1. 概述

1、什么是SpringEL

Spring3中引入了Spring表达式语言—SpringEL,SpEL是一种强大,简洁的装配Bean的方式,他可以通过运行期间执行的表达式将值装配到我们的属性或构造函数当中,更可以调用JDK中提供的静态常量,获取外部Properties文件中的的配置

2.、什么要使用SpringEL

我们平常通过配置文件或Annotaton注入的Bean,其实都可以称为静态性注入,试想一下,若然我Bean A中有变量A,它的值需要根据Bean B的B变量为参考,在这场景下静态注入就对这样的处理显得非常无力,而Spring3增加的SpringEL就可以完全满足这种需求,而且还可以对不同Bean的字段进行计算再进行赋值,功能非常强大

3、如何使用SpringEL

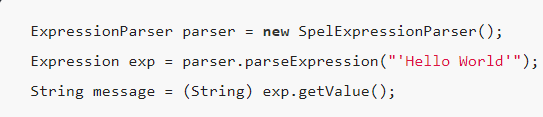
SpringEL从名字来看就能看出,和EL是有点关系的,SpringEL的使用和EL表达式的使用非常相似,EL表达式在JSP页面更方便的获取后台中的值,而SpringEL就是为了更方便获取Spring容器中的Bean的值,EL使用${},而SpringEL使用#{}进行表达式的声明

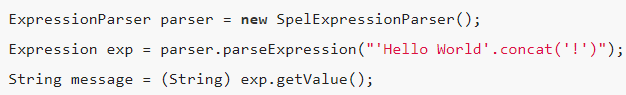
4、官方文档

<https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#expressions>

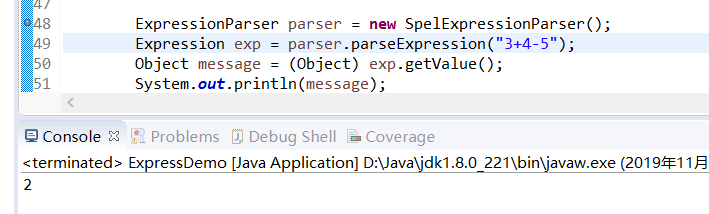
1. 使用场景
2. 求值

（1）求取字符串相关的值

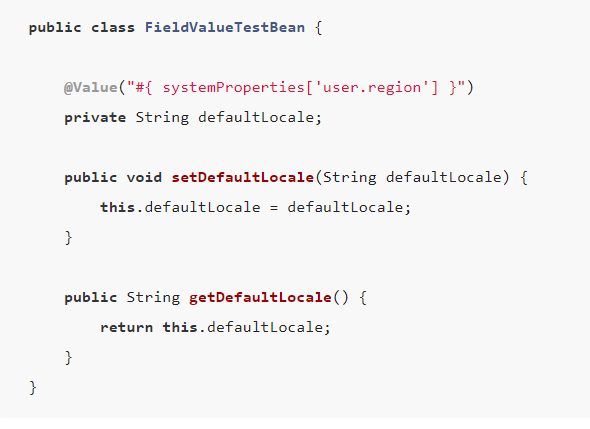




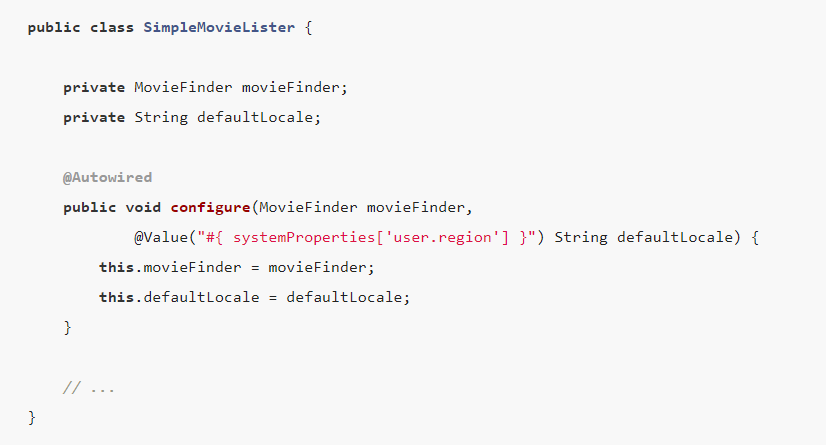
2.计算表达式



3.为bean设置属性值



4.为bean设置方法默认值



1. SpringEL表达式使用详情

1、自定义表达式使用

StandardEvaluationContext ctx = new StandardEvaluationContext();  
  
