



# 答辩文档

## 一、项目介绍

### 1.1 核心信息

本项目基于Go语言以及ProtoBuf&gRPC通信协议，利用git进行了项目管理，参考了GFS的框架完成了一个通过api调用进行交互的简易分布式存储系统，实现了部分高可用，并利用LevelDB持久化存储的元数据内容。

### 1.2 Github地址

| [Kirov7/FayDFS \(github.com\)](#)

## 二、项目分工

队名：能不能别挂提前批队 队号：3306

考虑到该项目为分布式存储系统的开发，因此本团队基于三大模块client，datanode以及namenode开发作为基本准则，再完善基本功能后又对进阶功能进行了任务的再次分配。

团队成员	主要贡献
------	------

孙天昊	namenode的主要开发成员，负责开发了namenode模块的大多数功能，实现了元数据的存储帮助完成了测试工作，负责了初期的框架调研的组织工作，完成了初期的接口设计工作，期间协助各个模块进行接口的统一管理并完成了LevelDB的引入工作，实现了元数据持久化功能。完成了datanode高可用关键部分宕机副本转移。完成并主要负责后期系统测试与故障修复。
叶竞成	clinet的主要开发成员，负责和翟嘉诚开发client模块，实现了用户无感知下的API调用流程，完成了测试工作，并协助调研raft的部分内容。负责了日常会议和进度的组织管理工作，后续负责答辩文档的撰写工作。负责团队的项目答辩。
翟嘉诚	clinet的主要开发成员，负责和叶竞成开发client模块，实现了用户无感知下的API调用流程。系统raft模块的主要调研和开发人员，试完成raft的相关开发，限于时间暂未实现。在client实现datanode高可用部分宕机无感读取副本。参与后期测试与故障修复。
田家兴	datanode的主要开发人员，负责开发datanode模块，实现了存在多副本备份的datanode并完成了datanode的模块测试以及联测工作，后续辅助了raft的部分开发。参与后期测试与故障修复。负责拍摄项目演示视频。
张林飞	namenode的主要开发成员，负责开发了namenode的部分对外通信接口，协助完成了部分测试工作。协助调研了部分levelDB的相关内容。并协助调研raft的部分内容。参与后期测试与故障修复
何君健	负责调研了部分namenode以及levelDB的相关背景和内容。
唐振	负责调研了部分datanode的相关背景和内容，参与了前期框架的制定讨论。
叶昕骅	负责调研了部分namenode以及levelDB的相关背景和内容。

## 三、项目实施

### 3.1 技术选型

#### 3.1.1 面临的技术问题

本团队经过对于题目的解读和行业规范的部分调研，经过讨论以及开发过程中的亲身经历总结出以下问题：

1. 面对开发需求，如何进行分布式存储，应当选取怎样的框架，是本团队需要考虑的第一步。
2. 通信是分布式框架的基础，应该选取怎样的通信协议才可以使各模块间高效通信，而团队又该怎么统一服务接口间所需要传输的消息。
3. 虽然元数据的本身体积很小，但是面对大量业务时，如何进行元数据存储，可以避免KV形式的数据存储在内存上也是重要的课题之一。
4. 真实的系统将会拥有大量的datanode节点，此时namenode如何有效的分配存储空间，如何实现多副本存储尤为重要。

5. namenode虽然只有一个，但是却是整个系统的核心，如何保证namenode的高可用性是整个存储系统可靠的基石。
6. 重命名的接口虽然简单，但是在仅修改元数据的情况下，如何避免未被修改的datanode中的副本在后续过程中重名，也是团队开发中遇到的一个小问题。

### 3.1.2 选用的开发框架

本节将针对3.3.1节提出的问题，予以本团队在该项目中的框架选用以及简单的解决方案。而具体的功能实现流程，会在本章的3.3节开发文档中详细介绍。

1. 考虑到分布式存储框架种类众多，由于团队基础薄弱，因此先从GFS的论文入手，之后参考了hdfs的相关源码，最终得出了采用传统的client与用户直接进行交互、namenode处理请求及存储元数据、datanode负责数据存储的三个模块形式。
2. 本团队一开始准备选用HTTP协议进行通信，但后来由于RPC的报文体积更小，并且可以高效的进行二进制传输，并且自带了负载均衡策略，因此改用grpc作为本项目开发的通信协议。
3. 面对KV存储的问题，本团队选用了目前市场上广泛使用的存储引擎LevelDB，以将原本保存在内存上的元数据持久化，保存成日志形式，写入磁盘中。
4. 本团队参考了GFS的设计将datanode的存储单元以64MB作为分割标准，分割成了若干个block，namenode将会把每个部分随机分配三个副本到不同的datanode中，以保证在少量datanode损坏或离线时，不会造成数据丢失。
5. 经过对于一致性共识算法的调研，本项目将采用raft框架来完成元数据的一致性保护，以保证在最为核心的namenode损坏时仍然保证整个系统的可用性。
6. 对于这个存储小问题，我们引入了uuid通过唯一标识符固定了client但是在实现时发现或许在每个副本结尾加上一个时间戳也可以更为简单的达到相同的目的，便放弃了使用uuid。

