从 Drools 规则引擎到风控反洗钱

——《Drools7.0.0.Final 规则引擎教程》

修订日期	变更版本	变更描述	修订方法	修订人
2017/7/05	V0.1	创建、新增《Drools 简介》	创建	朱智胜
2017/7/06	V0.1	追溯 Drools5 的使用	新增	朱智胜
2017/7/07	V0.1	Hello World 实例	新增	朱智胜
2017/7/11	V0.1	KIE&FACT	新增	朱智胜
2017/7/12	V0.1	KIE API 解析	新增	朱智胜
2017/7/15	V0.2.1	规则文件	新增	朱智胜
2017/7/15	V0.2.1	no-loop&lock-on-active	新增	朱智胜
2017/7/18	V0.2.1	ruleflow-group& salience	新增	朱智胜
2017/7/19	V0.2.1	agenda-group& auto-focus	新增	朱智胜
2017/7/20	V0.2.1	activation-group& dialect& date-effective& date-expires&duration& enabled	新增	朱智胜
2017/7/21	V0.2.2	timer	新增	朱智胜
2017/7/27	V0.2.2	calendar	新增	朱智胜
2017/7/28	V0.2.3	LHS	新增	朱智胜
2017/7/29	V0.2.3	Pattern&约束	新增	朱智胜
2017/8/1	V0.2.3	与 Springboot 集成	新增	朱智胜
2017/8/2	V0.2.3	动态加载规则实例&约束	新增	朱智胜
2017/8/3	V0.2.4	RHS 语法	新增	朱智胜
2017/8/4	V0.2.4	结果条件	新增	朱智胜
2017/8/5	V0.2.4	注释	新增	朱智胜
2017/8/5	V0.2.4	异常&关键字	新增	朱智胜
2017/8/11	V0.3.0	1、相同对象 and List 使用 demo 2、获取规则名称和包名 demo	新增	朱智胜
2017/8/11	V0.3.1	global 全局变量	新增	朱智胜
2017/8/20	V0.3.2	Query 基本语法&实例	新增	朱智胜

目录

1 Drools 简介

1.1 什么是规则引擎

规则引擎是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。

大多数规则引擎都支持规则的次序和规则冲突检验，支持简单脚本语言的规则实现，支持通用开发语言的嵌入开发。目前业内有多个规则引擎可供使用，其中包括商业和开放源码选择。开源的代表是 Drools，商业的代表是 Visual Rules ,I Log。

1.2 Drools 规则引擎

Drools（JBoss Rules ）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，从而检验是否已编码的规则执行了所需的业务规则。

JBoss Rules 的前身是 Codehaus 的一个开源项目叫 Drools。现在被纳入 JBoss 门下，更名为 JBoss Rules，成为了 JBoss 应用服务器的规则引擎。

Drools 是为 Java 量身定制的基于 Charles Forgy 的 RETE 算法的规则引擎的实现。具有了 OO 接口的 RETE,使得商业规则有了更自然的表达。

1.3 Drools 使用概览

Drools 是 Java 编写的一款开源规则引擎，实现了 Rete 算法对所编写的规则求值，支持声明方式表达业务逻辑。使用 DSL(Domain Specific Language)语言来编写业务规则，使得规则通俗易懂，便于学习理解。支持 Java 代码直接嵌入到规则文件中。

Drools 主要分为两个部分：一是 Drools 规则，二是 Drools 规则的解释执行。规则的编译与运行要通过 Drools 提供的相关 API 来实现。而这些 API 总体上游可分为三类：规则编译、规则收集和规则的执行。

Drools 是业务规则管理系统（BRMS）解决方案，涉及以下项目：

-  Drools Workbench：业务规则管理系统
-  Drools Expert：业务规则引擎
-  Drools Fusion：事件处理
-  jBPM：工作流引擎
-  OptaPlanner：规划引擎

1.4 Drools 版本信息

目前 Drools 发布的最新版本为 7.0.0.Final，其他版本正在研发过程中。官方表示后续版本会加快迭代速度。本系列也是基于此版本进行讲解。

从 Drools6.x 到 7 版本发生重大的变化项：

- @PropertyReactive 不需要再配置，在 Drools7 中作为默认配置项。同时向下兼容。
- Drools6 版本中执行 sum 方法计算结果数据类型问题修正。
- 重命名 TimedRuleExecutionOption。
- 重命名和统一配置文件。

Drools7 新功能：

- 支持多线程执行规则引擎，默认为开启，处于试验阶段。
- OOPath 改进，处于试验阶段。
- OOPath Maven 插件支持。
- 事件的软过期。
- 规则单元 RuleUnit。

1.5 JDK 版本及 IDE

从 Drools6.4.0 开始已经支持 JAVA8，最低版本 JDK1.5。可通过 Eclipse 插件进行集成，也可通过 Intellij IDEA 中插件进行集成开发。Drools 提供了一个 Eclipse 的集成版本，不过它核心依赖于 JDK1.5。

关键 Eclipse 的集成官方有详细的文档可参考，这里不再赘述。Drools7.0.0.Final 要求的最低 JDK 版本为 JDK 1.8（核心包采用此版本编译，低于此版本的 JDK 运行时会抛出异常）。本系列后续项目及示例演示均采用 JAVA8 和 Intellij IDEA。

1.6 官方资料

官网地址：http://www.drools.org/
官方最新文档：
https://docs.jboss.org/drools/release/7.0.0.Final/drools-docs/html_single/index.html

2 追溯 Drools5 的使用

2.1 Drools5 简述

上面已经提到 Drools 是通过规则编译、规则收集和规则的执行来实现具体功能的。Drools5 提供了以下主要实现 API：

- KnowledgeBuilder
- KnowledgeBase
- KnowledgePackage
- StatefulKnowledgeSession
- StatelessKnowledgeSession

它们起到了对规则文件进行收集、编译、查错、插入 fact、设置 global、执行规则或规则流等作用。

2.2 Drools5 之 HelloWorld

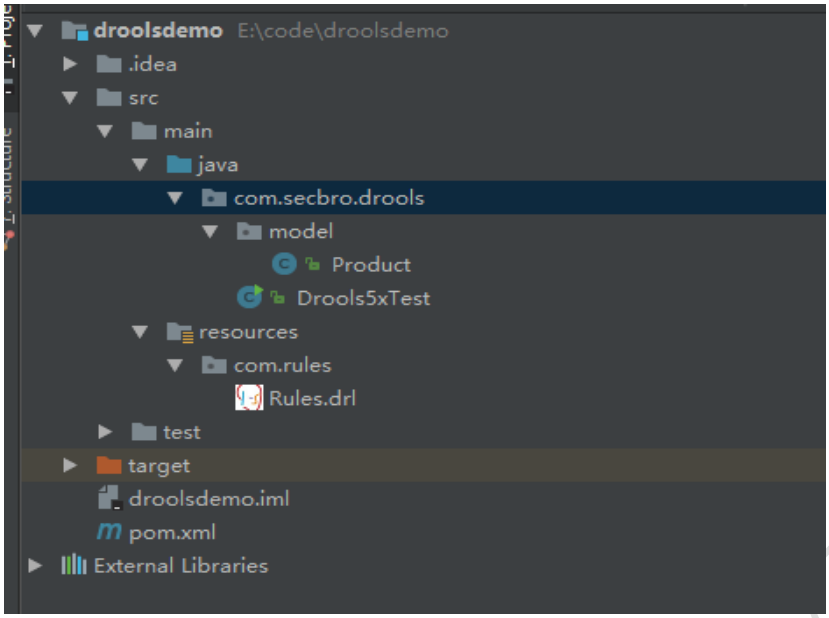
下面结合实例，使用上面的 API 来实现一个简单规则使用实例。随后简单介绍每个 API 的主要作用。Drools7 目前依旧包含上面提的 Drools5 的 API，因此本实例直接使用 Drools7 的 jar 包。

2.2.1 业务场景

目前有两种商品钻石（diamond）和黄金（Gold），需要对这两种商品分别制定销售折扣（discount）。如果使用 Drools 规则引擎就是为了适用两种商品折扣的各种变化，不用修改代码就可以实现复杂业务组合的变更。当然简单的情况，使用普通的 if else 或配置项也可以达到变更的目的，那就不需要 Drools，也就不是本节讨论的范畴了。

2.2.2 代码实例

整体目录结构如下图：



首先创建 JAVA 项目，使用 maven 进行管理。创建之后 maven 的 pom.xml 文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.secbro</groupId>
    <artifactId>drools-demo</artifactId>
    <version>1.0-SNAPSHOT</version>
    <properties>
        <drools-version>7.0.0.Final</drools-version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.drools</groupId>
            <artifactId>drools-compiler</artifactId>
            <version>${drools-version}</version>
        </dependency>
    </dependencies>

</project>
```

创建产品类 Product，如下：

```
package com.secbro.drools.model;
```

```
/**
 * 产品类
 * Created by zhuzs on 2017/7/4.
 */
public class Product {

    public static final String DIAMOND = "DIAMOND"; // 钻石

    public static final String GOLD = "GOLD"; // 黄金

    private String type;

    private int discount;

    // 省略 getter/setter 方法
}
```

在项目的 resources 目录下创建 com/rules 目录，并在创建 Rules.drl，内容如下：

```
package com.rules

import com.secbro.drools.model.Product

rule Offer4Diamond
    when
        productObject : Product(type == Product.DIAMOND)
    then
        productObject.setDiscount(15);
    end
rule Offer4Gold
    when
        productObject: Product(type == Product.GOLD)
    then
        productObject.setDiscount(25);
    end
```

创建执行规则的测试类 Drools5Test：

```
package com.secbro.drools;

import com.secbro.drools.model.Product;
import org.kie.api.io.ResourceType;
import org.kie.internal.KnowledgeBase;
import org.kie.internal.KnowledgeBaseFactory;
import org.kie.internal.builder.KnowledgeBuilder;
import org.kie.internal.builder.KnowledgeBuilderFactory;
import org.kie.internal.definition.KnowledgePackage;
```

```
import org.kie.internal.io.ResourceFactory;
import org.kie.internal.runtime.StatefulKnowledgeSession;

import java.util.Collection;

/**
 * Created by zhuzs on 2017/7/4.
 */
public class Drools5xTest {

    public static void main(String[] args) {
        Drools5xTest test = new Drools5xTest();
        test.oldExecuteDrools();
    }

    private void oldExecuteDrools() {

        KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
        kbuilder.add(ResourceFactory.newClassPathResource("com/rules/Rules.drl",
                this.getClass()), ResourceType.DRL);
        if (kbuilder.hasErrors()) {
            System.out.println(kbuilder.getErrors().toString());
        }

        Collection<KnowledgePackage> pkgs = kbuilder.getKnowledgePackages();
        // add the package to a rulebase
        KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
        //  将 KnowledgePackage 集合添加到 KnowledgeBase 当中
        kbase.addKnowledgePackages(pkgs);

        StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
        Product product = new Product();
        product.setType(Product.GOLD);
        ksession.insert(product);
        ksession.fireAllRules();
        ksession.dispose();

        System.out.println("The discount for the product " + product.getType()
                + " is " + product.getDiscount()+"%");
    }
}
```

现在执行，main 方法，打印出来的结果为：

```
The discount for the product 1 is 25%
```

2.2.3 实例详解

通过上面的实例我们已经完成了 Drools 规则引擎 API 的使用。下面，针对实例逐步讲解每个 API 的使用方法及 drl 文件的语法。

类名	使用说明
KnowledgeBuilder	在业务代码中收集已编写的规则，并对规则文件进行编译，生成编译好的 KnowledgePackage 集合，提供给其他 API 使用。通过其提供的 hasErrors()方法获得编译过程中是否有错，getErrors()方法打印错误信息。支持.drl 文件、.dslr 文件和 xls 文件等。
KnowledgePackage	存放编译之后规则的对象
KnowledgeBase	收集应用当中知识（knowledge）定义的知识库对象（KnowledgePackage），在一个 KnowledgeBase 当中可以包含普通的规则（rule）、规则流(rule flow)、函数定义(function)、用户自定义对象(type model)等，并创建 session 对象(StatefulKnowledgeSession 和 StatelessKnowledgeSession)
StatefulKnowledgeSession	接收外部插入的数据 fact 对象（POJO），将编译好的规则包和业务数据通过 fireAllRules()方法触发所有的规则执行。使用完成需调用 dispose()方法以释放相关内存资源。
StatelessKnowledgeSession	对 StatefulKnowledgeSession 的封装实现，与其对比不需要调用 dispose()方法释放内存，只能插入一次 fact 对象。

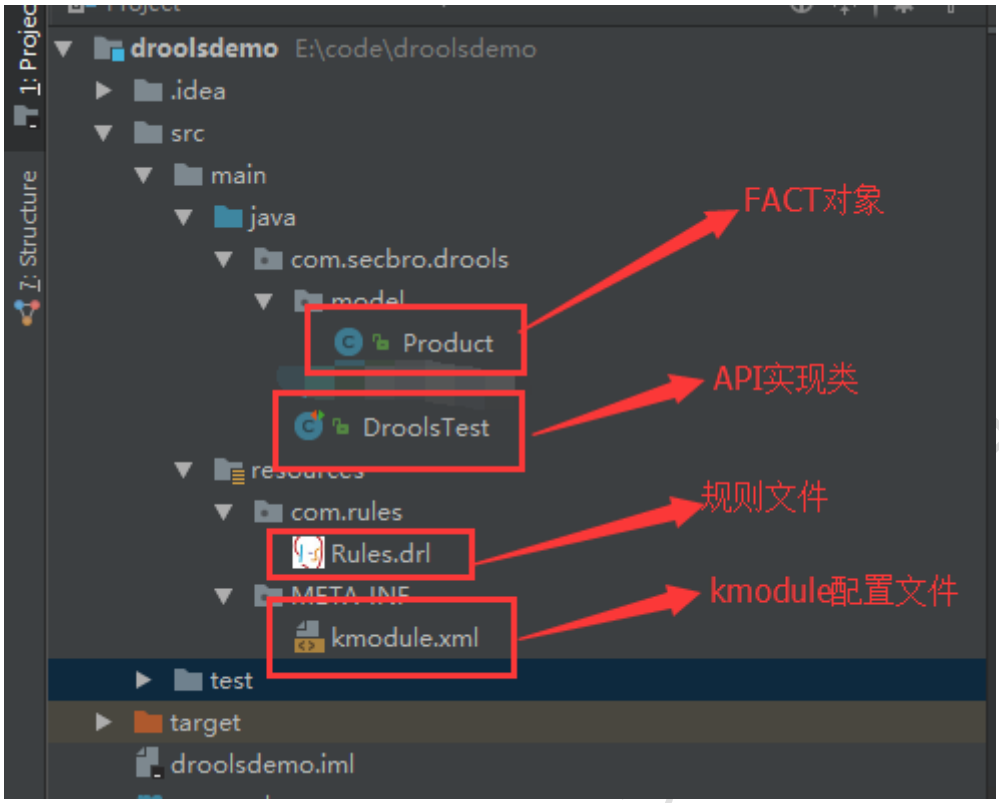
以上是针对 Drools5x 版本 API 相关使用简介，Drools7 版本已经不再使用此系列的 API，此处章节就不展开描述。规则的语法也放在 Drools7 对应章节中进行详细介绍。

3 Drools7 之 HelloWorld

3.1 Hello World 实例

在上一章中介绍了 Drools5x 版本中规则引擎使用的实例，很明显在 Drools7 中 KnowledgeBase 类已被标注为“@Deprecated”——废弃。在本章节中介绍 Drools7 版本中的使用方法。后续实例都将默认使用此版本。

先看一下 Drools 项目的目录结构：



Maven pom.xml 文件中依赖配置：

