1 SPDK 存储轻量化技术

SPDK（Storage Performance Development Kit，存储性能开发工具包）[1–3]提供了一组用于编写高性能、可扩展、用户态的存储应用程序的工具和库。它移动驱动程序（例如 NVMe SSD 驱动）到用户空间，以轮询模式运行而不是中断，避免内核上下文切换并消除中断处理开销，实现了内核旁路（bypass kernel），是一种存储轻量化技术。

把内核驱动放到用户态以后，需要实现一套用户空间的软件 I/O 栈。基于内核的文件系统不能使用了。SPDK 提供了简单的文件系统 blobfs，但是它不支持 POSIX 接口，因此，使用 SPDK 需要做一些代码移植工作。

SPDK 的实际应用场景[4]：

- 提供块设备结构的存储应用后端，例如 iSCSI Target、NVMe-oF Target
- 加速虚拟机中 I/O，例如 Linux 系统下 QEMU/KVM 作为 Hypervisor 管理虚拟机的场景
- 加速数据库存储引擎，例如 RocksDB

1.1前置知识与名词解释

1.1.1 NVMe、PCIe、AHCI

NVMe 即 NVM Express，又称 NVMHCIS。NVM 代表非易失性存储器（non-volatile memory），是固态硬盘（SSD）常见的闪存形式。NVMe 是一个逻辑设备接口规范，用于访问通过 PCIe（PCI Express）总线附加的 NVM 介质。

NVMe 是一种协议，PCIe 是实际的物理连接通道。

在 NVMe 之前，有一个更古老的协议 AHCI（Advanced Host Controller Interface），允许软件与 SATA 存储设备通信。但是随着存储介质的发展，AHCI 逐渐成为了闪存 SSD 的性能瓶颈，逐渐被 NVMe 取代了。

1.1.2 SCSI、SAN、iSCSI

SCSI 即 Small Computer System Interface，小型计算机系统接口。它规范了一种并行的 I/O 总线和相关的协议，SCSI 的数据传输是以块的方式进行的。

SAN 即 Storage Area Network，是一种连接外接存储设备和服务器的架构。连接到服务器的存储设备，将被操作系统视为直接连接的存储设备。传输的对象

是数据块。使用光纤交换机的 SAN 方案称为 FC-SAN，使用现有 IP 网络的 SAN 方案称为 IP-SAN。

iSCSI 将原来只用于本机的 SCSI 协议通过网络发送，是 IP-SAN 的热门实现方式。

1.1.3 NVMe-oF

NVMe-oF（NVMe over Fabrics）扩展了 NVMe 规范在 PCIe 总线上的实现，把 NVMe 映射到多种物理网络传输通道，实现高性能的存储设备网络共享访问。NVMe-oF 定义使用了多种通用的传输层协议来实现 NVMe 功能，支持把 NVMe 映射到多个 Fabrics 传输选项，包括 FC、InfiniBand、RoCE-v2、iWARP 和 TCP。

NVMe-oF 中的客户端称为 initiator，服务端称为 target。iSCSI 也是类似的表述。

1.1.4 用户态 I/O

在用户态实现驱动，就要有用户态 I/O 的手段。用户态 I/O 有 UIO 和 VFIO 两种方法，这两种方法 SPDK 都支持。基于用户态 I/O，我们可以进行用户态 DMA。

用户态 DMA 有几个要点：通过 IOMMU，内存地址可以让设备认知；设备和 CPU 之间的缓存一致性让 CPU 对内存的更新对设备可见；人工 Pin 内存的方式确保了物理内存在位；通过大页降低了缺页的 CPU 开销。

UIO

UIO 框架最早于 Linux 2.6.32 版本引入，提供了在用户态实现设备驱动的可能性。要在用户态实现设备驱动，主要需要解决以下两个问题：

如何访问设备的内存？Linux 通过映射物理设备的内存到用户态来提供访问，但是这种方法会引入安全性和可靠性的问题。UIO 通过限制不相关的物理设备的映射改善了这个问题。由此基于 UIO 开发的用户态驱动不需要关心与内存映射相关的安全性和可靠性的问题。

如何处理设备产生的中断？中断本身需要在内核处理，因此针对这个限制，还需要一个小的内核模块通过最基本的中断服务程序来处理。这个中断服务程序可以只是向操作系统确认中断，或者关闭中断等最基础的操作，剩下的具体操作可以在用户态处理。UIO 架构如图 1-1 所示，用户态驱动和 UIO 内核模块通过

/dev/uioX 设备来实现基本交互，同时通过 sysfs 来得到相关的设备、内存映射、内核驱动等信息。

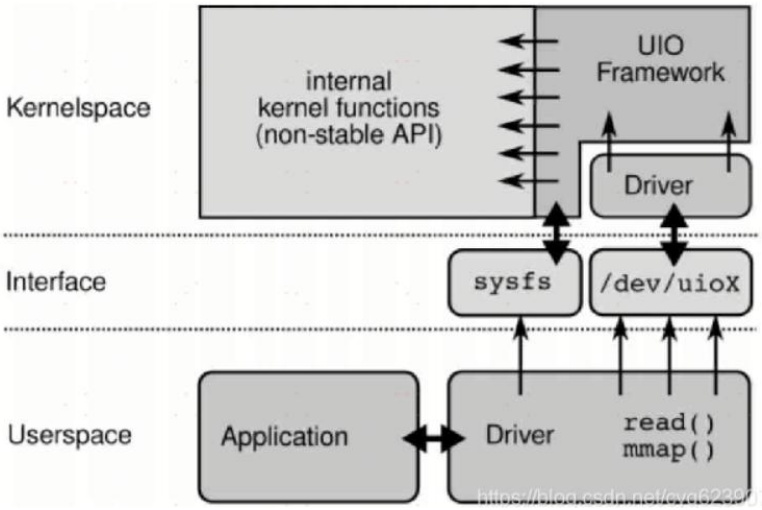


图 1-1 UIO 架构

VFIO

相对于 UIO，VFIO 不仅提供了 UIO 所能提供的两个最基础的功能，更多的是从安全角度考虑，把设备 I/O、中断、DMA 暴露到用户空间，从而可以在用户空间完成设备驱动的框架。这里的一个难点是如何将 DMA 以安全可控的方式暴露到用户空间，防止设备通过写内存的任意页来发动 DMA 攻击。

IOMMU（I/O Memory Management Unit）的引入对设备进行了限制，设备 I/O 地址需要经过 IOMMU 重映射为内存物理地址。那么恶意的或存在错误的设备就不能读/写没有被明确映射过的内存。操作系统以互斥的方式管理 MMU 和 IOMMU，如图 1-2 所示，这样物理设备将不能绕过或污染可配置的内存管理表项。

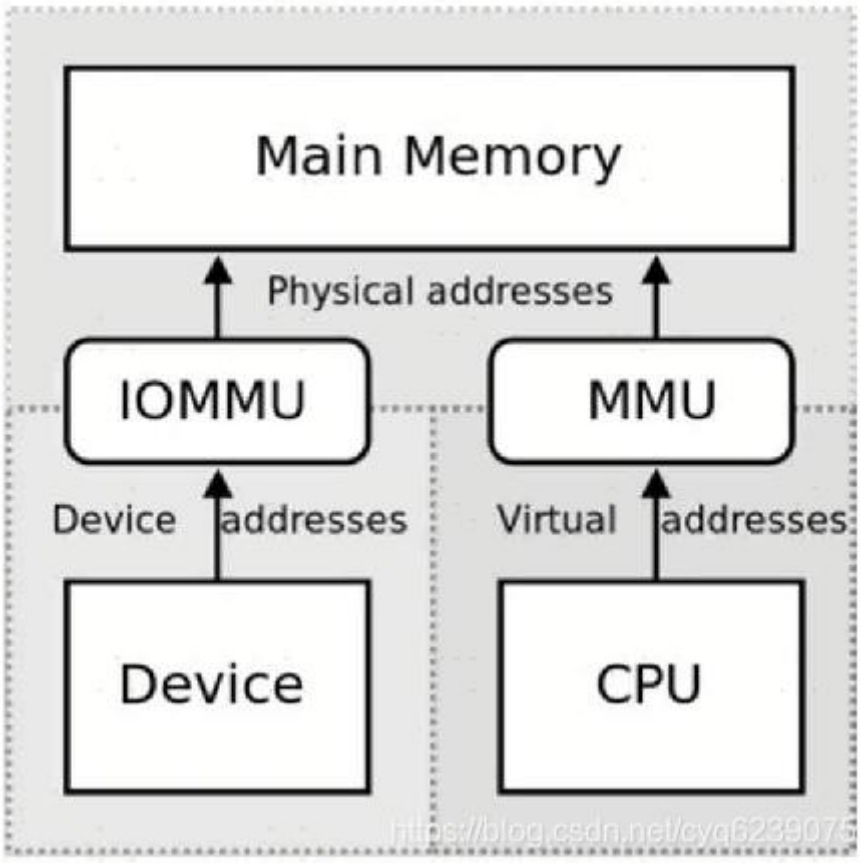


图 1-2 加上 IOMMU 后的地址映射

解决了通过安全可控的方式暴露 DMA 到用户空间这个问题后，用户基本上就可以实现使用用户态驱动操作硬件设备了。考虑到 NVMe SSD 是一个通用的 PCI 设备，VFIO 的 PCI 设备实现层（vfio-pci 模块）提供了和普通设备驱动类似的作用，可高效地穿过内核若干抽象层，在/dev/vfio 目录下为设备所在的 IOMMU group 生成相关文件，继而将设备暴露出来。