## 3.2 架构设计

### 3.2.1 基础架构

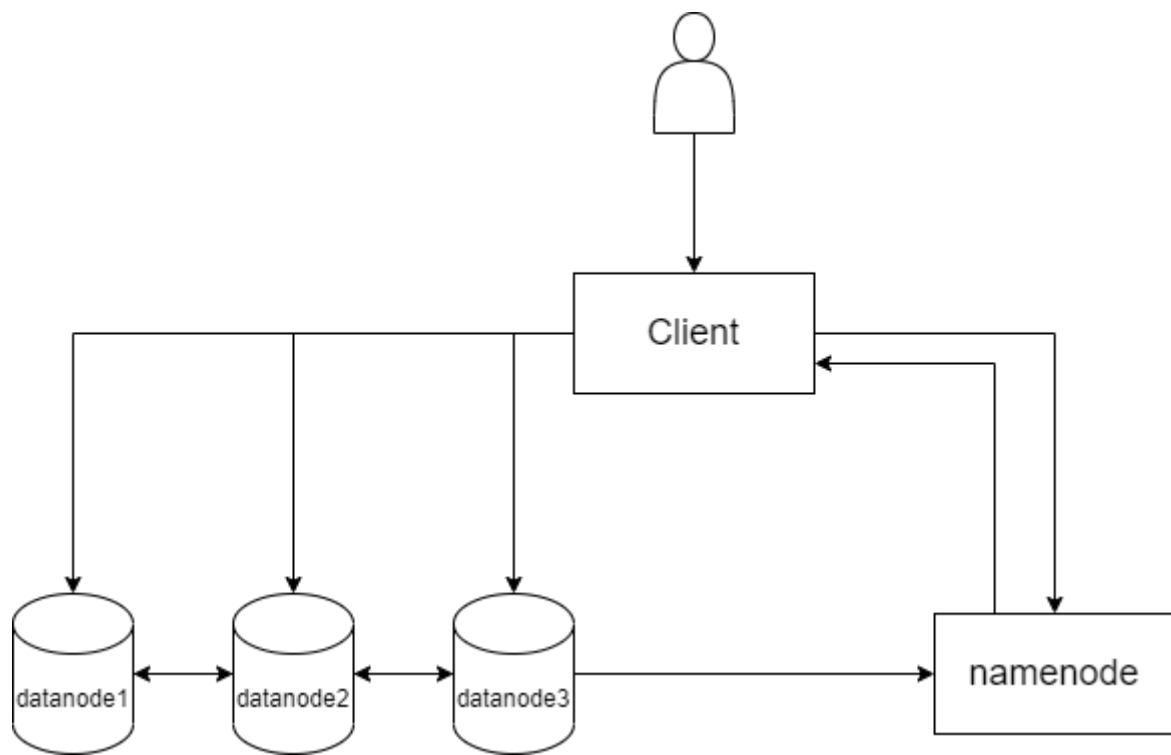


图1 GFS基础架构

本团队所设计的分布式存储系统参考了传统设计，基础结构共分为三大块，分别是Client、namenode、和datanode。

- **Client**主要负责和用户直接进行交互，是用户无感知的使用项目的相关通用接口来获取，管理，和存储数据。
- **namenode**主要负责元数据的管理，是整个系统管理的核心，任何数据的读取存储都需要通过namenode来处理，为了降低整体项目的复杂度，整个项目仅有一个处于工作状态的namenode，详细的流程设计将会在开发文档中阐述。
- **datanode**主要负责数据的存储，本项目的datanode设计为类似于GFS的chunk设计，在代码中体现为block。每一个datanode拥有若干个block，而每个block将存储文件的部分内容，实现多副本存储以及，将文件分布式存储的效果。

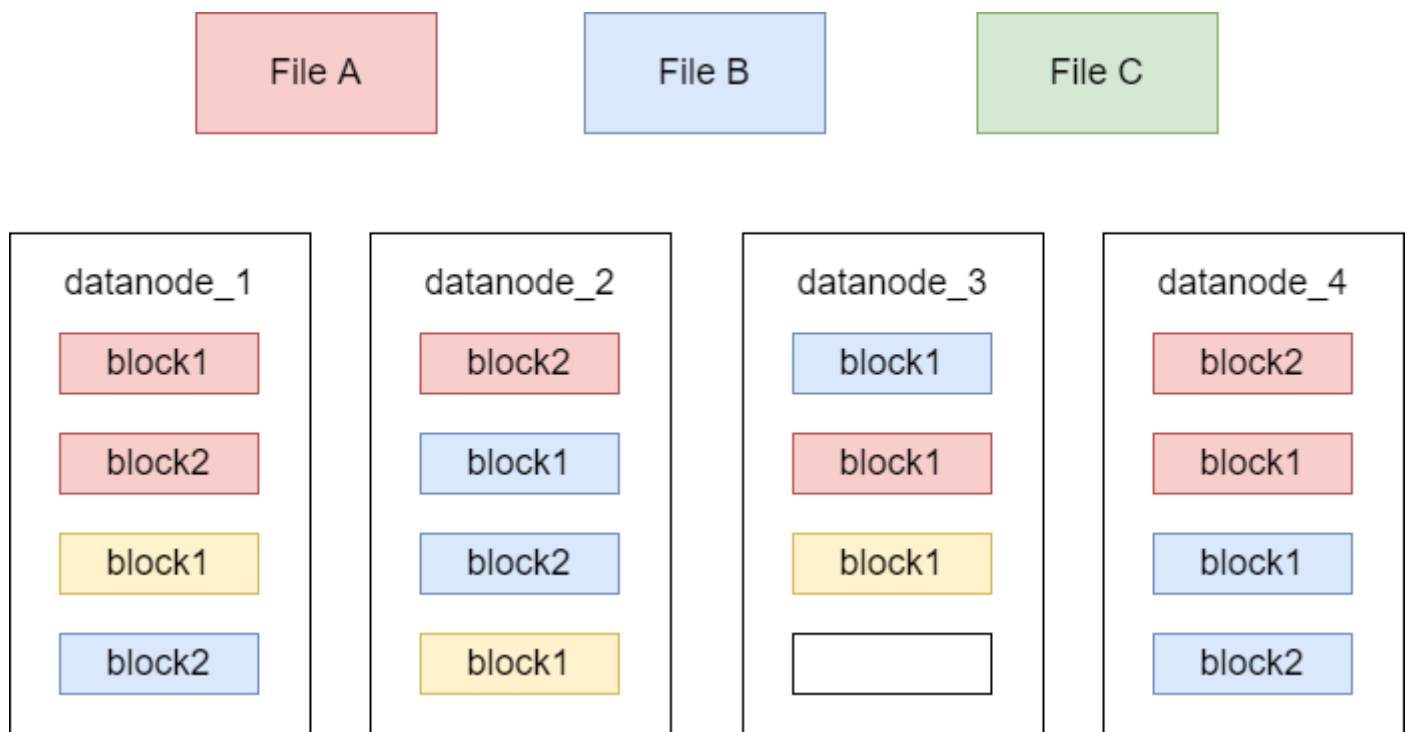


图2 datanode存储结构

如上图所示，每个文件将会有三个副本存储在不同的datanode之中，以保证在少量namenode因特殊情况离线的环境下仍然可以运行，并且本项目以64MB这类大尺寸Chunk作为容器可以减少通讯需求，并且对于大文件也可以降低读写和通讯次数以减轻namenode的负担。

