

什么是 Paxos 协议



扫码试看/订阅

《ZooKeeper实战与源码剖析》视频课程

Paxos 算法做什么

Paxos 算法是一个一致性算法，作用是让 Asynchronous non-Byzantine Model 的分布式环境中的各个 agent 达成一致。

我打一个比方，7 个朋友要决定晚上去哪里吃饭。一致性算法就是保证要么这 7 个朋友达成一致选定一个地方去吃饭，要么因为各种异常情况达不成一致，但是不能出现一些朋友选定一个地方，另外一些朋友选定另外一个地方的情况。

Asynchronous non-Byzantine Model

一个分布式环境由若干个 agent 组成，agent 之间通过传递消息进行通讯：

- agent 以任意的速度运行，agent 可能失败和重启。但是 agent 不会出 Byzantine fault 。
- 消息需要任意长的时间进行传递，消息可能丢失，消息可能会重复。但是消息不会 corrupt 。

Paxos 算法中的 agent 角色

client :发送请求给 Paxos 算法服务。

proposer :发送 prepare 请求和 accept 请求。

acceptor :处理 prepare 请求和 accept 请求。

learner :获取一个 Paxos 算法实例决定的结果。

Paxos 算法伪代码

```
proposer
=====
propose(v):
    while not decided:
        // 阶段 1a
        choose n, unique and higher than any n seen so far
        send prepare(n) to all servers including self
        if prepare_ok(n, na, va) from majority:
            // 阶段 2a
            v' = va with highest na; choose own otherwise
            send accept(n, v') to all
            if accept_ok(n) from majority:
                send decided(v') to all

acceptor
=====
np: 承诺过或者接受过的最大提案号
na, va: 接受过的最大提案号和对应的值

// 阶段 1b
promise(n):
    if n > np
        np = n
        reply promise_ok(n, na, va)
    else
        reply promise_reject

// 阶段 2b
accept(n, v):
    if n >= np
        np = n
        na = n
        va = v
        reply accept_ok(n)
    else
        reply accept_reject
```

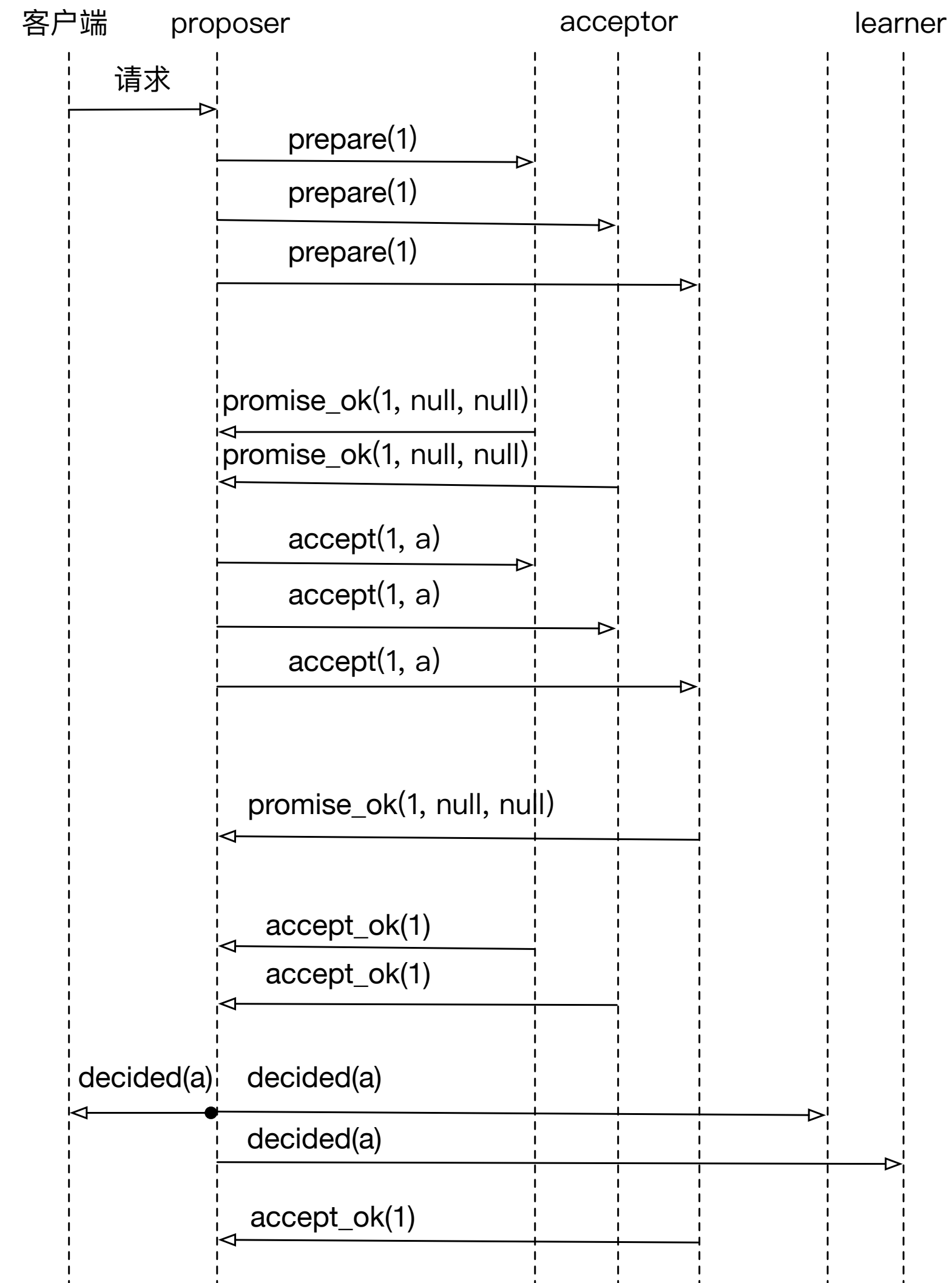

Paxos 算法描述

一个 Paxos 算法实例正常运行的前提是大多数 acceptor 正常运行。换句话说就是 Paxos 提供允许少数 acceptor 失败的容错能力。

子阶段	描述
阶段 1a 准备(prepare)	选择一个提案号码(proposal number) n ，发送 prepare(n)消息给 Paxos 集群中所有的 acceptor。这个 n 需要大于这个 proposer 以前使用过的所有提案号。
阶段 1b 承诺(promise)	如果 prepare 消息中的提案号大于 acceptor 承诺过或者接受过的最大提案号 n_p ，也就是说 acceptor 没有承诺过不小于 n 的 prepare 消息，那么把 n_p 更新为 n 。如果 acceptor 以前接受过提案，返回的消息需要包括接受过的最大提案号和对应的值。
阶段 2a 发起接受(accept)请求	一个 proposer 在接受到 Paxos 组中大多数 acceptor 的 promise_ok 消息之后，进入这个阶段。如果有 promise_ok 消息包含 n_a 和 v_a ，把 v' 设为具有最高 n_a 值的 v_a 。否则的话，proposer 自己决定 v' 。之后 acceptor 把 accept(n, v')消息发送给所有的 acceptor。
阶段 2b 处理接受请求	如果 accept 消息中的提案号大于等于 acceptor 承诺过或者接受过的最大提案号 n_p ，也就是说 acceptor 没有接受过大于 n 的 prepare 消息或者 accept 消息，acceptor 把 n_p 和 n_a 更新为 n ，把 v_a 更新为 v 。

Paxos 算法的消息流

以下是一个 Paxos 算法实例的完成的消息流：

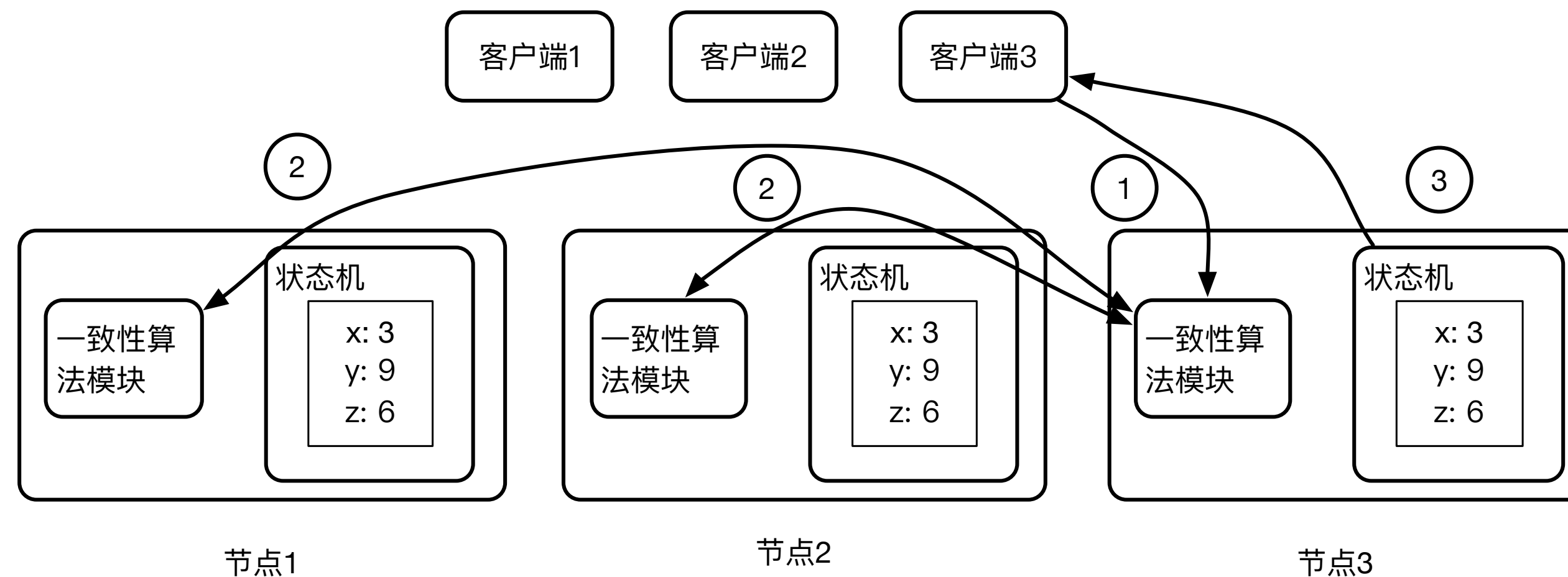


复制状态机 (Replicated State Machine)

