东 莞 理 工 学 院

本 科 毕 业 论 文

|  |  |  |
| --- | --- | --- |
| 毕业设计题目： | 网关 | |
| 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，包含路由匹配模块的设计与 TLS 连接与 HTTP 内容压缩的支持。
3. 实现流量的过滤、速率控制与负载均衡。
4. 实现 OAuth外部身份验证（微服务的访问控制），心跳检测（动态管理）、gRPC 管理接口（自动化配置）与 Docker（容器化，移植性）等云原生功能。

**关键词：**云原生网关，高性能服务器，操作系统

云原生（Cloud Native） 是一种架构理念，旨在利用云计算的 弹性、分布式特性和自动化能力，构建 高可扩展、易维护和高可靠 的应用。云原生架构微服务化、容器化、动态化、自动化等特性能高效地适应现代复杂业务需求。

**目录（这里页码是乱的）**

[第1章 引言 1](#_Toc164630450)

[1.1背景介绍 1](#_Toc164630451)

[1.2问题陈述 1](#_Toc164630452)

[1.3研究目标和意义 2](#_Toc164630453)

[1.4论文的主要工作 3](#_Toc164630454)

[1.5章节安排 3](#_Toc164630455)

[第2章 相关技术和理论模型 5](#_Toc164630456)

[2.1深度学习相关理论 5](#_Toc164630457)

[2.2姿态提取模块 12](#_Toc164630458)

[2.3相关开发环境 13](#_Toc164630459)

[2.4本章小结 15](#_Toc164630460)

[第3章 系统设计与实现 16](#_Toc164630461)

[3.1系统设计整体框架 16](#_Toc164630462)

[3.2 I/O 模型的对比与选择 17](#_Toc164630463)

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

[3.4 流量控制 18](#_Toc164630464)

[3.4.1 速率控制 20](#_Toc164630467)

[3.4.2 负载均衡 18](#_Toc164630466)

[3.4.3 流量过滤 20](#_Toc164630467)

[3.5 云原生功能支持 22](#_Toc164630468)

[3.5.1 OAuth 外部认证 22](#_Toc164630469)

[3.5.2 心跳检测 24](#_Toc164630470)

[3.5.2 gRPC 管理接口 24](#_Toc164630470)

[3.5.2 Docker 容器化 24](#_Toc164630470)

[3.6本章小结 30](#_Toc164630471)

[第4章 系统测试与评估 31](#_Toc164630472)

[4.1实验结果展示 31](#_Toc164630473)

[4.2模型准确度评估 33](#_Toc164630474)

[4.3系统存在的缺陷 38](#_Toc164630475)

[4.4本章小结 39](#_Toc164630476)

[第五章 总结与展望 40](#_Toc164630477)

[参考文献 41](#_Toc164630478)

[致谢 44](#_Toc164630479)

# 第3章 系统设计与实现

## 3.1 概述

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。

## 3.2 I/O 架构的测试与选择

3.2.1 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：

3.2.2 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：

3.2.3 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。

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。

3.2.4 三种 I/O 架构性能测试

一个系统的吞吐量通常用 QPS 来进行描述，其计算公式为：

在不同的并发压力下，我们可以得到三种 I/O 架构的 QPS 如下图所示：

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 报文中的路由 （ url ）字段进行匹配，并在改写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在处理HTTP与Websocket 数据时的代理机制，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

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。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”。两种模式工作流程如下图所示：

TLS 连接的建立通常包含一个握手过程。在握手阶段，客户端与服务端首先交换支持的加密算法及参数，然后服务端通过数字证书向客户端证明其身份（通常是基于 CA 的公钥证书），随后双方使用非对称加密完成密钥协商，并最终协商出用于后续通信的对称加密密钥。握手完成后，双方就进入了加密通信阶段。详细的 TLS 建立过程如下图所示：

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）。另外在编程实现层面，TLS 握手通常通过调用 OpenSSL 库实现。

3.2.4 HTTP 代理与内容压缩

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，与正则表达式匹配实现的数据解析性能对比如下：

HTTP/1.1 协议状态机示意图如下：

在完成了请求的基本解析之后，网关不仅需要正确转发数据，还常常承担对报文内容进行处理与优化的职责。其中，内容压缩是一项常见的手段，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，可读字符使用 ASCII 编码：

