Raft实战——线性一致性实现方法及性能优化

原创 Q的博客 Q的博客 2020-12-18 12:11:35 手机阅读

壹伴图
Q的博客
qblog0
月发文数目：**
月平均阅读：**
文章工具
已发文
采集图文 合成多
采集样式 查看

收录于合集

#Raft 7 #分布式系统 7 #共识算法 7 #一致性 7 #架构 7

本文是《Raft 实战系列——理论篇》的最后一文，首先介绍 Raft 协议场景采用"写主读从"、"写主读主"两种模式均无法保障线性一致性的原因；其次介绍基于 Raft 协议实现工程系统时，如何来保证线性一致性；最后介绍 Raft 针对一致性所做的读性能优化的具体策略。

系列前文快速链接：

在前面5篇文章中，我们分别介绍了《Raft 基本概念》、《Raft 选主机制》、《Raft 基于日志复制实现状态机机制》、《Raft 选主及状态机维护的安全性》、《Raft 集群变更防脑裂 & 解决数据膨胀》，《线性一致性概念介绍》，系统学习 Raft 协议建议从头阅读。

---

1. 线性一致性基础

数据一致性是为提升系统可用性所采用多副本机制所带来的新问题。在上文《Raft 实战番外篇——线性一致性》中我们重点解析了线性一致性的模型及挑战，对线性一致性不了解的同学，先阅读上文再学习下面内容会更容易理解吸收~

2. Raft 线性一致性读

在了解了什么是线性一致性之后，我们将其与 Raft 结合来探讨。首先需要明确一个问题，使用了 Raft 的系统都是线性一致的吗？不是的，Raft 只是提供了一个基础，要实现整个系统的线性一致还需要做一些额外的工作。

假设我们期望基于 Raft 实现一个线性一致的分布式 kv 系统，让我们从最朴素的方案开始，指出每种方案存在的问题，最终使整个系统满足线性一致性

2.1 写主读从缺陷分析

写操作并不是我们关注的重点，如果你稍微看了一些理论部分就应该知道，所有写操作都要作为提案从 leader 节点发起，当然所有的写命令都应该简单交给 leader 处理。真正关键的点在于**读操作的处理方式，这涉及到整个系统关于一致性方面的取舍**。

在该方案中我们假设读操作直接简单地向 follower 发起，那么由于 Raft 的 Quorum 机制（大部分节点成功即可），针对某个提案在某一时间段内，集群可能会有以下两种状态：

- 某次写操作的日志尚未被复制到一少部分 follower，但 leader 已经将其 commit。
- 某次写操作的日志已经被同步到所有 follower，但 leader 将其 commit 后，心跳包尚未通知到一部分 follower。

以上每个场景客户端都可能读到**过时的数据**，整个系统显然是不满足线性一致的。

2.2 写主读主缺陷分析

在该方案中我们限定，所有的读操作也必须经由 leader 节点处理，读写都经过 leader 难道还不能满足线性一致？**是的！！** 并且该方案存在不止一个问题！！

问题一：状态机落后于 committed log 导致脏读

回想一下前文讲过的，我们在解释什么是 commit 时提到了写操作什么时候可以响应客户端：

所谓 commit 其实就是对日志简单进行一个标记，表明其可以被 apply 到状态机，并针对相应的客户端请求进行响应。

也就是说一个提案只要被 leader commit 就可以响应客户端了，Raft 并没有限定提案结果在返回给客户端前必须先应用到状态机。所以从客户端视角当我们的某个写操作执行成功后，下一次读操作可能还是会读到旧值。

这个问题的解决方式很简单，在 leader 收到读命令时我们只需记录下当前的 commit index，当 apply index 追上该 commit index 时，即可将状态机中的内容响应给客户端。

问题二：网络分区导致脏读

假设集群发生网络分区，旧 leader 位于少数派分区中，而且此刻旧 leader 刚好还未发现自己已经失去了领导权，当多数派分区选出了新的 leader 并开始进行后续写操作时，连接到旧 leader 的客户端可能就会读到旧值了。

