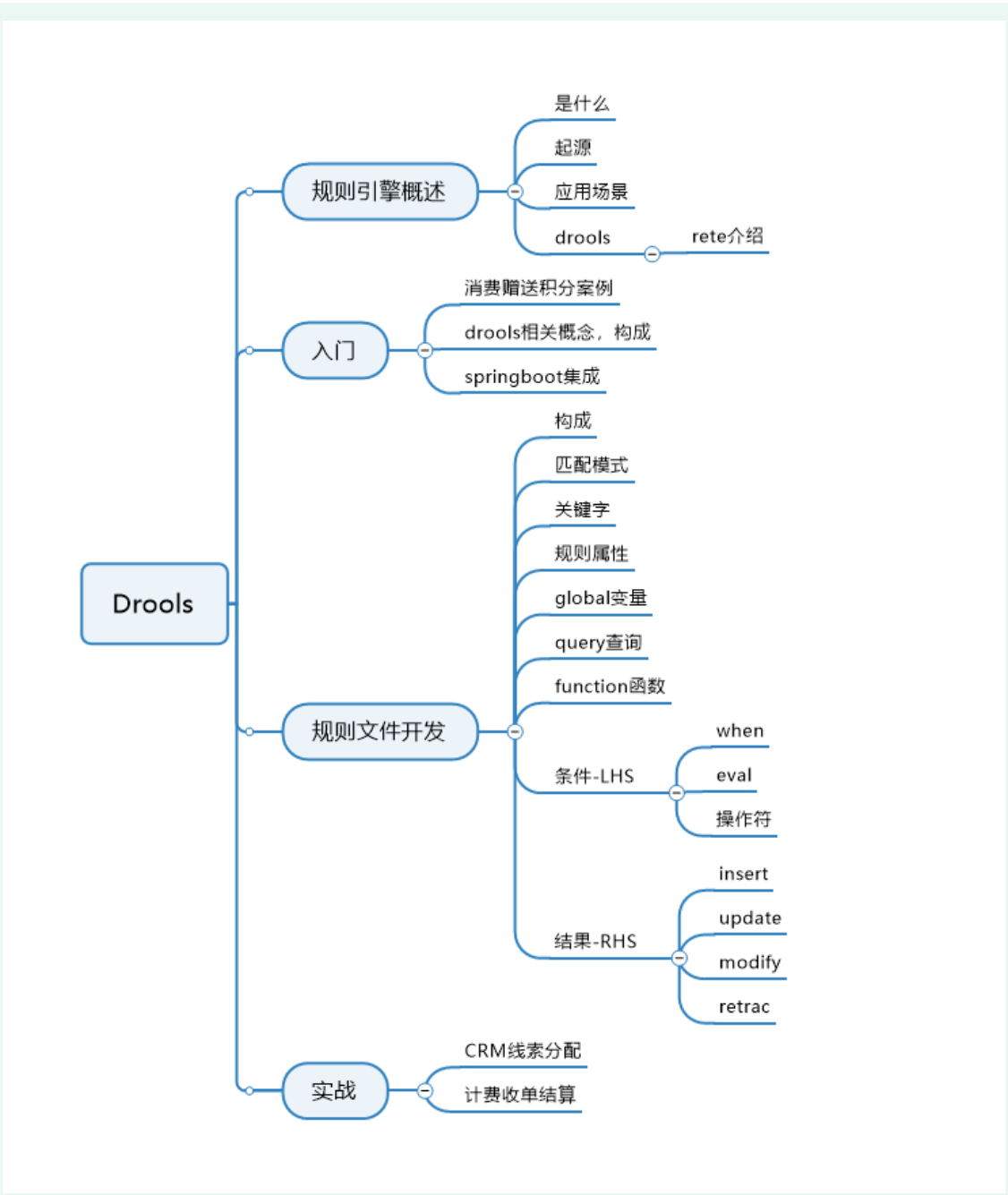
规则引擎-drools

规则引擎



1 .场景

1.1需求

```
商城系统消费赠送积分

100元以下, 不加分
100元-500元 加100分
500元-1000元 加500分
1000元 以上 加1000分
......
```

1.2传统做法

1.2.1 if...else

```
if (order.getAmout() <= 100){
    order.setScore(0);
    addScore(order);
}else if(order.getAmout() > 100 && order.getAmout() <= 500){
    order.setScore(100);
    addScore(order);
}else if(order.getAmout() > 500 && order.getAmout() <= 1000){
    order.setScore(500);
    addScore(order);
}else{
    order.setScore(1000);
    addScore(order);
}
```

1.2.2 策略

```
interface Strategy {
    addScore(int num1,int num2);
}

class Strategy1 {
    addScore(int num1);
}
.....................
interface StrategyN {
    addScore(int num1);
}

class Environment {
    private Strategy strategy;

    public Environment(Strategy strategy) {
        this.strategy = strategy;
    }

    public int addScore(int num1) {
        return strategy.addScore(num1);
    }
}
```

1.2.3 问题?

以上解决方法问题思考：
如果需求变更，积分层次结构增加，积分比例调整？
数据库？

遇到的问题瓶颈：
第一，我们要简化`if else`结构,让业务逻辑和数据分离！
第二，分离出的业务逻辑必须要易于编写，至少单独编写这些业务逻辑，要比写代码快！
第三，分离出的业务逻辑必须要比原来的代码更容易读懂！
第四，分离出的业务逻辑必须比原来的易于维护，至少改动这些逻辑，应用程序不用重启！

2.是什么

2.1概念

规则引擎由**推理引擎**发展而来，是一种嵌入在应用程序中的组件，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

需要注意的是规则引擎并不是一个具体的技术框架，而是指的一类系统，即业务规则管理系统。目前市面上具体的规则引擎产品有：drools、VisualRules、iLog等

在很多企业的 IT 业务系统中，经常会有大量的业务规则配置，而且随着企业管理者的决策变化，这些业务规则也会随之发生更改。为了适应这样的需求，我们的 IT 业务系统应该能快速且低成本的更新。适应这样的需求，一般的作法是将业务规则的配置单独拿出来，使之与业务系统保持低耦合。目前，实现这样的功能的程序，已经被开发成为规则引擎。

2.2 起源

2.3 原理--基于 rete 算法的规则引擎

2.3.1 原理

在 AI 领域，产生式系统是一个很重要的理论，产生式推理分为正向推理和逆向推理产生式，其规则的一般形式是：IF 条件 THEN 操作。rete 算法是实现产生式系统中正向推理的高效模式匹配算法，通过形成一个 rete 网络进行模式匹配，利用基于规则的系统的时间冗余性和结构相似性特征，提高系统模式匹配效率

正向推理（Forward-Chaining）和反向推理（Backward-Chaining）

（1）正向推理也叫演绎法，由事实驱动，从一个初始的事实出发，不断地从应用规则得出结论。首先在候选队列中选择一条规则作为启用规则进行推理，记录其结论作为下一步推理的证据。如此重复这个过程，直到再无可用规则可被选用或者求得了所要求的解为止。

（2）反向推理也叫归纳法，由目标驱动，首先提出某个假设，然后寻找支持该假设的证据，若所需的证据都能找到，说明原假设是正确的，若无论如何都找不到所需要的证据，则说明原假设不成立，此时需要另作新的假设。

2.3.2 rete算法

Rete 算法最初是由卡内基梅隆大学的 Charles L.Forgy 博士在 1974 年发表的论文中所阐述的算法，该算法提供了专家系统的一个高效实现。自 Rete 算法提出以后，它就被用到一些大型的规则系统中，像 ILog、Jess、JBoss Rules 等都是基于 RETE 算法的规则引擎 。

Rete 在拉丁语中译为”net”，即网络。Rete 匹配算法是一种进行大量模式集合和大量对象集合间比较的高效方法，通过网络筛选的方法找出所有匹配各个模式的对象和规则。

其核心思想是将分离的匹配项根据内容动态构造匹配树，以达到显著降低计算量的效果。Rete 算法可以被分为两个部分：规则编译和规则执行 。当 Rete 算法进行事实的断言时，包含三个阶段：匹配、选择和执行，称做 match-select-act cycle。

2.4 规则引擎应用场景

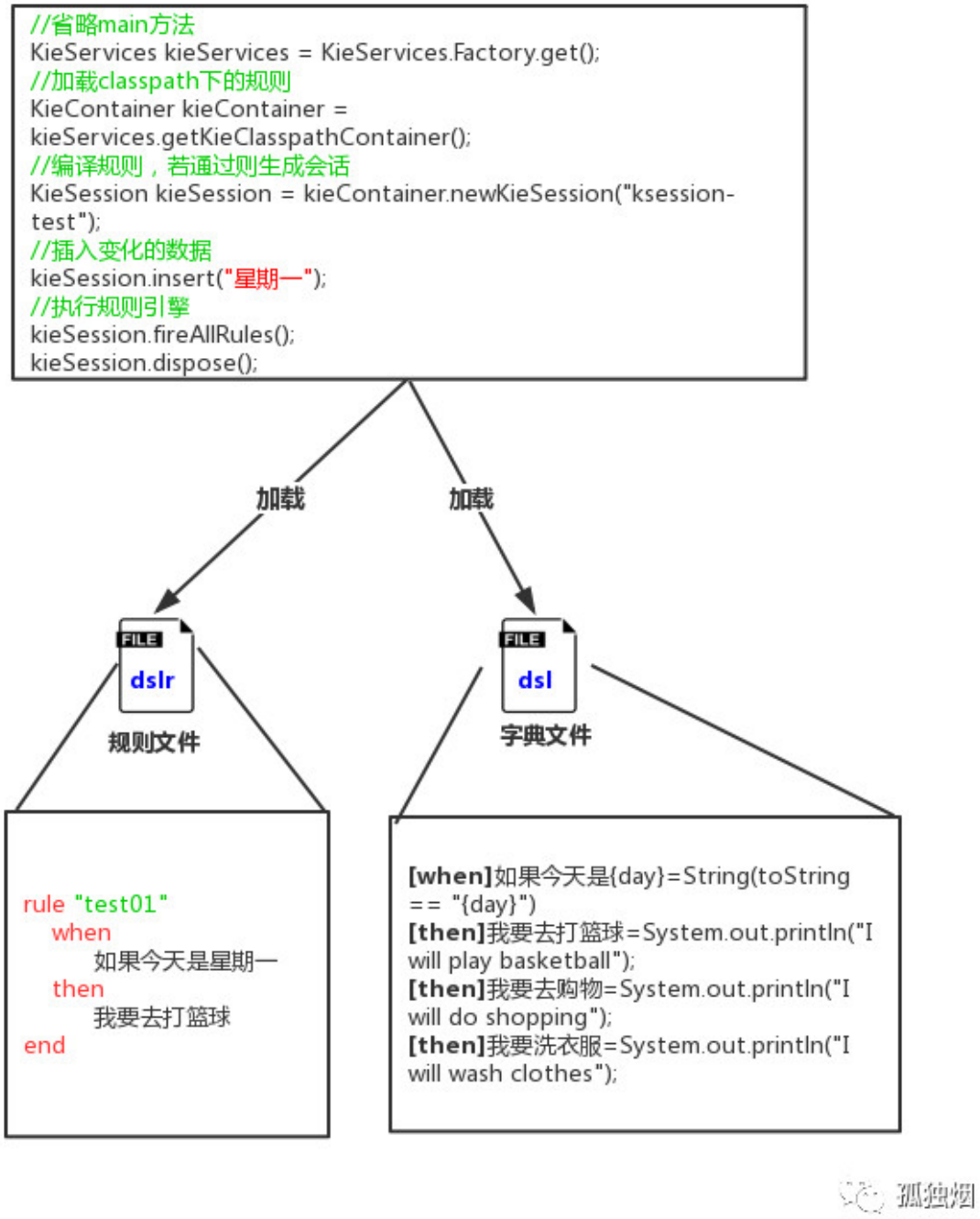
业务领域	示例
财务决策	贷款发放，征信系统
库存管理	及时的供应链路
票价计算	航空，传播，火车及其他公共汽车运输
生产采购系统	产品原材料采购管理
风控系统	风控规则计算
促销平台系统	满减，打折，加价购

2.5 Drools 介绍

Drools 具有一个易于访问企业策略、易于调整以及易于管理的开源业务 **规则引擎**，符合业内标准，速度快、效率高。业务分析师或审核人员可以利用它轻松查看业务规则，从而检验已编码的规则是否执行了所需的业务规则。其前身是 Codehaus 的一个开源项目叫 Drools，后被纳入 JBoss 门下，更名为 JBoss Rules，成为了 JBoss 应用服务器的规则引擎。

Drools 被分为两个主要的部分：编译和运行时。编译是将规则描述文件按 ANTLR 3 语法进行解析，对语法进行正确性的检查，然后产生一种中间结构“descr”，descr 用 AST 来描述规则。目前，Drools 支持四种规则描述文件，分别是：drl 文件、 xls 文件、brl 文件和 dsl 文件，其中，常用的描述文件是 drl 文件和 xls 文件，而 xls 文件更易于维护，更直观，更为被业务人员所理解。运行时是将 AST传到 PackageBuilder，由 PackagBuilder来产生 RuleBase，它包含了一个或多个 Package 对象。

3 .消费赠送积分案例



上图为实际用法：

3.1 第一步： 创建工程，引入jar

> 由于当前java开发，普通使用springboot ， 本课程以springboot为基本框架演示

jar 依赖，注意，排除spring相关依赖

```
<!-- 规则引擎 -->
        <dependency>
            <groupId>org.kie</groupId>
```

```
            <artifactId>kie-spring</artifactId>
            <version>${drools.version}</version>
            <exclusions>
                <exclusion>
                    <groupId>org.springframework</groupId>
                    <artifactId>spring-tx</artifactId>
                </exclusion>
                <exclusion>
                    <groupId>org.springframework</groupId>
                    <artifactId>spring-beans</artifactId>
                </exclusion>
                <exclusion>
                    <groupId>org.springframework</groupId>
                    <artifactId>spring-core</artifactId>
                </exclusion>
                <exclusion>
                    <groupId>org.springframework</groupId>
                    <artifactId>spring-context</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
```

