

Easy Rules --Easy的规则引擎

为什么要用规则引擎？

我遇到了个需求，需求要求进行非常灵活动态的配置，估算了一下要用四层循环，在四层循环中每个参数的不同可能带来不同的逻辑流程，也就是四个for循环里要进行一堆if else嵌套，想想画面就太美。

还有万一刚写完多层嵌套的嵌套逻辑，需求突然得改，心情肯定是崩溃的。而规则引擎此时也显得更加灵活，后续修改逻辑时不用回到眼花缭乱的代码中进行修改，只需要对外部的规则进行修改即可，实现起来会比较更加优雅。

规则引擎还有个好处就是，逻辑更加透明，原来if else都是程序员写的，可能表达的跟实际需求不太一样，而上级领导又难以发现，又或者某个程序员如果在某优惠券发放逻辑中留一个只有自己知道的后门，那就比较可怕了。所以规则引擎也给代码逻辑带来更具有管理性的优点。

什么是Easy Rules？

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

运行环境

Easy Rules是一个Java库，需要运行在Java 1.7及以上。

maven依赖

```
<!--easy rules核心库-->
<dependency>
  <groupId>org.jeasy</groupId>
  <artifactId>easy-rules-core</artifactId>
  <version>3.3.0</version>
</dependency>

<!--规则定义文件格式，支持json,yaml等-->
<dependency>
  <groupId>org.jeasy</groupId>
  <artifactId>easy-rules-support</artifactId>
  <version>3.3.0</version>
</dependency>

<!--支持mvel规则语法库-->
<dependency>
  <groupId>org.jeasy</groupId>
```

```
<artifactId>easy-rules-mvel</artifactId>
<version>3.3.0</version>
</dependency>
```

规则 Rule

大多数业务规则可以由以下定义表示：

- 名称 name：规则命名空间中的唯一规则名称
- 说明 description：规则的简要说明
- 优先级 priority：相对于其他规则的规则优先级
- 事实 fact：去匹配规则时的一组已知事实
- 条件 condition：为了匹配该规则，在给定某些事实的情况下应满足的一组条件
- 动作 action：当条件满足时要执行的一组动作（可以添加/删除/修改事实）

Easy Rules为定义业务规则的每个关键点提供了抽象。

在Easy Rules中，一个规则由 `Rule` 接口表示：

```
public interface Rule {

    /**
     * 直译：评估。
     * @return 如果提供的事实适用于该规则返回true，否则，返回false
     */
    boolean evaluate(Facts facts);

    /**
     * 如果评估结果为true则会执行execute
     * @throws 如果在执行过程中发生错误将抛出Exception
     */
    void execute(Facts facts) throws Exception;

    //Getters and setters for rule name, description and priority omitted.

}
```

evaluate方法封装了必须求值为TRUE才能触发规则的条件。

execute方法封装了在满足规则条件时应执行的操作。条件和动作 Condition and Action 接口表示。

规则可以用两种不同的方式定义：

- 通过在POJO上添加注释，以声明方式定义
- 通过RuleBuilder API，以编程方式定义
- 通过在yml配置文件定义

1. 用注解定义规则

这些是定义规则的最常用方法，但如果需要，还可以实现 `Rule` 接口或继承 `BasicRule` 类。

```
@Rule(name = "my rule", description = "my rule description", priority = 1)
public class MyRule {

    @Condition
```

```

    public boolean when(@Fact("fact") fact) {
        //my rule conditions
        return true;
    }

    @Action(order = 1)
    public void then(Facts facts) throws Exception {
        //my actions
    }

    @Action(order = 2)
    public void finally() throws Exception {
        //my final actions
    }
}

```

@Condition注解标记计算规则条件的方法。此方法必须是公共的，可以有一个或多个用@Fact注解的参数，并返回布尔类型。只有一个方法能用@Condition注解。

@Action注解标记要执行规则操作的方法。规则可以有多个操作。可以使用order属性按指定的顺序执行操作。默认情况下，操作的顺序为0。

2. 用RuleBuilder API定义规则

```

Rule rule = new RuleBuilder()
    .name("myRule")
    .description("myRuleDescription")
    .priority(3)
    .when(condition)
    .then(action1)
    .then(action2)
    .build();

```

在这个例子中，Condition实例condition，Action实例是action1和action2。

3. 用yaml定义规则

```

name: "alcohol rule"
description: "children are not allowed to buy alcohol"
priority: 2
condition: "person.isAdult() == false"
actions:
  - "System.out.println(\"Shop: Sorry, you are not allowed to buy alcohol\");"

```

抽到配置文件中非常易于管理，但是没有代码提示补全，定义和修改起来比较难受，个人还是倾向第二种。

定义事实 Facts

Facts API是一组事实的抽象，在这些事实上检查规则。在内部，Facts实例持有HashMap<String, Object>，这意味着：

- 事实需要命名，应该有一个唯一的名称，且不能为空
- 任何Java对象都可以充当事实

这里有一个实例定义事实：

```
Facts facts = new Facts();
facts.add("rain", true);
```

Facts 能够被注入规则条件，action 方法使用 @Fact 注解。在下面的规则中，rain 事实被注入itRains方法的rain参数：

```
@Rule
class WeatherRule {

    @Condition
    public boolean itRains(@Fact("rain") boolean rain) {
        return rain;
    }

    @Action
    public void takeAnUmbrella(Facts facts) {
        System.out.println("It rains, take an umbrella!");
        // can add/remove/modify facts
    }

}
```

Facts 类型参数 被注入已知的 facts中 (像action方法 takeAnUmbrella 一样)。

如果缺少注入的fact, 这个引擎会抛出 RuntimeException 异常。

定义规则引擎

从版本3.1开始，Easy Rules提供了RulesEngine接口的两种实现：

- DefaultRulesEngine：根据规则的自然顺序（默认为优先级）应用规则。
- InferenceRulesEngine：持续对已知事实应用规则，直到不再应用规则为止。

创建一个规则引擎

要创建规则引擎，可以使用每个实现的构造函数：

```
RulesEngine rulesEngine = new DefaultRulesEngine();
// or
RulesEngine rulesEngine = new InferenceRulesEngine();
```

