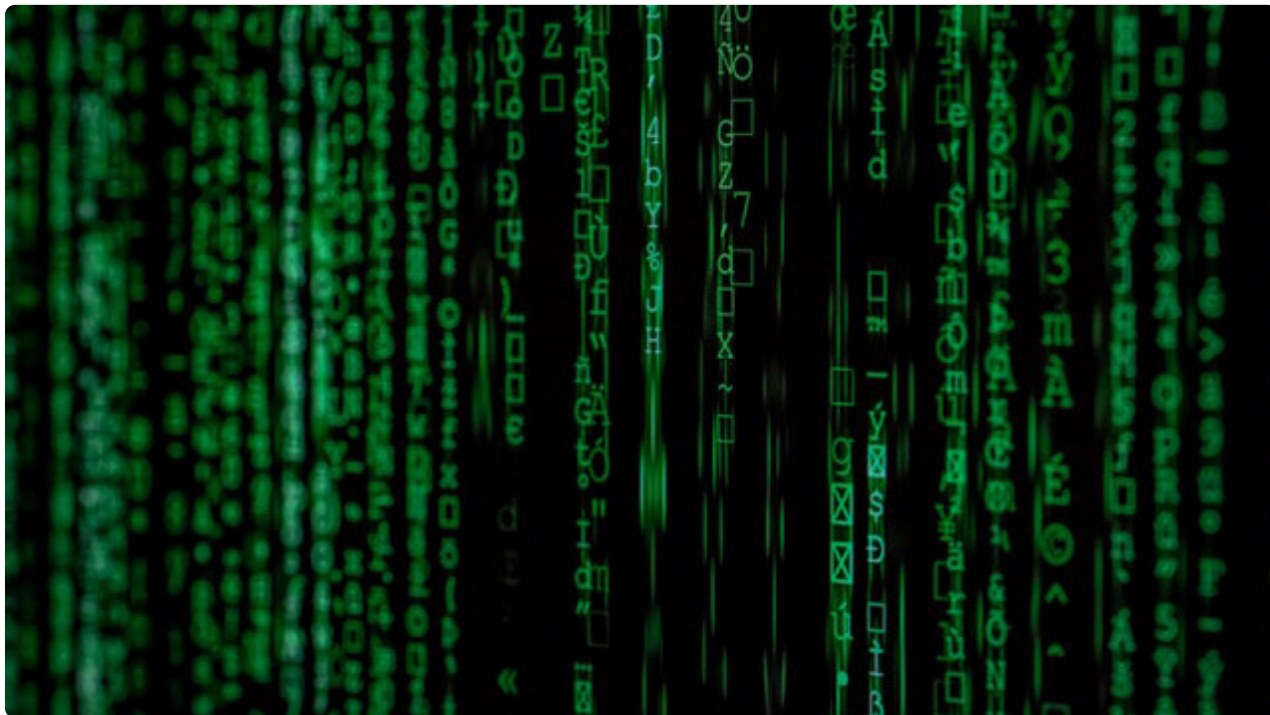


35 SpEL 操作符应用示例及背后原理探究

更新时间：2020-08-14 09:56:53



“知识犹如人体的血液一样宝贵。——高士其”

