Curve文件系统元数据管理（已实现）

1、设计一个分布式文件系统需要考虑的点：

1. 文件系统的元数据是否全缓存？
2. 元数据持久化在单独的元数据服务器上？在磁盘上？在volume上？
3. inode+dentry方式？当前curve块存储的kv方式？
4. 是否有单独的元数据管理服务器？

2、其他文件系统的调研总结

fs	中心化元数据	内存namespace元数据	内存空间分配元数据	元数据持久化	元数据扩展	小文件优化	空间管理单位	数据持久化	其他

moosefs（mfs）	有元数据服务器	全内存 fsnode → hashtable(inode id) fsedge → hashtable (parent inode + name)	全内存 chunk → hashtable(chunk id)	log + dump record	差	否	chunk	链式多副本	overwirte有数据不一致风险
chubaofs（cfs）	有元数据服务器	inode → b tree(key ino) dentry → b tree (key parentIno + name)	extent → B+ tree 这个在inode的ExtentsTree字段	meta partition(raft group) Btree、B+ tree	好	有 tiny extent，多个文件共用 normal extent，属于一个文件	partition	append→ master slave协议 overwrite → raft	更适合大文件顺序写
fastcfs	有元数据服务器	inode和dentry放一个结构体。 inode → hashtable（key是ino，全局） dentry → skip list （key是name，每个目录下一个）	计算出来的	binlog，随时间会越来越大	差		DG	Master/Slave	
glusterfs	无中心化服务器 dht算法 hash 扩展时大量迁移	client缓存 inode→ hashtable(gfid) dentry→ hashtable(name)	inode扩展属性字段	和写数据一样	好			写多份	overwirte有数据不一致风险
curve	有元数据服务器	lru cache缓存 kv → hashtable(key parent inode + name)	segment kv → hashtable(key inode + offset)	etcd	差	块设备，最小10GB	segment + chunk	raft	块设备的元数据管理
cephfs									

3、各内存结构体

	时间复杂度	空间复杂度	特点	可用实现
Btree			一个节点上保存多条数据，减少树的层次(4~5层)，方便从盘上读取数据，减少去盘上读取次数。适合在盘上和内存组织目录树。	google，https://github.com/abseil/abseil-cpp/tree/master/ontainer 实现了btree map和btree set，(Apache)。 google，https://code.google.com/archive/p/cpp-btree/，btr，btree_set, btree_multimap, and btree_multiset ，（Apach
B+tree			内部结点不保存数据，只有叶子结点保存数据。	https://github.com/begeekmyfriend/bplustree，（MIT），实现盘
BST	O(log(n))	O(n)		c++ stl 模板

skip list	O(log(n))	O(n)		level db，https://github.com/google/leveldb/blob/master/dlist.h，(BSD) libfastcommon，https://github.com/happyfish100/libfastcommee/master/src，(LGPL)
hash table	O(1)~O(n)	O(n) + table	需要占用额外空间，性能和hash表的大小有关，最理想可以达到O(1)复杂度，最差O(n)复杂度。	c++ stl unordered_map moose，使用c实现

4、curve文件系统的元数据内存组织

curve文件系统元数据主要有3个类型，inode， dentry，extent。

4.1 inode定义：

inode定义见：curve文件系统元数据proto（代码接口定义，已实现）

```
typedef uint64_t InodeId;

enum FileType {
    TYPE_DIRECTORY = 0,
    TYPE_FILE,
    TYPE_SYM_LINK,
};

class VolumeSpaceItem {
    uint64_t volumeOffset;    //
 uint64_t length; //
 bool used;         //
 // TODO:
};
class Inode {
 public:
    Inode();

 private:
    InodeId inode;
 uint64_t fsId;
    // uint32_t btime;     /* birth / create time */
    uint32_t ctime;     /* status change time */
    uint32_t mtime;     /* modify time */
    uint32_t atime;     /* access time */
    uint32_t uid;       /* user id *
    uint32_t gid;       /* group id */
 uint32_t mode;      /* file mode */
    int nlink;          /* ref count for hard link */
    int64_t size;
    FileType type;
    SpaceList std::map<uint64_t fsOffset, VolumeSpaceItem item>;  //
    string symLink;     /* for symLink only */
 // Inode
 ::curve::common::RWLock lock_;
};
```

