```
}
```

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。因为声明一次只能绑定一个 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()
```

pattemType 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：

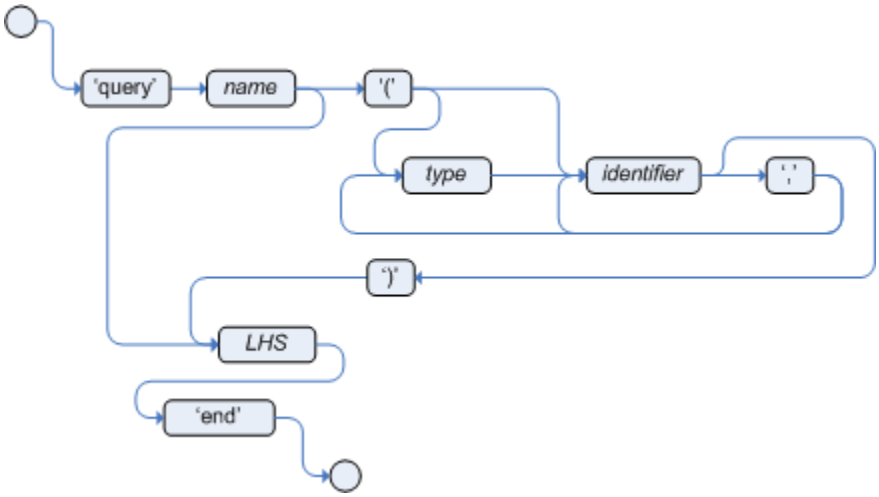
```
modify ( <fact-expression> ) {
    <expression> [ , <expression> ]*
}
```

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)!");
```

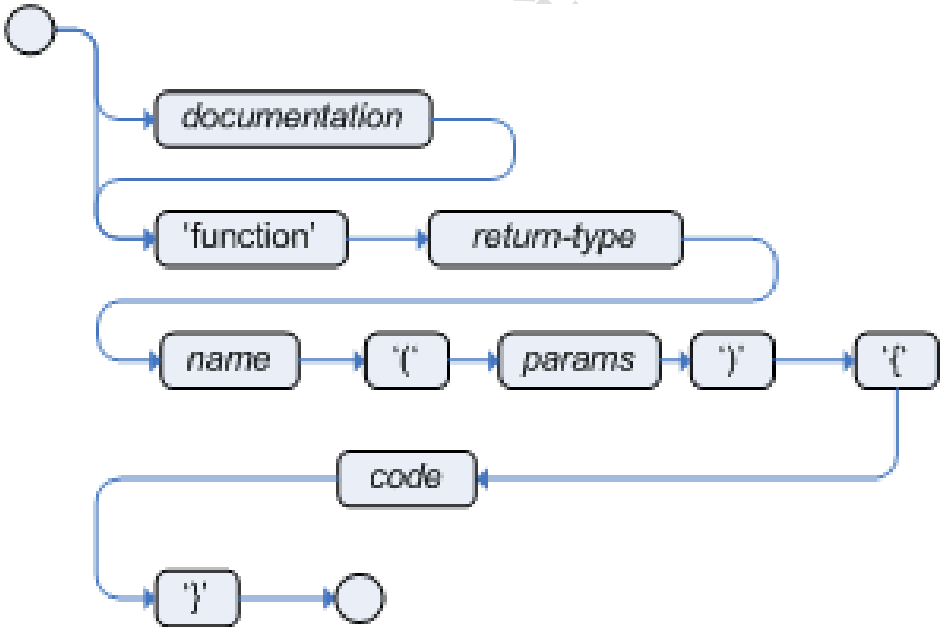
```
            QueryResults results = kieSession.getQueryResults("query-2","Tom");
            System.out.println("results size is " + results.size());
            for(QueryResultsRow row : results){
                Person person = (Person) row.get("$person");
                System.out.println("Person 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 Function 函数



函数是将语义代码放置在规则文件中的一种方式，就相当于 java 类中的方法一样。函数并不会比辅助类做更多的事情，实际上，编译器会在幕后生成助手类。使用函数的好处是可以将业务逻辑集中放置在一个地方，根据需要可以对函数进行修改。但它既有好处也有坏处。函数对于调用规则的后果部分操作是最有用处的，特别是只有参数变化但执行的操作完全相同时。这里的函数可以对照 java 中方法的抽取封装来理解。

典型的函数声明如下所示：

