分布式系统大作业——分布式文件系统

姓名	李俊陶
班级	计算机科学与技术（系统结构方向）
学号	21307184
日期	2023年12月26日

一、题目要求

设计一个分布式文件系统。该文件系统可以是client-server架构，也可以是P2P非集中式架构。 要求文件系统具有基本的访问、打开、删除、缓存等功能，同时具有一致性、支持多用户特点。 在设计过程中能够体现在分布式课程中学习到的一些机制或者思想，例如Paxos共识、缓存更新机制、访问控制机制、并行扩展等。

要求：

- 编程语言不限，选择自己熟悉的语言，但是推荐用Python或者Java语言实现；
- 文件系统中不同节点之间的通信方式采用RPC模式，可选择Python版本的RPC、gRPC等；
- 文件系统具备基本的文件操作模型包括：创建、删除、访问等功能；
- 作为文件系统的客户端要求具有缓存功能即文件信息首先在本地存储搜索，作为缓存的介质可以是内存也可以是磁盘文件；
- 为了保证数据的可用性和文件系统性能，数据需要创建多个副本，且在正常情况下，多个副本不在同一物理机器，多个副本之间能够保持一致性（可选择最终一致性即延迟一致性也可以选择瞬时一致性即同时写）；
- 支持多用户即多个客户端，文件可以并行读写（即包含文件锁）；
- 对于上述基本功能，可以在本地测试，利用多个进程模拟不同的节点，需要有相应的测试命令或者测试用例，并有截屏或者video支持；
- 提交源码和报告，压缩后命名方式为：学号_姓名_班级

- 实验报告长度不超过20页；

加分项：

- 加入其它高级功能如缓存更新算法；
- Paxos共识方法或者主副本选择算法等；
- 访问权限控制；
- 其他高级功能；

二、解决思路

我在本次的项目中参照HDFS分布式文件系统的模型进行设计，在满足分布式系统设计的基本要求上，主要优势为系统有重要的可伸缩性，单个主服务器可以控制数百个块服务器：

文件结构：My_HDFS

```
My_HDFS
├─ namenode               // 命名节点
│
├─ datanode               // 数据节点
│
├─ secondarynamenode      // 命名节点备份存储
│
├─ client                 // 客户端
│
├─ file_system.proto      // Protocol Buffers定义文件
│
├─ file_system_pb2        // grpc产生文件
│
└─ file_system_pb2_grpc   // grpc产生文件
```

1. **命名节点（Namenode）**：负责维护文件系统的命名空间和元数据信息，包括文件的路径、权限、所有者等、负责处理客户端的元数据操作请求，例如创建文件、删除文件、获取文件元数据等，并通过心跳机制与DataNode通信，维护DataNode的可用性信息。
2. **命名节点备份存储（SecondaryNamenode）**：负责与NameNode交互，定时将下载NameNode中的元数据并在本地进行持久化存储，当NameNode需要恢复时，可从SecondaryNamenode下载文件元数据。
3. **数据节点（Datanode）**：负责实际存储文件数据，每个DataNode都维护一部分文件的数据。处理客户端的数据操作请求，例如读取文件数据、写入文件数据。支持数据复制，将文件数据在多个DataNode之间进行复制，以实现分区容错性。
4. **客户端（Client）**：提供用户接口，使用户能够通过命令行与分布式文件系统进行交互。实现了一些常用文件系统操作，如创建目录、进入目录、回退目录、创建文件、写入数据、读取数据、删除文件等。
5. **Protocol Buffers定义文件**：包含用于定义消息结构、字段和其他相关信息的 ProtoBuf 语法。

