“架构师技术联盟”微信公众号专注技术架构和行业解决方案，构建专业交流平台，分享一线技术实践，洞察行业前沿趋势，内容覆盖云计算、大数据、超融合、软件定义网络、数据保护和解决方案等。

《详解 DPDK 和 SPDK 技术知识点》

公众号作者把历史原创文章进行了总结、归类和细化，梳理成电子书(含免费)，方便相关从业者参阅。很多读者读完反馈受益良多，作者也倍感欣慰。核心电子书收少许整理费，一则可把书传递给真正需要的读者手中，二则算是对作者的略微肯定，重点书目如下所示(持续更新…)：

《数据备份和副本管理技术全面解析》
《容器技术架构、网络和生态详解》
《闪存技术、产品和发展趋势全面解析》
《虚拟化技术最详细解析》
《传统企业存储知识完全解析》
《IO 知识和系统性能深度调优全解》
《业界主流数据中心存储双活完全解析》
《Ceph 技术架构、生态和特性详细对比分析》
《数据中心大二层交换技术详解》
《VMware 云数据中心(私有云)解决方案详解》
《大数据时代数据重删技术详解》
《高性能计算 HPC 技术、方案和行业全面解析》
《详解 DPDK 和 SPDK 技术知识点》
……

说明：免费电子书下载地址(实时更新防止链接失效)->请关注“架构师技术联盟”微信号或在“架构师电子书店”首页，按照提示语或说明获取下载地址。

“架构师技术联盟”微信公众号

架构师电子书店

目录

1、DPDK 背景介绍

随着芯片技术与高速网络接口技术的一日千里式发展，报文吞吐需要处理 10Gbps 端口处理能力，世面上大量的 25G、40G 甚至 100G 高速端口已经出现，主流处理器的主频仍停留在 3GHz 左右。

IO 超越 CPU 的运行速率，是横在行业面前的技术挑战。2009 年开始，以 Venky Venkastraen，Walter Gilmore，Mike Lynch 为核心的 Intel 团队，开始了可行性研究，希望借助软件技术来实现，很快他们取得了一定的技术突破，设计了运行在 Linux 用户态的网卡程序架构。随后，Intel 与 6wind 进行了更进一步的合作,共同交付了早期的 DPDK 软件开发包。2011 年开始，6wind、Windriver、Tieto、Radisys 先后宣布了对 Intel DPDK 的商业服务支持。Intel 起初只是将 DPDK 以源代码方式分享给少量客户，作为评估 IA 平台和硬件性能的软件服务模块，随着时间推移与行业的大幅度接受，2013 年 Intel 将 DPDK 这一软件以 BSD 开源方式，分享在 Intel 的网站上，供开发者免费下载。

2013 年 4 月，6wind 联合其他开发者成立了 www.dpdk.org 的开源社区，DPDK 走上了开源的大道。DPDK 在代码开源后，任何开发者被允许通过 www.dpdk.org 提交代码，随着开发者社区进一步扩大，Intel 持续加大了在开源社区的投入，同时在 NFV 浪潮下，越来越多的公司和个人开发者加入了这一社区，比如 Brocade，Cisco，RedHat，VMWARE，IBM,他们不再只是 DPDK 的消费者，角色向生产者转变，开始提供代码，对 DPDK 的代码进行优化，整理。起初 DPDK 完全专注于 Intel 的服务器平台技术，专注于利用处理器与芯片组高级特性，支持 Intel 的网卡产品线系列。

DPDK 2.1 版本在 2015 年 8 月发布，几乎所有行业主流的网卡设备商都已经加入了 DPDK 社区，提供源代码级别支持。例如 Emulex 收购的 Broadcom 网卡，Mellanox，Chelsio 以及 Cisco 等等。另外除了支持通用网卡之外，能否将 DPDK 应用在特别的加速芯片上，是一个有趣的话题，有很多工作在

进行中，Intel 最新提交了用于 Crypto 设备的接口设计接口,可以利用类似 Intel 的 QuickAssit 的硬件加速单元，实现一个针对数据包加解密与压缩处理的软件接口。

在多架构支持方面，DPDK 社区也取得了很大的进展，IBM 中国研究院的祝超博士，启动了将 DPDK 移植到 Power 体系架构的工作， Freescale 的中国开发者也参与修改，Tilera 与 Ezchip 的工程师也花了不少精力将 DPDK 运行在 Tile 架构下。很快 DPDK 从单一的基于 Intel 平台为基石的软件，逐步演变成一个相对完整的生态系统，覆盖了多个处理器，多个以太网和硬件加速技术。

在 Linux 社区融合方面，DPDK 也开始和一些主流的 Linux 社区合作，并得到了越来越多的响应。作为 Linux 社区最主要的贡献者之一 RedHat，尝试了在 Fedora Linux 集成了 DPDK; 接着 Redhat Enterprise Linux 在安装库里也加入 DPDK 支持，用户可以自动下载安装 DPDK 扩展库。Redhat 工程师还尝试了将 DPDK 与 Container 集成测试，公开发布了运行结果。传统虚拟化的领导者 VMWARE 的工程师也加入了 DPDK 社区，负责 VMXNET3-PMD 模块的维护。Canonical 在 Ubuntu 15 中加入了 DPDK 的支持。

2、DPDK 架构和关键技术

2.1 概念和术语

随着互联网的高速发展、云产业的快速突起，基础架构网络逐渐偏向基于通用计算平台或模块化计算平台的架构融合,以支持多样化和大数据下的网络功能，传统的 PC 机器在分布式计算平台上的优势更为明显。在这些针对海量数据处理或海量用户的服务场景，高性能编程显得尤为重要。DPDK 应运而生。

DPDK 英文全称为 Data Plane Development Kit，DPDK 是一套源码编程库，或者可以说是一个开源的数据平面开发工具集,可以为 Intel Architecture(IA)

处理器架构下用户空间高效的数据包处理提供库函数和驱动的支持，它不同于Linux 系统以通用性设计为目的，而是专注于网络应用中数据包的高性能处理。

DPDK 通过环境抽象层旁路内核协议栈、轮询模式的报文无中断收发、优化内存/缓冲区/队列管理、基于网卡多队列和流识别的负载均衡等多项技术，实现了在 x86 处理器架构下的高性能报文转发能力，目前已经验证可以运行在大多数 Linux 操作系统上，包括 FreeBSD 9.2、Fedora release18、Ubuntu 12.04 LTS、RedHat Enterprise Linux 6.3 和 Suse Enterprise Linux 11 SP2 等。DPDK 使用了 BSD License,极大的方便了企业在其基础上来实现自己的协议栈或者应用。用户可以在 Linux 用户态空间开发各类高速转发应用,也适合与各类商业化的数据平面加速解决方案进行集成。

英特尔在 2010 年启动了对 DPDK 技术的开源化进程，于当年 9 月通过 BSD 开源许可协议正式发布源代码软件包，并于 2014 年 4 月在 www.dpdk.org 上正式成立了独立的开源社区平台，为开发者提供支持。开源社区的参与者们大幅推进了 DPDK 的技术创新和快速演进,而今它已发展成为 SDN 和 NFV 的一项关键技术。

主要术语：

DPDK：Data Plane Development Kit，DPDK 是一套源码编程库，可以为 Intel 处理器提升基础数据平面功能。

FreeBSD：FreeBSD 是一种 UNIX 操作系统，是由经过 BSD、386BSD 和 4.4BSD 发展而来的 Unix 的一个重要分支。FreeBSD 为不同架构的计算机系统提供了不同程度的支持。它是一个自由的(英文 free 也可以说是免费的)类 UNIX 操作系统(Unix-like)，经由 BSD UNIX 由 AT&T UNIX 衍生而来，FreeBSD 由于法律原因不能称为 UNIX，但由于直接衍生于 BSD UNIX，并且一些原来 BSD UNIX 的开发者后来转到 FreeBSD 的开发，使得 FreeBSD 在内部结构和系统 API 上和 UNIX 有很大的兼容性。

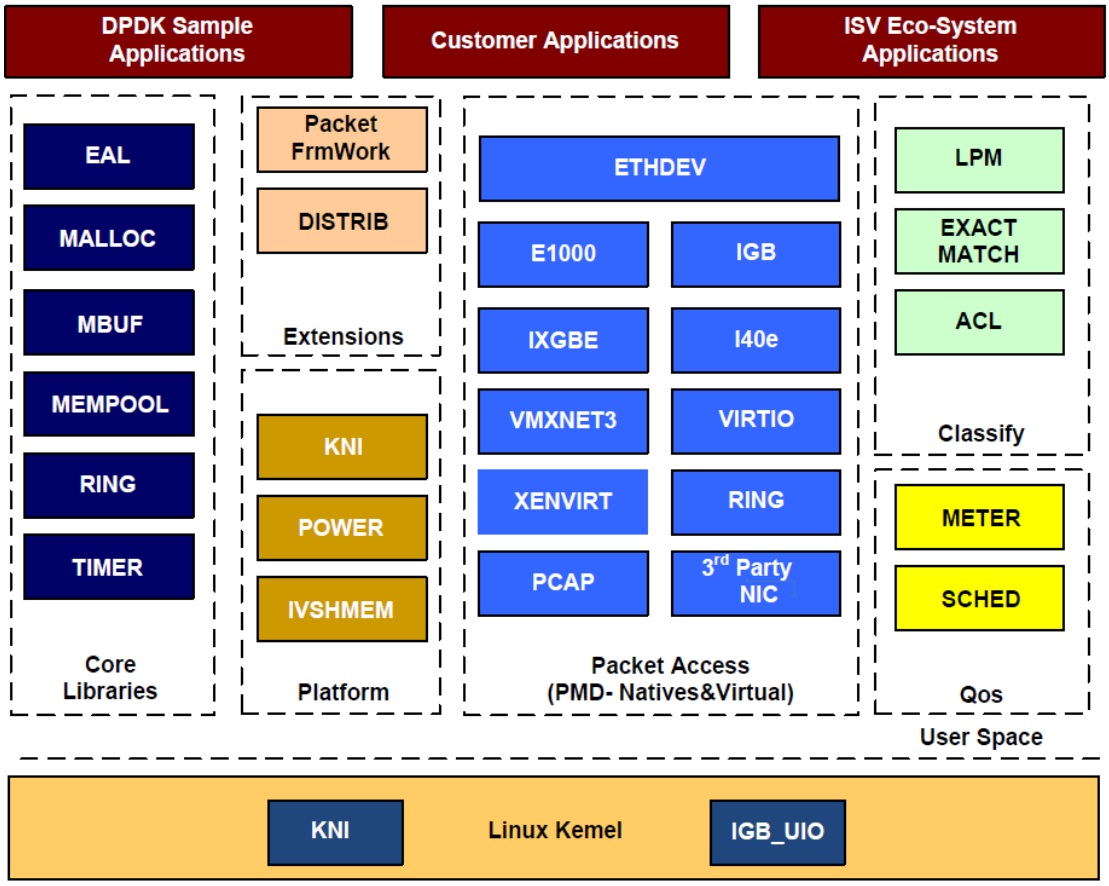
KNI：Kernel NIC Interface 是 DPDK 提供的一种允许用户空间应用程序访问的 Linux 协议栈的接口，类似于 linux 的 TUN/TAP。

UIO：linux Userspace I/O 子系统，是运行在用户空间的 I/O 技术。Linux 系统中一般的驱动设备都是运行在内核空间,而在用户空间用应用程序调用即可,而 UIO 则是将驱动的很少一部分运行在内核空间,而在用户空间实现驱动的绝大多数功能。使用 UIO 可以避免设备的驱动程序需要随着内核的更新而更新的问题。

VFIO：Virtual Function I/O 是一套用户态驱动框架，VFIO 由平台无关的接口层与平台相关的实现层组成，它提供两种基本服务：向用户态提供访问硬件设备的接口、向用户态提供配置 IOMMU 的接口。

2.2 DPDK 架构介绍

DPDK 的组成架构如下图所示，在最底部的内核态（Linux Kernel）DPDK 有两个模块：KNI 与 IGB_UIO。其中，KNI 提供给用户一个使用 Linux 内核态的协议栈,以及传统的Linux 网络工具(如 ethtool, ifconfig)。IGB_UIO(igb_uio.ko 和 kni.ko. IGB_UIO）则借助了 UIO 技术，在初始化过程中将网卡硬件寄存器映射到用户态。



DPDK 的上层用户态由很多库组成,主要包括核心部件库(Core Libraries)、平台相关模块(Platform)、网卡轮询模式驱动模块（PMD-Natives&Virtual）、QoS 库、报文转发分类算法（Classify）等几大类，用户应用程序可以使用这些库进行二次开发，下面分别简要介绍。

核心部件库：该模块构成的运行环境是建立在 Linux 上，通过环境抽象层(EAL)的运行环境进行初始化，包括：HugePage 内存分配、内存/缓冲区/队列分配与无锁操作、CPU 亲和性绑定等；其次，EAL 实现了对操作系统内核与底层网卡 I/O 操作的屏蔽（I/O 旁路了内核及其协议栈），为 DPDK 应用程序提供了一组调用接口，通过 UIO 或 VFIO 技术将 PCI 设备地址映射到用户空间，方便了应用程序调用，避免了网络协议栈和内核切换造成的处理延迟。另外，核心部件还包括创建适合报文处理的内存池、缓冲区分配管理、内存拷贝、以及定时器、环形缓冲区管理等。

平台相关模块：其内部模块主要包括 KNI、能耗管理以及 IVSHMEM 接口。其中，KNI 模块主要通过 kni.ko 模块将数据报文从用户态传递给内核态协议栈处理，以便用户进程使用传统的 socket 接口对相关报文进行处理；能耗管理则提供了一些 API，应用程序可以根据收包速率动态调整处理器频率或进入处理器的不同休眠状态；另外，IVSHMEM 模块提供了虚拟机与虚拟机之间，或者虚拟机与主机之间的零拷贝共享内存机制，当 DPDK 程序运行时，IVSHMEM 模块会调用核心部件库 API，把几个 HugePage 映射为一个 IVSHMEM 设备池，并通过参数传递给 QEMU，这样，就实现了虚拟机之间的零拷贝内存共享。

轮询模式驱动模块：PMD 相关 API 实现了在轮询方式下进行网卡报文收发，避免了常规报文处理方法中因采用中断方式造成的响应延迟，极大提升了网卡收发性能。此外，该模块还同时支持物理和虚拟化两种网络接口，从仅仅支持 Intel 网卡，发展到支持 Cisco、Broadcom、Mellanox、Chelsio 等整个行业生态系统，以及基于 KVM、VMWARE、 XEN 等虚拟化网络接口的支持。

DPDK 还定义了大量 API 来抽象数据平面的转发应用，如 ACL、QoS、流分类和负载均衡等。并且，除以太网接口外，DPDK 还在定义用于加解密的软硬件加速接口（Extensions）。

总体而言，DPDK 技术具有如下特征：

（1）采用 BSD License，保证了可合法用于商业产品

（2）支持 RedHat、CentOS、Fedora、Ubuntu 等大多数 Linux 系统，已开始进入主流 Linux 发布版本。

（3）DPDK 支持 Run to Completion 和 Pipeline 两种报文处理模式，用户可以依据需求灵活选择，或者混合使用。Run to Completion 是一种水平调度方式，利用网卡的多队列，将报文分发给多个 CPU 核处理，每个核均独立处理到达该队列的报文，资源分配相对固定，减少了报文在核间的传递开销，可以随着核的数目灵活扩展处理能力；Pipeline 模式则通过共享环在核间传递数据报文或消息，将系统处理任务分解到不同的 CPU 核上处理，通过任务分发来减少处理等待时延。

（4）DPDK 的库函数和样例程序十分丰富，包括 L2/L3 转发、Hash、 ACL、QoS 等大量示例供用户参考，具体可访问：
http://dpdk.org/doc/guides/sample_app_ug/index.html

2.3 大页技术

x86 处理器硬件在缺省配置下，页的大小是 4K，但也可以支持更大的页表尺寸，例如 2M 或 1G 的页表。使用了大页表功能后，一个 TLB 表项可以指向更大的内存区域，这样可以大幅减少 TLB miss 的发生。早期的 Linux 并没有利用 x86 硬件提供的大页表功能，仅在 Linux 内核 2.6.33 以后的版本，应用软件才可以使用大页表功能，具体的介绍可以参见 Linux 的大页表文件系统（hugetlbfs）特性。

DPDK 则利用大页技术，所有的内存都是从 HugePage 里分配，实现对内存池(mempool)的管理，并预先分配好同样大小的 mbuf，供每一个数据包使用。

2.4 轮询技术

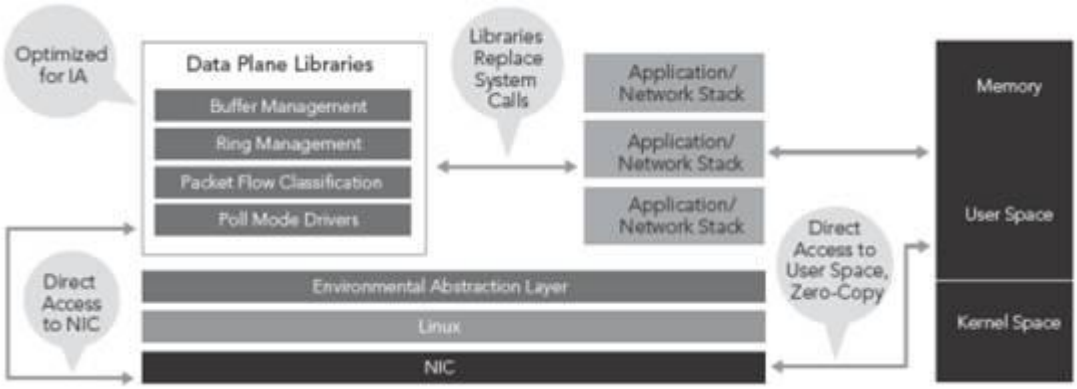
为了减少中断处理开销，DPDK 使用了轮询技术来处理网络报文。网卡收到报文后，直接将报文保存到处理器 cache 中（有 DDIO（Direct Data I/O）技术的情况下），或者保存到内存中（没有 DDIO 技术的情况下），并设置报文到达的标志位。应用软件则周期性地轮询报文到达的标志位，检测是否有新报文需要处理。整个过程中完全没有中断处理过程，因此应用程序的网络报文处理能力得以极大提升。

2.5 CPU 亲和技术

CPU 亲和技术，就是将某个进程或者线程绑定到特定的一个或者多个核上执行，而不被迁移到其它核上运行，这样就保证了专用程序的性能。

DPDK 使用了 Linux pthread 库，在系统中把相应的线程和 CPU 进行亲和性绑定，然后相应的线程尽可能使用独立的资源进行相关的数据处理。

2.6 DPDK 的应用模型



3、DPDK 技术应用优势

（1）提供高性能网络处理能力。

（2）友好的商业许可证。

（3）Intel 主推，业界比较认可，产品可靠性和稳定性较高。

（4）主流网卡厂商基本都有支持。

（5）支持 KVM\XEN\Vmware 等主流虚拟化平台。

4、DPDK 初始化和转发流程

在 Host 上运行用户态 EVS，借助于 DPDK 的网卡管理 API 和大页内存，来提升物理网卡收发包性能和处理能力。

虚拟机启动时，在 XML 中配置为 vhost-usr 类型虚拟网卡，qemu 保证 vhost-usr 后端驱动可以直接访问大页内存中的报文。

用户态 EVS 和 vhost-user 后端驱动之间通过共享收发队列的内存来传递报文，利用批处理、轮询机制和多核转发提升报文处理能力。

4.1 初始化流程

在创建 vswitch 实例时，支持创建用户态 EVS 类型，通过 dpdk 高速数据通道从物理网卡收包,然后批量转发,将报文交给 VM 虚拟网卡后端驱动 vhost-user，再交给 VM 虚拟网卡；或者从 VM 虚拟网卡后端驱动 vhost-user 收包，然后批量转发，将报文交给物理网卡，物理网卡将报文发送出去。

4.2 批量转发流程

初始化完成后，在转发线程内进行轮询，遍历当前线程所服务的每个端口，尝试接收一组报文，然后对每个报文做转发判断，找到目的端口后暂时存放在该目的端口的缓存数组中,当前端口的这些报文都处理结束后,再处理下一个端口。期间如果某个目的端口的缓存数组已满，则直接刷新发送出去。

5、DPDK 技术原理简介

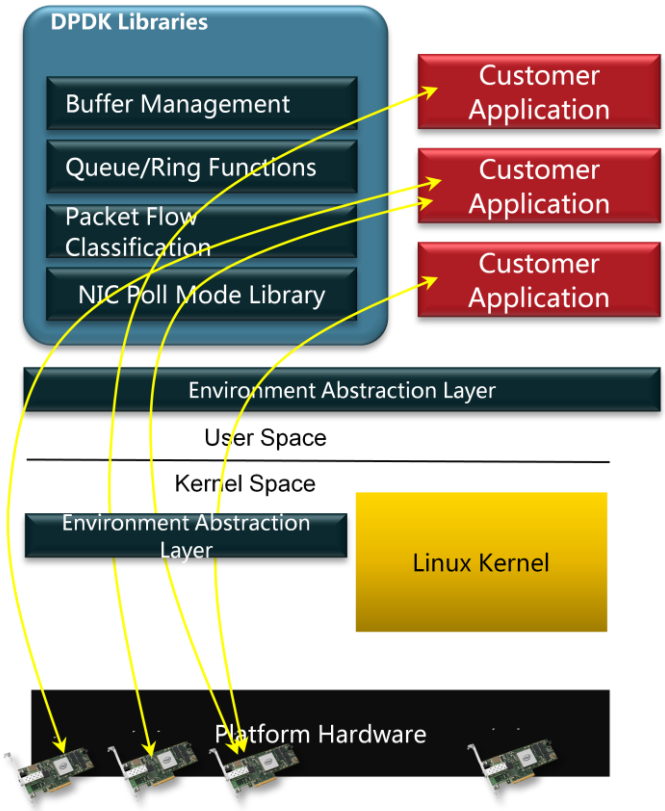
DPDK 是 intel 基于其自家 x86 芯片上开发的一系列的用于快速包处理的驱动及相关基础库。DPDK 大部分跑在 linux 用户态（除 UIO 接口部分会跑在内核态），目前 DPDK 以 BSD license 对外分发，它包括以下几个主要的组件：

- multicore framework：多核框架，dpdk 库面向 intel i3/i5/i7/ep 等多核架构芯片，内置了对多核的支持
- hugepage memory： 内存管理，dpdk 库基于 linux hugepage 实现了一套内存管理基础库，为应用层包处理做了很多优化
- ring buffers：共享队列，dpdk 库提供的无锁多生产者-多消费者队列，是应用层包处理程序的基础组件
- poll-mode drivers：轮询驱动，dpdk 库基于 linux uio 实现的用户态网卡驱动

这些库被用于：

- 在最小的 CPU cycles 内发送或是接收数据包（通常小于 80 个 cycles，DDIO 功能实现）
- 开发快速的报文捕获算法（如 tcpdump 之类的）
- 用于加速第三方的协议栈

典型的 DPDK 组件如下图所示：



上图中 DPDK Libraries 及 Environment Abstraction Layer 均为 DPDK 提供，其中 EAL 模块必须是最先初始化的代码，同一系统下只有一个 EAL 进程为主进程，其它的均为 slave 进程。内核态 EAL 为一驱动，需要事先加载。这部分详见后文描述。下面分别对 DPDK 的各个组件进行简单的介绍。

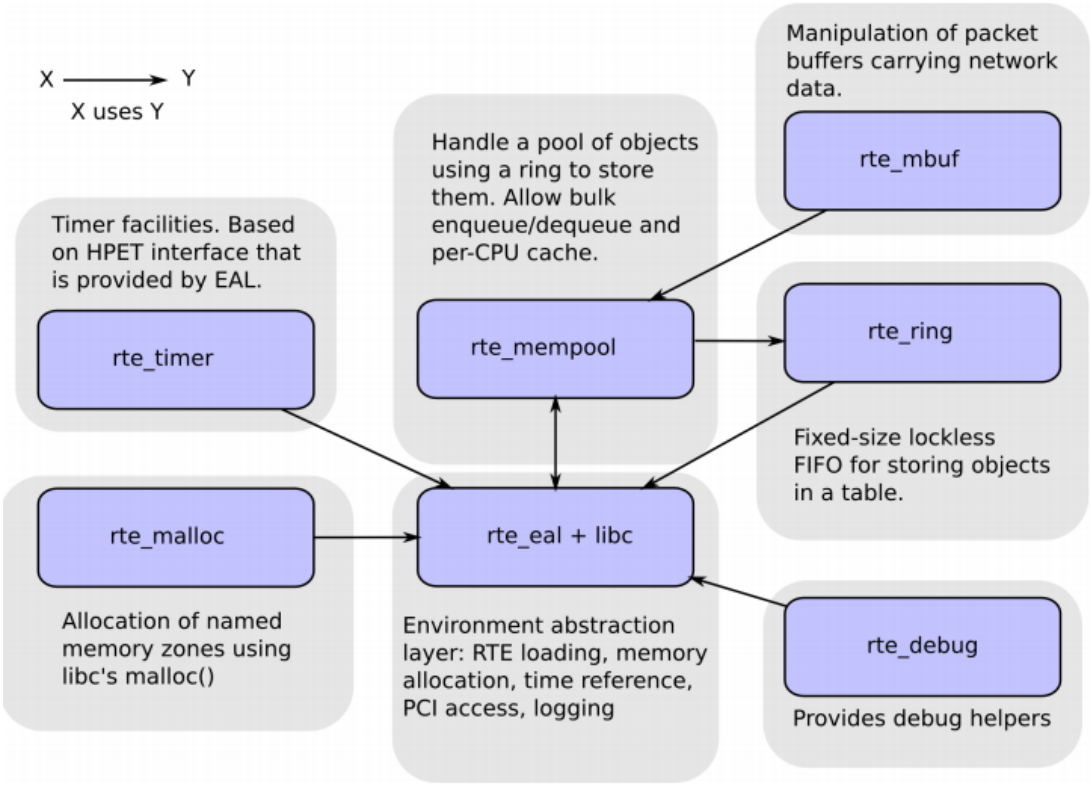
5.1 环境抽象层概述

环境抽象层（Environment Abstraction Layer）提供了一层通用的接口给 DPDK 的各种库使用，同时屏蔽了硬件的各种形态带来的差异化，它提供了以下服务：

- DPDK 的加载及各个 core 间的分发

- CPU CORE 的亲和设置
- 系统内存的申请及描述
- 原子操作及锁操作
- 时间参考源
- PCI 总线访问
- Trace 及 debug 功能

5.2 核心组件分析



从上图中可以看出，DPDK 所有的核心组件均基于 EAL，通过 EAL 提供的相关功能，DPDK 提供了 timer,malloc,mempool,mbuf,ring,debug 等核心功能，另外上图中没有画出的功能还有 FLOW classification 功能，此功能在当前 DPDK（1.7.0）版本中暂未提供，后续版本中会实现；另外关于 POLL-MODE driver 在上图中也没有描述，在接下来的文档中，会有详细的描述。上图中的核心组件分别简要描述如下：

- 内存管理(librte_malloc)：提供 API 从堆中申请内存

- Ring 管理(librte_ring)：从特定大小的表中申请无锁 FIFO。
- 内存池管理(librte_mempool)：从内存中申请对象池，每个内存池由 name 及一个 ring 组成。可提供基于 core 的对象 cache 及对齐，能够保证对象在所有的 RAM 通道的一致性。
- 网络报文缓冲管理(librte_mbuf)：提供基于 DPDK 的应用存储消息所需的 buffer 的创建及销毁，所有的 mbuf 都存储在内存池中，使用 DPDK 的内存池管理(librte_mempool)。
- 定时器管理(librte_timer)：向 DPDK 的应用提供定时器服务用于执行异步功能，它使用 EAL 模块提供的 HPET 接口来获取精确的时间参考源。

DPDK 的 C 运行库选用的是 newlib，一个是基于 license 的考虑，newlib 使用的是 BSD 的 license，另外就是在嵌入式的系统里面，newlibc 也能够有比较优秀的表现，可方便基于 DPDK 的 APP 的移植。Newlib 的所有库函数都建立在 20 个桩函数的基础上，这 20 个桩函数完成一些 newlib 无法实现的功能：

1) I/O 和文件系统访问(open、close、read、write、lseek、stat、fstat、fcntl、link、unlink、rename)；

2) 扩大内存堆的需求(sbrk)；

3) 获得当前系统的日期和时间(gettimeofday、times)；

4) 各种类型的任务管理函数(execve、fork、getpid、kill、wait、_exit)；

这 20 个桩函数在语义、语法上与 POSIX 标准下对应的 20 个同名系统调用是完全兼容的。Newlib 为每个桩函数提供了可重入的和不可重入的两种版本。两种版本的区别在于，如果不可重入版桩函数的名字是 xxx，则对应的可重入版桩函数的名字是_xxx_r，如 close 和_close_r，open 和_open_r，等等。此外，可重入的桩函数在参数表中含有一个_reent 结构指针，这个指针使得系统的实现者能在库和目标操作环境之间传送上下文相关的信息，尤其是发生错误时，能够便捷的传送 errno 的值到适当的任务中。DPDK 内使用的大多数是可重入版本的桩函数。