```
function String hello(String name) {
```

```
    return "Hello "+name+"!";
}
```

实例规则代码如下：

```
package com.rules

function String hello(String name){
    return "Hello " + name + "!";
}

rule helloSomeone

    agenda-group "function-group"

    when
        eval(true);
    then
        System.out.println(hello("Tom"));
    end
```

测试代码如下：

```
@Test
    public void testFunction(){
        KieSession kieSession = this.getKieSession("function-group");
        int count = kieSession.fireAllRules();
        kieSession.dispose();
        System.out.println("Fire " + count + " rule(s)!");
    }
```

执行结果：

```
Hello Tom!
Fire 1 rule(s)!
```

需要注意的是，function 虽然不是 java 的一部分，但是依然可以在这里使用。函数的参数根据需要可以有一个，也可以有多个，也可以没有。返回结果的类型定义和正常的 java 语法没有区别。

前面我们已经讲过如何引入 java 中的静态方法，此处的 function 也可以用静态方法来代替，具体使用参考相关章节，这里就不再赘述。

4.9 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。

4.10 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。

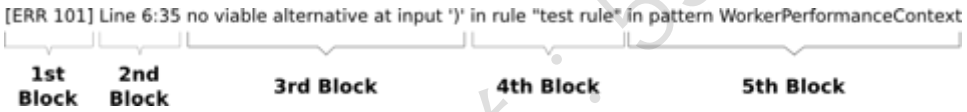
多行注释与 Java 相同，采用“/*注释内容*/”，来进行注释，示例如下：

```
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
```

4.11 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。

4.12 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

4.13 元数据

4.14 DSL

http://blog.csdn.net/u012373815/article/details/53837345

4.15 规则流

5 session 使用说明

KieSession 是用来与规则引擎进行交互的会话。在 Drools 7 当中分有状态的 session 和无状态的 session：KieSession 和 StatelessKieSession。

5.1 有状态 session

通过 KieContainer 可以获取 KieSession，在 kmodule.xml 配置文件中如果不指定 ksession 的 type 默认也是有状态的 session。有状态 session 的特性是，我们可以通过建立一次 session 完成多次与规则引擎之间的交互，在没有调用 dispose 方法时，会维持会话状态。使用 KieSession 的一般步骤为，获取 session，insert Fact 对象，然后调用 fireAllRules 进行规则匹配，随后调用 dispose 方法关闭 session。

5.2 无状态 session

StatelessKieSession 提供了一个更加便利的 API，是对 KisSession 的封装，不再调用 dispose 方法进行 session 的关闭。它隔离了每次与规则引擎的交互，不会再去维护会话的状态。同时也不再提供 fireAllRules 方法。

使用场景：

（1）数据校验

（2）运算

（3）数据过滤

（4）消息路由

（5）任何能被描述成函数或公式的规则

具体示例：

规则代码：

```
package com.stateless
import com.secbro.drools.model.Person
rule "test-stateless"

when
    $p : Person()
then
    System.out.println($p.getAge());
 end
```

测试代码：

```
public void testStateLessSession(){
        StatelessKieSession kieSession = this.getStatelessKieSession();
```

```
            List<Person> list = new ArrayList<>();

            Person p = new Person();
            p.setAge(35);
            list.add(p);
            Person p1 = new Person();
            p1.setAge(20);
            list.add(p1);
//              kieSession.execute(p);
            kieSession.execute(list);
      }
protected StatelessKieSession getStatelessKieSession(){
            KieServices kieServices = KieServices.get();
            KieContainer kieContainer = kieServices.getKieClasspathContainer();
            StatelessKieSession kieSession = kieContainer.newStatelessKieSession("stateless-
rules");

            return kieSession;
      }

      protected StatelessKieSession getStatelessKieSession(String agendaGroupName){
            StatelessKieSession kieSession    = getStatelessKieSession();
            return kieSession;
      }
```

6 Logging

7 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();
```

