第 2 章 Chapter 2

性能之巅

——新型对象存储引擎 BlueStore

BlueStore 最早在 Jewel 版本中引入，用于取代传统的 FileStore，作为新一代高性能对象存储后端。BlueStore 在设计中充分考虑了对下一代全 SSD 以及全 NVMe SSD 闪存阵列的适配，例如将一直沿用至今、用于高效索引元数据的 DB 引擎由 LevelDB 替换为 RocksDB[⊖]（RocksDB 基于 LevelDB 发展而来，并针对直接使用 SSD 作为后端存储介质的场景做了大量优化）。FileStore 因为仍然需要通过操作系统自带的本地文件系统间接管理磁盘，所以所有针对 RADOS 层的对象操作，都需要预先转换为能够被本地文件系统识别、符合 POSIX 语义的文件操作，这个转换的过程极其繁琐，效率低下。

针对 FileStore 的上述缺陷，BlueStore 选择绕过本地文件系统，由自身接管裸设备（例如磁盘），直接进行对象操作，不再进行对象和文件之间的转换，从而使得整个对象存储的 I/O 路径大大缩短，这是 BlueStore 能够提升性能的根本原因。除此之外，考虑到元数据的索引效率对于性能有着致命影响，BlueStore 在设计中将元数据和用户数据严格分离，因此 BlueStore 中的元数据可以单独采用高速固态存储设备，例如使用 NVMe SSD 进行存储，能够起到性能加速的作用。最后，与传统机械磁盘相比，SSD 普遍采用 4K 或者更大的块大小，因此 SSD 采用位图进行空间管理可以取得比较高的空间收益（机械磁盘块大小为扇区，SSD 块大小为 4K，假定磁盘容量均为 1TB，如果使用位图管理磁盘

⊖ https://github.com/facebook/rocksDB

空间，那么两者的位图大小分别为 256MB 和 32MB，相差 8 倍！现在主流 SSD 容量远比主流机械磁盘容量小，所以 SSD 对应的位图更小，使其常驻内存成为可能），同时，因为位图相较当前主流的段索引方式而言，碎片化程度更低、效率更高，所以 BlueStore 的磁盘空间管理回归传统，采用位图方式。

本章按照如下形式组织：首先，我们回顾 BlueStore 需要重点解决的问题，以及期望具有的特性，因为 BlueStore 的设计理念与 FileStore 是如此的不同，所以几乎所有磁盘数据结构都需要重新设计；其次，BlueStore 虽然绕过了本地文件系统，但是本地文件系统中诸如缓存、磁盘空间管理等技术对存储系统而言具有通用性，通过针对一些现有方案进行对比分析，我们期望找到一种高度契合 BlueStore 定位同时兼具优异性能的缓存 / 磁盘空间管理方案来和 BlueStore 进行适配；再次，数据库引擎是 BlueStore 的心脏，RocksDB 作为一种通用的键值对数据库解决方案，具有高度可定制化的特性，通过对 RocksDB 的一些组件进行替换并合理地进行功能裁剪，我们可以从 RocksDB 获得更加卓越的性能表现，从而使得 BlueStore 心跳更加有力；最后，通过针对 BlueStore 上电、读写等几个关键流程进行分析，我们简要探讨了 BlueStore 的内部实现，并据此给出安装部署 BlueStore 的步骤及注意事项。

2.1 设计理念与指导原则

存储系统中，所有读操作都是同步的，即除非在缓存中命中，否则必须要从磁盘中读到指定的内容后才能向前端返回。写操作则不一样，一般而言，出于效率考虑，所有写操作都会预先在内存中进行缓存，由文件系统进行合适的组织后，再批量写入磁盘。理论上，数据写入缓存即可向前端返回写入完成应答，但是由于内存数据在掉电后会丢失，因此出于数据可靠性考虑，我们无法这么做。一种可行的替代方案是将数据先写入相较普通磁盘而言性能更好并且掉电后数据不会丢失的中间设备，等待数据写入普通磁盘后再释放中间设备上的相应空间。这个写中间设备的过渡过程称为写日志，中间设备相应地被称为日志设备，一般可由 NVRAM 或者 SSD 等高速固态存储设备充当。引入日志设备后，数据在写入日志设备后即可向前端应答写入完成，如果此时掉电，即使数据尚未写入普通设备，系统重新上电后也可以通过日志重放进行数据恢复。因此，日志系统的引入可以在不影响数据可靠性的基础上对写操作进行加速，包含日志的存储系统也被称为日志型存储系统。日志型存储系统一个显而易见的缺点是引入了日志设备，需要

消耗额外的硬件资源，但是因为日志空间可以回收供重复使用，所以日志设备不需要配置很大的容量，因此实现上也可以多个普通磁盘共享一个高速日志设备。除此之外，因为同一份用户数据要先后在日志设备和普通磁盘上写两次，所以日志型存储系统还存在“双写”的问题。特别的，当日志设备和普通磁盘合一时，因为“双写”问题所带来的性能损失则是灾难性的。

除了数据可靠性之外，存储系统一般而言还需要考虑数据一致性的问题，即涉及数据修改相关的操作，要么全部完成，要么没有任何变化，而不能是介于此两者之间的某个中间状态（All or nothing）。符合上述语义的存储系统也称为事务型存储系统，其最大的特点是所有的修改操作都符合事务的 ACID 语义。ACID，是保证事务正确执行四个基本要素的缩写，即：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。一个支持事务（Transaction）语义的系统（典型如数据库），必须具有这四种特性，否则在事务执行过程中无法保证数据的正确性。

BlueStore 可以理解为一个事务型的本地日志文件系统（但是实际上存储的是对象），因为其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，需要尽可能减小由于日志系统存在而引入的“双写”问题所带来的负面影响，以提升写性能（事实上，因为目前 Ceph 的主要商用备份策略依然是跨节点的多副本，所以 Ceph 的写性能一直为人诟病）。当前，全闪存阵列普遍使用普通 SSD 充当数据盘，为了起到写加速的作用，此时日志设备（与数据盘相对，以下简称日志盘）只能由性能更高的 NVRAM 或者 NVMe SSD 等高速固态存储设备充当。与传统阵列普遍使用机械磁盘充当数据盘、使用普通 SSD 充当日志盘不同，全闪存阵列后端固态存储介质的主要性能开销不再是寻址时间，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时，写入日志盘的时延和直接写入数据盘的时延不再具有明显优势（参见表 2-1），此时日志存在的必要性大大减弱。

表 2-1 不同存储介质 512KB 顺序读写时延

机械磁盘（HitachiHUA722010CLA330）与普通 SSD（S3500）读 / 写时延相差约为 64/162 倍
普通 SSD（S3500）和 NVMe SSD（P3600）读 / 写时延相差约为 2/4 倍

磁盘类型	512KB 顺序读时延（ms）	512KB 顺序写时延（ms）
HitachiHUA722010CLA330	515.32	473.37
Intel S3500	8.38	2.91
Intel P3600	3.36	0.74

一个可行的改进方案是使用增量日志，即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日志，其他部分因为不需要执行 RMW，则可以直接执行重定向写。

为了更好地理解上述改进方案，首先介绍与之相关的几个术语：

（1）block-size（块大小）

磁盘块大小指对磁盘进行操作的最小粒度（也称为原子粒度），例如对普通机械盘而言，这个最小粒度为 512 字节，即一个扇区；现代 SSD 普遍使用更大的块大小，例如 4KB。

（2）RMW（Read Modify Write）

指当覆盖写（即改写已有内容）发生时，如果本次改写的内容不足一个磁盘块大小，那么需要先将对应的块读上来，然后将待修改内容与原先的内容进行合并（从这个角度而言，也可以将 RMW 中的 M 理解为 Merge），最后将更新后的块重新写入原先的位置。RMW 引入了两个问题：一是额外的读惩罚；二是因为要针对已有内容执行覆盖写，所以如果磁盘中途异常掉电，那么会导致潜在的数据损坏风险。

（3）COW（Copy-On-Write）

指当覆盖写发生时，不是直接更新磁盘对应位置的已有内容，而是重新在磁盘上分配一块新的空间，用于存放本次新写入的内容，这个过程也称为写时重定向。当新写完成、对应的地址指针更新后，即可释放原有数据对应的磁盘空间。COW 理论上可以解决 RMW 引入的两个问题，但是自身也存在缺陷。

首当其冲的是 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，前端任何大范围的顺序读后续都将变成随机读，因为读性能对整个存储系统的性能表现有着至关重要的影响，所以这在机械磁盘作为主流存储介质大行其道的年代后果将是灾难性的。不过，近年来随着 SSD 的逐渐普及，这种情况有所好转——众所周知，机械磁盘的读写时延主要是寻道时延，所以随机读写和顺序读写时延可能相差一个数量级或以上，而 SSD 则不然，SSD 的随机读写相较于顺序读写时延一般而言并无数量级的差异，而且因为 COW 机制也有将任意随机写转化为顺序写的能力，所以其对写性能也有一定的补偿作用。

其次，针对非块大小对齐的小块覆盖写，综合来看，采用 COW 依然得不偿失，这

是因为：

- 将新的内容直接写入新块后，原有的块因为仍然保留了部分有效内容，所以 COW 之后不能释放（至少不能全部释放）。这样，这部分读惩罚会被转嫁到后续针对波及范围内的读操作本身，即执行 COW 之后，后续所有针对波及范围内的读操作，都需要两次或者多次读操作，再执行 Merge 操作后才能拼凑出前端最终需要读取的内容。显而易见，上述操作将大大影响读性能。
- 因为 COW 涉及空间重分配和地址指针重定向，所以 COW 最终将引入更多元数据。对存储系统而言，元数据的多寡关乎其功能丰富与否，一般而言，元数据越多，其功能越丰富；反之，则功能相对单一。因为任何操作必然涉及元数据，所以元数据是存储系统中当之无愧的热点数据，因此如果元数据过多，导致其无法常驻缓存，那么必然会导致系统性能大打折扣。从这个角度而言，减少系统的元数据开销，特别是与空间管理相关的元数据开销（因为其与管理的空间大小成正比），对于性能的提升作用不言而喻。

了解上述基本概念后，理解 BlueStore 的写策略变得简单。简言之，BlueStore 针对写操作综合运用了 RMW 和 COW 策略——任何一个写请求，根据磁盘块大小，将其切分为三个部分，即首尾非块大小对齐部分和中间块大小对齐部分，然后针对中间块对齐部分采用 COW 策略，首尾非块对齐部分采用 RMW 策略。考虑到 RMW 策略存在数据损坏的隐患，还需要针对这类操作引入日志，即将对应的数据先写入日志盘后才去真正更新数据盘，数据盘更新完成之后才能释放日志。

针对上层，BlueStore 主要提供了读写两种类型的多线程访问接口，这些接口都是基于 PG 粒度的。因为读请求之间可以并发，而写请求（这里的写请求泛指修改操作，下同）则是排它的，所以 PG 内部使用读写锁来实现上述语义。此外，因为所有读请求都是同步的，而写请求出于效率考虑一般需要设计成异步，所以实现上，还需要为每个 PG 设计一个队列，用于对所有操作该 PG 的写请求进行保序。这个队列称为 OpSequencer，不同类型的 ObjectStore 实现略有不同，在 BlueStore 的实现中，OpSequencer 主要包含两个 FIFO（First In First Out，即先进先出）队列，分别用于对写请求的不同阶段进行保序。如前所述，BlueStore 将所有写请求划分为普通和带日志两种。带日志的写请求创建后，其生命周期分为两个阶段——写日志和覆盖写数据，只有当写日志完成之后，才可以开始覆盖写数据，因为后一阶段在实现策略上使用独立的线程池完成，所以需要一个额外的队列，用于将所有进入覆盖写数据阶段的带日志写请求在线程池中再次进行排序。所

有写请求通过标准（由 ObjectStore 定义）的 queue_transactions 接口提交至 BlueStore 处理。顾名思义，通过 queue_transactions 提交的是多个事务，这些事务形成一个事务组，作为一个整体同样需要符合 ACID 语义。当前受限于 PG 实现，一个事务组并不能真正针对同一个 PG 当中的多个对象操作[⊖]。

通过 BlueStore 提供的接口来操作 PG 下的某个对象，首先需要找到 BlueStore 中对应的 PG 上下文和对象上下文，因为这两类上下文保存了关键的 PG 和对象元数据信息，而且 BlueStore 通过 kvDB 固化这两类上下文至磁盘，所以我们首先需要了解这两类上下文的磁盘数据结构（On-Disk Format，磁盘数据结构一般需要保持相对稳定，改动时需要考虑兼容性）。

2.2 磁盘数据结构

相较 FileStore 而言，BlueStore 因为绕过了系统的本地文件系统，由自身接管磁盘，所以其磁盘数据结构远比 FileStore 复杂。除了 PG 和对象相应的管理结构以外，还需要大量用于组织磁盘数据（典型如将对象中一段逻辑数据映射到磁盘等）的结构。需要注意的是，每种数据结构一般都有磁盘和内存两种格式，在 BlueStore 中习惯上磁盘格式以“_t”结尾并且全部小写，而内存格式不以“_t”结尾并且只有首字母大写。另外，前面已经提及，在 BlueStore 的设计实现中，因为元数据对性能有着至关重要的影响，所以所有元数据都设计成可以和用户数据分开存放，目前统一以键值对的形式存放在 kvDB 中，推荐使用单独的、高性能 SSD/NVMe SSD/NVRAM 设备承载。

2.2.1 PG

Ceph 对集群中所有存储资源进行池化管理。资源池（pool，也称为存储池）实际上是一个虚拟概念，表示一组约束条件，例如可以针对一个 pool 设计一组 CRUSH 规则，限制其只能使用某些存储资源，或者尽量将所有数据副本分布在物理上隔离的、不同的安全域；也可以针对不同的 pool 指定不同的副本策略，例如针对时延敏感的应用采用多副本备份策略，针对一些不重要的备份数据，为提升空间利用率则采用纠删码备份策略；甚至可以分别为每个 pool 指定独立的 scrub、压缩、校验策略等。和 Linux 的设计哲学类

⊖ 关于多对象事务语义支持，参见 https://github.com/ceph/ceph/pull/9398。

似，Ceph 将任意类型的前端数据都抽象为对象（类似 Linux 当中的文件）这个小巧但是精致的概念，每个对象采用一定的策略可以生成一个全局唯一的对象标识（即 Object ID，简称 OID），进一步的，基于此全局唯一的对象标识最终可以形成一个扁平的寻址空间，从而大大提升索引效率。

为了实现不同 pool 之间的策略隔离，Ceph 并不是将任何上层数据一步到位映射到磁盘（OSD），而是引入了一个中间结构，称为 PG，实现两级映射，如图 2-1 所示。

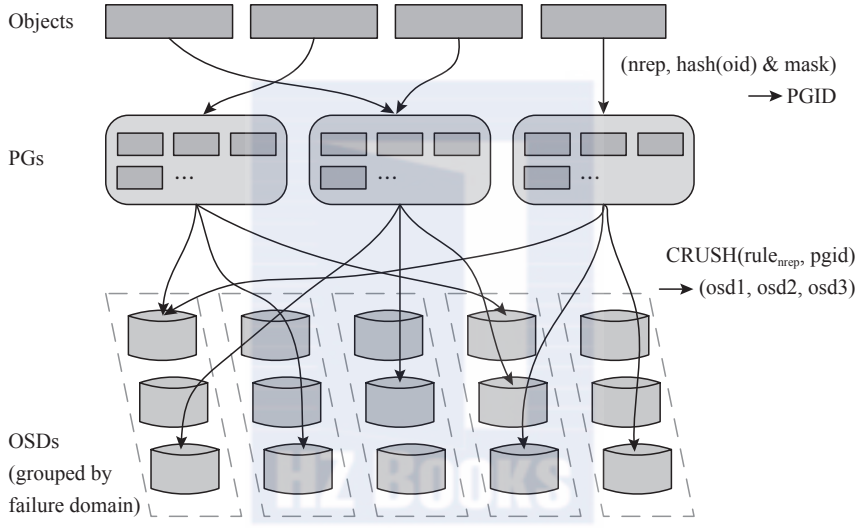

图 2-1 对象以 PG 为单位进行组织，PG 通过 CRUSH 分布在不同安全域中的 OSD 之上

第一级映射是静态的，负责将任何前端类型的应用数据按照固定大小进行切割、编号后作为伪随机哈希函数输入，均匀映射至 PG，以实现负载均衡策略；第二级映射实现 PG 到 OSD 的映射，这级映射仍然采用伪随机哈希函数（以保证 PG 在 OSD 之间分布的均匀性），但是其输入除了全局唯一的 PGID 之外，还引入了集群拓扑，并且使用 CRUSH 规则对计算过程进行调整，以帮助 PG 在不同 OSD 之间进行灵活迁移，进而实现数据可靠性、自动平衡等高级特性。最终，pool 以 PG 作为基本单位进行组织，因此 PG 实际上是一些对象的逻辑载体（集合）。

为了维持扁平寻址空间，实际上要求 PG 也拥有一个全集群唯一的 ID——PGID。因为集群所有的 pool 由 Monitor 统一管理，所以 Monitor 可以为每个 pool 分配一个集群内唯一的 pool-id（在设计上，出于高可靠性的考虑，要求 Monitor 必须也是分布式的，因此为了保证生成 pool-id 的全局唯一性，实际上要求 Monitor 实现分布式一致性，当前采

用 Paxos 实现）。基于此，我们只需要为 pool 内的每个 PG 再分配一个 pool 内唯一的编号即可。我们假定某个 pool 的 pool-id 为 1，创建此 pool 时指定了 256 个 PG，那么容易理解这些 PGID 应当具有形如 1.0，1.2，...，1.255 这样的格式。

Ceph 通过 C/S（Client/Server）模式实现外部应用和存储集群之间的数据交互。任何时候，客户端需要访问集群时，首先由特定类型的 Client 根据其操作的对象名（例如针对 RBD 应用，对象名是由 RBD Client 负责生成的，形如“rbd_data.12d72ae8944a.0000000000000002a7”的字符串），计算出一个 32 位的哈希值，然后根据其归属的 pool 及此哈希值，通过简单的运算，例如取模，即可找到最终承载该对象的 PG。假定 pool-id 为 1 的 pool 当中的某个对象，经过计算后 32 位的哈希值为 0x4979FA12，该 pool 配置的 PG 数目为 256，则：

```
0x4979FA12 mod 256 = 18
```

