OSDC

参考文章：

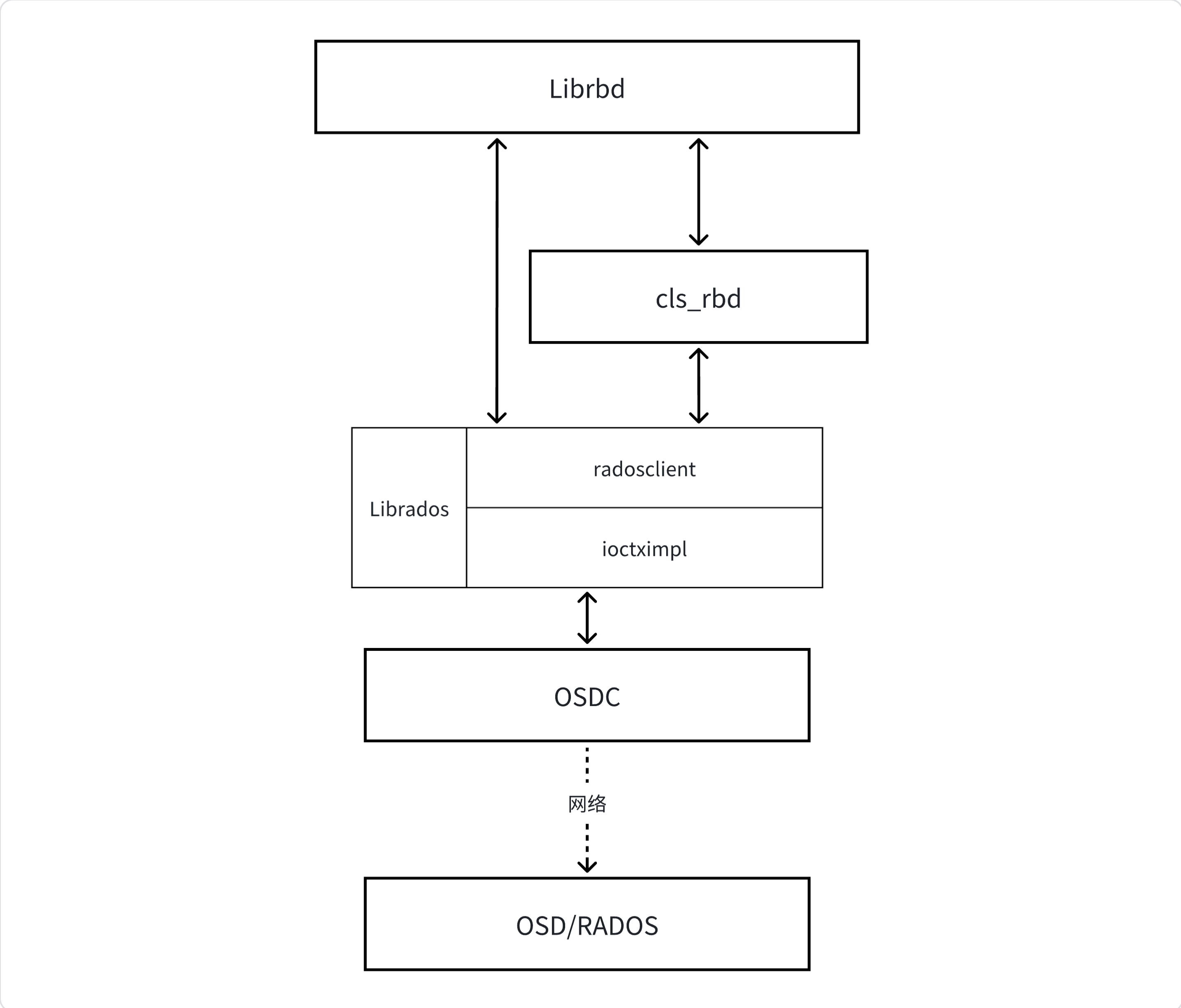
Ceph OSDC源码分析 下

Ceph OSDC源码分析 上

介绍

OSDC 是 osd client 模块的简称，主要用来和rados交互，这个模块里面完成了几个主要的功能：

1. 地址空间转换，从RBD（RADOS Block Device）的一维地址空间转换到对象的三维地址空间。
2. object_cacher：一个object级别的缓存
3. Crush算法定位OSD：转换为三维地址空间之后，使用Crush算法进行对象的数据定位。



地址空间转换

来看看什么叫做一维转三维：

1. 对于一个文件，会被分成多个对象，每个对象的空间都是连续的
2. 一个缓存区，缓存这些正在读写的数据：
3. 对象和相应数据在磁盘内的排列方式，第一维是objectset，称为对象组，文件可以有多个对象组，第二维是stripe条带号，图中有三个条带，第三维是条带内偏移，指示对象在条带内的具体位置。可以看到，条带化后地址空间在逻辑上是连续的。

