上海交通大学硕士学位论文

VPPNGX：基于 FD.io VPP 的高性能 NGINX 实现

硕士研究生：张泽宇

学　　号：117037910046

导　　师：李健教授

申请学位：专业学位硕士

学　　科：软件工程

所在单位：电子信息与电气工程学院

答辩日期：2020 年 02 月 26 日

授予学位单位：上海交通大学

Dissertation Submitted to Shanghai Jiao Tong University
for the Degree of Master

VPPNGX: FD.IO VPP BASED HIGH-PERFORMANCE NGINX IMPLEMENTATION

Candidate:	Zeyu Zhang
Student ID:	117037910046
Supervisor:	Prof. Jian Li
Academic Degree Applied for:	Master of Professional
Speciality:	Software Engineering
Affiliation:	School of Electronic Information and Electrical Engineering
Date of Defence:	Feb. 26th, 2020
Degree-Conferring-Institution:	Shanghai Jiao Tong University

上海交通大学

学位论文原创性声明

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，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。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。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期： 2020 年 02 月 26 日

上海交通大学

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保密□，在___年解密后适用本授权书。

本学位论文属于

不保密 ☑。

（请在以上方框内打“√”）

学位论文作者签名： 

日期：2020 年 02 月 26 日

指导教师签名： 

日期：2020 年 02 月 26 日

VPPNGX：基于 FD.io VPP 的高性能 NGINX 实现

摘　要

如今日益增长的网络请求给企业级 Web 网站的请求处理带来了巨大的压力。很多企业级 Web 网站使用 NGINX 来处理网络请求。NGINX 可以将网络请求进行处理或者转发至上游服务器进行分流，然后将回复发送回客户端程序。基于原生操作系统的原生 NGINX 依赖于操作系统内核的网络协议栈。然而，内核态网络协议栈中的上下文切换、共享资源竞争等开销限制了网络包处理的性能。于是很多研究都使用用户态网络协议栈来替换内核态网络协议栈以消除内核协议栈中的性能开销。

本论文提出了 VPPNGX——一种基于 FD.io VPP 的针对公网请求处理的高性能 NGINX 实现。FD.io VPP 是一种基于 DPDK 的高性能且拥有丰富功能的用户态包处理框架，可以运行在通用商业 CPU 上，且支持 L2-L7 的用户态网络协议栈。为了将 FD.io VPP 的用户态网络协议栈应用到 NGINX，并解决直接应用 FD.io VPP 时带来的性能问题，我们首先提出了 VPP 会话索引透传机制（VPP Session Index Passthrough）以安全移除 VPP 会话锁层（VPP Session Lock Layer）。VPP 会话锁层负责管理所有用于替代套接字锁的会话锁，且会对 NGINX 的可扩展性和请求处理延迟造成负面影响。其次，我们还为 VPP 事件队列（VPP Event Queue）引入了基于令牌的无锁化保序机制（Token-Based Lock-Free Order-Preserving）来保证在移除了 VPP 会话锁层后，从 NGINX 发向 VPP 的控制事件消息在入队 VPP 事件队列时，消息的元数据不会发生乱序。最后，我们设计了用户态联合阻塞式 epoll 机制（User-Space Unified Blocking epoll）去替换原本 NGINX 使用的忙等轮询式 epoll 机制（Busy-Wait Polling epoll），以在原来能同时接收用户态和内核态 epoll 事件的基础上，实现阻塞功能减少 CPU 占用率，并且减少因进

行内核 epoll 事件检查而造成的上下文切换开销。

基于吞吐量的可扩展性实验表明，VPPNGX 在 RPS（请求数/秒）方面达到了基于内核的原生 NGINX 的 3.33 倍吞吐量。VPPNGX 能够在 RPS 方面达到很好的可扩展性，并且其 RPS 比直接使用 VPP 而没有任何优化的 NGINX 的 RPS 还要高。延迟实验表明，VPPNGX 的请求处理延迟比 F-Stack NGINX、直接使用 VPP 的 NGINX 还有基于内核的原生 NGINX 要分别低 46.7%、34.1% 和 25.3%。CPU 占用率实验和内核 epoll 事件检查开销实验表明，VPPNGX 也能够在网络负载较低时减少 CPU 占用率，并减少因进行内核 epoll 事件检查而造成的上下文切换开销。

关键词：FD.io VPP, 用户态 TCP/IP 协议栈, NGINX, 网络请求处理

VPPNGX: FD.IO VPP BASED HIGH-PERFORMANCE NGINX IMPLEMENTATION

ABSTRACT

The ever-increasing number of network requests exert pressure on enterprise-level websites and the bulk of websites use NGINX to handle requests, processing or offloading requests to upstream servers and sending back responses to clients. Native NGINX relies on kernel networking stack. However, the inefficiency of kernel networking stack limits packet processing performance. Hence prior studies use user-space networking stack to replace kernel networking stack with the aim of overcoming the inefficiency of kernel stack.

This work presents VPPNGX, a high-performance NGINX implementation for public network request processing. VPPNGX is based on FD.io VPP, which is a DPDK-based high-performance, fully-featured user-space packet processing framework supporting L2-L7 user-space networking stack to run on commodity CPUs. In order to employ the networking stack of FD.io VPP to NGINX and solve performance problems that result from directly employing FD.io VPP to NGINX, we first propose VPP session index passthrough approach to removing VPP session lock layer that is designed to replace socket lock in kernel and that has negative impacts on scalability and latency of NGINX. Next, after removing session lock layer, for VPP event queue we propose a token-based lock-free order-preserving approach to preserving the order of metadata of control event messages sent from NGINX to VPP when they are enqueued into VPP event queue. Finally, we design a user-space unified blocking epoll scheme to replace the original busy-wait polling epoll when receiving user-space and kernel epoll events together, in

order to empower our VPPNGX to reduce CPU usage via blocking mechanism and reduce context switch for kernel epoll event checking.

Consequently, the scalability test on throughput demonstrates that VPPNGX can achieve 3.33x higher throughput in RPS (requests per second) than kernel-based NGINX. It is capable of scaling up requests per second well and better than original VPP-based NGINX. The latency test shows that the latency of VPPNGX is 46.7%, 34.1%, and 25.3% lower than that of F-Stack NGINX, original VPP-based NGINX, and kernel-based NGINX. Tests on CPU usage and overhead of kernel epoll event checking demonstrate that VPPNGX is also able to reduce CPU usage when load is low as well as context switch for kernel epoll event checking.

KEY WORDS: FD.io VPP, User-Space TCP/IP Stack, NGINX, Network Request Processing

目 录

插图索引

表格索引

主要符号对照表

VPP	Vector Packet Processing（向量包处理）
LDP	LD_PRELOAD（动态库预加载）
VCL	VPP Communication Library（VPP 通信库）
VLS	VCL Locked Session（VPP 通信库带锁会话）
FD	File Descriptor（文件描述符）
FIFO	First In, First Out（先入先出队列）
IPC	Inter-Process Communication（进程间通信）
epoll	Event Poll（事件轮询）
AF_INET	Address Family Internet（英特网地址族）
RPS	Requests per Second（每秒请求数）
C10K	Concurrency 10000（10000 并发量问题）
i-cache	指令缓存
d-cache	数据缓存
RX	数据接收队列
TX	数据传输队列
TSO	TCP Segmentation Offloading（TCP 报文分割分流）
DMA	Direct Memory Access（直接内存访问）
condvar	Condition Variable（条件变量）

第一章 绪论

1.1 背景简介

随着如智能手机、智能手表、智能电视等网络终端设备数量的不断增长，主流企业级网站的访问数量也在巨幅增长。巨大的网站访问数量给企业级网站处理网络请求带来了不小的压力。于是，很多企业网站使用“负载均衡”[1-5]方法将接收到的网络请求分流并转发给上游（后端）服务器进行并行化的处理。这些上游服务器一般都同属一个数据中心。在这一过程中，反向代理[6-10]软件或硬件负责接收所有客户端的网站访问请求，并将这些请求均衡地分发到不同的上游服务器进行处理。反向代理软件或硬件再接收所有上游服务器的回复，并将回复发送回各个客户端，就好像是反向代理软件或硬件自己处理了所有的网络请求一样。

在所有的反向代理软件中，NGINX[6, 11, 12]是一个被广泛使用的代理服务软件。NGINX 本身是一个静态 Web 服务软件，但是它同时还是一个软件化的拥有带内健康检查（In-Band Health Check）[13]的负载均衡器。NGINX 起初被编写出来的目的是为了在性能上超越 Apache Web Server[14]。NGINX 现在正被如 Facebook、GitHub、淘宝、Dropbox 等知名企业用作软件化负载均衡器，负责在数据中心边缘处接收并转发来自公网的巨量网站访问请求到上游服务器进行处理[15, 16]。

传统上，直接在 Linux 操作系统上运行的原生 NGINX 依赖于 Linux 内核的网络协议栈。而先前的研究表明，商业操作系统内核拥有很多影响 TCP/IP 协议栈网络包处理性能的缺陷[17-23]。比如，系统调用的开销、为每个网络数据包分配或回收重量级的数据结构比如“sk_buff”[24-26]、还有硬件中断开销等都会影响网络包处理的效率。另外，操作系统内核里的通用资源共享，也会导致多核系统上资源竞争的情况发生[24, 27]。

1.2 内核态 TCP/IP 协议栈相关优化介绍

为了能够解决上述内核态 TCP/IP 协议栈中的这些缺陷，有很多有关性能优化的方法策略被提出。它们大致可以分为以下四类：

- **内核态 TCP/IP 协议栈优化**：有很多研究直接优化了操作系统内核的 TCP/IP 协议栈。比如有 MegaPipe[24]、StackMap[28]、FlexSC[29]、FastSocket[30]以及 Affinity-Accept[27]。他们都使用了 Zero Copy（零拷贝）[31-33]去消除 TCP/IP 协议栈中数据拷贝引起的开销。

其中，MegaPipe 和 StackMap 使用 Zero Copy 的方法减少 TCP/IP 协议栈中网络包拷贝的开销。但是 MegaPipe 和 StackMap 都需要对网络应用程序的源代码进行修改。而 FlexSC、FastSocket、Affinity-Accept 和其他的 Zero Copy Socket[34-36] 相关研究则不需要对网络应用程序的源代码进行修改。

但是，尽管这些研究工作对内核态 TCP/IP 协议栈有一定的优化，内核中的一些开销仍然存在——比如上下文切换的开销仍然存在[37]。

- **新的操作系统架构**：也有诸如 IX[38] 和 Arrakis[39] 这种新的操作系统架构被提出来。IX 和 Arrakis 它们使用虚拟化去实现隔离和安全性。它们都将控制平面放入内核态中，并将数据平面从内核态中分离。

  IX 将它的 TCP/IP 协议栈实现在了 Non-Root Ring 0 中。IX 使用 lwIP[40] 来实现它的 TCP/IP 协议栈。lwIP 是一个轻量的且简化的 TCP/IP 协议栈实现。lwIP 以面向嵌入式系统为主，因此它只有基本的 TCP/IP 协议栈功能。IX 使用 Multi-Stage Batching（多阶段批处理）方法和 Adaptive Batching（适应性批处理）方法去增加网络包处理的吞吐量并降低处理延迟。

  Arrakis 直接将 vNIC 虚拟网卡映射进网络应用程序中，这样网络应用程序可以直接从用户态访问 vNIC 虚拟网卡来获取数据。Arrakis 将自己的 TCP/IP 协议栈实现在用户态中，并使用 IOMMU 和 SR-IOV 等硬件虚拟化特性来提供安全性保护。为了降低网络包处理的延迟，Arrakis 并没有使用批处理方式。IX 和 Arrakis 都是面向数据中心的。而我们更关注的是在数据中心边缘处用于公网请求处理和转发的高性能网络应用程序。
- **使用 RDMA 替换内核态 TCP/IP 协议栈**：先前还有很多研究工作[41-44] 使用 RDMA 来分流内核态 TCP/IP 协议栈中的网络协议处理。RDMA 可以降低网络包处理延迟[45-47]。但是，基于 RDMA 的方法需要在一条网络连接的双端服务器处均有专用适配器硬件进行 RDMA 支持[48, 49]。

  SocksDirect[37] 实现了一种方式，将 RDMA 与内核的 TCP/IP 协议栈进行结合，让服务器主机可以根据对端服务器是否支持 RDMA 来自动选择使用 RDMA 还是 TPC/IP 协议栈来进行网络通信。然而，SocksDirect 的 TPC/IP 协议栈仍然依赖于内核的原生 Socket 去检查对端服务器主机是否支持 RDMA 功能。
- **用户态 TCP/IP 协议栈**：除了前面提到的新的操作系统架构 IX 和 Arrakis，还有很多研究将自己的 TCP/IP 协议栈设计实现在用户态空间中。这些用户态 TCP/IP 协议栈都是构建在高性能用户态网络包 I/O 框架之上。这些网络包 I/O 框架有 DPDK[50]、Netmap[26] 还有 PF_RING[51] 等。这种方法使得应

用程序能够直接在用户态访问网卡。SandStorm[52]、mTCP[25]、Seastar[53]、F-Stack[54] 还有 FD.io VPP[55] 在它们的用户态 TCP/IP 协议栈中使用 Zero Copy（零拷贝）[31]、Batching（批处理）[56, 57] 还有无锁化数据结构[58, 59] 以提升网络包处理的吞吐量。

lwIP[40, 60] 是一个用户态 TCP/IP 协议栈——一个轻量的且简化的 TCP/IP 协议栈实现。因为 lwIP 以面向嵌入式系统为主，因此它只有基本的 TCP/IP 协议栈功能，而不具备全网络协议栈丰富的功能。lwIP 使用的进程模型将所有的网络协议处理逻辑放入一个普通的 Linux 进程当中，使得其可以与操作系统内核进行分离。而这也使得 lwIP 能够轻松地在不同的操作系统之间进行移植[40]。

SandStorm[52]、mTCP[25] 和 Seastar[53] 仅为 L4 协议栈而不是拥有完整且丰富的功能的全网络协议栈。SandStorm、mTCP 和 Seastar 都是基于 DPDK 或者 Netmap。mTCP 中的 ARP 和 IP 的实现过于简单，还不能够产品化[61]。SandStorm、mTCP 和 Seastar 都需要开发工作者修改网络应用程序的源代码。

F-Stack[54] 是一个基于 FreeBSD 的用户态全网络协议栈。然而 F-Stack 是为单应用多进程的网络应用程序设计的。它尚不支持多线程应用程序[62]。F-Stack 也需要开发工作者修改网络应用程序的源代码并且 F-Stack 无法同时支持多种网络应用[62]。除此之外，由于 F-Stack 仅仅使用简单的批处理方式来增大网络包处理的吞吐量，因此它拥有很高的网络包处理延迟。

FD.io VPP[55] 也拥有功能完善的用户态全网络协议栈。FD.io VPP 自身是一个高性能的包处理框架。其构建于 DPDK 之上。FD.io VPP 提供了功能丰富的、高度优化的网络包转发引擎（Forwarding Engine），用于通用的商业 CPU 上。FD.io VPP 采用了很多针对通用微处理器的优化策略[63]，且支持 Linux 和 FreeBSD。FD.io VPP 的网络协议栈可同时支持多种网络应用，并且通过使用 VPP 的 LDP API[64]（也就是 LD_PRELOAD 方法），这些网络应用程序可以在不修改源代码的情况下直接使用 FD.io VPP 的网络协议栈。除此之外，FD.io VPP 的用户态全网络协议栈支持插件化的传输层协议，包括 TCP、QUIC、TLS 还有 UDP 等，让用户可以根据自己的需求选择需要的定制化协议模块[65]。FD.io VPP 支持 TLS 并且其 TLS 对于应用程序而言是透明的。这种方法减少了应用程序源代码的修改并增加了它们的可移植性。用户可以根据自己的需求选择使用具体的 VPP TLS 实现——比如有 OpenSSL 的实现，还有 Mbed TLS 的实现。

1.3 论文工作简介

在本篇论文中，我们提出了一种新的 NGINX 实现——VPPNGX。VPPNGX 使用 FD.io VPP 的用户态 TCP/IP 协议栈以充分利用 FD.io VPP 所采用的通用处理器优化技术，并充分利用 FD.io VPP 带来的网络包处理速度和低延迟性。VPPNGX 主要在数据中心边缘处负责处理和转发来自公网的大量网站访问请求。

除了将 FD.io VPP 应用到 NGINX，我们还在 VPPNGX 上提出了下面几点优化，以消除 FD.io VPP 引入的 Session Lock Layer（会话锁层）和 Busy-Wait Polling epoll（忙等轮询式 epoll 机制）带来的负面影响：

- **VPP 会话索引透传机制**：我们使用 VPP Session Index Passthrough（VPP 会话索引透传机制）将 FD.io VPP 的 Session Lock Layer（会话锁层）安全删除。Session Lock Layer 对 VPPNGX 的可扩展性有负面影响，并且还增加了网络包处理的延迟。
- **基于令牌的无锁化保序机制**：当 FD.io VPP 的 Session Lock Layer 被去除后，从 NGINX 发向 VPP 的控制事件消息在入队 VPP Event Queue（VPP 事件队列）时它们的元数据可能会发生乱序。我们为 VPP Event Queue 设计了一个 Token-Based Lock-Free Order-Preserving Approach（基于令牌的无锁化保序机制）以在去除 Session Lock Layer 后，在不重新引入巨锁的情况下保证控制事件消息在入队 VPP Event Queue 时它们的元数据不会发生乱序。
- **用户态联合阻塞式 epoll 机制**：在 NGINX 端，NGINX 使用 Busy-Wait Polling epoll（忙等轮询式 epoll 机制）去同时检查用户态的会话 epoll 事件和内核态的 epoll 事件。Busy-Wait Polling epoll 会导致 NGINX 的所有工作进程无论是否有网络请求处理均始终百分之百地占用 CPU，并且还会增加上下文切换的开销（陷入内核检查内核 epoll 事件）。我们设计了 User-Space Unified Blocking epoll（用户态联合阻塞式 epoll 机制）去替换 Busy-Wait Polling epoll，以实现阻塞功能减少 CPU 占用率，并减少进行内核 epoll 事件检查引起的上下文切换开销。

VPPNGX 在 RPS（Requests per Second，请求数/秒）方面达到了基于内核的 NGINX 的 3.33 倍。并且相比于直接使用 VPP 的 NGINX，VPPNGX 在 RPS 方面有很好的可扩展性。而且 VPPNGX 的 RPS 还比直接使用 VPP 的 NGINX 的更高。VPPNGX 的请求处理延迟比 F-Stack NGINX、直接使用 VPP 的 NGINX、基于内核的 NGINX 要分别低 46.7%、34.1%、25.3%。得益于 User-Space Unified Blocking epoll（用户态联合阻塞式 epoll 机制），VPPNGX 能够在网络请求负载较小的时候减少 CPU 的使用率，并且减少因内核 epoll 事件检查而造成的上下文切换开销。

1.4 论文结构安排

本篇论文一共有七大章节。第一章节为绪论部分，简单介绍了 NGINX 的应用背景以及内核态 TCP/IP 协议栈对 NGINX 的影响，并介绍了优化内核态 TCP/IP 协议栈的相关研究。绪论部分还介绍了本文的主要贡献。第二章节是详细的背景介绍，主要介绍 NGINX 的起源和应用、用户态 TCP/IP 协议栈的任务目标、FD.io VPP 向量包处理技术以及 VPP LDP NGINX（直接使用 VPP 的 NGINX）的工作流程总览。第三章节介绍了 VPP LDP NGINX 中影响 NGINX 请求处理性能的挑战。第四章节为设计部分，主要介绍 VPPNGX 解决第三章节中出现的问题的设计方案。第五章节从代码实现出发，主要介绍设计部分具体的技术实现。第六章节通过实验测试评估了 VPPNGX 的可扩展性、吞吐量、延迟、CPU 占用率以及进行内核 epoll 事件检查的 CPU 开销。第七章节对全文做出了总结，并介绍了 VPPNGX 目前仍存在的一些问题以及未来可以继续进行的优化。

第二章　相关背景

2.1　NGINX

NGINX[11, 15] 是一个用于 Web 服务、内容缓存、反向代理、负载均衡还有媒体流等的开源软件。NGINX 最开始是为了能够最大化网络服务性能和网络服务稳定性而设计的一款 Web 服务器软件。除了它最初提供的 HTTP 服务器功能之外，NGINX 还可以充当电子邮件的代理服务器（比如支持 SMTP、POP3 和 IMAP 等的代理服务器）以及 HTTP、TCP 和 UDP 服务器的反向代理服务器和负载均衡器。

2.1.1　NGINX 的起源

2.1.1.1　C10K 问题

NGINX 最初是由俄罗斯人 Igor Sysoev 编写的。Igor Sysoev 最初是为了解决 C10K 问题[66] 而编写的 NGINX。有关 C10K 问题的解释如下：

- C10K 是 *Concurrency 10000*（并发量 10000）的简写——当服务器端软件接收的客户端连接数量超过了 10000 个时，无论服务器端的硬件性能是否完全充足，服务器端的软件依然无法为所有的客户端连接正常地提供 Web 服务。简而言之，就是服务器单机同时服务 10000 个并发连接时遇到的问题。C10K 问题存在的原因是——在早期计算机发展的过程中，人们并没有意识到未来互联网的飞速发展。于是，在人们设计 Unix 操作系统时，仅仅采用 16 位有符号的整数来表示一个进程的 PID。这样，一台计算机上的操作系统最多只能创建 32767 个进程。然而，一台计算机至少还得留出一部分进程数以服务其他的一些应用软件，导致能留给 Web 服务的进程数量更少。

  当然，在早期这并不是什么问题，因为在互联网发展的初期，一台服务器能有数百个在线用户已经是非常不得了的事情了。在互联网发展的后期，Unix 操作系统已经可以支持 64 位的 PID，因此一台服务器能够创建的进程数量已经没有了上限。

  但是，在如今互联网技术高速发展的时代，随着网络用户数量的巨幅增长，用一个进程来负责服务一个客户端连接显然仍是不可行的，因为一台服务器接收的并发连接越多，它需要创建的进程数就越多，而大量的进程会消耗系统大量的内存（操作系统中每个进程的进程控制块 PCB 都会占据一定

的内存），最后导致内存耗尽，使操作系统性能下降甚至不可用；另外，由于服务器的 CPU 核数有限，不可能让一个 CPU 核只服务一个进程，于是一个 CPU 核上出现的大量进程间切换开销也会降低服务器提供 Web 服务的能力。这就是 C10K 问题出现的原因。

为了解决 C10K 问题，NGINX 通过使用基于事件驱动[67-69] 的异步体系架构，革新了服务器在高性能环境中服务高并发连接数时的操作方式——通过基于事件驱动的方法，一个进程可以服务多个客户端连接以大量减少内存占用并减少进程间切换开销。NGINX 也由此成为了当时最快的 Web 服务器。

在 2004 年对 NGINX 项目进行开源并观察到其被使用数量呈指数型增长之后，NGINX 的作者 Igor Sysoev 创立了 NGINX 公司以支持 NGINX 的持续开发，并将具有商业功能的 NGINX Plus 作为具有针对企业客户的附加功能的商业产品推向国际市场。现如今，开源的 NGINX 和商用的 NGINX Plus 都可以同时处理数十万个并发连接，已经为国际互联网上超过 50% 的企业级网站提供了强大的请求处理性能[11]。

2.1.1.2 NGINX 用作 Web 服务器

我们首先介绍一下什么是 Web 服务器（Web Server）[70, 71]。

在互联网环境中，文件服务器、数据库服务器、邮件服务器和 Web 服务器分别使用不同类型的服务器软件。这些服务器软件中的每一个都可以访问存储在实际物理服务器上的不同类型的文件，并将这些文件用于相应的服务。Web 服务器的工作是负责在国际互联网上服务 Web 网站。为了实现该目标，Web 服务器充当服务器计算机和客户端计算机之间的中间人。它根据每个来自客户端计算机的用户请求从服务器计算机中提取文件内容，并将其传递到互联网上返回给客户端计算机。