Class<ExpressDemo> cl = ExpressDemo.class;  
Method[] methods = cl.getDeclaredMethods();//获取类中的方法列表  
  
ctx.setVariable("a", 5);  
ctx.setVariable("b", 7);  
//将方法名称跟方法组合成一个MAP  
Map<String, Method> methodMap = Arrays.*stream*(methods).collect(Collectors.*toMap*(Method::getName, Function.*identity*()));  
//将map中的方法注册到上下文中  
methodMap.forEach((key,value)->{ctx.registerFunction(key, value);});  
ExpressionParser parser = new SpelExpressionParser();  
//执行表达式  
Object expression = parser.parseExpression("#sum(10,#sum(#a,#b))").getValue(ctx);  
System.*out*.println(expression);

其中

sum法为ExpressDemo类中的一个方法：

public static int sum(int a,int b) {  
 return a+b;  
}

SpringEL就可以根据表达式试去执行ExpressDemo类中的方法。我们可以自己定义我们的方法，然后借助SpringEl去执行自定义的方法

2、变量的定义

定义表达式只需要向上下文中增加变量值即可，如：

ctx.setVariable("a", 5);

其中ctx为上文中定义的StandardEvaluationContext上下文

3、方法的定义

定义表达式只需要向上下文中注册方法即可，如:

ctx.registerFunction(key, value);

其中key为方法名称，value为mehtod类型的方法

4、

四、SringEL的执行原理

（一）部分重要的类

1、EvaluationContext

为SpringEL引擎的上下文，SpringEL执行表达式遇见方法或者变量的时候会去这个上下文中寻找。有两种实现方式：

（1）SimpleEvaluationContext ——只包含SpEL的部分配置项，适用于不严格按照SpEL规范，包括数据绑定表达式，基于属性过滤的操作等等。

(2) StandardEvaluationContext    ——可以设置SpEL的所有配置，可以指定默认的基本对象，并执行与计算相关的所有操作。

2、ExpressionParser

解析表达式的接口，将字符型的表达式解析为SpringEL封装的Expression类型的表达式，实现：

（1）TemplateAwareExpressionParser 是个抽象方法，封装基本的解析过程

（2）SpelExpressionParser 常用的实现类，最终会委托InternalSpelExpressionParser类进行解析

（3）InternalSpelExpressionParser真正的解析类

2、Expression

表达式对象。能够根据上下文对象对自身进行计算的表达式。封装以前分析的表达式字符串的详细信息。实现类为SpelExpression

（二）解析语法

解析器最终都会调用InternalSpelExpressionParser作为真正的解析器

1、解析为token串

解析器会首先将表达式转换为一个char数组：

public Tokenizer(String inputData) {  
 this.expressionString = inputData;  
 this.charsToProcess = (inputData + "\0").toCharArray();  
 this.max = this.charsToProcess.length;  
 this.pos = 0;  
}

然后封装成一个Tokenizer对象。Tokenizer对象会将刚才的数组字符逐个解析，生成不同的Token对象，其中Token对象的结构为：

class Token {  
  
 TokenKind kind;  
  
 @Nullable  
 String data;  
  
 int startPos; // index of first character  
  
 int endPos; // index of char after the last character

data封装的数据，TokenKind为一个枚举。这是解析后的主要对象。下图是部分枚举对象：

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*("["),

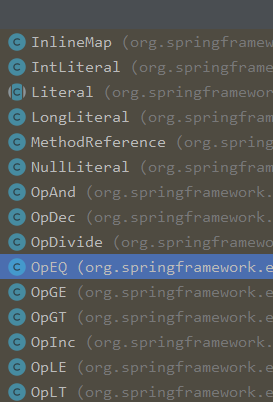
表达式的不同部分会被解析成不同的枚举TokenKind，然后封装成一个Token.最终生成一个Token的集合。封装到解析器中。

