武汉理工大学

硕士学位论文

基于Java规则引擎的动态数据清洗研究与设计

姓名：曹永亮

申请学位级别：硕士

专业：计算机应用技术

指导教师：王舜燕

20080501

摘 要

在运营管理过程中，企业积累了大量的、极为重要的电子数据。业务决策者在进行分析决策时对这些数据的依赖性日益增强，错误或冲突的数据很可能会导致错误的决策，从而给企业造成巨大的损失。因此在这些数据进入决策系统之前需要对其进行处理，以提高决策系统的可信度和可用性。

为解决上述问题，业界提出了数据清洗的解决方案，即从大量原始数据中按一定规则（领域知识规则）检测出“脏数据”并按一定的规则（清洗动作规则）修复或丢弃之。

传统的数据清洗工具存在以下不足：“脏数据”的检测和修复逻辑被嵌入到复用性差的硬编码中或依赖于灵活但低效的手工判断。当“脏数据”的定义发生变化时需要修改源代码并重新编译生成清洗软件，这在实际使用中是低效的。Java 规则引擎的出现，为基于动态、可配置规则的数据清洗方式提供了可行的技术基础。

本文介绍了规则引擎的基本原理，分析了 Java 规则引擎的工作机制及其核心算法——Rete 算法，并对一种开源的 Java 规则引擎软件包——Drools 的 API 使用方法及其规则配置文件的结构及含义做了系统地研究分析。

本文着重阐述了一种基于Drools规则引擎的动态数据清洗系统的设计方案。给出了领域知识规则和清洗动作规则的巴科斯范式定义，为规则的持久化存储提供了基础。

本文设计并实现了使用 Drools 规则引擎描述并执行清洗逻辑，能处理多种数据质量问题的动态数据清洗系统，弥补了现有数据清洗工具的不足。这种动态性主要体现在规则的持久化存储和 Drools 规则配置文件的动态更新。文中还详细介绍了系统的规则数据库设计、功能模块划分、架构和工作流程，给出了主要模块的部分代码，并对系统做出了实验性能分析。

关键词：规则引擎，动态数据清洗，Drools，数据转换

Abstract

In course of operation management, companies have accumulated a mass of vital electronic data. Decision-makers become increasingly dependent on the above-mentioned data while carrying through analysis and strategies as wrong or conflicting data will likely result in unsuccessful maneuver, which in return can breed disastrous loss. Hence it is essential that data should be processed before entering into decision-making system in view of improving its credibility and availability.

To ravel out the above-mentioned problem, experts have put forward a solution called data cleansing which refers to inspecting "dirty data" from massive data according to certain rule (domanial knowledge) and to repairing or discarding it in the light of some rule (cleansing action rule).

Traditional tools for data cleansing have the following insufficiency: it is inefficient in practice in that modification and recompilation are required for the generation of cleansing software, the reason of which is that logic for inspecting and repairing "dirty data" is embedded into code or relies on agile but inefficient manual judgment. It is no other than the appearance of Java Rule Engine that provide feasible technological foundation for people to find such a data cleansing mode that based on dynamic and configurable rules.

The thesis presented the basic principles of Rule Engine and investigated the working mechanism of Java Rule Engine and its core algorithm—Rete algorithm. The thesis also introduced a kind of open-source Java Rule Engine software package—Drools and systematically investigated its API usage, the structure and meanings of its rule configuration file.

The thesis mainly elaborated on the design scheme of dynamic data cleansing system based on Drools rule engine and investigated the BNF (Backus-Naur Form) definition of domanial knowledge and cleansing rules laying a solid foundation for the persistence of rules.

The thesis presented the design and implementation of a kind of dynamic cleansing system, which adopts Drools Rule Engine to describe and execute cleansing

logic and can deal with many kinds of problems related to data quality. This system remedied the defect that existing data cleansing tools have. The main reasons for achieving this are persistent storage of cleansing rules and dynamic update of Drools rule configuration file. The thesis detailedly presented rule database design, division of functional module, architecture, working flow of the system, some code segments of main modules. The thesis also presented the result of experimental performance's analysis.

Key Words: Rule Engine, Dynamic Data Cleansing, Drools, Data Transformation

独 创 性 声 明

本人声明，所呈交的论文是本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得武汉理工大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

签 名：黄永亮 日 期：2008.5.19

关于论文使用授权的说明

本人完全了解武汉理工大学有关保留、使用学位论文的规定，即学校有权保留、送交论文的复印件，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存论文。

（保密的论文在解密后应遵守此规定）

签 名：黄永亮 导师签名：王海亮 日 期：2008.5.19

第1章 绪论

1.1 业务规则的含义

业务规则(Business Rule)是对业务定义和约束的描述，用于维持业务结构或控制和影响业务的行为[1]。业务规则技术的基本思想是将系统处理的业务逻辑从程序代码中抽取出来，将其转变为简单的业务规则，以结构化的业务规则数据来表示业务行为，采用类自然语言来描述，并集中存储在规则库（以数据库或结构化的形式）中。业务规则由业务人员创建、实时更新和调试，业务规则之间的复杂逻辑关系由规则引擎处理[2]。业务规则技术改变了传统的、以过程形式处理业务逻辑的方式。

业务规则具有以下特性[3]：

1）原子性，每条业务规则不可再分，且只定义一种判断和操作，复杂的业务逻辑由多条规则协同处理。

2）独立性，业务规则彼此之间独立，复杂的逻辑关系由规则引擎来处理。业务规则存储在规则库中，独立于数据和程序。

3）简单性，业务规则用简单直接的类自然语言来描述，很容易被业务人员和技术人员所理解。

4）动态性，业务人员可以实时地修改业务规则，快捷地更新系统，低成本地维护系统。

5）逻辑性，业务规则至少包含条件和执行两个部分，条件是对业务数据作用的判定，执行是对业务数据的处理。

1.2 规则引擎的发展和研究现状

运营管理者对企业级 IT 系统的开发有如下的要求：尽管现代业务规则异常复杂，但为提高效率，管理流程必须自动化；市场要求业务规则经常变化，IT 系统必须依据业务规则的变化快速、低成本的更新；为了快速、低成本的更新，业务人员应能直接管理 IT 系统中的规则，而不需要程序开发人员的参与。这种情况下，项目开发人员就会碰到以下问题：有些复杂的商业规则很难推导出算

法和抽象出数据模型；软件工程要求从需求—设计—编码，然而业务规则常常在需求阶段可能还没有明确，在设计和编码后还在变化，业务规则往往嵌在系统各处代码中；对程序员来说，系统维护、更新困难，更不可能让业务人员来管理[4]。

随着商业以及市场的迅猛发展，业务逻辑的变化频率已经从以前的平均 18 个月缩短到平均 6 个月甚至更短，这意味着企业级应用的更新和升级变得越来越频繁。同时又由于业务逻辑层没有标准的框架和统一的开发方式，使系统升级变的既昂贵又麻烦。因此，迫切需要一个框架或统一的方法来开发业务逻辑层。

规则引擎的出现为开发人员解决上述问题提供了契机。规则引擎是推理引擎的一种，起源于基于规则的专家系统（详见 2.1.1）。规则引擎能够将业务逻辑从应用程序代码中分离出来，接受数据输入，解释业务规则，并根据规则做出业务决策。规则引擎的核心内容是其所采用的匹配算法，早期的匹配算法主要是索引计数匹配法。这种方法的匹配效率并不高，应用并不广泛。在索引计数匹配算法的基础上，Forge 于 1979 年提出了 Rete 算法。Rete 算法牺牲了一部分空间来提高算法的时间效率，使该算法有了广泛的应用空间。目前，几乎所有成熟的规则引擎框架的实现都是基于该算法的。但是，基于这些框架的企业级应用却并不多，原因是这些框架都有自己的实现方式，有自己的规则定义语言。并且，绝大多数规则引擎框架都是商业产品，价格非常昂贵。这种情况直到 JSR94[5] (Java 规范要求，Java Specification Request)规则引擎标准和开源规则引擎的出现才逐渐好转。

目前，满足 JSR94 标准并且比较成熟的规则引擎框架主要有 JRules、JESS 和 Drools 等。其中，JRules 是 ILog 公司的一个商业产品，价格很高；JESS 是由美国 Sandia 国家实验室分布式系统计算组成员 Ernest J. Friedman Hill 在 1995 年以应用广泛的 CLIPS 专家系统外壳为基础开发出来的，所采用的规则描述语言是类 CLIPS 描述语言，比较难懂，而且也是一个商业产品；Drools 是一个开源并且免费的框架，采用 XML 或 Drl 格式的规则描述语言描述业务逻辑，是最近才逐渐成熟起来的一个框架，还需要进一步的完善[6]。本文就是采用 Drools 规则引擎来描述并执行数据清洗逻辑。

1.3 数据清洗的应用背景

在构建业务数据库时，用户的录入错误、企业环境随时间推移的改变，都会影响所存放数据的质量。数据质量问题可以分成：单数据源中的数据质量问题和多数据源中的数据质量问题。单数据源和多数据源中的数据质量问题都还可以分成模式层和实例层的数据质量问题[7,8,9]。

单数据源中的数据质量问题的实质是违背了约束（参照性约束，完整性约束及用户自定义约束），可以分为有不符合模式约束和记录约束两种，分别对应于记录质量问题和记录集质量问题。记录质量问题主要有：字段值缺失、字段值不在值域范围内、单字段中包含多个字段值、字段值不符合业务规则约束、引用字段值缺失或错误。可以在数据转换中发现和处理记录质量问题。记录集质量问题主要有：记录集中存在相似重复记录和记录集之间关联不符合关联约束条件。记录集质量问题无法在转换过程中发现，需要在转换之前进行特殊的处理。多数据源中的质量问题主要有：模式结构不一致、命名不一致，编码格式不一致，多数据源中存在冗余重叠记录。多数据源中的质量问题是数据转换主要解决的问题，而不是数据清洗的目标。

数据清洗指的是在大量原始数据中使用一系列的逻辑规则和领域知识检测出“脏数据”并修复或丢弃之。数据清洗一般是作为数据抽取、转换和加载(ETL, Extraction-Transformation-Loading) 工具的一个功能来实现。也存在大量数据清洗专用工具，它们被称为数据质量工具。

1.4 数据清洗的研究现状

数据清洗的相关研究最早可追溯到 1959 年[10]。从那时起，汇总来自不同数据源的数据一直被认为是一个重要而困难的问题。近年来，随着信息化的进展，人们开始系统地研究数据清洗问题。主要成果可分类如下：

1）特殊领域的数据清洗

特殊域清洗工具主要解决某些特定领域的数据清洗问题，例如姓名和地址数据[5]。这是目前研究得较多的领域，也是应用最成功的。如商用系统：Trillinm Sowtfare、Pure Integrate(Oracle)、Quick Address(QAS Systems)等。它们用一个匹配工具抽取被清洗的数据，并把姓名和地址信息转换成单个标准元素、有效的

街道名、城市和邮政编码。它们具体表现为使用一个大的预定义规则库来处理在清洗过程中发现的问题。

2）与领域无关的数据清洗

与领域无关的数据清洗研究主要集中在清洗重复的记录上[11]，其主要工具包括：Data Blade Module, Choice Maker, Integrity, Megre/Pugre Library(Sagent/QM Sotfware), Match IT(Help IT Systems), Master Megre(Pitney Bowes), Data Cleanser(EDD)等。

3）数据抽取、转换和加载 (ETL, Extraction-Transformation-Loading) 工具中的数据清洗

数据抽取、转换和加载是数据仓库系统中数据处理的关键操作。ETL 就是根据数据处理的需要，将源数据对象经过转换后加载到目标数据对象中[12]。很多商业工具在多方面支持数据仓库的 ETL 过程，如 Ardent 的 Data Stage、Microsoft 的 Data Transformation Service、SAS 的 Warehouse Administrator 和 Informatica 的 Power Mart 等。这些工具在关系数据库系统上建立一个存储器，以统一的方式管理关于数据源、目标模式、映射、脚本程序等所有元数据。通过文件、关系数据系统、以及标准接口(如 JDBC、ODBC 等)，从数据源中抽取模式和数据。并提供图形化的界面来定义数据转换逻辑规则。这些 ETL 工具提供了大量数据的抽取、转换和加载操作，但只对数据清洗提供有限支持。ETL 工具并不是完全针对数据清洗而设计的。

传统的数据清洗软件采用的策略有：1）将检测“脏数据”和进行修复操作的逻辑嵌入到系统各处的程序代码中；2）让用户进行灵活但低效的手工判断。前者当“脏数据”的定义发生变化时，需要重新编译数据清洗软件，这在实际使用中是不可行的；后者则不能保证人工判断的完整性和准确性[7]。

文献[12]为本文指出了一个可能的方向。该文提出了数据清洗规则的概念，给出了基于规则的数据清洗思路。

本文的研究课题就是在上述背景下提出的，采取的方法是将一种开源的 Java 规则引擎——Drools 应用于数据清洗，以此实现清洗规则的动态配置和及时更新，从而提高了数据清洗系统的可复用性、准确性和可扩展性。

1.5 本文主要研究内容

1）研究分析了规则引擎的基本原理；

2）研究分析了 Java 规则引擎的工作机制；

3）研究分析了 Java 规则引擎的核心算法——Rete 算法；

4）研究分析了一种开源的 Java 规则引擎软件包——Drools，并系统地研究分析了其 API 的使用方法和规则配置文件的结构和含义；

5）研究分析了数据质量和数据清洗的相关问题；

6）研究分析了领域知识和清洗动作规则的巴科斯范式(BNF, Backus-Naur Form)定义，阐述了一种基于 Drools 规则引擎的动态数据清洗系统的设计方案；

7)详细介绍了基于 Drools 规则引擎的动态数据清洗系统的规则数据库设计、系统功能模块划分、系统结构和工作流程，并给出了主要模块的部分代码和系统实验性能分析结果。

1.6 本文的组织形式

本文的主要内容有 5 章，第 1 章是绪论，主要分析了本课题的研究背景，同时综述了数据清洗技术的研究现状及本文所做的主要工作。

第 2 章分析了基于规则的专家系统的原理，Java 规则引擎的工作机制和 Rete 算法的原理。对一种开源 Java 规则引擎——Drools 的 API 的使用进行了系统分析和介绍，并结合实例分析了其规则配置文件的结构及含义。

第 3 章详细介绍了“脏数据”的概念及数据质量的概念及分类，并详细介绍了数据清洗的概念、模型和流程。

第 4 章阐述了基于 Drools 规则引擎的动态数据清洗系统的设计方案。首先给出了系统使用的两种规则。接着详细介绍了系统的详细设计。

第 5 章对全文进行了总结，并对系统需要进一步所做的工作进行了展望。

第 2 章 Java 规则引擎的研究

2.1 规则引擎的原理

规则引擎是基于规则专家系统(RBES, Rule-Based Expert System)的重要组成部分。下面先简要地介绍一下 RBES，以便深入地了解 Java 规则引擎。

2.1.1 基于规则的专家系统简介

RBES 由三部分组成：规则库/知识库(Rule Base/Knowledge Base)、工作内存/事实库(Working Memory/Fact Base)和推理引擎(Inference Engine)。它们之间的关系如图 2-1 所示。

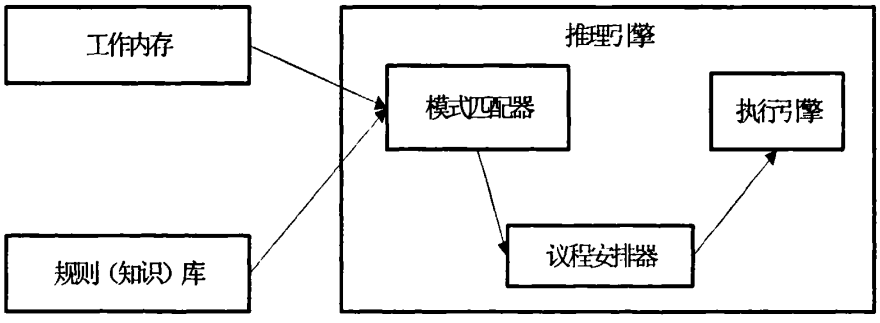

图 2-1 基于规则的专家系统构成

规则(知识)库是中心数据库，存储着各种模拟人类问题求解的产生式规则。每一条规则分为两部分：前件部分和后件部分。前件(Antecedent)又称条件部分、模式部分或左部(Left-Hand-Side，LHS)，是规则触发的条件。单独的一个条件称为条件元素或一个模式；后件(Consequent)又称右部(Right-Hand-Side，RHS)，是规则触发时将要执行的一系列动作。

