# 2.3 规则引擎调研

## 1.概述

> 规则引擎是一种嵌入在应用程序中的组件，勇于执行定义、执行规则以及规则后续的行为，实现了将业务决策从应用程序代码中分离出来，并使用预定义的语义模块编写业务决策。接受数据输入，解释业务规则，并根据业务规则做出业务决策，一个好的规则引擎能大大提高系统的灵活性，扩展性。
>
> 规则条件匹配的效率决定了引擎的性能，引擎需要迅速测试工作区中的数据对象，从加载的规则集中发现符合条件的规则，生成规则执行实例。

规则引擎通常是作为Business Rule Management System(BRMS)软件系统一部分。为什么要单独有开发规则引擎呢？最重要的原因是，**业务规则是比应用代码变化更频繁的部分**（软件设计模式中单一职责原则，将变化的部分与不变的部分分离出来）

当业务规则变更时，不需要改代码，也不需要发布上线等额外的人员参与。**提高系统的灵活性，扩展性。**

规则引擎起始与1990s，主要用于业务规则比较重的领域，比如金融与保险（各种条款），后续的发展，业务规则中的所有的系统都会选择使用合适的规则引擎系统改造系统，增加系统的灵活性。

规则引擎，可以作为推理引擎的一种分支，因为推理引擎就是输入数据中间经过处理（规则、决策树、神经网络、模式识别）输出数据，规则引擎是作为推理引擎的某个分支，但是也不是完全的推理引擎，因为推理引擎可能会更改原始的数据，二规则引擎则不会改变原始数据，不匹配的时候，什么都不做。

## 2.规则引擎核心

### 2.1 关键术语

- BRM: Business Rule Management，业务规则管理
- LHS: left hand side，if部分也就条件部分;
- RHS: right hand side, then 部分也就是通过后的行为;
- FACT：事实，已经发生的事或者也叫做数据;
- RULE：规则 if then的推理语句；

### 2.2 规则描述语言

表达式语言MVEL、SpEL、JEXL等

### 2.3 模式匹配算法

Rete算法是一种模式匹配算法（类似于正则表达式），主要用于规则引擎中，算法的输入是规则与数据对象（facts）得到匹配的规则。算法是卡耐基梅隆大学的查尔斯·L·福吉于1974年发表，1979年在他的博士毕业论文中有更详细的论证。

最简单的一种执行规则匹配算法就是循环遍历，但是这种方式效率比较低，性能比较差，随着规则数量的增长，会越来越慢；Rete算法使用了一种类似于单词查找树的机制（B树，B+树）加快查询，查尔斯·L·福吉为什么起这个名字，就是因为这个名字在拉丁语中的含义就是network，为什么要叫网络，也是因为受到了毛细血管系统或者神经元系统的网络状的启发。

