第 5 章　用户态调度框架 AULSF 改进 RBD 存储后端的研究

5.1　引言

如第二章所述，Ceph 的 Reef 版本（v18.2 版本）为其最近的稳定版本。虽然社区已开始逐步支持基于 Seastar 框架重构的 OSD（RBD 的存储后端）的特性，但并不稳定且有诸多问题，因此此功能尚未合并到 Reef 版本的稳定分支中。目前 Reef 版本依然默认采用多线程架构来处理存储请求、数据复制和网络通信等核心任务。OSD 通过多线程模型依赖操作系统的任务调度实现任务并发处理。然而，随着 RDMA 技术和 SPDK 用户态磁盘驱动等用户态技术（见 2.4 节）的出现，在高并发和低延迟的场景下，原生的多线程设计中的锁竞争等问题引发的频繁陷入内核态、多个线程在同一个 CPU 核心上同内核线程一起由操作系统调度引发的频繁上下文切换等问题浪费了大量 CPU 资源，没有有效利用新一代用户态技术带来的性能优势。

本研究设计了一套完全在用户态运行的任务调度框架 AULSF（Asynchronous User Level Schedul Frame，异步用户态调度框架），基于该框架对 RBD 的存储后端 OSD 的调度流程和元数据模块进行了重新设计，以优化原生多线程模型框架在高并发场景下的问题。其基本原理是通过 Linux 的 Grubby[75] 工具将目标 CPU 核心从操作系统隔离，不让内核线程运行在被隔离出来的 CPU 上并且调度框架独占此 CPU。然后在用户态构建轮询驱动的任务执行框架和回调驱动的异步读写流程执行机制。框架自行维护用户态各类任务的状态、实现各类任务的切换和实现轻量级的同步机制，在提供完善的调度功能的前提下避免了陷入内核或者 CPU 上下文切换。因此，整个任务框架完全运行在用户态绕开了操作系统并且充分利用了现代多核 CPU 的计算能力。

本章首先分别从基本概念、调度框架的基本实现原理介绍了 AULSF 的实现；然后介绍了该框架的辅助工具的实现以及这些辅助工具的必要性；接着，介绍了基于 AULSF 重新设计的 OSD 线程模型，并且以硬件的视角介绍了此设计基本原理。此外本研究摒弃了原生 OSD 中使用 Rocksdb 管理元数据的方式，让元数据跟随业务数据进行处理写入同一块磁盘（元数据具体实现见第六章）。进一步介绍了如何实现节点内读写流程的无锁设计，该设计下框架如何结合 RDMA 技术和 SPDK 用户态驱动技术完成读写流程。修改后的设计将网络资源、元数据资源、IO 设备资源按照 CPU 核心和硬件资源的 NUMA 亲和性进行了分配和整合；

最后，本研究设计了一个对比实验，分别对比了基于 AULSF 调度框架改进的 OSD 和 Ceph Reef 版本原生 OSD 的性能对比实验，实验结果显示在相同硬件和软件配置（能够使用 RDMA 和 SPDK 技术的相关硬件和软件配置）下，基于 AULSF 的 OSD 在同读写并发下时延更低，且最大 IOPS 是原生 OSD 的 3 到 4 倍。表明 AULSF 框架相比原生多线程框架更能发挥出 RDMA 和 SPDK 等用户态技术带来的性能优势，更适合

现代高性能分布式存储架构。

5.2 用户态调度框架 AULSF 设计

5.2.1 基本概念介绍

AULSF（Asynchronous User Level Schedul Frame，纯异步用户态任务调度框架）：管理了多个用户态独占线程，这些线程独占一个 CPU 核心；对异步任务进行管理。

RUP（Run Unit Pool，运行单元队列）：实现上为一个基于 CAS（Compare-And-Swap，对比并交换）实现的循环队列，主要用于处理一批由用户指定的任务，支持设置优先级和调度数。不同的 RUP 按照优先级和调度数由调度器分批次单个或者批量运行队列内的运行单元任务。

RU（Run unit，运行单元）：基本调度单位，可执行多种类型的任务，其组成包含了使用者自定义的任务处理函数和任务上下文，按照需求归属于某一个 RUP。

PS（Pool Scheduler，任务队列调度器）：用户态独占线程中不同任务的调度器，使用轮询的方式不间断处理 RU，提供不同类型的调度支持，决定不同 RU 的执行顺序。

ULET（Usr Level Exclusive Thread，用户态独占线程）：本质上可以视作是一个独占某个 CPU 核心的 Pthread 线程，绑定 CPU 核心后该线程运行过程中上下文切换均在用户态进行，任务的调度顺序由自定义调度器 PS 决定。

5.2.2 CPU 的隔离和绑定

CPU 的隔离：为了排除操作系统以及其他程序的干扰，ULET 创建时会指定线程的 CPU 亲和性。通过 Grubby 工具修改 Linux 系统引导配置，将目标 CPU 核从通用调度器中隔离出去，避免普通任务、中断处理和 RCU 回调在这些核心上运行导致陷入内核以及上下文切换从而影响到 AULSF 调度框架的调度，增加额外的性能开销。

CPU 的绑定：通过 Taskset[76] 工具将目标线程绑定到指定的 CPU 核心上运行。

5.2.3 用户态任务 RU 调度的基本原理

5.2.3.1 RU 的组成和 PS 调度器调度 RU 的语义

RU 的组成：RU 中包含了具体任务执行函数、任务执行的上下文和用于描述该任务属于哪个 RUP 等元数据信息。根据任务的不同，不同的 RU 中的任务需要根据 RU 中任务执行的上下文自行控制任务执行的进度，以及在任务无法完成时在上下文中维护并且记录执行进度。

PS 调度器调度 RU 的语义：按照函数调用的方式，运行 RU 中的函数，并且参数为 RU 中包含的任务执行的上下文。整个调度过程实际上等价于函数的调用过程，调度器只是按照不同的调度策略执行不同的 RU 任务中的函数。因此任务被调度开始执行

后，其执行流程是不可中断的，需要用户保证任务中不引入阻塞函数。即使引入了阻塞函数，本章 5.2.5 节也介绍了阻塞检测工具能够协助用户定位此问题。

5.2.3.2　用户态任务 RU 的调度流程

非抢占式用户态任务调度流程的实现：AULSF 框架不显式维护 RU 的栈指针、程序计数器等元信息。所以对于使用此框架的人来说，因为没有赖以恢复的上下文，所有的任务都是重新执行且任务执行完毕被 PS 调度器切换后无法继续从切换点继续执行。用户需要结合 RU 上运行的业务，在调度流程中自己维护业务所需的上下文并且维护好上下文中业务可能需要依赖的状态的更新。这样做的好处是函数切换效率更高，整个 CPU 核上的任务切换可以等价于不同函数之间的切换，额外开销极低但是任务执行的过程中无法被其他任务抢占打断。此种调度设计比较适合结合了 SPDK 用户态驱动和 RDMA 技术的高性能存储框架。

从 ULET 的角度看，每个 ULET 启动的时候都会设置 CPU 亲和性通过 5.2.2 节的方法让 ULET 独占某个 CPU 核。进程正常运行期间整个调度流程由无限循环的调度轮次组成，单个调度轮次中，不同 RUP 中的 RU 按照 PS 的调度策略被按顺序依次排列取出并执行，不同 RUP 中的 RU 都被调度过一次之后，一轮调度结束。因此从单轮调度的视角看，每一轮任务的调度可以看做按照不同的调度策略（不同的调度优先级和调度数设置），将 RUP 中 RU 的任务函数有序执行的过程。OSD 的线程模型的重新设计中本研究将网络资源的操作任务、IO 资源的操作任务和普通计算任务进行了区分，放在三个不同的 ULET 上分开调度处理，分别表示为 RUP_{net}、RUP_{diskio}、RUP_{normal}。

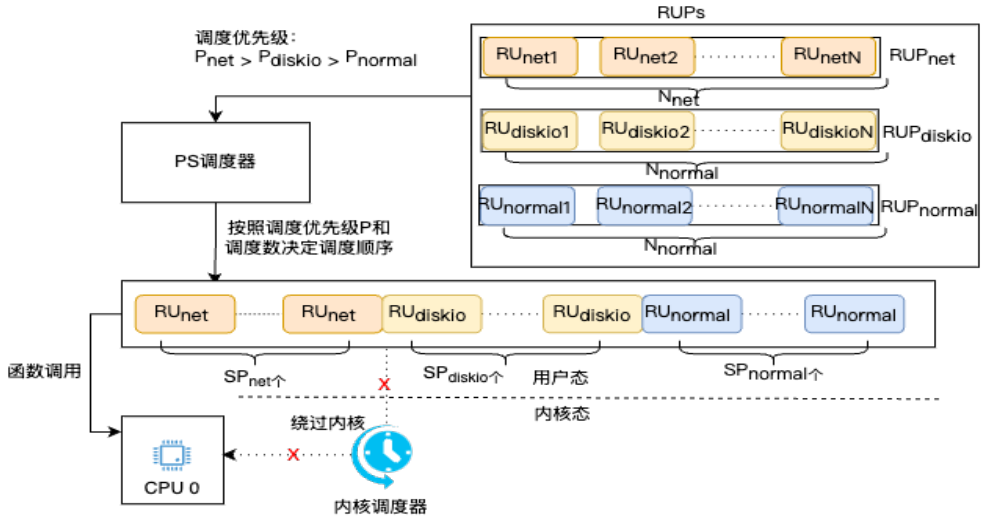

图 5.1　RU 调度原理

Figure 5.1　Run-Unit Scheduling Principle

图 5.1 展示了 CPU 亲和性被设为 0 即 ULET 独占 0 号 CPU 时 ULET 的内部的调度原理。其中设上述三类任务对应 RUP 的 RU 数目分别为 N_{net}、N_{diskio}、N_{normal}，RU_{netid}、

