Google云计算原理

第3章

武汉大学 计算机学院 软件工程系

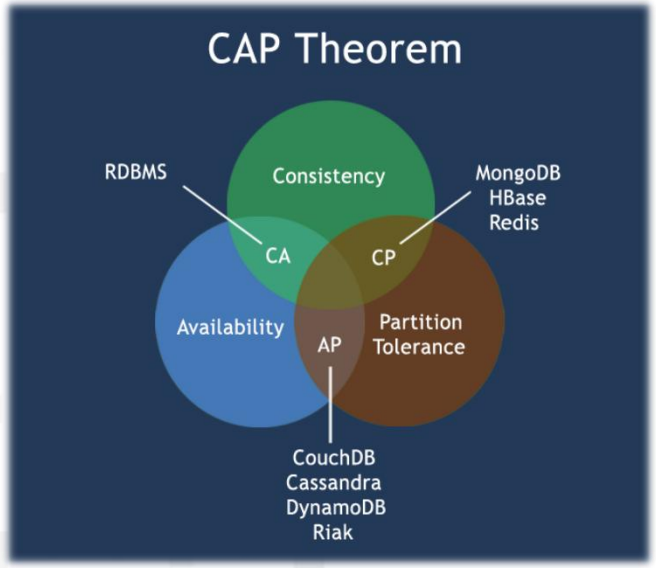
目 录

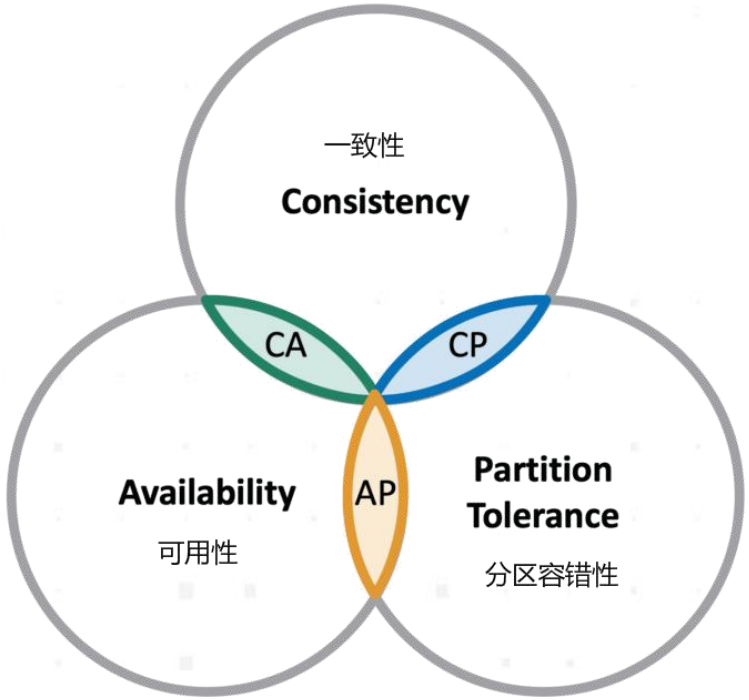
3.1 Google文件系统GFS

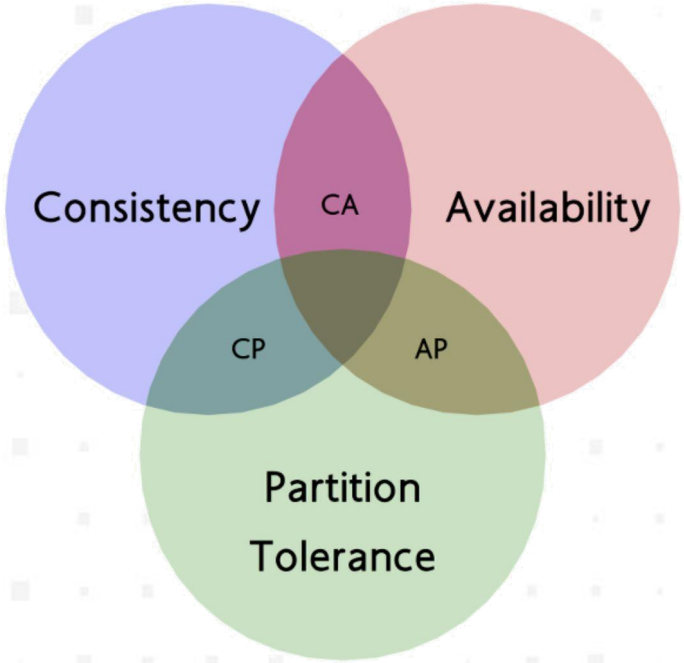
3.2 分布式数据处理MapReduce

3.3 分布式锁服务Chubby

3.4 分布式结构化数据表Bigtable

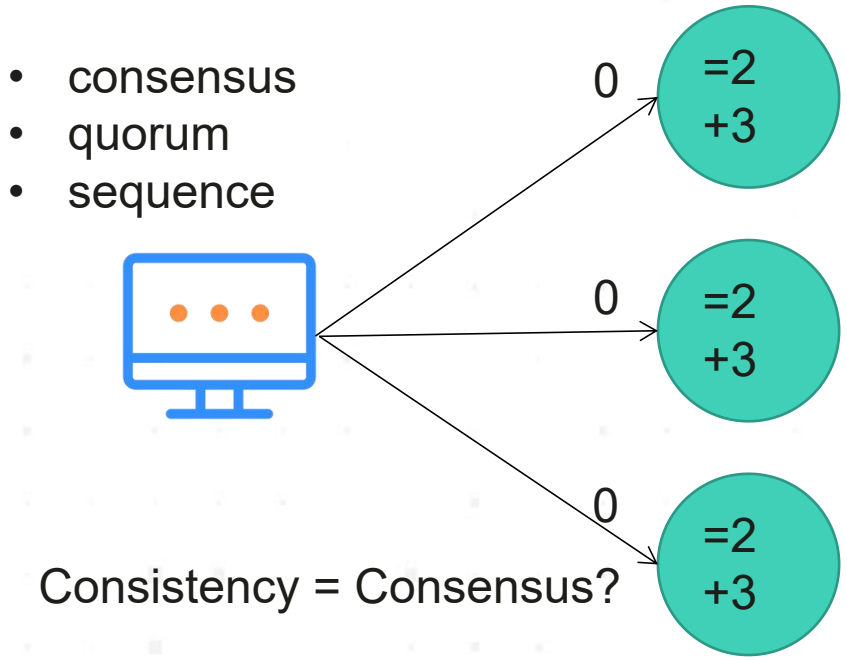






3.3 分布式锁服务Chubby

- 3.3.1 Paxos算法
- 3.3.2 Chubby系统设计
- 3.3.3 Chubby中的Paxos
- 3.3.4 Chubby文件系统
- 3.3.5 通信协议
- 3.3.6 正确性与性能