应用到的分布式系统概念以及原理等如下所示：

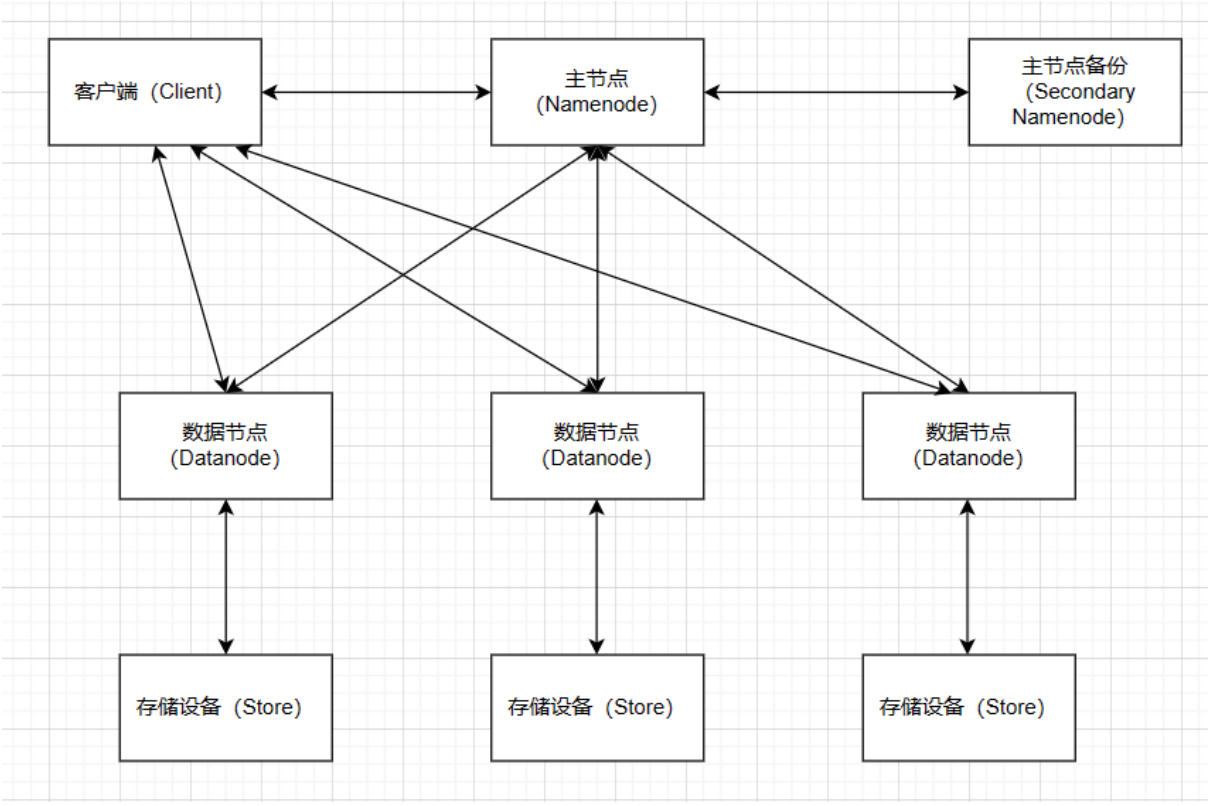
1. **分布式系统**：分布式文件系统是建立在多个节点上的文件系统，节点之间协同工作以提供高性能、高可用性和可伸缩性。分布式系统致力于通过将计算任务分配到多个计算机上来提高系统性能和容错性。
2. **通信机制**：通信是分布式系统中的基础，涉及到节点之间的消息传递。在本次项目实现中使用了gRPC作为通信框架，实现了高效的远程过程调用，同时采用了 Protocol Buffers 进行数据序列化。

3. **线程**：系统节点中使用线程的方式运行一些定期函数，例如发送心跳信息、清理过期节点等，使用多线程的方式可以避免陷入引起中断，造成性能下降。
4. **容错性**：分布式系统要具备容错性，即使在节点故障的情况下仍能够继续提供服务。实现中通过数据节点中数据的多次复制（副本机制）来保障系统在节点故障时的可用性。此外，通过DataNode定期向NameNode发送心跳消息，以确保各个节点的正常运行状态，有助于及时发现并处理节点故障，提高了系统的容错性。容错性的实现还包含了启动Secondnary NameNode对主NameNode的文件元数据进行备份，实现持久化存储，从而使得主节点能够从错误中正确恢复。
5. **两阶段提交协议**：使用两阶段提交协议，将数据节点分为主节点与次节点，保证写入操作的对于所有存储的数据节点通信时一致，从而保证数据存储的一致性。
6. **一致性**：文件系统需要保证一致性，即多个节点对于相同数据的访问要保持一致。本次项目中，应用了瞬时一致性，多个数据节点中的修改时同时发生的。
7. **内容分发**：文件系统中保证服务器与客户端缓存一致时，采用了客户端基于Pull的更新方式，这样就无需再服务器端保存有缓存的客户端，从而提升服务器的性能。
8. **并发控制**：并发控制是确保多个并发操作不会导致数据不一致的机制，引入了文件锁定机制，以确保在某一时刻只有一个客户端能够修改文件
9. **安全性**：安全性是分布式系统中至关重要的一个方面，涉及到用户身份验证、访问控制等问题。本次项目中中考虑了管理员、文件的所有者、读写权限等概念，建立了POSIX权限模型以确保系统的安全性。

说明：HDFS实现中将文件进行分片存储，分片信息存储在NameNode节点中，项目中进行简化：对文件不进行分片。

三、实现细节

实现的分布式文件系统体系结构流程图大致如下所示，箭头指向代表数据流动方向：



实现功能：

- **使用grpc在不同节点之间进行通信**

- **具有常用的文件操作**，包括：创建删除目录、进入以及后退目录、列出目录下文件、创建文件、打开文件、删除文件、写入数据、读取数据、对文件设置权限。
- **文件系统的客户端具有缓存功能**，文件信息首先在本地存储搜索，若未搜索到再去请求服务器，同时附加实现了缓存更新算法
- **数据的可用性高**，数据在不同的物理机器上创建了多个副本，且多个副本之间能够保持瞬时一致性。
- **支持多用户即多个客户端**，并行读写文件（锁的粒度为文件大小，读写锁能够支持并发读取、独占修改）
- **访问权限控制**，将用户角色区分为管理员与普通用户，管理员可以设置所有文件的读写权限，文件所有者可以设置对应文件的读写权限，参照posix权限模型实现。