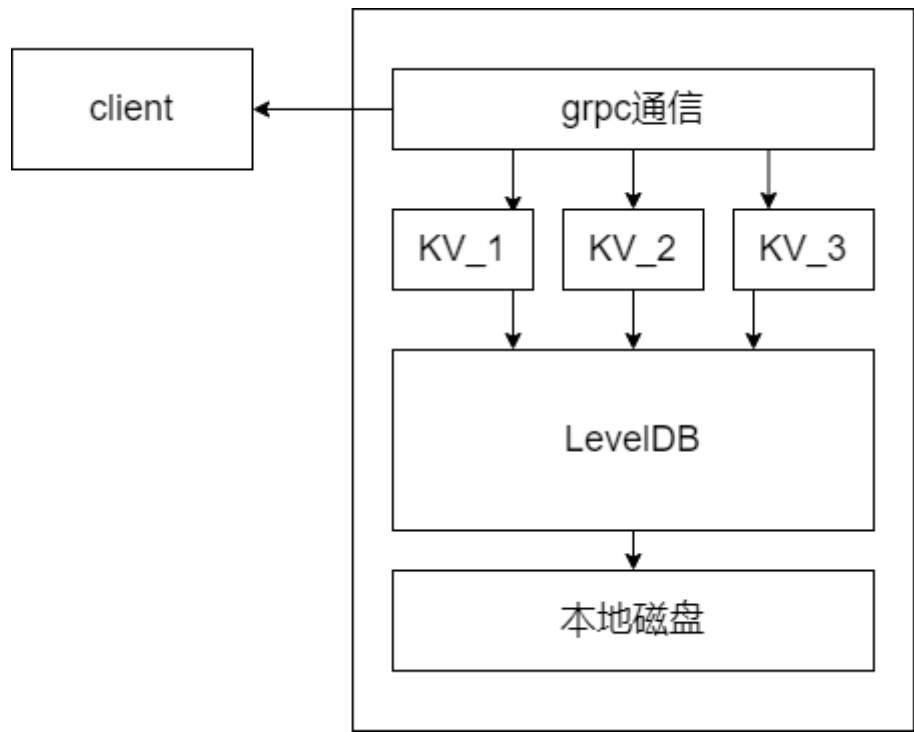


图3 namenode存储结构

如上图所示，namenode将通过grpc和client通信，通信后将获取到的信息转化为元数据，再通过LevelDB存储到本地磁盘中，而具体的操作流程，将再开发文档中阐述。

### 3.2.2 通信接口设计

由3.2.1节的内容可以知道，其实clinent需要同时和namenode和datanode进行交互，而datanode也需要找到自己所在系统中唯一的namenode因此，三个模块间都需要有相应的通信协议以支持接口操作，具体的功能函数将在3.4节中介绍。

通信双方	主要功能
client - namenode	1. 获取元数据 2. 获得待写入文件block位置 3.源数据修改
client - datanode	根据namenode所分配的位置信息按block写入文件
datanode - namenode	1. 每3s发送心跳 2.每1h汇报block状态 3.向namenode进行注册

### 3.2.3 通用接口设计

通用接口即用户感知的可进行操作的接口，即常见数据库的功能。

函数名称	函数功能
Put(localFilePath, remoteFilePath string) Result	将本地文件传入存储系统中
Get(remoteFilePath, localFilePath string) Result	将存储系统中的文件拉取到本地
Delete(remoteFilePath string) Result	删除存储系统中的文件
Stat(remoteFilePath string) Result	获取存储系统中目标文件的元数据
Rename(renameSrcPath, renameDestPath string) Result	将系统中文件路径重命名
Mkdir(remoteFilePath string) Result	再存储系统中创建路径
List(remoteDirPath string) Result	获取存储系统中的目标文件夹下的元数据

## 3.3 开发文档

- 开发环境

只需安装1.18+golang环境即可，表格中的资源包均由go mod管理。

开发环境	版本
Golang	1.18+
资源包	版本
<a href="https://google.golang.org/protobuf">google.golang.org/protobuf</a>	v1.28.0

<a href="https://google.golang.org/grpc">google.golang.org/grpc</a>	v1.48.0
<a href="https://github.com/satori/go.uuid">github.com/satori/go.uuid</a>	v1.2.0
<a href="https://github.com/syndtr/goleveldb">github.com/syndtr/goleveldb</a>	v1.0.0

- 代码获取

```
1 git clone https://github.com/Kirov7/FayDFS.git
```

- 使用步骤

1. 配置资源包

在FayDFS路径下执行：

```
1 go mod tidy
```

2. 启动namenode以及datanode服务

```
1 go run ./namenode/main.go
2 go run ./datanode/main.go -dir data0 -dn_addr 127.0.0.1:8010 -ppl_addr 127.0.0.1:5
3 go run ./datanode/main.go -dir data1 -dn_addr 127.0.0.1:8011 -ppl_addr 127.0.0.1:5
4 go run ./datanode/main.go -dir data2 -dn_addr 127.0.0.1:8012 -ppl_addr 127.0.0.1:5
5 go run ./datanode/main.go -dir data3 -dn_addr 127.0.0.1:8013 -ppl_addr 127.0.0.1:5
```

