东 莞 理 工 学 院

本 科 毕 业 论 文

|  |  |  |
| --- | --- | --- |
| 毕业设计题目： | 面向云原生架构的高性能流量网关设计与实现 | |
| 学生姓名： | 周梓健 | |
| 学 号： | 202041417344 | |
| 院 系： | 计算机科学与技术学院 | |
| 专业班级： | 计算机科学与技术3 班 | |
| 指导教师姓名及职称： | | 任子良 副教授 |
| 起止时间： 2024年 12月--2025年 5月 | | |

**摘要**

随着云原生应用和微服务架构的发展，传统的单体应用逐渐被多个独立服务所替代，这种分布式系统架构虽然在灵活性和扩展性上有显著提升，但也带来了新的挑战，尤其是在流量管理、安全性、服务互操作性和系统监控等方面。为了有效应对这些挑战，网关作为中介层的解决方案应运而生，成为现代系统架构中的重要组成部分。在没有网关的架构中，每个服务需要独立处理请求路由、负载均衡、协议转换、身份验证、安全策略等多项功能，这不仅增加了服务的开发与运维成本，还导致了系统高度耦合，难以实现灵活的扩展和升级。此外，服务间缺乏统一的流量管理和监控，导致系统维护的复杂度大幅提高。现代网关能够通过集中管理流量、策略和安全认证，简化系统架构并提升系统可维护性和可扩展性。

本论文的主要工作如下：

1. 对每连接一线程 、多路复用 、反应堆三个 I/O 架构进行性能测试，同时进行非阻塞优化，并选择 Reactor 作为最终实现网关的架构。
2. 实现 TCP、WebSocket、HTTP 协议的代理，包含 TLS 连接与 HTTP 内容压缩与零拷贝优化。
3. 实现基于 IP 的防火墙、速率控制与负载均衡。
4. 实现 OAuth外部身份验证，心跳检测、gRPC 管理接口与 Docker等云原生功能。

**关键词：**云原生网关，高性能服务器，操作系统

**目录（这里页码是乱的）**

[第1章 引言 1](#_Toc164630450)

[1.1课题背景与研究意义 1](#_Toc164630451)

[第2章 系统架构设计与技术选型 5](#_Toc164630456)

[2.1系统总体架构设计 5](#_Toc164630457)

[2.3相关开发环境与工具 13](#_Toc164630459)

[第3章 系统设计与实现 16](#_Toc164630461)

[3.1系统设计整体框架 16](#_Toc164630462)

[3.1 I/O 模型的对比与选择 17](#_Toc164630463)

[3.2 TCP，HTTP，Websocket协议代理 17](#_Toc164630463)

[3.3 流量控制 18](#_Toc164630464)

[3.3.1 速率控制 20](#_Toc164630467)

[3.3.2 负载均衡 18](#_Toc164630466)

[3.3.3 流量过滤 20](#_Toc164630467)

[3.4 云原生功能支持 22](#_Toc164630468)

[3.4.2 gRPC 管理接口 24](#_Toc164630470)

[3.4.3 心跳检测 24](#_Toc164630470)

[3.4.4 Docker 容器化 24](#_Toc164630470)

[3.5本章小结 30](#_Toc164630471)

[第4章 系统测试与评估 31](#_Toc164630472)

[4.1 系统功能与性能测试 31](#_Toc164630473)

[4.2系统存在的缺陷 38](#_Toc164630475)

[第5章 总结与展望 40](#_Toc164630477)

[参考文献 41](#_Toc164630478)

[致谢 44](#_Toc164630479)

# 引言

## 1.1课题背景与研究意义

随着互联网架构的不断演进，系统间通信的复杂性日益提高。早期的 Web 系统往往采用单体结构，通信路径简单，服务器直接处理来自客户端的请求。然而，随着服务规模的扩大与系统职责的细化，微服务架构逐步成为主流选择。在这一架构下，系统被拆分为大量独立服务，服务间的通信频繁，且面临跨协议、跨网络、跨环境的种种挑战。为此，网关（Gateway）这一组件逐渐成为系统架构中不可或缺的关键角色。

网关的雏形可以追溯到反向代理服务器的广泛应用，例如 Nginx、Apache HTTPD 等，这些工具最初主要承担请求转发与负载均衡的功能。在微服务时代，传统的反向代理逐渐演化为具备更强服务治理能力的 API 网关，典型代表如 Kong、Traefik、Envoy 等。这些现代网关不仅支持多协议代理，还集成了认证授权、限流熔断、监控追踪等一系列功能，成为服务网络的统一入口，兼顾了安全性、可观测性与灵活性。

尽管这些成熟的网关方案在工业界已有广泛应用，但对于特定的教学研究或轻量化部署场景而言，其较高的使用门槛与复杂的配置机制，往往不利于系统的精细掌控和底层机制的深入理解。因此，本文旨在设计并实现一款具备高性能通信能力与基础云原生特性的自研网关系统，探索其在多协议转发与服务接入控制中的实际应用能力。

本文所设计的网关基于 C++ 实现，采用事件驱动与异步非阻塞的 I/O 模型，并使用自实现的解析器实现了高效的 HTTP 请求解析与协议分发逻辑。在协议支持方面，系统支持 TCP、HTTP 及 WebSocket 等常见网络协议的代理，满足前后端在不同场景下的通信需求。在此基础上，系统进一步引入了流量控制机制，包括速率限制、基于轮询的负载均衡与请求过滤逻辑，以提升系统在高并发场景下的稳定性与安全性。

此外，为适配现代服务治理体系，本系统集成了若干云原生功能：通过 OAuth 协议实现与外部认证服务的对接，利用 gRPC 提供统一的配置与管理接口，内置心跳检测机制用于后端服务的健康监测，并支持基于 Docker 的容器化部署，方便系统在多环境下的交付与运维。

# 系统架构设计与技术选型

## 2.1 系统总体架构设计

系统整体架构由各模块以及负责不同功能的线程与一个任务队列组成，架构图如下所示：

## 这图还得改，负载均衡要画上，心跳检测，OAuth 和 IP 防火墙合在一起叫安全控制模块

每个普通客户端的请求在进入网关后，将依次经过以下处理模块：首先是流量过滤模块，用于拦截来自黑名单 IP 的连接，并进行 OAuth 身份认证；随后通过速率控制模块限制请求频率，以防止恶意请求造成系统过载；接着进入TLS 模块，负责建立安全连接；之后由协议解析模块对请求内容进行解析，支持 HTTP 与 WebSocket 协议；随后通过路由匹配模块，根据请求的路径与方法匹配路由，得到候选结果集合；在负载均衡模块中，通过轮询等策略在候选集合中选择目标后端；最后，网关根据匹配结果决定将请求转发至远程服务器，或直接返回本地文件数据。

本论文所实现的网关系统主要由三个核心线程组成：主线程、gRPC 管理线程和工作线程。其中，主线程负责读取配置文件、初始化 IO 框架，并启动 gRPC 管理线程和工作线程。gRPC 管理线程提供管理接口，支持外部通过 gRPC 协议发送控制指令，实现对网关运行参数的动态修改，如更新速率控制上限、启用或关闭 TLS 支持、调整路由表等。工作线程则负责处理实际的客户端请求，通过轮询和执行任务队列中的非阻塞任务来实现高效并发处理。该设计支持工作线程的横向扩展，即通过增加线程数量可提升网关的并发处理能力，从而适应不同规模的网络负载。

## 2.1 相关开发环境与工具

2.1.1 开发环境与工具

|  |  |
| --- | --- |
| 类别 | 版本号 |
| Operating system | Ubuntu 24.04 |
| CPU | 12th Gen Intel(R) Core(TM) i5-12600K |
| C++ | 17 |
| CMake | 3.25.0 |
| vcpkg | 8.9.2 |
| Docker | 27.3.1 |

2.1.2 程序依赖库

1. Boost.Asio: Boost.Asio 是一个跨平台的异步 IO 编程库，提供了统一的接口来处理底层系统的网络、定时器、信号等异步事件，支持同步和异步操作，广泛用于构建高性能网络服务器与客户端应用。该库基于 IO 模型，如 epoll（Linux）或 IOCP（Windows），隐藏了底层平台差异，是构建分布式系统或高并发程序的重要组件。
2. spdlog：spdlog 是一个高性能的 C++ 日志库，具有轻量级、接口简单、无依赖、开箱即用的特点。它支持异步日志写入，能够在高并发场景下极大减小日志对主流程的性能影响。spdlog 支持多种日志格式、文件滚动（按大小或日期分割）、控制台输出彩色日志等特性，适用于从嵌入式项目到大型服务器程序的各种应用。
3. OpenSSL：OpenSSL 是当前最广泛使用的开源密码学库之一，提供了全面的加密算法和协议支持。常用于实现 RSA、AES 等对称与非对称加密算法，MD5、SHA256 等哈希算法，以及 Base64 和 Base64Url 编码。OpenSSL 还实现了 TLS（传输层安全协议）协议栈，是构建 HTTPS、加密通信、证书管理等安全系统的核心组件。
4. nlohmann-json：nlohmann/json 是一个现代 C++ JSON 库，以简洁易用而著称。它使用 STL 容器和操作符重载，使得 JSON 的构造、解析、访问、序列化与反序列化过程自然、直观。支持 JSON Schema、Unicode、兼容标准 JSON 文本格式，是处理配置文件、网络数据交换、API 开发中不可或缺的组件。
5. jwt-cpp: jwt-cpp 是一个专注于 JSON Web Token（JWT）处理的 C++ 库，用于生成、签名、验证和解析 JWT。它支持多种签名算法，如 HS256、RS256 等，并与 OpenSSL 等加密库兼容。该库可以方便地在认证授权流程中实现用户身份验证和权限控制，是构建安全 RESTful API 和单点登录系统的关键工具之一。

# 第3章 系统设计与实现

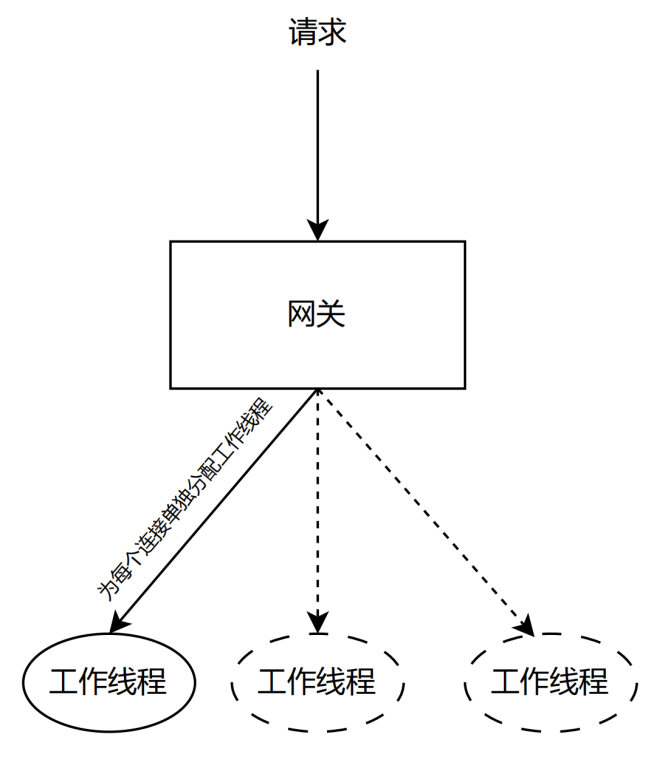
## 3.1 I/O 架构的测试与选择

在服务器开发中，I/O（Input / Output，输入输出）是指程序与外部世界（如客户端、磁盘等）之间进行数据交换的过程。I/O 架构，则是指系统在处理这些 I/O 操作时所采用的组织方式和程序结构。它描述了程序如何发起 I/O 请求、如何等待 I/O 完成、以及在高并发场景下如何管理多个 I/O 连接之间的协作关系。