5.3 DPDK 环境抽象层

环境抽象层用于 APP 访问底层资源，如硬件、内存等。它提供了一种通用的接口来屏蔽底层硬件或是库定义的接口，用于初始化运行环境（内存空间、PCI 设备、定时器、控制台等）。

典型的 EAL 层提供的服务包括：

- DPDK 的加载与运行：intel DPDK 必须与应用程序一起链接成一个 APP，并且必须以某种方式来进行加载并运行。
- 核的亲和设置及工作分配：EAL 提供一种机制将特定的执行单元分配到不同的 core 运行。
- 系统内存预留：EAL 预留不同的内存区域供使用，如预留物理内存给设备交互使用。
- PCI 地址抽象：EAL 提供接口去访问 PCI 设备的地址空间。
- 公用接口：提供 libc 里没有的 spinlock 及原子操作

本章节仅主要介绍 EAL 实现的功能，其中会涉及到多个库的实现，关于这些库的详细实现，请见后面原理分析章节，本章节只做简要描述。

5.3.1 LIBC 与 EAL 的区别

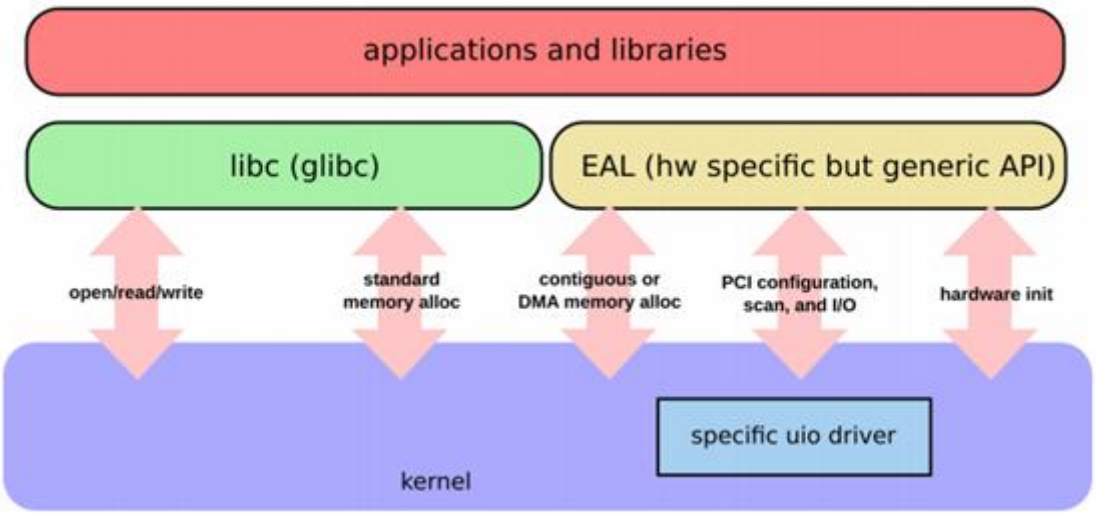
从前面的描述中,我们可以看出 libc 实际上也完也了 EAL 提供的相关接口，关于它们之间的区别实际上是很，对于一个应用程序来说，使用 EAL+LIBC 基本上就可以使用任何应用程序或是库的接口了：

C 标准库基本上提供了通用的操作，如输入/输出、字符处理、类型以及 ISO C 标准里面定义的功能。

而 EAL 是 DPDK 特定使用的，提供了一系列访问特定硬件或是在用户态访问内核的接口供 APP 使用。

在实际的应用编写过程中，可以灵活的使用这两种方式，下图是一个典型的

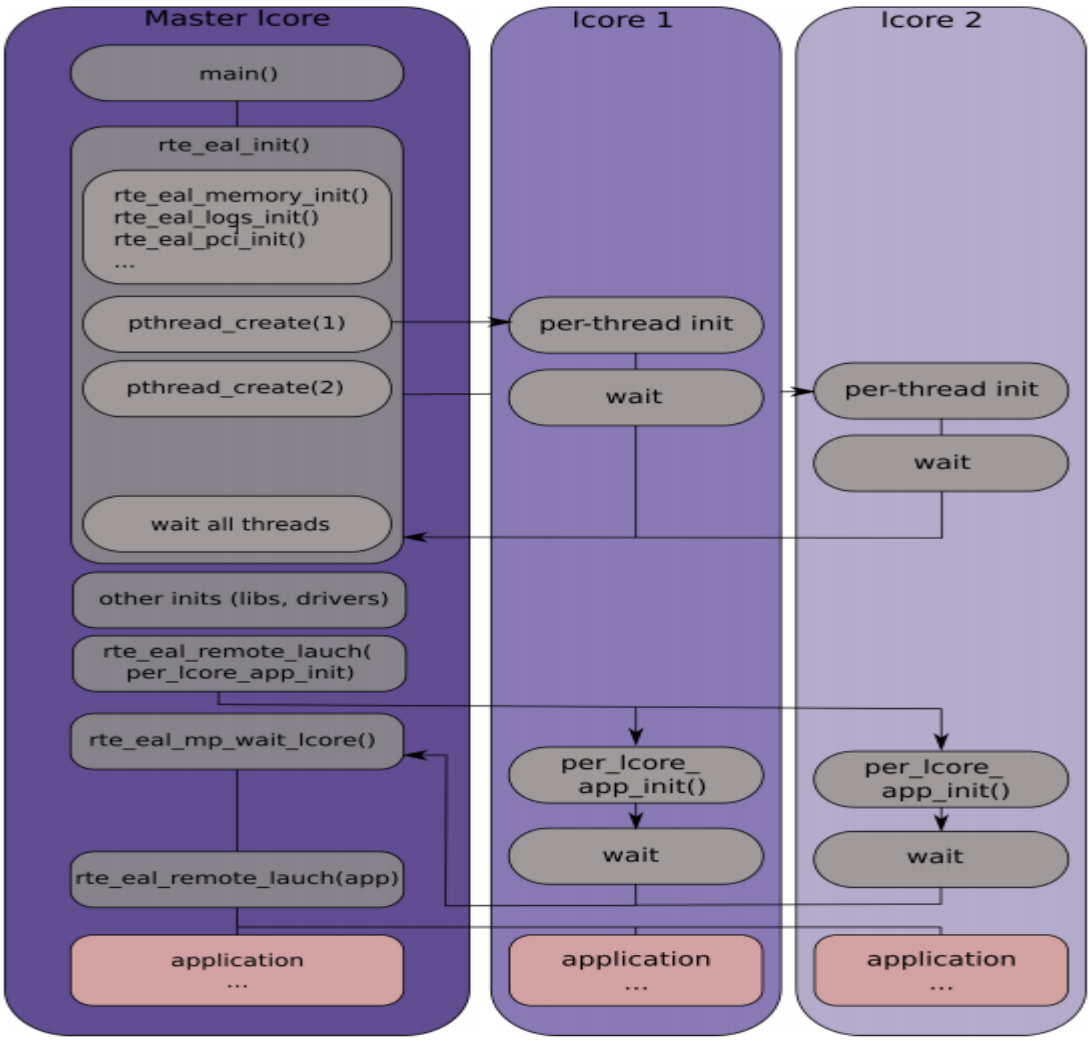
混合用 libc+EAL 的程序框架。



5.3.2 EAL 加载过程

在 linux 用户态的运行环境,DPDP 所有的实例都使用 pthread 库来运行,PCI 设备所有的信息及地址空间的发布是通过内核提供的/sys 下面的文件接口来实现的。同时 EAL 使用 hugetlbfs 来保留一段内存，然后通过 mmap 的方式来使用这些物理内存，并且使用了 huge page 来提高性能，mmap 出来的内存会暴露给 DPDK 的 mempool 使用。DPDK 被初始化后，会通过 pthread 的调用将特定的执行单元绑定到逻辑 core 上去作为用户态的线程运行。

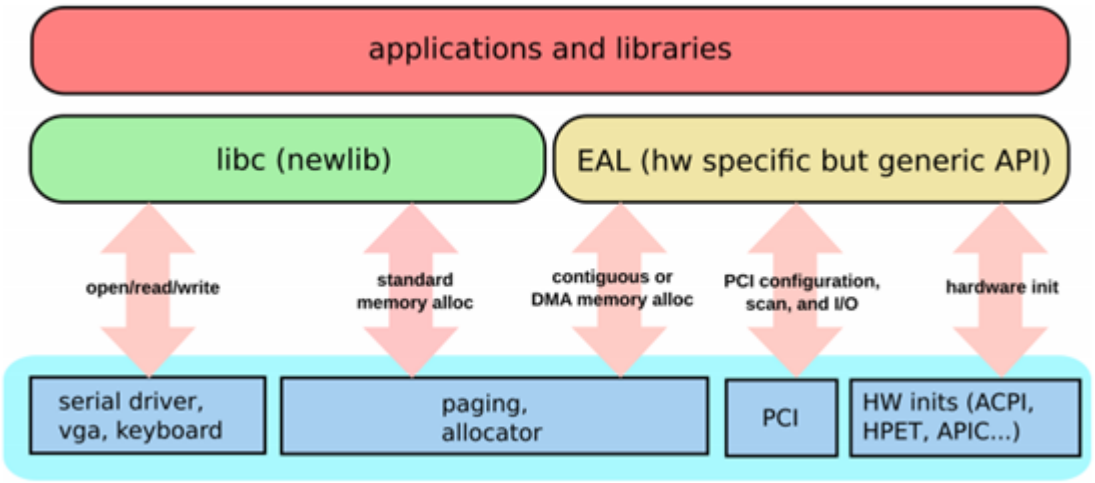
下图是 linux 用户态模式下基于 DPDK 的 APP 初始化及 core 任务分发的过程：



在 APP 运行的时候，首先调的是 main 函数，这个是 glibc 调用，然后在 main 函数内部会去调用 ret_eal_init,这个函数是 DPDK 环境建立起来的初始化函数，会进行 core 的初始化及任务分发操作、内存初始化、LOG 初始化、PCI 设备初始化等操作，这些和硬件强相关的操作完成后，就会根据 core 的数量及配置文件进行线程的创建及线程的绑定，然后紧接着进行其它库及驱动的初始化，所有初始化完成后，就会将最小的执行单元通过 rte_eal_remote_lanch 分发到不同的 core 上去执行。

需要注意的是，在 PCI 访问的时候，EAL 是使用的/sys(具体点就是/sys/bus/pci 及/sys/bus/pci_express)下面的文件去扫描总线上的 PCI 设备，而访问 PCI 设备的内存则是通过 UIO 或是 libpciaccess 来实现的。

注：DPDK 也可以用在裸环境下面，即没有任何操作系统的环境，在这种环境下需要将 DPDK 的镜象通过 GRUB 来进行引导。由于我们没有使用这个功能，本文暂不介绍。



5.3.3 内存分片介绍

由于与硬件交互时必须要使用连续的物理内存，而 DPDK 的内存申请是依赖 OS 的机制的（如 huge page），OS 并没有办法保证所有申请的物理内存的地址是连续的（因为有空洞存在），所以申请的内存是以在一个表中的多个描述符的形式进行分片组织起来的，每一个分片描述符（rte_memseg）表示了一部分物理内存、虚拟地址都连续，并且都在同一个 socket，pagesize 也相同的 hugepage 页面的集合，这样做的好处就是优化内存。

而 memzone 是通过 rte_memzone_reserve 来从 rte_memseg 中分配那些基 dpdk hugepage 的属于同一个物理 cpu 的物理内存连续的虚拟内存也连续的一块地址。Memzone 是 DPDK 内存管理最终向用户程序提供的基础接口，ret_mempool 内的组件均依赖于 rte_memzone 来实现。

6、DPDK 内存管理功能介绍

6.1 Malloc 函数库介绍

librte_malloc 库提供了 API 去申请任意大小的内存空间，需要注意的是此库所有提供内存申请接口要比基于 mempool 的申请要慢一些，出于性能的考虑，此类接口不要用在数据处理流程中，最好是只用在配置过程。

使用此库进行申请内存的时候，需要为所申请的内存命名并指定对齐方式，如为网络配置申请内存，则格式如下：

```
/* args are type, size, align */
rte_malloc("net_config", sizeof(x), 0);
```

同时为了避免内存泄露，可以为每一个申请的内存指定最大可申请的内存阈值，如配置 net_config 最多只能使用 4K 的大小，则可用以下方式来限定(1.7.0 版本暂未实现)：

```
rte_malloc_set_limit(“net_config”, 4096);
```

另外还有一些接口函数比较重要：

rte_malloc_validate：用于验证给定的指针及大小是否处于有效的内存申请区域，这个可以有效的防止缓冲区溢出。

rte_malloc_dump_stats：打印指定 type name 的内存的统计信息，如果 type name 为空则打印所有使用此库申请的内存统计信息，如下：

```
fprintf(f, "Socket:%u\n", socket);
fprintf(f, "\tHeap_size:%zu,\n", sock_stats.heap_totalsz_bytes);
fprintf(f, "\tFree_size:%zu,\n", sock_stats.heap_freesz_bytes);
fprintf(f, "\tAlloc_size:%zu,\n",
```

sock_stats.heap_allocsz_bytes);

fprintf(f, "\tGreatest_free_size:%zu,\n”, sock_stats.greatest_free_size);

fprintf(f, "\tAlloc_count:%u,\n",sock_stats.alloc_count);

fprintf(f, "\tFree_count:%u,\n", sock_stats.free_count);

rte_malloc_socket /rte_zmalloc_socket/ rte_calloc_socket:

申请指定堆上的内存，可以通过 soket ID 来指定靠近哪个 CPU。

6.2 Ring 函数库介绍

Ring 库用于队列管理，相比连表，rte_ring 有以下特点：

- FIFO 机制
- 最大长度固定，指针存在于表中
- 无锁机制
- 多个/单个生产者入队
- 多个/单个消费者出队
- 批量入队
- 批量出队

rte_ring 相对于链表队列具有以下的优势：

1. 更快，只有一个 CAS 指令，而一般的链表队列需要多个 dCAS 指定来完成原子操作
2. Rte_ring 的无锁机制比链表队列的无锁机制要更简单(关于无锁 ring buffer 的分析，更详细的内容请见：http://lwn.net/Articles/340400/)
3. 适用于批量排队/出队操作，因为指针是存储在一个表中，多个对象出队不会像链表队列一样产生较多缓存未命中（因为链表需要多次的指针操作）。此外，多个对象一次批量出队并不比单个对象的出队开销要大。

当然，对比链表队列，缺点也是明显的：

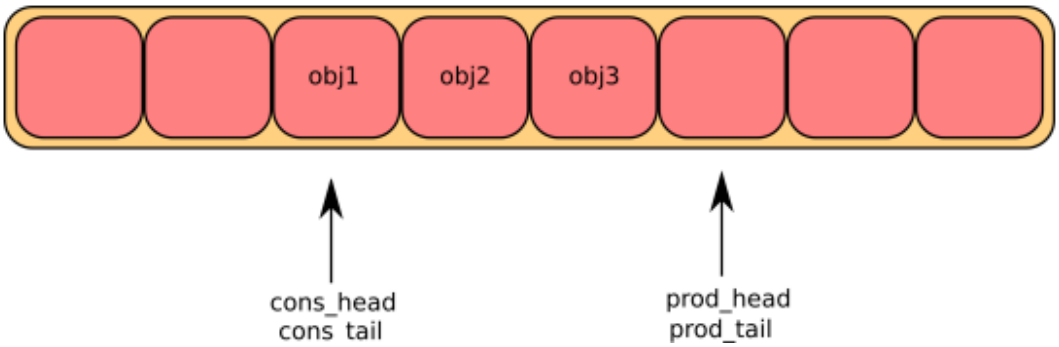
1. 大小是固定的，这个是由其一开始申请时传入的大小决定的，而链表队列大小几乎是没有限制的。
2. 比链表队列要花更多的内存，即使成员是全空的，也至少有 N 个 ring 的指针

rte_ring 的结构如下：

```
struct rte_ring {
    char name[RTE_RING_NAMESIZE];    /**< Name of the ring. */
    int flags;                       /**< Flags supplied at creation. */

    /** Ring producer status. */
    struct prod {
        uint32_t watermark;      /**< Maximum items before EDQUOT. */
        uint32_t sp_enqueue;     /**< True, if single producer. */
        uint32_t size;           /**< Size of ring. */
        uint32_t mask;           /**< Mask (size-1) of ring. */
        volatile uint32_t head;  /**< Producer head. */
        volatile uint32_t tail;  /**< Producer tail. */
    } prod __rte_cache_aligned;

    /** Ring consumer status. */
    struct cons {
        uint32_t sc_dequeue;     /**< True, if single consumer. */
        uint32_t size;           /**< Size of the ring. */
        uint32_t mask;           /**< Mask (size-1) of ring. */
        volatile uint32_t head;  /**< Consumer head. */
```

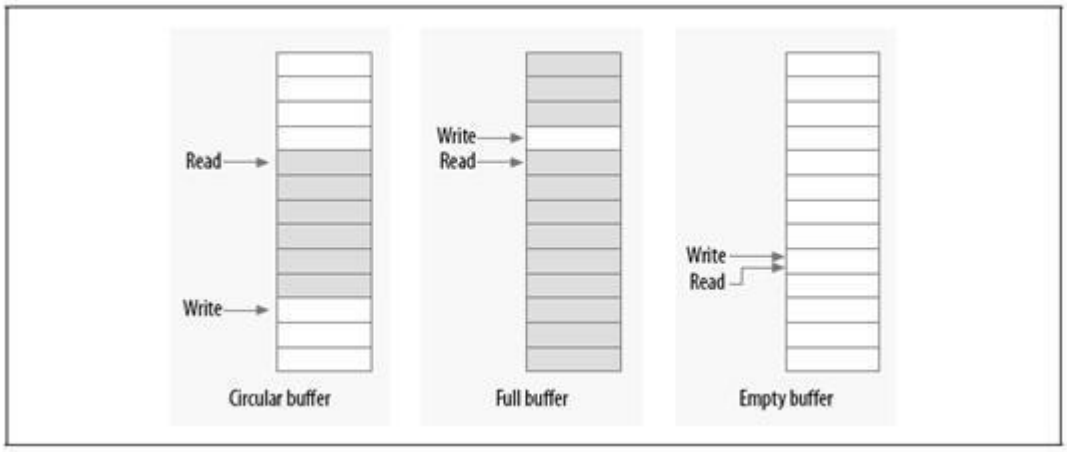
```
        volatile uint32_t tail;  /**< Consumer tail. */
#ifdef RTE_RING_SPLIT_PROD_CONS
    } cons __rte_cache_aligned;
#else
    } cons;
#endif

#ifdef RTE_LIBRTE_RING_DEBUG
    struct rte_ring_debug_stats stats[RTE_MAX_LCORE];
#endif

    void * ring[0] __rte_cache_aligned; /**< Memory space of ring starts
here.
                                         * not volatile so need to be
careful
                                         * about compiler re-ordering
*/
};
```

示意图如下：



在 linux 内核的 ring 处理过程中，生产者将数据放入数组的尾端，而消费者从数组的另一端移走数据，当达到数组的尾部时，生产者绕回到数组的头部。

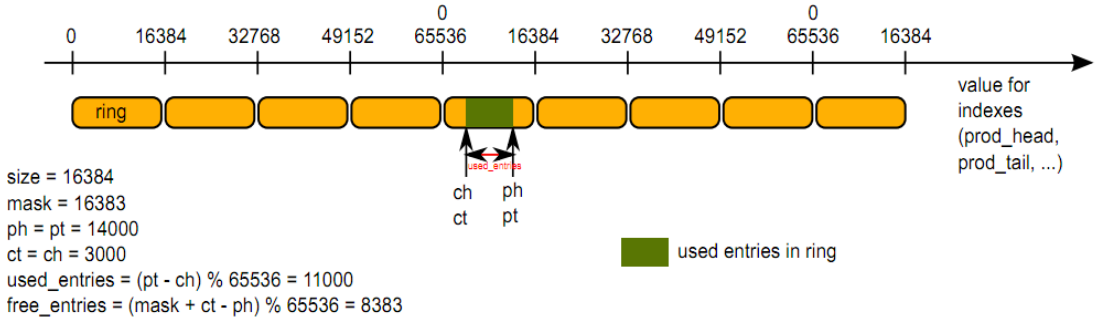
如果只有一个生产者和一个消费者，那么就可以做到免锁访问整个 ring buffer。写入索引只允许生产者访问并修改，只要写入者在更新索引之前将新的值保存到缓冲区中，则用户将始终看到一致的数据结构。同理，读取索引也只允许消费者访问并修改。内核的 ring 如下图所示：



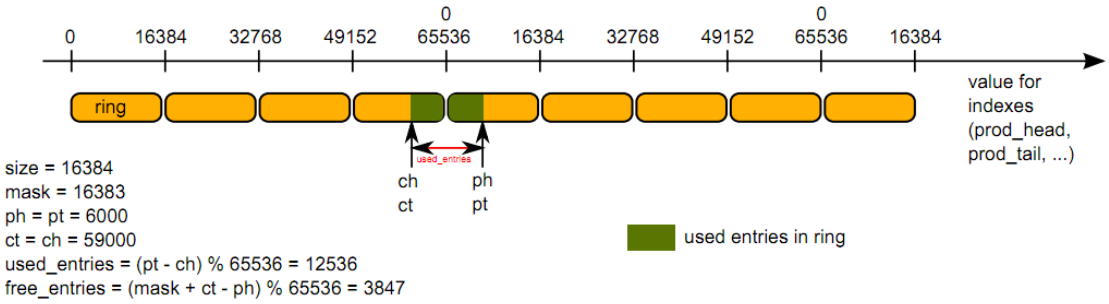
如图所示，当读者和写者指针相等时，表明缓冲区是空的，而只要写入指针在读取指针后面时，表明缓冲区已满。

而在 DPDK 中，实际上使用的思想也是差不多的，但为了提高效率，采用了无锁的设计。下面我们将以实际的出队入列操作过程来描述其处理机制：

在一个 ret_ring 结构体里面会有生产者和消费者两个成员结构体，而在每个成员结构体里面均有 head 和 tail 两个指针，我们在下面的处理过程中称之为：prod_head，prod_tail，cons_head，cons_tail，这四个变量均代表着一个 32 位的索引值。为了简明的指出这些索引是如何在一个 ring 里面起作用的，我们假设是运行在一个 16 位的环境，同时这几个值也是 16 位的，详见下面两张图：



从上图中可以看出，这个 ring 使用的 entries 是 11000，还剩 5383 可用.



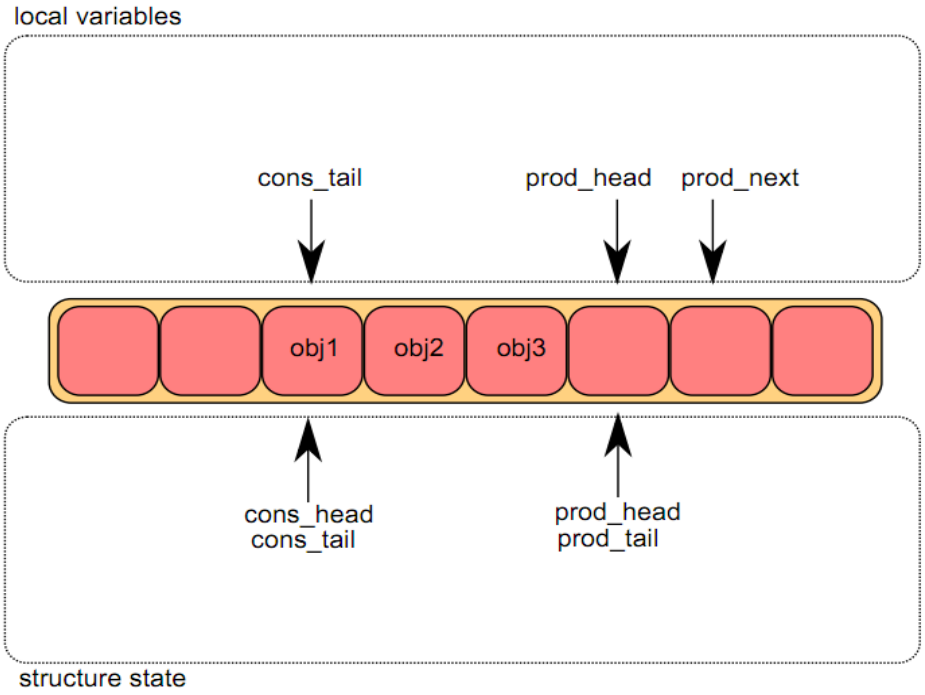
从上中图可以看出，这个 ring 使用的 entries 是 12536，还剩 3847 可用.
实际上代码自动维护了在 0~(size-1)的范围内生产者和消费者之间的距离，由于这个设计利用了整数的溢出，实际上可用的 entry 和剩余的 entry 可以用下面的两个公式来代替：

```
uint32_t entries = (prod_tail - cons_head);
uint32_t free_entries = (mask + cons_tail - prod_head);
```

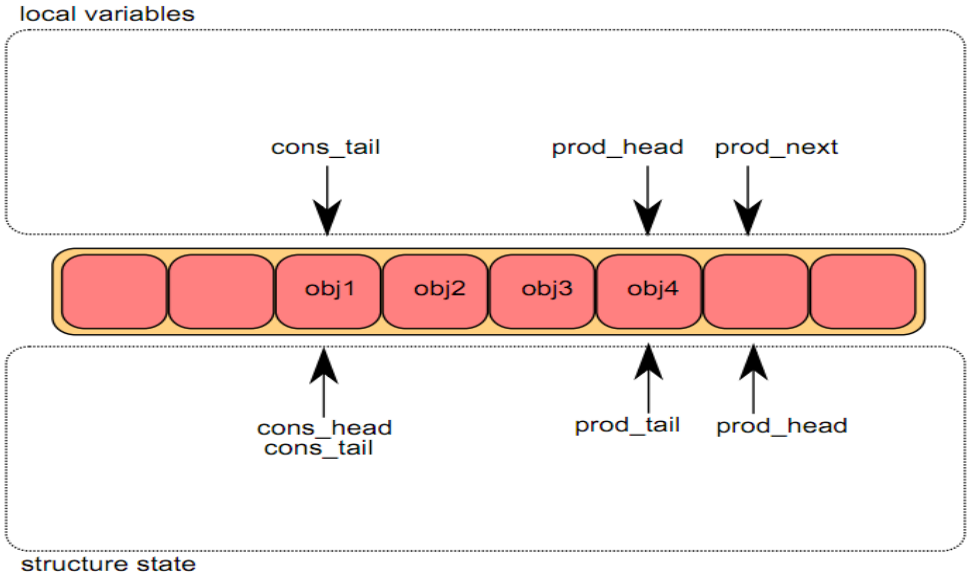
6.2.1 单个生产者入队

当一个生产者想加一个对象加入 ring 的时候，会产生一个入队的操作，在整个操作过程中仅有生产者的 pord_head 和 prod_tail 被修改，并且同一时候仅有一个生产者。在入队操作前，pord_head 和 prod_tail 必须是指向同一个位置（即上一次入队操作已完成）。

第一步：ring->prod_head 和 ring->cons_tail 将会被拷贝至本地临时变量将临时变量 prod_next 指向表中的下一个元素或是下 N 个元素（批量入队），如果 ring 中没有足够多的空间容纳这些元素(通过检查 cons_tail，即查询 free_entries 的个数，公式：mask + cons_tail - prod_head)，将会返回 error。整个过程如下图所示：



第二步：修改 ring 结构体里面的 ring->prod_head 的值，让其移动到 pord_next 的地方，同时被加入的对象的指针也会拷贝到 ring 里面去，如下图所示：



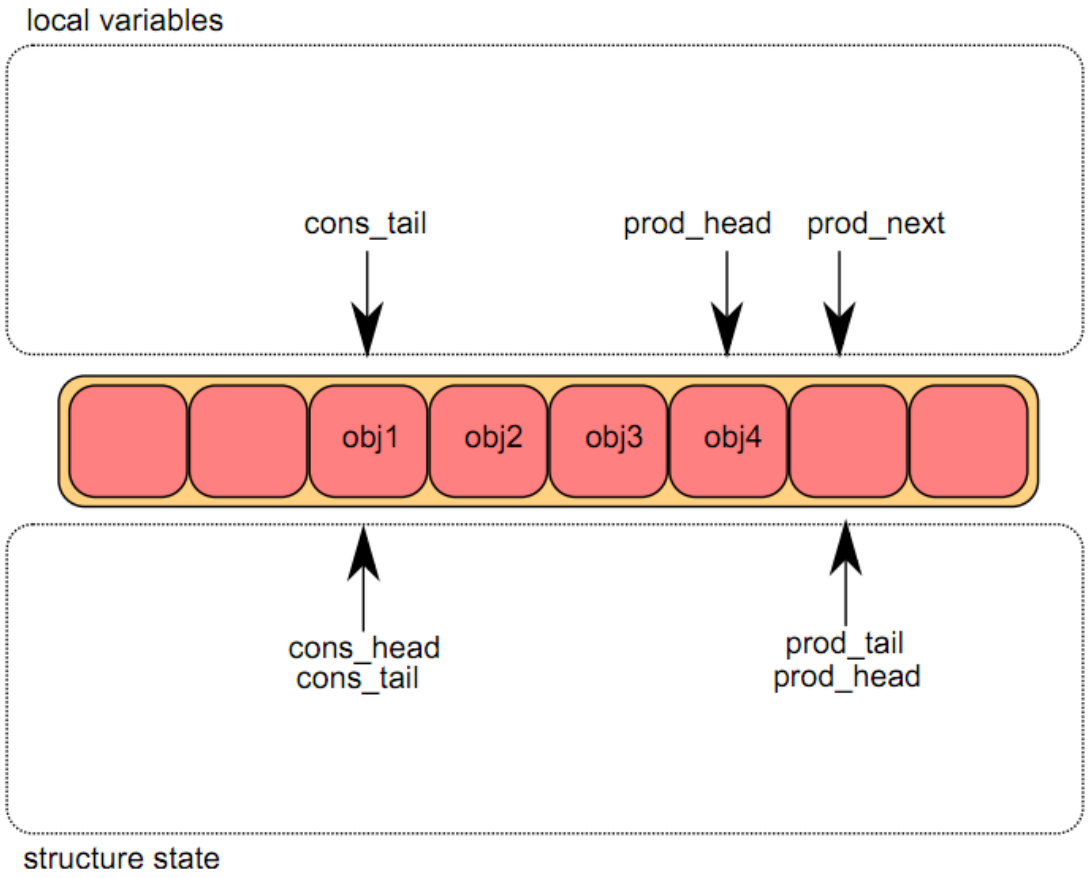
指针拷贝的方式由一个宏来完成，同时需要执行 barrier 操作避免编译器对指令乱序。

```
/* write entries in ring */
```

```
ENQUEUE_PTRS();
rte_compiler_barrier();
```

