

规则引擎及规则表达式调研

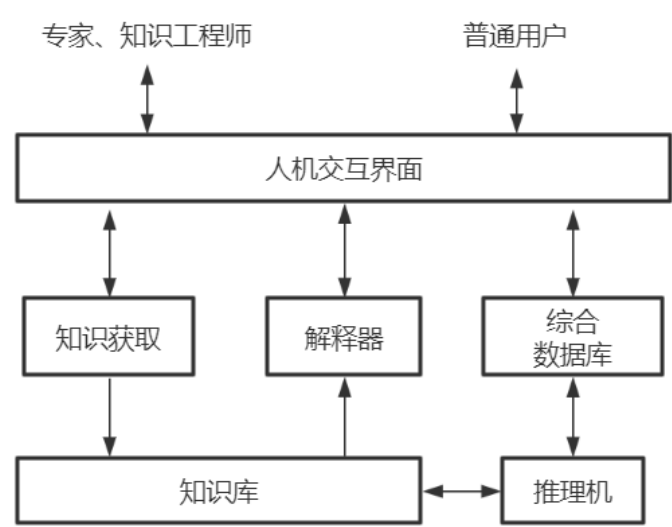
1 规则引擎

1.1 背景介绍

- 专家系统

专家系统是具有大量专门知识和经验的程序系统，模拟人类专家的决策过程，根据某领域多个专家提供的知识和经验进行推理和判断，以此来解决需要人类专家处理的复杂问题。

- 专家系统组成结构：



知识库：问题求解所需的领域知识集合，包含基本事实、规则等信息。知识表示形式：语义网络、框架、谓词逻辑、规则等。

推理机：实施问题求解的核心执行结构，在一定的控制策略下，根据问题信息及知识库中的知识对问题进行求解。

知识获取：负责建立、修改和扩充知识库，知识获取方式：手工、半自动、全自动知识获取。

解释器：用于对求解过程作出说明，让用户理解程序正在做什么以及为什么这么做。

综合数据库：用于存放系统运行过程中产生的信息，包括用户输入信息、推理中间结果、推理过程记录等。

人机交互界面：知识管理、回答系统提出的问题、推理结果输出。

- 规则引擎

- 1、知识库中知识的表示形式为规则性知识。
- 2、基于规则的推理机易于理解、获取、管理，并被广泛采用，因此这种推理引擎被称为“规则引擎”。

1.2 引擎优点

- 声明式编程：简化对于复杂问题的逻辑表述
- 业务逻辑和数据分离，简化系统架构
- 业务知识集中化，规则可以作为数据存储、管理和更新
- 规则热加载

1.3 常用规则引擎

Drools : JBOSS开源规则引擎

URule: 国产，分开源版、商业版

Esper: 事件流处理和事件关联的引擎

2 表达式引擎

2.1 基本介绍

- 表达式引擎轻量通常作为一个嵌入式的“规则引擎”在业务系统中一起使用
- 主要用于表达式的动态求值
- 通常没有规则管理后台页面，需要根据业务需求进行深度开发定制

2.2 常用表达式引擎

IK Expression: 基于java语言实现，超轻量级，功能有限，公式化语言解析执行工具包

Aviator: 基于java语言实现，高性能、轻量级，编译执行，国产

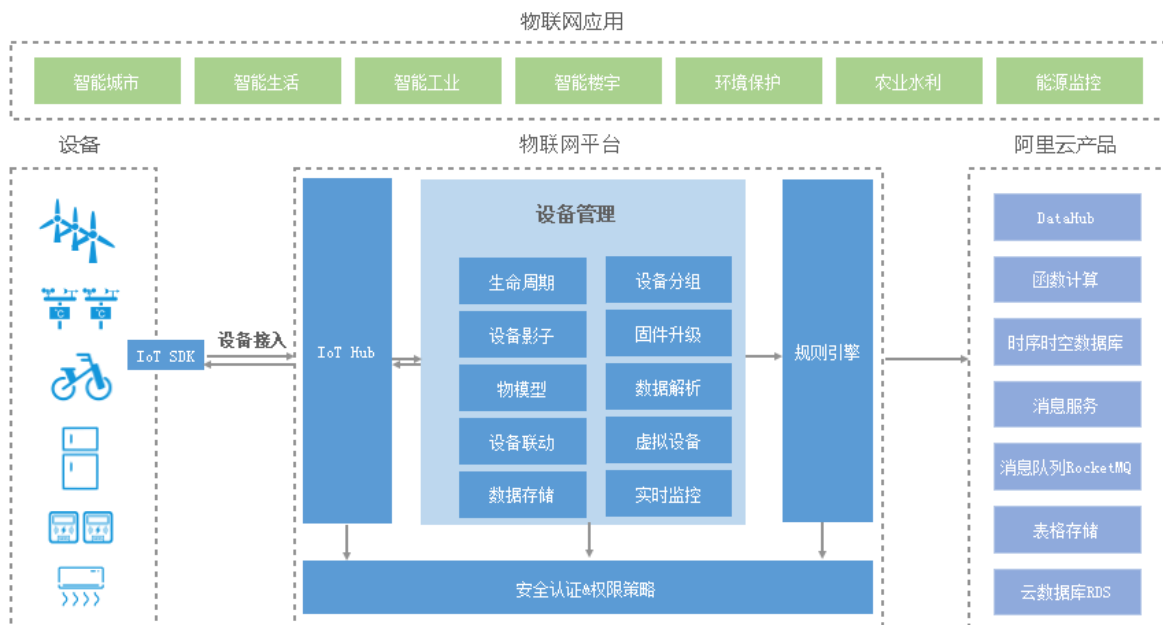
Spel: Spring表达式语言，旨在向Spring社区提供一种受良好支持的表达式语言，不需要与Spring绑定，可独立使用

QLEXPRESS: 阿里开源，线程安全、高效，跟groovy性能相当

3 规则应用场景

3.1 阿里云物联网

阿里云物联网产品总体架构图：

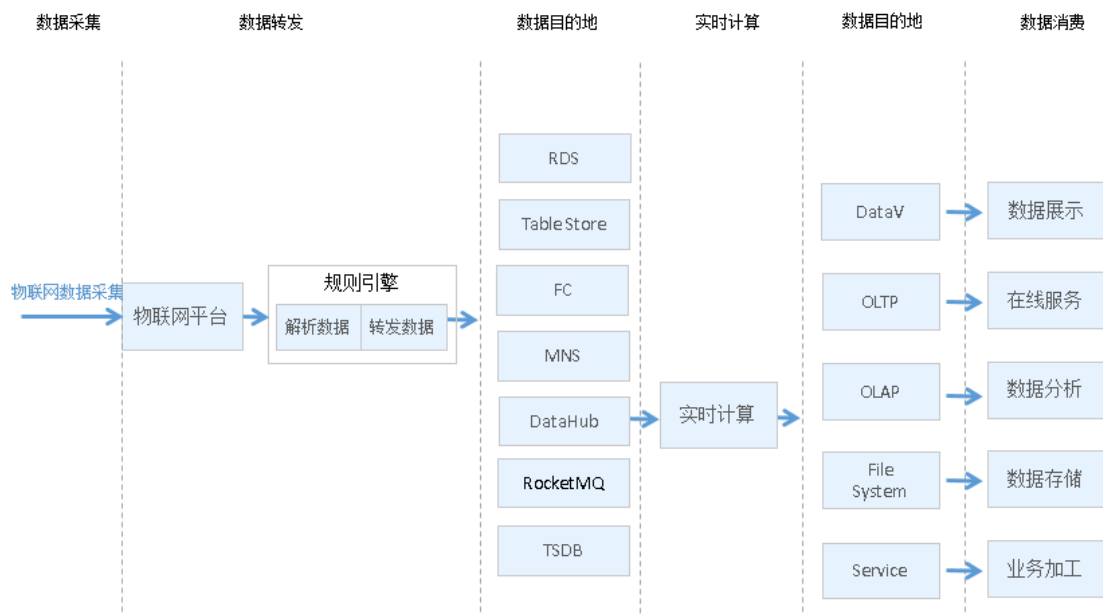


规则引擎在阿里云物联网平台的作用：

1. 数据流转
2. 场景联动

3.1.1 数据流转

数据流转流程图



数据流转使用限制

- 1、数据流转基于**Topic**对数据进行处理。只有通过**Topic**进行通信时，才能使用数据流转。
- 2、通过SQL对**Topic**中的数据进行处理
- 3、SQL语法暂时不支持子查询。
- 4、SQL语句支持部分函数，如**deviceName()**获取当前设备名称

流转规则配置界面

1、数据转发内容规则配置

2、转发目的地配置

编写SQL

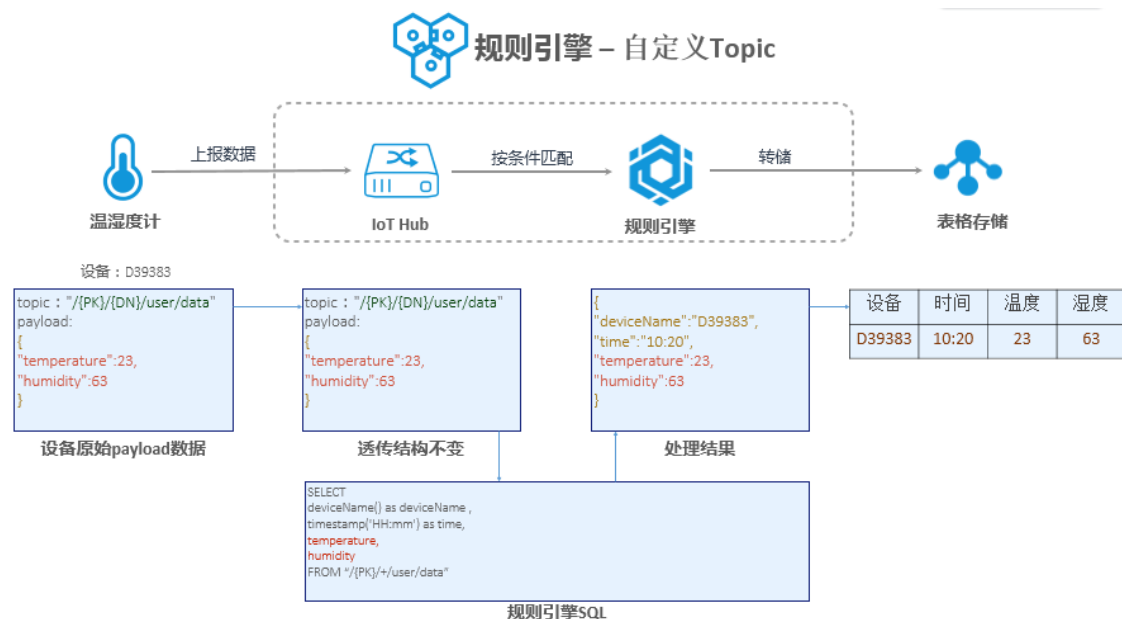
规则查询语句：
SELECT TargetDevice,Switch,Timestamp
FROM "/a1hRrzd****/ControlApp/user/command"
WHERE
字段
TargetDevice,Switch,Timestamp
Topic
自定义
apptest
ControlApp
user/command

