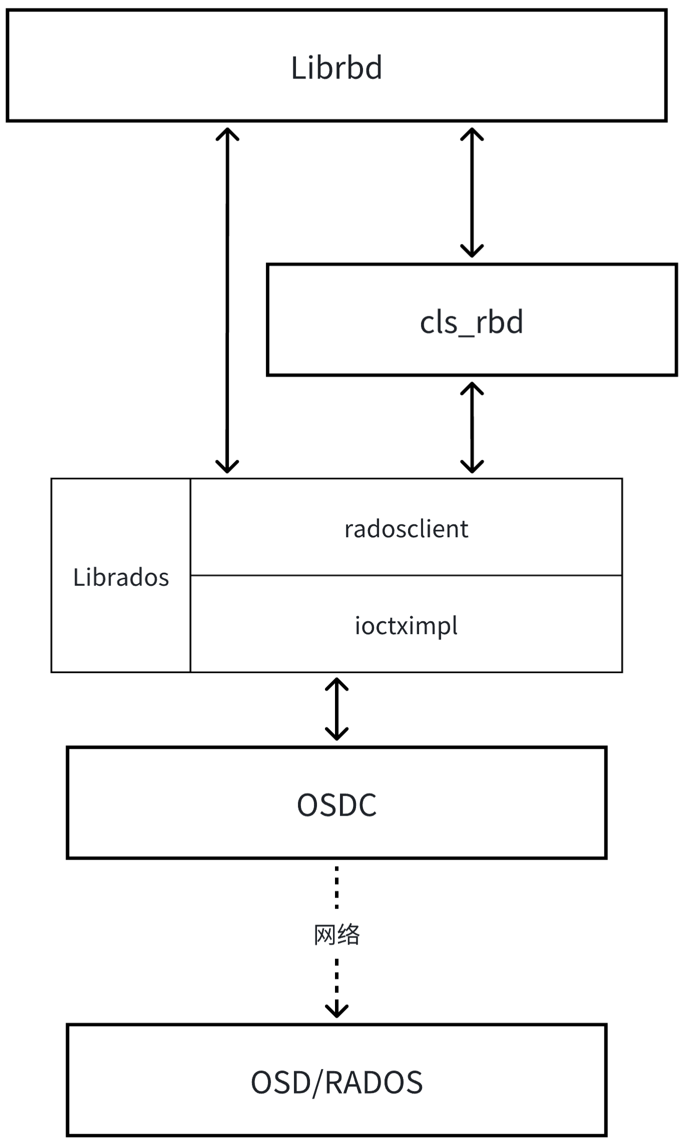
Librbd

ceph版本：https://github.com/ceph/ceph/tree/v18.2.1

RBD（RADOS Block Device）是由Ceph提供的存储结果，对于虚拟机来说，rbd设备就是本机的磁盘，对于ceph来说，rbd设备是分散在各个OSD节点的数据。

Librbd模块实现了RBD接口，基于librados实现对rbd的基本操作。



cls_rbd是一个扩展模块，实现了RBD的元数据相关操作，RBD的数据直接通过librados访问，最底层由OSDC完成数据的发送。

Cls

Cls是允许用户自定义对象的操作接口和实现方法，为用户提供了一种比较方便的接口扩展方式，比如cls_rbd就被rbd用来做元数据的操作。

类classHandler用来管理所有的扩展模块，函数register_class用来注册模块：

```

class ClassHandler {
    // 注册的模块名 ——> 模块元数据信息
    std::map<std::string, ClassData> classes;

    struct ClassData {
        enum Status {
            CLASS_UNKNOWN,
            CLASS_MISSING,      // missing
            CLASS_MISSING_DEPS, // missing dependencies
            CLASS_INITIALIZING, // calling init() right now
            CLASS_OPEN,         // initialized, usable
        } status = CLASS_UNKNOWN;

        std::string name;
        ClassHandler *handler = nullptr;
        void *handle = nullptr;

        std::map<std::string, ClassMethod> methods_map;   // 注册的方法
        std::map<std::string, ClassFilter> filters_map;   // 注册过滤方法

        std::set<ClassData *> dependencies;         /* our dependencies */
        std::set<ClassData *> missing_dependencies; /* only missing dependencies */
    };
    struct ClassMethod {
        const std::string name;
        using func_t = std::variant<cls_method_cxx_call_t, cls_method_call_t>; // 函数指针类型
        func_t func;    // 函数指针
        int flags = 0;
        ClassData *cls = nullptr;   // 所属模块的ClassDate指针
    };
}
```

注册一个模块

```
int cls_register(const char *name, cls_handle_t *handle) {
    ClassHandler::ClassData *cls =
```

```
        ClassHandler::get_instance().register_class(name);
    *handle = (cls_handle_t)cls;
    return (cls != NULL);
}
```

注册一个模块所属的函数

```
int cls_register_method(cls_handle_t hclass, const char *method,
                        int flags,
                        cls_method_call_t class_call, cls_method_handle_t *handle) {
    if (!(flags & (CLS_METHOD_RD | CLS_METHOD_WR)))
        return -EINVAL;
    ClassHandler::ClassData *cls = (ClassHandler::ClassData *)hclass;
    cls_method_handle_t hmethod = (cls_method_handle_t)cls->register_method(method, flags, class_call);
    if (handle)
        *handle = hmethod;
    return (hmethod != NULL);
}
```

执行方法

```
int PrimaryLogPG::do_osd_ops(OpContext *ctx, vector<OSDOp> &ops):
case CEPH_OSD_OP_CALL: {
    string cname, mname;
    bufferlist indata;

    bp.copy(op.cls.class_len, cname);
    bp.copy(op.cls.method_len, mname);
    bp.copy(op.cls.indata_len, indata);

    ClassHandler::ClassData *cls;
    result = ClassHandler::get_instance().open_class(cname, &cls);

    ClassHandler::ClassMethod *method = cls->get_method(mname);

    int flags = method->get_flags();
    if (flags & CLS_METHOD_WR)
        ctx->user_modify = true;

    bufferlist outdata;

```

```
    int prev_rd = ctx->num_read;
    int prev_wr = ctx->num_write;
    result = method->exec((cls_method_context_t)&ctx, indata, outdata);

    op.extent.length = outdata.length();
    osd_op.outdata.claim_append(outdata);
} break;
```

cls_rbd

cls_rbd.cc

注册了rbd模块和自定义的方法

```
cls_register("rbd", &h_class);
cls_register_cxx_method(h_class, "create",
                        CLS_METHOD_RD | CLS_METHOD_WR,
                        create, &h_create);
```

上面注册了create方法，其实现也在cls_rbd.cc中，这一类方法有三个参数：操作上下文，输入缓冲区，输出缓冲区。

```
int create(cls_method_context_t hctx, bufferlist *in, bufferlist *out)
```

注册完后，在Librados层的ioctx中会封装为CEPH_OSD_OP_CALL的操作

```
void create_image(librados::ObjectWriteOperation *op, uint64_t size,
                  uint8_t order, uint64_t features,
                  const std::string &object_prefix, int64_t data_pool_id)
{
    bufferlist bl;
    encode(size, bl);
    encode(order, bl);
    encode(features, bl);
    encode(object_prefix, bl);
    encode(data_pool_id, bl);

    op->exec("rbd", "create", bl);
}

void librados::ObjectOperation::exec(const char *cls, const char *method,
```

```
                                       bufferlist& inbl)
{
    ceph_assert(impl);
    ::ObjectOperation *o = &impl->o;
    o->call(cls, method, inbl);
}
```

```
void call(const char *cname, const char *method, ceph::buffer::list &indata) {
    add_call(CEPH_OSD_OP_CALL, cname, method, indata, NULL, NULL, NULL);
}

void add_call(int op, std::string_view cname, std::string_view method,
            const ceph::buffer::list &indata,
            fu2::unique_function<void(boost::system::error_code,
                                      const ceph::buffer::list&) &&> f) {
    OSDOp& osd_op = add_op(op);

    set_handler([f = std::move(f)](boost::system::error_code ec,
                                   int,
                                   const ceph::buffer::list& bl) mutable {
                  std::move(f)(ec, bl);
                });

    osd_op.op.cls.class_len = cname.size();
    osd_op.op.cls.method_len = method.size();
    osd_op.op.cls.indata_len = indata.length();
    osd_op.indata.append(cname.data(), osd_op.op.cls.class_len);
    osd_op.indata.append(method.data(), osd_op.op.cls.method_len);
    osd_op.indata.append(indata);
}


