

**Bluestore流程****梳理**

**张杨 冯宇**

目录

[Bluestore流程梳理 1](#_Toc462819442)

[1、 概述 2](#_Toc462819443)

[2、 BlueStore概述 2](#_Toc462819444)

[3、 osd启动流程 3](#_Toc462819445)

[4、 BlueStore元数据逻辑结构 4](#_Toc462819446)

[5、 bluestore空间管理 5](#_Toc462819447)

[5.1 BlueFs的空间管理 5](#_Toc462819448)

[5.2 BlueStore的空间管理 6](#_Toc462819449)

[5.3 BitMapAllocator分析 6](#_Toc462819450)

[5.4 BitmapFreelistManager分析 8](#_Toc462819451)

[6、 Bluestore读写过程 9](#_Toc462819452)

[6.1 写数据 9](#_Toc462819453)

[6.2 读数据 11](#_Toc462819454)

[6.3 删除数据 12](#_Toc462819455)

[7、 Bluefs介绍 12](#_Toc462819456)

[7.1 bluefs元数据 12](#_Toc462819457)

[7.2 bluefs文件操作 13](#_Toc462819458)

[7.3 总结 14](#_Toc462819459)

[8、 BlueStore事务处理流程 14](#_Toc462819460)

[8.1 BlueStore事务概述 14](#_Toc462819461)

[8.2 BlueStore事务处理流程 14](#_Toc462819462)

## 概述

本文主要关注数据落盘的具体过程，即bluestore的io逻辑和主要读写的流程。

参考代码版本：v11.0.0

## BlueStore概述

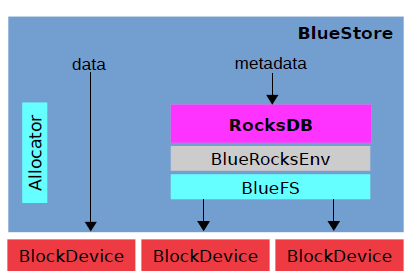


图 1 bluestore整体结构

Bluestore为解决filestore在写数据前先写journal产生的写放大，并针对ssd做了相应的优化。Bluestore直接操作裸盘，尽可能的避免文件系统（xfs、ext4）的开销。操作裸盘则盘空间管理由allocator（默认是BitmapAllocator）处理。Bluestore在io过程中产生的元数据，元数据则存放到rocksdb kv数据库中。rocksdb存数据依赖文件系统，但是rocksdb将底层系统相关的抽象为env，用户程序只需要实现相应的接口就可以提供底层的系统的封装。Bluestore实现了BlueRocksEnv，并实现了Bluefs支撑BlueRocksEnv。Bluefs的日志和数据文件都是保存在裸盘上，和用户数据共享裸盘设备，也可以分开指定不同设备。



图 2 osd设备数据分区

osd目录分区：默认占用100M，默认挂载xfs文件系统。提供基本的描述信息，包括whoami（osd编号），osd的type，osd的魔术字，block块设备入口等。

Block dev label分区：默认占用4K。存储bluestore\_bdev\_label\_t结构体信息，包含osd\_uuid，块设备size，设备描述信息，创建时间等。

Bluefs supper block分区：默认占用4K。存储bluefs\_super\_t结构体信息，包含osd\_uuid，version和block\_size等。

DB数据分区：默认占用MAX[1G，块设备总大小的4%]，保存DB数据和日志数据。

对象数据分区：默认占用剩余空间，存放对象业务数据。

## osd启动流程



图 3 osd启动流程

osd服务启动主要工作：

1、接收，检查参数，根据参数确定osd工作内容，图3为osd创建完成后的启动流程。

2、初始化网络层。

3、加载objectstore，这里以bluestore为例说明。

4、恢复pg，确定osd crush location。

5、注册网络接收函数dispatcher，开始监听等待消息。

## BlueStore元数据逻辑结构

Onode是bluestore中存放元数据的重要数据结构，由于bluestore直接管理裸盘，则需要元数据用来管理落盘的对象。Onode平时工作在内存中，会在io完成后把更新持久化到rockdb中。 BlueStore加载时会从DB中载入一部分Onode，其引用同时存入map和LRU链表。LRU链表的作用是控制内存中Onode数量上限。若要访问的Onode不在map中，则从DB中读取。

