浅析开源项目之 Ceph

前言

Ceph 是一个极其复杂的统一分布式存储系统，运维操作门槛高、稳定性不错，性能差强人意，虽然各大厂都在自研分布式存储，但 Ceph 是不可或缺的参考对象。本文参考了 Ceph 源代码以及网上各路大神文章，如有侵权，联系删除。简要分析 Ceph 的架构、重要的模块以及基于 Seastar 的未来规划，使读者对 Ceph 有一个大致清晰的认识。

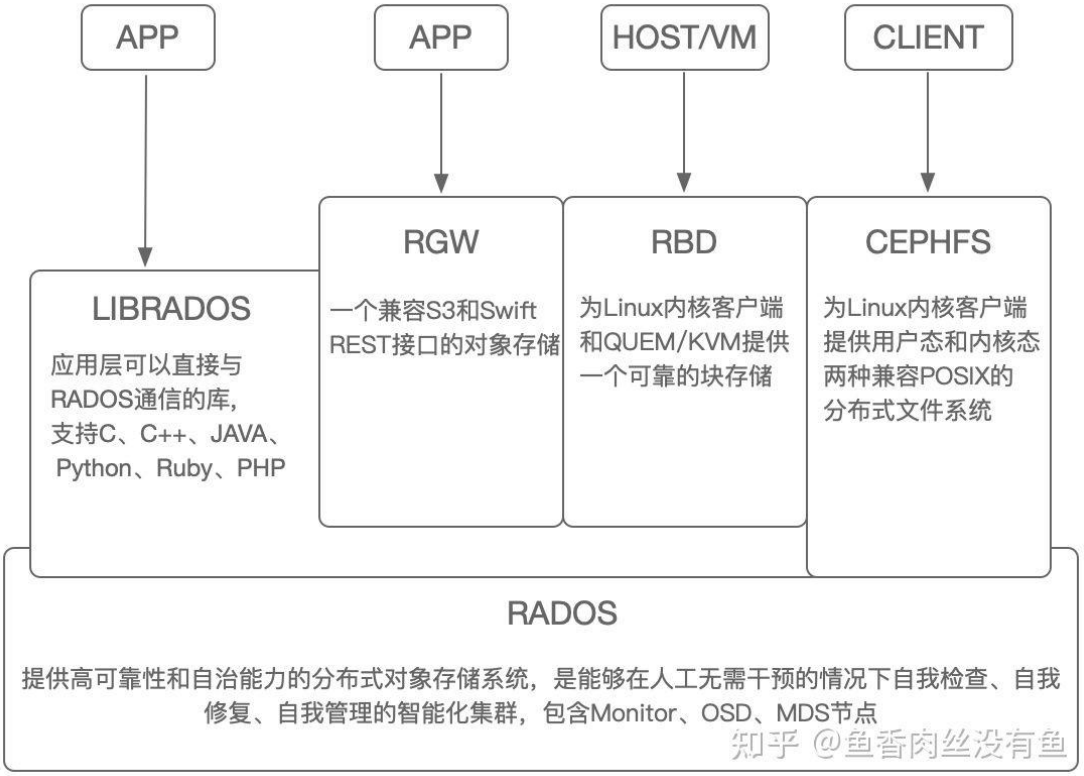
目录

1 Ceph 概述

Ceph 是由学术界(Sage Weil 博士论文)在 2006 年提出的一个开源的分布式存储系统的解决方案，最早致力于下一代高性能分布式文件存储，经过十多年的发展，还提供了块设备、对象存储 S3 的接口，成为了统一的分布式存储平台，进而成为开源社区存储领域的明星项目，得到了广泛的实际应用。

Ceph 是一个可靠的、自治的、可扩展的分布式存储系统，它支持文件存储、块存储、对象存储三种不同类型的存储，满足存储的多样性需求。整体架构如下：

- **接口层**：提供客户端访问存储层的的各种接口，支持 POSIX 文件接口、块设备接口、对象 S3 接口，以及用户可以自定义自己的接口。
- **Librados**：提供上层访问 RADOS 集群的各种库函数接口，libcephfs、librbd、librgw 都是 Librados 的客户端。
- **RADOS**：可靠的、自治的分布式对象存储，主要包含 Monitor、OSD、MDS 节点，提供了一个统一的底层分布式存储系统，支持逻辑存储池概念、副本存储和纠删码、自动恢复、自动 rebalance、数据一致性校验、分级缓存、基于 dmClock 的 QoS 等核心功能。



2 核心组件

- **CephFS**：Ceph File System，Ceph 对外提供的文件系统服务，MDS 来保存 CephFS 的元数据信息，数据写入 Rados 集群。
- **RBD**：Rados Block Device，Ceph 对外提供的块设备服务，Ceph 里称为 Image，元数据很少，保存在特定的 Rados 对象和扩展属性中，数据写入 Rados 集群。
- **RGW**：Rados Gateway，Ceph 对外提供的对象存储服务，支持 S3、Swift 协议，元数据保存在特定的 Pool 里面，数据写入 Rados 集群。

- **Monitor**：保存了 MONMap、OSDMap、CRUSHMap、MDSMap 等各种 Map 等集群元数据信息。一个 Ceph 集群通常需要 3 个 Mon 节点，通过 Paxos 协议同步集群元数据。
- **OSD**：Object Storage Device，负责处理客户端读写请求的守护进程。一个 Ceph 集群包含多个 OSD 节点，每块磁盘一个 OSD 进程，通过基于 PGLog 的一致性协议来同步数据。
- **MDS**：Ceph Metadata Server，文件存储的元数据管理进程，CephFS 依赖的元数据服务，对外提供 POSIX 文件接口，不是 Rados 集群必须的。
- **MGR**：Ceph Manager，负责跟踪运行时指标以及集群的运行状态，减轻 Mon 负担，不是 Rados 集群必须的。
- **Message**：网络模块，目前支持 Epoll、DPDK(剥离了 seastar 的网络模块，不使用其 share-nothing 的框架)、RDMA，默认 Epoll。
- **ObjectStore**：存储引擎，目前支持 FileStore、BlueStore、KVStore、MemStore，提供类 POSIX 接口、支持事务，默认 BlueStore。
- **CRUSH**：数据分布算法，秉承着无需查表，算算就好的理念，极大的减轻了元数据负担(但是感觉过于执着减少元数据了，参考意义并不是很大)，但同时数据分布不均，不过已有 CRUSH 优化 Paper。

- **SCRUB**：一致性检查机制，提供 scrub(只扫描元数据)、deep_scrub(元数据和数据都扫描)两种方式。
- **Pool**：抽象的存储池，可以配置不同的故障域也即 CRUSH 规则，包含多个 PG，目前类型支持副本池和纠删池。
- **PG**：Placement Group，对象的集合，可以更好的分配和管理数据，同一个 PG 的读写是串行的，一个 OSD 上一般承载 200 个 PG，目前类型支持副本 PG 和纠删 PG。
- **PGLog**：PG 对应的多个 OSD 通过基于 PGLog 的一致性协议来同步数据，仅保存部分操作的 oplog，扩缩容、宕机引起的数据迁移过程无需 Mon 干预，通过 PG 的 Peering、Recovery、Backfill 机制来自动处理。
- **Object**：Ceph-Rados 存储集群的基本单元，类似文件系统的文件，包含元数据和数据，支持条带化、稀疏写、随机读写等和文件系统文件差不多的功能，默认 4MB。

3 IO 流程

此处以 RBD 块设备为例简要介绍 Ceph 的 IO 流程。

1. 用户创建一个 Pool，并指定 PG 的数量。
2. 创建 Pool/Image，挂载 RBD 设备，映射成一块磁盘。
3. 用户写磁盘，将转换为对 librbd 的调用。

4. librbd 对用户写入的数据进行切块并调用 librados，每个块是一个 object，默认 4MB。
5. librados 进行 stable_hash 算法计算 object 所属的 PG，然后再输入 pg_id 和 CRUSHMap，根据 CRUSH 算法计算出 PG 归属的 OSD 集合。
6. librados 将 object 异步发送到 Primary PG，Primary PG 将请求发送到 Secondary PG。
7. PG 所属的 OSD 在接收到对应的 IO 请求之后，调用 ObjectStore 存储引擎层提供的接口进行 IO。
8. 最终所有副本都写入完成才返回成功。

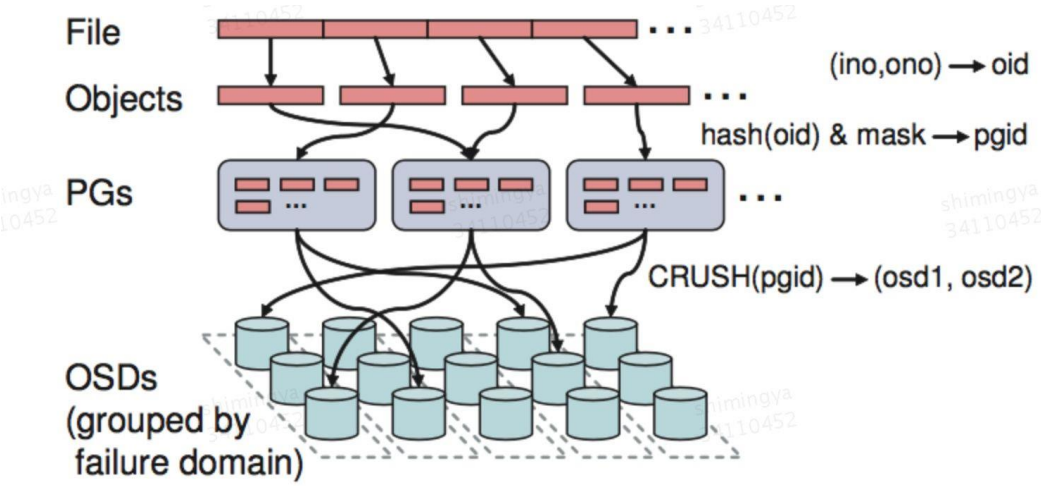


Figure 3: Files are striped across many objects, grouped into *placement groups* (PGs), and distributed to OSDs via CRUSH, a specialized replica placement function.

