软件背后的算法

哈希 / 布隆过滤器 / 一致性哈希 / CAP定理和最终一致性 / PAXOS

我们将聊些哪些内容?

- 哈希函数 (Hash Function)
- 布隆过滤器 (Bloom Filter)
- 一致性哈希 (Consistent Hashing)
- CAP定理 (CAP Theorem)
- 最终一致性 (Eventual Consistency)
- 两阶段提交 (2 Phase Commit)
- Paxos

我们不聊哪些内容?

- 具体代码
- 数学公式
- 数学归纳法...

工作中最常用的算法是什么？

很多人告诉我：

“基本工资 + 绩效工资 - 迟到扣款 - 个税代缴 - 社保代扣 - 公积金代扣 = ？”

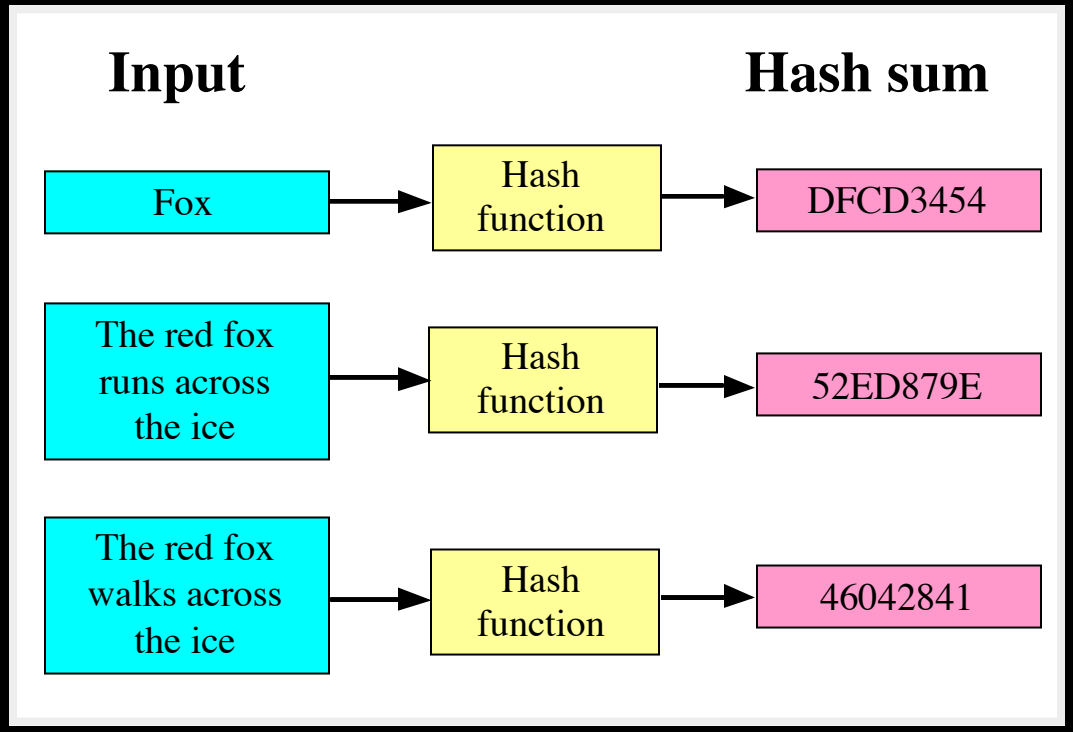
为什么要学习算法?

知乎用户:

“算法培养的是思考问题解决问题的通性通法，而不是某一种具体的方法。”

哈希函数

- 从任何一种数据中创建小的数字“指纹”（特征码）的方法
- 如果得到的“指纹”不同，那么认为输入是不同的两个东西



哈希函数的应用

- 密码哈希 (MD5)
- 冗余校验 (数据传输中是否出错)
- 资源识别 (资源是否被篡改、替换)
- 哈希表

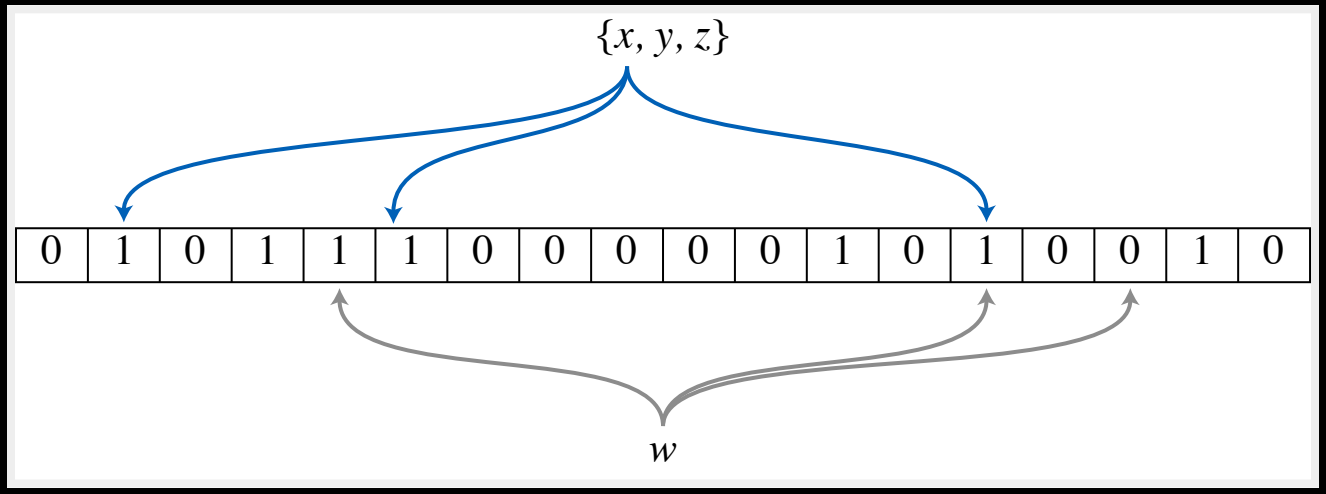
网络爬虫的问题：

“网页中的链接很可能形成一个环，如何判断一个URL是否已抓过？”