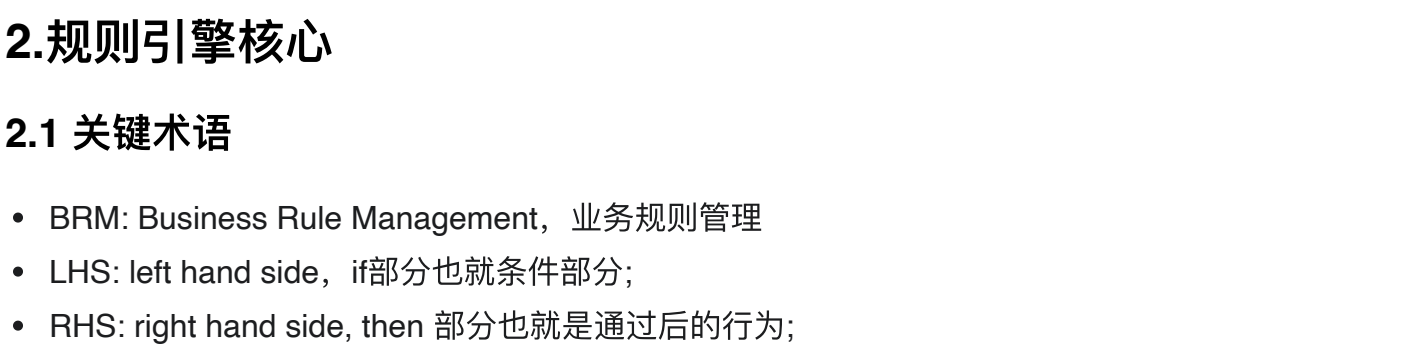



图1 推理过程

Rete算法有2个主要的作用：

- 将定义的规则分解为Rete网络，这个叫做作规则编译，Rete网络就是编译的结果，可以是规则的网络表示形式或者有向无环图关系表示形式；
- 完成路径推理过程，最终找到匹配的规则。

Rete Network由2部分组成：

- Top part=Alpha Network，由1-input的node类型叫做alpha node与alpha memory组成
- Bottom part=Beta Network，由2-input的node类型叫做beta node与beta memroy组成

