分类号：TP311.5　　单位代码：10335

密　级：无　　学　　号：21351197

浙江大学

硕士学位论文

中文论文题目：基于金融实时风控规则自动化验证技术的研究与应用

英文论文题目：Research and Application of Rule Automatic Verification Technology in Financial Real-Time Risk Management

申请人姓名：吴必阳

指导教师：贝毅君 讲师

合作导师：

专业学位类别：工程硕士

专业学位领域：软件工程

所在学院：软件学院

论文提交日期 2015 年 4 月 20 日

学位论文书脊示例

基于金融实时风控规则自动化验证技术的研究与应用

吴必阳

浙江大学

基于金融实时风控规则自动化验证技术的研究与应用

论文作者签名:_____

指导教师签名:_____

论文评阅人 1：_____

评阅人 2：_____

评阅人 3：_____

评阅人 4：_____

评阅人 5：_____

答辩委员会主席：_____

委员 1：_____

委员 2：_____

委员 3：_____

委员 4：_____

委员 5：_____

答辩日期：_____

Research and Application of Rule Automatic Verification Technology in Financial Real-Time Risk Management

Author's signature:_____

Supervisor's signature:_____

Thesis reviewer 1: _____

Thesis reviewer 2: _____

Thesis reviewer 3: _____

Thesis reviewer 4: _____

Thesis reviewer 5: _____

Chair: _____
(Committee of oral defence)

Committeeman 1: _____

Committeeman 2: _____

Committeeman 3: _____

Committeeman 4: _____

Committeeman 5: _____

Date of oral defence: _____

浙江大学研究生学位论文独创性声明

本人声明所呈交的学位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。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 **浙江大学** 或其他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谢意。

学位论文作者签名：　　签字日期：　　年　　月　　日

学位论文版权使用授权书

本学位论文作者完全了解 **浙江大学** 有权保留并向国家有关部门或机构送交本论文的复印件和磁盘，允许论文被查阅和借阅。本人授权 **浙江大学** 可以将学位论文的全部或部分内容编入有关数据库进行检索和传播，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后适用本授权书）

学位论文作者签名：　　导师签名：

签字日期：　　年　　月　　日　　签字日期：　　年　　月　　日

摘要

在三方支付企业迅速崛起的今天，越来越多的企业关注金融风控规则的实用性，一条准确性高的规则可以控制很多不正常的交易流水，以此来控制企业的资金流失。规则的自动化验证也是在这种情况下产生的，它可以准确的验证规则的有效性，快速的上线进行使用，这对企业来说无疑是他们所期望的。

规则自动化验证技术的研究与应用使得金融风控规则的使用变得更加准确和高效。规则验证主要分为两块，第一块在规则投入使用之前，这一块会模拟交易流水数据进行规则的准确性验证，正确性达到一定指标后投入使用；第二块在规则使用后，这一块主要针对触发的风险再对规则做回归验证。第一块验证用到了缓存数据，这些缓存数据通过缓存处理插件进行分布式存储，采用 redis 存储系统，在规则平台上验证流水的时候，可以直接通过获取缓存数据进行规则验证，速度将大大提升。第二块验证用到了标准数据库，这些标准数据是由数据装载插件将流水数据转化得到，通过触发的风险和流水，结合标准库数据来回归统计对应规则的误报率，验证其准确性。

规则自动化验证技术的研究使用了数据装载、线程池、redis 和消息队列等相关技术。论文后面对规则自动化验证的应用进行了详细的说明，并对规则自动化验证进行了效果展示。

关键词：规则自动化验证，redis，消息队列，数据装载

Abstract

Today，as rapid rise of third party payment companies, more and more companies started to concern about the practicality of financial risk control rules. A rule with high accuracy could control a lot of under-usual transaction flow, by which way to control the financial loss for the company. Rules of automatic verifications thus came into being in such circumstances. It can accurately verify the validity of the rules and be fast applied on-line, which undoubtedly meet the expectations of the enterprises well.

Research and Application of Rules automatic verification technique makes the financial risk control rules more accurate and efficient. Rules of automatic verification mainly contains two parts.One is to verify the accuracy of the rules by monitoring the transaction,and then put into use when the accuracy reach a certain standard. The other is to verify the rules twice aimed to the risk triggered. The first verification applied the cache data, The cache data distributes by applying redid storage system, and to verify the rules by obtaining the cache data by verifying the transaction on ruled platform, by which way the speed will rise a lot. The second verification applied the standard data system, which is obtained by uploading plugins to transform the transaction datas. By combining the risk triggered and transaction, and the standard statistics to summarize the failure rate of the rules, and by this way to verify its accuracy.

Research of Rules automatic verification technique applied data loading, thread pool, redis and message queues, and other related technologies. There are detailed manifest about the rules automatic verification, and also effect show for it .

Key Words: rules of Automated validation, redis, message queues, data loading

目录

图目录

表目录

第1章 绪论

1.1 课题背景与意义

如今，随着国内金融业务的发展，针对金融业务的风险控制也在不断加强。从最近几年的数起金融风险事件来看，一些违法分子作案手法日趋专业，盗卡，盗刷，可疑交易，反洗钱，套现等等一系列案件频繁出现，一些简单传统的金融风险规则已无法控制风险资金的流失，现有的风险规则机制出现了大量的漏洞，提高风险规则的监控水平迫在眉睫[1]。随着第三方支付公司支付业务规模的迅速扩张，风险规则验证和配置的灵活性上已不能满足现有监控需求，为保障第三方支付公司、用户消费的资金安全，实现对异常交易进行阻断等功能，灵活的规则配置和验证极为重要，它将保证金融风控规则能否准确实时的投入使用。金融实时风控主要是对有关金融的交易流水进行风险识别，将有风险的用户进行归档，限制其后续交易的过程。在国内，第三方支付企业面临用户和商户的严重欺诈行为，恶意透支，盗刷，虚假交易，信用卡套现，商户欺诈，洗钱等风险事件不断增加[2][3]。这对第三方支付企业来说是一个极大的经营风险，对这些金融交易的风险控制刻不容缓，针对这些风险交易，金融实时风控能发挥其真正的作用。将金融实时风控模块以插件形式接入到三方支付系统，作为一个过滤器实时地过滤掉存在风险隐患的交易流水，从而保证三方支付系统正常的进行交易，具体的风险控制图 1.1 如下：

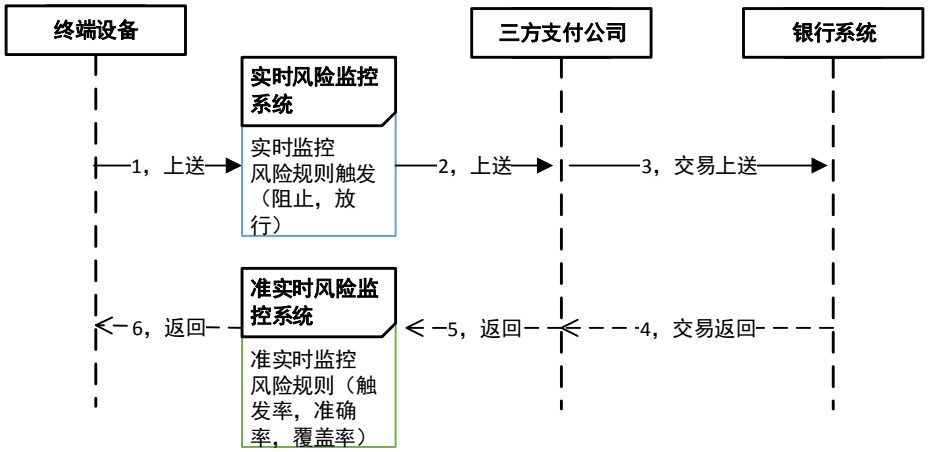

图 1. 1 三方支付风控图

金融实时风控最主要的一部分就是规则管理平台，在这个平台上可以灵活的定制相关规则，但这些规则如果得不到及时的验证也会影响其投入生产的效率。

例如风险人员发现目前流水中出现了一种规律化的模式交易，要立刻运行符合这种规律的新规则，如果规则的正确性和覆盖性得不到好的验证，将会大大影响规则的及时上线。本文主要结合金融实时风控中的规则平台，研究如何在金融规则平台上进行风险规则自动化验证以及简单的应用。

金融风控规则是规则平台一个重要模块，主要负责复杂规则的编写，配置和上线。其中规则管理平台主要由方法库、条件库、技术规则库、业务规则库和规则测试库组成。技术规则库主要用来编写一些用户日常的行为规则，例如某一用户一天累积交易额（单位：分）；条件库主要是对计算规则库里面的一些中间变量设置限额和简单的过滤条件，例如某一用户一天累积交易额限定不大于 1000（单位：元），其中 1000 就是限额；业务规则库中的规则主要由条件库中的条件进行组合，验证后上线使用，例如某一用户一天累积交易额限定不大于 1000 并且用户的交易时间在敏感时间段（凌晨 6 点到凌晨 7 点），这是由两个条件组成的业务规则；规则测试库里面的规则都是设计好的规则，主要针对这些规则进行验证。本文接下来将会针对规则自动化验证以及应用展开讨论[4]。

金融风险规则自动化验证分为两大块，第一大块是针对规则平台本身的业务规则验证，也就是规则上线之前验证其正确性，例如以用户和商户为维度，可以定制一些日交易量和日交易额的规则，通过规则测试界面输入这些用户或商户的相关参数，当这些用户和商户的交易数据条件满足规则制定的参数时，风险就会触发。当然也可以进行规则组合，将一些简单的规则组合成复杂的多条件规则进行测试。规则自动化验证第二大块是针对规则触发的风险数据进行回归验证，也就是规则上线之后通过反查数据的方式进行风险规则的有效性评估，评估的内容主要包括规则的误报率、漏报率、覆盖率等等。

1.2 金融风险规则自动化验证的国内外现状

金融实时风险控制在我国处于起步阶段，金融实时风险控制想要快速长远发展，风险规则的准确性和可用性是一个重要保障，金融风险规则自动化验证能确保风险规则有效快速的投入使用。目前金融风险规则自动化验证的应用很少，大多数人认为金融风控系统产生的风险都来至于计算机处理的结果，排除业务逻辑出错外，计算机出错的概率几乎为 0 。也就是说经过针对规则平台本身的规则验证已经可以确保规则的准确性和有效性。其实，在风险数据产生后，触发的风险是有误差的，例如某一用户一天累积交易额限定不大于 1000 这一条业务规则理想情况下触发条数为 158 条，实际上触发了 156 条，理想与实际相差 2 条，这些

相差的数量很难被检测到。

1.3 问题的提出

基于金融实时风控功能模块的特点，规则平台对规则的自动化验证必须有效快速的执行。时间因素是一个巨大的考验，在规则投入使用之前对规则进行有效的验证，我们摒弃了复杂传统的数据库查询，利用数据装载插件将业务流水库转化成标准数据入消息队列，通过缓存处理模块计算这些标准数据，最后将历史统计值保存到 redis 缓存数据库。规则平台结合缓存数据和需要测试的流水数据进行规则自动化验证，这样大大降低了规则自动化验证的时间。在规则投入使用之后，针对投入使用的规则产生的风险数据，一个个验证准确性会大大提高工作成本，例如产生的风险数据达到几千万的时候，一个个手工验证将是一个不可预估的成本，我们通过风险唯一序列号建立数据表之间的对应关系，通过这些表的合并来反查数据库，比较此条风险的准确性，以此来定量评估风控规则的覆盖率，漏报率和误报率。

1.4 全文组织结构

论文主要分为以下七个部分， 具体说明如下：

第一章 绪论。本章针对金融实时风控规则自动化验证缺乏的现状，介绍了金融实时风控规则自动化验证技术研究与应用的背景。

第二章 规则验证涉及的相关技术。本章主要介绍了规则自动化验证用到的一些关键技术。

第三章 规则自动化验证与应用整体架构设计。本章主要讲述了规则自动化验证的整体流程和整个部署结构，涉及到数据转载插件、消息中间件和缓存处理模块，结合已有的金融风控平台，将设计好的规则进行自动化验证与应用。

第四章 规则自动化验证技术的研究。主要介绍了数据转载插件的设计、消息队列的设计以及缓存处理模块的设计。数据装载插件可以将业务流水数据转化成标准数据，供缓存处理模块计算得到历史统计值保存到内存数据库。消息队列主要负责各模块间的信息传递。

第五章 风控规则自动化验证的应用。主要介绍了规则自动化验证两大模块的应用。第一块是在风控规则还未投入使用之前就可以进行风控规则自动化验证，定量地评估风控规则的正确性。自动化验证的第二块是规则回归验证的应用，主要对已触发的规则进行回归测试，该模块通过已产生的风险数据反查数据库，查

看历史交易库里面的交易流水是否满足此条风险规则触发的要求和条件，定量评估风控规则的覆盖率，漏报率和误报率。

第六章 风控规则自动化验证应用效果展示。主要介绍了规则自动化验证的一些效果图。

第七章 总结与展望。本文展示了风控规则自动化验证这两大块的研究成果，总结了这两大块风控规则验证的特点，以及各自需要改进的一些地方。

1.5 本章小结

本章结合金融风控的特点，对其中风险规则管理平台和业务规则自动化验证进行了阐述。结合风险规则平台，探讨了如何验证金融风控规则的准确率，覆盖率，漏报率和误报率，这些直接影响着第三方支付公司业务的有效运作。后面又阐述了金融风控规则自动化验证的两大模块，第一大块负责将验证好的规则上线使用，第二大块负责将投入使用的规则进行回归验证，统计风险规则的覆盖率，漏报率和误报率。总的来说，就是在这些规则投入使用之前，必须有效的验证规则的正确率，以保证规则的可用性；规则触发之后，必须有效的验证规则的覆盖率，漏报率和误报率，以保证规则的完整性。

第2章 规则验证涉及的相关技术

2.1 java 线程并发类

金融规则自动化验证需要抓取大量的交易流水，这些流水需要通过 java 线程去定时抓取，接着进行数据标准化，过滤掉流水中不影响规则验证的字段；根据三方支付公司每天的交易情况来看，交易流水少则几十个字段，多则上百个字段，如果不对交易流水这个大对象进行处理，将会影响规则的验证。因为一个大对象非常消耗系统资源。本次流水数据装载采用了 java.util.concurrent 并发包里面的 ThreadPoolExecutor 和 Executors 类来创建线程池[5]。以 Callable 和 Future 来进行定时任务的运行和获取，每一个任务都需要将流水转化成标准数据对象，例如针对一批流水，可能会分成多个 Callable 去执行任务，当 Future 获取到所有的执行任务结果时，代表这一批交易流水转化结束。针对性能好的机器，Java 线程并发类能有效的利用机器本身的性能优势，将抓取交易流水的速度提上来[6]。例如，在本次银联商务进行规则验证的过程中，以交易时间为增量抓取交易流水数量的 tps（每秒抓取的交易流水数）趋势如图 2.1：