ENQUEUE_PTRS 宏的操作实际上就是将对象的指针按字节拷贝到 rte_ring 里面的对齐的内存区域里面。

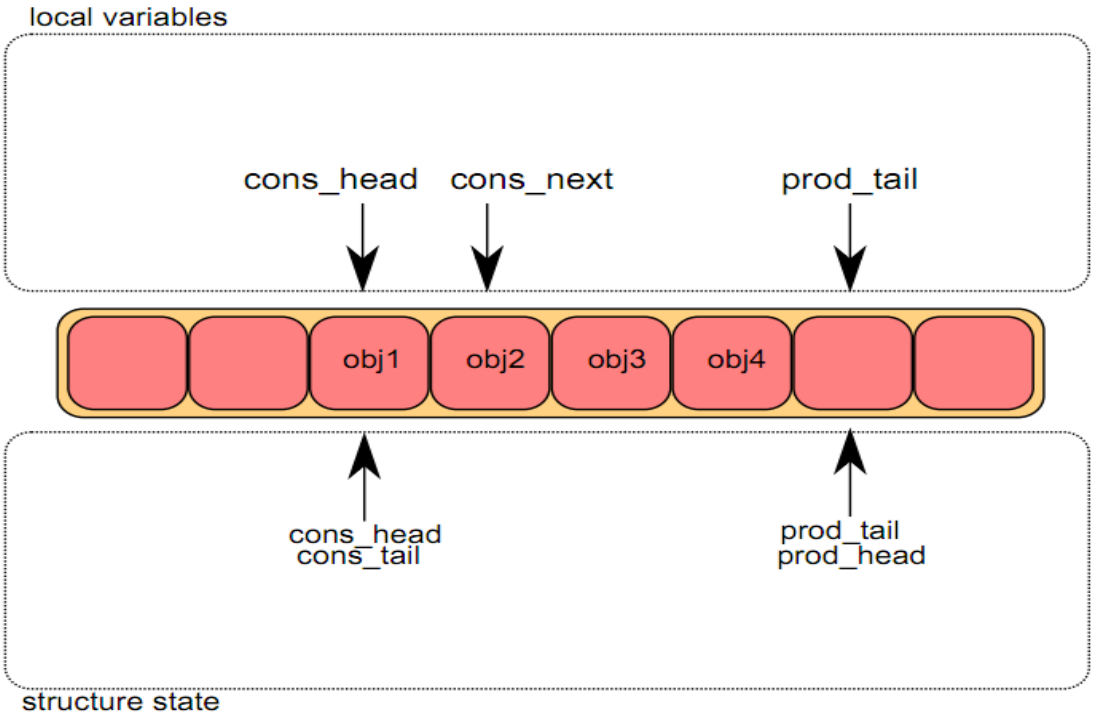
最后一步：当已经完成对象指针添加到 rte_ring 结构体后，需要将 rte_ring 结构体里面的 prod_tail 指向本地临时变量 prod_next 所指向的位置，这样才算一次完整的入队操作完成，示意图如下图所示：



6.2.2 单个消费者出队

同单个生产者入队，在单个消费者出队的时候，仅会修改消费者的 cons_head 和 cons_tail，前提条件也是要求消息者的 cons_head 和 cons_tail 指向同一个位置。

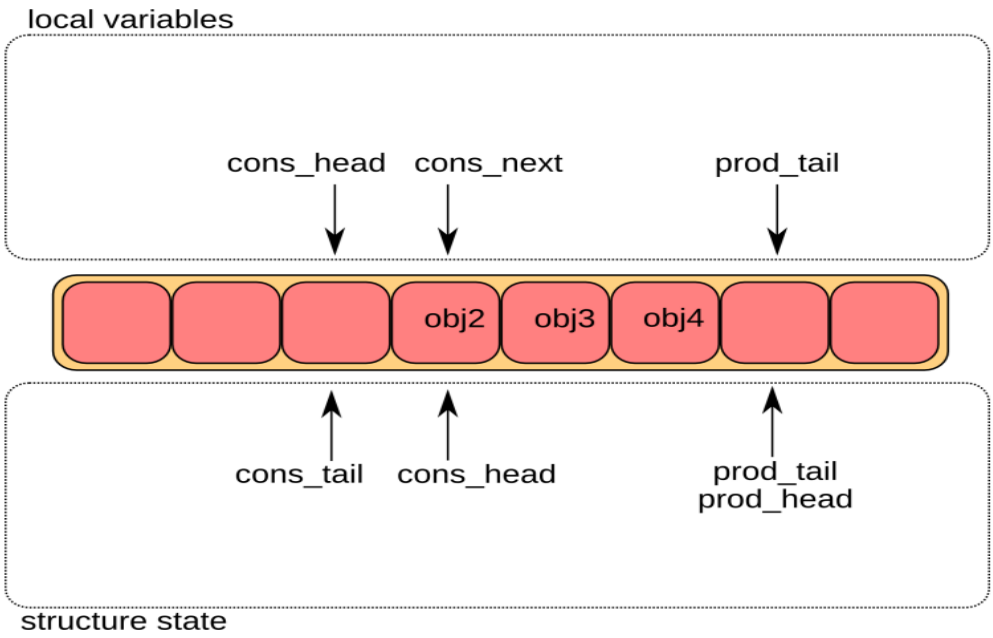
第一步：将 rte_ring 结构体里面的 cons_head 和 prod_tail 拷贝到本地临时变量，将临时变量 cons_next 指向表中的下一个元素或是 N 个元素的位置（批量出队），通过检查 prod_tail 来判断是否有足够的元素来出队，没有的话就返回 error(即查询 entries 的个数，公式： prod_tail - cons_head)，示意图如下所示：



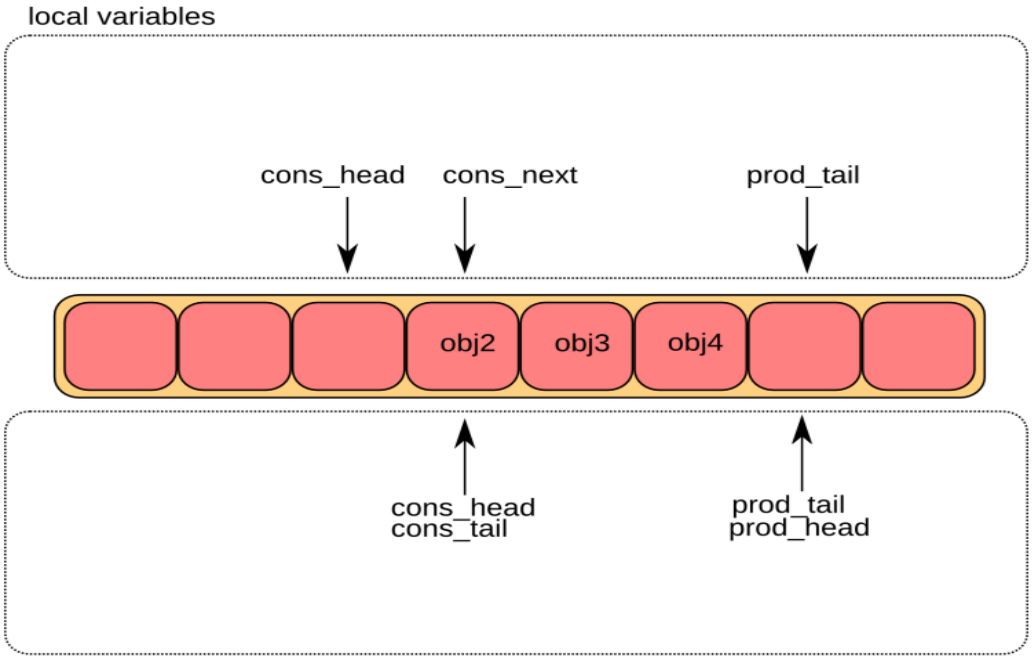
第二步:修改 rte_ring 结构体里面 cons_head,将其指向 cons_next 的位置，然后将中间出队的对象的指针拷贝给用户传入的对象。

拷贝操作由 DEQUEUE_PTRS 宏完成，同样也需要 barrier 操作。

```
/* copy in table */
DEQUEUE_PTRS();
rte_compiler_barrier();
```



第三步：修改 rte_ring 结构体里面的 cons_tail，将其指向 rte_ring 结构体 cons_next 的位置，至此一个完整的出队操作完成，示意图如下：

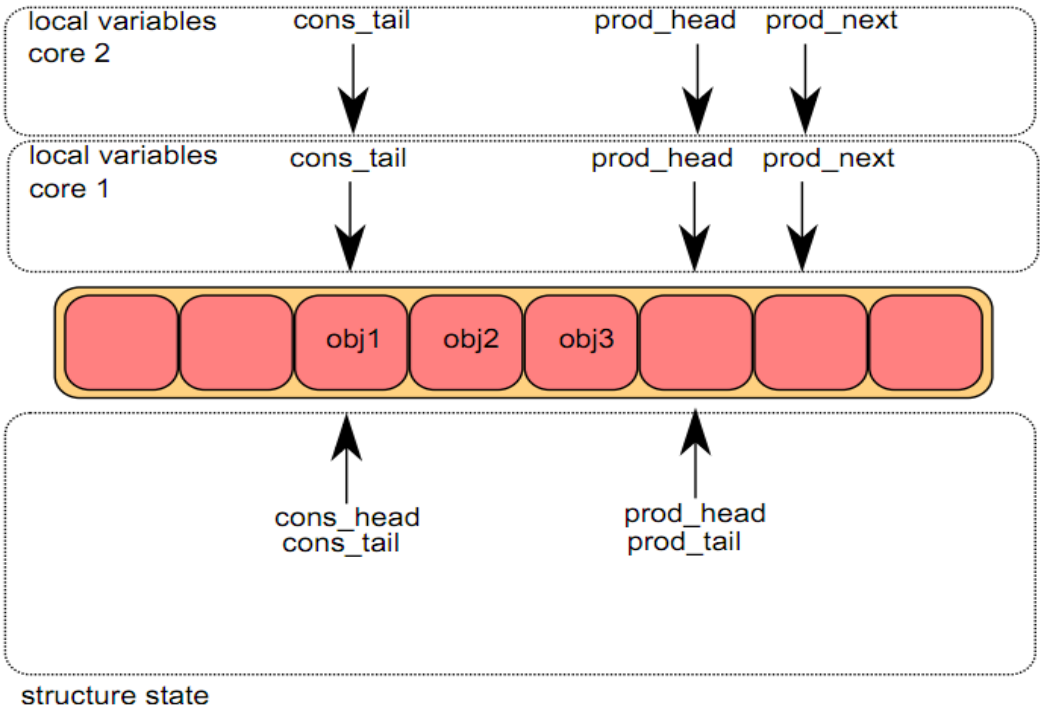


6.2.3 多个生产者入队

在 DPDK 中，ring 一般是作用于一个 core 内，当两个以上的 core 使用到了同一个 ring 的时候，就有可能会产生多个 core 上的生产者同时在执行入队操作，这

种情况下就需要用到多个生产者入队的相关操作接口，当多个生产者同时入队的操作过程中，只有生产者的 head 和 tail 会被修改，前提条件依然是 prod_head 和 prod_tai 要指向同一个位置。下面我们以 2 个 core 同时发起入队操作为例进行说明：

第一步：两个 core 上均将 rte_ring 的 pord_head 和 cons_tail 拷至本地的临时变量，临时变量 prod_next 指向表中的一下个或是下 N 个元素（批量入队），同时通过 cons_tail 来检查 free_entries 的数量来判断是否有足够的空间容纳待入队的元素（细心的读者在这里已经会有是否产生冲突的疑问了，请接着看）。示意图如下所示：



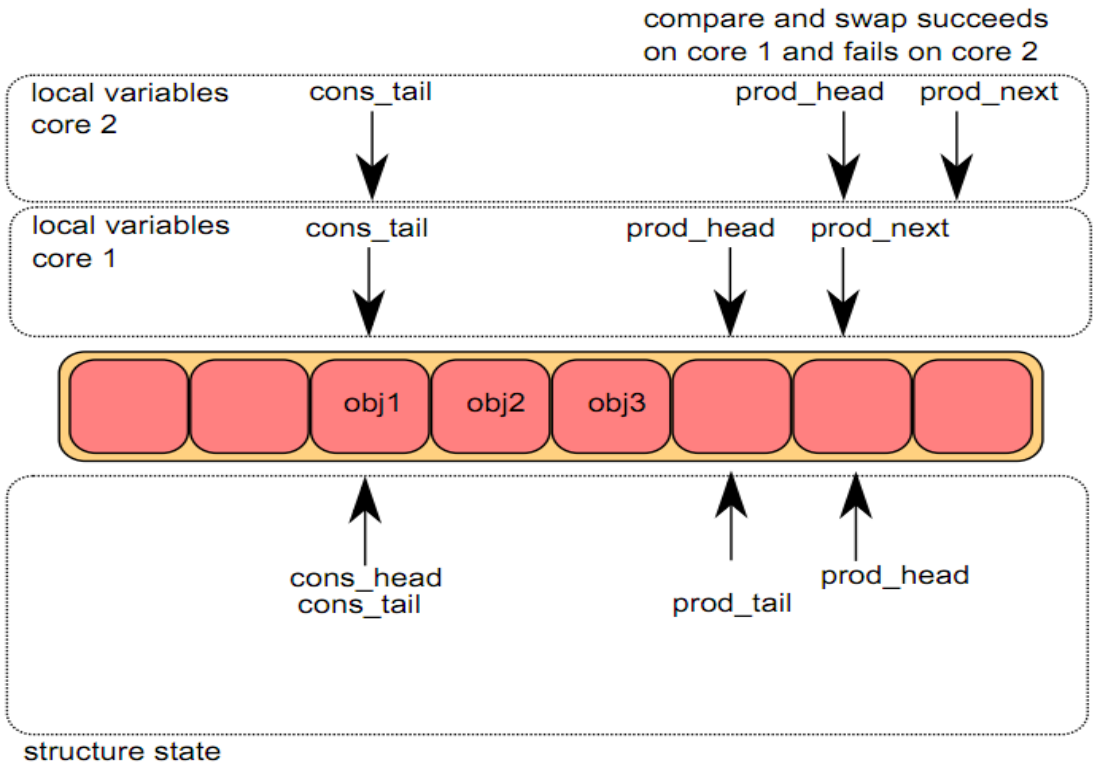
第二步：改写 rte_ring 里面的 prod_head 值为本地临时变量 prod_next，当两个 core 同时操作的时候，肯定就会有冲突产生，所以这里使用了 CAS 的指令，利用这个指令执行以下操作：

1、将 rte_ring 的 prod_head 与本地临时变量 prod_head 进行对比，当一致时 CAS 会将 rte_ring 的 prod_head 值修改为 prod_next 然后返回成功，紧接着

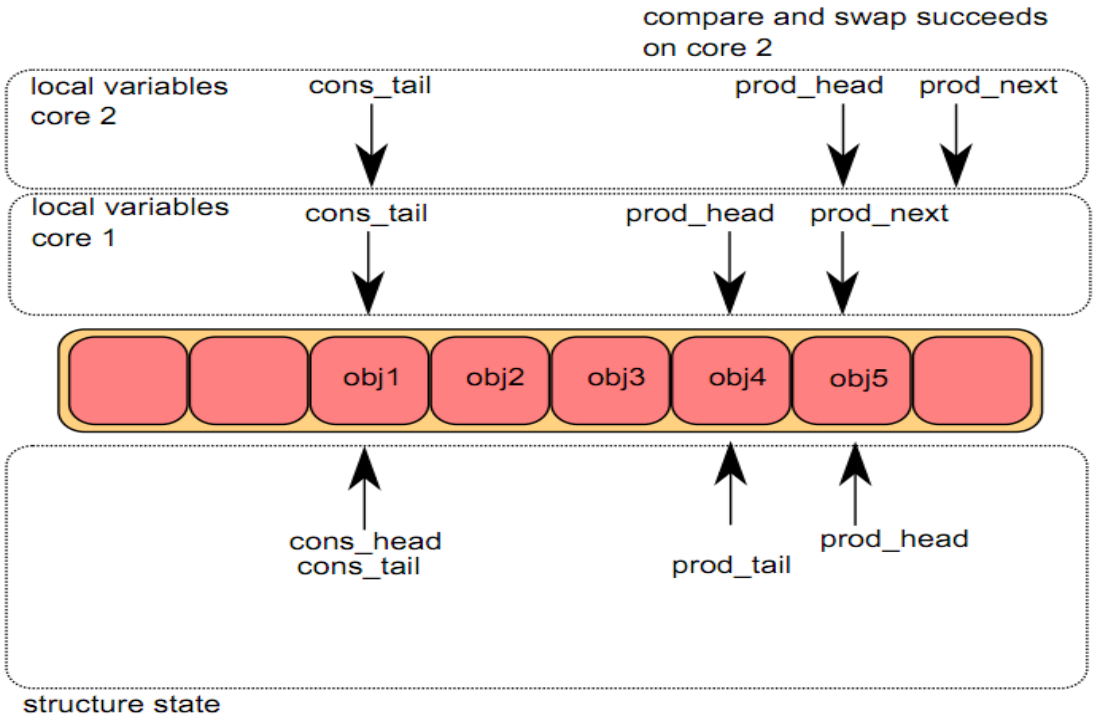
进行第三步，由于有两个 core 同时操作，而 CAS 是个原子操作，这也就是说总会有一个失败，那么如果失败的话就进入下面的操作。

2、执行失败的 CAS 操作的 CPU 将重新执行第一步的操作过程。

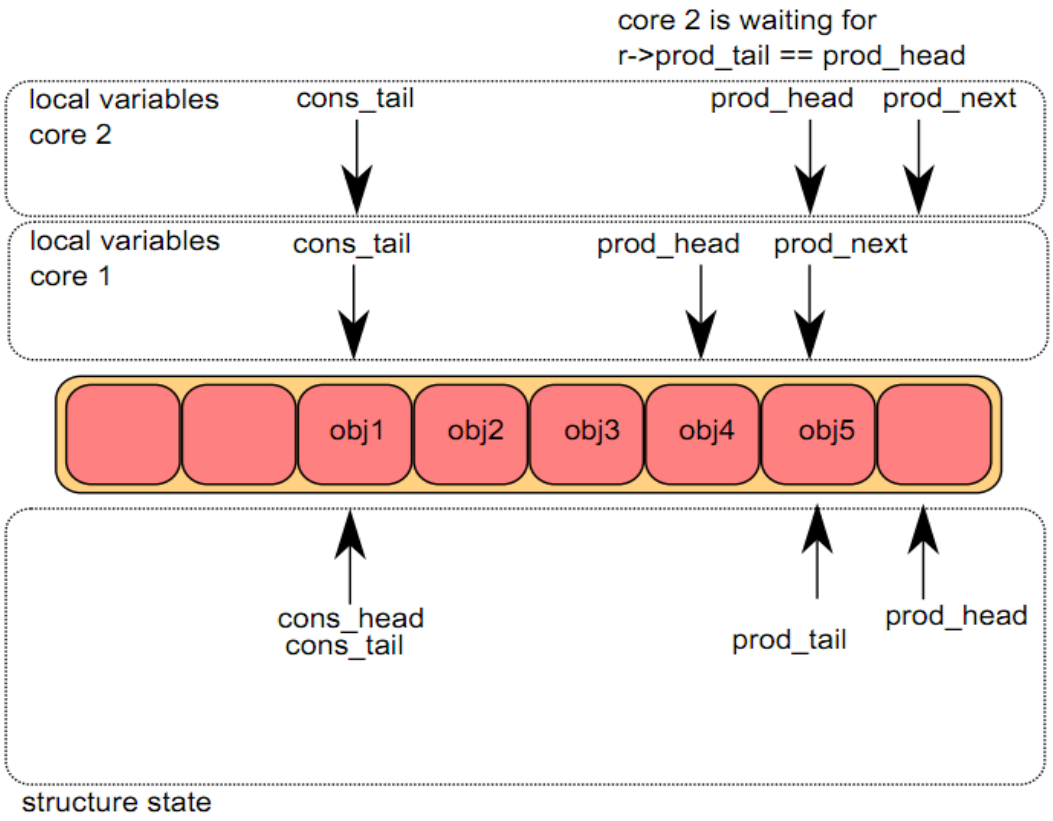
就这样，通过 CAS 可以完美的解决多并发情况下的互斥问题，也没有带来锁操作，第二步示意图如下，这个图表示 core 1 上的执行成功，而 core 2 上的执行失败后进入第一步重新执行的情况：



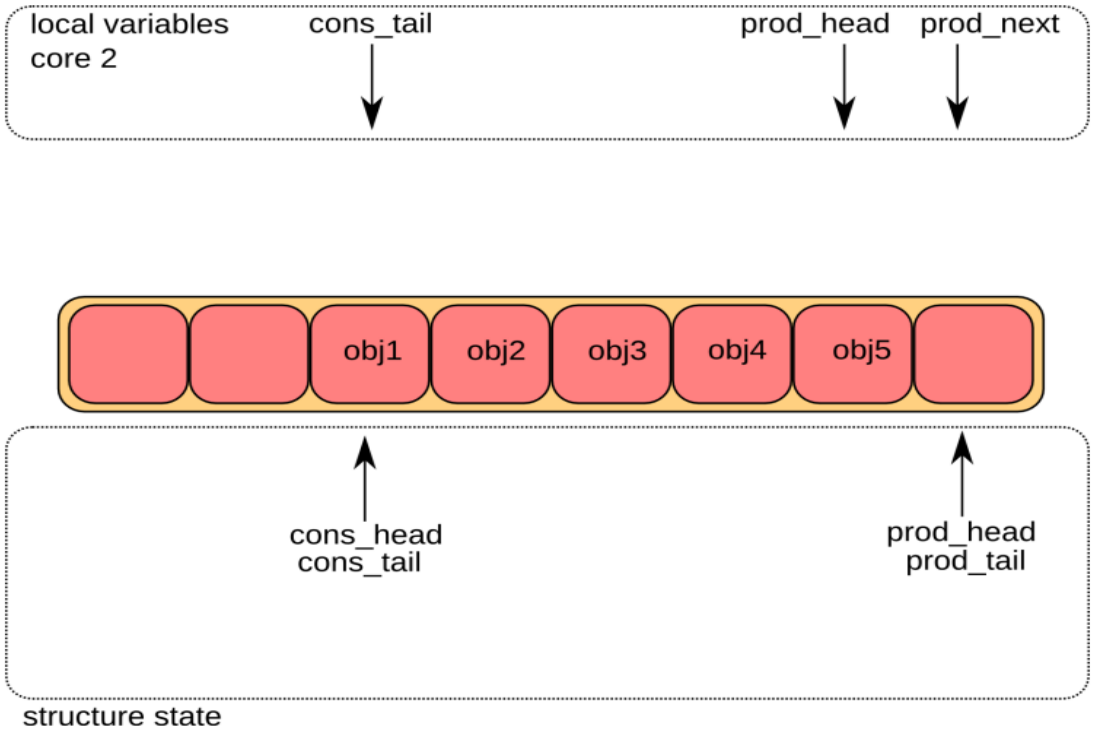
第三步：当 core 2 上面的 CAS 操作被执行成功后，core 1 和 core2 先后将元素指针 obj4,obj5 加入 rte_ring 里面，这里的操作同单个生产者的入队操作，示意图如下：



第四步:所有的元素加入后,2个core都想更新rte_ring里面的prod_tail,那么谁来完成更新呢?这里不会用到CAS了,而是通过两部更新来完成的,首先通过判断哪个core上面的本地临时变量prod_head与rte_ring的prod->tail相等,则证明这个CPU是先完成的,然后会将rte_ring的prod->tail更新为临时变量prod_next的值,示意图如下所示:



第五步：当一个 CPU 更新完 rte_ring 的 prod_tail 后，另一个 CPU 实际上也在等这个值与自己本地临时变量 prod_head 相等的时候，一旦更新完成后，另一个 CPU 的条件也满足了，这个时候它会马上更新 rte_ring 的 prod_tail 为本地临时变量 prod_next 的值，示意图如下所示：



Core2 的等待过程是有延时的，DPDK 通过 rte_pause()来实现，如果 CPU 支持 SSE2 的指令，那么这个等待是会调用_mm_pause，这个是一个轻度的循环等待，否则的话就是忙等待。

3.2.4 多个消费者的出队

多个消费者的出队中互斥的解决办法同多个生产者的入队，整个处理过程及索引值的改变结合单个消费者的出队及多个生产者的入队很容易推导出来，这里不再赘述。

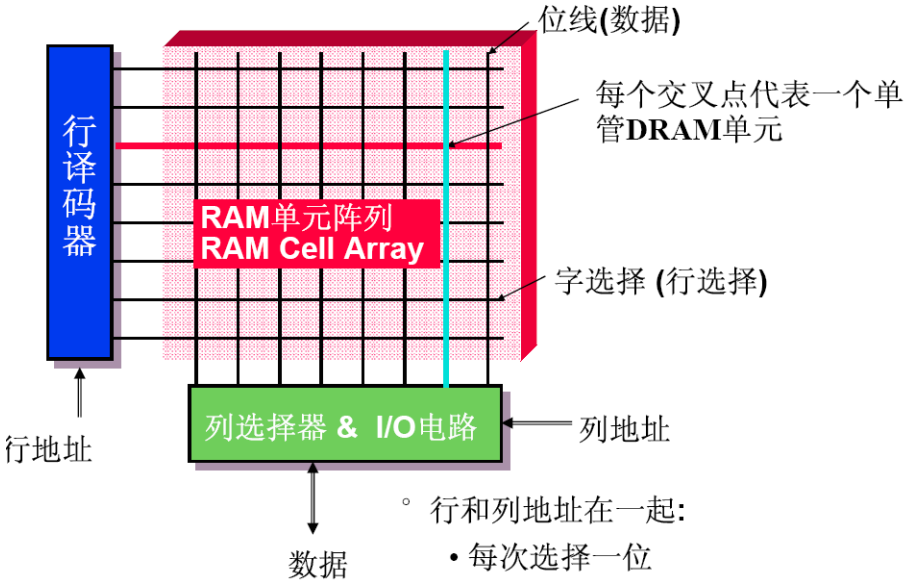
6.3 Mempool 函数库介绍

Mempool 库用于申请固定大小的对象，通常由一个字符串组成 name 来定义一个 mempool，使用一个 rte_ring 来存储可用的对象单元。它可以提供基于 core 的 cache 以及对齐机制来保证对象在所有的 RAM 通道上平均分布。Mempool 库被 DPDK mbuf 库及 EAL 层使用（EAL 层主要使用这个库用于日志记录）。

