

分布式的键值存储系统

日期	姓名	学号	班级
2021.1.26	陈琮昊	18340013	计科一班

一、题目要求：

1.题目：

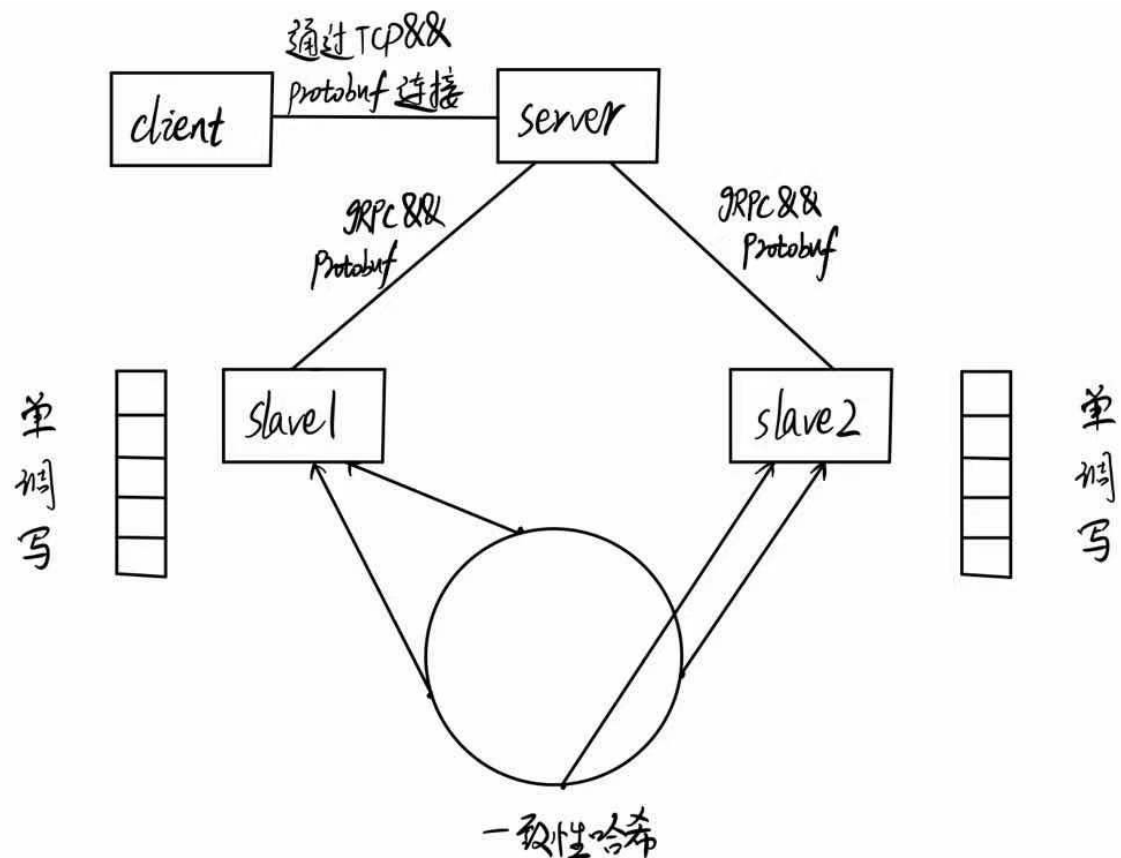
设计并实现一个分布式键值（key-value）存储系统，可以是基于磁盘的存储系统，也可以是基于内存的存储系统，可以是主从结构的集中式分布式系统，也可以是 P2P 式的非集中式分布式系统。能够完成基本的读、写、删除等功能，支持缓存、多用户和数据一致性保证。

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；

二、解决思路：

1.整体结构：



2.系统结构:

实现的系统是主从结构的集中式分布式系统，主节点处理客户端请求，并把相应的请求转发到从节点上，得到结果再发送回给客户端。同时，它也是基于内存的存储系统。

3.RPC:

RPC（远程过程调用），第三方的客户程序可以通过RPC运行一些定义在其他程序或计算机进程的黑盒函数，并获得函数返回的数据。换言之，RPC框架可以使用户更方便地在分布式系统中调用远程服务，被调用的服务在其他联网的计算机上执行，并返回结果。

这里使用的是Google开源的gRPC框架，利用ProtoBuf作为接口描述语言。

4.一致性哈希:

使用字符串哈希算法，将每个服务器映射为多个虚拟节点，然后维护一个有序数组，每次请求都使用二分查找在这个数组上查询大于等于请求哈希值的第一个服务器，将数据存储在服务器上。