3. 启动或修改test.go以进行接口调用或测试（需要修改测试文件中，待测试文件路径）

```
1 go run ./test.go
```

## 3.4 功能实现流程及模块设计详解

- 由于部分功能的相似度较高，这里将选取三个模块间的基本运作流程，部分接口的详细实现流程进行阐述。

- put流程

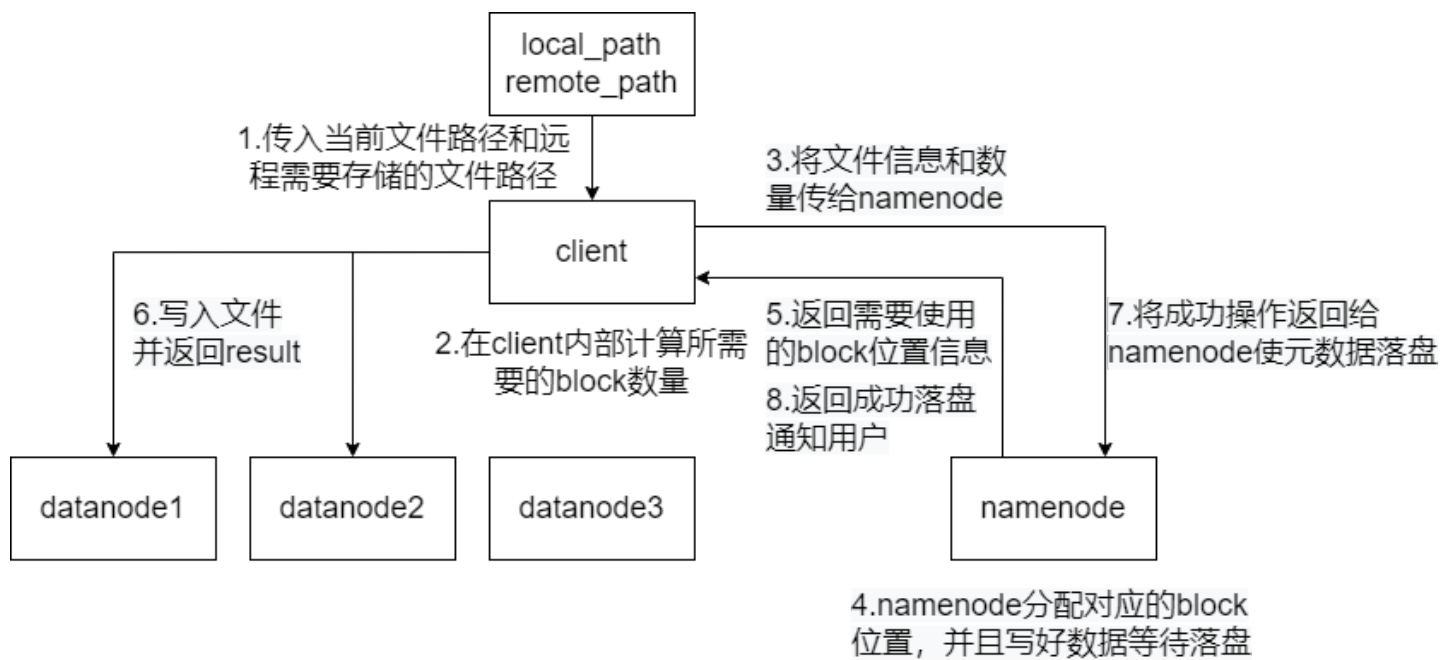


图4 put操作流程

#### get流程

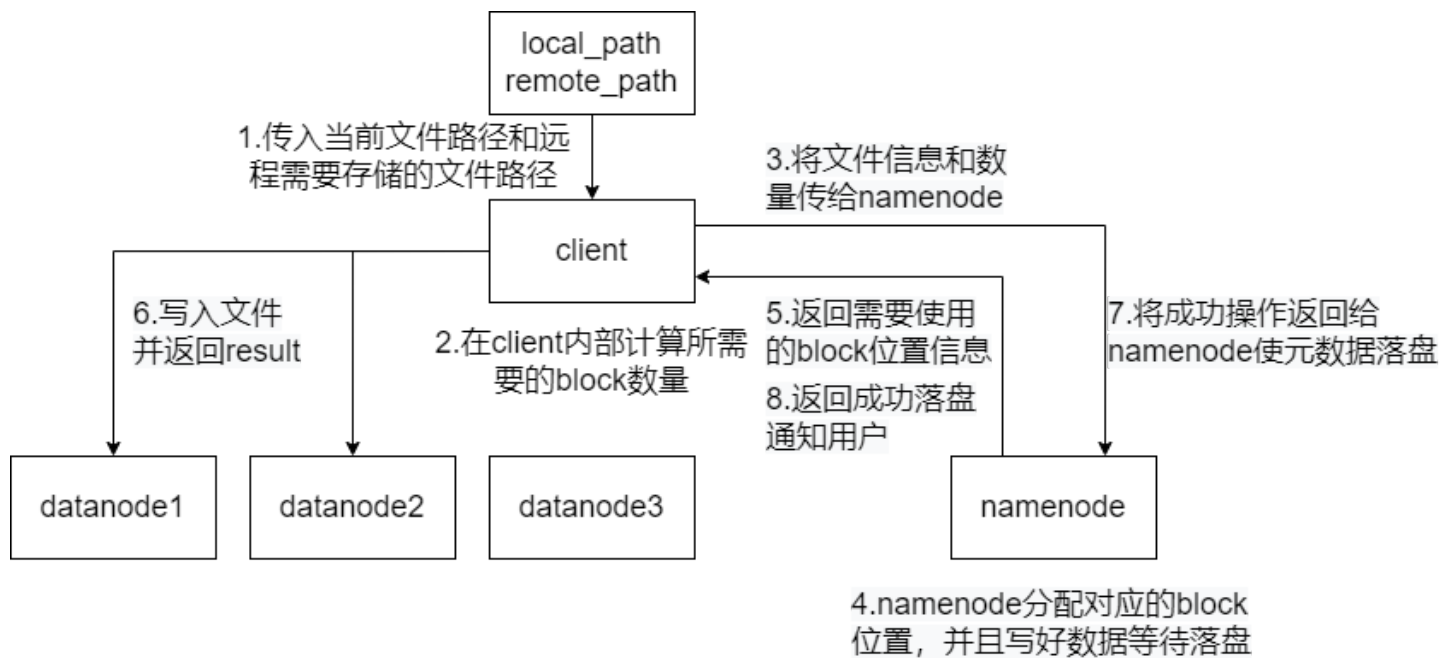


图5 get操作流程

#### delete流程



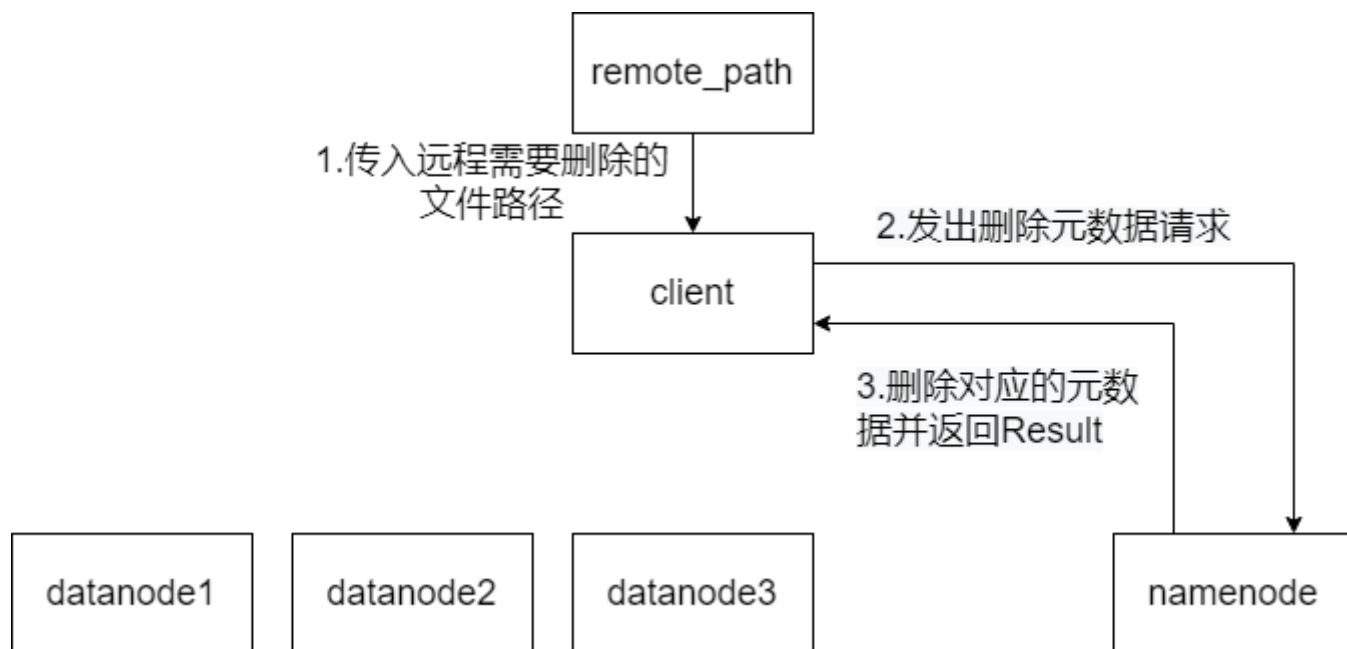


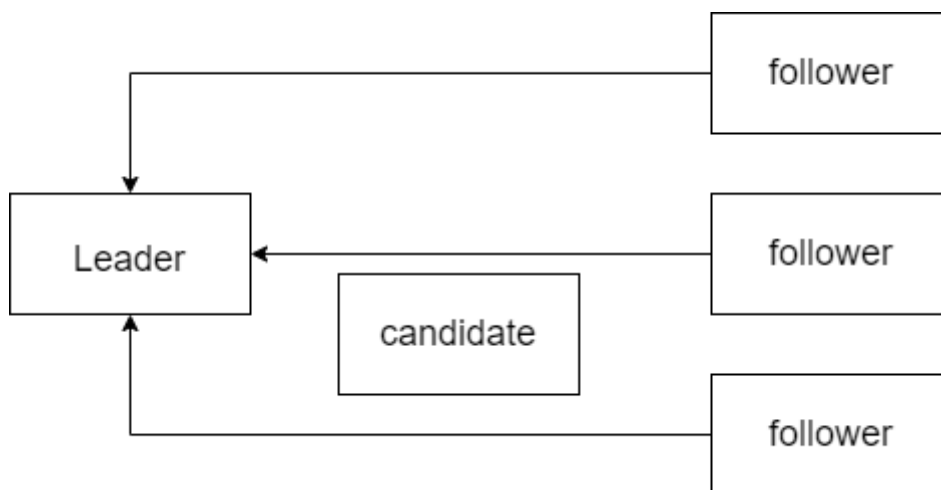
图6 get操作流程

- 值得一提的是，该操作并不会涉及datanode的相关操作，仅仅是删除namenode中的元数据，类似于hdfs，所以其实文件仍然保存在datanode中，而如何定期讲无对应元数据的垃圾回收也是本项目未来的优化目标之一。

