SOFAJRaft 线性一致读实现剖析 | SOFAJRaft 实现原理

mp.weixin.qq.com/s

> **SOFA**Stack
>
> **S**calable **O**pen **F**inancial **A**rchitecture Stack 是蚂蚁金服自主研发的金融级分布式架构，包含了构建金融级云原生架构所需的各个组件，是在金融场景里锤炼出来的最佳实践。

本文为《剖析 | SOFAJRaft 实现原理》第三篇，本篇作者米麒麟，来自陆金所。《剖析 | SOFAJRaft 实现原理》系列由 SOFA 团队和源码爱好者们出品，项目代号：<SOFA:JRaftLab/>，目前领取已经完成，感谢大家的参与。

SOFAJRaft **是一个基于** Raft **一致性算法的生产级高性能** Java **实现，支持** MULTI-RAFT-GROUP**，适用于高负载低延迟的场景。**

SOFAJRaft ：

https://github.com/sofastack/sofa-jraft

前言

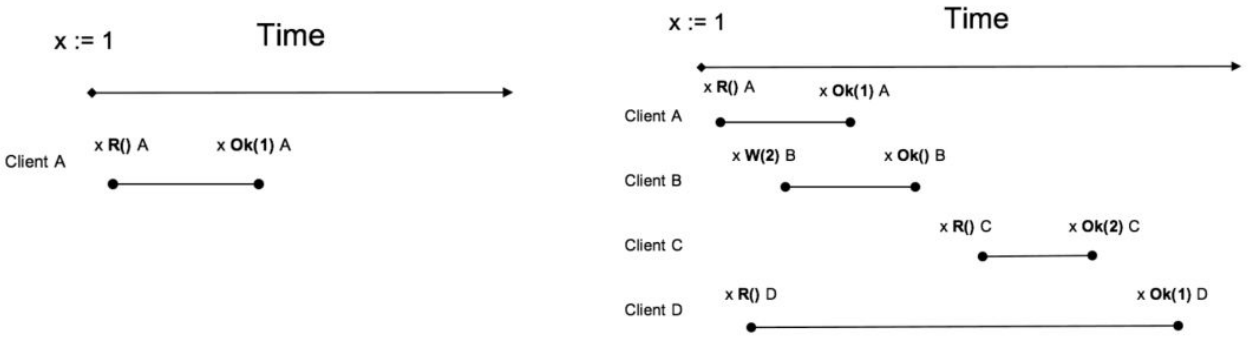
线性一致读是在分布式系统中实现 Java volatile **语义，当客户端向集群发起写操作的请求并且获得成功响应之后，该写操作的结果要对所有后来的读请求可见。实现线性一致读常规手段是走** Raft **协议，将读请求同样按照** Log **处理，通过日志复制和状态机执行获取读结果返回给客户端，**SOFAJRaft **采用** ReadIndex **替代走** Raft **状态机的方案。**

本文将围绕 Raft Log Read**，**ReadIndex Read **以及** Lease Read **等方面剖析线性一致读原理，阐述** SOFAJRaft **如何使用** ReadIndex **和** Lease Read **实现线性一致读：**

- 什么是线性一致读？共识算法只能保证多个节点对某个对象的状态是一致的，以 Raft 为例只能保证不同节点对 Raft Log 达成一致，那么 Log 后面的状态机的一致性呢？
- 基于 ReadIndex 和 Lease Read 方式 SOFAJRaft 如何实现高效的线性一致读？

