西南交通大学

硕士学位论文

drools规则引擎模式匹配效率优化研究及实现

姓名：刘金龙

申请学位级别：硕士

专业：计算机应用技术

指导教师：楼新远

20070501

摘 要

drools 规则引擎是一款开源的规则引擎项目，它支持 jsr94 规范，用面向对象方法实现了 Rete 模式匹配算法，Rete 算法是一个的高效的模式匹配算法，这是经过实践证明的，但是这种高效是以消耗大量内存为代价的，即 Rete 算法记录了所有匹配的中间结果，也就是创建了所有断言的所有元组，将所有的元组保留到缓存中，每声明/撤销/修改一个工作内存中的事实对象时，都要对相关联的元组进行更新和传播，这会对 drools 规则引擎在性能上产生负的影响。

本论文针对 Rete 算法的这一不足，将处理事实对象的过程采用 Leaps 算法进行改进，Leaps 算法的优点就是不会创建所有断言的元组，它采取了一种折衷的方案，规则引擎在匹对简单的条件约束时不会占用太多资源，所以不会创建元组，即仅仅在必要的时候才创建元组，也就是仅仅存在 not 和 exist 条件时才创建元组，缓存匹配的中间结果，当规则引擎操作工作内存中的事实对象时，维护并传播这些元组，这就是 Lazy 条件评估策略。这样，从理论上是可以提高 drools 规则引擎的模式匹配算法的效率，并降低内存使用。

本设计首先比较 Rete 算法和 Leaps 算法的算法模型，然后分析 Rete 算法的不足和 Leaps 算法的折衷方案，最后给出 Leaps 算法的一个设计实现并给出证明。目前应用 rete 算法的产品比较多，例如 CLIPS，Jess，Eclipse 和 OPS5 等，如果 leaps 算法在优化 Rete 算法中取得成功，那么将对众多应用 Rete 算法的产品产生重大影响。

关键词：规则引擎，drools，rete，leaps

Abstract

Drools is an open-source rule engine,which supports the jsr94 specification and performs the matching algorithm of the Rete pattern with the object-oriented(OO) method. Experimental results have shown that Rete algorithm is a highly effective matching-pattern,but its high efficiency is at the cost of a mass of memory. That is to say, Rete algorithm remembers all intermediate results by recording all tuples of all predicate that are buffered in the memory,and the associating tuples are updated and propageted when a fact object is asserted into working memory or is retracted/deleted from working memory, which has a negative effect on effectivity of drools.

Aimming at the deficiency of the Rete algorithm, this paper uses the Leaps algorithm to improve the operation process of the objects. Leap algorithm does not materialize all the tuples, which is a main excellency, and this algorithm takes a compromised way.Engine does not cost many resources when matching the simple condition restriction, so engine need not materialize the tuples only when needed on the condition of including not or exist condition in a rule in order to memory intermediate results when they are updated and propagated, which is the Lazy condition evaluation strategy.In this way,it can enhance the efficiency of the algorithm and reduce the usage of memory theoretically.

This paper compares the algorithm models between Rete and Leaps, then analyses insufficiency of Rete and introduces the compromised method of Leaps，finally， provides the design and implementation. At present, there are many products based on Rete algorithm, for example, CLIPS, Jess, Eclipse and OPS5, which are influenced deeply if Leaps algorithm is proved to be efficient.

Key Words: Rule Engine, Algorithm, drools, Rete, Leaps

第一章 绪 论

规则引擎[1]自从出现以来就肩负着重要的任务，如软件管理流程自动化，软件系统需依据业务规则的变化而快速低成本的变化，业务人员直接管理软件业务规则，而不需要开发人员的参与，同时规则引擎也肩负着让开发人员能应对开发需求快速变化的要求，规则引擎这些方面的目标已经或者大部分已实现，但是，规则引擎的本身固有的问题却越来越明显的暴露出来，它就是效率问题[2]。

1.1 现实问题

当前大部分企业级软件开发，如果业务规则非常复杂，不可避免的会用到规则引擎中间件，目前商业规则引擎最成熟是，功能强大的要数被光大银行采用的 ILOG 公司的 Jrules 规则引擎[3][4]，而大部分规则引擎产品为了提高效率，对于规则和事实的模式匹配都是在缓存中进行的，要在内存中保存并处理大量的中间数据，这就对于规则引擎服务器的硬件提出更好的要求。这种情况下，企业规则依旧大量的加入的话，难免对硬件的要求成了规则引擎效率的一个瓶颈。所以在现有的硬件条件下，优化模式匹配的算法以提高效率成了一个规则引擎产品开发商共同探讨的问题。

目前大部分规则引擎产品都是采用了一种被认为效率高的模式匹配算法，Rete 算法[5]，如果硬件条件不改变，而采用的模式匹配算法一样，那么规则引擎产品的效率不会有明显的改善，所以一直有规则引擎厂商优化 Rete 算法，以期望能提高模式匹配效率，现在大部分规则引擎产品都不是源 Rete 算法，而是 Rete 算法改进版本。

Rete 算法将业务规则对象在缓存中构建了一个 Rete 网络，这样本意是可以降低事实对象进入 Rete 网络的进行的模式匹配的次数，在业务规则和事实对象发生完全匹配时将相应的规则放置到议程上，等到所有事实对象匹配完毕后，应用程序向规则引擎发出执行消息，规则引擎开始运行议程上的规则。这样，在整个匹配的过程中，所有的断言的所有的元组都被创建和传播，由于工作内存中的对象发生变化时，都会造成缓存中的元组的

更新，所以对整个规则引擎来说会有一个负的影响。而 Leaps 算法在事实和规则进行模式匹配时采用了一种“lazy”的条件评估策略，即不会创建所有断言的元组，它采取了一种折衷的方案，规则引擎在匹配简单的条件约束时不会占用资源太多资源，所以不会创建元组，即仅仅在必要的时候才创建元组[6]，也就是仅仅存在 not 和 exist 条件时才创建元组[7]，缓存匹配的中间结果，当工作内存中的事实对象发生变化时，维护并传播这些元组，这就是 Lazy 条件评估策略。从理论上讲，Leaps 算法能够提高模式匹配的效率和降低内存使用。

1.2 基于规则专家系统

规则引擎起源于基于规则的专家系统[8](Rule Based Expert System 缩写为 RBES)，RBES 是人工智能的一个分支，它模仿人类的推理方式，使用试探性的方法进行推理，并使用人类能理解的术语解释和证明它的推理结论。

RBES 主要包括三部分：规则库（Rule Base 或者 Knowledge Base）、工作内存（Working Memory 或者称为事实库 Fact Base）和 Inference Engine（推理引擎）。

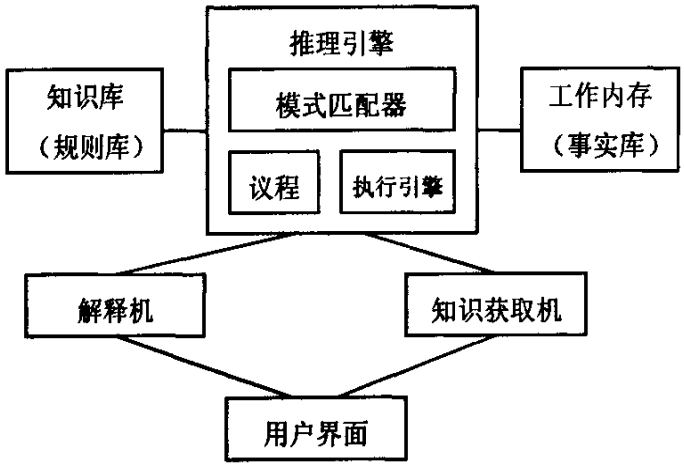

图1-1 基于规则的专家系统结构示意图

基于规则的专家系统结构示意图如图 1-1 所示，推理引擎主要包括三部分：模式匹配器(Pattern Matcher)、议程(Agenda)和执行引擎(Execution

Engine)。模式匹配器决定何时执行哪个规则；议程管理模式匹配器挑选出来的规则的执行次序；执行引擎负责执行规则和其他动作。

推理引擎通过决定哪些规则满足事实或目标，并授予规则优先级，满足事实或目标的规则被加入议程，议程决定哪些规则将被执行。推理引擎从议程中取得一条规则，并且从工作记忆中取得相关的事实对象，执行该条规则结论，当该规则结论执行完毕，规则引擎将从议程中取得第二条规则执行，直到议程中没有规则为止。

存在两种推理方式：演绎法和归纳法。演绎法即正向链式（Forward Chaining）推理，从一个初始的事实出发，不断地应用规则得出结论（或执行指定的动作）。而归纳法即是反向链式（Backward Chaining）推理，则是从假设出发，不断地寻找符合假设的事实。

1.3 演绎推理

知识推理方法即是知识触发而产生新的事实或者结论的模型方法。例如：所有的商品都有使用价值，且棉花是商品，所以棉花有使用价值。本节将介绍两种推理方法：演绎推理[9]和归纳推理[10]。

演绎推理，也被称作正向推理，是从一般性的原理原则中推出个别事物的结论，其思维过程是从一般到个别，演绎推理的结论，原则上不会超出前提的范围，演绎推理的结论和前提的联系是必然的，只要前提真实，推理形式正确，则结论一定可靠。演绎推理符合正常思维过程，所以演绎推理在业务规则中首先得到应用。

Rete 算法是 Charles Forgy 博士在 1978-1979 年发明的，1982 年公开发表[11]。Rete 算法是当前被证明了一个高效的演绎推理算法[12]，已在几款专家系统中得到了应用，例如 clips，mycin 等。Leaps 算法也是一个演绎推理的著名的算法。

演绎推理是由普通性的前提推出特殊性结论的推理。演绎推理有三段论、假言推理和选言推理等形式，在 drools 规则引擎中应用的是三段论，下面针对三段论作专门介绍。

三段论[13]是指由两个简单判断作前提和一个简单判断作结论组成的演绎推理。三段论中三个简单判断只包含三个不同的概念，每个概念都重复出现一次。这三个概念都有专门名称：结论中的宾词叫“大词”，结论中的

主词叫“小词”，结论不出现的那个概念叫“中词”，在两个前提中，包含大词的叫“大前提”，包含小词的叫“小前提”。例如：

大前提：科研型大学应以科研为主。

小前提：西南交大是一所科研型大学。

结论：西南交大应以科研为主。

三段论的格就是由于中项在前提中的位置不同所决定的三段论的形式。

第一格：中项在大前提中是主项，在小前提中是谓项。

第二格：中项在两个前提中都是谓项。

第三格：中项在两个前提中都是主项。

第四格：中项在大前提中是谓项，在小前提中是主项。

三段论的规则是检验三段论有效性的标准，也是正确进行三段论推理的依据。

与演绎推理相对的就是归纳推理，所谓归纳推理，就是从个别性知识推出一般性结论的推理。归纳推理分为完全归纳推理和不完全归纳推理，完全归纳推理是根据某类的每一个对象具有（或不具有）某种属性，推出一个关于某类的一般性知识的结论的归纳推理。

不完全归纳推理有分为两种：简单枚举法和科学归纳法。简单枚举法它是依据经验认识，根据某种属性在部分同类对象中的不断重复而没有遇到反例，从而推出该类的所有对象都具有某种属性。科学归纳法以科学分析为主要依据，依据某类事物中部分对象与其属性之间具有因果关系，推出该类事物的全部对象都具有某种属性的归纳推理。

论文中涉及到的推理方法，如果没有特别指明，均指演绎推理中的三段论式推理。

1.4 推理过程

推理引擎结构示意图如图 1-2 所示：

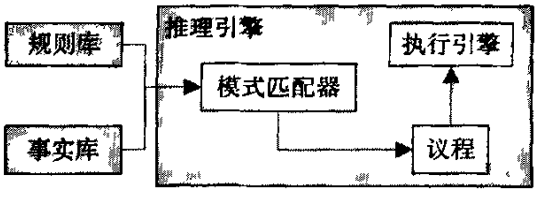

图 1-2 推理引擎结构示意图

规则引擎的推理步骤[14]如下：

a. 将初始事实输入至工作内存。

b. 使用模式匹配器将规则库中的规则和事实比较。

c. 如果执行规则存在冲突，即同时激活了多个规则，将冲突的规则放入冲突集合。

d. 解决冲突规则集，将激活的规则按顺序放入议程。

e. 执行议程中的规则。

f. 重复步骤b至e，直到执行完毕议程中的所有规则。

任何一个规则引擎都需要很好地解决规则的推理机制和规则条件匹配的效率问题。

当引擎执行时，会根据规则执行队列中的优先顺序逐条执行规则实例，由于规则的执行部分可能会改变工作寄存器中的数据对象，从而会使队列中的某些规则实例因为条件改变而失效，必须从队列中撤销，也可能会激活原来不满足条件的规则，生成新的规则实例进入队列。于是就产生了一种“动态”的规则执行链，形成规则的推理机制。这种规则的“链式”反应完全是由工作寄存器中的数据驱动的。

规则条件匹配的效率决定了引擎的性能，引擎需要迅速评估工作记忆中的数据对象，从加载的规则集中发现符合条件的规则，生成规则执行实例。Charles L. Forgy发明的Rete算法，很好地解决了这方面的问题。

1.5 规则引擎

规则引擎起源于基于规则的专家系统，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，并根据规则做出业务决策[15]。

规则引擎包括规则库，工作内存，模式匹配器，议程等模块，规则引擎结构示意图如图1-3所示。

一个开放的业务规则引擎应该可以潜入在应用程序的任何位置，不同位置的规则引擎使用不同的规则集，用于处理不同的应用程序数据对象[16]。

当规则引擎执行时，会依据议程中的规则队列中的优先顺序，逐条执行规则实例，由于规则的执行可能会导致工作内存中的事实对象发生改变，

或添加或删除，从而导致议程队列中的规则因为条件改变而失效或者其条件得到完全满足而被进入议程队列，于是就产生了一个“动态”的规则执行链，形成规则的推理机制，这种规则的链式反应完全是由工作内存中的数据驱动的[17]。

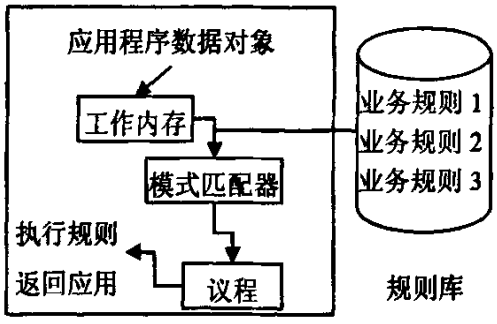

图 1-3 规则引擎结构示意

1.6 研究现状

算法是规则引擎的核心部分，其中 Rete 算法是目前为止应用最广的算法，基于规则的语言大多数都使用了 Rete 算法，如 CLIPS、Eclipse、Jess 和 OPS5，Rete 算法将事实和规则中的模式相竞争，以确定哪些规则满足了它们的条件。Rete 算法将匹配的过程记录下来，在每一次更新了事实对象，仅仅需要记住与该事实对象相关联的规则的更新即可。

目前研究 Rete 算法优化的思路很多，例如，优化 Rete 网络的编译模式[18]，优化事实存取结构以提高删除事实时效率[19][20]等。

Rete 最主要的缺点是内存使用量大，将所有的事实和所有的模式进行简单的比较不会使用大量内存，存储使用了模式匹配和部分模式匹配的系统的状态消耗的大量的内存。

Leaps 算法，Leaps 算法是专门用于产生式系统，用一个“Lazy”的方法进行条件评估[21]。古典 Leaps 算法[22]将所有声明进入引擎的事实对象按照进入引擎的顺序存储到一个主堆栈中，引擎将依次弹出事实对象进行模式匹配以找到相关规则，一旦找到相匹配的规则，系统将记住该位置以备稍后继续迭代，并执行该规则。该规则的执行一旦完成，系统就会回到刚刚记住的位置继续进行匹配，当一个对象进行模式匹配完成后，就会从主

堆栈中弹出。系统将会弹出堆栈的第二个对象作为主对象，继续进行模式匹配，直到堆栈为空为止。

从理论上讲，Leaps 模式匹配算法在处理工作内存中的事实对象时是一种效率高的算法。

除了 Rete 算法和 Leaps 算法之外，还有其他一些模式匹配算法，例如：KMP 算法，该算法是 D.E.Knuth 、J.H.Morris 和 V.R.Pratt 同时发现的，因此被称为 KMP 算法。DFSA 算法，这是一种基于有限状态自动机的多模式匹配算法。还有其他一些算法，这里就不做介绍了。

如果说算法是核心思想的话，那么产品就是算法的表现形式，当前各种基于规则的语言和基于规则的中间件产品，都试图找到一种效率最高的算法或者算法混合体，这些产品比较有影响的有：Drools、Mandarax、Jlisa、ILOG 公司的 JRules 等。

Drools 框架是一个 Bob McWhirter 开发的开源项目[23]，它是用 Java 语言编写的一款基于演绎推理的规则引擎，使用 Rete 算法对所编写的规则求值，实现了 JSR94 Rule Engine API 并提供了单元测试代码[24]。Drools 允许使用声明方式表达业务逻辑，也可以使用 Java/XML 语法编写规则，可以将 Java 代码直接嵌入规则文件中，从内部机制上讲，它使用了和 Forgy 算法相同的概念和方法，但是增加了可与面向对象语言无缝连接的节点类型。

JLisa 是一个利用 java 构建业务规则的强大的开源框架[25]。它和 Drools 是支持 JSR94 的开源项目。

Mandarax 是基于反向推理(归纳法)一款开源规则引擎，能够较容易地实现多个数据源的集成。例如，数据库记录能方便地集成为事实集(facts set)，reflection 用来集成对象模型中的功能。目前不支持 JSR94。

ILOG 公司的 Jrules，是一款支持 JSR94 规范的商业项目，它是目前最为成熟，功能强大的产品。国内一些重要客户使用了该产品，例如太平人寿保险有限公司，光大银行等。

规则语言是规则引擎应用程序的重要组成部分,所有的业务规则都必须用某种语言定义并且存储于规则执行集中,从而规则引擎可以装载和处理它们。由于没有关于规则如何定义的公用规范,市场上大多数流行的规则引擎都各有其的规则语言，目前便有许多种规则语言正在应用，因此，当需要将应用移植到其他的 Java 规则引擎实现时，可能需要变换规则定义，如