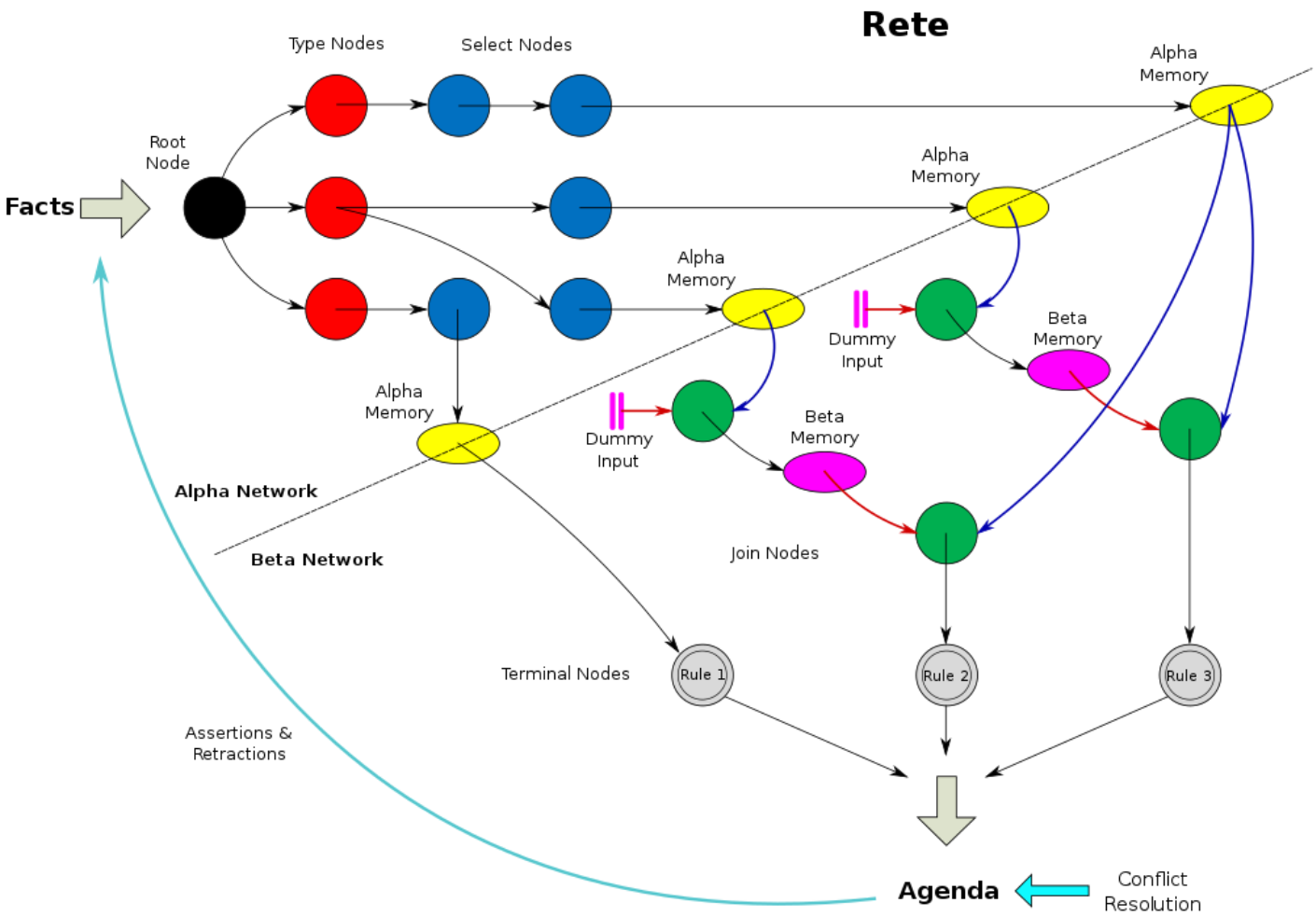



图2 Rete网络

## 2.4 规则引擎分类

| 分类 | 描述 | 产品 |
| --- | --- | --- |
| 正向推理 | 推理引擎的一个子类，适用模式匹配的方式匹配规则与数据，常使用Linear, Rete, Treat, Leaps等算法 | Drools |
| 有限状态机 | 有限个状态以及在这些状态之间的转移和动作 | Azure IoT Explorer |
| Condition/Action | 正向推理的特殊的形式，只支持if then action这种结构 | |
| 决策树 | 由一个决策图和可能的结果（包括资源成本和风险）组成， 用来创建到达目标的规划。决策树建立并用来辅助决策，是一种特殊的树结构 | |
| Complex Event Processing (CEP) | 用于处理实时事件并在事件流到达时从事件流中提取信息。 复杂事件处理的目标是在实时情况下识别有意义的事件并尽快做出响应 | Jetlink/Rule-Engine |
| Flow Based | 它将应用定义为黑箱进程的网络，它们经过预先定义的连接，通过消息传递来交换数据，而这里的连接是在“外部”指定给进程的。这些黑箱进程不需要更改内部，就可以无尽的重新连接而形成不同的应用。FBP因而是天然组件化的 | |

# 3.规则引擎应用架构

大多数系统使用规则的系统，通常都是规则与规则的执行流程耦合的；可能规则是外部配置的，但是整体仍是强绑定的；这种情况下，业务规则与整个处理流程的复用性都会变差，一种新的方法是将规则与规则的执行分开2部分：