6.3.1 内存对齐的约束

在介绍内存齐齐的约束之前，我们先得了解一点内存的基础知道，目前大多数使

用的内存为 DDR,而 DDR 实际上是由行列地址及片选等硬件电路组成的存储单元，其大致结构如下：



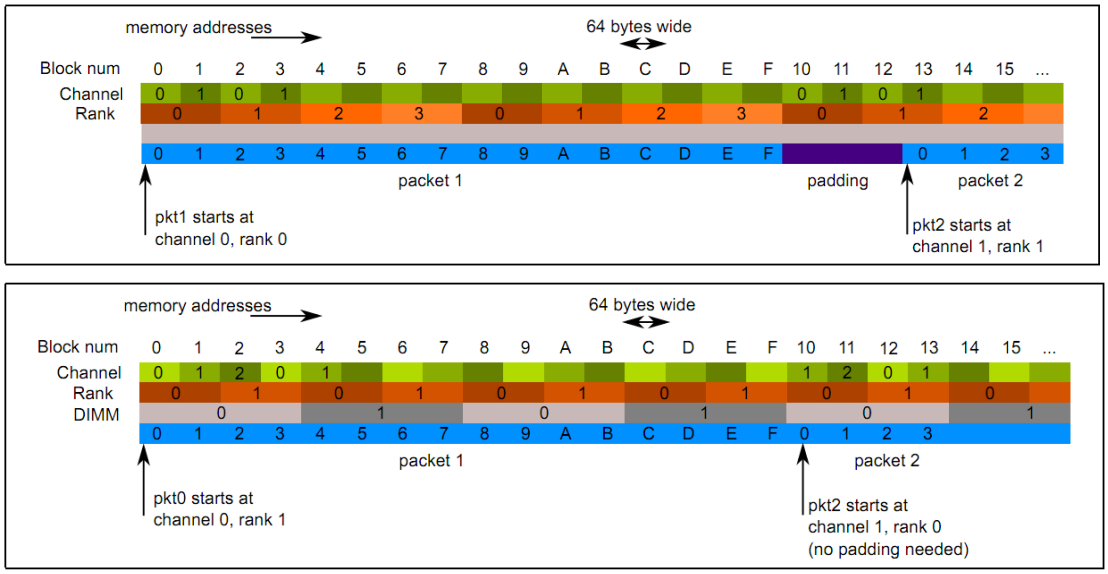
行地址及列地址可以交织出多层的存储单元，每一层我们叫做一个 block，而多个 block 就构成了完整的存储阵列，这个叫逻辑 bank。而将一组内存颗粒集合起来，拼接成处理器所需要的数据位宽，这样的结构就是物理的 BANK，也有叫 RANK，Channel 则指 CPU 所出的内存通道，每个内存通道可以挂接独立的 DDR DIMM,CPU 在访问 DDR 的时候,位于不同的通道的数据可以并发的进行访问，逻辑 BANK 用来减少总线冲突，物理的 BAND（RANK）也可以用来并发访问。

通过上面的介绍，我们应该知道，为了有更好的性能表现，最好是将对象平均的分布在同不的通道不同的 RANK 内，这样可以最大限度的提升内存的访问性能。Mempool 库所提供的内存对齐操作做的就是这方面的功作，它可以确保每个对象的起始地址位于存储器不同的 channel 和 rank。这种方式尤其适用于报文缓冲做三层转发或者是流分类，因为这类报文仅仅是头 64 个字节被访问，所以当报文分布在不同的 channel 的时候，CPU 可以并发的处理这些报文。

但是不同的 CPU 有着不同的通道数及 RANK 数，有些情况下，数据分布在不同的 rank 及 channel 的时候，为了保证数据的对齐读写，mempool 会将对象间进行特定数据的填充，以保证每个对象的起始地址是位于不同通道的不同 rank；而有些情况，则不需要进行填充，或者说是大多数情况下不需要填充，只有当对

象的大小超过 N*CHANNEL*64BYTES 后才需要填充。

下面是两副示意图，代表着不需要填充和需要填充的情况：



此功能可以通过用户在创建 mempool 的时候的入参来指定是否使用此功能。

6.3.2 CPU 本地 Cache

如果多个 core 都在访问一个 mempool 的 ring 里面的空闲 buffer 时，这样会导致 CPU 的占用率较高（为了避免竞争必须要用到锁，同时数据也没有被 cache 住，需要更远的内存拖取数据）。为了避免对于 mempool 的 ring 访问太过于频繁，mempool 库为每个 CPU 申请了一部分 cache 并且对 mempool ring 采取批量操作的方式。这样的话，只有 cache 满了才会需要对实际的 mempool ring 进行操作，这样就减少很多对 mempool ring 的锁操作，从而提高了 CPU 的效率。

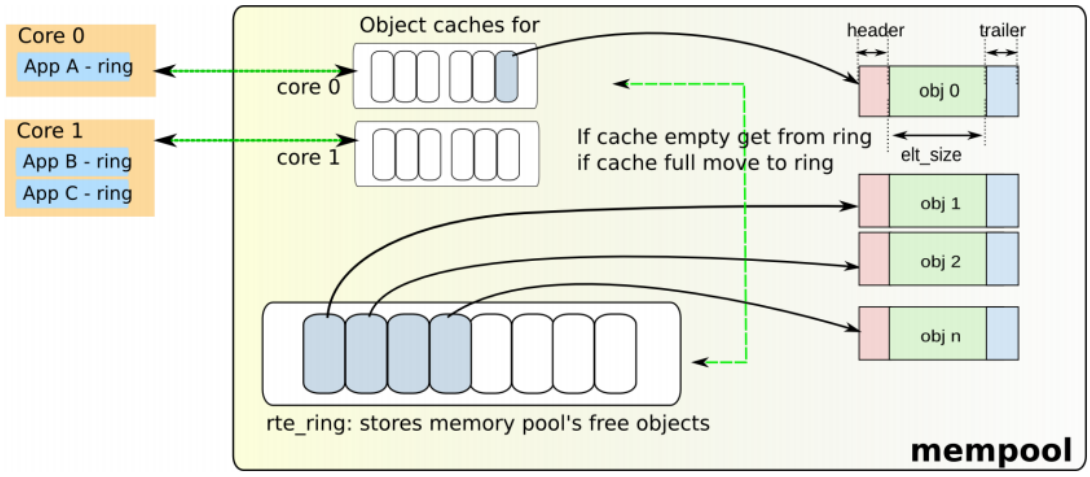
在这种处理方式下，每个 core 针对自己的空闲对象有着完整的访问权限，当 cache 满了这个时候 core 需要从 mempool ring 里面获取空闲的对象，或者当 cache 空了需要初始化对象时就会从 mempool ring 里面取数据。

当然这样处理意味着每个 core 可能都会有一些空闲的 buffer 没有被 core 使用，但是相对的，cache 命中及更少的锁也会带来更好的性能表现。同时，cache

是否开启也可以由创建 mempool 时的入参来决定，cache 的最大可用值（CONFIG_RTE_MEMPOOL_CACHE_MAX_SIZE）这个是编译的时候决定的，也可以在申请时指定，详见函数：rte_dom0_mempool_create 或者是 rte_mempool_create 的参数 cache_size，关于这个参数的定义如下：

```
 * @param cache_size
 *   If cache_size is non-zero, the rte_mempool library will try to
 *   limit the accesses to the common lockless pool, by maintaining a
 *   per-lcore object cache. This argument must be lower or equal to
 *   CONFIG_RTE_MEMPOOL_CACHE_MAX_SIZE. It is advised to choose
 *   cache_size to have "n modulo cache_size == 0": if this is
 *   not the case, some elements will always stay in the pool and will
 *   never be used. The access to the per-lcore table is of course
 *   faster than the multi-producer/consumer pool. The cache can be
 *   disabled if the cache_size argument is set to 0; it can be useful
to
 *   avoid losing objects in cache. Note that even if not used, the
 *   memory space for cache is always reserved in a mempool structure,
 *   except if CONFIG_RTE_MEMPOOL_CACHE_MAX_SIZE is set to 0.
```

下图显示的是 cache 使用的示意图：



6.4 Mbuf 函数库

Mbuf library 用于向基于 DPDK 的 APP 提供消息缓冲申请/释放的接口，所有的 mbuf 均存储于 mempool。目前每个一 rte_mbuf 结构可以通过消息类型来定指作为特定的用途，如网络报文使用 RTE_MBUF_PKT，通用的控制消息使用 RTE_MBUF_CTRL，这个消息内容可以灵活的扩充，同时 rte_mbuf 结构体也会尽量保持精得，同时也是 cache line 对齐的。

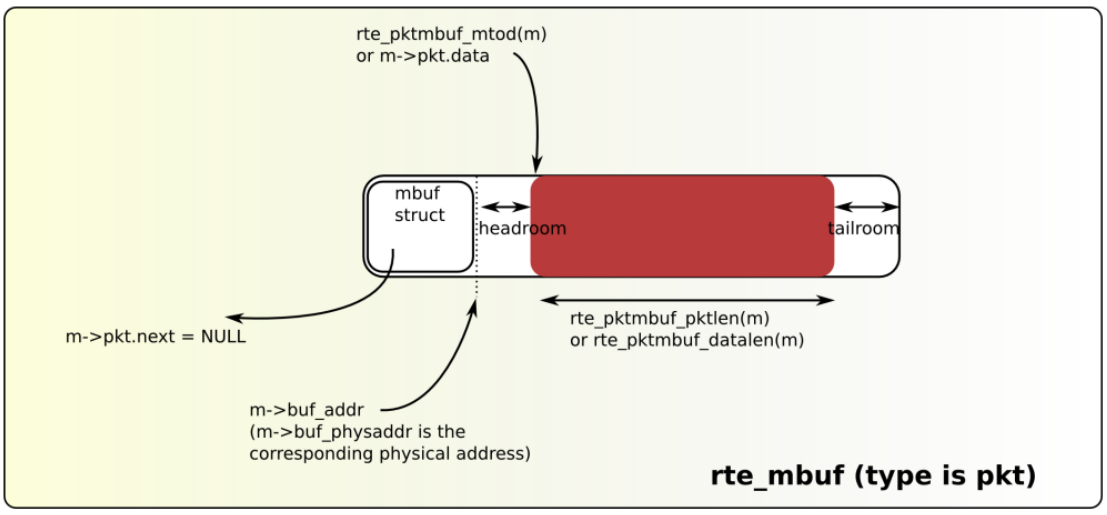
通常而言对于一个消息报文的数据（包括协议头），有两种做法：

1、在一个单一存储空间集成元数据结构及相邻的固定大小的报文数据

2、分别为元数据结构及报文数据使用独立的存储空间

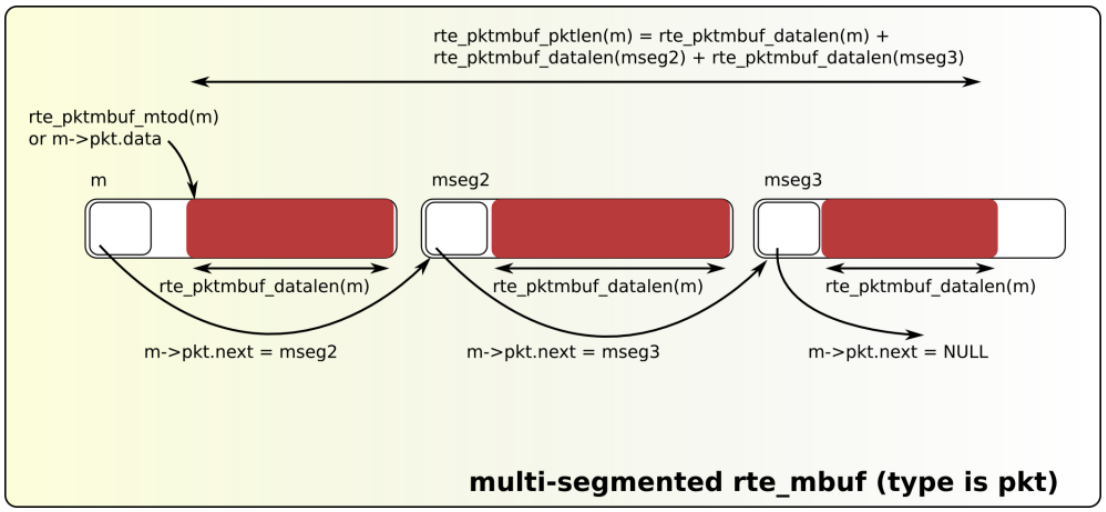
第一种方法的好处是申请/释放整个报文的内存空间只需要一个操作，但从另一个角度来说，就没有第二种法方灵活。DPDK 选用的是第一种方式，元数据包括一些控制信息如消息类型、长度、数据的指针、以及下一个 mbuf 的结构指针（用于消息链表，对于消息类型为 RTE_MBUF_PKT 的网络消息的 jumbo 帧就会于消息链表将这些组织起来->pkt.next）。

下图是一个 mbuf 结构的示意图：



上图中的 headroom 是指用于做 cache aglin 时预留的 128bytes 数据，真正的数

据存储从 buf_addr+ RTE_PKTMBUF_HEADROOM 开始算起。如果有多个 pkt 消息组成链表，则如下图所示：



6.5 DPDK 内存对象分布

前面章节描述的四个库，均是基于内存来做一些动作的，那么，这些内存是从哪里来的呢？linux 里面有个技术叫 hugepage，在前面的 DPDK 简介里面稍微提到了一下，linux 普通的一个 page 的大小是 4K，同时还存在换出的问题（物理内存不够用了，会由虚拟内存管理机制替换到 swap 分区）。而 hugepage 相对于普通的 page(4K)来说有以下几个个特点：

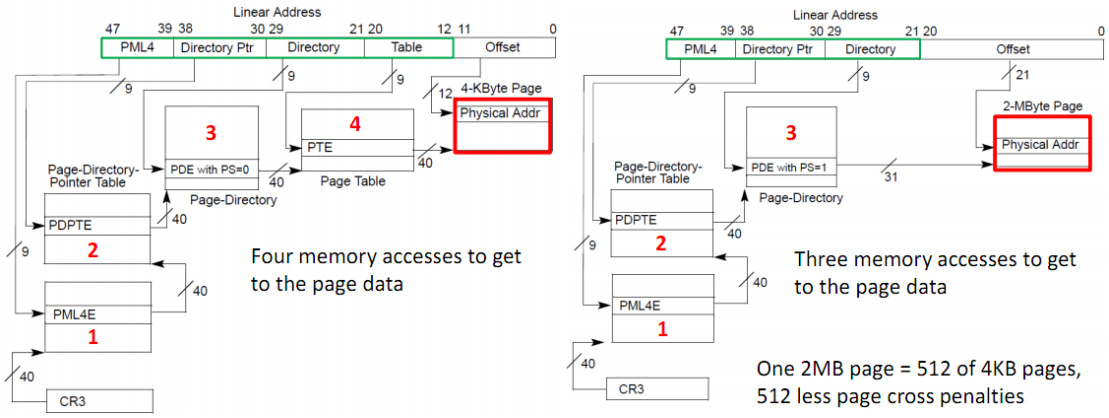
1. hugepage 页面不受虚拟内存管理影响，不会被替换出内存，而普通的 4kpage 有换出问题。
2. 同样的内存大小，hugepage 产生的页表项数目远少于 4kpage。如：用户进程需要使用 4M 大小的内存，若采用 4Kpage，需要 1K 的页表项存放虚拟地址到物理地址的映射关系，而采用 hugepag 则只产生 2 条页表项。这样会带来两个好处：

   - 一是使用 hugepage 的内存产生的页表比较少，这对于数据库系统等动不动就需要映射非常大的数据到进程的应用来说，可以有效减少页表的开销，所以很多数据库系统都采用 hugepage 技术。

- 二是 TLB 冲突率会大大减少，TLB 驻留在 cpu 的 1 级 cache 里，是芯片访问最快的缓存，一般只能容纳 100 多条页表项，如果采用 hugepage，则可以极大减少 TLB miss 导致的开销（以 64 位为例）：

  - TLB 命中，立即就获取到物理地址，这个命中率会比 4Kpage 要高很多。
  - TLB 不命中，传统 4Kpage 需要查 CR3->PML4E->PDPTE->PDE->PTE->物理内存，如果这页框被虚拟内存系统替换到交互区，则还需要交互区 load 回内存，而 hugepage 则可以忽略这一步。总之，TLB miss 是性能大杀手，而采用 hugepage 可以有效降低 tlb miss。

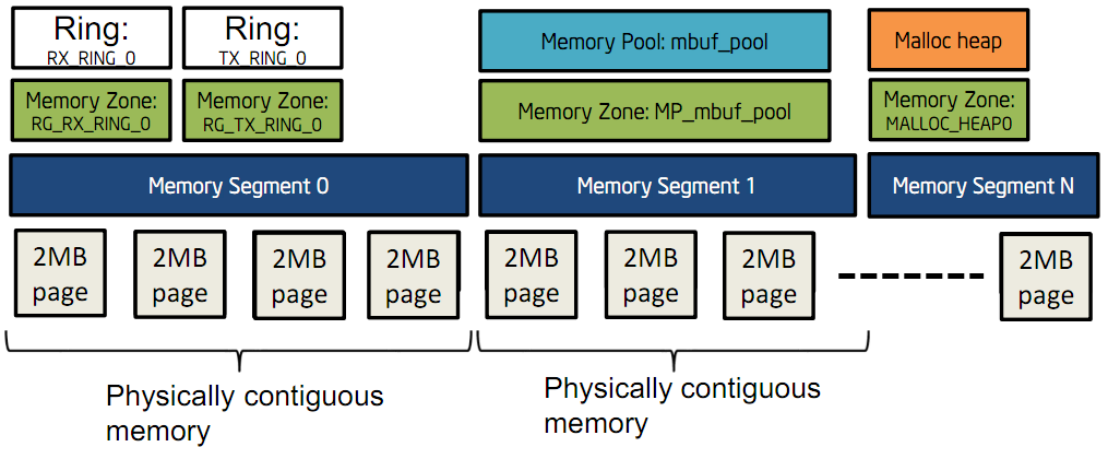
进一步解释如下：关于 TLB 的，CR3 是 page-translation tables 的基址，X86 是 32 位，X64 下扩展为 64 位，而 page-translation tables 里面的数据结构就是页的集合，不同的 CPU 支持的页大小不一样，如 AMD 实现 4 种尺寸的页：4K page、2M page、4M page 以及 1G page，而 Intel 支持 1G page、4K page、2M page 以及 4M page，不同的页面大小决定了不同的 page_tables 的结构。在 X86 32 位下，线性地址进行 paging 时，其 page-translation tables 经过 2 级寻址，分别寻址 page tabls 以及 pages，这个会经过 PDT 和 PT，在 X86 64 位下，线性地址进行 paging 时，其 page-translation tables 经过 3 级寻址，分别寻址 page-directory pointer tables、page-directory tables、page tables 以及 pages 这个过程会涉及 PML4T（page map level4 tables）、PDPT（page directory pointer tables）、PDT（page directory tables）以及 PT（page tables）这四个数据结构体。详情请见下面的示意图：



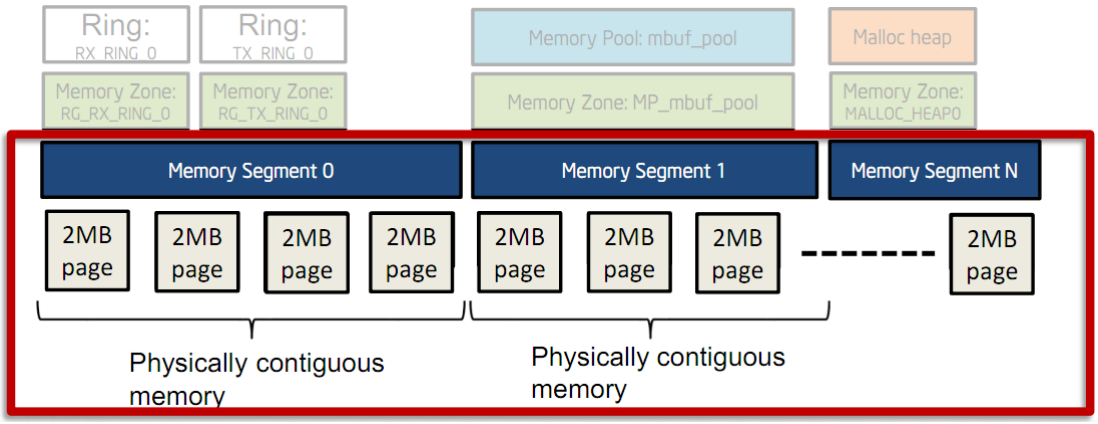
同时，在 linux 中，使用 hugepage 的方式也很简单，以使用 2M 的 hugepage 为例：

1. 进入/sys/kernel/mm/hugepages/hugepages-2048kB/目录，通过修改这个目录下的文件可以修 hugepage 页面的大小和总数目。
2. linux 将 hugepage 实现为一种文件系统 hugetlbfs，需要将该文件系统 mount 到某个路径，如：mount -t hugetlbfs nodev /mnt/huge。
3. 在用户进程里通过 mmap 映射 hugetlbfs mount 的目标文件，这个 mmap 返回的地址就是大页面的了，如：mmap /mnt/huge。

说了这么多 hugepage 的好处，DPDK 实际上也是采用的 hugepage 来给上面章节讲的四个库来提供内存的。下面一个图可以直观的显示出 DPDK 内存对象的分布情况：

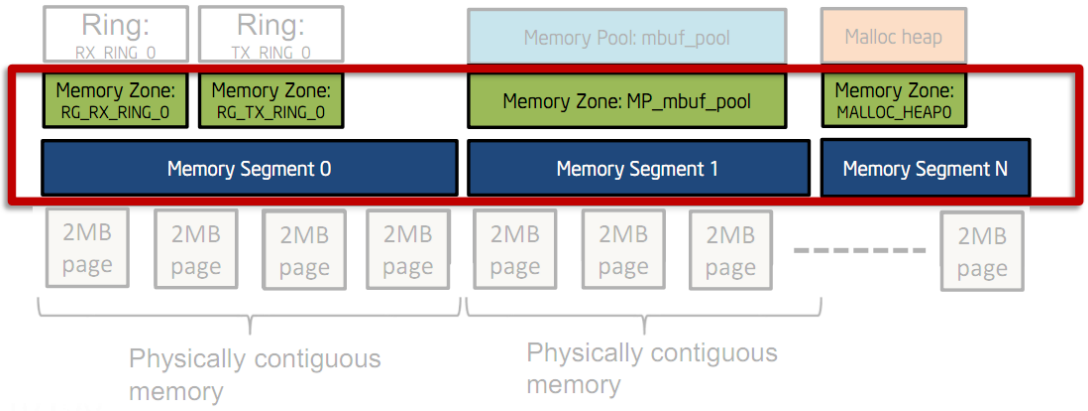


上图中，所有的 2M page 均是通过 hugepage 提供的内存。从代码上来看，所有的 2M page 均有结构体 struct hupepage 来表示，在此结构之上，为了把所有物理地址连续的 2M page 组合在一起，又有一个结构体 struct rte_memseg 来表示，同时，为了表示在一个 memory segment 里面被分配出去的内存，又有一个结构来表示已划分出去的内存区域，struct rte_memzone，memory segment 和 memory zone 的区别可以见下面两张图：



上图表示 memory segments 的示意图，特点如下：

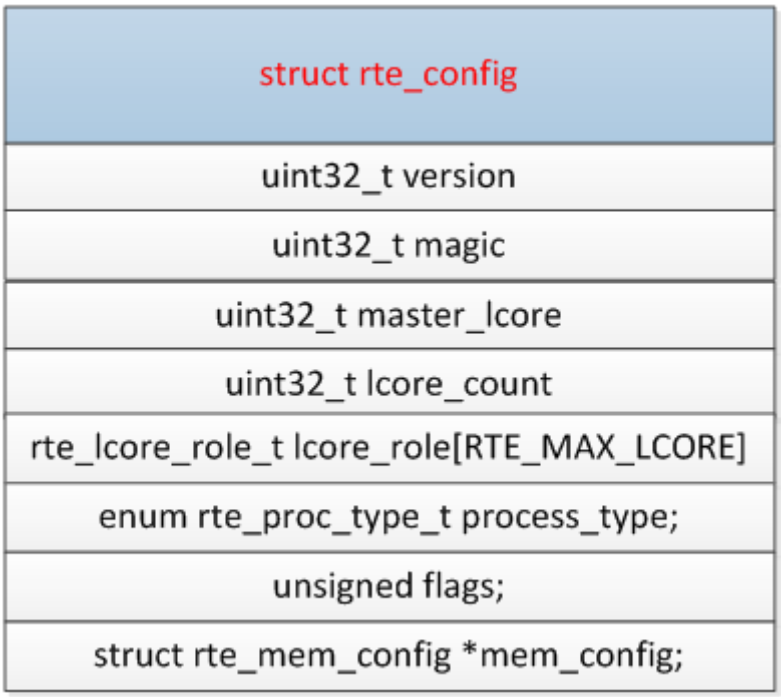
- 所有内存管理的内部单元均为 memory segment（申请、释放、删除等）。
- 只支持基于 HUGEPAGE（2M/1G PAGE）的内存
- 每一段 memory segment 均是由连续的物理地址及虚拟地址的 2M page 组成
- 某些对象可以将一个 memory segment 分解成更小的 memory zones.



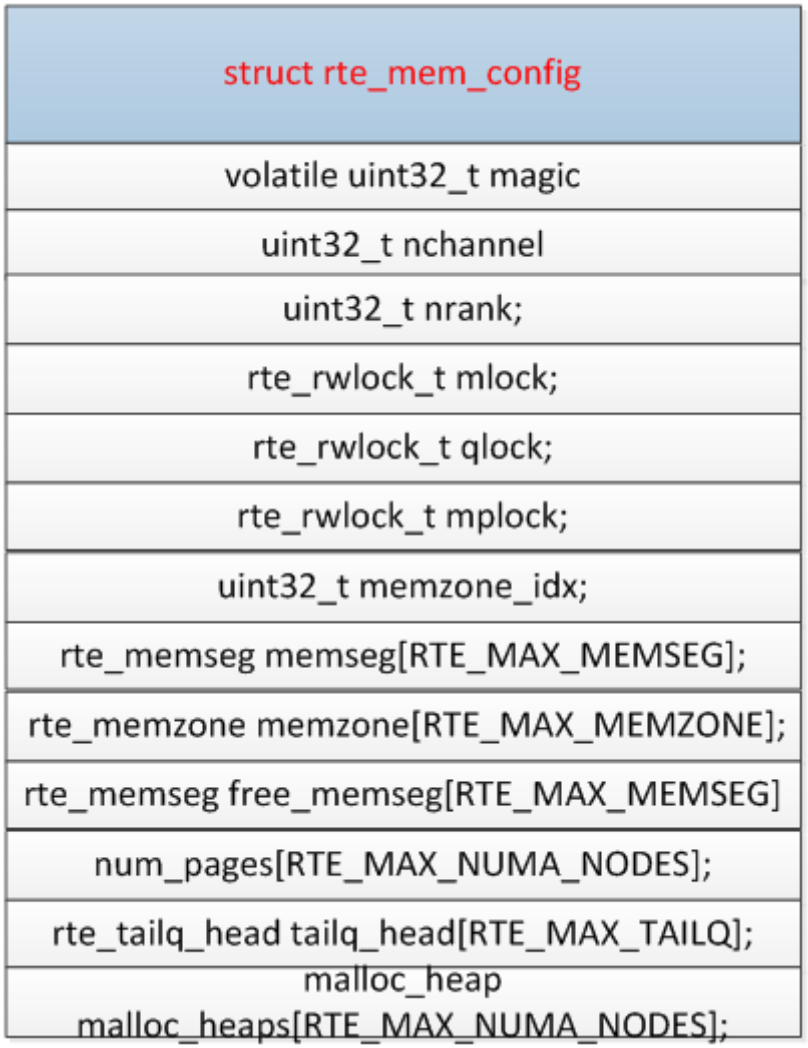
上图表示 memory zones 的示意图，特点如下：

- 每一个 memory zone 均是内存分配的最小基本单元(会通过 name 进行命令)
- 只有 reserver 接口，没有释放接口（因为本来就不涉及到内存的申请，只不过是占用）
- 由于应用时有连续物理地址的要求，所以并不能跨 memory segment
- 调用者可以看到分配的 memory zone 所代表的块的物理地址。

我们再看看 DPDK 内存管理的结构体，下面是一个内存全局配置结构体：



在上面这个结构体中,有个 struct ret_mem_config 结构体,成员是 mem_config,这个是各个 DPDK 程序共享的内存配置结果，被 mmap 到/var/run/.rte_config,这也是实现多进程共享 DPDK 内存及实现多进程处理的基础，这点稍后再说。我们先进入 mem_config 看看：



这个结构里面即有我们前面讲的 memseg 及 memzone。

7、DPDK Poll 模型驱动

DPD 在其发行包内包括了一些网络设备的用户态驱动，这些驱动均以库的形式存在，同时由于这些用户态驱动均是使用的 polling 模式来替代原有的中断处理机制，所以被称之为 Poll Mode Driver，后面我们简称 PMD。同时，DPDK 为了更好的管理这些不同的 PMDs，它在驱动之上抽象了一层网络层，提供统一的网络设备注册、发送、接收及配置等操作接口供 APP 直接使用，这一层也是以库的方式提供的（librte_ether）。

目前 DPDK 所支持的网卡 PMDs 如下：

Cisco

enic (UCS Virtual Interface Card)

Emulex

oce (OneConnect OCe14000 family)

Intel

e1000 (82540, 82545, 82546)

e1000e (82571..82574, 82583, ICH8..ICH10, PCH..PCH2)

igb (82575..82576, 82580, I210, I211, I350, I354, DH89xx)

ixgbe (82598..82599, X540, X550)

i40e (X710, XL710)

Mellanox

mlx4 (ConnectX-3, ConnectX-3 Pro)

Paravirtualization

virtio-net or virtio-net + uio (QEMU)

xenvirt (Xen)

vmxnet3 or vmxnet3 + uio (VMware ESXi)

memnic

在 NUMA 架构下面，每个 socket 上的 CPU 去访问其它 CPU 的内存控制器下的资源时，其效率会比访问本低的慢 10 倍以上，为了解决 NUMA 架构下的这个问题，就要保证每一个逻辑 core 尽可能的访问它本地的内存，为了这个目的，PMDs 在设计的时候就需要考虑如下的方案：