```

OSD收到操作后，通过exec接口调用具体的函数实现

```
int PrimaryLogPG::do_osd_ops(OpContext *ctx, vector<OSDOp> &ops):
case CEPH_OSD_OP_CALL:
    result = method->exec((cls_method_context_t)&ctx, indata, outdata);
```

Librbd

使用示例, 完整版见examples/librbd/hello_word.cc

```
#include <iostream>
#include <rados/librados.hpp>
#include <rbd/librbd.hpp>
#include <sstream>
#include <string>

int main(int argc, const char **argv) {
    ret = rados.connect();
// 创建pool
    ret = rados.pool_create(pool_name);
    ret = rados.ioctx_create(pool_name, io_ctx);

    std::string name = "librbd_test";
    uint64_t size = 2 << 20;
    int order = 0;
    librbd::RBD rbd;
    librbd::Image image;
// 创建image
    ret = rbd.create(io_ctx, name.c_str(), size, &order);
// 打开image
    ret = rbd.open(io_ctx, image, name.c_str(), NULL);

    int TEST_IO_SIZE = 512;
    char test_data[TEST_IO_SIZE + 1];
    int i;

    for (i = 0; i < TEST_IO_SIZE; ++i) {
        test_data[i] = (char)(rand() % (126 - 33) + 33);
    }
    test_data[TEST_IO_SIZE] = '\0';

    size_t len = strlen(test_data);
    ceph::bufferlist bl;
    bl.append(test_data, len);

// 写数据
    ret = image.write(0, len, bl);

    ceph::bufferlist bl_r;
    int read;
// 读数据
```

```
    read = image.read(0, TEST_IO_SIZE, bl_r);

    std::string bl_res(bl_r.c_str(), read);

    int res = memcmp(bl_res.c_str(), test_data, TEST_IO_SIZE);
// 关闭image
    image.close();

    ret = rbd.remove(io_ctx, name.c_str());

    ret = EXIT_SUCCESS;

    int delete_ret = rados.pool_delete(pool_name);

    rados.shutdown();

    return ret;
}

```

创建RBD

internal.cc

librbd.cc中librbd::create有很多个版本，最终都是调用internal.cc下的create

```
int create(IoCtx &io_ctx, const std::string &image_name,
           const std::string &image_id, uint64_t size,
           ImageOptions &opts,
           const std::string &non_primary_global_image_id,
           const std::string &primary_mirror_uuid,
           bool skip_mirror_enable) {
// 1. 如果image_id为空会生成一个新的id
    std::string id(image_id);
    if (id.empty()) {
        id = util::generate_image_id(io_ctx);
    }

    CephContext *cct = (CephContext *)io_ctx.cct();
    uint64_t option;
// 2. 不支持这两种选项
    if (opts.get(RBD_IMAGE_OPTION_FLATTEN, &option) == 0) {
        return -EINVAL;
    }
```

```
    if (opts.get(RBD_IMAGE_OPTION_CLONE_FORMAT, &option) == 0) {
        return -EINVAL;
    }

// 3. 获取image格式
    uint64_t format;
    if (opts.get(RBD_IMAGE_OPTION_FORMAT, &format) != 0)
        format = cct->_conf.get_val<uint64_t>("rbd_default_format");
    bool old_format = format == 1;

// 4. 检查image是否已存在，如果已存在就返回错误
    int r = detect_format(io_ctx, image_name, NULL, NULL);

// 5. 获取order，决定了RBD对象的大小
// image的order表示其数据会被分成2^order个部分
    uint64_t order = 0;
    if (opts.get(RBD_IMAGE_OPTION_ORDER, &order) != 0 || order == 0) {
        order = cct->_conf.get_val<uint64_t>("rbd_default_order");
    }
    // 验证order
    r = image::CreateRequest<>::validate_order(cct, order);

// 6. 创建，如果是旧格式，且设置了RBD_FORCE_ALLOW_V1(略)使用create_v1
    if (old_format) {
        r = create_v1(io_ctx, image_name.c_str(), size, order);
    } else {
// 6.1 否则使用AsioEngine创建image
        AsioEngine asio_engine(io_ctx);

        ConfigProxy config{cct->_conf};
        api::Config<>::apply_pool_overrides(io_ctx, &config);

        uint32_t create_flags = 0U;
        uint64_t mirror_image_mode = RBD_MIRROR_IMAGE_MODE_JOURNAL;
        if (skip_mirror_enable) {
            create_flags = image::CREATE_FLAG_SKIP_MIRROR_ENABLE;
        } else if (opts.get(RBD_IMAGE_OPTION_MIRROR_IMAGE_MODE,
                            &mirror_image_mode) == 0) {
            create_flags = image::CREATE_FLAG_FORCE_MIRROR_ENABLE;
        }

        C_SaferCond cond;
        image::CreateRequest<> *req = image::CreateRequest<>::create(
            config, io_ctx, image_name, id, size, opts, create_flags,
            static_cast<cls::rbd::MirrorImageMode>(mirror_image_mode),
            non_primary_global_image_id, primary_mirror_uuid,
            asio_engine.get_work_queue(), &cond);
```

```
        req->send();

        r = cond.wait();
    }

    int r1 = opts.set(RBD_IMAGE_OPTION_ORDER, order);
    ceph_assert(r1 == 0);
    return r;
}

```

CreateRequest

```
template <typename ImageCtxT = ImageCtx>
class CreateRequest {
public:
  static CreateRequest *create(const ConfigProxy& config, IoCtx &ioctx,
                               const std::string &image_name,
                               const std::string &image_id, uint64_t size,
                               const ImageOptions &image_options,
                               uint32_t create_flags,
                               cls::rbd::MirrorImageMode mirror_image_mode,
                               const std::string &non_primary_global_image_id,
                               const std::string &primary_mirror_uuid,
                               asio::ContextWQ *op_work_queue,
                               Context *on_finish) {
    return new CreateRequest(config, ioctx, image_name, image_id, size,
                             image_options, create_flags,
                             mirror_image_mode, non_primary_global_image_id,
                             primary_mirror_uuid, op_work_queue, on_finish);
  }

  static int validate_order(CephContext *cct, uint8_t order);

  void send();
}

template<typename I>
void CreateRequest<I>::send() {

  int r = validate_features(m_cct, m_features);

  r = validate_order(m_cct, m_order);

  r = validate_striping(m_cct, m_order, m_stripe_unit, m_stripe_count);
```

```
  if (((m_features & RBD_FEATURE_OBJECT_MAP) != 0) &&
      (!validate_layout(m_cct, m_size, m_layout))) {
    complete(-EINVAL);
    return;
  }

  validate_data_pool();
}

template <typename I>
void CreateRequest<I>::validate_data_pool() {
  m_data_io_ctx = m_io_ctx;
  if ((m_features & RBD_FEATURE_DATA_POOL) != 0) {
    librados::Rados rados(m_io_ctx);
    int r = rados.ioctx_create(m_data_pool.c_str(), m_data_io_ctx);
    m_data_pool_id = m_data_io_ctx.get_id();
    m_data_io_ctx.set_namespace(m_io_ctx.get_namespace());
  }

  if (!m_config.get_val<bool>("rbd_validate_pool")) {
    add_image_to_directory();
    return;
  }

  auto ctx = create_context_callback<
    CreateRequest<I>, &CreateRequest<I>::handle_validate_data_pool>(this);
  auto req = ValidatePoolRequest<I>::create(m_data_io_ctx, ctx);
  req->send();
}
```

ValidatePoolRequest

```
class ValidatePoolRequest {
public:
  static ValidatePoolRequest* create(librados::IoCtx& io_ctx,
                                     Context *on_finish) {
    return new ValidatePoolRequest(io_ctx, on_finish);
  }

  ValidatePoolRequest(librados::IoCtx& io_ctx, Context *on_finish);

  void send();
}

