Raft 算法

前半部分是简要描述 详细描述在后面

Raft 是工程上使用较为广泛的强一致性、去中心化、高可用的分布式协议。解决分布式系统中**多个副本的一致性**问题

Raft 将共识问题分解成两个相对独立的问题，**leader election，log replication**

在两个问题中，各自在进行过程中需要满足一些 Safety 的保证。

raft 会先选举出 leader，leader 完全负责 replicated log 的管理。leader 负责接受所有客户端更新请求，然后复制到 follower 节点，并在“安全”的时候执行这些请求。如果 leader 故障，followes 会重新选举出新的 leader。

Leader Election

Raft 协议中，一个节点任一时刻处于以下三个状态之一：

- Leader follower candidate

给出状态转移图能很直观的直到这三个状态的区别

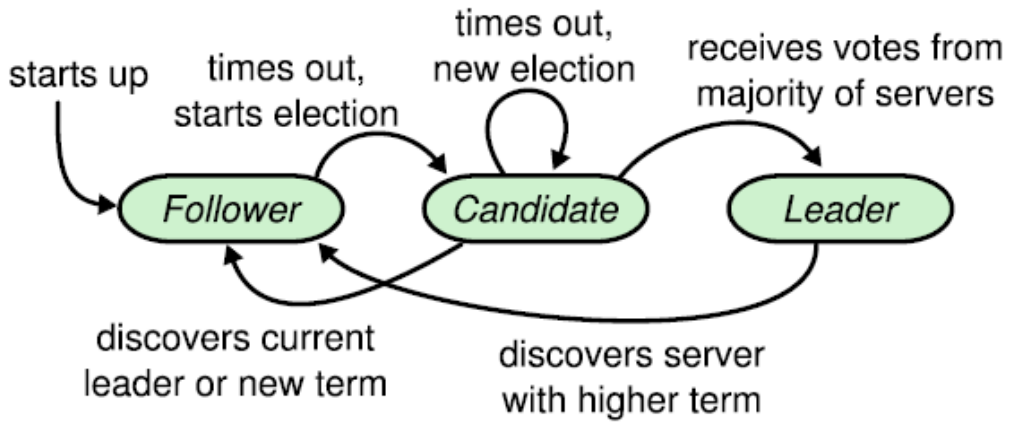


Figure 4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

可以看出所有节点启动时都是 follower 状态；在一段时间内如果没有收到来自 leader 的心跳，从 follower 切换到 candidate，发起选举；如果收到 majority 的造成票（含自己的一票）则切换到 leader 状态；如果发现其他节点比自己更新，则主动切换到 follower。

总之，系统中最多只有一个 leader，如果在一段时间里发现没有 leader，则大家通过选举-投票选出 leader。leader 会不停的给 follower 发心跳消息，表明自己的存活状态。如果 leader 故障，那么 follower 会转换成 candidate，重新选出 leader。

term

从上面可以看出，哪个节点做 leader 是大家投票选举出来的，每个 leader 工作一段时间，然后选出新的 leader 继续负责。这根民主社会的选举很像，每一届新的履职期称之为

一届任期，在 raft 协议中，也是这样的，对应的术语叫 ***term***。

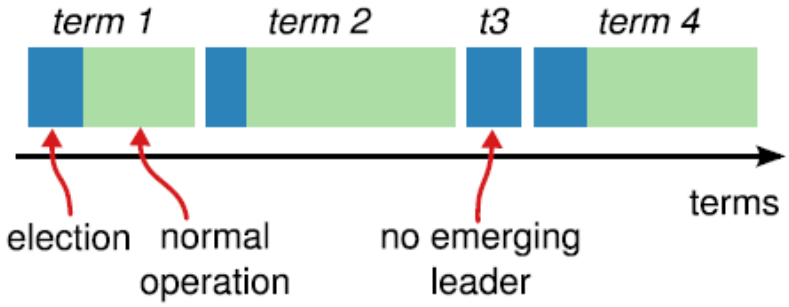


Figure 5: Time is divided into terms, and each term begins with an election. After a successful election, a single leader manages the cluster until the end of the term. Some elections fail, in which case the term ends without choosing a leader. The transitions between terms may be observed at different times on different servers.