工作内存是应用于规则（知识）库的全局数据库，它保存系统的当前状态。

推理引擎通过决定哪些规则满足工作内存中的事实或目标，而授予规则优先级，并将满足事实或目标的规则加入议程安排器[13]。如图 2-1 所示，推理引擎

又可分成三部分：模式匹配器、议程安排器和执行引擎。模式匹配器通过比较事实和规则的模式部分，决定选择执行哪个规则，何时执行规则。它是基于规则的推理引擎的关键，并决定了推理引擎的推理效率；议程安排器是由推理引擎创建的一个规则优先级表，这些规则都匹配工作内存中的事实。如果同时有多个规则和事实匹配，则优先级最高的先被触发。被触发规则的动作可能会产生新的事实，这些新的事实也会被加入工作内存；执行引擎负责执行议程安排器中的规则和其他动作。

和人类的思维类似，推理引擎存在两种推理方式：演绎法(Forward-Chaining)和归纳法(Backward-Chaining)。演绎法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。而归纳法则是根据假设，不断地寻找符合假设的事实。Rete 算法是目前效率最高的一个演绎法推理算法，许多 Java 规则引擎都是基于 Rete 算法来进行推理计算的。

推理引擎的推理步骤如下：

1）将初始事实数据载入工作内存；

2）使用模式匹配器比较规则库中的规则和工作内存中的事实数据；

3）如果执行规则时存在冲突(Conflict)，即同时激活了多个规则，则将冲突的规则放入冲突集合；

4）解决冲突，将激活的规则按顺序放入议程安排器；

5）使用执行引擎执行议程安排器中的规则；

6）重复步骤 2)至 5)，直到议程安排器中的所有规则被执行完毕。

Java 规则引擎就是从这一规则引擎的原始架构演变而来[14]。

2.1.2 Java 规则引擎的工作机制

Java 规则引擎检索提交到引擎的数据对象，根据这些对象的当前属性值和它们之间的关系，从加载到引擎的规则库中发现符合条件的规则，创建这些规则的执行实例[15]。这些实例将在引擎接到执行指令时依照某种优先顺序依次执行。一般来讲，Java 规则引擎内部由如下几部分构成：工作内存(Working Memory)或工作区，用于存放被引擎引用的数据对象集合；规则执行队列，用于存放被激活的规则执行实例；静态规则区，用于存放所有被加载的业务规则，这些规则将按照某种数据结构组织。当工作区中的数据发生改变后，引擎需要迅速根

据工作区中的对象状态，调整规则执行队列中的规则执行实例。Java 规则引擎工作机制的示意图如图 2-2 所示。

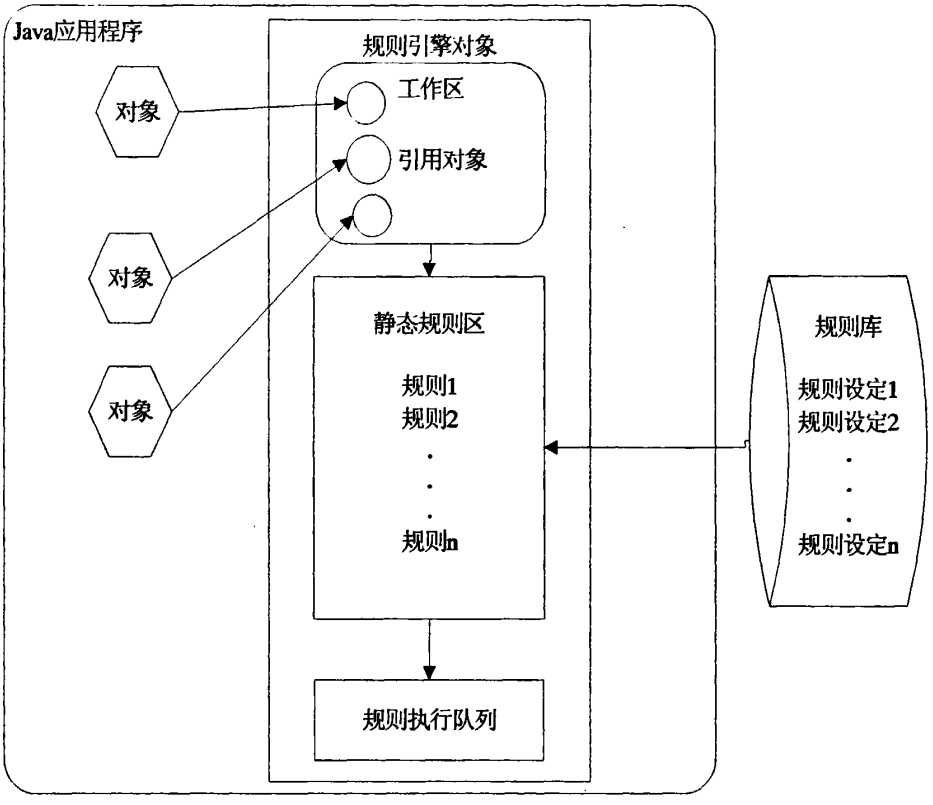

图 2-2 Java 规则引擎工作机制 [14]

当引擎执行时，会根据规则执行队列中的优先级顺序逐条执行规则执行实例，由于规则的执行部分可能会改变工作区的数据对象，从而会使队列中的某些规则执行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，生成新的规则执行实例进入队列。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”反应完全是由工作区中的数据驱动的。

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。规则条件匹配的效率决定了引擎的性能，引擎需要迅速测试工作区中的数据对象，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，最

后输出匹配结果[16]。1979 年，美国卡耐基·梅隆大学的 Charles L. Forgy 博士在其博士论文[17]中首次提出了一种叫做 Rete 的算法，很好地解决了这方面的问题。目前世界顶尖的商用规则引擎产品基本上都使用 Rete 算法。

2.2 Rete 算法研究

"Rete"是拉丁语，相当于英语中的"Net"，是网络的意思。Rete 算法可以分为两部分：规则编译(Rule Compilation)和运行时执行(Runtime Execution)。

编译算法描述了产生式内存区或静态规则区(Production Memory)中的规则如何产生一个有效的辨别网络(Discrimination Network)。辨别网络通过数据在网络的传播过程中来过滤数据。在辨别网络的顶端将会有很多匹配的数据。当顺着网络向下走，匹配的数据将会越来越少。在网络的最底部是终端节点(Terminal Node)。在 Dr.Forgy 的论文[17]中，描述了 4 种基本节点：根节点(Root Node)，单输入结点(1-Input Node)，双输入结点(2-Input Node)和终端结点(Terminal Node)。如图 2-3 所示，在 Drools 规则引擎的辨别网络中定义了 8 种节点类型：对象类型结点(ObjectType Node)，阿尔法结点(Alpha Node)，左输入适配器结点(LeftInputAdapter Node)，求值结点(Eval Node)，根节点(Rete Node)，联接结点(Join Node)，存在结点(Not Node)和终端结点(Terminal Node)。下面的描述均使用图 2-3 中的图例来表示各种结点。

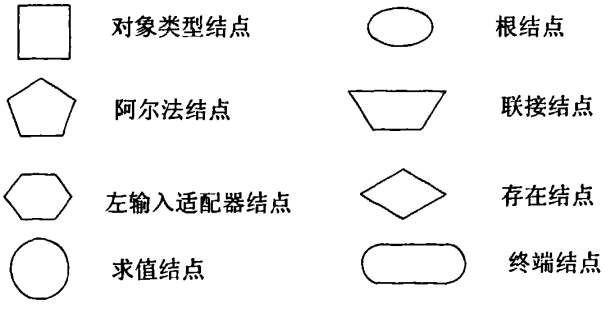

图 2-3 Drools 引擎中定义的 8 种节点类型

根节点是所有的对象进入网络的入口。然后，从根节点立即进入到对象类型结点，其作用是使引擎只做它需要做的事情。例如，有两个对象集：帐户类

(Account)和订单类(Order)。如果规则引擎需要对每个对象都进行一个周期的评估，那会浪费很多的时间。为了提高效率，引擎将只让匹配对象类型的对象到达节点。通过这种方法，如果某个应用声明一个新的帐户类对象，它不会被传递到订单类对象类型节点中。很多现代 Rete 算法实现都有专门的对象类型结点。其中一些对象类型结点被用散列法进行了优化。图 2-4 说明了这一过程。

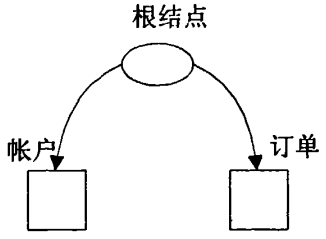

图 2-4 根结点向对象类型结点的传递

对象类型结点能够传播到阿尔法结点（对应单输入结点），左输入适配器结点和双输入结点（对应联接结点和存在结点）。阿尔法结点被用来评估字面条件(Literal Conditions)。尽管在 Dr. Forgy 的论文里只提到了相等条件（字面相等），很多 Rete 实现还支持其他的操作。例如， Account.name =="Zhang Shan" 是一个字面条件。当一条规则对于一种对象类型结点有多条字面条件时，这些字面条件将被链接在一起。就是说，如果一个应用声明一个 Account 类对象，在它能到达下一个阿尔法结点之前，它必须先满足上一个字面条件。图 2-5 说明了 Account 类的阿尔法结点组合：name = ="Zhang Shan",Id = ="123456789"。

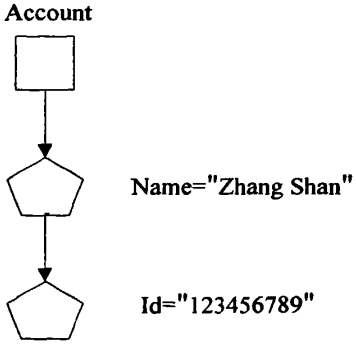

图 2-5 阿尔法结点的组合使用

Drools 通过散列法优化了从对象类型结点到阿尔法结点的传播。当一个阿尔法结点被加到一个对象类型结点的时候，就以字面值(Literal Value)作为键，以阿尔法结点作为值加入哈希表(HashMap)。当一个新的类实例进入对象类型结点的时候，不用传递到每一个阿尔法结点，可以直接从哈希表中获得正确的阿尔法结点，避免了不必要的字面检查。

双输入节点对应于 Drools 中的两种结点：联接结点和存在结点（共同被称为贝塔结点——BetaNodes)。双输入节点被用来比较两个对象及其属性。这两个对象可以是同种类型，也可以是不同类型。双输入节点的两个输入称为左边输入(Left)和右边输入(Right)。左边输入通常是一列对象(A list of objects)。在 Drools 中，这是一个数组。右边输入是单一对象。两个存在结点可以完成存在性检查。Drools 将索引应用在双输入节点上，从而扩展了 Rete 算法。图 2-6 说明了一个联接结点的使用。

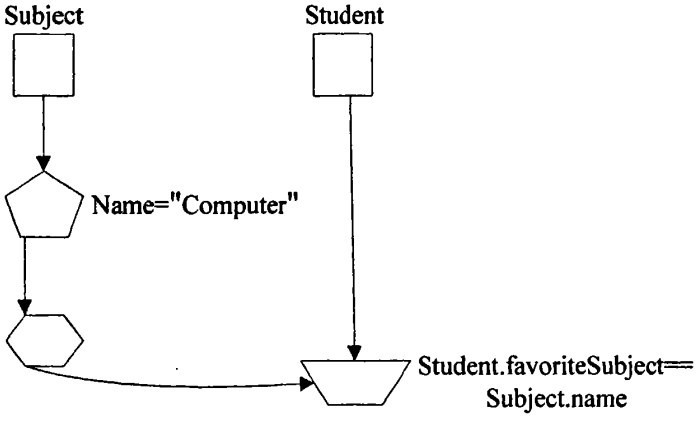

图 2-6 联接结点的使用

图中的左边输入用到了一个左输入适配器结点，它的作用是将一个单一对象转化为一个单对象数组(Single Object Array)，再传播到联接节点。因为上面提到过左边输入通常是一列对象(A list of objects)。

终端结点被用来表明一条规则已经匹配了它的所有条件(Conditions)。在这点，这条规则有了一个完全匹配(Full Match)。在一些情况下，一条带有“或”条件的规则可以有超过一个的终端结点。

Drools 引擎通过节点的共享来提高规则引擎的性能。因为很多的规则可能

存在部分相同的模式，节点的共享可以对内存中的节点数量进行压缩，以提供遍历节点的过程。下面的两个规则就共享了部分节点。

```
Rule1:
    when
        Subject($subject: name == "Computer")
        $student:Student(favouriteSubject == $subject)
    then
        System.out.println($person.getName()+ " likes "+$subject.getName() );
    end

Rule2:
    when
        Subject($subject:name ==   "Computer" )
        $student:Student(favouriteSubject != $subject)
    then
      System.out.println($student.getName()+"doesnot like "+$subject.getName() );
    end
```

这两条规则的中的左边(LHS, Left-Head-Side)基本是一样的，只是最后的favouriteSubject ，一条规则是等于$subject ，而另一条规则是不等于$subject 。图 2-7 是这两条规则的节点共享示意图。

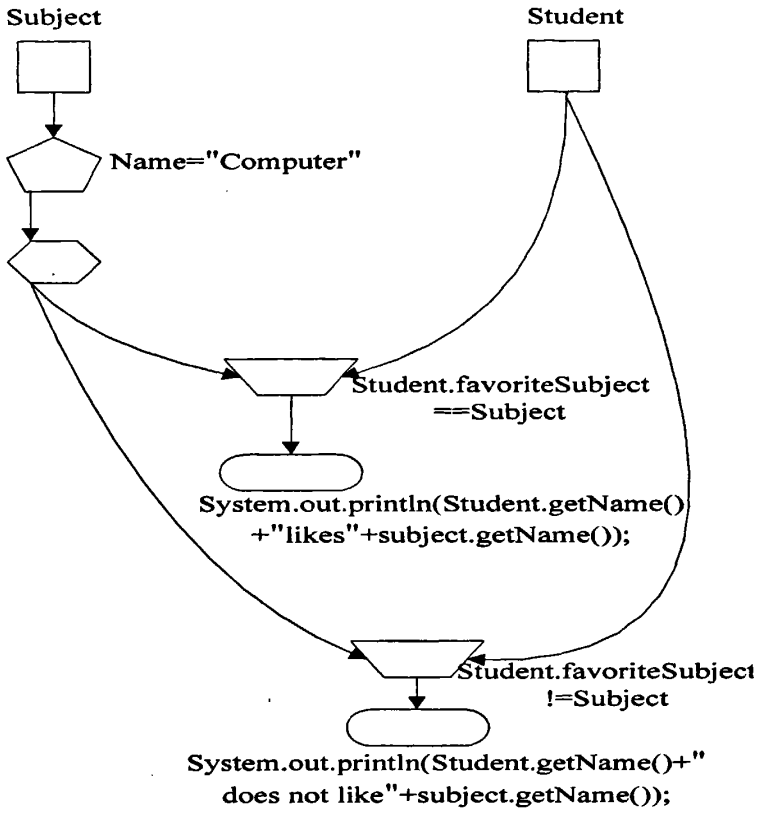

图 2-7 规则节点的共享

从图 2-7 可以看到，编译后的 Rete 网络中，阿尔法结点是共享的，而双输入结点不是共享的。这两条规则都有各自的终端结点。

Rete 算法的第二个部分是运行时执行(Runtime Execution)。当某个应用声明一个对象后，Drools 引擎将数据传递到根结点。然后它进入对象类型结点并沿着网络向下传播。当数据匹配一个节点的条件，节点就将它记录到相应的内存中。这样做的主要原因是可以带来更快的性能。虽然记住完全或部分匹配的对象需要大量的内存，但这使它具有速度快和可伸缩的优点。当一条规则的所有条件都满足，这就是完全匹配。而只有部分条件满足，就是部分匹配。

2.3 Drools 规则引擎研究

Drools 是一款成功的开源 Java 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，采用的语言是当前流行的面向对象语言——Java。

2.3.1 Drools 规则引擎 API 分析

Drools 分为两个主要部分：构建(Authoring)时组件和运行时(Runtime)组件。

构建的过程涉及到 .drl 或 .xml 规则文件的创建，它们被读入一个解析器，使用 ANTLR3(即语言识别的另一个工具，ANother Tool for Language Recognition)语法进行解析。解析器对语法进行正确性的检查，然后产生一种中间结构——"descr"，"descr"用 AST 来描述规则。AST 然后被传到 PackageBuilder 类，由 PackagBuilder 类来产生 Package 对象。PackageBuilder 类还承担着一些代码产生和编译的工作，这对于产生 Package 对象都是必需的。Package 对象是一个可以配置的，可序列化的，由一个或多个规则组成的对象。图 2-8 说明了它们之间的相互关系。