不同的 I/O 架构在处理模型、资源利用效率和并发能力方面各有特点。本小节将对常见的 I/O 架构进行性能测试，并选择最优者作为后续网关开发的基础架构。

3.1.1 每连接一线程（ Thread-per-Connection ）

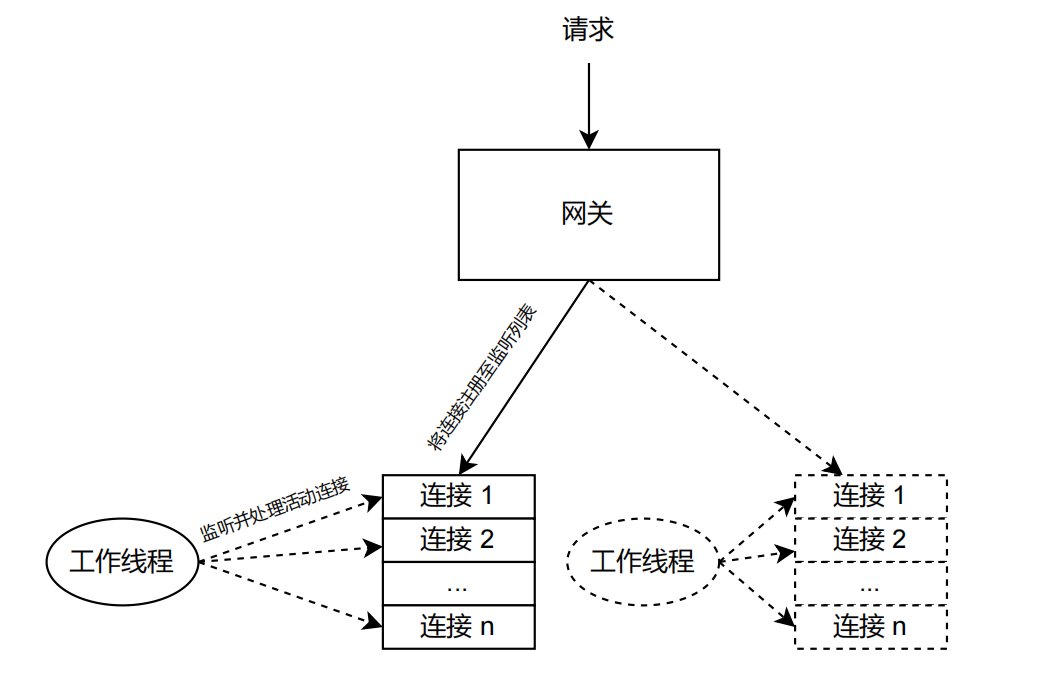
在每连接一线程的架构中，对于每一个 TCP 连接，服务器都会启用一个新的线程处理后续的数据交换。但是在高并发的网络环境中，大量的线程创建将导致服务器资源快速枯竭，严重影响整体系统可用性。下面的示意图展示了这种架构的大致工作流程：



3.1.2 多路复用 （Multiplexing I/O）

不同于 3.2.1 中所述为每个连接都创建一个线程而导致大量资源消耗的架构，多路复用架构可以在单个线程上监听并处理多个连接。其原理是线程通过 select，poll 或epoll之一的系统调用向内核提交一组需要监听的文件描述符，并阻塞在该系统调用上。在某个文件描述符据准备就绪（连接状态建立或释放，有数据到达或出现文件结束符号等事件产生）的时候，系统调用返回，进程停止阻塞并可以执行相应的处理逻辑，之后进入下一轮事件循环。

在这样的架构下，一个线程能同时处理多个连接，有效地节省了系统开销。下面的示意图展示了这种架构的大致工作流程：



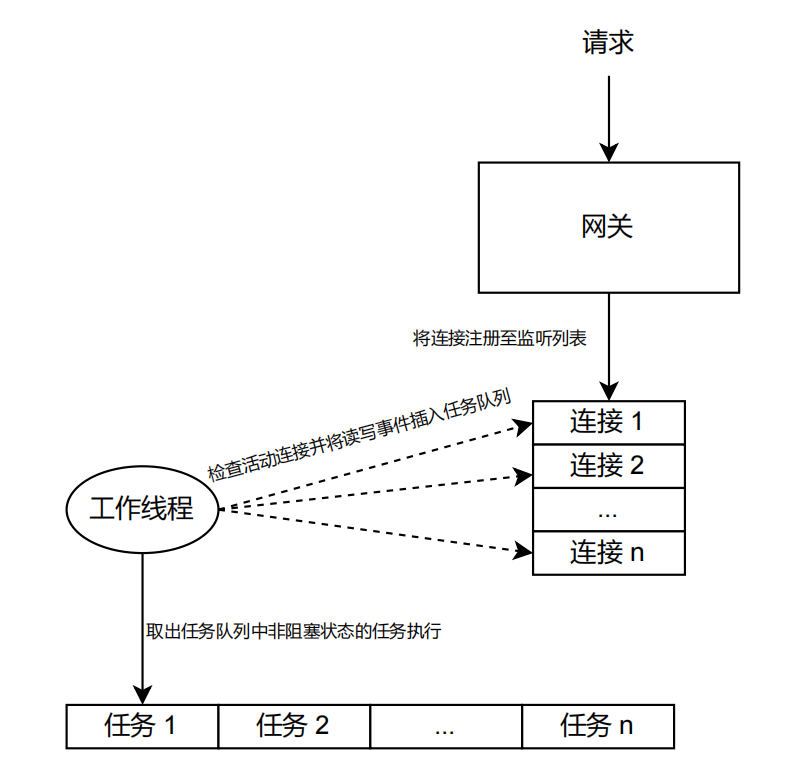
3.1.3 反应堆 （Reactor）

在上述 I/O 架构的处理流程中，网络数据的读写靠着 read ( ) 和 write ( ) 这样的系统调用 ( System Call )完成，默认情况下，这种读写将以阻塞的方式进行。

在操作系统中，阻塞（Blocking）是指进程在发起了一个系统调用之后，由于该系统调用的操作不能立即完成，需要等待一段时间，于是内核将进程挂起为等待状态， 以确保它不会被调度执行，占用 CPU 资源。而 Linux 操作系统将线程看作轻量级线程（LWP，Light-Weight Process）。所以阻塞的概念在线程调度中仍然适用。

使用阻塞的方式进行数据读写意味着在发起 read ( ) 和 write ( ) 这样的系统调用后，线程将等待至文件描述符就绪才会返回。假设程序以单线程的方式运行，在阻塞的期间，程序将无法接受新的连接，直到数据读写完毕。

在反应堆模式中，数据读写一般以非阻塞的方式进行。在 Unix 类的系统中，套接字可以通过设置 O\_NONBLOCK 标志位进入非阻塞模式。在这个模式下，假设在读写数据时数据并未准备就绪，线程将直接进入下一个事件的处理。下面的示意图展示了这种架构的工作流程，其中工作线程的数量同样可根据需求横向扩展：



反应堆模式通常与事件队列相结合，每个工作线程都将不断检查队列中是否存在可以处理的事件，随后取出事件执行相应的处理函数。在这个架构下，工作线程的数量可以根据实际情况便捷地进行水平扩展。上述架构也被称为事件驱动架构，现代编程网络库中诸如Boost Asio， libevent 和 Netty 都采用了这样的事件驱动架构。

3.1.4 三种 I/O 架构性能测试

一个系统的吞吐量通常用 QPS (Queries Per Second)来进行描述，其计算公式为：

在 Intel Core i5-12600KF 芯片上 ，只使用单线程运行 I/O 架构。在接收到请求之后，为了模拟网络延迟，使当前线程睡眠 100 至 500 毫秒，并在接收到客户端请求后返回相同的页面。使用 wrk 压力测试工具模拟10分钟，20个 TCP 连接的压力，相同的实验重复进行3 次并取平均数。我们可以得到三种 I/O 架构的性能如下表所示：

|  |  |  |  |
| --- | --- | --- | --- |
| I/O 架构 | 平均响应时间（ms） | 平均最长响应时间（ms） | QPS（s⁻¹） |
| 每连接一线程 | 1260.0 | 1820.0 | 3.26 |
| 多路复用 | 1180.0 | 1950 | 3.39 |
| 反应堆 | 2.3 | 167.5 | 40234.2 |

由于每连接一线程与多路复用的 I/O 架构都会因为模拟的网络延迟阻塞当前线程，而反应堆架构会在没有数据可读时检查队列中的下一个任务，所以反应堆架构得到的测试结果都会远远优于前二者。可见在高并发场景下，反应堆架构有着最好的吞吐量。后续我们将继续使用这个 I/O 架构完成网关的开发。

## 3.2 TCP，HTTP，Websocket 协议代理

3.2.1 总述

网关的基本功能是将到来的数据根据配置的目的地进行转发。但不同于传统的交换机在内网中简单地根据链路层地址进行转发，现代网关通常需要根据复杂的规则进行请求的处理，常见的例子包括：

1. 根据 HTTP 报文中的请求路由进行匹配，并在改写请求路由字段后发送数据
2. 在 HTTP 头部字段中寻找 Authorization 并提取 token 进行身份验证
3. 将 HTTP 请求转换为 Websocket 请求
4. 检查数据的 IP 地址和端口号是否在黑名单内并进行过滤