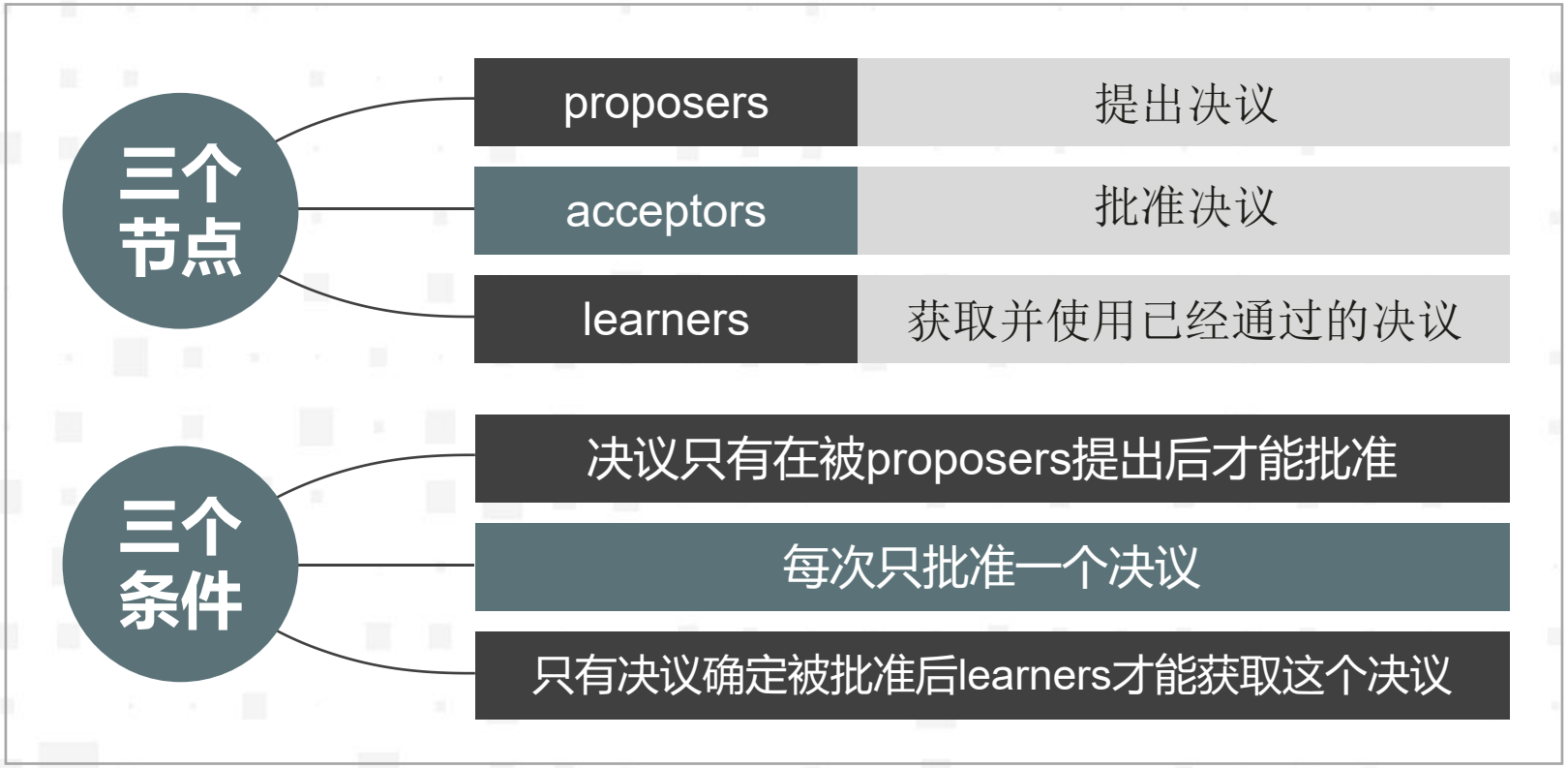
3.3 分布式锁服务Chubby

- **Paxos 算法背景知识**

1	processor可以担任三个角色“proposer”、“accepter”和“learner”中的一个或多个角色。
2	proposal和value： proposal一般译为“提案”，value一般译为“决议”。
3	proposer可以propose（提出）proposal； accepter可以accept（接受）proposal
4	各个processor之间信息的传递可以延迟、丢失，但是在这个算法中假设传达到的信息都是正确的

3.3 分布式锁服务Chubby

- Paxos算法



3.3 分布式锁服务Chubby

- 系统的约束条件

p1：每个acceptor只接受它得到的第一个决议。

