绝对干货！初学者也能看懂的DPDK解析

DPDK社区 DPDK与SPDK开源社区 1周前

01 网络IO的处境和趋势

从我们用户的使用就可以感受到网速一直在提升，而网络技术的发展也从1GE/10GE/25GE/40GE/100GE的演变，从中可以得出单机的网络IO能力必须跟上时代的发展。

1. 传统的电信领域

IP层及以下，例如路由器、交换机、防火墙、基站等设备都是采用硬件解决方案。基于专用网络处理器（NP），有基于FPGA，更有基于ASIC的。但是基于硬件的劣势非常明显，发生Bug不易修复，不易调试维护，并且网络技术一直在发展，例如2G/3G/4G/5G等移动技术的革新，这些属于业务的逻辑基于硬件实现太痛苦，不能快速迭代。传统领域面临的挑战是急需一套软件架构的高性能网络IO开发框架。

2. 云的发展

私有云的出现通过网络功能虚拟化（NFV）共享硬件成为趋势，NFV的定义是通过标准的服务器、标准交换机实现各种传统的或新的网络功能。急需一套基于常用系统和标准服务器的高性能网络IO开发框架。

3. 单机性能的飙升

网卡从1G到100G的发展，CPU从单核到多核到多CPU的发展，服务器的单机能力通过横行扩展达到新的高点。但是软件开发却无法跟上节奏，单机处理能力没能和硬件门当户对，如何开发出与时并进高吞吐量的服务，单机百万千万并发能力。即使有业务对QPS要求不高，主要是CPU密集型，但是现在大数据分析、人工智能等应用都需要在分布式服务器之间传输大量数据完成作业。这点应该是我们互联网后台开发最应关注，也最关联的。

02 Linux + x86网络IO瓶颈

在数年前曾经写过《网卡工作原理及高并发下的调优》一文，描述了Linux的收发报文流程。根据经验，在C1（8核）上跑应用每1W包处理需要消耗1%软中断CPU，这意味着单机的上限是100万PPS（Packet Per Second）。从TGW（Netfilter版）的性能100万PPS，AliLVS优化了也只到150万PPS，并且他们使用的服务器的配置还是比较好的。假设，我们要跑满10GE网卡，每个包64字节，这就需要2000万PPS（注：以太网万兆网卡速度上限是1488万PPS，因为最小帧大小为84B《Bandwidth, Packets Per Second, and Other Network Performance Metrics》），100G是2亿PPS，即每个包的处理耗时不能超过50纳秒。而一次Cache Miss，不管是TLB、数据Cache、指令Cache发生Miss，回内存读取大约65纳秒，NUMA体系下跨Node通讯大约40纳秒。所以，即使不加上业务逻辑，即使纯收发包都如此艰难。我们要控制Cache的命中率，我们要了解计算机体系结构，不能发生跨Node通讯。

