33 SpEL String应用示例及背后原理探究

更新时间：2020-08-04 19:31:30

> 不安于小成，然后足以成大器；不诱于小利，然后可以立远功。——方孝孺

背景

在以前的篇章中，JavaBean 都是静态的，简单的赋值。本节开始，我们开始尝试着让 JavaBean 动起来！

想要让 JavaBean 动起来，就需要支持很多动态的赋值和复杂的运算方式，正如有过 JSP + Servlet 基础的童鞋一定会联想到 EL 表达式。

Spring 也提供了对 EL 表达式的支持，Spring Expression Language 简称 SpEL。

SpEL 概述

SpEL 创建的初衷是给 Spring 社区提供一种简单而高效的表达式语言，一种可贯穿整个 Spring 产品组的语言。这种语言的特性基于 Spring 产品的需求而设计，这是它出现的一大特色。

SpEL 是一种强大的表达式语言，支持在 bean 创建时或运行时查询和操作对象。它类似于其他表达式语言，如 JSP EL、OGNL、MVEL 和 JBoss EL 等，还有一些附加特性，如方法调用和基本的字符串模板功能。

语法形式：

```
#{ expression }
```

它可以用来：

它可以用于注入一个 bean 或另一个 bean 中的一个 bean 属性：

```
<property name="car" value="#{car}" />
```

或者：

```
<!-- 引用其他对象的属性 -->
<property name="carName" value="#{car.name}" />
```

它可以用来调用另一个 bean 中的一个 bean 方法：

```
<!-- 引用其他对象的方法 -->
<property name="carPrint" value="#{car.print()}" />
```

它可用于执行任何标准的数学、逻辑或关系操作：

```
<!-- 3 -->
<property name="num" value="#{2+1}" />
<!-- 1 -->
<property name="num" value="#{2-1}" />
<!-- 4 -->
<property name="num" value="#{2*2}" />
<!-- 3 -->
<property name="num" value="#{9/3}" />
```

它可以用来执行条件检查三元运算符：

```
<!-- 真 -->
<property name="numStr" value="#{(10>3)?'真':'假'}" />
```

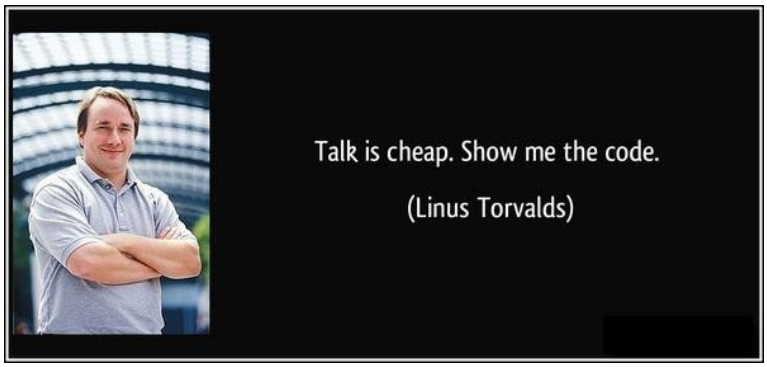
它可用于获取映射或列表的元素：

```
<property name="mapA" value="#{testBean.map['MapA']}"/>
<property name="list" value="#{testBean.list[0]}"/>
```

> **Tips**：SpEL 并不依附于 Spring 容器，它也可以独立于容器解析。因此，我们在书写自己的逻辑、框架的时候，也可以借助 SpEL 定义支持一些高级表达式 。

SpEL 深入探秘

想要看看 SpEL 是如何运作的吗？



那就来一个最简单的示例程序：

```
package com.davidwang456.test;

import org.springframework.expression.Expression;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;

public class ExpressionTest {
    public static void main(String[] args) {
        ExpressionParser parser = new SpelExpressionParser();
        Expression exp = parser.parseExpression("new String('hello world').toUpperCase()");
        String message = exp.getValue(String.class);
        System.out.println(message);
    }
}
```

