Lab5-分布式键值对存储系统

实验目的

(1). 学习注册中心 Zookeeper，了解分布式系统开发。
(2). 学习使用 RPC 框架进行网络编程，实现并发编程、数据备份、负载均衡等机制。
(3). 增强分布式经典算法的掌握与理解。
(4). 学习分布式系统架构设计

实验要求

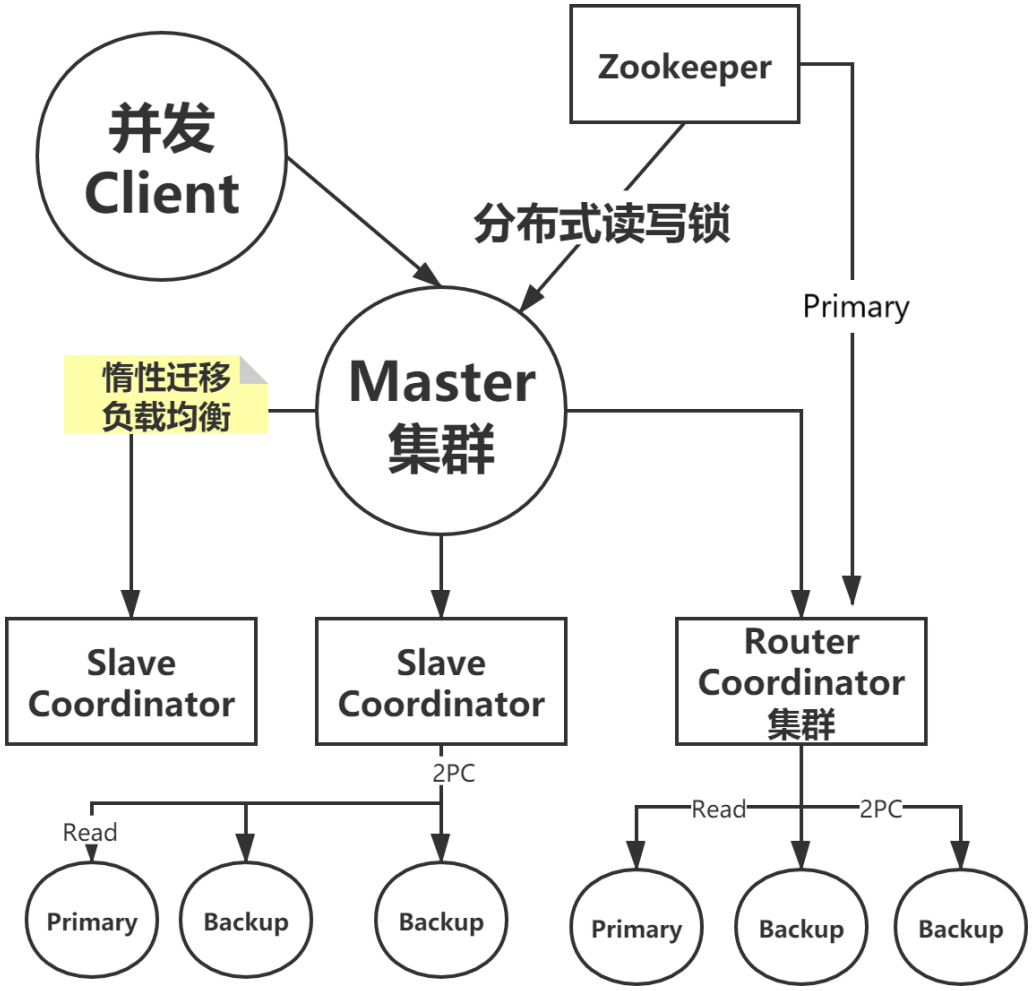
基于 Zookeeper 和 RPC 框架，开发分布式键值对存储系统。该系统应当具备 READ,PUT,DELETE 三个原语，具备高 Concurrency、Availability、Scalability，同时满足负载均衡。