从这些数据，我希望可以直接感受一下这里的挑战有多大，理想和现实，我们需要从中平衡。问题都有这些

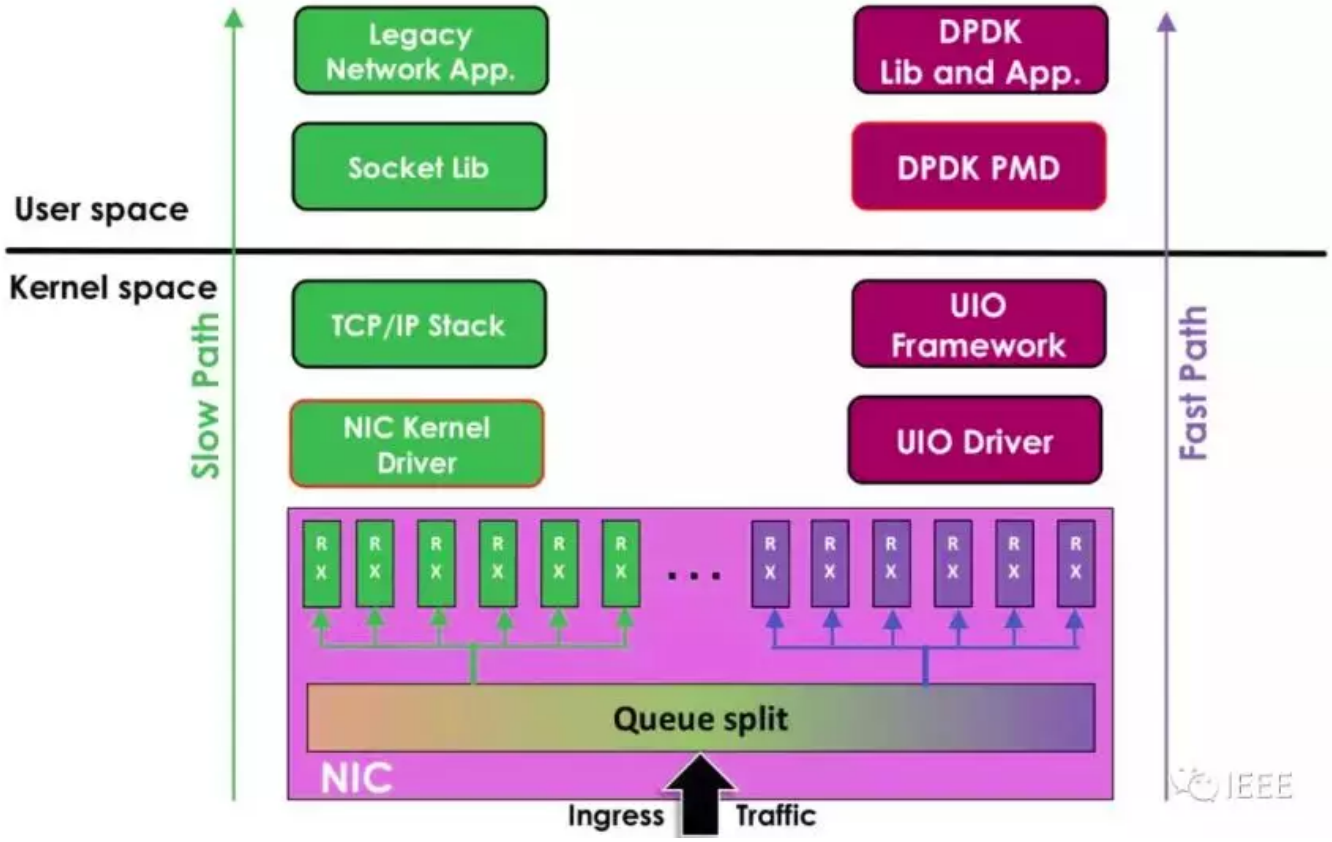
1.传统的收发报文方式都必须采用硬中断来做通讯，每次硬中断大约消耗100微秒，这还不算因为终止上下文所带来的Cache Miss。

2.数据必须从内核态用户态之间切换拷贝带来大量CPU消耗，全局锁竞争。

3.收发包都有系统调用的开销。
4.内核工作在多核上，为可全局一致，即使采用Lock Free，也避免不了锁总线、内存屏障带来的性能损耗。
5.从网卡到业务进程，经过的路径太长，有些其实未必要的，例如netfilter框架，这些都带来一定的消耗，而且容易Cache Miss。

03 DPDK的基本原理

从前面的分析可以得知IO实现的方式、内核的瓶颈，以及数据流过内核存在不可控因素，这些都是在内核中实现，内核是导致瓶颈的原因所在，要解决问题需要绕过内核。所以主流解决方案都是旁路网卡IO，绕过内核直接在用户态收发包来解决内核的瓶颈。
Linux社区也提供了旁路机制Netmap，官方数据10G网卡1400万PPS，但是Netmap没广泛使用。其原因有几个：
1.Netmap需要驱动的支持，即需要网卡厂商认可这个方案。
2.Netmap仍然依赖中断通知机制，没完全解决瓶颈。
3.Netmap更像是几个系统调用，实现用户态直接收发包，功能太过原始，没形成依赖的网络开发框架，社区不完善。
那么，我们来看看发展了十几年的DPDK，从Intel主导开发，到华为、思科、AWS等大厂商的加入，核心玩家都在该圈子里，拥有完善的社区，生态形成闭环。早期，主要是传统电信领域3层以下的应用，如华为、中国电信、中国移动都是其早期使用者，交换机、路由器、网关是主要应用场景。但是，随着上层业务的需求以及DPDK的完善，在更高的应用也在逐步出现。
DPDK旁路原理：