5.一致性保证:

这里采用的是单调写一致性这种面向客户的一致性的方法，在实现过程中只保证PUT操作是按顺序执行的。具体的实现是，每个节点都维护一个写队列，每次PUT请求不马上执行，而是将请求放入队列中，节点从队列中按顺序取出写请求然后执行返回。

6.通信连接:

client到master使用socket的TCP通信，master维护一个线程池，把每次请求丢给一个线程处理，然后继续监听；master到slave使用gRPC通信。

7.数据迁移：

每次有新的 slave 加入时，需要计算出它覆盖了原本一致性哈希环上的哪些区间，将这些区间发送给对应的slave，把它们区间内的数据打包发送给新 slave 。当数据完成迁移的时候新 slave 才能执行其他操作。注意，数据迁移是一个原子的（atomic）操作，此时不能执行任何 PUT 操作，但是可以执行 GET 操作。

三、实现细节：

1.环境配置：

运行环境：Windows10 + Python3.6

需要安装相关包的命令如下：

```
pip install grpcio
pip install grpcio-tools googleapis-common-protos
pip install lmdb
```

其中 lmdb 是一种非常快的内存映射型数据库，同样他也属于key-value型数据库。

2.代码实现：

1.ProtoBuf接口：

有两个接口文件存放于protos文件夹内，分别为 rpc_package.proto 和 tcp_protobuf.proto。

在文件 rpc_package.proto 内定义了如下内容：

Request 结构体中包含2个成员变量key和value，类型都是字符串。Response 结构体中包含2个成员变量message和value，分别表示client请求操作的执行结果，以及操作的返回值，类型都是字符串。TransForm 结构体提供4个成员变量start，end，aim_ip和aim_port，start和end表示要传送的key的哈希值的范围为start~end，aim_ip和aim_port表示目标slave的ip和port。BackUp 有2个成员变量，keys和values，用来存储数据迁移时的key和value。

在文件 tcp_protobuf.proto 内，则是定义了 Request 和 Response 两个结构体，Request 结构体中包含3个成员变量operation、key和value，Response 结构体同上。

2.服务器端：

服务器的代码见 server.py，这里以类（class）为一个单位对其实现进行介绍：

DATA 类是server调用slave服务器的 gRPC 服务，实现 slave 上的数据库操作，里面实现的函数有put、get、delete、transform，这里面的实现比较简单，比如对于操作 PUT，处理成功则返回请求，失败则发送错误消息：

```
class DATA():
    def __init__(self, slave_ip, slave_port):
        self.ip = slave_ip
        self.port = slave_port
        self.channel = grpc.insecure_channel('{}:{}'.format(slave_ip,
slave_port))
        self.stub = src.DatastoreStub(self.channel)

    def __del__(self):
        self.channel.close()
```

```

def put(self, key, value):

def get(self, key):

def delete(self, key):

def transform(self, start, end, ip, port):

```

```

def put(self, key, value):
    try:
        response = self.stub.put(src.Request(key=key, value=value))
        return response
    except grpc._channel._Rendezvous:
        # 当消息处理失败时，在try的except处返回处理失败的消息给用户
        return src.Response(message='Failed!')

```

HashRing 这个类则是实现一致性哈希，初始化时，使用一个字典（dict）存储了每个 slave 节点最近一次的访问时间：dict 的 key 为（ip, port）的字符串对，value 为当前的时间；查询操作（GET）则是使用二分查找大于等于请求哈希值的第一个节点；添加节点时，需要注意的一点就是要把原来处理这个区间的节点的数据进行迁移，迁移至新的节点，即 transform 操作，可见 add_node 函数；还有就是节点的删除实现于 remove_node 函数内：

```

class HashRing():
    # 输入虚拟节点的数目（默认为0，即每个节点只在一致性哈希环上有一个节点）
    def __init__(self, virtual_number=0):

    # 字符串哈希
    def hash(self, slavename):

    # 根据key的哈希决定它的数据存在哪个slave中
    def get(self, key):

    # 根据slave的ip和port将它加入到一致性哈希环中
    def add_node(self, ip, port):
        # 如果是本来就存在的节点则不执行任何操作
        if (ip, port) in self.slave_last_access_time:
            return
        data = DATA(ip, port)
        tmp_hash = []
        tmp_data = []
        count = 0
        for i in range(self.node_number):
            slavename = '{}:{}'.format(ip, port, str(i))
            slavehash = self.hash(slavename)
            tmp_hash.append(slavehash)
            tmp_data.append(data)
            # 只有当前一致性哈希环上有节点的时候才需要数据迁移
            if self.circle_hash.shape[0] == 0:
                continue
            # 新节点加入时需要把原来处理这个区间的节点的数据进行迁移
            index = bisect.bisect_left(self.circle_hash, slavehash)
            if index >= self.circle_hash.shape[0]:
                index = 0
            last_index = index - 1
            if last_index < 0:

```

```

        last_index = self.circle_hash.shape[0] - 1
        # transform是目标节点的操作，将这个新节点的ip和port还有它覆盖的哈希范围发给目标
        # 节点，目标节点再调用新节点的receive服务将对应的key和value发送给它
        try:
            response = self.circle_data[index].transform(
                self.circle_hash[last_index] + 1, slavehash, ip, port)
        except:
            response = src.Response(message='Failed!')
        if response.message != 'Done!':
            # 如果迁移失败就不入网
            tmp_hash.pop()
            tmp_data.pop()
            count += 1
            print('Failed to add new slave {}:{} #{}'.format(ip, port, i))
        if count == self.node_number:
            return
        self.slave_last_access_time[(ip, port)] = time.time()
        # 将结果补充到环上
        self.circle_hash = np.append(self.circle_hash,
            tmp_hash).astype(np.int64)
        self.circle_data = np.append(self.circle_data, tmp_data)
        # 保证这个哈希环是有序的，保证能够使用二分查找
        sorted_index = self.circle_hash.argsort()
        self.circle_hash = self.circle_hash[sorted_index]
        self.circle_data = self.circle_data[sorted_index]

        # 根据slave的ip和port将它移除一致性哈希环
        def remove_node(self, ip, port):
            remove_indices = []
            for i in range(self.node_number):
                slavename = '{}:{}'.format(ip, port, str(i))
                slavehash = self.hash(slavename)
                remove_index = bisect.bisect_left(self.circle_hash, slavehash)
                # 防止一些不必要的bug（比如刚好有2个slave的哈希值相同）
                while remove_index < self.circle_hash.shape[0] and
                    self.circle_hash[remove_index] == slavehash:
                    if self.circle_data[remove_index].ip != ip or self.circle_data[
                        remove_index].port != port:
                        remove_index += 1
                    else:
                        break
                # 如果出bug，这个节点本来就不在哈希环上，那就什么都不操作
                if remove_index >= self.circle_hash.shape[0] or
                    self.circle_hash[remove_index] != slavehash:
                    continue
                remove_indices.append(remove_index)

            # 节点删除
            if (ip, port) in self.slave_last_access_time:
                self.slave_last_access_time.pop((ip, port))
            self.circle_hash = np.delete(self.circle_hash, remove_indices)
            self.circle_data = np.delete(self.circle_data, remove_indices)

```

RPCServer 类，该类则是服务器提供建立slave连接的服务，用于识别新加入的slave，以及更新访问时间：

```

# server提供建立slave连接的服务，用于识别新加入的slave
class RPCServer(src.SlaveServicer):
    def __init__(self, hashing):
        self.hashring = hashing

    # 如果这个slave没出现过，加入一致性哈希环；更新它的访问时间
    def add_slave_setting(self, request, context):
        print('Slave({}, {}), welcome!'.format(request.ip, request.port))
        if not (request.ip, request.port) in self.hashring.slave_last_access_time:
            self.hashring.add_node(request.ip, request.port)
        else:
            self.hashring.slave_last_access_time[(request.ip, request.port)] = time.time()
        return src.SlaveCode(message='Success!')

```

还有一个类 TCPServer，通过建立TCP连接来处理客户端的请求：

```

class TCPServer():
    # server_tcp_port负责监听client通信，server_rpc_port负责监听是否有新的slave加入
    def __init__(self, server_ip='localhost', server_tcp_port=8000, server_rpc_port=8001):
        self.server_ip = server_ip
        self.server_tcp_port = server_tcp_port
        self.server_rpc_port = server_rpc_port
        # 建立slave节点数据分发的一致性哈希环
        self.hashring = HashRing(virtual_number=3)
        # 建立新slave服务器入分布式系统的监听服务
        self.rpcserver = grpc.server(ThreadPoolExecutor(max_workers=1))
        src.add_slave_servicer_to_server(RPCServer(self.hashring), self.rpcserver)
        self.rpcserver.add_insecure_port('{}:{}'.format(server_ip, server_rpc_port))
        self.rpcserver.start()
        print('Slave rpc service start!')
        # 建立处理TCP连接的线程池
        self.executor = ThreadPoolExecutor(max_workers=30)
        # 建立监听client请求的TCP服务
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.bind((self.server_ip, self.server_tcp_port))

    def __del__(self):
        self.rpcserver.stop(0)
        self.sock.close()

    def _listen(self):
        print('Start listening!')
        # 无限循环，监听客户端的TCP请求
        while True:
            con, addr = self.sock.accept()
            t = self.executor.submit(_processing_request, self.hashring, con, addr)

    def listen(self):
        print('waiting for connection...')
        self.sock.listen(20)