2.生成语法树

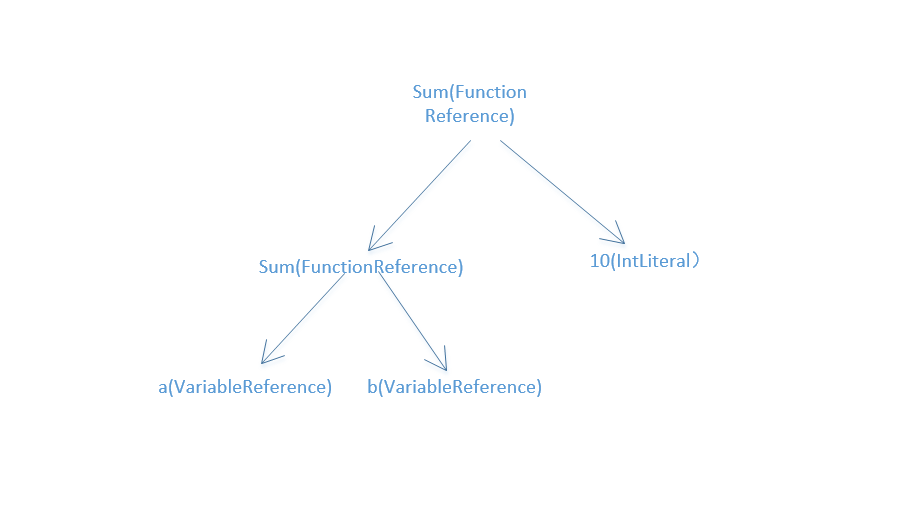
解析器将上述生成的token对象集合逐个解析，根据一定的语法规则生成对应的SpelNodeImpl节点树。SpelNodeImpl的结构如下：

public abstract class SpelNodeImpl implements SpelNode, Opcodes {  
  
 private static final SpelNodeImpl[] *NO\_CHILDREN* = new SpelNodeImpl[0];  
  
  
 private final int startPos;  
  
 private final int endPos;  
  
 protected SpelNodeImpl[] children = SpelNodeImpl.*NO\_CHILDREN*;  
  
 @Nullable  
 private SpelNodeImpl parent;

这个节点数有不同的实现类，分别对象不同的解析对象，部分实现类如下：



解析器会根据不同的Token对象以及语义生成不同的节点，并且形成树。如示例中的方法最终会生成如下的语法树：



最后将生产的语法树封装成SpelExpression对象，完成表达式解析。

（三）公式计算

1、环境准备

解析器会将上下文再次封装为一个ExpressionState，然后委托给语法树的执行器去执行，代码如下：

ExpressionState expressionState = new ExpressionState(context, this.configuration);  
Object result = this.**ast**.getValue(expressionState);  
checkCompile(expressionState);

注意有下划线的ast对象，这个对象类型为SpelNodeImpl。它就是上面解析完了的语法树对象，即语法树的根节点对象。

getValue方法是属于SpelNodeImpl的方法，是一个模板方法，如下：

public final Object getValue(ExpressionState expressionState) throws EvaluationException {  
 return getValueInternal(expressionState).getValue();  
}

这个getValueInternal是一个抽象方法，也就是每个SpelNodeImpl节点的实现都得实现这个方法。计算的具体代码就是在这里实现的。如下：

public abstract TypedValue getValueInternal(ExpressionState expressionState) throws EvaluationException

最终其实是调用不同节点的getValueInternal方法，计算出结果。

2、FunctionReference的计算（#sum（） 这种带“#“的函数会解析成这个）

示例中的方法，通过上述的代码最终会执行到根节点的名称为sum的FunctionReference类型的节点的getValueInternal方法。这个方法部分代码如下：

TypedValue value = state.lookupVariable(this.name);  
if (value == TypedValue.*NULL*) {  
 throw new SpelEvaluationException(getStartPosition(), SpelMessage.*FUNCTION\_NOT\_DEFINED*, this.name);  
}  
if (!(value.getValue() instanceof Method)) {  
 // Possibly a static Java method registered as a function  
 throw new SpelEvaluationException(  
 SpelMessage.*FUNCTION\_REFERENCE\_CANNOT\_BE\_INVOKED*, this.name, value.getClass());  
}  
  
try {  
 return executeFunctionJLRMethod(state, (Method) value.getValue());

首先会根据名称（这里为sum）去上下文中的变量中去寻找真正的方法。因为已经在开始注册了这个方法，所以这里寻找到真正的Method对象的方法。

然后执行executeFunctionJLRMethod方法。在这个方法去寻找变量。然后通过反射执行方法部分代码如下（中间有些判断以及处理，这里忽略）：

Object[] functionArgs = getArguments(state);

ReflectionUtils.*makeAccessible*(method);  
Object result = method.invoke(method.getClass(), functionArgs);  
compilable = !argumentConversionOccurred;  
return new TypedValue(result, new TypeDescriptor(new MethodParameter(method, -1)).narrow(result));