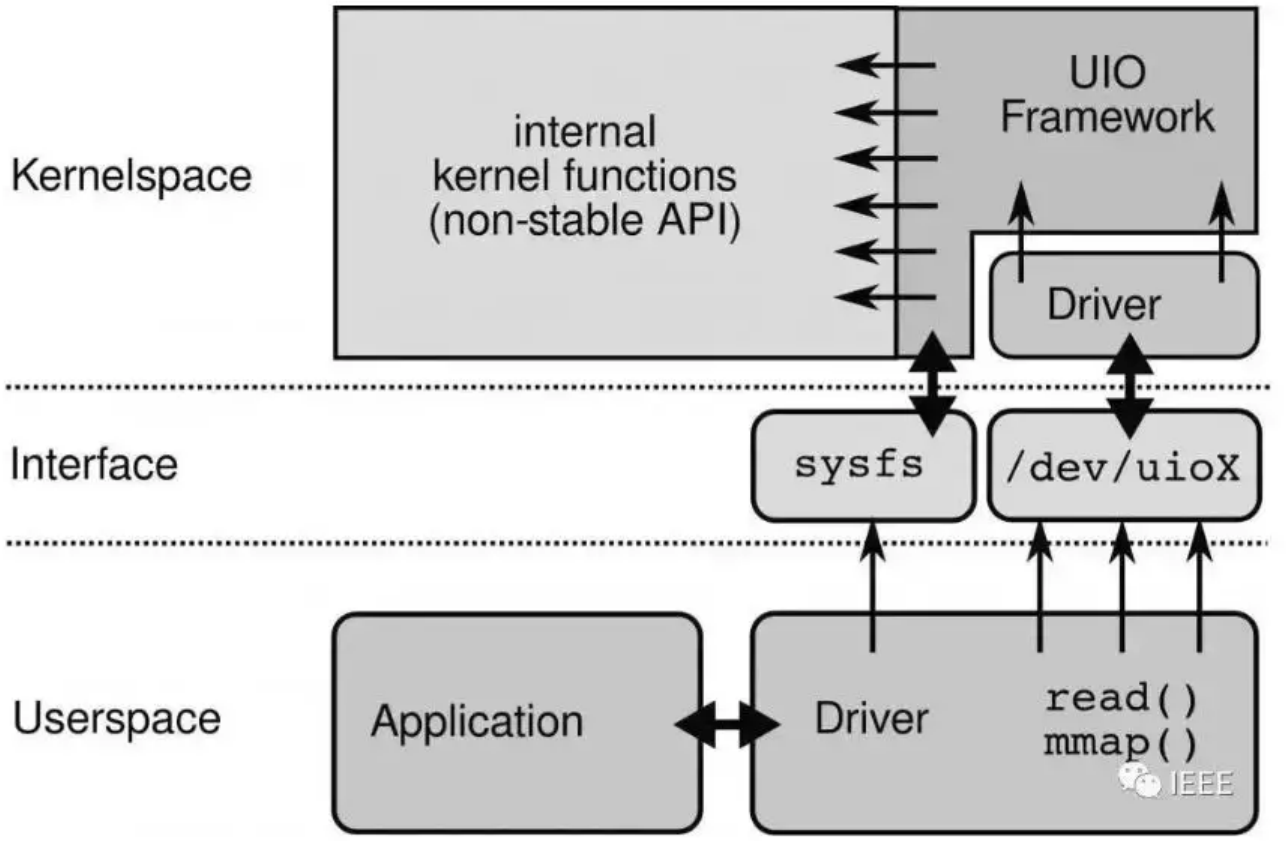


图片引自Jingjing Wu的文档《Flow Bifurcation on Intel® Ethernet Controller X710/XL710》
左边是原来的方式数据从 网卡 -> 驱动 -> 协议栈 -> Socket接口 -> 业务