运行结果为：

> HELLO WORLD

是不是比较神奇？

程序也简单到作为 java 程序员都懂的地步：

第一步：创建一个表达式解析器 ExpressionParser，调用解析器的 parseExpression 方法解析 String 的表达式；

第二步：根据表达式，计算最终结果。

这个就涉及到一个计算机的概念，抽象语法树（Abstract Syntax Tree，AST），或简称语法树（Syntax tree），是源代码语法结构的一种抽象表示。它以树状的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构。

要想生成 AST，就需要分词和语法分析。Spring 提供了分词器和语法分析器：

分词器 Tokenizer

通过 debug 上述的程序，可以看到分词后的结果：

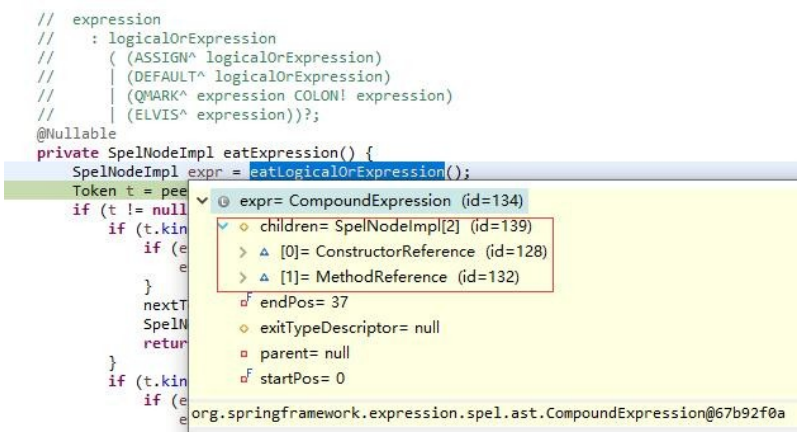
```
[[IDENTIFIER:new](0,3), [IDENTIFIER:String](4,10), [LPAREN(()](10,11), [LITERAL_STRING:'hello world'](11,24), [RPAREN())](24,25), [DOT(.)](25,26), [IDENTIFIER:toUpperCase](26,37), [LPAREN(()](37,38), [RPAREN())](38,39)]
```

分词的词根在 TokenKind.java 中，定义了一个枚举来定义所有的分词，定义多达 95 中词根：

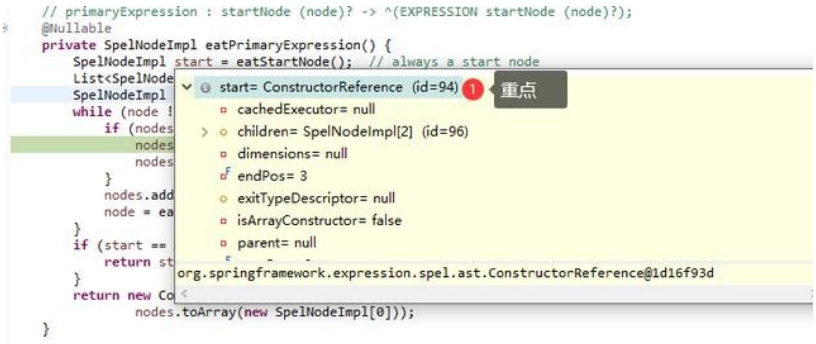
```
enum TokenKind {
    // ordered by priority - operands first
    LITERAL_INT,
    LITERAL_LONG,
    LITERAL_HEXINT,
    LITERAL_HEXLONG,
    LITERAL_STRING,
    LITERAL_REAL,
    LITERAL_REAL_FLOAT,
    LPAREN("("),
    RPAREN(")"),
    COMMA(","),
    IDENTIFIER,
    COLON(":"),
    HASH("#"),
    RSQUARE("]"),
    LSQUARE("["),
    LCURLY("{"),
    RCURLY("}"),
    DOT("."),
    PLUS("+"),
    STAR("*"),
    MINUS("-"),
    SELECT_FIRST("^["),
    SELECT_LAST("$["),
```

语法分析器 InternalSpelExpressionParser

进一步分析的结果是最终生成节点 SpelNodeImpl 树的实现，本文中生成两个节点实现，一个 ConstructorReference 和一个 MethodReference。



