北京大学

硕士研究生学位论文

题目：**基于规则的物联网消息处理系统的设计与实现**

姓　　名：	陈　肯
学　　号：	2001210200
院　　系：	软件与微电子学院
专　　业：	软件工程
研究方向：	智能科技
导师姓名：	陈向群　教授

二〇二三 年 六 月

版权声明

摘要

物联网消息处理系统是统一接收物联网设备消息，并为物联网应用软件提供数据访问接口的软件系统。为满足物联网应用软件的业务需求，消息处理系统需要对设备产生的实时数据进行拦截验证，根据验证结果执行相应的业务逻辑，例如将消息发布到消息队列，或者向外部系统发送 HTTP 请求。设备消息的验证条件与后续的业务逻辑并称为消息处理规则。消息处理规则的部署是物联网系统实现自动化与智能化的关键，控制设备通常从特定的消息队列中取出控制指令，而外部系统能为设备数据分析提供额外支持。由于每个设备的型号功能不同，应用场景也不同，消息处理规则的编写工作变得十分琐碎。如果使用硬编码的方式实现规则，将导致消息处理系统的维护难度过大。

本文参考 Drools 规则引擎与 Thingsboard 物联网平台的设计思想，以及领域特定语言的相关理论，使用 Go 语言设计并实现了一个基于规则的物联网消息处理系统。参考 DSL 相关理论，设计了用于定义消息处理规则的规则描述语言 DCA。系统利用 DCA 及四个运行时组件将消息处理规则与系统程序代码解耦。其中，消息接收器统一接收并存储设备消息；动作执行器执行具体的业务逻辑；异常状态管理器向物联网应用软件推送异常状态报警信息；规则调度器提供部署消息处理规则的 HTTP 接口，物联网应用软件使用 DCA 语言编写规则，并将其传入规则调度器。规则调度器随后使用规则解释函数解析 DCA 规则文本，并生成模式匹配函数。模式匹配函数的作用是验证消息接收器接受的某些设备消息，并通知动作执行器和异常状态管理器执行相应的业务逻辑与信息推送。规则调度器将模式匹配函数部署到子协程中持续运行，该子协程称为模式匹配器，规则调度器统一管理所有模式匹配器的运行状态。最后，给出了部署系统的一般步骤，并以某畜牧集团智慧农场的物联网设备消息处理为示例，部署了系统。为了使系统兼容更多物联网场景，满足不同层次的应用需求，给出了系统框架的扩展方案，并针对特殊应用场景给出了优化方案。

本文设计的基于规则的物联网消息处理系统以简明的规则描述语言控制物联网设备消息的处理流程。相比 Thingsboard 平台的规则链模型，系统支持配置多个设备的联动处理规则。为物联网应用软件提供了灵活的消息处理规则配置方式，降低了消息处理系统的维护成本。

关键词：规则引擎，物联网，消息处理，领域特定语言

DESIGN AND IMPLEMENTATION OF RULE-BASED IOT MESSAGE PROCESSING SYSTEM

Ken Chen(Software engineering)
Directed by Professor Xiangqun Chen

ABSTRACT

The IoT message processing system is the software system that uniformly receives IoT device messages and provides data access interfaces for IoT application software. In order to meet the business needs of IoT application software, the message processing system needs to intercept and verify the real-time data generated by the device, and execute the corresponding business logic according to the verification results, such as publishing the message to the message queue or sending HTTP requests to the external system. The validation conditions for device messages and the subsequent business logic are called message processing rules. The deployment of message processing rules is the key to the automation and intelligence of IoT systems, control devices usually take control instructions from specific message queues, and external systems can provide additional support for device data analysis. Due to the different model functions of each device and the different application scenarios, the writing of message processing rules becomes very trivial. If rules are implemented in a hard-coded way, it will make the maintenance of the message processing system too difficult.

This paper refers to the design ideas of the Drools rule engine and the Thingsboard IoT platform, as well as the relevant theories of domain-specific languages, and designs and implements a rules-based IoT message processing system using the Go language. Referring to DSL related theory, DCA, a rule description language for defining message processing rules, is designed. The system uses DCA and four runtime components to decouple message processing rules from system program code. Among them, the message receiver receives and stores device messages in a unified manner; Action executor executes specific business logic; The exception state manager pushes abnormal status alarm information to the IoT application software; The rule scheduler provides an HTTP interface for deploying message processing rules, and IoT application software uses the DCA language to write rules and pass them into the rule scheduler. The rule scheduler then parses the DCA rule text using the rule interpretation function and generates a pattern matching function. The role of the pattern

matching function is to verify some device messages accepted by the message receiver, and notify the action executor and the exception state manager to execute the corresponding business logic and push information. The rule scheduler deploys pattern matching function to a child goroutine called pattern matcher. Rule scheduler manages the state of all pattern matchers. Finally, the general steps for deploying the system are given, and the system is deployed using the IoT device message processing of a livestock group's smart farm as an example. In order to make the system compatible with more IoT scenarios and meet the application requirements of different levels, the expansion scheme of the system framework is given, and the optimization scheme is given for special application scenarios.

The rule-based IoT message processing system designed in this paper controls the processing flow of IoT device messages in a concise rule description language. Compared with the rule chain model of the Thingsboard platform, this system supports configuring linkage processing rules for multiple devices. It provides a flexible way to formulate message processing rules for IoT application software, which reduces the maintenance cost of the message processing system.

KEY WORDS: Rule engine, Internet of thing, Message processing, Domain specific language

目录

第一章 引言.....1
1.1 研究背景及意义.....1
1.2 国内外研究现状.....2
1.2.1 国外研究现状.....2
1.2.2 国内研究现状.....3
1.3 论文研究目标.....3
1.4 论文组织结构.....4
第二章 相关理论与技术.....5
2.1 Drools 规则引擎.....5
2.2 Thingsboard 物联网平台.....6
2.3 DSL 理论.....7
2.3.1 DSL 的定义与分类.....7
2.3.2 DSL 解释器.....8
2.3.3 语义模型.....10
2.4 Go 语言的并发机制.....12
2.4.1 协程并发模型.....12
2.4.2 管道通信机制.....13
2.4.3 定时任务框架.....13
第三章 消息处理系统的框架.....15
3.1 系统框架.....15
3.2 消息接收器.....17
3.2.1 设备消息规范.....17
3.2.2 消息接收器的工作逻辑.....18
3.2.3 数据源管理器与消息数据库.....19
3.3 规则调度器.....20
3.3.1 规则调度器的工作逻辑.....21
3.3.2 模式匹配器的调度.....22
3.3.3 规则的访问权限.....23
3.4 动作执行器.....24

3.4.1 动作执行器的工作逻辑 24
3.4.2 动作参数列表的定义与解析 25
3.4.3 子执行器的调用 26
3.5 异常状态管理器 26
3.5.1 异常状态管理器的工作逻辑 26
3.5.2 异常消息体格式 28
3.6 本章小结 28
第四章 规则描述语言的设计与实现 29
4.1 语言设计与规则解释函数 29
4.1.1 规则描述语言 DCA 的设计 29
4.1.2 规则解释函数的总体流程 31
4.2 数据源实体的解析 32
4.2.1 双符号表策略 32
4.2.2 外符号表生成算法 33
4.3 规则主体的解析 34
4.3.1 表达式匹配条件的解析 34
4.3.2 函数匹配条件的解析 38
4.3.3 动作语句的解析 38
4.4 模式匹配函数的生成 39
4.4.1 模式匹配函数的逻辑 39
4.4.2 模式匹配算法 41
4.5 本章小结 42
第五章 系统部署及扩展优化 43
5.1 系统的部署 43
5.1.1 部署系统的一般步骤 43
5.1.2 部署案例 43
5.2 系统框架的扩展与优化 49
5.2.1 系统框架的扩展 49
5.2.2 针对特殊应用场景的优化方案 51
5.3 本章小结 54
第六章 总结与展望 55

6.1 总结 55
6.2 进一步工作 55
参考文献 57
致谢 59
北京大学学位论文原创性声明和使用授权说明 60

第一章 引言

1.1　研究背景及意义

物联网架构通常包含三个组成部分，分别是物联网设备、物联网消息处理系统、物联网应用软件。物联网设备包括各种类型的传感器和控制器，传感器用于采集环境信息，控制器可以接收指令并控制现场设备。物联网应用软件提供直观的可视化页面，供终端用户查看设备上传的数据。

物联网消息处理系统的作用是统一管理物联网设备上传的消息，为物联网应用软件提供数据访问接口。物联网设备会周期性地采集现场环境信息，并将信息整合成消息，发送到消息处理系统。消息中包含了设备采集的环境数据、设备采集数据的时间、设备的运行状态等信息。

物联网设备消息的拦截验证和后续的业务逻辑并称为消息处理规则。由于设备的种类不同，设备部署的应用场景不同，导致判断设备是否正常的标准也不同。消息处理系统需要维护的消息处理规则因此变得非常繁杂，并且每当有新的设备接入后，都需要为该设备实现新的规则，这导致消息处理系统的维护成本变得很高。

仅仅将物联网设备消息上传并展示给用户的意义不大，只有通过对消息进行拦截验证，持续监控设备与现场状态，并自动化地执行业务逻辑，才能有效提高生产效率。因此，消息处理系统需要判断设备数据是否正常，并自动执行一定的控制策略。例如，为温度传感器设置温度阈值，如果检测到室温超过了阈值，就执行一些降温动作，比如通过控制器将现场的通风设施打开。

多个设备的联动控制也是常见的需求。联动控制指的是将多个设备的消息数据进行一定运算后，根据运算结果来控制另一批设备的状态。例如在农业场景中，监控一个种植大棚中的三个温度传感器，计算它们的平均温度，如果温度超过某值，就将该种植大棚的四个通风设备都打开。在实际场景中，三个温度传感器的型号可能不同，它们的消息格式不同，传输协议也不同，导致策略代码的复杂度较高，并且如果某个传感器进行了型号升级，那么之前编写的代码也需要做相应的改动工作才能正常运行。

随着应用场景发生变化，已部署的设备消息处理规则也需要及时改变。以个人穿戴式设备的实时监控与报警为例，用户在非运动状态时，心率监控报警的阈值会设置的较低，而用户在运动时，心率监控报警的阈值就会相应提高，同时因个人体质的不同，心率阈值也设置的不同。

经过上文分析可知，如果将消息处理规则硬编码到系统中，将导致消息处理规则

的扩展与维护难度过大。由此可见，物联网消息处理系统需要使用一种更加灵活简便的方式，来管理这些型号功能各异的设备消息，并以一个固定的模板来制定消息处理规则，使得规则可以随时改变，随意改变，且系统的程序代码不需要做太多修改。

1.2 国内外研究现状

1.2.1 国外研究现状

在计算机软件领域中，通常使用规则引擎来应对策略规则变化快速的场景。规则引擎是一种将规则定义与软件实现进行解耦的软件技术。规则通常是一组 IF-THEN 形式的语句对，其含义是如果满足了 IF 语句中描述的条件，就执行 THEN 语句中描述的动作。规则引擎可以解析规则语句，并将规则实际部署到软件系统中。

在国外，已经有许多在物联网领域使用规则引擎的相关工作。Charbel El Kaed 等人分析了工业环境中应用物联网设备控制策略的困难之处，并将规则引擎引入工业网关，实现了基于规则的灵活控制策略[1]。M. P. R. Sai Kiran 等人将规则引擎应用到医疗设备的数据上传中，由规则引擎从数据中提取重要特征，决定是否传输采集的数据，以此降低了设备功耗和网络流量[2]。Luca Mainetti 等人实现了事件-条件-行动（ECA）规则形式，在物联网体系结构中将规则的语义推理与执行细节分离开[3]。Bertha Mazon-Olivo 等人设计了由规则引擎和复杂事件处理器组成的双层物联网应用程序体系结构，规则引擎配置动态规则，调节不同源的输入数据，并为终端用户规划执行器、警报和通知的控制操作[4]。

开源物联网平台 Thingsboard[5]提供了较为完整的物联网解决方案，并实现了一种链式的规则引擎，作为设备数据管理的核心组件，在物联网领域内应用较为广泛。Lucio Tommaso De Paolis 等人使用 Thingsboard 平台统一收集传感器数据，并使用 Spark Streaming 执行数据分析[6]。T M Kadarina 与 R Priambodo 在物联网应用开发中，使用 Arduino 单片机收集心率和血氧饱和度数据，之后用树莓派将异常数据统一转发至 Thingsboard 进行详细分析[7]。

Thingsboard 的规则引擎使用 JavaScript 语法定义规则，虽然大幅提升了规则制定的灵活度，但也增加了用户的学习成本和使用门槛。同时，Thingsboard 的规则引擎只能针对单一设备进行规则配置，难以实现设备的联动控制。

更通用的规则引擎能够实现复杂的控制规则，但面对物联网场景却普遍存在着性能问题。通用规则引擎主要使用 Charles L. Forgy 提出的 RETE[8]算法进行模式匹配。RETE 网络将待匹配的对象称为事实，模式匹配指的是事实与规则的匹配过程。RETE

算法将用户定义的规则编译为一种更加利于并行计算，且可以大幅度减少运算量的网络结构，称为 RETE 网络。

RETE 网络采用了 Alpha Memory 与 Beta Memory 的设计，两种 Memory 分别作用在模式匹配的不同阶段，将每次模式匹配的中间结果保存起来，当下一批数据到达时，若某个待匹配对象的数据没有发生改变，那么就可以直接使用上一次的中间运算结果。

在物联网应用场景中，设备采集的环境数值通常变化快速，且上传频率较高。在数据集合快速变化的情况下，RETE 网络的中间节点复用率低，导致生成复杂 RETE 网络的性价比也很低。

1.2.2　国内研究现状

纪配宇等人在物联网应用中使用规则引擎分离了大数据预警信息的业务逻辑和业务数据[9]。陈癮等人基于 Thingsboard 的规则引擎实现了异构传感器的物联网水质检测系统[10]。田瑞琴等人设计并实现了物联网网关中的轻量化规则引擎，解决了 JRules 等传统规则引擎在物联网网关应用中运行时间和响应时间较长的问题[11]。杨慧等人使用 Java 语言实现了基于规则的消息处理引擎，用于处理物联网异构平台之间的信息[12]。

国内一些物联网消息处理的相关工作使用了 Drools[13]规则引擎。叶昊等人利用 Drools 规则引擎实现了物联网复杂事件处理框架[14]。陈雪勇等人利用 Drools 实现了物联网平台中的脚本引擎[15]。冯俊辉等人提出了一套生成 Drools 规则代码的框架，并针对物联网环境构建了 Drools 规则[16]。Drools 的规则描述语言是基于 Java 语言的，在将其应用到非 Java 语言实现的系统中时，存在着兼容性问题。

1.3　论文研究目标

本文的研究目标是设计并实现一个基于规则的物联网消息处理系统，以规则描述语言来定义异构设备的消息处理规则，将消息处理的逻辑从系统的程序代码中分离出来。消息处理系统对不同通信协议与消息格式的物联网设备保持兼容性，力求在各种物联网场景中具备可用性。在设备消息格式改变时，消息处理系统需要修改的代码量应该尽量少。设计简洁明了的规则描述语言来制定设备的消息处理规则，在满足需求的前提下，尽量降低规则语法的复杂度。

主要工作内容包括：

（1）设计并实现基于规则的物联网消息处理系统框架，系统使用规则描述语言统一管理异构设备消息的处理逻辑。

（2）设计适用于物联网消息处理场景的规则描述语言，用于描述物联网设备消息的消

息处理规则，并实现规则语言的解释函数。

（3）给出本文提出的消息处理系统部署到实际物联网场景中的一般步骤，利用规则描述语言满足应用场景的消息处理需求。

1.4 论文组织结构

第一章引言，分析了物联网消息处理场景面临的主要挑战。由于在计算机软件领域中通常使用规则引擎来解决类似挑战，因此调研了规则引擎在物联网领域中的相关研究，明确了本文的研究方向与主要工作内容。

第二章相关理论与技术，介绍了 Drools 规则引擎的框架、DSL 的相关理论、Thingsboard 平台的整体架构，以及 Go 语言的并发机制，为实现基于规则的物联网消息处理系统提供了理论与技术支撑。

第三章消息处理系统的框架，设计并实现了基于规则的物联网消息处理系统的总体结构，以及消息接收器、规则调度器、动作执行器、异常状态管理器四个关键组件。

第四章规则描述语言的设计与实现，详细设计了基于规则的物联网消息处理系统的规则描述语法，实现了规则解释函数。

第五章系统部署及扩展优化，给出了部署系统的一般步骤，以某畜牧集团智慧农场的物联网场景作为案例，使用规则描述语言部署了符合实际生产需求的消息处理规则。给出了系统框架的扩展方案，及针对特殊应用场景的优化方案。

第六章总结与展望，总结了全文工作，并提出了进一步工作的重点。

第二章 相关理论与技术

本章介绍规则引擎与物联网消息处理系统的相关理论与技术。包括 Drools 规则引擎的结构、Thinsgboard 平台的整体架构、领域特定语言的相关理论，以及 Go 语言提供的高效并发机制。

2.1 Drools 规则引擎

Drools 规则引擎是一个应用广泛的基于 Java 语言实现的规则引擎。Drools 规则描述语言功能较为完备，其规则文件将被编译为 Rete 网络运行。Drools 规则文件的基本编写逻辑如下。

```
Drools 规则文件
rule “name” // 规则的命名
    attributes // 规则的基本配置
    when
        LHS // 触发条件
    then
        RHS // 后续动作
end
结束
```

规则文件的首行需要有固定标识 rule，其后跟随该规则文件的名字。在 rule 后可以进行基本的规则配置，如 no-loop，lock-on-active。when 块指明该规则匹配的条件部分，then 块指明匹配成功后需要执行的动作。Drools 规则文件支持全局变量定义、包管理等复杂语义。Drools 规定规则文件需要以固定的后缀名命名并放入指定的文件夹，以便引擎读取和应用。

Drools 规则引擎实现了 RETE 算法的全部细节，并且在其基础上进行了优化和扩展。在 Drools 规则引擎的架构中，Production Memory 用于存储用户上传的规则文件，Working Memory 用于保存事实及其数据，facts 与 rules 会进入 Pattern Matcher 进行匹配，匹配成功的 rules 如果不存在冲突，将被放入 Agenda 中等待执行。Pattern Matcher 的实现基于 RETE 算法或 Leaps 算法，且针对 RETE 算法的缺点有所改进，Drools 将其改进后的 RETE 算法称为 ReteOO（Rete Object Oriented）。