```
<properties>
        <drools-version>7.0.0.Final</drools-version>
</properties>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
        <dependency>
            <groupId>org.drools</groupId>
            <artifactId>drools-compiler</artifactId>
            <version>${drools-version}</version>
        </dependency>
    </dependencies>
```

Fact 对象对应的实体类依旧为 Product：

```
package com.secbro.drools.model;

/**
 * 产品类
 * Created by zhuzs on 2017/7/4.
 */
```

```
public class Product {

    public static final String DIAMOND = "DIAMOND"; //  钻石

    public static final String GOLD = "GOLD"; //  黄金

    private String type;

    private int discount;
    //  省略 getter/setter 方法
}
```

规则文件依旧为 Rules.drl:

```
package com.rules

import com.secbro.drools.model.Product

rule Offer4Diamond
    when
        productObject : Product(type == Product.DIAMOND)
    then
        productObject.setDiscount(15);
    end
rule Offer4Gold
    when
        productObject: Product(type == Product.GOLD)
    then
        productObject.setDiscount(25);
    end
```

与 Drools5 不同的是 Drools 中引入了 kmodule.xml 文件。配置内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://www.drools.org/xsd/kmodule">
    <kbase name="rules" packages="com.rules">
        <ksession name="ksession-rule"/>
    </kbase>
</kmodule>
```

以下对配置说明进行简单说明，后续会专门针对此配置文件进行详细解释：

- Kmodule 中可以包含一个到多个 kbase，分别对应 drl 的规则文件。
- Kbase 需要一个唯一的 name，可以取任意字符串。
- packages 为 drl 文件所在 resource 目录下的路径。注意区分 drl 文件中的 package 与此处的 package 不一定相同。多个包用逗号分隔。默认情况下会扫描 resources

目录下所有（包含子目录）规则文件。

- kbase 的 default 属性，标示当前 KieBase 是不是默认的，如果是默认的则不用名称就可以查找到该 KieBase，但每个 module 最多只能有一个默认 KieBase。
- kbase 下面可以有一个或多个 ksession，ksession 的 name 属性必须设置，且必须唯一。

实例代码，本实例采用单元测试的方法进行编写：

```
    @Test
    public void testRules() {
        // 构建 KieServices
        KieServices ks = KieServices.Factory.get();
        KieContainer kieContainer = ks.getKieClasspathContainer();
        // 获取 kmodule.xml 中配置中名称为 ksession-rule 的 session，默认为有状态的。
        KieSession kSession = kieContainer.newKieSession("ksession-rule");

        Product product = new Product();
        product.setType(Product.GOLD);

        kSession.insert(product);
        int count = kSession.fireAllRules();
        System.out.println("命中了" + count + "条规则！");
        System.out.println("商品" +product.getType() + "的商品折扣为" + product.getDiscount() + "%。");

    }
```

运行单元测试打印结果为：

```
命中了 1 条规则！
商品 GOLD 的商品折扣为 25%。
```

以上实例首先定义了一个商品，支持 DIAMOND 和 GOLD，并在规则文件中配置了这两种商品的折扣信息。然后传入商品类型为 GLOD 的 FACT 对象，并调用规则引擎，规则引擎执行了 1 条规则，并返回了此商品的折扣。

至此，我们已经完成了一个规则引擎的使用。通过上面的实例我们可以清楚的看到 Drools7 版本与 Drools5 版本之间所使用的 API 是完全两套 API。

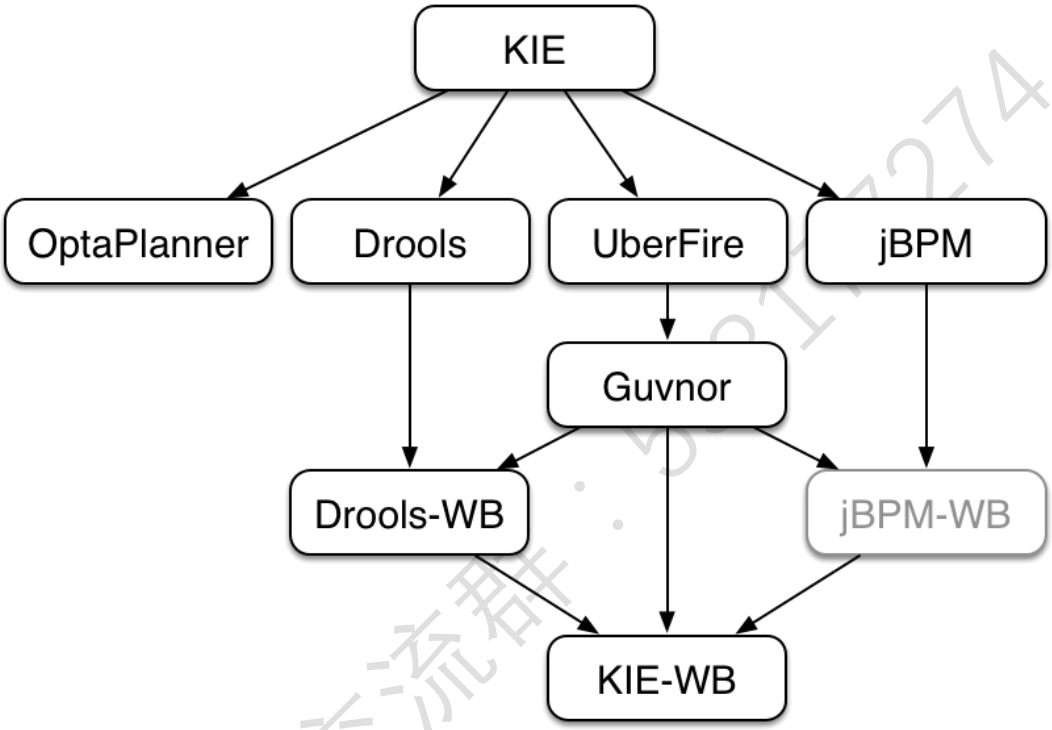
3.2 API 解析

针对上面的实例，我们逐步了解一下使用到的 API 的使用说明及相关概念性知识。

3.2.1 什么是 KIE

KIE（Knowledge Is Everything），知识就是一切的简称。JBoss 一系列项目的总称，在《Drools 使用概述》章节已经介绍了 KIE 包含的大部分项目。它们之间有一定的关联，通用一些 API。比如涉及到构建（building）、部署（deploying）和加载（loading）等方面都会以 KIE 作为前缀来表示这些是通用的 API。

下图为 KIE 所包含的子项目结构图：



3.2.2 KIE 生命周期

无论是 Drools 还是 JBPM，生命周期都包含以下部分：

- 编写：编写规则文件，比如：DRL，BPMN2、决策表、实体类等。
- 构建：构建一个可以发布部署的组件，对于 KIE 来说是 JAR 文件。
- 测试：部署之前对规则进行测试。
- 部署：利用 Maven 仓库将 jar 部署到应用程序。
- 使用：程序加载 jar 文件，通过 KieContainer 对其进行解析创建 KieSession。
- 执行：通过 KieSession 对象的 API 与 Drools 引擎进行交互，执行规则。
- 交互：用户通过命令行或者 UI 与引擎进行交互。
- 管理：管理 KieSession 或者 KieContainer 对象。

3.2.3 FACT 对象

Fact 对象是指在使用 Drools 规则时，将一个普通的 JavaBean 对象插入到规则引擎的 WorkingMemory 当中的对象。规则可以对 Fact 对象进行任意的读写操作。Fact 对象不是对原来的 JavaBean 对象进行 Clone，而是使用传入的 JavaBean 对象的引用。规则在进行计算时需要的应用系统数据设置在 Fact 对象当中，这样规则就可以通过对 Fact 对象数据的读写实现对应用数据的读写操作。

Fact 对象通常是一个具有 getter 和 setter 方法的 POJO 对象，通过 getter 和 setter 方法可以方便的实现对 Fact 对象的读写操作，所以我们可以简单的把 Fact 对象理解为规则与应用系统数据交互的桥梁或通道。

当 Fact 对象插入到 WorkingMemory 当中后，会与当前 WorkingMemory 当中所有的规则进行匹配，同时返回一个 FactHandler 对象。FactHandler 对象是插入到 WorkingMemory 当中 Fact 对象的引用句柄，通过 FactHandler 对象可以实现对 Fact 对象的删除及修改等操作。

前面的实例中通过调用 insert 方法将 Product 对象插入到 WorkingMemory 当中，Product 对象插入到规则中之后就是说为的 FACT 对象。如果需要插入多个 FACT 对象，多次调用 insert 方法，并传入对应 FACT 对象即可。

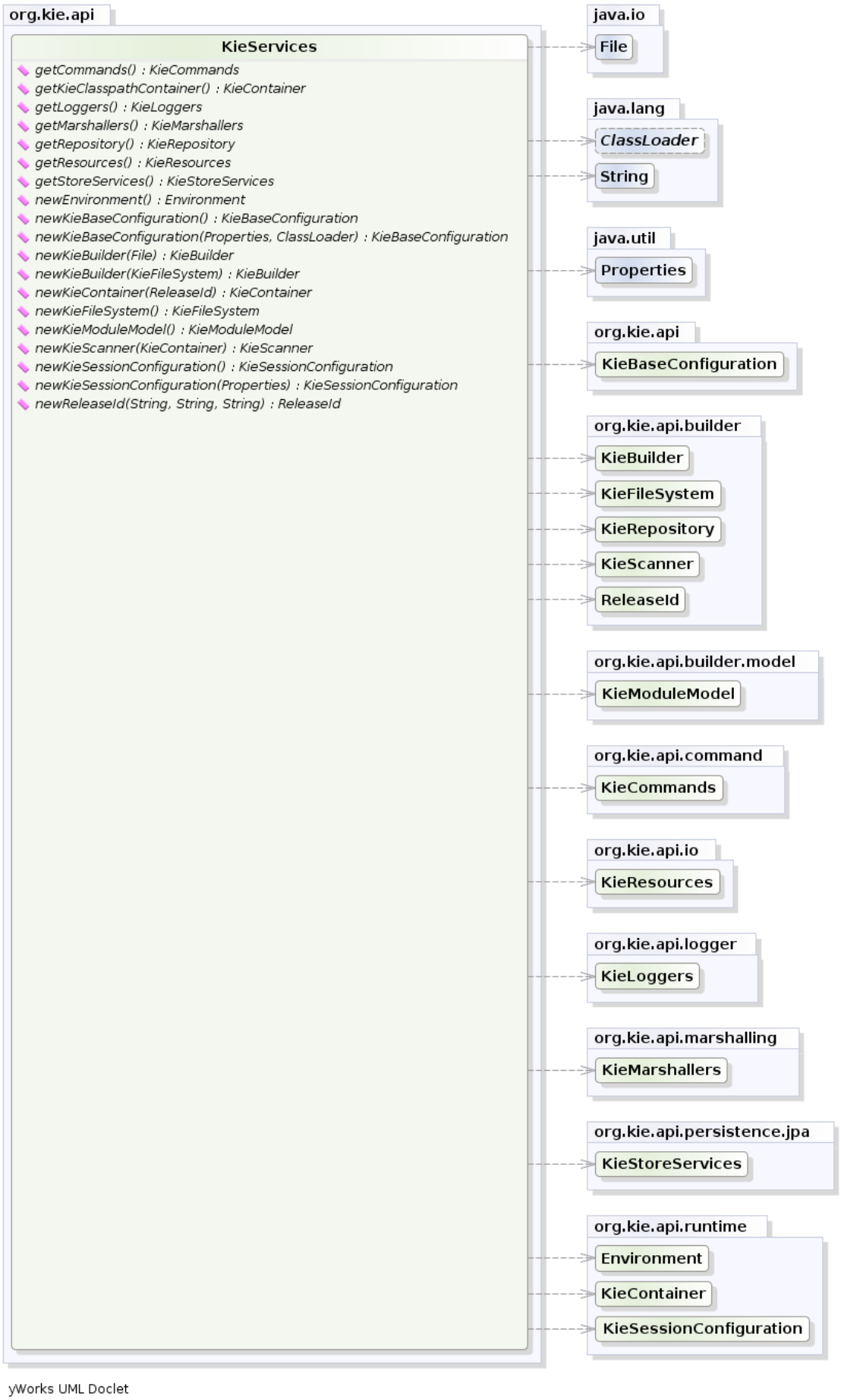
3.2.4 KieServices

该接口提供了很多方法，可以通过这些方法访问 KIE 关于构建和运行的相关对象，比如说可以获取 KieContainer，利用 KieContainer 来访问 KBase 和 KSession 等信息；可以获取 KieRepository 对象，利用 KieRepository 来管理 KieModule 等。

KieServices 就是一个中心，通过它来获取的各种对象来完成规则构建、管理和执行等操作。

示例 demo：

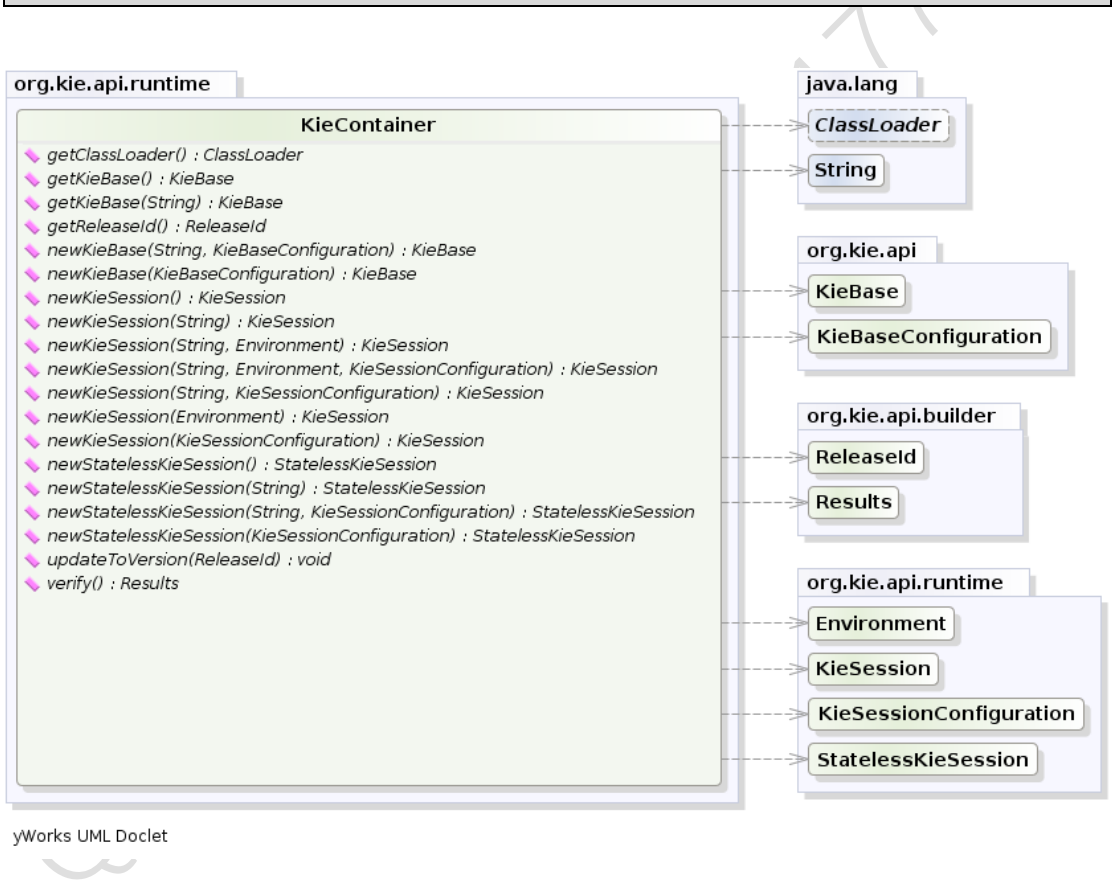
```
// 通过单例创建 KieServices
KieServices kieServices = KieServices.Factory.get();
// 获取 KieContainer
KieContainer kieContainer = kieServices.getKieClasspathContainer();
// 获取 KieRepository
KieRepository kieRepository = kieServices.getRepository();
```



