**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**

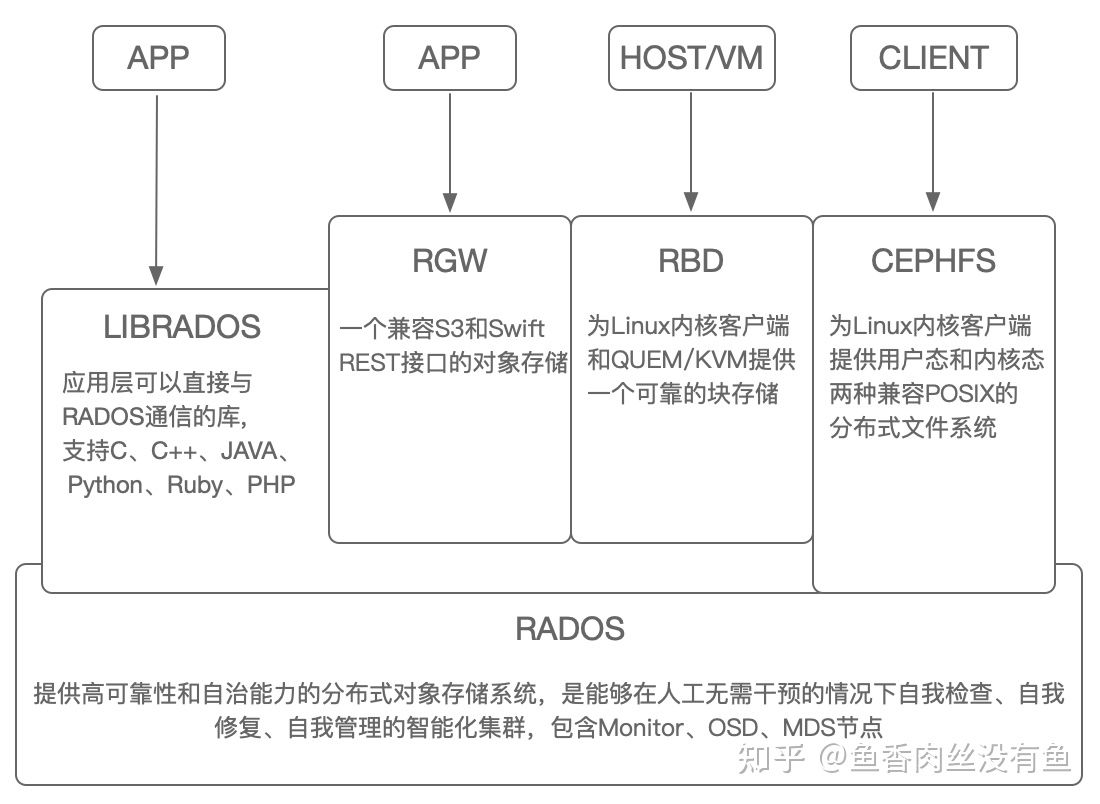
* 1 Ceph概述
* 2 核心组件
* 3 IO流程
* 4 IO顺序性
* 5 PG一致性协议
  + 5.1 StateMachine
  + 5.2 Failover Overview
  + 5.3 PG Peering
  + 5.4 Recovery/Backfill
* 6 引擎概述
* 7 FileStore
  + 7.1 架构设计
  + 7.2 对外接口
  + 7.3 日志类型
  + 7.4 幂等操作
* 8 BlueStore
  + 8.1 架构设计
  + 8.2 BlockDevice
  + 8.3 磁盘分配器
  + 8.4 BlueFS
  + 8.5 对象IO
* 9 未来规划