上述的例子说明网关应该具备一定的协议解析能力。

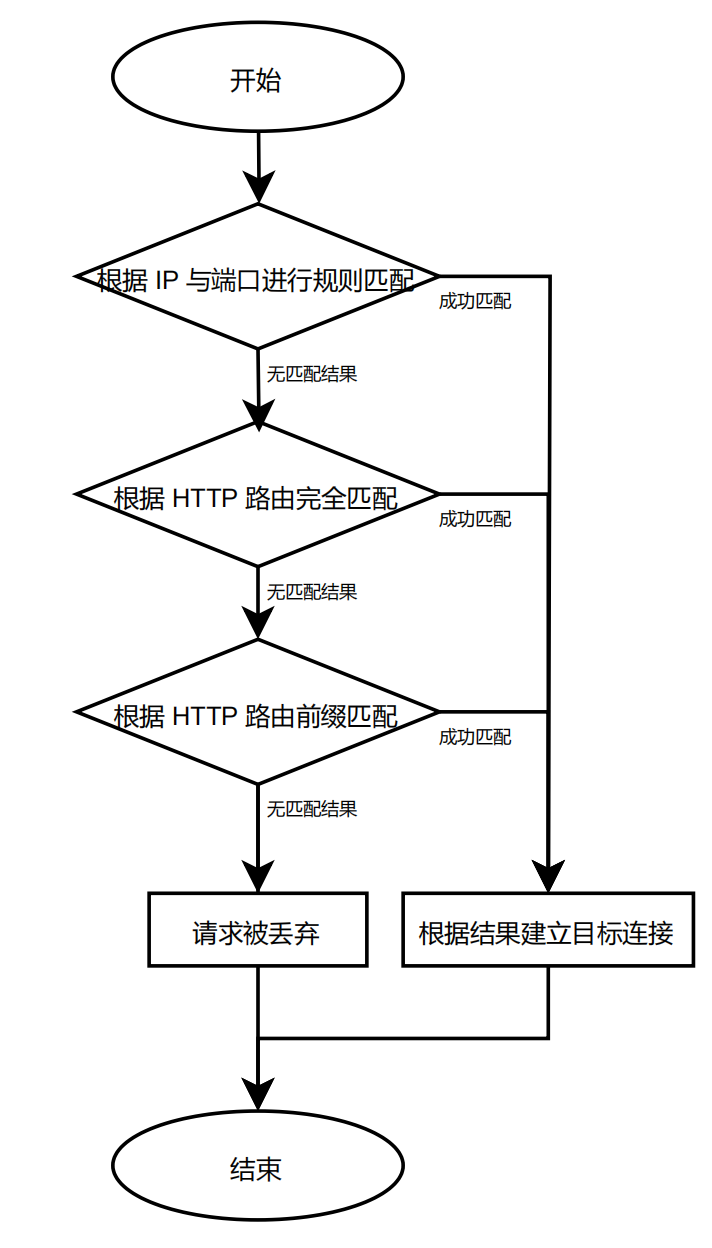
本小节将详细介绍网关对HTTP与Websocket 解析与代理的机制，并说明在传输安全（ TLS 连接）与性能优化 （HTTP 内容压缩）的关键实现细节。

3.2.2 路由匹配模块

当请求流量到达网关并成功通过流量过滤模块时，网关将会根据流量使用的协议与内部配置的规则决定流量应该向何处转发。本网关主要处理 TCP，HTTP 与 WebSocket 三种协议的匹配与转发。

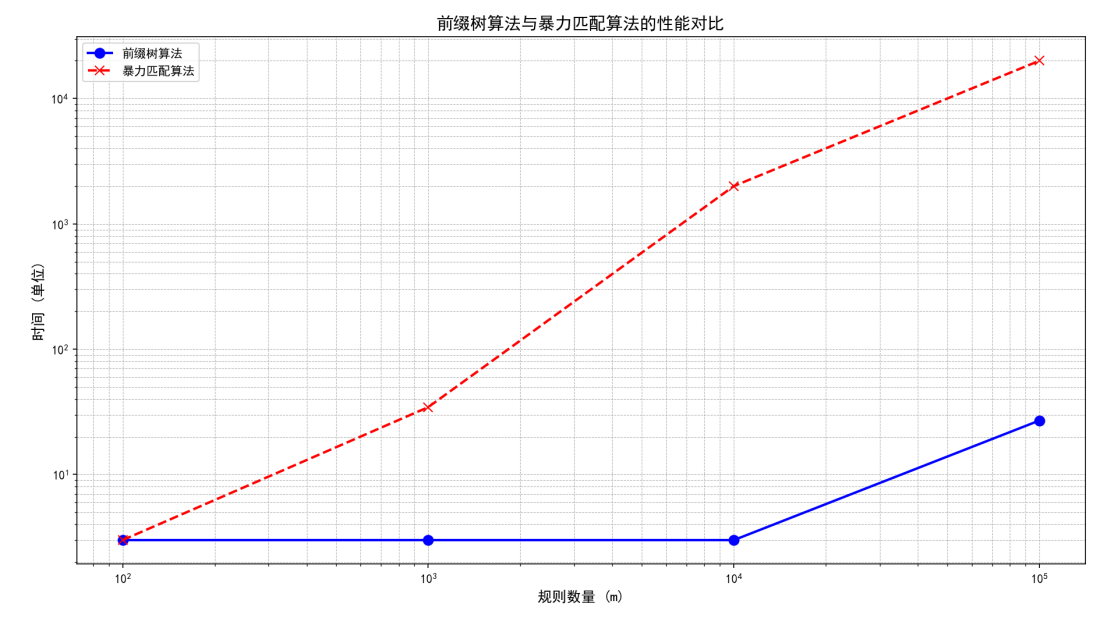
由于 TCP 属于 HTTP 与 WebSocket 的下层协议，配置的规则可能会发生冲突。因此本网关在不同网络层协议冲突时优先处理下层协议，即 TCP 协议将被优先处理，同时尽管 IP 地址的概念属于网络层而不是 TCP 协议本身，由于网络应用通常使用 IP 与端口号的组合标识网络地址，本网关在实现上提供了更加便利的支持。

对于 HTTP 与 WebSocket 协议，可配置基于请求路由的匹配规则。匹配模式主要分为完全匹配与前缀匹配，当计算完全匹配未获得结果时会再次进行前缀匹配。若仍无匹配结果，则请求将被丢弃。路由模块工作流程如下图所示：



不管是 IP与端口号组合或是 HTTP 路由匹配中的完全匹配，网关的实现都是通过在规则集合里遍历寻找值完全相同的字符串。而对于HTTP 路由匹配中的前缀匹配而言，在规则集合中进行遍历将是低效的。假设多个规则都共享相同的前缀，这些前缀将会被重复遍历匹配。因此本网关的前缀匹配将借助前缀树（Trie）实现。

前缀树通过共享相同前缀的节点来减少空间消耗，同时在查找时避免了多次遍历相同的前缀字符。用 n 代表平均字符串长度，m 代表存储的规则数量。查找的时间复杂度为 O(n)，建树的时间复杂度为 O(n\*m)。由于路由规则通常在网关运行前就规定好，所以建树事件可以忽略不计。在 Intel Core i5-12600KF芯片上，假设平均规则长度为 15 个 ASCII 字符。不同数量的路由规则下，前缀树算法与暴力匹配算法的性能对比如下图所示：

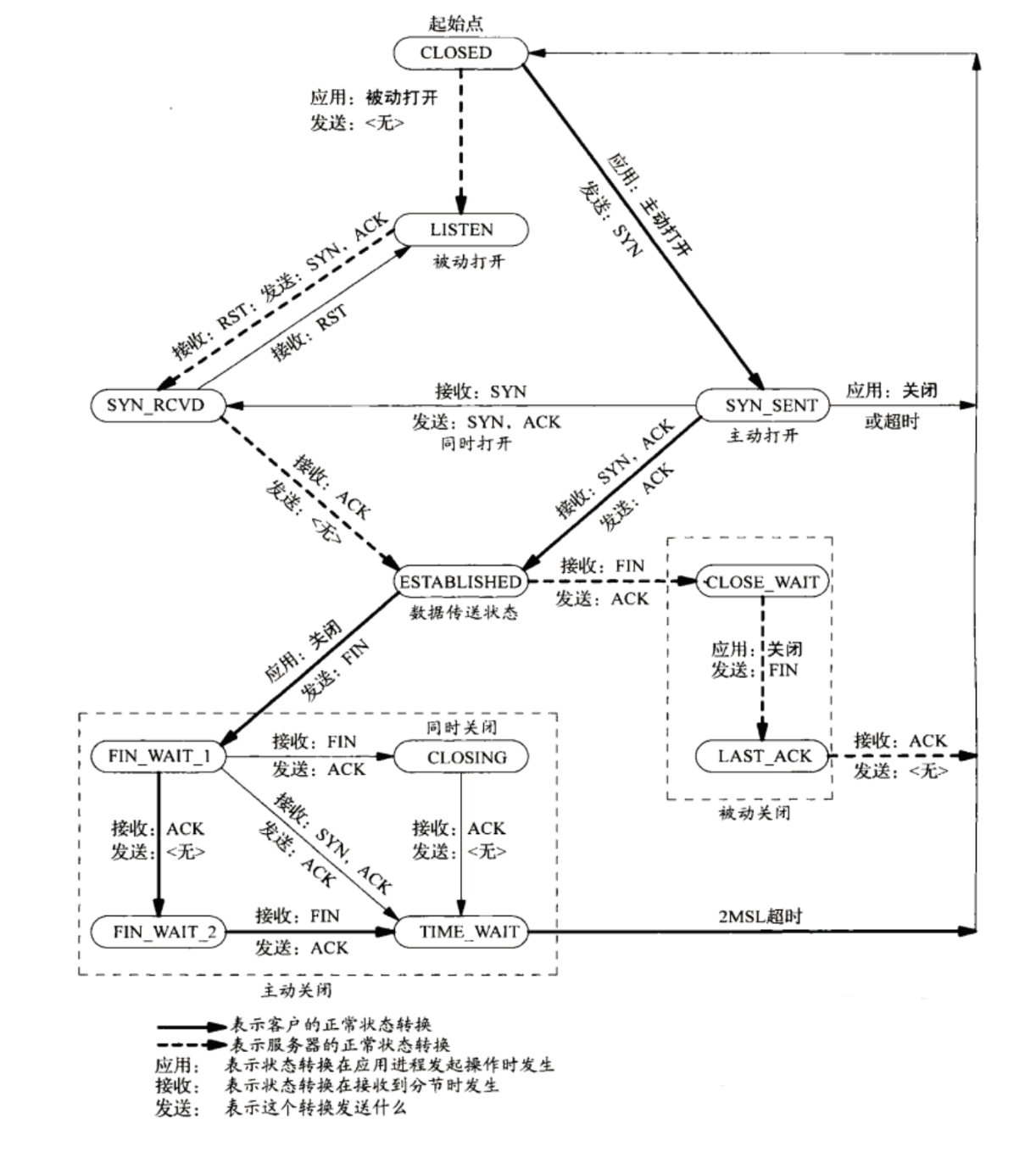


在规则数少于 10000条时，前缀树算法保持着稳定的 3毫秒的处理时间。可见前缀树匹配算法有着更好的性能。

3.2.3 TCP 与 TLS 连接

TCP（Transmission Control Protocol，传输控制协议）是一种面向连接的传输层协议，具备可靠传输、顺序保证以及错误检测能力，是 HTTP、WebSocket 等应用层协议赖以通信的基础。它通过三次握手建立连接，四次挥手关闭连接，并在传输过程中依赖序列号、确认号与窗口机制保障数据的有序与完整。

现代操作系统已经对 TCP 协议进行了完整实现，因此在应用开发中通常无需手动处理 TCP 报文的拼装与重传，只需关注连接的建立、维持与释放过程。此外，系统接口还允许开发者配置若干与性能相关的 TCP 参数，例如发送/接收缓冲区大小（通过 SO\_SNDBUF 和 SO\_RCVBUF 设置）、是否启用 Nagle 算法（通过 TCP\_NODELAY 控制是否推迟小包发送以提高带宽利用率）、连接保活机制（通过 SO\_KEEPALIVE 定期探测死连接）等。本论文的网关开发使用默认的 TCP 参数，实现 TCP 连接的管理与分配。后文的 Websocket 与 HTTP 等应用层协议都建立在 TCP 的基础上，客户端的所有请求都将从建立 TCP 连接开始。TCP 连接与释放状态机如下图所示：



