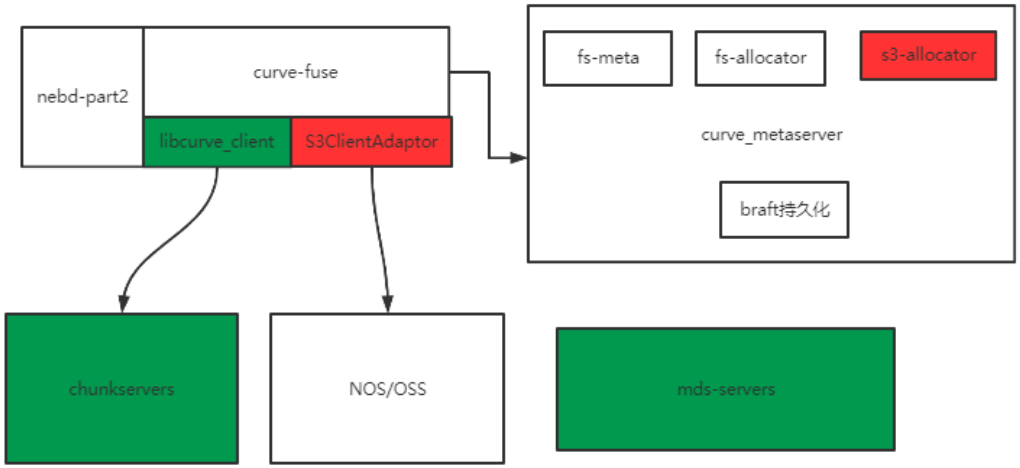
curvefs对接s3方案设计（过程文档）

时间	修订人	修订内容
2021-05-20	胡遥	初稿
2021-07-20	胡遥	细化write和read流程

整体架构



S3ClientAdaptor模块：负责将文件数据进行chunk，以及block的拆分为s3的object，并写入/读取s3的object。
S3-allocator模块：负责分配s3-object唯一标识。

整体思路

curvefs对接s3和对接volume主要的区别在于数据持久化和空间分配部分，而元数据的操作尽量保持统一。因此我们涉及到修改client的流程主要在read/write/flush，以及空间分配申请（s3不需要释放空间，可直接删除对应s3 object）

文件首先会按照chunk进行拆分，每个chunk固定64M/1G（待定），chunk内部会划分为多个block，每个block最大4M，每个block对应s3上一个object。

s3上对象已chunkid_indexblock_version进行命名，元数据则已S3ChunkInfo（见数据结构）的方式存储在inode中。对于文件顺序写场景，文件0~4M的s3对象必然为chunkid_0_0,4M~8M为chunkid_1_0，以此类推，还有一种情况是文件先写了0~2M，然后在写2M~4M，这里会采用append到同一个对象的方式进行写，而不是额外upload到一个新的对象；元数据则为{2,0,0,8M}。对于覆盖写，为了区分新老数据，则会对version进行++，比如覆盖写了0~4M，则数据会写到chunkid_0_1的对象，则元数据包含了2个S3Chunkinfo{2,0,0,8M}和{2,1,0,4M}。

接口和关键数据结构

common.proto

```
enum FSType {
    TYPE_VOLUME = 1;
    TYPE_S3 = 2;
}

message S3Info {
    required string ak = 1;
    required string sk = 2;
    required string endpoint = 3;
    required string bucketname = 4;
    required uint64 blockSize = 5;
    required uint64 chunkSize = 6;
}
```

mds.proto

```
import "curvefs/proto/common.proto";
message CreateFsRequest {
    required string fsName = 1;
    required uint64 blockSize = 2;
    required FSType fsType = 3;
    optional common.Volume volume = 4;
    optional common.S3Info s3Info = 5;
}

message FsInfo {
    required uint32 fsId = 1;
    required string fsName = 2;
    required FsStatus status = 3;
    required uint64 rootInodeId = 4;
    required uint64 capacity = 5;
    required uint64 blockSize = 6;
    required uint32 mountNum = 7;
    repeated MountPoint mountpoints = 8;
    required FSType fsType = 9;
    optional common.Volume volume = 10;
    optional common.S3Info s3Info = 11;
}
```

client端数据结构

```
// clients3
class S3ClientAdaptor {
 public:
    S3ClientAdaptor() {}
    void Init(const  S3ClientAdaptorOption  option, S3Client *client);
    int write(Inode *inode, uint64_t offset,
              uint64_t length, const char* buf);
    int read(Inode *inode, uint64_t offset,
              uint64_t length, char* buf);
 private:
   S3Client* client_;
   uint64_t blockSize_;
   uint64_t chunkSize_;
};

//s3
class S3Client {
 public:
    S3Client() {}
    virtual ~S3Client() {}
    virtual void Init(curve::common::S3AdapterOption option) = 0;
    virtual int Upload(std::string name, const char* buf, uint64_t length) = 0;
    virtual int Append(std::string name, const char* buf, uint64_t length) = 0;
    virtual int Download(std::string name, char* buf, uint64_t offset, uint64_t length) = 0;
};
```