Ceph 的 IO 通常都是异步的，所以往往伴随着各种回调，以 FileStore 为例看下 ObjectStore 层面的回调：

1. on_journal：数据写入到 journal，通常通过 DirectIO + Libaio 的方式，Journal 的数据是 sync 到磁盘上的。
2. on_readable：数据写入 Journal 且写入 Pagecache 中，返回客户端可读。
3. on_commit：Pagecache 中的数据 sync 到磁盘上，返回客户端真正写成功。

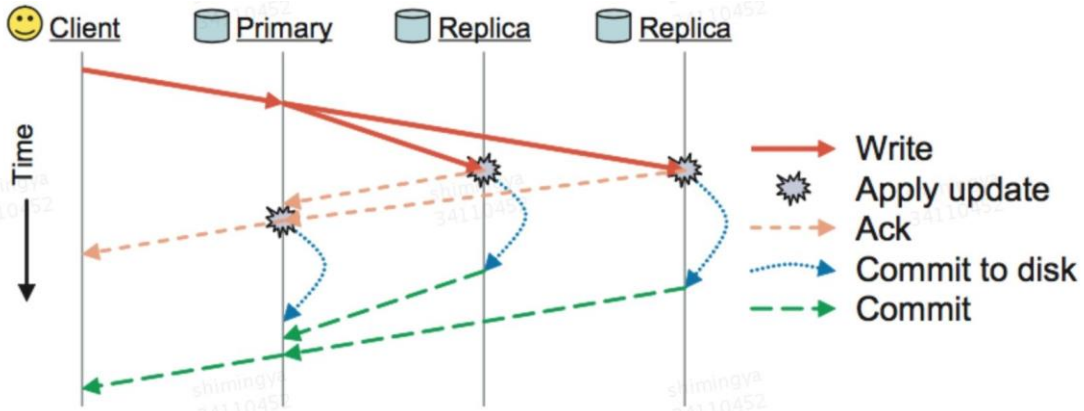


Figure 4: RADOS responds with an *ack* after the write has been applied to the buffer caches on all OSDs replicating the object. Only after it has been safely committed to disk is a final *commit* notification sent to the client.

4 IO 顺序性

分布式系统中通常需要考虑对象读写的顺序性和并发性，如果两个对象没有共享资源，那么就可以并发访问，如果有共享资源就需要加锁操作。对于同一个对象的并发读写来说，通常是通过队列、锁、版本控制等机制来进行并发控制，以免数据错乱，Ceph 中对象的并发读写也是通过队列和锁机制来保证的。

PG

Ceph 引入 PG 逻辑概念来对对象进行分组，不同 PG 之间的对象是可以并发读写的，单个 PG 之间的对象不能并发读写，也即理论上 PG 越多并发的对象也越多，但对于系统的负载也高。

不同对象的并发控制

落在不同 PG 的不同对象是可以并发读写的，落在统一 PG 的不同对象，在 OSD 处理线程中会对 PG 加锁，放进 PG 队列里，一直等到调用 queue_transactions 把 OSD 的事务提交到 ObjectStore 层才释放 PG 的锁，也即

对于同一个 PG 里的不同对象，是通过 PG 锁来进行并发控制，不过这个过程中不会涉及到对象的 IO，所以不太会影响效率。

同一对象的并发控制

同一对象的并发控制是通过 PG 锁实现的，但是在使用场景上要分为单客户端、多客户端。

1. 单客户端：单客户端对同一个对象的更新操作是串行的，客户端发送更新请求的顺序和服务端收到请求的顺序是一致的。

2. 多客户端：多客户端对同一个对象的并发访问类似于 NFS 的场景，RADOS 以及 RBD 是不能保证的，CephFS 理论上应该可以。

所以接下来主要讨论单客户端下同一对象的异步并发更新。

Message 层顺序性

1. TCP 层是通过消息序列号来保证一条连接上消息的顺序性。
2. Ceph Message 层也是通过全局唯一的 tid 来保证消息的顺序性。

PG 层顺序性

从 Message 层取到消息进行处理时，OSD 处理 OP 时划分了多个 shard，每个 shard 可以配置多个线程，PG 通过哈希的方式映射到不同的 shard 里面。OSD 在处理 PG 时，从拿到消息就会 PG 加了写锁，放入到 PG 的 OpSequencer 队列，等到把 OP 请求下发到 ObjectStore 端才释放写锁。对于同一个对象的并发读写通过对象锁来控制。

对同一个对象进行写操作会加 write_lock，对同一个对象的读操作会加 read_lock，也就是读写锁，读写是互斥的。写锁从 queue_transactions 开始到数据写入到 Pagecache 结束。

对同一个对象上的并发写操作，实际上并不会发生，因为放入 PG 队列是有序的，第一次写从 PG 取出放到 ObjectStore 层之后就会释放锁，然后再把第二次写从 PG 取出放入到 ObjectStore 层，取出写 OP 放到 ObjectStore 层都是调的异步写的接口，这就需要 ObjectStore 层来保证两次写的顺序性了。

ObjectStore 层顺序性

ObjectStore 支持 FileStore、BlueStore，也都需要保证 IO 顺序性。对于写请求，到达 ObjectStore 层之后，会获取 OpSequencer(每个 PG 一个，用来保证 PG 内 OP 顺序)。

FileStore：对于写事务 OP 来说(都有一个唯一递增的 seq)，会按照顺序放进 writeq 队列，然后 write_thread 线程通过 Libaio 将数据写入到 Journal 里面，此时数据已经是 on_disk 但不可读，已完成 OP 的 seq 序号按序放到 journal 的 finisher 队列里（因为 Libaio 并不保证顺序，会出现先提交的 IO 后完成，因此采用 op 的 seq 序号来保证完成后处理的顺序），如果某个 op 之前的 op 还未完成，那么这个 op 会等到它之前的 op 都完成后才一起放到 finisher 队列里，然后把数据写入到 Pagecache 和 sync 到数据盘上。

BlueStore：bluestore 在拿到写 OP 时会先通过 BlockDevice 提供的异步写(Libaio/SPDK/io_uring)接口先把数据写到数据盘，然后再通过 RocksDB 的 WriteBatch 接口批量的写元数据和磁盘分配器信息到

RocksDB。由于也是通过异步写接口写的，也需要等待该 OP 之前的 OP 都完成，才能写元数据到 RocksDB。

5 PG 一致性协议

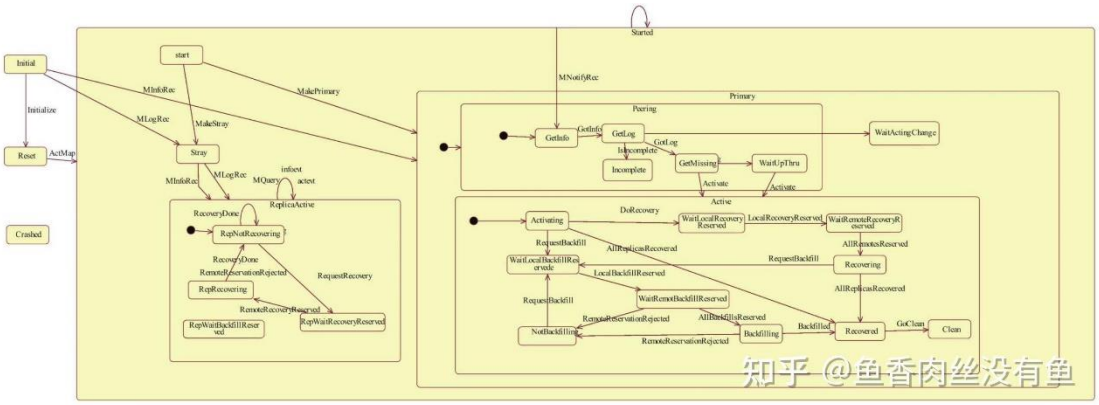
在 Ceph 的设计和实现中，自动数据迁移、自动数据均衡等各种特性都是以 PG 为基础实现的，PG 是最复杂和最难理解的概念，Ceph 也基于 PG 实现了数据的多副本和纠删码存储。基于 PG LOG 的一致性协议也类似于 Raft 实现了强一致性。