背景

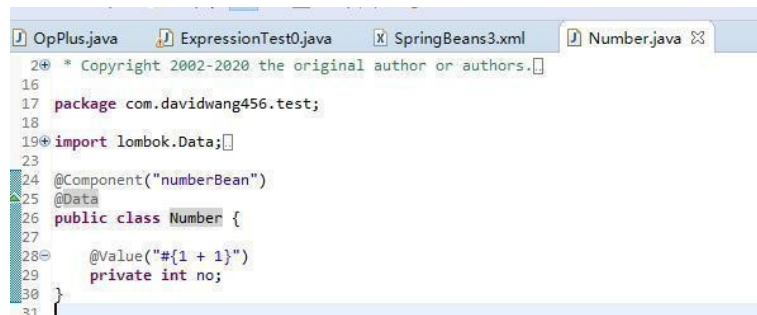
Spring EL 支持大多数标准的数学，逻辑和关系运算符。

- 关系运算符：相等（`==`, `eq`），不等（`!=`, `ne`），小于（`<`, `lt`），小于等于（`<=`, `le`），大于（`>`, `gt`），大于等于（`>=`, `ge`）；
- 逻辑运算符：与（`and`, `&&`），或（`or`, `||`），非（`not`, `!`）；
- 数学运算符：加（`+`），减（`-`），乘（`*`），除（`/`），模（`%`），幂指数（`^`）；
- 赋值运算符：`=`。

Tips: 与 Java 中的运算相同，SpEL 中的 `+` 同样支持字符串拼接操作。

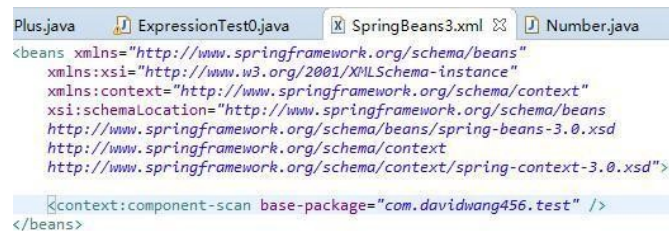
SpEL 操作符示例

说干就干，准备最简单的示例：



```
20 * Copyright 2002-2020 the original author or authors.
16
17 package com.davidwang456.test;
18
19 import lombok.Data;
20
21 @Component("numberBean")
22 @Data
23 public class Number {
24
25     @Value("#{1 + 1}")
26     private int no;
27 }
28
```

配置文件：



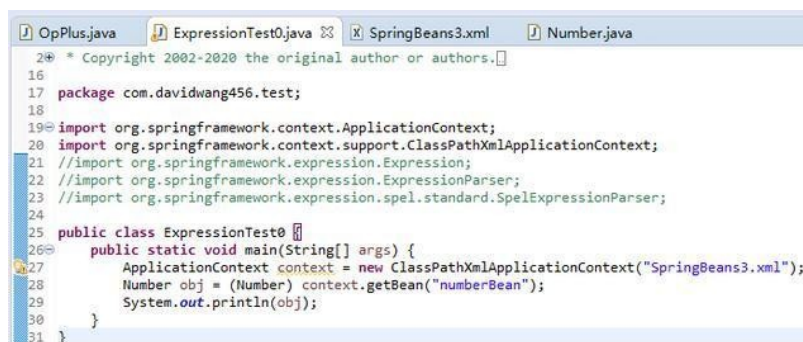
```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:component-scan base-package="com.davidwang456.test" />
</beans>

```

测试类：



```

20 * Copyright 2002-2020 the original author or authors.
16
17 package com.davidwang456.test;
18
19 import org.springframework.context.ApplicationContext;
20 import org.springframework.context.support.ClassPathXmlApplicationContext;
21 //import org.springframework.expression.Expression;
22 //import org.springframework.expression.ExpressionParser;
23 //import org.springframework.expression.spel.standard.SpelExpressionParser;
24
25 public class ExpressionTest0 {
26
27     public static void main(String[] args) {
28         ApplicationContext context = new ClassPathXmlApplicationContext("SpringBeans3.xml");
29         Number obj = (Number) context.getBean("numberBean");
30         System.out.println(obj);
31     }
32 }
33
```

上面使用的是注解 `@value` 的方式，当然我们还可以使用硬编码的方式：

```

28
29     ExpressionParser parser = new SpelExpressionParser();
30     Expression exp= parser.parseExpression(" 1 + 1 ");
31     int ret = exp.getValue(Integer.class);
32     System.out.println(ret);
33
```