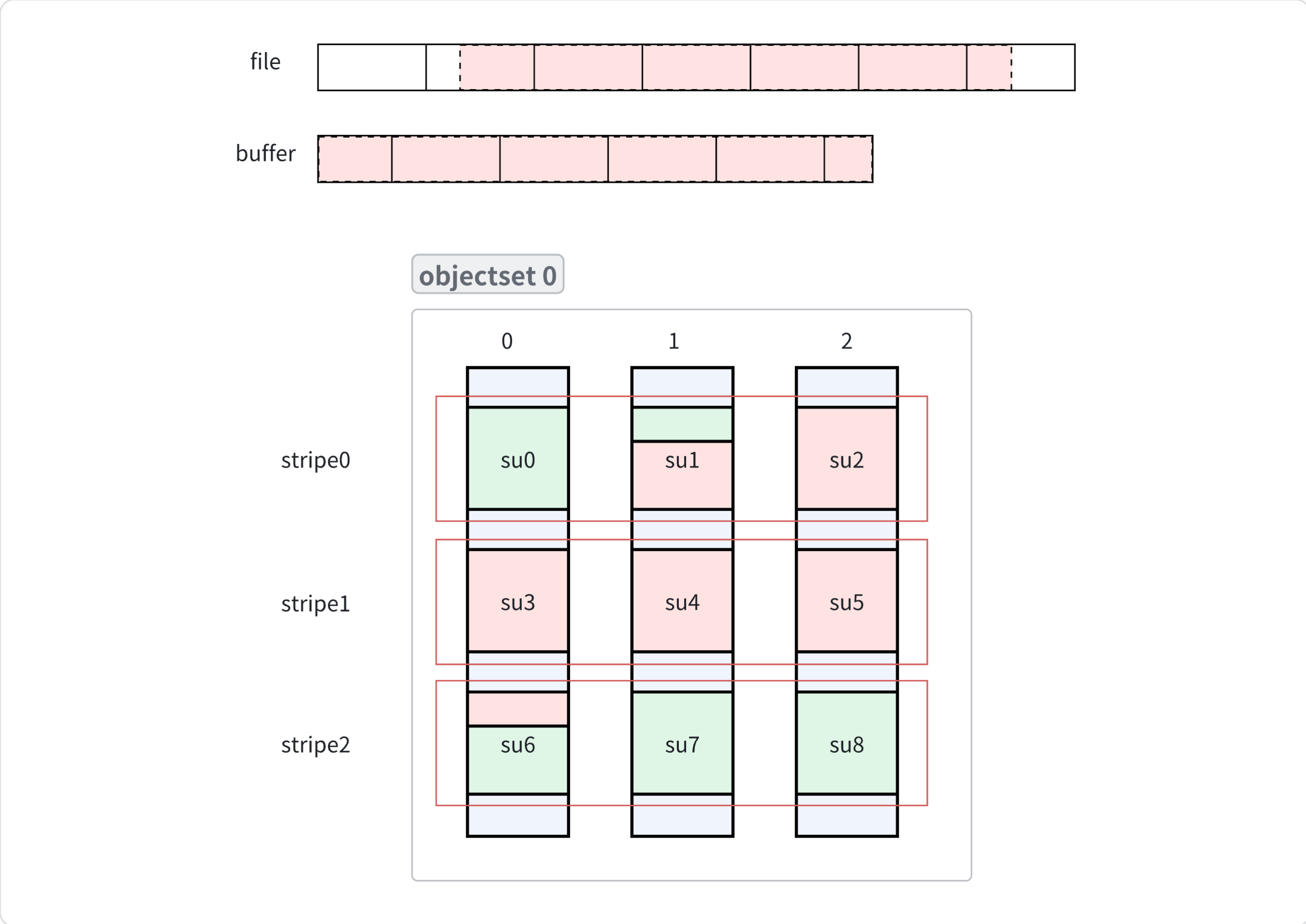


图1 地址空间转换

1. 灰色柱状代表一个rados底层对象，默认为4M
2. su是条带单元
3. 每一行是一个条带
4. object代表对象组，一般一个对象组属于同一个文件
5. object_size：对象的大小，灰色柱状部分，一般默认4M
6. stripe_count 条带宽度，就是一个stripe跨多少个对象，也就是objectset中对象的个数，上面是3个对象
7. stripes_per_object，一个对象的分片数，上图一个对象被分成了三份