$RU_{diskioid}$、$RU_{normalid}$ 则分别表示不同 RUP 中下标为 id 的任务。RUP_{net}、RUP_{diskio}、RUP_{normal} 的调度优先级分别为 P_{net}、P_{diskio}、P_{normal} 且 $P_{net} > P_{diskio} > P_{normal}$，他们的调度数分别为 SP_{net}、SP_{diskio}、SP_{normal}。RUP 的调度优先级决定了 RUP_{net}、RUP_{diskio}、RUP_{normal} 的调度顺序，RUP 的调度数则规定了轮到某个 RUP 调度时，一次性连续调度执行的 RU 个数；若 RUP 中对应 RU 不足对应调度数则结束本 RUP 的调度然后调度别的 RUP，直到本轮所有 RU 都被执行过一次后该轮调度完成进入下一轮。

5.2.3.3 CPU 让渡、同步操作和跨线程通知机制的实现原理

如第二章所述，由于 SPDK 和 RDMA 技术的出现，磁盘读写和网络读写能在极短的时间（微秒级别）内完成，传统的中断处理方式频繁切换至内核态的消耗成为了原始 RBD 存储后端的瓶颈。所以如上节所述，整个调度框架设计为基于轮询的非抢占式调度。由于是非抢占式调度，为了便于 RU 任务自行控制任务的执行时长，避免长时间占用 CPU 资源导致其他 RU 无法及时被调度，本节提供了 RUyeild 原语用于主动让出 CPU。同时，单个任务可能依赖其他资源的完成（例如磁盘 IO），为了支持任务之间进行状态同步，确保多个任务可以有序地进行，本研究在用户态实现了类似 Linux 条件变量（或信号量）的同步机制 RUwait/RUset 操作。此外，由于一些管理 ULET 的存在，从设计上难免会需要跨 ULET 处理某些任务。对于这些任务，本研究设计了跨 ULET 通知的机制，通过将其他 ULET 的任务转发至目标 ULET 进行同步处理。

RUyeild 原语实现原理：RU 任务在每次执行时都会从头开始，多个 RU 之间相互独立，无需保存程序计数器（PC）、寄存器和栈指针等上下文状态。虽然这种设计避免了上下文切换的额外开销，简化了可能因为任务抢占需要额外维护的调度复杂度，能够最大化资源利用率和任务执行效率。但是这种实现方式在 RU 的使用上引入了额外的复杂度，RU 的使用者需要自己及时识别并管理 RU 任务的状态。因为本质上是函数调用，所以 RUyeild 的实现为立刻终止本次任务，让对应函数提前返回即可。RU 使用者需要在 RUyeild 之前正确维护在 RU 上运行的任务的状态，以便任务恢复时可以继续执行。任务的恢复则只需等待本轮 ULET 调度完成，下一轮调度重新调度到此 RU 继续执行即可。

RUwait/RUset 原语实现原理：RUwait/RUset 原语是一个基于无锁循环队列实现的信号量机制，该无锁循环队列的同步基于 CAS 实现，追加或者出队时使用完全在用户态的 CAS 机制替代需要陷入内核的 Pthread 锁机制保证流程的线程安全性。RUwait/RUset 使用时需要指定等待的自定义信号，设该信号名为用户指定的字符串设该字符串为 STR_{sig}，信号对应的无锁循环队列为 $SigQueue_{STRsig}$。

RUwait 的处理逻辑为：

（1）若 STR_{sig} 对应的 $SigQueue_{STRsig}$ 未被创建则创建 $SigQueue_{STRsig}$。

（2）若 $SigQueue_{STRsig}$ 已被创建则立刻中断本 RU 执行流程将其从对应 RUP 中移除，以追加的方式将 RU 放入信号对应的循环队列队尾等待 RUset 唤醒此任务。

RUset 的处理逻辑为：

（1）若 STR_{sig} 对应的 $SigQueue_{STRsig}$ 未被创建则不做任何处理

（2）若 $SigQueue_{STRsig}$ 已被创建，从队头出队信号量队列中的 RU 并且追加至 RUP 的队尾，等待重新调度执行。这种设计确保了任务的有序性，实现了无锁同步机制。

图 5.2 描述了 RUP 中 RU_a 和 RU_b 任务使用此机制实现 RU_c 和 RU_d 的同步处理流程，其中这四个 RU 按顺序执行：RU_a 被调度时调用 RUwait 需要等待某资源，立刻返回此任务从 RUP 中移除并追加写入 STR_{sig} 对应的 $SigQueue_{STRsig}$ 中，RU_b 被调度时调用 RUwait 同样等待此资源，操作同 RU_a 追加至对应 $SigQueue_{STRsig}$。RU_c 被调度完成了此资源的使用调用 RUset，STR_{sig} 对应的 $SigQueue_{STRsig}$ 则按照 FIFO（先入先出）的方式将 RU_a 任务追加放回 RUP 队列，RU_d 亦是如此。最终等到调度到被放回的 RU_a 和 RU_b 后即可继续执行。

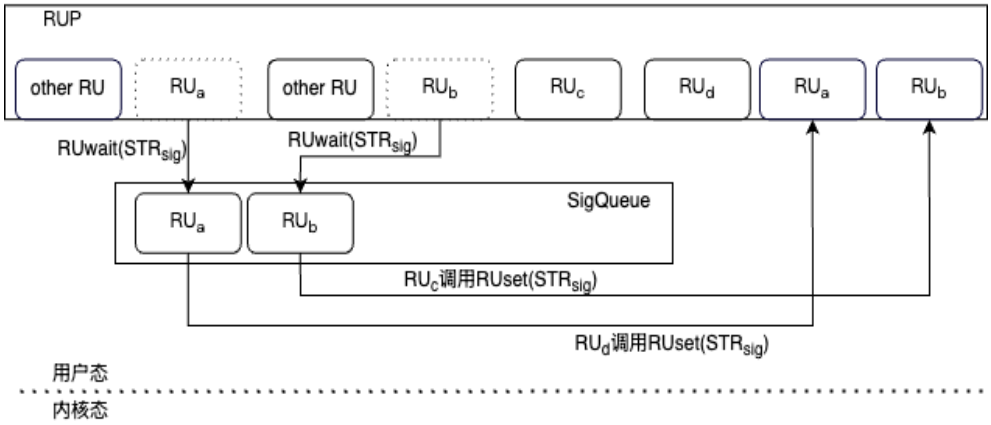

图 5.2　RUwait/RUset 原理

Figure 5.2　RUwait/RUset Principle

跨 ULET 通知机制的基本原理：在真实的业务设计上，完全理想的无锁设计是几乎不可能的，难免会有部分资源跨 ULET 使用，需要对应 ULET 处理。对于这类任务，本研究在每个 ULET 对应线程的私有栈内存中构造了一个同 RUwait 实现一样的无锁循环队列结构，用于接收其他 ULET 发送的任务。

图 5.3 展示了该同步机制的原理，设该无锁循环队列为 $MsgRing$，需要跨线程处理的消息分别为 msg_1 和 msg_2：

（1）当其他 ULET（记作 $ULET_c$）需要操作本 ULET（记作 $ULET_s$）中某些资源时，会以消息的形式发送到 $ULET_s$ 的 $MsgRing_s$ 中并且在消息中注册结果处理任务。对于 $ULET_c$ 而言任务发送完毕后即可继续处理 $ULET_c$ 的其他任务不阻塞其他任务流程。

（2）当一轮 ULET 开始调度的时候，会先遍历 $MsgRing_s$ 中的消息 msg_1 和 msg_2，根据消息中的信息创建消息任务 RU_{msg1} 和 RU_{msg2}，ULET 调度时优先执行消息任务，

将执行结果通过同样的跨线程消息通知的方式发送回 $ULET_c$ 的 $MsgRing_c$，$ULET_c$ 经过类似流程生成结果处理 RU 获取跨线程的执行结果。因为 ULET 中同一时刻只有一个 RU 在执行，以此方式规避了多 ULET 同时操作同一个资源的情况，不同 ULET 之间的对目标资源的操作顺序是固定的，避免可能导致程序处理错误的场景。

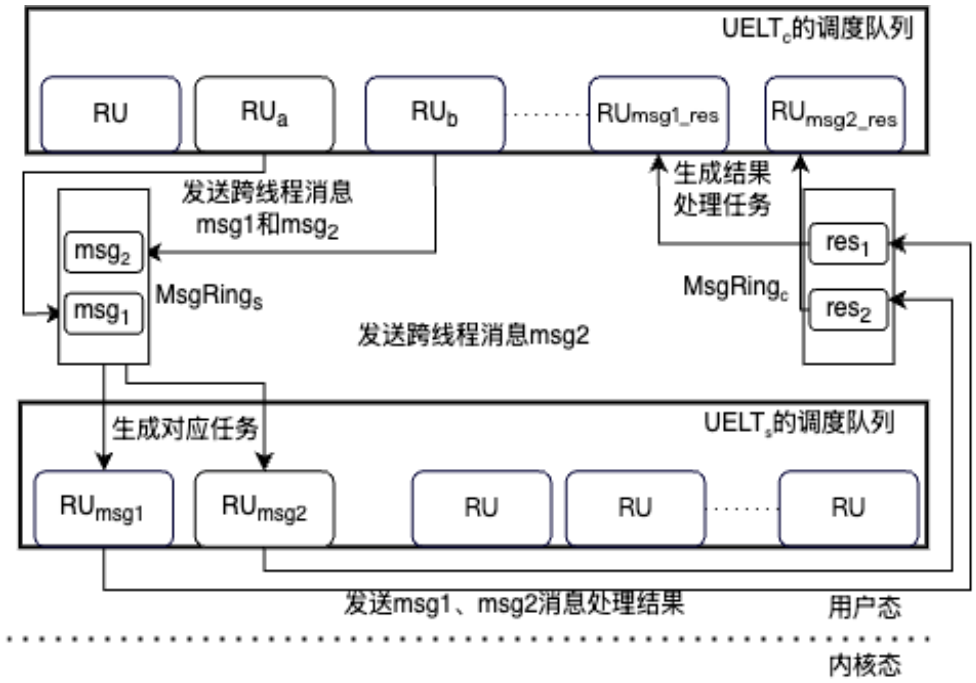

图 5.3　跨 ULET 消息处理机制

Figure 5.3　Cross-ULET Message Handling Mechanism

5.2.3.4　RU 的状态转换

RU 作为 AULSF 框架中的最小调度单位，从创建到销毁可能经历的状态分别有：Creating、Ready、Running、Waiting 和 Stopped。图 5.4 描述了 RU 在不同状态之间的流转：