Drools 规则引擎的架构如图 2.1 所示。

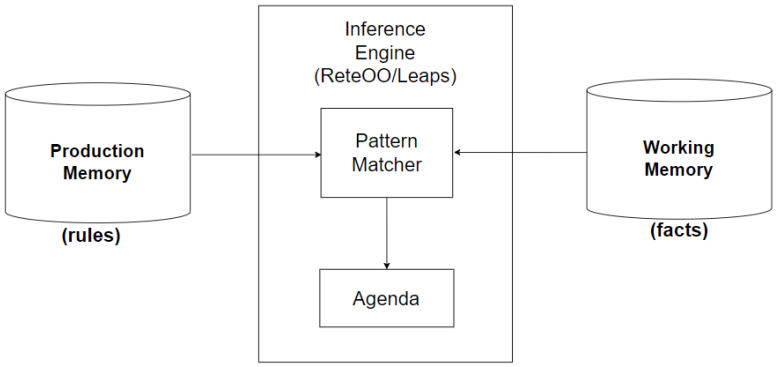

图 2.1 Drools 规则引擎的架构

2.2 Thingsboard 物联网平台

Thingsboard 是一个开源物联网平台项目，其架构如图 2.2 所示。

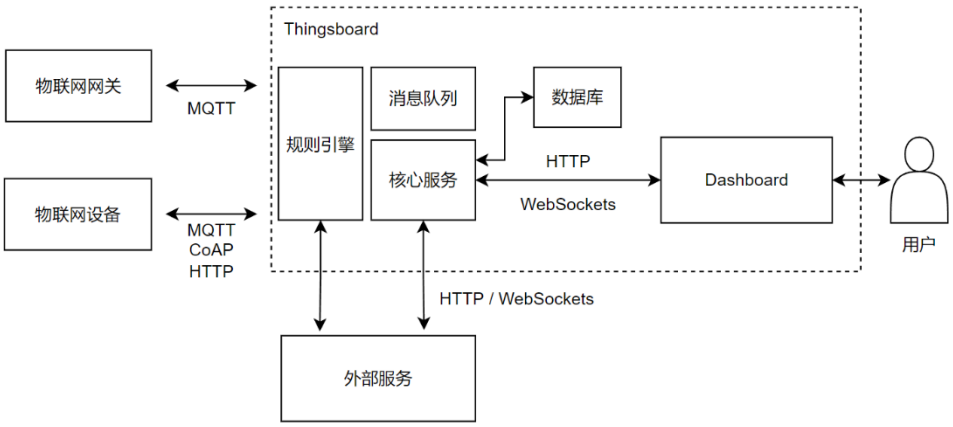

图 2.2 ThingsBoard 物联网平台

Thingsboard 支持 MQTT[17]协议接入物联网网关，以及 MQTT、HTTP、CoAP[18]三种协议接入物联网设备。当设备和网关接入平台后，规则引擎将对上传的数据进行规则匹配，数据会依据匹配成功的规则被路由到指定的消息队列或者执行规则中指明的动作。Dashboard 是 Thingsboard 的可视化用户界面集合，其与核心服务使用 Websocket[19]协议保持长链接，以便将数据推送至 Dashboard。规则配置使用基于 HTTP 的服务端 API。同时规则引擎与核心服务通过 HTTP 与 Websocket 协议连接至外部系统，提供更多样化的服务。

Thingsboard 的规则引擎以链式结构实现，为设备配置的完整规则称为规则链。规

则链主要由三种规则节点组成，分别是用于过滤和路由消息属性的 Filter 节点、用于更新入栈消息元数据的 Enrichment 节点、用于根据入栈消息匹配结果执行各种动作的 Action 节点。三种节点的功能与 RETE 算法的 Rules、Facts、Action 类似。ThingsBoard 将完整的规则拆分为规则节点，并将这些节点组合成规则链，以此实现灵活的规则配置与部署方案。

ThingsBoard 的规则引擎支持用户自定义规则并上传，且在 Dashboard 中实现了图形化的规则配置界面。在规则配置页面中，用户为单个设备分配处理规则，从数据的输入节点开始，配置后续的分流处理逻辑，根据规则与数据匹配的结果执行不同的动作。针对每个规则节点可以指明处理成功、失败、超时三种情况的下一步执行动作。

2.3　DSL 理论

规则引擎属于一种 IF-TEHN 形式的 DSL（Domain Specific Languages，领域特定语言）。本节介绍 DSL 的相关理论，旨在参考 DSL 的设计与实现理论，结合物联网消息处理系统的应用场景，为规则描述语言与规则解释函数的设计提供帮助。

2.3.1　DSL 的定义与分类

DSL 是针对某一特定领域，具有受限表达性的一种计算机程序设计语言[20]21。DSL 的作用范围是一个明确的小领域，解决特定的问题，其语法相比于通用的编程语言，例如 Java、Python、C/C++等，要更加简单直观易于理解。通用编程语言的语法在设计上更加偏向计算机的工作逻辑，使用者需要有一定的计算机专业知识，才能准确理解语义。而 DSL 的语法在设计上更偏向于贴近自然语言的逻辑，隐藏了计算机的结构特性，以便于非计算机专业人员也能轻松学习和使用。

规则引擎的规则描述语言属于 DSL 的一个子集，设计规则引擎就是在设计一种 IF-THEN 形式的 DSL，以及支持其运行的解释器。

DSL 主要分为三类：外部 DSL、内部 DSL 与语言工作台，语言工作台是一种专用的 IDE，用于定义和构建 DSL[20]21-22，语言工作台与本文相关性较低，因此下文主要讨论外部 DSL 与内部 DSL。

外部 DSL 拥有独立于通用编程语言之外的语法，如针对数据库交互设计的 SQL 语言就是一种外部 DSL[20]21-22。外部 DSL 的特点在于语法简洁、需要更加完善的解释器来支持其工作、对用户使用更加友好，但适用领域相对比较受限。外部 DSL 的典型例子除了 SQL 之外，还包括：XML，正则表达式，JSON，YAML 等。

相比外部 DSL，内部 DSL 可实现的逻辑功能要更加完备。内部 DSL 是在通用编

程语言的基础上实现的，其语法中继承了一部分通用编程语言的语法，并在此基础上定义了自己的语言模式。内部 DSL 可以看作是通用编程语言的特定用法[20]22。由于与通用编程语言绑定，内部 DSL 的学习成本通常高于外部 DSL，但功能性也相应地更强，与此同时，其有效作用范围通常被限制在使用某个特定的通用编程语言实现的系统内。基于 Java 语言的 Drools 规则引擎和基于 Go 语言的 gengine[21]规则引擎都是以内部 DSL 的方式实现的规则引擎。

外部 DSL 的优势在于语法简洁明了，易于学习，且在使用时与各种编程语言实现的系统都具有兼容性，其劣势在于功能较为单一。内部 DSL 的优势在于功能更加完备，但同时也带来了语法更加复杂且兼容性受限的缺陷。

2.3.2 DSL 解释器

（1）词法分析

词法分析作为解析 DSL 的第一步，负责对文本进行扫描，对其中的语句进行逐一解释。在词法分析阶段，输入到解析器中的文本会被转化为一种特殊的数据结构，通常被称为 token，其包含两个基本属性：类型与值。词法分析器将整个 DSL 文件转化为一串 token 序列，供之后的语法分析使用。

在 DSL 的设计中，语法只需要满足应用领域的需求即可。在物联网消息处理场景中，规则语法仅需要支持用户指明规则作用于哪些设备，以及消息数据的处理过程和匹配成功之后要执行的动作。物联网消息处理场景的 DSL 语法结构如图 2.3 所示。

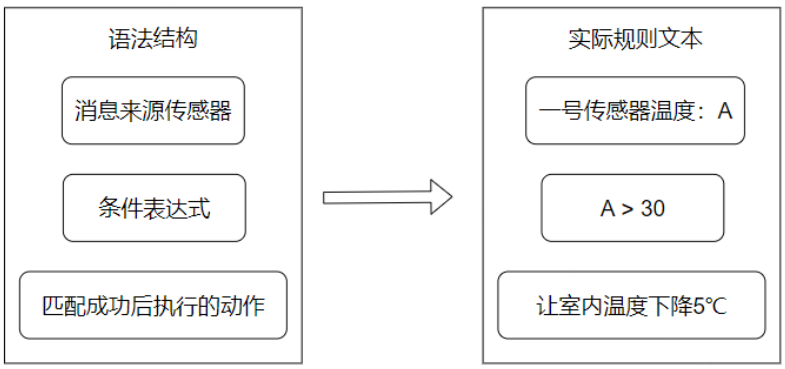

图 2.3 语法结构

以条件表达式$((A-C)>3)\&(B>20)$为例，词法分析将$-$、$>$、$\&$ 解析为操作符类型的 token，将 A、B、C 解析为变量类型的 token，将 3、20 解析为数值类型的 token，将左右括号将解析为括号类型的 token。词法分析结果如图 2.4 所示。

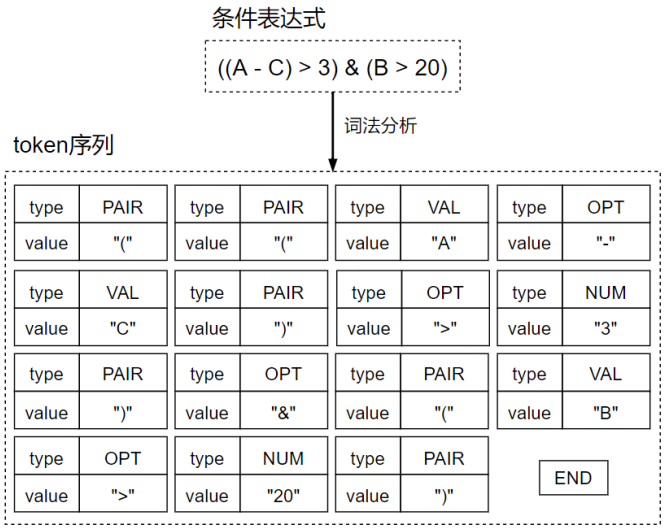

图 2.4 词法分析结果

（2）语法分析

当解析外部 DSL 时，要将一串文本分解成某种结构，通过这种结构来理解文本的含义，这个“结构化”的过程称为语法分析[20]72。语法分析是解析 DSL 的第二步。

规则解释器先对条件表达式使用词法分析，获得 token 序列，此时的 token 序列中将包含变量类型、数值类型、操作符类型、括号类型的 token。对于这种 token 序列来讲，其语法分析过程类同于字符串表达式的中缀表示法转后缀表示法[22]的过程，模式匹配的过程与对后缀表示法的表达式进行求值的过程类似。

表达式中缀表示法的形式为：对于二元运算，操作符位于左右运算数中间；对于一元运算，操作符位于运算数左侧。后缀表示法的形式为：对于二元运算，操作符位于最右侧，其后紧跟着左操作数与右操作数；对于一元运算，操作符位于运算数右侧。中缀表示法的形式符合数学中对表达式的一般描述方式，易于常人理解。以中缀表达式$(A-C)>3$为例，其对应的后缀表达式为$AC-3>$。由于后缀表达式的求值是顺序进行的，操作符出现的顺序就代表了执行的优先级，因此括号的存在是多余的，在中缀表示法转后缀表示法的过程中，会直接去除括号。

后缀表示法配合 stack 这种数据结构，可以轻松完成表达式求值，更易于计算机运算。大致的求值算法为遍历后缀表示法的表达式，如果遇到数值类型的元素，则直接压入 stack 中，如果遇到操作符类型元素，则弹出 stack 顶部的元素并结合操作符计算，

然后将计算结果再次压入 stack 中，最后留在 stack 中的值即为后缀表达式的运算结果。

常见的规则解释器的语法分析方案有两种，其中一种是基于 AST（Abstract Syntax Tree, 抽象语法树）的方案，AST 是可以描述表达式逻辑的树形结构，AST 将表达式的操作符放在中间节点，将操作数放在叶子节点，只要后续遍历 AST 即可得到表达式的后缀表示法。基于 AST 的语法分析思路是先依据表达式的中缀表示法构造 AST，然后后序遍历 AST，得到后缀表示法。中缀表达式$(A-C)>3$对应的 AST 如图 2.5 所示。构造 AST 作为中间过程的语法分析方案在 DSL 的最终目的为生成代码的情况下很有效，因为 AST 天生就具有语法表达能力[20]126-127。

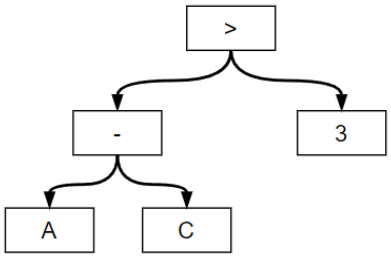

图 2.5 示例中缀表达式对应的 AST

规则解释器的另一种常见的语法分析方案是使用直接将中缀表达式转化为后缀表达式的 Shunting Yard 算法[23]，此算法由荷兰计算机科学家 Edsger Wybe Dijkstra 引入。基于 AST 的语法分析方案将语法分析分解为两步执行，而 Shunting Yard 算法将语法分析压缩在一个算法内，算法逻辑更加复杂但通常执行效率高于前者。后者在 DSL 的语法结构较为简单，且最终目的不是生成代码的情况下比较适用。

（3）符号表

符号表是用来建立 DSL 中表示对象的符号与符号对应的实际对象之间映射关系的一种数据结构，最常使用 map 来实现，并以字符串类型的符号作为 key[20]129-131。符号表的 value 可以是最终的对象，也可以是最终对象的某种生成器，两种方法各有优劣。使用符号表可以大幅降低规则解释器的复杂度。

2.3.3 语义模型

语义模型为实现 DSL 的最终目的，它是 DSL 的输出所生成的。若 DSL 的最终目的是生成某种代码，语义模型就是能表述目标代码语法逻辑的结构。例如 Makefile 的语义模型能转化为编译器指令构成的序列，CMake 的语义模型为能转换为 Makefile 的某种结构，HTML 的语义模型为 DOM 等。

在物联网消息处理系统中，规则语言这种 DSL 的实际输出就是实际要调用的动作

函数，因此消息处理系统中规则引擎的语义模型是能指明模式匹配成功后执行哪个动作的某种结构。在 DSL 的语义模型构建理论中，有三种比较常见的构建策略[20]125-127。

策略一是将语义模型的方法调用放入 DSL 解析算法，在解析过程中生成语义模型。策略一在物联网消息处理场景中的具体形式为将动作函数的方法调用直接放入解析算法，这样的策略会导致在开发后期时动作函数的扩展和维护难度上升，因为动作函数与解析算法是紧耦合的。策略一如图 2.6 所示。

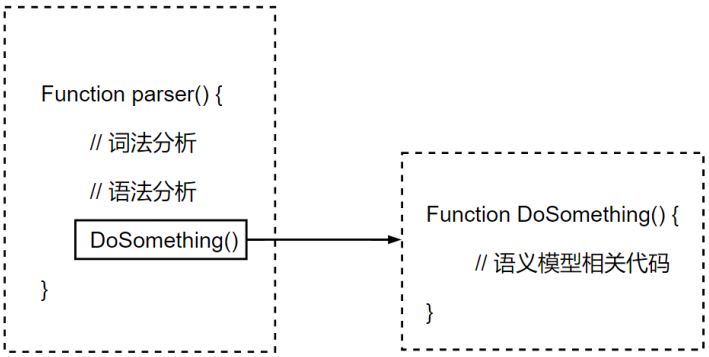

图 2.6 将语义模型的方法调用放入解析算法的策略

策略二是先构建 AST，再遍历 AST 生成语义模型。策略二在最终目的为生成代码的 DSL 中很实用，因为 AST 天生就具有表达复杂逻辑的功能[20]126-127。但在物联网消息处理场景中不需要生成代码，因此生成 AST 的中间过程不是必须的。策略二如图 2.7 所示。

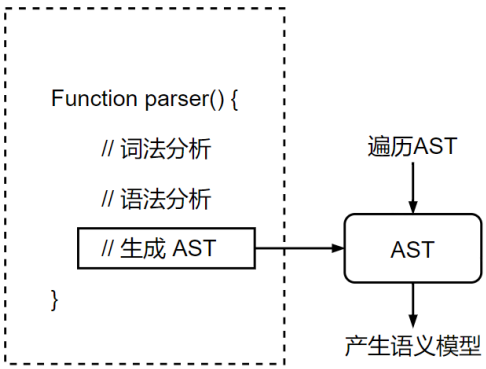

图 2.7 基于中间语法树生成语义模型的策略

策略三是在规则的解析过程中执行解释，并直接输出结果，不直接生成语义模型。策略三虽然不生成具体的语义模型，但可以将输出结果与分支之间建立映射关系，并实现一个专门的语义模型生成器，由解析过程产生的输出结果告知生成器应该生成哪一种语义模型。策略三如图 2.8 所示。

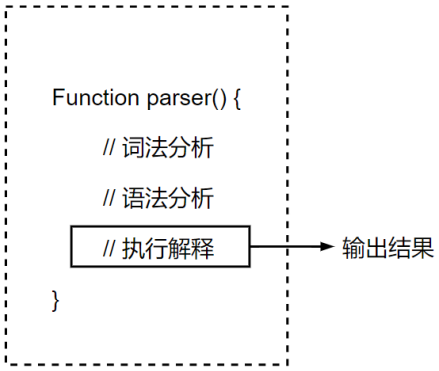

图 2.8 不直接生成语义模型的策略

2.4 Go 语言的并发机制

本节介绍 Go 语言[24]并发相关的机制。Go 语言是一种应用于高并发场景下的服务端语言。由于 Go 语言具有高效的并发模型与便捷的并发通信工具，因此非常适合用于开发物联网消息处理系统。

2.4.1 协程并发模型

Go 语言实现了一套高级语言层面的并发模型，称为协程（goroutine）。协程是比线程更轻量级的并发模型，切换成本低于线程。协程的实际工作是依赖于线程的，其实现基于一种名为 GMP 的模型，其中 G 代表 goroutine 协程，M 代表作为实际执行者的操作系统线程，P 代表单个 M 的调度器。G 与 M 是多对多的映射关系，一个 M 上可以通过 P 执行多个 G，同时当某个 M 变为阻塞态后，其上运行的 G 可以随 P 转到其他的 M 上继续运行。当某个 OS 级别的线程被阻塞，其对应的调度器 P 将转到另一个 OS 级别的线程中继续运行，只留下导致原线程阻塞的 G 继续等待系统调用执行完毕。

协程的调度比操作系统中线程的调度资源开销要低得多，因为它不会导致用户态向内核态的切换，协程调度的触发条件基于 Go 语言自身定义的某些语法机制，例如触发 time.Sleep()，或协程调用 Channel 陷入阻塞，与线程调度不同的是此调度机制不基于硬件定时器。理论上来讲，所有的协程可以运行在同一个线程中。

2.4.2　管道通信机制

管道（Channel）是 Go 语言中支持的一种协程通信机制。协程既可以监听管道，获取最新消息，也可以向管道中发送一条消息。管道机制很适合用在物联网消息处理系统中进行消息监听与传输。

Go Channel 在创建时需要指明可发送数据的类型，并使用 Go 语言的内置函数 make 来创建实际对象。make 函数将对象生成到公共内存区，并返回一个指向该对象的引用。Go 语言提供的管道分为带缓存的管道与不带缓存的管道两种，两种管道都是同步式管道。向不带缓存的管道中发送一个消息，将使得发送者阻塞，直到另一个协程从管道中取走消息；带缓存的管道维护了一个缓冲区，存放所有的消息，缓冲区大小在创建管道时指明，当发送者向带缓存的管道中发送一条消息时，若缓冲区满，则发送者阻塞，直到缓冲区中的某条消息被取走；当接收者企图取走一条消息时，若缓冲区为空，它同样会被阻塞，直到有新的消息到达。同时管道还分为单向管道和双向管道两种。

2.4.3　定时任务框架

定时任务框架可以以特定的周期执行特定的函数，使用定时任务框架来调度规则解析生成的模式匹配函数是一个不错的方案。在 Go 语言中，cron[25]是用于管理定时任务的库，cron.Cron 类是定时任务的实现类，它在实例化时接收两个参数，一个代表定时任务的执行时间间隔，另一个代表需要执行的函数对象，之后通过调用 Cron.Start 方法和 Cron.Stop 方法即可开始定时任务与停止定时任务。

第三章 消息处理系统的框架

本章设计了基于规则的物联网消息处理系统的框架与关键组件。系统的关键组件是消息接收器、规则调度器、动作执行器和异常状态管理器，它们相辅相成，共同完成物联网设备消息的监听与处理。

