ceph bluestore缓存中保存的是object的信息，包括元数据和实际数据，元数据是用LRU Cache实现的，实际数据是用TwoQ Cache实现的。

1 内存管理

bluestore定义了一些命名空间，这些命名空间有自己的变量声明，如下

src/include/mempool.h文件中

```
// Namespace mempool

#define P(x)                                        \
  namespace x {                                       \
    static const mempool::pool_index_t id = mempool::mempool_##x;    \
    template<typename v>                              \
    using pool_allocator = mempool::pool_allocator<id,v>;        \
                                                             \
    using string = std::basic_string<char,std::char_traits<char>,       \
                                     pool_allocator<char>>;           \
                                                             \
    template<typename k,typename v, typename cmp = std::less<k> >    \
    using map = std::map<k, v, cmp,                            \
             pool_allocator<std::pair<const k,v>>>;        \
                                                             \
    template<typename k,typename v, typename cmp = std::less<k> >       \
    using compact_map = compact_map<k, v, cmp,                      \
             pool_allocator<std::pair<const k,v>>>;         \
                                                             \
    template<typename k, typename cmp = std::less<k> >             \
    using compact_set = compact_set<k, cmp, pool_allocator<k>>;     \
                                                             \
    template<typename k,typename v, typename cmp = std::less<k> >    \
    using multimap = std::multimap<k,v,cmp,                    \
                  pool_allocator<std::pair<const k,    \
                               v>>>;    \
                                                             \
    template<typename k, typename cmp = std::less<k> >          \
    using set = std::set<k,cmp,pool_allocator<k>>;            \
                                                             \
    template<typename v>                              \
    using list = std::list<v,pool_allocator<v>>;            \
                                                             \
    template<typename v>                              \
    using vector = std::vector<v,pool_allocator<v>>;            \
                                                             \
    template<typename k, typename v,                    \
         typename h=std::hash<k>,                 \
         typename eq = std::equal_to<k>>                \
    using unordered_map =                           \
```

```
    std::unordered_map<k,v,h,eq,pool_allocator<std::pair<const k,v>>>;\
                                                                      \
  inline size_t allocated_bytes() {                       \
    return mempool::get_pool(id).allocated_bytes();            \
  }                                         \
  inline size_t allocated_items() {                       \
    return mempool::get_pool(id).allocated_items();            \
  }                                         \
};

DEFINE_MEMORY_POOLS_HELPER(P)

#undef P

};
```

使用的是mempool的方式管理内存，其中allocated_items和allocated_bytes返回该pool所分配空间的情况

DEFINE_MEMORY_POOLS_HELPER宏定义如下

```
// --------------------------------------------------------------
// define memory pools

#define DEFINE_MEMORY_POOLS_HELPER(f) \
  f(bloom_filter)                \
  f(bluestore_alloc)              \
  f(bluestore_cache_data)          \
  f(bluestore_cache_onode)          \
  f(bluestore_cache_other)          \
  f(bluestore_fsck)               \
  f(bluestore_txc)               \
  f(bluestore_writing_deferred)          \
  f(bluestore_writing)              \
  f(bluefs)                 \
  f(buffer_anon)              \
  f(buffer_meta)              \
  f(osd)                 \
  f(osd_mapbl)               \
  f(osd_pglog)               \
  f(osdmap)                 \
  f(osdmap_mapping)              \
  f(pgmap)                 \
  f(mds_co)                 \
  f(unittest_1)               \
  f(unittest_2)
```

这两部分宏就是定义了一些命名空间，以及和命名空间绑定的行为

2 重载new和delete运算符

```
// Use this in some particular .cc file to match each class with a
// MEMPOOL_CLASS_HELPERS().
#define MEMPOOL_DEFINE_OBJECT_FACTORY(obj,factoryname,pool)        \
  MEMPOOL_DEFINE_FACTORY(obj, factoryname, pool)            \
  void *obj::operator new(size_t size) {                \
    return mempool::pool::alloc_##factoryname.allocate(1); \
  }                                     \
  void obj::operator delete(void *p)  {                 \
    return mempool::pool::alloc_##factoryname.deallocate((obj*)p, 1);    \
  }
```