在 TCP 提供可靠传输的基础上，实际部署中的网关通常还需对传输内容进行加密，以防止中间人攻击和敏感数据泄露。为此，TLS（Transport Layer Security）协议作为一种建立在 TCP 之上的加密层，被广泛应用于保护网络通信的安全。在 TLS 连接之上，应用层协议如 HTTP 和 WebSocket 可演化为加密版本，即 HTTPS 和 WSS（WebSocket over TLS）。

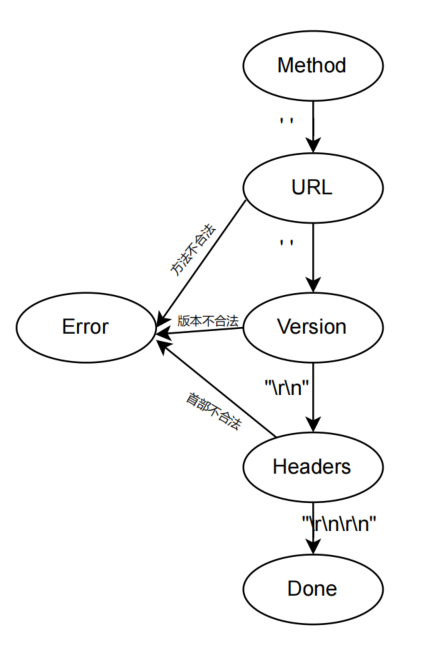
网关可以作为所有 TLS 流量的入口，在数据解密后将未受保护的数据转发至内网应用，也可以直接转发加密的数据至内网应用。前者称为“透传模式”，后者称为“终止模式”。借助 OpenSSL 库，本论文网关实现了终止模式的数据代理。

3.2.4 HTTP 代理与相关优化

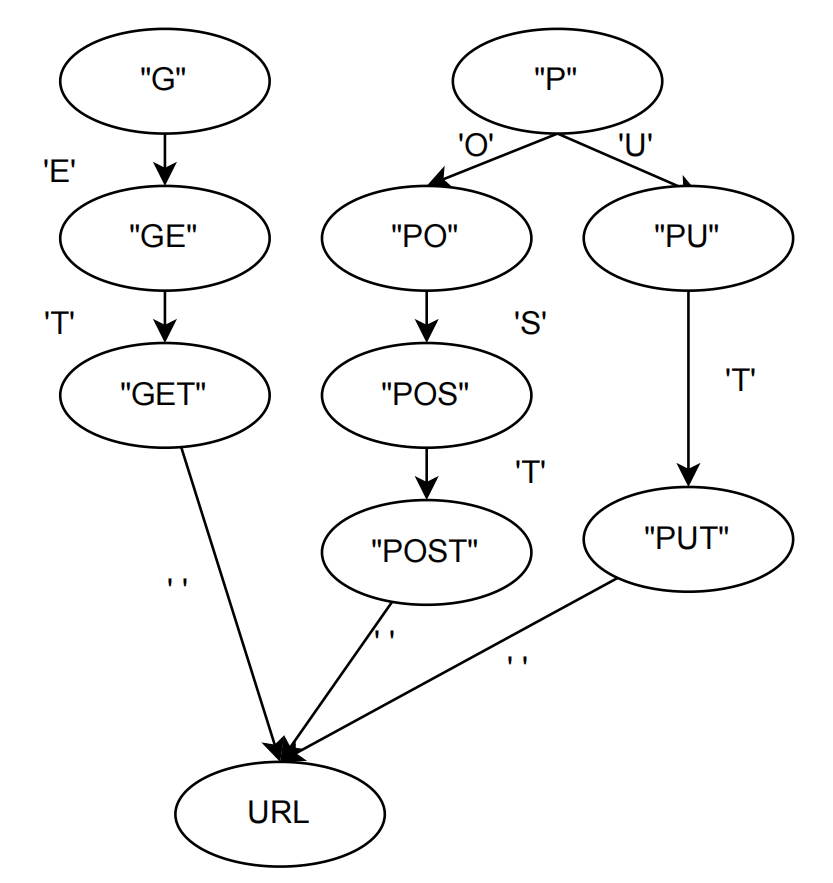
HTTP（HyperText Transfer Protocol，超文本传输协议）早期是一种基于 TCP 的应用层协议，广泛用于浏览器、客户端与服务器之间的数据交换。随着需求的发展，HTTP 协议经历了多个版本的演进，从传统的 HTTP/1.1（RFC 2616），到支持多路复用与头部压缩的 HTTP/2（RFC 7540），再到基于 UDP 和 QUIC 协议的 HTTP/3（RFC 9114），逐步提升了性能与可靠性。尽管新版本在性能上有所改进，但当前大多数应用程序仍以 HTTP/1.1 为主，因其兼容性好、实现成熟，且部署更为广泛。HTTP/1.1 协议数据格式如下图所示：

网络协议的解析通常由人为编写的状态机（State Machine）实现。状态机逐字符对到来的报文进行扫描，并及时更新内部状态。由于不涉及回溯与嵌套，状态机解析请求的时间复杂度为 O (N)，比使用正则表达式的字符匹配实现（复杂情况下能达到指数级的时间复杂度）更为高效。

本论文网关自实现了对 HTTP/1.1 协议的解析，其中状态转移过程示意图如下：



其中 Method 状态存在内部的状态转移，由于 HTTP 协议标准规定了请求方法（GET，POST，PUT，OPTIONS等），Method 内部应该在这些序列之间进行状态转移，其他的输入将会使状态机进入错误状态，提示调用者报文数据格式错误。同理于 HTTP 版本字段的解析，由于本论文网关只支持 1.1 版本的 HTTP 协议，在解析 HTTP 版本字段的时候应当逐字符解析到 “HTTP/1.1”这个序列。部分 HTTP 请求方法状态转移示意图如下：

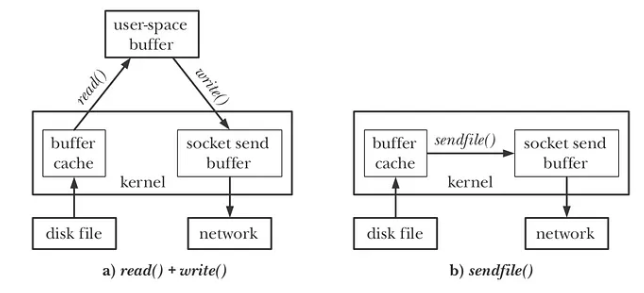


鉴于 HTTP 是应用最为广泛的网络协议，为了提升代理的性能与传输效率，本系统在 HTTP 代理模块中引入了 IO\_URING 机制下的零拷贝与（Zero-Copy）与内容压缩（Content-Encoding）两项优化手段：

（1）零拷贝：

当进行 HTTP 代理时，若路由匹配模块计算后得到的目标路径为本地文件（比如使用网关代理前端网页或静态文件），即进行文件代理服务，则涉及服务器本地磁盘读写。在高并发环境下，本地磁盘读写通常会成为性能瓶颈。

传统文件传输通常需要服务器将文件数据从内核空间拷贝到用户空间，处理后再通过套接字将数据写回内核空间发送给网络设备。这种“多次拷贝”的过程会带来较大的 CPU 开销，尤其在进行大量静态资源转发时效率较低。为此本系统采用了零拷贝技术，通过内核态直接在文件系统与网络设备之间转移数据，避免不必要的用户态拷贝。二者对比如下图所示：



在 Linux 中，常见的零拷贝机制包括 sendfie()，splice() 等系统调用，其中 sendfile()系统调用适用于从文件系统向套接字传递数据。在Intel Core i5-12600KF 芯片上，测试传统 “read() + write()” 读写与 sendfile() 对 HTML 网页进行代理的性能。其中测试程序 wrk 使用 4 线程与 100 个 TCP 连接，实验重复进行三次，每次进行 60 秒。结果如下表所示:

|  |  |  |
| --- | --- | --- |
| 文件传输方式 | 平均响应时间（ms） | 每秒处理请求个数（/s） |
| read() + write | 2.35 | 42029.64 |
| sendfile() | 1.37 | 72609.88 |

由表中数据得知使用 sendfile() 进行文件传输能大幅度提升文件传输性能。

（2）内容压缩：内容压缩是一项常见的优化手段，用于减小传输数据的体积、降低带宽占用。标准 HTTP 协议中，与内容压缩有关的的头部字段与常见值如下表所示，其中 “\*” 表示接受任意压缩方式：

|  |  |  |
| --- | --- | --- |
| 首部名 | 常见值 | 说明 |
| Accept-Encoding | gzip，deflate，br，\* | 客户端首次请求网页时，服务器将接受的压缩算法填入这个头部字段返回 |
| Content-Encoding | gzip，deflate，br | 用于指示数据载荷的压缩算法 |
| Transfer-Encoding | chunked | 设置为 chunked 的时候 Content-Length 字段无效 |

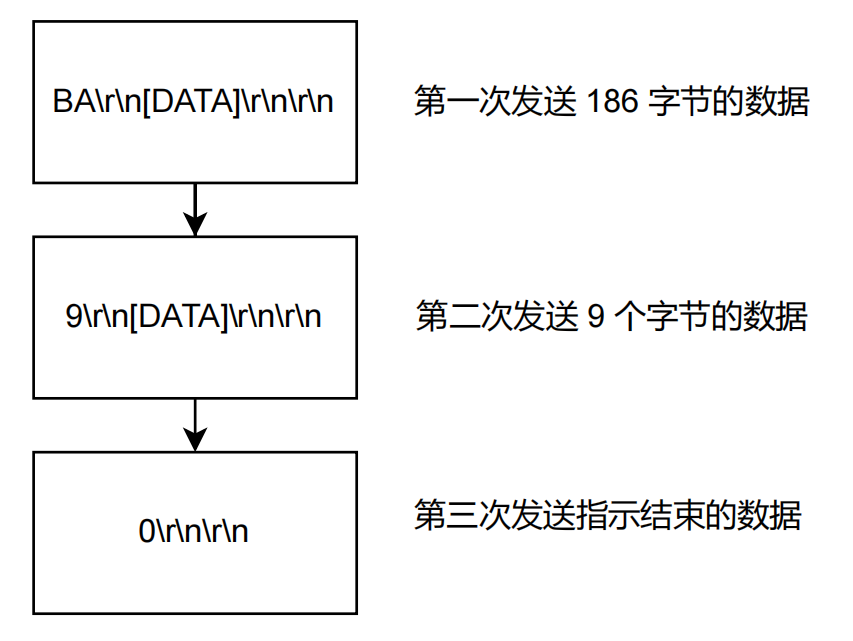
尽管对数据进行压缩能减轻网络压力，提升网络传输速度。但对于大文件的传输而言，数据的压缩与解压会消耗大量时间，引入不可忽略的网络时延。因此 gzip 与 br 等压缩库都提供了流式编程接口 （Streaming API）以对数据分块压缩。同时通过设置 HTTP 头部字段中的 Transfer-Encoding 为 chunked，发送端可以在压缩完小部分数据后就进行数据传输，而不是在所有数据都压缩完毕后再一并传输。