（1）Creating→Ready：开始创建 RU 时 RU 即为 Creating 状态，RU 创建成功并放入 RUP 后从 Creating 状态变为 Ready 状态等待 PS 调度。

（2）Ready→Running：PS 调度器遍历到对应 RUP 中的 RU 并且开始执行后，RU 从 Ready 状态变为 Running 状态。

（3）Runing→Waiting：执行过程中发现某些资源不满足时需要等待此资源，由 RUwait 操作触发，对应 RU 被放入信号量队列从 Running 状态变为 Waiting 状态。

（4）Waiting→Ready：RUset 遍历信号量队列将对应 RU 放入 RUP 后，RU 从 Waiting 状态变为 Ready 状态，等待 PS 调度器调度。

（5）Running→Stopped：完成 RU 的所有任务后 RU 生命周期结束，进入 Stopped 状态等待清理 RU 的资源。

(6) Running→Ready：RU 为一直运行不退出的常驻任务，本轮处理完成从 Running 状态变为 Ready 状态，等待下一轮执行；或者 RU 为非常驻任务但是执行过程中评估本轮执行不完，调用 RUyeild 主动让出 CPU 从 Running 状态变为 Ready 状态等待下一轮调度。

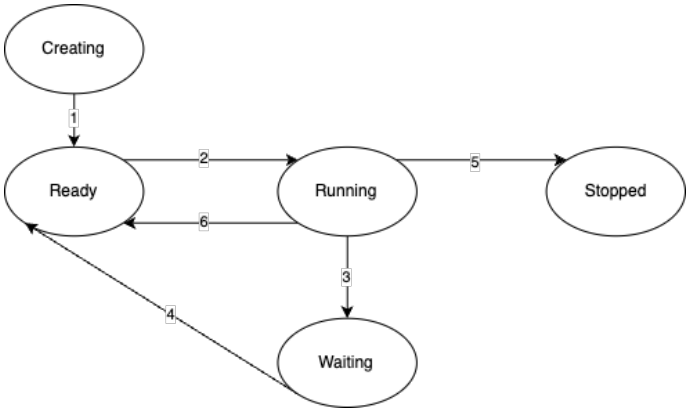

图 5.4　RU 状态转换

Figure 5.4　Run-Unit State Transition

5.2.4　辅助工具的实现原理和必要性

5.2.4.1　栈溢出保护设计

在用户态实现的调度框架中为了高效 RU 使用的内存通常由内存池统一管理，由于缺乏操作系统提供的内存保护机制，RU 使用的内存溢出问题难以被及时检测和定位，AULSF 也是如此。当 RU 的任务内存空间不足时，可能导致内存越界，覆盖相邻任务的内存区域，引发难以复现的程序崩溃或数据损坏。虽然目前有很多成熟的内存问题检测和定位工具，例如 ASAN（AddressSanitizer）通过在编译阶段插桩和运行时插入“毒区”（Red Zone）来检测栈/堆溢出、释放后使用等问题。ASAN 的优势在于能检测多种类型的内存错误并提供详细的错误报告（如源码行号、调用栈）。但其设计内存开销过高、性能开销不可接受等问题使其不适合用户态调度框架。因此，本研究设计并实现了基于 mprotect 系统调用的栈溢出保护机制。

图 5.5 展示了基于 mprotect 的栈溢出保护机制：在为 RU 分配栈空间时，额外申请一个内存页面作为保护页，并使用 mprotect 将该页面设置为不可访问（PROTNONE）。在 Linux 系统中，这个页面会被标记为不可读、不可写（即图中黄色区域）。如果栈指针越界并访问（图中红色点状区域）到这个保护页，CPU 将触发内存访问异常 SIGSEGV 信号。当栈溢出触碰到保护页时，程序可以捕获到 SIGSEGV 信号，进行异常处理或资源回收，防止更严重的问题发生。

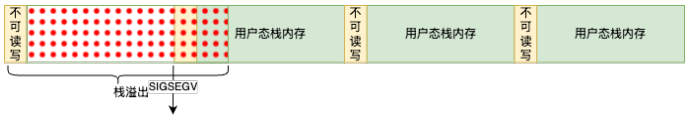

图 5.5　栈溢出检测机制

Figure 5.5　Stack Overflow Detection Mechanism

该设计不仅可以有效地防止栈溢出覆盖其他内存区域，还能够通过捕获异常提供明确的错误定位信息且相比于 ASAN 更加轻量级，方便用户快速排查问题。

5.2.5　基于信号和看门狗机制的 RU 阻塞检测机制

对于 AULSF 非抢占式的调度框架而言，单个 RU 的执行时间过长或者出现了某些处理阻塞 CPU 时间过长，则会严重影响在此 ULET 上所有的任务的执行，需要检测手段定位此类问题。因此本研究设计了基于信号和看门狗机制的 RU 阻塞检测机制。该机制首先使用类似看门狗的机制，通过业务 ULET 每次调度 RU 之前执行“喂狗”操作防止看门狗定时器超时，及时识别到不同 ULET 出现的 RU 阻塞的情况。其次结合了信号流程预先注册信号处理函数，一旦识别到某个 ULET 阻塞则发送对应信号，在信号处理函数中对问题 ULET 做特殊标记并且保存堆栈现场后立刻退出此信号处理。等到阻塞函数执行完毕后，下一次调度执行之前即可识别到这个特殊的标记进行 RU 和具体堆栈的记录。

图 5.6 详细的描述了在 $ULET_1$ 中出现了执行时长超过 T_{limit} 的 RU 情况下，看门狗设计机制检测问题 RU 的工作原理：

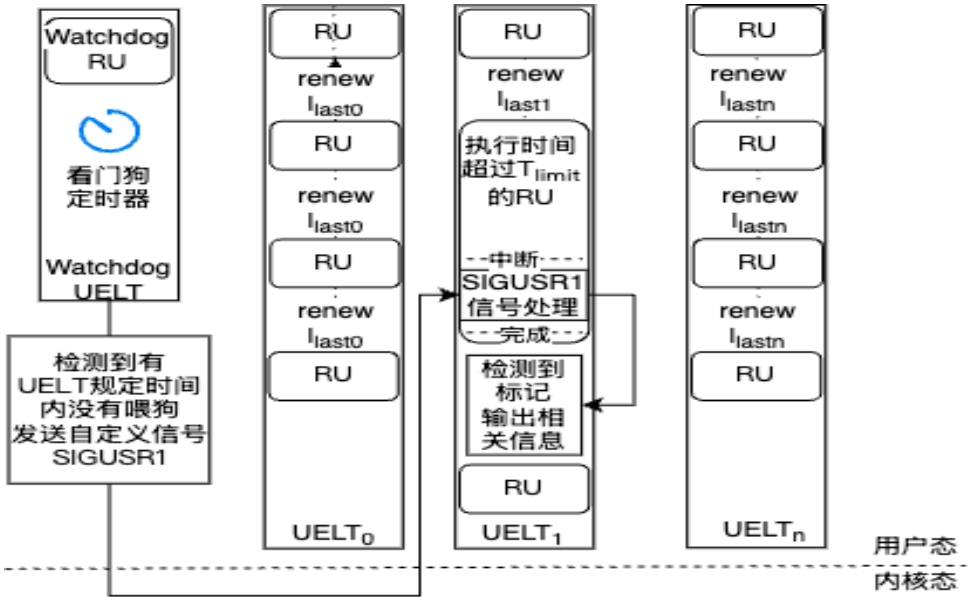

图 5.6　RU 阻塞检测机制

Figure 5.6　RU Blocking Detection Mechanism

设 $ULET_1$ 中出现了阻塞时长超过 T_{limit} 的 RU，每个业务 ULET 维护用于记录上一次 RU 执行完毕的时间 T_{last} 的看门狗变量，该变量随着 RU 被不断调度执行完毕而更新。看门狗所在的 ULET 记为 $ULET_{Watchdog}$，该 ULET 以 T_{timer} 时间定期遍历获取当前时间戳 T_{cur} 读取每个业务 ULET 当前的 T_{last} 并以超时门限值为 T_{limit} 进行超时检测，监测到超时则立刻发送的信号为 SIGUSR1。某个时刻 $ULET_{Watchdog}$ 上的 $RU_{watchdog}$ 检测到 T_{cur} - $T_{last1} > T_{limit}$ 表示 $ULET_1$ 有任务执行时间超过 T_{limit}，立刻发送 SIGUSR1 信号给 $ULET_1$ 所在的线程，进入信号处理函数。在 SIGUSR1 信号处理函数中将当前的 RU 的堆栈信息保存到磁盘后退出信号处理。此外，为了避免信号处理时间过长影响 $ULET_{Watchdog}$ 的判断，因此会限制堆栈打印的长度以快速保存堆栈信息。