1. 将URL存入数据库
2. 将URL存入哈希表
3. 将URL->MD5存入哈希表
4. 布隆过滤器

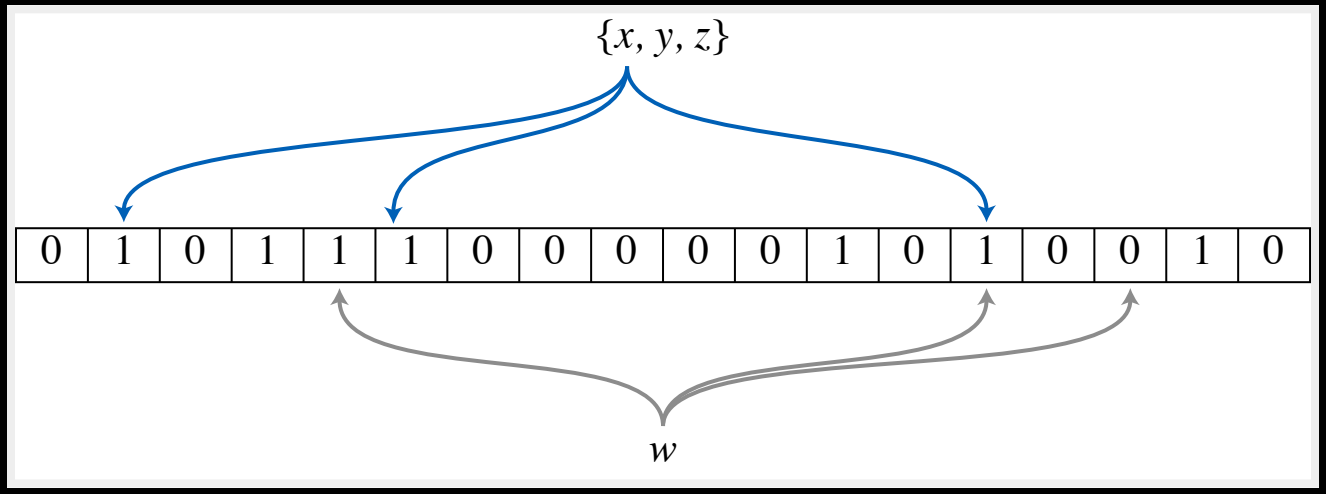
布隆过滤器

- 布隆过滤器管理着一个大小不变的 bit 数组，初始值均为 0
- 每提交一个新数据，会将其哈希后对应的位置设为 1
- 如果新数据哈希后对应的位置均已为 1，则认为已存在



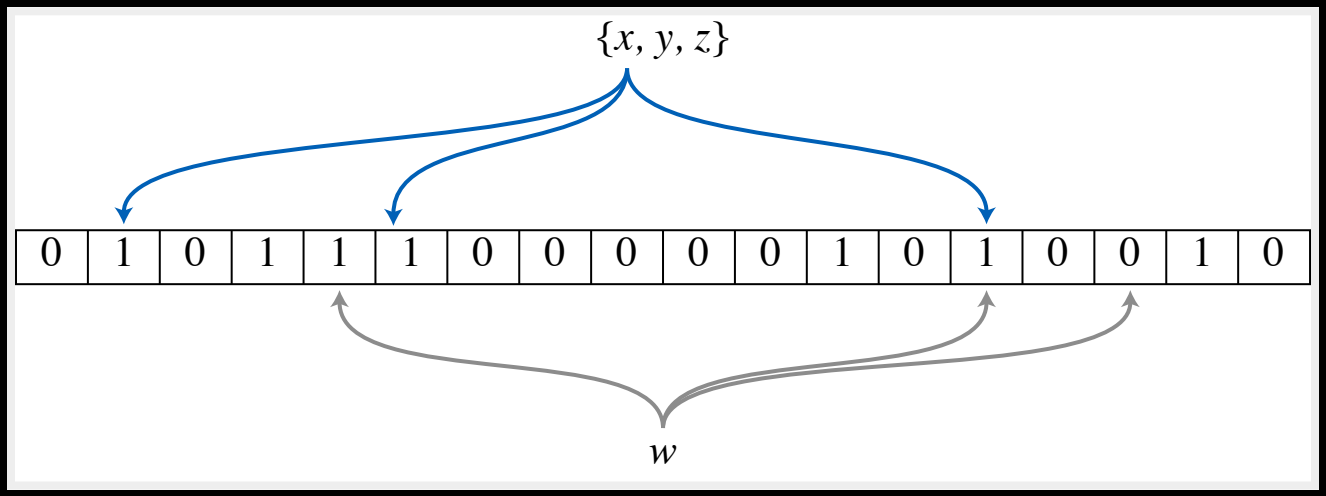
布隆过滤器的优点

- 时间: 查询速度快 (常数时间)
- 空间: 占用空间极小



布隆过滤器的缺点

- 存在误判率
- 无法删除元素 (会影响其他元素)