- 表示知识的业务规则
- 驱动业务规则执行的任务与后续一系列行为的驱动与流转（工作流系统、流处理系统）

一种基于事件的规则引擎系统定义

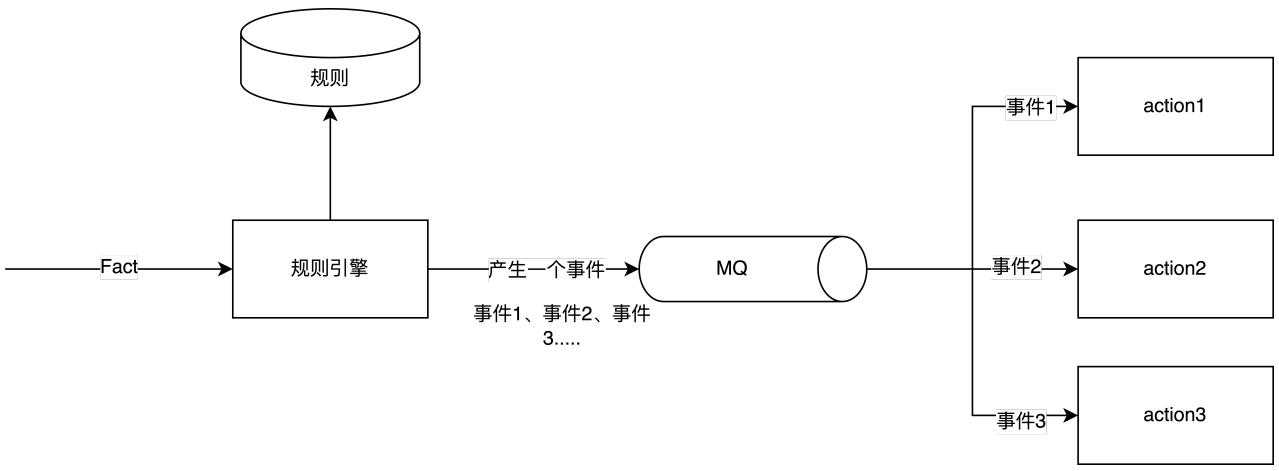



图3 基于事件的规则引擎系统

整个规则的复用性比较高，比较灵活。

## 3.1 基于微服务的流式架构

基于微服务的流式架构+规则引擎=实时的CEP（Complex Event Processing）。比较适合IOT的场景。

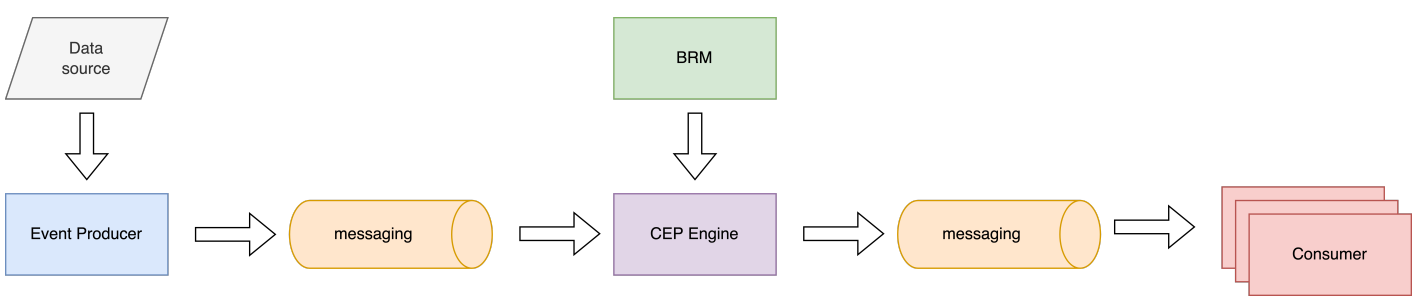



图4 CEP系统数据流

## 3.2 设计原则

- 业务规则与应用代码分离
- 决策过程与行为过程分离

# 4.常用的规则引擎

常见的规则引擎由RedHat Drools、EasyRule、QLExpress

## 4.1 Drools规则引擎系统

Drools是一个业务规则管理系统（BRMS）解决方案，它包含了基于Rete算法实现的规则引擎；包含好多组件

- Drools Workbench，管理存储知识库的存储库与UI
- Drools Expert（rule engine），规则引擎
- Drools Flow（workflow），提供了工作流相关的内容
- Drools Fusion (CEP)，提供了时许数据处理的一些特性
- Drools Planner（自动规划）

Drools是使用纯java语言实现的，核心的规则引擎部分是非常小的，可以自由集成到自己的应用中。核心只有3个JAR包。核心JAR如下：

- knowledge-api.jar 提供接口抽象定义，包括用户API与引擎API;
- knowledge-internal-api.jar 内部使用的接口定义API;
- drools-core.jar Drools engine 运行时组件，包含了RETE engine与 the LEAPS engine. 这是唯一的运行时依赖，如果之前规则被编译了的话，编译后的规则是KnowledgePackage对象或者KnowledgeBase对象；
- drools-compiler.jar 编译规则的组件创建规则执行库，依赖drools-core组件，如果规则已经被编译过了，就不需要，没有编译的话要依赖；
- drools-jsr94.jar 是drools-compiler.jar的jsr94标准实现.
- drools-decisiontables.jar -decision tables编译器, 会使用到drools-compiler组件，决策表支持CSV与EXCEL2种格式。

关键抽象：

- KieServices，一个通用的工具类
- KieContainer，业务规则容器
- KieBase，知识库
- KieModule，非常重要的模块，包含所有的知识库规则等业务定义；通常等价于kmodule.xml，如果没有kmodule.xml文件，可以用编程的方式创建KieModule
- KieSession，规则引擎实例
- KieFileSystem，一个内存的Maven文件系统
- KieRepository，用来存储KieModel的单例，在生成KieContainer时有用