Paxos 算法可以用来实现复制状态机的一致性算法模块。

这里面的状态机是一个 KV 系统。通过复制状态机可以把它变成一个容错的 3 节点分布式 KV 系统。下面是处理 $z=6$ 这个写操作的过程：

1. 客户端 3 发送一个 $z=6$ 请求给节点 3 的一致性算法模块。
2. 节点 3 的一致性算法发起一个算法实例。
3. 如果各个节点的一致性算法模块能一起达成一致，节点 3 把 $z=6$ 应用到它的状态机，并把结果返回给客户端 3。



MultiPaxos

基本的 Paxos 算法在决定一个值的时候需要的执行两个阶段，这涉及大量消息交换。MultiPaxos 算法的提出就是为了解决这个问题。MultiPaxos 保持一个长期的 leader，这样决定一个值只需要执行第二阶段。一个典型的 Paxos 部署通常包括奇数个服务器，每个服务器都有一个 proposer，一个 acceptor 和一个 learner。

比较 Chubby 和 ZooKeeper

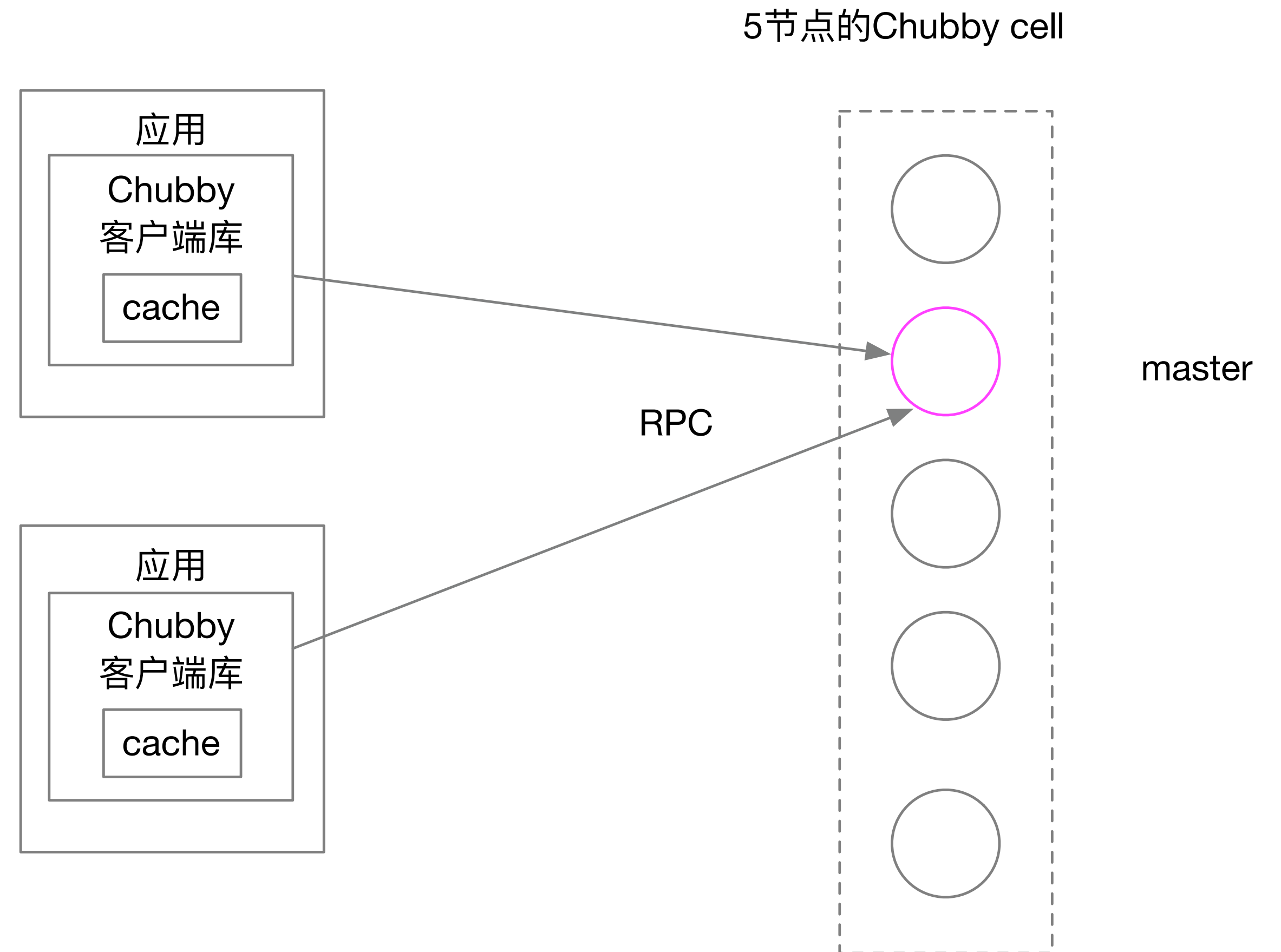
什么是 Chubby

Chubby 是一个分布式锁系统，广泛应用于 Google 的基础架构中，例如知名的 GFS 和 Bigtable 都用 Chubby 来做协同服务。

ZooKeeper 借鉴了很多 Chubby 的设计思想，所以它们之间有很多相似之处。

Chubby 系统架构

- 一个 Chubby 的集群叫作一个 cell，由多个 replica 实例组成，其中一个 replica 是整个 cell 的 master。所有的读写请求只能通过 master 来处理。
- 应用通过引入 Chubby 客户端库来使用 Chubby 服务。Chubby 客户端在和 master 建立 session 之后，通过发 RPC 给 master 来访问 Chubby 数据。
- 每个客户端维护一个保证数据一致性的 cache。