```

```
template <typename I>
void ValidatePoolRequest<I>::send() {
  read_rbd_info();
}

template <typename I>
void ValidatePoolRequest<I>::read_rbd_info() {
  ldout(m_cct, 5) << dendl;

  auto comp = create_rados_callback<
    ValidatePoolRequest<I>,
    &ValidatePoolRequest<I>::handle_read_rbd_info>(this);

  librados::ObjectReadOperation op;
  op.read(0, 0, nullptr, nullptr);

  m_out_bl.clear();
  int r = m_io_ctx.aio_operate(RBD_INFO, comp, &op, &m_out_bl);
  ceph_assert(r == 0);
  comp->release();
}
```

异步回调处理 AioCompletion

1. create_rados_callback

提供多种重载，应对不同的回调方式，可以看到均返回AioCompletion* 类型，在上一个函数中， 调用的是第三个重载版本。

```
librados::AioCompletion *create_rados_callback(Context *on_finish) {
    return create_rados_callback<Context, &Context::complete>(on_finish);
}

template <typename T>
librados::AioCompletion *create_rados_callback(T *obj) {
    return librados::Rados::aio_create_completion(
      obj, &detail::rados_callback<T>);
}

template <typename T, void(T::*MF)(int)>
librados::AioCompletion *create_rados_callback(T *obj) {
    return librados::Rados::aio_create_completion(
      obj, &detail::rados_callback<T, MF>);
}

```

```
// 这个版本带一个完成后删除对象的选项
template <typename T, Context*(T::*MF)(int*), bool destroy=true>
librados::AioCompletion *create_rados_callback(T *obj) {
    return librados::Rados::aio_create_completion(
      obj, &detail::rados_state_callback<T, MF, destroy>);
}
```

1.1 detail::rados_callback

这个函数有两个重载，而本质上都是在调用传入的函数

```
template <typename T>
void rados_callback(rados_completion_t c, void *arg) {
  reinterpret_cast<T*>(arg)->complete(rados_aio_get_return_value(c));
}

template <typename T, void(T::*MF)(int)>
void rados_callback(rados_completion_t c, void *arg) {
  T *obj = reinterpret_cast<T*>(arg);
  int r = rados_aio_get_return_value(c);
  (obj->*MF)(r);
}
```

rados_state_callback 和rados_callback不同的是它会根据执行结果和destroy标志选择删除对象

```
template <typename T, Context*(T::*MF)(int*), bool destroy>
void rados_state_callback(rados_completion_t c, void *arg) {
  T *obj = reinterpret_cast<T*>(arg);
  int r = rados_aio_get_return_value(c);
  Context *on_finish = (obj->*MF)(&r);
  if (on_finish != nullptr) {
    on_finish->complete(r);
    if (destroy) {
      delete obj;
    }
  }
}
```

1.2 aio_create_completion

传入参数和回调方法，返回AioCompletion对象

```
librados::AioCompletion *librados::Rados::aio_create_completion(void *cb_arg,
                                                                callback_t
cb_complete)
{
    AioCompletionImpl *c;
    int r = rados_aio_create_completion2(cb_arg, cb_complete, (void**)&c);
    ceph_assert(r == 0);
    return new AioCompletion(c);
}

extern "C" int rados_aio_create_completion2(void *cb_arg,
                                            rados_callback_t cb_complete,
                                            rados_completion_t *pc)
{
    librados::AioCompletionImpl *c = new librados::AioCompletionImpl;
    if (cb_complete) {
        c->set_complete_callback(cb_arg, cb_complete);
    }
    *pc = c;
    return 0;
}

int set_complete_callback(void *cb_arg, rados_callback_t cb) {
    std::scoped_lock l{lock};
    callback_complete = cb;
    callback_complete_arg = cb_arg;
    return 0;
}
```

2. 回调类的发送

得到AioCompletion对象后，传入执行操作的函数

```
// AioCompletion* comp = funcxxx
// librados::ObjectReadOperation op
m_io_ctx.aio_operate(RBD_INFO, comp, &op, &m_out_bl);
```

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

```
}

int librados::IoCtxImpl::aio_operate_read(const object_t &oid,
                                          ::ObjectOperation *o,
                                          AioCompletionImpl *c,
                                          int flags,
                                          bufferlist *pbl,
                                          const blkin_trace_info *trace_info) {
    Context *oncomplete = new C_aio_Complete(c);

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

    Objecter::Op *objecter_op = objecter->prepare_read_op(
        oid, oloc,
        *o, snap_seq, pbl, flags | extra_op_flags,
        oncomplete, &c->objver, nullptr, 0, &trace);

    objecter->op_submit(objecter_op, &c->tid);
    trace.event("rados operate read submitted");

    return 0;
}
```

上面代码中能看到，AioCompletionImpl又被封装了一层，变成了C_aio_Complete。

下一步，oncomplete被塞到Op中：

```
Op *prepare_read_op(...) {
  ...
// 这里，oncomplete摇身一变成了onack,参与Op的构造
    Op *o = new Op(oid, oloc, std::move(op.ops), flags | global_op_flags |
                   CEPH_OSD_FLAG_READ, onack, objver,
                   data_offset, parent_trace);
   ...
    return o;
}

// nack在这里又换了一个名字，叫fin，然后fin赋值给了onfinish
Op(const object_t& o, const object_locator_t& ol, osdc_opvec&& _ops,
    int f, Context* fin, version_t *ov, int *offset = nullptr,
    ZTracer::Trace *parent_trace = nullptr) :
    target(o, ol, f),
    ops(std::move(_ops)),
    out_bl(ops.size(), nullptr),
    out_handler(ops.size()),
```

```
    out_rval(ops.size(), nullptr),
    out_ec(ops.size(), nullptr),
    onfinish(fin),
    objver(ov),
    data_offset(offset) { ... }
```

AioCompletion全部都是按照上面类似的流程，封装到Op中并发送到服务端。

3. 执行回调*

当收到服务端响应后，网络模块将消息转到ms_dispatch

```
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;
...
}
```

处理响应时，调用之前设置好的回调函数

```
void Objecter::handle_osd_op_reply(MOSDOpReply *m) {
    if (op->has_completion()) {
        num_in_flight--;
        onfinish = std::move(op->onfinish);
        op->onfinish = nullptr;
    }
    if (Op::has_completion(onfinish)) {
        Op::complete(std::move(onfinish), osdcode(rc), rc);
    }
}

// 这里用到了C++17的多个特性，不展开了
// 总之在当前情景，它实际调用了onfinish->complete(r)

static void complete(decltype(onfinish)&& f, boost::system::error_code ec,
                     int r) {
  std::visit([ec, r](auto&& arg) {
               if constexpr (std::is_same_v<std::decay_t<decltype(arg)>,
                                     Context*>) {
                 arg->complete(r);
```

```
              } else if constexpr (std::is_same_v<std::decay_t<decltype(arg)>,
                                   fu2::unique_function<OpSig>>) {
                std::move(arg)(ec);
              } else {
                arg->defer(std::move(arg), ec);
              }
            }, std::move(f));
}
```

ofinish是C_aio_Complete类型，再来看

```
// C_aio_Completion并没有complete函数，在其父类Context中
  virtual void complete(int r) {
    finish(r);
    delete this;
  }
  virtual void finish(int r) = 0;


// C_aio_Completion
void librados::IoCtxImpl::C_aio_Complete::finish(int r) {
    c->lock.lock();
    ...
    c->complete = true;
    c->cond.notify_all();

// 将CB_AioComplete加入到异步队列中，等待执行
// 再深入真太难看了，鬼知道队列中什么逻辑，这里当个黑盒先吧，反正是会调用CB_AioComplete
    if (c->callback_complete ||
        c->callback_safe) {
        boost::asio::defer(c->io->client->finish_strand, CB_AioComplete(c));
    }
    ...
    c->put_unlock();
}

// CB_AioComplete类中operator()执行的就是之前设置的回调函数
struct CB_AioComplete {
  AioCompletionImpl *c;

  explicit CB_AioComplete(AioCompletionImpl *cc) : c(cc) {
    c->_get();
  }

  void operator()() {
```

```
    rados_callback_t cb_complete = c->callback_complete;
    void *cb_complete_arg = c->callback_complete_arg;
    if (cb_complete)
      cb_complete(c, cb_complete_arg);

    rados_callback_t cb_safe = c->callback_safe;
    void *cb_safe_arg = c->callback_safe_arg;
    if (cb_safe)
      cb_safe(c, cb_safe_arg);

    c->lock.lock();
    c->callback_complete = NULL;
    c->callback_safe = NULL;
    c->cond.notify_all();
    c->put_unlock();
  }
};
```