无论是在互联网发展的初期还是在其蓬勃发展的现在，Web 服务器面临的最大挑战一直都是同时为大量的 Web 用户提供服务——每个 Web 用户都会向服务器计算机请求不同的 Web 页面。而 Web 服务器则负责处理以不同编程语言（比如 PHP、Python、Java 或 Golang 等）编写的文件。Web 服务器将这些不同编程语言编写的文件动态地转换为静态的 HTML 文件，并将静态的 HTML 文件发回给 Web 用户并在 Web 用户的浏览器中提供这些文件。因此，Web 服务器可被视为负责服务器计算机与客户端计算机之间通信的工具。

我们在前面已经提到，最初 NGINX 背后的目标是成为世界上最快的 Web 服务器，并且始终保持卓越性能一直是 NGINX 项目的中心目标。NGINX 在衡量 Web

服务器性能的基准测试中始终击败 Apache Web Server 和其他的 Web 服务器。但是，自从最初 NGINX 开源以来，互联网技术已经飞速发展了十几年，而且 Web 网站已经从最初简单的静态 HTML 页面扩展到动态的、包含多方面内容的网页。为了贯彻其始终保持性能卓越的中心目标，NGINX 在随后的发展中，已经能够支持现代 Web 服务的所有组件——包括 WebSocket、HTTP/2 和多种视频流格式（比如 HDS、HLS 还有 RTMP 等）的流式传输。

2.1.1.3 NGINX 不止用作 Web 服务器

尽管 NGINX 已经成为了高性能的 Web 服务器软件，并且因此而名闻天下，但是事实证明，在多核服务器上可扩展的基础体系架构已成为了除了提供 Web 内容之外的众多 Web 任务的重要选择之一。因为 NGINX 可以同时处理大量的客户端连接，因此它通常也被用作反向代理服务软件和负载均衡器软件，用以管理传入服务器的流量并将其分配给处理速度较慢的上游服务器（比如旧数据库服务器、旧 Web 服务器还有微服务等）[6, 11, 15]。

另外，NGINX 还经常被放置在客户端和第二个 Web 服务器之间，用作 SSL/TLS 终端代理或者 Web 加速器。作为中间人，NGINX 能够有效地处理可能会使 Web 服务器变慢的一些任务——比如协商 SSL/TLS 或者压缩和缓存 Web 内容以提高 Web 服务的性能。很多动态 Web 站点（比如使用从 PHP 到 JSP 到 Node.js 再到 Django 等的动态内容构建的动态站点）通常将 NGINX 部署为 Web 内容缓存器和反向代理软件，减少了应用程序服务器上的负载并充分有效地利用了服务器底层的硬件资源。

2.1.2 NGINX 的设计

NGINX 在网络处理性能方面处于领先地位。这全都归功于 NGINX 的设计方式。尽管许多的 Web 服务器和应用程序服务器使用简单的基于线程或基于进程的体系架构，但是 NGINX 凭借其使用的复杂的基于事件驱动的体系架构脱颖而出。基于事件驱动的体系架构使 NGINX 能够在现代通用商业服务器硬件上同时处理成千上万的并发客户端连接。下面，我们将介绍 NGINX 的具体设计。

首先，我们提一下 NGINX 的进程模型。NGINX 运行的时候，有一个 Master 进程、多个 Worker 进程还有一些辅助进程。Master 进程负责执行特权操作，比如读取配置文件和绑定网络端口。Worker 进程则负责接收连接、处理连接以及关闭连接。辅助进程一般负责进行 Web 内容缓存的管理工作。

任何 Unix 应用程序都是基于线程或者进程的。如果从 Linux 操作系统的角

度来看，线程和进程几乎是相同的。线程和进程的主要区别是——一个进程拥有自己的一个独立的内存空间。进程与进程之间无法相互访问各自不同的内存空间。而线程则不同，线程是 Linux 操作系统 CPU 调度的最基本单位。一个普通的进程可以说是拥有一个主线程（当然实际并没有主线程这个说法，因为所有线程都是同等地位的），并且一个进程还可以拥有多个线程。一个进程中的所有线程共享同一个进程内存空间。在 Linux 操作系统上，大多数复杂的应用程序并行地运行多个线程或进程（即多线程或者多进程应用），原因主要有两个：

- 它们可以同时地使用更多的 CPU 计算核。
- 使用线程和进程可以非常轻松地进行并行化的操作（比如可以同时处理多个客户端连接）。

因此，在互联网发展的初期，设计网络应用程序时最常使用的方法就是为每个客户端连接分配一个线程或者一个进程。这样的体系架构既简单又非常容易实现。但是当应用程序需要同时处理成千上万个并发客户端连接时，这样的体系架构便无法实现良好的可扩展性。因为，进程和线程的创建是需要消耗计算机硬件资源的。每个进程或者线程都需要使用计算机内存和其他的操作系统资源，并且需要在操作系统内核态与用户态之间进行切换（也就是我们常说的上下文切换）。大多数现代服务器可以同时处理数百个小型的活动线程或进程，但是一旦进程或线程数过多，计算机内存耗尽或者高网络 I/O 负载导致大量的上下文切换，计算机的处理性能就会严重地下降。

那么，NGINX 为了解决这个问题，使用了一种新的进程模型。这个进程模型有下面一些组件：

- **Master 进程**：Master 进程可以负责执行特权级别的操作——比如读取配置文件或者绑定到指定的网络端口，然后还可以负责创建下面三种类型的 Child 进程。
- **Cache Loader 进程**：Cache Loader 进程在启动时运行，用以将基于磁盘的缓存数据加载到内存当中，然后退出。NGINX 对 Cache Loader 进程的调度比较保守，因此 Cache Loader 进程对实际的计算机资源需求比较低。
- **Cache Manager 进程**：Cache Manager 进程是周期性地运行的，负责整理和修剪磁盘高速缓存中的数据条目，以使它们能够保持在配置的大小之内。
- **Worker 进程**：Worker 进程负责处理其他所有的工作。每个 Worker 进程都单独地负责处理网络连接，负责将 Web 内容读写到磁盘，并负责与上游服务器进行通信。

在大多数情况下，每个 CPU 核只运行一个 NGINX 的 Worker 进程。这样一

个 CPU 核只负责一个进程，从而减少了单核上的进程间切换开销。这样 NGINX 便可以最有效地利用计算机的硬件资源。当 NGINX 服务器正处于活动的工作状态时，只有 NIGNX 的 Worker 进程处于繁忙的工作状态。每个 Worker 进程以基于事件驱动的非阻塞的方式处理多个客户端连接，从而减少了上下文切换的次数。每个新的客户端连接都会被分配一个文件描述符 FD，并且每个文件描述符 FD 在 Worker 进程中只会占用少量的内存。因此，每个客户端连接基本上是没有额外的开销的。另外，每个 Worker 进程都是单线程的，并且独立地运行。多个 Worker 进程可以各自独立地接收新的客户端连接并进行并行化的处理。多个 Worker 进程之间可以使用共享内存进行通信，以共享连接持久性数据、缓存内容以及其他的一些共享资源。这样，NGINX 便能够在多核服务器系统上很好地进行扩展，以支持每个 Worker 进程数十万左右的客户端连接数。

2.1.3 NGINX 与 Apache 的比较

2.1.3.1 Apache 简介

Apache[14, 72] 是一个开源的免费的 Web 服务器软件。它为全球大量的企业级网站提供 Web 服务支持。Apache Web 服务器软件的正式名称为 *Apache HTTP Server*。它是由 Apache Software Foundation 开发和维护的。

Apache HTTP Server 允许网站管理员在国际互联网上提供 Web 内容，因此它被命名为 “Web 服务器”。Apache HTTP Server 是最古老、最可靠的 Web 服务器之一，其第一版于 1995 年发布。Apache 最初基于 NCSA HTTPd 服务器。在 NCSA 代码工作停滞之后，Apache 于 1995 年初开始进行开发。Apache 在互联网的最初发展中发挥了关键的作用，并迅速取代了 NCSA HTTPd 成为主要的 HTTP 服务器。Apache 自从 1996 年 4 月以来一直广受各大网站的欢迎。在 2009 年，Apache HTTP Server 成为第一个给超过一亿个网站提供 Web 服务的 Web 服务器软件[14]。

2.1.3.2 Apache 的设计

在互联网发展的早期，Apache 仅仅实现了一个简单的体系架构——即基于进程的体系架构（一个进程服务一个客户端连接）。而如今，随着 Apache 的发展，Apache 提供了多种多样的 MultiProcessing Module（MPM）[73]。MPM 允许 Apache 以基于进程的架构、混合（基于进程和线程）架构或者基于事件驱动的混合架构来运行，以便更好地满足每个特定环境的需求。因此，MPM 的配置和选择非常重要。在必须折衷处理性能的地方，相比于简单地处理更多请求，Apache 旨在减少处理延迟并增加吞吐量，从而保证在合理的时间范围内对请求进行可靠并且一致

的处理。

2.1.3.3 NGINX 与 Apache 的比较

在性能方面，NGINX 得益于其使用的基于事件驱动的非阻塞式的设计架构，相比于 Apache 占用更少的内存资源并且拥有更高的请求处理性能。在 NGINX 出现的早期，Apache 仍然使用的是一个简单的体系架构——即基于进程的体系架构（一个进程服务一个客户端连接）。因此在早期 NGINX 与 Apache 的比较中，NGINX 的性能远远超过 Apache（在吞吐量方面 NGINX 大约可以达到 Apache 的 4 倍左右）。但后来随着 Apache 的发展，Apache 提供了多种多样的 MultiProcessing Module（MPM），以允许 Apache 以基于进程的架构、混合（基于进程和线程）架构或者基于事件驱动的混合架构来运行。这样，Apache 在性能方面开始追赶 NGINX。但是，由于 Apache 相对复杂和重量级的设计，Apache 在性能方面并不能完全超越 NGINX。

但是，Apache 也有它相比于 NGINX 的诸多优点。比如相比于 NGINX，Apache 的模块化程度更高，并且模块的热更新和配置要更加便捷和灵活。另外，Apache 更加稳定，在处理请求的过程中很少出现无法服务的情况。Apache 更适合于服务动态的 Web 网页，而 NGINX 则适合于服务静态的 Web 网页。因此，在大多数情况下，Apache 服务器软件经常被用作上游的后端 Web 服务器，用来处理实际的客户端请求；而 NGINX 则作为下游的前端反向代理服务器，将来自客户端的请求转发给后端的 Apache 服务器进行处理，再将 Apache 服务器的回复转发回客户端。

2.2 用户态 TCP/IP 协议栈

2.2.1 内核态 TCP/IP 协议栈的不足

随着现代互联网的飞速发展，各种各样的互联网资源（比如文字、图片、视频、音频等资源）日益增多。而人们对这些网络服务器资源的访问量也在巨幅地增长。这样，提供这些网络服务的服务器就必须能够应对如今大量的用户访问。于是，人们开始竭尽所能地去不断压榨现有服务器的所有硬件资源——比如 CPU 资源、内存资源、网络带宽资源等。而服务器厂商、硬件厂商也在不断地扩展硬件性能——比如提升单个 CPU 核的处理性能或者增加一个物理 CPU 所能容纳的 CPU 核数等。然而可惜的是，这样的努力并没能够使现代网络服务应用的处理性能线性地提升。产生这个结果的其中一个主要原因就是，操作系统内核提供的 TCP/IP 协议栈成为了限制现代网络服务应用性能提升的一个巨大瓶颈。

而操作系统内核提供的 TCP/IP 协议栈成为性能提升瓶颈的诸多原因，有系统调用开销、内核态和用户态之间数据包的拷贝开销、为每个网络数据包分配或者回收重量级数据结构的开销、硬件中断开销、共享数据结构竞争开销以及大量的缓存未命中开销等[25, 38, 39, 53-55, 74]。我们接下来将一一介绍操作系统内核态 TCP/IP 协议栈中的这些开销。

系统调用开销：当一个网络应用程序需要实际使用服务器物理硬件资源的时候，它需要通过系统调用，从应用程序所处的用户态陷入操作系统内核所处的内核态，以执行特权指令来实际访问物理硬件资源。在物理硬件资源访问结束后，CPU 还要从内核态退出到用户态以继续执行网络应用程序的代码。然而，在 CPU 从用户态切换到内核态再从内核态切换到用户态的这一过程，CPU 需要消耗大量的时间去做寄存器数据的保存和恢复工作。而这就会造成大量的 CPU 时间开销[75]。比如，当网络应用程序执行 I/O 操作时，比如 *read()*、*write()*、*open()* 还有 *close()* 等，它都需要通过系统调用来和操作系统的内核进行交互，以实现服务器数据、网络数据的实际读写。当网络应用程序需要处理大量的网络请求时，它就需要频繁地调用 *read()* 和 *write()* 等，造成大量的系统调用开销。

内核态和用户态之间数据包的拷贝开销：在传统的操作系统当中，当网络应用程序要将网络数据发送到网络时，它实际上要将原本处在自己的用户态内存空间的网络数据通过系统调用拷贝进内核态的内存空间当中，然后操作系统内核才会将处在内核内存空间中的网络数据通过网卡发送到网络当中；同理，如果操作系统内核接收到了来自网络的网络数据包，它便会首先将网络数据放进内核内存空间中的缓存数据结构当中，然后再将内核内存空间中的网络数据拷贝进网络应用程序的用户态内存空间当中。当 CPU 从内核态切换回用户态后，网络应用程序便可以从自己的用户态内存空间中直接读取到来的网络数据。而这种工作模式所造成的后果便是——当网络应用程序需要大量地收发网络数据时，这个过程就会伴随大量的内核态和用户态之间的数据拷贝，从而造成大量的时间开销。

为每个网络数据包分配或者回收重量级数据结构的开销：在 Linux 操作系统中，内核为了对每一个网络数据包进行方便管理，使用数据结构 “sk_buff” 来对每一个网络数据包进行抽象表示。这样，数据结构 sk_buff 便成为了 Linux 系统中网络子系统部分重要的支柱数据结构，而各个网络协议层都依赖于数据结构 sk_buff 来进行正常的网络包处理工作。当网络应用程序大量地处理网络请求时，操作系统内核态 TCP/IP 协议栈便需要处理大量的网络数据包。而这个过程便会涉及到数据结构 sk_buff 的大量内存分配和内存回收。因此，这个过程便会造成了为每个网络数据包分配或者回收重量级数据结构的时间开销。

硬件中断开销：在早期的 Linux 操作系统中，当有网络数据包到达服务器主机时，网卡是通过硬件中断的方式来告知 CPU 网络数据包的到来。网卡收包的流程大致为——当网络数据包到达服务器主机的网卡时，网卡会通过 DMA（Direct Memory Access）方式直接将网络数据包写入到内存当中。接下来网卡会通过硬件中断 IRQ 来通知 CPU 有新的网络数据包到来。CPU 收到通知后，便会通过中断向量表来调用相应的中断函数，最后去调用驱动程序来处理到来的网络数据包。当网络应用程序处理大量的网络请求的时候，如果网卡一直是通过这种中断的方式来接收网络数据包，那么便会出现大量的硬件中断开销。这样最终导致的后果就是，硬件中断会源源不断地打断操作系统处理网络数据包时的软中断，然后之前先到来的网络数据包就没法被及时地进行处理，总是会被硬件中断所打断。最后，网络数据包大量堆积而且不能被及时地处理，最终导致操作系统内核处理网络数据包的性能下降。

共享数据结构竞争开销：在现代的多 CPU 核的服务器上，为了能够使网络应用程序的处理性能随着 CPU 核数的增长也能够线性增长（即实现良好的可扩展性），那么多核之间不能出现资源竞争。在多核服务器系统上一旦有资源竞争，多核并行处理的性能就会下降。在传统的操作系统内核中，有很多共享资源存在[24, 27]。比如，多线程应用程序的所有线程都会共享同一个监听套接字（Listening Socket）。当一个新的网络连接到来时，所有线程会通过监听套接字去竞争接收这个新的网络连接（也就是通过竞争从共享的接收队列中取出并接收这个新的网络连接）。再比如，在一个进程中，文件描述符表是被共享的。当进程中的每个线程创建一个新的文件描述符时，它要遵循最小可用编号分配原则，通过加锁方式竞争访问文件描述符表，并从中找出最小可用的文件描述符。这种方式也会影响多线程应用程序在多核系统上的可扩展性。还有，网络套接字与普通文件通过虚拟文件系统（VFS）共享文件描述符，也会影响涉及文件描述符打开与关闭的短连接处理场景的网络应用程序的性能。

大量的缓存未命中开销：在传统的操作系统当中，会有几种造成大量缓存未命中开销的情况。一种是在多核操作系统上频繁地进行跨核访问共享资源引起的缓存未命中开销，还有一种是因未采用批处理方式成批地处理网络数据包而遗留的缓存未命中开销。缓存未命中涉及了指令缓存 i-cache 未命中和数据缓存 d-cache 未命中。无论哪种缓存未命中都会降低操作系统内核处理网络数据包的效率。缓存未命中也会造成严重的数据读写时延。这不仅仅会影响网络包处理的吞吐量，还会影响网络包处理的延迟，甚至增大包处理延迟的抖动性——这对网络包处理的稳定性是非常不利的。

2.2.2 用户态 TCP/IP 协议栈

网络协议栈（包含 TCP/IP 协议栈）是计算机网络协议组件的一个具体实现。网络协议栈中的每个协议层都负责进行已经协议好的具体的网络数据处理工作，以分层次地对网络数据进行处理。而网络协议栈无论是实现在操作系统的内核态中还是实现在其用户态中，网络协议栈执行的特定网络功能是不会改变的。但是，将网络协议栈实现在内核态和用户态的差异体现在其性能、部署的灵活性、迭代开发的简易程度等方面上。

正如前面小节 2.2.1所提到的那样，传统的内核态网络协议栈在如今的现代商用服务器上会引起系统调用开销、内核态和用户态之间数据包的拷贝开销、为每个网络数据包分配或者回收重量级数据结构的开销、硬件中断开销、共享数据结构竞争开销以及大量的缓存未命中开销等。人们为了降低甚至消除这些开销，研究了很多方案（比如直接优化内核网络协议栈、使用 RDMA 或者选择直接绕过操作系统内核）。这么多方案中被广泛使用的其中之一便是 Kernel Bypass（内核旁路或者绕过内核）方法。而选择将网络协议栈实现在用户态就是内核旁路方法的一种体现。

其实，操作系统中的网络协议栈只负责对已经进入内存的网络数据进行处理——将网络数据帧经过层层网络协议转换为应用层使用的应用数据，或者将应用层使用的应用数据经过层层网络协议转换为网络数据帧。所以，用户态的网络协议栈也是对内存中的这些网络数据进行处理。只不过，它是在操作系统用户态中对这些网络数据进行处理，而不是像内核网络协议栈那样在操作系统内核态中处理数据。这样便可以减少内核态和用户态之间进行切换的开销。采用用户态网络协议栈大致有下面这些优点：

- 减少了内核态和用户态之间进行切换的开销。
- 方便使用 Zero Copy（零拷贝）[31-33] 方式来减少用户态和内核态之间的数据拷贝。
- 方便使用 Batching（批处理）[56, 57, 76] 方式来减少处理网络数据包时 i-cache 和 d-cache 的高缓存未命中率。
- 方便对用户态网络协议栈的功能进行频繁的迭代式更新。如果是内核态网络协议栈，增加或删除一个功能需要对内核代码进行修改和审查，导致功能迭代周期时间非常长，不利于灵活而且快速的开发。
- 方便设计和使用无锁化的数据结构，来增强用户态网络协议栈在多核服务器系统上的可扩展性。
- 方便根据用户需求，对用户态网络协议栈进行个性化的定制和开发，并能

够根据不同时段的需求，灵活地对网络协议栈进行再配置和再部署。

目前在学术界和工业界比较知名的一些用户态 TCP/IP 协议栈有 lwIP[40, 60]、mTCP[25]、SandStorm[52]、Seastar[53]、F-Stack[54] 还有 FD.io VPP[55] 等。

由于用户态网络协议栈只负责处理已经进入内存的网络数据，因此为了让整条网络包处理路径上不出现性能瓶颈，人们还必须对网络包从网卡到内存这一个 I/O 通路进行优化。于是很多用户态的高性能网络包 I/O 框架被提出来——比如有 DPDK[50]、Netmap[26] 还有 PF_RING[51] 等。现有的用户态网络协议栈大多数都是基于这些包 I/O 框架进行构建的。用户态的高性能网络包 I/O 框架的优势在于——它们使用轮询（Polling）[77] 的方式来代替传统的使用网卡硬件中断进行收包的方式；它们将网卡接收到的网络数据包直接放入用户态的内存中以绕过操作系统内核；它们还使用一些常用的包处理优化方法（比如批处理）来增大吞吐量。

2.3 FD.io VPP 向量包处理

FD.io VPP（Vector Packet Processing）[55] 是一个用户态的高性能网络包处理加速框架。VPP 是基于用户态高性能包 I/O 框架 DPDK 进行构建的，并能够在通用商业微处理器上运行。使用 VPP 的好处在于它的高性能包处理、模块化设计理念和灵活易用并且丰富的功能。

我们首先介绍一下 VPP 的向量包处理概念（Vector Packet Processing）[78]。顾名思义，VPP 使用向量（Vector）处理而非标量（Scalar）处理。标量数据包处理指的是一次处理一个数据包（网卡在接收到一个网络数据包时，通过硬件中断来告知 CPU 已经获取了单个包，然后通过一组函数调用对这个网络包进行处理）。在传统的操作系统中，使用传统的标量数据包处理会引起下面这些问题：

- 指令缓存 i-cache 经常会发生抖动（Thrashing）。
- 每个网络数据包都会引起一系列相同的指令缓存未命中。
- 除了提供更大的缓存之外，没有其他的解决方法能够直接解决上面这些问题。

相比之下，向量包处理方法便可以一次处理一个以上的批量网络数据包。向量包处理的方法的好处之一就是它解决了指令缓存 i-cache 的抖动问题。它同时还减轻了指令缓存读取延迟的相关问题（通过预取缓存指令降低甚至消除了读取延迟）。除了能够优化指令缓存 i-cache，VPP 的向量包处理还能够优化数据缓存 d-cache。

使用向量包处理方法后，随着向量大小（一组数据包中包的个数）的增加，平均分摊到每个数据包的处理成本也会逐渐降低。

我们接下来简单介绍一下 VPP 的模块化设计理念。VPP 使用图的方式来组织各个功能模块。每个图节点便可以执行一个特定的功能。图节点与图节点之间通过边连接，那么数个图节点之间便形成了特定的功能链。VPP 这个基于图的模块化设计，使得任何用户都可以向图中插入一个新的图节点，也就是一个新的功能。这使得 VPP 具有良好的功能可扩展性。用户可以自定义自己的功能模块，然后将定制的模块作为一个新的图节点插入到 VPP 的图中。那么，前面提到的向量包处理，便是将每一组网络数据包沿着 VPP 的这些图节点进行处理——VPP 从 RX 接收队列获取所有可用的网络数据包，以形成数据包向量（Vector）。数据包向量会沿着整个图的节点（包括用户自定义的插件节点）被处理。所有的图节点是解耦的，以方便用户插入、删除或者重组图中的节点。

在云环境中，VPP 经常被用作虚拟交换机或者虚拟路由器，以提供高性能的包处理功能。VPP 可以用在容器中、虚拟机中或裸机上作为主机栈。另外，VPP 基于自身的包处理加速框架，实现了自己的用户态高性能 L2-L7 全网络协议栈。我们将在下面的小节 2.4中介绍把 VPP 应用于 NGINX 的具体理由。