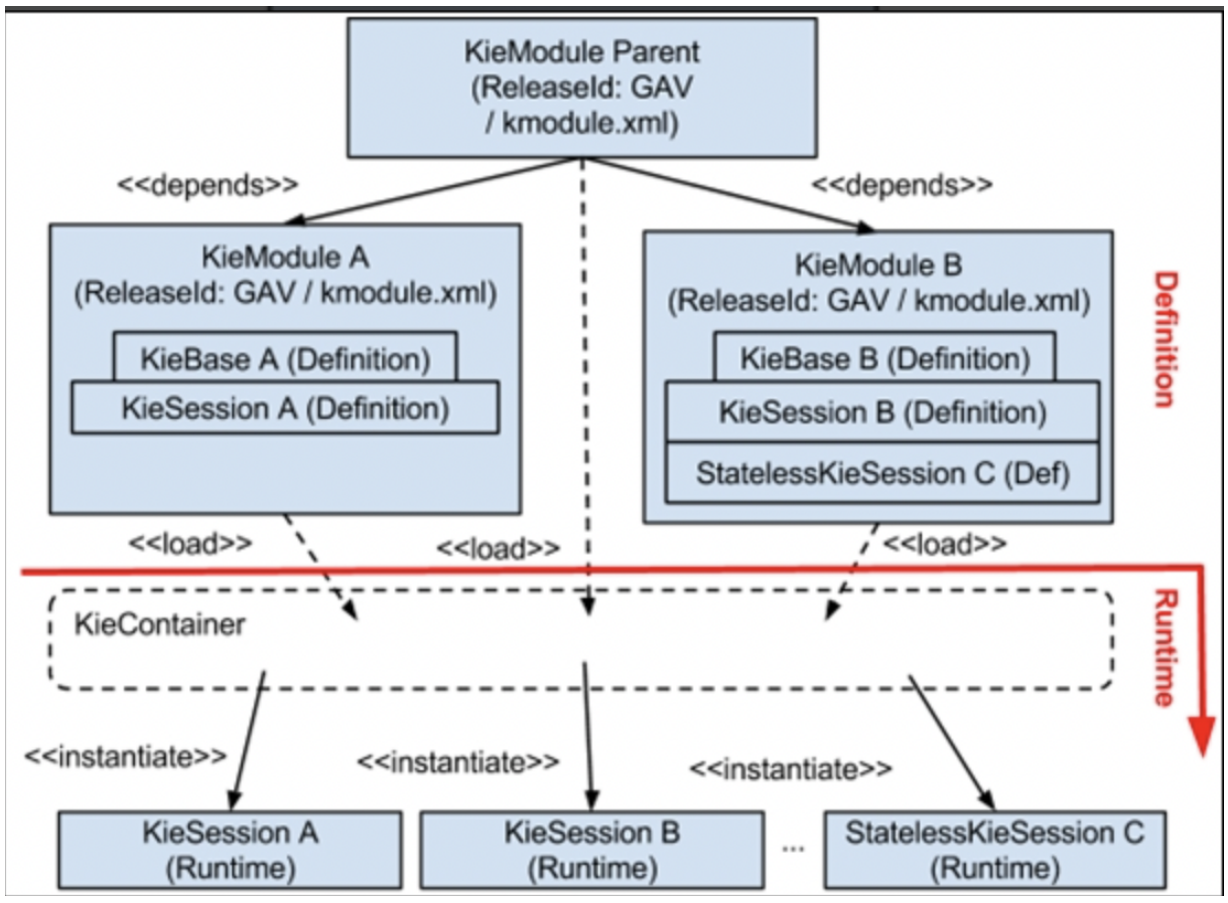



图5 组件结构

声明式基本用法：

- 在项目resource下面创建META-INF/kmodule.xml，这是一种声明式用法

HTML

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule">
    <kbase name="rules" packages="com.sample.rules">
        <ksession name="ksession-rules"/>
    </kbase>
    <kbase name="dtables" packages="com.sample.dtables">
        <ksession name="ksession-dtables"/>
    </kbase>
    <kbase name="process" packages="com.sample.process">
        <ksession name="ksession-process"/>
    </kbase>
</kmodule>
```

- 在resource的目录下定义规则文件drl

TOML