在BlueStore.cc中使用宏 MEMPOOL_DEFINE_OBJECT_FACTORY

```
// bluestore_cache_onode
MEMPOOL_DEFINE_OBJECT_FACTORY(BlueStore::Onode, bluestore_onode,
                  bluestore_cache_onode);

// bluestore_cache_other
MEMPOOL_DEFINE_OBJECT_FACTORY(BlueStore::Buffer, bluestore_buffer,
                  bluestore_cache_other);
MEMPOOL_DEFINE_OBJECT_FACTORY(BlueStore::Extent, bluestore_extent,
                  bluestore_cache_other);
MEMPOOL_DEFINE_OBJECT_FACTORY(BlueStore::Blob, bluestore_blob,
                  bluestore_cache_other);
MEMPOOL_DEFINE_OBJECT_FACTORY(BlueStore::SharedBlob, bluestore_shared_blob,
                  bluestore_cache_other);

// bluestore_txc
MEMPOOL_DEFINE_OBJECT_FACTORY(BlueStore::TransContext, bluestore_transcontext,
                  bluestore_txc);
```

下面看下bluestore_cache_onode，将其展开

> 下面的代码是根据上面的宏进行了展开

```
namespace mempool{
    namespace bluestore_cache_onode{
        pool_allocator<BlueStore::Onode> alloc_bluestore_onode = {true};
    }
}
void * BlueStore::Onode::operator new(size_t size) {
  return mempool::bluestore_cache_onode::alloc_bluestore_onode.allocate(1);
}                                     \
void BlueStore::Onode::operator delete(void *p)  {
  return mempool::bluestore_cache_onode::alloc_bluestore_onode.deallocate((BlueStore::Onode*)p, 1);
}
```

对于bluestore_cache_onode作用域，其pool_allocator是

> 这个代码也是对上面定义的宏进行展开

```
    static const mempool::pool_index_t id = mempool::mempool_bluestore_cache_onode;    \
    template<typename v>                                   \
    using pool_allocator = mempool::pool_allocator<id,v>;        \
```

```
// 这里定义了pool_allocator
pool_allocator(bool force_register=false) { // 这里是true
  init(force_register);
}

//这里定义了init
  void init(bool force_register) {
    pool = &get_pool(pool_ix); //这里的pool_ix就是bluestore_cache_onde
    if (debug_mode || force_register) {
      type = pool->get_type(typeid(T), sizeof(T));
    }
  }

//这里定义了get_pool的操作
mempool::pool_t& mempool::get_pool(mempool::pool_index_t ix)
{
  // We rely on this array being initialized before any invocation of
  // this function, even if it is called by ctors in other compilation
  // units that are being initialized before this compilation unit.
  static mempool::pool_t table[num_pools];
  return table[ix];
}

// debug_mode流程
if (debug_mode || force_register) {
      type = pool->get_type(typeid(T), sizeof(T));
}

// get_type的流程
type_t *get_type(const std::type_info& ti, size_t size) {
    std::lock_guard<std::mutex> l(lock);
    auto p = type_map.find(ti.name());
    if (p != type_map.end()) {
      return &p->second;
    }
    type_t &t = type_map[ti.name()];
    t.type_name = ti.name();
    t.item_size = size;
    return &t;
  }
```

可以看到pool的个数是num_pools，对于bluestore_cache_onode作用域就返回bluestore_cache_onode的pool

重载的new函数如下

```
根据mempool中的设计模式重载流程
return mempool::bluestore_cache_onode::alloc_bluestore_onode.allocate(1);

// allocate的流程
T* allocate(size_t n, void *p = nullptr) {
    size_t total = sizeof(T) * n;
    shard_t *shard = pool->pick_a_shard();
    shard->bytes += total;
    shard->items += n;
    if (type) {
      type->items += n;
    }
    T* r = reinterpret_cast<T*>(new char[total]);
    return r;
  }

// pick_a_shard的流程
shard_t* pick_a_shard() {
    // Dirt cheap, see:
    //   http://fossies.org/dox/glibc-2.24/pthread__self_8c_source.html
    size_t me = (size_t)pthread_self();
    size_t i = (me >> 3) & ((1 << num_shard_bits) - 1);
    return &shard[i];
}
```

