**引言**

Controller是Kubevirt的重要逐渐之一，其核心思想是事件驱动，进行相关的业务逻辑处理。为了使Controller支持HA，通常需要部署多个Controller实例，由于是事件驱动，因此会导致相关业务逻辑被处理多次。为了解决这一个问题，需要引入分布式锁机制，使得某一个时刻仅有一个Controller实例提供服务。本文重点介绍相关调研，以及一些示例代码。



**第一部分：分布式一致性算法 Raft 原理 及 Etcd 介绍**

1. 分布式一致性算法 Raft 概述

“工欲善其事，必先利其器。” 懂得原理方能触类旁通，立于不败之地。本场 Chat 将分 4 节详细解读著名的分布式一致性算法——Raft，在此基础上，再介绍 Etcd 的架构和典型应用场景。

1.1 Raft 背景

在分布式系统中，一致性算法至关重要。在所有一致性算法中，Paxos 最负盛名，它由莱斯利 · 兰伯特（Leslie Lamport）于 1990 年提出，是一种基于消息传递的一致性算法，被认为是类似算法中最有效的。

Paxos 算法虽然很有效，但复杂的原理使它实现起来非常困难，截止目前，实现 Paxos 算法的开源软件很少，比较出名的有 Chubby、libpaxos。此外 Zookeeper 采用的 ZAB（Zookeeper Atomic Broadcast）协议也是基于 Paxos 算法实现的，不过 ZAB 对 Paxos 进行了很多的改进与优化，两者的设计目标也存在差异——ZAB 协议主要用于构建一个高可用的分布式数据主备系统，而 Paxos 算法则是用于构建一个分布式的一致性状态机系统。

由于 Paxos 算法过于复杂、实现困难，极大的制约了其应用，而分布式系统领域又亟需一种高效而易于实现的分布式一致性算法，在此背景下，Raft 算法应运而生。

Raft 算法由斯坦福的 Diego Ongaro 和 John Ousterhout 于 2013 年发表：《In Search of an Understandable Consensus Algorithm》。相较于 Paxos，Raft 通过逻辑分离使其更容易理解和实现，目前，已经有十多种语言的 Raft 算法实现框架，较为出名的有 etcd、Consul 。

1.2 Raft 角色

一个 Raft 集群包含若干节点，Raft 把这些节点分为三种状态：Leader、 Follower 、Candidate，每种状态负责的任务也是不一样的，正常情况下，集群中的节点只存在 Leader 与 Follower 两种状态。

* **Leader（领导者）**：负责日志的同步管理，处理来自客户端的请求，与 Follower 保持 heartBeat 的联系；
* **Follower（追随者）**：响应 Leader 的日志同步请求，响应 Candidate 的邀票请求，以及把客户端请求到 Follower 的事务转发（重定向）给 Leader；
* **Candidate（候选者）**：负责选举投票，集群刚启动或者 Leader 宕机时，状态为 Follower 的节点将转为 Candidate 并发起选举，选举胜出（获得超过半数节点的投票）后，从 Candidate 转为 Leader 状态；

1.3 Raft 概述

通常，Raft 集群中只有一个 Leader，其它节点都是 Follower 。Follower 都是被动的：它们不会发送任何请求，只是简单的响应来自 Leader 或者 Candidate 的请求。Leader 负责处理所有的客户端请求（如果一个客户端和 Follower 联系，那么 Follower 会把请求重定向给 Leader）。为简化逻辑和实现，Raft 将一致性问题分解成了三个相对独立的子问题：

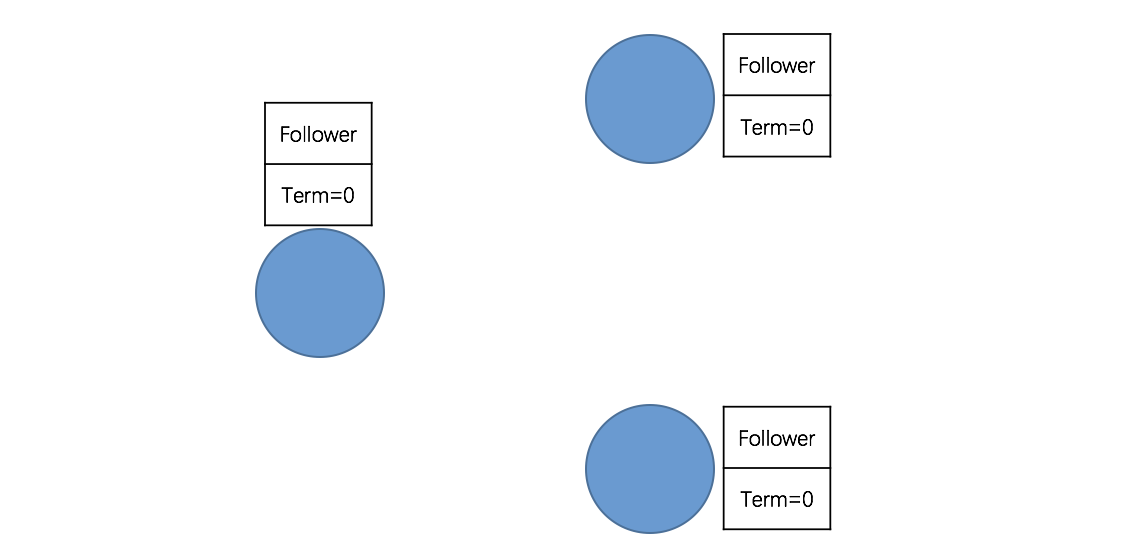
* **选举（Leader election）**：当 Leader 宕机或者集群初创时，一个新的 Leader 需要被选举出来；
* **日志复制（Log replication）**：Leader 接收来自客户端的请求并将其以日志条目的形式复制到集群中的其它节点，并且强制要求其它节点的日志和自己保持一致。
* **安全性（Safety）**：如果有任何的服务器节点已经应用了一个确定的日志条目到它的状态机中，那么其它服务器节点不能在同一个日志索引位置应用一个不同的指令。

2. Raft 算法之 Leader election 原理

根据 Raft 协议，一个应用 Raft 协议的集群在刚启动时，所有节点的状态都是 Follower，由于没有 Leader，Followers 无法与 Leader 保持心跳（Heart Beat），因此，Followers 会认为 Leader 已经 down，进而转为 Candidate 状态。然后，Candidate 将向集群中其它节点请求投票，同意自己升级为 Leader，如果 Candidate 收到超过半数节点的投票（N/2 + 1），它将获胜成为 Leader。

第一阶段：所有节点都是 Follower

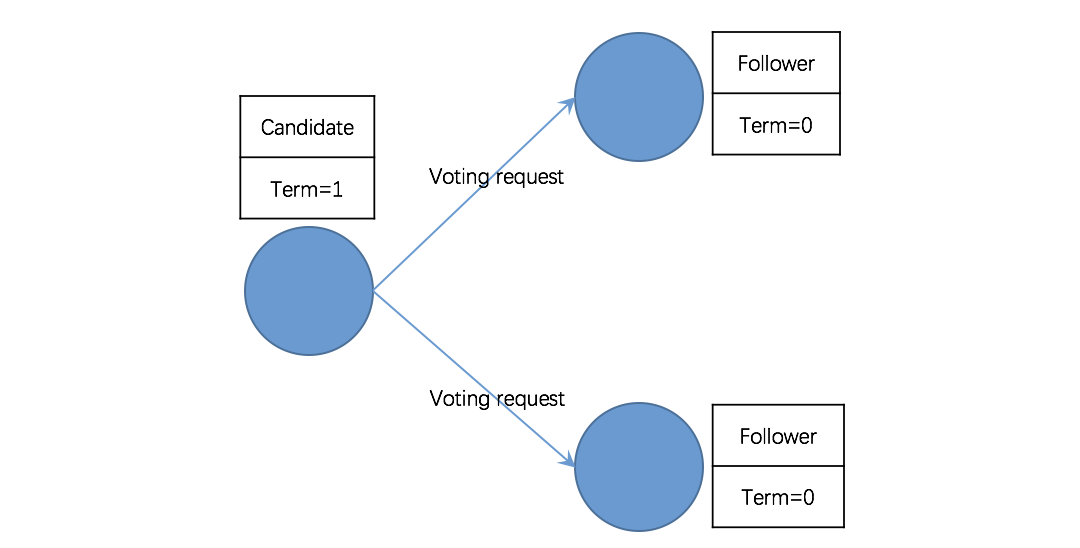
根据 Raft 协议，一个应用 Raft 协议的集群在刚启动时（或者 Leader 宕机），所有节点的状态都是 Follower，初始 Term（任期）为 0。同时启动选举定时器，每个节点的选举定时器超时时间都在 100-500 毫秒之间且并不一致（避免同时发起选举）。



第二阶段：Follower 转为 Candidate 并发起投票

由于没有 Leader，Followers 无法与 Leader 保持心跳（Heart Beat），节点启动后在一个选举定时器周期内未收到心跳和投票请求，则状态转为候选者 candidate 状态、Term 自增，并向集群中所有节点发送投票请求并且重置选举定时器。

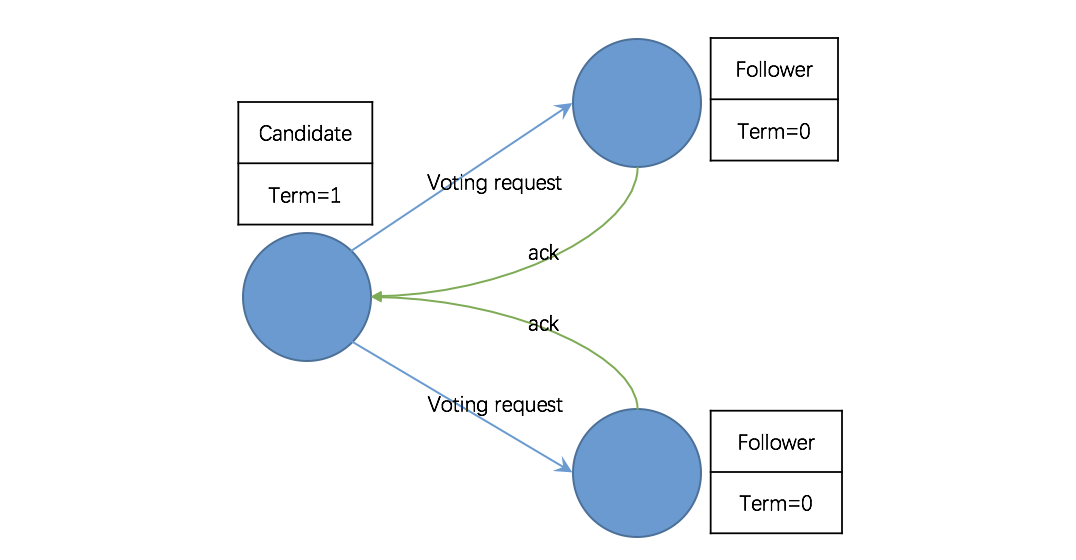
注意：由于每个节点的选举定时器超时时间都在 100-500 毫秒之间，且不一致，因此，可以避免所有的 Follower 同时转为 Candidate 并发起投票请求。换言之，最先转为 Candidate 并发起投票请求的节点将具有成为 Leader 的 “先发优势”。