然后，您可以按以下方式触发注册规则：

```
rulesEngine.fire(rules, facts);
```

规则引擎参数

Easy Rules 引擎可以配置以下参数：

Parameter	Type	Required	Default
rulePriorityThreshold	int	no	MaxInt
skipOnFirstAppliedRule	boolean	no	false
skipOnFirstFailedRule	boolean	no	false
skipOnFirstNonTriggeredRule	boolean	no	false

- skipOnFirstAppliedRule：告诉引擎规则被触发时跳过后面的规则。
- skipOnFirstFailedRule：告诉引擎在规则失败时跳过后面的规则。
- skipOnFirstNonTriggeredRule：告诉引擎一个规则不会被触发跳过后面的规则。
- rulePriorityThreshold：告诉引擎如果优先级超过定义的阈值，则跳过下一个规则。版本3.3已经不支持更改，默认MaxInt。

可以使用RulesEngineParameters API指定这些参数：

```
RulesEngineParameters parameters = new RulesEngineParameters()
    .rulePriorityThreshold(10)
    .skipOnFirstAppliedRule(true)
    .skipOnFirstFailedRule(true)
    .skipOnFirstNonTriggeredRule(true);

RulesEngine rulesEngine = new DefaultRulesEngine(parameters);
```

如果要从引擎获取参数，可以使用以下代码段：

```
RulesEngineParameters parameters = myEngine.getParameters();
```

这允许您在创建引擎后重置引擎参数。

引用文章: <https://zhuanlan.zhihu.com/p/91158525>

再结合官方的demo来理解上面的概念

```
//引擎发射器
public class Launcher {

    public static void main(String[] args) {
        // 定义事实，就是当前的程序的真实条件。温度30度。
        Facts facts = new Facts();
        facts.put("temperature", 30);

        // 定义rule
        Rule airConditioningRule = new RuleBuilder()
            .name("air conditioning rule")
            //condition 当满足“很热”这个条件
            .when(itIsHot())
            //action 就执行“空调调低一度”的动作
            .then(decreaseTemperature())
            .build();

        Rules rules = new Rules();
        rules.register(airConditioningRule);
    }
}
```

```

// 如果满足条件满足就会持续执行，直到条件不满足为止
RulesEngine rulesEngine = new InferenceRulesEngine();
//引擎点火
rulesEngine.fire(rules, facts);
}

}

```

condition 和 action需要我们自己定义一下, 我们可以定义一个类去实现接口, 也可以用lambda实现函数式接口, 个人比较喜欢后者这种方式 :)

```

//高温条件 等价于 fact->facts.get("temperature")>25
public class HighTemperatureCondition implements Condition {

    static HighTemperatureCondition itIsHot() {
        return new HighTemperatureCondition();
    }

    @Override
    public boolean evaluate(Facts facts) {
        Integer temperature = facts.get("temperature");
        return temperature > 25;
    }

}

```

```

//调低温度动作 等价于 fact-> { ...; facts.put("temperature", temperature - 1); }
public class DecreaseTemperatureAction implements Action {

    static DecreaseTemperatureAction decreaseTemperature() {
        return new DecreaseTemperatureAction();
    }

    @Override
    public void execute(Facts facts) {
        System.out.println("It is hot! cooling air..");
        Integer temperature = facts.get("temperature");
        System.out.println("now temperature is " + temperature + ",turn it to" +
            (temperature - 1));
        facts.put("temperature", temperature - 1);
    }

}

```

执行Launcher, 打印结果:

```

It is hot! cooling air..
now temperature is 30,turn it to29
It is hot! cooling air..
now temperature is 29,turn it to28
It is hot! cooling air..
now temperature is 28,turn it to27
It is hot! cooling air..
now temperature is 27,turn it to26
It is hot! cooling air..
now temperature is 26,turn it to25

```

果真很easy，但是这种简单条件用规则引擎有点大材小用，反而不如用if else. 所以

自己动手写一下.

```
//目标逻辑
if(isRainy){
    if(hasUmbrella){
        if(takeGoodUmbrella){
            //输出没被淋湿
        }else if(takeBadUmbrella){
            //输出伞坏了 被淋湿了
        }
    }else if(noUmbrella){
        //输出被淋湿了
    }
}else{
    //打印天气晴朗
}
```

接下来将if语句转换成rule规则

```
public class Launcher {
    public static void main(String[] args) {
        //定义规则
        Rule rainyRule = new RuleBuilder()
            .name("rainy rule")
            .description("if it rains then if person take umbrella?")
            .when(facts -> facts.get("rainy").equals("yes"))
            .then(facts -> System.out.println("it rains ..."))
            .priority(1)
            .build();

        Rule noUmbrellaRule = new RuleBuilder()
            .name("no umbrella rule")
            .description("person does not take umbrella")
            .when(facts -> facts.get("rainy").equals("yes") &&
facts.get("takeUmbrella").equals("no"))
            .then(facts -> System.out.println("and person dose not take an
umbrella. he gets wet."))
            .priority(3)
            .build();

        Rule hasGoodUmbrellaRule = new RuleBuilder()
            .name("has healthy umbrella rule")
            .description("person takes a good umbrella")
            .when(facts -> facts.get("rainy").equals("yes") &&
facts.get("takeUmbrella").equals("yes")
            && facts.get("umbrellaHeathy").equals("yes"))
            .then(facts -> System.out.println("and person takes a good
umbrella. he does not get wet."))
            .priority(4)
            .build();

        Rule hasBadUmbrellaRule = new RuleBuilder()
            .name("has bad umbrella rule")
            .description("person takes a bad umbrella")
```

```

        .when(facts -> facts.get("rainy").equals("yes") &&
facts.get("takeUmbrella").equals("yes")
        && facts.get("umbrellaHeathy").equals("no"))
        .then(facts -> System.out.println("and person takes a bad
umbrella. he still gets wet."))
        .priority(5)
        .build();

Rule sunnyRule = new RuleBuilder()
        .name("sunny rule")
        .description("it is sunny,everything is good")
        .when(facts -> facts.get("rainy").equals("no"))
        .then(facts -> System.out.println("sunshine is beautiful ..."))
        .priority(2)
        .build();

Rules rules = new
Rules(rainyRule,noUmbrellaRule,hasBadUmbrellaRule,hasGoodUmbrellaRule,sunnyRule)
;

//定义事实
Facts facts = new Facts();
Fact<String> weatherFact = new Fact<>("rainy", "yes");
Fact<String> hasUmbrellaFact = new Fact<>("takeUmbrella", "yes");
Fact<String> umbrellaHeath = new Fact<>("umbrellaHeathy", "no");
facts.add(weatherFact);
facts.add(hasUmbrellaFact);
facts.add(umbrellaHeath);

RulesEngine rulesEngine = new DefaultRulesEngine();
rulesEngine.fire(rules,facts);
}
}

```

```

//输出
it rains ...
and person takes a bad umbrella. he still gets wet.

