1. 简介

本文档提供软件架构信息，开发环境及优化方案。
有关编程示例以及如何编译运行这些示例，请参阅《DPDK 示例用户指南》。
有关编译运行应用程序的基本信息，请参阅《DPDK 入门指南》。

1.1. 文档路线图

以下是一份建议顺序阅读的 DPDK 参考文档列表:

- **发行公告**：提供特定发行版本的信息，包括支持的特性、限制条件、修复的问题、已知的问题等等。此外，还以 FAQ 的方式提供了常见问题的解决方法。
- **入门指南**：介绍如何安装及配置 DPDK 软件，旨在帮助用户快速上手。
- **FreeBSD* 入门指南**：DPDK1.6.0 发布版本之后添加了 FreeBSD* 平台上的入门指南。有关如何在 FreeBSD* 上安装配置 DPDK，请参阅这个文档。
- **编程指南**（本文档），描述了如下内容：
  - 软件架构以及如何使用（示例介绍），特别是在 Linux 用户环境中的使用
  - DPDK 的主要内容，系统构建（包括可以在 DPDK 根目录 Makefile 中用来构建工具包和应用程序的命令）及应用移植细则。
  - 软件中使用的，以及新开发中需要考虑的一些优化。
- **API 参考**：提供有关 DPDK 功能、数据结构和其他编程结构的详细信息。
- **示例程序用户指南**：描述了一组例程。每个章节描述了一个用例，展示了具体的功能，并提供了有关编译、运行和使用的说明。

1.2. 相关刊物

以下文档提供了与使用 DPDK 开发应用程序相关的信息：

- Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3A: System Programming Guide

2. 概述

本章节给出了 DPDK 架构的一个全局概述。

DPDK 的主要目的就是为数据面快速报文处理应用程序提供一个简洁完整的框架。用户可以通过代码来理解其中使用的一些技术，构建自己的应用程序或添加自己的协议栈。 Alternative ecosystem options that use the DPDK are available。

通过创建环境抽象层（EAL），DPDK 框架为每个特殊的环境创建了一组运行库。这个库特定于 Intel 架构（32 或 64 位），Linux*用户空间编译器或其他特定的平台。这些环境通过一些 makefile 和配置文件创建。一旦 EAL 库编译完成，用户可以通过链接这些库来构建自己的应用程序。除了 EAL，还有一些其他的库，包括哈希算法、最长前缀匹配、环形缓冲区等。DPDK 提供了一些应用程序实例来指导如何使用这些特定来创建自己的应用程序。

DPDK 实现了报文处理的 RTC 模型，在这种模型中，数据面应用程序在调用之前必须预先分配好所有的资源，并作为执行单元运行在逻辑核上。这种模型并不支持调度，且所有的设备通过轮询的方式访问。不使用中断方式的主要原因就是中断处理增加了性能开销。

作为 RTC 模型的扩展，通过使用 Ring 在不同逻辑核之间传递报文和消息，也可以实现报文处理的流水线模型（Pipeline）。流水线模型允许操作分阶段进行，在多核代码执行中可能更高效。

2.1. 开发环境

DPDK 工程创建要求 Linux 环境及相关的工具链，例如一个或多个编译工具、汇编程序、 make 工具、编辑器及 DPDK 组件用到的库。

当指定环境和架构的库编译出来时，这些库就可以用于创建我们自己的数据面处理程序。

创建 Linux 用户空间应用程序时，需要用到 glibc 库。对于 DPDK 应用程序，必须使用两个全局环境变量（RTE_SDK 和 RTE_TARGET），这两个变量需要在编译应用程序之前配置好：

```
export RTE_SDK=/home/user/DPDK
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

也可以参考《DPDK 入门指南》来获取更多搭建开发环境的信息。

2.2. 环境适配层

环境适配层为应用程序和库提供了通用的接口，隐藏了底层环境细节。EAL 提供的服务有：

- DPDK 的加载和启动
- 多线程和多进程执行方式支持
- CPU 亲和性设置
- 系统内存分配和释放
- 原子操作和锁操作

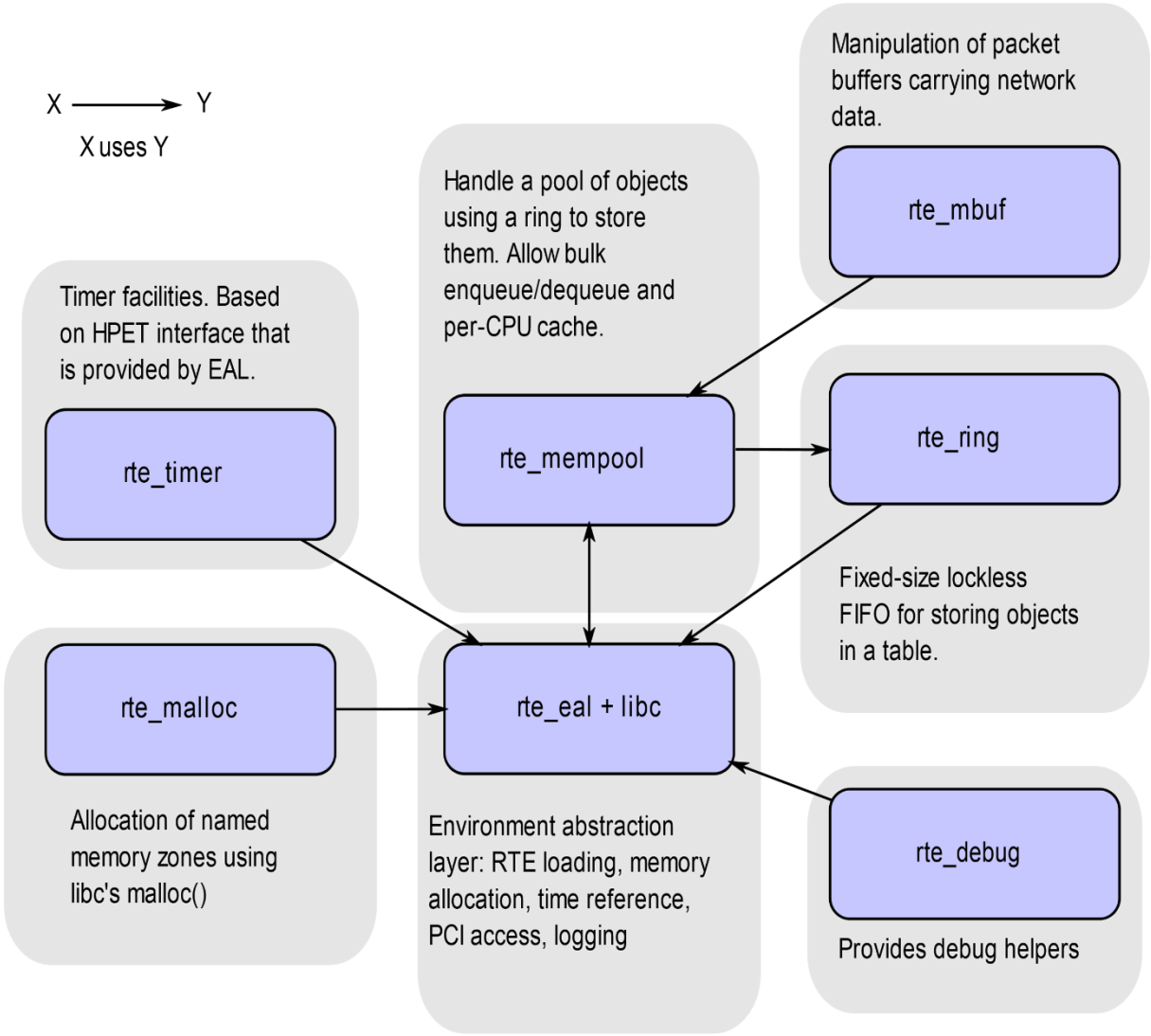
- 定时器引用
- PCI 总线访问
- 跟踪调试功能
- CPU 特性识别
- 中断处理
- 警告操作
- 内存管理

EAL 更详细的描述请参阅本文档“环境适配层”章节。

2.3. 核心组件

核心组件指的是一系列库，用于为高性能包处理程序提供所有必须的元素。核心组件及其之间的关系如下图所示：

Figure 2-1 Core Components Architecture



2.3.1. 环形缓冲区管理（librte_ring）

Ring 数据结构提供了一个无锁的多生产者，多消费者的 FIFO 表处理接口。相对于无锁队列来讲，它容易部署，适合大量的操作，而且更快。Ring 库在“内存池库（librte_mempool）”中使用，而且，ring 还用于不同逻辑核上处理单元之间的通信。

环形缓冲区及其使用可以参考章节“环形缓冲区库”描述。

2.3.2. 内存池管理（librte_mempool）

内存池管理的主要职责就是在内存中分配指定数目对象的 Pool。每个 Pool 以名称来唯一标识，并且使用一个 Ring 来存储空闲的对象节点。它还提供了一些其他的服务，如对象节点的每核缓存备份，及自动对齐以保证对象能够均衡分布到内存通道上。

内存池分配器的具体行为请参考章节“内存池库”描述。

2.3.3. 网络报文缓冲区管理（librte_mbuf）

报文缓冲区库提供了创建和销毁报文缓冲区的能力，DPDK 应用程序中使用这些缓冲区来存储消息。这些缓冲区通常在程序开始时通过 DPDK 的内存池库（librte_mempool）申请并存储在内存池中。缓冲区库（librte_mbuf）提供了报文申请和释放的 API，通常情况下，消息 Buffer 用于缓存消息，报文 Buffer 用于缓存网络报文。

报文缓冲区管理的具体行为请参考章节“缓冲区库”描述。

2.3.4. 定时器管理（librte_timer）

这个库位 DPDK 的执行单元提供了定时器服务，为函数异步执行提供支持。定时器可以设置成周期调用，或者只调用一次。使用 EAL 提供的接口可以获取高精度时钟，并且能在每个核上根据需要初始化。

具体请参考章节“定时器库”描述。

2.4. 以太网轮询模式驱动架构

DPDK 的 PMD 驱动支持 1G、10G、40G。 同时 DPDK 提供了虚拟的以太网控制器，被设计成非异步，基于中断信号的模式。

详细内容参考 章节“轮询模式驱动”描述。

2.5. 报文转发算法支持

DPDK 提供了哈希（librte_hash）、最长前缀匹配（librte_lpm）算法库用于支持相应的分组转发算法。

详细内容查看章节“哈希算法”和“最长前缀匹配”。

2.6. 网络库（librte_net）

这个库包括 IP 协议的一些定义及常见的宏定义。这些定义都是基于 FreeBSD*中 IP 协议栈的代码，包括协议号（用于 IP 头部）、IP 相关的宏、IPv4/IPv6 头部结构体以及 TCP、UDP 和 STCP 头部结构体。

3. 环境抽象层

环境抽象层（EAL）为底层资源如硬件和存储空间的访问提供了接口。这些接口为上层应用程序和库隐藏了不同环境的特殊性。初始化程序负责决定如何分配这些资源（即内存空间、PCI 设备、计时器、控制台等托）。

EAL 提供的服务如下：

- DPDK 的加载和启动：DPDK 和特定的应用程序链接成一个独立进程，并以某种方式加载。
- CPU 亲和性和分配处理：EAL 提供了将执行单元分配给特定 Core 及创建执行实例的机制。
- 系统内存预留：EAL 实现了不同区域内存预留，例如用于设备交互的物理内存。
- PCI 地址抽象：EAL 提供了对 PCI 地址空间的访问接口
- 跟踪调试功能：日志信息，堆栈打印、异常挂起等等。
- 公用功能：提供了标准 Libc 库缺失的自旋锁、原子计数器等。
- CPU 特征识别：运行时确定是否支持指定功能，如 Intel AVX。确定当前 CPU 是否支持二进制编译的功能集。
- 中断处理：提供接口用于向中断注册/解注册中断处理回调函数。
- 告警功能：提供接口用于设置/取消指定时间环境下运行的毁掉函数。

3.1. Linux 用户执行环境中的 EAL

在 Linux 用户空间环境中，DPDK 应用程序通过 pthread 库作为一个用户态程序运行。设备的 PCI 信息和地址空间通过/sys 内核接口及内核模块如 uio_pci_generic 或 igb_uio 来发现的。详细信息请参阅 Linux 内核文档中 UIO 描述，设备的 UIO 信息是在程序中用 mmap 重新映射的。

EAL 使用 mmap 接口从 hugetlb 中实现物理内存的分配。这部分内存暴露给 DPDK 服务层，如 内存池库。

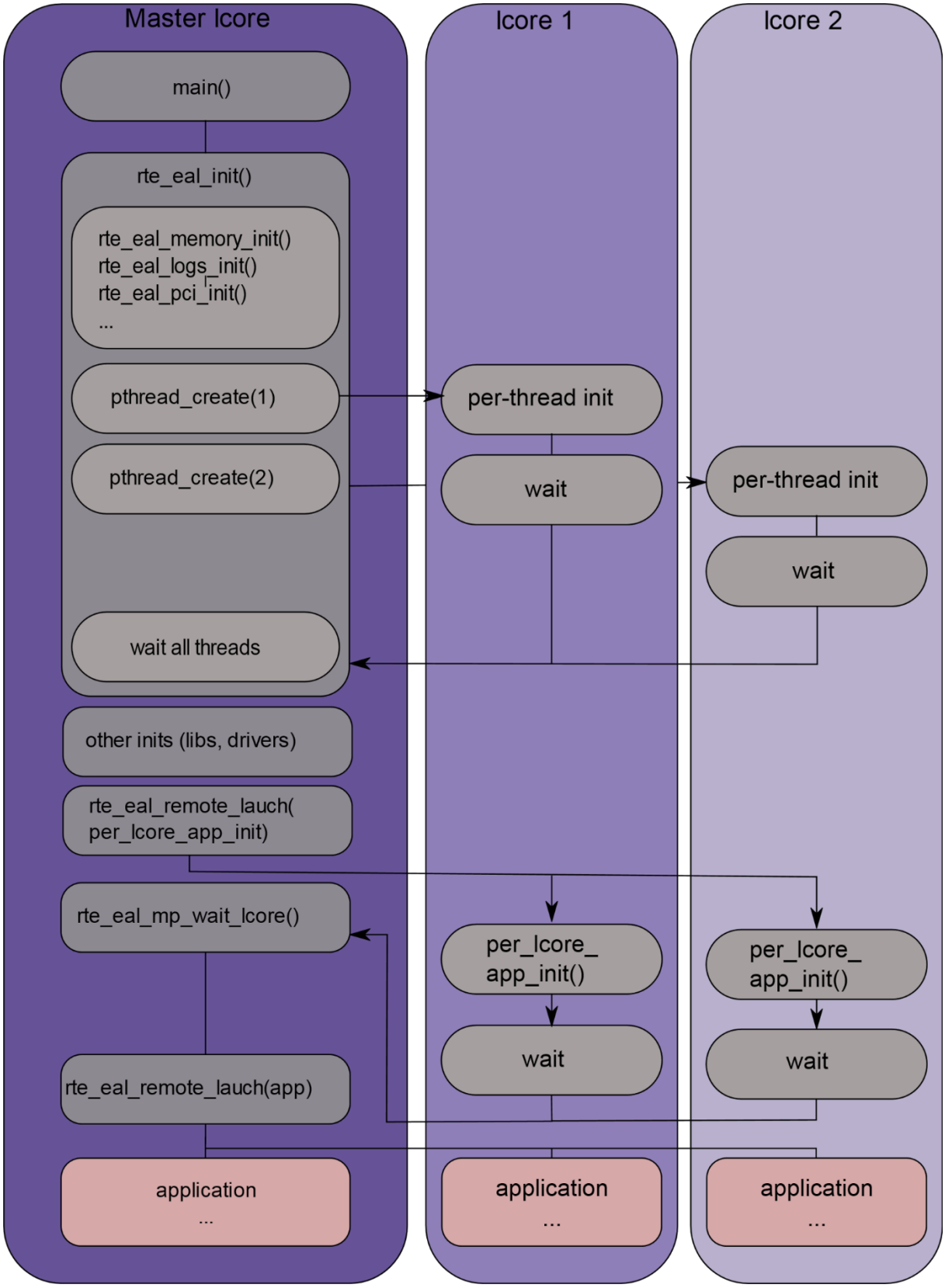
据此，DPDK 服务层可以完成初始化，接着通过设置线程亲和性调用，每个执行单元将会分配给特定的逻辑核，以一个 user-level 等级的线程来运行。

定时器是通过 CPU 的时间戳计数器 TSC 或者通过 mmap 调用内核的 HPET 系统接口实现。

3.1.1. 初始化及核心启动

部分初始化操作从 Glibc 的开始函数处就执行了。初始化过程中还执行一个检查，用于保证配置文件所选择的微架构类型是本 CPU 所支持的，然后才开始调用 main()函数。Core 的初始化和运行是在 rte_eal_init()接口上执行的（参考 API 文档）。它包括对 pthread 库的调用（更具体的说是 pthread_self(), pthread_create()和 pthread_setaffinity_np()）。

Figure 3-2 EAL Initialization in a Linux Application Environment



注意：
对象的初始化，例如内存区间、ring、内存池、lpm 表或 hash 表等，必须作为整个程序初始化的一部分，在主逻辑核上完成。 创建和初始化这些对象的函数不是多线程安全的，但是，一旦初始化完成，这些对象本身可以作为安全线程运行。

3.1.2. 多进程支持

Linuxapp EAL 允许多进程和多线程部署模式。详细信息请参阅“多进程支持”章节描述。

3.1.3. 内存映射发现及内存预留

大型连续的物理内存分配是通过 hugetlbfs 内核文件系统来实现的。EAL 提供了相应的接口用于预留指定名字的连续内存空间。这个 API 同时会将这段连续空间的地址返回给用户程序。

注意：
内存申请是使用 rte_malloc 接口来做的，它也是 hugetlbfs 文件系统大页支持的。

3.1.4. 无 Huge-TLB 的 Xen Domain 0 支持

现有的内存管理是基于 Linux 内核的大页机制。然而，Xen Dom0 并不支持大页，所以要将一个新的内核模块 rte_dom0_mem 加载上，以便避开这个限制。

EAL 使用 IOCTL 接口用于通告 Linux 内核模块 rte_mem_dom0 去申请指定大小的内存块，并从该模块中获取内存段的信息。EAL 使用 MMAP 接口来映射这段内存。对于申请到的内存段，在其内的物理地址都是连续的，但是实际上，硬件地址只在 2M 内连续。

3.1.5. PCI 访问

EAL 使用 Linux 内核提供的文件系统/sys/bus/pci 来扫描 PCI 总线上的内容。 内核模块 uio_pci_generic 提供了/dev/uioX 设备文件及/sys 下对应的资源文件用于访问 PCI 设备。 DPDK 特有的 igb_uio 模块也提供了相同的功能用于 PCI 设备的访问。 这两个驱动模块都用到了 Linux 内核提供的 uio 特性（用户空间驱动）。

3.1.6. 每逻辑核变量和共享变量

注意：

逻辑核就是处理器的逻辑单元，有时也称为硬件线程。

默认的做法是使用共享变量。每逻辑核变量的实现则是通过线程局部存储技术 TLS 来实现的，它提供了每个线程本地存储的功能。

3.1.7. 日志

EAL 提供了日志信息接口。默认情况下，在 Linux 应用程序中，日志信息被发送到 syslog 和 console 中。当然，用户可以通过使用不同的日志机制来重写 DPDK 中的日志函数。

3.1.7.1. 跟踪与调试功能

Glibc 中提供了一些调试函数用于打印堆栈信息。Rte_panic()函数可以产生一个 SIG_ABORT 信号，这个信号可以触发产生 coredump 文件，我们可以通过 gdb 来加载调试。

3.1.8. CPU 特性标识

EAL 可以在运行时查询 CPU 状态（使用 rte_cpu_get_feature()接口），用于判断哪个 CPU 特性可用。

3.1.9. 用户空间中断事件

3.1.9.1. 主机线程中的用户空间中断和报警处理

EAL 创建一个主机线程用于轮询 UIO 设备描述文件描述符以检测中断。可以通过 EAL 提供的函数为特定的中断事件注册或注销回调函数，回掉函数在主机线程中被异步调用。EAL 同时也允许像 NIC 中断那样定时调用中断处理回调。

注意：
在 DPDK 的 PMD 中，主机线程只对连接状态改变的中断处理，例如网卡的打开和关闭，以及设备突然移除中断。

3.1.9.2. RX 中断事件

PMD 提供的报文收发程序并不只限制于轮询模式下执行。为了缓解小吞吐量下轮询模式对 CPU 资源的浪费，暂停轮询并等待唤醒事件发生是一种有效的手段。收包中断是这种场景的一种很好的选择，当然也不是唯一的。

EAL 为事件驱动模式提供了相关的 API。以 Linuxapp 为例，其实现依赖于 epoll 技术。每个线程可以监控一个 epoll 实例，而在实例中可以添加所有需要的 wake-up 事件文件描述符。事件文件描述符根据 UIO/VFIO 规范创建并映射到指定的中断向量上。从 bspapp 角度看，可以使用 kqueue 来代替，但是目前尚未实现。

EAL 初始化中断向量和事件文件描述符之间的映射关系，同时每个设备初始化中断向量和队列之间的映射关系，这样，EAL 实际上并不知道在指定向量上发生的中断，由设备驱动负责执行后面的映射。

注意：
每队列 RX 中断事件只有 VFIO 模式支持，VFIO 支持多个 MSI-X 向量。在 UIO 中，RX 中断和其他中断共享中断向量，因此，当 RX 中断和 LSC（连接状态改变）中断同时发生时（(intr_conf.lsc == 1 && intr_conf.rxq == 1），只有前者才有能力区分。

RX 中断由 API（rte_eth_dev_rx_intr_*）来实现控制、使能、关闭。当 PMD 不支持时，这些 API 返回失败。Intr_conf.rxq 标识用于打开每个设备的 RX 中断。

3.1.9.3. 设备移除事件

当总线上的设备被移除时就出发该事件。设备底层资源可能不再可用（即 PCI 映射未完成）。PMD 必须保证在这种情况下，应用程序仍然可以安全地使用其中断回调。

可以使用链接状态改变中断事件相同的方式来订阅这个中断事件。执行上下文是相同的，即专用的中断线程。

考虑到，应用程序可能想要关闭发出设备删除事件的设备，在这种情况下，调用 rte_eth_dev_close() 可能触发它注销自己的设备删除事件回调。因此，必须注意不要在中断处理程序上下文中关闭设备 。必须重新安排这种关闭操作。

3.1.10. 黑名单

EAL PCI 设备的黑名单功能是用于标识指定的 NIC 端口，以便 DPDK 忽略该端口。可以使用 PCIe 设备地址描述符（Domain:Bus:Device:Function）将对应端口标记为黑名单。

3.1.11. 杂项功能

每个架构不同的锁和原子操作（i686 和 x86_64）。

3.2. 内存段和内存区域

物理内存映射就是通过 EAL 的这个特性实现的。物理内存块之间可能是不连续的，所有的内存通过一个内存描述符表进行管理，且表中的每个描述符指向一块连续的物理内存。

基于此，内存区域分配器的作用就是保证分配到一块连续的物理内存。 这些区域被分配出来时会用一个唯一的名字来标识。

Rte_memzone 描述符也在配置结构体中，可以通过 rte_eal_get_configuration()接口来获取。 通过名字访问一个内存区域会返回对应内存区域的描述符。

内存分配可以从指定开始地址和对齐方式来预留（默认是 cache line 大小对齐），对齐一般是以 2 的次幂来的，并且不小于高速缓存行的大小（64 字节）对齐。内存区域也可以从 2M 或 1G 大小的内大页内存中获取，这两者系统都支持。

3.3. 多线程

DPDK 通常为每个 Core 指定一个线程，以避免任务切换的开销。这有利于性能的提升，但不总是有效的，并且缺乏灵活性。

电源管理通过限制 CPU 的运行频率来提升 CPU 的工作效率。当然，我们也可以通过充分利用 CPU 的空闲周期来使用 CPU 的全部功能。

通过使用 cgroup 技术，CPU 的使用量可以很方便的分配，这也提供了新的方法来提升 CPU 性能，但是这里有个前提，DPDK 必须处理每个核上多个线程的上下文切换。

想要更多的灵活性，就要设置线程的 CPU 亲和性是针对对 CPU 集合而不是 CPU 了。

3.3.1. EAL 线程与逻辑核亲和性

术语“lcore”指一个 EAL 线程，这是一个真正意义上的 Linux/FreeBSD pthread。“EAL pthread”由 EAL 创建和管理，并执行 remote_launch 发出的任务。在每个 EAL pthread 中，有一个称为_lcore_id 的 TLS（线程本地存储）用于唯一标识线程。由于 EAL pthread 通常将物理 CPU 绑定为 1：1，所以_locore_id 通常等于 CPU ID。

但是，当使用多线程时，EAL pthread 和指定的物理 CPU 之间的绑定不再总是 1：1 了。EAL pthread 可能与一组 CPU 相关，因此_lcore_id 将不同于 CPU ID。基于这个原因，EAL 有一个运行参数选项“-lcores”用来定义分配的 CPU 亲和性。对于执行的 lcore ID 或 ID 组，该选项允许设置该 EAL pthread 的 CPU 组。

设置格式如下：

```
注意：
-lcores='<lcore_set>[@cpu_set][,<lcore_set>[@cpu_set],…]"
其中 lcore_set 和 cpu_set 可以是单个数值，区间或者组。
数值可以是“digit([0-9]+)”
区间可以是“<number>-<number>”
组可以是“(<number|range>[,<number|range>,...])”
```

如果‘@cpu_set’值未指定，‘cpu_set’的值默认与‘lcore_set’相等。

举例："--lcores='1,2@(5-7),(3-5)@(0,2),(0,6),7-8'" 表示启动了 9 个 EAL pthread：

lcore 0 运行于 CPU 组 0x41，也就是 CPU（0，6）
lcore 1 运行于 CPU 组 0x2，也就是 CPU（1）
lcore 2 运行于 CPU 组 0xe0，也就是 CPU（5，6，7）
lcore 3-5 运行于 CPU 组 0x5 也就是 CPU（0，2）
lcore 6 运行于 CPU 组 0x41，也就是 CPU（0，6）
lcore 7 运行于 CPU 组 0x80，也就是 CPU（7）
lcore 8 运行于 CPU 组 0x100，也就是 CPU（8）

使用这个选项，对于给定的 lcore ID，可以分配对应的 CPU 组。它也兼容 corelist（' - l'）选项的模式。

3.3.2. 非 EAL 线程支持

可以在任何用户线程（non-EAL 线程）上执行 DPDK 任务上下文。 在 non-EAL pthread 中，*_lcore_id* 始终是 LCORE_ID_ANY，它标识一个 no-EAL 线程的有效、唯一的 *_lcore_id*。有些库可会使用一个唯一的 ID 替代（如 TID），有些库将不受影响，有些库则会受到限制（如定时器和内存池库）。

所有这些影响将在“已知问题”章节中提到。

3.3.3. 公用线程 API

DPDK 为线程操作引入了两个公共 API rte_thread_set_affinity() 和 rte_pthread_get_affinity()。当他们在任何线程上下文中调用时，将获取或设置线程本地存储(TLS)。

这些 TLS 包括 _cpuset 和 _socket_id：

- _cpuset 存储了与线程亲和的 CPU 位图。
- _socket_id 存储了 CPU set 所在的 NUMA 节点。如果 CPU set 中的 cpu 属于不同的 NUMA 节点，_socket_id 将设置为 SOCKET_ID_ANY。

3.3.4. 已知问题

- rte_mempool

  rte_mempool 在 mempool 中使用 per-lcore 缓存。对于 non-EAL pthread，rte_lcore_id() 无法返回一个合法的值。因此，当 rte_mempool 与 non-EAL 线程一起使用时，put/get 操作将绕过默认的 mempool 缓存，这个旁路操作将造成性能损失。结合 rte_mempool_generic_put()和 rte_mempool _generic_get() 可以在 non-EAL 线程中使用用户拥有的外部缓存。

- rte_ring

  rte_ring 支持多生产者入队和多消费者出队操作。 然而，这是非抢占的，这使得 rte_mempool 操作都是非抢占的。

  注意：
  “非抢占” 意味着：
  1. 在给定的 ring 上做入队操作的 pthread 不能被另一个在同一个 ring 上做入队的 pthread 抢占
  2. 在给定 ring 上做出对操作的 pthread 不能被另一个在同一 ring 上做出队的 pthread 抢占

  绕过此约束则可能造成第二个进程自旋等待，知道第一个进程再次被调度为止。 此外，如果第一个线程被优先级较高的上下文抢占，甚至可能造成死锁。

  这并不意味着不能使用它，简单讲，当同一个 core 上的多线程使用时，需要缩小这种情况.
  1. 它可以用于任一单一生产者或者单一消费者的情况。
  2. 它可以由多生产者/多消费者使用，要求调度策略都是 SCHED_OTHER(cfs)。用户需要预先了解性能损失。
  3. 它不能被调度策略是 SCHED_FIFO 或 SCHED_RR 的多生产者/多消费者使用。
- rte_timer

  不允许在 non-EAL pthread 上运行 rte_timer_manager()。但是，允许在 non-EAL pthread 上重置/停止定时器。
- rte_log

  在 non-EAL pthread 上，没有 per thread loglevel 和 logtype，但是 global loglevels 可以使用。
- Misc

  在 non-EAL pthread 上不支持 rte_ring, rte_mempool 和 rte_timer 的调试统计信息。

3.3.5. cgroup 控制

以下是 cgroup 控件使用的简单示例，在同一个核心($CPU)上两个线程(t0 and t1)执行数据包 I/O。我们期望只有 50%的 CPU 消耗在数据包 IO 操作上。

```
mkdir /sys/fs/cgroup/cpu/pkt_io

mkdir /sys/fs/cgroup/cpuset/pkt_io

echo $cpu > /sys/fs/cgroup/cpuset/cpuset.cpus

echo $t0 > /sys/fs/cgroup/cpu/pkt_io/tasks
echo $t0 > /sys/fs/cgroup/cpuset/pkt_io/tasks

echo $t1 > /sys/fs/cgroup/cpu/pkt_io/tasks
echo $t1 > /sys/fs/cgroup/cpuset/pkt_io/tasks