3.2.5 KieContainer

可以理解 KieContainer 就是一个 KieBase 的容器。提供了获取 KieBase 的方法和创建 KieSession 的方法。其中获取 KieSession 的方法内部依旧通过 KieBase 来创建 KieSession。

```
// 通过单例创建 KieServices
KieServices kieServices = KieServices.Factory.get();
// 获取 KieContainer
KieContainer kieContainer = kieServices.getKieClasspathContainer();

// 获取 KieBase
KieBase kieBase = kieContainer.getKieBase();
// 创建 KieSession
KieSession kieSession = kieContainer.newKieSession("session-name");
```

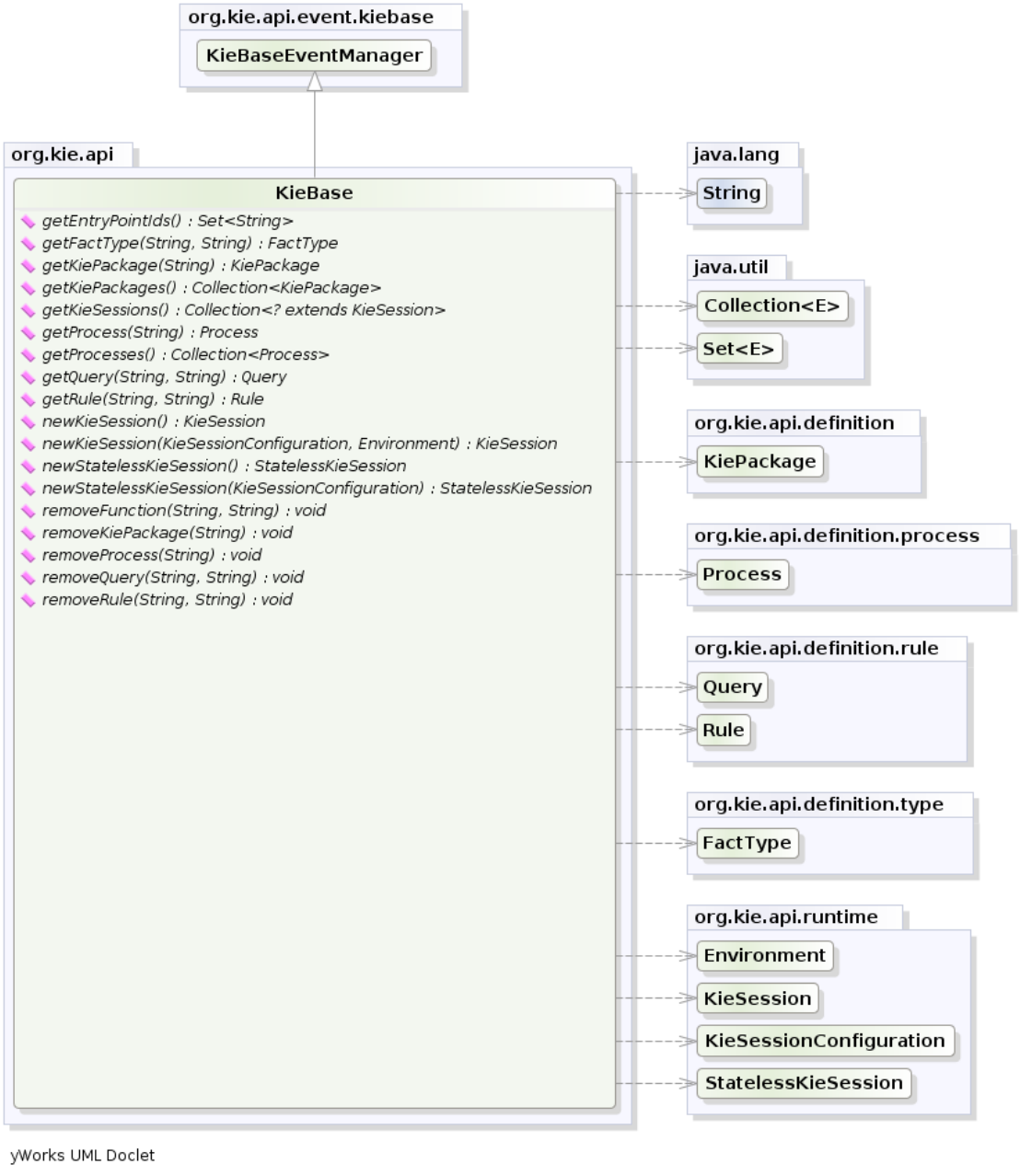


3.2.6 KieBase

KieBase 就是一个知识仓库，包含了若干的规则、流程、方法等，在 Drools 中主要就是规则和方法，KieBase 本身并不包含运行时的数据之类的，如果需要执行规则 KieBase 中的规则的话，就需要根据 KieBase 创建 KieSession。

```
// 获取 KieBase
```

```
KieBase kieBase = kieContainer.getKieBase();
KieSession kieSession = kieBase.newKieSession();
StatelessKieSession statelessKieSession = kieBase.newStatelessKieSession();
```

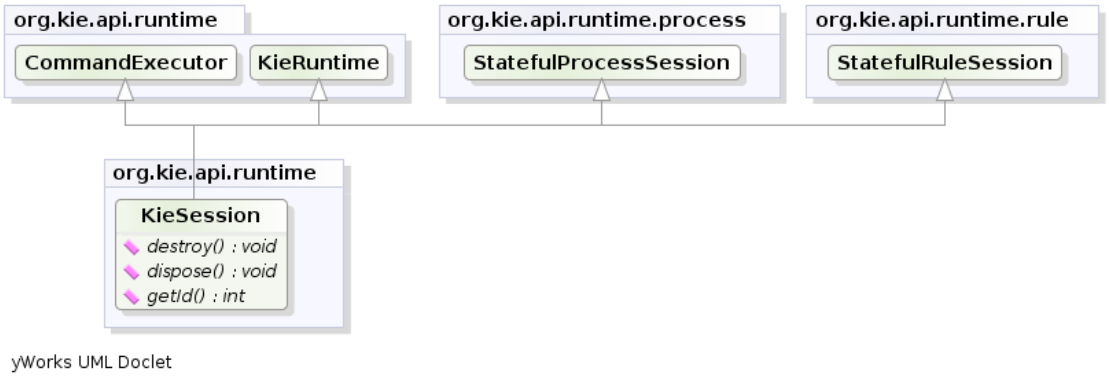


3.2.7 KieSession

KieSession 就是一个跟 Drools 引擎打交道的会话，其基于 KieBase 创建，它会包含运行时数据，包含“事实 Fact”，并对运行时数据实时进行规则运算。通过 KieContainer 创建 KieSession 是一种较为方便的做法，其本质上是从 KieBase 中创建出来的。KieSession 就是应用程序跟规则引擎进行交互的会话通道。

创建 KieBase 是一个成本非常高的事情，KieBase 会建立知识（规则、流程）仓库，而创

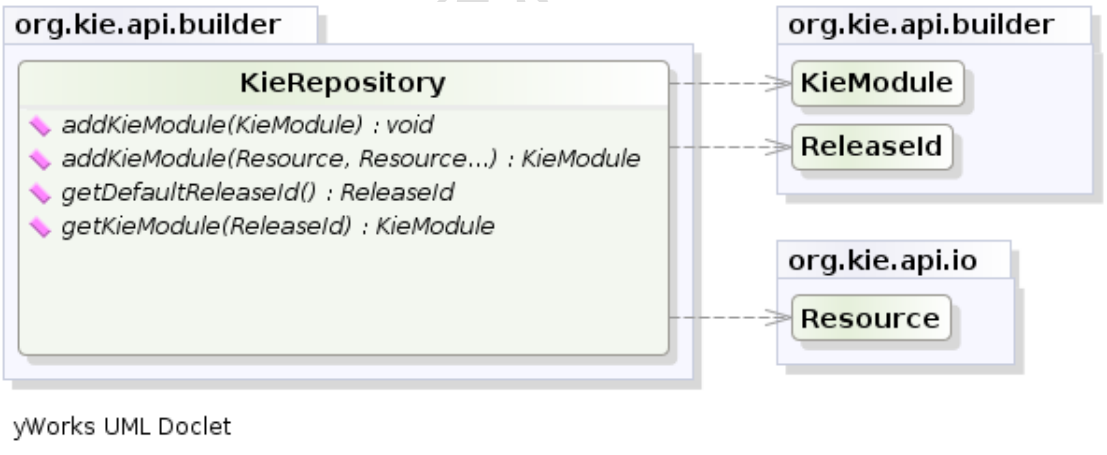
建 KieSession 则是一个成本非常低的事情，所以 KieBase 会建立缓存，而 KieSession 则不必。



3.2.8 KieRepository

KieRepository 是一个单例对象，它是存放 KieModule 的仓库，KieModule 由 kmodule.xml 文件定义（当然不仅仅只是用它来定义）。

```
// 通过单例创建 KieServices
KieServices kieServices = KieServices.Factory.get();
// 获取 KieRepository
KieRepository kieRepository = kieServices.getRepository();
```



3.2.9 KieProject

KieContainer 通过 KieProject 来初始化、构造 KieModule，并将 KieModule 存放到 KieRepository 中，然后 KieContainer 可以通过 KieProject 来查找 KieModule 定义的信息，并根据这些信息构造 KieBase 和 KieSession。

3.2.10 ClasspathKieProject

ClasspathKieProject 实现了 KieProject 接口，它提供了根据类路径中的 META-INF/kmodule.xml 文件构造 KieModule 的能力，是基于 Maven 构造 Drools 组件的基本保障之一。意味着只要按照前面提到过的 Maven 工程结构组织我们的规则文件或流程文件，只用很少的代码完成模型的加载和构建。

3.2.11 kmodule.xml

kmodule 的简单配置规则在上面的实例中已经简单介绍，下面具体介绍具体的配置。

kbase 的属性：

属性名	默认值	合法的值	描述
name	none	any	KieBase 的名称，这个属性是强制的，必须设置。
includes	none	逗号分隔的 KieBase 名称列表	意味着本 KieBase 将会包含所有 include 的 KieBase 的 rule、process 定义制品文件。非强制属性。
packages	all	逗号分隔的字符串列表	默认情况下将包含 resources 目录下面（子目录）的所有规则文件。也可以指定具体目录下面的规则文件，通过逗号可以包含多个目录下的文件。
default	false	true, false	表示当前 KieBase 是不是默认的，如果是默认的话，不用名称就可以查找到该 KieBase，但是每一个 module 最多只能有一个 KieBase。
equalsBehavior	identity	identity,equality	顾名思义就是定义“equals”（等于）的行为，这个 equals 是针对 Fact（事实）的，当插入一个 Fact 到 Working Memory 中的时候，Drools 引擎会检查该 Fact 是否已经存在，如果存在的话就使用已有的 FactHandle，否则就创建新的。而判断 Fact 是否存在的依据通过该属性定义的方式来进行的：设置成 identity，就是判断对象是否存在，可以理解为用==判断，看是否是同一个对象； 如果该属性设置成 equality 的话，就是通过 Fact 对象的 equals 方法来判断。
eventProcessingMode	cloud	cloud, stream	当以云模式编译时，KieBase 将事件视为正常事实，而在流模式下允许对其进行时间推理。

declarativeAgenda	disabled	disabled,enabled	这是一个高级功能开关，打开后规则将可以控制一些规则的执行与否。

ksession 的属性：

属性名	默认值	合法的值	描述
name	none	any	KieSession 的名称，该值必须唯一，也是强制的，必须设置。
type	stateful	stateful, stateless	定义该 session 到底是有状态（stateful）的还是无状态（stateless）的，有状态的 session 可以利用 Working Memory 执行多次，而无状态的则只能执行一次。
default	false	true, false	定义该 session 是否是默认的，如果是默认的话则可以不用通过 session 的 name 来创建 session，在同一个 module 中最多只能有一个默认的 session。
clockType	realtime	realtime,pseudo	定义时钟类型，用在事件处理上面，在复合事件处理上会用到，其中 realtime 表示用的是系统时钟，而 pseudo 则是用在单元测试时模拟用的。
beliefSystem	simple	simple,defeasible, jtms	定义 KieSession 使用的 belief System 的类型。

4 规则

在前面的章节中我们已经实现了一个简单规则引擎的使用。留心的朋友可能已经发现规则引擎的优势所在，那就是将可变的部分剥离出来放置在 DRL 文件中，规则的变化只用修改 DRL 文件中的逻辑即可，而其他相关的业务代码则几乎不用改动。

本章节的重点介绍规则文件的构成、语法函数、执行及各种属性等。

4.1 规则文件

一个标准的规则文件的格式为已“.drl”结尾的文本文件，因此可以通过记事本工具进行编辑。规则放置于规则文件当中，一个规则文件可以放置多条规则。在规则文件当中也可以存放用户自定义的函数、数据对象及自定义查询等相关在规则当中可能会用到的一些对象。

从架构角度来讲，一般将同一业务的规则放置在同一规则文件，也可以根据不同类型处理操作放置在不同规则文件当中。不建议将所有的规则放置与一个规则文件当中。分开放置，当规则变动时不至于影响到不相干的业务。读取构建规则的成本业务会相应减少。

标准规则文件的结构如下：

```
package package-name
```

```
imports

globals

functions

queries

rules
```

package：在一个规则文件当中 package 是必须的，而且必须放置在文件的第一行。package 的名字是随意的，不必必须对应物理路径，这里跟 java 的 package 的概念不同，只是逻辑上的区分，但建议与文件路径一致。同一的 package 下定义的 function 和 query 等可以直接使用。

比如，上面实例中 package 的定义：

```
package com.rules
```

import：导入规则文件需要的外部变量，使用方法跟 java 相同。像 java 的是 import 一样，还可以导入类中的某一个可访问的静态方法。（特别注意的是，某些教程中提示 import 引入静态方法是不同于 java 的一方面，可能是作者没有用过 java 的静态方法引入。）另外，目前针对 Drools7 版本，static 和 function 关键字的效果是一样的。

```
import static com.secbro.drools.utils.DroolsStringUtils.isEmpty;
import function com.secbro.drools.utils.DroolsStringUtils.isEmpty;
```

rules：定义一个条规则。rule “ruleName”。一条规则包含三部分：属性部分、条件部分和结果部分。rule 规则以 rule 开头，以 end 结尾。