// 第一次发送 186个字节的数据

BA\r\n[DATA]\r\n\r\n

// 第二次发送 9个字节的数据

9\r\n[DATA]\r\n\r\n

// 第三次发送指示结束的数据块

0\r\n\r\n

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

3.2.5 WebSocket 代理

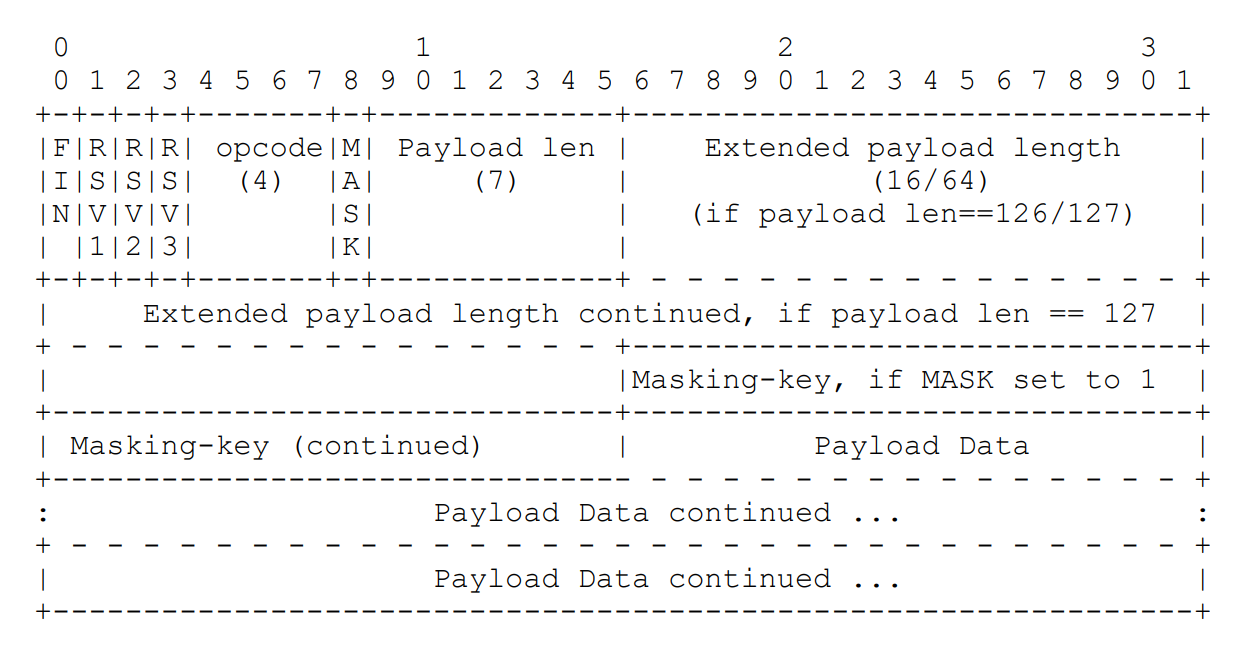
WebSocket 是由 HTTP 协议扩展的全双工应用层协议，广泛应用于在线聊天与多人游戏等需要实时数据传输的场景。为了将 HTTP 协议升级为 WebSocket 协议，需要设定特定的首部字段，并向服务器发送 GET 请求，需要设置的首部如下表所示：

|  |  |
| --- | --- |
| 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 与 Extended payload 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）



## 3.4 流量控制

3.4.1 概述

流量控制是网络网关的重要功能之一，确保在处理大量请求时，系统能够高效、安全地管理流量，避免出现瓶颈或过载问题。本小节将介绍三种常见的流量控制策略：速率控制、负载均衡和流量过滤。这些策略共同作用，能够优化系统的性能、保证服务的可用性，并有效提升网络资源的利用率。