Onode包含一个set存放了多个extent，这个extent是作为逻辑单元来存放用户的业务数据的，包含这段数据的逻辑起始地址offset和数据的长度length，也称logical extent。extent又是存放到一个叫blob逻辑结构里，一个blob可以对应N个extent（N>=1）。blob和具体物理盘相对应，blob包含一个vector<bluestore\_pextent\_t>，bluestore\_pextent\_t就是由allocator分配的物理块，它指向物理盘上可用的空间。

总的来说，一个对象对应一个onode，一个onode包含多个extent，多个extent存放在一个或者多个blob里，每个blob又对应到多个pextent，就是具体的物理块，这样就完成了extent到pextent的映射。



图 4 元数据类关系图

元数据映射关系：



图 5 元数据映射关系

## bluestore空间管理

与存储相关的类图如下。



图 6 BlueStore存储空间管理相关类图

Allocator类（纯虚类）负责对块设备空间进行管理，该类具体有StupidAllocator和BitMapAllocator两个子类，默认采用后者。

BlueStore和BlueFs都要在块设备上存取数据，两者都包含有指向Allocator子类实例的指针，对各自的块设备存储空间进行管理。

默认配置下，bluestore\_block\_db\_create和bluestore\_block\_wal\_create都为false，表示不对BlueFs配置单独的存放DB和WAL数据的路径，此情况下，BlueFs和BlueStore共享块设备。

BlueStore和BlueFs（BlueFs加载是BlueStore加载过程中的一个步骤）在加载时，会创建各自的Allocator子类实例时，此时该实例并不知道其管理的存储空间有哪些已经被分配出去了，需要BlueStore和BlueFs恢复其空间分配信息。

### 5.1 BlueFs的空间管理

BlueFs的存储空间分配信息是保存在BlueFs日志文件中的。

运行osd服务前，需要先创建BlueFs。具体流程是：

1. 为BlueFs指定块设备。
2. 创建Allocator子类实例，将初始分配给BlueFs的块设备地址空间（由配置文件以及块设备大小确定），添加到Allocator作为未分配的空间。
3. 将以上分配地址空间的信息作为一个事务操作写入BlueFs的日志文件，事务操作类型为OP\_ALLOC\_ADD。日志文件是BlueFs创建的第一个文件，创建该文件时会向Allocator申请一段地址空间用于存放日志文件的内容，大小由配置项bluefs\_max\_log\_runway确定。
4. 写入BlueFs的超级块信息到块设备上的固定位置（offset为4K，长度为4K的区间），超级块中有日志文件的元数据信息（结构体bluefs\_fnode\_t），可以索引到存放日志文件内容的具体block。

启动osd服务时，BlueFs通过以下步骤恢复当前实际存储空间分配状态：

1. 指定块设备。
2. 创建Allocator子类实例，此时Allocator还没有未分配的地址空间。
3. 从块设备超级块所在位置读取BlueFs超级块信息。
4. 通过超级块中的日志文件元数据信息，读取日志文件，回放日志。

其中，对操作类型为OP\_ALLOC\_ADD的事务，处理即为向Allocator中添加地址空间作为BlueFs的未分配空间。

由于BlueStore和BlueFs之间可以动态调整存储空间大小，所以也可能从BlueFs中回收空间，此情况下日志中会有操作类型为OP\_ALLOC\_RM的事务，对此的处理为将这部分空间从Allocator未分配空间中移除。

1. 对BlueFs中已经存在的文件，将其使用的空间从Allocator未分配空间中移除。

说明：除了日志文件的元数据是通过超级快获取，BlueFs中其它文件的元数据都是通过日志回放恢复的，创建\删除\更新文件的操作，都会记录到日志文件中。

### 5.2 BlueStore的空间管理

与BlueFs通过日志保存存储空间分配信息不同，BlueStore借助FreelistManager的子类（ExtentFreelistManager和BitmapFreelistManager，默认采用后者）将存储空间分配信息保存到DB中。可以这样理解，Allocator负责实际存储空间的分配，FreelistManager配合Allocator将分配信息记录到DB中，以便重启后能恢复空间分配状态。Allocator分配或者回收空间后，都会通过FreelistManager更新DB中的分配信息。