5.3　基于 AULSF 框架优化 RBD 存储后端 OSD 的线程模型设计

本研究基于 AULSF 框架重新设计了 OSD 模块的线程模型（ULET 本质上是一个 Pthread 线程，只不过独占了一个 CPU 核），该设计基于 AULSF 中 CPU 亲和性和 NUMA 亲和性将 PG 中的读写操作按照网络资源、元数据资源和磁盘资源按照哈希的规则映射给固定的 CPU 处理。因为存储节点 OSD 进程启动后使用的 CPU 核数不会变，PG 的数目不会变且 SPDK 的用户态 NVMe 磁盘驱动为每一个 CPU 核维护了一个读写队列。所以，这些资源的布局信息是确定的，按照哈希分配的结果也不会改变，为无锁化设计提供了理论基础。

基于上述特性，某个 CPU 核对应的 ULET 只会处理固定范围的 PG 的元数据和读写请求并且读写数据的内容保存在对应 NUMA 的物理内存条对应的内存地址中，最后可以用本 CPU 核的读写队列对指定的 NVMe 磁盘下发读写请求。通过这样的方式实现了读写操作从网络收到开始，到最终写入磁盘几乎都在一个 ULET 中处理，即无锁化处理。

图 5.7 描述了硬件视角下基于 NUMA 架构 CPU 的结合 AULSF 框架的 OSD 线程模型，以及各类资源的物理布局，描述了整个无锁化处理的过程。图中描述的硬件为常见双 NUMA 架构的 CPU，每个 NUMA 上插了一个 IB 网卡、多个内存条、多个 NVMe SSD 且拥有多个 CPU 核心。设对应 NUMA NODE 上的 CPU 核心数目分别为 N_{numa0}，N_{numa1}，本节点一共有 N_{PG} 个 PG：

PG 的元数据的布局：每个 CPU 核处理的 PG 个数 N_{pm} 为：$N_{PG}/(N_{numa0}+N_{numa1})$，根据上述布局信息考虑与 NUMA 亲和性深度结合，在加载 PG 元数据的过程中或者在创建 PG 的时候在对应 NUMA 的内存条对应的物理内存上申请，避免出现跨 NUMA 申请内存的场景，否则跨 NUMA 访问的数据需要经过 NUMA 之间的总线通道传输，此场景效率较低。

用户态 NVMe SSD 的读写队列 QP 的布局：基于 SPDK 用户态 NVMe 磁盘驱动的特性（见 2.4.2 节），每个 NVMe 磁盘对每个 CPU 核心都会维护一个读写队列 QP

(QPairs)，用于处理某个核上对于本磁盘的 IO 请求，结合 ULET 的亲和性设置，QP 可以绑定到对应 ULET，而对应的 IO 处理则可以基于 RU 完成，在 RU 中调用 SPDK 中的读写任务即可。而整个读写操作的调度则交给 PS 处理。

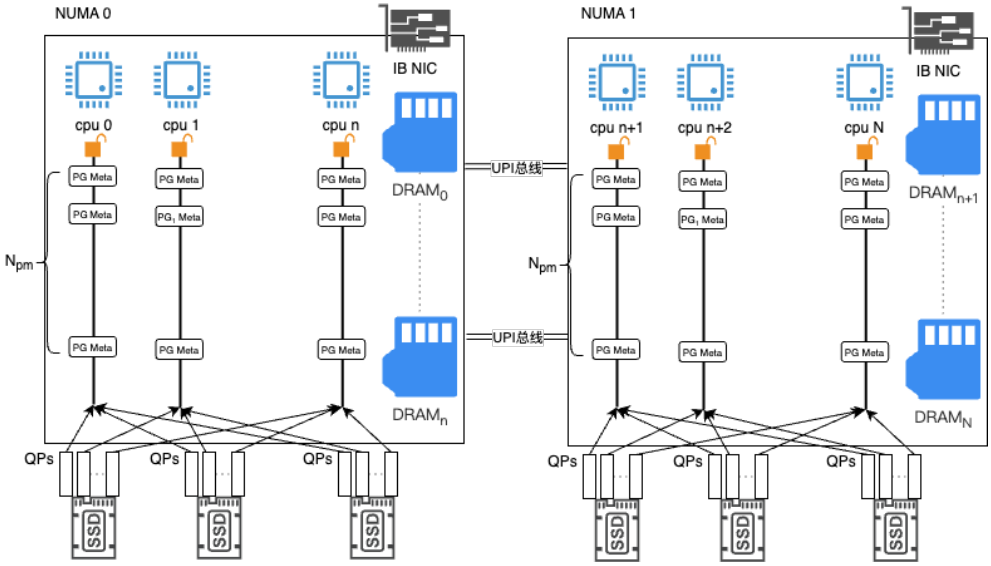

图 5.7　硬件视角下的 OSD 线程模型

Figure 5.7　OSD Thread Model from a Hardware Perspective

5.3.1　无锁化设计和以回调驱动的异步读写操作处理原理

如图 5.8，结合上节的 AULSF 框架的设计，本节以单个 PG 的视角出发，介绍了无锁化设计和以回调驱动的纯异步读写操作处理原理，以处理步骤的形式阐述如下：

STEP 1： 进程启动的时候在网络 RUP 中注册了检测网络请求完成队列的常驻 RU，检测是否有网络请求准备好，一旦有网络请求准备好则进入预先注册的请求处理函数 STEP3；

STEP 2： 进程启动的时候在网络 RUP 中注册了网络请求处理 RU，一旦收到网络请求将结果保存到对应 NUMA 上的内存（此处实现了异步接收网络数据）后立刻返回让出 CPU，然后等待 STEP1 的检测任务检测是否有请求完成。

STEP 3： 识别请求中的元数据信息，确认此读写请求属于哪个 PG 进而确认属于哪个 ULET 处理（通过哈希的方式实现了无锁处理，即哪个 PG 在哪个 ULET 执行是部署时就确认并且不会变化的），并且获取对应内存里数据信息的引用，索引到具体需要写入哪个磁盘后，此任务进入写磁盘阶段。创建包含读写盘结果处理函数的结果任务 RU，将此读写请求丢入磁盘驱动队列 QP 后立刻让出 CPU（此处体现了读写盘的异步处理）。

STEP 4： 设备读写盘完成后，将处理结果写入 QP 的完成队列中（因为 QP 也是每个 CPU 核独占的所以读写盘也实现了无锁处理）。

STEP 5： STEP3 创建的包含写盘结果处理函数的结果任务 RU 识别到请求完成后，调用结果函数创建读写操作结果的网络请求，并且将结果放入网络发送队列，等待对应 RU 将此请求通过网络发送给客户端。

综上所述、整个读写请求的处理实现了无锁化处理，并且读写流程中可能阻塞的网络收发和磁盘读写操作均实现了异步处理。整个读写操作处理过程中均基于其完成的事件调用对应回调函数，回调函数驱动了读写操作开始执行到完成并且回复客户端的整个流程。

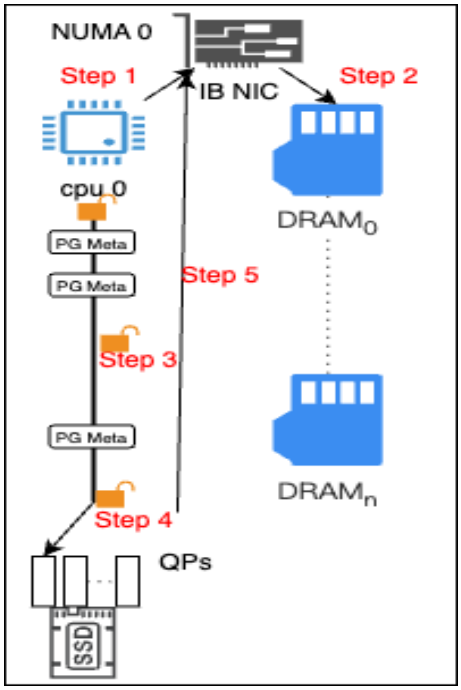

图 5.8　以回调驱动的异步读写操作处理原理

Figure 5.8　The Principle of Callback-Driven Asynchronous Read/Write Operation

5.4　实验结果与分析

5.4.1　实验环境

该实验在局域网内部搭建一个 3 节点集群，集群中各节点硬件配置和软件配置分别同第三章表 3.2 和表 3.3，各节点配置相同，集群拓扑同图 3.4。

实验前准备：首先，创建一个冗余策略为三副本的存储池 POOL01，存储池中加入三个节点的 18 块盘。然后创建 4 个 4T 的卷 vol1 到 vol4，分别通过 nvmf 协议挂载到客户端，客户端看到的盘符分别为 NVMe0n1，NVMe1n1，NVMe2n1，NVMe3n1。最后，分别预写写满这些卷的 100% 的数据空间也就是 16T，避免读到无效数据导致实验误差。

为了保证整个对比测试中只有任务调度模型不一致，本研究基于 AULSF 框架实现

5.5 本章小结

本章针对 Ceph Reef 版本在多线程架构下因锁竞争、上下文切换及内核态资源争用导致的 CPU 效率低下问题，提出用户态异步调度框架 AULSF 的 OSD 优化方案并结合 AULSF 重新设计了 OSD 的任务调度的无锁化设计。该设计摒弃传统多线程模型，采用 AULSF 框架实现 CPU 核心绑定的单线程处理模式，使读写操作全程在用户态完成，消除跨核锁竞争与操作系统调度开销。实验表明，改进后的 OSD 在相同硬件条件下，较原生 Ceph Reef 版本显著降低单核读写操作时延并提升了性能上限，并且扩展 CPU 核心时性能扩展度更优。表明了用户态异步调度与结合此设计的 OSD 在高并发场景下，能有效利用新一代用户态技术带来的性能优势。