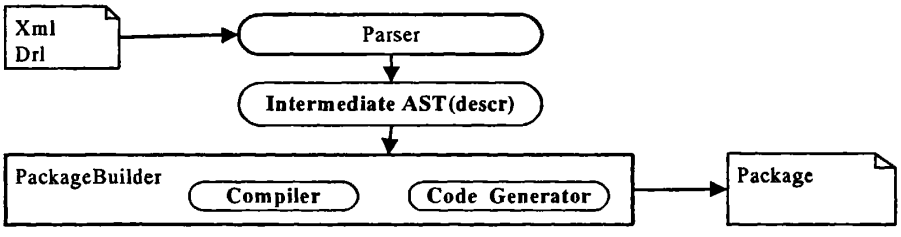

图 2-8 Drools 规则引擎的构建时组件

RuleBase 是一个运行时组件，它包含了一个或多个 Package 对象。可以在任何时刻将一个 Package 对象加入或移出 RuleBase 对象。一个 RuleBase 对象可以在任意时刻实例化一个或多个 WorkingMemory 对象，在它的内部保持对这些 WorkingMemory 的弱引用。WorkingMemory 由一系列子组件组成。当应用程序中的对象被声明(assert)进 WorkingMemory，可能会导致一个或多个激活(Activation)的产生，然后由 Agenda 负责安排这些激活的执行。图 2-9 说明了它

们的相互关系。

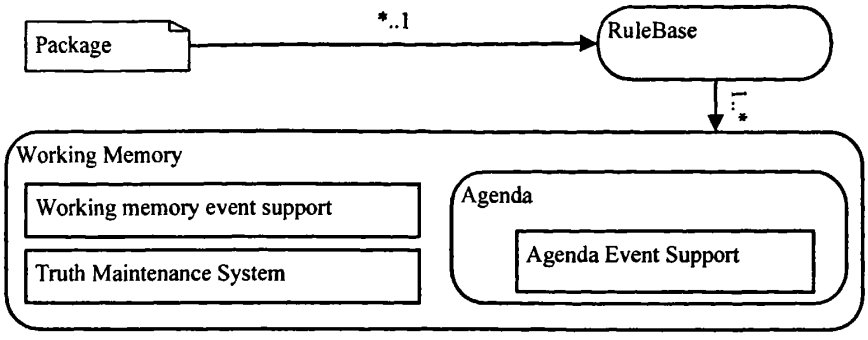

图 2-9 Drools 规则引擎的运行时组件

主要有三个类用来完成规则的构建过程：DrlParser， XmlParser 和 PackageBuilder。前两个解析器类 (DrlParser 类和 XmlParser 类) 从传入的 Reader 类实例产生 descr AST 模型。

下面将详细分析 Drools 规则引擎主要类（接口）的使用。

1. PackageBuilder 类提供了简便的 API，使得可以忽略 DrlParser 和 XmlParser 这两个解析器类的存在。这两个简单的方法是："addPackageFromDrl()" 和"addPackageFromXml()"，两个方法都只要传入一个 Reader 类的实例作为参数。下面的例子说明了如何从类路径(ClassPath)下的 xml 或 drl 文件创建一个 Package 对象。所有传入同一个 PackageBuilder 实例的规则源，都必须是在相同的包(package)命名空间中。下面的代码段演示了如何从指定的类路径下读取规则配置文件，并生成一个 Package 对象：

```
PackageBuilder builder = new PackageBuilder();
builder.addPackageFromDrl(new InputStreamReader(getClass().
        getResourceAsStream("package1.drl")));
builder.addPackageFromXml(new InputStreamReader(getClass().
        getResourceAsStream("package2.drl")));
Package pkg = builder.getPackage();
```

PackageBuilder 类的使用如图 2-10 所示。

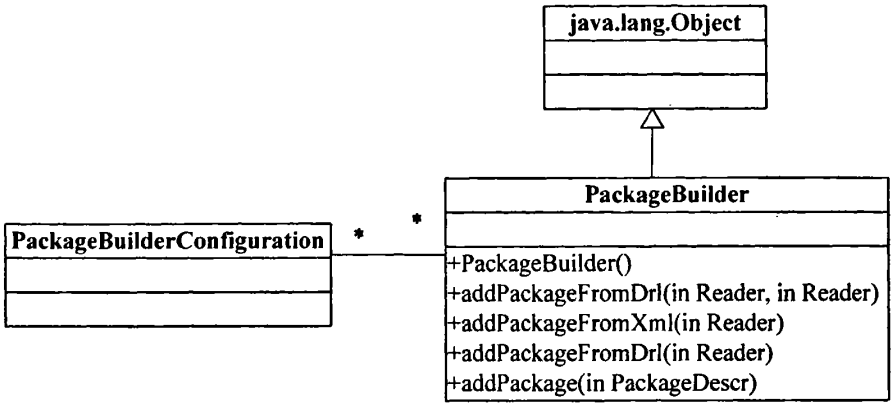

图 2-10 PackageBuilder 类的类图

PackageBuilder 可以通过 PackageBuilderConfiguration 类进行配置。通常，可以指定另一个父类加载器(parent ClassLoader)和编译器类型（默认是 Eclipse JDT)。

PackageBuilderConfiguration 类的使用如图 2-11 所示。

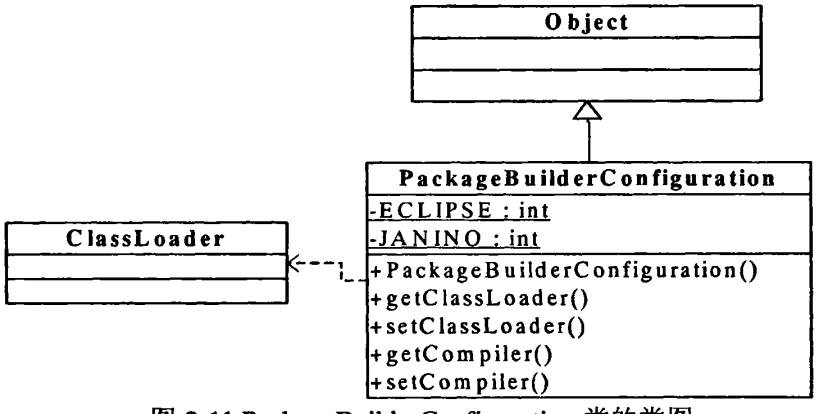

图 2-11 PackageBuilderConfiguration 类的类图

下面的代码段演示了如何指定 JANINO 编译器：

```
PackageBuilderConfiguration conf = new PackageBuilderConfiguration();
conf.setCompiler( PackageBuilderConfiguration.JANINO );
PackageBuilder builder = new PackageBuilder( conf );
```

下面的代码段演示了如何设置 JDK 的版本为 1.5：

```
PackageBuilderConfiguration conf = new PackageBuilderConfiguration();
conf.setJavaLanguageLevel( "1.5" );
PackageBuilder builder = new PackageBuilder( conf );
```

2. 一个 RuleBase 类包含了多个将被使用的规则包(Packages of Rules)的引用。RuleBase 类是可序列化的，所以它可以被配置到 JNDI(Java Naming and Directory Interface)或其他类似的服务中。通常，第一次使用时，一个 RuleBase 类被创建并缓存。RuleBase 类由工厂类 RuleBaseFactory 来实例化，默认返回一个 RETEOO 型的 RuleBase。可以传入参数来指定返回的 RuleBase 类的类型（包括 RETEOO 或 LEAPS 两种）。然后，用 addPackage 方法加入 Package 实例。可以加入有相同命名空间(NameSpace)的多个 Package 实例。

RuleBase 类的使用如图 2-12 所示。

RuleBase
-LEAPS : int -RETEOO : int
+addPackage(in Package) +getPackages() +getWorkingMemories() +newWorkingMemory() +newWorkingMemory(in boolean) +removeRule(in String) +removePackage(in String, in String)

图 2-12 RuleBase 类的类图

下面的代码段演示了如何创建一个 RuleBase 类的对象：

```
RuleBase ruleBase = RuleBaseFactory.newRuleBase();//默认返回一个 RETEOO 型
RuleBase ruleBase = RuleBaseFactory.newRuleBase(RuleBase.LEAPS);//LEAPS 型
```

RuleBase 实例是线程安全的，所以可以让一个 RuleBase 实例在多个线程中共享。对于一个 RuleBase 的最常用的操作是产生一个新的 WorkingMemory 类。

一个 RuleBase 实例保持着到它所产生的 WorkingMemory 类的弱引用，所以在长时间运行的 WorkingMemory 类中，如果其中的规则发生改变，这些 WorkingMemory 类可以及时地根据最新的规则进行自动更新，而不必重启

WorkingMemory 类。也可以指定 RuleBase 不必保持一个弱引用，但是要保证 RuleBase 类不能被更新。

下面的代码段演示了如何由 RuleBase 对象生成一个 WorkingMemory 实例：

```
ruleBase.newWorkingMemory();  //保持指向 WorkingMemory 的弱引用
ruleBase.newWorkingMemory(false);  //不保持指向 WorkingMemory 的弱引用
```

任何时候，Package 类实例可以被加入或移除；这些改变都会被反映到现存的 WorkingMemory 类中。下面的代码段演示了如何加入和移除一个包实例：

```
ruleBase.addPackage(pkg); //加入一个包实例
ruleBase.removePackage("com.dc.rules"); //移除同一包下的所有规则
ruleBase.removeRule("com.dc.rules ", "Hello_Rules"); //移除某一包下的一条规则
```

虽然有删除一个单独规则的方法，但是却没有加入一个单独规则的方法，要达到这个目的只有加入一个只含有一条规则的包。

3. RuleBaseConfiguration 类可以指定 RuleBase 类的附加行为。在加入 RuleBase 后，RuleBaseConfiguration 就变成不可变对象。下面的代码段演示了如何设置工作内存声明对象的模式为相等模式（见 WorkingMemory 类的分析）：

```
RuleBaseConfiguration conf = new RuleBaseConfiguration ();
conf.setProperty( RuleBaseConfiguration.PROPERTY_ASSERT_BEHAVIOR,
RuleBaseConfiguration.WM_BEHAVIOR_EQUALITY );
RuleBase ruleBase = new ReteooRuleBase (conf);
```

4. WorkingMemory 类是运行时规则引擎的核心类。它保持了所有被声明到 WorkingMemory 的数据的引用，直到撤消(Retracted)。WorkingMemory 是有状态对象。它们的生命周期可长可短。如果从一个短生命周期的角度来同一个引擎进行交互，意味着可以使用 RuleBase 对象来为每个会话产生一个新的 WorkingMemory ，然后在结束会话后释放这个 WorkingMemory（产生一个 WorkingMemory 是一个廉价的操作）。另一种形式，就是在一个相当长的时间中，保持一个 WorkingMemory ，并且对于新的 Facts 保持持续的更新。当希望释放一个 WorkingMemory 的时候，最好的实践是调用它的 dispose() 方法，此时 RuleBase 中对它的引用将会被移除（尽管这是一个弱引用）。不管怎样最后它将会被当成垃圾收集掉。

图 2-13 给出了 WorkingMemory 类的主要方法的使用。

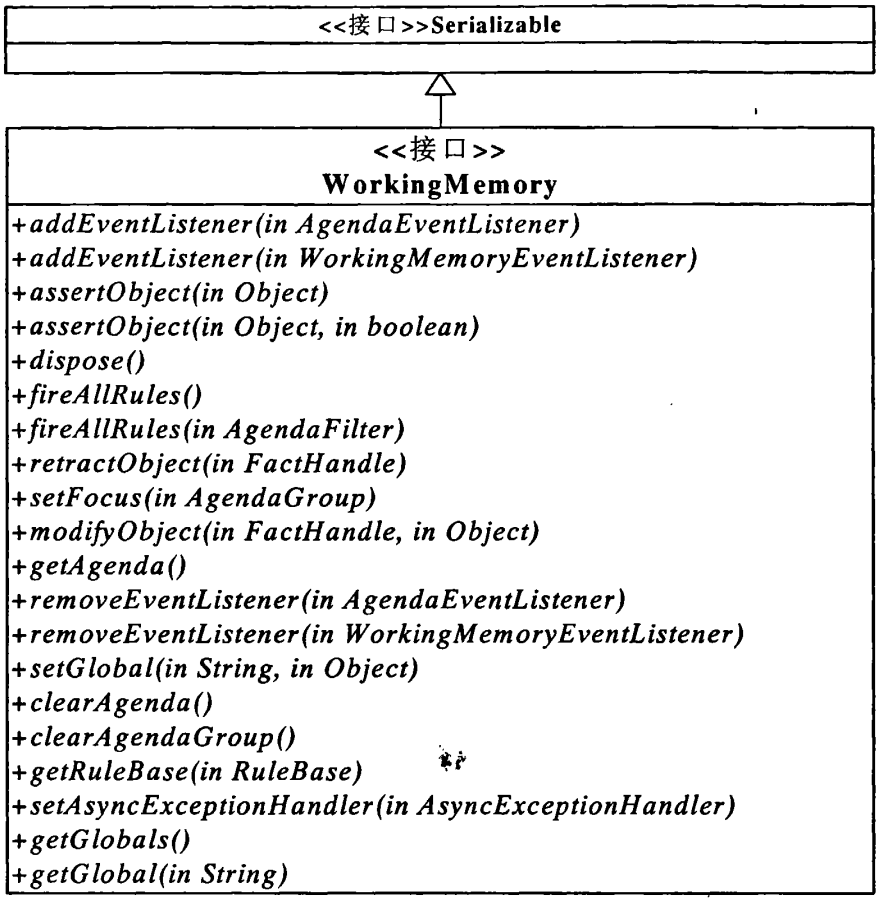

图 2-13 WorkingMemory 类的类图

WorkingMemory 类的使用分为如下几方面：

1) Facts

Facts 是在某一应用中，被声明到 WorkingMemory 中的标准 JavaBean 对象。Facts 是规则可以访问的任意的 Java 对象。规则引擎中的 Facts 并不是拷贝应用中的数据，它只是持有应用中数据的引用。String 类和其他没有 getter 和 setter 方法的类不是有效的 Fact 。这样的类不能使用域约束(Field Constraints)，因为使用域约束要依靠 JavaBean 标准的 getter 和 setter 方法来同对象交互。

2) Assertion

Assertion 是将 Facts 告诉给 WorkingMemory 类的动作，即 WorkingMemory.assertObject (yourObject)。当应用中声明了一个 Fact，它将被检

查是否匹配规则。这意味着所有的匹配工作将会在声明的过程中完成。尽管如此，当声明完 Fact 之后，还要调用 fireAllRules()方法来执行规则。

当一个对象被声明后，会返回一个 FactHandle 类。这个 FactHandle 类是一个代表在 WorkingMemory 中的已声明对象的令牌(Token)。当希望收回(Retract)或者修改(Modify)一个对象时，这个令牌用来同 WorkingMemory 进行交互。

下面的代码片段演示了这一过程：

```
Student stu=new Student ("jason");
FactHandle jasonHandle=workingMemory.assertObject (stu);
```

WorkingMemory 有两种声明对象的模式：同一模式(Identity)和相等模式(Equality)，默认是同一模式。

在同一模式下， WorkingMemory 使用一个 IdentityHashMap 类来存储所有已声明的对象(Object)。这个模式下，当被声明的对象(Object)是同一个实例时（即有相等的 hashCode()方法返回值），它返回同一个 FactHandle。

在相等模式下，WorkingMemory 使用一个 HashMap 来存储所有已声明对象。这个模式下，当被声明的对象相等(即有相同的 equals()方法返回值)时，它也返回同一个 FactHandle。

3) Retraction

基本上就是声明(assert)的逆操作。当撤消(Retract)一个 Fact 时，WorkingMemory 将不再跟踪那个 Fact。任何被激活(Activated)并依赖那个 Fact 的规则将被取消。完全有可能存在某条规则是依赖于一个不存在的 Fact。在这种情况下，撤消(Retract)一个 Fact 可能导致一条规则被激活。要撤消一个已声明的对象，必须用声明该对象时返回的那个 FactHandle 类实例做为参数。