确认 取消

编辑操作

选择操作
发布到另一个Topic
Topic
/a1D0UyVeQ1d/{TargetDevice}/user/set
自定义
智能灯123
{TargetDevice}
user/set
确定 取消

应用实例



规则引擎SQL表达式定义



SQL示例：

```
SELECT temperature as t, deviceName() as deviceName, location FROM  
"/a1hRrzd****/+user/update" WHERE temperature > 38
```

3.1.2 场景联动

规则模型：TCA模型，触发器（Trigger）、执行条件（Condition）、执行动作（Action）三部分组成。

联动规则配置示例

场景联动规则

触发器 (Trigger) ?

触发器1

定时触发

0 18 ***

+ 新增触发器

执行条件 (Condition) ?

执行条件1

设备状态

温度传感器

temperatureSensor01

温度

>

26

+ 新增执行条件

* 执行动作 (Action)

执行动作1

设备输出

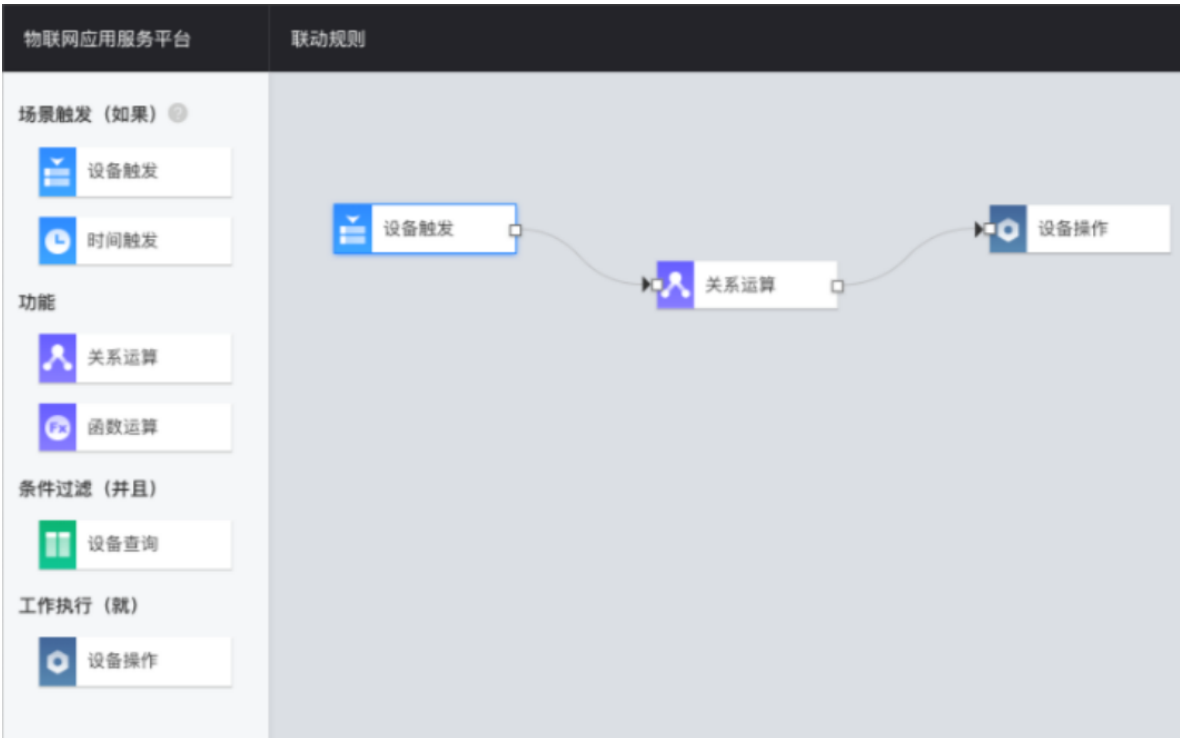
空调

airconditioner_1

电源开关

开启-1

联动规则配置可视化



组件名称
设备触发

产品选择
光照传感器

实例
选择设备实例

实例信息
空间位置: 1号楼/1F/大厅
产品名称: 光照传感器
DeviceName: 5ei2fANyBaUDQwT1esu2

1、触发器配置

组件名称
关系运算

数据源
属性: 光照度检测值

输出结果
☒ true ☐ false

规则配置
大于(>) 2000

+添加规则

2、触发条件

组件名称
设备操作

产品选择
射灯

操作类型
☒ 操作单个设备实例 ☐ 操作指定空间下的批量设备

实例
选择设备实例

实例信息
空间位置: 1号楼/1F/101
产品名称: 射灯
DeviceName: BjCyNdw6Tl5xKyB7GKQQ

操作配置
☒ 设置属性 收起
电源开关_1 开启-1

3、触发动作

3.1.3 技术选型

阿里开源规则引擎：QLExpress

腾讯云数据流转规则表示形式也是基于SQL

腾讯云

API 中心

数据结构

错误码

TBaaS 3.0

云监控 3.0

访问管理 3.0

标签 3.0

企业组织 3.0

密钥管理系统 3.0

云审计 3.0

迁移服务平台 3.0

数据万象

计费网关 3.0

搜索本产品下的文档内容

文档中心

TopicRulePayload

创建规则请求包体

被如下接口引用: CreateTopicRule, ReplaceTopicRule。

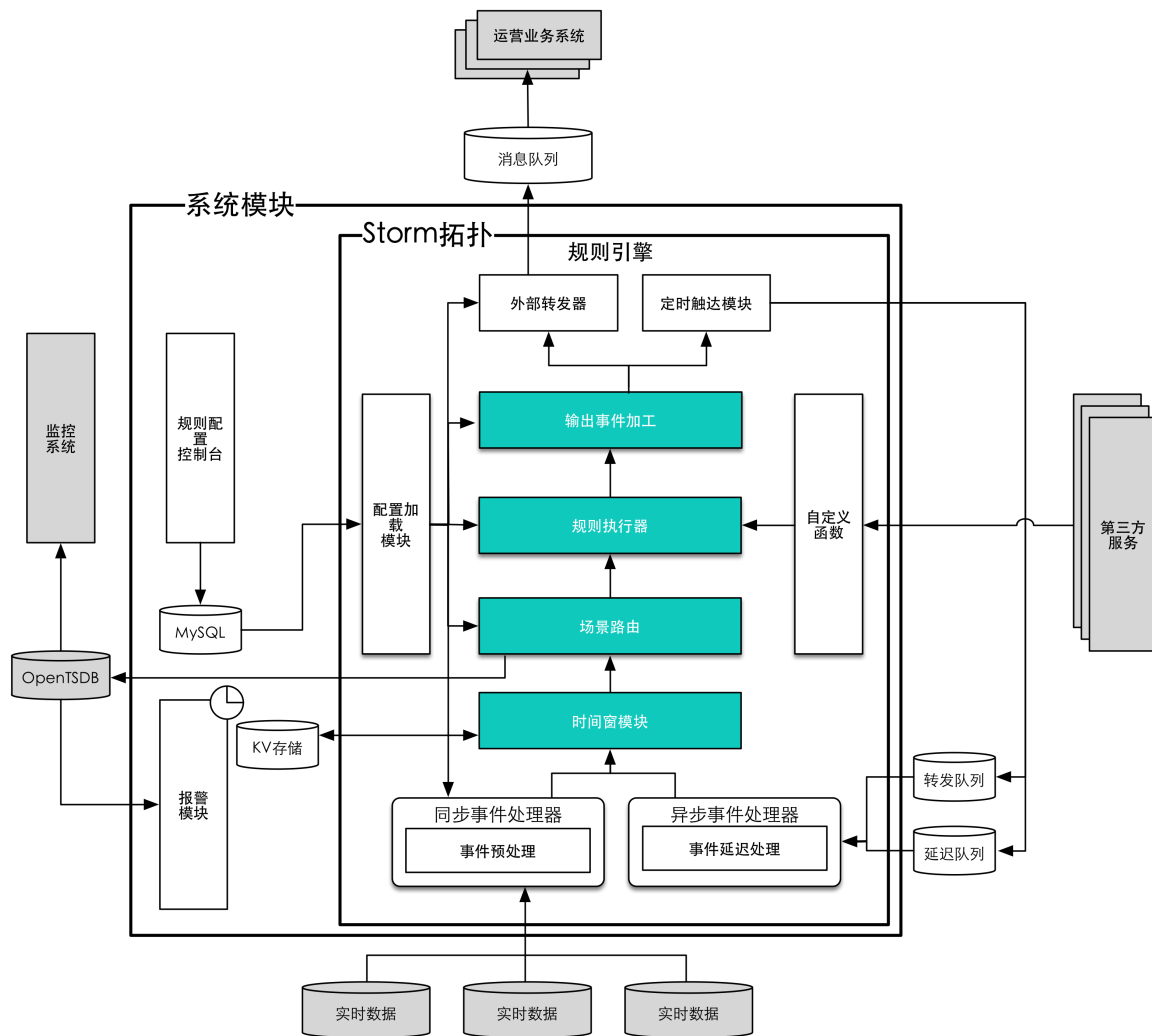
名称	类型	必选	描述
Sql	String	是	规则的SQL语句, 如: SELECT * FROM 'pid/dname/event', 然后对其进行base64编码, 得: U0VMRUNUICogRIJPTSAncGikL2RuYWw1IL2V2ZW50Jw==
			行为的JSON字符串, 大部分种类举例如下: [{ "republish": { "topic": "TEST/test" } }]