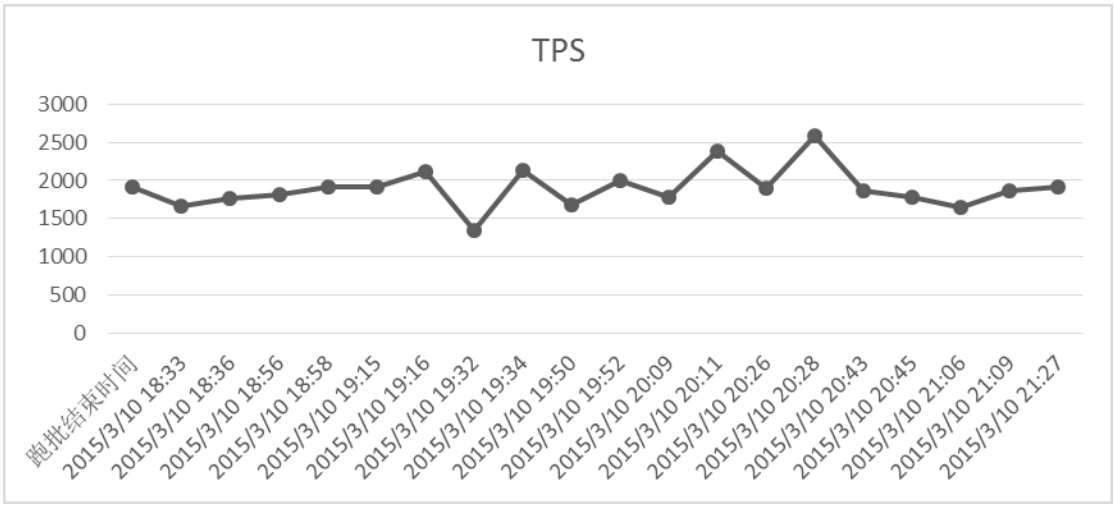

图 2. 1 java 并发类抓取交易流水数量趋势图

2.2 key-value 存储系统

金融规则自动化验证需要大量的历史流水统计数据，这些历史流水统计数据需要提前计算出来存储到机器内存当中，这样规则验证才能正常的进行。由于这些历史流水统计值的变量名都是唯一的，key-value 存储系统非常适合通过主键进

行查询，但不适合复杂的条件进行查询，而规则验证里面很少涉及复杂的条件查询，通过 key-value 存储系统存储历史流水统计值，解决了规则验证读取历史统计值的问题。一个好的 key-value 存储系统具备查询速度快、存放数据量大和支持高并发等特点。规则验证的过程当中会频繁的对历史流水统计数据进行查询，需要将这些数据以 key-value 的形式存储在机器的内存当中，这样大大减少规则验证的时间，间接的增快了规则上线的速度。本文主要使用了 redis 存储系统，它支持存储的 value 类型相对较多，包括字符串（string）、链表（list）、集合（set）、有序集合（sorted set）和哈希类型（hash）。毫无疑问，这些数据类型都支持插入、删除、查询、求交集、求并集、求差集以及其他更丰富的操作，而且这些操作都是原子性的。redis 是一个高性能的 key-value 数据库。 redis 不仅拥有 memcached 这类 key/value 存储的特点，而且支持复杂的结构模型存储。在本次规则自动化验证当中，使用 redis 进行历史值的存储使得复杂规则验证成为可能，大大减少了验证过程当中读取数据所花去的时间，保证了规则自动化验证的高效性[7]。

2.3 消息队列

金融规则自动化验证的过程中，各个模块之间的信息传递都是通过消息队列来进行的。以前大多数都是通过查询数据库来进行数据的传递，一个 SQL 语句里面嵌套几个子查询，大大的增加了我们的等待时间，导致我们的工作效率极其低下。采用消息队列可以不用考虑数据库这一层，一个模块可以直接作为生产者将转化的数据发送到消息队列，另一模块检测到这一消息队列有消息时，会立即进行消费处理[8]。本次规则验证采用了 IBM MQ 消息中间件，IBM MQ 英文全称是 IBM Message Queue，它是 IBM 一款商业化的中间件产品，适用于 WINDOWS 操作系统和 LINUX 操作系统[9]。消息队列提供了一个信息中转站，各模块应用都可以对其进行生产或消费。消息队列中的消息可以位于机器的内存或磁盘上,队列存储这些消息直到消息被对应的监听程序消费掉。消息队列的运行状态不影响监听程序的独立运行。本次采用了 IBM MQ 消息中间件，不同于 hornetq 和 activeMQ 等一些开源消息中间件的结构，IBM MQ 主要分为队列管理器、队列和通道三个大模块，一个队列管理器可以管理一个以及多个队列，建立好队列，就可以进行消息的发送与接受。IBM MQ 的使用使得规则自动化验证更加简单，它避免了模块和数据库之间的数据读取，降低了模块之间的耦合度，大大减少了数据流动的时间，使得规则验证得以高效地运行[10]。

2.4 Spring-Boot 的使用

Spring-boot 能快速搭建一个应用项目，配置灵活方便，其设计的目的主要是用来简化新 Spring 应用的开发过程和初始搭建。该框架使用了非常独特的方式来进行配置，从而使技术开发人员不再需要定义模板化的配置。Spring-boot 使得技术开发人员不仅不再需要编写 XML，而且在一些场景中甚至不需要编写繁琐的 java 代码[11]。针对一些 Web 项目，Spring-boot 还可以内置一些 Web 服务器，例如 tomcat。

本次规则自动化验证利用了 Spring-boot 灵活的加载方式，搭建了规则验证过程中的数据装载模块和缓存处理模块，使得管理这些模块非常方便。针对不同的规则验证需求，数据装载模块和缓存处理模块修改好配置参数后即可投入使用。

2.5 本章小结

本章主要介绍了在规则自动化验证中涉及的一些相关技术，主要包括 java 线程并发类、key-value 存储系统、消息队列以及 Spring-boot，这些技术的运用使得规则自动化验证能有效快速地进行。

第3章 规则自动化验证与应用整体架构设计

3.1 规则自动化验证与应用整体流程图

规则自动化验证主要针对设计好的风险规则，取出一定量的交易流水数据和缓存数据进行规则验证。其中缓存数据来源于缓存处理模块，此模块主要将规则中用到的一些中间值通过内存数据库进行存储。在已有的规则平台下对这些流水数据和缓存数据进行验证，验证设计好的规则是否有效，有效的规则即可投入使用。规则自动化验证与应用的整体流程图 3.1 如下：

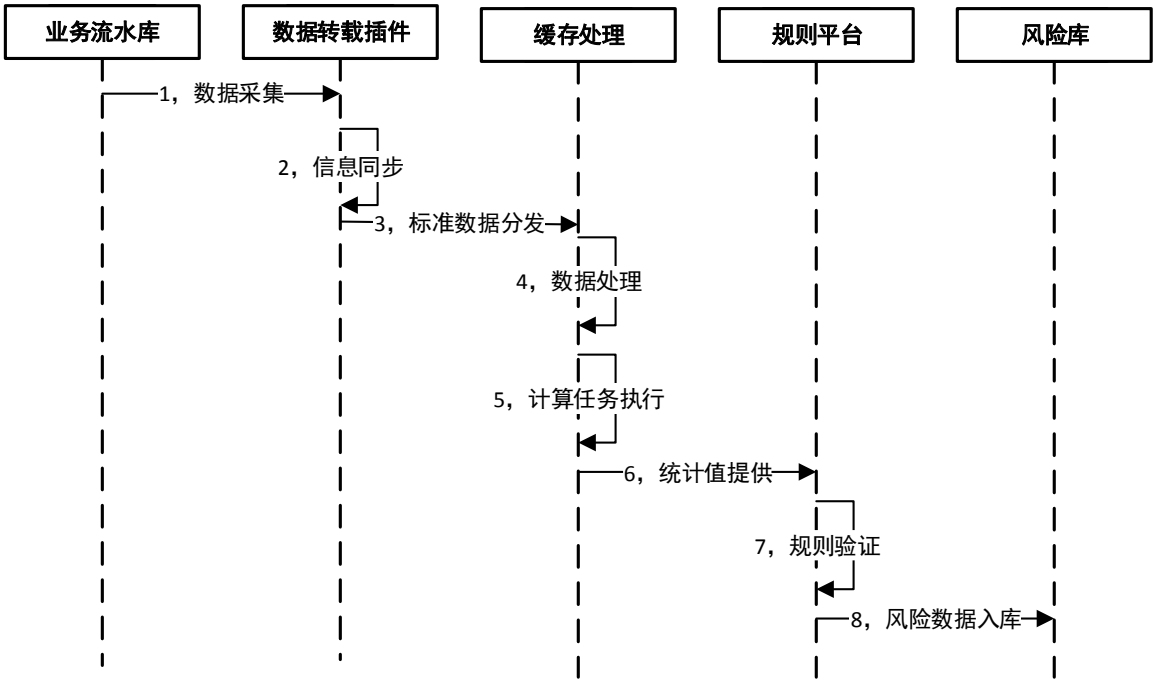

图 3. 1 规则自动化验证与应用的流程图

图 3.1 说明了规则自动化验证与应用的流程图，主要步骤如下：

1.数据采集：三方支付系统的交易流水会以时间的推移不断增多，要获取实时的流水数据，必须不定时的进行数据采集。

2.信息同步：交易流水的字段有时会上百个，而针对规则自动化验证的字段只有很少的几个,信息同步主要是对流水中的关键字段同步到标准数据表中相对应的字段。

3.标准数据分发：数据装载模块得到标准数据后会及时的发送给缓存处理模块，发送的同时，也会入标准数据库进行历史记录存储。

4.数据处理：缓存处理模块根据规则的一些过滤条件对标准数据进行统计处理，得到规则验证需要的统计值。

5.计算任务执行：执行缓存处理的每一个线程，将得到的统计值存入内存数据库。

6.统计值提供：缓存处理模块处理完标准数据后，规则需要的统计值都会存储在机器的内存中。规则平台随时可以获取得到内存统计值。

7.规则验证：规则平台针对设计好的规则结合模拟流水和历史统计值进行规则验证，以此来判断规则的正确性。

8.风险数据入库：规则验证正确后，会将这一条规则上线投入使用，使用后产生的风险将会存储到风险库当中，方便后面规则的回归验证。

缓存处理模块的运行离不开数据装载插件，首先通过数据装载插件获取业务流水库数据，对这些数据进行采集，转化成规则验证需要的标准数据；其次通过缓存处理模块对这些标准数据进行处理，将得到的一些统计值存储到内存数据库；最后，结合这些历史统计值，在规则平台上就可以验证这些流水是否可以触发设计好的规则，进而验证这些规则的正确性，对于准确率高的规则即可投入使用。规则投入使用后会根据风险库和标准库进行规则的回归验证，统计出规则的误报率，以此来说明规则的正确性是可靠的。

3.1.1 各模块间的通信

规则自动化验证与应用是结合已有的规则平台进行规则的自动化验证，其中除了调用了已有的规则平台之外，结合规则自动化验证的特点，几大模块之间需要大量的数据通信，例如数据装载插件和缓存处理模块之间需要数据传递才能正常的进行规则自动化验证。本次通过 IBM MQ 消息中间件将这几大模块联合在一起，即将业务流水库和数据转载插件通过消息中间件的某一个队列来连接，业务流水库负责生产，数据装载插件负责消费；同理，数据转载插件和缓存处理也是由一个队列进行连接。规则平台只需结合设计好的规则，通过测试的流水和缓存数据即可以进行规则的自动化验证，具体通信过程如图 3.2：

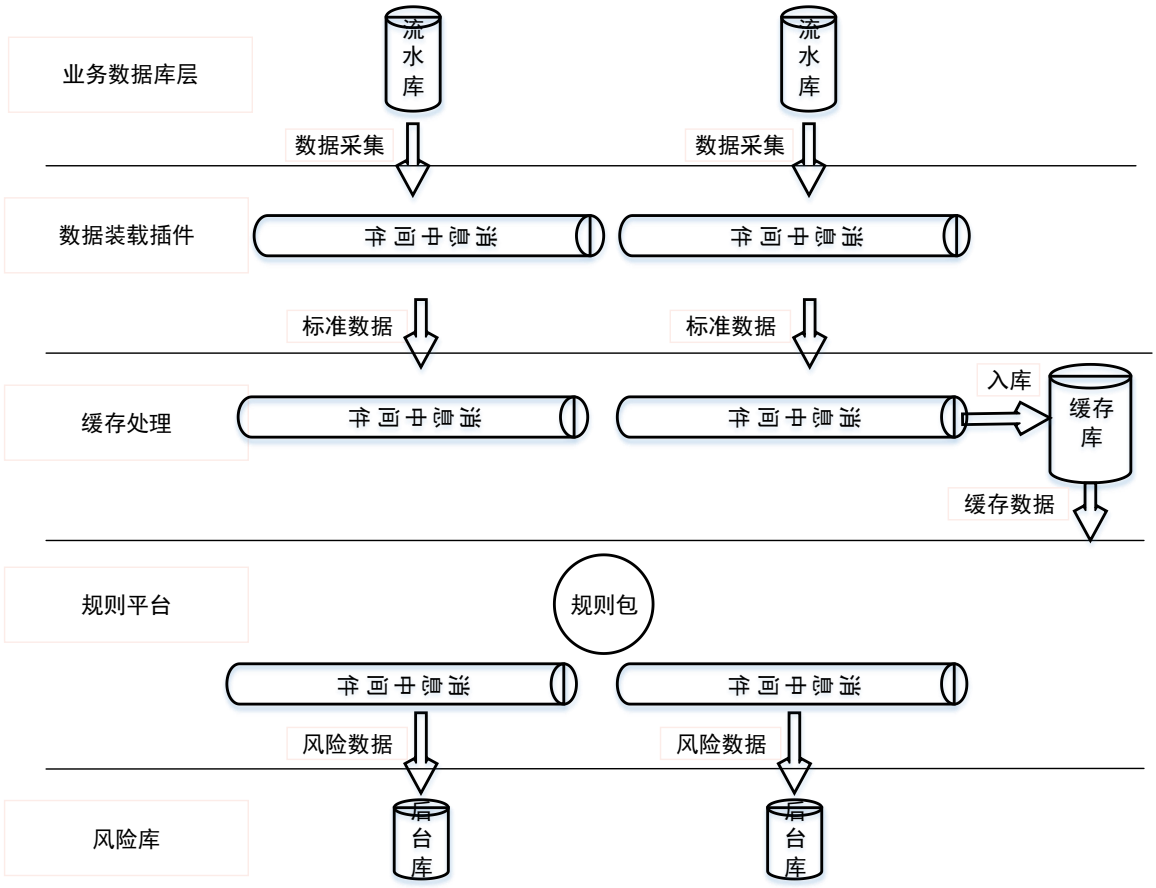

图 3. 2 规则自动化验证通信图

如上图所述，规则自动化验证与应用用到的数据除了从业务数据库采集以外，其他转化的数据几乎没有从数据库读取，直接采用消息中间件队列的方式进行生产和消费，大大节省了规则自动化验证的时间，使得这些设计好的规则能快速的投入使用，每个模块通过生产消费的模型进行点对点的通信。