p2：一旦某个决议得到通过，之后通过的决议必须和该决议保持一致。

p2a：一旦某个决议v得到通过，之后任何acceptor再批准的决议必须是v。

p2b：一旦某个决议v得到通过，之后任何proposer再提出的决议必须是v。

p2c：如果一个编号为n的提案具有值v，那么存在一个“多数派”，要么它们中没有谁批准过编号小于n的任何提案，要么它们进行的最近一次批准具有值v。

为了保证决议的唯一性，acceptors也要满足一个约束条件：当且仅当 acceptors 没有收到编号大于n的请求时，acceptors 才批准编号为n的提案。

3.3 分布式锁服务Chubby

- 一个决议分为两个阶段

1 准备阶段

proposers选择一个提案并将它的编号设为n

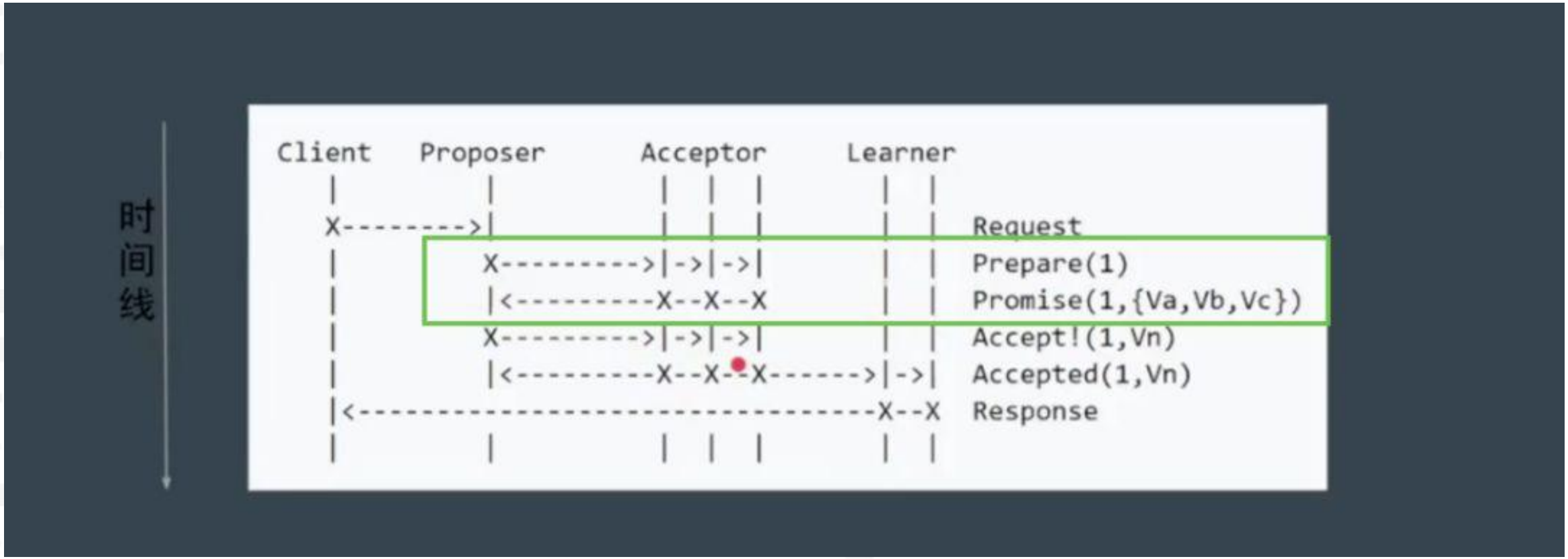
将它发送给acceptors中的一个“多数派”