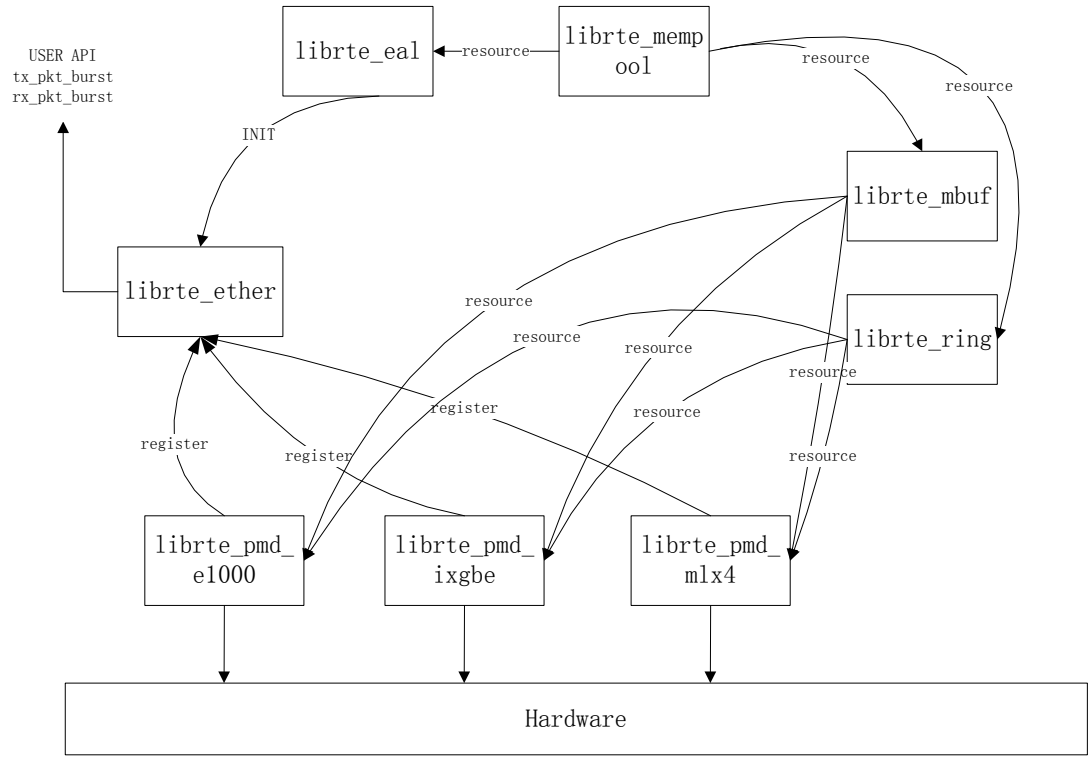
- 将每一个 NIC 的接收队列放到一个逻辑 core 上去，让所有的报文接收过及处理过程在这个逻辑 core 上以 run-to-complete 的模型完成。
- 在 NIC 的接收队列所属的 lcore 的本地内存所属的 mempool 申请 ring 描述符。
- 每个 lcore 申请自己独有的发送队列，并且在本地内存所属的 mempoll

申请发送队列 ring 描述符。

当如，如果系统内的所有的逻辑 core 的数据要比 NIC 硬件的最大发送或是接收队列还要多的话，可以采用一些其它的处理措施，如特定的几个逻辑 core 用来处理发送或是接收队列，但有可能的话，一定要保证所有的发送队列是平均的分布在所有的逻辑 core 上面，这样可以避免某个 lcore 的瓶颈造成的性能低下。

另外，由于多核间是会共享发送队列的，这样的话对于这些队列的访问的互斥就显得尤为重要，相对于添加 lock 操作，最好的方式是在 PMD 的传输功能里面增加无锁的操作功能或者是在传输前给出明确的信号，这个是 PMDs 在开发时需要注意的。

下面说明 DPDK 中 PMD 的整个调用框架，在此之前我们先看一个图：



针对网络设备的注册，提供了rte_eth_driver_register，用户态驱动只要输入eth_driver的结构体，剩下的初始化操作可以全部交给EAL，它会调用

rte_eal_pci_probe来完成新增的PCIe设备用户态驱动的加载。PMD只用完成下面的结构体的初始化：

```
static struct eth_driver rte_igb_pmd = {
      {
            .name = "rte_igb_pmd",
            .id_table = pci_id_igb_map,
            .drv_flags = RTE_PCI_DRV_NEED_MAPPING |
RTE_PCI_DRV_INTR_LSC,
      },
      .eth_dev_init = eth_igb_dev_init,
      .dev_private_size = sizeof(struct e1000_adapter),
};
```

eth_igb_dev_init函数主要是用来初始化实际的硬件设备，同时给rte_eth_dev设备的相关成员赋值。而rte_eth_dev则是由librte_ether提供的一个抽象化的网络设备结构体，PMDs还需要填充实际网卡设备的结构体：

```
struct rte_eth_dev {
      eth_rx_burst_t rx_pkt_burst; /**< Pointer to PMD receive function.
*/
     eth_tx_burst_t tx_pkt_burst; /**< Pointer to PMD transmit function.
*/
      struct rte_eth_dev_data *data;  /**< Pointer to device data */
      const struct eth_driver *driver;/**< Driver for this device */
      struct eth_dev_ops *dev_ops;     /**< Functions exported by PMD */
      struct rte_pci_device *pci_dev; /**< PCI info. supplied by probing
*/
      struct rte_eth_dev_cb_list callbacks; /**< User application
callbacks */
```

};

其中rx_pkt_burst就指向pmd(poll-mode driver)提供的接收函数，tx_pkt_burst指向pmd的发送函数，下面几个结构体分别都是抽像出来的数据以及设备结构，同时还会有pci设备的结构体用于probe，而在probe的过程中，需要使用到具体的网卡硬件配置的接口，这个时候DPDK EAL提供了librte_ether里面的结构体eth_dev_ops来完成各种功能

```
struct eth_dev_ops {
	eth_dev_configure_t		dev_configure; /**< Configure device.
*/
	eth_dev_start_t			dev_start;	/**< Start device. */
	eth_dev_stop_t			dev_stop;	/**< Stop device. */
	eth_dev_set_link_up_t		dev_set_link_up;	/**< Device link
up. */
	eth_dev_set_link_down_t	dev_set_link_down; /**< Device link
down. */
	eth_dev_close_t			dev_close;	/**< Close device. */
	eth_promiscuous_enable_t	promiscuous_enable; /**< Promiscuous
ON. */
	eth_promiscuous_disable_t	promiscuous_disable;/**< Promiscuous
OFF. */
	eth_allmulticast_enable_t	allmulticast_enable;/**< RX multicast
ON. */
	eth_allmulticast_disable_t allmulticast_disable;/**< RX
multicast OF. */
	eth_link_update_t		link_update;	/**< Get device link
state. */
	eth_stats_get_t			stats_get;	/**< Get device
```

```
statistics. */
      eth_stats_reset_t          stats_reset;   /**< Reset device
statistics. */
      eth_queue_stats_mapping_set_t queue_stats_mapping_set;
      /**< Configure per queue stat counter mapping. */
      eth_dev_infos_get_t        dev_infos_get; /**< Get device info.
*/
      mtu_set_t                  mtu_set; /**< Set MTU. */
      vlan_filter_set_t          vlan_filter_set;  /**< Filter VLAN
Setup. */
      vlan_tpid_set_t            vlan_tpid_set;      /**< Outer VLAN
TPID Setup. */
      vlan_strip_queue_set_t     vlan_strip_queue_set; /**< VLAN
Stripping on queue. */
      vlan_offload_set_t         vlan_offload_set; /**< Set VLAN
Offload. */
      vlan_pvid_set_t            vlan_pvid_set; /**< Set port based TX
VLAN insertion */
      eth_queue_start_t          rx_queue_start;/**< Start RX for a
queue.*/
      eth_queue_stop_t           rx_queue_stop;/**< Stop RX for a
queue.*/
      eth_queue_start_t          tx_queue_start;/**< Start TX for a
queue.*/
      eth_queue_stop_t           tx_queue_stop;/**< Stop TX for a
queue.*/
      eth_rx_queue_setup_t       rx_queue_setup;/**< Set up device RX
queue.*/
      eth_queue_release_t        rx_queue_release;/**< Release RX
```

```
queue.*/
      eth_rx_queue_count_t       rx_queue_count; /**< Get Rx queue
count. */
      eth_rx_descriptor_done_t   rx_descriptor_done;  /**< Check rxd
DD bit */
      eth_tx_queue_setup_t       tx_queue_setup;/**< Set up device TX
queue.*/
      eth_queue_release_t        tx_queue_release;/**< Release TX
queue.*/
      eth_dev_led_on_t           dev_led_on;    /**< Turn on LED. */
      eth_dev_led_off_t          dev_led_off;   /**< Turn off LED. */
      flow_ctrl_get_t            flow_ctrl_get; /**< Get flow control.
*/
     flow_ctrl_set_t            flow_ctrl_set; /**< Setup flow control.
*/
      priority_flow_ctrl_set_t   priority_flow_ctrl_set; /**< Setup
priority flow control.*/
      eth_mac_addr_remove_t      mac_addr_remove; /**< Remove MAC
address */
      eth_mac_addr_add_t         mac_addr_add;  /**< Add a MAC address
*/
      eth_uc_hash_table_set_t    uc_hash_table_set;  /**< Set Unicast
Table Array */
……/*省略*/
}
```

而每一个网络设备的用户态驱动，需要按照其实现的填充上面的相关功能。对于每一个网络设备的私有数据，会有另外一个单独的结构体进行管理，同时支持多核间的配置及读取rte_eth_dev_data。

MELLANOX提供了ROCE的用户态驱动给APP调用，它是通过一个PMD的driver库（librte_pmd_mlx4）来实现的用户态驱动,用户态驱动使用时，需要执行rte_eal_pci_probe,这个会调用到mlx4_pci_devinit-->rte_eth_dev_allocate,然后申请出rte_eth的结构体.

mlx4的初始化：

eth_driver->devinit->mlx4_pci_devinit->mlx4_dev_ops->mlx4_rx_queue_setup/mlx4_tx_queue_setup->mlx4_tx_burst/mlx4_rx_burst_sp

rx/tx queue的申请：

eth_dev_ops->dev_configure->mlx4_dev_configure->rte_eth_dev->data->rx/tx_queue

rx queue的初始化：

eth_driver->devinit->mlx4_pci_devinit->mlx4_dev_ops->mlx4_rx_queue_setup

tx queue的初始化：

eth_driver->devinit->mlx4_pci_devinit->mlx4_dev_ops->mlx4_tx_queue_setup

从上面调用关系可以看出，如果我们基于 DPDK 来实现用户态的 PMD，只需要按其框架填充关键的功能特性就可以，intel 的整个框架实际上也是借鉴了内核标准 PCIe 设备驱动的思路，早已经准备好整体框架，所有的用户态驱动的功能就是通过注册，函数指针传递的方式来搞定。

8、DPDK 多进程分析

DPDK 使用了 pthread 的库，所以 DPDK 是天然的支持多线程的，但是多进程的支持程序呢？对比多进程与多线程，两种性能的差别呢？ 在介绍这些之前，我们先来了解一些基本本的知识。

8.1 进程的创建

Linux的线程实现是在核外进行的,核内提供的是创建进程的接口do_fork()。内核提供了两个系统调用__clone()和fork()，最终都用不同的参数调用do_fork()核内API。 do_fork() 提供了很多参数，包括CLONE_VM（共享内存空间）、CLONE_FS（共享文件系统信息）、CLONE_FILES（共享文件描述符表）、CLONE_SIGHAND（共享信号句柄表）和CLONE_PID（共享进程ID，仅对核内进程，即0号进程有效）。当使用fork系统调用产生多进程时，内核调用do_fork()不使用任何共享属性，进程拥有独立的运行环境。当使用pthread_create()来创建线程时，则最终设置了所有这些属性来调用__clone()，而这些参数又全部传给核内的do_fork()，从而创建的”进程”拥有共享的运行环境，只有栈是独立的，由 __clone()传入。

即：Linux下**不管是多线程编程还是多进程编程，最终都是用do_fork实现**的多进程编程，只是进程创建时的参数不同，从而导致有不同的共享环境。Linux线程在核内是以轻量级进程的形式存在的，拥有独立的进程表项，而所有的创建、同步、删除等操作都在核外pthread库中进行。pthread 库使用一个管理线程（__pthread_manager() ，每个进程独立且唯一）来管理线程的创建和终止，为线程分配线程ID，发送线程相关的信号，而主线程pthread_create()） 的调用者则通过管道将请求信息传给管理线程。我们可以总结如下：

创建进程= fork -> do_fork(不使用共享属性)

创建线程= pthread_create -> __clone -> do_fork(共享地址空间（代码区、数据区）、页表、文件描述符、信号…)

另外，对于一个进程来说必须有的数据段、代码段、堆栈段是不是全盘复制呢？对于多进程来说，代码段是不用复制的，因为父进程和各子进程的代码段是相同的。数据段和堆栈段则不一定，因为在Linux里广泛使用的一个技术叫

copy-on-write， 即写时拷贝。copy-on-write 意味节省资源，假设有一个变量 x 在父进程里存在，当这个父进程创建一个子进程或多个子进程时，子进程和父进程使用同一个内存空间的变量,但当子进程或父进程要改变变量 x 的值时就会复制该变量，从而导致父子进程里的变量值不同。

8.2 调度与切换

由于核内的线程本质就是进程，其调度过程跟进程一样。切换，不论是进程切换还是线程切换，都需要替换运行环境（内核堆栈，运行时寄存器等），对于内存的切换，内核部分内存是一样的，用户空间部分：如果是进程，需要替换页目录基址寄存器，如果是线程，不需要替换；总体而言，linux 进程和线程的切换,从内存寄存器、内核堆栈寄存器、其他寄存器等的换值开销应该是差不多的。但是由于多线程共享地址空间,从一个线程切换到同一个进程上另一个线程运行,页表,数据区等很多都已经在内存甚至缓存里,而从一个进程切换到另一个进程,可能由于刚切换进来的进程的页面被虚拟内存管理模块替换出去导致的页面替换开销，另外还有缓存 tlb 失效导致的缓存更新开销，这里性能有所差别。

8.3 地址空间共享

进程地址空间是独立的，这意味着，不同进程的内存天生就是不共享的，如果要共享，则需要开发者自己构建共享机制，比如使用 IPC。

线程地址空间是共享的,这意味着,同一进程不同线程的内存天生是共享的,如果想要不共享，需要开发者自己实施，比如使用线程本地变量。

上述的差别带来两个问题：

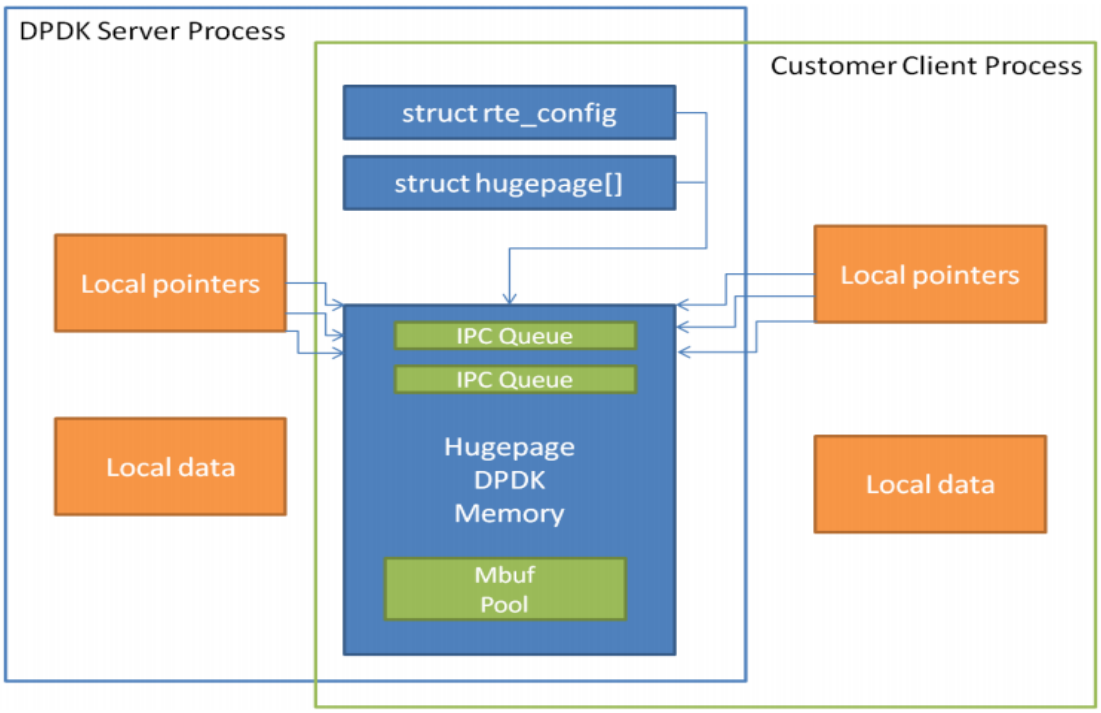
1、安全性：从地址共享来说，进程是比线程要更安全、更健壮、更容易开

发

2、性能上：由于进程间共享内存是需要进程间通信的，因此会带来额外的开销，对于传统的应用来说，自然性能会稍微低一下。

综合以上三个过程：创建、调度与切换、地址空间共享，我们可以看出，在传统的进程与线程模型中，可能在性能上线程会稍微有一些优势，但在 DPDK 环境下呢？我们知道 DPDK 提供了 share memory pools，lock free rings 等特性，同时还有读写锁、原子操作等。

更进一步的，DPDK 由于是基于 HUGEPAGE 进行内存管理的，它通过两个文件描述了所有的内存使用状态 /var/run/dpdk_config 及 /var/run/dpdk_hugepage_map，前者是内存全局表，后者是 hugepage 映射表，这样地址空间共享的问题就不会存在了，同时由于 hugepage 不存在换出的问题，在调度与切换的时候不会产生太多的 TLB MISS，这样就比较好的支持了多进程的处理机制，同时也不会因为 DPDK 带来性能上的损失，APP 唯一要做的就是合理的分发其任务到不同的进程或是不同的线程。DPDK 的多进程内存共享方式如下图所示意：



另外 DPDK 的机制还能够保证其所有的 hugepage memory 在不同的进程里面均为相同的线性地址，对于进程间的通信来说，一旦传递地址，不需要复杂的地址转换过程，拿到了地址就可以在本地直接使用，这也是一个提升处理速度的方式。

9、DPDK 技术总结

DPDK 是一个开源的数据平面开发工具集，提供了一个用户空间下的高效数据包处理库函数，它通过环境抽象层旁路内核协议栈、轮询模式的报文无中断收发、优化内存/缓冲区/队列管理、基于网卡多队列和流识别的负载均衡等多项技术，实现了在 x86 处理器架构下的高性能报文转发能力，用户可以在 Linux 用户态空间开发各类高速转发应用，也适合与各类商业化的数据平面加速解决方案进行集成。

DPDK 的上层用户态由很多库组成，主要包括核心部件库（Core Libraries）、平台相关模块(Platform)、网卡轮询模式驱动模块（PMD-Natives&Virtual）、QoS 库、报文转发分类算法（Classify）等几大类，用户应用程序可以使用这些库进行二次开发，下面分别简要介绍。

核心部件库：该模块构成的运行环境是建立在 Linux 上，通过环境抽象层(EAL)的运行环境进行初始化，包括：HugePage 内存分配、内存/缓冲区/队列分配与无锁操作、CPU 亲和性绑定等；其次，EAL 实现了对操作系统内核与底层网卡 I/O 操作的屏蔽（I/O 旁路了内核及其协议栈），为 DPDK 应用程序提供了一组调用接口，通过 UIO 或 VFIO 技术将 PCI 设备地址映射到用户空间，方便了应用程序调用，避免了网络协议栈和内核切换造成的处理延迟。另外，核心部件还包括创建适合报文处理的内存池、缓冲区分配管理、内存拷贝、以及定时器、环形缓冲区管理等。

平台相关模块：其内部模块主要包括 KNI、能耗管理以及 IVSHMEM 接口。其中，KNI 模块主要通过 kni.ko 模块将数据报文从用户态传递给内核态协议栈处理，以便用户进程使用传统的 socket 接口对相关报文进行处理；能耗管理则提供了一些 API，应用程序可以根据收包速率动态调整处理器频率或进入处理器的不同休眠状态；另外，IVSHMEM 模块提供了虚拟机与虚拟机之间，或者虚拟机与主机之间的零拷贝共享内存机制，当 DPDK 程序运行时，IVSHMEM 模块会调用核心部件库 API，把几个 HugePage 映射为一个 IVSHMEM 设备池，并通过参数传递给 QEMU，这样，就实现了虚拟机之间的零拷贝内存共享。

轮询模式驱动模块：PMD 相关 API 实现了在轮询方式下进行网卡报文收发，避免了常规报文处理方法中因采用中断方式造成的响应延迟，极大提升了网卡收发性能。此外，该模块还同时支持物理和虚拟化两种网络接口，从仅仅支持 Intel 网卡，发展到支持 Cisco、Broadcom、Mellanox、Chelsio 等整个行业生态系统，以及基于 KVM、VMWARE、 XEN 等虚拟化网络接口的支持。

DPDK 还定义了大量 API 来抽象数据平面的转发应用，如 ACL、QoS、流分类和负载均衡等。并且，除以太网接口外，DPDK 还在定义用于加解密的软硬件加速接口（Extensions）。

处理器的内存管理包含两个概念：物理内存和虚拟内存。Linux 操作系统里面整个物理内存按帧（frames）来进行管理，虚拟内存按照页（page）来进行管理。内存管理单元（MMU）完成从虚拟内存地址到物理内存地址的转换。内存管理单元进行地址转换需要的信息保存在一个叫页表（page table）的数据结构里面，页表查找是一种极其耗时的操作。为了减少页表的查找过程，Intel 处理器实现了一块缓存来保存查找结果，这块缓存被称为 TLB（Translation Lookaside Buffer），它保存了虚拟地址到物理地址的映射关系。所有虚拟地址在转换为物理地址以前，处理器会首先在 TLB 中查找是否已经存在有效的映射关系，如果没有发现有效的映射，也就是 TLS miss，处理器再进行页表的查找。页表的查找过程对性能影响极大，因此需要尽量减少 TLB miss 的发生。

x86 处理器硬件在缺省配置下，页的大小是 4K，但也可以支持更大的页表尺寸，例如 2M 或 1G 的页表。使用了大页表功能后，一个 TLB 表项可以指向更大的内存区域，这样可以大幅减少 TLB miss 的发生。早期的 Linux 并没有利用 x86 硬件提供的大页表功能，仅在 Linux 内核 2.6.33 以后的版本，应用软件才可以使用大页表功能，具体的介绍可以参见 Linux 的大页表文件系统（hugetlbfs）特性。

DPDK 则利用大页技术，所有的内存都是从 HugePage 里分配，实现对内存池(mempool)的管理，并预先分配好同样大小的 mbuf，供每一个数据包使用。

传统网卡的报文接收/发送过程中，网卡硬件收到网络报文，或发送完网络报文后，需要发送中断到CPU，通知应用软件有网络报文需要处理。在x86处理器上，一次中断处理需要将处理器的状态寄存器保存到堆栈，并运行中断服务程序，最后再将保存的状态寄存器信息从堆栈中恢复。整个过程需要至少300个处理器时钟周期。对于高性能网络处理应用，频繁的中断处理开销极大降低了网络应用程序的性能。

为了减少中断处理开销，DPDK 使用了轮询技术来处理网络报文。网卡收到报文后，直接将报文保存到处理器 cache 中（有 DDIO（Direct Data I/O）技术的情况下），或者保存到内存中（没有 DDIO 技术的情况下），并设置报文到达的标志位。应用软件则周期性地轮询报文到达的标志位，检测是否有新报文需要处理。整个过程中完全没有中断处理过程，因此应用程序的网络报文处理能力得以极大提升。

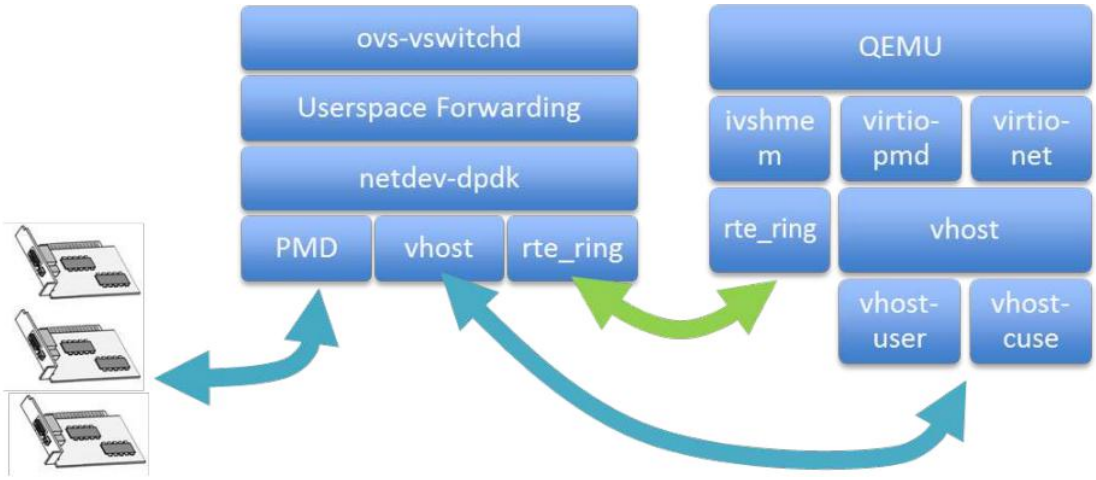
现代操作系统都是基于分时调用方式来实现任务调度，多个进程或线程在多核处理器的某一个核上不断地交替执行。每次切换过程，都需要将处理器的状态寄存器保存在堆栈中，并恢复当前进程的状态信息，这对系统其实是一种处理开

销。将一个线程固定一个核上运行，可以消除切换带来的额外开销。另外将进程或者线程迁移到多核处理器的其它核上进行运行时，处理器缓存中的数据也需要进行清除，导致处理器缓存的利用效果降低。

CPU亲和技术，就是将某个进程或者线程绑定到特定的一个或者多个核上执行，而不被迁移到其它核上运行，这样就保证了专用程序的性能。

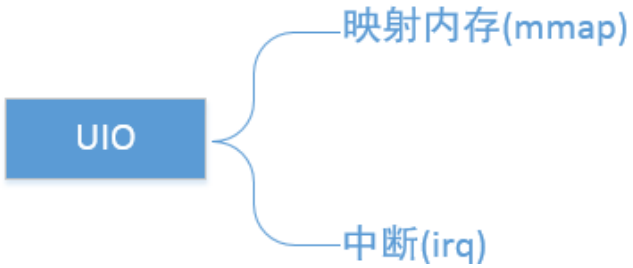
DPDK 使用了 Linux pthread 库，在系统中把相应的线程和 CPU 进行亲和性绑定，然后相应的线程尽可能使用独立的资源进行相关的数据处理。