1.2背景

现在，云计算高速发展，云存储服务是云计算的重要组成部分。用户往既希望云存储服务既能够提供高吞吐和低延迟，又不至于消耗过多的资源。而存储服务的质量和代价是由软件和硬件共同决定的。在存储设备速度较低的年代，软件的开销和硬件比起来不值一提，所以存储服务的性能主要是由硬件决定的。但是快速存储设备，例如 NVMe SSD 出现以后，硬件执行的单次 I/O 操作延迟降低到了几毫秒的级别。现在，基于 3D Xpoint 技术的存储介质，例如 Intel Optane SSD[5]，它们的延迟甚至已经接近了 DRAM 的延迟，如图 1-3 所示。软件的开销则变成了改进存储服务性能的瓶颈。

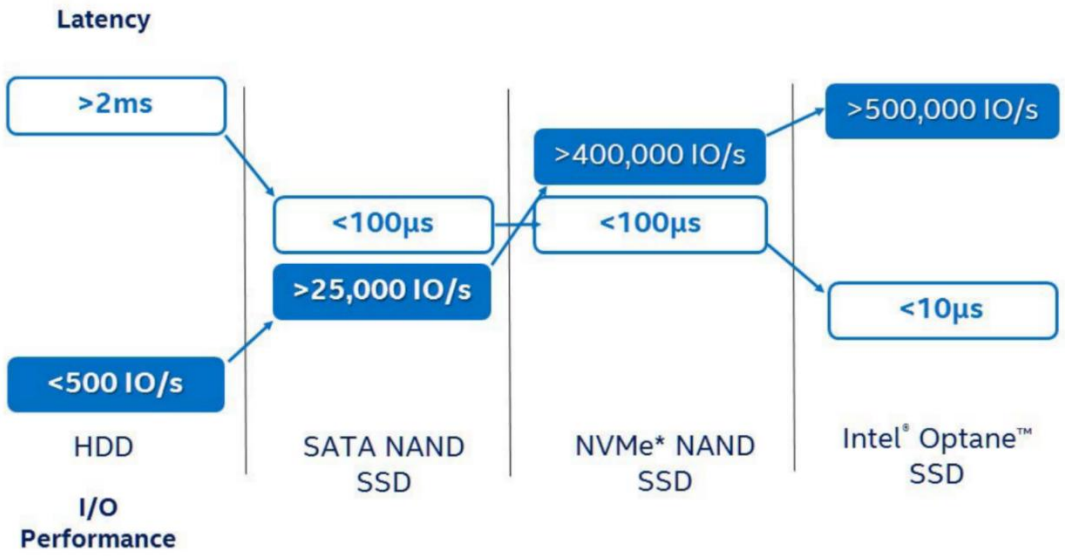


图 1-3 存储介质的 I/O 性能和延迟

在存储服务的耗时中，内核软件 I/O 栈耗时占据了一大部分，包括上下文切换、内核空间用户空间之间的数据复制、中断、I/O 栈的共享资源争用等。为了解决这种问题，Intel 提出了 SPDK 技术。

1.3SPDK 系统设计和实现

SPDK 提供了一系列设计、实现高性能可伸缩的存储应用的库和工具，如图 1-4 所示。目前，SPDK 提供了用户空间的更高性能的 NVMe 驱动和 I/OAT 驱动，也提供了一些定制化的更高性能的存储应用，例如 SPDK NVMe-oF target/initiator。

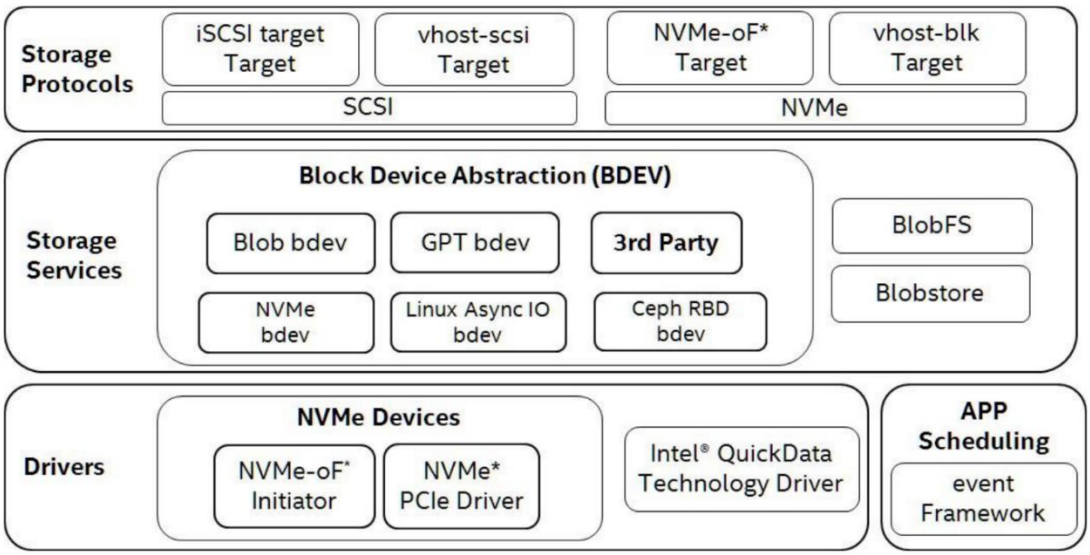


图 1-4 SPDK 组件

SPDK 主要包括四个组件：

- APP scheduling（应用程序框架），基于事件（event）编写异步、轮询、无共享的应用
- Drivers（驱动），提供给应用程序用户空间轮询模式的、零拷贝、高并行、直接访问的 NVMe 驱动、I/OAT DMA engine 驱动
- Storage Services（存储服务），基于驱动和应用程序框架，抽象化驱动程序等导出的设备，提供 bdev、blobfs、blobstore 给上层应用
- Storage protocols（存储协议），基于存储服务提供的接口，包含在 SPDK 框架上实现的加速应用程序，以支持各种不同的存储协议