file_to_extents

上图中一个个条带连起来可以看成是逻辑上连续的、相当于线性的一维空间。file_to_extent函数是计算如何将一维坐标变成三维坐标。

```
void Striper::file_to_extents(
    CephContext *cct, // Ceph 上下文指针
    const file_layout_t *layout, // 指向文件布局信息的指针
```

```
    uint64_t offset, // 文件中的起始偏移量
    uint64_t len, // 要处理的长度
    uint64_t trunc_size, // 截断大小
    uint64_t buffer_offset, // 缓冲区偏移量
    striper::LightweightObjectExtents* object_extents) // 指向对象范围集合的指针
{

  __u32 object_size = layout->object_size;        // 对象大小
  __u32 su = layout->stripe_unit;                 // 条带单元大小
  __u32 stripe_count = layout->stripe_count;      // 条带中对象的个数
  ceph_assert(object_size >= su);                 // 对象大小应大于等于条带单元大小

  // 如果只有一条条带，条带单元大小等于对象大小
  if (stripe_count == 1) {
    su = object_size;
  }
  // 每个对象中的条带数量，如果不能整除，直接舍弃余数部分
  uint64_t stripes_per_object = object_size / su;

  uint64_t cur = offset;   // 当前的偏移量
  uint64_t left = len;     // 要处理的长度
  while (left > 0) {
    // 将文件布局映射到对象
    uint64_t blockno = cur / su;  // 当前块号
    uint64_t stripeno = blockno / stripe_count;             // 条带号
    uint64_t stripepos = blockno % stripe_count;            // 条带内块的位置
    uint64_t objectsetno = stripeno / stripes_per_object; // 对象组编号
    uint64_t objectno = objectsetno * stripe_count + stripepos;  // 对象号

    // 映射块到对象中的位置
    uint64_t block_start = (stripeno % stripe_count) * su;
    uint64_t block_off = cur % su;
    uint64_t max = su - block_off; // 最大可写入长度

    uint64_t x_offset = block_start + block_off; // 对象内的偏移量
    uint64_t x_len;
    if (left > max) // 如果剩余长度大于最大可写入长度
      x_len = max;  // 设置写入长度为最大可写入长度
    else
      x_len = left; // 否则设置为剩余长度

    striper::LightweightObjectExtent* ex = nullptr;
    auto it = std::upper_bound(object_extents->begin(), object_extents->end(),
                               objectno, OrderByObject()); // 找到对象号的迭代器

    striper::LightweightObjectExtents::reverse_iterator rev_it(it);
    if (rev_it == object_extents->rend() ||              // 如果迭代器到达了末尾
```

```
        rev_it->object_no != objectno ||                 // 或者对象号不匹配
        rev_it->offset + rev_it->length != x_offset) {   // 或者偏移量和长度不匹配

      ex = &(*object_extents->emplace(
        it, objectno, x_offset, x_len,
        object_truncate_size(cct, layout, objectno, trunc_size))); // 创建并添加新对象
    } else {
      ex = &(*rev_it); // 更新现有对象
      ceph_assert(ex->offset + ex->length == x_offset); // 断言偏移量和长度匹配
      ex->length += x_len; // 增加对象长度
    }

    ex->buffer_extents.emplace_back(cur - offset + buffer_offset, x_len); // 添加缓冲区范围
    left -= x_len;  // 减少剩余长度
    cur += x_len;   // 移动到下一个块
  }
}
// 三维地址分片的信息全在object_extents中，每个条目的结构如下：
// 每个条目保存一个对象内的分片信息
struct LightweightObjectExtent {
  LightweightObjectExtent() = delete;
  LightweightObjectExtent(uint64_t object_no, uint64_t offset,
                          uint64_t length, uint64_t truncate_size)
    : object_no(object_no), offset(offset), length(length),
      truncate_size(truncate_size) {
  }

  uint64_t object_no;     // 分片序号
  uint64_t offset;        // 对象内偏移
  uint64_t length;        // 长度
  uint64_t truncate_size; // 截断。不知道截什么
  LightweightBufferExtents buffer_extents; // 在buffer内的信息
};
```

对于图一，假设文件的offset是5M，长度是20M。文件的分片信息: stripe_unit为4M，stripe_count为3（三个对象），object_size为12M。

转换后的结果就是:

```
LightweightObjectExtent[obj0] = {
    objectno = 0
    offset = 4M
    length = 5M
```

```
    buffer_extent = { [7, 4], [19, 1] }
}

LightweightObjectExtent[obj1] = {
    objectno = 1
    offset = 1M
    length = 7M
    buffer_extent = { [0, 3], [11, 4] }
}

LightweightObjectExtent[obj2] = {
    objectno = 2
    offset = 0M
    length = 8M
    buffer_extent = { [3, 4], [15, 4] }
}
每一次循环的值:
blockno: 1 stripeno: 0 stripepos: 1 objectsetno: 0 objectno: 1 block_start: 0 block_off: 1 max: 3 x_offset: 1 x_len: 3
blockno: 2 stripeno: 0 stripepos: 2 objectsetno: 0 objectno: 2 block_start: 0 block_off: 0 max: 4 x_offset: 0 x_len: 4
blockno: 3 stripeno: 1 stripepos: 0 objectsetno: 0 objectno: 0 block_start: 4 block_off: 0 max: 4 x_offset: 4 x_len: 4
blockno: 4 stripeno: 1 stripepos: 1 objectsetno: 0 objectno: 1 block_start: 4 block_off: 0 max: 4 x_offset: 4 x_len: 4
blockno: 5 stripeno: 1 stripepos: 2 objectsetno: 0 objectno: 2 block_start: 4 block_off: 0 max: 4 x_offset: 4 x_len: 4
blockno: 6 stripeno: 2 stripepos: 0 objectsetno: 0 objectno: 0 block_start: 8 block_off: 0 max: 4 x_offset: 8 x_len: 1
```