线性一致读

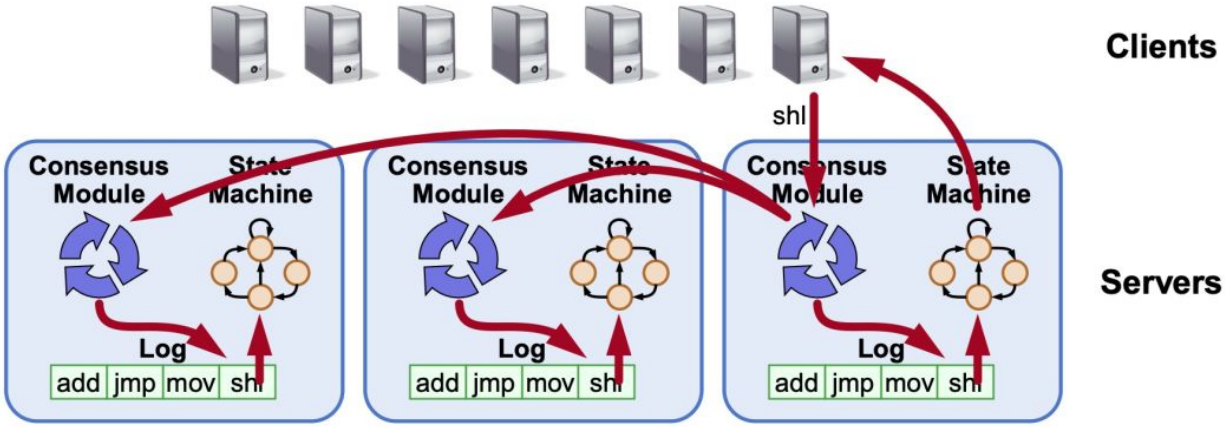
什么是线性一致读? 所谓线性一致读，一个简单的例子是在 t1 的时刻我们写入了一个值，那么在 t1 之后，我们一定能读到这个值，不可能读到 t1 之前的旧值(想想 Java 中的 volatile 关键字，即线性一致读就是在分布式系统中实现 Java volatile 语义)。简而言之是需要在分布式环境中实现 Java volatile 语义效果，即当 Client 向集群发起写操作的请求并且获得成功响应之后，该写操作的结果要对所有后来的读请求可见。和 volatile 的区别在于 volatile 是实现线程之间的可见，而 SOFAJRaft 需要实现 Server 之间的可见。



如上图 Client A、B、C、D 均符合线性一致读，其中 D 看起来是 Stale Read，其实并不是，D 请求横跨 3 个阶段，而 Read 可能发生在任意时刻，所以读到 1 或 2 都行。

Raft Log read

实现线性一致读最常规的办法是走 Raft 协议，将读请求同样按照 Log 处理，通过 Log 复制和状态机执行来获取读结果，然后再把读取的结果返回给 Client。因为 Raft 本来就是一个为了实现分布式环境下线性一致性的算法，所以通过 Raft 非常方便的实现线性 Read，也就是将任何的读请求走一次 Raft Log，等此 Log 提交之后在 apply 的时候从状态机里面读取值，一定能够保证这个读取到的值是满足线性要求的。



当然，因为每次 Read 都需要走 Raft 流程，Raft Log 存储、复制带来刷盘开销、存储开销、网络开销，走 Raft Log不仅仅有日志落盘的开销，还有日志复制的网络开销，另外还有一堆的 Raft “读日志” 造成的磁盘占用开销，导致 Read 操作性能是非常低效的，所以在读操作很多的场景下对性能影响很大，在读比重很大的系统中是无法被接受的，通常都不会使用。

在 Raft 里面，节点有三个状态：Leader，Candidate 和 Follower，任何 Raft 的写入操作都必须经过 Leader，只有 Leader 将对应的 Raft Log 复制到 Majority 的节点上面认为此次写入是成功的。所以如果当前 Leader 能确定一定是 Leader，那么能够直接在此 Leader 上面读取数据，因为对于 Leader 来说，如果确认一个 Log 已经提交到大多数节点，在 t1 的时候 apply 写入到状态机，那么在 t1 后的 Read 就一定能读取到这个新写入的数据。

那么如何确认 Leader 在处理这次 Read 的时候一定是 Leader 呢？在 Raft 论文里面，提到两种方法：

- ReadIndex Read
- Lease Read

ReadIndex Read

第一种是 ReadIndex Read，当 Leader 需要处理 Read 请求时，Leader 与过半机器交换心跳信息确定自己仍然是 Leader 后可提供线性一致读：