因为源代码实现内容较多（包含一些错误输出信息），因此在下面具体展示中节选重要代码进行展示，详细代码以及注释可以参照文件夹中的源代码部分

1. grpc实现方式

- 在文件 file_system.proto 中定义了服务接口、消息类型以及服务方法

```
service FileSystemService {
    // File metadata management
    rpc CreateFile(FileRequest) returns (FileResponse);
    rpc DeleteFile(FileRequest) returns (FileResponse);
    rpc RenameFile(RenameRequest) returns (FileResponse);
    rpc GetFileMetadata(FileRequest) returns (FileMetadataResponse);
    // More service
    ...
}
message FileRequest {
    string file_path = 1;
    string owner = 2;
    string permissions = 3;
}
// More message
...
```

- 使用 grpc.server创建了gRPC服务器，并将服务器实例添加为服务的实现，将服务器绑定到指定的端口，然后通过启动服务器，使其开始监听gRPC请求。

```
name_node = NameNode()  # 创建一个 NameNode 实例
server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
file_system_pb2_grpc.add_FileSystemServiceServicer_to_server(name_node, server)
server.add_insecure_port('[::]:50051')
server.start()
```

- 在客户端中，创建gRPC客户端通道，用于与服务器进行通信。 通过调用stub对象的方法（如read_data），实现了对远程gRPC服务的调用。这些调用包含了传递的消息参数，通过Protocol Buffers序列化和反序列化。

```
data_node_channel = grpc.insecure_channel(selected_data_node)
            data_node_stub =
file_system_pb2_grpc.FileSystemServiceStub(data_node_channel)
            data_node_response =
data_node_stub.ReadData(file_system_pb2.DataRequest(file_path=file_path.strip(),of
fset=int(offset)))
```

2. 文件操作：

- 客户端交互页面：
  将终端通过文本提示用户输入，用户能够查看到当前所在目录，并通过help命令查看文件系统所支持的命令格式。将用户输入的命令进行合理划分之后，交给具体的函数实现各个功能。

```
if command == "help":...
elif command.startswith("create_file"):...
elif command.startswith("get_m"):...
else:...
```

- 创建目录、进入目录、后退目录、列出目录下文件：
  目录数据均记录在namenode当中，在本次项目中使用树状结构对目录信息进行记录（模拟真实情况中数据存储在namenode节点的内存中），实现树状结构需要定义类以及一些类方法（更多是数据结构的知识在此不详细展开）。
  客户端根据用户的命令构建相应的gRPC请求对象，并通过gRPC连接调用NameNode提供的相应接口，传递目录路径、新目录的名称或其他相关信息。
  NameNode 接收到请求后，根据请求的类型执行相应的目录操作，比如创建目录、列出目录内容、改变当前工作目录等，涉及到在文件系统的元数据中更新目录结构的信息，并且在执行结束之后响应客户端执行结果。

```
def MakeDirectory(self, request, context):  # 创建目录
    directory = request.directory
    new_directory = request.new_directory
    parent_node = self.traverse_directory(directory)
    if parent_node is not None:
        parent_node.add_child(new_directory)
        return file_system_pb2.FileResponse(success=True, message="Directory
created successfully.")
    else:
        return file_system_pb2.FileResponse(success=False, message="Parent
directory not found.")
def ListFiles(self, request, context): # 列出目录
    directory = request.directory
    directory_node = self.traverse_directory(directory)
    if directory_node is not None:
        files = directory_node.list_children()
        return file_system_pb2.FileListResponse(files=files)
    else:
        return file_system_pb2.FileListResponse()
```

```
elif command == "cd..": # 回退目录
    self.current_directory = "/".join(self.current_directory.split("/")[:-1])
elif command.startswith("cd"):  # 进入目录
    _, directory = command.split(maxsplit=1)
    self.current_directory = f"{self.current_directory}/{directory.strip()}"
```