整个流程主要作用就是更新pool中share的记录，然后真正申请内存

因此每次调用new Onode时，都会将申请的内存信息更新到对应pool的shard中

同理对应其它作用域效果也是一样的

3 onode缓存数据插入

数据插入到缓存是利用_buffer_cache_write函数实现的

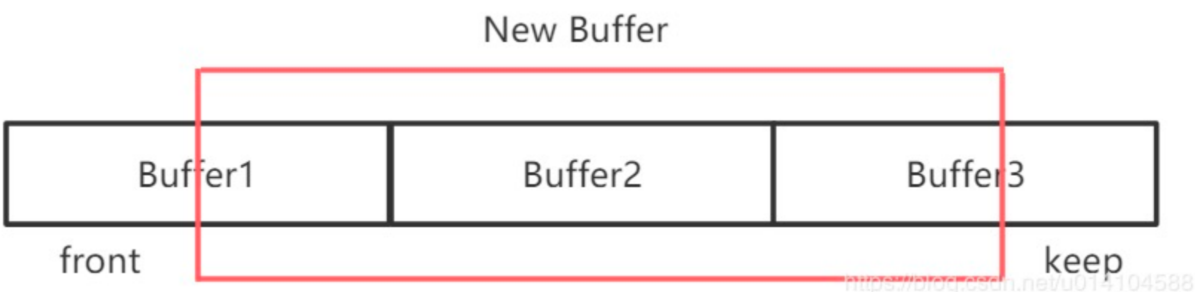
```
void _buffer_cache_write(
    TransContext *txc,
    BlobRef b,
    uint64_t offset,
    bufferlist& bl,
    unsigned flags) {

    // 流程1
    b->shared_blob->bc.write(b->shared_blob->get_cache(), txc->seq, offset, bl,
                 flags);  // shared_blob的cache关联的是bluestore中的tqcache
    // 流程2
    txc->shared_blobs_written.insert(b->shared_blob);
  }

// 对于流程1
    void write(Cache* cache, uint64_t seq, uint32_t offset, bufferlist& bl,
           unsigned flags) {
      std::lock_guard<std::recursive_mutex> l(cache->lock);
      Buffer *b = new Buffer(this, Buffer::STATE_WRITING, seq, offset, bl,
                 flags);
      b->cache_private = _discard(cache, offset, bl.length());
      _add_buffer(cache, b, (flags & Buffer::FLAG_NOCACHE) ? 0 : 1, nullptr);
    }

// _add_buffer函数
    void _add_buffer(Cache* cache, Buffer *b, int level, Buffer *near) {
      cache->_audit("_add_buffer start");
      buffer_map[b->offset].reset(b);
      if (b->is_writing()) {
    b->data.reassign_to_mempool(mempool::mempool_bluestore_writing);
        if (writing.empty() || writing.rbegin()->seq <= b->seq) {
          writing.push_back(*b);
        } else {
          auto it = writing.begin();
          while (it->seq < b->seq) {
            ++it;
          }

          assert(it->seq >= b->seq);
          // note that this will insert b before it
          // hence the order is maintained
          writing.insert(it, *b);
        }
      } else {
    b->data.reassign_to_mempool(mempool::mempool_bluestore_cache_data); // 先调整原来的，再更新现在的
    cache->_add_buffer(b, level, near);
      }
      cache->_audit("_add_buffer end");
    }
```

在_buffer_cache_write函数中会将Buffer插入到writing中，代表这个数据正在被写到磁盘，同时更新mempool_bluestore_writing对应pool的记录，同时将shared_blob插入到shared_blobs_written