term（任期）以选举（election）开始，然后就是一段或长或短的稳定工作期（normal Operation）。从上图可以看到，任期是递增的，这就充当了逻辑时钟的作用；另外，term 3 展示了一种情况，就是说没有选举出 leader 就结束了，然后会发起新的选举，后面会解释这种 *split vote* 的情况。

选举过程详解

上面已经说过，如果 follower 在 *election timeout* 内没有收到来自 leader 的心跳，（也许此时还没有选出 leader，大家都在等；也许 leader 挂了；也许只是 leader 与该 follower 之间网络故障），则会主动发起选举。步骤如下：

- **增加节点本地的 *current term*，切换到 candidate 状态**
- **投自己一票**
- **并行给其他节点发送 *RequestVote RPCs***
- **等待其他节点的回复**

在这个过程中，根据来自其他节点的消息，可能出现三种结果

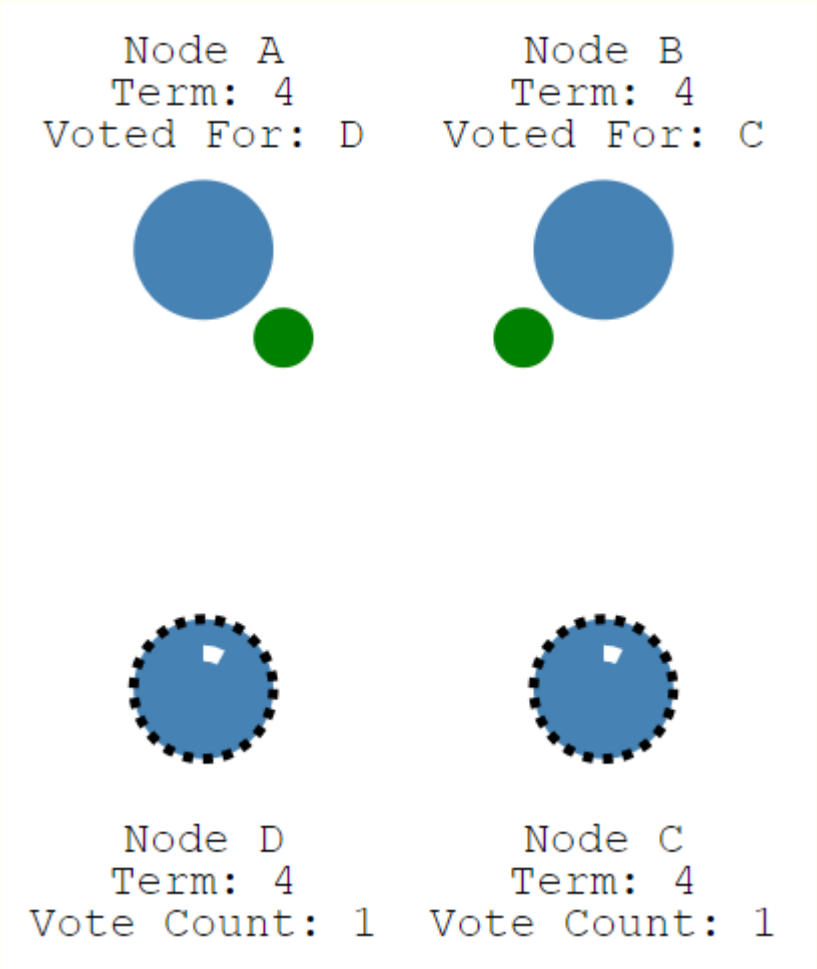
A. **收到 majority 的投票（含自己的一票），则赢得选举，成为 leader**
B. **被告知别人已当选，那么自行切换到 follower**
C. **一段时间内没有收到 majority 投票，则保持 candidate 状态，重新发出选举**

第一种情况，赢得了选举之后，新的 leader 会立刻给所有节点发消息，广而告之，避免其余节点触发新的选举。在这里，先回到投票者的视角，投票者如何决定是否给一个选举请求投票呢，有以下约束：