当 Transfer-Encoding 设置为 chunked 时，HTTP 标准对数据载荷的格式做出了如下要求：

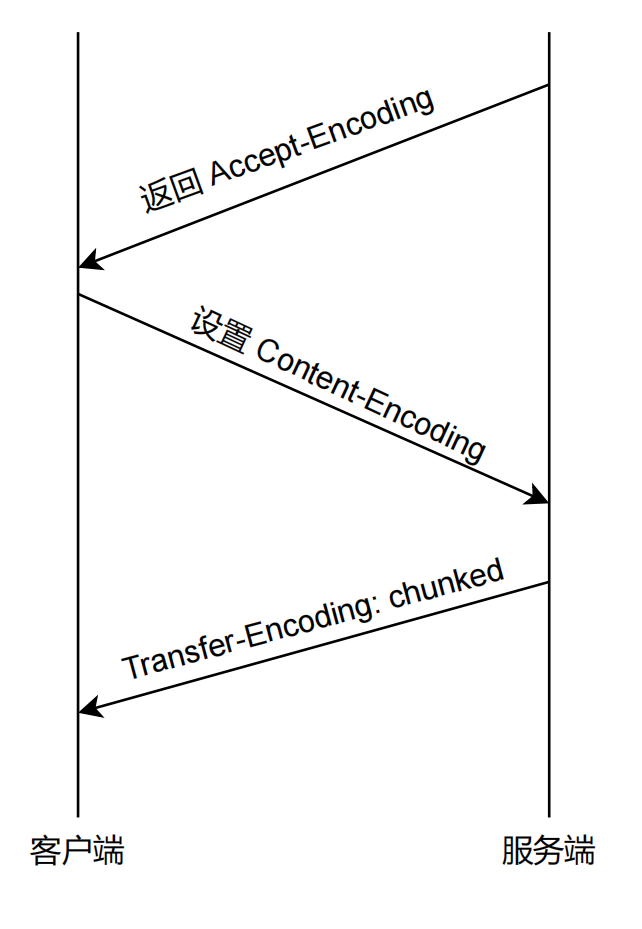
（1）每个载荷应该包括压缩数据块的长度与压缩数据块，以两个 CRLF 代表载荷的结束。其中数据块的长度应该用十六进制进行表示，并将数值转换为 ASCII 字符串。压缩数据块长度与压缩数据块本身以CRLF符号分隔。

（2）在所有压缩数据块都发送完毕之后，发送端应该发送一个长度为 0 的数据压缩块指示数据的结束。

假设一个文件需要分成两个压缩块进行发送，用 “[DATA]”代表压缩数据块，一个内容压缩与分块传输的例子如下，方框内的字符代表每次传输的数据载荷。可读字符使用 ASCII 编码：



本论文网关实现了基于 gzip 压缩算法的分块传输的支持，整体的请求与响应流程示意如下：



3.2.5 WebSocket 代理

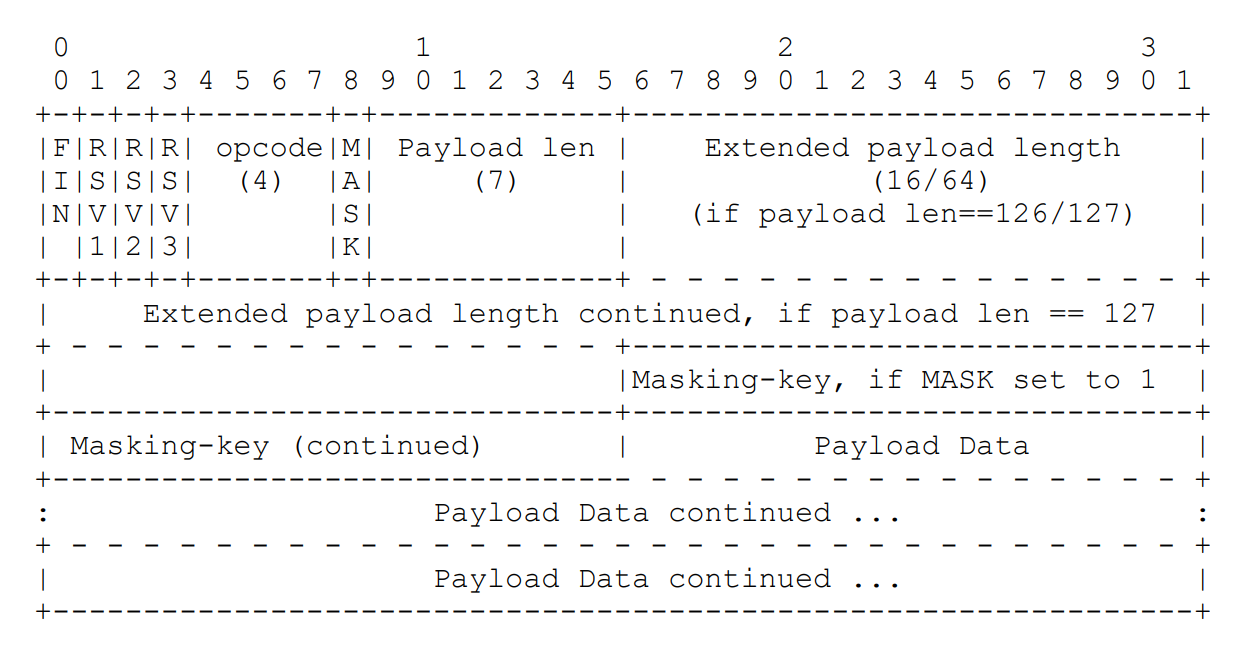
WebSocket 是由 HTTP 协议扩展的全双工应用层协议，广泛应用于在线聊天与多人游戏等需要实时数据传输的场景。为了将 HTTP 协议升级为 WebSocket 协议，需要设定特定的首部字段，并向服务器发送 GET 请求，需要设置的首部如下表所示：

|  |  |
| --- | --- |
| 首部名称 | 说明 |
| Upgrade | 通常值为 websocket |
| Connection | 通常值为 Upgrade |
| Sec-WebSocket-Key | 由客户端生成的随机字节 |

如果服务器支持 WebSocket 协议并同意协议升级，会返回状态码为 101 （Switching Protocols）的响应报文，并设置下表中所示字段：

|  |  |
| --- | --- |
| 首部名称 | 说明 |
| Upgrade | 通常值为 websocket |
| Connection | 通常值为 Upgrade |
| Sec-WebSocket-Accept | 将客户端发送的 Sec-WebSocket-Key 值与服务端生成的随机密钥拼接，进行 SHA-1 哈希运算， Base64 编码后得到 |

与上文所述代理 HTTP 协议相比，WebSocket的数据帧的长度应从 Payload len与 Extended payload length 字段中获取，同时应该为使用 WebSocket 的流量保持 TCP 长连接。WebSocket 协议数据帧格式如下图所示：（RFC 6455）



由于 WebSocket 的握手首部与 HTTP/1.1 兼容，在本论文网关的 WebSocket 代理实现中，将使用3.2.4 节中的 HTTP 解析器同时对 WebSocket 与 HTTP 报文进行解析。当解析器在首部发现“Connection: Upgrade” 这样的信息时，网关将使用 WebSocket 的方式进行通信。

## 3.3 安全控制模块

3.3.1 概述

在现代微服务架构中，安全性是系统设计的关键考虑因素之一。Azugate 安全控制模块主要包括基于 IP 地址的防火墙机制、OAuth 外部认证与访问控制策略。其中，OAuth 协议虽广泛应用于云原生架构，但本系统中更强调其作为访问控制组件的角色，因此将其归入安全控制模块进行统一阐述。通过将这些关键功能整合至网关层，可有效减少服务之间冗余的安全实现逻辑，提升系统的统一性与安全性。

3.3.2 IP 防火墙

本论文设计的网关在网络层实现了基于源 IP 地址的防火墙机制。当接收到的请求源 IP 地址被列入黑名单时，网关将阻止其 TCP 连接的建立，从而实现对恶意流量的屏蔽。

为此，网关内部维护了一个 IP 地址的哈希表，使用 C++ 标准库提供的 std::unordered\_set 来存储和管理被封禁的 IP 地址。该容器在底层通过哈希函数将每个 4 字节的源 IP 地址映射到固定大小的桶数组中，并通过开放寻址或链表等方式处理哈希冲突。在编程层面，IP 地址被存储在一个 uint32\_t 类型的变量中。设插入的新值为 x，哈希表的长度为 n。 C++ 标准库对于这个类型的变量采用的哈希函数H(x) 如下：

3.3.3 OAuth 外部认证（没体现在综述里）

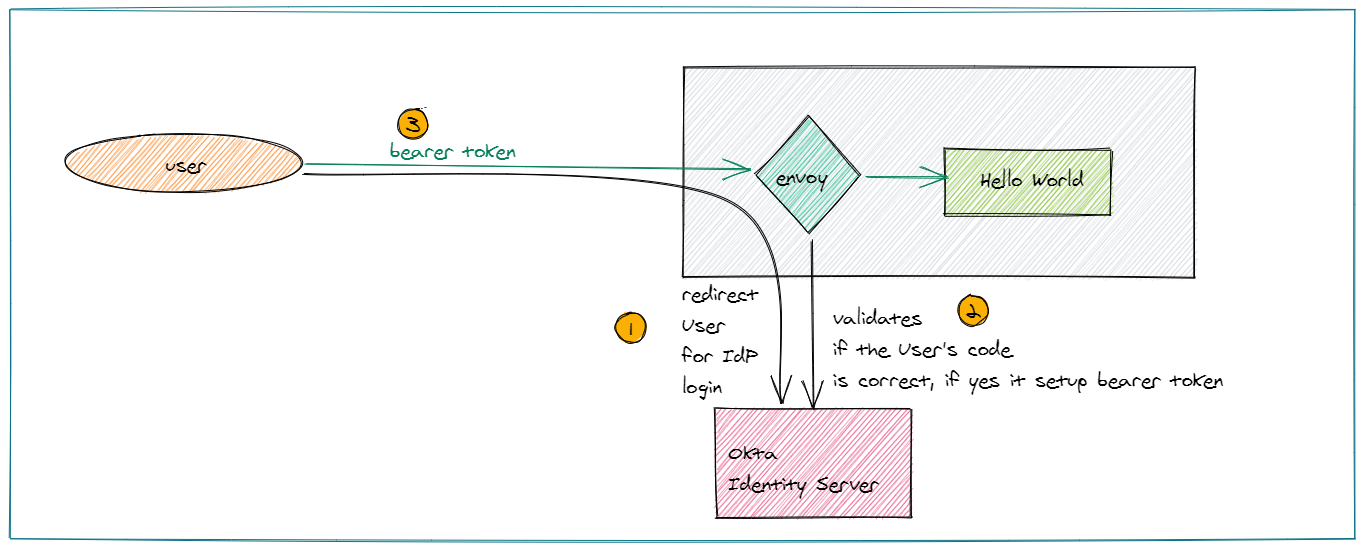
在传统架构中，服务认证往往是由各个业务服务独立实现的，导致认证逻辑分散、安全策略不一致，难以维护和审计。但在微服务体系中，服务数量众多，若每个服务都自行处理认证，将会导致用户身份信息在多个服务主机上冗余存储。这种设计不仅增加了开发成本与维护成本，也有着很大的安全漏洞。

