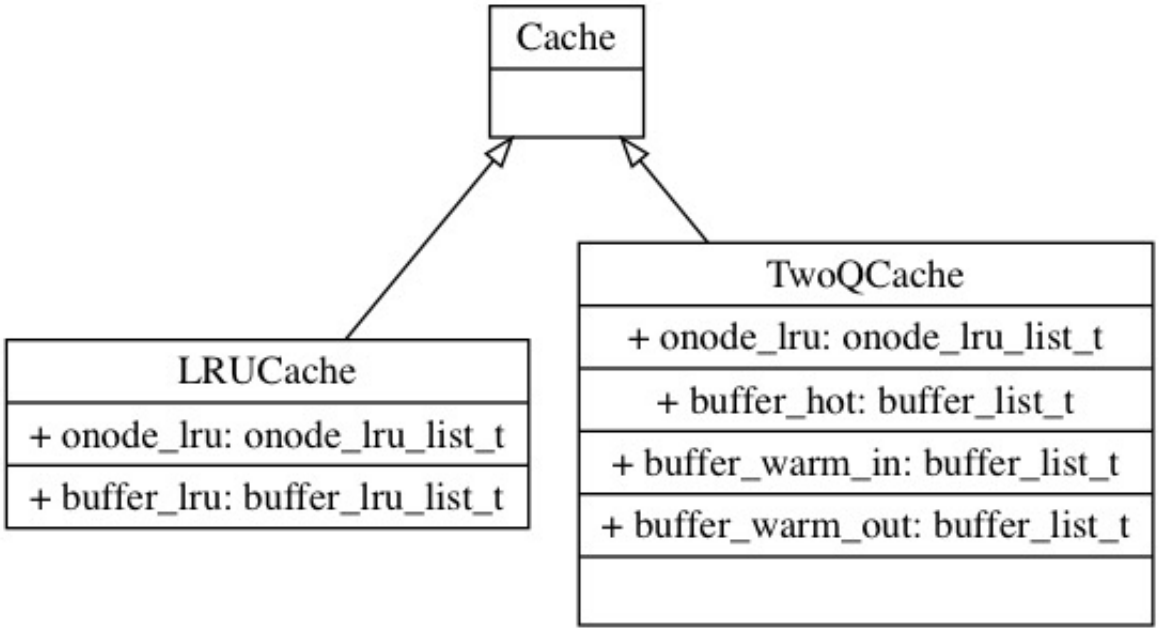
Introduction

BlueStore自己管理裸设备，没有文件系统，所以操作系统的page cache利用不上，需要自己管理缓存，包括元数据的缓存和object data的缓存，缓存系统的性能直接影响整个BlueStore的性能。

缓存的主要对象包括pg对应的Collection，即Cnode，object的元信息Onode以及object data对应的Buffer等。osd负责的pg个数比较少，在osd启动的时候，就会将所有pg对应的Collection加载到内存，而对于Onode，单个osd对应的object成千上万，条件允许可以尽最大可能将object的元信息缓存在内存中，此类缓存采用LRU策略，对于object的数据Buffer，显然是不可能全部缓存住的，尽可能缓存热点object，所以此类缓存采用的2Q算法，而不是LRU。LRU和2Q的主要区别在于，LRU仅仅采用单个链表，而2Q采用多个链表。实现的时候，抽象出Cache类，然后提供LRUCache和TwoQCache两种实现，两种实现都支持Onode和Buffer的缓存，LRUCache中Onode和Buffer各采用一个链表，淘汰用LRU策略，TwoQCache中，Onode仍然采用一个链表，用LRU策略，而buffer采用三个链表，用2Q策略。



Cache的实现本身比较简单，本质上就是将数据放入链表或根据某种策略放入多个链表，重点关注怎么使用以及缓存空间的参数调优。

Cache Init

```
BlueStore::BlueStore(CephContext *cct, const string& path)
  : ObjectStore(cct, path),
    throttle_bytes(cct, "bluestore_throttle_bytes",
           cct->_conf->bluestore_throttle_bytes),
    throttle_deferred_bytes(cct, "bluestore_throttle_deferred_bytes",
               cct->_conf->bluestore_throttle_bytes +
               cct->_conf->bluestore_throttle_deferred_bytes),
    deferred_finisher(cct, "defered_finisher", "dfin"),
    kv_sync_thread(this),
    kv_finalize_thread(this),
    mempool_thread(this)
{
  _init_logger();
  cct->_conf->add_observer(this);
  set_cache_shards(1);  // 初始化cache
}
```

BlueStore::set_cache_shards()函数

```
void BlueStore::set_cache_shards(unsigned num)
{
  dout(10) << __func__ << " " << num << dendl;
  size_t old = cache_shards.size();
  assert(num >= old);
  cache_shards.resize(num);
  for (unsigned i = old; i < num; ++i) {
    cache_shards[i] = Cache::create(cct, cct->_conf->bluestore_cache_type,
                    logger); // 默认采用2Q
  }
}
```