2.4 为什么将 VPP 应用于 NGINX

无疑地，像 NGINX 一类的 Web 服务器软件需要高性能的网络协议栈作为其底层网络通信的基础。用户态的网络协议栈已经被广泛的研究证明，其性能要比内核态的网络协议栈更高[25, 38, 54, 79]。用户态网络协议栈避免了应用程序和内核进行 I/O 处理时产生的繁重的上下文切换开销。如表 2–1所示，以前的研究都是使用 Zero Copy（零拷贝）和 Batching（批处理）去增加吞吐量，并使用无锁化的数据结构以减少锁竞争。它们也都使用基于事件驱动的方法比如 epoll[80]、kqueue 等以提升请求处理性能。

FD.io VPP 是一个高性能的包处理框架。其构建于 DPDK 之上。FD.io VPP 提供了功能丰富的、高度优化的网络包转发引擎（Forwarding Engine），用于通用的商业 CPU 上[81]。VPP 可以同时支持多个网络应用程序。通过使用 VPP 的 LDP API[64]（也就是 LD_PRELOAD 方法），这些网络应用程序可以在不修改源代码的情况下直接使用 VPP。VPP 基于自身的框架实现了一个 L2-L7 的用户态网络协议栈，以充分利用自身包处理框架的性能和速度。然而，不像其他一些直接将用户态的网络协议栈实现为一个 Library OS 的进程模型[25, 38, 54, 82]，VPP 的进程模型则是将用户态网络协议栈放置于一个独立的普通的 Linux 进程中。这种方式既能够将网络协议栈功能的开发与内核开发和应用程序开发进行解耦合，还能够将网络服务与 CPU 配置和线程调度进行解耦合[83]。VPP 的网络协议栈也使用 Zero

表 2–1 FD.io VPP 和其他用户态 TCP/IP 协议栈的比较

Table 2–1 Comparison of FD.io VPP and Other Work on User-Space TCP/IP Stack

	IX	Arrakis	SandStorm	mTCP	Seastar	F-Stack	FD.io VPP
TCP/IP Stack in an Ordinary Process							✓
Zero Copy	✓	✓	✓	✓	✓	✓	✓
Batching	✓		✓	✓	✓	✓	✓
Adaptive Batching for Latency Reducing	✓						✓
Multi-Stage Batching on Each Protocol Stage	✓						✓
Lock-Free Data Structures in TCP/IP Stack	✓	✓	✓	✓	✓	✓	✓
Fully-Featured L2-L7 Networking Stack						✓	✓
Transparent TLS Support							✓
Pluggable Transport Protocols							✓
epoll	✓	✓	✓	✓	✓	✓(kqueue)	✓
Transparent Socket API	✓	✓					✓
Compatible with Regular TCP Peers	✓	✓	✓	✓	✓	✓	✓
QoS (Performance Isolation)	Kernel	NIC	NIC	NIC	NIC	NIC	NIC

Copy 和 Batching，将一组网络包在一系列插件化的图功能节点上同时进行批量处理。这种方法可以优化 i-cache 和 d-cache 的命中率[81]。VPP 也像 IX[38] 一样使用 Multi-Stage Batching（多阶段批处理）和 Adaptive Batching（适应性批处理）[84, 85] 来增加网络包处理的吞吐量并减少处理延迟。VPP 的网络协议栈能在多核系统上很好地进行扩展[86]。另外，VPP 的 TCP/IP 协议栈支持插件化的传输层协议，包括 TCP、QUIC、TLS 还有 UDP 等，让用户可以根据自己的需求选择需要的定制化协议模块[65]。VPP 支持 TLS 并且其 TLS 对于应用程序而言是透明的。这种方法减少了应用程序源代码的修改并增加了它们的可移植性。因此，我们使用 FD.io VPP 来对 NGINX 进行优化，以充分利用 VPP 的所有优势特性。

2.5 VPP LDP NGINX 总览

VPP LDP NGINX 指的是通过 LD_PRELOAD 方式直接使用 FD.io VPP 的 NGINX。图 2–1是 VPP LDP NGINX 的总览图。我们将首先介绍 VPP LDP NGINX 的工作流程，然后再详细地介绍各个模块的具体功能。

2.5.1 VPP LDP NGINX 工作流程简介

原生 NGINX 通过使用事件驱动模型，根据接收到的 epoll 事件的类型，来对网络连接进行不同的处理。在 NGINX 启动时，NGINX Master 会创建监听套接字并将其绑定 IP 地址和端口号。然后 NGINX Master 会 fork 出子进程（NGINX Worker）。这样每个 NGINX Worker 就会共享同一个监听套接字，用于接收新连接。每个 NGINX Worker 也会创建独立的 epoll 套接字以初始化事件驱动逻辑（Event-Driven Logic）。事件驱动逻辑负责接收所有的 epoll 事件，并根据事件的类型执行相应的操作——比如接收新连接、读写连接或者读取来自 NGINX Master 的 IPC Message（进程间通信消息）等。

同样，使用 VPP 的 VPP LDP NGINX 仍然需要以事件驱动模型为基础来处理网络连接。我们在本小节首先介绍 VPP LDP NGINX 的工作流程。

VPP LDP NGINX 使用的 Session（会话）相当于 Linux 中的 Socket（套接字）。在图 2–1所示的总览图中，多个 NGINX Worker 使用一个 **VPP Worker (④)** 提供的传输层实现。VPP Worker 在一个独立的 Linux 进程中提供网络协议栈的功能。因此，NGINX Worker 进程要与 VPP Worker 进程协同工作，需要通过共享内存中的队列数据结构进行信息交换。

在 VPP LDP NGINX 启动时，NGINX Master 进程首先通过共享内存对接 VPP Worker 进程。然后 NGINX Master 创建 Listener Session 并将其绑定好 IP 地址和端

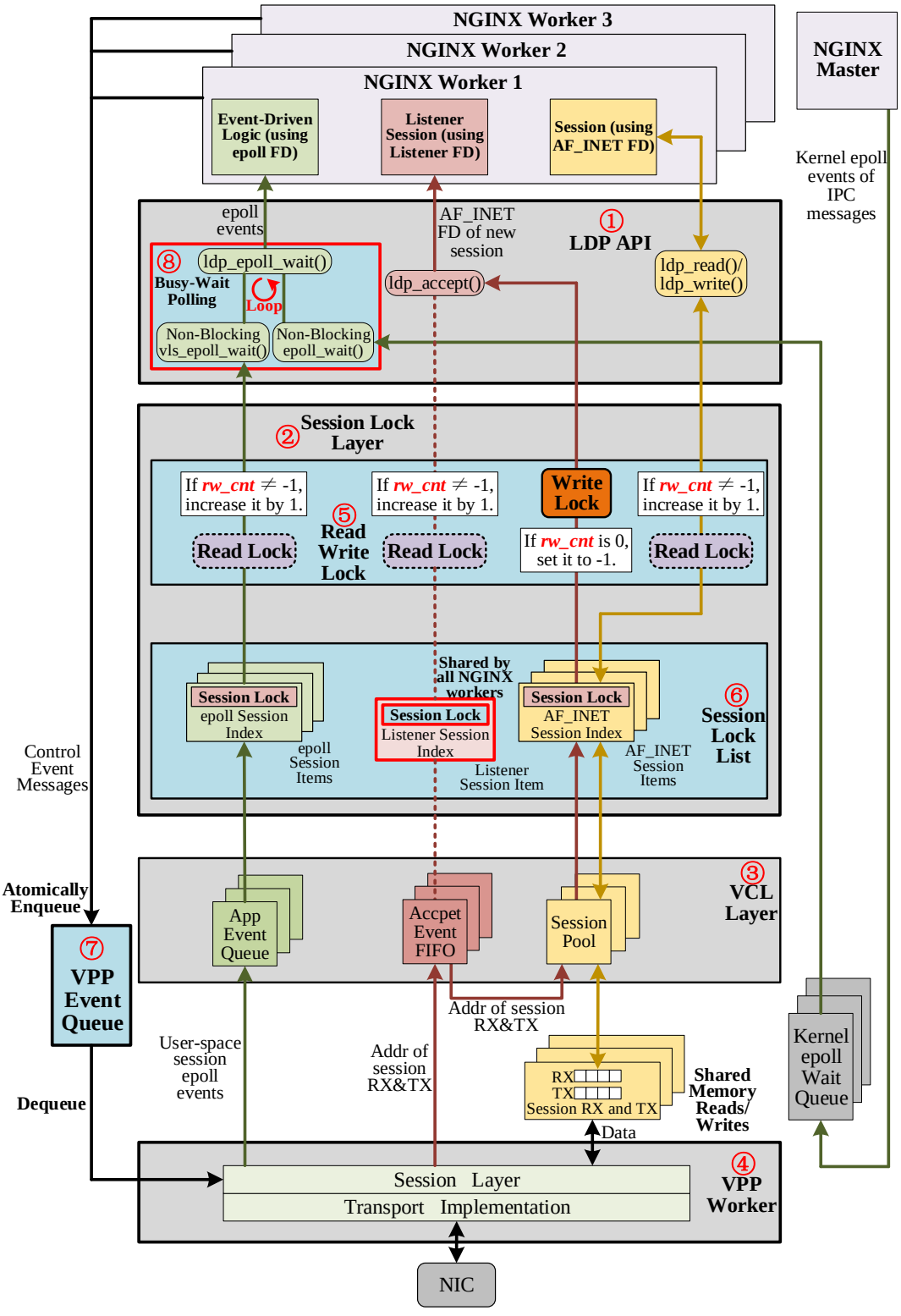

图 2–1 VPP LDP NGINX 总览

Figure 2–1 Overview of VPP LDP NGINX

口号。之后 NGINX Master 会 fork 出子进程（NGINX Worker 进程）。每个 NGINX Worker 进程便共享这个 Listener Session 以接收新连接（即新 Session）。每个 NGINX Worker 还会创建自己独立的 epoll Session 以初始化自己的事件驱动逻辑（Event-Driven Logic）。VPP LDP NGINX 的 NGINX Worker 中的事件驱动逻辑负责接收所有的 epoll 事件，并根据 epoll 事件的类型执行不同的操作。

事件驱动逻辑（Event-Driven Logic）接收 epoll 事件：NGINX Worker 中的事件驱动逻辑使用 **Busy-Wait Polling epoll（⊛ 忙等轮询式 epoll 机制）**，通过轮询的方式主动且不间断地检查用户态 epoll 事件和内核态 epoll 事件。用户态 epoll 事件是与 Session 相关的，包含 Session 的 RX 通知、Accept 通知等；内核态的 epoll 事件与 IPC Message（进程间通信消息）相关，是 NGINX Master 向 NGINX Worker 发送的 IPC Message 的通知。然后，事件驱动逻辑会根据 epoll 事件通知的类型，来决定接下来是接收新的 Session、读写 Session 还是读取 IPC Message 等。用户态的 Session 的 epoll 事件都是从共享内存中的队列数据结构 App Event Queue（应用事件队列）中获取的，而内核态的 epoll 事件则是从 Kernel epoll Wait Queue（内核 epoll 事件等待队列）中获取。事件驱动逻辑需要使用 epoll FD（epoll 文件描述符）调用函数 *ldp_epoll_wait()* 才能获取 epoll 事件。

Listener Session 接收新连接：当 NGINX Worker 中的事件驱动逻辑接收到了新连接到来的 epoll 事件（Accept 通知），它便会让 Listener Session 去接收这个新的连接（即新的 Session）。于是，Listener Session 使用 Listener Session FD 调用函数 *ldp_accept()* 最终获得了新 Session 的 AF_INET Session FD。以后，只要有关该 Session 的读写 epoll 事件到来，事件驱动逻辑便会让 Session 去读写数据。新 Session 的接收过程涉及了对共享内存中的队列数据结构 Accept Event FIFO（接收事件先入先出队列）的操作。

Session 读写数据：当 NGINX Worker 中的事件驱动逻辑接收到了一个 Session 的读写相关的 epoll 事件，它便会让这个 Session 去读写数据。于是，这个 Session 便会使用 AF_INET Session FD 去调用函数 *ldp_read()/ldp_write()* 进行数据读写。通过 AF_INET Session FD，Session 可以最终找到在共享内存中的 Session RX 和 TX。通过 Session RX 和 TX，Session 才能够与 VPP Worker 中的高性能用户态传输层实现进行数据交换。

以上便是 VPP LDP NGINX 基于事件驱动模型的工作流程的简介。在接下来的几个小节中，我们将从细节上对这个工作流程涉及的模块进行具体介绍。

2.5.2 LDP API

LDP API（①）为 NGINX 提供了 POSIX API 让 NGINX 可以通过该 API 处理 Session。LDP API 是通过 LD_PRELOAD 方法将 NGINX 原本使用的 Socket 函数重新定向到 VPP 自己实现的库中。所有的 LDP 函数都以“*ldp_*”作为命名前缀。

2.5.3 VCL Layer

VCL 是 *VPP Communication Library* 的缩写。在 **VCL Layer**（③）中，有一些队列数据结构通过共享内存负责 NGINX Worker 与 VPP Worker 之间的信息交换——每个 NGINX Worker 都拥有一个 App Event Queue（应用事件队列）来接收用户态的 Session 的 epoll 事件；每个 NGINX Worker 也拥有一个 Accept Event FIFO（接收事件先入先出队列）来接收新的 Session 的 RX/TX 地址；每个 NGINX Worker 还拥有一个 Session Pool（会话池）来存放所有的已建立连接的 Session 的 RX/TX 地址。所有的 NGINX Worker 通过 Session 的 RX/TX 来与 VPP Worker 中的高性能用户态传输层实现进行数据交换。这些队列数据结构均存在于共享内存之中。

2.5.4 Session Lock Layer

Session Lock Layer（② 会话锁层）负责管理所有的 Session Lock（会话锁）和 Session Index（会话索引）。Session Lock 相当于 Linux 中为多线程应用设计的 Socket Lock（套接字锁）。而 Session Index 则相当于是管理一个 Session 所使用的 Session Handler（会话句柄）。通过 Session Index，各类 Session 才能够定位访问 VCL Layer 中的队列数据结构，以与 VPP Worker 进行信息交换。

Session Lock Layer 是由一个 **Session Lock List（⑥ 会话锁表）**和一个 **Read-/Write Lock（⑤ 读写锁）**组成。

- **Session Lock List（会话锁表）**：每一个 Session（会话）拥有一个 Session Lock（会话锁）和一个 Session Index（会话索引）。一个 Session 的 Session Lock 和 Session Index 均存放于 Session Lock List 中的一个 Session Item（Session 表项）中。每个 Session Item 是由文件描述符 FD 进行索引定位的。
  在所有的 Session Item 中，只有 Listener Session Item 是由所有的 NGINX Worker 共享的。虽然所有的 NGINX Worker 共享着同一个 Listener Session Item，但是每个 NGINX Worker 最终都会通过共享的 Listener Session Index 来访问它们自己独有的 Accept Event FIFO。
- **Read/Write Lock（读写锁）**：Read/Write Lock 的引入是为了保护对 Session Lock List 的读写。一个 NGINX Worker 读取 Session Lock List 中的表项时需

要加读锁，增加或删除表项时需要加写锁。VPP 通过一个变量 ***rw_cnt*** 来实现这个 Read/Write Lock。如果 ***rw_cnt*** 值为-1，则表明有一个写者正持有写锁；如果 ***rw_cnt*** 的值为非负整数，则该值代表的是正持有读锁的所有读者的数量。

当一个 Session 要去访问 VCL Layer 中的队列数据结构时，它必须首先要持有读锁，然后再访问 Session Lock List，并通过文件描述符 FD 定位找到属于自己的 Session Item 表项（FD 相当于表项的索引）。然后 Session 从自己的 Session Item 表项中找到 Session Lock 和 Session Index。最后，它会先持有 Session Lock，然后再通过 Session Index 去访问 VCL Layer 中的队列数据结构。

2.5.5 连接建立过程的具体细节

VPP LDP NGINX 接收一条新的连接的具体过程如图 2–1所示：

- 当 VPP Worker 的传输层实现接收了一条连接时，VPP Worker 会首先为这条连接创建一个 Session RX/TX。接下来 VPP Worker 向 App Event Queue 中添加一个 epoll 事件表明新连接到来，并向 Accept Event FIFO 中添加新 Session RX/TX 的地址。
- 一个 NGINX Worker 的事件驱动逻辑会调用函数 *ldp_epoll_wait()*，然后函数 *ldp_epoll_wait()* 调用非阻塞的函数 *vls_epoll_wait()* 去首先获取 Session Lock Layer 中的读锁。为了获取读锁，该 NGINX Worker 原子性地将变量 *rw_cnt* 增加 1。作为持有读锁的读者，该 NGINX Worker 通过 epoll FD 从 Session Lock List 中找出 epoll Session Item，然后获取并持有其中的 epoll Session Lock。获取了 epoll Session Lock 后，NGINX Worker 通过 epoll Session Item 中的 epoll Session Index 访问 VCL Layer 中的 App Event Queue，并从 App Event Queue 中取出 epoll 事件，得知有新的连接到来。
- 于是接下来 NGINX Worker 中的 Listener Session 调用函数 *ldp_accept()*，先原子性地将 *rw_cnt* 增加 1 来获取读锁，然后通过 Listener FD 从 Session Lock List 中找出 Listener Session Item，并持有其中的 Listener Session Lock，再使用 Listener Session Index 访问 Accept Event FIFO，将其中的新 Session RX/TX 地址取出并放入 Session Pool 中的一个新 Session Slot（会话槽）中。
- 指向新 Session Slot 的索引将作为 AF_INET Session Index 和新 Session Lock 一起被放入一个新的 AF_INET Session Item 中。然后 NGINX Worker 将变量 *rw_cnt* 设置为-1 来获取写锁，并将新的 AF_INET Session Item 放入 Session Lock List。最后，指向新的 AF_INET Session Item 的索引将被作为 AF_INET

FD 传递回 Listener Session。

- 在此之后，该 NGINX Worker 的事件驱动逻辑一旦收到了 Session 读写相关的 epoll 事件，Session 便可以调用函数 *ldp_read()/ldp_write()*，通过文件描述符 AF_INET FD 从 Session Lock List 中找出 AF_INET Session Item，并使用其中的 AF_INET Session Index 找到 Session RX/TX 地址，然后通过共享内存中的 Session RX/TX 与 VPP Worker 中的高性能用户态传输层实现进行数据交换。

2.5.6 VPP Event Queue

每一个 VPP Worker 拥有一个 **VPP Event Queue（㋐VPP 事件队列）**去接收来自所有 NGINX Worker 的 Control Event Message（控制事件消息）。在新连接被接收或者旧连接被关闭时，每个 NGINX Worker 会通过 VPP Event Queue 向 VPP Worker 发送 Accept Control Event Message（连接接收控制事件消息）或者 Disconnect Control Event Message（连接关闭控制事件消息），以告诉 VPP Worker 连接已经被接收或者关闭。然后 VPP Worker 在 Session Layer 中便会根据控制事件消息来修改 Session 的状态。一个 VPP Event Queue 包含了一个 Message Metadata Queue（消息元数据队列）和一个 Control Event Data Ring（控制事件数据环形队列）。

- **Control Event Data Ring**：Control Event Data Ring（控制事件数据环形队列）用于存储控制事件消息的数据。Control Event Data Ring 中的每一个 Data Slot（数据槽）会和一个控制事件消息绑定。
- **Message Metadata Queue**：Message Metadata Queue（消息元数据队列）用于存储消息的元数据。一个控制事件消息的元数据包含了指向 Control Event Data Ring 中与该消息绑定的 Data Slot 的索引（Index）。
- **所有控制事件消息都是原子性地进入 VPP Event Queue**：当一个 NGINX Worker 持有读锁从 Session Lock List 中访问 Listener Session Item 并持有其中的 Listener Session Lock 时，它会通过 VPP Event Queue 向 VPP Worker 发送一个 Accept Control Event Message（连接接收控制事件消息）；当一个 NGINX Worker 持有写锁从 Session Lock List 中删除一条 AF_INET Session Item 时，它会通过 VPP Event Queue 向 VPP Worker 发送一个 Disconnect Control Event Message（连接关闭控制事件消息）。

  如图 2–2所示，在 Read/Write Lock 和 Listener Session Lock 的共同保护下，所有的 Accept Control Event Message 和 Disconnect Control Event Message 会原子性地进入 VPP Event Queue。

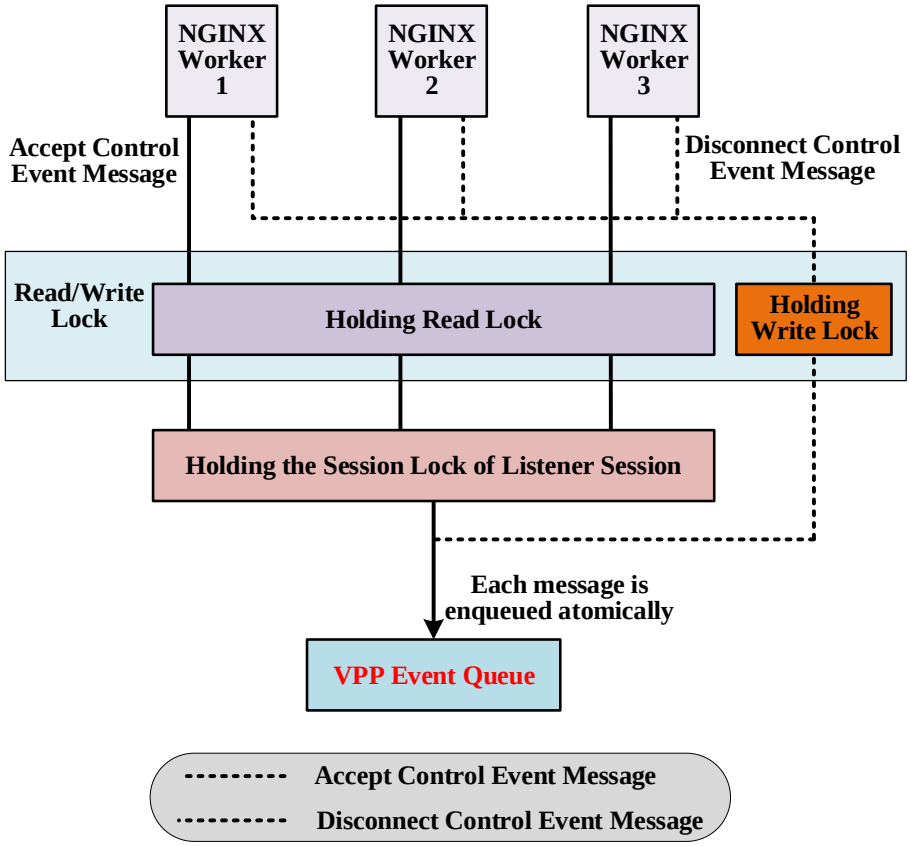

图 2–2 控制事件消息原子性地进入 VPP Event Queue

Figure 2–2 Each Control Event Message Is Atomically Enqueued into VPP Event Queue

2.5.7 Busy-Wait Polling epoll

NGINX Master 会向 NGINX Worker 发送 IPC Message（进程间通信消息）来控制 NGINX Worker。在 NGINX Worker 端，每个 NGINX Worker 的事件驱动逻辑（Event-Driven Logic）使用 Busy-Wait Polling epoll（忙等轮询式 epoll 机制），通过轮询的方式主动且不间断地检查 App Event Queue（应用事件队列）中来自 VPP Worker 的 Session 的用户态 epoll 事件和 Kernel epoll Wait Queue（内核 epoll 事件等待队列）中来自 NGINX Master 的 IPC Message 的内核态 epoll 事件。通过这种轮询的方式，事件驱动逻辑才既能及时收到用户态的 epoll 事件也能及时收到内核态的 epoll 事件。