下面的代码段演示了如何撤消一个已声明的对象：

```
yourObject  obj = new  yourObject( "obj_name" );
FactHandle  myHandle= workingMemory.assertObject (obj);
.....//某些操作
workingMemory.retractObject (myHandle);
```

4) Modification

当一个 Fact 被修改了，必须通知规则引擎进行重新处理。在规则引擎内部实际上是对旧的 Fact 进行撤消操作 ，然后对新的对象再进行声明操作。要使用 modifyObject() 方法来通知 WorkingMemory，被改变的 Object 并不会自己通知

规则引擎。modifyObject()方法总是要把被修改的 Object 做为第二参数，这就可以把一个不可变对象替换为另一个新对象。下面的代码段演示了如何修改一个已声明对象：

```
yourObject   obj= new yourObject( " obj_name " );
FactHandle   myHandle= workingMemory.assertObject (obj);
……..//某些操作
obj.setName ("another_name");
workingMemory.modifyObject (myHandle, obj);
```

5) Globals

Global 是一个能够被传进 WorkingMemory 但不需要声明的命名对象。大多数这样的对象被用来作为静态信息或服务。这些服务被用在一条规则的后件(RHS, Right-Hand-Side)，或者可能是从规则引擎返回对象的一种方法。下面的代码段声明的一个名为"list"的元素为 String 型顺序表对象：

```
List<String> list = new ArrayList<String> ();
workingMemory.setGlobal ("list", list);
```

setGlobal()方法传进去的命名对象必须同 RuleBase 中的定义具有相同的类型（即同规则文件中用 Global 关键字所定义的类型相同），否则会抛出一个运行时异常(RuntimeException)。如果在 setGlobal 方法调用之前就引用了定义的 Global 对象，则会抛出一个空指针异常(NullPointerException)。

6) 属性改变监听器(Property Change Listener)

如果 Fact 对象是一个 JavaBean（具有 getter 和 setter 方法），可以为它们实现一个属性改变监听器，然后把它告诉给规则引擎。这意味着，当一个 Fact 改变时，规则引擎将会自动知道，并进行相应的动作（这时就不需要调用 modifyObject()方法来通知 WorkingMemory）。要让属性改变监听器生效，首先要将 Fact 设置为动态(Dynamic)模式，这可以通过将布尔值 true 做为 assertObject() 方法的第二个参数来实现。下面的代码段演示了这一过程：

```
Student jason = new Student ("Jason");
FactHandle jasonHandle =
workingMemory.assertObject( jason, true);   //指定以动态模式加载一个对象
```

然后要在 Student 类中加入一个 PropertyChangeSupport 类的实例变量和两个方法：addPropertyChangeListener()和 removePropertyChangeListener()。最后要

在该对象(Student)的 setter 方法中通知 PropertyChangeSupport 所发生的变化。示例代码如下：

```
private final PropertyChangeSupport changes=new  PropertyChangeSupport(this);
  public  void  addPropertyChangeListener( final  PropertyChangeListener l) {
  this .changes.addPropertyChangeListener(l);
   }
  public  void  removePropertyChangeListener(final  PropertyChangeListener l)
   {
    this .changes.removePropertyChangeListener(l);
   }
  public  void  setName( final  String newName)
   {
      String oldName = this.getName();
      this.name= newName;
      this.changes.firePropertyChange("name", oldName, newName);
   }
```

5. Agenda 是 Rete 算法的一个特点。在一个工作内存活动发生时，可能会有多条规则发生完全匹配。当一条规则完全匹配的时候，一个激活(Activation)类就被创建(它引用了这条规则和与其匹配的 Facts)，然后被放进 Agenda 中。Agenda 通过使用冲突解决策略(Conflict Resolution Strategy)来安排这些激活的执行。

Drools 规则引擎工作在一个“两阶段”(Two-Phase)模式下：

1）工作内存活动阶段：声明进新的 Facts，修改现有的 Facts 和收回现有的 Facts 都是工作内存活动。通过在应用程序中调用 fireAllRules()方法，会使引擎转换到议程器评估阶段。

2）议程安排器评估阶段：尝试选择一条规则进行触发(Fire)。如果规则没有找到就退出，否则它就尝试触发这条规则，然后再次转换到工作内存活动阶段。

这个过程一直重复，直到议程安排器为空，此时控制权就回到应用程序中。当工作内存活动发生时，没有规则正在被触发。

图 2-14 说明了这个两阶段工作循环的过程。

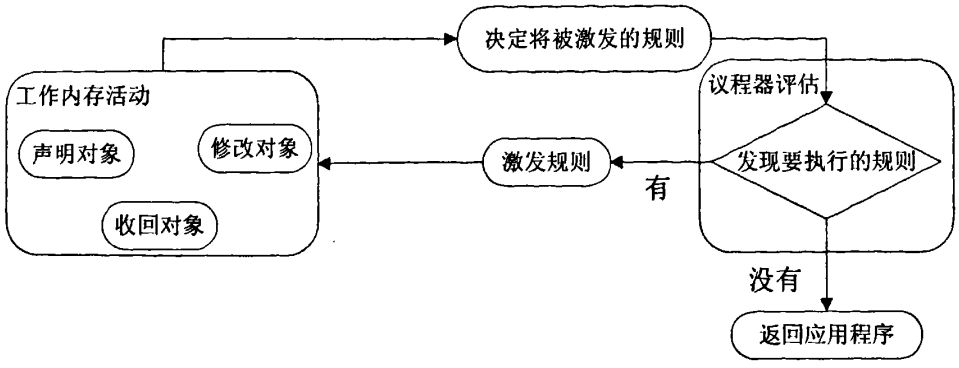

图 2-14 Drools 规则引擎的两阶段工作过程

议程安排器(Agenda)的工作任务表现在以下两个方面：

1）解决冲突(Conflict Resolution)

当有多条规则在议程安排器中，就需要解决冲突。当触发一条规则时，会对工作内存产生副作用。规则引擎需要知道规则以什么顺序被触发（例如，触发规则 A 可能会引起规则 B 被从议程安排器中移除。）

Drools 采取的冲突解决策略有两种，按照使用频度分为： Salience，LIFO(Last-In-First-Out)。最常用的策略是 Salience，即优先级策略，用户可以为某个规则指定一个高一点的优先级（通过赋给它一个比较大的数字）。具有高 Salience 值的规则将被优先激发。

2) 议程安排器组(Agenda Groups)

议程安排器组是划分议程安排器中规则（即处于激活状态的规则）的一种方法。在任意一个时刻，只有一个议程安排器组拥有焦点 (Focus)，这意味着只有在那个组中的激活才是有效的。

议程安排器组是在规则组之间创建一个流(Flow)的简便的方法。可以在规则引擎中，或是用 API 来切换具有焦点的组。如果规则有很明确的多阶段(Phases)或多序列(Sequences)的处理，可以考虑用议程安排器组来达到这个目的。

每次调用 setFocus()方法的时候，那个议程安排器组就会被压入一个堆栈，当这个有焦点的组为空时，它就会被弹出，然后下一个组就会被执行。一个议程安排器组可以出现在堆栈的多个位置。默认的议程安排器组是"MAIN"，所有没有被指定议程安排器组的激活态规则都被放到那个组中，这个组总是被放在堆栈的第一个组，并默认给予焦点。

6. AgendaFilter 接口，用来允许或禁止一个激活的规则能够被触发(Fire)。Drools 提供了下面几种方便的默认实现类：RuleNameEndWithAgendaFilter、RuleNameEqualsAgendaFilter 和 RuleNameStartsWithAgendaFilter。

要使用一个过滤器就要在调用 fireAllRules()方法的时候指定它。下面的代码段将对所有名字以"Pass"结尾的规则进行过滤，即禁止其被触发：

workingMemory.fireAllRules (new RuleNameEndsWithAgendaFilter ("Pass"));

7. Event 包里提供了规则引擎的事件机制，包括规则被触发，对象已经被声明等事件。可以使用事件机制来进行面向切面的编程(AOP,Aspect-Oriented Programming) 编程。

有两种类型的事件监听器：WorkingMemoryEventListener 和 AgendaEventListener。

图 2-15 和图 2-16 给出了它们的类图。

图 2-15 WorkingMemoryEventListener 类的类图

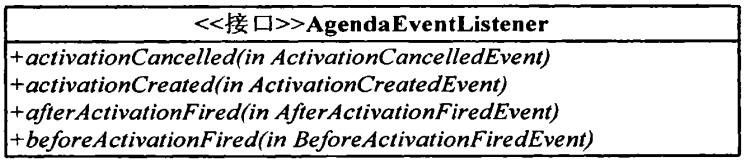

图 2-16 AgendaEventListener 类的类图

对两个 EventListener 接口都提供了默认实现，但在方法中并没有做任何事情。可以继承 DefaultAgendaEventListener 和 DefaultWorkingMemoryEventListener 这两个默认实现类来完成自己的实现。

下面的代码段演示了如何扩展一个 DefaultAgendaEventListener 并把它加到 WorkingMemory 类中，例子中只实现了 afterActivationFired()方法。

```
workingMemory.addEventListener (new DefaultAgendaEventListener () {
    public  void  afterActivationFired(AfterActivationFiredEvent event) {
```

```
        super.afterActivationFired( event );
        System.out.println(event);
    }
});
```

Drools 规则引擎也提供了 DebugWorkingMemoryEventListener 和 DebugAgendaEventListener 两个事件监听实现类，在其方法中实现了调试(Debug)信息的输出。下面的代码将 DebugWorkingMemoryEventListener 这个事件监听类加到工作内存中。

```
workingMemory.addEventListener (new DebugWorkingMemoryEventListener ());
```

2.3.2 Drools 规则配置文件(.drl)的结构和含义

因为本文在第四章将要介绍基于 Drools 规则引擎的动态数据清洗，并采用 XML 文件做为其规则配置文件的格式，在此先结合一个完整的配置文件的实例来分析 Drools 规则配置文件的结构及含义。

在写此文时，Drools 已经被合并到 JBoss 组织下，改名为 JBossRules。为了实现规则的动态、实时、自动地配置，本文以 Drools 2.1 为例，因为其规则配置文件是 XML 格式的，便于使用 JDOM，DOM4J 等工具对其进行读写和更新。

Drools 2.1 的规则配置文件的结点结构由 XSD(XML Schema Definition)格式文档所定义，详见附录一。

一个完整的 Drools 2.1 规则配置文件（.drl 文件）的实例如下：

```
<?xml version="1.0"?>
<rule-set name="BusinessRulesSample" xmlns="http://drools.org/rules"
    xmlns:java="http://drools.org/semantics/java"
    xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
    xs:schemaLocation="http://drools.org/rules rules.xsd
http://drools.org/semantics/java java.xsd">
    <java:import>java.lang.Object</java:import>
    <java:import>java.lang.String</java:import>
    <java:import>drools.Student</java:import>
     <java:import>drools.Recommendation</java:import>
```

```
        <java:functions>
    public static void printStudent(Student student){
    System.out.println("\nStudent Name:"+student.getStudentName()
    +"\n Sudent Age: "+student.getStudentAge()
    +"\n Student Sex:"+student.getStudentSex()
    +"\n Recommend "+student.getStudentName()
    +"go to schoole "
    +student.getSchoole()
    +": "
    +student.getRecommend());
    }
    </java:functions>
        <rule name="USSTSchoole" salience="-1">
            <parameter identifier="student">
                <class>drools.Student</class>
            </parameter>
            <java:condition>
                student.getSchoole().equals("USST")
            </java:condition>
            <java:condition>student.getStudentAge() > 15 &amp;amp;&amp;amp;
student.getGender=='F'</java:condition>
            <java:consequence>
                student.setRecommend(Recommendation.YES);
printStudent(student);
            </java:consequence>
        </rule>
        <rule name="FUDANSchoole" salience="-1">
            <parameter identifier="student">
                <class>drools.Student</class>
            </parameter>
            <java:condition>
```

```
            student.getSchoole().equals("FUDAN")
        </java:condition>
        <java:condition>student.getStudentAge() > 18</java:condition>
        <java:consequence>
        student.setRecommend(Recommendation.YES); printStudent(student);
        </java:consequence>
    </rule>
</rule-set>
```

从以上的规则配置文件例子可以分析得出，Drl 文件的结构如图 2-17 所示。

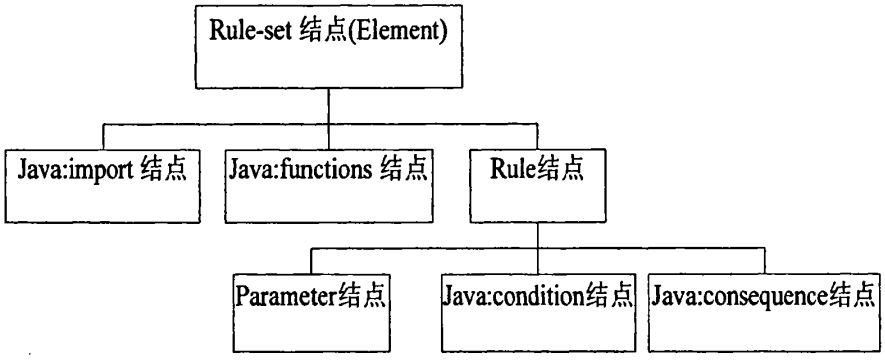

图 2-17 Drools 规则配置文件的结构

如图 2-17 所示，每个配置文件有且仅有一个 rule-set 结点；每个 rule-set 结点一般有三部分组成：java:import 结点（可有多个），java:functions 结点，rule 结点（可有多个，以 name 属性区分）；每个 rule 结点有三部分组成：parameter 结点，java:condition 结点，java:consequence 结点。

配置文件必须满足以下要求：必须有一个 rule-set 结点；每个 rule-set 结点至少有一个 rule 结点；每个 rule 结点必须在 rule-set 结点中有唯一的名字(name)；每个 rule 结点至少有一个 parameter 结点，且在 rule 内必须有唯一的标识符(identifer)；java:condition 结点中不能有分号；java:consequence 结点中必须有分号。

需要注意的是，在 Drools 规则配置文件中，一些在 Java 中的运算符号不能

直接写，比如“>”，“<”，“&&”，例如上面的“&&”就写成“&&”，如果是“<”，就应该写成“<”，以此类推。

Drl 配置文件中各个结点的含义如下：

每个 java:import 结点用来导入将要在 java:functions 结点，java:condition 结点和 java:consequence 结点中用到的 Java 类；在 java:functions 结点中定义了将要在 java:condition 结点和 java:consequence 结点中用到的函数；java:condition 结点中定义了规则被触发的条件，多个 java:condition 结点间是逻辑与的关系；parameter 结点声明了将要在 java:condition 结点和 java:consequence 结点中用到的变量名；java:consequence 结点定义了规则被触发时将要执行的动作；每个 java:condition 结点中的内容是一个逻辑表达式；每个 java:consequence 结点中的内容是一条或多条可被 JVM(Java Virtual Machine)执行的 Java 代码段。

2.4 本章小结

本章首先分析了基于规则的专家系统的结构原理和 Java 规则引擎的工作机制，然后分析了规则引擎的核心——Rete 算法的原理，最后对一种开源 Java 规则引擎——Drools 的 API 的使用进行了系统地分析和介绍，并结合实例分析了其规则配置文件的结构及含义。这些，都为后面详细阐述基于 Drools 规则引擎的动态数据清洗系统提供了理论基础。

第 3 章 数据清洗的概念

3.1 数据质量

3.1.1 脏数据的概念

“脏数据”是指数据不在给定的范围内或对于实际业务毫无意义，或是数据格式非法，以及错误操作发生等。主要表现为：数据格式错误，数据不一致，数据重复、错误，业务逻辑的不合理，违反业务规则等。例如，未经验证的身份证号码、日期字段等，年龄超过取值范围，滥用缩写词、数据输入错误、重复记录、丢失值、拼写变化、不同的计量单位和过时的编码等等。当今用户所面对的是一个多厂商异种数据库、异种操作系统和异种网络的环境，异种数据库间互联成为人们越来越迫切的需求。对于实际运行的系统来说，有许多因素都可能引起数据库系统之间的差异性，因此在异构数据集成中存在大量 “脏数据”[18]。