数据结构

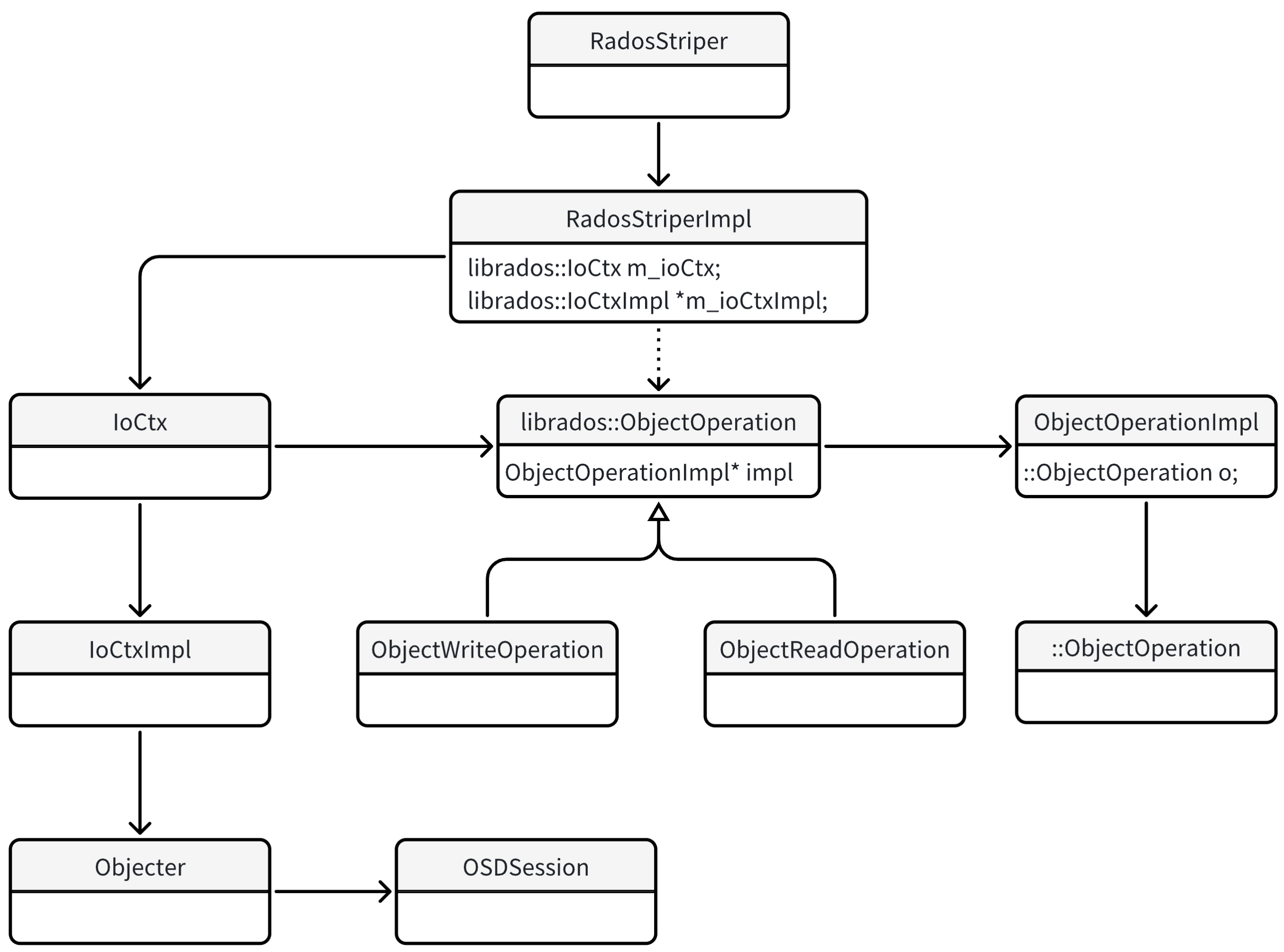
```
struct ObjectOperation {
  small_vector<OSDOp, osdc_opvec_len> ops; //操作队列
  int flags = 0;
  int priority = 0;
  small_vector<ceph::buffer::list*, osdc_opvec_len> out_bl;  // 输出缓冲区队列
  small_vector<
    fu2::unique_function<void(boost::system::error_code, int,
                              const ceph::buffer::list& bl) &&>,
    osdc_opvec_len> out_handler;                            // 回调函数队列
    small_vector<int*, osdc_opvec_len> out_rval;            // 操作结果
    boost::container::small_vector<boost::system::error_code*,
                                 osdc_opvec_len> out_ec;    // 错误代码
```

```
}

struct OSDOp {
  ceph_osd_op op;  // 各种操作码和参数
  sobject_t soid;  // 操作对象

  ceph::buffer::list indata, outdata;  // 输入输出缓冲区
  errorcode32_t rval;    // 操作结果
}
```

类图



写操作

```
// 创建IO上下文
rados_ioctx_create

rados_write_full
>> IoCtxImpl::write_full
```

```
>>>> IoCtxImpl::write
>>>>>> IoCtxImpl::operate
>>>>>>>> Objecter::Op::op_submit

Objecter::_op_submit
>> _calc_target
>> _get_session
>> _send_op
```

在 `hello_world_c.c` 中给出了使用librados库的接口的示例：

```c
#include <rados/librados.h>
#include <stdio.h>

int main() {
    rados_t rados;
    rados_ioctx_t io_ctx;
    const char *pool_name = "hello_world_pool";
    const char* hello = "hello world!";
    const char* object_name = "hello_object";

    // 创建 RADOS 集群连接
    rados_create(&rados, "admin");
    rados_conf_parse_argv(rados, 0, NULL);
    rados_connect(rados);

    // 创建池
    rados_pool_create(rados, pool_name);

    // 创建 IO 上下文
    rados_ioctx_create(rados, pool_name, &io_ctx);

    // 直接写入对象
    rados_write_full(io_ctx, object_name, hello, strlen(hello));

    // 设置扩展属性
    rados_setxattr(io_ctx, object_name, "version", "1", 2);

    // 执行复合写操作，这里设置多个属性，然后一起提交(op_operate)
    // 目的是确保原子性
    rados_write_op_t write_op = rados_create_write_op();
    rados_write_op_write_full(write_op, "v2", 4);
    rados_write_op_setxattr(write_op, "version", "2", 2);
    rados_write_op_operate(write_op, io_ctx, object_name, NULL, 0);