假设有N台缓存设备，哈希算法是

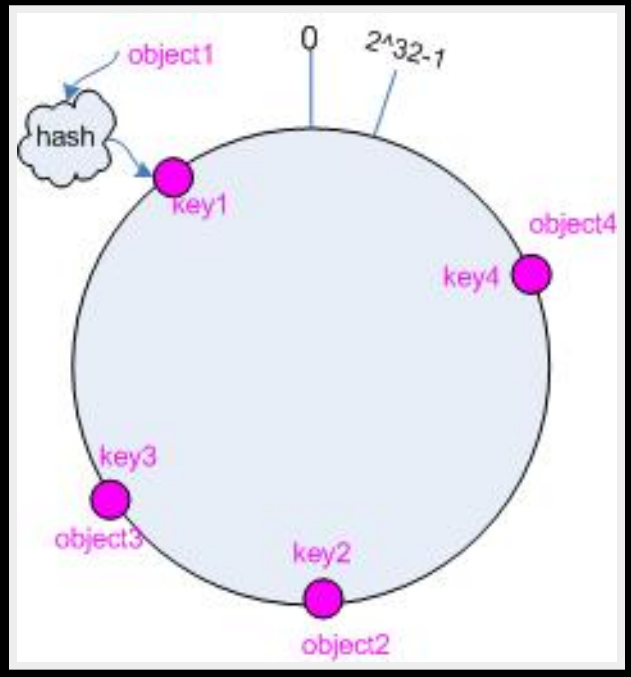
```
function getCacheNode(key) {
  return hash(key) % N
}
```