右边是DPDK的方式，基于UIO（Userspace I/O）旁路数据。数据从 网卡 -> DPDK轮询模式-> DPDK基础库 -> 业务
用户态的好处是易用开发和维护，灵活性好。并且Crash也不影响内核运行，鲁棒性强。
DPDK支持的CPU体系架构：x86、ARM、PowerPC（PPC）
DPDK支持的网卡列表：https://core.dpdk.org/supported/，我们主流使用Intel 82599（光口）、Intel x540（电口）

04 DPDK的基石UIO

为了让驱动运行在用户态，Linux提供UIO机制。使用UIO可以通过read感知中断，通过mmap实现和网卡的通讯。
UIO原理：



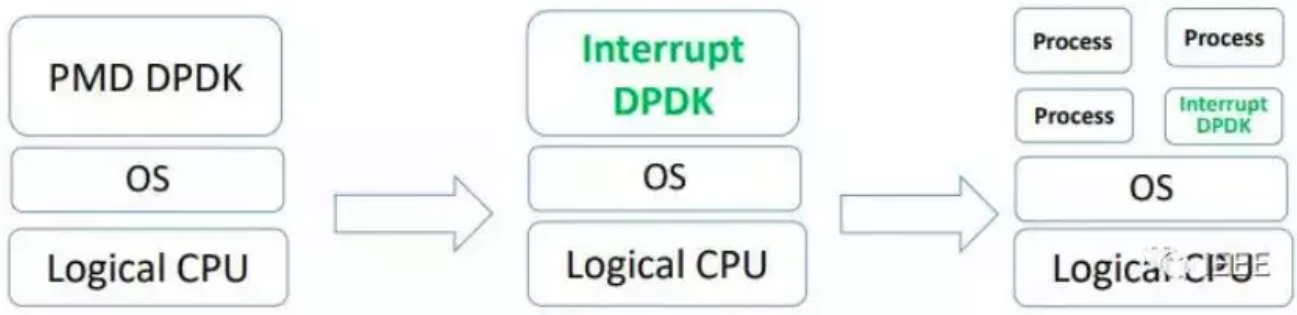
要开发用户态驱动有几个步骤：
1.开发运行在内核的UIO模块，因为硬中断只能在内核处理
2.通过/dev/uioX读取中断
3.通过mmap和外设共享内存

05 DPDK核心优化：PMD

DPDK的UIO驱动屏蔽了硬件发出中断，然后在用户态采用主动轮询的方式，这种模式被称为PMD（Poll Mode Driver）。
UIO旁路了内核，主动轮询去掉硬中断，DPDK从而可以在用户态做收发包处理。带来Zero Copy、无系统调用的好处，同步处理减少上下文切换带来的Cache Miss。
运行在PMD的Core会处于用户态CPU100%的状态

```
top - 23:49:15 up 66 days, 23:17,  3 users,  load average: 16.08, 16.04, 16.05
Tasks: 311 total,   2 running, 309 sleeping,   0 stopped,   0 zombie
%Cpu0  : 18.8 us, 79.2 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  2.0 si,  0.0 st
%Cpu1  : 99.0 us,  1.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu6  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu7  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu8  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu9  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu10 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu11 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu12 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu13 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu14 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu15 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 65689704 total,   291504 free, 64180408 used,  1217792 buff/cache
KiB Swap: 32964604 total, 32191076 free,   773528 used.   301476 avail Mem
```

网络空闲时CPU长期空转，会带来能耗问题。所以，DPDK推出Interrupt DPDK模式。
Interrupt DPDK：



图片引自David Su/Yunhong Jiang/Wei Wang的文档《Towards Low Latency Interrupt Mode DPDK》
它的原理和NAPI很像，就是没包可处理时进入睡眠，改为中断通知。并且可以和其他进程共享同个CPU Core，但是DPDK进程会有更高调度优先级。