3.1 系统框架

本文设计了一种高度灵活可扩展的物联网消息处理系统框架，同时设计了一种特定的语言来部署消息处理规则，称为规则描述语言。基于规则的物联网消息处理系统框架如图 3.1 所示。

为了统一异构设备消息的处理方式，系统需要一个组件来同时接收所有异构消息，并将它们转变为统一的可访问格式，实现这一功能的组件是消息接收器。消息接收器同时连接多个异构网络，将不同设备发来的消息进行格式转换，并汇总到消息数据库中，为系统其他组件提供了统一的数据获取方式。

消息处理规则由设备消息的验证条件与后续的业务逻辑组成。设备的类型不同，应用场景不同，导致消息处理规则的编写工作是琐碎繁杂的。如果将消息处理规则硬编码到系统中，将导致系统的维护难度过大。为了将消息处理规则与系统实现解耦，系统将执行设备消息验证的单元与执行业务逻辑的单元进行分离。

规则的运行时结构是执行设备消息验证的单元，本文参考 Drools 规则引擎中 Working Memory 的设计，将规则的运行时结构设计为两部分，分别是数据源管理器和模式匹配器。系统中部署的每一则消息处理规则都对应于一个模式匹配器，所有的模式匹配器都从数据源管理器中获取待验证的消息，而数据源管理器的消息则来自于消息接收器。系统中所有需要执行的业务逻辑都由动作执行器执行，动作执行器受各模式匹配器的信号控制，当模式匹配器对某条消息验证通过时，就会给动作执行器发送信号，业务逻辑随后由动作执行器执行。

规则调度器是统一管理系统中所有模式匹配器的组件。规则调度器提供了部署规则的 HTTP 接口，规则的具体形式是一段遵循特定语法的文本，语法由规则描述语言规定。规则描述语言是本文设计的一种领域特定语言，专用于在本系统中定义消息处理规则。规则调度器接收规则描述语言制定的规则，将其维护在规则数据库，当需要部署某规则时，规则调度器调用规则解释函数解析规则，解释函数将生成模式匹配函数，模式匹配函数的作用是对数据源管理器中的指定数据集合进行验证，如果验证通过，

则向动作执行器发送一个信号，动作执行器随即执行业务逻辑。规则调度器将模式匹配函数交给一个子协程循环执行，此子协程即称作模式匹配器。规则调度器向物联网应用软件提供控制模式匹配器生存周期的接口。

消息处理系统是一个实时系统，有必要将紧急状态及时汇报给物联网应用软件。系统利用异常状态管理器来连接物联网应用软件，并将异常状态消息实时推送出去。本文参考 Thingsboard 平台中规则引擎与 Dashboard 的连接方式，提出了基于 WebSocket 协议的异常消息推送方案。

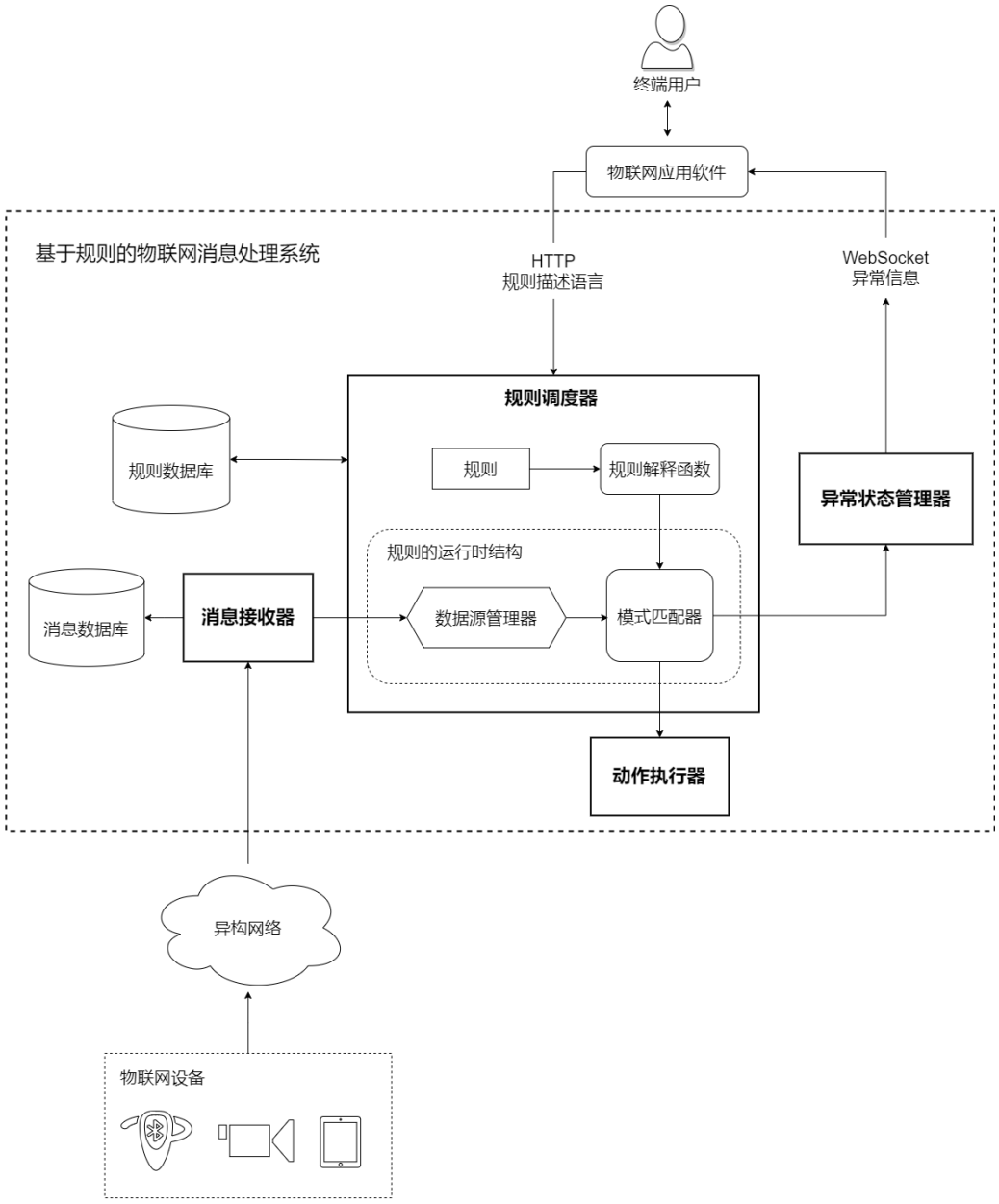

图 3.1 系统框架

下文将基于规则的物联网消息处理系统简称为消息处理系统。物联网设备经由异构网络接入消息处理系统。物联网应用软件与终端用户直接交互，并利用消息处理系统间接管理设备。消息处理系统的运行逻辑如下。

（1）物联网应用软件使用 HTTP 协议向规则调度器部署消息处理规则，指明实时监听的设备，以及规则的条件与业务逻辑动作。规则调度器将规则存入规则数据库，在部署时取出，并调用规则解释函数生成运行时结构。规则的运行时结构由数据源管理器和模式匹配器组成。

（2）消息接收器接收所有设备消息，将它们转变为结构化数据后存入消息数据库。同时，消息接收器将处于实时监听状态的设备消息发送到数据源管理器中等待模式匹配。

（3）当设备消息满足了某规则的条件时，模式匹配器将通知动作执行器执行相应的动作。

（4）物联网应用软件使用 WebSocket 协议连接到异常状态管理器。当模式匹配器检测到设备消息异常时，异常状态管理器将生成报警信息，并推送给物联网应用软件。

3.2　消息接收器

消息接收器是为了管理异构设备，统一设备消息的访问方式，并使得设备类型更加容易扩展而设计的组件。物联网设备的多样化程度高，设备消息的传输协议不同，消息体中包含的属性也不同。物联网设备上传的消息经过异构网络发送到消息接收器中，消息接收器根据设备类型，使用不同的解析程序将消息转化为统一格式。

3.2.1　设备消息规范

为了降低异构设备的管理难度，对物联网设备上传的消息提出格式规范：接入到物联网消息处理系统中的设备，其与消息处理系统交互所使用的消息体中至少应该包含设备唯一编号、会话次数这两个基本属性。

（1）对于传感器类型设备而言，其上传的消息体中应该包含传感器采集的环境信息，以及设备唯一编号和会话次数。会话次数在物联网设备采集到有效数据后产生，在消息从网关发出前整合到消息体中。

（2）对于控制类型设备而言，其与消息处理系统约定的命令格式中应该包含设备唯一编号与会话次数的属性。会话次数在消息处理系统下发指令时产生，消息处理系统会维护一个用于记录命令下发次数的全局变量。

消息处理系统为每类设备分配一个设备类型标识符，之后使用<设备类型标识符，设备唯一编号>二元组来表示某个设备。会话次数标识了消息产生的先后顺序，以防止

由网络拥挤造成的消息乱序到达。消息处理系统使用会话次数辨识最新消息，而不使用消息实际到达的系统时间。

3.2.2 消息接收器的工作逻辑

消息接收器为每类设备维护一个消息接收协程和一个消息类型标识符。消息接收协程的实现方式视设备类型集合而定，它们的作用是接收某类设备的全部消息，将其转为统一结构，并附上消息类型标识符，之后将结构化的消息存入消息数据库和数据源管理器。消息接收器的工作逻辑如图 3.2 所示。

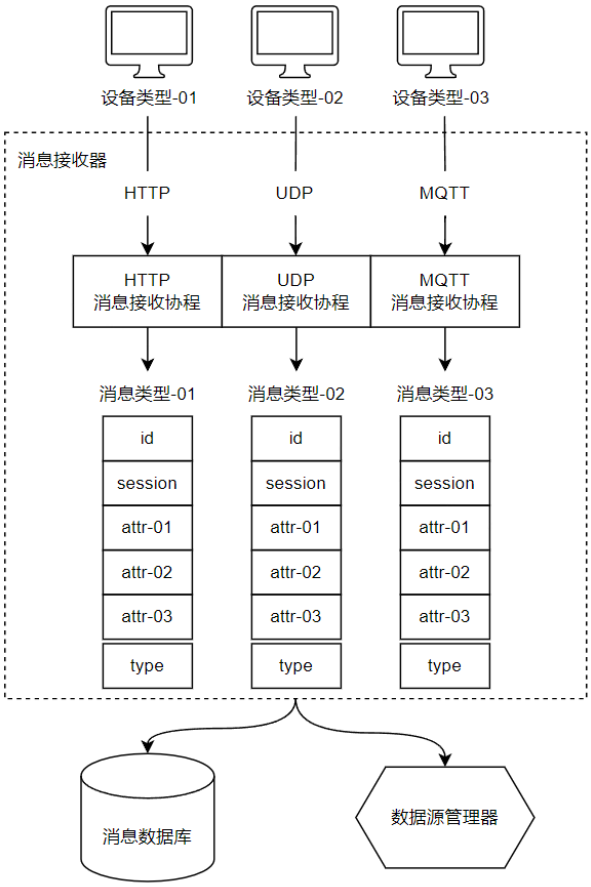

图 3.2 消息接收器的工作逻辑

（1）对接入到消息处理系统的每类设备维护消息接收协程，将经由异构网络到达的非结构化数据转化为结构化消息。设备的消息格式不会经常发生改变，因此消息解析程序以硬编码的方式实现。

（2）消息接收器解析出结构化消息后，为每条消息增加 type 属性，用以标识消息来源的设备类型。结构化消息由三个控制属性和多个非控制属性组成。控制属性中，id 代

表设备唯一编号，type 代表设备类型，session 代表会话次数。attr 表示非控制属性，即采集的环境数值，或者其他有效载荷。

（3）将结构化消息存入消息数据库的同时，存入数据源管理器等待进行模式匹配。

3.2.3　数据源管理器与消息数据库

（1）数据源管理器

数据源指的是<id，type，attr>三元组，id 是设备的唯一编号，type 是设备类型，attr 是消息属性。为了减少内存开销，数据源管理器维护了消息处理系统中某一时刻所有规则需求的数据源。使用引用计数法[26]管理数据源，引用计数标记了系统中需求数据源的规则数量，若引用计数减少为 0，则说明该数据源不再被系统需求，应当将其销毁。

数据源管理器是一个以<id，type>二元组作为 key 的 map，其 value 是另一种 map 类型的数据结构 key_attr。key_attr 的 key 集合是 attr 的子集，存放的数据类型是<ref_num，value>二元组，对应引用计数与最新数据。

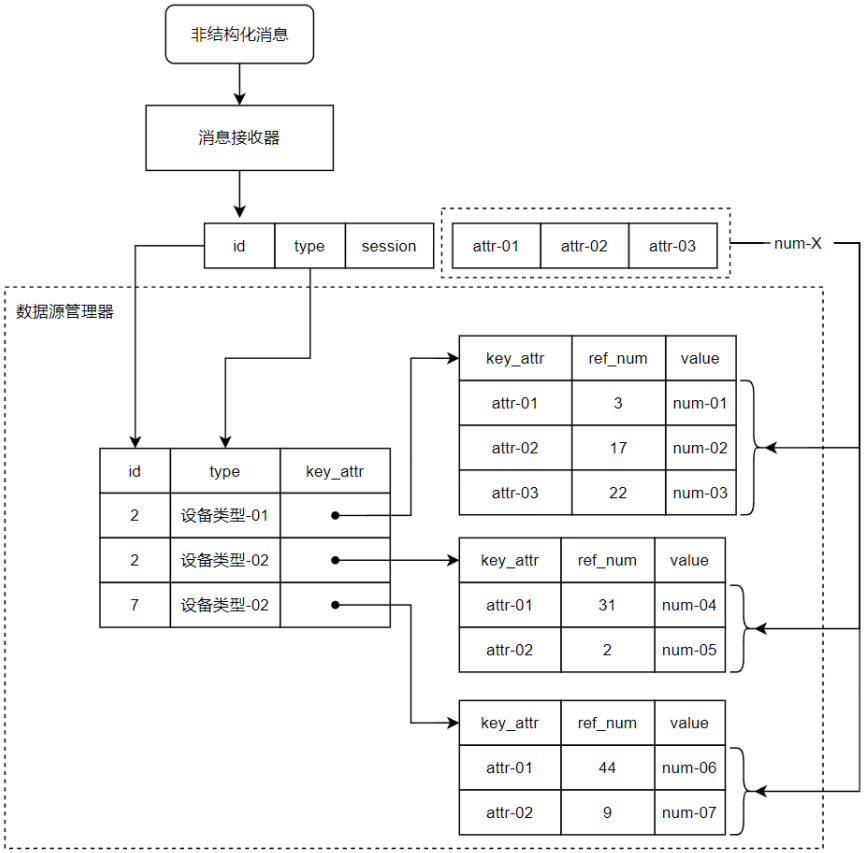

图 3.3 数据源管理器的更新

消息接收器对数据源管理器的更新流程如图 3.3 所示。

1、查找数据源管理器中是否存在以<id，type>作为 key 的元素。如果不存在则终止本次操作。如果存在则进入 2。

2、遍历以<id，type>作为 key 的 key_attr，用 attr 的最新数据更新其 value 属性。

在部署消息处理系统时，统计设备消息属性的数据类型，根据统计结果实现多种类型的数据源管理器，一般来讲，只要实现了整型、浮点型和字符串类型的数据源管理器，就可以满足绝大部分应用场景的需求。

（2）消息数据库

物联网设备产生的数据具有较高应用价值，数据变化的趋势反应了实际生产场景的运行状况，经过深层次的数据挖掘可以发现潜在问题。消息接收器将所有结构化消息统一存放到消息数据库，以便进行数据分析。

在消息数据库中，每类设备的消息单独存放在一张数据表，以 id 作为主键索引。结构化消息中只有 type 属性不会被保存。在每张数据表中使用 session 属性进行排序，体现消息到达的先后顺序。消息数据库的插入如图 3.4 所示。

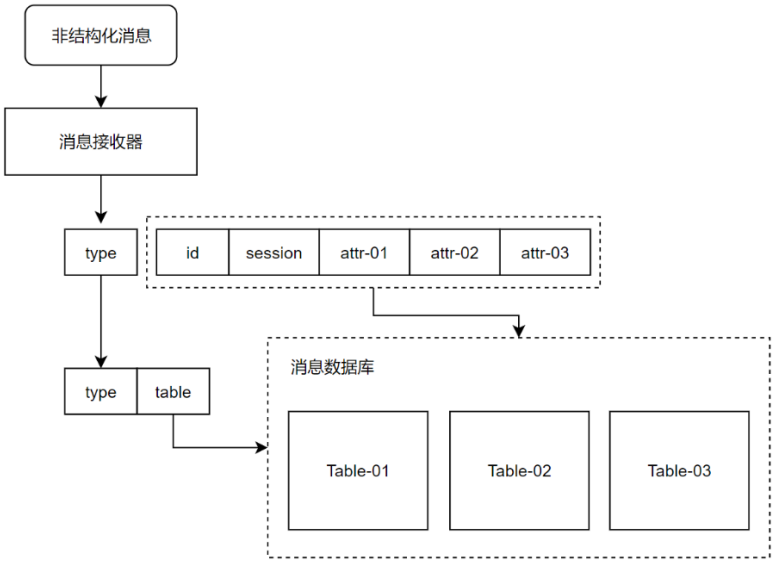

图 3.4 消息数据库的插入

3.3 规则调度器

规则调度器是为了控制消息处理规则的生命周期而设计的组件。规则调度器统一管理消息处理规则对应的模式匹配器，并且规则的开始、结束、存储、更新等都由规则调度器负责。

3.3.1　规则调度器的工作逻辑

规则调度器向物联网应用软件提供部署消息处理规则的 HTTP 接口。物联网应用软件在接口参数列表中写明需要部署的规则 ID 与规则的生命周期相关信息，指明规则的开始时间、结束时间。规则调度器为每则运行中的规则创建一个子协程，该子协程会循环运行由规则生成的模式匹配函数，因此该子协程称为模式匹配器。

系统使用 Go 语言的 Cron 定时任务框架来实现模式匹配器，因此传入参数中还需要包含模式匹配的执行周期，用于初始化 Cron。规则调度器使用两个 Go Timer 定时器控制 Cron 的开始与停止，亦即控制规则的生命周期。

规则在被解释前会先由预解析函数提取出该规则所需要的数据源集合，并解析出数据源集合在数据源管理器中的索引集合。在两个控制规则生命周期的 Timer 到时后，用数据源索引修改数据源管理器中相应单元的引用计数。