存在的问题：

- 如果其中一台设备挂掉(N-1)，或新增一台设备(N+1)都将导致几乎所有缓存失效，压力瞬间冲向后端
- 硬件配置越来越好，但却无法让后续节点承担更多压力

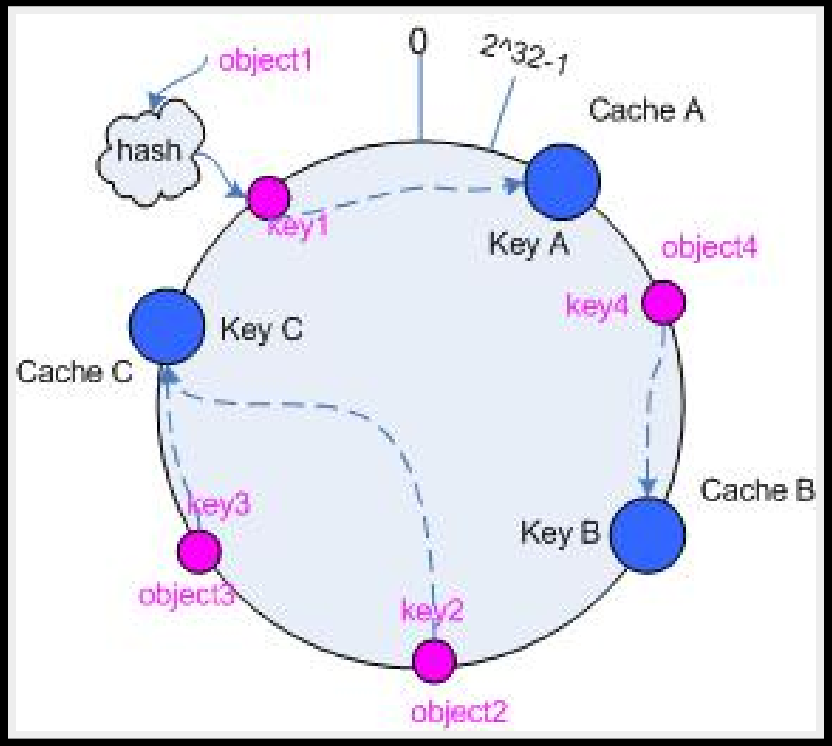
一致性哈希

- 想象一个 2^32 个点组成的圆，从顶点 0 开始顺时针直到 2^32-1
- 一致性哈希将 key 映射到圆环的一个点



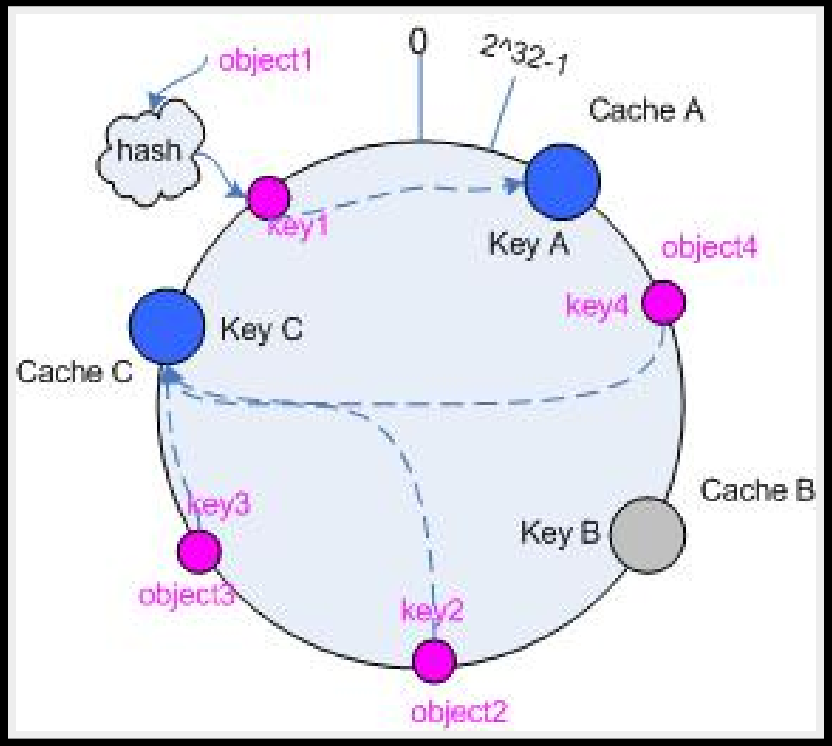
一致性哈希

- 将缓存设备也映射到圆上，每个 key 沿着圆环边顺时针进行查找，直到发现缓存后即保存在此设备



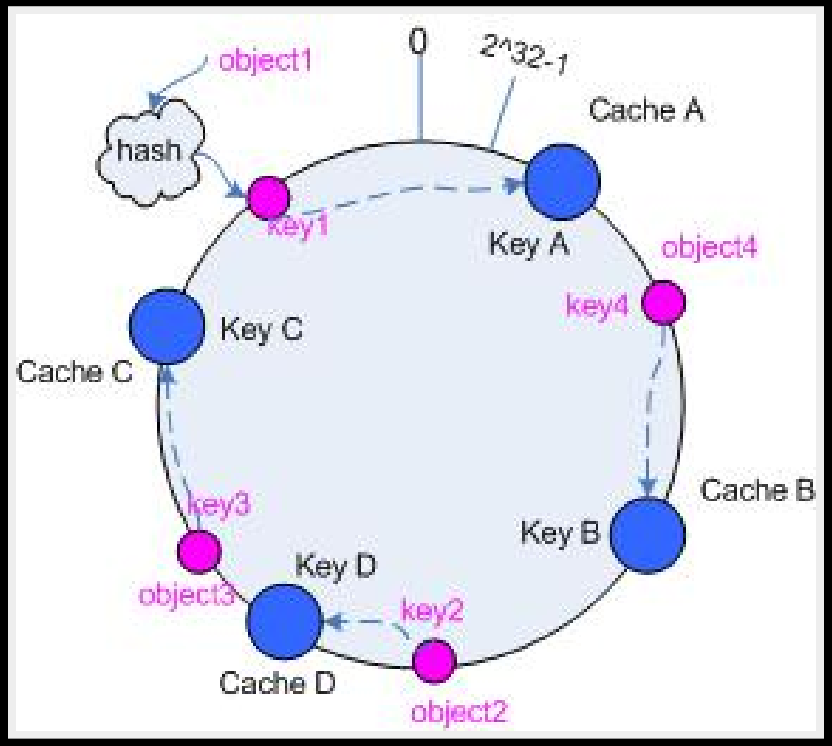
一致性哈希

- 当设备 Cache B 挂掉，key4 的请求将前往 Cache C，这里只导致了 key4 失效



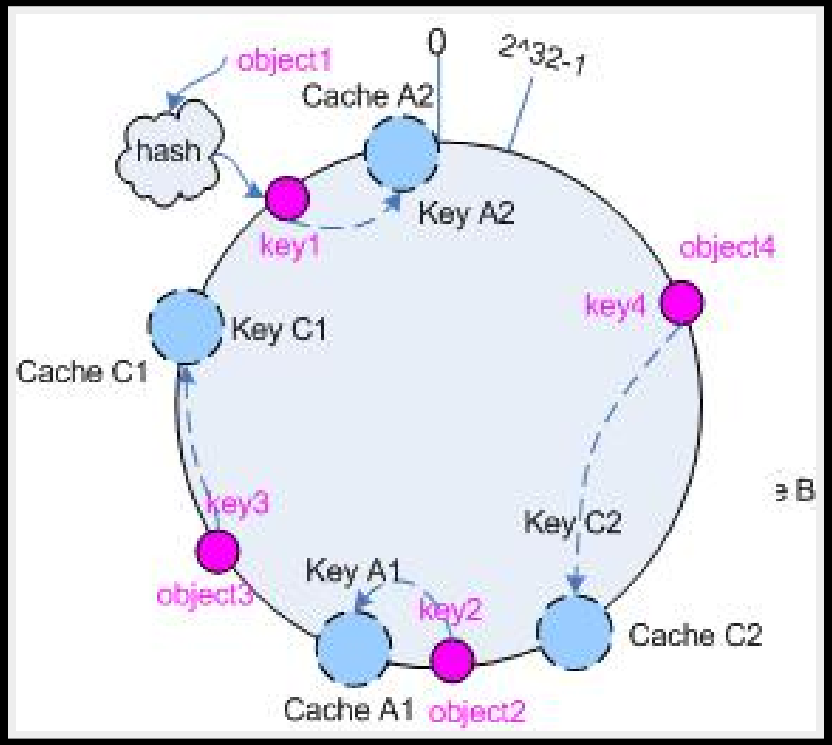
一致性哈希

- 当新增一台 Cache D，key2 的请求将前往 Cache D，这里只导致了 key2 失效



一致性哈希

- 虚拟节点：可以让一台缓存设备，在圆上建立多个映射



一致性哈希的应用

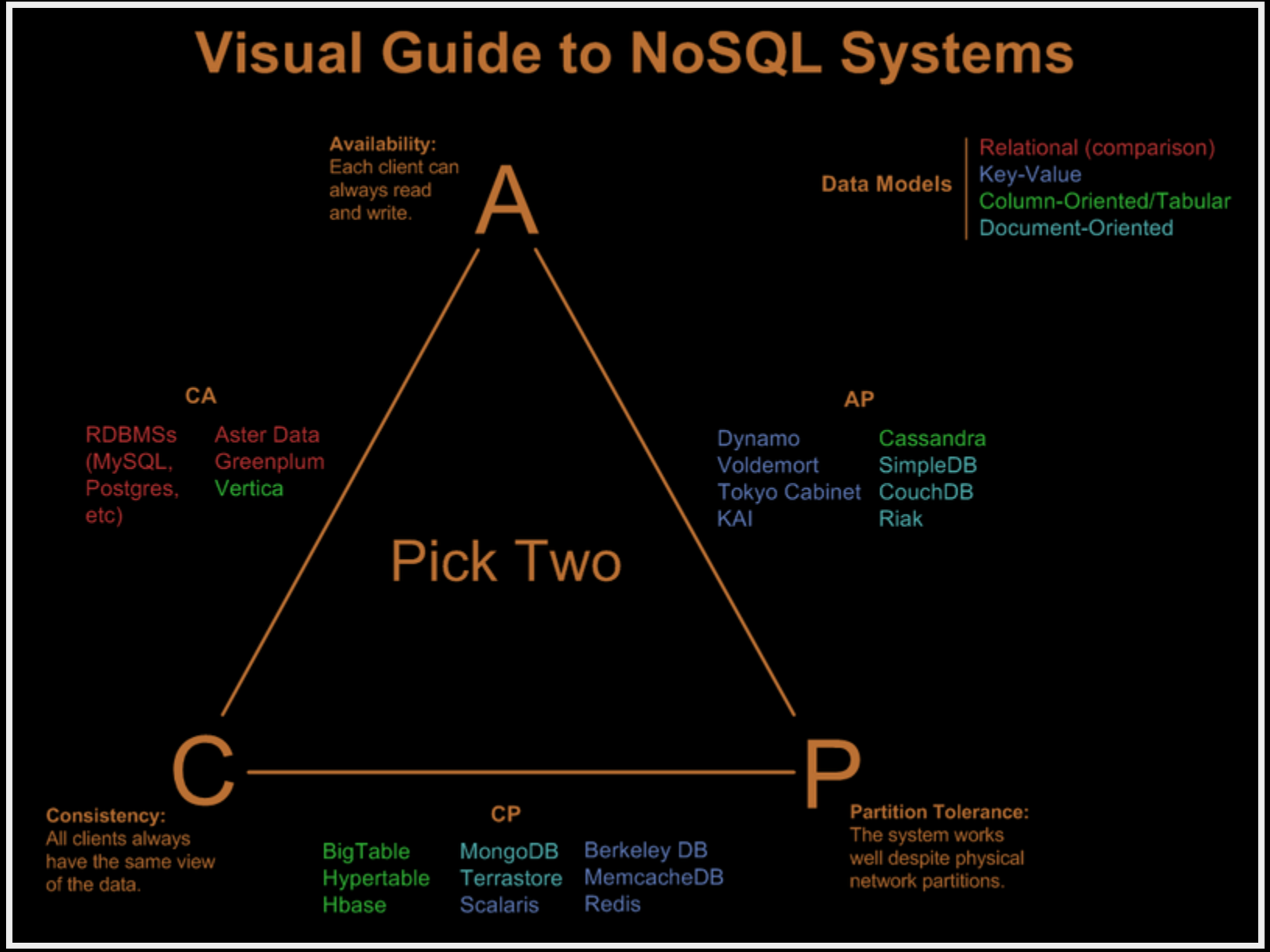
- Openstack's Object Storage Service Swift
- Partitioning component of Amazon Dynamo
- Data partitioning in Apache Cassandra
- Data Partitioning in Voldemort
- Akka's consistent hashing router
- Riak, a distributed key-value database