cd /sys/fs/cgroup/cpu/pkt_io
echo 100000 > pkt_io/cpu.cfs_period_us
```

```
echo  50000 > pkt_io/cpu.cfs_quota_us
```

3.4. 内存申请操作（Malloc）

EAL 提供了一个 malloc API 用于申请任意大小内存。

这个 API 的目的是提供类似 malloc 的功能，以允许从 hugepage 中分配内存并方便应用程序移植。《DPDK API 参考手册》详细介绍了接口的功能。

通常，这些类型的分配操作不应该在数据面处理中进行，因为他们比基于池的分配慢，并且在分配和释放路径中使用了锁操作。但是，他们可以在配置代码中使用。

更多信息请参阅《DPDK API 参考手册》中 rte_malloc()函数描述。

3.4.1. Cookies

当 CONFIG_RTE_MALLOC_DEBUG 开启时，分配的内存包括保护字段，这个字段用于帮助识别缓冲区溢出。

3.4.2. 对齐和 NUMA 限制

接口 rte_malloc()传入一个对齐参数，该参数用于请求在该值的倍数上对齐的内存区域 (这个值必须是 2 的幂次)。

在支持 NUMA 的系统上，对 rte_malloc()接口调用将返回在调用函数的 Core 所在的插槽上分配的内存。DPDK 还提供了另一组 API，以允许在指定 NUMA 插槽上直接显式分配内存，或者分配另一个 NUAM 插槽上的内存。

3.4.3. 用例

这个 API 旨在由初始化时需要类似 malloc 功能的应用程序调用。
需要在运行时分配/释放数据，在应用程序的快速路径中，应该使用内存池库。

3.4.4. 内部实现

3.4.4.1. 数据结构

Malloc 库中内部使用两种数据结构类型：

- struct malloc_heap：用于在每个插槽上跟踪可用内存空间
- struct malloc_elem：库内部分配和释放空间跟踪的基本要素

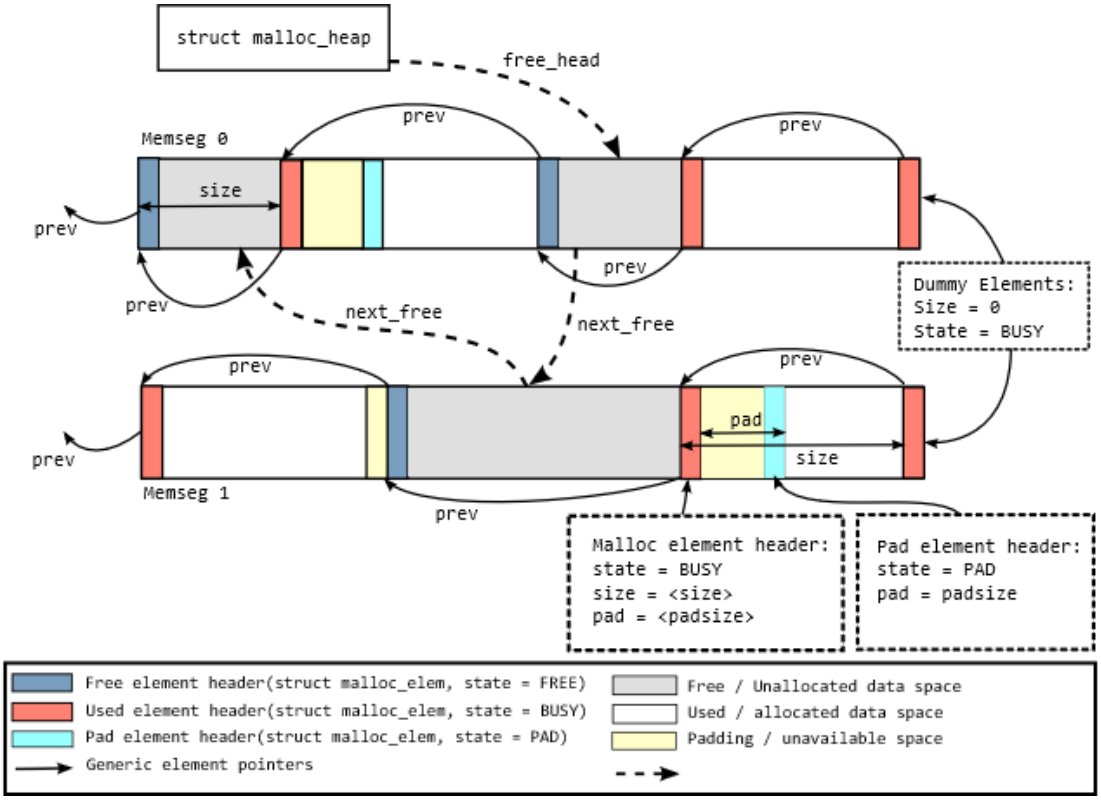
3.4.4.1.1. malloc_heap

数据结构 malloc_heap 用于管理每个插槽上的可用内存空间。在内部，每个 NUMA 节点有一个堆结构，这允许我们根据此线程运行的 NUMA 节点为线程分配内存。虽然这并不能保证在 NUMA 节点上使用内存，但是它并不比内存总是在固定或随机节点上的方案更糟。

堆结构及其关键字段和功能描述如下：

- lock：需要锁来同步对堆结构的访问。假定使用链表来跟踪堆中的可用空间，我们需要一个锁来防止多个线程同时处理该链表。
- free_head：指向这个 malloc 堆的空闲结点链表中的第一个元素。

Figure 3-3 Example of a malloc heap and malloc elements within the malloc library



注意：
数据结构 malloc_heap 并不会跟踪使用的内存块，因为除了要再次释放它们之外，它们不会 i 被接触，需要释放时，将指向块的指针作为参数传递给 free 函数。

3.4.4.1.2. malloc_elem

数据结构 malloc_elem 用作各种内存块的通用头结构。 它以三种不同的方式使用，如上图所示：

- 作为一个释放/申请内存的头部，正常使用
- 作为内存块内部填充头

- 作为内存结尾标记

结构中重要的字段和使用方法如下所述：

- heap：这个指针指向了该内存块从哪个堆申请。它被用于正常的内存块，当他们被释放时，将新释放的块添加到堆的空闲列表中。
- prev：这个指针用于指向紧跟着当前 memseg 的头元素。当释放一个内存块时，该指针用于引用上一个内存块，检查上一个块是否也是空闲。如果空闲，则将两个空闲块合并成一个大块。
- next_free：这个指针用于将空闲块列表连接在一起。它用于正常的内存块，在 malloc() 接口中用于找到一个合适的空闲块申请出来，在 free() 函数中用于将内存块添加到空闲链表。
- state：该字段可以有三个可能值：FREE, BUSY 或 PAD。前两个是指示正常内存块的分配状态，后者用于指示元素结构是在块开始填充结束时的虚拟结构，即，由于对齐限制，块内的数据开始的地方不在块本身的开始处。在这种情况下，pad 头用于定位块的实际 malloc 元素头。对于结尾的结构，这个字段总是 BUSY，它确保没有元素在释放之后搜索超过 memseg 的结尾以供其它块合并到更大的空闲块。
- pad：这个字段为块开始处的填充长度。在正常块头部情况下，它被添加到头结构的结尾，以给出数据区的开始地址，即在 malloc 上传回的地址。在填充虚拟头部时，存储相同的值，并从虚拟头部的地址中减去实际块头部的地址。
- size：数据块的大小，包括头部本身。对于结尾结构，这个大小需要指定为 0，虽然从未使用。对于正在释放的正常内存块，使用此大小值替代 “next” 指针，以标识下一个块的存储位置，在 FREE 情况下，可以合并两个空闲块。

3.4.4.2. 内存申请

在 EAL 初始化时，所有 memseg 都将作为 malloc 堆的一部分进行设置。这个设置包括在 BUSY 状态结束时放置一个虚拟结构，如果启用了 CONFIG_RTE_MALLOC_DEBUG，它可能包含一个哨兵值，并在开始时为每个 memseg 指定一个适当的元素头。然后将 FREE 元素添加到 malloc 堆的空闲链表中。

当应用程序调用类似 malloc 功能的函数时，malloc 函数将首先为调用线程索引 lcore_config 结构，并确定该线程的 NUMA 节点。NUMA 节点将作为参数传给 heap_alloc() 函数，用于索引 malloc_heap 结构数组。参与索引参数还有大小、类型、对齐方式和边界参数。

函数 heap_alloc()将扫描堆的空闲链表，尝试找到一个适用于所请求的大小、对齐方式和边界约束的内存块。

当已经识别出合适的空闲元素时，将计算要返回给用户的指针。紧跟在该指针之前的内存的高速缓存行填充了一个 malloc_elem 头部。由于对齐和边界约束，在元素的开头和结尾可能会有空闲的空间，这将导致已下行为：

- 检查尾随空间。如果尾部空间足够大，例如 > 128 字节，那么空闲元素将被分割。否则，仅仅忽略它（浪费空间）。
- 检查元素开始处的空间。如果起始处的空间很小， <=128 字节，那么使用填充头，这部分空间被浪费。但是，如果空间很大，那么空闲元素将被分割。

从现有元素的末尾分配内存的优点是不需要调整空闲链表，空闲链表中现有元素仅调整大小指针，

并且后面的元素使用 “prev” 指针重定向到新创建的元素位置。

3.4.4.3. 内存释放

要释放内存，将指向数据区开始的指针传递给 free 函数。从该指针中减去 malloc_elem 结构的大小，以获得内存块元素头部。如果这个头部类型是 PAD，那么进一步减去 pad 长度，以获得整个块的正确元素头。

从这个元素头中，我们获得指向块所分配的堆的指针及必须被释放的位置，以及指向前一个元素的指针，并且通过 size 字段，可以计算下一个元素的指针。 这意味着我们永远不会有两个相邻的 FREE 内存块，因为他们总是会被合并成一个大的块。

4. 环形缓冲区库

环形缓冲区支持队列管理。rte_ring 并不是具有无限大小的链表，它具有如下属性：

- 先进先出（FIFO）
- 最大大小固定，指针存储在表中
- 无锁实现
- 多消费者或单消费者出队操作
- 多生产者或单生产者入队操作
- 批量出队 - 如果成功，将指定数量的元素出队，否则什么也不做
- 批量入队 - 如果成功，将指定数量的元素入队，否则什么也不做
- 突发出队 - 如果指定的数目出队失败，则将最大可用数目对象出队
- 突发入队 - 如果指定的数目入队失败，则将最大可入队数目对象入队

相比于链表，这个数据结构的优点如下：

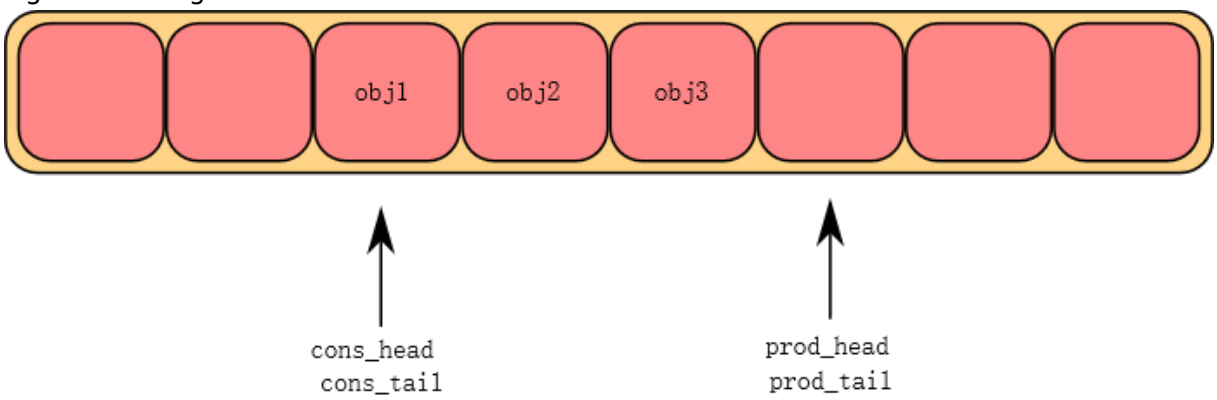
- 更快，只需要一个 sizeof(void *)的 Compare-And-Swap 指令，而不是多个双重比较和交换指令。
- 更像是一个完全无锁队列。
- 适应批量入队/出队操作。因为指针是存储在表中的，多个对象的出队将不会产生与链表队列中一样多的 cache miss。此外，批量出队成本并不比单个对象出队高。

缺点：

- 大小固定。
- 大量 ring 相比于链表，消耗更多的内存，空 ring 至少包含 n 个指针。

数据结构中存储的生产者和消费者头部和尾部指针显示了一个简化版本的 ring。

Figure 4-4 Ring Structure



4.1. FreeBSD*中环形缓冲区实现参考

FreeBSD 8.0 中添加了如下代码，并应用到了某些网络设备驱动程序中（至少 Interl 驱动中应用了）：

- bufring.h in FreeBSD
- bufring.c in FreeBSD

4.2. Linux*中的无锁环形缓冲区

参考链接 Linux Lockless Ring Buffer Design。

4.3. 附加特性

4.3.1. 名字

每个 ring 都有唯一的名字。 用户不可能创建两个具有相同名称的 ring（如果尝试调用 rte_ring_create()这样做的话，将返回 NULL）。

4.4. 用例

Ring 库的使用情况包括：

- DPDK 应用程序之间的交互
- 用于内存池申请

4.5. 环形缓冲区解析

本节介绍 ring buffer 的运行方式。Ring 结构有两组头尾指针组成，一组被生产者调用，一组被消费者调用。以下将简单称为 prod_head、prod_tail、cons_head 及 cons_tail。
每个图代表了 ring 的简化状态，是一个循环缓冲器。本地变量的内容在图上方表示，Ring 结构的内容在图下方表示。

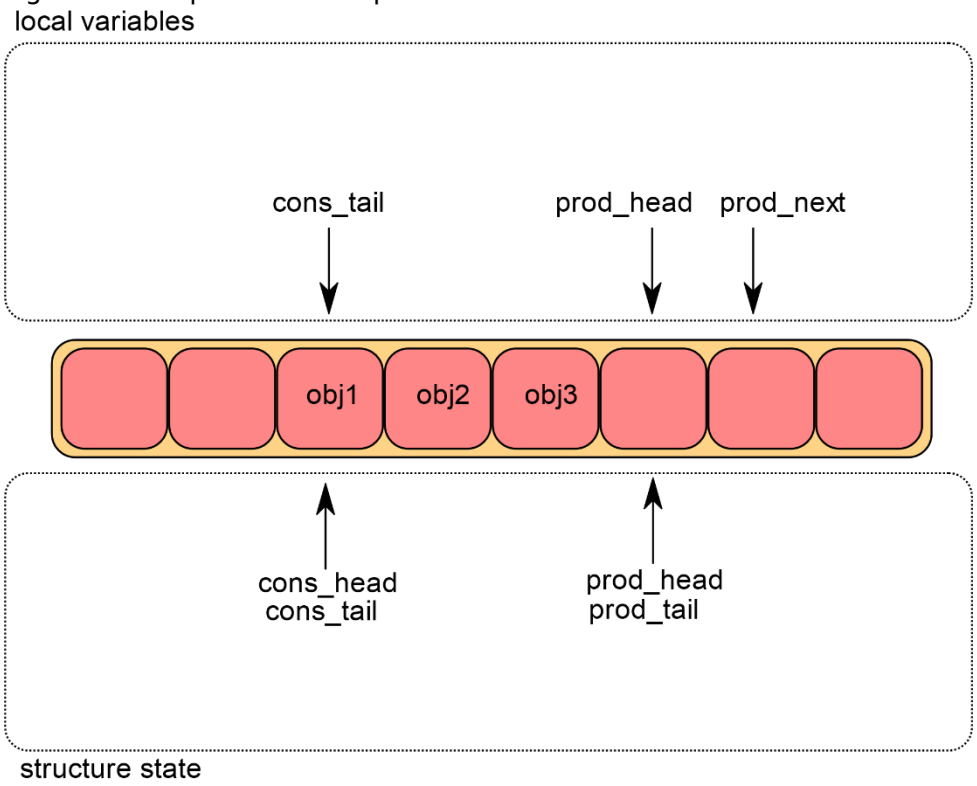
4.5.1. 单生产者入队

本节介绍了一个生产者向队列添加对象的情况。在本例中，只有生产者头和尾指针(prod_head and prod_tail)被修改，只有一个生产者。初始状态是将 prod_head 和 prod_tail 指向相同的位置。

4.5.1.1. 入队第一步

首先， ring->prod_head 和 ring->cons_tail*复制到本地变量中。*prod_next 本地变量指向下一个元素，或者，如果是批量入队的话，指向下几个元素。如果 ring 中没有足够的空间存储元素的话(通过检查 cons_tail 来确定)，则返回错误。

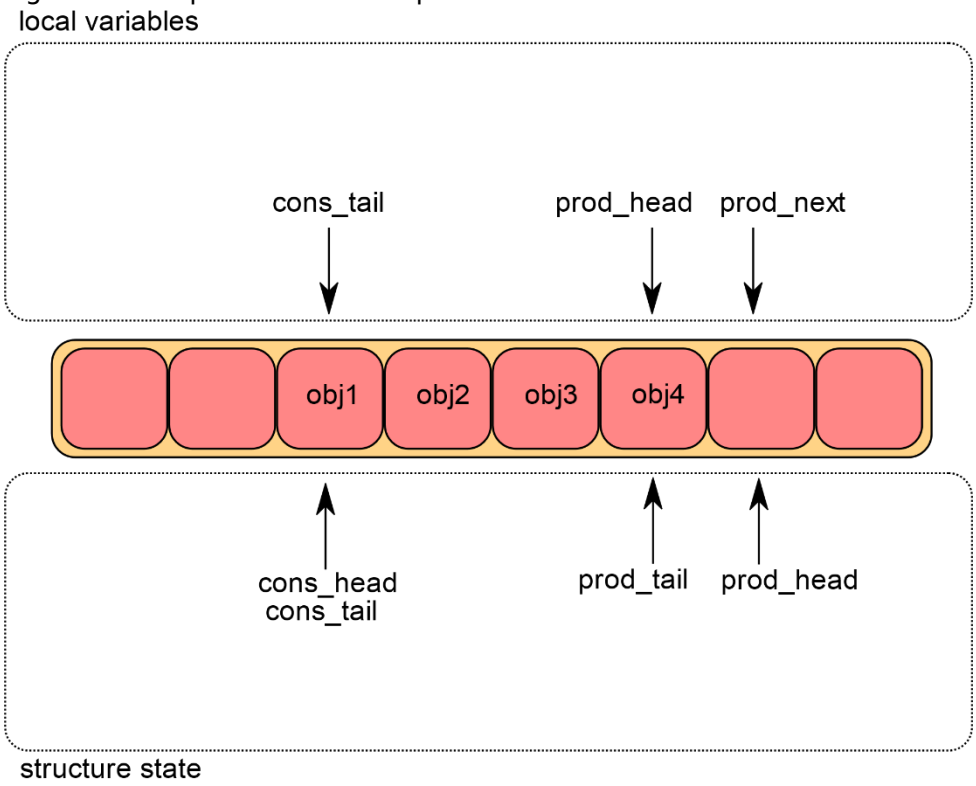
Figure 4-5 Enqueue first step



4.5.1.2. 入队第二步

第二步是在环结构中修改 ring->prod_head，以指向与 prod_next 相同的位置。指向待添加对象的指针被复制到 ring 中。

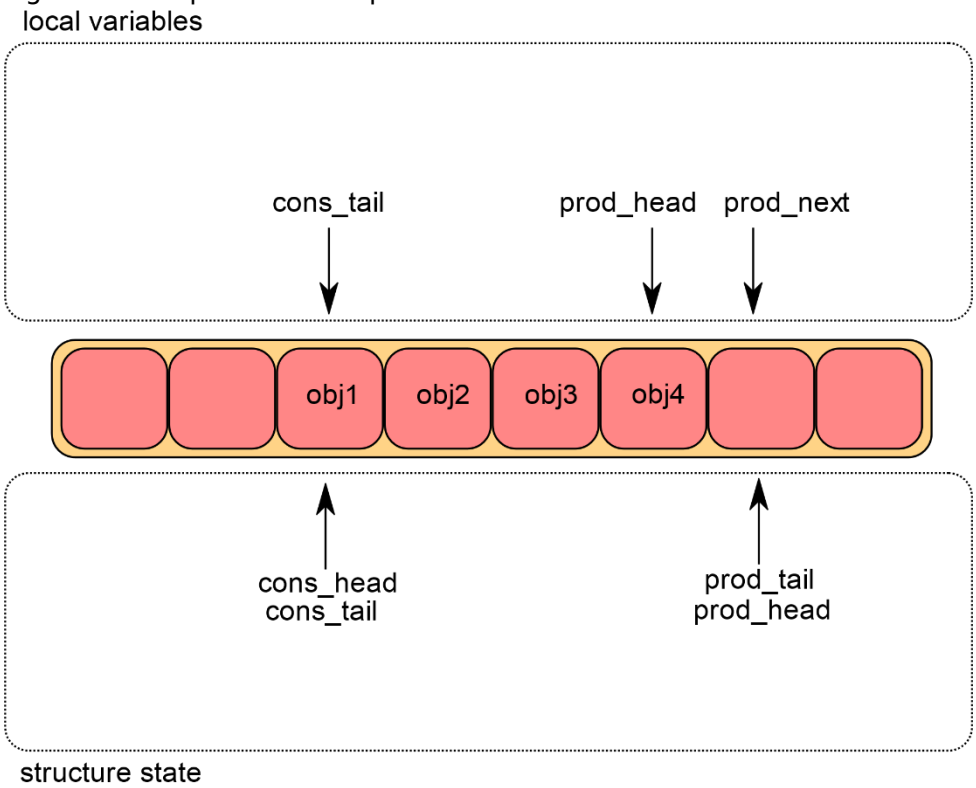
Figure 4-6 Enqueue second step



4.5.1.3. 入队最后一步

一旦将对象添加到 ring 中，ring 结构中的 ring->prod_tail 将被修改，指向与 ring->prod_head 相同的位置。入队操作完成。

Figure 4-7 Enqueue last step



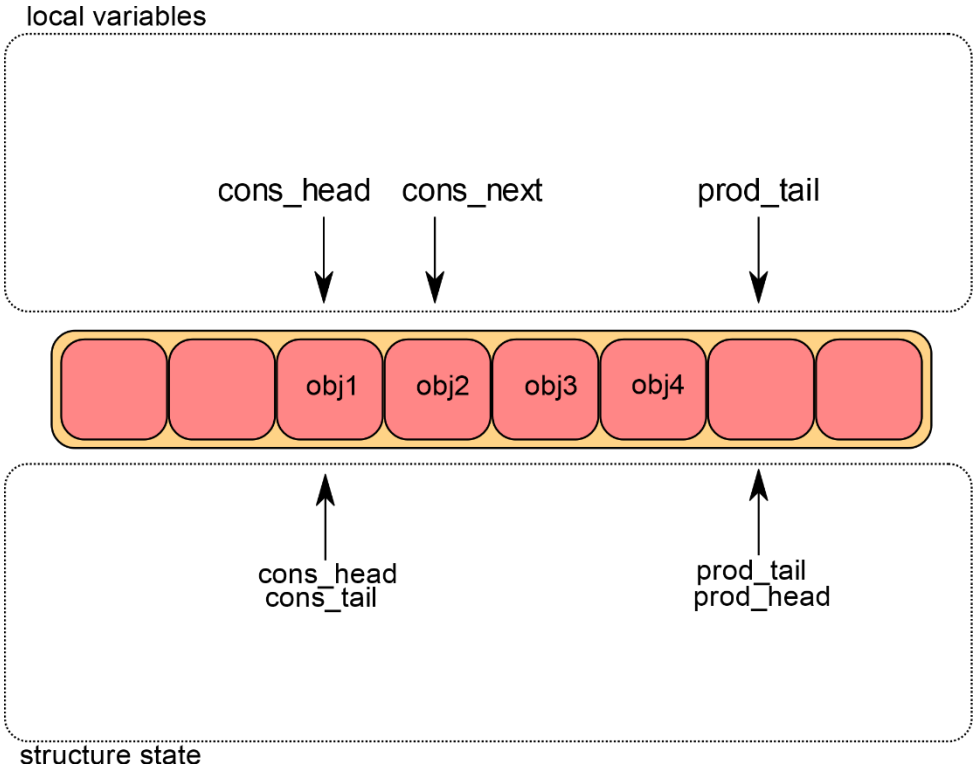
4.5.2. 单消费者出队

本节介绍一个消费者从 ring 中取出对象的情况。在本例中，只有消费者头尾指针(cons_head and cons_tail)被修改，只有一个消费者。初始状态是将 cons_head 和 cons_tail 指向相同位置。

4.5.2.1. 出队第一步

首先，将 ring->cons_head 和 ring->prod_tail*复制到局部变量中。 *cons_next 本地变量指向表的下一个元素，或者在批量出队的情况下指向下几个元素。如果 ring 中没有足够的对象用于出队(通过检查 prod_tail)，将返回错误。

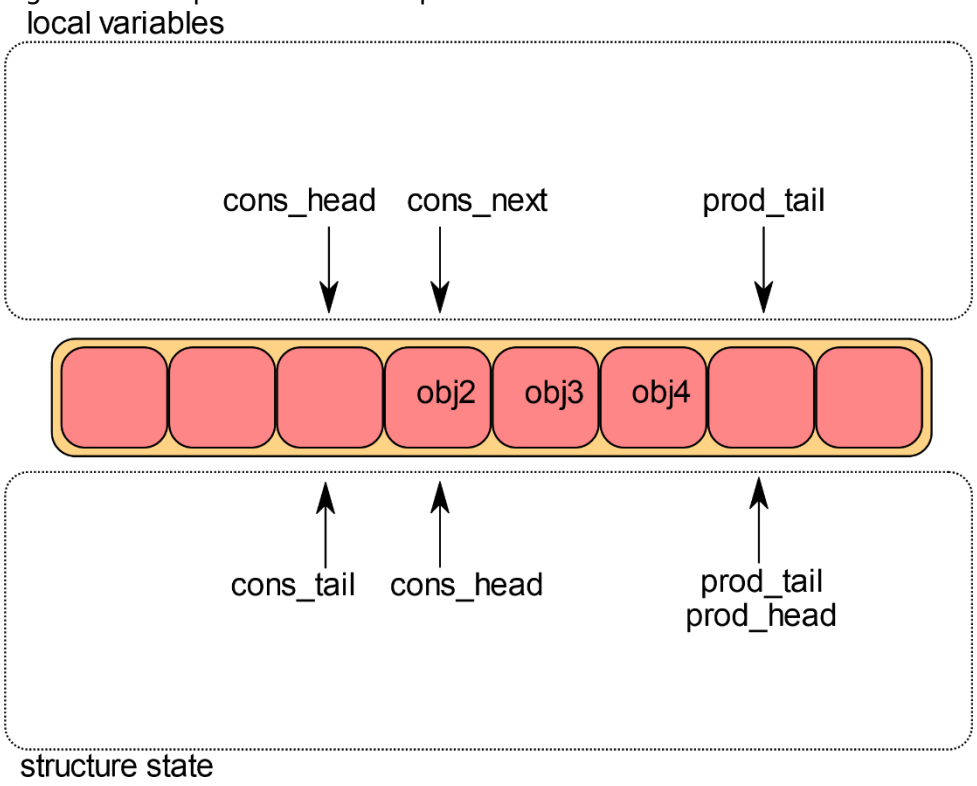
Figure 4-8 Dequeue first step



4.5.2.2. 出队第二步

第二步是修改 ring 结构中 ring->cons_head，以指向 cons_next 相同的位置。指向出队对象(obj1) 的指针被复制到用户指定的指针中。

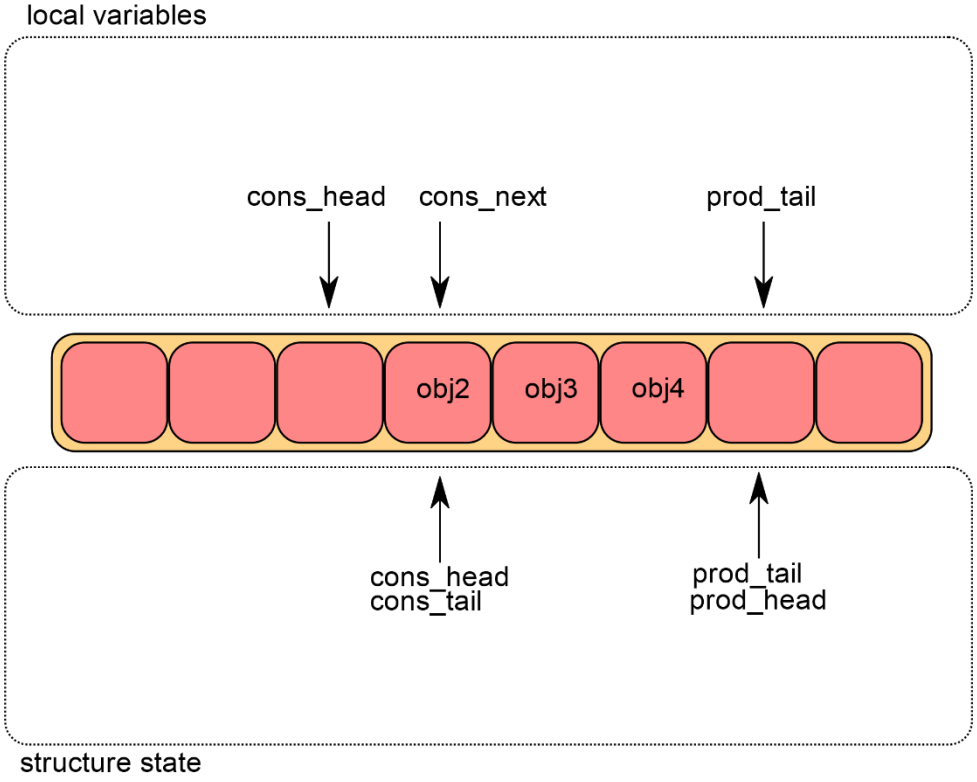
Figure 4-9 Dequeue second step



4.5.2.3. 出队最后一步

最后，ring 中的 ring->cons_tail 被修改为指向 ring->cons_head 相同的位置。 出队操作完成。

Figure 4-10 Dequeue last step



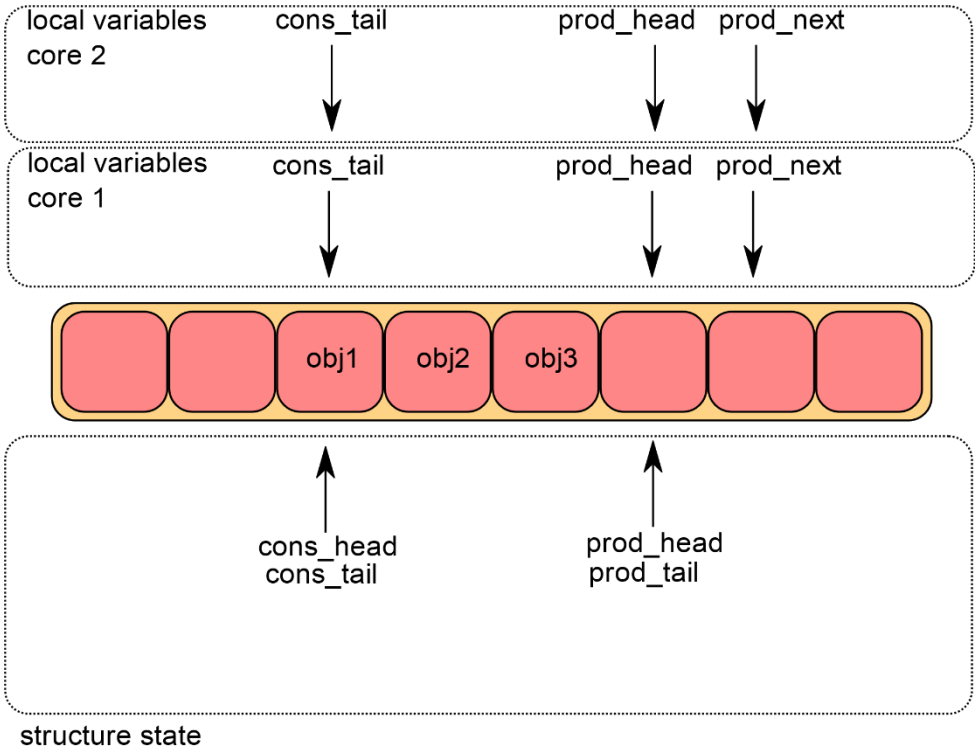
4.5.3. 多生产者入队

本节说明两个生产者同时向 ring 中添加对象的情况。在本例中，仅修改生产者头尾指针(prod_head 和 prod_tail)。初始状态是将 prod_head 和 prod_tail 指向相同的位置。

4.5.3.1. 多生产者入队第一步

在生产者的两个 core 上， ring->prod_head 及 ring->cons_tail 都被复制到局部变量。 局部变量 prod_next 指向下一个元素，或者在批量入队的情况下指向下几个元素。
如果 ring 中没有足够的空间用于入队(通过检查 cons_tail)，将返回错误。

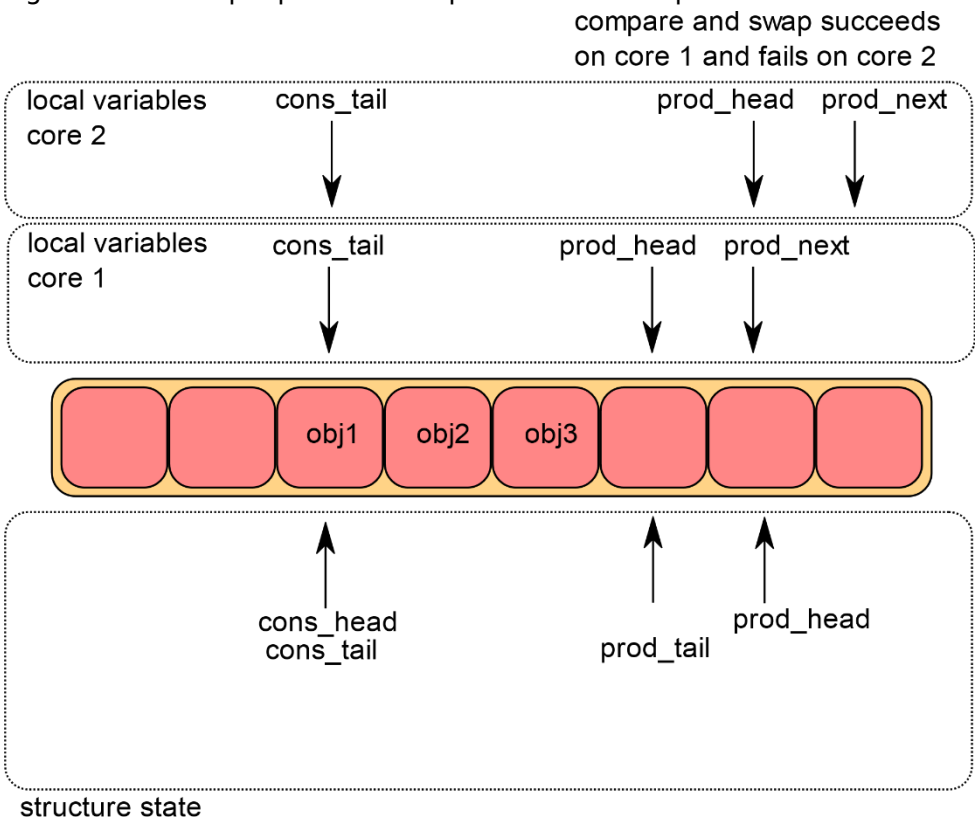
Figure 4-11 Multiple producer enqueue first step



4.5.3.2. 多生产者入队第二步

第二步是修改 ring 结构中 ring->prod_head，来指向 prod_next 相同的位置。 此操作使用比较和交换(CAS)指令，该指令以原子操作的方式执行以下操作：
如果 ring->prod_head 与本地变量 prod_head 不同，则 CAS 操作失败，代码将在第一步重新启动。
否则，ring->prod_head 设置为本地变量 prod_next，CAS 操作成功并继续下一步处理。
在图中，core1 执行成功，core2 重新启动步骤 1。

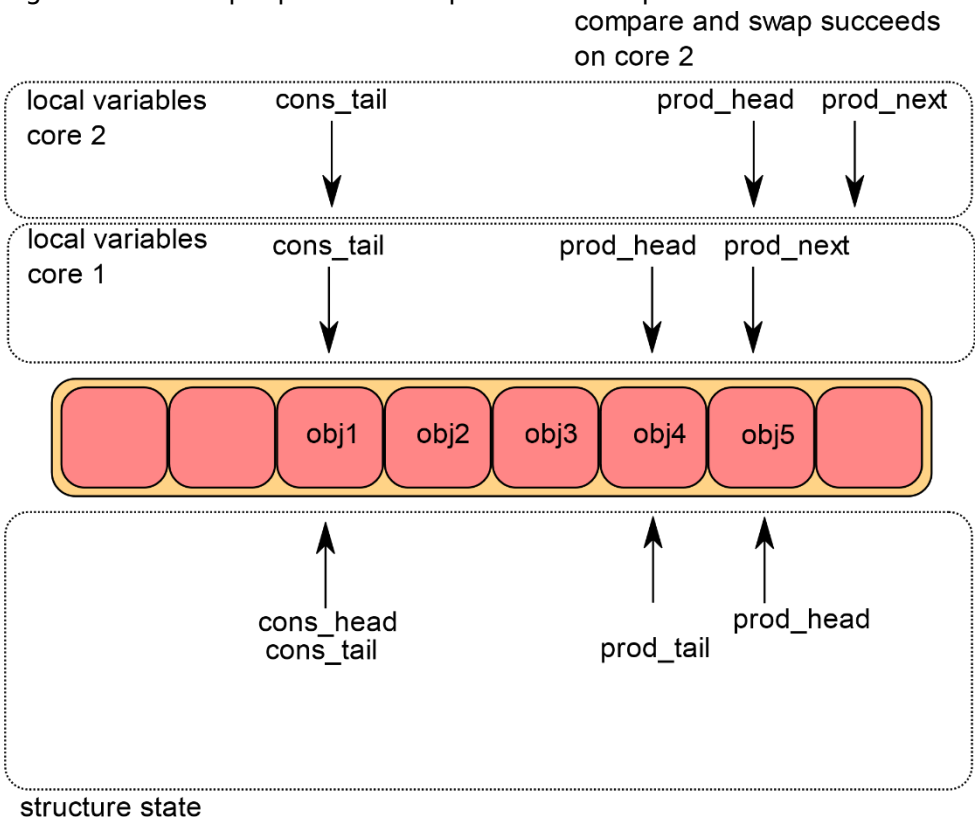
Figure 4-12 Multiple producer enqueue second step



4.5.3.3. 多生产者入队第三步

core 2 的 CAS 操作成功重试。
core 1 更新一个对象(obj4)到 ring 上。Core 2 更新一个对象(obj5)到 ring 上

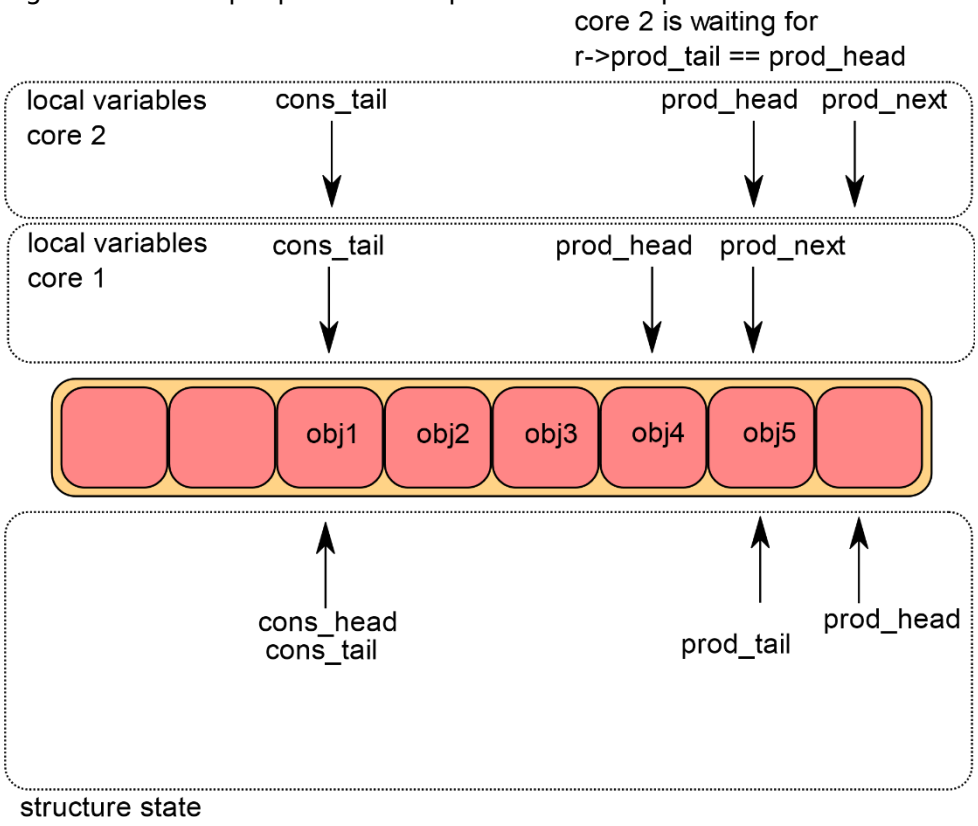
Figure 4-13 Multiple producer enqueue third step



4.5.3.4. 多生产者入队第四步

每个 core 现在都想更新 ring->prod_tail。 只有 ring->prod_tail 等于 prod_head 本地变量，core 才能更新它。 当前只有 core 1 满足，操作在 core 1 上完成。

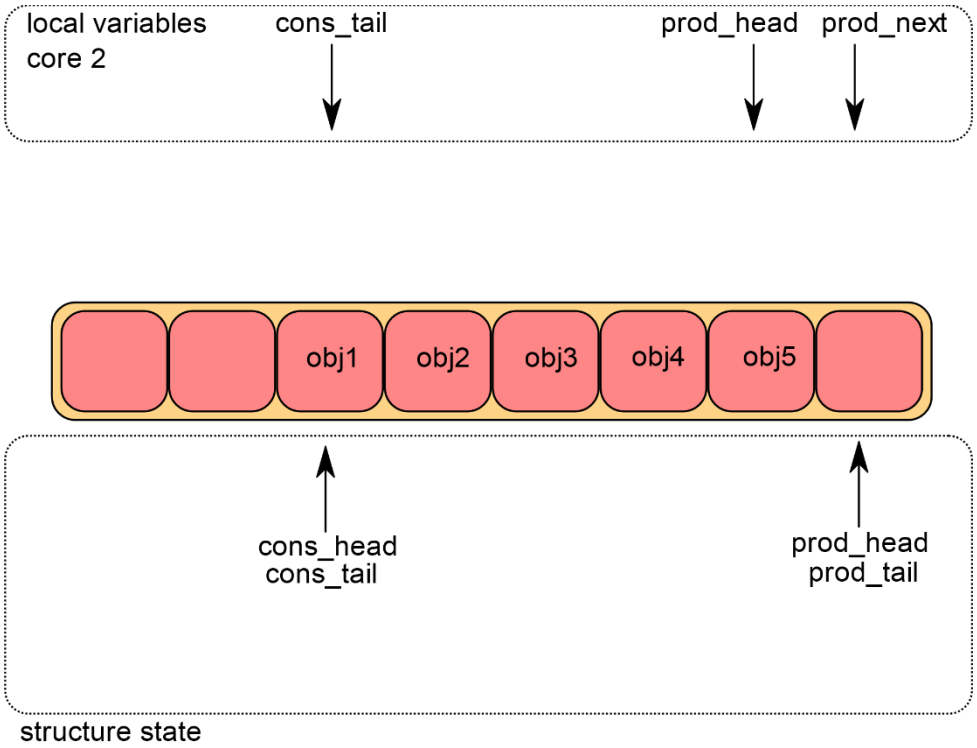
Figure 4-14 Multiple producer enqueue fourth step



4.5.3.5. 多生产者入队最后一步

一旦 ring->prod_tail 被 core 1 更新完，core 2 也满足条件，允许更新。core 2 上也完成了操作。

Figure 4-15 Multiple producer enqueue last step



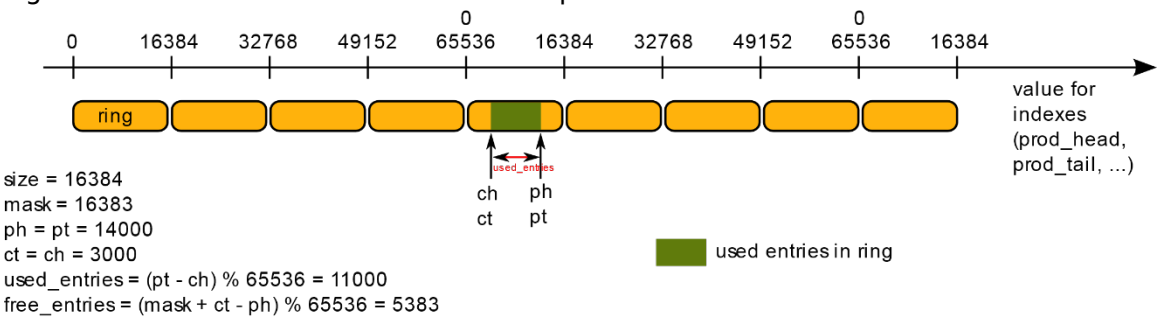
4.5.4. 32-bit 模索引值

在前面的图中，prod_head，prod_tail，cons_head 和 cons_tail 索引由箭头表示。但是，在实际实现中，这些值不会假定在 0～（size(ring)-1）之间。 索引值在 0～（2^32-1）之间，当我们访问 ring 本身时，我们掩码取值。32bit 模数也意味着如果溢出 32bit 的范围，对索引的操作将自动执行 2^32 模。

以下是两个例子，用于帮助解释索引值如何在 ring 中使用。

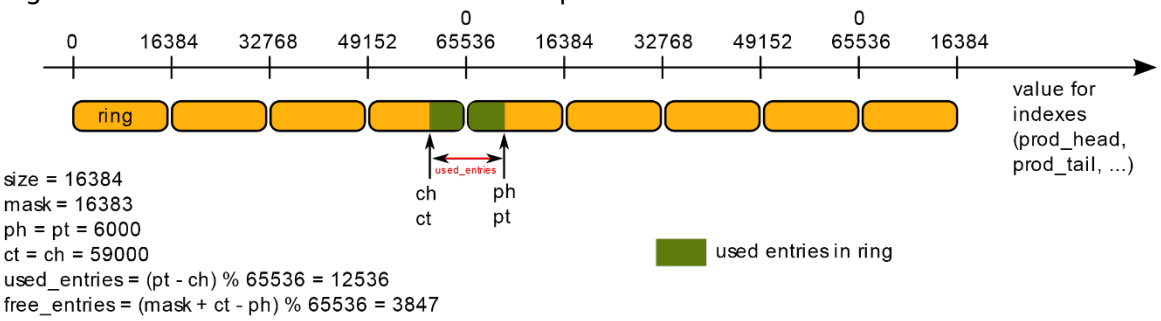
> 注意：
> 为了简化说明，使用模 16bit 操作，而不是 32bit。另外，四个索引被定义为 16bit 无符号整数，与实际情况下的 32bit 无符号数相反。

Figure 4-16 Modulo 32-bit indexes - Example 1



这个 ring 包含 11000 对象。

Figure 4-17 Modulo 32-bit indexes - Example 2



这个 ring 包含 12536 个对象。

> 注意：
> 为了便于理解，我们在上面的例子中使用模 65536 操作。 在实际执行情况下，这种低效操作是多余的，但是，当溢出时会自动执行。

代码始终保证生产者和消费者之间的距离在 0～（size(ring)-1）之间。基于这个属性，我们可以对两个索引值做减法，而不用考虑溢出问题。
任何情况下，ring 中的对象和空闲对象都在 0～（size(ring)-1）之间，即便第一个减法操作已经溢出：

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

4.6. 参考

- bufring.h in FreeBSD (version 8)
- bufring.c in FreeBSD (version 8)
- Linux Lockless Ring Buffer Design

5. 内存池库

内存池是固定大小的对象分配器。在 DPDK 中，它由名称唯一标识，并且使用 mempool 操作来存储空闲对象。默认的 mempool 操作是基于 ring 的。它提供了一些可选的服务，如 per-core 缓存和对齐帮助，以确保对象被填充，方便将他们均匀扩展到 DRAM 或 DDR3 通道上。

这个库由报文缓冲区库使用。

5.1. Cookies

在调试模式（CONFIG_RTE_LIBRTE_MEMPOOL_DEBUG 使能）中，将在内存块的开头和结尾处添加 cookies。 分配的对象包含保护字段，以帮助调试缓冲区溢出。

5.2. Stats

在调试模式（CONFIG_RTE_LIBRTE_MEMPOOL_DEBUG 使能）中，从池中获取/释放的统计信息存放在 mempool 结构体中。统计信息是 per-lcore 的，避免并发访问统计计数器。

5.3. 内存对齐限制

根据硬件内存配置，可以通过在对象之间添加特定的填充来大大提高性能。其目的是确保每个对象开始于不同的内存通道上，并在内存中排列，以便实现所有通道负载均衡。

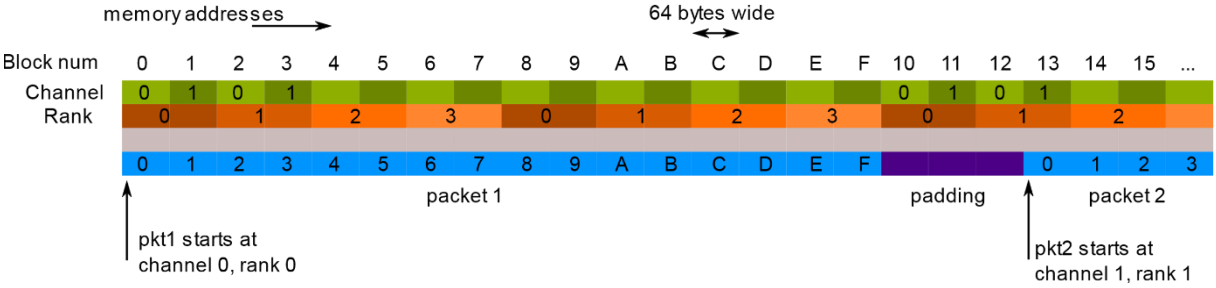
特别是当进行 L3 转发或流分类时，报文缓冲对齐尤为重要。此时访问报文的前 64B，因此可以通过在将对象的起始地址分布到不同的信道上来提升性能。

DIMM 上的 rank 数目是可访问 DIMM 完整数据位宽的独立 DIMM 集合的数量。 由于他们共享相同的路径，因此 rank 不能被同时访问。 DIMM 上的 DRAM 芯片的物理布局无需与 rank 数目相关。

当运行应用程序时，EAL 命令行选项提供了添加内存通道和 rank 数目的能力。
命令行必须始终指定处理器的内存通道数目。

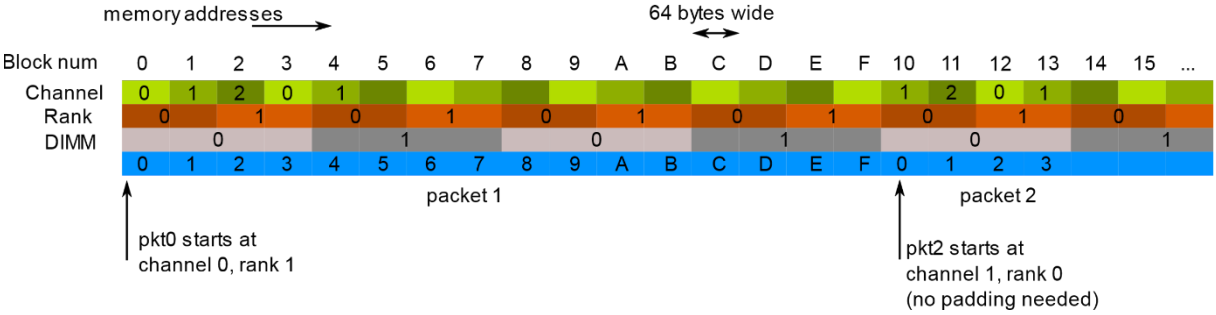
不同 DIMM 架构的对齐实例如下图所示。

Figure 5-18 Two Channels and Quad-ranked DIMM Example



在这个例子中，假设报文时 16 块 64B 存储就不成立了。
Intel® 5520 芯片组有三个通道，因此，在大多数情况下，对象之间不需要填充。（除了大小为 n x 3 x 64B 的块）。

Figure 5-19 hree Channels and Two Dual-ranked DIMM Example



当创建一个新内存池时，用户可以指定使用此功能。

5.4. 本地缓存

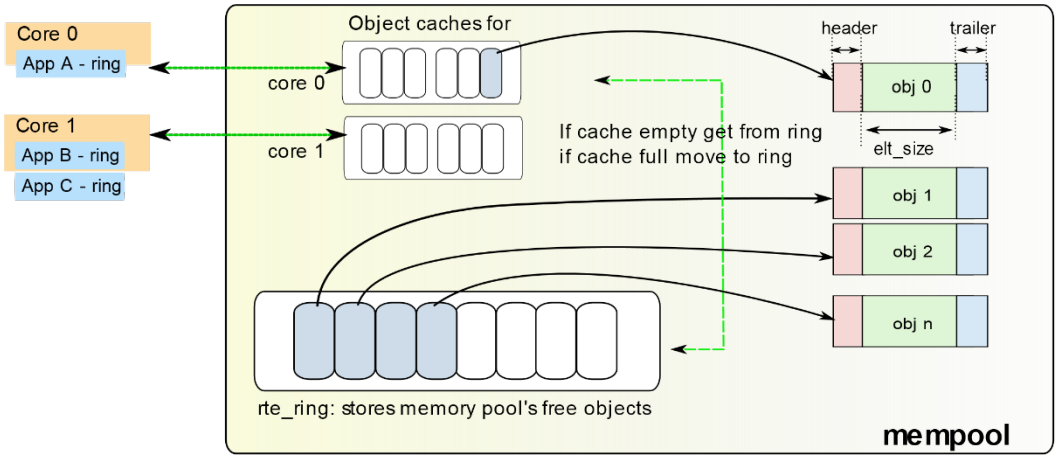
考虑 CPU 的使用率，由于每个访问需要 compare-and-set (CAS)操作，所以多核访问内存池环形缓冲器中的空闲节点成本比较高。为了避免对内存池 ring 的访问请求太多，内存池分配器可以维护 per-core cache，并通过实际内存池中具有较少锁定的缓存对内存池 ring 执行批量请求。通过这种方式，每个 core 都可以访问自己空闲对象的缓存（带锁），只有当空闲缓存被填充时，内核才需要将这些对象重新放回到缓冲池 ring，或者当本地缓存空时，从缓冲池中获取更多对象。

虽然这意味着一些 buffer 可能在某些 core 的本地缓存上处于空闲状态，但是 core 可以无锁访问其自己的缓存提供了性能上的提升。

缓存由一个小型的 per-core 表及其长度组成。可以在创建池时启用/禁用此缓存。

缓存大小的最大值是静态配置，并在编译时定义的(CONFIG_RTE_MEMPOOL_CACHE_MAX_SIZE)。

Figure 5-20 A mempool in Memory with its Associated Ring



不同于 per-lcore 缓存，应用程序可以通过接口 rte_mempool_cache_create()，rte_mempool_cache_free()和 rte_mempool_cache_flush() 创建和管理外部缓存。这些用户拥有的缓存可以被显式传递给 rte_mempool_generic_put() 和 rte_mempool_generic_get() 。接口 rte_mempool_default_cache() 返回默认内部缓存。与默认缓存相反，用户拥有的高速缓存可以由非 EAL 线程使用。

5.5. 内存池操作

这允许外部存储子系统，如外部硬件存储管理系统和基于软件的存储管理与 DPDK 一起使用。

mempool 处理包括两方面：

- 添加你自己新的 mempool 操作代码。这是通过添加 mempool ops 代码，并使用 MEMPOOL_REGISTER_OPS 宏来实现的。
- 使用新的 API 调用 rte_mempool_create_empty() 及 rte_mempool_set_ops_byname() 用于创建新的 mempool，并制定用户要使用的操作。

在同一个应用程序中可能会使用几个不同的 mempool 处理。可以使用 rte_mempool_create_empty() 创建一个新的 mempool，然后用 rte_mempool_set_ops_byname() 将 mempool 指向相关的 mempool 处理回调（ops）结构体。

传统的应用程序可能会继续使用旧的 rte_mempool_create() API 调用，它默认使用基于 ring 的 mempool 处理。这些应用程序需要修改为新的 mempool 处理。

对于使用 rte_pktmbuf_create() 的应用程序，有一个配置设置（RTE_MBUF_DEFAULT_MEMPOOL_OPS），允许应用程序使用另一个 mempool 处理。

5.6. 用例

需要高性能的所有分配器应该使用内存池实现。以下是一些使用实例：

- 报文缓冲区库

- 环境抽象层
- 任何需要在程序中分配固定大小对象，并将被系统持续使用的应用程序

6. 报文缓冲区库

报文缓冲区库（Mbuf）提供了申请和释放缓冲区的功能，DPDK 应用程序使用这些 buffer 存储消息缓冲。消息缓冲存储在 mempool 中，使用内存池库 。

数据结构 rte_mbuf 可以承载网络数据包 buffer 或者通用控制消息 buffer(由 CTRL_MBUF_FLAG 指示)。也可以扩展到其他类型。rte_mbuf 头部结构尽可能小，目前只使用两个缓存行，最常用的字段位于第一个缓存行中。

6.1. 报文缓冲区设计

为了存储数据包数据（包括协议头部），考虑了两种方法：

1. 在单个存储 buffer 中嵌入 metadata，后面跟着数据包数据固定大小区域
2. 为 metadata 和报文数据分别使用独立的存储 buffer。

第一种方法的优点是他只需要一个操作来分配/释放数据包的整个存储表示。但是，第二种方法更加灵活，并允许将元数据的分配与报文数据缓冲区的分配完全分离。

DPDK 选择了第一种方法。Metadata 包含诸如消息类型，长度，到数据开头的偏移量等控制信息，以及允许缓冲链接的附加 mbuf 结构指针。

用于承载网络数据包 buffer 的消息缓冲可以处理需要多个缓冲区来保存完整数据包的情况。许多通过下一个字段链接在一起的 mbuf 组成的 jumbo 帧，就是这种情况。

对于新分配的 mbuf，数据开始的区域是 buffer 之后 RTE_PKTMBUF_HEADROOM 字节的位置，这是缓存对齐的。 Message buffers 可以在系统中的不同实体中携带控制信息，报文，事件等。Message buffers 也可以使用起 buffer 指针来指向其他消息缓冲的数据字段或其他数据结构。

Figure 6-21 An mbuf with One Segment

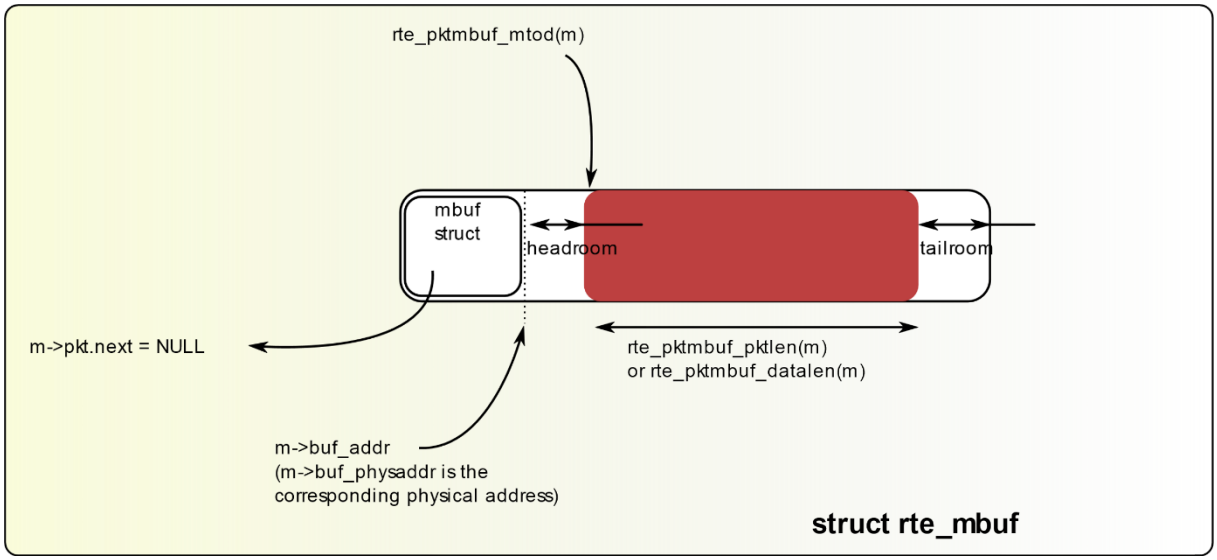
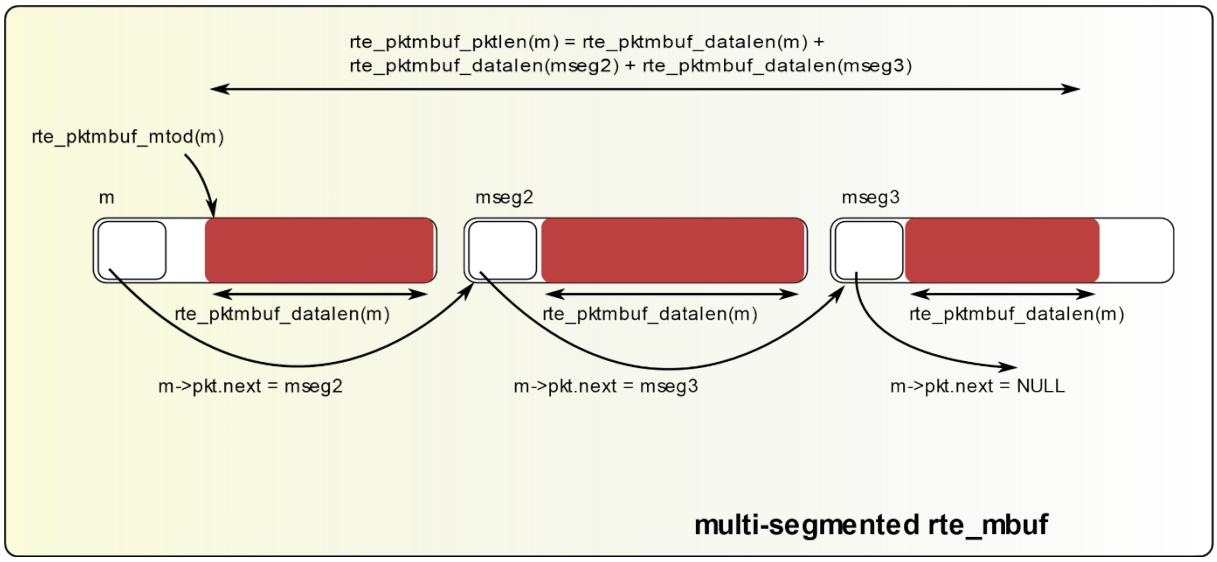


Figure 6-22 An mbuf with Three Segments



Buffer Manager 实现了一组相当标准的 buffer 访问操作来操纵网络数据包。

6.2. 存储在内存池中的缓冲区

Buffer Manager 使用内存池库来申请 buffer。因此确保了数据包头部均衡分布到信道上，有利于 L3 处理。mbuf 中包含一个字段，用于表示它从哪个池中申请出来。当调用 rte_ctrlmbuf_free(m) 或 rte_pktmbuf_free(m)，mbuf 被释放到原来的池中。

6.3. 构造函数

Packet 及 control mbuf 构造函数由 API 提供。接口 rte_pktmbuf_init()及 rte_ctrlmbuf_init()初始化 mbuf 结构中的某些字段，这些字段一旦创建将不会被用户修改（如 mbuf 类型、源池、缓冲区起始地址等）。此函数在池创建时作为 rte_mempool_create()函数的回掉函数给出。

6.4. 缓冲区申请及释放

分配一个新 mbuf 需要用户指定从哪个池中申请。对于任意新分配的 mbuf，它包含一个段，长度为 0。缓冲区到数据的偏移量被初始化，以便使得 buffer 具有一些字节（RTE_PKTMBUF_HEADROOM）的 headroom。

释放 mbuf 意味着将其返回到原始的 mempool。当 mbuf 的内容存储在一个池中（作为一个空闲的 mbuf）时，mbuf 的内容不会被修改。由构造函数初始化的字段不需要在 mbuf 分配时重新初始化。

当释放包含多个段的数据包 mbuf 时，他们都被释放，并返回到原始 mempool。

6.5. 缓冲区操作

这个库提供了一些操作数据包 mbuf 中的数据的功能。例如：

- 获取数据长度
- 获取指向数据开始位置的指针
- 数据前插入数据
- 数据之后添加数据
- 删除缓冲区开头的数据（rte_pktmbuf_adj()）
- 删除缓冲区末尾的数据（rte_pktmbuf_trim()）详细信息请参阅 DPDK API Reference

6.6. 元数据信息

数据包的一些信息由网络驱动程序检索并存储在 mbuf 中使得处理更简单。例如，VLAN、RSS 哈希结果（参见 Poll Mode Driver）及校验和由硬件计算的标志等。

一个报文缓冲区中还包含数据源端口和报文链中 mbuf 数目。对于链接的 mbuf，只有链的第一个 mbuf 存储这个元信息。

例如，对于 IEEE1588 数据包，RX 侧就是这种情况，时间戳机制，VLAN 标记和 IP 校验和计算。

在 TX 端，应用程序还可以将一些处理委托给硬件。 例如，PKT_TX_IP_CKSUM 标志允许卸载 IPv4 校验和的计算。

以下示例说明如何在 vxlan 封装的 tcp 数据包上配置不同的 TX offloads：out_eth/out_ip/out_udp/vxlan/in_eth/in_ip/in_tcp/payload

- 计算 out_ip 的校验和:

```
mb->l2_len = len(out_eth)
mb->l3_len = len(out_ip)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CSUM
set out_ip checksum to 0 in the packet
```

配置 DEV_TX_OFFLOAD_IPV4_CKSUM 支持在硬件计算。

- 计算 out_ip 和 out_udp 的校验和:

```
mb->l2_len = len(out_eth)
mb->l3_len = len(out_ip)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CSUM | PKT_TX_UDP_CKSUM
set out_ip checksum to 0 in the packet
set out_udp checksum to pseudo header using rte_ipv4_phdr_cksum()
```

配置 DEV_TX_OFFLOAD_IPV4_CKSUM 和 DEV_TX_OFFLOAD_UDP_CKSUM 支持在硬件上计算。

- 计算 in_ip 的校验和:

```
mb->l2_len = len(out_eth + out_ip + out_udp + vxlan + in_eth)
mb->l3_len = len(in_ip)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CSUM
set in_ip checksum to 0 in the packet
```

这以情况 1 类似，但是 l2_len 不同。 配置 DEV_TX_OFFLOAD_IPV4_CKSUM 支持硬件计算。注意，只有外部 L4 校验和为 0 时才可以工作。

- 计算 in_ip 和 in_tcp 的校验和:

```
mb->l2_len = len(out_eth + out_ip + out_udp + vxlan + in_eth)
mb->l3_len = len(in_ip)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CSUM | PKT_TX_TCP_CKSUM
在报文中设置 in_ip 校验和为 0
使用 rte_ipv4_phdr_cksum()将 in_tcp 校验和设置为伪头
```

这与情况 2 类似，但是 l2_len 不同。 配置 DEV_TX_OFFLOAD_IPV4_CKSUM 和 DEV_TX_OFFLOAD_TCP_CKSUM 支持硬件实现。注意，只有外部 L4 校验和为 0 才能工作。

- segment inner TCP:

```
mb->l2_len = len(out_eth + out_ip + out_udp + vxlan + in_eth)
mb->l3_len = len(in_ip)
mb->l4_len = len(in_tcp)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CKSUM | PKT_TX_TCP_CKSUM | PKT_TX_TCP_SEG;
在报文中设置 in_ip 校验和为 0
```

将 in_tcp 校验和设置为伪头部，而不使用 IP 载荷长度

配置 DEV_TX_OFFLOAD_TCP_TSO 支持硬件实现。 注意，只有 L4 校验和为 0 时才能工作。

- 计算 out_ip, in_ip, in_tcp 的校验和:

```
mb->outer_l2_len = len(out_eth)
mb->outer_l3_len = len(out_ip)
mb->l2_len = len(out_udp + vxlan + in_eth)
mb->l3_len = len(in_ip)
mb->ol_flags|=PKT_TX_OUTER_IPV4|PKT_TX_OUTER_IP_CKSUM | PKT_TX_IP_CKSUM |
PKT_TX_TCP_CKSUM;
设置 out_ip 校验和为 0
设置 in_ip 校验和为 0
使用 rte_ipv4_phdr_cksum()设置 in_tcp 校验和为伪头部
```

配 置 DEV_TX_OFFLOAD_IPV4_CKSUM 、 DEV_TX_OFFLOAD_UDP_CKSUM 、 DEV_TX_OFFLOAD_OUTER_IPV4_CKSUM 支持硬件实现。

Flage 标记的意义在 mbuf API 文档(rte_mbuf.h)中有详细描述。 更多详细信息还可以参阅 testpmd 源码(特别是 csumonly.c)。

6.7. 直接缓冲区和间接缓冲区

直接缓冲区是指缓冲区完全独立。间接缓冲区的行为类似于直接缓冲区，但缓冲区的指针和数据偏移量指的是另一个直接缓冲区的数据。这在数据包需要复制或分段的情况下是很有用的，因为间接缓冲区提供跨越多个缓冲区重用相同数据包数据的手段。

当使用接口 rte_pktmbuf_attach() 函数将缓冲区附加到直接缓冲区时，该缓冲区变成间接缓冲区。每个缓冲区有一个引用计数器字段，每当直接缓冲区附加一个间接缓冲区时，直接缓冲区上的引用计数器递增。类似的，每当间接缓冲区被分裂时，直接缓冲区上的引用计数器递减。如果生成的引用计数器为 0，则直接缓冲区将被释放，因为它不再使用。

处理间接缓冲区时需要注意几件事情。首先，间接缓冲区从不附加到另一个间接缓冲区。尝试将缓冲区 A 附加到间接缓冲区 B（且 B 附加到 C 上了），将使得 rte_pktmbuf_attach() 自动将 A 附加到 C 上。其次，为了使缓冲区变成间接缓冲区，其引用计数必须等于 1，也就是说它不能被另一个间接缓冲区引用。最后，不可能将间接缓冲区重新链接到直接缓冲区（除非它已经被分离了）。

虽然可以使用推荐的 rte_pktmbuf_attach()和 rte_pktmbuf_detach()函数直接调用附加/分离操作，但建议使用更高级的 rte_pktmbuf_clone()函数，该函数负责间接缓冲区的正确初始化，并可以克隆具有多个段的缓冲区。

由于间接缓冲区不应该实际保存任何数据，间接缓冲区的内存池应配置为指示减少的内存消耗。可以在几个示例应用程序中找到用于间接缓冲区的内存池（以及间接缓冲区的用例示例）的初始化示例，例如 IPv4 组播示例应用程序。

6.8. 调试

在调试模式（CONFIG_RTE_MBUF_DEBUG 使能）下，mbuf 库的函数在任何操作之前执行完整性检查(如缓冲区检查、类型错误等)。

6.9. 用例

所有网络应用程序都应该使用 mbufs 来传输网络数据包。

7. 轮询模式驱动

DPDK 包括 1Gigabit、10Gigabit 及 40Gigabit 和半虚拟化 IO 的轮询模式驱动程序。

轮询模式驱动程序(PMD)由通过在用户空间中运行的 BSD 驱动提供的 API 组成，以配置设备及它们的队列。此外，PMD 直接访问 RX 和 TX 描述符，且不会有任何中断（链路状态更改中断除外）产生，这可以保证在用户空间应用程序中快速接收，处理和传送数据包。本节介绍 PMD 的要求、设计原则和高级架构，并介绍了以太网 PMD 的对外通用 API。

7.1. 前提与假设

DPDK 环境支持两种模式的数据包处理，RTC 模式和 pipeline 模式：

- 在 run-to-completion 模式中，通过调用 API 来轮询指定端口的 RX 描述符以获取报文。 紧接着，在同一个 core 上处理报文，并通过 API 调用将报文放到接口的 TX 描述符中以发送报文。
- 在 pipe-line 模式中，一个 core 轮询一个或多个接口的 RX 描述符以获取报文。然后报文经由 ring 被其他 core 处理。其他 core 可以继续处理报文，最终报文被放到 TX 描述符中以发送出去。

在同步的 run-to-complete 模式中，每个逻辑核处理数据包的流程包括以下步骤：

- 通过 PMD 报文接收 API 来获取报文
- 一次性处理每个数据报文，直到转发阶段
- 通过 PMD 发包 API 将报文发送出去

相反地，在异步的 pipeline 模式中，一些逻辑核可能专门用于接收报文，其他逻辑核用于处理前面收到的报文。收到的数据包通过报文 ring 在逻辑核之间交换。数据包收包过程包括以下步骤：

- 通过 PMD 收包 API 获取报文
- 通过数据包队列向逻辑核提供接收到的数据包

数据包处理过程包括以下步骤：

- 从数据包队列中获取数据包
- 处理接收到的数据包，直到重新发送出去

为了避免任何不必要的中断处理开销，执行环境不得使用任何异步通知机制。即便有需要，也应该尽量使用 ring 来引入通知信息。

在多核环境中避免锁竞争是一个关键问题。为了解决这个问题，PMD 旨在尽可能地使用每个 core 的私有资源。例如，PMD 每个端口维护每个 core 单独的传输队列。同样的，端口的每个接收队列都被分配给单个逻辑核并由其轮询。

为了兼容 NUMA 架构，内存管理旨在为每个逻辑核分配本地（相同插槽）中的专用缓冲池，以最大限度地减少远程内存访问。数据包缓冲池的配置应该考虑到 DIMMs、channels 和 ranks 等底层物理内存架构。应用程序必须确保在内存池创建时给出合适的参数。具体内容参阅内存池库。

7.2. 设计原则

Ethernet* PMDs 的 API 和架构设计遵考虑到以下原则。

PMDs 必须能够帮助上层的应用实现全局的策略。反之，不能阻止或妨碍上层应用的实施。

例如，PMD 的发送和接收函数都有大量的报文或描述符需要轮询。这允许 RTC 处理协议栈通过不同的全局循环策略静态修护或动态调整其行为如：

- 立即接收，处理并以零碎的方式一次传送数据包。
- 尽可能所的接收数据包，然后处理所有数据包，再发送。
- 接收给定的最大量的数据包，处理接收的数据包，累加，最后将累加的数据包发送出去。

为了实现最优性能，需要考考整体软件设计选择和纯软件优化技术，并与可用的低层次硬件优化功能（如 CPU 缓存属性、总线速度、NIC PCI 带宽等）进行考虑和平衡。报文传输的情况就是突发性网络报文处理是软硬件权衡问题的一个例子。 在初始情况下，PMD 只能导出一个 rte_eth_tx_one 函数，以便在给定的队列上一次传输一个数据包。最重要的是，可以轻松构建一个 rte_eth_tx_burst 函数，循环调用 rte_eth_tx_one 函数以便一次传输多个数据包。然而，PMD 有效地实现了 rte_eth_tx_burst 函数，以通过以下优化来最小化每个数据包的驱动级传输开销：

- 在多个数据包之间共享调用 rte_eth_tx_one 函数的非摊销成本。
- 启用 rte_eth_tx_burst 函数以利用 burst-oriented 硬件特性（缓存数据预取、使用 NIC 头/尾寄存器）以最小化每个数据包的 CPU 周期数， 例如，通过避免对环形缓传输描述符的不必要的读取寄存器访问，或通过系统地使用精确匹配告诉缓存行边界大小的指针数组。
- 使用 burst-oriented 软件优化技术来移除失败的操作结果，如 ring 索引的回滚。

还通过 API 引入了 Burst-oriented 函数，这些函数在 PMD 服务中密集使用。这些函数特别适用于 NIC ring 的缓冲区分配器，他们提供一次分配/释放多个缓冲区的功能。例如，一个 mbuf_multiple_alloc 函数返回一个指向 rte_mbuf 缓冲区的指针数组，它可以在向 ring 添加多个描述符来加速 PMD 的接收轮询功能。

7.3. 逻辑核、内存和网络接口卡队列的关系

当处理器的逻辑核和接口利用其本地存储时，DPDK 提供 NUMA 支持，以提供更好的性能。因此，与本地 PCIE 接口相关的 mbuf 分配应从本地内存中创建的内存池中申请。如果可能，缓冲区应该保留在本地处理器上以获取最佳性能，并且应使用从本地内存中分配的 mempool 中申请的 mbuf 来填充 RX 和 TX 缓冲区描述符。

如果数据包或数据操作在本地内存中，而不是在远程处理器内存上，则 RTC 模型也会运行得更好。只要所有使用的逻辑核位于同一个处理器上，pipeline 模型也将获得更好的性能。

所个逻辑核不应共享接口的接收或发送队列，因为这将需要全局上锁保护，而导致性能下降。

7.4. 设备标识和配置

7.4.1. 设备标识

每个 NIC 端口（总线/桥、设备、功能）由其 PCI 标识符唯一指定。该 PCI 标识符在 DPDK 初始化时执行的 PCI 探测/枚举功能分配。根据 PCI 标识符，NIC 端口被分配了两个其他的表示：

- 一个端口索引，用于在 PMD API 导出的所有函数中指定 NIC 端口
- 端口名称，用于在控制消息中指定端口，主要用于管理和调试目的。为了便于使用，端口名称包括端口索引。

7.4.2. 设备配置

每个 NIC 端口的配置包括以下步骤：

- 分配 PCI 资源
- 将硬件复位为公知的默认状态
- 设置 PHY 和链路
- 初始化统计计数器

PMD API 还必须导出函数用于启动/终止端口的全部组播功能，并且可以在混杂模式下设置/取消设置端口。

某些硬件卸载功能必须通过特定的配置参数在端口初始化时单独配置。例如，接收侧缩放（RSS）和数据中心桥接（DCB）功能就是这种情况。

7.4.3. 即时配置

所有可以“即时”启动或停止的设备功能（即不停止设备），无需 PMD API 来导出函数实现这些功能。

所需要的是只是设备 PCI 寄存器的映射地址，以在驱动程序之外使用特殊的函数来配置实现这些功能。

为此，PMD API 导出一个函数提供可用于在驱动程序外部设置给定设备功能的设备相关联的所有信息。这些信息包括 PCI 供应商标识符，PCI 设备标识符，PCI 设备寄存器的映射地址以及驱动程序的名称。

这种方法的主要优点是可以自由地选择 API 来启动、配置、停止这些设备功能。

例如，testpmd 应用程序中的英特尔®82576 千兆以太网控制器和英特尔®82599 万兆以太网控制器

控制器的 IEEE1588 功能配置。

可以以相同的方式配置端口的 L3 / L4 5-Tuple 包过滤功能等其他功能。以太网流控（暂停帧）可以在单个端口上进行配置。有关详细信息，请参阅 testpmd 源代码。此外，只要数据包 mbuf 设置正确，就可以为单个数据包启用网卡的 L4（UDP / TCP / SCTP）校验和卸载。相关详细信息，请参阅硬件 offload 。

7.4.4. 发送队列配置

每个传输队列都独立配置了以下信息：

- 发送环上的描述符数目
- NUMA 架构中，用于标识从哪个 socket 的 DMA 存储区分配传输环的标识
- 传输队列的 Prefetch，Host 及 Write-Back 阈值寄存器的值
- 传输报文释放的最小阈值。当用于传输数据包的描述符数量超过此阈值时，应检查网络适配器以查看是否有回写描述符。在 TX 队列配置期间可以传递 0，以指示应使用默认值。tx_free_thresh 的默认值为 32。这使得 PMD 不会去检索完成的描述符，直到 NIC 已经为此队列处理了 32 个报文。
- RS 位最小阈值。在发送描述符中设置报告状态（RS）位之前要使用的最小发送描述符数。请注意，此参数仅适用于 Intel 10GbE 网络适配器。如果从最后一个 RS 位设置开始使用的描述符数量（直到用于发送数据包的第一个描述符）超过发送 RS 位阈值（tx_rs_thresh），则 RS 位被设置在用于发送数据包的最后一个描述符上。简而言之，此参数控制网络适配器将哪些传输描述符写回主机内存。在 TX 队列配置期间可以传递值为 0，以指示应使用默认值。tx_rs_thresh 的默认值为 32。这确保在网络适配器回写最近使用的描述符之前至少使用 32 个描述符。这样可以节省 TX 描述符回写所产生的上游 PCIe *带宽。重要的是注意，当 tx_rs_thresh 大于 1 时，应将 TX 写回阈值（TX wthresh）设置为 0。有关更多详细信息，请参阅英特尔®82599 万兆以太网控制器数据手册。

对于 tx_free_thresh 和 tx_rs_thresh，必须满足以下约束：

- tx_rs_thresh 必须大于 0。
- tx_rs_thresh 必须小于环的大小减去 2。
- tx_rs_thresh 必须小于或等于 tx_free_thresh。
- tx_free_thresh 必须大于 0。
- tx_free_thresh 必须小于环的大小减去 3。
- 为了获得最佳性能，当 tx_rs_thresh 大于 1 时，TX wthresh 应设置为 0。

TX 环中的一个描述符用作哨兵以避免硬件竞争条件，因此是最大阈值限制。当配置 DCB 操作时，在端口初始化时，发送队列数和接收队列数必须设置为 128。

7.4.5. 按要求释放 TX 缓冲区

许多驱动程序并没有在数据包传输后立即将 mbuf 释放回到 mempool 或本地缓存中。相反，他们将 mbuf 留在 Tx 环中，当需要在 Tx 环中插入，或者 tx_rs_thresh 已经超过时，执行批量释放。

应用程序请求驱动程序通过接口 rte_eth_tx_done_cleanup() 释放使用的 mbuf。该 API 请求驱动程序释放不再使用的 mbufs，而不管 tx_rs_thresh 值是否已超过。有两种情况会使得应用程序可能想要立即释放 mbuf：

- 当给定的数据包需要发送到多个目标接口（对于第 2 层洪泛或第 3 层多播）。 一种方法是复制数据包或者复制需要操作的数据包头部。 另一种方法是发送数据包，然后轮询 rte_eth_tx_done_cleanup() 接口直到报文引用递减。 接下来，这个报文就可以发送到下一个目的接口。 该应用程序仍然负责管理不同目标接口之间所需的任何数据包操作，但可以避免数据复制。 该 API 独立于数据包是传输还是丢弃，只是 mbuf 不再被接口使用。
- 一些应用程序被设计为进行多次运行，如数据包生成器。 为了运行的性能原因和一致性，应用程序可能希望在每个运行之间重新设置为初始状态，其中所有 mbufs 都返回到 mempool。 在这种情况下，它可以为其已使用的每个目标接口调 rte_eth_tx_done_cleanup() API 以请求它释放所有使用的 mbuf。

要确定驱动程序是否支持该 API，请检查 Network Interface Controller Drivers 文档中的 Free Tx mbuf on demand 功能。

7.4.6. 硬件 offload

根据 rte_eth_dev_info_get() 提供的驱动程序功能，PMD 可能支持硬件 offload 功能，如校验和 TCP 分段或 VLAN 插入。

这些 offload 功能的支持意味着将专用状态位和值字段添加到 rte_mbuf 数据结构中，以及由每个 PMD 导出的接收/发送功能的适当处理。标记列表及其精确含义在 mbuf API 文档及报文缓冲区库中元数据章节有描述。

7.5. 轮询模式驱动 API

7.5.1. 概述

默认情况下，PMD 提供的所有外部函数都是无锁函数，这些函数假定在同一目标设备上不会在不同的逻辑 core 上并行调用。例如，PMD 接收函数不能再两个逻辑核上并行调用，以轮询相同端口的相同 RX 队列。当然，这个函数可以由不同的 RX 队列上的不同逻辑核并行调用。上级应用程序应该保证强制执行这条规则。

如果需要，多个逻辑核到并行队列的并行访问可以通过专门的在线加锁来显式保护，这些加锁函数是建立在相应的无锁 API 之上的。

7.5.2. 通用分组表示

数据包由数据结构 rte_mbuf 表示，这是一个包含所有必要信息的通用元数据结构。这些信息包括与硬件特征相对应的字段和状态位，如 IP 头部和 VLAN 标签的校验和。

数据结构 rte_mbuf 包括以通用方式表示网络控制器提供的硬件功能对应的字段。对于输入数据包，rte_mbuf 的大部分字段都由 PMD 来填充，包括接收描述符中的信息。相反，对于输出数据包，rte_mbuf 的大部分字段由 PMD 发送函数用于初始化发送描述符。

数据结构 mbuf 的更全面的描述，请参阅 Mbuf Library 章节。

7.5.3. 以太网设备 API

以太网 PMD 驱动导出的以太网设备 API 请参阅 《DPDK API 参考手册》 描述。

7.5.4. 扩展的统计 API

扩展的统计 API 允许每个独立的 PMD 导出一组唯一的统计信息。每个统计信息狗有三个属性：

- Name：下文描述的用户可读的格式化字符串
- Id：仅表示统计信息的整数
- Value：一个无符号的 64bit 整数，标识统计结果

请注意，扩展统计信息标识符是驱动程序特定的，因此对于不同的端口可能不一样。API 由各种 rte_eth_xstats _*()函数组成，允许应用程序灵活地检索统计信息。

7.5.4.1. 用户可读名称命名方案

对于暴露给 API 的客户端的字符串，存在一个命名方案。这是为了允许刮取 API 用于感兴趣的统计信息。命名方案使用由单个下划线_分割的字符串。方案如下：

- direction
- detail 1
- detail 2
- detail n
- unit

常规统计示例字符串如下，符合上面的方案：

- rx_bytes
- rx_crc_errors
- tx_multicast_packets

该方案虽然简单，但可以灵活地显示和读取统计字符串中的信息。以下示例说明了命名方案 rx_packets 的使用。 在这个例子中，字符串被分成两个组件。第一个 rx 表示统计信息与 NIC 的接收端相

关联。 第二个 packets 表示测量单位是数据包。

一个更为复杂的例子是 tx_size_128_to_255_packets 。 在这个例子中， tx 表示传输， size 是第一个细节， 128 等表示更多的细节， packets 表示这是一个数据包计数器。

元数据中的一些方案补充如下：
- 如果第一部分不符合 rx 或 tx ，统计计数器与传送或接收不相关。
- 如果第二部分的第一个字母是 q 且这个 q 后跟一个数字，则这个统计数据是特定队列的一部分。

使用队列号的示例如： tx_q7_bytes 表示此统计信息适用于队列号 7，并表示该队列上传输的字节数。

7.5.4.2. API 设计

xstats API 使用 name，id 和 value 来允许执行查询特定的统计信息。执行查找意味着两件事情：
- 快速路径中的统计信息不执行名称字符串比较
- 允许仅请求感兴趣的统计信息

API 通过将统计信息的名称映射到唯一的 ID 来确保满足这些要求，该唯一 ID 用作快速路径中的查找 Key。API 允许应用程序请求一个 id 值数组，以便 PMD 只执行所需的计算。预期的用法是应用程序扫描每个统计信息的名称，并且如果对该统计信息感兴趣则缓存该 id。在快速路径上，该整数可用于检索 id 表示的统计信息的实际值。

7.5.4.3. API 函数

只输出了少量的函数，用于检索统计信息的数量以及这些统计信息的名称、ID 和数值。
- rte_eth_xstats_get_names_by_id()：返回统计信息的名称，当给定一个 NULL 参数时，函数返回可用的统计数目。
- rte_eth_xstats_get_id_by_name()：搜索与 xstat_name 匹配的统计 ID。如果找到，则设置 id 值。
- rte_eth_xstats_get_by_id()：根据配提供的 ids 数组，填充一系列 uint64_t 值。如果 ids 数组为 NULL，则返回所有可用的统计信息。

7.5.4.4. API 用例

考虑一下，应用程序需要查看丢弃的数据包数目。如果没有数据包被丢弃，由于性能原因，应用程序不会读取任何其他指标。如果数据包丢弃，应用程序将具有一组特定的统计信息。这一组统计信息允许应用程序决定下一步执行的步骤。以下代码片段展示了如何使用 xstats API 来实现此目标。

第一步就是获取所有统计信息的名称，并列出来。

```
struct rte_eth_xstat_name *xstats_names;
uint64_t *values;
int len, i;

/* Get number of stats */
```

```
len = rte_eth_xstats_get_names_by_id(port_id, NULL, NULL, 0);
if (len < 0) {
   printf("Cannot get xstats count\n");
   goto err;
}

xstats_names = malloc(sizeof(struct rte_eth_xstat_name) * len);
if (xstats_names == NULL) {
   printf("Cannot allocate memory for xstat names\n");
   goto err;
}

/* Retrieve xstats names, passing NULL for IDs to return all statistics */
if (len != rte_eth_xstats_get_names_by_id(port_id, xstats_names, NULL, len)) {
   printf("Cannot get xstat names\n");
   goto err;
}

values = malloc(sizeof(values) * len);
if (values == NULL) {
   printf("Cannot allocate memory for xstats\n");
   goto err;
}

/* Getting xstats values */
if (len != rte_eth_xstats_get_by_id(port_id, NULL, values, len)) {
   printf("Cannot get xstat values\n");
   goto err;
}

/* Print all xstats names and values */
for (i = 0; i < len; i++) {
   printf("%s: %"PRIu64"\n", xstats_names[i].name, values[i]);
}
```

该应用程序可以访问 PMD 暴露的所有统计信息的名称。应用程序自己决定哪些统计信息是感兴趣的，通过查找名称来缓存这些统计信息的 ID。

```
uint64_t id;
uint64_t value;
const char *xstat_name = "rx_errors";

if(!rte_eth_xstats_get_id_by_name(port_id, xstat_name, &id)) {
   rte_eth_xstats_get_by_id(port_id, &id, &value, 1);
```

```
   printf("%s: %"PRIu64"\n", xstat_name, value);
}
else {
   printf("Cannot find xstats with a given name\n");
   goto err;
}
```

API 为应用程序提供了灵活性，以便它可以使用包含多个 ID 号的数组查找多个统计信息。这样可以减少检索统计信息的函数调用开销，并使得应用程序更统一查找多个统计信息。

```
#define APP_NUM_STATS 4
/* application cached these ids previously; see above */
uint64_t ids_array[APP_NUM_STATS] = {3,4,7,21};
uint64_t value_array[APP_NUM_STATS];