由此，我们知道此对象由 pool 内编号为 18 的 PG 负责承载，同时可以得到其对应的完整 PGID 为 1.18。

一般而言，将每个对象映射至对应的 PG 时，我们并不会使用全部的 32 位哈希值（按照每个 OSD 承载 100 个 PG 估算，用掉全部 32 位哈希值一共需要集群拥有 42949672 个 OSD！），因此会出现不同对象被映射到同一个 PG 上的现象。例如我们很容易验证 pool 内的如下对象通过模运算同样会被映射到 PGID 为 1.18 的 PG 上：

```
0x4979FB12 mod 256 = 18
0x4979FC12 mod 256 = 18
0x4979FD12 mod 256 = 18
...
```

可见，针对这个例子，我们仅使用了这个“全精度”32 位哈希值的后 8 位。因此，如果 pool 内的 PG 数目可以写成 2^n 的形式（例如这里 256 可以写成 2^8），即可以被 2 整除时，容易验证前端每个对象执行到 PG 映射时，其低 n 比特是有意义的；进一步地，我们很容易验证此时归属于同一个 PG 的对象，其 32 位哈希值中低 n 比特都是相同的，基于此，我们将 2^n-1 称为 PG 的掩码，其中 n 为 PG 掩码的位数。相反，如果 PG 数目不能被 2 整除，即无法写成 2^n 形式，仍然假定此时其最高位为 n（说明：从 1 开始计数），则此时我们使用普通的取模操作无法保证“归属某个 PG 下的所有对象其低 n 比特都相同”这个特性。例如假定 PG 数目为 12，此时 $n = 4$，容易验证对于如下序列，其哈希值只有低 2 比特是相同的：

```
0x00 mod 12 = 0
0x0C mod 12 = 0
0x18 mod 12 = 0
0x24 mod 12 = 0
...
```

因此需要针对普通的取模操作加以改进，以保证针对不同的输入，当其结果相等时，维持“这些输入具有尽可能多的相同的低比特位”这样一个相对“稳定”的特性。

一种改进的方案是使用掩码来替代取模操作，例如仍然假定 PG 数目对应的最高比特位为 n，则其掩码为 2^n-1，需要将某个对象映射至 PG 时，直接执行 hash & (2^n-1) 即可。但是这个改进方案仍然存在一个潜在问题——如果 PG 数目不能被 2 整除，那么直接采用这种方式进行映射会产生空穴，即将某些对象映射到一些实际上并不存在的 PG 上，如图 2-2 所示。

图 2-2 使用掩码方式进行对象至 PG 映射这里 PG 数目为 12，$n=4$，可见执行 hash & (2^4-1) 会产生 0 ～ 15 共计 16 种不同的结果而实际上编号为 12 ～ 15 的 PG 并不存在

因此需要进一步对上述改进方案进行修正。由 n 为 PG 数目中的最高比特位，必然有：

$$\text{PG 数目} \geqslant 2^{n-1}$$

即 $[0, 2^{n-1}]$ 内的 PG 必然都是存在的，于是可以通过 hash & $(2^{n-1}-1)$ 将那些实际上并不存在的 PG 重新映射到 $[0, 2^{n-1}-1]$ 区间，如图 2-3 所示。

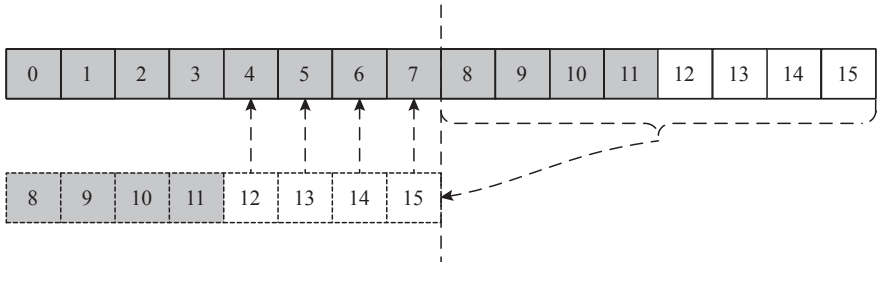

图 2-3 使用修正后的掩码方式对图 2-2 中的映射结果进行校正，效果等同于将这些空穴通过平移的方式重定向到前半个对称区间

这样，退而求其次，如果 PG 数目不能够被 2 整除，我们只能保证相对每个 PG 而

言，映射到该 PG 下的所有对象，其低 $n-1$ 比特都是相同的。改进后的映射方法称为 stable_mod，其逻辑如下：

```
if ((hash & (2^n - 1)) < pg_num)
      return (hash & (2^n - 1));
else
      return (hash & (2^(n-1)-1));
```

仍然以 PG 数目等于 12 为例，可以验证对于如下序列，采用 stable_mod 方式可以保证其输入的低 $n-1=3$ 比特都是相等的：

```
0x05 stable_mod 12 = 5
0x0D stable_mod 12 = 5
0x15 stable_mod 12 = 5
0x1D stable_mod 12 = 5
…
```

综上，无论 PG 数目是否能够被 2 整除，使用 stable_mod 都可以产生一个最好的、相对稳定的结果，这是 PG 分裂的一个重要理论基础。

Ceph 的主要设计理念之一是高可扩展性。初期规划的 Ceph 集群容量一般而言都会偏保守，无法应对随着时间推移而呈爆炸式的数据增长需求，因此扩容成为一种常态。对 Ceph 而言，需要关注的一个问题是扩容前后负载在所有 OSD 之间的重新均衡——即如何能让新加入的 OSD 立即参与负荷分担，从而实现集群性能与规模呈线性扩展这一优雅特性。在 Ceph 的实现中，这是通过 PG 自动迁移和重平衡实现的，然而遗憾的是存储池中的 PG 数目并不会随着集群规模增长而自动增加，这在某些场景下往往会导致潜在的性能瓶颈（相关的分析我们将在第 4 章中进行）。为此，Ceph 提供了一种手动增加存储池中 PG 数目的机制。当集群中的 PG 数目增加后，新的 PG 会被随机、均匀地映射至所有 OSD 之上，又因为存储池中的 PG 数目发生了变化，由图 2-1 所示，此时执行前端对象至 PG 的映射时，作为 stable_mod 输入之一的 PG 数目已经发生了变化，所以会导致某些对象也被从老 PG 中重新映射至新 PG 中，因此需要同步转移这部分对象。上述过程称为 PG 分裂，具体步骤如下：

1）Monitor 检测到 pool 中的 PG 数目发生修改，发起并完成信息同步，随后将包含了变更信息的新 OSDMap 推送至相关的 OSD。

2）OSD 接收到新 OSDMap，与老 OSDMap 进行对比，判断对应的 PG 是否需要进

行分裂。如果新老 OSDMap 中某个 pool 的 PG 数目发生了变化（Ceph 目前只支持增加 pool 中的 PG 数目），则需要执行分裂。

3）假定这个 pool 老的 PG 数目为 2^4、新的 PG 数目为 2^6，以 pool 中某个老 PG（PGID = Y.X，其中 X = 0b$X_3X_2X_1X_0$）为例，容易验证其中已有对象的哈希值都可以分成如图 2-4 所示的四种类型（按前面的分析，分裂前后，对象哈希值中只有低 6 比特有效，图中只展示这 6 个比特）：

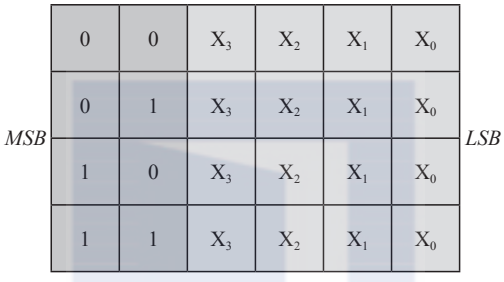

图 2-4 PG 分裂（2^4–>2^6）过程中，对象哈希值特征

依次针对这四种类型的对象使用新的 PG 数目再次执行 stable mod，结果如表 2-2 所示。

表 2-2 针对 PG（Y.X）使用新的 PG 数目（2^4–>2^6）重新执行 stable_mod

对象哈希值（32 位，二进制）	执行 stable_mod 结果（X=0b $X_3X_2X_1X_0$）
0b???? ???? ???? ???? ???? ???? ??00 $X_3X_2X_1X_0$	X
0b???? ???? ???? ???? ???? ???? ??01 $X_3X_2X_1X_0$	1 * 16 + X
0b???? ???? ???? ???? ???? ???? ??10 $X_3X_2X_1X_0$	2 * 16 + X
0b???? ???? ???? ???? ???? ???? ??11 $X_3X_2X_1X_0$	3 * 16 + X

可见，切换到新的 OSDMap 后，因为 PG 数目发生了变化，仅有第一种类型（X_5X_4 = 00）的对象使用 stable_mod 方法仍然能够映射到老的 PG 上，其他三种类型的对象并无实际 PG 对应，因此我们需要创建 3 个新的 PG（容易验证这 3 个新的 PG 在存储池内的编号可以通过 (m * 16) + X, m = 1,2,3 得到），来转移部分来自老 PG 上的对象，使得客户端更新到最新的 OSDMap 后，使用 stable_mod 方法仍然能够重定向到正确的 PG 上。

针对所有老的 PG 重复上述过程，最终可以将存储池中的 PG 数目调整为原来的 4 倍，即 2^6，又因为在调整 PG 数目的过程中，我们总是基于老的 PG（称为祖先 PG）产生新的孩子 PG，并且新的孩子 PG 中最初的对象全部来自于老的 PG，所以这个过程被形

象地称为 PG 分裂。在 FileStore 实现中，因为 PG 对应一个文件目录，其下的对象全部使用文件保存，所以出于索引效率和 PG 分裂的考虑，FileStore 对目录下的文件进行分层管理。采用 stable_mod 执行对象至 PG 的映射后，因为同一个 PG 下的所有对象，总是可以保证它们的哈希值至少低（$n-1$）比特是相同的，所以一个自然的想法是使用对象哈希值逆序之后作为目录分层的依据（同时也能满足上层应用需要按照某种顺序遍历 PG 下所有对象的需求），例如假定某个对象的 32 位哈希结果为 0xA4CEE0D2，则其可能的一个目录层次为 ./2/D/0/E/E/C/4/A/0xA4CEE0D2，这样我们最终可以得到一个符合常规的、树形的目录结构。例如仍然假定某个 PG 归属的 pool 分裂之前共有 256 个 PG（此时 n=8，对应的掩码为 2^8-1= 255），PG 对应的 PGID 为 Y.D2，则某个时刻其一个可能的目录结构为（这里仅仅为了举例说明问题，实际上触发目录分裂需要当前目录下的文件数目超过某个阈值）：

```
.
└── 2
    └── D
        ├── 0
        │   ├── 0x324140D2
        │   └── 0x78AF30D2
        ├── 1
        │   └── 0x9898A1D2
        ├── 2
        │   ├── 0x1578A2D2
        │   └── 0x787AE2D2
        ├── 3
        │   ├── 0x789713D2
        │   └── 0x8976E3D2
        ├── 4
        │   └── 0x123174D2
        ├── 5
        │   └── 0xAA3CE5D2
        ├── 6
        │   ├── 0x6873F6D2
        │   └── 0xFABE36D2
        ├── 7
        │   └── 0xA675F7D2
        ├── 8
        │   ├── 0x9786A8D2
        │   └── 0x990028D2
        ├── 9
        │   └── 0x787329D2
```

```
├── A
│   └── 0x67812AD2
├── B
│   └── 0x7760FBD2
├── C
│   └── 0x798ADCD2
├── D
│   └── 0x787ABDD2
├── E
│   ├── 0x014ACED2
│   └── 0x123EEED2
└── F
    ├── 0x018BCFD2
    └── 0xBCC34FD2
```

如果分裂为4096个PG（此时 $n = 12$，对应的掩码为 $2^{12}-1 = 4095$），则原来每个PG对应分裂成16个PG，仍然以PGID为Y.D2的PG为例，通过简单计算可以得到这15个新PG的PGID分别为：

```
Y.1D2
Y.2D2
…
Y.ED2
Y.FD2
```

可见这些新的PG对应的对象分别存储于老PG的如下目录：

```
./2/D/1/
./2/D/2/
…
./2/D/E/
./2/D/F/
```

因此，此时可以不用移动对象（文件），而是直接修改文件夹的归属即可完成底层对象在PG之间的转移。

引入PG分裂之后，如果仍然直接使用PGID作为CRUSH输入，据此计算新增孩子PG在OSD之间的映射结果，因为此时每个PG的PGID都不相同，那么将引发大量新增孩子PG在OSD之间迁移。考虑到分裂之前PG在集群OSD之间的分布已经趋于均衡，更好的办法是让同一个祖先诞生的孩子PG和其保持相同的分布，这样当分裂完成之后，整个集群的PG分布仍然是均衡的。为此，每个存储池除了记录当前的PG数目之外，为了应对PG分裂，还需要记录分裂之前祖先PG的个数，后者称为PGP数目（pgp_num）。

最后，利用 CRUSH 将每个 PG 分布到不同安全域中的 OSD 之上时，我们不是直接使用 PGID，而是转而使用其在存储池内的唯一编号针对 PGP 数目执行 stable_mod，再和 pool-id 一起，哈希之后作为 CRUSH 的特征输入（即 x），从而保证每个孩子 PG 和其祖先取得相同的 CRUSH 计算结果（因为此时同一个祖先的孩子 PG 产生的 x 都相同）。

新的 BlueStore 实现中，因为同一个 OSD 下所有对象共享一个扁平的寻址空间，所以 PG 分裂时，甚至不需要类似 FilesStore 执行分裂时对象在不同文件夹之间转移的过程，因而更加高效。PG 对应的磁盘数据结构称为 bluestore_cnode_t（后续简称 cnode），定义见表 2-3，目前仅包含一个字段，用于指示执行 stable_mod 时，PG 的掩码位数。

表 2-3 bluestore_cnode_t

成员	含义
bits	指示归属于 PG 的对象，在执行到 PG 的映射过程中，其 32 位的全精度哈希值（从低位开始计算）有多少位是有效的

2.2.2 对象

BlueStore 中的对象非常类似于文件，例如每个对象拥有 BlueStore 实例内唯一的编号、独立大小、从 0 开始进行逻辑编址、支持扩展属性等，因此对象的组织形式，也是采用类似文件的形式，基于逻辑段（extent）进行的。

参考文件系统，原理上，每个 extent 都可以写成形如——`{offset, length, data}` 的三元组形式，各成员含义如表 2-4 所示。

表 2-4 extent 抽象类型

成员	含义
offset	对象内逻辑偏移，从 0 开始编址
length	逻辑段长度
data	逻辑段包含的数据，为抽象数据类型

其中 data 是个抽象数据类型，主要用于从磁盘上索引对应逻辑段包含的用户数据。考虑到磁盘空间碎片化严重时，我们可能无法保证为每个逻辑上连续的段（即 extent）分配物理上也连续的一段空间，所以逻辑段和磁盘上的物理空间段应该是一对多的关系，因此 data 在设计上主要包含一些物理空间段的集合，每个段对应磁盘上的一块独立存储空间，BlueStore 称为 bluestore_pextent_t（简称 pextent），其成员如表 2-5 所示。

表 2-5 bluestore_pextent_t

成员	含义
offset	磁盘上的物理偏移
length	长度

前面已经提及，出于访问效率考虑，磁盘的最小访问单元不可能设计为比特，而是块大小，所以 pextent 中的 offset 和 length 也必须是块大小对齐的。因此，如果 extent 中的 offset 不是块大小对齐的，则上述物理空间强制对齐约束会使得 extent 的逻辑起始地址和对应的物理起始地址之间产生一个偏移；相应的，如果 extent 中的 length 不是块大小对齐的，则 extent 中的逻辑结束地址和对应的物理结束地址也可能产生一个偏移。后面我们将会看到，这两个偏移的存在将大大增加数据处理的难度。

因为 extent 是对象内的基本数据管理单元，所以很多扩展功能——例如数据校验、数据压缩、对象间的数据共享等，都是基于 extent 粒度实现的，下面分别予以阐述：

（1）数据校验

数据校验指对数据实施正确性检测。任何我们使用的计算机组件都不是完美的，因此都可能产生静默数据错误（Silent Data Corruption）。静默数据错误，顾名思义，是不能被计算机组件自身觉察的错误，因此其潜在危害性极大。单个组件静默数据错误出现几率极低，参考欧洲原子能研究机构——CERN 的研究报告[⊖]，一般在 10^{-7} 水平，因而极易被忽略，但是因为每个组件都可能发生（例如磁盘、RAID 5 控制器、内存等），如果考虑长时间海量数据的场景，那么静默数据错误的产生几乎是必然的。

解决静默数据错误的主要手段是引入校验和，即采用某种特定的校验算法，使用原始数据作为输入，得到固定长度输出，此输出即为校验和。之后将校验和与原始数据分开存储，后续读取数据时，分别读取原始数据与校验和，然后针对原始数据重新计算校验和。如果此校验和与之前保存的校验和相等，那么认为数据是正确的，反之则认为数据出错。这里存在的一个潜在问题是：如果重新计算出来的校验和，与之前保存的校验和不相等，那么我们到底应该认为是原始数据出错，还是保存的校验和出错了呢？BlueStore 的解决方案比较简单，它直接将校验和单独使用 kvDB 保存，借助于数据库的 ACID 特性，我们知道校验和总是可靠的，因此如果重新计算出来的校验和与 kvDB 保存的校验和不一致，则可以断定是原始数据出错。

⊖ https://indico.cern.ch/event/13797/contributions/1362288/attachments/115080/163419/Data_integrity_v3.pdf

一种设计良好的校验算法应当具有极低的冲突概率（指使用不同输入得到相同输出的概率，显然冲突概率越低，说明校验算法发生误检的可能性越小，对应的校验算法越好），当然这也取决于输出长度，例如针对同一种算法，输出 64 位校验和显然比输出 32 位校验和冲突的概率要低得多，因为前一种算法可能输出 2^{64} 种不同的结果，而后一种算法只能输出 2^{32} 种不同的结果。此外，冲突概率一般还和校验算法的执行效率相关，一般而言，更低的冲突概率总是对应更复杂的计算过程，使得整个校验算法执行效率低下，因此实际选择校验算法时，也不能一味追求低冲突概率，而是需要根据自身需求在这两者（冲突概率和执行效率）之间进行权衡。

