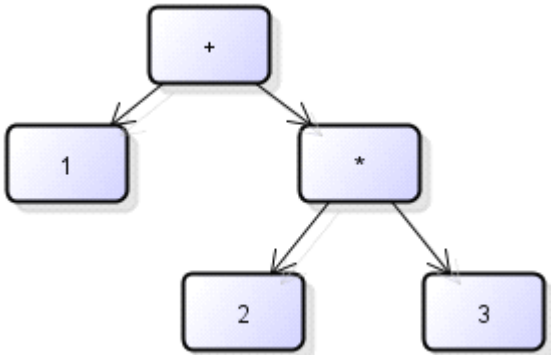
Fel 表达式引擎

一、名词解释

Fel：全称是 Fast Expression Language，一种开源表达式引擎。
EL：表达式语言，用于求表达式的值。
Ast：抽象语法树，一般由语法分析工具生成。1+2*3 会解析结果如下所示：



二、简介

Fel（Fast Expression Language)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。

1. 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200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），

2. 不足

支持脚本：否。

3. 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引擎的地方（如果工作流、公式计算、数据有效性校验等等）

三、安装

1. 获取 Fel

项目主页：http://code.google.com/p/fast-el/

下载地址：http://code.google.com/p/fast-el/downloads/list

2. Jdk1.6 环境

使用：将 fel.jar 加入 classpath 即可。

构建 Fel：下载 fel-all.tar.gz，解压后将 src 作为源码文件夹，并且将 lib/antlr-min.jar 加入 classpath 即可。

3. Jdk1.5 环境：

与 jdk1.6 环境下的区别在于，需要添加 jdk 内置的 tools.jar 到 classpath。

四、功能

1. 算术表达式：

```
FelEngine fel= new FelEngineImpl();
Object result= fel.eval("5000*12+7500");
System.out.println(result);
```

输出结果：67500

2. 变量

使用变量，其代码如下所示：

```
FelContext ctx= fel.getContext();
ctx.set("单价", 5000);
ctx.set("数量", 12);
ctx.set("运费", 7500);
Object result= fel.eval("单价*数量+运费");
System.out.println(result);
```

输出结果：67500

3. 调用 JAVA 方法

```
FelEngine fel= new FelEngineImpl();
FelContext ctx= fel.getContext();
ctx.set("out", System.out);
fel.eval("out.println('Hello Everybody'.substring(6))");
```

输出结果：Everybody

4. 自定义上下文环境

```
//负责提供气象服务的上下文环境
FelContext ctx= new AbstractConetxt() {
    public Object get(Object 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

5. 多层上下文环境（命名空间）

```
FelEngine fel= new FelEngineImpl();
```

```
String costStr= "成本";
String priceStr="价格";
FelContext baseCtx= fel.getContext();
//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);
```

输出结果：
期望利润：50
实际利润：30

6. 编译执行

```
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);
```

执行结果：67500
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。

7. 自定义函数

```
        //定义 hello 函数
    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);
```

执行结果：
hello fel hello fel

8. 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。

