodifiableList方法将列表设置为不可修改。

```
1  @Test
2  public void test7() {
3      ExpressionParser parser = new SpelExpressionParser();
4      //将返回不可修改的空List
5      List<Integer> result2 = parser.parseExpression("{}").getValue(List.class);
6      //对于字面量列表也将返回不可修改的List
7      List<Integer> result1 = parser.parseExpression("{1,2,3}").getValue(List.class);
8      Assert.assertEquals(new Integer(1), result1.get(0));
9      try {
10         result1.set(0, 2);
11     } catch (Exception e) {
12         e.printStackTrace();
13     }
14     //对于列表中只要有一个不是字面量表达式，将只返回原始List，
15     //不会进行不可修改处理
16     String expression3 = "{{1+2,2+4},{3,4+4}}";
17     List<List<Integer>> result3 = parser.parseExpression(expression3).getValue(List.class);
18     result3.get(0).set(0, 1);
19     System.out.println(result3);
20     //声明二维数组并初始化
21     int[] result4 = parser.parseExpression("new int[2]{1,2}").getValue(int[].class);
22     System.out.println(result4[1]);
-- }
```

```
23 | //定义一维数组并初始化
24 | int[] result5 = parser.parseExpression("new int[1]").getValue(int[].class);
25 | System.out.println(result5[0]);
26 | }
```

在Bean定义中使用spel表达式

xml风格的配置

SpEL支持在Bean定义时注入，默认使用“#{SpEL表达式}”表示，其中“#root”根对象默认可以认为是ApplicationContext，只有ApplicationContext实现默认支持SpEL，获取根对象属性其实是获取容器中的Bean。

如：

```
1 | <bean id="world" class="java.lang.String">
2 |     <constructor-arg value="#{' World!'}/>
3 | </bean>
4 |
5 | <bean id="hello1" class="java.lang.String">
6 |     <constructor-arg value="#{'Hello'}#{world}"/>
7 | </bean>
8 |
9 | <bean id="hello2" class="java.lang.String">
10 |    <constructor-arg value="#{'Hello' + world}"/>
11 | </bean>
12 |
13 | <bean id="hello3" class="java.lang.String">
14 |    <constructor-arg value="#{'Hello' + @world}"/>
15 | </bean>
```

模板默认以前缀“#{”开头，以后缀“}”结尾，且不允许嵌套，如“#{‘Hello’#{world}}”错误，如“#{‘Hello’ + world}”中“world”默认解析为Bean。当然可以使用“@bean”引用了。

是不是很简单，除了XML配置方式，Spring还提供一种注解方式@Value，接着往下看吧。

注解风格的配置

基于注解风格的SpEL配置也非常简单，使用 **@Value**注解来指定**SpEL表达式**，该注解 可以放到字段、方法及方法参数上。

测试Bean类如下，使用@Value来指定SpEL表达式：

```
1 | public class SpELBean {
2 |     @Value("#{ 'Hello' + world}")
3 |     private String value;
4 | }
```

在Bean定义中SpEL的问题？

如果有同学问“#{我不是SpEL表达式}”不是SpEL表达式，而是公司内部的模板，想换个前缀和后缀该如何实现呢？

我们使用BeanFactoryPostProcessor接口提供postProcessBeanFactory回调方法，它是在IoC容器创建好但还未进行任何Bean初始化时被ApplicationContext实现调用，因此在这个阶段把SpEL前缀及后缀修改掉是安全的，具体代码如下：

```
1 | package com.javacode2018.spel.test1;
2 |
3 | import org.springframework.beans.BeansException;
4 | import org.springframework.beans.factory.config.BeanExpressionResolver;
5 | import org.springframework.beans.factory.config.BeanFactoryPostProcessor;
6 | import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
7 | import org.springframework.context.expression.StandardBeanExpressionResolver;
8 |
```

```

8  import org.springframework.stereotype.Component;
9
10
11 @Component
12 public class SpelBeanFactoryPostProcessor implements BeanFactoryPostProcessor {
13     @Override
14     public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {
15         BeanExpressionResolver beanExpressionResolver = beanFactory.getBeanExpressionResolver();
16         if (beanExpressionResolver instanceof StandardBeanExpressionResolver) {
17             StandardBeanExpressionResolver resolver = (StandardBeanExpressionResolver) beanExpressionResolver;
18             resolver.setExpressionPrefix("%{");
19             resolver.setExpressionSuffix("}");
20         }
21     }
22 }

```

上测试代码

```

1  package com.javacode2018.spel.test1;
2
3  import org.springframework.beans.factory.annotation.Value;
4  import org.springframework.stereotype.Component;
5
6  @Component
7  public class LessonModel {
8      @Value("你好,{@name},{@msg}")
9      private String desc;
10
11      @Override
12      public String toString() {
13          return "LessonModel{" +
14              "desc='" + desc + '\'' +
15              '}';
16      }
17  }