当前主流的校验算法有 CRC、xxhash 等（BlueStore 默认都支持），可以根据配置项灵活进行选择。

（2）数据压缩

数据压缩针对原始数据进行转换，以期得到长度更短的输出，目的是为了节省磁盘空间。与数据校验不同，数据压缩的转换过程必须是可逆的，即采用输出作为输入，我们反过来可以还原得到原始数据，后面这个过程称为数据解压缩。

常见的压缩算法大多是基于模式匹配的，因此一般而言，其过程迭代次数决定了压缩收益，这意味着对大多数压缩算法而言，更高的压缩比几乎总是对应更多的性能损耗（因为需要更多的迭代次数，即更长的压缩时间）。因此实际选用压缩算法时，同样需要考虑在这两者（压缩收益和执行效率）之间进行权衡。

采用数据压缩算法需要固化两个关键信息——一是选用的压缩算法；二是压缩后的数据长度。BlueStore 使用压缩头保存这两个信息，如表 2-6 所示。

表 2-6　bluestore_compression_header_t

成员	含义
type	压缩算法类型
length	数据压缩后的长度

需要注意的是：不同于其他元数据，压缩头是和压缩后的数据一起保存的，这主要是因为存储压缩头也需要额外的磁盘空间，所以需要纳入到压缩后的数据之中，据此一并计算压缩收益，即压缩率。当压缩率小于某个阈值（目前为 0.875，亦即要求至少压缩掉原始数据长度的 1/8），即如果采用压缩算法获得的净收益太小，BlueStore 将拒绝压缩。

引入数据压缩的一个重大缺陷在于其对不完全覆盖写的负面影响。假定我们需要针对某个压缩后的 extent 执行不完全覆盖写，一般而言有两种方案：一是先将对应的内容从磁盘上读出来，解压缩，然后再执行正常的覆盖写流程；二是直接执行重定向写，即

完全重新分配一个 extent，且允许其与需要被覆盖写的 extent 存在部分重合的内容。其中，方案一的缺点在于严重影响写效率；方案二的缺点一是存在空间浪费，二是增加了元数据，使得读操作效率降低。而且因为压缩可以重复进行，即新分配的 extent 又可以再次执行压缩，所以多次不完全覆盖写后，采用方案二对象当中的部分内容可能被多个 extent 重复保存多次[⊖]，进一步浪费空间和降低性能，有违引入压缩算法的初衷。

目前 BlueStore 是基于方案二实现数据压缩策略的，因为这远非一种完美的压缩策略，所以默认没有开启。

（3）数据共享

数据共享主要是指 extent 在不同对象间的共享，基于 extent 的数据共享一般由对象克隆操作引入。当某个 extent 的数据被多个 extent 共享时，需要使用一个中立结构来表明这些数据的共享信息。这些信息主要包含共享数据的起始地址、数据长度和被共享的次数。因此，这个中立结构由形如——`{offset, length, refs}` 三元组记录组成，BlueStore 称为 bluestore_shared_blob_t，其主要内容是一张基于 extent 的引用计数表。因为这张表在对象之间共享，所以需要和对象分开，单独进行存储。

extent 之间除了数据共享之外，还存在一种比较特殊的共享方式——存储空间共享。因为 BlueStore 自身管理磁盘空间，所以可以自定义最小可分配空间。一般情况下，这个最小可分配空间和磁盘块大小相等，但是出于分配效率考虑（更大的块大小对应更小的位图，从而也对应着更快的索引速度），也可以将其提升为块大小的整数倍。例如假定磁盘块大小为 4K、最小可分配空间为 16K，此时即使写入 1 个字节的数据，也需要为之分配 16K 的磁盘空间，但是实际上这已分配的 16K 空间中，只有一个 4K 的块真正被使用，其他 3 个块均未被使用，因此存在被其他 extent 共同使用的可能。

至此，我们已经讨论了 extent 三元组中的 data 抽象数据类型目前所期望支持的所有特性，据此可以定义其磁盘数据结构如表 2-7 所示（BlueStore 称为 bluestore_blob_t，简称 blob）。

表 2-7 bluestore_blob_t

成员	含义
extents	磁盘上的物理段集合，单个 extent 的类型为 bluestore_pextent_t

⊖ 即存在垃圾数据。在 Luminous 版本中，BlueStore 引入了一种针对这类垃圾数据进行回收的机制，称为 GC（Garbage Collector）。

（续）

成员		含义
flags	FLAG_MUTABLE	blob 对应数据可以被修改，例如没有被压缩
	FLAG_COMPRESSED	blob 对应数据经过压缩
	FLAG_CSUM	blob 对应数据经过校验
	FLAG_HAS_UNUSED	blob 物理空间中包含未被使用的块
unused		blob 所有未被使用物理块的集合。以块大小为单位对整个 blob 的物理空间进行划分，使用单个比特标识每个区域的状态，如果置位，表明该区域包含用户数据；反之表明该区域不包含任何用户数据
compressed_length_orig		压缩控制。 分别对应压缩前后用户数据长度
compressed_length		
csum_type		校验控制。 csum_type 用于指定一种校验算法类型，典型如 CRC32。 csum_chunk_order 用于指定计算校验和时，每次输入的原始数据块大小，例如共计 16K 的原始数据，csum_chunk_order 为 12，则每次输入 4K（2^{12}）原始数据计算其校验和，共需要 16K/4K = 4 次才能完成校验。值得注意的是，每次计算得到的校验和，其长度是固定的，具体取决于所选择的校验算法，例如选择 CRC32，则每次输入 4K 的原始数据，总是得到固定 4 个字节的校验和。 csum_data 用于保存具体的校验和。 因为 BlueStore 使用 kvDB 保存校验和，测试结果表明每个键值对中键或者值长度超过一定范围时，kvDB 的操作效率会显著降低，所以出于性能考虑，我们一般会限制单个 blob 所能保存数据的最大长度，进而限制产生的检验和长度。假定此时校验算法选择 CRC32，blob 允许保存的最大数据长度为 512KB，那么至多会产生 (512K/4K)*4 = 512 字节长度的校验数据，以免显著降低 kvDB 的访问性能（注意：出于同样的原因，我们会将对象的 extent map 切成若干个更小的集合进行存储，参考下文）
csum_chunk_order		
csum_data		

相应的 extent 磁盘数据结构见表 2-8。

表 2-8　extent

成员	含义
logical_offset	逻辑段起始地址
length	逻辑段长度
blob	负责将逻辑段内的数据映射至磁盘，参考表 2-7
blob_offset	当 logical_offset 不是磁盘块大小自然对齐时，将对应逻辑段内的数据通过 blob 映射到磁盘物理段（或者集合中）会产生物理段内的偏移，这个偏移称为 blob_offset；反之如果 logical_offset 天然块大小对齐，则 blob_offset 始终为 0

表中 logical_offset、length 和 blob_offset 几个成员的关系如图 2-5 所示。

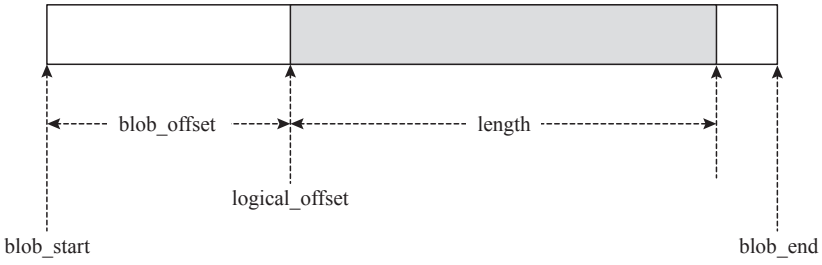

图 2-5 extent 中的逻辑段与物理段之间的关系

同一个对象下所有 extent 可以进一步组成一个 extent map，据此可以索引本对象下的所有有效数据。需要注意的是：因为支持稀疏写，所以 extent map 中的 extent 可以是不连续的（即 extent map 中的相邻两个 extent，前一个 extent 的逻辑结束地址小于后一个 extent 的逻辑起始地址），因此 extent map 中可能存在空穴。此外，如果单个对象内的 extent 过多（例如小块随机写），或者磁盘碎片化严重等，都可能导致整个 extent map 编码之后变得十分臃肿，从而严重影响 kvDB 的访问效率，所以需要对超过一定大小的 extent map 进行分片，并且将这些分片信息从归属对象的元数据信息中独立出来，单独保存。这些分片信息 BlueStore 称为 shard_info，其关键数据成员如表 2-9 所示。

表 2-9 shard_info

成员	含义
offset	分片对应的逻辑起始地址
bytes	分片编码后的长度

针对 extent map 进行分片并不是一个十分精确的过程，即我们只需要保证每个分片进行编码后，其长度落在一个指定的、相对宽松的范围内即可。因此实现时，总是基于上一次的分片信息预先估算当前 extent map 每个 extent 编码后的平均大小，再以此计算本次分片过程中每个新分片的逻辑起始地址和逻辑结束地址，作为后续真正将 extent map 编码存盘时的依据。由于不是基于精确计算（即不是先将每个 extent 编码之后，再确定分片信息），这个分片过程有时可能需要进行多次；此外，如果某个 blob 被两个相邻的 extent 共享，而这两个 extent 又恰好隶属于两个不同的分片范围，那么会因为该 blob “跨越”两个分片而无法确定其归属，此时 BlueStore 将优先针对该 blob 执行分裂操作，如果分裂失败（例如由于该 blob 被压缩、设置了 FLAG_SHARED 标志等），则将该 blob 加入 spanning_blob_map，后续将整个 spanning_blob_map 和对象的其他元数据（例如 onode，参考下文）一并作为单条记录进行保存（BlueStore 假定单个对象内这类 blob 数量不是很多）。

针对 extent map 分片的另一个好处在于可以实现对于 extent 的按需加载，即只有需要访问特定分片范围内的 extent 时，才从磁盘上读出对应的分片，通过解码还原出该分片内的所有 extent 至 extent map；同样，当某个分片内的 extent 被修改时，也只需要更新对应的分片即可。这样做一方面减少了常驻内存的元数据数量，另一方面也减少了操作对象上下文时波及的元数据数量，两者都有助于改善性能。

综上我们可以定义 extent map 的磁盘数据结构（实际上也包含部分内存数据结构，BlueStore 未进行严格隔离）如表 2-10 所示。

表 2-10 extent map

成员		含义
extent_map		逻辑段集合
shards	key	将 extent map 对应的分片写入 kvDB，需要全局唯一的 key，这个 key 由对应的对象 key + 分片的起始逻辑地址 + 固定的“x”后缀组成
	offset	分片的逻辑起始地址
	shard_info	参见表 2-9。 因为 extent map 独立于对象存储，所以对象的磁盘结构中也需要相应的固化一些关键信息，用于从 kvDB 中还原 extent map
	loaded	对象上下文从 kvDB 加载时，我们并不需要同步加载完整的 extent map，只有当对应范围内的 extent 被访问时，才从 kvDB 加载包含该 extent 的分片。本标志位指示对应的分片已经从 kvDB 中加载
	dirty	指示对应分片中的 extent 被修改过，后续需要对本分片重新编码、存盘
spanning_blob_map		对象内所有跨越两个分片的 blob 集合

至此，我们已经分析得到了 BlueStore 中构建一个对象的所有关键元素，下面我们开始介绍对象的磁盘数据结构，BlueStore 称为 bluestore_onode_t，简称 onode。

介绍 onode 之前，首先澄清一个容易混淆的概念——即 BlueStore 看到的对象和上层看到的对象，这两者实际上并不相同。原则上，BlueStore 只需要通过一个唯一索引和上层对象建立联系即可，因为上层的每个对象具有全局唯一的 OID，所以这个索引可以直接由 OID 充当。但是随着 Ceph 本身的发展，我们不断地往对象中追加诸如命名空间、快照等一些与特定应用绑定的关键信息，这些信息虽然对 BlueStore 而言并不可见，但是会使得 OID 全局唯一性丧失（例如快照的引入，使得两个不同的对象拥有相同的 OID 但是不同的快照 ID），所以这些额外信息也需要在执行上层对象至 onode 的映射过程中一并考虑。

老的 FileStore 直接将上层对象进行转义后作为文件名存储（如果转义后的文件名仍然很长，例如超过 255 个字符，FileStore 还需要再次执行哈希），BlueStore 做法类似，但是因为所有对象相关的元数据都使用 kvDB 存储，所以转义后的对象名不是作为文件名而是作为 kvDB 表中的唯一索引。转义的过程主要有两个关注点：一是在包含所有必要信息的前提下，尽可能地减少转义后的字符串长度，这是因为无论转义后的对象名作为文件名还是 DB 索引，长度之于性能都有着至关重要的影响；二是区分对象名中定长和非定长部分——原始对象名中，既有诸如 OID 和命名空间等长度不定的字符串，也有哈希、快照 ID 等定长整数类型，实现时需要将这两种类型进行区别对待，以提升编码效率。转义过程具体描述如下：

1）所有定长整型部分，直接转换为等宽的字符串，例如 32 位的哈希值 0x30313233，编码后对应字符串“1234”（注意，存储一个 char 类型需要一个 Byte，即 8 个比特；另外能够这样做对于选用的 kvDB 有要求，例如不能将 0x00 作为默认的字符串结束符）。

2）所有变长字符串部分，需要进行字符串转义，以界定其结束位置。具体实现时，BlueStore 将原字符串中任意小于等于 '#' 的字符，都转换为长度固定为 3、形如 "#XY" 的字符串；将原字符串中任意大于等于 '~' 的字符，都转换为长度固定为 3、形如 "~XY" 的字符串；最后再将转换后的字符串使用 '!' 作为结束符（因为 '!' 小于 '#'，所以如果 '!' 出现在原始字符串中，将被转义，亦即原始字符串中经过处理后不可能出现 '!' 字符）。

3）依次将原始对象名中的所有成员按照上述原则进行预处理后，再按照固定顺序直接拼接成一个完整的字符串即可。

建立上层对象至 onode 的映射关系后，即可以通过对象名索引到 onode。和 FileStore 类似，onode 也包含四个部分，分别为：数据、扩展属性、omap 头部和 omap 条目。数据及扩展属性和文件系统中文件相关概念类似；omap 存储的内容则和扩展属性十分类似，但是两者分别位于不同的地址空间，换言之，omap 当中的某个条目可以和扩展属性拥有相同的键但是不同的值（内容）；另外，一般的文件系统对于扩展属性长度都有限制（典型如 XFS 限制单个扩展属性长度至多为 4K），但是使用 omap 则无此限制，从这个角度而言，也可以认为扩展属性只适用于保存少量小型属性对，而 omap 则适用于保存大量大型属性对。基于此，BlueStore 中扩展属性是和 onode 一并保存的，而 omap 则分开保存，表 2-11 罗列了目前为止所有与 onode 相关的、需要在 kvDB 中进行存储的条目类型：

表 2-11 kvDB 当中与 onode 相关的条目类型

条目类型	唯一索引
onode	固定前缀“O”+ onode-key + 固定后缀“o”
extent map	onode 索引 + offset + 固定后缀“x”
omap	固定前缀“M”+ BlueStore 实例内全局唯一 id
ref map(bluestore_shared_blob_t)	固定前缀“X”+ BlueStore 实例内全局唯一 id

由表 2-11 可见，对于 extent map，我们生成索引的策略稍有不同，不是一味追求缩短其长度，而是使用了对应 onode 本身的索引作为固定前缀，这主要是考虑了局部性原理——我们在加载了 onode 之后，有很大的概率需要继续加载 extent map，以方便接下来进行数据读写，所以将每个 onode 和其关联的 extent map 相邻存储，按照局部性原理可以提升数据库的访问性能。

在本节的最后，我们给出 onode 的磁盘数据结构，如表 2-12 所示。

表 2-12 bluestore_onode_t

成员		含义
nid		逻辑标识，单个 BlueStore 实例内唯一。 nid 主要用来保证对象构建关联的 omap 索引时的唯一性（参见表 2-11）
size		对象大小
attrs		扩展属性对
flags	FLAG_OMAP	对象关联的 omap 是否使用
extent_map_shards		对象关联的 extent map 的分片概要信息（参见表 2-9），用于从 kvDB 中索引某个具体分片
expected_object_size		上层应用提示信息，用于优化基于对象的读、写、压缩控制等策略
expected_write_size		
alloc_hint_flags		

2.3 缓存管理

2.3.1 常见的缓存淘汰算法

在现代计算机系统中，为了适配不同组件（例如 CPU、内存、磁盘等）之间处理数据速度之间的差异，一般都需要使用缓存。缓存最开始被集成在 CPU 内部，特指 CPU 高速缓存，如今概念已被扩充，常见的内存即是一种缓存，甚至磁盘内部也有缓存。一般而言，内存容量远大于 CPU 高速缓存容量，磁盘容量则远大于内存容量，因此无论是哪一种层次的缓存都面临一个同样的问题：当容量有限的缓存空闲页面全部用完后，又

有新的页面需要被添加至缓存时，如何挑选并舍弃原有的部分页面，从而腾出空间放入新的页面？解决这个问题的算法被称为缓存淘汰（也称为替换）算法，典型的缓存淘汰算法有如下两种：

（1）LRU（Least Recently Used）

LRU 算法总是淘汰缓存中当前保存的所有页面中最长时间未使用的页面。LRU 算法是基于时间局部性原理提出的：如果缓存中的某个页面正在被访问，那么在近期内它很有可能再次被访问。基于 LRU 产生了许多变种，典型如增强时钟算法。如果请求队列体现出了很好的时间局部特性，那么可以证明此时 LRU 是最优算法。此外，LRU 算法容易实现，也是其优点之一。

（2）LFU（Least Frequently Used）

LFU 算法是基于另外一种页面访问模型 SDD 而提出的：它假定 CPU 读写主要存储设备中每个页面（对应磁盘，则以块为单位）的概率是独立的，并且整体上遵循一个固定的概率分布模型（例如正态分布）。符合 SDD 模型的系统中，有的页面被访问的频率很高，而有的相对较低，因此保留缓存中访问频率较高的页面可以提高命中率。基于 SDD 产生了 LFU 算法，它总是选择淘汰缓存中访问频率最低的页面。然而历史访问频率较高的页面，并不见得在将来很长一段时间内都会被持续访问，因此 LFU 算法一般也需要兼顾频率的时效性。