### 5.3 BitMapAllocator分析

BitMapAllocator是BlueStore默认采用的Allocator的实现，它以位图形式记录每个块是否已经被分配，相关类图如下。



图 7 BitMapAllocator类图

创建BitMapAllocator实例时，会根据传入的块设备大小以及块大小，构建了一颗树，如下图所示。



图 8 BitMapAllocator树状组织图

BitMapAreaIN为非叶子节点。BitAllocator继承自BitMapAreaIN，对应于根节点。BitMapAreaLeaf对应叶子节点，BitMapZone作为BitMapAreaLeaf的子节点（所以严格来说，BitMapAreaLeaf并不是叶子节点），保存位图数组。BmapEntry实际是一个long型整数，每一位bit表示一个block的是否被使用，每个一个Zone最多可以表示bluestore\_bitmapallocator\_blocks\_per\_zone个连续的block的使用情况。

父节点（Zone没有子节点）最多可以有bluestore\_bitmapallocator\_span\_size个子节点。每个父节点下的子节点都按顺序从0开始编号，每个子节点都覆盖一段连续的block空间，相邻序号的子节点表示的block空间是连续的。那么，父节点所覆盖的block空间就是所有子节点的block空间合并起来的一个更大的连续block空间。

分配或者释放block时，即通过offset和长度从根节点找到Zone节点，修改对应的BmapEntry中的bit位。

BitMapAllocator实例化后，每个block初始都是已分配状态，需要BlueStor或者BlueFs在加载过程中调用init\_add\_free接口恢复其未分配的空间。

*问题：为何要引入BitMapZone，而不是在BitMapAreaLeaf中保存位图数组？*

### 5.4 BitmapFreelistManager分析

BitmapFreelistManager也是使用位图表示block的分配状态，位图数据被写入RocksDB中。

RocksDB是key/value数据库，我们可以把它简单地理解为对调用者提供了一个map。如图6所示，BitmapFreelistManager将其管理的存储空间划分为多个段，每段的起始地址作为key。Key对应的value就是表示这段连续的block分配状态的位图数组，某一bit位置1表示对应的block已经被分配。



图6 BitmapFreelistManager存储空间管理示意图

上文2.2节提到，BlueStore依赖FreelistManager恢复实际存储空间分配状态，BitmapFreelistManager对此是通过遍历每个key的value，找到所有bit位为0的区间，返回给BlueStore，由BlueStore添加到Allocator作为未分配的空间。

*问题：BitmapFreelistManager在分配或者释放时，是将待操作的block对应的bit位置1，然后和DB中保存的位图数据做异或操作，那么就需要上层调用者保证对某个block的分配和释放必须是交替出现的，而且必须是先分配再释放。如果对某一个block连续分配两次，DB该block对应的bit的变化过程是 0 🡪 1 🡪 0，变为未分配状态，而连续释放两次也会将block变为已分配状态。修改DB前，需要做一个判断来防止重复分配或者释放。*

## Bluestore读写过程

到达bluestore的io的offset和length都是当前对象内的逻辑地址，offset是相对于这个对象的逻辑起始地址。

### 6.1 写数据

写流程分为small write和big write，首先根据写入数据长度来确定是小写还是大写，通过对min\_alloc\_size（blob的最小分配大写默认64K）取模，如果小于64K为小写流程，如果大于64K则为大写流程。下图为一个io被拆分的几种情况：



图 9 小写大写场景

1）do\_write\_small

小写处理io长度小于64k的情况，先根据offset查找extent，如果有extent落在offset附近，则判断offset+length是否在已有的blob范围内，所以存在3种场景：1、没有合适的blob，需要新创建blob。2、有合适的blob，将数据写到未使用的空间上。3、有合适的blob，数据会有覆盖写情况。数据写盘是按照block\_size（默认是4K）对齐的，按照direct I/O方式写盘，所以offset+length需要按照4K对齐，不能对齐部分需要补0。

