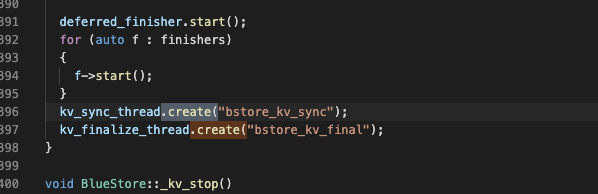
**BlueStore源码分析-核心数据结构**

**数据结构分析**

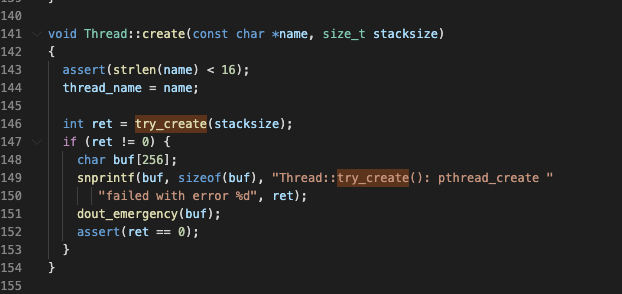
**Thread**

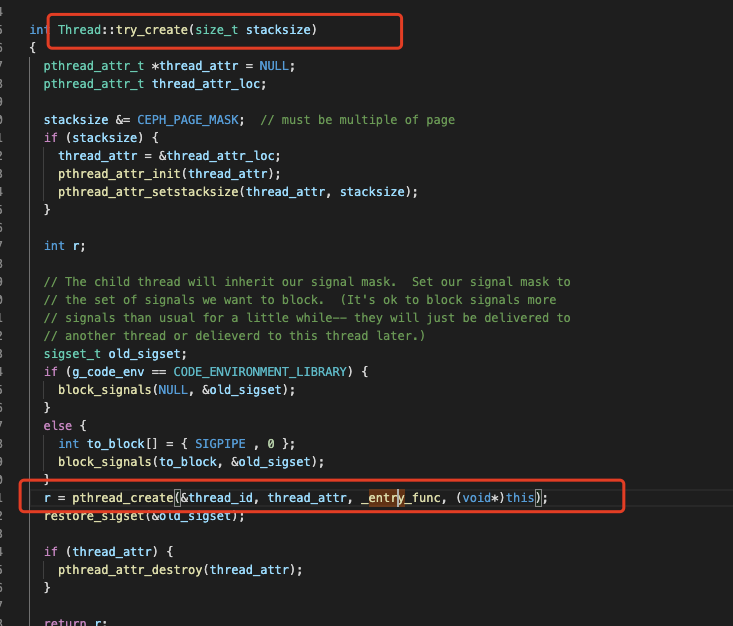
先看下怎么使用的:

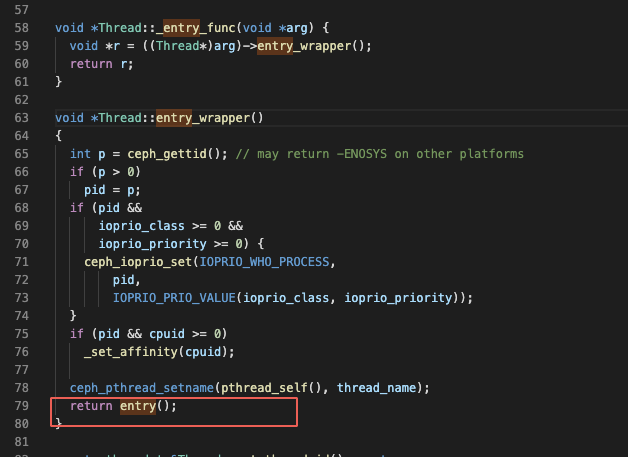
使用方法类似这样:调用create接口:



下边是代码细节:







Thread是对底层Linux线程的封装.主要是包含了一些优先级的信息,方便调度.