10、DPDK 和 VOS 的关系



图：OVS 的用户接口

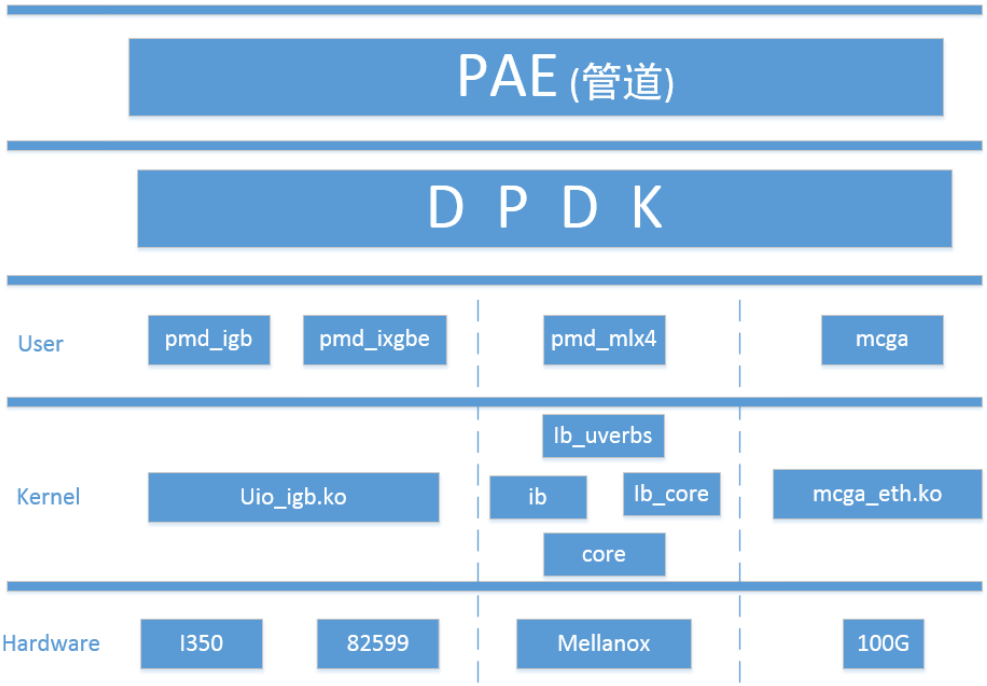
1、UIO：用户态IO，主要作用是将内核BAR空间映射到用户空间中。



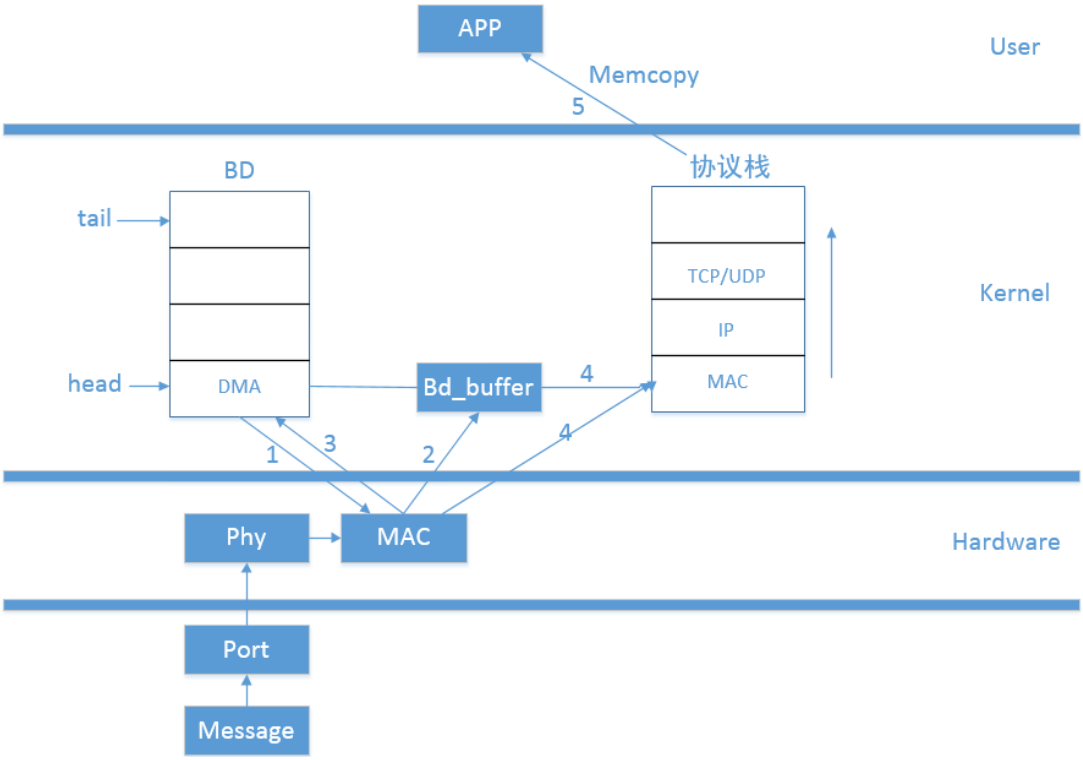
对 DPDk 管理的 PCI 设备而言

由于其配置空间在 kernel 就已经搞定，而 IO 空间 DPDK 又用不到，所以最主要的就是将 BAR 空间从内核映射到用户空间中，而该操作就是由 UIO 来完成的。

2、我们目前用到的网卡与 DPDK 的框架



2、协议栈接受包流程：



当消息通过 port 传送到 Phy 层的 MAC 上后，主要的操作为：

①BD 将 DMA 的地址写到 MAC 的 mem 中

②MAC 通过 DMA 将 message 写到 BD_Buffer 中

③MAC 向第②步 BD_Buffer 对应的 BD 中写入具体的信息

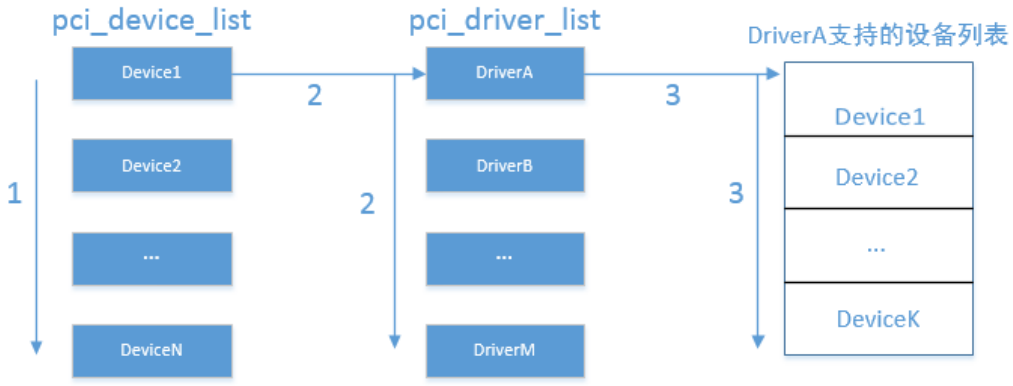
④MAC 通过 SoftIRQ 的方式通知内核将 BD_Bufeer 通过协议栈的方式向上层解析

⑤将解析后的消息通过 memcopy 的方式拷贝到用户空间

4、DPDK 的方式不走协议栈，他通过 mmap 将内核态中的 DMA 地址映射到用户态中，省去了一次拷贝，提高了性能。

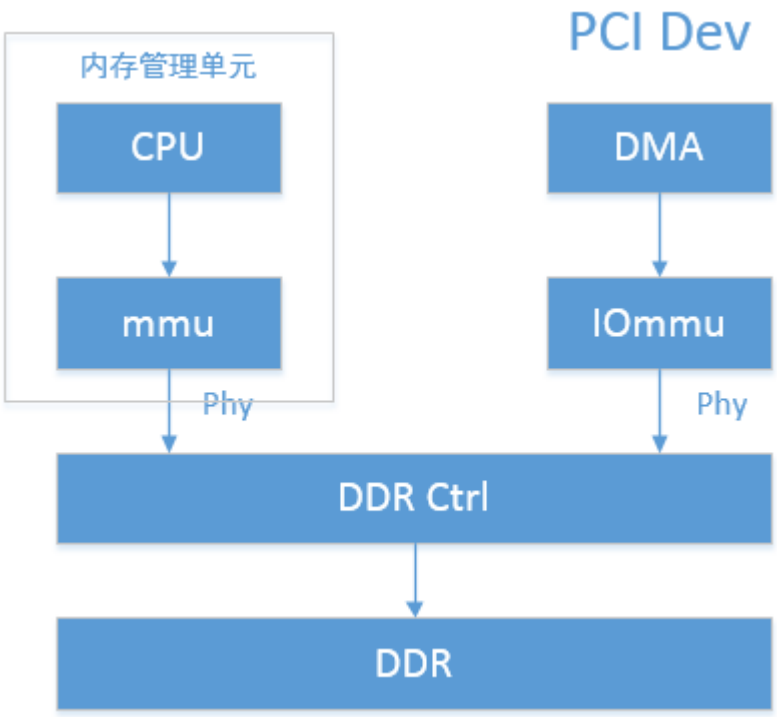
5、设备管理

```c
/* Init the PCI EAL subsystem */
int
rte_eal_pci_init(void)
{
    TAILQ_INIT(&pci_driver_list);
    TAILQ_INIT(&pci_device_list);
```



pci_device 在最开始就通过构造函数初始化 device 列表。

遍历 pci_device_list(如 Device1)，然后会去遍历 pci_driver_list(如 DriverA)，之后会进入到 DriverA 支持的设备列表中遍历，看看自己在不在其中，若不在，则退出继续遍历下一个 DriverB 的设备支持列表，直至把所有的 pci_driver_list 都遍历后，依然没有支持自己的 driver，gg。继续下一个设备 Device2 进行相同的操作，直至将 pci_device_list 遍历完成。



6、硬件设备需要得到物理地址连续的内存，硬件设备存在于内存管理单元之外，他并不理解什么是虚拟地址。

11、SPDK 背景介绍

固态存储设备正在取代数据中心。如今新一代的闪存存储，比起传统的磁盘设备，在性能、功耗和机架密度上具有显著的优势。这些现有的优势将会继续增大，使闪存作为下一代存储设备进入市场。

用户使用现在的固态设备，比如 Intel® SSD DC P4600 Series Non-Volatile Memory Express（NVMe）驱动，面临一个主要的挑战：因为固态设备的吞吐量和延迟性能比起传统的磁盘好太多，如今总的处理时间中，存储软件占用了更大的比例。换句话说，存储软件栈的性能和效率在整个存储系统中越来越重要。随着存储设备继续发展，它将面临远远超过现今使用的软件体系结构的风险（即存储设备受制于相关软件的不足而不能发挥全部性能）同时，在接下来的几年里，存储设备将会继续飞快发展到令人难以置信的程度。

为了提供一个完善的、端对端的参考存储体系结构—— Storage Performance Development Kit（SPDK）应运而生。SPDK 已经证明很容易达到每秒钟数百万次 I/O 读取，通过使用多个处理器核心和多个 NVMe SSD 进行存储，而不需要额外的硬件。SPDK 在 BSD license 许可协议下通过 Github 分发提供其全部的 Linux 参考架构的源代码。博客、邮件列表、文档和社区参与都可以在 http://www.spdk.io 中找到。

12、SPDK 软件体系结构

SPDK 如何工作？达到这样的超高性能运用了两个关键技术：运行于用户态和轮询模式。让我们进一步了解这两个软件工程术语。

首先，我们的设备驱动代码运行在用户态。这意味着，根据定义，驱动代码不会在内核中运行。避免内核上下文切换和中断可以节省大量的处理开销，从而允许更多的时钟周期被用来做实际的数据存储。无论存储算法（去冗，加密，压缩，空白块存储）多么复杂，浪费更少的时钟周期意味着更好的性能和很低的延迟。这并不是说内核会增加不必要的开销；相反，内核增加了那些可能不适用于专用存储堆栈的通用计算用例相关的开销。SPDK 的指导原则是通过减少每一处额外的软件开销来达到最低时延和最高效率。

其次，轮询模式驱动（Polled Mode Drivers，PMDs）改变了 I/O 的基本模型。在传统的 I/O 模型中，应用程序提交读写请求后睡眠，一旦 I/O 完成，中断就会将其唤醒。PMDs 的工作方式则不同，应用程序提交读写请求后继续执行其他工作，以一定的时间间隔回过头来检查 I/O 是否已经完成。这种方式避免了中断带来的延迟和开销，并使得应用程序提高 I/O 的效率。在旋转设备时代（磁带和机械硬盘），中断开销只占整个 I/O 时间的很小的比例，因此给系统带来了巨大的效率提升。然而，在固态设备的时代，持续引入更低延迟的持久化设备，中断开销已然成为了整个 I/O 时间中不能被忽视的部分。这个问题在更低延迟的设

备上只会越来越严重。系统已经能够每秒处理数百万个 I/O，所以消除数百万个事务的这种开销，能够节省额外的 CPU 资源。

SPDK 由众多子组件组成，相互链接并共享用户态操作和轮询模式操作的共有部分。每一个子组件的创建都是为了客户在构造端到端 SPDK 体系结构时遇到的特定功能和性能需求。同时，每一个子组件也可以被集成到非 SPDK 架构中，允许用户利用 SPDK 中用到的经验和技术来加速自己的软件。

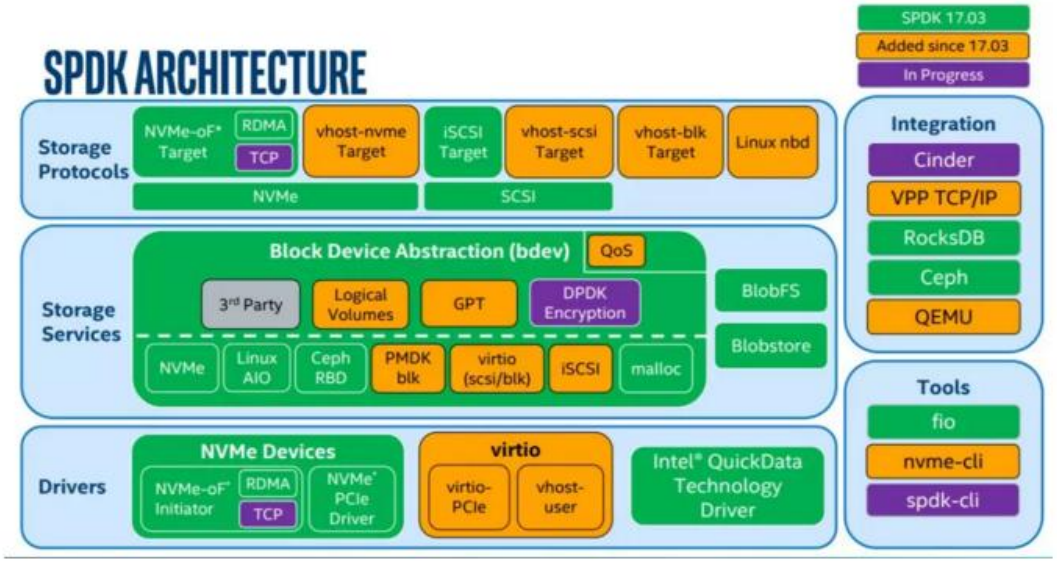


图1 SPDK Architecture

（该构架图主要对应了 SPDK 18.07 发布版本）

SPDK 架构图分为四层，从底层到上层分别是：

驱动层：提供高度优化无锁的 NVMe 驱动。

块设备：提供不同的后端块设备接口，方便不同场景的扩展。

存储服务：块设备抽象，对接上层不同的存储协议，并与后端的设备驱动粘合。并提供存储的一些额外功能（磁盘阵列，压缩，去冗余等）

存储协议：不同类型的存储访问协议。

12.1 SPDK 主要组件

SPDK 从 NVMe 开始不断完善，现在包括 NOF INI/TGT，支持虚拟机，加速，用户态块设备。根据协议发展同步发展，同时在进一步增强 Qos，管控面，存储特性比如简单 RAID 功能。SPDK 作为分布式或者云上存储,是一个比较好的解决方案，能够比较快的部署，实现高性能存储层。从下往上构建，主要的组件包括：

12.1.1 SPDK 驱动层

NVMe Driver： SPDK 的基础组件，这个高优化无锁的驱动有着高扩展性、高效性和高性能的特点。

Intel QuickData Technology： 也称为 Intel I/O Acceleration Technology（Inter IOAT，英特尔 I/O 加速技术），这是一种基于 Xeon 处理器平台上的 copy offload 引擎。通过提供用户空间访问，减少了 DMA 数据移动的阈值，允许对小尺寸 I/O 或 NTB 的更好利用。

NVMe over Fabrics（NVMe-oF）initiator： 从程序员的角度来看，本地 SPDK NVMe 驱动和 NVMe-oF 启动器共享一套共同的 API 命令。这意味着，例如本地/远程复制将十分容易实现。

12.1.2 块设备层

NVMe over Fabrics（NVMe-oF）initiator：本地 SPDK NVMe 驱动和 NVMe-oF initiator 共享一套公共的 API，简化本地远程 API 调用。

RBD：将 rbd 设备作为 spdk 的后端存储。

Blobstore Block Device：基于 SPDK 技术设计的 Blobstore 块设备，应用于虚机或数据库场景。

Linux AIO：spdk 与内核设备（如机械硬盘）交互。

12.1.3 存储服务层

Block device abstration layer（bdev）：这种通用的块设备抽象是连接到各种不同设备驱动和块设备的存储协议的粘合剂。并且还在块层中提供灵活的API，用于额外的用户功能，如磁盘阵列、压缩、去冗等等。

Blobstore：为SPDK实现一个高精简的文件式语义（非POSIX）。这可以为数据库、容器、虚拟机或其他不依赖于大部分POSIX文件系统功能集（比如用户访问控制）的工作负载提供高性能基础。

Blobstore Block Device：由SPDK Blobstore分配的块设备，是虚拟机或数据库可以与之交互的虚拟设备。这些设备得到SPDK基础架构的优势，意味着零拷贝和令人难以置信的可扩展性。

Logical Volume：类似于内核软件栈中的逻辑卷管理，SPDK通过Blobstore的支持，同样带来了用户态逻辑卷的支持，包括更高级的按需分配、快照、克隆等功能。

Ceph RADOS Block Device（RBD）：使Ceph成为SPDK的后端设备，比如这可能允许Ceph用作另一个存储层。

Linux Asynchrounous I/O（AIO）：允许SPDK与内核设备（比如机械硬盘）交互。

12.1.4 存储协议层

iSCSI target：建立了通过以太网的块流量规范，大约是内核LIO效率的两倍。现在的版本默认使用内核TCP/IP协议栈，后期会加入对用户态TCP/IP协议栈的集成。

NVMe-oF target：实现了NVMe-oF规范。将本地的高速设备通过网络暴露出来，结合SPDK通用块层和高效用户态驱动，实现跨网络环境下的丰富特性和高

性能。支持的网络不限于 RDMA 一种，FC，TCP 等作为 Fabrics 的不同实现，会陆续得到支持。

vhost target： KVM/QEMU 的功能利用了 SPDK NVMe 驱动，使得访客虚拟机访问存储设备时延迟更低，使得 I/O 密集型工作负载的整体 CPU 负载减低，支持不同的设备类型供虚拟机访问，比如 SCSI，Block，NVMe 块设备。

SPDK 不适合于所有的存储体系结构。一些常见问题的解答也许会帮助用户判断 SPDK 组件是否适合自己的体系结构。

技术背景

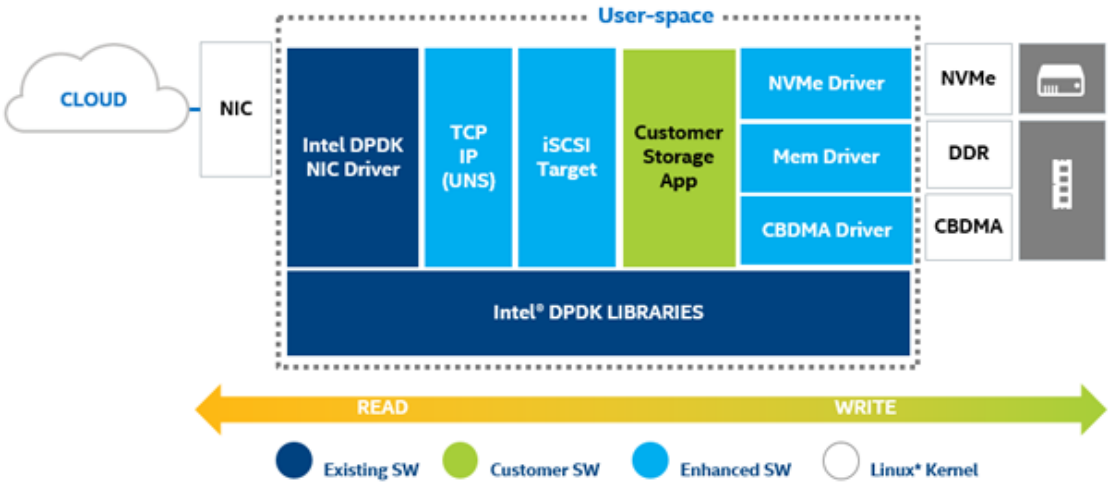
由于固态硬盘在数据中心逐步普及，新的闪存截止在性能，功耗，机架空间均相对于传统的存储介质存在较大优势。新的 NVMe 硬
盘的吞吐量和延时表现极佳（2GB/s 左右的读写带宽，45 万的每秒随机读和 17 万的每秒随机写，20μm 级别的时延），比传统的 SAS 和
SATA 盘快上千倍，也比 SATA SSD 快 5~10 倍。因此，NVMe 的推出，导致当前的性能瓶颈集中在存储软件栈上。SPDK 就是在这样的
背景下产生的。（第一次由硬件推动软件的进步）

SPDK

SPDK(Storage Performance Development Kit)，Intel 开源的包含一套驱动程序以及一整套端到端的存储参考架构。通过用户
态运行的从网卡到磁盘的高性能通路来提高性能和效率。目前 SPDK 与其另一个开源项目 DPDK 紧密配合，形成完整的存储软件栈。
Intel 目的是推广其 NVMe 驱动及高性能网卡。

SPDK 整体解决方案：

是一整套端到端的完整的存储参考架构，目标是能够把硬件平台的计算、网络、存储的最新性能进展充分发挥出来。



SPDK 中有三类子组件：网络前端、处理框架、后端。

网络前端：包括 DPDK 网卡驱动和用户态网络服务 UNS(用户态协议栈)

处理框架：得到数据包内容将 iSCSI 转化为 SCSI 块级命令，同时 SPDK 在这一层提供了一套 API 框架，支持用户自动以处理逻辑，例如：去重、压缩、加密、RAID 计算等。

后端：SPDK 提供的用户态 PMD(Polling Mode Driver)，支持 NVMe、Linux AIO、RAMDISK 等。

SDPK 关键技术：

1、轮询模式：使用专门的计算资源（通常是绑核）轮询，减少 IO 中断的处理开销；(绑核)

2、用户态驱动:NVMe 驱动运行在用户态,避免内核上下文切换和中断处理开销;

3、**集成 DPDK 收包和用户态协议栈;（通常在网络存储使用）。**

12.2 SPDK 技术总结

- SPDK为Intel开源的用户态驱动，主要应用场景是高性能的NVMe/ColdStream SSD，NOF。
- SPDK是一种比较好的用户态驱动框架，可以完全借鉴，快速开发。
- SPDK软件架构可以对接用户态块设备，虚拟机，Ceph等，在云计算、云存储和分布式存储中有比较高的应用价值。

- SPDK社区比较开放，可以积极参与，不拒绝支持ARM。
- SPDK不仅作为一种软件技术，而且可以作为高性能的一个宣传技术点。

12.3 SPDK 存储的应用策略

- **SPDK价值**：（1）基于SPDK开源代码提高用户开发效率；（2）利用SPDK的高性能编程模型支持高性能应用；（3）驱动不依赖产品独立交付，可以适应更多产品，走向通用驱动。（4）采用SPDK框架可以利用SPDK其他模块特性，扩展性更好。
- **技术策略**：需要在X86和ARM上都把SPDK基础技术积累起来，能够识别出其中的关键技术点和应用风险。主要用利用NVMe，NOF，用户态块设备。积极跟进SPDK社区的发展，特别是和云，虚拟化的配合。
- **应用策略**：在偏向云，分布式，服务器硬件，NOF硬盘框的场景优先考虑采用SPDK用户态架构；企业存储控制器继承基于存储基础设施的用户态驱动。
- **开源策略**：积极利用开源SPDK，对于bug类可以申请合入，对于对外接口，兼容性，管控面特性可以申请合入，对于内部关键DFx特性不开源。

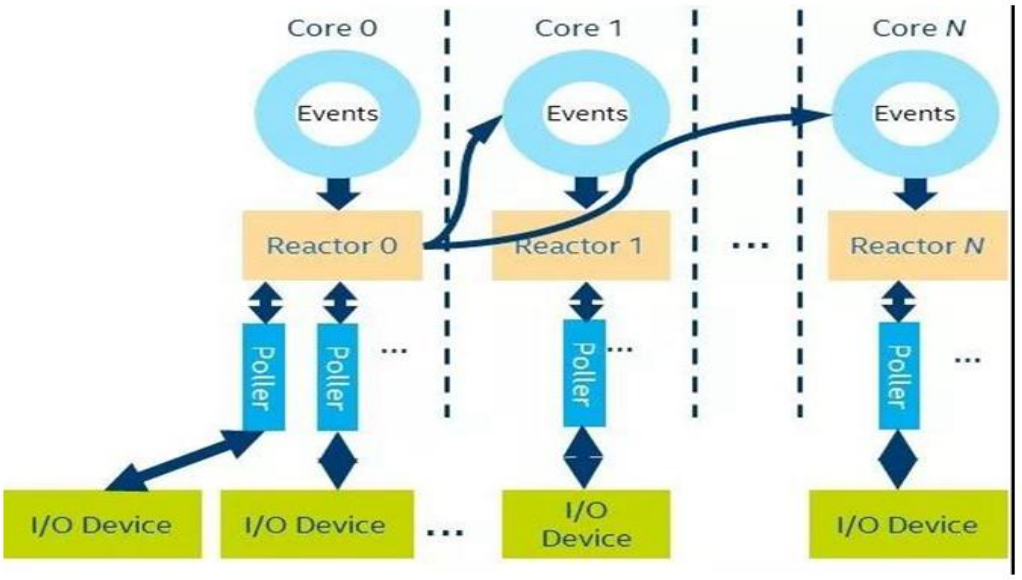
12.4 SPDK 存在问题

- **应用场景**：当前没有明确的产品化应用场景，需要在JBOF和分布式存储中落地基于SPDK的驱动，NOF前端也优先考虑基于SPDK。
- **技术掌握**：目前盘古平台还未完全掌握SPDK技术，没有完成详细分析和原型验证，未完成基于ARM的编译和验证，需要在技术项目中积累能力。

13、 SPDK 特点和其他技术

SPDK 开源做得比较好，以高性能为核心，不断发展壮大，intel 有专人负责开发社区，而且很乐意 intel 之外的公司积极参与社区发展，提交意见和补丁。社区有比较完成的极致，提交补丁后有自动化测试。

13.1 SPDK 应用编程框架



SPDK 的应用框架主要有三个部分：

1)、CPU Core 和线程的管理：通过初始化时绑核，每个核上运行一个 thread（Reactor），while(1)死循环运行，几乎 100%占用该核。

2)、线程间的高效通信：无锁机制，使用 Event 事件通知机制，每个 Event 的 Ring 缺省使用 DPDK 的机制，采用线性锁机制。

3)、IO 处理模型及数据路径的无锁化：读写及其对应的检查是否完成的函数在同一个 Core 上执行，不跨线程。当多个 Core 的 thread 操作同一 block device 时，使用 I/O Channel 的概念。

13.2 SPDK 应用案例

阿里介绍了基于 SPDK 的用户态存储驱动,阿里主要用了 SPDK 的基础用户态设施功能。同时介绍了几个用户态编程的问题,显示出阿里的用户态还是在刚刚入门,很多技术问题也是摸石头过河，感觉不像我司有完善的用户态基础设施。在硬盘可靠性上也开始重视，设计了注入工具来注入存储 IO 错误，用于测试系统的可靠性。

日立工程师讲了 iSCSI 的管控面优化。
华云网际工程师讲了应用 SPDK 的性能优化成果。
华为标准部代表讲了 OpenSDS 与 SPDK 的管控面结合项目。

1、用在提供块设备接口的后端存储应用，如 iSCSI Target 和 NVMe-oF Target。 （Ceph 提供共享存储集群的远端访问）

2、对虚拟机中 I/O（vitio）的加速，缩短了 host OS 中的 I/O 栈。 （腾讯云盘，阿里新春红包下利用的高 IO 本地磁盘）

3、加速数据库引擎，ByPass Kernel 的文件系统。例如 RocksDB 中，使用 SPDK 中的 blobfs/blobstore 与其集成，加速 RocksDB
的引擎使用。 （阿里的数据库集群也有使用 SPDK 和 RDMA）

13.3 Optane 结合 SPDK 技术

Intel 还重点介绍了 Intel 的 SSD，特别是 Optane。Optane 结合 SPDK 能发挥出最好的性能。还介绍了 ISA-L 加速库，可以广泛用于存储有关的加速，后续可以考虑在产品用应用。

13.4 SPDK 中国峰会介绍

SPDK 中国技术峰会 2018 在北京举行， 由 Intel 主办，SDNLAB 协办，目的在于探讨 SPDK 的现状和未来发展趋势，同时也为了推广 Intel 的新技术。
目前存储的介质有了飞速的发展，从机械硬盘到基于 SATA 协议的 NAND SSD，到基于 NVMe 协议的 NAND SSD，以及最新的基于 NVME 协议的 3D XPOINT SSD。存储设备的容量在不断的提升，I/O 存取的速度在不断的变快，从毫秒级到微妙级甚至到纳秒级。种种迹象表明存储硬件在快速发展，但是我们的存储软件设计还依然停留在原有的基于慢速存储设备的时代。为此，Intel 推出了 SPDK (Storage Performance Development Kit) 这个开源软件，旨在帮助客户优化存储系统的性能。

SPDK 中国技术峰会 2018 圆满结束。最后附上相关资料：

会议胶片：链接：https://pan.baidu.com/s/1wlEE77FkGlqBOdOEzMc29A 密码：5ppw

会议照片：链接：https://pan.baidu.com/s/1iYbKbJOEcyZZHVDnWW4kTA 密码：4xzu

视频（可回看）：http://www.itdks.com/eventlist/detail/1999

- Intel希望把SPDK变得更开放成为一个真正的开源社区，而不只是intel版本的开源项目。采用了Gerrithub作为补丁发布平台。支持持续集成，支持Trello看板。
- SPDK变得更加开放，非Intel的patch合入已经占到12.8%
- SPDK的架构一直在发展，超出了简单的用户态驱动，正在构建一个存储生态。
  最新计划支持基于 TCP 的 NOF，先支持内核 TCP 协议栈，未来支持 VPP 用户态协议栈；Qos：计划支持基于 bdev 的限速 Qos。
  virtio-blk 轮询驱动：计划支持 QEMU 虚拟机的 virtio-blk 存储。
  部署：正在开展简化 SPDK 部署工作，支持 JSON 配置。