图 3.2 规则自动化验证通信图说明如下：

a)数据装载插件和业务流水：业务数据库层主要提供交易流水，这一部分流水有些是发送到流水队列,数据装载插件检测到对应的流水队列中有流水数据的时候就会进行消费。

b)数据装载插件和缓存处理模块：数据装载插件产生的标准数据一份会发送到标准数据队列，同样一份会记录到标准数据入库队列中，缓存处理模块检测到对应的标准数据队列有消息就会进行处理。入库程序检测到标准数据入库队列有消息就会进行消费。

c)风险库：针对上线的规则触发的风险会发送到对应的风险队列，风险入库程序检测到有风险就会进行消费。

3.1.2 规则自动化验证与应用整体架构部署

规则平台采用自动化验证与应用需要熟悉整个架构的部署，业务库中的交易流水会不定时的存储到流水库或是消息队列，为了获取这些实时的交易数据，数据装载插件会以半增量模式监控流水库或消息队列中新来的数据，一般以交易流水的交易时间作为增量字段（也可以用交流流水的更新时间作为增量字段）。数据装载插件将这一批流水转化成标准数据后入标准数据队列，一部分提供给缓存处理模块，一部分数据用来入标准数据库。缓存处理模块会根据一些规则中需要的参数过滤这些标准数据，得到中间变量值存储到 redis 当中；在准备好的模拟流水情况下，此时规则验证只需要获取 redis 当中的统计值就可以进行规则的验证，当模拟的流水都能验证某一规则的正确性，这样就可以针对设计好的规则进行正确性验证，对正确性好的规则进行上线应用。规则的上线应用由于不属于本文范畴，下面简单叙述：缓存处理插件得到中间变量值存储到 redis 的同时，会封装这些标准数据为探头数据入队列，一部分用来调金融风控系统，一部分用来入标准库。金融风控系统会根据规则平台的规则过滤探头数据，结合 redis 缓存数据库的中间变量值，产生风险数据入风险队列，风险入库程序会对风险队列进行消费入到风险库。以上是规则上线以后的流程，产生风险后，结合风险库和标准库要进行规则的回归验证，也就是对规则进行逆向验证，保证其正确性是可靠的。

针对规则验证需要数据装载插件、消息队列和缓存处理模块这三个模块，除了这三个模块以外，还需要有 oracle 辅助数据库，结合规则验证的特点，考虑到 redis 存储的时候机器内存不足，以及数据库服务器宕机的情况下，整体部署图如图 3.3：

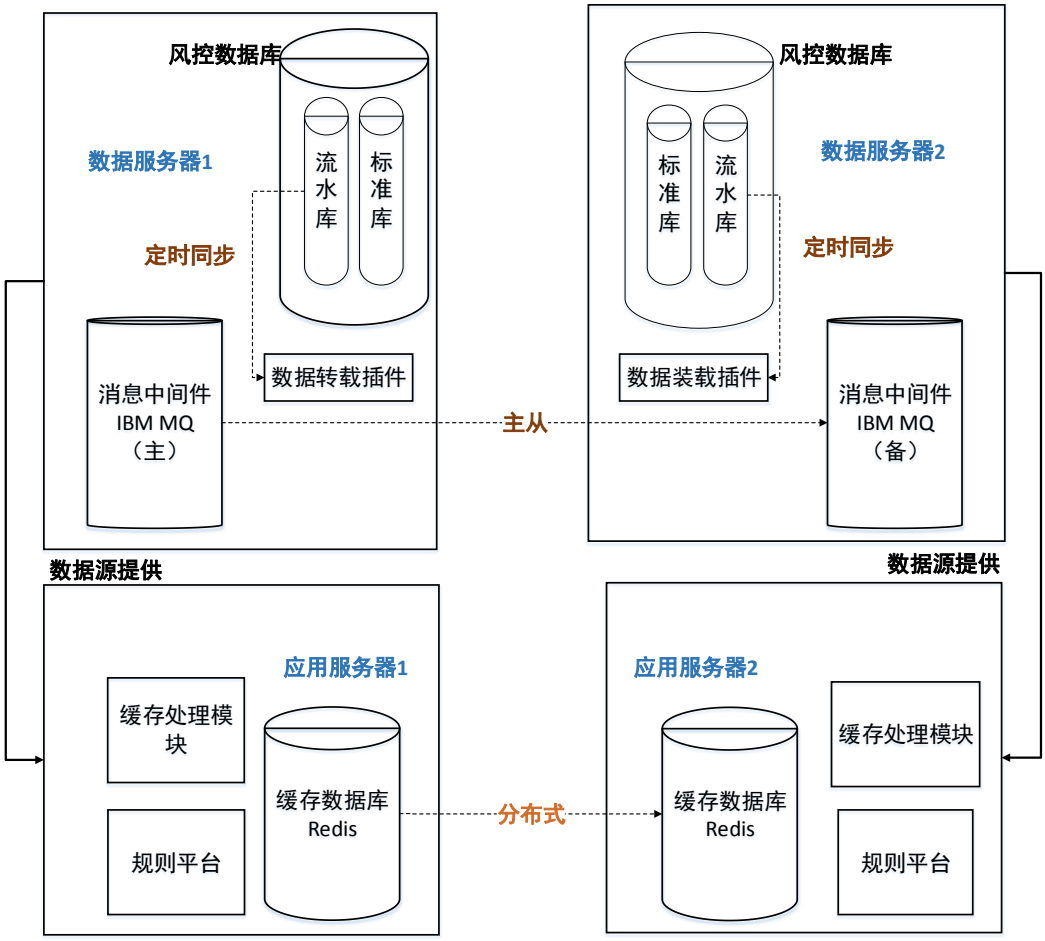

图 3. 3 规则自动化验证架构图

数据服务器 1 和数据服务器 2 互为热备，应用服务器 1 和应用服务器 2 互为热备[12]。具体的数据库热备方案、应用程序的热备和消息中间件主从配置是由同组的其他人完成[13]。本人负责数据装载插件的编写、缓存处理模块的编写、redis 分布式的配置、规则平台的规则编写以及规则的验证。在数据服务器上，数据库服务器采用了 oracle 数据库，除了部署 oracle 外，也部署了数据装载插件和消息中间件；在应用服务器上，部署了规则平台、缓存处理模块和 redis。针对以上部署说明：

a)数据服务器上，将数据转载插件和数据库绑在一块，能够直接快速的获取业务流水并进行数据转化。将消息中间件和数据装载插件绑一起，是因为有些业务流水是直接发送到消息队列的，而没有经过业务流水库；在这种情况下，数据装载插件也能很快的进行数据装载。消息队列最主要的作用还是进行模块间的通信，由于消息队列中的数据发送给各个模块处理的同时会入库保存历史记录，所以和数据库会平凡打交道，因此将其部署到数据库服务器会大大减少网络的带宽带来的延时。

b)应用服务器上，由于 redis 的数据存储需要消耗大量的内存空间，应用服务器的内存相比较数据服务器而言，拥有较大的内存空间，所以将 redis 部署在应用服务器中，由于缓存处理模块会经常的操作 redis 服务器，也必须要将缓存处理模块和 redis 绑在一台机器上，这样可以大大的减少网络带宽带来的延时。规则平台是一个可视化的操作界面，规则平台中规则验证用到的缓存数据比较多，操作也平凡，两者部署到一起可以拥有较高的带宽，增加系统的测试速度，因此将其部署在应用服务器上[14]。

3.2 规则自动化验证与应用两大模块

规则自动化验证技术分为两大块，第一块是规则投入使用之前对规则进行正确性验证，这需要用到规则自动化验证中的规则管理平台和 redis，结合规则管理平台配置好的业务规则进行规则的自动化验证；第二块是规则投入使用之后对规则进行回归验证，主要针对标准数据库的流水化查询。

3.2.1 规则自动化验证（规则投入使用之前）

在整个规则自动化验证中，首先要进行规则的自动化验证，在验证之前，需要保证 redis 里面必须存有规则所需要的统计值，否则规则验证将得不到运行结果。其次，如果 redis 里面的统计值不是实时的，那么规则验证将得不到正确的结果。redis 里面的值来至于缓存处理模块的结果，缓存处理模块提前会根据规则的过滤逻辑对一个相应时间段的标准数据进行统计处理。为了达到规则自动化验证的效果，缓存处理模块必须实时的进行统计值的合并、拆分等操作，以此来保证 redis 中的数据值是最新的。验证规则除了注意 redis 中的数据外，还需要保证模拟流水的数据和最新的交易数据一致。为了保证缓存处理模块的实时，就必须得保证数据装载插件的实时，数据装载插件不定时的抓取最新流水，进行标准转化发送到标准队列，缓存处理模块检测到标准队列有数据会立即进行消费处理。如果不考虑消息队列的生产消费时间的话，缓存处理模块和数据装载插件应该在同时处理数据，可以保证 redis 中的数据是最新的。规则验证完之后，可以对于正确性高的规则投入应用，由于规则的上线应用不属于本次论文研究的范畴，具体上线操作在此就不详述了。规则上线后，交易流水会过风控系统，风控系统会将流水转化成为探头数据，逐一的比较上线的风险规则，看其是否触发风险，针对触发风险的规则，我们会对其做回归验证。

3.2.2 规则回归验证（规则投入使用之后）

在整个规则自动化验证中，针对上线规则触发的风险，我们会进行规则的回归验证，根据投入使用的规则逻辑进行反查，在反查之前，必须要弄清楚交易流水、

规则和风险的对应关系，一条流水要么就触发一条风险，要么就不触发风险，而一条风险有时候会包含一条规则，但在很多时候一条风险会包含多条规则，如果风险信息存储在一张表（通常交易流水和风险存储在一起，一一对应），规则信息存储在另一张表，那么前后两张表就是一对多的关系。其次，在规则回归验证之前，必须要保证标准数据入库程序的正常运行，也就是要保证标准数据库中记录了所有的交易流水，如果入库这一块程序出错，连接这一块程序的消息队列也会报错，很有可能的就是消息队列中的数据将不会被消费，不仅很多交易流水将会丢失，而且队列会阻塞，影响整体架构的正常运行。除了以上这些，还需要编写规则回归测试的验证程序，这些程序主要是和标准数据库和风险库打交道，从风险库里面获取风险信息、对应的规则信息以及触发这条风险的交易流水，统计出所有符合这一条规则的所有风险，即获取触发这条规则的所有流水，提前把需要回归验证的规则逻辑编写好后，并将这些流水逐一测试，就可以统计规则的漏报率和覆盖率，进行比较验证，以此来监控规则正确的可靠性。

3.3 本章小结

本章主要介绍了整个规则自动化验证的具体流程、各个模块之间的通信方式、整个规则自动化验证和应用的部署框架以及规则自动化验证与应用的两大模块。针对设计好的规则，在投入使用之前要进行验证，保证规则的准确率；投入使用之后，也要对其进行回归性测试，以此来保证规则正确的可靠性。

第4章 规则自动化验证技术的研究

为了实现第三章规则自动化验证与应用的整体架构，还需要针对数据装载插件、消息队列以及缓存处理模块进行具体的研究设计。以下小节主要说明了数据装载插件如何去抓取数据、消息队列如何进行模块间的数据传递以及缓存处理模块如何进行标准数据的处理。

4.1 数据转载插件的设计

规则自动化验证需要当前模拟交易的相关历史统计值，而这些历史统计值需要提前对交易流水进行处理，例如验证同一用户过去 5 天内的平均交易金额大于等于 50000 元这一规则，假设模拟交易流水的用户号为 0908070038、交易金额为 1000 元，那么数据装载插件必须提前加载用户号为 0908070038 前 5 天的交易流水，将其前 5 天的平均交易金额统计出来。数据转载插件做的任务就是预先加载交易流水，为后面的规则验证提供数据准备。数据转载插件会以定时任务的增量方式不断的加载数据库或消息队列中新出现的流水，当有新流水过来的时候，数据装载插件就会工作，将得到的流水进行标准转化，将得到的标准数据入消息队列，为缓存处理模块提供数据源。每次规则验证都会以一批真实可靠的交易流水数据作为测试案例，其中包含静态数据（一些用户、商户，卡号等等静态信息）和动态数据（每天交易的真实流水数据），静态信息更新频率低，可以选择一天更新一次，但交易流水会不定时的产生。针对这两类数据，数据装载插件的定时方式必须设置好。而且静态数据需要和动态数据在日期上面对应，这样才能保证交易流水里面关联到的静态信息是一一对应的，不影响后面规则自动化验证的结果。

4.1.1 静态数据插件设计

静态数据主要是指一些不经常变化的个人信息（用户号、商户号、用户名称、商户名称、手机号码、联系地址、注册日期、更新时间等等）和一些结算等级信息（VIP 商户 1 天一结算、高级商户 3 天一结算、普通商户 5 天已结算等等），例如杭州市黄龙体育中心沃尔玛这一大商户，一般不出什么大的情况，很少有负责人会去修改它的静态信息，有的静态信息可能好几年都没有变化，对于这些数据，数据装载插件加载完一次后，完全可以不用再次加载。这样的话可以设置一个定时任务去增量查询这些静态数据，以这些静态表的更新时间为增量字段，可以设置静态数据的定时任务在固定的时间段内执行，例如每天晚上的十点钟开始装载，第二天凌晨 2 点装载结束。一共装载 4 个小时，装载完为止。每次装载从静态数据的最小更新

时间开始，依次增量一天，从数据库中查询出更新时间间隔为一天的静态数据，这些数据量也不会太多，因为以一天为间隔，修改这些静态信息的记录往往比较少。与流水动态数据装载不一样，动态数据装载只会加载最新的交易流水，不会从头开始，而静态数据每一次装载开始之前，都会全量的扫描表中所有的静态数据，例如用户信息表一共 4568792 条数据，其中更新时间最小为 2007 年 3 月 12 号，最大更新时间为 2015 年 4 月 12 号,那么静态数据每天晚上 10 点到第二天凌晨要把这更新时间 2007 年 3 月 12 号到最新更新时间全部加载一遍，加载的过程当中是以一天为步长往前递推的，只到加载到最新时间，由于只运行 4 个小时，静态数据装载线程也不会一直运行，到第二天凌晨 2 点自动关掉。那么第二天关联查询就可以得到最新的静态数据。以下是静态数据每天晚上 10 点（22:00）到第二天凌晨 2 点（2:00）开始装载的具体的设计流程，如下图 4.1：

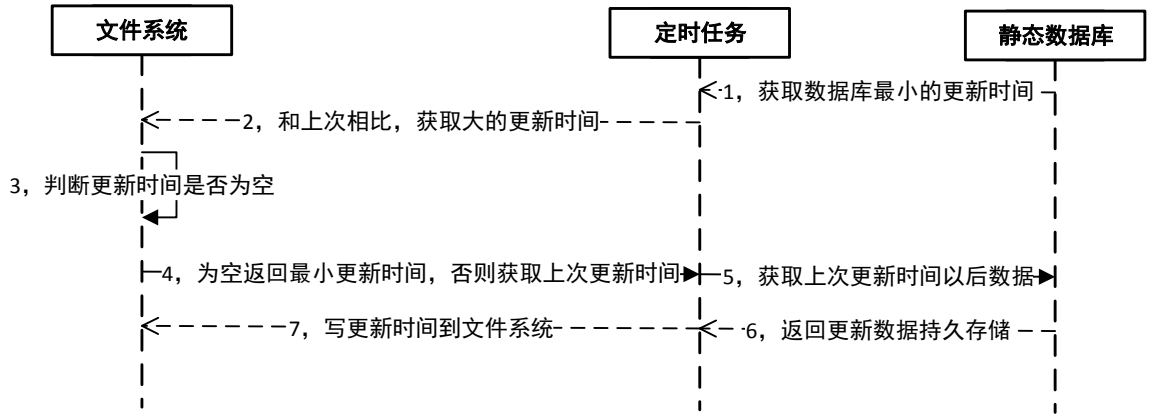

图 4. 1 静态数据插件装载流程图

a)文件系统：主要提供静态数据装载任务中更新时间字段的存储。采用输入输出流将更新时间以 LONG 类型存储在某一固定名的文件当中，文件内容以 map<key:value> 类型存储，主要由开始时间（startTime）和结束时间（endTime）两个主键组成，格式类似于{ startTime：‘2007 年 3 月 12’，endTime：‘2007 年 3 月 13’}，静态数据装载任务每天 10 点开始装载后，首先会新建这个文件名，startTime 和 endTime 初始值为空。第一次装载 startTime 为静态数据的最小更新时间，endTime 为最小更新时间加上一天。通过（startTime，endTime）时间段从数据库里面加载静态数据。后续每一次加载 startTime 和 endTime 都会向前加一天,当 endTime 的值大于静态数据中最大更新时间时，（startTime，endTime）时间段内将没有数据，只到第二天凌晨 2 点任务结束。

b)定时任务：主要通过读取文件系统中的更新时间来装载静态数据，抓取静态数据。一般定时任务启动后，会以一秒的频率去读取一次文件系统，获取新的时间段，查询完这一时间段的数据后又将新的更新时间写到文件系统中，供下一个任务使用。例如任务一从文件系统里面获取更新时间 startTime，抓取（startTime，endTime）的数据后，又将 endTime 赋值给 startTime 供下一个任务调用。

c)静态数据库：静态数据库管理了一些经常不变化的数据。静态定时任务会获取这些静态数据，进行对应字段的转化。将需要显示的风险数据字段存储到标准数据库对应的表中。例如流水产生的风险需要显示商户的名称，而交易流水里面没有这个字段，只有商户号，那么就必须通过商户号关联商户静态表获取商户名称。