acceptors 收到后，如果提案的编号大于它已经回复的所有消息，则acceptors将自己上次的批准回复给proposers，并不再批准小于n的提案。

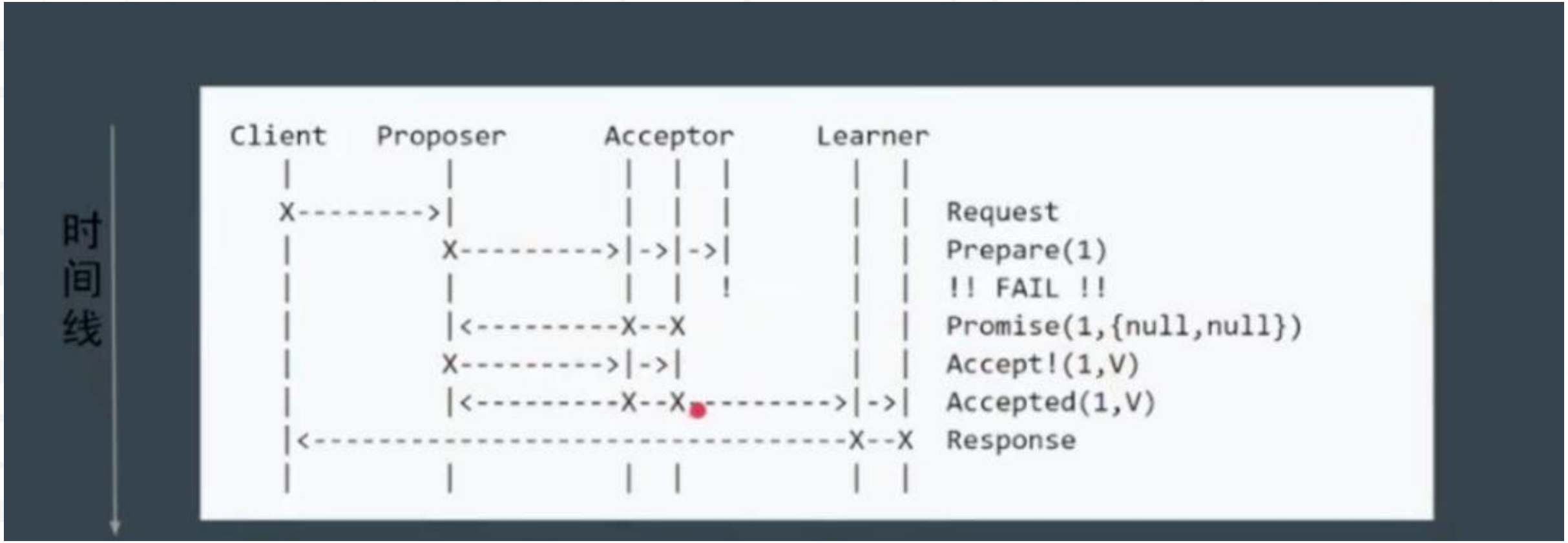
2 批准阶段

当proposers接收到acceptors 中的这个“多数派”的回复后，就向回复请求的acceptors发送accept请求，在符合acceptors一方的约束条件下，acceptors收到accept请求后即批准这个请求。