- 系统需要至少具备一个存储元数据的 Master 节点、至少两个存储数据的 Data 节点
- 请求按照 key 转发给不同 Data 节点，数据互斥的存在不同节点上
- 系统中的所有数据应当具有备份，配置至少两个 Backup。
- 系统的数据节点增加时,负载与数据切分的重新平衡应当具备可拓展性。
- 系统的 Master 节点应当具有备份。
- 所有备份之间应当满足一致性
- 在动态部署 Backup 节点时，Backup 应当与 Primary 进行同步。
- 在 Backup 节点宕机时，Backup 节点应当决议一者自动成为 Primary 节点。
- 实现分布式锁保护事务的并发正确性

实验环境

(1). 操作系统版本: Windows 10:18363.900
(2). 编程语言版本: java:1.8.0_251
(3). 平台版本: apache-zookeeper-3.6.1，apache-dubbo-2.7.7
(4). Zookeeper 配置:节点以单机多进程测试, zookeeper 解压后设置 zoo.cfg 使用 cmd 运行，监听本地 127.0.0.1:2181 端口,设置 dataDir 为本地文件夹，maxClientCnxns 设置为 600，以提高节点承载额。

实验架构



本项目由十个模块组成，均为独立 maven 项目

Client 模块 - 模拟客户端进行 API 操作

Master 模块(Stateless) - 转发 Client 请求给 Slave

Slave 模块(Stateless) - 通过 **2PC** 控制 Data 一致性

Data 模块(Replicas) – 提供键值存储的数据

- 基于 **COW** 支持事务操作，并在部署新增 Backup 时主动数据同步

Router Coordinator 模块(Stateless) - 通过 **2PC** 控制 Router Data 一致性

Router Data 模块(Replicas) – 提供键值存储的元数据

- 基于 **COW** 支持事务操作，并在部署新增 Backup 时主动数据同步

API 模块 - 定义 RPC 接口，使节点支持 RPC

Load Balance 模块 – 提供被调用者集群的调用策略

- 基于**一致性 hash** 和**惰性数据迁移**实现的的负载均衡策略（Master->Slave）
- 读 Primray，写所有分组 Data 节点的负载均衡策略(Slave -> Data)
- 指定地址，定向访问的负载均衡策略(Data -> Slave)
- 读 Primary，写所有 RouterData 的负载均衡策略(Router Coordinator->Router Data)
- 默认的随机负载均衡策略(Client->Master + Master->Router Coordinator)

Lock 模块 - 基于临时顺序节点的**分布式读写锁**

zkClient 模块 - 基于 curator zookeeper 客户端基础拓展需求的 API

实验设计

1. 数据存储

Master 节点集群

维护数据切分，根据路由表与一致性哈希进行负载均衡将请求转发给 Slave 节点。

Slave 节点集群

维护数据一致性，充当 2PC 协议中的 Coordinator 角色。

读 Primary，2PC 写所有 Data 节点。具备组别。

Data 节点集群

存储数据，充当 2PC 协议中的 Participant 角色。具备备份

事务遵循 All-or-Nothing 原则，基于 COW 实现。具备组别。

Primary/Backup 间保持数据一致性，Backup 节点可动态同步 Primary。

RouterCoordinator 节点集群

维护路由表元数据一致性，充当 2PC 协议中的 Coordinator 角色。

读 Primary，2PC 写所有 Data 节点。

RouterData 节点集群

存储路由表元数据，充当 2PC 协议中的 Participant 角色。

事务遵循 All-or-Nothing 原则，基于 COW 实现。

Primary/Backup 间保持数据一致性，Backup 节点可动态同步 Primary。

2.RPC 通信

本实验基于 Apache 的 Dubbo RPC 框架开发，RPC 的配置通过注解完成，通过独立的 API 模块规定 RPC 协议，调用者和被调用者本身无模块依赖。

运行时，被调用者在 zookeeper 中注册，调用者监听所有可用服务，远程进行依赖注入 RPC 的 Stub。调用时通过负载均衡策略决定被调用者。

3.数据切分与可拓展性：负载均衡与数据均衡

Concurrent Hash

在这个问题中,保持节点变化前后数据映射的一致性十分重要,相较于普通的负载均衡,这里切分的不仅是请求,更是数据。如果映射变化剧烈，会导致大量的数据迁移。

虚拟节点

我使用了标准的一致性哈希算法，为每个实际节点创建十个虚拟节点。在环形 hash 表中，存放的虚拟节点是实际地址增加后缀生成的。在查询虚拟节点后，通过截取后缀即可得知实际地址。虚拟节点保证了负载的均匀性。