在接下来的章节 三中，我们将介绍把 VPP 的这些策略应用到 NGINX 时出现的一些问题，并指出针对这些问题可能进行的优化。

2.6 本章小结

本章节主要介绍了 NGINX 的起源和应用背景，并介绍了内核态 TCP/IP 协议栈对 NGINX 的负面影响以及用户态 TCP/IP 协议栈的任务目标。本章节还介绍了使用向量包处理技术的 FD.io VPP 平台以及其提供的高性能用户态网络协议栈，并阐释了将 VPP 应用到 NGINX 的理由。最后，本章节介绍了 VPP LDP NGINX（直接使用 VPP 的 NGINX）的工作流程总览。

第三章　VPP LDP NGINX 的问题与挑战

对于 VPP LDP NGINX 而言，NGINX 的进程通过共享内存中的队列数据结构与 VPP 的进程交换信息和数据，从而利用了 VPP 的高性能用户态网络协议栈实现。但是，在 VPP LDP NGINX 的工作流程中，Session Lock Layer 中的锁竞争造成了 NGINX 的可扩展性及延迟问题。而移除 Session Lock Layer 又会引起 VPP Event Queue 中控制事件消息元数据乱序问题。另外，NGINX 的事件驱动逻辑使用的 Busy-Wait Polling epoll（忙等轮询式 epoll 机制）也会造成满负荷 CPU 占用率问题以及内核 epoll 事件检查开销问题。接下来，我们将具体介绍和分析这些问题。

3.1　可扩展性及延迟问题

在 Session Lock Layer 中，有两大类型的竞争条件对 NGINX 的可扩展性和延迟造成影响。两种竞争分别是对 Read/Write Lock 读写锁的竞争和对 Listener Session Lock 的竞争。

- **对 Read/Write Lock 的竞争**：对 Read/Write Lock 读写锁的竞争也可以分为两类：

  1). 写者与读者对读写锁的竞争：当一个 NGINX Worker 在接收或者关闭一个 Session 的时候，它会将读写锁的变量 *rw_cnt* 置为-1 以获取写锁，然后增删 Session Lock List 中的 Session Item。在这个 NGINX Worker 持有写锁的同时，其他的 NGINX Worker 是无法获取读锁或者写锁来对 Session Lock List 进行操作的。这样就会导致其他 NGINX Worker 无法继续进行连接的建立或者关闭操作，也无法继续读写 Session。由于 NGINX Worker 只会在连接建立或关闭时才会去获取写锁，因此这种写者与读者竞争读写锁的情况只会对短连接请求（Non-Keep-Alive Request）的并行处理造成负面影响。

  2). 读者与读者竞争读写变量 *rw_cnt*：由于变量 *rw_cnt* 是被所有 NGINX Worker 共享的，因此每个 NGINX Worker 对其进行读写时需要原子性地对其进行读写。当一个 NGINX Worker 每次获取或者释放读锁时，它都需要原子性地对变量 *rw_cnt* 增加或者减少 1。如果多个 NGINX Worker 同时去获取或者释放读锁，它们会去竞争读写变量 *rw_cnt*。因为每次一个 NGINX Worker 处理一个网络请求时，它都会获取和释放读锁至少各一次，因此高速率的请求处理都会伴随着对变量 *rw_cnt* 的大量竞争，并且这种竞争会对

长连接请求（Keep-Alive Request）的并行处理造成负面的影响。毕竟相比于短连接请求，长连接请求的处理速率是很高的并且伴随着大量读锁的获取和释放。

- **对 Listener Session Lock 的竞争**：所有的 NGINX Worker 都共享着 NGINX Master 在程序一开始就创建的 Listener Session。因此，所有的 NGINX Worker 也都共享着 Listener Session 的 Listener Session Lock。因此，当一个 NGINX Worker 在接收一个新连接的时候，为了访问 Listener Session 它必须首先通过竞争来获取 Listener Session Lock。这个时候，如果其他的 NGINX Worker 也去接收新的连接，它们也要去竞争 Listener Session Lock 以使用 Listener Session。所以，这种对 Listener Session Lock 的竞争会对频繁涉及连接建立的请求——短连接请求的并行处理造成负面的影响。

上述的对 Read/Write Lock 的竞争和对 Listener Session Lock 的竞争不仅会损害 NGINX 的可扩展性，还会增大请求处理的延迟。除此之外，Session Lock Layer 中的一些代码逻辑，比如查询 Session Lock List 中的 Session Item、增删 Session Item、获取或释放 Read/Write Lock 和 Session Lock，也会增加请求处理的延迟。

实际上，当处理高速率的网络请求时（在我们的实验中大约是 3,000,000 的 RPS），所有的 NGINX Worker 会对读写锁的变量 *rw_cnt* 进行大量的竞争读写（至少是 6,000,000/秒）。这种竞争会对长连接请求的并行处理产生巨大的负面影响。我们做了一次实验来证明这种竞争造成的 CPU 开销。在实验中，我们在一台服务端服务器上设置了 5 个 VPP Worker 和 8 个 NGINX Worker。在客户端机器上，我们总共使用 56 个 CPU 核向服务端的 NGINX 建立了 112 个连接，并为每个连接设置了 50 的并发量等级以向 NGINX 不断发送长连接请求，请求大小为 64B 的内存缓存文件。测试时间持续了 1 分钟。在整个实验中，我们使用 perf 命令去记录 8 个 NGINX Worker 的 CPU 开销。表 3–1展示了所有的 Non-Inline（非内联）VLS 函数的 CPU 开销。VLS 表示 *VPP Communication Library Locked Session*。函数 *vls_get_w_dlock()* 占据了 21.05% 的 CPU 开销。函数 *vls_get_w_dlock()* 实际只调用了内联函数 *vls_table_rlock()* 和静态函数 *vls_get_and_lock()*。内联函数 *vls_table_rlock()* 负责获取 Read/Write Lock 中的读锁，而静态函数 *vls_get_and_lock()* 则是负责获取 Session Lock。从表 3–1中我们可以看出，负责获取 Session Lock 的静态函数 *vls_get_and_lock()* 只有 0.68% 的 CPU 开销，因此 21.05% 的 CPU 开销中剩下的大部分则是花在了负责获取读锁的内联函数 *vls_table_rlock()* 上。由此可见，8 个 NGINX Worker 同时竞争读写变量 *rw_cnt* 以获取读锁的行为会引起大量的 CPU 开销和性能损耗。

表 3–1 8 个 NGINX Worker 的 VLS 函数的 CPU 开销

Table 3–1 CPU Overhead of VLS Functions of 8 NGINX Workers

CPU Overhead	Shared Object	Function Symbol
21.05%	libvppcom.so	vls_get_w_dlock
19.56%	libvppcom.so	vls_write_msg
9.53%	libvppcom.so	vls_recvfrom
7.08%	libvppcom.so	vls_epoll_wait
0.68%	libvppcom.so	vls_get_and_lock
0.33%	libvppcom.so	vls_get
0.09%	libvppcom.so	vls_app_fork_parent_handler
0.04%	libvppcom.so	vls_attr
0.01%	libvppcom.so	vls_alloc
0.00%	libvppcom.so	vls_close
0.00%	libvcl_ldpreload.so	vls_write_msg@plt
0.00%	libvppcom.so	vls_epoll_ctl
0.00%	libvcl_ldpreload.so	vls_recvfrom@plt
0.00%	libvppcom.so	vls_accept
0.00%	libvppcom.so	vls_share_vcl_session
0.00%	libvcl_ldpreload.so	vls_epoll_wait@plt

潜在地，我们可以像小节 4.1中设计的那样，为 VPP LDP NGINX 去除 Session Lock Layer。这种优化的可行性依据是——除了 Listener Session 以外，所有的 NGINX Worker 并不共享任何 Session，因此它们并不需要 Session Lock；另外，虽然 Listener Session 是被共享的（Listener Session Lock 也因此是被共享的），但是由于每个 NGINX Worker 最终只会操作自己独立的 Accept Event FIFO，因此 Listener Session Lock 也是不需要的且可以被去除。所以，最终 Session Lock List 中包含 Session Lock 的 Session Item 表项完全不需要放在共享的 Session Lock List 中。这样 Session Lock List 和它的 Read/Write Lock 就可以被安全地移除。

3.2 控制事件消息元数据乱序问题

将一个 Control Event Message（控制事件消息）原子性地添加进 VPP Event Queue 中的步骤可以分为以下三步：

- **从 Control Event Data Ring 中分配一个 Data Slot（数据槽）**：当一个 NGINX

Worker 向 VPP Event Queue 中添加一个控制事件消息时，根据 Control Event Data Ring 的 Tail 值，该控制事件消息首先会获取一个 Data Slot Index（数据槽索引）。然后 Control Event Data Ring 的 Tail 值便会被更新（Tail 值被增加 1）。该数据槽索引指向的数据槽 Data Slot 在之后便会和该控制事件消息绑定，用于存储控制事件消息的实际数据。

这一整个步骤是原子性的，因为需要读取和修改共享的 VPP Event Queue 中 Control Event Data Ring 的 Tail 值。

- **操作 Control Event Data Ring 的 Data Slot**：在这一步中，NGINX Worker 会初始化在第一步中从 Control Event Data Ring 分配到的 Data Slot，并将数据写入到这个 Data Slot 中。

  在这一步中，NGINX Worker 操作的 Data Slot 只属于自己。此步骤不涉及任何对共享数据结构的竞争。
- **将控制事件消息的元数据添加进 Message Metadata Queue**：在第三步中，于第一步获得的 Data Slot Index 将被放入控制事件消息的元数据中，然后这个元数据会被添加进 Message Metadata Queue 中。

  这一整个步骤也是原子性的。

  第三步不允许先于第二步，这是因为，只有控制事件消息的元数据被添加进 Message Metadata Queue 之后，这个控制事件消息才真正意义上被添加进 VPP Event Queue 中。如果先把元数据添加进 Message Metadata Queue，但是 Data Slot 还没有被写入数据，这样就会导致 VPP Worker 可能会从 VPP Event Queue 中取出该消息，并读取了 Data Slot 中没有意义的数据。

但是，在移除了 Session Lock Layer 之后，将一个控制事件消息添加进 VPP Event Queue 中的过程便不再是原子性的了。一个控制事件消息的三步入队过程可能会与其他控制事件消息的三步入队过程发生交替。而这种情况便会导致控制事件消息的元数据在 Message Metadata Queue 中发生乱序。

图 3–1展示了一个控制事件消息的元数据发生乱序的例子。在移除了 Session Lock Layer 后，NGINX Worker 2 和 NGINX Worker 3 打算将各自的控制事件消息添加进 VPP Event Queue 中：

- ①. NGINX Worker 2 的控制事件消息首先从 Control Event Data Ring 中获取了 Data Slot Index 2。在之后 Data Slot Index 2 指向的 Data Slot 2 将会与 NGINX Worker 2 的这条控制事件消息绑定。
- ②. NGINX Worker 2 开始将 Data Slot 2 进行初始化并将数据写入其中。
- ③. 接着，NGINX Worker 3 的控制事件消息从 Control Event Data Ring 中获

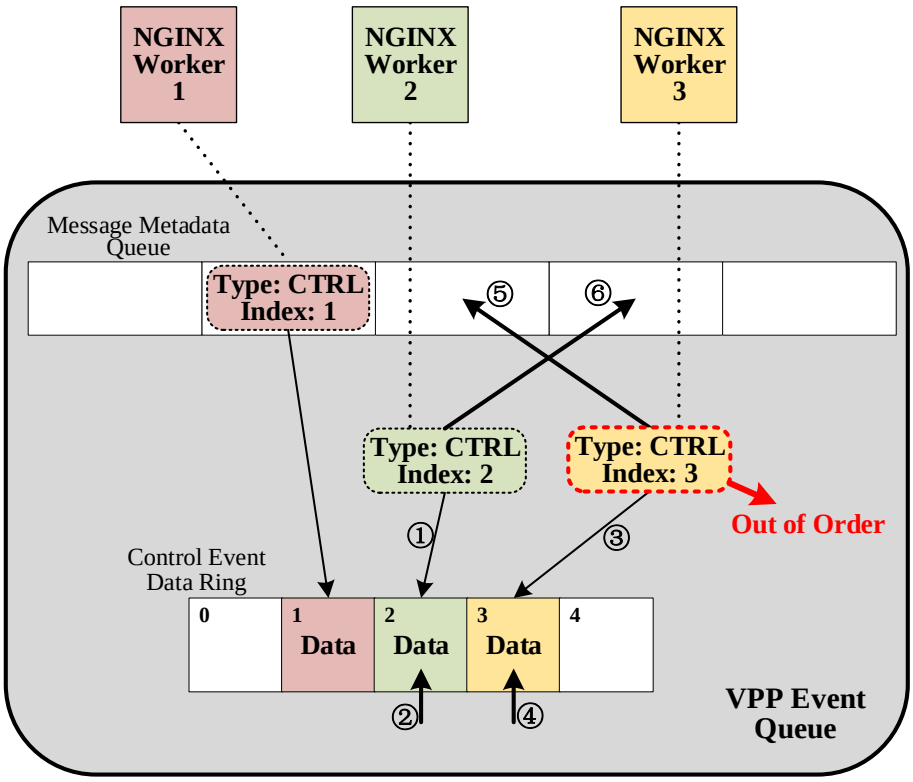

图 3–1 控制事件消息乱序示例

Figure 3–1 Out-of-Order Metadata of Control Event Messages

取了 Data Slot Index 3。在之后 Data Slot Index 3 指向的 Data Slot 3 将会与 NGINX Worker 3 的这条控制事件消息绑定。

- ④. NGINX Worker 3 开始将 Data Slot 3 进行初始化并将数据写入其中。
- ⑤. 然而，NGINX Worker 3 首先完成了对 Data Slot 的数据操作，于是 NGINX Worker 3 的控制事件消息的元数据会被立刻添加进 Message Metadata Queue 中。而这个元数据便发生了乱序。
- ⑥. NGINX Worker 2 在之后才完成对自己的 Data Slot 的操作，然后 NGINX Worker 2 的控制事件消息的元数据才被添加进 Message Metadata Queue 中。

结果，NGINX Worker 2 和 NGINX Worker 3 的两个控制事件消息的 Data Slot 的分配顺序和它们的元数据的入队顺序产生了不一致。

发生乱序的控制事件消息的元数据会在 VPP Worker 从 VPP Event Queue 中取出这条元数据时损坏其他控制事件消息的实际数据。当 VPP Worker 取出一个乱序的元消息时，VPP Worker 会将紧邻的另一条控制事件消息的 Data Slot 标记为空（通过简单地更新 Control Event Data Ring 的 Head 值）。如果此时又有大量的新

控制事件消息进入 VPP Event Queue，那么这个被标记为空的 Data Slot 会被新控制事件消息的数据覆盖。然而这个被标记为空的 Data Slot 其实还有未被处理的数据。这样这个未被处理的数据便会丢失。所以控制事件消息的元数据不能乱序。

为了在移除 Session Lock Layer 并且不重新引入 Giant Lock（巨锁）的情况下，让控制事件消息的 Data Slot 的分配顺序与它们的元数据的入队顺序一致，我们使用 Token-Based Lock-Free Order-Preserving Approach（基于令牌的无锁化保序机制）去保证控制事件消息入队 VPP Event Queue 时它们的元数据不会发生乱序。这个保序机制将在小节 4.2中被介绍。

3.3 忙等轮询式 epoll 机制问题

每个 NGINX Worker 使用 Busy-Wait Polling epoll(忙等轮询式 epoll 机制)，通过轮询的方式主动且不间断地检查 App Event Queue（应用事件队列）中来自 VPP Worker 的 Session 的用户态 epoll 事件和 Kernel epoll Wait Queue（内核 epoll 等待队列）中来自 NGINX Master 的 IPC Message（进程间通信消息）的内核态 epoll 事件。Busy-Wait Polling epoll 会对 VPP LDP NGINX 造成以下两个方面的问题：

- **满负荷的 CPU 占用率**：即使当前没有网络请求需要处理，也没有来自 NGINX Master 的 IPC Message（进程间通信消息）需要处理，但是 VPP LDP NGINX 的每个 NGINX Worker 也会通过轮询的方式主动且不间断地调用非阻塞式函数 *vls_epoll_wait()* 和非阻塞式函数 *epoll_wait()*，以分别检查 App Event Queue 中来自 VPP Worker 的 Session 的用户态 epoll 事件和 Kernel epoll Wait Queue 中来自 NGINX Master 的 IPC Message 的内核态 epoll 事件。而这会造成每个 NGINX Worker 的 CPU 占用率始终为百分之百，导致不必要的能源浪费和 CPU 资源的浪费。
- **进行内核 epoll 事件检查引起上下文切换开销**：一个 NGINX Worker 不断调用非阻塞式函数 *epoll_wait()* 去检查 Kernel epoll Wait Queue 中来自 NGINX Master 的 IPC Message 的内核态 epoll 事件。调用非阻塞式函数 *epoll_wait()* 会陷入内核，引起上下文切换开销。如果当前 NGINX Master 没有向 NGINX Worker 发送 IPC Message，并且 NGINX Worker 正在处理大量的网络请求，那么每个 NGINX Worker 仍然会不断地检查内核 epoll 事件以在内核 epoll 事件到来时能及时响应它们。因此，这种情况会引起因调用非阻塞式函数 *epoll_wait()* 而造成的上下文切换开销。

VPP 已经通过条件变量的方法为它的 Session 的用户态 epoll 机制实现了阻塞功能。当当前没有 Session 的用户态 epoll 事件到来时，阻塞式的 *vls_epoll_wait()*

函数会使一个 NGINX Worker 阻塞在条件变量 *condvar*（条件变量 *condvar* 存在于 NGINX Worker 的 App Event Queue 中）上，以进入进程等待状态并让出 CPU。条件变量 *condvar* 放置于 Protected Shared Memory（保护式共享内存）中，并且这个保护式共享内存仅仅能被这个 NGINX Worker 和 VPP Worker 访问。这个保护式共享内存的设计是为了防止 NGINX Worker 的 Session 数据结构（比如 App Event Queue）被其他进程潜在的漏洞代码访问和操作进而导致安全性问题。当一个 Session 的用户态 epoll 事件到达，VPP Worker 将会通过向条件变量 *condvar* 发送信号以唤醒这个 NGINX Worker。被唤醒的 NGINX Worker 便会接收这个 epoll 事件并对其进行处理。这个阻塞式的 epoll 机制仅对用户态的 Session 有效。

然而，潜在地，如小节 4.3所设计的那样，我们可以扩展 VPP 的这个阻塞式 epoll 机制，以在原本能同时接收用户态和内核态 epoll 事件的基础上实现阻塞功能。当 NGINX Master 向一个 NGINX Worker 发送 IPC Message 后，它可以通过间接地向保护式共享内存里的条件变量 *condvar* 发送信号来唤醒这个 NGINX Worker。被唤醒的 NGINX Worker 便可以处理 IPC Message 的内核态 epoll 事件。另外，在小节 4.3的设计中，我们还引入了一个用户态的 Flag U（标记 U）去指示是否有来自 NGINX Master 的 IPC Message 的内核态 epoll 事件。用户态标记 U 的引入可以消除因陷入内核进行内核 epoll 事件检查而导致的上下文切换开销。

3.4 本章小结

本章节介绍了在 VPP LDP NGINX（直接使用 VPP 的 NGINX）运行时，Session Lock Layer 对 VPP LDP NGINX 可扩展性的影响以及对其延迟的影响。其次，本章节还介绍了在移除 Session Lock Layer 时引起的 VPP Event Queue 中控制事件消息元数据乱序问题。最后，本章节还介绍了 NGINX Worker 的事件驱动逻辑使用的 Busy-Wait Polling epoll（忙等轮询式 epoll 机制）对 NGINX Worker 的 CPU 占用率的影响，以及该机制进行内核 epoll 事件检查造成的影响。

第四章　VPPNGX 的设计

如图 4–1所示，VPPNGX 的创新之处有以下三个部分：

- **可扩展性及延迟问题解决方案——VPPNGX API 使用的 VPP Session Index Passthrough（VPP 会话索引透传机制）**：为了移除 Session Lock Layer，我们必须将在 Session Lock List 中存放的所有 Session Index 取出放置到别的地方。为此，我们设计了 VPP Session Index Passthrough（VPP 会话索引透传机制），将 Session Index 作为 NGINX Worker 使用的文件描述符 FD 来使用。这样，NGINX Worker 的事件驱动逻辑、Listener Session 和普通 Session 便可以直接使用各自的 Session Index 来定位访问 VCL Layer 中的队列数据结构，以与 VPP Worker 的 Session Layer 进行通信。
- **控制事件消息元数据乱序问题解决方案——VPP Event Queue 使用的 Token-Based Lock-Free Order-Preserving Approach（基于令牌的无锁化保序机制）**：当多个 NGINX Worker 向 VPP Event Queue 中添加多个控制事件消息以控制 VPP Worker 时，由于移除了 Session Lock Layer，这些控制事件消息的入队过程将无法再被保证原子性。为了让入队过程无法再被保证原子性的控制事件消息的元数据在 VPP Event Queue 的 Message Metadata Queue 中不会发生乱序，我们为 VPP Event Queue 设计了 Token-Based Lock-Free Order-Preserving Approach（基于令牌的无锁化保序机制）。这个保序机制为每条控制事件消息入队 VPP Event Queue 的过程，引入一个令牌分配阶段（Token Allocating Stage）和一个令牌检查阶段（Token Checking Stage）来实现控制事件消息的元数据的保序功能。Token-Based Lock-Free Order-Preserving Approach 可以保证在移除 Session Lock Layer 之后 VPP Event Queue 中的控制事件消息的 Data Slot 分配顺序和这些消息的元数据进入 Message Metadata Queue 的顺序一致。
- **忙等轮询式 epoll 机制问题解决方案——NGINX Worker 事件驱动逻辑使用的 User-Space Unified Blocking epoll（用户态联合阻塞式 epoll 机制）**：现在，一个 NGINX Worker 的事件驱动逻辑可以使用 User-Space Unified Blocking epoll（用户态联合阻塞式 epoll 机制）在原来能同时接收用户态和内核态 epoll 事件的基础上，实现阻塞功能并减少进行内核 epoll 事件检查导致的上下文切换开销。当 App Event Queue 和 Kernel epoll Wait Queue 中都没有 epoll 事件时，这个 NGINX Worker 可以进入进程等待状态以让出 CPU，从而