第 6 章　RBD 存储后端的优化与实现

6.1　引言

OSD 是 Ceph 中承载数据存储和管理的核心模块亦是 RBD 的存储后端。本章基于前述章节所述使用 Raft 算法改进 RBD 的研究、Raft 算法本身的工程化实现中的优化研究和用户态调度框架的研究取得的研究成果为基础实现了 RaftStoreOSD 模块。该模块在维护了原始 OSD 提供的功能的前提下，缓解了存储系统的长尾时延问题，重新设计了 OSD 的元数据结构并构建了更适用于 RDMA 技术和 SPDK 用户态磁盘驱动技术的高性能存储后端。本节将从 RaftStoreOSD 模块的整体架构设计、Raft 算法状态机设计与实现、元数据与业务数据内存结构和磁盘结构设计与实现和 RaftStoreOSD 的读写流程依次进行阐述。

最后，实验章节设计了一个对比实验，对比了使用 RaftStoreOSD 新设计的 RBD 同原生设计的 RBD（Reef 版本），在相同的支持 RDMA 和 SPDK 技术的硬件环境下的不同读写模型下的性能。实验结果显示，使用 RaftStoreOSD 新设计的 RBD 在相同的业务并发下：4KB 随机读写性能是原生的 4.56 倍、1MB 混合顺序读写带宽则为 2.27 倍。结合前述章节的实验结果同本章实验结果表明：新设计下 RBD 在缓解长尾时延问题的同时性能更优。

6.2　RaftStoreOSD 架构设计

图 6.1 展示了 RaftStoreOSD 模块的整体架构设计其中：

（1）Raft Manager：本模块是整个 RaftStoreOSD 的核心，主要结合其他模块实现 Raft 算法和基于该算法的副本相关功能。Raft Manager 与其他模块的交互如下：其从块协议层处接收读写请求；使用 Raft NetWork Interface（Raft 网络接口模块）提供的网络接口与其他节点进行通信和数据交互；使用 PersistLocalStore 模块提供的接口实现 apply、append 本地日志等需要持久化到本地磁盘的操作；基于第五章所述 AULSF 调度模块进行 Raft 算法流程的推进。

（2）PersistLocalStore（持久化本地存储）：本模块负责数据的持久化存储，构建在 SPDK 用户态磁盘驱动以及 BlobStore 之上（见 2.4 节），重新设计了 OSD 元数据、PG 元数据和 OBJ 元数据的内存索引结构和磁盘存储结构并以 Blob 为单位进行数据的持久化管理。功能上则包含了 Raft 状态机的实现和 Raft 日志模块的管理，对 Raft Manager 层提供 Apply，Append entries，Commit 和第四章所述快照同步流程等需要持久化写入磁盘的具体语义实现。

（3）Raft NetWork Interface：本模块负责不同节点间的网络通信和数据传输。如第

三、四章所述本模块基于 RDMA 的 Infiniband 协议结合简单重试和网络收发模式设计了实现 Raft 算法交互所需要的 AppendEntries 流程、RequestVote 流程和增量快照同步流程所需的网络接口。

(4) AULSF：本模块负责整个 RaftStoreOSD 中不同任务的调度，实现如第五章所述非阻塞用户态调度框架。

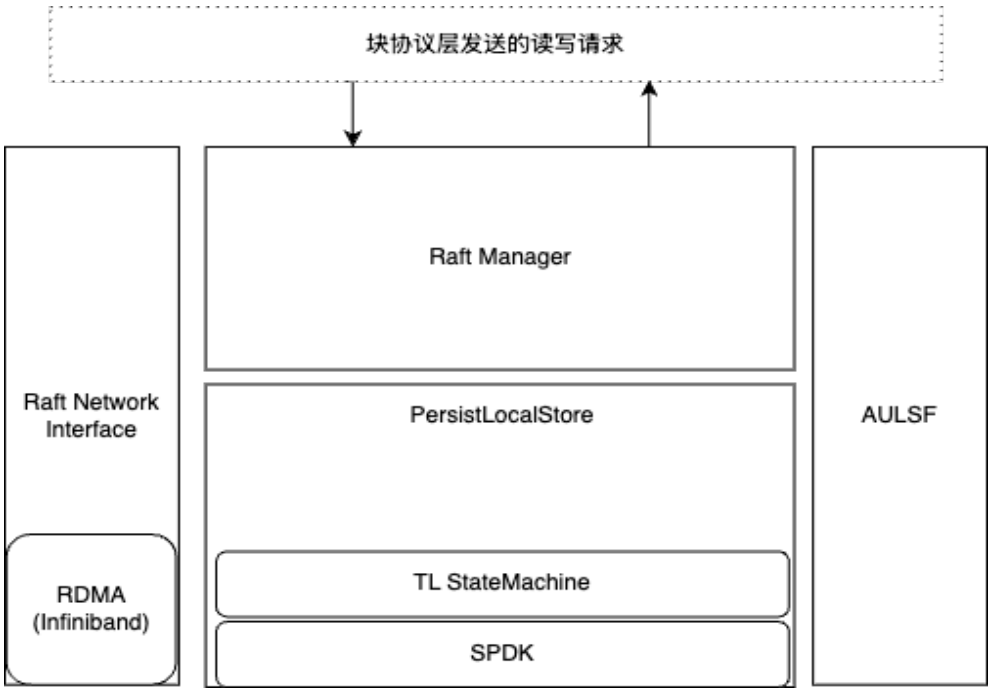

图 6.1 RaftStoreOSD 架构

Figure 6.1 RaftStoreOSD Architecture

6.3 状态机设计与实现

由于应用层的应用决定了 Raft 日志的设计结构，Raft 日志则包含了一个能够在集群节点之间达成共识的一系列操作序列，这些操作序列则通过 apply 操作改变状态机的状态。因此，为了讨论清楚状态机的设计需要先明确应用层的应用场景，接着设计对应的日志结构，进一步地结合应用层阐明 apply 操作的语义最后再介绍状态机的实现和状态转换的关系。

6.3.1 应用场景

应用场景：适合现代用户态技术（RDMA 和 SPDK）的分布式块存储 RBD。其设计目标为支撑 PB 级别的数据量，且最多支持创建 256 个 POOL、一百万个 VOL、16384 个 PG 且单个 PG 最多支持 1TB 数据（集群最大容量规格在 16PB 左右）。

应用层应用场景及其同 Raft 日志和状态机设计的联系：为了保障节点间数据的一致性，Raft 日志包含了一系列操作序列，这些操作序列即决定了读写的操作顺序。通过 Raft 的 Log Repication 流程可实现跨节点的数据复制，因此 Raft 日志同样应该包含这些读写操作的数据 Buffer。提供类似键值对（KV）接口的应用层状态机是一种简单且高效的实现方式。

6.3.2　Raft 日志条目结构设计

基于上节应用场景设计 Raft 日志结构如图 6.2 所示：1、64 位的 currentTerm 为 Raft 算法本身要求；2、8 位的 POOLID 最大表示 256 个 POOL；3、22 位的 VOLID 最大表示百万级的 VOL；4、14 位的 PGID 最大能够支持 16384PG；5、22 位的 OBJID 则是因为单个 PG 最大 1TB 最多支持百万级的 OBJ；6、10 位 Offset 是因为单个 OBJ 内部以 4KB 进行管理，用于表示本次读写请求在单个 OBJ 内部的起始偏移；7、Length：22 位的 Length 则表示本次写入数据的大小；8、10 位的 Opt 表示数据操作类型；9、64 位的数据 Buffer 指针指向数据存储的具体内存起始地址，其余为 64 位对齐字段。

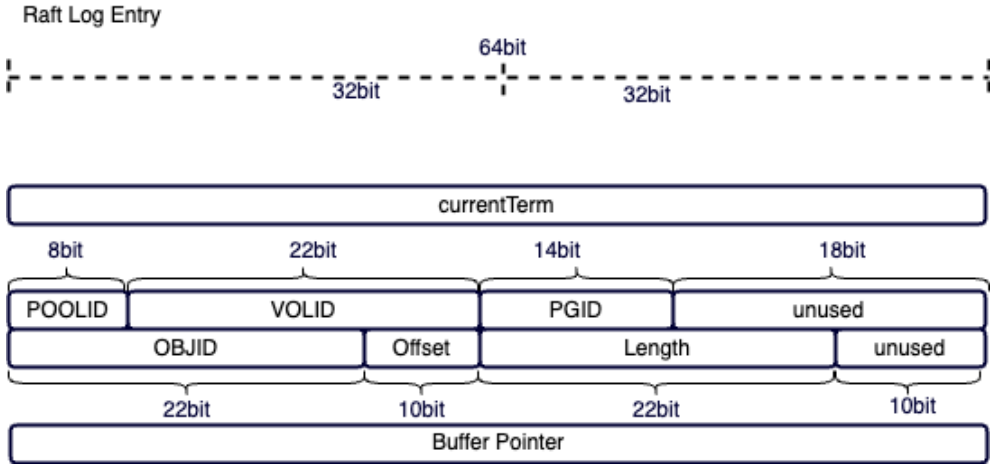

图 6.2　Raft 日志结构

Figure 6.2　Raft Log Structure

综上所述、从整个集群的视角结合了 POOLID、VOLID、PGID、OBJID 的四元组信息能够表示集群内唯一的一块 OBJ 地址，即可看作键值对中的键。而数据 Buffer 以及 Offset 则规定了修改内容。因此，根据此日志信息即可实现按照不同的操作类型操作对应数据。

6.3.3　两层磁盘状态机设计

存储在磁盘上的状态机设计：由 2.3 节描述的 Raft 算法流程可知，包含了读写数据的 Raft 日志需要持久化写入磁盘，并且状态机中的数据因为数据量过大也需要写入

磁盘。因此整个写操作流程中，对应用数据进行了两次写入带来了额外的写放大增加了读写时延。本研究设计了两层状态机 TL-SM（Two Level StateMachine），其主要思想是 apply 操作写入内存状态机，此为第一层状态机。在内存中缓存一定量的读写数据后，通过适合的合并和下刷策略后台异步将此部分合并后的内存数据批量写入磁盘，磁盘上的状态机数据为第二层状态机，以此缓解上述写放大问题。