1.3.1 APP scheduling

APP scheduling 是 SPDK 提供的异步、轮询、无共享的应用框架。这个框架是可选的，结构如图 1-5 所示。

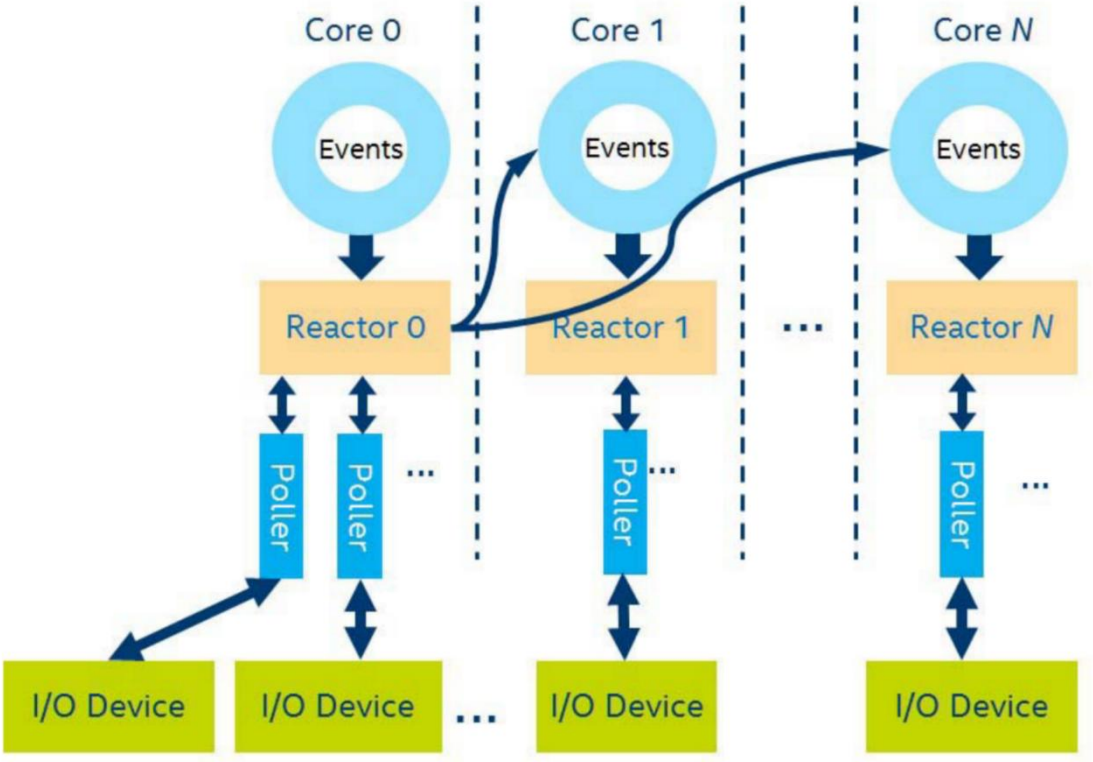


图 1-5 APP scheduling 结构

本框架定义了几个概念：events reactors pollers。本框架在每个 CPU core 都对应一个线程（即 reactor），消息在线程之间以 event 的形式通过无锁队列异步地传递。（在现代 CPU 中，消息传递往往比传统的锁更快。）

Events

Event 包含了绑定的函数指针及其参数。event 通过 `spdk_event_allocate()` 创建，通过 `spdk_event_call()` 执行。不像依赖操作系统在有限的 core 上调度进程、执行阻塞式 I/O 的 thread-per-connection 设计，本事件驱动的模型使用异步 I/O，I/O 完毕以后使用回调函数。

Reactors

每个 core 上运行着一个线程，就是 reactor。它本质上是一个线程加上一个无锁队列，线程上运行着 event loop，主要职责就是先进先出地执行来自队列的 event。来自任何 core 的线程都可以将 event 插入到任何其他 core 的队列中。Event 中的函数应当非阻塞并且执行得很快，因为它们直接从 reactor 线程中的 event loop 执行，不然就阻塞后面的 event 了。

Pollers

Poller 是框架定义的另一种函数，旨在轮询硬件以代替中断，可以用 `spdk_poller_register()` 注册。Poller 类似于 event 的一点是可以绑定上函数和参数，发送给特定的 core 执行。但是 poller 会一直执行直到被注销。Reactor 将 poller 与其他的 event 交错执行，不过如果不需要低延迟，poller 也可以使用定时器周期地执行。

1.3.2 Drivers

驱动最经典的实现方式是内核态驱动。当内核驱动模块在内核加载成功后，会被标识成块设备或者字符设备，同时定义相关的访问接口，包括管理接口、数据接口等。这些接口直接或间接和文件系统子系统结合，提供给用户态的应用程序，应用程序通过系统调用的方式发起控制和读/写操作。

SPDK 将设备驱动实现放在用户态，好处是可以消除大量系统调用开销、用户空间和内核空间之间的数据拷贝，相对于内核态可以以更少的时钟周期完成 I/O 操作。

SPDK 使用轮询代替中断，节省了调用内核中断处理程序的开销，减少了上下文切换[6]。

异步轮询模式

UIO 和 VFIO 需要在内核实现最基本的中断功能来响应设备的中断请求，SPDK 更进一步，这些中断请求不需要通知到用户态来处理，在 UIO 和 VFIO 内核模块做最简化的处理就可以了。SPDK 采用异步轮询的方式摆脱了对中断的依赖。

应用可以调用 SPDK 的异步读写接口来发送 I/O 请求，随即可以使用相应的检查 IO 完成情况的函数轮询来获得 I/O 情况。总的来说，比阻塞 I/O 延迟低、IOPS 高了。

无锁结构

传统的内核 I/O 栈在进行 I/O 的时候会使用一些共享资源，例如内核 NVMe 驱动会在多个 CPU core 之间产生对 NVMe I/O 队列的争用。为了消除资源共享的开销，SPDK 使用了无锁结构，一是要求读/写处理要在一个 CPU core 上完成，避免 core 间的缓存同步；二是要求单核上的处理，对资源的分配是无锁化的。

第一个问题，可以通过线程亲和性的方法，来将某个处理线程绑定到某个特定的核上，同时通过轮询的方式占住该核的使用，避免操作系统调度其他的线程到该核上面。当应用程序接收到这个核上的读/写请求的时候，采用运行直到完成（Run To Completion）的方式，把这个读/写请求的整个生命周期都绑定在这个核上来完成。

第二个问题，在处理该核上的读/写请求时，需要分配相关的资源，如 Buffer。这些 Buffer 主要通过大页分配而来。DPDK 为 SPDK 提供了基础的内存管理，单核上的资源依赖于 DPDK 的内存管理，不仅提供了核上的专门资源，还提供了高效访问全局资源的数据结构，如 mempool、无锁队列、环等。

用户空间的 NVMe 驱动示例

SPDK 实现了一个用户空间的异步轮询 NVMe 驱动库，有一些表 1-1 所示的 API。内核驱动可能有更好的兼容性、维护性，但是 SPDK 提供的驱动专注于发挥高速 NVMe SSD 的性能。使用它只需要卸载 Linux 内核的 NVMe 驱动并绑定特定的 NVMe 设备到 Linux 内核中的 UIO 或者 VFIO 驱动。这个库直接将 PCI BAR（Base Address Register）映射到本地进程并且执行 MMIO 来控制 NVMe 设备。

表 1-1 SPDK 用户空间 NVMe 驱动 API

Key Functions	Decriptions
spdk_nvme_probe()	Attach the userspace NVMe driver to found NVMe device.
spdk_nvme_ctrlr_alloc_io_qpair()	Allocate an I/O queue pair (submission and completion queue).
spdk_nvme_ctrlr_get_ns()	Get a handle to a namespace for the given controller.
spdk_nvme_ns_cmd_read()	Submit a read I/O to the specified NVMe namespace.
spdk_nvme_ns_cmd_write()	Submit a data set management request to the specified NVMe namespace.
spdk_nvme_ns_cmd_dataset_management()	Submit a data set management request to the specified NVMe namespace.
spdk_nvme_ns_cmd_flush()	Submit a flush request to the specified NVMe namespace.
spdk_nvme_qpair_process_completions()	Process any outstanding completions for I/O submitted on a queue pair.
spdk_nvme_ctrlr_cmd_admin_raw()	Send the given admin command to the NVMe controller.
spdk_nvme_ctrlr_process_admin_completions()	Process any outstanding completions for admin commands.

NVMe 驱动在指定的队列对（queue pairs，`struct spdk_nvme_qpair`）上将 I/O 请求作为 NVMe 提交队列的条目（entry）提交。I/O 通过 `nvme_ns_cmd_xxx_()` 系列函数异步地提交，也就是说函数在命令执行完毕前就立刻返回了。应用要轮询每个具有未完成的 I/O 的 queue pair，查询它们的 I/O 完成情况，调用 `spdk_nvme_qpair_process_completions()` 来接收完成后的回调函数。

I/O 可能在不同的线程往多个 queue pair 上提交。因为 SPDK 的无锁结构，在任何时刻一个队列对只能由一个线程使用。程序员违反了这个规定会导致未定义行为。

大多数 NVMe 硬盘支持的 queue pair 的数量在 32 到 128 之间，不过一个 queue pair 往往就能发挥全部性能。例如，四个 QD（queue depth）为 32 的 queue pair 和一个 QD 为 128 的 queue pair，它们的性能是差不多的。因此，将固定数量的线程分别固定在不同的 CPU core 上，每个线程分配一个 queue pair 即可达到较高性能。

正是因为无锁结构，NVMe driver 才能达到高性能和可扩展性。出于同样的原因，应用也要减少线程之间的争用。

1.3.3 Storage services

Storage services 包含用户空间块设备 bdev、bdev 之上的 blob 管理层 blobstore、基于 blobstore 的简单的文件系统 blobfs。