Hash 算法

Key 和虚拟节点均通过 FNV1_32_HASH 算法映射至环上，在查询时，从 key 的节点开始向后进行查找，后续的第一个服务器作为选择的服务器。

惰性迁移：Transfer on Write Load Balance

一致性 Hash 注重负载绝对均衡，但是无法处理数据不均衡的问题。新加入的节点能够均分负载，但是本身却没有数据，无法应对新增的负载。

常见的做法是**新加入节点时整个系统停机**，所有 Data 节点共同参与数据迁移，重新划分数据。然而，这样不仅带来了巨大的开销，同时也大大降低了系统的 availability。随着 Data 数目的增加，停机的时间也会增加，进而不具备 scalability。

TOW 是我自己思考并实现的负载均衡策略，适用于数据切分场景。算法思路来自于操作系统中的 COW 机制，任何写操作必须在副本上先进行修改然后再提交。

TOW 的核心：如果数据被读取，我们依然能够在原节点找到它；如果数据被写入，我们才需要让请求转发给正确的节点。即是说，数据直到需要被迁移的时候才需要迁移。

我在系统中存放路由表元数据，用于记录历史的映射结果，同时也能充当 GET 时的 cache 而免于计算一致性 hash。通过 key 计算一致性 hash，得到数据新位置。如果计算结果和历史不符，说明发生了节点的变化。此时需要通过 TOW 进行调整

READ 操作

如果路由表中不存在对应的项，说明数据并不在 Data 中不需要迁移，直接使用计算结果并且存储于路由表中。

如果路由表中存在对应的项，说明数据在项对应的 Data 节点，使用历史结果。

WRITE 操作(PUT 和 DELETE)

如果路由表中不存在对应的项，说明数据并不在 Data 中不需要迁移，直接使用计算结果并且存储于路由表中。

如果路由表中存在对应的项并且和计算结果一致，说明数据已经正确划分。

如果路由表中存在对应的项并且与计算结果不同，直接使用计算结果并且存储于路由表中，相当于直接废弃旧数据写入新数据。

非法操作

从所有服务提供者中随机选择一者进行转发

性能分析

避免刚刚迁移即被修改的多余开销

如果旧数据在迁移后没有被读取就被修改，那么这次迁移实际上没有起到任何效

果，迁移的是无效数据。采用 TOW，我们并不需要迁移旧数据，而是直接写入新节点，旧数据在之后因为路由表导向新节点，因此自动废弃。

均摊开销

TOW 事实上只需要每次负载均衡时维护路由表的微小投入，而不需要停机维护数据，因此即使新节点加入，依然能够保持高可用性。节点数目上升时，因为路由表使用 Hash 表进行查找，依然具备高拓展性。

惰性均衡

在节点变化之后时，因为路由表中依然存储历史的路径，GET 负载仍然导向原先节点。随着写的数据增多，负载均衡和数据划分趋向于平均，最终渐进逼近均衡。当所有原先数据均被写之后，达到理想的负载均衡。

缓存

READ 请求在路由表中存在路由时可以直接获取，不需要进行一致性 hash 计算。

4.分布式锁：并发数据访问

业务场景分析

KV 存储系统是典型的读写者模型，READ 为读者，WRITE 和 DELETE 为写者，读常常远多于写，因此使用分布式读写锁能保证并发正确的同时保证性能。

此外，存储数据结构使用并发哈希表，确保了数据本身的并发安全。

算法

读者被写者阻塞，必须等待写者离开临界区。
写者被读者和写者阻塞，必须等待所有读写者均离开临界区。

实现

利用 zookeeper 的 sequential 节点构建隐式队列。读者和写者向锁对应的目录下写入代表读和写的临时节点，利用创建的临时节点递增性保证时序。

从目录下获取所有临时子节点,排序后可视为 FIFO 队列。
读者需要等待拿锁时最后的写锁释放，因此找到读者节点前最后的写者节点，监听其销毁事件，一旦销毁即被唤醒，视为拿锁。释放锁时销毁临时节点即可。
写者需要等待拿锁时最后的任意锁释放，因此直接找到写者节点前一节点，监听其销毁事件，一旦销毁即被唤醒，视为拿锁。释放锁时销毁临时节点即可。
当之前没有满足条件的其他要锁者时，可以直接拿锁不进入 await。
对于 Master 的操作，都需要在 key 级别的力度上加读写锁。