物联网应用软件在部署规则前需要先使用其他接口将规则存入规则数据库。之后使用规则数据库中规则的主键 ID 来访问规则。

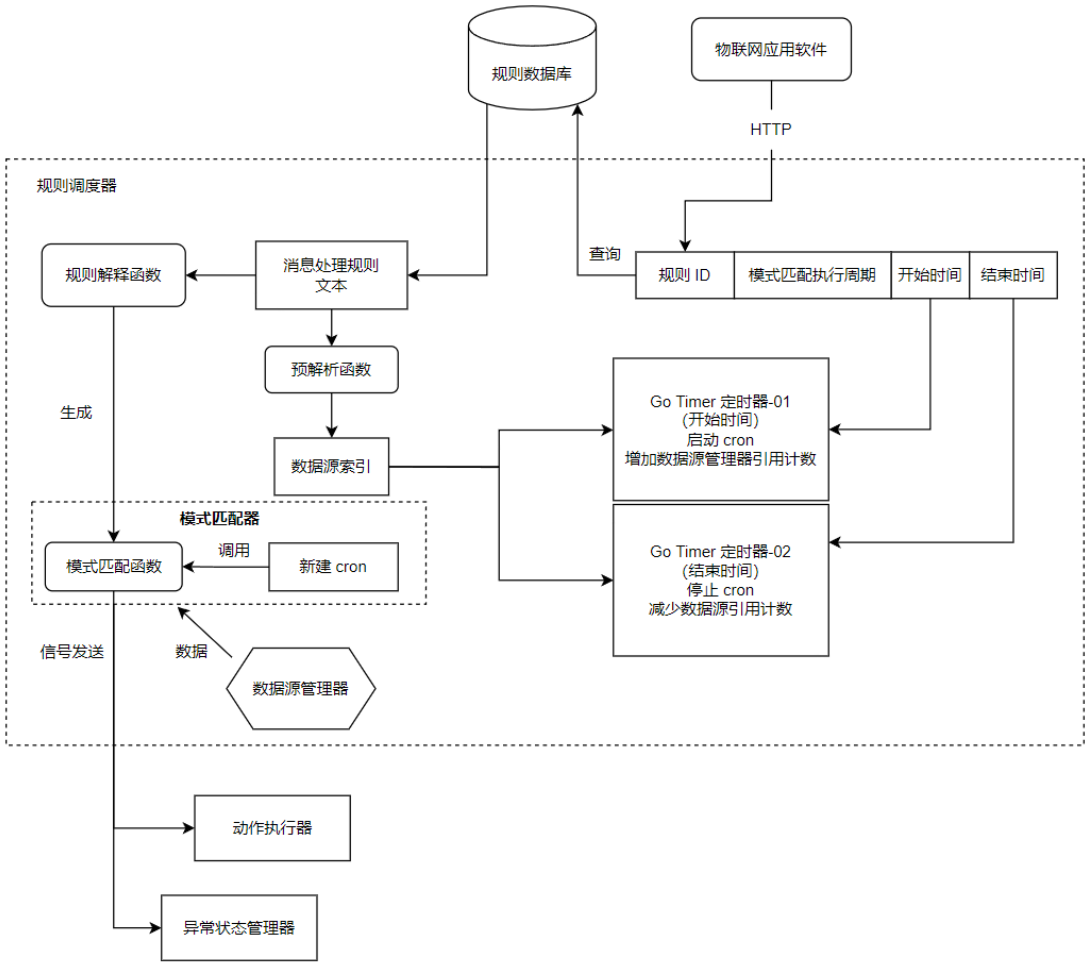

图 3.5　规则调度器的工作逻辑

规则调度器的工作逻辑如图 3.5 所示。

（1）物联网应用软件使用 HTTP 接口部署规则，除写明规则 ID 外，还需要指明规则的开始时间和结束时间，以及模式匹配的执行周期。

（2）规则调度器从规则数据库中取出规则，调用规则解释函数生成模式匹配函数，同时使用预解析函数获得规则所需求的数据源索引。

（3）规则调度器将模式匹配函数的调度工作交给 Go Cron 定时任务框架。定期执行模式匹配函数的 Cron 称为模式匹配器。

（4）规则调度器从请求的参数列表中提取出规则的开始时间和结束时间，用以初始化 Go 语言的定时器类型对象 Timer。Timer 在初始化时接收一个时间类型参数，表示其计时的时间量。

（5）规则开始时间的 Timer 到时后，启动模式匹配器，同时利用数据源索引增加数据源管理器中相应位置的引用计数。模式匹配器处于运行状态即代表规则处于部署状态。结束时间的 Timer 到时后，停止模式匹配器的运行，同时减少引用计数。

3.3.2 模式匹配器的调度

规则调度器利用规则解释函数将规则解析为模式匹配函数，并将其部署到 Go Cron 定时任务框架中，定期执行模式匹配。Cron 的启动时间、结束时间、执行间隔由物联网应用软件的传入参数指明。调度模式匹配器的过程如图 3.6 所示。

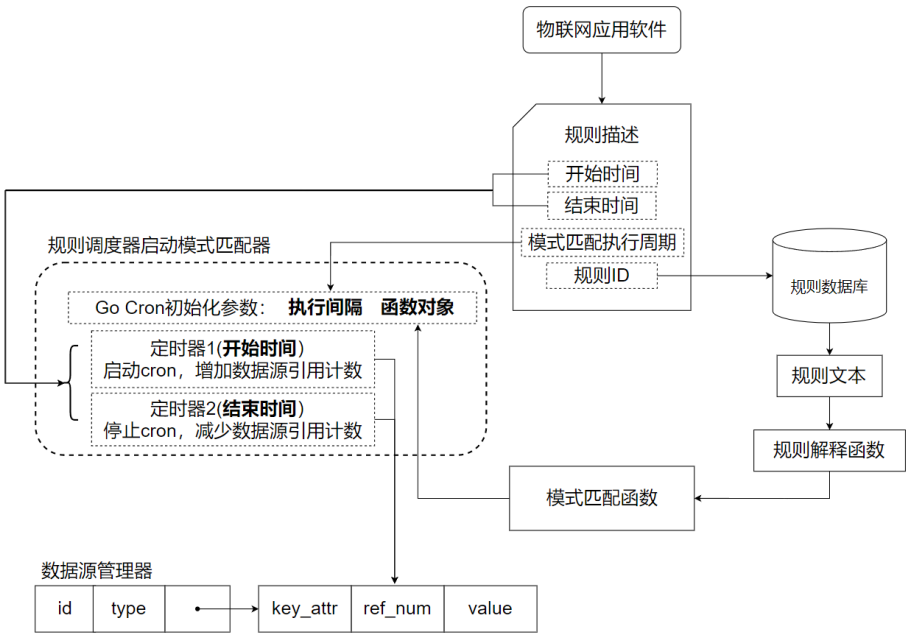

图 3.6 调度模式匹配器的过程

规则调度器在部署规则前预先解析规则文本，取出其中的数据源实体相关信息，用于维护数据源管理器中的引用计数。启动模式匹配器时，将该模式匹配器需求的所有数据源引用计数加一，停止模式匹配器时将引用计数减一。如果数据源不存在，则创建该数据源，引用计数初始化为一，值初始化为消息数据库中保存的最新数值。若某数据源的引用计数减为零，则销毁该数据源。更新数据源管理器时需要进行基本的并发控制，例如对数据源管理器加锁。

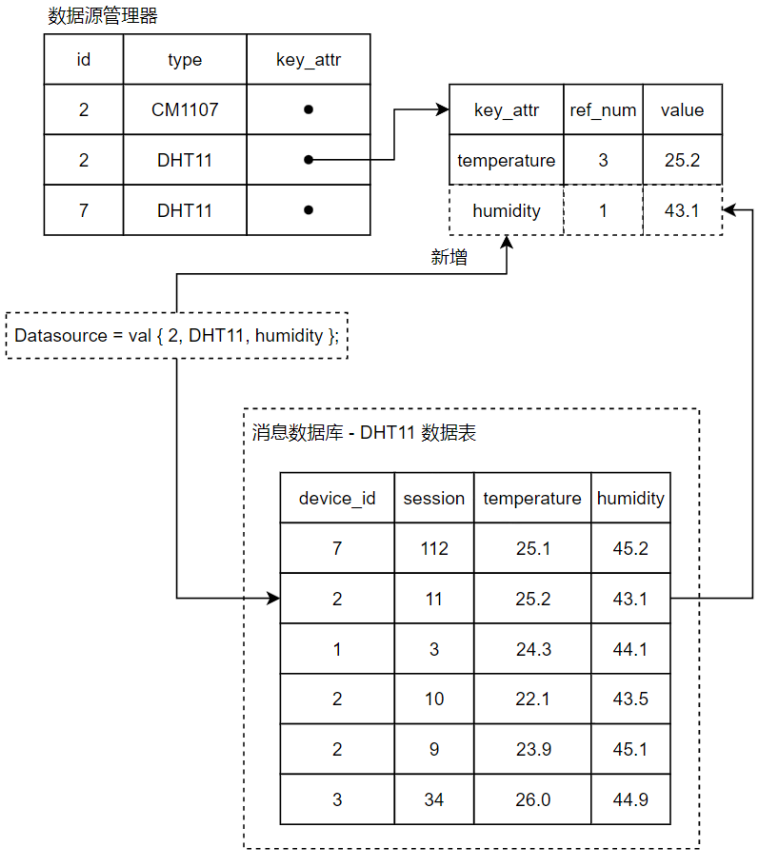

图 3.7 在开始时间对数据源管理器的更新

在规则开始时间对数据源管理器的更新过程如图 3.7 所示。规则结束时间的数据源更新过程与开始时间的过程类似，只不过减少 ref_num，当 ref_num 减少为 0 时释放数据源。

3.3.3 规则的访问权限

为了保证消息处理系统的平稳运行，以及维护数据的隐私性，消息处理系统为物联网设备消息的访问设置了权限控制。规则调度器访问某规则时同样会检查规则访问者的权限是否足够。

规则的访问权限受到规则中所需数据源的设备集合的访问权限限制。设规则中数据源的设备集合为 D，某用户具有访问权限的设备集合为 S，当且仅当 D∈S 时，该用户可以访问该规则。预解析函数会解析出数据源索引，规则调度器使用数据源索引来来检查用户权限是否充足。

设备访问控制的具体实现方式由应用场景的需求决定，在部署消息处理系统时，实现设备访问控制模型。

3.4 动作执行器

动作执行器是统一执行业务逻辑的组件。动作指的是当数据与用户规则匹配成功后，消息处理系统需要执行的一系列方法调用。动作可以是数据的转发，可以是向用户报警，也可以是对某个控制设备发送指令。而转发数据的协议也是多种多样的，这是因为消息处理系统常与外部系统连接在一起，并在整个物联网系统中承担消息路由的功能。

为了使消息处理系统具有充分的可扩展性，让代码易修改，可执行动作集合易扩充，本文在实现动作执行器时，将执行实际动作的工作交给了子执行器集合，动作执行器根据动作类型来调用子执行器。子执行器组成了消息处理系统的可执行动作集合，子执行器的实现依据实际需求的不同而不同，并且可以随时扩展和修改。

3.4.1 动作执行器的工作逻辑

动作执行器的职责为接受消息处理系统中各模式匹配器的命令，执行命令规定的动作。为保证动作执行的实时性，动作执行器以 Go Channel 的形式接收模式匹配器的命令。Go Channel 的发送方是各模式匹配器，接收方是动作执行器。下文将动作执行器使用的 Go Channel 称为动作执行信号管道。每一类动作的实际执行工作都由一个子执行器负责，子执行器可以是某个模块，也可以是某个函数，在这里不对它的实现方法做限制。动作执行器内部的每一根动作执行信号管道都最终连接到一个子执行器。

动作执行器的工作逻辑如图 3.8 所示。

（1）动作执行器为每类动作提供一个动作执行信号管道，其实际数据结构是带缓存的 Go Channel，传输数据类型为字符串。

（2）各模式匹配器在匹配成功时，将规则描述语句中写明的动作参数列表发送到相应的动作执行信号管道中。

（3）动作执行器使用 Go 语言的 select 机制同时监听所有动作执行信号管道。并在每一个 case 代码块中将参数列表字符串转发给相应的子执行器。每个子执行器具有两个

基本功能，其一是解析参数列表字符串，其二是执行实际动作。

动作执行器的实际动作执行单元是子执行器。子执行器的代码没有与动作执行器耦合，子执行器仅提供给动作执行器一个可调用的方法名。

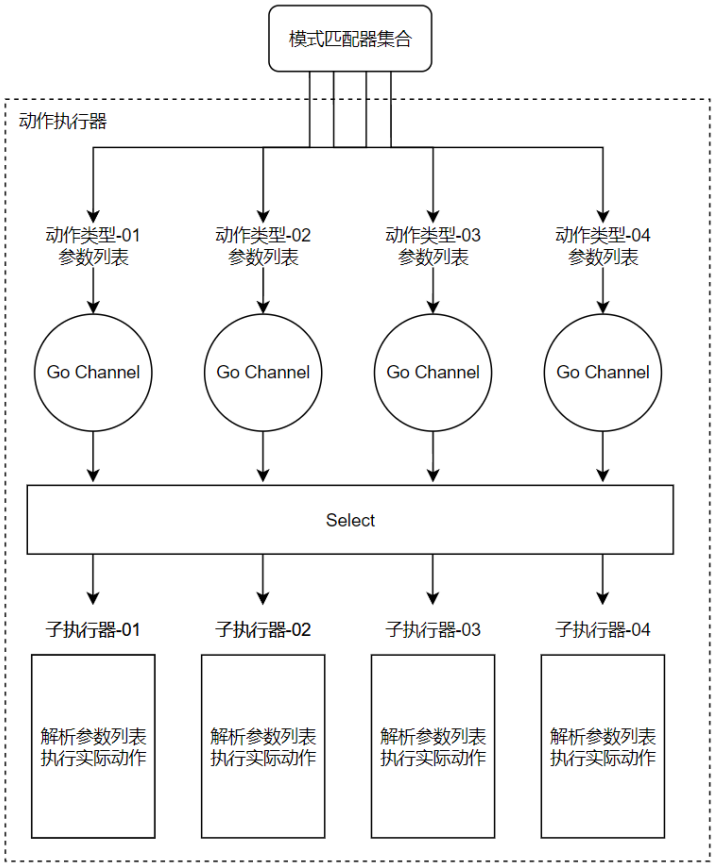

图 3.8 动作执行器的工作逻辑

3.4.2 动作参数列表的定义与解析

物联网应用软件在部署规则时，使用规则描述语言描述了规则匹配成功后需要执行的动作。描述执行动作的语句语法由子执行器定义并解析，动作执行器只负责监听与转发代表动作参数列表的字符串。

参数列表是实际动作执行函数运行所需求的。例如需要执行的动作是向某个服务器发送 HTTP 请求，那么服务器的 IP 地址、端口号，以及发送的有效载荷构成了参数列表。一类可执行动作对应着一个子执行器，所有子执行器维护自己的解析程序，并与物联网应用软件约定本类动作的参数列表格式。

如图 3.9 所示，子执行器定义了参数列表的结构，并将结构告知应用软件以达成约定，模式匹配器在条件匹配成功时将参数列表转发给动作执行器，子执行器解析程序解析参数列表并执行动作。

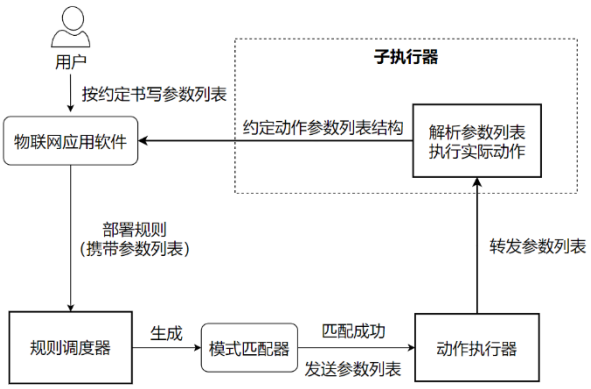

图 3.9 参数列表的定义与解析

3.4.3 子执行器的调用

物联网设备消息的监听与处理属于高并发场景，而执行实际动作的步骤通常是整个流程中比较耗时的部分。为了提高系统的响应速度，动作执行器使用协程池来执行实际动作。每当某动作执行信号管道的数据到达，协程池就派遣一个协程去帮助子执行器执行实际动作。

协程池类似于线程池，是一种支撑高并发应用的技术。协程池由一组工作协程和任务队列构成。每当协程池接收到一个任务后，就唤醒一个工作线程去执行该任务。协程池减少了协程创建与销毁的性能开销，同时限制了动作执行协程的数量。消息处理系统中用于执行动作的协程数量不会超出协程池的最大容量，超出部分的待执行任务将在协程池的队列中排队等候执行。

3.5 异常状态管理器

消息处理系统作为实时系统，存在着实时推送消息到客户端的需求。为了满足此需求，本文设计了异常状态管理器。

3.5.1 异常状态管理器的工作逻辑

物联网应用软件需要主动发起与异常状态管理器的 WebSocket 连接，才能接收异常状态的实时推送。WebSocket 协议是基于 TCP[27]协议的长连接协议，于 2008 年诞生，2011 年成为国际标准，起初是为了解决 HTTP 协议无法向客户端推送消息而产生的。WebSocket 协议与 HTTP 协议有着良好的兼容性，WebSocket 协议的握手阶段采用 HTTP，因此握手不容易被屏蔽。WebSocket 协议具有数据格式轻量、性能开销小，以及通信高效的特点，该协议既可以发送文本，也可以发送二进制数据。

如果要实现及时获取设备报警信息的功能，基于 HTTP 的方案只能是轮询，这既达不到足够的实时性，也增加了消息处理系统处理请求的负担。引入 WebSocket 协议后，消息处理系统可以主动向物联网应用软件推送信息。

异常状态管理器的工作逻辑如图 3.10 所示。下文将物联网应用软件简称为客户端，异常状态管理器简称为服务端，分别对应 WebSocket 协议中的客户端与服务端。

（1）服务端将所有 WebSocket 连接维护在一个全局的 map 中，以用户 ID 作为 key。该 map 称为连接管理器。

（2）连接管理器的 value 由两个 Go Channel 管道组成。其中一个是接收需要发送给客户端的消息的消息管道，另一个管道是退出信号监听管道。

（3）服务端的消息发送协程监听消息管道，每有消息到达，发送程序就将消息转发给连接对应的客户端，客户端部署的所有规则的处理情况与异常状态报警信息都将通过消息管道传送到 WebSocket 服务端消息发送协程。

（4）当客户端不再需要实时获取运行情况时，可以使用接口断开与服务端的连接，接口服务程序向退出信号监听管道发送一个信号，服务端主协程就断开对应的 WebSocket 连接，并将连接管理器中与该客户端对应的对象移除。

（5）断开连接后，不会对还未结束的模式匹配器产生影响，模式匹配器会在数据库中继续更新匹配结果，客户端可以通过其他接口获取规则的匹配情况。

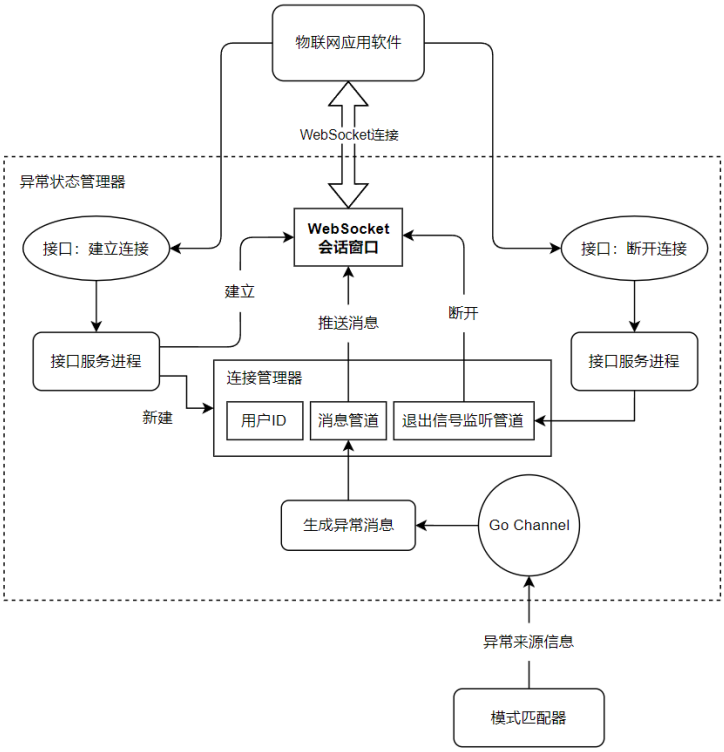

图 3.10 异常状态管理器的工作逻辑

连接管理器的作用域为整个消息处理系统，这意味着可以在任何需要的地方使用连接管理器维护的 WebSocket 连接，将需要推送给客户端的消息发送到该客户端对应的消息管道中即可。