4.2 dentry的定义：

dentry定义见：curve文件系统元数据proto（代码接口定义，已实现）

```
typedef uint64_t InodeId;

class Dentry{
 uint64_t fsId;
 InodeId parentId;
    InodeId inode;
    string name;
};
```

4.3 内存组织

inode和dentry的关系需要在内存中通过某种方式组织起来。

还需要额外考虑一下的hard link, symlink，rename的处理。

fastcfs的inode和dentry没有分开，两者在同一个结构体里面。这种方式如何应对硬链接？

看了下fastcfs的实现，在硬链接这里是有问题的。

考虑inode和dentry的内存组织形式，可以考虑hashmap，skiplist，btree等，但是无论选择哪种方式组织，节点都可以抽象成一个Key - Value的形式。

inode可以抽象成 key : fdid+inodeId, value : struct inode;

dentry可以抽象成 key : fsid+parentId+name , value : struct dentry;

分别从不同场景上进行分析，curve文件系统的元数据应该有以下的操作：

1、系统加载的时候，元数据从持久化介质加载。

2、业务运行过程中，元数据的增删改查。

3、系统退出的时候，元数据持久化。

场景一：系统加载的时候，元数据从持久化介质中加载。

元数据进行恢复的时候，有两种情况。

一种系统必须等到元数据全部加载到内存才能提供服务，这种情况下，元数据需要在内存全缓存。这种方式，对性能友好，但是需要消耗比较多的内存，元数据服务的扩展性受限于内存，而且在元数据服务启动的时候，需要等待一段时间加载内存。

一种是元数据需要全部加载到内存，这种情况下，元数据只需要加载一小部分主要的元数据，比如说super block这种，剩下的比如inode，dentry这种，按需加载，而且使用淘汰机制，内存中不常用的元数据可以淘汰出去。这种方式，扩展性好，元数据服务的扩展性不受限于内存，服务上的内存只有几百GB，而硬盘空间按照20块1.6TB的盘来计算，一个服务器上可以有32TB的空间，硬盘的空间比内存到100多倍。但是这种方式，由于数据不能去全部缓存到内存，在查询元数据的时候，需要去盘上读数据，而且在文件系统这种使用场景下，一次对文件的查找，需要在磁盘上读取多次。

我们的文件系统定位是一个高性能的通用文件系统，元数据的缓存倾向于全缓存。

系统加载的时候从持久化介质中进行加载，需要把一条条持久化的记录加载到内存里。实现把string转化为inode结构体，再插入内存结构中。

场景二：业务运行过程中，元数据的增删改查。

如果采用raft的方式对元数据持久化进行保证，所有元数据的处理都是先写WAL，再修改内存结构。那么任何对元数据的增删改查，对应着一条记录，根据记录去修改内存数据。

按照之前的讨论，curve文件系统的元数据管理采取先写log的方式。这里先不考虑log的组成形式。

那么curve文件系统的应该是先写log，log落盘之后，更新内存。

场景三：系统退出的时候，元数据的持久化

如果采用raft的方式对元数据持久化，任务数据的修改都先持久化再修改内存。那么就不存在的系统推出的时候对元数据持久化。

对业务逻辑进行进一步抽象，忽略业务细节，会发现，元数据的内存管理需要提供这些功能。收到一条record，解析record，然后根据不同的opcode在内存对元数据进行处理。