```

将takeUmbrella改为"no"

```

//输出
it rains ...
and person dose not take umbrella. he gets wet.

```

将umbrellaHeathy改为"yes"

```

//输出
it rains ...
and person takes a good umbrella. he does not get wet.

```

将rainy改为"no"

```

//输出
sunshine is beautiful ...

```


那么这个逻辑已经开发完成了，虽然代码量看起来不少，但是很多的是对规则的命名和描述。如果再多两层if else，Rule看起来还是比if else清爽的。

和if else不同的是，你不再需要进行嵌套，你只需要关注具有行为的“叶子逻辑分支”，为每一个“叶子逻辑分支”定义一条Rule即可。

Fact类 参数是泛型，自定义一个类，在action中更改fact的属性和状态就可以玩出更复杂的动作了。

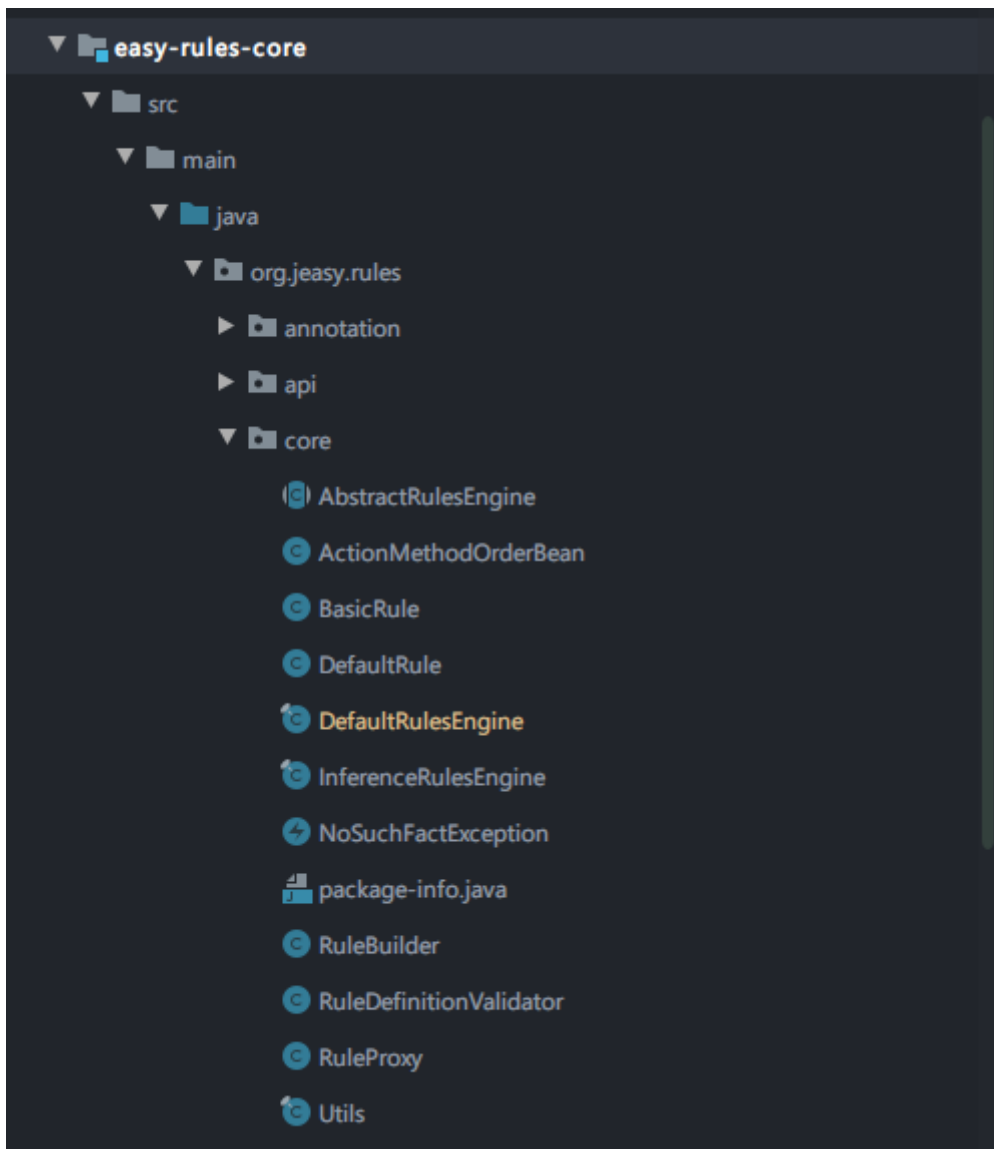
easy rule还提供复合规则的三种实现

- 1、UnitRuleGroup：单位规则组是充当单位的组合规则：**要么应用所有规则，要么不应用任何规则。**
- 2、ActivationRuleGroup：激活规则组是一个组合规则，**它会触发第一个适用的规则，而忽略该组中的其他规则**（XOR逻辑）。规则首先按组中的自然顺序（默认情况下为优先级）排序。利用这个组，在上面的例子，我们可以把好伞坏伞分为一组，晴朗下雨分为一组，相当于if else-if.
- 3、ConditionalRuleGroup：条件规则组是一个组合规则，其中具有最高优先级的规则作为条件：**如果具有最高优先级的规则求值为true，则将触发其余规则。**

手撕规则引擎

看完了规则引擎的使用，秉着求知的态度对高大上的规则引擎进行探索。

从github中拉下源码，核心逻辑就在这个包。



我们从DefaultRulesEngine中最具有仪式感的fire方法看起。

```
@Override
public void fire(Rules rules, Facts facts) {
    triggerListenersBeforeRules(rules, facts);
    doFire(rules, facts);
    triggerListenersAfterRules(rules, facts);
}
```

fire中triggerListenersBeforeRules、doFire、triggerListenersAfterRules. 显然关键在于doFire,监听器一会儿再看。

进入到doFire中这里循环了rules `for (Rule rule : rules) {`, 而rules就是我们fire时传入的, rules中的rule就是我们rules实例调用register放进去的rule.

