Curve支持S3 数据缓存方案

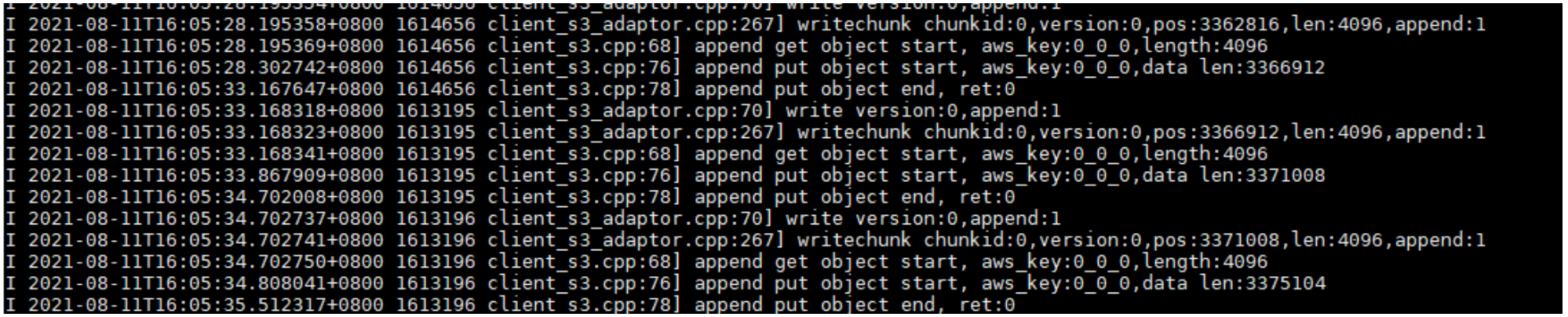
版本	时间	修改者	修改内容
1.0	2021/8/18	胡遥	初稿

背景

基于s3的daemon版本基于基本的性能测试发现性能非常差。具体数据如下：

```
huyao@pubbeta1-nostest2:~/mnt$ fio -bs=4k -direct=1 --fallocate=none -size=10M -iodepth=1 -filename=hello2 -rw=write -ioengine=libaio -numjobs=1 -name=test2
test2: (g=0): rw=write, bs=4K-4K/4K-4K/4K-4K, ioengine=libaio, iodepth=1
fio-2.16
Starting 1 process
test2: Laying out IO file(s) (1 file(s) / 10MB)
Jobs: 1 (f=1): [W(1)] [30.0% done] [0KB/4KB/0KB /s] [0/1/0 iops] [eta 19m:57s]
```

通过日志初步分析有2点原因



1.append接口目前采用先从s3 get，在内存中合并完后再put的方式，对s3操作过多
2.对于4k 小io每次都要和s3交互，导致性能非常差。
因此需要通过Cache模块解决以上2个问题。

整体设计

整个dataCache的设计思路，在写场景下能将数据尽可能的合并后flush到s3上，在读场景上，能够预读1个block大小，减少顺序读对于底层s3的访问频次。从这个思路上该缓存方案主要针对的场景是顺序写和顺序读，而对于随机写和随机读来说也会有一定性能提升，但效果可能不会太好。

元数据采用2层索引

由于chunk大小是固定的（默认64M），所以Inode中采用map<uint64, S3ChunkInfoList>
s3ChunkInfoMap用于保存对象存储的位置信息。采用2级索引的好处是，根据操作的offset可以快速定位到index，则只需要遍历index相关的S3ChunkInfoList，减少了遍历的范围。

对象名设计

对象名采用chunkId+blockindex+compaction（后台碎片整理才会使用，默认0）+inodeId。增加inodeId的目的是为了后续从对象存储上遍历，反查文件，这里就要求inodeId是永远不可重复。

读写缓存分离

读写缓存的设计采用的是读写缓存分离的方案。即读写缓存相互没影响不相关，写缓存一旦flush即释放，读缓存采用可设置的策略进行淘汰（默认LRU），对于小io进行block级别的预读。

缓存层级

缓存层级分为fs->file->chunk->datacache 4层，通过inodeId找到file，通过index找到chunk，然后通过offset~len找到是否有合适的datacache或者new datacache。

对外接口

流程上对于读写缓存有影响的接口包括：write，read，releaseCache，flush，fssync。后面会详细介绍这些接口流程。这里不需要提供truncate接口，可以由client直接修改inode的len，由metaserver的碎片整理（马杰负责）模块进行truncate的无效数据清理