```

```

# 等待连接，并且把连接信息丢给线程处理
t = threading.Thread(target=self._listen)
t.setDaemon(True)
t.start()
try:
    while True:
        time.sleep(60 * 60 * 24)
except KeyboardInterrupt:
    print('Close server!')

def start():
    pass

```

3.客户端：

客户端代码见 client.py，该代码实现比较简单。首先就是发送消息和接收消息；然后对于每个操作（PUT、GET、DEL）都单独建立一次TCP连接；然后就是运行客户端，在运行客户端时会提示你输入要执行的操作，以及对应的key、value。客户端运行的主要代码如下：

```

def client_start(args):
    print('Please press Ctrl+C to exit the client.')
    client = Client(args.server_ip, args.server_tcp_port)
    try:
        while True:
            # 输入一个操作
            operation = input('input operation: ')
            if operation == 'put':
                key = input('input key: ')
                value = input('input value: ')
                client.put(key, value)
            elif operation == 'get':
                key = input('input key: ')
                client.get(key)
            elif operation == 'delete':
                key = input('input key: ')
                client.delete(key)
            else:
                print('operation error!')
    except KeyboardInterrupt:
        print('Close client!')

```

4.从节点 (slave)：

从节点有关的代码实现在 slave.py 内，这里的核心部分就是实现了单调写一致性：

```

# 用队列+condition函数的方式实现单调写
def dandiaowrite(db_name, queue):
    while True:
        # queue.get在队列是空的时候会自动阻塞当前线程
        cond, operation, key, value, response = queue.get()
        with cond:
            if operation == 'put':
                with lmdb.open(db_name) as env:
                    beginning = env.begin(write=True)
                    result = beginning.put(key.encode(ENCODING),
value.encode(ENCODING))

```

```

        beginning.commit()
    if result:
        response.message = 'Done!'
    else:
        response.message = 'Failed!'
elif operation == 'delete':
    with lmdb.open(db_name) as env:
        beginning = env.begin(write=True)
        result = beginning.delete(key.encode(ENCODING))
        beginning.commit()
    if result:
        response.message = 'Done!'
    else:
        response.message = 'Failed!'
elif operation == 'recieve':
    result = 0
    if len(key) == 0:
        response.message = 'Done!'
    else:
        with lmdb.open(db_name) as env:
            beginning = env.begin(write=True)
            for k, v in zip(key, value):
                result += beginning.put(k.encode(ENCODING),
v.encode(ENCODING))
            beginning.commit()
        if result > 0:
            response.message = 'Done!'
        else:
            response.message = 'Failed!'
# 通知对应的线程，服务已经commit了
cond.notify()

```

对于 GET 操作不需要考虑太多，因为该操作只是读取数据；但对于 PUT、DEL 操作则要保证单调写一致性，这里使用了线程的 condition() 函数，它可以保证阻塞当前任务，直到单调写队列把这个写任务完成。比如对于 DEL 操作，这里给出相应代码如下：

```

def delete(self, request, context):
    print('Delete request from {}: key({}),
value({})'.format(context.peer(), request.key, request.value))
    # 使用Condition阻塞当前任务，直到单调写把这个写任务完成
    c = Condition()
    response = Response(message='Failed!')
    with c:
        self.queue.put((c, 'delete', request.key, request.value, response))
        result = c.wait(WAITTIME)
    if not result:
        response.message = 'Failed!'
    return response

```

然后就是从节点的运行函数slave_start:

```

def slave_start(args):
    # 数据库名
    db_name = 'db_{}_{}'.format(args.ip, args.port)
    # 单调写队列
    write_queue = Queue()