```

```
    // 关闭资源
    rados_ioctx_destroy(io_ctx);
    rados_shutdown(rados);
    return 0;
}
```

rados_write_full

```
#define CEPH_RADOS_API
CEPH_RADOS_API int rados_write_full(rados_ioctx_t io, const char *oid,
      const char *buf, size_t len);

#define LIBRADOS_C_API_DEFAULT_F(fn) fn
extern "C" int LIBRADOS_C_API_DEFAULT_F(rados_write_full)(
  rados_ioctx_t io,
  const char *o,
  const char *buf,
  size_t len)
{
    librados::IoCtxImpl *ctx = (librados::IoCtxImpl *)io;
    object_t oid(o);
    bufferlist bl;
    bl.append(buf, len);

    int retval = ctx->write_full(oid, bl);
    return retval;
}
```

IoCtxImpl

实现单个pool层面的对象读写操作，主要是将请求封装成ObjectOperation，然后添加pool的地址信息，并封装为Object::Op，调用op_submit发送给相应的OSD。

```
int librados::IoCtxImpl::write_full(const object_t& oid, bufferlist& bl)
{
    ::ObjectOperation op;
    op.write_full(bl);
    return operate(oid, &op, NULL);
}

::ObjectOperation
void write_full(ceph::buffer::list& bl) {
    add_data(CEPH_OSD_OP_WRITEFULL, 0, bl.length(), bl);
```

```
}

int librados::IoCtxImpl::operate(const object_t& oid, ::ObjectOperation *o,
                                 ceph::real_time *pmtime, int flags)
{
...
  Objecter::Op *objecter_op = objecter->prepare_mutate_op(
    oid, oloc, // oloc中含有pool的信息
    *o, snapc, ut,
    flags | extra_op_flags,
    oncommit, &ver);
  objecter->op_submit(objecter_op);
...
}
```

读操作

同写操作，从librados层开始调用，将刚刚写入的读出来

```
rados_aio_read(io_ctx, object_name, read_completion, read_buf, read_len, 0);
```

转而调用IoctxImpl的方法

```
int librados::IoCtxImpl::aio_read(const object_t oid, AioCompletionImpl *c,
                                  char *buf, size_t len, uint64_t off,
                                  uint64_t snapid, const blkin_trace_info *info)
{
  FUNCTRACE(client->cct);

  OID_EVENT_TRACE(oid.name.c_str(), "RADOS_READ_OP_BEGIN");
  Context *oncomplete = new C_aio_Complete(c);

  c->is_read = true;
  c->io = this;
  c->bl.clear();
  c->bl.push_back(buffer::create_static(len, buf));
  c->blp = &c->bl;
  c->out_buf = buf;

  Objecter::Op *o = objecter->prepare_read_op(
    oid, oloc,
```

```
        off, len, snapid, &c->bl, extra_op_flags,
        oncomplete, &c->objver, nullptr, 0, &trace);
    objecter->op_submit(o, &c->tid);
    return 0;
}
```

条带写

```
#include "rados/librados.hpp"
#include "radosstriper/libradosstriper.hpp"
#include <iostream>
#include <string>

int main(int argc, char* argv[])
{
    uint32_t strip_count = std::stoi(argv[1]);   // 一个条带包含的对象数量
    uint32_t obj_size = std::stoi(argv[2]);      // 对象大小
    std::string fname = argv[3];             // 文件名
    std::string obj_name = argv[4];          // 对象名
    std::string pool_name = argv[5];         // 池名称

// IO上下文, ceph客户端,条带化对象的智能指针
    librados::IoCtx io_ctx;
    librados::Rados cluster;
    libradosstriper::RadosStriper* rs = new libradosstriper::RadosStriper;

// 加载配置
    ret = cluster.init2("client.admin", "ceph", 0);
    ret = cluster.conf_read_file("ceph.conf");
// 建立连接
    ret = cluster.connect();
// 为指定的Pool创建一个IO上下文
    ret = cluster.ioctx_create(pool_name.c_str(), io_ctx);
// 创建一个RadosStriper对象, 用于条带化写入
    ret = libradosstriper::RadosStriper::striper_create(io_ctx, rs);
// 获取Pool的对齐要求
    uint64_t alignment = 0;
    ret = io_ctx.pool_required_alignment2(&alignment);

    rs->set_object_layout_stripe_unit(alignment);     // su 条带大小
    rs->set_object_layout_stripe_count(strip_count);  // 条带中对象数量
    rs->set_object_layout_object_size(obj_size);      // 对象大小

    librados::bufferlist bl;
```

```
    bl.read_file(fname.c_str());
    // 写操作
    rs->write_full(obj_name, bl);
    std::cout << "done with: " << fname << std::endl;

    delete rs;
    io_ctx.close();
    cluster.shutdown();
}
```