将 Drools 私有的 DRL 规则语言转换成标准的 ruleML，Jess 规则语言转换成 ruleML 等。这个工作一般由 XSLT 转换器来完成，如表 1-1 所列举集中常用的规则语言。

除了算法和产品之外，能显现发展现状的就是规范了，例如 SUN 公司的 JSR94 规范。

JSR94 为规则引擎提供了公用标准 API,为实现规则管理 API 和运行时 API 提供了指导规范,并没有提供规则和动作该如何定义以及该用什么语言定义规则，也没有为规则引擎如何读和评价规则提供技术性指导，JSR94 规范将上述问题留给了规则引擎的厂商。

表 1-1 规则语言

Rule Markup language (RuleML)	http://www.ruleml.org/
Simple Rule Markup Language (SRML)	http://xml.coverpages.org/srml.html
Business Rules Markup Language (BRML)	http://xml.coverpages.org/brml.html
A Semantic Web Rule Language Combining OWL and RuleML (SWRL)	http://www.daml.org/2003/11/swrl/

1.7 研究内容

1. 研究当前规则引擎发展所面临的效率方面的现实问题，并分析了 Rete 算法在模式匹配方面的缺陷。
2. 研究 Leaps 算法的主要贡献是 Lazy 评估策略，得出结论：Leaps 的这一贡献可以弥补 Rete 算法的在模式匹配算法的缺陷。
3. 研究当前一款开源的规则引擎产品 drools 项目，并以 drools 为容器实现 Leaps 算法。
4. 给出 Leaps 算法的详细设计和实现。
5. 测试 Leaps 算法的性能并得出结论。

1.8 组织结构

全文共分五章：

第一章 绪论 介绍规则引擎发展所面临的效率方面的现实问题，以

及规则引擎的背景知识，包括基于规则的专家系统、推理方式、推理过程等，最后介绍当前的研究现状，和本论文的研究内容；

第二章 Rete 算法和 Leaps 算法 介绍了当前一款规则引擎产品 drools 项目规则引擎项目，和 drools 使用的模式匹配算法 Rete 算法和另外一种高效的模式匹配算法 Leaps 算法；

第三章 Leaps 算法功能设计 给出 Leaps 算法的详细设计；

第四章 Leaps 算法实现 给出 Leaps 算法的实现；

第五章 Leaps 算法性能测试和分析 选取并创建测试用例，进行 Rete 算法和 Leaps 算法在效率和内存方面的比较，并得出结论。

第二章 Rete算法和Leaps算法

2.1 drools规则引擎

drools是一款成功的开源的规则引擎项目，它采用了高效的模式匹配算法Rete算法，实现了逻辑与数据的分离，它采用的技术是当前流行的面向对象语言java和xml。

2.1.1 设计目标

drools是开源项目，其规划的是软件的目标而非过程，drools优势在于，它使得非常复杂的问题的解决方案简单化，简化软件需求变更的解决方案，并且提供一个方案解释其原因。drools本身规范了其如下目标。

1. 逻辑和数据的分离[26]

drools使得数据保存在域对象中，逻辑保存在规则中，这本身就打破了OO中关于数据和规则的耦合。这可能因祸得福，如果将逻辑规则集中部署在规则文件中，这可能会使得在不久的将来的变化更加容易得到满足。

2. 速度和效率

Rete算法，Leaps算法以及它们的派生算法，例如drools Reteoo和Leaps算法的实现，提供了非常高效的方式来匹配规则和域对象。对于规则不是完全改变时，drools规则引擎可以缓存中间结果，这样它们是更加高效的。

3. 业务规则集中部署

通过使用业务规则，可以建立一个可部署的规则库，这就是说这个规则库是一个业务规则库，理想情况下，业务规则就象文档一样容易阅读理解。

4. 集成开发工具

象eclipse这样的开发工具提供了支持drools规则引擎的插件，可以所见即所得的编辑规则，在不久的将来，还可以进行调试，验证等。

5. 业务规则可读性

通过创建问题域的对象模型，规则看上去非常象自然语言，这可使商业决策者在维护业务规则变得非常容易。

2.1.2 应用范围

用简单的话说就是，用传统的方法无法找到满意的解决方案时就可以考虑使用业务规则引擎，例如：

1. 用传统编码太过复杂。
2. 可能问题不是太复杂，但是找不到一种可伸缩性的方法来解决问题。
3. 不能用通用的方法来解决的问题。
4. 解决起来相对复杂的问题，或没有传统解决方案的问题，或不易理解的问题。
5. 业务逻辑可能是简单的，但是规则变化可能过于频繁的问题。

在其他一些的情况下，软件版本本身可能不是经常变化，但是使用业务规则引擎可以增加软件的灵活性。

非技术专家通常通晓业务规则，业务规则可以帮助他们用他们特有的方式表达业务逻辑需要，当然，这些通晓业务规则而技术不是很精湛的专家也必须要精密的思考。

通常情况下，在现在面向对象（OO）的应用中，将应用中的关键业务逻辑应用规则引擎，尤其是那些确实难以解决的逻辑部分。这是OO概念的一个颠倒，它将所有的业务逻辑封装到了对象中，这不是抛弃了OO思想，相反，在任何现实世界的应用中，业务逻辑只是应用程序的一部分。如果曾经注意到了“if”“else”“switch”和一些繁琐的逻辑，那么一定觉得在代码有这些东西而感觉非常不恰当。当回来修改的时候，要么请开发人员要花大力气理解这些逻辑，要么就犯错了，这时就需要业务规则。当遇到糟糕的问题，又没有算法规则和模式可以用时，这时也需要业务规则。

对于当前业务规则应用不是很成熟，但是应用中又使用了类似业务规则的应用，通常我们称为策略。策略的使用在很多方面是受到限制的[27]，比如：

1. 策略信息存储在数据库服务器中的，由于持久化这些数据是非常昂

贵的操作，所以大大的降低了系统的效率。

2. 策略维护解析非常困难，而且目前没有专门的策略解析服务器，这个部分由开发人员来做，大大降低了应用软件开发的效率。
3. 策略维护管理过于复杂，由于一个企业应用中可能的策略多达上千条，虽然也采取了策略组管理，但是维护策略仍然是一个代价昂贵的操作。
4. 策略不够智能化，策略没有解决策略冲突策略问题，所以，策略冲突将会给应用带来非常大的负面影响。

对于当前企业级应用中的策略的大量使用，有企业已经开始借助于规则引擎产品来解决实际问题。

规则引擎只是一个复杂应用中的一小部分，实际上，规则引擎不是用来处理工作流的，它也不是工作流引擎用来开发规则的工具。当然，可以选择专门的工具来做这个事情，但这不是 drools 设计的初衷[28]。

规则引擎是动态的，所谓动态即是指规则的存储，管理和更新将随着数据的变化而进行，它们通常被看作软件部署的一个解决方案，大多数 IT 部门的目的是防止软件的报废，这正是使用规则引擎的原因。

2.1.3 技术实现方式

drools 采用纯 java 技术和 xml 技术，drools 允许采用声明方式表达业务逻辑，可以使用 java/xml 编写业务规则，可以将 java 代码直接嵌入到 rule 文件中[29]。

在 drools2.5 中，规则的编辑采用 xml 文件格式，这样有利于同其他 java 流行的框架集成，例如 spring 框架等。由于 drools 现在是一个正在成熟的技术架构，在 drools3.0 和以后的版本中，drools 可能采用文本格式编辑规则，这更增加了可读性，从而降低了 drools 的难度。

由于在 drools2.5 和 drools3.0 中涉及到的 API 有很大的不同，本设计中主要采用 drools2.5final 及以后版本，在某些问题上依据 drools2.5final 版的解决方案。

2.2 Rete 算法

Rete 在拉丁文中是网络的意思，Rete 算法实现可分成两个部分：规则编译(Rule Compilation)和运行时执行(Runtime execution)[29][30]。

2.2.1 规则编译

规则编译描述的是如何将缓存中的规则构建生成一个有效的网络，这个网络就是用来过滤数据的，就如同产生数据一样。在顶端，有很多节点要匹配，随着深入到网络的底层，需要匹配的节点将变少，到了这个网络的最底端就是到了网络的终点。Rete 算法中主要涉及到 4 种节点：根节点，单输入节点，双输入节点，终节点。Rete 算法中涉及的节点类型如图 2-1 所示(注：本论文涉及到 Rete 规则都采用通用规则表示法[31]，如本图所示)。

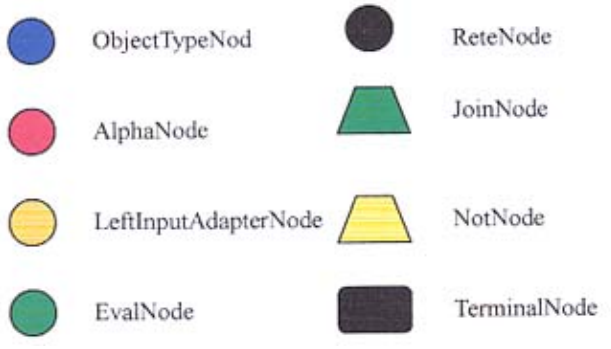

图 2-1 Rete 算法中的节点类型

根节点 ReteNode 是所有对象进入 Rete 网络的入口，从根节点，对象可以直接到达 ObjectTypeNode 节点，ObjectTypeNode 节点保存了规则引擎中涉及到的事实对象的类型，规则引擎该把对象传递给匹配对象类型的节点，以确保规则引擎不做多余的工作。例如有两个对象：account 和 order，如果规则引擎试图计算所有单个节点的值，那么它将浪费很多的资源，示例图如图 2-2 所示。

如果应用程序声明了一个 account 对象，那么它将不会由于 order 对象而产生任何节点。当一个对象被创建时，它将通过查找一个 HashMap 找回

一个 ObjectTypesNode 的列表。如果这个列表不存在，那么它将浏览所有的 ObjectTypeNode，来查找被缓存在 List 中的有效匹配。这确保 drools 能根据对象类的类型匹配。

ObjectTypdeNode 节点可以传递到 AlphaNode 节点、LeftInputAdapterNode 节点和 BetaNode 节点。AlphaNode 节点用来计算字面条件约束。1982 年 Rete 算法发表的时候只是支持相等的操作，但是现在 Rete 算法的实现支持很多的操作。

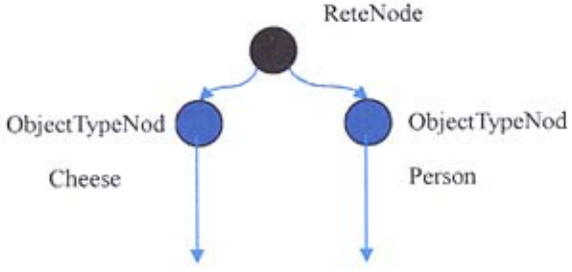

图 2-2 ObjectTypeNode 节点示例

一个规则针对一个单一的对象类型有多个字面条件时，它们被连接到一起。如果应用声明了一个 account 对象，那么它只有满足了第一个字面条件，才能进入到下一个 AlphaNode，进行匹配第二个字面条件匹配，示例图如图 2-3 所示。

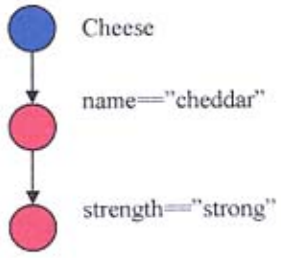

图 2-3 AlphaNode 节点示例图

drools 扩展了 Rete 算法，通过哈希算法优化了从 ObjectTypdeNode 到 AlphaNode 的传递，每次一个 AlphaNode 节点加入到 ObjectTypeNode 时，它将加入到一个哈希表中，值作为键，AlphaNode 作为值。当一个对象进入

ObjectTypeNode后，并不是总是进入所有的AlphaNode，它可以根据HashMap得到正确的AlphaNode，以避免不必要的比对。

有两种双输入节点：JoinNode 和 NotNode，这两种节点都属于BetaNode节点，BetaNode 节点用来比较两个对象及其属性。对象可以是相同类型或不用类型。一般来说，把这两种输入分别定义称为左输入和右输入。对于一个 BetaNode 来说，左输入是一个对象的列表，在 drools 中被称为元组（tuple），右输入是单个的对象。此外，BetaNode 节点还有记忆功能，左输入称为 BetaMemory，用来记忆所有的元组，右输入称为 AlphaMemory，用来记忆所有的输入工作内存的对象。drools 通过 BetaNode 节点索引功能扩展了 Rete 算法，例如，假定有一个 BetaNode 节点正在对一个字符串进行检查，当每个对象进入工作内存时，可以通过哈希表来查找该字符串的值。当事实对象进入 BetaNode 时，规则引擎不需要遍历所有的对象来查找交点，只需要确定与事实相关的交点即可，当找到一个有效的节点时，元组和对象互相关联，这称为部分匹配，然后传递到下一个节点。

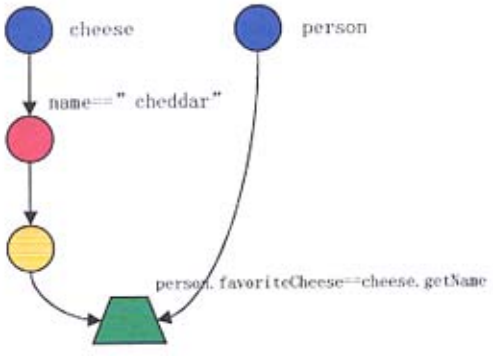

图 2-4 JoinNode 节点示例图

JoinNode 节点示例图如图 2-4 中，cheese 对象进入网络后，通过LeftInputNodeAdapter节点把对象cheese当作输入项并产生一个单对象的元组。

终节点表示一个规则匹配了所有的条件，即这条规则进行了完全的模式匹配。如果规则中使用了 or 连接符，那么一条规则可以被分解为几条子规则，所以，一条规则可能有多个终节点。

drools 还实现了节点共享，多条规则重复了相同的模式，这样就可以省去某些重复的匹配模式，不需要对重复的模式进行评估计算。

如下面规则，由 Rete 算法解析得到如图 2-5 所示的 Rete 网络，在 Rete 网络中共享了 AlphaNode，但没有共享 BetaNode，每个 Beta 节点都有自己的终节点，当然，如果第二个节点也相同，那么也会共享的。

```
rule
    when
        Cheese( $cheddar : name == "cheddar" )
        $person : Person( favouriteCheese == $cheddar )
    then
        System.out.println( $person.getName() + " likes cheddar" );
end
rule
    when
        Cheese( $cheddar : name == "cheddar" )
        $person : Person( favouriteCheese != $cheddar )
    then
        System.out.println( $person.getName() + " does not like cheddar" );
end
```

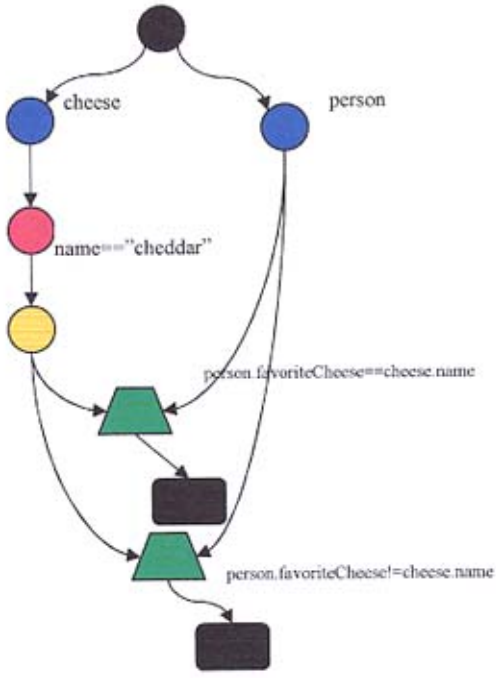

图 2-5 Rete 网络综合实例图

如果这时添加/删除规则，那么引擎将重新构建 Rete 网络图，例如添加

一条规则如下所示的规则，那么 Rete 网络将重新构建成如图 2-6 所示（为了便于表示本文将节点编号）。

```
rule
    when
        Cheese( $cheddar : name == "NESCAFE" )
        $person : Person( favouriteCheese == $cheddar )
    then
        System.out.println( $person.getName() + " likes nescafe" );
end
```

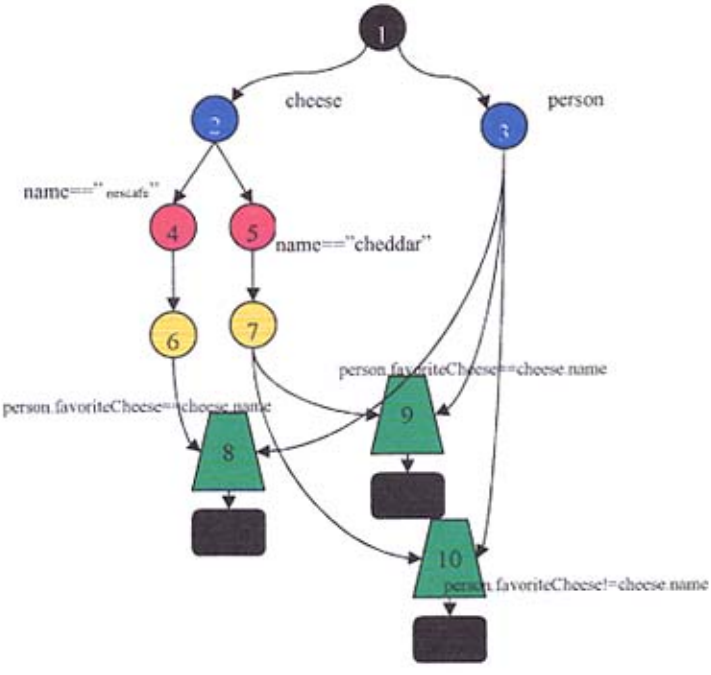

图 2-6 添加规则后 Rete 网络图

2.2.2 运行时执行

运行时执行，即是指规则引擎在执行声明/撤销/修改对象等操作时，对象在 Rete 网络中的传播过程。

在Rete网络中，ObjectTypeNode、AlphaNode、LeftInputAdapterNode和BetaNode等节点都具有记忆功能，即记录了它所关联的其他类型的节点的集合，例如图2-6中，ObjectTypeNode节点②保留了与之关联的AlphaNode节点④和节点⑤。