代码定义位置:src/common/Thread.h Thread.cc

**KVFinalizeThread**

1. struct KVFinalizeThread : public Thread
2. {
3. BlueStore \*store;//bluestore实例
4. explicit KVFinalizeThread(BlueStore \*s) : store(s) {}
5. void \*entry()//重写了entry函数
6. {
7. store->\_kv\_finalize\_thread();//处理kv\_finalize队列的IO
8. return NULL;
9. }
10. };

**KVSyncThread**

同上

1. struct KVSyncThread : public Thread
2. {
3. BlueStore \*store;
4. explicit KVSyncThread(BlueStore \*s) : store(s) {}
5. void \*entry() override
6. {
7. store->\_kv\_sync\_thread();//处理kv\_queue的IO事件
8. return NULL;
9. }
10. };

**MempoolThread**

//暂时没看懂为啥有这个,内存池?还是线程?

1. struct MempoolThread : public Thread
2. {
3. BlueStore \*store;
4. Cond cond;
5. Mutex lock;
6. bool stop = false;
7. }

**OldExtent**

1. struct OldExtent
2. {
3. boost::intrusive::list\_member\_hook<> old\_extent\_item;//hook,记录本对象内的双向链表
4. Extent e;//old对应的逻辑Extent
5. PExtentVector r;//old对应的物理硬盘上的Extent
6. bool blob\_empty;
7. }
9. typedef boost::intrusive::list<
10. OldExtent,
11. boost::intrusive::member\_hook<
12. OldExtent,
13. boost::intrusive::list\_member\_hook<>,
14. &OldExtent::old\_extent\_item>>
15. old\_extent\_map\_t;
16. 使用的时候是typedef了一个old\_extent\_map\_t,可以抽象为map使用,使用oldExtent就能O(1)时间找到这个实例的位置,而且删除也很方便,遍历其它old也很方便,查找就像map,删除啥的就像list,都是O(1)时间复杂度(不知道理解的对不对),这个是boost的入侵式容器.

在bluestore里面,OldExtent主要是用来记录deferredWrite的时候,一些需要回收的空间,这些回收空间是因为被覆盖写了,所以旧的空间需要被记录然后回收.

**Cache**

Cache是BlueStore用来实现PG操作的缓存的基类.

1. /// a cache (shard) of onodes and buffers
2. struct Cache
3. {
4. CephContext \*cct;
5. PerfCounters \*logger;//缓存命中率相关统计
6. std::recursive\_mutex lock; ///< protect lru and other structures
8. std::atomic<uint64\_t> num\_extents = {0};//缓存的Extent数量
9. std::atomic<uint64\_t> num\_blobs = {0};//缓存的blob数量
10. ...
11. }

缓存既可以缓存用户数据,也可以缓存元数据.所以缓存面向用户数据和Onode进行设计.因为在元数据中,主要大的数据类型有Collection和Onode,而Collection是内存管理结构,可以直接加载在内存,而Onode是和磁盘容量正相关的,所以可能需要进行缓存淘汰.

**TwoQCache**

具体原理暂时未了解，但是和ARC(A Self-Tuning, Low Overhead Replacement Cache)自调优、低开销的替换缓存是一个原理，主要是在frequency和recency之间动态调整缓存策略。

**LRUCache**

1. struct LRUCache : public Cache
2. {
3. private:
4. typedef boost::intrusive::list<
5. Onode,
6. boost::intrusive::member\_hook<
7. Onode,
8. boost::intrusive::list\_member\_hook<>,
9. &Onode::lru\_item>>
10. onode\_lru\_list\_t;
11. typedef boost::intrusive::list<
12. Buffer,
13. boost::intrusive::member\_hook<
14. Buffer,
15. boost::intrusive::list\_member\_hook<>,
16. &Buffer::lru\_item>>
17. buffer\_lru\_list\_t;
19. onode\_lru\_list\_t onode\_lru;
21. buffer\_lru\_list\_t buffer\_lru;
22. uint64\_t buffer\_size = 0;
23. }

主要是提供LRU逻辑的Onode缓存功能,当需要生成一个空闲的Onode的时候,尝试从缓存中获取.LRU的逻辑由LRUCache实现,存储的逻辑由底层的boost库实现.