在循环里面调用了rule.evaluate(facts) 我们可以猜到, 它对我们定义的事实进行了评估, 判断rule是否满足fact的条件.

```
boolean evaluationResult = false;
try {
    evaluationResult = rule.evaluate(facts);
}
```

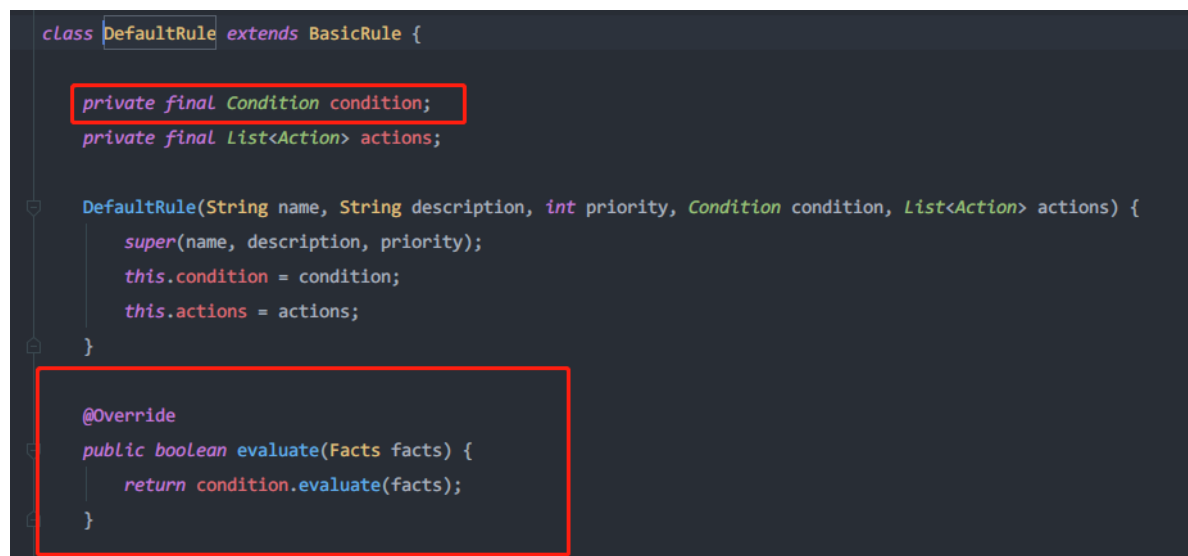
点进这个方法

看看evaluate的原理

我们看到Rule是一个接口，我们通过idea看看它的继承类。DefaultRule应该会看到我们想看的 -- evaluate被如何实现。



进入DefaultRule，它封装了Condition的evaluate方法，我们需要再进入Condition。



Condition也是接口，类上标注了函数式接口，除了evaluate方法之外，类中还有两个Condition常量，他们就是该类的两个lambda实现，一个包含永真evalute，一个永假。

```

    */
    @FunctionalInterface
    public interface Condition {

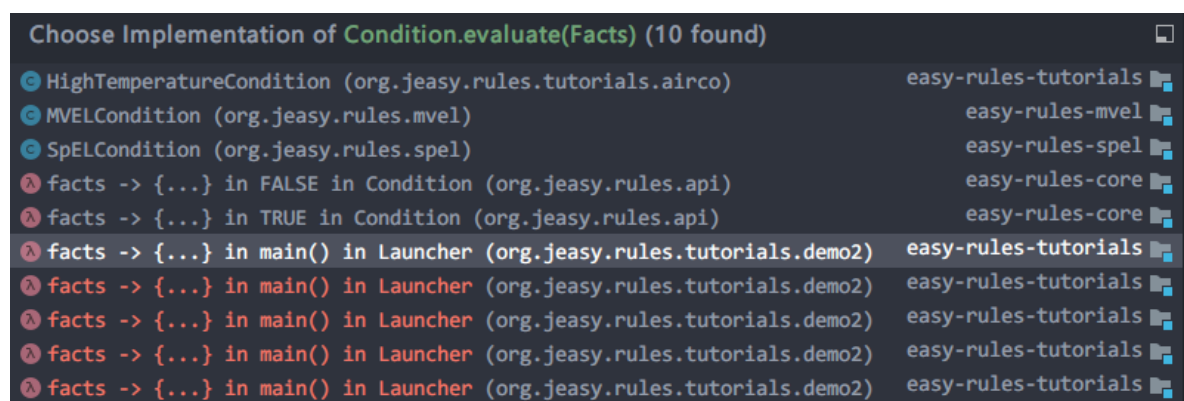
        /**
         * Evaluate the condition according to the known facts.
         *
         * @param facts known when evaluating the rule.
         *
         * @return true if the rule should be triggered, false otherwise
         */
        boolean evaluate(Facts facts);

        /**
         * A NoOp {@link Condition} that always returns false.
         */
        Condition FALSE = facts -> false;

        /**
         * A NoOp {@link Condition} that always returns true.
         */
        Condition TRUE = facts -> true;
    }

```

继续看evaluate还有什么实现，其实就是我们自己用lambda实现的条件，当中返回了boolean类型的值。



```

.when(facts -> facts.get("rainy").equals("yes"))

```

回到引擎中，我们知道了这里就是对规则的评估就是判断condition是否符合事实，是则返回true，则会执行下面这段代码。

先看比较核心的execute，

后面还有几个listener和skip的方法待会儿再看。

```

if (evaluationResult) {
    LOGGER.debug("规则 '{}' 触发", name);
    triggerListenersAfterEvaluate(rule, facts, evaluationResult: true);
    try {
        triggerListenersBeforeExecute(rule, facts);
        rule.execute(facts);
        LOGGER.debug("规则 '{}' 执行成功", name);
        triggerListenersOnSuccess(rule, facts);
        if (parameters.isSkipOnFirstAppliedRule()) {
            LOGGER.debug("Next rules will be skipped since parameter skipOnFirstAppliedRule is set");
            break;
        }
    } catch (Exception exception) {
        LOGGER.error("Rule '" + name + "' performed with error", exception);
        triggerListenersOnFailure(rule, exception, facts);
        if (parameters.isSkipOnFirstFailedRule()) {
            LOGGER.debug("Next rules will be skipped since parameter skipOnFirstFailedRule is set");
            break;
        }
    }
}
}