1. Leader 将自己当前 Log 的 commitIndex 记录到一个 Local 变量 ReadIndex 里面；
2. 接着向 Followers 节点发起一轮 Heartbeat，如果半数以上节点返回对应的 Heartbeat Response，那么 Leader就能够确定现在自己仍然是 Leader；
3. Leader 等待自己的 StateMachine 状态机执行，至少应用到 ReadIndex 记录的 Log，直到 applyIndex 超过 ReadIndex，这样就能够安全提供 Linearizable Read，也不必管读的时刻是否 Leader 已飘走；
4. Leader 执行 Read 请求，将结果返回给 Client。

使用 ReadIndex Read 提供 Follower Read 的功能，很容易在 Followers 节点上面提供线性一致读，Follower 收到 Read 请求之后：

1. Follower 节点向 Leader 请求最新的 ReadIndex；
2. Leader 仍然走一遍之前的流程，执行上面前 3 步的过程(确定自己真的是 Leader)，并且返回 ReadIndex 给 Follower；
3. Follower 等待当前的状态机的 applyIndex 超过 ReadIndex；
4. Follower 执行 Read 请求，将结果返回给 Client。

不同于通过 Raft Log 的 Read，ReadIndex Read 使用 Heartbeat 方式来让 Leader 确认自己是 Leader，省去 Raft Log 流程。相比较于走 Raft Log 方式，ReadIndex Read 省去磁盘的开销，能够大幅度提升吞吐量。虽然仍然会有网络开销，但是 Heartbeat 本来就很小，所以性能还是非常好的。

Lease Read

虽然 ReadIndex Read 比原来的 Raft Log Read 快很多，但毕竟还是存在 Heartbeat 网络开销，所以考虑做更进一步的优化。Raft 论文里面提及一种通过 Clock + Heartbeat 的 Lease Read 优化方法，也就是 Leader 发送 Heartbeat 的时候首先记录一个时间点 Start，当系统大部分节点都回复 Heartbeat Response，由于 Raft 的选举机制，Follower 会在 Election Timeout 的时间之后才重新发生选举，下一个 Leader 选举出来的时间保证大于 Start+Election Timeout/Clock Drift Bound，所以可以认为 Leader 的 Lease 有效期可以到 Start+Election Timeout/Clock Drift Bound 时间点。Lease Read 与 ReadIndex 类似但更进一步优化，不仅节省 Log，而且省掉网络交互，大幅提升读的吞吐量并且能够显著降低延时。

Lease Read 基本思路是 Leader 取一个比 Election Timeout 小的租期（最好小一个数量级），在租约期内不会发生选举，确保 Leader 不会变化，所以跳过 ReadIndex 的第二步也就降低延时。由此可见 Lease Read 的正确性和时间是挂钩的，依赖本地时钟的准确性，因此虽然采用 Lease Read 做法非常高效，但是仍然面临风险问题，也就是存在预设的前提即各个服务器的 CPU Clock 的时间是准的，即使有误差，也会在一个非常小的 Bound 范围里面，时间的实现至关重要，如果时钟漂移严重，各个服务器之间 Clock 走的频率不一样，这套 Lease 机制可能出问题。