06 DPDK的高性能代码实现

1. 采用HugePage减少TLB Miss
默认下Linux采用4KB为一页，页越小内存越大，页表的开销越大，页表的内存占用也越大。CPU有TLB（Translation Lookaside Buffer）成本高所以一般就只能存放几百到上千个页表项。如果进程要使用64G内存，则64G/4KB=16000000（一千六百万）页，每页在页表项中占用16000000 * 4B=62MB。如果用HugePage采用2MB作为一页，只需64G/2MB=2000，数量不在同个级别。
而DPDK采用HugePage，在x86-64下支持2MB、1GB的页大小，几何级的降低了页表项的大小，从而减少TLB-Miss。并提供了内存池（Mempool）、MBuf、无锁环（Ring）、Bitmap等基础库。根据我们的实践，在数据平面（Data Plane）频繁的内存分配释放，必须使用内存池，不能直接使用rte_malloc，DPDK的内存分配实现非常简陋，不如ptmalloc。

2. SNA（Shared-nothing Architecture）

软件架构去中心化，尽量避免全局共享，带来全局竞争，失去横向扩展的能力。NUMA体系下不跨Node远程使用内存。

3. SIMD（Single Instruction Multiple Data）

从最早的mmx/sse到最新的avx2，SIMD的能力一直在增强。DPDK采用批量同时处理多个包，再用向量编程，一个周期内对所有包进行处理。比如，memcpy就使用SIMD来提高速度。

SIMD在游戏后台比较常见，但是其他业务如果有类似批量处理的场景，要提高性能，也可看看能否满足。

4. 不使用慢速API

这里需要重新定义一下慢速API，比如说gettimeofday，虽然在64位下通过vDSO已经不需要陷入内核态，只是一个纯内存访问，每秒也能达到几千万的级别。但是，不要忘记了我们在10GE下，每秒的处理能力就要达到几千万。所以即使是gettimeofday也属于慢速API。DPDK提供Cycles接口，例如rte_get_tsc_cycles接口，基于HPET或TSC实现。

在x86-64下使用RDTSC指令，直接从寄存器读取，需要输入2个参数，比较常见的实现：

```
static inline uint64_t
rte_rdtsc(void)
{
	uint32_t lo, hi;

	__asm__ __volatile__ (
			"rdtsc" : "=a"(lo), "=d"(hi)
			);

	return ((unsigned long long)lo) | (((unsigned long long)hi) << 32);
}
```

这么写逻辑没错，但是还不够极致，还涉及到2次位运算才能得到结果，我们看看DPDK是怎么实现：

```
static inline uint64_t
rte_rdtsc(void)
{
    union {
        uint64_t tsc_64;
        struct {
            uint32_t lo_32;
            uint32_t hi_32;
        };
    } tsc;

    asm volatile("rdtsc" :
             "=a" (tsc.lo_32),
             "=d" (tsc.hi_32));
    return tsc.tsc_64;
}
```

巧妙的利用C的union共享内存，直接赋值，减少了不必要的运算。但是使用tsc有些问题需要面对和解决

1) CPU亲和性，解决多核跳动不精确的问题

2) 内存屏障，解决乱序执行不精确的问题

3) 禁止降频和禁止Intel Turbo Boost，固定CPU频率，解决频率变化带来的失准问题

5. 编译执行优化

1) 分支预测

现代CPU通过pipeline、superscalar提高并行处理能力，为了进一步发挥并行能力会做分支预测，提升CPU的并行能力。遇到分支时判断可能进入哪个分支，提前处理该分支的代码，预先做指令读取编码读取寄存器等，预测失败则预处理全部丢弃。我们开发业务有时候会非常清楚这个分支是true还是false，那就可以通过人工干预生成更紧凑的代码提示CPU分支预测成功率。

```
#pragma once

#if !__GLIBC_PREREQ(2, 3)
#    if !define __builtin_expect
#        define __builtin_expect(x, expected_value) (x)
#    endif
#endif

#if !defined(likely)
#define likely(x) (__builtin_expect(!!(x), 1))
#endif

#if !defined(unlikely)
#define unlikely(x) (__builtin_expect(!!(x), 0))
#endif
```