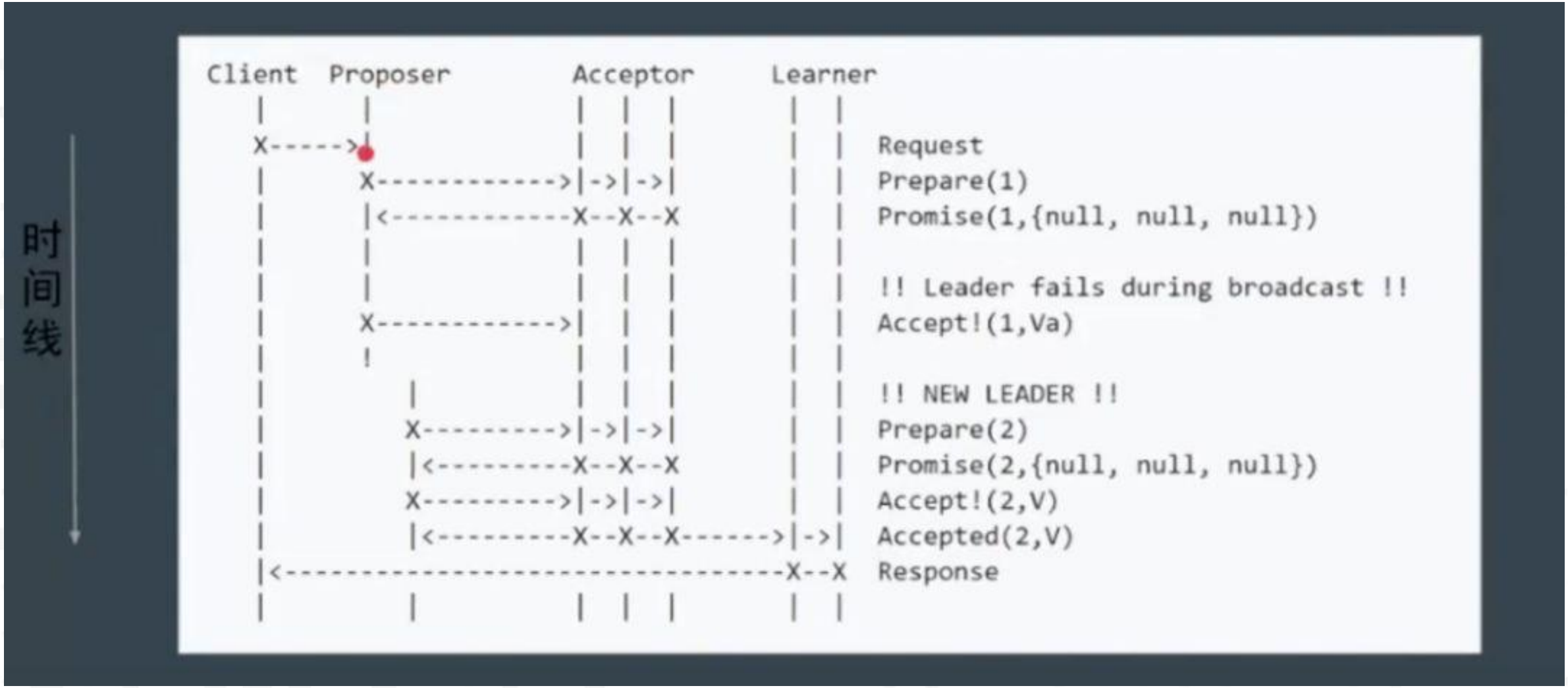
3.3 分布式锁服务Chubby



3.3 分布式锁服务Chubby

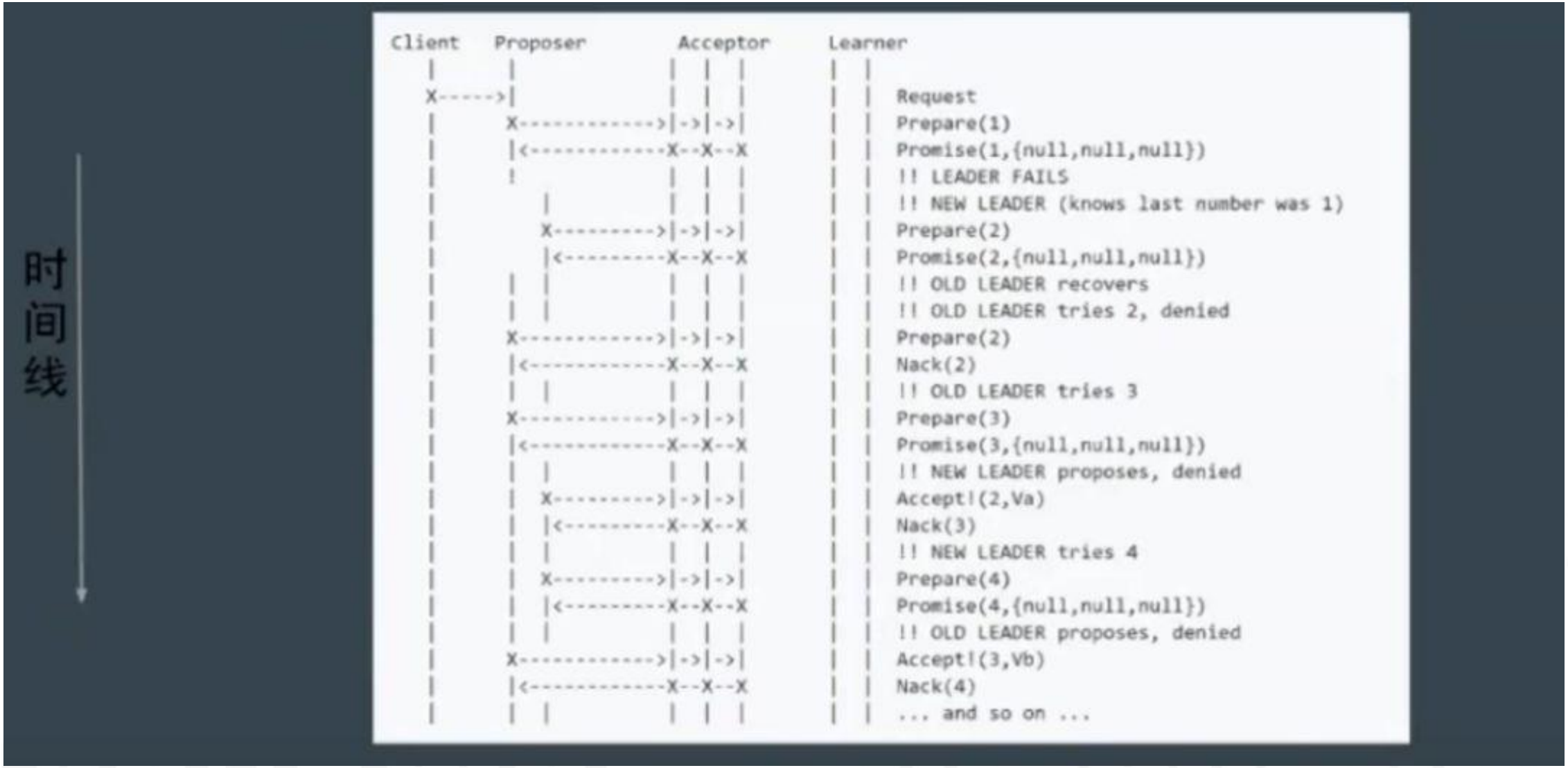


3.3 分布式锁服务Chubby



