分布式系统大作业

姓名：薛锦俊

班级：计科五班

学号：21307296

**分布式的健值存储系统**

1. 题⽬ ：

设计并实现⼀个分布式键值（key-value）存储系统，可以是 基于磁盘的存储系统，也可以是基于内存的存储系统，可以是主从 结构的集中式分布式系统，也可以是 P2P 式的⾮集中式分布式系 统。能够完成基本的读、写、删除等功能，⽀持缓存、多⽤户和数 据⼀致性保证, 提交时间为考试之后⼀周内。

2. 要求 :

1）、必须是分布式的键值存储系统，⾄少在两个节点或者两个进程中测试；

2）、可以是集中式的也可以是⾮集中式；

3）、能够完成基本的操作如：PUT、GET、DEL 等；

4）、⽀持多⽤户同时操作；

5）、⾄少实现⼀种⾯向客户的⼀致性如单调写；

6）、需要完整的功能测试⽤例；

7）、涉及到节点通信时须采⽤ RPC 机制；

8)、提交源码和报告，压缩后命名⽅式为：学号\_姓名\_班级

加分项： 1）、具备性能优化措施如 cache 等； 2）、具备失效容错⽅法如：Paxos、Raft 等； 3）、具备安全防护功能； 4）、其他⾼级功能；

1. 实验分析

分布式键值存储是一种分布式数据存储系统，它将数据存储为键值对的形式，并将这些键值对分散在多个节点上。每个节点都可以独立地处理一部分数据，从而实现高可用性和可扩展性。在设计分布式系统时， CAP（一致性，可用性，分区容错性） 定理很重要。

1. 环境配置

Python 3.9, grpcio, grpcio-tools, redis,

Redis-x64-3.2.100Redis-x64-3.2.100

1. 代码实现

实现原理：

节点之间的通信使用grpc，节点开始服务之后就可以开始监听客户端的请求，并且持续保持监听操作。

grpc服务部分：**KeyValueStoreServicer** 类作为 gRPC 服务端的实现，提供了 PUT、GET、DELETE、SET、SYNC\_DATA 操作。

Put为写操作，获取键对应的锁，写入数据，然后调用 sync\_data 方法将数据同步到其他节点； Get为读操作，通过 gRPC 请求获取数据，返回键对应的值；Delete为删除操作，获取键对应的锁，删除键值对，然后同步数据到其他节点；SyncData是数据同步操作，当 一个节点收到来自其他节点的数据同步请求时，获取锁并更新本地数据，确保节点之间的数据一致性。

在中间件中可以创建多个grpc通道分别连接到多个节点，创建多个存根以进行grpc调用。多用户同时操作就会出现很多一致性问题和冲突问题。对于多线程问题就不可避免的使用到锁机制，但是简单的锁机制仅仅对单一节点的多线程有效果。而在分布式环境中，需要更加复杂的机制来确保一致性和解决冲突。在分布式系统中，可能会遇到以下一致性和冲突问题：并发写入问题： 多个用户或节点同时对同一数据进行写入，可能导致数据不一致；读取过期数据问题： 当一个节点写入数据后，其他节点可能仍然读取过期的数据；分布式事务问题： 多个节点之间的事务一致性问题，例如某些节点成功提交而其他节点失败。需要通过实现一致性来解决冲突问题。在本次项目中使用了分布式锁机制。确保同一时刻只有一个用户能够对某个键进行操作，并且要在同步数据之后才能释放锁。同时还需要增加简单的负载均衡操作，防止对于某个节点出现过载现象。在本实验中负载均衡仅采用最简单的随机分配方法。此外中间件还增加了缓存功能，用于提高系统性能。

在客户端则需要进行模拟多用户的操作。通过采用多线程并发执行操作来模拟多用户同时操作的情形。然后通过用中间件提供的接口来实现功能，并进行测试。

具体实现：

编写storage.proto文件，并利用其生成storage\_pb2.py文件和storage\_pb2\_grpc.py文件。

storage.proto具体信息如下：

syntax = "proto3";  
  
package storage;  
  
service KeyValueStore {  
 rpc Put(Request) returns (Response);  
 rpc Get(Request) returns (Response);  
 rpc Delete(Request) returns (Response);  
 rpc Set(Request) returns (Response);  
 rpc SyncData(SyncRequest) returns (SyncResponse);  
}  
  
message Request {  
 string key = 1;  
 string value = 2;  
 string source\_node = 3; // 新增的字段  
}  
  
message Response {  
 string value = 1;  
}  
  
message SyncRequest {  
 map<string, string> data = 1;  
 string source\_node = 2;  
}  
  
message SyncResponse {  
 string message = 1;  
}

**对该文件的信息进行解释：**

**KeyValueStore这个服务有五个远程过程调用（RPC）：**

* **Put: 存储键值对。**
* **Get: 获取键对应的值。**
* **Delete: 删除指定的键值对。**
* **Set: 设置键值对，如果不存在则创建，存在则更新。**
* **SyncData: 同步数据。**

