Bluestore代码分析

1. BlueStore的需求及接口

BlueStore是ceph单机存储引擎的最新实现，继承实现了抽象类ObjectStore，BlueStore的需求及接口与ObjectStore是完全一致的。

- ObjectStore的接口见src/os/ObjectStore.h，核心接口是存储和读取一个对象的三要素：对象数据，基本属性xattrs 和omap，其中对于对象数据的读写可以只针对对象的一部分。同时，ObjectStore需要让这些操作满足ACID中的原子性和持久性（没有一致性约束需要满足，隔离性需要调用者自己保证），后文将这些操作称为事务。
- ObjectStore通过queue_transactions接口让用户传入需要执行的事务，该接口的函数原型为：

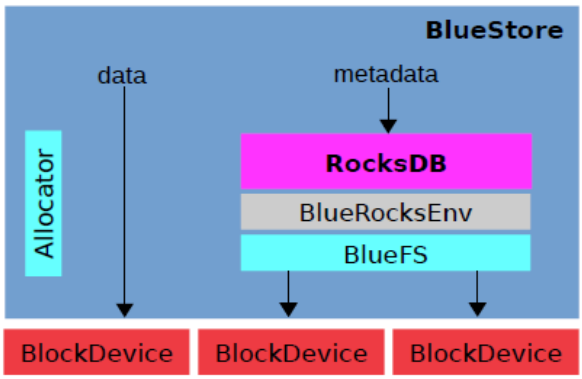
```
virtual int queue_transactions（Sequencer *osr, vector<Transaction>& tls,TrackedOpRef op = TrackedOpRef(),ThreadPool::TPHandle *handle = NULL) = 0;
```

其中第二个参数为要执行的事务队列，ObjectStore需要保证从用户视角来看，所有事务按顺序实现。每个Transaction 对象中，除了操作类型和操作数据，还提供了完成commit和apply时分别需要执行的回调函数。事务完成commit意味着数据完成落盘，即便系统此时崩溃，重启时也能重放事务，apply意味着事务对数据的更改可以被用户可见。Sequencer类提供flush接口，用户调用flush接口可以阻塞自己直到所有事务完成持久化。

- ObjectStore的实现最早是FileStore，这种实现为了实现事务的原子性需要先写一份日志完成commit，再把日志中的数据搬到可被读取的位置，写入一份数据，需要两倍的磁盘I/O，为了解决这个问题，更好地优化性能，ceph社区实现了BlueStore。

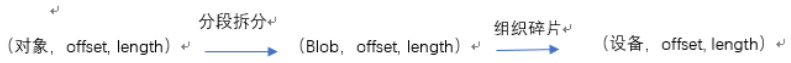
2. BlueStore的基本原理

BlueStore跳过文件系统层直接对裸设备进行管理，架构方式如下图所示（图片来源于互联网）



由图可见，对象数据直接写入裸盘，记录对象逻辑空间到设备物理空间映射的元数据，和对象的元数据xattr,omap统一由 KV数据库RocksDB保证其持久化。

- 由于RocksDB对设备是通过文件系统接口访问的，因此BlueStore专门为RocksDB提供了一个极简的文件系统BlueFS。对象数据在逻辑地址空间中以Blob（默认大小为64KB）为单位进行管理。每个Blob内部记录了从逻辑地址到物理地址的映射，一个Blob的逻辑空间可以由若干物理空间上的碎片拼接而成。在对对象数据进行读写时，地址的映射转换如下图所示：