```



```

# 监听来自server关于服务器操作的rpc服务
server = grpc.server(ThreadPoolExecutor(max_workers=10))
add_DatastoreServicer_to_server(Slave(db_name, write_queue), server)
server.add_insecure_port('{}:{}'.format(args.ip, args.port))
server.start()
# 单独建立一个线程处理单调写队列里面的任务
t = Thread(target=dandiaowrite, args=(db_name, write_queue))
t.setDaemon(True)
t.start()
print('Datastore service start!')
# slave建立后, 调用dictionary_server的grpc服务器, 进行这个slave服务器识别
print('Slave start!')
# 每隔一段时间向server发一个信号表示存在感
try:
    while True:
        try:
            channel = grpc.insecure_channel('{}:{}'.format(args.server_ip,
args.server_rpc_port))
            stub = SlaveStub(channel)
            errcode = stub.add_slave_setting(SlaveInformation(ip=args.ip,
port=args.port))
            channel.close()
        except grpc._channel._Rendezvous:
            print('Link to dictionary server failed, try again.')
            time.sleep(HEARTBEAT)
    except KeyboardInterrupt:
        server.stop(0)
        print('Close slave!')

```

3.代码运行:





首先用 cd 命令切换至 proto 文件夹内, 输入如下命令来生成 gRPC 服务的 Python 代码:

```

python -m grpc_tools.protoc -I./ --python_out=. --grpc_python_out=.
rpc_package.proto
python -m grpc_tools.protoc -I./ --python_out=. --grpc_python_out=.
tcp_protobuf.proto

```

生成后可以看到有以下几个文件, 将这些文件移动至 src 文件夹内:

 rpc_package_pb2.py	2021/1/26 11:49	JetBrains PyCharm	14 KB
 rpc_package_pb2_grpc.py	2021/1/26 11:50	JetBrains PyCharm	11 KB
 tcp_protobuf_pb2.py	2021/1/26 11:49	JetBrains PyCharm	5 KB
 tcp_protobuf_pb2_grpc.py	2021/1/26 11:49	JetBrains PyCharm	1 KB

与此同时, 要更改一下 rpc_package_pb2_grpc.py 里文件的内容, 将原来的:

```
import rpc_package_pb2 as rpc__package__pb2
```

改为:

```
import src.rpc_package_pb2 as rpc__package__pb2
```

然后打开终端, 通过如下命令来运行代码:

```
python server.py
python slave.py --port 8002
python slave.py --port 8003
python client.py
python client.py
```

四、实验结果：

注：默认情况下，所有节点均未添加，只有执行 PUT 操作时才会添加。

按照上面的命令建立连接后进行如下功能测试：

在开启两个slave后，可以发现其在两个slave间进行了transform操作：

```
Transform request from ipv6:::1:54406: start(828594474), end(2187133089), aim_ip(localhost), aim_port(8003)
```

首先在 client.py（客户端）下执行 PUT 操作：

```
input operation: put
input key: 12
input value: 7
message: "Done!"
```

可以看到slave的确收到了该请求：

```
Put request from ipv6:::1:60936: key(12), value(7)
```

然后执行 GET 操作，看是否能查到key=12对应的值：

```
input operation: get
input key: 12
message: "Done!"
value: "7"
```

可以看到成功查询到value=7。与此同时，slave也会显示接收到了该请求：

```
Get request from ipv6:::1:60936: key(12), value()
```

再打开一个客户端，同样对key=12的节点进行 PUT 操作：

```
input operation: put
input key: 12
input value: 10
message: "Done!"
```

然后进行 GET 操作查询key=12对应的值此时为多少：

```
input operation: get
input key: 12
message: "Done!"
value: "10"
```

可以看到此时value=10，符合单调写一致性。

最后进行 DEL 操作，在客户端键入delete：

```
input operation: delete
input key: 12
message: "Done!"
```

slave接受了该请求：

```
Delete request from ipv6:::1:60936: key(12), value()
```

然后再进行 GET 查询key=12的操作，发现此时查询不到其value，证明删除成功：

```
input operation: get
input key: 12
message: "Failed!"
```

五、问题与总结：

本次实验实现了分布式的键值存储系统。主要就是要有一个整体上的构思。根据给出的题目要求，能够建立起一个系统结构，就像前面那个我画的图一样，画出来之后就比较清晰了，什么地方该用什么就一目了然了。通过这次实验还复习了计算机网络的相关知识，比如使用socket建立TCP连接（当然也忘得差不多了。。）。gRPC 由于之前作业有实现过所以相对比较简单、一致性哈希也并不是那么复杂。个人认为主要难度就是在于一致性问题上。当时写代码的时候感觉很乱，理了好久才理清思路。我还选择的是较为简单的单调写一致性。。至于本次实现，有一个问题就是本次实现里并没有数据备份，因此节点如果down了数据就可能丢失了。