1. 没有合适的blob，新建blob

创建新的blob，大小为min\_alloc\_size 根据标识判断是否需要写到cache中，再对齐block\_size前后补0，用offset和length生成新的extent，最后等待分配可用物理块再落盘。



图 10 新建blob

1. 有合适的blob，将数据写到未使用空间上

先找到可用的blob，再把数据按照block\_size对齐补0，然后再看下当前blob有没有未使用的空间存放这段数据，如果有的则创建新extent，然后再提交aio在等待队列中。



图 11 空闲位置创建extent

1. 有合适的blob，数据会有覆盖写情况

如果找到可用的blob，但是blob中没有满足可用的空间了，说明这笔io有覆盖原数据的情况。如果覆盖写是block\_size对齐的，则不需要读原数据。如果覆盖写没对齐，则需要把没对齐原数据部分读出来，与当前数据组成一个新的对齐的buffer，然后再生成新的extent，再调整extent位置，记录需回收的extent。再生成一个wal对象，先将wal写到db中，然后再写到盘上。



图 12 覆盖写过程

2）do\_write\_big

处理IO长度为min\_alloc\_size整数倍的情况，生成新blob，blob长度可以为min\_alloc\_size的整数倍，再生成extent，如果有标识表明需要写cache时先写到cache，最后再写入物理盘。如果有覆盖的情况，则要标记原有extent后续做释放回收。

### 6.2 读数据

首先检查读请求里是否有预读cache的标识位，如果有则需要将读到未命中的数据添加到cache中。如果此时有io的wal未完成，需等待wal数据写到块设备，完成整个wal I/O流程后，再开始读数据。先根据offset查找extent，可能存在一种情况就是有效数据和开始读取位置之间存在一段空洞hole，需要跳过这段hole，再然后确定数据在blob内的偏移b\_off和在blob中读取的数据长度b\_len。然后先读cache，如果cache有不命中的部分记录下来，记录到blobs2read中，然后再根据blobs2read中记录的blob的位置再从盘上把数据同步读出来，如果有预读为true，还需要添加到cache中。最后构造数据buffer返回。



图 13 读数据

### 6.3 删除数据

先等wal流程完成，数据全下盘后，再标记onode中extent，然后释放相关的extent，然后将onode size置为0，再把omap相关在db中数据全部删除，再将onode置为无效，最后把db中记录的onode数据删除。

## Bluefs介绍

bluefs一个简单的用户态的文件系统，专门设计是用来支撑rockdb，对接rockdb的Env接口。它不是一个通用文件系统，只需要支持rockdb需要的文件操作。bluefs只支持一层目录，不支持随机写。

### 7.1 bluefs元数据

bluefs在mount后会在内存中生成3个数据结构：bluefs的超级块**supper**，bluefs全局目录表**dir\_map**，bluefs全局文件表**file\_map**。supper数据起始位置在块设备上第4K，数据的长度为4K。它内容包含uuid、block\_size及log\_fnode等，其中log\_fnode记录了bluefs的事务日志，启动mount bluefs的时候会把这块事务日志读取出来生成bluefs所需要的元数据，这步在\_replay()里完成，\_replay()之后就会得到bluefs当前最新的目录和文件信息即dir\_map和file\_map。dir\_map包含dir\_name和DirRef键值对，Dir包含了一个描述本目录内所有文件的表file\_map，最后描述文件的file结构体，包含了bluefs\_fnode\_t和refs引用计数等，bluefs\_fnode\_t描述了allocator分配给file的物理块bluefs\_extent\_t，后续落盘会用使用。全局file\_map维护ino与file对应关系，ino为从1开始的整型数，ino为1的是特殊的事务日志文件，新增文件会将ino加1。