举例说明对象如何在Rete网络中传播的，如图2-6所示的Rete网络中传播。

在工作内存中依次声明如下对象。

person={name=" LI",favoriteCheese=" nescafe"}

cheese={name=" nescafe"}

当第一个对象person被声明进入Rete网络时，首先进入的是ReteNode节点①，从ReteNode节点①，引擎可以获得工作内存中的所有的对象类型节点ObjectTypeNode集合，进入与对象类型相匹配的ObjectTypeNode节点③，与person关联的规则不存在字面约束条件，所以直接进入与之关联的BetaNode节点的右输入，包括节点⑧、节点⑨、节点⑩，当有事实对象进入BetaNode节点时，BetaNode节点缓存了这些事实对象，并判断左输入元组，由于工作内存中不存在其他对象，BetaNode节点节点⑧、节点⑨、节点⑩都没有完全匹配，所以person对象进入网络的传播路径是：节点①->节点③->节点⑧；节点①->节点③->节点⑨；节点①->节点③->节点⑩。

当第二个对象cheese被声明进入Rete网络时，首先进入的是ReteNode节点①，从ReteNode节点①，引擎可以获得工作内存中的所有的对象类型节点ObjectTypeNode集合，进入与对象类型相匹配的ObjectTypeNode节点②，由于与cheese关联的规则存在字面约束条件，通过ObjectTypeNode节点②对象cheese进入满足字面约束条件的AlphaNode节点④，如果还有其他的字面约束条件，则依次匹配直到发现有字面约束条件不能得到满足为止，如果全部字面约束条件都满足，那么将检测是否存在or条件或exists条件，继续进行匹配。cheese对象进入AlphaNode节点④后，将通过LeftInputAdapterNode节点⑥创建元组缓存之前匹配的结果，然后该元组进入与节点⑥关联的BetaNode节点⑧。Cheese对象进入网络的传播路径是：节点①->节点④->节点⑥->节点⑧。

当与cheese对象关联的元组进入BetaNode节点⑧时，与节点⑧的右输入进行匹配，如匹配完全满足，那么将该元组置入议程。

执行引擎将议程中的元素取出，获得相关联的规则的结论（Consequence）并执行。

2.3 Leaps 算法

实验结果[32]证明 Leaps 算法能产生最快的、连续的、可执行的 OPS5 规则集，执行时间可以比使用 Rete 算法和 Treat 算法的 OPS5 编译器快几个数量级，Leaps 算法是通过运用复杂的数据结构和规则搜索算法以提高规则模式匹配算法的效率。

2.3.1 指针和容器

为了能够更加准确的描述 Leaps 算法，这里引入了两种描述性的规则语言 OPS5 和 CLIPS（见附录一）。Leaps 算法规则实例用 CLIPS 描述如下：

```
defrule bookcard ""
  (cardlost(reason type))
  =>
  (assert (response action activate_loss_function))
```

Leaps 算法中综合许多其他语言的优势，例如在 Leaps 算法中存在了 P2 语言的相似的很多概念，P2 语言是基于 C 语言的一个超集，这里介绍 Leaps 算法从 P2 语言中继承来的以下重要的概念，它们是：指针、容器和合成指针，这对理解 Leaps 算法非常重要。

容器就是具有相同数据类型的元素的序列或者集合，在其他需要高级语言中都实现了容器的概念，例如 Java 语言中 Array，List，Set 等，容器中的元素仅可以被一个运行时对象引用，该运行时的对象称为指针。

合成指针，复杂的数据结构通常包含了多个容器，各个容器中的元素通过一个合成指针相互连接。假定有 n 个容器 $C_1, C_2 \cdots\cdots C_n$，那么必存在一个 n 元组的集合$\langle e_1, e_2 \cdots\cdots e_n \rangle$，其中元素 e_i 属于容器 C_i，那么每一个数组$\langle e_1, e_2 \cdots\cdots e_n \rangle$就是一个 n 元合成指针。

如图 2-7 所示表达了容器 A、B、C 的关系，其中存在了如下合成指针。

{(a2, b1, c1), (a2, b1, c3), (a3, b2, c4), (a1, b3, c2) , (a1, b3, c4)}

合成指针表达了 n 个容器的关联关系，合成指针是有 n 个指针的有序组合，每个容器有且仅有一个指针。合成指针在规则设计中起到了重要作用，在 P2 程序中，容器可以取得一个或者多个别名，所以，容器就可以与其他容器或者自身进行关联。

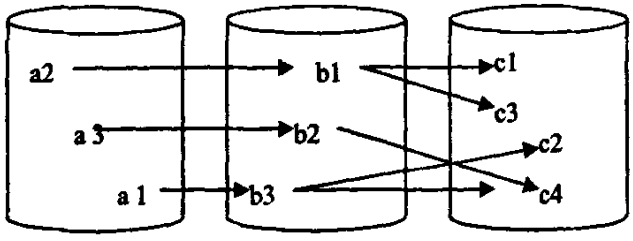

图 2-7 容器和合成指针示意图[36]

2.3.2 Leaps 算法

向前链式推理引擎，包括 Leaps 算法在内，都采用了“匹配-选取-执行”循环周期。当一条规则的全部断言都被满足时，那么与该规则关联的 n 元组被选取并且其关联的规则被执行。该循环一直进行下去直到没有可评估的规则为止。在 Rete 和 Treat 模式匹配算法中，引擎创建了满足断言的所有的元组，并存储到专门的数据结构中，如果执行规则那么这些元组将被更新，所以这会对引擎效率会有一个负的影响。Leaps 算法的主要贡献是 Lazy 元组评估，只有在必要的时候才会创建元组，如当规则条件中存在 not 和 exists 条件时创建并评估元组[33][34]。

在 Leaps 算法中，每一个元素分配了一个时间戳（TimeStamp），标志该元素是何时被声明进入工作内存或者从工作内存撤销的。当元素被声明或者撤销时，该元素的句柄被置入堆栈中，也就是说，堆栈中保留了元素的时间戳的逆序，即最近被声明或撤销的元素位于堆栈的顶部，最远被声明或撤销的元素位于堆栈的底部。

在一个规则执行周期中，堆栈顶部的元素首先被选中，称为主对象（Dominant Object，简称 DO）。DO 播种给所有规则的断言，引擎按照某一个特定顺序播种给规则，具体确定规则顺序的是 Leaps 算法规则冲突解决

策略。如果确定DO不能播种给指定的规则，那么将跳到下一条规则并给之播种，如果所有的规则都被检查或者播种过了，那么该DO就被抛出堆栈。如果一个元组完全匹配事实对象，那么其相应的规则将被激活并执行，规则的执行将可能会导致堆栈中元素的变化，例如插入或删除，这可能导致更多的元素置入堆栈中。当一条规则执行之后，那么继续从堆栈中取得另一个 DO，开始下一个规则执行周期。该执行周期将一直进行下去，除非遇到某一个固定的终点，例如，堆栈为空。

一个元素对象可能会在堆栈中出现两次：一次是插入一次是删除。很显然，元素插入堆栈早于堆栈中的元素被删除。工作内存中的元素在堆栈中可能会出现0，1，2次。

对于元素对象给规则播种，如果该元素对象位于堆栈中，显然，其作用是用该对象去匹配相关规则的元组，来过滤并选取相关的规则。但是，如果从堆栈中删除某对象，也会使该对象对相关规则播种，这和语义否定相联系的，某些规则会由于元素对象的删除而被激活执行。

在Leaps算法冲突策略[35]中，用于事实对象冲突策略有先入先出策略，用于规则冲突策略有：1.规则优先级策略，即通过规则属性 salience 设置规则优先级，2.正条件元素数目策略，即正条件元素数目越多优先级越高，3.先入先出策略，越早置入议程的规则优先级越高。

2.3.3 算法改进

如上节介绍了原Leaps算法，在将原Leaps算法用面向对象方法实现之前仍需要对算法做适当优化改进，这里总结了前人经验列出了几个主要的优化。

1. not 和 exists 处理方式[36]

与古典算法相比，变化最大的就是处理 not 和 exists 条件元素的方式。在古典算法中，采用的是浅容器，当处理 not 或 exists 时，通过遍历浅容器来判断元素对象的存在与否。当前处理方式是采用“Lazy”方式，通过创建元组延迟规则的激活。这和Rete算法中的“negative”节点处理方式类似。

2.优化堆栈存储暗对象[37]

之前提到过，事实对象被保存在堆栈中，包含声明进入工作内存的对象和从工作内存中撤销的对象，将从工作内存中撤销的对象定义为暗对象（shadows），统计表明暗对象很少会成功的播种规则，如果暗对象较多，那么花在处理暗对象的时间就非常多。

一种优化办法就是，将暗对象压入堆栈底而非栈顶，这违反了堆栈按照对象被声明或者撤销的时间戳来保存事实对象，但这会使得规则引擎更加高效。

3. 优化事实撤销操作[38]

在原 Leaps 算法中，删除操作和添加操作采用同一种方式。称此方式为基于再匹配删除。主要思想是给每个添加或删除操作一个标签，用来表明此操作是添加事实或撤销事实。此方法与其他方法相比，速度较慢。因为删除操作与添加操作的工作量几乎相同。在添加事实过程中所获得的信息并没有在执行删除操作时加以利用。

在基于链式存储结构中删除事实对象使用了如下方法，给事实集合的数据结构上增添额外的指针，当某个事实被删除时，可以沿着这些指针删除需要删除的元素。缺点就是需要大量的空间来存储链表，同时在创建和维护这些链表时也可能花费大量的额外时间。经验表明，采用此方法比原始方法要节省时间。

2.4 总结

本章通过首先对 drools 规则引擎设计目标、应用范围和技术实现方式等方面做了简单介绍，因为稍后的 Leaps 算法的设计和实现将以 drools 规则引擎作为其容器。

drools 规则引擎的模式匹配算法是 Rete 算法，本章详细介绍了 Rete 算法的规则编译和运行时执行，并详细介绍了另外一种模式匹配算法 Leaps 算法，和 Leaps 算法的改进，为后面章节的设计和实现提供理论支持。

第三章 Leaps 算法功能设计

3.1 功能结构

本设计主要实现 Leaps 算法的模式匹配功能，主要功能模块有：规则转化模块，Lazy 条件评估模块，元素对象存取模块，工作内存模块，规则库模块等，其总体用例图如图 3-1 所示。

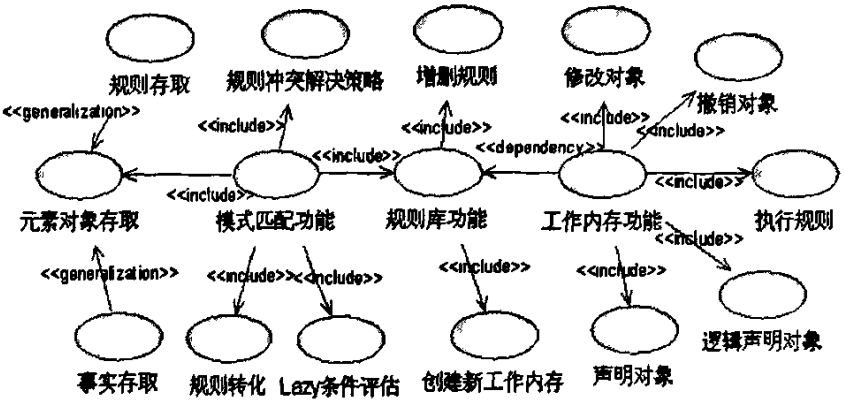

图 3-1 总体用例图

规则转化模块，主要功能是负责将 Rete 算法中的规则转化成符合 Leaps 算法规范的规则。由于本设计基本保留了 drools 引擎容器和 Rete 算法规则表现方法和执行容器，所以在使用新的 Leaps 模式匹配算法时需要将 Rete 规则转换为 Leaps 规则，并保留了两种规则之间的对应关系，在完成模式匹配后，规则的执行需要利用 Rete 算法的规则执行组件。该模块主要是在添加规则时使用，所以其功能分别在规则库设计和规则设计中介绍。

元素对象存取模块，主要功能是建立元素对象存取的统一容器，提供针对事实对象和规则对象的存取功能，其主要容器有事实表、堆栈和队列三种。drools 规则引擎已经设计实现了队列容器，本设计中不做详细介绍。

Lazy 条件评估组件，Leaps 算法采取的是在执行规则的时候进行条件评估，而不是在模式匹配过程中进行，这时 Lazy 条件评估策略的主要思想。Lazy 条件评估组件严格限定了何时才会创建能缓存中间匹配结果的元组，当添加规则，撤销对象和执行规则时涉及到 Lazy 条件评估组件功能，所以

除了 Lazy 条件评估组件本身功能之外，还介绍了 Lazy 条件评估组件的适应范围。

规则冲突解决策略模块，主要功能是定制Leaps算法限定的规则冲突解决策略，并且使规则冲突策略成为开放的功能模块，以备定制其他的规则冲突解决策略，规则冲突策略将在容器设计时做详细介绍。

规则库模块，主要功能是构建新工作内存以及规则的存取，包括添加规则，删除规则等。

工作内存模块，主要功能是为规则和事实进行模式匹配提供了场所，是规则引擎的核心组件。它提供规则引擎对应用程序的接口，其主要功能接口有声明对象，逻辑声明对象，撤销对象，修改对象，执行规则等。此外，工作内存保留了规则引擎的全部功能组件的引用和关联数据，所以其他组件只要关联了工作内存，就可以使用规则引擎的其他组件或者服务。

设计除了以上几个大的功能模块之外，还涉及了一些重要的组件，例如规则，事实，议程，堆栈等，有的功能模块在这些组件中所有涉及，所以这些组件会首先介绍其详细设计，接下来才会涉及到主要的功能模块。

3.2 组件设计

3.2.1 容器设计

本设计详细介绍Leaps算法中的两种存取容器：事实表和堆栈，这两种存储容器在规则引擎中各自负责不同的功能，其详细设计如下。

I 事实表设计[39]

事实表，即一个双向链表，用于存放事实对象和与事实相关的规则信息和元组信息，事实表不仅装载事实对象，而且装载规则对象和元组数据，事实表为其中的元素提供了严格的排序规范，该排序规范即规则冲突解决策略和事实冲突解决策略。

由于在 J2EE 中 java.util.TreeSet 类支持严格的排序功能，所以事实表可以通过继承扩展 TreeSet 类来实现。

事实表用例图如图 3-2 所示，其功能包含通用功能和特殊功能，通用功能有添加元素，删除元素，判断元素是否存在，清空事实表等，这些功能

和通常的双向链表功能和设计一样。

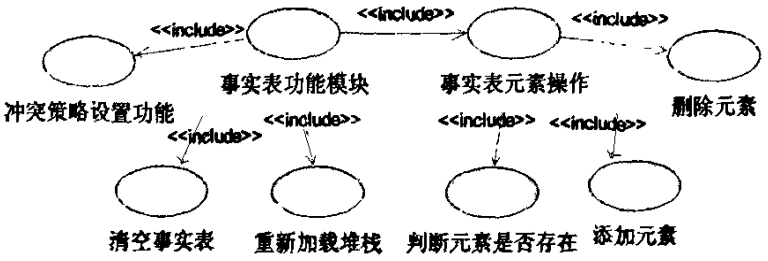

图 3-2 事实表用例图

冲突策略设置功能，事实和规则是存储在事实表中的，所以将事实和规则冲突解决策略加载在事实表上，在创建事实表时设置冲突解决策略，在使用过程中不能改变冲突策略。

在 J2EE 中，关于 java.util.TreeSet 类的排序是通过在创建 TreeSet 对象时注入 java.util.Comparator 接口的实现类对象来实现的，具体策略需要实现 Comparator 接口，这样以后再有元素进入事实表，则依据比较器的比较结果进行排序。

重新加载堆栈，即将事实表中的部分对象重新加载到堆栈中。假定规则引擎的最后一次执行规则的时刻 t_0 和事实对象进入工作内存的时刻 t_1，如果在添加规则时，发现存在这样的事实对象：其 t_1 晚于 t_0，且其在事实表中但不在堆栈中。那么需要将这些对象重新加载到堆栈中，引擎下一次执行规则时将对这些事实对象进行模式匹配操作。

II 堆栈设计[40]

堆栈主要用于被声明进入工作内存待进行模式匹配的事实对象，所有曾经进入引擎的对象都在事实表中按照先后顺序保留了备份，所以事实表就是一个事实库，而堆栈是事实对象等待模式匹配的容器，一旦对象模式匹配完毕将离开堆栈。

为了满足 Leaps 算法对堆栈的特殊的要求，例如：在执行一条规则的时候，可以对堆栈中的事实对象进行删除操作，设计在传统意义堆栈的基础上，增加了删除栈中元素的功能。

同时堆栈支持浅对象压入栈底操作，请参考 2.3.3 一节算法优化 2。

由于在 J2EE 中 java.util.Stack 类[41]严格符合堆栈要求，所以堆栈类通过扩展 Stack 类来实现。

堆栈用例图如图3-3所示。其功能有压入栈顶，压入栈底，弹出堆栈，判断堆栈是否为空，取尾功能，删除栈中元素等操作。

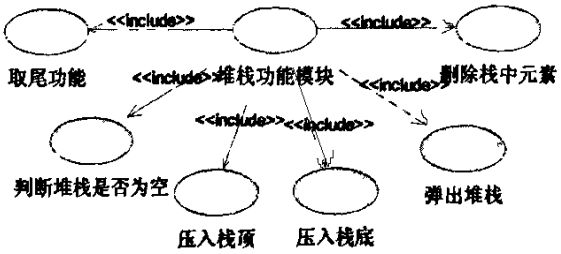

图3-3 堆栈用例图

压入栈顶，弹出堆栈，判断堆栈是否为空都是堆栈的常用操作，与通用的堆栈功能相同。

取尾功能，即指返回栈顶元素，但是不会将栈顶元素从堆栈中删除。

删除栈中元素，即允许从堆栈中的任何位置删除堆栈中的元素。

压入栈底，将元素压入栈底，原有的元素依次向栈顶移动一个位置。

堆栈结构示意图如3-4所示。

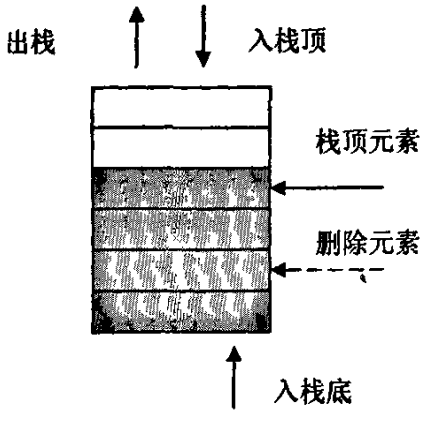

图3-4 堆栈结构示意图

3.2.2 规则和事实构建

Leaps算法的规则和事实[42]在设计结构上不同于基于规则的专家系统和Rete算法中的规则，而本设计采用drools容器的规则编写规范和规则解析执行组件，所以需要考虑将Rete规则转换成Leaps规则，这个构造过程是