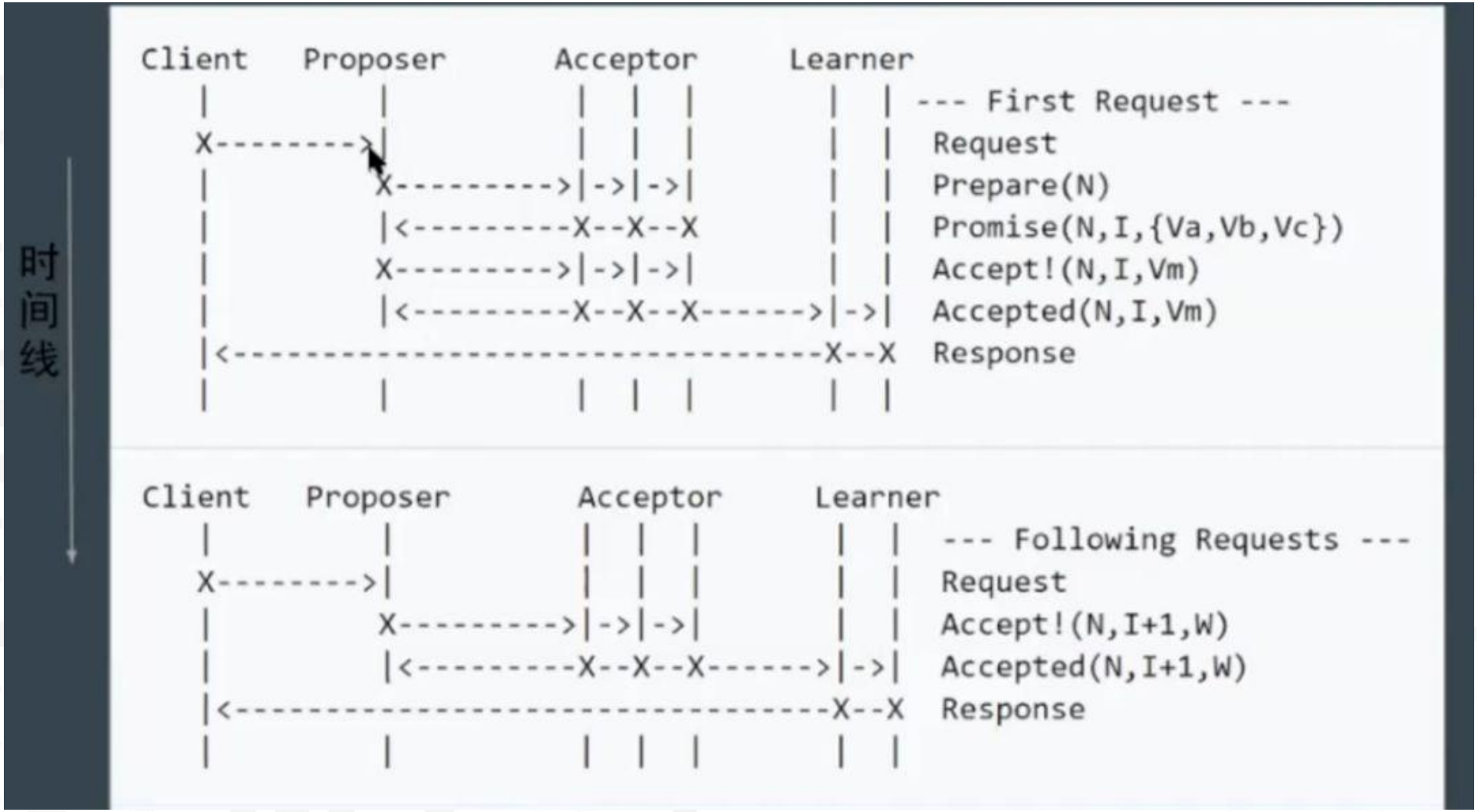
3.3 分布式锁服务Chubby

潜在问题：活锁



3.3 分布式锁服务Chubby

Multi Paxos引入 Leader，也就是唯一的 proposer，所有的请求都需经过此 Leader。

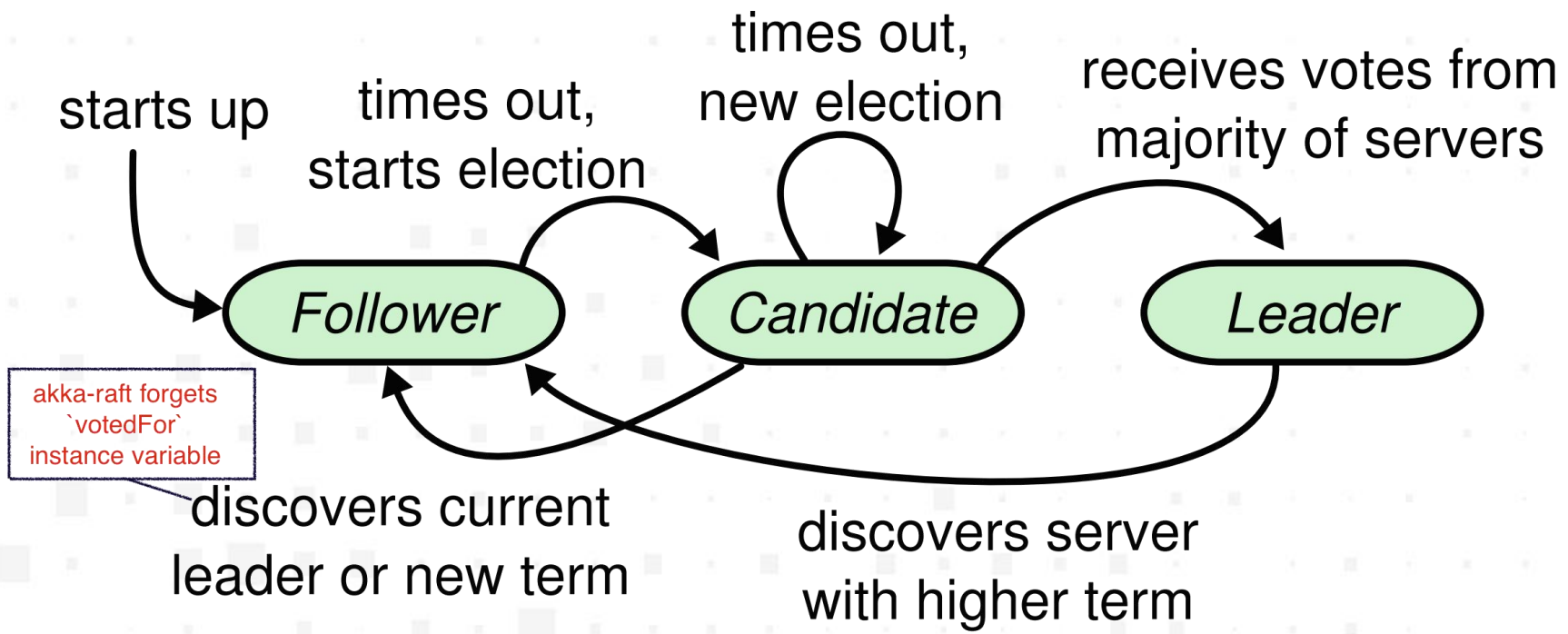