综上所述，首先静态数据定时任务获取静态数据库中静态数据表的最小更新时间，与上次存储在文件系统的更新时间比较，如果小于上次系统存储的更新时间（每一次定时任务结束之前，都会将上次查询完的最大时间写到文件系统），则取上次的更新时间，否则取最小更新时间进行全部装载（初始化的时候用到）。除了第一次初始化之外，每一次的更新时间都是上次系统存储过的更新时间，当更新时间超过最大更新时间，也就是拿上次的更新时间到当前时间（不超过系统当前时间）这个时间段去查询静态表，看是否有新的静态数据变化（新增和修改），将得到的这些变化数据进行持久化存储，供动态数据间接查询和使用。

4.1.2 动态数据插件设计

动态数据主要是指三方支付系统每天过来的交易流水，这些流水复杂多变，来至于不同的客户端、不同的地点以及不同的时间段。动态数据装载插件主要负责收集这些不断变化的交易流水。和静态数据不同，这些交易流水随机性很大，短时间内可能交易量过万。在特殊时段，交易数据都是并发产生，不可能以一天为步长去加载这些数据，这样会出现数据拥堵，因为有些大的三方支付公司一天的流水量完全可以超过上百万[15]。对于一台普通机器，且不说会造成数据查询时间慢、影响整体的性能，一次性加载上百万数据完全可以让一个系统死机。不同于静态数据的装载方式，动态数据装载选取流水的交易时间作为增量字段，每次增量的步长为 5 分钟，不会设置晚上 10 点才开始执行定时任务，而是每时每刻都去执行这些装载任务。第一次运行动态数据装载插件是全量加载，加载到最新交易时间后会进行增量加载。例如 2015 年 3 月 12 号开始加载交易流水，每隔 5 分钟去装载一次流水数据，这样才能保证每一次抓到的数据大小适当，当数据装载当交易时间和系统当前时间一样

的时候，每次的步长将会和系统时间同步，例如交易流水表从 2015 年 3 月 12 开始装载流水，每次数据装载的步长以 5 分钟的递增，此时属于全增量方式加载交易流水，当装载的时间等于系统当前时间，那么每次动态数据装载的步长将变为一秒，此时属于半增量方式加载流水。针对全增量加载，具体的设计流程如图 4.2：

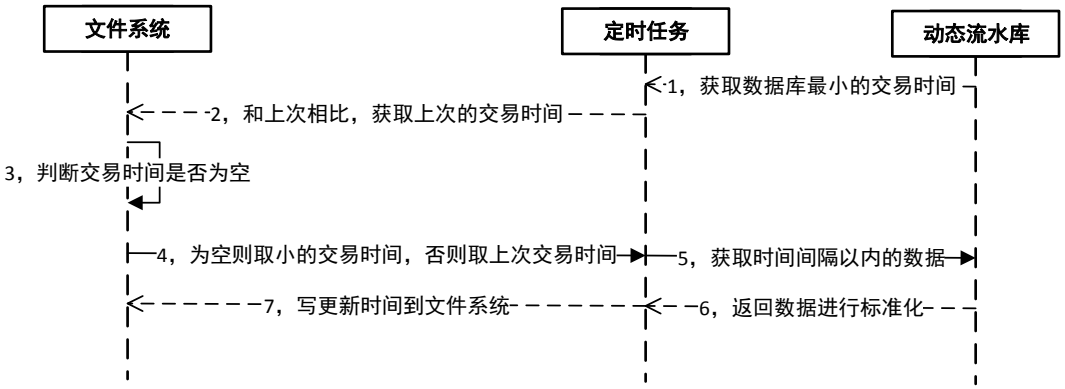

图 4. 2 动态数据插件装载流程图（全增量）

a)文件系统：主要提供动态数据装载任务中交易时间字段的存储。采用输入输出流的方式将交易时间以 LONG 类型存储在某一固定名的文件当中，文件内容以 map<key:value>类型存储，主要由开始时间（startTime）和结束时间（endTime）两个主键组成，格式类似于{ startTime：‘2007-3-12 12:34:23’，endTime：‘2007-3-12 12:34:28’}，动态数据装载任务执行后，首先会新建一个文件名存储 startTime 和 endTime 这两个主键，初始值为空。第一次装载 startTime 为流水的最小交易时间，endTime 为最小交易时间加上 5 分钟。通过（startTime，endTime）时间段从流水库里面加载交易数据。后续每一次加载 startTime 和 endTime 都会向前加 5 分钟。

b)定时任务：主要通过读取文件系统中的交易时间来装载交易流水数据。一般定时任务启动后，会以一秒的频率去读取一次文件系统，获取新的时间段，查询完这一时间段的数据后又将新的更新时间写到文件系统中,供下一个任务使用。例如任务一从文件系统里面获取交易时间 startTime，抓取（startTime，endTime）的数据后，又将 endTime 赋值给 startTime 供下一个任务调用。

c)动态流水库：动态流水库无时无刻都在进行流水数据的增加。动态定时任务要及时的获取这些流水数据，在刚开始之前，会以全量的方式进行加载，因为第一次动态流水的装载不是从系统当前时间开始的,而是由规则中的时间

参数规定的,例如同一用户过去 5 天内的平均交易金额大于等于 50000 元这一规则，动态数据装载插件会提前加载好前 5 天（系统时间向前推五天）的历史流水数据。

综上所述，首先动态数据定时任务获取动态流水库中流水表的最小、最大交易时间，和上次文件系统存储的交易时间（交易时间与间隔时间之和）进行比较，如果上次的交易时间加上间隔时间小于流水表的最大交易时间，定时任务会以【上次交易时间，上次的交易时间+时间间隔】时间段去查询流水库，以一秒的定时间隔不断加载动态数据,当上次的交易时间加上间隔时间大于了流水表的最大交易时间，每一次查询流水表时间段的上限取系统当前时间，即用【上次流水时间，系统当前时间】时间段去查询流水数据库，以后动态数据的加载都会以半增量方式进行加载，针对半增量加载，具体的设计流程如图 4.3：

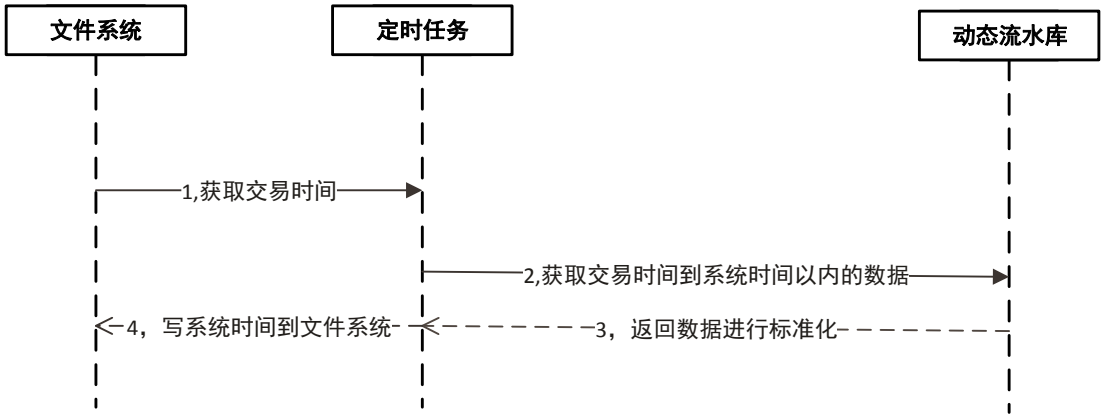

图 4. 3 动态数据插件装载流程图（半增量）

a)文件系统：主要提供动态数据装载任务中交易时间字段的存储。在半增量模式下，文件系统存储的交易时间和系统当前时间同步。

b)定时任务：主要通过读取文件系统中的交易时间来装载交易流水数据。此时在半增量的模式下，定时任务装载的交易流水数据都是实时的交易信息。

c)动态流水库：动态流水库无时无刻都在进行流水数据的增加。此时在半增量的模式下，当动态流水库有数据增加，它才能装载到数据，否则将得不到数据。

4.1.3 静态，动态数据装载架构设计

规则自动化验证除了需要大量的静态数据和动态数据之外，还必须要考虑到配置的灵活性，比如切换数据库地址、表名等等一些经常需要修改的参数。针对静态

数据和动态数据的抓取，采用了 Spring-Boot 的架构方式，这种方式能够快速的构建你的应用，可以将所有的参数配置项统一管理，代替原有的 xml 的方式，方便修改，避免参数变化导致修改代码的繁琐局面,大大节省测试的时间。具体数据装载插件如图 4.4 所示：

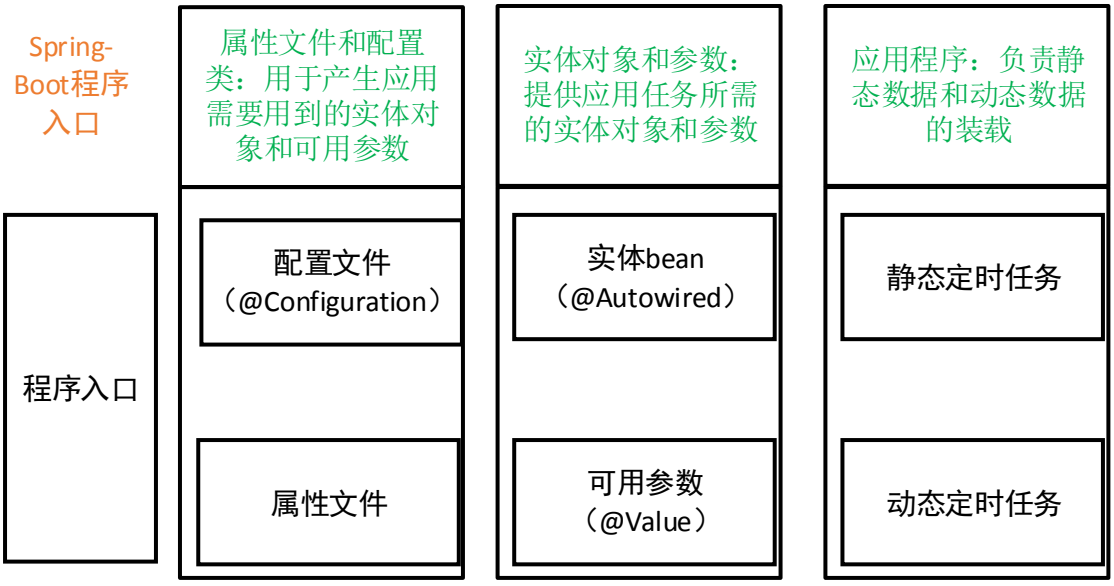

图 4. 4 静态、动态数据装载架构设计图

a)程序入口：用 Spring-Boot 搭建的应用程序都有一个入口，无论是将其打包成 jar 运行还是直接运行，应用程序首先回去加载这个入口类。

b)配置文件和属性文件：配置文件是.java 为后缀名文件，属性文件是.properties 为后缀名的文件，两者都提供了应用程序需要的实体和参数。

c)实体 bean 和可用参数：实体 bean 是已经被 Spring 容器创建的对象，可用参数也是由 Spring 管理，两者直接可以直接给应用程序使用。

d)静态定时任务和动态定时任务：静态定时任务和动态定时任务可以对实体 bean 和可用参数进行使用和操作。

综上所述，在整个规则自动化验证中，数据的来源通过 Spring-Boot 进行搭建，首先通过读取程序入口来快速进行初始化配置（配置有@EnableAutoConfiguration 和@ComponentScan 注解的）获取想要的实体 bean 和可用参数，整个应用会扫描所有的配置文件（即标记有@Configuration 的所有配置类），然后结合属性文件中配置的参数来生成应用程序所需要的实体对象，实体对象在定时任务中就可以通过@Autowired 注解来进行使用，参数在定时任务中以@Value 注解进行使用，这样很简洁的就完成了一次数据取样，当需要获取其他表和其他时间段的数据时，只需修

改属性文件就可以进行部署应用，快速获得想要测试的数据样本。

4.2 消息队列的设计

规则自动化验证除了需要数据装载插件装载静态数据和流水数据外，还需要通过消息队列进行各个模块间的数据传递，消息队列作为一个数据的中转站，进行数据的接受和发送，能很小延时的将数据送到各个模块，例如及时的将标准数据发送到缓存处理模块，这样大大减少了规则自动化验证的时间，间接的提高了规则上线的速度，这在紧急的情况下非常关键。例如某三方支付公司突然出现一些很奇怪的流水，为了阻止这些流水，针对这些奇怪的流水设计了一些规则，为了及时验证这些规则的有效性，不可能大量的去查询数据库进行验证，这时候通过消息队列可以大大减少操作数据库的时间。本文规则化验证采用的消息中间件为 IBM MQ，是一款 IBM 官方的消息中间件，当然也可以用开源的 hornetQ 等等一些开源的消息中间件，在本文就不作详述了。IBM MQ 主要提供了队列管理器，队列和通道三大块的创建，一个队列管理器可以管理多个队列以及拥有多个通道，一个队列可以分为本地队列（即发送队列）、远程队列（即接受队列）以及传输队列（即通讯队列），一个通道可以分为发送通道和接受通道两种类型。考虑到简化型，本文将本地队列、远程队列和传输队列合并成一个队列。本次规则自动化验证采用了一个队列管理器、五个队列以及一个通道。将 IBM MQ 的安装好之后，就可以通过图形界面客户端或者命令行界面来进行这些资源的创建，本次创建采用命令行方式，具体设计如下：

a)安装好 MQ 后，使用 IBM MQ 的用户登入系统，直接键入 crtmqm -q DLQ 命令创建队列管理器。crtmqm 是创建队列管理器的命令，DLQ 是队列管理器名称。

b)队列管理器创建好后，使用 strmqm DLQ 命令开启创建的队列管理器，使其生效。

c)队列管理器生效后，使用 runmqsc DLQ < define_frms.tst > out.txt 命令创建队列和通道，define_frms.tst 是当前文件夹下面的文件，里面定义了五个队列和一个通道，创建成功后，out.txt 文件里面会提示成功创建。

define_frms.tst 文件内容如下：

```
define qlocal (PREDSPAYORDERQUEUE)
define qlocal (FRMSDSPAYORDERQUEUE)
define qlocal (FRMSRISKARCHIVEQUEUE) ……
DEFINE CHANNEL ('SERVERCONN') CHLTYPE(SVRCONN)
```

define 是定义一个队列或通道的命令，qlocal 代表创建本地队列，对应括号里面代表的是队列名，本次规则验证主要用到了 PREDSPAYORDERQUEUE（连接数据装载插件和缓存处理模块）、FRMSDSPAYORDERQUEUE（连接数据装载插件和标准数据库）和 FRMSRISKARCHIVEQUEUE（连接规则平台和风险库）三个队列；CHANNEL 代表创建通道，对应括号里面代表的是通道名，CHLTYPE 代表是通道类型，对应括号里面是 SVRCONN 类型通道。

d)创建好队列和通道后，使用 runmqlsr -t tcp -p 1414 -m DLQ 命令启动 IBM MQ 的监听。

4.2.1 队列的设计

针对规则自动化验证，要求的是能快速有效的进行验证。这主要体现在数据装载插件能快速的抓取流水、缓存处理模块能及时的抓取数据装载插件过来的标准数据、标准数据及时入标准库等等。这些需要消息队列来进行消息的接受和发送。消息队列可以及时的将标准数据发送给缓存处理模块，将历史流水数据快速的统计出来进行规则的验证匹配，这些统计数据的延时远远小于操作数据库的时间，消息中间件可以短暂的存放一部分消息，但也有一定的容量，设计一个大的队列进行所有信息的存储显然是不够的，而且一个大的队列存放的数据越多，其他进程访问的速度也会有所下降。本次规则自动化验证设计了五个队列，一个队列对应两个模块，这样访问起来就比较方便。队列采用点对点的方式，采用监听方式进行访问，当有数据的时候采取存放或消费[16]。五个队列结合几大模块形成了一个流式的处理模型，具体的设计如图 4.5：