13.5 SPDK 开源友好性

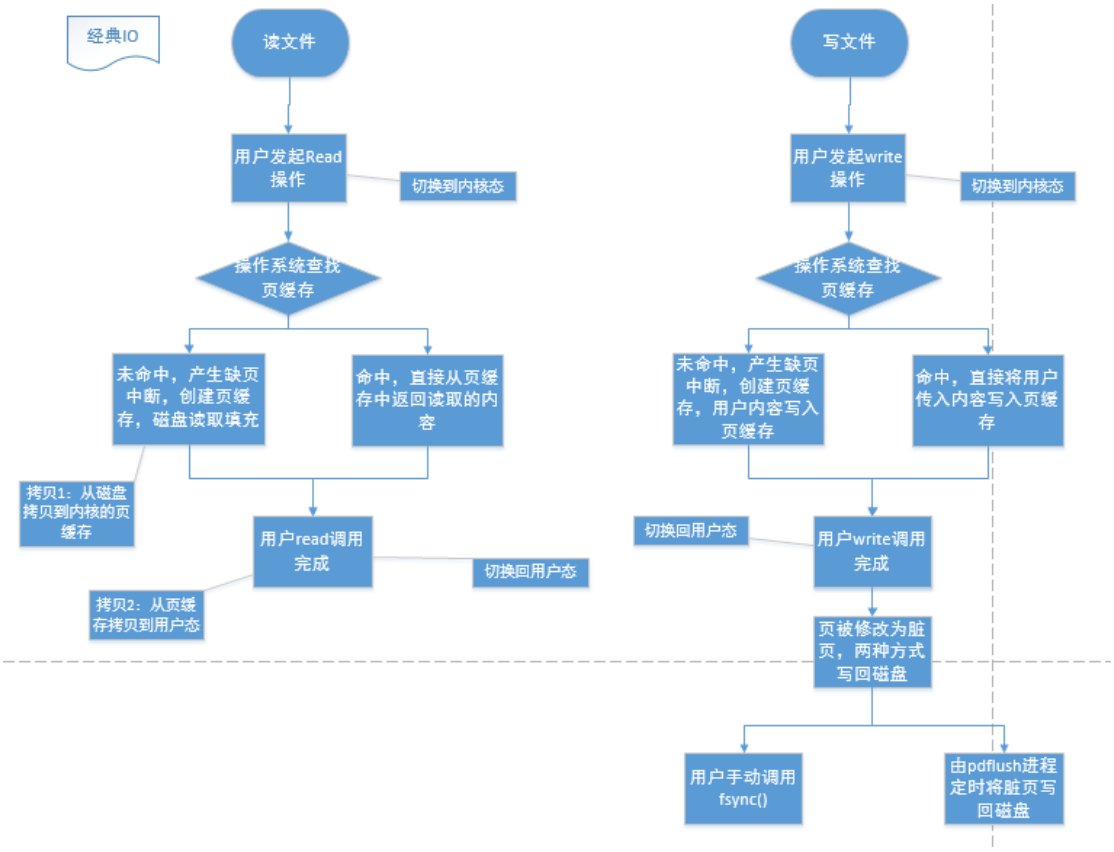
SPDK 和 DPDK 一样都来自 Intel 的孵化，SPDK 提供的是整套存储加速解决方案，里面用到了 DPDK 库。开源协议方面，SPDK 协议友好，DPDK 部分核心组件采用的是 GPLv2。

组件	开源协议版本	分析
SPDK	BSD license	商业友好，可随意修改
DPDK core lib and drivers	BSD-3-Clause	商业友好
DPDK kernel components	GPL-2.0	对修改不友好

14、SPDK 和当前技术对比

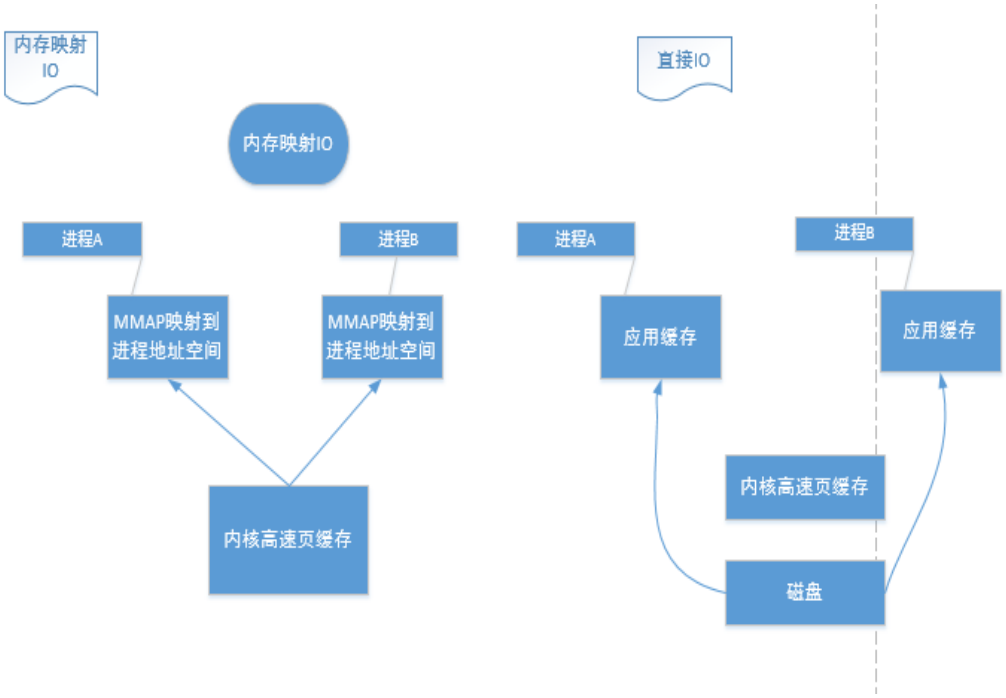
14.1 基于 OS 的文件操作

经典 IO：通过 OS 提供的 write/read 标准接口，中间经过系统的页缓存，由操作系统控制写。



内存映射 IO：通过 OS 提供的内存映射接口，将内核高速缓存直接映射到用户进程的地址空间。

直接 IO：不经过高速缓存，直接从物理磁盘读取数据，常用于数据库系统。



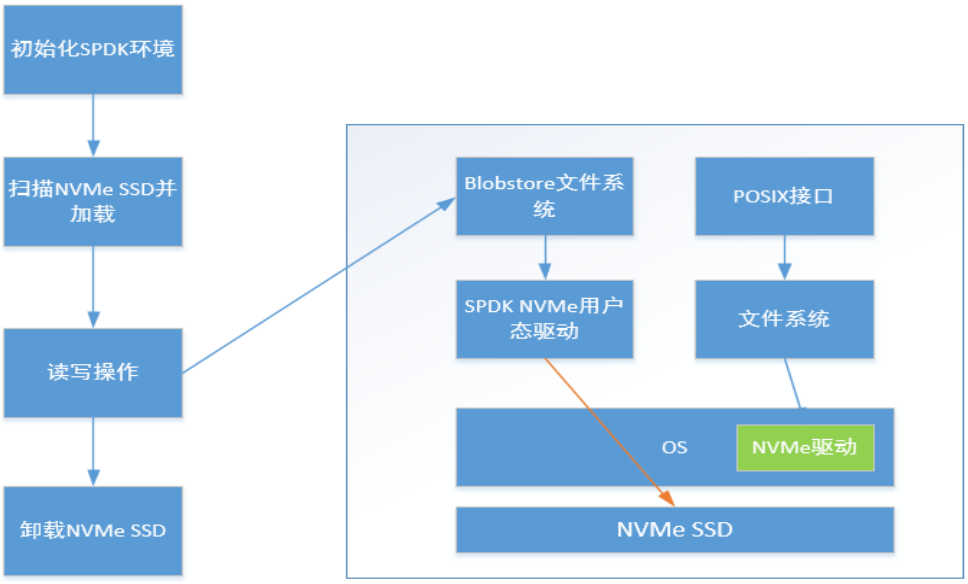
14.2 基于 SPDK 架构的文件操作

基于 SPDK 的基本操作：

1、SPDK 的用户态驱动需要绑核；

2、SPDK 提供了简单的文件系统 Blobstore 的非 Posix 接口；

3、SPDK 的 NVMe 驱动在用户态，Bypass 内核的驱动。



SPDK 当前不足：

1、把内核驱动放到用户态，需要在用户态实施一套基于用户态软件驱动的完整 I/O 栈。

2、无法使用当前 Linux 的内核文件系统，EXT4 和 btrfs 等。仅提供了简单的 blobfs/blostore 文件系统，并不支持 POSIX 接口。

根据前面的 SPDK 应用场景及案例分析，以及结合云核的现状分析，建议后续研究方向如下：

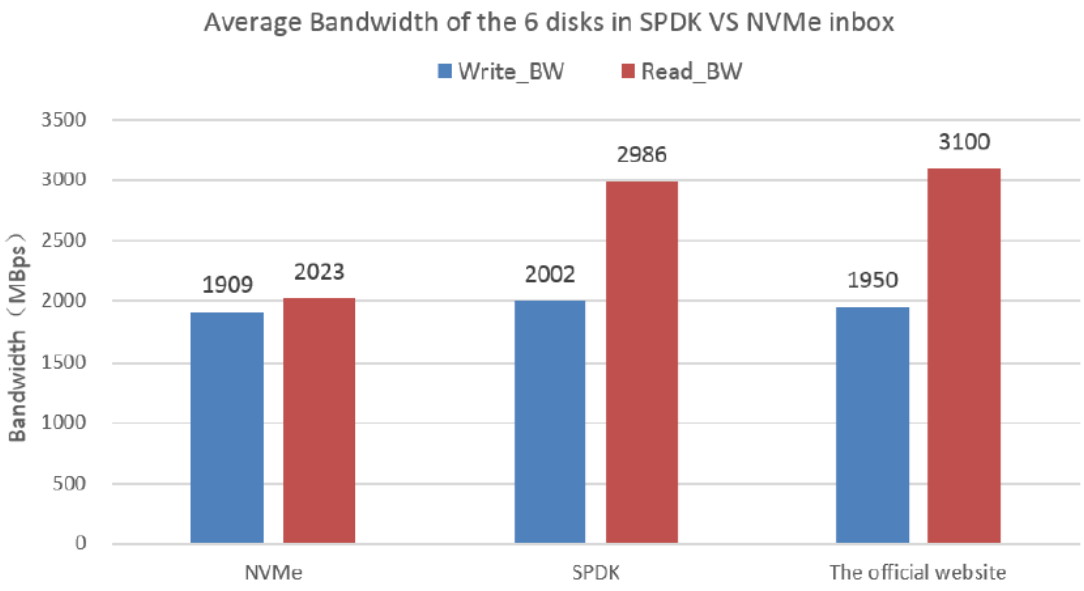
1、分析如果直接使用 SPDK 当前的应用 API 框架，适配和改造工作量；
2、基于当前 SPDK 的用户态驱动架构和块存储模型，是否可以提供/改造为更友好的支持 Posix 接口的文件系统；

14.3 SPDK 测试对比分析

在此，通过华为ES3000 SSD磁盘为例，说明使用SPDK功能以后性能优化情况。ES3000 V3采用华为Hi1812 ASIC SSD高性能控制器。支持NVMe标准，多队列IO技术提升SSD性能及QoS表现。创新NVMControl技术，保障产品高性能和高可靠。

14.3.1 带宽测试结果对比

测试环境:主机通过千兆交换机与服务器相连,服务器配置了12个ES3000 V3 NVMe盘，RHEL7.0 x64，SPDK 17.03，SPDK17.03自带的工具。在SPDK和NVMe驱动下，6个盘的平均带宽

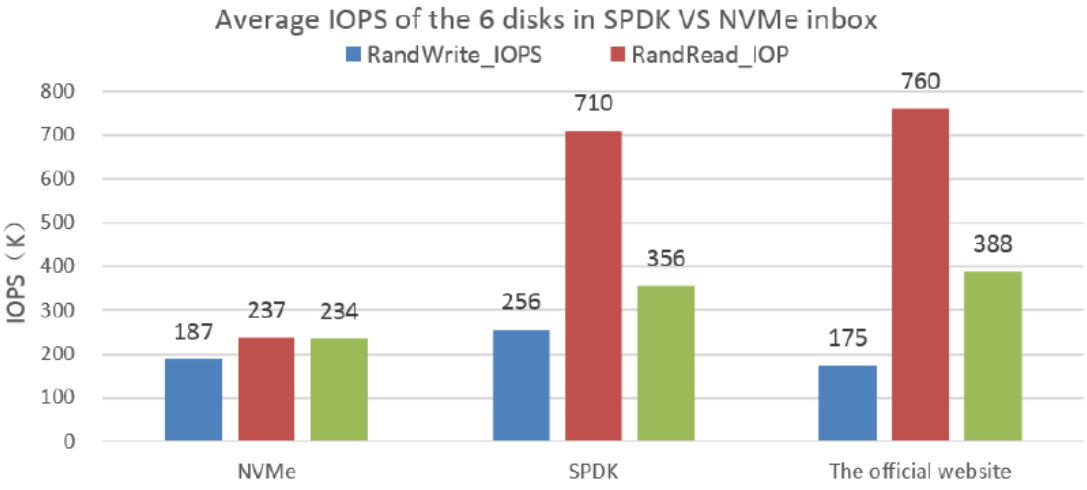


写带宽：SPDK驱动比NVMe开源好5%。

读带宽：SPDK驱动比NVMe开源好32%。

14.3.2 IOPS 测试结果对比

在SPDK和NVMe驱动下，6个盘的平均IOPS



写IOPS：SPDK驱动比NVMe开源好26%。

读IOPS：SPDK驱动比NVMe开源好66%。

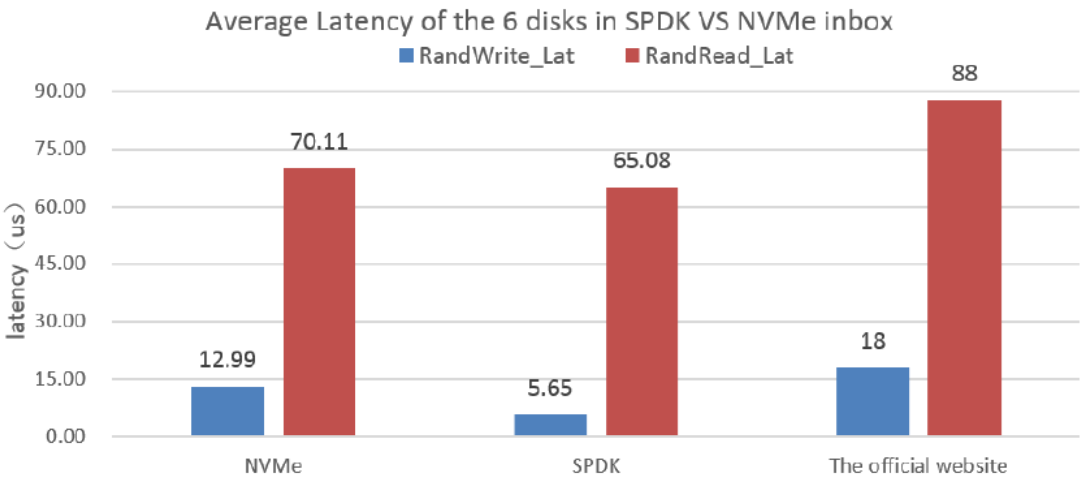
混合7:3 IOPS：SPDK驱动比NVMe开源好33%。

14.3.3 时延测试结果对比

在SPDK和NVMe驱动下，6个盘的平均Latency

写时延：SPDK驱动比NVMe开源好100%。

读时延：SPDK驱动比NVMe开源好7%。



从上面的测试数据来看，SPDK驱动的测试结果明显好于NVMe开源驱动的测试结果，并且SPDK驱动的测试结果优于官网宣称值，6个盘在SPDK驱动下性能可以呈线性增长。

15、SPDK 存储模型 Blobstore

SPDK 建立在 SPDK blobstore 之上，因为 blobstore 提供了高性能、低开销的群集分配。在 blobstore 中，也支持精简配置、快照和写入时复制。在最新的 SPDK 版本中，它实现了具有逻辑卷功能。

15.1 blobstore 介绍

Blobstore 被定义为一种块分配的存储池，从应用上讲，blobstore 是构建在底

层块设备上的一种逻辑块分配的存储系统，类似于传统意义上的文件系统，但并非文件系统。Blobstore 也不支持 posix 语义，同时为了和传统 posix 语义上的文件系统区分开，blobstore 使用了 blob 对象来表示传统文件系统中的 files 和 objects 等对象。

从层次上，blobstore 为上层的应用提供了基础的块存储切片服务，blobstore 原始设计是基于数据库应用的，但可以基于 blobstore 构建很多现有的上层存储服务。例如：逻辑卷、数据库、文件系统（BlobFS 和 RocksDB 等）、key/value 存储等等，甚至包括 SAN、NAS 和各类分布式存储系统。

15.2 blobstore 中的对象

Blobstore 建立了抽象的层次式存储结构，其中包含的对象有：blobstore、blob、cluster、page 以及 logical block（逻辑块）。

逻辑块：底层块设备的最小访问单元，一般 512B 或 4KB

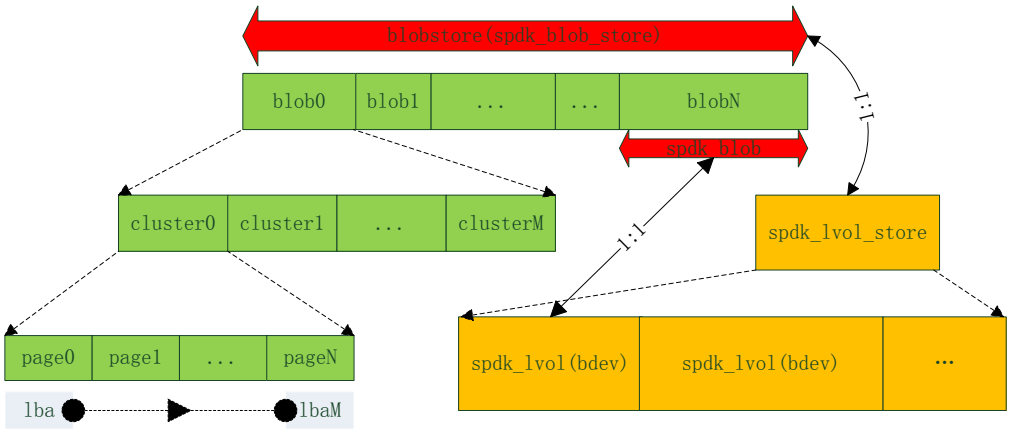
Page 页：包含了固定的逻辑块数目，page 中的逻辑块一定是物理连续的。Page 大小一般 4KB，SSD 设备必须能够原子的读写一个 Page。

Cluster 簇：包含了固定数目的 Page，这个数目是创建 blobstore 时指定的。一般一个 cluster 大小 1MB 或者 4MB。Cluster 内的 Page 也是物理上连续的。

Blob：blob 是一个有序簇组成的簇链表，应用程序会直接操作 blob。Blob 是持久化到物理块设备上的，因此断电和重启数据不丢失。应用程序通过 blobstore 中保存的 blob 标识符来访问特定的 blob，blob 的读写访问以 page 为单位，通过指定 page 在 blob 中的起始偏移即可。应用程序可以往 blob 中存储 key/value 键值对，这些键值对实际形成了 blob 的属性（xattrs）。

Blobstore：一个 blobstore 管理了整个底层块设备，基于 blobstore 的应用程序，被称作一个 blobstore。Blobstore 由元数据区域和多个 blobs 的集合组成。

下图描述了 blobstore 中的逻辑层次结构：



黄色区域描述了基于 blobstore 构建的逻辑卷应用，直接使用 blobstore 作为传统逻辑卷组的概念，而 blobstore 中的每一个 blob 则形成了逻辑卷组下的一个逻辑卷划分。每一个逻辑卷 spdk_lvol 会关联一个 bdev 设备(spdk_bdev)，这样就可以抽象成 spdk 中的块设备，通过注册 spdk_bdev_fn_table 完成虚拟块设备的读写注册。

15.3 blobstore 关键数据结构

从物理布局上，blobstore 中有两类元数据，一类是 blobstore 的元数据，主要包括占用一个 page 大小的 super_blob 和 3 张位图数组。3 张位图数组用于管理 blobstore 资源的分配，主要包括 blob 的元数据页的分配、簇的分配、blobid 的分配。另一类是 blob 的元数据，blob 的元数据记录在 blob 元数据页上，主要包括 blob 的 flag 标志位、blob 的属性(xattrs)、blob 的 extents(一个 extent 由一段物理上连续的多个簇组成)。

1.1. spdk_bs_super_block

该结构存储了 blobstore 的管理数据，其中使用了 3 张位图数组管理了整个 blobstore 的资源分配，主要包括簇分配位数组、blob 元数据页（元数据 Page）分配位数组和 blobid 分配位数组。每一个 blobid 的分配都对应一个 blob 的生成，blobstore 的所有 blob 中，存在一个特殊的 blob，即 super_blob。在创建 blobstore 时，会同时创建 super_blob，这个 blob 的主要作用是记录了用户程

序对该 blobstore 的一些定制内容，例如逻辑卷应用中，会使用 super_blob 来记录逻辑卷组的名称以及生成的uuid,这些信息以key/value对形式记录到blob的属性中（xattrs)。另外，super_blob 区别于普通 blob，它不分配数据簇。super_blob 主要字段解释如下：

```
struct spdk_bs_super_block {

    uint8_t       signature[8];   /*8个字节的签名，固定为SPDKBLOB */

    uint32_t        version;  /*blobstore的版本，目前是3 */

    uint32_t        length;     /* 整个super_blob的长度，目前固定是4KB
*/

    uint32_t   clean;

    /* 该字段只有正常关闭blobstore时为1，否则为0：

    这意味着每次打开blobstore时，都需要把该字段的物化值更新为0。如果打开blobstore时读取该值不为1，则blobstore会进入一种recovery修复模式，尝试修复blobstore的不一致数据，具体见代码实现。

    */

    spdk_blob_id  super_blob;    /* super_blob 对应的id */

    uint32_t   cluster_size; /*blobstore的簇大小,创建blobstore时指定，不指定默认为4MB*/

    uint32_t   used_page_mask_start;

    /* 记录blob元数据页分配的位图数组存储在物理块设备上的page偏移量
*/

    uint32_t   used_page_mask_len; /* 元数据页分配占用的page数 */
```

```
    uint32_t   used_cluster_mask_start;

    /*记录簇分配的位图数组存储在物理块设备上的page偏移量 */

    uint32_t   used_cluster_mask_len; /* 簇分配位数组占用的page数 */

    uint32_t   md_start; /* blob元数据页在物理块设备上的起始page偏移量
*/

    uint32_t   md_len; /* blob元数据占用的page数 */

    struct spdk_bs_type  bstype; /* blobstore类型 */

    uint32_t   used_blobid_mask_start; /* Offset from beginning of disk,
in pages */

    uint32_t   used_blobid_mask_len; /* blobid分配位数组占用的page数 */

    uint8_t       reserved[4012];  /* 保留字节 */

    uint32_t   crc;                  /* crc校验码 */

};
```

1.2. spdk_blob_store

该结构缓存了 blobstore 运行时关键的数据，其中一部分是从 super_block 中拷贝过来，一部分是通过计算得到的，另一部分是运行时可变的，例如三张位图数组。可以知道，该结构中的数据需要及时刷下去，否则会造成断电丢失。主要字段解释如下：

```
struct spdk_blob_store {

    uint64_t          md_start; /* blob元数据页的起始page偏移*/

    uint32_t          md_len; /* blob元数据页的page数 */
```

```
    struct spdk_io_channel      *md_channel;
    uint32_t          max_channel_ops;
    struct spdk_thread      *md_thread;
/* 元数据更新的线程，当前仅支持该线程持久化元数据 */
    struct spdk_bs_dev      *dev;  /* 连接blobstore和bdev的设备对象*/
    /* 三张位图，解释见super_block */
    struct spdk_bit_array       *used_md_pages;
    struct spdk_bit_array       *used_clusters;
    struct spdk_bit_array       *used_blobids;
    pthread_mutex_t          used_clusters_mutex;
    uint32_t          cluster_sz; /* 簇大小 */
    uint64_t          total_clusters; /*簇数目*/
    uint64_t          total_data_clusters; /* 数据簇数目*/
    uint64_t          num_free_clusters; /*未分配簇数目*/
    uint32_t          pages_per_cluster; /*一个簇对应的page数目*/
    spdk_blob_id          super_blob;
    struct spdk_bs_type      bstype;
    struct spdk_bs_cpl       unload_cpl; /* unload blobstore后的回调*/
    int           unload_err;
```

```
    TAILQ_HEAD(, spdk_blob)      blobs; /* blobstore中的blobs*/

    TAILQ_HEAD(, spdk_blob_list)    snapshots;    /*blobstore    中    的
snapshots*/

};
```

1.3. spdk_blob

该结构缓存了blob运行时的关机数据结构，其中包括了blob的属性、blob的状态、blob的读写flags标志位、该blob的元数据页等。主要字段解释如下：

```
struct spdk_blob {

    struct spdk_blob_store *bs; /* blobstore的运行时数据结构*/

    uint32_t   open_ref;   /* 打开引用计数*/

    spdk_blob_id   id;    /* 该blob的id，从该id能够找到blob的首个元数据
页的索引*/

    spdk_blob_id   parent_id;   /* clone和snapshot使用，记录父blob的id*/

    enum spdk_blob_state     state; /* blob的状态，dirty时表示需要刷新
到物理块设备*/

    /* Two copies of the mutable data. One is a version

     * that matches the last known data on disk (clean).

     * The other (active) is the current data. Syncing

     * a blob makes the clean match the active.

     */

/*

 Clean是底层物理设备上持久化的数据，active是blob中新更新过的数据，需要
在适当时刻刷到底层物理设备，一旦刷到底层设备，active会重新赋值给clean。

*/
```

```
    struct spdk_blob_mut_data   clean;
    struct spdk_blob_mut_data   active;
    bool        invalid;
    bool        data_ro;
    bool        md_ro;
/*blob对应的flags标志位，该标志位会持久化到blob的元数据页中 */
    uint64_t   invalid_flags;
    uint64_t   data_ro_flags;
    uint64_t   md_ro_flags;
/* 目前只有瘦分配会初始化该字段，该虚拟设备用作分配簇时的cow动作 */
    struct spdk_bs_dev *back_bs_dev;
    /* TODO: The xattrs are mutable, but we don't want to be
     * copying them unecessarily. Figure this out.
Blob 的 xattrs 属性
     */
    struct spdk_xattr_tailq xattrs;
    struct spdk_xattr_tailq xattrs_internal;
    /* blob的挂链项*/
    TAILQ_ENTRY(spdk_blob) link;
};
```

1.4. spdk_blob_md_page

该结构是最终持久化到物理块设备的blob元数据页数据结构体，每一个元数据page页持久化的数据主要保存到descriptors数组中，主要字段解释如下：

```
struct spdk_blob_md_page {
```

```
    spdk_blob_id     id;
/*blob id*/
    uint32_t        sequence_num;
    /*该blob元数据页的序列号*/
    uint32_t   reserved0; /* 保留字节*/
    uint8_t        descriptors[4072];
/*
该数组实际序列化了blob三类元数据：flags标志位、blob属性（xattrs)、blob的extents，最终会持久化到物理块设备上。
*/
    uint32_t   next;
/* 下一个该blob元数据页的page索引号，转换为lba后，可以递归读取下一个blob元数据页，例如：
    next_page = page->next;
    next_lba  =  _spdk_bs_page_to_lba(blob->bs,  blob->bs->md_start  + next_page);
*/
    uint32_t   crc;       /* 以上数据的crc校验码 */
};
```

1.5. blob 的元数据结构体

（1）blob 的 flags 标志位

```
struct spdk_blob_md_descriptor_flags {
    uint8_t    type;
    uint32_t   length;
    /* blob的三个读写标志位*/
    uint64_t   invalid_flags;
    uint64_t   data_ro_flags;
    uint64_t   md_ro_flags;
};
```

（2）blob的属性xattrs

```
struct spdk_blob_md_descriptor_xattr {
    uint8_t       type;
    uint32_t   length;
    uint16_t   name_length;
    uint16_t   value_length;
    char       name[0];     /*name 对应 key， value 紧跟在 name 后面， 由 name_length和value_length指定key和value的长度和偏移量。*/
    /* String name immediately followed by string value. */
```

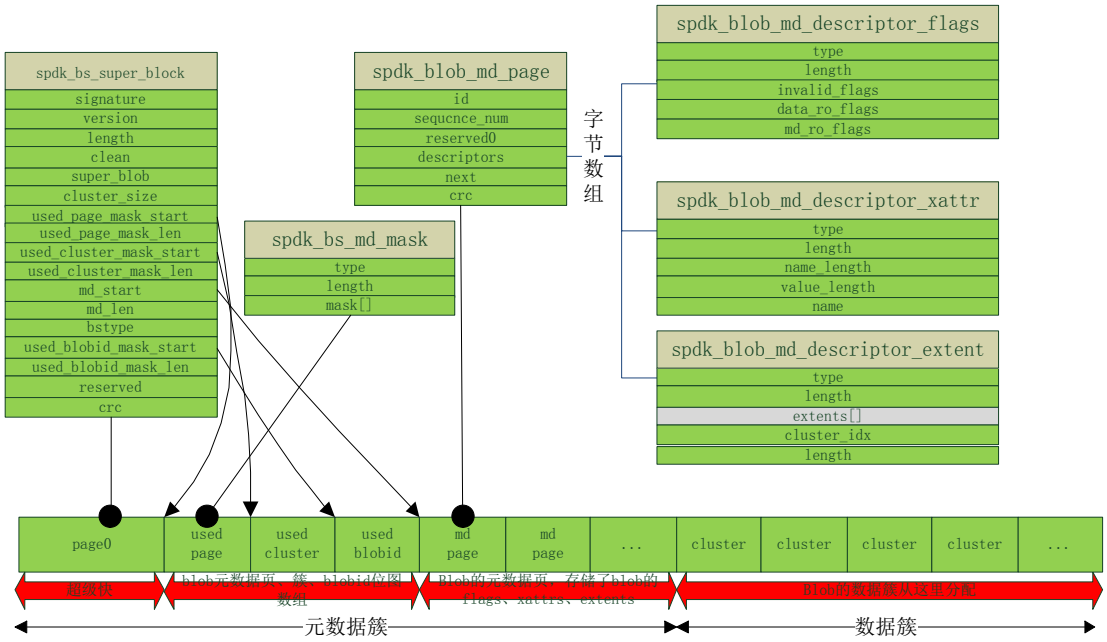
```
};
```

（3）blob的簇extents

```
struct spdk_blob_md_descriptor_extent {
    uint8_t       type;
    uint32_t  length;
    struct {
        uint32_t        cluster_idx;  // 其实簇的lba地址
        uint32_t        length; /* In units of clusters 该extents对应的簇数目 */
    } extents[0];
};
```