通过一个 LeapsRuleTransformer 规则转换组件来实现的。规则转换时序图如图 3-5 所示，注：步骤 6 示意将 Rete 规则中的列根据不同的类型解析返回不同的列约束，例如：not，exists，eval 等，这些列对象是构造 Leaps 规则的必须属性。

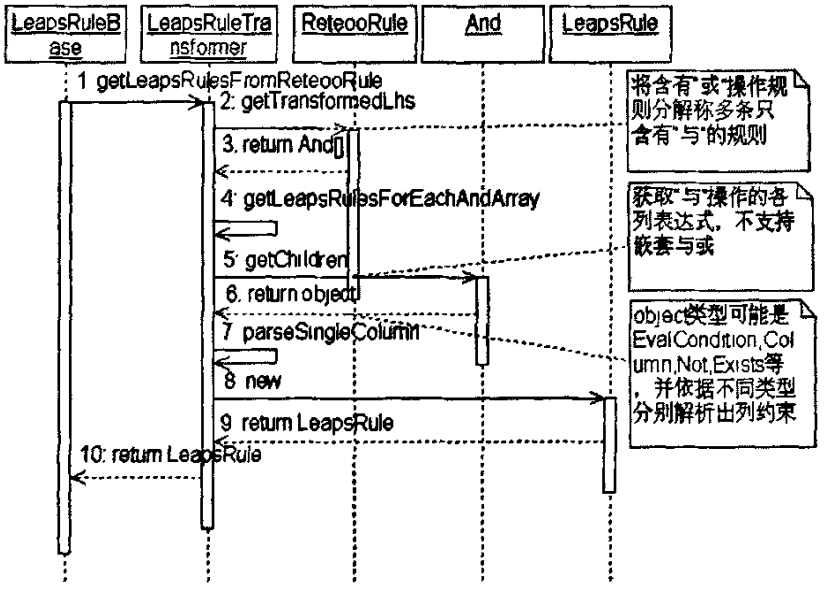

图 3-5 规则转换时序图

Leaps 算法中提到在“必要的时候”才构造元组缓存中间匹配结果，本设计设定这个必要的时候是当且仅当规则中存在 not 或 exist 时[43]。

所以在构造 Leaps 规则的时候，只针对包含 not、exist 条件元素的规则创建元组用以缓存中间结果。当一个与包含 not 或 exist 条件元素的规则相关联的事实对象被声明进入工作内存时，它需要获取与它本身相关联的元组信息，包括 not 元组和 exist 元组，以及已被激活的元组，这是 Leaps 算法中的事实句柄类的必要的属性。

句柄类在设计中起到封装类的作用，关联关系最密切的实体类对象。本设计构造了两个句柄类，一个是 Leaps 规则句柄类，一个是 Leaps 事实句柄类，通常情况下，实体类都是通过句柄类引用的。

规则引擎中规则和事实的构建都是在规则库和工作内存进行的，事实的构建直接通过事实句柄类包装并且获得事实在工作内存中的统一 ID 即可，对于对象关联的 not 元组和 exist 元组，通过 Lazy 条件评估组件装载的，请参考 Lazy 条件评估组件一节。

规则构建时序图如图 3-6 所示，步骤 6 中包含了 Leaps 规则转换，如图

3-5 所示。

应用程序首先构建规则包，规则包中可以包含一条或者多条 Rete 规则，然后应用程序调用规则库的添加包方法，将包中的 Rete 规则通过 Leaps 规则转换器，转换成 Leaps 规则，并且添加到事实表中。

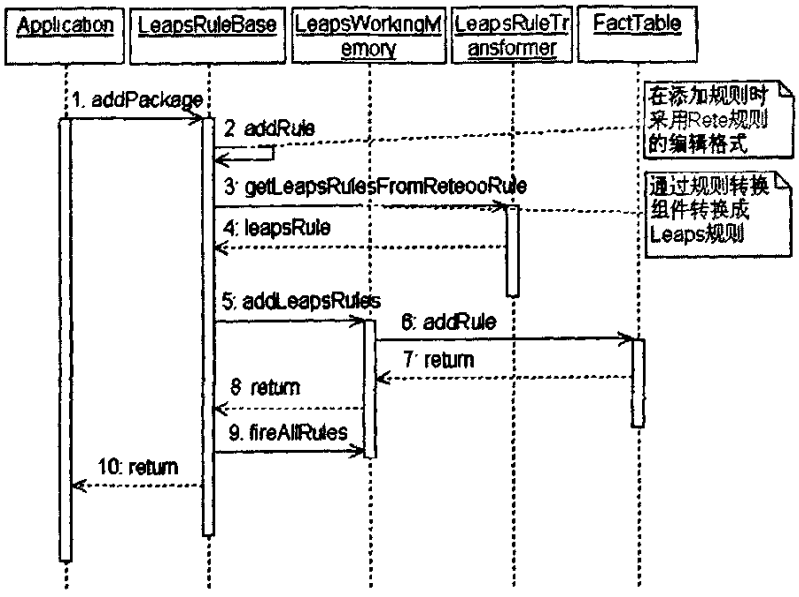

图 3-6 规则构建时序图

在添加规则时，当且仅当简单条件约束评估全部满足时才进行，需要对 eval，not，exists 条件进行评估，如果匹配完全满足，那么将所关联的元组缓存并置入议程，最后判断是否元组处于激活状态，如果是则执行其规则。添加规则时条件评估活动图如图 3-7 所示。

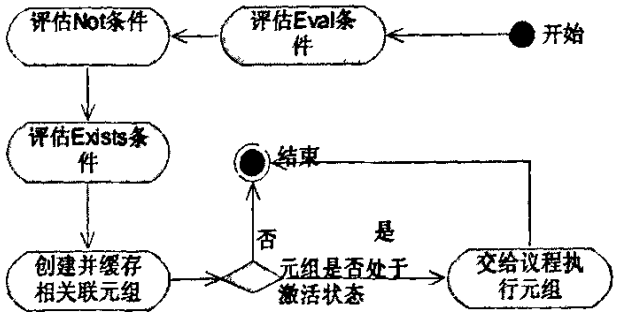

图 3-7 添加规则条件评估活动图

3.2.3 议程设计

Leaps算法中议程[44]是一个装载处于激活状态的规则的容器和执行规则的引擎。将议程按某种分类组合就组成一个议程组。

议程组可以处于聚焦状态，也可以出于非聚焦状态，只有处于聚焦状态的议程组中的议程中的激活才能被执行，处于聚焦状态的议程组保存在一个堆栈中，称作聚焦议程组堆栈。

议程和议程组的关系是多对多的关系，一个议程可以添加到多个议程组，一个议程组可以关联多个议程。每一个议程都必须至少属于一个议程组，如果没有给议程指定议程组，那么该议程属于默认议程组，其默认议程组名称为“MAIN”，MAIN 议程组是不可以被删除的。可以为议程添加关联其他议程组。MAIN 议程组始终处于聚焦状态，且位于聚焦议程组堆栈的栈底。

当议程接到执行议程元素的消息时，首先取出处于聚焦议程组堆栈中栈顶议程组，如果该议程组为空，那么从堆栈中弹出。如果不为空，那么取出其中的激活并执行。MAIN 议程组始终是最后一个执行的议程组，它在堆栈的栈底。

在执行议程中的激活时，可以设置过滤器，使引擎能够按照过滤条件执行指定的议程中的激活。

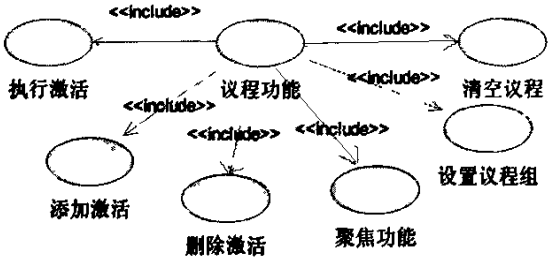

图 3-8 议程用例图

议程用例图如图 3-8 所示，其主要功能有：

1.添加激活，当元组完全匹配事实对象时，该元组关联的规则被包装成激活（Activation），放置到议程上，并将激活状态设置为 true。

2.删除激活，当议程上的激活因为工作内存中某些事件发生导致激活的

条件不再满足时，需要从议程上删除。注意：在从议程上删除激活时需要将激活设置为非激活状态，然后再删除，否则将会从激活组中删除。

3. 执行激活，如果工作内存对议程发出执行规则的消息，那么最终执行将回归到议程执行激活。

4. 设置议程组，将议程关联到某个议程组。

5. 聚焦功能，工作内存中可能存在多个议程，这些议程可能处于不同的议程组，只有处于聚焦状态的议程组的议程才能被规则引擎执行其中的激活，可以设置一个议程组处于聚焦状态或者非聚焦状态，以设定该议程或者议程组中的激活是否被执行。

6. 清空议程，依次删除议程中的所有激活。

执行激活流程如图 3-9 所示，当规则引擎发出执行的消息后，议程将执行所有相关的激活，激活的执行最终将递交给 Consequence 规则引擎组件去执行。

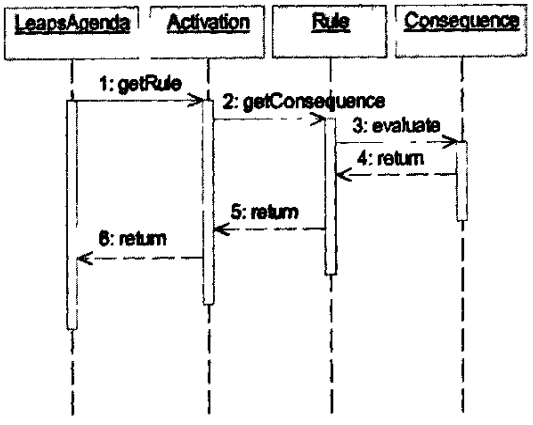

图 3-9 议程执行激活时序图

3.2.4 Lazy 条件评估设计

Leaps 算法的主要贡献是 Lazy 条件评估策略，所谓 Lazy 条件评估策略即是指引擎不会在声明对象时进行 Not 或 Exists 条件的评估，而是在执行规则的过程中进行，这一个策略会使得规则引擎的模式匹配效率得到优化，但同时也会对规则的执行产生一定的影响。

Lazy 条件评估发生在执行规则时，引擎对包含 not 或 exists 条件的规则进行条件评估。其中事实与其关联的元组并不是在评估时建立关联关系，

这种关联关系是在以下两种情况建立的：

1. 当添加规则的时候，如果规则中包含 not 或 exists 条件，那么需要将元组和事实对象建立关联关系。
2. 当撤销对象时，原来不满足的包含 not 或 exists 条件规则可能由于对象撤销而被激活，所有需要变更与事实对象关联的元组信息。

事实对象 fact 在进行条件评估之前应该处于的状态是：确定一条 Leaps 规则 rule 与之对应。其活动图如图 3-10 所示。

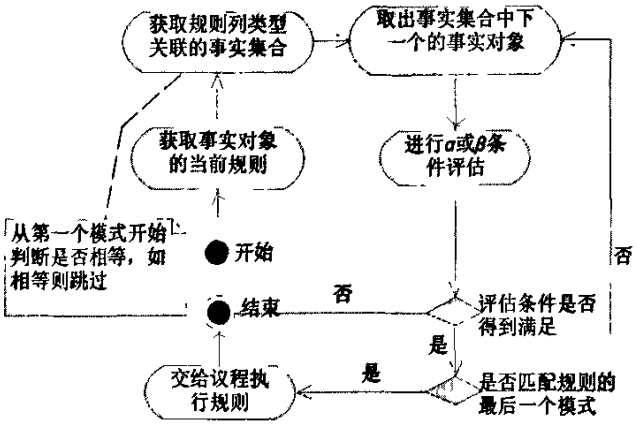

图 3-10 Lazy 条件评估活动图

3.3 规则库模块设计

Leaps 规则库是 Leaps 规则的集合，集中管理 Leaps 规则的创建、添加、删除等功能，并提供构造新工作内存功能。规则库用例图如图 3-11 所示。

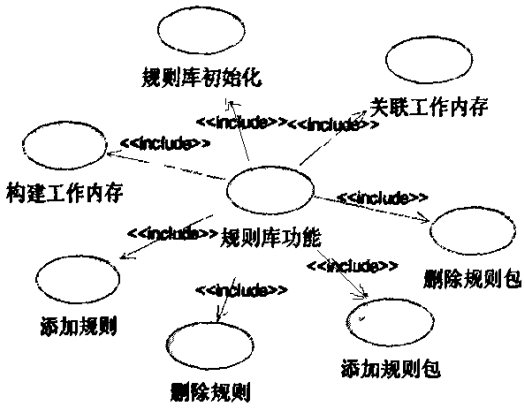

图 3-11 规则库用例图

规则库初始化时序图如图 3-12 所示，注：步骤 1-3 是构造一个空的规则库，步骤 4-11 是将读取到 Package 对象中的规则初始化到规则库中，即添加规则包（Package）的过程。具体添加规则请参考添加规则时序图 3-6 和规则转化时序图 3-5。

Leaps 规则库组件是规则引擎的一个运行时组件，它的最主要的功能就是构建一个新的工作内存，建立与工作内存的弱关联关系，并且提供了如下方法来创建一个新的工作内存。

1. LeapsRuleBase.newWorkingMomery()
2. LeapsRuleBase.newWorkingMemory(boolean)

当新加入的事实对象能自动的传播到所有关联的工作内存中，如果在创建一个工作内存取消了这个弱关联，那么需要应用程序手动去维护这个传播带来的对其他工作内存的更新。如图 3-13 所示。

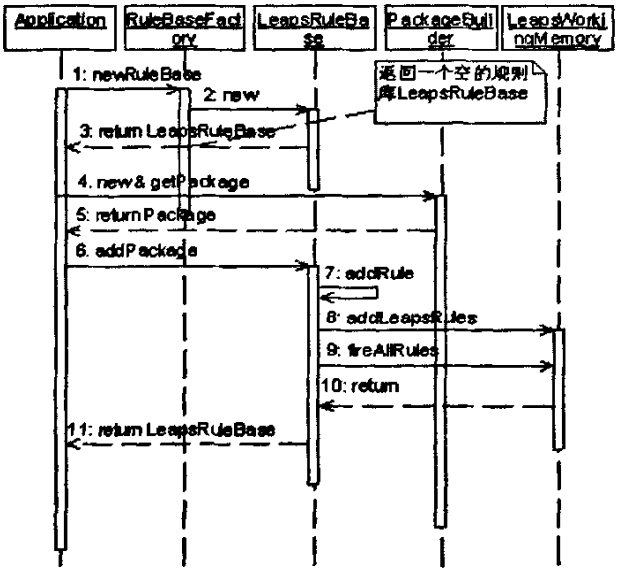

图 3-12 Leaps 规则库初始化时序图

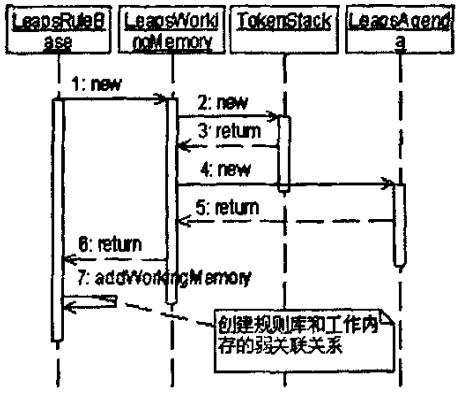

图 3-13 构造新的工作内存时序图

请参考规则添加时序图 3-6 和规则转换时序图 3-5 所示。

规则库删除规则时，除了删除规则库中缓存的规则，删除工作内存中缓存的规则，删除工作内存中缓存的句柄类等，还需要将真理维持系统中缓存的对象的依赖删除，如果规则处于激活状态，还需要从议程中删除，删除规则活动图如图 3-14 所示。

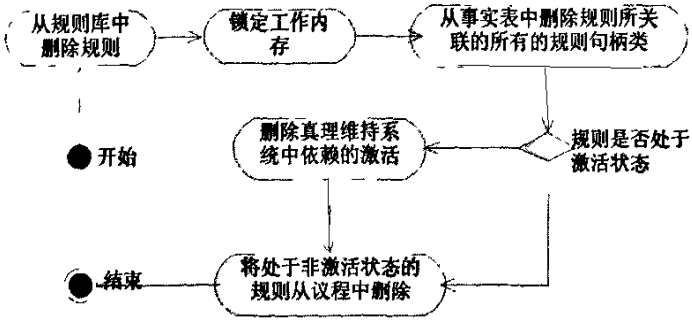

图 3-14 删除规则活动图

3.4 工作内存模块设计

Leaps 工作内存组件 LeapsWorkingMemory 是由 Leaps 规则库创建的。Leaps 工作内存主要功能是为应用程序数据事实对象和业务规则发生模式匹配提供场所，所有的应用程序数据对象都经过包装成 FactHandle，同样的，所有的业务规则也需要包装成 RuleHandle，在工作内存中保留了 FactHandle 和 RuleHandle 的引用。对于事实对象和规则对象，都采用了事实表的方式来存取。

同时，WorkingMemory 也保留了 RuleBase 的引用，与 RuleBase 保持双向关联关系，这里 RuleBase 和 WorkingMomery 构成了观察者模式[45]，从而实现对 Fact 对象的更新，RuleBase 和 WorkingMemory 保持同步。工作内存用例图如图 3-15。

