案例：

<https://gitee.com/zx19890628/spring-boot-example/tree/master/lab_068_util_math>

# 数据公式解析

一些数学计算公式，想提前了解一些这方面的东西，但不知该如何设计用于保存公式的数据结构。在网上搜索了一番，找到几个自认为还不错的API，通过简单的Demo，对自己的数据结构设计也有了点眉目，先集中收录下来。

# Jep

比较老的工具包，最后一次更新是在2005年，但是常见的数据计算都是没有问题的

首先，在使用jep之前先引入需要的jar

<dependency>

<groupId>jep</groupId>

<artifactId>jep</artifactId>

<version>2.24</version>

</dependency>

2：测试jep功能代码如下

public class JepTest {

public static void main(String[] args) {

JEP jep = new JEP();

// 设置公式

String expression = "100/a";

// 给变量赋值

jep.addVariable("a",3.0);

// 运算

jep.parseExpression(expression);

// 得出结果

System.out.println(jep.getValue());

}

}

jep.getValue()此方法的返回值为Double 类型的！

3：将变化的公式以及所有需要的参数存入到数据的字段中，无论算法怎么变，公式也就跟这边，我们只需要将参数赋予它对应的值就能计算出结果！

# Fast EL

## 介绍

Fel是轻量级的高效的表达式计算引擎（国产的工具）。

Fel在源自于企业项目，设计目标是为了满足不断变化的功能需求和性能需求。

Fel是开放的，引擎执行中的多个模块都可以扩展或替换。Fel的执行主要是通过函数实现,运算符(+、-等都是Fel函数），所有这些函数都是可以替换的，扩展函数也非常简单。

Fel有双引擎，同时支持解释执行和编译执行。可以根据性能要求选择执行方式。编译执行就是将表达式编译成字节码（生成java代码和编译模块都是可以扩展和替换的）

Fel基于Java1.5开发，适用于Java1.5及以上版本。

项目地址：<http://code.google.com/p/fast-el/>（推荐看这个）

下 载：<http://fast-el.googlecode.com/files/fel-all-0.7.jar>

文 档：<http://fast-el.googlecode.com/files/Fel%E8%A1%A8%E8%BE%BE%E5%BC%8F%E5%BC%95%E6%93%8E.doc>

性能测试：<http://code.google.com/p/fast-el/wiki/Performance>

## 特点

易用性：API使用简单，语法简洁，和java语法很相似。

轻量级：整个包只有300多KB。

高 效：目前没有发现有开源的表达式引擎比Fel快。

扩展性：采用模块化设计，可灵活控制表达式的执行过程。

根函数：Fel支持根函数，“$('Math')”在Fel中是常用的使用函数的方式。

$函数：通过$函数，Fel可以方便的调用工具类或对象的方法（并不需要任何附加代码），具体请看示例。

## 不足

支持脚本：否。

## 适应场景

Fel适合处理海量数据，Fel良好的扩展性可以更好的帮助用户处理数据。

Fel同样适用于其他需要使用表达式的地方（如果工作流、公式计算、数据有效性校验等等）

## 此版本改进

1：支持数组访问（array[i])和集合访问（list[i],set[i]等）。并支持多维数组及多维List。

2：设置变量时可以指定变量类型（FelContext.setVar(..))，同时调整FelContext接口。

3：四则运算返回的结果类型更精确。

4：性能再度提升。

5：解决求负求正运算在编译时返回类型不正确的BUG。

## 安全（始于0.8版本）

为了防止出现“${'System'}.exit(1)”这样的表达式导致系统崩溃。Fel加入了安全管理器，主要是对方法访问进行控制。安全管理器中通过允许访问的方法列表（白名单）和禁止访问的方法列表（黑名单）来控制方法访问。将"java.lang.System. \*"加入到黑名单，表示System类的所有方法都不能访问。将"java.lang.Math. \*"加入白名单，表示只能访问Math类中的方法。如果你不喜欢这个安全管理器，可以自己开发一个，非常简单，只需要实现一个方法就可以了。