伪码如下：

```
while (stop) {
 get and parse a record -> record
 switch (record->opcode) {
  case 1: deal 1
   break
  case 2: deal 2
   break
  case 3: deal 3
   break
 }
 err handle
}
```

inode和dentry的内存管理结构

```
class InodeKey {
 uint64_t fsId;
 InodeId inodeId;
};

class InodeManager {
 public:
 void Insert(const Inode &inode);
 bool Get(const InodeKey &key, Inode *inode);
 void Delete(const InodeKey &key);
 void Update(const Inode &inode);
    int Count();

 private:
 // Inodefsid + inodeid key
 std::unordered_map<InodeKey key, Inode inode> inodeMap_;
 // inode
 ::curve::common::RWLock lock_;
```

```
};

class DentryKey {
 uint64_t fsId;
 InodeId parentId;
    std::string name;
};

class DentryParentKey {
 uint64_t fsId;
 InodeId parentId;
};

class DentryManager {
 public:
 void Insert(const Dentry &dentry);
 bool Get(const DentryKey &key, Dentry *dentry);
    bool List(const DentryParentKey &key, std::list<Dentry dentry>);
 void Delete(const DentryKey &key);
 void Update(const Dentry &dentry);
    DentryKey GetKey(InodeId parentId, std::string &name);
    int Count();

 private:
 // dentryfsid + parentid + namekey
 std::unordered_map<DentryKey key, Dentry dentry> dentryRecordMap_;
 // dentryfsid + parentidkeyparentiddentryls
    std::unordered_map<DentryParentKey key, std::list<Dentry dentry *> lists> dentryListMap_;
 // dentry
 ::curve::common::RWLock lock_;
```

```
};
```

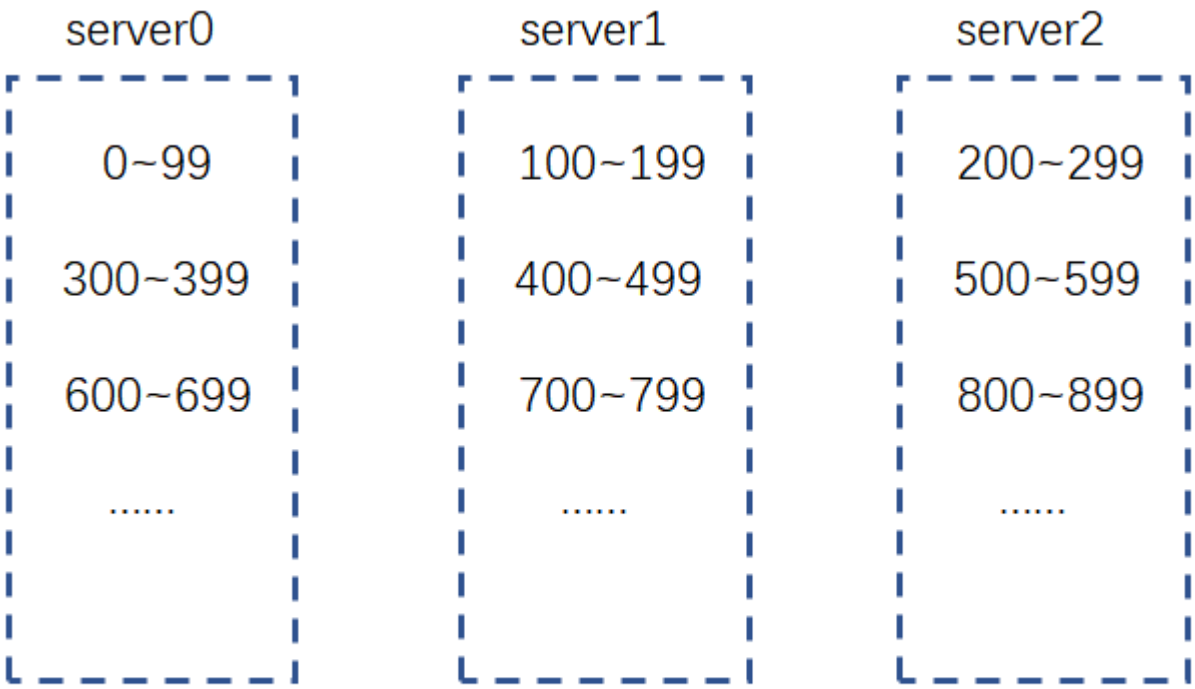
5 元数据分片

inode和dentry的组织是按照什么方式进行组织，还有一些因素需要考虑。

是mds节点上组成一个全局的结构体，还是分目录，按照一个目录进行组织。
这需要考虑的元数据管理的分片策略。当前curve文件系统目的是提供一个通用的文件系统，能够支持海量的文件，这就需要文件系统的元数据有扩展能力。元数据管理仅使用一台元数据管理服务器是不够的。使用多台元数据服务器需要对元数据进行合理的分片。