**每个 RPC 都有一个请求消息和一个响应消息**

**message Request { ... }: 定义了一个消息类型 Request，用于作为 gRPC 服务中 RPC 的请求消息。Request 消息有三个字段：**

**key (字段号 1): 存储键的字符串。**

**value (字段号 2): 存储值的字符串。**

**source\_node (字段号 3): 新增的字段，存储请求的源节点。**

**message Response { ... }: 定义了一个消息类型 Response，用于作为 gRPC 服务中 RPC 的响应消息。Response 消息有一个字段：**

**value (字段号 1): 存储值的字符串。**

**message SyncRequest { ... }: 定义了一个消息类型 SyncRequest，用于作为 SyncData RPC 的请求消息。SyncRequest 消息有两个字段：**

**data (字段号 1): 存储键值对的映射，使用字符串键和字符串值。**

**source\_node (字段号 2): 存储请求的源节点。**

**message SyncResponse { ... }: 定义了一个消息类型 SyncResponse，用于作为 SyncData RPC 的响应消息。SyncResponse 消息有一个字段：**

**message (字段号 1): 存储一条消息的字符串。**

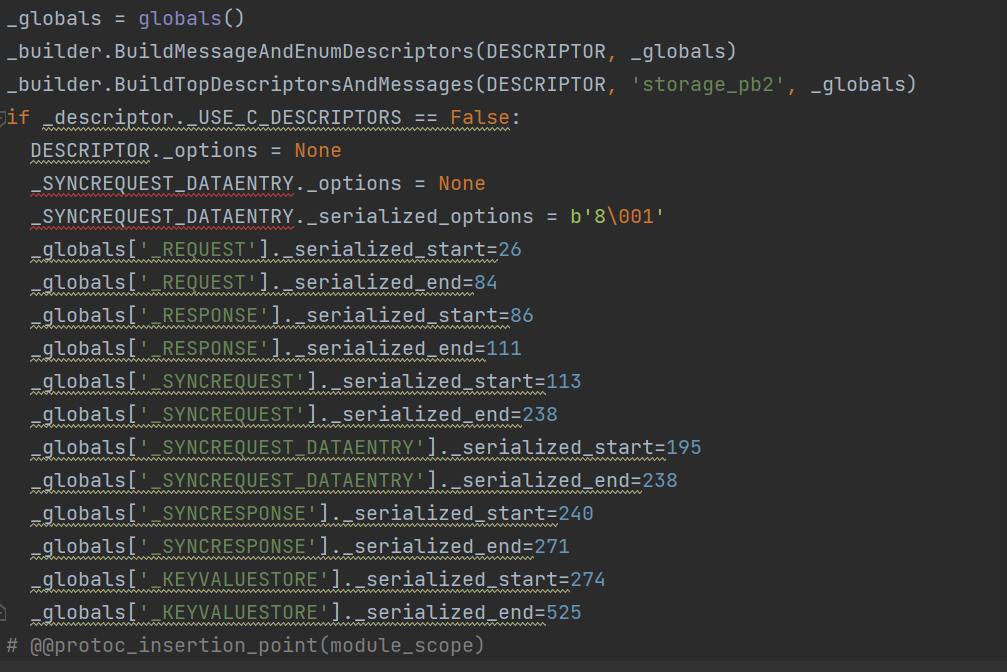
**通过使用**

**python -m grpc\_tools.protoc -I. --python\_out=. --grpc\_python\_out=. storage.proto**

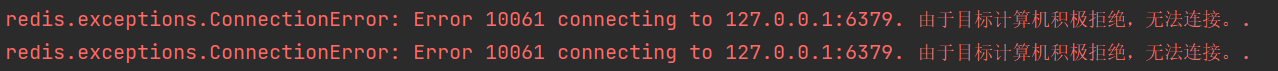
**可以生成storage\_pb2.py 和 storage\_pb2\_grpc.py 两个文件，分别包含 Protocol Buffers 消息和 gRPC 服务的 Python 代码。**