3.2 美团酒旅运营实时触达系统

3.2.1 场景介绍

运营业务通过挖掘用户的实时行为数据(如实时浏览、下单、退款、搜索等), 对满足特定条件或规则的用户, 发送实时触达的消息, 从而最大化运营活动效果。

3.2.2 系统模块



系统模块描述

规则引擎：集成于**Storm**拓扑中，执行运营活动条件转换成为的具体规则，作出对应响应。

时间窗模块：具有可选时间跨度的滑动时间窗功能，为规则判定提供时间窗因子。

定时触达模块：设定规则判定的执行时间，达到设定时间后，执行后续规则。

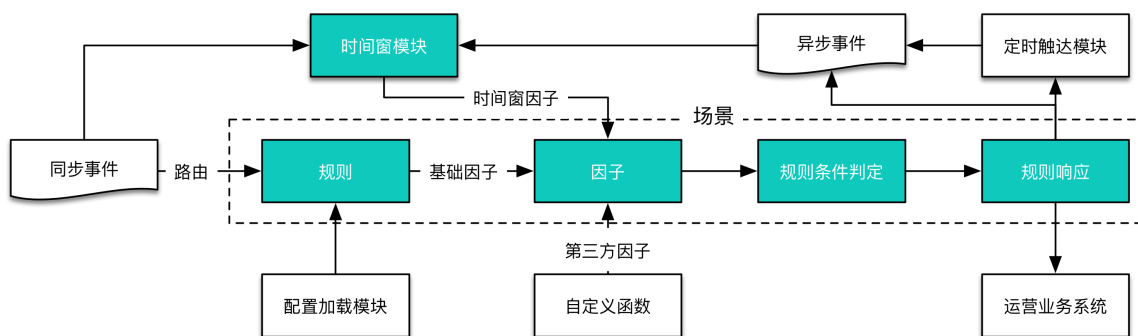
自定义函数：在**Aviator**表达式引擎基础函数之上，扩展规则引擎功能。

报警模块：定时检查系统处理的消息量，出现异常时为负责人发送报警信息。

规则配置控制台：提供配置页面，通过控制台新增场景及规则配置。

配置加载模块：定时加载活动规则等配置信息，供规则引擎使用。

规则引擎执行流程



执行流程描述

- 1、场景：业务需求的抽象，一个业务需求对应一个场景，一个场景由若干规则组成。由不同的规则组成时序和依赖关系以实现完整的业务需求。
- 2、规则：规则由规则条件及因子组成，由路由至所属场景的事件触发，规则由规则条件、因子及规则响应组成。
- 3、规则条件：规则条件由因子构成，为一个布尔表达式。规则条件的执行结果直接决定是否执行规则响应。
- 4、因子：因子是规则条件的基础组成部分，按不同来源，划分为基础因子、时间窗因子和第三方因子。基础因子来源于事件，时间窗因子来源于时间窗模块获取的时间窗数据，第三方因子来源于第三方服务，如用户画像服务等。
- 5、规则响应：规则执行成功后的动作，如将复合事件下发给运营业务系统，或发送异步事件进行后续规则判断等。
- 6、事件：事件为系统的基础数据单元，划分为同步事件和异步事件两种类型。同步事件按规则路由后，不调用定时触达模块，顺序执行；异步事件调用定时触达模块，延后执行。

3.2.3 技术选型

基于业务规则对时间窗功能、定时触达功能有较强的依赖，最终选择轻量级的表达式引擎：Aviator

3.3 网易考拉规则引擎平台

3.3.1 场景介绍

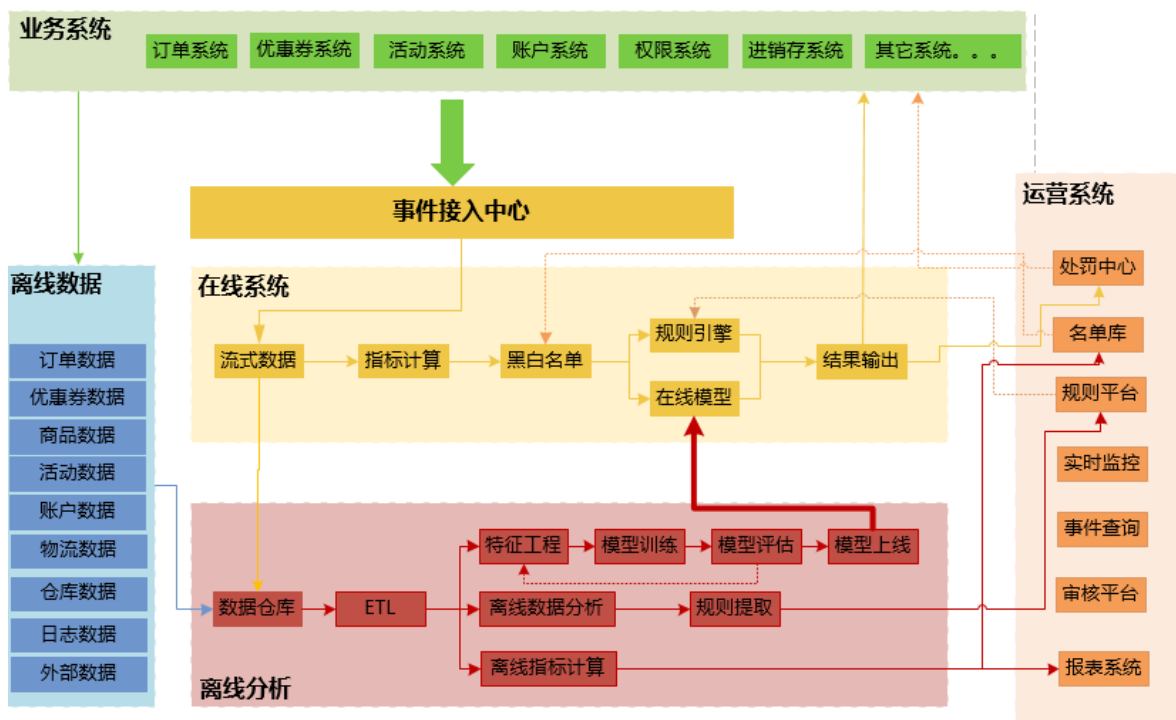
考拉安全全部技术负责两块业务：

- 1、内审：通过敏感日志管理平台搜集考拉所有后台系统的操作日志，经过大数据的实时计算、分析，主要提供行为查询、数据监控、事件追溯、风险大盘等功能。
- 2、业务风控：主要针对用户在下单、支付、优惠券、红包、签到等行为的风险控制，对抗的风险行为包括黄牛刷单、恶意占用库存、机器领券、撸羊毛等。

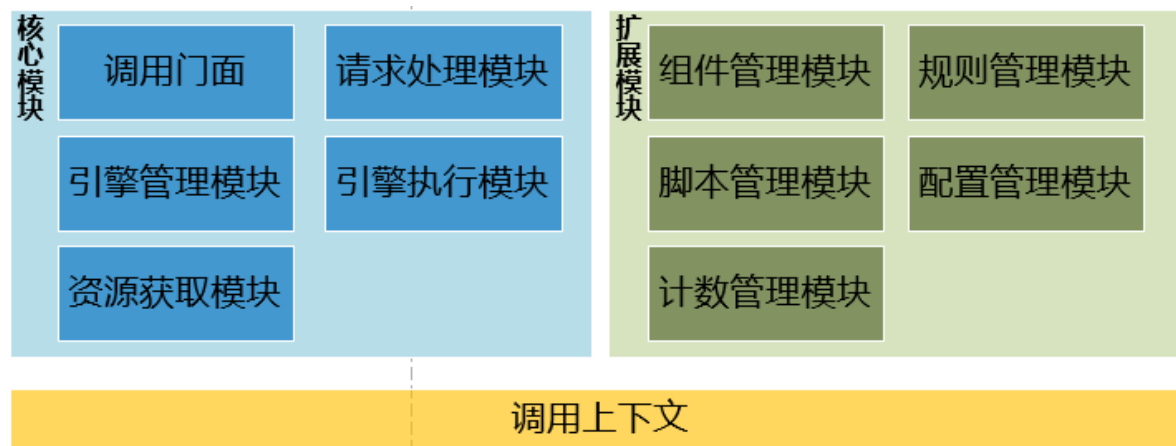
两个业务都有大量需要进行规则决策的场景。

3.3.2 系统模块

业务架构图



规则引擎模块



规则配置界面

规则配置界面 (Rule Configuration Interface)

规则名称: 测试1

事件类型: 下单

执行顺序: 0

业务类型: 下单 X

创建人: 郑和

创建时间: 2018年3月28日星期三上午10点43分

最后修改人: 公用测试账号

修改时间: 2018年3月28日星期三上午10点52分

编号	反向逻辑	左变量	操作	右变量	
1	<input type="radio"/>	下单dto.购买账号	equals_script	11111@163.com	X
2	<input type="radio"/>	下单dto.ip	equals_script	1.1.1.1	X

执行逻辑: 1&&2

生成默认逻辑

执行动作:

序号	动作	参数
1	命中主 下单dto.购买账号,发送 命中订单顶顶,发送	wb.zhenghe@mesg.corp.netease.com,抄送 hzcaohuanqing@corp.netease.com