当前的一个可行方案是按照inodeid进行分片。分片算法如何设计，热点如何解决下半年细化，当前简单按照算法为 `serverid = (inodeid / inode_per_segment) mod metaserver_num`
进行分片。分片算法的具体实现不影响下面的讨论。

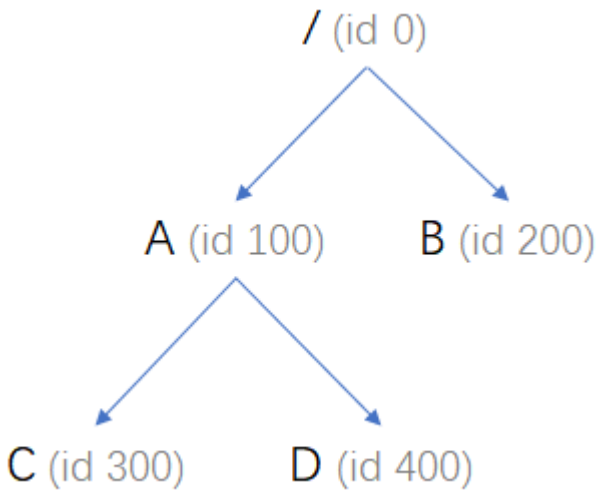
比如说分片规则按照每个分片管100个inodeid，如果有3个metaserver，那么分片信息就变成了这样。



Metaserver元数据分片策略，假设每个分片管理100个inode，3台metaserver。
根目录的Inodeid保留0，其他inodeid从1开始分配。

5.1 分片方式一：inode和dentry都按照parentid分片

现在假定文件系统有这样几个文件，根据上面的分片方式，parent为"/"和"/A/C"在server0上，parent为"/A"和"/A/D"在server1上，parent为"/B"在server2上。



元数据

server 0	server 1	server 2
inode信息		
0 → inode /	300 → inode C	
100 → inode A	400 → inode D	
200 → inode B		
dentry信息		
0 + A → 100	100 + C → 300	
0 + B → 200	100 + D → 400	

5.1.1 场景分析

查找：查找/A/C。

1、client给server0发送请求：parentid 0 + name "A"，查询"A"的inodeid为100

2、client给server0发送请求：inode 100，查询的"A"的inode信息。

3、client给server1发送请求：parentid 100 + name "C"，查询"C"的inode为300

4、client给server1发送请求：inode 300，查询"C"的inode信息。

创建：/A/C不在，创建/A/C

1、client给server0发送请求：parentid 0 + name "A"，查询"A"的inodeid为100

2、client给server0发送请求：inode 100，查询的"A"的inode信息。

3、client给server1发送请求：parentid 100 + name "C"，查询不到

4、client给server1发送请求：分配inodeid 300，创建文件C的inode。

生成记录300→ inode C

生成记录100 + C → 300

删除文件：删除/A/C

1、client给server0发送请求：parentid 0 + name "A"，查询"A"的inodeid为100

2、client给server0发送请求：inode 100，查询的"A"的inode信息。

3、client给server1发送请求：parentid 100 + name "C"，查询"C"的inode为300

删除记录 100 + C → 300

删除记录 300 → inode C

删除目录：删除/A

1、client给server0发送请求：parentid 0 + name "A"，查询"A"的inodeid为100

2、client给server0发送请求：inode 100，查询的"A"的inode信息。

3、client给server1发送请求：parentid 100，查询parentid 为100的dentry记录。查询到 100 + C → 300，目录非空，不能删除。

rename：rename /A/C到/B/E

1、client给server0发送请求：

parentid 0 + name "A"，查询"A"的inodeid为100

inode 100，查询的"A"的inode信息。

2、client给server1发送请求：

parentid 100 + name "C"，查询"C"的inode为300

inode 300，查询"C"的inode信息。

3、client给server0发送请求：

parentid 0 + name "B"，查询"B"的inodeid为200

inode 200，查询的"B"的inode信息。

server 0	server 1	server 2
inode信息		
0 → inode /	300 → inode C	
100 → inode A	400 → inode D	
200 → inode B		
dentry信息		
0 + A → 100	100 + C → 300	
0 + B → 200	100 + D → 400	