3.5.2 异常消息体格式

为了使物联网应用软件能分辨异常发生的原因，每类异常消息需要定义自己的消息体格式，异常状态管理器不会对消息格式进行解析，任何进入消息管道的消息都将被原样转发到客户端。

在这里给出本文建议的异常消息体格式，以模式匹配器生成的异常消息体为例。模式匹配器产生的异常消息体具有固定格式，物联网应用软件应该具备解析这种格式的能力，将异常以更清晰的方式呈现给用户。

消息处理系统的消息处理规则分为两种，表达式规则以自定义表达式方式描述规则匹配条件，函数规则以函数调用作为匹配条件。后者主要用于流程复杂且使用频繁的特殊匹配过程。

（1）异常消息体的首个字段代表消息类型标识，指明此消息来自于哪一类规则，所有表达式规则产生的模式匹配器共享一个消息类型标识，每一类函数规则独占一个消息类型标识。

（2）从第二个字段开始为模式匹配成功的设备组成的列表。每个列表内的设备类型相同，列表头部将标识设备类型。

模式匹配器的异常消息体格式如图 3.11 所示。

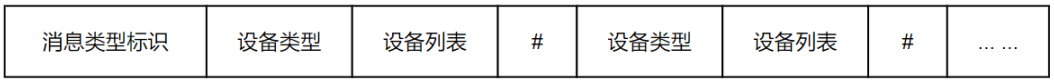

图 3.11 模式匹配器的异常消息体格式

3.6 本章小结

本章设计了基于规则的物联网消息处理系统的框架与关键组件。消息接收器负责统一接收物联网设备上传的消息；规则调度器向物联网应用软件提供部署规则的接口，同时控制规则的生命周期；动作执行器负责执行实际的业务逻辑；异常状态管理器负责将异常信息实时推送给物联网应用软件。

第四章 规则描述语言的设计与实现

本章设计了基于规则的物联网消息处理系统的规则描述语言，实现了规则解释函数，给出了规则解释函数生成模式匹配函数的过程。规则描述语言是制定物联网设备消息处理规则的语言。使用规则描述语言来定义消息在系统中各个组件之间的转发逻辑，部署自定义的消息处理规则。

4.1 语言设计与规则解释函数

设计规则描述语言与规则解释函数是实现自动化与智能化消息处理的关键步骤。规则解释函数以规则描述语言作为输入参数，输出模式匹配函数。模式匹配函数每被调用一次，就完成一次最新数据与规则的匹配。使用规则描述语言定义具体将哪些数据和怎样的规则进行匹配。

本文根据物联网消息处理场景的实际需求设计了规则描述语言 DCA。参考 DSL 的相关理论，实现了专用于解析 DCA 的规则解释函数。

4.1.1 规则描述语言 DCA 的设计

本文设计了基于规则的物联网消息处理系统的规则描述语言 DCA（Datasource，Condition，Action），由 DCA 描述的规则称为 DCA 消息处理规则或简称 DCA 规则。在设计规则描述语言时，需要考虑到该语言应当能够指明规则作用的设备消息实体，以及由条件和动作构成的规则实体。DCA 使用<Datasource，Condition，Action>三元组的形式定义数据源实体和处理规则。三元组中各元素的定义如下。

定义 4.1　Datasource 是数据源实体的集合。数据源实体由物联网设备的唯一标识号、设备类型标识符和关键属性组成。关键属性是消息体中需要实时监听的属性。

$$\begin{aligned}&\text{Datasource}\\&=\{name\{\text{id},\text{type},\text{attr}\}\}|\text{id}\in\text{Deivce},\text{type}\in\text{DeviceType},\text{attr}\in\text{DeviceAttribute}\}\end{aligned}$$

其中，name 是数据源的名字，id 表示设备的唯一编号，type 表示设备类型标识符，attr 表示关键属性。

定义 4.2　Condition 是数据源实体的模式匹配条件。匹配条件分为两类，一类是表达式条件，另一类是函数条件。表达式条件的定义形式是自定义表达式，函数条件的定义形式是类型标识符加参数列表。使用表达式条件的规则称为表达式规则，相应的有函数规则。

$$\text{Condition} = \text{F(d)}, \text{d} \in \text{Datasource}$$

其中，F 代表一个运算结果为布尔类型值的表达式或函数，d 是数据源实体集合的子集。

定义 4.3　Action 是触发 Condition 条件后执行的动作。

$$\text{Action} = \{(\text{type}, \text{P}(\$\text{d}))|\text{type} \in \text{ActionType}, \text{P} \in \text{ActionParams}, \text{d} \in \text{Datasource}\}$$

其中，type 表示动作类型标识符，P 表示动作执行器的参数列表，参数列表中可以使用$符号引用数据源实体。

（1）表达式规则的示例

对于表达式规则，规则描述语言 DCA 的示例如下。

$$\text{Datasource} = \text{val\_01}\{\text{id}_{01}, \text{deviceType}_{01}, \text{attribute}_{01}\}, \text{val\_02}\{\text{id}_{02}, \text{deviceType}_{02}, \text{attribute}_{02}\};$$

$$\text{Condition} = (\text{val\_01} + 10 > \text{val\_02})\&(\text{val\_02} < 30);$$

$$\text{Action} = \{\text{actionType\_01}, \text{P\_01}(\$\text{val}_{01}, \$\text{val\_02})\}, \{\text{actionType\_02}, \text{P\_02}(\$\text{val\_01}, \$\text{val\_02})\};$$

Datasource 字段中定义了两个数据源实体，分别命名为 val_01 与 val_02；DCA 语法允许使用大小写字母、数字和下划线的组合为数据源实体命名。Condition 字段制定了 val_01 和 val_02 的模式匹配条件。Action 字段规定模式匹配成功时执行两个动作，分别是 actionType_01 类型和 actionType_02 类型的动作，由于参数列表 P_01 和 P_02 以$符号引用了 val_01 和 val_02，因此模式匹配的实际数据将与参数列表一起传入动作执行器。

（2）函数规则的示例

对于函数规则，只有 Condition 语句与表达式规则有所不同，假设示例函数规则的 Datasource 语句和 Action 语句均与上例表达式规则相同，示例函数规则的 Condition 语句如下。

$$\text{Condition} = \text{type}, \text{val\_01}, \text{val\_02}, \text{const\_01}, \text{const\_02}, \ldots;$$

其中，type 是函数规则的类型标识符，之后是由数据源名与常数组成的参数列表。类型标识符指示使用哪个模式匹配函数，后续的参数列表会按序传入模式匹配函数，模式匹配函数会返回一个布尔类型的值，标识匹配是否成功。对于函数规则而言，模式匹配函数是在系统中预先编码好的，只需要用类型标识符选择即可。

DCA 语言将规则依据 Condition 的不同，分为表达式规则与函数规则。这样设计的原因在于，既要保证规则匹配条件的定义具有灵活性（表达式规则），又要保证 DCA 语言能够定义一些特殊的条件判断函数（函数规则），这些函数无法用单纯表达式描述，

但在特定的应用场景中却很常用。

4.1.2　规则解释函数的总体流程

规则解释函数专用于解析规则描述语言 DCA，生成模式匹配函数。模式匹配函数的一次调用执行的是实时数据与 DCA 规则的一次匹配过程。模式匹配函数由规则解释函数生成，并返回给规则解释函数的调用者——规则调度器。规则调度器随后会将模式匹配函数部署到一个持续运行的子协程中（本文使用 Go 语言实现系统，在非 Go 语言实现的系统中，这里是子线程）。部署了模式匹配函数的子协程就称作模式匹配器。

规则解释函数的工作逻辑如图 4.1 所示。

（1）规则调度器接收物联网应用软件的请求，在规则数据库中查询 DCA 规则，并调用规则解释函数解析 DCA。

（2）规则解释函数逐条解析 DCA 语句，解析过程如下:

1、使用 Datasource 字段生成符号表，符号表保存了数据源名字与数据源实体映射关系。符号表用数据源实体从数据源管理器中获取最新数据。

2、使用 Condition 字段生成代表匹配条件的对象。该对象为类型标识符和 TokenList，对于表达式规则需要分别执行词法分析与语法分析，对于函数规则只需要执行词法分析。

1、使用 Action 字段生成 ActionList 数据结构，ActionList 是由 Action 数据结构组成的列表。Action 数据结构是由动作类型标识符和动作参数列表组成的结构体。

2、规则解释函数将解析 DCA 语句生成的三组变量作为输入参数，调用模式匹配器生成函数，将模式匹配函数返回给规则调度器。

3、规则调度器将模式匹配函数部署到模式匹配器中，并控制模式匹配器的运行状态。

模式匹配器生成函数的命名是为了语言表达的简洁和直观。由于模式匹配器生成函数生成的是模式匹配函数，而不是模式匹配器，因此严格意义上来讲，应该叫做“模式匹配函数生成函数”，但如果叫做“模式匹配函数生成函数”，又难免会让人误解为模式匹配函数生成了某函数。因此取名为“模式匹配器生成函数”。

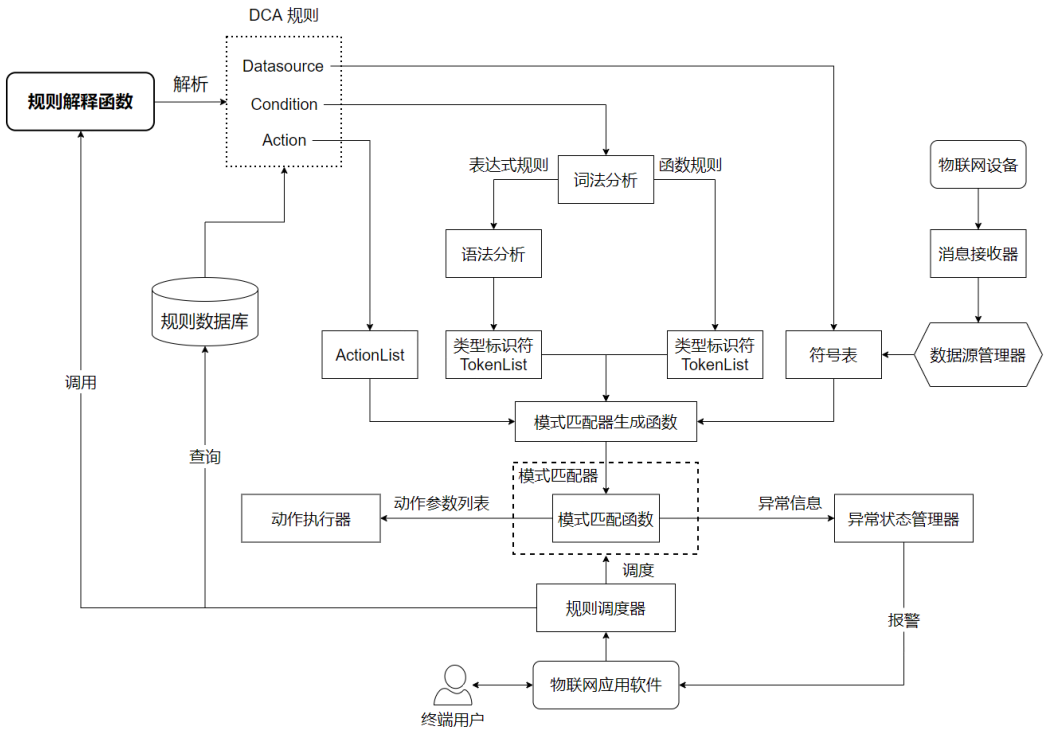

图 4.1 规则解释函数的工作逻辑

4.2 数据源实体的解析

Datasource 字段定义了数据源实体，并为数据源实体分配了 DCA 规则内可引用的名字符号。规则解释函数将 Datasource 字段解析为保存了数据源符号与数据源实体映射关系的符号表。

4.2.1 双符号表策略

本文参考 DSL 相关理论，在基于规则的消息处理系统中，使用符号表保存数据源实体的名字与其最新数据之间的映射关系[20]129-131。在 DCA 语言中，符号表是一个 key 为数据源实体名字的 map。模式匹配器使用符号表获取数据源实体的最新数据。每次模式匹配与动作执行时，使用符号表保存的数据替换 Condition 字段和 Action 字段中对数据源实体的引用。

本文使用两种符号表。一种是外符号表，它以数据源实体作为 value，用于在模式匹配器被调度时查询数据源管理器中的数据。另一种是内符号表，它以数据源实体的最新数值作为 value，用于在模式匹配过程中替换数据源实体的名字。

设计双符号表的原因在于物联网消息处理系统具有高实时性需求，应当既保证物

联网设备最新数据在进入系统时，以尽量快的速度传播到符号表进行模式匹配，同时保证在模式匹配的过程中，保存了最新数据集合的符号表不会被新数据覆盖。

每个规则都生成一张外符号表，所有外符号表共用数据源管理器。消息接收器将数据源管理器的数据维持在最新状态。通过外符号表将模式匹配器被调度时的数据源实体数据拷贝到内符号表，并使用内符号表进行模式匹配。模式匹配完成后，内符号表被释放。双符号表策略如图 4.2 所示。

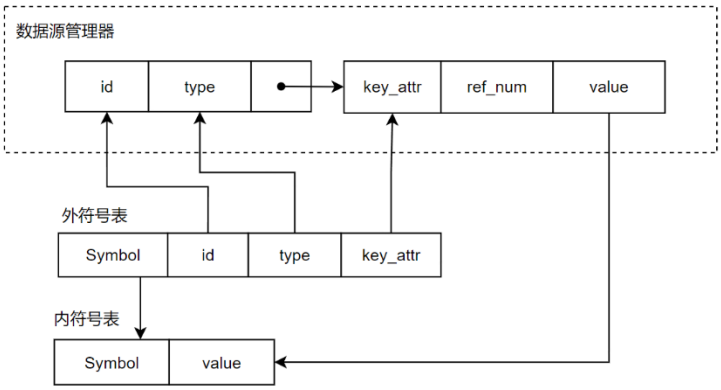

图 4.2 双符号表策略

4.2.2 外符号表生成算法

规则解释函数解析 Datasource 字段并生成外符号表。外符号表生成算法如下所示。

算法 4.1：外符号表生成算法

输入：代表 Datasource 字段的字符串 datasource。

输出：外符号表。

Start

1、初始化外符号表 symbolTable，symbolTable 是一个 map，key 为字符串类型，代表数据源的名字，value 为结构体类型，代表数据源实体在数据源管理器中的位置。

2、将 datasource 去除所有空格，并以逗号切分，得到数据源实体定义组成的列表 sourceList。

3、遍历 sourceList，设当前元素为 source。

4、将遍历到的元素以左括号切分列表，第一个元素是数据源实体的名字 name，第二个元素由设备唯一标识号 deviceId、设备类型标识符 type、关键属性 attribute 组成。

5、将 deviceId、type、attribute 作为 value 存入 symbolTable 中以 name 为 key 的存储单元中。若 sourceList 中还有元素，则跳转到 3。否则跳转到 6。

6、返回 symbolTable。

End

4.3 规则主体的解析

DCA 的条件语句 Condition 与动作语句 Action 构成了规则主体。在规则主体中，Condition 描述的匹配条件分为两类，一类是基于表达式的自定义匹配条件，称为函数匹配条件；另一类是基于类型标识符的函数匹配条件。Action 字段描述模式匹配成功后执行的动作。

4.3.1 表达式匹配条件的解析

表达式匹配条件由条件表达式构成，表达式中可以使用 Datasource 字段定义的数据源名字，条件表达式支持四则运算加（+）、减（-）、乘（*）、除（/），关系运算大于（>）、小于（<）、不等于（!=）、等于（==），以及逻辑运算与（&），条件表达式的执行结果必须为 true 或者 false，解释函数会对条件表达式做必要的语法检查。解析流程分为词法分析和语法分析两步，如图 4.3 所示。

首先，规则解释函数使用外符号表对 Condition 语句进行词法分析，生成 TokenList。TokenList 是由 Token 类型的对象组成的列表，数据结构 Token 包含 type 和 value 两个属性，属性类型均为字符串。type 指明 Token 的类型，value 指明 Token 中存放元素的具体值。类型共有四种，分别是数值 Token、运算符 Token、括号 Token、变量 Token。数值 Token 代表常量，变量 Token 代表 Datasource 字段中定义的某数据源。

之后，规则解释函数利用语法分析算法将 TokenList 转为后缀表示法，同时检查是否存在语法错误。

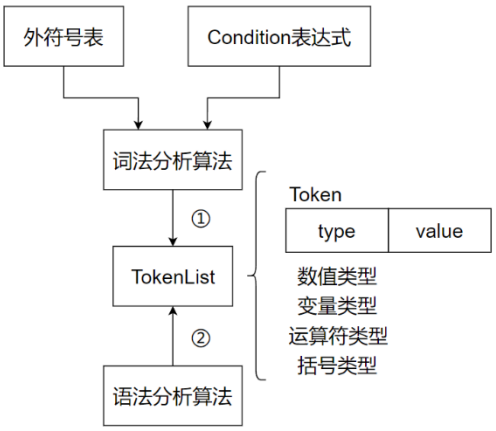

图 4.3 表达式匹配条件的解析流程

（1）表达式词法分析算法

词法分析会检查输入的条件字符串属于表达式条件还是函数条件，检测方法为检查

字符串中是否含有逗号，表达式匹配条件不含逗号，当确认输入的是表达式匹配条件时，词法分析函数自动生成一个用于指示此规则是表达式规则的类型标识符 type，并调用表达式词法分析算法解析表达式匹配条件。