4. 整体流程

这个类就是这样，不断的执行一个操作并设置一个回调，直到finish

```
1. 构造函数 (ValidatePoolRequest<I>::ValidatePoolRequest):
   - 创建一个ValidatePoolRequest类

2. send:
   - 启动验证过程。调用read_rbd_info读取RBD信息。

3. read_rbd_info:
   - 创建一个RADOS回调函数，设置回调 handle_read_rbd_info。
   - 执行异步读取操作，读取RBD_INFO对象。

4. handle_read_rbd_info:
   - 检查异步读取操作的返回值r。
   - r>=0，比较读取到的bufferlist (m_out_bl) 与预定义的字符串OVERWRITE_VALIDATED和VALIDATE。
   - - m_out_bl == OVERWRITE_VALIDATED，表示池已经被验证，直接调用 finish。
   - - m_out_bl == VALIDATE，表示快照已经成功创建，调用 overwrite_rbd_info。
   - 如果r < 0 && r!=-ENOENT，表示读取失败，调用finish并传递错误代码。
   - 若以上条件不满足，调用create_snapshot来创建一个新的快照。

5. create_snapshot 和 handle_create_snapshot:
   - 创建一个快照，设置回调 handle_create_snapshot
   = 处理快照创建的结果。成功，调用 write_rbd_info

```

```
6. write_rbd_info 和 handle_write_rbd_info:
   - 写新RBD的信息到RBD_INFO对象，设置回调 handle_write_rbd_info
   = 处理写操作的结果。成功，调用 remove_snapshot

7. remove_snapshot 和 handle_remove_snapshot :
   - 移除之前创建的验证快照，设置回调 handle_remove_snapshot
   = 处理快照移除的结果。成功，调用 overwrite_rbd_info

8. overwrite_rbd_info 和 handle_overwrite_rbd_info :
   - 覆盖RBD信息，设置回调 handle_overwrite_rbd_info
   = 处理覆盖操作的结果。无论是否成功，都finish。

9. finish :
   - delete this;
   - m_on_finish->complete(r)
```

CreateRequest 续

ValidatePoolRequest::finish中的回调函数，是之前CreateRequest<I>::validate_data_pool设置的。但是这里的机制又不一样了~~(真该死啊)~~。

create_context_callback

```
auto ctx = create_context_callback<
    CreateRequest<I>, &CreateRequest<I>::handle_validate_data_pool>(this);
auto req = ValidatePoolRequest<I>::create(m_data_io_ctx, ctx);
req->send();
```

提供多种重载版本，满足任何需求

```
// 基本回调适配器
template <typename T, void (T::*MF)(int) = &T::complete>
Context *create_context_callback(T *obj) {
    return new detail::C_CallbackAdapter<T, MF>(obj);
}

// 状态回调适配器 提供删除对象参数
template <typename T, Context *(T::*MF)(int *), bool destroy = true>
Context *create_context_callback(T *obj) {
    return new detail::C_StateCallbackAdapter<T, MF, destroy>(obj);
}

// 带引用计数的回调适配器
```

```
template <typename T, void (T::*MF)(int) = &T::complete>
Context *create_context_callback(T *obj, RefCountedPtr refptr) {
    return new detail::C_RefCallbackAdapter<T, MF>(obj, refptr);
}

// 带引用计数的状态回调适配器
template <typename T, Context *(T::*MF)(int *)>
Context *create_context_callback(T *obj, RefCountedPtr refptr) {
    return new detail::C_RefStateCallbackAdapter<T, MF>(obj, refptr);
}

// 下面两种太抽象了算了以后碰到再说
template <typename T, void (T::*MF)(int) = &T::complete, typename R>
typename std::enable_if<not std::is_base_of<RefCountedPtr, R>::value, Context *>::type
create_context_callback(T *obj, R *refptr) {
    return new detail::C_CallbackAdapter<T, MF>(obj);
}

template <typename T, Context *(T::*MF)(int *), typename R, bool destroy = true>
typename std::enable_if<not std::is_base_of<RefCountedPtr, R>::value, Context *>::type
create_context_callback(T *obj, R *refptr) {
    return new detail::C_StateCallbackAdapter<T, MF, destroy>(obj);
}
```

每一种模板返回的东西都完全不一样，在rbd_create场景中，使用的是第一个适配器

```
template <typename T, void (T::*MF)(int)>
class C_CallbackAdapter : public Context {
    T *obj;

public:
    C_CallbackAdapter(T *obj) : obj(obj) {
    }

protected:
    void finish(int r) override {
        (obj->*MF)(r);
    }
};
// 其它适配器虽然名字唬人，但其实做的事情类似，引用计数的就是多了个引用，状态的就是多考虑了下状态
```

这些适配器的父类是Context，之前提到它有一个虚函数和一个纯虚函数：

```
virtual void complete(int r) {
    finish(r);
    delete this;
}
virtual void finish(int r) = 0;
```

到这里context_callback的回调机制就分析完了，相比于rados_callback简单很多

complete

CreateRequest中有一个m_on_finish回调指针，由上层指定，在internal.cc的代码中，有这么一段

```
C_SaferCond cond;
image::CreateRequest<> *req = image::CreateRequest<>::create(
    config, io_ctx, image_name, id, size, opts, create_flags,
    static_cast<cls::rbd::MirrorImageMode>(mirror_image_mode),
    non_primary_global_image_id, primary_mirror_uuid,
    asio_engine.get_work_queue(), &cond);
req->send();

r = cond.wait();
```

m_on_finish被设置为cond，如果CreateRequest完成了操作，不管失败与否，都会调用complete函数

```
template<typename I>
void CreateRequest<I>::complete(int r) {
  ldout(m_cct, 10) << "r=" << r << dendl;

  m_data_io_ctx.close();
  auto on_finish = m_on_finish;
  delete this;
  on_finish->complete(r);
}

class C_SaferCond : public Context {
    void complete(int r) override {
      std::lock_guard l(lock);
      done = true;
      rval = r;
```

```
        cond.notify_all();
    }
}
```