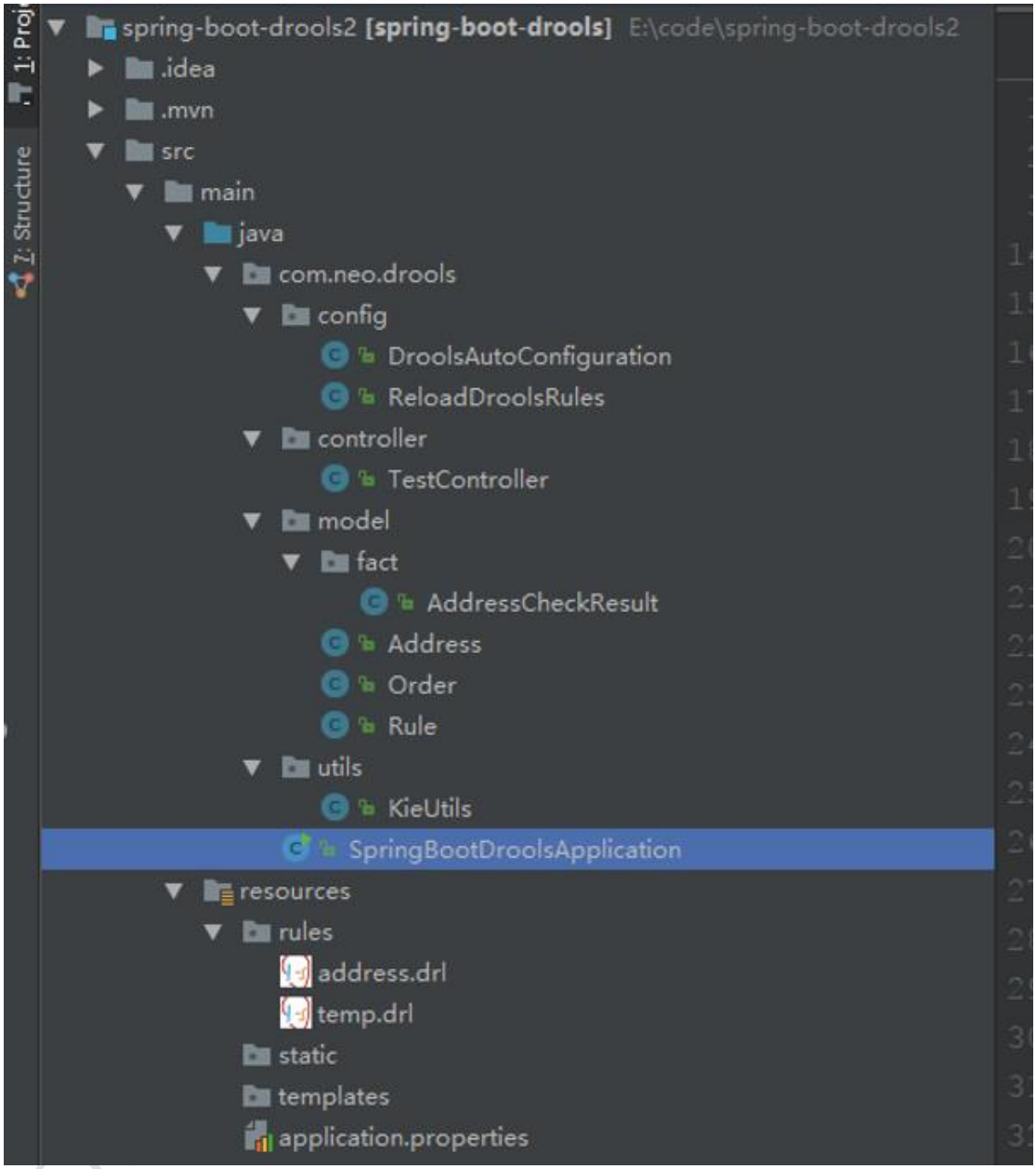
```
        address.setPostcode("99425");

        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。

8 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.getDefaultReleaseId());
    }

    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

9 应用实例集合

9.1 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!");
    }
```

9.2 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)!
```

10 kie-maven-plugin

```
  <packaging>kjar</packaging>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.kie</groupId>
        <artifactId>kie-maven-plugin</artifactId>
        <version>7.0.0.Final</version>
        <extensions>true</extensions>
      </plugin>
    </plugins>
  </build>
```

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！