3.2 创建 drools 自动配置类

drools 在spring 或者springboot中用法一样，其实就是创建好一些bean

```
package com.mashibing.drools.config;

import org.kie.api.KieBase;
import org.kie.api.KieServices;
import org.kie.api.builder.*;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;
import org.kie.internal.io.ResourceFactory;
import org.kie.spring.KModuleBeanFactoryPostProcessor;
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.Resource;
import org.springframework.core.io.support.PathMatchingResourcePatternResolver;
import org.springframework.core.io.support.ResourcePatternResolver;

import java.io.IOException;


/**
 * <p> 规则引擎自动配置类 </p>
 * @author 孙志强
 * @date 2019/9/10 11:20
 */
@Configuration
public class DroolsAutoConfiguration {

    private static final String RULES_PATH = "rules/";
```

```
    private KieServices getKieServices() {
        return KieServices.Factory.get();
    }

    @Bean
    @ConditionalOnMissingBean(KieFileSystem.class)
    public KieFileSystem kieFileSystem() throws IOException {
        KieFileSystem kieFileSystem = getKieServices().newKieFileSystem();
        for (Resource file : getRuleFiles()) {
            kieFileSystem.write(ResourceFactory.newClassPathResource(RULES_PATH +
file.getFilename(), "UTF-8"));
        }
        return kieFileSystem;
    }

    private Resource[] getRuleFiles() throws IOException {
        ResourcePatternResolver resourcePatternResolver = new
PathMatchingResourcePatternResolver();
        return resourcePatternResolver.getResources("classpath*:" + RULES_PATH +
"**/*.*");
    }

    @Bean
    @ConditionalOnMissingBean(KieContainer.class)
    public KieContainer kieContainer() throws IOException {
        final KieRepository kieRepository = getKieServices().getRepository();

        kieRepository.addKieModule(() -> kieRepository.getDefaultReleaseId());

        KieBuilder kieBuilder = getKieServices().newKieBuilder(kieFileSystem());
        kieBuilder.buildAll();

        KieContainer kieContainer =
getKieServices().newKieContainer(kieRepository.getDefaultReleaseId());

        return kieContainer;
    }

    @Bean
    @ConditionalOnMissingBean(KieBase.class)
    public KieBase kieBase() throws IOException {
        return kieContainer().getKieBase();
    }

}
```

3.2订单实体类

```
@Data
@Accessors(chain = true)
public class Order {
```

```
    /**
     * 订单原价金额
     */
    private int price;

    /**
     *下单人
     */
    private User user;

    /**
     *积分
     */
    private int score;

    /**
     * 下单日期
     */
    private Date bookingDate;
}
```

3.3规则引擎文件

```
package rules

import com.mashibing.drools.entity.Order

rule "zero"
    no-loop true
    lock-on-active true
    salience 1
    when
        $s : Order(amout <= 100)
    then
        $s.setScore(0);
        update($s);
end

rule "add100"
    no-loop true
    lock-on-active true
    salience 1
    when
        $s : Order(amout > 100 && amout <= 500)
    then
        $s.setScore(100);
        update($s);
end

rule "add500"
    no-loop true
    lock-on-active true
    salience 1
    when
        $s : Order(amout > 500 && amout <= 1000)
```

```
    then
        $s.setScore(500);
        update($s);
end

rule "add1000"
    no-loop true
    lock-on-active true
    salience 1
    when
        $s : Order(amout > 1000)
    then
        $s.setScore(1000);
        update($s);
end
```

3.4客户端

```
/**
 * 需求
 * 计算额外积分金额 规则如下: 订单原价金额
 * 100以下, 不加分
 * 100-500 加100分
 * 500-1000 加500分
 * 1000 以上 加1000分
 */
public class DroolsOrderTests extends DroolsApplicationTests {
    @Resource
    private KieContainer kieContainer;

    @Test
    public void Test() throws Exception {
        List<Order> orderList = getInitData();
        for (Order order : orderList) {
            if (order.getAmout() <= 100) {
                order.setScore(0);
                addScore(order);
            } else if (order.getAmout() > 100 && order.getAmout() <= 500) {
                order.setScore(100);
                addScore(order);
            } else if (order.getAmout() > 500 && order.getAmout() <= 1000) {
                order.setScore(500);
                addScore(order);
            } else {
                order.setScore(1000);
                addScore(order);
            }
        }
    }

    @Test
    public void droolsOrderTest() throws Exception {
        KieSession kieSession = kieContainer.newKieSession();
        List<Order> orderList = getInitData();
        for (Order order: orderList) {
```

```
            // 1-规则引擎处理逻辑
            kieSession.insert(order);
            kieSession.fireAllRules();
            // 2-执行完规则后，执行相关的逻辑
            addScore(order);
        }
        kieSession.dispose();
    }



    private static void addScore(Order o){
        System.out.println("用户" + o.getUser().getName() + "享受额外增加积分：" +
o.getScore());
    }

    private static List<Order> getInitData() throws Exception {
        List<Order> orderList = new ArrayList<>();
        DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
        {
            Order order = new Order();
            order.setAmout(80);
            order.setBookingDate(df.parse("2015-07-01"));
            User user = new User();
            user.setLevel(1);
            user.setName("Name1");
            order.setUser(user);
            order.setScore(111);
            orderList.add(order);
        }
        {
            Order order = new Order();
            order.setAmout(200);
            order.setBookingDate(df.parse("2015-07-02"));
            User user = new User();
            user.setLevel(2);
            user.setName("Name2");
            order.setUser(user);
            orderList.add(order);
        }
        {
            Order order = new Order();
            order.setAmout(800);
            order.setBookingDate(df.parse("2015-07-03"));
            User user = new User();
            user.setLevel(3);
            user.setName("Name3");
            order.setUser(user);
            orderList.add(order);
        }
        {
            Order order = new Order();
            order.setAmout(1500);
            order.setBookingDate(df.parse("2015-07-04"));
            User user = new User();
            user.setLevel(4);
            user.setName("Name4");
            order.setUser(user);
```

```
                orderList.add(order);
            }
            return orderList;
        }
    }
```

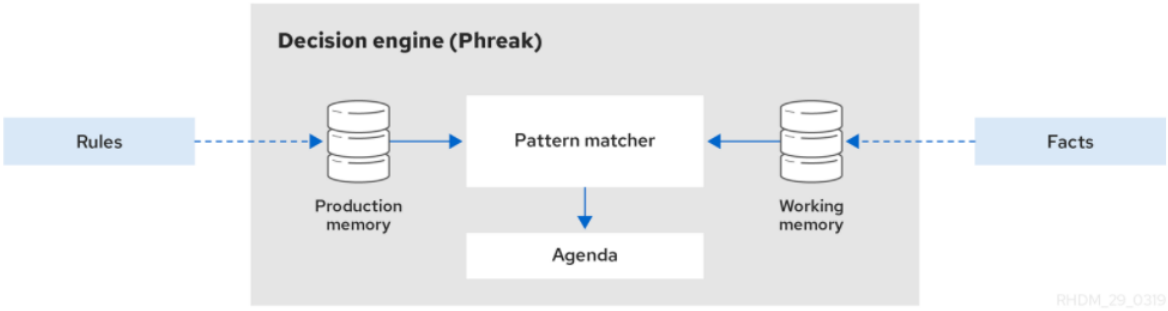
3.5 drools 开发小结

3.5.1 drools 组成

drools规则引擎由以下几部分构成：

- Working Memory（工作内存）
- Rules（规则库）
- Facts
- **Production memory**
- **Working memory:**
- **Agenda**

如下图所示：



3.5.2 相关概念说明

Working Memory：工作内存，drools规则引擎会从Working Memory中获取数据并和规则文件中定义的规则进行模式匹配，所以我们开发的应用程序只需要将我们的数据插入到Working Memory中即可，例如本案例中我们调用kieSession.insert(order)就是将order对象插入到了工作内存中。

Fact：事实，是指在drools 规则应用当中，将一个**普通的JavaBean插入到Working Memory后的对象**就是Fact对象，例如本案例中的Order对象就属于Fact对象。Fact对象是我们的应用和规则引擎进行数据交互的桥梁或通道。

Rules：规则库，我们在规则文件中定义的规则都会被加载到规则库中。

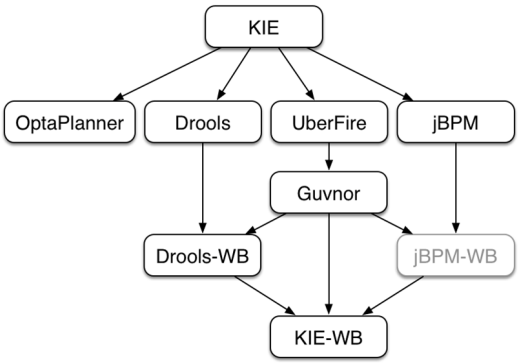
Pattern Matcher：匹配器，将Rule Base中的所有规则与Working Memory中的Fact对象进行模式匹配，匹配成功的规则将被激活并放入Agenda中。

Agenda：议程，用于存放通过匹配器进行模式匹配后被激活的规则。

3.5.3 规则引擎执行过程

3.5.4 KIE介绍

在上述分析积分兑换的过程中，简单地使用了 "kie "开头的一些类名，**Kie全称为Knowledge Is Everything**，即"知识就是一切"的缩写，是Jboss一系列项目的总称。官网描述：这个名字渗透在GitHub账户和Maven POMs中。随着范围的扩大和新项目的展开，KIE（Knowledge Is Everything的缩写）被选为新的组名。KIE的名字也被用于系统的共享方面；如统一的构建、部署和使用。



4.规则文件开发

4.1 规则文件构成

在使用Drools时非常重要的一个工作就是编写规则文件，通常规则文件的后缀为.drl。

drl是Drools Rule Language的缩写。在规则文件中编写具体的规则内容。

一套完整的规则文件内容构成如下：

关键字	描述
package	包名，只限于逻辑上的管理，同一个包名下的查询或者函数可以直接调用
import	用于导入类或者静态方法
global	全局变量
function	自定义函数
query	查询
rule end	规则体

Drools支持的规则文件，除了drl形式，还有Excel文件类型的。

4.2 规则体语法结构

规则体是规则文件内容中的重要组成部分，是进行业务规则判断、处理业务结果的部分。

规则体语法结构如下：

```
rule "ruleName"
    attributes
    when
        LHS
    then
        RHS
end
```

rule：关键字，表示规则开始，参数为规则的唯一名称。

attributes：规则属性，是rule与when之间的参数，为可选项。

when：关键字，后面跟规则的条件部分。

LHS(Left Hand Side)：是规则的条件部分的通用名称。它由零个或多个条件元素组成。**如果LHS为空，则它将被视为始终为true的条件元素。** （左手边）

then：关键字，后面跟规则的结果部分。

RHS(Right Hand Side)：是规则的后果或行动部分的通用名称。 （右手边）

end：关键字，表示一个规则结束。

4.3 注释

在drl形式的规则文件中使用注释和Java类中使用注释一致，分为单行注释和多行注释。

单行注释用"//"进行标记，多行注释以"/ *"开始，以* "/"结束。如下示例：

```
//规则rule1的注释，这是一个单行注释
rule "rule1"
    when
    then
        System.out.println("rule1触发");
end

/*
规则rule2的注释，
这是一个多行注释
*/
rule "rule2"
    when
    then
        System.out.println("rule2触发");
end
```

4.4 Pattern模式匹配

前面我们已经知道了Drools中的匹配器可以将Rule Base中的所有规则与Working Memory中的Fact对象进行模式匹配，那么我们就需要在规则体的LHS部分定义规则并进行模式匹配。LHS部分由一个或者多个条件组成，条件又称为pattern。

pattern的语法结构为：绑定变量名:Object(Field约束)

其中绑定变量名可以省略，通常绑定变量名的命名一般建议以$开始。如果定义了绑定变量名，就可以在规则体的RHS部分使用此绑定变量名来操作相应的Fact对象。Field约束部分是需要返回true或者false的0个或多个表达式。

例如我们的入门案例中：

```
rule "add100"
    no-loop true
    lock-on-active true
    salience 1
    when
        $order : Order(price > 100 && price <= 500)
    then
        $order.setScore(100);
        update($s);
end
```

通过上面的例子我们可以知道，匹配的条件为：

1、工作内存中必须存在Order这种类型的Fact对象-----类型约束

2、Fact对象的price属性值必须大于100------属性约束

3、Fact对象的price属性值必须小于等于500------属性约束

以上条件必须同时满足当前规则才有可能被激活。

绑定变量既可以用在对象上，也可以用在对象的属性上。例如上面的例子可以改为：

```
rule "add100"
    no-loop true
    lock-on-active true
    salience 1
    when
        $order : Order($price:price > 100 && amopriceut <= 500)
    then
        System.out.println("$price=" + $price);
        $s.setScore(100);
        update($s);
end
```

LHS部分还可以定义多个pattern，多个pattern之间可以使用and或者or进行连接，也可以不写，默认连接为and。

```
rule "add100"
    no-loop true
    lock-on-active true
    salience 1
    when
        $order : Order(price > 100 && price <= 500) and
        $user : User(level>3)
    then
        System.out.println($order.getUser());
        $order.setScore(100);
        update($order);
end
```

4.5 比较操作符

Drools提供的比较操作符，如下表：

符号	说明
>	大于
<	小于
>=	大于等于
<=	小于等于
==	等于
!=	不等于
contains	检查一个Fact对象的某个属性值是否包含一个指定的对象值
not contains	检查一个Fact对象的某个属性值是否不包含一个指定的对象值
memberOf	判断一个Fact对象的某个属性是否在一个或多个集合中
not memberOf	判断一个Fact对象的某个属性是否不在一个或多个集合中
matches	判断一个Fact对象的属性是否与提供的标准的Java正则表达式进行匹配
not matches	判断一个Fact对象的属性是否不与提供的标准的Java正则表达式进行匹配

前6个比较操作符和Java中的完全相同，下面我们重点学习后6个比较操作符。

4.5.1 语法

- **contains | not contains语法结构**

  Object(Field[Collection/Array] contains value)

  Object(Field[Collection/Array] not contains value)

- **memberOf | not memberOf语法结构**

  Object(field memberOf value[Collection/Array])

  Object(field not memberOf value[Collection/Array])

- **matches | not matches语法结构**

Object(field matches "正则表达式")

Object(field not matches "正则表达式")

contain是前面包含后面，memberOf是后面包含前面。

4.5.2 操作步骤

第一步：创建实体类，用于测试比较操作符

```
package com.mashibing.drools.entity;

import lombok.Data;
import lombok.experimental.Accessors;

import java.util.List;

/**
 * @author sunzhiqiang23
 * @date 2021-06-16 21:11
 */

@Data
@Accessors(chain = true)
public class ComparisonEntity {

    /**
     *名字集合
     */
    private String names;

    /**
     * 字符串集合
     */
    private List<String> list;

}
```

第二步：在/resources/rules下创建规则文件comparison.drl

```
package rules
import com.mashibing.drools.entity.ComparisonEntity

/*
 用于测试Drools提供的比较操作符
*/

//测试比较操作符contains
rule "rule_comparison_contains"
    when
        ComparisonEntity(names contains "张三")
        ComparisonEntity(list contains names)
    then
        System.out.println("规则rule_comparison_contains触发");
end
```

```
//测试比较操作符not contains
rule "rule_comparison_notContains"
    when
        ComparisonEntity(names not contains "张三")
        ComparisonEntity(list not contains names)
    then
        System.out.println("规则rule_comparison_notContains触发");
end

//测试比较操作符memberOf
rule "rule_comparison_memberOf"
    when
        ComparisonEntity(names memberOf list)
    then
        System.out.println("规则rule_comparison_memberOf触发");
end

//测试比较操作符not memberOf
rule "rule_comparison_notMemberOf"
    when
        ComparisonEntity(names not memberOf list)
    then
        System.out.println("规则rule_comparison_notMemberOf触发");
end

//测试比较操作符matches
rule "rule_comparison_matches"
    when
        ComparisonEntity(names matches "张.*")
    then
        System.out.println("规则rule_comparison_matches触发");
end

//测试比较操作符not matches
rule "rule_comparison_notMatches"
    when
        ComparisonEntity(names not matches "张.*")
    then
        System.out.println("规则rule_comparison_notMatches触发");
end
```

第三步：编写单元测试

```
package com.mashibing.drools.client;

import com.mashibing.drools.DroolsApplicationTests;
import com.mashibing.drools.entity.ComparisonEntity;
import org.junit.jupiter.api.Test;
import org.kie.api.KieBase;
import org.kie.api.runtime.KieSession;

import javax.annotation.Resource;
import java.util.ArrayList;
import java.util.List;

/**
```

```
 * @author sunzhiqiang23
 * @date 2021-06-17 23:46
 */
public class ComparisonTest extends DroolsApplicationTests {

    @Resource
    public KieBase kieBase;

    @Test
    public void testComparison(){
        KieSession kieSession = kieBase.newKieSession();
        ComparisonEntity comparisonEntity = new ComparisonEntity();
        comparisonEntity.setNames("张三");
        List<String> list = new ArrayList<>();
        list.add("张三");
        list.add("李四");
        comparisonEntity.setList(list);

        kieSession.insert(comparisonEntity);

        kieSession.fireAllRules();
        kieSession.dispose();
    }
}
```

4.6 执行指定规则

通过前面的案例可以看到，我们在调用规则代码时，满足条件的规则都会被执行。那么如果我们只想执行其中的某个规则如何实现呢?