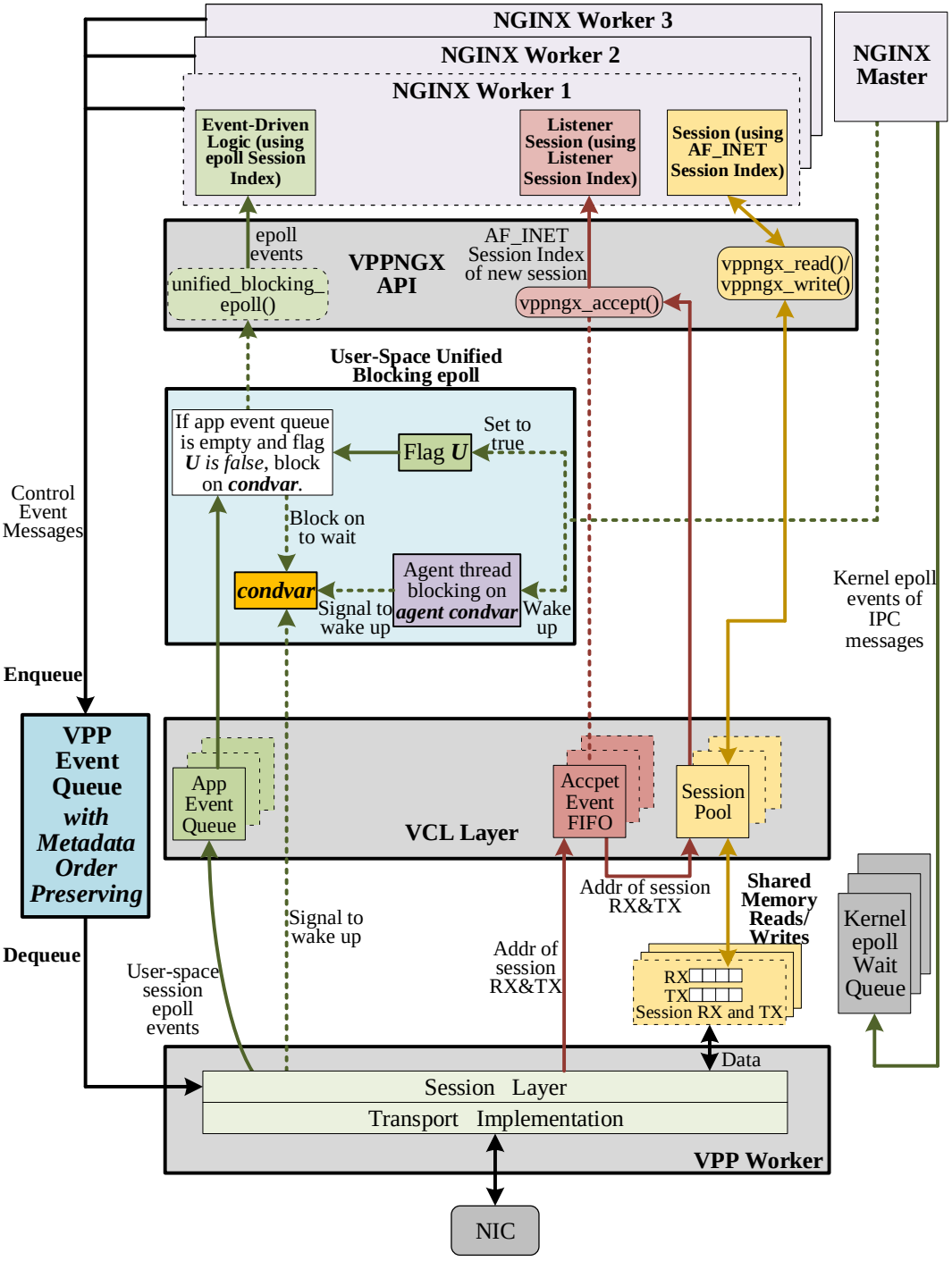

图 4–1 设计总览

Figure 4–1 Design Overview

减少 CPU 的占用率。当来自 VPP Worker 的 Session 的用户态 epoll 事件或者来自 NGINX Master 的 IPC Message 的内核态 epoll 事件到来时，VPP Worker 或者 NGINX Worker 能够主动地将处于进程等待状态的 NGINX Worker 唤醒，以让 NGINX Worker 去接收并处理这些到来的 epoll 事件。另外，检查内核态 epoll 事件不再需要通过系统调用进行——现在只需要检查一个用户态的标记便可以知道内核态 epoll 事件是否到来。

现在，我们将在接下来的小节中，详细地介绍 VPPNGX 引入的这些创新性优化策略。

4.1 可扩展性及延迟问题解决方案

我们将原本存放于 Session Lock List 中的 Session Index（epoll Session Index、Listener Session Index 和 AF_INET Session Index）从 Session Lock List 中取出，直接传递给 NGINX Worker 作为文件描述符 FD。原本，这些 Session Index 是一个一个地放置于 Session Lock List 中的每个 Session Item 表项中的。NGINX Worker 需要先获取 Session Lock Layer 中的读写锁才能去安全地访问 Session Lock List，并根据原来的文件描述符 FD 来找出 Session Index。而现在，NGINX Worker 使用的文件描述符 FD 就是一个一个的 Session Index。NGINX Worker 的事件驱动逻辑、Listener Session 和普通 Session 便可以直接使用各自的 Session Index 来定位访问 VCL Layer 中的队列数据结构，从而绕过了 Session Lock Layer。

因为 NGINX Worker 不再需要获取 Session Lock Layer 中的 Read/Write Lock 读写锁去访问 Session Lock List 以找到 Session Index，所以我们可以安全地删除 Read/Write Lock 读写锁和 Session Lock List，从而实现移除 Session Lock Layer。

现在新的连接建立过程如下：

- 当 VPP Worker 的传输层实现接收到一个新的连接的时候，VPP Worker 会为这条新的连接创建 Session RX/TX。

  然后 VPP Worker 将一个代表着新连接被建立的 epoll 事件放入 App Event Queue 中，并将新连接的 Session RX/TX 的地址放入 Accept Event FIFO 中。
- NGINX Worker 的事件驱动逻辑首先通过自己的文件描述符 epoll Session Index 找到自己的 App Event Queue 并从中取出一个 epoll 事件，发现新的连接到来需要被接收。

  于是 Listener Session 通过自己的文件描述符 Listener Session Index 找到自己的 Accept Event FIFO，并从中取出新的 Session RX/TX 的地址放入到 Session Pool 中的一个新的 Session Slot（会话槽）中。

- 接下来，指向这个新的 Session Slot 的 Session Index 将被作为 AF_INET Session Index 直接返回给 Listener Session。这个 AF_INET Session Index 便被视为新 Session 的 AF_INET FD。
  在这之后，NGINX Worker 的这个 Session 便可以直接使用它的 AF_INET Session Index 去找到它的 Session RX/TX，并通过这个 Session RX/TX 与 VPP Worker 中的传输层实现交换数据。

4.2 控制事件消息元数据乱序问题解决方案

我们为 VPP Event Queue 设计了一个 Token-Based Lock-Free Order-Preserving Approach（基于令牌的无锁化保序机制）来保证在移除 Session Lock Layer 之后，控制事件消息在入队 VPP Event Queue 时这些消息的 Data Slot 分配顺序和消息的元数据进入 Message Metadata Queue 的顺序一致。这种无锁化保序机制对于控制事件消息入队的效率非常重要。不然的话，在移除了 Session Lock Layer 之后，我们仍然需要引入巨锁（Giant Lock）来保证 VPP Event Queue 中的控制事件消息的 Data Slot 分配顺序和这些消息的元数据进入 Message Metadata Queue 的顺序一致。

如图 4–2所示，对于每条控制事件消息入队 VPP Event Queue 的过程，我们引入一个令牌分配阶段（Token Allocating Stage）和一个令牌检查阶段（Token Checking Stage）。令牌分配阶段和令牌检查阶段分别依赖于一个计数器 *token_num* 和一个计数器 *enqueue_num*。两个计数器拥有相同的初始值 0（为了能够正常服务应用程序启动后到来的第一个控制事件消息）。令牌分配阶段负责为一个控制事件消息分发一个令牌编号，该令牌编号代表着这个控制事件消息的 Data Slot 在 Control Event Data Ring 中的分配顺序。令牌检查阶段负责在将控制事件消息的元数据加入到 Message Metadata Queue 中前，检查该控制事件消息所持有的的令牌编号是否是当前对应的应入队元数据的消息的令牌编号。

在引入了令牌分配阶段和令牌检查阶段后，将一条控制事件消息添加进 VPP Event Queue 的过程可以分为以下新的三个步骤：

- 首先，当一个 NGINX Worker 向 VPP Event Queue 中添加一个控制事件消息时，根据 Control Event Data Ring 的 Tail 值该控制事件消息会首先获取一个 Data Slot Index（数据槽索引）。然后 Control Event Data Ring 的 Tail 值便会被更新（Tail 值被增加 1）。该数据槽索引指向的数据槽 Data Slot 在之后便会和该控制事件消息绑定，用于存储控制事件消息的实际数据。接着，通过读取计数器 *token_num* 的值，该控制事件消息获得一个令牌编号，然后计数器 *token_num* 的值被增加 1。

这一整个步骤是原子性的，因为需要读取和修改共享的 VPP Event Queue 中 Control Event Data Ring 的 Tail 值，以及读取和修改共享的计数器 *token_num* 的值。

- 在第二步中，NGINX Worker 会初始化在第一步中从 Control Event Data Ring 分配到的 Data Slot，并将数据写入到这个 Data Slot 中。在这一步中，NGINX Worker 操作的 Data Slot 只属于自己。
  此步骤不涉及任何对共享数据结构的竞争。因此，此步骤是无锁化的。当多个 NGINX Worker 都在将自己的控制事件消息添加进 VPP Event Queue 时，所有控制事件消息的这一步均可以被并行化地执行。
- 第三，NGINX Worker 在将自己的控制事件消息的元数据加入到 Message Metadata Queue 之前，这个 NGINX Worker 会在令牌检查阶段首先检查自己的这个控制事件消息持有的令牌编号是否等于计数器 *enqueue_num* 的值。如果令牌编号和计数器 *enqueue_num* 的值相等，则这条控制事件消息的元数据可以被直接加入到 Message Metadata Queue 中且不需要等待。在此之后，NGINX Worker 便将计数器 *enqueue_num* 的值加 1。将控制事件消息的元数据加入到 Message Metadata Queue 中和将计数器 *enqueue_num* 的值加 1 的整个过程是原子性的，不可以本分割。
  相反，如果控制事件消息持有的令牌编号和计数器 *enqueue_num* 的值不相等，这意味着，至少一条控制事件消息的元数据尚未被加入到 Message Metadata Queue 中。而当前的控制事件消息则需要等待。只有应该在当前控制事件消息元数据之前入队的所有控制事件消息的元数据都进入了 Message Metadata Queue 中后，当前的控制事件消息的元数据才能够被添加进 Message Metadata Queue 中。

图 4–2也展示了一个元数据保序的例子。我们将通过该例子说明 Token-Based Lock-Free Order-Preserving Approach（基于令牌的无锁化保序机制）是如何工作的。在该例子中，NGINX Worker 2 和 NGINX Worker 3 打算将它们的控制事件消息添加进 VPP Event Queue。

- ①. 首先 NGINX Worker 2 的控制事件消息第一个获得了 Data Slot Index 2。这样 Data Slot 2 便与 NGINX Worker 2 的控制事件消息绑定在一起。然后，NGINX Worker 2 通过读取计数器 *token_num* 的值（值为 N），为其控制事件消息获得了编号为 N 的令牌。
- ②. NGINX Worker 2 将计数器 *token_num* 的值加 1。这样计数器 *token_num* 的值从 N 变为了 N+1。

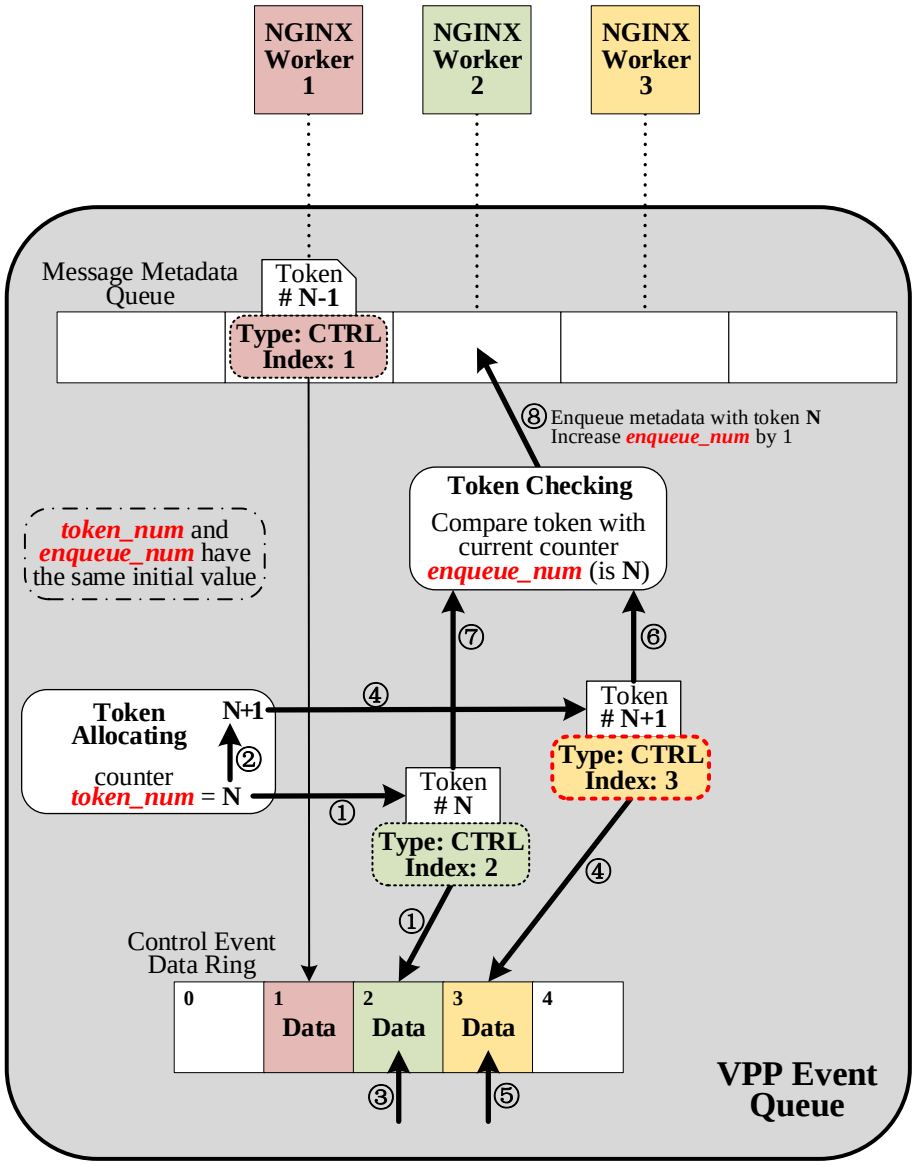

图 4–2 控制事件消息元数据保序示例

Figure 4–2 Order Preserving for Metadata of Control Event Messages

- ③. NGINX Worker 2 开始将 Data Slot 2 初始化并向其中写入相应的控制事件消息数据。
- ④. 接着，NGINX Worker 3 的控制事件消息获得了 Data Slot Index 3。这样 Data Slot 3 便与 NGINX Worker 3 的控制事件消息绑定在一起。然后，NGINX Worker 3 通过读取计数器 *token_num* 的值（值为 N+1），为其控制事件消息获得了编号为 N+1 的令牌。接下来，NGINX Worker 3 将计数器 *token_num* 的值加 1。这样计数器 *token_num* 的值从 N+1 变为了 N+2。
- ⑤. NGINX Worker 3 开始将 Data Slot 3 初始化并向其中写入相应的控制事件消息数据。
- ⑥. NGINX Worker 3 首先完成了向 Data Slot 3 写入控制事件消息数据的工作，然后发现其控制事件消息持有的令牌编号 N+1 不等于当前计数器 *enqueue_num* 的值 N。于是 NGINX Worker 3 通过循环方式不断地比较其控制事件消息持有的令牌编号和计数器 *enqueue_num* 的值来进行等待。
- ⑦. NGINX Worker 2 后来才完成了向 Data Slot 2 写入控制事件消息数据的工作，并且发现其控制事件消息持有的令牌编号 N 等于当前计数器 *enqueue_num* 的值 N。
- ⑧. 于是 NGINX Worker 2 首先将自己的控制事件消息的元数据添加进 Message Metadata Queue 中，并将计数器 *enqueue_num* 的值加 1。这样，计数器 *enqueue_num* 的值被更新为 N+1。

  紧接着，NGINX Worker 3 发现其控制事件消息持有的令牌编号 N+1 已经等于当前计数器 *enqueue_num* 的值 N+1，于是 NGINX Worker 3 便将自己的控制事件消息的元数据添加进 Message Metadata Queue 中。

4.3 忙等轮询式 epoll 机制问题解决方案

如图 4–1和图 4–3所示，我们设计了 User-Space Unified Blocking epoll（用户态联合阻塞式 epoll 机制）去替换 NGINX Worker 事件驱动逻辑原本使用的 Busy-Wait Polling epoll（忙等轮询式 epoll 机制）。使用 User-Space Unified Blocking epoll 可以在原来能同时接收用户态和内核态 epoll 事件的基础上，实现阻塞功能减少 CPU 占用率，并减少进行内核 epoll 事件检查引起的上下文切换开销。我们的 User-Space Unified Blocking epoll 包含下面几个组成部分：

- **用于内核 epoll 事件检查的用户态指示器**：我们为每一个 NGINX Worker 引入了一个用户态的指示器 Flag *U*（标记 *U*）去指示内核 epoll 事件的到来情况。Flag *U* 为真意味着 Kernel epoll Wait Queue 中存在有待处理的内核

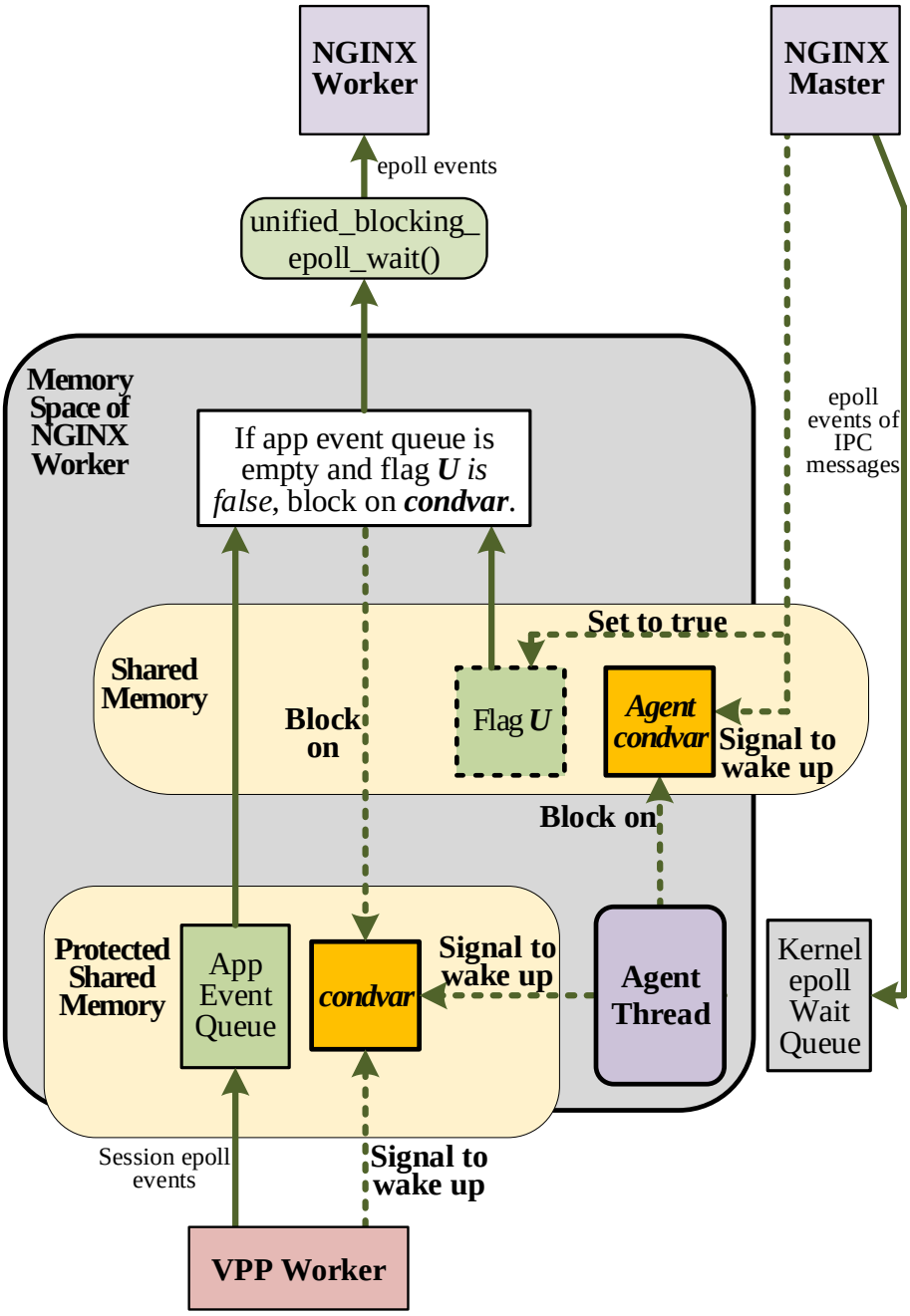

图 4–3 用户态联合阻塞式 epoll 机制

Figure 4–3 User-Space Unified Blocking epoll

epoll 事件。一个 NGINX Worker 通过检查 Flag *U* 的真值来判断 Kernel epoll Wait Queue 中是否有内核 epoll 事件，而不是先陷入内核态再去检查内核 epoll 事件。这样便可以减少因进行内核 epoll 事件检查而引起的上下文切换开销。

- **保护式共享内存**：保护式共享内存（Protected Shared Memory）仅仅可由 VPP Worker 和一个 NGINX Worker 访问。一个保护式共享内存是由 VPP Worker 创建的。VPP Worker 引入保护式共享内存是为了防止一个 NGINX Worker 的 Session 数据结构（比如 App Event Queue 等）被其他应用进程潜在的漏洞代码访问和破坏。
- **条件变量 *condvar***：如果一个 NGINX Worker 的 App Event Queue 没有用户态 epoll 事件并且其 Flag *U* 为假（没有内核态 epoll 事件），那么 NGINX Worker 必须进入进程等待状态以让出 CPU，降低 CPU 占用率。为了进入进程等待状态，NGINX Worker 会阻塞在内核提供的条件变量 *condvar*（由 VPP Worker 通过系统调用创建）上。条件变量 *condvar* 是放在保护式共享内存中的。如果 App Event Queue 中有 Session 的用户态 epoll 事件到来，则 VPP Worker 会通过向条件变量 *condvar* 发送信号来将阻塞在其上的 NGINX Worker 从进程等待状态中唤醒。
- **代理线程 Agent Thread 和代理条件变量 *agent condvar***：因为条件变量 *condvar* 是放在仅仅可由 VPP Worker 和一个 NGINX Worker 访问的保护式共享内存中的，因此 NGINX Master 进程无法访问并向条件变量 *condvar* 发送信号以将 NGINX Worker 从进程等待状态中唤醒。而我们为每一个 NGINX Worker 引入一个 Agent Thread 代理线程来帮助 NGINX Master 间接访问并间接向条件变量 *condvar* 发送信号以将 NGINX Worker 从进程等待状态中唤醒。当 NGINX Worker 正处在进程等待状态中时，代理线程 Agent Thread 也必须进入等待状态以让出 CPU，从而降低 CPU 的占用率。而代理线程 Agent Thread 进入等待状态的方式则是阻塞在一个也由内核提供的代理条件变量 *agent condvar*（通过系统调用创建）上。当 NGINX Master 向一个 NGINX Worker 发送了 IPC Message 后，NGINX Master 会通过向代理条件变量 *agent condvar* 发送信号来将代理线程 Agent Thread 从等待状态中唤醒。被唤醒的代理线程 Agent Thread 将会向 NGINX Worker 阻塞于的条件变量 *condvar* 发送信号，以将这个 NGINX Worker 从进程等待状态中唤醒。

图 4–3展示了 User-Space Unified Blocking epoll 的详细的内存布局。一个 NGINX Worker 的 App Event Queue 和条件变量 *condvar* 被存放在保护式共享内存中，