后台刷数据线程

启动后台线程，将写Cache定时刷到S3上，同时通过inodeManager更新inode缓存中的s3InfoList。具体细节见

本地磁盘缓存

如果有配置writeBack dev，则会调用diskStroage进行本地磁盘write，最终写到s3则由diskStroage模块决定。

关键数据结构

```
message S3ChunkInfo {
    required uint64 chunkId = 1;
    required uint64 compaction = 2;
    required uint64 offset = 3;
    required uint64 len = 4;  // file logic length
    required uint64 size = 5; // file size in object storage
};

message Inode {
    required uint64 inodeId = 1;
    required uint32 fsId = 2;
    required uint64 length = 3;
    required uint32 ctime = 4;
    required uint32 mtime = 5;
    required uint32 atime = 6;
    required uint32 uid = 7;
    required uint32 gid = 8;
    required uint32 mode = 9;
    required sint32 nlink = 10;
    required FsFileType type = 11;
    optional string symlink = 12;   // TYPE_SYM_LINK only
    optional VolumeExtentList volumeExtentList = 13;  // TYPE_FILE only
    map<uint64, S3ChunkInfoList> s3ChunkInfoMap = 14; // TYPE_S3 only, first is chunk index
    optional uint64 version = 15;
}

class ClientS3Adaptor {
 public:
    ClientS3Adaptor () {}
    void Init(const  S3ClientAdaptorOption  option, S3Client *client,
              std::shared_ptr inodeManager);
```

```
    int Write(Inode *inode, uint64_t offset,
              uint64_t length, const char* buf bool di);
    int Read(Inode *inode, uint64_t offset,
              uint64_t length, char* buf);
   int ReleaseCache(uint64_t inodeId);
   int Flush(Inode *inode);
   int FsSync();
   uint64_t GetBlockSize() {return blockSize_;}
   uint64_t GetChunkSize() {return chunkSize_;}
   CURVEFS_ERROR AllocS3ChunkId(uint32_t fsId);
   CURVEFS_ERROR GetInode(uint64_t inodeId, Inode *out);
private:
   S3Client *client_;
   uint64_t blockSize_;
   uint64_t chunkSize_;
   std::string metaServerEps_;
   std::string allocateServerEps_;
   Thread bgFlushThread_;
   std::atomic toStop_;
   std::shared_ptr fsCacheManager_;
   std::shared_ptr inodeManager_;
};


class S3ClientAdaptor;
class ChunkCacheManager;
class FileCacheManager;
class FsCacheManager;
using FileCacheManagerPtr = std::shared_ptr;
using ChunkCacheManagerPtr = std::shared_ptr;
using DataCachePtr = std::shared_ptr;
class FsCacheManager {
 public:
    FsCacheManager() {}
    FileCacheManagerPtr FindFileCacheManager(uint32_t fsId, uint64_t inodeId);
    void ReleaseFileCahcheManager(uint32_t fdId, uint64_t inodeId);
    FileCacheManagerPtr GetNextFileCacheManager();
    void InitMapIter();
```

```
    bool FsCacheManagerIsEmpty();
 private:
    std::unordered_map  fileCacheManagerMap_;  // first is inodeid
    std::unordered_map ::iterator fileCacheManagerMapIter_;
    RWLock rwLock_;
    std::list  lruReadDataCacheList;
    uint64_t lruMaxSize;
    std::atomic dataCacheNum_;
};

class FileCacheManager {
 public:
    FileCacheManager(uint32_t fsid, uint64_t inode) : fsId_(fsid), inode_(inode) {}
    ChunkCacheManagerPtr FindChunkCacheManager(uint64_t index);
    void ReleaseChunkCacheManager(uint64_t index);
    void ReleaseCache();
    CURVEFS_ERROR Flush();
 private:
    uint64_t fsId_;
    uint64_t inode_;
    std::map chunkCacheMap_;  // first is index
    RWLock rwLock_;
};

class ChunkCacheManager {
 public:
    ChunkCacheManager(uint64_t index) : index_(index) {}
    DataCachePtr NewDataCache(S3ClientAdaptor *s3ClientAdaptor, uint32_t chunkPos, uint32_t len, const char
*dataCacheType type);
    DataCachePtr FindWriteableDataCache(uint32_t pos, uint32_t len);
    CURVEFS_ERROR Flush();
 private:
    uint64_t index_;
    std::map dataWCacheMap_; // first is pos in chunk
    curve::common::Mutex wMtx_;
    std::map dataRCacheMap_; // first is pos in chunk
};
```

```
class DataCache {
 public:
    DataCache(S3ClientAdaptor *s3ClientAdaptor, ChunkCacheManager* chunkCacheManager, uint32_t chunkPos, uint32_t
len, const char *data)
            : s3ClientAdaptor_(s3ClientAdaptor), chunkCacheManager_(chunkCacheManager), chunkPos_(chunkPos),
len_(len) {
        data_ = new char[len];
        memcpy(data_, data, len);
    }
    virtual ~DataCache() {
        delete data_;
        data_ = NULL;
    }

    void Write(uint32_t cachePos, uint32_t len, const char* data);
    CURVEFS_ERROR Flush();
 private:
    S3ClientAdaptor *s3ClientAdaptor_;
    ChunkCacheManager* chunkCacheManager_;
    uint64_t chunkId;
    uint32_t chunkPos_;
```

```
    uint32_t len_;
    char* data_;
};
```