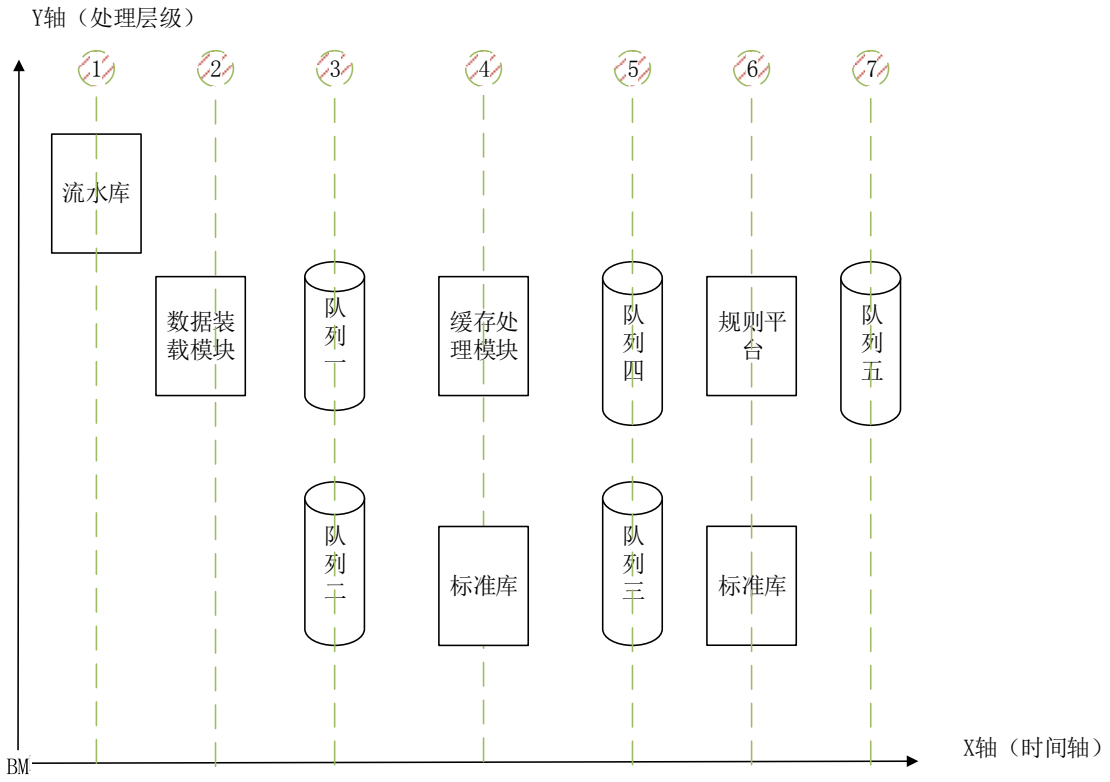

图 4. 5 队列处理时序图

a)X 轴（数据流的时间）：在规则自动化验证之前，X 轴代表了数据从流水数据库到规则平台的数据流时间，首先交易流水过数据装载插件，经数据装载插件处理后通过队列一和队列二同时分发给缓存处理模块和标准数据库；其次缓存处理模块处理标准数据后通过队列四和队列三同时分发给规则平台和标准数据库；最后规则平台通过对队列五分发给风险库。本文用到了队列一、队列二以及队列五。

b)Y 轴（处理层级）：Y 轴的最上层代表了流水库，属于三方公司的数据库；中间层代表了规则自动化验证所需的插件和应用平台；最下层代表了插件和应用平台所需的数据库。

如上图 4.5 所示，首先交易数据存入流水库，这一层主要和外来的数据打交道，流水库的作用主要用来接收外来的所有交易流水进行入库。在中间一层，数据装载插件程序会定时的去查询流水库，作为消息队列的生产者，将转化得到的标准数据发送到队列一和队列二；队列一和队列二几乎同时处理这些数据，队列一中的数据主要供缓存处理模块处理计算，队列二中的数据主要插入标准数据库，随着时间的推移，缓存处理模块统计好历史数据入 redis 的同时，又作为生产者，将转化得到的

简化流水数据发送队列四和队列三，队列四中的数据负责调用规则平台中的规则进行风险触发，队列三中的数据主要插入标准数据库进行历史记录，随着时间的推移，规则平台产生的风险发送到队列五，队列五中的风险信息最后入到风险库；本次规则化验证主要使用了队列一、队列二以及队列五，队列一负责将标准数据发送给缓存处理模块，队列二负责将标准数据入标准库，队列五负责将风险数据入风险库。在两大块规则自动化验证当中，队列一关联规则自动化验证（规则投入使用之前），队列二和队列五关联规则回归验证（规则投入使用之后）[17][18]。

4.2.2 队列的顺序处理

针对几大模块，五个队列的定义如下：

a)队列一：define qlocal (PREDSPAYORDERQUEUE)

b)队列二：define qlocal (FRMSDSPAYORDERQUEUE)

c)队列三：define qlocal (FRMSTRANSLOGQUEUE)

d)队列四：define qlocal (FRMSAUDITOBJECTQUEUE)

e)队列五：define qlocal (FRMSRISKARCHIVEQUEUE)

在消息队列的信息传送过程当中，信息的交易时间需要保持一定的顺序，这样才能保证数据装载插件和缓存处理模块能精准的统计历史流水数据，交易时间的错乱将会导致统计的历史值不准确，这样直接影响规则的验证。例如验证同一用户前一笔的刷卡金额大于 500 元这条规则，缓存处理模块会计算统计同一用户前一笔的刷卡金额这个中间值，如果交易时间不排序的话，redis 中很有可能不存在这个中间值，因为此用户前一笔交易很有可能在此笔交易的后面被处理，那么这个中间值将直接影响规则的自动化验证，影响准确率。

本次规则自动化验证主要用到了队列一、队列二和队列五。队列三和队列四主要针对规则上线使用，这里不作详述。队列一里面的信息都是按交易时间从小打到进行排序，当交易时间相等的时候，会按照流水主键进行排序，总之，队列一里面的信息会按照交易时间的先后顺序进行发送和接受。队列二是队列一的一个副本，两个队列完全一样，数据装载插件产生标准数据的时候会分发到这两个队列，队列一连接缓存处理模块，队列二连接标准数据库，也就是说标准数据库记录了所有过缓存处理模块的标准数据。队列五主要接受来至规则平台的风险，当规则自动化验证完成后，规则就会上线使用，这时候就会有风险触发，这些触发的风险会发送到消息队列五，队列五再入到风险库。最后会结合风险库和标准库进行规则回归验证。除了消息队列中的数据需要顺序外，队列与队列之间也有一定的先后，例如队列一和队列四这两个队列需要有先后关系，针对数据装载插件转载的一批数据，队列一

中的数据必须在缓存处理模块处理完后才可以将这一批数据转化后发给队列四，这样每一条数据去触发规则的时候，如果不能在内存数据库中找到想要的历史中间变量，这个规则本该触发风险但没有触发，或是本该不触发风险但触发了，将会导致规则的漏报或误报，直接影响规则的回归验证。保证队列一和队列四的数据处理顺序，规则的回归验证正确性才可靠。所有的队列都采用 Spring 中的 jms 方式进行配置，使用 JmsTemplate 进行消息的发送（send 方法）和接受（receive 方法）[19][20]。

4.3 缓存处理模块的设计

缓存处理模块主要是将数据装载插件传过来的标准数据进行计算处理，结合提供的规则参数，将得到的历史统计值存储到 redis。针对规则自动化验证，需要的不仅仅是当前这一笔交易流水,还可能涉及到这笔交易流水中一些维度的历史统计值。例如验证同一商户过去 10 日退单笔数大于 5 笔这一条规则,验证当前这笔交易是否满足规则的时候,还需要从缓存中拿出同一商户过去 10 天退单的总笔数，两者综合才能去判断规则的有效性。平时我们可能通过数据库的 sql 语句进行查询，当遇到数据量很大的时候，我们会浪费大量的时间。缓存处理模块能有效节省我们等待数据库的时间，它将我们需要的历史统计值存储到缓存，及时更新，当我们规则验证需要这些值的时候，就能很快的获取到。针对这些流水历史统计值，采用缓存数据进行存储，读取速度快，大大提高了规则自动化验证的效率。缓存处理模块最重要的是如何设计缓存的数据存储结构，即如何将历史统计值进行有效存储。以下小结针对历史统计值的存储进行说明：

4.3.1 缓存处理模块

数据装载插件将转化好的标准数据发送到消息队列，缓存处理模块会及时的进行消费，对这些数据按条件进行计算处理，最终以缓存模型进行存储。键值对是最常用的缓存模型，key-value 存储系统带有自己的模型结构，例如 memcached，支持简单的（key,value）键值对数据结构；针对规则自动化验证，本次验证选用 redis，redis 除了支持简单的（key,value）键值对数据结构外，还提供 list、set、hash 等结构的存储。本次规则自动化验证采用的是 redis 存储系统，数据结构选用的是（key1,map）结构,其中 key1 代表了这一业务、这一维度和这一维度值的唯一键；map 代表了一个（key2,value）结构，map 结构中 key2，代表了一个时间戳，value 代表了在这个时间戳下，针对 key1 这个键下面有多少和它相关的历史统计值[21][22]。缓存处理模块流程具体的设计如图 4.6：

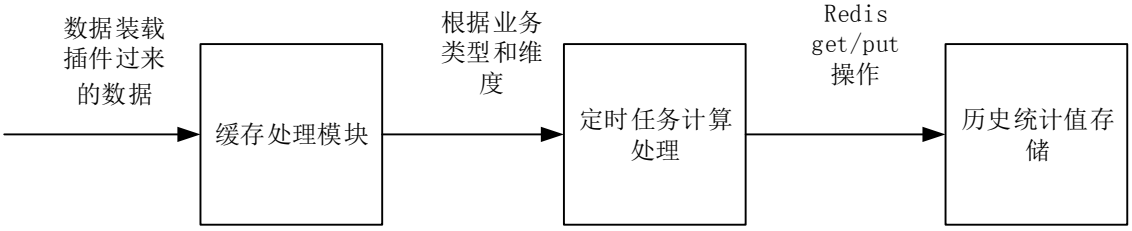

图 4.6 缓存处理模块流程图

a)缓存处理模块：主要负责接受数据装载插件过来的标准数据，将这些数据按照业务类型和维度进行过滤；例如标准数据中有手机业务、信用卡业务和终端业务，这三个业务里面都含有银行卡维度这个字段，如果给定的业务类型为信用卡业务，那么缓存会将手机和终端这两个业务过滤掉。

b)定时任务计算处理：包含了交易时间、交易状态等一些规则参数，通过某一维度进行标准数据统计，将得到的结果存到 redis 当中。例如统计信用卡业务中商户号和银行卡这两个维度的、交易失败的，时间回溯倒 5 天以前的历史数据，那么这些得到的统计值就可以用来验证很多规则，只要符合规则是同一商户或同一卡号、5 天之内并且交易失败等条件的都可以。

c)历史统计值存储：主要通过 redis 来进行历史统计值的存储。当 redis 里面还有相同的 key，新 key 的值会覆盖原先旧 key 的值，否则直接赋值。例如同一银行卡 5 天失败交易次数为 5 存入 redis 后，随着交易时间增加一天，同一银行卡 5 天失败交易次数为 13，那么 redis 里面存储的 key 为同一银行卡 5 天失败交易次数，值为 13。

缓存处理模块接收到标准数据后，定时任务会根据业务类型和维度计算这一批数据。例如业务类型为信用卡交易，维度为用户号，历史中间值为同一用户过去 5 天的平均交易金额，那么定时计算任务会根据这些条件，找出这一批交易流水中业务类型为信用卡交易，维度为用户号的数据，统计出这一用户过去 5 天内的平均交易金额，每次进行 redis 的 get/put 操作，会进行一次 merge 操作，当同一用户过去 5 天的平均交易金额这个中间值存在，会进行合并或者替换操作。由于涉及到的不仅仅是键值对的存储，这些数据需要一个固定的数据结构进行存储，例如用户号为 0908070038，时间戳为 1419317364000（对应时间为：2014-12-23 14:49:24），统计出来的同一用户过去 5 天的平均交易金额为 5000，在内存数据库中存储的模型如下图 4.7：

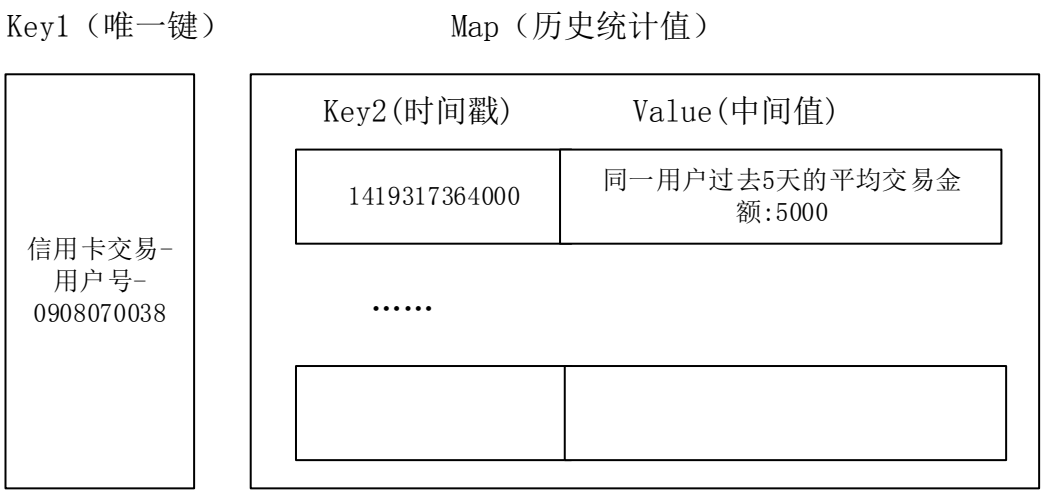

图 4.7 缓存处理模型图

a)信用卡-用户号-0908070038：唯一确认流水历史统计值。信用卡代表了流水的业务类型，用户号代表了流水的维度，0908070038 代表了流水的维度值，每一条流水过来，都可以通过这三个元素唯一确定这条流水的历史统计值。

b)历史统计值：规则平台进行规则自动化验证的时候，会根据验证流水的业务类别、维度以及维度值来获取对应的历史统计值，得到的数据是一个 Map 结构，里面主键是交易时间对应的时间戳，值是具体的历史统计数据值。