补充：分布式一致性协议Raft

Raft

Raft Demo1

Raft Demo2



补充：Zookeeper的ZAB算法

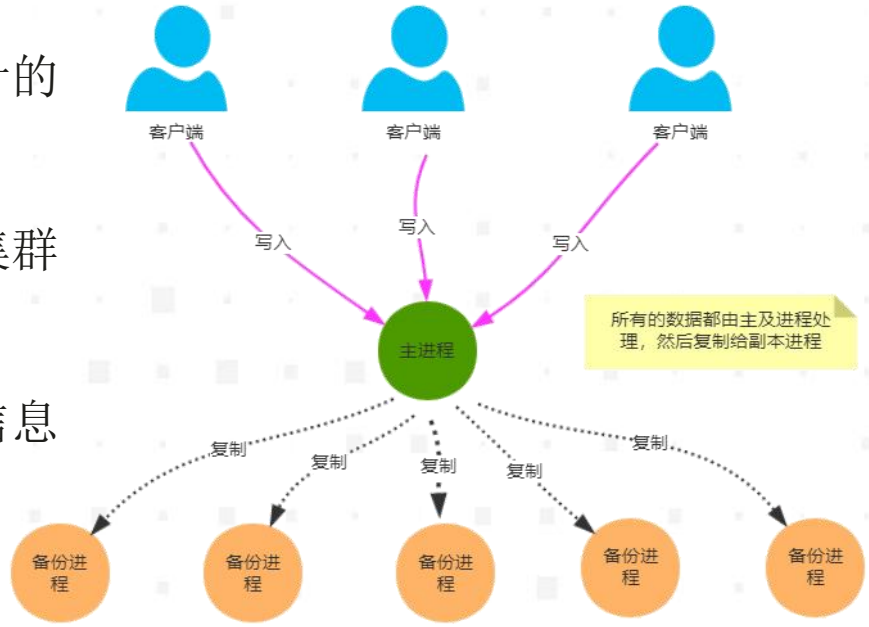
Zookeeper Atomic Broadcast（Zookeeper 原子广播协议）