## 简单使用

目前不知道为什么，知道在maven中找到一个版本，2013年最后一个发布版本

<dependency>

<groupId>org.eweb4j</groupId>

<artifactId>fel</artifactId>

<version>0.8</version>

</dependency>

// 1. 解析表达式

@Test

public void test1() {

FelEngine f1 = new FelEngineImpl();

Object r1 = f1.eval("5000\*12+7500");

System.out.println(r1);

System.out.println(r1.getClass());

}

// 2. 设置变量

@Test

public void test2() {

FelEngine f2 = new FelEngineImpl();

FelContext ctx = f2.getContext();

ctx.set("单价", 5000);

ctx.set("数量", 12);

ctx.set("运费", 7500);

Object r2 = f2.eval("单价\*数量+运费");

System.out.println(r2);

}

// 3.访问对象属性

@Test

public void test3() {

FelEngine f3 = new FelEngineImpl();

FelContext ctx3 = f3.getContext();

Foo foo = new Foo();

// 保存对象1

ctx3.set("foo", foo);

// 保存对象2

Map m = new HashMap();

m.put("ElName", "fel");

ctx3.set("m",m);

//调用foo.getSize()方法。

Object result = f3.eval("foo.size");

System.out.println(result);

//调用foo.isSample()方法。

result = f3.eval("foo.sample");

System.out.println(result);

//foo没有name、getName、isName方法

//foo.name会调用foo.get("name")方法。

result = f3.eval("foo.name");

System.out.println(result);

//m.ElName会调用m.get("ElName");

result = f3.eval("m.ElName");

System.out.println(result);

}

// 4.访问数组、集合、Map

@Test

public void test4() {

FelEngine fel = new FelEngineImpl();

FelContext ctx = fel.getContext();

//1.数组

int[] intArray = {1,2,3};

ctx.set("intArray",intArray);

//获取intArray[0]

String exp = "intArray[0]";

System.out.println(exp+"->"+fel.eval(exp));

//2.lis

List li = Arrays.asList(1,2,3);

ctx.set("li", li);

exp = "li[1]";

System.out.println(exp+"->"+fel.eval(exp));

//3.collection

Collection<String> col = Arrays.asList("A", "B", "c");

ctx.set("col", col);

exp = "col[2]";

System.out.println(exp+"->"+fel.eval(exp));

//4.迭代器

Iterator<String> iterator = col.iterator();

ctx.set("iterator", iterator);

exp = "iterator[1]";

System.out.println(exp+"->"+fel.eval(exp));

//5.map

Map<String, String> m = new HashMap();

m.put("k1", "v1");

ctx.set("m", m);

exp = "m.k1";

System.out.println(exp+"->"+fel.eval(exp));

//6.多维数组

int[][] arrs= {{11,12},{21,22}};

ctx.set("arrs", arrs);

exp = "arrs[0]";

System.out.println(exp+"->"+fel.eval(exp));

exp = "arrs[1][1]";

System.out.println(exp+"->"+fel.eval(exp));

}

// 5.调用JAVA方法

@Test

public void test5() {

FelEngine fel = new FelEngineImpl();

FelContext ctx = fel.getContext();

ctx.set("out", System.out);

fel.eval("out.println('Hello Everybody'.substring(6))");

}

//6:自定义上下文环境

@Test

public void test6() {

//负责提供气象服务的上下文环境

FelContext ctx = new AbstractContext() {

@Override

public Object get(String name) {

if("天气".equals(name)){

return "晴";

}

if("温度".equals(name)){

return 25;

}

return null;

}

};

FelEngine fel = new FelEngineImpl(ctx);

Object eval = fel.eval("'天气:'+ 天气 +';温度:' + 温度");

System.out.println(eval);

//输出结果：天气:晴;温度:25

}

//7:多层上下文环境（命名空间）

@Test

public void test7() {

FelEngine fel = new FelEngineImpl();

FelContext baseCtx = fel.getContext();

String costStr = "成本";

String priceStr="价格";

//父级上下文中设置成本和价格

baseCtx.set(costStr, 50);

baseCtx.set(priceStr,100);