- 在读取对象数据时，只需要按此流程完成数据的映射即可从设备的对应位置读取数据。在写对象数据时，对于新建的Block和重头到尾都需要被重写的Blob，直接从设备上分配新的Blob并对其进行写入，再将对元数据的改动写入RocksDB。对于部分Blob的覆盖写，BlueStore将操作数据和元数据改动写入RocksDB中，即可向用户返回写入成功。之后再在后台异步地将数据搬入对应Blob中。由于RocksDB提供了对于多个KV对写入的原子性保证，将上述流程中对于RocksDB的写入封装成一个事务，就可以实现整个流程的原子化。
- 另外值得一提的是，BlueStore在底层对磁盘的访问上结合SPDK（Stroage Performance Develoment Kit）使用了异步I/O，可以在对Blob发起I/O请求后直接处理下一个请求，I/O完成后由Linux aio库调用回调函数处理后面的流程。异步I/O可以更好地发挥硬件性能，尤其是针对SSD这样的高速设备，但由于I/O返回的顺序跟请求的顺序可能不一样，BlueStore需要对操作的序列进行保证，以满足接口语义。

3. BlueStore对于Linux Aio的封装

- BlueStore的异步I/O底层最终调用的是linux的libaio库，该库对于异步I/O的调用主要分为以下几步：
  - 首先，调用在初始化时调用io_setup为异步I/O请求构造上下文环境。
  - 然后，对于每个异步I/O请求，先通过io_prep_pwritev和io_prep_pread构造iocb结构体，该结构体记录了读写的buffer，位置和回调函数，然后调用io_submit接口真正发起异步I/O操作。

- 最后通过io_getevents接口检查是否有完成的异步I/O，如果有的话可以获取返回值，并通过aio_callback接口调用该异步I/O的回调函数。上述过程皆为libaio库自带接口。

- BlueStore对libaio库的封装实现在BlockDevice抽象类中。使用该抽象类进行异步I/O时，首先调用该类的aio_read或aio_write接口，在调用时需要传入一个IOContext类型的指针，aio_read和aio_write将该次I/O操作计入IOContext中，随后调用aio_submit(IOContext *ioc)将该IOContext中的I/O操作真正发出，完成之后会调用IOContext中的回调函数。BlockDevice有多个实现，在默认采用的实现KernelDevice中，aio_read或aio_write实际最终调用前文所述的libaio库中的io_prep_pwritev和io_prep_pread接口并将生成的iocb结构存入IOContext中，在aio_submit中批量为IOContext中的操作调用libaio库中的io_submit接口。另外，KernelDevice会启动一个线程_aio_thread，定期调用io_getevents并检查是否有完成的I/O，有的话修改IOContext的完成计数，如果全部完成则调用该I/O所属的IOContext的回调函数
- 有了BlockDevice的抽象，BlueStore的上层就能轻松的使用异步I/O。在使用时，在执行一个事务时，需要I/O的位置先调用aio_read和aio_write接口，所有需要的I/O记录后再统一调用io_submit发送所有的I/O请求。最后等待KernelDevice的回调线程调用自己注册的回调函数即可。

4. BlueStore对于块设备地址空间的管理

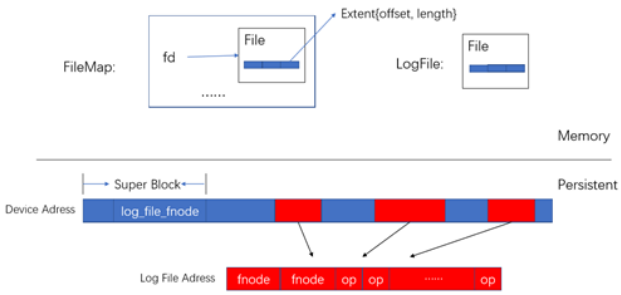
- BlueStore提供抽象类Allocator对块设备的物理空间进行管理。Allocator提供的最重要的接口，也就是获得物理地址空间的接口为：

```
virtual int64_t allocate(uint64_t want_size, uint64_t alloc_unit,
        uint64_t max_alloc_size, int64_t hint,
        AllocExtentVector *extents)
```