整体流程

现在思路清晰了

1. CreateRequest执行操作，如果需要和外部交互的话，将操作交给其他类，同时设置回调context_callback，自己就不管事了，等着内部类finish回调就好，如果失败，调用complete结束操作。
2. 要和外部交互的话，就设置rados_callback回调函数，等待异步回调，直到finish。
3. 这个类中，几乎全是按照这个模式走创建RBD的流程。

```
1. 构造函数 (CreateRequest<I>::CreateRequest):
   - 设置配置、I/O 上下文、图像名称和ID、大小、图像选项、创建标志、镜像模式、
   - 非主镜像全局ID、主镜像UUID、操作工作队列和回调。

2. send:
   - 启动创建image的流程
   - validate_features 检查请求的特性是否受支持
   - validate_order 确保order在区间[12, 25]，一个条带对象大小为2^order
   - validate_striping 验证条带大小和数量是否有效
   - validate_layout 验证图像大小是否与对象映射兼容
   - validate_data_pool 验证pool，
   - - 根据配置信息跳过验证pool，直接调用 add_image_to_directory
   - - 走ValidatePoolRequest流程(见上文)
   - - 设置context_callback类型回调, 回调函数是 handle_validate_data_pool

3. handle_validate_data_pool:
   - 处理pool验证结果。成功，调用 add_image_to_directory

4. add_image_to_directory 和 handle_add_image_to_directory:
   - IoCtx写操作，将新image添加到RBD目录中。设置rados_callback类型的回调
   = 验证结果。成功则调用 create_id_object

9. create_id_object 和 handle_create_id_object:
   - IoCtx写操作，创建image的ID对象。设置rados_callback类型的回调
   = 验证结果。成功则调用 negotiate_features

10. negotiate_features 和 handle_negotiate_features:
    - IoCtx读操作，获取服务器的特性集，设置rados_callback类型回调
    - 也可以根据配置，跳过这一步 直接调用 create_image
    = 和服务器的特性集比较，不一致则更新，目的在于使用服务器支持的特性创建image
```

```
    = 代码中没有例外情况，使用服务器支持的特性，调用 create_image

11. create_image 和 handle_create_image:
    - IoCtx写操作，创建镜像，设置rados_callback类型回调
    = 成功则调用 set_stripe_unit_count

12. set_stripe_unit_count 和 handle_set_stripe_unit_count:
    - 如果unit count都没被设置或者都为默认值，调用 object_map_resize
    - 否则需要设置，IoCtx写操作，在服务端改掉默认配置，设置rados_callback类型的回调
    = 成功则调用 object_map_resize

13. object_map_resize 和 handle_object_map_resize:
    - IoCtx写操作，调整object_map的大小，rados_callback
    = 成功则 fetch_mirror_mode

14. fetch_mirror_mode 和 handle_fetch_mirror_mode:
    - IoCtx读操作，获取镜像模式
    = 成功 journal_create

15. journal_create 和 handle_journal_create:
    - 创建一个新的日志请求，用于在RBD镜像中启用基于日志的快照和复制功能
    - 创建context_callback类型回调，回调函数是 handle_journal_create
    - 进入流程：librbd::journal::CreateRequest(见下文)
    = 创建成功 mirror_image_enable

16. mirror_image_enable 和 handle_mirror_image_enable 回调:
    - 启用镜像模式
    - 创建context_callback类型回调，回调函数是 handle_mirror_image_enable
    - 进入流程mirror::EnableRequest(见下文)
    = 成功则 complete

17. complete:
    - delete this
    - on_finish->complete(r);

18. 清理操作
如果出现错误，会调用清理操作，从“高层”到“低”的顺序为：
journal_remove
remove_object_map
remove_header_object
remove_id_object
remove_from_dir
```

> Ioctx写操作，在 OSDC 中介绍过

librbd::journal::CreateRequest

之前的部分已经非常详细的描述这类代码的运行框架，之后仅描述流程，不再涉及实现细节

```
1. 构造函数 (CreateRequest<I>::CreateRequest):
   - 初始化I/O上下文、镜像ID、日志顺序、展开宽度、对象池名称、
   - 标签类别、标签数据、客户端ID、操作工作队列和完成回调。

2. 发送请求 (CreateRequest<I>::send):
   - 启动创建日志的流程。
   - 确保 order在区间[12, 64], m_splay_width==0 // 这个参数没听过，据说是为了控制日志的分散程度，减少负载之类的，没具体了解
   - 调用函数get_pool_id

3. get_pool_id
   - 获取当前pool的id
   - 调用 create_journal

4. create_journal 和 handle_create_journal
   - 创建一个Journaler对象，用于管理镜像的日志，并启动日志创建过程
   - 设置context_callback类型回调，回调函数是 handle_create_journal
   - Journaler->create 使用IoCtx写操作，创建一个Journaler对象，并执行上层给的回调
   = 若创建成功，调用 allocate_journal_tag

5. allocate_journal_tag 和 handle_register_client
   - 为日志分配标签
   - context_callback类型回调，函数为handle_register_client
   - m_journaler->allocate_tag 同样将操作给到IoCtx层，完成后回调
   = 成功则 register_client

6. register_client 和 handle_register_client
   - 在日志中注册客户端，套路不变，m_journaler->register_client
   = 成功，shut_down_journaler

7. shut_down_journaler 和 handle_journaler_shutdown
   - 关闭Journaler对象，套路同上，m_journaler->shut_down
   = 成功，complete(0)

8. complete
    - m_on_finish->complete(r)
    - delete this

9. remove_journal 和 handle_remove_journal
   - 在创建好Journaler之后的所有失败处理，由这部分移除
   - 创建RemoveRequest对象，进行移除操作
   - complete(result)
```

这里主要是流程控制，具体创建操作是靠Journaler类实现，Journaler类又通过IoCtx执行写操作

mirror::EnableRequest

源码中给出了流程，那我就不仔细分析了。

代码框架前面已经写几遍了，无非就是Op——>handle_Op——>NextOp的循环，直到finish调用上层回调。

这个流程的目的是启用镜像复制

```
<start>
   |
   v
GET_MIRROR_IMAGE * * * * * * *             获取镜像的复制状态
   |                         * (on error)
   v (skip if not needed)    *
GET_TAG_OWNER  * * * * * * * *             获取标签所有者
   |                         *
   v (skip if not needed)    *
OPEN_IMAGE                   *             打开镜像
   |                         *
   v (skip if not needed)    *
CREATE_PRIMARY_SNAPSHOT  * * *             创建初始快照
   |                         *
   v (skip of not opened)    *
CLOSE_IMAGE                  *             关闭镜像
   |                         *
   v (skip if not needed)    *
ENABLE_NON_PRIMARY_FEATURE   *             启用非主特性，这个特性保证其只读
   |                         *
   v (skip if not needed)    *
IMAGE_STATE_UPDATE * * * * * *             更新镜像状态
   |                         *
   v                         *
<finish>   < * * * * * * * * *
```

至此，rbd镜像的创建流程分析完了

Open

```
int RBD::open(IoCtx& io_ctx, Image& image, const char *name,
            const char *snap_name)
{
```

```
    ImageCtx *ictx = new ImageCtx(name, "", snap_name, io_ctx, false);

    int r = ictx->state->open(0);

    image.ctx = (image_ctx_t) ictx;

    return 0;
}

struct ImageCtx {
    ImageState<ImageCtx> *state;
}

template <typename I>
int ImageState<I>::open(uint64_t flags) {
    C_SaferCond ctx;
    open(flags, &ctx);

    int r = ctx.wait();
    return r;
}

template <typename I>
void ImageState<I>::open(uint64_t flags, Context *on_finish) {
    CephContext *cct = m_image_ctx->cct;

    m_lock.lock();
    ceph_assert(m_state == STATE_UNINITIALIZED);
    m_open_flags = flags;

    Action action(ACTION_TYPE_OPEN);
    action.refresh_seq = m_refresh_seq;
    execute_action_unlock(action, on_finish);
}

template <typename I>
void ImageState<I>::send_open_unlock() {
    ceph_assert(ceph_mutex_is_locked(m_lock));
    CephContext *cct = m_image_ctx->cct;

    m_state = STATE_OPENING;

    Context *ctx = create_context_callback<ImageState<I>,
&ImageState<I>::handle_open>(this);
    image::OpenRequest<I> *req =
        image::OpenRequest<I>::create(m_image_ctx, m_open_flags, ctx);