4、client给server2发送请求：

parentid 200 + name "E"，查询不到

生成记录 300 → inode E

生成记录 200 + E → 300

server 0	server 1	server 2
inode信息		
0 → inode /	300 → inode C	300 → inode E
100 → inode A	400 → inode D	
200 → inode B		
dentry信息		

0 + A → 100	100 + C → 300	200 + E → 300
0 + B → 200	100 + D → 400	

5、client给server1发送请求：

删除记录 100 + C → 300

删除记录 300 → inode C

server 0	server 1	server 2
inode信息		
0 → inode /	400 → inode D	300 → inode E
100 → inode A		
200 → inode B		
dentry信息		
0 + A → 100	100 + D → 400	200 + E → 300
0 + B → 200		

这里rename的时候，涉及到inode信息跨节点迁移。需要引入分布式锁，是个难点。

symbolic link：

这个类型的文件和普通文件一样创建删除，区别在于，在inode信息中记录需要链接到的地址。

hardlink：生成一个hardlink /B/E，指向文件/A/C

1、client给server0发送请求：

parentid 0 + name "A"，查询"A"的inodeid为100

inode 100，查询的"A"的inode信息。

2、client给server1发送请求：

parentid 100 + name "C"，查询"C"的inode为300

inode 300，查询"C"的inode信息。

3、client给server0发送请求：

parentid 0 + name "B"，查询"B"的inodeid为200

inode 200，查询的"B"的inode信息。

4、client给server2发送请求：

parentid 200 + name "E"，查询不到

生成记录？inode 300，按照原文件/A/B，应该在A的inodeid映射的机器上；按照硬链接/B/E，应该在B的inodeid映射的机器上。

生成记录 200 + E → 300

5、client给server1发送请求：

修改记录 "C"的inode link++

这里涉及到增加dentry和增加link，这两个操作不在一个节点上，也需要使用分布式锁进行控制，做成事务。

list：遍历/A目录

1、client给server0发送请求：

parentid 0 + name "A"，查询"A"的inodeid为100

inode 100，查询的"A"的inode信息。

2、client给server1发送请求：

parentid 100，查询parent id为100的dentry所有的记录，查到dentry信息 [{"C", 300}, {"D", 400}]

inode 300，查询"C"的inode信息。

inode 400，查询"D"的inode信息。

5.1.2 好处

这种方案的好处在于，inode和dentry大概率落到一个分片上管理。在查询inode的过程中，第一步通过parentid和name查询inodeid，第二步通过inodeid查询inode结构体在同一个分片上处理。查询时，client只需要向metaserver发送一次请求，就可以完成上面两步的查询任务。

5.1.2 问题

在一种特殊的情况下，可能出现无法按照的parentid找到对应inode的情况。

还是是上面的文件系统，生成一个hard link，/B/E指向/A/C

元数据

server 0	server 1	server 2
inode信息		
0 → inode /	300 → inode C	
100 → inode A	400 → inode D	
200 → inode B		
dentry信息		
0 + A → 100	100 + C → 300	200 + E → 300
0 + B → 200	100 + D → 400	

删除/A/C之后，元数据变成了

server 0	server 1	server 2
inode信息		
0 → inode /	300 → inode C	
100 → inode A	400 → inode D	
200 → inode B		
dentry信息		
0 + A → 100	100 + D → 400	200 + E → 300
0 + B → 200		

这个时候，可以通过parent 200 + name "E"在server2上查到E的inodeid为300，但是在server2上，找不到对应的id为300的Inode的结构体。