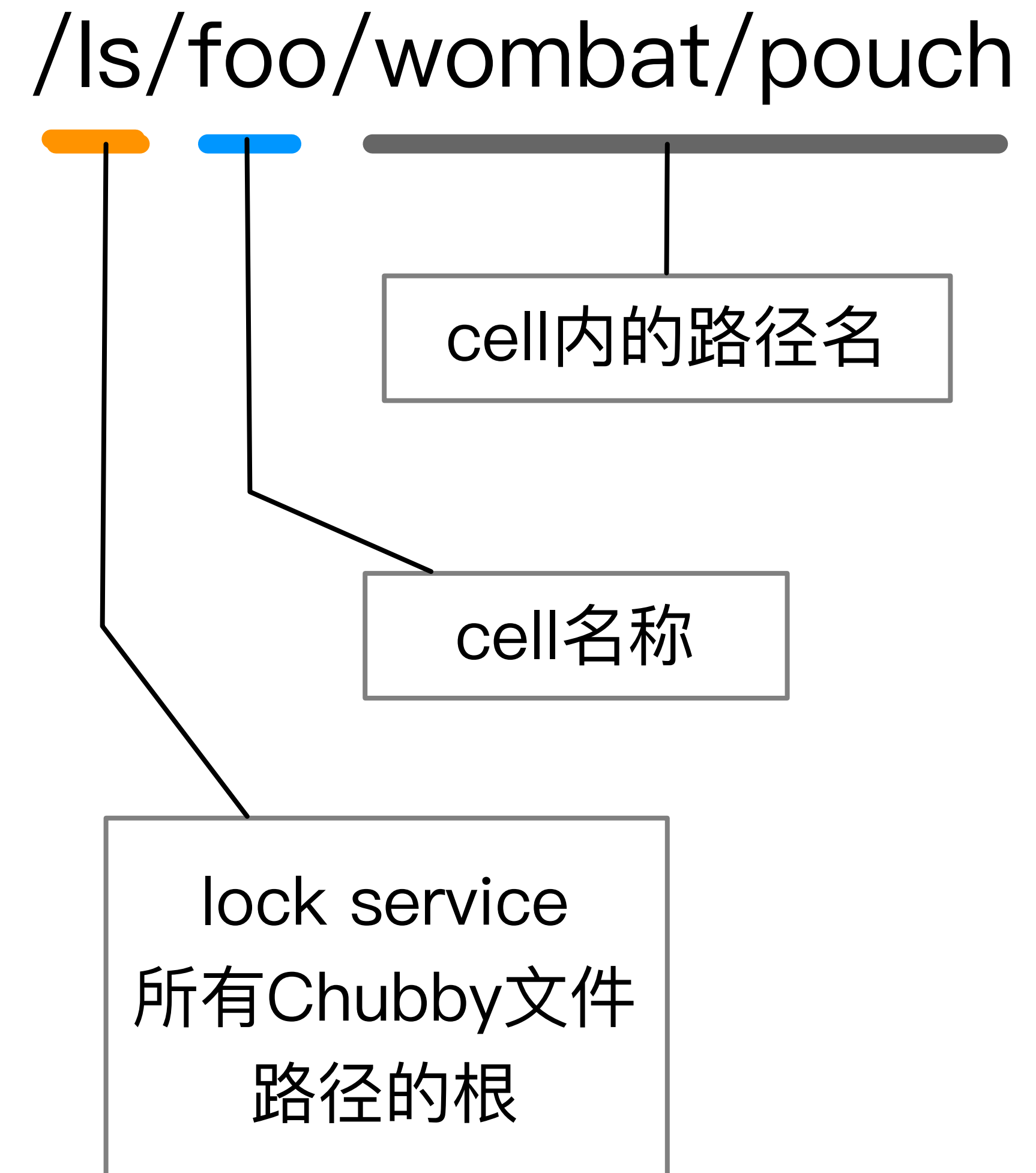


数据模型

Chubby 使用的是层次数据模型，可以看做一个简化的 UNIX 文件系统：

- Chubby 不支持部分内容的读写。
- Chubby 不支持 link 。
- Chubby 不支持依赖路径的文件权限。

不同于 ZooKeeper，Chubby 的命名空间是由多个 Chubby cell 构成的。



Chubby API

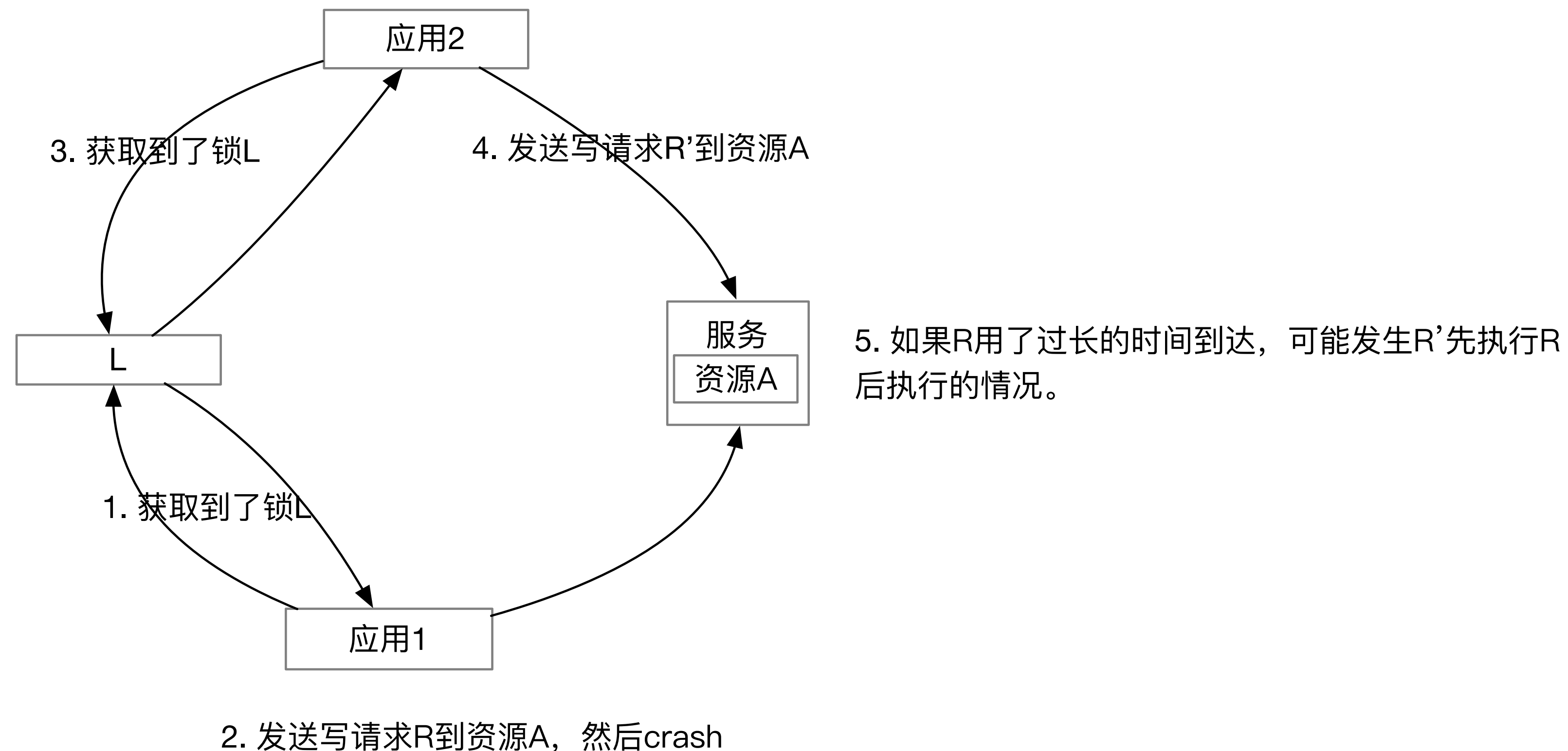
与 ZooKeeper 不同，Chubby 的 API 有一个文件句柄的概念。

- `Open()`：唯一使用路径名访问文件的 API，其他 API 都使用一个文件句柄。
- `Close()`：关闭文件句柄。
- `Poison()`：取消文件句柄上正在进行的操作，主要用于多线程场景。
- 文件操作 API：`GetContentsAndStat()`，`SetContents()`，`Delete()`。
- 锁 API：`Acquire()`，`TryAcquire()`，`Release()`，`GetSequencer()`，`SetSequencer()`，`CheckSequencer()`。

和 ZooKeeper 一样，Chubby 的 API 提供了事件通知机制、API 的异步和同步版本。

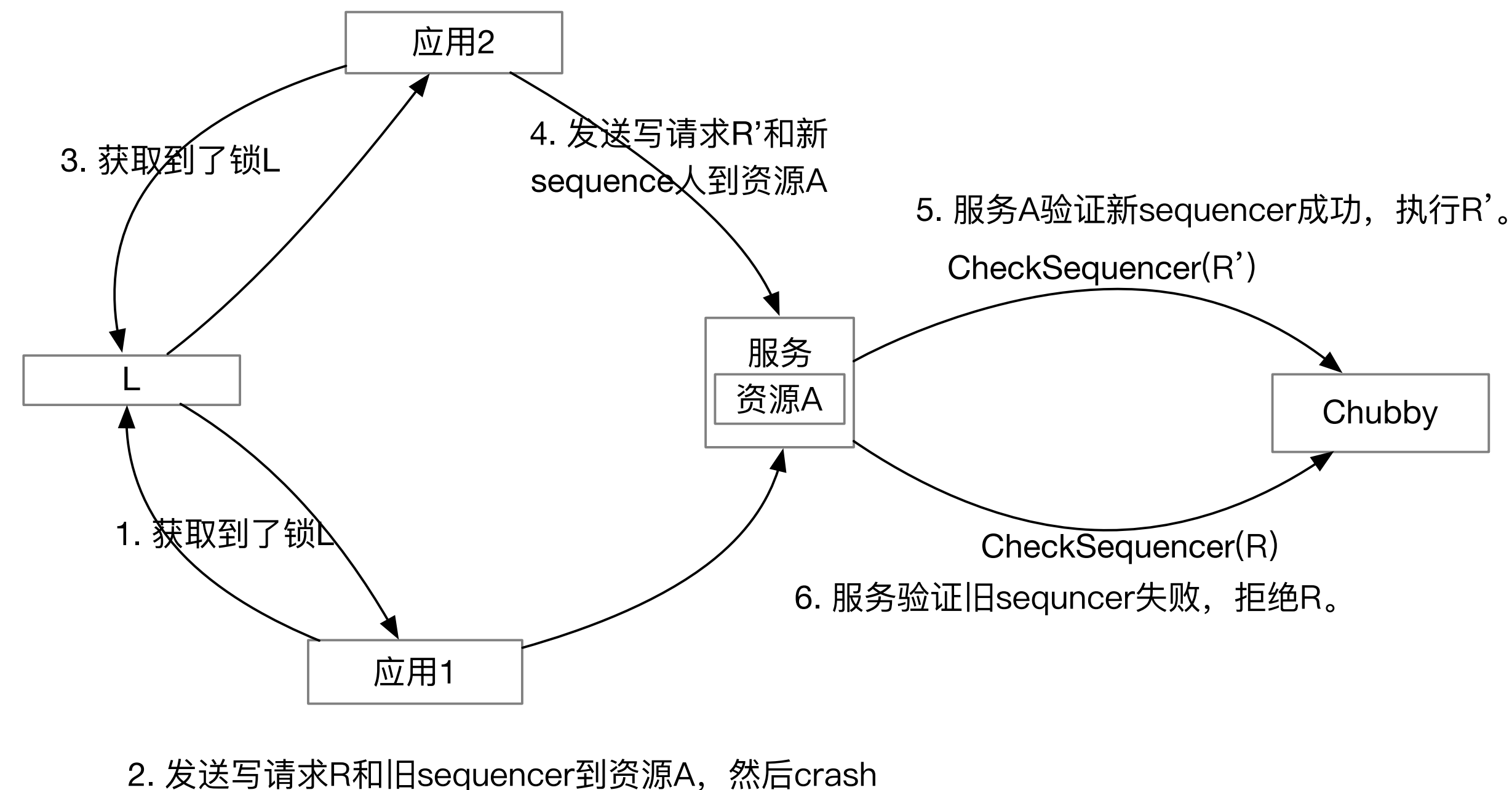
Locking

Chubby 使用 Advisory locking 。由于通信的不确定性和各种节点失败，分布式环境中的锁是非常复杂的。例如在下图所示的场景中，Chubby 锁 L 用来保护资源 A，但是会出现操作请求执行顺序颠倒的情况。