状态机状态转换设计：状态机从一个初始状态启动，通过接受一个输入并传入状态转移函数中，将产生一个输出以及新的状态，在下一个输入到达之前，其状态不会发生变化。在 RBD 状态机设计中 apply 操作即为此状态转移函数定义为：对 Raft 日志中记录的 K 的位置所对应节点的 Blob 的对应偏移，将 V 中记录的长度为 length 的数据 Buffer 覆盖写入，随着该文件在磁盘上更新，状态机的状态更新。对应算法如下：

算法 1 状态机状态转换

初始化状态 $state \leftarrow init()$, $lastAppliedIndex$
while true **do**
  K 输入 $in\_msgk \leftarrow \langle POOLID, VOLID, PGID, OBJID \rangle$
  V 输入 $in\_msgv \leftarrow \langle Offset, Length, Buffer \rangle$
  将输入消息追加到日志 $log \leftarrow log.append(\langle in\_msgk, in\_msgv \rangle)$
  通过元数据确认写入的文件位置 $write\_pos \leftarrow in\_msgk$
  确认写入的文件 Buffer $write\_Buffer \leftarrow in\_msgv$
  结果持久化本地磁盘 $output \leftarrow \langle write\_pos, write\_Buffer \rangle$
  更新 lastAppliedIndex $lastAppliedIndex \leftarrow lastAppliedIndex + 1$
end while

图 6.3 所示为 TL StateMachine 的设计，第一层状态机为 MSM（Memory State Machine，内存状态机）其中：Non-stable Table 为可变的内存状态机用于积累状态机请求，不断接受 apply 操作的写入；Stable-table 为 Non-Stable Table 超过一个 OBJ 大小的数据后转为只可读不可写的待持久化磁盘的内存结构；具体处理流程见下描述：

STEP 1： apply 语义执行时开始将 Raft 日志中记录的操作应用到 MSM 中，写入 Non-stable Table 进行缓存。

STEP 2： 写入 Non-stable Table 成功后，回复客户端此读写请求成功。

STEP 3： Non-stable Table 满足刷盘条件即缓存了超过一个 OBJ 大小的数据后，将 Non-stable Table 转换为 Stable-table。

STEP 4： Stable-table 后台异步持久化写入磁盘。

STEP 5： 在写入磁盘成功的异步回调处理里，更新 lastAppiedIndex。

TL-SM 的数据返回成功的方式不会导致操作成功的数据丢失，分析如下：**STEP 2** 回复了读写请求成功，此时即使服务器掉电，因为 apply 操作并未完成，此部分读写操作所在的 Raft 日志仍然保存在磁盘中可以通过回放恢复这部分数据，所以这么数据

安全性仍然可以保证不丢失；一直到 **STEP 5** 写入成功后对应 Raft 日志才会清除，此时写入的数据仍然不会丢失。

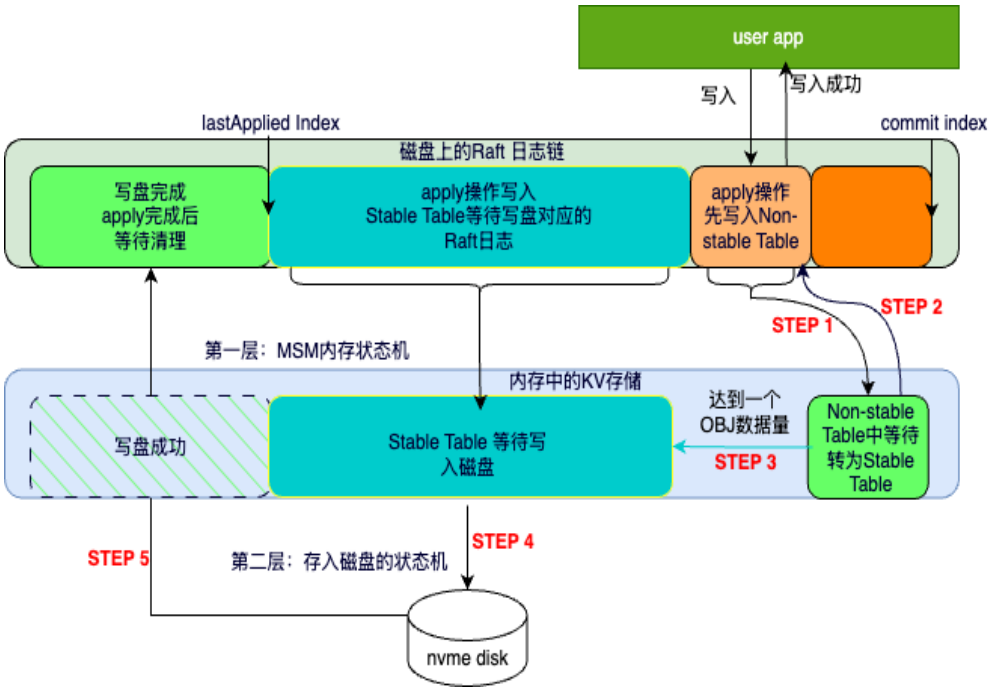

图 6.3　状态机设计

Figure 6.3　Design of the State Machine

6.3.4　结合 CRUSH 算法的 Raft 切主机制原理

Raft 博士论文中指出，在某些场景下当前的 Leader 节点不一定适合作为 Leader（比如当前接节点负载过大等原因），因此可以通过新增 TimeoutNow RPC 用于将 Leader 切换到指定节点。具体算法流程如下：设前任 Leader 为 S_{old}、期望切换的 Leader 为 S_{new}。

（1）S_{old} 停止接受新的客户端请求。

（2）S_{old} 通过 AppendEntries RPC 将目标服务器的日志完全更新为与自身一致。

（3）S_{old} 向 S_{new} 发送一个 TimeoutNow 请求。该请求的效果等同于 S_{new} 的选举计时器超时：S_{new} 会发起新的选举（增加其任期并成为 Candidate）。

6.3.5　Leader 节点对客户端写请求的回复处理

Leader 作为客户端写请求的处理中枢，上节描述了 Raft 日志设计和状态机设计，主要阐述了客户端的写请求如何结合 Raft 算法流程，应用到状态机并最终写入磁盘的过程。本节主要阐述 Leader 节点何时对客户端写请求进行回复处理，以及此处理是否对数据一致性有影响。

以客户端写请求处理的角度看，无论是同步等待或者异步等待请求完成，在一个现实系统中都需要设置一个等待时间。在该等待时间内客户端没有收到写请求的完成回复则进入超时处理流程，超时处理流程根据具体的应用不同可以选择重试或者报错。

以 Leader 的处理角度看，由于 Raft 算法的特殊性从客户端收到的写请求可能出现下述不同完成场景，设客户端的写请求处理的超时时间为 $Timeout_{client}$，这些场景下 Leader 的处理仍然能够保证数据一致性的讨论如下：

CASE 1、2： 集群中服务不稳定，Leader 未收到客户端发送的写请求或者收到后未来得及处理服务崩溃，因为写请求未完成，客户端请求超时失败，数据未写入成功，对数据一致性无影响。

CASE 3： 集群中服务不稳定，Leader 收到客户端发送的写请求并且发送给其余 Follower，多数派 Follower 写入失败，Leader 上对应日志条目无法在后续轮次修复。客户端请求超时失败，数据未写入成功，对数据一致性无影响。

CASE 4： 集群中服务不稳定，Leader 收到客户端发送的写请求并且发送给其余 Follower，多数派 Follower 写入成功，但是回复 Leader 失败，导致 Leader 在 $Timeout_{client}$ 内未完成请求。客户端请求超时失败，数据写入成功，但是根据状态机设计和 apply 语义，后续重试的 IO 覆盖写入此数据，结果不变。对数据一致性无影响。

CASE 5： 集群中服务不稳定，Leader 收到客户端发送的写请求并且发送给其余 Follower，多数派 Follower 写入成功，但是回复 Leader 失败，但是在 $Timeout_{client}$ 内选出新 Leader 并且日志修复成功。此时写入成功，对数据一致性无影响。

CASE 6： 集群中服务稳定，操作同 **CASE 5** 且 Leader 收到回复写入成功，对数据一致性无影响。

综上所述，Leader 节点无论何种场景一旦有写请求被提交，应该立刻回复客户端处理结果，因为此种处理客户端能容忍超时时间内的短暂故障而对整体正确性无影响且无其他副作用。

6.4 元数据内存结构和磁盘结构设计

作为 Ceph 中承载数据存储和管理的核心模块，其存储后端中的元数据分为 OSD 元数据、PG 元数据和 OBJ 元数据。其中，OSD 元数据作为索引所有元数据的入口用于描述磁盘的状态和 PG 元数据的布局。PG 元数据作为索引 OBJ 元数据的入口，描述了 PG 内部 OBJ 数据的布局情况、PG 的状态和 Raft 相关需要持久化的参数如 Raft 日志、currentTerm 等。OBJ 的元数据则描述了 OBJ 和其所属物理磁盘空间对应的 Blobstore 存储系统中的 Blob 文件。本节首先描述了多版本元数据设计如何保证节点内元数据一致的基本原理，然后分别从内存的角度和磁盘角度详细描述了 OSD 元数据的结构和 PG 元数据的结构，最后展示了 OSD 元数据、PG 元数据、Raft 元数据和用户数据在磁盘上的布局。

6.4.1　多版本元数据一致性保证

为了保证单节点元数据读写一致性，避免因为服务器重启、进程崩溃等原因导致元数据数据不一致，OSD 元数据、PG 元数据、OBJ 元数据均采用了多版本元数据一致性设计。该方案的核心是通过多版本冗余存储和版本号校验机制来保证元数据的一致性，即便在系统发生突发掉电的情况下，也能有效恢复数据。