5.1 StateMachine

PG 有 20 多种状态，状态的多样性也反映了功能的多样性和复杂性。PG 状态的变化通过事件驱动的状态机来驱动，比如集群状态的变化，OSD 加入、删除、宕机、恢复 、创建 Pool 等，最终都会转换为一系列的状态机事件，从而驱动状态机在不同状态之间跳转和执行处理。

- Active：活跃态，PG 可以正常处理来自客户端的读写请求，PG 正常的状态应该是 Active+Clean 的。
- Unactive：非活跃态，PG 不能处理读写请求。
- Clean：干净态，PG 当前不存在修复对象，Acting Set 和 Up Set 内容一致，并且大小等于存储池的副本数。
- Peering：类似 Raft 的 Leader 选举，使一个 PG 内的 OSD 达成一致，不涉及数据迁移等操作。

- Recovering：正在恢复态，集群正在执行迁移或恢复某些对象的副本。
- Backfilling：正在后台填充态，backfill 是 recovery 的一种特殊场景，指 peering 完成后，如果基于当前权威日志无法对 Peers 内的 OSD 实施增量同步(OSD 离线太久，新的 OSD 加入），则通过完全拷贝当前 Primary 所有对象的方式进行全量同步。
- Degraded：降级状态，Peering 完成后，PG 检测到有 OSD 有需要被同步或修复的对象，或者当前 ActingSet 小于存储池副本数。
- Undersized：PG 当前 Acting Set 小于存储池副本数。ceph 默认 3 副本，min_size 参数通常为 2，即副本数>=2 时就可以进行 IO，否则阻塞 IO。
- Scrubing：PG 正在进行对象的一致性扫描。
- 只有 Active 状态的 PG 才能进行 IO，可能会有 active+clean(最佳)、active+unclean(小毛病)、active+degraded(小毛病)等状态，小毛病不影响 IO。



为了避免全是文字，网上找了张图，如有侵权，联系删除。

5.2 Failover Overview

故障检测：Ceph 分为 MON 集群和 OSD 集群两部分，MON 集群管理者整个集群的成员状态，将 OSD 的信息存放在 OSDMap 中，OSD 定期向 MON 和 Peer OSD 发送心跳包，声明自己处于在线状态。MON 接收来自 OSD 的心跳信息确认 OSD 在线，同时也接收来自 OSD 对于 Peer OSD 的故障检测。当 MON 判断某个 OSD 节点离线后，便将最新的 OSDMap 通过心跳随机的发送给 OSD，当 Client 或者 OSD 处理 IO 请求时发现自身的 OSDMap 版本低于对方，便会向 MON 请求最新的 OSDMap，这种 Lasy 的更新方式，经过一段时间的传播之后，整个集群都会收到最新的 OSDMap。

确定恢复数据：OSD 在收到 OSDMap 的更新消息后，会扫描该 OSD 下所有的 PG，如果发现某些 PG 已经不属于自己，则会删掉其数据。如果该 OSD 上的 PG 是 Primary PG 的话，将会进行 PG Peering 操作。在 Peering 过程中，会根据 PGLog 检查多个副本的一致性，并

计算 PG 的不同副本的数据缺失情况，PG 对应的副本 OSD 都会得到一份对象缺失列表，然后进行后续的 Recovery，如果是新节点加入、不足以根据 PGLog 来 Recovery 等情况，则会进行 Backfill，来恢复整份数据。

数据恢复：在 PG Peering 过程中会暂停所有的 IO，等 Peering 完成后，PG 会进入 Active 状态，此时便可以接收数据的 IO 请求，然后根据 Peering 的信息来决定进行 Recovery 还是 Backfill。对于 Replica PG 缺失的数据 Primary PG 会通过 Push 来推送，对于 Primary PG 自身缺少的数据会通过 Pull 方式从其他 Replicate PG 拉取。在 Recovery 过程中，恢复的粒度是 4M 对象，对于无法通过 PGlog 来恢复的，则进行 Backfill 进行数据的全量拷贝，等到数据恢复完成后，PG 的状态会标记为 Clean 即所有副本数据保持一致。

5.3 PG Peering

PG 的 Peering 是使一个 PG 内的所有 OSD 达成一致的过程，相关重要概念如下：

- **up set：**pg 对应的副本列表，也即通过 CRUSH 算法选出来的 3 个副本列表，第一个为 primary，其他的为 replica。
- **active set：**对外处理 IO 的副本列表，通常和 up set 一致，当恢复时可能会存在临时 PG，则 active set 为临时 PG 的副本集

合，用于对外提供正常 IO，当完成恢复后，active set 调整为 up set。

- **pg_temp：**临时的 PG，当 CRUSH 算法产生新的 up set 的 primary 无法承担起职责(新加入的 OSD 或者 PGLog 过于落后的 OSD 成为了 primary，也即需要 backfill 的 primary 需要申请临时 PG，recovery 的 primary 不需要申请临时 PG)，osd 就会向 mon 申请一个临时的 PG 用于数据正常 IO 和恢复，Ceph 做了优化是在进行 CRUSH 时就根据集群信息选择是否预填充 pg_tmp，从而减少 Peering 的时间。此时处于 Remapped 状态，等到数据同步完成，需要取消 pg_tmp，再次通过 Peering 将 active_set 切回 up_set。
- **epoch：**每个 OSDMap 都会有一个递增的版本，值越大版本越新，当集群中 OSD 发生变化时，就会产生新的 OSDMap。
- **pg log：**保存操作的记录，是用于数据恢复的重要结构。并不会保存所有的 op log，默认 3000 条，当有数据需要恢复的时候就会保存 10000 条。
- **Interval：**每个 PG 都有 Interval(epoch 的操作序列)，每次 OSD 获取到新的 OSDMap 时，如果发现 **up set、up primary、active set、active primary** 没有改变，则 Interval 不用改变，否则就要生成新的 current interval，之前的变成 past_interval，只要该 PG 内部的 OSD 不发生变化，Interval 就不会变化。

主要包含三个步骤：

1. **GetInfo：**作用为确定参与 peering 过程的 osd 集合。主 OSD 会获取该 PG 对应的所有 OSD 的 pg_info 信息放入 peer_info。
2. **GetLog：**作用为选取权威日志。根据各个副本 OSD 的 pg_info 信息比较，选取一个具有权威日志的 OSD，如果主 OSD 不具备权威日志，那么就从该具有权威日志的 OSD 拉取权威日志，拉取完成之后进行合并就具有了权威日志，如果 primary 自身具有权威日志，则不用合并，否则合并的过程如下：
    1. 拉取过来的日志比 primary 具有更老的日志条目：追加到 primary 本地日志尾部即可。
    2. 拉取过来的日志比 primary 具有更新的日志条目：追加到 primary 本地日志头部即可。
    3. 合并的过程中，primary 如果发现自己有对象需要修复，便会将其加入到 missing 列表。
3. **GetMissing：**获取需要恢复的 object 集合。主 OSD 拉取其他从 OSD 的 PGLog，与自身权威日志进行对比，计算该 OSD 缺失的 object 集合。

5.4 Recovery/Backfill

Peering 进行之后，如果 Primary 检测到自身或者任意一个 Peer 需要修复对象，则进入 Recovery 状态，为了影响外部 IO，也会限制恢复的速度以及每个 OSD 上能够同时恢复的 PG 数量。Recovery 一共有两种状态：

1. **Pull：**如果 Primary 自身存在待恢复对象，则按照 missing 列表寻找合适的副本拉取修复对象到本地然后修复。
2. **Push：**如果 Primary 检测到其 Replica 存在待恢复对象，则主动推动待修复对象到 Replica，然后由 Replica 自身修复。

通常总是先执行 Pull 再执行 Push，即先修复 Primary 再修复 Replica，因为 Primary 承担了客户端的读写，需要优先进行修复，修复情况大致如下：

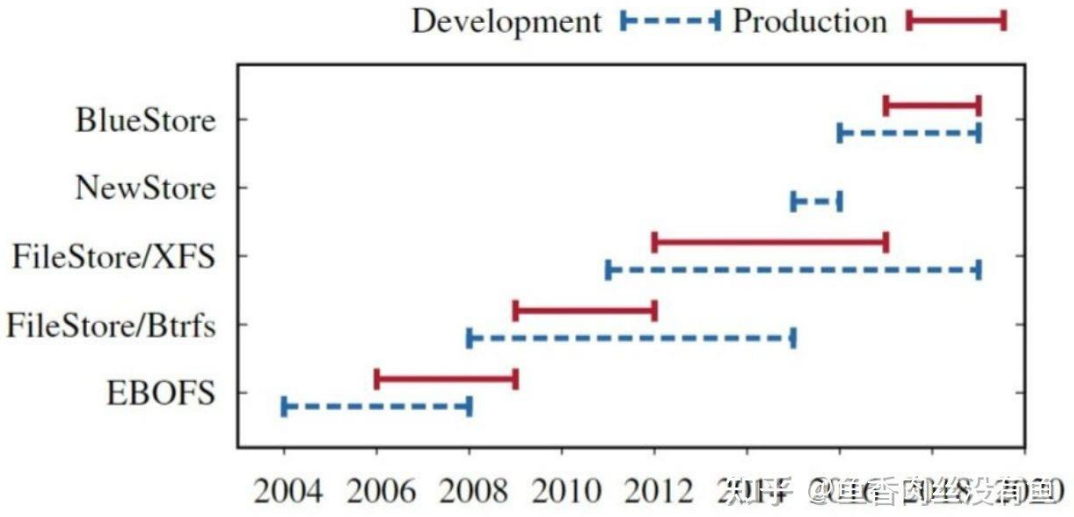
1. 客户端 IO 和内部恢复 IO 可以同时进行。
2. 读写的对象不在恢复列表中：按照正常 IO 即可。
3. 读取的对象在恢复列表中：如果 primary 有则可以直接读取，如果没有需要优先恢复该对象，然后读取。
4. 写入的对象在恢复列表中：优先恢复该对象，然后写入。
5. backfill 则是 primary 遍历当前所有的对象，将他们全量拷贝到 backfill 的 PG 中。
6. 恢复完成后，会重新进行 Peering，是 active set 和 up set 保持一致，变为 active + clean 状态。