/* Getting multiple xstats values from array of IDs */
rte_eth_xstats_get_by_id(port_id, ids_array, value_array, APP_NUM_STATS);

uint32_t i;
for(i = 0; i < APP_NUM_STATS; i++) {
   printf("%d: %"PRIu64"\n", ids_array[i], value_array[i]);
}
```

用于 xstats 数组查找的 API 允许应用程序创建多个统计信息组，并使用单个 API 调用查找这些 ID 值。作为最终查找结果，应用程序能够实现监视单个统计信息（在这种情况下为 rx_errors）的目标，如果显示数据包被丢弃，则可以使用 ID 数组轻松地检索组统计信息。

8. 通用流 API

8.1. 概述

此 API 提供了一种通用的方式来配置硬件以匹配特定的 Ingress 或 Egress 流量，根据用户的任何配置规则更改其操作或查询相关计数器。

所有 API 带有 rte_flow 前缀，在文件 rte_flow.h 中定义。

- 可以对报文数据(如协议头部，载荷)及报文属性(如关联的物理端口，虚拟设备 ID 等)执行匹配。
- 可能的操作包括丢弃流量，将流量转移到特定队列、虚拟/物理设备或端口，执行隧道解封、添加标记等操作。

它比其涵盖和替代（包括所有功能和过滤类型）的传统过滤框架层次更高，以便发布所有轮询模式驱动程序（PMD）明确行为的通用操作接口。

迁移现有应用程序的几种方法在 API 迁移中有描述。

8.2. 流规则

8.2.1. 描述

流规则是具有匹配模式的属性和动作列表的组合。流规则构成了此 API 的基础。

一个流规则可以具有几个不同的动作(如在将数据重定向到特定队列之前执行计数，封装，解封装等操作)，而不是依靠几个规则来实现这些动作，并且通过应用程序操作具体的硬件实现细节来顺序执行这些规则。

API 提供了基于规则的不同优先级支持，例如，当报文匹配两个规则时，强制先执行特定规则。然而，硬件是否支持多个优先级并不能保证。即使硬件支持，可用优先级的数量通常也较低，这也是为什么还需要通过 PMDs 在软件中实现(如通过重新排序规则可以模拟缺失的优先级)。

为了尽可能保持与硬件无关，默认情况下所有规则都被认为具有相同的优先级，这意味着重叠规则（当数据包被多个过滤器匹配时）之间的顺序是未定义的，不能保证谁先执行。

当给定一个优先级时，PMD 如果能够检测到的话（例如，如果模式匹配现有过滤器），可能会拒绝在此优先级下创建重叠规则。

因此，对于给定的优先级，可预测的结果只能通过非重叠规则来实现，在所有协议层上使用完全匹配。

流规则也可以分组，流规则的优先级特定于它们所属的组。因此，给定组中的所有流规则在另一个流规则组所有规则之前或之后。

根据规则支持多个操作可以在非默认硬件优先级之前内部实现，因此两个功能可能不能同时应用于应用程序。？？？？

考虑到允许的模式/动作组合不能提前知道，并且将导致不切实际地大量的暴露能力，提供了从当前设备配置状态验证给定规则的方法。这样，在启动数据路径之前，应用程序可以检查在初始化时是否支持所需的规则类型。该方法可以随时使用，其唯一要求是应该存在规则所需的资源（例如，应首先配置目标 RX 队列）。

每个定义的规则与由 PMD 管理的不透明句柄相关联，应用程序负责维护它。这些句柄可用于查询和规则管理，例如检索计数器或其他数据并销毁它们。为了避免 PMD 方面的资源泄漏，在释放相关资源（如队列和端口）之前，应用程序必须显式地销毁句柄。

以下小节覆盖如下内容：

- 属性（由 struct rte_flow_attr 表示）：流规则的属性，例如其方向（Ingress 或 Egress）和优先级。
- 模式条目（由 struct rte_flow_item 表示）：匹配模式的一部分，匹配特定的数据包数据或流量属性。也可以描述模式本身属性，如反向匹配。
- 匹配模式：要查找的流量属性，组合任意的模式。
- 动作（由 struct rte_flow_action 表示）：每当数据包被模式匹配时执行的操作。

8.2.2. 属性

8.2.2.1. Group

流规则可以通过为其分配一个公共的组号来分组。较低的值具有较高的优先级。组 0 具有最高优先级。

虽然是可选的，但是建议应用程序尽可能将类似的规则分组，以充分利用硬件功能（例如，优化的匹配）并解决限制（例如，给定组中可能只允许单个模式类型）。

请注意，并不保证支持多个组。

8.2.2.2. Priority

可以将优先级分配给流规则。像 Group 一样，较低的值表示较高的优先级，0 为最大值。

具有优先级 0 的 Group 8 流规则，总是在 Group 0 优先级 8 的优先级之后才匹配（Group 的优先级先得到保证）。

组和优先级是任意的，取决于应用程序，它们不需要是连续的，也不需要从 0 开始，但是最大数量因设备而异，并且可能受到现有流规则的影响。

如果某个报文在给定的优先级和 Group 中被几个规则匹配，那么结果是未定义的。 它可以采取任何路径，可能重复，甚至导致不可恢复的错误。

请注意，不保证能支持超过一个优先级。

8.2.2.3. Traffic direction

流量规则可以应用于入站和/或出站流量（Ingress/Egress）。

多个模式条目和操作都是有效的，可以在两个方向中使用。但是必须至少指定一个方向。

不推荐对给定规则一次指定两个方向，但在少数情况下可能是有效的（例如共享计数器）。

8.2.3. 模式条目

模式条目分成两类：

- 匹配协议头部及报文数据（ANY，RAW，ETH，VLAN，IPV4，IPV6，ICMP，UDP，TCP，SCTP，VXLAN，MPLS，GRE 等等），通常关联一个规范结构。
- 匹配元数据或影响模式处理（END，VOID，INVERT，PF，VF，PORT 等等），通常没有规范结构。

条目规范结构用于匹配协议字段（或项目属性）中的特定值。文档描述每个条目是否与一个条目及其类型名称相关联。

可以为给定的条目最多设置三个相同类型的结构：

- Spec：要匹配的数值（如 IPv4 地址）。
- Last：规格中的相应字段的范围上限。
- Mask：应用于 spec 和 last 的位掩码（如匹配 IPv4 地址的前缀）。

使用限制和期望行为：

- 没有 spec 就设置 mask 和 last 是错误的。
- 错误值如 0 或者等于 spce 中相应值的 last 字段将被忽略，他们不产生范围。不支持低于 spce 的非 0 值。
- 设置 spce 和可选的 last，而不设置 mask 会导致 PMD 使用该条目定义的默认 mask（定义为 rte_flow_item_{name}_mask 常量）。
- 不设置任何值（如果支持）相当于提供空掩码的广泛匹配。
- 掩码是用于 spec 和 last 的简单位掩码，如果不小心使用，可能会产生意想不到的结果。例如，对于 IPv4 地址字段，spec 提供 10.1.2.3，last 提供 10.3.4.5，掩码为 255.255.0.0，有效范围为 10.1.0.0～10.3.255.255。

匹配以太网头部的条目示例：

Table 8.1 Ethernet item

Field	Subfield	Value
Spec	Src	00:01:02:03:04
	Dst	02:2a:66:00:01
	Type	0x22aa
last	Unspecified	
mask	Src	00:ff:ff:ff:00
	Dst	00:00:00:00:ff
	Type	0x0000

无掩码的位表示任意的值（如下面显示的？），根据上面的条目，具有如下的属性以太头部的报文将被匹配：

- src：??:01:02:03:??
- dst：??:??:??:??:01
- type：0x????

8.2.4. 匹配模式

通过堆叠方式从最底层协议开始匹配条目的方式形成模式。这种堆叠限制不适用于那些可以放在任意位置，但是不影响匹配结构的元条目。

模式由 END 条目终结。

例子：

Table 8.2 TCPv4 as L4

Index	Item
0	Ethernet
1	IPv4
2	TCP
3	END

Table 8.3 TCPv6 in VXLAN

Index	Item
0	Ethernet
1	IPv4
2	UDP
3	VXLAN
4	Ethernet
5	IPv6

6	TCP
7	END

Table 8.4 TCPv4 as L4 with meta items

Index	Item
0	VOID
1	Ethernet
2	VOID
3	IPv4
4	TCP
5	VOID
6	VOID
7	END

这个例子显示了元条目如何不影响匹配结果，只要他们保持堆叠正确。这个例子得到的匹配结果与模式“TCPv4 as L4”相同。

Table 8.5 UDPv6 anywhere

Index	Item
0	IPv6
1	UDP
2	END

假如 PMD 支持，如上述例子，缺少 Ethernet 规范，忽略堆栈底部的一个或多个协议层，也可以匹配数据包总的任意指定位置。

Table 8.6 Invalid, missing L3

Index	Item
0	Ethernet
1	UDP
2	END

由于 L2（以太网）和 L4（UDP）之间的 L3 规范缺失，上述模式无效。也就是说，只允许在堆叠的底部或顶部忽略协议层。

8.2.5. 元条目类型

元条目只匹配元数据或影响模式处理过程，而不是直接匹配数据包数据，一般不需要规范结构。这种特殊性允许他们在堆栈中的任何位置，而不会对匹配结果造成影响。

8.2.5.1. END 条目

条目列表的结束标记。阻止进一步处理条目，从而结束模式匹配。

- 为了方便起见，其数值为 0。
- PMD 必须强制支持这个条目。
- 忽略 spec、last、mask 域。

Table 8.7 END

Field	Value
spec	ignored
last	Ignored
mask	Ignored

8.2.5.2. VOID 条目

方便起见，用作占位符，被 PMD 忽略并简单丢弃，跳过不处理。

- PMD 必须强制支持这个条目。
- 忽略 spec、last、mask 域。

Table 8.8 VOID

Field	Value
spec	ignored
last	Ignored
mask	Ignored

此类型条目的一个使用情景是快速生成共享共用前缀的规则，而无需重新分配内存，仅需要更新条目类型。

Table 8.9 TCP, UDP or ICMP as L4

Index	Item		
0	Ethernet		
1	IPv4		
2	UDP	VOID	VOID
3	VOID	TCP	VOID
4	VOID	VOID	ICMP
5	END		

8.2.5.3. INVERT 条目

反向匹配，即与模式不匹配的数据包的处理。

- 忽略 spec、last、mask 域。

Table 8.10 INVERT

Field	Value
spec	ignored

last	Ignored
mask	Ignored

下面的使用场景，匹配非 TCPv4 的报文：

Table 8.11 Anything but TCPv4

Index	Item
0	INVERT
1	Ethernet
2	IPv4
3	TCP
4	END

8.2.5.4. PF 条目

匹配寻址到设备物理功能的数据包。

如果底层设备功能与正常接收到匹配流量的功能不同，则指定此项可防止报文到达该设备，除非流规则包含 Action: PF。默认情况下，设备实例之间的数据包不会重复。

- 如果条目应用于 VF 设备，可能返回错误或不匹配任何流量。
- 可以和任意数据的条目组合：VF 组合以匹配 PF 或者 VF 流量。
- spec、last、mask 不能设置。

Table 8.12 PF

Field	Value
spec	unset
last	unset
mask	unset

8.2.5.5. VF 条目

匹配寻址到设备虚拟功能 ID 的数据包。

如果底层设备功能与正常接收匹配流量的功能不同，则指定此项可防止报文到达该设备，除非流规则包含 Action：VF。默认情况下，设备实例之间的数据包不会重复。

- 如果这导致 VF 设备匹配到不同 VF 的流量，则可能返回错误或不匹配任何流量。
- 可以指定多次以匹配寻址到多个 VF ID 的流量。
- 可以与 PF 项目组合以匹配 PF 和 VF 流量。
- 默认掩码匹配任何 VF ID。

Table 8.13 VF

Field	Subfield	Value
spec	Id	destination VF ID
last	Id	upper range value
mask	id	zeroed to match any VF ID

8.2.5.6. PORT 条目

匹配来自指定底层设备物理端口的数据包。

第一个 PORT 条目覆盖的物理端口通常与指定的 DPDK 输入端口（port_id）相关联。该条目可以提供多次以匹配其他物理端口。

请注意，当这些端口不在 DPDK 控制下时，物理端口不一定与 DPDK 输入端口（port_id）绑定。可能的值是特定于每个设备，它们不一定从零开始，并且可能不是连续的。

作为设备属性，可以通过其他方式检索允许的值列表以及与 port_id 关联的值。

Table 8.14 PORT

Field	Subfield	Value
spec	index	destination port index
last	index	upper range value
mask	index	zeroed to match any port index

8.2.6. 数据匹配条目类型

大多数的数据匹配条目是具有位掩码的基本协议头部定义。必须从最底层到最高层协议指定（堆叠方式）以形成匹配模式。

下面的描述并不详尽，将来会添加新的协议。

8.2.6.1. ANY

匹配当前层的任何协议，单个 ANY 也可以代表多个协议层。
当在数据包中的任意位置寻找协议时，通常将其指定为第一个模式条目。

- 默认掩码匹配任何数目的协议层。

Table 8.15 ANY

Field	Subfield	Value
spec	num	覆盖的层数
last	num	最大范围
mask	num	匹配任意层数

例如 VXLAN TCP 负载外部 L3（IPv4 或 IPv6）及 L4（UDP）使用第一个 ANY 来匹配，内部 L3（IPv4 或 IPv6）用第二个 ANY 来匹配：

Table 8.16 TCP in VXLAN with wildcards

Index	Item	Field	Subfield	Value

0	Ethernet			
1	ANY	Spec	Num	2
2	VXLAN			
3	Ethernet			
4	ANY	Spec	Num	1
5	TCP			
6	END			

8.2.6.2. RAW

匹配指定偏移量下指定长度的字节串。

偏移量可以是是绝对偏移（使用数据包开始）或者相对于堆栈中先前匹配项的结尾，相对偏移量允许为负值。

如果启用搜索，则使用偏移量作为起点。搜索区域可以通过将限制设置为非零值来定界，该值可以是可以开始模式的偏移量后的最大字节数。

允许匹配 0 长度，这样做会重置后续项目的相对偏移量。

- 这个条目不支持区间（last）。
- 默认的掩码精确匹配所有字段。

Table 8.17 RAW

Field	Subfield	Value
spec	relative	上一个条目之后的搜索模式
	search	搜索模式
	reserved	预留，必须为 0
	offset	绝对/相对偏移量
	limit	搜索区域限制
	length	模式长度
	pattern	要匹配的字节串
last	如果指定，必须全 0 或者与 spec 相等	
mask	应用于 spec	

使用组合的 RAW 条目在 UDP 有效载荷的各种偏移量下查找几个字符串的示例模式：

Index	Item	Field	Subfield	Value
0	Ethernet			
1	IPv4			
2	UDP			
3	RAW	spec	relative	1

			search	1
			offset	10
			limit	0
			length	3
			pattern	“foo”
4	RAW	spec	relative	1
			search	0
			offset	20
			limit	0
			length	3
			pattern	“bar”
5	RAW	spec	relative	1
			search	0
			offset	-29
			limit	0
			length	3
			pattern	“baz”
6	END			

含义如下：

- 在 UDP 有效载荷内偏移 10 个自己的地方匹配“foo”。
- 在“foo”之后 20 个字节的地方匹配“bar”。
- 在“bar”向后退 29 字节的地方匹配“baz”。

满足这样条件的报文可以表示为如下：

```
0                     >= 10 B           == 20 B
|                  |<--------->|     |<--------->|
|                  |           |     |           |
|-----|------|-----|-----|-----|-----|-----------|-----|------|
| ETH | IPv4 | UDP | ... | baz | foo | ......... | bar | .... |
|-----|------|-----|-----|-----|-----|-----------|-----|------|
          |                                 |
          |<------------------------------->|
             == 29 B
```

注意，匹配模式后续条目将在”baz”之后恢复，而不是“bar”，因为总是在堆栈的先前项之后执行匹配。

8.2.6.3. ETH

匹配以太头部。

- dst：目的 MAC。
- src：源 MAC。

- type：EtherType。
- 默认掩码仅匹配源 MAC 和目的 MAC。

8.2.6.4. VLAN

匹配 802.1Q/ad VLAN tag。
- tpid：标签协议标识符。
- tci：标签控制信息。
- 默认掩码仅匹配 tci。

8.2.6.5. IPv4

匹配 IPv4 头部。注意，IPv4 选项由专用模式条目处理。
- hdr：IPv4 头部定义（ret_ip.h）。
- 默认掩码仅匹配源和目的 IP 地址。

8.2.6.6. IPv6

匹配 IPv6 头部。注意，IPv6 选项由专用模式条目处理。
- hdr：IPv6 头部定义（ret_ip.h）。
- 默认掩码仅匹配源和目的 IPv6 地址。

8.2.6.7. ICMP

匹配 ICMP 头部。
- hdr：ICMP 头部定义（ret_icmp.h）。
- 默认掩码仅匹配 ICMP 类型及代码。

8.2.6.8. UDP

匹配 UDP 头部。
- hdr：UDP 头部定义（ret_udp.h）。
- 默认掩码仅匹配源端口和目的端口。

8.2.6.9. TCP

匹配 TCP 头部。
- hdr：UDP 头部定义（ret_tcp.h）。
- 默认掩码仅匹配源端口和目的端口。

8.2.6.10. SCTP

匹配 SCTP 头部。
- hdr：UDP 头部定义（ret_sctp.h）。

- 默认掩码仅匹配源端口和目的端口。

8.2.6.11. VXLAN

匹配 VXLAN 头部（RFC 7348）。
- flags：通常是 0x80。
- rsvd0：预留，通常为 0x00000。
- vni：VXLAN 网络标识符。
- rsvd1：预留，通常为 0x00。
- 默认掩码仅匹配 vni。

8.2.6.12. E_TAG

匹配 IEEE 802.1BR E-Tag。
- tpid：标签协议标识符，通常是 0x893F。
- epcp_edei_in_ecid_b：E-TAG 控制信息（E-TCI），E-PCB（3b），E-DEI（1b），ingress E-CDI base（12b）。
- rsvd_grp_ecid_b：reserver（2b），GRP（2b），C-CID base（12b）。
- in_ecid_e：ingress E-CID ext。
- ecid_e：E-CID ext。
- 默认掩码同时匹配 GRP 和 E-CID base。

8.2.6.13. NVGRE

匹配 NVGRE 头部（RFC7637）。
- c_k_s_rsvd0_ver：checksum（1b），key bit（1b），seq number（1b），reserved 0（9b），version（3b）。根据 RFC7637，这个字段必须是 0x2000。
- protocol：协议类型 0x6558.
- tni：虚拟子网 ID。
- flow_id：流 ID。
- 默认掩码只匹配 TNI。

8.2.6.14. MPLS

匹配 MPLS 头部。
- label_tc_s_ttl：label，TC，Bottom of stack 及 ttl。
- 默认掩码只匹配 label。

8.2.6.15. GRE

匹配 GRE 头部。
- c_rsvd0_ver：checksum，rsvd0，version。
- protocol：协议类型。
- 默认掩码只匹配协议类型。

8.2.7. 动作

每个可能的动作都由一个类型来表示。一些动作具有相关的配置结构。列表中组合的几个操作可能会影响流规则。列表并不有序。

动作分成以下三种：

- 终结动作（如 QUEUE、DROP、RSS、PF、VF），防止后续流程处理匹配的数据包，除非动作被 PASSTHRU 覆盖。
- 非终结动作（如 PASSTHRU、DUP），将匹配的数据包保留以便后续的流程规则进行额外的处理。
- 非终结元动作（END、VOID、MARK、FLAG、COUNT）等不会影响数据包最终结果的动作。

当流规则中组合了多个动作时，他们应该具有不同的类型（如丢弃数据包两次是不可能的）。相同类型的动作，只有最后一个会被考虑到，但是 PMD 仍然在动作链表中执行错误检查。

类似于匹配模式，动作链表被 END 条目终结。注意，只有 PASSTHUR 才能覆盖终结规则。
下面实例为将数据包重定向到队列索引为 10 的队列：

Table 8.18 Queue action

Field	Value
index	10

动作列表示例，其顺序不重要，应用程序必须考虑同时执行所有的动作：

Table 8.19 Count and drop

Field	Action
0	COUNT
1	DROP
2	END

Table 8.20 Mark, count and redirect

Index	Action	Field	Valie
0	MARK	make	0x2a
1	COUNT		
2	QUEUE	queue	10
3	END		

Table 8.21 Redirect to queue 5

Index	Action	Field	Valie
0	Drop		
1	QUEUE	queue	5
2	END		

在上例中，考虑到同时执行这两个动作，最终的结果只有 QUEUE 有效果。

Table 8.22 Redirect to queue 3

Index	Action	Field	Valie
0	QUEUE	queue	5
1	VOID		
2	QUEUE	queue	3
3	END		

如前所述，仅考虑列表中相同类型的最后一个动作。这个例子也显示了 VIOD 被忽略。

8.2.8. 动作类型

通用动作类型将在本节描述。与模式条目类型一样，这个列表并不完全，后续将添加新的动作类型。

8.2.8.1. END

END 标记动作列表结束，防止进一步处理动作。

- 方便起见，其数值为 0。
- PMD 需要强制支持。
- 不配置任何属性。

Table 8.23 END

Field
no properties

8.2.8.2. VOID

方便起见，用作占位符。它被 PMD 简单的忽略处理。

- PMD 需要强制支持。
- 不配置任何属性。

Table 8.24 VOID

Field
no properties

8.2.8.3. PASSTHUR

将报文留给后续流规则进行额外的处理。当流规则不包含终结动作时，这个操作时默认的，当时可以指定强制规则变为非终止。

- 不配置任何属性。

Table 8.25 PASSTHRU

Field

no properties

将数据包复制到队列并按照后续流规则继续处理的示例：

Table 8.26 Copy to queue 8

Index	Action	Field	Valie
0	PASSTHUR		
1	QUEUE	queue	8
2	END		

8.2.8.4. MARK

将一个整形数值附加到数据包并设置 PKT_RX_FDIR 和 PKT_RX_FDIR_ID mbuf 标志。

该值的意义可以由应用程序任意指定。最大允许值取决于底层实现。它在 hash.fdir.hi mbuf 字段中返回。

Table 8.27 MARK

Field	Value
id	数据包返回的整形数值

8.2.8.5. FLAG

标记数据包。与 MARK 动作类似，只是没有特定值，只能设置 PKT_RX_FDIR mbuf 标志。

Table 8.28 FLAG

Field
no properties

8.2.8.6. QUEUE

将报文重定向到指定队列。

- 默认为终结动作。

Table 8.29 QUEUE

Field	Value
index	队列索引

8.2.8.7. DROP

丢弃报文。

- 不配置任何属性。
- 默认为终结动作。
- 如果指定， PASSTHUR 可以覆盖这个动作。

Table 8.30 DROP

Field
no properties

8.2.8.8. COUNT

在规则中使能计数器。
这些计数器可以通过 rte_flow_query()检索和重置，具体请参阅 rte_flow_query_count 结构体描述。

- 计数器可以通过 rte_flow_query()检索。
- 不配置任何属性。

Table 8.31 COUNT

Field
no properties

检索和重置流量规则计数器的查询结构：
Table 8.32 COUNT query

Field	IO	Value
reset	in	查询之后重置计数器
hit_set	out	hits 字段设置
bytes_set	out	bytes 字段设置
hits	out	此规则 hit 的次数
bytes	out	此规则经过的 bytes

8.2.8.9. DUP

复制报文给指定的队列。
这个动作通常与 QUEUE 结合，但当单独使用时，实际上类似于 QUEUE+PASSTHUR。

- 默认为非终结动作。

Table 8.33 DUP

Field	Value
index	队列索引

8.2.8.10. RSS

与 QUEUE 类似，但是根据参数在数据包上额外执行 RSS 以将他们扩展到多个队列中。

注意：RSS 哈希结果存储在 hash.rss mbuf 字段中，该字段与 hash.fdir.lo 重叠。由于 MARK 动作仅设置 hash.fdir.hi 字段，因此可以同时查询这两个字段。

- 默认为终结动作。

Table 8.34 RSS

Field	Value
rss_conf	RSS 参数
num	Queue 队列中的数目
queue[]	使用的 Queue

8.2.8.11. PF

将数据包重定向到当前设备的物理功能（PF）。

- 不配置任何属性。
- 默认为终结动作。

Table 8.35 PF

Field
no properties

8.2.8.12. VF

将数据包重定向到当前设备的虚拟功能（VF）。
由 VF 模式项匹配的数据包可以重定向到其原始 VF ID 而不是指定的 VF ID。如果 VF 部分与先前的流规则相匹配，或者首先将数据包未发送到 VF，则此参数可能不可用，并且不能保证其正常工作。

- 默认为终结动作。

Table 8.36 VF

Field	Value
original	使用原始的 VF ID
vf	VF ID

8.2.9. 负数类型

所有指定的模式条目（枚举 rte_flow_item_type）和动作条目（枚举 rte_flow_action_type）都使用正的标识符。

负数空间用于在运行期间由 PMD 生成的动态类型。PMD 可能会遇到他们，但不能接受他们不知道的负数标识符。

生成负数类型的方法有待定义。

8.2.10. 计划扩展类型

随着新协议的实现，将添加模式条目类型。

通过专用模式项支持变量头，例如为了匹配特定的 IPv4 选项，IPv6 扩展头将在 IPv4 / IPv6 项目之后堆叠。

其他动作类型已经在计划，但尚未定义。包括以多种方式改变分组数据的能力，例如执行隧道报头的封装/解封装等。

DPDK 提供了一个极简的 API 用于所有的流规则管理。

每个创建的流规则都与 PMD 中不透明的句柄相关联，应用程序负责维护它，直到流规则被删除为止。

流规则由结构体 struct rte_flow 对象表示。

8.2.11. 验证

鉴于完全表达一组确定的设备功能是不实际的，因此，DPDK 提供了专用的函数来检查流规则是否被支持，并且可以被创建。

```
int
rte_flow_validate(uint8_t port_id,
		const struct rte_flow_attr *attr,
		const struct rte_flow_item pattern[],
		const struct rte_flow_action actions[],
		struct rte_flow_error *error);
```

这个函数验证流规则是否正确，以及是否可以由设备给定的资源所接受。根据当前设备模式和队列配置检查规则。还可以根据现有流规则和设备资源选择性地验证规则。此函数对目标设备无影响。

由于可能的冲突或资源限制，只要在同一时间没有成功调用 rte_flow_create() 或者 rte_flow_destroy()，并且没有任何影响流规则的设备参数被修改，那么返回值可以保证有效。

参数：

- port_id：以太设备端口标识。
- attr：流规则属性。
- pattern：指定模式（以 END 为结尾的链表）。
- actions：关联动作（以 END 为结尾的链表）。
- error：如果不为 NULL，执行详细错误报告。PMD 初始化这个结构以防止出错。

返回值：

- 如果流规则有效，并且可以被创建，那么返回 0。否则返回一个负的 error 值，定义为如下。
- -ENOSYS：底层设备不支持此功能。
- -EINVAL：未知或无效的流规则。
- -ENOTSUP：有效但不支持的规则（例如不支持部分位掩码）。
- EEXIST：与现有规则冲突。仅当设备支持流规则冲突检查并且存在流规则冲突时才返回这个值，如果不处理此返回值，则不能保证创建的流规则不会因为冲突而失败。
- ENOMEM：内存不足，或者设备支持资源验证，但是设备上的资源有限制。
- -EBUSY：由于设备繁忙，无法执行操作，如果受影响的队列甚至整个端口处于停止状态（参考接口 rte_eth_dev_rx_queue_stop()及 rte_eth_dev_stop()），则可能会执行。

8.2.12. 创建

创建流规则与验证流规则类型，除了实际创建规则并返回句柄。

```
struct rte_flow *
rte_flow_create(uint8_t port_id,
        const struct rte_flow_attr *attr,
        const struct rte_flow_item pattern[],
        const struct rte_flow_action *actions[],
        struct rte_flow_error *error);
```

参数：

- port_id：以太设备端口标识。
- attr：流规则属性。
- pattern：指定模式（以 END 为结尾的链表）。
- actions：关联动作（以 END 为结尾的链表）。
- error：如果不为 NULL，执行详细错误报告。PMD 初始化这个结构以防止出错。

返回值：
在创建成功时返回一个有效的句柄，否则为 NULL，并且 rte_errno 设置为 rte_flow_vaildate()定义的错误代码的正值。

8.2.13. 销毁

流规则的删除不是自动完成的，如果任何队列或端口仍然引用他们，则不应该释放。应用程序必须在释放资源之前执行此步骤。

```
int
rte_flow_destroy(uint8_t port_id,
        struct rte_flow *flow,
        struct rte_flow_error *error);
```

当其他流规则依赖于它时，删除流规则句柄可能会失败，并且删除操作可能会造成不一致的状态。

如果句柄与创建顺序相反的顺序执行销毁，则可以保证成功。

参数：

- port_id：以太设备端口标识。
- flow：要销毁的流规则。
- error：如果不为 NULL，执行详细错误报告。PMD 初始化这个结构以防止出错。

返回值：

- 成功返回 0，失败返回负值，且 rte_errno 被设置。

8.2.14. 清空

一个更加方便的功能，可以用于销毁与 port 关联的所有流规则。通过连续调用 rte_flow_destory()来实现。

```
int
rte_flow_flush(uint8_t port_id,
        struct rte_flow_error *error);
```

当执行失败时（不太可能发生），流规则句柄仍然视为已经销毁，不再有效，但是端口必须处于不一致的状态。

参数：

- port_id：以太设备端口标识。
- error：如果不为 NULL，执行详细错误报告。PMD 初始化这个结构以防止出错。

返回值：

- 成功返回 0，失败返回负值，且 rte_errno 被设置。

8.2.15. 查询

查询一个存在的流规则。
此功能允许检索特定于流的数据，如计数器等。数据通过必须存在于流规则定义中的特殊操作来收集。

```
int
rte_flow_query(uint8_t port_id,
        struct rte_flow *flow,
        enum rte_flow_action_type action,
        void *data,
        struct rte_flow_error *error);
```

参数：

- port_id：以太设备端口标识。
- flow：要查询的流规则句柄。
- actions：查询的动作类型。
- data：用于存储相关查询数据类型的指针
- error：如果不为 NULL，执行详细错误报告。PMD 初始化这个结构以防止出错。

返回值：

- 成功返回 0，失败返回负值，且 rte_errno 被设置。

8.3. 详细错误报告

对于想要调查与流规则管理有关的问题的用户或应用程序开发人员，定义的错误值可能不够详细。为此目的定义了专门的错误对象：

```
enum rte_flow_error_type {
   RTE_FLOW_ERROR_TYPE_NONE, /**< No error. */
   RTE_FLOW_ERROR_TYPE_UNSPECIFIED, /**< Cause unspecified. */
   RTE_FLOW_ERROR_TYPE_HANDLE, /**< Flow rule (handle). */
   RTE_FLOW_ERROR_TYPE_ATTR_GROUP, /**< Group field. */
   RTE_FLOW_ERROR_TYPE_ATTR_PRIORITY, /**< Priority field. */
   RTE_FLOW_ERROR_TYPE_ATTR_INGRESS, /**< Ingress field. */
   RTE_FLOW_ERROR_TYPE_ATTR_EGRESS, /**< Egress field. */
   RTE_FLOW_ERROR_TYPE_ATTR, /**< Attributes structure. */
   RTE_FLOW_ERROR_TYPE_ITEM_NUM, /**< Pattern length. */
   RTE_FLOW_ERROR_TYPE_ITEM, /**< Specific pattern item. */
   RTE_FLOW_ERROR_TYPE_ACTION_NUM, /**< Number of actions. */
   RTE_FLOW_ERROR_TYPE_ACTION, /**< Specific action. */
};