提交

3.3.3 技术选型

选用groovy，弃用drools

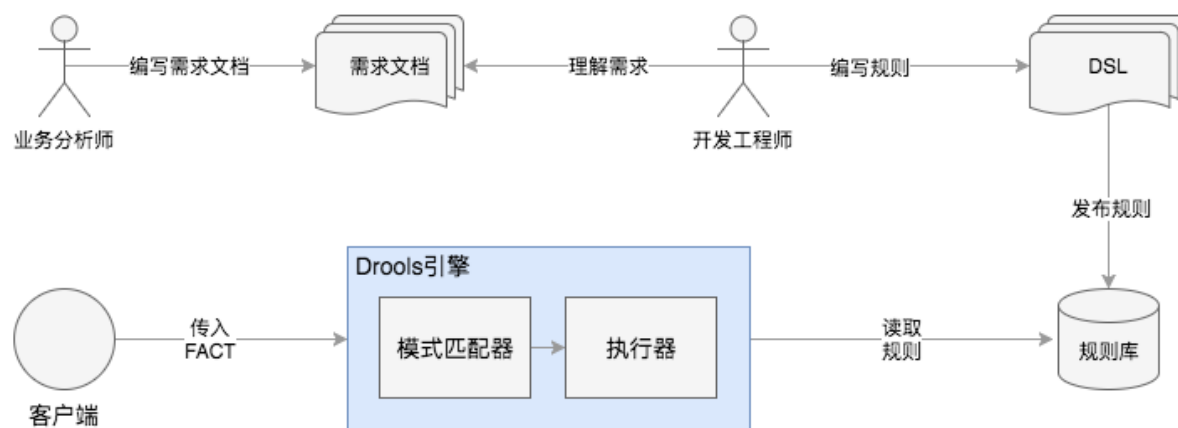
考虑点：

- drools相对来说有点重，单独的规则语言，对于开发和运行有学习成本
- drools使用起来没有groovy脚本灵活。groovy可以和spring完美结合，并且可以自定义各种组件实现插件化开发。
- 当规则集变得复杂起来时，使用drools管理起来有点力不从心。

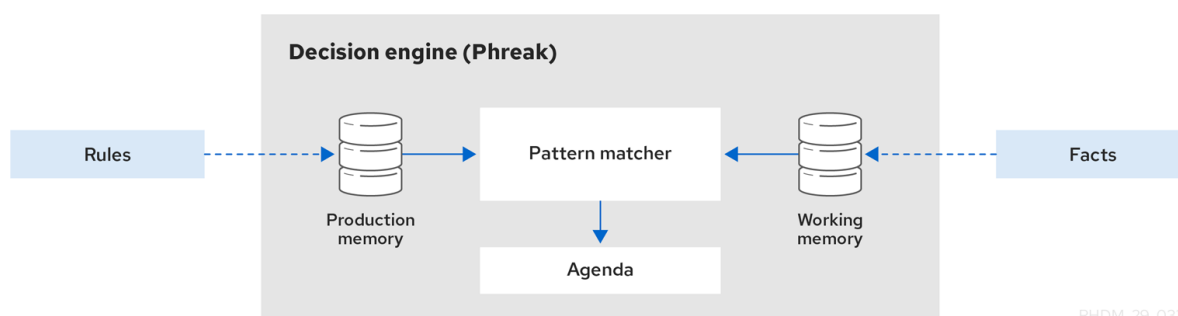
4 技术介绍

4.1 drools

4.1.1 使用流程



4.1.2 组件模块



RHDM_29_0319

组件介绍

Rules: 定义的业务规则，规则包含触发规则的条件和规则动作。

Facts: 要分析的情况/输入的事实数据，即与规则条件匹配以执行使用规则的数据

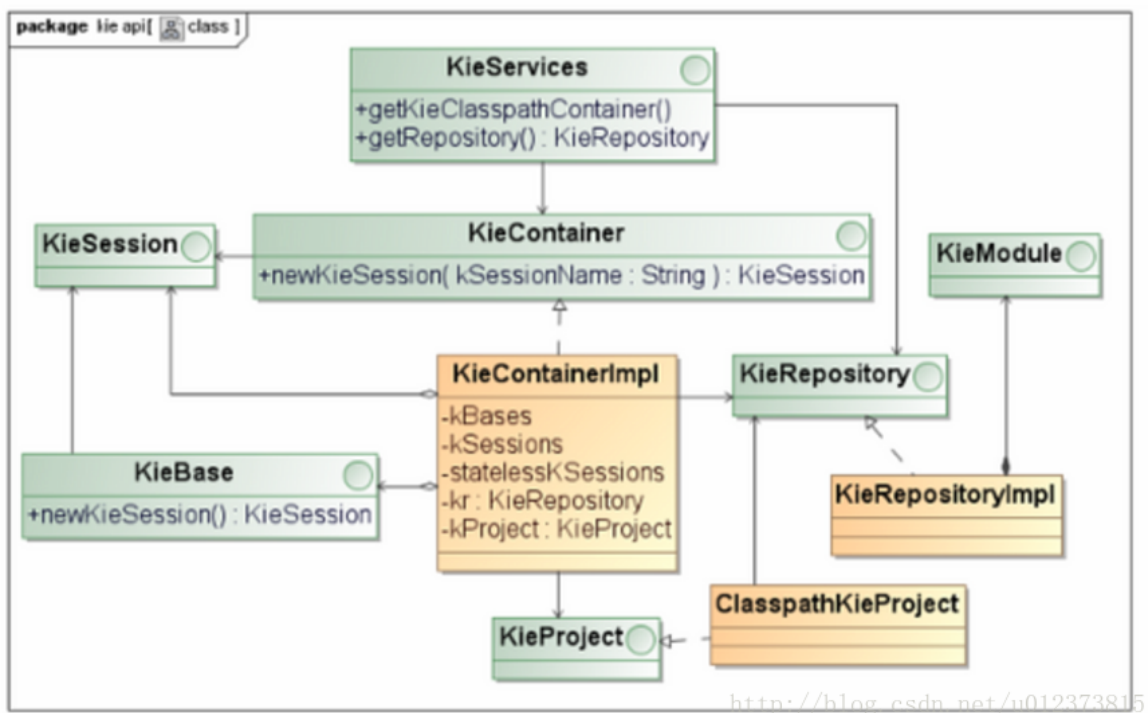
Production Memory: 用于存储规则(Rules)数据

Working Memory: 用于存储事实(Facts)数据

Patten matcher: 规则匹配器，根据输入的事实(Facts)匹配对应符合条件的规则(Rules)

Agenda: 存放匹配的规则，以准备执行

4.1.3 核心类



KieBase: 知识仓库，包含规则、流程等定义，不包含运行时数据

KieContainer: KieBase容器

KieSession: 跟drools引擎进行交互的会话，包含运行时数据，基于KieBase创建

KieModule: 相当于java中标准的maven工程，包含pom.xml、kmodule.xml、规则等必要资源

KieRepository: KieModule的容器，包含所有KieModule描述

KieServices: KIE服务的整体入口，线程安全的单例，可用来获取KieContainer、KieFileSystem等资源

4.1.4 规则描述

drools规则文件以.drl为扩展名，规则文件编写遵循drools规则语法。

规则文件整体结构

```
package package-name

imports

globals

functions

queries

rules
```

规则组成

```
rule "name"
  attributes
  when
    LHS //规则条件部分
  then
    RHS //规则执行部分
end
```

规则示例

```
package com.testspace.wb.dynamic;

###import对象
import com.testspace.wb.dynamic.Address;
import com.testspace.wb.dynamic.Item;

###定义全局变量
global com.example.rule.engine.drools.service.ProcessService processService;

###自定义函数
function String appendSuffix(String data) {
    return data + "_suffix";
}

###规则定义
rule "test-rule-1"

when
    $a:Address(name == "bj");
then
    $a.setAge(11);
    $a.setName(appendSuffix($a.getName()));
    System.out.println($a.getAddress() + "---" + $a.getName());
end

###规则定义
rule "test-rule-2"

when
    $i:Item(id == 1000);
then
    processService.process($i);
    $i.setPrice(($i.getPrice()*0.8f);
end
```

4.1.5 Workbench

4.1.5.1 基本介绍

1、workbench是KIE组件中的一个元素，是一个WEB-IDE，提供可视化的规则编辑器、可以基于插件进行扩展、生成规则包(k-jar)

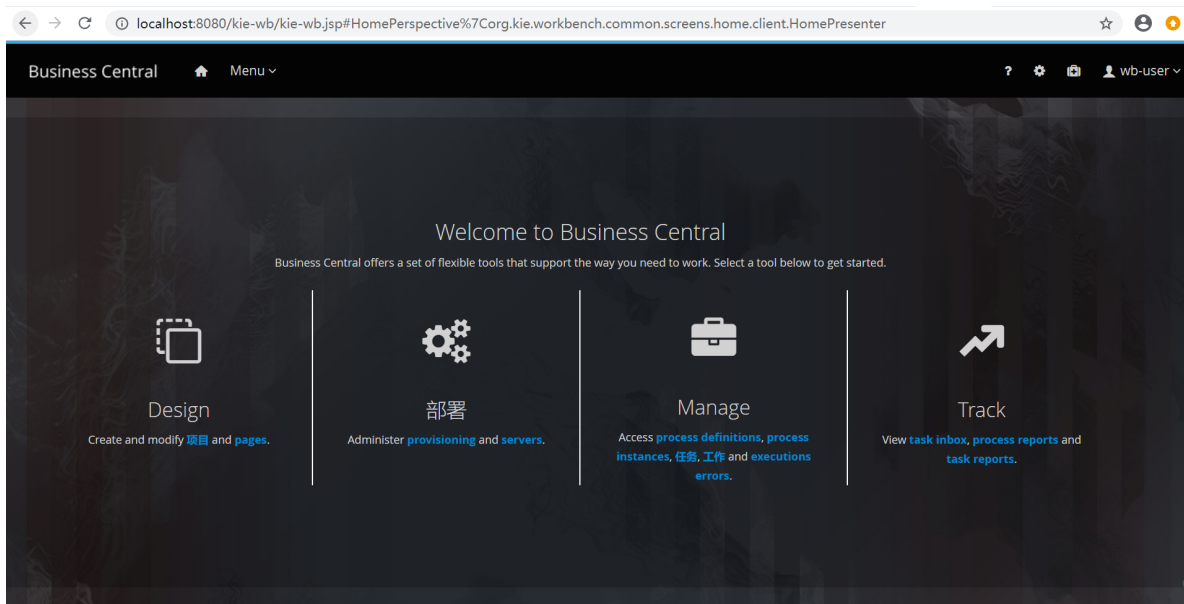
2、workbench是一个war包，最新版本只支持部署在wildfly容器中，不支持tomcat