图 14 bluefs元数据

### 7.2 bluefs文件操作

#### 7.2.1 创建目录

创建一个空目录在mkdir()中实现，首先遍历查询下是否是已存在的目录，如果不存在则向dir\_map添加一条记录，并记录事务日志。后续会将日志文件刷盘。

#### 7.2.2 创建文件

创建文件，首先是要得到指向这个文件的FileWriter对象指针，这步在open\_for\_write()中完成。open\_for\_write()先根据目录名称找到目录位置，然后再生成一个file对象，同时更新全局file\_map和dir\_map中的file信息，再把ino更新，如果是覆盖文件，还需要把覆盖的文件占用物理空间释放。然后把创建动作记录到事务日志中，最后再生成该文件的FileWriter对象并返回。在得到FileWriter对象后，就可以生成文件内容buffer。得到buffer后则需要刷盘这步是在fsync()中完成，fsync()先将文件buffer刷盘，然后再把事务日志刷盘。

#### 7.2.3修改名称

修改名称是把old\_dir中old\_file放到new\_dir的new\_file中，old\_dir和new\_dir可以是相同目录。过程是先找到old\_dir中old\_file对象，然后查找new\_dir，得到new\_dir后在查询是否已经存在和old\_file同名的文件，如果有则需要删除并释放空间，然后记录到事务日志中。如果没有则是new\_file指向old\_file的file对象，并且old\_dir删除old\_file记录，然后记录到事务日志中。（这个功能类似目录间文件拷贝操作）

### 7.3 总结

bluefs通过BlueRocksEnv来实现rockdb的相关底层操作，它封装了顺序读，随机读，顺序写等接口，这些接口提供给上层db调用。bluefs每做一次操作都会记录事务日志，记录日志的目的就是为了在启动bluestore的时候在mount bluefs的时候恢复出bluefs的元数据，相当于这块日志记录的是元数据。

## BlueStore事务处理流程

### 8.1 BlueStore事务概述

所有涉及到数据修改的请求，比如写对象数据请求、修改对象属性请求等，BlueStore都是以事务形式（调用者必须实例化一个事务传递给BlueStore）对外提供的，以此保证请求执行的原子性。

对外主要提供两个接口：

1. queue\_transaction，异步接口，事务处理完成后回调调用者。
2. apply\_transactions，同步接口，调用者所在线程会等待事务完成后继续往下执行。

同步接口实际上是对queue\_transaction进行了封装，增加了等待条件变量（pthread\_cond\_t）的步骤以阻塞当前线程，通过queue\_transaction提交的事务完成后，会回调唤醒该线程继续执行。

所以不管哪种接口，事务在BlueStore内部的处理是一样的，流程包括对象数据写入块设备、对象元数据和WAL数据（WAL数据描述了一个写操作的信息，包括要写的数据本身，以及数据要写到哪里的信息。覆盖写才可能会触发WAL）写DB、WAL数据写入块设备等。事务的状态会伴随着处理流程变迁。

### 8.2 BlueStore事务处理流程

BlueStore事务处理流程中涉及到的线程包括：

1. 接口调用者所在线程，典型的是OSD的osd\_op\_tp线程池线程，读写请求多是在该线程中发起的。
2. BlockDevice监测aio完成事件的线程，如KernelDevice的aio\_thread线程。
3. BlueStore的kv\_sync\_thread，主要负责持久化DB数据。
4. BlueStore的wal\_tp线程，负责将WAL数据写入块设备。
5. Finisher::finisher\_thread线程，负责回调接口调用者。



上图展示了一个需要写WAL数据的写请求事务处理流程，红色字体部分表示事务的状态。可见，其核心在于事务的状态变化处理逻辑，即图中的\_txc\_state\_proc函数，该函数根据事务所处的状态进行不同阶段的处理。下图是与之对应的事务状态迁移图。



需要说明的是，对上层的回调是在写DB完成后进行的，而不是整个事务流程结束才回调。对上层的回调包括两个层次。首先是ondisk回调，告诉上层数据已经持久化了。然后是onreadable回调，这是告诉上层数据可以读了。对于非WAL写请求，这种处理显然没有问题，因为此时对象数据和对象元数据都已经写到了它们该在的位置。对于WAL写请求，此时对象数据只是暂时性地写到了DB，如果上层立刻发起对该数据的读请求，则需要等待之前的WAL写请求事务处理完成，将对象数据从DB中“搬到”它应该在的实际位置之后才能继续。