1. 背景
   1. 读写请求分别进入自己的队列，即读和写分别进入读队列和写队列；
   2. 读队列中的请求由读线程处理，写队列中的请求由写线程处理；
2. 目标

总体来说，提升读写的性能，尽量避免队列中不相关的IO请求阻塞读写请求，具体来说：

* 1. ~~对于所有的读IO请求来说，依进入队列的先后顺序执行；~~这不保证读读相关；
  2. ~~对于所有的写IO请求来说，依进入队列的先后顺序执行；~~这不保证写写相关；
  3. 如果后进入队列的读IO与当前写队列中的写IO请求不存在相关性，则可以尽早满足该读IO；
  4. 如果后进入队列的读IO与当前写队列中的写IO请求存在相关性，则需要等待写IO请求执行完毕之后方可执行读IO；
  5. 如果后进入队列的写IO与当前读队列中的读IO请求不存在相关性，则可以尽早满足该读IO；
  6. 如果后进入队列的写IO与当前读队列中的读IO请求存在相关性，则需要等待读IO请求执行完毕之后方可执行写IO；
  7. 不保证读IO一定在它后面到达的IO请求执行之前被执行，也不保证写IO一定在它后面到达的IO请求执行之前被执行；
  8. 本设计中只考虑读写相关和写读相关；