官方描述

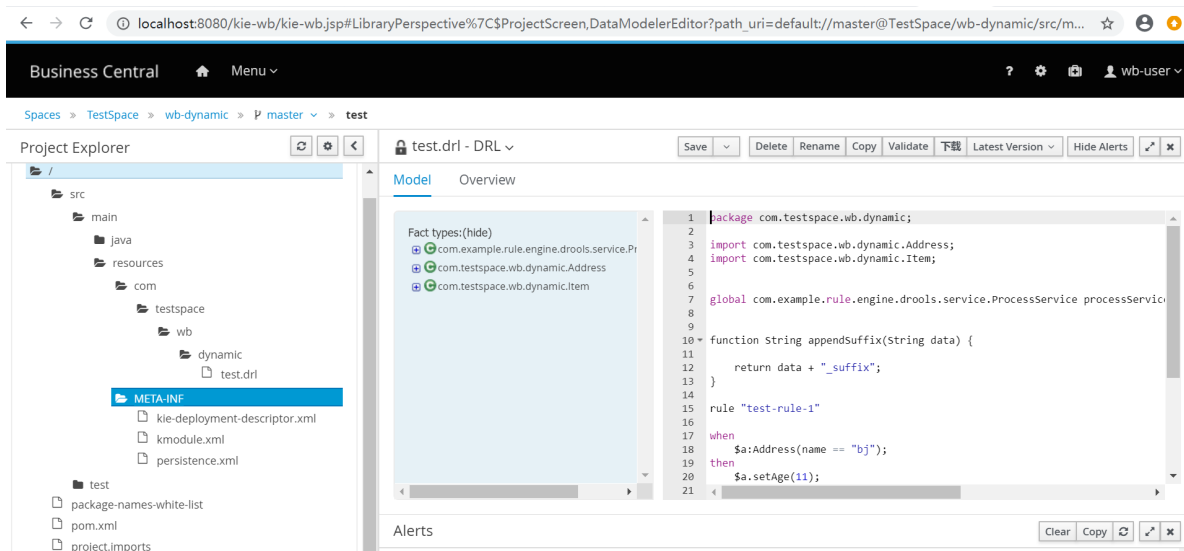
Business Central Workbench

Business Central Workbench is the web application and repository to govern Drools and jBPM assets. See [documentation](#) WildFly 19 WAR for details about installation.

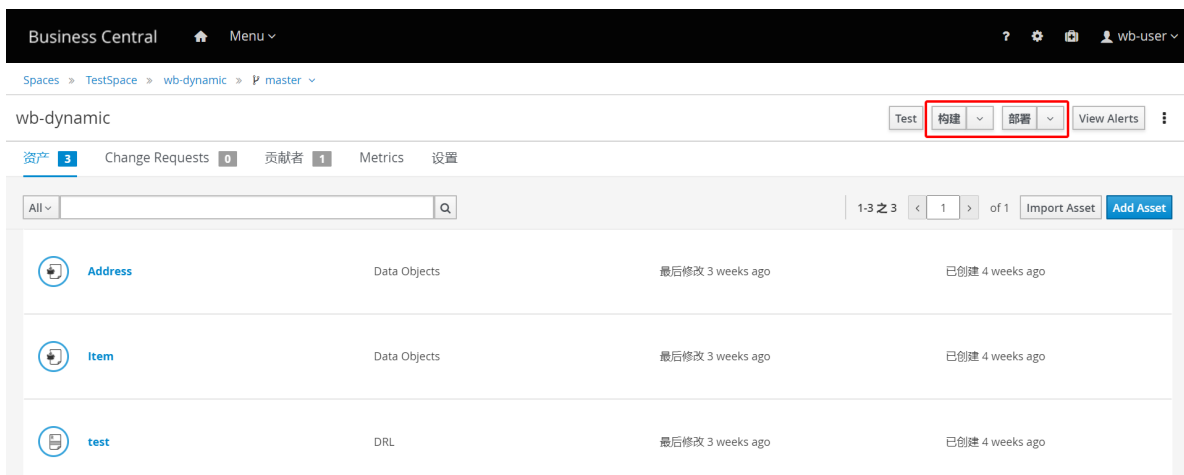
主功能界面



规则编写界面



构建&部署

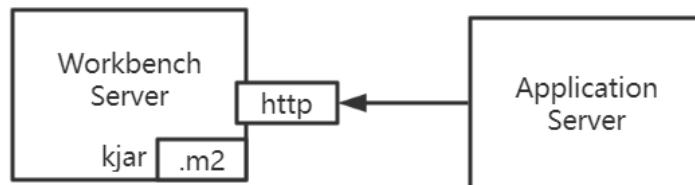


4.1.5.2 Workbench与Java项目交互

- 简单交互方式

通过HTTP方式

- 1、在Workbench Server编写规则文件，并打包生成kjar，kjar默认位于workbench Server本地maven库
- 2、workbench Server提供基于http方式下载kjar，并提供了简单的权限认证机制（用户名/密码）
- 3、应用服务端通过http下载kjar，利用drools api构建kmodule，并执行规则



基于MAVEN交互

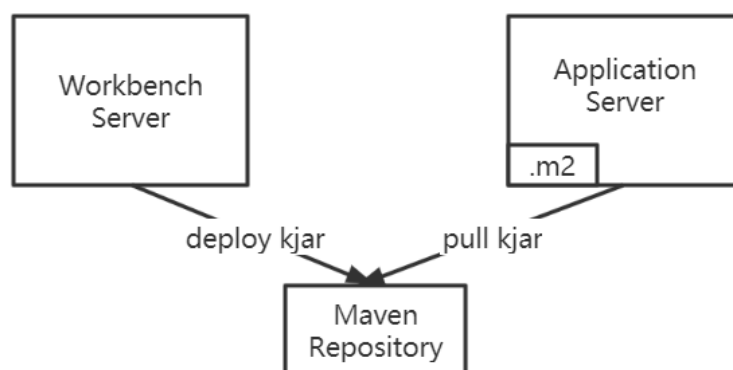
本地maven仓库方式：

- 1、workbench Server将kjar包发布到本地仓库
- 2、应用服务pom.xml文件配置workbench Server提供的仓库地址，拉取kjar到本地构建



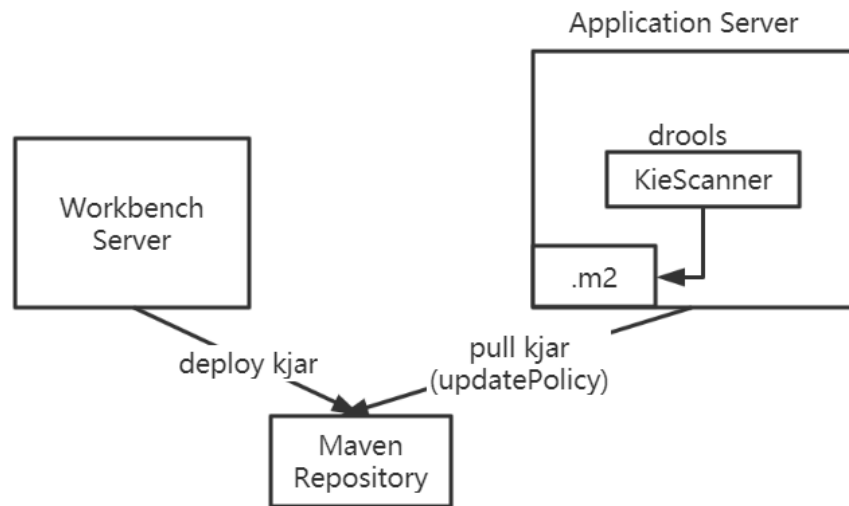
远程maven仓库方式：

- 1、在workbench Project的pom.xml中配置远程Maven仓库地址，打包时将kjar部署到远程仓库
- 2、应用服务从远程仓库拉取kjar到本地仓库



- 动态扫描方式

- 1、workbench Server将kjar包发布到远程仓库
- 2、应用服务器的maven配置定时更新策略，从远程仓库拉取最新kjar包到本地（考虑maven本地缓存问题）
- 3、应用服务通过使用drools的KieScanner定时从本地仓库的kjar包规则信息加载到内存