**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。



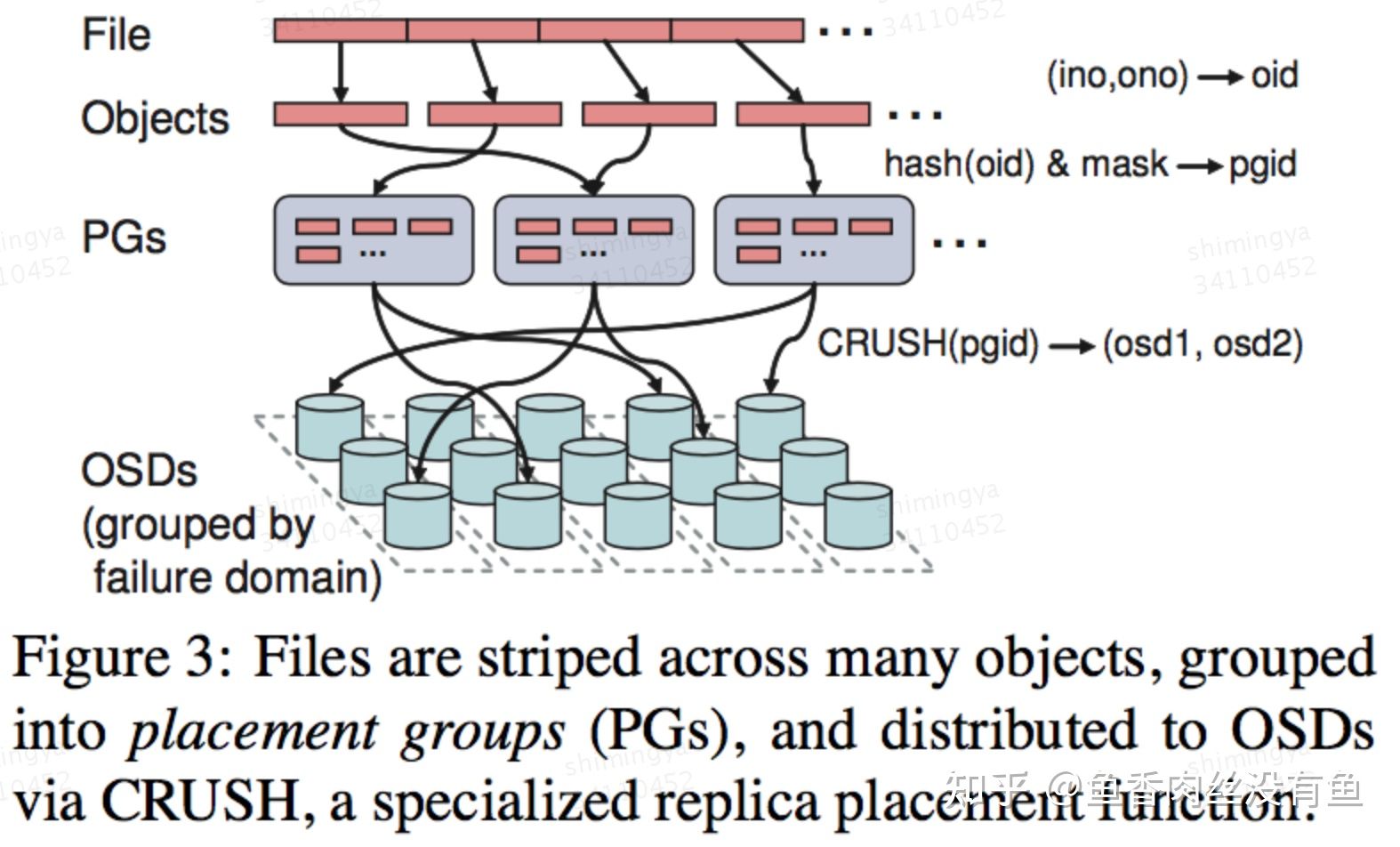
**2 核心组件**

* **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](https://link.zhihu.com/?target=https%3A//www.usenix.org/system/files/fast20-wang_li.pdf" \t "_blank)。
* **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**

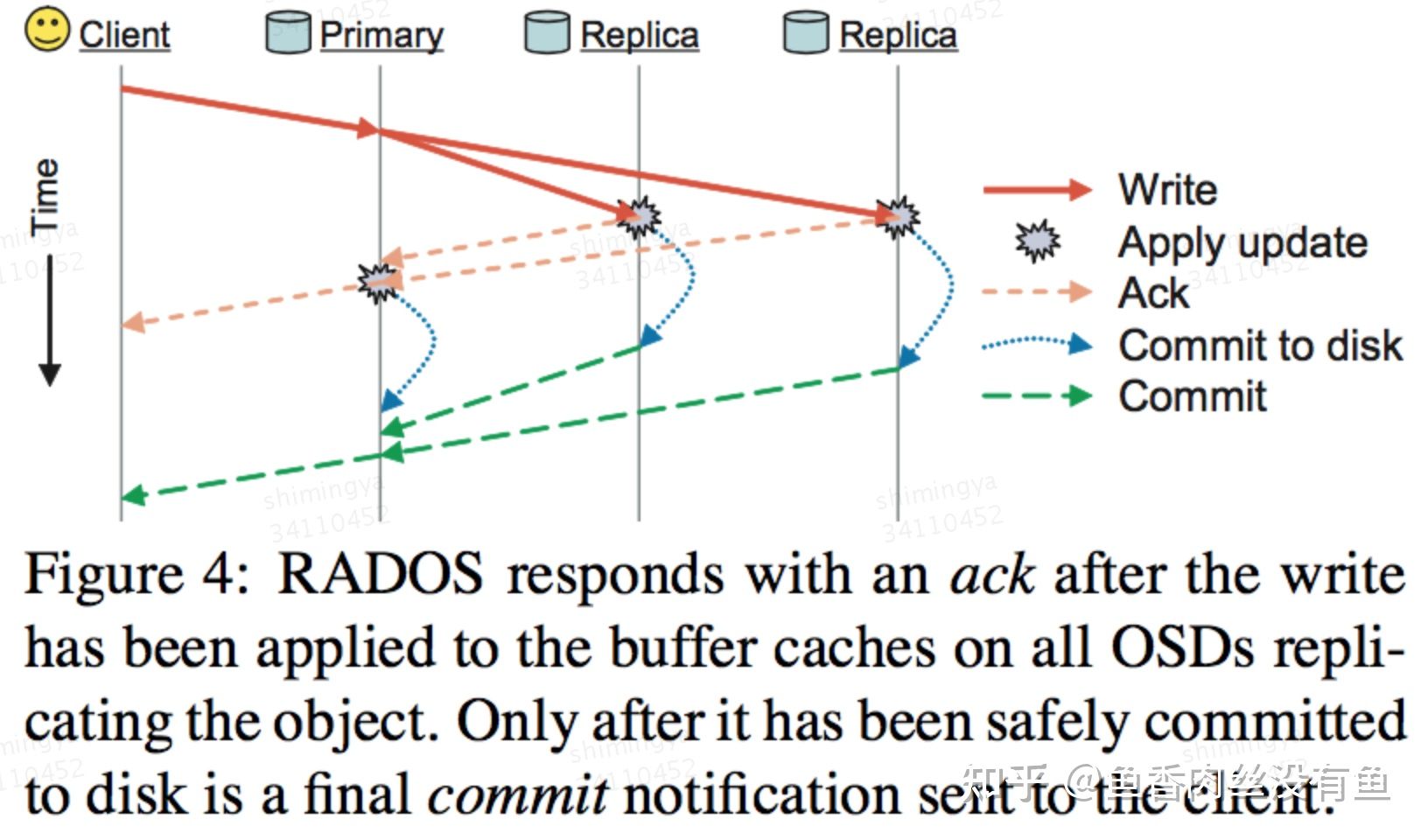
此处以RBD块设备为例简要介绍Ceph的IO流程。

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](https://zhuanlan.zhihu.com/p/104993573)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。