BDEV

SPDK bdev 抽象化用户空间 NVMe 驱动程序、Linux 异步 I/O（libaio）、Ceph rados 块设备 API 等导出的设备，提供用户空间块接口。类似于内核的通用块设备层。

Blobstore

因为基于 blobstore 的 blobfs 不是为了成为通用文件系统而被设计的，并且 blobfs 不符合 POSIX 接口，所以用了 blob 一词来表达类似于传统文件系统的文件的概念。Blob 通常较大，往往至少数百 KB，大小是底层块大小的倍数。Blobstore 分配和管理 blob，在一个名为 blobs 的块设备上对一组块进行异步、非缓存、并行的读写。

Blobfs

Blobfs 是一个基于 blobstore 的简单的轻量级文件系统，提供了部分对于文件操作的接口，把对文件的操作转换为对 blob 的操作。Blobfs 中的文件与 Blobstore 中的 Blob 一一对应。Blobfs 意在支持更上层的服务，例如 MySQL、Rocksdb 等。

通过实现 RocksDB 中的抽象文件类，SPDK 的 blobfs/blobstore 可以和 RocksDB 集成，加速在 NVMe SSD 上的 RocksDB 引擎。其实质是 bypass kernel 文件系统，完全使用基于 SPDK 的用户态 I/O 栈。此外，参照 SPDK 对 RocksDB 的支持，亦可以用 SPDK 的 blobfs/blobstore 整合其他的数据库存储引擎。

在 service 层编程

（1）Blobstore 层次概念

Blobstore 层次结构如图 1-6 所示。

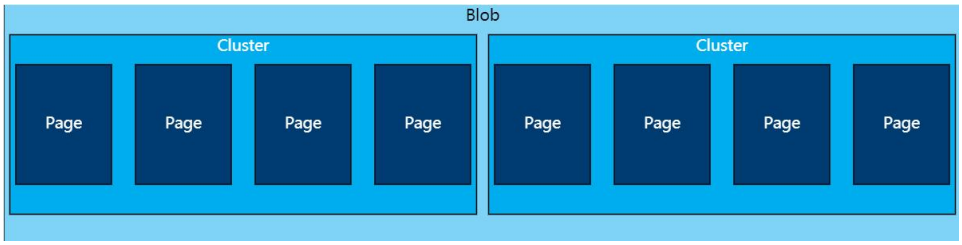


图 1-6 Blobstore 层次结构图

逻辑块：逻辑块由磁盘本身公开，磁盘编号从 0 到 N，其中 N 是磁盘中的块数。逻辑块通常是 512B 或 4KiB。

页面：页面定义为在 blobstore 创建时定义的固定数量的逻辑块。组成页面的逻辑块始终是连续的。页面大小通常是 4Kib。

Cluster 群集：群集是在 blobstore 创建时定义的固定页数。组成群集的页面始终是连续的。群集也从磁盘的开头编号，其中群集 0 是第一组群集页面，群集 1 是第二组页面等。群集通常是 1MiB 大小，或 256 页。

Blob：blob 是一个有序的 Cluster 列表。应用程序主要操作对象，与 blobfs 中的文件相对应。应用程序使用 blobstore 提供的标识符来访问特定 blob。通过指定从 blob 开头的偏移量，以页为单位读取和写入 blob。BlobStore 中的 cluster 不一定连续。

Blobstore：已由基于 blobstore 的应用程序初始化的 SSD。

Blobstore 拥有整个底层设备，该设备由私有 blobstore 元数据区域和由应用程序管理的 blob 集合组成。

（2）Blobstore 的块管理

在 Blobstore 中 cluster0 是一个特殊的 cluster，用于存放 Blobstore 的所有信息和元数据，对每个 blob 数据块的查找，分配都是依赖于 cluster 0 所记录的元数据进行的。Cluster 0 的结构如图 1-7 所示：

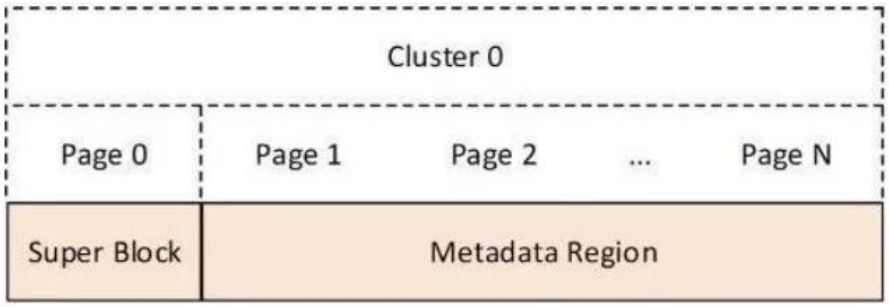


图 1-7 Cluster 0 结构

Cluster 0 中的第一个 page 作为 super block，blobstore 初始化后的一些基本信息都存放在 super block 中，例如 cluster 的大小、已使用 page 的起始位置、已使用 page 的个数、已使用 cluster 的起始位置、已使用 cluster 的个数、blobstore 的大小等信息。

Cluster 0 的其他 page 构成元数据域，元数据主要由以下几部分组成：

Metadata Page Allocation：用于记录所有元数据页的分配情况。在分配或释放元数据页后，将会对 metadata page allocation 中的数据做相应的修改。

Cluster Allocation：用于记录所有 cluster 的分配情况。在分配新的 cluster 或释放 cluster 后会对 cluster allocation 中的数据做相应的修改。

Blob Id Allocation：用于记录 blob id 的分配情况。对于 blobstore 中的所有 blob，都是通过唯一的标识符 blob id 将其对应起来。在元数据域中，将会在 blob allocation 中记录所有的 blob id 分配情况。

Metadata Pages Region：元数据页区域中存放着每个 blob 的元数据页。每个 blob 中所分配的 cluster 都会记录在该 blob 的元数据页中，在读写 blob 时，首先会通过 blob id 定位到该 blob 的元数据页，其次根据元数据页中所记录的信息，检索到对应的 cluster。对于每个 blob 的元数据页，并不是连续的。

为了实现对磁盘空间的动态分配管理，Blobstore 中为每个 blob 分配的 cluster 并不是连续的。对于每个 blob，通过相应的结构维护当前使用的 cluster 以及 metadata page 的信息：clusters 与 pages。Cluster 中记录了当前该 blob 所有 cluster 的 LBA 起始地址，pages 中记录了当前该 blob 所有 metadata page 的 LBA 起始地址。最后，blobstore 保证掉电不丢失数据（NVMe 这类 bdev）。

下面通过文件的读取来讲解 blobfs 与 blobstore 中的 I/O 流程。文件读取的流程图如图 1-8 所示。

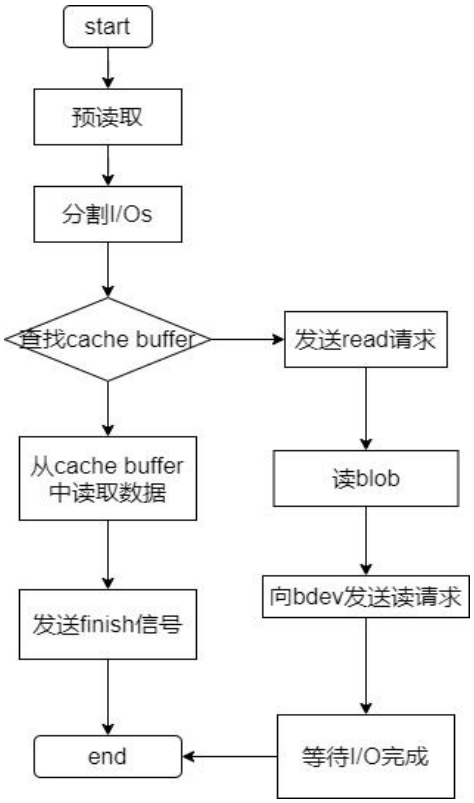


图 1-8 Blobfs 读流程

为了提高文件的读取效率，blobfs 在内存中提供了 cache buffer。在文件读写时，首先会进行事先读取操作，将一部分数据从磁盘预先读取到内存的 buffer 中。其后，根据 cache buffer 的大小，对文件的 I/O 进行切分，使每个 I/O 的最大长度不超过一个 cache buffer 的大小。对于拆分后的文件 I/O，会根据其 offset