- 在任一任期内，单个节点最多只能投一票
- 候选人知道的信息不能比自己的少（**候选人的日志 commit 版本不能比自己还落后 这可以确保选举出的 Leader 是最新的**）
- first-come-first-served 先来先得

第二种情况，比如有三个节点 A B C。A B 同时发起选举，而 A 的选举消息先到达 C，C 给 A 投了一票，当 B 的消息到达 C 时，已经不能满足上面提到的第一个约束，即 C 不会给 B 投票，而 A 和 B 显然都不会给对方投票。A 胜出之后，会给 B,C 发心跳消息，节点 B 发现节点 A 的 term 不低于自己的 term，知道有已经有 Leader 了，于是转换成 follower。

第三种情况，没有任何节点获得 majority 投票，比如下图这种情况：



总共有四个节点，Node C、Node D 同时成为了 candidate，进入了 term 4，但 Node A 投了 NodeD 一票，NodeB 投了 Node C 一票，这就出现了平票 split vote 的情况。这个时候大家都在等啊等，直到超时后重新发起选举。如果出现平票的情况，那么就延长了系统不可用的时间（没有 leader 是不能处理客户端写请求的），因此 raft 引入了 randomized election timeouts 来尽量避免平票情况。同时，leader-based 共识算法中，节点的数目都是奇数个，尽量保证 majority 的出现。

Log Replication

当有了 leader，系统应该进入对外工作期了。客户端的**一切请求来发送到 leader**，leader 来调度这些并发请求的顺序，并且保证 leader 与 followers 状态的一致性。raft 中的做法是，将这些**请求以及执行顺序告知 followers。leader 和 followers 以相同的顺序来执行这些请求，保证状态一致**。

简单来说：**相同的初识状态 + 相同的输入 = 相同的结束状态**。引文中有一个很重要的词 deterministic，就是说不同节点要以相同且确定性的函数来处理输入，而不要引入一下不确定的值，比如本地时间等。如何保证所有节点 get the same inputs in the same order，使用 replicated log 是一个很不错的注意，log 具有持久化、保序的特点，是大多数分布式系统的基石。

因此，可以这么说，在 raft 中，leader 将客户端请求（command）封装到一个个 log entry，将这些 log entries 复制（replicate）到所有 follower 节点，然后大家按相同顺序应用（apply）log entry 中的 command，则状态肯定是一致的。

下图形象展示了这种 log-based replicated state machine

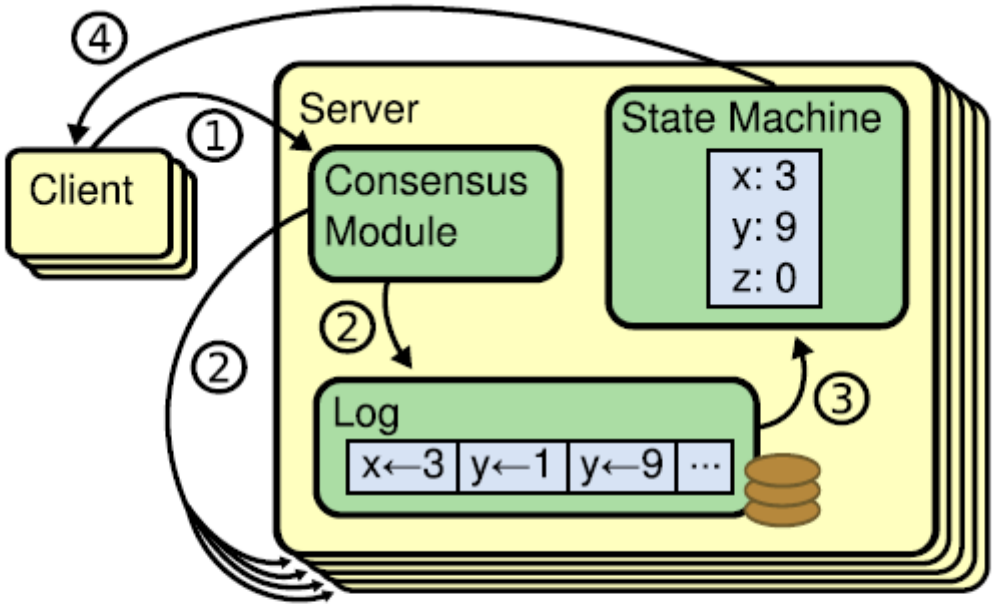


Figure 1: Replicated state machine architecture. The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