4.1.6 KIE Server

4.1.6.1 基本介绍

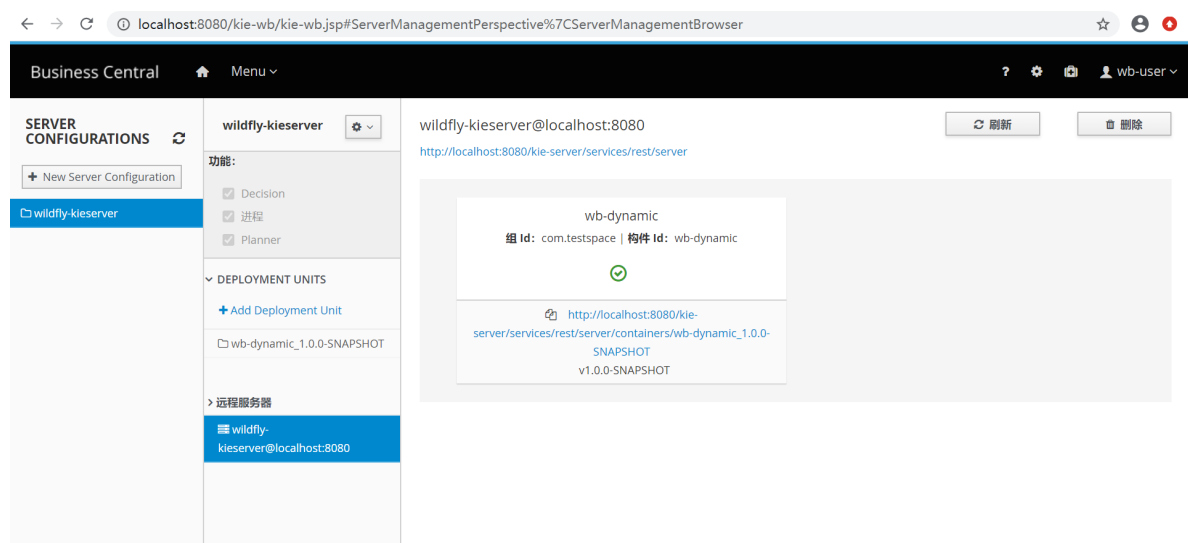
- 1、KIE-Server是一个独立的规则执行服务器，可通过REST、JMS或Java客户端API远程执行规则，跟Workbench无缝集成。
- 2、可配置化扩展能力，默认内置了BRM（规则管理）、BPM（流程管理）能力
- 3、

KIE Server简介

KIE Execution Server

Standalone execution server that can be used to remotely execute rules using REST, JMS or Java interface. Distribution zip contains WAR files for all supported containers. [Distribution ZIP](#)

Workbench 与 KIE Server集成



4.1.6.2 调用示例

- 1、初始化KieServer客户端

```

private final static String SERVER_URL = "http://localhost:8080/kie-server/services/rest/server";
private final static String USER_NAME = "kie-server";
private final static String PASSWORD = "oppo1234";
public static final String KIE_CONTAINER_ID = "wb-dynamic_1.0.0-SNAPSHOT";
private KieServicesClient kieServicesClient;
private RuleServicesClient ruleServicesClient;

@PostConstruct
public void init(){
    KieServicesConfiguration kieServicesConfiguration = KieServicesFactory.newRestConfiguration(SERVER_URL, USER_NAME, PASSWORD, timeout: 10000L);
    kieServicesConfiguration.setMarshallingFormat(MarshallingFormat.JSON);
    kieServicesClient = KieServicesFactory.newKieServicesClient(kieServicesConfiguration);
    ruleServicesClient = kieServicesClient.getServicesClient(RuleServicesClient.class);
}

```

2、调用KieServer客户端API触发规则

```

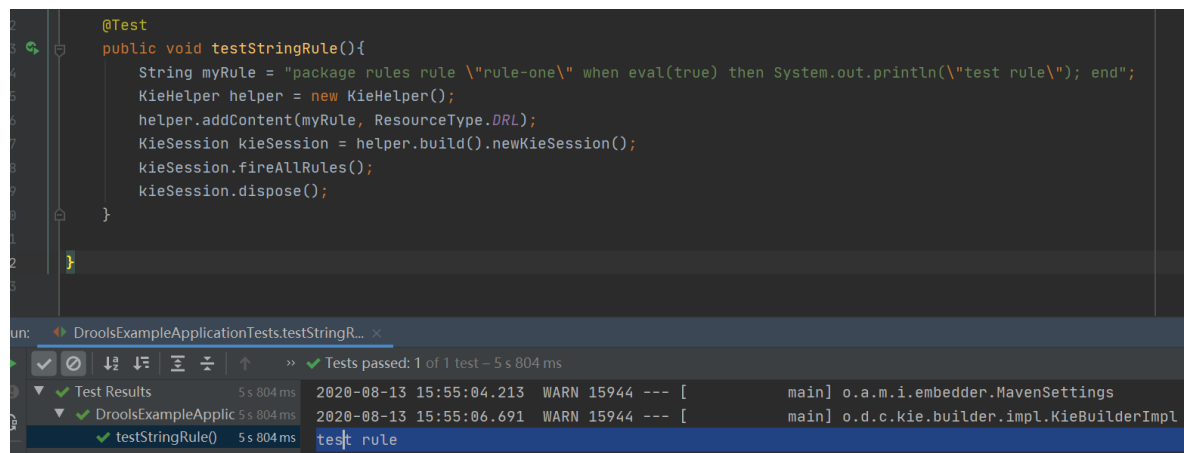
public void executeRules(){
    //1. mock fact
    Item item = new Item();
    item.setId(1000);
    item.setPrice(1000.0f);
    //2. commands build
    KieCommands kieCommands = KieServices.Factory.get().getCommands();
    List<Command> commandList = new LinkedList<>();
    commandList.add(CommandFactory.newInsert(item, outIdentifier: "item"));
    commandList.add(kieCommands.newFireAllRules());
    //3. invoke rules
    ServiceResponse<ExecutionResults> result = ruleServicesClient.executeCommandsWithResults(KIE_CONTAINER_ID,
        kieCommands.newBatchExecution(commandList, lookup: "ksessionId"));
    //4. print result
    Item itemResult = (Item) result.getResult().getValue( identifier: "item");
    System.out.println("item result:" + itemResult);
}

```

4.1.7 规则动态化实现方式

字符串方式

即将规则文件中的规则内容以字符串方式进行加载并执行



```

@Test
public void testStringRule(){
    String myRule = "package rules rule \"rule-one\" when eval(true) then System.out.println(\"test rule\"); end";
    KieHelper helper = new KieHelper();
    helper.addContent(myRule, ResourceType.DRL);
    KieSession kieSession = helper.build().newKieSession();
    kieSession.fireAllRules();
    kieSession.dispose();
}

```

Test Results: 1 of 1 test - 5 s 804 ms

testStringRule() 5 s 804 ms

test rule

指定规则文件方式

ResourceFactory.newClassPathResource:读取指定类路径下文件



```

@Test
public void testSpecificFile() throws Exception{
    Resource resource = ResourceFactory.newClassPathResource("rules\\test.drl");
    KieHelper helper = new KieHelper();
    helper.addResource(resource);
    KieSession kieSession = helper.build().newKieSession();
    kieSession.fireAllRules();
    kieSession.dispose();
}

```

Test Results: 1 of 1 test - 5 s 804 ms

testSpecificFile() 5 s 804 ms

test rule

ResourceFactory.newFileResource:读取绝对路径下文件

通过KieFileSystem可读取路径下多个文件

通过代码构建Kjar

类似于通过workbench构建kjar包，只是本方式通过Drools API方式动态构建kjar包（kjar对应Drools中的InternalKieModule对象）

示例代码：

```
@Test
public void testDynamicKjar(){
    // 规则内容
    String ruleContent = "package rules rule \"rule-one\" when eval(true) then System.out.println(\"test rule\"); end";
    KieServices kieServices = KieServices.Factory.get();
    // 指定构建kjar包的GAV
    ReleaseId releaseId = kieServices.newReleaseId( groupId: "rule", artifactId: "test", version: "1.0.0");
    // 构建kjar
    InternalKieModule kJar = buildKjar(ruleContent);
    // 将kjar加入KieRepository仓库
    KieRepository kieRepository = kieServices.getRepository();
    kieRepository.addKieModule(kJar);
    // 构建跟drools引擎交互的会话session
    KieContainer kieContainer = kieServices.newKieContainer(releaseId);
    KieSession kieSession = kieContainer.newKieSession();
    // 触发规则
    kieSession.fireAllRules();
    // 资源释放
    kieSession.dispose();
}
```

workbench + kie-server

在workbench编辑规则，并将规则发布到kie-server中，java项目通过kie-server api触发规则调用

动态扫描方式

workbench将kjar包发布到远程仓库，java项目配置maven更新策略从远程仓库拉取kjar包到本地仓库，并通过drools的KieScanner进行扫描加载

示例代码：

```
public void initKieScanner(){
    KieServices kieServices = KieServices.Factory.get();
    ReleaseId releaseId = kieServices.newReleaseId( groupId: "com.testspace", artifactId: "wb-dynamic", version: "1.0.0-SNAPSHOT");
    KieContainer kieContainer = kieServices.newKieContainer(releaseId);
    KieScanner kieScanner = kieServices.newKieScanner(kieContainer);
    kieScanner.addListener(new KieScannerEventListener(){
        @Override
        public void onKieScannerStatusChangeEvent(KieScannerStatusChangeEvent changeEvent) {
            System.out.println(changeEvent);
        }
        @Override
        public void onKieScannerUpdateResultsEvent(KieScannerUpdateResultsEvent updateResultsEvent) {
            System.out.println(updateResultsEvent);
        }
    });
    kieScanner.start( pollingInterval: 1000L);
}
```