Drools给我们提供的方式是通过规则过滤器来实现执行指定规则。对于规则文件不用做任何修改，只需要修改Java代码即可，如下：

```
//通过规则过滤器实现只执行指定规则
kieSession.fireAllRules(new kieSession.fireAllRules(new
RuleNameEqualsAgendaFilter("rule_filter_1"));
```

4.7 关键字

Drools的关键字分为：硬关键字(Hard keywords)和软关键字(Soft keywords)。

硬关键字是我们在规则文件中定义包名或者规则名时明确不能使用的，否则程序会报错。软关键字虽然可以使用，但是不建议使用。

硬关键字包括：true false null

软关键字包括：lock-on-active date-effective date-expires no-loop auto-focus activation-group agenda-group ruleflow-group entry-point duration package import dialect salience enabled attributes rule extend when then template query declare function global eval not in or and exists forall accumulate collect from action reverse result end over init

```
比如：
rule true  //不可以
rule "true" 可以
```

5. 规则属性 attributes

前面我们已经知道了规则体的构成如下：

```
rule "ruleName"
    attributes
    when
        LHS
    then
        RHS
end
```

本章节就是针对规则体的**attributes**属性部分进行讲解。Drools中提供的属性如下表(部分属性)：

属性名	说明
salience	指定规则执行优先级
dialect	指定规则使用的语言类型，取值为java和mvel
enabled	指定规则是否启用
date-effective	指定规则生效时间
date-expires	指定规则失效时间
activation-group	激活分组，具有相同分组名称的规则只能有一个规则触发
agenda-group	议程分组，只有获取焦点的组中的规则才有可能触发
timer	定时器，指定规则触发的时间
auto-focus	自动获取焦点，一般结合agenda-group一起使用
no-loop	防止死循环，防止自己更新规则再次触发
lock-on-active	no-loop增强版本。可防止别人更新规则再次出发

5.1 enabled属性

enabled属性对应的取值为true和false，默认值为true。

用于指定当前规则是否启用，如果设置的值为false则当前规则无论是否匹配成功都不会触发

```
package rules
import com.mashibing.drools.entity.AttributesEnabledEntity

/*
 用于测试Drools 属性:enabled
*/

//测试enabled
```

```
rule "rule_attributes_enabled"
    enabled false
    when
        AttributesEnabledEntity(num > 10)
    then
        System.out.println("规则rule_attributes_enabled触发");
end
```

5.2 dialect属性

dialect属性用于指定当前规则使用的语言类型，取值为java和mvel，默认值为java。

注：mvel是一种基于java语法的表达式语言。

虽然mvel吸收了大量的java语法，但作为一个表达式语言，还是有着很多重要的不同之处，以达到更高的效率，比如：mvel像正则表达式一样，有直接支持集合、数组和字符串匹配的操作符。

除了表达式语言外，mvel还提供了用来配置和构造字符串的模板语言。

mvel2.x表达式包含以下部分的内容：属性表达式，布尔表达式，方法调用，变量赋值，函数定义

5.3 salience属性

salience属性用于指定规则的执行优先级，**取值类型为Integer。数值越大越优先执行。**每个规则都有一个默认的执行顺序，如果不设置salience属性，规则体的执行顺序为由上到下。

drl文件内容如下：

```
package rules
import com.mashibing.drools.entity.AttributesSalienceEntity

/*
 用于测试Drools 属性:salience
*/

rule "rule_attributes_salience_1"
    when
        AttributesSalienceEntity(flag == true)
    then
        System.out.println("规则 rule_attributes_salience_1 触发");
end

rule "rule_attributes_salience_2"
    when
        AttributesSalienceEntity(flag == true)
    then
        System.out.println("规则 rule_attributes_salience_2 触发");
end

rule "rule_attributes_salience_3"
    when
        AttributesSalienceEntity(flag == true)
    then
        System.out.println("规则 rule_attributes_salience_3 触发");
end
```

通过控制台可以看到，由于以上三个规则没有设置salience属性，所以执行的顺序是按照规则文件中规则的顺序由上到下执行的。接下来我们修改一下文件内容：

```
package rules
import com.mashibing.drools.entity.AttributesSalienceEntity

/*
 用于测试Drools 属性:salience
*/

rule "rule_attributes_salience_1"
    salience 10
    when
        AttributesSalienceEntity(flag == true)
    then
        System.out.println("规则 rule_attributes_salience_1 触发");
end

rule "rule_attributes_salience_2"
    salience 20
    when
        AttributesSalienceEntity(flag == true)
    then
        System.out.println("规则 rule_attributes_salience_2 触发");
end

rule "rule_attributes_salience_3"
    salience 1
    when
        AttributesSalienceEntity(flag == true)
    then
        System.out.println("规则 rule_attributes_salience_3 触发");
end
```

通过控制台可以看到，规则文件执行的顺序是按照我们设置的salience值由大到小顺序执行的。

建议在编写规则时使用salience属性明确指定执行优先级。

5.4 no-loop属性

no-loop属性用于防止死循环，当规则通过update之类的函数修改了Fact对象时，可能使当前规则再次被激活从而导致死循环。取值类型为Boolean，默认值为false。测试步骤如下：

第一步：编写规则文件

```
package rules
import com.mashibing.drools.entity.AttributesNoLoopEntity

/*
 用于测试Drools 属性:no-loop
*/

rule "rule_attributes_noloop"
    //no-loop true
    when
        $attributesNoLoopEntity:AttributesNoLoopEntity(num > 1)
```

```
    then
        update($attributesNoLoopEntity)
        System.out.println("规则 rule_attributes_noloop 触发");
end
```

第二步：编写单元测试

```
    @Test
    public void test(){
        KieSession kieSession = kieBase.newKieSession();
        AttributesNoLoopEntity attributesNoLoopEntity = new AttributesNoLoopEntity();
        attributesNoLoopEntity.setNum(20);

        kieSession.insert(attributesNoLoopEntity);

        kieSession.fireAllRules();
        kieSession.dispose();
    }
```

通过控制台可以看到，由于我们没有设置no-loop属性的值，所以发生了死循环。接下来设置no-loop的值为true再次测试则不会发生死循环。

5.5 lock-on-active属性

lock-on-active这个属性，可以限制当前规则只会被执行一次，包括当前规则的重复执行不是本身触发的。取值类型为Boolean，默认值为false。测试步骤如下：

第一步：编写规则文件

```
package rules
import com.mashibing.drools.entity.AttributesLockOnActiveEntity

/*
 用于测试Drools 属性:lock-on-active
*/

rule "rule_attributes_lock_on_active_1"
    no-loop true
    when
        $attributesLockOnActiveEntity:AttributesLockOnActiveEntity(num > 1)
    then
        update($attributesLockOnActiveEntity)
        System.out.println("规则 rule_attributes_lock_on_active_1 触发");
end

rule "rule_attributes_lock_on_active_2"
    no-loop true
    lock-on-active true
    when
        $attributesLockOnActiveEntity:AttributesLockOnActiveEntity(num > 1)
    then
        update($attributesLockOnActiveEntity)
        System.out.println("规则 rule_attributes_lock_on_active_2 触发");
end
```

第二步：编写单元测试

```
    @Test
    public void test(){
        KieSession kieSession = kieBase.newKieSession();
        AttributesLockOnActiveEntity attributesLockOnActiveEntity = new
AttributesLockOnActiveEntity();

        attributesLockOnActiveEntity.setNum(20);

        kieSession.insert(attributesLockOnActiveEntity);

        kieSession.fireAllRules();
        kieSession.dispose();
    }
```

no-loop的作用是限制因为modify等更新操作导致规则重复执行，但是有一个限定条件，是当前规则中进行更新导致当前规则重复执行。而不是防止其他规则更新相同的fact对象，导致当前规则更新,lock-on-active可以看作是no-loop的加强版，不仅能限制自己的更新，还能限制别人的更新造成的死循环。

5.6 activation-group属性。

activation-group属性是指**激活分组**，取值为String类型。具有相同分组名称的规则只能有一个规则被触发。

第一步：编写规则文件

```
package rules
import com.mashibing.drools.entity.AttributesActivationGroupEntity

/*
 用于测试Drools 属性: activation-group
*/

rule "rule_attributes_activation_group_1"
    activation-group "customGroup"
    when
        $attributesActivationGroupEntity:AttributesActivationGroupEntity(num > 1)
    then
        System.out.println("规则 rule_attributes_activation_group_1 触发");
end

rule "rule_attributes_activation_group_2"
    activation-group "customGroup"
    when
        $attributesActivationGroupEntity:AttributesActivationGroupEntity(num > 1)
    then
        System.out.println("规则 rule_attributes_activation_group_2 触发");
end
```

第二步：编写单元测试

```
    @Test
    public void test(){
        KieSession kieSession = kieBase.newKieSession();
        AttributesActivationGroupEntity attributesActivationGroupEntity = new
AttributesActivationGroupEntity();
        attributesActivationGroupEntity.setNum(20);

        kieSession.insert(attributesActivationGroupEntity);

        kieSession.fireAllRules();
        kieSession.dispose();
    }
```

通过控制台可以发现，上面的两个规则因为属于同一个分组，所以只有一个触发了。同一个分组中的多个规则如果都能够匹配成功，具体哪一个最终能够被触发可以通过salience属性确定。

5.7 agenda-group属性

agenda-group属性为**议程分组**，属于另一种可控的规则执行方式。用户可以通过设置agenda-group来控制规则的执行，只有获取焦点的组中的规则才会被触发。

第一步：编写规则文件

```
package rules
import com.mashibing.drools.entity.AttributesAgendaGroupEntity

/*
 用于测试Drools 属性: agenda-group
*/

rule "rule_attributes_agenda_group_1"
    agenda-group "customAgendaGroup1"
    when
        $attributesAgendaGroupEntity:AttributesAgendaGroupEntity(num > 1)
    then
        System.out.println("规则 rule_attributes_agenda_group_1 触发");
end

rule "rule_attributes_agenda_group_2"
    agenda-group "customAgendaGroup1"
    when
        $attributesAgendaGroupEntity:AttributesAgendaGroupEntity(num > 1)
    then
        System.out.println("规则 rule_attributes_agenda_group_2 触发");
end


rule "rule_attributes_activation_group_3"
    agenda-group "customAgendaGroup2"
    when
        $attributesAgendaGroupEntity:AttributesAgendaGroupEntity(num > 1)
    then
```

```
        System.out.println("规则 rule_attributes_activation_group_3 触发");
end

rule "rule_attributes_agenda_group_4"
    agenda-group "customAgendaGroup2"
    when
        $attributesAgendaGroupEntity:AttributesAgendaGroupEntity(num > 1)
    then
        System.out.println("规则 rule_attributes_agenda_group_4 触发");
end
```

第二步：编写单元测试

```
    @Test
    public void test(){
        KieSession kieSession = kieBase.newKieSession();
        AttributesAgendaGroupEntity attributesAgendaGroupEntity = new
AttributesAgendaGroupEntity();
        attributesAgendaGroupEntity.setNum(20);

        kieSession.insert(attributesAgendaGroupEntity);
        kieSession.getAgenda().getAgendaGroup("customAgendaGroup2").setFocus();

        kieSession.fireAllRules();
        kieSession.dispose();
    }
```

通过控制台可以看到，只有获取焦点的分组中的规则才会触发。与activation-group不同的是，activation-group定义的分组中只能够有一个规则可以被触发，而agenda-group分组中的多个规则都可以被触发。

5.8 auto-focus属性

auto-focus属性为**自动获取焦点**，取值类型为Boolean，默认值为false。一般结合agenda-group属性使用，当一个议程分组未获取焦点时，可以设置auto-focus属性来控制。

第一步：编写规则文件

```
package rules
import com.mashibing.drools.entity.AttributesAutoFocusEntity

/*
 用于测试Drools 属性: auto-focus
*/

rule "rule_attributes_auto_focus_1"
    agenda-group "customAgendaGroup1"
    when
        $attributesAutoFocusEntity:AttributesAutoFocusEntity(num > 1)
    then
        System.out.println("规则 rule_attributes_auto_focus_1 触发");
end

rule "rule_attributes_auto_focus_2"
```

```
    agenda-group "customAgendaGroup1"
    when
        $attributesAutoFocusEntity:AttributesAutoFocusEntity(num > 1)
    then
        System.out.println("规则 rule_attributes_auto_focus_2 触发");
end

rule "rule_attributes_auto_focus_3"
    agenda-group "customAgendaGroup2"
//    auto-focus true
    when
        $attributesAutoFocusEntity:AttributesAutoFocusEntity(num > 1)
    then
        System.out.println("规则 rule_attributes_auto_focus_3 触发");
end

rule "rule_attributes_auto_focus_4"
    agenda-group "customAgendaGroup2"
    when
        $attributesAutoFocusEntity:AttributesAutoFocusEntity(num > 1)
    then
        System.out.println("规则 rule_attributes_auto_focus_4 触发");
end
```

第二步：编写单元测试

```
    @Test
    public void test(){
        KieSession kieSession = kieBase.newKieSession();
        AttributesAutoFocusEntity attributesAutoFocusEntity = new
AttributesAutoFocusEntity();
        attributesAutoFocusEntity.setNum(20);

        kieSession.insert(attributesAutoFocusEntity);

        kieSession.fireAllRules();
        kieSession.dispose();
    }
```

通过控制台可以看到，设置auto-focus属性为true的规则都触发了。

注意：同一个组，只要有个设置auto-focus true 其他的设置不设置都无所谓啦。都会起作用的。

5.9 timer属性

timer属性可以通过定时器的方式指定规则执行的时间，使用方式有两种：

方式一：timer (int: ?)