当前缓存的数据和新的数据可能有重叠的区域，因此需要将重叠的部分删除，这就是同时_discard函数来实现的，如下所示：

```
int BlueStore::BufferSpace::_discard(Cache* cache, uint32_t offset, uint32_t length)
{
  // note: we already hold cache->lock
  ldout(cache->cct, 20) << __func__ << std::hex << " 0x" << offset << "~" << length
           << std::dec << dendl;
  int cache_private = 0;
  cache->_audit("discard start");
  auto i = _data_lower_bound(offset); //在buffer_map中找到包含这个offset的buffer
  uint32_t end = offset + length;
  while (i != buffer_map.end()) {
    Buffer *b = i->second.get();
    if (b->offset >= end) {
      break;
    }
    if (b->cache_private > cache_private) { //如果任何一段的缓存级别比当前级别高就提升级别
      cache_private = b->cache_private;
    }
    // 第一部分
    if (b->offset < offset) { //处理第一个重叠的buffer
      int64_t front = offset - b->offset;
      if (b->end() > end) {
    // drop middle (split)
    uint32_t tail = b->end() - end;
    if (b->data.length()) {
      bufferlist bl;
      bl.substr_of(b->data, b->length - tail, tail);
      Buffer *nb = new Buffer(this, b->state, b->seq, end, bl);
      nb->maybe_rebuild();
      _add_buffer(cache, nb, 0, b); //将新Buffer插入到2Q Cache中
    } else {
      _add_buffer(cache, new Buffer(this, b->state, b->seq, end, tail),
             0, b);
    }
    if (!b->is_writing()) {
      cache->_adjust_buffer_size(b, front - (int64_t)b->length); //仅仅更新一些数据记录
    }
    b->truncate(front); //clear b中原油的数据，从新缓存0~front范围内的数据
    b->maybe_rebuild();
    cache->_audit("discard end 1");
    break;
      } else {
    // drop tail
    if (!b->is_writing()) {
      cache->_adjust_buffer_size(b, front - (int64_t)b->length);
```

```
    }
    b->truncate(front);
    b->maybe_rebuild();
    ++i;
    continue;
      }
    }
    // 第二部分
    if (b->end() <= end) { //中间全部重叠的Buffer
      // drop entire buffer
      _rm_buffer(cache, i++);
      continue;
    }
    // drop front
    uint32_t keep = b->end() - end;
    //第三部分
    if (b->data.length()) { // 处理最后一个重叠的Buffer
      bufferlist bl;
      bl.substr_of(b->data, b->length - keep, keep);
      Buffer *nb = new Buffer(this, b->state, b->seq, end, bl);
      nb->maybe_rebuild();
      _add_buffer(cache, nb, 0, b);
    } else {
      _add_buffer(cache, new Buffer(this, b->state, b->seq, end, keep), 0, b);
    }
    _rm_buffer(cache, i);
    cache->_audit("discard end 2");
    break;
  }
  return cache_private;
}
```



(1) 第一部分代码就是处理Buffer1的情况，利用truncate函数将原来Buffer1中的数据缩短到只剩前front大小

(2) 第二部分代码就是处理Buffer2的情况，这里直接调用_rm_buffer函数将这个全部覆盖的Buffer删除，_rm_buffer函数会从对应的2Q队列中删除这个Buffer

(3) 第三部分代码就是处理Buffer3的情况，这里先常见一个keep大小的心Buffer，并插入到2Q Cache和buffer_map中，然后将Buffer3删除

在写请求的处理过程中，最后会调用_txc_state_proc，其中在最后的一个STATE_FINISHING状态会处理writing中的缓存，如下：

```
// _txc_state_proc函数主体
void BlueStore::_txc_state_proc(TransContext *txc)

// STATE_FINISHING的处理
case TransContext::STATE_FINISHING:
      txc->log_state_latency(logger, l_bluestore_state_finishing_lat);
      _txc_finish(txc);
      return;

// _txc_finish函数主体
void BlueStore::_txc_finish(TransContext *txc)
{
  dout(20) << __func__ << " " << txc << " onodes " << txc->onodes << dendl;
  assert(txc->state == TransContext::STATE_FINISHING);

  for (auto& sb : txc->shared_blobs_written) { // 这里需要重点关注
    sb->finish_write(txc->seq);
  }
  txc->shared_blobs_written.clear();

// _finish_write函数主体
void BlueStore::BufferSpace::_finish_write(Cache* cache, uint64_t seq)
{
  auto i = writing.begin();
  while (i != writing.end()) {
    if (i->seq > seq) {
      break;
    }
    if (i->seq < seq) {
      ++i;
      continue;
    }

    Buffer *b = &*i;
    assert(b->is_writing());

    if (b->flags & Buffer::FLAG_NOCACHE) {
      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);
      ldout(cache->cct, 20) << __func__ << " added " << *b << dendl;
    }
  }
```

```
  cache->_audit("finish_write end");
}

// 往2Q Cache中添加cache
void BlueStore::TwoQCache::_add_buffer(Buffer *b, int level, Buffer *near)
{
  dout(20) << __func__ << " level " << level << " near " << near
       << " on " << *b
       << " which has cache_private " << b->cache_private << dendl;
  if (near) {
    b->cache_private = near->cache_private;
    switch (b->cache_private) {
    case BUFFER_WARM_IN:
      buffer_warm_in.insert(buffer_warm_in.iterator_to(*near), *b);
      break;
    case BUFFER_WARM_OUT:
      assert(b->is_empty());
      buffer_warm_out.insert(buffer_warm_out.iterator_to(*near), *b);
      break;
    case BUFFER_HOT:
      buffer_hot.insert(buffer_hot.iterator_to(*near), *b);
      break;
    default:
      assert(0 == "bad cache_private");
    }
  } else if (b->cache_private == BUFFER_NEW) {
    b->cache_private = BUFFER_WARM_IN;
    if (level > 0) {
      buffer_warm_in.push_front(*b);
    } else {
      // take caller hint to start at the back of the warm queue
      buffer_warm_in.push_back(*b);
    }
  } else {
    // we got a hint from discard
    switch (b->cache_private) {
    case BUFFER_WARM_IN:
      // stay in warm_in.  move to front, even though 2Q doesn't actually
      // do this.
      dout(20) << __func__ << " move to front of warm " << *b << dendl;
      buffer_warm_in.push_front(*b);
      break;
    case BUFFER_WARM_OUT:
      b->cache_private = BUFFER_HOT;
      // move to hot.  fall-thru
    case BUFFER_HOT:
      dout(20) << __func__ << " move to front of hot " << *b << dendl;
      buffer_hot.push_front(*b);
      break;
    default:
      assert(0 == "bad cache_private");
```

```
      }
    }
    if (!b->is_empty()) {
      buffer_bytes += b->length;
      buffer_list_bytes[b->cache_private] += b->length;
    }
}
```