Lease Read 实现方式包括：

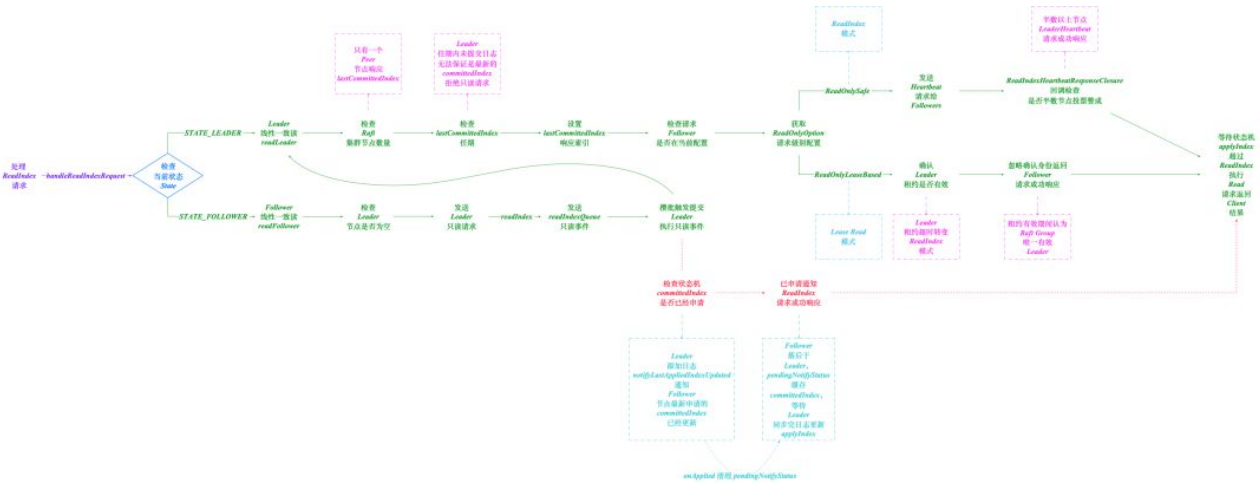
1. 定时 Heartbeat 获得多数派响应，确认 Leader 的有效性；
2. 在租约有效时间内，可以认为当前 Leader 是 Raft Group 内的唯一有效 Leader，可忽略 ReadIndex 中的 Heartbeat 确认步骤(2)；
3. Leader 等待自己的状态机执行，直到 applyIndex 超过 ReadIndex，这样就能够安全的提供 Linearizable Read。

SOFAJRaft 线性一致读实现

SOFAJRaft 采用 ReadIndex 替代走 Raft 状态机的方案，简而言之是依靠 ReadIndex 原则直接从 Leader 读取结果：所有已经复制到多数派上的 Log（可视为写操作）被视为安全的 Log，Leader 状态机只要按照顺序执行到此条 Log之后，该 Log 所体现的数据就能对客户端 Client 可见，具体分解为以下四个步骤：

- Client 发起 Read 请求；
- Leader 确认最新复制到多数派的 LogIndex；
- Leader 确认身份；
- 在 LogIndex apply 后执行 Read 操作。

通过 ReadIndex 优化，SOFAJRaft 能够达到 RPC 上限的 80%。上面的步骤中发现第 3 步仍然需要 Leader 通过向 Followers 发送心跳确认自己的 Leader 身份，因为 Raft 集群中的 Leader 身份随时可能发生改变。所以 SOFAJRaft 采用 Lease Read 的方式把第 3 步 RPC 省略掉。租约理解为 Raft 集群给 Leader 一段租期 Lease 的身份保证，在此期间不会剥夺 Leader 的身份，这样当 Leader 收到 Read 请求之后，如果发现租期尚未到期，无需再通过和 Followers 通信来确认自己的 Leader 身份，这样跳过第 3 步的网络通信开销。通过 Lease Read 优化，SOFAJRaft 几乎已经能够达到 RPC 的上限。然而通过时钟维护租期本身并不是绝对的安全（时钟漂移问题），所以 SOFAJRaft 默认配置是线性一致读，因为通常情况下线性一致读性能已足够好。



ReadIndex Read 实现

默认情况下，SOFAJRaft 提供的线性一致读是基于 Raft 协议的 ReadIndex 实现，三副本的情况下 Leader 读的吞吐接近于 RPC 的吞吐上限，延迟取决于多数派中最慢的一个 Heartbeat Response。使用 Node#readIndex(byte [] requestContext, ReadIndexClosure done) 发起线性一致读请求，当安全读取时传入的 Closure 将被调用，正常情况下从状态机中读取数据返

回给客户端， SOFAJRaft 将保证读取的线性一致性。线性一致读在任何集群内的节点发起，并不需要强制要求放到 Leader 节点上，允许在 Follower 节点执行，因此大大降低 Leader 的读取压力。

RaftServerService#handleReadIndexRequest 接口根据当前节点状态为 STATE_LEADER，STATE_FOLLOWER 或者 STATE_TRANSFERRING 情况处理 ReadIndex 请求：

1、当前节点状态是 STATE_LEADER 即为 Leader 节点，接收 ReadIndex 请求调用 readLeader(request, ReadIndexResponse.newBuilder(), done) 方法提供线性一致读：