**Shard**

**ExtentMap**

索引一个对象下边所有的Extent

1. /// a sharded extent map, mapping offsets to lextents to blobs
2. struct ExtentMap
3. {
4. Onode \*onode;
5. extent\_map\_t extent\_map; ///< map of Extents to Blobs,一个对象内的所有Extent
6. blob\_map\_t spanning\_blob\_map; ///< blobs that span shards,用来存储夸分片共享的blob们
8. struct Shard
9. {
10. bluestore\_onode\_t::shard\_info \*shard\_info = nullptr;
11. unsigned extents = 0; ///< count extents in this shard
12. bool loaded = false; ///< true if shard is loaded
13. bool dirty = false; ///< true if shard is dirty and needs reencoding
14. };
15. mempool::bluestore\_cache\_other::vector<Shard> shards; ///< shards
16. ...
17. }

因为存在稀疏写,所以每个extent不一定都是相邻的,我们叫这种现象叫做空洞,可以理解为碎片,如果空洞过多,可能磁盘的碎片化会比较严重,所以,需要对超过一定大小的元数据信息进行独立存储,称作shard\_info.

具体的分片过程是大致估算一下当前ExtentMap中每个Extent的编码后大小,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ExtentMap编码存盘的依据.

此外,如果blob被某两个相邻的Extent共享且这两个Extent又隶属于不同的分片范围,那会因为该blob夸分片而无法确定归属,所以会有限针对该blob进行分裂,如果失败的话就会放入spanning\_blob\_map,后续将spanning\_blob\_map和对象的其它元数据一并作为单条记录进行保存.

**shard\_info**

1. struct shard\_info {
2. uint32\_t offset = 0; ///< logical offset for start of shard
3. uint32\_t bytes = 0; ///< encoded bytes
4. DENC(shard\_info, v, p) {
5. denc\_varint(v.offset, p);
6. denc\_varint(v.bytes, p);
7. }
8. void dump(Formatter \*f) const;
9. };

**Extent**

是用来管理对象的逻辑结构,称为逻辑段,数据校验,压缩,数据共享等都是基于extent粒度实现的.

1. struct Extent : public ExtentBase
2. {
3. MEMPOOL\_CLASS\_HELPERS();
5. uint32\_t logical\_offset = 0; ///< logical offset
6. uint32\_t blob\_offset = 0; ///< blob offset
7. uint32\_t length = 0; ///< length
8. BlobRef blob; ///< the blob with our data,负责将逻辑段内的数据映射至物理盘
9. ...
10. }

**pextent**

是描述物理介质的

1. /// pextent: physical extent
2. struct bluestore\_pextent\_t : public AllocExtent {
3. //继承了两个核心的成员
4. //offset:磁盘上的物理偏移
5. //length:长度
6. }

**DeferredBatch**

**Collection**

磁盘对应的内存管理结构

**BufferSpace**

为了考虑缓存和上层管理结构之间的对应关系,可以使用blob中用户数据到缓存之间建立一个二级索引,这样能通过该结构在缓存和用户数据之间进行访问.

1. /// map logical extent range (object) onto buffers
2. struct BufferSpace
3. {
4. typedef boost::intrusive::list<
5. Buffer,
6. boost::intrusive::member\_hook<
7. Buffer,
8. boost::intrusive::list\_member\_hook<>,
9. &Buffer::state\_item>>
10. state\_list\_t;
11. mempool::bluestore\_cache\_other::map<uint32\_t, std::unique\_ptr<Buffer>>
12. buffer\_map;
13. // we use a bare intrusive list here instead of std::map because
14. // it uses less memory and we expect this to be very small (very
15. // few IOs in flight to the same Blob at the same time).
16. state\_list\_t writing; ///< writing buffers, sorted by seq, ascending,包含脏数据的缓存队列
17. };

**Buffer**

是BufferSpace管理的基本单元,一个Buffer Blob当中的一段数据.