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。



**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。

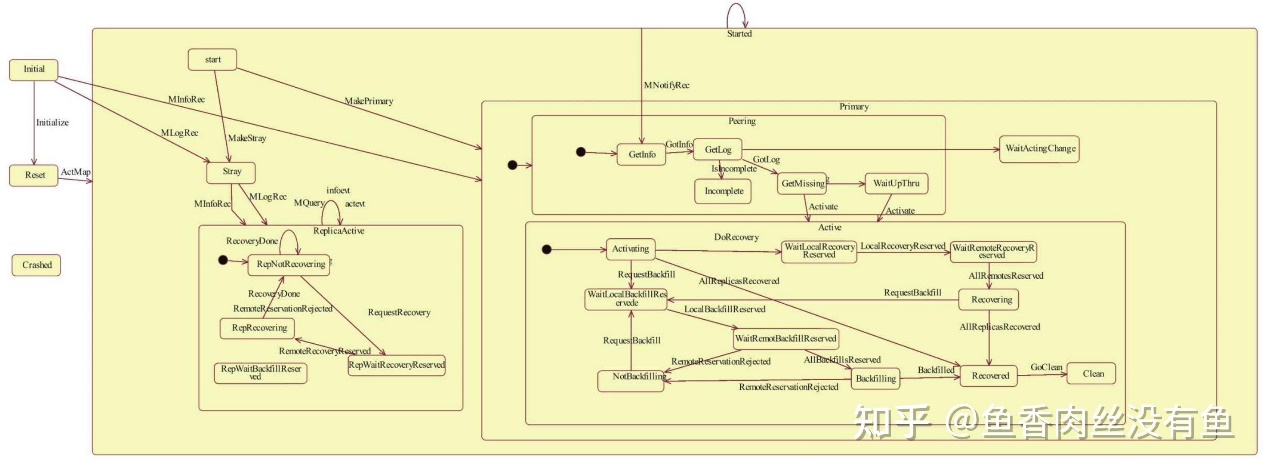
**5 PG一致性协议**

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：

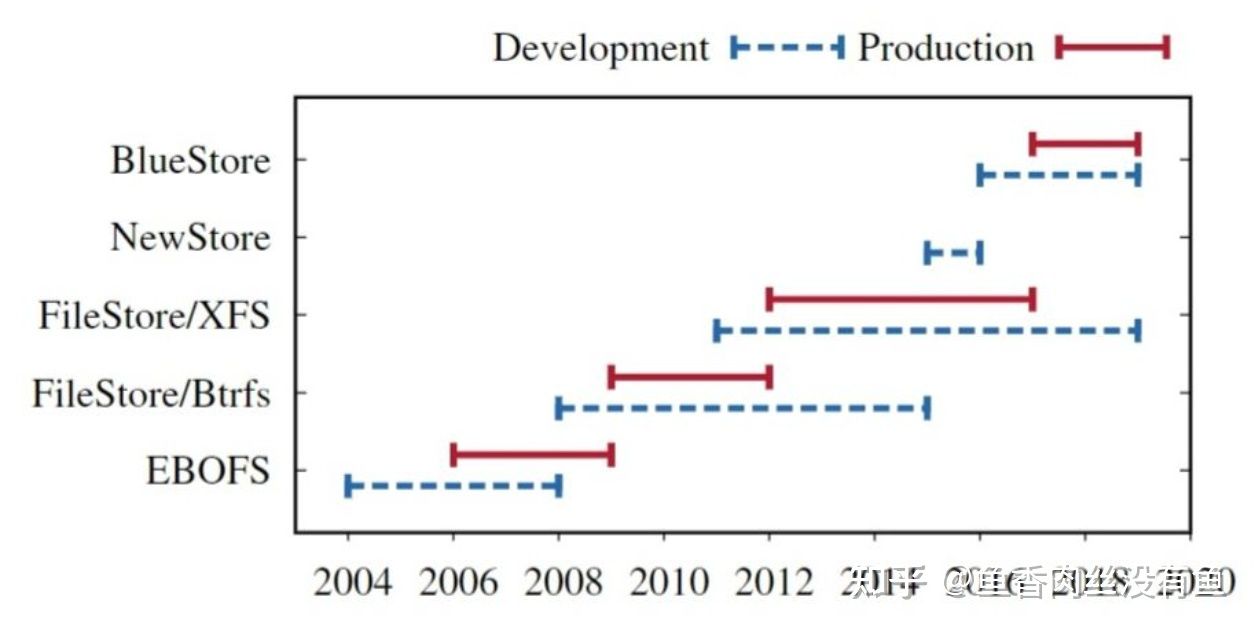
1. 客户端IO和内部恢复IO可以同时进行。
2. 读写的对象不在恢复列表中：按照正常IO即可。
3. 读取的对象在恢复列表中：如果primary有则可以直接读取，如果没有需要优先恢复该对象，然后读取。
4. 写入的对象在恢复列表中：优先恢复该对象，然后写入。
5. backfill则是primary遍历当前所有的对象，将他们全量拷贝到backfill 的PG中。
6. 恢复完成后，会重新进行Peering，是active set 和up set保持一致，变为active + clean状态。

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](https://link.zhihu.com/?target=https%3A//github.com/ceph/ceph/pull/21722" \t "_blank)。

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](https://link.zhihu.com/?target=https%3A//docs.ceph.com/en/latest/dev/osd_internals/async_recovery/%23%3A~%3Atext%3DCeph%2520Placement%2520Groups%2520%28PGs%29%2520maintain%2Cto%2520scan%2520all%2520RADOS%2520objects." \t "_blank)。