请求完整流程

当系统（leader）收到一个来自客户端的写请求，到返回给客户端，整个过程从 leader 的视角来看会经历以下步骤（**写请求的完整处理步骤**）：

- leader append log entry
- leader issue AppendEntries RPC in parallel

- leader wait for majority response
- leader apply entry to state machine
- leader reply to client
- leader notify follower apply log

可以看到日志的提交过程有点类似两阶段提交(2PC)，不过与 2PC 的区别在于，leader 只需要大多数（majority）节点的回复即可，这样只要超过一半节点处于工作状态则系统就是可用的。

那么日志在每个节点上是什么样子的呢

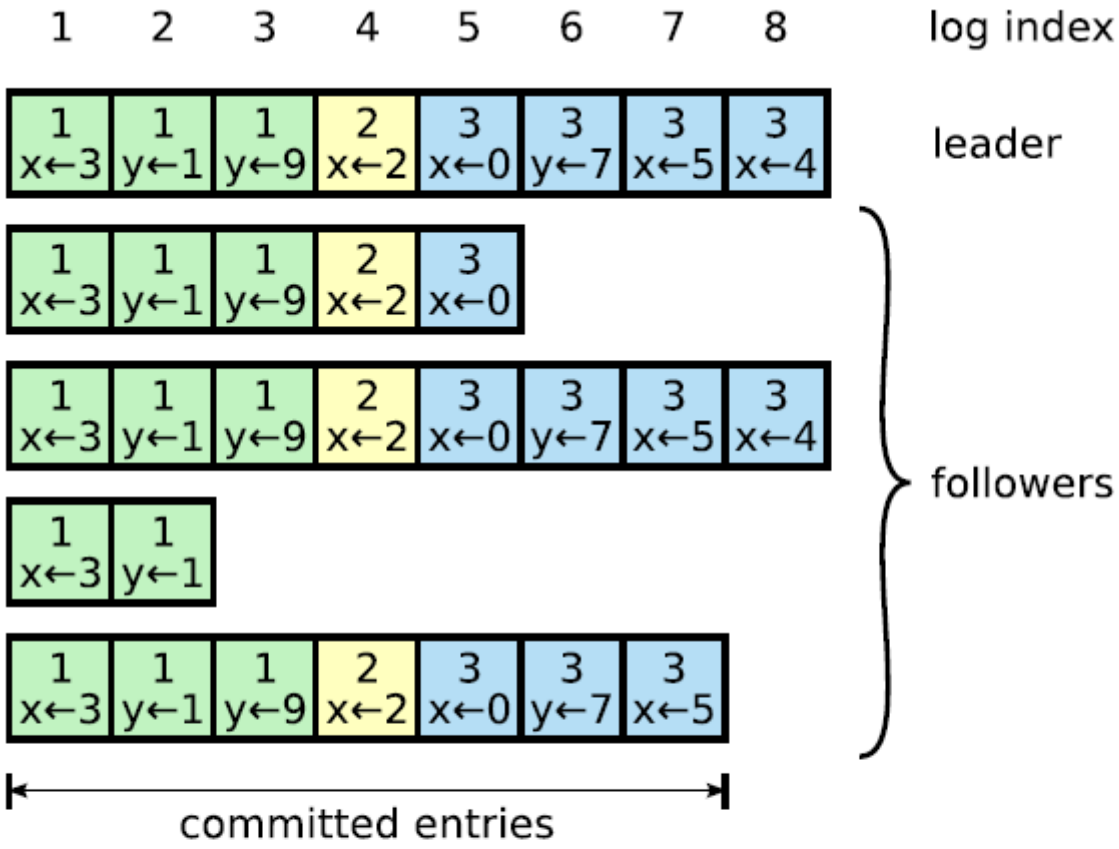


Figure 6: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

不难看到，logs 由顺序编号的 log entry 组成 ，每个 log entry 除了包含 command，还包含产生该 log entry 时的 leader term。从上图可以看到，五个节点的日志并不完全一致，raft 算法为了保证高可用，并不是强一致性，而是最终一致性，leader 会不断尝试给 follower 发 log entries，直到所有节点的 log entries 都相同。