第三阶段：投票策略

节点收到投票请求后会根据以下情况决定是否接受投票请求：

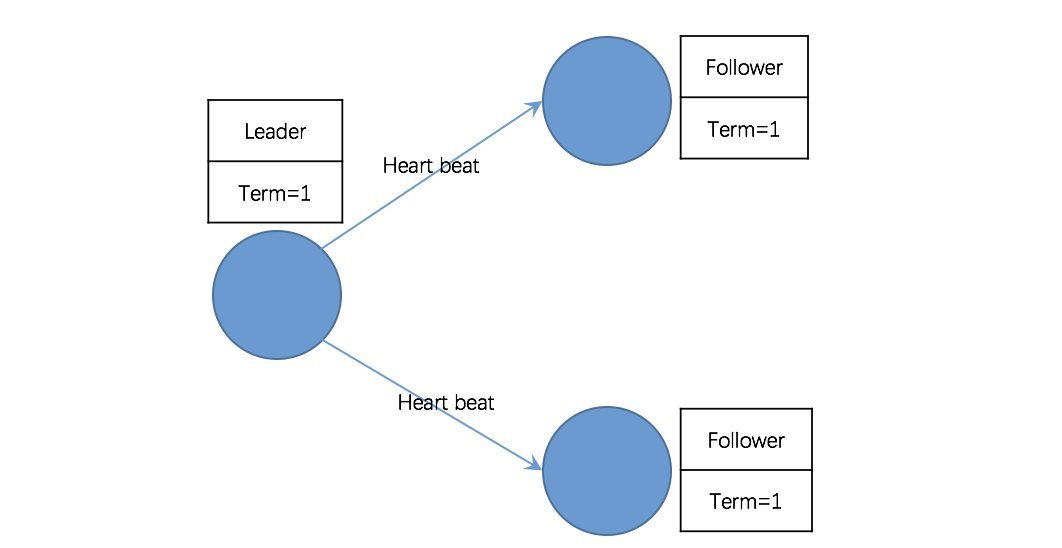
* 请求节点的 Term 大于自己的 Term，且自己尚未投票给其它节点，则接受请求，把票投给它；
* 请求节点的 Term 小于自己的 Term，且自己尚未投票，则拒绝请求，将票投给自己。



第四阶段：Candidate 转为 Leader

一轮选举过后，正常情况下，会有一个 Candidate 收到超过半数节点（n/2 + 1）的投票，那么它将胜出并升级为 Leader，然后定时发送心跳给其它的节点，其它节点会转为 Follower 并与 Leader 保持同步，如此，本轮选举结束。

注意：有可能一轮选举中，没有 Candidate 收到超过半数节点投票，那么将进行下一轮选举。

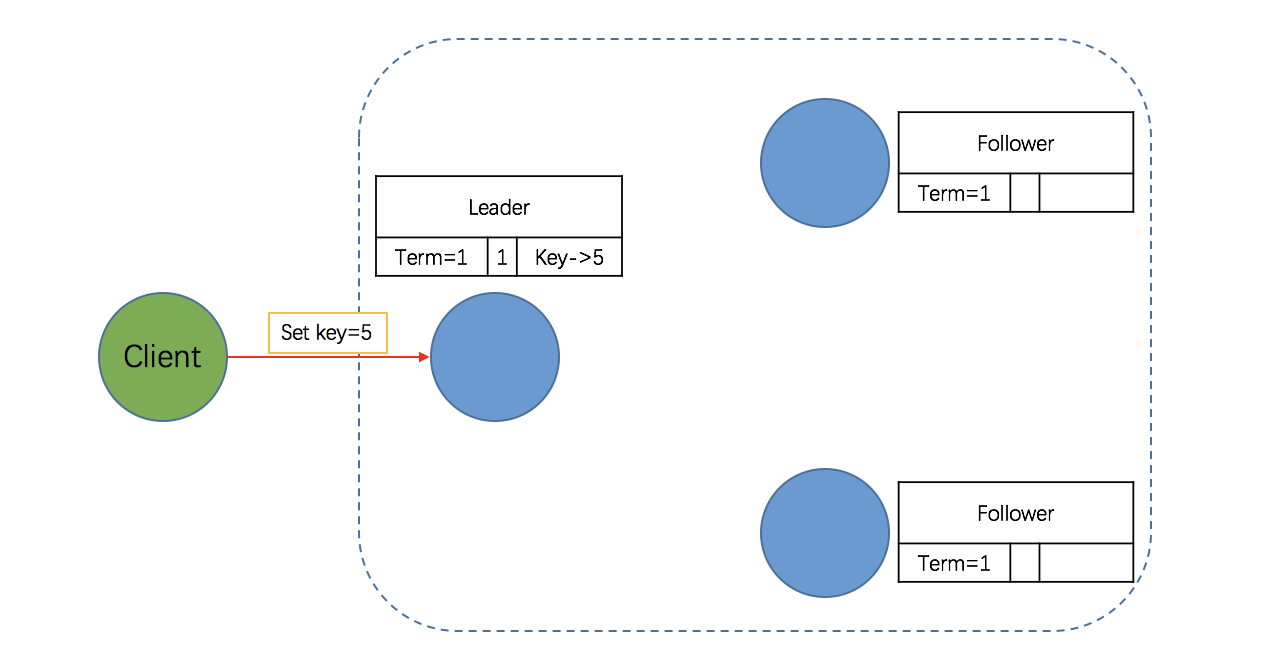


3. Raft 算法之 Log replication 原理

在一个 Raft 集群中只有 Leader 节点能够处理客户端的请求（如果客户端的请求发到了 Follower，Follower 将会把请求重定向到 Leader），客户端的每一个请求都包含一条被复制状态机执行的指令。Leader 把这条指令作为一条新的日志条目（entry）附加到日志中去，然后并行的将附加条目发送给 Followers，让它们复制这条日志条目。当这条日志条目被 Followers 安全的复制，Leader 会应用这条日志条目到它的状态机中，然后把执行的结果返回给客户端。如果 Follower 崩溃或者运行缓慢，再或者网络丢包，Leader 会不断的重复尝试附加日志条目（尽管已经回复了客户端）直到所有的 Follower 都最终存储了所有的日志条目，确保强一致性。

第一阶段：客户端请求提交到 Leader

如下图所示，Leader 收到客户端的请求：如存储一个数据：5；Leader 收到请求后，会将它作为日志条目（entry）写入本地日志中。需要注意的是，此时该 entry 的状态是未提交（uncommitted），Leader 并不会更新本地数据，因此它是不可读的。



第二阶段：Leader 将 entry 发送到其它 Follower

Leader 与 Floolwers 之间保持者心跳联系，随心跳 Leader 将追加的 entry（AppendEntries）并行的发送到其它的 Follower，并让它们复制这条日志条目，这一过程称为复制（replicate）。有几点需要注意：

**1. 为什么 Leader 向 Follower 发送的 entry 是 AppendEntries 呢？**

因为 Leader 与 Follower 的心跳是周期性的，而一个周期间 Leader 可能接收到多条客户端的请求，因此，随心跳向 Followers 发送的大概率是多个 entry，即 AppendEntries。当然，在本例中，我们假设只有一条请求，自然也就是一个 entry 了。

**2. Leader 向 Followers 发送的不仅仅是追加的 entry（AppendEntries）。**

在发送追加日志条目的时候，Leader 会把新的日志条目紧接着之前的条目的索引位置（prevLogIndex）和 Leader 任期号（term）包含在里面。如果 Follower 在它的日志中找不到包含相同索引位置和任期号的条目，那么它就会拒绝接收新的日志条目，因为出现这种情况说明 Follower 和 Leader 是不一致的。

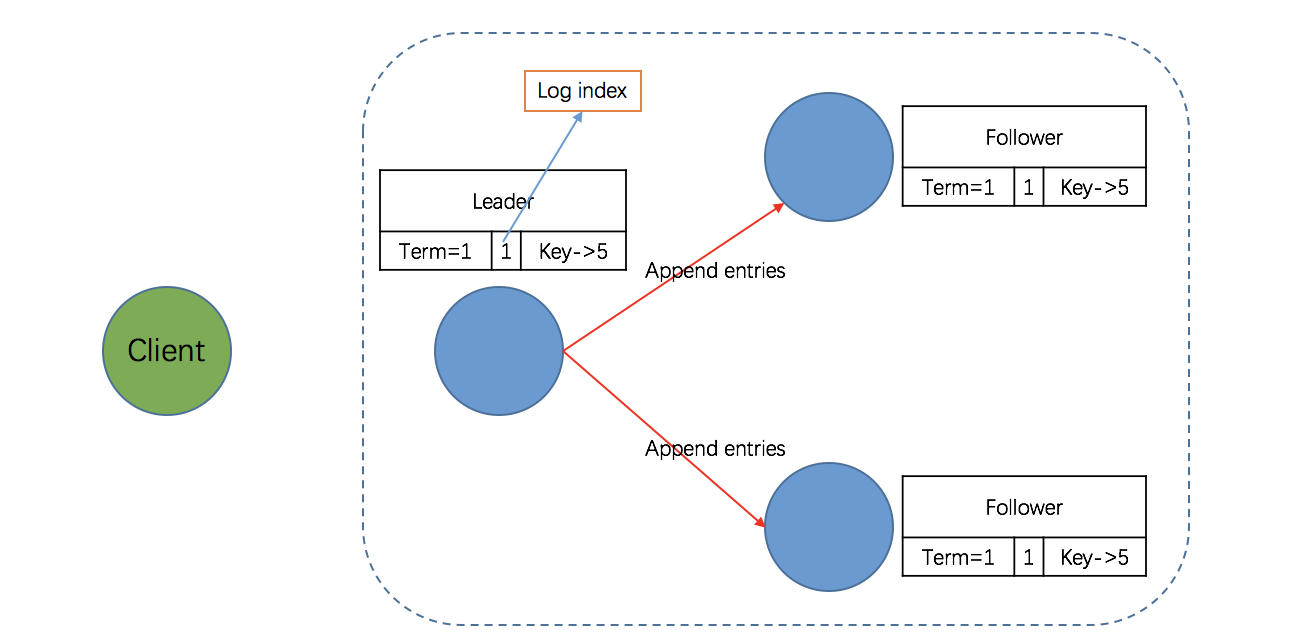
**3. 如何解决 Leader 与 Follower 不一致的问题？**

在正常情况下，Leader 和 Follower 的日志保持一致性，所以追加日志的一致性检查从来不会失败。然而，Leader 和 Follower 的一系列崩溃的情况会使得它们的日志处于不一致的状态。Follower 可能会丢失一些在新的 Leader 中有的日志条目，它也可能拥有一些 Leader 没有的日志条目，或者两者都发生。丢失或者多出日志条目可能会持续多个任期。