解决方案 1-Sequencer

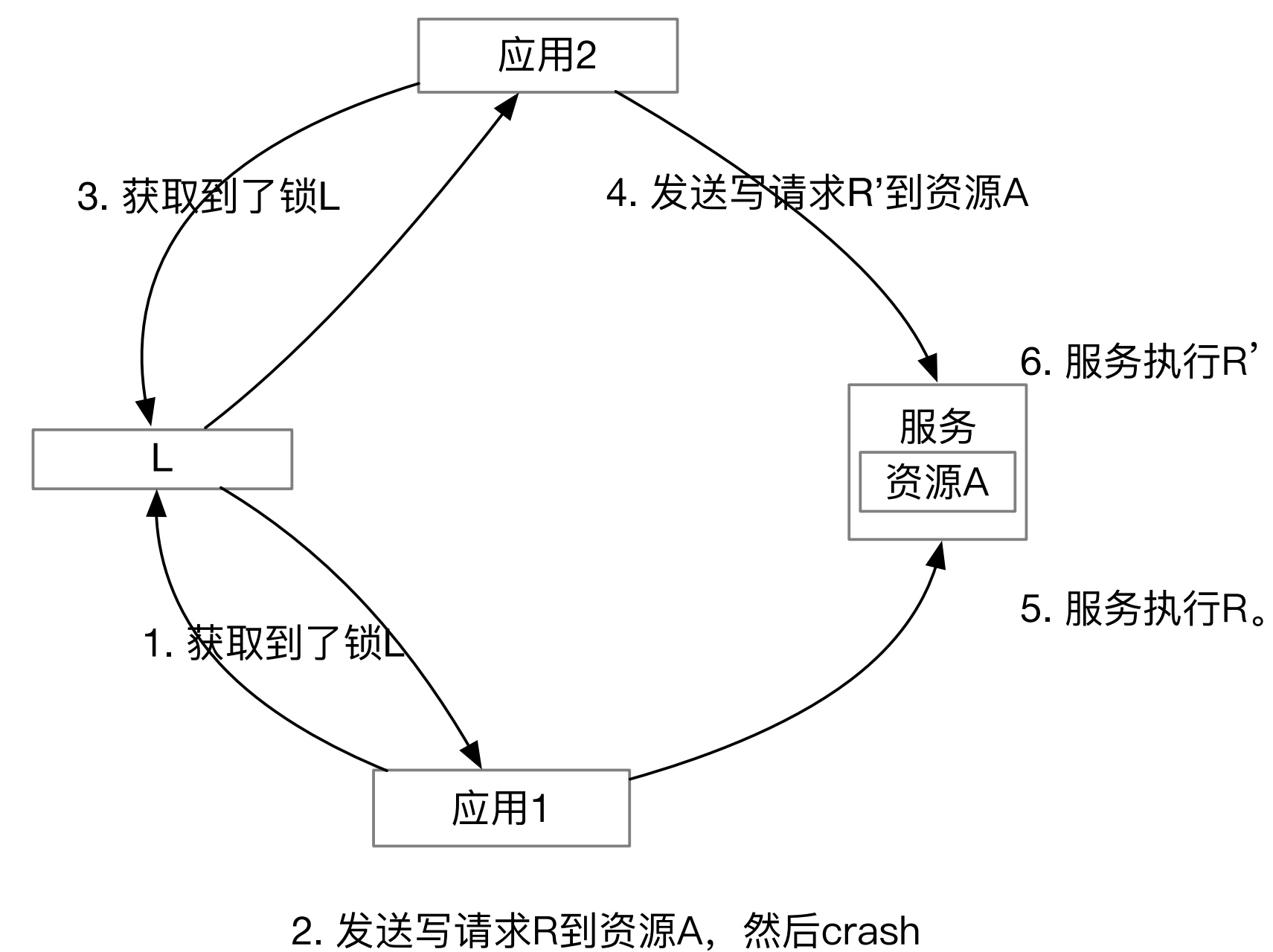
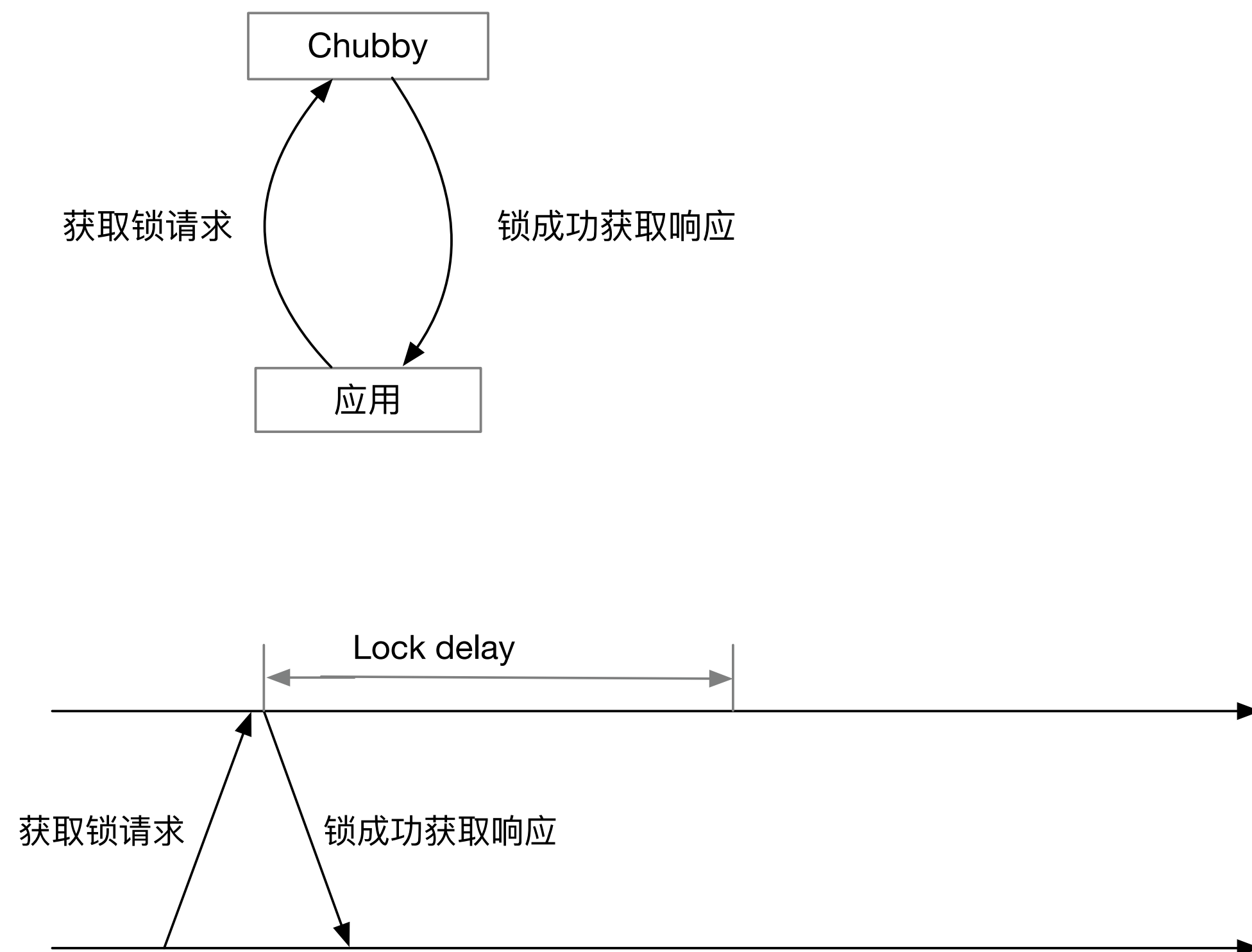
一个锁持有者可以使用 GetSequencer() API 向 Chubby 请求一个 sequencer。之后锁持有者发送访问资源的请求时，把 sequencer 一起发送给资源服务。资源服务会对 sequencer 进行验证，如果验证失败，就拒绝资源访问请求。sequencer 包含的数据有锁的名称、锁的模式和 lock generation number。



ZooKeeper 锁 recipe 里面表示锁请求的 znode 的序列号可以用作 sequencer，从而也可以实现解决方案 1。

解决方案 2-lock delay

如果锁 L 的持有者失败了或者访问不到 Chubby cell 中的节点了，Chubby 不会立刻处理对锁 L 的请求。Chubby 会等一段时间（默认1分钟）才会把锁 L 分配给其他的请求。这样也可以保证应用 2 在更晚的时刻获得到锁 L，从而在更晚的时刻发送请求 R'，保证 R 先于 R' 执行。