在恢复对象时，由于 PGLog 并未记录关于对象修改的详细信息(offset、length 等)，所以目前对象的修复都是全量对象(4M)拷贝，不过社区已经支持部分对象修复。

同时在恢复对象时，由于 ObjectStore 支持覆盖写，所以在对象上新的写不能丢弃老的对象，需要等老的对象恢复完之后，才能进行该对象新的写入，不过社区已经支持异步恢复。

6 引擎概述

Ceph 提供存储功能的核心组件是 RADOS 集群，最终都是以对象存储的形式对外提供服务。但在底层的内部实现中，Ceph 的后端存储引擎在近十年来经历了许多变化。现如今的 Ceph 系统中仍然提供的后端存储引擎有 FileStore、BlueStore。但该三种存储引擎都是近年来才提出并设计实现的。Ceph 的存储引擎也先后经历了 EBOFS-->FileStore/btrfs-->FileStore/xfs-->NewStore-->BlueStore。同时 Ceph 需要支持文件存储，所以其存储引擎提供的接口是类 POSIX 的，存储引擎操作的对象也具有类似文件系统的语义，也具有其自己的元数据。

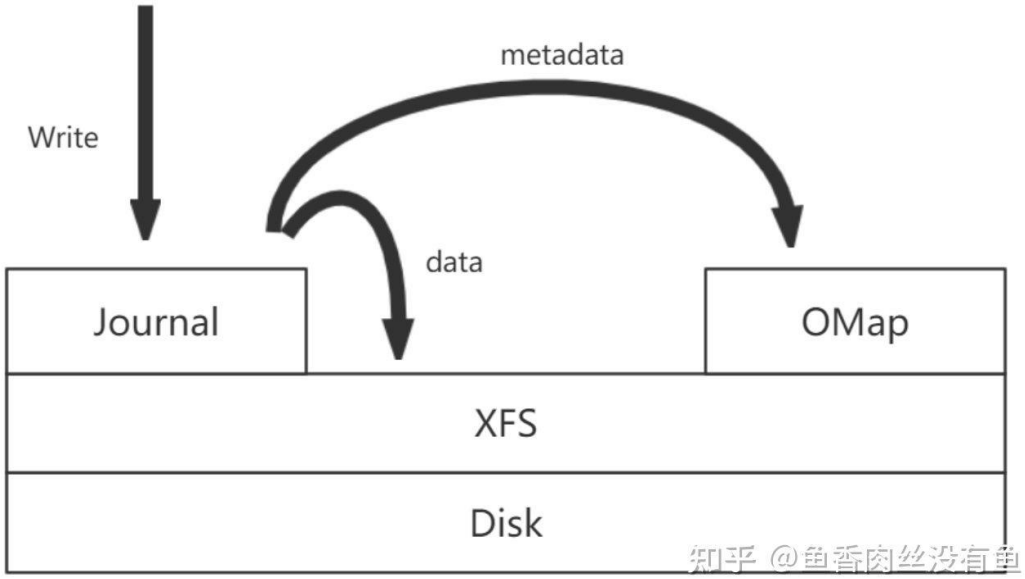


7 FileStore

FileStore 是 Ceph 基于文件系统的最早在生成环境比较稳定的单机存储引擎，虽然后来出现了 BlueStore，但在一些场景中仍然不能代替 FileStore，比如在全是 HDD 的场景中 FileStore 可以使用 NVME 盘做元数据和数据的读写 Cache，从而加速 IO，BlueStore 就只能加速元数据 IO。

7.1 架构设计

FileStore 是基于文件系统的，为了维护数据的一致性，写入之前数据会先写 Journal，然后再写到文件系统，会有一倍的写放大。不过 Journal 也起到了随机写转换为顺序写、支持事务的作用。



引用网上图片，如有侵权，联系删除。

7.2 对外接口

对象的元数据使用 KV 形式保存，主要有两种保存方式：

- xattrs：保存在本地文件系统的扩展属性中，一般都有大小的限制。
- omap：object map，保存在 LevelDB/RocksDB 中。

有些文件系统不支持扩展属性，或者扩展属性大小有限制。一般情况下 xattr 保存一些比较小且经常访问的元数据，omap 保存一些大的不经常访问的元数据。

同时 ObjectStore 使用 Transaction 类来实现相关的操作，将元数据和数据封装到 bufferlist 里面，然后写 Journal。大致包含

OP_TOUCH、OP_WRITE、OP_ZERO、OP_CLONE 等 42 种事务操作。提供的对外接口大致有：

ObjectStore 本身的接口：mount、umount、fsck、repair、mkfs 等。

Object 本身的接口：read、write、omap、xattrs、snapshot 等。

7.3 日志类型

在 FileStore 的实现中，根据不同的日志提交方式，有两种不同的日志类型：

- Journal writeahead：先提交数据到 Journal 上(通常配置成一块 SSD 磁盘)，然后再写入到 Pagecache，最后 sync 到数据盘上。适用于 XFS、EXT4 等不支持快照的文件系统，是 FileStore 默认的实现方式。
- Journal parallel：数据提交到 Journal 和 sync 到数据盘并行进行，没有完成的先后顺序，适用于 BTRFS、ZFS 等支持快照的文件系统，由于文件系统支持快照，当写数据盘出错，数据不一致时，文件系统只需要回滚到上一次快照，并 replay 从上次快照开始的日志就可以，性能要比 writeahead 高，但是 Linux 下 BTRFS 和 ZFS 不稳定，线上生产环境几乎没人用。

日志处理有三个阶段：

1. 日志提交(journal submit)：数据写入到日志盘，通常使用 DirectIO+Libaio，一个单独的 write_thread 不断从队列取任务执行。
2. 日志应用(journal apply)：日志对应的修改更新到文件系统的文件上，此过程仅仅是写入到了 Pagecache。
3. 日志同步(journal commit)：将文件系统的 Pagecache 脏页 sync 到磁盘上，此时数据已经持久化到数据盘，Journal 便可以删除对应的数据，释放空间。

7.4 幂等操作

在机器异常宕机的情况下，Journal 中的数据不一定全部都 sync 到了数据盘上，有可能一部分还在 Pagecache，此时便需要在 OSD 重启时保证数据的一致性，对 Journal 做 replay。FileStore 将已经 sync 到数据盘的序列号记录在 commit_op_seq 中，replay 的时候从 commit_op_seq 开始即可。

但是在 replay 的时候，部分 op 可能已经 sync 到数据盘中，但是 commit_op_seq 却没有体现，序列化比其小，此时如果仍然 replay，可能会出现非幂等操作，导致数据不一致。

假设一个事务包含如下 3 个操作：

1. clone a 到 b。
2. 更新 a。
3. 更新 c。

假设上述操作都做完也已经持久化到数据盘上了，然后立马进程或者系统崩溃，此时 sync 线程还未来得及更新 commit_op_seq，重启回放时，第二次执行 clone 操作就会 clone 到 a 新的数据版本，就会发生不一致。

FileStore 在对象的属性中记录最后操作的三元组(序列号、事务编号、OP 编号)，因为 journal 提交的时候有一个唯一的序列号，通过这个序列号， 就可以找到提交时候的事务，然后根据事务编号和 OP 编号最终定位出最后操作的 OP。对于非幂等的操作，操作前先检查下，如果可以继续执行就执行操作，执行完之后设置一个 guard。这样对于非幂等操作，如果上次执行过， 肯定是有记录的，再一次执行的时候 check 就会失败，就不继续执行。

8 BlueStore

Ceph 早期的单机对象存储引擎是 FileStore，为了维护数据的一致性，写入之前数据会先写 Journal，然后再写到文件系统，会有一倍的写放大，而同时现在的文件系统一般都是日志型文件系统(ext 系列、xfs)，文件系统本身为了数据的一致性，也会写 Journal，此时便相当

于维护了两份 Journal；另外 FileStore 是针对 HDD 的，并没有对 SSD 作优化，随着 SSD 的普及，针对 SSD 优化的单机对象存储也被提上了日程，BlueStore 便由此应运而出。

BlueStore 最早在 Jewel 版本中引入，用于在 SSD 上替代传统的 FileStore。作为新一代的高性能对象存储后端，BlueStore 在设计中便充分考虑了对 SSD 以及 NVME 的适配。针对 FileStore 的缺陷，BlueStore 选择绕过文件系统，直接接管裸设备，直接进行对象数据 IO 操作，同时元数据存放在 RocksDB，大大缩短了整个对象存储的 IO 路径。BlueStore 可以理解为一个支持 ACID 事物型的本地日志文件系统。