在 cache buffer tree（叶子节点为 buffer node）中查找相应的 buffer。若存在，则直接从 cache buffer 中读取数据，进行 memcpy。而对于没有缓存到 cache buffer 中的数据，将会对该文件的读取，转换到该文件对应的 Blob 进行读取。对 Blob 读取时候，根据已打开的 blob 结构中记录的信息，可以获取该 blob 所有 cluster 的 LBA 起始位置，并根据读取位置的 offset 信息，计算相应的 LBA 地址。最后向 SPDK bdev 层发送异步的读请求，并等待 I/O 完成。BlobFS 所提供的读操作为同步读，I/O 完成后会在 callback 函数中，通过信号量通知 BlobFS 完成，至此文件读取结束。

1.3.4 Storage Protocol

Storage Protocols 包含在 SPDK 框架上实现的加速应用程序，以支持各种不同的存储协议。例如，用于 iSCSI 服务加速的 iSCSI taeget，用于加速 VM 中的 virtio-scsi/blk 的 vhost-scsi/blk target 以及用于加速基于 NVMe-oF 协议服务的 NVMe-oF target。

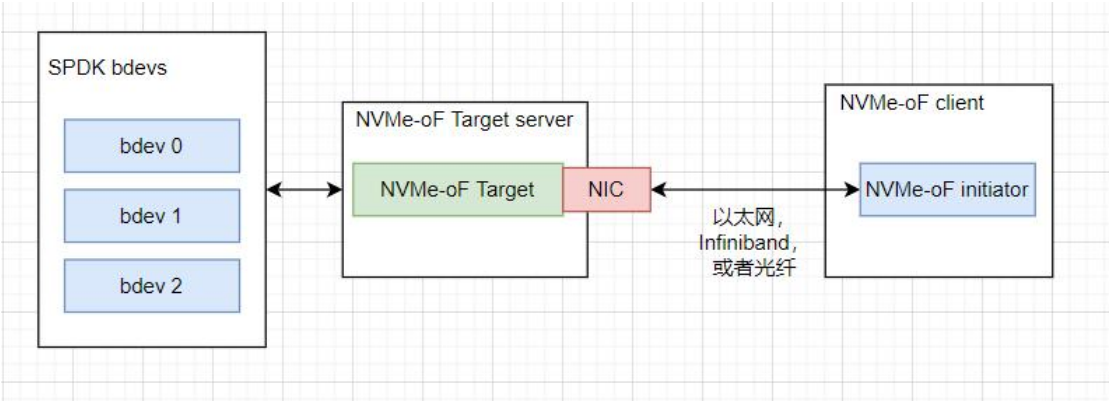


图 1-9 NVMe-oF target/initiator

以 NVMe-oF target 为例，如图 1-9 所示。NVMe-oF target 能基于 RDMA 通过网络呈现块设备。在用户空间 NVMe 驱动程序中，SPDK 提供了 NVMe-oF initiator，使得该驱动程序能够连接到远端 NVMe-oF target，并以与本地 NVMe SSD 相同的方式与它们进行交互。图 2-8 展示了 SPDK NVMe-oF target 的一般体系结构，NVMe-oF target 由 NVMf 库，NVMe bdev 以及 NVMe 用户空间驱动程序组成。其中，NVMf 库用于接收沿网络传输而来的请求，提取 NVMe 命令，并将其发送到 bdev，最后通过异步函数调用将结果返回给 NVMe-oF initiator。

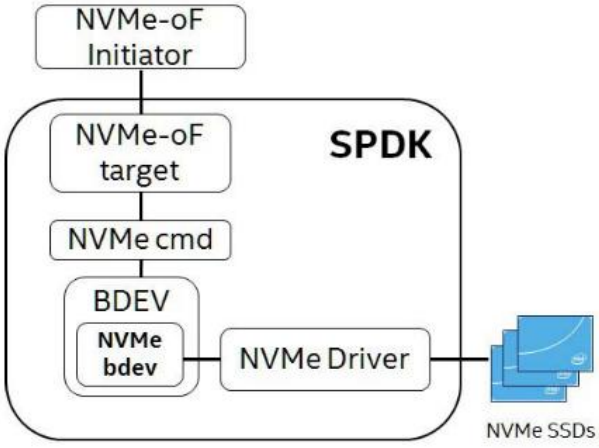


图 1- 10 NVMe-oF target 结构

NVMe-oF target 作为 SPDK 的一大用处，在兼容性和通用性方面，一直在进步。早期的 NVMe-oF 的规范中，只支持 RDMA 协议的网络传输，而在新的规范中，加入了基于 TCP/IP 协议的支持，所以在没有 RDMA 支持的网卡上也能运行，但性能不及 RDMA 网卡。而且如今的 NVMe-oF 协议能像 iSCSI target 一样，导出非 PCIe SSD，使得整个方案更加兼容。例如在 SPDK 中可以用 malloc 的 bdev 或者基于 libaio 的 bdev 来模拟出 NVMe 盘，将 NVMe 协议导入到 SPDK 通用 bdev 的语义，就能够给远端的用户呈现出 NVMe 盘。

SPDK 的 NVMe-of target 与 Linux Kernel NVMe-oF target 相比，具有以下优势：

- SPDK 的 NVMe-oF target 可以直接使用由 SPDK NVMe 用户态驱动封装的 bdev。相对于内核 NVMe 驱动而言，具有极大的性能优势。
- SPDK NVMe-oF target 完全采用 SPDK 所提供的编程框架，在所有 I/O 路径上都采用了无锁的机制，极大地提高了性能。
- SPDK 对于 RDMA 传输的实现融合了 SPDK 的编程框架，降低了在多并发连接情况下 I/O 处理的延迟。

1.4SPDK 性能评估

在 SPDK 论文中，作者使用 SPDK 的性能评估工具（nvme perf 等）对**一个 CPU core** 上的内核 NVMe 驱动和 SPDK 用户空间 NVMe 驱动做了 IOPS 的性能对比。随着驱动的 NVMe 硬盘的增加，内核 NVMe 驱动没有获得性能提升，但是 SPDK NVMe 驱动获得了近乎线性的提升，一个核就能驱动八块 SSD，如图