在上面的流程中，leader 只需要日志被复制到大多数节点即可向客户端返回，一旦向客户端返回成功消息，那么系统就必须保证 log（其实是 log 所包含的 command）在任何异常的情况下都不会发生回滚。这里有两个词：**commit（committed），apply(applied)**，前者是指日志

被复制到了大多数节点后日志的状态；而后者则是节点将日志应用到状态机，真正影响到节点状态。

在 Leader Election 以及 Log Replication 的一些 Safety 保证

在上面提到只要日志被复制到 majority 节点，就能保证不会被回滚，即使在各种异常情况下，这根 leader election 提到的选举约束有关。在这一部分，主要讨论 raft 协议在各种各样的异常情况下如何工作的。

衡量一个分布式算法，有许多属性，如

- safety：nothing bad happens,
- liveness： something good eventually happens.

在任何系统模型下，都需要满足 safety 属性，即在任何情况下，系统都不能出现不可逆的错误，也不能向客户端返回错误的内容。比如，raft 保证被复制到大多数节点的日志不会被回滚，那么就是 safety 属性。而 raft 最终会让所有节点状态一致，这属于 liveness 属性。

raft 协议会保证以下属性

Election Safety: at most one leader can be elected in a given term. §5.2

Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries. §5.3

Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index. §5.3

Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms. §5.4

State Machine Safety: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. §5.4.3

Figure 3: Raft guarantees that each of these properties is true at all times. The section numbers indicate where each property is discussed.

Election safety

选举安全性，即任一任期内**最多一个 leader 被选出**。这一点非常重要，在一个复制集中任何时刻只能有一个 leader。系统中同时有多余一个 leader，被称之为脑裂（brain split），这是非常严重的问题，会导致数据的覆盖丢失。在 raft 中，两点保证了这个属性：