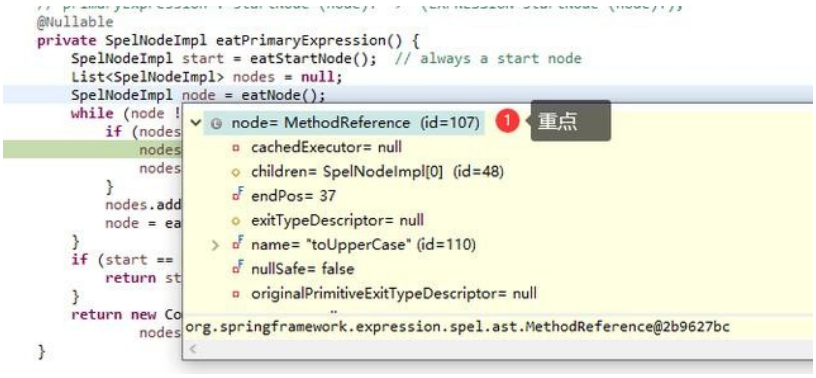
生成代码树的代码在语法分析器 InternalSpelExpressionParser 的 eatPrimaryExpression 方法



其中 ConstructorReference 的生成代码如下：

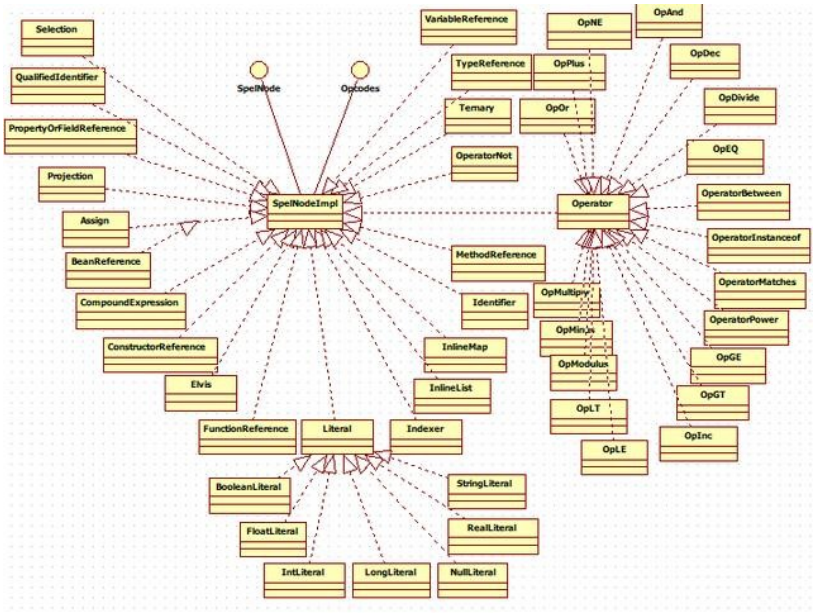
```
private boolean maybeEatConstructorReference() {
    if (peekIdentifierToken("new")) {
        Token newToken = takeToken();
        // It looks like a constructor reference but is NEW being used as a map key?
        if (peekToken(TokenKind.RSQUARE)) {
            // looks like 'NEW]' (so NEW used as map key)
            push(new PropertyOrFieldReference(false, newToken.stringValue(), newToken.startPos, newToken.endPos));
            return true;
        }
        SpelNodeImpl possiblyQualifiedConstructorName = eatPossiblyQualifiedId();
        List<SpelNodeImpl> nodes = new ArrayList<>();
        nodes.add(possiblyQualifiedConstructorName);
        if (peekToken(TokenKind.LSQUARE)) {
            // array initializer
            List<SpelNodeImpl> dimensions = new ArrayList<>();
            while (peekToken(TokenKind.LSQUARE, true)) {
                if (!peekToken(TokenKind.RSQUARE)) {
                    dimensions.add(eatExpression());
                }
                else {
                    dimensions.add(null);
                }
                eatToken(TokenKind.RSQUARE);
            }
            if (maybeEatInlineListOrMap()) {
                nodes.add(pop());
            }
            push(new ConstructorReference(newToken.startPos, newToken.endPos,
                    dimensions.toArray(new SpelNodeImpl[0]), nodes.toArray(new SpelNodeImpl[0])));
        }
        else {
            // regular constructor invocation
            eatConstructorArgs(nodes);
            // TODO correct end position?
            push(new ConstructorReference(newToken.startPos, newToken.endPos, nodes.toArray(new SpelNodeImpl[0])));
        }
        return true;
    }
    return false;
}
```