8.1 架构设计

BlueStore 是一个事务型的本地日志文件系统。因为面向下一代全闪存阵列的设计，所以 BlueStore 在保证数据可靠性和一致性的前提下，需要尽可能的减小日志系统中双写带来的影响。全闪存阵列的存储介质的主要开销不再是磁盘寻址时间，而是数据传输时间。因此当一次写入的数据量超过一定规模后，写入 Journal 盘(SSD)的延时和直接写入数据盘(SSD)的延迟不再有明显优势，所以 Journal 的存在性便大大减弱了。但是要保证 OverWrite(覆盖写)的数据一致性，又不得不借助于 Journal，所以针对 Journal 设计的考量便变得尤为重要了。

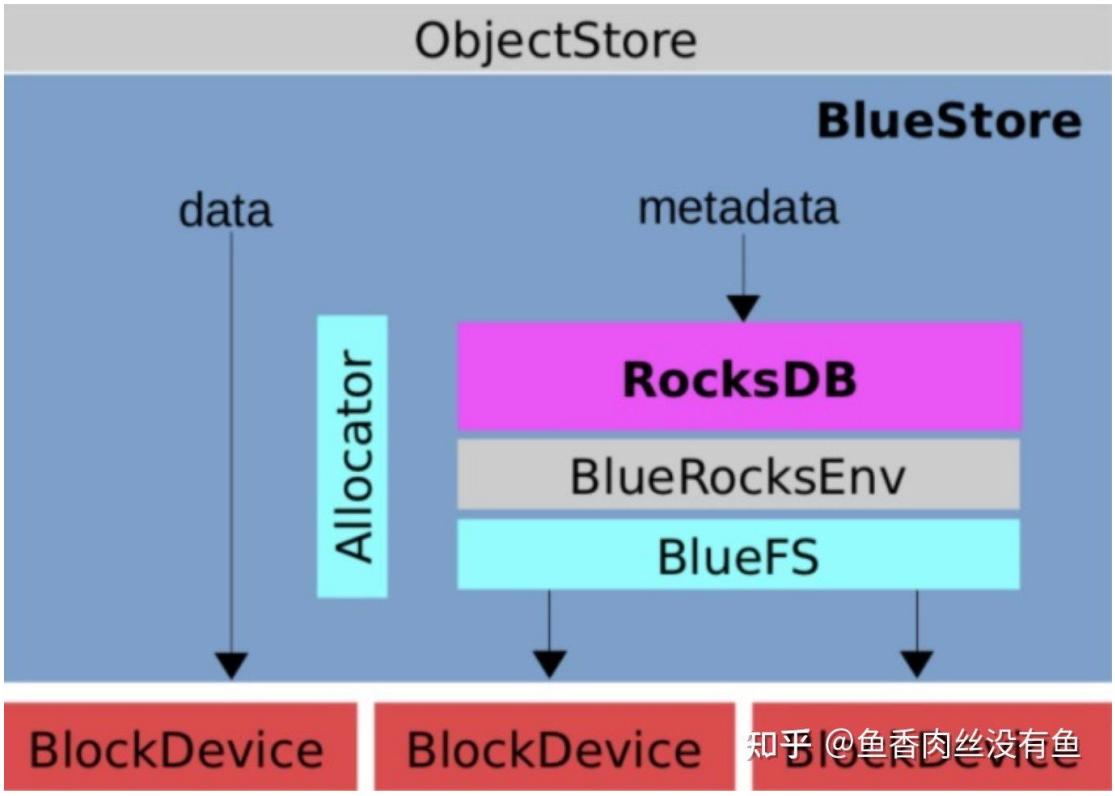
一个可行的方式是使用增量日志。针对大范围的覆盖写，只在其前后非磁盘块大小对齐的部分使用 Journal，即 RMW，其他部分直接重定向写 COW 即可。

RWM(Read-Modify-Write)：指当覆盖写发生时，如果本次改写的内容不足一个 BlockSize，那么需要先将对应的块读上来，然后再内存中将原内容和待修改内容合并 Merge，最后将新的块写到原来的位置。但是 RMW 也带来了两个问题：**一是**需要额外的读开销；**二是**如果磁盘中途掉电，会有数据损坏的风险。为此我们需要引入 Journal，先将待更新数据写入 Journal，然后再更新数据，最后再删除 Journal 对应的空间。

COW(Copy-On-Write)：指当覆盖写发生时，不是更新磁盘对应位置已有的内容，而是新分配一块空间，写入本次更新的内容，然后更新对应的地址指针，最后释放原有数据对应的磁盘空间。理论上 COW 可以解决 RMW 的两个问题，但是也带来了其他的问题：**一是** COW 机制破坏了数据在磁盘分布的物理连续性。经过多次 COW 后，读数据的顺序读将会便会随机读。**二是**针对小于块大小的覆盖写采用 COW 会得不偿失。**是因为**：**一是**将新的内容写入新的块后，原有的块仍然保留部分有效内容，不能释放无效空间，而且再次读的时候需要将两个块读出来做 Merge 操作，才能返回最终需要的数据，将大大影响读性能。**二是**存储系统一般元数据越多，功能越丰富，元数据越少，功能越简单。而且任何操作必然涉及元数据，所以元数据是系统中的热

点数据。COW 涉及空间重分配和地址重定向，将会引入更多的元数据，进而导致系统元数据无法全部缓存在内存里面，性能会大打折扣。

基于以上设计理念，BlueStore 的写策略综合运用了 COW 和 RMW 策略。**非覆盖写**直接分配空间写入即可；**块大小对齐的覆盖写**采用 COW 策略；**小于块大小的覆盖写**采用 RMW 策略。整体架构设计如下图：



- BlockDevice：物理块设备，使用 Libaio、SPDK、io_uring 操作裸设备，AsyncIO。
- RocksDB：存储对象元数据、对象扩展属性 Omap、磁盘分配器元数据。

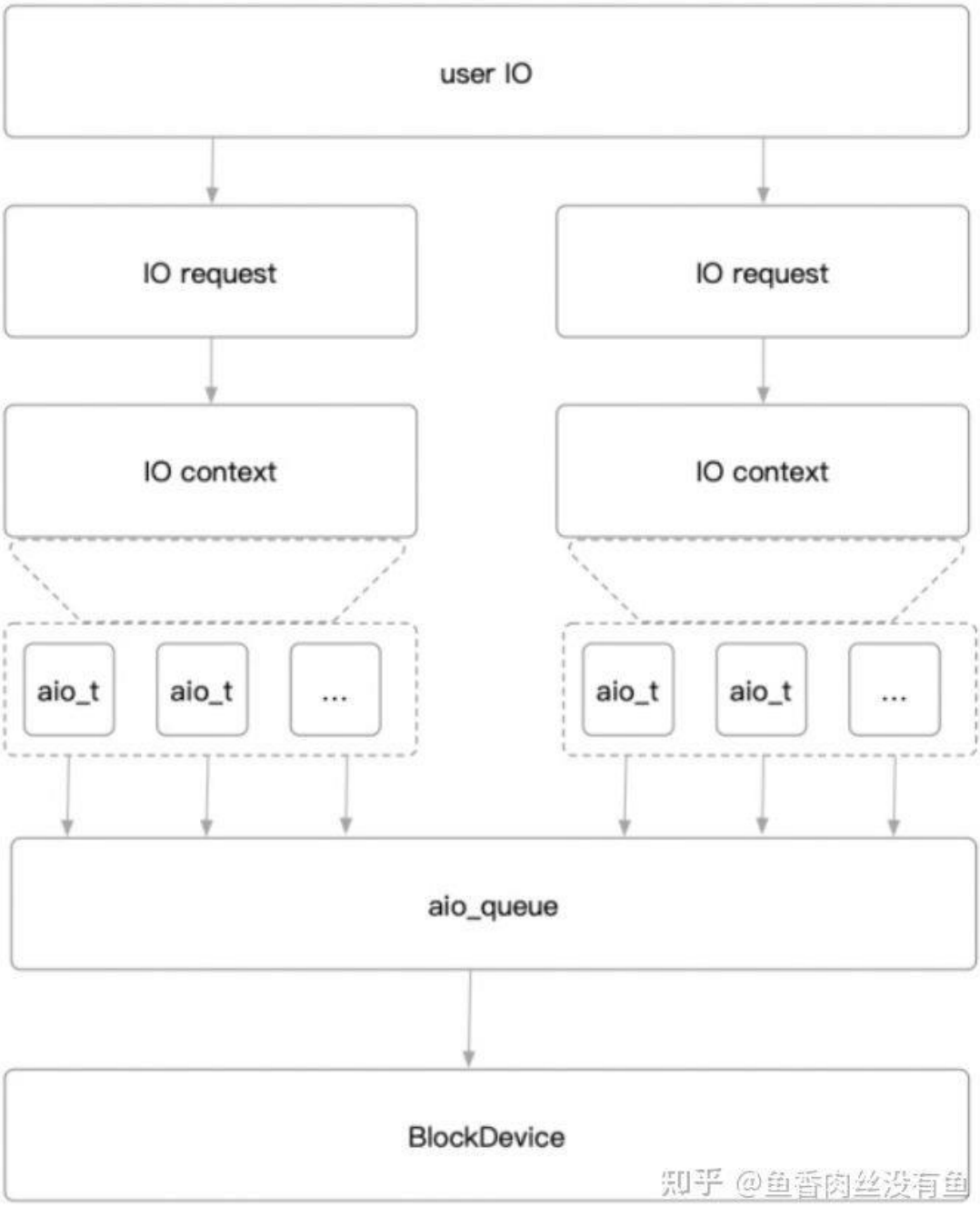
- BlueRocksEnv：抛弃了传统文件系统，封装 RocksDB 文件操作的接口。
- BlueFS：小型的 Append 文件系统，实现了 RocksDB::Env 接口，给 RocksDB 用。
- Allocator：磁盘分配器，负责高效的分配磁盘空间。
- Cache：实现了元数据和数据的缓存。