LRU 和 LFU 算法只能在请求序列呈现明显的时间局部性或者空间局部性时取得较高的收益，因此单独而言都不是一种普适性的算法。基于 LRU 和 LFU 算法产生了 ARC（Adaptive Replacement Cache）算法[⊖]。ARC 综合考虑了 LRU 和 LFU 算法的长处，同时使用两个队列对缓存中的页面进行管理：MRU（Most Recently Used）队列保留最近访问过的页面；MFU（Most Frequently Used）队列保留最近一段时间内至少被访问过两次的页面。ARC 算法的关键之处在于两个队列的长度是可变的，会根据请求序列所呈现的特性自动进行调整，以取得在 LRU 算法和 LFU 算法之间的某种平衡：当系统中的请求序列呈现明显的时间局部性时，此时 MFU 队列长度为 0，ARC 退化为 LRU 算法；反之当系统中的请求序列呈现明显的空间局部性时，此时 MRU 队列长度为 0，ARC 退化为 LFU 算法。因此，无论请求序列呈现何种特性，ARC 通过自身参数的调整，都能够始终保持良好的缓存命中率，同时，因为这些参数的调整过程不需要人工干预，所以 ARC 是自适

⊖ https://www.usenix.org/legacy/event/fast03/tech/full_papers/megiddo/megiddo.pdf

应的。ARC 的缺点在于维护队列众多（MRU、MRF 队列及其对应的影子队列，共计 4 个）、算法复杂因此执行效率较低，在一些专有系统，特别是对于时延敏感的系统——例如数据库系统当中不是特别适用。

2Q[⊖]（2Q 基于 LRU/2 发展而来，后者本质上是一种 LFU 算法）是一种针对数据库，特别是关系型数据库系统优化的缓存淘汰算法。数据库系统因为需要保证每个操作的原子性，经常存在多个事务操作同一块热点数据的场景，因此针对数据库系统的缓存淘汰算法主要聚焦在如何识别多个并发事务之间的数据相关性问题。与 ARC 类似，2Q 也使用了多个队列来管理整个缓存空间，分别称为 A1in、A1out 和 Am。这些队列都是 LRU 队列，其中 A1in 和 Am 是真正的缓存队列，A1out 则是 A1in 和 Am 的影子队列，即 A1out 只保存相关页面的管理结构而不保存真实数据。新页面总是被首先加入 A1in，当某个页面在 A1in 期间被频繁访问时，2Q 认为这些访问是相关的，不会针对该页面执行任何热度提升操作，直至其被正常淘汰至 A1out。当 A1out 中某个页面被再次命中时，2Q 认为这些访问不再相关，此时执行页面热度提升，将其加入 Am 队列头部；Am 队列中的页面再次被命中时，同样将其转移至 Am 头部进行页面热度提升；从 Am 中淘汰的页面也进入 A1out。参考上述过程，2Q 实现的关键在于使用 A1in 队列来识别真正的热点数据——即如果缓存中的同一个页面在一个较短的时间段内被连续索引，则将这些索引视作“相关的”，不进行重复统计。这个时间段称为“相关索引间隔”（Correlated Reference Period），在 2Q 的实现中取决于 A1in 队列的容量。同理，A1out 的容量决定了一个页面被从 A1in 或者 Am 当中淘汰时，其之前累计的访问热度还能持续多长时间，称为“热度保留间隔”（Retained Information Period）。“相关索引间隔”和“热度保留间隔”是 2Q 判定页面热度的主要依据。在 2Q 的实现中，因为这两个参数的调整基于缓存容量自动进行，所以 2Q 的实现是极其高效的，特别适合于类似数据库系统这类时延敏感的应用。

综上，我们讨论了 LRU 和 LFU 两种基本缓存淘汰算法，以及由它们发展而来的 ARC 和 2Q 算法。实际上，任何算法都不是万能的，原因在于缓存和次级存储设备容量之间存在的巨大差异，因此无法建立两者之间的一一对应关系，而缓存淘汰算法的要义在于对请求序列进行预测，尽可能保留将来使用概率高的页面而淘汰将来使用概率低的页面。因此如果请求序列完全随机（即访问次级存储设备中任何一个位置数据的概率都

⊖ http://www.vldb.org/conf/1994/P439.PDF

是相等的），那么任何算法都不可避免存在误淘汰的可能，并且误淘汰的概率与所使用的缓存大小成反比。

2.3.2 BlueStore 中的缓存管理

BlueStore 目前使用了两种类型的缓存算法：LRU 和 2Q。如前所述，2Q 非常类似于一个简化版本的 ARC，区别在于只使用一个影子队列，用于保存从 A1in 和 Am 队列当中被淘汰的空页面。参考 Theodore 和 Dennis 的测试结论[⊖]，推荐 A1in 和 Am 队列的容量配比为 1:1，A1out 队列保存的空页面数量则为 A1in 和 Am 队列所能保存的页面之和。该推荐配比在所有测试场景下都可以取得一个比较好的命中性能，因而具有普适性。另外，出于减少锁碰撞的目的，BlueStore 会实例化多个缓存（这主要是因为不同 PG 之间的客户端请求可以并发处理，所以为了提升处理性能，每个 OSD 相应会设置多个 PG 工作队列，BlueStore 中缓存实例数与之对应）。本节首先介绍 Cache 基类，然后介绍由 Cache 派生而来的 TwoQCache 类。至于 LRUCache，因为比较简单并且可以视作 TwoQCache 的一种特例，则不做展开分析。

Cache 的基本数据成员如表 2-13 所示。

表 2-13 Cache

成员	含义
logger	用于缓存命中率相关的统计
lock	互斥锁。缓存相关的所有操作，几乎都需要在 lock 的保护下进行
num_extents	当前缓存的 extent 总数
num_blobs	当前缓存的 blob 总数
last_trim_seq	BlueStore 使用 mempool 对自身使用的内存进行全局统计和追踪，并启用一个专门的监听线程，周期性的更新内存相关的统计数据。 该线程每被唤醒一次，即将内部的 mempool_seq 计数器加 1，同时更新内存使用相关的统计数据。在上层读写线程（也包括 BlueStore 自身的 sync 线程）的驱动下，BlueStore 通过 Cache 的 trim() 方法，使得整个 BlueStore 的内存使用不超过给定的阈值。trim() 通过比对 Cache 内置的 last_trim_seq 和 BlueStore 内置的 mempool_seq 两个独立计数器，即可感知相应的内存使用数据是否更新。如果没有更新（两个计数器相等），表明可以跳过本次 trim() 操作；否则将本计数器更新为 mempool_seq，然后执行 tim() 操作

BlueStore 的 Cache 既可以用于缓存用户数据，也可以用于缓存元数据。从设计的角

⊖ http://www.vldb.org/conf/1994/P439.PDF，P441

度来看，因为 2Q 特别适合于数据库应用，所以 BlueStore 使用 2Q 作为默认的缓存类型应该尽量用来缓存元数据而不是用户数据（事实上，目前 BlueStore 缓存元数据的比重受 bluestore_cache_meta_ratio 控制，默认为 0.9，亦即 BlueStore 中 cache 的 90% 用来缓存元数据）。

参考 2.2 节，BlueStore 大的元数据类型有两种：Collection 和 Onode，其中 Collection 对应 PG 在 BlueStore 当中的内存管理结构。考虑到单个 BlueStore 实例所能管理的 Collection 数量有限（Ceph 推荐每个 OSD 承载大约 100 个 PG），而且 Collection 管理结构本身比较小巧，所以 BlueStore 将所有 Collection 设计成常驻内存。Onode 则不同，一个 Collection 本身能够容纳的 Onode 仅受磁盘空间的限制，所以单个 BlueStore 实例能够管理的 Onode 数量和其管理的磁盘空间成正比，这决定了通常情况下 Onode 几乎不可能常驻内存，于是需要引入淘汰机制。因此 Cache 设计上主要面向两种类型的数据——用户数据和 Onode。

和其他类型的元数据类似（例如 OSDMap），Onode 也是直接采用 LRU 队列管理的（这意味着实际上 BlueStore 所有的元数据都不是采用 2Q 进行管理的，只有数据才是，有违我们引入 2Q 的初衷）。理论上可以直接将所有 Onode 在 Cache 中使用单个 LRU 队列管理，这样效率最高，但是因为每个 Onode 唯一归属于某个特定的 Collection，所以还需要引入一个中间结构，来建立 Onode 和其归属的 Collection 之间的联系，方便针对 Collection 级别的操作（例如删除 Collection 时，不需要完整遍历 Cache 中的 Onode 队列，逐个检查其与被删除 Collection 之间的关系）。这个中间结构称为 OnodeSpace，其关键数据成员如表 2-14 所示。

表 2-14 OnodeSpace

成员	含义
cache	表明自身归属于哪个 Cache 实例（BlueStore 可以包含多个 Cache 实例）
onode_map	查找表

基于同样的考虑，针对用户数据，除了在 Cache 中使用通用队列进行管理之外，也需要考虑建立它和上层管理结构之间的对应关系。考虑到 Onode 中 Extent 是管理用户数据的基本单元，而 Blob 则真正负责执行用户数据到磁盘空间映射，所以我们基于 Blob 引入一个中间结构——BufferSpace，负责建立每个 Blob 中用户数据到缓存之间的二级索引，其关键数据成员如表 2-15 所示。

表 2-15 BufferSpace

成员	含义
cache	表明自身归属于哪个 Cache 实例
buffer_map	查找表
writing	包含脏数据的缓存队列（所有归属于同一个 Blob 并且当前状态为 BUFFER_WRITING 的 Buffer 使用同一个 writing_list 管理）

顾名思义，BufferSpace 管理的基本单元是 Buffer，一个 Buffer 管理 Blob 当中的一段数据（注意：不一定是干净的数据）。Buffer 的关键数据成员如表 2-16 所示。

表 2-16 Buffer

成员		含义
space		表明自身归属于哪个 BufferSpace 实例
state	STATE_EMPTY	Buffer 当前没有保存任何数据。如果 Cache 类型为 2Q，表明对应的 Buffer 已经被淘汰，Buffer 目前位于 A1out 队列当中
	STATE_CLEAN	Buffer 当前保存了干净数据（Buffer 中的数据没有被改写，并且和磁盘数据一致）
	STATE_WRITING	Buffer 当前保存了脏数据（Buffer 中的数据被改写，并且和磁盘数据不一致），Buffer 不存在于 Cache 当中。 当对应的写操作完成，Buffer 状态会转化为 STATE_CLEAN，同时取决于 Buffer 的标志位，Buffer 会被加入到 Cache 当中或者删除
cache_private		当 Buffer 存在于 Cache 中时，例如 2Q，用于将 Buffer 在 Cache 的不同队列之间进行调整；也表示 Buffer 当前位于哪个队列
flags	FLAG_NOCACHE	指示当前写入的数据不是热点数据，写操作完成后，对应的 Buffer 直接释放，不需要转入 Cache
offset		三元组，分别对应数据（在 extent 当中的）逻辑起始地址、逻辑长度和数据本身
length		
data		
seq		对应写操作的序列号。写操作完成后，由回调函数按序列号批量处理对应的 Buffer，依据 Buffer 的 flags 写入 Cache 或者直接删除
lru_item		用于将 buffer 插入 Cache 的数据缓存队列
state_item		用于将 buffer 插入对应 BufferSpace 中的 writing_list

参考上表，Buffer 只有三种状态，因此对应的状态转换也比较简单，如图 2-6 所示：

另外，因为 Blob（对应 Extent）是我们操作用户数据的一个基本单位，所以我们对于缓存的操作一般也是基于 BufferSpace 的粒度进行的，而不是 Buffer 本身。据此，定义 BufferSpace 当中的一些关键方法，如表 2-17 所示。

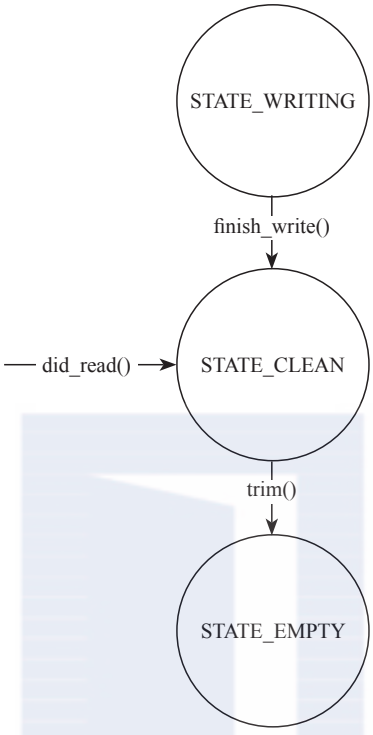

图 2-6　Buffer 状态转换

表 2-17　BufferSpace 公共接口

接口名称	含义
discard()	丢弃指定范围（offset + length）内所有波及的 Buffer 当中的数据。过程为：针对指定范围内的所有 Buffer，逐个测试其管理的数据范围与指定范围的交集，如果 Buffer 管理的数据范围包含在交集内，则完全删除 Buffer；否则执行分裂，产生 (至多两个) 新的 Buffer，这些 Buffer 仅包含原有的、不在交集内的部分数据。 discard() 适用于数据需要被覆盖写的场景（此时需要先丢弃缓存中的原有数据，然后再写入新的数据）
read()	读取指定范围（offset + length）内的数据，返回全部或者部分命中的数据段
did_read()	如果对应的读操作在 BufferSpace 未命中或部分未命中，则对应的读操作完成后，会创建一个或者多个 Buffer，关联未命中部分的数据，加入 BufferSpace（并最终加入 Cache）
write()	向 BufferSpace 中写入指定偏移和长度的脏数据，过程为：新建一个 Buffer，设置其状态为 STATE_WRITING，保存本次待写入的脏数据（offset + length + data）；然后执行 discard()，丢弃缓存内与本次写入范围重合的原有数据；最后将新创建的 Buffer 加入 BufferSpace 中的 writing_list
finish_write()	当写操作完成时，调用本方法将相关的 Buffer 从 writing_list 移除，同时按照 Buffer 的 flags，设置 Buffer 状态为 STATE_CLEAN，加入 Cache ；或者直接删除（如果对应的 Buffer 设置了 FLAG_NOCACHE 标志）

所有的 Onode 和 Buffer 最终都需要加入 Cache，进行全局热度识别和应用淘汰策略。在 BlueStore 的 2Q 实现中，这两类数据分别应用了不同的淘汰策略——针对 Onode 采用 LRU；针对 Buffer 才是真正的 2Q，因此 TwoQCache 相应的管理结构如表 2-18 所示。

表 2-18 TwoQCahce

成员	含义
onode_lru	全局 Onode LRU 队列
warm_in	参考 2Q 相关的理论分析，对应关系如下： warm_in：A1in buffer_hot：Am wam_out：A1out
buffer_hot	
warm_out	
buffer_bytes	全局缓存容量使用统计及每种队列容量使用统计，用于应用淘汰策略
buffer_list_bytes	

值得注意的是，虽然 TwoQCache 内部对 Onode 和 Buffer 分开管理，但是整个缓存空间却是全局共享的，因此需要为两者指定一个合适的分割比例。在实际测试中，因为用户数据的缓存和操作系统以及磁盘级的缓存有所重叠，所以社区也有人提议将整个 TwoQCache 全部用做 Onode 缓存，不过实际效果仍然待验证。此外，除了这些通用的缓存之外，还有其他一些出于特殊目的而引入的缓存——例如用于在 Onode 之间实现数据共享的 SharedBlob，我们在相关的章节再进行详细介绍。

2.4 磁盘空间管理

2.4.1 常见磁盘空间管理模式

块是磁盘进行数据操作的最小单位，任意时刻磁盘中的每个块只能是“空闲”或“占用”两种状态之一，因而使得采用一个比特表示磁盘中一个块的状态成为可能。如果将磁盘空间按照块进行切分、编号，然后每个块使用唯一的一个比特表示其状态，那么所有比特最终会形成一个有序的比特流，通过这个比特流，进而可以以块为粒度，索引磁盘中任意（存储）空间的状态，这种磁盘空间管理方式称为位图。

显然，位图和所管理的磁盘空间成正比。例如每管理 1GB 的磁盘空间，以块大小为扇区计，固定需要大小为 256KB 的位图。使用位图进行磁盘空间管理的一个重大缺陷在于如果位图无法全部装入内存，那么其管理效率就会大打折扣。早期的计算机系统中，因为磁盘容量有限（当然内存容量也有限），所以使得本地文件系统直接采用位图管理磁

盘空间成为可能。然而随着存储系统逐步朝着专业化、高端化的方向发展，一个现代存储系统中内存和磁盘两种存储介质的容量可能达到一个相当悬殊的比例——例如 EMC 的 VMAXe 宣称最大可支持的内存和磁盘容量分别为 512GB 和 1.3PB（1:2662），这样整个系统的位图将超过 300GB。显然，这种量级的位图如果作为一个整体，其索引效率是极其低下的，因此在大容量、集中式的高端存储系统中不可能被直接采用。

一个改进的方向是使用段管理磁盘空间。每个段包含两个成员：offset 和 length，前者指示被管理磁盘空间的起始地址，后者指示其长度。假定 offset 和 length 都是 64 位无符号整数，这样单个段管理的磁盘空间范围为 $[0, 2^{64}(16EB)]$，而自身固定需要 128 = 16B 的存储空间。可见如果每个段管理的磁盘空间足够大，那么使用段式磁盘空间管理可以取得极高的收益；反之，如果每个段固定只用于管理一个磁盘基本块，那么所耗费的存储空间将是位图的 128 倍。因此，段式磁盘空间管理相较于位图而言比较灵活，适用于上层应用对于磁盘空间申请范围比较宽泛的场景。

无论使用何种方式管理磁盘空间，当管理的磁盘空间足够大时，都需要考虑其索引效率，而索引效率一般和其对应的内存组织形式有关。例如针对位图，假定某个位图当中包含 n 个比特，那么直接将这 n 个比特进行顺序排列或者树状排列（常用的树状排列形式为 B-tree、AVL tree 等二叉树形式），针对单个比特的查找操作时间复杂度分别为 $O(n)$ 和 $O(\log_2 n)$（指二叉树），当 n 足够大时，两种排列形式的索引效率可以相差几个数量级。此外，因为上层应用对于磁盘空间需求形式各异，采用不同的空间分配策略所取得的效果也会大相径庭。例如针对段式管理，常见的空间分配策略有 3 种：