此种方式遵循java.util.Timer对象的使用方式，第一个参数表示几秒后执行，第二个参数表示每隔几秒执行一次，第二个参数为可选。

方式二：timer(cron:)

此种方式使用标准的unix cron表达式的使用方式来定义规则执行的时间。

第一步：编写规则文件

```
package rules
import com.mashibing.drools.entity.AttributesTimerEntity

/*
 用于测试Drools 属性: timer
*/

rule "rule_attributes_timer_1"
    timer(5s 2s)
    when
        $attributesTimerEntity:AttributesTimerEntity(num > 1)
    then
        System.out.println("规则 rule_attributes_timer_1 触发");
end

rule "rule_attributes_timer_2"
    timer(cron:0/1 * * * * ?)
    when
        $attributesTimerEntity:AttributesTimerEntity(num > 1)
    then
        System.out.println("规则 rule_attributes_timer_2 触发");
end
```

第二步：编写单元测试

```
@Test
    public void test() throws InterruptedException {

        KieSession kieSession = kieBase.newKieSession();
        AttributesTimerEntity attributesTimerEntity = new AttributesTimerEntity();
        attributesTimerEntity.setNum(20);

        kieSession.insert(attributesTimerEntity);
        kieSession.fireUntilHalt();

        Thread.sleep(10000);
        kieSession.halt();

        kieSession.dispose();
    }
```

注意：如果规则中有用到了timer属性，匹配规则需要调用kieSession.fireUntilHalt();这里涉及一个规则引擎的执行模式和线程问题，关于具体细节，我们后续讨论。

5.10 date-effective属性

date-effective属性**用于指定规则的生效时间**，即只有当前系统时间大于等于设置的时间或者日期规则才有可能触发。默认日期格式为：dd-MMM-yyyy。用户也可以自定义日期格式。

第一步：编写规则文件

```
package rules
```

```
import com.mashibing.drools.entity.AttributesDateEffectiveEntity

/*
 用于测试Drools 属性: date-effective
*/

rule "rule_attributes_date_effective"
//    date-effective "20-七月-2021"
    date-effective "2021-02-20"
    when
        $attributesDateEffectiveEntity:AttributesDateEffectiveEntity(num > 1)
    then
        System.out.println("规则 rule_attributes_date_effective 触发");
end
```

第二步：编写单元测试

```
    @Test
    public void test(){
        KieSession kieSession = kieBase.newKieSession();
        AttributesDateEffectiveEntity attributesDateEffectiveEntity = new
AttributesDateEffectiveEntity();
        attributesDateEffectiveEntity.setNum(20);

        kieSession.insert(attributesDateEffectiveEntity);

        kieSession.fireAllRules();
        kieSession.dispose();
    }
```

注意：需要在VM参数上加上日期格式:-Ddrools.dateformat=yyyy-MM-dd，在生产环境所在规则引擎的JVM设置中，也需要设置此参数，以保证开发和生产的一致性。

5.11 date-expires属性

date-expires属性用于指定规则的**失效时间**，即只有当前系统时间小于设置的时间或者日期规则才有可能触发。默认日期格式为：dd-MMM-yyyy。用户也可以自定义日期格式。

第一步：编写规则文件/resource/rules/dateexpires.drl

```
package rules
import com.mashibing.drools.entity.AttributesDateExpiresEntity

/*
 用于测试Drools 属性: date-expires
*/

rule "rule_attributes_date_expires"
    date-expires "2021-06-20"
    when
        $attributesDateExpiresEntity:AttributesDateExpiresEntity(num > 1)
    then
```

```
        System.out.println("规则 rule_attributes_date_expires 触发");
end
```

第二步：编写单元测试

```
@Test
public void test(){
    KieSession kieSession = kieBase.newKieSession();
    AttributesDateExpiresEntity attributesDateExpiresEntity = new
AttributesDateExpiresEntity();
    attributesDateExpiresEntity.setNum(20);

    kieSession.insert(attributesDateExpiresEntity);

    kieSession.fireAllRules();
    kieSession.dispose();
}
```

注意：需要在VM参数上加上日期格式:-Ddrools.dateformat=yyyy-MM-dd，在生产环境所在规则引擎的JVM设置中，也需要设置此参数，以保证开发和生产的一致性。

6. Drools高级语法

前面章节我们已经知道了一套完整的规则文件内容构成如下：

关键字	描述
package	包名，只限于逻辑上的管理，同一个包名下的查询或者函数可以直接调用
import	用于导入类或者静态方法
global	全局变量
function	自定义函数
query	查询
rule end	规则体

本章节我们就来学习其中的几个关键字。

6.1 global全局变量

global关键字用于在规则文件中**定义全局变量**，它可以让应用程序的对象在规则文件中能够被访问。可以用来为规则文件提供数据或服务。

语法结构为：**global 对象类型 对象名称**

在使用global定义的全局变量时有两点需要注意：

1、如果对象类型为**包装类型**时，在一个规则中改变了global的值，那么**只针对当前规则有效**，对其他规则中的global不会有影响。可以理解为它是当前规则代码中的global副本，规则内部修改不会影响全局的使用。

2、如果对象类型为**集合类型或JavaBean**时，在一个规则中改变了global的值，对java代码和所有规则都有效。

下面我们通过代码进行验证：

第一步：编写规则文件

```
package rules
import com.mashibing.drools.entity.GlobalEntity

/*
 用于测试Drools 全局变量 : global
*/

global java.lang.Integer globalCount
global java.util.List globalList

rule "rule_global_1"
    when
        $globalEntity:GlobalEntity(num > 1)
    then
        System.out.println("规则 rule_global_1 开始...");
        globalCount++ ;
        globalList.add("张三");
        globalList.add("李四");

        System.out.println(globalCount);
        System.out.println(globalList);
        System.out.println("规则 rule_global_1 结束...");
end

rule "rule_global_2"
    when
        $globalEntity:GlobalEntity(num > 1)
    then
        System.out.println("规则 rule_global_2 开始...");
        System.out.println(globalCount);
        System.out.println(globalList);
        System.out.println("规则 rule_global_2 结束...");
end
```

第二步：编写单元测试

```
    @Test
    public void test(){
        KieSession kieSession = kieBase.newKieSession();
        GlobalEntity globalEntity = new GlobalEntity();
        globalEntity.setNum(20);

        ArrayList<Object> globalList = new ArrayList<>();

        Integer globalCount = 10;
```

```
        kieSession.setGlobal("globalCount", 10);
        kieSession.setGlobal("globalList", globalList);

        kieSession.insert(globalEntity);

        kieSession.fireAllRules();
        kieSession.dispose();
        System.out.println("globalCount=" + globalCount);
        System.out.println("globalList=" + globalList);
    }
```

注意：

1-后面的代码中定义了全局变量以后，前面的test都需要加，不然会出错。

2-属性当中的 关于时间的属性，如果涉及格式问题，也不要忘记，jvm启动参数添加相关配置

6.2 query查询

query查询提供了一种**查询working memory中符合约束条件的Fact对象**的简单方法。它仅包含规则文件中的LHS部分，不用指定“when”和“then”部分并且以end结束。具体语法结构如下：

```
query 查询的名称(可选参数)
    LHS
end
```

具体操作步骤：

第一步：编写规则文件

```
package rules
import com.mashibing.drools.entity.QueryEntity

/*
 用于测试Drools 方法: query
*/

//无参查询
query "query_1"
    $queryEntity:QueryEntity(age>20)
end

//有参查询
query "query_2"(Integer qAge,String qName)
    $queryEntity:QueryEntity(age > qAge && name == qName)
end
```

第二步：编写单元测试

```
    @Test
    public void test(){
        KieSession kieSession = kieBase.newKieSession();
```

```
        QueryEntity queryEntity1= new QueryEntity();
        QueryEntity queryEntity2= new QueryEntity();
        QueryEntity queryEntity3= new QueryEntity();

        queryEntity1.setName("张三").setAge(10);
        queryEntity2.setName("李四").setAge(20);
        queryEntity3.setName("王五").setAge(30);


        kieSession.insert(queryEntity1);
        kieSession.insert(queryEntity2);
        kieSession.insert(queryEntity3);

        QueryResults results1 = kieSession.getQueryResults("query_1");
        QueryResults results2 = kieSession.getQueryResults("query_2", 1, "张三");


        for (QueryResultsRow queryResultsRow : results1) {
            QueryEntity queryEntity = (QueryEntity)
(queryResultsRow.get("$queryEntity"));
            System.out.println(queryEntity);
        }

        for (QueryResultsRow queryResultsRow : results2) {
            QueryEntity queryEntity = (QueryEntity)
(queryResultsRow.get("$queryEntity"));
            System.out.println(queryEntity);
        }


        kieSession.fireAllRules();
        kieSession.dispose();
    }
```

6.3 function函数

function关键字用于在规则文件中定义函数，就相当于java类中的方法一样。可以在规则体中调用定义的函数。使用函数的好处是可以将业务逻辑集中放置在一个地方，根据需要可以对函数进行修改。

函数定义的语法结构如下：

```
function 返回值类型 函数名(可选参数){     //逻辑代码}
```

具体操作步骤：

第一步：编写规则文件/resources/rules/function.drl

```
package rules
import com.mashibing.drools.entity.FunctionEntity

/*
 用于测试Drools 方法: function
*/
```

```
//定义一个 假发 方法
function Integer add(Integer num){
    return num+10;
}

rule  "function"
    when
        $functionEntity:FunctionEntity(num>20)
    then
        Integer result = add($functionEntity.getNum());
        System.out.println(result);
end
```

第二步：编写单元测试

```
    @Test
    public void test(){
        KieSession kieSession = kieBase.newKieSession();

        FunctionEntity functionEntity = new FunctionEntity();
        functionEntity.setNum(30);

        kieSession.insert(functionEntity);

        kieSession.fireAllRules();
        kieSession.dispose();
    }
```

6.4 条件-LHS加强

前面我们已经知道了在规则体中的LHS部分是**介于when和then之间的部分**，主要用于模式匹配，只有匹配结果为true时，才会触发RHS部分的执行。本章节我们会针对LHS部分学习几个新的用法。

6.4.1 复合值限制in/not in

复合值限制是指超过一种匹配值的限制条件，类似于SQL语句中的in关键字。Drools规则体中的LHS部分可以使用in或者not in进行复合值的匹配。具体语法结构如下：

Object(field in (比较值1,比较值2...))

举例：

```
package rules
import com.mashibing.drools.entity.LhsInEntity

/*
 用于测试Drools LHS: in not in
*/


rule "lhs_in"
    when
        $lhsInEntity:LhsInEntity(name in ("张三","李四","王五"))
    then
```

```
        System.out.println("规则 lhs_in 触发");
end

rule "lhs_not_in"
    when
        $lhsInEntity:LhsInEntity(name not in ("张三","李四","王五"))
    then
        System.out.println("规则 lhs_not_in 触发");
end
```

6.4.2 条件元素eval

eval用于规则体的LHS部分，并返回一个Boolean类型的值。语法结构如下：

eval(表达式)

举例：

```
package rules
import com.mashibing.drools.entity.LhsEvalEntity

/*
 用于测试Drools LHS: in not in
*/


rule "lhs_eval"
    when
        $lhsInEntity:LhsEvalEntity(age > 10) and eval(2>1)
    then
        System.out.println("规则 lhs_eval 触发");
end
```

6.4.3 条件元素not

not用于判断Working Memory中是否存在某个Fact对象，如果不存在则返回true，如果存在则返回false。语法结构如下：

not Object(可选属性约束)

举例：

```
package rules
import com.mashibing.drools.entity.LhsNotEntity

/*
 用于测试Drools LHS: not
*/


rule "lhs_not"
    when
       not $lhsNotEntity:LhsNotEntity(age > 10)
    then
```

```
        System.out.println("规则 lhs_not 触发");
end
```

6.4.4 条件元素exists

exists的作用与not相反，用于判断Working Memory中是否存在某个Fact对象，如果存在则返回true，不存在则返回false。语法结构如下：

exists Object(可选属性约束)

举例：

```
package rules
import com.mashibing.drools.entity.LhsEvalEntity

/*
 用于测试Drools LHS: exists
*/


rule "lhs_exists"
    when
       exists $lhsInEntity:LhsEvalEntity(age > 10)
    then
        System.out.println("规则 lhs_eval 触发");
end
```

Java代码：

```
package com.mashibing.drools.client;

import com.mashibing.drools.DroolsApplicationTests;
import com.mashibing.drools.entity.LhsExistsEntity;
import com.mashibing.drools.entity.LhsNotEntity;
import org.junit.jupiter.api.Test;
import org.kie.api.KieBase;
import org.kie.api.runtime.KieSession;

import javax.annotation.Resource;

/**
 * @author sunzhiqiang23
 * @date 2021-06-17 23:46
 */
public class LhsNotTest extends DroolsApplicationTests {

    @Resource
    public KieBase kieBase;

    @Test
    public void test(){
        KieSession kieSession = kieBase.newKieSession();
        LhsNotEntity lhsNotEntity = new LhsNotEntity();
        lhsNotEntity.setAge(1);
```

```
        kieSession.insert(lhsNotEntity);

        kieSession.fireAllRules();
        kieSession.dispose();
    }

}
```

可能有人会有疑问，我们前面在LHS部分进行条件编写时并没有使用exists也可以达到判断Working Memory中是否存在某个符合条件的Fact元素的目的，那么我们使用exists还有什么意义？

两者的区别：当向Working Memory中加入多个满足条件的Fact对象时，使用了exists的规则执行一次，不使用exists的规则会执行多次。

例如：

规则文件(只有规则体)：

```
package rules
import com.mashibing.drools.entity.LhsExistsEntity

/*
 用于测试Drools LHS: exists
*/


rule "lhs_exists_1"
    when
       exists $lhsExistsEntity:LhsExistsEntity(age > 10)
    then
        System.out.println("规则 lhs_exists_1 触发");
end


rule "lhs_exists_2"
    when
       $lhsExistsEntity:LhsExistsEntity(age > 10)
    then
       System.out.println("规则 lhs_exists_2 触发");
end
```

Java代码：

```
    @Test
    public void test(){
        KieSession kieSession = kieBase.newKieSession();
        LhsExistsEntity lhsExistsEntity = new LhsExistsEntity();
        lhsExistsEntity.setAge(30);

        LhsExistsEntity lhsExistsEntity2 = new LhsExistsEntity();
        lhsExistsEntity2.setAge(30);
```

```
        kieSession.insert(lhsExistsEntity);
        kieSession.insert(lhsExistsEntity2);

        kieSession.fireAllRules();
        kieSession.dispose();
    }
```

上面第一个规则只会执行一次，因为Working Memory中存在两个满足条件的Fact对象，第二个规则会执行两次。

6.4.5 规则继承

规则之间可以使用extends关键字进行规则条件部分的继承，类似于java类之间的继承。

例如：

```
    @Test
    public void test(){
        KieSession kieSession = kieBase.newKieSession();
        LhsExistsEntity lhsExistsEntity = new LhsExistsEntity();
        lhsExistsEntity.setAge(30);

        LhsExistsEntity lhsExistsEntity2 = new LhsExistsEntity();
        lhsExistsEntity2.setAge(30);

        kieSession.insert(lhsExistsEntity);
        kieSession.insert(lhsExistsEntity2);

        kieSession.fireAllRules();
        kieSession.dispose();
    }
```

6.5 结果-RHS

规则文件的 `RHS` 部分的主要作用是通过**插入，删除或修改工作内存中的Fact数据**，来达到控制规则引擎执行的目的。Drools提供了一些方法可以用来操作工作内存中的数据，**操作完成后规则引擎会重新进行相关规则的匹配**，原来没有匹配成功的规则在我们修改数据完成后有可能就会匹配成功了。