对于在_buffer_cache_write中插入的shared_blob，_txc_finish会调用每一个shared_blob的finish_write函数，这个函数的作用就是负责将数据插入到2Q缓存队列中，同时更新mempool_bluestore_cache_data对于pool的记录信息。对于新数据会先插入到buffer_warm_in队列，后面如果再次对该数据读写的话，会插入到buffer_hot中。

4 onode元数据插入缓存

```
void BlueStore::TwoQCache::_touch_onode(OnodeRef& o)
{
  auto p = onode_lru.iterator_to(*o);
  onode_lru.erase(p);
  onode_lru.push_front(*o);
}

void _add_onode(OnodeRef& o, int level) override {
      if (level > 0)
    onode_lru.push_front(*o);
      else
    onode_lru.push_back(*o);
    }
```

5 缓存trim

```
void *BlueStore::MempoolThread::entry()
{
  Mutex::Locker l(lock);

  std::list<PriorityCache::PriCache *> caches;
  caches.push_back(store->db);
  caches.push_back(&meta_cache);
  caches.push_back(&data_cache);
  autotune_cache_size = store->osd_memory_cache_min;

  utime_t next_balance = ceph_clock_now();
  utime_t next_resize = ceph_clock_now();

  bool interval_stats_trim = false;
```

```
  bool interval_stats_resize = false;
  while (!stop) {
    _adjust_cache_settings();

    // Before we trim, check and see if it's time to rebalance/resize.
    double autotune_interval = store->cache_autotune_interval;
    double resize_interval = store->osd_memory_cache_resize_interval;

    if (autotune_interval > 0 && next_balance < ceph_clock_now()) {
      // Log events at 5 instead of 20 when balance happens.
      interval_stats_resize = true;
      interval_stats_trim = true;
      if (store->cache_autotune) {
        _balance_cache(caches);
      }

      next_balance = ceph_clock_now();
      next_balance += autotune_interval;
    }
    if (resize_interval > 0 && next_resize < ceph_clock_now()) {
      if (ceph_using_tcmalloc() && store->cache_autotune) {
        _tune_cache_size(interval_stats_resize);
        interval_stats_resize = false;
      }
      next_resize = ceph_clock_now();
      next_resize += resize_interval;
    }

    // Now Trim
    _trim_shards(interval_stats_trim); //从这里正式开始
    interval_stats_trim = false;

    store->_update_cache_logger();
    utime_t wait;
    wait += store->cct->_conf->bluestore_cache_trim_interval;
    cond.WaitInterval(lock, wait);
  }
  stop = false;
  return NULL;
}
```

```
void BlueStore::MempoolThread::_trim_shards(bool interval_stats)
{
  auto cct = store->cct;
  size_t num_shards = store->cache_shards.size();

  int64_t kv_used = store->db->get_cache_usage();
  int64_t meta_used = meta_cache._get_used_bytes(); //计算元数据大小
  int64_t data_used = data_cache._get_used_bytes(); // 计算数据大小
```

```
  uint64_t cache_size = store->cache_size;
  int64_t kv_alloc =
     static_cast<int64_t>(store->db->get_cache_ratio() * cache_size);
  int64_t meta_alloc =
     static_cast<int64_t>(meta_cache.get_cache_ratio() * cache_size);
  int64_t data_alloc =
     static_cast<int64_t>(data_cache.get_cache_ratio() * cache_size);

  if (store->cache_autotune) {
    cache_size = autotune_cache_size;

    kv_alloc = store->db->get_cache_bytes();
    meta_alloc = meta_cache.get_cache_bytes();
    data_alloc = data_cache.get_cache_bytes();
  }

  if (interval_stats) {
    ldout(cct, 5) << __func__  << " cache_size: " << cache_size
                  << " kv_alloc: " << kv_alloc
                  << " kv_used: " << kv_used
                  << " meta_alloc: " << meta_alloc
                  << " meta_used: " << meta_used
                  << " data_alloc: " << data_alloc
                  << " data_used: " << data_used << dendl;
  } else {
    ldout(cct, 20) << __func__  << " cache_size: " << cache_size
                   << " kv_alloc: " << kv_alloc
                   << " kv_used: " << kv_used
                   << " meta_alloc: " << meta_alloc
                   << " meta_used: " << meta_used
                   << " data_alloc: " << data_alloc
                   << " data_used: " << data_used << dendl;
  }

  uint64_t max_shard_onodes = static_cast<uint64_t>(
      (meta_alloc / (double) num_shards) / meta_cache.get_bytes_per_onode());//获取onode的个数,
get_bytes_per_onode函数调用了_get_used_bytes和_get_num_onodes
  uint64_t max_shard_buffer = static_cast<uint64_t>(data_alloc / num_shards);

  ldout(cct, 30) << __func__ << " max_shard_onodes: " << max_shard_onodes
                 << " max_shard_buffer: " << max_shard_buffer << dendl;

  for (auto i : store->cache_shards) {
    i->trim(max_shard_onodes, max_shard_buffer); // 接着就是看trim函数的实现了
  }
}
```

```
void BlueStore::TwoQCache::_trim(uint64_t onode_max, uint64_t buffer_max)
{