BlueStore::Cache::create

```
BlueStore::Cache *BlueStore::Cache::create(CephContext* cct, string type,
                       PerfCounters *logger)
{
  Cache *c = nullptr;

  if (type == "lru") // LRU
    c = new LRUCache(cct);
  else if (type == "2q") // 2q
    c = new TwoQCache(cct);
  else
    assert(0 == "unrecognized cache type");

  c->logger = logger;
  return c;
}
```

看上去初始化流程比较简单，而且把分片的shard写死为1？实际上，在osd初始化的时候，会根据参数调整shard的值，隐藏的比较深，稍不注意就疏忽了：

```
int OSD::init()
{
  ..............
  store->set_cache_shards(get_num_op_shards());
  ..............

int OSD::get_num_op_shards()
{
  if (cct->_conf->osd_op_num_shards)
    return cct->_conf->osd_op_num_shards; // 默认0
  if (store_is_rotational)
    return cct->_conf->osd_op_num_shards_hdd; // 默认5
  else
    return cct->_conf->osd_op_num_shards_ssd; // 默认8
}
```

Cnode

cnode指的是pg对应collection的磁盘结构

```
/// collection metadata
struct bluestore_cnode_t {
  uint32_t bits;   // 只有一个bit表示pg的有效位，在stable_mod的时候使用
  ............
};
```

在创建pg的时候，会将pg的元信息存在kv中

```
int BlueStore::_create_collection(
  TransContext *txc,
  const coll_t &cid,
  unsigned bits,
  CollectionRef *c)
{
  ..............................
  c->reset(new Collection(this,cache_shards[cid.hash_to_shard(cache_shards.size())],cid)); // 给pg指定一个
cache
  (*c)->cnode.bits = bits;
  coll_map[cid] = *c; // collection的map
  ::encode((*c)->cnode, bl);
  txc->t->set(PREFIX_COLL, stringify(cid), bl); // 将pg信息持久化
  ..................................
}
```

在osd上电的时候，会创建collection，并从K/V加载所有collection信息，同时会制定一个cache给pg，参见函数_open_collections:

```
int BlueStore::_open_collections(int *errors)
{
  KeyValueDB::Iterator it = db->get_iterator(PREFIX_COLL);
  for (it->upper_bound(string()); it->valid(); it->next()) { // 遍历pg对应的collection k/v元信息
    coll_t cid;
    if (cid.parse(it->key())) {
      CollectionRef c(
    new Collection(
      this,
      cache_shards[cid.hash_to_shard(cache_shards.size())],
      cid)); // 创建collection
      bufferlist bl = it->value(); // 获取对应的value
      bufferlist::iterator p = bl.begin();
      try {
        ::decode(c->cnode, p); // 解码
      } catch (buffer::error& e) {
        derr << __func__ << " failed to decode cnode, key:"
             << pretty_binary_string(it->key()) << dendl;
        return -EIO;
      }
      dout(20) << __func__ << " opened " << cid << " " << c
           << " " << c->cnode << dendl;
      coll_map[cid] = c; // 更新collection map
    } else {
      derr << __func__ << " unrecognized collection " << it->key() << dendl;
      if (errors)
    (*errors)++;
    }
  }
  return 0;
}
```

从上面的内容可知，cache shard的分片数是固定的，一般也不太大，所以多个pg可能会共用同一个cache。如果内存比较充足，缓存的数据特别多，分片的大小可以适当调大，避免cache内部的链表太长。

Onode

onode是object的元信息，读写或者其他操作的时候，都需要onode信息，而object属于某个pg(collection)的，内部通过一个map记录目前缓存的对象:

```
struct Collection : public CollectionImpl {
    // cache onodes on a per-collection basis to avoid lock
    // contention.
    OnodeSpace onode_map; // object name -> onode
};

struct OnodeSpace {
    /// forward lookups
    mempool::bluestore_cache_other::unordered_map<ghobject_t,OnodeRef> onode_map; // pg内部的object
};
```

几乎所有关于object的操作入口函数，实现的时候都会首先获取onode信息:

```
BlueStore::OnodeRef BlueStore::Collection::get_onode(
  const ghobject_t& oid,
  bool create)
{
  ......................
  OnodeRef o = onode_map.lookup(oid); // 查找map
  if (o)
    return o;
  ......................
  int r = store->db->get(PREFIX_OBJ, key.c_str(), key.size(), &v); //从k/v系统中加载元信息
  ......................
  return onode_map.add(oid, o); //加入缓存
}
```

```
BlueStore::OnodeRef BlueStore::OnodeSpace::add(const ghobject_t& oid, OnodeRef o)
{
  ......................
  onode_map[oid] = o; // 加入map
  cache->_add_onode(o, 1); // 加入缓存系统
  return o;
}
```

object的数据可能对应很多buffer，通过BufferSpace统一管理:

```
/// map logical extent range (object) onto buffers
struct BufferSpace {
    mempool::bluestore_cache_other::map<uint32_t, std::unique_ptr<Buffer>>
      buffer_map; // 所有的buffer

    // we use a bare intrusive list here instead of std::map because
    // it uses less memory and we expect this to be very small (very
    // few IOs in flight to the same Blob at the same time).
    state_list_t writing;   ///< writing buffers, sorted by seq, ascending
    // 正在写的buffer
};
```