6.5.1 insert方法

insert方法的作用是向工作内存中插入数据，并让相关的规则重新匹配。

第一步：编写规则文件

```
package rules
import com.mashibing.drools.entity.RhsInsertEntity

/*
 用于测试Drools RHS: insert
*/


rule "rhs_insert_1"
    when
```

```
        $rhsInsertEntity:RhsInsertEntity(age <= 10)
    then
        RhsInsertEntity rhsInsertEntity = new RhsInsertEntity();
        rhsInsertEntity.setAge(15);
        insert(rhsInsertEntity);
        System.out.println("规则 rhs_insert_1 触发");
end

rule "rhs_insert_2"
    when
        $rhsInsertEntity:RhsInsertEntity(age <=20 && age>10)
    then
        RhsInsertEntity rhsInsertEntity = new RhsInsertEntity();
        rhsInsertEntity.setAge(25);
        insert(rhsInsertEntity);
        System.out.println("规则 rhs_insert_2 触发");
end

rule "rhs_insert_3"
    when
        $rhsInsertEntity:RhsInsertEntity(age > 20 )
    then
        System.out.println("规则 rhs_insert_3 触发");
end
```

第二步：编写单元测试

```
    public void test(){
        KieSession kieSession = kieBase.newKieSession();
        RhsInsertEntity rhsInsertEntity = new RhsInsertEntity();
        rhsInsertEntity.setAge(5);

        kieSession.insert(rhsInsertEntity);

        kieSession.fireAllRules();
        kieSession.dispose();
    }
```

通过控制台输出可以发现，3个规则都触发了，这是因为首先进行规则匹配时只有第一个规则可以匹配成功，但是在第一个规则中向工作内存中插入了一个数据导致重新进行规则匹配，此时第二个规则可以匹配成功。在第二个规则中同样向工作内存中插入了一个数据导致重新进行规则匹配，那么第三个规则就出发了。

6.5.2 update方法

update方法的作用是更新工作内存中的数据，并让相关的规则重新匹配。 （要避免死循环）

第一步：编写规则文件

```
package rules
import com.mashibing.drools.entity.RhsUpdateEntity

/*
 用于测试Drools RHS: update
```

```
*/


rule "rhs_update_1"
    when
        $rhsUpdateEntity:RhsUpdateEntity(age <= 10)
    then
        $rhsUpdateEntity.setAge(15);
        update($rhsUpdateEntity);
        System.out.println("规则 rhs_update_1 触发");
end

rule "rhs_update_2"
    when
        $rhsUpdateEntity:RhsUpdateEntity(age <=20 && age>10)
    then
        $rhsUpdateEntity.setAge(25);
        update($rhsUpdateEntity);
        System.out.println("规则 rhs_update_2 触发");
end

rule "rhs_update_3"
    when
        $rhsUpdateEntity:RhsUpdateEntity(age > 20 )
    then
        System.out.println("规则 rhs_update_3 触发");
end
```

第二步：编写单元测试

```
    @Test
    public void test(){
        KieSession kieSession = kieBase.newKieSession();
        RhsUpdateEntity rhsUpdateEntity = new RhsUpdateEntity();
        rhsUpdateEntity.setAge(5);

        kieSession.insert(rhsUpdateEntity);

        kieSession.fireAllRules();
        kieSession.dispose();
    }
```

通过控制台的输出可以看到规则文件中定义的三个规则都触发了。

在更新数据时需要注意防止发生死循环。

6.5.3 modify方法

modify方法的作用跟update一样，是更新工作内存中的数据，并让相关的规则重新匹配。只不过语法略有区别 （要避免死循环）

第一步：编写规则文件

```
package rules
import com.mashibing.drools.entity.RhsModifyEntity

/*
 用于测试Drools RHS: modify
*/


rule "rhs_modify_1"
    when
        $rhsModifyEntity:RhsModifyEntity(age <= 10)
    then
        modify($rhsModifyEntity){
            setAge(15)
        }
        System.out.println("规则 rhs_modify_1 触发");
end

rule "rhs_modify_2"
    when
        $rhsModifyEntity:RhsModifyEntity(age <=20 && age>10)
    then
        modify($rhsModifyEntity){
            setAge(25)
        }
        System.out.println("规则 rhs_modify_2 触发");
end

rule "rhs_modify_3"
    when
        $rhsModifyEntity:RhsModifyEntity(age > 20 )
    then
        System.out.println("规则 rhs_modify_3 触发");
end
```

第二步：编写单元测试

```
    @Test
    public void test(){
        KieSession kieSession = kieBase.newKieSession();
        RhsModifyEntity rhsModifyEntity = new RhsModifyEntity();
        rhsModifyEntity.setAge(5);

        kieSession.insert(rhsModifyEntity);

        kieSession.fireAllRules();
        kieSession.dispose();
    }
```

通过控制台的输出可以看到规则文件中定义的三个规则都触发了。

在更新数据时需要注意防止发生死循环。

6.5.4 retract/delete方法

retract方法的作用是删除工作内存中的数据，并让相关的规则重新匹配。

第一步：编写规则文件

```
package rules
import com.mashibing.drools.entity.RhsRetractEntity

/*
 用于测试Drools RHS: retract
*/


rule "rhs_retract_1"
    when
        $rhsRetractEntity:RhsRetractEntity(age <= 10)
    then
//        retract($rhsRetractEntity);
        System.out.println("规则 rhs_retract_1 触发");
end

rule "rhs_retract_2"
    when
        $rhsRetractEntity:RhsRetractEntity(age <= 10)
    then
        System.out.println("规则 rhs_retract_2 触发");
end
```

第二步：编写单元测试

```
    public void test(){
        KieSession kieSession = kieBase.newKieSession();
        RhsRetractEntity rhsRetractEntity = new RhsRetractEntity();
        rhsRetractEntity.setAge(5);

        kieSession.insert(rhsRetractEntity);

        kieSession.fireAllRules();
        kieSession.dispose();
    }
```

通过控制台输出可以发现，只有第一个规则触发了，因为在第一个规则中将工作内存中的数据删除了导致第二个规则并没有匹配成功。

6.6 RHS加强

RHS部分是规则体的重要组成部分，当LHS部分的条件匹配成功后，对应的RHS部分就会触发执行。一般在RHS部分中需要进行业务处理。

在RHS部分Drools为我们提供了一个内置对象，名称就是drools。本小节我们来介绍几个drools对象提供的方法。

6.5.1 halt

halt方法的作用是**立即终止后面所有规则的执行。**

```
package rules
import com.mashibing.drools.entity.RhsHaftEntity

/*
 用于测试Drools RHS: haft
*/


rule "rhs_haft_1"
    when
        $rhsHaftEntity:RhsHaftEntity(age <= 10)
    then
        drools.halt();
        System.out.println("规则 rhs_haft_1 触发");
end

rule "rhs_haft_2"
    when
        $rhsHaftEntity:RhsHaftEntity(age <= 20)
    then
        System.out.println("规则 rhs_haft_2 触发");
end
```

6.5.2 getWorkingMemory

getWorkingMemory方法的作用是返回工作内存对象。

```
rule "rhs_get_working_memory_1"
    when
        $rhsDroolsMethodsEntity:RhsDroolsMethodsEntity(age <= 10)
    then
        System.out.println(drools.getWorkingMemory());
        System.out.println("规则 rhs_get_working_memory_1 触发");
end
```

6.5.3 getRule

getRule方法的作用是返回规则对象。

```
rule "rhs_rule_2"
    when
        $rhsDroolsMethodsEntity:RhsDroolsMethodsEntity(age <=20)
    then
        System.out.println(drools.getRule());
        System.out.println("规则 rhs_rule_2 触发");
end
```

6.6 规则文件编码规范（重要）

我们在进行drl类型的规则文件编写时尽量遵循如下规范：

- 所有的规则文件(.drl)应统一放在一个规定的文件夹中，如：/rules文件夹
- 书写的每个规则应尽量加上注释。注释要清晰明了，言简意赅
- 同一类型的对象尽量放在一个规则文件中，如所有Student类型的对象尽量放在一个规则文件中
- 规则结果部分(RHS)尽量不要有条件语句，如if(...)，尽量不要有复杂的逻辑和深层次的嵌套语句
- 每个规则最好都加上salience属性，明确执行顺序
- Drools默认dialect为"Java"，尽量避免使用dialect "mvel"

7. WorkBench

7.1 WorkBench简介

WorkBench是KIE组件中的元素，也称为KIE-WB，是Drools-WB与JBPM-WB的结合体。它是一个**可视化的规则编辑器**。WorkBench其实就是一个war包。

WorkBench经过几次版本迭代，已经不提供tomcat启动的war包，综合考虑，本课程仍然采用 tomcat版本作为演示。

环境：

- apache-tomcat-9.0.29
- kie-drools-wb-7.6.0.Final-tomcat8 下载地址： **Drools - Download**

说明：

准备jar包：需要放到tomcat lib中，否则启动失败

具体安装步骤：

7.1.1 配置 Tomcat

1.修改tomcat-user.xml,添加用户

```
<?xml version="1.0" encoding="UTF-8"?>
<tomcat-users xmlns="http://tomcat.apache.org/xml"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://tomcat.apache.org/xml tomcat-users.xsd"
              version="1.0">

  <!--定义admin角色-->
  <role rolename="admin"/>
  <!--定义一个用户，用户名为kie，密码为kie，对应的角色为admin角色-->
  <user username="kie-web" password="kie-web123" roles="admin"/>
  <user username="admin" password="admin" roles="manager-gui,manager-script,manager-
jmx,manager-status"/>
</tomcat-users>
```

此账号密码用于登录WorkBench管理控制台

2.修改server.xml

```
      <Host name="localhost"  appBase="webapps"
            unpackWARs="true" autoDeploy="true">

        <!-- SingleSignOn valve, share authentication between web applications
             Documentation at: /docs/config/valve.html -->
        <!--
        <Valve className="org.apache.catalina.authenticator.SingleSignOn" />
        -->

        <!-- Access log processes all example.
             Documentation at: /docs/config/valve.html
             Note: The pattern used is equivalent to using pattern="common" -->
        <Valve className="org.apache.catalina.valves.AccessLogValve" directory="logs"
               prefix="localhost_access_log" suffix=".txt"
               pattern="%h %l %u %t &quot;%r&quot; %s %b" />
        <Valve className="org.kie.integration.tomcat.JACCValve"/>
      </Host>
```

host节点下添加

3.复制jar到tomcat根目录的lib下面：

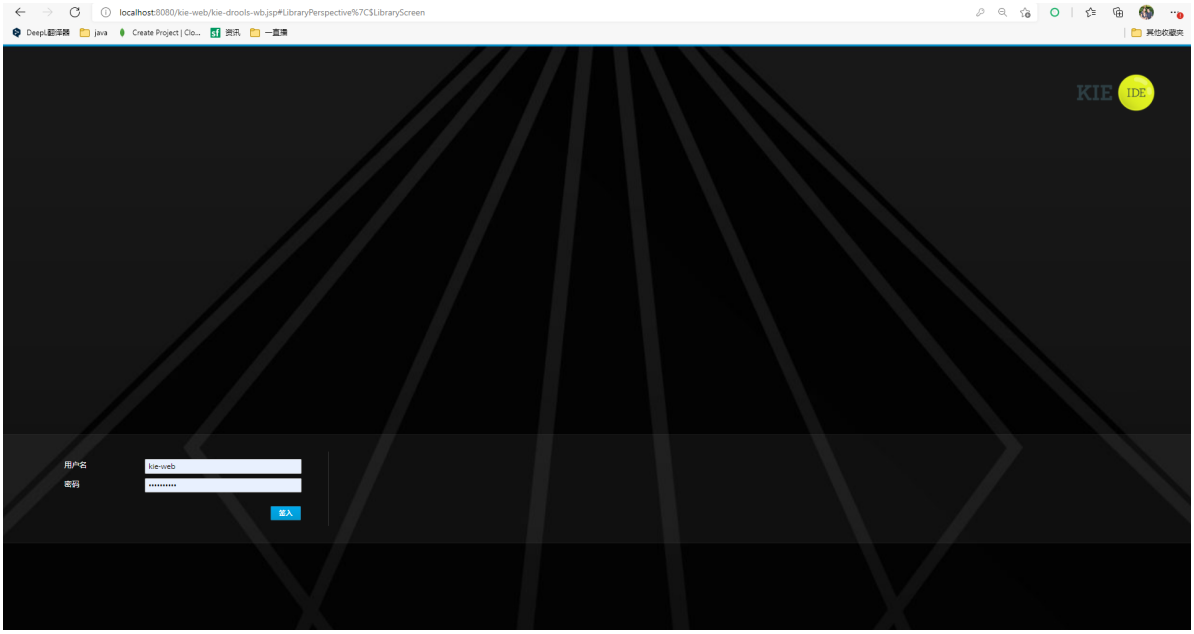
```
kie-tomcat-integration-7.10.0.Final.jar
javax.security.jacc-api-1.5.jar
slf4j-api-1.7.25.jar
```