  dout(20) << __func__ << " onodes " << onode_lru.size() << " / " << onode_max
```

```
    << " buffers " << buffer_bytes << " / " << buffer_max
    << dendl;

_audit("trim start");

// buffers
if (buffer_bytes > buffer_max) {
  uint64_t kin = buffer_max * cct->_conf->bluestore_2q_cache_kin_ratio;
  uint64_t khot = buffer_max - kin;

  // pre-calculate kout based on average buffer size too,
  // which is typical(the warm_in and hot lists may change later)
  uint64_t kout = 0;
  uint64_t buffer_num = buffer_hot.size() + buffer_warm_in.size();
  if (buffer_num) {
    uint64_t buffer_avg_size = buffer_bytes / buffer_num;
    assert(buffer_avg_size);
    uint64_t calculated_buffer_num = buffer_max / buffer_avg_size;
    kout = calculated_buffer_num * cct->_conf->bluestore_2q_cache_kout_ratio;
  }

  if (buffer_list_bytes[BUFFER_HOT] < khot) {
    // hot is small, give slack to warm_in
    kin += khot - buffer_list_bytes[BUFFER_HOT];
  } else if (buffer_list_bytes[BUFFER_WARM_IN] < kin) {
    // warm_in is small, give slack to hot
    khot += kin - buffer_list_bytes[BUFFER_WARM_IN];
  }

  // adjust warm_in list
  int64_t to_evict_bytes = buffer_list_bytes[BUFFER_WARM_IN] - kin;
  uint64_t evicted = 0;

  while (to_evict_bytes > 0) {
    auto p = buffer_warm_in.rbegin();
    if (p == buffer_warm_in.rend()) {
      // stop if warm_in list is now empty
      break;
    }

    Buffer *b = &*p;
    assert(b->is_clean());
    dout(20) << __func__ << " buffer_warm_in -> out " << *b << dendl;
    assert(buffer_bytes >= b->length);
    buffer_bytes -= b->length;
    assert(buffer_list_bytes[BUFFER_WARM_IN] >= b->length);
    buffer_list_bytes[BUFFER_WARM_IN] -= b->length;
    to_evict_bytes -= b->length;
    evicted += b->length;
    b->state = Buffer::STATE_EMPTY;
    b->data.clear();
```

```
      buffer_warm_in.erase(buffer_warm_in.iterator_to(*b));
      buffer_warm_out.push_front(*b);
      b->cache_private = BUFFER_WARM_OUT;
    }

    if (evicted > 0) {
      dout(20) << __func__ << " evicted " << byte_u_t(evicted)
               << " from warm_in list, done evicting warm_in buffers"
               << dendl;
    }

    // adjust hot list
    to_evict_bytes = buffer_list_bytes[BUFFER_HOT] - khot;
    evicted = 0;

    while (to_evict_bytes > 0) {
      auto p = buffer_hot.rbegin();
      if (p == buffer_hot.rend()) {
        // stop if hot list is now empty
        break;
      }

      Buffer *b = &*p;
      dout(20) << __func__ << " buffer_hot rm " << *b << dendl;
      assert(b->is_clean());
      // adjust evict size before buffer goes invalid
      to_evict_bytes -= b->length;
      evicted += b->length;
      b->space->_rm_buffer(this, b);
    }

    if (evicted > 0) {
      dout(20) << __func__ << " evicted " << byte_u_t(evicted)
               << " from hot list, done evicting hot buffers"
               << dendl;
    }

    // adjust warm out list too, if necessary
    int64_t num = buffer_warm_out.size() - kout;
    while (num-- > 0) {
      Buffer *b = &*buffer_warm_out.rbegin();
      assert(b->is_empty());
      dout(20) << __func__ << " buffer_warm_out rm " << *b << dendl;
      b->space->_rm_buffer(this, b);
    }
  }

  // onodes
  if (onode_max >= onode_lru.size()) {
    return; // don't even try
  }
```

```
    uint64_t num = onode_lru.size() - onode_max;

    auto p = onode_lru.end();
    assert(p != onode_lru.begin());
    --p;
    int skipped = 0;
    int max_skipped = g_conf->bluestore_cache_trim_max_skip_pinned;
    while (num > 0) {
      Onode *o = &*p;
      dout(20) << __func__ << " considering " << o << dendl;
      int refs = o->nref.load();
      if (refs > 1) {
        dout(20) << __func__ << "  " << o->oid << " has " << refs
             << " refs; skipping" << dendl;
        if (++skipped >= max_skipped) {
          dout(20) << __func__ << " maximum skip pinned reached; stopping with "
                   << num << " left to trim" << dendl;
          break;
        }

        if (p == onode_lru.begin()) {
          break;
        } else {
          p--;
          num--;
          continue;
        }
      }
      dout(30) << __func__ << " " << o->oid << " num=" << num <<" lru size="<<onode_lru.size()<< dendl;
      if (p != onode_lru.begin()) {
        onode_lru.erase(p--);
      } else {
        onode_lru.erase(p);
        assert(num == 1);
      }
      o->get();  // paranoia
      o->c->onode_map.remove(o->oid);
      o->put();
      --num;
    }
  }
```