**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。

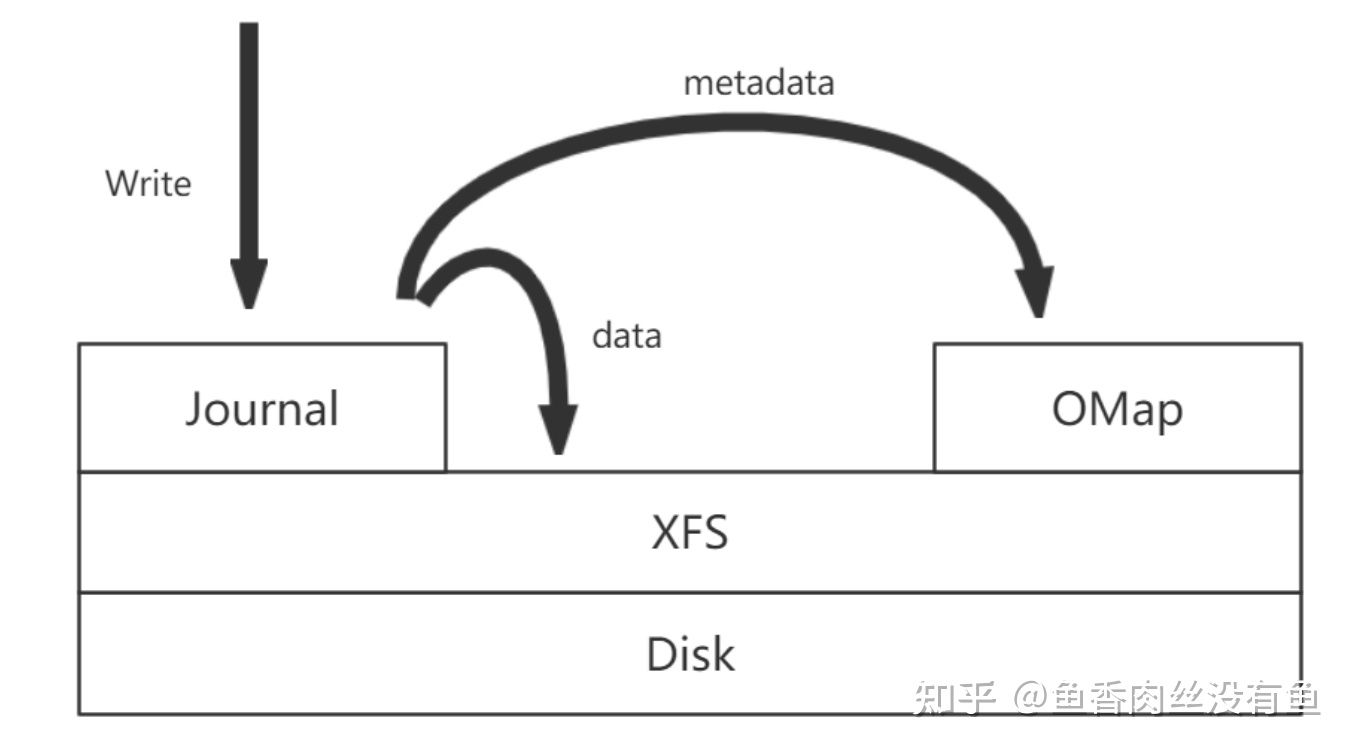


**7 FileStore**

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](https://link.zhihu.com/?target=https%3A//github.com/ceph/ceph/blob/master/src/os/Transaction.h%23L107" \t "_blank)。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**

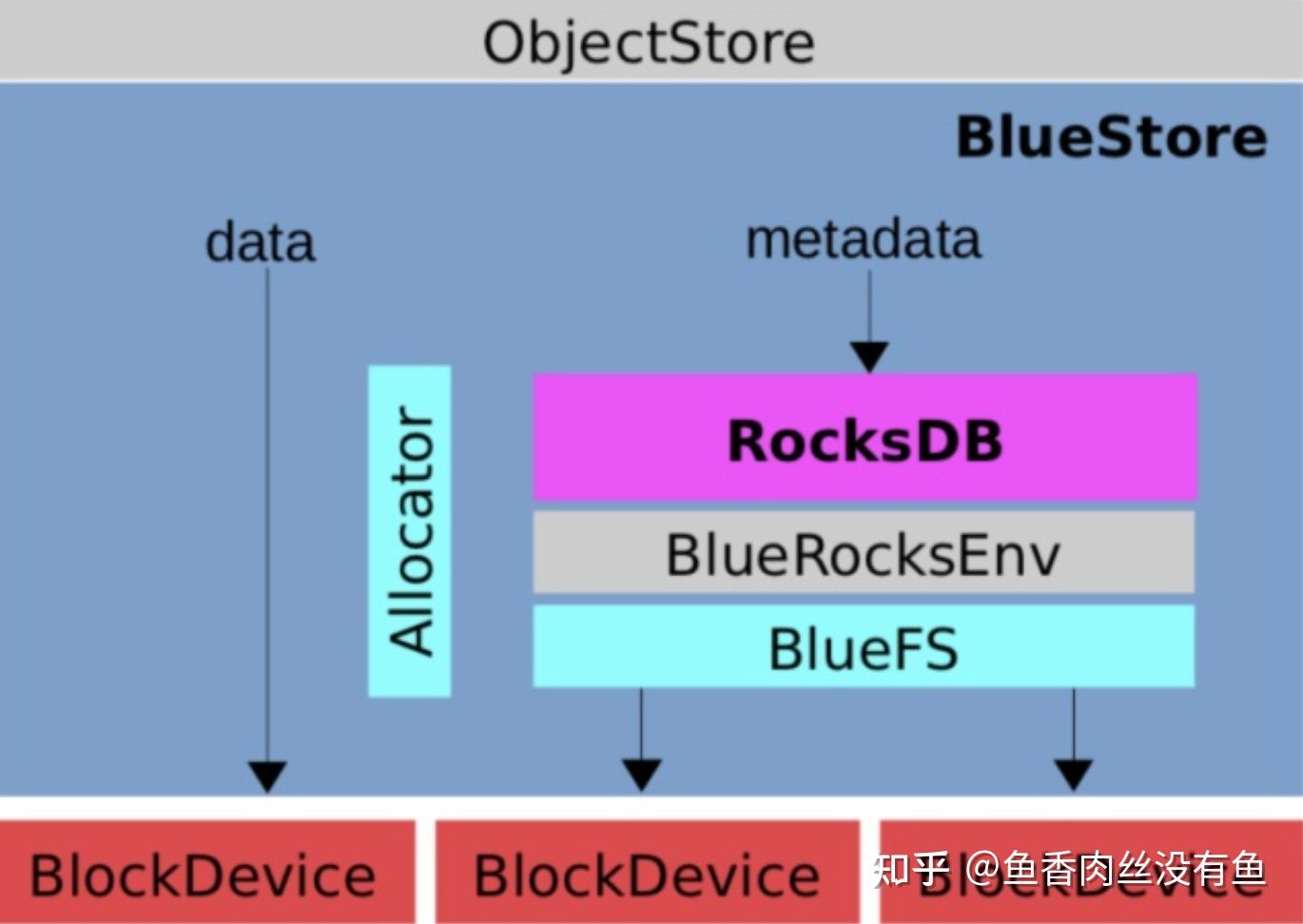
**BlueStore是一个事务型的本地日志文件系统**。因为面向下一代全闪存阵列的设计，所以BlueStore在保证数据可靠性和一致性的前提下，需要尽可能的减小日志系统中双写带来的影响。全闪存阵列的存储介质的主要开销不再是磁盘寻址时间，而是数据传输时间。因此当一次写入的数据量超过一定规模后，写入Journal盘(SSD)的延时和直接写入数据盘(SSD)的延迟不再有明显优势，所以Journal的存在性便大大减弱了。但是要保证OverWrite(覆盖写)的数据一致性，又不得不借助于Journal，所以针对Journal设计的考量便变得尤为重要了。