4.复制 kie-drools-wb-7.6.0.Final-tomcat8.war 到tomcat webapp下面并修改成kie-web.war

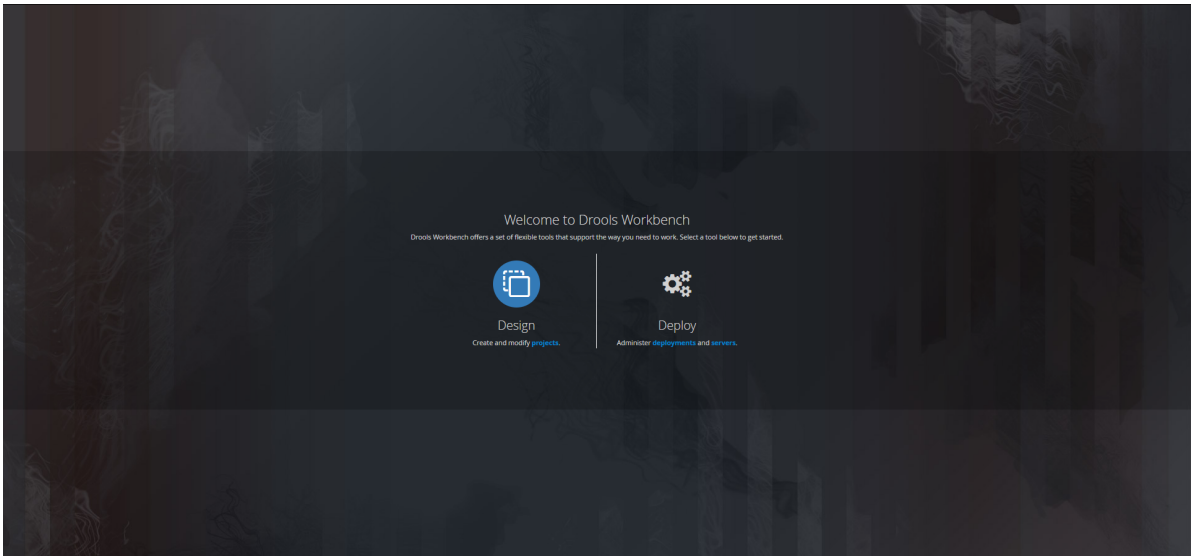
7.1.2启动服务器

启动tomcat

访问 http://localhost:8080/kie-web ，可以看到WorkBench的登录页面。使用前面创建的kie-web/kie-web123登录



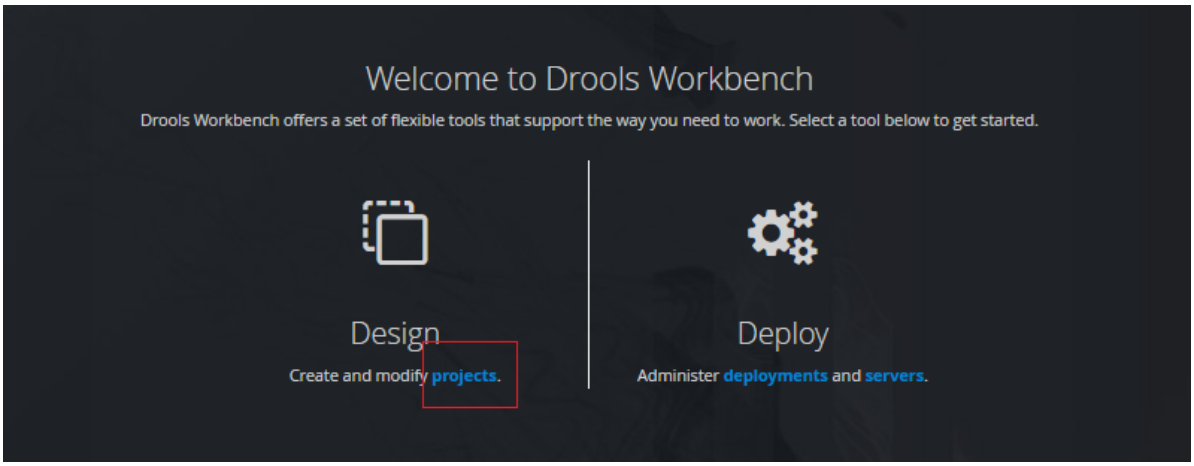
登录成功后进入系统首页：

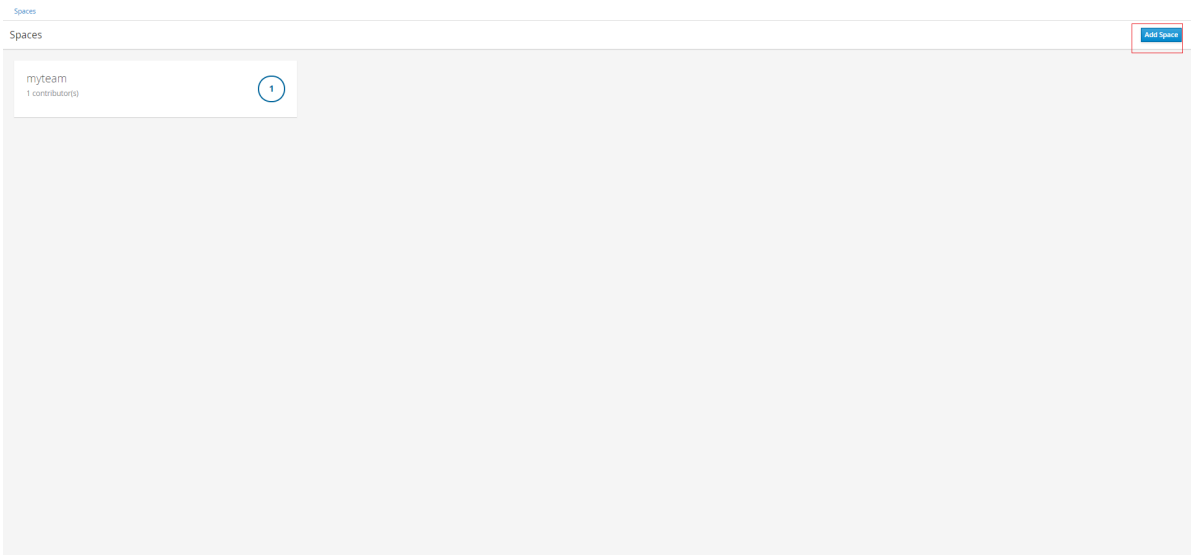


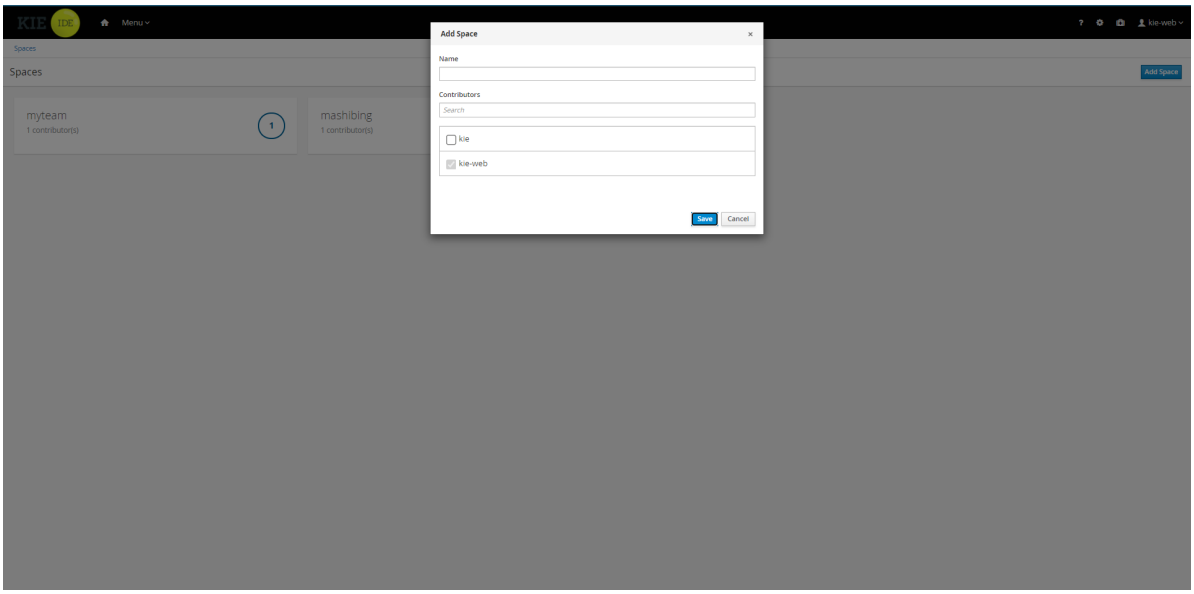
7.2WorkBench使用

7.2.1创建空间、项目

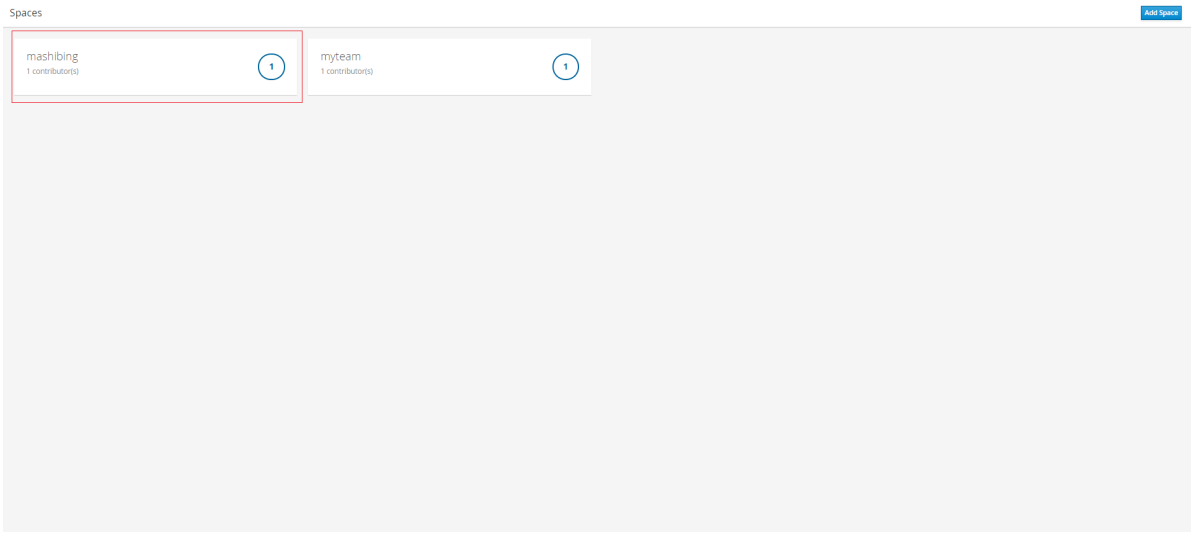
首页中点击 project，创建空间



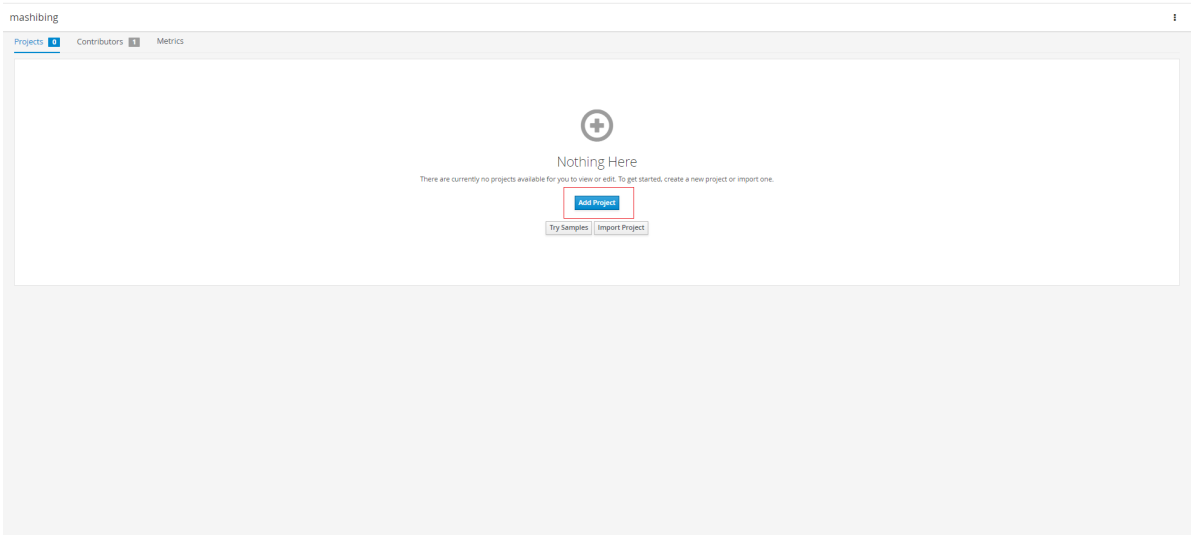




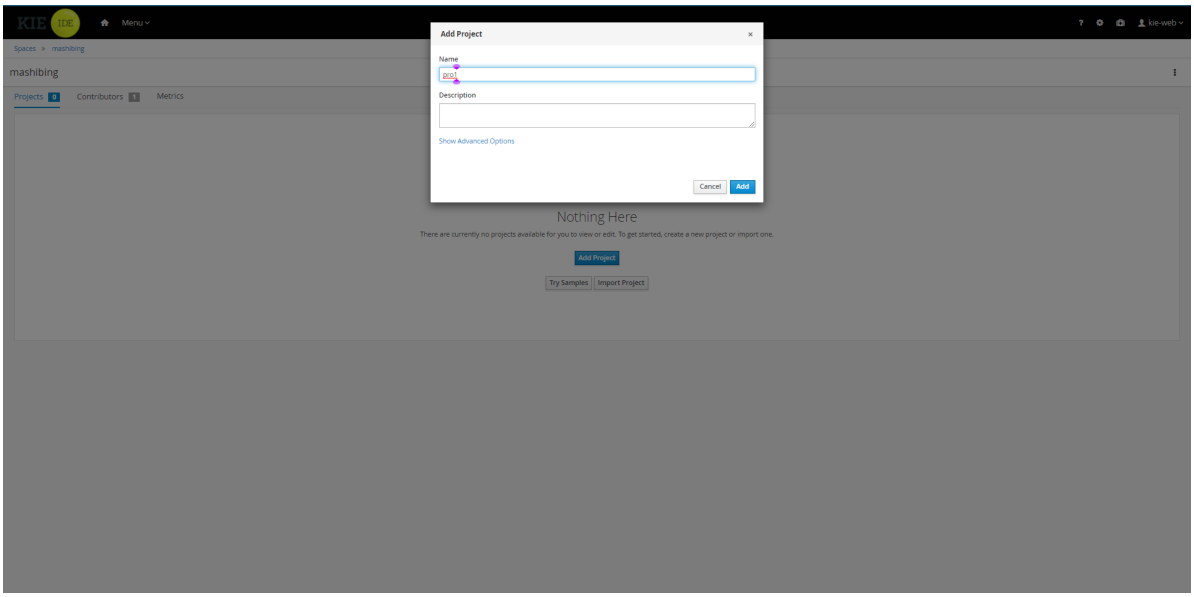
我们创建一个 mashibing 的工作空间。点击 Save，保存。



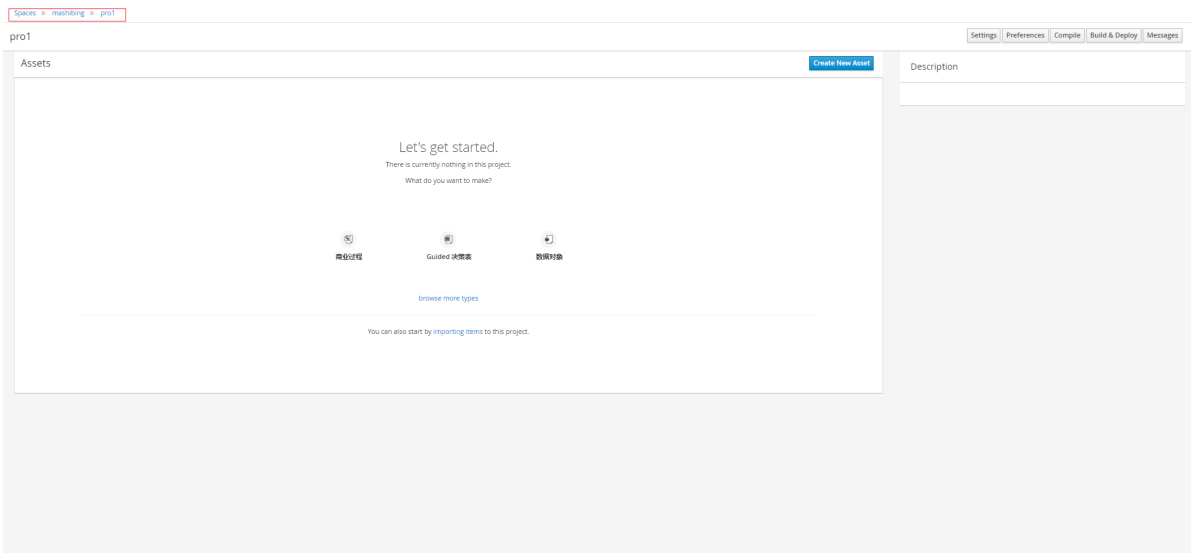
点击工作空间当中的 mashibing，进入空间



点击Add Project添加项目



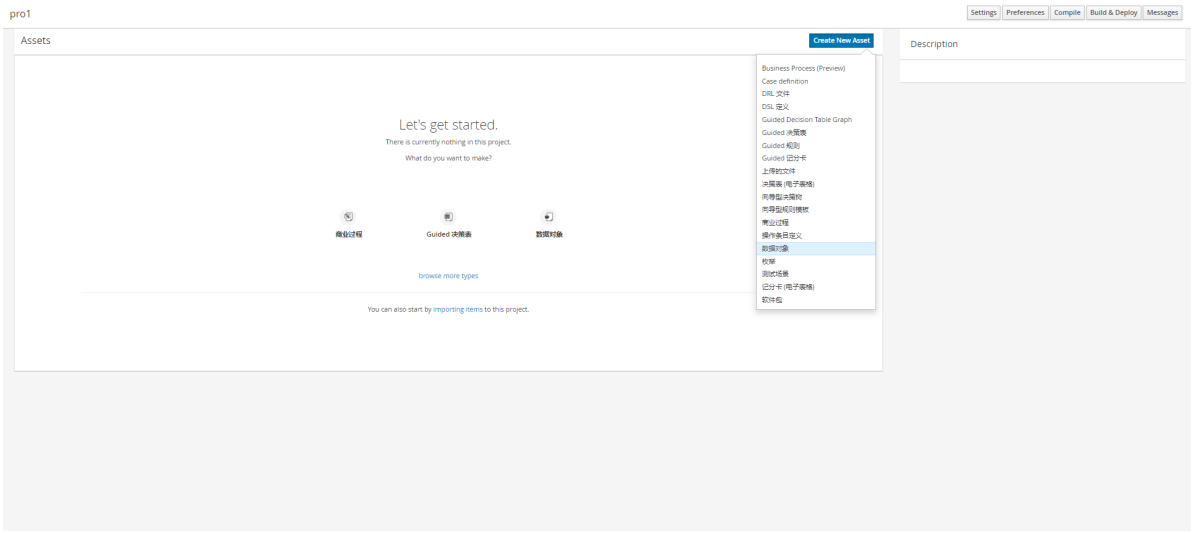