3.1.2 数据质量的概念

数据质量问题并不仅仅是指数据错误。文献[19]以形式化的方法定义了数据的一致性(Consistency)、正确性(Correctness)、完整性(Completeness)和最小性(Minimality)，数据质量被定义为这 4 个指标在信息系统中得到满足的程度。文献[20]提出了数据工程中数据质量的需求分析和模型，认为存在很多候选的数据质量衡量指标，用户应根据应用的需求选择其中一部分。数据质量衡量指标分为两类：数据质量指示器和数据质量参数，前者是客观的信息，比如数据的收集时间，来源等，而后者是主观性的，比如数据来源的可信度(Credibility)、数据的及时性(Timeliness)等。文献[21]提出了一些数据质量的评估指标。在进行数据质量评估时，要根据具体的数据质量评估需求对数据质量评估指标进行相应的取舍。数据质量评估至少应该包含以下两方面的评估指标[22-23]。

1） 可信度。它的具体含义如下：

精确性：描述数据是否与其对应的客观实体的特征相一致。

完整性：描述数据是否存在缺失记录或缺失字段。

一致性：描述同一实体的同一属性的值在不同的系统是否一致。

有效性：描述数据是否满足用户定义的条件或在一定的域值范围内。

唯一性：描述数据中是否存在重复记录。

2） 可用性。它的具体含义如下：

时效性：描述数据是当前数据还是历史数据。

稳定性：描述数据是否是稳定的，是否在其有效期内。

3.1.3 数据质量问题的分类

根据处理的是单数据源还是多数据源以及问题出在模式层还是实例层，文献[7]将数据质量问题分为四类：单数据源模式层问题、单数据源实例层问题、多数据源模式层问题和多数据源实例层问题，如表 3-1 所示。

表 3-1 数据质量分类

数据源	单数据源		多数据源	
层次	模式层	实例层	模式层	实例层
产生原因	缺少完整性约束；差的模式设计	数据记录错误	异构的数据模式和模式设计	不一致的数据汇总
表现形式	违反实体完整性约束和参照完整性约束	拼写错误等；相似重复记录	命名冲突；模式冲突	冗余，矛盾及不一致的数据

无论是模式层问题还是实例层问题，都可以分成字段、记录、记录类型和数据源四种不同的问题范围，分别说明如下：

字段：错误仅局限于单个字段值中。

记录：错误表现在同一条记录中不同字段值之间出现的不一致。

记录类型：错误表现在同一个数据源中不同记录之间的不一致关系。

数据源：错误表现在数据源中的某些字段值和其它数据源中相关值的不一致关系。

1. 单数据源中的数据质量问题

表 3-2 和表 3-3 分别给出了单数据源中模式层和实例层中数据质量问题的实例。

表 3-2 单数据源中模式层的数据质量问题

范围	问题	脏数据	原因
字段	不合法值	Birthday='1981-13-76'	超出值域
记录	违反属性依赖	Age=65,birthday='2007-12-01'	age=sysdate-birthday
记录类型	违反唯一性	Porvider1:Name='A1',No='P001'; Porvider2:Name='A2',No='P001'	编号不唯一
数据源	违反参照完整性	Provider:Name='A1', CityNo=101	编号为 101 的城市不存在

表 3-3 单数据源中实例层的数据质量问题

范围	问题	脏数据	原因
字段	空值	Phone-No=(0000)000000	值为缺省值，可能数据未输入或已丢失
	拼写错误	City='汉钟'	一般为数据录入错误
	含义模糊的值	Position='DBPorg'	DBPorg 意义不清
	多值嵌入	Name='汉中钢铁集团 723000 汉中'	一个字段中输入多个字段的值
	字段值错位	City='汉台区'	某个字段的值输入到另一个字段中
记录	违反属性依赖	City='汉中',zip='710000'	城市和邮政编辑之间不一致
记录类型	重复记录	供应商 1:('汉中钢铁集团','汉中') 供应商 2:('汉台区钢铁集团'，'汉中')	出于输入错误，同一个供应商信息输入了两次
	冲突的值	供应商 1:('汉中汉航集团'，'1') 供应商 2:('汉中汉航集团'，'2')	一个供应商用不同的值表示
数据源	引用错误	供应商 :Name='汉中烟草集团'， CityNo='1'	供应商与城市编号不能一一对应

2. 多数据源数据质量问题

多数据源集成中的数据清洗问题是信息化建设将面临的一个重要问题。当多个数据源集成时，发生在单数据源中的这些问题会更加严重。由于每个数据源都是为了特定应用而单独开发、部署和维护的，这在很大程度上导致数据管理系统、数据模型、模式设计和实际数据的不同。

在模式级，出现的主要问题是命名冲突和结构冲突[24-25]。命名冲突主要表现为不同的对象可能使用同一个命名，而同一对象可能使用不同的命名；结构冲突存在很多种不同的情况，一般是指在不同数据源中同一对象有不同表示，如不同的组成结构、不同的数据类型、不同的完整性约束等。

除了模式级的冲突，很多冲突仅出现在实例级上，即数据冲突。由于不同数据源中数据的表示可能会不同，单数据源中的所有问题都可能会出现，比如重复的记录、冲突的记录等。此外，在整个数据源中，尽管有时不同的数据源中有相同的字段名和类型，仍可能存在不同的数值表示，如对性别的描述，数据源 A 中可能用"true/false"来描述，数据源 B 中可能会用"T/F"来描述，或者对一些事物的不同度量单位来表示，如数据源 A 中采用美元作为货币单位，而数据源 B 中采用人民币作为货币单位。

3.2 数据清洗

3.2.1 数据清洗的概念

数据清洗(Data Cleansing)，也被称为数据净化(Data Scrubbing) 和数据清理(Data Cleaning)，是为了改进数据质量而从原始数据中检测并消除错误和冲突的过程[26]。

包含数据清洗过程的主要领域有三个：数据仓库(DW, Data Warehouse)，数据库中的知识发现(KDD, Knowledge Discovery in Databases)和综合数据质量管理(TDQM, Total Data Quality Management)[19]。

在数据仓库环境下，数据清洗是抽取转换装载过程(ETL, Extraction-Transformation-Loading)的一个重要部分；在数据库知识发现中的数据清洗主要是提高数据的可利用性，如去除噪声、无关数据、空白数据域，考虑时间顺序和数据的变化等；全面数据质量管理中的数据清洗是减少错误和不一

致性、解决对象识别的过程[7]。

数据清洗按照实现方式与范围，可分为 4 种[7]：

1. 手工实现

这是数据清洗的最简单、最基本的方法，即将数据的值与其真实值相比较。例如，要查清客户数据是否正确，可以每年做一次客户调查，确认其正确的姓名，地址与工作单位等。当然，这样比较的成本最昂贵，并且比较与真实的差别对避免将来的错误没有任何帮助。

2. 应用程序实现

该方法是通过编写程序检测并改正错误，从而避免花时间与实际数据进行比较。这个方法可推广到多数据库的情形，比较一致的数据就认为是正确的，否则就是不正确的，需要进一步考查与更正。数据清洗是一个反复进行的过程，清理程序复杂、系统工作量大。

3. 解决特定应用域的问题

如根据概率统计学原理查找数值异常的记录，对姓名、地址、邮政编码等进行清洗。这种方法要利用专家知识和人工智能的有关知识。

4. 与应用领域无关的数据清洗

这一部分的研究主要集中在缺失值处理和重复值处理[18]。

本文主要研究在数据仓库环境下，数据抽取、转换和加载(ETL, Extraction-Transformation-Loading)过程中基于 Drools 规则引擎的动态数据清洗。

3.2.2 数据清洗的模型和流程

数据清洗的输入数据一般是 3.1.1 节所述的“脏数据”。数据清洗对输入的“脏数据”进行修正，消除其中的错误和冲突，输出满足要求的“干净”数据。图 3-1 描述了数据清洗的基本概念模型。

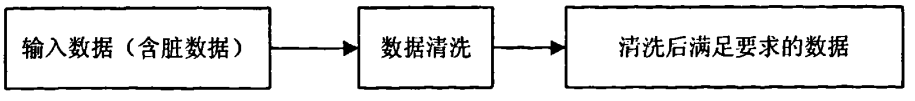

图 3-1 数据清洗概念模型[27]

一般的数据清洗的流程如图 3-2 所示。包括以下步骤：

1）数据分析：为了能自动检测错误和不一致，需要进行详细的数据分析。除了手工检查数据或数据采样，应尽可能采用程序方式自动获得关于数据质量的元数据；

2）定义清洗规则：用户根据数据质量元数据，定义相应的清洗规则；

3）执行清洗规则；

4）反复执行 2）和 3），直到将检测到的数据质量问题处理完毕；

5）反复执行 1）至 4），直到数据的质量满足要求为止。因为有的数据质量问题是在其它问题处理后才会出现的。

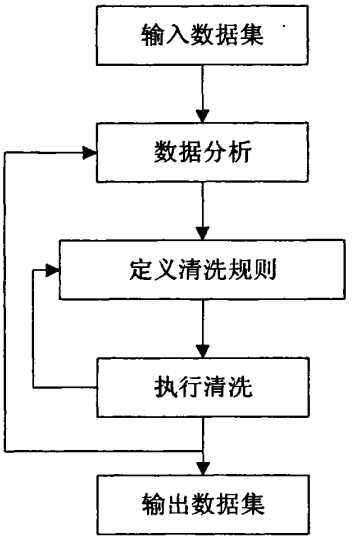

图 3-2 数据清洗过程[27]

3.3 本章小结

本章首先详细介绍了“脏数据”的概念及数据质量的概念和评估指标，接着结合实例给出了数据质量问题的详细分类：单数据源模式层问题、单数据源实例层问题、多数据源模式层问题和多数据源实例层问题，最后详细介绍了数据清洗的概念，模型及过程。这些，都为后面详细阐述基于 Drools 规则引擎的动态数据清洗系统提供了理论基础。

第 4 章 基于 Drools 规则引擎的动态数据清洗系统

4.1 动态数据清洗系统中使用的规则

本系统使用的规则分为两类：检测“脏数据”的逻辑规则(领域知识规则)和对“脏数据”采取的动作即修复或丢弃 (清洗动作规则)的逻辑。可以用这两类规则来描述数据清洗的整个过程。

4.2 规则引擎应用于动态数据清洗的优点

文献[28]没有给出基于规则的数据清洗的设计。但数据清洗规则最终表现为复杂的业务逻辑。如何描述、存储、更新和高效地执行这些逻辑是基于规则引擎的动态数据清洗技术的重点。

数据质量问题的定义不是一尘不变的，它不仅和具体的业务领域有关而且还会随着企业业务的变化而变化的。正因为如此，数据清洗规则也会相应地发生改变。这就要求基于规则引擎的动态数据清洗技术的实现必须能灵活地定义规则，灵活地更新规则，即实现规则的动态实时配置。

本文使用 Drools 规则引擎来解决上述问题，将硬编码的程序性判断转化为软编码的可动态配置的规则集，并且动态地生成规则库，从而能做到灵活地定义规则和动态调整规则。这种动态性主要体现在规则的持久化存储和 Drools 规则配置文件的动态更新。

4.3 DREBDDC 系统概述

基于 Drools 规则引擎的动态数据清洗系统(DREBDDC, Drools Rule Engine –Based Dynamic Data Cleansing)主要用于对“脏数据”的自动识别和对其进行的操作，包括以下功能模块：领域知识规则和清洗动作规则的输入及持久化模块、规则配置文件动态生成模块、表达式扫描转换模块、通用数据存取接口模块、规则引擎执行模块等。规则引擎执行模块采用开源的 Java 规则引擎——Drools

2.1。通用数据存取接口采用基于 JDBC 的数据库连接池技术进行数据加载和持久化。

由于在实际应用中，待清洗的多个数据源可能位于不同操作系统平台上的多种数据库管理系统中，本系统采用面向对象的规则引擎实现 Drools 和基于 JDBC 的数据库连接池访问接口，便于与不同平台、多种数据库产品的集成。

4.4 DREBDDC 系统的设计与实现

4.4.1 领域知识规则的巴科斯范式定义

要很好的完成数据清洗工作，一定要结合特定应用领域的知识。因此，需要将领域知识以规则的形式表示出来，在整个数据清洗过程中加以引用，并允许添加和修改。

领域知识规则以一组表的形式存储在规则数据库中。由多个规则组组成，每个规则组都具体反映某一个领域的知识。

结合 ETL 工具[27]中的巴科斯范式(BNF, Backus-Naur Form)语言，领域知识规则组的定义形式如下：

<Business Rule>::=

RNO<ID>;

RuleGroup (<Declaration of Rule Group>);

CondList (<Declaration of Condition>);{(<Declaration of Condition>;)}

ConcNO (<Declaration of Conclusion);

定义领域知识规则所属的规则组标识，字符串；

<Declaration of Rule Group>::=<Group Name>

定义领域知识规则的条件：是由函数、输入记录的字段名称、常量、判断运算符、算术运算符、字符串运算符、布尔运算符组成。

<Declaration of Condition>::=

<CondNO><ID>;

<ResField>(<Declaration of Restrict String>;)

<BriefDesc>(<Brief Description>;)

[<DetailDesc>(<Detail Description>;)]

[<KnowledgeList>(<Declaration of Knowledge>);{(<Declaration of Knowledge>;)}]

[<CalDir>(<Declaration of Illustration>;)]

定义限制字段，字符串，条件所限制的字段名，不能为空串；

<Declaration of Restrict String>::=String

简短描述，字符串，对条件简明扼要的描述，不能为空串；

<Brief Description>::=String

详细描述，字符串，可以为空，若条件的简明描述不甚明了，可对条件进行详细描述。

<Detail Description>::=String

定义条件所使用的领域知识，可以为空，如下所示：

<Declaration of Knowledge>::=

<KnoNO>(ID)

<KnoType>(<Declaration of Knowledge Type>;)

<Concept>(<Declaration of Concept>);{(<Declaration of Concept>)})

<Element>(<Declaration of Element>),{(<Declaration of Element>)})

定义领域知识规则知识的数据类型，字符串型，分为 isa(是一个)、ispart(是...的一部分)两种：

<Declaration of Knowledge Type>::=isa|ispart

定义概念名词，字符串，不能为空：

<Declaration of Concept>::=String

定义集合元素，字符串，格式为用逗号隔开的知识项，不能为空

<Declaration of Element>::=String

定义领域知识规则的结论：

<Declaration of Conclusion>::=<ConcNO><ConcStr>

<ConcStr>::={结论本身或结论的计算调用说明，字符串，不能为空}

4.4.2 清洗动作规则的巴科斯范式定义

清洗动作规则的范式定义如下[27]：

<DataClean Rule>:=

RuleNo<ID>;

RuleType (<Declaration of Rule Type>);

CondList (<Declaration of Condition>);{(<Declaration of Condition>;)}

[RecordSetList (<Declaration of RecordSet);{(<Declaration of RecordSet;)}]

[FieldList (<Declaration of Field);{(<Declaration of Field;)}]

[KeyList (<Declaration of Key);{(<Declaration of Key;)}]

[DealPolicy<Ignore|Throw Out|Automation>]

[DealScript (Declaration of Deal Script)].