String exp = priceStr+"-"+costStr;

Object baseCost = fel.eval(exp);

System.out.println("期望利润：" + baseCost);

FelContext ctx = new ContextChain(baseCtx, new MapContext());

//通货膨胀导致成本增加（子级上下文 中设置成本，会覆盖父级上下文中的成本）

ctx.set(costStr, 50+20);

Object allCost = fel.eval(exp, ctx);

System.out.println("实际利润：" + allCost);

}

//8:编译执行(备注：适合处理海量数据，编译执行的速度基本与Java字节码执行速度一样快)

@Test

public void test8() {

FelEngine fel = new FelEngineImpl();

FelContext ctx = fel.getContext();

ctx.set("单价", 5000);

ctx.set("数量", 12);

ctx.set("运费", 7500);

Expression exp = fel.compile("单价\*数量+运费",ctx);

Object result = exp.eval(ctx);

System.out.println(result);

}

//9:自定义函数

@Test

public void test9() {

Function fun = new CommonFunction() {

public String getName() {

return "hello";

}

/\*

\* 调用hello("xxx")时执行的代码

\*/

@Override

public Object call(Object[] arguments) {

Object msg = null;

if(arguments!= null && arguments.length>0){

msg = arguments[0];

}

return ObjectUtils.toString(msg);

}

};

FelEngine e = new FelEngineImpl();

//添加函数到引擎中。

e.addFun(fun);

String exp = "hello('fel')";

//解释执行

Object eval = e.eval(exp);

System.out.println("hello "+eval);

//编译执行

Expression compile = e.compile(exp, null);

eval = compile.eval(null);

System.out.println("hello "+eval);

}

//10:调用静态方法

// 如果你觉得上面的自定义函数也麻烦，Fel提供的$函数可以方便的调用工具类的方法 熟悉jQuery的朋友肯定知道"$"函数的威力。

// Fel东施效颦，也实现了一个"$"函数,其作用是获取class和创建对象。结合点操作符，可以轻易的调用工具类或对象的方法。

// 通过"$('class').method"形式的语法，就可以调用任何等三方类包（commons lang等）及自定义工具类的方法，

// 也可以创建对象，调用对象的方法。如果有需要，还可以直接注册Java Method到函数管理器中。

@Test

public void test10() {

//调用Math.min(1,2)

Object eval = FelEngine.instance.eval("$('Math').min(1,2)");

System.out.println(eval);

//调用new Foo().toString();

eval = FelEngine.instance.eval("$('com.zx.fel.Foo.new').toString()");

System.out.println(eval);

}

//11:大数值计算（始于0.9版本）

@Test

public void test11() {

// Fel发布后，有些网友希望提供大数值计算功能，于是，大数值计算功能就有了。

FelEngine fel = FelBuilder.bigNumberEngine();

String input = "111111111111111111111111111111+22222222222222222222222222222222";

Object value = fel.eval(input);

Object compileValue = fel.compile(input, fel.getContext()).eval(fel.getContext());

System.out.println("大数值计算（解释执行）:" + value);

System.out.println("大数值计算（编译执行）:" + compileValue);

//由上例子可以看出，大数值计算引擎和常规计算引擎在使用方法是相同的。

//如果表达式数值比较大，要求精度高，可使用大数值计算引擎。不足之处是效率没有常规计算引擎高。

}

# jeval

http://www.oschina.net/p/jeval（介绍）

http://sourceforge.net/projects/jeval/files/（下载地址）

# ik-expression(国产)

https://code.google.com/p/ik-expression/

# ExpressionJ

http://sourceforge.net/projects/expressionj/

# expression-analyzer(国产)

http://www.oschina.net/p/expression-analyzer

# BeanShell

http://www.beanshell.org/download.html

# Spads(国产)

此人看样子比较牛，在CSDN、OSChina、ITeye上均有发布，支持乘方^和阶乘!运算，很实用

http://blog.csdn.net/shanelooli/article/details/8142726

<http://download.csdn.net/detail/shanelooli/4726670（下载地址）>

很多网络应用中，涉及到一些内部运算的具体求值逻辑并不能够在开发阶段就定下来。这些逻辑需要随着使用，探测用户反应，不断修正。符合的情况，常见的就是网络游戏中的行动收获计算公式，等等。现 Spads 开发组推出了针对此种情况的解析公式系统，供大家使用。

