
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
2.0	2022/8/4	胡遥	根据当前代码实现进行文档更新

- 背景
- 整体设计
 - 缓存设计
 - 文件到s3对象映射
 - 元数据采用2层索引
 - 对象名设计
 - 读写缓存分离
 - 对外接口
 - 后台刷数据线程
 - 本地磁盘缓存
- 关键数据结构
- 详细设计
 - Write流程
 - Read流程
 - ReleaseCache流程
 - Flush流程
 - FsSync流程
 - 后台流程

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

```

I 2021-08-11T16:05:28.195354+0800 1614656 client_s3_adaptor.cpp:70] write version:0,append:1
I 2021-08-11T16:05:28.195358+0800 1614656 client_s3_adaptor.cpp:267] writechunk chunkid:0,version:0,pos:3362816,len:4096,append:1
I 2021-08-11T16:05:28.195369+0800 1614656 client_s3.cpp:68] append get object start, aws_key:0_0_0,length:4096
I 2021-08-11T16:05:28.302742+0800 1614656 client_s3.cpp:76] append put object start, aws_key:0_0_0,data len:3366912
I 2021-08-11T16:05:33.167647+0800 1614656 client_s3.cpp:78] append put object end, ret:0
I 2021-08-11T16:05:33.168318+0800 1613195 client_s3_adaptor.cpp:70] write version:0,append:1
I 2021-08-11T16:05:33.168323+0800 1613195 client_s3_adaptor.cpp:267] writechunk chunkid:0,version:0,pos:3366912,len:4096,append:1
I 2021-08-11T16:05:33.168341+0800 1613195 client_s3.cpp:68] append get object start, aws_key:0_0_0,length:4096
I 2021-08-11T16:05:33.867909+0800 1613195 client_s3.cpp:76] append put object start, aws_key:0_0_0,data len:3371008
I 2021-08-11T16:05:34.702008+0800 1613195 client_s3.cpp:78] append put object end, ret:0
I 2021-08-11T16:05:34.702737+0800 1613196 client_s3_adaptor.cpp:70] write version:0,append:1
I 2021-08-11T16:05:34.702741+0800 1613196 client_s3_adaptor.cpp:267] writechunk chunkid:0,version:0,pos:3371008,len:4096,append:1
I 2021-08-11T16:05:34.702750+0800 1613196 client_s3.cpp:68] append get object start, aws_key:0_0_0,length:4096
I 2021-08-11T16:05:34.808041+0800 1613196 client_s3.cpp:76] append put object start, aws_key:0_0_0,data len:3375104
I 2021-08-11T16:05:35.512317+0800 1613196 client_s3.cpp:78] append put object end, ret:0

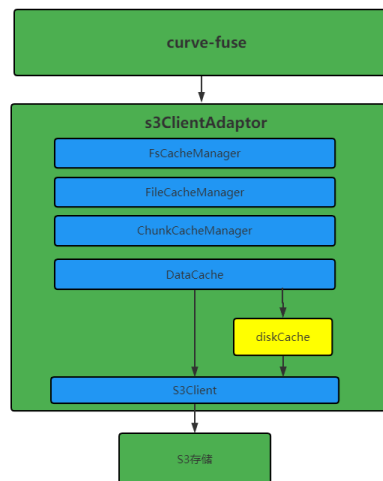
```

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的设计思路包括：

1. 为了快速查询到对应的缓存，整个缓存分为3层file->chunk->datacache。

2. 整个文件到s3对象的映射分为4层，file->chunk->datacach->block。

3. 文件对应s3元数据采用2层索引。

缓存设计

从上面的整体架构图来看，整个缓存分为3层，file->chunk->datacache 3层，通过inodeId找到file，通过offset计算出chunkindex找到chunk，然后通过offset~len找到是否有合适的datacache或者new

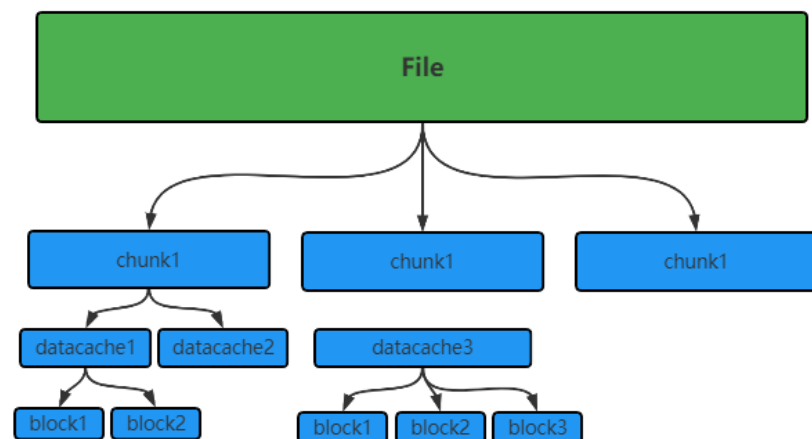
datacache。

这里FileCacheManager，ChunkCacheManager维护的缓存主要都是一张map表，只有真正到了dataCache这一层才涉及到真正的数据。

这里对chunk，block的概念在详细说明一下，当文件系统创建的时候会指定chunkSize，blockSize，这个时候chunk和block的大小就固定了，不可修改。因此一个文件的chunk和block也是固定的，任意一个offset的访问都可以通过这2个值计算到属于那个chunk，以及chunk内属于哪个block。注意这里block不是全局的index，而是chunk内的index（好像用全局的blockindex也是没问题的）