其中这个获取参数的方法是寻找自己的子节点，然后调用子节点的getValueInternal计算出真正的值（碰见函数就依次往下递归），代码如下：

private Object[] getArguments(ExpressionState state) throws EvaluationException {  
 // Compute arguments to the function  
 Object[] arguments = new Object[getChildCount()];  
 for (int i = 0; i < arguments.length; i++) {  
 arguments[i] = this.children[i].getValueInternal(state).getValue();  
 }  
 return arguments;  
}

3、MethodReference的计算（sum() 这种直接写函数名的会被解析成这个节点）

这个节点的信息会从当前活动的上下文获取，也就是上下文中的rootObject对象。如下：

EvaluationContext evaluationContext = state.getEvaluationContext();  
Object value = state.getActiveContextObject().getValue();  
TypeDescriptor targetType = state.getActiveContextObject().getTypeDescriptor();

public TypedValue getActiveContextObject() {  
 if (CollectionUtils.*isEmpty*(this.contextObjects)) {  
 return this.rootObject;  
 }  
 return this.contextObjects.element();  
}

这里获取参数的方式跟FunctionReference一样，都是调用语法树子节点的getValueInternal，里面可能存在递归调用。

获得参数后，根据参数列表，获取各个参数的类型

List<TypeDescriptor> argumentTypes = getArgumentTypes(arguments);

然后根据参数类型，rootObject以及rootObject对象的类型，还有上下文去寻找方法执行器（会先从缓存中获取）：

executorToUse = findAccessorForMethod(argumentTypes, value, evaluationContext);

寻找方法执行器会从注册的MethodResolver去寻找：

private MethodExecutor findAccessorForMethod(List<TypeDescriptor> argumentTypes, Object targetObject,  
 EvaluationContext evaluationContext) throws SpelEvaluationException {  
  
 AccessException accessException = null;  
 List<MethodResolver> methodResolvers = evaluationContext.getMethodResolvers();  
 for (MethodResolver methodResolver : methodResolvers) {  
 try {  
 MethodExecutor methodExecutor = methodResolver.resolve(  
 evaluationContext, targetObject, this.name, argumentTypes);  
 if (methodExecutor != null) {  
 return methodExecutor;  
 }  
 }  
 catch (AccessException ex) {  
 accessException = ex;  
 break;  
 }  
 }

MethodResolver会具体去解析这个方法变量所代表的方法。上下文会默认注册一个ReflectiveMethodResolver方法加载器，会通过反射加载对应的方法。也可以自定义方法加载器去加载方法，然后注册到上下文中。注册的方法是调用StandardEvaluationContext. addMethodResolver方法去注册。反射的加载方法就是去反射这个rootObject，检查有没有能匹配得方法，有就包装成MethodExecutor返回，没有就返回空。

如果遍历完所有的MethodResolver都没有找到对应的Method则抛出异常。

最后如果找到了MethodExecutor会去执行里面的execute,通过反射真正的方法，执行方法后返回结果：

public TypedValue execute(EvaluationContext context, Object target, Object... arguments) throws AccessException {  
 try {  
 this.argumentConversionOccurred = ReflectionHelper.*convertArguments*(  
 context.getTypeConverter(), arguments, this.originalMethod, this.varargsPosition);  
 if (this.originalMethod.isVarArgs()) {  
 arguments = ReflectionHelper.*setupArgumentsForVarargsInvocation*(  
 this.originalMethod.getParameterTypes(), arguments);  
 }  
 ReflectionUtils.*makeAccessible*(this.methodToInvoke);  
 Object value = this.methodToInvoke.invoke(target, arguments);  
 return new TypedValue(value, new TypeDescriptor(new MethodParameter(this.originalMethod, -1)).narrow(value));  
 }  
 catch (Exception ex) {  
 throw new AccessException("Problem invoking method: " + this.methodToInvoke, ex);  
 }  
}