如图 3-12 所示创建工作内存时序图，在创建工作内存时仅仅是创建了 3 个对象，所以从技术上讲创建一个工作内存是一个廉价的操作。

工作内存提供了声明对象，逻辑声明对象，撤销对象，修改对象等操作，并提供了围绕 TokenStack 的相关操作，还有执行规则操作 fireAllRules。声明操作分为 stated 声明和 logical 声明[46]，设计首先针对 stated 声明操作详细介绍如下。

工作内存 stated 声明流程图如图 3-16 所示，首先判断该对象是否存在，如果存在，然后判断该对象是 stated 还是 justified，如果是 stated 则返回已存在的 FactHandle，如果是 justified 则覆盖 justified 并待声明的对象设置为 stated 和返回已存在的对象的 FactHandle。如果该对象不存在，

则判断是否存在一个等 Equal 的对象，如果不存在则返回一个新的 FactHandle，如果如果存在，则判断该对象是 stated 还是 justified，如果是 stated 则返回一个新的 FactHandle，如果是 justified 则提示是否放弃逻辑声明，如果放弃，则覆盖 justified，并设置 stated，将已有的 FactHandle 赋值给新的对象。如果不放弃，则覆盖 justified 并设置 stated，返回已存在对象的 FactHandle。

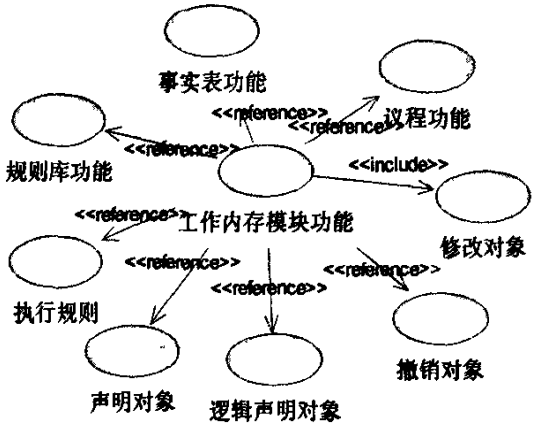

图 3-15 工作内存用例图

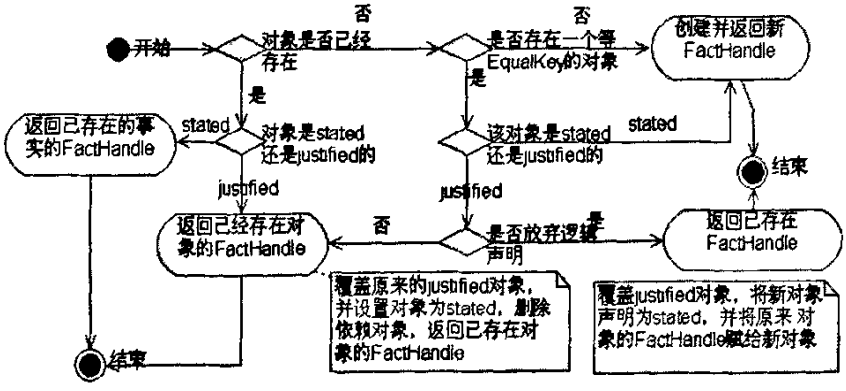

图 3-16 工作内存 stated 声明流程图

逻辑声明（Logical Assertion）是充分利用了真理维持系统，体现了事实对象之间的逻辑依赖关系，在声明对象是建立逻辑关系，在撤销被依赖对象时，这种关系自动解除，并且依赖对象自动撤销。逻辑声明的流程图如图 3-17 所示。

例如，当火灾发生（Emergency（fire））时，有毒气泄漏（Gas_giveaway（））；消防人员需要带氧气面具（Oxygen_mask（））；当火灾被扑灭，且

没有毒气泄漏时，消防人员不需要带氧气面具了。

这时在声明 Emergency (fire)，(Gas_giveaway () 和 Oxygen_mask () 时，需要建立一种逻辑依赖关系，当前两个对象被撤销时，将自动撤销 Oxygen_mask ()。

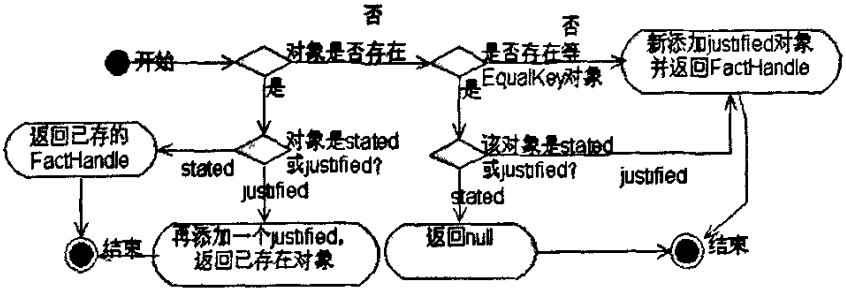

图 3-17 工作内存 logical 声明流程图

本设计工作内存撤销事实对象提供了两种设计方案，一种是传统的方式，即依次删除事实表中的事实对象及其相关信息，一种方案是将待删除的事实对象压入栈底，在执行规则时做出处理。这两种设计方案如下。

方案 I

流程图如图 3-18 所示，首先从事实表中删除该对象，删除该事实对象关联的处于激活状态的元组，然后从关联的 not 元组和 exists 元组删除该事实对象，并重新评估这些元组，如果元组处于激活状态则交给议程执行，如果处于非激活状态则钝化，最后从堆栈中删除该事实对象。

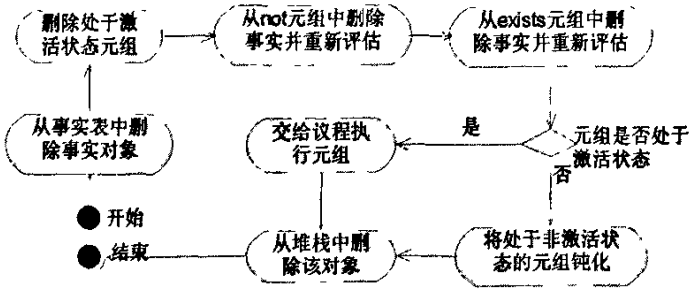

图 3-18 工作内存撤销事实对象流程图 I

方案 II

方案采用的是浅事实处理方式，由于撤销事实对象而进入堆栈的事实对象成为浅事实对象，如 2.3.3 算法优化一节提到的，如果浅对象过多，将导致引擎处理效率的低下，本设计采用了一种将浅事实对象延后处理的策略，即将浅事实对象压入栈底。这样在执行规则之前，就可以提出满足以

下条件的事实对象：第一次由于声明进入栈顶，第二次由于撤销进入栈底，如果前者没有进行模式匹配，那么就可以将这两次操作互相消除，从而大大提高规则引擎的执行效率。如果第一次声明已经进行模式匹配，那么需要采用方案 I 的方式进行撤销操作。其流程图如图 3-19 所示。

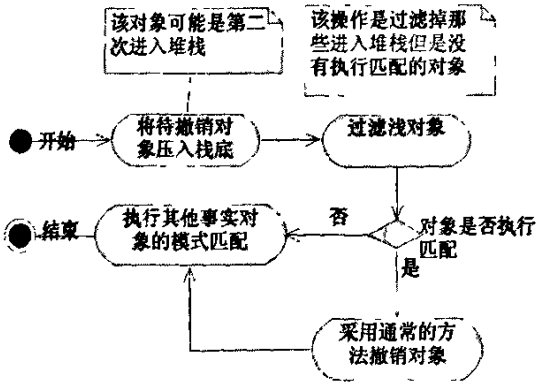

图 3-19 工作内存撤销事实对象流程图 II

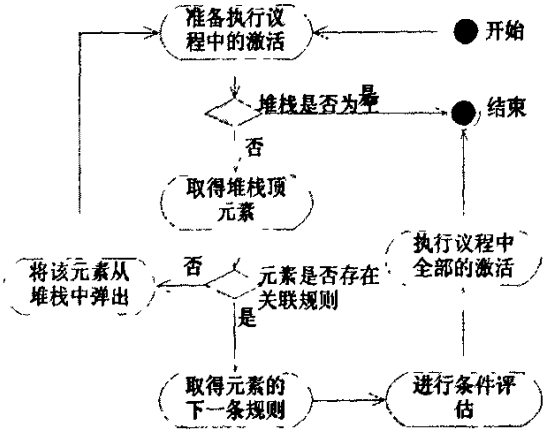

图 3-20 执行规则流程图

工作内存执行规则流程图如图 3-20 所示，请结合图 3-9 议程执行时序图。

1. 从堆栈中弹出一个主对象元素（token）。
2. 判断 token 是否可用，如果不可用则跳过并下继续一条规则；如果不可用且没有下一条规则了，那么从堆栈中弹出此元素。
3. 如果该元素关联的主对象不为空，那么设置 token 为可用状态。
4. 对规则进行条件评估，如果该元素处于激活状态则置入议程。

5. 依次执行议程上的全部激活。
6. 如果堆栈不为空，则从堆栈中弹出另一个主对象。
7. 重复步骤 2 到 6，直到某个事件发生，例如堆栈为空，或者引擎执行的规则超过了所设置的最大执行规则数。

3.5 规则冲突解决策略设计

如果同时激活了多个规则，则称之为执行规则存在冲突，解决冲突策略是,将激活的规则安顺序放入议程。

在 3.2.1 容器设计一节的事实表设计中，定义了规则引擎集成策略的统一策略接口是 java.util.Comparator，要使得规则引擎集成多种策略，需在统一策略接口中再集成 java.util.Comparator 实例，应用门面模式[38]，其结构示意图如图 3-21 所示。

Leaps 允许在进行事实对象交叉模式匹配之前执行规则，这样 Leaps 算法可以获得更高的效率。相反，在 Rete 算法中，将冲突解决策略放到最前面，也就是说在模式匹配之前必须作如下事情：

1、所有模式匹配必须被尝试进行过。

2、激活必须进行基于冲突策略的排序。

3、首元素的 Consequence 必须被开始执行。

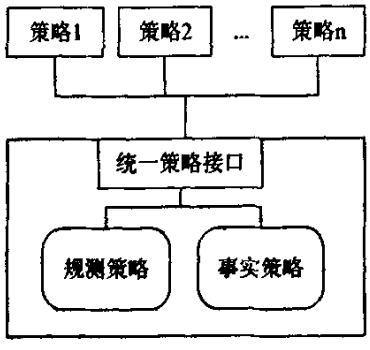

图 3-21 冲突策略结构示意图

Leaps 却没有这些限定，Leaps 算法总共设定了三种规则冲突策略[40]，当然可以依需求再设定其他策略。

策略 1. 规则优先级策略

策略 2. 规则复杂度策略

策略 3. 先入先出策略（FIFO）[47]

策略1和策略3都在Rete算法中涉及到了，这里仅策略2，如果在Leaps规则中有多个条件元素（CE），那么在进行匹配时优先与条件元素数目较多的规则进行匹配。

这三个冲突策略的优先级顺序时1，2，3依次递减。

3.6 总结

本章是Leaps算法的一个具体设计，包括堆栈和事实表、规则和事实对象、议程、元组、规则库、工作内存等，这些组件和模块的合理的面向对象的设计对于真正提高drools规则引擎的效率起到至关重要的作用。

本设计将规则冲突策略模块设计为一个开放的模块，可以根据不同的应用轻松定义多种冲突策略。本设计采用多个Hash表组成的树型结构存储事实对象和与事实对象相关的信息，在撤销对象时可以提高效率，请参考论文2.3.3小节。本设计在撤销对象时增加了一种浅对象的处理方式，对于规则中存在大量的not和exists条件时可以明显的提高模式匹配和规则执行的效率，请参考论文2.3.3小节。

drools规则引擎用面向对象思想严格实现了Rete算法，本设计以此为Leaps算法实现的容器，重新设计了模式匹配模块，其对应用接口依旧不变。

第四章 Leaps 算法实现

本设计实现采用面向对象语言 java 实现的，将开源项目 drools 规则引擎和 Rete 算法的实现作为 Leaps 算法实现的容器，JDK1.5.0 作为虚拟机，Eclipse3.2 作为编辑工具，并安装 Drools Rule WorkBench 3.0.5 插件。请参考 5.1 节的测试环境。

4.1 容器和冲突策略实现

4.1.1 容器实现

Leaps 元素对象容器主要是堆栈和事实表，事实对象和规则对象存储在堆栈和事实表中，从面向对象的角度考虑，将事实和规则包装成统一的元素对象： Element，该元素类是双向链表中的节点，它仅保留了左节点和右节点的引用，该类的重要用途是使规则引擎依据节点对象来取得元素的迭代。

由于在 J2EE 中，Stack 类基本满足 leaps 算法中对堆栈的特殊要求，堆栈实现类 MainStack 直接扩展了 java.util.Stack 类，提供了 remove 方法删除栈中元素，所以实现类 MainStack 继承扩展 Stack 类功能，并添加一个压入栈底方法即可。堆栈实现示意性代码如下：

```
public class MainStack<E> extends Stack<E> implements Serializable {
  public E pushBottom(E element){
    add(0, element);
    return element;
  }
}
```

如图 4-1 所示事实表类图，首先将对事实表的主要类说简要说明如下。

Table：一个双向链表，其元素是规则或者事实对象，同时具备定制规则或事实冲突解决策略的功能，这是通过将冲突策略定义为 Comparator 接

口的实现而实现的。

RuleTable：用来存取规则对象的规则表，其功能与 Table 完全一致，为了设计思路清晰故定义 RuleTable 来存放规则，与 FactTable 处于同一层次。

FactTable：用来存取事实对象的事实表，同时保留了事实对象与规则的关联关系以及事实对象与元组的关联关系。

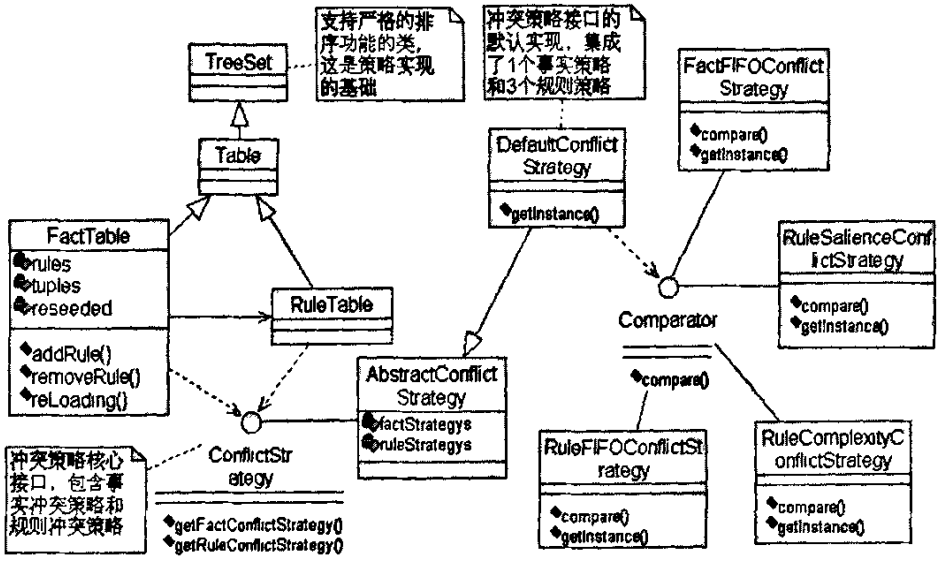

图 4-1 事实表冲突策略类图

如图 4-1 事实表类图所示，其主要属性和方法说明如下。

rules 属性：事实对象关联的规则集合，即如果规则的列中含有某对象的类型，那么该对象的关联的规则集合中就包含该规则。

tuples 属性：当规则中包含 not 或 exist 时，引擎将创建元组，并与事实对象建立关联关系。

reseeded 属性：在添加规则之前声明进入工作内存的事实对象是否需要重新进行模式匹配，如果为真，则将满足条件的事实对象重新装载到堆栈中。

reLoading 私有方法：如果在 reseeded 属性为真，那么工作内存中已有的对象重新放到堆栈中。例如，规则引擎执行规则之后，再添加一条规则，之后需将最后声明的对象放置到堆栈中。

4.1.2 冲突策略解析器实现

冲突策略包含了规则冲突策略和事实冲突策略，设计提供了通用的冲突策略接口，如图4-1所示，其主要类和接口说明如下。

ConfilctStrategy 接口：冲突策略解析器核心接口，提供了获取事实和规则冲突策略方法，这里使用了门面设计模式，统一了多种具体策略。

AbstractConflictStrategy 抽象类：ConfilctStrategy 冲突策略解析器的抽象实现，并且提供了两个冲突策略解析器 RuleConflictStrategy 和 FactConflictStrategy。

DefaultConflictStrategy：ConfilctStrategy 的默认实现，为规则和事实冲突策略解析器提供了默认实现，并保留了四种策略的简单实现。

FactFIFOConflictStrategy：用于事实的策略，即先进先出（FIFO）原则，通过记录进入工作内存的先后顺序，从而决定对象先进行模式匹配的顺序，或者决定再次进入堆栈的顺序。其示意性代码如下（以下几种策略的实现类似）：

```
public class FactFIFOConflictStrategy implements Comparator{
  public int compare(Object o1,Object o2) {
    return (-1) * AbstractConflictStrategy.compare(
      ((DefaultFactHandle) o1).getRecency(),
      ((DefaultFactHandle) o2).getRecency());
  };
}
```

RuleFIFOConflictStrategy：用于规则的策略，即先进先出（FIFO）原则，通过记录进入规则库的先后顺序，从而决定哪个规则先进行模式匹配。

RuleComplexityConflictStrategy：用于规则的策略，即根据规则的复杂度来确定规则的优先级，所谓复杂度即是指规则正 CE 的数目，正 CE 数目越多那么规则的优先级越高。

RuleSalienceConflictStrategy：用于规则的策略，依据 Rete 算法中的规则定义时的 salience 属性来判断规则的优先级，salience 数值越小优先级越高。