- 检查当前 Raft 集群节点数量，如果集群只有一个 Peer 节点直接获取投票箱 BallotBox 最新提交索引 lastCommittedIndex 即 Leader 节点当前 Log 的 commitIndex 构建 ReadIndexClosure 响应；
- 日志管理器 LogManager 基于投票箱 BallotBox 的 lastCommittedIndex 获取任期检查是否等于当前任期，如果不等于当前任期表示此 Leader 节点未在其任期内提交任何日志，需要拒绝只读请求；
- 校验 Raft 集群节点数量以及 lastCommittedIndex 所属任期符合预期，那么响应构造器设置其索引为投票箱 BallotBox 的 lastCommittedIndex，并且来自 Follower 的请求需要检查 Follower 是否在当前配置；
- 获取 ReadIndex 请求级别 ReadOnlyOption 配置，ReadOnlyOption 参数默认值为 ReadOnlySafe，ReadOnlySafe 通过与 Quorum 通信来保证只读请求的可线性化。按照 ReadOnlyOption 配置为ReadOnlySafe 调用 Replicator#sendHeartbeat(rid, closure) 方法向 Followers 节点发送 Heartbeat 心跳请求，发送心跳成功执行 ReadIndexHeartbeatResponseClosure 心跳响应回调；
- ReadIndex 心跳响应回调检查是否超过半数节点包括 Leader 节点自身投票赞成，半数以上节点返回客户端Heartbeat 请求成功响应，即 applyIndex 超过 ReadIndex 说明已经同步到 ReadIndex 对应的 Log 能够提供 Linearizable Read。

2、当前节点状态是 STATE_FOLLOWER 即为 Follower 节点，接收 ReadIndex 请求通过 readFollower(request, done) 方法支持线性一致读：

- 检查当前 Leader 节点是否为空，如果 Leader 节点为空表示当然任期没有 Leader 节点；
- Follower 节点调用 RpcService#readIndex(leaderId.getEndpoint(), newRequest, -1, closure) 方法向 Leader 发送 ReadIndex 请求，Leader 节点调用 readIndex(requestContext, done) 方法启动可线性化只读查询请求，只读服务添加请求发布 ReadIndex 事件到队列 readIndexQueue 即 Disruptor 的 Ring Buffer；

- ReadIndex 事件处理器 ReadIndexEventHandler 通过 MPSC Queue 模型攒批消费触发使用 executeReadIndexEvents(events) 执行 ReadIndex 事件，轮询 ReadIndex 事件封装 ReadIndexState 状态列表构建 ReadIndexResponseClosure 响应回调提交给 Leader 节点处理 ReadIndex 请求；

- Leader 节点调用 handleReadIndexRequest(request, readIndexResponseClosure) 方法进行 readLeader 线性一致读过程，返回投票箱 BallotBox 的 lastCommittedIndex。ReadIndex 响应回调遍历状态列表记录当前提交日志 Index，检查申请状态机最新 Log Entry 的 committedIndex 是否已经申请即比较状态机 appliedIndex 是否大于等于当前 committedIndex。由于 Leader 节点处理添加 Log Entry 请求发送心跳后投票箱 BallotBox 更新 lastCommittedIndex，当 Leader 节点的 lastCommittedIndex 大于当前的 lastCommittedIndex 就会创建提交 Log Entry 异步任务发布到 taskQueue 队列，申请任务处理器 ApplyTaskHandler 执行提交 LogEntry 申请任务，通知 Follower 节点最新申请的 committedIndex 已经更新。如果当前申请状态机的 applyIndex 超过 ReadIndex，那么通知 ReadIndex 请求成功返回给客户端。当前 Follower 节点落后于 Leader 时把 Leader 节点返回的committedIndex 放到 pendingNotifyStatus 缓存等待 Leader 节点同步完日志更新 applyIndex。