```

同样我们去看DefaultRule的实现。这里会发现和evaluate的做法几乎是一样的。

```

rule.execute(facts);
LOGGER.de
triggerLi
if (param
    LOGGE
    break
}
} catch (Exce
    LOGGE.er
    triggerLi

```

Choose Implementation of Rule.execute(Facts) (9 found)

- ActivationRuleGroup (org.jeasy.rules.support.composite)
- BasicRule (org.jeasy.rules.core)
- CompositeRule (org.jeasy.rules.support.composite)
- ConditionalRuleGroup (org.jeasy.rules.support.composite)
- DefaultRule (org.jeasy.rules.core)**
- MVELRule (org.jeasy.rules.mvel)
- SpELRule (org.jeasy.rules.spel)
- TestRule in ConditionalRuleGroupTest (org.jeasy.rules.support.com)
- UnitRuleGroup (org.jeasy.rules.support.composite)

```

class DefaultRule extends BasicRule {

    private final Condition condition;
    private final List<Action> actions;

    DefaultRule(String name, String description, int priority, Condition condition) {
        super(name, description, priority);
        this.condition = condition;
        this.actions = actions;
    }

    @Override
    public boolean evaluate(Facts facts) {
        return condition.evaluate(facts);
    }

    @Override
    public void execute(Facts facts) throws Exception {
        for (Action action : actions) {
            action.execute(facts);
        }
    }
}

```

Choose Implementation of Action.execute(Facts) (8 found)

DecreaseTemperatureAction (org.jeasy.rules.tutorials.airco)	easy-rules-tutorials
MVELAction (org.jeasy.rules.mvel)	easy-rules-mvel
SpELAction (org.jeasy.rules.spel)	easy-rules-spel
facts -> {...} in main() in Launcher (org.jeasy.rules.tutorials.demo2)	easy-rules-tutorials
facts -> {...} in main() in Launcher (org.jeasy.rules.tutorials.demo2)	easy-rules-tutorials
facts -> {...} in main() in Launcher (org.jeasy.rules.tutorials.demo2)	easy-rules-tutorials
facts -> {...} in main() in Launcher (org.jeasy.rules.tutorials.demo2)	easy-rules-tutorials
facts -> {...} in main() in Launcher (org.jeasy.rules.tutorials.demo2)	easy-rules-tutorials

其他实现好像和yml方式有关。红色的就是我们自己的lambda实现了。那再看看这舒适的流式编程api底层究竟怎么做的？

```

.when(facts -> facts.get("rainy").equals("yes"))
.then(facts -> System.out.println("it rains ..."))

```

```

public RuleBuilder then(Action action) {
    this.actions.add(action);
    return this;
}

```

啊..好简单啊.其实核心我们已经看完了。Easy Rules表里如一都很简单，简单又而不简陋。

我们再回来看刚刚的listener，

它就类似于一些回调函数，在规则评估前后触发，在规则执行前后触发。类似的设计很多，就比如前端的回调函数在初入前端坑时常常困扰着我，这种监听设计也在spring这种框架中提供了优秀的可扩展性。

对RuleListener一探究竟。这个接口就定义了一些生命周期方法。

```
/**
 *
 */
public interface RuleListener {

    /**
     * Triggered before the evaluation of a rule.
     *
     * @param rule being evaluated
     * @param facts known before evaluating the rule
     * @return true if the rule should be evaluated, false otherwise
     */
    default boolean beforeEvaluate(Rule rule, Facts facts) {
        return true;
    }

    /**
     * Triggered after the evaluation of a rule.
     *
     * @param rule that has been evaluated
     * @param facts known after evaluating the rule
     * @param evaluationResult true if the rule evaluated to true, false otherwise
     */
    default void afterEvaluate(Rule rule, Facts facts, boolean evaluationResult) { }

    /**
     * Triggered on condition evaluation error due to any runtime exception.
     *
     * @param rule that has been evaluated
     * @param facts known while evaluating the rule
     * @param exception that happened while attempting to evaluate the condition.
     */
    default void onEvaluationError(Rule rule, Facts facts, Exception exception) { }

    /**
     * Triggered before the execution of a rule.
     *
     * @param rule the current rule
     * @param facts known facts before executing the rule
     */
}
```