如图 6.4，设原始数据版本分别为 $n-4$ 到 $n-1$，新写入了 n 和 $n+1$ 版本的元数据，其中 $n+1$ 版本写入途中崩溃或者数据损坏：

1. 多版本冗余存储：同一个元数据块在硬盘上存储 4 份副本，这些副本的版本不同。每个副本中包含两个字段：版本号和校验和。其中，版本号标识该副本的写入时间顺序，校验和用于检测该副本是否因掉电或其他原因被损坏。

2. 写操作：每次写入元数据时，新的数据将被写入硬盘中的一个副本，并且该副本的版本号将递增。同一时刻只能写入一个版本，并且新版本会覆盖最旧版本的副本。写操作会保证每个副本的版本号和校验和一致性。

3. 读操作：读元数据时，系统会同时加载这 4 个副本。对于每个副本，系统会检查校验和是否正确。如果某个副本的校验和错误（如写过程中因掉电而损坏），则丢弃该副本。剩余的有效副本中，版本号最大的副本会作为最终的有效数据返回，确保数据一致性。即图 6.4 中 $Version_{n+1}$ 计算出来的校验和和 $checksum_{n+1}$ 不匹配，因此采用 $Version_n$ 版本的元数据。

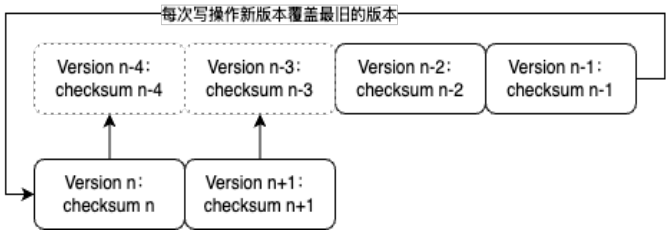

图 6.4　多版本元数据一致性原理

Figure 6.4　Principle of Multi-Version Metadata Consistency

综上、写操作过程中发生掉电，系统会通过校验和的验证机制丢弃损坏的副本，保留未损坏的副本。版本号较新的副本会覆盖掉版本号较旧的副本，从而确保最终的数据一致性。这种方案保证了元数据的容错能力，通过多版本存储、版本号控制和校验和机制，能够在硬件故障或掉电等意外情况发生时，尽可能恢复正确的元数据。

6.4.2　OBJ 元数据内存结构设计

如图 6.5，OBJ 的元数据主要用于 PersistLocalStore 模块根据 Raft 日志中的索引信息定位到具体的 Blob 文件。因此单个 OBJ 的元数据仅由：PGID、OBJID 和 BlobID 组成的，128 位即 16B 可表示。

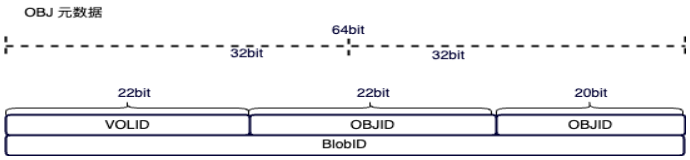

图 6.5　OBJ 元数据内存结构

Figure 6.5　OBJ Metadata Memory Structure

根据 6.3.1 节应用场景：PG 的设计规格为 1TB，单个 OBJ 大小为 4MB，因此一个 PG 内有 1TB/4MB = 256K 个 OBJ，这 256K 个 OBJ 对应的元数据整体大小为 256K * 16B = 4MB，而多版本元数据设计需要保存四份元数据。因此最大规格下集群中的 PG 内需要描述的 OBJ 元数据量为 16384*16MB=256GB。对于现代服务器而言，256GB 的元数据完全可以全部缓存在内存中，因此可以不需要考虑内存不够需要跟磁盘交换数据的问题。所以出于效率考虑尽可能减少元数据持久化磁盘的消耗和加快内存索引，并且能够描述这 256K 个 OBJ 在 PG 内部的布局信息为前提。本研究基于哈希桶设计了 PG 元数据的内存结构，节点内能在 O(1) 的时间复杂度，通过哈希函数快速根据 Raft 日志中的描述信息快速索引到对应 OBJ 元数据，并从其中获取本次读写操作应该操作的 BlobID。

6.4.3　PG 元数据内存结构设计

图 6.6 展示了 PG 元数据的内存结构，展示了基于多级哈希的 PG 元数据索引结构 $ObjHashBucketMap$：第一级哈希将整个 PG 的数据空间分为 256 个哈希桶记作 $OBJHashBucket$，其中每个 $OBJHashBucket$ 存了 1K 个的大小为 16B 的 OBJ 元数据四份总大小为 64KB。为了避免 $OBJHashBucket$ 内数据过长，第二级哈希将 $OBJHashBucket$ 的数据空间分为了 4 份称为 $OBJHashBucket_2$，每一份 $OBJHashBucket_2$ 长度为 256 的大小为 16B 的 OBJ 元数据链，其总大小为 16K。

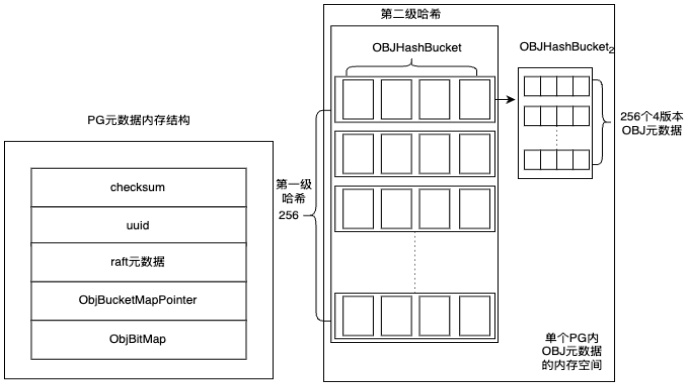

图 6.6　PG 元数据内存结构

Figure 6.6　PG Metadata Memory Structure

根据此内存的多级哈希设计，因此 PG 的元数据中除了描述 PG 本身属性如校验和 (checksum、PGuuid 等) 的数据外，只需要包含 $ObjHashBucketMap$ 的首地址的指针即可，单个 PG 元数据大小大小在 1KB，PG 元数据整体大小为 1KB*4*16384=64MB。

6.4.4　OSD 元数据内存结构设计

OSD 元数据用于描述 OSD 上 PG 的布局以及 OSD 的状态数据，图 6.7 展示了，OSD 元数据内存结构：1、用于描述 OSD 的属性的数据：校验和，OSDuuid，OSD 状态等 2、PG 元数据数组的指针：以 PGID 作为下标即可快速访问对应 PG 的 4 个版本的元数据。3、PG 位图（PGBitMap）：用于快速查询某个 PG 是否存在，总大小位 16384 位即 2KB。因此 OSD 元数据总量在 64MB 左右。

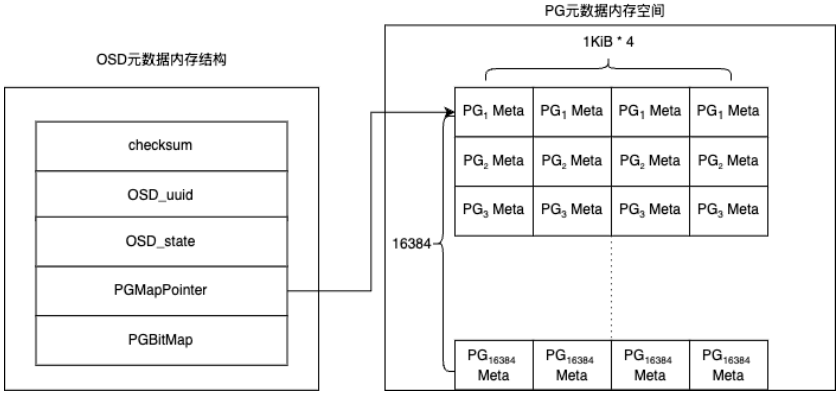

图 6.7　OSD 元数据内存结构

Figure 6.7　OSD Metadata Memory Structure

6.4.5　元数据磁盘结构设计

上节描述了 OSD 元数据、PG 元数据和 OBJ 元数据在内存中的结构，这些结构分别描述了 OSD 内部 PG 的布局，PG 内部 OBJ 的布局，OBJ 对应的磁盘 Blob 文件的 BlobID。对于存储系统而言，这些描述了集群数据布局和管理信息的数据需要持久化写入磁盘，随着服务重启重新加载这些元数据到内存中以提供索引服务，否则节点就会因为没有索引信息无法提供服务。而对于磁盘结构的设计需要作相应修改，因为内存中的 PG 元数据数组指针和 ObjBucket 指针信息，仅代表了当时内存中对应数据的相对地址，随着每次装入内存该指针信息会改变。因此无法直接保存指针的值，需要保存这些指针指向的内容所持久化写入的磁盘 Blob 文件的 BlobID，并且随着各类元数据更新，这些 Blob 也随之更新。由于 Blobstore 将磁盘划分为了以 Blob 文件为单位进行存储（Blobstore 原理见 2.4.3），本节从整个磁盘的视角，介绍了上述元数据在磁盘上的结构和布局。

图 6.8，从磁盘的视角描述元数据和数据的布局，图上的绘图元素大小不代表真实磁盘空间占用大小。真实磁盘空间占用大小关系为：数据 Blob 占用 ≫ 哈希表元数据 Blob 占用 ≫ Raft 日志 Blob 占用 > PG 元数据占用 > OSD 元数据 Blob 占用。