要使 Follower 的日志与 Leader 恢复一致，Leader 必须找到最后两者达成一致的地方（说白了就是回溯，找到两者最近的一致点），然后删除从那个点之后的所有日志条目，发送自己的日志给 Follower。所有的这些操作都在进行附加日志的一致性检查时完成。

Leader 针对每一个 Follower 维护了一个 nextIndex，这表示下一个需要发送给 Follower 的日志条目的索引地址。当一个 Leader 刚获得权力的时候，它初始化所有的 nextIndex 值为自己的最后一条日志的 index 加 1。如果一个 Follower 的日志和 Leader 不一致，那么在下一次的附加日志时的一致性检查就会失败。在被 Follower 拒绝之后，Leader 就会减小该 Follower 对应的 nextIndex 值并进行重试。

最终 nextIndex 会在某个位置使得 Leader 和 Follower 的日志达成一致。当这种情况发生，附加日志就会成功，这时就会把 Follower 冲突的日志条目全部删除并且加上 Leader 的日志。一旦附加日志成功，那么 Follower 的日志就会和 Leader 保持一致，并且在接下来的任期里一直继续保持。

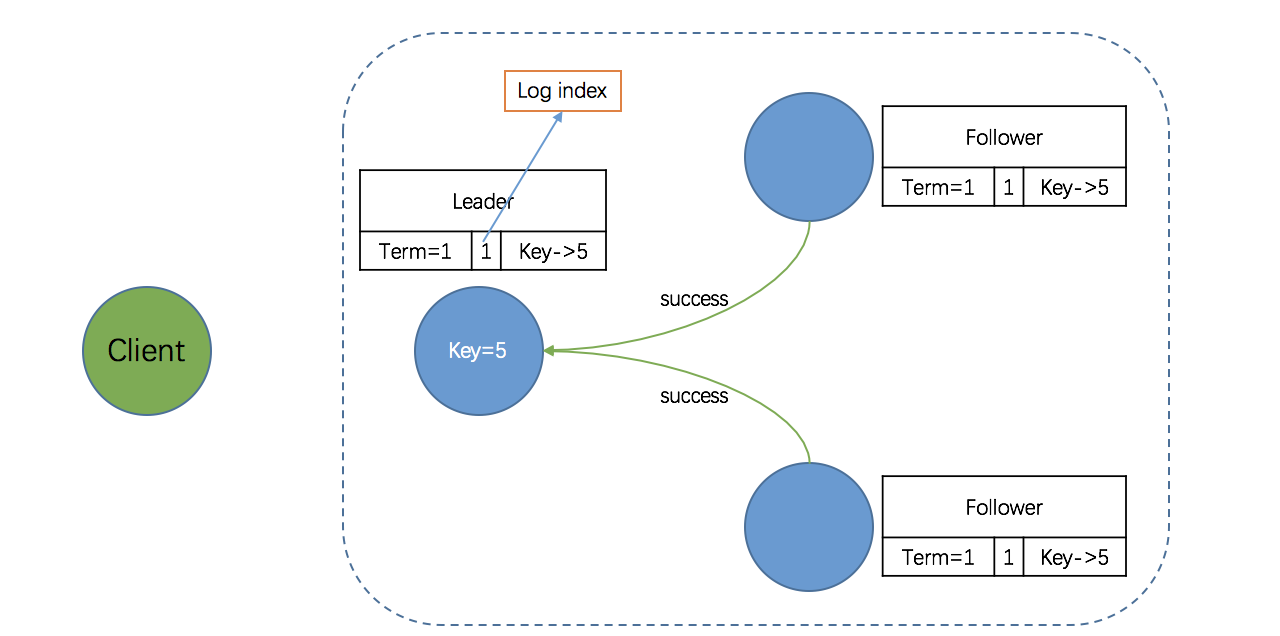


第三阶段：Leader 等待 Followers 回应

Followers 接收到 Leader 发来的复制请求后，有两种可能的回应：

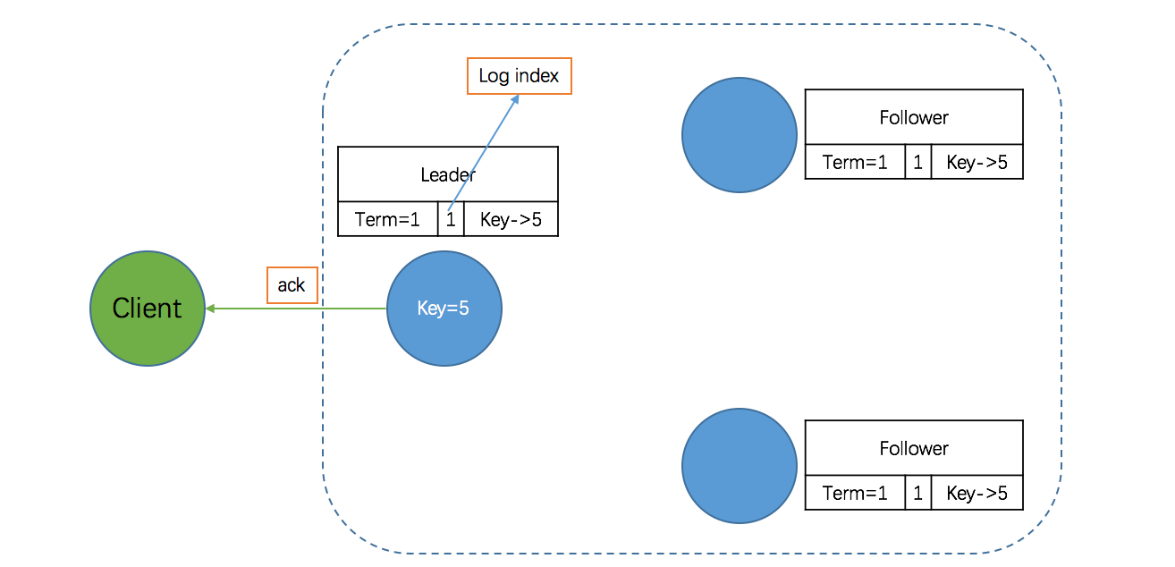
* 写入本地日志中，返回 Success；
* 一致性检查失败，拒绝写入，返回 false，原因和解决办法上面已经详细说明。

需要注意的是，此时该 entry 的状态也是未提交（uncommitted）。完成上述步骤后，Followers 会向 Leader 发出回应 - success，当 Leader 收到大多数 Followers 的回应后，会将第一阶段写入的 entry 标记为提交状态（committed），并把这条日志条目应用到它的状态机中。



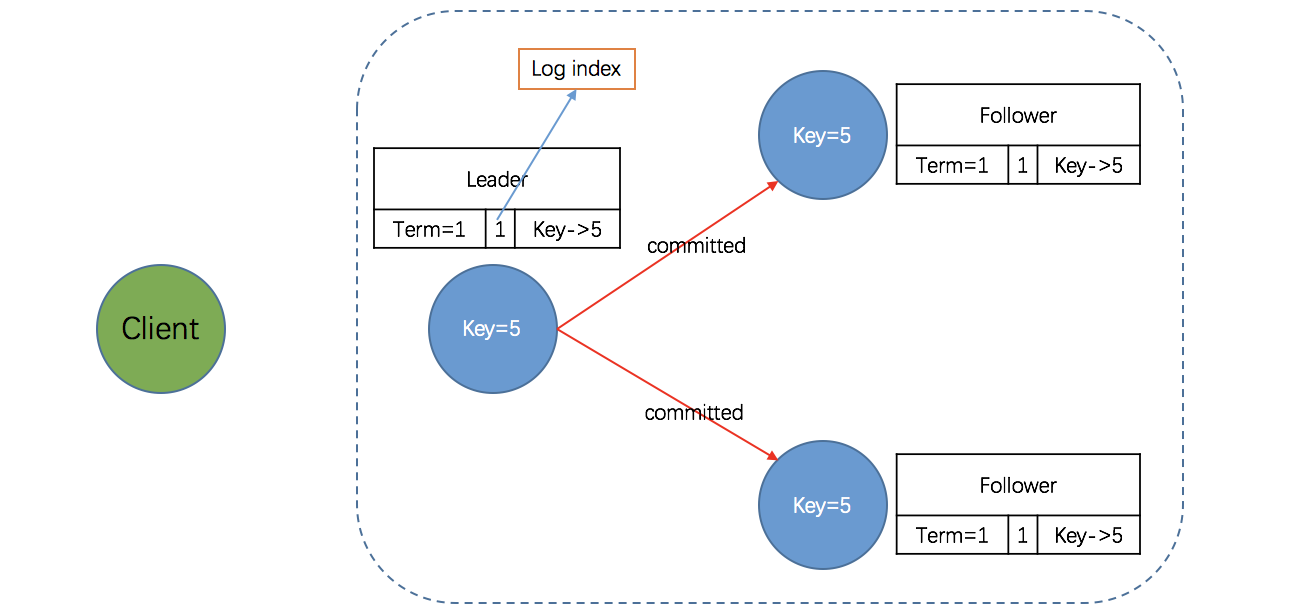
第四阶段：Leader 回应客户端

完成前三个阶段后，Leader 会回应客户端 -OK，写操作成功。



第五阶段：Leader 通知 Followers entry 已提交

Leader 回应客户端后，将随着下一个心跳通知 Followers，Followers 收到通知后也会将 entry 标记为提交状态。至此，Raft 集群超过半数节点已经达到一致状态，可以确保强一致性。需要注意的是，由于网络、性能、故障等各种原因导致 “反应慢” 、“不一致” 等问题的节点，也会最终与 Leader 达成一致。



4. Raft 算法之安全性

前面的章节里描述了 Raft 算法是如何选举 Leader 和复制日志的。然而，到目前为止描述的机制并不能充分的保证每一个状态机会按照相同的顺序执行相同的指令。例如，一个 Follower 可能处于不可用状态，同时 Leader 已经提交了若干的日志条目；然后这个 Follower 恢复（尚未与 Leader 达成一致）而 Leader 故障；如果该 Follower 被选举为 Leader 并且覆盖这些日志条目，就会出现问题：不同的状态机执行不同的指令序列。