为了解决这一问题，现代生产环境往往会引入外部认证服务器统一存储并管理用户的身份信息。访问应用程序的请求应该先从外部认证服务器中获取身份令牌，之后携带身份令牌访问应用程序。由于在微服务架构中服务数量众多，且服务实例动态改变，若每一个服务都需要在外部认证服务器中先进行注册，将导致难以维护与难以扩展的问题。因此将使用网关统一对接认证逻辑。

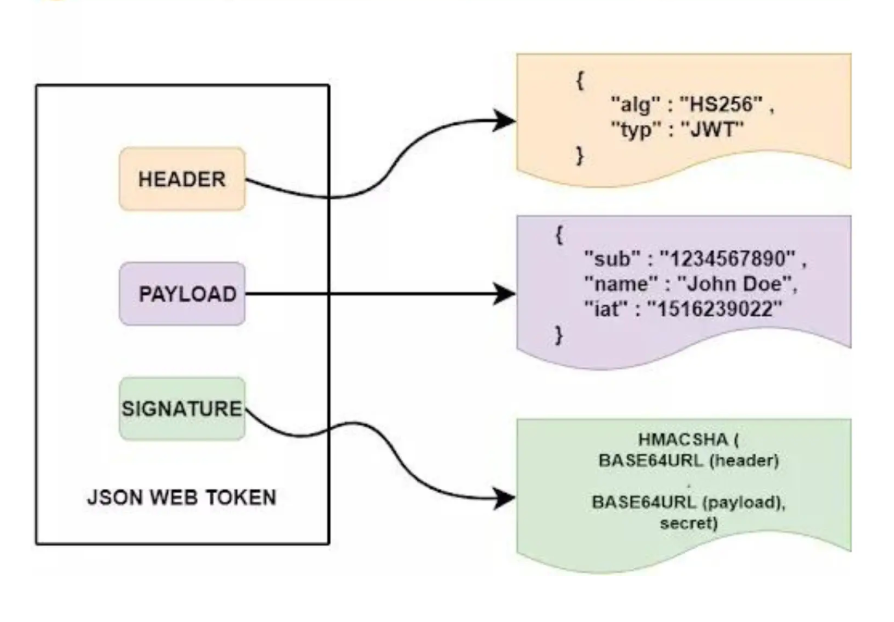
OAuth（Open Authorization）是一个用于资源授权的开放标准协议，最初由 Twitter 等公司提出，目前已广泛应用于 Web 与移动应用程序的第三方访问控制场景中。OAuth 2.0 协议的核心目标在于，使客户端应用程序在不直接接触用户凭据的前提下，通过令牌机制访问用户受保护的资源，从而提升系统安全性与用户隐私保护能力。

本网关在代理 HTTP协议的时候支持通过 OAuth 协议与外部认证服务器集成，统一对接认证逻辑。具体流程如下：

1. 用户访问应用程序前会被网关重定向至外部认证服务器获取 Code
2. 用户向网关提交 Code，网关向外部服务器验证 Code 并获取用户身份信息
3. 若网关成功验证用户身份信息，将返回访问应用程序的需要的访问令牌 (Access Token)。



本论文网关生成的访问令牌采用 JWT（JSON Web Token）格式，这是一种基于 JSON 的轻量级令牌格式，常用于在网络应用中安全地传递身份验证信息。JWT 通常由三部分组成：头部（Header）、载荷（Payload）和签名（Signature），通过数字签名保证令牌的完整性和可信性，客户端可使用该令牌在后续请求中携带用户身份信息，实现无状态的认证机制。一个常见的 JWT 例子如下图所示：



上文所述的这种集中式认证机制不仅增强了系统的安全性和一致性，也简化了下游服务的开发负担，使得服务本身可以专注于业务逻辑，而无需重复实现认证功能。

## 3.4 流量控制

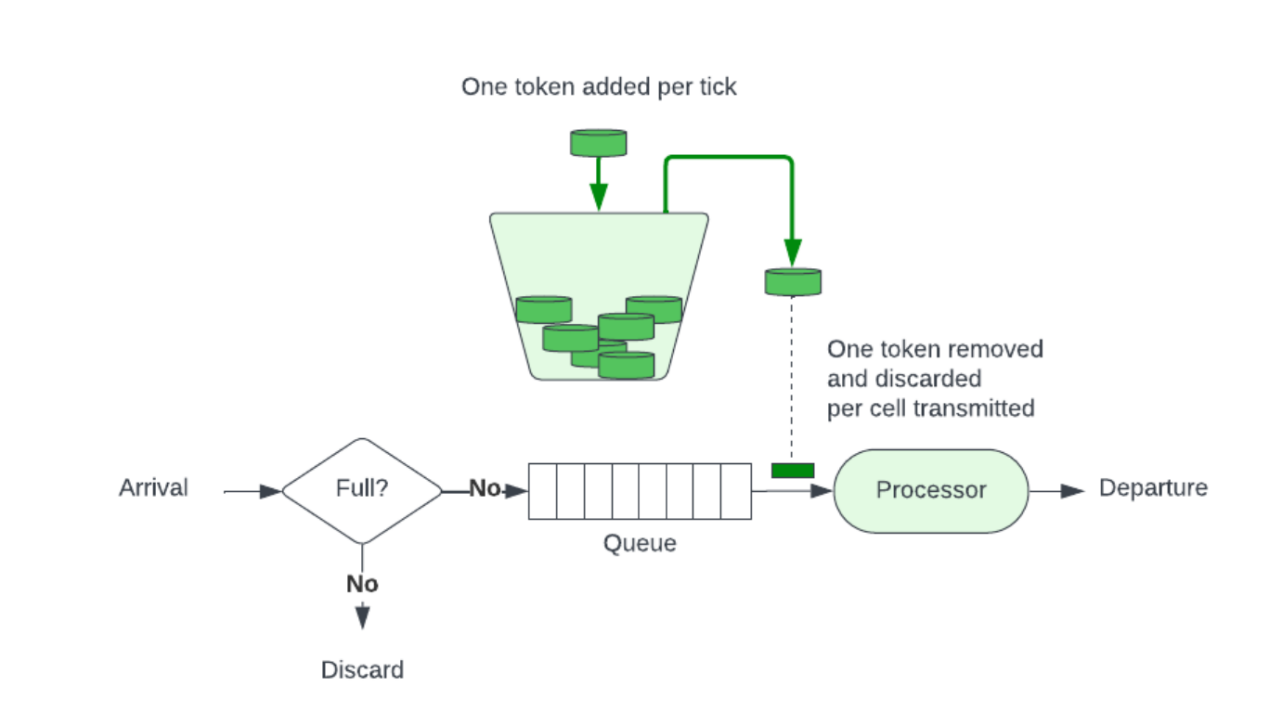
3.4.1 速率控制

速率控制的主要意义在于平衡系统负载，防止因短时间内大量请求的涌入导致系统过载或性能下降。它通过控制请求的处理速率，确保系统在高并发情况下仍能稳定运行。常见的速率控制算法包括令牌桶算法（Token Bucket），漏桶算法（Leaky Bucket）以及滑动窗口（Sliding Window）算法。三种算法的大致流程如下：

1. 令牌桶算法：令牌以固定速率放入桶中，请求到达时消耗令牌，若桶内有令牌则允许请求通过，否则拒绝或延迟。
2. 漏桶算法：请求按固定速率流入桶中，桶以恒定速率流出，若桶满则丢弃请求，确保输出流量平稳。
3. 滑动窗口算法：每个请求记录时间，检查当前时间窗口内的请求数量，若未超出限制则允许请求，通过时间滑动更新窗口。

对上述算法流程进行对比可发现漏桶算法和滑动窗口在遭遇突发的大规模流量时，会将超过容量部分的请求直接丢弃，这样的设计缺乏泛用性，难以应用在网关中。同时滑动窗口算法中的窗口大小较难调整：过小的窗口将导致频繁的请求数检查，占用 CPU 资源。而过大的窗口可能导致流量集中在窗口初始化初期，而窗口剩余的时间流量无法通过。

本论文网关采用了令牌桶算法作为速率控制的实现，由于令牌是按照均匀速率生成，相比起使用滑动窗口算法将获得更平滑的流量。而相比起漏桶算法直接丢弃超过桶容量的请求，使用令牌桶算法的服务器可以配置多余的请求等待新令牌生成后放行或直接被丢弃。令牌桶算法大致工作流程如下图所示：



3.4.2 负载均衡

负载均衡在云原生和云计算系统中的意义在于通过主动配置将流量合理分配到不同的服务实例，确保系统的高可用性和稳定性。通过配置合理的路由策略，负载均衡能够优化资源利用率，避免单点故障，提高系统性能。

本论文网关的负载均衡模块建立在 3.2 节所述的路由匹配模块之上，该模块计算得出一个目标主机的集合，并在这个集合中进行轮询并取出一个目标地址，随后处理流程进入代理模块。通过不同目标地址的分配，网关实现了基本的负载均衡功能。

## 3.5 云原生功能支持

3.5.1 概述

随着云计算和微服务架构的发展，现代网关系统不仅需要具备基本的流量转发和路由能力，还应支持一系列云原生特性，以满足在弹性伸缩、服务治理、安全认证及自动化运维等方面的需求。本节将介绍该网关在云原生场景下的功能支持情况，包括用于服务健康检测的心跳机制，便于远程配置和管理的 gRPC 接口，以及基于 Docker 的容器化部署方式。这些能力共同构成了网关在云环境下稳定运行、易于集成与扩展的技术基础。

3.5.2 gRPC 远程管理接口

在现代分布式系统中，管理和监控功能对确保应用的高效运行至关重要，特别是在微服务架构和网关的场景下。系统的管理接口提供了一种统一的方式，能够实时监控服务的健康状况、性能数据以及配置状态等关键信息。

gRPC（Google Remote Procedure Call）是一种高性能且开源的远程过程调用（RPC）框架，它基于 HTTP/2 协议，并使用 Protobuf（Protocol Buffers）作为接口定义语言。与传统的 RESTful API 相比，gRPC 在传输过程中提供了更低的延迟和更高的吞吐量。通过二进制格式的数据传输，gRPC 避免了文本格式（如 JSON）带来的性能开销，从而使网络通信更加高效。此外，gRPC 生态支持多种编程语言，开发者无需过多关注底层网络细节，可以专注于应用逻辑的实现，从而实现跨平台、跨语言的高效网络通信。