5. 数据备份：高可用性 + 可伸缩性

为了保证数据一致性，选用 2PC，引入额外的 Coordinator 节点。

在 property 配置文件中写入 group.id，相同 group.id 的 Slave 节点和 Data 节点可视作一个整体，提供对应数据的访问并通过 2PC 保证一致性。Router Coordinator 和 Router Data 同理实现，区别在于不需要进行分组。

2PC 协议

所有写事务的提交均可分为两个阶段:

- Can commit: 准备阶段，要求各个参与者执行事务操作，并根据执行结果告知是否已经准备好 COMMIT。
- Commit: 提交阶段，Coordinator 收集各个参与者的结果，如果全部准备好了，广播通知各个参与者 COMMIT；否则全体 ABORT 回退到事务执行前。

COW

对任意对象的写操作，都需要先创建对象的副本在其上进行操作，如果成功，则直接用副本替换对象，否则丢弃副本。通过 COW，可以有效地实现 All or nothing。

Primary/Backup Load Balance

根据框架提供的抽象类自定义负载均衡规则，确切说负载不均衡，事实上是转发策略

对于 READ/SYNC 指令，转发给 Primary 节点(初始由服务列表的第一个担任)。

对于 PUT/DELETE/COMMIT 指令，转发给所有节点。

Primary/Backup Synchronization

Backup Data 节点部署后应当主动向 Primary Data 节点发送 SYNC 指令进行同步，由于 Primary 由 Slave 负责发现，因此 Backup 向 Slave 发出 SYNC，由 Slave 转发给对应的 Primary 获取全部数据，并且将本地数据同步为 Primary 数据。

实现

2PC

需要进行加锁，防止等待准备过程中插入其他事务从而导致副本错误，保证原子性。写操作转发给所有数据节点，等待全部返回后发出 COMMIT 请求。

COW

Data 节点在执行写方法时先对表创建副本，执行操作，并将副本存在本地。

- 如果收到 COMMIT 请求，那么将副本赋值给内存中的表，达成 COMMIT。
- 否则副本将在下次写操作时被覆盖，写操作丢失，达成 ABORT。

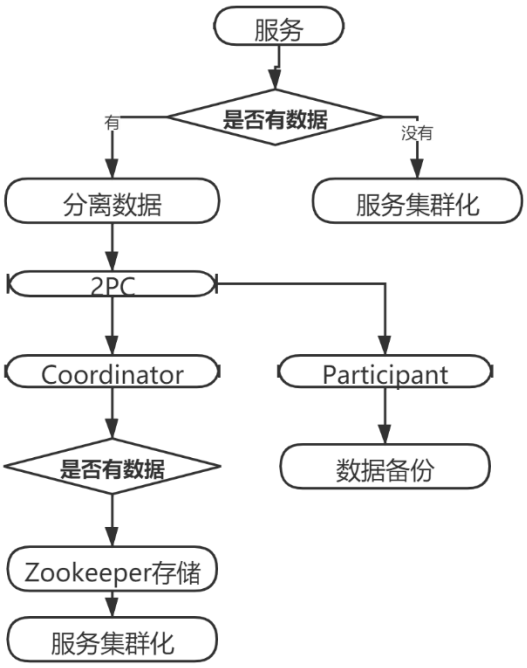
Primary/Backup Load Balance

所有对 Data 的操作通过 Load Balance 完成，读操作和写操作基于不同逻辑进行转发，Primary 必须等待 Backup 调用完成后才能进行事务，读操作直接转发给 Primary。Primary 地址的元数据存储在 zookeeper 中

Primary/Backup Synchronization

在配置文件中设置 hotfix 属性，当 hotfix 为 1 时，Backup 在写操作时先向 Slave 请求同步。Slave 由 Primary 获得数据后转发给 Backup，Backup 同步数据。Slave 服务主动在 RPC 上下文中提供自身地址以便 Backup 定向访问自身组别的 Slave。