也可以在 XML 中定义：



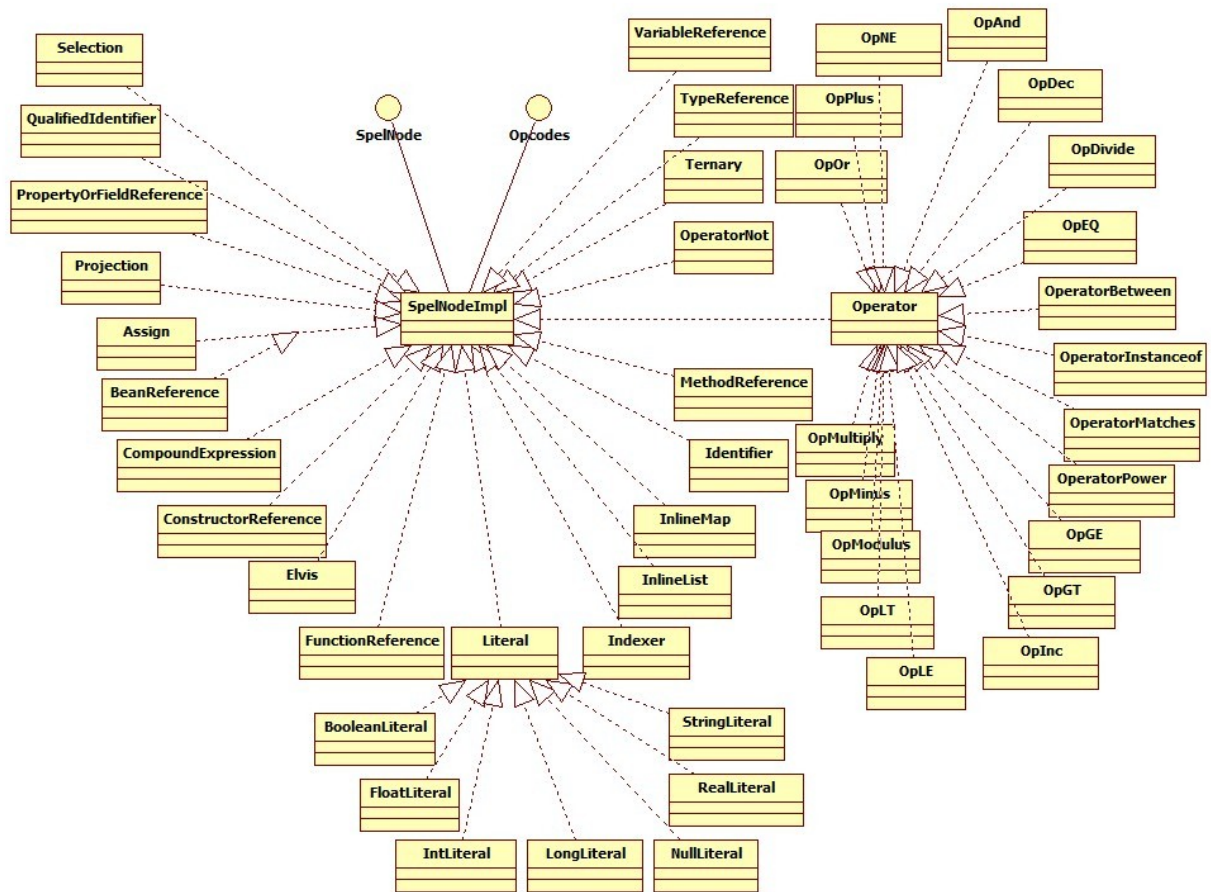
```

1 <beans xmlns="http://www.springframework.org/schema/beans"
2       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3       xmlns:context="http://www.springframework.org/schema/context"
4       xsi:schemaLocation="http://www.springframework.org/schema/beans
5 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
6 http://www.springframework.org/schema/context
7 http://www.springframework.org/schema/context/spring-context-3.0.xsd">
8
9     <context:component-scan base-package="com.davidwang456.test" />
10    <bean id="numberBean" class="com.davidwang456.test.Number">
11        <property name="no" value="#{ 1 + 1 }"></property>
12    </bean>
13 </beans>

```

SpEL 操作符探秘

我们知道，SpEL 最终要转换为 AST，然后执行 `SpelNodeImpl` 的实现类来完成相应的计算，那么本示例中，加法运算需要借助 OpPlus 实现，如下图所示：