- 该接口的第一个参数want_size为需要申请的物理空间的大小，最后一个参数extents是一个物理空间碎片的地址，每个extent作为一个碎片主要记录了一个设备物理空间的偏移量和长度，这些碎片拼接起来就是此次allocate返回的物理地址空间。alloc_unit为分配的空间需要进行对齐的单位，max_alloc_size为调用者希望的单个extent的空间最大值，hint为希望申请的地址起始点。用户通过该接口可以获得可用的物理地址空间，释放空间时调用release(uint64_t offset, uint64_t length)就可以将物理地址空间返回给Allocator。需要注意的是，Allocator是一个无状态的组件，不负责自身信息的持久化，分配信息的持久化应该由Allocator的使用者来完成。
- BlueStore默认使用的Allocator实现是StupidAllocator。StupidAllocator的核心思想类似于内存分配中的伙伴系统，将剩余（free）的空间按照2的次幂分成若干组，每组内部用B+树组织起来。在allocate函数中，StupidAllocator调用allocate_int函数申请一块不超过want_size的最大单块空间碎片，插入extents数组中，再对want_size还没申请到的部分继续调用allocate_int直到申请到足够大小的空间。而release的实现就是把要释放的空间拆成2的整数次幂大小放回对应的btree即可。
- Allocator的调用者有两个，第一个是在写对象的数据时，在alloc_write_data函数中使用，第二个是对接RocksDB的BlueFS的使用。用于存储对象数据部分的物理地址空间，全部存储在Blob对象中并持久在RocksDB中，每次启动BlueStore时需要从RocksDB中读取所有Blob已经分配得物理空间，并调用Alloctor的init_rm_free接口通知Alloctor这部分空间已经使用。存储对象数据的元数据结构将在后文详细讲述。

5. BlueFS的实现

- BlueFS是一个实现及其简单地文件系统。在内存中，和内核VFS的设计类似，每个文件在以一个File对象将存储文件的元数据。File对象最重要的是一个extent数组，用来记录从该文件空间到设备物理空间的映射。整个BlusFS在内存中使用一个FileMap来组织所有的File对象。
- BlueFS通过下述策略来完成元数据的持久化。所有文件的元数据信息都存在文件系统的logfile中。所有文件的元数据变更，包括创建文件，删除文件，为文件申请更多空间，都以操作（op）的形式写入logfile中，当logfile太大时，就会生成一个类似当前快照的结构，在这个结构中，每个File的元数据以fnode的形式进行持久化，类似内核VFS的inode，而log_file的元数据，也就是log_file的fnode，则持久化存入文件系统的superblock中。这样，在BlusFS初始化也就是mount时，只需要先从superblock中读取logfile的元数据，通过它读取logfile中记录的的快照，再重放后面以日志形式追加的元数据操作，就可以还原内存中的文件系统元数据。其原理可由下图所示。



6. BlueStore对象元数据管理

- BlueStore中最上层的数据结构为Collection，Collection的抽象语义为一组对象的结合，OSD在使用ObjectStore时通常把一个PG作为一个Collection，该PG中的所有对象都存入这个Collection中。BlueStore中实现的Collection主要包括以下几个核心数据结构，OnodeSpace对象中存储了每个对象的映射信息，SharedBlobSet对象中记载了所有Blob的元数据信息，另外BlueStore对象指针和Cache对象指针指向所有

Collection共享的BlueStore和Cache对象，通过BlueStore对象可以直接访问共用的RocksDB,通过Cache可以获得所有I/O操作需要的缓冲区。

- OnodeSpace主要存储的就是一个从对象id映射到该对象的Onode的map。Onode存储了一个对象的元数据。Onode中除了记录对象id等关键元数据之外，主要成员是ExtentMap，其实就是一个Extent的set，Extent对象是Onode中主要记录映射信息的结构。每个Extent中记录了自己在对象逻辑地址空间的logical_offset，在对应Blob中的blob_offset，指向Blob对象的指针，以及数据的长度。
- 每个Blob对象首先通过PExtentVector记录了自己的逻辑空间由哪些物理空间上的碎片组成，随后指向一个ShareBlob结构，该结构的作用是建立Blob和BlueStore的BufferCahe之间的联系。每次Blob进行非direct的读写时，都会通过该结构访问Cache。