并且这个保护式共享内存仅仅可由 VPP Worker 和一个 NGINX Worker 访问。Flag *U*（标记 *U*）和代理条件变量 *agent condvar* 被存放在一个 NGINX Master 和所有 NGINX Worker 均可访问的普通共享内存中。NGINX Worker 的代理线程 Agent Thread 既可以访问条件变量 *condvar*，也可以访问代理条件变量 *agent condvar*。

4.4 本章小结

本章节首先介绍了 VPPNGX 为了安全移除 Session Lock Layer 而引入的 VPP Session Index Passthrough（VPP 会话索引透传机制）。该机制最终让 VPPNGX 拥有了良好的可扩展性以及更低的请求处理延迟。其次，本章节还介绍了在移除 Session Lock Layer 后为 VPP Event Queue 引入的 Token-Based Lock-Free Order-Preserving Approach（基于令牌的无锁化保序机制）。该保序机制保证了控制事件消息在入队 VPP Event Queue 时它们的元数据不会发生乱序。最后，本章节介绍了 User-Space Unified Blocking epoll（用户态联合阻塞式 epoll 机制）。该机制替代了原来的 Busy-Wait Polling epoll（忙等轮询式 epoll 机制），在网络负载较低时能够减少 NGINX Worker 的 CPU 占用率，并能够减少因进行内核 epoll 事件检查而造成的 CPU 开销。

第五章　VPPNGX 的技术实现

5.1　可扩展性及延迟问题解决方案的实现

我们首先简单介绍一下 VCL Layer 中的 VCL 函数。如前面章节所述，VPP LDP API 能够将应用程序原本使用的 libc 实现的 Socket 函数通过 LD_PRELOAD 方式重定向到 VPP 实现的函数库中，并且不需要修改应用程序的源代码。在没有移除 Session Lock Layer 的时候，NGINX Worker 首先调用 LDP 函数去获取 Session Lock Layer 中的读写锁，并访问 Session Lock List 中的 Session Item。在获取了 Session Item 中的 Session Lock 和 Session Index 后，NGINX Worker 持有 Session Lock 并调用 VCL Layer 中的 VCL 函数 *vppcom_session_*()* 来对 Session 进行操作。Session Index 会被作为参数传递给这些 VCL 函数 *vppcom_session_*()* 用作 Session Handler（会话句柄）。

为了移除 Session Lock Layer（也就是将 Read/Write Lock 和 Session Lock List 安全删除），我们修改了 VPP LDP API 中所有的 LDP 函数接口，以将 VCL Layer 中的所有 VCL 函数直接暴露给 NGINX Worker，从而绕过 Session Lock Layer。我们将这些 LDP 函数 *ldp_*()* 重命名为 *vppngx_*()* 以将它们与原来的 LDP 函数进行区别。Session Index 的实际创建和操作都是通过 VCL 函数进行的。因此，一个 NGINX Worker 直接通过 VCL 函数 *vppcom_session_accept()* 或者 *vppcom_session_create()* 获取一个新的 Session Index，并将新的 Session Index 作为文件描述符 FD 直接使用。为了操作 Session，NGINX Worker 通过函数 *vppngx_*()* 直接将作为文件描述符 FD 的 Session Index 作为参数传递给 VCL 函数。这样的话，原本存放于 Session Lock List 中的 Session Index 就直接透传给 NGINX Worker，从而绕过了 Session Lock Layer，使得其可以被安全地移除。

5.2　控制事件消息元数据乱序问题解决方案的实现

令牌分配阶段（Token Allocating Stage）和令牌检查阶段（Token Checking Stage）分别依赖于一个计数器变量 *token_num* 和一个计数器变量 *enqueue_num*。计数器变量 *token_num* 用于向控制事件消息分配带有编号的令牌，而计数器变量 *enqueue_num* 则用于检查控制事件消息分配到的令牌编号并决定该控制事件消息是否可以进入 Message Metadata Queue。这两个计数器变量都是无符号整型变量，并且要么都是无符号整型（unsigned int）变量，要么都是无符号长整型（unsigned

long int）变量。两个计数器变量的初始化值均为 0（为了能够正常服务应用程序启动后到来的第一个控制事件消息）。

一个 NGINX Worker 在令牌分配阶段会读取计数器变量 *token_num* 的值并将其值赋值给该 NGINX Worker 的一个局部变量 *msg_token_num*。局部变量 *msg_token_num* 代表着这个 NGINX Worker 当前正要入队的控制事件消息所持有的令牌编号。之后，NGINX Worker 便将计数器变量 *token_num* 的值加 1。在 NGINX Worker 将自己的新的控制事件消息的元数据添加进 Message Metadata Queue 中之前，它要将计数器变量 *enqueue_num* 与自己的局部变量 *msg_token_num* 进行比较。这个操作通过代码“while (PREDICT_TRUE (*enqueue_num* == *msg_token_num*))”进行实现。宏命令 PREDICT_TRUE 最终调用编译器的内建函数 *__builtin_expect()* 来进行分支预测，以优化流水线 CPU 的处理性能。在实验中我们发现表达式“while (PREDICT_TRUE (*enqueue_num* == *msg_token_num*))”为真的概率约为 49/50，因此我们使用宏命令 PREDICT_TRUE 而不是宏命令宏命令 PREDICT_FALSE 来进行分支预测。表达式结果为真说明新的控制事件消息的元数据没有发生乱序，可以被直接添加进 Message Metadata Queue 中；而如果表达式结果为假，表达式中的 while 循环则会一直持续下去以不断地进行表达式真值的判断，直到其他的 NGINX Worker 更新了计数器变量 *enqueue_num* 并使得表达式的结果为真。

如果计数器变量 *token_num* 和计数器变量 *enqueue_num* 都达到了当前无符号整型变量的最大值，它们都将会重新从 0 开始计数。

计数器变量 *enqueue_num* 必须是 volatile 易变类型的变量。这是因为在执行代码“while (PREDICT_TRUE (*enqueue_num* == msg_*token_num*ber))”时，如果计数器变量 *enqueue_num* 不是 volatile 易变类型的变量的话，那么 *enqueue_num* 的值将会由于编译器的优化而直接从寄存器中读取。当其他 NGINX Worker 更新了计数器变量 *enqueue_num* 时，当前这个 NGINX Worker 仍然会从寄存器中读取计数器变量 *enqueue_num* 的旧值。然而，如果计数器变量 *enqueue_num* 是 volatile 易变类型的变量，那么当前这个 NGINX Worker 将会从内存中读取计数器变量 *enqueue_num* 的最新的值，而不是从寄存器中读取计数器变量 *enqueue_num* 的旧值。

5.3 忙等轮询式 epoll 机制问题解决方案的实现

我们的 User-Space Unified Blocking epoll（用户态联合阻塞式 epoll 机制）的实现通过函数接口 *unified_blocking_epoll_wait()* 来提供给 NGINX Worker 使用。通过 VPP 提供的函数 *clib_mem_alloc()*，我们在 NGINX Master 和 NGINX Worker 均可以访问的共享内存中，为每个 NGINX Worker 都分配了一个代理条件变

量 *agent condvar* 和一个布尔变量 *U*。代理条件变量 *agent condvar* 是通过函数 *pthread_cond_init()* 创建的。连同代理条件变量 *agent condvar* 和布尔变量 *U*，我们还通过函数 *pthread_create()* 为每个 NGINX Worker 创建了一个 Agent Thread（代理线程）。

如果没有来自 NGINX Master 的 IPC Message 和内核 epoll 事件，一个 NGINX Worker 的代理线程 Agent Thread 将会通过函数 *pthread_cond_wait()* 阻塞于代理条件变量 *agent condvar* 上以让出 CPU。当 NGINX Master 通过函数 *ngx_pass_open_channel()* 向一个 NGINX Worker 发送了一个 IPC Message，NGINX Master 将会把这个 NGINX Worker 的布尔变量 *U* 设置为真，并通过函数 *pthread_cond_broadcast()* 向它的代理条件变量 *agent condvar* 发送信号以将代理线程 Agent Thread 唤醒。然后被唤醒的代理线程 Agent Thread 会通过函数 *pthread_cond_signal()* 向保护式共享内存中的条件变量 *condvar* 发送信号，以将 NGINX Worker 唤醒。

当一个 NGINX Worker 调用函数 *unified_blocking_epoll_wait()* 后，它会首先使用分支预测宏 PREDICT_FALSE 来检查其布尔变量 *U* 是否为真，并判断其 App Event Queue 是否有 Session 的用户态 epoll 事件。如果布尔变量 *U* 为真，则这个 NGINX Worker 将会执行代码 “while (*epoll_wait()*)” 来将来自 NGINX Master 的 IPC Message 的内核 epoll 事件取出，并将布尔变量 *U* 设置为假。代码 “while (*epoll_wait()*)” 中的 while 循环是为了防止这么一种情况——在 NGINX Worker 第一次调用函数 *epoll_wait()* 时，可能内核 epoll 事件还尚未进入 Kernel epoll Wait Queue 中，这样会导致 NGINX Worker 扑空且没法取出内核 epoll 事件。如果 App Event Queue 不为空，那么其中的 Session 的用户态 epoll 事件将被取出。而如果布尔变量 *U* 为假并且 App Event Queue 没有用户态 epoll 事件，则 NGINX Worker 会阻塞在条件变量 *condvar* 上，等待 epoll 事件到来时被 NGINX Master 或者 VPP Worker 唤醒。被唤醒后，NGINX Worker 会像前面的步骤一样，再次检查布尔变量 *U* 和 App Event Queue 以从 Kernel epoll Wait Queue 或 App Event Queue 中取出到来的 epoll 事件。

值得注意的是，使用条件变量 *condvar* 的时候，经常会出现函数 *pthread_cond_broadcast()* 先于函数 *pthread_cond_wait()* 或者函数 *pthread_cond_timewait()* 被调用的情况。这种情况会导致 VPP Worker 向条件变量 *condvar* 发送的信号丢失。VPP Worker 在向 App Event Queue 中添加 Session 的用户态 epoll 事件后，调用函数 *pthread_cond_broadcast()* 时很有可能 NGINX Worker 进程尚未阻塞在条件变量 *condvar* 上。而应对这种情况的方法则是让

NGINX Worker 首先检查 App Event Queue 中是否有 epoll 事件。如果有则直接取出它们进行处理。这样的话即使 NGINX Worker 错过了 VPP Worker 向条件变量 *condvar* 发送的信号，这个 NGINX Worker 仍然能够正确获取和处理 Session 的用户态 epoll 事件。而代理线程 Agent Thread 在向条件变量 *condvar* 发送信号时，该信号也有可能出现丢失（也就是说 NGINX Worker 此时尚未阻塞在条件变量 *condvar* 上）。这种情况的应对方法和前面检查 App Event Queue 一样，NGINX Worker 先检查它的用户态布尔变量 *U* 来判断 Kernel epoll Wait Queue 中是否有内核 epoll 事件。如果有则直接从 Kernel epoll Wait Queue 中取出这些内核 epoll 事件进行处理。这样的话即使 NGINX Worker 错过了代理线程 Agent Thread 向条件变量 *condvar* 发送的信号，这个 NGINX Worker 仍然能够正确获取和处理内核态的 epoll 事件。

5.4 本章小结

本章节从代码实现的角度，分别介绍了 VPP Session Index Passthrough（VPP 会话索引透传机制）、Token-Based Lock-Free Order-Preserving Approach（基于令牌的无锁化保序机制）以及 User-Space Unified Blocking epoll（用户态联合阻塞式 epoll 机制）的技术实现。VPP Session Index Passthrough 用于解决可扩展性及延迟问题。Token-Based Lock-Free Order-Preserving Approach 用于解决控制事件消息元数据乱序问题。User-Space Unified Blocking epoll 则用于解决忙等轮询式 epoll 机制问题。

第六章　测试与评估

在本章节中，我们将通过实验评估 VPPNGX 的可扩展性（基于 RPS 和吞吐量）、请求处理延迟、CPU 占用率以及内核 epoll 事件检查开销。

6.1　实验配置

6.1.1　实验基准

如今大量的研究已经表明，用户态的 TCP/IP 协议栈要比基于内核的 TCP/IP 协议栈有着更高的吞吐量。另外，VPP 声明其要比现在的几乎所有的用户态 TCP/IP 协议栈在性能方面更加出色。因此，我们在接下来的实验中，仅把我们的 VPPNGX 与 Kernel-Based NGINX（基于内核的 NGINX）、VPP LDP NGINX（直接使用 VPP 的 NGINX）和 F-Stack NGINX 进行对比。其中，F-Stack[54] 和 VPP 被我们选作典型的用户态网络协议栈的代表。

在将 VPPNGX 与这些作为实验基准的 TCP/IP 协议栈框架进行对比的时候，我们选择测试 VPPNGX 的可扩展性、延迟（处理时间/请求）、CPU 占用率以及内核 epoll 事件检查开销。

6.1.2　服务器配置

我们准备了两台 Dell PowerEdge R730 服务器——服务器 A（Server A）和服务器 B（Server B）。两台服务器通过两个 40GbE 网卡进行网线直连。两台服务器的具体配置如表 6–1所示。

6.1.3　实验方法

- **可扩展性测试方法**：我们设计了一个既有长连接请求又有短连接请求的混合请求场景去验证去除 Session Lock Layer 后的 VPPNGX 的可扩展性。对于 VPPNGX、Kernel-Based NGINX、VPP LDP NGINX 和 F-Stack NGINX，所有的 NGINX Worker 都被放置在服务器 A 的 NUMA Node 0 上。对于 Kernel-Based NGINX，内核的 TSO（TCP Segmentation Offloading）功能被打开（通过命令“sudo ethtool -K enp8s0f1 tso on”）。我们也通过命令“sudo ethtool -L enp8s0f1 combined 56”为 Kernel-Based NGINX 将网卡传输和接收队列绑定到 56 个 CPU 核上，使得 56 个 CPU 核可以同时参与网络包的收发工

表 6–1 服务器配置

Table 6–1 Server Configuration

	Server A	**Server B**
Server Processor	2 × Xeon E5-2690 v4 2.6GHz	2 × Xeon E5-2680 v4 2.4GHz
Hyper-Threading	On	On
CPU Turbo Boost	Off	On
Logical Cores	56	56
NIC	1 × XL710 - 40GbE	1 × XL710 - 40GbE
NUMA Node for NIC	Node 0	Node 0
Memory	128G	128G
Huge Pages	20 × 1G	None

作。对于 VPPNGX 和 VPP LDP NGINX，我们在服务器 A 的 NUMA Node 0 上为他们都设置了 5 个 VPP 工作线程。NGINX 接收长连接请求的功能在 nginx.conf 配置文件中被打开。由客户端应用程序请求的文件被直接写入 nginx.conf 文件中作为内存缓存文件（Memory Cached File）。当 NGINX 启动时它会读取 nginx.conf 中的配置，将内存缓存文件的内容直接读入内存进行缓存。服务器 B 上的客户端应用程序负责向服务器 A 上的 NGINX 发送请求，并计算 NGINX 的 RPS（Requests per Second）和吞吐量。在服务器 B 上，我们选择了一些 CPU 核并将它们分成了两组。其中一组有 36 个 CPU 核，并且其中的 18 个 CPU 核是在 NUMA Node 1 上；另外一组有 10 个 CPU 核，并且其中的 5 个 CPU 核也是在 NUMA Node 1 上。在第一组的所有 36 个 CPU 核中，我们通过在每个 CPU 核上使用命令 wrk[87] 执行 “wrk -t1 -c50 -d60s URL” 以总并发量 1800 来不断地向 NGINX 发送长连接请求，并持续 60 秒。同时，我们也在服务器 A 和服务器 B 上均设置了 1000 个虚拟的 IP 地址，并使用第二组的 10 个 CPU 核以 20000/秒的总速率向 NGINX 不断地发送短连接请求。短连接请求的发送是通过 ab（ApacheBench）命令实现的。

- **延迟测试方法**：我们对 VPPNGX、Kernel-Based NGINX、VPP LDP NGINX 和 F-Stack NGINX 进行延迟测试时，延迟的测试标准是测试每个请求从发送直到接收到回复之间的时间（Time per Request，处理时间/请求）。我们使用 ab（ApacheBench）命令在服务器 B 上不断请求不同大小的内存缓存

文件并计算请求处理的延迟。这些内存缓存文件的大小是从 0B 到 1200B。对于每种大小的文件，我们使用命令“ab -n 100 -c 1 -k URL”进行 30 组的测试，并通过箱式图展示所有的延迟测试结果。

- **NGINX Worker 的 CPU 占用率测试方法**：测试 NGINX Worker 的 CPU 占用率时，我们只对比了 VPPNGX 和 VPP LDP NGINX。我们在服务器 A 上为 VPPNGX 和 VPP LDP NGINX 均设置了 5 个 VPP Worker 和 8 个 NGINX Worker。在服务器 B 上我们用客户端测试程序不断以不同的并发量（并发量等级从 0 到 200）向服务器 A 上的 NGINX 发送请求，请求 64B 大小的内存缓存文件。同时在服务器 A 上我们用 htop 命令监测并记录每个 NGINX Worker 的 CPU 占用率。对于每一种并发等级的测试，我们为每一个 NGINX Worker 均进行了 30 次实验记录，并将 CPU 占用率的最大值、中值和最小值记录到最终的实验结果图中。
- **内核 epoll 事件检查的 CPU 开销测试方法**：我们只测试 VPPNGX 和 VPP LDP NGINX 在做内核 epoll 事件检查时的 CPU 开销。为了进行测试，我们使用 perf 命令去测量相关函数的 CPU 消耗量。对于 VPP LDP NGINX，我们通过 perf 对非阻塞式函数 *epoll_wait()* 进行 CPU 开销的测量。VPP LDP NGINX 使用非阻塞式函数 *epoll_wait()* 陷入内核来检查内核里的 epoll 事件。对于 VPPNGX，我们测量的是 VPPNGX 通过检查用户态空间的 Flag *U*（标记 *U*）来判断是否有内核 epoll 事件的 CPU 开销。为了能够通过 perf 命令测出检查标记 *U* 的 CPU 开销，我们将检查标记 *U* 的判断语句放入了一个静态函数中，并使用 perf 命令来测量该静态函数的 CPU 开销，以达到间接测量的目的。在服务器 B 上我们用客户端测试程序向服务器 A 上的 NGINX 发送请求，请求不同大小的内存缓存文件。这些内存缓存文件的大小范围是从 0B 一直到 1200B。

6.2 测试结果

6.2.1 可扩展性测试结果

图 6–1、图 6–2、图 6–3和图 6–4共同展示了不同 NGINX 实现的可扩展性测试的结果。结果分别以 RPS 和吞吐量进行展示。所有图的横坐标代表的是 NGINX Worker 的数量（从 1 到 8）。该实验结果分为 64B 小文件测试结果和 1KB 大文件测试结果。

图 6–1和图 6–2分别展示了客户端应用程序请求 64B 内存缓存小文件时的 RPS 和吞吐量结果。图 6–3和图 6–4则分别展示了客户端应用程序请求 1KB 内