早期的datacache设计的是连续的内存，但是连续的内存存在多个dataCache的合并上会有大内存的申请和释放，耗时较大。目前已经采用page的方式存储在datacache中，每个page是64k，这样在datacache合并的时候只需要分配少量的内存，或者是对page的迁移就能达到合并的效果。所以datacache逻辑上是对应一段连续的file data，但是在内存中实际上是由许多不连续的page组成。

文件到s3对象映射



整个文件到s3对象的映射分为4层：file，chunk，datacache，block。file是由固定大小的chunk组成，chunk是由变长且连续的数据cache组成，datacache是由固定大小的block组成。所有datacache（这里指的是写datacache）没有交集（因为如果有，在write流程中就合并了，没有合并的必然是没有交集的）。在datacache flush的流程中，将数据写到s3上。考虑到datacache是连续的变长数据，而我们写到s3上是固定大小的对象，所以引入了block的概念。因此会根据datacache对应的chunk的便宜pos，计算block index，从而构成对象名，写到s3上。

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fsid+inodeId+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。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。这里读缓存的产生会有2种场景，第1是用户读请求s3产生的数据会加入到读缓存中，第2是写flush后的datacache会转化为读缓存。这里有一种情况，读缓存在新增datacache的时候，可能和老的读缓存会有重叠的部分，这种情况肯定是以新加入的读缓存为主，将老的读缓存删除掉

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，Truncate，增对cto场景又增加了FlushAllCache。后面会详细介绍这些接口流程。

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
    required uint64 zero = 6;
};

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_;

```



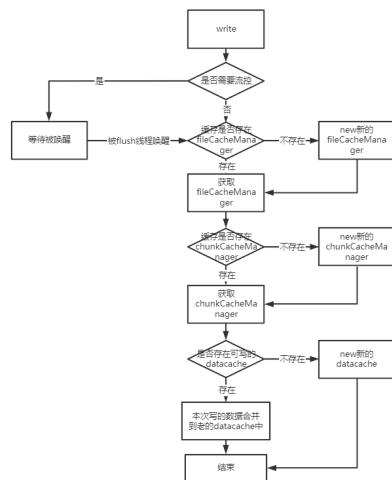
```

uint32_t len_;
char* data_;
};

```

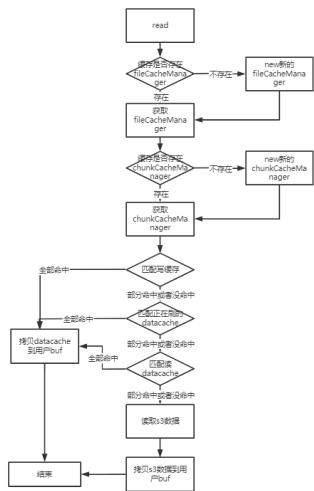
详细设计

Write流程



1. 流控，如果文件系统已使用的写缓存达到水位，则请求先等待。flush释放后会唤醒继续往下走。
2. 加写锁，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。
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，加ChunkCacheManager锁，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。
6. 释放ChunkCacheManager锁，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，加ChunkCacheManager读锁，根据请求的chunkPos和len从dataCacheMap中找到一个可读的DataCache。
- 2.1 由于做了缓存分离，这个时候一个chunkCacheManager实际上有3种datacache缓存存在：写datacache，flushing datacache，和读datacache，从生成的关系来看，写datacache肯定是最新的数据，其次是flushing datacache，在然后是读datacache。所以用户offset~len查找缓存的顺序是先去写datacache中查找，然后去flushing datacache中查找，最后去读datacache中查找。
- 2.2 查找的结果又会有存在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，主要在函数GenerateS3Request中实现
- 3.1 这里新加入了重试机制，一般情况如果s3那边读失败，则对于对象不存在，则需要一直重试，这里有2种
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. FileCacheManager::Flush处理流程。加写锁，获取FileCacheManager的chunkCacheMap拷贝到临时变量tmp，解写锁。这里对于chunkCacheManager的flush做了并发异步化的优化，遍历tmp中的ChunkCacheManager列表，将每个chunkCacheManager封装到FlushChunkCacheContext的一个task数组中，由线程池来并发处理不同chunkCacheManager的flush。
3. ChunkCacheManager::Flush的处理流程。
- 3.1 从写datacache中获取一个datacache赋值给flushingDataCache，并移出写datacache的map中（之前的做法是不移除，但是会出现已经在flush的datacache又合并了新data，导致s3重复数据过多的问题）
- 3.2 执行flushingDataCache的flush，如果成功则datacache转成读缓存，同时调用ReleaseWriteDataCache，更新写缓存相关计数。如果失败，则分为2种情况，一种是s3写失败，则一直退避方式重试，一种是inode获取不存在，则文件已被删，相关写缓存直接释放。
4. dataCache::Flush流程。
- 4.1 从mds上获取全局递增的chunkid来关联该datacache，用于chunkid越大表示该对象对应的数据越新。
- 4.2 异步的写s3或者diskcache。
- 4.3 更新对应s3info信息，添加到inode的s3infoList中。

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. 后台执行FsCacheManager::FsSync，最后调用的FileCacheManager::Flush，和flush流程保持一致