因此，仅仅是直接读 leader 状态机的话，系统仍然不满足线性一致性。

2.3 Raft Log Read

为了确保 leader 处理读操作时仍拥有领导权，我们可以将读请求同样作为一个提案走一遍 Raft 流程，当这次读请求对应的日志可以被应用到状态机时，leader 就可以读状态机并返回给用户了。

这种读方案称为 **Raft Log Read**，也可以直观叫做 **Read as Proposal**。

为什么这种方案满足线性一致？因为该方案根据 commit index 对所有读写请求都一起做了线性化，这样每个读请求都能感知到状态机在执行完前一写请求后的最新状态，将读写日志一条一条的应用到状态机，整个系统当然满足线性一致。但该方案的缺点也非常明显，那就是**性能差**，读操作的开销与写操作几乎完全一致。而且由于所有操作都线性化了，我们无法并发读状态机。

3. Raft 读性能优化

接下来我们将介绍几种优化方案，它们在不违背系统线性一致性的前提下，大幅提升了读性能。

3.1 Read Index

与 Raft Log Read 相比，Read Index 省掉了同步 log 的开销，能够**大幅提升**读的**吞吐**，**一定程度上降低**读的**时延**。其大致流程为：

1. Leader 在收到客户端读请求时，记录下当前的 commit index，称之为 read index。
2. Leader 向 followers 发起一次心跳包，这一步是为了确保领导权，避免网络分区时少数派 leader 仍处理请求。
3. 等待状态机**至少**应用到 read index（即 apply index **大于等于** read index）。
4. 执行读请求，将状态机中的结果返回给客户端。

这里第三步的 apply index **大于等于** read index 是一个关键点。因为在该读请求发起时，我们将当时的 commit index 记录了下来，只要使客户端读到的内容在该 commit index 之后，那么结果**一定都满足线性一致**（如不理解可以再次回顾下前文线性一致性的例子以及2.2中的问题一）。

3.2 Lease Read

与 Read Index 相比，Lease Read 进一步省去了网络交互开销，因此更能**显著降低**读的**时延**。

基本思路是 leader 设置一个**比选举超时（Election Timeout）更短的时间作为租期**，在租期内我们可以相信其它节点一定没有发起选举，集群也就一定不会存在脑裂，所以在这个时间段内我们直接读主即可，而非该时间段内可以继续走 Read Index 流程，Read Index 的心跳包也可以为租期带来更新。

Lease Read 可以认为是 Read Index 的时间戳版本，额外依赖时间戳会为算法带来一些不确定性，如果时钟发生漂移会引发一系列问题，因此需要谨慎的进行配置。

3.3 Follower Read

在前边两种优化方案中，无论我们怎么折腾，核心思想其实只有两点：

- 保证在读取时的最新 commit index 已经被 apply。
- 保证在读取时 leader 仍拥有领导权。

这两个保证分别对应2.2节所描述的两个问题。

其实无论是 Read Index 还是 Lease Read，最终目的都是为了解决第二个问题。换句话说，读请求最终一定都是由 leader 来承载的。

那么读 follower 真的就不能满足线性一致吗？其实不然，这里我们给出一个可行的读 follower 方案：**Follower 在收到客户端的读请求时，向 leader 询问当前最新的 commit index，反正所有日志条目最终一定会被同步到自己身上，follower 只需等待该日志被自己 commit 并 apply 到状态机后，返回给客户端本地状态机的结果即可**。这个方案叫做 **Follower Read**。

注意：Follower Read 并不意味着我们在读过程中完全不依赖 leader 了，在保证线性一致性的前提下完全不依赖 leader 理论上是不可能做到的。

至此，我们 Raft 系列的原理篇就已经完结了。

如果你一路坚持看了下来，相信已经对 Raft 算法的理论有了深刻的理解。当然，理论和工程实践之间存在的鸿沟可能比想象的还要大，实践中有众多的细节问题需要去面对。在后续的源码分析及实践篇中，我们会结合代码讲解到许多理论部分没有提到的这些细节点，并介绍基础架构设计的诸多经验，敬请期待！

上一篇
深度解析 Raft 分布式一致性协议（长文）

下一篇
Raft实战（番外篇）—— 线性一致性介绍