4.2 QLExpress

4.2.1 基本介绍

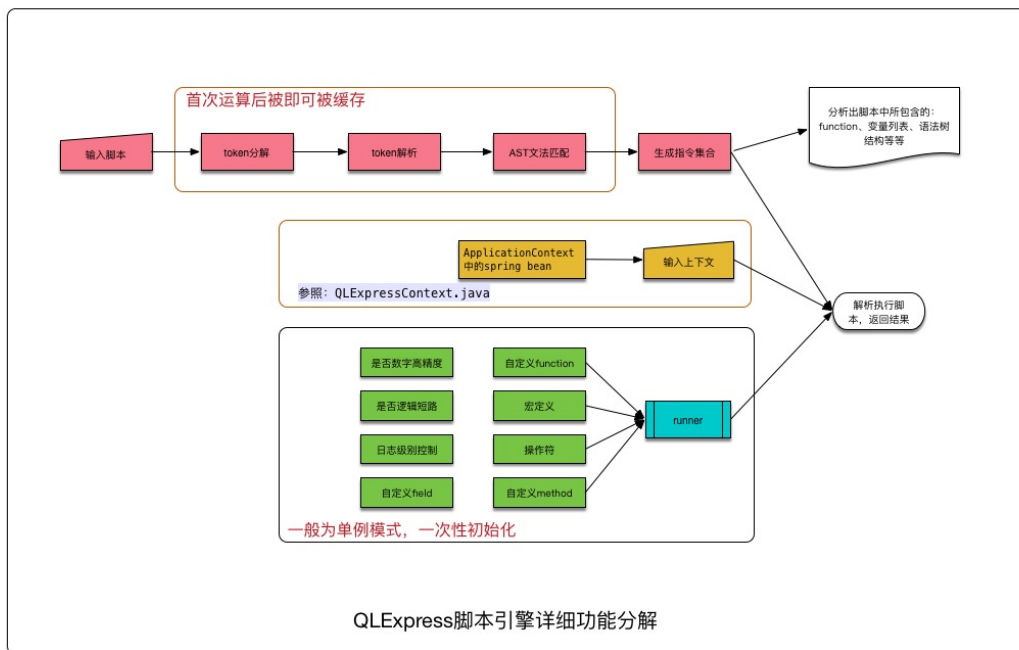
Github

地址: <https://github.com/alibaba/QLExpress>

star: 2.2k

- 1、弱类型语言，支持普通java语法，不支持try/catch、不支持lambda、不支持泛型
- 2、代码精细，依赖少，jar包250k
- 3、支持自定义函数、自定义操作符
- 4、支持java类绑定以及对象method绑定
- 5、线程安全，引擎运算临时变量都是threadlocal类型
- 6、支持指令集缓存，比较耗时的脚本编译过程可缓存在本地HashMap
- 7、脚本执行支持超时设置
- 8、可配置禁止调用不安全系统API，如System.exit
- 9、与Spring框架无缝集成，自定义扩展的IExpressContext可以put任何变量，如Spring Bean

4.2.2 功能模块



4.2.3 关键代码

解析表达式，得到指令集InstructionSet

```

ExpressRunner.java
548 * @return
549 * @throws Exception
550 */
551 public Object execute(String expressString, IExpressContext<String,Object> context,
552     List<String> errorList, boolean isCache, boolean isTrace, Log aLog)
553     throws Exception {
554     InstructionSet parseResult = null;
555     if (isCache == true) {
556         parseResult = expressInstructionSetCache.get(expressString);
557         if (parseResult == null) {
558             synchronized (expressInstructionSetCache) {
559                 parseResult = expressInstructionSetCache.get(expressString);
560                 if (parseResult == null) {
561                     parseResult = this.parseInstructionSet(expressString);
562                     expressInstructionSetCache.put(expressString,
563                         parseResult);
564                 }
565             }
566         }
567     } else {
568         parseResult = this.parseInstructionSet(expressString);
569     }
570     return InstructionSetRunner.executeOuter( runner: this, parseResult, this.loader, context, errorList,
571         isTrace, isCatchException: false, aLog, isSupportDynamicFieldName: false);
572 }

```

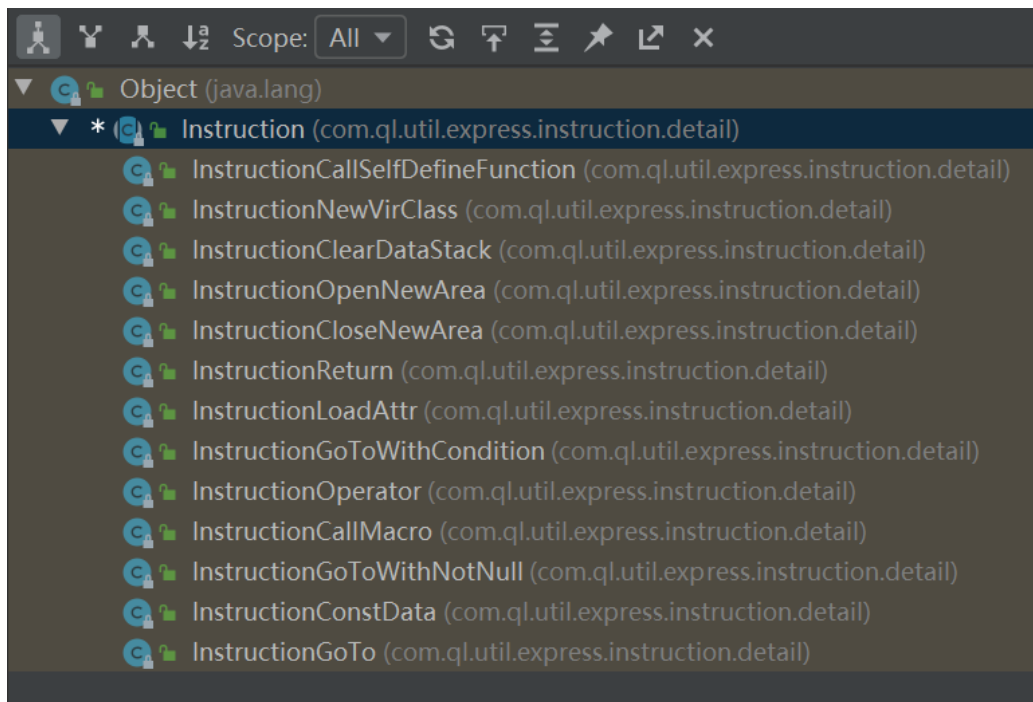
构建运行时环境RunEnvironment，执行指令集中各指令

```

InstructionSetRunner.java
49
50 public static Object execute(InstructionSet set,
51     InstructionSetContext context, List<String> errorList, boolean isTrace, boolean isCatchException,
52     boolean isReturnLastData, Log aLog) throws Exception {
53
54     RunEnvironment environmen = null;
55     Object result = null;
56     environmen = OperateDataCacheManager.getRunEnvironment(set,
57         (InstructionSetContext) context, isTrace);
58     try {
59         CallResult tempResult = set.execute(environmen, context, errorList,
60             isReturnLastData, aLog);
61         if (tempResult.isExit() == true) {
62             result = tempResult.getReturnValue();
63         }
64     } catch (Exception e) {
65         if (isCatchException == true) {
66             if (aLog != null) {
67                 aLog.error(e.getMessage(), e);
68             } else {
69                 log.error(e.getMessage(), e);
70             }
71         } else {
72             throw e;
73         }
74     }
75 }

```

QLExpress中定义的指令



4.3 Aviator

4.3.1 基本介绍

github

<https://github.com/killme2008/aviator>

特性

- 1、定位：高性能、轻量级、寄宿于 JVM 之上的脚本语言，主要用于各种表达式的动态求值
- 2、编译运行：直接将表达式编译成Java字节码，交给JVM执行，编译结果可配置是否缓存
- 3、轻量化的模块系统，方便将一系列相关函数组织进行管理
- 4、完整的脚本语法支持，支持闭包、lambda、异常处理、大整数/高精度计算
- 5、支持导入java静态方法、实例方法，反射调用

4.3.2 功能模块



Lexer: 分词器
Parser: 解析器
CodeGenerator: 代码生成器

运行模式

1、执行速度优先（默认模式）

`AviatorEvaluator.setOption(Options.OPTIMIZE_LEVEL, AviatorEvaluator.EVAL)`

编译时花更多时间做优化，如常量折叠、公共变量提取优化。适合长期运行的表达式。

2、编译速度优先

`AviatorEvaluator.setOption(Options.OPTIMIZE_LEVEL, AviatorEvaluator.COMPILE)`

以编译速度优先，不做任何编译优化，牺牲一定的运行时性能。适合需要频繁编译表达式场景。

4.3.3 关键代码

入口类: `AviatorEvaluatorInstance`

表达式编译:

- 1、构建分词器 `ExpressionLexer`
- 2、根据代码优化配置选项构建对应的代码生成器 `CodeGenerator`
- 3、构建解析器 `ExpressionParser`
- 4、执行解析

```

private Expression innerCompile(final String expression, final boolean cached) {
    ExpressionLexer lexer = new ExpressionLexer(instance: this, expression);
    CodeGenerator codeGenerator = new CodeGenerator(cached);
    ExpressionParser parser = new ExpressionParser(instance: this, lexer, codeGenerator);
    Expression exp = parser.parse();
    if (getOptionValue(Options.TRACE_EVAL).bool) {
        ((BaseExpression) exp).setExpression(expression);
    }
    return exp;
}

```

字节码生成

- 1、生成 JVM 字节码，并动态生成匿名 Class
- 2、实例化 匿名Class，生成最终的Expression对象

```

ASMCodeGenerator.java
790 */
791 @Override
792 public Expression getResult(final boolean unboxObject) {
793     end(unboxObject);
794
795     byte[] bytes = this.classWriter.toByteArray();
796     try {
797         Class<?> defineClass =
798             ClassDefiner.defineClass(this.className, Expression.class, bytes, this.classLoader);
799         Constructor<?> constructor =
800             defineClass.getConstructor(AviatorEvaluatorInstance.class, List.class, SymbolTable.class);
801         ClassExpression exp = (ClassExpression) constructor.newInstance(this.instance,
802             new ArrayList<String>(this.varTokens.keySet()), this.symbolTable);
803         exp.setLambdaBootstraps(this.lambdaBootstraps);
804         exp.setFuncsArgs(this.funcsArgs);
805         return exp;
806     } catch (ExpressionRuntimeException e) {
807         throw e;
808     } catch (Throwable e) {
809         if (e.getCause() instanceof ExpressionRuntimeException) {
810             throw (ExpressionRuntimeException) e.getCause();
811         }
812         throw new CompileExpressionErrorException("define class error", e);
813     }
814 }

```

debug info:

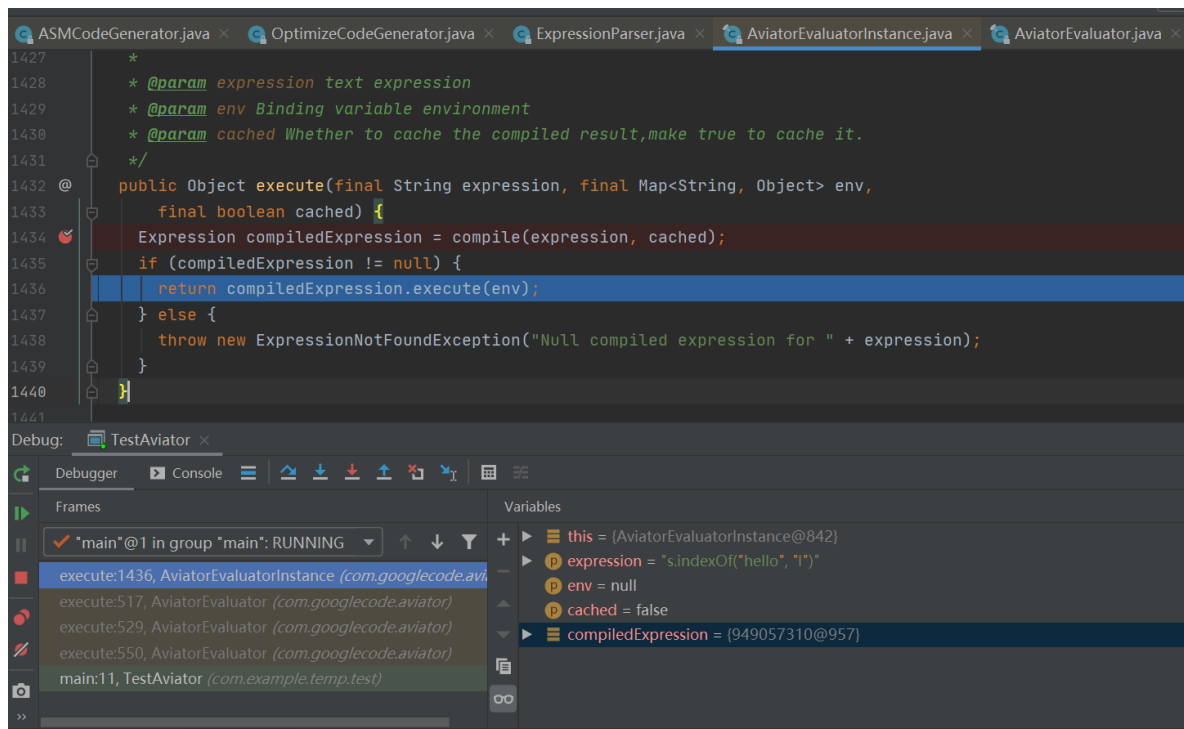
```

Variables
+ ▶ this = {ASMCodeGenerator@888}
- ▶ unboxObject = true
  ▶ bytes = {byte[1369]@907}
  ▶ defineClass = {Class@941} *class Script_1597216142999_0/949057310" ... Navigate
  ▶ constructor = {Constructor@946} *public Script_1597216142999_0/949057310(com.googlecode.aviator.AviatorEvaluatorInstan... View
  ▶ exp = {949057310@957}
  ▶ this.funcsArgs = null

```

执行结果

传入表达式上下文参数对象，并使用生成的Expression实例求解表达式



4.3.4 QLEXPRESS VS Aviator简单性能测试

测试代码

求值表达式“a+b*3”，进行1w次计算，对比QLEXPRESS与Aviator平均耗时

```

@BenchmarkMode(Mode.AverageTime)
@Threads(1)
@Fork(1)
@Measurement(iterations = 10, timeUnit= TimeUnit.MILLISECONDS)
@Warmup(iterations = 0)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
public class TestQLEExpress {

    @Benchmark
    public void testQLENoCache() throws Exception{
        for(int i=0; i<1000;i++) {
            ExpressRunner runner = new ExpressRunner();
            DefaultContext<String, Object> context = new DefaultContext<>();
            context.put("a", 1);
            context.put("b", 2);
            context.put("c", 3);
            String express = "a+b*c";
            Object r = runner.execute(express, context, errorList: null, isCache: false, isTrace: false);
        }
    }

    @Benchmark
    public void testQLEWithCache() throws Exception {
        for(int i=0; i<1000;i++) {
            ExpressRunner runner = new ExpressRunner();
            DefaultContext<String, Object> context = new DefaultContext<>();
            context.put("a", 1);
            context.put("b", 2);
            context.put("c", 3);
            String express = "a+b*c";
            Object r = runner.execute(express, context, errorList: null, isCache: true, isTrace: false);
        }
    }

    @Benchmark
    public void testAviator() {
        for(int i=0; i<1000;i++) {
            String express = "a+b*c";
            Expression expression = AviatorEvaluator.compile(express);
            Map<String, Object> params = ImmutableMap.of( k1: "a", v1: 1, k2: "b", v2: 2, k3: "c", v3: 3);
            Object result = expression.execute(params);
        }
    }

    public static void main(String[] args) throws RunnerException {
        Options opt = new OptionsBuilder()
            .include(TestQLEExpress.class.getSimpleName())
            .build();
        new Runner(opt).run();
    }
}

```

测试结果

在简单表达式求值方面，aviator比QLEExpress性能较好

Benchmark	Mode	Samples	Score	Score error	Units
c.x.t.q.TestQLEExpress.testAviator	avgt	10	425.177	157.542	ms/op
c.x.t.q.TestQLEExpress.testQLENoCache	avgt	10	1654.250	460.747	ms/op
c.x.t.q.TestQLEExpress.testQLEWithCache	avgt	10	1428.294	301.341	ms/op

4.4 uRule

4.4.1 基本介绍

URULE是一款基于RETE算法的纯Java规则引擎，提供规则集、决策表、决策树、评分卡，规则流等各种规则表现工具及基于网页的可视化设计器，可快速开发出各种复杂业务规则。

Urule商业版与开源版功能对比

URULE PRO版与开源版主要功能比较		
特性	URULE PRO版	URULE开源版
向导式决策集	有	有
脚本式决策集	有	有
决策树	有	有
决策流	有	有
决策表	有	有
交叉决策表	有	无
复杂评分卡	有	无
文件名、项目名重构	有	无
参数名、变量常量名重构	有	无
Excel决策表导入	有	无
规则集模版保存与加载	有	无
中文项目名和文件名支持	有	无
服务器推送知识包到客户端功能的支持	有	无
知识包优化与压缩的支持	有	无
客户端服务器模式下大知识包的推拉支持	有	无
规则集中执行组的支持	有	无
规则流中所有节点向导式条件与动作配置的支持	有	无
循环规则多循环单元支持	有	无
循环规则中无条件执行的支持	有	无
导入项目自动重命名功能	有	无
规则树构建优化	有	无
对象查找索引支持	有	无
规则树中短路计算的支持	有	无
规则条件冗余计算缓存支持	有	无
基于方案的批量场景测试功能	有	无
知识包调用监控	有	无
更为完善的文件读写权限控制	有	无
知识包版本控制	有	无
SpringBean及Java类的热部署	有	无
技术支持	有	无

5 技术选型

5.1 优缺点对比

技术	优点	缺点
Drools	1、策略规则和执行逻辑解耦方便维护 2、提供了具备完善功能的规则管理、规则发布后台 3、提供了可独立部署的规则执行服务kie-server，可分布式部署 4、可与Spring集成	1、某些特定场景下，其特有的规则描述语言，对于业务分析师学习成本比较高，最终还是需要靠开发人员去维护规则 2、规则的语法仅适合扁平的规则，对于嵌套条件语义（then里嵌套when...then子句）的规则，只能将条件进行笛卡尔积组合以后进行配置，不利于维护 3、规则加载比较耗时，不适合规则较多且变化频繁的场景 4、Drools利用空间换性能，比较耗内存 5、WorkBench+kieServer方式更多适合于公司内部规则变化不频繁的场景，如员工绩效评定系统等
Urule	1、国产 2、提供规则集、决策表、交叉决策表（决策矩阵）、决策树、评分卡、复杂评分卡、规则流等八种类型的业务规则设计工具 3、提供在线体验平台	1、分为开源版本、商业版本，开源版本支持功能较少 2、开源项目的代码贡献者较少 3、小公司有使用，大公司较少使用
Aviator	1、编译执行，将表达式编译成java字节码交由JVM执行，执行效率高 2、2020年发布的5.0版本，升级成AviatorScript脚本语言 3、支持绑定Java静态/实例方法	1、对java语法支持比较少，主要用于简单表达式求值、判断 2、内置了很多函数库，需要额外的熟悉成本
QLEExpress	1、支持与Spring无缝集成，可将任何变量注入到Context中，如Spring Bean 2、支持绑定Java静态/实例方法 3、提供了较好的语法糖，支持大部分java语法，易上手	1、解释执行，执行速度弱于aviator 2、java语法支持不完整，不支持try/catch、lambda等语法，可通过绑定java类或对象方法规避

5.2 OPPO技术团队选型案例

5.2.1 业务安全风险团队风控系统

技术选型

QLEExpress

考虑点

综合开源程度、社区活跃度、易用性，选择QLEExpress

Drools易用性差而放弃

联系人

曾德康

5.2.2 金融业务团队风控中台

技术选型

Drools（如果重新选型，会更倾向于其他脚本语言）

实现细节

- 1、没有使用Drools自带的Workbench管理规则，自建规则管理后台，将规则翻译为drools规则
- 2、没有使用Drools提供的KieServer，通过封装KIE引擎API

联系人

陈志新

5.3 IOT物联网平台需求

应用场景

智能家居规则联动

特点

- 1、规则创建面向互联网用户，用户量大(百/千万级)
- 2、规则数量多、变化频繁
- 3、规则条件有定时触发、设备触发，有比较精确的时间要求

技术选型

倾向基于QLExpress做封装