这个问题可以有两个解决办法：

一、遍历所有的metaserver，去所有的metaserver上查询id为300的inode信息。

二、通过一个额外的缓存，缓存inode id和partition的映射关系。这个缓存可以在挂载文件系统的时候缓存在client端。不缓存具体的Inode的结构体，仅仅缓存(inodeid, partitionid)的映射，如果inodeid为uint64类型，partitionid为uint64_t类型，那么一条记录需要16字节。一个文件系统按照10亿的元数据统计，10亿 * 16字节 = 1.5GB，全部缓存到内存需要1.5GB的内存。除了缓存需要占用的内存资源之外，如果涉及到多挂载的场景，还需要处理inode缓存失效的问题。

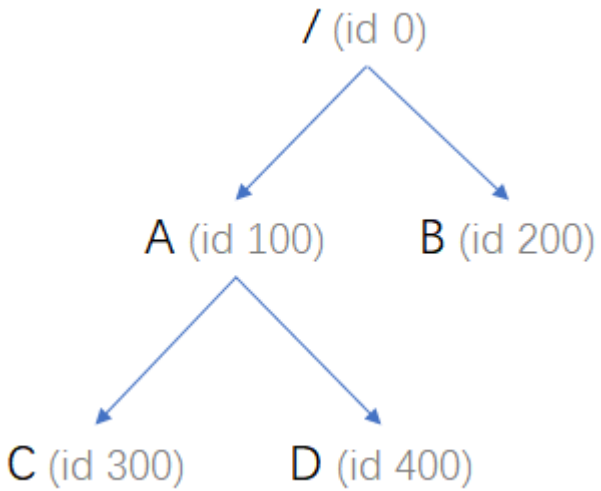
5.2 分片方式二：Inode按照inodeid进行分片，Dentry按照parentid进行分片

这种分片方式的，inode和dentry的分布没有任何关系，查找inode和查找dentry的大概率需要不同的分片进行处理。这样第一步通过parentid和name去查询inodeid，第二步通过inodeid去查询inode结构体，这两步就必须通过两次请求。相对于分片方式一，这种方式，client向metaserver进行查询的时候，rpc请求的个数近似翻倍，多一倍的查询，对性能上的影响能否接受。

inode的查询在client端其实有缓存。只有第一次查询的时候，client才需要去metaserver进行查询，以后对文件的操作，只需要使用本地缓存的inode信息。这样一来，查询inode多一倍的rpc请求，对性能的影响可能没有想象中的大。一旦client知道了文件或者目录的inodeid，后续对inode的修改，都不需要去先查询dentry信息，可能直接对inode进行修改。

还有将来如果支持多挂载或者一写多读或者多写多读的场景，那么面临着client的缓存失效的问题，这个时候需要去metaserver重新查询inode的信息，这个查询也不需要重新查询dentry信息。因为一个文件或者目录，一旦创建出来之后，inodeid是不会发生变化的，哪怕dentry发生了变化，文件rename到其他地方，也不会影响到inodeid，只需要client的缓存去Inodeid所在的分片直接查询即可。

现在假定文件系统有这样几个文件，根据上面的分片方式，dentry分片parent为"/"和"/A/C"在server0上，parent为"/A"和"/A/D"在server1上，parent为"/B"在server2上；inode分片，inode为"/"和"/A/C"在server0上，inode为"/A"和"/A/D"在server1上，inode为"/B"在server2上。



元数据在不同的server上的分布如下：

server 0	server 1	server 2
inode信息		
0 → inode /	100 → inode A	200 → inode B
300 → inode C	400 → inode D	
dentry信息		
0 + A → 100	100 + C → 300	

0 + B → 200	100 + D → 400	

上面有疑问的rename和hardlink这里分析一下。

rename：rename /A/C到/B/E

1、client给server0发送请求：

parentid 0 + name "A"，查询"A"的inodeid为100

2、client给server1发送请求：

inode 100，查询的"A"的inode信息。

parentid 100 + name "C"，查询"C"的inode为300

3、client给server0发送请求：

inode 300，查询"C"的inode信息。

parentid 0 + name "B"，查询"B"的inodeid为200

3、client给server2发送请求：

inode 200，查询的"B"的inode信息。

parentid 200 + name "E"，查询不到

4、client给server2发送请求：

生成记录 200 + E → 300

server 0	server 1	server 2
inode信息		
0 → inode /	100 → inode A	200 → inode B
300 → inode C/E	400 → inode D	
dentry信息		
0 + A → 100	100 + C → 300	200 + E → 300
0 + B → 200	100 + D → 400	