4.1.3 冲突策略设置

冲突策略的设置示意性代码如下所示，L4-L8 为 FactTable 类构造器，用来设置策略解析器，L1-L3 为 FactTable 的父类 Table 利用策略解析器保证策略被设置到事实表中（L2）。L9 为工作内存中在创建事实表时设置策略解析器，即策略的应用。

```
1 public Table(final Comparator comparator) {
2    this.set = new TreeSet(new RecordComparator(comparator));
3 }
4 public LeapsFactHandleTable(final ConflictResolver conflictResolver) {
5    super(conflictResolver.getFactConflictResolver());
6    this.rules = new LeapsRuleHandleTable(conflictResolver.getRuleConflictResolver());
7    ......
8}
9table = new FactTable( DefaultConflictResolver.getInstance());
```

4.2 规则和事实实现

Leaps 算法将 not 和 exist 条件定义为特殊匹配符号，关于其表现和解析功能的实现由 LeapsRule 和规则转换器组件类 LeapsRuleTransformer 完成的，LeapsRule 类保留了源属性，这些属性包括 columnConstraints、notColumnConstraints 、 existColumnConstraints 、 evalConditons 、evalConditionsPresent、evalsNotsClasses、agendaGroup 等，这些属性是只读的，仅在创建时进行初始化。其解析由 LeapsRuleTransformer 组件完成。

如图 4-2 所示，设计构建了 LeapsRuleHandle 作为 LeapsRule 的句柄类，所以将 LeapsRule 当作事实对象来存储是一个不错的选择，故 LeapsRuleHandle 直接扩展了 DefaultFactHandle 类。在 Leaps 算法的规则冲突解决策略中有一个先入先出策略，所以需要一个属性来标志规则进入工作内存的先后顺序。用来标志先入先出的属性和功能可以在其父类中实现，这样就可以与事实对象句柄类共用。另外 LeapsRuleHandle 是与规则的正 CE 相关联的，属性 dominantPosition 表示正 CE 在规则中的位置。

规则中 not，exist 条件评估功能实现分为两步：操作解析和操作执行。

操作解析是指将 Rete 规则中的“非”或“存在”转换为 Leaps 规则中的“非”或“存在”操作，这个功能是由规则转换器（LeapsRuleTransformer）完成的。

规则转换器类方法说明如下：

Rule. getTransformedLhs 方法：若规则中不存在 or 连接符，则返回一个以 and 连接元素组合；若存在 or 操作符，则返回以 or 为分隔符的以 and 连接元素组合的数组。

getLeapsRules 方法：解析 Rete 算法规则，返回 Leaps 算法规则，其示意性代码如下：

```
List getLeapsRules(And and,Rule rule) {
...
  for (Iterator it = and.getChildren().iterator();it.hasNext();) {
    Object object = it.next();
    if (object instanceof EvalCondition ) {
      EvalCondition eval = (EvalCondition) object;
      evalConditions.add(eval);
    }else{
      constraints = LeapsRuleTransformer.
          parseSingleColumn((Column)object);
    }
    ...
    notCols.add(constraints);
    existsCols.add(constraints);
    cols.add(constraints);
  }
  leapsRules.add(new LeapsRule( rule,cols,notCols,existsCols,evalConditions));
  return leapsRules;
}
```

parseSingleColumn 静态方法：针对由 and 连接的条件元素集合进行解析，以此判断连接元素对象是否是 EvalCondition、Column、Not、Exist，并依据这些对象获取列约束，然后构建一个 LeapsRule 对象。

在 leaps 算法中，规则和事实都是存储在工作内存的事实表中的，对规则和事实的存取都是通过其句柄来实现的，所以设计可以将 leaps 规则看作事实，故实现把 LeapsFactHandle 和 LeapsRuleHandle 都作为 DefaultFactHandle 的子类。

事实对象句柄类 LeapsFactHandle，这里保留了三个属性：activatedTuple、notTuples、existsTuples，并提供了这三个属性的读写方法。

activatedTuple 属性：对于事实而言，需要关联与之关联的已激活的元组，在适当的时候激活这些元组。在 Rete 算法中，元组是和节点关联的，AlphaNode，LeftInputAdapterNode，JoinNode，NotNode 都关联了元组，这些节点都负责维护和传播其关联的元组，而在 Leaps 算法中则不是，这是 leaps 算法的性能优越所在。

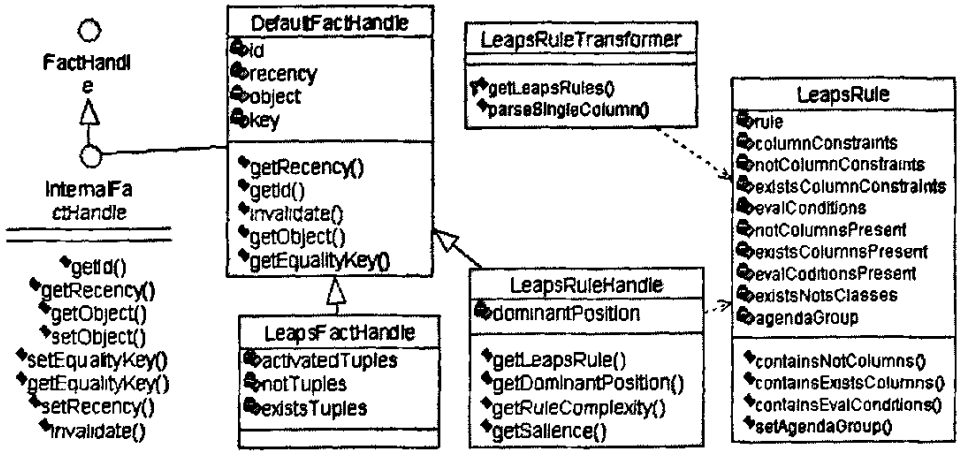

图 4-2 事实和规则类图

notTuples 和 existsTuples 属性：设计仅仅在规则中存在 not 和 exists 操作符是才创建元组，所以 LeapsRule 需要维护和传播这些元组。

4.3 元组设计实现

元组[48]（tuple）是模式匹配的中间结果，它在 Rete 网络中传播，当元组满足完全匹配的条件时，即被放置到议程上，等待系统发出执行消息，规则引擎会从议程上逐一执行相应的规则，直到议程为空，或者达到某一特定的限制，比如达到引擎允许执行的最大规则数。

在添加规则时，如果规则中仅存在 not 或 exist 条件，那么就规则引擎就会创建元组，并有 Lazy 条件评估组件对 LeapsRule 进行简单评估，其结果保留在 LeapsRule 中。

事实对象被声明进入了工作内存后，可能会导致某些元组的完全匹配条

件不再完全满足，也可能导致某些元组由部分满足变为完全满足。当元组完全匹配时，它将处于激活状态并放置到议程上，所以元组是一个事实，规则，激活的合成体。

LeapsTuple 是元组实现类如图 4-3 所示，包含了 LeapsFactHandle、Activation 和 LeapsRule 对象的引用，并且提供了这些属性的读/写方法。

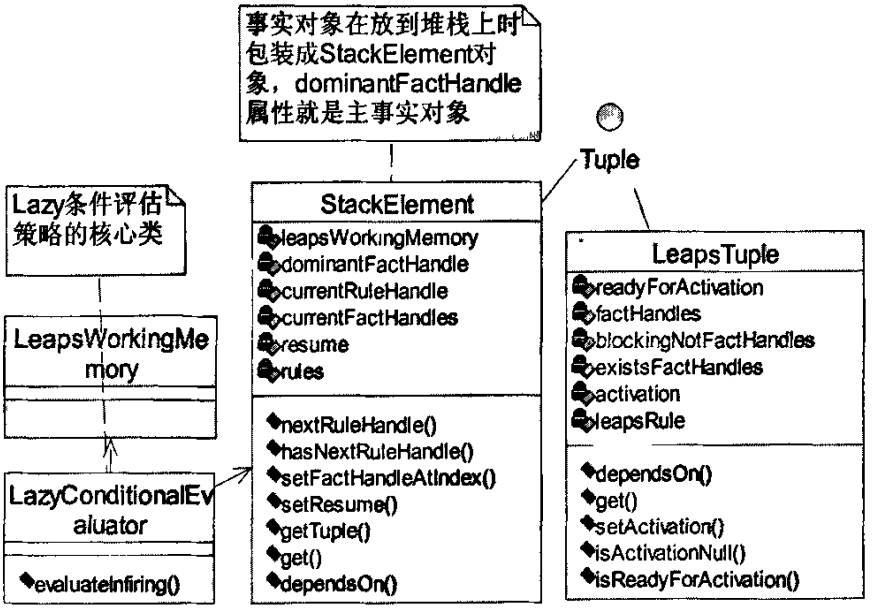

图 4-3 Lazy 条件评估和元组类型

readyForActivation 属性：标志是否满足激活的条件。当且仅当该元组关联的 LeapsRule 为空或者 LeapsRule 没有 exist 列或 not 列时为真，否则为假。

blockingNotFactHandles 属性：标志 LeapsRule 中是否存在不满足匹配的 not 条件。

existFactHandles 属性：标志是否含有 exist 操作的事实对象。

isActivationNull 方法：该方法标志 activation 是否为 null，用以表示是否处于激活状态，如果是，那么在进行匹配时应该钝化它。LeapsTuple 的设计满足工组内存的需求。

dependsOn 方法：查看元组是否依赖某一个事实对象。其示意性代码如下：

```
public boolean dependsOn(final FactHandle handle) {
    for (int i = 0, length = this.factHandles.length; i < length; i++) {
        if (handle.equals(this.factHandles[i])) {
            return true;
        }
    }
    return false;
}
```

4.4 Lazy 条件评估实现

Lazy 条件评估功能是由 LazyConditionalEvaluator 类实现的，其实现类图如图 4-3 所示，事实对象在声明进入堆栈时包装成 StackElement 对象，所有的评估计算都反映在 StackElement 元素中。下面对 StackElement 的主要属性和方法做详细说明。

dominantFactHandle 属性：StackElement 类的主要属性，用来标识主事实对象，当事实对象被声明进入工作内存时，被包装成 StackElement 放到堆栈中。

currentRuleHandle 属性：在规则库中可能存在多条规则与主对象相关联（即规则以来于某类型的事实对象），在执行规则时，需要将这些规则依次进行评估，该属性即保留的当前评估的规则的引用。

currentFactHandles 属性：与 currentRuleHandle 关联的规则的条件依赖的事实对象集合。

resume 属性：表示是否该元素是否处于正常状态。主要用来判断主事实对象和当前规则是否满足匹配关联关系，如果不能满足那么进行下一条规则。

setFactHandleAtIndex 方法：设置当前规则 currentRuleHandle 关联的事实对象。

setResume 方法：设置该元素是否处于正常状态。与 setFactHandleAtIndex 方法一样都是仅在用于评估过程中访问的方法。

getTuple 方法：如果当前规则 currentRuleHandle 得到完全满足，那么依据信息创建元组并把元素的执行交给议程。

hasNextRuleHandle 方法：判断是否还存在与主事实对象关联的规则。

正常情况是这样的：事实对象是在上一次执行规则之后被声明进入工作内存的，这仅需要判断与主对象关联的规则集合是否为空即可。但也可能存在这样的情况：引擎在执行规则之后，仍有对象处于堆栈中，例如在被执行规则的结论撤销了某对象，那么该对象第二次进入堆栈。这时只需要执行规则之后添加的规则即可，所以需要过滤调已经执行的规则。示意性代码如下：

```
01long levelId = leapsWorkingMemory.getIdLastFireAllAt();
02if (dominantFactHandle.getRecency() >= levelId) {
03   hasNext = this.rules.hasNext();
04} else {
05   boolean done = false;
06   while (!done) {
07      if (this.rules.hasNext()) {
08         if (((LeapsRuleHandle)((TableIterator)rules).
                              peekNext()).getRecency() > levelId) {
09            hasNext = true;
10            done = true;
11         } else {
12            rules.next();
13         }
14      } else {
15         hasNext = false;
16         done = true;
17      }
18   }
19}
```

L2 用于判断该对象是否是在最后一次执行规则之后进入堆栈的。当主对象在执行规则之后仍然在堆栈中，那么需要将执行规则之前的规则忽略掉，L8 是判断该规则是否是在执行规则之后添加的。

Lazy 条件评估类 LazyConditionalEvaluator 仅有一个方法用来在引擎执行规则过程中进行条件评估，该方法是 evaluateInfiring，其示意性代码如下：

```
01TableIterator[] iterators = new TableIterator[numberOfColumns];
02...
03boolean skip = token.isResume();
04if (!currentIterator.isEmpty()) {
```

```
05   if (skip && currentIterator.hasNext() &&
        !currentIterator.peekNext().equals(token.get(i))) {
06      skip = false;
07   }
08}
09boolean localMatch = false;
10LeapsTuple leapsTule = null;
11localMatch = leapsRule.getColumnConstraintsAtPosition(idx).
                                                    isAllowed(...);
12if (idx == (numberOfColumns-1)) {
13   leapsTuple = stackElement.getTuple();
14   evaluateNotExistsOnly(leapsTuple,leapsRule,workingMemory);
15}else{
16   idx++;
17}
18workingMemory.assertTuple(leapsTuple);
```

假定当前规则有 n 个列，那么就创建 n 个事实表迭代器，如代码 L1，然后依次从事实表中截取与规则相关联的事实集合，代码这个部分省略。L3-L8 行代码是判断哪些事实满足当前规则的匹配，如果满足就不需要在进行评估。L11 行是进行 α 和 β 评估，如果当前规则的最后一个模式也得到满足，那么将创建并传播元组，否则将进行下一个模式的评估，如代码 L16。

4.5 议程实现

当事实完全匹配规则时，规则将会处于激活状态且被置到议程上。规则引擎将议程元素定义为 AgendaItem，AgendaItem 类图如图 4-4 所示，AgendaItem 通过实现 Activation 接口以满足了激活的要求。其主要属性方法如下：

activated 属性：标志议程元素是否处于激活状态，如果是则可以被规则引擎执行，如果否则不能被执行，当有元素出队列时，将会导致议程元素变为非激活状态。

index 属性：当前激活所处位置的索引序号，可以用作元素出队。

justified 属性：用来转载激活和事实对象关联关系属性，用于逻辑声明时建立对象之间的逻辑依赖关系。

议程 LeapsAgenda 实现类如图 4-4 所示，设计提供了重要的执行规则（激活）的方法 fireActivation，在 fireActivation 方法中调用父类

DefaultAgenda 的 fireActivation 方法，执行置于议程上的规则，即是激活 Activation，最终执行交给 Consequence.evaluate 方法，借助于 KnowledgeHelper 执行规则。

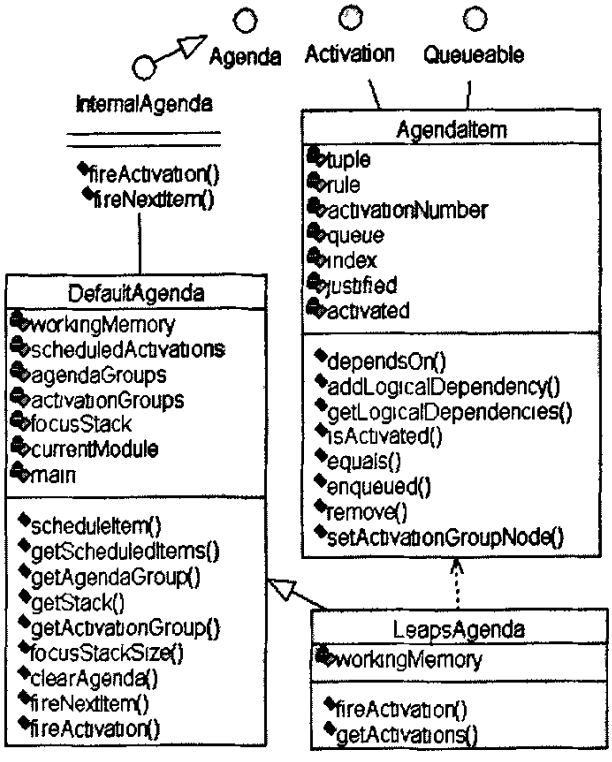

图 4-4 议程元素类图

议程执行规则示意性下代码所示，请参考图 3-8 议程执行激活时序图和工作内存实现一节中的规则执行代码。

```
1LeapsAgenda.fireActivation(){
2 .....
3 activation.getRule().getConsequence().
4 evaluate(knowledgeHelper,this.workingMemory );
5 .....
6}
```

议程类 LeapsAgenda 中主要属性和方法如下。

workingMemory 属性：议程关联的 LeapsWorkingMemory 对象，提供议程所需要的其他非关联属性和数据。

fireActivation 方法：该方法主要是，如果规则是 Query，那么将 Query

的执行结果缓存到工作内存，否则，将执行权通过父类最终交给Consequence.evaluate方法。

4.6 规则库功能实现

Leaps规则库是一个leapsRule容器，实现类是通过保留了一个Map引用实现的，并且提供了创建新的workingMemory方法，和添加/删除rule的方法。如图4-5规则库类图。

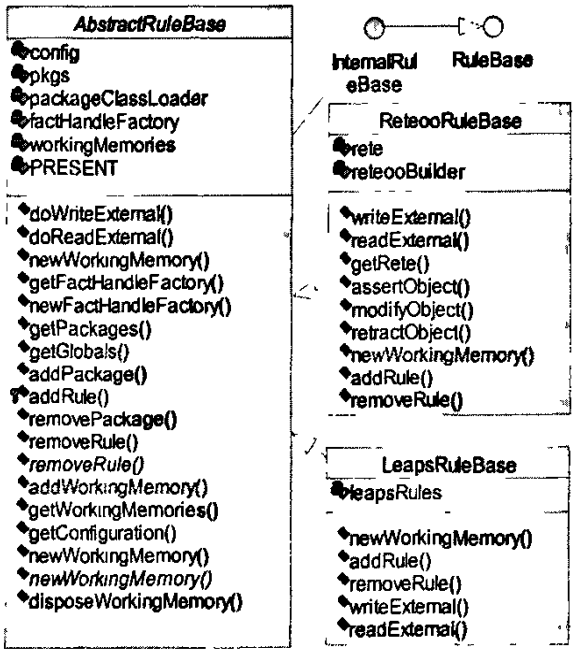

图4-5 规则库类图

Leaps规则库主要方法如下：

newWorkingMemory方法，构建一个新的workingMemory，可以通过设定boolean型参数来指定是否与规则库保留弱关联关系。通常情况下，先有ruleBase，然后才有与其相关联的workingMemory。这是一个廉价的操作。

addRule方法，添加规则到leapsRuleBase，首先需要调用Leaps规则转换器组件将Rete算法中的Rule转换成LeapsRule规则，然后将LeapsRule添加到每一个与ruleBase相关联的workingMemory中，并且调用fireAllRules方法。其示意性代码如下，首先将Rete规则保留在规则库