- 首次拟合法（First Fit）。首次拟合法总是查找所有段中第一个满足所需求空间大小的段进行分配后返回。
- 最佳拟合法（Best Fit）。最佳拟合法总是查找所有段中某个空间与所需求空间大小最接近的段进行分配后返回。
- 最差拟合法（Worst Fit）。最差拟合法总是查找所有段中空间最大的段，从中分配所需求的空间后返回。

上述 3 种空间分配策略各有优劣。一般而言，最佳拟合法适合于请求空间范围较为广泛的系统。因为按照最佳拟合法进行空间分配时，总是查找和分配管理空间与请求空间大小最为匹配的段，从而使得整个系统中所有段管理的空间处于相差甚远的状态。相反，最差拟合法因为每次都从管理空间最大的段开始分配，从而使得整个系统中所有段管理的空间趋于一致，所以适合于请求空间范围较窄的系统。而首次拟合法的分配方式

是随机的，因此它的适用场景也介于两者之间，常见于事先无法对请求空间范围进行预测的通用系统。从分配效率来看，最佳拟合法和最差拟合法一般都需要针对所有的段按其管理的空间大小进行排序，而首次拟合法一般按照每个段的起始地址自然排序，考虑随着时间推移，空间会逐步趋于碎片化，为了应对后续潜在的大块连续空间分配请求，也为了提升索引效率（索引效率和段的绝对数量强相关，减少段的数量可以提升索引效率），需要将已归还的、物理上连续的段再次合并为一个独立的大段，首次拟合法在处理上述段合并过程中具有天然优势，因此其综合效率最高。

综上，一般需要综合考虑请求空间大小的分布规律、效率对于系统的重要性等因素来制定合适的空间分配策略，例如一种常见的做法是在系统可用空间比较充裕时采用首次拟合法，以提升分配效率；当系统空间碎片化程度较高时，再切换到最佳拟合法，以减少空间碎片。

作为一种通用存储系统的默认存储后端，原理上 BlueStore 应该采用段式磁盘空间管理，但是因为 Ceph 天然面向分布式设计的特性（这使得每个节点上内存和磁盘容量可以控制在一个较低的水平，并且这种分而治之的思想使得每个节点的磁盘空间管理相对独立），加上 BlueStore 设计之初就被定位为面向 SSD 等拥有比传统机械磁盘更大基本块、更小标称容量的高速固态存储设备，所以 BlueStore 空间管理默认采用位图（实际上也支持段，可以根据需要切换）。前面已经提及，磁盘所有基本块任意时刻只有“空闲”和“占用”两种状态，因此，磁盘空间管理也可以分为追踪所有空闲空间列表和追踪所有已分配空间列表两个部分。显然，这两个部分是强相关的，例如从空闲空间列表中新分配空间需要同时将其加入已分配空间列表，表明其已被占用；反过来从已分配空间列表中释放空间也需要同步将其加入空闲空间列表，供再次分配，这说明任意时刻我们都可以由一张列表推导出另一张列表。因此，BlueStore 进行空间管理时，并不需要将两张列表全部存盘。考虑到每个 Onode 已经详细记录了存放数据时所对应的磁盘空间，并且我们一般在释放空间时进行合并操作（指将物理上连续的多个段合并为一个独立的大段，这样最终空闲空间列表中的条目相对较少），所以 BlueStore 选择将空闲空间列表存盘。系统上电时，通过加载空闲空间列表，最终可以在内存中还原出完整的已分配空间列表。

BlueStore 中，FreelistManager 组件负责管理空闲空间列表，Allocator 组件负责管理已分配空间列表，两种组件又各有段和位图两种实现形式。因为 BlueStore 默认使用位图形式，所以这里仅介绍两种组件的位图实现，分别对应 BitmapFreelistManager 和 BitmapAllocator。

2.4.2 BitmapFreelistManager

BitmapFreelistManager 以块为粒度，将连续、数量固定的多个块进一步组成一个段，从而将整个磁盘空间划分为若干连续的段进行管理。每个段以其在磁盘中的对应起始地址进行编号，可以得到一个 BlueStore 实例内唯一的索引，从而可以使用 kvDB 固化 BitmapFreelistManager 中的所有段信息。如前所述，单个块的状态可以使用一个比特进行标记，因此每个段的值部分是一个长度固定的比特流，比特流中的某个比特置位，表明对应的块已经被分配，反之则表明对应的块空闲。此外，因为使用 kvDB 存储段信息，所以需要合理调整 BitmapFreelistManager 中的段大小设置，过大（此时值长度变大）或者过小（此时键值对数量增加）的段设置都不利于充分发挥 kvDB 的性能。

系统运行过程中，BitmapFreelistManager 中的块需要频繁地在“空闲”和“占用”两种状态之间切换，由表 2-19 所示的布尔运算实现。

表 2-19 基于布尔运算实现单个块状态快速翻转

原有状态	掩码	XOR
0	1	0^1=1
1	1	1^1=0

可见，固定使用“1”作为掩码，基于异或运算可以实现每个块在“空闲”（对应“0”）和“占用”（对应“1”）两种状态之间快速切换，因此，BitmapFreelistManager 分配或者释放一段空间的运算逻辑是相同的，这是 BitmapFreelistManager 设计的理论依据。

BitmapFreelistManager 定义的公共接口如表 2-20 所示。

表 2-20 BitmapFreelistManager 公共接口

接口名称	含义
create()	通过 BlueStore 的 mkfs() 接口创建一个 BitmapFreelistManager。 因为 BitmapFreelistManager 中的一些关键参数例如块大小、每个段包含的块数目等等可配置，所以需要通过 create() 固化到 kvDB 中。后续重新上电时，这些参数将直接从 kvDB 读取，防止因为配置变化而导致 BitmapFreelistManager 无法正常工作
init()	初始化 BitmapFreelistManager。上电时调用，用于从 kvDB 中加载块大小、每个段包含的块数目等可配置参数
allocate()	从 BitmapFreelistManager 中分配指定范围（[offset, offset + length]）空间
release()	从 BitmapFreelistManager 中释放指定范围（[offset, offset + length]）空间
enumerate_reset()	上电时，BlueStore 通过这两个接口遍历 BitmapFreelistManager 中所有空闲段，并将其从 Allocator 中同步移除，从而还原得到上一次下电时 Allocator 对应的内存结构
enumerate_next()	

2.4.3 BitmapAllocator

BitmapAllocator 实现了一个块粒度的内存版本磁盘空间分配器。和 BitmapFreelistManager 不同，因为 BitmapAllocator 中的所有段信息不需要使用 kvDB 存盘，所以可以采用非扁平方式进行组织，以提升索引效率。实现上，BitmapAllocator 中的块是以树状形式组织的，如图 2-7 所示。

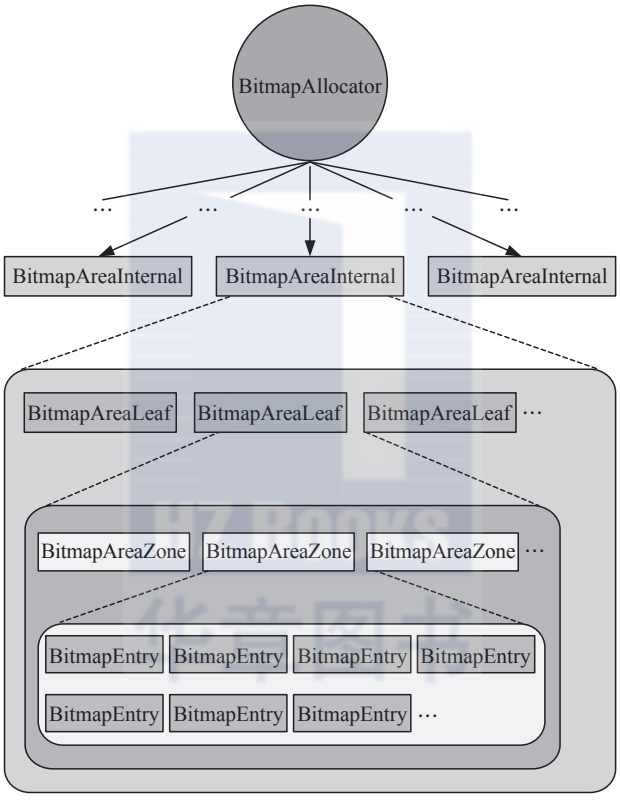

图 2-7 BitmapAllocator 层次结构

BitmapEntry 是整个 BitmapAllocator 中的最小可操作单位，例如可以定义为 64 位无符号整数，这样单个 BitmapEntry 可以同时记录 64 个物理上相邻块的状态。一定数量的 BitmapEntry 可以进一步组成一个 BitmapZone。BitmapZone 是 BitmapAllocator 中单次最大可分配单位，其大小可以配置。BitmapZone 拥有独立的锁逻辑，所有 API 都被设计成原子的，因此不同 BitmapZone 之间的操作可以并发。如果 BitmapAllocator 管理的磁盘空间过大，为了提升索引效率，可以进一步引入一些中间逻辑结构，将所有 BitmapZone 再次

进行树状组织。这个中间树状结构称为 BitmapArea，其中叶子节点称为 BitmapAreaLeaf，一个 BitmapAreaLeaf 叶子节点包含固定数量的 BitmapZone；多个 BitmapAreaLeaf 叶子节点可以组成一个 BitmapAreaInternal 中间节点；多个 BitmapAreaInternal 中间节点可以再次作为更上一级 BitmapAreaInternal 中间节点的孩子节点，直至最终到达根节点——BitmapAllocator。整个 BitmapAllocator 的拓扑结构都是可配置的，主要受两个参数约束：一是 BitmapZone 大小，这个参数不但决定了 BitmapAllocator 单次可分配的最大连续空间，也决定了并发操作的粒度；二是 BitmapArea 中单个（中间或叶子）节点的跨度（span-size），这个参数决定了 BitmapArea 的高度，进而决定了整个 BitmapAllocator 的索引效率。

为了减少空间碎片化，同时提升索引效率，BitmapAllocator 也允许自定义最小可分配空间。顾名思义，最小可分配空间是 BitmapAllocator 分配空间的最小粒度，所有通过 BitmapAllocator 分配的空间必须是最小可分配空间的整数倍，相应的，这要求最小可分配空间必须配置为基本块大小的整数倍。实际实现时，BitmapAllocator 总是以最小可分配空间为单位，直至分配到上层所需求的空间为止（例如最小可分配空间为 16 个基本块大小，则每找到 16 个连续的空闲比特视为完成一次分配；如果需求空间为 256 个基本块，则一共需要进行 16 次这样的分配），物理上相邻的空间会在以段形式返回给上层应用时自动进行合并。可见，假定最小可分配空间与块大小相同（这是默认情况！），BitmapAllocator 进行空间分配时实际上是在逐位扫描，这种方式实际上只在磁盘空间碎片化程度很高时被证明是行之有效的；反之，如果整个磁盘的空间碎片化程度很低同时有大量空间需求比较大的分配请求，那么采用这种方式效率实际上是非常低的。一个改进的方向是使用“大嘴法”，即每次分配空间时，不是找到最小可分配空间即完成一次分配，而是找到尽可能多、同时为最小可分配空间整数倍的连续空间才算完成一次分配。

BitmapAllocator 定义的公共接口如表 2-21 所示。

表 2-21 BitmapAllocator 公共接口

接口名称	含义
init_add_free()	参见表 2-20，BlueStore 上电时，通过 FreelistManager 读取磁盘中空闲的段，然后调用本接口将 BitmapAllocator 中相应的段空间标记为空闲
init_rm_free()	将 BitmapAllocator 指定范围的空间（[offset, offset + length]）标记为已分配
reserve()	预留空间 / 释放预留空间。 因为 BitmapAllocator 支持多线程访问，所以通过 BitmapAllocator 进行空间分配时，需要先调用 reserve() 接口进行空间预留，以保证后续通过 allocate() 接口能够分配到所请求的空间。 如果通过 allocate() 分配的空间比之前 reserve() 少，那么差值需要通过 unreserved() 返还
unreserve()	

（续）

接口名称	含义
allocate()	分配 / 释放空间。 分配的空间不一定是连续的，有可能是一些离散的段。 allocate() 接口可以同时指定 hint 参数，用于对下次开始分配的起始地址（例如可以是上一次成功分配后所返回空间的结束地址）进行预测，以提升分配速率
release()	
get_free()	返回当前 BitmapAllocator 实例中的所有空闲空间大小

需要注意的是，因为 BlueStore 将不同类型的数据严格分开并且允许使用不同的设备存储，所以一个 BlueStore 实例中可能存在多个 BitmapAllocator 实例。

2.5 BlueFS

2.5.1 RocksDB 与 BlueFS

诞生于 2011 年的 LevelDB 是基于 Google 的 BigTable 数据库系统发展而来的，是键值对类型的日志型非关系数据库，专为存储海量（例如千万级别）键值对设计。理论上，LevelDB 中的键或者值只受存储容量的限制，可以为任意长度的字节流，所有键值对严格按照键排序。LevelDB 继承并发展了 BitTable 中 LSM-Tree[⊖] + SSTable（Sorted String Table，有序字符串表，键值对的磁盘存储格式。BigTable 针对键值对的修改操作采用了写时重定向的策略，即从不对已有记录进行覆盖写，以避免 RMW，从而提升性能，因此 SSTable 的内容是只读的）的概念，将 SSTable 在磁盘上进行分级存储，进一步提升索引性能，这也是 LevelDB 的由来。

然而随着 SSD 的逐渐普及，LevelDB 使用单线程进行 SSTable 压缩以及利用 mmap 将 SSTable 读入内存等做法已经无法充分发挥 SSD 的性能[⊖]。基于 LevelDB 诞生了 RocksDB，后者致力于为新型高性能固态存储介质例如 SSD、NVRAM 等提供更加卓越的键值对类型数据库访问性能。表 2-22 展示了 LevelDB 和 RocksDB 的性能对比。

RocksDB 具有如下特性：

- 专为使用本地闪存设备作为存储后端、且数据库容量不超过几个 TB 的应用程序设计，是一种内嵌式的非分布式数据库。

⊖ http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.44.2782&rep=rep1&type=pdf

⊖ http://rocksdb.blogspot.com/

表 2-22 LevelDB 和 RocksDB 性能对比[①]

测试结论表明：LevelDB 在空间利用率上更有优势，而 RocksDB 在绝大多数场景下性能更有优势

Test step	LevelDB	RocksDB
Write 100M values	36m8.29s	21m18.60s
DB Size	2.7GB	3.2GB
Query 100M values	2m55.37s	2m44.99s
Delete 50M values	3m47.64s	1m53.84s
Compaction	3m59.87s	3m20.27s
DB Size	1.4GB	1.6GB
Query 50M values	12.12s	13.59s
Write 50M values	3m5.28s	1m26.90s
DB Size	673MB	993MB

① https://www.influxdata.com/benchmarking-leveldb-vs-rocksdb-vs-hyperleveldb-vs-lmdb-performance-for-influxdb/

- 适合于存储小型或者中性键值对；性能随键值对长度上升下降很快。
- 性能随 CPU 核数以及后端存储设备的 I/O 能力呈线性扩展。

除此之外，RocksDB 还拥有诸多 LevelDB 所不具备的特性，典型如对于列簇（Column Families）、备份和还原点、Merge 操作符的支持等。RocksDB 采用 C++ 进行开发，在设计上非常灵活，几乎所有组件都可以根据需要进行替换，这其中就包括用于固化 SSTable 和 WAL（Write Ahead Log，指日志）的本地文件系统。因为 RocksDB 设计理念和 BlueStore 高度一致，所以 BlueStore 默认采用 RocksDB 取代 LevelDB 作为元数据存储引擎。同时，因为多数操作系统默认的本地文件系统（典型如 XFS、ext4、ZFS 等）对于 RocksDB 而言很多功能不是必须的，所以为了进一步提升 RocksDB 的性能，需要对本地文件系统的功能进行裁剪。当然更彻底的解决办法是为 RocksDB 量身定制一款本地文件系统，在此背景下，BlueFS 应运而生。

BlueFS 是个简易的用户态日志型文件系统，它恰到好处地实现了 RocksDB::Env 所定义的全部接口，后者用于固化 RocksDB 运行过程中产生的 .sst（对应 SSTable）和 .log（对应 WAL）文件。基于同样的道理（参考本章第 1 节，引入日志的目的一般都是为了进行写加速），WAL 对于提升 RocksDB 的性能至关重要，所以 BlueFS 在设计上支持将 .sst 和 .log 文件分开存储，以方便将 .log 文件单独使用速度更快的固态存储设备例如 NVMe SSD 或者 NVRAM 存储。

这样，引入 BlueFS 后，BlueStore 将所有存储空间从逻辑上分成了三个层次：

（1）慢速（Slow）空间

这类空间主要用于存储对象数据，可由普通大容量机械磁盘提供，由 BlueStore 自行管理。

（2）高速（DB）空间

这类空间主要用于存储 BlueStore 内部产生的元数据（例如 onode），可由普通 SSD 提供，容量需求比（1）小。因为 BlueStore 的元数据都交由 RocksDB 管理，而 RocksDB 最终通过 BlueFS 将数据存盘，所以这类空间由 BlueFS 直接管理。

（3）超高速（WAL）空间

这类空间主要用于存储 RocksDB 内部产生的 .log 文件，可由 NVMe SSD 或 NVRAM 等时延相较普通 SSD 更小的设备充当，容量需求和（2）相当（实际上还取决于 RocksDB 相关参数设置）。超高速空间也由 BlueFS 直接管理。

需要注意的是，上述高速、超高速空间需求不是固定的，和慢速空间的使用情况紧密相关（BlueStore 元数据数量和对象数量、对象中数据稀疏程度、被覆盖写的次数等相关；而 RocksDB 中 WAL 的数量则和记录（即 BlueStore 中的元数据）数量、访问习惯、自身的压缩策略等都相关），如果高速、超高速空间规划的较为保守，BlueFS 也允许使用慢速空间进行数据转存。因此，在设计上，BlueStore 将自身管理的一部分慢速空间拿出来和 BlueFS 进行共享，并在运行过程中进行实时监控和动态调整，具体策略为：BlueStore 通过自身周期性被唤醒的同步线程实时查询 BlueFS 的可用空间，如果 BlueFS 的可用空间在整个 BlueStore 可用空间中的占比过小，则新分配一定量的空间至 BlueFS（如果 BlueFS 所有空间绝对数量不足设定的最小值，则一次性将其管理的空间总量追加至最小值）；反之如果 BlueFS 的可用空间在整个 BlueStore 可用空间中占比过大，则从 BlueFS 中回收一部分空间至自身。