在本论文中，网关的各项配置参数不仅可以通过本地文件进行配置，还能够通过 gRPC 接口实现远程配置。通过 gRPC 生态系统中的服务反射（Service Reflection）功能，调用网关管理接口的程序可以自动快速地获取接口相关信息。与传统的互联网开发中需要事先编写复杂 API 说明文档的方式相比，这种设计显著提升了自动化开发效率，同时大大降低了后续接口维护的成本。本网关提供的 gRPC 管理接口如下表所示：

|  |  |
| --- | --- |
| 接口名称 | 说明 |
| api.v1.GetConfig | 获取当前配置相关信息 |
| api.v1.UpdateConfig | 对基本配置（例如开启 HTTP 压缩，TLS 连接，速率控制）进行更新 |
| api.v1.GetIPBlackList | 获取 IP 黑名单 |
| api.v1.UpdateIPBlackList | 设置 IP 黑名单 |
| api.v1.GetRouter | 获取路由设置 |
| api.v1.UpdateRouter | 更新路由设置 |
| api.v1.RegisterHealthz | 注册心跳检测服务 |

3.5.3 心跳检测（没体现在综述里）

在分布式系统中，心跳检测是确保各个组件健康、服务可用性的关键机制之一。随着微服务架构和网关的普及，服务之间需要频繁地进行通信，确保互相之间的健康状态可以被及时检测到。如果某个服务因为故障或异常中断，心跳检测能够帮助网关及时发现并切换到其他健康的服务，从而提高系统的可靠性和可用性。

现代应用中的心跳检测通常有两种实现方式：一种是基于 TCP 连接的保活机制，另一种是基于应用层协议（如 HTTP 或 WebSocket）自行实现的心跳检测机制。基于 TCP 保活机制的检测方式不需要额外的系统模块，但它仅能保证 TCP 连接的存活，并不能完全反映应用程序的健康状态。而基于应用层协议的心跳检测需要应用程序统一实现心跳检测接口，以供网关进行定期检查。

本论文中的网关心跳检测实现将基于应用层协议，并采用 gRPC 协议。服务端需要根据预定义的接口实现心跳检测功能，并通过网关配置文件进行注册。在生产环境中，网关与服务端通过 gRPC 长连接进行通信，网关定期向服务端发送心跳请求，以检测其健康状态。如果服务端未能在规定的时间内响应心跳请求，网关将视该服务为不可用，并采取相应的故障处理措施。具体 gRPC 接口定义如下：

message HealthCheckRequest {

string service\_name = 1;

}

message HealthCheckResponse {

// 服务健康状态

bool healthy = 1;

// 提供的附加信息（如错误信息）

string message = 2;

}

// 健康检查服务定义

service HealthCheck {

rpc Check (HealthCheckRequest) returns (HealthCheckResponse);

}

3.5.4 Docker 容器化

在现代分布式系统和微服务架构中，服务的快速部署、跨平台迁移以及环境一致性是应用运行和维护的核心挑战。传统的部署方式通常依赖手动配置和环境管理，导致开发、测试、生产环境中常常出现配置差异和兼容性问题，从而影响服务的稳定性和可维护性。尤其在多云或混合云环境中，如何确保应用能够一致地在不同环境中运行，成为系统设计中的一大难题。

为了解决这些问题，Docker 容器化技术提供了高效且便捷的解决方案。通过将网关及其依赖打包到独立的容器中，Docker 确保了应用程序的环境一致性，开发者无需关心底层环境的配置差异。在本论文中，网关将通过 Docker 容器化运行，实现快速部署，并确保在不同云平台和环境中的高效运作，从而保证跨平台兼容性和环境一致性。

为了支持 Docker 工具构建应用程序的运行环境，需要编写 Dockerfile。在本论文的实现中，网关将使用 Dockerfile 的多阶段构建功能：一个容器用于编译网关的二进制文件，另一个容器则用于运行已编译的二进制文件。在程序分发阶段，仅需对后者进行分发。其中本论文网关使用的 Dockerfile 如下：

Docker 后台将根据此文件先进行程序构建工具与依赖第三方库的下载，之后进行代码编译，最后将二进制文件拷贝到最终分发的目标容器中。

## 3.6 本章小结

本章节讨论了三种常见 I/O 架构（每请求一连接，多路复用，反应堆）的优劣并进行性能测试对比，最终选择反应堆作为网关的最终架构。接着讨论了网关对于常见协议（TCP，HTTP，WebSocket）的解析与代理，其中包括了路由匹配模块的设计，并讨论了 TLS 连接与 HTTP 内容压缩的优化。接着从流量过滤，速率控制，负载均衡三个方面讨论了网关对流量控制的实现。这些内容支撑起了网关的基本架构。最后针对云原生相关功能，说明了本网关中对外部认证，gRPC 远程管理接口，心跳检测与 Docker 容器化等高级功能的实现。

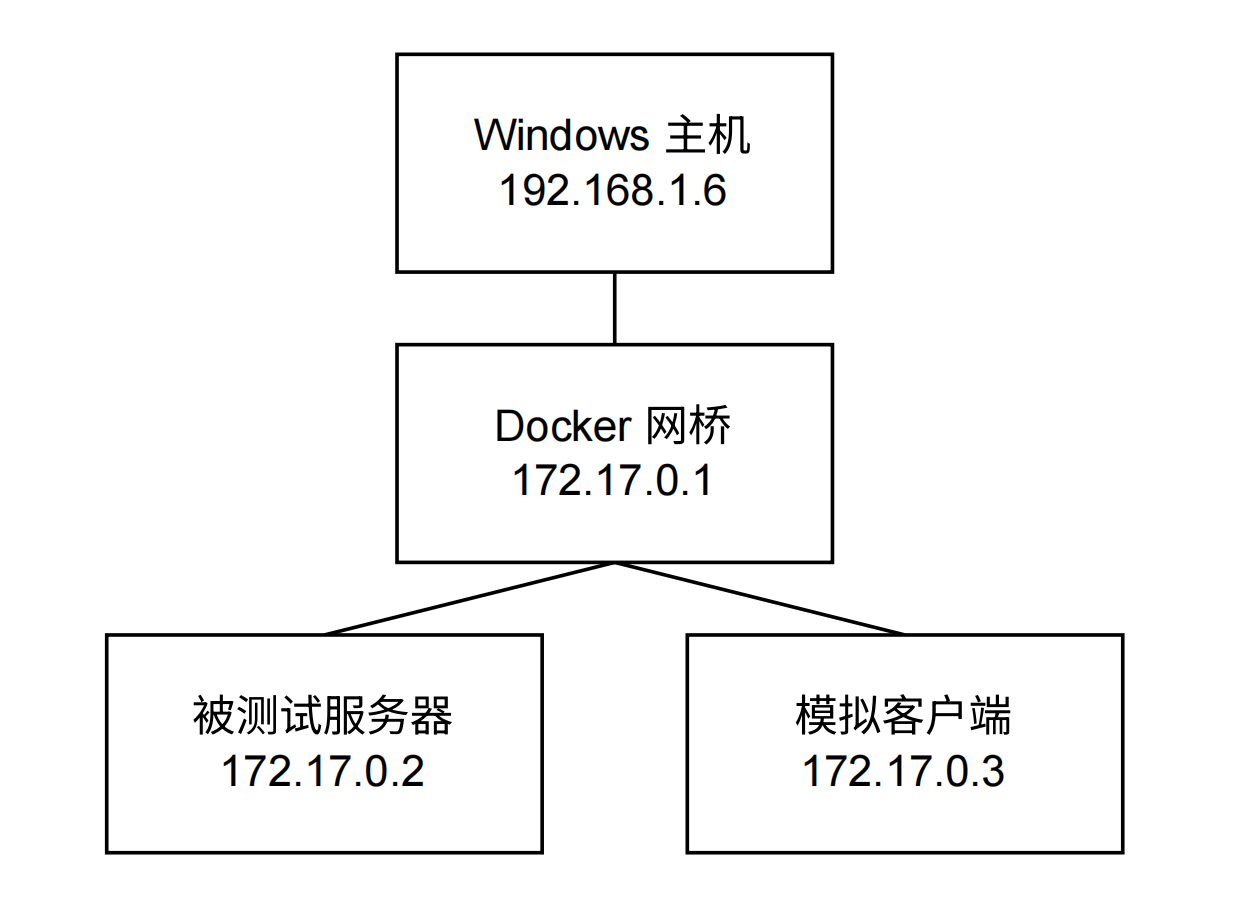
# 第4章 系统测试与评估

## 4.1 测试工具与环境

本章节使用测试工具如下表所示：

|  |  |
| --- | --- |
| 类别 | 版本号 |
| Operating system | Ubuntu 24.04 |
| CPU | 12th Gen Intel(R) Core(TM) i5-12600K |
| wrk | 2.1.1 |
| Docker | 27.3.1 |
| Google Chrome | 135.0.7049.85 |
| Python | 3.13 |
| Golang | 1.22.1 |
| C++ | 17 |
| CMake | 3.25.0 |
| vcpkg | 8.9.2 |
| Nginx | 1.27.4 |
| Postman | 11.40.6 |
| Keycloak | 26.1 |
| curl | 8.7.1 |
| Bytebase | 3.5.2 |

测试环境网络拓扑图如下所示：



为了创建隔离的网络环境，同时更好地限制被测试服务器的资源，本测试使用 Docker 容器运行被测试服务器以及模拟客户端。

Docker 工具运行时，其后台守护进程会在宿主机构建一个虚拟网桥用于宿主机以及 Docker 网络的连接。由图可知被测试服务器以及模拟客户端都连接在 IP 地址为 172.17.0.1 的网桥上进行网络通信。

同时在本次实验中限制被测试服务器仅能使用 2 个宿主机 CPU 核心。

## 4.2 系统性能测试与功能验证

这里应该放网页截图，然后还有文件具体包含哪些？

4.2.1 整体性能测试

为评估本网关 Azugate 在静态资源代理方面的性能，选取某登录界面 login.html 作为测试目标。该页面由多个静态资源（如 CSS、JavaScript 等）组成。图 4.1 展示了该网页的实际界面，表 4.2 给出了该页面所涉及的资源及其大小。

使用 4 线程模式的 wrk 压力测试工具对 Azugate、Nginx、Python SimpleHTTPServer 和 Golang net/http 等常见服务器或框架进行对比测试，得到的终端输出结果如下图所示。

从图中分析吧啦吧啦吧

将每秒成功处理的请求数（QPS）绘制成柱状图，如下所示：

Azugate 在本次测试中表现出良好的性能。在多线程模式下，其 QPS 达到工业级网关 Nginx 的约 92%，为 Golang net/http 库的约 144%。在单线程场景中，Azugate 的性能为 Python SimpleHTTPServer 的约 30 倍，展现出显著的性能优势。

4.2.2 核心功能验证

（1）应用代理示例

本小节测试网关对于 HTTP 与 WebScoket 协议的代理能力。选取一个什么应用同时具有 WebSocket 与 HTTP 功能进行测试。

（2）HTTP 内容压缩与 TLS 连接测试

本实验通过 curl 工具模拟客户端访问场景，借助 Azugate 的 gRPC 管理接口动态启用或关闭 HTTP 内容压缩与 TLS 支持，以评估网关在安全性与传输效率方面的灵活配置能力。测试结果如下图所示 a～c 所示：