1. /// cached buffer
2. struct Buffer
3. {
4. MEMPOOL\_CLASS\_HELPERS();
6. enum
7. {
8. STATE\_EMPTY, ///< empty buffer -- used for cache history
9. STATE\_CLEAN, ///< clean data that is up to date
10. STATE\_WRITING, ///< data that is being written (io not yet complete)
11. };
12. enum
13. {
14. FLAG\_NOCACHE = 1, ///< trim when done WRITING (do not become CLEAN)
15. // NOTE: fix operator<< when you define a second flag
16. };
18. BufferSpace \*space;//归属的BufferSpace实例
19. uint16\_t state; ///< STATE\_\*
20. uint16\_t cache\_private = 0; ///< opaque (to us) value used by Cache impl
21. uint32\_t flags; ///< FLAG\_\*
22. uint64\_t seq;
23. uint32\_t offset, length;
24. bufferlist data;
26. boost::intrusive::list\_member\_hook<> lru\_item;
27. boost::intrusive::list\_member\_hook<> state\_item;
28. ...
29. }

**AioContext**

**TransContext**

**Blob**

是extent和pextent之间的映射结构，而且extent和blob不是一对一的关系，一个blob只是表示它代表的哪些pextent.

**KeyValueDB**

**DB**

**Env**

**Slice**

**Snapshot**

**RocksDBStore**

**Allocator**

**FreelistManager**

**BlueFS**

**Collection**

collection可以看做是PG在bluestore上的逻辑结构

**Cache**

**OnodeSpace**

这个结构主要是用来建立Onode和Collection之间的映射的关系的.这样在Collection级别操作Onode的时候不需要遍历.

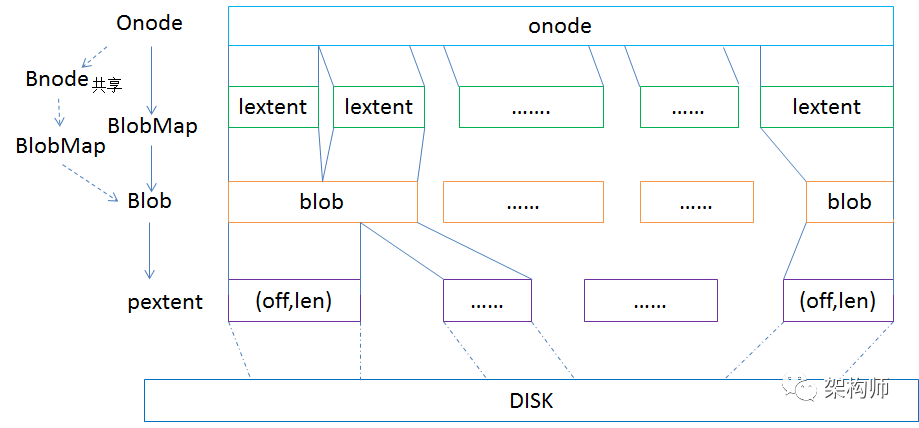
1. struct OnodeSpace
2. {
3. private:
4. Cache \*cache;//归属的cache,一个Bluestore中可能会有多个cache实例
6. /// forward lookups
7. mempool::bluestore\_cache\_other::unordered\_map<ghobject\_t, OnodeRef> onode\_map;//查找表
9. friend class Collection; // for split\_cache()
11. }