属性部分：定义当前规则执行的一些属性等，比如是否可被重复执行、过期时间、生效时间等。

条件部分，简称 LHS，即 Left Hand Side。定义当前规则的条件，处于 when 和 then 之间。如 when Message();判断当前 workingMemory 中是否存在 Message 对象。LHS 中，可包含 0~n 个条件，如果没有条件，默认为 eval(true)，也就是始终返回 true。条件又称之为 pattern（匹配模式），多个 pattern 之间用可以使用 and 或 or 来进行连接，同时还可以使用小括号来确定 pattern 的优先级。

结果部分，简称 RHS，即 Right Hand Side，处于 then 和 end 之间，用于处理满足条件之后的业务逻辑。可以使用 LHS 部分定义的变量名、设置的全局变量、或者是直接编写 Java 代码。

RHS 部分可以直接编写 Java 代码，但不建议在代码当中有条件判断，如果需要条件判断，那么需要重新考虑将其放在 LHS 部分，否则就违背了使用规则的初衷。

RHS 部分，提供了一些对当前 Working Memory 实现快速操作的宏函数或对象，比如 insert/insertLogical、update/modify 和 retract 等。利用这些函数可以实现对当前 Working Memory 中的 Fact 对象进行新增、修改或删除操作 ;如果还要使用 Drools 提供的其它方法，可以使用另一个宏对象 drools，通过该对象可以使用更多的方法；同时 Drools 还提供了一个名为 kcontext 的宏对象，可以通过该对象直接访问当前 Working Memory 的 KnowledgeRuntime。

标准规则的结构示例：

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

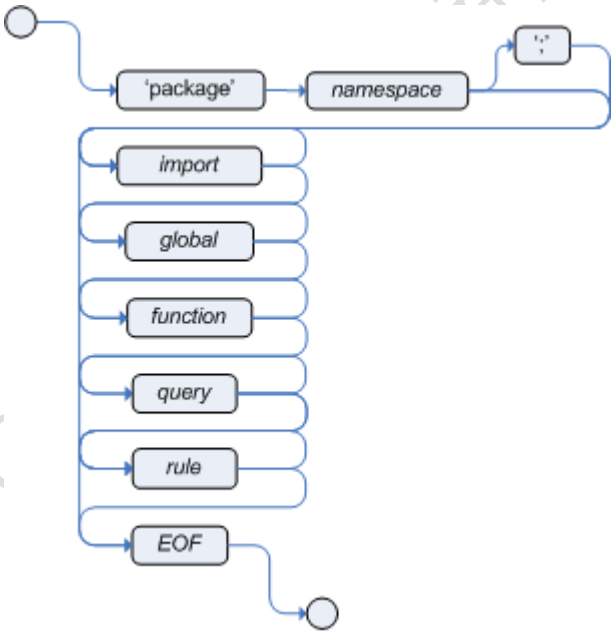
LHS 为空示例:

```
rule "name"
        when
        then
        end
```

4.2 包（package）

在 Drools 中，包的概念与 java 基本相同，区别的地方是 Drools 的中包不一定与物理路径相对应。包一组规则和相关语法结构，比如引入和全局变量。在同一包下的规则相互之间有一定的关联。

一个包代表一个命名空间，一般来讲针对同一组规则要确保包的唯一性。包名本身是命名空间，并且不以任何方式关联到文件或文件夹。



4.2.1import

Drools 的导入语句的作用和在 java 一样。需要为在规则中使用的对象指定完整路径和类型名字。 Drools 会自动从同名的 java 包中导入类，默认会引入 java.lang 下面的类。

4.2.2global 全局变量

global 用来定义全局变量，它可以让应用程序的对象在规则文件中能够被访问。通常，可以用来为规则文件提供数据或服务。特别是用来操作规则执行结果的处理和从规则返回数据，比如执行结果的日志或值，或者与应用程序进行交互的规则的回调处理。

全局变量并不会被插入到 Working Memory 中，因此，除非作为常量值，否则不应该将全局变量用于规则约束的判断中。对规则引擎中的 fact 修改，规则引擎根据算法会动态更新决策树，重新激活某些规则的执行，而全局变量不会对规则引擎的决策树有任何影响。在约束条件中错误的使用全局变量会导致意想不到的错误。

如果多个包中声明具有相同标识符的全局变量，则必须是相同的类型，并且它们都将引用相同的全局值。

实例代码如下：

规则文件内容：

```
package com.rules
import com.secbro.drools.model.Risk
import com.secbro.drools.model.Message

global com.secbro.drools.EmailService
emailService

rule "test-global"

agenda-group "test-global-group"

when
then
    Message message = new Message();
    message.setRule(drools.getRule().getName());
    message.setDesc("to send email!");
    emailService.sendEmail(message);
end
```

测试代码：

```
@Test
public void testGlobal(){
    KieSession kieSession =
this.getKieSession("test-global-group");
    int count = kieSession.fireAllRules();
    kieSession.dispose();
    System.out.println("Fire " + count  + "
```

```
rule(s)!");
}
```

实体类：

```
public class Message {

    private String rule;

    private String desc;
    // getter/setter
    }
```

操作类：

```
public class EmailService {

    public static void sendEmail(Message message){
        System.out.println("Send message to
email,the fired rule is '" + message.getRule()
                + "', and description  is '" +
message.getDesc() + "'");
    }
}
```

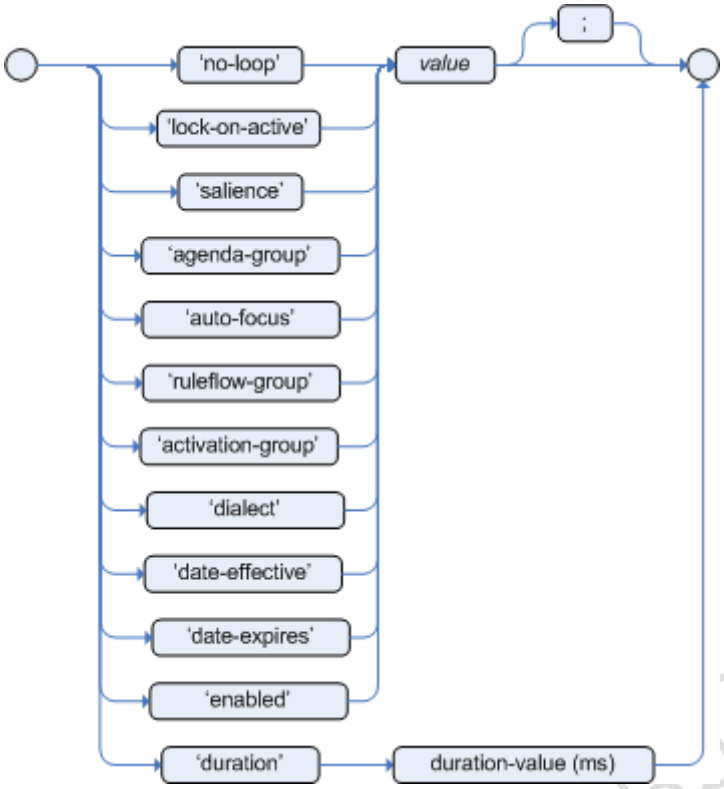
执行之后，打印结果：

```
Send message to email,the fired rule is 'test-global', and description    is 'to send email!'
Fire 1 rule(s)!
```

上面的实例完成了一个规则从触发到通过 global 调用 emailService 方法的实现。

4.3 规则属性

规则属性提供了一种声明性的方式来影响规则的行为。有些很简单，而其他的则是复杂子系统的一部分，比如规则流。如果想充分使用 Drools 提供的便利，应该对每个属性都有正确的理解。



4.3.1 no-loop

定义当前的规则是否不允许多次循环执行，默认是 false，也就是当前的规则只要满足条件，可以无限次执行。什么情况下会出现规则被多次重复执行呢？下面看一个实例：

```
package com.rules

import com.secbro.drools.model.Product;

rule updateDistcount
    no-loop false
    when
        productObj:Product(discount > 0);
    then
        productObj.setDiscount(productObj.getDiscount() + 1);
        System.out.println(productObj.getDiscount());
        update(productObj);
    end
```

其中 Product 对象的 discount 属性值默认为 1。执行此条规则时就会发现程序进入了死循环。也就是说对传入当前 workingMemory 中的 FACT 对象的属性进行修改，并调用 update 方法就会重新触发规则。从打印的结果来看，update 之后被修改的数据已经生效，在重新执行规则时并未被重置。当然对 Fact 对象数据的修改并不是一定需要调用 update 才可以生效，简单的使用 set 方法设置就可以完成，但仅仅调用 set 方法时并不会重新触发规则。所以，对 insert、retract、update 之类的方法使用时一定要慎重，否则极可能会造成死循环。

可以通过设置 no-loop 为 true 来避免规则的重新触发，同时，如果本身的 RHS 部分有 insert、retract、update 等触发规则重新执行的操作，也不会再次执行当前规则。

上面的设置虽然解决了当前规则的不会被重复执行，但其他规则还是会收到影响，比如下面的例子：

```
package com.rules

import com.secbro.drools.model.Product;

rule updateDistcount
	no-loop true
	when
		productObj:Product(discount > 0);
	then
		productObj.setDiscount(productObj.getDiscount() + 1);
		System.out.println(productObj.getDiscount());
		update(productObj);
	end

rule otherRule
	when
	productObj : Product(discount > 1);
	then
	System.out.println("被触发了" + productObj.getDiscount());
	end
```

此时执行会发现，当第一个规则执行 update 方法之后，规则 otherRule 也会被触发执行。如果注释掉 update 方法，规则 otherRule 则不会被触发。那么，这个问题是不是就没办法解决了？当然可以，那就是引入 lock-on-active true 属性。

4.3.2 ruleflow-group

在使用规则流的时候要用到 ruleflow-group 属性，该属性的值为一个字符串，作用是将规则划分为一个个的组，然后在规则流当中通过使用 ruleflow-group 属性的值，从而使用对应的规则。该属性会通过流程的走向确定要执行哪一条规则。在规则流中有具体的说明。

代码实例：

```
package com.rules

rule "test-ruleflow-group1"
	ruleflow-group "group1"
	when
	then
		System.out.println("test-ruleflow-group1 被触发");
	end
```

```
rule "test-ruleflow-group2"
    ruleflow-group "group1"
    when
    then
        System.out.println("test-ruleflow-group2 被触发");
    end
```

4.3.3 lock-on-active

当在规则上使用 ruleflow-group 属性或 agenda-group 属性的时候，将 lock-on-active 属性的值设置为 true，可避免因某些 Fact 对象被修改而使已经执行过的规则再次被激活执行。可以看出该属性与 no-loop 属性有相似之处，no-loop 属性是为了避免 Fact 被修改或调用了 insert、retract、update 之类的方法而导致规则再次激活执行，这里的 lock-on-active 属性起同样的作用，lock-on-active 是 no-loop 的增强版属性，它主要作用在使用 ruleflow-group 属性或 agenda-group 属性的时候。lock-on-active 属性默认值为 false。与 no-loop 不同的是 lock-on-active 可以避免其他规则修改 FACT 对象导致规则的重新执行。

因 FACT 对象修改导致其他规则被重复执行示例：

```
package com.rules

import com.secbro.drools.model.Product;

rule rule1
    no-loop true
    when
        obj : Product(discount > 0);
    then
        obj.setDiscount(obj.getDiscount() + 1);
        System.out.println("新折扣为：" + obj.getDiscount());
        update(obj);
    end

rule rule2
    when
        productObj : Product(discount > 1);
    then
        System.out.println("其他规则被触发了" + productObj.getDiscount());
    end
```

执行之后打印结果为：

```
新折扣为：2
其他规则被触发了 2
第一次执行命中了 2 条规则！
```

其他规则（rule2）因 FACT 对象的改变而被出发了。

通过 lock-on-active 属性来避免被其他规则更新导致自身规则被重复执行示例：

```
package com.rules

import com.secbro.drools.model.Product;

rule rule1
	no-loop true
	when
		obj : Product(discount > 0);
	then
		obj.setDiscount(obj.getDiscount() + 1);
		System.out.println("新折扣为：" + obj.getDiscount());
		update(obj);
	end

rule rule2
	lock-on-active true
	when
		productObj : Product(discount > 1);
	then
		System.out.println("其他规则被触发了" + productObj.getDiscount());
	end
```

很明显在 rule2 的属性部分新增了 lock-on-active true。执行结果为：

```
新折扣为：2
第一次执行命中了 1 条规则！
```

标注了 lock-on-active true 的规则不再被触发。

4.3.4 salience

用来设置规则执行的优先级，salience 属性的值是一个数字，数字越大执行优先级越高，同时它的值可以是一个负数。默认情况下，规则的 salience 默认值为 0。如果不设置规则的 salience 属性，那么执行顺序是随机的。

示例代码：

```
package com.rules

rule salience1
	salience 3
	when
	then
		System.out.println("salience1 被执行");
	end

rule salience2
```

```
    salience 5
    when
    then
        System.out.println("salience2 被执行");
    end
```

执行结果：

```
salience2 被执行
salience1 被执行
```

显然，salience2 的优先级高于 salience1 的优先级，因此被先执行。

Drools 还支持动态 saline，可以使用绑定绑定变量表达式来作为 salience 的值。比如：

```
package com.rules

import com.secbro.drools.model.Product

rule salience1
    salience sal
    when
        Product(sal:discount);
    then
        System.out.println("salience1 被执行");
    end
```

这样，salience 的值就是传入的 FACT 对象 Product 的 discount 的值了。

4.3.5 agenda-group

规则的调用与执行是通过 StatelessKieSession 或 KieSession 来实现的，一般的顺序是创建一个 StatelessKieSession 或 KieSession，将各种经过编译的规则添加到 session 当中，然后将规则当中可能用到的 Global 对象和 Fact 对象插入到 Session 当中，最后调用 fireAllRules 方法来触发、执行规则。

在没有调用 fireAllRules 方法之前，所有的规则及插入的 Fact 对象都存放在一个 Agenda 表的对象当中，这个 Agenda 表中每一个规则及与其匹配相关业务数据叫做 Activation，在调用 fireAllRules 方法后，这些 Activation 会依次执行，执行顺序在没有设置相关控制顺序属性时（比如 salience 属性），它的执行顺序是随机的。

Agenda Group 是用来在 Agenda 基础上对规则进行再次分组，可通过为规则添加 agenda-group 属性来实现。agenda-group 属性的值是一个字符串，通过这个字符串，可以将规则分为若干个 Agenda Group。引擎在调用设置了 agenda-group 属性的规则时需要显示的指定某个 Agenda Group 得到 Focus（焦点），否则将不执行该 Agenda Group 当中的规则。

规则代码：

```
package com.rules

 rule "test agenda-group"
```

```
    agenda-group "abc"
    when
    then
        System.out.println("规则 test agenda-group  被触发");
    end
rule otherRule

    when
    then
        System.out.println("其他规则被触发");
    end
```

调用代码：

```
KieServices kieServices = KieServices.Factory.get();
KieContainer kieContainer = kieServices.getKieClasspathContainer();
KieSession kSession = kieContainer.newKieSession("ksession-rule");

kSession.getAgenda().getAgendaGroup("abc").setFocus();
kSession.fireAllRules();
kSession.dispose();
```

执行以上代码，打印结果为：

```
规则 test agenda-group  被触发
其他规则被触发
```

如果将代码 kSession.getAgenda().getAgendaGroup("abc").setFocus()注释掉，则只会打印出：

```
其他规则被触发
```

很显然，如果不设置指定 AgendaGroup 获得焦点，则该 AgendaGroup 下的规则将不会被执行。

4.3.6 auto-focus

在 agenda-group 章节，我们知道想要让 AgendaGroup 下的规则被执行，需要在代码中显式的设置 group 获得焦点。而此属性可配合 agenda-group 使用，代替代码中的显式调用。默认值为 false，即不会自动获取焦点。设置为 true，则可自动获取焦点。