4、PropertyOrFieldReference获取值（“sum(a,b)”里面的a,b这种不带#号会解析成这种节点）

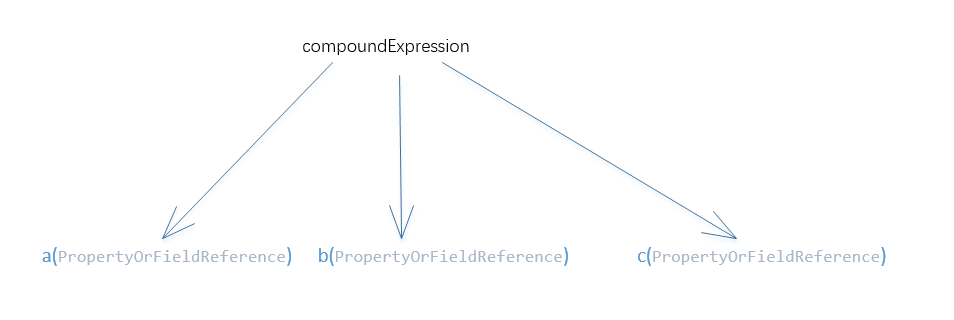
这个节点的解析跟MethodReference节点差不多，是通过rootObject以及名称属性去注册的PropertyAccessor列表中去获取值。

for (PropertyAccessor accessor : accessorsToTry) {  
 if (accessor.canRead(evalContext, contextObject.getValue(), name)) {  
 if (accessor instanceof ReflectivePropertyAccessor) {  
 accessor = ((ReflectivePropertyAccessor) accessor).createOptimalAccessor(  
 evalContext, contextObject.getValue(), name);  
 }  
 this.cachedReadAccessor = accessor;  
 return accessor.read(evalContext, contextObject.getValue(), name);  
 }  
}

也可以注册自定义属性加载器，去加载对应的属性。注册的方法为ExtensionResolverAccessor. addPropertyAccessor,去加载对应的方法

5、CompoundExpression复杂对象的计算（a.b这种解析）

复杂对象会被解析为CompoundExpression节点，这个节点本身不存储数据，所有数据会存储到它的子节点下，比如a.b.c这种结构，会被解析成以下的结构：



然后从左往右以此解析，将上一个解析完的结果作为下一个解析的rootObject，然后从外往里解析出最终的值：

protected ValueRef getValueRef(ExpressionState state) throws EvaluationException {  
 if (getChildCount() == 1) {  
 return this.children[0].getValueRef(state);  
 }  
  
 SpelNodeImpl nextNode = this.children[0];  
 try {  
 TypedValue result = nextNode.getValueInternal(state);  
 int cc = getChildCount();  
 for (int i = 1; i < cc - 1; i++) {  
 try {  
 state.pushActiveContextObject(result);  
 nextNode = this.children[i];  
 result = nextNode.getValueInternal(state);  
 }  
 finally {  
 state.popActiveContextObject();  
 }  
 }  
 try {  
 state.pushActiveContextObject(result);  
 nextNode = this.children[cc - 1];  
 return nextNode.getValueRef(state);  
 }

最终解析出结果来。

四、最终实现

有了上述的几个节点后，就可以自己定义我们的方法加载器，属性加载去，然后就按照正常的公式计算即可（不用加一堆”#”号，或者其他符号去标识谁是变量谁是常量等）。示例如下：

public class OptimizeFormulaExpressDemo {  
 //定义的函数  
 public static int sum(Integer a, Integer b) {  
 return a + b;  
 }  
  
 public static void main(String[] args) throws NoSuchMethodException {  
  
 StandardEvaluationContext ctx = new StandardEvaluationContext();  
  
  
 Map<String, Object> map = new HashMap<>();  
 //创建方法加载器  
 MethodResolver r = new MethodResolver() {  
 @Override  
 public MethodExecutor resolve(EvaluationContext context, Object targetObject, String name, List<TypeDescriptor> argumentTypes) throws AccessException {  
  
 if (targetObject instanceof Map) {//如果当前的rootObject是map，则直接从Map里面取  
 Map<String, Object> map = (Map<String, Object>) targetObject;  
 Object o = map.get(name);  
 if (o != null && o instanceof Method) {//找到对应的方法  
 return new ReflectiveMethodExecutor((Method) o);  
  
 }  
  
  
 }  
  
 return null;  
 }  
 };  
 //属性解析器因为是把map作为了rootObject，所以这里直接用map属性解析器  
 MapAccessor mapAccessor = new MapAccessor();  
  
 Method sumMethod = OptimizeFormulaExpressDemo.class.getMethod("sum", Integer.class, Integer.class);  
 map.put("sum", sumMethod);  
 map.put("a", 10);  
 map.put("b", 20);  
 ctx.setRootObject(map);  
 ctx.addMethodResolver(r);  
 ctx.addPropertyAccessor(mapAccessor);  
  
 ExpressionParser parser = new SpelExpressionParser();  
 //执行表达式  
 Object expression = parser.parseExpression("sum(a,sum(a,b))").getValue(ctx);  
 System.*out*.println(expression);  
  
  
 }  
}