此公式解析系统能够计算各种算术运算、逻辑运算和比较运算，可以连接本地函数，支持括号分级，允许逐级设置公式内临时变量，提供了分支运算符，并且支持 Json 数据格式的运算。系统编写中使用了很多设计方法，包括概念抽取、继承、多态、面向接口设计、枚举、递归、工厂等等。数据结构也使用了好用但少见的双端队列。很适合对应情况的实用以及学习 Java 的设计思路、编程方式。本程序由 Shane Loo LI 设计和编写，所有的细节都注重了运行效率，在很多细节上都制作了数倍优于爪哇（Java）系统类库的方法；由于只需要处理公式，所以语法树结构单一，也因此保证了比现有其它注入语言的代码解释器更高的执行速度。

下面来介绍一下公式解析系统的用法。

首先，公式通过 ?= 来给变量赋值。这种赋值并不是一个运算，而是公式每一行需要且只能出现一次的专属符号。

Result ?= 38 \* 6

公式解析系统用 Result 来表示最终的结果。以上公式表示最终的结果为 228 。

本系统支持以下算术运算：

求负 -

加 +

减 -

乘 \*

除 /

求余 %

乘方 ^

阶乘 !

比如

Result ?= ((2^5 - (-13)) % 10)!

能够得到 120 。

接下来，让我们看看公式系统的分支控制。

DayCount ?= 31

DayCost ?= 58.5

Limit ?= 1500

Result ?= DayCount \* DayCost > Limit ? "超支了！" ~ "没超支"

我们看到整个公式分为四行。每行声明了一个变量。

实际上在这个公式系统中，并不严格要求上边的公式不能够使用下边公式声明的变量，但如果嵌套使用则会抛出异常。

上述公式将得到一个字符串，内容是“超支了”。

支持的有关运算如下：

大于 >

小于 <

等于 =

与 &

或 |

非 -（这个和 C 语言系列的 ! 有所不同）

条件 ?

分支 ~（这个和 C 语言系列的 : 有所不同）

这个系统最重要的特性，就是能够把公式中出现的函数，和本地函数绑定起来。

只需要制作 Function 接口的实现类，并且在 FunctionEnum 对其进行注册，就能够得到一种可以在公式中识别的函数。

我最近学习了“约定优于明示”的思想，近期会将这种 FunctionEnum 配置方式进行一定程度的更改。

比如，我提供了内置函数 TOGETHER ，其含义是以第一个参数为名称，第二个参数为值，构成键值对；第三个参数为名称，第四个参数为值，构成键值对；等等。最终将其组合成 Json 字符串。请看如下公式。

Name ?= "Shane"

Level ?= 16

Result ?= TOGETHER("name", Name, "graduated", Level > 10 ? "毕业了" ~ "没毕业")

最终结果为 {"name": "Shane", "graduated": "毕业了"}

说了这么多公式解析系统的用法。现在来从编程角度介绍一下如何启用此系统功能。

首先，肯定是要导入 .jar 包。然后参看以下代码。

EvaluatorFactory factory = EvaluatorFactory.INST;

Evaluator eva = factory.getEvaluator("Result ?= 28 + 2 / 10.0");

ExpValue result = eva.evaluate();

System.out.println(result);

或者

String form = "Result ?= 28 + 2 / 10.0";

ExpValue result = EvaluatorFactory.INST.getEvaluator(form).evaluate();

首先，计算器工厂是单例的，但以后如果要扩展函数库，则可以改写成多种工厂。获取工厂后，传入公式字符串（多行），以通过工厂获取针对此公式的计算器。然后调用计算器的计算方法，即可获得公式最终的结果。

# JDK

另外，从JDK1.6开始，默认加入了javax.script包，如果不想引入第三方包，也可以使用这个。