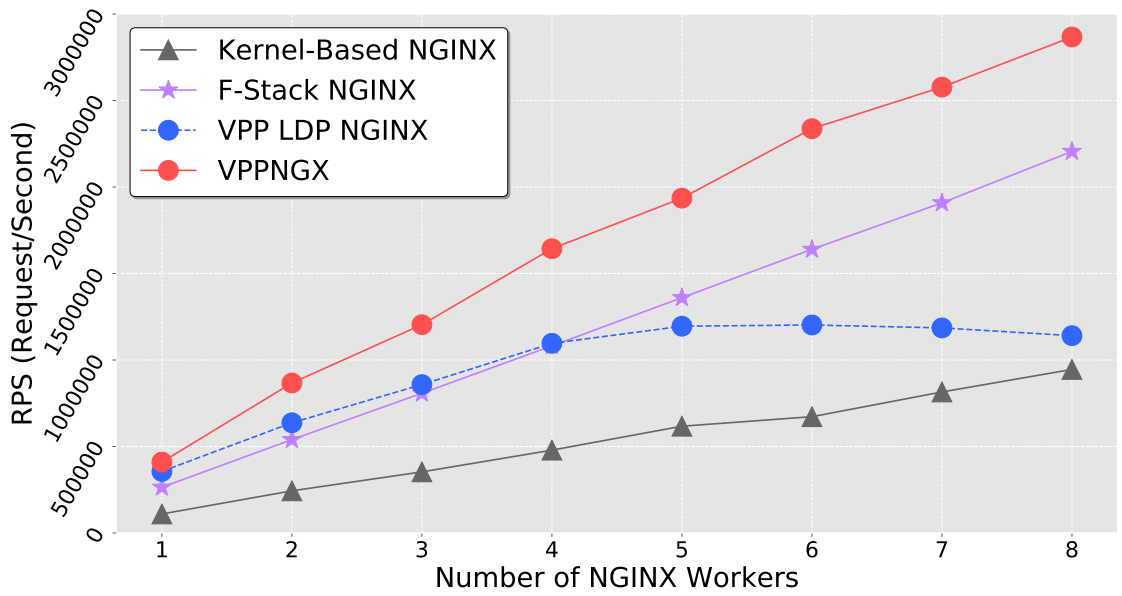

图 6–1 64B 小文件的 RPS

Figure 6–1 RPS - 64B Small File

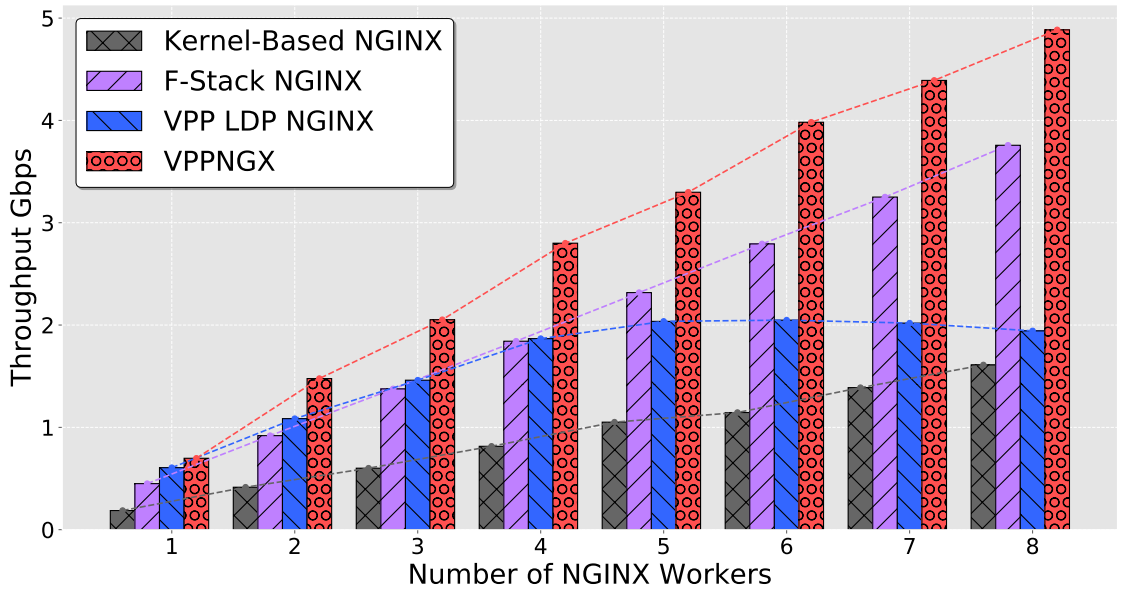

图 6–2 64B 小文件的吞吐量

Figure 6–2 Throughput - 64B Small File

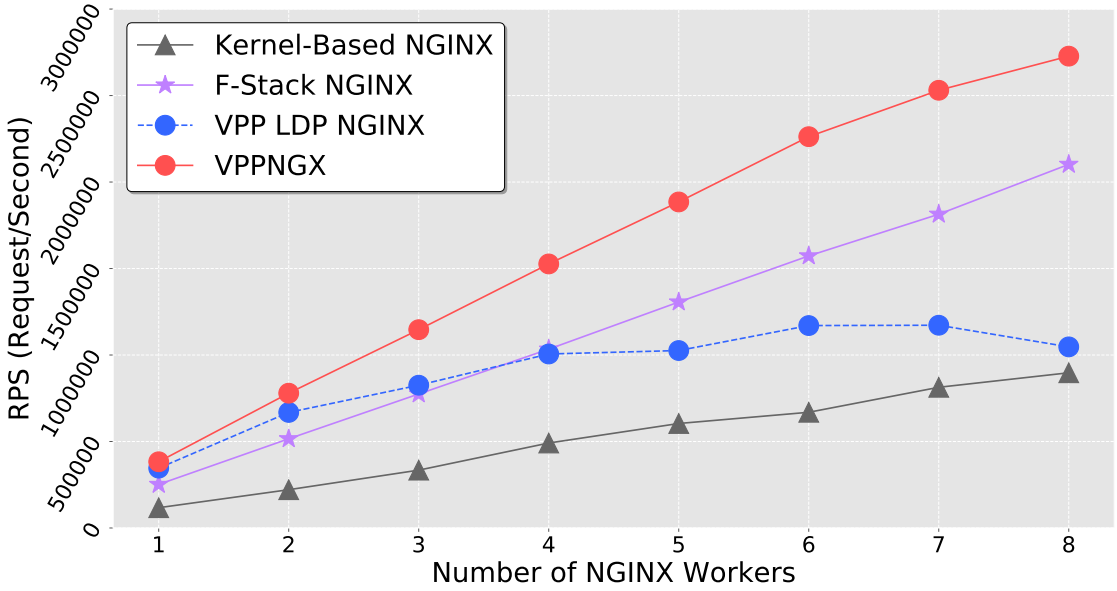

图 6–3 1KB 大文件的 RPS

Figure 6–3 RPS - 1KB Large File

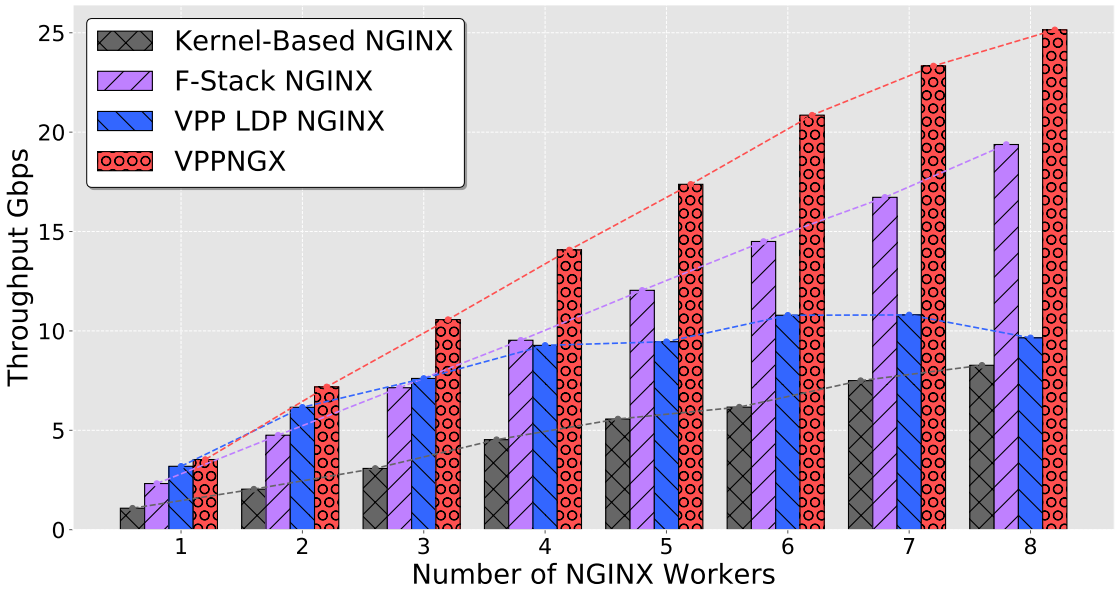

图 6–4 1KB 大文件的吞吐量

Figure 6–4 Throughput - 1KB Large File

存缓存大文件时的 RPS 和吞吐量结果。从这四张图中我们可以看出，虽然在 NGINX Worker 数量较少时 VPP LDP NGINX 的 RPS 和吞吐量要比 F-Stack NGINX 的要高，但是随着 NGINX Worker 数量的不断增长，VPP LDP NGINX 的 RPS 和吞吐量的增速逐渐放缓甚至开始下降，最后大幅度低于 F-Stack NGINX 的 RPS 和吞吐量。这种现象的产生原因是——VPP LDP NGINX 的多个 NGINX Worker 竞争读写 Read/Write Lock 的 *rw_cnt* 变量，还有竞争 Listenser Session Lock。这些竞争使得 VPP LDP NGINX 的请求处理效率极大降低。因此 NGINX Worker 数量较高时，VPP LDP NGINX 的性能要低于 F-Stack NGINX。从这四张图中我们还可以看出，VPPNGX 和 F-Stack NGINX 在 RPS 和吞吐量上均有很好的可扩展性。Kernel-Based NGINX 在可扩展性方面要稍微比 VPPNGX 和 F-Stack NGINX 低一些。并且，Kernel-Based NGINX 的 RPS 和吞吐量要大幅度低于 VPPNGX 和 F-Stack NGINX 的 RPS 和吞吐量。在图 6–3和图 6–4中，我们可以看到当 NGINX Worker 的数量为 7 和 8 时，VPPNGX 的 RPS 和吞吐量的增长速率开始放缓。这是因为在服务器 B 上客户端应用程序在发送请求和接收回复时，处理性能达到了 36 个 CPU 核的瓶颈。瓶颈并非出现在服务器 A 上的 NGINX 端。

从这四张图中我们还可以看出，VPPNGX 的 RPS 和吞吐量要比 F-Stack NGINX 的高，约为 F-Stack NGINX 的 1.5 倍。VPPNGX 的 RPS 和吞吐量约为 Kernel-Based NGINX 的 3 倍。另外，当服务器 A 上只有 1 个 NGINX Worker 的时候，VPPNGX 的 RPS 和吞吐量仍比 VPP LDP NGINX 的高。这种性能提升是因为去除 Session Lock Layer 后，VPPNGX 节省下了处理 Session Lock 的 CPU 开销，并且消除了进行内核 epoll 事件检查的上下文切换开销。

F-Stack 有着和 VPP 不一样的进程模型。我们在前面章节提到，VPP 是网络协议栈和应用程序分离的，各属于不同的进程。而 F-Stack 的网络协议栈和应用程序逻辑是同属于一个进程的。在我们的实验中，F-Stack NGINX 是为 8 个 NGINX Worker 提供了 8 个 CPU 核。而 VPPNGX 和 VPP LDP NGINX 则总共占用了 13 个 CPU 核，并且其中 8 个是给 8 个 NGINX Worker 使用，另外 5 个 CPU 核是给 5 个 VPP Worker 使用。如果我们将 VPP Worker 和 NGINX Worker 放置在同一个 CPU 核上，则 VPPNGX 和 VPP LDP NGINX 的性能会出现下降。这是因为出现了进程切换和缓存未命中。然而 F-Stack 并不像 VPP 一样是个通用框架，它不能同时支持多个应用程序。F-Stack NGINX 只能作为 NGINX 使用。而在服务 VPPNGX 或者 VPP LDP NGINX 的时候，VPP Worker 与此同时可以继续服务其他的网络应用程序。除此之外，F-Stack NGINX 的请求处理延迟要明显比 VPPNGX 和 VPP LDP NGINX 高出很多。请求处理延迟方面的具体实验结果将在下一小节 6.2.2中进行

阐述。

6.2.2 延迟测试结果

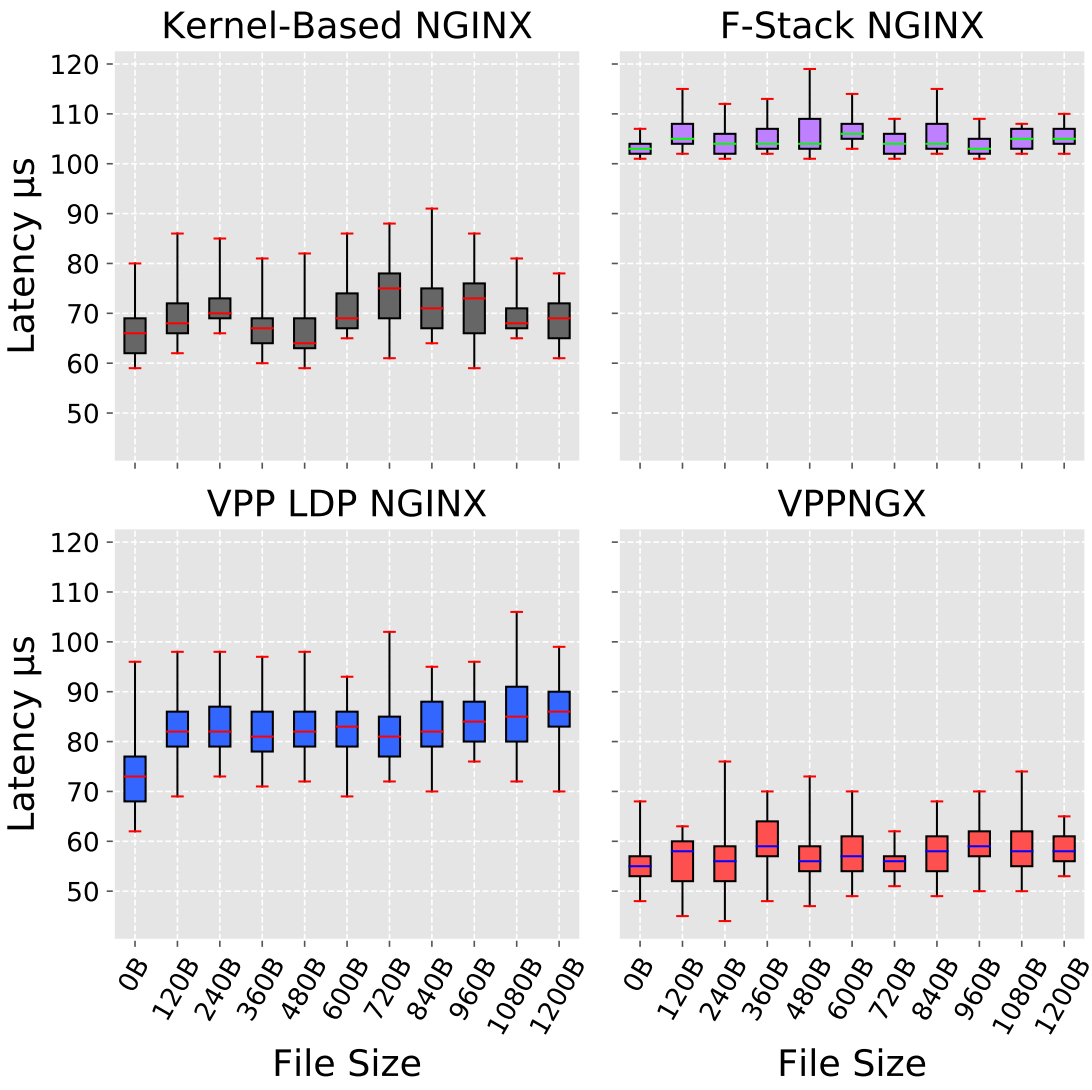

图 6–5 延迟（处理时间/请求）测试结果

Figure 6–5 Latency (Time per Request)

从图 6–5中我们可以看出，客户端程序请求的文件大小并不会明显影响 NGINX 的请求处理延迟。相比于其他三种 NGINX 实现，VPPNGX 有着最低的请求处理延迟。VPPNGX 的请求处理延迟大约在 55 微秒左右。Kernel-Based NGINX 大约有 70 微秒的请求处理延迟。VPP LDP NGINX 的延迟则在 83 微秒左右。而 F-Stack

NIGNX 的请求处理延迟却高达约 105 微秒。四个 NGINX 实现中，F-Stack NIGNX 的请求处理延迟是最高的。VPP LDP NGINX 的延迟要比 Kernel-Based NGINX 的高，然而 VPPNGX 的请求处理延迟要比 Kernel-Based NGINX 低百分之二十左右。

F-Stack NGINX 拥有最高的请求处理延迟是因为，F-Stack NGINX 使用了简单的批处理（Simple Batching）方法去增加 RPS 和吞吐量。而 VPP 使用适应性批处理（Adaptive Batching）方法，在增大吞吐量的同时减少网络包处理延迟。因此，VPPNGX 和 VPP LDP NGINX 要比 F-Stack NGINX 拥有更低的请求处理延迟。另外，因为 Session Lock Layer 中的各种变量竞争和锁竞争被消除了，并且 Session Lock Layer 中一些消耗 CPU 的代码逻辑如查询 Session Lock List、增删 Session Item 表项、获取 Session Lock 等也被消除了，所以 VPPNGX 的请求处理延迟要比 VPP LDP NGINX 的低 32.9%。

6.2.3 NGINX Worker 的 CPU 占用率测试结果

图 6–6展示了 VPPNGX 和 VPP LDP NGINX 的 NGINX Worker 的 CPU 占用率测试结果。从图 6–6中我们可以看出，无论客户端程序的请求并发量等级是多少，VPP LDP NGINX 的所有 8 个 NGINX Worker 的 CPU 占用率始终是百分之百。这是因为 VPP LDP NGINX 的所有 NGINX Worker 都是使用 Busy-Wait Polling epoll（忙等轮询式 epoll 机制）通过轮询方式不断地检查用户态的 Session 的 epoll 事件和内核态的 IPC 消息的 epoll 事件。当并发量等级为 0 时，VPPNGX 的每一个 NGINX Worker 的 CPU 占用率都为 0。这是因为 VPPNGX 的每个 NGINX Worker 都会阻塞在自己的条件变量 *condvar* 上以进入进程等待状态，让出 CPU。当并发量等级逐渐增加时，VPPNGX 的 NGINX Worker 的 CPU 使用率也在逐渐增加，最后在并发量等级为 200 时接近百分之百的占用率。这是因为，在并发量等级较低时，VPPNGX 的 NGINX Worker 在大多数时间是处在进程等待状态的。当 Session 的 epoll 事件或者内核 epoll 事件到来时，VPP Worker 或者 NGINX Master 将会主动地唤醒 NGINX Worker，然后被唤醒的 NGINX Worker 将会处理到来的 epoll 事件。这种方法达到了在网络负载较低时节省 CPU 的目的。

6.2.4 内核 epoll 事件检查的 CPU 开销测试结果

表 6–2分别比较了当客户端应用程序向 NGINX 请求不同大小的内存缓存文件（0B - 1200B）时，调用函数 *epoll_wait()* 和检查用户态 Flag *U*（标记 *U*）去进行内核 epoll 事件检查的 CPU 开销。从表 6–2中我们可以观察到，内存缓存文件的大小并不明显影响调用函数 *epoll_wait()* 或检查用户态标记 *U* 去进行内核 epoll 事件

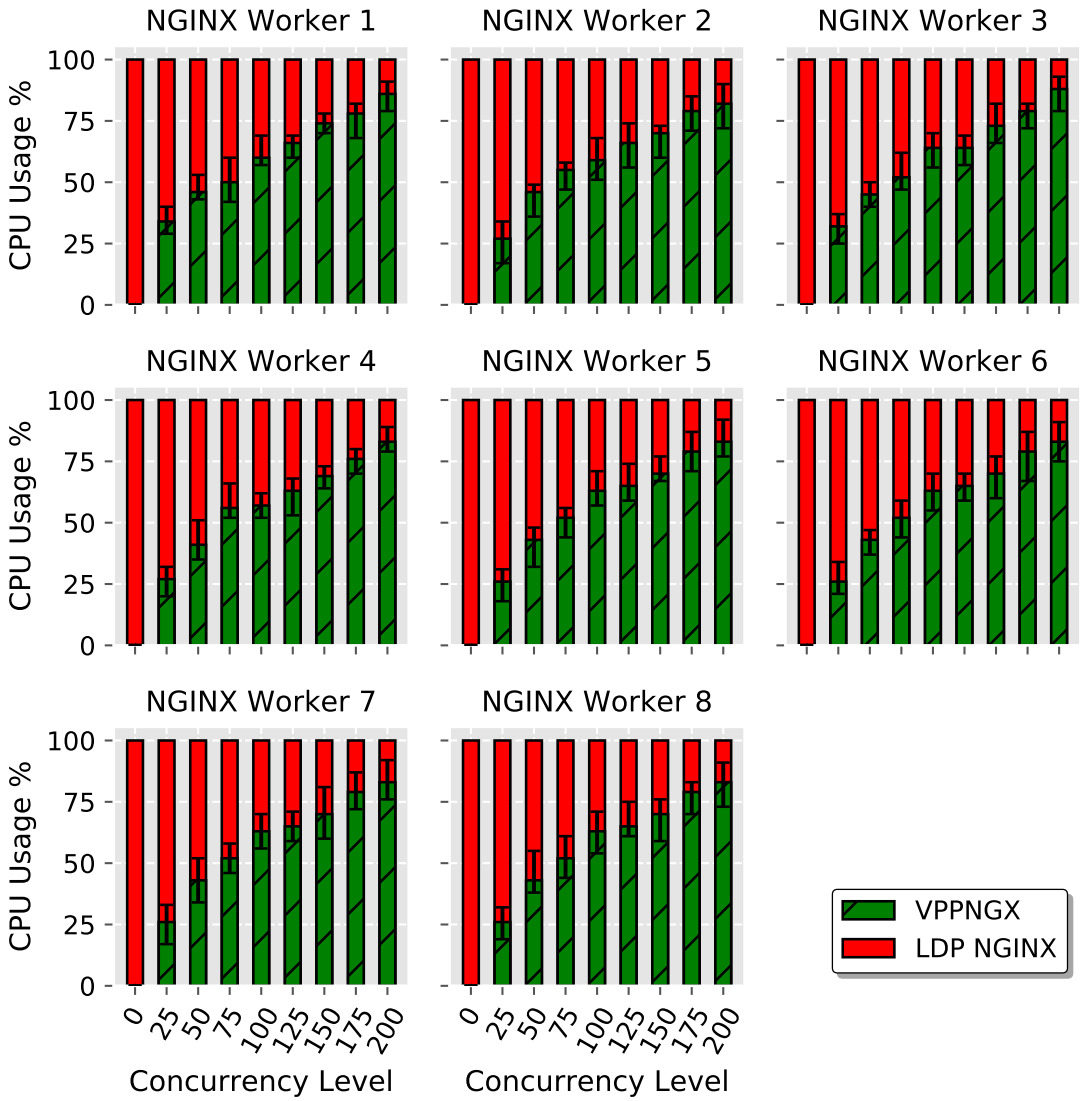

图 6–6 NGINX Worker 的 CPU 占用率测试结果

Figure 6–6 CPU Usage of NGINX Workers

表 6–2 内核 epoll 事件检查的 CPU 开销比较

Table 6–2 Comparison of CPU Overhead of Kernel epoll Event Checking

File Size	VPPNGX: CPU Overhead of Checking Flag *U* for Kernel epoll Event Checking	VPP LDP NGINX: CPU Overhead of *epoll_wait()* for Kernel epoll Event Checking
0B	<0.01%	4.24%
120B	<0.01%	4.86%
240B	<0.01%	5.15%
360B	<0.01%	4.77%
480B	<0.01%	4.72%
600B	<0.01%	4.38%
720B	<0.01%	5.59%
840B	<0.01%	5.69%
960B	<0.01%	4.85%
1080B	<0.01%	5.27%
1200B	<0.01%	4.89%

检查的 CPU 开销。这是因为内核 epoll 事件检查的频率基本是保持不变的。VPP LDP NGINX 调用函数 *epoll_wait()* 进行内核 epoll 事件检查的 CPU 开销在 4% - 5% 的范围左右，因为调用函数 *epoll_wait()* 检查内核 epoll 事件需要陷入内核造成上下文切换开销。对于 VPPNGX 而言，检查用户态的标记 *U* 来判断内核 epoll 事件的 CPU 开销不足 0.01%，这主要是因为 VPPNGX 的 NGINX Worker 并不需要陷入内核去检查 Kernel epoll Wait Queue 来判断内核 epoll 事件是否到来。VPPNGX 的 NGINX Worker 只需要读取用户态的标记 *U* 便可以判断内核 epoll 事件是否到来，从而减少上下文切换带来的 CPU 开销。

尽管引入用户态标记 *U* 来进行内核 epoll 事件检查仅减少了 4% - 5% 的 CPU 开销，但它确实也是对 Busy-Wait Polling epoll 的一种优化。除此之外，引入标记 *U* 还有其他的一个功能。当 VPPNGX 的一个 NGINX Worker 被从进程等待状态唤醒后，它能够通过该用户态标记 *U* 来判断是谁将它唤醒（要么是 VPP Worker 要么是 NGINX Master）。如果标记 *U* 为真，则说明是 NGINX Master 将它唤醒的，这样 NGINX Worker 便可以直接从 Kernel epoll Wait Queue 中取出内核 epoll 事件。否则，如果标记 *U* 为假，则说明是 VPP Worker 将它唤醒的，这样它就直接从 App

Event Queue 中取出 Session 的 epoll 事件。因此，引入用户态标记 U，不仅能够提供这个功能，还能够减少内核 epoll 事件检查引起的上下文切换开销。

6.3 本章小结

本章节对 VPPNGX 的可扩展性、吞吐量、请求处理延迟、NGINX Worker 的 CPU 占用率以及进行内核 epoll 事件检查的 CPU 开销进行了测试与评估。评估实验证明了 VPPNGX 有很好的可扩展性，并且相比于 Kernel-Based NGINX、F-Stack NGINX 和 VPP LDP NGINX 有更高的吞吐量和更低的请求处理延迟。VPPNGX 也能够在网络负载较小时减少 NGINX Worker 的 CPU 占用率，并且能减少因进行内核 epoll 事件检查而导致的上下文切换开销。

第七章　总结与展望

7.1　全文总结

VPPNGX 是一种基于 FD.io VPP 的用于公网请求处理的高性能 NGINX 实现。VPPNGX 使用目前先进的用户态网络协议栈——Vector Packet Processing（VPP）平台提供的网络协议栈，去克服目前内核网络协议栈的缺陷。

除了使用这种新颖的 VPP 架构来优化 NGINX，我们在这篇论文中的主要贡献还包括：首先，我们引入了 VPP Session Index Passthrough（VPP 会话索引透传机制）以安全地移除 Session Lock Layer，使 VPPNGX 获得了良好的可扩展性以及更低的网络请求处理延迟；第二，我们为 VPP Event Queue 引入了 Token-Based Lock-Free Order-Preserving Approach（基于令牌的无锁化保序机制）去保证在移除了 Session Lock Layer 后控制事件消息在入队 VPP Event Queue 时它们的元数据不会发生乱序；最后，我们设计了 User-Space Unified Blocking epoll（用户态联合阻塞式 epoll 机制）去替换原来 NGINX Worker 使用的 Busy-Wait Polling epoll（忙等轮询式 epoll 机制），以在原来能同时接收用户态和内核态 epoll 事件的基础上，实现阻塞机制减少 NGINX Worker 的 CPU 占用率，并减少因进行内核 epoll 事件检查而导致的上下文切换开销。