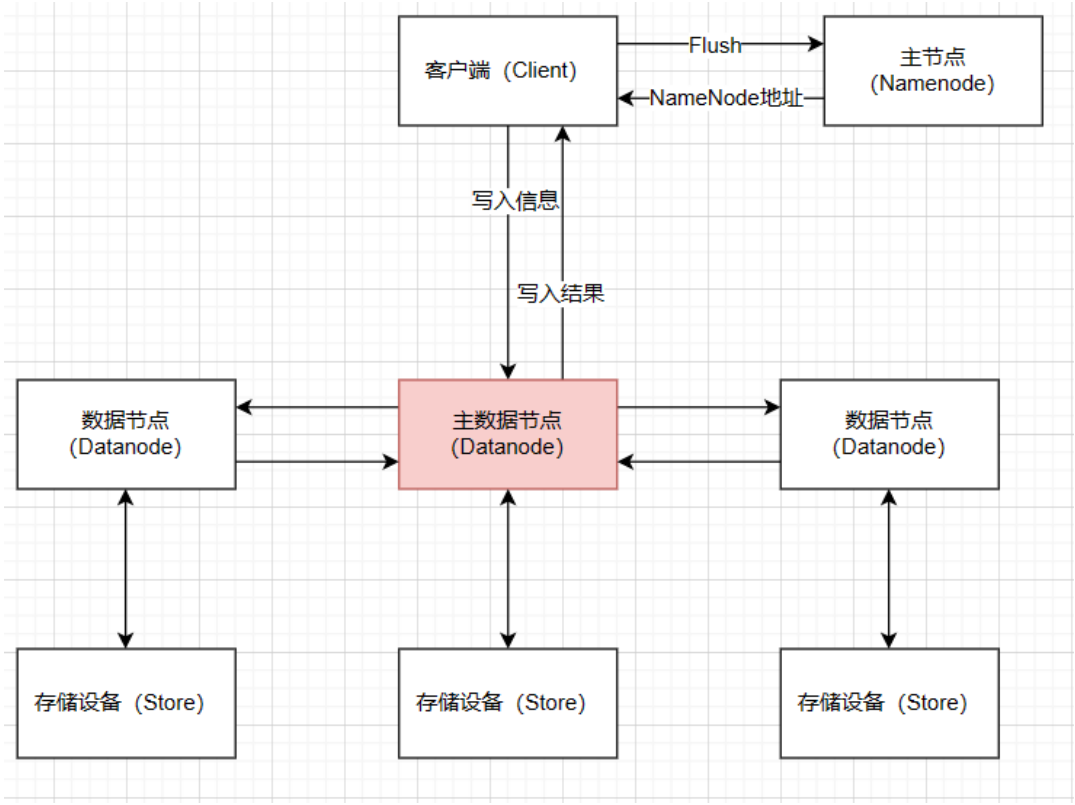
- 创建文件： 客户端通过gRPC连接调用NameNode提供的相应接口，传递文件路径、拥有者、权限信息。

```
def CreateFile(self, request, context):
    file_path = request.file_path
    owner = request.owner
    permissions = request.permissions
    if file_path in self.file_metadata:  # 如果该文件已经存在
        return file_system_pb2.FileResponse(success=False, message="File already
exists.")
    # 获取文件所在的目录
    directory = os.path.dirname(file_path)
    parent_node = self.traverse_directory(directory)
    if parent_node is not None: # 更新目录结构
        parent_node.add_child(os.path.basename(file_path))
        self.file_metadata[file_path] =
file_system_pb2.FileMetadata(file_path=file_path, size=0, owner=owner,
permissions=permissions, version=0)
        return file_system_pb2.FileResponse(success=True, message=f"File
'{file_path}' created successfully.")
    else:
        return file_system_pb2.FileResponse(success=False, message=f"Directory
'{directory}' not found.")
```

- 删除文件、重命名文件： 操作过程与创建文件类似，只需要修改namenode中用于管理文件的数据结构即可，非本次项目技术重点(故省略)

- 打开文件：
  客户端调用函数get_lock获得对于需要打开文件的读者锁，通过gRPC连接调用NameNode提供的相应接口，传递文件路径、用户，NameNode判断是否有权限打开文件并返回结果，详细实现见下文的锁机制。(打开文件之后可以执行写入、读取、关闭三个操作)

- 写入数据



  - 客户端首先将文件的最新版本缓存到本地中，然后在本地缓存中进行写并将缓存的dirty位置位，将文件版本+1，随后在关闭文件时，若文件为dirty则进行flush。

```
    def write_file_content(self, file_path, data, offset):
      # 首先更新缓存中的内容，确保是在最新数据中写入
      self.read_file_content(file_path, offset)
      ...# 修改数据
      self.file_cache[file_path]["dirty"] = True  #设置脏位
  def flush_cache(self, file_path):
      if self.file_cache[file_path]["dirty"]:
          self.file_cache[file_path]["version"] += 1
      # 获取 DataNode 地址
      response = self.name_node_stub.WriteFile(
          file_system_pb2.FileRequest(file_path=file_path.strip(),
owner=self.client_id))
      if response.success:
          data_node_addresses = response.data_node_addresses
          selected_data_node = data_node_addresses[0]
          # 建立选定主节点
          data_node_channel = grpc.insecure_channel(selected_data_node)
          data_node_stub =
file_system_pb2_grpc.FileSystemServiceStub(data_node_channel)
          # 刷写缓存中的数据到 DataNode
          data_node_response = data_node_stub.WriteData(
              file_system_pb2.DataRequest_main(
                  file_path=file_path.strip(),
                  data=self.file_cache[file_path]["content"],
                  offset=0,
                  nodes=data_node_addresses
```

```
                )
            )
        ...# 一些输出信息省略
```

- 客户端通过gRPC连接调用NameNode提供的相应接口，传递文件路径、用户信息。NameNode接受请求，**判断用户是否具有写入文件的权限。**满足权限，若第一次写入则返回多个随机的DataNode地址(**可以使用一些负载均衡算法**)；若是后续修改则返回存有文件的DataNode地址。客户端从返回的地址中，选择主DataNode节点进行连接并传输需要写入的文件数据。