metaserver.proto

```
enum FileType {
    TYPE_DIRECTORY = 1;
    TYPE_FILE = 2;
    TYPE_SYM_LINK = 3;
    TYPE_S3 = 4;
};
```

```
// inodes3chunk
message S3ChunkInfo {
    required uint64 chunkId = 1;
    required uint64 version = 2;
    required uint64 offset = 3;
    required uint64 len = 4;  // file logic length
    required uint64 size = 5; // file size in object storage
};

message S3ChunkInfoList {
    repeated S3ChunkInfo s3Chunks = 1;
};

message UpdateInodeS3VersionRequest {
    required uint64 inodeId = 1;
    required uint32 fsId = 2;
}

message UpdateInodeS3VersionResponse {
    required MetaStatusCode statusCode = 1;
    required uint64 version;
}

message UpdateInodeRequest {
    required uint64 inodeId = 1;
    required uint32 fsId = 2;
    optional uint64 length = 3;
    optional uint32 ctime = 4;
    optional uint32 mtime = 5;
    optional uint32 atime = 6;
    optional uint32 uid = 7;
    optional uint32 gid = 8;
    optional uint32 mode = 9;
    optional VolumeExtentList volumeExtentList = 10;  // TYPE_FILE only
    optional S3ChunkInfoList s3ChunkInfoList = 11; // TYPE_S3 only
}

message UpdateInodeResponse {
```

```
    required MetaStatusCode statusCode = 1;
}

service MetaServerService {
    rpc UpdateInodeS3Version(UpdateInodeS3VersionRequest) returns (UpdateInodeS3VersionResponse);
    rpc UpdateInode(UpdateInodeRequest) returns (UpdateInodeResponse);
}
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
```

```
    optional S3ChunkInfoList s3ChunkInfoList = 14; // TYPE_S3 only
    optional uint64 version = 15;
}
```

space相关数据结构和proto

```
enum AllocateType {
    NONE = 0;
    SMALL = 1;     //
    BIG = 2;       //
    S3 = 3;
}

message AllocateS3ChunkRequest {
    required uint32 fsId = 1;
}

message AllocateS3ChunkResponse {
   required SpaceStatusCode status = 1;  //
   required uint64 chunkId = 2;
}

service SpaceAllocService {
    // space interface
    rpc InitSpace(InitSpaceRequest) returns (InitSpaceResponse);
    rpc AllocateSpace(AllocateSpaceRequest) returns (AllocateSpaceResponse);
    rpc DeallocateSpace(DeallocateSpaceRequest) returns (DeallocateSpaceResponse);
    rpc StatSpace(StatSpaceRequest) returns (StatSpaceResponse);
    rpc UnInitSpace(UnInitSpaceRequest) returns (UnInitSpaceResponse);
    rpc AllocateS3Chunk(AllocateS3ChunkRequest) returns (AllocateS3ChunkResponse);
}

class S3Allocator {
 public:
    explicit S3Allocator(uint64_t startChunkId) : chunkId_(startChunkId) {}
```

```
    uint64_t NextChunkId() {
        auto id = chunkId_.fetch_add(1, std::memory_order_relaxed);
        return id;
    }

private:
    std::atomic<uint64_t> chunkId_;
```

```
};
```

关键流程

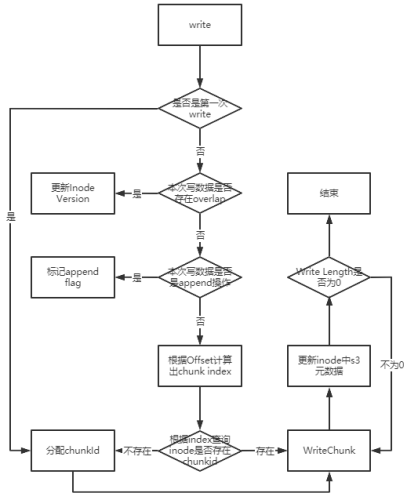
关键流程包括S3ClientAdaptor的init，write，read，delete和后台元数据整理以及数据回收流程

init流程

1.将conf中blockSize，chunkSize，metaServer和allocateServer ip保存在S3ClientAdaptor中

2.将conf中的S3相关信息：保罗ak，sk，s3address，bucketname等透传给S3Adapter模块。这里要注意，S3Adapter为原来curve块存储就有的模块，负责将快照数据上传到S3，这里我们对于s3的操作主要复用了这个模块。该模块使用的AWS的sdk，并没有实现append接口

write流程



主要流程逻辑见上面的流程图，对流程补充有以下几点：

1.对于overlap的场景，会将inode中的versio+1，但是不会处理被overlap的相关数据，由后台进行处理。
2.如果是带了append flag则在writechunk的时候会调用s3的append接口追加写到同一个block object。
3.更新inode中s3元数据的时候，现在只会将可以直接合并的S3Info进行了合并，后面需要考虑如果S3Info太大，需要进行rewrite将元数据进行重新合并
4.inode我们只更新s3Info，并不更新length，length由client在外面流程统一更新

read流程

1.read流程的难点在于，inode中存在的S3ChunkInfo情况的多样性，因此在read前，需要先对s3info信息进行优化，将overlap的chunk进行拆分，version大的覆盖version小的。最后得到的是一组没有overlap的chunks。

2.在将这些chunks按照offset进行大小进行排序，方便处理后面的read操作。

3.将read的offset，len和s3info可能交互的场景分别进行处理，分别获取要读取的每个S3ChunkInfo的offset len，封装到request中，具体可见代码的处理逻辑。

4.根据request进一步获取到s3 object去读取对象，将结果保存在response中。

5.最后根据所有的response将buff整合，返回给上层