成功后，我们可以看见下图



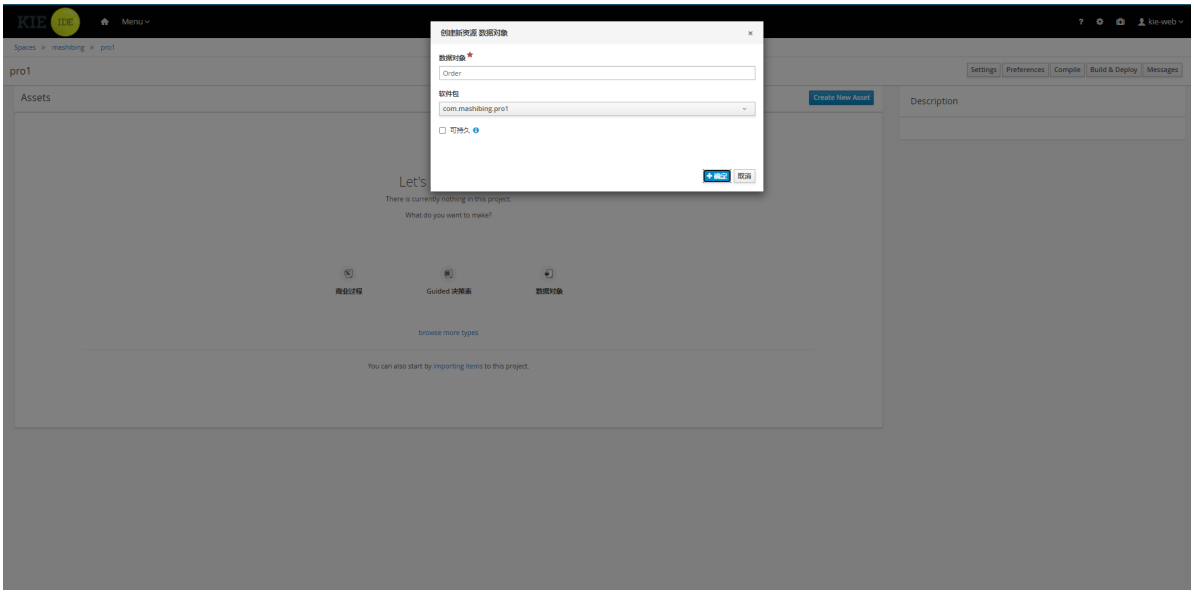
左上角的导航条，可以在空间和project之间切换

7.2.2创建数据对象和drl文件

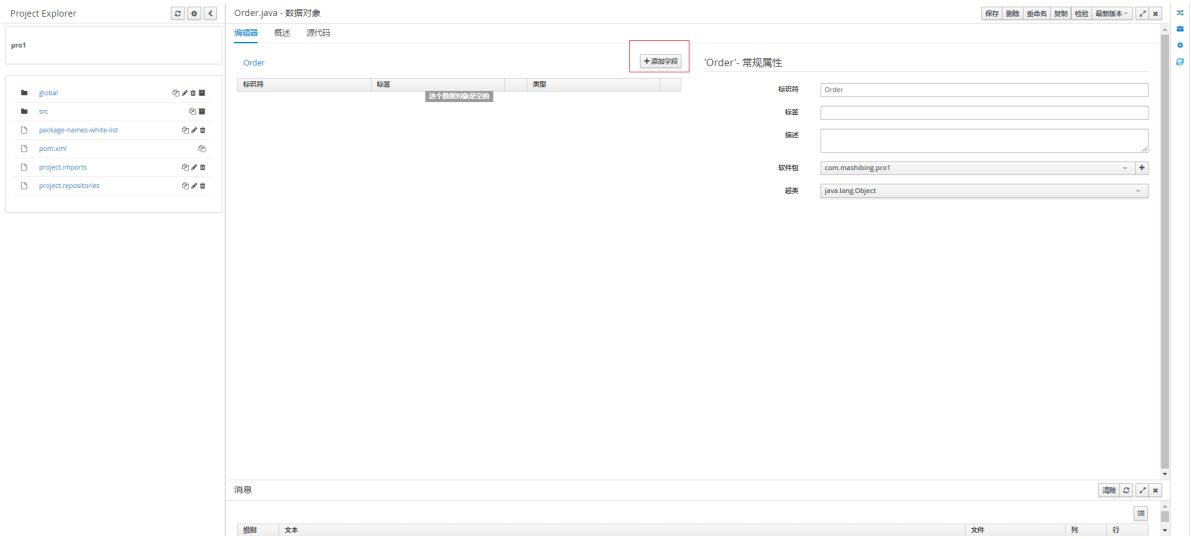
切换到pro1项目内，点击 Create New Assert



选中数据对象：

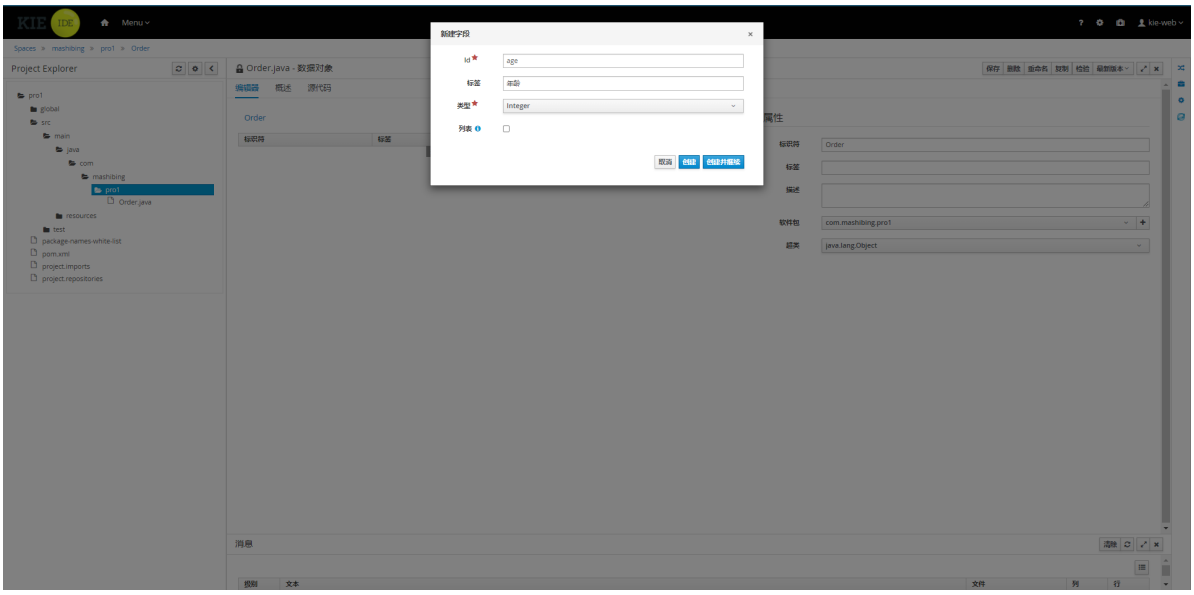


输入Order，点击确定，成功后跳转如下页面

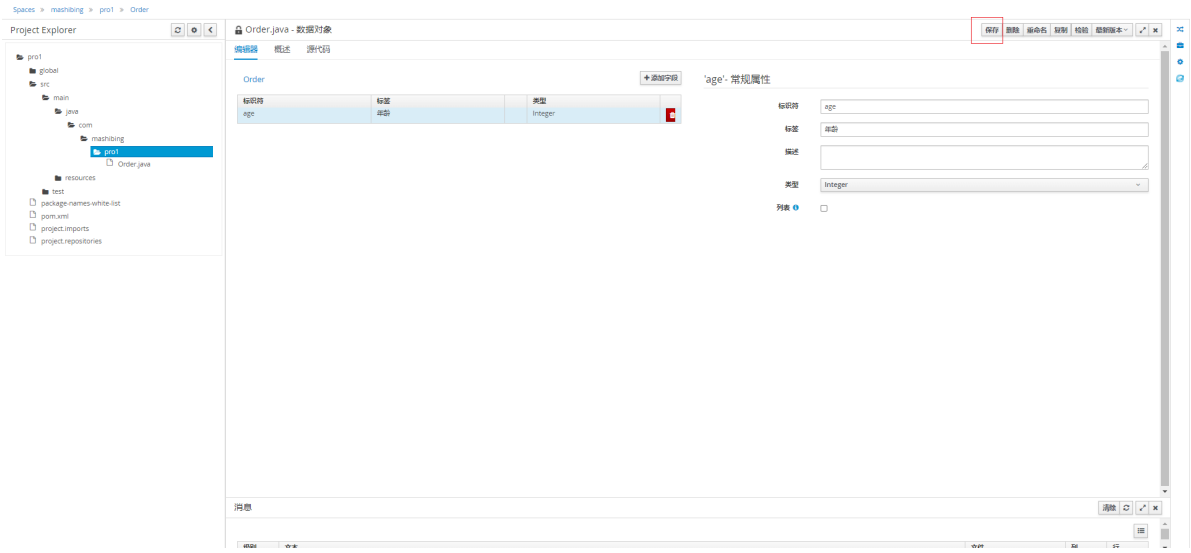


Order相当于我们代码中的实体类，在左侧 Project Explorer视图中，可以看见项目结构

接下来添加字段，点击添加字段按钮：



ID 位置，输入java bean的字段，标签是备注信息，类型选择对应的字段类型，保存，点击创建，关闭弹窗，点击创建并继续，可以继续创建。



点击右上角的保存，至此，一个数据对象我们就创建完成，可以在源代码中查看代码内容。

接下来我们创建一个drl文件，创建过程跟创建bean类似，drl文件内容如下

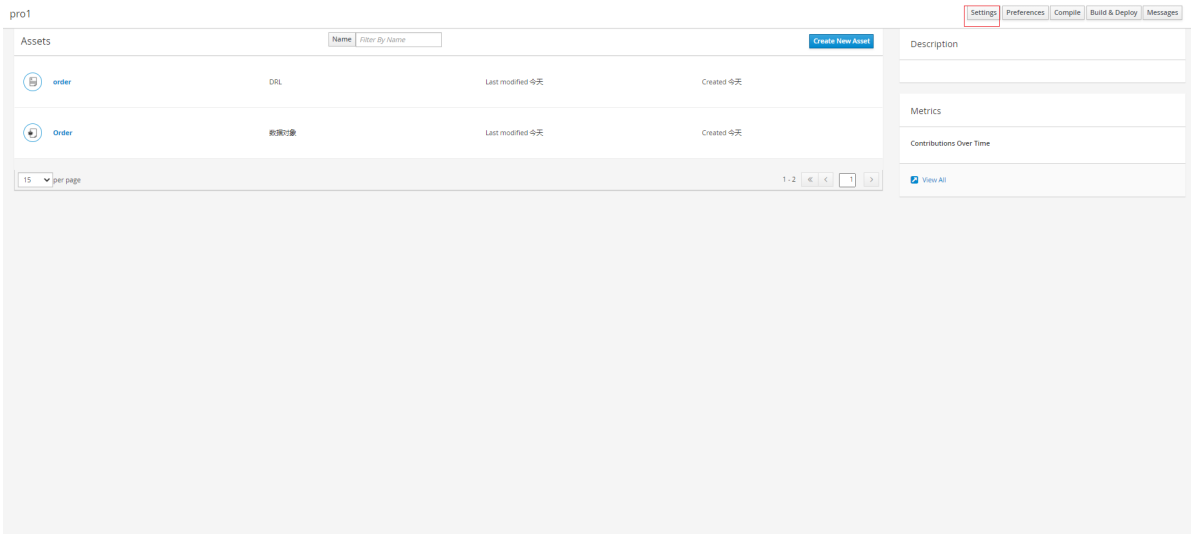
```
package com.mashibing.pro1;

rule "rule_1"
    when
        $order:Order(age > 10)
    then
        System.out.print("rule run...");
end
```