5、client给server1发送请求：

删除记录 100 + C → 300

server 0	server 1	server 2
inode信息		
0 → inode /	100 → inode A	200 → inode B
300 → inode C/E	400 → inode D	
dentry信息		
0 + A → 100	100 + D → 400	200 + E → 300
0 + B → 200		

这里rename的时候，涉及到inode不需要变动，只是dentry改变。存在着一个中间状态，新的dentry生成，旧的dentry还未删除，这对文件的inode本身不会改变。这里如何保证事务性，上半年的demo先不考虑，留到下半年解决。

hardlink：生成一个hardlink /B/E，指向文件/A/C

1、client给server0发送请求：

parentid 0 + name "A"，查询"A"的inodeid为100

2、client给server1发送请求：

inode 100，查询的"A"的inode信息。

parentid 100 + name "C"，查询"C"的inode为300

3、client给server0发送请求：

inode 300，查询"C"的inode信息。

parentid 0 + name "B"，查询"B"的inodeid为200

3、client给server2发送请求：

inode 200，查询的"B"的inode信息。

parentid 200 + name "E"，查询不到

4、client给server2发送请求：

生成记录 200 + E → 300

server 0	server 1	server 2

inode信息		
0 → inode /	100 → inode A	200 → inode B
300 → inode C/E	400 → inode D	
dentry信息		
0 + A → 100	100 + C → 300	200 + E → 300
0 + B → 200	100 + D → 400	

5、client给server0发送请求：

修改记录 "C"/"E"的inode link++

这里只需要增加一台dentry，然后inode中link++。同样这里的操作也分为两步，事务性也需要处理，留到下半年考虑。

6、curve文件系统的多文件系统的设计

curve文件系统设计上支持多文件系统。文件系统的super block元数据设计。

多文件系统相对于单文件系统，多了一个fsid，在上面的inode和dentry中，需要添加相应的fsId字段。并且在查询inode和dentry的过程中，也需要带上fsId字段进行查询。

```
Enum FsState {
 CLEAN,    // CLEANMOUNTED
 MOUNTED,  //
 //
};

//
struct VolumeInfo {
 //
 uint64_t volumeSize;
 // 4KB
 uint64_t minAllocSize;
 //
 std::string volumeName;
 // user
```

```
 std::string user;
 // passworduserroot
 std::string password;
};

// proto
enum FSErrorNum {
 FS_OK = 0,
 // TODO
 FS_UNKNOWN = 1000,
};
// S3super block
class VolumeFs {
 public:
 // open
 VolumeFs(VolumeInfo volumeInfo, std::string name);
 uint64_t GetId();
 std::string GetName();
 uint64_t GetCapacity();
 VolumeInfo GetVolume();
 FsState GetState();
 Inode GetRootInode();
 //
 int32_t UpdateVolume(VolumeInfo volumeInfo);
 //
 int32_t mount(std::string mountPoint);
 //
 int32_t umount(std::string mountPoint);
 // mountNum0
 int32_t Delete();
 uint64_t GetUsedSize();

 private:
 // idmkfs
 uint64_t id;
 //
 uint32_t mountNum;
 // inode
```

```
 uint64_t inodeNum;
 //
 FsState state;
 //
 std::string name;
 // inode
 InodeId rootId;
 //
    std::list<std::string> mountPoints;
 //
    VolumeInfo volumeInfo;
 //
 uint64_t capacity;
 //
 uint64_t used;
 //
 uint32_t minAllocSize;
};

// S3super block
class S3Fs {
 // TODO
};
```

```
// fsfs name
std::map<std::string name, VolumeFs> fsMap;
```