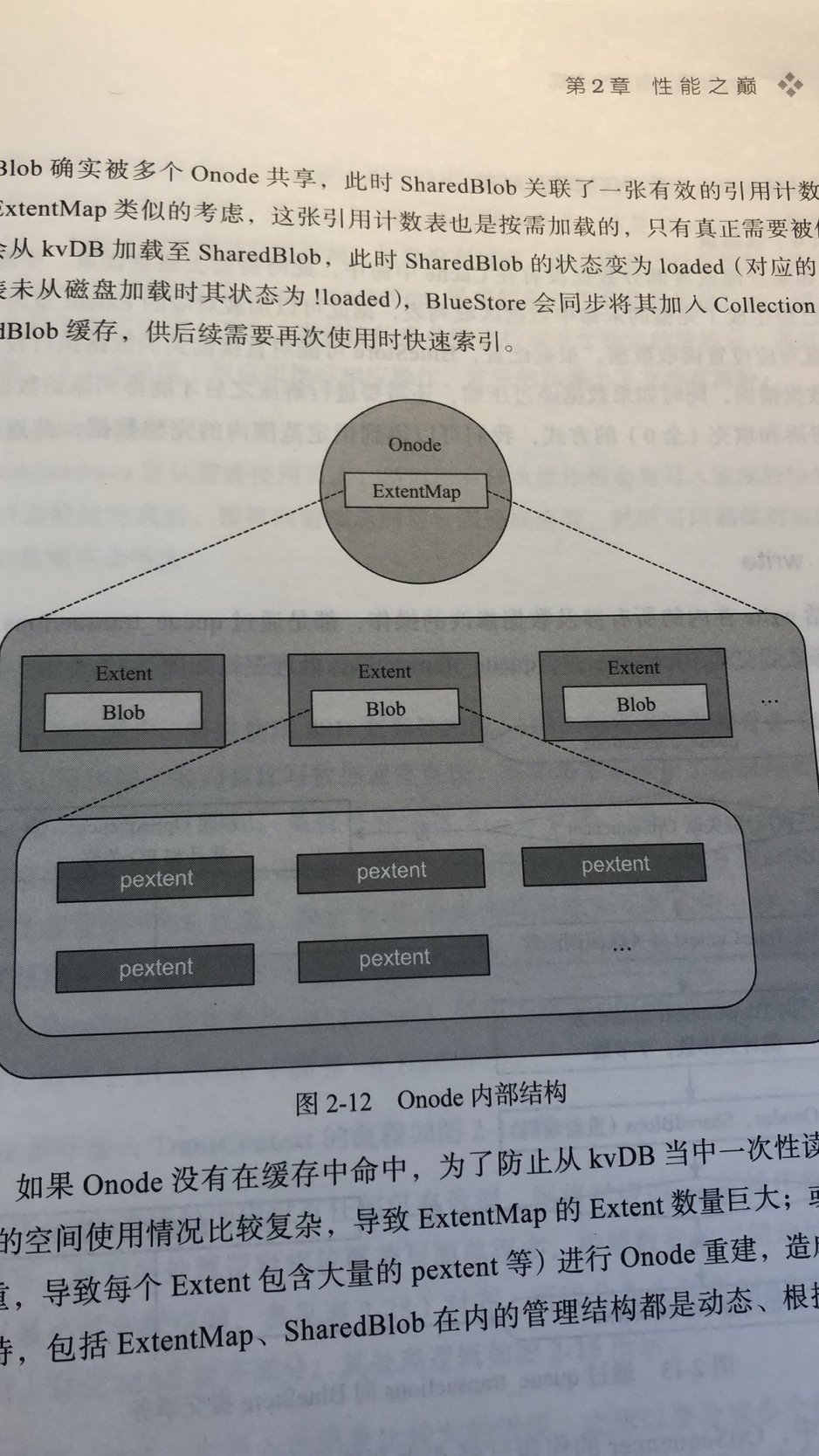
**Onode**

Onode是Bluestore对对象在内存中的描述.需要注意的是,这里所描述的对象和用户操作的对象不是完全对等的,用户操作的对象含义比较多,除了用OID进行全局唯一标识之外,还有命名空间,应用绑定信息等等其它和存储本身不相关的东西,所以一般对象到Onode是会进行转义的,主要是建立一个映射关系,这个关系是一个Key-value,key就是编码后的key,value就是Onode,其中key的编码主要做一些字符串转义的事情.