- 一个节点某一任期内最多只能投一票；
- 只有获得 majority 投票的节点才会成为 leader。

因此，**某一任期内一定只有一个 leader**。

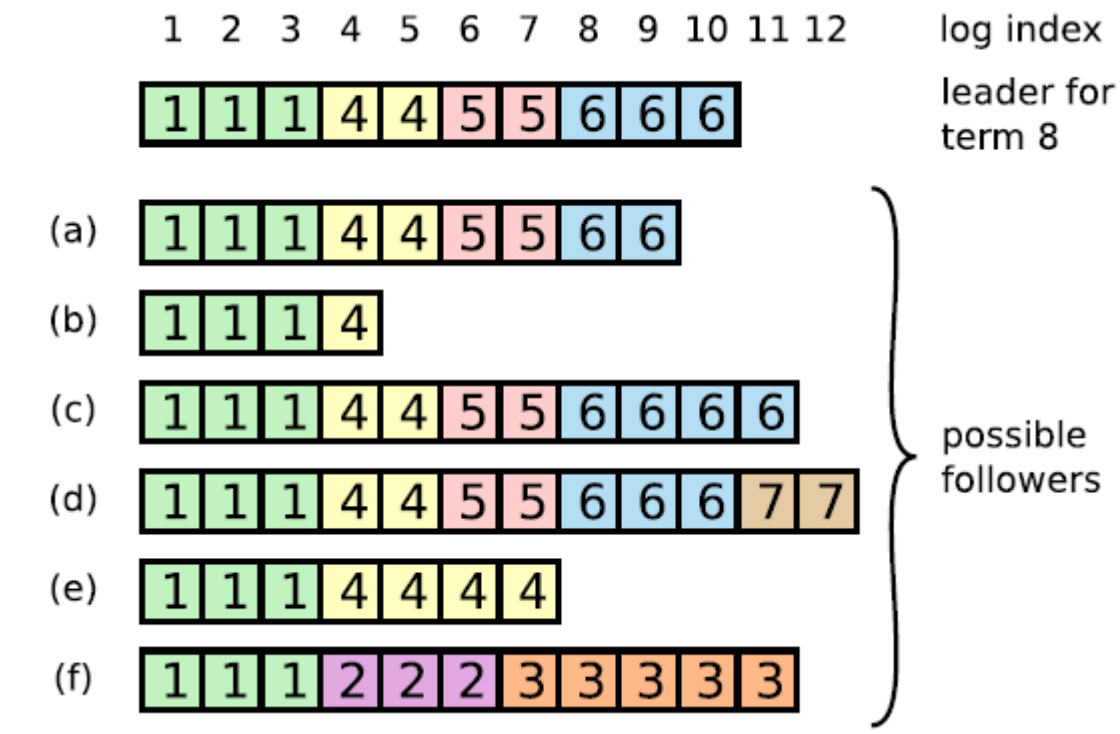
log matching

很有意思，log 匹配特性， 就是说如果两个节点上的某个 log entry 的 log index 相同且 term 相同，那么在该 index 之前的所有 log entry 应该都是相同的。如何做到的？依赖于以下两点

- If two entries in different logs have the same index and term, then they store the same command.
- If two entries in different logs have the same index and term, then the logs are identical in all preceding entries.

首先，leader 在某一 term 的任一位置只会创建一个 log entry，且 log entry 是 append-only。其次，consistency check。leader 在 AppendEntries 中包含最新 log entry 之前的一个 log 的 term 和 index，**如果 follower 在对应的 term index 找不到日志，那么就会告知 leader 不一致**。

在没有异常的情况下，log matching 是很容易满足的，但如果出现了 node crash，情况就会变得复杂。比如下图



注意：上图的 a-f 不是 6 个 follower，而是某个 follower 可能存在的六个状态

leader、follower 都可能 crash，那么 follower 维护的日志与 leader 相比可能出现以下情况

- 比 leader 日志少，如上图中的 ab

- 比 leader 日志多，如上图中的 cd
- 某些位置比 leader 多，某些日志比 leader 少，如 ef（多少是针对某一任期而言）

当出现了 leader 与 follower 不一致的情况，**leader 强制 follower 复制自己的 log**

To bring a follower’s log into consistency with its own, the leader must find the latest log entry where the two logs agree, delete any entries in the follower’s log after that point, and send the follower all of the leader’s entries after that point.

leader 会维护一个 nextIndex[]数组，记录了 leader 可以发送每一个 follower 的 log index，初始化为 leader 最后一个 log index 加 1， 前面也提到，leader 选举成功之后会立即给所有 follower 发送 AppendEntries RPC（不包含任何 log entry， 也充当心跳消息）,那么流程总结为：

s1 leader 初始化 nextIndex[x]为 leader 最后一个 log index + 1
s2 AppendEntries 里 prevLogTerm prevLogIndex 来自 logs[nextIndex[x] - 1]
s3 如果 follower 判断 prevLogIndex 位置的 log term 不等于 prevLogTerm，那么返回 false，否则返回 True
s4 leader 收到 follower 的回复，如果返回值是 True，则 nextIndex[x] -= 1，跳转到 s2. 否则
s5 同步 nextIndex[x]后的所有 log entries

leader completeness vs elcetion restriction

leader 完整性：如果一个 log entry 在某个任期被提交（committed），那么这条日志一定会出现在所有更高 term 的 leader 的日志里面。这个跟 leader election、log replication 都有关。

- 一个日志被复制到 majority 节点才算 committed
- 一个节点得到 majority 的投票才能成为 leader，而节点 A 给节点 B 投票的其中一个前提是，B 的日志不能比 A 的日志旧。下面的引文指处了如何判断日志的新旧

voter denies its vote if its own log is more up-to-date than that of the candidate.

If the logs have last entries with different terms, then the log with the later term is more up-to-date. If the logs end with the same term, then whichever log is longer is more up-to-date.

上面两点都提到了 majority：commit majority and vote majority，根据 Quorum，这两个 majority 一定是有重合的，因此被选举出的 leader 一定包含了最新的 committed 的日志。

raft 与其他协议（Viewstamped Replication、mongodb）不同，raft 始终保证 leader 包含最新的已提交的日志，因此 leader 不会从 follower catchup 日志，这也大大简化了系统的复杂度。