1-11 所示。不仅如此，SPDK NVMe 驱动还能取得更低的执行时间，如图 1-12 所示。

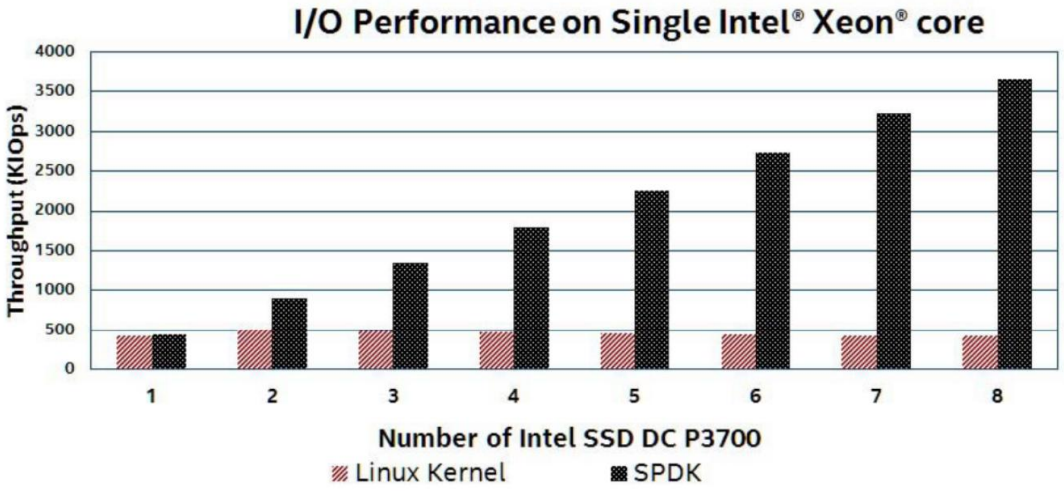


图 1-11 单个 core 的 I/O 性能测试结果

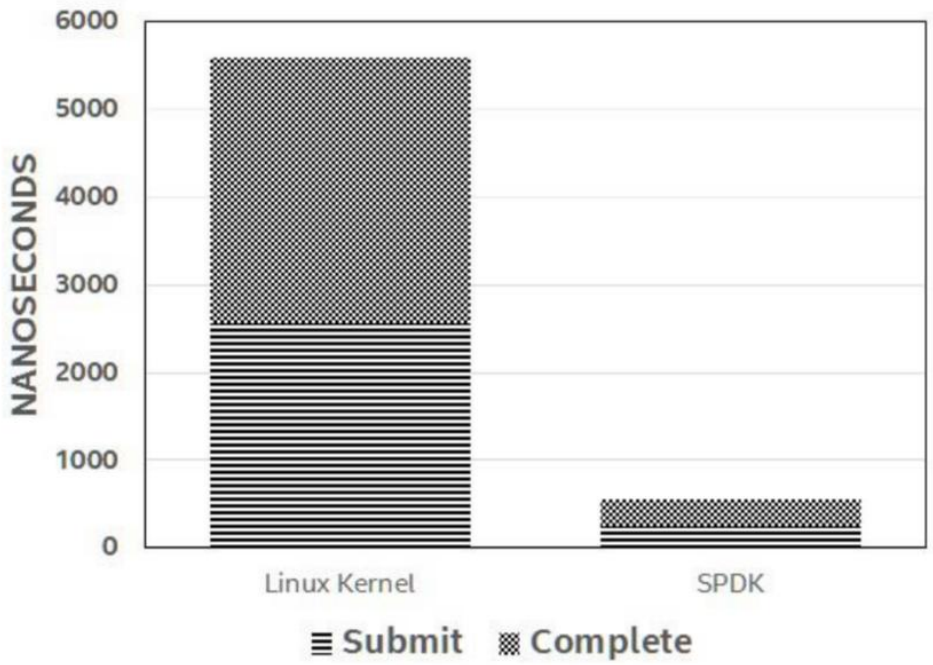


图 1-12 NVMe 驱动 I/O submit、complete 时间对比

这样的性能提升，主要归功于 SPDK 的异步轮询模式、无系统调用和数据拷贝开销、I/O 栈简化、无锁结构。

1.5SPDK 配置

（1）获取代码

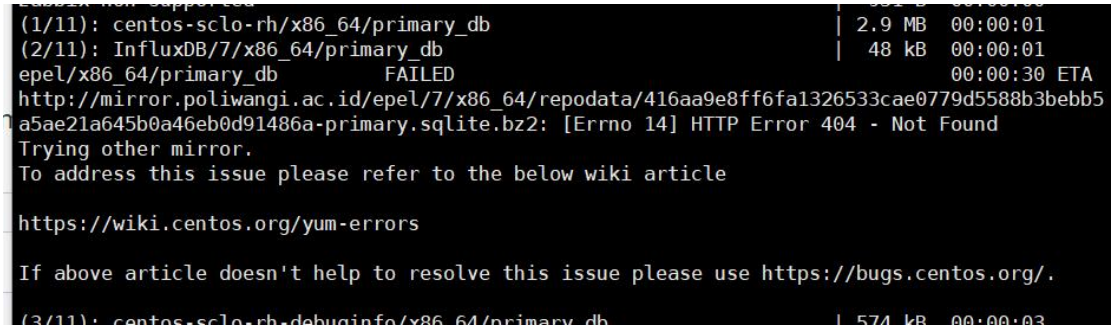
```
git clone https://github.com/spdk/spdk
cd spdk
```

```
git submodule update --init
```

（2）安装依赖

scripts/pkgdep.sh 脚本将自动安装构建 SPDK 所需的最小依赖项。

```
sudo scripts/pkgdep.sh
```



加上 - all 安装完整版依赖项依赖项。

```
sudo scripts/pkgdep.sh -all
```

（3）building

Linux：

```
./configure
make
```

配置脚本有很多选项，可以通过运行一下命令来查看

```
./configure -help
```

请注意，并非默认情况下启用所有功能。 例如，默认情况下不启用 RDMA 支持（因此不支持基于 Fabric 的 NVMe）。 您可以通过执行以下操作启用它：

```
./configure --with-rdma
make
```

（4）Running the Unit Tests

脚本末尾的最后一条消息指示成功或失败。

```
./test/unit/unittest.sh
```

（5）运行示例应用程序

在运行 SPDK 应用程序之前，必须分配一些大页面，并且所有 NVMe 和 I/OAT 设备都必须与本机内核驱动程序解除绑定。SPDK 包含一个脚本，可以在 Linux 上自动执行此过程。该脚本应以 root 用户身份运行。它只需要在系统上运行一次

```
sudo scripts/setup.sh
```

要将设备重新绑定到内核，可以运行.

```
sudo scripts/setup.sh reset
```

默认情况下，脚本分配 2048MB 的大页面。要更改此数字，请按以下方式指

定 HUGEMEM（以 MB 为单位）：

```
sudo HUGEMEM=4096 scripts/setup.sh
```

可以通过运行查看所有可用的参数

```
scripts/setup.sh help
```

示例代码位于 examples 目录中。 这些示例将在构建过程中自动编译。 只需调用任何不带参数的示例即可查看帮助输出。 如果您的系统启用了 IOMMU，则可以以常规用户身份运行示例。 如果不是，则需要以特权用户（root）身份运行。

在引用 spdk 在使用 spdk 之前，需要卸载内核中 NVMe 驱动程序并将指定的 NVMe 设备绑定到 Linux 内核中的 UIO 驱动程序上。在对使用到 SPDK 库的应用程序代码进行编译时，需要引用 SPDK 中 mk 文件夹下对应的链接库（主要是 spdk.common.mk 和 spdk.lib.mk），同时在运行时需要指定对应的配置文件，并配置文件中指定块设备名称等信息。

1.6实例分析

以 examples/bdev/hello_word 为例

```
int main(int argc, char **argv)
{
   struct spdk_app_opts opts = {};
   int rc = 0;
   struct hello_context_t hello_context = {};

   //使用默认值初始化 opts
   spdk_app_opts_init(&opts);
   opts.name = "hello_bdev";

   //这是没有指定具体的 bdev，使用默认的 Malloc0
   if ((rc = spdk_app_parse_args(argc, argv, &opts, "b:",
NULL, hello_bdev_parse_arg,
                                 hello_bdev_usage))    !=
SPDK_APP_PARSE_ARGS_SUCCESS) {
      exit(rc);
   }
   if (opts.config_file == NULL) {
      SPDK_ERRLOG("configfile must be specified using -c
```

```
<conffile> e.g. -c bdev.conf\n");
      exit(1);
   }
   hello_context.bdev_name = g_bdev_name;
```

//通过 spdk_app_start（）库会自动生成所有请求的线程，用户的执行函数 hello_start 跑在该函数分配的线程上，直到应用程序通过调用 spdk_app_stop（）终止，或者在调用调用者提供的函数之前，在 spdk_app_start（）内的初始化代码中发生错误情况。

```
   rc = spdk_app_start(&opts, hello_start, &hello_context);
   if (rc) {
      SPDK_ERRLOG("ERROR starting application\n");
   }

   //当应用程序停止时，释放我们分配的内存
   spdk_dma_free(hello_context.buff);
   //关闭 spdk 子系统
   spdk_app_fini();
   return rc;
}
```

Hell_word 的具体执行函数是在 hello_start 中，hello_start 运行在 spdk 分配的线程上：

```
static void
hello_start(void *arg1)
{
   struct hello_context_t *hello_context = arg1;
   uint32_t blk_size, buf_align;
   int rc = 0;
   hello_context->bdev = NULL;
   hello_context->bdev_desc = NULL;

   SPDK_NOTICELOG("Successfully started the
application\n");

   //根据 bdev_name，这里使用默认的 Malloc0，获取 bdev
   hello_context->bdev =
```

```
spdk_bdev_get_by_name(hello_context->bdev_name);
   if (hello_context->bdev == NULL) {
      SPDK_ERRLOG("Could not find the bdev: %s\n",
hello_context->bdev_name);
      spdk_app_stop(-1);
      return;
   }
```

//通过调用 spdk_bdev_Open（）打开 bdev 函数将返回一个描述符

```
   SPDK_NOTICELOG("Opening the bdev %s\n",
hello_context->bdev_name);
   rc = spdk_bdev_open(hello_context->bdev, true, NULL, NULL,
&hello_context->bdev_desc);
   if (rc) {
      SPDK_ERRLOG("Could not open bdev: %s\n",
hello_context->bdev_name);
      spdk_app_stop(-1);
      return;
   }

   SPDK_NOTICELOG("Opening io channel\n");
```

// 通过描述符获取 io channel

```
   hello_context->bdev_io_channel =
spdk_bdev_get_io_channel(hello_context->bdev_desc);
   if (hello_context->bdev_io_channel == NULL) {
      SPDK_ERRLOG("Could not create bdev I/O channel!!\n");
      spdk_bdev_close(hello_context->bdev_desc);
      spdk_app_stop(-1);
      return;
   }
```