下边是Onode的定义:

1. /// an in-memory object
2. struct Onode
3. {
4. MEMPOOL\_CLASS\_HELPERS();
6. std::atomic\_int nref; ///< reference count
7. Collection \*c;
9. ghobject\_t oid;
11. /// key under PREFIX\_OBJ where we are stored
12. mempool::bluestore\_cache\_other::string key;
14. boost::intrusive::list\_member\_hook<> lru\_item;
16. bluestore\_onode\_t onode; ///< metadata stored as value in kv store
17. bool exists; ///< true if object logically exists
19. ExtentMap extent\_map;
20. ...
21. }





Onode有了,但是上层对象和Onode的对应关系也需要保存,所以用omap进行保存.

**bluestore\_onode\_t**

1. /// onode: per-object metadata
2. struct bluestore\_onode\_t {
3. uint64\_t nid = 0; ///< numeric id (locally unique),逻辑标识,单个bluestore内唯一,主要用来保证对象构建关联的omap索引时的唯一性
4. uint64\_t size = 0; ///< object size
5. map<mempool::bluestore\_cache\_other::string, bufferptr> attrs; ///< attrs
7. struct shard\_info {
8. uint32\_t offset = 0; ///< logical offset for start of shard
9. uint32\_t bytes = 0; ///< encoded bytes
10. DENC(shard\_info, v, p) {
11. denc\_varint(v.offset, p);
12. denc\_varint(v.bytes, p);
13. }
14. void dump(Formatter \*f) const;
15. };
16. vector<shard\_info> extent\_map\_shards; ///< extent map shards (if any),对象关联的extet map的分片概要信息,用于从kvDB中索引某个对象.
17. uint32\_t expected\_object\_size = 0;
18. uint32\_t expected\_write\_size = 0;
19. uint32\_t alloc\_hint\_flags = 0;
20. uint8\_t flags = 0;
21. enum {
22. FLAG\_OMAP = 1,
23. };
24. ...
25. }

**BlobMap**

**bluestore\_blob\_t**

1. /// blob: a piece of data on disk
2. struct bluestore\_blob\_t {
3. private:
4. PExtentVector extents; ///< raw data position on device
5. uint32\_t logical\_length = 0; ///< original length of data stored in the blob
6. uint32\_t compressed\_length = 0; ///< compressed length if any
7. public:
8. enum {
9. LEGACY\_FLAG\_MUTABLE = 1, ///< [legacy] blob can be overwritten or split
10. FLAG\_COMPRESSED = 2, ///< blob is compressed
11. FLAG\_CSUM = 4, ///< blob has checksums
12. FLAG\_HAS\_UNUSED = 8, ///< blob has unused map
13. FLAG\_SHARED = 16, ///< blob is shared; see external SharedBlob
14. };
15. uint32\_t flags = 0; ///< FLAG\_\*
16. unused\_t unused = 0; ///< portion that has never been written to (bitmap)
17. uint8\_t csum\_type = Checksummer::CSUM\_NONE; ///< CSUM\_\*
18. uint8\_t csum\_chunk\_order = 0; ///< csum block size is 1<<block\_order bytes
20. bufferptr csum\_data; ///< opaque vector of csum data
22. }

一般为了限制校验和的长度,避免存储的kv过大(value过大或者key过长都会导致)都会对blob大小进行限制.

**bluestore\_pextent\_t**

保存数据在磁盘上的偏移和长度.

**OpSequencer**

PG的事务队列,用来进行保序.主要包含两个FIFO队列,分别用于写请求的不同阶段,一种是普通的写,write\_big这种,一种是需要先写日志系统,然后再覆盖写硬盘中的数据.

PG

**UML**