- GlusterFS, a network-attached storage file system

CAP 定理

- Consistency: 一致性，访问任何节点得到的数据一致
- Availability: 可用性，任意节点可以提供服务
- Partition tolerance: 分区容忍性，机器之间无法在期望的时间内完成数据同步（包括断网、单方宕机）

> *“创建一个具备完全一致性、可用性与分区容错性的分布式数据库，是不可能的。你只能保证三者中的两个”*

Visual Guide to NoSQL Systems



MYSQL 之于 CAP

- 水平切分的数据库叫做“数据分区”，相对的是“数据镜像”，后者才存在“网络分区”情况
- MySQL Master-Slave 由于延迟会导致网络分区(P)的情况，因此属于 AP (在网络分区时选择了可用性)

MONGODB 之于 CAP

- MongoDB 默认只能在 Master 上进行读写，从而保证了一致性(C)
- 在网络分区后集群会选举出新的 Master，因此属于 CP (在网络分区时选择了一致性)

COUCHDB 之于 CAP

- CouchDB 每个节点均可写入，并记录每次写入的版本号(Rev)，每次更新需提交上一次的版本号，以验证是否过期
- 当网络分区导致两个节点数据冲突时，会自行选择一个版本生效，另一个版本标记为“冲突”，可以由程序按自己的规则解决
- 因此 CouchDB 属于 CA，只保证最终一致性。同属 CA 阵营的 Dynamo 和 Riak 采用了向量时钟

最终一致性的故事

- 我是一名好孩子™推车部门的销售，每次公司发布新款，总监都会给我们培训
- 开年会喝大了，第二天发现自己身处山西黑煤窑，与世隔绝只能挖煤保命……这一挖就是 3 年
- 某天我嘴里叼着半拉馍，被矿长媳妇拦住了：“听说你懂推车？现在好孩子™最好的推车是什么？好用我就放你走”