如果想了解调用链，有以下三种方式：

- 第一种方式： debug 进去，一步步开始调试 F5 进去，最终得到整个调用链。这种方式最慢；
- 第二种方式： 从 OpPlus 的 getValueInternal 的方法开始，使用类似 eclipse 的 Open Call Hirarcy，这种方式可能会遇到多个分支的情况；
- 第三种方式： 修改 OpPlus 的 getValueInternal 方法源码，在其中抛出一个异常：

```
throw new NullPointerException();
```

执行时会抛出整个异常调用链接，我们可以根据控制台打印的错误日志，找到调用链。

我们最终得到的调用链如下图所示，因一个图太长，分成两个图：

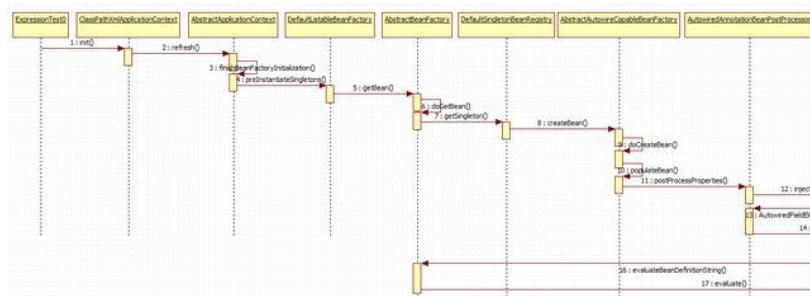


图 1 调用链

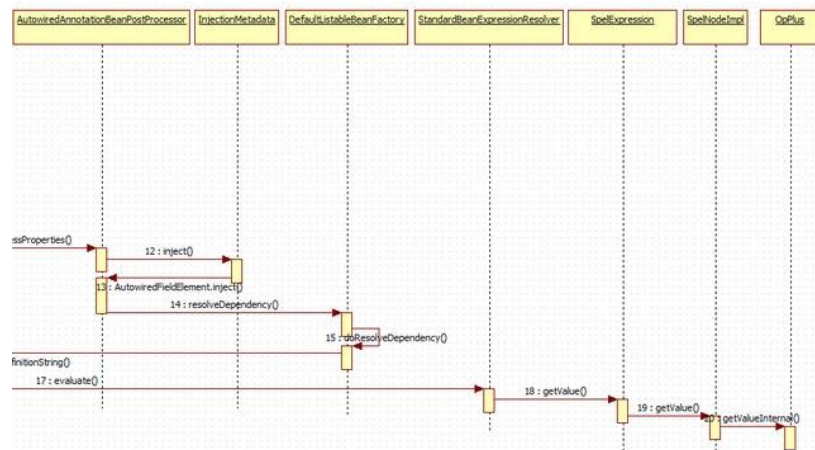


图 2 调用链

重点 1: **StandardBeanExpressionResolver** 封装了 **ExpressionParser**, 并且利用各种 **PropertyAccessor** 的实现来读取属性:

```
@Override
@Nullable
public Object evaluate(@Nullable String value, BeanExpressionContext evalContext) throws BeansException {
    if (!StringUtils.hasLength(value)) {
        return value;
    }
    try {
        Expression expr = this.expressionCache.get(value);
        if (expr == null) {
            expr = this.expressionParser.parseExpression(value, this.beanExpressionParserContext);
            this.expressionCache.put(value, expr);
        }
        StandardEvaluationContext sec = this.evaluationCache.get(evalContext);
        if (sec == null) {
            sec = new StandardEvaluationContext(evalContext);
            sec.addPropertyAccessor(new BeanExpressionContextAccessor());
            sec.addPropertyAccessor(new BeanFactoryAccessor());
            sec.addPropertyAccessor(new MapAccessor());
            sec.addPropertyAccessor(new EnvironmentAccessor());
            sec.setBeanResolver(new BeanFactoryResolver(evalContext.getBeanFactory()));
            sec.setTypeLocator(new StandardTypeLocator(evalContext.getBeanFactory().getBeanClassLoader()));
            ConversionService conversionService = evalContext.getBeanFactory().getConversionService();
            if (conversionService != null) {
                sec.setTypeConverter(new StandardTypeConverter(conversionService));
            }
            customizeEvaluationContext(sec);
            this.evaluationCache.put(evalContext, sec);
        }
        return expr.getValue(sec);
    }
    catch (Throwable ex) {
        throw new BeanExpressionException("Expression parsing failed", ex);
    }
}
```