write_full

```
int libradosstriper::RadosStriperImpl::write_full(const std::string& soid,
                                                    const bufferlist& bl)
{
  return write(soid, bl, bl.length(), 0);
}

int libradosstriper::RadosStriperImpl::write(const std::string& soid,
      const bufferlist& bl, size_t len, uint64_t off)
{
  ceph_file_layout layout;
  std::string lockCookie;
  int rc = createAndOpenStripedObject(soid, &layout, len+off, &lockCookie,
true);
  if (rc) return rc;
  return write_in_open_object(soid, layout, lockCookie, bl, len, off);
}

int libradosstriper::RadosStriperImpl::createAndOpenStripedObject(const
std::string& soid,
                                                                  ceph_file_layout *layout,
                                                                  uint64_t size,
                                                                  std::string *lockCookie,
                                                                  bool isFileSizeAbsolute)
{
  // 创建写操作
  librados::ObjectWriteOperation writeOp;
  writeOp.create(true);

  // object_size
```

```
  std::ostringstream oss_object_size;
  oss_object_size << m_layout.fl_object_size;
  bufferlist bl_object_size;
  bl_object_size.append(oss_object_size.str());
  writeOp.setxattr(XATTR_LAYOUT_OBJECT_SIZE, bl_object_size);

  // stripe unit
  ...
  writeOp.setxattr(XATTR_LAYOUT_STRIPE_UNIT, bl_stripe_unit);

  // stripe count
  ...
  writeOp.setxattr(XATTR_LAYOUT_STRIPE_COUNT, bl_stripe_count);

  // size
  ...
  writeOp.setxattr(XATTR_SIZE, bl_size);

  // 执行原子写入操作，尝试创建对象
  std::string firstObjOid = getObjectId(soid, 0);
  int rc = m_ioCtx.operate(firstObjOid, &writeOp); // 执行操作

  // 对象已存在，则返回错误
  if (rc && -EEXIST != rc) return rc;

  // 如果没有错误或者对象已存在，继续打开条带化对象进行写入
  uint64_t fileSize = size; // 设置文件大小
  return openStripedObjectForWrite(soid, layout, &fileSize, lockCookie,
isFileSizeAbsolute);
}
```

```
int libradosstriper::RadosStriperImpl::write_in_open_object(
                          const std::string& soid, // 条带化对象的ID
                          const ceph_file_layout& layout, // 文件布局信息
                          const std::string& lockCookie, // 锁定cookie
                          const bufferlist& bl, // 要写入的数据
                          size_t len, // 要写入的数据长度
                          uint64_t off) { // 写入操作的偏移量

  ...
  // 调用内部的异步写入API，执行数据写入操作
  int rc = internal_aio_write(soid, c, bl, len, off, layout);
  if (!rc) {
```

```
    // 如果写入操作成功，等待写入操作完成
    c->wait_for_complete_and_cb();
    // 等待数据安全确认
    c->wait_for_safe_and_cb();
    // 等待解锁操作完成
    unlock_completion->wait_for_complete();
    // 获取并返回写入操作的结果
    rc = c->get_return_value();
  }
  return rc;
}
```

ObjectExtent和LightweightObjectExtent区别不大，就是多了几个字段，LightxxxExtent作为条带对象的中间变量，最终数据会转存在ObjectExtent中参与读写请求。

```
class ObjectExtent {
public:
    object_t oid; // object id
    uint64_t objectno;
    uint64_t offset;         // in object
    uint64_t length;         // in object
    uint64_t truncate_size; // in object

    object_locator_t oloc; // object locator (pool etc)

    std::vector<std::pair<uint64_t, uint64_t>> buffer_extents; // off -> len. extents in buffer being mapped (may be fragmented bc of striping!)

    ObjectExtent() : objectno(0), offset(0), length(0), truncate_size(0) {}
    ObjectExtent(object_t o, uint64_t ono, uint64_t off, uint64_t l, uint64_t ts) : oid(o), objectno(ono), offset(off), length(l), truncate_size(ts) {}
};
}
```

```
int libradosstriper::RadosStriperImpl::internal_aio_write(const std::string &soid,
                                                          libradosstriper::MultiAioCompletionImplPtr c,
                                                          const bufferlist &bl,
```

```
                                                          size_t len,
                                                          uint64_t off,
                                                          const ceph_file_layout &layout) {
    int r = 0;
    if (len > 0) {
        vector<ObjectExtent> extents;
        std::string format = soid;
        boost::replace_all(format, "%", "%%");
        format += RADOS_OBJECT_EXTENSION_FORMAT;
        file_layout_t l;
        l.from_legacy(layout);

        // 分片
        Striper::file_to_extents(cct(), format.c_str(), &l, off, len, 0, extents);
        // go through the extents
        for (vector<ObjectExtent>::iterator p = extents.begin(); p != extents.end(); ++p) {
            // assemble pieces of a given object into a single buffer list

            // 将每个对象的buffer合并
            bufferlist oid_bl;
            for (vector<pair<uint64_t, uint64_t>>::iterator q = p->buffer_extents.begin();
                 q != p->buffer_extents.end();
                 ++q) {
                bufferlist buffer_bl;
                buffer_bl.substr_of(bl, q->first, q->second);
                oid_bl.append(buffer_bl);
            }
            // and write the object
            // 写操作
            r = m_ioCtx.aio_write(p->oid.name, rados_completion, oid_bl,
                                  p->length, p->offset);
        }
    }
    c->finish_adding_requests();
    return r;
}
```

```
int librados::IoCtxImpl::aio_write(const object_t &oid, AioCompletionImpl *c,
                                   const bufferlist &bl, size_t len,
                                   uint64_t off, const blkin_trace_info *info) {
```

```

    c->io = this;
    queue_aio_write(c);

    Objecter::Op *o = objecter->prepare_write_op( // CEPH_OSD_OP_WRITE
        oid, oloc,
        off, len, snapc, bl, ut, extra_op_flags,
        oncomplete, &c->objver, nullptr, 0, &trace);
    objecter->op_submit(o, &c->tid);

    return 0;
}
```