MethodReference 的生成代码如下：



那么除了上面简单示例中提到的 SpelNodeImpl 节点实现外，还有其他哪些节点 SpelNodeImpl 实现呢？

节点 SpelNodeImpl 树的实现类：



根据不同的节点类型，生成不同的节点过程如下：

```
@Nullable
private SpelNodeImpl eatStartNode() {
    if (maybeEatLiteral()) {
        return pop();
    }
    else if (maybeEatParenExpression()) {
        return pop();
    }
    else if (maybeEatTypeReference() || maybeEatNullReference() || maybeEatConstructorReference() ||
            maybeEatMethodOrProperty(false) || maybeEatFunctionOrVar()) {
        return pop();
    }
    else if (maybeEatBeanReference()) {
        return pop();
    }
    else if (maybeEatProjection(false) || maybeEatSelection(false) || maybeEatIndexer()) {
        return pop();
    }
    else if (maybeEatInlineListOrMap()) {
        return pop();
    }
    else {
        return null;
    }
}
```

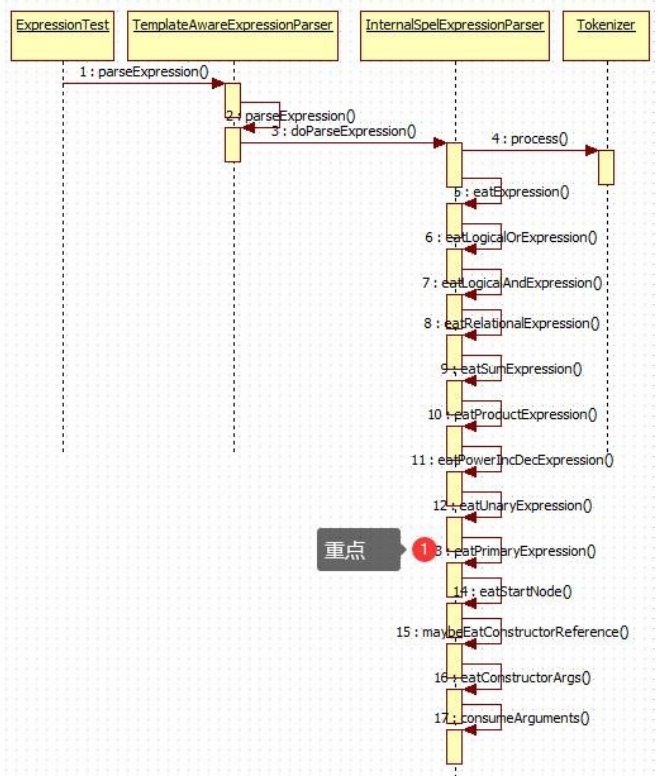
表达式计算

表达式计算一般通过调用 SpelNodeImpl 实现类的 getValueInternal 方法。

SpEL 运行流程

通过深入 debug 上述示例，可以程序的整个流程。它分成三个部分：

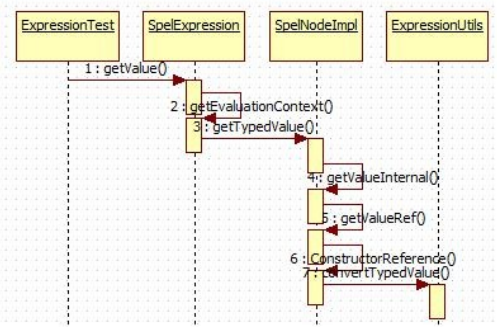
- 节点树生成



重点代码在 eatPrimaryExpression 中：

```
// primaryExpression : startNode (node)? -> ^(EXPRESSION startNode (node)?);
@Nullable
private SpelNodeImpl eatPrimaryExpression() {
    SpelNodeImpl start = eatStartNode();  // always a start node
    List<SpelNodeImpl> nodes = null;
    SpelNodeImpl node = eatNode();
    while (node != null) {
        if (nodes == null) {
            nodes = new ArrayList<>(4);