```
package com.sample.rules

import com.sample.DroolsTest.Message;

rule "Hello World"
    when
        m : Message( status == Message.HELLO, myMessage : message )
    then
        System.out.println( myMessage );
        m.setMessage( "Goodbye cruel world" );
        m.setStatus( Message.GOODBYE );
        update( m );
end

rule "GoodBye"
    when
        Message( status == Message.GOODBYE, myMessage : message )
    then
        System.out.println( myMessage );
end
```

- 编写代码启动规则匹配

Java

```java
package com.sample;

import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

/**
 * This is a sample class to launch a rule.
 */
public class DroolsTest {

    public static final void main(String[] args) {
        try {
            // load up the knowledge base
            KieServices ks = KieServices.Factory.get();
            KieContainer kContainer = ks.getKieClasspathContainer();
            KieSession kSession = kContainer.newKieSession("ksession-rules");

            // go !
            Message message = new Message();
            message.setMessage("Hello World");
            message.setStatus(Message.HELLO);
            kSession.insert(message);
            kSession.fireAllRules();
        } catch (Throwable t) {
            t.printStackTrace();
        }
    }

    public static class Message {

        public static final int HELLO = 0;
        public static final int GOODBYE = 1;

        private String message;

        private int status;

        public String getMessage() {
            return this.message;
        }

        public void setMessage(String message) {
            this.message = message;
        }

        public int getStatus() {
            return this.status;
        }

        public void setStatus(int status) {
            this.status = status;
        }

    }

}
```

Drools的：

- 开源，长期维护，资料比较全；
- 使用Rete算法，执行速度快；
- 轻量级使用特别方便;
- 内存消耗量会比较大;

# 4.2 Easy Rules规则引擎

Easy Rules是一个简单而强大的Java规则引擎，提供以下功能：

- 轻量级框架和易于学习的API
- 基于POJO的开发与注解的编程模型
- 定义抽象的业务规则并轻松应用它们
- 支持从简单规则创建组合规则的能力
- 支持使用表达式语言（如MVEL和SpEL）定义规则的能力

Martin Fowler在一篇有关规则引擎的文章中提到：

> You can build a simple rules engine yourself. All you need is to create a bunch of objects with conditions and actions, store them in a collection, and run through them to evaluate the conditions and execute the actions.

使用方法：

HTML

```
<dependency>
    <groupId>org.jeasy</groupId>
    <artifactId>easy-rules-core</artifactId>
    <version>4.1.0</version>
</dependency>
```

Java

```
package com.zyx.java.easyrule;

import org.jeasy.rules.annotation.Action;
import org.jeasy.rules.annotation.Condition;
import org.jeasy.rules.annotation.Rule;
import org.jeasy.rules.api.Facts;

@Rule(name = "Hello World rule", description = "Always say hello world")
public class HelloWorldRule {

    @Condition
    public boolean when(final Facts facts) {
        return "world".equals(facts.get("hello"));
    }

    @Action
    public void then(final Facts facts) throws Exception {
        System.out.println("hello world");
    }
}
```

Java

```
package com.zyx.java.easyrule;

import org.jeasy.rules.api.Facts;
import org.jeasy.rules.api.Rules;
import org.jeasy.rules.api.RulesEngine;
import org.jeasy.rules.core.DefaultRulesEngine;

public class Launcher {

    public static void main(final String[] args) {

        // create facts
        final Facts facts = new Facts();
        facts.put("hello", "world");