**部分代码如下：**





**分布式锁利用redis**



**报错的原因是因为Redis服务没有启动**

**所以需要下载**到[GitHub](https://so.csdn.net/so/search?q=GitHub&spm=1001.2101.3001.7020)下载能**下运行的Redis服务**：这里是使用windows版本的  
[Redis-x64-3.2.100.zip](https://github.com/microsoftarchive/redis/releases/download/win-3.2.100/Redis-x64-3.2.100.zip)

下载解压之后点击redis-server.exe运行后即可启动redis服务。

使用python redis库实现分布式锁，包括上锁和释放锁两个操作。具体实现如下：

#distributed\_lock.py  
import time  
import uuid  
from multiprocessing import Process  
import redis  
  
redis\_client = redis.Redis(host='127.0.0.1', port=6379)  
  
  
# 加锁的过程  
def acquire\_lock(lock\_name, args, acquite\_timeout=30, time\_out=120):  
 identifier = str(uuid.uuid4())  
 # 客户端获取锁的结束时间  
 end = time.time() + acquite\_timeout  
 lock\_names = "lock\_name:" + lock\_name  
 print(f"进程 {str(args)} end\_time:{end}")  
 while time.time() < end:  
 # setnx(key,value) 只有key不存在情况下，将key的值设置为value 返回True,若key存在则不做任何动作,返回False  
 if redis\_client.setnx(lock\_names, identifier):  
 # 设置键的过期时间，过期自动剔除，释放锁  
 print('获得锁:进程' + str(args))  
 # print(f'分布式锁value:{identifier}')  
 redis\_client.expire(lock\_name, time\_out)  
 return identifier  
 # 当锁未被设置过期时间时，重新设置其过期时间  
 elif redis\_client.ttl(lock\_name) == -1:  
 redis\_client.expire(lock\_name, time\_out)  
 time.sleep(0.001)  
 return False  
  
# 锁的释放  
def release\_lock(lock\_name, identifire):  
 lock\_names = "lock\_name:" + lock\_name  
 pipe = redis\_client.pipeline(True)  
 while True:  
 try:  
 # 通过watch命令监视某个键，当该键未被其他客户端修改值时，事务成功执行。当事务运行过程中，发现该值被其他客户端更新了值，任务失败  
 pipe.watch(lock\_names)  
 print(pipe.get((lock\_names)))  
 if pipe.get(lock\_names).decode() == identifire: # 检查客户端是否仍然持有该锁  
 # multi命令用于开启一个事务，它总是返回ok  
 # multi执行之后， 客户端可以继续向服务器发送任意多条命令， 这些命令不会立即被执行， 而是被放到一个队列中， 当 EXEC 命令被调用时， 所有队列中的命令才会被执行  
 pipe.multi()  
 # 删除键，释放锁  
 pipe.delete(lock\_names)  
 # execute命令负责触发并执行事务中的所有命令  
 pipe.execute()  
 return True  
 pipe.unwatch()  
 break  
 except redis.exceptions.WatchError:  
 # # 释放锁期间，有其他客户端改变了键值对，锁释放失败，进行循环  
 pass  
 return False

**关于节点服务器的功能实现：**

**主要实现KeyValueStoreServicer类，其中键值对数据以字典形式存储。包含对数据库中的键值对的增删查改操作，Put, Get, Delete, Set, SyncData: 分别对应 gRPC 定义的服务中的不同操作，实现了对数据的增、查、删、设置和同步等操作。并且包含简单的锁机制。其粒度比分布式锁更小，是主要针对单一节点中的锁。get\_lock可以获取键对应的锁，确保并发操作时的线程安全。值得注意的是 这里再每个操作执行修改数据库之后都会与其他节点进行同步操作。**

class KeyValueStoreServicer(storage\_pb2\_grpc.KeyValueStoreServicer):  
 def \_\_init\_\_(self):  
 self.data = {}  
 self.stub\_node2 = None  
 self.stub\_node3 = None  
 self.locks = {} # 用于存储键值对的锁  
  
 def Put(self, request, context):  
 with self.get\_lock(request.key):  
 self.data[request.key] = request.value  
 self.sync\_data()  
 return storage\_pb2.Response(value=f"Key {request.key} set successfully in Node 1")  
  
 def Get(self, request, context):  
 value = self.data.get(request.key, "")  
 return storage\_pb2.Response(value=value)  
  
 def Delete(self, request, context):  
 with self.get\_lock(request.key):  
 if request.key in self.data:  
 del self.data[request.key]  
 self.sync\_data()  
 return storage\_pb2.Response(value=f"Key {request.key} deleted successfully in Node 1")  
 else:  
 return storage\_pb2.Response(value=f"Key {request.key} not found in Node 1")  
  