            nodes.add(start);
        }
        nodes.add(node);
        node = eatNode();
    }
    if (start == null || nodes == null) {
        return start;
    }
    return new CompoundExpression(start.getStartPosition(), nodes.get(nodes.size() - 1).getEndPosition(),
            nodes.toArray(new SpelNodeImpl[0]));
}
```

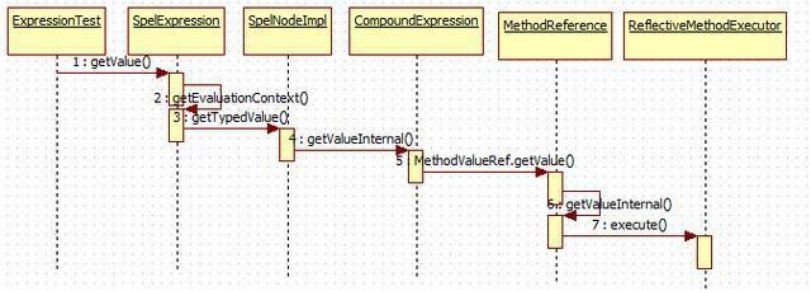
- **Constructor** 表达式计算



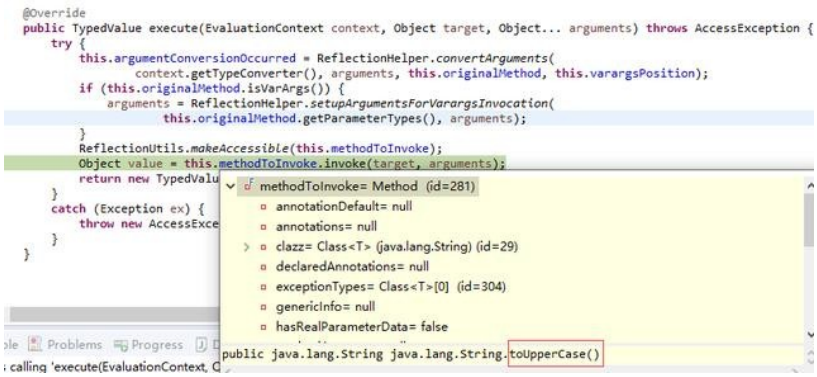
重点代码：

```
/**
 * Evaluates a compound expression. This involves evaluating each piece in turn and the
 * return value from each piece is the active context object for the subsequent piece.
 * @param state the state in which the expression is being evaluated
 * @return the final value from the last piece of the compound expression
 */
@Override
public TypedValue getValueInternal(ExpressionState state) throws EvaluationException {
    ValueRef ref = getValueRef(state);
    TypedValue result = ref.getValue();
    this.exitTypeDescriptor = this.children[this.children.length - 1].exitTypeDescriptor;
    return result;
}
```

- **Method** 表达式计算



执行方法 ReflectiveMethodExecutor.java#execute



总结

Spring3.x 引入的 SpEL 可谓非常的惊艳，它的实现非常的复杂，但它的使用却异常的简单和灵活。它给 Spring 外部化配置注入了更多的活力，它让我们在运行时赋值、改变值都轻松的成为了可能~

- 分词器 **Tokenizer**：将 String 转化为语法分析器可以识别的词；
- 语法分析器 **SpelExpressionParser**：将可分析的词组成 ast 树；
- 计算部分 **SpelNodeImpl**：的实现类中通过 getValueInternal 获取。

}

32 数据转换总结及常见面试题目解析

34 SpEL List和Map 引用应用示例及背后原理探究