15.4 blobstore 元数据物理分布

第 2 章节描述了 blobstore 和 blob 相关的数据结构，这些数据结构中有三类，会分别存储到底层物理块设备上。第一，super_block；第二，资源分配数组；第三，blob 的元数据页。具体物理存储结构如下图：



15.5 元数据页的分配计算

（1） super_block

super_block 固定占据块设备的第 0 个 page（4KB）。

（2） 位图数组

Blobstore 默认为每一个簇分配一个 Page 的元数据页，这是为了支持最小的 blob（仅一个簇）时，blob 至少需要一个 Page 来记录 blob 的元数据。

假设 blobstore 的总的簇数量为 N，那么 md_len 为 N。

由于第 0 个 page 是 super_block，那么 used_page_mask_start 恒为 1，used_page_mask_len 的计算如下：

```
divide_round_up(sizeof(struct spdk_bs_md_mask) +
                divide_round_up(bs->md_len, 8),
SPDK_BS_PAGE_SIZE)
```

后面的 used_cluster 和 used_blobid 位图数组依次往后排，并按照上述方式计算。

如下的一个 super_block，可以看到 used_page_mask_start 为 1，used_page_mask_len 为 12；used_cluster_mask_start 为 13，used_cluster_mask_len 为 12；used_blobid_mask_start 为 25，used_blobid_mask_len 为 12。可以知道，默认情况下三张位图的大小是一样的，这里都占据 12 个 page 页。

```
00000000  53 50 44 4b 42 4c 4f 42  03 00 00 00 00 10 00 00  |SPDKBLOB........|
00000010  01 00 00 00 00 00 00 00  01 00 00 00 00 00 40 00  |..............@.|
00000020  01 00 00 00 0c 00 00 00  0d 00 00 00 0c 00 00 00  |................|
00000030  25 00 00 00 6a d2 05 00  4c 56 4f 4c 53 54 4f 52  |%...j...LVOLSTOR|
00000040  45 00 00 00 00 00 00 00  19 00 00 00 0c 00 00 00  |E...............|
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

```
    signature: 8B(SPDKBLOB) version: 4B(3) length: 4B(4KB)
    clean: 4B(1)  super_blob: 8B(0x0000 0001 0000 0000) cluster_size: 4B(0x00400000B)
used_page_mask_start:4B(1) used_page_mask_len:4B(12) used_cluster_mask_start:4B(13:12)
 md_start:4B(37) md_len:4B(381546) bstype:16B(LVOLSTORE)
  used_blobid_mask_start:4B(25) used_blobid_len:4B(12)
  reserved: 4012B
```

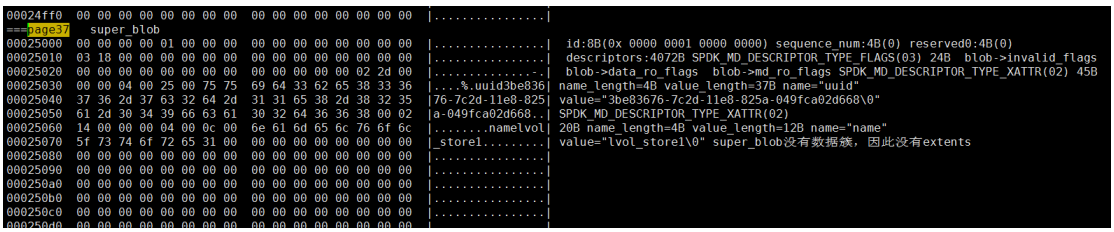
used_md_pages 位图数组示例如下：

```
00000fe0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000ff0  00 00 00 00 00 00 00 00  00 00 00 00 e5 18 6a a3  |..............j.|   crc: 4B(0xa36a18e5)
===page1  usded_md_pages
00001000  00 6a d2 05 00 03 00 00  00 00 00 00 00 00 00 00  |.j..............|  SPDK_MD_MASK_TYPE_USED_PAGES(0) 元数据page个数：381546(0005d26a)
00001010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|  第0（对应page37） 和 第1(对应page38) 元数据page被使用
00001020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000010a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000010b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000010c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

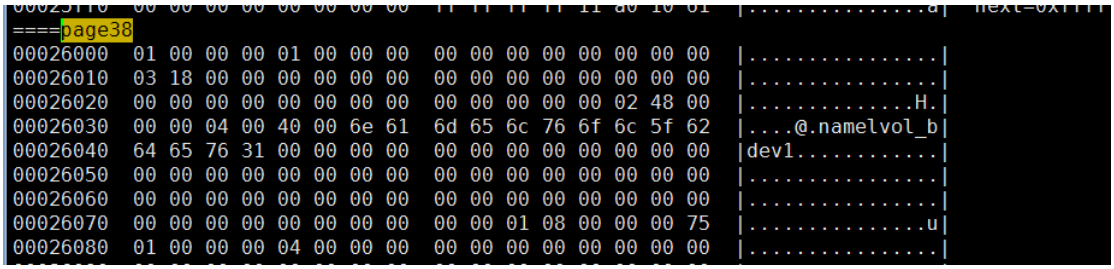
（3） Blob 的元数据页

位图数组一旦分配完，就得到了 blob 元数据页的起始 page，如上 md_start 为 37，是由 1 + 12 + 12 + 12 得到的。而 md_len 固定由底层块设备的簇数量决定，这里为 381546 个 Page。由此可知元数据共占据 381546 + 37 = 381583 个 Page。

Super_blob 的元数据页示例如下：



某个逻辑卷对应的 blob 元数据页示例如下：



（4） 数据簇

将上述元数据按照占用的 page 数量进行簇整数边界对齐后，去掉这些簇，剩余的簇就是数据簇，数据簇由 blobstore 在创建 blob 时分配给 blob（瘦分配在写入时才进行分配）。分配数据簇时要修改位图数组，因此位图数组需要及时刷到物理块设备上。

如上，元数据使用的簇数量 divide_round_up(num_md_pages, bs->pages_per_cluster)

= divide_round_up(381583, 4MB / 4KB)

=373。

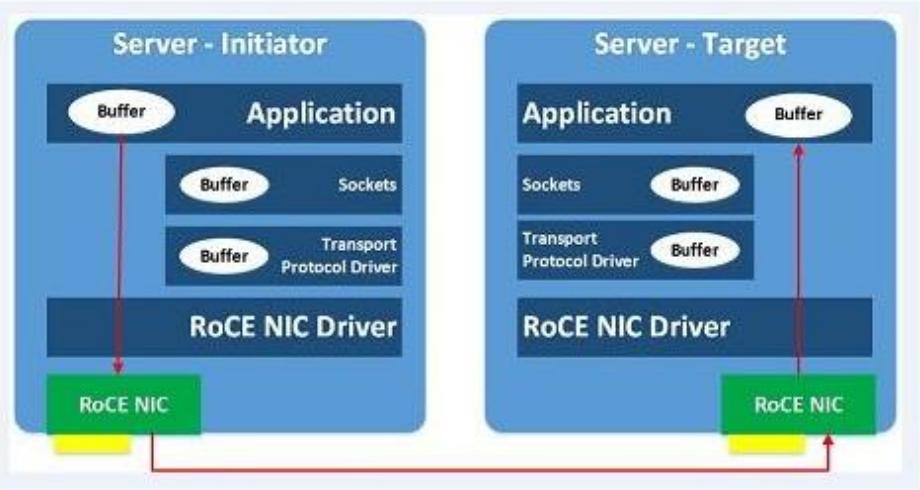
因此数据簇数量 381546 - 373 = 381173。

至此也可以计算 blobstore 的空间利用率：100% * 381173 / 381546.0 = 99.90223%，元数据管理浪费的空间不到千分之一。

16、相关技术介绍

16.1 RDMA 高性能网络框架

Remote DMA。相对于传统的 TCP socket 数据传输，RDMA 技术可以将应用程序 Buffer 中的数据直接拷贝到网卡内存并发送到远端，远端把数据 DMA 到应用的缓存中，相当于在应用之间建立了更快速更直接的通道。



RDMA技术优势：

1）zero-copy：数据不需要在网络协议栈的各个层之间来回拷贝，这缩短了数据流路径。

2）kernel-bypass：应用直接操作设备接口，不再经过系统调用切换到内核态，没有内核切换开销。

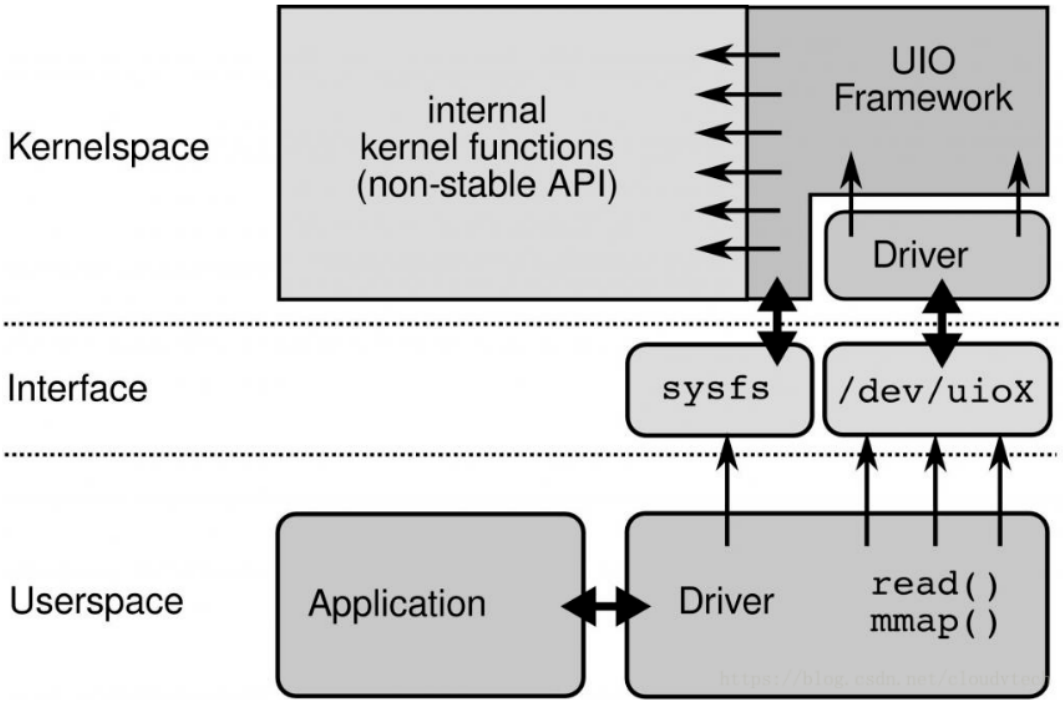
3）none-CPU：数据传输无须CPU参与，完全由网卡搞定，无需再做发包收包中断处理，不耗费CPU资源。

16.2 用户态 IO 技术 UIO

传统的Linux Kernel IO模型：所有设备IO均需要经过内核处理，在高并发的情况下，大量的硬件中断会降低内核数据包的处理能力，内核和用户控件的数据拷

贝也会造成大量的计算资源浪费。

Linux UIO：将硬件操作直接映射到用户空间的kernel bypaas的方案。



UIO如何完成一个设备驱动的两个任务：

1、存取设备的内存：

UIO实现mmap()可以处理物理，逻辑，虚拟内存。

2、处理设备的中断：

只将设备的中断的应答放在内核（仅仅做应答中断和禁止中断），其余空间工作全部留给用户空间完成。

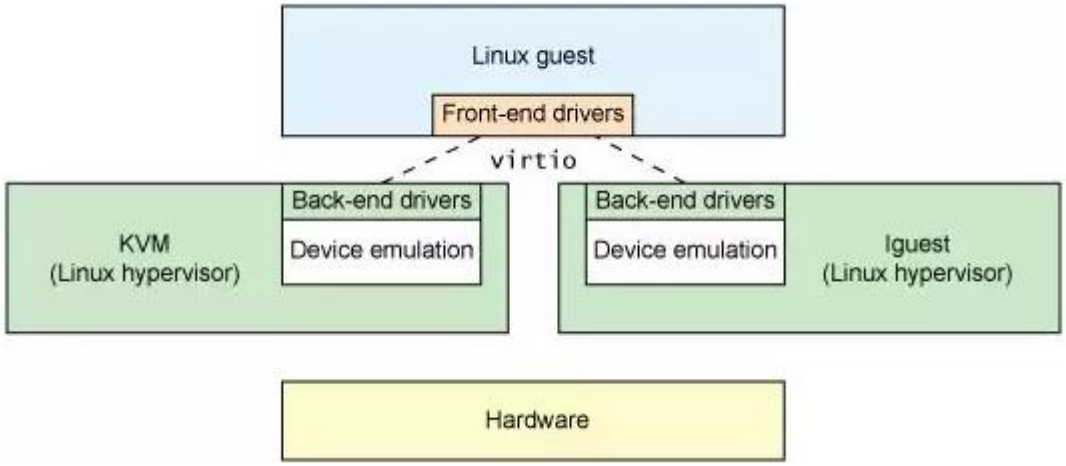
等待中断：可以阻塞在./dev/uioX/的read上，也可以使用seletct来做有限时间等待。

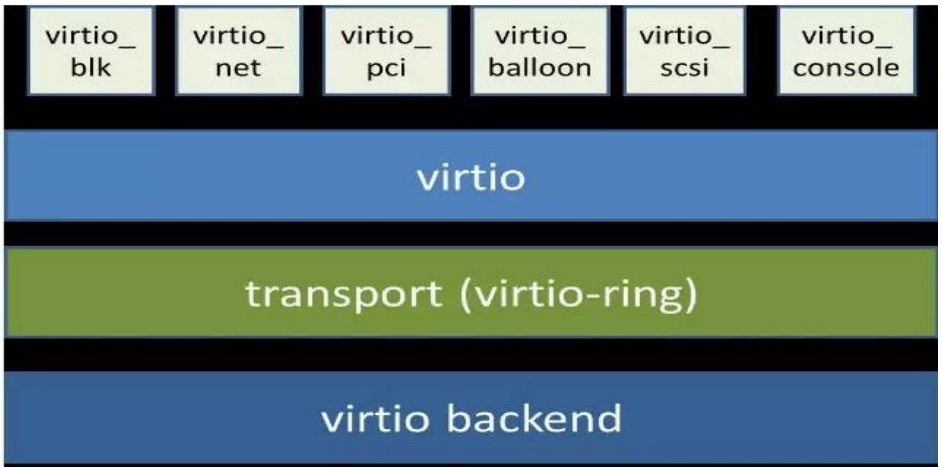
设备控制：通过/sys/class/uio下的各个文件读写来完成。

16.3 Virtio 技术介绍

Virtio出现的原因：解决虚拟化时，IO虚拟化带来的损失，使用半虚拟化，硬件

辅助的方式提升IO效率。Virtio是一套标准，来固定guest里的前端驱动和后端的Hypervisor之间的驱动的接口。





Virtio架构：

1、前端Guest OS里的各种驱动程序：virtio-blk，virtio-net，virtio-pci等。

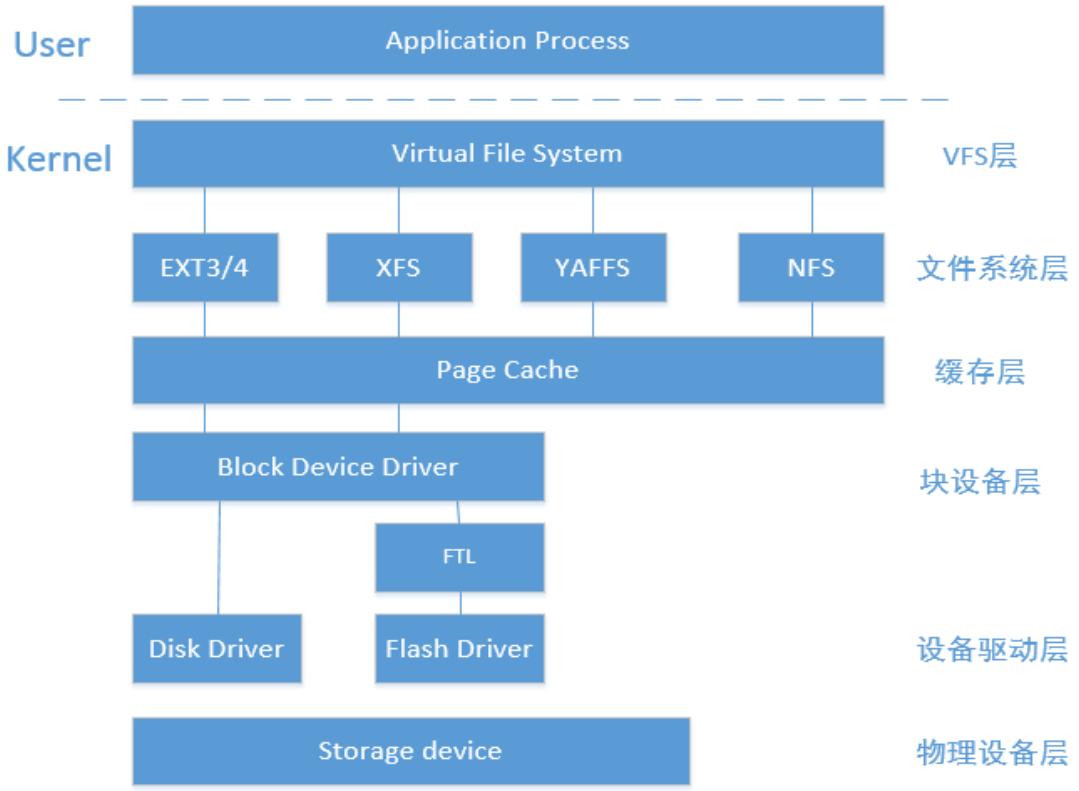
2、通信层：用于前端和后端通信的virtio层和virtio-ring层。Ring实现了两个环形缓冲区，分别保存前端和后端处理程序执行的信息。

3、后端Hypervisor里的驱动，负责接收前端的驱动指令，并和硬件交互。

16.4 NVMf 技术介绍

NVMe over Fabric，可以看做是除了PCI-e之外访问NVMe的另一种途径。Fabrics链路可以是RDMA或者是FC。

16.5 Linux 文件系统架构介绍



Linux文件系统层次架构：

用户层：使用Posix接口操作文件。

VFS层：提供通用的文件系统System call调用接口，屏蔽下层的具体文件系统。

文件系统层：不同的文件系统实现了VFS定义的函数，并注册到VFS框架。

缓存层：Page Cache，Linux有自己的IO管理机制，数据均缓存在这里。

块设备层：提供块设备的读写接口，磁盘的驱动将从此读取命令。

磁盘驱动层：驱动程序将读写命令转换为各自的磁盘的协议，包括ATA，SCSI等。

磁盘物理层：读写数据到磁盘介质。

17、SPDK 关键技术分析

SPDK 开发组件一是关于存储，二是关于高性能的。SPDK 提供 一系列的工具和库来对实现存储的应用。最大的特点就是，SPDK 是完全存在于用户态的，无论是驱动还是框架都采用轮询代替了中断，并且数据的搬迁都是“零拷贝”的。 基于这些特点，SPDK 的性能较内核态相比，有了很大的提高。

1. SPDK 主要特性

- 将所用必须的驱动移入到用户态，避免了 syscalls 系统调用，并实现了用户程序的 0 拷贝;
- 使用轮询机制轮询硬件操作完成状态，替代了中断机制
- 通过消息的传输，在 IO 路径中避免了所有的锁

2. SPDK 其他特性

- SPDK 主要实现的基石就是用户态、轮询机制、无锁、异步的 NVME 驱动
- 提供类似于内核 Block 层的机制与相同的接口
- SPDK 提供 NVMe-oF、iSCSI 和 vhost servers 的支持

17.1 Message 传递与并发

17.1.1 技术原理

传统上，软件在 并发时，经常将共享数据放置在堆中，使用锁来进行保护，当需要访问共享数据时则需要获取所来进行后续操作。这样的模型会有下列明显的属性：

- 可以相对轻松的将单个进程转换为多进程的模型，因为不需要多加其他操作，只需要在数据附近加入一把锁即可
- 进程可以被打断并被休眠
- 当进程增加越来越多，共享数据越来越多时，锁的数量和实现机制则会越来越多，越来越复杂。并且多进程对锁获取的尝试次数也会增加，因此会增加系统消耗的时间。

以此，SPDK 采用了完全不同的涉及。它将所有进程需要获取锁来访问的共享数据，分派给了每一个独立的进程。当进程需要访问数据时，会传递一个消息给已获取到数据的进程，要求交出维护权。在 Message 中通常包含了 a function pointer 和一个指向 context 的指针，并使用一个 lockless ring 被传递在进程之间。

17.1.2 消息传递基础架构

First，spdk_thread 是执行进程的一个抽象，spdk_poller 是指定进程定期执行函数的抽象。当需要使用到 SPDK 时，需要调用 spdk_allocate_thread()来获取执行的 thread。

动态库中也提供了两个抽象：spdk_io_device 和 spdk_io_channel。spdk_io_channel 就是下发 IO 的通道 ，与 thread 相关，而 spdk_io_device 则是处理 IO 的设备的抽象指针。

17.1.3 事件架构介绍

SPDK 提供了 SPDK 事件框架,这个库处理设置所有的消息传递，通过设置 Handle 机制关闭信号处理程序，实现了周期轮询器，并完成基本的命令行解析。当开始调用 spdk_app_start()时，建架构会自动开启所有的进程请求并链接他们，使用合适的函数指针来调用 spdk_allocate_thread()。以此来实现全新的 spdk 应用。

同时，SPDK 的框架中有三个重要的概念：reactors、events 和 pollers。Events framework 会在每个 core 上生成一个进程（reactor）。消息（events）可以在进程中进行传递。Event Framework 的公共接口声明在 event.h 中。

Events

Event Framework 在每个 CPU core 上运行一个 event 轮询进程。这些进程被称为 reactors，主要的作用则是处理队列中发送进来的 event。一个 event 中包含了一捆函数指针和函数的参数，并指定了特定的 CPU core。

Events 通过 spdk_event_allocate()创建

Events 通过 spdk_event_call()执行

Reactors

每个 reactor 有一个无锁队列来处理发送到这个 core 中的 event，其他任何 core 上的进程可以插入 events 到任何一个 core 上的队列里。Reactor 循环运行在每个 core 上来检查进入的 events 并且以 FIFO 顺序进行执行。Event 的功能函数应该永远不会阻塞并且执行迅速。

Pollers

Pollers 由函数 spdk_poller_register()注册。Pollers 类似 events，其中捆绑的函数和参数都可以被执行。但是与 events 不同的是，pollers 是一直重复运行在它所注册的进程上的，直至它被注销掉才停止运行。Pollers 会一直轮询硬件设备。Pollers 可以一直循环执行,也可以通过设置 timer 来间歇执行。

Application Framework

应用架构是更高层的一种抽象，被称为”app”。一旦 spdk_app_start()被调用,它会block 当前进程直到 spdk_app_stop()被调用或者 spdk_app_start()自己执行有误。

17.2 SPDK 用户态内存管理

NVMe 设备传输数据与系统内存使用直接存储器访问(DMA)。具体地说,他们在整个 PCI 总线发送消息请求数据传输。数据传输发生没有涉及 CPU,MMU 负责访问内存的。

SPDK 依靠 DPDK 分配固定内存。在 Linux 上,DPDK 通过分配 hugepages(默认情况下,2 M)。Linux 内核的区别对待 hugepages 比普通 4KB 页面。具体来说,操作系统永远不会改变他们的物理位置。在未来的版本中可能会改变,但是今天确实已经沿用数年(参见后面的一节 IOMMU 技术解决方案)。DPDK 煞费苦心分配 hugepages,这样它可以串很长运行的物理页面,这样可以连续分配大于单个页面的空间。

通过这个解释,希望现在能清楚解释为什么所有数据缓冲区传递给 SPDK 必须使用 spdk_dma_malloc ()分配空间。

SPDK 的接口声明在 include/spdk/env.h

DPDK 的接口声明在 lib/env_dpdk

17.3 块设备层编程

SPDK block device layer，通常被称为 bdev，主要提供了下列功能：

- 一个可插拔模块的 API，实现不同块存储设备的块设备接口
- NVMe、malloc、linux AIO、virtio-scsi、Ceph RBD、Pmem 和 Vhost-SCSI 驱动模块
- 提供例如 read、write、unmap 等 SPDK 块设备上的应用 API
- 通过 JSON-RPC 配置块设备
- 请求的排队、超时和重置管理
- 块设备下发 IO 的无锁多队列

用户可以使用现有的 bdev 模块，或者创建自己的模块在任何类型的设备上。SPDK 也提供 vbdev 模块，来创建块设备在已存在的 bdev 上。

先决条件：块设备层是 C 库代码，并且接口公开在 bdev.h 文件中。下列 SPDK 的配置描述都是通过使用 JSON-RPC 命令完成的。SPDK 提供了一个 python-based 命令行工具来发送 RPC 命令（scripts/rpc.py）。
通用 RPCs：使用 RPC 命令 get_bdevs 可以获取到所有可用的设备的信息列表。同时通过 name 来指定设备时，使用此命令可以查看到这个设备的详细信息。

使用 RPC 命令 delete_bdev 可以移除已经创建的 bdev 设备。通过 name 来指定设备。

17.4 编写 Blcok 设备模块

Block Device Module 是 SPDK 中与内核 dvice drive 对等的意义存在。已经提供了一系列的函数来 service 块设备 I/O 请求。支持了很多设备例如 NVMe、RAM-disk 和 Ceph RBD。

17.4.1 创建一个新的组件

块设备组件都是放在 lib/bdev/<module_name>中的，可以看到当前 SPDK 代码中大致分为 nvme、null、rbd、aio 等等。创建一个新的组件则在此处新建一个组件名称的目录，放入 C 和 Makefile。

17.4.2 创建 bdevs

当组件调用 spdk_bdev_register()是会创建新的 bdevs。此组件必须要申请一个 spdk_bdev 结构体并正确的填充它，然后把结构体传递到 register call 中。一定要填 fn_talbe！并且新的组件要时间这些回调函数！

新组件中需要实现 spdk_bdev_io_type 中的 IO 操作类型，在最简单的组件实现中,仅 SPDK_BDEV_IO_TYPE_READ 和 SPDK_BDEV_IO_TYPE_WRITE 需要实现。

SPDK_BDEV_IO_TYPE_NVME_ADMIN, SPDK_BDEV_IO_TYPE_NVME_IO, and SPDK_BDEV_IO_TYPE_NVME_IO_MD 是传递未加工的 NVMe 命令的 IO 类型。

块设备层编程指导：http://www.spdk.io/doc/bdev_pg.html

17.5 JSON-RPC 服务介绍

SPDK 实现了一个 JSON-RPC 2.0 服务端来允许外部管理工具动态配置 SPDK 的设置。SPDK 关于 JSON-RPC 的描述：http://www.spdk.io/doc/jsonrpc.html

17.6 NVME 驱动介绍

NVMe 设备驱动的动态库通过映射 PCI BAR 以及使用 Memory-mapped I/O 来实现对 NVMe 设备的控制和操作。驱动接口定义在 spdk/nvme.h 中。

IO 通过 nvme_ns_cmd_xxx 函数的调用来被提交到 NVME 的一个 namespace 中。NVMe 驱动通过命令提交 IO 请求到指定的请求队列中。Spdk_nvme_qpair_process_completions()函数被轮询调用用来查询 IO 的完成状态。

17.7 NVMe 热插拔技术

在 NVMe 驱动层，SPDK 提供了一些关于热插拔的支持：

热插拔事件探测：用户可以定期调用 NVMe 库中的 spdk_nvme_probe()来探测发现热插拔事件。Probe_cb 函数会在新设备被探测到时调用。用户也可以提供一个 remove_cb 来完成设备被移除出系统时的操作。
spdk_nvme_probe、spdk_nvme_ctrlr_process_admin_completions

当 IO 正在传输时，如果设备被热拔，所有对 PCI BAR 的访问都会返回一个 SIGBUS 错误。NVMe 驱动会自动处理这种情况，通过安装一个 SIGBUS 处理器和重映射 PCI BAR 到一块新的内存。

参考
http://www.spdk.io/doc/index.html