表达式词法分析算法将条件表达式从字符串形式转化为具有具体含义的 TokenList，表达式词法分析算法如图 4.4 所示。

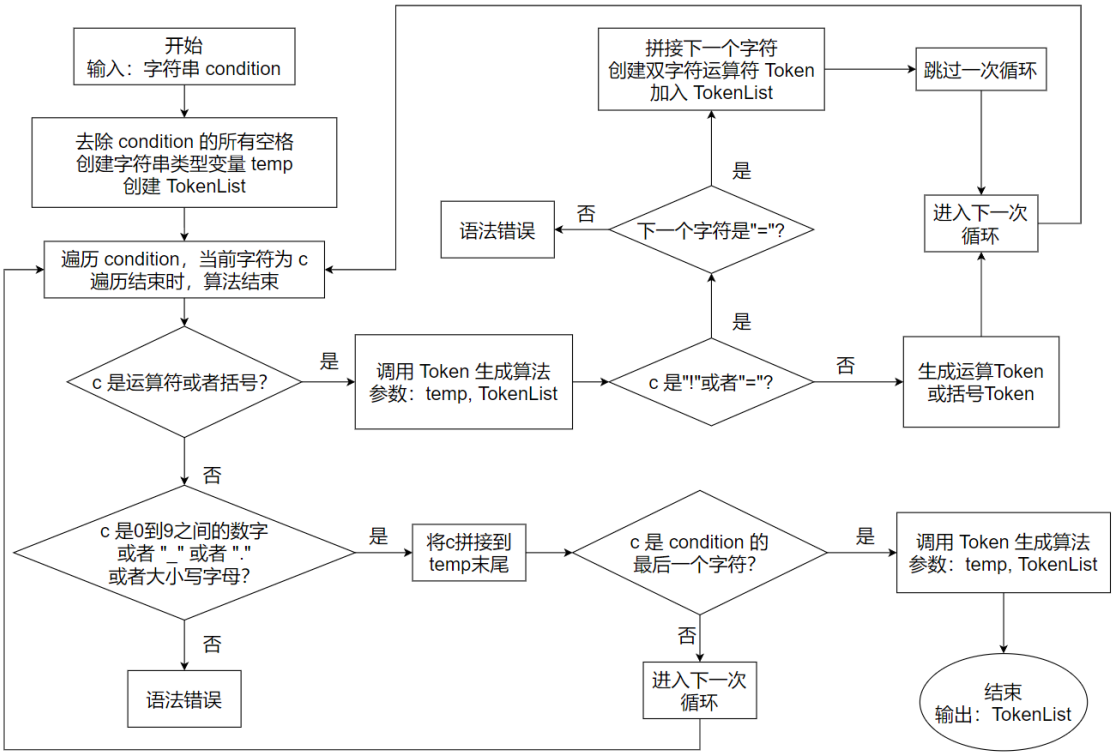

图 4.4 表达式词法分析算法

算法 4.2：表达式词法分析算法

输入：代表表达式匹配条件的字符串 condition。

输出：TokenList。

Start

1、去除 condition 中的所有空格，然后创建一个临时变量 temp，用于保存每次解析出的 Token 的 value 值，同时创建作为输出结果的 token 列表 TokenList。

2、遍历 condition，设遍历到的字符为 c。

3、若 c 是运算符或者括号，则调用 Token 生成算法，将 temp 和 TokenList 作为传入参数。同时判断 c 是否是!或者=。如果是，则跳转到 4，否则跳转到 5。

4、判断下一个字符是否是=，如果不是，则报错；如果是则使用 c 和下一个字符创建

一个运算符 Token，也加入到 TokenList 中，并跳过一次循环，然后跳转到 2。

5、使用 c 生成一个运算符 Token 或括号 Token，加入 TokenList，然后跳转到 2。

6、若 c 不是运算符或者括号，则判断 c 是否是数字或字母。如果不是，则出现语法错误；如果是，则跳转到 7。

7、将 c 拼接到 temp 的末尾，并判断 c 是否是 condition 的最后一个字符，如果是最后一个字符，则调用 Token 生成算法，生成最后一个 Token，并结束循环。否则将当前字符拼接到 temp 的最末尾，跳转到 2，进入下一次循环。

8、返回 TokenList

End

算法 4.3：Token 生成算法

输入：temp 的引用，TokenList 的引用。

输出：无。

Start

1、判断 temp 是否能转为数值类型，包括整数和浮点数。如果是，则跳转到 2，否则跳转到 3。

2、初始化数值类型 Token 对象 token，其 value 为 temp 转换的数值。

3、判断符号表中是否存在以 temp 为 key 的元素。如果是，则跳转到 4，否则抛出错误，算法结束。

4、初始化数值类型 Token 对象 token，其 value 为 temp。

5、将 token 加入 TokenList，清空 temp。

End

（2）语法分析算法

经过词法分析得到 TokenList 后，利用语法分析算法将 TokenList 由中缀表示法转为后缀表示法，检查错误语法并去除其中的括号 Token。本文参考 Shunting Yard 算法的思路，设计了语法分析算法。语法分析算法如图 4.5 所示。

算法 4.4：语法分析算法

输入：中缀表示法的表达式 TokenList。

输出：后缀表示法的表达式 TokenList。

Start

1、创建两个 stack，记为 optSt 和 postSt。

2、遍历 TokenList，设当前 Token 为 t。

3、如果 t 是一个数字 Token 或者变量 Token，则将 t 压入 postSt 中。

4、如果 t 是一个括号 Token，且其值是左括号，则压入 optSt。

5、如果 t 的值是右括号，则不断地将 optSt 中的顶部元素弹出，并压入 postSt 中，直到遇到左括号 Token，当遇到左括号 Token 后，将其与 t 一起舍弃，如果直到弹出到栈底，optSt 中仍不存在左括号，则算法返回语法错误。

6、如果 t 是一个运算符，则检查 optSt 的栈顶是否存在另一个运算符 Token，如果不存在，则将 t 压入 optSt；如果存在，则对比两个 token 的优先级，若 t 的优先级更高，则将 t 放入 postSt，否则将 optSt 栈顶 Token 弹出并放入 postSt，之后将 t 压入 optSt。

7、遍历结束时，依次将 optSt 的剩余 Token 弹出并压入 postSt。

8、最后将 postSt 的 Token 依次弹出，使用与弹出顺序相反的顺序构造一个新的 Token 序列，此 Token 序列即为要返回的后缀序列。

End

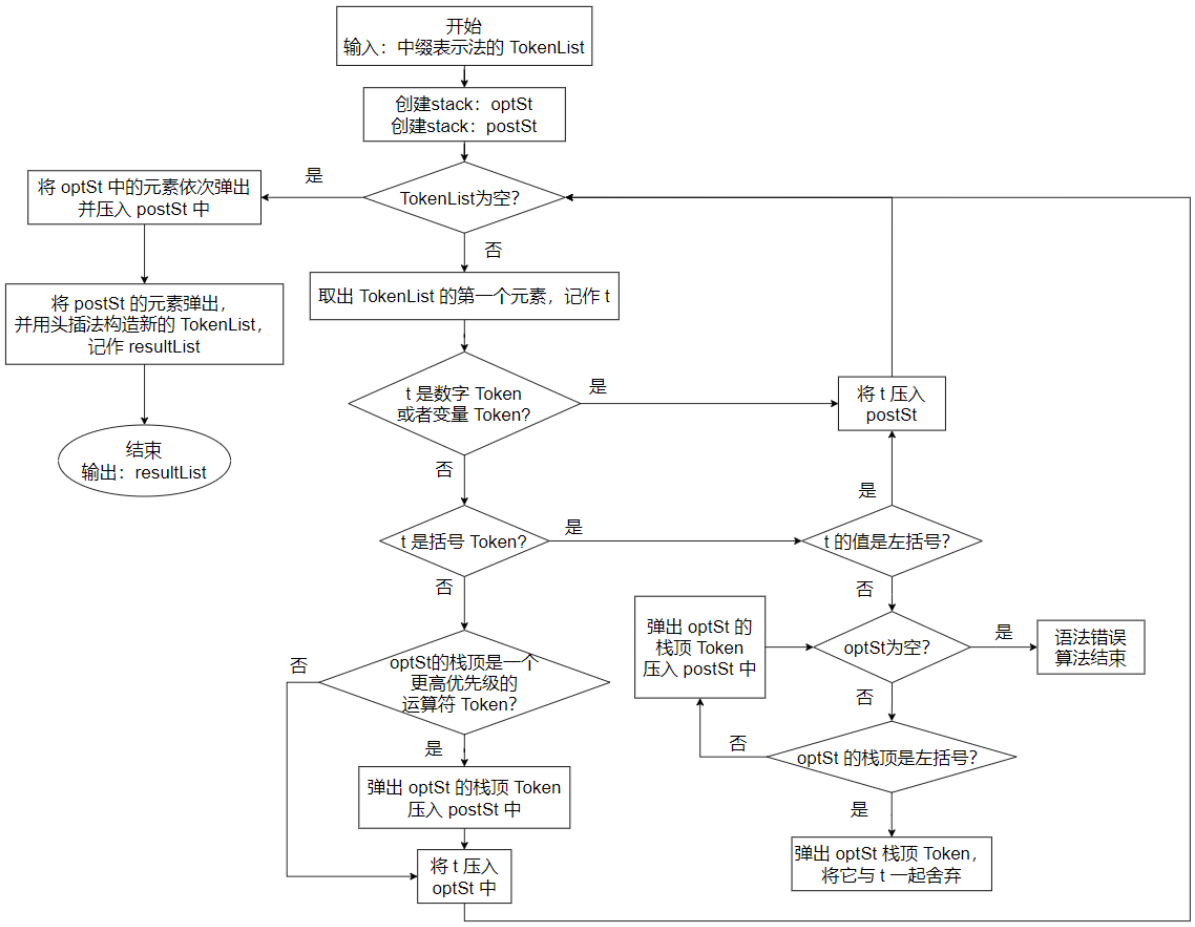

图 4.5 语法分析算法

表达式匹配条件的条件表达式支持的运算符及其优先级如表 4.1 所示，优先数越

小，优先级越大。

表 4.1 运算符优先级

运算符	优先数
*	1
/	1
+	2
-	2
>	3
<	3
!=	3
==	3
&	4

考虑到实际需求和开发成本，本文没有实现算术取余运算符%。同时，由于关系运算符>=、<=，以及逻辑或运算符（|）、逻辑非运算符（!），均可以由其他已实现的关系运算符和逻辑运算符组合实现，因此在本文中也没有实现以上运算符。

4.3.2 函数匹配条件的解析

函数匹配条件代表的是匹配流程无法用自定义表达式描述的过程，消息处理系统将这些过程硬编码为模式匹配算法。并将它们存放在一个 map 中，以类型标识符 type 作为索引。

函数匹配条件的解析过程只包含词法分析一步，且词法分析的过程比较简单。函数规则的词法分析算法检测到 Condition 语句中存在逗号时，就确定了这是一个函数匹配条件，这是因为表达式匹配条件中不存在逗号这一符号。词法分析以逗号作为分隔符，得到函数规则的类型标识符 type 和参数列表。利用外符号表，将参数列表转为 TokenList，函数匹配条件的解析过程到此结束。

4.3.3 动作语句的解析

Action 字段用于描述模式匹配成功后执行的动作，可以同时定义多个匹配成功的动作，动作执行没有先后顺序。描述一个动作时需要指明两个参数，分别为动作类型标识符和参数列表。动作类型标识符组成的集合与执行动作的子执行器集合形成单一映射关系。

规则解释函数将 Action 字段解析为 ActionList，ActionList 是由数据结构 Action 组

成的列表，Action 拥有两个字符串类型的属性。一个是 type，代表动作类型标识符；另一个是 params，代表动作的参数列表。

DCA 语法允许使用$符号在 params 中将某段字符串标记为数据源实体对象，只有当模式匹配成功时，使用内符号表的相应数值替换标记字符串，动作语句解析算法不会对 params 进行数值替换。

算法 4.5：动作语句解析算法

输入：代表 Action 字段的字符串 action。

输出：ActionList。

Start

1、创建 Action 类型的列表 ActionList。

2、将 action 以逗号切分，得到动作定义组成的列表 actions。

3、遍历 actions，设遍历到的元素为 a。

4、去除 a 的两端括号后，从头到尾扫描，以出现的首个逗号为切分点，切分 a，切分后的左侧元素为动作类型标识符 type，右侧元素为参数列表 params。

5、使用 type 和 params 初始化一个 Action 类型对象，加入到 ActionList 中。

6、遍历结束时返回 ActionList。

End

4.4　模式匹配函数的生成

规则解释函数在解析完所有 DCA 语句后调用模式匹配器生成算法，模式匹配器生成算法将外符号表、TokenList 与类型标识符、ActionList 组合成模式匹配函数。模式匹配函数生成后返回给规则调度器，自此规则解释函数结束。

4.4.1　模式匹配函数的逻辑

DCA 语句解析后产生了四个对象，分别是外符号表、TokenList、类型标识符，以及 ActionList。

外符号表用于生成包含本次匹配数据的内符号表，TokenList 与类型标识符共同组成模式匹配的条件，模式匹配函数使用类型标识符查询该类规则的模式匹配算法，并将 TokenList 和内符号表传入模式匹配算法中。

表达式规则的类型标识符由词法分析函数添加，函数规则的类型标识符在 Condition 语句中明文给出。表达式规则的匹配条件是表达式，因此表达式规则使用字符串表达式求值算法作为模式匹配算法。每种函数规则的模式匹配算法都不同，它们

共同满足应用场景的特定需求。

当模式匹配算法返回 true 时，模式匹配函数将 ActionList 中的每个 params 发往相应的动作执行信号管道。同时，将外符号表的内容转为模式匹配器的异常消息体格式，最后将异常消息通过异常状态管理器发送给物联网应用软件。

模式匹配函数的逻辑如图 4.6 所示。

（1）模式匹配函数用外符号表生成内符号表，并对内符号表进行模式匹配。

（2）模式匹配函数用 type 查找模式匹配算法，并将 TokenList 和内符号表送入模式匹配算法。

（3）模式匹配函数在匹配成功时，查询异常状态管理器中是否存在以部署规则的用户 ID 为 key 的消息管道，如果存在则利用外符号表相关信息和类型标识符生成异常消息体，发送到消息管道中。部署规则的用户 ID 在规则调度器调用规则解释函数时，作为输入参数传入。

（4）当模式匹配成功时，模式匹配函数将内符号表的元素转为字符串类型，用于替换动作参数列表中以$标记的字符串。动作执行器将动作类型标识符与子执行器的映射关系维护在执行器查找表中。模式匹配器在匹配成功时，查找该表获取动作执行信号管道，将参数列表通过相应管道发送到动作执行器。

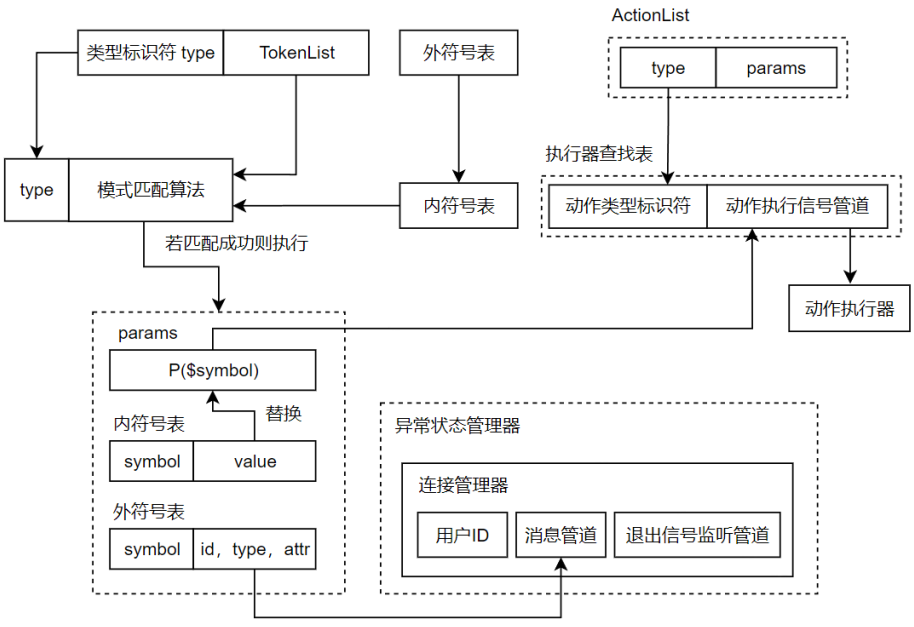

图 4.6 模式匹配函数的逻辑

DCA 语句的解析过程创建了四个对象，分别是外符号表、类型标识符、TokenList，以及 ActionList。规则解释函数调用模式匹配器生成算法，传入这四个参数生成模式匹

配函数。

算法 4.6：模式匹配器生成算法

输入：外符号表 symbolTable，类型标识符 type，TokenList，ActionList。

输出：模式匹配函数。

Start

返回一个没有输入，没有返回值的函数对象，其算法流程如下：

1、创建一个 map，key 为字符串类型，value 为数值类型，该 map 代表模式匹配器的内符号表 currData。

2、遍历 symbolTable，设遍历到的当前键值对的 key 值为 name，value 为数据源实体 data。用 data 查询数据源管理器，找到相应的最新数据 curr，将 curr 存储到 currData 中以 name 为 key 的存储单元。

3、内符号表构建完毕后，以 type 查找模式匹配算法，并将 TokenList 和 currData 传入模式匹配算法，等待算法返回执行结果。

4、若匹配成功，则遍历 ActionList，将各参数列表发往对应的动作执行信号管道。同时检查以部署规则用户 ID 为 key 的消息管道是否存在，如果存在，则用外符号表的 value 集合与规则类型标识符生成异常消息体，发送到消息管道中。

End

4.4.2　模式匹配算法

模式匹配算法的通用逻辑是利用内符号表将 TokenList 中的变量 Token 替换为数值 Token，然后对替换后的 TokenList 进行求值，得到匹配结果。

函数规则的模式匹配算法集合在系统部署时实现，与应用场景紧密相关。表达式规则的模式匹配算法与字符串表达式求值算法类似，利用 stack 数据结构，经过一次遍历求出表达式的值。表达式规则的模式匹配算法如图 4.7 所示。

算法 4.7：表达式规则的模式匹配算法

输入：TokenList，内符号表 currData。

输出：模式匹配的结果，true 代表匹配成功，false 代表匹配失败。

Start

1、创建一个 stack，名为 st。之后遍历 TokenList，记当前 Token 为 t。

2、如果 t 是数值 Token，则将 t 压入 st。

3、如果 t 是变量 Token，则根据 t 的 value 字段中的值，在 currData 中查找对应的 key，并将其 value 取出，取出的数就是该变量对应数据源的最新数据，用取出的数替换

t 的 value 字段，并通过修改 t 的 type 字段，将 t 从变量 Token 转为数值 Token，最后将 t 压入 st。

4、如果 t 是运算符 Token，则取出 st 中的栈顶两个元素，调用对应的运算函数，并将执行结果压入 st 中，运算函数的返回值也是 Token 类型，返回值的 Token 可能有两种，一种是数值 Token，另一种是布尔 Token（布尔 Token 是仅在表达式规则的模式匹配算法中临时使用的 Token 类型，其 type 属性的值为 BOOL，value 属性的值为 true 或 false），只有布尔 Token 可以被&运算符对应的运算函数调用。

5、遍历结束后，st 中保存着的即是模式匹配的结果，它是一个布尔 Token，将该布尔 Token 的 value 属性作为返回值返回。

End

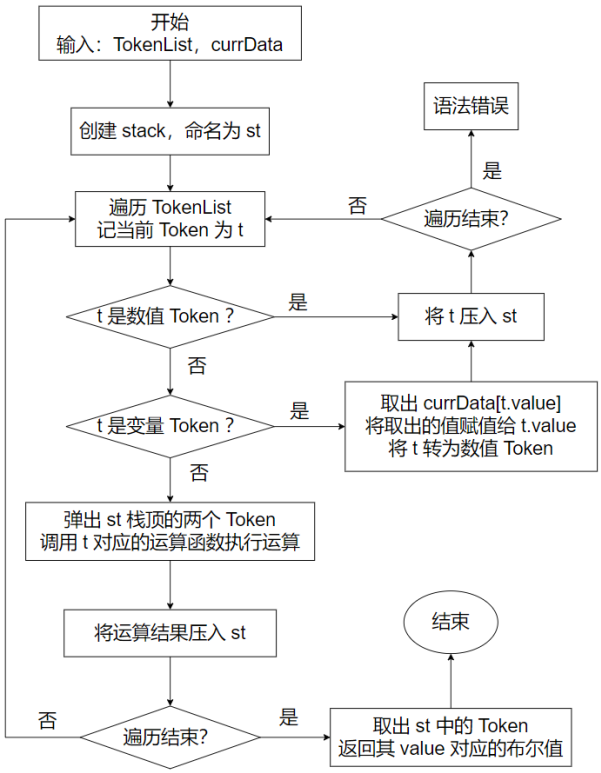

图 4.7 表达式规则的模式匹配算法

4.5 本章小结

本章设计了基于规则的物联网消息处理系统的规则描述语言 DCA，实现了规则解释函数。给出了规则解释函数将 DCA 规则转为模式匹配函数的全过程。

第五章 系统部署及扩展优化

本章给出了部署基于规则的物联网消息处理系统的一般步骤，并以某畜牧集团智慧农场的物联网设备消息处理作为部署案例。之后给出了系统框架的扩展方案，以及系统面对特殊场景时的优化方案。