#### ◦ rename、mkdir、List、Stat流程

- 这两个操作较为简单，因此不再画图描述，仅需要client发出对namenode元数据的修改请求即可，需要注意的是，由于文件命名需要符合一定的规范，所以不可能跨路径或使用特殊符号命名，在源码中我们对命名和修改规范也加以了限制，具体可以参开源码部分。List与Stat流程同理。

#### ◦ raft保证namenode高可用流程（由于时间问题，此模块并未完成实现）



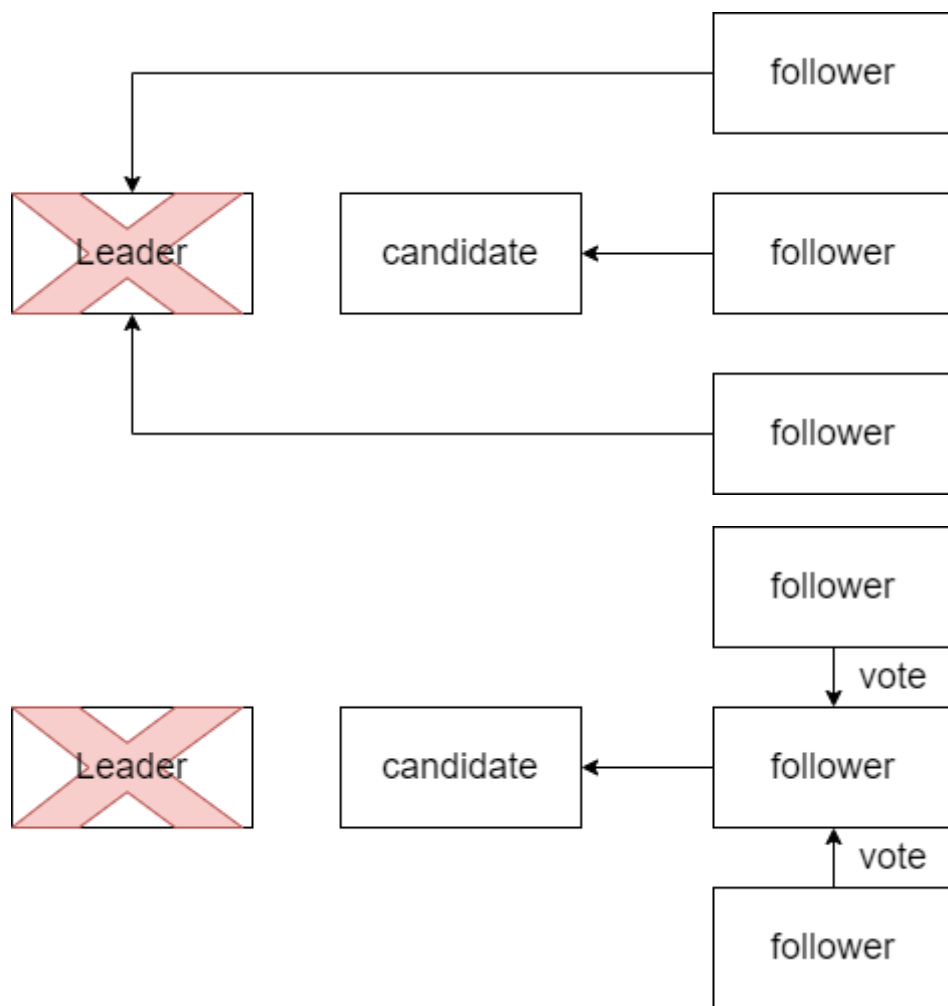


图7 raft流程

- 如上图所示，raft的整体流程可以总结为以下几步：
  - a. 正常只有一个leader在工作，而其他的follorow将会持续和leader交互并复制leader
  - b. 当leader因意外卡死或离线后，率先发现的follower将会申请成为candidate
  - c. 为了避免只是短暂卡死，或者是follower本身错误，candidate需要让大多数follower进行投票，即其他follower也认为原先的leader以及离线
  - d. 此时获得多数票的candidate将会成为新的leader保证系统的高可用性。
- **datanode心跳汇报流程**