**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：



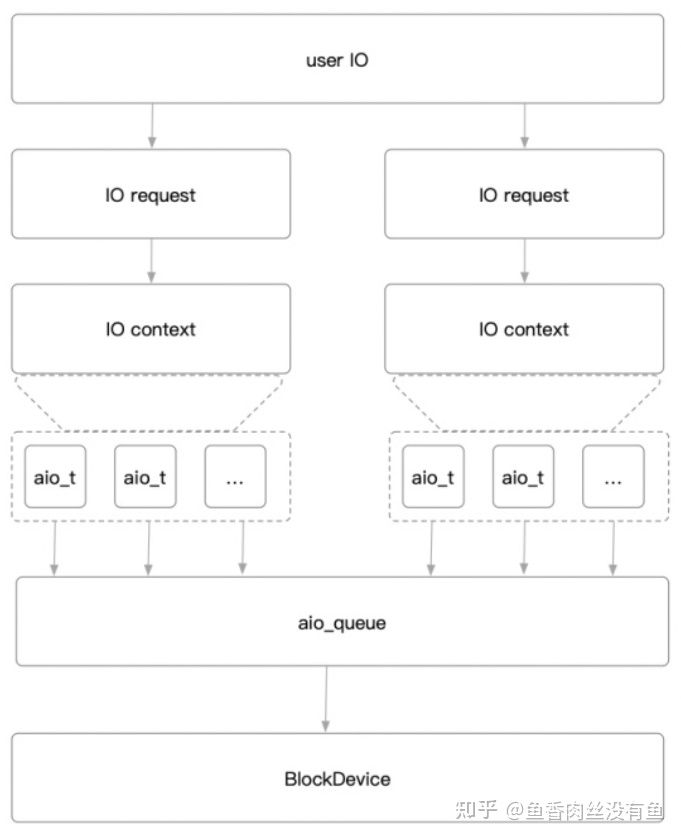
* BlockDevice：物理块设备，使用Libaio、SPDK、io\_uring操作裸设备，AsyncIO。
* RocksDB：存储对象元数据、对象扩展属性Omap、磁盘分配器元数据。
* BlueRocksEnv：抛弃了传统文件系统，封装RocksDB文件操作的接口。
* BlueFS：小型的Append文件系统，实现了RocksDB::Env接口，给RocksDB用。
* Allocator：磁盘分配器，负责高效的分配磁盘空间。
* Cache：实现了元数据和数据的缓存。

**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](https://zhuanlan.zhihu.com/p/91020703)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：