我们的实验证明了 VPPNGX 有很好的可扩展性，并且相比于 Kernel-Based NGINX、F-Stack NGINX 和 VPP LDP NGINX，VPPNGX 有着更高的网络吞吐量和更低的请求处理延迟。VPPNGX 也能够在网络负载较小时减少 NGINX Worker 的 CPU 占用率，并且在 NGINX Master 不向 NGINX Worker 发送 IPC Message 时，能减少因进行内核 epoll 事件检查而导致的上下文切换开销。我们未来的工作重点是提高 VPPNGX 的可移植性，并降低因一个 NGINX Worker 与多个 VPP Worker 进行通信而引起的高缓存未命中率。

7.2　研究展望

目前的 VPPNGX 尚有一些不足之处，我们将这些不足之处以及针对这些不足之处的未来工作分为以下三点：

- 目前，VPPNGX 的所有 Socket 相关的函数均依赖于 VPP 的网络协议栈。因此，作为一个反向代理服务器应用，VPPNGX 向上游服务器中的后端网络应用建立连接时，也是依赖于 VPP 的网络协议栈，而不是依赖于内核的网

络协议栈。

这样，如果后端网络应用和 VPPNGX 在同一个服务器上，并且这些后端网络应用没有使用 VPP 而是使用 Linux 内核的网络协议栈，那么 VPPNGX 则无法与这些后端网络应用进行通信。因为在同一主机上，VPP 的网络协议栈和内核的网络协议栈之间没有数据通路，也就是说没有网络包的回环路径。

但是如果这些后端网络应用也使用 VPP 作为网络协议栈，那么在同一个主机上，VPPNGX 可以和这些后端网络应用通过 VPP Cut-Through 的方式进行通信。

然而，如果这些后端网络应用与 VPPNGX 不在同一台服务器上，那么无论这些后端网络应用在自己的服务器上使用何种网络协议栈，VPPNGX 均可以与它们进行通信。

在未来的工作中，我们需要让 VPPNGX 在与后端网络应用通信的时候，能根据实际情况和需求，自动选择使用 VPP 的网络协议栈或者内核的网络协议栈。

- 除此之外，为了能让 NGINX Master 唤醒 NGINX Worker，我们引入了 Agent Thread（代理线程）。而引入 Agent Thread 时我们需要对 NGINX 的源代码进行修改。

  在未来，我们希望能够将这种修改移除，以保证 VPPNGX 的可扩展性。如果我们能够将条件变量 *condvar* 从 NGINX Master 无法访问的保护式共享内存中取出，那么不引入 Agent Thread 是可行的。

  然而，将条件变量 *condvar* 从 NGINX Master 无法访问的保护式共享内存中取出可能会引发安全性问题。未来的工作仍需要我们去继续和 FD.io VPP 社区的工作人员进行沟通，以决定我们是否可以只将条件变量 *condvar* 从保护式共享内存中取出。

- 对于目前的 VPPNGX，每一个 VPP Worker 是可以服务所有的 NGINX Worker 的。也因此，每个 NGINX Worker 可以同时被多个 VPP Worker 服务。

  当一个 NGINX Worker 与不同 CPU 核上的多个 VPP Worker 进行通信时，会发生大量的缓存未命中现象。这是因为这个 NGINX Worker 会不断地访问不同 VPP Worker 使用的不同的共享内存。

  在未来的工作中，我们打算让一个 VPP Worker 只去服务一部分的 NGINX Worker。这样，就可以让一个 NGINX Worker 只被一个或者少数 VPP Worker

服务，从而降低了缓存未命中率，并能增加请求处理速率。VPP 应该根据 VPP Worker 的数量、NGINX Worker 的数量以及当前的网络负载情况，来决定选择如何的具体服务策略。

参考文献

[1] VELUSAMY G, LENT R. Smart Load-Balancer for Web Applications//ICSDE ’ 17: Proceedings of the 2017 International Conference on Smart Digital Environment. Rabat, Morocco: Association for Computing Machinery, 2017: 19-26. https://doi.org/10.1145/3128128.3128132. DOI: 10.1145/3128128.3128132.

[2] TAKAHASHI K, AIDA K, TANJO T, et al. A Portable Load Balancer for Kubernetes Cluster//HPC Asia 2018: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region. Chiyoda, Tokyo, Japan: Association for Computing Machinery, 2018: 222-231. https://doi.org/10.1145/3149457.3149473. DOI: 10.1145/3149457.3149473.

[3] BONALD T, JONCKHEERE M, PROUTIÉRE A. Insensitive Load Balancing//SIGMETRICS ’ 04/Performance ’ 04: Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems. New York, NY, USA: Association for Computing Machinery, 2004: 367-377. https://doi.org/10.1145/1005686.1005729. DOI: 10.1145/1005686.1005729.

[4] 杨安. 基于 Web 集群的反向代理负载均衡研究与实践. 信息通信, 2019(11): 13+31.

[5] 黄小玲, 杨桂芹, 邵军花, 等. 软件定义网络中蚁群优化的负载均衡算法. 测控技术, 2020, 39(01): 108-112.

[6] REESE W. Nginx: The High-Performance Web Server and Reverse Proxy. Linux J., 2008, 2008(173).

[7] FRAUNHOLZ D, RETI D, DUQUE ANTON S, et al. Cloxy: A Context-Aware Deception-as-a-Service Reverse Proxy for Web Services//MTD ’ 18: Proceedings of the 5th ACM Workshop on Moving Target Defense. Toronto, Canada: Association for Computing Machinery, 2018: 40-47. https://doi.org/10.1145/3268966.3268973. DOI: 10.1145/3268966.3268973.

[8] ELTON P. Linux as a Proxy Server. Linux J., 1997, 1997(44es): 3-es.

[9] 邓庚盛, 付爱英, 熊永春. Nginx 反向代理技术在移动应用服务架构中的应用. 科技广场, 2017(09): 83-87.

[10] 邢颖, 黄启俊, 易凡, 等. 基于 Reactor 模式的高性能反向代理服务器设计. 自动化技术与应用, 2019, 38(03): 43-48.

[11] NGINX: High Performance Load Balancer, Web Server, and Reverse Proxy. Port of F5. Available at https://www.nginx.com. 2019.

[12] 朱来雪. 基于 Nginx 技术的 Web 系统安全部署方案. 信息与电脑 (理论版), 2019, 31(17): 172-173+176.

[13] HTTP Health Checks in NGINX. NGINX Document Page. Available at https://docs.nginx.com/nginx/admin-guide/load-balancer/http-health-check. 2019.

[14] The Apache HTTP Server Project. Apache. Available at https://httpd.apache.org. 2019.

[15] What Is Nginx? A Basic Look at What It Is and How It Works. Kinsta Knowledge Base. Available at https://kinsta.com/knowledgebase/what-is-nginx. 2019.

[16] The Tengine Web Server. Taobao.org. Available at https://tengine.taobao.org. 2019.

[17] HONDA M, HUICI F, RAICIU C, et al. Rekindling Network Protocol Innovation with User-Level Stacks. SIGCOMM Comput. Commun. Rev., 2014, 44(2): 52-58. https://doi.org/10.1145/2602204.2602212. DOI: 10.1145/2602204.2602212.

[18] CHEN R, SUN G. A Survey of Kernel-Bypass Techniques in Network Stack//CSAI ' 18: Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence. Shenzhen, China: Association for Computing Machinery, 2018: 474-477. https://doi.org/10.1145/3297156.3297242. DOI: 10.1145/3297156.3297242.

[19] KERRISK M. The SO_REUSEPORT Socket Option. LWN.net. Available at https://lwn.net/Articles/542629. 2013.

[20] TAN J, LIANG C, XIE H, et al. VIRTIO-USER: A New Versatile Channel for Kernel-Bypass Networks//KBNets ' 17: Proceedings of the Workshop on Kernel-Bypass Networks. Los Angeles, CA, USA: Association for Computing Machinery, 2017: 13-18. https://doi.org/10.1145/3098583.3098586. DOI: 10.1145/3098583.3098586.

[21] LEI J, SUO K, LU H, et al. Tackling Parallelization Challenges of Kernel Network Stack for Container Overlay Networks//11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19). Renton, WA: USENIX Association, 2019. https://www.usenix.org/conference/hotcloud19/presentation/lei.

[22] The mTCP-NGINX: HTTP Server on User-Level mTCP Stack Accelerated by DPDK. Linaro Connect. Available at https://connect.linaro.org/resources/bud17/bud17-df05. 2017.

[23] 周丹, 陈楚康, 蔡万强, 等. 基于 Linux 内核的用户态网络协议栈的实现. 信息通信, 2019(07): 200-204.

[24] HAN S, MARSHALL S, CHUN B G, et al. MegaPipe: A New Programming Interface for Scalable Network I/O//OSDI '12: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. Hollywood, CA, USA: USENIX Association, 2012: 135-148.

[25] JEONG E Y, WOO S, JAMSHED M, et al. MTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems//NSDI '14: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation. Seattle, WA: USENIX Association, 2014: 489-502.

[26] RIZZO L. Netmap: a novel framework for fast packet I/O//21st USENIX Security Symposium (USENIX Security 12). 2012: 101-112.

[27] PESTEREV A, STRAUSS J, ZELDOVICH N, et al. Improving Network Connection Locality on Multicore Systems//EuroSys '12: Proceedings of the 7th ACM European Conference on Computer Systems. Bern, Switzerland: Association for Computing Machinery, 2012: 337-350. https://doi.org/10.1145/2168836.2168870. DOI: 10.1145/2168836.2168870.

[28] YASUKATA K, HONDA M, SANTRY D, et al. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs//2016 USENIX Annual Technical Conference (USENIX ATC 16). Denver, CO: USENIX Association, 2016: 43-56. https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata.

[29] SOARES L, STUMM M. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls//OSDI ' 10: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. Vancouver, BC, Canada: USENIX Association, 2010: 33-46.

[30] LIN X, CHEN Y, LI X, et al. Scalable Kernel TCP Design and Implementation for Short-Lived Connections. ASPLOS ' 16 2016: 339-352. https://doi.org/10.1145/2872362.2872391. DOI: 10.1145/2872362.2872391.

[31] SUZUMURA T, TATSUBORI M, TRENT S, et al. Highly Scalable Web Applications with Zero-Copy Data Transfer//WWW ' 09: Proceedings of the 18th International Conference on World Wide Web. Madrid, Spain: Association for Computing Machinery, 2009: 921-930. https://doi.org/10.1145/1526709.1526833. DOI: 10.1145/1526709.1526833.

[32] KATO S, AUMILLER J, BRANDT S. Zero-Copy I/O Processing for Low-Latency GPU Computing//ICCPS ' 13: Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems. Philadelphia, Pennsylvania: Association for Computing Machinery, 2013: 170-178. https://doi.org/10.1145/2502524.2502548. DOI: 10.1145/2502524.2502548.

[33] 赵成青, 李宥谋, 刘永斌, 等. LWIP 中零拷贝技术的研究与应用. 计算机技术与发展, 2018, 28(07): 182-186.

[34] CHU H K J. Zero-Copy TCP in Solaris//ATEC ' 96: Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference. San Diego, CA: USENIX Association, 1996: 21.

[35] CORBET J. Zero-Copy Networking. Available at https://lwn.net/Articles/726917. 2017.

[36] THADANI M N, KHALIDI Y A. An Efficient Zero-Copy I/O Framework for UNIX. USA: Sun Microsystems, Inc., 1995.

[37] LI B, CUI T, WANG Z, et al. Socksdirect: Datacenter Sockets Can Be Fast and Compatible//SIGCOMM ' 19: Proceedings of the ACM Special Interest Group on Data Communication. Beijing, China: Association for Computing Machinery, 2019: 90-103. https://doi.org/10.1145/3341302.3342071. DOI: 10.1145/3341302.3342071.

[38] BELAY A, PREKAS G, PRIMORAC M, et al. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. ACM Trans. Comput. Syst., 2016, 34(4). https://doi.org/10.1145/2997641. DOI: 10.1145/2997641.

[39] PETER S, LI J, ZHANG I, et al. Arrakis: The Operating System Is the Control Plane. ACM Trans. Comput. Syst., 2015, 33(4). https://doi.org/10.1145/2812806. DOI: 10.1145/2812806.

[40] DUNKELS A. Design and Implementation of the lwIP TCP/IP Stack. Swedish Institute of Computer Science, 2001, 2(77).

[41] DRAGOJEVIĆ A, NARAYANAN D, HODSON O, et al. FaRM: Fast Remote Memory//NSDI '14: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation. Seattle, WA: USENIX Association, 2014: 401-414.

[42] JOSE J, SUBRAMONI H, LUO M, et al. Memcached Design on High Performance RDMA Capable Interconnects//ICPP '11: 2011 International Conference on Parallel Processing. 2011: 743-752. DOI: 10.1109/ICPP.2011.37.

[43] MITCHELL C, GENG Y, LI J. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store//Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13). San Jose, CA: USENIX, 2013: 103-114. https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell.

[44] ONGARO D, RUMBLE S M, STUTSMAN R, et al. Fast Crash Recovery in RAMCloud//SOSP ' 11: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. Cascais, Portugal: Association for Computing Machinery, 2011: 29-41. https://doi.org/10.1145/2043556.2043560. DOI: 10.1145/2043556.2043560.

[45] XUE J, CHAUDHRY M U, VAMANAN B, et al. Fast Congestion Control in RDMA-Based Datacenter Networks//SIGCOMM ' 18: Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos. Budapest, Hungary: Association for Computing Machinery, 2018: 24-26. https://doi.org/10.1145/3234200.3234216. DOI: 10.1145/3234200.3234216.

[46] GUO C, WU H, DENG Z, et al. RDMA over Commodity Ethernet at Scale//SIGCOMM ' 16: Proceedings of the 2016 ACM SIGCOMM Conference. Florianopolis, Brazil: Association for Computing Machinery, 2016: 202-215. https://doi.org/10.1145/2934872.2934908. DOI: 10.1145/2934872.2934908.

[47] LIU J, POFF D, ABALI B. Evaluating High Performance Communication: A Power Perspective//ICS '09: Proceedings of the 23rd International Conference on Supercomputing. Yorktown Heights, NY, USA: Association for Computing Machinery, 2009: 326-337. https://doi.org/10.1145/1542275.1542322. DOI: 10.1145/1542275.1542322.

[48] KIM B, JUNG H. LW-RDMA: Design and Implementation of a Lightweight RDMA API for InfiniBand-Based Clusters//RACS ' 15: Proceedings of the 2015 Conference on Research in Adaptive and Convergent Systems. Prague, Czech Republic: Association for Computing Machinery, 2015: 395-399. https://doi.org/10.1145/2811411.2811471. DOI: 10.1145/2811411.2811471.

[49] MITTAL R, SHPINER A, PANDA A, et al. Revisiting Network Support for RDMA//SIGCOMM ' 18: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. Budapest, Hungary: Association for Computing Machinery, 2018: 313-326. https://doi.org/10.1145/3230543.3230557. DOI: 10.1145/3230543.3230557.

[50] Data Plane Development Kit. Intel. Available at https://dpdk.org. 2014.

[51] DERI L. Improving Passive Packet Capture: Beyond Device Polling//Proceedings of SANE: vol. 2004. 2004: 85-93.

[52] MARINOS I, WATSON R N, HANDLEY M. Network Stack Specialization for Performance. SIGCOMM ' 14 2014: 175-186. https://doi.org/10.1145/2619239.2626311. DOI: 10.1145/2619239.2626311.

[53] Seastar: High-Performance Server-Side Application Framework. Available at https://seastar.io. 2019.

[54] F-Stack: High-Performance Network Framework Based on DPDK. Available at https://f-stack.org. 2019.

[55] FD.io VPP. FD.io Wiki Page. Available at https://wiki.fd.io/view/VPP. 2019.

[56] GAO P, YU L, WU Y, et al. Low Latency RNN Inference with Cellular Batching//EuroSys ' 18: Proceedings of the Thirteenth EuroSys Conference. Porto, Portugal: Association for Computing Machinery, 2018. https://doi.org/10.1145/3190508.3190541. DOI: 10.1145/3190508.3190541.

[57] BAR-NOY A, GUHA S, KATZ Y, et al. Throughput Maximization of Real-Time Scheduling with Batching//SODA ' 02: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms. San Francisco, California: Society for Industrial, 2002: 742-751.

[58] COHEN N. Every Data Structure Deserves Lock-Free Memory Reclamation. Proc. ACM Program. Lang., 2018, 2(OOPSLA). https://doi.org/10.1145/3276513. DOI: 10.1145/3276513.

[59] YUAN X, WILLIAMS-KING D, YANG J, et al. Making Lock-Free Data Structures Verifiable with Artificial Transactions//PLOS ' 15: Proceedings of the 8th Workshop on Programming Languages and Operating Systems. Monterey, California: Association for Computing Machinery, 2015: 39-45. https://doi.org/10.1145/2818302.2818309. DOI: 10.1145/2818302.2818309.

[60] Project lwIP. Available at https://savannah.nongnu.org/projects/lwip. 2018.

[61] F-Stack: A Full User-Space Network Development Kit. DPDK Summit China 2017. Available at https://www.dpdk.org/wp-content/uploads/sites/35/2018/06/DPDK-China2017-Wang-FStack.pdf. 2017.

[62] F-Stack: A Full User-Space Network Service with DMM. FD.io Wiki Page. Available at https://wiki.fd.io/images/c/cb/F-Stack.pptx. 2018.

[63] VPP Performance. FD.io Document Page. Available at https://fd.io/docs/vpp/master/whatisvpp/performance.html. 2019.

[64] VCL-ldpreload: a LD_PRELOAD library that uses the VPP Communications Library (VCL). FD.io Document Page. Available at https://docs.fd.io/vpp/19.08/vcl_ldpreload_doc.html. 2019.

[65] TCP Host Stack of VPP. FD.io Document Page. Available at https://fd.io/docs/vpp/master/whatisvpp/hoststack.html. 2019.

[66] LIU D, DETERS R. The Reverse C10K Problem for Server-Side Mashups//International Conference on Service-Oriented Computing. 2008: 166-177.

[67] MCCLURG J, HOJJAT H, FOSTER N, et al. Event-Driven Network Programming//PLDI ’16: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. Santa Barbara, CA, USA: Association for Computing Machinery, 2016: 369-385. https://doi.org/10.1145/2908080.2908097. DOI: 10.1145/2908080.2908097.

[68] 李慧霸, 田甜, 彭宇行, 等. 网络程序设计中的并发复杂性. 软件学报, 2011, 22(01): 132-148.

[69] HEILMANN C, 罗小平. 事件驱动型 Web 应用设计. 程序员, 2007(03): 102-104.

[70] Web Server. Wikipedia. Available at https://en.wikipedia.org/wiki/Web_server. 2020.

[71] 李兵. Linux 部署高并发 WEB 服务器性能优化策略. 电脑知识与技术, 2019, 15(31): 19-20.

[72] 徐红梅. Apache 服务器与动态网页技术的整合与研究. 无线互联科技, 2018, 15(21): 149-150.

[73] KEW N. The Apache Modules Book: Application Development with Apache. Prentice Hall Professional, 2007.

[74] 别体伟, 华蓓. 用户空间协议栈的并行化与性能优化. 电子技术, 2016, 45(08): 50-56.

[75] VICENTE E, MATIAS R, BORGES L, et al. Evaluation of Compound System Calls in the Linux Kernel. SIGOPS Oper. Syst. Rev., 2012, 46(1): 53-63. https://doi.org/10.1145/2146382.2146394. DOI: 10.1145/2146382.2146394.

[76] 徐周波, 陈帅, 常亮, 等. 利用符号 OBDD-LIST 设计批处理包过滤防火墙. 小型微型计算机系统, 2017, 38(05): 1013-1016.

[77] MAQUELIN O, GAO G R, HUM H H J, et al. Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling//ISCA ’96: Proceedings of the 23rd Annual International Symposium on Computer Architecture. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996: 179-188. https://doi.org/10.1145/232973.232992. DOI: 10.1145/232973.232992.

[78] What is Vector Packet Processing? FD.io Wiki Page. Available at https://fdio-vpp.readthedocs.io/en/latest/overview/whatisvpp/what-is-vector-packet-processing.html. 2019.

[79] LIU D, LUO X, REN F. MTSL: Making mTCP Stack Transparent to Network Applications//ISCC ' 18: 2018 IEEE Symposium on Computers and Communications (ISCC). 2018: 192-197. DOI: 10.1109/ISCC.2018.8538626.

[80] The epoll Event Distribution Interface. Linux Man Page. Available at https://linux.die.net/man/4/epoll. 2019.

[81] CHOI S, LONG X, SHAHBAZ M, et al. The Case for a Flexible Low-Level Backend for Software Data Planes//APNet ' 17: Proceedings of the First Asia-Pacific Workshop on Networking. Hong Kong, China: Association for Computing Machinery, 2017: 71-77. https://doi.org/10.1145/3106989.3107000. DOI: 10.1145/3106989.3107000.

[82] KAFFES K, CHONG T, HUMPHRIES J T, et al. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency//16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). Boston, MA: USENIX Association, 2019: 345-360. https://www.usenix.org/conference/nsdi19/presentation/kaffes.

[83] MARTY M, de KRUIJF M, ADRIAENS J, et al. Snap: A Microkernel Approach to Host Networking//SOSP ' 19: Proceedings of the 27th ACM Symposium on Operating Systems Principles. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019: 399-413. https://doi.org/10.1145/3341301.3359657. DOI: 10.1145/3341301.3359657.

[84] CHO C, CHUN B, SEO J. Adaptive Batching Scheme for Real-Time Data Transfers in IoT Environment//ICCBDC ' 17: Proceedings of the 2017 International Conference on Cloud and Big Data Computing. London, United Kingdom: Association for Computing Machinery, 2017: 55-59. https://doi.org/10.1145/3141128.3141145. DOI: 10.1145/3141128.3141145.

[85] De SÁ A S, SILVA FREITAS A E, de ARAÚJO MACUNDEFINEDDO R J. Adaptive Request Batching for Byzantine Replication. SIGOPS Oper. Syst. Rev., 2013, 47(1): 35-42. https://doi.org/10.1145/2433140.2433149. DOI: 10.1145/2433140.2433149.

[86] How to Push Extreme Limits of Performance and Scale with Vector Packet Processing Technology. Cisco Live 2017. Available at https://www.ciscolive.com/c/dam/r/ciscolive/us/docs/2017/pdf/DEVNET-1221.pdf. 2017.

[87] The wrk - a HTTP benchmarking tool. GitHub.com. Available at https://github.com/wg/wrk. 2019.

致　谢

本篇论文是在我的导师李健老师的指导下完成的。非常感谢李健老师对我论文的悉心指导。即使在年底工作繁忙的时候，或者在春节应该休息的时候，李健老师都会抽出时间帮我修改论文。在此，谨向李健老师致以我最衷心的感谢！

同时还要感谢英特尔工程师虞平老师对我工程项目上的指导。没有虞平老师，我就没法参与 FD.io 社区的 VPP 项目，并与国际工程师进行密切的交流与合作。是虞平老师给我提供了宝贵的锻炼自身能力的机会。

此外，本篇论文的顺利完成，也离不开 SDIC 实验室的胡小康、钱建民等学长为我提出的宝贵建议。在生活和学习上我还得到了同班以及同实验室同学的关心和帮助，感谢你们！感谢和我一起度过硕士研究生生涯的韩易忱、孙国傲、郭鹤林还有史蔚威。感谢班长张锐同学对班级的辛勤付出。同时，还要感谢其他各位班委以及作为学习委员的我自己。最后，我还要感谢培养我长大并且一直为我的学习生活提供源源不断的信念的父母，谢谢你们！

攻读硕士学位期间参与的项目

[1] 参与 SDIC 实验室与英特尔合作的 FD.io 社区 VPP 项目 (2018 年 10 月–2019 年 10 月)

攻读硕士学位期间申请的专利

[1] 第二发明人，“一种网络请求处理系统和方法”，专利申请号 202010059255.0