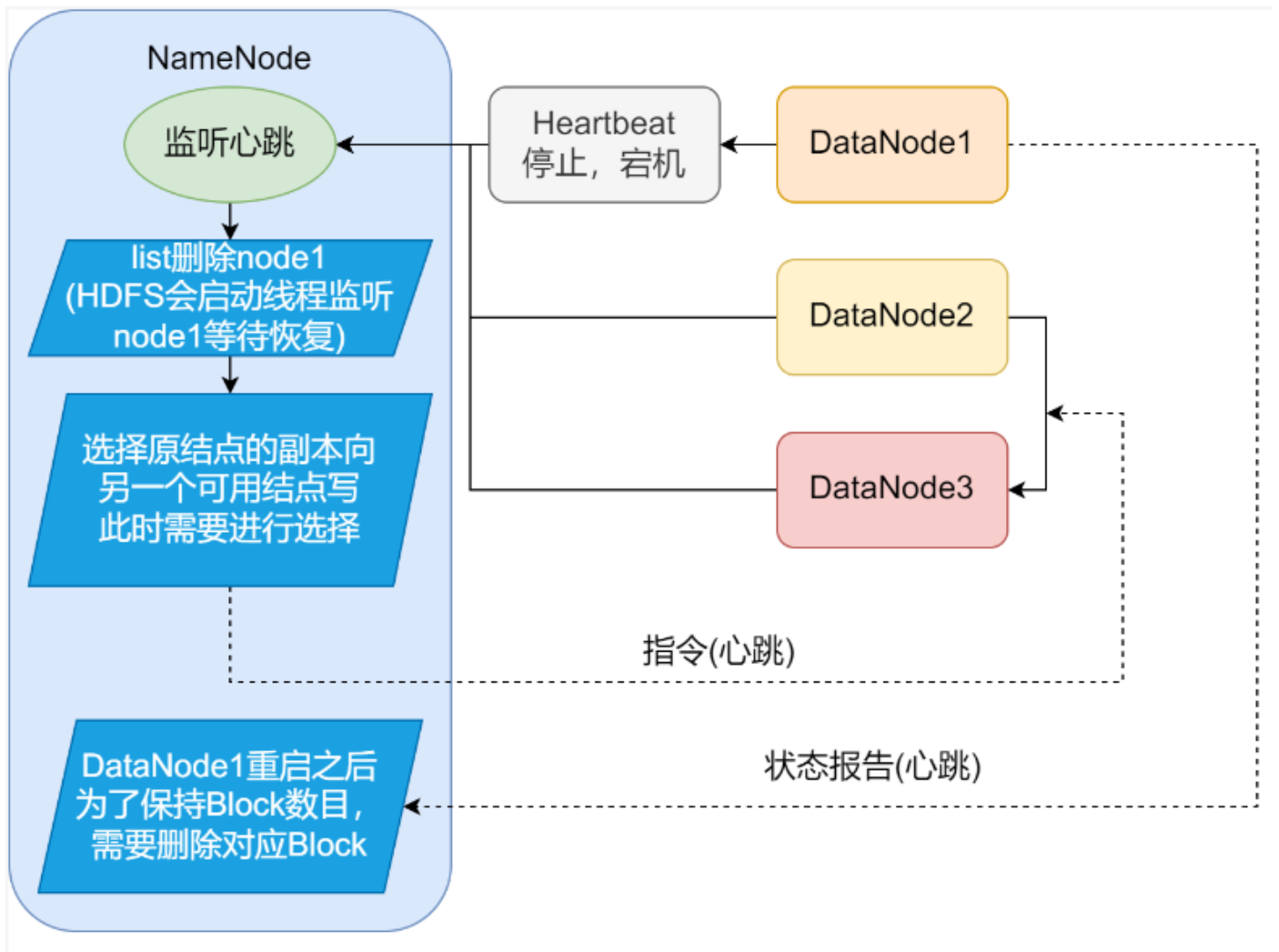


图8 datanode心跳汇报流程

- LevelDB流程

namenode内置了  
三个levelDB服务

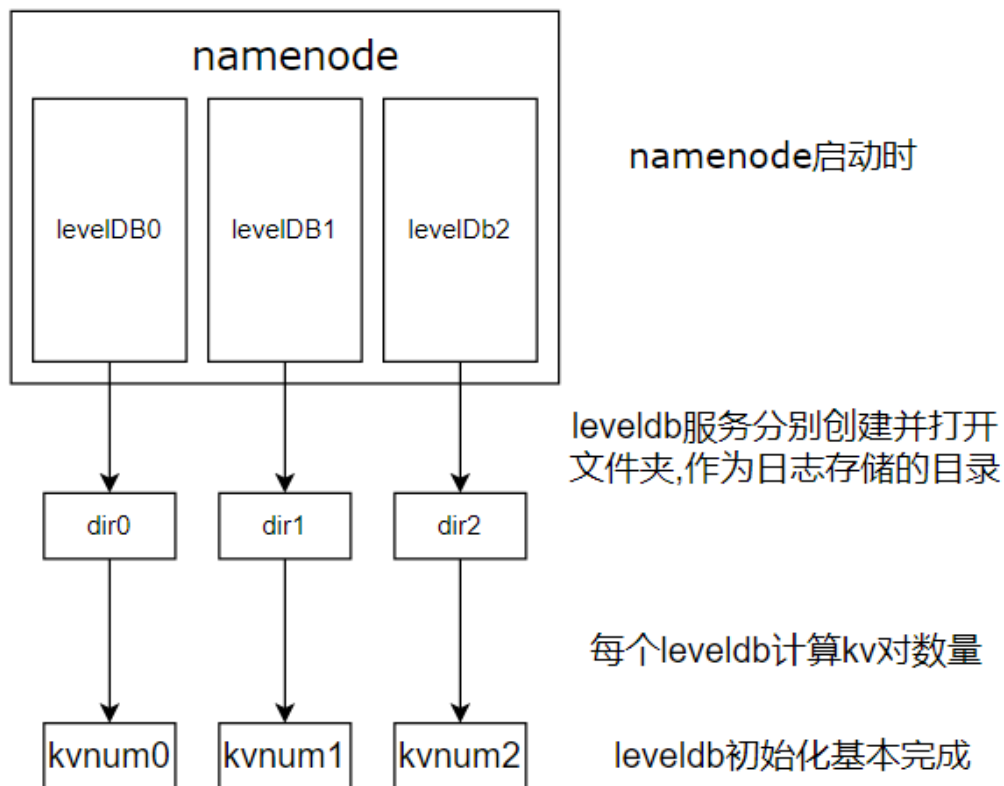


图9 levelDB初始化

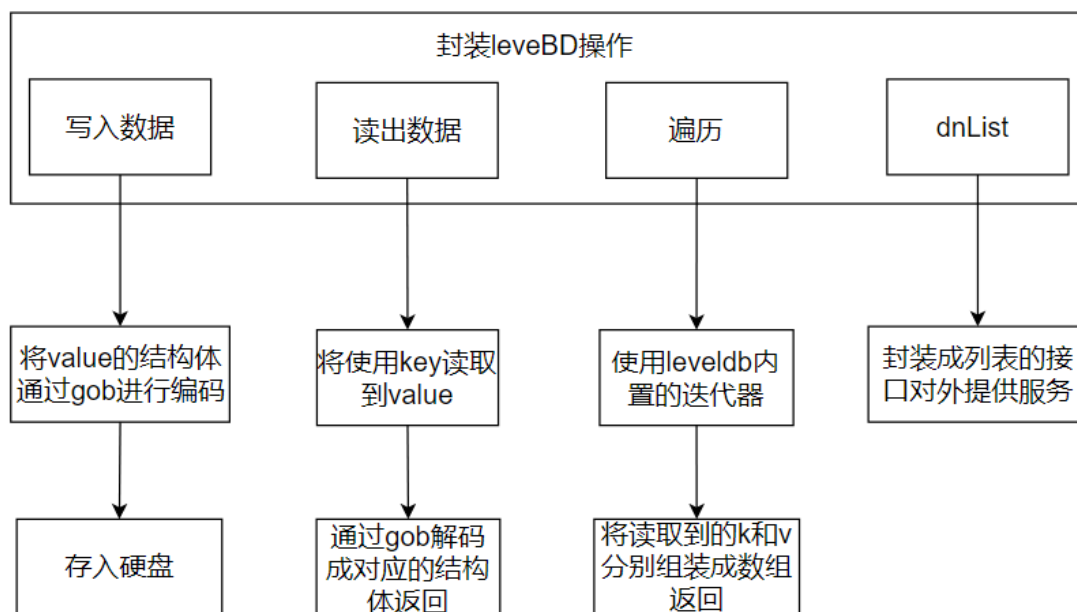


图10 封装levelDB操作

- 多副本备份流程

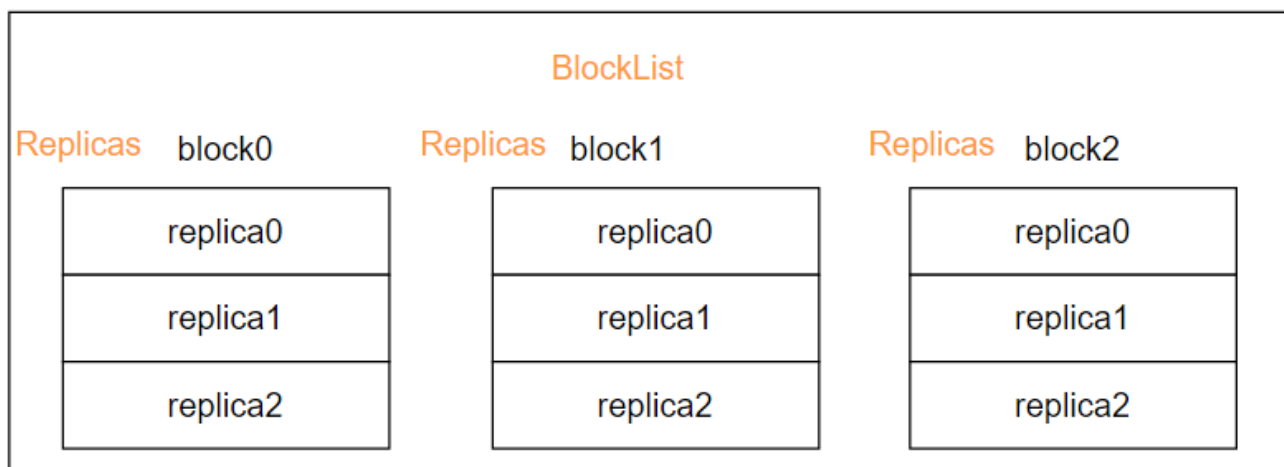


图11 Block结构

- 多副本写入是通过client一次性写完所有副本，我们封装的blocklist是一个二维数组，第一维是数据分片后应该存入的不同的block,第二维是每个block的所有相同数据不同地址的副本。在client写入是会将所有的副本写完，后续如果有存储节点损坏会向没有存储过改副本数据的存储节点自动进行数据迁移。

### 3.4 项目代码结构介绍

- 整体框架

```

1  .
2  |—— client
3  |   |—— client.go - client端功能的实现
4  |   |—— service
5  |       |—— clientAPI.go - client端对外接口的定义
6  |—— config
7  |   |—— config.go - 配置文件数据获取
8  |   |—— config.json - 配置文件
9  |—— datanode
10 |   |—— main.go - datanode启动入口
11 |   |—— message
12 |       |—— message.go - datanode通信信息体
13 |       |—— service
14 |       |—— dataNode.go - datanode功能具体实现
15 |—— namenode
16 |   |—— main.go - namenode启动入口
17 |   |—— service
18 |       |—— leaseManger.go - 租约管理
19 |       |—— leveldb.go - leveldb功能的封装
20 |       |—— nameNode.go - namenode功能具体实现

```

```

21 |—— proto
22 |   |—— faydfs.pb.go - grpc server API