规则引擎抽象类中存放着监听器的集合，嗅觉灵敏的朋友应该已经感知到了观察者模式的香气。

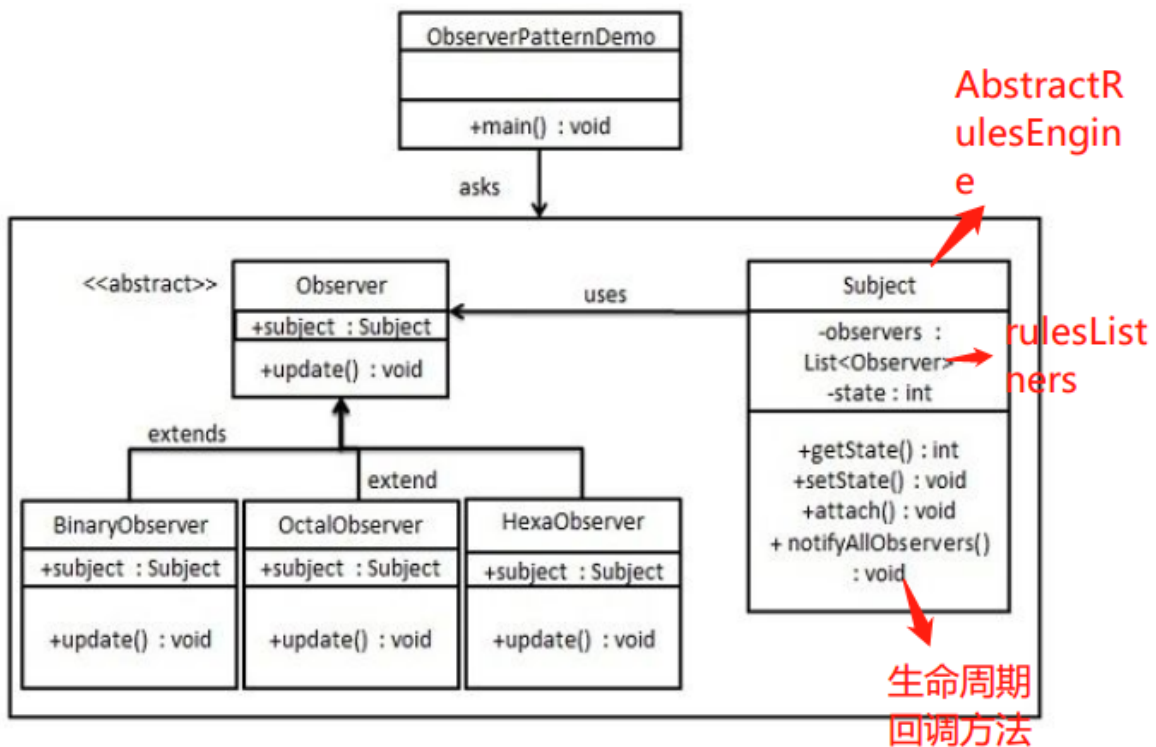
```
public abstract class AbstractRulesEngine implements RulesEngine {

    RulesEngineParameters parameters;
    List<RuleListener> ruleListeners;
    List<RulesEngineListener> rulesEngineListeners;

    AbstractRulesEngine() { this(new RulesEngineParameters()); }

    AbstractRulesEngine(final RulesEngineParameters parameters) {
        this.parameters = parameters;
        this.ruleListeners = new ArrayList<>();
        this.rulesEngineListeners = new ArrayList<>();
    }
}
```

拿菜鸟教程观察者模式示意图和规则引擎的结构比对一下。Observer(对应listener)中存的就是生命周期的方法。



监听器在add方法或构造方法中被设入，是用户的自定义的实现，重写生命周期的方法。


```
List<RuleListener> ruleListeners;
List<RulesEngineListener>

Usages of ruleListeners in All Places

AbstractRulesEngine() { t
    AbstractRulesEngine.java 52 → this.ruleListeners = new ArrayList<>();
    AbstractRulesEngine.java 76 → return Collections.unmodifiableList(ruleListeners);
    AbstractRulesEngine.java 89 → ruleListeners.add(ruleListener);
    AbstractRulesEngine.java 93 → this.ruleListeners.addAll(ruleListeners);
    DefaultRulesEngine.java 177 → ruleListeners.forEach(ruleListener -> ruleListener.onFailure(rule, facts, exception));
    DefaultRulesEngine.java 181 → ruleListeners.forEach(ruleListener -> ruleListener.onSuccess(rule, facts));
    DefaultRulesEngine.java 185 → ruleListeners.forEach(ruleListener -> ruleListener.beforeExecute(rule, facts));
    DefaultRulesEngine.java 189 → return ruleListeners.stream().allMatch(ruleListener -> ruleListener.beforeEvaluate(rule, fa
    DefaultRulesEngine.java 193 → ruleListeners.forEach(ruleListener -> ruleListener.afterEvaluate(rule, facts, evaluationResu
    DefaultRulesEngine.java 197 → ruleListeners.forEach(ruleListener -> ruleListener.onEvaluationError(rule, facts, exception)

AbstractRulesEngine(final
    this.parameters = par
    this.ruleListeners =
    this.rulesEngineListe
}
```