```

```
    m_lock.unlock();
    req->send();
}
```

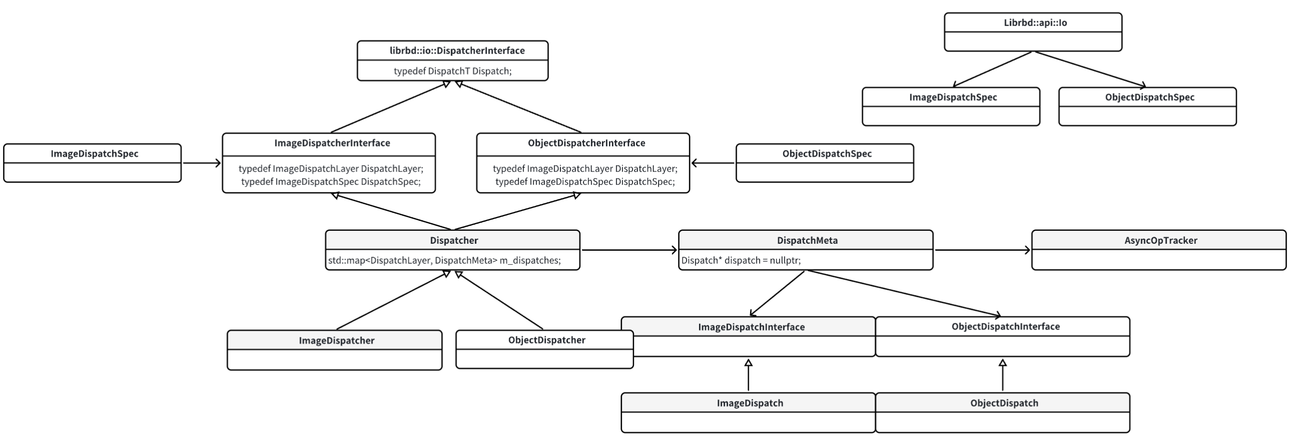
```
<start>
   |
   | (v1)
   |-----> V1_DETECT_HEADER 尝试检测RBD镜像的v1格式头部
   |             |
   |             \-------------------------------\
   | (v2)                                        |
   \-----> V2_DETECT_HEADER                      /
         检测v2格式头部                            |
                 |                               |
                 v                               |
           获取镜像的ID或名称呼                     |
           V2_GET_ID|NAME                        |
                 |                               |
                 v (skip if have name)           |
     在正常位置找不到镜像名称，尝试在垃圾箱中检索        |
           V2_GET_NAME_FROM_TRASH                |
                 |                               |
                 v                               |
     获取镜像的初始元数据，如大小、对象前缀和特性。       |
           V2_GET_INITIAL_METADATA               |
                 |                               |
                 v                               |
           获取条带单元大小、数量
           V2_GET_STRIPE_UNIT_COUNT (skip if     |
                 |                   disabled)   |
                 v                               |
           获取镜像创建时的时间戳
           V2_GET_CREATE_TIMESTAMP               |
                 |                               |
                 v                               |
           获取镜像的访问和修改时间戳
           V2_GET_ACCESS_MODIFIY_TIMESTAMP       |
                 |                               |
                 v                               |
           获取镜像所在pool信息
           V2_GET_DATA_POOL --------------> REFRESH 使用获取的元数据刷新镜像状态
                                                 |
                                                 v
                                            初始化插件注册表
                                            INIT_PLUGIN_REGISTRY
```

```
                              |
                              v
                          初始化缓存
                         INIT_CACHE
                              |
                              v
                          注册镜像更改监视
                         REGISTER_WATCH (skip if
                              |              read-only)
                              v
                          设置快照,没有指定快照就跳过
                         SET_SNAP (skip if no snap)
                              |
                              v
                         <finish>
                              ^
    (on error)                |
    * * * * * * > CLOSE ------------------------/

```

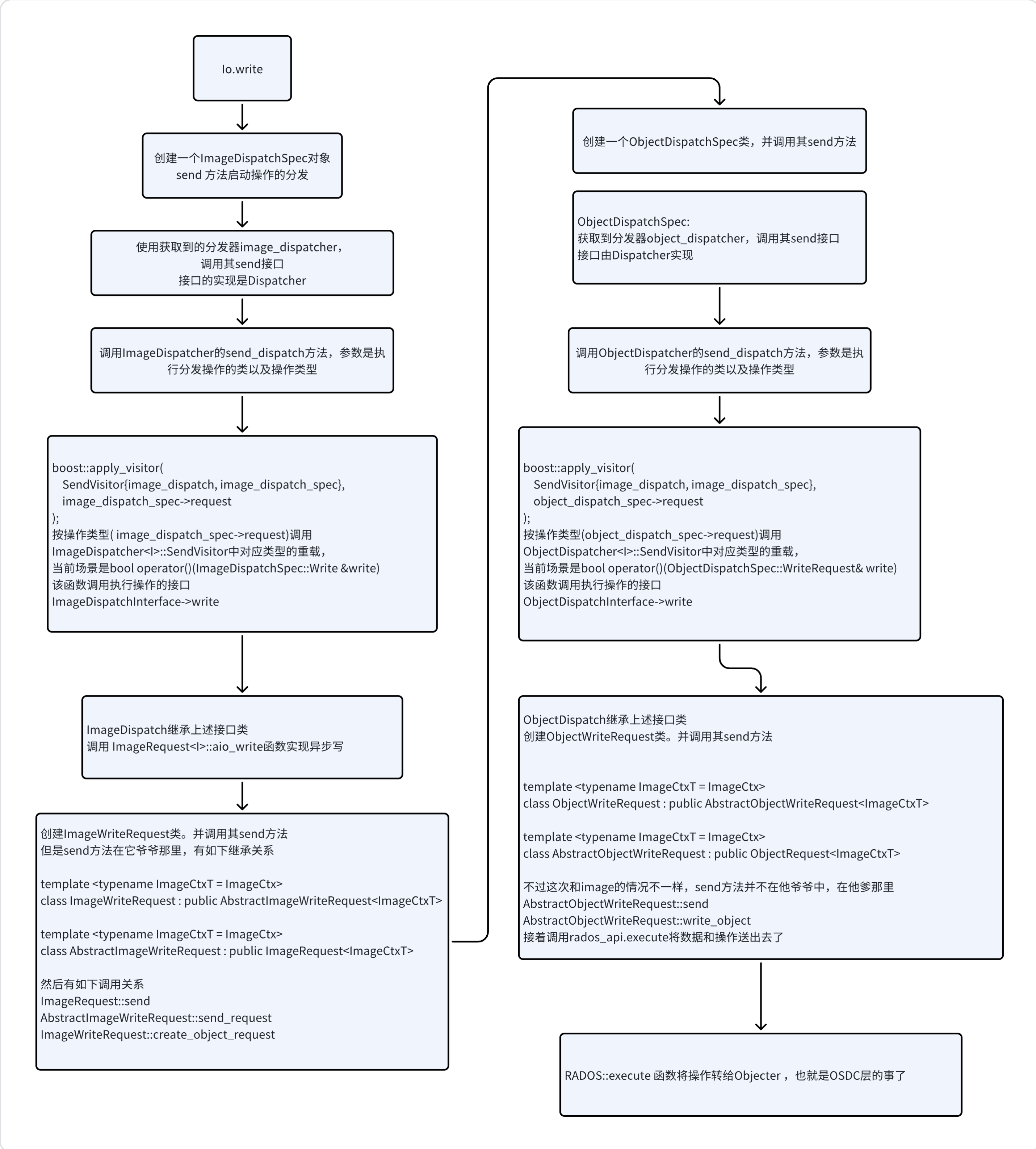
写数据

类图



总流程

站在顶层来看，分发的过程比较简单：



API调用

```
ssize_t Image::write(uint64_t ofs, size_t len, bufferlist &bl) {
    ImageCtx *ictx = (ImageCtx *)ctx;

    int r = api::Io<>::write(*ictx, ofs, len, bufferlist{bl}, 0);

    return r;
}
```

API实现

```
template <typename ImageCtxT = ImageCtx>
    struct Io {
    static ssize_t read(ImageCtxT &image_ctx, uint64_t off, uint64_t len,
                        io::ReadResult &&read_result, int op_flags);
    static ssize_t write(ImageCtxT &image_ctx, uint64_t off, uint64_t len,
                         bufferlist &&bl, int op_flags);
    ...
} // namespace librbd::api
```

```
template <typename I>
ssize_t Io<I>::write(
    I &image_ctx, uint64_t off, uint64_t len, bufferlist &&bl, int op_flags) {
    auto cct = image_ctx.cct;

    C_SaferCond ctx;
    auto aio_comp = io::AioCompletion::create(&ctx);
    aio_write(image_ctx, aio_comp, off, len, std::move(bl), op_flags, false);

    r = ctx.wait();

    return len;
}

template <typename I>
void Io<I>::aio_write(I &image_ctx, io::AioCompletion *aio_comp, uint64_t off,
                      uint64_t len, bufferlist &&bl, int op_flags,
                      bool native_async) {
    auto req = io::ImageDispatchSpec::create_write(
        image_ctx, io::IMAGE_DISPATCH_LAYER_API_START, aio_comp,
        {{off, len}}, io::ImageArea::DATA, std::move(bl), op_flags, trace);
    req->send();
}
```