23 |   |—— faydfs.proto - grpc服务定义
24 |   |—— faydfs_grpc.pb.go - grpc client API
25 |—— public
26 |   |—— const.go - 常量定义
27 |   |—— errors.go - 错误定义
28 |   |—— util.go - 工具函数
29 |—— go.mod
30 |—— go.sum
31 |—— README.md
32 |—— test.go

```

- 参数配置（**config**配置参数说明）

```

1  type Config struct {
2      BlockSize          int64  - 存储块大小
3      Replica            int    - 副本数
4      IoSize             int    - datanode文件读写缓冲区大小
5      NameNodePort       string - namenode端口
6      EditLog            string - 操作日志
7      DataDir           string - 文件存储路径
8      NameNodeHost       string - namenodeIP地址
9      HeartbeatInterval int    - datanode心跳发送周期
10     HeartbeatTimeout   int    - heartMonitor定时检测周期
11     BlockReportInterval int    - datanode块信息汇报周期
12     LeaseSoftLimit     int    - 租约软限制
13     LeaseHardLimit     int    - 租约硬限制
14 }

```

## 四、测试结果

### 4.1 功能测试

本项目根据大项目提出的要求完成了以下得分点：


已完成的评分项	评分权重
实现用户接入客户端	10
实现文件的读写删接口	10
实现文件/目录元数据操作接口	10

实现多副本策略存储	5
实现数据分区，文件按block拆分	5
支持存储节点单点故障不影响已有数据读取	3
支持元数据节点感知故障并且副本补全	2
完成测试性能报告	5
使用git进行团队代码管理	1.5