在DefaultRulesEngine中定义了监听器触发方法，对应notifyAllObserver。它遍历listener，调用被实现的生命周期方法。

```
private void triggerListenersOnFailure(final Rule rule, final Exception exception, Facts facts) {
    ruleListeners.forEach(ruleListener -> ruleListener.onFailure(rule, facts, exception));
}

private void triggerListenersOnSuccess(final Rule rule, Facts facts) {
    ruleListeners.forEach(ruleListener -> ruleListener.onSuccess(rule, facts));
}

private void triggerListenersBeforeExecute(final Rule rule, Facts facts) {
    ruleListeners.forEach(ruleListener -> ruleListener.beforeExecute(rule, facts));
}

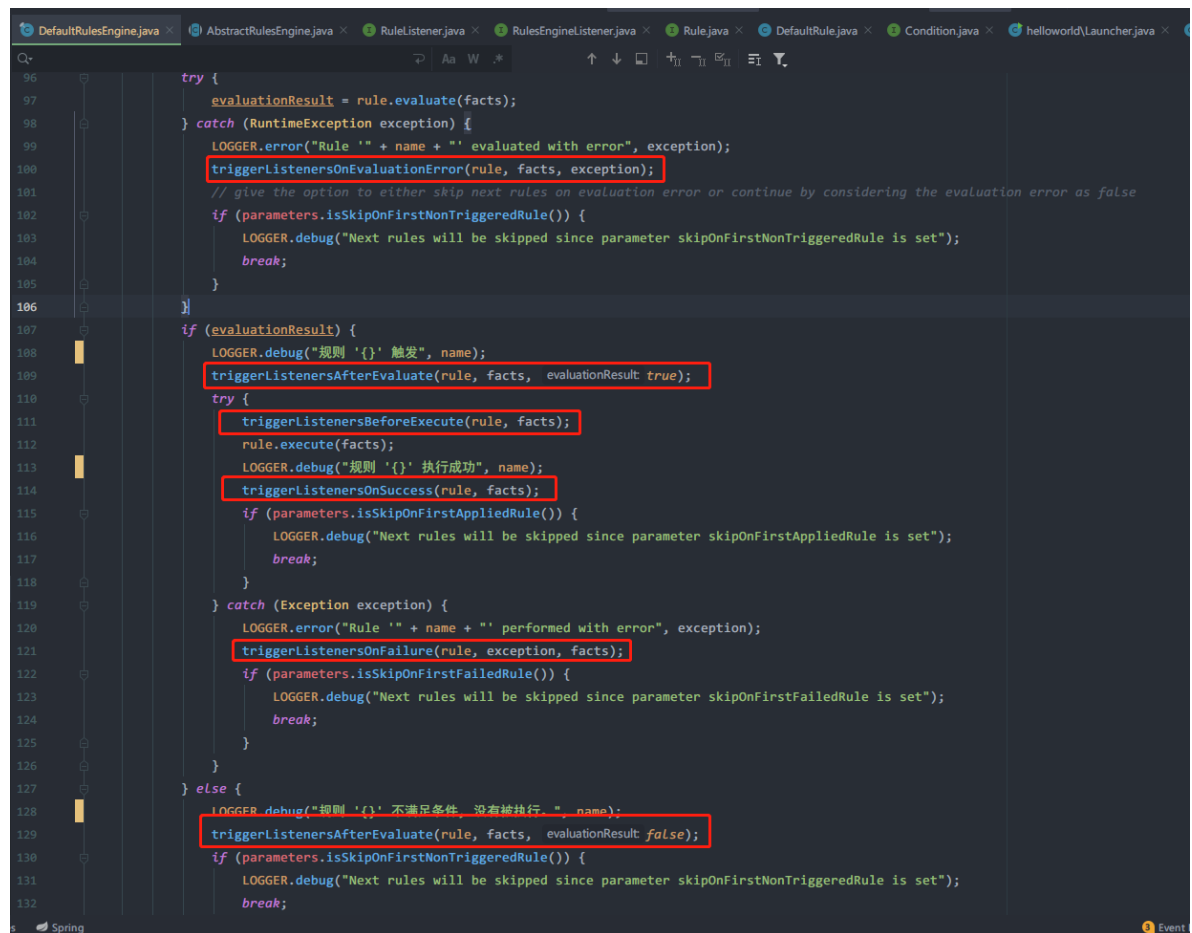
private boolean triggerListenersBeforeEvaluate(Rule rule, Facts facts) {
    return ruleListeners.stream().allMatch(ruleListener -> ruleListener.beforeEvaluate(rule, facts));
}

private void triggerListenersAfterEvaluate(Rule rule, Facts facts, boolean evaluationResult) {
    ruleListeners.forEach(ruleListener -> ruleListener.afterEvaluate(rule, facts, evaluationResult));
}

private void triggerListenersOnEvaluationError(Rule rule, Facts facts, Exception exception) {
    ruleListeners.forEach(ruleListener -> ruleListener.onEvaluationError(rule, facts, exception));
}

private void triggerListenersBeforeRules(Rules rule, Facts facts) {
    rulesEngineListeners.forEach(rulesEngineListener -> rulesEngineListener.beforeEvaluate(rule, facts));
}
```

在对应的时机前后调用他们。



```
96     try {
97         evaluationResult = rule.evaluate(facts);
98     } catch (RuntimeException exception) {
99         LOGGER.error("Rule '" + name + "' evaluated with error", exception);
100         triggerListenersOnEvaluationError(rule, facts, exception);
101         // give the option to either skip next rules on evaluation error or continue by considering the evaluation error as false
102         if (parameters.isSkipOnFirstNonTriggeredRule()) {
103             LOGGER.debug("Next rules will be skipped since parameter skipOnFirstNonTriggeredRule is set");
104             break;
105         }
106     }
107     if (evaluationResult) {
108         LOGGER.debug("规则 '{} 触发", name);
109         triggerListenersAfterEvaluate(rule, facts, evaluationResult true);
110         try {
111             triggerListenersBeforeExecute(rule, facts);
112             rule.execute(facts);
113             LOGGER.debug("规则 '{}' 执行成功", name);
114             triggerListenersOnSuccess(rule, facts);
115             if (parameters.isSkipOnFirstAppliedRule()) {
116                 LOGGER.debug("Next rules will be skipped since parameter skipOnFirstAppliedRule is set");
117                 break;
118             }
119         } catch (Exception exception) {
120             LOGGER.error("Rule '" + name + "' performed with error", exception);
121             triggerListenersOnFailure(rule, exception, facts);
122             if (parameters.isSkipOnFirstFailedRule()) {
123                 LOGGER.debug("Next rules will be skipped since parameter skipOnFirstFailedRule is set");
124                 break;
125             }
126         }
127     } else {
128         LOGGER.debug("规则 '{}' 不满足条件, 没有被执行.", name);
129         triggerListenersAfterEvaluate(rule, facts, evaluationResult false);
130         if (parameters.isSkipOnFirstNonTriggeredRule()) {
131             LOGGER.debug("Next rules will be skipped since parameter skipOnFirstNonTriggeredRule is set");
132             break;
133         }
134     }
135 }
```

其实高大上的东西剖析完，发现你和优秀开源框架设计者最大的差距就是，第一是差在抽象问题的能力和对设计模式熟练的应用，第二是差在会不会起高大上的名字... 就算是spring的生命周期，也应该是这样做的，只是会包含更多复杂逻辑。

接下来看看skip方法。

在定义规则引擎中介绍了规则引擎的参数，就有几个skip.忘了可以回去看看他们是什么意思，然后我们看看它在源码中的实现。skip返回的是布尔值，我们初始规则引擎时设置了true就会执行，中间debug了一条日志，然后break了。easy..

```
boolean evaluationResult = false;
try {
    evaluationResult = rule.evaluate(facts);
} catch (RuntimeException exception) {
    LOGGER.error("Rule '" + name + "' evaluated with error", exception);
    triggerListenersOnEvaluationError(rule, facts, exception);
    // give the option to either skip next rules on evaluation error or continue by considering the evaluation
    if (parameters.isSkipOnFirstNonTriggeredRule()) {
        LOGGER.debug("Next rules will be skipped since parameter skipOnFirstNonTriggeredRule is set");
        break;
    }
}
if (evaluationResult) {
    LOGGER.debug("规则 '{}' 触发", name);
    triggerListenersAfterEvaluate(rule, facts, evaluationResult: true);
    try {
        triggerListenersBeforeExecute(rule, facts);
        rule.execute(facts);
        LOGGER.debug("规则 '{}' 执行成功", name);
        triggerListenersOnSuccess(rule, facts);
        if (parameters.isSkipOnFirstAppliedRule()) {
            LOGGER.debug("Next rules will be skipped since parameter skipOnFirstAppliedRule is set");
            break;
        }
    } catch (Exception exception) {
        LOGGER.error("Rule '" + name + "' performed with error", exception);
        triggerListenersOnFailure(rule, exception, facts);
        if (parameters.isSkipOnFirstFailedRule()) {
            LOGGER.debug("Next rules will be skipped since parameter skipOnFirstFailedRule is set");
            break;
        }
    }
} else {
    LOGGER.debug("规则 '{}' 不满足条件，没有被执行。", name);
    triggerListenersAfterEvaluate(rule, facts, evaluationResult: false);
    if (parameters.isSkipOnFirstNonTriggeredRule()) {
        LOGGER.debug("Next rules will be skipped since parameter skipOnFirstNonTriggeredRule is set");
    }
}
```