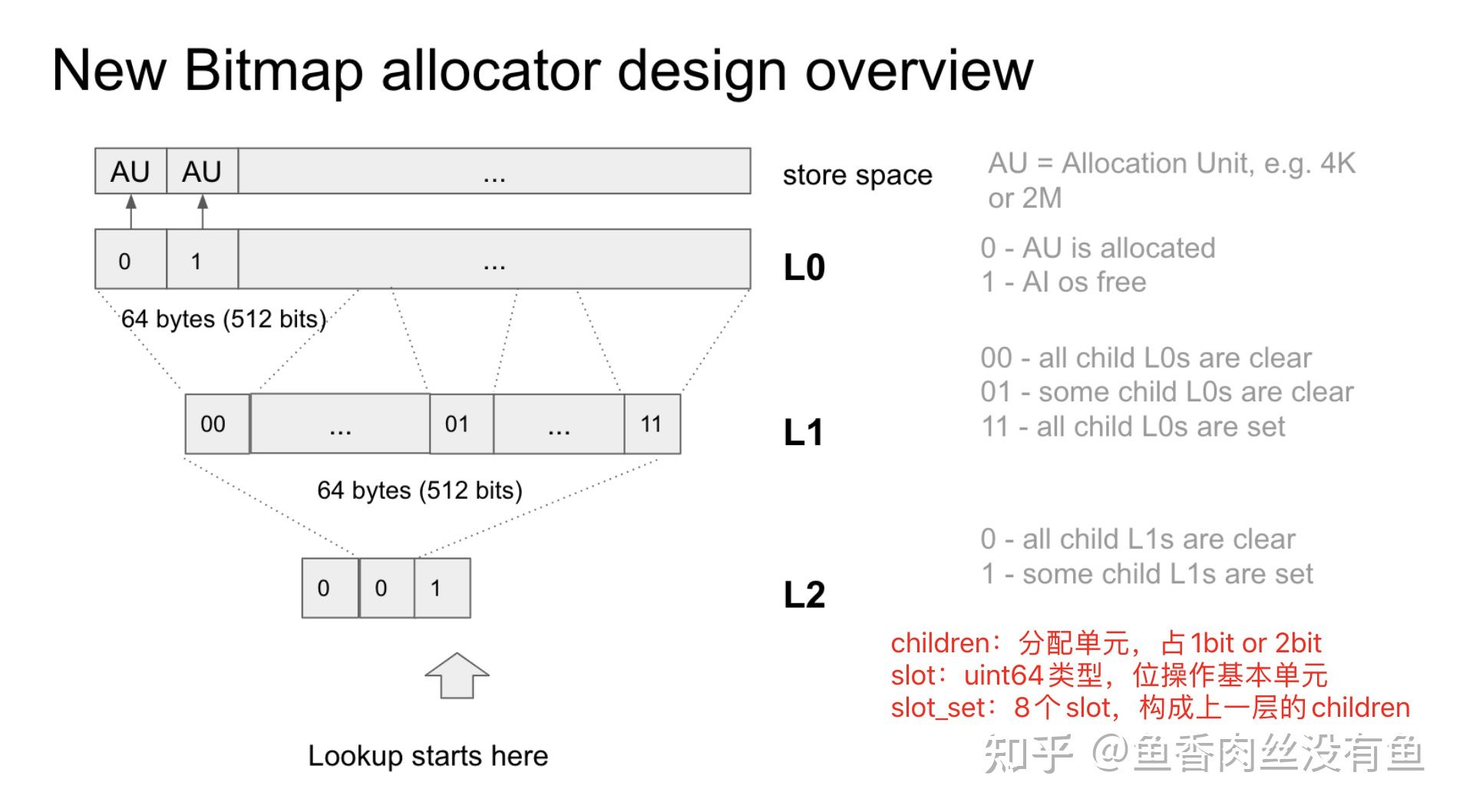


**8.3 磁盘分配器**

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)](https://zhuanlan.zhihu.com/p/91018497)和[Stupid分配器(基于Extent)](https://zhuanlan.zhihu.com/p/91019062)，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](https://link.zhihu.com/?target=https%3A//docs.google.com/presentation/d/1_1Otkgv76fbCU2Zogjz748sEAG-1Nfiidbb6mgTON-A/edit%23slide%3Did.p" \t "_blank)，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](https://zhuanlan.zhihu.com/p/91025247)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**

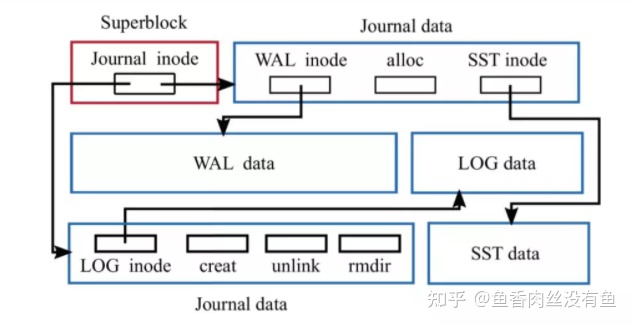
RocksDB是通过BlueRocksEnv来使用BlueFS的，BlueRocksEnv实现了文件读写和目录操作，其他的都继承自rocksdb::EnvWrapper。

* 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。

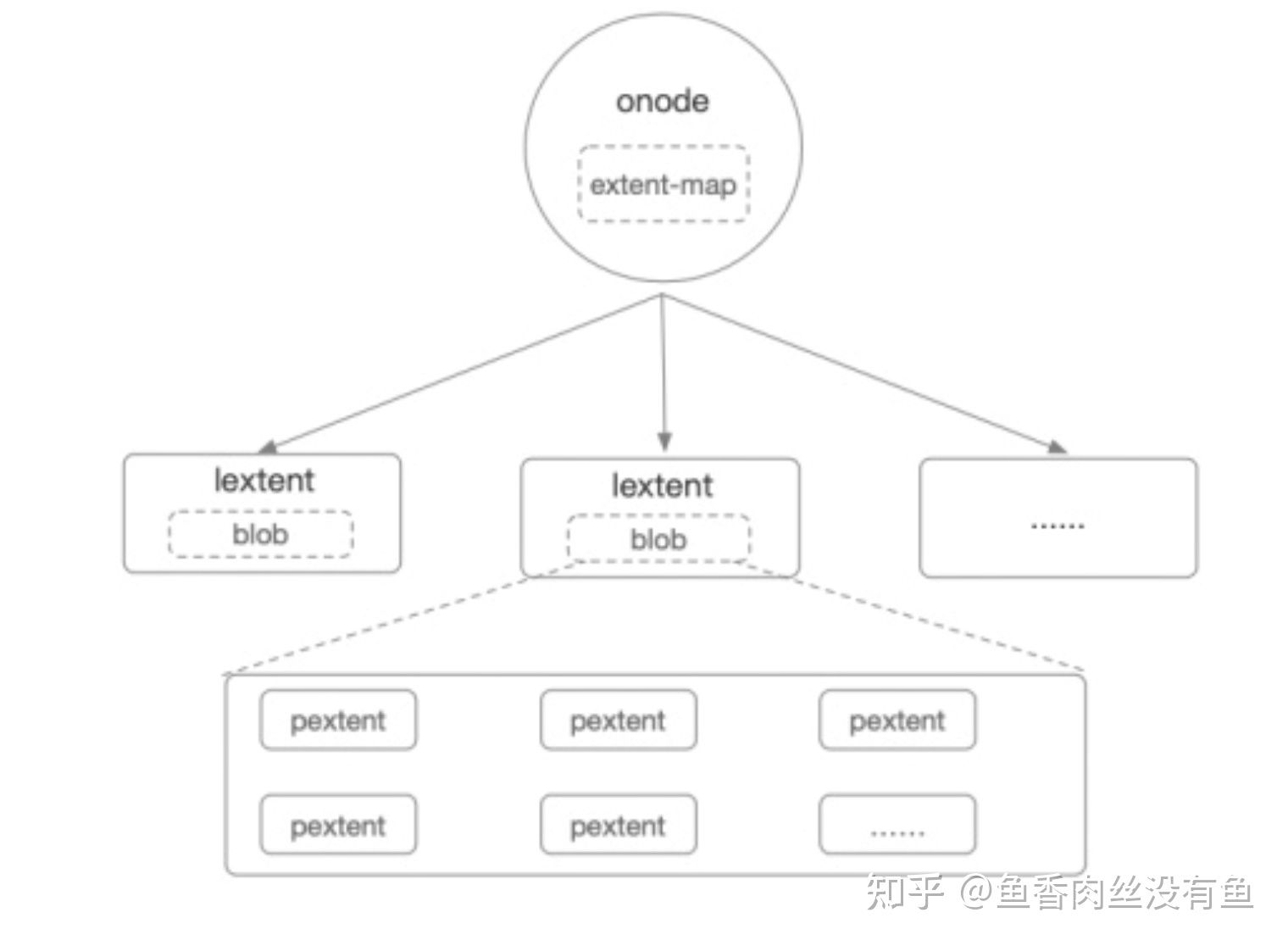
1. open file for write  
   打开文件句柄，如果文件不存在则创建新的文件，如果文件存在则会更新文件fnode中的mtime，在事务log\_t中添加更新操作，此时事务记录还不会持久化到journal中。
2. append file  
   将数据追加到文件当中，此时数据缓存在内存当中，并未落盘，也未分配新的空间。
3. flush data(写数据)  
   判断文件已分配剩余空间（fnode中的 allocated - size）是否足够写入缓存数据，若不够则为文件分配新的空间；如果有新分配空间，将文件标记为dirty加到dirty\_files当中，将数据进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](https://zhuanlan.zhihu.com/p/92397191)和[BlueStore源码分析之事物状态机](https://zhuanlan.zhihu.com/p/92850284)。