以下是功能测试环节：


API	输入	输出	测试
Put	<ul style="list-style-type: none"><li>本地文件路径 localFilePath</li><li>远程文件路径 remoteFilePath</li></ul>	<ul style="list-style-type: none"><li>成功，输出 successes put</li><li>失败，报错</li></ul>	Put("D://
Get	<ul style="list-style-type: none"><li>远程文件路径 remoteFilePath</li><li>本地文件路径 localFilePath</li></ul>	<ul style="list-style-type: none"><li>成功，输出 write to local success</li><li>失败，报错 write to local fail</li></ul>	Get("/tes
Delete	<ul style="list-style-type: none"><li>待删除的远程文件路径 remoteFilePath</li></ul>	<ul style="list-style-type: none"><li>成功，输出 delete success</li><li>失败，报错</li></ul>	Delete("/
Stat	<ul style="list-style-type: none"><li>远程文件路径 remoteFilePath</li></ul>	<ul style="list-style-type: none"><li>成功，输出 stat success</li><li>失败，报错</li></ul>	Stat("/m
Mkdir	<ul style="list-style-type: none"><li>待创建的远程目录文件路径 remoteFilePath</li></ul>	<ul style="list-style-type: none"><li>成功，输出 mkdir success</li><li>失败，报错</li></ul>	Mkdir("/r
Rename	<ul style="list-style-type: none"><li>原文件路径名 renameSrcPath</li><li>待更新的文件路径名 renameDestPath</li></ul>	<ul style="list-style-type: none"><li>成功，输出 reneme success</li><li>失败，报错</li></ul>	Rename( "/mydir/t
List	<ul style="list-style-type: none"><li>远程目录路径 remoteDirPath</li></ul>	<ul style="list-style-type: none"><li>成功，输出 show list success</li><li>失败，报错</li></ul>	List("/my

Demo如下



file\_v2\_4f0861c1-1b30-4af8-9ed9-23e12e7ec3dg\_API.mp4

28.47MB



## 4.2 性能测试

API读写时延如下

上传文件时延:	46.8546ms
下载文件时延:	4.539ms
新建目录时延:	2.7364ms
状态查看时延:	1.0394ms
文件列表时延:	1.0335ms
重命名时延:	2.1004ms
删除文件时延:	1.5534ms

## 五、演示 Demo

### 5.1 系统正常运行演示

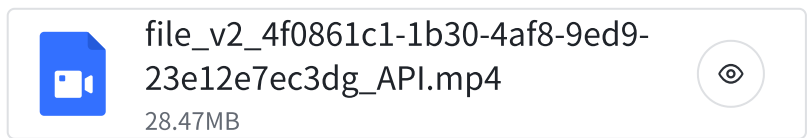
下列演示逻辑上有顺序性

操作顺序	API	测试	结果
1	Put	Put("D://testGPU.py", "/testGPU.py")	successes put
2	Get	Get("/testGPU.py", "D://testSuccess.py")	write to local success
3	Mkdir	Mkdir("/mydir")	mkdir success
4	Put	Put("D://testGPU.py", "/mydir/testGPU.py")	successes put
5	Stat	Stat("/mydir")	stat success filesize: "0" isDir: true
6	List	List("/mydir")	show list success metaList:{fileName : " /mydir
7	Get	Get("/mydir/testGPU.py", "D://testGPU.py")	write to local success
8	Rename	Rename("/mydir/testGPU.py", "/mydir/test.py")	reneme success
9	Get	Get("/mydir/test.py", "D://testRename.py")	write to local success



10	List	List("/mydir")	show list success metaList:{fileName: " /mydir,
11	Delete	Delete("/mydir/test.py")	delete success
12	Stat	Stat("/mydir")	stat success filesize: "0"isDir:true
13	List	List("/mydir")	空

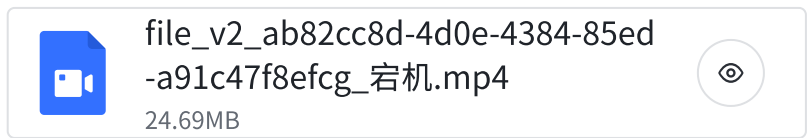
演示视频如下：



## 5.2 数据节点扩容及单点故障

流程如下

- 1.启动namenode 集群，注册datanode服务器
2. client连接集群，写入文件
- 3.注册新的datanode服务器
- 4.杀死老的datanode服务器，可以发现对应损坏的副本数据迁移到了新的 datanode服务器上
- 5.client读操作，client会读取可用的副本



# 六、项目总结与反思

## 6.1 目前仍存在的问题

经过了大半个月的开发，本团队已经完成了大部分基础功能的开发工作，目前存在的问题为：

1. 该项目似乎无法部署在linux系统下，在虚拟机中运行节点的main函数会直接假死，即既不报错也不运行。
2. 目前的一致性共识算法仍然不够成熟。
3. 虽然支持节点的自动扩容但是无法做到负载均衡和节点的自动迁移。

#### 4. 未添加纠删码等EC算法的存储策略

## 6.2 已识别出的优化项

1. 垃圾回收机制，目前的delete仅仅删除了元数据，但datanode的数据仍然保留，定期回收这类数据是接下来的优化项之一
2. datanode的负载均衡算法，datanode作为存储的核心，负载均衡可以使每一个datanode损坏的成本相近，避免严重事故的发生。
3. 除了负载均衡外，如果可以让不同副本不仅不存在同一个datanode而且不存在同一个机架中就可以有效避免大面积datanode离线导致所有副本全部丢失的可能。
4. 通过更加人性化的前端展示系统性能，并引入普罗米修斯进行性能测试。

## 6.3 架构演进的可能性

1. 对海量小文件存储的优化。
2. 引入分布式链路追踪方便后续优化调试，如zipkin等。
3. namenode中数据组织结构优化，以B-Tree的数据结构管理DFS的文件目录树。并对节点选择算法进行改善与优化。
4. datanode统一集成管理，实现监控大盘与自动预警，实现自动化运维管理。

## 6.4 项目过程中的反思与总结

### • 项目总结

1. 本次项目总的来说对与团队的每个成员都是有一次不错的历练，我们从一开始的陌生以及基础不同一同选题调研，形成了一个小的team，并且两天一次的组会使得我们的进度也可以有效推进，这让我们明白了一个好的团队管理的对于开发的重要性，这也是担任项目所无法实现的。每次组会的code review和code talking大家都积极的参与并表达自己的观点，各司其职的联调和单测，也让每个模块之间的同学更加熟悉。
2. GFS本身就并不是一个简单的项目，从一开始的调研就在慢慢学习，也会使用掘金等软件搜索相关的教程，让我们学到了最新的很多框架和知识，也把青训营的相关内容应用到了项目当中。
3. git对于项目的管理真的很有帮助，我们这次是第一次多人使用git进行项目管理，每位开发人员有自己的分支可以很方便的相互查看，再通过测试之后再把自己的部分合并到主分支的时候也有一种被认可的很大的成就感。

### • 项目反思

1. 在项目初期缺少相对详细具体的任务计划安排，团队成员在项目启动初期学习内容相对宽泛而不具体。同时也在项目初期阶段团队成员之间的交流不足，出现了一部分开发内容冲突的现象。
2. 项目整体架构设计不够完善，在后续仍出现了服务的修改，致使代码较大范围变动。