当数据写完成后，通过标志决定是否加入缓存系统：

```
void BlueStore::BufferSpace::_finish_write(Cache* cache, uint64_t seq)
{
  auto i = writing.begin();
  while (i != writing.end()) {
    ..................
    if (b->flags & Buffer::FLAG_NOCACHE) { // 直接删除
      writing.erase(i++);
      ldout(cache->cct, 20) << __func__ << " discard " << *b << dendl;
      buffer_map.erase(b->offset);
    } else {
      b->state = Buffer::STATE_CLEAN;
      writing.erase(i++);
      b->maybe_rebuild();
      b->data.reassign_to_mempool(mempool::mempool_bluestore_cache_data);
      cache->_add_buffer(b, 1, nullptr); // 加入cache中
    }
  }
  cache->_audit("finish_write end");
}
```

同理，当读取完成后，也会考虑加入缓存：

```
int BlueStore::_do_read(
  Collection *c,
  OnodeRef o,
  uint64_t offset,
  size_t length,
  bufferlist& bl,
  uint32_t op_flags,
  uint64_t retry_count)
{
  ..................
  // generally, don't buffer anything, unless the client explicitly requests it.
  // 设置是否缓存
  bool buffered = false;
  if (op_flags & CEPH_OSD_OP_FLAG_FADVISE_WILLNEED) {
    dout(20) << __func__ << " will do buffered read" << dendl;
    buffered = true;
  }
  ..............
  if (buffered) {
    bptr->shared_blob->bc.did_read(bptr->shared_blob->get_cache(), 0,
                       raw_bl);
  }
  ..................
}
```

```
void did_read(Cache* cache, uint32_t offset, bufferlist& bl) {
      std::lock_guard<std::recursive_mutex> l(cache->lock);
      Buffer *b = new Buffer(this, Buffer::STATE_CLEAN, 0, offset, bl);
      b->cache_private = _discard(cache, offset, bl.length());
      _add_buffer(cache, b, 1, nullptr); // 加入cache
    }
```

Trim

BlueStore很多元信息对象，都是通过内存池管理的，Onode和Buffer也是同样的方式，后台由线程监控内存的使用情况，超过内存规定的限制就做trim：

```
void *BlueStore::MempoolThread::entry()
{
  while (!stop) {
    // 通过参数计算出trim的目标
    ..................
    // Now Trim
    _trim_shards(interval_stats_trim); // 执行trim
    interval_stats_trim = false;

    ..................
    utime_t wait;
    wait += store->cct->_conf->bluestore_cache_trim_interval;
    cond.WaitInterval(lock, wait);  // 定时唤醒执行trim
  }
  stop = false;
  return NULL;
}
```

一些配置参数

```
// BlueStore中cache_shards的分片大小依赖的参数
osd_op_num_shards // 默认为0
osd_op_num_shards_hdd // 默认为5
osd_op_num_shards_ssd // 默认为8

bluestore_cache_trim_interval // cache trim的间隔时间，默认为0.2，可适当调大
bluestore_cache_trim_max_skip_pinned // trim cache的时候，如果遇见item是pin的，计数+1，计数超过此值后，停止做trim。默认为64
bluestore_cache_type // 默认为2q
bluestore_2q_cache_kin_ratio // in链表的占比，默认为0.5
bluestore_2q_cache_kout_ratio // out链表的占比，默认为0.5

// 缓存空间大小，需要根据物理内存大小以及osd的个数设置合理值
bluestore_cache_size // 默认为0
bluestore_cache_size_hdd // 默认为1GB
bluestore_cache_size_ssd // 默认为3G

bluestore_cache_meta_ratio // metadata占用缓存的比率，默认为0.01
bluestore_cache_kv_ratio // rocksdb database cache占用缓存的比率，默认为0.99
bluestore_cache_kv_max // rocksdb database占用缓存的上限，默认为512MB
```

总结

- OSD启动的时候，提供参数初始化BlueStore的cache分片大小，供后续pg对应的collection使用
- OSD从磁盘读取collection信息，将pg对应的collection全部加载到内存中，并分配一个负责缓存

的cache给collection

- 执行Object操作的时候，会首先读取Onode元信息并将其加入缓存管理
- 写入Object的时候，会根据标志，将Object数据的Buffer加入到缓存管理中
- Onode/Buffer等对象统一使用内存池分配，后台线程定期检查内存使用情况，并将超出的部分trim掉