7. BlueStore中的读流程

- BlueStore的读操作接口继承了ObjctStore, 用户传入Collection id 和object id，以及要读的偏移量及长度，BlueStore需要把读取到的结果返回到用户给的buffer中。BlueStore先从coll_map根据Collection id获得Collection对象，再从Collection对象获得Onode对象，随后将两个id替换成具体对象之后进入读操作的核心函数_do_read。
- 进入_do_read后，先通过o->extent_map.seek_lextent(offset)找到读取范围上的第一个extent，随后逐个extent读取其中需要的部分，放入记录结果的数据结构ready_regions中，并向后遍历extent_map，直到超出了读取的范围。
- 对于每一个extent，先在buffer_cache中查找它对应的Blob的需要读取的区域是否在缓存中，先将已经缓存的部分读取到变量cache_res中，随后把剩下的部分放入变量blobs2read中。blobs2read是一个map,key是Blob对象指针，value是一个记录了这个Blob上需要被读的区域的list。这样遍历完extent_map之后，命中缓存的部分已经被放到了ready_regions中，需要I/O读取的部分记录在了blobs2read中。
- 接下来将blobs2read转换成具体的I/O操作，首先生成一个IOContext，接下来依次对blobs2read中的元素调用bdev->read或bdev->aio_read，如果是压缩的Blob,只能整个Blob的数据都读出来，如果没有压缩，可以只读取对应的部分对齐之后的范围。如果需要读取的部分大于1，则调用异步读取，否则同步读取。Blob将自己的逻辑地址转换为设备物理地址的实现比较优雅，这里介绍一下。地址的转换实际上包括在Blob对象的map接口中。该接口的函数定义为int map(uint64_t x_off, uint64_t x_len, std::function<int(uint64_t,uint64_t)> f)，前两个参数是需要操作的对象地址空间的其实偏移量和长度，第三个参数是一个闭包，该闭包指的是对于设备地址空间上的一段空间应该进行怎样的操作。在Blob对象的map函数的实现中，Blob对象会遍历自己的extents，将逻辑空间上的地址转换成为物理空间上的地址，并以物理地址片段的偏移量和长度作为参数，调用map函数传入的闭包。这样map函数的调用者就不需要关心从Blob逻辑地址空间到物理地址空间的映射的细节，只需要传入处理的逻辑地址范围，和处理的方法即可。按上述过程遍历完之后blobs2read，同步读I/O已经完成，异步读I/O记录在IOContext中。
- 因此，接下来BlueStore会调用aio_submit接口，然后调用aio_wait()等待返回结果。I/O结果返回之后，如果该Blob被压缩，则调用_decompress进行解压。最后，读到的结果被buffer_cache的did_read接口写到buffe_cahe中，然后将所有读到的数据放到bufferlist中并返回。

8. BlueStore中事务的执行

- 对于需要改变存储状态的请求，都需要从事务执行的入口queue_transactions进入，该函数进入后，首先为传入的事务数组生成一个OpSequencer，该结构是对传入的Sequencer指针的参数的封装，该结构保证共用一个Sequencer的事务在语义上能按顺序被执行，尽管queue_transactions的调用者来自不同的线程，以及实际上IO可能会异步的执行。该函数接下来对于传入的每一个事务，调用_txc_add_transaction将它们merge进一个TransContext结构中。TransContext中记录了所有对onede的改动，所有新生成的blob，以及记录所有I/O操作的I/O Context。对于不同类型的操作，_txc_add_transaction会执行不一样的逻辑，下文将介绍读写流程在这里执行的逻辑。总而言之，_txc_add_transaction执行完之后，事务需要对元数据的改变就已经完成，需要执行的I/O也已经记录在缓冲区等待执行，剩下的工作仅仅只是讲TransContext中记录的信息持久化而已。因此，从_txc_add_transaction函数之后，所有类型的操作都只需要执行一样的逻辑即可。下文所述的事务，指的即是该TransContext包括的操作内容。
- TransContext生成之后，会调用_txc_write_nodes函数，该函数遍历TransContext中记录的对元数据做的改动，并将这些元数据做的改动生成一个KeyValue数据库的事务，将这个事务也存在TransContext中。随后TransContext被检查是否有defer的需要，如果有则需要为defer生成一个key-value对写入kv数据库中。所谓defer，指的是前文提到过的先将改动写入RocksDB，再异步地将数据搬回设备地址空间的机制。完成这些之后，就会调用_txc_state_proc(txc)，进入复杂的事务处理状态机。