struct rte_flow_error {
   enum rte_flow_error_type type; /**< Cause field and error types. */
   const void *cause; /**< Object responsible for the error. */
   const char *message; /**< Human-readable error message. */
};
```

错误类型 RTE_FLOW_ERROR_TYPE_NONE 代表无错误，在这种情况下可以忽略其他字段。其他错误类型描述了由 cause 指向的对象类型。

如果 cause 不为 NULL，cause 指向的对象用于描述错误。对于一个流规则，这个对象可能是个模式条目或单个动作。

该对象通常由应用程序分配，并在 PMD 发生错误的情况下由 PMD 设置，消息指向不需要被应用程

序释放的常量字符串，但是只要它的关联 DPDK 端口保持配置。关闭底层设备或卸载 PMD 使其无效。

8.4. 注意事项

- DPDK 不会自动跟踪流规则定义或流规则对象。应用程序必须跟踪后者，也可以跟踪前者。PMD 每部也可以实现，但是应用程序不能依赖于此。
- 在连续的端口初始化之间不保留流规则。应用程序退出而没有释放的话，重启时必须重新创建。
- API 操作是同步和阻塞的（EAGAIN 不能作为返回值）。
- 尽管没什么措施来防止不同的设备被同时配置，仍然没有方法保证重入/多线程安全，需要特别注意。
- 管理流规则时，不需要停止数据路径（RX/TX）。如果不能自然地实现或解决，则必须返回适当的错误代码（EBUSY）。
- PMD 负责在停止和重新启动端口或执行可能影响端口的其他操作时维护流规则配置，而不是应用程序。但是流规则只能被应用程序明确地销毁。

对于暴露多个端口共享流规则影响的全局设置的设备：
- DPDK 控制下的多有端口必须一致，PMD 负责确保端口上的现有流规则不受其他端口的影响。
- 不受 DPDK 控制的端口，由用户负责处理。他们可能会影响现有的流规则并导致未定义的行为。感知这种情况的 PMD 可能会阻止流规则的创建。

8.5. PMD 接口

PMD 接口在 rte_flow_driver.h 中定义。它不受 API/ABI 版本控制的约束，因为它不会暴露于应用程序，并可能独立发展。

目前，它通过过滤器类型 RTE_ETH_FILTER_GENERIC 实现了传统过滤框架，该类型接受单个操作 RTE_ETH_FILTER_GET 以返回包含在 struct rte_flow_ops 内的 PMD 特定的 rte_flow 回调。

为了保持与遗留过滤框架的兼容性，这个开销是暂时必需的，但是这些框架应该最终消失。

- PMD 回调完全实现了规则管理中描述的接口，除了已经转换为指向底层结构 rte_eth_dev 的指针的端口 ID 参数外。
- 在调用 PMD 函数之前，公共 API 函数根本不处理流规则定义（无基本错误检查，无任何验证）。 他们只确保这些回调非 NULL 或返回 ENOSYS（不支持功能）错误。

此接口另外定义了以下帮助函数：

- rte_flow_ops_get()：从端口获取泛型流操作结构。
- rte_flow_error_set()：初始化通用流错误结构。

8.6. 设备兼容性

没有已知的实现可以支持所有描述的功能。

出于性能原因，PMD 不支持的功能或组合不会在软件中完全模拟实现。硬件执行大部分工作（例如队列重定向和数据包识别），部分支持的功能在软件中完成。

只要这样做不会影响现有流规则的行为，PMD 将尽力通过解决硬件限制来满足应用程序请求。

以下部分提供了这种情况的一些示例，并描述了 PMD 如何处理它们，它们基于之前的 API 中内置的限制。

8.6.1. 全局的 bit-mask

每个流规则都带有自己的每层位掩码，而硬件可能仅支持给定层类型的单个设备范围的位掩码，以便两个 IPv4 规则不能使用不同的位掩码。
在这种情况下，预期的行为是 PMD 根据创建的第一个流规则的需要自动配置全局位掩码。
仅当其位掩码与其匹配时才允许后续规则，否则应返回 EEXIST 错误代码。

8.6.2. 不支持的 layer 类型

可以通过使用 RAW 类型条目来模拟许多协议。
PMD 可以依靠此功能来模拟对不被硬件直接识别的头部的协议的支持。

8.6.3. ANY 模式条目

这种模式条目代表任何东西，这可能难以转换为硬件理解的信息，特别是如果跟随更具体的类型。

考虑如下的模式：
Table 8.37 Pattern with ANY as L3

Index	Item		
0	Ethernet		
1	ANY	num	1
2	TCP		
3	END		

我们知道 TCP 对 IPv4 和 IPv6 之外的其他东西并不合适，这样的模式可能被转换为两个流规则：
Table 8.38 ANY replaced with IPV4

Index	Item
0	Ethernet

1	IPv4(zeroed mask)
2	TCP
3	END

Table 8.39 ANY replaced with IPV6

Index	Item
0	Ethernet
1	IPv6(zeroed mask)
2	TCP
3	END

请注意，一旦任何规则覆盖了多个层次，这种方法可能会产生大量隐藏的流规则。 因此建议仅支持最常见的情况（ANY 处于 L2 和/或 L3）。

8.6.4. 不支持的动作

- 当与 QUEUE 动作组合时，只要目标队列由单个规则使用，可以在软件中实现数据包计数（COUNT 动作）和标记（MARK 动作或 FLAG 动作）。
- 指定动作 DUP + QUEUE 的规则可能会转换为两个隐藏的规则 QUEUE 和 PASSTHRU 动作。
- 当提供单个目标队列时，动作 RSS 也可以通过 QUEUE 来实现。

8.6.5. 流规则优先级

虽然自然是有意义的，但是由于以下几个原因，流量规则不能被假定为与其创建相同的顺序由硬件处理：

- 它们可以作为树或哈希表而不是列表在内部进行管理。
- 在添加另一个流规则之前删除流规则可以将新规则放在列表的末尾或重新使用释放的条目。
- 当数据包被几个规则匹配时，可能会发生重复。

对于重叠的规则（特别是为了使用动作 PASSTHRU），可预测的行为只能通过使用不同的优先级来保证。

优先级不一定在硬件中实现，或者可能受到严重限制（例如单个优先级位）。

由于这些原因，优先级可以纯粹在 PMD 的软件中实现。

- 对于期望以正确顺序添加流规则的设备，PMD 可能会在添加具有较高优先级的新规则之后破坏并重新创建现有规则。
- 可以在初始化时间创建可配置数量的空或空规则，以节省高优先级槽位供以后使用。
- 为了节省优先级，PMD 可以评估规则是否可能相应地发生冲突并调整其优先级。

8.7. 未来演变

- 设备配置文件选择功能，可用于强制永久性配置文件，而不是依靠现有流程规则自动配置。
- 通过 PMD 即时配置生成的特定模式条目和动作类型优化 rte_flow 规则的方法。DPDK 应该为这些类型分配负数值，以便不与现有类型相冲突。具体参考负数类型。
- 添加指定 Egress 模式条目和动作，如 Traffic direction 中所描述。
- PMD 无法处理所请求的流规则时，可以选择软件实现作为后备，使得应用程序无需自己实现。

8.8. API 迁移

这一部分描述在 rte_eth_ctrl.h 中找到的已弃用的过滤器类型（通常以 RTE_ETH_FILTER_为前缀）的完整列表以及将其转换为 rte_flow 规则的方法。

8.8.1. MACVLAN to ETH□VF，RF

MACVLAN 可以转换为基本条目：ETH 流规则，终止操作为 VF 或 PF。

Table 8.40 MACVLAN conversion

Pattern				Action
0	ETH	Spec	Any	VF、PF
		Last	N/A	
		Mask	Any	
1	END			END

8.8.2. ETHERTYPE to ETH□QUEUE，DROP

ETHERTYPE 可以看成是条目：具有终止操作的 ETH 流规则，动作为 QUEUE 或 DROP。

Table 8.41 ETHERTYPE conversion

Pattern				Action
0	ETH	Spec	Any	QUEUE、DROP
		Last	N/A	
		Mask	Any	
1	END			END

8.8.3. FLEXIBLE to RAW□QUEUE

FLEXIBLE 可以转换为一个条目：RAW 模式，终止动作为 QUEUE 和定义的优先级。

Table 8.42 FLEXIBLE conversion

Pattern				Action
0	RAW	Spec	Any	QUEUE
		Last	N/A	
		Mask	Any	
1	END			END

8.8.4. SYN to TCP□QUEUE

SYN 条目：只有 syn 位启用和屏蔽的 TCP 规则，以及终止动作 QUEUE。

优先级可以设置为模拟高优先级位。

Table 8.43 SYN conversion

Pattern					Action
0	ETH	Spec	unset		QUEUE
		Last	unset		
		Mask	unset		
1	IPv4	Spec	unset		END
		Last	unset		
		Mask	unset		
2	TCP	Spec	Syn	1	
		Mask	Syn	1	
3	END				

8.8.5. NTUPLE to IPV4，TCP，UDP□QUEUE

NTUPLE 类似于指定一个空的 L2， IPV4 作为 L3， TCP 或 UDP 做为 L4，终止动作为 QUEUE。也可以指定优先级。

Table 8.44 NTUPLE conversion

Pattern				Action
0	ETH	Spec	unset	QUEUE
		Last	unset	
		Mask	unset	
1	IPv4	Spec	any	END
		Last	unset	
		Mask	any	
2	TCP/UDP	Spec	any	
		Last	unset	
		Mask	any	

3	END			

8.8.6. TUNNEL to ETH，IPV4，IPV6，VXLAN□QUEUE

TUNNEL 匹配常见的基于 IPv4 和 IPv6 L3 / L4 的隧道类型。
在下表中，ANY 条目用于覆盖可选的 L4。

Table 8.45 TUNNEL conversion

Pattern					Action
0	ETH	Spec	any		QUEUE
		Last	unset		
		Mask	unset		
1	IPv4/IPv6	Spec	any		END
		Last	unset		
		Mask	any		
2	ANY	Spec	any		
		Last	unset		
		Mask	num	0	
3	VXLAN, GENEVE, TEREDO, NVGRE, GRE, ...	Spec	any		
		Last	unset		
		Mask	any		
3	END				

8.8.7. FDIR to most item types□QUEUE，DROP，PASSTHRU

FDIR 比任何其他类型更复杂，有几种方法来模拟其功能。 大部分在下表中进行了总结。
部分功能有意不实现支持：

- 配置整个设备的匹配输入集和掩码的能力，PMD 应根据请求的流规则自动处理。
  例如，如果设备每个协议类型仅支持一个位掩码，则源/地址 IPv4 位掩码可以通过第一个创建的规则变成不可变的。 随后的 IPv4 或 TCPv4 规则只能在兼容的情况下创建。
  请注意，只有现有流规则影响的协议位掩码是不可变的，其他可以稍后更改。相关的流量规则被破坏后，它们再次变得可变。
- 使用 flex 字节过滤时返回四个或八个字节的匹配数据。虽然具体的操作可以实现它，但它与支持它的设备上更有用的 32 位标记冲突。
- 对整个设备的 RSS 处理的副作用。不允许与当前设备配置冲突的流规则。 同样，当它影响现有流规则时，不应允许设备配置。
- 设备操作模式。 “none”不受支持，因为只要存在流规则，就不能禁用过滤。
- 应根据创建的流规则自动设置“MAC VLAN”或“隧道”完美匹配模式。
- 签名模式的操作未定义，但如果需要，可以通过特定的项目类型处理。

Table 8.46 FDIR conversion

Pattern				Action
0	ETH/RAW	Spec	any	QUEUE 、 DROP、PASSTHRU
		Last	N/A	
		Mask	any	
1	IPv4/IPv6	Spec	any	MARK
		Last	N/A	
		Mask	any	
2	TCP、UDP、SCTP	Spec	any	END
		Last	N/A	
		Mask	any	
3	VF、RF	Spec	any	
		Last	N/A	
		Mask	any	
3	END			

8.8.8. HASH

没有这个过滤器类型的对应物，因为它转换为全局设备设置而不是模式项。设备设置根据创建的流规则自动设置。

8.8.9. L2_TUNNEL to VOID VXLAN

所有数据包都匹配。这种类型改变了传入的数据包，将它们封装在选定的隧道类型中，也可以将它们重定向到 VF。

可以使用动作 DUP 使用其他流规则来模拟基于标签的转发的目标池。

Table 8.47 L2_TUNNEL conversion

Pattern				Action
0	VOID	Spec	N/A	VXLAN, GENEVE, ...
		Last	N/A	
		Mask	N/A	
1	END			VF
2				END

9. 加密设备库

暂时不投入这个东西，目前来说没什么用，也不容易理解这些概念，还是先把业务理解完再回头看。

DPDK 的 cryptodev 库提供了用于管理和配置软硬件加密轮询模式驱动程序的加密设备框架，该框架定义了支持多种不同 Crypto 操作的通用 API。 框架目前只支持加密，认证，链接加密/认证和 AEAD 对称加密操作。

9.1. 设计原则

加密设备库（cryptodev）遵循与 DPDK 以太网设备框架中使用的相同基本原则。Crypto 框架提供了一个通用的 Crypto 设备框架，它支持物理（硬件）和虚拟（软件）加密设备以及通用的 Crypto API，这些 API 可以对 Crypto 设备进行管理和配置，并支持在加密轮询模式驱动程序执行 Crypto 操作。

9.2. 设备管理

9.2.1. 设备创建

9.2.2. 设备标识

9.2.3. 设备配置

9.2.4. 队列对配置

9.2.5. 逻辑核，内存和队列对的关系

9.3. 设备特性和功能

9.3.1. 设备特性

9.3.2. 设备操作能力

9.3.3. 能力发现

9.4. 操作处理

9.4.1. 入队/出队突发 API

9.4.2. 操作表示

9.4.3. 运行管理与分配

9.5. 对称密码支持

9.5.1. 会话及会话管理

9.5.2. 转换及转换链

9.5.3. 对称操作

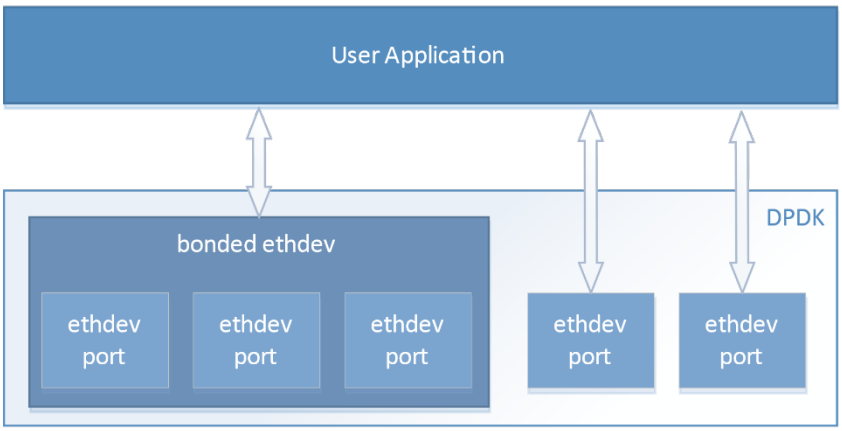
9.6. 不对称加密

9.6.1. 加密设备 API

10. 链路绑定 PMD

除了用于物理和虚拟硬件的轮询模式驱动程序（PMD）之外，DPDK 还包括一个纯软件库，可将多个物理 PMD 绑定在一起以创建单个逻辑 PMD。

Figure 10-23 Bonded PMDs



Link Bonding PMD 库（librte_pmd_bond）支持绑定相同速度和双工的 rte_eth_dev 端口组，以提供类似于 Linux 绑定驱动程序中的功能，以允许将多个（从属）NIC 聚合到服务器和交换机中的单个逻辑接口。然后，新的聚合的 PMD 将根据指定的操作模式处理这些接口，以支持冗余链路，容错和/或负载均衡等功能。

librte_pmd_bond 库导出一个 C 语言 API，包括用于创建绑定设备的 API，以及配置和管理绑定设备及其从属设备的 API。

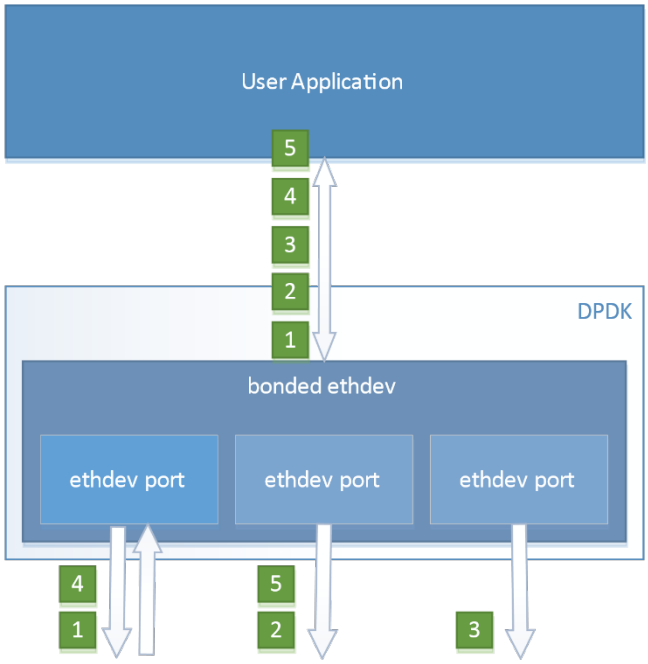
> 注意：
> 链路绑定 PMD 库默认情况下在构建配置文件中启用，可以通过设置 CONFIG_RTE_LIBRTE_PMD_BOND = n 并重新编译 DPDK 来禁用该库。

10.1.链路绑定模式概述

目前，Link Bonding PMD 库支持以下网卡绑定模式。

10.1.1. 轮询（模式 0）

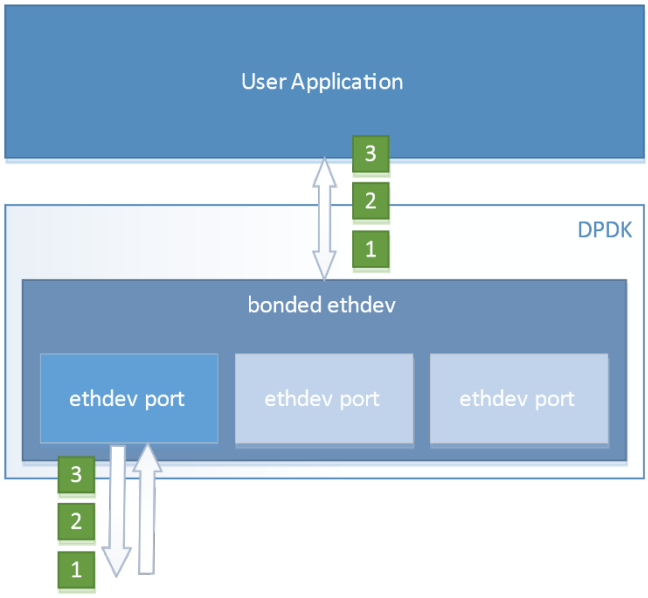
Figure 10-24 Round-Robin (Mode 0)



轮询模式通过从第一个可用从设备到最后一个的顺序来传输数据包，以提供负载平衡和容错。数据包是从设备批量出队，然后以循环方式提供服务。这种模式不能保证接收到数据包仍然有序，下行流需要能够处理乱序数据包。

10.1.2. 主动备份（模式 1）

Figure 10-25 Active Backup (Mode 1)

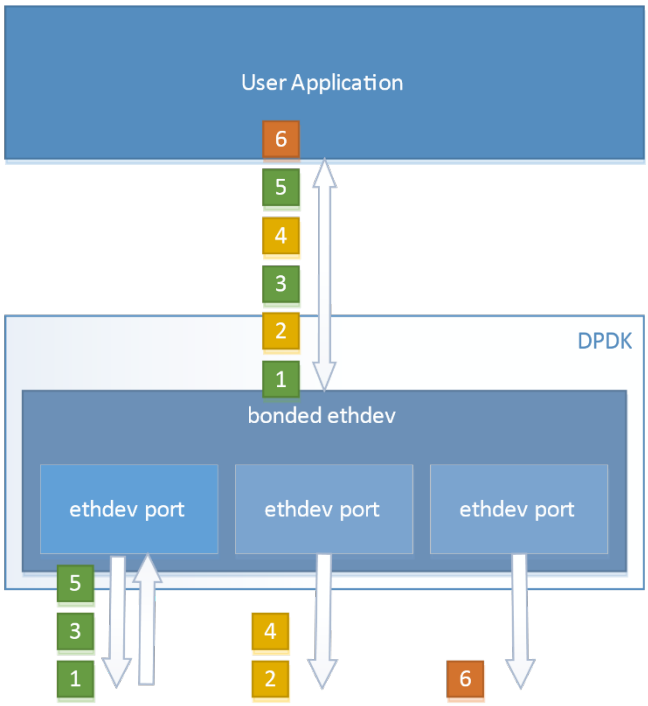


在此模式下，在任何时间只有一个从设备处于活动状态，当且仅当当前活跃从设备发生故障时，不同的从设备才会激活，从而为故障设备提供容错。单个逻辑绑定接口的 MAC 地址只能在一个

NIC（端口）上外部可见，以避免网络交换混淆。

10.1.3. 平衡策略

Figure 10-26 Balance XOR (Mode 2)

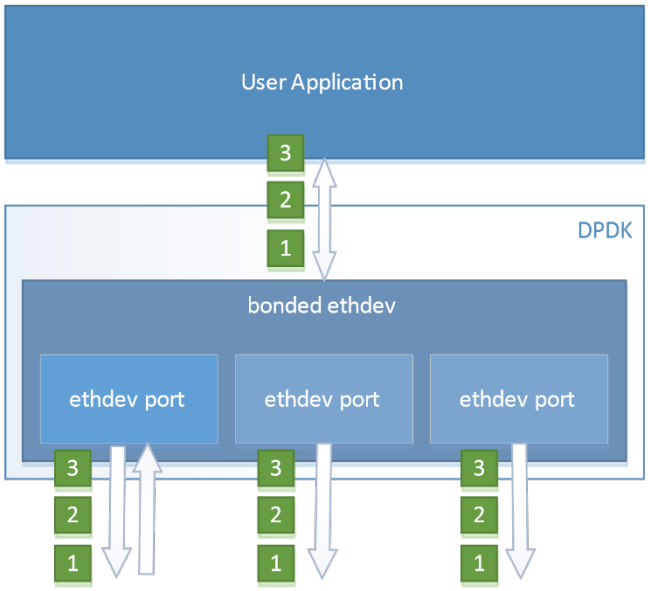


此模式提供传输负载均衡（基于所选传输策略）和容错。默认策略（layer2）使用基于报文流的源和目标 MAC 地址的简单计算以及绑定设备可用活动从设备的数量，将数据包分类到特定从设备进行传输。额外支持的备用传输策略是 L2 + L3，这将 IP 源和目标地址用于传输从端口的计算，最终需要支持的策略是 L3 + L4 层，这使用 IP 源和目标地址以及 TCP / UDP 源和目的端口进行计算。

注意： 报文的着色差异用于识别由所选择的传输策略计算的不同流分类

10.1.4. 广播策略

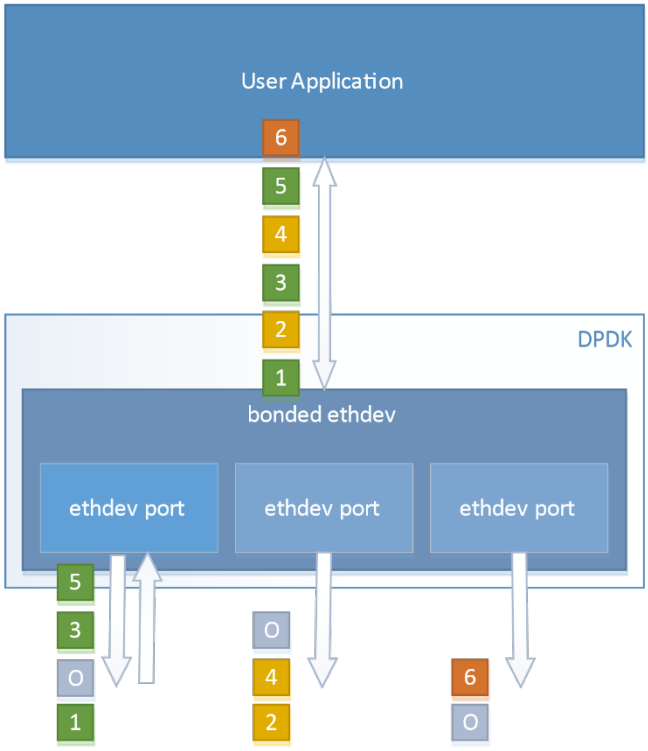
Figure 10-27 Broadcast (Mode 3)



这种模式通过在所有从设备端口上传输数据来实现容错。

10.1.5. 链路聚合 802.3AD

Figure 10-28 Link Aggregation 802.3AD (Mode 4)



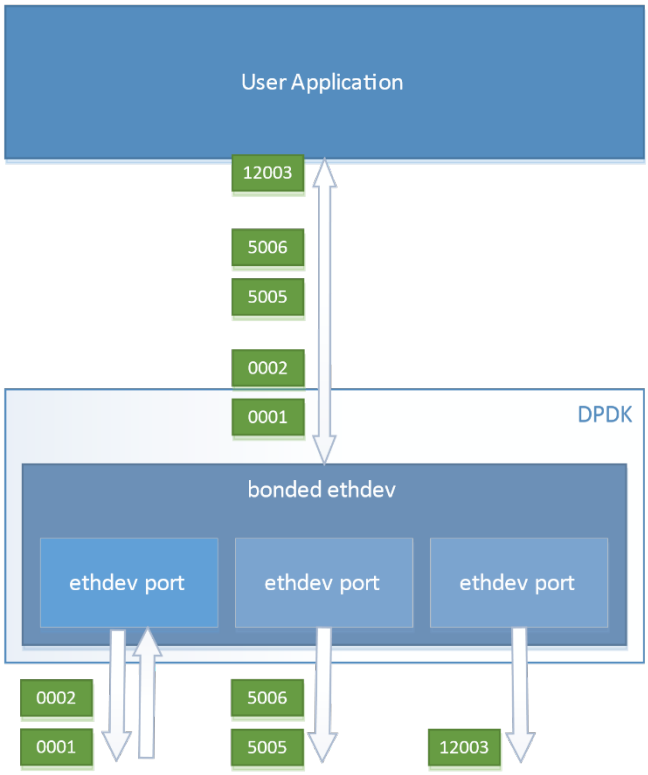
此模式根据 802.3ad 规范提供了动态链路聚合。它使用所选择的均衡传输策略来协商和监视共享相同速度和双工设置的聚合组，以平衡出口流量。

这种模式的 DPDK 实现对应用程序提供了一些额外的要求。

- 需要调用 rte_eth_tx_burst 和 rte_eth_rx_burst，间隔时间小于 100ms。
- 对 rte_eth_tx_burst 的调用必须至少具有 2xN 的缓冲区大小，其中 N 是从设备数。这是 LACP 帧所需的空间。另外 LACP 数据包也包含在统计信息中，但不会返回给应用程序。

10.1.6. 传输负载均衡策略

Figure 10-29 Transmit Load Balancing (Mode 5)



此模式提供自适应传输负载均衡。它根据计算的负载动态地更改发送从设备。以 100ms 的间隔收集统计数据，每 10ms 调度一次。

10.2.实现细节

librte_pmd_bond 绑定设备与 DPDK API 参考中描述的以太网 PMD 导出的以太网设备 API 兼容。

链路绑定库支持在 EAL 初始化期间的应用程序启动时使用 –vdev 选项以及通过 C 语言 API 接口 rte_eth_bond_create 函数以编程方式创建绑定的设备。

绑定设备支持使用接口 rte_eth_bond_slave_add / rte_eth_bond_slave_remove 实现动态添加和移除。

在将从设备添加到绑定设备后，从设备使用 rte_eth_dev_stop 停止，然后使用 rte_eth_dev_configure 进行重新配置，也可以使用 rte_eth_tx_queue_setup / rte_eth_rx_queue_setup 重新配置 RX 和 TX 队列，

并配置用于配置绑定设备的参数。如果启用绑定设备的 RSS，则此模式也将在新从站上启用并进行配置。
设置用于将设备绑定到 RSS 的多队列模式，使其完全具有 RSS 功能，因此所有从设备都与其配置同步。此模式旨在提供用于客户端应用程序实现的从站上的 RSS 配置。

绑定设备存储其自己的 RSS 设置版本，即 RETA，RSS 散列函数和 RSS 密钥，用于设置其从设备。这就是为了将绑定装置的 RSS 配置的含义定义为整个绑定（作为一个单元）的所需配置，而不指向任何从属内部。需要确保一致性并使其更具错误性。

用于绑定设备的 RSS 散列函数集，是所有绑定从站支持的 RSS 哈希函数的最大集合。RETA 大小是其所有 RETA 大小的 GCD，因此即使从属 RETA 的大小不同，它也可以轻松地用作提供预期行为的模式。如果没有为绑定设备设置 RSS 键，则在从站上不更改，并且使用设备的默认密钥。

所有设置都通过绑定端口 API 进行管理，并始终沿一个方向传播（从绑定到从站）。

10.2.1. 链路状态改变中断与轮询

链路绑定设备支持链路状态更改回调的注册，使用 rte_eth_dev_callback_register 接口，当绑定设备的状态发生更改时，将调用此函数进行处理。例如，在具有 3 个从设备的绑定设备中，当所有从设备变为不活跃时，链路状态变为 DOWN，当一个从设备变为活动状态时，链路状态将变为 UP。当单个从设备更改状态并且不满足先前的条件时，没有回调通知。如果用户希望监视单个从设备，则它们必须直接向该从设备注册回调。

链路绑定库还支持不实现链路状态改变中断处理的设备，这是通过使用接口 rte_eth_bond_link_monitoring_set 设置的周期轮询设备链路状态来实现的，默认轮询间隔为 10ms。当设备作为从设备添加到绑定设备时，使用 RTE_PCI_DRV_INTR_LSC 标志确定设备是支持中断还是通过轮询来监视链路状态。

10.2.2. 要求与限制

目前的实现只支持相同速度和双工的设备作为从设备提供给同一个绑定设备。绑定设备从添加到绑定设备的第一个活动从设备上继承这些属性，然后添加到绑定设备的所有其他从设备必须支持这些参数。

绑定设备本身启动之前，必须至少一个从设备。

为了有效地使用绑定设备动态 RSS 配置功能，还需要所有的从设备都应该是具有 RSS 能力和支持的，至少有一个通用的散列函数可用于它们。只有当所有从设备支持相同的密钥大小时才可以更改 RSS 密钥。

为了防止从设备对于如何处理数据包产生矛盾，一旦将设备添加到绑定设备，RSS 配置应通过绑定设备 API 进行管理，而不是直接在从设备上进行管理。

像所有其他 PMD 一样，PMD 导出的所有功能都是无锁功能，假定不会在不同逻辑核心上并行调用以操作同一目标对象。

还应该注意的是，PMD 接收功能在它们已经到达绑定设备之后不应该直接在从设备上被调用，因为直接从从设备读取的数据包将不再可用于绑定设备读取。

10.2.3. 配置

链路绑定设备使用 rte_eth_bond_create API 创建，该 API 需要传入唯一的设备名称，绑定模式和套接字 ID 来分配绑定设备的资源。绑定设备的其他可配置参数是其从设备，主从，用户定义的 MAC 地址，如果设备处于平衡 XOR 模式还需要定义要使用的传输策略。

10.2.3.1. 从设备

绑定设备支持相同速度和双工的设备，最大数目为 RTE_MAX_ETHPORTS。每个以太网设备可以作为从设备添加到最多一个绑定设备上。从设备在被加入绑定设备时被重新配置为绑定设备的配置。

绑定还保证将从设备的 MAC 地址返回到其原始值。

10.2.3.2. 主从

主从关系用于定义绑定设备处于主动备份模式（模式 1）时使用的默认端口。当且仅当当前主端口关闭时，才会使用不同的端口。如果用户没有指定主端口，则默认为添加到绑定设备的第一个端口。

10.2.3.3. MAC 地址

绑定设备可以配置用户指定的 MAC 地址，该地址将由某些或所有从设备根据操作模式继承。如果设备处于主动备份模式，则只有主设备具有用户指定的 MAC，所有其他从设备将保留其原始 MAC 地址。在模式 0,2,3,4 中，所有从站设备都配置了绑定设备的 MAC 地址。

如果未定义用户定义的 MAC 地址，则绑定设备将默认使用主从站 MAC 地址。

10.2.3.4. 均衡 XOR 模式传输策略

对于在均衡 XOR 模式下运行的绑定设备，有 3 种支持的传输策略。层 2，层 2 + 3，层 3 + 4。

- Layer 2：默认的传输策略是以太网基于 MAC 地址的均衡策略。它对包的源 MAC 地址和目的 MAC 地址使用简单的 XOR 计算，然后计算该值的模数，以计算需要输出数据包的从设备。
- Layer2+3：以太网 MAC 地址和基于 IP 地址的均衡策略使用源/目的 MAC 地址和数据包的源/目的 IP 地址组合来决定数据包将被传输的从设备端口。
- Layer3+4：IP 地址和 UDP 基于端口的均衡策略使用源/目的 IP 地址和数据包的数据包的源/目的 UDP 端口的组合来决定数据包将被传输的从设备端口。

所有这些策略都支持 802.1Q VLAN 以太网报文，还支持 IPv4，IPv6 和 UDP 协议进行负载分担。

10.3.使用链路绑定设备

librte_pmd_bond 库支持两种设备创建模式，库导出完整的 C API 或使用 EAL 命令行在应用程序启动时静态配置链路绑定设备。使用 EAL 选项，可以透明地使用链接绑定功能，而不需要库 API 的具体知识，例如，可以使用这种功能来将绑定功能（如主动备份）添加到不了解链接的现有应用程序上。

10.3.1. 程序中使用轮询模式驱动

使用 librte_pmd_bond 库 API，可以在任何应用程序内动态创建和管理链路绑定设备。链路绑定设备使用 rte_eth_bond_create API 创建，该 API 需要唯一的设备名称，用于初始化设备的链路绑定模式，以及最后将要分配设备资源的套接字 ID。在成功创建绑定设备之后，必须使用通用的以太网设备配置 API rte_eth_dev_configure 来配置，然后使用 rte_eth_tx_queue_setup/rte_eth_rx_queue_setup 将要使用的 RX 和 TX 队列进行设置。

可以使用 rte_eth_bond_slave_add/rte_eth_bond_slave_remove API 对链路绑定设备动态添加和删除从设备，但在使用 rte_eth_dev_start 启动链路绑定设备之前，必须至少添加一个从设备。

绑定设备的链路状态由其从设备的链路状态决定，如果所有从设备链路状态都关闭，或者所有从设备都从链路绑定设备中删除，则绑定设备的链路状态为 DOWN。

还可以使用提供的 rte_eth_bond_mode_set/get，rte_eth_bond_primary_set/get，rte_eth_bond_mac_set/reset 和 rte_eth_bond_xmit_policy_set/get 来配置/查询绑定设备的控制参数的配置。

10.3.2. 在 EAL 命令行中使用链路绑定设备

链路绑定设备可以在应用程序启动时使用 --vdev EAL 命令行选项创建。 设备名称必须以 net_bonding 前缀开头，后跟数字或字母。每个设备的名称必须是唯一的。每个设备可以有多个选项，以逗号分隔列表排列。可以多次调用--vdev 选项来安排多个设备定义。

设备名称和绑定选项必须用逗号分隔，如下所示：

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 --vdev 'net_bonding0,bond_opt0=..,bond opt1=..'--vdev 'net_bonding1,bond _opt0=..,bond_opt1=..'
```

10.3.2.1. 链路绑定 EAL 选项

只要遵守以下两个规则，可以对多种定义方式组合使用：

- 提供了一种独特的设备名称，格式为 net_bondingX，其中 X 可以是数字和/或字母的任意组合，名称不大于 32 个字符。
- 为每个绑定设备定义提供至少一个从设备。
- 提供了所创建的绑定设备的操作模式。

不同的选项包括：

- 模式：定义设备的绑定模式的整数值。目前支持模式 0,1,2,3,4,5（循环，主动备份，平衡，广播，链路聚合，传输负载均衡）。
- 从设备：定义将作为从设备添加到绑定设备的 PMD 设备。可以多次选择此选项，每个设备要作为从设备添加。物理设备应使用其 PCI 地址指定，格式为 domain:bus:devid.function。
- 主设备：定义主从端口的可选参数用于主动备份模式，以便在数据 TX / RX 可用时选择主从机。 当主端口未被用户定义时，主端口也用于选择要使用的 MAC 地址。如果未指定该设备，则默认为添加到设备的第一个从设备。主设备必须是绑定设备的从设备。
- Socket_id：可选参数，用于选择 NUMA 设备上将分配绑定设备资源的哪个套接字。
- Mac：可选参数，选择链路绑定设备的 MAC 地址，这将覆盖主设备的值。
- xmit_policy：绑定设备处于均衡模式时定义传输策略的可选参数。如果没有用户指定，则默认为 l2（第 2 层）转发，其他可用的传输策略为 l23（第 2 层+3 层）和 l34 层（3 + 4 层）。
- lsc_poll_period_ms：可选参数，用于定义不支持 lsc 中断的设备以毫秒为单位的轮询间隔，检查设备链路状态的变化。
- ups delay：可选参数，增加了设备链路状态传播的延迟（以毫秒为单位），默认情况下该参数为零。
- down_delay：可选参数，以毫秒为单位，将设备链路状态 DOWN 的传播延迟，默认情况下，该参数为零。

10.3.2.2. 使用实例

以轮询模式创建一个绑定设备，两个从设备由其 PCI 地址指定：

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 --vdev 'net_bonding0,mode=0,
slave=0000:00a:00.01,slave=0000:004:00.00' -- --port-topology=chained
```

以轮询模式创建一个绑定设备，其中两个从站由其 PCI 地址和覆盖 MAC 地址指定：

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 --vdev 'net_bonding0,mode=0,
slave=0000:00a:00.01,slave=0000:004:00.00,mac=00:1e:67:1d:fd:1d' -- --port-topology=chained
```