6. 数据与服务分离：集群化避免单点故障



初始时，可以认为存在 Master-Slave 这两种服务，并且 Master 和 Slave 分别保存元数据和数据。数据和服务的耦合带来了高度的单点故障隐患，应当均可伸缩

在第一次分离时，数据和提供数据的 Slave 服务分离，Slave 服务变为 2PC 架构，coordinator 是无状态的服务，participant 是保持一致性的数据备份。此时数据和服务均可伸缩。

在第二次分离时，元数据和数据切分的 Master 服务分离，Master 服务无状态，从 Router 获取元数据。此时 Master 服务可伸缩。

在第三次分离时，元数据与提供元数据的 Router 服务分离，Router 服务变为 2PC 架构，coordinator 是无状态的服务，participant 是保持一致性的元数据备份。此时数据和服务均可伸缩。

实验结果

节点部署

初始化

- 在配置文件中填写组别 group.id = 0/1,分组运行 Data Node
- 在配置文件中填写组别 group.id = 0/1,分组运行 Slave Node
- 运行多个 Router Data + 多个 Router Coordinator + 多个 Master Node
- 运行 Client Node，模拟连续写入和读取，观察各服务的日志，确定负载

Slave 伸缩性

- 在配置文件中填写组别 group.id = 2,运行 Data Node
- 在配置文件中填写组别 group.id = 2,分组运行 Slave Node
- 运行 Client Node，模拟连续写入和读取，观察各服务的日志，确定负载变化

服务伸缩性

- 仅保留一组 Router Data + Router Coordinator + Master Node
- 运行 Client Node，工作正常

动态部署 Backup（可伸性）

- 在配置文件中填写组别 group.id = 2, group.hotfix =1 运行 Data Node
- 运行 ClientNode，进行任意写操作同步

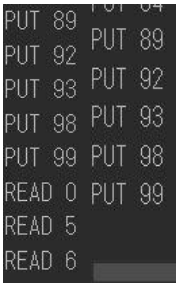
动态销毁 Primary（可伸性）

- 停止 group2 的 Primary 节点
- 运行 Client Node，模拟连续读取，观察数据是否同步以及负载变化

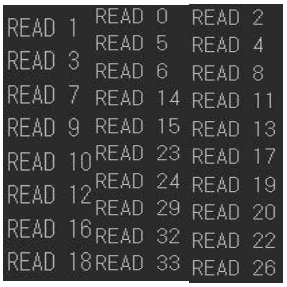
样例执行结果

结果表明分布式键值对存储系统 workload 下正确处理了数据切分，同时，我们的所有节点均能够实现可伸缩，允许动态增加备份或销毁。

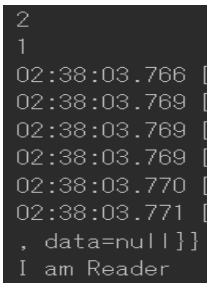
Primary/Backup



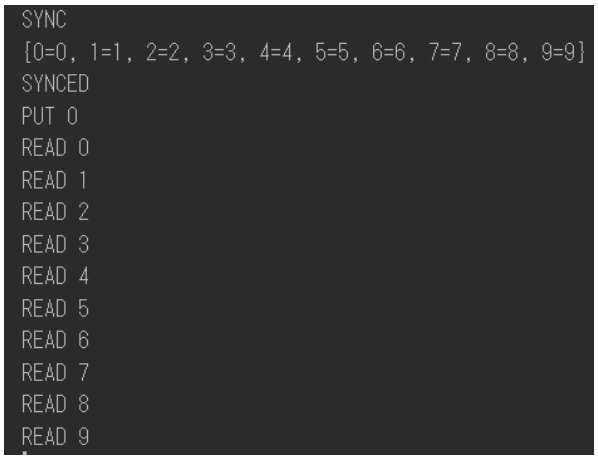
负载均衡



读者等待写者倒计时结束拿锁



新增 Backup 节点，与 Primary 进行数据同步



演示流程

知乎视频
https://www.zhihu.com/zvideo/1261856862208352256
操作流程即上文节点部署流程