_txc_state_proc(txc)状态机依次分为STATE_PREPARE，STATE_AIO_WAIT，STATE_IO_DONE，STATE_KV_QUEUED，STATE_KV_SUBMITTED，STATE_KV_DONE，STATE_DEFERRED_QUEUED，STATE_DEFERRED_CLEANUP，STATE_DEFERRED_DONE，STATE_FINISHING，STATE_DONE，STATE_DEFERRED_QUEUED，STATE_DEFERRED_CLEANUP，STATE_DEFERRED_DONE，STATE_FINISHING，STATE_DONE。下面依次进行简单介绍。

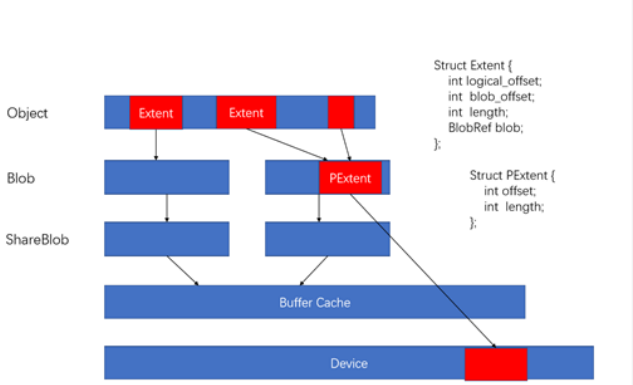
- 从queue_transactions进入状态机后首先进入的状态是STATE_PREPARE，在该状态中，状态机会检查IOContext中是否还有没有发出的I/O，如果有的话则调用_txc_aio_submit将I/O发出，并将状态置为STATE_AIO_WAIT后返回，也就是在本线程中退出了状态机。
- 随后，KernelDevice启动的线程_aio_thread会检查回调完成的请求，如果发现一个IOContext中的aio请求全部完成，则会从注册的aio_cb函数入口，一路回调aio_finish,最后重新调用_txc_state_proc(txc)回到状态机。回到状态机后，会调用_txc_finish_io(txc)函数，该函数首先将该事务的状态推进到STATE_IO_DONE，然后会检查事务所属的OpSequencer中记录的事务队列，如果在队列上自己之前的的事务还有没

有完成I/O的，则自己暂时也不能继续推进，也就是暂时被阻塞，直接返回。如果自己没有被阻塞，则会检查队列上排在自己后面的事务有没有因自己而阻塞的。最后为自己和被自己阻塞的事务依次调用_txc_state_proc回到状态机的STATE_IO_DONE阶段。

- 回到状态机，此时事务的I/O已经处理完成，接下来需要处理的是KV的持久化。如果配置项bluestore_sync_submit_transaction选择的是同步完成KV持久化，并且此时没有之前的线程在等待KV持久化，则此时会同步调用db->submit_transaction，完成后将状态推进到STATE_KV_SUBMITTED，否则状态停留在STATE_KV_QUEUED。不论事务处于哪个状态，都将事务放入到kv_queue中，返回退出状态机。
- kv_queue的处理发生在_kv_sync_thread线程中，_kv_sync_thread线程入口函数会为kv_queue中的事务调用db->submit_transaction，并通过信号量通知其他因KV持久化而阻塞的线程，随后将这些事务放入kv_committing_to_finalize队列中，至此就完成了本线程的任务。_kv_finalize_thread线程将kv_committing_to_finalize队列中的事务状态标记为STATE_KV_SUBMITTED，再次进入_txc_state_proc。
- 以STATE_KV_SUBMITTED状态进入_txc_state_proc后，会调用_txc_committed_kv函数把事务注册的oncommit，onreadable，oncommits回调函数插入finishers队列中，等待finishers线程调用queue_transactions调用者传入的回调函数。然后把状态标记为STATE_KV_DONE，并马上进入状态机的这一状态。在STATE_KV_DONE状态中，如果发现该事务有defer需求，则将状态置为STATE_DEFERRED_QUEUED，插入deferred_queue队列并返回退出。否则直接进入STATE_FINISHING状态。