在平衡模式下创建一个绑定设备，其中两个从站由其 PCI 地址指定，第 3 + 4 层传输策略：

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 --vdev 'net_bonding0,mode=2,
slave=0000:00a:00.01,slave=0000:004:00.00,xmit_policy=l34' -- --port-topology=chained
```

11. 定时器库

定时器库为 DPDK 执行单元提供定时器服务，使得执行单元可以为异步操作执行回调函数。定时器库的特性如下：

- 定时器可以周期执行，也可以执行一次。
- 定时器可以在一个核心加载并在另一个核心执行。但是必须在调用 rte_timer_reset()中指定它。
- 定时器提供高精度（取决于检查本地核心的定时器到期的 rte_timer_manage()的调用频率）。
- 如果应用程序不需要，可以在编译时禁用定时器，并且程序中不调用 rte_timer_manage()来提高性能。

定时器库使用 rte_get_timer_cycles()获取高精度事件定时器（HPET）或 CPU 时间戳计数器（TSC）提供的可靠时间参考。

该库提供了添加，删除和重新启动定时器的接口。API 基于 BSD callout()，可能会有一些差异。详细请参考 callout 手册。

11.1. 实现细节

定时器以每个逻辑核为基础进行跟踪，一个逻辑核上要维护的所有挂起的定时器，按照定时器到期顺序插入到跳跃表数据结构。

所使用的跳跃表有十个层，表中的每个条目都以 1/4 的概率显示在每个层上。这意味着所有条目都存在于第 0 层中，每 4 个条目中的 1 个条目存在于第一层，每 16 个中 1 个条目存在于第 2 层，等等。同时，这意味着从逻辑核的定时器列表中添加和删除条目可以在 log(n)时间内完成，最多 4 ^ 10 个条目，即每个逻辑核约有 1,000,000 个定时器。

定时器结构包含一个称为状态的特殊字段，它是定时器状态（stopped，pending，running，config）及其所有者（lcore id）的联合体。根据定时器状态，我们可以知道定时器当前是否存在于列表中：

- STOPPED：没有所有者，不再链表中。
- CONFIG：由一个逻辑核持有，其他逻辑核不能修改，是否存在于跳表中取决于以前的状态。
- PENDING：由一个逻辑核持有，当前在跳表中。
- RUNNING：由一个逻辑核持有，当前在跳表中，不能由其他逻辑核修改。

不允许在定时器处于 CONFIG 或 RUNNING 状态时复位或停止定时器。当修改定时器的状态时，应使用 CAP 指令来保证状态修改操作（状态+所有者）是原子操作。

在 rte_timer_manage()函数里面，跳跃表作为常规的链表，通过沿着包含所有计时器条目的第 0 层链表迭代，直到遇到尚未到期的条目为止。当列表中有条目，但是没有任何条目定时器到期时，为了提高性能，第一个定时器条目的到期时间保存在每个逻辑和计时器列表结构本身内部。在 64 位平台上，可以直接检查该值，而无需对整个结构进行锁定。（由于到期时间维持为 64 位值，所以在 32 位平台上无法直接对该值进行检查，而不使用（CAS）指令或使用锁机制，因此，一旦数据结构被上锁，此额外的检查将被跳过。）在 64 位和 32 位平台上，在调用逻辑核的计时器列表为空的情

况下，对 rte_timer_manage（）的调用将直接返回而不进行锁定。=

11.2. 用例

定时器库用于定期调用，如垃圾收集器或某些状态机（ARP，桥接等）。

11.3. 参考

- callout manual：唤醒功能，提供定时器到期执行的功能。
- HPET：有关高精度事件定时器（HPET）的信息。

12. 哈希库

DPDK 提供了一个用于创建哈希表的哈希库，哈希表可以用于快速查找。哈希表是针对一组条目进行搜索而优化的数据结构，每个条目由唯一 Key 标识。为了提高性能，DPDK 哈希要求所有的 Key 值具有与哈希创建时指定的相同字节数。

12.1.哈希 API 概述

哈希的主要配置参数包括：

- 哈希条目总数（哈希容量）。
- Key 的字节数。

哈希还允许配置一些低级实现相关的参数：

- 将 Key 转换为哈希桶索引值的哈希函数

哈希库导出的主要方法包括：

- 使用 Key 值添加条目：Key 值作为输入参数。如果新条目成功添加到指定 Key 的哈希中，或者已经有指定 Key 的条目，则返回该条目的位置。如果操作不成功，例如由于在哈希中缺少空闲条目，则返回负值;
- 使用 Key 删除条目：Key 值作为输入参数。如果在哈希中找到具有指定 Key 的条目，则会从哈希中删除该条目，并返回该条目在哈希中找到的位置。如果哈希中没有指定 Key 的条目存在，则返回一个负值；
- 使用 Key 查找条目：Key 值作为输入参数。如果在哈希（查找命中）中找到具有指定 Key 的条目，则返回条目的位置，否则返回（查询未命中）一个负值。

除了这些方法，API 还为用户提供了三个选项：

- 使用 Key 和 precomputed hash 来查找/添加/删除条目：Key 及其 precomputed hash 都作为输入。这允许用户更快地执行操作，因为已经预先计算了散列。
- 使用 Key 和数据来查找/添加条目：提供 Key-value 作为输入。这允许用户不仅存储 Key，还可以存储 8byte 的整形或是一个指向外部数据的指针（数据超过 8byte）。
- 上述两个选项的组合：用户可以提供 Key、precomputed hash 或是 data。

此外，API 包含一种方法，允许用户在突发中查找条目，实现比查找单个条目更高的性能，因为该函数在与第一个条目操作时预取下一个条目，显著降低了必要的内存访问。请注意，此方法使用 8 个条目（4 个阶段 2 条目）的流水线，因此强烈建议每个突发使用至少 8 个条目。

与每个 Key 相关联的实际数据可以由用户使用单独的表格进行管理，该表格根据哈希条目数量和每个条目的位置来反映哈希表，如以下部分中描述的流分类用例所示，当然，也可以直接存储在哈希表本身。

L2 / L3 转发示例应用程序中的哈希表根据由五元组查找标识的数据包流定义将数据包转发到哪个端口。然而，该表还可以用于更复杂的特征，并提供可以在分组和流上执行的许多其他功能和动作。

12.2.多进程支持

哈希库可以在多进程环境中使用，只需查找线程安全即可。只能在单进程模式下使用的唯一函数是rte_hash_set_cmp_func()，它设置一个自定义的比较功能，分配给一个函数指针（因此在多进程模式下不支持）。

12.3.实现细节

哈希表有两个主表：

- 第一个表是一组条目，进一步分为桶，每个桶中具有相同数量的连续数组条目。每个条目包含计算的给定 Key 的主要和次要散列（如下所述）和第二个表的索引。
- 第二个表是存储在哈希表中的所有 Key 的数组及其与每个 Key 相关联的数据。

哈希库使用 Cuckoo hash（布谷鸟散列）方法来解决冲突。对于任何输入 Key，有两个可能的桶（主要和次要/替代位置），其中该 Key 可以存储在散列中，因此只有当查询 Key 时才需要检查桶中的条目。与通过线性扫描阵列中的所有条目的基本方法相反，通过将散列条目的总数减少到两个哈希桶中的条目数来减少要扫描的条目数以提升查找速度。哈希使用散列函数（可配置）将输入 Key 转换为 4 字节 Key 签名。桶索引值是将哈希 Key 签名对哈希桶数取模数的值。

一旦识别出桶，哈希添加，删除和查找操作的范围就减少到这些存储区中的条目（很可能条目在主存储桶中）。

为了加快桶内的搜索逻辑，每个散列条目将 4 字节 Key 签名与每个哈希条目的完整 Key 一起存储。对于大的 Key，将输入 Key 与来自存储桶的 Key 进行比较比将输入 Key 的 4 字节签名与来自存储桶的 Key 签名进行比较要花费更多的时间。因此，首先完成签名比较，仅在签名匹配时才完成 Key 比较。完全 Key 比较仍然是必要的，因为来自相同存储桶的两个输入 Key 仍然可能具有相同的 4 字节签名，尽管对于该组输入密钥提供良好的均匀分布的散列函数，该事件相对较少。

查找实例：
首先，主桶被识别，条目可能存储在那里。如果签名存储在那里，我们将其 Key 与提供的 Key 进行比较，并返回其存储位置和/或与该密钥相关联的数据（如果有匹配）。如果签名不在主桶中，则查找辅助桶，在那里执行相同的过程。如果没有匹配，Key 对应条目被认为不在表中。

添加实例：
像查找操作一样，Key 标识主和二级桶。如果主桶中有一个空槽，则主签名和辅助签名存储在该槽中，Key 和数据（如果有的话）被添加到第二个表中，并且第二个表中的位置的索引被存储在第一张表上。如果主桶中没有空槽，则该桶中的一个条目将被推送到其替代位置，并将要添加的 Key 插入第一个条目的位置上。要知道驱逐条目（第一个条目）的替代桶的哪个位置，则查找器辅助签名，并从上面的模数中计算备用桶索引。如果替代桶中有空间，则将被驱入的条目存储在其中。如果没有，则重复相同的过程（其中一个条目被推送），直到找到非完整的数据桶。请注意，尽管所有的条目移动都在第一张表中，第二张表没有被触动，这也将在性能上受到很大影响。

在非常不太可能的事件中，该表进入循环，其中相同的条目被无限期地驱逐，则认为 Key 不能被存储。使用随机 Key，该方法允许用户获取约 90%的表利用率，而不必放弃任何存储的条目（LRU）或分配更多内存（扩展桶）。

12.4.哈希表中的条目分发

如上所述，如果有一个新的条目要被添加到哪个主桶，而当前已经有数据在里面时，则将数据推送到他们的替代位置，Cuckoo 哈希实现了将元素推出他们的存储区。

因此，当用户向哈希表添加更多条目时，桶中散列值的分布将发生变化，其中大部分位于主要位置，并且其次要位置会随之增加，随后表将增加。

这些信息是非常有用的，因为随着更多条目逐出其次要位置，性能可能会降低。

下表显示了表利用率增加时的示例条目分布。

Table 12.48 Entry distribution measured with an example table with 1024 random entries using jhash algorithm

%Table used	%In Primary location	%In Secondary location
25	100	0
50	96.1	3.9
75	88.2	11.8
80	86.3	13.7
85	83.1	16.9
90	77.3	22.7
95.8	64.5	35.5

Table 12.49 Entry distribution measured with an example table with 1 million random entries using jhash algorithm

%Table used	%In Primary location	%In Secondary location
50	96	4
75	86.9	13.1
80	83.9	16.1
85	80.1	19.9
90	74.8	25.2
94.5	67.4	32.6

注意：
上表上的最后值是具有随机密钥和使用 Jenkins 散列函数的平均最大表利用率。

12.5.用例：流分类

流分类用于将每个输入数据包映射到它所属的连接/流。这种操作是必需的，因为每个输入分组的处理通常在其连接的上下文中进行，因此相同的操作集合被应用于来自相同流的所有分组。

使用流分类的应用通常具有要管理的流表，每个单独的流具有与该表相关联的条目。流表条目的大小是特定于应用程序的，典型值为 4,16,32 或 64 字节。

使用流分类的每个应用通常具有被定义为从输入报文中读取一个或多个字段来构成 Key，用于标识流。一个例子是使用由 IP 和传输层数据包头的以下字段组成的 DiffServ 5 元组：源 IP 地址，目标 IP 地址，协议，源端口，目标端口。

DPDK 哈希提供了一种通用的方法来实现应用程序指定的流分类机制。 给定一个用数组实现的流表，应用程序应该创建与流表相同数量的条目的哈希对象，并将哈希密钥大小设置为所选流 Key 中的字节数。

应用侧的流程表操作如下：

- 添加流：将流 Key 添加到哈希。如果返回的位置有效，则使用它来访问流表中用于添加新流或更新与现有流相关联的信息的流条目。否则，流添加失败，例如由于缺少用于存储新流的空闲条目。
- 删除流：从哈希中删除流 Key。如果返回的位置有效，则使用它来访问流表中的流条目以使与流相关联的信息无效。
- 查找流：在哈希中查找流 Key。如果返回的位置有效（流查找命中），则使用返回的位置来访问流表中的流条目。否则（流查找未命中）表示当前数据包没有注册流。

12.6.参考

Donald E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition), 1998, Addison-Wesley Professional

13. 弹性流分配器库

13.1.简介

如今，在数据中心中，分布式工作负载的集群和调度是非常常见的任务。许多工作负载需要在一组机器之间确定性划分平面空间。当数据包进入集群时，Ingress 节点将数据包引导到其处理节点上。例如，具有分解存储的数据中心使用存储 metadata 表将 I / O 请求转发到正确的后端存储集群，状态数据包检查将使用流入表中的匹配传入流到签名，以将传入的数据包发送到其预期的深度数据包检查（DPI）设备等。

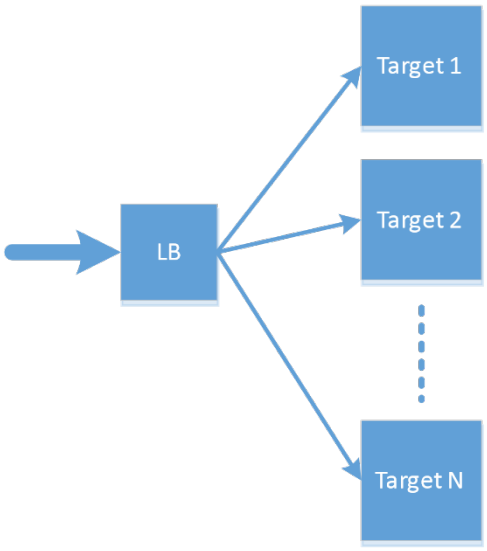
EFD 是使用完美散列来为给定传入流 Key 决定目标/值的分发器库。它具有以下优点：首先，因为它使用完美的散列，它不存储 Key 本身，因此查找性能不依赖于 Key 大小。第二，目标/值可以是任意的任意值，因此系统设计者和/或操作者可以更好地优化服务速率和群集间网络流量定位。第三，由于存储要求远小于基于散列的流表（即更适合于 CPU 缓存），所以 EFD 可以扩展到数百万个流 Key。最后，利用当前优化的库实现，性能根据任意数量的 CPU Core 线性扩展。

13.2.基于流的分发

13.2.1. 基于计算的方案

流量分配或负载均衡可以简单地使用无计算完成，如使用轮询或基于流 Key 作为输入的简单计算。举个例子，可以使用哈希函数来基于流 Key（h(key) mod n）将特定的流引到目的地上，其中 h(key) 是流 Key 的哈希值，n 是所有可能的目的地数目。

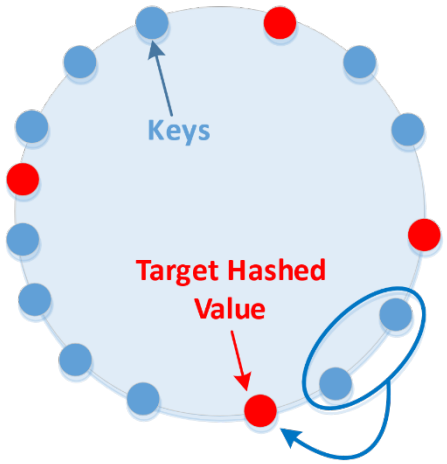
Figure 13-30 Load Balancing Using Front End Node



在该方案（上图）中，前端服务器/分发器/负载均衡器从输入报文中提取流 Key，并应用计算来确

定该流应该在哪里引导。直观地说，这个方案非常简单，不需要在前端节点保持状态，因此对存储的要求是最小的。

Figure 13-31 Consistent Hashing



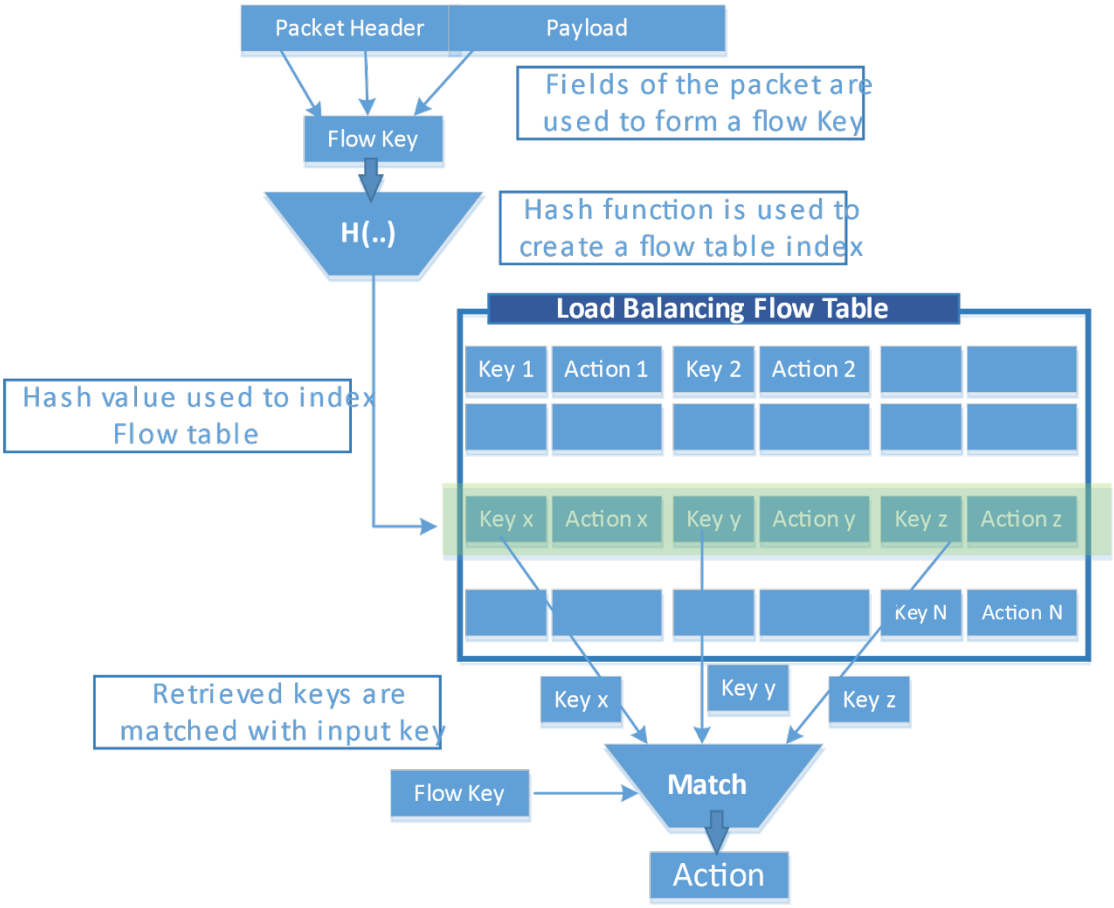
广泛使用的流量分配器属于同一类基于计算的方案是一致性哈希，如上图所示。目标（以红色显示）与流量 Key（蓝色显示）相同的空间散列，按顺时针方式将 Key 映射到最近的目标。动态添加和删除具有一致散列的目标只需要平均重新映射 K / n 个 Key，其中 K 是 Key 的数量，n 是目标的数量。相比之下，在传统的基于散列的方案中，目标数量的变化导致几乎所有的 Key 都被重新映射。

虽然基于计算的方案是简单的，并且需要非常少的存储要求，但是它们具有缺点，即系统设计者/操作者不能完全控制目标来分配特定的 Key，这是由散列函数规定的。确定性地将 Key 共同定位在一起（例如，最小化服务器间流量或优化网络流量状况，目标负载等）是不可能的。

13.2.2. 基于流表的方案

与基于计算的方案相反，当使用基于流表的方案来处理流量分配/负载均衡时，系统设计人员可以灵活地将给定的流量分配给任何给定的目标。流表（例如 DPDK RTE 哈希库）将简单地存储流 Key 和目标值。

Figure 13-32 Table Based Flow Distribution



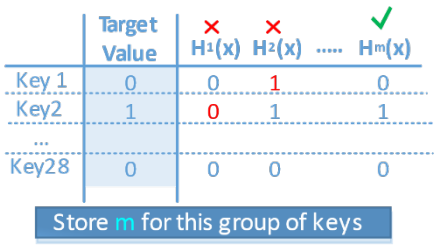
如图所示，当进行查找时，流表被索引，流 Key 的哈希和存储在该索引中的 Key（因为哈希冲突，会有多个可能），并检索相应的值。检索的 Key 与输入流 Key 匹配，如果匹配，则返回值（目标 ID）。

使用哈希表进行流量分配/负载均衡的缺点是存储需求，因为流表需要存储 Key，签名和目标值。这不允许此方案扩展到数百万个流 Key。大型表通常不适用于 CPU 缓存，因此，由于访问主内存的延迟，查找性能下降。

13.2.3. 基于 EFD 的方案

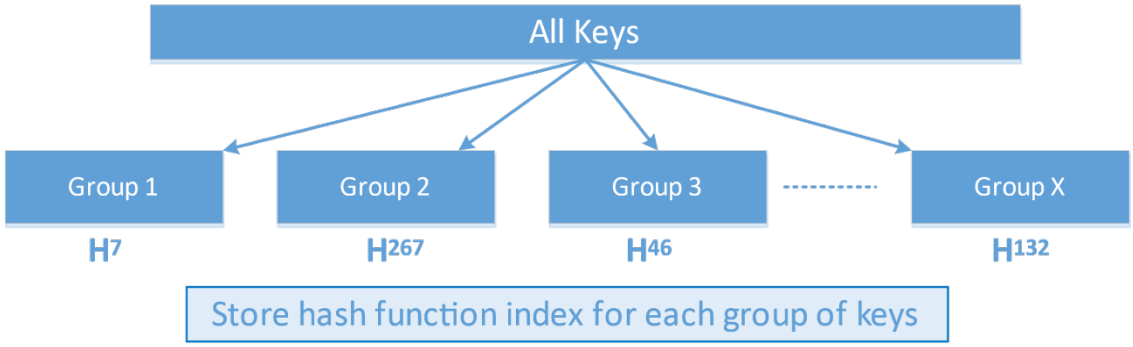
EFD 结合了基于流表和基于计算两种方案的优势。它不需要基于流表的方案所需的大量存储（因为 EFD 不存储如下所述的 Key），并且它支持任何给定 Key 的任意值。

Figure 13-33 Searching for Perfect Hash Function



EFD 的基本思想是当插入给定的 Key 时，搜索一系列哈希函数，直到找到将输入 Key 映射到正确值的正确哈希函数为止，如上图所示。然而，EFD 并不是显式地存储所有 Key 及其关联的值，而只存储将 Key 映射到值的散列函数的索引，从而比传统的基于流的表消耗更少的空间。查找操作非常简单，类似于基于计算的方案：给定输入 Key，查找操作被减少为具有正确哈希函数的哈希散列。

Figure 13-34 Divide and Conquer for Millions of Keys



直观地，找到将大量（百万）输入 Key 的每一个映射到正确的输出值的散列函数是不可能的，EFD，如上图所示，将问题分解成小块（划分和征服）。EFD 将整个输入 Key 集合分成许多小组。每个组由约 20-28 个 Key（库的可配置参数）组成，然后，对于每个小组，进行强力搜索以找到为组中的每个 Key 产生正确输出的散列函数。

应该提到的是，由于 EFD 的在线查找表不存储 Key 本身，因此 EFD 表的大小与 Key 大小无关，因此 EFD 查找性能几乎不变，与 Key 的长度无关是一个非常需要的功能，特别是对于较长的 Key。

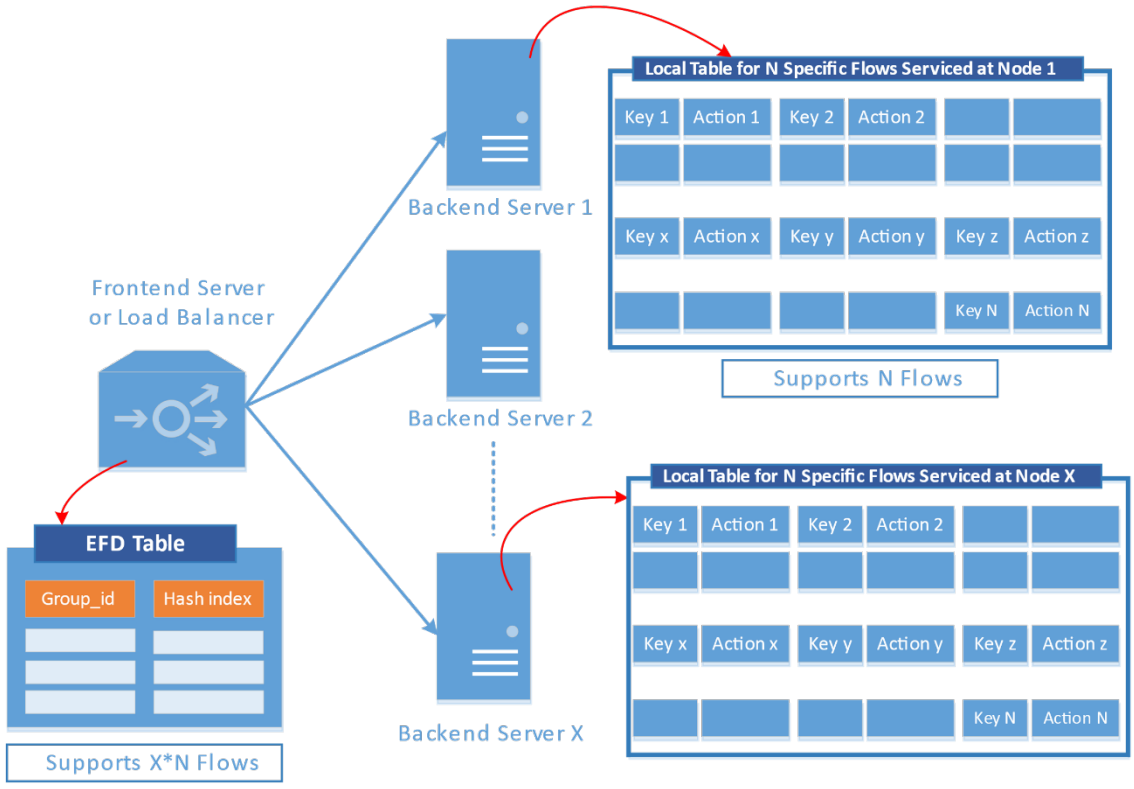
总而言之，EFD 是一种设置的分离数据结构，支持数百万个 Key。它用于将给定的 Key 分发到预期的目标。本身 EFD 不是与输入流 Key 完全匹配的 FIB 数据结构。

13.3.EFD 库使用实例

EFD 可以被许多网络功能和中间框的数据路径使用。如前所述，它可以用作<key，value>对的索引表，对象的元数据，流负载均衡器等。下图展示出了使用 EFD 作为流量负载均衡器的示例，其中在前端服务器处接收流量，然后将其转发到目标后端服务器进行处理。系统设计人员将确定性地将

流程共同定位，以最小化跨服务器交互。（例如，请求某些网页对象的流程共同位于一起，以最小化跨服务器的公共对象的转发）。

Figure 13-35 EFD as a Flow-Level Load Balancer



如图所示，前端服务器将有一个 EFD 表，为每个组存储满足正确输出的完美散列索引。因为表的大小很小并且适合高速缓存（因为没有存储密钥），所以它维持大量的流（N * X，其中 N 是 X 个可能目标的每个后端服务器所服务的最大流量数）。

使用输入流 Key，计算组 ID（例如，使用 CRC 哈希的最后几位），然后将 EFD 表与组 ID 进行索引，以检索相应的使用的散列索引。一旦检索到索引，则使用该哈希函数对 Key 进行散列，并且结果将是该流被应用于处理的预期正确目标。

应该注意的是，由于 EFD 不匹配精确的 Key，而是基于完美的散列索引将流分发到目标后端节点，以前没有被插入的 Key 将被分发到有效的目标。因此，使用存储在每个节点处服务的流的本地表，并且与输入 Key 精确匹配以排除在流之前从未见过的新的。

13.4.库 API 概述

13.4.1. EFD 表创建

13.4.2. EFD 插入和更新

13.4.3. EFD 查询

13.4.4. EFD 删除

13.5.库内部实现

13.5.1. 插入功能内部实现

13.5.2. 查询功能内部实现

13.5.3. 组自平衡功能实现

13.6.参考

14. LPM 库

DPDK LPM 库组件实现了 32 位 Key 的最长前缀匹配（LPM）表搜索方法，该方法通常用于在 IP 转发应用程序中找到最佳路由。

14.1.LPM API 概述

LPM 组件实例的主要配置参数是要支持的最大数量的规则。LPM 前缀由一对参数（32 位 Key，深度）表示，深度范围为 1 到 32。LPM 规则由 LPM 前缀和与前缀相关联的一些用户数据表示。该前缀作为 LPM 规则的唯一标识符。在该实现中，用户数据为 1 字节长，被称为下一跳，与其在路由表条目中存储下一跳的 ID 的主要用途相关。

LPM 组件导出的主要方法有：

- 添加 LPM 规则：LPM 规则作为输入参数。如果表中没有存在相同前缀的规则，则将新规则添加到 LPM 表中。如果表中已经存在具有相同前缀的规则，则会更新规则的下一跳。当没有可用的规则空间时，返回错误。
- 删除 LPM 规则：LPM 规则的前缀作为输入参数。如果具有指定前缀的规则存在于 LPM 表中，则会被删除。
- LPM 规则查找：32 位 Key 作为输入参数。该算法用于选择给定 Key 的最佳匹配的 LPM 规则，并返回该规则的下一跳。在 LPM 表中具有多个相同 32 位 Key 的规则的情况下，算法将最高深度的规则选为最佳匹配规则（最长前缀匹配），这意味着该规则 Key 和输入的 Key 之间具有最高有效位的匹配。

14.2.实现细节

目前的实现使用 DIR-24-8 算法的变体，可以改善内存使用量，以提高 LPM 查找速度。该算法允许以典型的单个存储器读访问来执行查找操作。在统计上看，即便是不常出现的情况，当即最佳匹配规则的深度大于 24 时，查找操作也仅需要两次内存读取访问。因此，特定存储器位置是否存在于处理器高速缓存中将很大程度上影响 LPM 查找操作的性能。

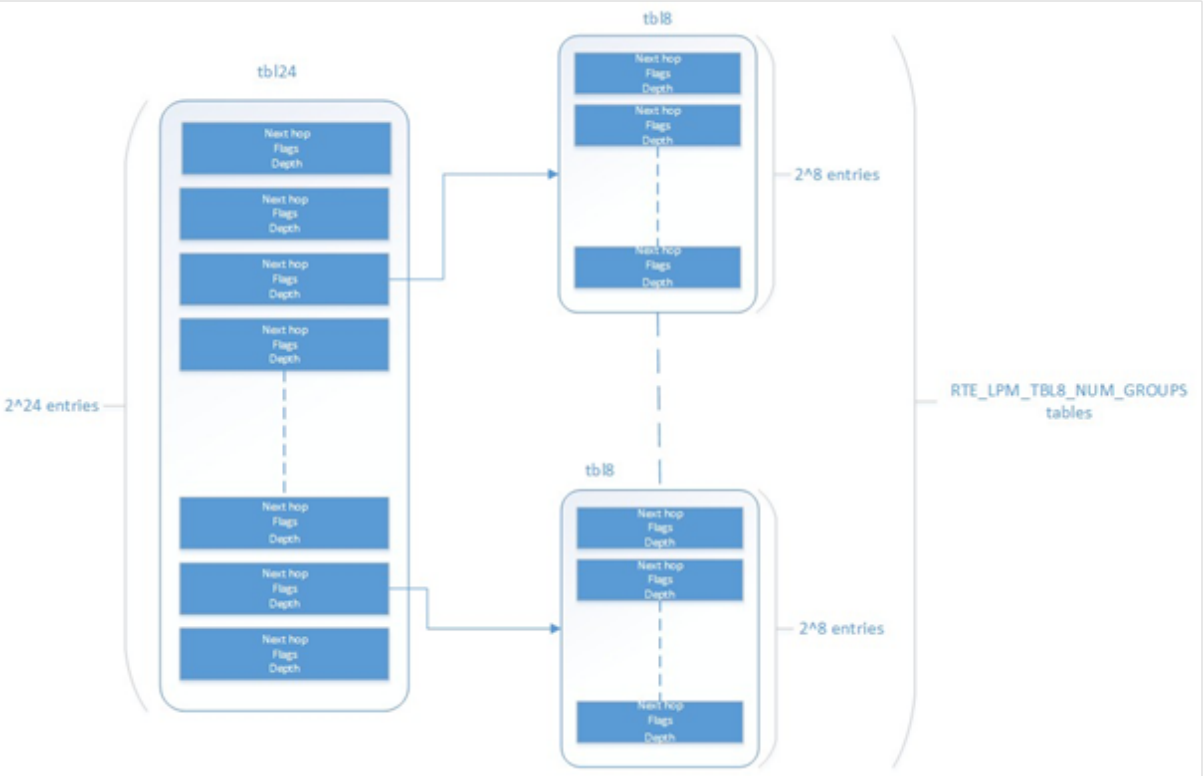
主要数据结构使用以下元素构建：

- 一个 2^24 个条目的表。
- 多个表（RTE_LPM_TBL8_NUM_GROUPS），每个表有 2 ^ 8 个条目。

第一个表，称为 tbl24，使用要查找的 IP 地址的前 24 位进行索引；而第二个表，称为 tbl8 使用 IP 地址的最后 8 位进行索引。这意味着根据输入数据包的 IP 地址与存储在 tbl24 中的规则进行匹配的结果，我们可能需要在第二级继续查找过程。

由于 tbl24 的每个条目都可以指向 tbl8，理想情况下，我们将具有 2 ^ 24 tbl8，这与具有 2 ^ 32 个条目的单个表占用空间相同。因为资源限制，这显然是不可行的。相反，这种组织方法就是利用了超过 24 位的规则是非常罕见的这一特定。通过将这个过程分为两个不同的表/级别并限制 tbl8 的数量，

我们可以大大降低内存消耗，同时保持非常好的查找速度（大部分时间仅一个内存访问）。
Figure 14-36 Table split into different levels



tbl24 中的条目包含以下字段：

- 下一跳，或者下一级查找表 tbl8 的索引值。
- 有效标志。
- 外部条目标志。
- 规则深度。

第一个字段可以包含指示查找过程应该继续的 tbl8 的数字，或者如果已经找到最长的前缀匹配，则可以包含下一跳本身。两个标志字段用于确定条目是否有效，以及搜索过程是否分别完成。规则的深度或长度是存储在特定条目中的规则的位数。

tbl8 中的条目包含以下字段：

- 下一跳。
- 有效标志。
- 有效组。
- 深度。

下一跳和深度包含与 tbl24 中相同的信息。两个标志字段显示条目和表分别是否有效。

其他主要数据结构是包含有关规则（IP 和下一跳）的主要信息的表。这是一个更高级别的表，用于不同的东西：

- 在添加或删除之前，检查规则是否已经存在，而无需实际执行查找。
- 删除时，检查是否存在包含要删除的规则。这很重要，因为主数据结构必须相应更新。

14.2.1. 添加

添加规则时，存在不同的可能性。如果规则的深度恰好是 24 位，那么：

- 使用规则（IP 地址）作为 tbl24 的索引。
- 如果条目无效（即它不包含规则），则将其下一跳设置为其值，将有效标志设置为 1（表示此条目正在使用中），并将外部条目标志设置为 0（表示查找 此过程结束，因为这是匹配的最长的前缀）。

如果规则的深度正好是 32 位，那么：

- 使用规则的前 24 位作为 tbl24 的索引。
- 如果条目无效（即它不包含规则），则查找一个空闲的 tbl8，将该值的 tbl8 的索引设置为该值，将有效标志设置为 1（表示此条目正在使用中），并将外部 条目标志为 1（意味着查找过程必须继续，因为规则尚未被完全探测）。

如果规则的深度是任何其他值，则必须执行前缀扩展。这意味着规则被复制到所有下一级条目（只要它们不被使用），这也将导致匹配。

作为一个简单的例子，我们假设深度是 20 位。这意味着有可能导致匹配的 IP 地址的前 24 位的 2^（24 - 20）= 16 种不同的组合。因此，在这种情况下，我们将完全相同的条目复制到由这些组合索引的每个位置。

通过这样做，我们确保在查找过程中，如果存在与 IP 地址匹配的规则，则可以在一个或两个内存访问中找到，具体取决于是否需要移动到下一个表。前缀扩展是该算法的关键之一，因为它通过添加冗余来显着提高速度。

14.2.2. 查询

查找过程要简单得多，速度更快。在这种情况下：

- 使用 IP 地址的前 24 位作为 tbl24 的索引。如果该条目未被使用，那么这意味着我们没有匹配此 IP 的规则。如果它有效并且外部条目标志设置为 0，则返回下一跳。
- 如果它是有效的并且外部条目标志被设置为 1，那么我们使用 tbl8 索引来找出要检查的 tbl8，并且将该 IP 地址的最后 8 位作为该表的索引。类似地，如果条目未被使用，那么我们没有与该 IP 地址匹配的规则。如果它有效，则返回下一跳。

14.2.3. 规则数目的限制

规则数量受到诸多不同因素的限制。第一个是规则的最大数量，这是通过 API 传递的参数。一旦达到这个数字，就不可能再添加任何更多的规则到路由表，除非有一个或多个删除。

第二个因素是算法的内在限制。如前所述，为了避免高内存消耗，tbl8 的数量在编译时间有限（此

值默认为 256）。如果我们耗尽 tbl8，我们将无法再添加任何规则。特定路由表中需要多少路由表是很难提前确定的。

只要我们有一个深度大于 24 的新规则，并且该规则的前 24 位与先前添加的规则的前 24 位不同，就会消耗 tbl8。如果相同，那么新规则将与前一个规则共享相同的 tbl8，因为两个规则之间的唯一区别是在最后一个字节内。

默认值为 256 情况下，我们最多可以有 256 个规则，长度超过 24 位，且前三个字节都不同。由于长度超过 24 位的路由不太可能，因此在大多数设置中不应该是一个问题。即便如此，tbl8 的数量也可以通过设置更改。

14.3.用例：IPv4 转发

LPM 算法用于实现 IPv4 转发的路由器所使用的无类别域间路由（CIDR）策略。

14.4.参考

- RFC1519 Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy，http://www.ietf.org/rfc/rfc1519
- Pankaj Gupta, Algorithms for Routing Lookups and Packet Classification, PhD Thesis, Stanford University, 2000（http://klamath.stanford.edu/~pankaj/thesis/ thesis_1sided.pdf）

15. LPM6 库

LPM6（LPM for IPv6）库组件实现了 128 位 Key 的最长前缀匹配表查找方法，该方法通常用于在 IPv6 转发应用程序中找到最佳匹配路由。

15.1.LPM6 API 概述

LPM6 库主要配置参数有：

- 支持的 LPM 规则最大数目：这定义了保存规则的表的大小，也就是最多可以添加的规则数目。
- tbl8 的数量：tbl8 是 trie 的一个节点，是 LPM6 算法的基础。

tbl8 与您可以拥有的规则数量相关，但无法准确预测持有特定数量规则所需的空间，因为它强烈依赖于每个规则的深度和 IP 地址。一个 tbl8 消耗 1 kb 的内存。作为推荐，65536 个 tbl8 应该足以存储数千个 IPv6 规则，但可能因情况而异。

LPM 前缀由一对参数（128 位 Key，深度）表示，深度范围为 1 到 128。LPM 规则由 LPM 前缀和与前缀相关联的一些用户数据表示。该前缀作为 LPM 规则的唯一标识符。在当前实现中，用户数据为 21 位长，称为“下一跳”，对应于其主要用途，用于存储路由表条目中下一跳的 ID。

为 LPM 组件导出的主要方法有：

- 添加 LPM 规则：LPM 规则作为输入参数。如果表中没有存在相同前缀的规则，则将新规则添加到 LPM 表中。如果表中已经存在具有相同前缀的规则，则会更新规则的下一跳。当没有可用空间时返回错误。
- 删除 LPM 规则：LPM 前缀作为输入参数。如果具有指定前缀的规则存在于 LPM 表中，则会被删除。
- 查找 LPM 规则：128 位 Key 作为输入参数。该算法选择代表给定 Key 的最佳匹配的规则，并返回该规则的下一跳。在 LPM 表中存在多个具有相同 128 位 Key 值的规则的情况下，算法选择最高深度的规则作为最佳匹配规则，这意味着该规则在输入键和规则 Key 之间具有最高有效位数匹配。

15.2.实现细节

这个实现是用 IPv4 的算法做的修改（参见 IPv4 LPM 实现细节）。在这种情况下，不是使用两级表，而是使用一级的 tbl24 和 14 级的 tbl8。

该实现可以看作是一个 Multi-bit trie，在每个级别上检查的步长或位数根据级别有所不同。具体来说，在根节点检查 24 位，剩下的 104 位以 8 位的组进行检查。这意味根据添加到表中的规则，该 trie 最多具有 14 个级。

该算法允许用户直接通过存储器访问操作来执行规则查找，存储器访问次数直接取决于规则长度，以及在数据结构中是否存在具有较大深度的其他规则和相同的 Key。它可以在 1 到 14 次访存操作之

间变化，IPv6 中最常用的长度的平均值为 5 次访问操作。
主要数据结构使用以下元素构建：

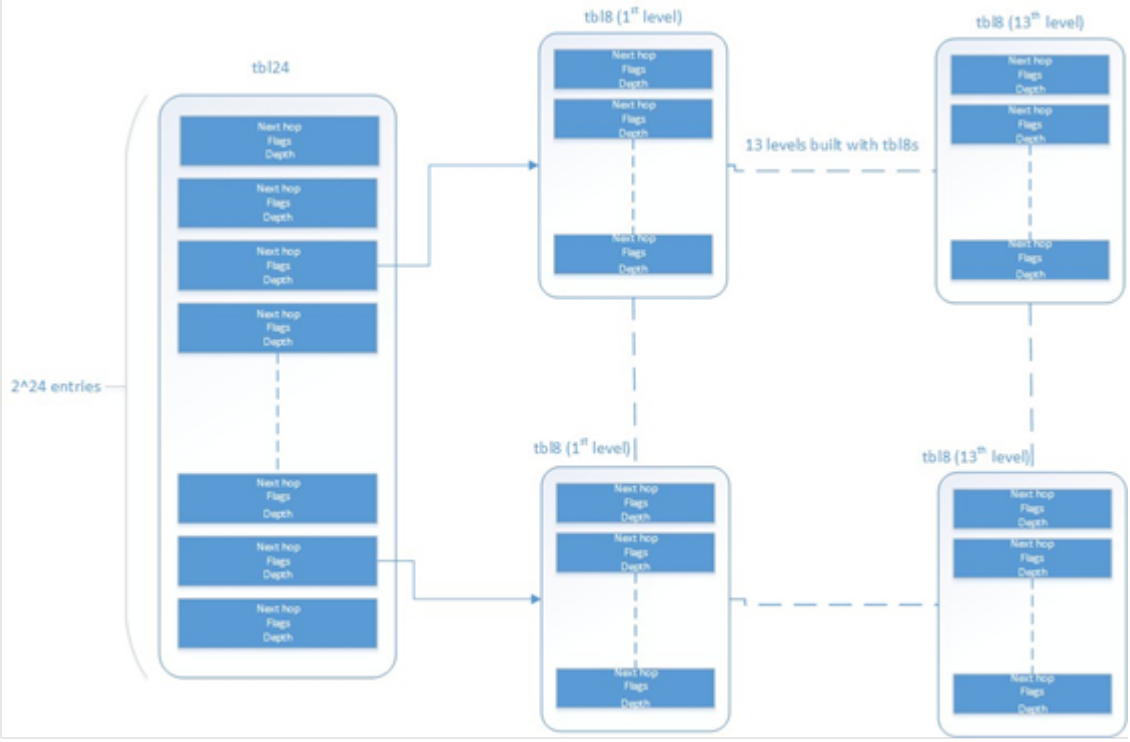
- 一个有 224 个条目的表
- 具有 28 个条目的表，表的数目由 API 配置

第一个表称为 tbl24，使用要查找的 IP 地址的前 24 位进行索引，其余表称为 tbl8，使用 IP 地址的其余字节进行索引，大小为 8 位。这意味着尝试将输入数据包的 IP 地址与存储在 tbl24 或后续 tbl8 中的规则进行匹配的结果，我们可能需要在较深级别的树中继续查找过程。

类似于 IPv4 算法中的限制，为了存储所有可能的 IPv6 规则，我们需要一个具有 2 ^ 128 个条目的表。 由于资源限制，这显然是不可行的。

通过将查找过程分成不同的表/级别并限制 tbl8 的数量，我们可以大大减少内存消耗，同时保持非常好的查找速度（每级一个内存访问）。

Figure 15-37 *Table split into different levels*



表中的条目包含以下字段：

- 下一跳信息或者 tbl8 索引
- 规则深度
- 有效标志
- 有效组标志
- 外部条目标志

第一个字段可以包含指示查找过程应该继续的 tbl8 的索引，或者如果已经找到最长的前缀匹配，则可以包含下一跳本身。规则的深度或长度是存储在特定条目中的规则的位数。标志位用于确定条目

/表是否有效以及搜索过程是否分别完成。
两种类型的表共享相同的结构。

另一个主要数据结构是一个包含规则（IP，下一跳和深度）的主要信息的表。这是一个更高级别的表，用于不同的目的：

- 在添加或删除之前，检查规则是否已经存在，而无需实际执行查找。

删除时，检查是否存在包含要删除的规则是很重要的，因为主数据结构必须相应更新。

15.2.1. 添加

添加规则时存在不同的可能性。如果规则的深度恰好是 24 位，那么：

- 使用规则（IP 地址）作为 tbl24 的索引。
- 如果条目无效（即表中原来不包含规则），则将其下一跳设置为其值，将有效标志设置为 1（表示此条目正在使用中），并将外部条目标志设置为 0（表示查找过程结束，因为这是匹配的最长的前缀）。

如果规则的深度大于 24 位，但倍数为 8，则：

- 使用规则（IP 地址）作为 tbl24 的索引。
- 如果条目无效（即它不包含规则），则查找一个空闲的 tbl8，将该值的 tbl8 的索引设置为该值，将有效标志设置为 1（表示此条目正在使用中），并将外部条目标志为 1（意味着查找过程必须继续，因为规则尚未被完全探测）。
- 使用规则的下 8 位作为下一个 tbl8 的索引。
- 重复该过程，直到达到正确级别的 tbl8（取决于深度），并将其填充到下一跳，将下一个条目标志设置为 0。

如果规则的深度是其他值，则必须执行前缀扩展。这意味着规则被复制到所有条目（尽管它们不被使用）以实现致匹配。

举一个简单的例子，我们假设深度是 20 位。这意味着有可能导致匹配的 IP 地址的前 24 位的 2^（24-20）= 16 种不同的组合。因此，在这种情况下，我们将完全相同的条目复制到由这些组合索引的每个位置。

通过这样做，我们确保在查找过程中，如果存在与 IP 地址匹配的规则，则最多可以在 14 个内存访问中找到，具体取决于需要移动到下一个表的次数。前缀扩展是该算法的关键之一，因为它通过添加冗余显著提高速度。

前缀扩展可以在任何级别执行。因此，例如，深度是 34 位，它将在第三级（第二个基于 tbl8 的级别）执行。

15.2.2. 查询

查找过程要简单得多，速度更快。在这种情况下：

- 使用 IP 地址的前 24 位作为 tbl24 的索引。如果该条目未被使用，那么这意味着我们没有匹配此 IP 的规则。如果它有效并且外部条目标志设置为 0，则返回下一跳。
- 如果它有效并且外部条目标志被设置为 1，那么我们使用 tbl8 索引来找出要检查的 tbl8，并且将该 IP 地址的下一个 8 位作为该表的索引。类似地，如果条目未被使用，那么我们没有与该 IP 地址匹配的规则。如果它是有效的，那么检查外部条目标志以检查新的 tbl8。
- 重复该过程，直到找到无效条目（查找未命中）或外部条目标志设置为 0 的有效条目。在后一种情况下返回下一跳。

15.2.3. 规则数目限制

有不同的因素限制可以添加的规则数量。第一个是规则的最大数量，这是通过 API 传递的参数。一旦达到这个数字，就不可能再添加任何更多的规则到路由表，除非有一个或多个删除。

第二个限制是可用的 tbl8 数量。如果我们耗尽 tbl8s，我们将无法再添加任何规则。很难提前确定其中有多少是特定的路由表所必需的。

在该算法中，单个规则可以消耗的 tbl8 的最大数量为 13，这是级别数减 1，因为前三个字节在 tbl24 中被解析。然而：

- 通常，在 IPv6 上，路由不超过 48 位，这意味着规则通常需要 3 个 tbl8。

如在 LPM for IPv4 算法中所解释的，根据它们的第一个字节是多少，很可能会有几个规则共享一个或多个 tbl8。如果它们共享相同的前 24 位，例如，第二级的 tbl8 将被共享。这可能会在更深的级别再次发生，所以有效的是，如果两个 48 位长的规则在最后一个字节中唯一的区别就可能使用相同的三个 tbl8。

由于其对内存消耗的影响以及可以添加到 LPM 表中的数量或规则，tbl8 的数量是在该版本的算法中通过 API 暴露给用户的参数。一个 tbl8 消耗 1KB 的内存。

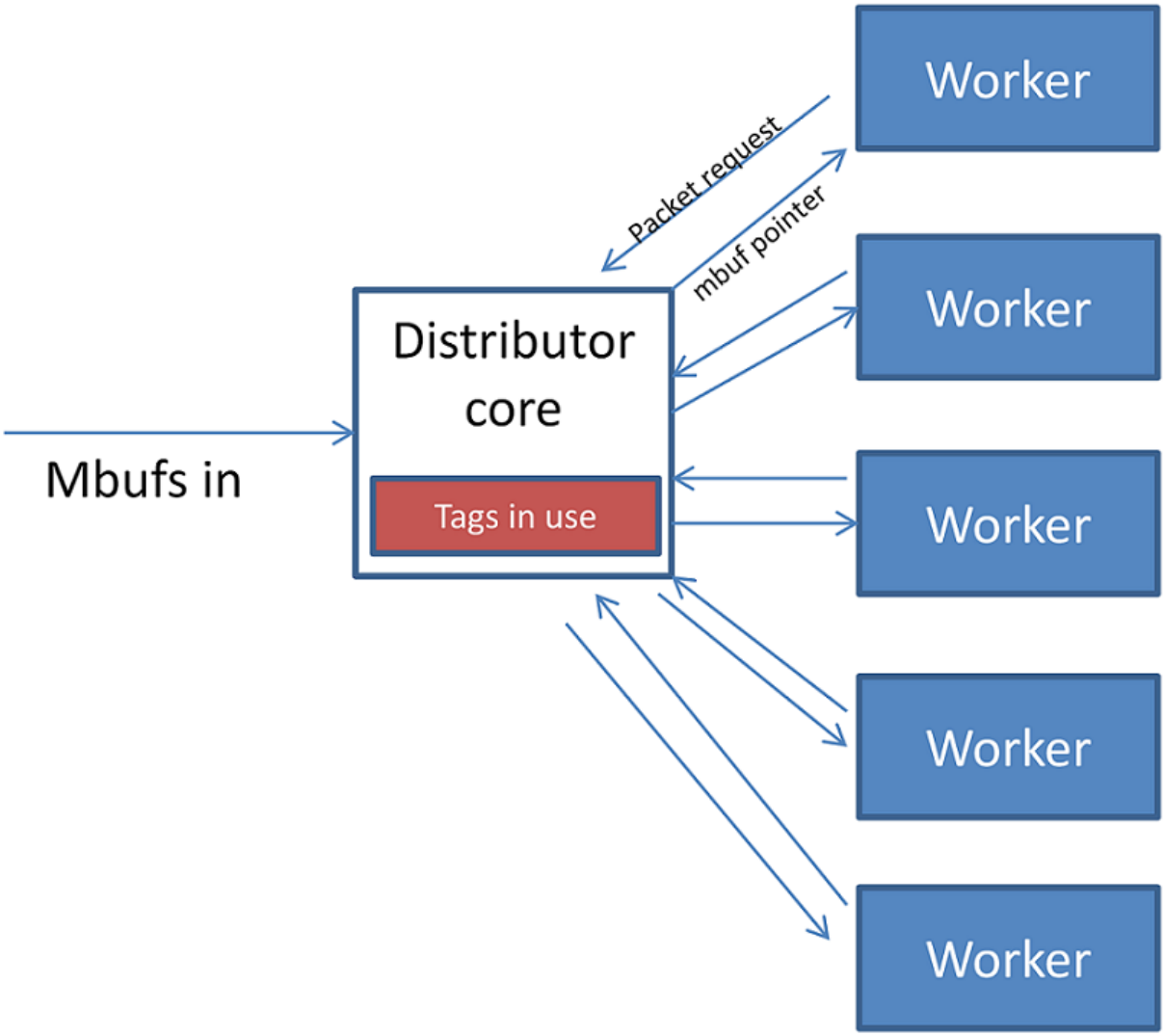
15.3.用例：IPv6 转发

LPM 算法用于实现实现 IP 转发的路由器所使用的无类别域间路由（CIDR）策略。

16. 报文分发库

DPDK 报文分发器是一种库，用于在一次 操作中获取单个数据包，以支持流量的动态负载均衡。当使用这个库时，需要考虑两种角色的逻辑核：首先是负责负载均衡及分发数据包的分发逻辑核，另一个是一组工作逻辑核，负责接收来自分发逻辑核的数据包并对其进行操作。

操作模式如下图所示：
Figure 16-38 Packet Distributor mode of operation



在报文分发器库中有两种 API 操作模式：一种是使用 32bit 的 flow_id，一次向一个 worker 发送一个报文；另一种优化模式是一次性最多发送 8 个数据包给 worker，使用 15bit 的 flow_id。该模式由 rte_distributor_create()函数中的类型字段指定。

16.1.分发逻辑核操作

分发逻辑核执行了大部分的处理以确保数据包在 worker 之间公平分发。分发逻辑核的运作情况如下：

- 分发逻辑核的 lcore 线程通过调用 rte_distributor_process()来获取数据包。
- 所有的 worker lcore 与 distributor lcore 共享一条缓存线行，以便在 worker 和 distributor 之间传递消息和数据包。执行 API 调用将轮询所有 worker 的缓存行，以查看哪些 worker 正在请求数据包。
- 当有 worker 请求数据包时，distributor 从第一步中传过来的一组数据包中取出数据包，并将其分发给 worker。它检查每个数据包中存储在 mbuf 中 RSS 哈希字段中的“tag”，并记录每个 worker 正在处理的 tag。
- 如果输入报文集中的下一个数据包有一个已经被 worker 处理的 tag，则该数据包将排队等待 worker 的处理，并在下一个 worker 请求数据包时，优先考虑其他的数据包。这可以确保不会并发处理具有相同 tag 的两个报文，并且，具有相同 tag 的两个报文按输入顺序被处理。
- 一旦传递给执行 API 的所有报文已经分发给 worker，或者已经排队等待给定 tag 的 worker 处理，则执行 API 返回给调用者。

Distributor lcore 可以使用的其他功能有：
- rte_distributor_returned_pkts()
- rte_distributor_flush()
- rte_distributor_clear_returns()

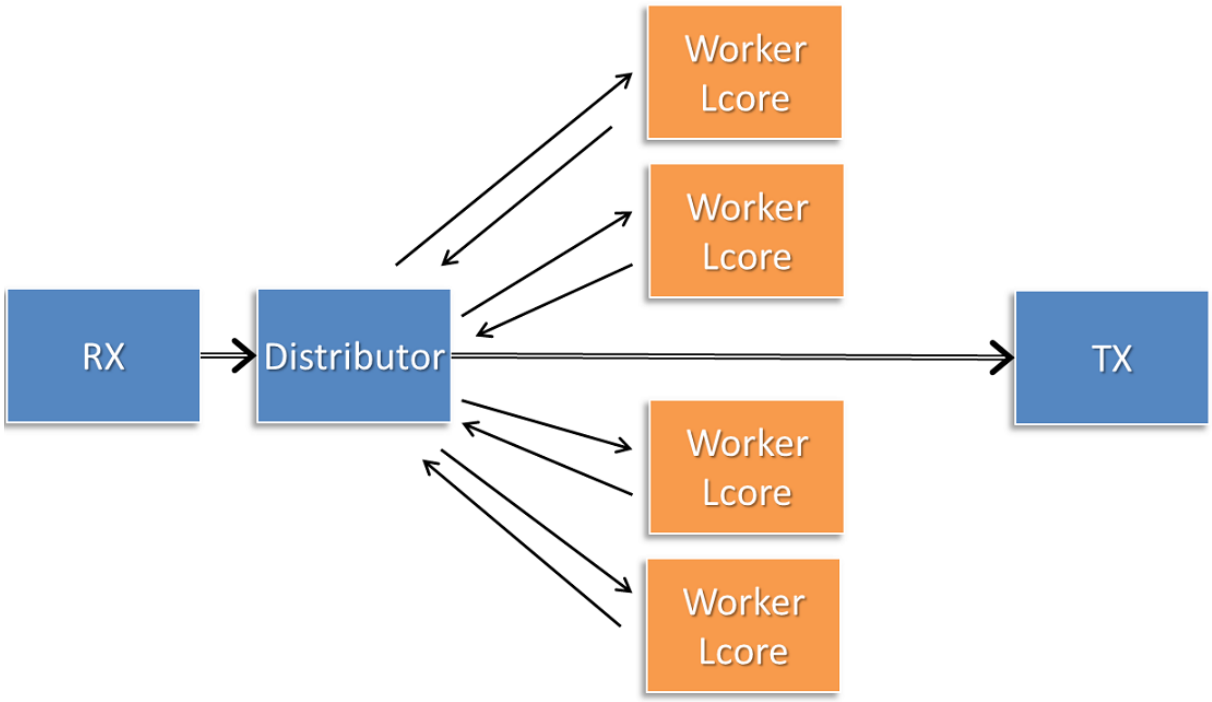
其中最重要的 API 调用是“rte_distributor_returned_pkts()”，它只能在调用进程 API 的 lcore 上调用。 它将所有 worker core 完成处理的所有数据包返回给调用者。在这组返回的数据包中，共享相同标签的所有数据包将按原始顺序返回。

注意：
如果 worker lcore 在内部缓存数据包进行批量传输，则共享 tag 的数据包可能会出现故障。一旦一个 worker lcore 请求一个新的数据包，distributor 就会假定它已经完成了先前的数据包，因此具有相同 tag 的附加数据包可以安全地分配给其他 worker，然后他们可能会更早地刷新缓冲的数据包，使数据包发生故障。

注意：
对于不共享公共数据包 tag 的数据包，不提供数据包排序保证。

使用上述执行过程及 returned_pkts API，可以使用以下应用程序工作流，同时允许维护由 tag 识别的数据包流中的数据包顺序。

Figure 16-39 Application workflow



之前提到的 flush 和 clear_returns API 调用可能不太用于进程和 returned_pkts APIS，并且主要用于帮助对库进行单元测试。可以在 DPDK API 参考文档中找到这些功能及其用途的描述。

16.2.Worker Operation

Worker lcore 是对 distributor 分发的数据包进行实际操作的 lcore。 Worker 调用 rte_distributor_get_pkt() API 在完成处理前一个数据包时请求一个新的数据包。前一个数据包应通过将其作为最终参数传递给该 API 调用而返回给分发器组件。

有时候可能需要改变 worker lcore 的数量，这取决于业务负载，即在较轻的负载时节省功率，可以 worker 通过调用 rte_distributor_return_pkt()接口停止处理报文，以指示在完成当前数据包处理后，不需要新的数据包。

17. 排序器库

重新排序库提供了一种根据序列号重新排序 mbufs 的机制。

17.1.操作

重新排序库本质上是一个重新排列 mbufs 的缓冲区。用户将乱序的 mbufs 插入到重新排序缓冲区中，并从中输出顺序 mbufs。

在给定的时间，重新排序缓冲区包含序列号在序列窗口内的 mbufs。顺序窗口由缓冲区配置的可以维护的最小序列号和条目数决定。例如，给定具有 200 个条目并且最小序列号为 350 的重排序缓冲器，序列窗口分别具有 350 和 550 的低和高限制。

当插入 mbufs 时，重新排序库根据插入的 mbuf 的序列号区分 valid，late 和 early 的 mbuf：

- Valid：序列号在有序窗口的限制内。
- Late：序列号在窗口限制外，小于下限。
- Early：序列号在窗口限制外，大于上限。

重新排序缓冲区直接返回 late mbufs，并尝试适应 early mbufs。

17.2.实现细节

重新排序库被实现为一对缓冲区，称为 Order buffer 和 Ready buffer。

在插入调用时，valid mbufs 将直接插入到 Order buffer 中，late mbufs 将直接返回给用户错误。

对于 early buffer 的情况，重排序缓冲区将尝试移动窗口（递增最小序列号），以使 mbuf 成为有效的一个。为此，Order buffer 中的 mbufs 被移动到就 Ready buffer 中。任何尚未到达的 mbufs 都被忽略，且将变成 late mbufs。这意味着只要 Ready buffer 中有空间，窗口将被移动以适应 early mbufs，否则将在重新排序窗口之外。

例如，假设我们有一个具有 350 个最小序列号的 200 个条目的缓冲区，并且我们需要插入一个具有 565 序列号的 early mbuf。这意味着我们需要移动窗口至少 15 个位置来容纳 mbuf。只要在 Ready buffer 中有空间，重新排序缓冲区将尝试将至少在 Order buffer 中的下一个 15 个槽中的 mbufs 移动到 Ready buffer 区。在这一点上的顺序缓冲区中的任何间隙将被跳过，并且这些数据包在报文到达时将被报告为 late buffer 的数据包。将数据包移动到 Ready buffer 的过程继续超出所需的最小值，直到遇到了缓冲区中的间隙，即缺少 mbuf。

排出 mbufs 时，重新排序缓冲区首先返回 Ready buffer 中的 mbufs，然后从 Order buffer 返回到尚未到达的 mbufs。

17.3.用例：报文分发

使用 DPDK 数据包分发器的应用程序可以利用重新排序库以与它们相同的顺序传送数据包。

基本的报文分配器用例将由具有多个 worker cors 的分配器组成。worker 对数据包的处理不能保证按顺序进行，因此可以使用重排序缓冲区来尽可能多地重排数据包。

在这种情况下，distributor 将序列号分配给 mbufs，然后再将其发送给工作人员。随着 worker 完成处理数据包，distributor 将这些 mbufs 插入重排序缓冲区，最后传输排出的 mbufs。

18. IP 分片及重组库

IP 分段和重组库实现 IPv4 和 IPv6 报文的分片和重组。

18.1.报文分片

报文分段例程将输入报文划分成多个分片。rte_ipv4_fragment_packet()和 rte_ipv6_fragment_packet()函数都假定输入 mbuf 数据指向报文的 IP 报头的开始（即 L2 报头已经被剥离）。为了避免复制实际数据包的数据，使用零拷贝技术（rte_pktmbuf_attach）。对于每个片段，将创建两个新的 mbuf：

- Direct mbuf：mbuf 将包含新片段的 L3 头部。
- Indirect mbuf：源数据包附加到 mbuf。数据字段指向原始数据包数据的附加数据偏移量开始处。

然后将 L3 头部从原始 mbuf 复制到“direct”mbuf 并更新以反映新的碎片状态。 请注意，对于 IPv4，不会重新计算头校验和，其值设置为零。

最后，通过 mbuf 的下 next 字段将每个片段的“direxT”和“indirect”mbuf 链接在一起，以构成新片段的数据包。

调用者可以明确指定哪些 mempools 应用于从中分配“direct”和“indirect”mbufs。

有关 direct 和 indirect mbufs 的信息，请参阅直接和间接缓冲区。

18.2.报文重组

18.2.1. IP 分片表

报文分片表中维护已经接收到的数据包片段的信息。

每个 IP 数据包由三个字段：<源 IP 地址>，<目标 IP 地址>，<ID>唯一标识。

请注意，报文分片表上的所有更新/查找操作都不是线程安全的。因此，如果不同的执行上下文（线程/进程）要同时访问同一个表，那么必须提供一些外部同步机制。

每个表项可以保存最多 RTE_LIBRTE_IP_FRAG_MAX（默认值为 4）片段的数据包的信息。

代码示例，演示了创建新的片段表：