（L2），然后通过 LeapsRule 转换组件获得相关的 LeapsRule 列表（L3），接下来是将新添加的规则传播到其他已存在的工作内存（L5-L7）和重新执行规则（L8-L10）。LeapsWorkingMemory.fireAllRules 方法请参考工作内存实现一节代码。

```
01LeapsRuleBase addRule(final Rule rule){
02   super.addRule(rule);
03   List<LeapsRule> rules = LeapsRuleTransformer.processRule(rule);
04   this.leapsRuleMap.put(rule, rules);
05   for (Iterator it =this.getWorkingMemories().iterator(); it.hasNext();) {
06      ((LeapsWorkingMemory) it.next()).addLeapsRules(rules);
07   }
08   for (final Iterator it = this.getWorkingMemories().iterator(); it.hasNext();) {
09      (LeapsWorkingMemory) it.next().fireAllRules();
10  }
11}
```

removeRule 方法，从 leapsRuleBase 中删除一条规则，从每一个与 ruleBase 相关联的 workingMemory 中删除 LeapsRule。

下列方法是从父类 AbstractRuleBase 中继承过来的方法，针对部分方法做着重说明：

disposeWorkingMemory 方法：解除规则库和工作内存的弱关联关系。

创建新工作内存的代码如下代码所示，首先创建一个新的工作内存（L2），依次将当前规则库中的规则依次添加入新创建的工作内存（L3-L5），最后依据形参建立规则库和新创建的工作内存（L6），维护规则库和工作内存的关联关系由父类负责。

```
1LeapsRuleBase.newWorkingMemory(final boolean keepReference) {
2   LeapsWorkingMemory workingMemory = new LeapsWorkingMemory(this);
3   for ( final Iterator<List> it = this.leapsRuleMap.values().iterator(); it.hasNext(); ) {
4         workingMemory.addLeapsRules(it.next());
5   }
6   super.addWorkingMemory(workingMemory,keepReference);
7   return workingMemory;
8}
```

addRule 方法：添加规则到规则库中，设计没有提供单条规则添加进入规则库中的方法，如果要这么做，就只有将该规则包装成一个 package 然后将 package 添加入库。

addWorkingMemory 方法：手动创建规则库和工作内存的弱关联关系。

4.7 工作内存功能实现

在 Leaps 算法中 workingMemory 是规则和事实对象发生作用的容器，也是应用程序和规则引擎交互的地方，设计中直接继承并扩展 AbstractWorkingMemory 类。LeapsWorkingMemory 类图如图 4-6 所示。

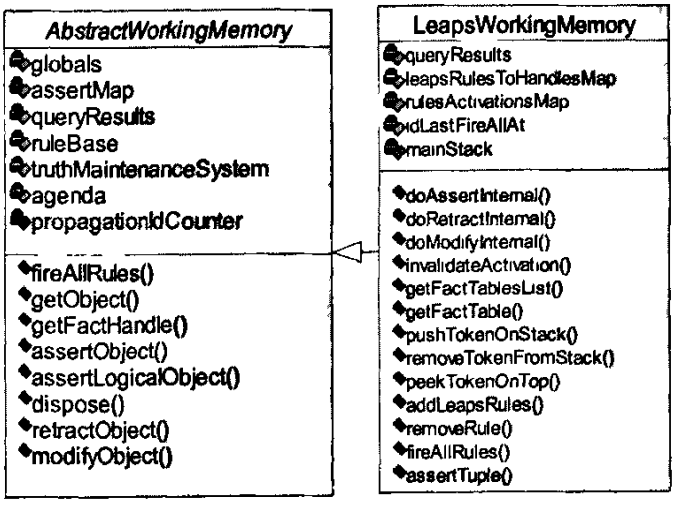

图 4-6 工作内存结构类图

其主要属性和方法如下：

mainStack 属性，用于装载声明进入在工作内存的事实对象，当引擎执行规则之后，那么堆栈中的事实对象将从堆栈中弹出，但是事实对象会保留在事实表中。

idLastFireAllAt 属性，规则引擎从主对象起开始和关联的规则进行模式匹配，当找到与事实对象完全模式匹配的规则时，就记录该位置，等执行完毕规则的 activation 时，从记录的位置开始继续下面的规则的模式匹配。

doAssertInternal()方法，将一个事实对象声明入工作内存。

doRetractInternal()方法，从工作内存撤销一个事实对象。

doModifyInternal()方法，修改工作内存中的一个事实对象，修改操作是撤销和声明的组合。

getFactTablesList(Class c)，依据参数 c 获得相应事实对象的事实表

factTable 集合。比如说一个事实对象实现了多个接口，引擎可以依据不同接口都可以找到这个对象。

pushTokenOnStack()，将事实对象包装成 Token 并压入堆栈。

removeTokenFromStack()，从堆栈中删除一个 Token。

peekTokenOnTop()，通过事实对象的句柄 factHandle 从堆栈中弹出一个元素。

addLeapsRules()，将 LeapsRule 添加到 workingMemory 中。

removeRule()，从 workingMemory 中删除一条规则 LeapsRule。

工作内存的核心功能是声明对象、撤销对象、修改对象、执行规则，下面设计将详细声明对象、撤销对象和执行规则这两个操作的实现。

I 声明对象

声明对象如下代码所示，首先将事实对象压入堆栈（L2）和添加到相关事实表（L4），然后从事实表中获取元组信息（L5），并进行迭代处理。处理 not 条件并且在必要时候钝化激活（L10，详见稍后代码）；处理 exist 条件并且在必要时候声明元组到工作内存，对于满足完全匹配的元组则构建激活并置入议程上（L14）。

```
01LeapsWorkingMemory.doAssertInternal(InternalFactHandle factHandle,
                                      Object object,PropagationContext context) {
02  this.pushTokenOnStack(factHandle, new StackToken(this, factHandle,context));
03  ……
04  factTable.add(factHandle);
05  for (final Iterator tuples = factTable.getTuplesIterator(); tuples.hasNext();) {
06     LeapsTuple tuple = (LeapsTuple) tuples.next();
07     if (!tuple.isActivationNull()) {
08        ColumnConstraints[] not = tuple.getLeapsRule().getNotColumnConstraints();
09        ……
10      invalidateActivation(tuple);
11   }else{
12     ColumnConstraints[] exists = tuple.getLeapsRule().getExistsColumnConstraints();
13     ……
14     this.assertTuple(tuple);
15   }
16}
17}
```

工作内存钝化元组如下代码所示，首先将议程元素出队并将激活的

activated 属性设置为 false（L4），然后将元组的激活属性设置为 null，从真理维持系统中删除逻辑依赖的规则。

```
1LeapsWorkingMemory.invalidateActivation(final LeapsTuple tuple) {
2   Activation activation = tuple.getActivation();
3   ……
4   activation.remove();
5   tuple.setActivation(null);
6   ……
7   this.truthMaintenanceSystem.removeLogicalDependencies(
            activation, tuple.getContext(),tuple.getLeapsRule().getRule());
8}
```

声明元组主要示意性代码如下代码所示，首先为元组所关联的 LeapsRule 设置议程组（L3），将该议程组设置为聚焦状态（L4），构建议程元素（L5）并添加到议程组（L6），并将议程元素设置为激活状态（L8）。

```
1LeapsWorkingMemory.assertTuple(final LeapsTuple tuple) {
2   ……
3   leapsRule.setAgendaGroup(agendaGroup);
4   this.agenda.setFocus(agendaGroup);
5   agendaItem = new AgendaItem(
          context.getPropagationNumber(), tuple,context, rule);
6   agendaGroup.add(agendaItem);
7   tuple.setActivation(agendaItem);
8   agendaItem.setActivated(true);
9}
```

II 撤销对象

撤销对象采用了两种处理方式，请参考 3.4 小节。其实现详细介绍如下。

方案 I

方案 I 示意性代码如下，首先删除事实表中的事实对象，如代码 L02-L04；然后删除已经处于激活状态的元组，如果元组处于激活状态那么需要钝化它，如代码 L05-L09；最后删除该事实对象关联的元组信息，并重新获取与该事实关联的处于激活状态的元组，主要是由于撤销对象而是某些规则由原来的不完全匹配变成完全匹配，并传播这些元组，如代码 L10-L13。

```
01doRetractInternal(){
02   for (final Iterator it = this.getFactTablesList() {
03      ((FactTable) it.next()).remove(factHandle);
04   }
05   Iterator tuples = ((LeapsFactHandle) factHandle).getActivatedTuples();
06   for (; tuples != null && tuples.hasNext();) {
07      LeapsTuple tuple = (LeapsTuple) tuples.next();
08      removeFromQueryResults(tuple.getLeapsRule().getRule().getName(),tuple);
09   }
10   it = ((LeapsFactHandle) factHandle).getNotTupleAssemblies();
11   it = ((LeapsFactHandle) factHandle).getExistsTupleAssemblies();
12   for(){...}
13   for(){ ... assertTuple(tuple); ... }
14}
```

方案 II

方案 II 撤销对象示意性代码如下，引擎撤销对象时，将对象标志为由于撤销操作进入堆栈的，如代码 L3，StackElement 构造器的第 4 个参数。然后将对象压入栈底。

```
1doRetractInternalShadows(InternalFactHandle handle,
2                                  PropagationContext context){
3   StackElement element = new StackElement(this,handle,context,true);
4   pushTokenOnStack(handle, element);
5}
```

在执行规则时，首先将首先对消堆栈中存在的第一次由于声明进入堆栈第二次由于撤销进入堆栈的对象。其示意性代码如下：

```
for(int i = 1; i< mainStack.size(); i++){
   element = mainStack.get(i);
   for(int j = 0; j< i; j++){
      currentElement = mainStack.get(j);
      if(element != null && currentElement != null){
         if(element.getDominantFactHandle().
               equals(currentElement.getDominantFactHandle())
               && (element.isRetraction() == false
               && currentElement.isRetraction() == true)){
            mainStack.remove(i);
            mainStack.remove(j);
         }
      }
   }
}
```

执行引擎首先判断事实对象是否由于撤销操作进入堆栈的，如果是那么执行撤销操作。其示意性代码如下：

```
if(element.isRetraction() == true){
  doRetractInternal(element.getDominantFactHandle(),
            element.getPropagationContext());
  continue;
}
```

III 规则执行

执行规则是工作内存的重要的功能，其示意性代码如代码 4-9 所示，首先从堆栈中取得栈顶元素 token（L3），并获取关联的事实对象 fact，如果 fact 没有与之关联的规则，那么就从堆栈中弹出（L9）。如果有与之关联的规则，那么依次进行 Lazy 条件评估（L15），将满足完全匹配的元组放到议程上，执行议程上的激活（L21-L23）。

```
01LeapsWorkingMemory.fireAllRules(){
02   while (!this.mainStack.empty()) {
03      StackToken token = this.peekTokenOnTop();
04      while ( !done ) {
05        if ( !token.isResume() ) {
06          if ( token.hasNextRuleHandle() ) {
07            token.nextRuleHandle();
08          } else {
09            this.removeTokenFromStack((LeapsFactHandle)
10            token.getDominantFactHandle());
11            done = true;
12          }
13        }
14        if ( !done ) {
15          TokenEvaluator.evaluate(token);
16          if ( token.getDominantFactHandle() != null ) {
17            token.setResume( true );
18            done = true;
19          }
20      }
21      while ( this.agenda.fireNextItem());
22   }
23   while ( this.agenda.fireNextItem());
24   }
25}
```

工作内存将执行权交给议程（代码 4-6 的 L21 和 L23），如下代码所示，引擎将执行权由 LeapsAgenda.fireNextItem 方法递交给

LeapsAgenda.fireActivation 方法（L3），最终执行将递交给 Consequence.evaluate 方法，并借助于 KnowledgeHelpers 类和 LeapsWorkingMemory 类完成规则的执行（L8-L9）。

```
01LeapsAgenda.fireNextItem(){
02  ……
03  fireActivation(item)
04  ……
05}

06LeapsAgenda.fireActivation(){
07  ……
08  activation.getRule().getConsequence().
09  evaluate(knowledgeHelper,this.workingMemory );
10  ……
11}
```

4.8 总结

本章在 drools 规则引擎的基础上对其模式匹配算法进行了改造，主要对以下功能和组件进行了重新实现：容器和规则冲突解决策略，规则和事实对象，元组，议程，规则库功能，工组内存功能等，在实现过程中，采用类图和示意性代码相结合的方式，具体代码请参考源代码。下一章将对该 Leaps 算法实现做性能测试和分析。

第五章 Leaps 算法性能测试和分析

5.1 测试环境搭建

- 硬件环境

  CPU:AMD Sempron(tm) 2200+ 1.50GHz

  内存：1.46G

- 软件环境

  操作系统：Windows XP sp2

  Java 虚拟机：JDK1.5.0

  编辑工具：eclipse3.2

  drools 容器：drools3.0

5.2 测试用例

本测试用例[49]依据西南交通大学图书馆规则条例整理如下借阅规则，其系统设计将不做详细介绍，仅针对业务规则做简要说明。其借阅规则如下：

1. 本科生最多可以借阅 6+1 本（其中 1 是文体类图书，以下相同）。
2. 研究生可以借阅最多 14+1 本。
3. 博士最多可以借阅 29+1 本。
4. 本科，硕士，博士可以预约最多 2 本。
5. 如果有图书证处于挂失，超期，已借满状态则不可以借阅。
6. 借阅日期为一个月（30 天）。
7. 每续借一次归还日期自当前开始推迟 30 天。
8. 预约书如果到馆之后 5 日内不来馆取视为放弃。

依据需求梳理如下系统类和规则。

Book 类：系统 BO 类，用于装在系统书的数据。

BookCard 类：图书证类，用于装在学生信息，借阅信息，预约信息，并作出超期、挂失、借满处理，及其文艺类图书和普通图书的借阅数目处理。

BookCardException 类：用于标志处理各种导致不能正常借阅的异常信息。

Student 类：用于表示学生信息，主要标志类别，本科、硕士和博士。

Main 类：测试主类。

本测试用例依据需求建立了 2 个规则包（package）：InitialRules.drl 和 ReadingRules.drl，前者用于初始化借阅者的数据，比如可以借阅普通图书和文艺类图书的最大数量。后者用于判断在借阅的过程中是否符合图书借阅规则，如果不符合借阅规则则依据不同类型作出处理并给出提示。前者仅仅用于初始化数据，以后在借阅时可以使用借阅规则包，如果用于其他功能，可以装载其他规则包。

5.3 正确性测试

依据需求，分析得出规则如下表示，这里列出了部分测试用例匹配的规则，其中规则 R1 属于 InitialRules.drl，其作用是初始化本科生图书证初始化数据。规则 R7 和 R8 属于 ReadingRules.drl，规则 R7 用于判断图书证是否处于挂失状态，规则 R8 用于捕获异常对象，撤销图书证实体对象。

```
rule "R1"
    no-loop true
    salience 100
    when
        bc:BookCard(owner:owner,state==BookCard.NORMAL)
        s:Student(kind==Student.GRADUATE)
    then
        bc.setLiteraryMax(1);
        bc.setReadingMax(7);
        bc.setBookingMax(2);
        bc.setBookingDelay(5);
        bc.setRenewDelay(30);
        System.out.println("R1");
end
```

```
rule "R7"
  no-loop true
  when
    bc:BookCard(owner:owner,state==BookCard.LOSS)
  then
    BookCardException exception = new
BookCardException("the card is loss.");
    bc.setException(exception);
    assert(exception);
    System.out.println("R7");
end


rule "R8"
  no-loop true
  when
    bc:BookCard(owner:owner)
    exists(BookCardException())
  then
    retract(bc);
    System.out.println("R8");
end
```

测试用例数据如下，依次是学生信息和该学生已经借阅的图书信息。

```
Book book1 = new Book();
book1.setBookid("TPJ5623");
book1.setName("JAVA与模式");
book1.setLiterary(false);
Book book2 = new Book();
book2.setBookid("TPJ5696");
book2.setName("精通Hibernate");
book2.setLiterary(false);
Book book3 = new Book();
book3.setBookid("WY8956");
book3.setName("资治通鉴");
book3.setLiterary(true);
```

```
Student s = new Student();
s.setStudentid("04041222");
s.setName("刘飞翔");
s.setKind(Student.GRADUATE);
```

如下是图书证信息，其中 L6-L7 是关联图书证已经借阅图书信息和学生信息。

```
1Student student = Students.getStudent();
2List<Book> books = Books.getBooks();
3BookCard bookCard = BookCards.getBookCard();
4bookCard.setBookCardid("Y020040812");
5bookCard.setState(BookCard.NORMAL);
6bookCard.setReadingBooks(books);
7bookCard.setOwner(student);
8bookCard.refresh();
```

下面是声明对象和执行规则代码。

```
workingMemory.assertObject(student);
FactHandle fh = workingMemory.assertObject(bookCard);
workingMemory.fireAllRules();
```

运行代码测试用例之后控制台打印如图 5-1 信息。

```
R1
2007-05-14 02:47:42,546 INFO [org.drools.leaps.LeapsWorkingMemory]
---------------------- 初始化数据 --------------------------
BookCard = (
        bookCardid = Y02004D812
        owner = 刘飞翔
        message =
        literaryMax = 1
        literaryCount = 1
        literaryFull = true
        readingMax = 7
        readingCount = 3
        readingFull = false
        bookingingMax = 2
        bookingCount = 0
        bookingFull = false
        state = 0
        renewDelay = 30
        bookingDelay = 5
       )
```

图 5-1 初始化数据控制台信息

由于添加学生是本科生，故依据规则 R1 初始化数据，得到数据与规则 R1 相符，故取得了正确的规则匹配。用例将图书证设置为挂失状态后，然后执行，则控制台打印了如图 5-2 信息。

```
R7
2007-05-14 03:21:21,078 INFO [org.drools.leaps.LeapsWorkingMemory]
R8
the card is loss.
```