```
    //调用 Math.min(1,2)
FelEngine.instance.eval("$('Math').min(1,2)");
//调用 new Foo().toString();
FelEngine.instance.eval("$('com.greenpineyu.test.Foo.new').toString()");
```

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。

五、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（黑名单）和禁止访问的方法列表（白名单）来控制方法访问。将**"java.lang.System. *"**加入到黑名单，表示 System 类的所以方法都不能访问。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。

六、语法

1. 简介

Fel 的语法非常简单，基本和 Java 语法没没有区别。Fel 语法是 Java 语法的一个子集，支持的运算符有限。如果熟悉语法的话，只需要关注一下 Fel 支持的运算即可。

Fel 表达式由常量、变量、函数、运算符组成。详见下文。

2. 常量

常量	说明
Number	1,1.0 等
Boolean	true,false
String	'abc',"abc"

3. 变量

变量命名规则与 java 变量相同。变量名由字母、数字、下划线、$组成。

变量	说明
abc	以字母开头
abc	以下划线开头
$abc	以$开头
变量	中文变量名也可以

4. 函数

函数名的语法规则与变量相同

函数	说明
$('Math')	$是变量名，'Math'是参数

5. 运算符

运算符	类型	优先级	说明
？：	条件	优先级从低到高	三元操作符
||	逻辑		或操作
&&			与操作
==、!=	关系		等于、不等于
>、<、<=、>=			大于、小于、大于等于、小于等于
+、-	算术		加减
*、/、%			乘、除、取模
!	逻辑		取反操作

七、结构

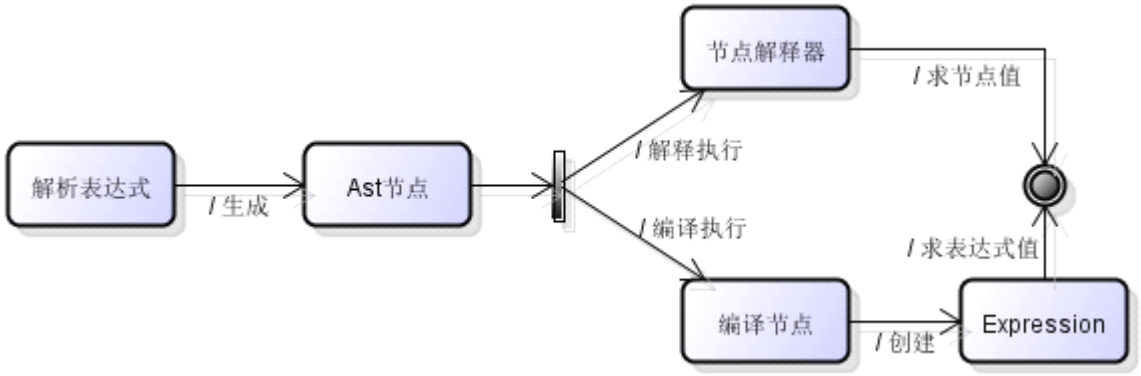
1. 主要组成部分

引擎由四部分组成，每个组成部分都可以被替换。组成部分如下所示：

组件	说明
解析器	将表达式解析成 Ast 节点
解释执行	负责执行节点
执行上下文	负责提供变量
编译器	负责生成代码并编译

2. 主要流程

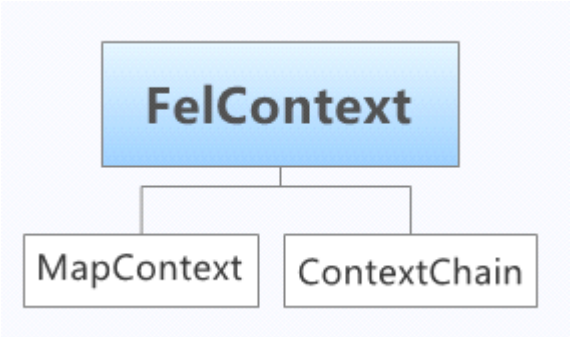
执行表达式的流程如下所示：



组件	说明
解析表达式	默认 Antlr 解析表达式，生成 Ast 节点。
Ast 节点	由解析器生成的抽象语法树节点
节点解释器	每一个节点都有一个解释器，负责解释节点。
编译器	负责生成表达式对应 Java 类并编译，生成 Expression 对象

3. 引擎上下文

引擎上下文负责存取变量，它是脚本引擎与 Java 对象之间的桥梁。其结构如下所示：



组件	说明
FelContext	负责存储和保存变量
MapContext	使用 Map 来保存变量
ContextChain	上下文链，保存两个上下文的引用。存取变量委托引用的上下文处理。

4. 表达式解析

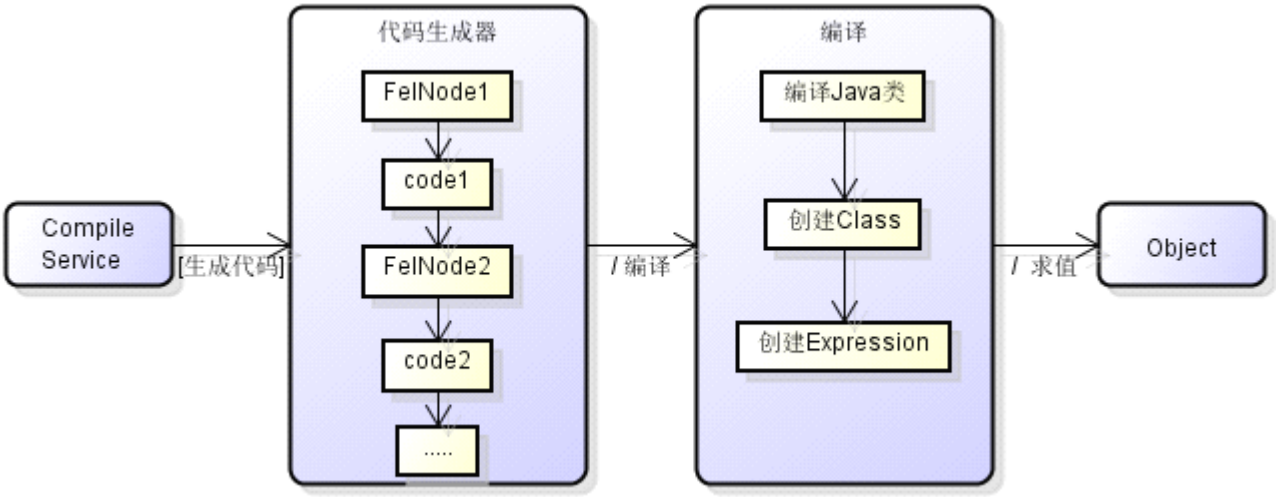
解析表达式的过程就是将表达式解析成 Ast 节点的过程，目前 Fel 由 Antlr 负责解析表达式，生成 Ast 节点（FelNode),流程如下所示：

5. 解释执行

FelNode(Ast 节点）中都包含一个解释器，不同的节点有不同的解释器，解释器负责求节点值，流程如下所示：

6. 编译执行

FelNode(Ast 节点）中都包含一个代码生成器，负责将 Fel 表达式转换成 java 表达式。编译模块负责将 java 表达式封装成 Java 类、编译、创建 Expression。再通过调用 Expression.eval 求表达式的值。其过程如下所示：



7. 小结

在上文介绍的 Fel 组件中，绝大多数组件都是可能被替换的。通过在运行时替换一些小粒度的组件，就可以灵活控制 Fel 的执行过程，很有实用价值。有些变化较少的组件(Antlr 解析器)的替换过程还不是挺方便，随着 Fel 的发展，会慢慢重构，努力使 Fel 转变成让人满意的模块化结构。