综上，如果 3 类空间分别使用不同的设备管理，那么一般情况下 BlueFS 中的可用空间一共有 3 种。对于 .log 文件以及自身产生的日志（BlueFS 本身也是一种日志型的本地文件系统），BlueFS 总是优先选择使用 WAL 类型的设备空间，如果不存在或者 WAL 设备空间不足，则选择 DB 类型的设备空间，如果仍然不存在或者 DB 设备空间也不足，则选择 Slow 类型的设备空间。对于 .sst 文件，则优先使用 DB 类型的设备空间，如果不存在或者 DB 设备空间不足，则选择 Slow 类型的设备空间。Slow 类型的设备因为由 BlueStore 直接管理，所以与 BlueFS 共享的空间也是由 BlueStore 直接管理，后者会将所

有已经成功分配给 BlueFS 的空间段单独使用一个名字叫作 bluefs_extents 的结构进行管理，并从自身的 Allocator 中扣除。bluefs_extents 是一个集合，每个元素对应 Slow 设备中的一个空间段，每次更新后会作为 BlueStore 的元数据存盘。这样，后续 BlueFS 上电时，通过 BlueStore 预先加载的 bluefs_extents，即可正确初始化这部分共享空间对应的 Allocator。至于 DB 设备以及 WAL 设备，因为单独由 BlueFS 管理并且对 BlueStore 不可见，所以在上电时由 BlueFS 自身负责初始化，亦即通常情况下 BlueFS 在上电时会初始化 3 个 Allocator 实例，分别管理 3 种类型的可用空间。

在 2.4 节中我们曾经提及——对于磁盘空间的管理，一般固化空闲空间列表和已分配空间列表中的任意一种即可。BlueStore 选择固化空闲空间列表，同时所有已分配空间信息也保存在每个 Onode 之中，因此这两类信息实际上存在重复。但是由于一个 BlueStore 实例管理的 Onode 数目原理上只受存储容量的限制，实际场景中 Onode 数目可能十分巨大，因此为了加速上电过程，BlueStore 需要额外固化一张空闲空间列表。BlueFS 则不同，一方面 BlueFS 存储的数据十分有限，其规模为 BlueStore 的千分之一到百分之一之间（常见的存储系统中元数据和数据比重一般都在这个范围之内）；另一方面 RocksDB 生成的 .sst 文件大小固定，并且从不进行修改，所以 BlueFS 中绝大部分磁盘空间需求都是比较统一和固定的。因此，基于上述两个因素，BlueFS 既不保存空闲空间列表，也不保存已分配空间列表，而是通过上电时遍历所有文件的元数据信息，据此生成完整的已用空间列表，即 Allocator。

2.5.2 磁盘数据结构

在上一节中，我们介绍了 BlueFS 相关概念，本节介绍 BlueFS 的磁盘数据结构。和其他通用的文件系统类似，BlueFS 也有目录和文件的概念，而且因为是日志型文件系统，所以 BlueFS 的磁盘数据主要包括文件、目录和日志三种类型。

传统文件系统普遍采用层级（树状）结构对目录和文件进行组织，这种组织方式在面向存储海量文件的设计中，因为具有较高的单点查找效率（以二叉树为例，单个查找操作的时间复杂度为 $O(\log_2 n)$），并且可以以目录为单位对文件进行区域隔离，所以被实践证明是行之有效的。然而凡事皆有两面性，采用层级结构的文件系统的磁盘数据格式一般而言都比较复杂，因此如果存储的文件数量没有达到一定规模，其效率反而会比直接采用扁平结构更低。如前所述，BlueFS 因为只用于存储单个 BlueStore（对应一块磁盘）的元数据，所存储的文件规格比较统一（绝大多数为 SSTable）并且数量十分有限，所以

可以直接采用扁平结构进行组织。实际实现时，BlueFS 使用两类表来追踪所有管理的文件及其目录层级关系，如图 2-8 所示。

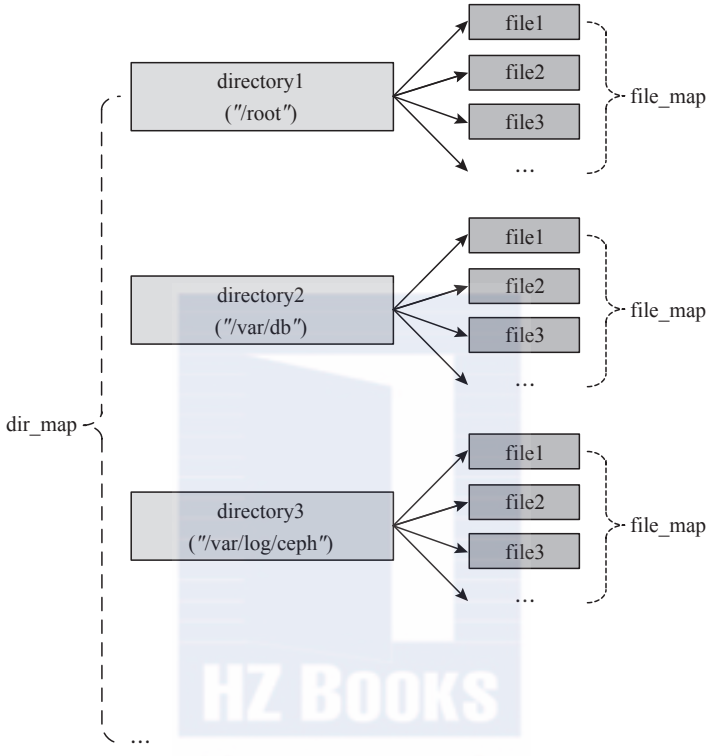

图 2-8 BlueFS 的文件组织形式

由图 2-8 可见，BlueFS 定位某个具体文件一共需要经过两次查找：第一次通过 dir_map 找到文件所在的最底层文件夹；第二次通过该文件夹下的 file_map 找到对应的文件。需要注意的是：因为是扁平组织，所以 dir_map 中每个条目描述的都是文件的绝对路径，即条目之间没有隶属关系。每个文件也采用一个类似 inode 的结构进行管理，BlueFS 称为 bluefs_fnode_t（简称 fnode，下同）。因此，file_map 建立的实际上是文件名和 fnode 之间的映射关系，fnode 相应的磁盘数据结构如表 2-23 所示。

表 2-23 bluefs_fnode_t

成员	含义
ino	唯一标识一个 fnode
size	文件大小
mtime	文件上一次被修改的时间

（续）

成员	含义
prefer_bdev	存储该文件优先使用的块设备，例如 .log 文件或者 BlueFS 自身的日志文件将优先使用 WAL 设备
extents	磁盘上的物理段集合

原则上，表 2-23 中的 extents 中的单个 extent 可以复用 bluestore_pextent_t（参考表 2-5），但是因为每个文件都可能使用来自多个不同块设备（WAL、DB 和 Slow）的空间，所以 BlueFS 的 extent 还需要额外记录其归属的块设备标识，具体见表 2-24。

表 2-24 bluefs_extent_t

成员	含义
bdev	归属块设备标识
offset	归属块设备上的物理地址
length	空间长度

和其他日志型文件系统类似，BlueFS 所有修改操作也是基于日志的。每个修改操作都会生成一个独立的日志事务，然后再通过 flush_log() 调用进行批量提交。以 mkdir() 操作为例，BlueFS 只是简单生成一条日志记录，即可向上层应用返回操作成功：

```
op_code: OP_DIR_CREATE
op_data: dir(string)
```

为了减少每次需要刷盘的日志量以提升效率，BlueFS 采用增量日志模式。因此，随着时间推移，BlueFS 中日志数量会逐渐膨胀，这一方面会造成不必要的空间浪费，尤其日志本身使用的是容量较小的高速 WAL 设备空间；另一方面会导致 BlueFS 在上电时需要进行大量的日志重放，才能得到完整的 dir_map 和相应的 file_map，上电时间变长，所以 BlueFS 需要定期对日志进行清理。

这个清理的过程称为日志压缩，其主要逻辑是将当前最新的 dir_map 和所有 file_map 加入到一个独立的日志事务中并存盘，这样，下次上电时，BlueFS 通过重放这个日志事务，即可还原出完整的 dir_map 和其对应的 file_map。因此，这个日志事务可以重新作为后续增量日志事务的基准，为每个日志事务分配一个独一无二的序列号之后，所有序列号小于此基准序列号的日志事务都可以删除，从而释放日志空间。早期的实现中，日志压缩过程是完全同步的。日志压缩需要独占式地访问 dir_map 和 file_map 从而造成写停顿（write stalls），所以当前实现了日志压缩的一个改进版本。改进后的日志压缩流程除了生成内存基准日志事务的过程严格同步之外（通过持有 BlueFS 全局的排他锁实现），其他过程都是异步的，因此可以大大改善由于日志压缩所引起的写停顿。综上，我们可以定义日志事务的磁盘数据结构如表 2-25 所示。

表 2-25 bluefs_transaction_t

成员	含义
uuid	日志事务归属 bluefs 对应的 uuid
seq	全局唯一序列号
op_bl	编码后的日志事务条目，可以包含多条。每个条目由操作码和操作涉及的相关数据组成

因为上电时，我们总是通过日志重放来得到 BlueFS 所有元数据，所以还需要一个固定入口，用于索引日志（日志本身采用一个单独的文件进行保存）所对应的存储位置。这个入口称为超级块（SuperBlock），BlueFS 总是将其写入由自身接管的 DB 设备的第二个 4K 存储空间，其结构如表 2-26 所示。

表 2-26 bluefs_super_t

成员	含义
uuid	bluefs 关联的 uuid
osd_uuid	bluefs 关联的 OSD 对应的 uuid
version	超级块当前版本
block_size	DB/WAL 设备块大小，固定为 4K
log_fnode	日志文件对应的 fnode

BlueFS 提供的 API 与传统文件系统并无二致，这里不再赘述。最后，我们给出引入 RocksDB 和 BlueFS 之后的 BlueStore 逻辑架构图如图 2-9 所示。

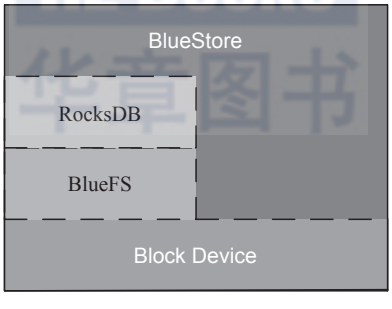

图 2-9 BlueStore 逻辑架构（使用 RocksDB + BlueFS）

2.5.3 块设备

在上一节中，我们知道引入 RocksDB + BlueFS 后，BlueStore 至多可以管理三种类型的存储空间——Slow、DB 和 WAL，这三类空间的容量和性能需求不尽相同，因此实际部署时可以分别使用不同类型的块设备（Block Device）提供。

在 Linux 的设计中，一切皆文件，因此块设备也被内核当作文件管理。对于块设备的访问最终通过相应的驱动程序实现，并且只能以块（从 512B 到 4KB 不等）为粒度进行，这也是块设备名称的由来。不同块设备其传输速率除了取决于制造工艺外，还受限于所使用的总线（传输）标准，例如采用传统 SATA 3.0 总线标准的块设备的理论传输速率上限为 6Gbps，而如果采用 PCIe 3.0X4 总线标准则其理论传输速率上限高达 32Gbps。SATA 总线标准及其对应的 AHCI 接口其实是为高延时的机械磁盘设计的，但是目前依然为主流的 SSD 所采用。随着 SSD 的性能逐渐增强（摩尔定律指出：当价格不变时，集成电路上可容纳的元器件的数目，约每隔 18 ～ 24 个月便会增加一倍，性能也将提升一倍），这些标准已经成为限制 SSD 发展的一大瓶颈，专为机械硬盘而设计的 AHCI 标准并不太适合低延时的 SSD，于是 NVMe 应运而生。NVMe（Non-Volatile Memory express）与 AHCI 类似，都是一种逻辑设备接口标准。与传统的 SATA SSD 相比，基于 PCIe 的 NVMe SSD 能够提供数十倍或更高的 IOPS，同时平均时延下降至几十分之一或更低。

SPDK（Storage Performance Development Kit）是 Intel 专为高性能、可扩展、用户态的存储类应用程序开发的工具套。SPDK 取得高性能的关键在于将所有必需的驱动程序移植到用户态，同时采用主动轮询的模式来替代中断，从而避免内核上下文切换以及中断处理带来的额外开销。SPDK 自带 NVMe 驱动，目前 BlueStore 基于 SPDK 实现了对于 NVMe SSD 这类新型块设备的支持，从而在设计上支持使用 NVMe SSD 充当 DB 及 WAL 设备（目前社区正在计划增加对于 NVRAM 的支持，后续可以将 WAL 保存在性能更好的 NVRAM 设备上）进行性能增强，进而使得在未来，基于 BlueStore 构建全闪存的高性能 Ceph 存储集群成为可能。

2.6 实现原理

在前几节中，我们已经介绍了 BlueStore 一些基本设计理念及相关支撑组件，本节我们通过几个主要的流程介绍 BlueStore 的具体实现，分别是 mkfs、mount、read 和 write。

2.6.1 mkfs

mkfs 主要固化一些用户指定的配置项到磁盘，这样后续 BlueStore 上电时，这些配置项将直接从磁盘读取，从而不受配置文件改变的影响（这也说明每个 BlueStore 实例的配置项可以是不同的）。之所以需要固化这些配置项，是因为 BlueStore 使用不同的配置

项对于磁盘数据的组织形式不同（BlueStore 的每一种组件，例如 FreelistManager，都支持多种不同实现方式），如果前后两次上电使用不同的配置项访问磁盘数据有可能导致数据发生永久性损坏。一些需要在 mkfs 过程中写入磁盘的典型配置项及其含义如表 2-27 所示。

表 2-27 需要通过 mkfs 固化的配置项

元数据类型	作用
os_type	ObjectStore 类型，目前有 FileStore 和 BlueStore 两种，这两种 OS 对于磁盘数据的管理形式完全不同
fsid	唯一标识一个 BlueStore 实例
freelist_type	标识 FreelistManager 的类型。 如前所述，因为 BlueStore 固化所有空闲空间列表至 kvDB，所以 FreelistManager 不能动态变化，否则上电时无法正常从 kvDB 读取所有空闲空间信息。 基于 freelist_type 可以创建相应类型的 FreelistManager，然后由 FreelistManager 从 kvDB 读取所有空闲空间信息，进而在内存中重建得到完整的空闲空间列表。 基于 FreelistManager 的空闲空间信息总是可以得到 Allocator，这意味着 Allocator 的具体类型在上电时是可以动态改变的
kv_backend	使用何种类型的 kvDB，目前有 LevelDB 和 RocksDB 可选
bluefs	如果使用 RocksDB 作为默认的 kv_backend，是否使用 BlueFS 替换 RocksDB 默认的本地文件系统接口

2.6.2 mount

OSD 进程上电时，BlueStore 通过 mount 操作完成正常上电前的检查和准备工作，其处理逻辑如图 2-10 所示。

由图 2-10 可见，mount 操作主要包含以下几个步骤：

（1）校验 ObjectStore 类型

因为 ObjectStore 有多种实现，不同实现对于磁盘的管理方式不同，所以需要在 mkfs 时固化 ObjectStore 类型至磁盘，并在 mount 进行类型校验，防止将磁盘数据写坏。

（2）fsck 或者 deep-fsck

fsck 扫描并校验 BlueStore 实例当中的所有元数据。如果打开 deep 选项，会进一步深度扫描并校验所有对象数据。如果 BlueStore 中对象数量比较多，那么在 mount 操作中执行 fsck 或者 deep-fsck 会大大延长 OSD 上电时间，因此也可以通过 Ceph 提供的工具选择在业务比较空闲、或者执行例行维护时手动进行。目前 mount 过程中的 fsck 选项默

认是关闭的。

图 2-10 mount

（3）加载并锁定 fsid

fsid 唯一标识一个 BlueStore 实例。锁定 fsid 的目的是为了防止对应的磁盘被多个 OSD 进程对应的 BlueStore 实例同时打开和访问，从而引起数据一致性问题。

（4）加载主块设备

如前所述，主块设备用作存储对象数据，一般可由大容量的慢速机械磁盘充当，由 BlueStore 直接管理。

（5）加载数据库，读取元数据

表 2-28 中列举了 mount 过程中需要从数据库读取的主要元数据。

表 2-28 mount 过程中需要加载的元数据

元数据类型	作用
nid_max	每个对象拥有 BlueStore 实例内唯一的 nid。nid_max 用于标识当前 BlueStore 最小未分配的 nid，新建对象的 nid 总是从当前的 nid_max 开始分配
blobid_max	类似 nid，整个 BlueStore 实例内唯一。引入 blobid 的目的主要是实现 blob 在对象之间的共享，而共享信息（为 bluestore_shared_blob_t，参考“2.2.2 对象”）需要独立于对象存储，因此需要一个全局唯一的标识
freelist_type	标识 FreelistManager 的类型，参考表 2-27
min_min_alloc_size	为了提升空间管理效率同时降低空间碎片化程度，BlueStore 也允许自行配置最小可分配空间（Minimal Allocable Size，简称 MAS）。 BlueStore 总是记录历史 MAS 中的最小值，并据此创建 Allocator（举例来说，假定我们指定磁盘的基本块为扇区，那么我们后续将基本块大小调整为扇区的整数倍也是能正常工作的，反之则不行）
bluefs_extents	从主设备分配给 BlueFS，供 BlueFS 使用的额外空间

（6）加载 Collection

如前所述，因为 Collection 数量有限，所以可以常驻内存。

完成上述步骤后，mount 随后会创建一些工作线程，比如用于进行数据同步的同步线程，用于执行回调函数的 finisher 线程，用于统计内存使用的监控线程等，如果上次下电不是优雅下电，那么还可能需要执行日志重放进行数据恢复。最后，在设置了一些诸如校验算法、压缩算法之类的全局参数之后，mount 操作全部完成，上层应用可以正常读写 BlueStore 当中的数据。