```
frag_cycles = (rte_get_tsc_hz() + MS_PER_S - 1) / MS_PER_S * max_flow_ttl;
bucket_num = max_flow_num + max_flow_num / 4;
frag_tbl = rte_ip_frag_table_create(max_flow_num, bucket_entries, max_flow_num, frag_cycles, socket_id);
```

内部片段表是一个简单的哈希表。 基本思想是使用两个哈希函数和<bucket_entries> *关联性。 这为每个 Key 在散列表中提供了 2 * <bucket_entries> 可能的位置。当发生冲突并且所有 2 * <bucket_entries>都被占用时，ip_frag_tbl_add()只是返回失败，而不是将现有的 Key 重新插入到另外的位置。

此外，驻留在表中的条目如果比<max_cycles>更长，被认为是无效的，可以被新的条目删除/替换。

请注意，重新组合需要分配很多 mbuf。在任何给定时间（2 * bucket_entries * RTE_LIBRTE_IP_FRAG_MAX * <每个数据包的最大 mbufs 数>>）可以存储在等待剩余片段的 Fragment Table 中。

18.2.2. 报文重组

报文分组处理和重组由 rte_ipv4_frag_reassemble_packet()/rte_ipv6_frag_reassemble_packet()完成。它们返回一个指向有效 mbuf 的指针，它包含重新组合的数据包，或者返回 NULL（如果数据包由于某种原因而无法重新组合）。

这些功能包括：

1. 搜索片段表，输入数据包的<IPv4 源地址，IPv4 目的地址，数据包 ID>。
2. 如果找到该条目，则检查该条目是否已经超时。如果是，则释放所有以前收到的碎片，并从条目中删除有关它们的信息。
3. 如果没有找到这样的 Key 的条目，那么尝试通过以下两种方法之一创建一个新的：
   a) 用作空条目。
   b) 删除一个超时条目，与它 mbufs 关联的空闲 mbufs，并在其中存储一个带有指定键的新条目。
4. 使用新的片段信息更新条目，并检查是否可以重新组合数据包（数据包的条目包含所有片段）。
   a) 如果是，则重新组装数据包，将表的条目标记为空，并将重新组装的 mbuf 返回给调用者。
   b) 如果否，则向调用者返回一个 NULL。

如果在分组处理的任何阶段遇到错误（例如：不能将新条目插入片段表或无效/超时片段），则该函数将释放所有与分组片段相关联的标记表条目 作为无效并将 NULL 返回给调用者。

18.2.3. 调试日志及统计收集

RTE_LIBRTE_IP_FRAG_TBL_STAT 配置宏用于控制片段表的统计信息收集。默认情况下未启用。

RTE_LIBRTE_IP_FRAG_DEBUG 控制 IP 片段处理和重组的调试日志记录。默认情况下禁用。请注意，在日志记录包含大量详细信息时，会减慢数据包处理速度，并可能导致丢失大量数据包。

19. Librte_pdump 库

librte_pdump 库为 DPDK 中的数据包捕获提供了一个框架。该库将 Rx 和 Tx mbufs 的完整复制到新的 mempool，因此会降低应用程序的性能，故建议只使用该库进行调试。

该库提供以下 API 来初始化数据包捕获框架，启用或禁用数据包捕获，或者对其进行反初始化：

- rte_pdump_init()：初始化数据包捕获框架。
- rte_pdump_enable()：在给定的端口和队列上进行数据包捕获。注意：API 中的过滤器选项是用于未来增强功能的占位符。
- rte_pdump_enable_by_deviceid()：启用在给定设备 ID（vdev 名称或 pci 地址）和队列上的数据包捕获。 注意：API 中的过滤器选项是用于未来增强功能的占位符。
- rte_pdump_disable()：禁用给定端口和队列上的数据包捕获。
- rte_pdump_disable_by_deviceid()：禁用给定设备 ID（vdev 名称或 pci 地址）和队列上的数据包捕获。
- rte_pdump_uninit()：反初始化数据包捕获框架。
- rte_pdump_set_socket_dir()：设置服务器和客户端套接字路径。注意：此 API 不是线程安全的。

19.1.操作

librte_pdump 库适用于客户端/服务器型号。服务器负责启用或禁用数据包捕获，客户端负责请求启用或禁用数据包捕获。

数据包捕获框架作为程序初始化的一部分，在 pthread 中创建 pthread 和服务器套接字。调用框架初始化的应用程序将创建服务器套接字，可能是在应用程序传入的路径，也可能是默认路径（root 用户的/var/run/.dpdk，非 root 用户～/.dpdk）下创建。

请求启用或禁用数据包捕获的应用程序将在应用程序传入的路径下或默认路径（root 用户的/var/run/.dpdk，非 root 用户～/.dpdk）下创建客户机套接字，用户将请求发送到服务器。服务器套接字将监听用于启用或禁用数据包捕获的客户端请求。

19.2.实现细节

库 API rte_pdump_init()通过创建 pthread 和服务器套接字来初始化数据包捕获框架。 pthread 上下文中的服务器套接字将监听客户端请求以启用或禁用数据包捕获。

库 API rte_pdump_enable()和 rte_pdump_enable_by_deviceid()启用数据包捕获。每次调用这些 API 时，库创建一个单独的客户端套接字，生成“pdump enable”请求，并将请求发送到服务器。在套接字上监听的服务器将通过对给定的端口或设备 ID 和队列组合的以太网 Rx/TX 注册回调函数来接收请求并启用数据包捕获。然后，服务器将镜像数据包到新的 mempool 并将它们入队到客户端传递给这些 API 的 rte_ring。服务器还将响应发送回客户端，以了解处理过的请求的状态。从服务器收到响应后，客户端套接字关闭。

库 API rte_pdump_disable()和 rte_pdump_disable_by_deviceid()禁用数据包捕获。每次调用这些 API 时，库会创建一个单独的客户端套接字，生成“pdump disable”请求，并将请求发送到服务器。正在监听套接字的服务器将通过对给定端口或设备 ID 和队列组合的以太网 RX 和 TX 删除回调函数来执行请求并禁用数据包捕获。服务器还将响应发送回客户端，以了解处理过的请求的状态。从服务器收到响应后，客户端套接字关闭。

库 API rte_pdump_uninit()通过关闭 pthread 和服务器套接字来初始化数据包捕获框架。

库 API rte_pdump_set_socket_dir()根据 API 的类型参数将给定路径设置为服务器套接字路径或客户端套接字路径。如果给定路径为 NULL，则将选择默认路径（即 root 用户的/var/run/.dpdk 或非 root 用户的～/.dpdk）。如果服务器套接字路径与默认路径不同，客户端还需要调用此 API 来设置其服务器套接字路径。

19.3.用例:抓包

DPDK 应用程序/pdump 工具是基于此库开发的，用于捕获 DPDK 中的数据包。用户可以用它来开发自己的数据包捕获工具。

20. 多进程支持

在 DPDK 中，多进程支持旨在允许一组 DPDK 进程以简单的透明方式协同工作，以执行数据包处理或其他工作负载。为了支持此功能，已经对核心的 DPDK 环境抽象层（EAL）进行了一些增加。

EAL 已被修改为允许不同类型的 DPDK 进程产生，每个 DPDK 进程在应用程序使用的 hugepage 内存上具有不同的权限。现在可以指定两种类型的进程：

- primary processes：可以初始化，拥有共享内存的完全权限
- secondary processes：不能初始化共享内存，但可以附加到预初始化的共享内存并在其中创建对象。

独立 DPDK 进程是 primary processes，而 secondary processes 只能与主进程一起运行，或者主进程已经为其配置了 hugepage 共享内存。

为了支持这两种进程类型以及稍后描述的其他多进程设置，EAL 还提供了两个附加的命令行参数：

- --proc-type：用于将给定的进程实例指定为 primary processes 或 secondary processes DPDK 实例。
- --file-prefix：以允许不希望协作具有不同存储器区域的进程。

DPDK 提供了许多示例应用程序，演示如何可以一起使用多个 DPDK 进程。这些用例在《DPDK Sample Application 用户指南》中的“多进程示例应用”一章中有更详尽的记录。

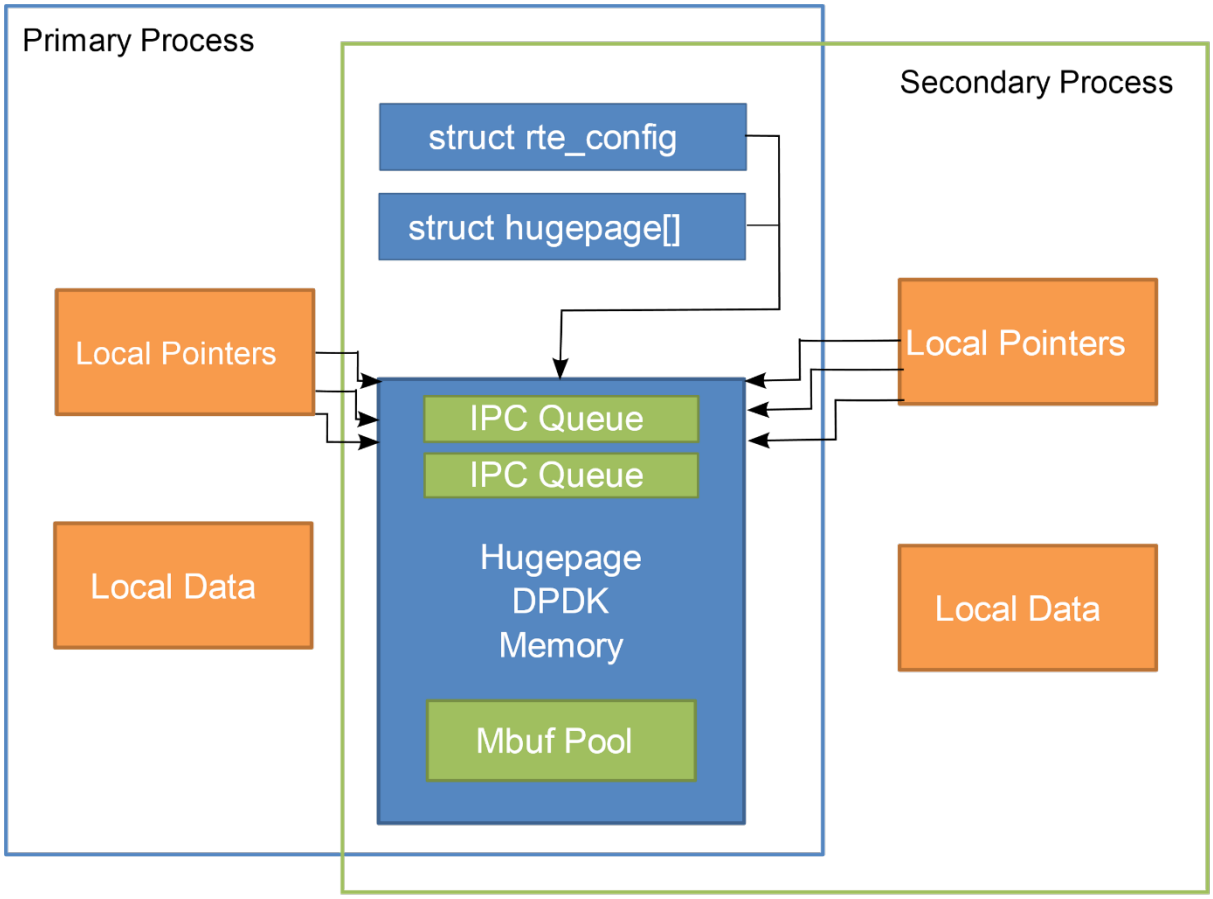
20.1.内存共享

使用 DPDK 的多进程应用程序工作的关键要素是确保内存资源在构成多进程应用程序的进程之间正确共享。一旦存在可以通过多个进程访问的共享存储器块，则诸如进程间通信（IPC）的问题就变得简单得多。

在独立进程或者 primary processes 启动时，DPDK 向内存映射文件中记录其使用的内存配置的详细信息，包括正在使用的 hugepages，映射的虚拟地址，存在的内存通道数等。当 secondary processes 启动时，这些文件被读取，并且 EAL 在 secondary processes 中重新创建相同的内存配置，以便所有内存区域在进程之间共享，并且所有指向该内存的指针都是有效的，并且指向相同的对象。

有关 Linux 内核地址空间布局随机化（ASLR）如何影响内存共享的详细信息，请参阅多进程限制章节。

Figure 20-40 Memory Sharing in the DPDK Multi-process Sample Application



EAL 还支持自动检测模式（由 EAL -proc-type = auto 标志设置），如果主实例已经在运行，则 DPDK 进程作为辅助实例启动。

20.2.部署模式

20.2.1. 对称/对等进程

DPDK 多进程支持可用于创建一组对等进程，每个进程执行相同的工作负载。该模型相当于具有多个线程，每个线程都运行相同的主循环功能，如大多数提供的 DPDK 示例应用程序中所完成的一样。 在此模型中，应使用--proc-type = primary EAL 标志生成第一个生成的进程，而所有后续实例都应使用--proc-type = secondary 标志生成。

simple_mp 和 symmetric_mp 示例应用程序演示了此模型的用法。它们在《DPDK Sample Application 用户指南》中“多进程示例应用”一章中有描述。

20.2.2. 非对称/非对等进程

可用于多进程应用程序的替代部署模型是具有单个 primary process 实例，充当负载均衡器或

distributor，在作为 secondary processes 运行的 worker 或客户机线程之间分发接收到的数据包。在这种情况下，广泛使用 rte_ring 对象，它们位于共享的 hugepage 内存中。

client_server_mp 示例应用程序显示此模型用法。在《DPDK Sample Application 用户指南》中“多进程示例应用”一章中有描述。

20.2.3. 运行多个独立的 DPDK 应用程序

除了涉及多个 DPDK 进程的上述情况之外，可以并行运行多个 DPDK 进程，这些进程都可以独立工作。使用 EAL 的--file-prefix 参数提供对此使用场景的支持。

默认情况下，EAL 使用 rtemap_X 文件名在每个 hugetlbfs 文件系统上创建 hugepage 文件，其中 X 的范围为 0 到最大的 hugepages -1。同样，当以 root 身份运行（或以非 root 用户身份运行时为$ HOME / .rte_config），如果文件系统和设备权限为空，则会在每个进程中使用/var/run/.rte_config 文件名创建共享配置文件）。以上每个文件名的部分可以使用 file-prefix 参数进行配置。

除了指定 file-prefix 参数外，并行运行的任何 DPDK 应用程序都必须明确限制其内存使用。这通过将-m 标志传递给每个进程来指定每个进程可以使用多少 hugepage 内存（以兆字节为单位）（或通过--socket-mem 来指定每个进程可以使用每个套接字的多少 hugepage 内存）。

注意，在单台机器上并行运行的独立 DPDK 实例无法共享任何网络端口。一个进程使用的任何网络端口都应该在其他进程中列入黑名单。

20.2.4. 运行多个独立的 DPDK 应用程序组

以同样的方式，可以在单个系统上并行运行独立的 DPDK 应用程序，这也可以简单地扩展到并行运行 DPDK 应用程序的多进程组。在这种情况下，secondary processes 必须使用与其共享内存连接的 primary process 相同的--file-prefix 参数。

并行运行的多个独立 DPDK 进程的所有限制和问题也适用于此使用场景。

20.3.多进程限制

运行 DPDK 多进程应用程序时存在一些限制。其中一些记录如下：

- 多进程功能要求在所有应用程序中都存在完全相同的 hugepage 内存映射。Linux 安全功能，地址空间布局随机化（ASLR）可能会干扰此映射，因此可能需要禁用此功能才能可靠地运行多进程应用程序。

禁用地址空间布局随机化（ASLR）可能具有安全隐患，因此建议仅在绝对必要时才被禁用，并且只有在了解了此更改的含义时。

- 作为单个应用程序运行并使用共享内存的所有 DPDK 进程必须具有不同的 coremask / corelist 参

数。任何相同的逻辑内核不可能拥有 primary 和 secondary 实例或两个 secondary 实例。尝试这样做可能会导致内存池缓存的损坏等问题。

- 中断的传递，如 Ethernet *设备链路状态中断，在 secondary process 中不起作用。所有中断仅在 primary process 内触发。在多个进程中需要中断通知的任何应用程序都应提供自己的机制，将中断信息从 primary process 转移到需要该信息的任何 secondary process。
- 不支持使用基于不同编译二进制文件运行的多个进程之间的函数指针，因为在一个进程中给定函数的位置可能与其中一个进程的位置不同。这样可以防止 librte_hash 库在多线程实例中正常运行，因为它在内部使用了一个指向散列函数的指针。

要解决此问题，建议多进程应用程序通过直接从代码中调用散列函数，然后使用 rte_hash_add_with_hash()/rte_hash_lookup_with_hash()函数来执行哈希计算，而不是内部执行散列的函数，例如 rte_hash_add()/rte_hash_lookup()。

- 根据所使用的硬件和所使用的 DPDK 进程的数量，可能无法在每个 DPDK 实例中都使用 HPET 定时器。可用于 Linux *用户空间的 HPET comparators 的最小数量可能只有一个，这意味着只有第一个 primary DPDK 进程实例可以打开和 mmap / dev / hpet。如果所需 DPDK 进程的数量超过可用的 HPET comparators 数量，则必须使用 TSC（此版本中的默认计时器）而不是 HPET。

21. 内核网络接口卡接口

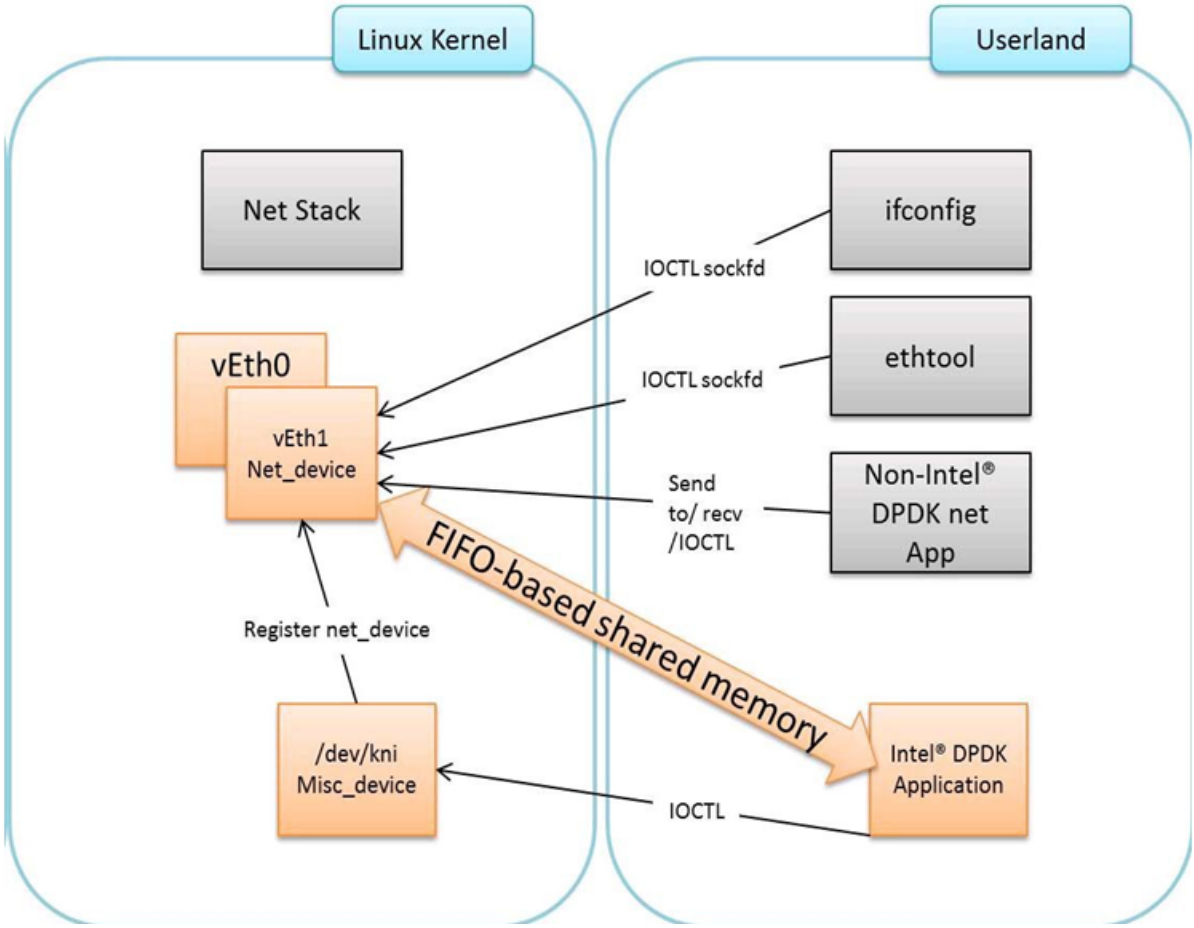
DPDK Kernel NIC Interface（KNI）允许用户空间应用程序访问 Linux *控制面。

使用 DPDK KNI 的好处是：

- 比现有的 Linux TUN / TAP 接口更快（通过消除系统调用和 copy_to_user()/copy_from_user()操作）。
- 允许使用标准 Linux 网络工具（如 ethtool，ifconfig 和 tcpdump）管理 DPDK 端口。
- 允许与内核网络堆栈的接口。

使用 DPDK 内核 NIC 接口的应用程序的组件如图所示。

Figure 21-41 Components of a DPDK KNI Application



21.1.DPDK KNI 内核模块

KNI 内核可加载模块支持两种类型的设备：

- 其他设备：
  - 创建网络设备（通过 ioctl 调用）。

- 维护所有 KNI 实例共享的内核线程上下文（模拟网络驱动程序的 RX 端）。
- 对于单内核线程模式，维护所有 KNI 实例共享的内核线程上下文（模拟网络驱动程序的 RX 端）。
- 对于多个内核线程模式，为每个 KNI 实例（模拟新驱动程序的 RX 侧）维护一个内核线程上下文。

- 网络设备：
  - 通过实现由 struct net_device 定义的诸如 netdev_ops，header_ops，ethtool_ops 之类的几个操作提供的 Net 功能，包括支持 DPDK mbufs 和 FIFO。
  - 接口名称由用户空间提供。
  - MAC 地址可以是真正的 NIC MAC 地址或随机的。

21.2.KNI 创建及删除

KNI 接口由 DPDK 应用程序动态创建。接口名称和 FIFO 详细信息由应用程序通过 ioctl 调用使用 rte_kni_device_info 结构提供，该结构包含：

- 接口名称。
- 相关 FIFO 的相应存储器的物理地址。
- Mbuf mempool 详细信息，包括物理和虚拟（计算 mbuf 指针的偏移量）。
- PCI 信息。
- Core。

有关详细信息，请参阅 DPDK 源代码中的 rte_kni_common.h。

物理地址将重新映射到内核地址空间，并存储在单独的 KNI 上下文中。

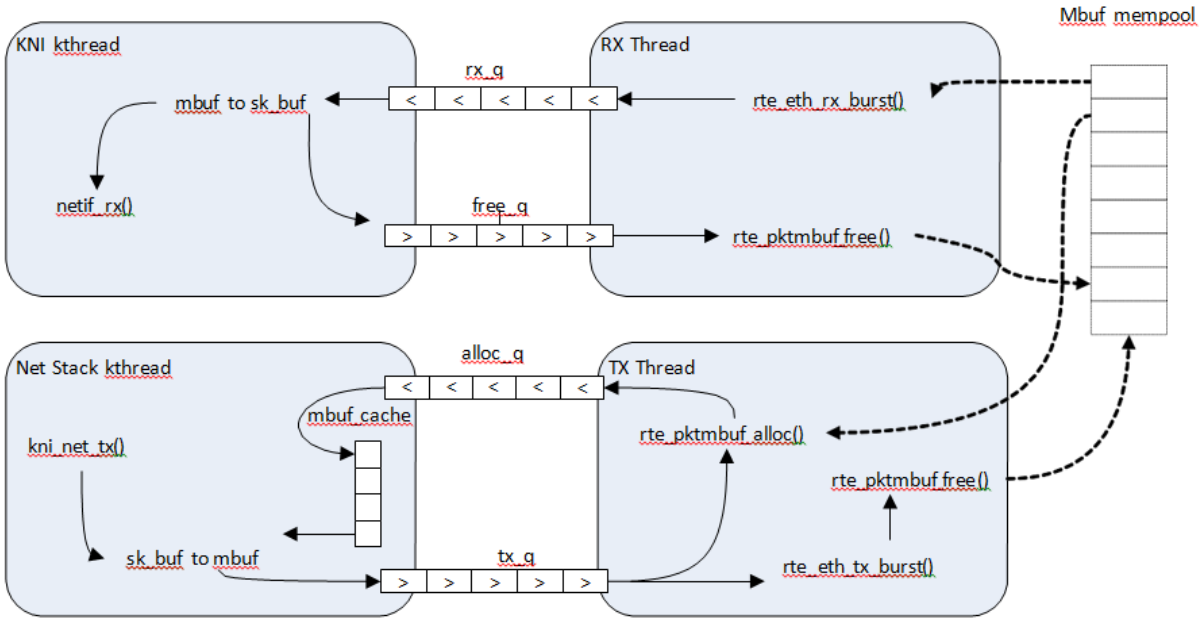
内核 RX 线程（单线程和多线程模式）的亲和力由 force_bind 和 core_id 配置参数控制。

创建后，DPDK 应用程序可以动态删除 KNI 接口。此外，所有未删除的 KNI 接口将在杂项设备（DPDK 应用程序关闭时）的释放操作中被删除。

21.3.DPDK 缓冲区流

为了最小化在内核空间中运行的 DPDK 代码的数量，mbuf mempool 仅在用户空间中进行管理。内核模块可以感知 mbufs，但是所有 mbuf 分配和释放操作将仅由 DPDK 应用程序处理。

Figure 21-42 Packet Flow via mbufs in the DPDK KNI



21.4.用例: Ingress

在 DPDK RX 侧，mbuf 由 PMD 在 RX 线程上下文中分配。该线程将 mbuf 入队到 rx_q FIFO 中。KNI 线程将轮询所有 KNI 活动设备。如果 mbuf 出队，它将被转换为 sk_buff，并通过 netif_rx()发送到网络协议栈。必须释放出队的 mbuf，将指针返回到 free_q FIFO 中。

RX 线程在相同的主循环中轮询该 FIFO，并在出队之后释放 mbuf。

21.5.用例: Egress

对于数据包出口，DPDK 应用程序必须首先入队几个 mbufs 才能在内核端创建一个 mbuf 缓存。

通过调用 kni_net_tx()回调，从 Linux 网络堆栈接收数据包。mbuf 出队（因为使用缓存，所以无需等待），并填充了来自 sk_buff 的数据。然后释放 sk_buff，并将 mbuf 发送到 tx_q FIFO。

DPDK TX 线程执行 mbuf 出队，并将其发送到 PMD（通过 rte_eth_tx_burst()）。 然后将 mbuf 放回缓存中。

21.6.以太网工具

Ethtool 是 Linux 专用工具，在内核中具有相应的支持，每个网络设备必须为支持的操作注册自己的回调。目前的实现使用 igb / ixgbe 修改的 Linux 驱动程序进行 ethtool 支持。i40e 和 VM（VF 或 EM 设备）不支持 Ethtool。

21.7.链路状态及 MTU 改变

链路状态和 MTU 变化是通常通过 ifconfig 完成的网络接口操作。该请求是从内核端（在 ifconfig 进程的上下文中）发起的，由用户空间 DPDK 应用程序处理。应用程序轮询请求，调用应用程序处理程序并将响应返回到内核空间。

应用处理程序可以在创建接口时注册，也可以在运行时再注册/卸载。这提供了多进程方案（其中 KNI 在 primary process 中创建，在 secondary process 中处理回调）的灵活性。约束是单个进程可以注册和处理请求。

22. DPDK 功能的线程安全

DPDK 由几个库组成。这些库中的某些功能可以同时被多个线程安全地调用，而另一部分则不能。本节介绍开发人员在构建自己的应用程序时考虑这些问题。

DPDK 的运行时环境通常是每个逻辑核上的单个线程。但是，在某些情况下，它不仅是多线程的，而且是多进程的。通常，最好避免在在线程和/或进程之间共享数据结构。如果不可能，则执行块必须以线程安全的方式访问数据。可以使用诸如原子操作或锁的机制，这将允许执行块串行操作。但是，这可能会对应用程序的性能产生影响。

22.1.快速路径 API

在数据面中运行的应用程序对性能敏感，但这些库中的某些函数可能不会多线程并发调用。PMD 中的 Hash，LPM 和 mempool 库以及 RX / TX 都是这样的例子。

通过设计，Hash 和 LPM 库线程不安全，不能并行调用，以保持性能。然而，如果需要，开发人员可以在这些库之上添加封装层以提供线程安全性。在所有情况下都不需要锁，并且在哈希和 LPM 库中，可以在多个线程中并行执行值的查找。但是，当访问单个哈希表或 LPM 表时，添加，删除或修改值不能不使用锁在多个线程中完成。锁的另一个替代方法是创建这些表的多个实例，允许每个线程自己的副本。

PMD 的 RX 和 TX 是 DPDK 应用程序中最关键的方面，建议不要使用锁，因为它会影响性能。但是请注意，当每个线程在不同的 NIC 队列上执行 I/O 时，这些功能可以安全地从多个线程使用。如果多个线程在同一个 NIC 端口上使用相同的硬件队列，则需要锁定或某种其他形式的互斥。

Ring 库的实现基于无锁缓冲算法，保持其原有的线程安全设计。此外，它可以为多个或单个消费者/生产者入队/出队操作提供高性能。mempool 库基于 DPDK 无锁 ring 库，因此也是多线程安全的。

22.2.非性能敏感 API

在第 25.1 节描述的性能敏感区域之外，DPDK 为大多数其他库提供线程安全的 API。例如，malloc 和 memzone 功能可以安全地用于多线程和多进程环境中。

PMD 的设置和配置不是性能敏感的，但也不是线程安全的。在多线程环境中 PMD 设置和配置期间的多次读/写可能会被破坏。由于这不是性能敏感的，开发人员可以选择添加自己的层，以提供线程安全的设置和配置。预计在大多数应用中，网络端口的初始配置将由启动时的单个线程完成。

22.3.库初始化

建议 DPDK 库在应用程序启动时在主线程中初始化，而不是随后在转发线程中初始化。但是，

DPDK 会执行检查，以确保库仅被初始化一次。如果尝试多次初始化，则返回错误。
在多进程情况下，共享内存的配置信息只能由 primary process 初始化。此后，primary process 和 secondary process 都可以分配/释放最终依赖于 rte_malloc 或 memzone 的任何内存对象。

22.4.中断线程

DPDK 在轮询模式下几乎完全用于 Linux 用户空间。对于诸如接收 PMD 链路状态改变通知的某些不经常的操作，可以在主 DPDK 处理线程外部的附加线程中调用回调。这些函数回调应避免操作也由普通 DPDK 线程管理的 DPDK 对象，如果需要这样做，应用程序就可以为这些对象提供适当的锁定或互斥限制。

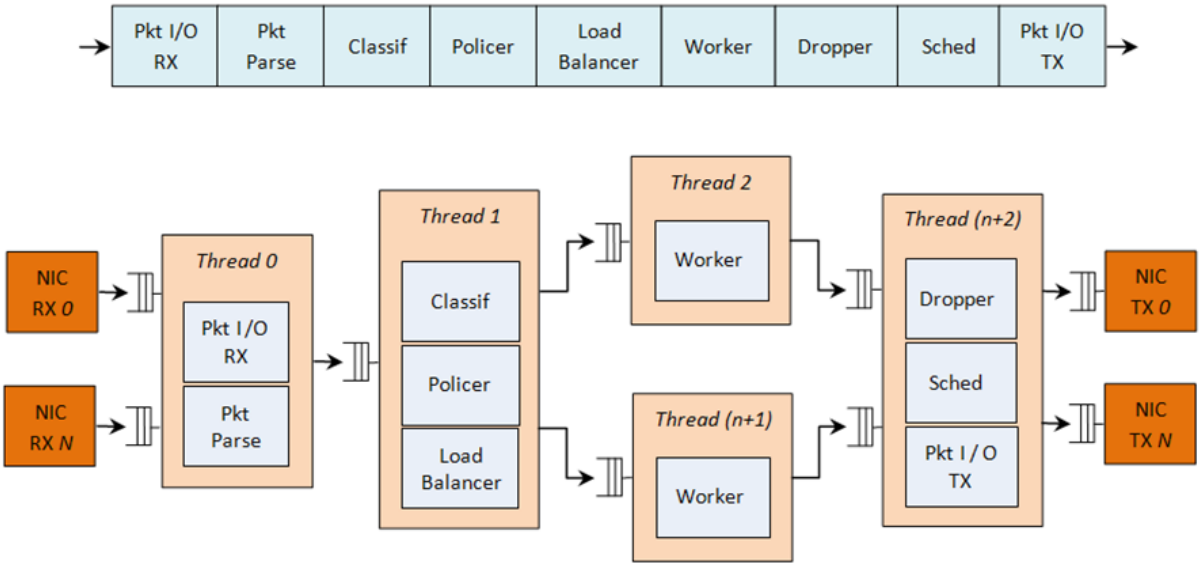
23. QoS 框架

本章介绍了 DPDK 服务质量（QoS）框架。

23.1.支持 QoS 的数据包水线

具有 QoS 支持的复杂报文处理流水线的示例如下图所示。

Figure 23-43 Complex Packet Processing Pipeline with QoS Support



这个水线使用可重复使用的 DPDK 软件库构建。在这个流程中实现 QoS 的主要模块有：策略器，缓存器和调度器。下表列出了各块的功能描述。

#	Block	Functional Description
1	Packet I/O RX & TX	多个 NIC 端口的报文接收/传输。用于 Intel 1GbE/10GbE NIC 的轮询模式驱动程序（PMD）。
2	Packet parser	识别输入数据包的协议栈。检查数据包头部的完整性。
3	Flow classification	将输入数据包映射到已知流量上。使用可配置散列函数（jhash，CRC 等）和桶逻辑来处理冲突的精确匹配表查找。
4	Policer	使用 srTCM（RFC 2697）或 trTCM（RFC2698）算法进行数据包测量。
5	Load Balancer	将输入数据包分发给应用程序 worker。 为每个 worker 提供统一的负载。 保持流量对 worker 的亲和力和每个流程中的数据包顺序。
6	Worker threads	客户指定的应用工作负载的占位符（例如 IP 堆栈等）。
7	Dropper	拥塞管理使用随机早期检测（RED）算法（Sally Floyd-Van Jacobson 的论文）或加权 RED（WRED）。根据当前调度程序

		队列的负载级别和报文优先级丢弃报文。当遇到拥塞时，首先丢弃优先级较低的数据包。
8	Hierarchical Scheduler	具有数千（通常为 64K）叶节点（队列）的 5 级分层调度器（级别为：输出端口，子端口，管道，流量类和队列）。实现流量整形（用于子站和管道级），严格优先级（对于流量级别）和加权循环（WRR）（用于每个管道流量类中的队列）。

整个数据包处理流程中使用的基础架构块如下表所示。

#	Block	Functional Description
1	Buffer manager	支持全局缓冲池和专用的每线程缓存缓存。
2	Queue manager	支持水线之间的消息传递。
3	Power saving	在低活动期间支持节能。

水线块到 CPU cores 的映射可以根据每个特定应用程序所需的性能级别和为每个块启用的功能集进行配置。一些块可能会消耗多个 CPU cores（每个 CPU core 在不同的输入数据包上运行同一个块的不同实例），而另外的几个块可以映射到同一个 CPU core。

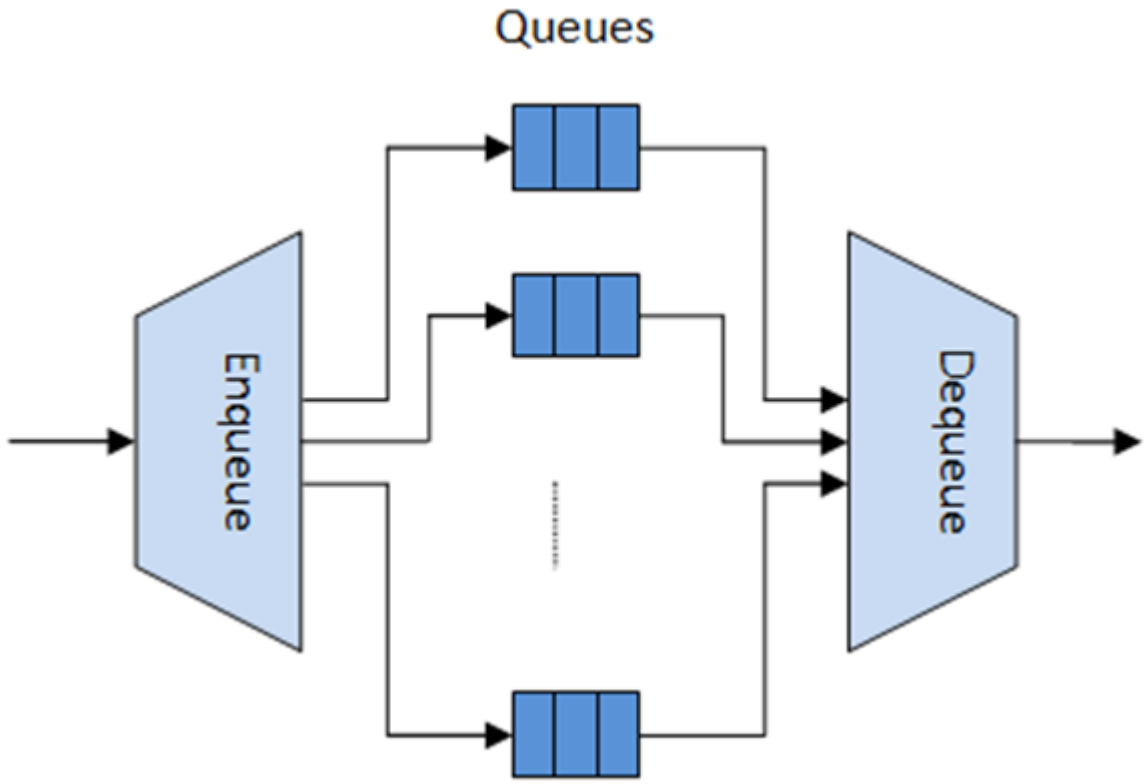
23.2.分层调度

分层调度块（当存在时）通常位于发送阶段之前的 TX 侧。其目的是根据每个网络节点的服务级别协议（SLA）指定的策略来实现不同用户和不同流量类别的数据包传输。

23.2.1. 概述

分层调度类似于网络处理器使用的流量管理，通常实现每个流（或每组流）分组排队和调度。它像缓冲区一样工作，能够在传输之前临时存储大量数据包（入队操作）；由于 NIC TX 正在请求更多的数据包进行传输，所以这些数据包随后被移出，并且随着分组选择逻辑观察预定义的 SLA（出队操作）而交给 NIC TX。

Figure 23-44 Hierarchical Scheduler Block Internal Diagram



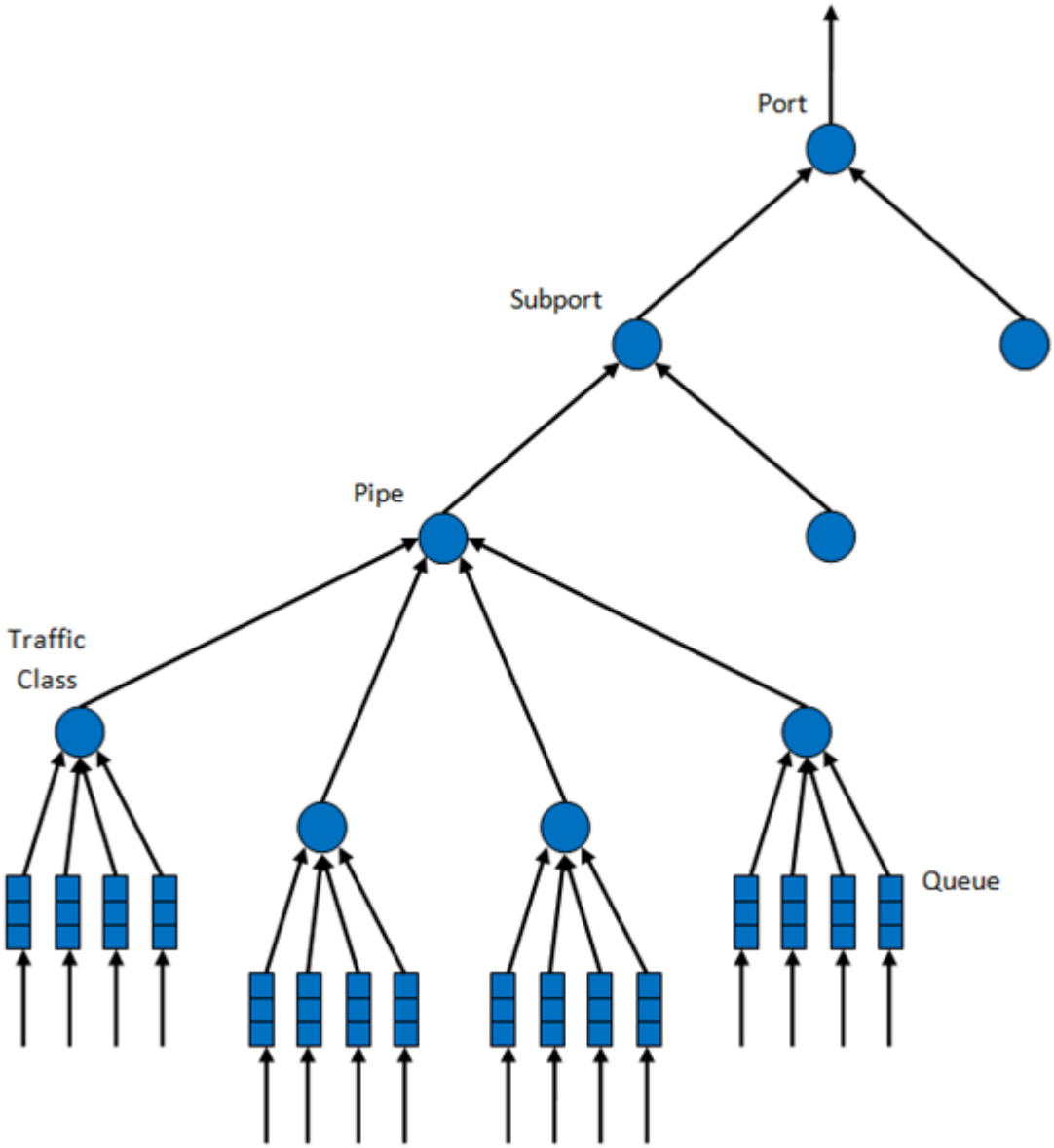
分层调度针对大量报文队列进行了优化。当只需要少量的队列时，应该使用消息传递队列而不是这个模块。有关更多详细的讨论，请参阅“Worst Case Scenarios for Performance”。

23.2.2. 调度层次

调度层次结构如下图所示。层次结构的第一级是以太网 TX 端口 1/10/40 GbE，后续层次级别定义为子端口，管道，流量类和队列。

通常，每个子端口表示预定义的用户组，而每个管道表示单个用户/订户。每个流量类是具有特定丢失率，延迟和抖动要求（例如语音，视频或数据传输）的不同流量类型的表示。每个队列都承载属于同一用户的同一类型的一个或多个连接的数据包。

Figure 23-45 Scheduling Hierarchy per Port



下表列出了各层次的功能。

#	Level	Siblings per Parent	Functional Description
1	Port		1. 输出以太网端口 1/10/40 GbE。 2. 多个端口以轮询方式调度，所有端口具有相同的优先级。
2	Subport	Configurable (default: 8)	1. 流量整形使用令牌桶算法（每个子口一个令牌桶）。 2. Subport 层对每个流量类（TC）强制执行上限。 3. 较低优先级的 TC 能够重用较高优先级的 TC 当前未使用的子端口带宽。

3	Pipe	Configurable (default: 4K)	使用令牌桶算法进行流量整形（每个 pipe 一个令牌桶）
4	TC	4	1. 相同 pipe 的 TC 以严格的优先级顺序处理。 2. 在 pipe 级别每 TC 执行上限。 3. 较低优先级的 TC 能够重用当前未被较高优先级的 TC 使用的 pipe 带宽。 4. 当子卡 TC 超额（配置时间事件）时，管道 TC 上限被限制为由所有子端口管道共享的动态调整值。
5	Queue	4	根据预定权重，使用加权循环（WRR）对相同 TC 的队列进行服务。

23.2.3. 编程接口

23.2.3.1. Port 调度配置 API

rte_sched.h 文件包含 port，subport 和 pipe 的配置功能。

23.2.3.2. Port 调度入队 API

Port 调度入队 API 非常类似于 DPDK PMD TX 功能的 API。