定义清洗动作规则的类型：

<Declaration of Rule Type>::=1-输入错误|2-字段值缺失|3-字段值不在值域范围内|4-单字段中包含多个字段值|5-相似重复记录的合并|6-引用字段值缺失或错误|7-不符合业务规则约束|8-记录集之间不符合关联约束|9-异常|10-手工|11-自定义

定义清洗动作规则的条件：是由函数、输入记录的字段名称、常量、判断运算符、算术运算符、字符串运算符、布尔运算符组成。

<Declaration of Condition>::=<RecordSet Name><Condition Expression>

<Condition Expression>::=[NOT]<Condition>[AND|OR<Condition>]

<Condition>::=<operand><comparison operator><operand>|<Boolean Function>

<comparison operator>::=">"|">="|"<"|"<="|"=="|"<>"

<Boolean Function>::=<Boolean Function Name>(<Function Parameter>)

<Function Parameter>::=<operand>[,<operand>]

<operand>::=<RecodeSet Name>.<Field Name><Key Name>|<Const>|<Function>

[<operator><operand>]

<operator>:="+"|"-"|"*"|"/"

<Function>::=<Function Name>" ("<operand>")"

定义记录集的名称：

<Declaration of RecordSet>::=<RecordSet Name>

定义字段的名称：

<Declaration of Field>::=<Field Name>

定义关键字的名称：

<Declaration of Key>::=<Key Name>

定义处理脚本：是由输入记录的字段名称、连接符"+"和七种基本操作组成；连接符"+"表示基本操作的组合。脚本中基本操作的顺序就是执行时的工作流。七种基本操作将在 4.4.3 中加以描述。

<Declaration of Script>::=<Operarion Expression>

<Operarion Expression>::=<Operator>[<RecodeSet Name>][<Field Name>][<Key Name>][+<Operater>[<RecodeSet Name>][<Field Name>][<Key Name>]]

<Operator>::=Format|Combine|Divide|Select|Filter|RefModify|Aggregate

与领域知识规则的定义相比较，清洗动作规则没有 RuleGroup 和 ConcNo 两项，即规则组和结论两项。这是因为在清洗动作规则中，选择清洗动作规则的类型作为分组的标准，清洗动作规则的类型本身就可以看做结论。

4.4.3 清洗动作规则执行的基本操作

清洗动作规则执行的基本操作有以下几种：格式化(Format)、合并(Combine)、分割(Divide)、选择(Select)、过滤(Filter)、关联修正(RefModify)、聚集(Aggregate)[27]。

格式化主要针对于数据实例清洗，用于错误数据清除和语义定义不一致数据的替换，这种方法需要使用替换列表来将整个数据或者其中部分数据映射替换为正确的数据。替换过程中首先需要发现原值中需要替换的项，然后搜索替换列表中的替换项。

合并主要针对相似重复记录的消除，检查整个数据是否有相似重复记录存在，并且消除这些冗余冲突。合并一般需要先使用格式化对数据进行预处理清除错误数据或语义替换。在合并操作中一般需要包含适用的匹配函数和合并函数。

分割主要是为消除模式冲突，可以将单个属性分割成多个属性或者将多个属性的合并成一个属性，也可以和选择结合将一个表中属性分割成几个不同意义的属性(值不变)或者分割到几个表的属性中。

选择主要针对模式的冲突，选择一个数据对象中的几个属性组成另外一个

对象，一般与合并、分割一起使用。该种方法可以理解为投影。

过滤主要与分割一起使用，可以通过某些属性的值来获取所需要的数据行。在统一数据模型的记录集定义中定义了数据过滤条件。

关联修正是专门针对数据值之间的关联错误的，主要检查关联属性的值是不是错误让用户自己进行选择替换。

聚集主要针对语义定义不一致中的不同层次上的信息冲突，使用聚集将低一层的数据信息聚集成高一层的数据信息。

4.4.4 DREBDDC 系统的规则数据库设计

在 DREBDDC 系统中使用数据库系统来持久化用户从前端界面输入的规则，包括领域知识规则和清洗动作规则。表的结构参照了 4.4.1 和 4.4.2 中对规则的 BNF 范式定义。

1. 领域知识规则表结构(Tb_KnowRule)

表 4-1 领域知识表结构

字段名	类型	备注
KnowRule _ID	Numeric(10)	非空，主键
KnowRuleGroup_ID	Numeric(10)	外键，非空
Cond_ExpStr	Varchar2(1024)	非空
Result_ExpStr	Varchar2(1024)	非空

KnowRule _ID 是标识每条规则的主键；KnowRuleGroup_ID 为外键，引用 Tb_KnowRuleGroup 表主键；Cond_ExpStr 定义业务规则的条件表达式列表，多个表达式用特殊字符分割（如"#"等），各表达式间是合取关系。每个表达式由函数、输入记录的字段名称、常量、判断运算符、算术运算符、字符串运算符、布尔运算符组成。Result_ExpStr 是结论本身或结论的计算调用说明。

2. 领域知识规则组表结构(Tb_KnowRuleGroup)

表 4-2 领域知识组表结构

字段名	类型	备注
KnowRuleGroup_ID	Numeric(10)	主键，用于标识每一组规则，非空
Group_Name	Varchar2(50)	规则组名，用于标识一组规则，非空
Group_Desc	Varchar2(255)	规则组描述

3.清洗动作规则表结构(Tb_CleanRule)

表 4-3 清洗动作规则表结构

字段名	类型	备注
CleanRule_ID	Numeric(10)	主键，用于标识一条清洗规则，非空
CleanRule_Type	Numeric(4)	定义清洗动作规则的类型：1.输入错误；2.字段值缺失；3.字段值不在值域范围内；4.单字段中包含多个字段值；5.相似重复记录的合并；6.引用字段值缺失或错误；7.不符合业务规则约束；8.记录集之间不符合关联约束；9.异常；10.手工；11.自定义。非空
Cond_ExpStr	Varchar2(1024)	定义清洗动作规则的条件表达式列表，多个表达式用特殊字符分割，各表达式间是合取关系，是由函数、输入记录的字段名称、常量、判断运算符、算术运算符、字符串运算符、布尔运算符组成，非空
RecsetNames	Varchar2(1024)	定义清洗动作规则所操作记录集的名称列表，用特殊字符分割，非空
EieldNames	Varchar2(1024)	字段名称列表
DealPolicy	Numeric(2)	处理策略：1.忽略 2.抛出异常 3.自动
DealActionExp	Varchar2(1024)	是多个操作表达式的组合，操作表达式的顺序就是执行时的工作流顺序；<处理动作表达式>::=<操作表达式>；<操作表达式>::=<操作方法>[<记录集名>][<字段名>][+操作表达式] <操作方法>::=格式化|合并|分割|投影|过滤|引用性修改|聚合

4.4.5 DREBDDC 系统功能模块设计与分析

本系统由以下 5 个系统功能模块构成，其相互关系如 4.4.6 节的图 4-1 所示。

1. 领域知识规则和清洗动作规则的输入及持久化模块

提供基于 B/S 结构的 Web 表单页面，包括领域知识规则输入页面和清洗动作规则输入页面，让熟悉待清洗数据源结构及相关领域知识的专业人员输入各种数据清洗规则（领域知识规则，清洗动作规则），表单的内容与上述的数据库表结构一致。用户提交表单数据后，在后台服务器上调用通用数据访问接口将这些规则持久化到数据库中，用于生成规则配置文件。

2. Drools 2.1 规则配置文件(.drl 文件)动态生成模块

当用户选择执行某些清洗规则时，从数据库中取出对应的规则数据，然后按照 Drools 规则配置文件的格式生成.drl 文件。因为此文件是 XML 格式的，此过程可以采用 DOM4J 生成配置文件。在规则配置文件的 Condition 结点和 Consequence 结点中是 Java 语言可执行的语句代码，因此必须先调用表达式扫描转换模块将这些表达式将它们转换成 Java 代码。

3. 规则引擎执行模块

这是系统的总控模块，负责调用通用数据访问接口模块从待清洗数据源中读入数据，并根据用户选择的规则从数据库中读取规则数据，然后调用规则配置文件动态生成模块将其转换为 Drools 规则配置文件，最后对待清洗数据调用规则引擎核心执行规则配置文件中定义的领域知识规则和清洗动作规则,调用通用数据访问接口将清洗后数据持久化到目标数据库中。

4. 通用数据存取接口模块

采用基于 JDBC 的数据库连接池技术取得数据库连接，按照调用者的要求完成数据读取，主要包括以下几部分：

1）调用者输入数据库名，表名，查询语句，返回对对应于该数据表的对象化数据，如果出现错误，则抛出相应的异常；

2）调用者输入数据库名，表名，数据更新语句，将要持久化的对象化数据，将其持久化到指定数据库的相应表中。如果出现错误，返回 false，并抛出相应的异常，否则返回 true，表示操作成功。

5. 表达式扫描转换模块

本模块主要处理以下的表达式有下面两种：

1）领域知识规则表中的条件表达式(Cond_ExpStr)及结果表达式(Result_ExpStr)；

2）清洗动作规则表中的条件表达式(Result_ExpStr)及处理动作表达式(DealActionExp)。

本模块的作用是用将上述表达式转换成 Java 可执行的代码，并嵌入到生成的规则配置文件中。

转换的过程如下：

1）逻辑表达式，例如对于表 student，字段 age，用户从前台输入的表达式为：

"student.age>=25 && student.age<=30"

则转换后变为：

Student.getAge()>=25 && Student.getAge()<=30

这里，表 student 被映射为对象 Student，映射方法按照标准的对象/关系映射原则进行，其它算术运算符和逻辑运算符则保持不变。

2）赋值表达式，主要用于清洗动作规则的处理表达式中，例如对表 student，字段 addr，用户输入的表达式为：

Student.addr="陕西省汉中市"

则转换后变为：

Student.setAddr("陕西省汉中市");

这一将准代码字符串转换为 Java 语言可识别的程序代码的功能由一个词法分析模块提供，它对输入的表达式字符串进行分析，并输出 Java 代码串。

4.4.6 DREBDDC 的系统架构

DREBDDC 的系统架构如图 4-1 所示。

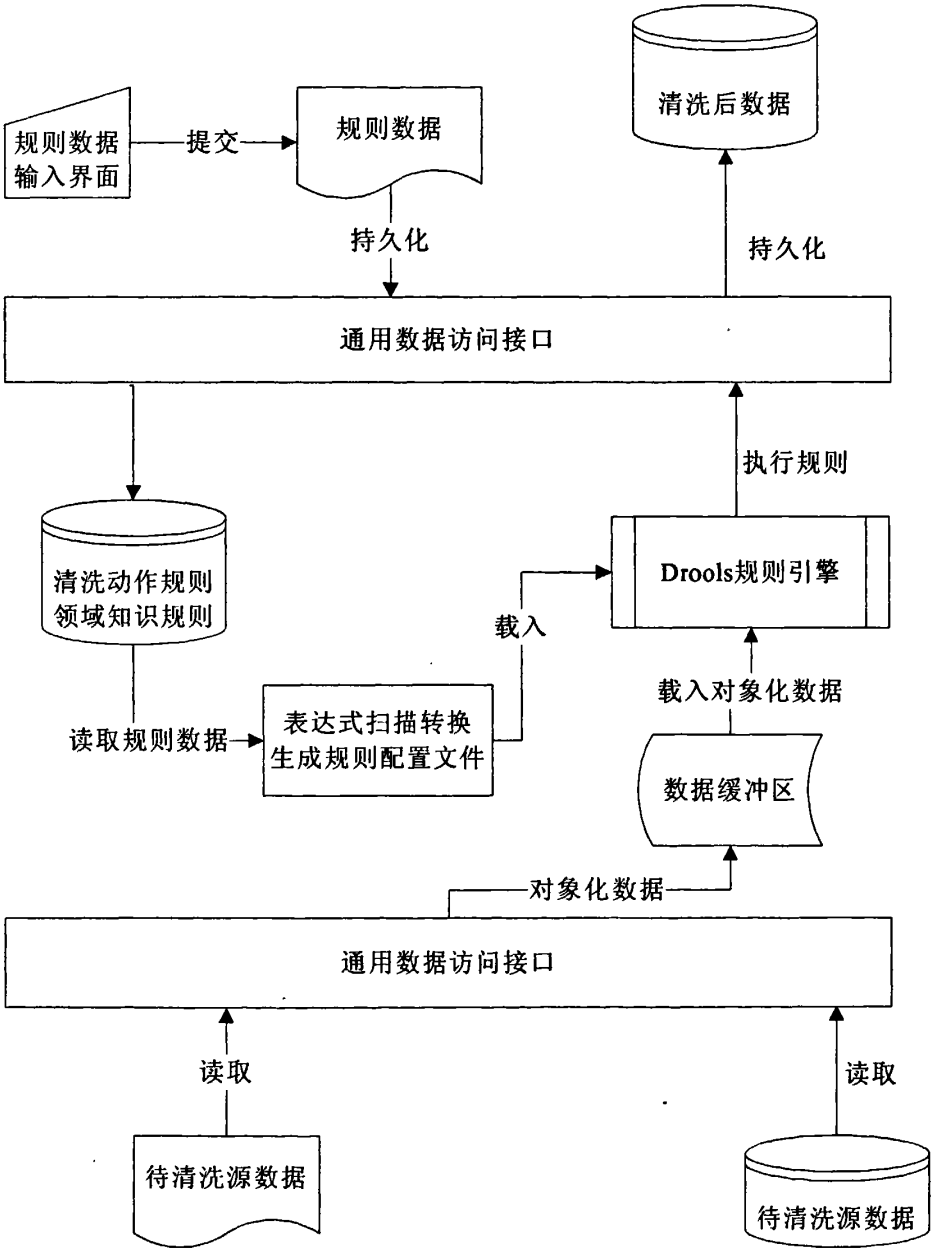

图 4-1 DREBDDC 的系统架构图

如上图所示，利用通用数据访问接口加载待清洗数据到内存缓冲区中；通过规则引擎加载生成的规则配置文件；对数据缓冲区中的数据执行规则；将清洗完成的数据持久化到数据库中。其中，待清洗多数据源包括： ODBC 数据源、非 ODBC 类关系型数据库数据源、应用数据、电子商务数据、各种文本数据等。

4.4.7 DREBDDC 系统的工作流程

DREBDDC 系统的工作流程如图 4-2 所示。

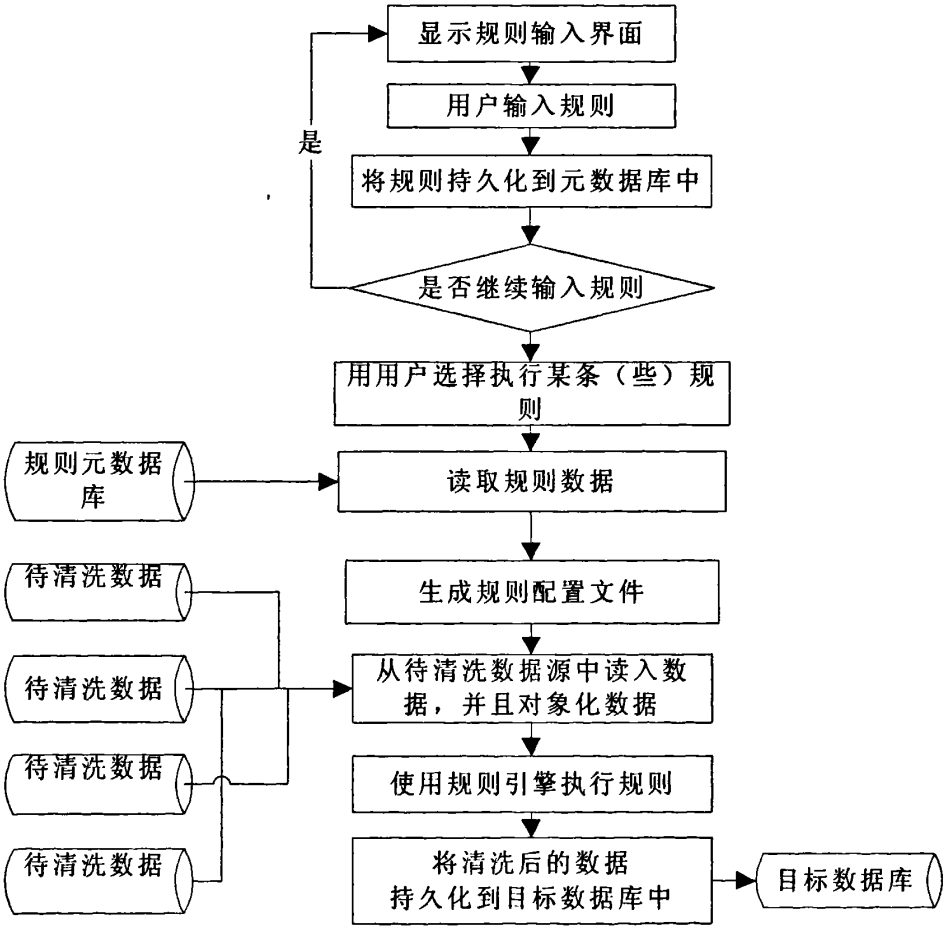

图 4-2 DREBDDC 系统的工作流程

4.4.8 DREBDDC 系统的开发运行环境

系统的开发运行环境如下：

开发语言：Java (JDK 1.4)