4.3.2 redis 的使用

针对规则自动化验证，本次采用了两台 redis 服务器，默认端口号 6379，由于只做规则自动化验证，没有考虑 redis 的主备模式，本次采用了 redis 客户端 jedis 做简单的 redis 操作，利用 jedis 中 ShardedJedisPool 和 JedisShardInfo 来做简单的分布式读和写。ShardedJedisPool 每一次都会通过 key 来找到连接池中一台对应的 redis 缓存进行读写。ShardedJedisPool 默认的读写采用的是一致性 hash 算法[23]。虽然统计的历史数据量不是很大，但随着规则数量的增加和时间的推移，还是会产生大量的历史数据，这样会大量的占据机器内存，本次在银联商务的规则验证当中，两台 redis 服务器占用内存情况如表 4.1：

表 4.1 风险示例数据表

redis1	redis2	应用服务器一	应用服务器二	应用服务器一 CPU	应用服务器二 CPU
62.18G	59.47G	234.5G	226G	20%	15%

（1）redis1 和 redis2：redis 数据库采用分布式存储，在验证 48 条规则的情况下，两台 redis 服务器分别占用 62.18G 和 59.47G 内存。

（2）应用服务器一和应用服务器二：两台服务器分别拥有 256G 的内存，可使用的内存分别为 234.5G 和 226G。

（3）应用服务器 CPU：在验证 48 条规则的情况下，两台应用服务器 CPU 使用率分别为 20%和 15%。

从表中可以看出，随着时间的推移和增加许多复杂规则的情况下，缓存数据占用内存会逐渐增大，由于缓存对象是整个规则自动化验证的重要组成部分，需要保证足够多的内存空间。但是机器的内存不可能毫无止尽，没有那台机器的内存可以无止境的存储数据，应当采取一定的措施。本次规则验证采用的办法是当缓存数据达到内存使用的 50%的时候，就应该考虑一定的算法，当历史统计值最近很少被使用的时候，将会被 redis 替换出去。redis 中默认采用最近最少使用算法进行历史统计值的替换，例如同一用户过去 5 天的平均交易金额这一统计值，当 redis 存储容量达到一定值后，如果这一统计值最近一直没被使用就会被 redis 替换出去。不管怎样，这样会这也会使得规则的验证有误差[23]。

4.4 本章小结

本章主要介绍了数据装载插件、消息队列和缓存处理三个模块。整个规则自动化验证当中的第一块：数据装载。主要包括静态数据装载和动态数据装载，动态数据复杂多变，定时任务需实时进行装载；静态数据改变频率小，采用一定时间间隔的定时任务进行装载。针对定时任务获取的取样数据会时常修改一些查询参数，接下来就介绍了用 Spring-Boot 的方式快速构建应用程序，使得可变化的参数能够灵活修改。整个规则自动化验证当中的第二块：消息中间件队列的安装与设计、队列按时间顺序进行存放消息。创建好队列后，结合几大模块就可以对流水数据按时间顺序进行处理，最后进行规则自动化验证，这样使得规则自动化验证的效率大大增加，正确性也会明显提升。整个规则自动化验证的第三块：缓存处理模块会根据数据装载过来的标准数据按业务类型和维度进行过滤，通过定时任务进行处理得到历史统计值，最后将这些历史中间数据值按缓存模型进行存储。采用了 jedis 方式进行操作 redis 数据库，通过 ShardedJedisPool 来进行简单的 get 和 put 操作。

第5章 风控规则自动化验证的应用

5.1 规则自动化验证的应用

规则自动化验证的应用主要分为两大部分，第一部分是规则投入使用之前进行规则自动化验证，即规则自动化验证；第二部分是规则投入使用之后进行规则自动化验证，即回归验证。第一部分主要结合风控规则平台，风控规则平台主要结合是drools 规则引擎、ExtJS 和 Spring 框架搭建起来的一款方便用户操作的可视化平台[24][25]。该平台可以灵活的配置规则，有效的进行参数配置，支持规则的热部署等等，结合风险触发的本次规则自动化验证主要结合了规则平台的一些功能模块和缓存数据库，进行对已配置好的规则进行测试，验证其准确性。第二部分主要结合风险库和标准数据库，将风险库中触发的流水和流水触发的规则匹配出来，选取其中的流水号关联查询标准数据库，将得到的结果与所触发的规则进行比较，看是否对应，验证其漏报率和覆盖率。具体设计方案如图 5.1：

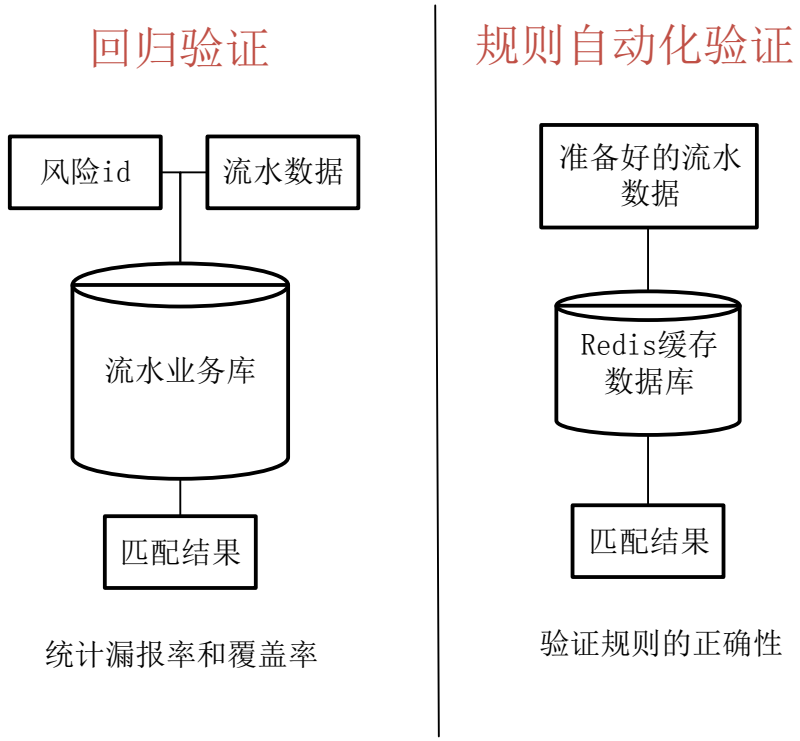

图 5. 1 规则自动化验证比较图

5.2 规则自动化验证应用的功能模块

为了做好规则自动化验证的初始工作，需要对已有的规则平台进行一定量的修

改和配置，前面介绍了数据转载插件、消息队列和缓存处理，在规则平台中需要用到缓存中的历史数据以及需要测试的流水。首先需要对规则平台里面 redis 连接池进行配置，保证规则平台能够及时的抓取到缓存中的历史中间值，其次就是消息队列里面的交易流水，由于本次规则自动化验证数据量少，仅仅通过配置就可以模拟一条交易流水进行验证。除此之外，那就是针对规则的设计编写了，规则的设计和编写是规则验证的基本条件，规则自动化验证应用的结构分布如图 5.2：

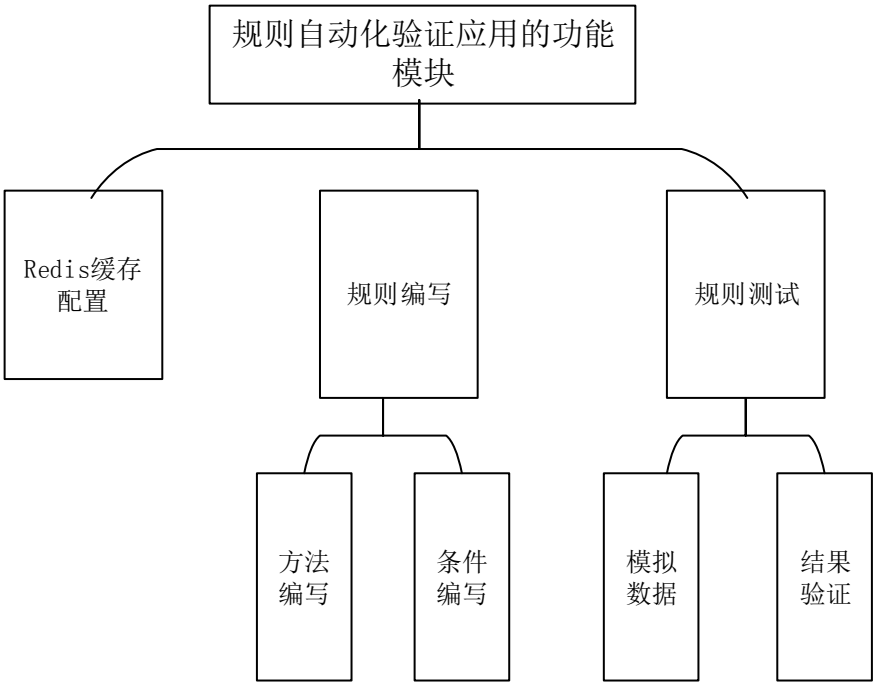

图 5. 2 规则自动化验证应用模块图

5.2.1 redis 缓存配置

redis 缓存配置主要提供了规则平台访问缓存数据库的连接池，规则平台需要从缓存数据库拿历史统计值，redis 的配置连接是否正确是规则平台正常运行的重要条件。针对规则自动化验证，本次采用了两台 redis 缓存数据库，默认端口号 6379，原始的规则平台采用单节点方式进行配置，现在改为分布式的方式进行配置，本次测试采用了 198.17.1.191 和 198.17.1.192 两台测试机[26]。具体配置如下：

```
    <bean id="node1" class="redis.clients.jedis.JedisShardInfo">
                <constructor-arg value="198.17.1.191" />
               <constructor-arg value="6379" />
                </bean>
    <bean id="node2" class="redis.clients.jedis.JedisShardInfo">
                <constructor-arg    value="198.17.1.192" />
                <constructor-arg    value="6379" />
                </bean>
<bean id="shardedJedisPool" class="redis.clients.jedis.ShardedJedisPool">
                <constructor-arg>
                <ref bean="poolConfig"/>
                </constructor-arg>
                <constructor-arg>
                <list>
                <ref bean="node1" />
                <ref bean="node2" />
                </list>
                </constructor-arg>
      </bean>
```

以上配置节点 node1 和节点 node2，作为参数传递给 shardedJedisPool，shardedJedisPool 每一次通过 key 键值去读取的时候，shardedJedisPool 默认会根据一致性哈希算法找到那一台 redis 缓存数据库，获取相应的数据。在此之前，原始规则平台采用单节点进行配置，具体配置如下：

```
<bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate">
<property name="connectionFactory" ref="connectionFactory" />
</bean>
```

redisTemplate 支持单点的 redis 操作，而 shardedJedisPool 可以支持多台 redis 服务器进行操作，配置好 redis 的连接池以后，规则平台会通过业务类型和维度值从 redis 中获取历史统计值，缓存处理模块负责历史数据的 set 操作，相反，规则平台通过 shardedJedisPool 来进行 get 操作，具体代码如下：

```
List<MemCachedItem> items = new ArrayList<MemCachedItem>(10);
ShardedJedis shardedJedis = null;
ShardedJedisPipeline pipeline = null;
shardedJedis = shardedJedisPool.getResource();
pipeline = shardedJedis.pipelined();
pipeline.get(keySerializer.serialize(keyId));
List<Object> list = pipeline.syncAndReturnAll();
for (Object o : list) {
items.add(valueSerializer.deserialize((byte[]) o));
} shardedJedisPool.returnResource(shardedJedis);
```

5.2.2 规则编写

规则自动化验证的过程当中，首先你要设计好你想要用的规则，本次规则自动化验证主要包括编写方法和编写条件，再将这几个条件进行组合，组合得到的就是我们设计好的规则。例如设计一条规则：商户 10 日退单笔数大于 5 笔，这一条规则可以拆分成一个条件（商户 X 日退单笔数大于 Y 笔）和一个方法（统计退单笔数），具体方法和条件编写如下：

a)条件编写：主要编写规则中的常量参数、模拟流水字段以及逻辑函数。例如商户 10 日退单笔数大于 5 笔这一条规则,拆分成条件后变成商户 X 日退单笔数大于 Y 笔，将 10 日变成 X 日，将 5 笔变成 Y 笔；在条件中，你需要对 X 和 Y 变量赋予初始值，条件中配置这两个参数的时候默认设置成为 10 和 5，当你在规则中修改这些值的时候，初始值将会被覆盖；例如将 X 变为 20，Y 变成 10，那么新规则将变成商户 20 日退单笔数大于 10 笔。除此之外，条件中还需要配置一些流水字段参数和逻辑判断的方法，流水字段参数结合逻辑方法能够返回商户 X 日退单笔数，将得到的数值与 Y 比较大小，以此来判断是否产生风险数据。

b)方法编写：主要结合条件一起使用。例如将商户 X 日退单笔数得到的数值与 Y 比较大小的这个方法设计成为 function boolean getBackOrders(X，Y，时间戳，状态，历史中间值)，这一方法返回 boolean 类型，输入参数为历史中间值、交易时间、交易状态、未知数 X 和 Y。历史中间值可以从缓存数据库中获取，交易时间和交易状态可以从流水数据中获取，未知数 X 和 Y 是规则中配置的常量数据。首先通过历史中间值、交易时间和 X 日来获取商户

X 日退单笔数 M，然后判断当前这笔流水是否也是退单状态，如果是，则 M=M+1，最后 M 与 Y 比较大小，返回布尔类型值（return M>Y）。

5.2.3 规则测试

规则自动化验证最重要的一块就是规则测试，首先规则测试需要准备充分的流水数据，结合已有的规则、条件和方法来验证结果的正确性。其次需要检查验证结果中的缓存信息（来源于缓存数据库），最后验证有无风险数据，如果出现的 risk 和缓存信息相匹配，说明规则是准确的，对于一批流水，如果正确率达到 100%,那么这条规则就可以投入使用了。规则测试步骤如下：

流水准备：本次规则验证只支持少量的单个流水数据验证，对于消息中间件过来的一批流水，暂时还无法满足多条流水一起验证。针对每一条流水，必须符合一定的数据结构标准，例如测试商户 10 日退单笔数大于 5 笔这一条规则，首先必须找出这一条规则中涉及的字段值，商户号（merchant_no）、交易状态（trans_status）、交易时间（trans_times）和业务类型（buss_code），这四个值必须找到且字段名要一样,因为在规则的验证过程当中会比较交易流水字段与条件中使用的字段的一致性，如果两方数据结构不一致，即使其他过程都正确，这个规则也会不满足条件当成错误处理。所以在模拟数据的时候必须看清规则中用到了那个字段名，例如规则中用到了 merchant_no，那么在你的模拟的流水里面必须有 merchant_no 这个字段。业务类型字段（buss_code）不同于一般流水字段，它代表的是一大类业务类型，例如信用卡交易和手机业务等等，商户 10 日退单笔数大于 5 笔这一条规则可以用于信用卡交易业务，也可以用于手机业务，这取决于 buss_code 是取那个值，这里不要搞混淆了。