ImageDispatchSpec

```
ImageDispatchSpec(ImageDispatcherInterface *image_dispatcher,
                  ImageDispatchLayer image_dispatch_layer,
```

```
                     AioCompletion *aio_comp, Extents &&image_extents,
                     ImageArea area, Request &&request, IOContext io_context,
                     int op_flags, const ZTracer::Trace &parent_trace)
    : dispatcher_ctx(this), image_dispatcher(image_dispatcher),
      dispatch_layer(image_dispatch_layer), aio_comp(aio_comp),
      image_extents(std::move(image_extents)), request(std::move(request)),
      io_context(io_context), op_flags(op_flags), parent_trace(parent_trace) {}

// ImageDispatcherInterface->send
void ImageDispatchSpec::send() {
    image_dispatcher->send(this);
}

```

Dispatcher

```
template <typename ImageCtxT, typename DispatchInterfaceT>
class Dispatcher : public DispatchInterfaceT {

    void send(DispatchSpec *dispatch_spec) {
        auto dispatch_layer = dispatch_spec->dispatch_layer;

        while (true) {
            m_lock.lock_shared();
            dispatch_layer = dispatch_spec->dispatch_layer;

// std::map<DispatchLayer, DispatchMeta> m_dispatches;
            auto it = m_dispatches.upper_bound(dispatch_layer);
            if (it == m_dispatches.end()) {
                // the request is complete if handled by all layers
                dispatch_spec->dispatch_result = DISPATCH_RESULT_COMPLETE;
                m_lock.unlock_shared();
                break;
            }

            auto &dispatch_meta = it->second;
            auto dispatch = dispatch_meta.dispatch;
            auto async_op_tracker = dispatch_meta.async_op_tracker;
            dispatch_spec->dispatch_result = DISPATCH_RESULT_INVALID;

            // prevent recursive locking back into the dispatcher while
handling IO
            async_op_tracker->start_op();
            m_lock.unlock_shared();

```

```
            // advance to next layer in case we skip or continue
            dispatch_spec->dispatch_layer = dispatch->get_dispatch_layer();

// 当前情景下 dispatch = ImageDispatchInterface
// dispatch_spec = ImageDispatchSpec::Write
            bool handled = send_dispatch(dispatch, dispatch_spec);
            async_op_tracker->finish_op();

            // handled ops will resume when the dispatch ctx is invoked
            if (handled) {
                return;
            }
        }

        // skipped through to the last layer
        dispatch_spec->dispatcher_ctx.complete(0);
    }
}
```

ImageDispatchSpec

用于处理分发

```
class ImageDispatchSpec {
    // 设定各种操作，比如read，write，CompareAndWrite，Flush等
    struct Read {
        ReadResult read_result;
        int read_flags;

        Read(ReadResult &&read_result, int read_flags)
            : read_result(std::move(read_result)), read_flags(read_flags) {
        }
    };
    struct Write {
        bufferlist bl;

        Write(bufferlist &&bl) : bl(std::move(bl)) {
        }
    };

    // Request是全部可能操作的数据集合，调度时，根据数据类型分发操作
    typedef boost::variant<Read,
                           Discard,
                           Write,
                           WriteSame,
```

```
                           CompareAndWrite,
                           Flush,
                           ListSnaps> Request;

    // 比如创建一个Write方法时，会创建一个Write{std::move(bl)}，并赋值给Request
    template <typename ImageCtxT = ImageCtx>
    static ImageDispatchSpec *create_write(
        ImageCtxT &image_ctx, ImageDispatchLayer image_dispatch_layer,
        AioCompletion *aio_comp, Extents &&image_extents, ImageArea area,
        bufferlist &&bl, int op_flags, const ZTracer::Trace &parent_trace) {
        return new ImageDispatchSpec(image_ctx.io_image_dispatcher,
                                     image_dispatch_layer, aio_comp,
                                     std::move(image_extents), area,
                                     Write{std::move(bl)},
                                     {}, op_flags, parent_trace);
    }
}
```

封装好操作后，ImageDispatcherInterface->send(ImageDispatchSpec)

在这里面又有 send_dispatch(ImageDispatchInterface, ImageDispatchSpec)

apply_visitor有两个参数，一个是SendVisitor，一个是image_dispatch_spec->request

ImageDispatcher：public Dispatcher

```
template <typename I>
bool ImageDispatcher<I>::send_dispatch(
    ImageDispatchInterface *image_dispatch,
    ImageDispatchSpec *image_dispatch_spec) {
    if (image_dispatch_spec->tid == 0) {
        image_dispatch_spec->tid = ++m_next_tid;

        bool finished = preprocess(image_dispatch_spec);
        if (finished) {
            return true;
        }
    }

    return boost::apply_visitor(
        SendVisitor{image_dispatch, image_dispatch_spec},
        image_dispatch_spec->request);
}
```

SendVisitor 结构如下， 它定义了很多个操作的转发方法，根据传入的request具体类型，由SendVisitor调用对应的operatot()函数

```
template <typename I>
struct ImageDispatcher<I>::SendVisitor : public boost::static_visitor<bool> {
    ImageDispatchInterface *image_dispatch;
    ImageDispatchSpec *image_dispatch_spec;

    SendVisitor(ImageDispatchInterface *image_dispatch,
                ImageDispatchSpec *image_dispatch_spec)
        : image_dispatch(image_dispatch),
          image_dispatch_spec(image_dispatch_spec) {}

    bool operator()(ImageDispatchSpec::Read &read) const {
        return image_dispatch->read(
            image_dispatch_spec->aio_comp,
            std::move(image_dispatch_spec->image_extents),
            std::move(read.read_result), image_dispatch_spec->io_context,
            image_dispatch_spec->op_flags, read.read_flags,
            image_dispatch_spec->parent_trace, image_dispatch_spec->tid,
            &image_dispatch_spec->image_dispatch_flags,
            &image_dispatch_spec->dispatch_result,
            &image_dispatch_spec->aio_comp->image_dispatcher_ctx,
            &image_dispatch_spec->dispatcher_ctx);
    }

    bool operator()(ImageDispatchSpec::Write &write) const {
        return image_dispatch->write(
            image_dispatch_spec->aio_comp,
            std::move(image_dispatch_spec->image_extents), std::move(write.bl),
            image_dispatch_spec->op_flags, image_dispatch_spec->parent_trace,
            image_dispatch_spec->tid, &image_dispatch_spec->image_dispatch_flags,
            &image_dispatch_spec->dispatch_result,
            &image_dispatch_spec->aio_comp->image_dispatcher_ctx,
            &image_dispatch_spec->dispatcher_ctx);
    }
};
```

这一步算调度完了，由具体的类执行操作

ImageDispatch : public ImageDispatchInterface

```
template <typename I>
```

```
bool ImageDispatch<I>::write(
    AioCompletion *aio_comp, Extents &&image_extents, bufferlist &&bl,
    int op_flags, const ZTracer::Trace &parent_trace,
    uint64_t tid, std::atomic<uint32_t> *image_dispatch_flags,
    DispatchResult *dispatch_result, Context **on_finish,
    Context *on_dispatched) {
    auto cct = m_image_ctx->cct;
    auto area = get_area(image_dispatch_flags);

    start_in_flight_io(aio_comp);

    *dispatch_result = DISPATCH_RESULT_COMPLETE;
    ImageRequest<I>::aio_write(m_image_ctx, aio_comp, std::move(image_extents),
                               area, std::move(bl), op_flags, parent_trace);
    return true;
}
```

ImageRequest

```
template <typename I>
void ImageRequest<I>::aio_write(I *ictx, AioCompletion *c,
                                Extents &&image_extents, ImageArea area,
                                bufferlist &&bl, int op_flags,
                                const ZTracer::Trace &parent_trace) {
    ImageWriteRequest<I> req(*ictx, c, std::move(image_extents), area,
                             std::move(bl), op_flags, parent_trace);
    req.send();
}
```