对于规则的执行的控制，还可以使用 org.kie.api.runtime.rule. AgendaFilter 来实现。用户可以实现该接口的 accept 方法，通过规则当中的属性值来控制是否执行规则。

方法体如下：

```
boolean accept(Match match);
```

在该方法当中提供了一个 Match 参数，通过该参数可以获得当前正在执行的规则对象和属性。该方法要返回一个布尔值，返回 true 就执行规则，否则不执行。

auto-focus 使用示例代码：

规则代码：

```
package com.rules
```

```
rule "test agenda-group"

    agenda-group "abc"
    auto-focus true

    when
    then
        System.out.println("规则 test agenda-group  被触发");
    end
```

执行规则代码：

```
KieServices kieServices = KieServices.Factory.get();
KieContainer kieContainer = kieServices.getKieClasspathContainer();
KieSession kSession = kieContainer.newKieSession("ksession-rule");
kSession.fireAllRules();
kSession.dispose();
```

执行结果：

```
规则 test agenda-group  被触发
```

这里，我们没有在代码中显式的让 test agenda-group 获取焦点，但规则同样被执行了，说明属性配置已生效。

AgendaFilter 代码实例

规则文件代码：

```
package com.rules

rule "test-agenda-group"

    when
    then
        System.out.println("规则 test-agenda-group  被触发");
    end

rule other

    when
    then
        System.out.println("规则 other 被触发");
    end
```

实现的 MyAgendaFilter 代码：

```
package com.secbro.drools.filter;

import org.kie.api.runtime.rule.AgendaFilter;
import org.kie.api.runtime.rule.Match;

/**
```

```
 * Created by zhuzs on 2017/7/19.
 */
public class MyAgendaFilter implements AgendaFilter{

    private String ruleName;

    public MyAgendaFilter(String ruleName) {
        this.ruleName = ruleName;
    }

    @Override
    public boolean accept(Match match) {
        return match.getRule().getName().equals(ruleName) ? true : false;
    }
    // 省略 getter/setter 方法
}
```

测试方法：

```
KieServices kieServices = KieServices.Factory.get();
KieContainer kieContainer = kieServices.getKieClasspathContainer();
KieSession kSession = kieContainer.newKieSession("ksession-rule");

AgendaFilter filter = new MyAgendaFilter("test-agenda-group");
kSession.fireAllRules(filter);
kSession.dispose();
```

执行结果：

```
规则 test-agenda-group 被触发
```

在执行规则的 Filter 中传入的规则名称为 test-agenda-group，此规则被执行。而对照组的规则 other，却未被执行。

4.3.7 activation-group

该属性将若干个规则划分成一个组，统一命名。在执行的时候，具有相同 activation-group 属性的规则中只要有一个被执行，其它的规则都不再执行。可以用类似 salience 之类属性来实现规则的执行优先级。该属性以前也被称为异或（Xor）组，但技术上并不是这样实现的，当提到此概念，知道是该属性即可。

实例代码：

```
package com.rules

 rule "test-activation-group1"
    activation-group "foo"
    when
    then
        System.out.println("test-activation-group1 被触发");
```

```
    end

rule "test-activation-group2"
    activation-group "foo"
    salience 1
    when
    then
        System.out.println("test-activation-group2 被触发");
    end
```

执行规则之后，打印结果：

```
test-activation-group2 被触发
```

以上实例证明，同一 activation-group 优先级高的被执行，其他规则不会再被执行。

4.3.8 dialect

该属性用来定义规则（LHS、RHS）当中要使用的语言类型，可选值为“java”或“mvel”。默认情况下使用 java 语言。当在包级别指定方言时，这个属性可以在具体的规则中覆盖掉包级别的指定。

```
dialect "mvel"
```

4.3.9 date-effective

该属性是用来控制规则只有在到达指定时间后才会触发。在规则运行时，引擎会拿当前操作系统的时间与 date-effective 设置的时间值进行比对，只有当系统时间大于等于 date-effective 设置的时间值时，规则才会触发执行，否则执行将不执行。在没有设置该属性的情况下，规则随时可以触发。

date-effective 的值为一个日期型的字符串，默认情况下，date-effective 可接受的日期格式为“dd-MMM-yyyy”。例如 2017 年 7 月 20 日，在设置为 date-effective 值时，如果操作系统为英文的，那么应该写成“20-Jul-2017”；如果是中文操作系统则为“20-七月-2017”。

目前在 win10 操作系统下验证，中文和英文格式均支持。而且在上面日期格式后面添加空格，添加其他字符并不影响前面日期的效果。

示例代码：

```
package com.rules

rule "test-date"
//    date-effective "20-七月-2017 aa"
//    date-effective "20-七月-2017"
//    date-effective "20-Jul-2017aaa"
    date-effective "20-Jul-2017"
    when
    then
```

```
        System.out.println("规则被执行");
    end
```

值得注意的是以上注释掉的格式均能成功命中规则与后面的字符无关，因为默认时间格式只取字符串的指定位数进行格式化。

晋级用法：上面已经提到了，其实针对日期之后的时间是无效的。那么如果需要精确到时分秒改如何使用呢？可以通过设置 drools 的日期格式化来完成任意格式的时间设定，而不是使用默认的格式。在调用代码之前设置日期格式化格式：

```
System.setProperty("drools.dateformat", "yyyy-MM-dd HH:mm");
```

在规则文件中就可以按照上面设定的格式来传入日期：

```
date-effective "2017-07-20 16:31"
```

4.3.10 date-expires

此属性与 date-effective 的作用相反，用来设置规则的过期时间。时间格式可完全参考 date-effective 的时间格式。引擎在执行规则时会检查属性是否设置，如果设置则比较当前系统时间与设置时间，如果设置时间大于系统时间，则执行规则，否则不执行。实例代码同样参考 date-effective。

4.3.11 duration

已废弃。设置该属性，规则将指定的时间之后在另外一个线程里触发。属性值为一个长整型，单位是毫秒。如果属性值设置为 0，则标示立即执行，与未设置相同。

4.3.12 enabled

设置规则是否可用。true：表示该规则可用；false：表示该规则不可用。

4.4 定时器和日历

4.4.1定时器

规则用基于 interval（间隔）和 cron 的定时器（timer），替代了被标注过时的 duration 属性。timer 属性的使用示例：

```
timer ( int: <initial delay> <repeat interval>? )
timer ( int: 30s )
timer ( int: 30s 5m )

timer ( cron: <cron expression> )
timer ( cron:* 0/15 * * * ? )
```

间隔定时器用 int 来定义，它遵循 java.util.Timer 对象的使用方法。具有延迟和重复执行的选择。其中第一个参数表示启动之后延迟多长时间执行，第二个参数表示每隔多久执行一次。

Cron 定时器用 cron 来定义，使用标准的 Unix cron 表达式。示例代码如下：

```
rule "Send SMS every 15 minutes"
      timer (cron:* 0/15 * * * ?)
when
      $a : Alarm( on == true )
then
      channels[ "sms" ].insert( new Sms( $a.mobileNumber, "The alarm is still on" );
end
```

上面代码实现了每隔 15 分钟发送一封邮件的部分规则代码。

下面以一个模拟的系统报警器来示例一下 Timer 的使用。规则 timer 每隔一秒执行一次，当满足触发规则返回结果至 ResultEvent 对象中，业务系统拿到报警信息，并打印。为了达到模拟的效果，使用了 KieSession 的 fireUntilHalt 方法和 halt 方法。示例代码如下。

规则文件：

```
package com.rules
import java.util.Date
import java.util.List
import com.secbro.drools.testTimer.Server

global com.secbro.drools.testTimer.ResultEvent event

rule "timerTest"
      timer (cron:0/1 * * * * ?)
      when
            server : Server(times > 10)
      then
      System.out.println("已经尝试"+server.getTimes()+"次，超过预警次数！");
      event.getEvents().add(new java.util.Date() + " - 服务器已经尝试" + server.getTimes()
+ "次，依旧失败，特发次报警信息！");
end
```

Server 类：

```
package com.secbro.drools.testTimer;

/**
 * Created by zhuzs on 2017/7/21.
 */
public class Server {
      // 尝试次数
      private int times;

      Server(int times) {
            this.times = times;
```

```
    }
//省略 getter/setter 方法
}
```

返回结果 ResultEvent 类：

```
package com.secbro.drools.testTimer;

import java.util.ArrayList;
import java.util.List;

/**
 * Created by zhuzs on 2017/7/21.
 */
public class ResultEvent {
    private List<String> events = new ArrayList<>();
    //省略 getter/setter 方法
  }
```

测试类：

```
package com.secbro.drools.testTimer;

import org.junit.Test;
import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;
import org.kie.api.runtime.rule.FactHandle;

/**
 * Created by zhuzs on 2017/7/21.
 */
public class TimerRulesTest {

    @Test
    public void timerTest() throws InterruptedException {

        final KieSession kieSession = createKnowledgeSession();
        ResultEvent event = new ResultEvent();
        kieSession.setGlobal("event", event);

        final Server server = new Server(1);

        new Thread(new Runnable() {
            public void run() {
                kieSession.fireUntilHalt();
            }
        }).start();
```

```
            FactHandle serverHandle = kieSession.insert(server);

            for (int i = 8; i <= 15; i++) {
                Thread.sleep(2000);
                server.setTimes(++i);
                kieSession.update(serverHandle, server);
            }

            Thread.sleep(3000);
            kieSession.halt();
            System.out.println(event.getEvents());
        }

        private KieSession createKnowledgeSession() {
            KieServices kieServices = KieServices.Factory.get();
            KieContainer kieContainer = kieServices.getKieClasspathContainer();
            KieSession kSession = kieContainer.newKieSession("ksession-rule");
            return kSession;
        }

}
```

kmodule.xml 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://www.drools.org/xsd/kmodule">
    <kbase name="rules" packages="com.rules">
        <ksession name="ksession-rule"/>
    </kbase>
</kmodule>
```

控制台打印：

```
已经尝试 11 次，超过预警次数！
已经尝试 11 次，超过预警次数！
已经尝试 13 次，超过预警次数！
已经尝试 13 次，超过预警次数！
已经尝试 15 次，超过预警次数！
已经尝试 15 次，超过预警次数！
已经尝试 15 次，超过预警次数！
[Fri Jul 21 21:04:11 CST 2017 - 服务器已经尝试 11 次，依旧失败，特发次报警信息！, Fri Jul 21 21:04:12 CST 2017 - 服务器已经尝试 11 次，依旧失败，特发次报警信息！, Fri Jul 21 21:04:13 CST 2017 - 服务器已经尝试 13 次，依旧失败，特发次报警信息！, Fri Jul 21 21:04:14 CST 2017 - 服务器已经尝试 13 次，依旧失败，特发次报警信息！, Fri Jul 21 21:04:15 CST 2017 - 服务器已经尝试 15 次，依旧失败，特发次报警信息！, Fri Jul 21 21:04:16 CST 2017 - 服务器已经尝试 15 次，依旧失败，特发次报警信息！, Fri Jul 21
```

```
21:04:17 CST 2017 - 服务器已经尝试 15 次，依旧失败，特发次报警信息！]
```

很显然，定时器每隔一秒执行一次，当满足规则触发条件时，将结果放入 ResultEvent 中。

4.4.2日历

日历可以单独应用于规则中，也可以和 timer 结合使用在规则中使用。通过属性 calendars 来定义日历。如果是多个日历，则不同日历之间用逗号进行分割。

在 Drools 中，日历的概念只是将日历属性所选择的时间映射成布尔值，设置为规则的属性，控制规则的触发。Drools 可以通过计算当期日期和时间来决定是否允许规则的触发。

此示例首先需要引入 quarts 框架：

```
<dependency>
    <groupId>org.opensymphony.quartz</groupId>
    <artifactId>quartz</artifactId>
    <version>1.6.1</version>
</dependency>
```

实现 Quarts 的 Calendar 转换为 Drools 的 Calendar 的转换器 CalendarWrapper：

```
public class CalendarWrapper implements Calendar{

    private WeeklyCalendar cal;

    public CalendarWrapper(WeeklyCalendar cal) {
        this.cal = cal;
    }

    @Override
    public boolean isTimeIncluded(long timestamp) {
        return cal.isTimeIncluded(timestamp);
    }

    public WeeklyCalendar getCal() {
        return cal;
    }

    public void setCal(WeeklyCalendar cal) {
        this.cal = cal;
    }

}
```

规则文件：

```
package com.rules
```

```
rule "calenderTest"

	calendars "weekday"
//	  timer (int:0 1s) //  可以和 timer 配合使用

	when
		str : String();
	then
		System.out.println("In rule - " + drools.getRule().getName());
		System.out.println("String matched " + str);
	end
```

测试方法：

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

		final KieSession kieSession = createKnowledgeSession();

		WeeklyCalendar weekDayCal = new WeeklyCalendar();
		// 默认包含所有的日期都生效
		weekDayCal.setDaysExcluded(new boolean[]{false, false, false, false, false, false,
false,false,false});
//		weekDayCal.setDayExcluded(java.util.Calendar.THURSDAY, true); // 设置为
true 则不包含此天，周四
		Calendar calendar = new CalendarWrapper(weekDayCal);

		kieSession.getCalendars().set("weekday", calendar);

		kieSession.insert(new String("Hello"));
		kieSession.fireAllRules();

		kieSession.dispose();
		System.out.println("Bye");
	}

	protected KieSession createKnowledgeSession() {
			KieServices kieServices = KieServices.Factory.get();
			KieSessionConfiguration conf = kieServices.newKieSessionConfiguration();

			KieContainer kieContainer = kieServices.getKieClasspathContainer();
			KieSession kSession = kieContainer.newKieSession("ksession-rule", conf);
			return kSession;
	 }
```

执行测试方法打印结果：

```
In rule - calenderTest
```

```
String matched Hello
Bye
```

其中测试过程中的注意点已经在代码中进行标注，比如 Calendar 可以和 timer 共同使用；如何设置 WeeklyCalendar 中哪一天执行，哪一天不执行。

4.5 LHS 语法

4.5.1 LHS 简介

在规则文件组成章节，我们已经了解了 LHS 的基本使用说明。LHS 是规则条件部分的统称，由 0 个或多个条件元素组成。前面我们已经提到，如果没有条件元素那么默认就是 true。

没有条件元素，官方示例：

```
rule "no CEs"
when
    // empty
then
    ... // actions (executed once)
end

// The above rule is internally rewritten as:

rule "eval(true)"
when
    eval( true )
then
    ... // actions (executed once)
end
```

如果有多条规则元素，默认它们之间是“和”的关系，也就是说必须同时满足所有的条件元素才会触发规则。官方示例：

```
rule "2 unconnected patterns"
when
    Pattern1()
    Pattern2()
then
    ... // actions
end

// The above rule is internally rewritten as:

rule "2 and connected patterns"
```

```
when
    Pattern1()
    and Pattern2()
then
    ... // actions
end
```

和“or”不一样，“and”不具有优先绑定的功能。因为生命一次只能绑定一个 FACT 对象，而当使用 and 时就无法确定声明的变量绑定到哪个对象上了。以下代码会编译出错。

```
$person : (Person( name == "Romeo" ) and Person( name == "Juliet"))
```