```

@name: 容器中name的bean

@msg: 容器中msg的bean

下面我们来个配置类，顺便定义name和msg这2个bean，顺便扫描上面2个配置类

```

1  package com.javacode2018.spel.test1;
2
3  import org.springframework.context.annotation.Bean;
4  import org.springframework.context.annotation.ComponentScan;
5  import org.springframework.context.annotation.Configuration;
6
7  @ComponentScan
8  @Configuration
9  public class MainConfig {
10      @Bean
11      public String name() {
12          return "路粉";
13      }
14
15      @Bean
16      public String msg() {
17          return "欢迎和我一起学习java各种技术! ";
18      }
19  }

```

测试用例

```

1  @Test
2  public void test12() {
3

```



```

3
4     AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();
5     context.register(MainConfig.class);
6     context.refresh();
7     LessonModel lessonModel = context.getBean(LessonModel.class);
8     System.out.println(lessonModel);
9 }

```

运行输出

LessonModel{desc='你好,路粉,欢迎和我一起学习java各种技术! '}

总结

Spel功能还是比较强大的，可以脱离spring环境独立运行

spel 可以用在一些动态规则的匹配方面，比如 监控系统中监控规则的动态匹配；其他的一些条件动态判断等等

案例源码

<https://gitee.com/javacode2018/spring-series>

实例

通过aop方式实现日志记录、权限校验、数据统计等功能

日志记录

```

import com.google.common.collect.Lists;
import lombok.extern.slf4j.Slf4j;
import org.apache.commons.collections4.CollectionUtils;
import org.apache.commons.lang3.StringUtils;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.aspectj.lang.reflect.MethodSignature;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.LocalVariableTableParameterNameDiscoverer;
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.Expression;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;

import java.lang.reflect.Method;
import java.util.*;

@Slf4j
@Configuration
@Aspect
public class LogAspect {

    @Autowired
    ILogService logService;

    private final LocalVariableTableParameterNameDiscoverer discoverer = new LocalVariableTableParameterNameDiscoverer();

    //1.定义切点 切注解
    @Pointcut("@annotation(com.xxx.aop.LogAnnotation)")
    public void logPoint() {
        // 应该是空的。其实该方法本身只是一个标识，供@Pointcut注解依附。
    }

    //2.定义增强 后置返回增强 在目标方法正常结束后被注入 如果，目标方法发生异常 不会被织入
    @AfterReturning("logPoint()")
    public void logAfter(JoinPoint joinPoint) {
        // joinPoint 连接点对象 得到切点相关的信息
    }

```

```

MethodSignature signature = (MethodSignature) joinPoint.getSignature();
Method method = signature.getMethod();
LogAnnotation annotation = method.getAnnotation(LogAnnotation.class);
// 获取ids和权限点
List<String> ids = extractIdList(joinPoint, annotation);
if (CollectionUtils.isEmpty(ids)) {
    // 动作 (需要识别的操作动作)
    String action = annotation.action().getAction();
    ids.forEach(id -> {
        if (Objects.isNull(id)) {
            return;
        }
    });
    logService.insertLog(Collections.singletonList(Long.parseLong(id)), action);
}
}
}

```

```

private List<String> extractIdList(JoinPoint pjp, LogAnnotation annotation) {
    String spelExpression = annotation.spelExpression();
    if (StringUtils.isBlank(spelExpression)) {
        return null;
    }
    Object[] args = pjp.getArgs();
    // 从切入点织入点处通过反射机制获取织入点处的方法
    MethodSignature signature = (MethodSignature) pjp.getSignature();
    // 获取切入点所在的方法
    Method method = signature.getMethod();
    // 实例化spel表达式解析器
    SpelExpressionParser spelExpressionParser = new SpelExpressionParser();
    // 绑定参数到上下文
    EvaluationContext context = this.bindParam(method, args);
    // 解析表达式内容
    Expression expression = spelExpressionParser.parseExpression(spelExpression);
    // 声明StandardEvaluationContext对象, 用于设置上下文对象。
    Object result = expression.getValue(context);
    return parseIds(result);
}

```

```

/**
 * 将方法的参数名和参数值绑定
 *
 * @param method 方法, 根据方法获取参数名
 * @param args 方法的参数值
 * @return context
 */
private EvaluationContext bindParam(Method method, Object[] args) {
    // 获取方法的参数名
    String[] params = discoverer.getParameterNames(method);
    // 将参数名与参数值对应起来
    EvaluationContext context = new StandardEvaluationContext();
    for (int len = 0; len < params.length; len++) {
        context.setVariable(params[len], args[len]);
    }
    return context;
}

```

```

private List<String> parseIds(Object object) {
    if (object == null) {
        return null;
    }
    if (object instanceof String) {
        String[] arr = object.toString().split(",");
        return Arrays.asList(arr);
    }
    if (object instanceof List) {
        List<String> stringList = new ArrayList<>();
        for (Object o : (List) object) {
            stringList.addAll(parseIds(o));
        }
    }
}

```

```
        return stringList;
    }
    return Lists.newArrayList(object.toString());
}
}
```