//这里是获取 bdev 的块大小和最小内存对齐，这是使用 spdk_dma_zmalloc 所要求的，当然了你也可以不用 spdk_dma_zmalloc 申请内存使用通用的内存申请方式。

```
   blk_size =
spdk_bdev_get_block_size(hello_context->bdev);
   buf_align =
spdk_bdev_get_buf_align(hello_context->bdev);
   hello_context->buff = spdk_dma_zmalloc(blk_size,
buf_align, NULL);
```

```
   if (!hello_context->buff) {
      SPDK_ERRLOG("Failed to allocate buffer\n");

   spdk_put_io_channel(hello_context->bdev_io_channel);
      spdk_bdev_close(hello_context->bdev_desc);
      spdk_app_stop(-1);
      return;
   }
```

//初始化 buf

```
   snprintf(hello_context->buff,  blk_size,  "%s",  "Hello
World!\n");
```

//开始写

```
   hello_write(hello_context);
}
```

开始写操作

```
static void
hello_write(void *arg)
{
   struct hello_context_t *hello_context = arg;
   int rc = 0;
   uint32_t                    length                    =
spdk_bdev_get_block_size(hello_context->bdev);
```

//异步写，写完成后调用回调函数 write_complete

```
   SPDK_NOTICELOG("Writing to the bdev\n");
   rc     =     spdk_bdev_write(hello_context->bdev_desc,
hello_context->bdev_io_channel,
             hello_context->buff, 0, length, write_complete,
hello_context);

   if (rc == -ENOMEM) {
      SPDK_NOTICELOG("Queueing io\n");
      /* In case we cannot perform I/O now, queue I/O */
      hello_context->bdev_io_wait.bdev                   =
hello_context->bdev;
      hello_context->bdev_io_wait.cb_fn = hello_write;
      hello_context->bdev_io_wait.cb_arg = hello_context;
      spdk_bdev_queue_io_wait(hello_context->bdev,
hello_context->bdev_io_channel,
                &hello_context->bdev_io_wait);
   } else if (rc) {
      SPDK_ERRLOG("%s error while writing to bdev: %d\n",
```

```
spdk_strerror(-rc), rc);

   spdk_put_io_channel(hello_context->bdev_io_channel);
      spdk_bdev_close(hello_context->bdev_desc);
      spdk_app_stop(-1);
   }
}
```

写的回调函数

```
static void
write_complete(struct spdk_bdev_io *bdev_io, bool success,
void *cb_arg)
{
   struct hello_context_t *hello_context = cb_arg;
   uint32_t length;


   //响应完成，用户必须调用 spdk_bdev_free_io（）来释放资源
   spdk_bdev_free_io(bdev_io);

   if (success) {
      SPDK_NOTICELOG("bdev     io     write     completed
successfully\n");
   } else {
      SPDK_ERRLOG("bdev io write error: %d\n", EIO);

   spdk_put_io_channel(hello_context->bdev_io_channel);
      spdk_bdev_close(hello_context->bdev_desc);
      spdk_app_stop(-1);
      return;
   }


   //初始化 buf 为读做准备
   length = spdk_bdev_get_block_size(hello_context->bdev);
   memset(hello_context->buff, 0, length);
//开始读
   hello_read(hello_context);
}
```

读操作

```
static void
hello_read(void *arg)
{
```

```
   struct hello_context_t *hello_context = arg;
   int rc = 0;
   uint32_t                    length                    =
spdk_bdev_get_block_size(hello_context->bdev);
```

//读和写差不多，也是异步执行，等待回调函数

```
   SPDK_NOTICELOG("Reading io\n");
   rc      =      spdk_bdev_read(hello_context->bdev_desc,
hello_context->bdev_io_channel,
            hello_context->buff, 0, length, read_complete,
hello_context);

   if (rc == -ENOMEM) {
      SPDK_NOTICELOG("Queueing io\n");
      /* In case we cannot perform I/O now, queue I/O */
      hello_context->bdev_io_wait.bdev                    =
hello_context->bdev;
      hello_context->bdev_io_wait.cb_fn = hello_read;
      hello_context->bdev_io_wait.cb_arg = hello_context;
      spdk_bdev_queue_io_wait(hello_context->bdev,
hello_context->bdev_io_channel,
               &hello_context->bdev_io_wait);
   } else if (rc) {
      SPDK_ERRLOG("%s error while reading from bdev: %d\n",
spdk_strerror(-rc), rc);

   spdk_put_io_channel(hello_context->bdev_io_channel);
      spdk_bdev_close(hello_context->bdev_desc);
      spdk_app_stop(-1);
   }
}
```

读完成后回调函数被调用

```
static void
read_complete(struct spdk_bdev_io *bdev_io, bool success,
void *cb_arg)
{
   struct hello_context_t *hello_context = cb_arg;

   if (success) {
      SPDK_NOTICELOG("Read  string  from  bdev  :  %s\n",
hello_context->buff);
   } else {
      SPDK_ERRLOG("bdev io read error\n");
   }
```

```
    //结束后关闭 channel，释放资源
    spdk_bdev_free_io(bdev_io);
    spdk_put_io_channel(hello_context->bdev_io_channel);
    spdk_bdev_close(hello_context->bdev_desc);
    SPDK_NOTICELOG("Stopping app\n");
    spdk_app_stop(success ? 0 : -1);
}
```

2 经典超算 HPC 分布式文件系统

2.1GekkoFS

GekkoFS[7]是临时部署的、node-local 的用户态 burst buffer 文件系统。它致力于解决超算为生成、处理、分析海量实验数据而产生的大量元数据操作、数据同步、随机读写、小文件 I/O 的需求[8]。为了提高性能，它还放松了一些 POSIX 语义，如不支持 rename 操作。

通常的 node-local burst buffer 文件系统是一个用于过渡的，快速的或者称为临时（intermediate， not persistent）的存储系统，目的是减少后端用于持久化存储并行文件系统的负载[9,10]。它和运行的计算任务并存，有着和计算任务一致的生命周期，同时依赖后端的并行文件系统，有些情况下受后端文件系统的管理。例如，文件的安全管理由后端文件系统来处理，GekkoFS 不做这方面工作。

2.1.1 放松 POSIX 语义

经典并行文件系统实现了比较多的 POSIX（portable operating system interface）语义，虽然提供了很好的兼容性，但是也有一些代价：原子操作意味着读写中心化的数据结构（例如多个进程在同一个目录下创建大量文件时，POSIX 语义会导致所有的操作都是串行化）和对分布式锁的要求；分布式缓存意味着大规模的网络通信。

然而 POSIX 的少部分语义对大多数 HPC 应用来说，其实很少用到，比如容错和安全相关的一些语义。过多的 POSIX 语义给并行文件系统的性能带来了瓶颈。和并行文件系统相比，burst buffer 文件系统也实现了大部分的 POSIX 语义，适应大部分应用程序，提高了系统的性能而不用修改原有的应用程序。同时 GekkoFS 为了突破性能瓶颈，选择了放松某些 POSIX 语义的实现，用一个

key-value 的存储系统来保存文件系统的元数据，用于提高元数据的性能，同时也：

- 不支持分布式锁（distribute locking）
- 所有的元数据操作没有缓存，避免缓存一致性等复杂问题
- 不支持 rename 和 link 操作，因为 HPC 应用中很少用到这两个操作。

2.1.2 部署结构

在每个节点上，GekkoFS 都有客户端和服务端的概念。GekkoFS 在客户端以库的形式提供服务，在服务端设置 daemon 进程。它们的架构如图 2-1 所示。GekkoFS 客户端通过 lib 库的形式，截获所有文件系统的系统调用：如果是 GekkoFS 文件系统的操作，就通过网络转发给 GekkoFS 服务端，如果不是 GekkoFS 文件系统的操作，就调用原始的 glibc 库的系统调用；使用 file map 管理文件描述符到打开的文件的映射（和内核的独立）；还有基于 RPC 的通信层。服务端通过 key-value 存储（RocksDB）来保存文件系统的元数据，通过 I/O 持久化层用本地文件系统来保存数据，也有 RPC 通信层。客户端访问文件时，通过一致性哈希找到文件所在的服务器。