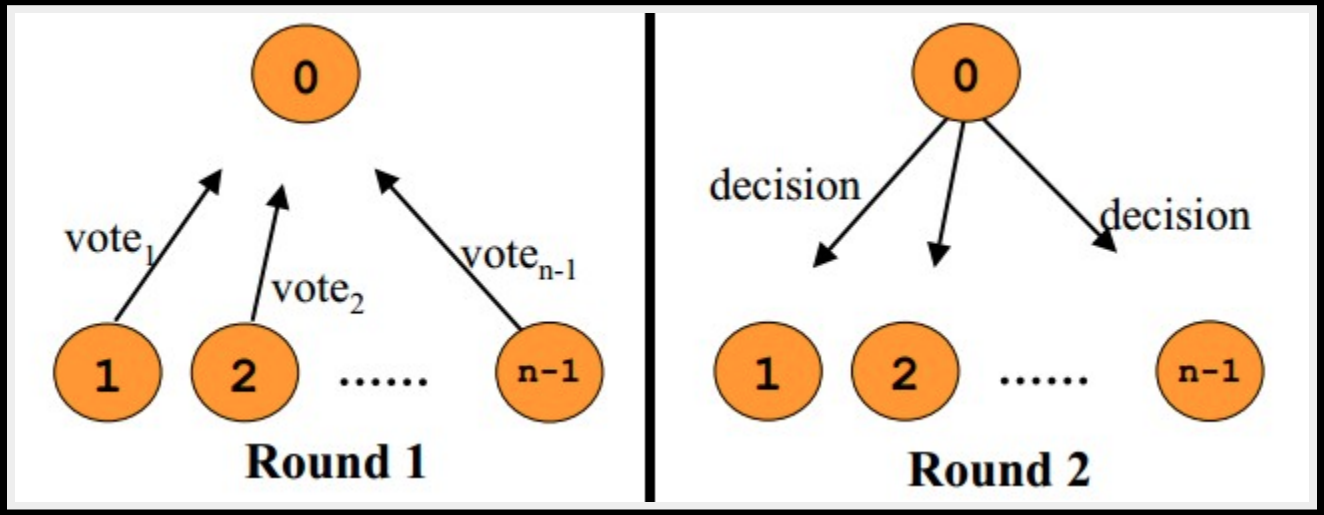
Happy Ending (保持可用, 最终一致性)

- 我：“蜂鸟，妥妥的”(我所知道的，3 年前的新款)
- 老板娘也接受了这款并放走了我，回到公司后我了解到现在最新的是口袋车，从此过上了幸福的生活

Bad Ending (保持强一致性)

- 我：“我不知道了”
- ……

2PC 两阶段提交



- 牧师分别问新郎新娘：你是否愿意...不管生老病死...(Prepare)
- 当双方都同意时牧师会说：我宣布你们...(Commit)

2PC 的问题

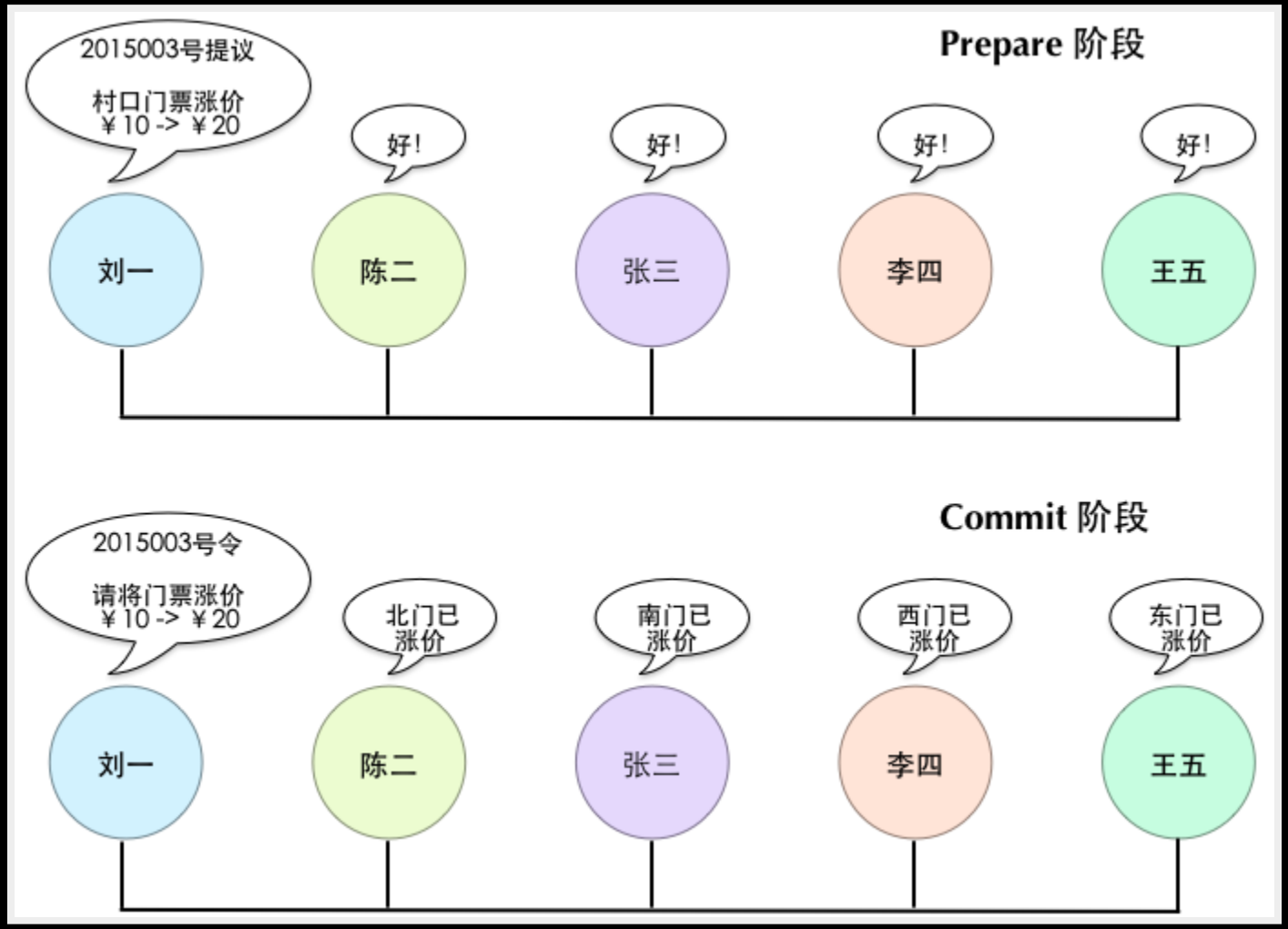
- Prepare 阶段是阻塞的
- 如果在 Commit 之前协调者突然消失，那么每个参与者此时都不能确定自己数据是一致（最新）的，需要等待协调者恢复