开发工具：Eclipse3.2+MyEclipse5.5

操作系统：Window 2000 Advanced Server

数据库服务器：Oracle9i+MSSQLServer 2000+MySQL 5.0.22

应用服务器：BEA WebLogic 8.14

4.4.9 DREBDDC 系统核心功能实现代码介绍

1.以下是 DREBDDC 系统的数据清洗转换类的部分代码。它从指定路径读取 Drools 规则配置文件（使用 loadRules()方法），以便生成规则库，然后将其应用到由参数传入的数据对象上，并将转换后的数据持久化到目标数据库中。

```
public class Transformer
{
    private static String   filepath;
    private static RuleBase    businessRules    = null;
        private static void loadRules() throws Exception
    {
        if (businessRules == null)
        {
            businessRules = RuleBaseLoader.loadFromUrl(Transformer.class
                                .getResource(CLEAN_RULE_FILE));
        }
    }
        public static void transform(Data dataToProcess)
    {
        try
        {
            loadRules();
        }
```

```
        catch (Exception e1)
        {
            e1.printStackTrace();
        }
    WorkingMemory workingMemory = businessRules.newWorkingMemory();
    workingMemory.addEventListener(new DebugWorkingMemoryEventListener());
        try
        {
            workingMemory.assertObject(dataToProcess);
            workingMemory.fireAllRules();
        }
        catch (Exception e2)
        {
            e2.printStackTrace();
        }
    }
    public static String getFilepath()
    {
        return filepath;
    }
    public static void setFilepath(String filepath)
    {
        BusinessLayer.filepath = filepath;
    }
}
```

2. 以下是 DREBDDC 系统的规则配置文件动态生成类的部分代码。这段代码的功能是从数据库中读取规则元数据，然后调用 Dom4J 软件包以生成 Drools 规则引擎可以识别的规则配置文件——drl 文件，并存入到指定目录。

```
public class RulesUpdater
{
    public boolean FormatRules(String filename)
```

```
    {
        boolean returnValue = false;
        try
        {
            SAXReader saxReader = new SAXReader();
            Document document = saxReader.read(new File(filename));
            XMLWriter writer = null;
            OutputFormat format = OutputFormat.createPrettyPrint();
        writer = new XMLWriter(new FileOutputStream((filename)), format);
            writer.write(document);
            writer.close();
            returnValue = true;
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
        return returnValue;
    }
    public boolean ModifyRules(String sfilename, String tfilename,
        String[] conditions, String concequence, String rulename,String
VOName)
    {
        boolean issuccess = false;
        try
        {
            SAXReader saxReader = new SAXReader();
    Document document = saxReader.read(new FileInputStream(sfilename));
            Element rule_set = document.getRootElement();
            Element temp = null;
            Element rule_x = rule_set.addElement("rule");
```

```
        rule_x.addAttribute("name", rulename);
        Element para = rule_x.addElement("parameter");
        para.addAttribute("identifier", "student");
        Element clazz = para.addElement("class");
        clazz.setText(VOName);
        for (int i = 0; i < conditions.length; i++)
        {
            temp = rule_x.addElement("java:condition");
            temp.setText(conditions[i]);
        }
        Element consequence = rule_x.addElement("java:consequence");
        consequence.setText(concequence);
        issuccess = true;
        try
        {
            XMLWriter writer = new XMLWriter(
                    new FileOutputStream(tfilename));
            writer.write(document);
            writer.close();
            issuccess = true;
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
    return issuccess;
```

```
    }
}
```

下面是上述 RulesUpdater 类的 ModifyRules 方法生成的一个规则配置文件，用于执行如下清洗逻辑：判断每条记录的 Name 字段是否为空，如为空，则置为缺省值："张三"。

```
<?xml version="1.0"?>
<rule-set name="Salary" xmlns="http://drools.org/rules"
    xmlns:java="http://drools.org/semantics/java"
    xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
    xs:schemaLocation="http://drools.org/rules rules.xsd
http://drools.org/semantics/java java.xsd">
    <java:import>java.lang.Object</java:import>
    <java:import>java.lang.String</java:import>
    <java:import>drebddc.vo.Student</java:import>
    <java:functions>
public static void printStudent(Student student){
System.out.println("\nStudent Name:"+student.getStudentName()
+"\n Sudent Age: "+student.getStudentAge()
+"\n Student Sex:"+student.getStudentSex()
+"\n Recommend "+student.getStudentName()
}
</java:functions>
<rule name="isSalaryNull">
<parameter identifier="data">
<class> drebddc.vo.Student </class>
</parameter>
<java:condition>data.getName()==null</java:condition>
<java:consequence>data.setName("张三");</java:consequence>
</rule>
</rule-set>
```

该文件中定义了一个规则集(见<rule-set>标记)，其中可以包含多条规则(见

<rule>标记)。每个规则中又定义了很多逻辑条件(见<java:condition>标记)。这些逻辑条件间是“与”的关系。Drools 规则引擎将根据这些逻辑条件动态地构建最优匹配树，高效地检验数据(见<parameter> 标记) 是否符合逻辑条件。如是，则触发<java:consequence>标记中的 Java 代码，在此处可判断任意逻辑(例如，根据记录值到另一数据库中查询参照值以决定当前记录是否是脏数据)，执行任意动作(保留、修复或丢弃)；如否，则不触发。

在实际中用到的规则前件可能是一个非常复杂逻辑表达式，后件可能是上述的七种基本操作或简单的赋值操作等，而这些前件和后件都是由专业人员定义的。

3.以下是 DREBDDC 系统的通用数据访问模块的部分代码：

```
public class CommUtils
{
 private static final Logger     logger= Logger.getLogger(CommUtils.class);
 private static final String LOG4JFILEPATH="com\\drebddc\\config\\log4j.xml";
 private static final String
 DATASOURCECONFIGPATH="com\\drebddc\\config\\datasource.properties";
    static
    {
        URL logUrl = Thread.currentThread().getContextClassLoader()
                .getResource(LOG4JFILEPATH);
        DOMConfigurator.configure(logUrl);
    }
    private static CommUtils  instance                = null;
    private CommUtils()
    {
    }
    public static final CommUtils getInstance()
    {
        if (instance == null)
        {
            instance = new CommUtils();
```

```
        }
        return instance;
    }
    private static final String getConfigParaByName(String paraname)
    {
        Properties props = new Properties();
        String retvalue = null;
        InputStream istream = null;
        istream = Thread.currentThread().getContextClassLoader()
                .getResourceAsStream(DATASOURCECONFIGPATH);
        try
        {
            props.load(istream);
            retvalue = props.getProperty(paraname);
        }
        catch (IOException e)
        {
            e.printStackTrace();
            logger.error(e.toString());
        }
        finally
        {
            try
            {
                if (istream != null)
                {
                    istream.close();
                }
                istream = null;
            }
            catch (IOException e)
```

```
            {
                e.printStackTrace();
                logger.error(e.toString());
            }
        }
        return retvalue;
    }
    public Connection getConnection()
    {
        DataSource ds = null;
        Context ctx = null;
        Connection myConn = null;
        try
        {
            ctx = getInitialContext();
            ds = (javax.sql.DataSource) ctx
                    .lookup(getConfigParaByName("datasource.name"));
        }
        catch (Exception e)
        {
            e.printStackTrace();
            logger.error(e.toString());
        }
        try
        {
            myConn = ds.getConnection();
        }
        catch (Exception e)
        {
            e.printStackTrace();
            logger.error(e.toString());
```

```
    }
    return myConn;
}
public void cleanup(ResultSet rs, Statement cstmt, Connection conn)
{
    try
    {
        if (rs != null)
        {
            rs.close();
            rs = null;
        }
        if (cstmt != null)
        {
            cstmt.close();
            cstmt = null;
        }
        if (conn != null)
        {
            conn.close();
            conn = null;
        }
    }
    catch (Exception e)
    {
        e.printStackTrace();
        logger.error(e.getMessage());
    }
}
private static Context getInitialContext() throws Exception
{
```

```
        Properties properties = null;
        InitialContext context = null;
        try
        {
            properties = new Properties();
            properties.put(Context.INITIAL_CONTEXT_FACTORY,
                    getConfigParaByName("context.initialfactory"));
            properties.put(Context.PROVIDER_URL,
                    getConfigParaByName("context.provider.url"));
            context = new InitialContext(properties);
        }
        catch (Exception e)
        {
            e.printStackTrace();
            logger.error(e.toString());
        }
        return context;
    }
}
```

下面是 datasource.properties 文件的一部分内容：

```
datasource.name=ds/oracle/drebddc
context.provider.url=t3://127.0.0.1:7001
context.initialfactory=weblogic.jndi.WLInitialContextFactory
```

以上代码通过 JDBC 连接池方式取得数据库连接，连接池配置在应用服务器 WebLogic8.14 上，根据 datasource.properties 文件的内容进行。CommUtils 类主要提供了一个用于取得数据库连接的方法：public Connection getConnection()。该类还采用了 Log4J 提供了记录日志的功能。

通用数据访问接口能够跨平台、网络访问数据，支持在不同类型数据源间建立连接。通过连接池方式取得数据库连接是为了提高系统的运行效率。

4.5 DREBDDC 系统实验结果性能分析

4.5.1 实验数据

本文对DREBDDC系统的性能进行了实验测试。实验分别记录了使用Drools规则引擎和硬编码（即将清洗逻辑直接嵌入到程序代码中）条件下，在相同清洗规则数量及相同记录数量的两种情况下的单数据源数据清洗时间。

实验采用的部分待清洗源数据如表4-4所示。

表4-4 测试用待清洗数据（部分）

Student_id	Name	Hometown	MajorKey	Sex
1001	'ZZZ'	'SX'	001	'F'
1002	'BBB'	'HB'	005	'M'
1003	'CCC'	'HZ'	008	'F'
1004	'DDD'	'HZ'	004	'M'
1005	'EEE'	'SZ'	005	'F'
1006	'FFF'	'WH'	008	'M'
1007	'GGG'	'TY'	007	'F'
1008	'HHH'	'BJ'	010	'M'
1009	'III'	'SH'	009	'F'
1010	'JJJ'	'WH'	012	'M'
1011	'KKK'	'TY'	013	'F'
1012	'LLL'	'XA'	017	'M'
1013	'MMM'	'WN'	015	'F'
1014	'NNN'	'AK'	005	'M'
1015	'OOO'	'HT'	001	'F'
1016	'PPP'	'NZ'	002	'M'

测试用到的部分清洗规则及相应硬编码如表 4-5 所示。

表 4-5 测试用清洗规则及相应硬编码对照表（部分）

清洗规则	硬编码
<parameter identifier="stu"><class>Student</class></parameter><java:condition>stu.getName()=="ZZZ"</java:condition><java:consequence>stu.remove();</java:consequence>	if (stu.getName() =="ZZZ") stu.remove();
<parameter identifier="stu"><class>Student</class></parameter><java:condition>stu.getMajorKey()+1000!=stu.getStudent_Id()</java:condition><java:consequence>stu.remove();</java:consequence>	if (stu.getMajorKey() +1000!= stu.getStudent_Id()) stu.remove();
<parameter identifier="stu"><class>Student</class></parameter><java:condition>stu.getSex()=="F"</java:condition><java:consequence>stu.setSex("女");</java:consequence>	if (stu.getSex() =="F") stu.setSex("女");
<parameter identifier="stu"><class>Student</class></parameter><java:condition>stu.getSex()=="M"</java:condition><java:consequence>stu.setSex("男");</java:consequence>	if (stu.getSex() =="M") stu.setSex("男");

在表 4-5 中：第一条规则过滤掉学生姓名为"ZZZ"的记录；第二条规则过滤掉学生专业代码和学号不满足条件的记录；第三条规则对性别的表示进行转换。第四条规则与第三条规则的作用类似。

部分清洗后数据如表 4-6 所示。

表 4-6 清洗后数据

Student_id	Name	Hometown	MajorKey	Sex
1004	'DDD'	'HZ'	004	'男'
1005	'EEE'	'SZ'	005	'女'
1007	'GGG'	'TY'	007	'女'
1009	'III'	'SH'	009	'女'

实验测试结果数据如表 4-7 所示。

表 4-7 DREBDDC 系统实验测试结果数据

清洗规则数量（条）	记录数量（条）	使用 Drools 规则引擎的清洗时间（ms）	使用 Drools 规则引擎的单位清洗时间（ms/记录）	使用硬编码的清洗时间（ms）	使用硬编码的清洗时间/使用 Drools 规则引擎的清洗时间(比率)
3	10	13.18	1.32	0.92	0.07
3	100	28.48	0.28	6.63	0.23
3	1000	198.14	0.20	81.57	0.41
3	10000	775.45	0.08	328.56	0.42
12	10	16.25	1.63	0.97	0.06
12	100	38.29	0.38	9.12	0.24
12	1000	209.68	0.21	88.37	0.42
12	10000	951.15	0.10	348.29	0.37
30	10	19.21	1.92	1.43	0.07
30	100	80.09	0.80	10.24	0.13
30	1000	282.26	0.28	90.76	0.32
30	10000	1040.85	0.10	592.38	0.57

4.5.2 实验结果分析

1. 在记录条数相同的情况下，随着清洗规则数量的增加，使用 Drools 规则引擎的清洗时间缓慢上升。这说明 Drools 规则引擎的性能受规则数量的影响不是很大。如图 4-3 如示。

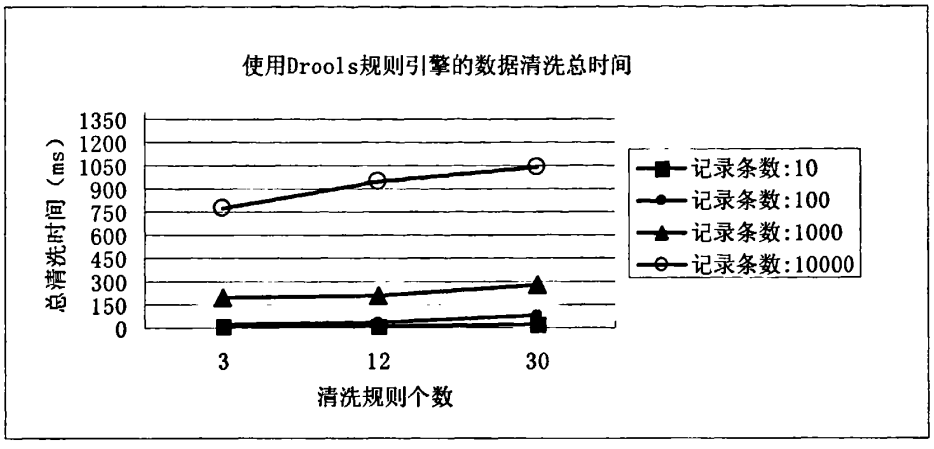

图 4-3 使用 Drools 规则引擎的数据清洗时间

2. 随着记录数的增加，单条记录清洗时间快速下降。当单表记录数大于 10000 时，单条记录清洗时间稳定在 0.2ms 左右，这说明当记录数很大时，单条记录的平均清洗时间趋于稳定，这说明系统在处理大量数据时性能比较理想。如图 4-4 所示：

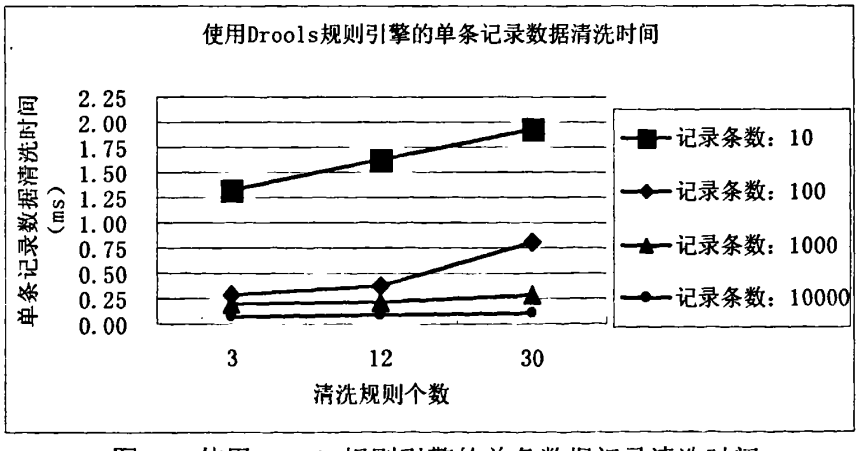

图 4-4 使用 Drools 规则引擎的单条数据记录清洗时间

3. 当记录数大于 1000 时，使用硬编码的数据清洗性能和使用 Drools 规则引擎的性能之比稳定在的 40%左右。这说明当记录数足够大时，使用 Drools 规则引擎和使用硬编码清洗总时间之比趋于稳定。尽管使用 Drools 规则引擎的清洗总时间只有使用硬编码清洗总时间的 40%，但性能损失换来的是系统的通用性和规则的动态可配置性。如图 4-5 所示：