4.5.2 Pattern (条件元素)

Pattern 元素是最重要的一个条件元素，它可以匹配到插入 working memory 中的每个 FACT 对象。一个 Pattern 包含 0 到多个约束条件，同时可以选择性的进行绑定。

通过上图可以明确的知道 Pattern 的使用方式，左边变量定义，然后用冒号分割。右边 pattern 对象的类型也就是 FACT 对象，后面可以在括号内添加多个约束条件。最简单的一种形式就是，只有 FACT 对象，没有约束条件，这样一个 pattern 配到指定的 patternType 类即可。

比如，下面的 pattern 定义表示匹配 Working Memory 中所有的 Person 对象。

```
Person()
```

patternType 并不需要使用实际存在的 FACT 类，比如下面的定义表示匹配 Working Memory 中所有的对象。很明显，Object 是所有类的父类。

```
Object() // 匹配 working memory 中的所有对象
```

如下面的示例，括号内的表达式决定了当前条件是否会被匹配到，这也是实际应用中最常见的使用方法。

```
Person( age == 100 )
```

Pattern 绑定：当匹配到对象时，可以将 FACT 对象绑定到指定的变量上。这里的用法类似于 java 的变量定义。绑定之后，在后面就可以直接使用此变量。

```
rule ...
when
    $p : Person()
then
    System.out.println( "Person " + $p );
end
```

其中前缀$只是一个约定标识，有助于在复杂的规则中轻松区分变量和字段，但并不强制要求必须添加此前缀。

4.5.3约束（Pattern 的一部分）

前面我们已经介绍了条件约束在 Pattern 中位置了，那么什么是条件约束呢？简单来说就是一个返回 true 或者 false 的表达式，比如下面的 5 小于 6，就是一个约束条件。

```
Person( 5 < 6 )
```

从本质上来讲，它是 JAVA 表达式的一种增强版本（比如属性访问），同时它又有一些小的区别，比如 equals 方法和==的语言区别。下面我们就深入了解一下。

访问 JavaBean 中的属性

任何一个 JavaBean 中的属性都可以访问，不过对应的属性要提供 getter 方法或 isProperty 方法。比如：

```
Person( age == 50 )

// 与上面拥有同样的效果
Person( getAge() == 50 )
```

Drools 使用 java 标准的类检查，因此遵循 java 标准即可。同时，嵌套属性也是支持的，比如：

```
Person( address.houseNumber == 50 )

// 与上面写法相同
Person( getAddress().getHouseNumber() == 50 )
```

在使用有状态 session 的情况下使用嵌套属性需要注意属性的值可能被其他地方修改。要么认为它们是不可变的，当任何一个父引用被插入到 working memory 中。或者，如果要修改嵌套属性值，则应将所有外部 fact 标记更新。在上面的例子中，当 houseNumber 属性值改变时，任何一个包含 Address 的 Person 需要被标记更新。

Java 表达式

在 pattern 的约束条件中，可以任何返回结果为布尔类型的 java 表达式。当然，java 表达式也可以和增强的表达式进行结合使用，比如属性访问。可以通过使用括号来更改计算优先级，如在任一逻辑或数学表达式中。

```
Person( age > 100 && ( age % 10 == 0 ) )
```

也可以直接使用 java 提供的工具方法来进行操作计算：

```
Person( Math.round( weight / ( height * height ) ) < 25.0 )
```

在使用的过程中需要注意，在 LHS 中执行的方法只能是只读的，不能在执行方法过程中改变 FACT 对象的值，否则会影响规则的正确执行。

```
Person( incrementAndGetAge() == 10 ) //不要像这样在比较的过程中更新 Fact 对象
```

另外，FACT 对象的相关状态除了在 working memory 中可以进行更新操作，不应该在每次调用时使状态发生变化。

```
Person( System.currentTimeMillis() % 1000 == 0 ) // 不要这样实现
```

标准 Java 运算符优先级也适用于此处，详情参考下面的运算符优先级列表。所有的操作符都有标准的 Java 语义，除了==和！=。它们是 null 安全的，就相当于 java 中比较两个字符串时把常量字符串放前面调用 equals 方法的效果一样。

约束条件的比较过程中是会进行强制类型转换的，比如在数据计算中传入字符串“10”，则能成功转换成数字 10 进行计算。如果传入的值无法进行转换，比如传了“ten”，会抛出异常。

逗号分隔符

逗号可以对约束条件进行分组，它的作用相当于“AND”。

```
// Person 的年龄要超过 50，并且重量 超过 80 kg
Person( age > 50, weight > 80 )
```

虽然“&&”和“,”拥有相同的功能，但是它们有不同的优先级。“&&”优先于“||”，“&&”和“||”又优先于“,”。建议优先使用“,”分隔符，因为它更利于阅读理解和引擎的操作优化。同时，逗号分隔符不能和其他操作符混合使用，比如：

```
Person( ( age > 50, weight > 80 ) || height > 2 ) // 会编译错误

// 使用此种方法替代
Person( ( age > 50 && weight > 80 ) || height > 2 )
```

绑定变量

一个属性可以绑定到一个变量：

```
// 2 person 的 age 属性值相同
Person( $firstAge : age ) // 绑定
Person( age == $firstAge ) // 约束表达式
```

前缀$只是个通用惯例，在复杂规则中可以通过它来区分变量和属性。为了向后兼容，允许（但不推荐）混合使用约束绑定和约束表达式。

```
// 不建议这样写
Person( $age : age * 2 < 100 )

// 推荐（分离绑定和约束表达式）
Person( age * 2 < 100, $age : age )
```

使用操作符“==”来绑定变量，Drools 会使用散列索引来提高执行性能。

内部类分组访问

通常情况，我们访问一个内部类的多个属性时会有如下的写法：

```
Person( name == "mark", address.city == "london", address.country == "uk" )
```

Drools 提供的分组访问可以更加方便进行使用：

```
Person( name == "mark", address.( city == "london", country == "uk") )
```

注意前缀'.'是用来区分嵌套对象约束和方法调用所必需的。

内部强制转换

在使用内部类的时候，往往需要将其转换为父类，在规则中可以通过“#”来进行强制转换：

```
Person( name == "mark", address#LongAddress.country == "uk" )
```

上面的例子将 Address 强制转换为 LongAddress.，使得 getter 方法变得可用。如果无法强制转换，表达式计算的结果为 false。强制转换也支持全路径的写法：

```
Person( name == "mark", address#org.domain.LongAddress.country == "uk" )
```

多次内部转换语法：

```
Person( name == "mark", address#LongAddress.country#DetailedCountry.population > 10000000 )
```

也可以使用 instanceof 操作符进行判断，判断之后将进一步使用该属性进行比较。

```
Person( name == "mark", address instanceof LongAddress, address.country == "uk" )
```

日期字符

规则语法中除了支持 JAVA 标准字符，同时也支持日期字符。Drools 默认支持的日期格式为“dd-mmm-yyyy”，可以通过设置系统变量“drools.dateformat”的值来改变默认的日期格式。

```
Cheese( bestBefore < "27-Oct-2009" )
```

List 和 Map 的访问

访问 List：

```
// 和 childList(0).getAge() == 18 效果相同
Person( childList[0].age == 18 )
```

根据 key 访问 map：

```
// 和 credentialMap.get("jsmith").isValid()相同
Person( credentialMap["jsmith"].valid )
```

&&和||

约束表达式可以通过&&和||来进行判断比较，用法与标准 Java 相似，当组合使用时，可通过括号来区分优先级。

```
Person( age > 30 && < 40 )
Person( age ( (> 30 && < 40) ||(> 20 && < 25) ) )
Person( age > 30 && < 40 || location == "london" )
```

DRL 特殊操作符

“< ⇐ > >=”操作符用于属性的比较时按照默认的排序，比如日期属性使用小于号比较，将按照日期前后排序；当使用在 String 字符串的比较时，则按照字母顺序进行排序。此操作符仅适用于可进行比较的属性值。

```
Person( firstName < $otherFirstName )
Person( birthDate < $otherBirthDate )
```

“!.”提供了一个默认空校验的操作。使用此操作符时，会先校验当前对象是否为 null，如果不为 null 再调用其方法或属性进行判断。一旦当前操作对象为 null，则相当于判断结果为 false。

```
Person( $streetName : address!.street )

// 上面的写法相当于
Person( address != null, $streetName : address.street )
```

matches 操作符可使用 Java 的正则表达式进行字符串的匹配，通常情况下使用正则表达式字符串进行匹配，但也支持变量值为正确的表达式的方式。此操作符仅适用于字符串属性。如果属性值为 null，匹配的结果始终为 false。

```
Cheese( type matches "(Buffalo)?\\S*Mozzarella" )
```

not matches 方法与 matches 相同，唯一不同的是返回的结果与之相反。

```
Cheese( type not matches "(Buffalo)?\\S*Mozzarella" )
```

contains 操作符判断一个集合属性或元素是否包含指定字符串或变量值。仅适用于集合属性。也可以用于替代 String.contains()来检查约束条件。not contains 用法与之相同，结果取反。

```
CheeseCounter( cheeses contains "stilton" ) // 包含字符串
CheeseCounter( cheeses contains $var ) // 包含变量

Cheese( name contains "tilto" )
Person( fullName contains "Jr" )
String( this contains "foo" )
```

memberOf 用来检查属性值是否为集合，此集合的表示必须为变量。not memberOf 使用方法相同，结果取反。

```
CheeseCounter( cheese memberOf $matureCheeses )
```

soundslike 的效果与 matches 相似，但它用来检查一个字符串的发音是否与指定的字符十分相似（使用英语发音）。

```
// 匹配 "fubar" 或 "foobar"
Cheese( name soundslike 'foobar' )
```

str 操作用来比较一个字符串是否以指定字符串开头或结尾，有可以用于比较字符串的长度。

```
Message( routingValue str[startsWith] "R1" )

Message( routingValue str[endsWith] "R2" )

Message( routingValue str[length] 17 )
```

in 和 notin 用来匹配一组数据中是否含一个或多个匹配的字符串，使用的方法与数据库中 in 的使用方法相似。待匹配的数据可以是字符串、变量。

```
Person( $cheese : favouriteCheese )
Cheese( type in ( "stilton", "cheddar", $cheese ) )
```

运算符优先级

操作类型	操作符	备注
（嵌套/空安全）属性访问	.!.	非标准 java 语义
List/Map 访问	[ ]	非标准 java 语义
约束绑定	:	非标准 java 语义
乘除	*/%	
加减	\+-	
移位	<<>>>>	
关系	<>⇐>=instanceof	
等	==!=	未使用标准 java 语义，某些语义相当于 equals。
非短路 AND	&	
非短路异或	^	
非短路包含 OR	|	
逻辑与	&&	
逻辑或	||	
三元运算符	? :	
逗号分隔，相当于 and	,	非标准 java 语义

4.5.4其他语法

其他不常用语法读者可自行查阅官方文档使用，后续补充相关内容。

4.6 RHS 语法

4.6.1使用说明

RHS 是满足 LHS 条件之后进行后续处理部分的统称，该部分包含要执行的操作的列表信息。RHS 主要用于处理结果，因此不建议在此部分再进行业务判断。如果必须要业务判断需要考虑规则设计的合理性，是否能将判断部分放置于 LHS，那里才是判断条件应该在的地方。同时，应当保持 RHS 的精简和可读性。

如果在使用的过程中发现需要在 RHS 中使用 AND 或 OR 来进行操作，那么应该考虑将一根规则拆分成多个规则。

RHS 的主要功能是对 working memory 中的数据进行 insert、update、delete 或 modify 操作，Drools 提供了相应的内置方法来帮助实现这些功能。

update(object,handle)：执行此操作更新对象（LHS 绑定对象）之后，会告知引擎，并重新触发规则匹配。

update(object)：效果与上面方法类似，引擎会默认查找对象对应的 handle。

使用属性监听器，来监听 JavaBean 对象的属性变更，并插入到引擎中，可以避免在对象更改之后调用 update 方法。当一个字段被更改之后，必须在再次改变之前调用 update 方法，否则可能导致引擎中的索引问题。而 modify 关键字避免了这个问题。

insert(newSomething())：创建一个新对象放置到 working memory 中。

insertLogical(newSomething())：功能类似于 insert，但当创建的对象不再被引用时，将会被销毁。

delete(handle)：从 working memory 中删除对象。

其实这些宏函数是 KnowledgeHelper 接口中方法对应的快捷操作，通过它们可以在规则文件中访问 Working Memory 中的数据。预定义变量 drools 的真实类型就是 KnowledgeHelper，因此可以通过 drools 来调用相关的方法。具体每个方法的使用说明可以参考类中方法的说明。

通过预定义的变量 kcontext 可以访问完整的 Knowledge Runtime API，而 kcontext 对应的接口为 KieContext。查看 KieContext 类会发现提供了一个 getKieRuntime()方法，该方法返回 KieRuntime 接口类，该接口中提供了更多的操作方法，对 RHS 编码逻辑有很大作用。

4.6.2insert 函数

insert 的作用与在 Java 类当中调用 KieSession 的 insert 方法效果一样，都是将 Fact 对象插入到当前的 Working Memory 当中，基本用法格式如下：

```
insert(newSomething());
```

调用 insert 之后，规则会进行重新匹配，如果没有设置 no-loop 为 true 或 lock-on-active 为 true 的规则，如果条件满足则会重新执行。update、modify、delete 都具有同样的特性，因此在使用时需特别谨慎，防止出现死循环。

规则文件 insert.drl

```
package com.rules

import com.secbro.drools.model.Product

rule "insert-check"
    salience 1
    when
       $p : Product(type == GOLD);
     then
        System.out.println("insert-check:insert
Product success and it's type is " +
$p.getType());
     end
```

```
rule "insert-action"
    salience 2
    when
    then
        System.out.println("insert-action : To
insert the Product");
        Product p = new Product();
        p.setType(Product.GOLD);
        insert(p);
    end
```

测试代码：

```
@Test
public void commonTest(){
    KieServices kieServices = KieServices.get();
    KieContainer kieContainer =
kieServices.getKieClasspathContainer();
    KieSession kieSession =
kieContainer.newKieSession("ksession-rule");
    int count = kieSession.fireAllRules();
    kieSession.dispose();
    System.out.println("Fire " + count + "
rules!");
}
```

打印日志：

```
insert-action : To insert the Product
insert-check:insert Product success and it's type is GOLD
Fire 2 rules!
```

根据优先级首先执行 insert 操作的规则，然后执行结果检测。

4.6.3update 函数

update 函数可对 Working Memory 中的 FACT 对象进行更新操作，与 StatefulSession 中的 update 的作用基本相同。查看 KnowledgeHelper 接口中的 update 方法可以发现，update 函数有多种参数组合的使用方法。在实际使用中更多的会传入 FACT 对象来进行更新操作。具体的使用方法前面章节已经有具体例子，不再重复示例。

```
void update(FactHandle handle, Object newObject);

void update(FactHandle newObject);
void update(FactHandle newObject, BitMask mask, Class<?> modifiedClass);

void update(Object newObject);
void update(Object newObject, BitMask mask, Class<?> modifiedClass);
```

4.6.4delete 函数

将 Working Memory 中的 FACT 对象删除，与 kession 中的 retract/delete 方法效果一样。同时 delete 函数和 retract 效果也相同，但后者已经被废弃。