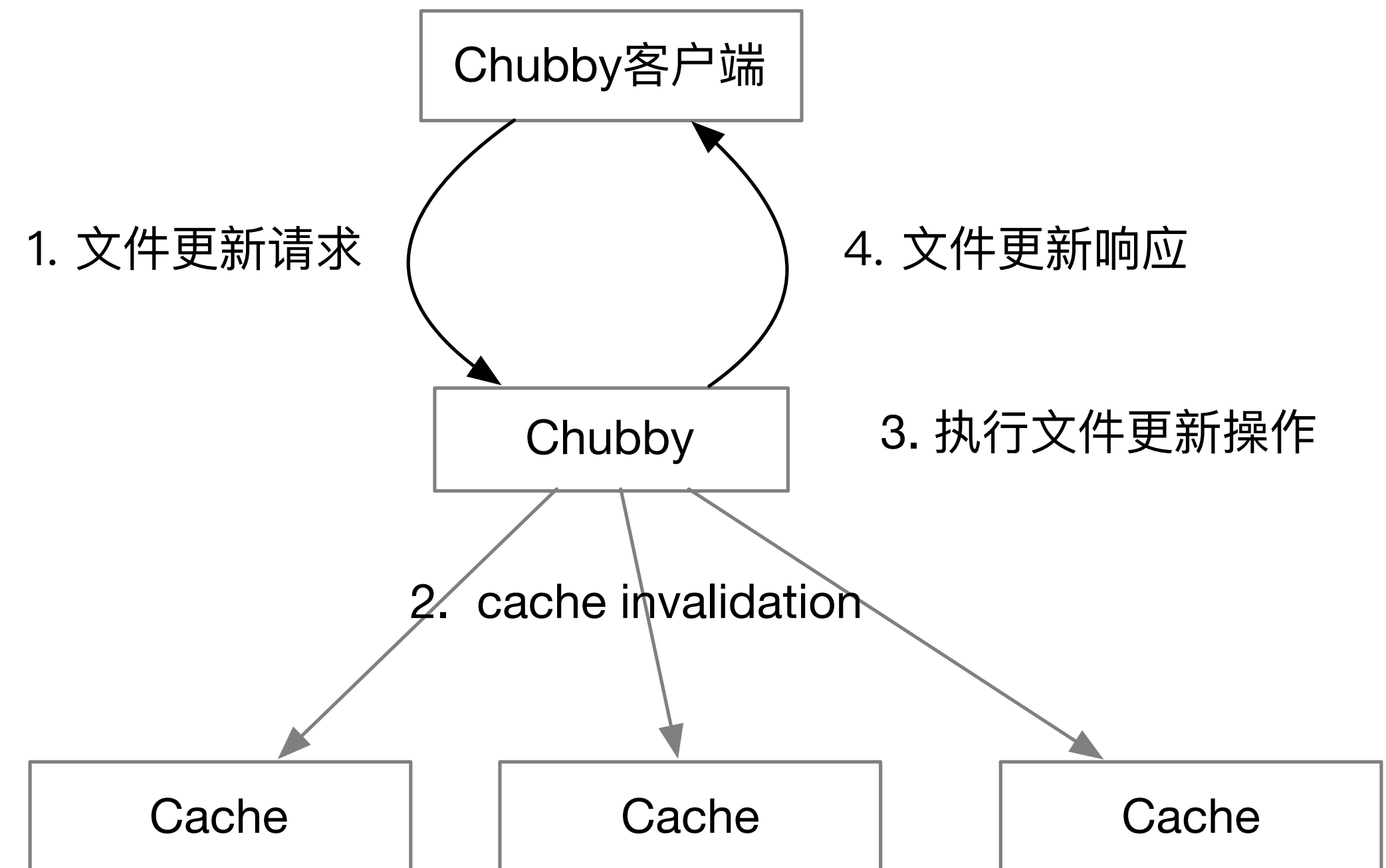


cache

Chubby 的客户端维护一个 write-through 的 cache，能保证 cache 中的数据 and Chubby 节点上的数据是一致的。master 只有在所有的 cache 失效之后（收到客户端 cache invalidation 的响应或者客户端的 lease 失效了），才进行文件更新操作。

lease 是客户端维持 session 有效的时间段。如果过了这段时间，客户端还没有 renew lease 的话，客户端停止任何 Chubby 的操作，并断开 session。

ZooKeeper 没有办法提供和 Chubby 一样的 cache。原因是 ZooKeeper 是先更新再发通知，没有办法避免 cache 中有旧数据。



相同之处

- Chubby 的文件相当于 ZooKeeper 的 znode 。Chubby 的文件和 znode 都是用来存储少量数据。
- 都只提供文件的全部读取和全部写入。
- 都提供类似 UNIX 文件系统的 API 。
- 都提供接收数据更新的通知，Chubby 提供 event 机制，ZooKeeper 提供 watch 机制。
- 它们的客户端都通过一个 session 和服务器端进行交互。
- 写操作都是通过 leader/master 来进行。
- 都支持持久性和临时性数据。
- 都使用复制状态机来做容错。

不同之处

Chubby	ZooKeeper
Chubby 内置对分布式锁的支持。	ZooKeeper 本事不提供锁，但是可以基于 ZooKeeper 的基本操作来实现锁。
读操作也必须通过 master 节点来执行。相应的，Chubby 保证的数据一致性强一些，不会有读到旧数据的问题。	读操作可以通过任意节点来执行。相应的，ZooKeeper 保证的数据一致性弱一些，有读到旧数据的问题。
Chubby 提供一个保证数据一致性的 cache。有文件句柄的概念。	ZooKeeper 不支持文件句柄，也不支持 cache，但是可以通过 watch 机制来实现 cache。但是这样实现的 cache 还是有返回旧数据的问题。
Chubby 基本操作不如 ZooKeeper 的强大。	ZooKeeper 提供更强大的基本操作，例如对顺序性节点的支持，可以用来实现更多的协同服务。
Chubby 使用 Paxos 数据一致性协议。	ZooKeeper 使用 Zab 数据一致性协议。

Raft 协议解析

什么是 Raft

Raft 是目前使用最为广泛的一致性算法。例如新的协同服务平台 etcd 和 Consul 都是使用的 Raft 算法。

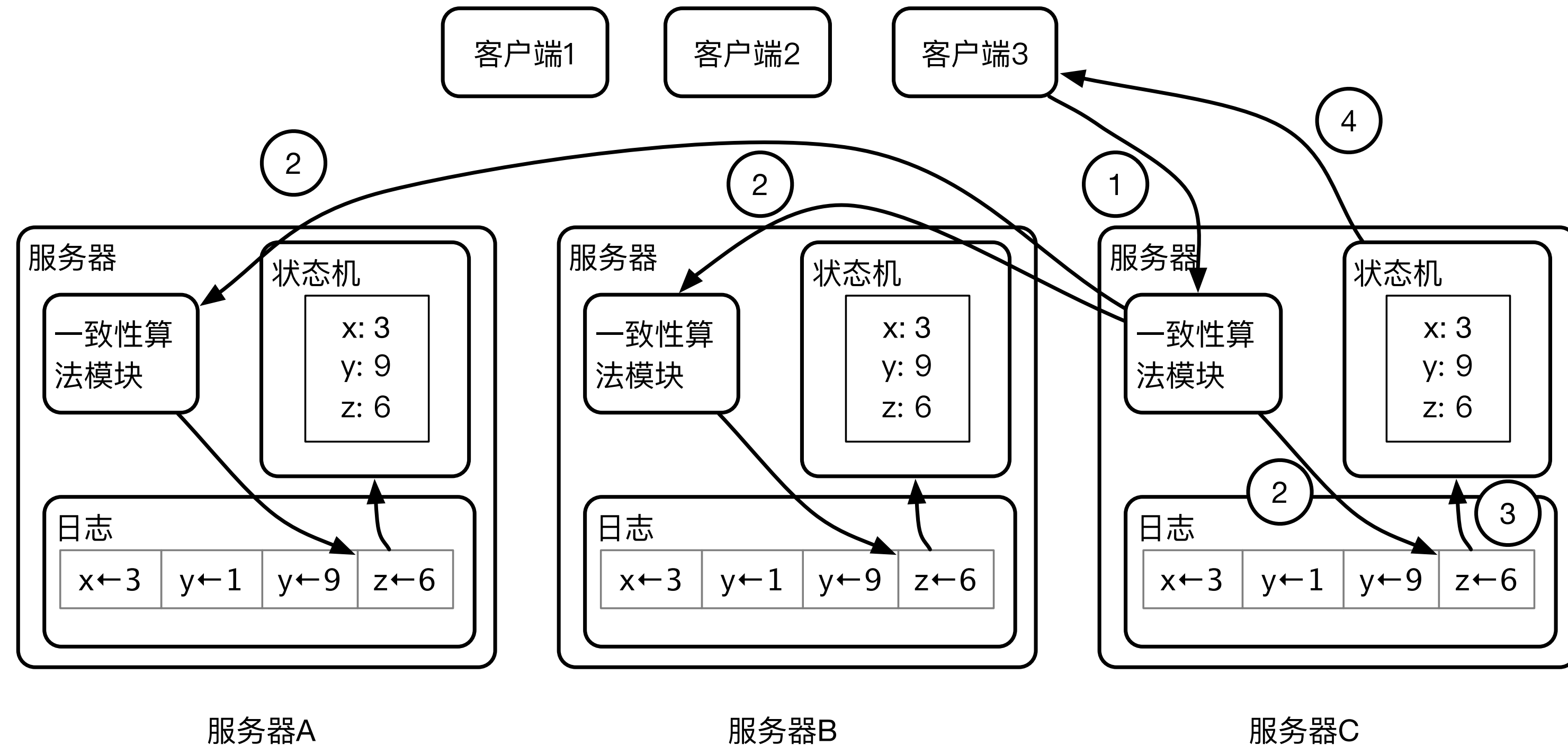
在 Raft 出现之前，广泛使用的一致性算法是 Paxos。Paxos 的基本算法解决的是如何保证单一客户端操作的一致性，完成每个操作需要至少两轮的消息交换。和 Paxos 不同，Raft 有 leader 的概念。Raft 在处理任何客户端操作之前必须选举一个 leader，选举一个 leader 需要至少一轮的消息交换。但是在选取了 leader 之后，处理每个客户端操作只需要一轮消息交换。