总结一下上面的代码:

(1) 计算缓存中onode的个数，这是通过mempool::bluestore_cache_onode::allocated_items()实现的， 同时计算元数据的大小是通过mempool::bluestore_cache_other::allocated_bytes()和mempool::bluestore_cache_onode::allocated_bytes()实现的

(2) 计算每一恶搞shard的期望大小和每个onode的平均元数据大小

(3) 对于每一个shard，调用BlueStore::Cache::trim计算要保留的onode的个数和最大缓存数据的大小，然后调用BlueStore::TwoQCache::_trim去做真正的删除

(4) 2Q Cache有三个缓存队列，

- bufer_bot 保存热数据

- buffer_warm_in保存最新添加的数据

- buffer_warm_out保存从buffer_warm_in中淘汰出来的数据

还有几个其它的数据结构

- buffer_max 代表了buffer_warm_in和buffer_hot之和的最大值

- buffer_2q_cache_kin_ratio代表buffer_warm_in所占buffer_max的比例

- buffer_2q_cache_kout_ratio代表了buffer_warm_out所占buffer_max的比例

(5) 从buffer_warm_in队列末尾开始删除多余的数据，删除的数据会直接加入到buffer_warm_out的头部

(6) 从buffer_hot末尾开始删除多余的数据，这里的数据会直接从buffer_hot队列中删除，同理对于buffer_warm_out的删除，其行为和buffer_hot是一样的

(7) 删除onode_lru中对于的onode，这里值得注意的是对于删除onode缓存，也要从Collection的onode_map中删除