PAXOS

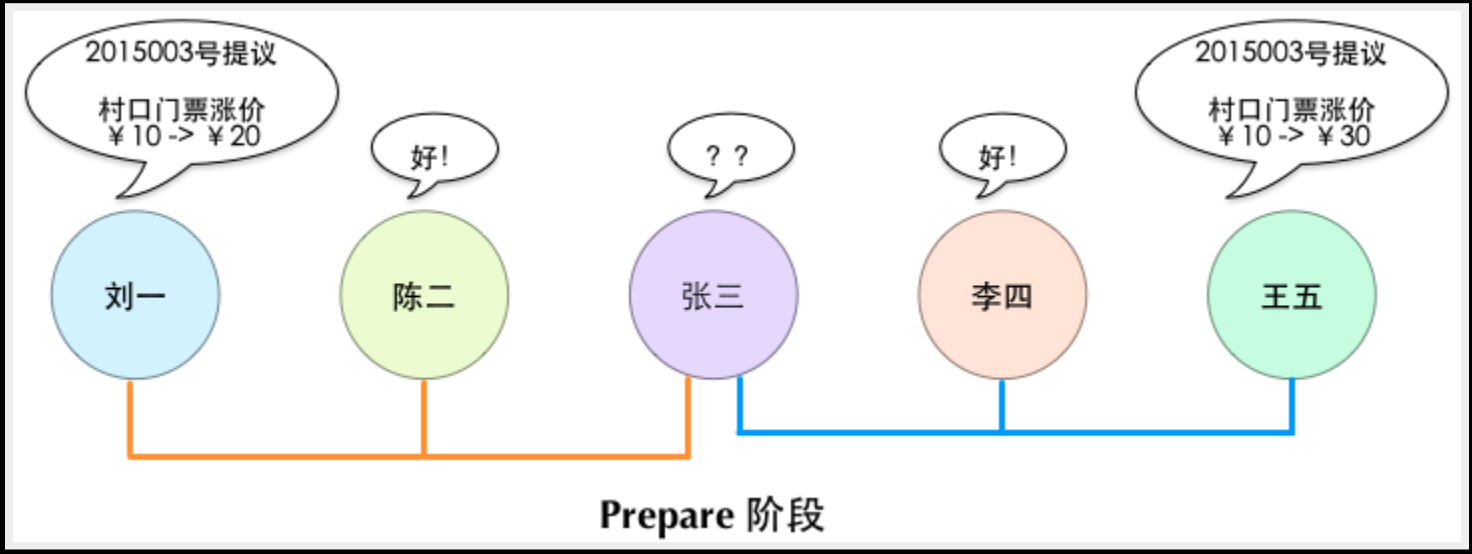
- 作者是 Leslie Lamport，即 LaTeX 的 "La"
- 最初在 1990 年提出，因难以理解而没有引起重视，他就把这个故事一直讲到了 2006 年