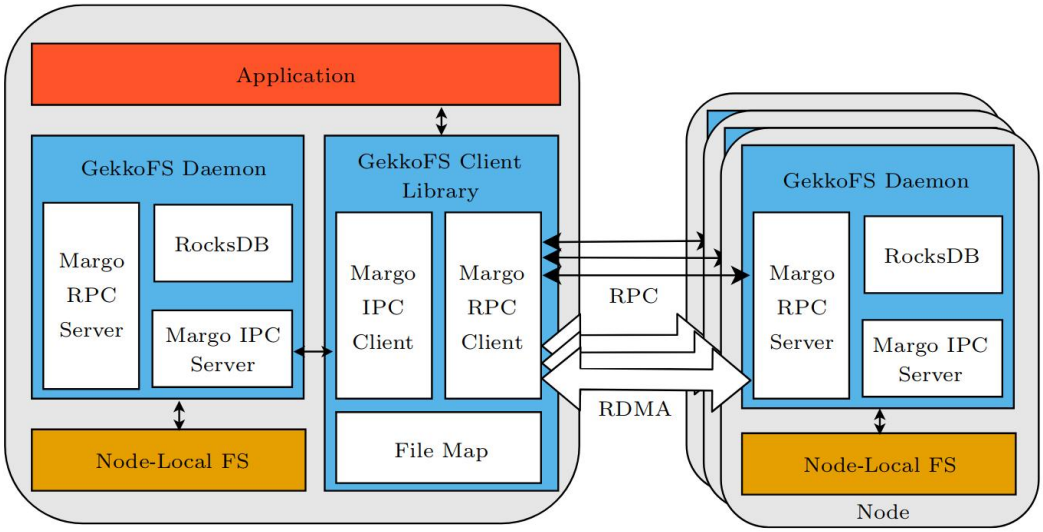


图 2-1 GekkoFS 结构

2.1.3 数据管理

对于一个 HPC 任务来说，文件的元数据也不是全部都是重要信息。比如权限、时间戳等信息，默认是不存储的，不过也可以通过配置来启用。元数据的存储方式则放弃了传统的目录方式，而是把文件路径作为一个 key，元数据作为

value 存放到 key-value 存储中。因为 GekkoFS 的每一个节点都可以独立处理信息，所以元数据的管理不需要一个中心化的结构，而是通过 nodeID = hash(path) mod nodeNum 的一致性哈希方式来确定一个文件对应的元数据存放到哪一个节点。这样，所有节点对某个文件发起读写元数据的请求，只需要通过一致性哈希函数计算出来存放元数据的节点编号，然后去相应的节点让其 daemon 进程利用 key-value 存储读写数据就好了，元数据一致性由 key-value 存储保证。

文件数据的存储方法也类似，不过数据是分成等大的 chunk 存放的。这些 chunk 的分布方式如图 2-2 所示，也是使用一致性哈希来计算 chunk 存放的节点。

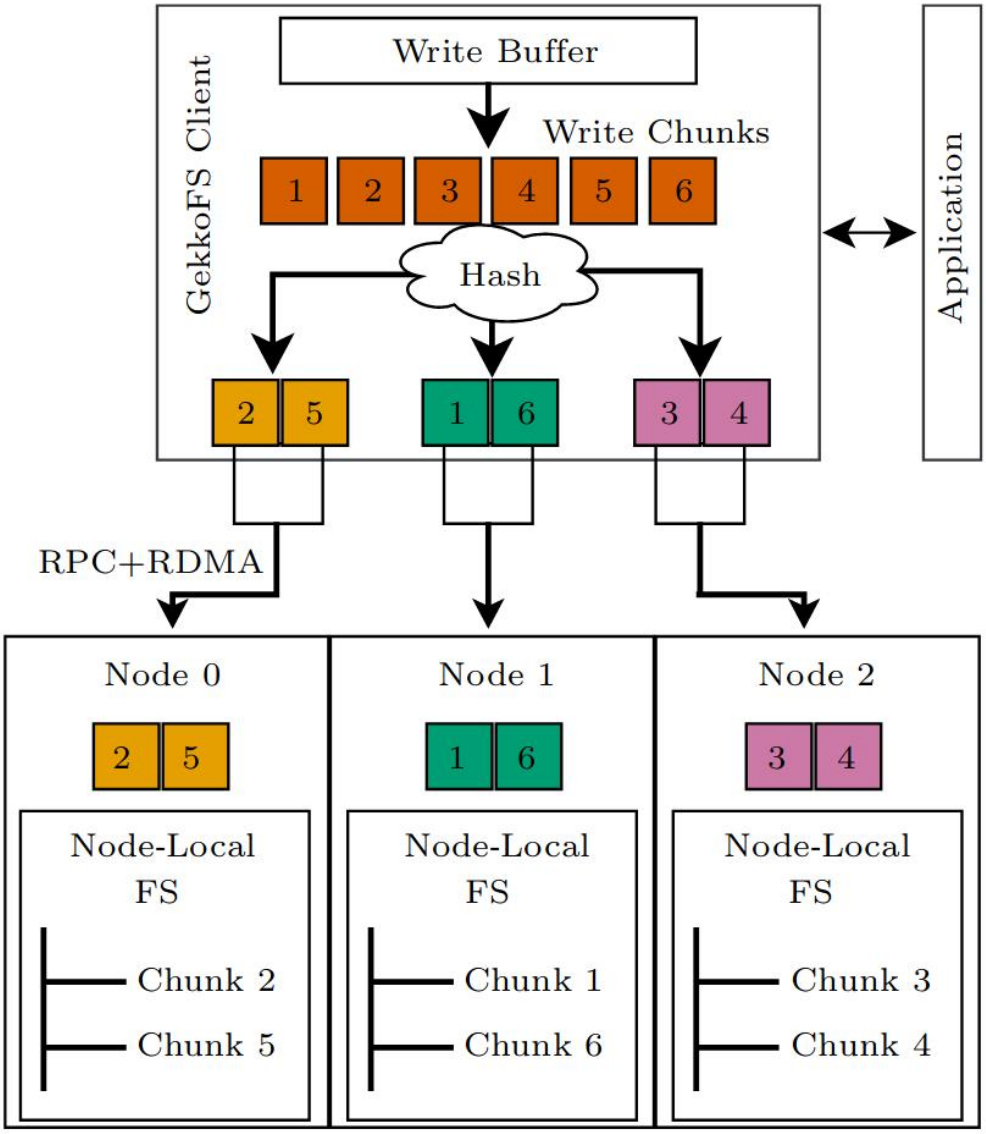


图 2-2 GekkoFS 数据的存放方式

Chunk 通过 RPC 和 RDMA 的方式发送给别的节点。如果是自己节点，就使

用 CMA（cross memory attach）。Chunk 的一致性由后端文件系统保证，因为后端文件系统会序列化对相同块文件的访问。

2.1.4 性能评估

详细的性能评估见原论文。GekkoFS 做到了文件操作速度、读写速度和节点数量成线性关系，有着良好的扩展性。在 32 个节点的时候，读写速度达到所用 SSD 的峰值性能。

3 参考文献

[1] Yang Z, Harris J R, Walker B, 等. SPDK: A Development Kit to Build High Performance Storage Applications[A]. 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)[C]. 2017: 154–161.

[2] Storage Performance Development Kit[EB/OL]. 2022-11-12/2022-11-12. https://spdk.io.

[3] Storage Performance Development Kit[EB/OL]. 2022-11-10/2022-11-10. https://github.com/spdk/spdk.

[4] souy_c. spdk 探秘 ----- 基本框架及 bdev 范例分析[EB/OL]. 2020-06-13/2022-11-10. https://blog.csdn.net/cyq6239075/article/details/106732499.

[5] Intel® Optane™ SSDs[EB/OL]. Intel. 2022-11-12/2022-11-12. https://www.intel.com/content/www/cn/zh/products/docs/memory-storage/solid-state-drives/optane-ssds.html.

[6] Yang J, Minturn D B, Hady F. When Poll is Better than Interrupt[J]. : 7.

[7] Vef M-A, Moti N, Süß T, 等. GekkoFS — A Temporary Burst Buffer File System for HPC Applications[J]. Journal of Computer Science and Technology, 2020, 35(1): 72–91.

[8] Nieuwejaar N, Kotz D, Purakayastha A, 等. File-access characteristics of parallel scientific workloads[J]. IEEE Transactions on Parallel and Distributed Systems, 1996, 7(10): 1075–1089.

[9] Liu N, Cope J, Carns P, 等. On the role of burst buffers in leadership-class storage systems[A]. 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)[C]. 2012: 1–11.

[10] Wang T, Mohror K, Moody A, 等. An Ephemeral Burst-Buffer File System for Scientific Applications[A]. SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis[C]. 2016: 807–818.