8.2 BlockDevice

Ceph 新的存储引擎 BlueStore 已成为默认的存储引擎，抛弃了对传统文件系统的依赖，直接管理裸设备，通过 Libaio 的方式进行读写。抽象出了 BlockDevice 基类，提供统一的操作接口，后端对应不同的设备类型的实现(Kernel、NVME、PMEM)。

- **KernelDevice：**通常使用 Libaio 或者 io_uring，适用于 HDD 和 SATA SSD。
- **NVMEDevice：**通常使用 SPDK 用户态 IO，提升 IOPS 缩短延迟，适用于 NVME 磁盘。
- **PMEMDevice：**当做磁盘来用，使用 libpmem 库来操作。

IO 架构图如下所示：



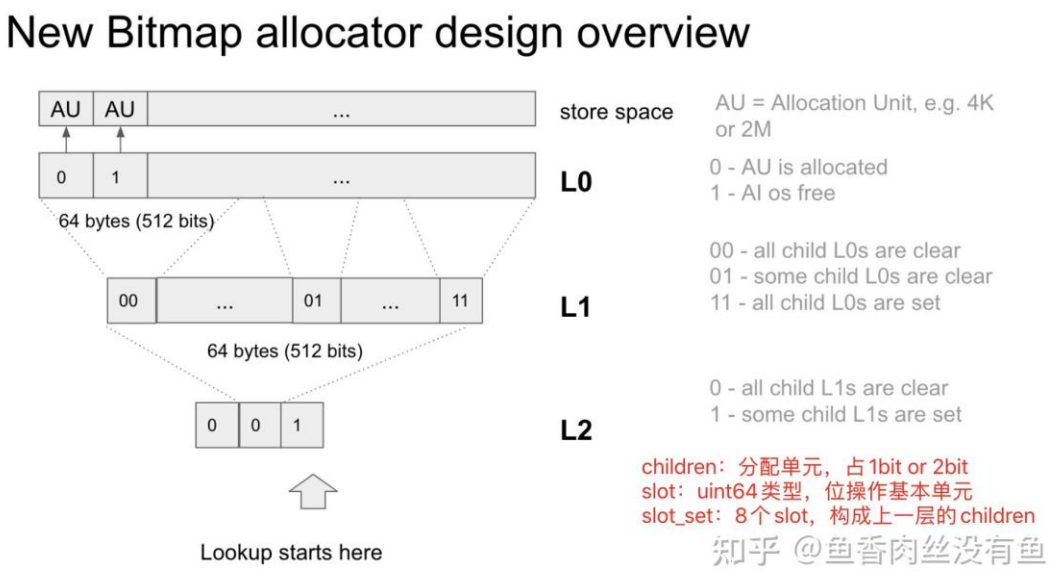
8.3 磁盘分配器

BlueStore 直接管理裸设备，那么必然面临着如何高效分配磁盘中的块。BlueStore 支持基于 Extent 和基于 BitMap 的两种磁盘分配策

略，有 BitMap 分配器(基于 Bitmap)和 Stupid 分配器(基于 Extent)，原则上都是尽量顺序分配而达到顺序写。

刚开始使用的是 BitMap 分配器，由于性能问题又切换到了 Stupid 分配器。之后 Igor Fedotov 大神重新设计和实现了新版本 BitMap 分配器，性能也比 Stupid 要好，默认的磁盘分配器又改回了 BitMap。

新版本 BitMap 分配器以 Tree-Like 的方式组织数据结构，整体分为 L0、L1、L2 三层。每一层都包含了完整的磁盘空间映射，只不过是 slot 以及 children 的粒度不同，这样可以加快查找，如下图所示：



新版本 Bitmap 分配器分配空间的大体策略如下：

1. 循环从 L2 中找到可以分配空间的 slot 以及 children 位置。
2. 在 L2 的 slot 以及 children 位置的基础上循环找到 L1 中可以分配空间的 slot 以及 children 位置。

3. 在 L1 的 slot 以及 children 位置的基础上循环找到 L0 中可以分配空间的 slot 以及 children 位置。
4. 在 1-3 步骤中保存分配空间的结果以及设置每层对应位置分配的标志位。

新版本 Bitmap 分配器整体架构设计有以下几点优势：

1. Allocator 避免在内存中使用指针和树形结构，使用 vector 连续的内存空间。
2. Allocator 充分利用 64 位机器 CPU 缓存的特性，最大程序的提高性能。
3. Allocator 操作的单元是 64 bit，而不是在单个 bit 上操作。
4. Allocator 使用 3 级树状结构，可以更快的查找空闲空间。
5. Allocator 在初始化时 L0、L1、L2 三级 BitMap 就占用了固定的内存大小。
6. Allocator 可以支持并发的分配空闲，锁定 L2 的 children(bit) 即可，暂未实现。

BlueStore 直接管理裸设备，需要自行管理空间的分配和释放。Stupid 和 Bitmap 分配器的结果是保存在内存中的，分配结果的持久化是通过 FreelistManager 来做的。

FreelistManager 最开始有 extent 和 bitmap 两种实现，现在默认为 bitmap 实现，extent 的实现已经废弃。空闲空间持久化到磁盘也是

通过 RocksDB 的 Batch 写入的。FreelistManager 将 block 按一定数量组成段，每个段对应一个 k/v 键值对，key 为第一个 block 在磁盘物理地址空间的 offset，value 为段内每个 block 的状态，即由 0/1 组成的位图，1 为空闲，0 为使用，这样可以通过与 1 进行异或运算，将分配和回收空间两种操作统一起来。

8.4 BlueFS

RocksDB 不支持对裸设备的直接操作，文件的读写必须实现 rocksdb::EnvWrapper 接口，RocksDB 默认实现有 POSIX 文件系统的读写接口。而 POSIX 文件系统作为通用的文件系统，其很多功能对于 RocksDB 来说并不是必须的，**同时 RocksDB 文件结构层次比较简单，不需要复杂的目录树，对文件系统的使用也比较简单，只使用追加写以及顺序读随机读。**为了进一步提升 RocksDB 的性能，需要对文件系统的功能进行裁剪，而更彻底的办法就是考虑 RocksDB 的场景量身定制一套本地文件系统，BlueFS 也就应运而生。相对于 POSIX 文件系统有以下几个优点：

1. 元数据结构简单，使用两个 map(dir_map、file_map)即可管理文件的所有元数据。
2. 由于 RocksDB 只需要追加写，所以每次分配物理空间时进行提前预分配，一方面减少空间分配的次数，另一方面做到较好的空间连续性。

3. 由于 RocksDB 的文件数量较少，可以将文件的元数据全部加载到内存，从而提高读取性能。
4. 多设备支持，BlueFS 将存储空间划分了 3 个层次：Slow 慢速空间(存放 BlueStore 数据)、DB 高速空间(存放 sstable)、WAL 超高速空间(存放 WAL、自身 Journal)，空间不足或空间不存在时可自动降级到下一层空间。
5. 新型硬件支持，抽象出了 block_device，可以支持 Libaio、io_uring、SPDK、PMEM、NVME-ZNS。

接口功能

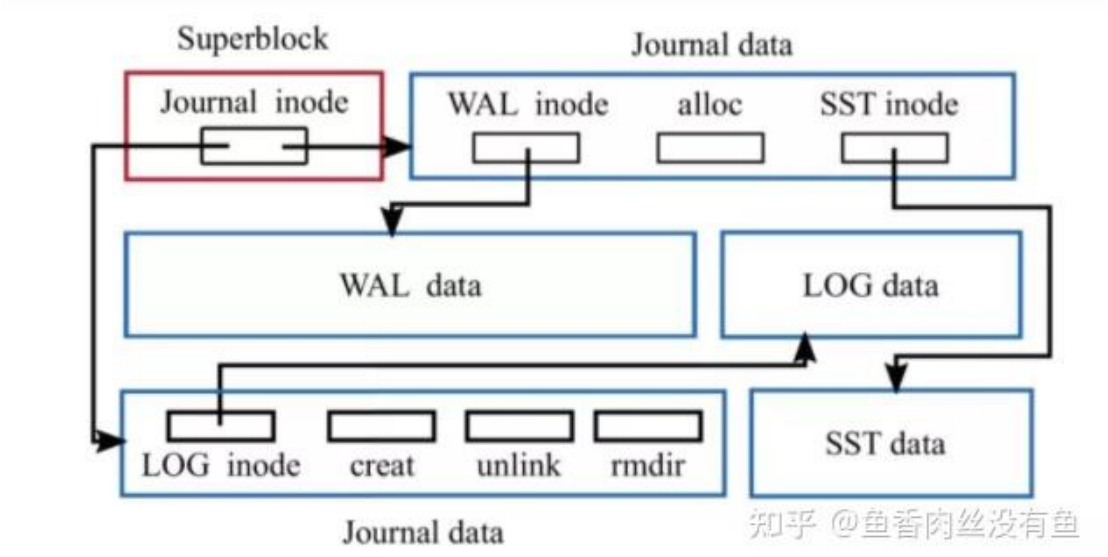
RocksDB 是通过 BlueRocksEnv 来使用 BlueFS 的，BlueRocksEnv 实现了文件读写和目录操作，其他的都继承自 rocksdb::EnvWrapper。