条带读

```
int libradosstriper::RadosStriper::read(const std::string& soid,
                                        bufferlist* bl,
                                        size_t len,
                                        uint64_t off)
{
  bl->clear();
  bl->push_back(buffer::create(len));
  return rados_striper_impl->read(soid, bl, len, off);
}

int libradosstriper::RadosStriperImpl::read(const std::string &soid,
                                            bufferlist *bl,
                                            size_t len,
                                            uint64_t off) {
    // create a completion object
    librados::AioCompletionImpl c;

    // call asynchronous method
    int rc = aio_read(soid, &c, bl, len, off);

    // and wait for completion
    if (!rc) {
        // wait for completion
        c.wait_for_complete_and_cb();
        // return result
        rc = c.get_return_value();
    }
    return rc;
}
```

```
int libradosstriper::RadosStriperImpl::aio_read(const std::string &soid,
                                                librados::AioCompletionImpl *c,
                                                bufferlist *bl,
                                                size_t len,
                                                uint64_t off) {
    ceph_file_layout layout;
    uint64_t size;
    std::string lockCookie;
// 打开条带化对象
    int rc = openStripedObjectForRead(soid, &layout, &size, &lockCookie);

// 计算实际可读取的字节数
    uint64_t read_len;
    if (off >= size) {
        // nothing to read ! We are done.
        read_len = 0;
    } else {
        read_len = std::min(len, (size_t)(size - off));
    }

// 获取要读的数据列表
    vector<ObjectExtent> *extents = new vector<ObjectExtent>();
    ...
    Striper::file_to_extents(cct(), format.c_str(), &l, off, read_len,
                             0, *extents);
    ...

    int r = 0, i = 0;
    for (vector<ObjectExtent>::iterator p = extents->begin(); p != extents->end(); ++p) {

        bufferlist *oid_bl = &((*resultbl)[i++]);
        for (vector<pair<uint64_t, uint64_t>>::iterator q = p->buffer_extents.begin();
             q != p->buffer_extents.end();
             ++q) {
            bufferlist buffer_bl;
            buffer_bl.substr_of(*bl, q->first, q->second);
            oid_bl->append(buffer_bl);
        }
        ...
// 读取
```

```
        r = m_ioCtx.aio_read(p->oid.name, rados_completion, oid_bl, p->length,
p->offset);

    }
    nc->finish_adding_requests();
    return r;
}
```

```
int librados::IoCtxImpl::aio_read(const object_t oid, AioCompletionImpl *c,
                                  bufferlist *pbl, size_t len, uint64_t off,
                                  uint64_t snapid, const blkin_trace_info
*info) {
    FUNCTRACE(client->cct);

    OID_EVENT_TRACE(oid.name.c_str(), "RADOS_READ_OP_BEGIN");
    Context *oncomplete = new C_aio_Complete(c);

    c->is_read = true;
    c->io = this;
    c->blp = pbl;

    Objecter::Op *o = objecter->prepare_read_op(
        oid, oloc,
        off, len, snapid, pbl, extra_op_flags,
        oncomplete, &c->objver, nullptr, 0, &trace);
    objecter->op_submit(o, &c->tid);
    return 0;
}
```

Objecter::op_submit

```
void Objecter::op_submit(Op *op, ceph_tid_t *ptid, int *ctx_budget)
{
  _op_submit_with_budget(op, rl, ptid, ctx_budget);
}

void Objecter::_op_submit_with_budget(Op *op,
                                      shunique_lock<ceph::shared_mutex>& sul,
                                      ceph_tid_t *ptid,
                                      int *ctx_budget)
```

```
{
  ceph_assert(initialized);

  ceph_assert(op->ops.size() == op->out_bl.size());
  ceph_assert(op->ops.size() == op->out_rval.size());
  ceph_assert(op->ops.size() == op->out_handler.size());

// 流量限制，代码略

  _op_submit(op, sul, ptid);
}
```

```
void Objecter::_op_submit(Op *op, shunique_lock<ceph::shared_mutex>& sul, ceph_tid_t *ptid)
{
  OSDSession *s = NULL;

  int r = _calc_target(&op->target, nullptr);

  r = _get_session(op->target.osd, &s, sul);

  _session_op_assign(s, op);  // op->session = to;

  _send_op(op);
}
```