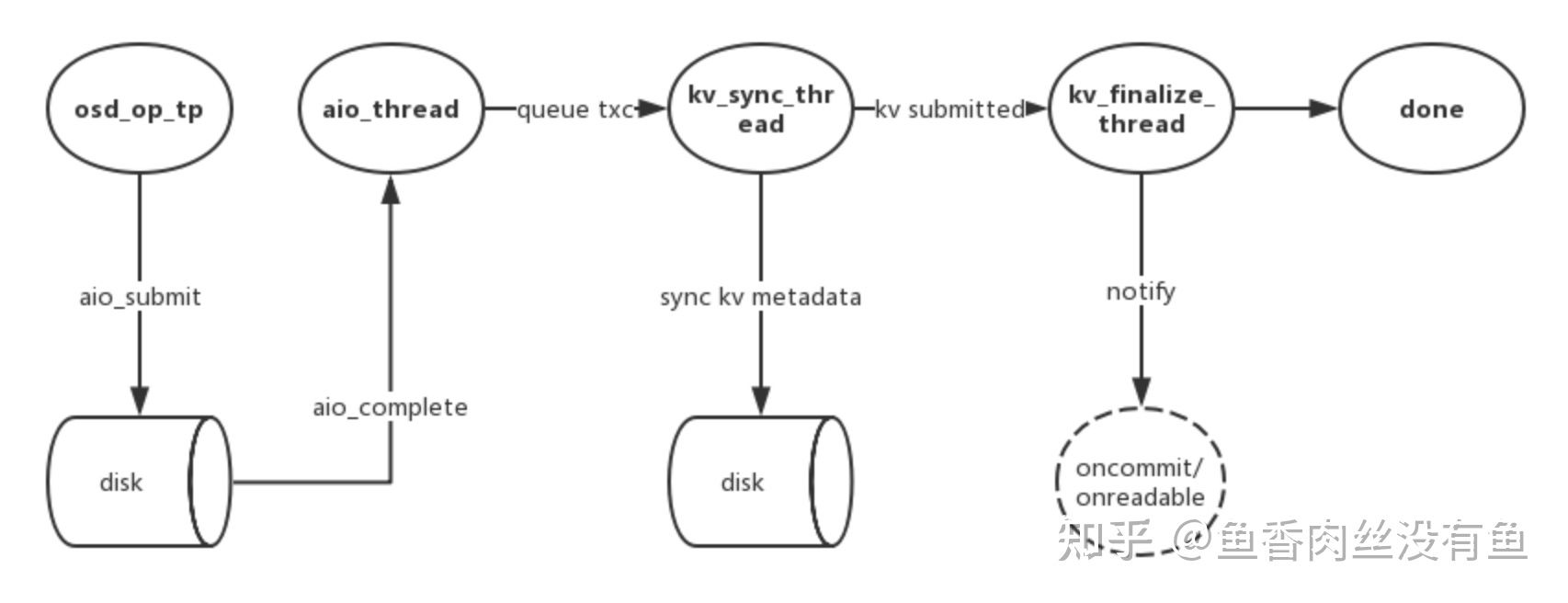


BlueStore中磁盘的最小分配单元是min\_alloc\_size，HDD默认64K，SSD默认16K，里面有2种磁盘分配的写类型(分配磁盘空间，数据还在内存)：