- 文件操作：追加写、顺序读(适用于 WAL 的读，也会进行预读)、随机读(sstable 的读，不会进行预读)、重命名、sync、文件锁。
- 目录操作：目录的创建、删除、遍历，目录只有一级，即 /a 、/a/b、/a/b/c 为同一级目录，整体元数据 map 可表示为：map<string(目录名), map<string(文件名), file_info(文件元数据)>>。

磁盘布局

BlueFS 的数据结构比较简单，主要包含三部分，superblock、journal、data。

- superblock：主要存放 BlueFS 的全局信息以及日志的信息，其位置固定在 BlueFS 的头部 4K。
- journal：存放元数据操作的日志记录，一般会预分配一块连续区域，写满以后从剩余空间再进行分配，在程序启动加载的时候逐条回放 journal 记录，从而将元数据加载到内存。也会对 journal 进行压缩，防止空间浪费、重放时间长。压缩时会遍历元数据，将元数据重新写到新的日志文件中，最后替换日志文件。
- data：实际的文件数据存放区域，每次写入时从剩余空间分配一块区域，存放的是一个个 sstable 文件的数据。



元数据

BlueFS 元数据：主要包含：superblock、dir_map、file_map、文件到物理地址的映射关系。

文件数据：每个文件的数据在物理空间上的地址由若干个 extents

表：一个 extent 包含 bdev、offset 和 length 三个元素，bdev 为设备标识，因为 BlueFS 将存储空间设备划分为三层：慢速（Slow）空间、高速（DB）空间、超高速（WAL），bdev 即标识此 extent 在哪块设备上，offset 表示此 extent 的数据在设备上的物理偏移地址，length 表示该块数据的长度。

```
struct bluefs_extent_t {
  uint64_t offset = 0;
  uint32_t length = 0;
  uint8_t bdev;
}

// 一个sstable 就是一个fnode
struct bluefs_fnode_t {
  uint64_t ino;
  uint64_t size;
  utime_t mtime;
  uint8_t prefer_bdev;
  mempool::bluefs::vector<bluefs_extent_t> extents;
  uint64_t allocated;
}
```

按照 9T 盘、sstable 8MB，文件元数据 80B 来算，所需内存 9 * 1024 * 1024 / 8 * 80 / 1024 / 1024 = 90MB，说明把元数据全部缓存到内存并不会占用过多的内存。

加载流程

1. 加载 superblock 到内存。
2. 初始化各存储空间的块分配器。
3. 日志回放建立 dir_map、file_map 来重建整体元数据。
4. 标记已分配空间：BlueFS 没有像 BlueStore 那样使用 FreelistManager 来持久化分配结果，因为 sstable 大小固定从不修改，所以 BlueFS 磁盘分配需求都是比较同意和固定的。会遍历每个文件的分配信息，然后移除相应的磁盘分配器中的空闲空间，防止已分配空间的重复分配。

读写数据

读数据：先从 dir_map 和 file_map 找到文件的 fnode(包含物理的 extent)，然后从对应设备的物理地址读取即可。

写数据：BlueFS 只提供 append 操作，所有文件都是追加写入。RocksDB 调用完 append 以后，数据并未真正落盘，而是先缓存在内存当中，只有调用 sync 接口时才会真正落盘。

1. open file for write

   打开文件句柄，如果文件不存在则创建新的文件，如果文件存在则会更新文件 fnode 中的 mtime，在事务 log_t 中添加更新操作，此时事务记录还不会持久化到 journal 中。

2. append file

   将数据追加到文件当中，此时数据缓存在内存当中，并未落盘，也未分配新的空间。

3. flush data(写数据)

   判断文件已分配剩余空间（fnode 中的 allocated - size）是否足够写入缓存数据，若不够则为文件分配新的空间；如果有新分配空间，将文件标记为 dirty 加到 dirty_files 当中，将数据进行磁盘块大小对其后落盘，此时数据已经写到硬盘当中，元数据还未更新，同时 BlueFS 中的文件都是追加写入，不存在原地覆盖写，就算失败也不会污染原来的数据。

4. flush_and_sync_log(写元数据)

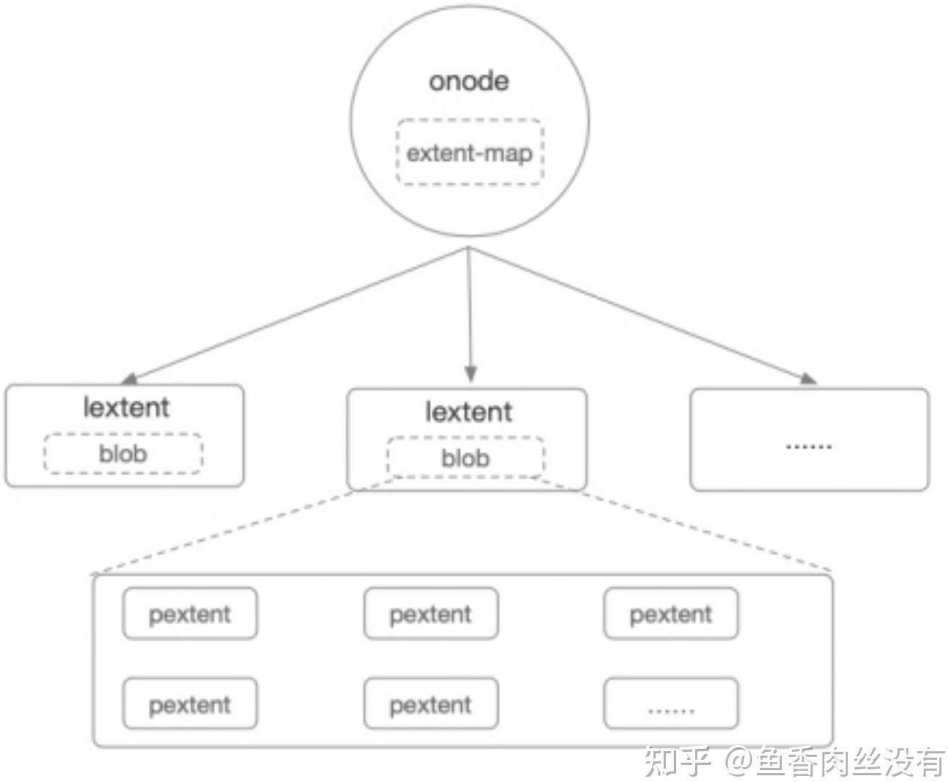
   从 dirty_files 中取到 dirty 的文件，在事务 log_t 中添加更新操作（即添加 OP_FILE_UPDATE 类型的记录），将 log_t 中的内容 sync 到 journal 中，然后移除 dirty_files 中已更新的文件。

第 3 步是写数据、第 4 步是写元数据，都涉及到 sync 落盘，整体一个文件的写入需要两次 sync，已经算是很不错了。

8.5 对象 IO

BlueStore 中的对象非常类似于文件系统中的文件，每个对象在 BlueStore 中拥有唯一的 ID、大小、从 0 开始逻辑编址、支持扩展属性等，因此对象的组织形式，类似于文件也是基于 Extent。

BlueStore 的每个对象对应一个 Onode 结构体，每个 Onode 包含一张 extent-map，extent-map 包含多个 extent(lextent 即逻辑的 extent)，每个 extent 负责管理对象内的一个逻辑段数据并且关联一个 Blob，Blob 包含多个 pextent(物理的 extent，对应磁盘上的一段连续地址空间的数据)，最终将对象的数据映射到磁盘上。具体可参考 BlueStore 源码分析之对象 IO 和 BlueStore 源码分析之事物状态机。