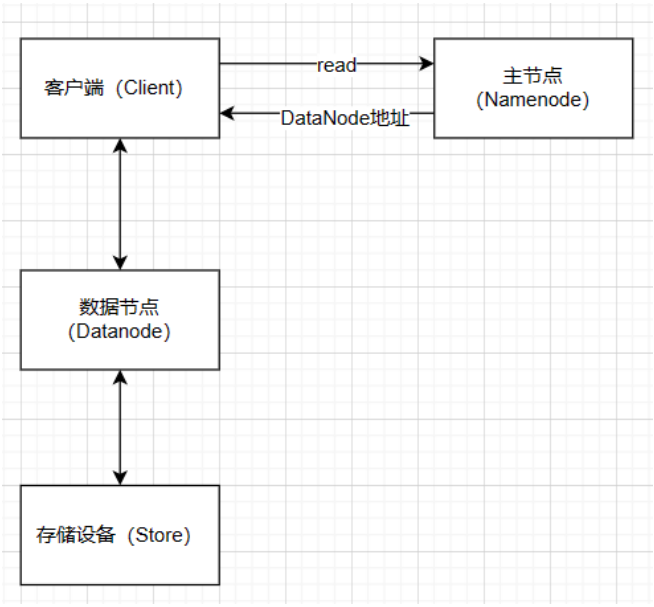
```
  def WriteFile(self, request, context):
      file_path = request.file_path
      num_replicas = 3  # 假设每个数据块有3个副本
      ... # 一些必要目录判断（重复多次了）
      # 文件已存在，返回DataNode地址
      data_nodes = self.ChooseDataNodes(file_path, num_replicas)
      return file_system_pb2.DataNodeListResponse(success=True,
data_node_addresses=data_nodes)
  def ClientWriteComplete(self, request, context):
      ...# 处理客户端写入完成的消息
      # 更新元数据
      new_metadata = file_system_pb2.FileMetadata(
          file_path=file_path,
          size=self.file_metadata[file_path].size,
          data_node_addresses=data_node_addresses,
          owner=self.file_metadata[file_path].owner,
          permissions=self.file_metadata[file_path].permissions,
          version=version,
      )
      self.file_metadata[file_path] = new_metadata
      return file_system_pb2.FileResponse(success=True, message="Write
complete message processed.")
```

- 主DataNode节点收到客户端请求后，**使用两阶段提交协议保证写入的原子性。**主DataNode节点向多个次节点发送Vote-request信息，次节点判断之后根据自身情况返回Vote-commit或者Vote-abort信息，主节点统计返回信息并作出最终决定，发送Global-commit或者Global-abort给次节点，所有节点对于数据作出相同响应。

```
  def WriteData(self, request, context):
      # 获取主节点和次节点列表
      main_node = request.nodes[0]
      secondary_nodes = request.nodes[1:]
      # 阶段一：向所有次节点发送Prepare请求
      prepare_responses = []
      for secondary_node in secondary_nodes:
          vote_request = file_system_pb2.FileRequest(file_path=file_path)
          with grpc.insecure_channel(secondary_node) as channel:
              stub = file_system_pb2_grpc.FileSystemServiceStub(channel)
              prepare_response = stub.Prepare(vote_request)
```

```
                prepare_responses.append(prepare_response)
        if all(response.success for response in prepare_responses): # 检查所有
Prepare响应
            # 若所有相应均为Vote-Commit 阶段二：向所有次节点发送Global-Commit
            for secondary_node in secondary_nodes:
                with grpc.insecure_channel(secondary_node) as channel:
                    stub = file_system_pb2_grpc.FileSystemServiceStub(channel)
                    commit_response = stub.Commit(request)
            ...# 更新数据存储
            return file_system_pb2.FileResponse(success=True, message=f"write
completed")
        else:
            ...# 如果有任何Prepare请求失败，阶段二：向所有次节点发送Global-Abort
    def Prepare(self, request, context):
        # 进行简化，都返回可以Commit 实际文件系统需要要进行判断
        return file_system_pb2.FileResponse(success=True,
message=f"Vote_Commit")
    def Commit(self, request, context):
        # 更新数据存储
        current_data = self.data_storage[file_path]
        updated_data = current_data[:offset] + data + current_data[offset +
len(data):]
        self.data_storage[file_path] = updated_data
        return file_system_pb2.FileResponse(success=True, message="Copy
successful")
```

- 读取数据



客户端首先在缓存中寻找是否有对应文件内容，并通过与NameNode通信获取该文件的最新版本号。若缓存中是最新版本，则直接读取缓存，若不存在缓存或缓存过期则运行读取的过程。

客户端通过gRPC连接调用NameNode提供的相应接口，传递文件路径、用户信息。NameNode接受请求，判断用户是否具有读取文件的权限，若有则返回多个存储有该文件内容的DataNode节点地址。客户端从返回的地址多个地址中，选择最合适的DataNode节点进行连接并读取文件数据（本次实验中使用的是随机选择，具体实践中应考虑网络耗费等）

```
def read_file_content(self, file_path, offset):
    # 尝试从缓存中获取文件内容
    cached_entry = self.file_cache.get(file_path)
    file_response =
self.name_node_stub.GetFileVersion(file_system_pb2.FileRequest(file_path=file_path
.strip()))
    file_version = file_response.version
    if cached_entry:
        cached_content = cached_entry["content"]
        cached_version = cached_entry["version"]
        if cached_version >= file_version:
            return cached_content[offset:]
    if file_version == 0:
        self.file_cache[file_path] = {"content": b"", "version": file_version,
"dirty": False}
        return None
    # 如果缓存中不存在，从 DataNode 获取文件内容
    response = self.name_node_stub.GetDatanode(
        file_system_pb2.FileRequest(file_path=file_path.strip(),
owner=self.client_id))
    if response.success:
        data_node_addresses = response.data_node_addresses
        selected_data_node = random.choice(data_node_addresses)
        data_node_channel = grpc.insecure_channel(selected_data_node)
        data_node_stub =
file_system_pb2_grpc.FileSystemServiceStub(data_node_channel)
        data_node_response = data_node_stub.ReadData(
            file_system_pb2.DataRequest(
                file_path=file_path.strip(),
                offset=int(offset),
            )
        )
        # 将获取到的文件内容放入缓存
        self.file_cache[file_path] = {"content": data_node_response.data,
"version": file_version, "dirty": False}
        return data_node_response.data[offset:]
    else:
        return None
def GetDatanode(self, request, context):  # NameNode返回存有对应文件的数据节点
    ... #实现较简单 具体代码可以看源代码模块
def GetFileVersion(self, request, context): # NameNode返回存有对应文件的版本号
    ... #实现较简单 具体代码可以看源代码模块
def ReadData(self, request, context): #DataNode返回文件具体内容
    ... #实现较简单 具体代码可以看源代码模块
```