 def Set(self, request, context):  
 with self.get\_lock(request.key):  
 self.data = {request.key: request.value}  
 self.sync\_data()  
 return storage\_pb2.Response(value=f"Data set successfully in Node 1")  
  
 def SyncData(self, request, context):  
 with self.get\_lock(request.source\_node):  
 self.data.update(request.data)  
 return storage\_pb2.SyncResponse(message="Data synchronized successfully in Node 1")  
  
 def sync\_data(self):  
 if self.stub\_node2:  
 sync\_request = storage\_pb2.SyncRequest(data=self.data, source\_node="Node1")  
 response = self.stub\_node2.SyncData(sync\_request)  
 print(response.message)  
 if self.stub\_node3:  
 sync\_request = storage\_pb2.SyncRequest(data=self.data, source\_node="Node1")  
 response = self.stub\_node3.SyncData(sync\_request)  
 print(response.message)  
  
 def get\_lock(self, key):  
 if key not in self.locks:  
 self.locks[key] = threading.Lock()  
 return self.locks[key]

**多个节点之间的服务过程是使用grpc来连接的。某个节点通过gRPC 通道连接到其他节点，获取其他节点的存根 (stub\_node1)，以便在需要时进行通信，确保多个分布的节点服务器数据一致性。**

**节点一的服务的过程：**

def serve():  
 server = grpc.server(futures.ThreadPoolExecutor(max\_workers=10))  
 kv\_servicer = KeyValueStoreServicer()  
 storage\_pb2\_grpc.add\_KeyValueStoreServicer\_to\_server(kv\_servicer, server)  
 server.add\_insecure\_port("[::]:50051")  
 server.start()  
  
 # Connect to Node 2  
 with grpc.insecure\_channel("localhost:50052") as channel2,grpc.insecure\_channel("localhost:50053") as channel3:  
 kv\_servicer.stub\_node2 = storage\_pb2\_grpc.KeyValueStoreStub(channel2)  
 kv\_servicer.stub\_node3 = storage\_pb2\_grpc.KeyValueStoreStub(channel3)  
 try:  
 server.wait\_for\_termination()  
 except KeyboardInterrupt:  
 pass

**与节点一类似的，节点二的服务过程：**

def serve():  
 server = grpc.server(futures.ThreadPoolExecutor(max\_workers=10))  
 kv\_servicer = KeyValueStoreServicer()  
 storage\_pb2\_grpc.add\_KeyValueStoreServicer\_to\_server(kv\_servicer, server)  
 server.add\_insecure\_port("[::]:50052")  
 server.start()  
  
 # Connect to Node 1  
 with grpc.insecure\_channel("localhost:50051") as channel1,grpc.insecure\_channel("localhost:50053") as channel3:  
 kv\_servicer.stub\_node1 = storage\_pb2\_grpc.KeyValueStoreStub(channel1)  
 kv\_servicer.stub\_node3 = storage\_pb2\_grpc.KeyValueStoreStub(channel3)  
 try:  
 server.wait\_for\_termination()  
 except KeyboardInterrupt:  
 pass

**节点三则在此省略了。**

**客户端最终的使用过程中不会手动指定通道和服务器节点，所以需要提供一个函数使调用节点以及将指令操作转换的过程对客户端透明。所以需要增加中间件来实现这一透明化的调用过程。中间件的作用是隐藏底层通信和节点选择的细节，为客户端提供一个简化的接口。在项目中提供了perform\_operation的封装操作。在其中包括了分布式锁的调用以及通过随机选择节点的简单负载均衡操作。**

count = 0 # 添加全局变量 count  
  
def perform\_operation(nodes, key, value, operation\_type):  
 global count  
 identifire = acquire\_lock('test', count)  
 count += 1  
  
 if identifire:  
 # 随机选择一个节点  
 stub = random.choice(nodes)  
  
 try:  
 if operation\_type == "Put":  
 response = stub.Put(storage\_pb2.Request(key=key, value=value, source\_node="Client"))  
 elif operation\_type == "Get":  
 response = stub.Get(storage\_pb2.Request(key=key))  
 elif operation\_type == "Delete":  
 response = stub.Delete(storage\_pb2.Request(key=key))  
 # 添加其他操作类型的判断  
  
 print(f"Response - {operation\_type}: {response.value}")  
 except grpc.RpcError as e:  
 # 如果节点不可用，选择另一个节点  
 print(f"Error executing {operation\_type}: {e}")  
 nodes.remove(stub)  
 if not nodes:  
 print("All nodes are unavailable.")  
 return  
 else:  
 print("Choosing another node...")  
 stub = random.choice(nodes)  
 print(f"Selected node: {stub}")  
  