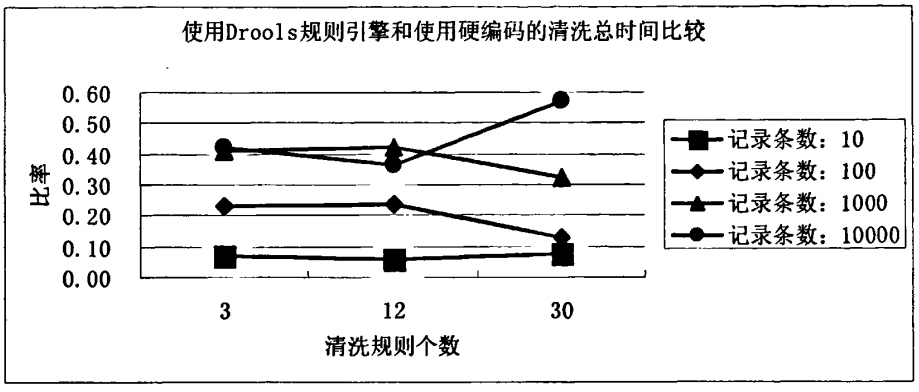

图 4-5 使用 Drools 规则引擎和使用硬编码清洗总时间比较

4.6 本章小结

本章阐述了基于 Drools 规则引擎的动态数据清洗系统的设计方案。首先给出了系统使用的两种规则，并给出了其巴科斯范式定义，从而为持久化规则提供了基础。接着详细介绍了系统的规则元数据库设计，系统功能模块划分，系统结构，工作流程，系统的开发运行环境，并给出了一些主要模块的部分代码。最后给出了系统的实验性能分析结果。测试结果表明：Drools 规则引擎的性能受规则数量的影响不是很大；当记录数很大时，单条记录的平均清洗时间趋于稳定；当记录数足够大时，使用 Drools 规则引擎和使用硬编码清洗总时间之比趋于稳定。尽管使用 Drools 规则引擎的清总时间只有使用硬编码清洗总时间的 40%，但性能损失换来的是系统的通用性和规则的动态可配置性。

第 5 章 总结和展望

5.1 总结

在运营管理过程中，企业积累了大量的电子数据，这些数据非常重要。为了更好地发挥企业信息系统的作用，必须提高信息系统的数据质量。因此，研究企业信息系统的数据清洗问题具有理论和应用价值。

本文主要做了以下几个方面的工作：

1. 分析了数据清洗问题产生的原因和提出的背景，给出了数据质量的概念及评价指标，数据清洗技术在各个应用领域中的定义以及国内外对数据清洗技术研究和应用的现状。总结了数据清洗技术的原理和方法，数据清洗的评价标准，并对数据清洗的基本流程进行了详细的描述。

2. 分析了规则引擎的原理，产生的背景及其使用方法。

3. 研究分析了一种开源的 Java 规则引擎软件包——Drools，并系统地研究分析了其 API 的使用方法和规则配置文件的结构和含义。

4. 研究分析了领域知识和清洗动作规则的巴科斯范式(BNF, Backus-Naur Form)定义。

5. 提出了采用 Drools 规则引擎执行清洗逻辑，能够处理多种数据质量问题的数据清洗架构——DREBDDC，从而解决了现有数据清洗工具依赖于复用性差的硬编码或灵活却低效的人工判断来进行数据清洗的问题。

5.2 展望

数据清洗技术发展的空间还很大，还有许多的工作要做，随着信息技术的发展，又给数据清洗技术提出了新的挑战。

1. 本文将 Java 规则引擎 Drools 就用于数据清洗，但是清洗规则却是依赖于一些清洗算法的，因而并没有降低数据清洗的算法复杂性。

2. 由于 Drools 规则引擎需要占用较大的内存空间，对系统配置要求很高，今后可考虑使用分批载入，利用外存等方式来节省内存作用，也可考虑采用性

能较好的数据缓存软件包将规则引擎引用的数据对象进行缓存，如 encache，oscache 和 swarmcache 等开源的软件包。

3. 由于待清洗数据源的复杂性，数据清洗具有较大的难度。在某些情况下，需要业务人员和专业技术人员进行领域知识规则和清洗动作规则的定义和输入，因此对规则定义者的要求较高。

4. 本文对基于 Drools 规则引擎的数据清洗方法的研究仍处于理论阶段，还没有把该方法应用到具体的实践中去。以后还需要进一步完善这种数据清洗方法的实践环节，并把它加入到数据清洗软件平台中去，成为一种实用的工具。

参考文献

[1] Kolber, A., Hay, D., Healy, K.A., and Hall, J. et al.GUIDE Business Rules Project.[OL] Final Report.Business Rules Group (www.businessrulesgroup.org), Chicago, 1997

[2] Ronald G Ross. Principles of the Business Rule Approach [M]. Addison-Wesley, 2003:8

[3] 闫丽萍，潘正运，梁冰等.基于业务规则管理技术的系统开发方法分析[J].信息工程大学学报, 2006.7(2):141~143+171

[4] 夏建军. 规则引擎：业务逻辑与应用分离的利器[OL]. http://blog.vsharing.com/Article.aspx?aid=433287. 2006.12

[5] Qusay H. Mahmoud. Getting started with the Java Rule Engine API (JSR 94).Toward Rule-Based Applications [OL]. http://lava.sun.com/developer/technicalArticles/J2SE/JavaRule.html.2005-7-26

[6] 刘际.规则引擎在业务逻辑层中应用的研究：[硕士学位论文].大连：大连海事大学. 2007

[7] Erhard Rahm, Hong Hai Do. Data Cleaning: Problems and Current Approaches[J].IEEE Data Engineering Bulletin, 2000, 23(4):3~13

[8] 郭志懋，周傲英. 数据质量和数据清洗研究综述[J]，软件学报，2002.13(11):2076~2081

[9] 俞荣华.数据质量和数据清洗关键技术研究：[硕士学位论文].上海：复旦大学:2002

[10] Newcombe H B.Kennedy J M.Axofrd S J. et al.Automatic linkgae of vital records[J], Science, 1959, 130:954~959

[11] 谭亚竹. 基于 XML 数据清洗的应用研究：[硕士学位论文].重庆：重庆大学:2006

[12] Duncan K, Wells D. A Rule Based Data Cleansing [J]. Data Warehousing, 1999, 4(3): 2

[13] 刘伟.Java 规则引擎—Drools 的介绍及应用[J].微计算机应用.2005.26(6):717~721

[14] 缴明洋，谭庆平. Java 规则引擎技术研究[J]. 计算机与信息技术. 2006 (03):41~43

[15] 何仁杰，梁冰等.用规则引擎替换代码[J].计算机世界报，2004-04-19: B07 版

[16] 黄翱，潘正运等. 业务规则引擎 ILog JRules 工作引擎的工作机制分析[J]. 微计算机信息. 2006.22(24)112~114+48

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

[18] 刘华.Web 信息集成中数据清洗的研究：[硕士学位论文].武汉：武汉理工大学.2007

[19] Aebi, D., Perrochon, L.Towards improving data quality[C]. In: Sarda, N.L., ed. Proceedings of the International Conference on Information Systems and Management of Data. Delhi, 1993. 273~281

[20] Wang, R.Y., Kon, H.B., Madnick, S.E. Data quality requirements analysis and modeling[C]. In: Proceedings of the 9th International Conference on Data Engineering. Vienna: IEEE Computer Society, 1993. 670~677

[21] Redman TC. The impact of poor data quality on the typical enterprise [J]. Communications of the ACM, 2001.36(7):79~82

[22] Khalil O E M, Harcar T D. Relationship marketing and data quality management [J]. Advanced management Jounal,1999.64(2):26~33

[23] Zeffane R, Cheek B, Meredith P. Does user involvement during Information System's development improve data quality?[J] Human Systems Management,1998.17(2):31~36

[24] Parent C. Spaccapietra S. Issues and approaches of database integration [J]. Communications of the ACM, 1998, 41(5):166~178

[25] Milo T. Zohar S. Using schema matching to simplify heterogeneous data translation[C]. In: Gupta A. Shmueli O. Widom J, eds. Proceedings of 24th International Conference on Very Large Databases.New York: Morgan Kaufmann.1998:122~133

[26] Galbardas H, et a1.An Extansible Framework for Data Cleaning [J]. Institutenational de Rechercheen Information et en automatique, Technical Report 1999

[27] 孟坚. 基于规则的交互式数据清洗技术：[硕士学位论文].南京：东南大学.2005.3

[28] Galhardas H. Florescu D. Shasha D. Declarative data cleaning: language, model, and algorithms[C]. In: Proceedings of the 27th Very Large DataBase Conference. Roma: Morgan Kaufinaim.2001:371~380

[29] Ernest J. Friedman-Hill，Jess, the Rule Engine for the Java Platform[OL],http://herzberg.ca.sandia.gov/jess/

[30] 鲍洪庆，石兵等，一个基于领域知识的数据清洗框架[J].信息技术与信息化，2005.17(5):101~102

[31] 孟坚，董逸生等.一种基于规则的交互式数据清洗技术[J].微机发展，2005.15(4): 142~143

[32] 张宁，贾自艳等.数据仓库中 ETL 技术的研究[J].计算机工程与应用.2002.38(24):213~216

[33] 查峰.数据仓库化中数据清洗问题的研究：[硕士学位论文].南京：东南大学:2002

[34] 邓莎莎，陈松乔.基于异构数据抽取清洗模型的元数据的研究[J].计算机工程与应用，2004.30:175~177

[35] 俞荣华，田增平，周傲英.一种检测多语言文本相似重复记录的综合方法[J].计算机科学，2002,29(1):118~121

[36] 张晓明，乔溪.数据清洗方法与构件的综合技术研究[J].石油化工高等学校学报，2005.18(6):68~69

[37] 余春红，许向阳.关系数据库中近似重复记录的识别[J].计算机应用研究，2003.20(9):36~37

[38] 邱越峰，田增平，季文等一种高效的检测相似重复记录的方法[J].计算机学报，2001.24(1):69~77

[39] Steve Demuth，Colleen McClintock．规则引擎及 J2EE 平台-灵活的企业应用平台[C]．Sun's 2003 Worldwide Java Developer Conference

[40] 张渊，夏清国．基于 Rete 算法的 Java 规则引擎[J]．科学技术与工程．2006.6(11) :1548~1550

[41] Jin L. Li C. Mebrotra S. Efficient record linkage in large data sets[C]. Eighth International Conference on Database Systems for Advanced Applications. Kyoto. 2003: 137~146

[42] Kaplan D. Krislman R. Assessing data quality in accounting information systems [J]. Communications of the ACM. 1998. 41(2): 72~78

[43] Jboss.org/Jboss Rules, Drools 3.0.6 API Documentation[OL], http://downloads.jboss.com/drools/docs/3.0.6/apidocs/index.html

[44] Mark Proctor, Michael Neale, Peter Lin, Michael Frandsen. Drools Documentation 3.0.6[OL]. http://downloads.jboss.com/drools/docs/3.0.6/html_single/index.html

[45] Timothy E. Ohanekwu. A pre and post data warehouse cleaning technique,[Master Paper]. Canada: Computer Science Department of University of Windsor, 2002:27

[46] Raman V, Hellerstein J. Potter's wheel: An interactive data cleaning system[C]. Proceedings of the 27th VLDB conference, Roma, Italy, 2001:381~390

附录一　Drools 2.1 规则配置文件的 XSD 格式定义

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
 xmlns:rules=http://drools.org/rules
xmlns:xs="http://www.w3.org/2001/XMLSchema"  elementFormDefault="qualified"
targetNamespace="http://drools.org/rules">
  <xs:element name="rule-set">
    <xs:complexType>
      <xs:sequence>
        <xs:element          maxOccurs="unbounded"          minOccurs="0"
ref="rules:abstractImport"/>
        <xs:element          maxOccurs="unbounded"          minOccurs="0"
ref="rules:application-data"/>
        <xs:element          maxOccurs="unbounded"          minOccurs="0"
ref="rules:abstractFunctions"/>
        <xs:element maxOccurs="unbounded" minOccurs="0" ref="rules:rule"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="description" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="rule">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" minOccurs="0" name="parameter">
          <xs:complexType>
            <xs:choice>
              <xs:element ref="rules:abstractClass"/>
```

```
                <xs:element ref="rules:class-field"/>
                <xs:element ref="rules:semaphore"/>
              </xs:choice>
              <xs:attribute name="identifier" type="xs:string" use="required"/>
            </xs:complexType>
          </xs:element>
          <xs:element maxOccurs="unbounded" minOccurs="0"
ref="rules:abstractCondition"/>
          <xs:element minOccurs="0" name="duration">
            <xs:complexType>
              <xs:attribute name="days" type="xs:nonNegativeInteger"/>
              <xs:attribute name="hours" type="xs:nonNegativeInteger"/>
              <xs:attribute name="minutes" type="xs:nonNegativeInteger"/>
              <xs:attribute name="seconds" type="xs:nonNegativeInteger"/>
            </xs:complexType>
          </xs:element>
        <xs:element ref="rules:abstractConsequence"/>
        </xs:sequence>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="salience" type="xs:integer"/>
        <xs:attribute name="no-loop" type="xs:boolean"/>
        <xs:attribute name="xor-group" type="xs:string"/>
        <xs:attribute name="description" type="xs:string"/>
      </xs:complexType>
      <xs:key name="ruleName">
        <xs:selector xpath="rules:rule"/>
        <xs:field xpath="@name"/>
      </xs:key>
    </xs:element>
    <xs:element name="class" substitutionGroup="rules:abstractClass"
type="xs:string"/>
```

```
<xs:element name="class-field">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="field" type="xs:string" use="required"/>
        <xs:attribute name="value" type="xs:string" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
<xs:element name="import" substitutionGroup="rules:abstractImport"
type="xs:string"/>
<xs:element name="semaphore">
  <xs:complexType>
      <xs:attribute name="type" use="required">
       <xs:simpleType>
         <xs:restriction base="xs:string">
           <xs:enumeration value="String"/>
           <xs:enumeration value="Integer"/>
           <xs:enumeration value="Long"/>
           <xs:enumeration value="Boolean"/>
           <xs:enumeration value="Char"/>
           <xs:enumeration value="Short"/>
           <xs:enumeration value="Float"/>
           <xs:enumeration value="Double"/>
             <xs:enumeration value="List"/>
             <xs:enumeration value="Map"/>
             <xs:enumeration value="Set"/>
         </xs:restriction>
       </xs:simpleType>
      </xs:attribute>
```

```
    </xs:complexType>
  </xs:element>
  <xs:element name="application-data">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute name="identifier" type="xs:string" use="required"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element abstract="true" name="abstractImport" type="xs:anyType"/>
  <xs:element abstract="true" name="abstractFunctions" type="xs:anyType"/>
  <xs:element abstract="true" name="abstractClass" type="xs:anyType"/>
  <xs:element abstract="true" name="abstractClassField" type="xs:anyType"/>
  <xs:element abstract="true" name="abstractCondition" type="xs:anyType"/>
  <xs:element abstract="true" name="abstractConsequence" type="xs:anyType"/>
</xs:schema>
```

附录二　攻读硕士学位期间发表的学术论文

[1] Wang Shunyan, Cao Yongliang, Zhong Luo.The Design and Implementation of Drools Rule Set's Dynamic Configuration[C]. Proceedings of the 2nd International Conference on Computer Science and Education,2007:934-937.被 SCI 收录（IDS Number: BGQ59, ISBN: 978-7-5615-2825-9）

[2] Wang Shunyan, Zhong Luo, Cao Yongliang. A Data Synchronization Mechanism for Cache on Mobile Client. [C] Proceedings of the Wireless Communications, Networking and Mobile Computing International Conference,2006:1-5.被 IEEE 收录

[3] 王舜燕主编.《Java 编程方法学》[M].北京邮电大学出版社.2008 年 8 月出版

致谢

首先要感谢的是我的导师王舜燕。王老师为人和蔼，治学严谨，辛勤敬业，注重开拓创新、锐意进取。研究生三年，得到了王老师的悉心指导，使我的研究和开发能力得到了很大的提高。王老师对事业的执着和勤勤恳恳的工作态度，在研究和做人方面都给了我很多的启示，我将终身难忘。

感谢所有教导过我的老师，他们给我的知识是本论文的基础。

感谢所有帮助过我的同学和朋友，他们伴随我度过了愉快和有益的三年。感谢我的父母，在漫长的求学生涯中，他们一直关心着我，鼓励着我，使我可以集中精力进行学习和研究。

在这里，我衷心地对关心我，爱护我，帮助我的父母，老师，同学道一声谢谢！

曹永亮

2007年10月于马房山

基于Java规则引擎的动态数据清洗研究与设计

作者：曹永亮
学位授予单位：武汉理工大学