详细设计

Write流程

1.加锁，根据inode和fsid找到对应的fileCacheManager，如果没有则生成新的fileCacheManager，解锁，调用fileCacheManager的Write函数。
2.考虑到同一个client同一个文件同时只能一个线程进行文件写，所以在Write函数中加写锁。
3.根据请求offset，计算出对应的chunk index和chunkPos。将请求拆分成多个chunk的WriteChunk调用。
4.在WriteChunk内，根据index找到对应的ChunkCacheManager，根据请求的chunkPos和len从dataCacheMap中找到一个可写的DataCache：
4.1 chunkPos~len的区间和当前DataCache有交集（包括刚好是边界的情况）即可写。
4.2 同时计算后续的多个DataCache是否和chunkPos~len有交集，如果有则一并获取
5. 如果有可写的DataCache，则调用Write接口将数据合并到DataCache中；如果没有可写的DataCache则new一个，加入到ChunkCacheManager的Map中。
5.完成后返回成功。

Read流程

1.根据请求offset，计算出对应的chunk index和chunkPos。将请求拆分成多个chunk的ReadChunk调用。
2.在ReadChunk内，根据index找到对应的ChunkCacheManager，根据请求的chunkPos和len从dataCacheMap中找到一个可读的DataCache，由于DataCache都是最小粒度为blockSIze的缓存，所以存在3种情况：要读的chunkPos~len的区间全部被缓存，部分被缓存，以及无缓存。将缓存部分buf直接copy到接口的buf指针对应的偏移位置，无缓存部分生成requestVer。
3.遍历requestVer，根据每个request的offset找到inode中对应index的S3ChunkInfoList，根据S3ChunkInfoList构建s3Request，最后生成s3RequestVer。
4.遍历s3RequestVer中request采用异步接口读取数据。
5.等待所有的request返回，更新读缓存，获取返回数据填充readBuf。

ReleaseCache流程

1.由于删除采用异步的方式，所以对于delete操作仅仅需要释放client的cache缓存。这里同时要保证的一点是：上层确保该文件没有被打开，才能调用该接口，因此不用考虑cache被删除的同时又有人来增加或修改
2.根据inodeId找到对应FileCacheManager ，调用ReleaseCache接口，一层一层将缓存释放。

Flush流程

1.根据InodeId找到对应的FileCacheManager，执行Flush函数。
2.在Flush函数中，加写锁，通过swap获取FileCacheManager的chunkCacheMap_到临时变量tmp，并清空chunkCacheMap_，解写锁。遍历tmp中的DataCache列表，执行Flush函数，并更新对应的元数据。
3.Flush返回成功。
4.如果DataCache的Flush失败，则整个Flush失败。但是缓存需要重新回退到chunkCacheMap_中，这里要注意一点：回退的过程，如果chunkCacheMap_为空，则直接swap回退。如果chunkCacheMap_不为空，则表示Flush的过程中有新的cache加入，则需要进行合并，合并的规则是新的cache如果和老的cache有重叠则覆盖老的cache。

FsSync流程

1.循环获取FileCacheManager，执行Flush函数。

后台流程

1. 在FsCacheManager中增加一个DataCacheNum_字段，如果该字段为0，表示没有cache需要flush，则线程由条件变量控制处于wait状态。
2. write流程会对后台线程处于wait状态的情况触发notify唤醒，同时修改DataCacheNum_。
3. 后台会遍历DataCache，达到flushwait的时间，或者DataCache size为chunksize，满足一个条件则调用DataCache的flush。
4. 更新元数据，清理DataCache缓存，DataCacheNum_减1。
5. 遍历完一轮DataCache后，获取DataCacheNum值，如果不为0，则继续遍历，如果为0则回到1步骤。

poc测试验证

根据上述设计，完成初步daemon，测试结果如下图

```
huyao@pubbeta1-nostest2:~/mnt$ fio -bs=4k -direct=1 --fallocate=none -size=10M -iodepth=1 -filename=hello2 -rw=write -ioengine=libaio -numjobs=1 -name=test2
test2: (g=0): rw=write, bs=4K-4K/4K-4K/4K-4K, ioengine=libaio, iodepth=1
fio-2.16
Starting 1 process
test2: Laying out IO file(s) (1 file(s) / 10MB)
Jobs: 1 (f=1): [W(1)] [30.0% done] [0KB/4KB/0KB /s] [0/1/0 iops] [eta 19m:57s]
```

目前看写性能有明显的提升，但时延仍然很高，需要进一步分析。