鉴于此，在 Leader 选举的时候需增加一些限制来完善 Raft 算法。这些限制可保证任何的 Leader 对于给定的任期号（Term），都拥有之前任期的所有被提交的日志条目（所谓 Leader 的完整特性）。关于这一选举时的限制，下文将详细说明。

4.1 选举限制

对于所有基于 Leader 机制的一致性算法，Leader 都必须存储所有已经提交的日志条目。为了保障这一点，Raft 使用了一种简单而有效的方法，以保证所有之前的任期号中已经提交的日志条目在选举的时候都会出现在新的 Leader 中。换言之，日志条目的传送是单向的，只从 Leader 传给 Follower，并且 Leader 从不会覆盖自身本地日志中已经存在的条目。

Raft 使用投票的方式来阻止一个 Candidate 赢得选举除非这个 Candidate 包含了所有已经提交的日志条目。Candidate 为了赢得选举必须联系集群中的大部分节点，这意味着每一个已经提交的日志条目在这些服务器节点中肯定存在于至少一个节点上。如果 Candidate 的日志至少和大多数的服务器节点一样新（这个新的定义会在下面讨论），那么它一定持有了所有已经提交的日志条目（多数派的思想）。投票请求的限制： 请求中包含了 Candidate 的日志信息，然后投票人会拒绝那些日志没有自己新的投票请求。

Raft 通过比较两份日志中最后一条日志条目的索引值和任期号确定谁的日志比较新。如果两份日志最后的条目的任期号不同，那么任期号大的日志更加新。如果两份日志最后的条目任期号相同，那么日志比较长的那个就更加新。

4.2 提交之前任期内的日志条目

如同 4.1 节介绍的那样，Leader 知道一条当前任期内的日志记录是可以被提交的，只要它被复制到了大多数的 Follower 上（多数派的思想）。如果一个 Leader 在提交日志条目之前崩溃了，继任的 Leader 会继续尝试复制这条日志记录。然而，一个 Leader 并不能断定一个之前任期里的日志条目被保存到大多数 Follower 上就一定已经提交了。这很明显，从日志复制的过程可以看出。

鉴于上述情况，Raft 算法不会通过计算副本数目的方式去提交一个之前任期内的日志条目。只有 Leader 当前任期里的日志条目通过计算副本数目可以被提交；一旦当前任期的日志条目以这种方式被提交，那么由于日志匹配特性，之前的日志条目也都会被间接的提交。在某些情况下，Leader 可以安全的知道一个老的日志条目是否已经被提交（只需判断该条目是否存储到所有节点上），但是 Raft 为了简化问题使用一种更加保守的方法。

当 Leader 复制之前任期里的日志时，Raft 会为所有日志保留原始的任期号, 这在提交规则上产生了额外的复杂性。但是，这种策略更加容易辨别出日志，因为它可以随着时间和日志的变化对日志维护着同一个任期编号。此外，该策略使得新 Leader 只需要发送较少日志条目。

5. Etcd 介绍

Etcd 是一个高可用、强一致的分布式键值（key-value）数据库，主要用途是共享配置和服务发现，其内部采用 Raft 算法作为分布式一致性协议，因此，Etcd 集群作为一个分布式系统 “天然” 就是强一致性的。而副本机制（一个 Leader，多个 Follower）又保证了其高可用性。

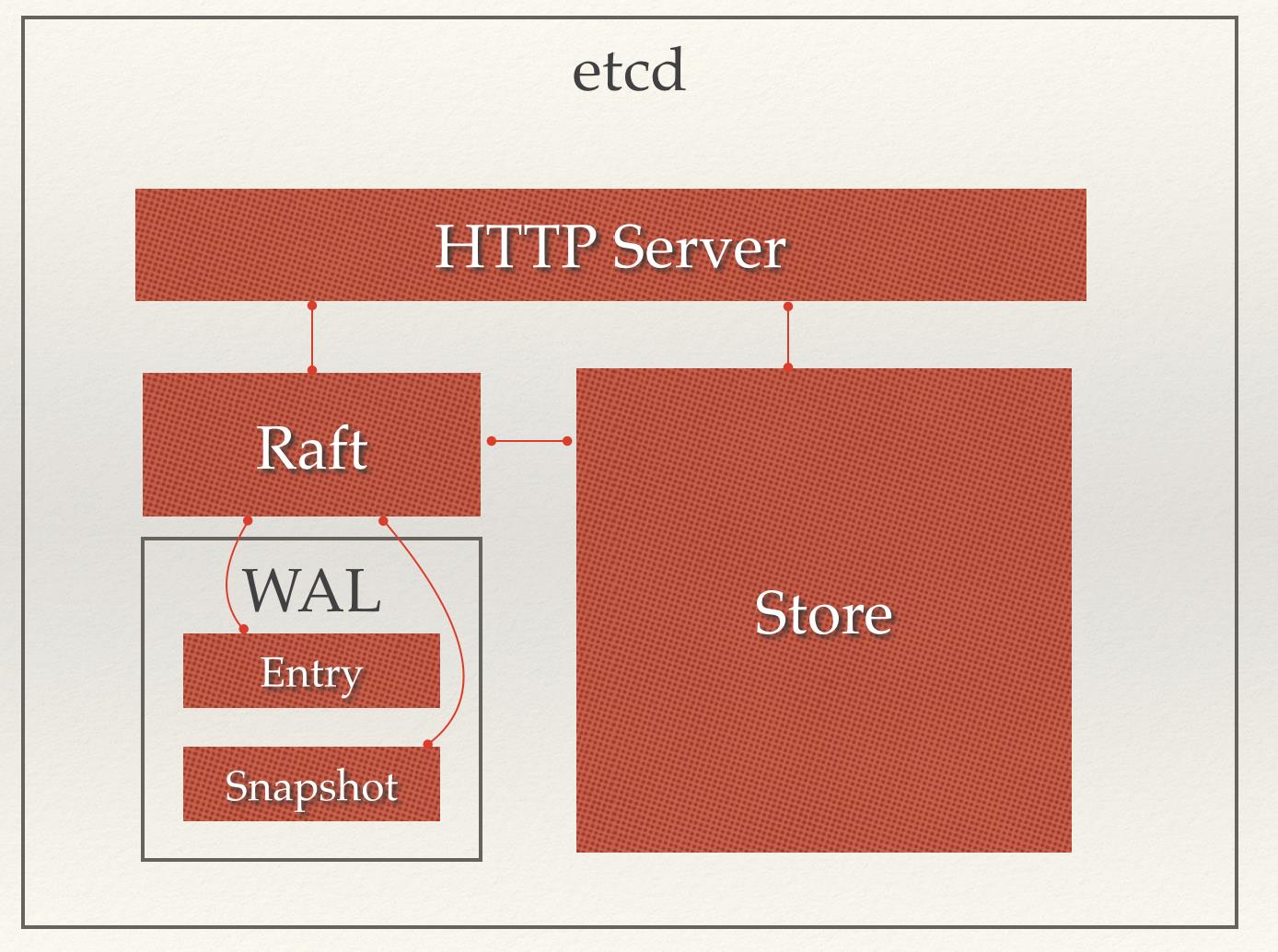
点击进入：[Etcd 官网](https://coreos.com/etcd/)

**关于 Etcd 命名的由来**

在 Unix 系统中，/etc 目录用于存放系统管理和配置文件；分布式系统（Distributed system）第一个字母是“d”。两者看上去并没有直接联系，但它们加在一起就有点意思了：分布式的关键数据（系统管理和配置文件）存储系统，这便是 etcd 命名的灵感之源。

5.1 Etcd 架构

Etcd 的架构图如下，从架构图中可以看出，Etcd 主要分为四个部分：HTTP Server、Store、Raft 以及 WAL。



* HTTP Server： 用于处理客户端发送的 API 请求以及其它 Etcd 节点的同步与心跳信息请求。
* Store：用于处理 Etcd 支持的各类功能的事务，包括数据索引、节点状态变更、监控与反馈、事件处理与执行等等，是 Etcd 对用户提供的大多数 API 功能的具体实现。
* Raft：Raft 强一致性算法的具体实现，是Etcd 的核心。
* WAL：Write Ahead Log（预写式日志），是 Etcd 的数据存储方式。除了在内存中存有所有数据的状态以及节点的索引外，Etcd 就通过 WAL 进行持久化存储。WAL中，所有的数据提交前都会事先记录日志。Snapshot 是为了防止数据过多而进行的状态快照；Entry 表示存储的具体日志内容。

通常，一个用户的请求发送过来，会经由 HTTP Server 转发给 Store 进行具体的事务处理；如果涉及到节点的修改，则交给Raft模块进行状态的变更、日志的记录，然后再同步给别的 Etcd 节点以确认数据提交；最后进行数据的提交，再次同步。

5.2 Etcd 的基本概念词

由于 Etcd 基于分布式一致性算法——Raft，其涉及的概念词与 Raft 保持一致，如下所示，通过前面Raft算法的介绍，相信读者已经可以大体勾勒出 Etcd 集群的运作机制。

* Raft：Etcd 的核心，保证分布式系统强一致性的算法。
* Node：一个 Raft 状态机实例。
* Member： 一个 Etcd 实例，它管理着一个 Node，并且可以为客户端请求提供服务。
* Cluster：由多个 Member 构成可以协同工作的 Etcd 集群。
* Peer：对同一个 Etcd 集群中另外一个 Member 的称呼。
* Client： 向 Etcd 集群发送 HTTP 请求的客户端。
* WAL：预写式日志，Etcd 用于持久化存储的日志格式。
* Snapshot：Etcd 防止 WAL 文件过多而设置的快照，存储 Etcd 数据状态。
* Leader：Raft 算法中通过竞选而产生的处理所有数据提交的节点。
* Follower：竞选失败的节点作为 Raft 中的从属节点，为算法提供强一致性保证。
* Candidate：当 Follower 超过一定时间接收不到 Leader 的心跳时转变为 Candidate 开始竞选。
* Term：某个节点成为 Leader 到下一次竞选期间，称为一个 Term（任期）。
* Index：数据项编号。Raft 中通过 Term 和 Index 来定位数据。

5.3 Etcd 能做什么

在分布式系统中，有一个最基本的需求——如何保证分布式部署的多个节点之间的数据共享。如同团队协作，成员可以分头干活，但总是需要共享一些必须的信息，比如谁是 leader、团队成员列表、关联任务之间的顺序协调等。所以分布式系统要么自己实现一个可靠的共享存储来同步信息，要么依赖一个可靠的共享存储服务，而 Etcd 就是这样一个服务。

[Etcd 官方](https://coreos.com/etcd/)介绍：

A distributed, reliable key-value store for the most critical data of a distributed system.