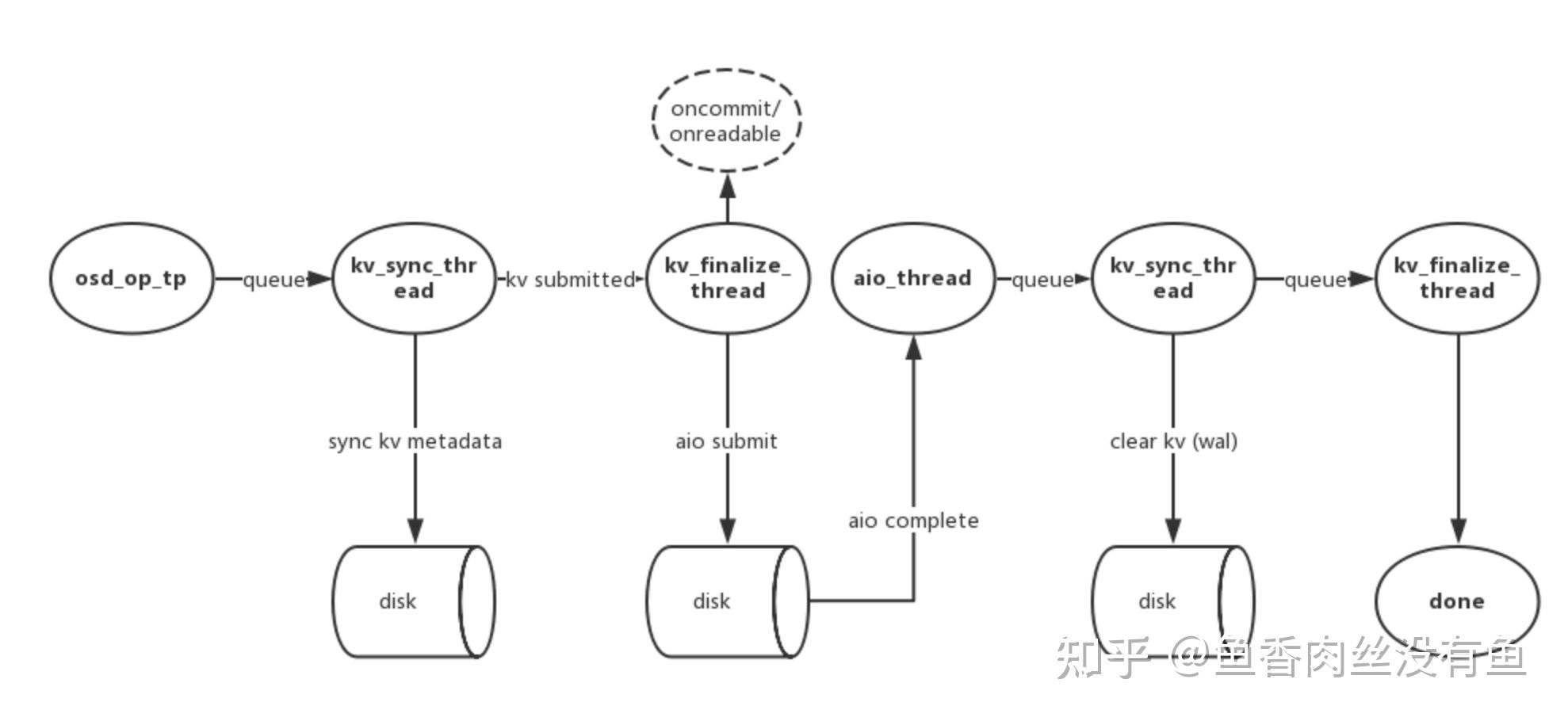
1. **big-write**：对齐到min\_alloc\_size的写我们称为大写(big-write)，在处理是会根据实际大小生成lextent、blob，lextent包含的区域是min\_alloc\_size的整数倍，如果lextent是之前写过的，那么会将之前lextent对应的空间记录下来并回收。
2. **small-write**：落在min\_alloc\_size区间内的写我们称为小写(small-write)。因为最小分配单元min\_alloc\_size，HDD默认64K，SSD默认16K，所以如果是一个4KB的IO那么只会占用到blob的一部分，剩余的空间还可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

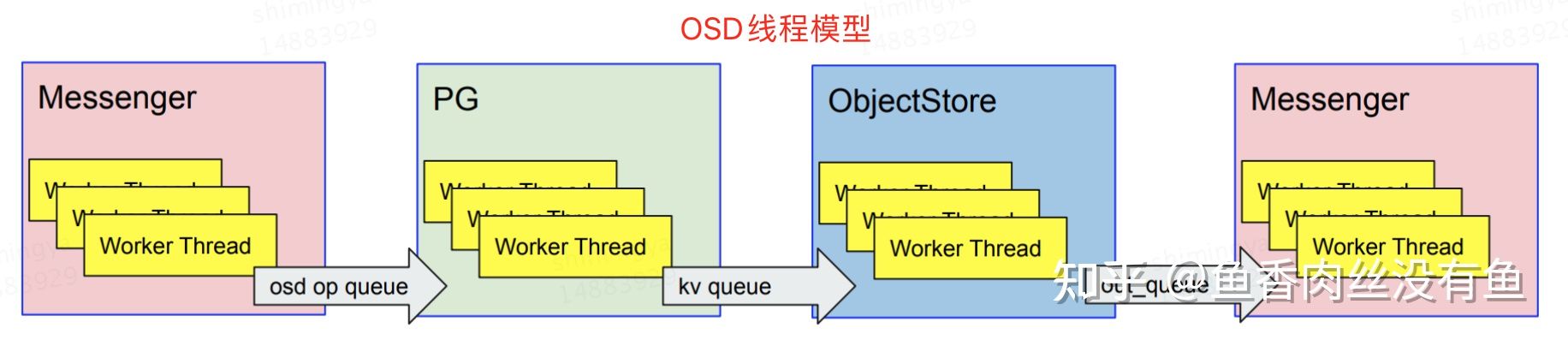
**9 未来规划**

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](https://link.zhihu.com/?target=https%3A//github.com/ceph/ceph/tree/master/src/crimson" \t "_blank)，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。