5.1 系统的部署

基于规则的物联网消息处理系统作为中间系统，部署在物联网设备与物联网应用软件之间。在部署时根据物联网设备与应用场景的具体需求，实现消息接收器、设备访问控制模型、子执行器，以及函数规则集合。

5.1.1 部署系统的一般步骤

基于规则的物联网消息处理系统的部署具有如下一般步骤。

（1）实现消息接收器：根据物联网设备的传输协议与消息体结构实现消息接收器的消息接收协程。

（2）实现设备访问控制：根据应用场景的实际需求，实现物联网应用软件对设备消息的访问控制模型。

（3）实现子执行器：根据控制器设备的功能，以及应用场景的实际需求，实现动作执行器的子执行器集合。

（4）部署 DCA 规则：

1、根据应用场景的需求，实现函数规则集合。

2、利用 DCA 语言部署符合应用场景需求的规则。

5.1.2 部署案例

本文将基于规则的物联网消息处理系统部署到某养殖集团旗下智慧农场的物联网设备消息处理中，作为部署系统的一般步骤示例。智慧农场在养殖场地中部署了物联网设备，并根据业务需求设计了物联网应用软件，此时需要使用消息处理系统统一管理物联网设备消息，并为物联网应用软件提供接口。

部署示例系统使用 Go 语言开发，基于 Gin[28]框架实现 web 服务；使用 GORM[29]与数据库交互；利用 gorilla[30]响应物联网应用软件发起的 WebSocket[19]连接请求，并维持 WebSocket 会话窗口。部署示例如图 5.1 所示。

（1）部署环境

为了满足智慧农场的管理需求，物联网应用软件设计的功能如下。

1、养殖场地的实时环境监控与自动环境调节。

2、农场中生物生命体征信息的实时监控。

3、对不同管理人员区分设备访问权限。

4、利用地图和电子围栏任务对放养生物进行逃逸分析。

5、养殖环境与生物异常状态的实时报警。

农场中已部署的物联网设备如下。

1、使用北斗短报文发送位置信息的生物定位项圈，位置信息被北斗卫星转发到卫星信号接收服务器，服务器提供 HTTP 接口，用于访问最新数据和历史数据。

2、使用 Lora 协议与 MQTT 协议传输数据的生命体征检测耳标与环境检测仪。

3、使用 PLC（Programmable logic controller）控制器进行控制的通风设施与喷淋设施，PLC 使用 MQTT 协议接收指令。

4、利用云服务器维护直播画面的摄像头，提供获取直播地址的 HTTP 接口。

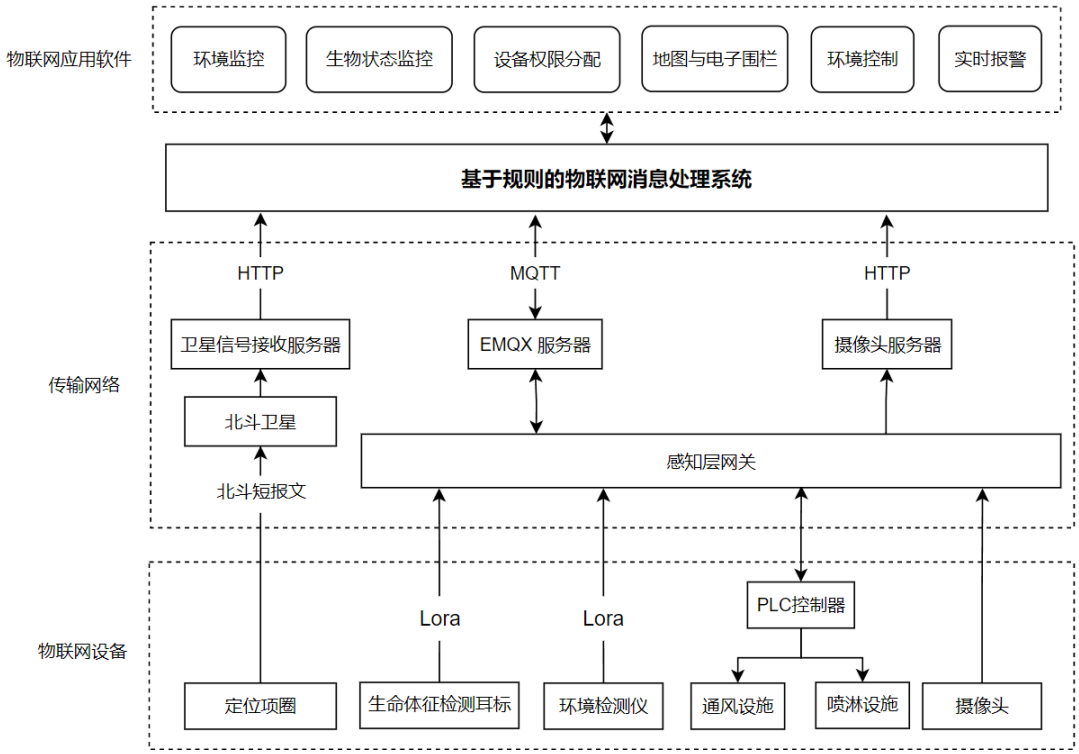

图 5.1 部署示例

（2）消息接收器的实现

消息接收器的实现分为定义设备类型标识符和实现消息接收协程两步。

1、生命体征检测耳标与环境检测仪的消息接收

以 EMQX[31]作为消息中转服务器，生命体征检测耳标和环境检测仪作为 EMQX 的客户端，向特定 topic 发布消息。生命体征检测耳标与环境检测仪的消息发布在不同的 topic 中。

EMQX 是一种 MQTT 代理。使用 MQTT 协议进行通信的双方需要同时连接到 MQTT 代理，MQTT 代理维护了存放消息的 topic。

生命体征检测耳标检测的数据为动物的体温，属性名为 temperature。环境检测仪同时检测温度与湿度，属性名为 temperature 和 humidity。

生命体征检测耳标与环境检测仪的消息接收实现如下。

1) 定义生命体征检测耳标的设备类型标识符：animal_tag。
2) 定义环境检测仪的设备类型标识符：environmental_detector。
3) 消息接收协程的工作流程如下。
   a) 创建 EMQX 客户端
   b) 分别订阅生命体征检测耳标和环境检测仪发布消息的 topic。
   c) 根据设备厂商提供的消息格式解析数据，将其转为结构化消息，存入消息数据库并更新数据源管理器。

2、定位项圈的消息接收

定位项圈的数据经过北斗卫星转发后，缓存在卫星信号接收服务器，服务器提供获取最新数据与历史数据的 HTTP 接口。定位项圈每 30 分钟上传一次数据，以 session 字段标记消息的先后顺序。消息接收器以轮询接口的方式从服务器获取最新数据。

定位项圈检测的数据为生物所在位置的经纬度，属性名为 longitude 和 latitude。

定位项圈的消息接收实现如下。

1) 定义定位项圈的设备类型标识符：position_collar。
2) 消息接收协程的工作流程如下。
   a) 利用 Go Cron 每 30 分钟访问一次最新数据接口。
   b) 每次获取到消息后，判断 session 是否等于上条消息的 session+1。如果等于，则接收该消息，30 分钟后再次获取数据。
   c) 如果 session 等于上条消息的 session+2，则说明出现消息遗漏，此时调用历史数据访问接口，获取遗漏的消息。
   d) 如果 session 与上条消息的 session 相同，则说明发生了消息晚到的情况，此

时调用 Go time.Sleep 方法休眠 30 秒后，再次尝试获取最新数据。

e) 获取到最新数据后，按照设备厂商提供的消息格式解析数据，存入消息数据库并更新数据源管理器。

（3）设备的访问控制模型

在智慧农场中，物联网设备的监控对象分为两类，一类是养殖场地，另一类是生物。消息接收器将设备消息存储在 MongoDB[32]数据库，每类设备的消息独占一个 collection。

设备的访问控制模型如图 5.2 所示。

1、将设备抽象为便携式设备与固定式设备两类。

2、固定式设备是对绑定在某个固定场所的设备抽象，在注册时需要指明其部署的养殖地。固定式设备包含摄像头和环境检测仪。

3、便携式设备是与生物具有一对一关系的设备抽象，在注册时需要指明其绑定的生物。便携式设备包含定位项圈与生命体征检测耳标。

4、用户仅拥有养殖地的权限，用户对物联网设备的访问权限依赖于其对养殖地的权限。

5、当用户拥有某养殖地的权限时，就同时拥有了该养殖地中所有固定式设备、生物、便携式设备的操作权限，当养殖地权限被取消时，后续权限也随之消失。

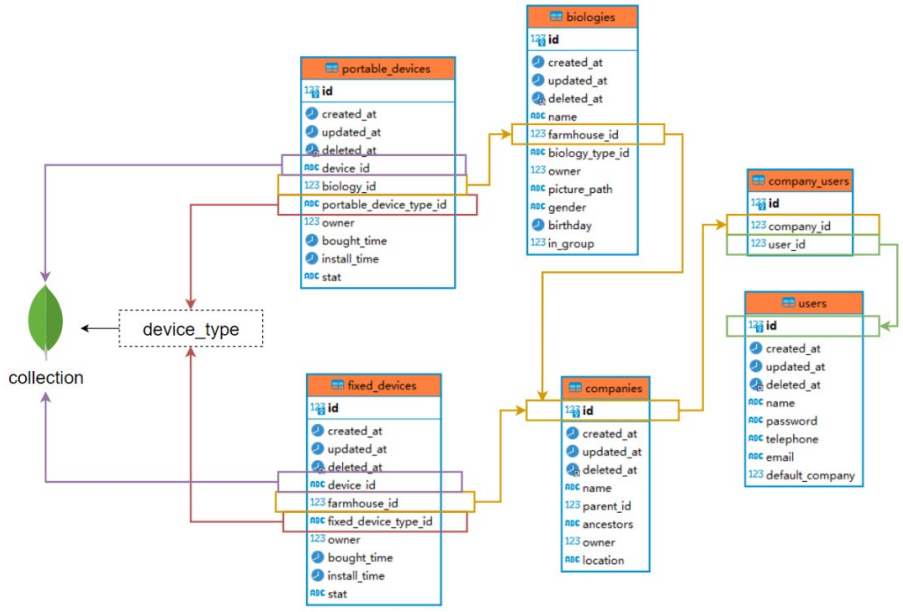

图 5.2 设备的访问控制模型

（4）子执行器的实现

子执行器的实现过程分为定义动作类型标识符、定义参数列表格式、实现动作函

数三步。在智慧农场的案例中，根据控制设备提供的功能与应用场景的需求，实现了两个子执行器。

1、实现向 PLC 发送控制指令的子执行器

在本例中，PLC 控制器使用 MQTT 协议接收控制命令。PLC 订阅 EMQX 的中名为 plc 的 topic，将拿到的消息进行解析并实现控制功能。

向 PLC 发送控制指令的子执行器实现如下。

1) 定义动作类型标识符：control_plc。
2) 定义参数列表格式：设备 id 号，设备类型，控制指令。
3) 实现动作函数：
   a) 将字符串形式的参数列表解析为结构化消息。
   b) 创建或调用 EMQX 客户端，将结构化消息发布到名为 plc 的 topic 中。

2、实现客户端定向消息推送的子执行器

客户端指的是物联网应用软件，客户端连接到异常状态管理器，接收消息推送。智慧农场中的第二个子执行器利用异常状态管理器，将自定义消息推送到指定客户端。某实体的管理员 c 使用此执行器将设备报警信息推送给该实体的其他管理员。

客户端定向消息推送的子执行器实现如图 5.3 所示。

1) 定义动作类型标识符：send_msg。
2) 定义参数列表格式：用户 ID，消息体 msg。
3) 实现动作函数：
   a) 使用用户 ID 在异常状态管理器的连接管理器中查找消息管道。
   b) 将 msg 发送到消息管道。

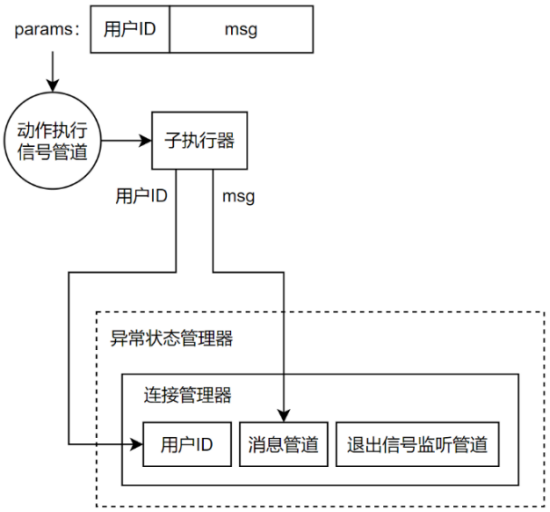

图 5.3 客户端定向消息推送的子执行器

（5）DCA 规则的部署

在实现了消息接收器与子执行器后，即可使用 DCA 语言部署规则。在智慧农场中，使用表达式规则的方式部署了生物和养殖地监控及环境控制策略。电子围栏任务较复杂，因此以函数规则的方式实现。

1、以表达式规则的方式部署实体监控与环境控制策略

1) 设生物 A 绑定的生命体征检测耳标的设备唯一编号为 a，监控其生命体征的 DCA 规则如下。当生物 A 的体温超过 30 度时，向用户 ID 为 3，4，7 的管理员推送一条报警信息，报警信息中携带检测到的生物温度。

Datasource=Animal_Temperature{a, animal_tag, temperature};

Condition=Animal_Temperature > 30;

Action={send_msg,3,4,7,This animal' temperature is too high!Current temperature: $Animal_Temperature};

2) 设生物 A 的养殖地部署的环境检测仪的设备唯一编号为 b，养殖地的通风设施设备唯一编号为 c，设备类型标识符为 air，喷淋设施的设备唯一编号为 d，设备类型标识符为 water。控制养殖地环境变量的 DCA 规则有如下两则。

a) 当生物 A 的体温高于 27 度，且养殖地环境温度高于 30 度时，打开养殖地的通风设施。

Datasource=Temp{b, environmental_detector, temperature},Animal_temp{a, animal_tag, temperature};

Condition=(Animal_temp>27)&(Temp>30);

Action={control_plc,c,air,open};

其中，Action 参数列表的最后一个属性 open 是命令 PLC 打开通风设施的控制指令。

b) 当养殖地的湿度小于 40 时，打开养殖地的喷淋设施。

Datasource=Hum{b, environmental_detector, humidity};

Condition=Hum<40;

Action={control_plc,d,water,open};

其中，Action 参数列表的最后一个属性 open 是命令 PLC 打开喷淋设施的控制指令。

2、以函数规则的方式实现电子围栏任务

函数规则的实现分为定义标识符和实现模式匹配算法两步。

物联网应用软件设计的电子围栏任务如下。

1) 用户使用物联网应用软件在地图上画出电子围栏。
2) 指明一个携带了定位项圈的生物。
3) 当生物离开电子围栏范围时，产生报警信息。

电子围栏任务的实现如下。

1) 定义函数规则标识符：fence。
2) 实现模式匹配算法：
   a) 接收参数：定位项圈的经度，定位项圈的纬度，电子围栏的坐标集合。
   b) 函数流程：利用内符号表解析 TokenList 后，获得点面关系判断算法的输入参数；对电子围栏坐标和生物坐标点进行点面包含关系判断，使用基于引射线法的点面关系判断算法。
   c) 返回值：布尔类型的值，代表是否逃逸。

设某放养生物携带的定位项圈的设备唯一编号为 e，使用 DCA 语言部署以下电子围栏任务。

Datasource=Ani_lon{e,position_collar,longitude},Ani_lat{e,position_collar,latitude};
Condition=fence, Ani_lon, Ani_lat, 116.403322, 39.920255, 116.410703, 39.897555, 116.402292, 39.892353, 116.389846, 39.891365;
Action=;

其中，Condition 语句的常量参数列表代表围成电子围栏的坐标点经纬度。Action 语句为空，表示模式匹配成功时不执行任何动作。若此生物逃逸，定位项圈的相关信息将通过异常状态管理器推送到客户端，异常消息的类型标识为 fence。

5.2 系统框架的扩展与优化

为了使基于规则的物联网消息处理系统能满足各种物联网应用场景的需求。本节从两个方面扩展与优化系统框架。

（1）系统框架的功能性扩展。

（2）系统面对特殊应用场景时可能面临的挑战及优化方案。

5.2.1 系统框架的扩展

DCA 语言代表了基于规则的物联网消息处理系统支持的功能集合。本文已实现的系统框架具有较高的可扩展性，经过扩展后，可以让 DCA 语言支持更多有用的功能，满足不同层次的应用场景需求。

（1）实现模式匹配器之间的数据传递

Thingsboard 平台的规则链满足了消息的多步处理需求。链式规则的好处在于实现了多规则节点之间的数据传递。在基于规则的物联网消息处理系统框架中，数据源管理器是规则的输入，子执行器是规则的输出。只要将上一个规则的输出与下一个规则的输入连接起来，即可实现与规则链类似的功能。

在数据源管理器中开辟一块不会被消息接收器访问的区域，供多个模式匹配器进行数据交互。设消息处理系统中不会出现唯一编号为-1 且设备类型标识符为 chain 的设备。利用两个子执行器来实现仅允许单用户部署的链式规则，如图 5.4 所示。

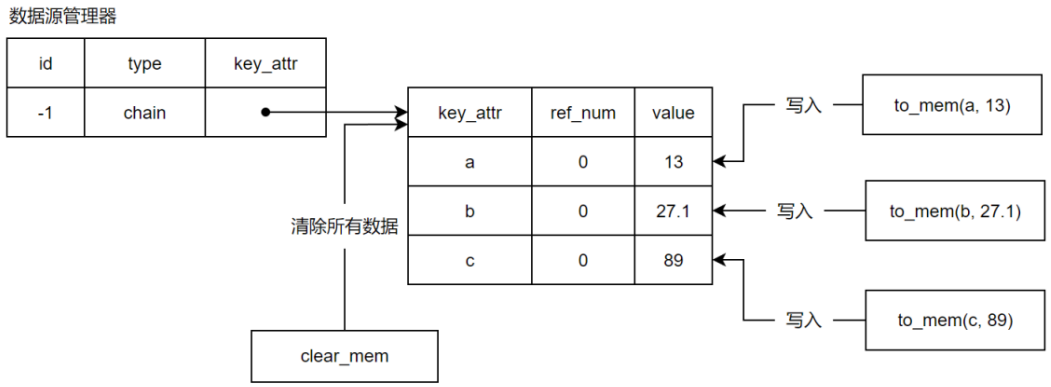

图 5.4 实现链式规则的子执行器

1、修改规则调度器的实现，使其不维护 id 为-1，type 为 chain 的数据源引用计数。

2、实现规则链子执行器 to_mem，将数据存入数据源管理器中的特定存储单元。

1) 定义动作类型标识符：to_mem。
2) 定义参数列表格式：key_attr，value。
3) 实现动作函数：
   a) 将字符串形式的参数列表解析为结构化消息。
   b) 将 key_attr，value 写入数据源管理器中 id 为-1，type 为 chain 的存储单元。