2.6.3 read

read 接口用于读取对象指定范围内的数据，目前 BlueStore 实现的 read 接口是同步的，其处理逻辑如图 2-11 所示。

参见图 2-11，read 逻辑比较简单，主要涉及查找 Collection、查找 Onode、读缓存和读磁盘 4 个步骤：

（1）查找 Collection

如前，BlueStore 上电时已经通过 mount 操作预先将所有 Collection 加载至内存，而且因为单个 BlueStore 管理的 PG 数量有限，所以这个查找过程耗时相对后续操作几乎可以忽略。成功查找到 Collection 之后，如果对应的 Collection 存在，将以阻塞的形式获取 Collection 内部读写锁中的读锁，亦即针对同一个 Collection，所有读操作可以并发。

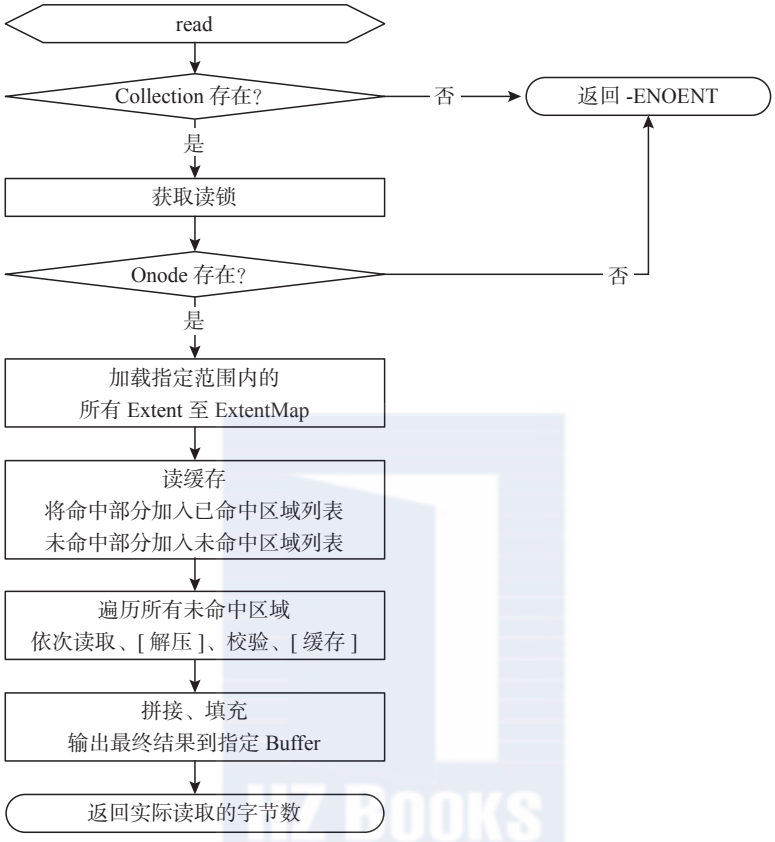

图 2-11 read

（2）查找 Onode

BlueStore 中 Onode 使用 LRU 队列进行管理，因此如果对应的 Onode 没有在缓存中命中，那么需要基于其磁盘格式在内存中进行重建。2.2.2 节中提到，每个 Onode 包含一张 ExtentMap，ExtentMap 包含若干个 Extent，每个 Extent 负责管理一段逻辑范围内的数据并关联一个 Blob，并最终由 Blob 通过若干个 pextent 负责将这些数据映射至磁盘。这几种管理结构的映射关系如图 2-12 所示。

每个 Blob 包含一个 SharedBlob。顾名思义，SharedBlob 是用于实现 Blob、也就是 Extent 之间的数据共享，其主要内容为一张基于 pextent 的引用计数表，表明对应的内容被引用（即克隆）的次数。除此之外，SharedBlob 还包含 BufferSpace，用于对 Blob 中的数据进行缓存，并最终纳入 Cache 管理。

引入 SharedBlob 之后，每个 Blob 有 shared 和 !shared 状态。如果为 shared 状态，则

表明该 Blob 确实被多个 Onode 共享，此时 SharedBlob 关联了一张有效的引用计数表，出于和 ExtentMap 类似的考虑，这张引用计数表也是按需加载的，只有真正需要被使用时，才会从 kvDB 加载至 SharedBlob，此时 SharedBlob 的状态变为 loaded（对应的，引用计数表未从磁盘加载时其状态为 !loaded），BlueStore 会同步将其加入 Collection 的全局 SharedBlob 缓存，供后续需要再次使用时快速索引。

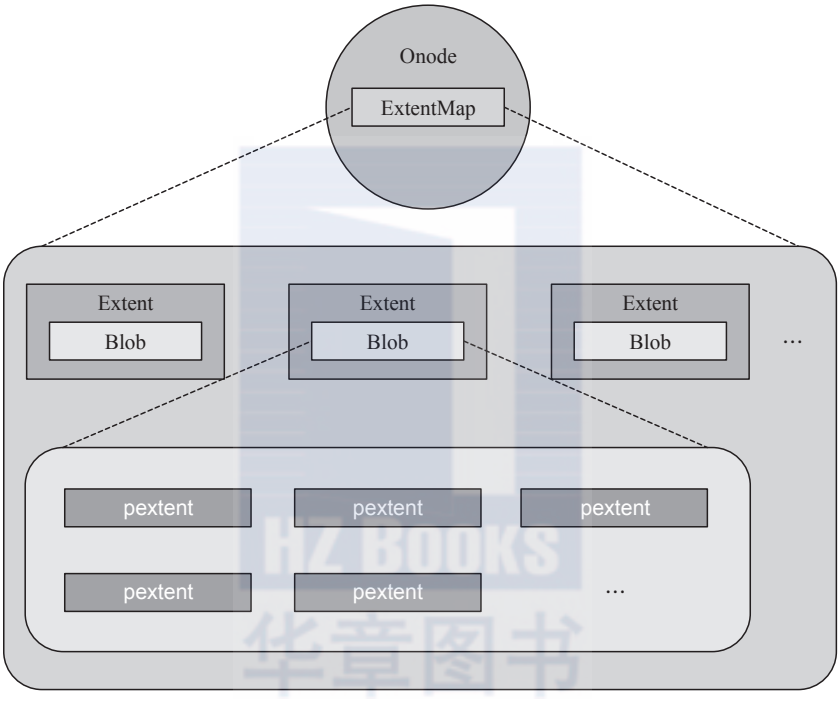

图 2-12 Onode 内部结构

综上，如果 Onode 没有在缓存中命中，为了防止从 kvDB 当中一次性读取大量数据（例如对象的空间使用情况比较复杂，导致 ExtentMap 的 Extent 数量巨大；或者磁盘碎片化比较严重，导致每个 Extent 包含大量的 pextent 等）进行 Onode 重建，造成前端线程长时间的等待，包括 ExtentMap、SharedBlob 在内的管理结构都是动态、根据实际需要进行加载的。

（3）读缓存

如果 Onode 直接在缓存中命中，那么可能有部分数据已经存在于全局 Cache 当中，此时 BlueStore 会尝试先从 Cache 读取指定范围内的数据。读完 Cache 之后会产生已命中

数据区域和未命中数据区域两张列表。

（4）读磁盘

如果全部或者部分数据没有在 Cache 中命中，此时需要去磁盘读取。步骤（3）中我们已经生成了完整的未命中数据区域列表，据此可以加载对应的 Extent 至内存，然后从磁盘对应位置读取数据。根据配置，BlueStore 可能对直接读到的数据执行校验以防止静默数据错误，同时如果数据经过压缩，还需要进行解压之后才能得到原始数据。最后，通过拼凑和填充（全 0）的方式，我们可以得到指定范围内的完整数据，并返回给上层应用。

2.6.4 write

包括 write 在内的所有涉及数据修改的操作，都是通过 queue_transactions 接口以事务组的形式提交至 BlueStore 的。queue_transactions 处理逻辑如图 2-13 所示。

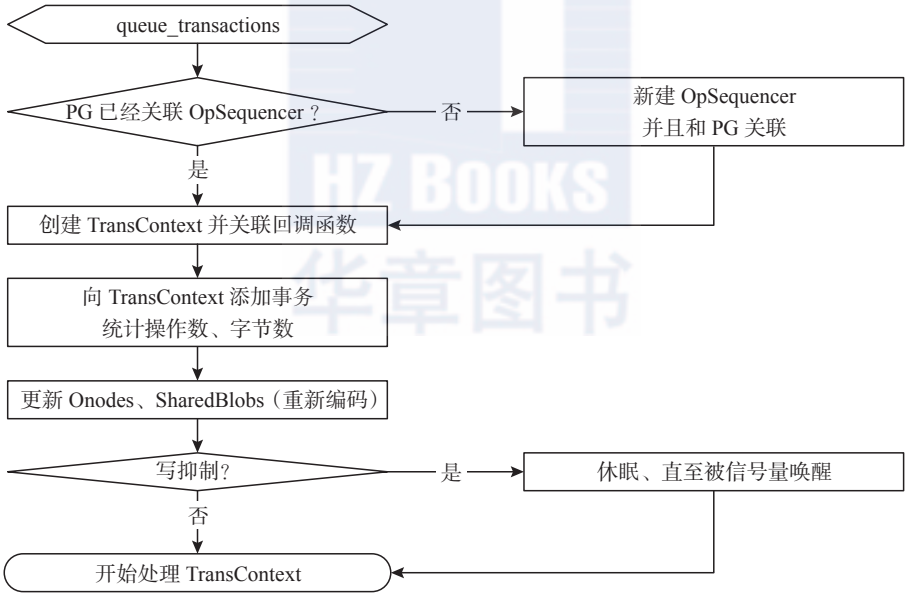

图 2-13　通过 queue_transactions 向 BlueStore 提交事务

上述流程中，OpSequencer 的作用已经在本章第 1 节中介绍，主要用于对同一个 PG 提交的多个事务组进行保序。针对同一个事务组中的多个操作，BlueStore 会创建一个事务上下文——TransContext，将每个修改操作顺序添加至 TransContext，然后进行批量处

理。所有修改操作添加完成后（此时修改操作对应的内存版本完成），通过 TransContext 汇总操作数以及波及的字节数，再提交至 Throttle（顾名思义，这是 BlueStore 内部的一种流控机制），判断是否需要进行写抑制。如果没有过载，即不需要进行写抑制，则开始执行 TransContext。所有通过 queue_transactions 提交的事务组都是异步执行的，因此需要指定若干种类型的回调上下文，当相应的事务组执行到某个特定的阶段后，通过执行对应的回调上下文来唤醒上层应用执行相应操作。常见的回调上下文共有两种：

（1）on_readable

因为 ObjectStore 默认需要使用日志，所以所有修改操作都会先写入速度较快的日志设备。写日志阶段完成后，即可向前端返回写日志完成应答，此时可以确保对应的修改操作涉及的数据不会丢失。

（2）on_commit

也称为 on_disk 或者 on_safe，顾名思义，指对应的数据已经成功写入主要存储设备。

FileStore 的实现中，普遍使用 SSD 充当日志盘，HDD 充当主要存储设备（也称为数据盘，下同），写日志一般而言比写数据速度要快，因此通常情况下上层应用会先后收到 on_readable 和 on_commit 通知。某些特殊情况下，如果 on_commit 先于 on_readable 到达，则上层应用可以跳过 on_readable 的执行。BlueStore 则不同，因为 BlueStore 每个写操作只会产生少量的 WAL 日志，并且 WAL 日志本身也作为元数据的一种，直接和其他元数据一并使用 kvDB 保存，所以 BlueStore 写日志要先于写数据完成，因此在对应的事务组完成后，BlueStore 会先执行 on_commit，然后才执行 on_readable（这实际上是出于兼容性考虑，原则上 BlueStore 不需要 on_readable）。

将 write 操作加入 TransContext 的流程如图 2-14 所示。

如果本次 write 操作范围内没有任何已有数据，则将对应的 write 操作称为新写；反之称为覆盖写。新写的处理逻辑相较覆盖写简单很多，按照数据的逻辑地址范围是否进行了 MAS（最小可分配空间，参见表 2-28）对齐，可以分为头尾非 MAS 对齐写和中间 MAS 对齐写。对应 MAS 对齐部分，其处理逻辑如图 2-15 所示。

上述流程中，对应一次写入数据量比较大的情况，之所以要分成多个新的 Extent 写入，原因之前已经分析过了，主要是防止生成的校验数据过大，影响其在 kvDB 中的索引效率。非 MAS 对齐写和上述流程类似，区别在于：

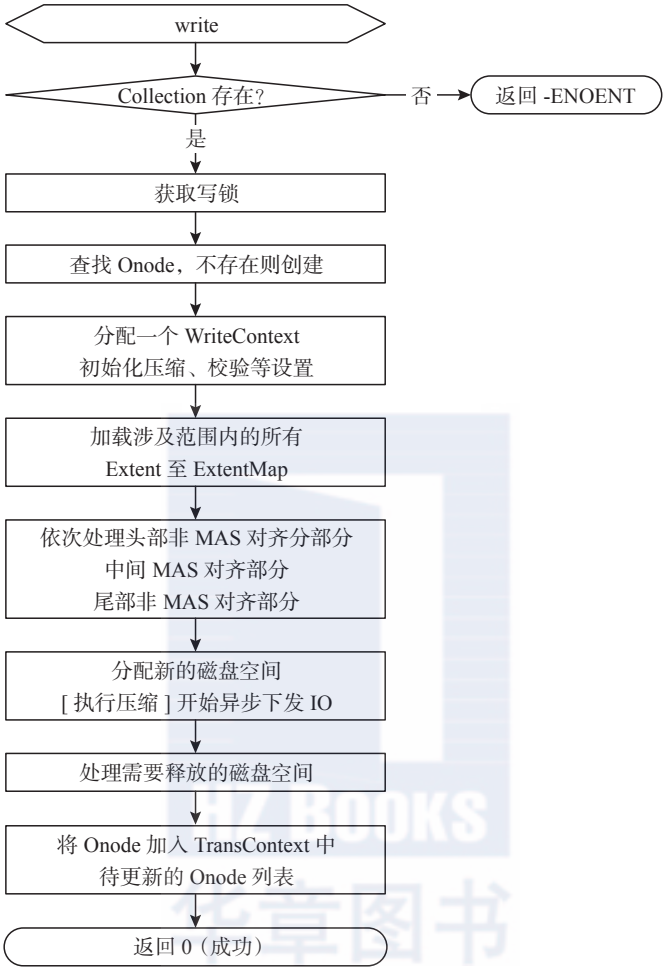

图 2-14　write

❑ 每次至多需要分配一个 Extent。

❑ Extent 中 blob_offset 不再为 0（参考表 2-7）。

❑ 待写入数据需要执行块对齐，无效部分使用全 0 填充，防止干净数据被污染。

对于覆盖写，如前所述，因为 BlueStore 对于 MAS 对齐部分总是执行 COW，所以其处理逻辑也和新写类似，不同之处在于此时需要找出所有波及范围内已经存在的 Extent 和它们占有的空间（当然也需要更新对应 Onode 的 ExtentMap，比如移除老的 Extent，然后加入新的 Extent 等），等待事务组同步完成后一并释放。对于头尾非 MAS 对齐的覆盖写，情况则比较复杂，需要额外考虑以下几个因素：

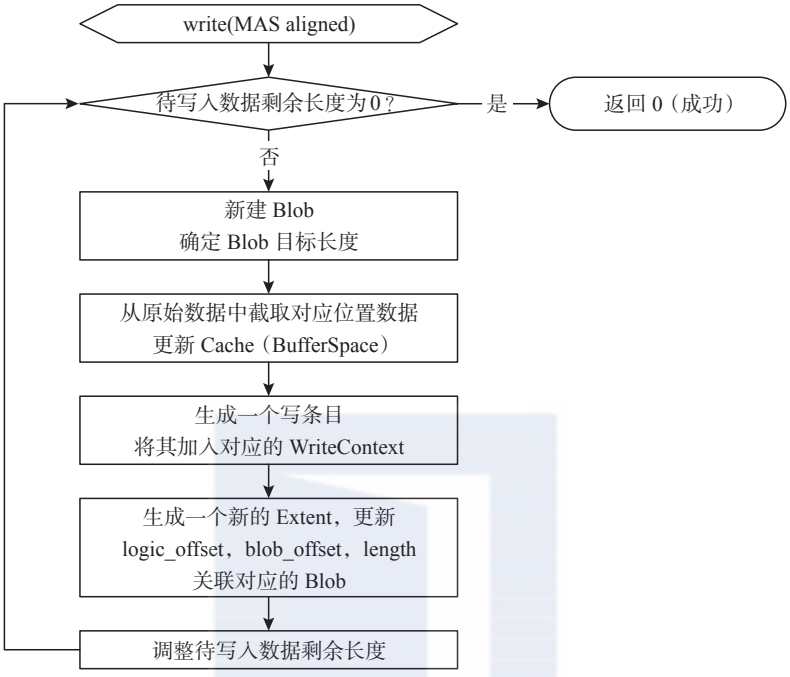

图 2-15 MAS 对齐写（新写）

（1）能否直接跳过，执行 COW

这种情况常见于对应的 Extent 被压缩过，因为此时执行 RMW 代价太大，所以直接采用 COW 策略。

（2）能否直接复用已有 Extent 的 unused 块

参考 2.2.2 节，如果设置的 MAS 大于基本块大小，那么 Extent 当中有可能产生以基本块大小为粒度的空穴，这些空穴会被 BlueStore 标记为 unused。因为块是磁盘操作的一个原子单位，所以针对这些状态为 unused 的块进行操作不会对 Extent 中的其他内容造成影响——例如写的过程中掉电，因为之前这部分内容本身就是无效的，所以即使写入了新的垃圾数据也不会造成任何负面影响。

当然如果新写入的内容不足一个 unused 块，那么无效部分（显而易见，无效部分在块的头部和 / 或尾部）仍然需要使用全 0 进行填充处理。

（3）是否需要执行 WAL 写

如果既不能执行 COW，也不能复用已有 Extent 的 unused 块，那么此时 RMW 操作已经不可避免，为了避免将已有数据写坏，此时需要使用 WAL。图 2-16 展示了 WAL 写

常见的几种情况（这里假定 MAS 为基本块大小。如果 MAS 不为基本块大小，则任何 WAL 写都可以分为头部非基本块对齐部分、中间基本块对齐部分和尾部非基本块对齐部分，其中头、尾非基本块对齐部分的处理逻辑和图 2-16 类似）：