```
int rte_sched_port_enqueue(struct rte_sched_port *port, struct rte_mbuf **pkts, uint32_t n_pkts);
```

23.2.3.3. Port 调度出队 API

Port 调度入队 API 非常类似于 DPDK PMD RX 功能的 API。

```
int rte_sched_port_dequeue(struct rte_sched_port *port, struct rte_mbuf **pkts, uint32_t n_pkts);
```

23.2.3.4. 用例

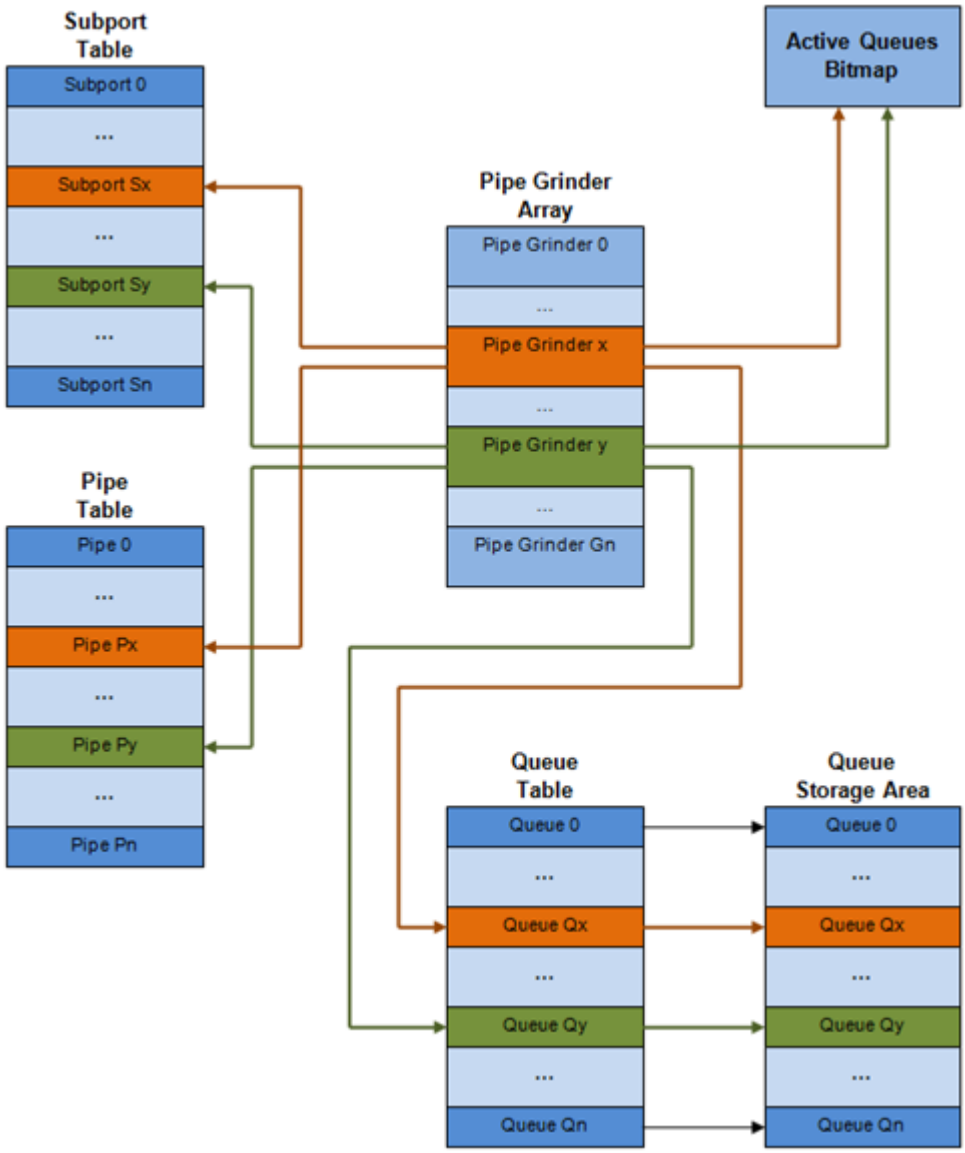
```
/* File "application.c" */

#define N_PKTS_RX   64
#define N_PKTS_TX   48
#define NIC_RX_PORT 0
#define NIC_RX_QUEUE 0
#define NIC_TX_PORT 1
#define NIC_TX_QUEUE 0

struct rte_sched_port *port = NULL;
struct rte_mbuf *pkts_rx[N_PKTS_RX], *pkts_tx[N_PKTS_TX];
uint32_t n_pkts_rx, n_pkts_tx;
```

```
/* Initialization */

<initialization code>

/* Runtime */
while (1) {
   /* Read packets from NIC RX queue */
   n_pkts_rx = rte_eth_rx_burst(NIC_RX_PORT, NIC_RX_QUEUE, pkts_rx, N_PKTS_RX);

   /* Hierarchical scheduler enqueue */
   rte_sched_port_enqueue(port, pkts_rx, n_pkts_rx);

   /* Hierarchical scheduler dequeue */
   n_pkts_tx = rte_sched_port_dequeue(port, pkts_tx, N_PKTS_TX);

   /* Write packets to NIC TX queue */
   rte_eth_tx_burst(NIC_TX_PORT, NIC_TX_QUEUE, pkts_tx, n_pkts_tx);
}
```

23.2.4. 实现

内部数据结构示意图，详细内容如下。

Figure 23-46 Internal Data Structures per Port



#	数据结构	大小	Per port	Access type		描述
				End	Deq	
1	子接口表条目	64	# subports per port		Rd, Wr	持续的子接口数据（信用，等）
2	Pipe 表条目	64	# pipes per port		Rd, Wr	在运行时更新的 pip，其 TC 及其队列的持久数据（信用等）。 pipe 配置参数在运行时不改变。 相同的 pipe 配置参数由多个 pipe 共享，因此它们不是 pipe 表条目的一部分。
3	队列条目	4	#queues per port	Rd, Wr	Rd, Wr	持续的队列数据（读写指针）。对于所有队列，每个 TC 的队列大小相

						同，允许使用快速公式计算队列基地址，因此这两个参数不是队列表条目的一部分。 任何给定 pipe 的队列表条目都存储在同一个高速缓存行中。
4	队列存储空间	Config (64x8)	# queues per port	Wr	Rd	每个队列的元素数组; 每个元素的大小是 8 字节（mbuf 指针）。
5	活动队列位图	1 bit per queue	1	Wr(set)	Rd, Wr (Clear)	位图为每个队列维护一个状态位：队列不活动（队列为空）或队列活动（队列不为空）。 队列位由调度程序入队设置，并在队列变空时由调度程序清除。 位图扫描操作返回下一个非空 pipe 及其状态（pipe 中活动队列的 16 位掩码）。
6	Grinder	~128	Config (default: 8)		Rd, Wr	目前正在处理的活动 pipe 列表。grinder 在 pipe 加工过程中包含临时数据。 一旦当前 pipe 排出数据包或信用点，它将从位图中的另一个活动管道替换。

23.2.4.1. 多核缩放策略

多核缩放策略如下：

1. 在不同线程上操作不同的物理端口。但是同一个端口的入队和出队由同一个线程执行。
2. 通过在不同线程上操作相同物理端口（虚拟端口）的不同组的子端口，可以将相同的物理端口拆分到不同的线程。类似地，子端口可以被分割成更多个子端口，每个子端口由不同的线程运行。但是同一个端口的入队和出队由同一个线程运行。仅当出于性能考虑，不可能使用单个 core 处理完整端口时，才这样处理。

23.2.4.1.1. 同一输出端口的出队和入队

上面强调过，同一个端口的出队和入队需要由同一个线程执行。因为，在不同 core 上对同一个输出端口执行出队和入队操作，可能会对调度程序的性能造成重大影响，因此不推荐这样做。

同一端口的入队和出队操作共享以下数据结构的访问权限：

1. 报文描述符
2. 队列表
3. 队列存储空区
4. 活动队列位图

可能存在使性能下降的原因如下：

1. 需要使队列和位图操作线程安全，这可能需要使用锁以保证访问顺序（例如，自旋锁/信号量）或使用原子操作进行无锁访问（例如，Test and Set 或 Compare and Swap 命令等））。前一种情况对性能影响要严重得多。
2. 在两个 core 之间对存储共享数据结构的缓存行执行乒乓操作（由 MESI 协议缓存一致性 CPU 硬件透明地完成）。

当调度程序入队和出队操作必须在同一个线程运行，允许队列和位图操作非线程安全，并将调度程序数据结构保持在同一个 core 上，可以很大程度上保证性能。

23.2.4.2. 性能缩放

扩展 NIC 端口数量只需要保证用于流量调度的 CPU 内核数量按比例增加即可。

23.2.4.3. 入队水线

每个数据包的入队步骤：
1. 访问 mbuf 以读取标识数据包的目标队列所需的字段。这些字段包括 port，subport，traffic class 及 queue，并且通常由报文分类阶段设置。
2. 访问队列结构以识别队列数组中的写入位置。如果队列已满，则丢弃该数据包。
3. 访问队列阵列位置以存储数据包（即写入 mbuf 指针）。

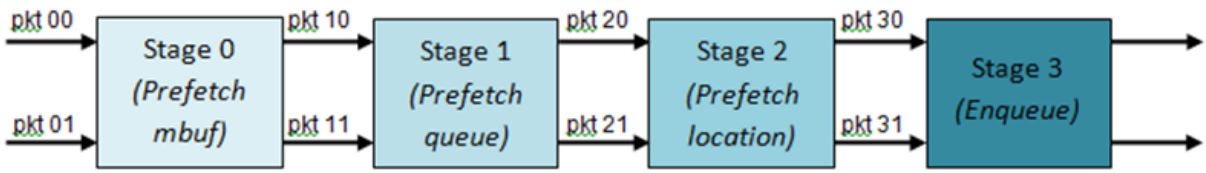
应该注意到这些步骤之间具有很强的数据依赖性，因为步骤 2 和 3 在步骤 1 和 2 的结果变得可用之前无法启动，这样就无法使用处理器乱序执行引擎上提供任何显着的性能优化。

考虑这样一种情况，给定的输入报文速率很高，队列数量大，可以想象得到，入队当前数据包需要访问的数据结构不存在于当前 core 的 L1 或 L2 data cache 中，此时，上述 3 个内存访问操作将会产生 L1 和 L2 data cache miss。就性能考虑而言，每个数据包出现 3 次 L1 / L2 data cache miss 是不可接受的。

解决方法是提前预取所需的数据结构。预取操作具有执行延迟，在此期间处理器不应尝试访问当前正在进行预取的数据结构，此时处理器转向执行其他工作。可用的其他工作可以是对其他输入报文执行不同阶段的入队序列，从而实现入队操作的流水线实现。

下图展示出了具有 4 级水线的入队操作实现，并且每个阶段操作 2 个不同的输入报文。在给定的时间点上，任何报文只能在水线某个阶段进行处理。

Figure 23-47 Prefetch Pipeline for the Hierarchical Scheduler Enqueue Operation



由上图描述的入队水线实现的拥塞管理方案是非常基础的：数据包排队入队，直到指定队列变满为

止；当满时，到这个队列的所有数据包将被丢弃，直到队列中有数据包出队。可以通过使用 RED/WRED 作为入队水线的一部分来改进，该流程查看队列占用率和报文优先级，以便产生特定数据包的入队/丢弃决定（与入队所有数据包/不加区分地丢弃所有数据包不一样）。

23.2.4.4. 出队状态机

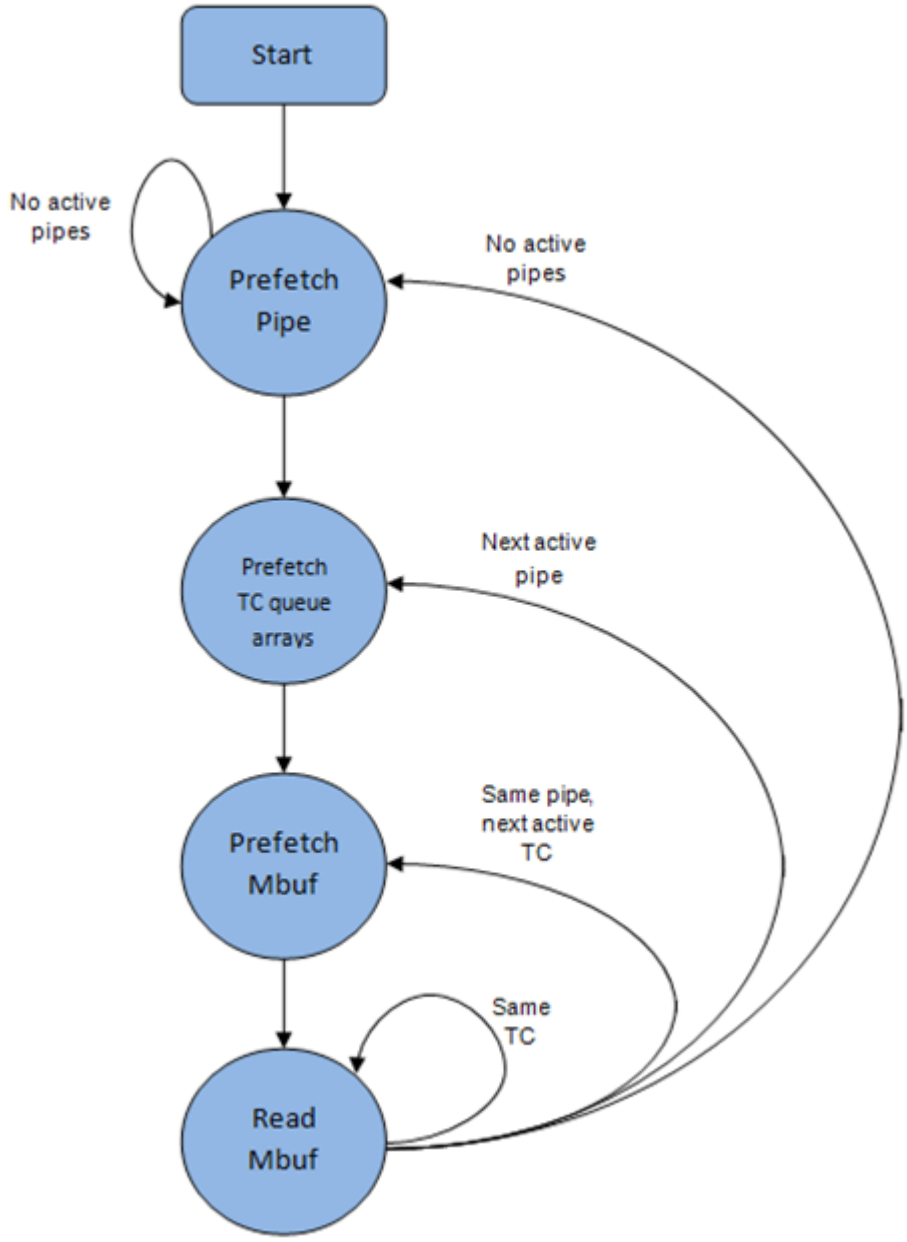
从当前 pipe 调度下一个数据包的步骤如下：

1. 使用位图扫描操作识别出下一个活动的 pipe（prefetch pipe）。
2. 读取 pipe 数据结构。更新当前 pipe 及其 subport 的信用。识别当前 pipe 中的第一个 active traffic class，使用 WRR 选择下一个 queue，为当前 pipe 的所有 16 个 queue 预取队列指针。
3. 从当前 WRR queue 读取下一个元素，并预取其数据包描述符。
4. 从包描述符（mbuf 结构）读取包长度。根据包长度和可用信用（当前 pipe，pipe traffic class，subport 及 subport traffic class），对当前数据包进行是否调度决策。

为了避免 cache miss，上述数据结构（pipe，queue，queue array，mbufs）在被访问之前被预取。隐藏预取操作的延迟的策略是在为当前 pipe 发出预取后立即从当前 pipe（在 grinder A 中）切换到另一个 pipe（在 grinderB 中）。这样就可以在执行切换回 pipe（grinder A）之前，有足够的时间完成预取操作。

出 pipe 状态机将数据存在处理器高速缓存中，因此它尝试从相同的 pipe TC 和 pipe（尽可能多的数据包和信用）发送尽可能多的数据包，然后再移动到下一个活动 TC pipe（如果有）或另一个活动 pipe。

Figure 23-48 Pipe Prefetch State Machine for the Hierarchical Scheduler Dequeue Operation



23.2.4.5. 时间和同步

输出端口被建模为字节槽的传送带，需要由调度器填充用于传输的数据。对于 10GbE，每秒需要由调度器填充 12.5 亿个字节的槽位。如果调度程序填充不够快，只要存在足够的报文和信用，则一些时隙将被闲置并且带宽将被浪费。

原则上，层次调度程序出队操作应由 NIC TX 触发。通常，一旦 NIC TX 输入队列的占用率下降到预定义的阈值以下，端口调度器将被唤醒（基于中断或基于轮询，通过连续监视队列占用）来填充更多的数据包进入队列。

23.2.4.5.1. 内部时间引用

调度器需要跟踪信用逻辑的时间演化，因为信用需要基于时间更新（例如，子流量和管道流量整形，流量级上限执行等）。

每当调度程序决定将数据包发送到 NIC TX 进行传输时，调度器将相应地增加其内部时间参考。因此，以字节为单位保持内部时间基准是方便的，其中字节表示物理接口在传输介质上发送字节所需的持续时间。这样，当报文被调度用于传输时，时间以（n + h）递增，其中 n 是以字节为单位的报文长度，h 是每个报文的成帧开销字节数。

23.2.4.5.2. 内部时间参考重新同步

调度器需要将其内部时间参考对齐到端口传送带的步速。原因是要确保调度程序不以比物理介质的线路速率更多的字节来馈送 NIC TX，以防止数据包丢失。

调度程序读取每个出队调用的当前时间。可以通过读取时间戳计数器（TSC）寄存器或高精度事件定时器（HPET）寄存器来获取 CPU 时间戳。 当前 CPU 时间戳将 CPU 时钟数转换为字节数：time_bytes = time_cycles / cycles_per_byte，其中 cycles_per_byte 是等效于线上一个字节的传输时间的 CPU 周期数（例如 CPU 频率 2 GHz 和 10GbE 端口，* cycles_per_byte = 1.6 *）。

调度程序维护 NIC time 的内部时间参考。 每当分组被调度时，NIC time 随分组长度（包括帧开销）增加。在每次出队调用时，调度程序将检查其 NIC time 的内部引用与当前时间的关系：

1. 如果 NIC time 未来（NIC time> =当前时间），则不需要调整 NIC time。这意味着调度程序能够在 NIC 实际需要这些数据包之前安排 NIC 数据包，因此 NIC TX 提供了数据包;
2. 如果 NIC time 过去（NIC 时间<当前时间），则 NIC time 应通过将其设置为当前时间来进行调整。 这意味着调度程序不能跟上 NIC 字节传送带的速度，因此由于 NIC TX 的数据包供应不足，所以 NIC 带宽被浪费了。

23.2.4.5.3. 调度器精度和粒度

调度器往返延迟（SRTD）是指调度器在同一个 pipe 的两次连续检验之间的时间（CPU 周期数）。

为了跟上输出端口（即避免带宽丢失），调度程序应该能够比 NIC TX 发送的 n 个数据包更快地调度 n 个数据包。

假设没有端口超过流量，调度程序需要跟上管道令牌桶配置的每个管道的速率。这意味着管道令牌桶的大小应该设置得足够高，以防止它由于大的 SRTD 而溢出，因为这将导致管道的信用损失（带宽损失）。

23.2.4.6. 信用逻辑

23.2.4.6.1. 调度决策

当满足以下所有条件时，从（subport S，pipe P，traffic class TC，queue Q）发送下一个分组的调度决定（分组被发送）：

- Subport S 的 Pipe P 目前由一个端口调度选择；
- 流量类 TC 是管道 P 的最高优先级的主要流量类别；
- 队列 Q 是管道 P 的流量类 TC 内由 WRR 选择的下一个队列；
- 子接口 S 有足够的信用来发送数据包；
- 子接口 S 具有足够的信用流量类 TC 来发送数据包；
- 管道 P 有足够的信用来发送数据包；
- 管道 P 具有足够的信用用于流量类 TC 发送数据包。

如果满足所有上述条件，则选择分组进行传输，并从子接口 S，子接口 S 流量类 TC，管道 P，管道 P 流量类 TC 中减去必要的信用。

23.2.4.6.2. 帧开销

由于所有数据包长度的最大公约数为 1 个字节，所以信用单位被选为 1 个字节。传输 n 个字节的报文所需的信用数量等于（n + h），其中 h 等于每个报文的成帧开销字节数。

#	Packet field	Length	Comments
1	前导码	7	
2	帧开始分隔符	1	
3	帧校验序列	4	当 mbuf 包长度字段中不包含时这里才需要考虑开销。
4	帧间隙	12	
5	总	24	

23.2.4.6.3. 流量整形

Subport 和 pipe 的流量整形使用每个 subport/pipe 的令牌桶来实现。令牌桶使用一个饱和计数器实现，该计数器跟踪可用信用数量。

令牌桶通用参数和操作如下表所示。

#	Token Bucket Parameter	Unit	Description
1	Bucket_rate	每秒信用值	每秒钟添加到桶中的信用
2	Bucket_size	信用值	桶最多可以存储多少信用

#	Token Bucket Operation	Description
1	Initilization	桶设置为预定义值，例如 零或一半的桶大小。
2	Credit Update	基于 bucket_rate，信用值将根据现有的信息添加到桶中，定期添加或按需添加。 信用值不能超过 bucket_size 定义的上限，因此在存储桶已满时，任何要添加到存储桶中的信用额将被丢弃。
3	Credit Consumption	作为分组调度的结果，从桶中移除必要的信用。 只有在桶中有足够的信用来发送完整数据包（数据包

		的数据包字节和帧开销）时，才能发送数据包。

为了实现上述的令牌桶通用操作，当前的设计使用表 23.8 中所示的数据结构，而令牌桶操作的实现在表 23.9 中描述。

#	Token Bucket Field	Unit	Description
1	tb_time	Bytes	最后一次更新信用的时间。以字节为单位而不是秒或 CPU 周期进行测量，便于信用消耗操作（因为当前时间也以字节为单位）。 有关为什么以字节为单位维护时间的说明，请参见第 26.2.4.5.1 节“内部时间参考”。
2	tb_period	Bytes	自上一次信用更新以来应该经过的时间段，以便桶被授予 tb_credits_per_period 价值或信用。
3	tb_credits_per_period	Bytes	每 tb_period 的信用值。
4	tb_size	Bytes	桶大小，即 tb_credits 的上限。
5	tb_credits	Bytes	当前在桶中的数值。

桶速率（以字节为单位）可以用以下公式计算：

bucket_rate = (tb_credits_per_period / tb_period) * r

其中 r =端口线路速率（以字节为单位）。

#	Token Bucket Operation	Description
1	Initilization	tb_credits = 0; or tb_credits = tb_size / 2;
2	Credit Update	信用更新操作： ● 每次为端口发送数据包时，请更新该端口的所有子端口和管道的信用。不可行。 ● 每次发送数据包时，请更新管道和子接口的信用。 非常准确，但不需要（过多计算）。 ● 每次选择管道（即由其中一个 grinders 选择）时，请更新管道及其子接口的积分。 当前的实现使用选项 3。 根据“出队状态机”小节描述，管道和子接口信用在管道和子接口信用实际使用之前，每出现一次出口流程选择管道时，都将更新。 只有在自上次更新以来已经过了至少一个完整的 tb_period 时，才能通过更新桶级别来实现精度和速度之间的权衡。 ● 可以通过选择 tb_credits_per_period = 1 的 tb_period 值来实现全精度。 ● 当不需要全精度时，通过将 tb_credits 设置为更大的值可以获得更好的性能。

		更新操作： ● n_periods = (time - tb_time) / tb_period; ● tb_credits += n_periods * tb_credits_per_period; ● tb_credits = min(tb_credits, tb_size); ● tb_time += n_periods * tb_period;

3	Credit Consumption
	（报文调度）
		作为分组调度的结果，从桶中移除必要的信用。
		只有在桶中有足够的信用来发送完整数据包（数据包的数据包字节和帧开销）时，才能发送数据包。
		
		调度操作：
		pkt_credits = pkt_len + frame_overhead;
		if（tb_credits> = pkt_credits）{
		tb_credits - = pkt_credits;
		}

23.2.4.6.4. 流量类