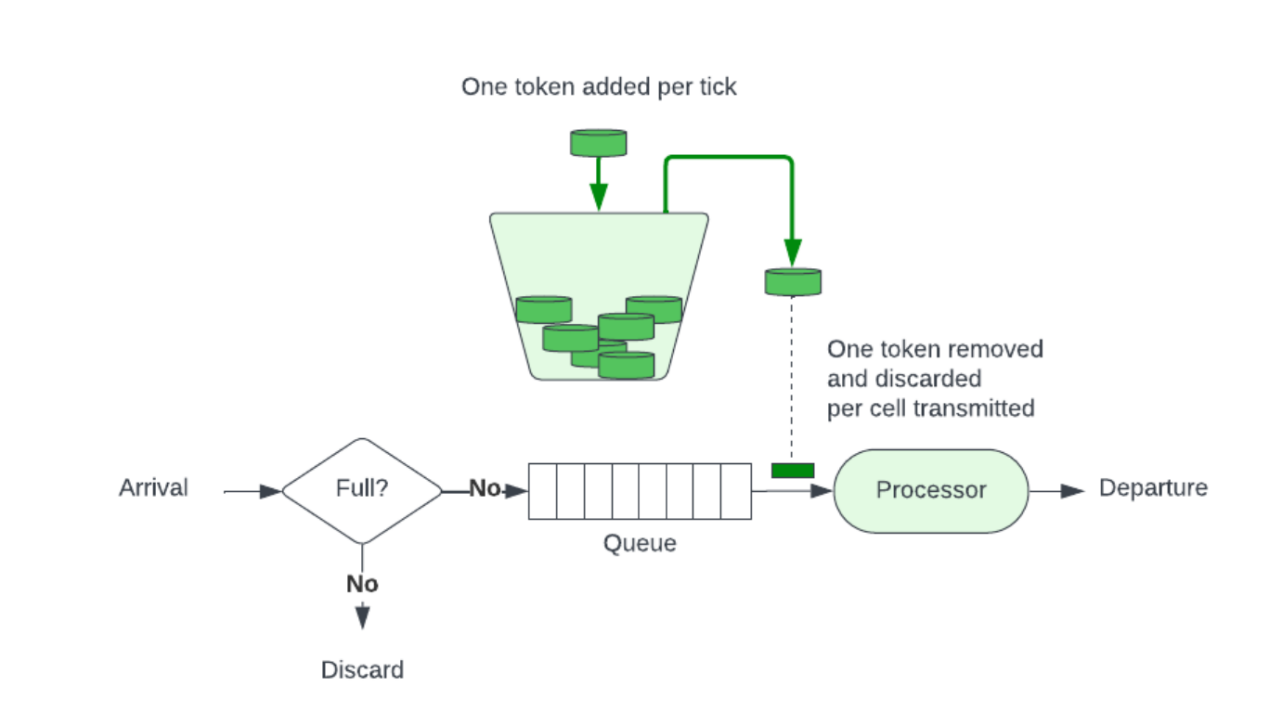
3.4.2 速率控制

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 流量过滤

本论文设计的网关在网络层实现了基于源 IP 地址的请求流量过滤机制。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，C++ 标准库对于这个类型的变量采用的哈希函数如下：

3.4.3 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 节所述的代理模块之上，路由匹配模块计算得出的结果实际上是一个目标主机的集合，负载均衡模块会在这个集合中进行轮询，将流量均匀分配到集合中的每一个主机上。

## 3.5 云原生功能支持

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，包括通过 OAuth 实现的外部认证机制，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。

3.5.1 OAuth 外部认证

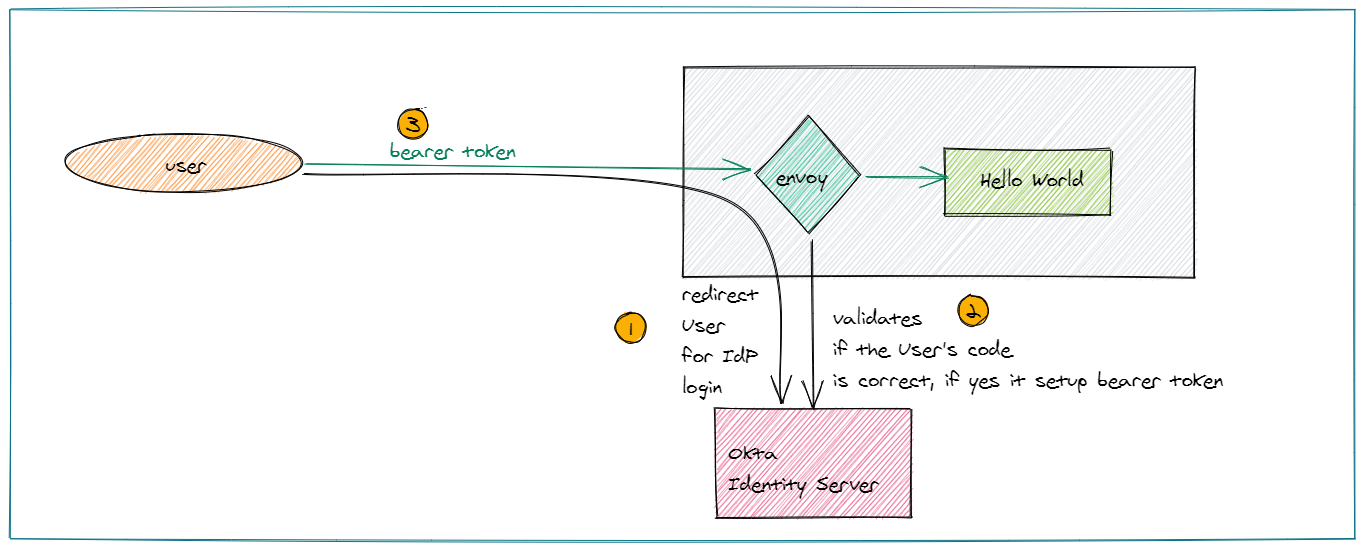
在传统架构中，服务认证往往是由各个业务服务独立实现的，导致认证逻辑分散、安全策略不一致，难以维护和审计。但在微服务体系中，服务数量众多，若每个服务都自行处理认证，将会导致用户身份信息在多个服务主机上冗余存储。这种设计不仅增加了开发成本与维护成本，也有着很大的安全漏洞。

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。

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。

3.5.2 心跳检测