1: parser
2: context
3: 封装各种 PropertyAccessor 实现

重点 2: 加法的实现:

```
OpPlus.java 33
92     if (leftNumber instanceof BigDecimal || rightNumber instanceof BigDecimal) {
93         BigDecimal leftBigDecimal = NumberUtils.convertNumberToTargetClass(leftNumber, BigDecimal.class);
94         BigDecimal rightBigDecimal = NumberUtils.convertNumberToTargetClass(rightNumber, BigDecimal.class);
95         return new TypedValue(leftBigDecimal.add(rightBigDecimal));
96     }
97     else if (leftNumber instanceof Double || rightNumber instanceof Double) {
98         this.exitTypeDescriptor = "D";
99         return new TypedValue(leftNumber.doubleValue() + rightNumber.doubleValue());
100    }
101    else if (leftNumber instanceof Float || rightNumber instanceof Float) {
102        this.exitTypeDescriptor = "F";
103        return new TypedValue(leftNumber.floatValue() + rightNumber.floatValue());
104    }
105    else if (leftNumber instanceof BigInteger || rightNumber instanceof BigInteger) {
106        BigInteger leftBigInteger = NumberUtils.convertNumberToTargetClass(leftNumber, BigInteger.class);
107        BigInteger rightBigInteger = NumberUtils.convertNumberToTargetClass(rightNumber, BigInteger.class);
108        return new TypedValue(leftBigInteger.add(rightBigInteger));
109    }
110    else if (leftNumber instanceof Long || rightNumber instanceof Long) {
111        this.exitTypeDescriptor = "J";
112        return new TypedValue(leftNumber.longValue() + rightNumber.longValue());
113    }
114    else if (CodeFlow.isIntegerForNumericOp(leftNumber) || CodeFlow.isIntegerForNumericOp(rightNumber)) {
115        this.exitTypeDescriptor = "I";
116        return new TypedValue(leftNumber.intValue() + rightNumber.intValue());
117    }
118    else {
119        // Unknown Number subtypes -> best guess is double addition
120        return new TypedValue(leftNumber.doubleValue() + rightNumber.doubleValue());
121    }
122 }
123
124 if (leftOperand instanceof String && rightOperand instanceof String) {
125     this.exitTypeDescriptor = "Ljava/lang/String";
126     return new TypedValue((String) leftOperand + rightOperand);
127 }
128
129 if (leftOperand instanceof String) {
130     return new TypedValue(
131         leftOperand + (rightOperand == null ? "null" : convertTypedValueToString(operandTwoValue, state)));
132 }
133
134 if (rightOperand instanceof String) {
135     return new TypedValue(
136         (leftOperand == null ? "null" : convertTypedValueToString(operandOneValue, state)) + rightOperand);
137 }
138
139 return state.operate(Operation.ADD, leftOperand, rightOperand);
140 }
141
```

总结

SpEL 为 bean 的属性进行动态赋值提供了便利。Spring EL 支持大多数标准的数学，逻辑和关系运算符。

StandardBeanExpressionResolver 实现了 BeanExpressionResolver，通过 spring 的表达式模块来解析和计算 Spring EL。

StandardEvaluationContext 是 EvaluationContext 的一个强有力的并且高可配置的实现，基于反射的方式来解析属性，方法和成员。

}



34 SpEL List和Map 引用应用示例
及背后原理探究

36 SpEL 正则表达式应用示例及
背后原理探究