结果验证：模拟好流水数据后，接下来就是对结果进行验证，点击验证按钮就可以得到一大批对象结果，其中包含你设计好的模拟流水对象、从缓存数据库里面获取这一商户的缓存对象以及满足规则触发的风险对象，要好好比较这三个对象之间的关系。例如验证商户 10 日退单笔数大于 5 笔这一条规则，首先看缓存对象，通过模拟流水中的交易时间（trans_times）向前推 10 个工作日找到的相应的退单笔数。再把找到的这 10 天的退单笔数和流水对象中的退单状态结合起来，比较是否退单笔数大于 5 笔（即是否可以触犯风险），有以下四种情况：

a)比较如果不能触发风险，而得到的结果里面出现了风险对象，那么规则有错误。

b)比较如果能触发风风险，而得到的结果里面没有出现风险对象，那么规则有错误。

c)比较如果不能触发风险的，得到的结果里面的确没有风险对象，那么规则正确。

d)比较如果能触发风险，得到的结果里面的确有风险对象，那么规则正确。

5.3 回归验证应用的功能模块

前小节介绍了规则自动化验证（规则投入使用之前），本节讲解针对规则投入使用之后的回归验证，回归验证主要针对流水触发的风险来验证规则的有效性、误报率和覆盖率。例如商户 10 日退单笔数大于 5 笔这一条规则，触发风险 1000 条，针对这 1000 条数据进行标准数据匹配，看是否真真符合这一规则，以此来验证规则的误报率和覆盖率，假设这 1000 条中有 3 条误报，那么误报率为 0.3%，假设这一条规则本可以触发 1050 条，那么覆盖率为 95.24%。如果误报率和覆盖率没有到达一定的标准，那么这条规则需要重新设计验证。回归验证的具体功能模块如图 5.3：

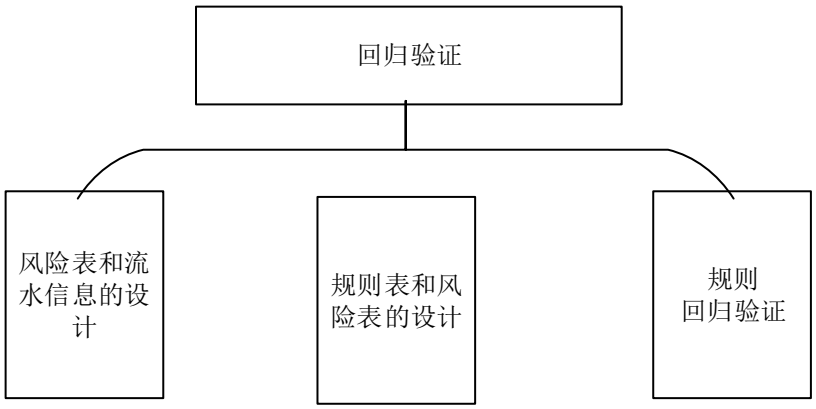

图 5. 3 规则回归验证模块图

a)风险表和流水信息的设计：针对规则平台产生的风险，风险数据会入库到风险表，储风险的同时，也存储了触发规则的模拟流水。

b)规则表和风险表的设计：风险生成的同时，会产生的风险编号，这个风险编号关联了对应的规则名称和规则编号，这些字段储存在规则表中，风险编号作为外键存储。

c)规则回归验证：根据规则的业务逻辑，将得到的流水进行匹配得到结果。

5.3.1 风险表和流水信息的设计

通过规则平台产生的风险信息都存储在风险表中，风险表很简单，除了主键风险编号和流水信息外，其他的就是一写需要在可视化界面中展示的一些信息，根据不同的业务需求，展示不同的效果。本次规则的回归验证只涉及到风险编号和流水信息，风险编号作为风险表的主键，流水信息作为 json 格式存储在风险表的一个字

段中，具体设计如表 5.1：

表 5. 1 规则自动化验证风险表

字段名	字段英文名	字段表示结构
风险编号	risks_id	一个 int，自增长，主键
流水信息	value	一个 json 格式,类似于： { merchant_id:"",biz_code:"", trans_time:"", trans_amount:"", ... , trans_status:"" }

上述表格中，风险编号在规则触发风险的时候就自动生成，流水信息是触发规则的那一条流水，会跟随风险信息一起入库，以 json 格式存储。例如流水（merchant_id:"0908070038", biz_code:"信用卡交易", trans_time:"2014-03-23 19:59:23", trans_amount:"1000", trans_status:"51"）触发了商户 10 日退单笔数大于 5 笔这一条规则，那么风险表存储的结果为表 5.2 所示：

表 5. 2 风险示例数据表

risks_id	1（自增长）
value	{ merchant_id:"0908070038", biz_code:"信用卡交易", trans_time:"2014-03-23 19:59:23", trans_amount:"1000", trans_status:"51"}

5.3.2 规则表和风险表的设计

本次规则回归验证，主要针对规则编号、规则名称和外键风险编号三个进行关联，通过规则名找到风险编号和流水信息，然后去查询标准数据库进行规则匹配。具体设计如表 5.3：

表 5. 3 规则自动化验证规则表

字段名	字段英文名	字段表示结构
规则编号	rules_id	一个 int ，自增长，主键
规则名称	rules_name	一个 varchar
风险编号	risks_id	来源风险表，外键键

产生的风险信息会同时入库到风险表和规则表，一条风险信息包含了规则信息、

流水信息和风险相关的属性信息，一部分入库到风险表，一部分入库到规则表，两则之间通过风险编号进行关联查询得到规则与流水之间的关系，一次来验证相关规则的有效性。例如流水（merchant_id:"0908070038", biz_code:"信用卡交易", trans_time:"2014-03-23 19:59:23", trans_amount:"1000", trans_status:"51"）触发了商户不是白名单商户和商户 10 日消费总金额大于 50000 元这两条规则，那么规则表和风险表的存储结果分别如表 5.4 和表 5.5 所示：

表 5. 4 规则示例数据表

rules_id	rules_name	risks_id
1	商户不是白名单商户	1
2	商户 10 日消费总金额大于 50000 元	1

表 5. 5 规则对应的风险数据表

risks_id	value
1	{ merchant_id:"0908070038", biz_code:"信用卡交易", trans_time:"2014-03-23 19:59:23", trans_amount:"1000", trans_status:"51"}

通过风险表和规则表就可以找出流水信息和规则的对应关系，以此来验证流水是否真真的能触发这条规则。有时候一个风险可能会触发 10 个及以上的规则，有时候一个风险仅触发一个规则，如果按风险和流水去验证规则的有效性将影响正常的回归验证，因为几个风险可能包含重复的规则，按风险和流水去验证规则的有效性将大量产生重复的逻辑判断。本次回归验证采用了规则表和风险表结合的方法，首先通过规则表中的规则名（一种规则名代表一种业务逻辑）来找出所有的风险编号，通过风险编号找出所有的流水信息，将这一批流水信息按照规则逻辑进行有效处理，最后结合标准数据库进行规则验证。

5.3.3 规则回归验证

规则的回归验证就像流程化处理一样，只要准备好了规则表和风险表中相应的规则和流水，通过规则的逻辑处理过程就可以进行回归验证。每一个规则的逻辑处理过程都不一样，但参数名几乎相同，可以事先设计好验证的规则逻辑处理模块。验证规则的参数主要是一些常量的参数、业务类型、维度以及交易流水时间等等。规则回归验证用到了标准数据库，标准数据库存储了数据装载插件转化的每一笔交

易流水，标准数据与流水数据几乎一摸一样，不同的地方在于，标准数据少了很多规则中不必要的字段，这些标准数据一部分会入到标准数据库，一部分会给缓存处理模块进行处理。具体模型如图 5.4：

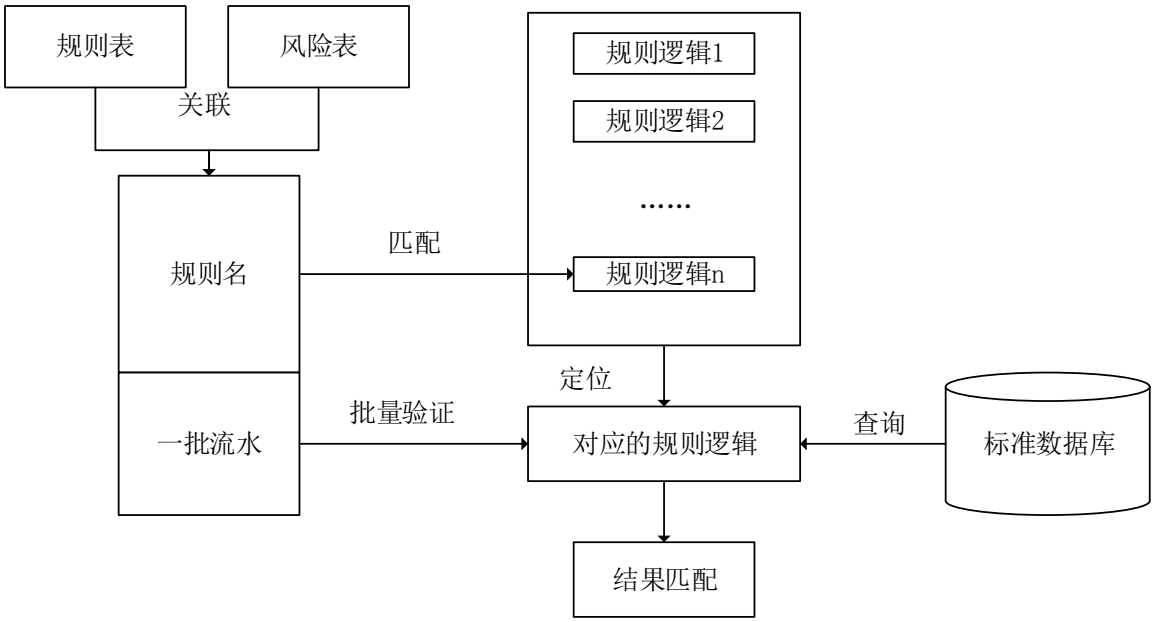

图 5. 4 回归验证流程图

针对上述模型，本次回归验证的重点就是设计规则逻辑，规则逻辑根据根据规则名来编写，例如商户 10 日消费总金额大于 50000 元这一规则逻辑，首先获取里面的数值常量 10 和 50000，其分别关联流水里面的 trans_time（交易时间）和 trans_amount（交易金额）两个字段；其次是规则的维度商户，关联对应字段 merchant_id(商户号)，选取好这些字段和值以后，最后就是写一条符合规则的 sql 语句，假设标准数据库里面的标准表为 ds_trans,则符合这条规则的 sql 语句为：select sum(d.trans_amount)>50000 from ds_trans d where d.merchant_id=merchantId &&d.trans_time>(transTime-10) && d.trans_time<= transTime，其中 50000 和 10 是规则中已知的参数，merchantId 是每一条流水中商户号字段，transTime 是每一条流水中时间字段，从数据库计算同一商户 merchantId 在时间段（transTime-10，transTime）累积交易金额是否大于 50000 元，如果是，结果返回 true，计数器加 1，否则返回 false，计数器减 1，将计数器的值与真真触发的风险数比较大小，统计误报率。假设任意一条流水（merchant_id:"0908070038", biz_code:"信用卡交易", trans_time:"2014-03-23 19:59:23", trans_amount:"1000", trans_status:"51"），对应的 sql 为 select sum(d.trans_amount)>50000 from ds_trans d where d.merchant_id='0908070038' &&d.trans_time> '2014-03-13 19:59:23' &&

d.trans_time<= '2014-03-23 19:59:23'

5.4 本章小结

本章主要介绍了规则自动化验证应用的两个模块：规则自动化验证（规则投入使用之前）和规则回归验证（规则投入使用之后）。规则自动化验证需要用到缓存处理模块和风控规则平台，风控规则平台可以编写规则、编写条件、编写方法以及模拟流水，规则平台还可以直观简单的进行规则测试。规则回归验证需要用到标准数据、风险表和规则表，这些标准数据是通过消息队列入到标准数据库的，和原始流水几乎一样，风险表和规则表中的数据是规则投入使用之后产生的信息，回归验证就是通过风险表和规则表得到得验证规则和对应的流水，再根据对应的规则逻辑进行匹配，统计误报率。

第6章 风控规则自动化验证应用效果展示

6.1 规则自动化验证应用效果

第五章讲述了两种规则验证的方法，在规则投入使用之前对规则进行验证，此部分用到了缓存数据库和规则平台；在规则投入使用之后对规则进行回归验证，此部分用到了标准数据库和规则模型。标准数据库和缓存数据库依赖数据转载插件和消息中间件。以下小节是针对这两种规则自动化验证应用的效果展示。

6.2 规则自动化验证（规则投入使用之前）

本文的第五章讲述了规则自动化验证的基本过程，首先配置好 redis 缓存数据库地址，紧接着设计好自己想要的规则（需要编写方法、条件和业务规则），最后准备好模拟的流水进行规则测试。规则自动化验证主要基于规则平台进行测试，例如对卡号不在白名单中这一规则进行验证，首先预先设计好这一规则，第一步,编写 notInWhiteCardNameList 方法如下图 6.1 所示：

frms_global *notInWhiteCard...