简言之，一个可用于存储分布式系统关键数据的可靠的键值数据库。关于可靠性自不必多说，Raft 协议已经阐明，但事实上，Etcd 作为 key-value 型数据库还有其它特点：Watch 机制、租约机制、Revision 机制等，正是这些机制赋予了 Etcd 强大的能力。

* **Lease 机制**：即租约机制（TTL，Time To Live），Etcd 可以为存储的 key-value 对设置租约，当租约到期，key-value 将失效删除；同时也支持续约，通过客户端可以在租约到期之前续约，以避免 key-value 对过期失效；此外，还支持解约，一旦解约，与该租约绑定的 key-value 将失效删除；
* **Prefix 机制**：即前缀机制，也称目录机制，如两个 key 命名如下：key1=“/mykey/key1" , key2="/mykey/key2"，那么，可以通过前缀-"/mykey"查询，返回包含两个 key-value 对的列表；
* **Watch 机制**：即监听机制，Watch 机制支持 Watch 某个固定的key，也支持 Watch 一个范围（前缀机制），当被 Watch 的 key 或范围发生变化，客户端将收到通知；
* **Revision 机制**：每个key带有一个 Revision 号，每进行一次事务加一，因此它是全局唯一的，如初始值为 0，进行一次 put 操作，key 的 Revision 变为 1，同样的操作，再进行一次，Revision 变为 2；换成 key1 进行 put 操作，Revision 将变为 3；这种机制有一个作用：通过 Revision 的大小就可以知道进行写操作的顺序，这对于实现公平锁，队列十分有益。

5.4 Etcd 的主要应用场景

从 5.3 节的介绍中可以看出，Etcd 的功能非常强大，其功能点或功能组合可以实现众多的需求，以下列举一些典型应用场景。

**应用场景 1：服务发现**

服务发现（Service Discovery）要解决的是分布式系统中最常见的问题之一，即在同一个分布式集群中的进程或服务如何才能找到对方并建立连接。服务发现的实现原理如下：

* 存在一个高可靠、高可用的中心配置节点：基于 Ralf 算法的 Etcd 天然支持，不必多解释。
* 服务提供方会持续的向配置节点注册服务：用户可以在 Etcd 中注册服务，并且对注册的服务配置租约，定时续约以达到维持服务的目的（一旦停止续约，对应的服务就会失效）。
* 服务的调用方会持续的读取中心配置节点的配置并修改本机配置，然后 reload 服务：服务提供方在 Etcd 指定的目录（前缀机制支持）下注册的服务，服务调用方在对应的目录下查服务。通过 Watch 机制，服务调用方还可以监测服务的变化。

**应用场景 2： 消息发布和订阅**

在分布式系统中，组件间通信常用的方式是消息发布-订阅机制。具体而言，即配置一个配置共享中心，数据提供者在这个配置中心发布消息，而消息使用者则订阅他们关心的主题，一旦有关主题有消息发布，就会实时通知订阅者。通过这种方式可以实现分布式系统配置的集中式管理和实时动态更新。显然，通过 Watch 机制可以实现。

应用在启动时，主动从 Etcd 获取一次配置信息，同时，在 Etcd 节点上注册一个 Watcher 并等待，以后每次配置有更新，Etcd 都会实时通知订阅者，以此达到获取最新配置信息的目的。

**应用场景 3： 分布式锁**

前面已经提及，Etcd 支持 Revision 机制，那么对于同一个 lock，即便有多个客户端争夺（本质上就是 put(lockName, value) 操作），Revision 机制可以保证它们的 Revision 编号有序且唯一，那么，客户端只要根据 Revision 的大小顺序就可以确定获得锁的先后顺序，从而很容易实现公平锁。

**应用场景 4： 集群监控与 Leader 竞选**

* **集群监控**：通过 Etcd 的 Watch 机制，当某个 key 消失或变动时，Watcher 会第一时间发现并告知用户。节点可以为 key 设置租约 (TTL)，比如每隔 30s 向 Etcd 发送一次心跳续约，使代表该节点的 key 保持存活，一旦节点故障，续约停止，对应的 key 将失效删除。如此，通过 Watch 机制就可以第一时间检测到各节点的健康状态，以完成集群的监控要求。
* **Leader 竞选**：使用分布式锁，可以很好的实现 Leader 竞选（抢锁成功的成为 Leader）。Leader 应用的经典场景是在搜索系统中建立全量索引。如果每个机器分别进行索引的建立，不仅耗时，而且不能保证索引的一致性。通过在 Etcd 实现的锁机制竞选 Leader，由 Leader 进行索引计算，再将计算结果分发到其它节点。

5.5 Etcd 的部署方法

Etcd 集群的部署比较简单，官方提供了详细的说明（[点击查看](https://coreos.com/etcd/docs/latest/demo.html)），网上也有很多博客，因此，本文就不在赘述了。另外，Etcd 提供了 Windows 版本，对于只有 Windows 环境的读者，可以放心了（[官方下载地址](https://github.com/etcd-io/etcd/releases)）。

**第二部分：基于 Etcd 的分布式锁实现原理及方案**

Etcd 的最新版本已经提供了支持分布式锁的基础接口（[官网说明](https://coreos.com/etcd/docs/latest/demo.html)），但本文并不局限于此。

本节将介绍两条实现分布式锁的技术路线：

* 从分布式锁的原理出发，结合 Etcd 的特性，洞见分布式锁的实现细节；
* 基于 Etcd 提供的分布式锁基础接口进行封装，实现分布式锁。

两条路线差距甚远，建议读者先看路线 1，以便了解 Etcd 实现分布式锁的细节。

6. 为什么选择 Etcd

官网介绍：Etcd 是一个分布式的，可靠的 key-value 存储系统，主要用于存储分布式系统中的关键数据。初见之下，Etcd 与一个 NoSQL 的数据库系统有几分相似，但作为数据库绝非 Etcd 所长，其读写性能远不如 MongoDB、Redis 等 key-value 存储系统。“让专业的人做专业的事！”

Ectd 作为一个高可用的键值存储系统，有很多典型的应用场景，本章将介绍 Etcd 的优秀实践之一：分布式锁。

6.1 Etcd 优点

目前，可实现分布式锁的开源软件还是比较多的，其中应用最广泛、大家最熟悉的应该就是 ZooKeeper，此外还有数据库、Redis、Chubby 等。但若从读写性能、可靠性、可用性、安全性和复杂度等方面综合考量，作为后起之秀的 Etcd 无疑是其中的 “佼佼者” 。它完全媲美业界 “名宿” ZooKeeper，在有些方面，Etcd 甚至超越了 ZooKeeper，如 Etcd 采用的 Raft 协议就要比 ZooKeeper 采用的 Zab 协议简单、易理解。

Etcd 作为 coreos 开源项目，有以下的特点：

* 简单：使用 Go 语言编写，部署简单；支持 curl 方式的用户 API （HTTP+JSON），使用简单；开源 Java 客户端使用简单；
* 安全：可选 SSL 证书认证；
* 快速：在保证强一致性的同时，读写性能优秀，详情可查看 [官方提供的 benchmark 数据](https://coreos.com/etcd/docs/3.2.17/op-guide/performance.html) ；
* 可靠：采用 Raft 算法实现分布式系统数据的高可用性和强一致性。

6.2 分布式锁的基本原理

分布式环境下，多台机器上多个进程对同一个共享资源（数据、文件等）进行操作，如果不做互斥，就有可能出现“余额扣成负数”，或者“商品超卖”的情况。为了解决这个问题，需要分布式锁服务。

首先，来看一下分布式锁应该具备哪些条件：

* 互斥性：在任意时刻，对于同一个锁，只有一个客户端能持有，从而保证只有一个客户端能够操作同一个共享资源；
* 安全性：即不会形成死锁，当一个客户端在持有锁的期间崩溃而没有主动解锁的情况下，其持有的锁也能够被正确释放，并保证后续其它客户端能加锁；
* 可用性：当提供锁服务的节点发生宕机等不可恢复性故障时，“热备” 节点能够接替故障的节点继续提供服务，并保证自身持有的数据与故障节点一致。
* 对称性：对于任意一个锁，其加锁和解锁必须是同一个客户端，即，客户端 A 不能把客户端 B 加的锁给解了。

6.3 Etcd 实现分布式锁的基础

Etcd 的高可用性、强一致性不必多说，前面章节中已经阐明，本节主要介绍 Etcd 支持的以下机制：Watch 机制、Lease 机制、Revision 机制和 Prefix 机制，正是这些机制赋予了 Etcd 实现分布式锁的能力。

* **Lease机制**：即租约机制（TTL，Time To Live），Etcd 可以为存储的 key-value 对设置租约，当租约到期，key-value 将失效删除；同时也支持续约，通过客户端可以在租约到期之前续约，以避免 key-value 对过期失效。Lease 机制可以保证分布式锁的安全性，为锁对应的 key 配置租约，即使锁的持有者因故障而不能主动释放锁，锁也会因租约到期而自动释放。
* **Revision机制**：每个 key 带有一个 Revision 号，每进行一次事务加一，因此它是全局唯一的，如初始值为 0，进行一次 put(key, value)，key 的 Revision 变为 1；同样的操作，再进行一次，Revision 变为 2；换成 key1 进行 put(key1, value) 操作，Revision 将变为 3。这种机制有一个作用：通过 Revision 的大小就可以知道进行写操作的顺序。在实现分布式锁时，多个客户端同时抢锁，根据 Revision 号大小依次获得锁，可以避免 “羊群效应” （也称 “惊群效应”），实现公平锁。
* **Prefix机制**：即前缀机制，也称目录机制。例如，一个名为 /mylock 的锁，两个争抢它的客户端进行写操作，实际写入的 key 分别为：key1="/mylock/UUID1"，key2="/mylock/UUID2"，其中，UUID 表示全局唯一的 ID，确保两个 key 的唯一性。很显然，写操作都会成功，但返回的 Revision 不一样，那么，如何判断谁获得了锁呢？通过前缀 /mylock 查询，返回包含两个 key-value 对的的 KeyValue 列表，同时也包含它们的 Revision，通过 Revision 大小，客户端可以判断自己是否获得锁，如果抢锁失败，则等待锁释放（对应的 key 被删除或者租约过期），然后再判断自己是否可以获得锁；
* **Watch机制**：即监听机制，Watch 机制支持 Watch 某个固定的 key，也支持 Watch 一个范围（前缀机制），当被 Watch 的 key 或范围发生变化，客户端将收到通知；在实现分布式锁时，如果抢锁失败，可通过 Prefix 机制返回的 KeyValue 列表获得 Revision 比自己小且相差最小的 key（称为 pre-key），对 pre-key 进行监听，因为只有它释放锁，自己才能获得锁，如果 Watch 到 pre-key 的 DELETE 事件，则说明 pre-key 已经释放，自己已经持有锁。