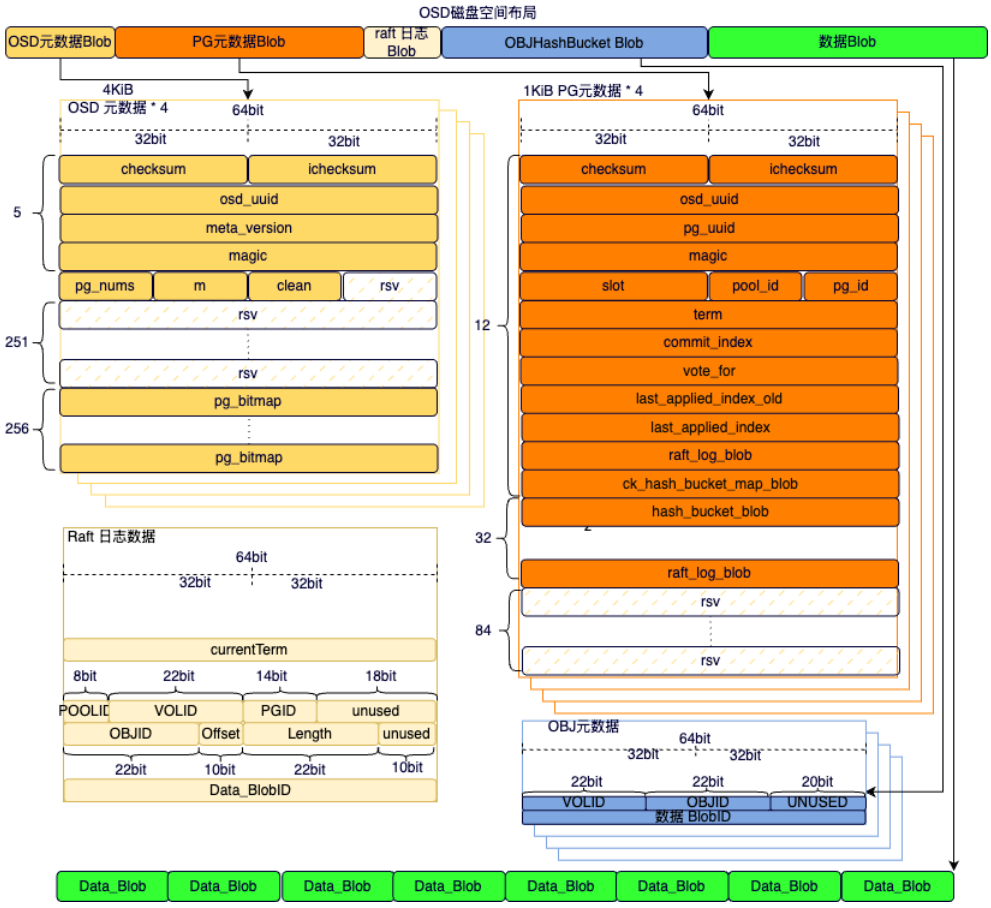

图 6.8　OSD 磁盘布局

Figure 6.8　OSD Disk Layout

OSD 元数据：OSD 元数据为该磁盘所有元数据的总入口，其主要保存了 PG 的元数据在 OSD 上的布局信息，以及为了保证自身一致性使用校验和算法所需要的额外信息，详细磁盘结构如下：

第一个 64 位中主要保存的校验和信息，用来保证数据完整性；第二个 64 位保存 OSDuuid 作为其在本节点唯一标识；第三个 64 位为 meta_version 保存此元数据的版本号同第一个 64 位用于 6.4.1 节一致性保障；第四个 64 位为魔术字用于判断此元数据是否损坏。第五个 64 位总共分为了四部分，第一个 16 位表示该 OSD 存储的 PG 数目，第二个 16 位表示在 PGBitmap 中目前最大的槽位号，第三个 16 位用来标识 OSD 是否安全退出，若非安全退出启动后需要走恢复流程，重建元数据信息，后面 16 位和接下来的 251B 作为预留空间，用于数据对齐或者版本升级新增字段。最后的 16384 位则为一个 PG 位图用来描述改 OSD 上的 PG 布局，对应槽位则为对应 PG 的 PGID，同

时可以通过 PGID 作为位移，索引该 PG 元数据空间的中的元数据。

PG 元数据：PG 元数据存储了该 PG 的数据空间在磁盘上的布局，以及存储了共识算法需要的信息，本身的校验信息。同 OSD 原数据一样，头 32B 的数据分别描述了校验和、OSD 和 PG 的唯一标识；第五个 64 位分别包含了 PG 在 OSD 的 PG 位图中的槽位号、POOLID 和 PGID；第 40B-80B 的空间用于存储 Raft 算法相关元数据；第 80B 到 96B 存储了 OBJ 哈希的索引信息，即内存中 $OBJHashBucket$ 结构对应的 Blob 文件地址。以及为了对齐到 1KB 预留的字段，可用于后续升级。

Raft 日志：Raft 算法要求 Raft 日志需要持久化，其结构基本如内存结构，仅相关数据指针信息变为了对应数据存储的 BlobID。

$OBJHashBucket$ 内存索引对应的元数据：其结构基本如内存结构，仅相关指针信息变为了对应数据存储的 BlobID。

综上所述，磁盘结构可以近似看作内存结构的对应数据按照 64 字节对齐持久化写入磁盘上对应 Blob 后的结构。

6.5　RaftStoreOSD 的写操作

图 6.9 展示了写流程在 RaftStoreOSD 内部不同模块中的处理步骤。写操作在 Raft Manager 模块的运行如下：

STEP 1：根据 PGID 通过 Hash 算法 1: 以 PersistLocalStore 的线程数做简单的哈希，确认该 PG 的所属 ULET，若同 PG 的所属的处理 ULET 不一致，则将写请求通过跨 ULET 通知机制（见 5.2.3.3 节）发送给对应 ULET 进行处理。

STEP 2：Raft Manager 通过协议网关接收来自客户端的写请求，该写请求中包含了具体写操作的目标 PG 的 PGID、OBJID 以及对应写请求的 Buffer 和相关描述信息。Raft Manager 将这些信息按照 Raft 算法的要求封装进 Raft 日志（结构如 6.3.2 节所述），通过调用 PersistLocalStore 模块提供的持久化接口，交给 PersistLocalStore 模块处理。

写操作在 PersistLocalStore 模块的运行如下：

STEP 3：从内存中对应 OSD 元数据中 PGMapPointer 以 PGID 为下标的方式获取 PG 的元数据。根据 PG 元数据中记录的 $ObjHashBucketMapPointer$ 指针，通过哈希算法 2：OBJID 余 256 确认 OBJ 所属 $OBJHashBucket_2$ 的地址，然后在 $OBJHashBucket_2$ 内部通过二分查找确认对应 OBJ 元数据。

STEP 4：根据 OBJ 元数据中记录的 BlobID 信息，确认此次写流程操作的 OBJ 所属的 Blob，结合写请求的 Buffer 和相关描述信息调用 SPDK 模块提供的 Blob 读写接口，交给 SPDK 模块处理。SPDK 模块基于这些信息确认文件所属设备，通过对应设备磁盘驱动写入对应存储设备的对应磁盘位置并且更新写请求完成结果。

STEP 5：写请求完成后，相关 RU 任务检测到并且调用注册好的完成回调函数一

层一层向上层返回，先返回给 PersistLocalStore 模块然后往上返回给 Raft Manager 模块最后返回给客户端。

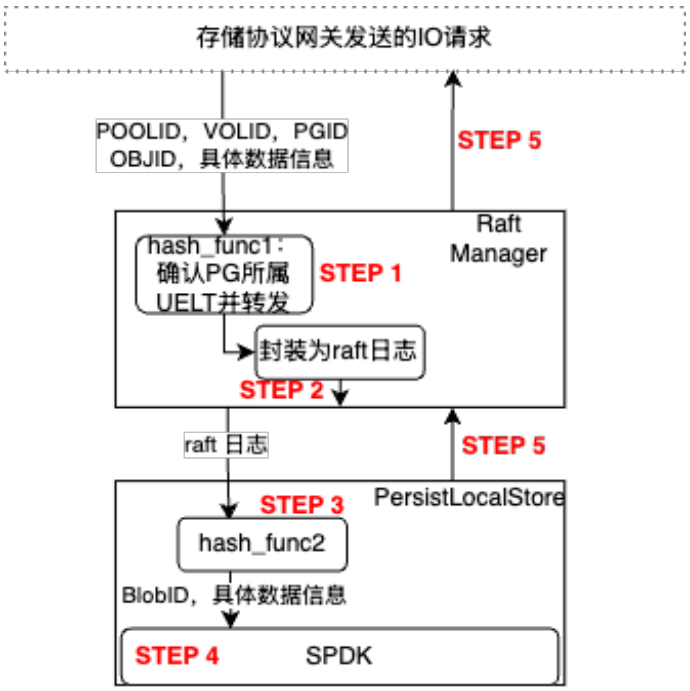

图 6.9　单节点内 RaftStoreOSD 的写操作处理

Figure 6.9　Write Operation Handling of RaftStoreOSD Within a Single Node

6.6　实验结果与分析

6.6.1　实验环境

该实验在局域网内部搭建一个 3 节点集群，集群中各节点硬件配置和软件配置分别同第三章表 3.2 和表 3.3，各节点配置相同，集群拓扑同图 3.4。而用于对比的 Ceph 版本以及相关配置同第五章表 5.1。

6.6.2　实验方法

表 6.1　读写模型

Table 6.1　Read/Write Model

IO 模型 (30min)	并发	测试点
4KB 随机 70% 读 30% 写	8,32,64,128,256	IOPS、平均时延
1MB 顺序 70% 读 30% 写	1,2,4,8,16,32	带宽、平均时延

6.7 本章小结

本章基于前述章节所述研究取得的研究成果为基础实现了 RaftStoreOSD 模块。并且分别介绍了详细的整体架构设计、Raft 算法状态机设计与实现、元数据与业务数据内存结构和磁盘结构设计，并且通过对比实验进一步证实了基于 RaftStoreOSD 模块实现的 RBD 相较原生 RBD，具有较大的性能优势，为适合用户态技术的现代分布式存储架构提供了详细的实现参考。