Raft 论文描述了一个基于 Raft 的复制状态机的整体方案，例如 Raft 论文描述了日志复制、选举机制和成员变更等这些复制状态机的各个方面。相反 Paxos 论文只是给了一个一致性算法，基于 Paxos 的系统都要自己实现这些机制。

基于 Raft 的复制状态机系统架构

右图展示了执行一条客户端写命令的过程
($z \leftarrow 6$ 表示把 6 写入 z)：

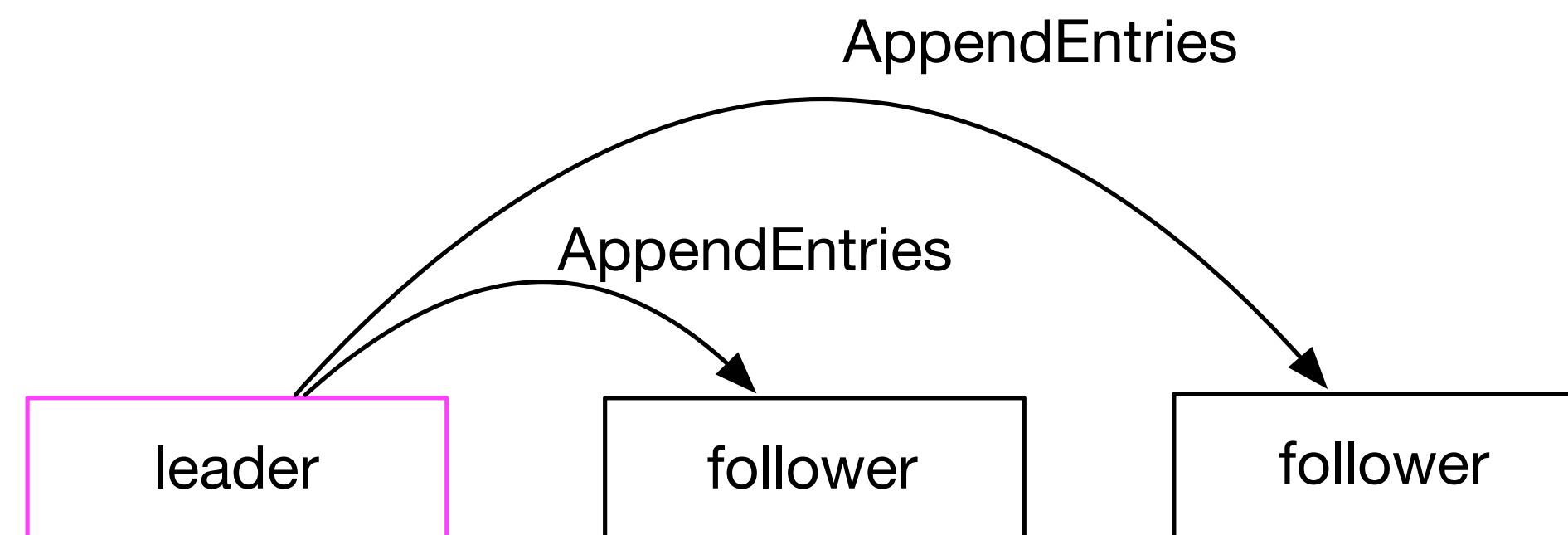
1. 客户端 3 发送一个状态机命令 $z \leftarrow 6$ 给服务器 C 的一致性算法模块。
2. 一致性算法模块把状态机命令写入服务器 C 的日志，同时发送日志复制请求给服务器 A 和服务器 B 的一致性算法模块。服务器 A 和服务器 B 的一致性算法模块在接收到日志复制请求之后，分别在给子的服务器上写入日志，然后回复服务器 C 的一致性算法模块。
3. 服务器 C 的一致性算法模块在收到服务器 A 和 B 对日志复制请求的回复之后，让状态机执行来自客户端的命令。
4. 服务器 C 的状态机把命令执行结果返回给客户端 3。



Raft 日志复制

一个 Raft 集群包括若干服务器。服务器可以处于以下三种状态：leader、follower 和 candidate。只有 leader 处理来自客户端的请求。follower 不会主动发起任何操作，只会被动的接收来自 leader 和 candidate 的请求。在正常情况下，Raft 集群中有一个 leader，其他的都是 follower。leader 在接受到一个写命令之后，为这个命令生成一个日志条目，然后进行日志复制。

leader 通过发送 AppendEntries RPC 把日志条目发送给 follower，让 follower 把接收到的日志条目写入自己的日志文件。另外 leader 也会把日志条目写入自己的日志文件。日志复制保证 Raft 集群中所有的服务器的日志最终都处于同样的状态。



Raft 的日志复制对应的就是 Paxos 的 accept 阶段，它们是很相似的。

日志条目的提交

leader 只有在客户端请求被提交以后，才可以在状态机执行客户端请求。提交意味着集群中多数服务器完成了客户端请求的日志写入，这样做是为了保证以下两点：

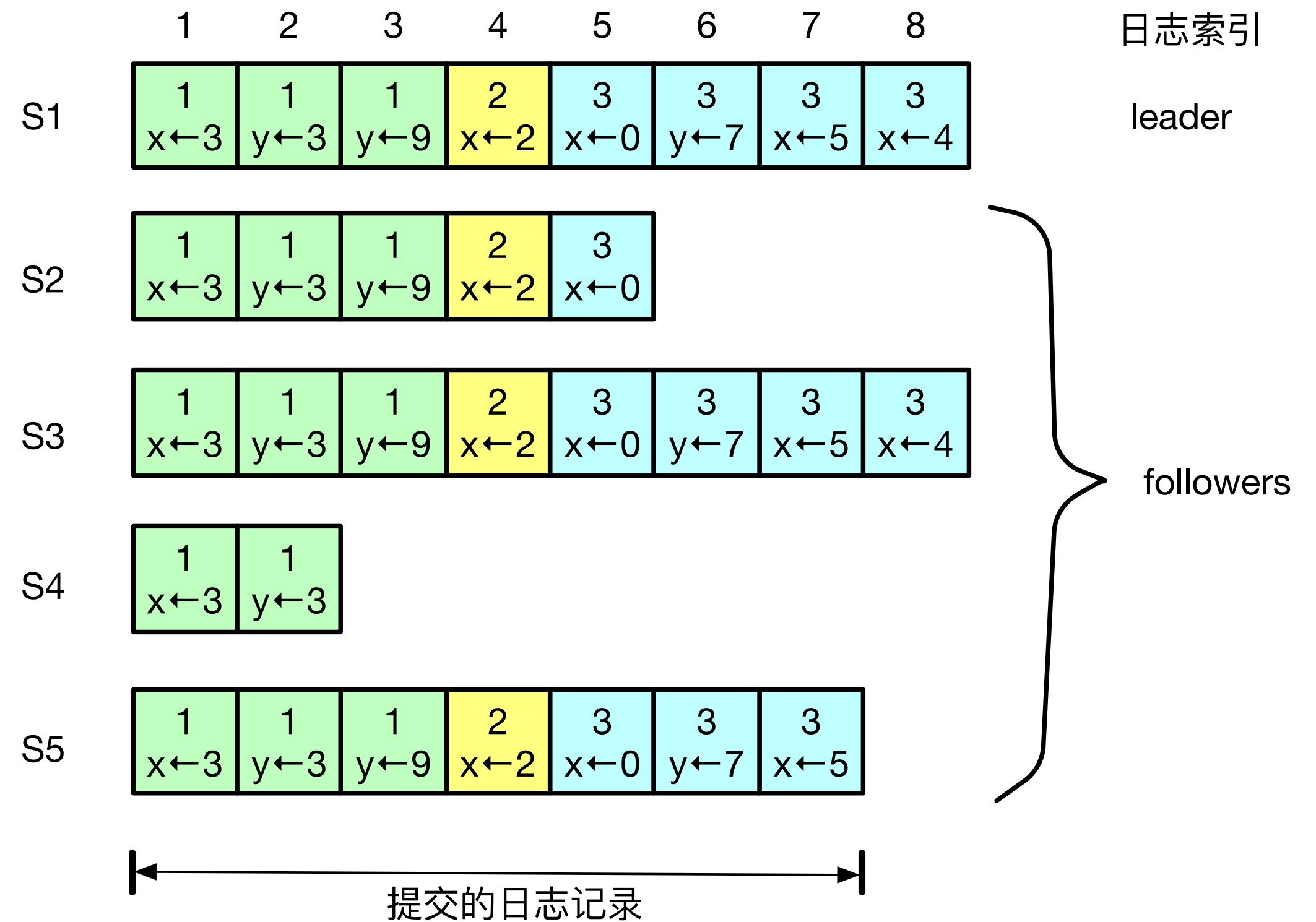
- 容错：在数量少于 Raft 服务器总数一半的 follower 失败的情况下，Raft 集群仍然可以正常处理来自客户端的请求。
- 确保重叠：一旦 Raft leader 响应了一个客户端请求，即使出现 Raft 集群中少数服务器的失败，也会有一个服务器包含所有以前提交的日志条目。