- 整个BlueStore对象中维护了一个deferred_queue队列，队列中排队的单位是OpSequencer，而每个OpSequencer对象内又有两个DeferBatch类型的数据成员——defer_pending和defer_runing。DeferBatch是一个类似TransContext的结构，有自己的IOConctext和记录需要defer数据的io_map。简而言之，OpSequencer把自己管理的多个事务的defer任务聚集到了一起放在一个DeferredBatch中。deferred_try_submit函数会将defer_queen中的事务提交I/O。与TransContext不同，DeferredBatch为自己注册的回调函数是_deferred_aio_finish，该函数将DeferredBatch中的事务状态全部记为STATE_DEFERRED_CLEANUP，并把该DeferredBatch放入deferred_done_queue。
- deferred_done_queue的处理也在前面提到的_kv_sync_thread线程中，_kv_sync_thread线程将deferred_done_queue中的defer任务的对应的KV对从RocksDB中删掉，因为defer的数据已经写到了磁盘上，不再需要从KV中读取。之后事务被放到deferred_stable_to_finalize中，在_kv_finalize_thread中再次调用_txc_state_proc回到状态机。
- 以STATE_DEFERRED_CLEANUP状态回到状态机后，直接进入STATE_FINISHING状态，调用_txc_finish函数，该函数清理释放所用资源，将状态置为STATE_DONE后退出整个流程，完成整个事务的执行。

上述过程非常复杂，为此下图梳理了整个过程中的线程模型。



图片的最上面列出了整个流程中涉及的线程，自上而下表示了每个线程需要处理状态机中的哪些流程。

9. BlueStore中写操作的执行

- 如上文所述，BlueStore的写操作需要走完整个事务执行的流程，写操作和其他事务操作的区别仅在于_txc_add_transaction这一步中会进入_do_write_data函数。写入的范围在该函数中会被min_alloc_size切分，具体来说会被切成三段，中间那段的开始和结束都可以和min_alloc_size对齐，剩下的部分就是头尾两部分。随后对头尾两部分调用_do_write_small，对中间那部分调用_do_write_big。这样区分的原因在于，对齐min_alloc_size的部分可以分配新的空间，不能对齐的部分可能会产生覆盖写，也就是有可能触发defer。
- 在_do_write_big函数中，先调用extent_map.punch_hole接口先将现在在extent_map中和要写的区间的重叠区间置为无效。然后遍历要写的范围内的所有extent，通过can_reuse_blob检查这个blob是否可以复用，如果不能复用，则新建一个Blob对象，这些信息都写入一个WriteContext对象中。_do_write_small函数的实现看起来很复杂，其实流程和big大同小异，只是增加了最后一种判断情况，即必须覆盖某个extent的时候，需要产生一个defer事务。WriteContext结构体主要记录了一个old_extent_map_t——表示需要defer处理的部分，和一个write_item的数组，每个write_item记录了对某一个Blob要写入的位置和数据。
- 在_do_write_big和_do_write_small完成了对WriteContext的填写之后，_do_alloc_write函数被调用，该函数逐个处理WriteContext中的每一个write_item，调用前文提到的allocate接口为新的要写入的内容申请硬盘上的空间，生成IOContext。这样TransContext就填写完成。后面走完事务处理的流程即可。