Paxos 是一个希腊岛屿，岛民采取民主制决定律法，但不要求每次大会所有人都出席

正常情况下的 PAXOS 就像 2PC

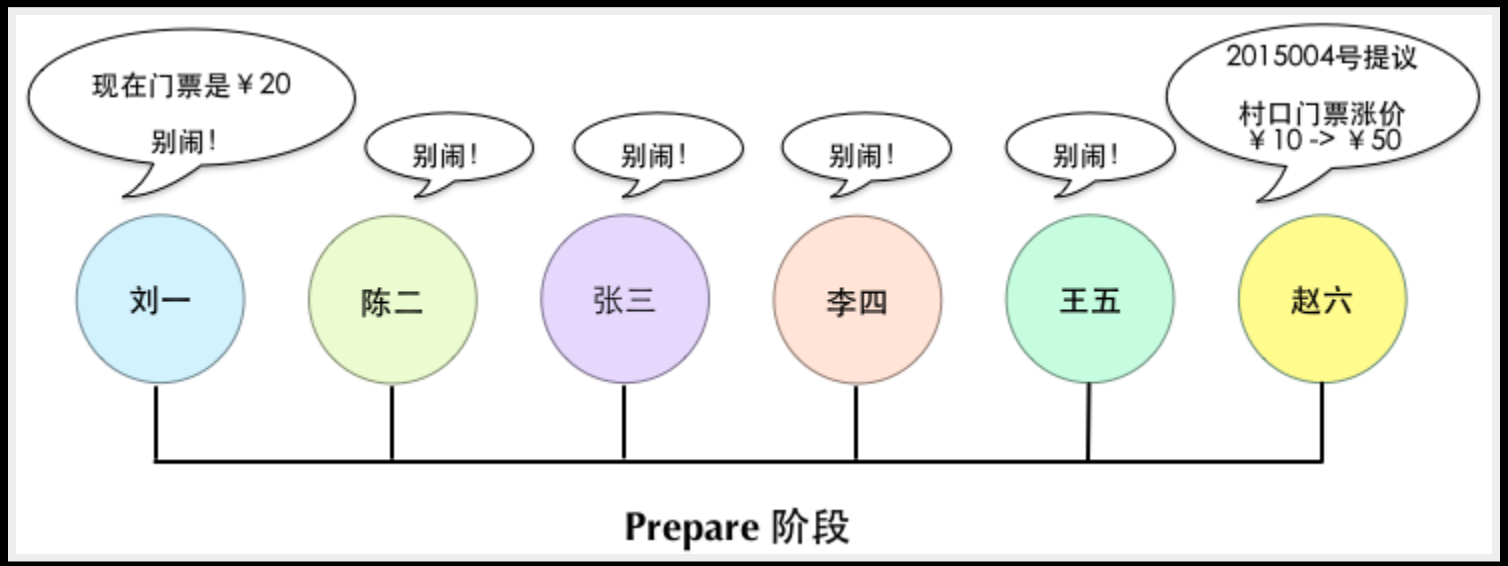


网络分区情况下的 PAXOS



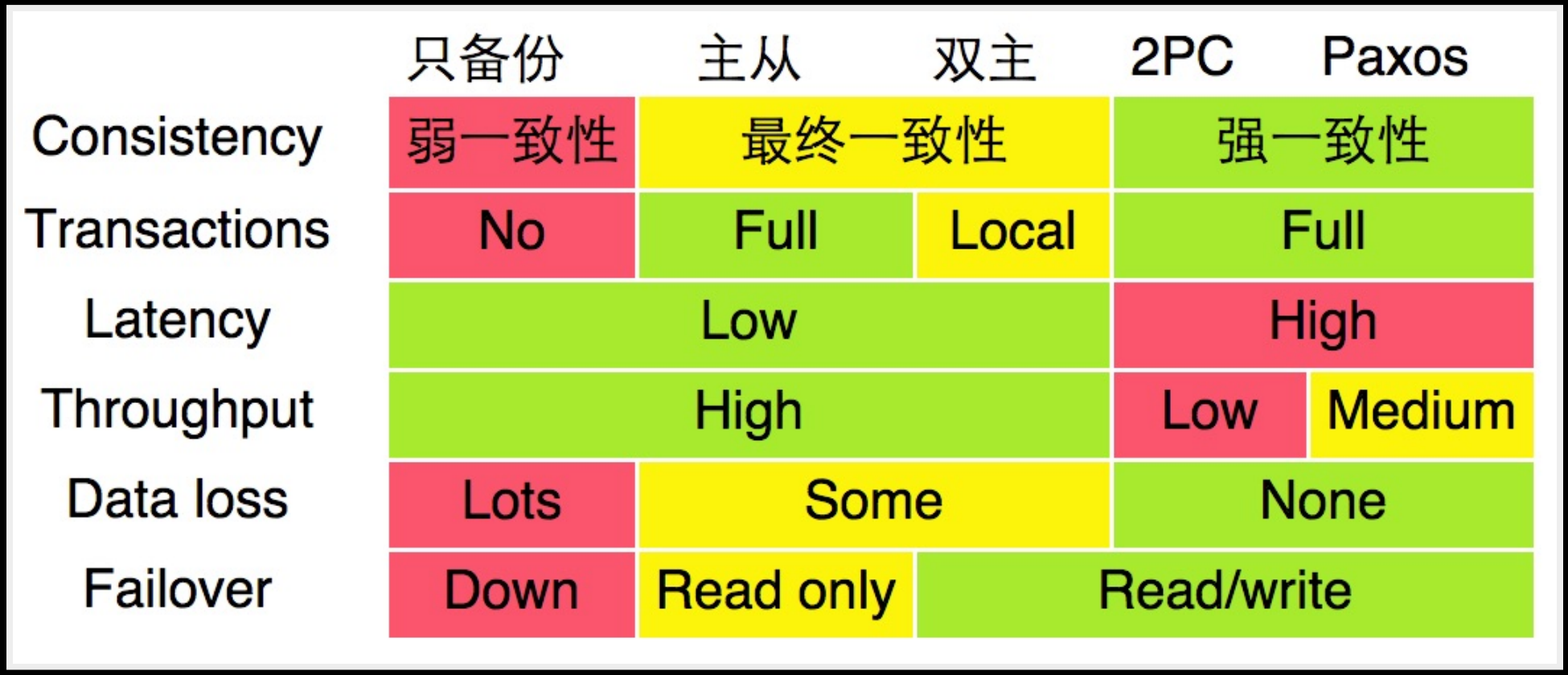
此时张三将决定哪一边形成“法定人数”(Quorum)

打工归来的赵六



- 赵六离开家时，门票还是￥10，见过大世面的他觉得应该涨到￥50
- 岛民们告诉赵六现在已经涨到￥20了，你回去想想再说

总结



	只备份	主从	双主	2PC	Paxos
Consistency	弱一致性	最终一致性		强一致性	
Transactions	No	Full	Local	Full	
Latency	Low			High	
Throughput	High			Low	Medium
Data loss	Lots	Some		None	
Failover	Down	Read only	Read/write		

Google I/O 2009 《Transaction Across DataCenter》

by Ryan Barrett

关于 2PC / Paxos 你可能还需要知道的

- 两将军问题
- 拜占庭将军问题

扩展阅读

- 《数学之美》
- 《大数据日知录》

算法学习

- Big-O Cheat Sheet
- 演算法筆記

算法练习

- LeetCode
- HackerRank

THE END

讨论环节