删除 复制 验证

```
function boolean notInWhiteCardNameList(Object obj, Object card_no, Object tag,
Object nameListType) {
    //不在白名单中
    if (obj != null) {
        System.out.println();
    }
    String memKey = MemCachedItem.getMemCachedKey(String.valueOf(card_no),
    String.valueOf(tag),String.valueOf(nameListType));
    System.out.println(memKey);
    if (obj instanceof Set) {
        Set nameListSet = (Set) obj;
        if (nameListSet.contains(memKey)) {
            return false;
        }
    }

    return true;
}
```

图 6. 1 notInWhiteCardNameList 方法图

第二步，编写条件里面的参数和方法，条件里面主要包含了一些常量参数、模拟流水中需要的字段和一些简单的方法，这里预先设定好流水字段的名称，为

后面规则测试做准备，这些流水名称和一些常量值一部分将作为参数给方法使用。 具体编写卡号不在白名单中这一方法的界面如下图 6.2 所示：

图 6. 2 卡号不在白名单中条件图

第三步，编写完条件和方法后，就可以设计卡号不属于白名单这一规则，新建卡号不在白名单中这一规则，将卡号不在白名单中这一条件勾选上，保存即可设计好规则，具体如下图 6.3 所示：

图 6. 3 卡号不在白名单中规则图

第四步，设计模拟流水，进行规则测试，模拟流水字段和条件里面的流水参数字段要对应，这样规则才能有效的执行，在条件中用到了业务类型(frms_biz_code)和卡号（frms_bank_card_no），具体配置如图 6.4 所示：

图 6. 4 模拟流水数据图

第五步，比较产生的对象，验证卡号不在白名单中这一条规则的正确性，具体如下图 6.5：

图 6. 5 验证结果图

图中展示了当前模拟的这一笔交易的流水（AuditObject）、缓存数据

（MemCachedItem 历史统计值）以及触发的风险(Risk)，比较这三个对象，验证卡号属于白名单这一条规则的正确性。在 MemCachedItem 缓存中,针对卡号 frms_bank_card_no（银行卡）只存在 FRMS_白名单_银行卡_123456 这个维度值，其银行卡的值为 123456，而根据第四步模拟的银行卡 1234567 不在这个缓存集合里面，即该卡号不属于白名单，应该触发风险。确实，在验证得到的结果中，风险（Risk）触发了，也就证明了卡号不在白名单中这一条规则的正确性。

图 6. 6 缓存数据信息验证图

反过来，将模拟流水对应的银行卡号改为如下图 6.7 所示的数据值：

图 6. 7 模拟流水图

验证得到的结果如下图，由于银行卡号 123456 在缓存的白名单中，风险（Risk）并没有触发，反过来也证明了卡号不在白名单中这一条规则的正确性。如图 6.8：

图 6. 8 验证结果图

6.3 规则回归验证（规则投入使用之后）

规则的回归性验证主要在规则投入使用之后，结合了标准数据库中的历史流水数据、规则表、风险表和规则业务逻辑进行规则的自动化验证。首先确定验证的某条规则，通过规则表和风险表来查出触发这一条规则的所有流水；其次写好规则的业务逻辑模块；最后结合标准数据库，让这一批流水分别得通过业务逻辑模块，统计漏报率和误报率。下面用以下七条规则为例来进行说明，需要进行回归验证的七条规则如下：

1:商户单日同卡交易笔数大于 5 且交易金额大于 100000

2:商户 5 日退货金额大于 20000 元

3:商户 5 日不成功交易占比大于 50%

4:商户信用卡单笔交易金额大于等于 100000

5:商户单日同卡 bin 交易笔数大于等于 10 笔且交易金额大于等于 100000 元

6:商户日交易额较前 5 日平均日交易额突增大于等于 200%且交易额突增大于等于 100000

7:商户 5 日交易总笔数大于等于 100 笔且总金额大于等于 1000000

针对以上七条规则，第一步，输入你要查询的规则，具体如下代码所示：

```
BufferedReader reader=new BufferedReader(new
    InputStreamReader(System.in));
System.out.println("规则查询方式有 1,2,3,4,5,6,7 七种方式");
System.out.println("1:商户单日同卡交易笔数大于 5 且交易金额大于
    100000");
System.out.println("2:商户 5 日退货金额大于 20000 元");
System.out.println("3:商户 5 日不成功交易占比大于 50%");
System.out.println("4:商户信用卡单笔交易金额大于等于 100000");
System.out.println("5:商户单日同卡 bin 交易笔数大于等于 10 笔且交易金
    额大于等于 100000 元");
System.out.println("6:商户日交易额较前 5 日平均日交易额突增大于等于
    200%且交易额突增大于等于 100000");
System.out.println("7:商户 5 日交易总笔数大于等于 100 笔且总金额大于
    等于 1000000");
System.out.println("请输入查询的方式： ");
int index=reader.read()-48;
```

第二步，针对你的选择进入相应的规则业务逻辑方法，执行相应的逻辑，具体代码如下：

```
while(true){
    if(index==0)
    break;
    switch (index) {case ONE:oneTest();break;
    case TWO:twoTest();break;
    case Three:threeTest();break;
    case FOUR:fourTest();break;
    case FIVE:fiveTest();break;
    case SIX:sixTest();break;
    case SENVE:senveTest();break;
    default:break;}
    index=reader.read()-48;}
```

第三步，规则回归验证。以上七个方法对应了七个逻辑模块，具体以 twoTest() 为例来展现商户 5 日退货金额大于 20000 元这一规则的回归验证，根据商户 5 日退货金额大于 20000 元这一规则名查询出所有的风险编号，针对每一个风险编号进行验证，具体逻辑如下：

1.针对每一个风险编号可以通过 SQL 语句获取相关流水信息，具体语句为 String sql3="select value from audit_object where ARCHIVES_ID=?"，执行 ps3 = conn3.prepareStatement(sql3)、ps3.setString(1，str)、

ResultSet resultSet=ps3.executeQuery()，以此来获取 json 格式流水对象，JSONObject json = JSON.parseObject(resultSet.getString(1));

2.获取到流水 json 后，根据商户 5 日退货金额大于 20000 元这一规则，找出 json 中商户号[27]。merId=json.get("frms_merchant_id").toString();找出交易时间 String mid=json.get("frms_trans_time").toString();根据 mid 时间获取前 5 天的交易时间 c.setTime(new Date(Long.valueOf(mid))) 、c.add(Calendar.DATE，-5)、fiveDaysTime=sdf.format(c.getTime())。

3.得到商户号 merId、当前交易时间 mid 和 5 天前的交易时间 fiveDaysTime，根据商户 5 日退货金额大于 20000 元这一规则，结合数据库，执行语句。String sql="select sum(trans_amt_chnl) from TBL_TRANS_LOG where MCHNT_CODE_OUT=?"+"and rec_crt_time>=? and rec_crt_time<=?and ROUT_BUSS_CODE in(?,?,?,?)";其中 trans_amt_chnl 是流水中的交易金额，MCHNT_CODE_OUT 指的是商户号，rec_crt_time 指的是交易时间，ROUT_BUSS_CODE 指的是退货状态，TBL_TRANS_LOG 指流水表，根据 ps = conn.prepareStatement(sql);ps.setString(1, merId);ps.setString(2, fiveDaysTime);ps.setString(3,time);ps.setString(4,"E84");ResultSet resultSet=ps.executeQuery(); 得到查询的总金额 daysAmount=Long.valueOf(resultSet.getString(1));如果 daysAmount 大于 20000，则成功总数加 1，否则失败总数加 1，最后通过失败的总数除以验证的总风险数获得规则的误报率[28]。

6.4 本章小结

本章主要介绍了规则验证应用的两大模块，以截图的方式展现了这两块应用的基本操作过程。一个是在规则投入使用之前进行规则的验证，一个是在规则投

入使用之后对规则进行回归验证，这两种方式的应用，保证了规则的有效使用，确保规则在整体项目的应用当中发挥高质量的效果。规则投入使用之前，会进行规则的准确率验证。这需要缓存数据来保证实时稳定的效果。在这些规则验证通过后，要针对触发的风险对规则进行回归验证，进一步确保规则的稳定性。

第7章 总结与展望

7.1 论文总结

论文主要结合第三方支付平台的规则进行了自动化验证的研究与应用，讲述了规则自动化验证的三大块设计，进行了规则自动化验证的效果展示。第三方支付平台需要严格准确的规则，规则自动化验证使得三方支付平台能快速进行规则的制定和使用。本文第二章介绍了规则验证涉及的相关技术；本文第三章总体概括了规则自动化验证的整体架构、整体流程图以及规则自动化验证的两大模块；第四章主要是规则自动化验证技术的研究，讲述了数据装载插件、消息队列以及缓存处理模块的研究与使用。其中数据装载插件采用时间增量模式进行数据的定时抓取，通过定时任务将数据标准化后入消息队列；消息队列主要负责几大模块之间的消息传输，提供消息存储的中转站，所有的消息供下一个模块处理的同时，会入标准数据库进行数据的副本保存；缓存处理模块采用了 redis 缓存数据库，缓存处理模块将符合规则的数据按一定条件进行处理统计，将得到的中间值按照缓存模型进行存储。第五章主要讲述了规则自动化验证的应用，介绍了在规则投入使用之前，结合规则平台如何进行规则的验证；在投则投入使用之后，结合标准数据库如何进行规则的回归验证，这两大块都属于规则自动化验证范畴，不同在于一前一后。第六章进行了规则自动化验证的效果展示，以截图的方式展示了规则自动化验证（规则投入使用之前），以代码的方式展示了规则的回归验证（规则投入使用之后）。

7.2 将来的工作

本文第四章讲述了规则自动化验证的研究，针对其中数据装载插件的设计，当交易时间小到纳秒级别的时候，会出现丢数据的情况，可以考虑使用联合主键进行增量查询；针对其中消息队列的设计，仅仅考虑了单队列（即生产消费都连接到这个队列），这样会造成当消息队列出问题的时候，不知道是那个模块出现了问题，一个好的消息队列应该是生产模块连接一个队列，消费模块连接一个队列，两者之间通过传输队列进行通讯；针对其中缓存处理模块的设计，仅仅考虑了简单的分布式，没有进行缓存数据的迁移和主备，当其中一台宕机后，对规则的验证将产生很大的影响。这三大块都应该考虑主从或是交叉主从，特别是消息队列的设计，一旦消息队列挂掉，连接消息队列的两端模块将会停止运行，直接

影响规则自动化验证，应保证消息队列挂掉之后其他队列能接替它进行正常的后续工作。本文是基于金融实时风控规则自动化技术的研究与应用，真真上来讲，还没有达到实时的要求，只能达到准实时的标准，因为流水已进入数据库了。针对规则的自动化验证，目前规则平台每仅支持一条流水进行验证（每一批），以后应该支持批量操作。以上这些问题都值得在今后的工作当中继续研究。

参考文献

[1]贺瑀,王力宾,单薇.关于金融安全预警机制与风险控制的研究[J].时代金融,2012(30):153-154.

[2]李东卫.互联网金融:国际经验、风险分析及监管[J].长春市委党校学报,2014(3):36-40.

[3]唐正伟.第三方支付风险防范机制研究[J].商场现代化,2014(23):206-207.

[4]J. LaFleur.,R. E. Nelson.&Y. Yao., *etal.* Validated risk rule using computerized data to identify males at high risk for fracture[J]. Osteoporosis International, 2012,23(3):1017-1027.

[5]Kidd, Nicholas.,Lammich, Peter.&Touili, Tayssir., *etal.* A decision procedure for detecting atomicity violations for communicating processes with locks[J]. International Journal on Software Tools for Technology Transfer, 2011,13(1):37-60.

[6]Joshua Bloch. Effective Java 中文版第 2 版[M].杨春花,俞黎敏,译.北京:机械工业出版社, 2013-6.

[7]王心妍.Memcached 和 Redis 在高速缓存方面的应用[J].无线互联科技,2012(09):8-9.

[8]Hai-xia Cheng.,Ling-tong Min.&Xu-dong Lü, *etal.* A context-aware medical instant message middleware [J]. Springer,2015, 20(1):113-117.

[9]何艳群,戴祝英.运用 WebSphere MQ 实现消息的安全传输[J].软件导刊.2006(09):32-33.

[10] 孟凡红,李荭娜.IBM Websphere MQ 中间件应用浅析[J].信息系统工程,2015(01):91-92.

[11] 计文柯.Spring 技术内幕:深入解析 Spring 架构与设计原理[M].北京:机械工业出版社,2012-2.

[12] Agosta, Lou. Oracle Dominates Data Warehousing Survey ; Legacy Databases Take a Dive [J]. DM Review, 2005,15(4):57.

[13] 姜召凤.Oracle RAC 数据库缓存优化方法研究[D].硕士学位论文,大连海事大学, 2009.

[14] 高俊峰.高性能 Linux 服务器构建实战[M].北京:机械工业出版社, 2014-9.

[15] Chen, Min.,Mao, Shiwen.&Liu, Yunhao. Big Data: A Survey [J].Mobile Networks and Applications, 2014,19(2): 171-209.

[16] Ahuja, Sanjay P.&Mupparaju, Naveen. Performance Evaluation and Comparison of Distributed Messaging Using Message Oriented Middleware[J].Computer and Information Science,2014,7(4):9-20.

[17] 陈小艳,陈刚.基于 WebSphere MQ 的收发消息程序[J].计算机与信息技术. 2005(04).

[18] 胡畅,黄婷.利用 IBM MQ 实现远程消息安全传递[J].华南金融电脑,2005,09:100-101.

[19] Xiao Jing-zhong, Xiao Li, Huang Tian-yun. The Design and Implementation of Event Management System of Message Middleware[J]. Energy Procedia, 2011, 11(11) :1122-1127.

[20] 张胜,程贝.基于 Spring JMS 的船舶申报系统[J].计算机与现代化, 2014(03):127-130.

[21] Ming Xu.,Xiaowei Xu.&Jian Xu., *etal.* A Forensic Analysis Method for Redis Database based on RDB and AOF File[J]. Journal of Computers,2014,9(11): 2538-2544.

[22] 吴霖,刘振宇,李佳.Redis 在订阅推送系统中的应用[J].电脑知识与技术, 2015,7(11):292:294.

[23] Tolk, Andreas.&Aaron, Robert D. Addressing Challenges of Transferring Explicit Knowledge, Information, and Data in Large Heterogeneous Organizations: A Case Example from a Data-Rich Integration Project at the U.S. Army Test and Evaluation Commande[J]. Engineering Management Journal, 2012,22(2): 44-55.

[24] 刘金龙. drools 规则引擎模式匹配效率优化研究及实现[D].硕士学位论文,西南交通大学, 2007.

[25] 黄灯桥. Ext JS 权威指南[M].北京:机械工业出版社,2012-06.

[26] 曾泉匀.基于 Redis 的分布式消息服务的设计与实现[D].硕士学位论文,北京邮电大学, 2014.

[27] 龚建华.JSON 格式数据在 Web 开发中的应用[J].办公自动化, 2013(20):46-48.

[28] 甄真,陈虎,张林亚. 列数据库的 SQL 查询语句编译与优化[J].计算机工程, 2013(06):61-65.

作者简历

教育经历：

2009 年 9 月 至 2013 年 6 月 本科就读于贵州大学计算机科学与技术专业。

2013 年 9 月 至 2015 年 6 月 硕士研究生就读于浙江大学软件工程专业。

工作经历：

2014 年 4 月至 2015 年 4 月杭州邦盛金融信息技术有限公司实习。

2013 年 9 月至 2014 年 3 月宁波电子服务研究所开发组实习。

攻读学位期间发表的论文和完成的工作简历：

2014 年 12 月至 2015 年 4 月参与金融实时风控规则自动化验证技术研究与应用。

致谢

转眼两年的时间就要到了，在读研究生的两年时光里，我感受到了宁静的校园环境和浓厚的学习氛围，在这里，不仅学习上得到了较大进步，生活上也收获颇多。在这段时间期间，我深深的喜欢上了有着良好的学术环境的实验室和图书馆，也喜欢上了周边浓烈的学习氛围，在导师的指导下，我在理论和实践应用方面都得到了很大提升，在论文完成的过程当中，更是得到了身边很多人的帮助，在此，我要对他们表示最真诚的感谢。

首先要感谢我的导师贝毅君，他在我的学习生活中给予了很多的帮助。在实验室学习时，他总会问问我们目前的学习近况以及指导我们制定学习计划，时常也会带着我们参加一些项目会议，并提供一些学习书籍，这些都让我积累了宝贵的理论和实践经验；在企业实习过程中，他时常会找我聊聊天，问问最近的实习情况以及工作去向的问题，在论文完成的整个过程中，帮助并确定论文的整体逻辑架构。在我眼里，贝老师是一个学术渊博、关心学生的好老师，没有他的帮助，我很难在短时间内得到这么大的提高，在此我要向他表达深深的谢意。

再次，要感谢和我一起实习的邦盛同事，他们帮助我在实习过程当中克服困难，让我很快能适应实习的工作节奏，同时，还要感谢同一个项目组的成员杨志强和王恺，他们经常给我讲解一些项目上的技术问题，指出我的错误并及时给予纠正，这对我的能力提升起到了很大的帮助。

同时，我要感谢我的父母，在我忙碌的时候他们总是记得打电话关心我，鼓励我努力学习，好好工作，无形之中给了我莫大的帮助。

最后，祝我的导师，父母，同事，朋友身体健康，平安幸福！

2015 年 4 月

吴必阳

于浙江大学软件学院