（3）速率控制

为验证 Azugate 的速率控制能力，本实验使用 wrk 工具对 “Welcome to Azugate” 页面发起高并发请求。通过调用 Azugate 提供的 gRPC 管理接口，动态开启目标路径的速率控制功能，并对比启用前后的请求处理速率，从而验证系统限流机制的有效性。测试结果如下图 a～c 所示：

实验过程中，Azugate 通过 gRPC 管理接口动态配置对目标路径的速率限制。在未启用速率限制时，Azugate 全量转发请求；启用限制后，部分请求根据设定的速率被限制，表现为 wrk 实际 QPS 降低。

1. 负载均衡

本小节通过 Azugate 的 gRPC 管理接口，动态添加 HTTP 路由配置，实现将对路径 “/” 的访问请求，平均分发至本地不同端口上的三个网页代理服务器。系统结构如下所示：

测试结果如下图 a～d 所示：

1. IP 防火墙

为验证 Azugate 的 IP 防火墙功能，本实验使用其 gRPC 管理接口动态添加指定黑名单 IP，并观察被封禁用户访问资源时的响应情况。通过对比设置前后的请求结果，评估访问控制机制的有效性，如下图所示。

1. OAuth 外部认证

为验证 Azugate 的外部认证功能，实验构建了一个以 Keycloak 为身份验证服务的环境。用户首次访问受保护资源时将被重定向至认证界面，认证通过后返回所请求资源页面 “Welcome to Azugate”。认证过程如图 a～d 所示。

（6）心跳检测

为验证 Azugate 的心跳检测功能，本小节构建了包含多个后端服务节点的代理集群，使用 Postman 工具模拟应用向 Azugate的 gRPC 管理接口发送注册心跳检测，接着通过关闭其中一个节点的方式模拟服务故障。Azugate 能够在检测到节点不可达后，及时进行预警，从而保障系统的可用性与稳定性。测试流程如下图 a ～ d 所示：

# 总结与展望

本文围绕 Azugate 网关系统的设计与实现展开，重点探讨其在基础代理功能与云原生架构适配方面的关键技术路径与性能优化策略。

在基础代理功能方面，Azugate 实现了对 TCP、HTTP 及 WebSocket 协议的统一代理支持，并具备 TLS 连接的协商与建立能力。系统自研的高性能协议解析器，统一处理 HTTP 与 WebSocket 协议，有效提升了协议栈处理的一致性与效率。

为进一步优化转发性能与资源利用率，Azugate 引入了包括零拷贝传输机制、基于 Gzip 的 HTTP 内容压缩策略，以及基于前缀树结构的高效路由匹配算法等多项关键技术手段。

此外，系统还集成了速率控制、负载均衡与 IP 层防火墙等核心能力，显著增强了其在复杂网络环境中的适应性与扩展性。

在云原生环境适配方面，Azugate 提供基于 gRPC 的动态管理接口，支持配置热更新；兼容 OAuth 协议以对接外部认证服务器，实现统一访问控制；通过内置心跳机制提升故障感知与服务可用性；同时具备良好的容器化部署能力，便于与微服务体系及服务网格架构深度集成。

性能评估结果表明，Azugate 在多线程环境下的请求处理能力达到 Go net/http 的 144%、Nginx 的 92%；在单线程场景下，其性能为 Python HTTPServer 的 30 倍，充分验证了其在高并发负载下的工程实用性与性能潜力。

尽管 Azugate 在性能和功能方面展现出良好表现，但与成熟网关相比仍存在以下较明显的不足：

1. 缺乏缓存机制：系统尚未设计静态资源缓存或反向代理缓存层，无法在高频访问场景中减轻后端压力，影响整体响应效率。这也是 Azugate 与 Nginx 性能差距的一大重要原因。

（2）不支持 HTTP 多路复用与长连接：当前仅实现 HTTP/1.1标准 ，缺乏对 HTTP/2 等现代协议的支持，也未启用 Keep-Alive 等连接复用机制，不利于高并发场景下的资源利用率。值得注意的是，HTTP/2 已在主流浏览器与服务器基础设施中广泛应用，若长期不支持该协议，将不利于系统的兼容性与性能发展。

（3）跨平台优化支持不足：系统在性能优化中使用了如 sendfile() 这样的 Linux 特定系统调用，在提升传输效率的同时，也限制了系统在其他平台（如 Windows 和 macOS）上的可移植性。目前尚未针对不同操作系统提供替代的优化方案，例如在 Windows 中可使用 TransmitFile()，在 macOS 中可使用 sendfile() 的 BSD 实现等。因此，系统在跨平台部署时，性能优化能力尚不具备一致性。

综上，Azugate 虽在基本功能和性能表现上已具备一定竞争力，但若要发展为可应用于实际生产环境的高性能网关系统，仍需在跨平台性、协议支持、性能优化等多个方面进行系统性提升。

# 参考文献

1. 宋震, 张宇姝, 杨刚. 人体动作识别与评价综述[J]. [中国传媒大学学报(自然科学版), 2021, XX(6): 61-65](https://journal.cuc.edu.cn/frmPdfShow.aspx?id=2042" \t "_blank)
2. 施克权,李祺,隋皓等.IEMAyoloViT：基于改进YOLOv8的水下目标检测算法[J/OL].电讯技术,1-10[2024-02-23].
3. 曹雨淇,徐慧英,朱信忠等.基于YOLOv8改进的打架斗殴行为识别算法：EFD-YOLO[J/OL].计算机工程与科学,1-14[2024-02-23].
4. 朱强军,胡斌,汪慧兰等.基于轻量化YOLOv8s交通标志的检测[J/OL].图学学报,1-10[2024-02-23].
5. 韩强. 面向小目标检测的改进YOLOv8算法研究[D].吉林大学,2024.DOI:10.27162/d.cnki.gjlin.2023.001647.
6. 刘莫尘,褚镇源,崔明诗等.基于改进YOLO v8-Pose的红熟期草莓识别和果柄检测[J].农业机械学报,2023,54(S2):244-251.
7. 丁晓慧,周磊.Wi-Fi动作识别在室内入侵检测中的应用[J].科技创新与应用,2023,13(24):64-67.
8. 黄倩,崔静雯,李畅.基于骨骼的人体行为识别方法研究综述[J/OL].计算机辅助设计与图形学学报,1-22[2024-02-23].
9. 王茹,刘大明,张健.Wear-YOLO：变电站电力人员安全装备检测方法研究[J/OL].计算机工程与应用,1-13[2024-02-23].
10. 陈俊颖,郭士杰,and 陈玲玲.基于解耦注意力与幻影卷积的轻量级人体姿态估计.计算机应用 1-13.
11. 周珍玉,秦学,蔡芳等.基于人物交互的学生课堂行为识别研究[J].现代教育技术,2024,34(02):53-61.
12. 韩康,李敬兆,陶荣颖.基于改进YOLOv7和ByteTrack的煤矿关键岗位人员不安全行为识别[J].工矿自动化,2024,50(03):82-91.DOI:10.13272/j.issn.1671-251x.2024030015.
13. 汪洋继鸿,张路,于越,等.一种轻量化三维人体姿态估计算法[J].通信与信息技术,2024,(02):32-35+41.
14. 赵小虎,王轲鑫,孟献峰,等.基于人体姿态估计的羽毛球动作评估方法研究[J/OL].华中科技大学学报(自然科学版),1-9[2024-04-21].https://doi.org/10.13245/j.hust.240261.
15. 王燕妮, et al.多尺度和多层级特征融合的人体姿态估计.计算机工程与应用 1-12.
16. Karácsony T ,Jeni A L ,Torre L D F , et al.Deep learning methods for single camera based clinical in-bed movement action recognition[J].Image and Vision Computing,2024,143104928-.
17. Weiguang Jiang,and Lieyun Ding.Unsafe hoisting behavior recognition for tower crane based on transfer learning. Automation in Construction 160.(2024) :105299-.
18. Xinpeng Yin, et al.Spatiotemporal Progressive Inward-Outward Aggregation Network for skeleton-based action recognition.Pattern Recognition 150.(2024):110262-.
19. Wang C ,Wang J ,Xu W .Double branch synergies with modal reinforcement for weakly supervised temporal action detection[J].Journal of Visual Communication and Image Representation,2024,99104090-.
20. Solimani F ,Cardellicchio A ,Dimauro G , et al.Optimizing tomato plant phenotyping detection: Boosting YOLOv8 architecture to tackle data complexity[J].Computers and Electronics in Agriculture,2024,218108728-.
21. Wang Y .Abnormal behavior identification of enterprise cloud platform financial system based on artificial neural network[J].Computers and Electrical Engineering,2024,115109110-.
22. Liu Q ,Liu Y ,Lin D .Revolutionizing Target Detection in Intelligent Traffic Systems: YOLOv8-SnakeVision[J].Electronics,2023,12(24):
23. Zheng W ,Yan L ,Siyuan D , et al.An efficient detection of non-standard
24. Shizhong Y ,Wei W ,Sheng G , et al.Strawberry ripeness detection based on YOLOv8 algorithm fused with LW-Swin Transformer[J].Computers and Electronics in Agriculture,2023,215
25. Hang Yu, et al.Research on an Intelligent Identification Method for Wind Turbine Blade Damage Based on CBAM-BiFPN-YOLOV8.Processes 12.1(2024):
26. Tomenotti F F ,Noceti N ,Odone F .Head pose estimation with uncertainty and an application to dyadic interaction detection[J].Computer Vision and Image Understanding,2024,243103999-.
27. Zhou L ,Chen Y ,Wang J .SlowFastFormer for 3D human pose estimation[J].Computer Vision and Image Understanding,2024,243103992-.
28. Xu X ,Liu L ,Yan S .SMPLer: Taming Transformers for Monocular 3D Human Shape and Pose Estimation.[J].IEEE transactions on pattern analysis and machine intelligence,2024,46(5):3275-3289.
29. Yang J G ,Kim H J ,Lee W S .Geometry-driven self-supervision for 3D human pose estimation[J].Neural Networks,2024,174106237-.
30. Chengang D ,Guodong D .An enhanced real-time human pose estimation method based on modified YOLOv8 framework[J].Scientific Reports,2024,14(1):8012-8012.

# 致谢

到此，整篇论文的内容就结束了，这也意味着本科生活就即将画上句号，我在完成本次毕业设计的过程当中也收获颇丰，在这里我也十分感谢指导我完成整篇毕业论文的任子良老师。

回想起这大学四年的学习生涯，我也不禁十分感慨，在这四年里，我觉得自己活的忙碌且充实。在同学、老师的帮助下，我不光学习了许多的专业知识，也学会了许多处理问题的解决方法，在完成毕业设计的时候我也遭遇过许多的困境，但最终还是在自己的不断探索下解决了大部分的问题，在这里我不光要感谢自己的辛勤付出，感谢四年以来同学们的陪伴，也要感谢老师、学校对我的栽培，正是在你们的支持下，我才能成长为一名合格的毕业生。

至此，我也谨以本文，来作为我大学本科四年生活的终点，亦为我人生旅途的一个新的起点。