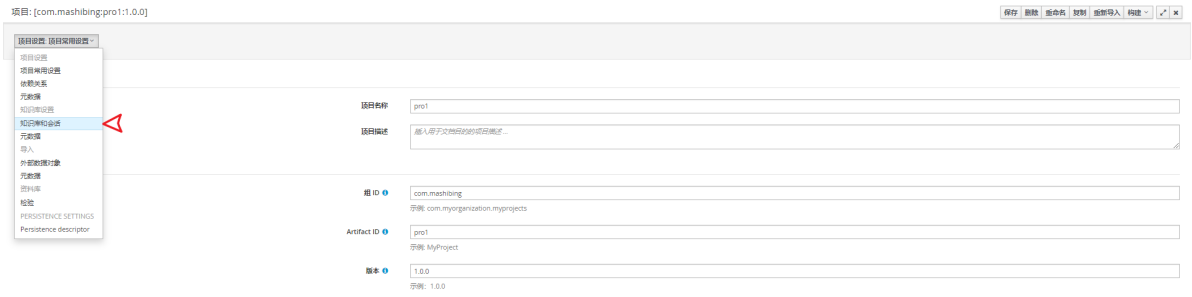
保存之后，点击导航条回到项目主页

7.2.3 设置KieBase+KieSession

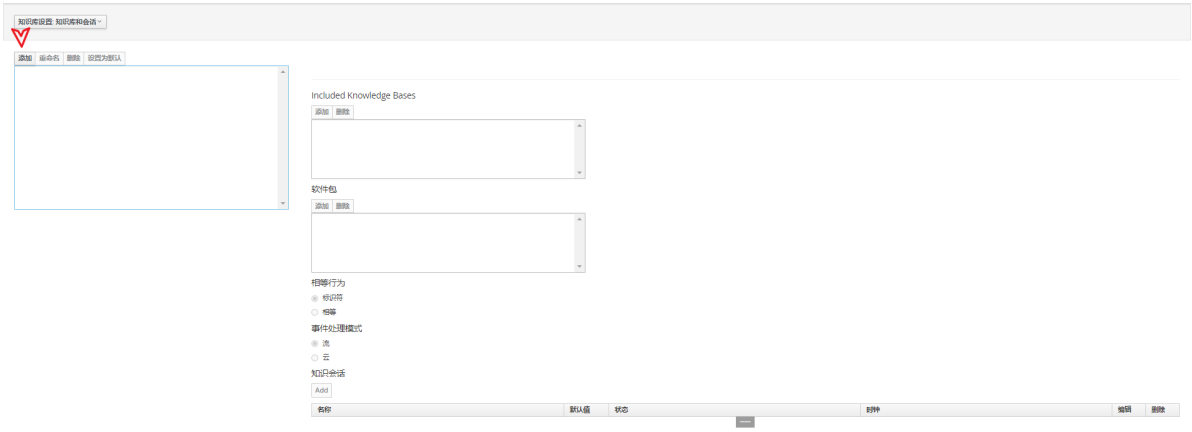
项目首页点击Settings



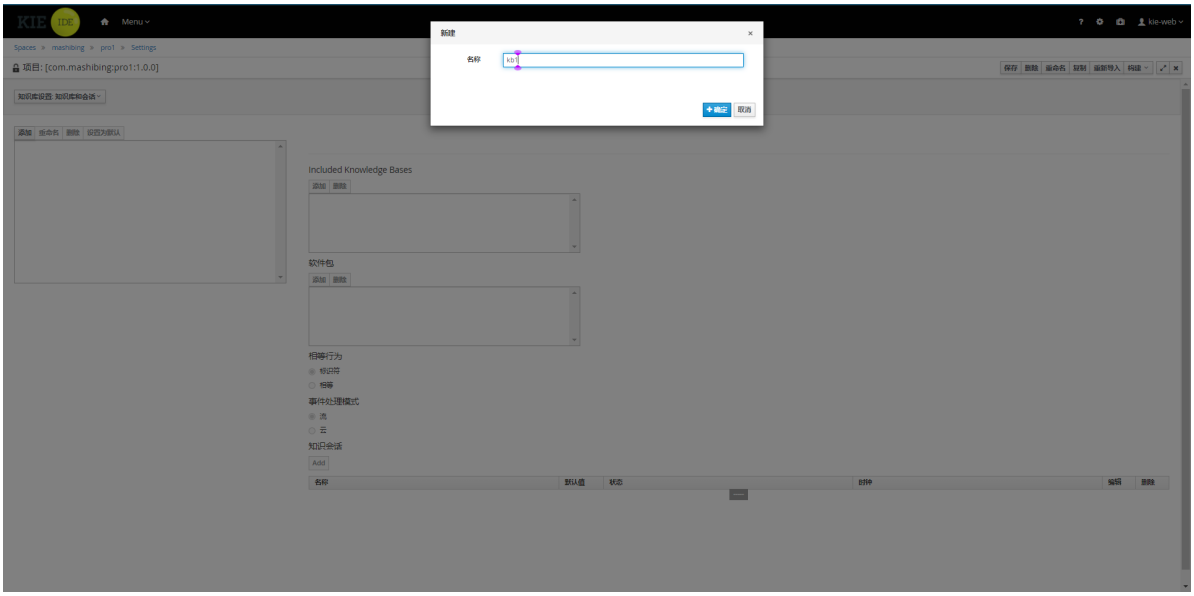
选择知识库跟会话



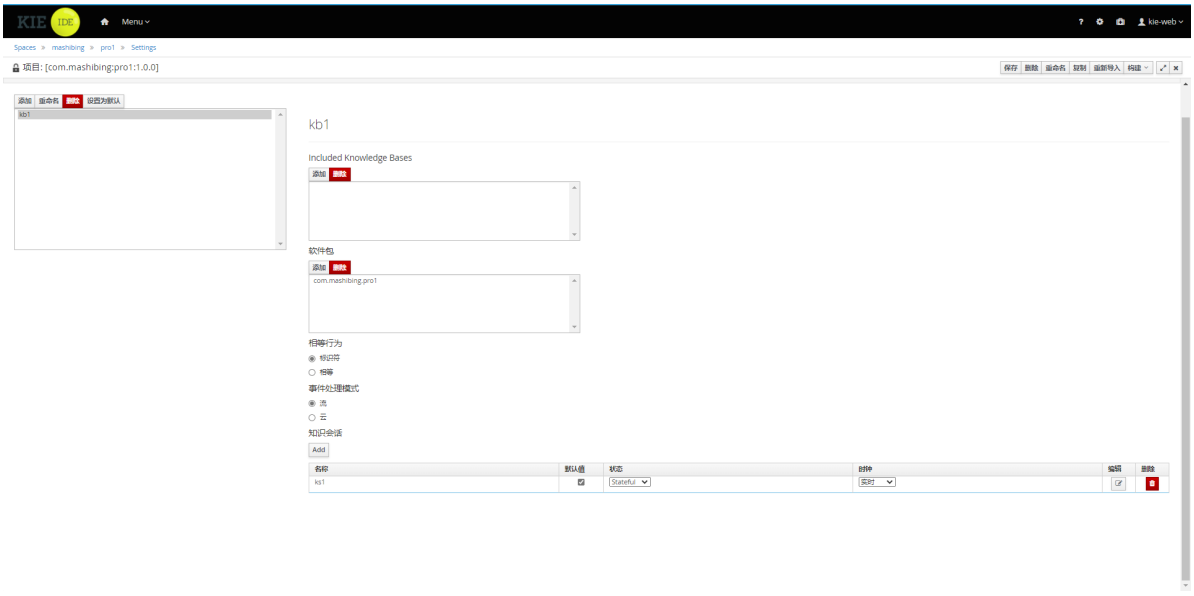
javascript:



弹出窗口，输入Kiebase名称即可，我们以kb1为例



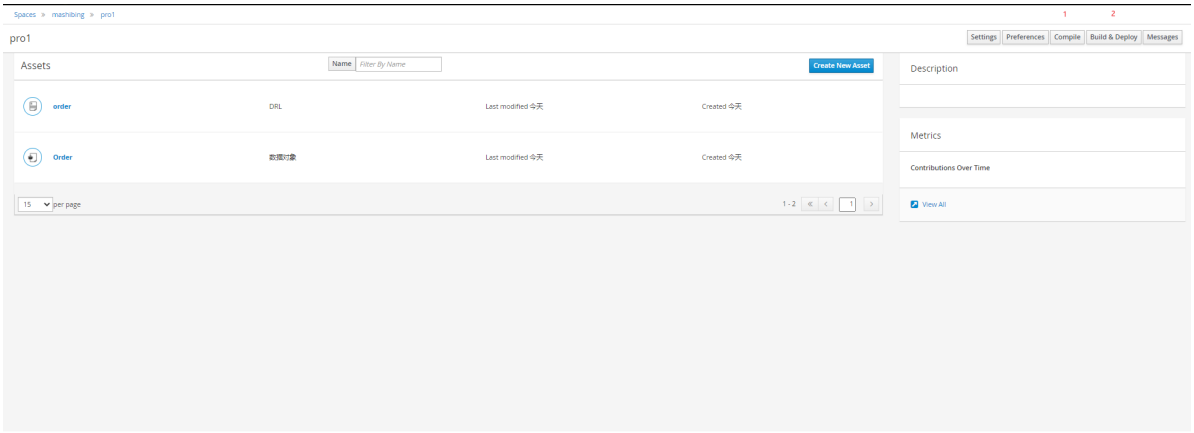
同理，我们补充完软件包信息，添加只是会话，即kiesession



操作完成后，不要忘记保存，此时，我们可在Project Explorer视图中，resource/META-INF/kmodule.xml中看见如下信息

```
<kmodule xmlns="http://www.drools.org/xsd/kmodule"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <kbase name="kb1" default="false" eventProcessingMode="stream"
equalsBehavior="identity" packages="com.mashibing.pro1">
    <ksession name="ks1" type="stateful" default="true" clockType="realtime"/>
  </kbase>
</kmodule>
```

导航回到项目首页，进行编译发布



发布成功后，我们可以在maven仓库中看到对应的jar

也可以访问： http://localhost:8080/kie-web/maven2/com/mashibing/pro1/1.0.0/pro1-1.0.0.jar 验证是否发布成功

7.2.4 代码使用

```
@Test
    public void test() throws Exception{
        //通过此URL可以访问到maven仓库中的jar包
        //URL地址构成：http://ip地址:Tomcat端口号/WorkBench工程名/maven2/坐标/版本号/xxx.jar
        String url = "http://localhost:8080/kie-
web/maven2/com/mashibing/pro1/1.0.0/pro1-1.0.0.jar";

        KieServices kieServices = KieServices.Factory.get();
        UrlResource resource = (UrlResource)
kieServices.getResources().newUrlResource(url);
        //认证
        resource.setUsername("kie-web");
        resource.setPassword("kie-web123");
        resource.setBasicAuthentication("enabled");
```

```
        KieRepository repository = kieServices.getRepository();

        //通过输入流读取maven仓库中的jar包数据，包装成KieModule模块添加到仓库中
        KieModule kieModule =
repository.addKieModule(kieServices.getResources().newInputStreamResource(resource.get
InputStream()));

        KieContainer kieContainer =
kieServices.newKieContainer(kieModule.getReleaseId());
        KieSession session = kieContainer.newKieSession();

        Order order = new Order();
        order.setName("张三");
        order.setAge(30);
        session.insert(order);

        session.fireAllRules();
        session.dispose();
    }
```

我们用URL流的方式，获取jar资源，并构造kiesession对象，即可动态访问workbench中的规则

8 其他

8.1 有状态session和无状态session

无状态session

无状态的KIE会话是一个不使用推理来对事实进行反复修改的会话。在无状态的KIE会话中，来自KIE会话先前调用的数据（先前的会话状态）在会话调用之间被丢弃，而在有状态的KIE会话中，这些数据被保留。一个无状态的KIE会话的行为类似于一个函数，因为它产生的结果是由KIE基础的内容和被传入KIE会话以在特定时间点执行的数据决定的。KIE会话对以前传入KIE会话的任何数据都没有记忆。

使用方法类似如下代码：

```
    @Test
    public void testStatelessSession() {
        StatelessKieSession statelessKieSession = kieBase.newStatelessKieSession();
        List<Command> cmds = new ArrayList<>();
        KieSessionEntity kieSessionEntity = new KieSessionEntity();
        kieSessionEntity.setNum(10);
        kieSessionEntity.setValid(false);
        cmds.add(CommandFactory.newInsert(kieSessionEntity, "kieSessionEntity"));
        statelessKieSession.execute(CommandFactory.newBatchExecution(cmds));

        System.out.println(kieSessionEntity);
    }
```

简单说来，无状态session执行的时候,不需要调用 fireAllRules(),也不需要执行dispose()，代码执行完execute之后，即销毁所有的数据。

使用场景：比如上述的校验num
验证数据: 比如计算积分，按揭房贷等

路有消息：比如对邮件排序，发送邮件等，行为类的场景

有状态session

有状态的KIE会话是一个使用推理来对事实进行反复修改的会话。在有状态的KIE会话中，来自KIE会话先前调用的数据（先前的会话状态）在会话调用之间被保留，而在无状态的KIE会话中，这些数据被丢弃了。

对比无状态session，有状态session调用fireAllRules()的时候采取匹配规则，就会执行规则匹配，除非遇见dispose()

示例：

数据模型

```
public class Room {
  private String name;
  // Getter and setter methods
}

public class Sprinkler {
  private Room room;
  private boolean on;
  // Getter and setter methods
}

public class Fire {
  private Room room;
  // Getter and setter methods
}

public class Alarm { }
```

规则文件

```
rule "When there is a fire turn on the sprinkler"
when
  Fire($room : room)
  $sprinkler : Sprinkler(room == $room, on == false)
then
  modify($sprinkler) { setOn(true) };
  System.out.println("Turn on the sprinkler for room "+$room.getName());
end

rule "Raise the alarm when we have one or more fires"
when
    exists Fire()
then
    insert( new Alarm() );
    System.out.println( "Raise the alarm" );
end
```

```
rule "Cancel the alarm when all the fires have gone"
when
    not Fire()
    $alarm : Alarm()
then
    delete( $alarm );
    System.out.println( "Cancel the alarm" );
end


rule "Status output when things are ok"
when
    not Alarm()
    not Sprinkler( on == true )
then
    System.out.println( "Everything is ok" );
end
```

代码

```
KieSession ksession = kContainer.newKieSession();

String[] names = new String[]{"kitchen", "bedroom", "office", "livingroom"};
Map<String,Room> name2room = new HashMap<String,Room>();
for( String name: names ){
    Room room = new Room( name );
    name2room.put( name, room );
    ksession.insert( room );
    Sprinkler sprinkler = new Sprinkler( room );
    ksession.insert( sprinkler );
}

ksession.fireAllRules();
```

输出

```
Console output
> Everything is ok
```

此时还可以继续输入

```
Fire kitchenFire = new Fire( name2room.get( "kitchen" ) );
Fire officeFire = new Fire( name2room.get( "office" ) );

FactHandle kitchenFireHandle = ksession.insert( kitchenFire );
FactHandle officeFireHandle = ksession.insert( officeFire );

ksession.fireAllRules();
```

```
Console output
> Raise the alarm
> Turn on the sprinkler for room kitchen
> Turn on the sprinkler for room office
```

继续输入

```
ksession.delete( kitchenFireHandle );
ksession.delete( officeFireHandle );

ksession.fireAllRules();
```

输出

```
Console output

> Cancel the alarm
> Turn off the sprinkler for room office
> Turn off the sprinkler for room kitchen
> Everything is ok
```

使用场景：

- 监测，如监测股票市场并使购买过程自动化
- 诊断，如运行故障查找过程或医疗诊断过程
- 物流，如包裹跟踪和配送供应
- 确保合规性，如验证市场交易的合法性

参考文档：

【1】 百度百科：规则引擎： https://baike.baidu.com/item/%E8%A7%84%E5%88%99%E5%BC%95%E6%93%8E/3076955?fr=aladdin

【2】开源规则引擎 drools： https://blog.csdn.net/sdmxdzb/article/details/81461744

【3】drools官网： Drools - Business Rules Management System (Java™, Open Source)