4.6.5modify 函数

modify 是基于结构化的更新操作，它将更新操作与设置属性相结合，用来更改 FACT 对象的属性。语法格式如下：

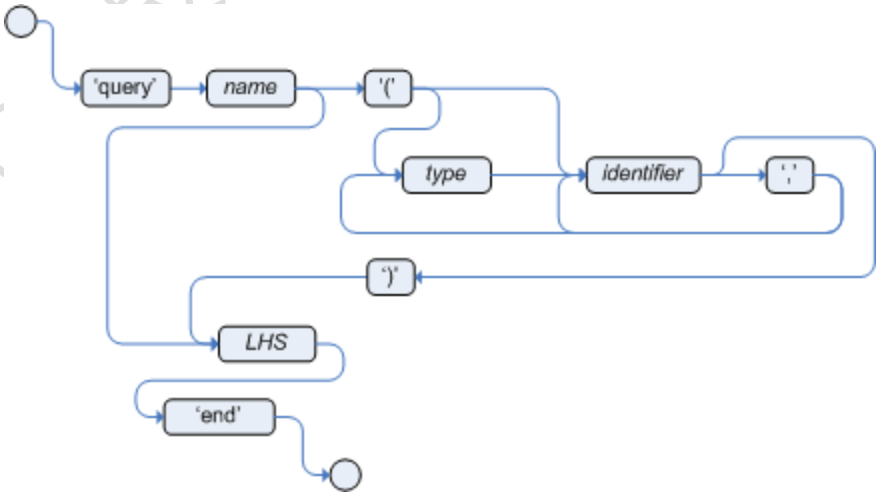
```
modify ( <fact-expression> ) {
    <expression> [ , <expression> ]*
}
```

其中<fact-expression>必须是 FACT 对象的引用，expression 中的属性必须提供 setter 方法。在调用 setter 方法时，不必再写 FACT 对象的引用，编译器会自动添加。

```
rule "modify stilton"
when
    $stilton : Cheese(type == "stilton")
then
    modify( $stilton ){
        setPrice( 20 ),
        setAge( "overripe" )
    }
end
```

4.7 Query 查询

首先，我们先来看一下 query 的语法结构图：



Query 语法提供了一种查询 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。Query 有一个可选参数集

合，每一个参数都有可选的类型。如果没有指定类型，则默认为 Object 类型。引擎会尝试强转为需要的类型。对于 KieBase 来说，query 的名字是全局性的，因此不要向同一 RuleBase 的不同包添加相同名称的 query。

使用 ksession.getQueryResults("name")方法可以获得查询的结果，其中 name 为 query 的名称，方法的返回结果一个列表，从中可以获取匹配查询到的对象。

下面是具体的实例：

```
package com.rules
import com.secbro.drools.model.Person;

rule "query-test"
	agenda-group "query-test-group1"

	when
		$person : Person()
	then
		System.out.println("The rule query-test fired!");
	end

query "query-1"
	$person : Person(age > 30)
end

query "query-2"(String nameParam)
	$person : Person(age > 30,name == nameParam)
end
```

测试代码一：

```
@Test
	public void queryTest() {
		KieSession kieSession = this.getKieSession("query-test-group1");

		Person p1 = new Person();
		p1.setAge(29);
		Person p2 = new Person();
		p2.setAge(40);

		kieSession.insert(p1);
		kieSession.insert(p2);
		int count = kieSession.fireAllRules();
		System.out.println("Fire " +count + " rule(s)!");

		QueryResults results = kieSession.getQueryResults("query-1");
		System.out.println("results size is " + results.size());
		for(QueryResultsRow row : results){
			Person person = (Person) row.get("$person");
```

```
            System.out.println("Person from WM, age : " + person.getAge());
        }

        kieSession.dispose();
    }
```

执行测试代码一打印结果：

```
The rule query-test fired!
The rule query-test fired!
Fire 2 rule(s)!
results size is 1
Person from WM, age : 40
```

通过执行结果可以看到，我们拿到了 WM 中的符合条件的结果。在测试代码中也展示了如何获取结果列表及从结果列表中获得对象的方法。

测试代码二：

```
@Test
    public void queryWithParamTest() {
        KieSession kieSession = this.getKieSession("query-test-group1");

        Person p1 = new Person();
        p1.setAge(29);
        p1.setName("Ross");
        Person p2 = new Person();
        p2.setAge(40);
        p2.setName("Tom");

        kieSession.insert(p1);
        kieSession.insert(p2);
        int count = kieSession.fireAllRules();
        System.out.println("Fire " +count + " rule(s)!");

        QueryResults results = kieSession.getQueryResults("query-2","Tom");
        System.out.println("results size is " + results.size());
        for(QueryResultsRow row : results){
            Person person = (Person) row.get("$person");
            System.out.println("Person from WM, age : " + person.getAge() + "; name :"
+ person.getName());
        }

        kieSession.dispose();
    }
```

此段代码执行的结果如下：

```
The rule query-test fired!
The rule query-test fired!
Fire 2 rule(s)!
```

```
results size is 1
Person from WM, age : 40; name :Tom
```

这段代码中我们添加了参数，通过参数可以进一步过滤结果。Query 支持多参数，通过逗号分隔具体的参数。具体的使用方法参考上面的代码。

4.8 结果条件

在 Java 中，如果有重复的代码我们会考虑进行重构，抽取公共方法或继承父类，以减少相同的代码在多处出现，达到代码的最优管理和不必要的麻烦。Drools 同样提供了类似的功能。下面我们以实例来逐步说明。

像下面最原始的两条规则，有相同的业务判断，也有不同的地方：

```
package com.rules.conditional
import com.secbro.drools.model.Customer;
import com.secbro.drools.model.Car;

rule "conditional1:Give 10% discount to customers older than 60"
	agenda-group "conditional1"
when
	$customer : Customer( age > 60 )
then
	modify($customer) { setDiscount( 0.1 ) };
	System.out.println("Give 10% discount to customers older than 60");
end

rule "conditional1:Give free parking to customers older than 60"
agenda-group "conditional1"
when
	$customer : Customer( age > 60 )
	$car : Car ( owner == $customer )
then
	modify($car) { setFreeParking( true ) };
	System.out.println("Give free parking to customers older than 60");
end
```

现在 Drools 提供了 extends 特性，也就是一个规则可以继承另外一个规则，并获得其约束条件。改写之后执行效果相同，代码如下：

```
package com.rules.conditional
import com.secbro.drools.model.Customer;
import com.secbro.drools.model.Car;

rule "conditional2:Give 10% discount to customers older than 60"
	agenda-group "conditional2"
when
	$customer : Customer( age > 60 )
```

```
then
    modify($customer) { setDiscount( 0.1 ) };
    System.out.println("conditional2:Give 10% discount to customers older than 60");
end

rule "conditional2:Give free parking to customers older than 60"
    extends "conditional2:Give 10% discount to customers older than 60"
agenda-group "conditional2"
when
    $car : Car ( owner == $customer )
then
    modify($car) { setFreeParking( true ) };
    System.out.println("conditional2:Give free parking to customers older than 60");
end
```

我们可以看到上面使用了 extends，后面紧跟的是另外一条规则的名称。这样，第二条规则同时拥有了第一条规则的约束条件。只需要单独写此条规则自身额外需要的约束条件即可。那么，现在是否是最优的写法吗？当然不是，还可以将两条规则合并成一条来规则。这就用到了 do 和标记。

```
package com.rules.conditional
import com.secbro.drools.model.Customer;
import com.secbro.drools.model.Car;

rule "conditional3:Give 10% discount to customers older than 60"
    agenda-group "conditional3"
when
    $customer : Customer( age > 60 )
    do[giveDiscount]
    $car : Car(owner == $customer)
then
    modify($car) { setFreeParking(true) };
        System.out.println("conditional3:Give free parking to customers older than 60");
then[giveDiscount]
    modify($customer){
        setDiscount(0.1)
    };
    System.out.println("conditional3:Give 10% discount to customers older than 60");
end
```

在 then 中标记了 giveDiscount 处理操作，在 when 中用 do 来调用标记的操作。这样也当第一个约束条件判断完成之后，就执行标记 giveDiscount 中的操作，然后继续执行 Car 的约束判断，通过之后执行默认的操作。

在 then 中还可以添加一些判断来执行标记的操作，这样就不必每次都执行 do 操作，而是每当满足 if 条件之后才执行：

```
package com.rules.conditional
import com.secbro.drools.model.Customer;
```

```
import com.secbro.drools.model.Car;

rule "conditional4:Give 10% discount to customers older than 60"
	agenda-group "conditional4"
when
	$customer : Customer( age > 60 )
	if(type == "Golden") do[giveDiscount]
	$car : Car(owner == $customer)
then
	modify($car) { setFreeParking(true) };
		System.out.println("conditional4:Give free parking to customers older than 60");
then[giveDiscount]
	modify($customer){
		setDiscount(0.1)
	};
	System.out.println("conditional4:Give 10% discount to customers older than 60");
end
```

同时，还可以通过 break 来中断后续的判断。

```
package com.rules.conditional
import com.secbro.drools.model.Customer;
import com.secbro.drools.model.Car;

rule "conditional5:Give 10% discount to customers older than 60"
	agenda-group "conditional5"
when
	$customer : Customer( age > 60 )
	if(type == "Golden") do[giveDiscount10]
	else if (type == "Silver") break[giveDiscount5]
	$car : Car(owner == $customer)
then
	modify($car) { setFreeParking(true) };
		System.out.println("conditional5:Give free parking to customers older than 60");
then[giveDiscount10]
	modify($customer){
		setDiscount(0.1)
	};
	System.out.println("giveDiscount10:Give 10% discount to customers older than 60");
then[giveDiscount5]
	modify($customer){
		setDiscount(0.05)
	};
	System.out.println("giveDiscount5:Give 10% discount to customers older than 60");
end
```

以上规则的执行测试代码如下，执行结果可自行尝试，源代码已经存放在 GitHub：

https://github.com/secbr/drools。

4.9 注释

像 Java 开发语言一样，Drools 文件中也可以添加注释。注释部分 Drools 引擎是会将其忽略调的。单行注释使用“//”，示例如下：

```
rule "Testing Comments"
when
    // this is a single line comment
    eval( true ) // this is a comment in the same line of a pattern
then
    // this is a comment inside a semantic code block
end
```

注意，使用“#”进行注释已经被移除。

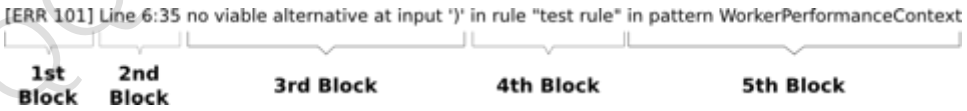
多行注释与 Java 相同，采用“/*注释内容*/”，来进行注释，示例如下：

```
rule "Test Multi-line Comments"
when
    /* this is a multi-line comment
       in the left hand side of a rule */
    eval( true )
then
    /* and this is a multi-line comment
       in the right hand side of a rule */
end
```

4.10 错误信息

Drools 5 引入了标准化的错误信息，可以快速的查找和解决问题。本节将介绍如何利用错误信息来进行快速定位问题和解决问题。

错误信息的各式如下图：



第一部分：错误编码；

第二部分：错误出现的行列信息；

第三部分：错误信息描述；

第四部分：上下午的第一行信息，通常表示发生错误的规则，功能，模板或查询。此部分并不强制。

第五部分：标识发生错误的 pattern（模式)。此部分并不强制。

下面以一组错误实例来分析常见的异常情况，首先用官网提供的例子来执行：

```
rule one
```

```
  when
    exists Foo()
    exits Bar()   // "exits"
  then
end
```

由于 exits 是错误的语法，因此会抛出异常，但此处需要注意的事在 Drools 7 中抛出的异常并非官网提供的异常。异常信息如下：

```
java.lang.RuntimeException: Error while creating KieBase[Message [id=1, kieBase=rules, level=ERROR, path=conditional1.drl, line=27, column=0
    text=[ERR 102] Line 27:6 mismatched input 'Bar' in rule "one" in pattern], Message [id=2, kieBase=rules, level=ERROR, path=conditional1.drl, line=0, column=0
    text=Parser returned a null Package]]
```

再看一个没有规则名称导致的错误：

```
rule
    when
     Object()
    then
       System.out.println("A RHS");
end
```

执行之后异常信息如下：

```
java.lang.RuntimeException: Error while creating KieBase[Message [id=1, kieBase=rules, level=ERROR, path=conditional1.drl, line=25, column=0
    text=[ERR 102] Line 25:3 mismatched input 'when' in rule], Message [id=2, kieBase=rules, level=ERROR, path=conditional1.drl, line=0, column=0
    text=Parser returned a null Package]]
```

很显然上面的异常是因为规则没有指定名称，而关键字 when 无法作为名称，因此在此处抛出异常。

格式不正确导致的异常：

```
rule test
    when
     foo3:Object(
```

异常信息如下：

```
java.lang.RuntimeException: Error while creating KieBase[Message [id=1, kieBase=rules, level=ERROR, path=conditional1.drl, line=0, column=0
    text=Line 26:16 unexpected exception at input '<eof>'. Exception: java.lang.NullPointerException. Stack trace:
  java.lang.NullPointerException
```

其他异常信息就不在这里赘述了，实际应用中不断的学习总结即可根据错误信息快速定位问题所在。

4.11 关键字

从 Drools 5 开始引入了硬关键字和软关键字。硬关键字是保留关键字，在命名 demo 对象，属性，方法，函数和规则文本中使用的其他元素时，不能使用任何硬关键字。以下是必须避免的硬关键字：

（1）true
（2）false
（3）null

软关键词只在它们的上下文中被识别，可以在其他地方使用这些词，尽管如此，仍然建议避免它们，以避免混淆。其中大多数关键字我们在前面的章节中已经介绍过。软关键词列表如下：

（1）lock-on-active
（2）date-effective
（3）date-expires
（4）no-loop
（5）auto-focus
（6）activation-group
（7）agenda-group
（8）ruleflow-group
（9）entry-point
（10）duration
（11）package
（12）import
（13）dialect
（14）salience
（15）enabled
（16）attributes
（17）rule
（18）extend
（19）when
（20）then
（21）template
（22）query
（23）declare
（24）function
（25）global
（26）eval
（27）not
（28）in
（29）or
（30）and
（31）0exists
（32）forall

（33）accumulate
（34）collect
（35）from
（36）action
（37）reverse
（38）result
（39）end
（40）over
（31）init

5 与 Springboot 集成

本篇主要介绍一下 Drools 与 Springboot 的集成使用方法，也是具体实践的一部分。具体的集成步骤不做过多介绍，阅读代码基本上可以了解全部。详细代码参考 github：https://github.com/secbr/drools

pom 文件

引入了 springboot 和 drools 的依赖，同时引入了 kie-spring 的集成依赖。

```
<dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <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>
        <dependency>
            <groupId>org.drools</groupId>
            <artifactId>drools-compiler</artifactId>
            <version>${drools.version}</version>
        </dependency>
    </dependencies>
```

配置类

基于 springboot 的初始化配置：

```
@Configuration
public class DroolsAutoConfiguration {

    private static final String RULES_PATH = "rules/";

    @Bean
    @ConditionalOnMissingBean(KieFileSystem.class)
    public KieFileSystem kieFileSystem() throws IOException {
        KieFileSystem kieFileSystem =
getKieServices().newKieFileSystem();
        for (Resource file : getRuleFiles()) {

kieFileSystem.write(ResourceFactory.newClassPathResource(RULES_P
ATH + 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(new KieModule() {
            public ReleaseId getReleaseId() {
                return kieRepository.getDefaultReleaseId();
            }
        });

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

        return
getKieServices().newKieContainer(kieRepository.getDefaultRelease
Id());
    }

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

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

    @Bean
    @ConditionalOnMissingBean(KieSession.class)
    public KieSession kieSession() throws IOException {
        return kieContainer().newKieSession();
    }

    @Bean

@ConditionalOnMissingBean(KModuleBeanFactoryPostProcessor.class)
    public KModuleBeanFactoryPostProcessor kiePostProcessor() {
        return new KModuleBeanFactoryPostProcessor();
    }
}
```