1. ~~可能的影响~~
   1. ~~对于写多副本来说，写性能受IO分布影响，而IO分布会影响IO调度（读写分离实际上是一种IO调度），会出现一定的波动；~~（不过当前没有读写分离的实现中也有该问题，不可避免）
   2. ~~读请求具有抢占性质，优先级较高，可能会有大量的后进入的读请求抢占写请求，导致写请求迟迟不被处理，尤其是对于Client端同步IO来说，必须尽量避免这种情况~~；(因为读写分别在不同的线程，所以不会出现这样的问题)
2. 更多的考虑
   1. 对Replicate的考虑
      1. 对于replicate来说，先后在两个OSS上分别执行Read和Write阶段，在两个OSS上应该分别作为读请求和写请求；
   2. 对Reselect的考虑
      1. 对于reselect来说，应该作为写请求；
   3. 对Redo的考虑
      1. 对于Redo来说，无需进入冲突队列（和读请求不存在相关性，和写请求即使存在相关性也按照先到先执行的顺序执行即可），作为写请求对待；
   4. 对OSS端异步IO的考虑
      1. 如果有Write cache，在Write cache之前相当于是同步IO，调用返回就表示执行完毕，就可以调度下一个IO，但是如果直接走异步IO（没有Write cache的情况下），则IO执行完毕，需要等待异步IO的回调被调用之后才表示执行完毕，如果后续被调度的IO依赖于前面的IO完成，这个过程相对较长，所以推荐读写分离在开启Write cache的情况下使用；
3. 概要设计
   1. 系统中引入读写线程对RWIOWorkerPair，这对线程处理的是同一个Chunk集合的Chunk IO；
   2. 由ObjectManager将IO请求提交给相应的读写线程对；
   3. 维护三个队列：等待队列，Write请求队列，Read请求队列，作用分别如下：
      1. 等待队列率属于读写线程对，管理自IO相关请求以来的所有IO请求；
      2. Write请求队列存放所有待执行的Write请求，它率属于写线程；
      3. Read请求队列存放所有待执行的Read请求，它率属于读线程；
   4. 系统中引入IO相关性判断逻辑，用以判断新接收到的IO请求是否和读/写队列中正在排队的请求以及所有已经从读/写队列中取出执行但是尚未接收到响应的请求存在相关性；
   5. 工作流程：
      1. 读写线程对接收到ObjectManager提交的请求，如果是IO请求，则判断新接收到的IO请求是否和读/写队列中正在排队的请求以及所有已经从读/写队列中取出执行但是尚未接收到响应的请求存在相关性；
         1. 如果存在相关性，则将该IO请求及其后续到达的请求放入等待队列中；
         2. 如果不存在相关性，则读IO请求进入读队列，写IO请求进入写队列，且更新IO相关性判断逻辑；
      2. 读线程和写线程分别从读队列和写队列中取出IO请求并执行；
      3. 当一个IO请求执行完毕时，首先更新IO相关性判断逻辑，如果它因为和后续IO请求存在相关性而阻塞了后续IO请求，则通知解除阻塞，此时需要向ObjectManager提交一个“解除阻塞”的请求；
      4. 读写线程对接收到的“解除阻塞”的请求，清除阻塞标识，处理等待队列中的请求以及后续接收到的请求；
      5. 为了解除阻塞请求被正确的线程处理，该请求需要指定特定的Chunk信息；
   6. IO相关性判断逻辑
      1. IO相关性判断逻辑主要用于当读写线程对接收到ObjectManager提交的请求时判断新接收到的IO请求是否和读/写队列中正在排队的请求以及所有已经从读/写队列中取出执行但是尚未接收到响应的请求存在相关性;
      2. IO相关性判断逻辑的基本数据结构是一颗基于区间树的锁，树中的每一个节点表示一个Chunk的某个访问区间，以及该区间内已经被授予的执行权限（读或者写），读权限是可以共享的，写权限则是排他的，**在申请访问权限时（IO进入读/写队列时）同一个Chunk的具有相同权限的区间会被合并，在释放访问权限时（接收到IO响应）会从树中移除相应的区间(合并的话会有问题！！！只有两个相邻的或者具有交叠的区间才可能被合并，但是合并的时候还必须分别为两个区间维护引用计数，因为在IO完成的时候，更新授权树的时候，必须考虑每个区间的引用计数，如该区间的引用计数为2，该区间不应该被释放，总之合并可能会比较麻烦，所以最终采用不合并，为每个区间单独维护，对于多个IO包括相同[start, end)区间的情况，通过引用计数解决之**)；
      3. 当某一个IO请求（blockee）被多个IO请求（blocker）阻塞的情况的处理：在每个blocker中记录它所阻塞的blockee，同时在blockee中记录它被多少个IO阻塞，每当一个blocker完成的时候，减少blockee被阻塞的IO数目；
      4. 当某一个IO请求（blocker）阻塞多个IO请求（blockee）的情况的处理：在该blocker中记录所有被阻塞的blockee，并在blocker完成的时候，通知所有被阻塞的blockee，并减少blockee被阻塞的IO数目；
   7. 优化
      1. 该优化是针对**不考虑**读读相关和写写相关情况下的优化；
      2. 如果等待队列设置了阻塞标识，则后续IO请求到达时，在判断IO相关性的时候，分别要和已经从读/写队列中取出执行但是尚未接收到响应的请求、读/写队列中正在排队的请求、等待队列中处于阻塞状态的请求进行IO相关性判断；
      3. 在和等待队列中处于阻塞状态的请求进行相关性判断时，采用一一比较的方式（而非IO相关性判断逻辑）；
      4. 虽然可能存在大量IO抢在被阻塞IO之前执行，该优化不会导致被阻塞的IO被饿死，因为一旦IO相关性被解除就会立即发送“解除阻塞”请求；
   8. 进一步改进
      1. 该改进是针对**不考虑**读读相关和写写相关情况下的优化；
      2. 将等待队列放到IO相关性判断逻辑里面，针对每一个Chunk除了范围锁组成的授权树以外，还包括两个冲突队列；
      3. 读/写分别有自己的授权树，读请求到达时只需和写所在的授权树进行相关性检测，写请求到达时也只需要同读所在的授权树进行相关性检测；
      4. 读/写也分别有自己的冲突队列，对于读/写请求，存在IO相关性的时候，都必须分别进入读/写自己的冲突队列；
      5. 如果当前不存在IO相关，对于后续的IO请求如果和已经被授予访问权限的IO不存在IO相关的情况下，可以直接提交给读/写请求队列；
      6. 即使当前存在IO相关，对于后续的IO请求如果和已经被授予访问权限的IO以及处于冲突队列中的IO都不存在IO相关的情况下，无需进入冲突队列排队等候，直接提交给读/写请求队列；
      7. 在提交请求到读/写请求队列之前，先更新授权树；
      8. 当一个存在IO相关的IO请求执行完毕时，更新授权树，同时从冲突队列中取出相应的IO请求提交给读写队列；
      9. 当一个不存在IO相关性的IO请求执行完毕时，更新授权树；
   9. 出错处理
      1. 如果正在执行的某个请求与后续请求存在IO相关性，但是该请求执行过程发生错误，则针对不同的错误码进行处理，该如何处理？（无需特别处理）
4. 详细设计