7. Etcd Java 客户端——Jetcd

Jetcd 是 Etcd 的 Java 客户端，为 Etcd 的特性提供了丰富的接口，使用起来非常方便。不过，需要注意的是：Jetcd 支持 Etcd V3 版本（Etcd 较早的版本是 V2），运行环境需 Java 1.8 及以上。

Git 地址：<https://github.com/etcd-io/jetcd>

7.1 Jetcd 基本用法

首先创建一个 maven 工程，导入 Jetcd 依赖。目前，最新的版本为 0.0.2：

<dependency>

<groupId>io.etcd</groupId>

<artifactId>jetcd-core</artifactId>

<version>${jetcd-version}</version>

</dependency>

**1. Key-Value 客户端：**

Etcd 作为一个 key-value 存储系统，Key-Value 客户端是最基本的客户端，进行 put、get、delete 操作。

*// 创建客户端，本例中Etcd服务端为单机模式*

Client client = Client.builder().endpoints("http://localhost:2379").build();

KV kvClient = client.getKVClient();

*// 对String类型的key-value进行类型转换*

ByteSequence key = ByteSequence.fromString("test\_key");

ByteSequence **value** = ByteSequence.fromString("test\_value");

*// put操作，等待操作完成*

kvClient.put(key, **value**).**get**();

*// get操作，等待操作完成*

CompletableFuture<GetResponse> getFuture = kvClient.**get**(key);

GetResponse response = getFuture.**get**();

*// delete操作，等待操作完成*

kvClient.delete(key).**get**();

**2. Lease 客户端：**

Lease 客户端，即租约客户端，用于创建租约、续约、解约，以及检索租约的详情，如租约绑定的键值等信息。

*// 创建客户端，本例中Etcd服务端为单机模式*

Client client = Client.builder().endpoints("http://localhost:2379").build();

*// 创建Lease客户端*

Lease leaseClient = client.getLeaseClient();

*// 创建一个60s的租约，等待完成，超时设置阈值30s*

Long leaseId = leaseClient.grant(60).**get**(30, TimeUnit.SECONDS).getID();

*// 使指定的租约永久有效，即永久租约*

leaseClient.keepAlive(leaseId);

*// 续约一次*

leaseClient.keepAliveOnce(leaseId);

*// 解除租约，绑定该租约的键值将被删除*

leaseClient.revoke(leaseId);

*// 检索指定ID对应的租约的详细信息*

LeaseTimeToLiveResponse lTRes = leaseClient.timeToLive(leaseId, LeaseOption.newBuilder().withAttachedKeys().build()).**get**();

**3. Watch 客户端：**

监听客户端，可为 key 或者目录（前缀机制）创建 Watcher，Watcher 可以监听 key 的事件（put、delete 等），如果事件发生，可以通知客户端，客户端采取某些措施。

*// 创建客户端，本例中Etcd服务端为单机模式*

Client client = Client.builder().endpoints("http://localhost:2379").build();

*// 对String类型的key进行类型转换*

ByteSequence key = ByteSequence.fromString("test\_key");

*// 创建Watch客户端*

Watch watchClient = client.getWatchClient();

*// 为key创建一个Watcher*

Watcher watcher = watch.watch(key);

*// 开始listen，如果监听的key有事件(如删除、更新等)发生则返回*

WatchResponse response = null;

**try**

{

response = watcher.listen();

}

**catch** (InterruptedException e)

{

System.**out**.println("Failed to listen key:"+e);

}

**if**(response != null)

{

List<WatchEvent> eventlist = res.getEvents();

*// 解析eventlist，判断是否为自己关注的事件，作进一步操作*

*// To do something*

}

**4. Cluster 客户端：**

为了保障高可用性，实际应用中 Etcd 应工作于集群模式下，集群节点数量为大于 3 的奇数，为了灵活的管理集群，Jetcd 提供了集群管理客户端，支持获取集群成员列表、增加成员、删除成员等操作。

*// 创建客户端，本例中Etcd服务端为单机模式*

Client client = Client.builder().endpoints("http://localhost:2379").build();

*// 创建Cluster客户端*

Cluster clusterClient = client.getClusterClient();

*// 获取集群成员列表*

List<Member> list = clusterClient.listMember().**get**().getMembers();

*// 向集群中添加成员*

String tempAddr = "http://localhost:2389";

List<String> peerAddrs = **new** ArrayList<String>();

peerAddrs.**add**(tempAddr);

clusterClient.addMember(peerAddrs);

*// 根据成员ID删除成员*

**long** memberID = 8170086329776576179L;

clusterClient.removeMember(memberID);

*// 更新*

clusterClient.updateMember(memberID, peerAddrs);

**5. Maintenance 客户端：**

Etcd 本质上是一个 key-value 存储系统，在一系列的 put、get、delete 及 compact 操作后，集群节点可能出现键空间不足等告警，通过 Maintenance 客户端，可以进行告警查询、告警解除、整理压缩碎片等操作。

*// 创建客户端，本例中Etcd服务端为单机模式*

Client client = Client.builder().endpoints("http://localhost:2379").build();

*// 创建一个Maintenance 客户端*

Maintenance maintClient = client.getMaintenanceClient();

*// 获取指定节点的状态,对res做进一步解析可得节点状态详情*

StatusResponse res = maintClient.statusMember("http://localhost:2379").**get**();

*// 对指定的节点进行碎片整理，在压缩键空间之后，后端数据库可能呈现内部碎片，需进行整理*

*// 整理碎片是一个“昂贵”的操作，应避免同时对多个节点进行整理*

maintClient.defragmentMember("http://localhost:2379").**get**();

*// 获取所有活跃状态的键空间告警*

List<AlarmMember> alarmList = maintClient.listAlarms().**get**().getAlarms();

*// 解除指定键空间的告警*

maintClient.alarmDisarm(alarmList.**get**(0));

8. Etcd 实现分布式锁：路线一

通过前面章节的铺垫，对于如何用 Etcd 实现分布式锁，相信读者已经心中有数，理解了原理，实现反而是简单的。在此，我给出一个 Demo 供读者参考。

8.1 基于 Etcd 的分布式锁的业务流程

下面描述使用 Etcd 实现分布式锁的业务流程，假设对某个共享资源设置的锁名为：/lock/mylock

**步骤 1: 准备**

客户端连接 Etcd，以 /lock/mylock 为前缀创建全局唯一的 key，假设第一个客户端对应的 key="/lock/mylock/UUID1"，第二个为 key="/lock/mylock/UUID2"；客户端分别为自己的 key 创建租约 - Lease，租约的长度根据业务耗时确定，假设为 15s；

**步骤 2: 创建定时任务作为租约的“心跳”**

当一个客户端持有锁期间，其它客户端只能等待，为了避免等待期间租约失效，客户端需创建一个定时任务作为“心跳”进行续约。此外，如果持有锁期间客户端崩溃，心跳停止，key 将因租约到期而被删除，从而锁释放，避免死锁。

**步骤 3: 客户端将自己全局唯一的 key 写入 Etcd**

进行 put 操作，将步骤 1 中创建的 key 绑定租约写入 Etcd，根据 Etcd 的 Revision 机制，假设两个客户端 put 操作返回的 Revision 分别为 1、2，客户端需记录 Revision 用以接下来判断自己是否获得锁。

**步骤 4: 客户端判断是否获得锁**

客户端以前缀 /lock/mylock 读取 keyValue 列表（keyValue 中带有 key 对应的 Revision），判断自己 key 的 Revision 是否为当前列表中最小的，如果是则认为获得锁；否则监听列表中前一个 Revision 比自己小的 key 的删除事件，一旦监听到删除事件或者因租约失效而删除的事件，则自己获得锁。

**步骤 5: 执行业务**

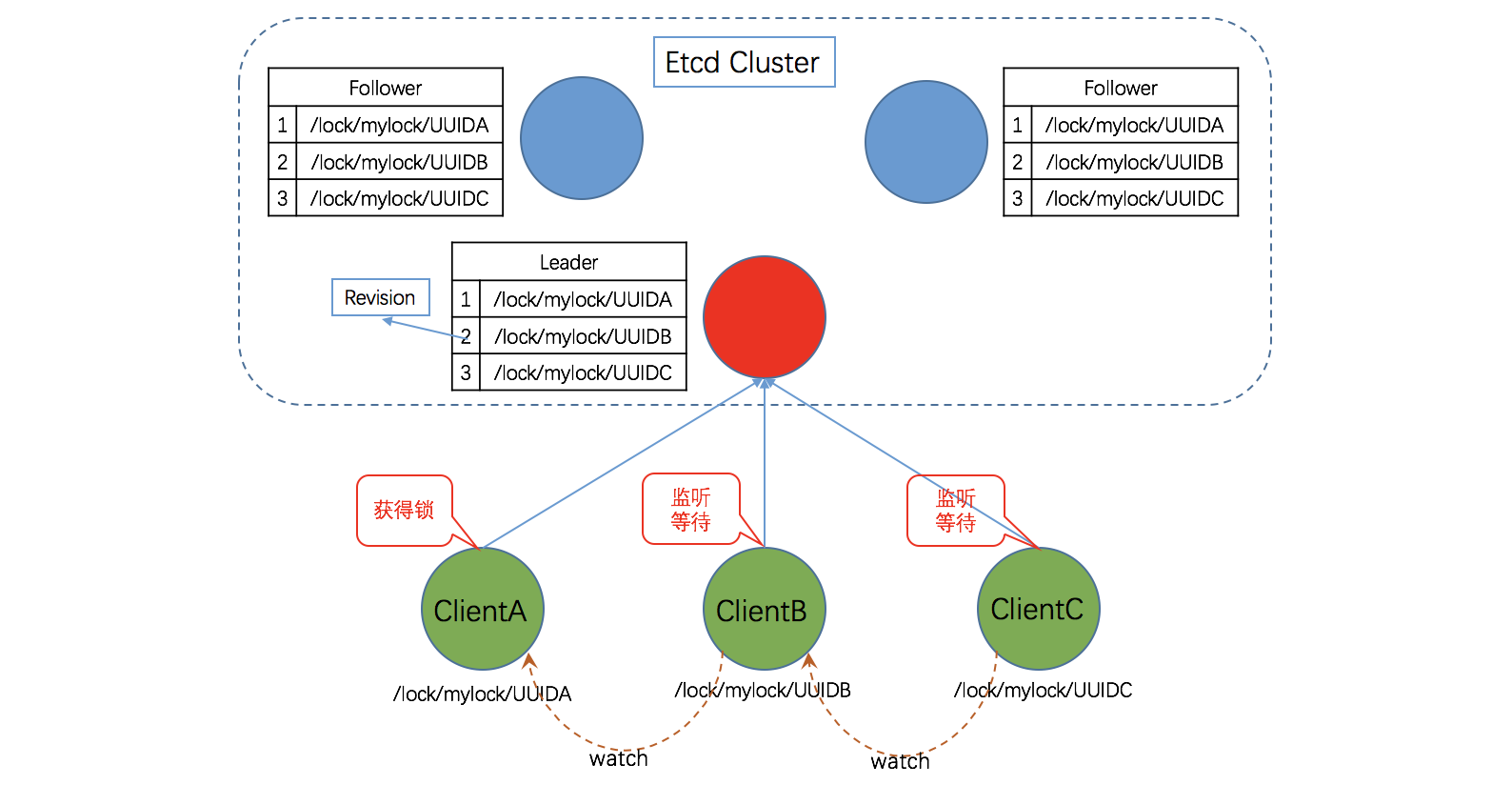
获得锁后，操作共享资源，执行业务代码。

**步骤 6: 释放锁**

完成业务流程后，删除对应的key释放锁。

8.2 基于 Etcd 的分布式锁的原理图

根据上一节中介绍的业务流程，基于 Etcd 的分布式锁示意图如下。



业务流程图：

<https://blog.csdn.net/koflance/article/details/78616206>

8.3 基于 Etcd 实现分布式锁的客户端 Demo

**import** java.util.List;

**import** java.util.concurrent.ExecutionException;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.ScheduledExecutorService;