Springboot 启动类

```
@SpringBootApplication
public class SpringBootDroolsApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringBootDroolsApplication.class,
args);
    }
}
```

需注意此类的放置位置，详情可参考 Springboot 相关使用教程。

实体类

```
public class Address {

    private String postcode;

    private String street;

    private String state;
    // 省略 getter/setter
}
```

规则返回结果类

```
public class AddressCheckResult {

    private boolean postCodeResult = false; // true:通过校验;
false: 未通过校验
    // 省略 getter/setter
}
```

规则文件

```
package plausibcheck.adress

import com.secbro.model.Address;
import com.secbro.model.fact.AddressCheckResult;
```

```
rule "Postcode should be filled with exactly 5 numbers"

    when
        address : Address(postcode != null, postcode matches
"([0-9]{5})")
        checkResult : AddressCheckResult();
    then
        checkResult.setPostCodeResult(true);
        System.out.println("规则中打印日志：校验通过!");
end
```

测试 Controller

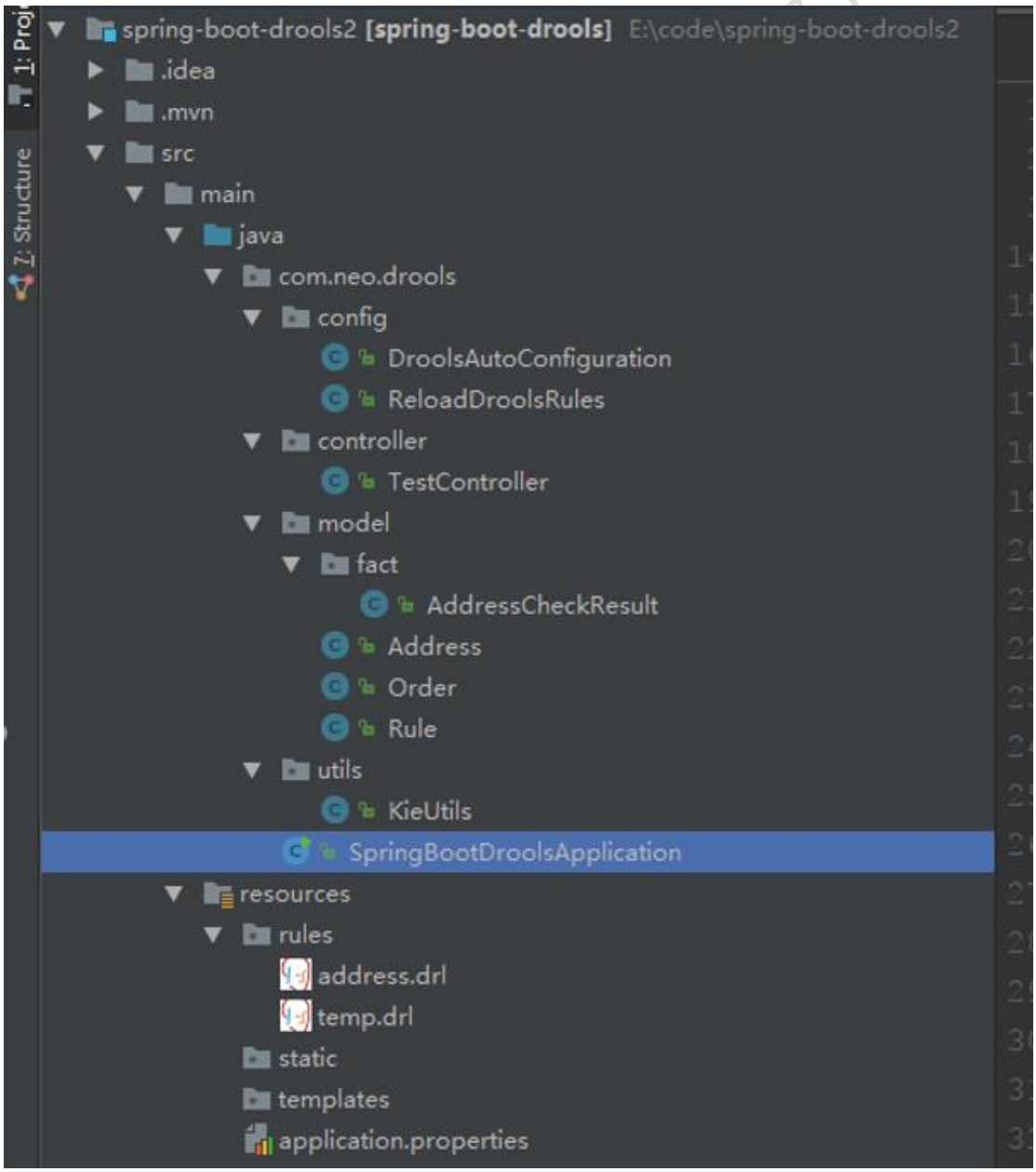
```
@RequestMapping("/test")
@Controller
public class TestController {

    @Resource
    private KieSession kieSession;

    @ResponseBody
    @RequestMapping("/address")
    public void test(){
        Address address = new Address();
        address.setPostcode("99425");

        AddressCheckResult result = new AddressCheckResult();
        kieSession.insert(address);
        kieSession.insert(result);
        int ruleFiredCount = kieSession.fireAllRules();
        System.out.println("触发了" + ruleFiredCount + "条规则");

        if(result.isPostCodeResult()){
            System.out.println("规则校验通过");
        }

    }
}
```

启动 Springboot 项目之后，默认访问 http://localhost:8080/test/address 即可触发规则。

6 基于 Springboot 动态加载规则实例

在于 Springboot 集成的章节，我们介绍了怎样将 Drools 与 Springboot 进行集成，本章将介绍集成之后，如何实现从数据库读取规则并重新加载规则的简单 demo。因本章重点介绍的是 Drools 相关操作的 API，所以将查询数据库部分的操作省略，直接使用数据库查询出的规则代码来进行规则的重新加载。另外，此示例采用访问一个 http 请求来进行重新加载，根据实际需要可考虑定时任务进行加载等扩展方式。最终的使用还是要结合具体业务场景来进行分析扩展。详细代码参考 github：https://github.com/secbr/drools。

整体项目结构图



上图是基于 intellij maven 的项目结构。其中涉及到 springboot 的 Drools 集成配置类，重新加载规则类。一些实体类和工具类。下面会抽出比较重要的类进行分析说明。

DroolsAutoConfiguration

```
@Configuration
public class DroolsAutoConfiguration {

    private static final String RULES_PATH = "rules/";

    @Bean
    @ConditionalOnMissingBean(KieFileSystem.class)
    public KieFileSystem kieFileSystem() throws IOException {
        KieFileSystem kieFileSystem =
getKieServices().newKieFileSystem();
        for (Resource file : getRuleFiles()) {

kieFileSystem.write(ResourceFactory.newClassPathResource(RULES_P
ATH + 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(new KieModule() {
            public ReleaseId getReleaseId() {
                return kieRepository.getDefaultReleaseId();
            }
        });
```

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

        KieContainer kieContainer =
getKieServices().newKieContainer(kieRepository.getDefaultRelease
Id());
        KieUtils.setKieContainer(kieContainer);
        return
getKieServices().newKieContainer(kieRepository.getDefaultRelease
Id());
    }

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

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

    @Bean
    @ConditionalOnMissingBean(KieSession.class)
    public KieSession kieSession() throws IOException {
        KieSession kieSession = kieContainer().newKieSession();
        KieUtils.setKieSession(kieSession);
        return kieSession;
    }

    @Bean

@ConditionalOnMissingBean(KModuleBeanFactoryPostProcessor.class)
    public KModuleBeanFactoryPostProcessor kiePostProcessor() {
        return new KModuleBeanFactoryPostProcessor();
    }
}
```

此类主要用来初始化 Drools 的配置，其中需要注意的是对 KieContainer 和 KieSession 的初始化之后都将其设置到 KieUtils 类中。

KieUtils

KieUtils 类中存储了对应的静态方法和静态属性，供其他使用的地方获取和更新。

```
public class KieUtils {

    private static KieContainer kieContainer;

    private static KieSession kieSession;

    public static KieContainer getKieContainer() {
        return kieContainer;
    }

    public static void setKieContainer(KieContainer
kieContainer) {
        KieUtils.kieContainer = kieContainer;
        kieSession = kieContainer.newKieSession();
    }

    public static KieSession getKieSession() {
        return kieSession;
    }

    public static void setKieSession(KieSession kieSession) {
        KieUtils.kieSession = kieSession;
    }
}
```

ReloadDroolsRules

提供了 reload 规则的方法，也是本章节的重点之一，其中从数据库读取的规则代码直接用字符串代替，读者可自行进行替换为数据库操作。

```
@Service
public class ReloadDroolsRules {

    public void reload() throws UnsupportedEncodingException {
        KieServices kieServices = KieServices.Factory.get();
        KieFileSystem kfs = kieServices.newKieFileSystem();
        kfs.write("src/main/resources/rules/temp.drl",
loadRules());
        KieBuilder kieBuilder =
kieServices.newKieBuilder(kfs).buildAll();
```

```
        Results results = kieBuilder.getResults();
        if (results.hasMessages(Message.Level.ERROR)) {
            System.out.println(results.getMessages());
            throw new IllegalStateException("### errors ###");
        }


KieUtils.setKieContainer(kieServices.newKieContainer(getKieServi
ces().getRepository().getDefaultReleaseId()));
        System.out.println("新规则重载成功");
    }

    private String loadRules() {
        // 从数据库加载的规则
        return "package plausibcheck.adress\n\n import
com.neo.drools.model.Address;\n import
com.neo.drools.model.fact.AddressCheckResult;\n\n rule
\"Postcode 6 numbers\"\n\n    when\n  then\n
System.out.println(\"规则 2 中打印日志：校验通过!\");\n end";

    }

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

}
```

TestController

提供了验证入口和 reload 入口。

```
@RequestMapping("/test")
@Controller
public class TestController {


    @Resource
    private ReloadDroolsRules rules;

    @ResponseBody
    @RequestMapping("/address")
    public void test(){
```

```
        KieSession kieSession = KieUtils.getKieSession();

        Address address = new Address();
        address.setPostcode("994251");

        AddressCheckResult result = new AddressCheckResult();
        kieSession.insert(address);
        kieSession.insert(result);
        int ruleFiredCount = kieSession.fireAllRules();
        System.out.println("触发了" + ruleFiredCount + "条规则");

        if(result.isPostCodeResult()){
            System.out.println("规则校验通过");
        }

    }

    @ResponseBody
    @RequestMapping("/reload")
    public String reload() throws IOException {
        rules.reload();
        return "ok";
    }
}
```

其他

其他类和文件内容参考 springboot 集成部分或 demo 源代码。操作步骤如下：启动项目访问 http://localhost:8080/test/address 会首先触发默认加载的 address.drl 中的规则。当调用 reload 之后，再次调用 address 方法会发现触发的规则已经变成重新加载的规则了。

CSDN demo 下载地址：http://download.csdn.net/detail/wo541075754/9918297

7 应用实例集合

7.1 相同对象 and List 使用

本节介绍一下怎么实现两个相同对象的插入和比较。向 session 中 insert 两个相同的对象，但对象的参数值有不同的地方，同时要求对两个 FACT 对象的属性进行判断，当同时满足（&&）时，通过规则校验，进行后续业务处理。下面，通过两种方式来实现此功能。

方式一

规则文件内容：

```
package com.rules

import com.secbro.drools.model.Customer;

rule "two same objects"
    agenda-group "two same objects"
    when
        $firstCustomer:Customer(age == 59);
        $secondCustomer:Customer(this != $firstCustomer,age ==
61);
    then
        System.out.println("firstCustomer age :" +
$firstCustomer.getAge());
        System.out.println("secondCustomer age :" +
$secondCustomer.getAge());
    end
```

测试端调用部分代码：

```
@Test
    public void testSameObjects() {
        KieSession kieSession = getKieSession("two same
objects");

        Customer customer = new Customer();
        customer.setAge(61);
        kieSession.insert(customer);

        Customer customer1 = new Customer();
        customer1.setAge(59);
        kieSession.insert(customer1);

        int count = kieSession.fireAllRules();

        kieSession.dispose();
        System.out.println("Fire " + count + " rules!");
    }
```

如此，则实现了上面场景的内容。值得注意的是规则文件中 this != $firstCustomer 的写法，此处可以排除两个对象属性相同导致的问题。

方法二

此方式采用 List 来传递两个相同的参数，规则文件内容如下：

```
package com.rules

import com.secbro.drools.model.Customer;
import java.util.List;

rule "two same objects in list"
    agenda-group "two same objects in list"
    when
        $list : List();
        $firstCustomer:Customer(age == 59) from $list;
        $secondCustomer:Customer(this != $firstCustomer,age ==
61) from $list;
    then
        System.out.println("two same objects in
list:firstCustomer age :" + $firstCustomer.getAge());
        System.out.println("two same objects in
list:secondCustomer age :" + $secondCustomer.getAge());
    end
```

测试类部分代码：

```
@Test
    public void testSameObjectsInList() {
        KieSession kieSession = getKieSession("two same objects
in list");

        List<Customer> list = new ArrayList<>();
        Customer customer = new Customer();
        customer.setAge(61);
        list.add(customer);

        Customer customer1 = new Customer();
        customer1.setAge(59);
        list.add(customer1);
        kieSession.insert(list);

        int count = kieSession.fireAllRules();

        kieSession.dispose();
        System.out.println("Fire " + count + " rules!");
    }
```

7.2 获取规则名称和包名

如果我执行了很多规则，调用 fireAllRules 方法只会返回触发了几条规则，那么我怎么知道哪些规则被触发了，并把这些触发的规则的名称存如数据库呢？

在前面的 RHS 语法章节中我们已经讲过预定义变量 drools 的简单实用，其实通过它就可以轻松的拿到规则相关的信息。下面看实例：

规则内容如下：

```
package com.rules

rule "Get name and package demo"

agenda-group "Name and package"

when
then
    System.out.println("The rule's name is '" +
drools.getRule().getName() + "'");
    System.out.println("The rule's package is  '" +
drools.getRule().getPackageName() + "'");
end
```

执行规则代码如下：

```
@Test
    public void test(){
        KieSession kieSession = this.getKieSession("Name and
package");
        int count = kieSession.fireAllRules();
        kieSession.dispose();

        System.out.println("Fire " + count + " rule(s)!");

    }
```

执行结果：

```
The rule's name is 'Get name and package demo'
The rule's package is  'com.rules'
Fire 1 rule(s)!
```

8 异常问题汇总

http://blog.csdn.net/wo541075754/article/details/75089038
http://blog.csdn.net/wo541075754/article/details/75089501

编者寄语：

此系列课程持续更新中，QQ 群：593177274，欢迎大家加入讨论。点击链接关注《Drools 博客专栏》。由于 Drools 资料较少，教程编写不易，每篇博客都是作者亲身实践并编写 demo。如果对你有帮助也欢迎赞赏（微信）！ 这也是对原创的最大支持！