本文的详细设计基于上述最终改进后的设计。

* 1. 线程模型：
     1. ObjectManager提交线程
        1. 提交请求给读写线程对；
        2. 对于每一个提交的请求，进行IO相关性判断，如果存在相关，则进入等待队列（这也是相关性判断逻辑的一部分），否则首先更新相关性判断逻辑，然后插入读/写队列(提交给StorageManager的读/写线程)；
     2. StorageManager读线程
        1. 依次处理读请求；
        2. 当请求执行完毕的时候，更新IO相关性逻辑，如果该IO和后续IO存在相关性，则通知后续IO(当前来说，一定是写IO，因为读IO阻塞写IO)，该IO与其已不存在相关性了，并且从等待队列中提交写请求到StorageManager的写队列；
     3. StorageManager写线程
        1. 依次处理写请求；
        2. 当请求执行完毕的时候，更新IO相关性逻辑，如果该IO和后续IO存在相关性，则通知后续IO(当前来说，一定是读IO，因为写IO阻塞读IO)，该IO与其已不存在相关性了，并且从等待队列中提交请求到StorageManager的读队列；
  2. 数据结构定义
     1. 被阻塞IO（blockee）和阻塞IO（blocker）

如果存在IO相关，则产生阻塞的IO被称为blocker，被阻塞的IO被称为blockee，抽象出来这两个结构主要用于阻塞和解除阻塞控制。

blockee中主要包含该blockee对应的OpCtx和阻塞它的blocker数目。

blocker中主要维护两项：有多少blocker共享该结构，以及所有被该blocker阻塞的blockee集合。

/\*IOs being blocked\*/

class Blockee {

public:

Blockee();

virtual ~Blockee();

private:

/\*original IO context\*/

OpCtx\* mCtx;

/\*number of blockers(including those IOs in ConflictQueue) block me\*/

ywb\_uint32\_t mBlockerNum;

};

/\*IOs block successive IO\*/

class Blocker {

public:

Blocker();

virtual ~Blocker();

private:

/\*

\* CANNOT have original IO OpCtx here for there maybe many OpCtxs

\* share the same byte range

\* \*/

/\*OpCtx\* mCtx;\*/

/\*accommodate all blockees blocked by me\*/

std::set<Blockee\*> mBlockedCtxes;

};

* + 1. 授权树
       1. V1版本（Chunk级别相关性检测）

class AuthorizationTreeV1 {

public:

AuthorizationTreeV1() : mIOInProgress(ywb\_false) {};

virtual ~AuthorizationTreeV1() {};

/\*called when authorization\*/

void Authorize(OpCtx\* ctx);

/\*called when un-authorization\*/

void UnAuthorized(OpCtx\* ctx);

protected:

/\*return true if there is IO conflict\*/

ywb\_bool\_t CheckConflict(OpCtx\* ctx);

/\*called when conflict resolved, blocked IO submission is expected\*/

void OnConflictResolved(OpCtx\* ctx);

private:

/\*

\* whether there is any in-progress IO, set when IO is submitted

\* to read/write queue of IOWorker, and clear when IO is done

\* \*/

ywb\_bool\_t mIOInProgress;

};

* + - 1. V2版本（字节级别相关性检测）

class IntervalNodeExtent {

public:

IntervalNodeExtent();

virtual ~IntervalNodeExtent();

private:

ywb\_uint32\_t mStart;

ywb\_uint32\_t mEnd;

};

class IntervalNode {

public:

IntervalNode() {};

virtual ~IntervalNode() {};

private:

IntervalNode\* mLeft;

IntervalNode\* mRight;

IntervalNode\* mParent;

/\*FIXME: anything else?\*/

IntervalNodeExtent mExtent;

/\*

\* number of OpCtx share the same byte range, notify all blockees

\* blocked by me only if @mSharedCtxNum decrease to 0

\* \*/

ywb\_uint32\_t mSharedCtxNum;

/\*make it be a pointer to save memory if there is no conflict\*/

Blocker\* mBlocker;

};

/\*interval tree for byte range authorization control\*/

typedef IntervalNode\* IntervalTree;

/\*byte level authorization management\*/

class AuthorizationTreeV2 {

public:

AuthorizationTreeV2() {};

virtual ~AuthorizationTreeV2() {};

/\*called when authorization\*/

void Authorize(OpCtx\* ctx);