 # 重新执行操作  
 if operation\_type == "Put":  
 response = stub.Put(storage\_pb2.Request(key=key, value=value, source\_node="Client"))  
 elif operation\_type == "Get":  
 response = stub.Get(storage\_pb2.Request(key=key))  
 elif operation\_type == "Delete":  
 response = stub.Delete(storage\_pb2.Request(key=key))  
 # 添加其他操作类型的判断  
  
 print(f"Response - {operation\_type}: {response.value}")  
  
 res = release\_lock('test', identifire)  
 print(f'Release status: {res}')  
 else:  
 print('Failed to acquire Redis distributed lock, another process is using it')

**其中的参数设置为（本次实验在本地测试三个服务器节点）：**

channel1 = grpc.insecure\_channel("localhost:50051")  
stub1 = storage\_pb2\_grpc.KeyValueStoreStub(channel1)  
channel2 = grpc.insecure\_channel("localhost:50052")  
stub2 = storage\_pb2\_grpc.KeyValueStoreStub(channel2)  
channel3 = grpc.insecure\_channel("localhost:50053")  
stub3 = storage\_pb2\_grpc.KeyValueStoreStub(channel3)  
nodes = [stub1, stub2,stub3]

**在以上基础上可以增加缓存来优化性能。关于缓存的设置层次不同，也会具有不同的效果。**

1. **客户端缓存： 客户端缓存是指缓存在应用程序的客户端一侧。每个客户端都维护其本地缓存，用于存储最近访问的数据。这有助于减少对服务器的请求，提高响应速度。但需要注意的是，由于每个客户端都有自己的缓存，缓存的一致性可能会成为挑战。**
2. **服务器端缓存： 服务端缓存是指缓存在服务端一侧。服务器维护一个共享的缓存，所有客户端可以共享。这样可以提高整体性能，减少对后端存储的访问。然而，需要解决缓存的一致性和更新问题，以确保缓存中的数据是最新的。**
3. **分布式缓存中间件： 为了更好地支持分布式环境，可以使用专门的分布式缓存中间件，如Redis、Memcached等。这些中间件提供了分布式、高性能、可扩展的缓存服务，可以被多个客户端和服务端共享。分布式缓存中间件通常能够提供更高级的功能，如缓存失效策略、数据分片和复制等。**

**这里选择在中间件中设置缓存，可以被多个用户和服务器共享。实现简单的缓存类包含缓存键值对和对应的状态。以及三种修改缓存数据以及状态的操作。同样的，需要注意操作的原子性，即避免缓存数据被修改，但是状态未修改的情况，所以还是增加了锁。当然由于python存在已经封装好的lru缓存机制，所以可以通过 lru\_cache 装饰器简化了缓存的实现即@lru\_cache(maxsize=500)。这不仅提高了代码的简洁性，还避免了手动实现缓存逻辑的复杂性。**

class Cache:  
 def \_\_init\_\_(self):  
 self.cache = {}  
 self.cache\_status = {}  
 self.lock = threading.Lock()  
  
  @lru\_cache(maxsize=500)  
 def get(self, key):  
 with self.lock:  
 if key in self.cache\_status and self.cache\_status[key] == "valid":  
 # 如果缓存状态为有效，直接返回缓存值  
 print(f"Cache - Get: {self.cache[key]} (from cache)")  
 return self.cache[key]  
 else:  
 # 缓存状态为无效或缓存中不存在，返回None  
 return None  
  
 def put(self, key, value):  
 with self.lock:  
 # 在缓存中更新键值对  
 self.cache[key] = value  
 # 更新缓存状态  
 self.cache\_status[key] = "valid"  
  
 def delete(self, key):  
 with self.lock:  
 # 删除缓存中的键值对  
 if key in self.cache:  
 del self.cache[key]  
 # 更新缓存状态为无效  
 self.cache\_status[key] = "invalid"

1. 总结

**在本次实验中也遇到了不少的问题和困难。万事开头难，比如在最开始并不知道整体的框架应该如何搭建，在参考老师提供的几个开源分布式键值存储系统的介绍以及大模型的建议之后，有了大致的思路。然后就是根据项目的要求，循序渐进，一点点地完成要求，从一个简单的仅有多节点的分布式存储系统，慢慢增加功能来完成多用、一致性等等的要求。但是受限于个人的能力，本次实现的项目性能并没有很优秀。**

**但是通过本次实验我对于分布式系统有了更进一步的认识，对于其中的rpc、一致性等知识不再局限于理论，对于分布式系统的用处更加的肯定了。**