Raft 日志复制示例

右图表示的是一个包括 5 个服务器的 Raft 集群的日志格式，S1 处于 leader 状态，其他的服务器处于 follower 状态。每个日志条目由一条状态机命令和创建这条日志条目的 leader 的 term 。

每个日志条目有对应的日志索引，日志索引表示这条日志在日志中的位置。

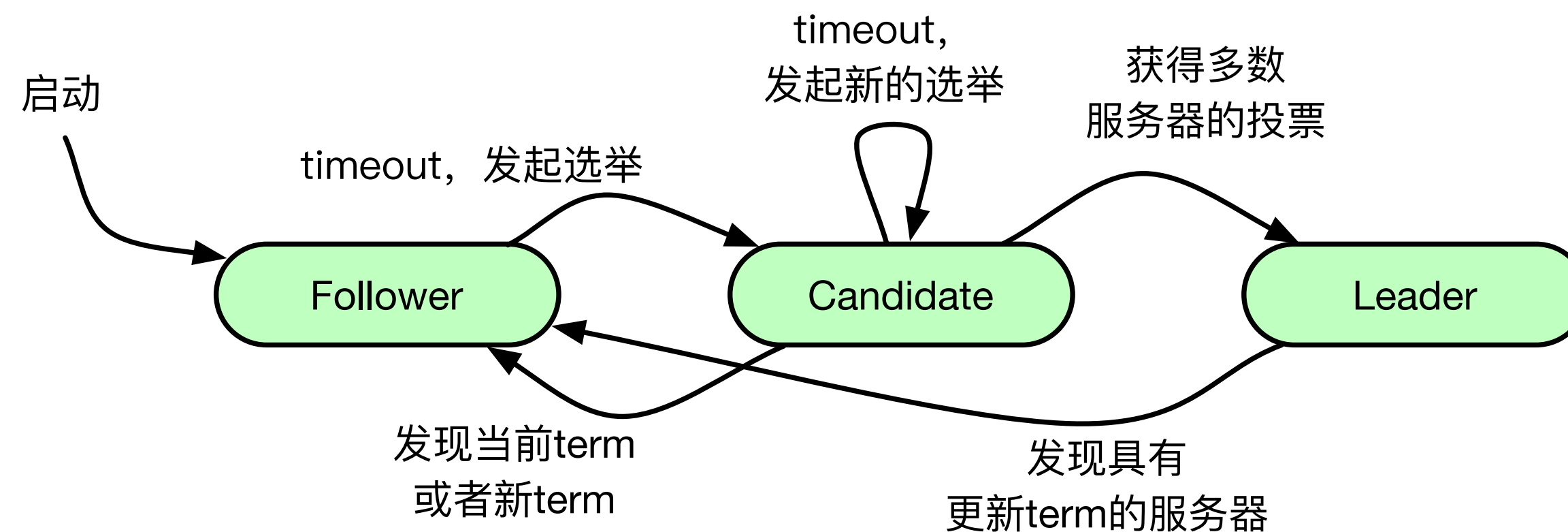
Raft 集群中提交的日志条目是 S5 上面的所有日志条目，因为这些日志条目被复制到了集群中的大多数服务器。



Raft选举算法

Raft 使用心跳机制来触发 leader 选取。一个 follower 只要能收到来自 leader 或者 candidate 的有效 RPC，就会一直处于 follower 状态。leader 在每一个 election timeout 向所有 follower 发送心跳消息来保持自己的 leader 状态。如果 follower 在一个 election timeout 周期内没有收到心跳信息，就认为目前集群中没有 leader。此时 follower 会对自己的 currentTerm 进行加一操作，并进入 candidate 状态，发起一轮投票。它会给自己投票并向其他所有的服务器发送 RequestVote RPC，然后会一直处于 candidate 状态，直到下列三种情形之一发生：

1. 这个 candidate 赢得了选举。
2. 另外一台服务器成为了 leader。
3. 一段时间之内没有服务器赢得选举。在这种情况下，candidate 会再次发起选举。



Raft 的选举对应的就是 Paxos 的 prepare 阶段，它们是很相似的。

日志匹配

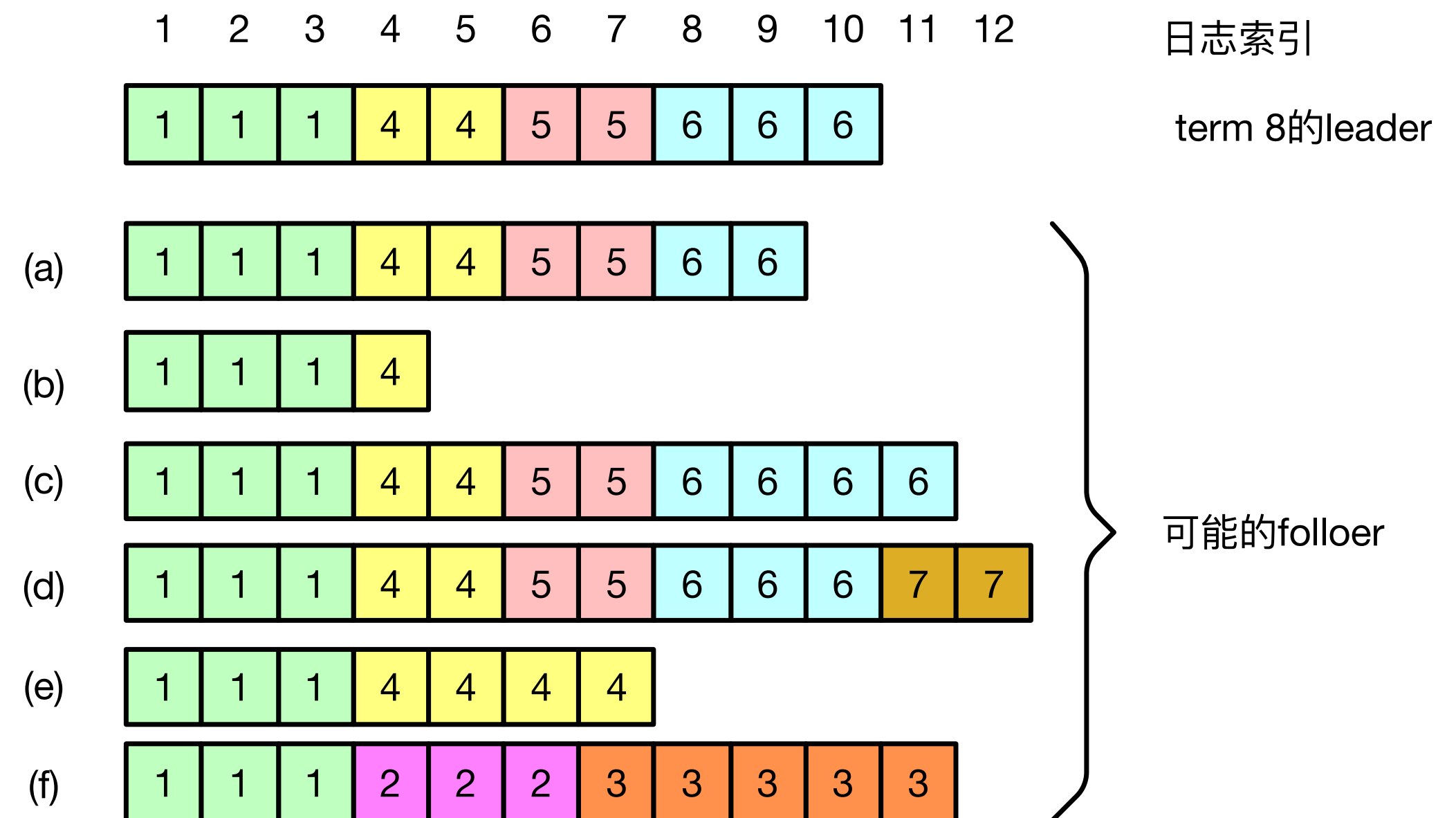
Raft 日志条目具备以下日志匹配属性：

1. 如果两个服务器上的日志条目有的相同索引和 term，那么这两个日志条目存储的状态机命令是一样的。
2. 如果两个服务器上的日志条目有的相同索引和 term，那么这两个日志从头到这个索引是完全一样的。

如果 Raft 处于复制状态，每个 follower 的日志是 leader 日志的前缀，显然日志匹配属性是满足的。在一个新的 leader 出来之后，follower 的日志可能和 leader 的日志一致，也可能处于以下三种不一致状态：

1. follower 与 leader 相比少一些日志条目。
2. follower 与 leader 相比多一些日志条目。
3. 包含（1）和（2）两种不一致情况。

例如在右图中，follower (a) 和 follower (b) 属于不一致情况 1，follower (c) 和 follower (d) 属于不一致情况 2，follower (e) 和 follower (f) 属于不一致情况 3。