3、实现规则链终点子执行器 clear_mem，清空 id 为-1，type 为 chain 的存储单元。

1) 定义动作类型标识符：clear_mem。
2) 定义参数列表格式：不接收输入参数。
3) 实现动作函数：。清空 id 为-1，type 为 chain 的存储单元。

to_mem 动作将数据写入数据源管理器的特定区域。如果要让模式匹配器 matcherA 的输出进入模式匹配器 matcherB，只需在 matcherA 的 Action 中加入 to_mem 动作，并在 matcherB 的 Datasource 中添加 to_mem 的目标数据源即可。

（2）选择性地接收异常状态提示

异常状态管理器以 WebSocket 协议作为客户端连接方式，在推送模式匹配器执行结果的同时，满足其他组件的消息推送需求。在系统的现有框架下，模式匹配成功时，模式匹配器默认向部署该规则的客户端推送异常消息。

为防止大量异常消息“轰炸”客户端，考虑扩展 DCA 语言，增加 Warning 字段。修改生成模式匹配函数的算法，使其在匹配成功时推送 Warning 字段中的内容，若 Warning 字段为空，则不推送任何消息。与 Action 字段类似，可以使用$符号在 Warning 字段中引用数据源，以此将实时数据放入异常消息体。

5.2.2　针对特殊应用场景的优化方案

克努特优化原则（Knuth's optimization principle）指出，过早优化是万恶之源[33]。在软件开发中提前优化通常不可取，但有必要分析在一些特殊场景中，基于规则的物联网消息处理系统可能面临的挑战，并提出优化方案。以此提升消息处理系统在各种物联网场景中的兼容性。

（1）模式匹配触发条件的替换方案

由于模式匹配函数由 Cron 定期调用，因此模式匹配的触发条件是 Cron 的执行间隔计时结束。当触发模式匹配时，从数据源管理器中读入相应的数据，在系统的现有框架下，无法判断读入的数据是否已经验证过一次，也无法检测是否有消息遗漏的情况发生。

若执行模式匹配时，数据源管理器中的相应数据源还没有被更新，就会导致处理重复消息的情况发生。当处于模式匹配的执行间隔期间，若数据源管理器连续更新两条及以上的数据，则只有最新的一条数据会进行模式匹配。

一种简单的解决方案是记录下每类设备的消息上传频率，并设置一个与该频率相近的模式匹配执行周期。但如果出现了网络拥挤或者物联网设备宕机的情况，数据源管理器就会迟迟得不到更新，处理重复消息的情况仍然会发生。当某条消息因为网络拥挤而晚到时，在之后的某执行周期内就会连续到达两条消息，导致消息遗漏的发生。

设置一个合理的模式匹配执行周期，可以避免大部分消息遗漏和重复的情况，个别消息的遗漏和重复在大多数应用场景中不会造成恶劣影响。但针对某些特殊场景，有必要给出模式匹配触发条件的替换方案。

本文给出三种模式匹配的触发条件及实现方式，在部署消息处理系统时根据应用场景的实际需求进行选择。

1、查询式触发条件

模式匹配函数主动查找数据源管理器，称为查询式触发条件。好处在于不用进行任何同步措施，速度快，并且可以控制执行周期。这是系统现有框架的实现方案。

2、推送式触发条件

数据源管理器主动将新数据使用管道推送到模式匹配器，称为推送式触发条件，用在每条消息都需要进行处理的场景。在使用推送式触发条件的系统中，不适合控制执行周期，因为在执行间隔时间内，数据如果没有被取走，将导致更新数据源管理器的协程被长期阻塞。推送式触发条件保证模式匹配器不遗漏地处理每条消息。

3、带 session 的查询式触发条件

在数据源管理器和外符号表中同时添加 session 字段，防止查到重复数据。每次执行匹配前，都查询外符号表中数据源的 session 是否小于数据源管理器，如果是，则更新外符号表 session，之后执行匹配；如果不是，则不执行匹配。带 session 的查询式触发条件可以避免消息重复，并检测到消息遗漏，但是无法保证不发生消息遗漏。

三种模式匹配触发条件的对比如表 5.1 所示。

表 5.1 三类模式匹配触发条件的对比

类型	实现方式	优点	缺点
查询式触发条件	利用 Cron 定期查询数据源管理器	内符号表构建速度快	可能发生遗漏处理和重复处理
推送式触发条件	用管道将数据推送至模式匹配器 使用无限循环的子协程实现模式匹配器	保证不发生遗漏处理和重复处理	内符号表构建速度慢
带 session 的查询式触发条件	利用 Cron 定期查询数据管理器，只在 session 大于上次匹配时的 session 时执行模式匹配	保证不发生重复处理 能检测到遗漏处理 内符号表的构建速度较快	可能发生遗漏处理

推送式触发条件的实现方法是在数据源管理器中存放以规则 ID 作为 key，管道作为 value 的 map，而不存放实际数值。消息接收器在更新某数据源时，向 map 中的所有管道发送相同的消息。规则解释函数将规则 ID 以闭包自由变量的方式，存入模式匹配函数。规则调度器使用一个执行无限循环的子协程来调用模式匹配函数，模式匹配函数利用规则 ID 和外符号表，监听每个数据源管道，每当某管道中有数据到达，就用它

填充内符号表，当内符号表被填充满后，随即执行模式匹配。

在推送式触发条件的实现方案中，数据源管理器不再需要引用计数，因为 map 容量就代表了需求数据源的规则数。规则调度器中的两个定时器不再修改引用计数，而是添加管道与删除管道。

推送式触发条件不适用与某些下发控制指令的场景。异常数据到达后，规则触发控制指令，若控制指令为温度下降五度，此时通常需要等待一段时间后再查看数据是否正常，如果没有恢复正常则再下降五度。若对每条消息都进行处理，可能产生大量重复的控制指令，现场环境将陷入危险境地。在查询式触发条件中，只要将模式匹配执行周期设置的大一些就可以避免这种情况的发生。

推送式触发条件适合做消息路由。例如过滤出异常数据，并将相关数据统一发送到某个消息队列或存储在某张数据表，以供后续数据分析。例如，当设备属性 A 高于 30 时，将 A,B,C,D 四个属性同时存储到某数据库表，之后使用这些数据来分析造成属性 A 异常的原因。

在使用推送式触发条件时，不建议编写消息到达频率相差太大的两个设备的联动处理规则。因为这会导致内符号表的构建速度变慢（原因在于，一个符号的数据到达后，可能需要很久另一个符号的数据才到达）。这时，如果又有几批新消息到达，向数据源管理器中的管道发送数据的协程会因为上一次模式匹配没有结束，管道中存在未被取走的数据，而陷入较长时间的阻塞，这将影响到其他数据源的更新。

查询式触发条件不适合做消息路由，因为无法判断查到的是不是重复消息，也无法判断是否发生消息遗漏。查询式触发条件的优势在于内符号表的构建速度快。

本文建议按照以下原则来选择模式匹配的触发条件。

1、可接受重复处理消息与消息遗漏，则使用推送式触发条件。

2、允许消息遗漏，严格不接受消息重复，且要求控制执行间隔，则使用带 session 的推送式触发条件。

3、不允许重复处理，也不允许遗漏，每条消息都要处理，且只能处理一次。此时使用推送式触发条件，虽然运行效率相对较低，但保证了消息的可靠处理。

（2）对复杂表达式进行模式匹配的优化方案

如果表达式规则的匹配条件表达式过于复杂，将导致模式匹配器的执行速度变慢，影响消息的处理效率。本文提出基于分治法执行模式匹配的优化方案，对表达式规则模式匹配算法的改进如下。

1、在执行模式匹配之前，先计算 TokenList 中 value 为&的运算符 Token 数量，记作 n。由于多个子条件是并列关系，不具备先后顺序，因此可以并发执行多个

子条件匹配。

2、当 n 大于某阈值时，将原 TokenList 划分为 n/m 个子表达式（其中 m 根据系统硬件性能设定）。

3、针对每个子表达式，单独开启一个协程去执行模式匹配。

4、最后将所有协程的计算结果做逻辑与运算，得到原表达式的匹配结果。

（3）对超大容量数据源管理器进行数据更新的优化方案

数据源管理器的实现基于 map 数据结构。由于消息接收器会查询每条到达消息中是否存在某个属性被数据源管理器需求，当设备数量过多时，查找数据源的耗时可能造成性能瓶颈。

本文参考 Key-Value 数据库 Redis[34]面对类似挑战时的解决方案，提出使用布隆过滤器[35]来优化消息接收器对数据源管理器的更新过程。布隆过滤器使用多个哈希函数将 key 映射到二进制向量的某位，0 与 1 指示了此 key 是否存在。其查找效率高，但存在一定的误判率。由于布隆过滤器不能删除数据，其指示存在的 key 可能不存在，其指示不存在的 key 一定不存在。布隆过滤器的查找效率远高于常规的 map，其特性也符合查找数据源的优化需求。

使用布隆过滤器优化后的数据源管理器更新步骤如下。

1、每当创建新的数据源管理器条目时，也同时创建一份布隆管理器中的相应条目。

2、消息接收器在收到一条数据并将其解析为结构化消息后，以 id 和 type 为索引去查找布隆过滤器，如果指示存在，则访问并更新数据源管理器。如果不存在则不进行下一步动作。

5.3 本章小结

本章分析了部署基于规则的物联网消息处理系统的一般步骤，包括实现具体的消息接收器、设备访问控制、子执行器集合，以及部署 DCA 规则。以某畜牧集团智慧农场的物联网场景为例部署了消息处理系统。给出了系统框架的扩展方案，以及系统面对特殊应用场景时的优化方案。

第六章 总结与展望

6.1 总结

本文设计并实现了一个基于规则的物联网消息处理系统。系统具备统一处理异构物联网设备消息的能力，设计了用于部署设备消息处理规则的规则描述语言 DCA。给出了消息处理系统部署在具体物联网应用场景的一般步骤，包括实现消息接收器、实现设备访问控制模型、实现子执行器集合，以及部署 DCA 规则。将消息处理系统部署在某畜牧集团的智慧农场物联网场景中，利用 DCA 语言满足了智慧农场的自动化养殖需求。论文的最后扩展了系统的框架，分析了系统在特殊应用场景中可能面临的挑战，并给出了优化方案。

规则描述语言 DCA 语法简洁易于理解，相比 Thingsboard 平台的规则链，前者优势在于能在一个规则中处理多个设备的消息。考虑到系统部署后可能面临的物联网应用软件需求变化，本文在设计系统时尽量降低了各个组件之间的耦合性，使得系统框架具有较强的可扩展性。

6.2 进一步工作

在基于规则的物联网消息处理系统的现有架构中，部署在同一数据源上的规则可能发生执行动作的语义冲突。因此本文的进一步工作是扩展规则调度器的功能，使其能够对规则进行冲突检测。同时，在规则调度器中实现基于优先级的调度算法，当数据源同时触发多个规则时，根据规则的优先级决定动作执行的优先级。

参考文献

[1] Kaed C E, Khan I, Van Den Berg A, etc. SRE: Semantic Rules Engine for the Industrial Internet-Of-Things Gateways[J]. IEEE Transactions on Industrial Informatics, 2018, 14(2): 715–724.

[2] Kiran M P R S, Rajalakshmi P, Bharadwaj K, etc. Adaptive rule engine based IoT enabled remote health care data acquisition and smart transmission system[A]. 2014 IEEE World Forum on Internet of Things (WF-IoT)[C]. 2014: 253–258.

[3] Mainetti L, Mighali V, Patrono L, etc. A novel rule-based semantic architecture for IoT building automation systems[A]. 2015 23rd International Conference on Software, Telecommunications and Computer Networks (SoftCOM)[C]. 2015: 124–131.

[4] Mazon-Olivo B, Hernández-Rojas D, Maza-Salinas J, etc. Rules engine and complex event processor in the context of internet of things for precision agriculture[J]. Computers and Electronics in Agriculture, 2018, 154: 347–360.

[5] ThingsBoard - Open-source IoT Platform[EB/OL]. /2023-03-25. https://thingsboard.io/.

[6] De Paolis L T, De Luca V, Paiano R. Sensor data collection and analytics with thingsboard and spark streaming[A]. 2018 IEEE Workshop on Environmental, Energy, and Structural Monitoring Systems (EESMS)[C]. 2018: 1–6.

[7] Kadarina T M, Priambodo R. Monitoring heart rate and SpO2 using Thingsboard IoT platform for mother and child preventive healthcare[J]. IOP Conference Series: Materials Science and Engineering, IOP Publishing, 2018, 453(1): 012028.

[8] Forgy C L. Rete: A fast algorithm for the many pattern/many object pattern match problem[J]. Artificial Intelligence, 1982, 19(1): 17–37.

[9] 纪佩宇, 印杰, 武旭红. 面向物联网的大数据预警信息高速处理规则引擎[J]. 计算机与数字工程, 2013, 41(9): 1478–1481.

[10] 陈癯, 王海峰, 贾建鑫, 等. 基于海量异构传感器的物联网水质监测系统[J]. 计算机应用与软件, 2020, 37: 5.

[11] 田瑞琴, 吴尽昭, 唐鼎. 物联网网关中轻量化规则引擎的设计与实现[J]. 计算机应用, 2015, 35(4): 1035–1039.

[12] 杨慧, 丁志刚, 郑树泉. 基于规则的消息处理引擎的设计与实现[J]. 计算机应用与软件, 2013, 30(10): 67–70.

[13] Drools - Drools - Business Rules Management System (Java™, Open Source)[EB/OL]. /2023-03-13. https://www.drools.org/.

[14] 叶昊, 杨建军. 基于物联网的复杂事件处理框架研究与应用[J]. 机械工程与自动化, 2017, 0(5): 63–65.

[15] 陈雪勇, 陈建, 张丹吉. 基于静态脚本引擎的物联网开放平台设计[J]. 信息技术与信息化. 2021(10).

[16] 冯俊辉, 刘晨, 郭浩然. 基于模板和规则的声明式代码生成[J]. 数字技术与应用, 2022, 40(2): 151–154.

[17] Quincozes S, Emilio T, Kazienko J. MQTT Protocol: Fundamentals, Tools and Future Directions[J]. IEEE Latin America Transactions, 2019, 17(09): 1439–1448.

[18] Shelby Z, Hartke K, Bormann C. The Constrained Application Protocol (CoAP)[R]. RFC 7252, Internet Engineering Task Force, 2014.

[19] Melnikov A, Fette I. The WebSocket Protocol[R]. RFC 6455, Internet Engineering Task Force, 2011.

[20] Martin Fowler. Domain Specific Languages[M]. 1st edition. China Machine Press, 2013.

[21] bilibili, Gengine[Source Code]. 2023.

[22] Ball J A, Scarlett R J. Algorithms for RPN calculators[M]. New York: Wiley, 1978.

[23] Fisun M, Horban H, Ihor K. Processing of Relational Algebra Expressions by the Shunting Yard Algorithm[A]. 2019 IEEE 14th International Conference on Computer Sciences and Information Technologies (CSIT)[C]. 2019, 3: 240–243.

[24] The Go Programming Language[EB/OL]. /2023-03-13. https://go.dev/.

[25] cron package[Source Code]. /2023-03-13. https://pkg.go.dev/github.com/robfig/cron.

[26] Wilson P R. Uniprocessor garbage collection techniques[C]//Memory Management: International Workshop IWMM 92 St. Malo, France, September 17–19, 1992 Proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005: 1-42.

[27] Transmission Control Protocol[R]. RFC 793, Internet Engineering Task Force, 1981.

[28] Gin Web Framework[EB/OL]. Gin Web Framework. /2023-03-13. https://gin-gonic.com/.

[29] Jinzhu. GORM[EB/OL]. GORM. 2023-03-08/2023-03-13. https://gorm.io/index.html.

[30] Gorilla, the golang web toolkit[EB/OL]. /2023-03-13. https://www.gorillatoolkit.org/.

[31] Inc E T. EMQX: The World's #1 Open Source Distributed MQTT Broker[EB/OL]. emqx.io. /2023-03-13. https://www.emqx.io/.

[32] MongoDB: The Developer Data Platform | MongoDB[EB/OL]. /2023-03-13. https://www.mongodb.com/.

[33] Knuth D E. The art of computer programming[M]. Pearson Education, 1997, 3.

[34] Redis. Bloom Filter Pattern[EB/OL]. Redis. 2022-07-14/2023-04-05. https://redis.com/blog/bloom-filter/.

[35] Bloom B H. Space/time trade-offs in hash coding with allowable errors[J]. Communications of the ACM, 1970, 13(7): 422–426.

致谢

时光荏苒，流光过隙，转眼间研究生三年的学习生活临近尾声。盛夏来临，毕业也悄然而至，求学生涯也在这个夏天告一段落。回想三年的曲折经历，一路属实不易，但收获颇丰，心存满满感激。

首先，真诚地感谢亦师亦友的导师——陈向群教授。三年的学业，离不开您的睿智指导与点拨，使困惑的我拨云见日、豁然开朗。本论文是在陈向群老师的悉心指导下完成，在论文的撰写、反复修改及最终定稿的过程中，陈老师总是耐心细心地阅读学生所写文章，进而提出宝贵的指导建议，小到表达错误和逻辑错误也一一指出。其间通过与陈老师的不断沟通讨论，让我深刻认识到了学术与实际项目之间的差异，理解了学术理论来源于实际项目，但又高于实际项目，其适用面应更广泛，更具备通用性。

其次，感谢北大给予的浓厚学术环境和众多的学习资源，感谢学院老师们提供的悉心教导，感谢师兄师姐给予的帮助，感谢与同门们三年的同窗之谊。感谢我的家人在三年读研期间提供的经济援助和精神支持，你们是我坚强的后盾。感谢大爷和大娘三年来对我的关心与指导，每次节假日的聚会都让我心中倍感温暖。还要感谢大爷大娘在读研时提供的生活方面的关心与帮助，让我能不被琐事打扰，一门心思做论文。

然后，感谢实习公司给我提供的锻炼机会，让我接触并学习了专业领域知识的实际应用，开拓了我的视野，拓宽了我的知识面，提升了我的专业技能。感谢实习项目组提供的宝贵机会，给予我极大的自主空间，不仅能充分发挥自己的专业能力，而且还能锻炼交流协作的能力。

最后，感谢三年来没有虚度光阴的自己，在有限的在校时间里最大限度的充实自己，提升自身各方面能力。前路漫漫，继续努力，一路奋进！

北京大学学位论文原创性声明和使用授权说明

原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名：陈肯　　日期：2023 年 5 月 27 日

学位论文使用授权说明

（必须装订在提交学校图书馆的印刷本）

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

- 按照学校要求提交学位论文的印刷本和电子版本；
- 学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务，在校园网上提供服务；
- 学校可以采用影印、缩印、数字化或其它复制手段保存论文；
- 因某种特殊原因需要延迟发布学位论文电子版，授权学校□一年/□两年/□三年以后，在校园网上全文发布。

（保密论文在解密后遵守此规定）

论文作者签名：陈肯　　导师签名：

日期：2023 年 5 月 27 日

提交终版学位论文承诺书

本人郑重承诺，所呈交的学位论文为最终版学位论文。本人知晓，该版学位论文将用于校学位评定委员会审议学位、国家和北京市学位论文抽检。论文版本呈交错误带来的结果将由本人承担。

论文作者签名：陈肯

日期：2023 年 5 月 27 日