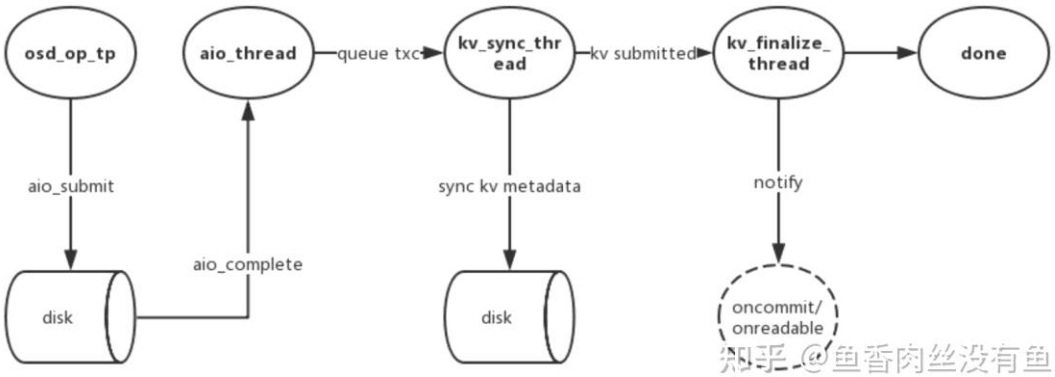


BlueStore 中磁盘的最小分配单元是 min_alloc_size，HDD 默认 64K，SSD 默认 16K，里面有 2 种磁盘分配的写类型(分配磁盘空间，数据还在内存)：

1. **big-write**：对齐到 min_alloc_size 的写我们称为大写(big-write)，在处理是会根据实际大小生成 lextent、blob，lextent 包含的区域是 min_alloc_size 的整数倍，如果 lextent 是之前写过的，那么会将之前 lextent 对应的空间记录下来并回收。
2. **small-write**：落在 min_alloc_size 区间内的写我们称为小写(small-write)。因为最小分配单元 min_alloc_size，HDD 默认 64K，SSD 默认 16K，所以如果是一个 4KB 的 IO 那么只会占用到 blob 的一部分，剩余的空间还可以存放其他的数据。所以小写会先根据 offset 查找有没有可复用的 blob，如果没有则生成新的 blob。

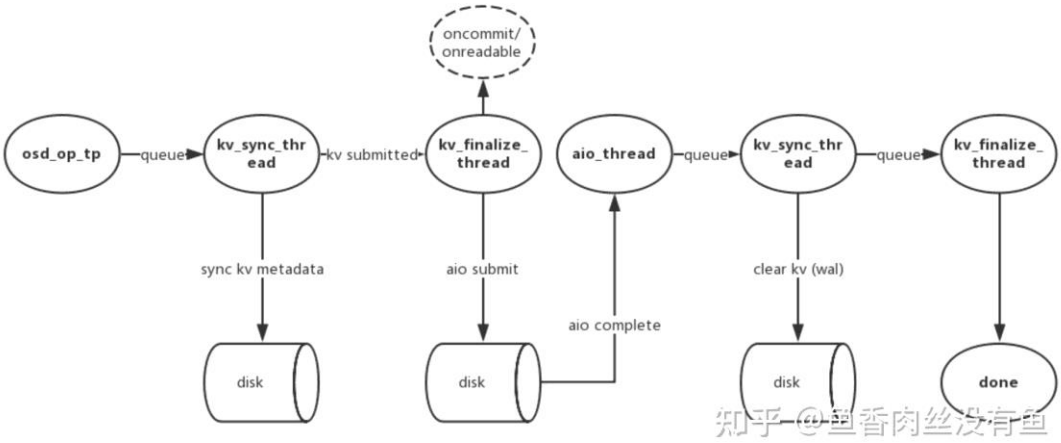
真正写磁盘时，有两种不同的写类型：

1、simple-write：包含对齐覆盖写(COW)和非覆盖写，先把数据写入新的磁盘 block，然后更新 RocksDB 里面的 KV 元数据，状态转换图如下：



这图话的比较好，拿过来直接用了，如有侵权，联系删除。

2、deferred-write：为非对齐覆盖写，先把数据作为 WAL 写 RocksDB 即先写日志，然后会进行 RMW 操作写数据到磁盘，最后 CleanupRocksDB 中的 deferred-write 的数据。



这图话的比较好，拿过来直接用了，如有侵权，联系删除。

3、simple-write + deferred-write：上层的一次 IO 很有可能同时涉及到 simple-write 和 deferred-write，其状态机就是上面两个加起来，只不过少了 deferred-write 的写 WAL 一步，因为可以在 simple-write 写元数据时就一同把 WAL 写入 RocksDB。

9 未来规划

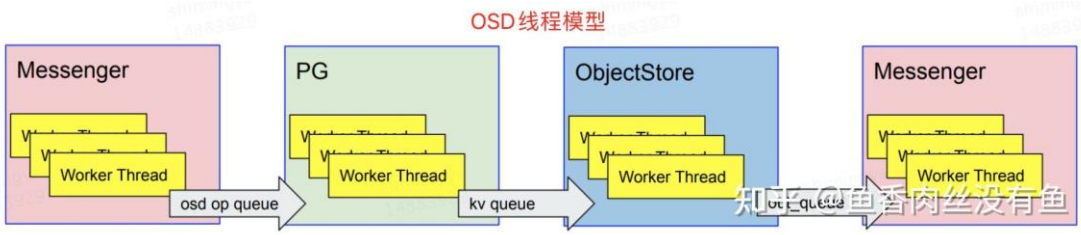
随着硬件的不断发展，IO 的速度越来越快，PMEM 和 NVME 也逐渐成为了存储系统的主流选择，相比之下 CPU 的速度没有那么快了，反而甚至成为了系统的瓶颈。如何高效合理的利用新型硬件是分布式存储不得不面临的一个重大问题。Ceph 传统的线程模型是多线程+队列

的模型，一个 IO 从发起到完成要经历重重队列和不同的线程池，锁竞争、上下文切换和 Cache Miss 比较严重，也导致 IO 延迟迟迟降不下来。通过 Perf 发现 CPU 主要都耗在了锁竞争和系统调用上，Ceph 自身的序列化和反序列化也比较消耗 CPU，所以需要一套新的编程框架来解决上述问题。Seastar 是一套基于 future-promsie 现代化高效的 share-nothing 的网络编程框架，从 18 年开始，Ceph 社区便基于 Seastar 来重构整个 OSD，项目代号 Crimson，来更好的解决上述问题。

Crimson 设计目标

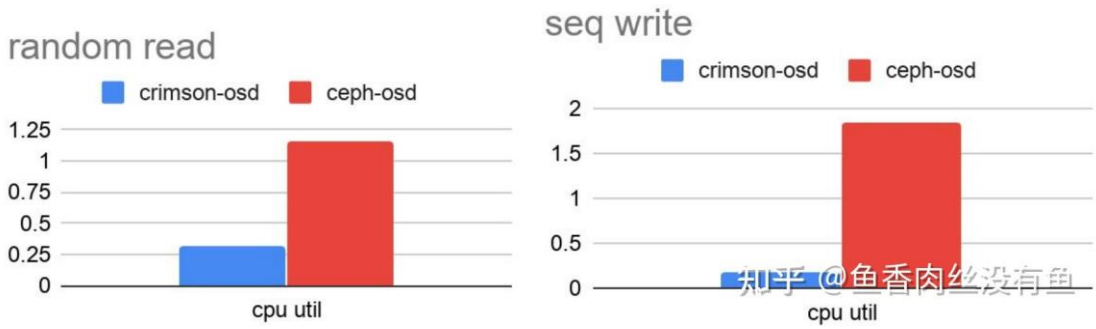
1. 最小化 CPU 开销。
2. 减少跨核通信。
3. 减少数据拷贝。
4. Bypass Kernel，减少上下文切换。
5. 支持新硬件：ZNS-NVME、PMEM 等。

线程模型



性能对比

测试 RBD 时，在达到同等 iops 和延迟时，crimson-osd 的 cpu 比 ceph-osd 的 cpu 少了好几倍。



BlueStore 适配

BlueStore 目前是 Ceph 里性能比较高的单机存储引擎，从设计研发到稳定差不多持续了 3 年时间，足以说明研发一个单机存储引擎的时间成本是比较高的。由于 BlueStore 不符合 Seastar 的编程模型，所以需要对 BlueStore 适配，目前有两种方案：

1. BlueStore-Alien：使用一个 Alien Thread，使用 Seastar 的编程模型专门向 Seastar-Reactor 提交 BlueStore 的任务。
2. BlueStore-Native：使用 Seastar-Env 来实现 RocksDB 的 Rocksdb-Env，从而更原生的适配。

但是由于 RocksDB 有自己的线程模型，外部不可控，所以无论怎么适配都不是最好的方案，理论上从 0 开始用基于 Seastar 的模型来写一个单机存储引擎是最完美的方案，于是便有了 SeaStore，而 BlueStore 的适配也作为中间过渡方案，最多可用于 HDD。

SeaStore

SeaStore 是下一代的 ObjectStore，适用于 Crimson 的后端存储，专门为了 NVME 设计，使用 SPDK 访问，同时由于 Flash 设备的特性，重写时必须先要进行擦除操作，也就是内部需要做 GC，是不可控的，所以 Ceph 希望把 Flash 的 GC 提到 SeaStore 中来做：

1. SeaStore 的逻辑段(segment)理想情况下与硬件 segment(Flash 擦除单位)对齐。
2. SeaStar 是每个线程一个 CPU 核，所以将底层按照 CPU 核进行分段，每个核分配指定个数的 segment。
3. 当磁盘利用率达到阈值时，将少量的 GC 清理工作和正常的写流量一起做。
4. 元数据使用 B+数存储，而不是原来的 RocksDB。
5. 所有 segment 都是追加顺序写入的。