- 关闭文件
  与打开文件类似，客户端调用函数unlock释放对应文件的读写锁，如果缓存中的文件为脏数据则进行flush。通过gRPC连接调用NameNode提供的相应接口，传递文件路径、用户，NameNode对文件解锁并返回结果，详细实现见下文的锁机制。

3. 缓存与缓存更新机制：

在本次项目中实现了缓存以及缓存更新，客户端读写一个文件时都先从本地缓存(使用字典来模拟真实客户端的内存)中寻找是否有相应的文件，若没有则再去服务器中读取。与此同时，**客户端会向NameNode节点请求对应文件的最新版本号，用pull的策略来保证服务器与本地缓存的一致性。**

采用了`write-back`(回写的缓存更新策略)，为每个本地缓存中文件维护一个`dirty`位，若进行了写入操作则置`dirty`位并在最终关闭文件的时候向服务器写回(flush)。`Write-back` 缓存写入策略的主要优点包括提高性能、减少主存带宽需求、累积写入以减少写回次数，并且通过版本号以及锁机制保证一致性。
代码在上文`write_data`与`read_data`操作中有过涉及，因此不再赘述。

4. 多副本之间一致性：

为了保证数据的可用性与文件系统的性能，数据创建了多个副本，且运行多个DataNode程序模拟多个机器。读取数据时，选择一个服务器的副本进行读取；写入数据时，**采用瞬时一致性**，客户端首先与NameNode通信获取写入DataNode地址，然后与主DataNode通信并传递写入数据。主DataNode与次DataNode之间采用**二阶段提交协议保证写入的一致性**（具体过程见上文写入数据操作）。两阶段提交协议保证了分布式系统中的事务一致性，即所有节点都要么提交事务，要么中止事务。

有关文件的元数据信息存储在NameNode当中，并且为了恢复，定时将数据信息传递给Secondary NameNode进行持久化存储，在此过程中保证NameNode中元数据的一致性。

5. 多用户并行读写(锁机制)：

文件系统支持多用户并发访问，采用读写锁的机制来保证不会冲突，读写锁以元数据的形式存储在NameNode节点中。读写锁能够读操作频繁的情况下允许多个线程同时读取数据，但在写操作进行时禁止读取和其他写操作。

```
def LockFile(self, request, context):
    ...# 读取
    lock = self.file_locks[file_path]
    lock.lock_time = time.time()  # 更新锁的时间
    if lock_type == "read":
        ...# 权限检查
        if lock.writer is None or lock.writer == user:
            lock.readers.add(user)
            return file_system_pb2.FileResponse(success=True,message=f"File opened
successfully with {lock_type} lock.")
        else:
            return file_system_pb2.FileResponse(success=False,
message="File is writen by another user,can not read.")
    elif lock_type == "write":
        ...# 权限检查
        if (lock.readers == {user} or not lock.readers) and (lock.writer is None
or lock.writer == user):
            lock.writer = user
            return file_system_pb2.FileResponse(success=True,
                    message=f"File opened successfully with {lock_type} lock.")
        else:
            return file_system_pb2.FileResponse(success=False,message="File is
read by other user, can not write.")
    return file_system_pb2.FileResponse(success=False, message="File is locked by
another user.")
```

```
def unlock_file(self, file_path, user):
    if file_path in self.file_locks:
        lock = self.file_locks[file_path]
        if user in lock.readers:
            lock.readers.remove(user)
        if lock.writer == user:
            lock.writer = None
        return True
    return False
```

设置文件锁超时机制：有助于避免死锁情况的发生，确保即使在异常情况下也能够及时释放锁资源。其次，超时机制提高了系统的可用性，防止长时间持有锁资源导致系统阻塞。第三，它促使资源的及时释放，防止长时间占用锁资源，提高系统的资源利用率。此外，文件锁超时机制还有助于降低锁等待时间，提高系统的响应性能，并能够在意外场景下保持系统的稳定性。

```
# 启动定期清理过期锁线程
lock_cleaning_thread = threading.Thread(target=name_node.clean_expired_locks,
daemon=True)
    lock_cleaning_thread.start()
def clean_expired_locks(self):
    while True:
        time.sleep(5)  # 每隔5秒清理一次
        current_time = time.time()
        for file_path, lock in list(self.file_locks.items()):
            if current_time - lock.lock_time > self.lock_timeout:
                self.unlock_file(file_path, lock.writer)
                for reader in lock.readers.copy():
                    self.unlock_file(file_path, reader)
```