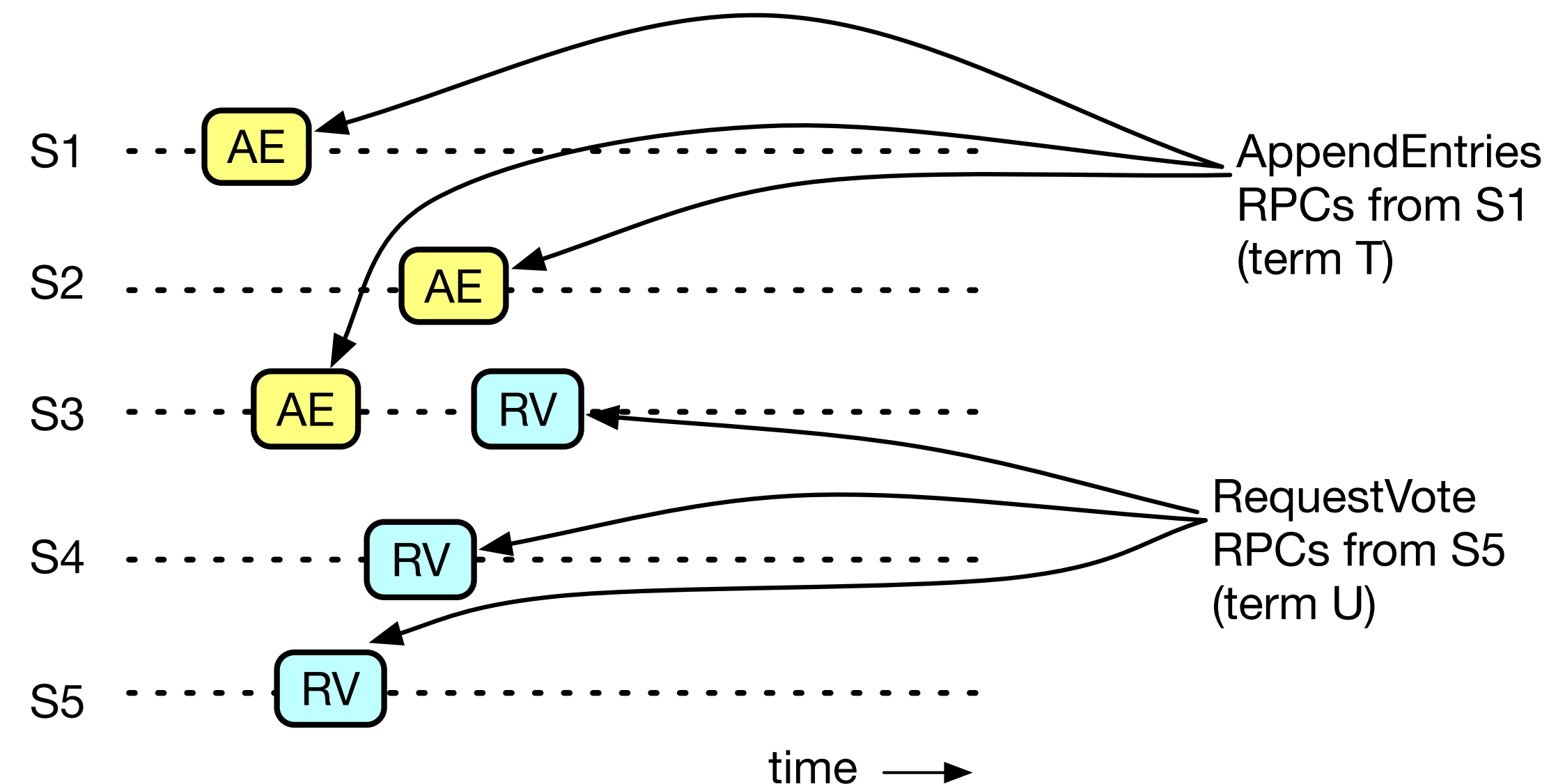


如何保证一个新 term 的 leader 保存了所有提交的日志条目

以下两点保证新 term 的 leader 保存了所有提交的日志条目：

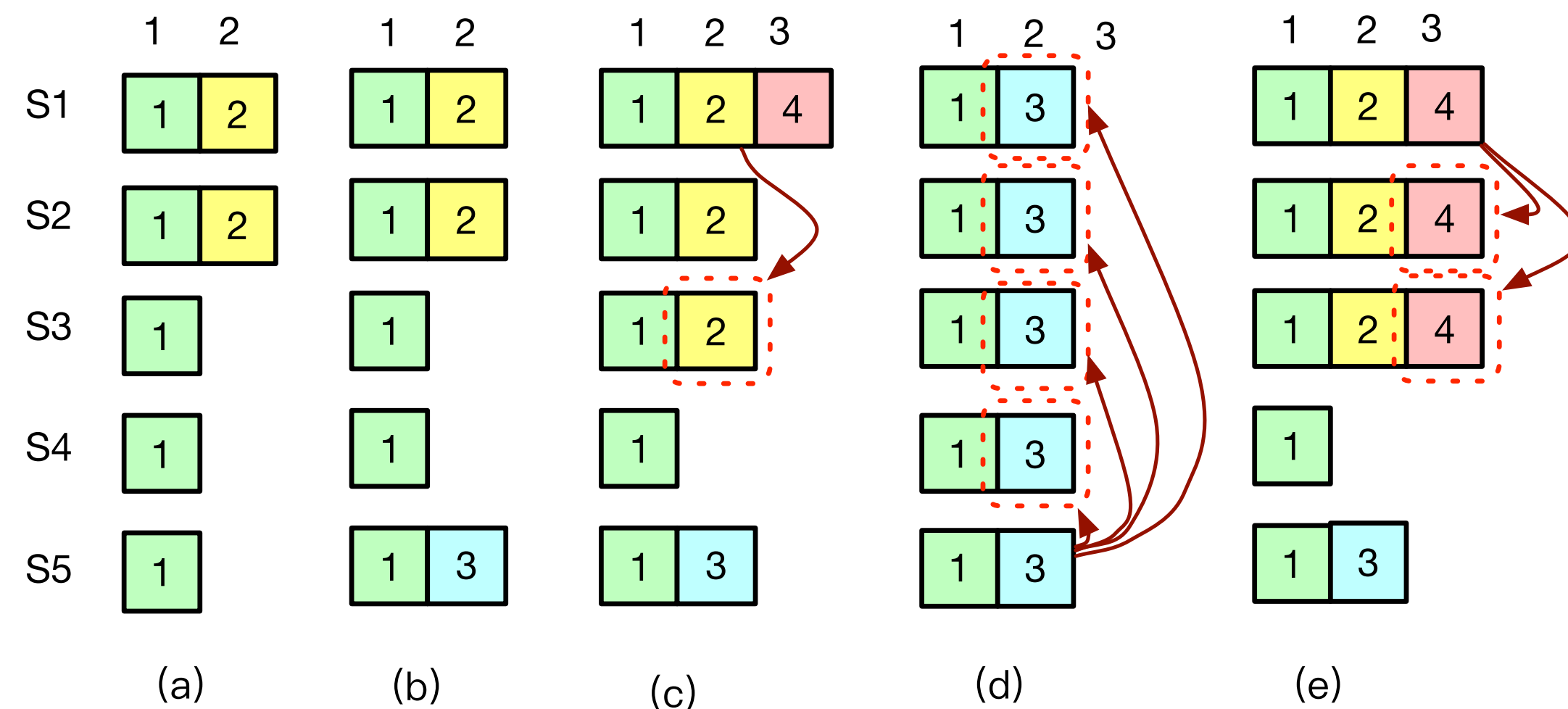
1. 日志条目只有复制到了多个服务器上，才能提交。
2. 一个 candidate 只有赢得了多个服务器的 vote，才能成为 leader。并且要求只有 candidate 的日志比自己的新的时候才能 vote。

下图的 Raft 集群有 5 个服务器：S1、S2、S3、S4、S5。S1 是 leader。S1 把一条日志复制了 S1、S2 和 S3 之后，提交这一条日志。以后发生了选举，S5 成了新 term U 的 leader。上面两点保证一定有一个服务器收到了这条日志并参与了 term U 的投票，这里是 S3。第 2 点又保证 S5 的日志比 S3 的新，所以 S5 必定保存了这条日志。



状态机命令的提交点

前面我说过，一条日志在被复制到多个服务器之后就可以提交。但这是一种不准确的说法。下图解释了为什么 leader 不能靠检查一条日志是否复制到了大多数服务器上来确定旧 term 的日志条目是否已经提交。



准确的说法：Raft 不会通过计算旧 term 的日志条目被复制到了多少台服务器来决定是否它已经被提交，只有当前 term 的日志条目提交状态可以这么决定。如果当前 term 的日志条目被提交，那么基于日志匹配属性，我们知道之前的日志条目也都被间接的提交了。

集群成员更新

我们把 Raft 集群中的机器集合称为集群配置。到目前位置，我们都假定集群配置是固定的。但在实际环境中，我们会经常需要更改配置，例如更换故障的机器或者更改日志的复制级别。我们希望 Raft 集群能够接受一个集群配置变更的请求，然后自动完成集群变更。而且在集群配置的变更中，Raft 集群可以继续提供服务。另外集群配置的变更还要做到一定程度的容错。

Raft 提供一个两阶段的集群成员更新机制。



扫码试看/订阅

《ZooKeeper实战与源码剖析》视频课程