**import** java.util.concurrent.TimeUnit;

**import** java.util.concurrent.TimeoutException;

**import** com.coreos.jetcd.Client;

**import** com.coreos.jetcd.KV;

**import** com.coreos.jetcd.Lease;

**import** com.coreos.jetcd.Watch.Watcher;

**import** com.coreos.jetcd.options.GetOption;

**import** com.coreos.jetcd.options.GetOption.SortTarget;

**import** com.coreos.jetcd.options.PutOption;

**import** com.coreos.jetcd.watch.WatchEvent;

**import** com.coreos.jetcd.watch.WatchResponse;

**import** com.coreos.jetcd.data.ByteSequence;

**import** com.coreos.jetcd.data.KeyValue;

**import** com.coreos.jetcd.kv.PutResponse;

**import** java.util.UUID;

*/\*\**

*\* Etcd 客户端代码，用多个线程“抢锁”模拟分布式系统中，多个进程“抢锁”*

*\**

*\*/*

**public** **class** **EtcdClient**

{

**public** **static** **void** **main**(String[] args) **throws** InterruptedException, ExecutionException,

TimeoutException, ClassNotFoundException

{

*// 创建Etcd客户端，Etcd服务端为单机模式*

Client client = Client.builder().endpoints("http://localhost:2379").build();

*// 对于某共享资源制定的锁名*

String lockName = "/lock/mylock";

*// 模拟分布式场景下，多个进程“抢锁”*

**for** (**int** i = 0; i < 3; i++)

{

**new** MyThread(lockName, client).start();

}

}

*/\*\**

*\* 加锁方法，返回值为加锁操作中实际存储于Etcd中的key，即：lockName+UUID，*

*\* 根据返回的key，可删除存储于Etcd中的键值对，达到释放锁的目的。*

*\**

*\* @param lockName*

*\* @param client*

*\* @param leaseId*

*\* @return*

*\*/*

**public** **static** String **lock**(String lockName, Client client, **long** leaseId)

{

*// lockName作为实际存储在Etcd的中的key的前缀，后缀是一个全局唯一的ID，从而确保：对于同一个锁，不同进程存储的key具有相同的前缀，不同的后缀*

StringBuffer strBufOfRealKey = **new** StringBuffer();

strBufOfRealKey.append(lockName);

strBufOfRealKey.append("/");

strBufOfRealKey.append(UUID.randomUUID().toString());

*// 加锁操作实际上是一个put操作，每一次put操作都会使revision增加1，因此，对于任何一次操作，这都是唯一的。(get,delete也一样)*

*// 可以通过revision的大小确定进行抢锁操作的时序，先进行抢锁的，revision较小，后面依次增加。*

*// 用于记录自己“抢锁”的Revision，初始值为0L*

**long** revisionOfMyself = 0L;

KV kvClient = client.getKVClient();

*// lock，尝试加锁，加锁只关注key，value不为空即可。*

*// 注意：这里没有考虑可靠性和重试机制，实际应用中应考虑put操作而重试*

**try**

{

PutResponse putResponse = kvClient

.put(ByteSequence.fromString(strBufOfRealKey.toString()),

ByteSequence.fromString("value"),

PutOption.newBuilder().withLeaseId(leaseId).build())

.get(10, TimeUnit.SECONDS);

*// 获取自己加锁操作的Revision号*

revisionOfMyself = putResponse.getHeader().getRevision();

}

**catch** (InterruptedException | ExecutionException | TimeoutException e1)

{

System.out.println("[error]: lock operation failed:" + e1);

}

**try**

{

*// lockName作为前缀，取出所有键值对，并且根据Revision进行升序排列，版本号小的在前*

List<KeyValue> kvList = kvClient.get(ByteSequence.fromString(lockName),

GetOption.newBuilder().withPrefix(ByteSequence.fromString(lockName))

.withSortField(SortTarget.MOD).build())

.get().getKvs();

*// 如果自己的版本号最小，则表明自己持有锁成功，否则进入监听流程，等待锁释放*

**if** (revisionOfMyself == kvList.get(0).getModRevision())

{

System.out.println("[lock]: lock successfully. [revision]:" + revisionOfMyself);

*// 加锁成功，返回实际存储于Etcd中的key*

**return** strBufOfRealKey.toString();

}

**else**

{

*// 记录自己加锁操作的前一个加锁操作的索引，因为只有前一个加锁操作完成并释放，自己才能获得锁*

**int** preIndex = 0;

**for** (**int** index = 0; index < kvList.size(); index++)

{

**if** (kvList.get(index).getModRevision() == revisionOfMyself)

{

preIndex = index - 1;*// 前一个加锁操作，故比自己的索引小1*

}

}

*// 根据索引，获得前一个加锁操作对应的key*

ByteSequence preKeyBS = kvList.get(preIndex).getKey();

*// 创建一个Watcher，用于监听前一个key*

Watcher watcher = client.getWatchClient().watch(preKeyBS);

WatchResponse res = **null**;

*// 监听前一个key，将处于阻塞状态，直到前一个key发生delete事件*

*// 需要注意的是，一个key对应的事件不只有delete，不过，对于分布式锁来说，除了加锁就是释放锁*

*// 因此，这里只要监听到事件，必然是delete事件或者key因租约过期而失效删除，结果都是锁被释放*

**try**

{

System.out.println("[lock]: keep waiting until the lock is released.");

res = watcher.listen();

}

**catch** (InterruptedException e)

{

System.out.println("[error]: failed to listen key.");

}

*// 为了便于读者理解，此处写一点冗余代码，判断监听事件是否为DELETE，即释放锁*

List<WatchEvent> eventlist = res.getEvents();

**for** (WatchEvent event : eventlist)

{

*// 如果监听到DELETE事件，说明前一个加锁操作完成并已经释放，自己获得锁，返回*

**if** (event.getEventType().toString().equals("DELETE"))

{

System.out.println("[lock]: lock successfully. [revision]:" + revisionOfMyself);

**return** strBufOfRealKey.toString();

}

}

}

}

**catch** (InterruptedException | ExecutionException e)

{

System.out.println("[error]: lock operation failed:" + e);

}

**return** strBufOfRealKey.toString();

}

*/\*\**

*\* 释放锁方法，本质上就是删除实际存储于Etcd中的key*

*\**

*\* @param lockName*

*\* @param client*

*\*/*

**public** **static** **void** **unLock**(String realLockName, Client client)

{

**try**

{

client.getKVClient().delete(ByteSequence.fromString(realLockName)).get(10,

TimeUnit.SECONDS);

System.out.println("[unLock]: unlock successfully.[lockName]:" + realLockName);

}

**catch** (InterruptedException | ExecutionException | TimeoutException e)

{

System.out.println("[error]: unlock failed：" + e);

}

}

*/\*\**

*\* 自定义一个线程类，模拟分布式场景下多个进程 "抢锁"*

*\*/*

**public** **static** **class** **MyThread** **extends** **Thread**

{

**private** String lockName;

**private** Client client;

MyThread(String lockName, Client client)

{

**this**.client = client;

**this**.lockName = lockName;

}

**@Override**

**public** **void** **run**()

{

*// 创建一个租约，有效期15s*

Lease leaseClient = client.getLeaseClient();

Long leaseId = **null**;

**try**

{

leaseId = leaseClient.grant(15).get(10, TimeUnit.SECONDS).getID();

}

**catch** (InterruptedException | ExecutionException | TimeoutException e1)

{

System.out.println("[error]: create lease failed:" + e1);

**return**;

}

*// 创建一个定时任务作为“心跳”，保证等待锁释放期间，租约不失效；*

*// 同时，一旦客户端发生故障，心跳便会中断，锁也会应租约过期而被动释放，避免死锁*

ScheduledExecutorService service = Executors.newSingleThreadScheduledExecutor();

*// 续约心跳为12s，仅作为举例*

service.scheduleAtFixedRate(**new** KeepAliveTask(leaseClient, leaseId), 1, 12, TimeUnit.SECONDS);

*// 1. try to lock*

String realLoclName = lock(lockName, client, leaseId);

*// 2. to do something*

**try**

{

Thread.sleep(6000);

}

**catch** (InterruptedException e2)

{

System.out.println("[error]:" + e2);

}

*// 3. unlock*

service.shutdown();*// 关闭续约的定时任务*

unLock(realLoclName, client);

}

}

*/\*\**

*\* 在等待其它客户端释放锁期间，通过心跳续约，保证自己的key-value不会失效*

*\**

*\*/*

**public** **static** **class** **KeepAliveTask** **implements** **Runnable**

{

**private** Lease leaseClient;

**private** **long** leaseId;

KeepAliveTask(Lease leaseClient, **long** leaseId)

{

**this**.leaseClient = leaseClient;

**this**.leaseId = leaseId;

}

**@Override**

**public** **void** **run**()

{

leaseClient.keepAliveOnce(leaseId);

}

}

}

Demo 运行结果如下：

[lock]: lock successfully. [revision]:44

[lock]: keep waiting until the lock is released.

[lock]: keep waiting until the lock is released.

[unLock]: unlock successfully.

[lock]: lock successfully. [revision]:45

[unLock]: unlock successfully.

[lock]: lock successfully. [revision]:46

[unLock]: unlock successfully.