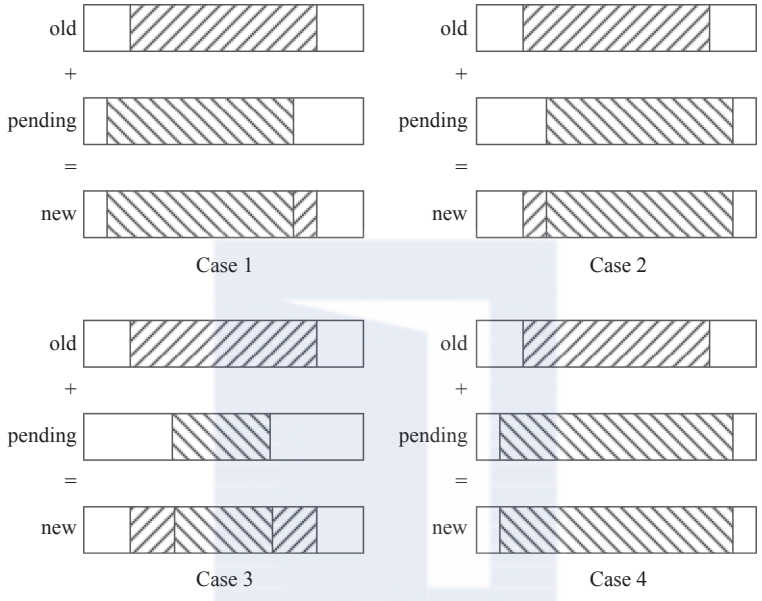

图 2-16　WAL 写

由图 2-16 可见，WAL 写的标准流程包括补齐读、合并、（使用全 0）填充 3 个步骤，完成填充后，最终得到一份新的待写入原有位置的数据，BlueStore 随后基于此数据生成一个日志事务，并将其加入对应的 TransContext。如前所述，此日志事务和 BlueStore 当中的其他元数据一样，是用 kvDB 保存的，只有对应的日志事务成功写入 kvDB 之后，才能开始对原有区域的数据执行覆盖写，加上之前的补齐读过程耗费了大量时间，所以 WAL 写效率实际上是非常低下的。

当所有的修改操作都被添加至 TransContext 时，就可以开始处理 TransContext 了，其处理过程可以分为如下 3 个泾渭分明的阶段：

（1）等待所有在途的写 I/O 完成

这一部分写 I/O 主要是所有非 WAL 写通过 COW 产生的 I/O，将被直接写入新的磁盘地址空间，因此可以在产生事务的过程中就开始同步执行。容易理解，如果此过程未完成即掉电，因为此时新的已分配出去的空间和老的待释放的空间尚未在 kvDB 中更新，

那么下次上电时不会产生任何影响。

（2）同步所有涉及的元数据修改至 kvDB

同一个事务组中所有修改所波及的元数据，包括已申请的空间和待释放的空间（如前所述，实际上只需要固化 FreelistManager），都会通过一个事务同步提交至 kvDB。由数据库的 ACID 属性，我们知道这个阶段要么全部完成，要么没有任何影响，从而可以保证数据一致性。

（3）通过 WAL 对应的日志事务执行覆盖写

至此阶段，因为对应 WAL 日志事务已经写入数据库，所以可以通过执行 WAL 日志重放安全的执行覆盖写。等待覆盖写的 I/O 完成之后，再次（在同步线程中同步）生成一个用于释放 WAL 日志事务条目的事务，并将其同步提交至 kvDB 等待其完成，最后将该事务组所有待释放磁盘空间加入 Allocator（如前所述，这是常驻内存的可分配磁盘空间列表），供后续事务组使用。需要注意的是，本阶段是可选的，如果没有 WAL 写，那么空间可以在（2）阶段完成后即释放至 Allocator。

如前所述，queue_transactions 接口被设计成异步的，因此上述所有涉及等待磁盘 I/O 完成的过程，都通过注册回调函数实现。后端块设备通过执行回调函数再次将对应的 TransContext 加入到 BlueStore 的同步线程，从而继续处理 TransContext，直至 TransContext 最终处理完成。

2.7 使用指南

至此，我们已经完整介绍了 BlueStore 的基本设计思想和主要实现细节，本节介绍如何部署 BlueStore，同时我们也列出了一些和 BlueStore 相关的配置参数，供高级用户进行性能调优时参考。

2.7.1 部署 BlueStore

如前所述，BlueStore 实现上非常灵活，一共可以支持 Slow、DB 和 WAL 3 种类型的块设备，其中 Slow 设备直接用于保存对象数据，DB 和 WAL 设备则用于保存和数据库相关的元数据。BlueStore 虽然实现上要比 FileStore 复杂，但是两者的部署却是类似的——例如 BlueStore 中的 DB 和 WAL 设备和 FileStore 当中的 Journal 设备类似，也是通过符号链接的形式在上电时挂载到 OSD 的当前工作目录下；每个 OSD 的主设备，除

了用于直接存储对象数据之外，还需要预留少量空间用于承载 OSD 启动时的引导数据，例如集群的 fsid，OSD 的 id（即序号）、fsid、keyring 等，这部分空间仍然需要依赖系统自带的本地文件系统接管（因为此时后端的 ObjectStore 还未正常上电，所以无法直接通过 ObjectStore 访问），因此如果使用 BlueStore，每个 OSD 的主设备还要在部署时再次划分为容量一大一小两个分区，其中较小的分区使用本地文件系统格式化，用于保存 OSD 启动时的引导数据；较大的分区则以裸设备的形式由 BlueStore 直接接管，充当 Slow 设备。据此，我们得到一个完整的基于 BlueStore 的 OSD 模型，如图 2-17 所示。

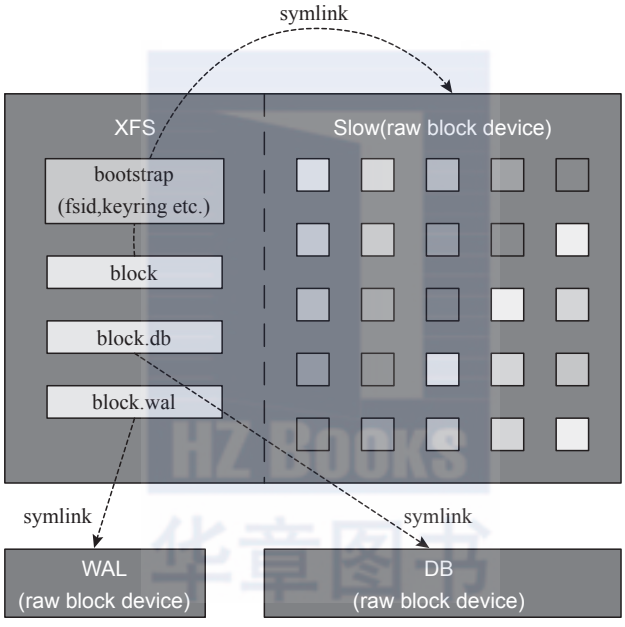

图 2-17　基于 BlueStore 的 OSD 组成模型

由图 2-17 可见，我们部署 BlueStore 时，首先将主设备划分为一大一小两个分区，其中小的分区（例如容量为 100MB）和 FileStore 类似，直接使用本地文件系统格式化后用于存放 OSD 的启动引导数据，同时作为 OSD 启动后的工作目录；大的分区直接作为一个裸的块设备，和另外两个裸块设备（或分区）一起通过符号链接的形式挂载到 OSD 的工作目录下。当然如果是全 SSD 阵列或者测试需要，也可以使用同一个块设备的 4 个分区完成 BlueStore 的部署。

如果所有分区或者块设备已经准备就绪，那么简单通过修改对应的 ceph.conf 文件即可完成 BlueStore 的部署：

```
[osd.0]
host = ceph-01
osd data = /var/lib/ceph/osd/ceph-0/
bluestore block path = /dev/disk/by-partlabel/osd-device-0-block
bluestore block db path = /dev/disk/by-partlabel/osd-device-0-db
bluestore block wal path = /dev/disk/by-partlabel/osd-device-0-wal
```

这样后续 BlueStore 上电时，将首先在 OSD 当前工作目录下，创建 3 个文件：

```
block
block.db
block.wal
```

然后通过读取 ceph.conf 配置文件，将这 3 个文件通过符号链接的形式指向对应分区或者块设备所在的真实路径。

当然也可以通过 ceph-disk 来自动完成 BlueStore 部署，为此，首先需要规划各个分区大小，同样可以通过 ceph.conf 文件指定，例如：

```
[global]
bluestore block db size = 67108864
bluestore block wal size = 134217728
bluestore block size = 5368709120
```

然后执行：

```
sudo ceph-disk prepare --bluestore /dev/sdb
```

上述命令将在 sdb 上创建 4 个分区，对应关系如下：

```
sudo sgdisk -p /dev/sdb
...
Number  Start (sector)    End (sector)  Size       Code  Name
   1            2048          206847   100.0 MiB   FFFF  ceph data
   2          206848          337919   64.0 MiB    FFFF  ceph block.db
   3          337920          600063   128.0 MiB   FFFF  ceph block.wal
   4          600064        11085823   5.0 GiB     FFFF  ceph block
```

当然也支持使用多个块设备，例如：

```
sudo ceph-disk prepare --bluestore /dev/sdb --block.db /dev/sdc
    --block.wal /dev/sdc
```

上述命令将在 sdb 上创建两个分区，然后通过符号链接，将 block 指向 sdb 的第 2 个分区；将 block.db 指向 sdc 的第 1 个分区；将 block.wal 指向 sdc 的第 2 个分区。或者更

进一步地通过：

```
sudo ceph-disk prepare --bluestore /dev/sdb --block.db /dev/sdc1 \
    --block.wal /dev/sdd1
```

将 block.db 和 block.wal 分离，分别指向不同块设备 sdc 和 sdd 的第 1 个分区。

需要注意的是，DB 和 WAL 设备的容量配置并非一成不变，而是需要根据 Slow 设备的容量、读写速率以及 RocksDB 本身的参数综合进行规划，目前推荐按照 Slow : DB : WAL = 100 : 1 : 1 进行配置，当然设置更大的 DB 和 WAL 设备容量效果更佳。

最后，因为目前 BlueStore 和 RocksDB 仍然被标记为实验性质，不推荐作为默认的 ObjectStore 后端，所以还需要额外增加如下全局配置才能正常启用 BlueStore：

```
[global]
enable experimental unrecoverable data corrupting features = bluestore rocksdb
osd objectstore = bluestore
```

2.7.2 配置参数

本节按照类别汇总截至目前所有和 BlueStore 相关的配置参数，如表 2-29 ～表 2-36 所示。

表 2-29 BlueStore 配置参数——部署

配置项	含义
bluestore_block_path	Slow 类型块设备所在路径
bluestore_block_db_path	DB 类型块设备所在路径
bluestore_block_wal_path	WAL 类型块设备所在路径
bluestore_bluefs	如果采用 RocksDB 作为默认的 kvDB 引擎，是否使用 BlueFS 作为 RocksDB 的本地文件系统，默认为 true
bluestore_kvbackend	采用的 kvDB 引擎，默认为“rocksdb”
bluestore_rocksdb_options	RocksDB 相关的配置选项[⊖]

表 2-30 BlueStore 配置参数——extent map

配置项	含义
bluestore_extent_map_inline_shard_prealloc_size	如果整个 extent map 编码后的长度比较小，那么可以直接将 extent map 作为一个整体进行存储（即不进行分片），此时可以将编码后的 extent map 直接在内存中进行缓存，本参数用于对相应的缓存进行预分配（亦即此类 extent map 编码后的期望长度）
bluestore_extent_map_shard_max_size	如果 extent map 中任意一个分片编码后的实际长度不在此范围内，则触发重新分片
bluestore_extent_map_shard_min_size	

⊖ https://github.com/facebook/rocksdb/wiki

（续）

配置项	含义
bluestore_extent_map_shard_target_size	单个 extent map 分片编码后的期望长度
bluestore_extent_map_shard_target_size_slop	当某个 Blob 跨越两个分片时，通过此系数对 bluestore_extent_map_shard_target_size 进行调整（以使得 Blob 不再跨越两个分片）

表 2-31 BlueStore 配置参数——Cache

配置项	含义
bluestore_2q_cache_kin_ratio	参考 2.3.1 节，如果 Cache 类型为 2Q，bluestore_2q_cache_kin_ratio 用于控制 A1in/Am 队列容量比例；bluestore_2q_cache_kout_ratio 用于间接控制“热度保留间隔”
bluestore_2q_cache_kout_ratio	
bluestore_cache_meta_ratio	缓存中元数据所占比重
bluestore_cache_size	单个 BlueStore 实例配置 Cache 大小，默认为 1GB
bluestore_cache_trim_interval	针对 Cache 执行淘汰的时间间隔
bluestore_cache_type	缓存类型，包含如下选项： ——2q（默认） ——lru

表 2-32 BlueStore 配置参数——磁盘空间管理

配置项	含义
bluestore_allocator	Allocator 类型，包含如下选项： ——bitmap（默认） ——stupid（本质上是 extent）
bluestore_bitmapallocator_blocks_per_zone	参考“2.4.3 BitmapAllocator”，这两个参数用于调整 BitmapAllocator 的拓扑结构
bluestore_bitmapallocator_span_size	
bluestore_freelist_type	FreelistManager 类型，包含如下选项： ——bitmap（默认） ——extent
bluestore_freelist_blocks_per_key	参考“2.4.2 BitmapFreelistManager”，用于调整将空间段以键值对形式写入 kvDB 时的值长度
bluestore_min_alloc_size	限制 Allocator 一次分配空间的上下限。bluestore_min_alloc_size 同时设置了 Allocator 分配空间的基本粒度，即 Allocator 分配的空间必须是 bluestore_min_alloc_size 的整数倍
bluestore_max_alloc_size	

表 2-33 BlueStore 配置参数——BlueFS

配置项	含义
bluefs_allocator	Allocator 类型，包含如下选项： ——bitmap（默认） ——stupid（本质上是 extent）
bluefs_alloc_size	最小可分配空间
bluefs_max_prefetch	如果进行预读，每次预读的最大字节数

表 2-34 BlueStore 配置参数——校验

配置项	含义
bluestore_csum_type	校验算法，包含如下选项： ——none ——xxhash32 ——xxhash64 ——crc32c（默认） ——crc32c_16 ——crc32c_8 不同的校验算法效率不同。对于同一种校验算法，最后的数字表明产生的校验数据长度，单位为比特。减少校验数据长度可以获得更好的数据库操作性能，但是会增加冲突概率

表 2-35 BlueStore 配置参数——压缩

配置项	含义
bluestore_compression_algorithm	压缩算法，包含如下选项： ——zlib ——snappy（默认）
bluestore_compression_mode	压缩策略，包含如下选项： ——force：强制进行压缩 ——aggressive：除非前端写请求携带不压缩提示，否则进行压缩 ——passive：除非前端写请求携带压缩提示，否则不进行压缩 ——none：不进行压缩（默认）
bluestore_compression_max_blob_size bluestore_compression_min_blob_size	如果允许对写入的数据进行压缩，BlueStore 将基于前端给出的不同写入提示设置合适的 Blob 大小。 例如对于顺序写，BlueStore 会将 Blob 的目标大小设置为 bluestore_compression_max_blob_size；否则（采用较为保守的策略）将 Blob 目标大小设置为 bluestore_compression_min_blob_size
bluestore_compression_required_ratio	针对数据执行压缩时，要求压缩后与压缩前数据所占用的目标磁盘空间比重必须小于此数值（即减小此数值要求压缩算法执行后取得更高的空间收益），否则放弃压缩

表 2-36 BlueStore 配置参数——事务

配置项	含义
bluestore_clone_cow	针对 clone 操作，是否执行 COW 策略
bluestore_max_bytes bluestore_max_ops bluestore_wal_max_bytes bluestore_wal_max_ops	流量控制（写抑制）。 bluestore_max_ops 用于指定最大并发操作（同事务，一个事务组可能包含多个事务）数；bluestore_max_bytes 用于指定最大并发字节数。如果包含 WAL 写，那么还需要同时满足并发数不得超出 bluestore_wal_max_ops 和 bluestore_wal_max_bytes 的约束。违反上述任一约束条件将触发写抑制
bluestore_wal_threads bluestore_wal_thread_ suicide_timeout bluestore_wal_thread_ timeout	BlueStore 使用线程池实现 WAL 写，这些参数用于调整线程池相关的配置。 bluestore_wal_threads 指定线程池中的服务线程个数。如果单个线程连续占用时间片超过 bluestore_wal_thread_timeout，那么对应的 OSD 进程会被标记为 unhealthy；如果超过 bluestore_wal_thread_suicide_timeout，那么对应的 OSD 进程会自杀

2.8 总结与展望

BlueStore 于 2015 年年中引入，目标是替换已经服役超过 10 年的 FileStore，作为新一代默认的 ObjectStore 引擎，提升 FileStore 一直为人诟病的写性能。FileStore 存在如下缺陷：

- 强烈依赖本地文件系统，但是真正能够完美适配 FileStore 的则几乎没有；默认的 XFS 仍然过于重量级，很多功能对于 FileStore 而言不是必须的。
- 基于 POSIX 语义的目录层级结构使得针对 PG 中的对象进行顺序遍历非常困难。
- 数据和元数据分离不彻底。
- 日志叠加日志的设计使得写放大现象异常严重，从而严重制约写性能。
- 流控机制不完整导致 IOPS 和带宽抖动（FileStore 自身无法控制本地文件系统的刷盘行为）。
- 频繁 syncfs 系统调用导致 CPU 利用率居高不下。

在定位上，BlueStore 期望解决 FileStore 上述缺陷的同时带来至少 2 倍的写性能提升和同等的读性能，此外，增加对于未来一些新型存储介质例如 NVMe SSD 以及诸如数据自校验、数据压缩等热点增值功能的支持也是必选项。基于当前版本的实测数据（测试工具为 FIO），这个目标在基于 NVMe SSD 的全闪存阵列当中已经部分实现，512KB 及以上块的顺序读写场景中实现了接近 2 倍的性能提升，但是 BlueStore 在主打机械磁盘的传统阵列中的表现仍然差强人意，小粒度例如 4KB 块的随机读写性能与 FileStore 相比提升有限，因此针对 BlueStore 进行性能优化依然任重而道远，好在 BlueStore 设计得非常灵活，几乎所有关键组件都可定制，可以随时使用更好的方案进行替换，这为 BlueStore 后续取得长足进步奠定了良好的基础。