最后一个比较有趣的点，注册rule的时候，这里循环的是Object实例，add方法参数是Rule实例。

这里又使用了代理模式，

将Object类代理成Rule类。

```
public void register(Object... rules) {
    Objects.requireNonNull(rules);
    for (Object rule : rules) {
        Objects.requireNonNull(rule);
        this.rules.add(RuleProxy.asRule(rule));
    }
}
```

看看RuleProxy中的asRule方法，RuleProxy实现了InvocationHandler接口。

注释：参数是加了Rule注解的对象，让返回的rule对象实现Rule接口，参数中也看到了实现Comparable接口。

含义是被注册的rule对象生成了实现Rule、Comparable的代理对象，调用rule对象方法时会被转发到ruleProxy对象的invoke方法中。

```
/**
 * Makes the rule object implement the {@link Rule} interface.
 *
 * @param rule the annotated rule object.
 * @return a proxy that implements the {@link Rule} interface.
 */
public static Rule asRule(final Object rule) {
    Rule result;
    if (rule instanceof Rule) {
        result = (Rule) rule;
    } else {
        ruleDefinitionValidator.validateRuleDefinition(rule);
        result = (Rule) Proxy.newProxyInstance(
            Rule.class.getClassLoader(),
            new Class[]{Rule.class, Comparable.class},
            new RuleProxy(rule));
    }
    return result;
}
```

去看看proxy对象的invoke方法。参数proxy就是代理对象，method是目标对象被调用的方法。这里反射获取都没用，switch简单粗暴。

```
@Override
public Object invoke(final Object proxy, final Method method, final Object[] args) throws Throwable {
    String methodName = method.getName();
    switch (methodName) {
        case "getName":
            return getRuleName();
        case "getDescription":
            return getRuleDescription();
        case "getPriority":
            return getRulePriority();
        case "compareTo":
            return compareToMethod(args);
        case "evaluate":
            return evaluateMethod(args);
        case "execute":
            return executeMethod(args);
        case "equals":
            return equalsMethod(args);
        case "hashCode":
            return hashCodeMethod();
        case "toString":
            return toStringMethod();
        default:
            return null;
    }
}
```

拿getRulePriority()看一下。逻辑就是先设置默认优先级，如果类注解上有priority参数就设置，如果有@Priority注解的方法就覆盖priority，返回。

```
private int getRulePriority() throws Exception {
    if (this.priority == null) {
        int priority = Rule.DEFAULT_PRIORITY;

        org.jeasy.rules.annotation.Rule rule = getRuleAnnotation();
        if (rule.priority() != Rule.DEFAULT_PRIORITY) {
            priority = rule.priority();
        }

        Method[] methods = getMethods();
        for (Method method : methods) {
            if (method.isAnnotationPresent(Priority.class)) {
                priority = (int) method.invoke(target);
                break;
            }
        }
        this.priority = priority;
    }
    return this.priority;
}
```

```
@Rule(name = "Hello World rule", description = "Always say hello world", priority = 1)
public class HelloWorldRule {
```

```
priority default org.jeasy.rules.api.Rule.DEFAULT_PRIORITY
Ctrl+向下箭头 and Ctrl+向上箭头 will move caret down and up in the editor Next Tip
```

再看看compareToMethod，首先getCompareToMethod，从目标对象中获取名为"compareTo"的方法，有就调用自定义的方法，没有就执行默认的compareTo方法。args[0]是参数，是需要进行对比的另一个rule对象。

```
private Object compareToMethod(final Object[] args) throws Exception {
    Method compareToMethod = getCompareToMethod();
    Object otherRule = args[0]; // validated upfront
    if (compareToMethod != null && Proxy.isProxyClass(otherRule.getClass())) {
        if (compareToMethod.getParameters().length != 1) {
            throw new IllegalArgumentException("compareTo method must have a single argument");
        }
        RuleProxy ruleProxy = (RuleProxy) Proxy.getInvocationHandler(otherRule);
        return compareToMethod.invoke(target, ruleProxy.getTarget());
    } else {
        return compareTo((Rule) otherRule);
    }
}
```

```
private Method getCompareToMethod() {
    if (this.compareToMethod == null) {
        Method[] methods = getMethods();
        for (Method method : methods) {
            if (method.getName().equals("compareTo")) {
                this.compareToMethod = method;
                return this.compareToMethod;
            }
        }
    }
    return this.compareToMethod;
}
```

查看默认的compareTo方法，通过对比他们的priority优先级，如果优先级相等 就对比名字，如果当前对象优先级小于otherRule的优先级，会返回-1。

```
private int compareTo(final Rule otherRule) throws Exception {  
    int otherPriority = otherRule.getPriority();  
    int priority = getRulePriority();  
    if (priority < otherPriority) {  
        return -1;  
    } else if (priority > otherPriority) {  
        return 1;  
    } else {  
        String otherName = otherRule.getName();  
        String name = getRuleName();  
        return name.compareTo(otherName);  
    }  
}
```

Rule之所以要实现Comparable接口重写compareTo的意义是Rule会被存在Rules的TreeSet中，我们知道add入TreeSet的元素是自然排序的，Rule若不实现Comparable接口，而把对象往TreeSet中添加，就会报ClassCastException。不能完成排序。

到这里Easy Rules的原理已经被剖析完了，代码风格简单干净又巧妙。能学到不少东西。

感谢阅读。