/\*called when un-authorization\*/

void UnAuthorize(OpCtx\* ctx);

protected:

/\*return true if there is IO conflict\*/

ywb\_bool\_t CheckConflict(OpCtx\* ctx);

/\*called when conflict resolved, blocked IO submission is expected\*/

void OnConflictResolved(OpCtx\* ctx);

private:

IntervalTree mTree;

};

* + - 1. 抽象版本（字节级别相关性检测）

#if WITH\_BYTE\_LEVEL\_AUTHORIZATION\_CONTROL

typedef AuthorizationTreeV2 AuthorizationTree;

#else

typedef AuthorizationTreeV1 AuthorizationTree;

#endif

* + 1. 冲突队列

typedef Queue<Blockee> ConflictQueue;

* + 1. IO相关控制单元

/\*manage authorization of single chunk\*/

class AuthorizationUnit {

public:

AuthorizationUnit() {};

virtual ~AuthorizationUnit() {};

/\*

\* check conflict and submit to read/write queue if no conflict,

\* otherwise wait in conflict queue

\* \*/

ywb\_status\_t Authorize(OpCtx\* ctx);

/\*update authorization tree and submit blocked request if necessary\*/

ywb\_status\_t UnAuthorize(OpCtx\* ctx);

private:

/\*authorization tree for read\*/

AuthorizationTree mRTree;

/\*authorization tree for write\*/

AuthorizationTree mWTree;

/\*conflict queue for read\*/

ConflictQueue mRQueue;

/\*conflict queue for write\*/

ConflictQueue mWQueue;

/\*these locks can be ignored if AuthorizationManager has its own lock\*/

// /\*protect @mRTree and @mRQueue\*/

// Mutex mRLock;

// /\*protect @mWTree and @mWQueue\*/

// Mutex mWLock;

};

* + 1. IO相关控制器

/\*manage authorization of all chunks belongs to the same RWIOWorkerPair\*/

class AuthorizationManager {

public:

AuthorizationManager() {};

virtual ~AuthorizationManager() {};

ywb\_status\_t Authorize(OpCtx\* ctx);

ywb\_status\_t UnAuthorize(OpCtx\* ctx);

private:

/\*

\* theoretically no lock protection is possible if we make sure

\* ObjectManager's submitting thread handle both construction

\* and de-construction of AuthorizationUnit itself, but meanwhile

\* there must be some mechanism that guarantee those stale

\* AuthorizationUnit be de-constructed, unfortunately no such!

\*

\* so a compromising solution is lots of latches are used with

\* each one manage a subset of AuthorizationUnits respectively

\* \*/

Mutex mLatches[DEFAULT\_LATCHES\_NUM];

/\*key: chunk id\*/

std::map<ywb\_uint64\_t, AuthorizationUnit\*> mAuthorization;

};

* + 1. RWIOWorkerPair

/\*managed by StorageManagerRequestHandler::IOWorkerGroup\*/

class RWIOWorkerPair {

public:

RWIOWorkerPair() {};

virtual ~RWIOWorkerPair() {};

/\*called by submitting thread of ObjectManager\*/

ywb\_status\_t Submit(OpCtx\* ctx);

/\*called when IO completed\*/

void Complete(OpCtx\* ctx);

private:

/\*IOWorker for read\*/

StorageManagerRequestHandler::IOWorker\* mRIOWorker;

/\*IOWorker for write\*/

StorageManagerRequestHandler::IOWorker\* mWIOWorker;

AuthorizationManager mAuthorization;

};

* + 1. 其它需要修改的数据结构

OpCtx：在V2版本中，为了在IO完成之后能够快速找到对应的IntervalNode， 需要在OpCtx中记录该IntervalNode的指针。

IOWorkerGroup：IOWorkerGroup直接管理的是RWIOWorkerPair，而非IOWorker。

* 1. 接口定义

暂无。

* 1. 关键流程
     1. ObjectManager提交线程提交IO请求时的处理
        1. V1版本（Chunk级别相关性检测）



* + - 1. V2版本（字节级别相关性检测）

暂无。

* + 1. 读/写线程中某个IO请求执行完毕时的处理



1. 疑问
   1. 区间锁的粒度以字节为单位，而非page为单位，在AIO只有Redo使用的情况下是没有问题的，但是其他情况会不会出现问题，待考证！