6. 访问权限控制：

本次项目中参考POSIX权限模型，将文件和目录权限分为三个基本类别：所有者、组和其他用户。每个类别都有读（Read）、写（Write）和执行（Execute）三种权限，分别用数字表示为100、010和001。通过三个数字的组合，分别对应所有者、组和其他用户的权限，形成一个三位数的权限表示。通过chmod命令可以动态地调整文件和目录的权限，保障系统安全性和数据隐私。在本次实验中将Client1设置为管理员，即可以对所有文件更改权限；当用户在创建文件时会将自己设置为文件的拥有者，可以对拥有的文件更改权限。这些数据保存在NameNode该文件的对应元数据中。

客户端通过gRPC连接调NameNode的`ChangePermissions`接口，传递文件路径、客户端身份标识和新的权限信息。(grpc过程与之前一致，因此不展示详细代码) NameNode 接收到请求后，进行权限校验，确保请求的用户有足够的权限来更改文件的权限。若权限校验通过，NameNode 就会执行更改权限的操作并相应客户端

```
def ChangePermissions(self, request, context):
    file_path = request.file_path
    owner = request.owner
    new_permissions = request.permissions
    # 根据 file_path 获取文件元数据并需要修改文件权限
    file_metadata = self.file_metadata[file_path]
```

```
    if not file_metadata:
        return file_system_pb2.FileResponse(success=False, message="File not
found.")
    if owner == "1" or owner == file_metadata.owner:
        file_metadata.permissions = new_permissions
        return file_system_pb2.FileResponse(success=True, message="Permissions
changed successfully.")
    else:
        return file_system_pb2.FileResponse(success=False, message="Permissions
denied.")
def AddGroup(self, request, context):
    # 根据 file_path 获取文件元数据
    file_metadata = self.file_metadata[file_path]
    # 在这里根据实际需要修改文件权限
    file_metadata.group.append(user)
    return file_system_pb2.FileResponse(success=True, message="User added
successfully.")
def CheckPermission(self, file_path, user): # 将字符串类型的权限转化为二进制
    file_metadata = self.file_metadata[file_path]
    permission = file_metadata.permissions
    if user == file_metadata.owner:
        binary_permission = bin(int(permission[0]))[2:].zfill(3)
    elif user in file_metadata.group:
        binary_permission = bin(int(permission[1]))[2:].zfill(3)
    else:
        binary_permission = bin(int(permission[2]))[2:].zfill(3)
    return binary_permission
```

7. NameNode容错性：

在本次项目中参照HDFS中的结构简化实现了Secondnary NameNode节点，持久化存储主节点中的文件元数据，用于故障修复。在HDFS中，Secondary NameNode的其主要目的是协助主要的NameNode执行检查点操作，以防止NameNode的元数据损坏或丢失，涉及合并编辑日志并创建新镜像等多个操作。本次项目中简化为，定期从NameNode中获取文件元数据并持久化存储，当NameNode发生故障丢生文件元数据时，可以从Secondnary NameNode获取并进行恢复。Secondary Namenode类采用写入文档模拟持久存储，主NameNode中在初始化时使用grpc调用读取备份文件元数据，关键函数如下所示：

```
def perform_checkpoint(self):   # 每隔一段时间执行checkpoint
    response = self.name_node_stub.GetMetadata(file_system_pb2.FileRequest())
    file_metadata_map = {metadata.file_path: metadata for metadata in
response.metadata}
    # 模拟写入持久存储
    self.save_metadata_to_file(file_metadata_map)
def save_metadata_to_file(self, metadata_map):
    with open(self.metadata_file_path, "w") as file:
        for file_path, metadata in metadata_map.items():
            file.write(f"{file_path} {metadata.SerializeToString().hex()}\n")
def load_metadata_from_file(self):
    metadata_map = {}
    with open(self.metadata_file_path, "r") as file:
```

```
            for line in file:
                parts = line.split()
                file_path = parts[0]
                metadata_hex = parts[1]
                metadata =
file_system_pb2.FileMetadata().FromString(bytes.fromhex(metadata_hex))
                metadata_map[file_path] = metadata
        return metadata_map
    def PeriodicCheckpoint(self):    # 每隔一段时间执行一次检查点
        while True:
            time.sleep(60)
            self.perform_checkpoint()
    def GetMetadata(self, request, context):     #通过grpc将元数据传递给主Namenode
        metadata_map = self.load_metadata_from_file()
        # 返回 SecondaryNameNode 中保存的元数据信息
        metadata_list = list(metadata_map.values())
        return file_system_pb2.MetadataResponse(metadata=metadata_list)

    if __name__ == '__main__':
        ...
        # 启动定期执行检查点的线程
        checkpoint_thread =
    threading.Thread(target=secondary_name_node.PeriodicCheckpoint, daemon=True)
        checkpoint_thread.start()
```