SOFAJRaft 基于 Batch+Pipeline Ack+ 全异步机制的 ReadIndex 核心逻辑：

```java
/**
 * Handle read index request.
 */
@Override
public void handleReadIndexRequest(final ReadIndexRequest request, final
RpcResponseClosure<ReadIndexResponse> done) {
    final long startMs = Utils.monotonicMs();
    this.readLock.lock();
    try {
        switch (this.state) {
            case STATE_LEADER:
                readLeader(request, ReadIndexResponse.newBuilder(), done);
                break;
            case STATE_FOLLOWER:
                readFollower(request, done);
                break;
            case STATE_TRANSFERRING:
                done.run(new Status(RaftError.EBUSY, "Is transferring leadership."));
                break;
            default:
                done.run(new Status(RaftError.EPERM, "Invalid state for readIndex: %s.", this.state));
                break;
        }
    } finally {
        this.readLock.unlock();
        this.metrics.recordLatency("handle-read-index", Utils.monotonicMs() - startMs);
        this.metrics.recordSize("handle-read-index-entries", request.getEntriesCount());
    }
}
```

Lease Read 实现

SOFAJRaft 针对更高性能要求场景保证集群内机器的 CPU 时钟同步需求，采用 Clock+Heartbeat 的 Lease Read 优化，通过设置 RaftOptions 的 ReadOnlyOption 参数为 ReadOnlyLeaseBased 实现，ReadOnlyLeaseBased 通过依赖 Leader 租约确保只读请求的可

线性化，可能受时钟漂移的影响。如果时钟漂移无限制，Leader 节点可能保持租约长于应有的时间（时钟可以向后移动/暂停而没有任何限制），此种情况下 ReadIndex 是不安全的。

SOFAJRaft 基于 Lease Read 线性一致读实现是通过 Leader 节点调用 handleReadIndexRequest 接口接收 ReadIndex 请求获取 ReadIndex 请求级别 ReadOnlyOption 配置，当 ReadOnlyOption 配置为 ReadOnlyLeaseBased 时确认 Leader 租约是否有效即检查 Heartbeat 间隔是否小于 election timeout 时间，Leader 租约超时需要转变为 ReadIndex 模式。Leader 租约有效期间认为当前 Leader 是 Raft Group 内的唯一有效 Leader，忽略 ReadIndex 发送 Heartbeat 确认身份步骤，直接返回 Follower 节点和本地节点 Read 请求成功响应。Leader 节点继续等待状态机执行，直到 applyIndex 超过 ReadIndex 安全提供 Linearizable Read。

SOFAJRaft 基于时钟和心跳实现的线性一致读 Lease Read 优化逻辑：

```
ReadOnlyOption readOnlyOpt = this.raftOptions.getReadOnlyOptions();
if (readOnlyOpt == ReadOnlyOption.ReadOnlyLeaseBased && !isLeaderLeaseValid()) {
    // If leader lease timeout, we must change option to ReadOnlySafe
    readOnlyOpt = ReadOnlyOption.ReadOnlySafe;
}

switch (readOnlyOpt) {
    case ReadOnlySafe:
        final List<PeerId> peers = this.conf.getConf().getPeers();
                Requires.requireTrue(peers != null && !peers.isEmpty(), "Empty peers");
        final ReadIndexHeartbeatResponseClosure heartbeatDone = new
ReadIndexHeartbeatResponseClosure(closure, respBuilder, quorum, peers.size());
        // Send heartbeat requests to followers
        for (final PeerId peer : peers) {
            if (peer.equals(this.serverId)) {
                continue;
            }
            this.replicatorGroup.sendHeartbeat(peer, heartbeatDone);
        }
        break;
    case ReadOnlyLeaseBased:
        // Responses to followers and local node.
        respBuilder.setSuccess(true);
        closure.setResponse(respBuilder.build());
        closure.run(Status.OK());
        break;
}
```

总结

本文围绕 Raft Log Read，ReadIndex Read 以及 Lease Read 线性一致读实现细节方面剖析 SOFAJRaft 线性一致读基本原理，阐述 SOFAJRaft 如何使用 Batch+Pipeline Ack+全异步机制和 Clock+Heartbeat 手段优化 ReadIndex 和 Lease Read 线性一致读具体实现。

欢迎 Star SOFAJRaft ：

https://github.com/sofastack/sofa-jraft