9. Etcd 实现分布式锁：路线二

Etcd 最新的版本提供了支持分布式锁的基础接口，其本质就是将第 3 节（路线一）中介绍的实现细节进行了封装。单从使用的角度来看，这是非常有益的，大大降低了分布式锁的实现难度。但，与此同时，简化的接口也无形中为用户理解内部原理设置了屏障。

9.1 Etcd 提供的分布式锁基础接口

在介绍 Jetcd 提供的 Lock 客户端之前，我们先用 Etcd 官方提供的 Go 语言客户端（etcdctl）验证一下分布式锁的实现原理。

解压官方提供的 Etcd 安装包，里面有两个可执行文件：etcd 和 etcdctl，其中 etcd 是服务端，etcdctl 是客户端。在服务端启动的前提下，执行以下命令验证分布式锁原理：

* (1) 分别开启两个窗口，进入 etcdctl 所在目录，执行以下命令，显式指定 API 版本为 V3，老版本 V2 不支持分布式锁接口。

export ETCDCTL\_API=3

* (2) 分别在两个窗口执行相同的加锁命令：

./etcdctl.exe **lock** mylock

* (3) 可以观察到，只有一个加锁成功，并返回了实际存储与 Etcd 中key值:

**$** ./etcdctl.exe lock mylock

mylock/694d65eb367c7ec4

* (4) 在加锁成功的窗口执行命令：ctrl+c，释放锁；与此同时，另一个窗口加锁成功，如下所示：

**$** ./etcdctl.exe lock mylock

mylock/694d65eb367c7ec8

很明显，同样的锁名 - mylock，两个客户端分别进行加锁操作，实际存储于 Etcd 中的 key 并不是 mylock，而是以 mylock 为前缀，分别加了一个全局唯一的 ID。是不是和 “路线一” 中介绍的原理一致？

9.2 Etcd Java 客户端 Jetcd 提供的 Lock 客户端

作为 Etcd 的 Java 客户端，Jetcd 全面支持 Etcd 的 V3 接口，其中分布式锁相关的接口如下。看上去很简单，但事实上存在一个问题：租约没有心跳机制，在等待其它客户端释放锁期间，自己的租约存在过期的风险。鉴于此，需要进行改造。抛砖引玉，我 9.3 节中提供了一个 Demo 供读者参考。

*// 创建客户端，本例中Etcd服务端为单机模式*

Client client = Client.builder().endpoints("http://localhost:2379").build();

*// 创建Lock客户端*

Lock lockClient = client.getLockClient();

*// 创建Lease客户端，并创建一个有效期为30s的租约*

Lease leaseClient = client.getLeaseClient()；

**long** leaseId = leaseClient.grant(30).**get**().getID();

*// 加、解锁操作*

**try**

{

*// 调用lock接口，加锁，并绑定租约*

lockClient.**lock**(ByteSequence.fromString("lockName"), leaseId).**get**();

*// 调用unlock接口，解锁*

lockClient.unlock(ByteSequence.fromString(lockName)).**get**();

}

**catch** (InterruptedException | ExecutionException e1)

{

System.**out**.println("[error]: lock failed:" + e1);

}

9.3 基于 Etcd 的 lock 接口实现分布式锁的 Demo

**第一部分：分布式锁实现**

**import** java.util.concurrent.ExecutionException;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.ScheduledExecutorService;

**import** java.util.concurrent.TimeUnit;

**import** com.coreos.jetcd.Client;

**import** com.coreos.jetcd.Lease;

**import** com.coreos.jetcd.Lock;

**import** com.coreos.jetcd.data.ByteSequence;

**public** **class** **DistributedLock**

{

**private** **static** DistributedLock lockProvider = **null**;

**private** **static** Object mutex = **new** Object();

**private** Client client;

**private** Lock lockClient;

**private** Lease leaseClient;

**private** **DistributedLock**()

{

**super**();

*// 创建Etcd客户端，本例中Etcd集群只有一个节点*

**this**.client = Client.builder().endpoints("http://localhost:2379").build();

**this**.lockClient = client.getLockClient();

**this**.leaseClient = client.getLeaseClient();

}

**public** **static** DistributedLock **getInstance**()

{

**synchronized** (mutex)

{

**if** (**null** == lockProvider)

{

lockProvider = **new** DistributedLock();

}

}

**return** lockProvider;

}

*/\*\**

*\* 加锁操作，需要注意的是，本例中没有加入重试机制，加锁失败将直接返回。*

*\**

*\* @param lockName: 针对某一共享资源(数据、文件等)制定的锁名*

*\* @param TTL : Time To Live，租约有效期，一旦客户端崩溃，可在租约到期后自动释放锁*

*\* @return LockResult*

*\*/*

**public** LockResult **lock**(String lockName, **long** TTL)

{

LockResult lockResult = **new** LockResult();

*/\*1.准备阶段\*/*

*// 创建一个定时任务作为“心跳”，保证等待锁释放期间，租约不失效；*

*// 同时，一旦客户端发生故障，心跳便会停止，锁也会因租约过期而被动释放，避免死锁*

ScheduledExecutorService service = Executors.newSingleThreadScheduledExecutor();

*// 初始化返回值lockResult*

lockResult.setIsLockSuccess(**false**);

lockResult.setService(service);

*// 记录租约ID，初始值设为 0L*

Long leaseId = 0L;

*/\*2.创建租约\*/*

*// 创建一个租约，租约有效期为TTL，实际应用中根据具体业务确定。*

**try**

{

leaseId = leaseClient.grant(TTL).get().getID();

lockResult.setLeaseId(leaseId);

*// 启动定时任务续约，心跳周期和初次启动延时计算公式如下，可根据实际业务制定。*

**long** period = TTL - TTL / 5;

service.scheduleAtFixedRate(**new** KeepAliveTask(leaseClient, leaseId), period, period,

TimeUnit.SECONDS);

}

**catch** (InterruptedException | ExecutionException e)

{

System.out.println("[error]: Create lease failed:" + e);

**return** lockResult;

}

System.out.println("[lock]: start to lock." + Thread.currentThread().getName());

*/\*3.加锁操作\*/*

*// 执行加锁操作，并为锁对应的key绑定租约*

**try**

{

lockClient.lock(ByteSequence.fromString(lockName), leaseId).get();

}

**catch** (InterruptedException | ExecutionException e1)

{

System.out.println("[error]: lock failed:" + e1);

**return** lockResult;

}

System.out.println("[lock]: lock successfully." + Thread.currentThread().getName());

lockResult.setIsLockSuccess(**true**);

**return** lockResult;

}

*/\*\**

*\* 解锁操作，释放锁、关闭定时任务、解除租约*

*\**

*\* @param lockName:锁名*

*\* @param lockResult:加锁操作返回的结果*

*\*/*

**public** **void** **unLock**(String lockName, LockResult lockResult)

{

System.out.println("[unlock]: start to unlock." + Thread.currentThread().getName());

**try**

{

*// 释放锁*

lockClient.unlock(ByteSequence.fromString(lockName)).get();

*// 关闭定时任务*

lockResult.getService().shutdown();

*// 删除租约*

**if** (lockResult.getLeaseId() != 0L)

{

leaseClient.revoke(lockResult.getLeaseId());

}

}

**catch** (InterruptedException | ExecutionException e)

{

System.out.println("[error]: unlock failed: " + e);

}

System.out.println("[unlock]: unlock successfully." + Thread.currentThread().getName());

}

*/\*\**

*\* 在等待其它客户端释放锁期间，通过心跳续约，保证自己的锁对应租约不会失效*

*\**

*\*/*

**public** **static** **class** **KeepAliveTask** **implements** **Runnable**

{

**private** Lease leaseClient;

**private** **long** leaseId;

KeepAliveTask(Lease leaseClient, **long** leaseId)

{

**this**.leaseClient = leaseClient;

**this**.leaseId = leaseId;

}

**@Override**

**public** **void** **run**()

{

*// 续约一次*

leaseClient.keepAliveOnce(leaseId);

}

}

*/\*\**

*\* 该class用于描述加锁的结果，同时携带解锁操作所需参数*

*\**

*\*/*

**public** **static** **class** **LockResult**

{

**private** **boolean** isLockSuccess;

**private** **long** leaseId;

**private** ScheduledExecutorService service;

LockResult()

{

**super**();

}

**public** **void** **setIsLockSuccess**(**boolean** isLockSuccess)

{

**this**.isLockSuccess = isLockSuccess;

}

**public** **void** **setLeaseId**(**long** leaseId)

{

**this**.leaseId = leaseId;

}

**public** **void** **setService**(ScheduledExecutorService service)

{

**this**.service = service;

}

**public** **boolean** **getIsLockSuccess**()

{

**return** **this**.isLockSuccess;

}

**public** **long** **getLeaseId**()

{

**return** **this**.leaseId;

}

**public** ScheduledExecutorService **getService**()

{

**return** **this**.service;

}

}

}

**第二部分：测试代码**

**public** **class** **DistributedLockTest**

{

**public** **static** **void** **main**(String[] args)

{

*// 模拟分布式场景下，多个进程 “抢锁”*

**for** (**int** i = 0; i < 5; i++)

{

**new** MyThread().start();

}

}

**public** **static** **class** **MyThread** **extends** **Thread**

{

**@Override**

**public** **void** **run**()

{

String lockName = "/lock/mylock";

*// 1. 加锁*

LockResult lockResult = DistributedLock.getInstance().lock(lockName, 30);

*// 2. 执行业务*

**if** (lockResult.getIsLockSuccess())

{

*// 获得锁后，执行业务，用sleep方法模拟.*

**try**

{

Thread.sleep(10000);

}

**catch** (InterruptedException e)

{

System.out.println("[error]:" + e);

}

}

*// 3. 解锁*

DistributedLock.getInstance().unLock(lockName, lockResult);

}

}

}

**第三部分：测试结果**

[lock]: start to lock.Thread-4

[lock]: start to lock.Thread-3

[lock]: start to lock.Thread-1

[lock]: start to lock.Thread-0

[lock]: start to lock.Thread-2

[lock]: lock successfully.Thread-3

[unlock]: start to unlock.Thread-3

[unlock]: unlock successfully.Thread-3

[lock]: lock successfully.Thread-2

[unlock]: start to unlock.Thread-2

[unlock]: unlock successfully.Thread-2

[lock]: lock successfully.Thread-1

[unlock]: start to unlock.Thread-1

[unlock]: unlock successfully.Thread-1

[lock]: lock successfully.Thread-0

[unlock]: start to unlock.Thread-0

[unlock]: unlock successfully.Thread-0

[lock]: lock successfully.Thread-4

[unlock]: start to unlock.Thread-4

[unlock]: unlock successfully.Thread-4

尾声