2) CPU Cache预取

Cache Miss的代价非常高，回内存读需要65纳秒，可以将即将访问的数据主动推送的CPU Cache进行优化。比较典型的场景是链表的遍历，链表的下一节点都是随机内存地址，所以CPU肯定是无法自动预加载的。但是我们在处理本节点时，可以通过CPU指令将下一个节点推送到Cache里。

API文档：https://doc.dpdk.org/api/rte__prefetch_8h.html

```
static inline void rte_prefetch0(const volatile void *p)
{
    asm volatile ("prefetcht0 %[p]" : : [p] "m" (*(const volatile char *)p));
}
```

```
#if !defined(prefetch)
#define prefetch(x) __builtin_prefetch(x)
#endif
```

3) 内存对齐

内存对齐有2个好处：

l 避免结构体成员跨Cache Line，需2次读取才能合并到寄存器中，降低性能。结构体成员需从大到小排序和以及强制对齐。参考《Data alignment: Straighten up and fly right》

```
#define __rte_packed __attribute__((__packed__))
```

l 多线程场景下写产生False sharing，造成Cache Miss，结构体按Cache Line对齐

```
#ifndef CACHE_LINE_SIZE
#define CACHE_LINE_SIZE 64
#endif

#ifndef aligined
#define aligined(a) __attribute__((__aligned__(a)))
#endif
```

4) 常量优化

常量相关的运算的编译阶段完成。比如C++11引入了constexp，比如可以使用GCC的__builtin_constant_p来判断值是否常量，然后对常量进行编译时得出结果。举例网络序主机序转换

```
#define rte_bswap32(x) ((uint32_t)(__builtin_constant_p(x) ?        \
                    rte_constant_bswap32(x) :        \
                    rte_arch_bswap32(x)))
```

其中rte_constant_bswap32的实现

```
#define RTE_STATIC_BSWAP32(v) \
    ((((uint32_t)(v) & UINT32_C(0x000000ff)) << 24) | \
     (((uint32_t)(v) & UINT32_C(0x0000ff00)) <<  8) | \
     (((uint32_t)(v) & UINT32_C(0x00ff0000)) >>  8) | \
     (((uint32_t)(v) & UINT32_C(0xff000000)) >> 24))
```

5）使用CPU指令

现代CPU提供很多指令可直接完成常见功能，比如大小端转换，x86有bswap指令直接支持了。

```
static inline uint64_t rte_arch_bswap64(uint64_t _x)
{
    register uint64_t x = _x;
    asm volatile ("bswap %[x]"
              : [x] "+r" (x)
              );
    return x;
}
```

这个实现，也是GLIBC的实现，先常量优化、CPU指令优化、最后才用裸代码实现。毕竟都是顶端程序员，对语言、编译器，对实现的追求不一样，所以造轮子前一定要先了解好轮子。

Google开源的cpu_features可以获取当前CPU支持什么特性，从而对特定CPU进行执行优化。高性能编程永无止境，对硬件、内核、编译器、开发语言的理解要深入且与时俱进。

07 DPDK生态

对我们互联网后台开发来说DPDK框架本身提供的能力还是比较裸的，比如要使用DPDK就必须实现ARP、IP层这些基础功能，有一定上手难度。如果要更高层的业务使用，还需要用户态的传输协议支持。不建议直接使用DPDK。

目前生态完善，社区强大（一线大厂支持）的应用层开发项目是FD.io（The Fast Data Project），有思科开源支持的VPP，比较完善的协议支持，ARP、VLAN、Multipath、IPv4/v6、MPLS等。用户态传输协议UDP/TCP有TLDK。从项目定位到社区支持力度算比较靠谱的框架。

腾讯云开源的F-Stack也值得关注一下，开发更简单，直接提供了POSIX接口。

Seastar也很强大和灵活，内核态和DPDK都随意切换，也有自己的传输协议Seastar Native TCP/IP Stack支持，但是目前还未看到有大型项目在使用Seastar，可能需要填的坑比较多。
我们GBN Gateway项目需要支持L3/IP层接入做Wan网关，单机20GE，基于DPDK开发。

— END —

文章转载自IEEE，长按二维码关注该公众号

点击“阅读原文”

阅读原文