_calc_target

完成对象到OSD寻址的过程

```
int Objecter::_calc_target(op_target_t *t, Connection *con, bool any_change)
{
    bool is_read = t->flags & CEPH_OSD_FLAG_READ;
    bool is_write = t->flags & CEPH_OSD_FLAG_WRITE;
    t->epoch = osdmap->get_epoch();
    // 获取pool
    const pg_pool_t *pi = osdmap->get_pg_pool(t->base_oloc.pool);
    // 找到pool的位置
    osdmap->object_locator_to_pg(t->target_oid, t->target_oloc, pgid);
    // 获取PG对应的OSD列表
    osdmap->pg_to_up_acting_osds(actual_pgid, &up, &up_primary,
                                 &acting, &acting_primary);
    // 如果是读操作，选择一个副本
```

```
    if ((t->flags & (CEPH_OSD_FLAG_BALANCE_READS |
CEPH_OSD_FLAG_LOCALIZE_READS)) &&
        !is_write && pi->is_replicated() && t->acting.size() > 1) {
        int osd;
        ceph_assert(is_read && t->acting[0] == acting_primary);

        if (t->flags & CEPH_OSD_FLAG_BALANCE_READS) { // 随机选一个
            int p = rand() % t->acting.size();
            if (p) t->used_replica = true;
            osd = t->acting[p];
        } else { // 尽量选本地副本读取
            int best = -1;
            int best_locality = 0;
            for (unsigned i = 0; i < t->acting.size(); ++i) {
                int locality = osdmap->crush->get_common_ancestor_distance(
                    cct, t->acting[i], crush_location);
                if (i == 0 ||
                    (locality >= 0 && best_locality >= 0 &&
                     locality < best_locality) ||
                    (best_locality < 0 && locality >= 0)) {
                    best = i;
                    best_locality = locality;
                    if (i)
                        t->used_replica = true;
                }
            }
            ceph_assert(best >= 0);
            osd = t->acting[best];
        }
        t->osd = osd;
    } else {
        // 写操作，选主OSD
        t->osd = acting_primary;
    }
    return RECALC_OP_TARGET_NO_ACTION;
}
```

_get_session

```
int Objecter::_get_session(int osd, OSDSession **session,
                           shunique_lock<ceph::shared_mutex> &sul) {
    ceph_assert(sul && sul.mutex() == &rwlock);

    auto p = osd_sessions.find(osd);

```

```
    auto s = new OSDSession(cct, osd);
    osd_sessions[osd] = s;
    s->con = messenger->connect_to_osd(osdmap->get_addrs(osd));
    s->con->set_priv(RefCountedPtr{s});

    s->get();
    *session = s;

    return 0;
}

```

_send_op

```
void Objecter::_send_op(Op *op) {
    // rwlock is locked
    // op->session->lock is locked
    ...
    ceph_assert(op->tid > 0);
    MOSDOp *m = _prepare_osd_op(op);

    if (op->target.actual_pgid != m->get_spg()) {
        m->set_spg(op->target.actual_pgid);
        m->clear_payload(); // reencode
    }

    ConnectionRef con = op->session->con;
    ceph_assert(con);

    op->incarnation = op->session->incarnation;

    if (op->trace.valid()) {
        m->trace.init("op msg", nullptr, &op->trace);
    }
    op->session->con->send_message(m);
}

```

收消息

服务端发送消息出去后，会给到ms_dispatch分发，根据不同的消息类型（如读写）给到不同的处理函数

```
bool Objecter::ms_dispatch(Message *m) {
    switch (m->get_type()) {
    case CEPH_MSG_OSD_OPREPLY:
        handle_osd_op_reply(static_cast<MOSDOpReply *>(m));
        return true;
}
```

这里不仔细看了，直接给出gpt的分析

```
void Objecter::handle_osd_op_reply(MOSDOpReply *m) {

    获取操作ID：从传入的MOSDOpReply消息对象m中获取操作的唯一标识符tid。
    获取会话信息：通过消息对象获取连接信息，并尝试获取与该连接关联的OSDSession对象。
    查找操作对象：在OSDSession对象中查找与tid关联的Op对象。如果找不到或会话信息不匹配，释放消息对象并返回。
    处理重试逻辑：如果操作是写操作，并且是第一次回复，可能会根据配置重试该操作。
    处理重定向回复：如果消息是一个重定向回复，更新操作对象的定位器，并重新提交操作。
    处理-EAGAIN错误：如果操作因为某些原因需要重新提交（例如，OSD请求重试），更新操作对象的定位器，并重新提交操作。
    更新操作结果：如果操作成功，更新操作对象的版本号、回复的epoch和数据偏移量。
    处理返回的数据：如果操作返回了数据，将其复制到操作对象指定的bufferlist中。
    处理操作列表：从消息对象中获取返回的操作列表，并与原始请求的操作列表进行比较。
    执行操作结果分发：遍历返回的操作列表，更新每个操作的结果、错误码和处理程序。
    完成操作处理：如果操作对象有关联的完成回调，记录操作完成，减少正在进行的操作计数，并调用完成回调。
    释放消息对象：最后，释放传入的MOSDOpReply消息对象。

}
```

注意

虽然大部分时候Ioctx 都是直接调用的 IoctxImpl的实现，且函数名都一模一样，但是也有少部分例外（有时候偷懒直接跳过IoCtx的实现，结果源码越来越难看），比如:

```
m_io_ctx.aio_operate(RBD_INFO, comp, &op, &m_out_bl);
```

这里ioctx调用的并不是对应Impl的aio_operate，ioctx的这个函数有多个版本的重载，而并不是所有版本都会调用IoCtxImpl::aio_operate，当前情况，如果op是ObjectReadOperation类型，应该是调用aio_operate_read

```
int librados::IoCtx::aio_operate(const std::string& oid, AioCompletion *c,
                                 librados::ObjectReadOperation *o,
                                 bufferlist *pbl)
{
  if (unlikely(!o->impl))
    return -EINVAL;
  object_t obj(oid);
  return io_ctx_impl->aio_operate_read(obj, &o->impl->o, c->pc, 0, pbl);
}
```