ImageWriteRequest

```

template <typename ImageCtxT = ImageCtx>
class ImageWriteRequest : public AbstractImageWriteRequest<ImageCtxT> {
public:
    ImageWriteRequest(ImageCtxT &image_ctx, AioCompletion *aio_comp,
                      Extents &&image_extents, ImageArea area, bufferlist &&bl,
                      int op_flags, const ZTracer::Trace &parent_trace)
        : AbstractImageWriteRequest<ImageCtxT>(
              image_ctx, aio_comp, std::move(image_extents), area,
```

```
                "write", parent_trace),
            m_bl(std::move(bl)), m_op_flags(op_flags) {
    }
}

template <typename ImageCtxT = ImageCtx>
class AbstractImageWriteRequest : public ImageRequest<ImageCtxT> {}

template <typename I>
void ImageRequest<I>::send() {
    I &image_ctx = this->m_image_ctx;
    ceph_assert(m_aio_comp->is_initialized(get_aio_type()));
    ceph_assert(m_aio_comp->is_started());

    CephContext *cct = image_ctx.cct;
    AioCompletion *aio_comp = this->m_aio_comp;

    update_timestamp();
    send_request();
}
```

AbstractImageWriteRequest

```
template <typename I>
void AbstractImageWriteRequest<I>::send_request() {
    I &image_ctx = this->m_image_ctx;
...
    AioCompletion *aio_comp = this->m_aio_comp;

    aio_comp->set_request_count(object_extents.size());

    send_object_requests(object_extents, image_ctx.get_data_io_context(),
journal_tid);
...
}

template <typename I>
void AbstractImageWriteRequest<I>::send_object_requests(
    const LightweightObjectExtents &object_extents, IOContext io_context,
    uint64_t journal_tid) {
    I &image_ctx = this->m_image_ctx;
    CephContext *cct = image_ctx.cct;

    AioCompletion *aio_comp = this->m_aio_comp;
    bool single_extent = (object_extents.size() == 1);
```

```
    for (auto &oe : object_extents) {

        C_AioRequest *req_comp = new C_AioRequest(aio_comp);
        auto request = create_object_request(oe, io_context, journal_tid,
                                             single_extent, req_comp);
        request->send();
    }
}

template <typename I>
ObjectDispatchSpec *ImageWriteRequest<I>::create_object_request(
    const LightweightObjectExtent &object_extent, IOContext io_context,
    uint64_t journal_tid, bool single_extent, Context *on_finish) {
    I &image_ctx = this->m_image_ctx;

    bufferlist bl;
    if (single_extent && object_extent.buffer_extents.size() == 1 &&
        m_bl.length() == object_extent.length) {
        // optimization for single object/buffer extent writes
        bl = std::move(m_bl);
    } else {
        assemble_extent(object_extent, &bl);
    }

    auto req = ObjectDispatchSpec::create_write(
        &image_ctx, OBJECT_DISPATCH_LAYER_NONE, object_extent.object_no,
        object_extent.offset, std::move(bl), io_context, m_op_flags, 0,
        std::nullopt, journal_tid, this->m_trace, on_finish);
    return req;
}
```

从写image转变为了写Object

ObjectDispatchSpec

```
template <typename ImageCtxT>
static ObjectDispatchSpec *create_write(
    ImageCtxT *image_ctx, ObjectDispatchLayer object_dispatch_layer,
    uint64_t object_no, uint64_t object_off, ceph::bufferlist &&data,
    IOContext io_context, int op_flags, int write_flags,
    std::optional<uint64_t> assert_version, uint64_t journal_tid,
    const ZTracer::Trace &parent_trace, Context *on_finish) {
    return new ObjectDispatchSpec(image_ctx->io_object_dispatcher,
```

```
                                   object_dispatch_layer,
                                   WriteRequest{object_no, object_off,
                                                std::move(data), write_flags,
                                                assert_version, journal_tid},
                                   io_context, op_flags, parent_trace,
                                   on_finish);
}
```

这一套的调度思想都是一致的，现在直接看结果：

```
template <typename I>
bool ObjectDispatch<I>::write(
    uint64_t object_no, uint64_t object_off, ceph::bufferlist&& data,
    IOContext io_context, int op_flags, int write_flags,
    std::optional<uint64_t> assert_version,
    const ZTracer::Trace &parent_trace, int* object_dispatch_flags,
    uint64_t* journal_tid, DispatchResult* dispatch_result,
    Context** on_finish, Context* on_dispatched) {
  auto cct = m_image_ctx->cct;

  *dispatch_result = DISPATCH_RESULT_COMPLETE;
  auto req = new ObjectWriteRequest<I>(m_image_ctx, object_no, object_off,
                                       std::move(data), io_context, op_flags,
                                       write_flags, assert_version,
                                       parent_trace, on_dispatched);
  req->send();
  return true;
}
```

```
template <typename I>
void AbstractObjectWriteRequest<I>::write_object() {
  I *image_ctx = this->m_ictx;

  neorados::WriteOp write_op;
  if (m_copyup_enabled) {
    if (m_guarding_migration_write) {
      auto snap_seq = (this->m_io_context->write_snap_context() ?
          this->m_io_context->write_snap_context()->first : 0);

      cls_client::assert_snapc_seq(
        &write_op, snap_seq, cls::rbd::ASSERT_SNAPC_SEQ_LE_SNAPSET_SEQ);
```

```
    } else {
      write_op.assert_exists();
    }
  }

  add_write_hint(&write_op);
  add_write_ops(&write_op);
  ceph_assert(write_op.size() != 0);

  image_ctx->rados_api.execute(
    {data_object_name(this->m_ictx, this->m_object_no)},
    *this->m_io_context, std::move(write_op),
    librbd::asio::util::get_callback_adapter(
      [this](int r) { handle_write_object(r); }), nullptr,
      (this->m_trace.valid() ? this->m_trace.get_info() : nullptr));
}


void RADOS::execute(const Object& o, const IOContext& _ioc, WriteOp&& _op,
                    std::unique_ptr<WriteOp::Completion> c, version_t* objver,
                    const blkin_trace_info *trace_info) {
  auto oid = reinterpret_cast<const object_t*>(&o.impl);
  auto ioc = reinterpret_cast<const IOContextImpl*>(&_ioc.impl);
  auto op = reinterpret_cast<OpImpl*>(&_op.impl);
  auto flags = op->op.flags | ioc->extra_op_flags;
  ceph::real_time mtime;
  if (op->mtime)
    mtime = *op->mtime;
  else
    mtime = ceph::real_clock::now();

  ZTracer::Trace trace;
  if (trace_info) {
    ZTracer::Trace parent_trace("", nullptr, trace_info);
    trace.init("rados execute", &impl->objecter->trace_endpoint, &parent_trace);
  }

  trace.event("init");
  impl->objecter->mutate(
    *oid, ioc->oloc, std::move(op->op), ioc->snapc,
    mtime, flags,
    std::move(c), objver, osd_reqid_t{}, &trace);
  trace.event("submitted");
}
```

Objecter

到这一步，就由OSDC层接管了。OSDC

```
void mutate(const object_t& oid, const object_locator_t& oloc,
            ObjectOperation&& op, const SnapContext& snapc,
            ceph::real_time mtime, int flags,
            std::unique_ptr<Op::OpComp>&& oncommit,
            version_t *objver = NULL, osd_reqid_t reqid = osd_reqid_t(),
            ZTracer::Trace *parent_trace = nullptr) {
  Op *o = new Op(oid, oloc, std::move(op.ops), flags | global_op_flags |
                 CEPH_OSD_FLAG_WRITE, std::move(oncommit), objver,
                 nullptr, parent_trace);
  o->priority = op.priority;
  o->mtime = mtime;
  o->snapc = snapc;
  o->out_bl.swap(op.out_bl);
  o->out_handler.swap(op.out_handler);
  o->out_rval.swap(op.out_rval);
  o->out_ec.swap(op.out_ec);
  o->reqid = reqid;
  op.clear();
  op_submit(o);
}
```