        // create rules
```

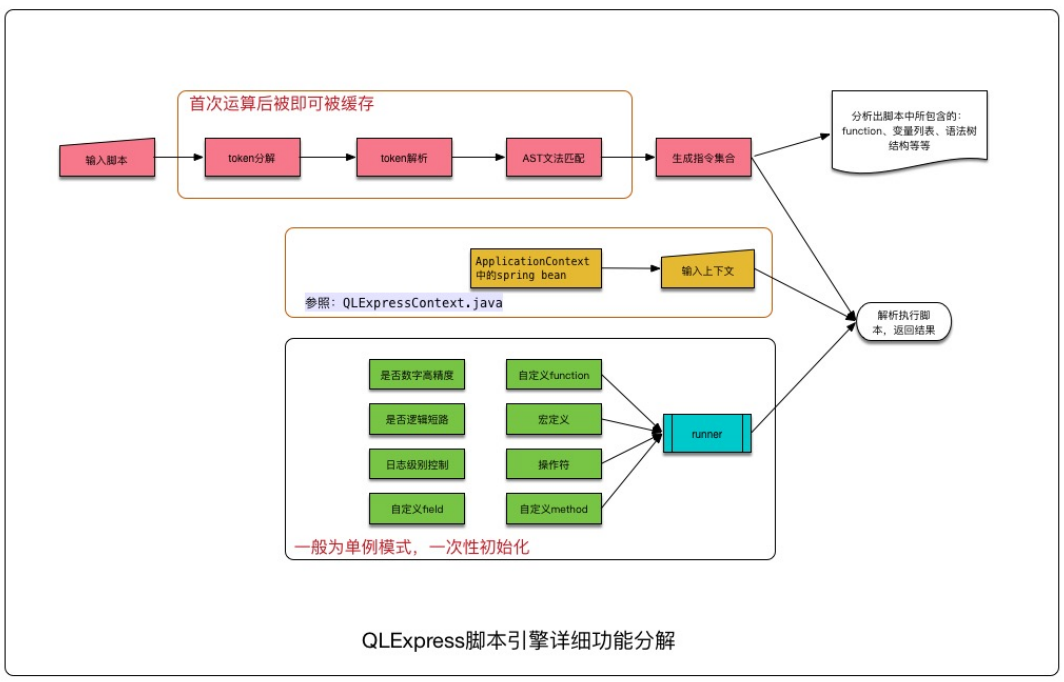
```
17          final Rules rules = new Rules();
18          rules.register(new HelloWorldRule());
19
20          // create a rules engine and fire rules on known facts
21          final RulesEngine rulesEngine = new DefaultRulesEngine();
22          rulesEngine.fire(rules, facts);
23
24      }
25  }
```

- 轻量级，上手简单
- 生产级别的规则，可能会造成性能问题
- 与表达式引擎没有太大区别

## 4.3 QLExpress脚本引擎

QLExpress是阿里开源出来的用于定义电商业务规则、表达式、动态脚本的一个脚本引擎库，功能非常强大，具有的特点：

- 线程安全;
- 性能高，主要是表达式、脚本、规则的编译缓存;
- 支持弱类型的脚本定义;
- 可以通过参数控制执行的行为;
- 库比较小,代码少，使用起来比较简单
- 扩展能力，定制能力




QLExpress脚本引擎详细功能分解

图5 功能分解

用法：

- 支持常用的类java语言的操作符，可以操作java对象;
- 支持自定义Function、Operator、宏;
- 支持脚本

总结：

- 比较灵活;
- 性能依赖脚本的编写;
- 多个规则仍然需要遍历的方式访问

## 4.4 比较

| 规则引擎 | 规则编写 | 性能 | 难度 | 完备性 | 业务规则表达能力 | 规则可读性 | 灵活性 | 扩展性 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Drools | MVEL | 高 | 中 | 推理引擎 | 高 | 高 | 中 | 低 |
| Easy Rules | MVEL、SpEL或者Java自定义 | 低 | 低 | 不完备 | 高 | 低 | 高 | 低 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| QLExpress | MVEL或者java自定义 | 中 | 低 | 不完备 | 高 | 低 | 高 | 低 |

# 5.总结

- 规则引擎是剥离多变的业务规则后形成的组件；
- 规则引擎多数都是基于Rete算法实现的，是属于正向推理的推理引擎；
- CEP+规则引擎适用实时数据流的多数场景，比如IoT、推荐、日志分析、行为分析等场景；
- 常见的规则引擎有Drools、Easy-Rule、QLExpress。