图 5-2 图书证挂失控制台信息

该图书证处于挂失状态，所以执行规则 R7 和 R8，并打印了提示信息，由以上信息知事实和规则取得了正确的匹配。

5.3 效率和内存测试

依据上述测试用例，选取了两种情况进行了 10 次测试，得到如下表数据并且计算其平均值作为分析参考依据。(内存单位是字节，时间单位是毫秒)。由于测试用例中的规则较少，且较简单，所以匹配时间和执行时间可能少于 1ms，因此这里将执行时间放大 10 倍。

表 5-1 不带有 exist 条件的测试结果

	Leaps 算法			Rete 算法		
	内存使用	匹配时间	执行时间	内存使用	匹配时间	执行时间
1	1380648	105	156	1057376	421	47
2	1380648	78	125	1057376	172	62
3	1380648	79	125	1057376	219	31
4	1380648	78	109	1057376	171	16
5	1380648	62	125	1056448	203	16
6	1365464	62	110	1056448	202	16
7	1365464	123	125	1056448	172	47
8	1365464	72	102	1056448	203	15
9	1365464	87	123	1056448	140	16
10	1365464	92	125	1056448	234	16
平均	1373056	83.8	122.5	1056819.4	213.7	28.2

表5-2 带有exist条件的测试结果

	Leaps算法			Rete算法		
	内存使用	匹配时间	执行时间	内存使用	匹配时间	执行时间
1	1073240	71	141	1222984	196	28
2	1073008	50	138	1222984	156	26
3	1073240	62	120	1222984	142	34
4	1073240	40	162	1224760	163	16
5	1073240	39	123	1220408	171	18
6	1073240	60	140	1222984	162	46
7	1073240	68	102	1222984	162	15
8	1073240	51	152	1222984	158	34
9	1073240	53	126	1222984	140	26
10	1073240	59	128	1222984	143	48
平均	1073216.8	55.3	133.2	1222904	159.3	29.1

5.4 结果分析

表5-1是进行匹配没有exist条件的规则的测试结果，表5-2是匹配带有exist条件的规则的测试结果。

由表5-1得出，在执行规则没有包含exist条件时，使用Leaps算法进行模式匹配数据为，匹配时间83.8，执行时间122.5，内存使用1373056，使用Rete算法进行模式匹配数据为，匹配时间213.7，执行时间28.2，内存使用1056819.4，由此可知，模式匹配时间效率提高了39.2%，执行规则时间则增加为原来的4.34倍，内存使用也增加了29.9%。

由表5-2得出，在规则包含exist条件时，使用Leaps算法进行模式匹配数据为，匹配时间55.3，执行时间133.2，内存使用1073216.8，使用Rete算法进行模式匹配数据为，匹配时间159.3，执行时间29.1，内存使用1222904，由此可知，模式匹配时间效率提高了34.7%，执行规则时间变为原来的4.57倍，内存使用降低了12.2%。

在这两种情况下，模式匹配效率都有所提高，分别是39.2%和34.7%，该结果与理论结果一致。

在这两种情况下，规则执行时间却大大增减，分别是原来的 4.34 倍和 4.57 倍，这时由于在 Leaps 算法采用了 Lazy 条件评估策略所致，后者在需要进行 not 或者 exists 条件评估，所以所需运行时间将会更长，与预期结果一致，但是运行时间与 Rete 算法的运行时间相比大大增加，这与理论结果相去甚远。

在这两种情况下，内存使用或有增加或有减少，这不是与理论结果一致。

在执行规则没有包含 exist 条件时，模式匹配效率明显提高，是由于 Leaps 算法不需要创建和传播保存中间匹配结果的元组，从而使得模式匹配效率得到提高。从理论上将内存使用也会减小，执行规则时间有所增加，但不会太多，但是这两个结果都有很大出入，由于影响结果的因素较多，暂时无法精确确定造成这一结果的原因，只能从理论角度推断最可能的原因，造成内存使用大的原因可能是规则转换组件和 Lazy 评估组建的使用，造成执行时间变大的原因可能是 Leaps 算法 Lazy 评估策略，Lazy 评估策略是在执行的时候进行评估的，或者是实现的原因。

在规则中包含 exist 条件时，模式匹配效率明显提高，内存使用降低了，规则执行时间仍旧是大大增加，最后一项测试与理论仍有出入。

影响测试结果还有其他因素，包括软件环境、硬件环境、规则引擎本身及相关影响等。

I 硬件环境

硬件环境影响，包括 CPU 主频率，内存容量等，由于运行有少量规则的测试用例时，所需要的 CPU 资源和内存极少，这将导致测试结果不明显。

II 软件环境

操作系统除了运行测试用例之外，还启动了系统进程和少量的应用软件进程，这些进程在动态的使用 CPU 和内存，所以会对测试结果的精度产生影响。

III 规则引擎的影响

规则引擎所支持的规则数目，以及本身的算法的实现都会对效率的测试产生影响。还有测试用例的选取，包括测试用例的大小，数量等。

IV 设计实现的影响

由于采用 Java 面向对象的设计语言，程序在运行时，需要由虚拟机交给 CPU 再执行，所以设计实现的方法也会对效率、内存等方面产生较大的影响。

本设计尽可能的保持测试环境一致的情况下，依据试验结果，无论在规则是否有 exist 条件下，其模式匹配的效率都得到提高，对于其他一些负面影响，例如内存使用和规则执行方面，还需要做后续的研究和试验。

总结与展望

随着企业级软件商业逻辑越来越趋于多样化和复杂化，规则引擎必将发挥越来越重要的作用，规则引擎的效率将称为开发商越来越关心的问题，对于规则引擎中模式匹配效率的研究也将趋于深入，本文正是进行了多种算法的互相补充从而提高效率的研究。

1、本文主要完成工作：

（1） 通过比较两种模式匹配算法 Rete 算法和 Leaps 算法，找出可以将 Rete 算法效率提高的方法。

（2） 以 drools 规则引擎作为容器，实现了 Leaps 算法对 Rete 算法的模式匹配的替换。

（3） 通过测试用例对 Leaps 算法的模式匹配效率进行测试，得出效率得到优化的结论。

2、今后的研究工作：

针对规则引擎效率 Rete 模式匹配算法的研究，将是一个重要的课题，如果确实存在一定环境下，可以对 Rete 算法的模式匹配效率进行优化，那么将具有重要的现实意义，要达到这一目标还需要做进一步的研究和试验。

（1） 优化两种算法结合的模型，取得效率提高理论依据。

（2） 用面向对象方法编写高质量的程序，从而避免出现内存泄漏等低级失误。

（3） 做更加合理严谨更加切合实际的测试，包括正确性测试和匹配效率测试。

致谢

能够顺利完成毕业设计，首先，我要感谢我的导师楼新远副教授。从论文的选题、工作的开展到论文的完成，楼老师一直给予了莫大的支持和帮助。楼老师广博的专业知识、敬业的工作精神、严谨的治学态度和丰富的科研经验都使我受益匪浅。

感谢我的同门师兄妹：黎则良、陈小云、王磊、杨雪、杜林春、孟盈，还有实验室的兄弟姐妹们，当我遇到困难的时候，是你们给予了我无私的帮助。这里良好的学习和研究氛围，为我的成长提供了一片有益的土壤。

感谢我辛勤的父母，你们的鼓励和长期的支持是我永远的动力。

感谢在研究生阶段给我上课的全体老师。

最后，感谢将要评审我的论文的专家们，谢谢你们宝贵的意见，我一定虚心接受。

参考文献

[1] Geoffrey Wiseman, A Rule Engine Primer, on Jun 19, 2006

[2] C. L. Forgy. On the Efficient Implementation of Production Systems. PhD thesis, Carnegie-Mellon University, Department of Computer Science, 1979.

[3] 作者不详,光大银行采用 ILOG 的规则引擎 JRules,中国金融电脑,2006 年第 7 期

[4] ILOG, changing the rules of business, http://www.ilog.com November 2006

[5] Joseph C. Giarratano Gary D. Riley，专家系统原理与编程，机械工业出版社 2006 年 8 月第 1 版第 1 次印刷 P326-P327

[6] Don Batory，The Leaps Algorithms，Department of Computer Science P6

[7] Mark Proctor, Michael Neale, Peter Lin, Michael Frandsen, Drools Documentation, http://www.drools.com, 2006

[8] Ajith Abraham, Rule-based Expert Systems, Oklahoma State University, Stillwater, OK, USA, 2005

[9] Joseph C. Giarratano Gary D. Riley，专家系统原理与编程，机械工业出版社 2006 年 8 月第 1 版第 1 次印刷 P79-P83, P103-P107

[10] Joseph C. Giarratano Gary D. Riley，专家系统原理与编程，机械工业出版社 2006 年 8 月第 1 版第 1 次印刷 P133-P135

[11] Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence, 19, 17--37.

[12] Joseph C. Giarratano Gary D. Riley，专家系统原理与编程，机械工业出版社 2006 年 8 月第 1 版第 1 次印刷 P326-P327

[13] Joseph C. Giarratano Gary D. Riley，专家系统原理与编程，机械工业出版社 2006 年 8 月第 1 版第 1 次印刷 P79-P82

[14] 张渊，夏清国，基于 Rete 算法的 Java 规则引擎，科学技术于工程，第 11 期，2006 年 6 月

[15] James Taylor, Business rules engines v Business rules management systems, Jun 19, 2006

[16] 彭磊，规则引擎原理分析，福建电脑，2006年第9期

[17] 胡金化, 陈彤兵, 黄秋波, 胡乃静, 一个基于动态规划的规则引擎构件技术研究，小型微型计算机系统，2006年12月

[18] 刘晓建, 陈平, RETE网络中的优化编译模式及其PVS形式验证, 计算机科学, 2003, Vol. 30NO. 6

[19] Pandurang Nayak, Comparison of the Rete and Treat Production Matchers for Soar, National Conference on Artificial Intelligence, 1998

[20] Milind Tambe, Robert Doorenbos, Allen Newell, The Match Cost of Adding a New Rule: A Clash of Views, 1992

[21] D. P. Miranker, D. Brant, B. J. Lofaso, D. Gadbois, On the performance of lazy matching in production systems, National Conference on Artificial Intelligence. Boston, MA, July, 1990

[22] Don Batory, The Leaps Algorithms, Department of Computer Science P6

[23] 刘伟, JAVA规则引擎Drools的介绍及应用, 微计算机信息, 2005, 6

[24] Java Community Process, Java Rule Engine API JSR-94, http://java.sun.com/jcp

[25] Jlisa.com, Jlisa Development Document, http://www. Jlisa.com

[26] Mark Proctor, Michael Neale, Peter Lin, Michael Frandsen, Drools Documentation, http://www.drools.com, 2006

[27] 中国建设银行UAAP项目组, 中国建设银行UAAP项目评估报告, 2006年12月

[28] Mark Proctor, Michael Neale, Peter Lin, Michael Frandsen, Drools Documentation, http://www.drools.com, 2006

[29] 作者不详, 规则引擎研究Rete算法, http://blog.sina.com.cn/zhiaijiangshuang 2007-04-18

[30] C.L. Forgy, RETE A Fast Algorithm for the Many Pattern, http://citeseer.ist.psu.edu/context/505087/0, 1982

[31] Mark Proctor, Michael Neale, Peter Lin, Michael Frandsen, Drools

Documentation, http://www.drools.com, 2006

[32] Don Batory, The Leaps Algorithms, Department of Computer Science P1 Introduction

[33] Mark Proctor, Michael Neale, Peter Lin, Michael Frandsen, Drools Documentation, http://www.drools.com, 2006

[34] Don Batory, The Leaps Algorithms, Department of Computer Science P6 3.2 Leaps Overview

[35] Don Batory, The Leaps Algorithms, Department of Computer Science P4-P10 Leaps Algorithms

[36] Don Batory, The Leaps Algorithms, Department of Computer Science P12-P14 Optimizations

[37] Nick Bassiliades, Ioannis Vlahavas, Compiling Production Rules into Event-Driven Rules Using Complex Events, Information and Software Technology 1997

[38] D. P. Miranker, D. Brant, B. J. Lofaso, D. Gadbois, On the performance of lazy matching in production systems, National Conference on Artificial Intelligence. Boston, MA, July, 1990

[39] 严蔚敏，吴伟民 数据结构（C 语言版） 清华大学出版社 1997 年 4 月第一版 P18-P43

[40] 严蔚敏，吴伟民 数据结构（C 语言版） 清华大学出版社 1997 年 4 月第一版 P44-P58

[41] Java™ 2 Platform Standard Ed. 5.0 Sun Microsystems

[42] Joseph C. Giarratano Gary D. Riley 著，专家系统原理与编程第 7.10 和 7.11 节，机械工业出版社 2006 年 8 月

[43] Don Batory, The Leaps Algorithms, Department of Computer Science

[44] Joseph C. Giarratano Gary D. Riley 著，专家系统原理与编程第 7.12 节，机械工业出版社 2006 年 8 月

[45] 阎宏 Java 与模式 电子工业出版社 2005 年 4 月第 9 次印刷 P671-P690

[46] Joseph C. Giarratano Gary D. Riley 著，专家系统原理与编程第 8.18 节，机械工业出版社 2006 年 8 月

[47] 严蔚敏，吴伟民 数据结构（C 语言版） 清华大学出版社 1997 年 4

月第一版 P58-P64

[48] JBoss.com,Logical dependency bug, justifier tuple contains retracted handles, logically depend object stays in working memory,Aug 25 2006

[49] Andrew Hunt、David Thomas,单元测试之道－Java 版,电子工业出版社 2005 年 1 月第 1 次印刷

[50] "Programming Expert Systems in OPS5", L. Brownston et al, A-W 1985

[51] "An OPS5 Primer", Sherman et al, comes with OPS5 for DOS

[52] "Rule-Based Programming in the Unix System", G.T. Vesonder, AT&T Tech J 67(1), 1988

[53] "Expert Systems: Principles and Programming", Joseph Giarratano and Gary Riley, PWS Publ 1994, ISBN 0-534-93744-6

攻读硕士期间发表的论文

[1] 刘金龙，杨雪，杜林春，王磊.SpringMVC 架构实现及控制器扩展应用研究.2006 年信息、电子与控制技术学术会议，2006.9

[2] 王磊，康智，刘金龙，楼新远.基于 VxWorks 的嵌入式软件单元测试研究.2006 年信息、电子与控制技术学术会议，2006.9

[3] 杨雪，刘金龙等.基于 JXTA 平台的同步协作系统,西南交通大学学报,2006 年 10 月

附录一 OPS5 和 CLIPS 语言简介

S5 是专门用于基于规则的产生式系统的一门编程语言，它所表达的规则是由条件和结论组成。用 OPS5 实现的系统首先检测工作内存中的规则的条件是否得到满足，如果得到满足，那么该规则的结论将被执行。

Charles L. Forgy 博士在 1977 年首次对 OPS5 的实现做了一个全面的解释，先后有 Lisp 系统和 BLISS 系统实现了该语言，其后 George Wood 和 Jim Kowalski 在 Common Lisp 中也实现了该语言。

OPS5 的许多思想对后来的其他规则语言产生了重大影响，例如：容器、指针、组合指针等。OPS5 有多个版本，例如：OPS5+, OPS83 等。

CLIPS（C Language Integrated Production System）是一门用于编写专家系统的语言，它是美国国家航空和宇宙航行局（NASA）约翰逊太空中心用 C 实现的。它是一种多范例编程语言，它支持基于规则的、面向对象的和面向过程的编程。基于规则的 CLIPS 编程语言的推理和表示能力和 OPS5 相似，但功能更强大。在语法方面，CLIPS 规则和 Eclipse、CLIPS/R2、Jess 语言的规则极为相似，CLIPS 仅支持正向链规则，不支持反向链规则。

CLIPS 面向对象的编程能力也就是指 CLIPS 面向对象语言（CLIPS Object-Oriented Language, COOL），它除了加入了许多新思想之外，还结合了其他面向对象语言的特性，如 Common Lisp 对象系统和 SmallTalk。

CLIPS 面向过程的编程语言的特征类似于 C、Ada 和 Pascal 语言，语法上类似于 Lisp。本附录参考文献[50]～[53]。

drools规则引擎模式匹配效率优化研究及实现

作者： 刘金龙
学位授予单位： 西南交通大学
被引用次数： 3次

本文读者也读过(10条)

1. 刘伟.LIU Wei Java规则引擎--Drools的介绍及应用[期刊论文]-微计算机应用2005,26(6)
2. 杨智 基于Rete算法规则引擎的研究及其实现与应用[学位论文]2007
3. 曹永亮 基于Java规则引擎的动态数据清洗研究与设计[学位论文]2008
4. 张宇.陈德礼 Drools规则引擎应用分析[期刊论文]-福建电脑2007(10)
5. 张渊.夏清国.ZHANG Yuan.XIA Qingguo 基于Rete算法的JAVA规则引擎[期刊论文]-科学技术与工程2006,6(11)
6. 郭芳.白建军 基于Rete算法的规则引擎JBoss Rules[期刊论文]-计算机时代2008(1)
7. 王重英.WANG Chongying 基于规则引擎的工作流系统设计[期刊论文]-现代电子技术2009,32(12)
8. 刘际 规则引擎在业务逻辑层中应用的研究[学位论文]2007
9. 朱昊 面向服务的规则引擎兼容模型[学位论文]2007
10. 任忠保.张艳晶.李立亚.Ren Zhong-bao.Zhang Yan-jing.Li Li-ya 基于Drools的策略体系设计[期刊论文]-计算机安全2007(8)

引证文献(4条)

1. 杨家芳.赖冬林.张丰.杜震洪.刘仁义 基于规则引擎的土地数据质量检查方法[期刊论文]-国土资源科技管理2015(2)
2. 张勇.张健.赵洁.邢春晓 基于规则引擎技术的铁路客运专线浮动票价系统研究[期刊论文]-山西大同大学学报（自然科学版） 2011(1)
3. 王兴.朱定真.苗春生 基于规则引擎的多元大气信息数据质量检查方法[期刊论文]-南京信息工程大学学报2011(3)
4. 王兴.苗春生.朱定真.李菊 规则引擎在气象资料质量控制中的应用研究[期刊论文]-计算机与现代化 2011(1)