8. 心跳机制：

通过gRPC通信，DataNode定期向NameNode发送心跳消息，其中包含DataNode的地址。NameNode接收心跳消息后，更新维护的`data_nodes`字典，记录各个DataNode最近一次的活跃时间戳。同时，NameNode通过定期清理线程，检测`data_nodes`字典中的时间戳，识别并清理长时间未发送心跳的DataNode，以维持文件系统的稳定性和可靠性。这一心跳机制有助于实时监测DataNode的状态，及时处理可能的故障或失效，从而提高整个分布式文件系统的鲁棒性。

```
# 启动定期清理线程
cleaning_thread = threading.Thread(target=name_node.clean_expired_data_nodes,
daemon=True)
cleaning_thread.start()
def Heartbeat(self, request, context):
   # 处理DataNode的心跳信息
   data_node_address = request.data_node_address
   self.data_nodes[data_node_address] = time.time()  # 更新时间戳
   return file_system_pb2.HeartbeatResponse(status="OK")   # 返回心跳响应
def clean_expired_data_nodes(self):
   # 定期清理过期的DataNode信息
   while True:
       time.sleep(60)  # 每隔60秒清理一次
       current_time = time.time()
       expired_nodes = [addr for addr, timestamp in self.data_nodes.items() if
                        current_time - timestamp > 60]  # 假设超过60秒没有心跳就
认为过期
```

```
        for addr in expired_nodes:
            del self.data_nodes[addr]
```

```
# 启动心跳发送进程
heartbeat_thread = threading.Thread(target=data_node.send_heartbeat, daemon=True)
heartbeat_thread.start()
def send_heartbeat(self):
    while True:
        time.sleep(10)  # 定期发送心跳消息给NameNode
        print("Sending heartbeat to NameNode")
        with grpc.insecure_channel(self.name_node_address) as channel:
            stub = file_system_pb2_grpc.FileSystemServiceStub(channel)
            response =
stub.Heartbeat(file_system_pb2.HeartbeatRequest(data_node_address=self.data_node_a
ddress))
```

四、结果展示

1. 构造运行环境
   下载安装成功依赖环境之后，运行Secondnary NameNode节点；运行NameNode服务器监听localhost:50051；运行五个DataNode服务器(每份文件共有三份副本，分别存在三个DataNode服务器中)，分别监听localhost:50052、localhost:50053、localhost:50054、localhost:50055、localhost:50056。启动三个客户端，Client1(管理员)、Client2、Client3。
2. 展示心跳机制
   DataNode启动之后，与NameNode之间显示心跳信息。
3. 展示文件文件操作
   Client1创建目录、文件，进入目录、回退目录、重命名文件、删除文件、打开文件`test`，写入数据`happy`并读取，最后关闭文件。
4. 展示多用户并发访问
   Client2、Client3同时对之前写入的文件`test`进行读取，可以实现并发读取(**读取的是不同副本，一致性的体现**)；此时，Client2想要写入`test`，会失败(因为只支持独占写)；随后Client2与Client3均关闭文件，Clinet2再次打开并写入数据`new year`,此时Clinet3想要打开文件进行读写会失败，因为Client2获得了写锁。
5. 展示缓存一致性机制
   在上面的步骤中，Client1会对`test`文件在本地形成缓存，但是随后Client2又修改文件内容；此时，Client1再次读取`test`文件，缓存能够命中但是发现过时，于是从服务器读取，从而保证一致性，读出数据为`happy new year`。
6. 展示权限模块
   Client2创建文件，权限为711，即只有文件所有者可以进行读写，Client3无法进行读写。Client1修改权限为777，使得Client3可以读写(还有修改权限组成员的部分在此不展示)。
7. 容错性——NameNode恢复
   运行过程中，重启NameNode程序，可以观察到仍能正确恢复到上一次检查点的状态，文件仍然存在。

测试过程如上所示，**运行结果与解释参照提交的video部分**，video部分有具体运行情况与解释，用以更好说明效果。

五、实验中遇到的问题

在实验过程中，我遇到了一些挑战和问题，主要包括：

gRPC是一个高性能、开源和通用的远程过程调用（RPC）框架，熟练高效使用需要一定的学习成本。通过阅读官方文档和示例代码，以及查阅相关资料，逐步掌握了gRPC的基本原理和使用方法。将课程理论内容实践为项目中的一部分也存在着一些挑战，一致性、并发控制、容错性、安全性等等，这些分布式系统的特性在课程中有许多不同实现方式，选择适合本项目的实现方式并顺利实现需要一定的尝试与调整。通过深入理解内容并多次修改代码，最终达到了希望的效果。

六、实验总结

通过完成这个分布式文件系统的实验，我深刻体会到分布式系统设计的复杂性。在这个过程中，我需要考虑多个节点的协同工作，同时处理各种潜在的问题，如网络故障和硬件故障。系统的容错性、一致性和性能等方面都是需要仔细思考和权衡的问题。

学习和使用gRPC是一个积极的体验。gRPC作为高效的通信框架，简化了节点间的远程过程调用，提高了系统的效率。尽管学习曲线较陡，但一旦掌握，它为分布式系统的通信提供了便利。除此之外，真正将课本中的理论内容实现过后，对于书中的概念原理等等都加深了认识。

通过这个实验，我不仅加深了对分布式系统设计的理解，还学到了如何应用各种技术来构建可靠和高性能的分布式文件系统。这些经验将对我未来在分布式系统领域的工作和学习产生积极的影响。