Zookeeper 是一个为分布式应用提供高效且可靠的分布式协调服务。在解决分布式一致性方面，Zookeeper 并没有使用 Paxos ，而是采用了 ZAB 协议。

ZAB 协议定义：ZAB 协议是为分布式协调服务 Zookeeper 专门设计的一种支持 **崩溃恢复** 和 **原子广播** 协议。

基于该协议，Zookeeper 实现了一种 **主备模式** 的系统架构来保持集群中各个副本之间**数据一致性**。

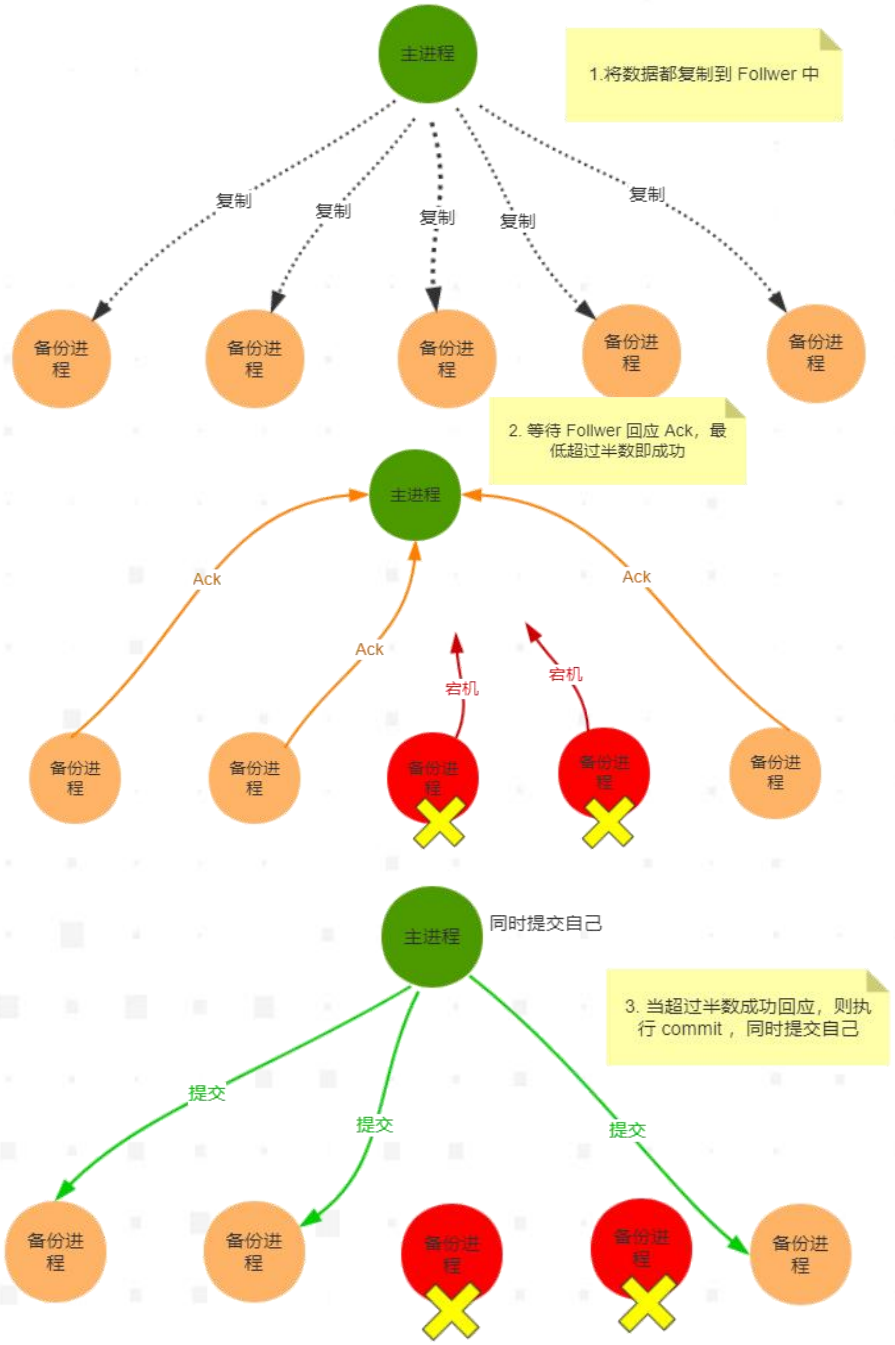
复制过程类似 2PC，ZAB 只需要 Follower 有一半以上返回 Ack 信息就可以执行提交，大大减小了同步阻塞，也提高了可用性。



ZAB算法

消息广播：

1. 将数据都复制到 Follower 中
2. 等待 Follower 回应 Ack，最低超过半数即成功
3. 当超过半数成功回应，则执行 commit ， 同时提交自己



ZAB算法

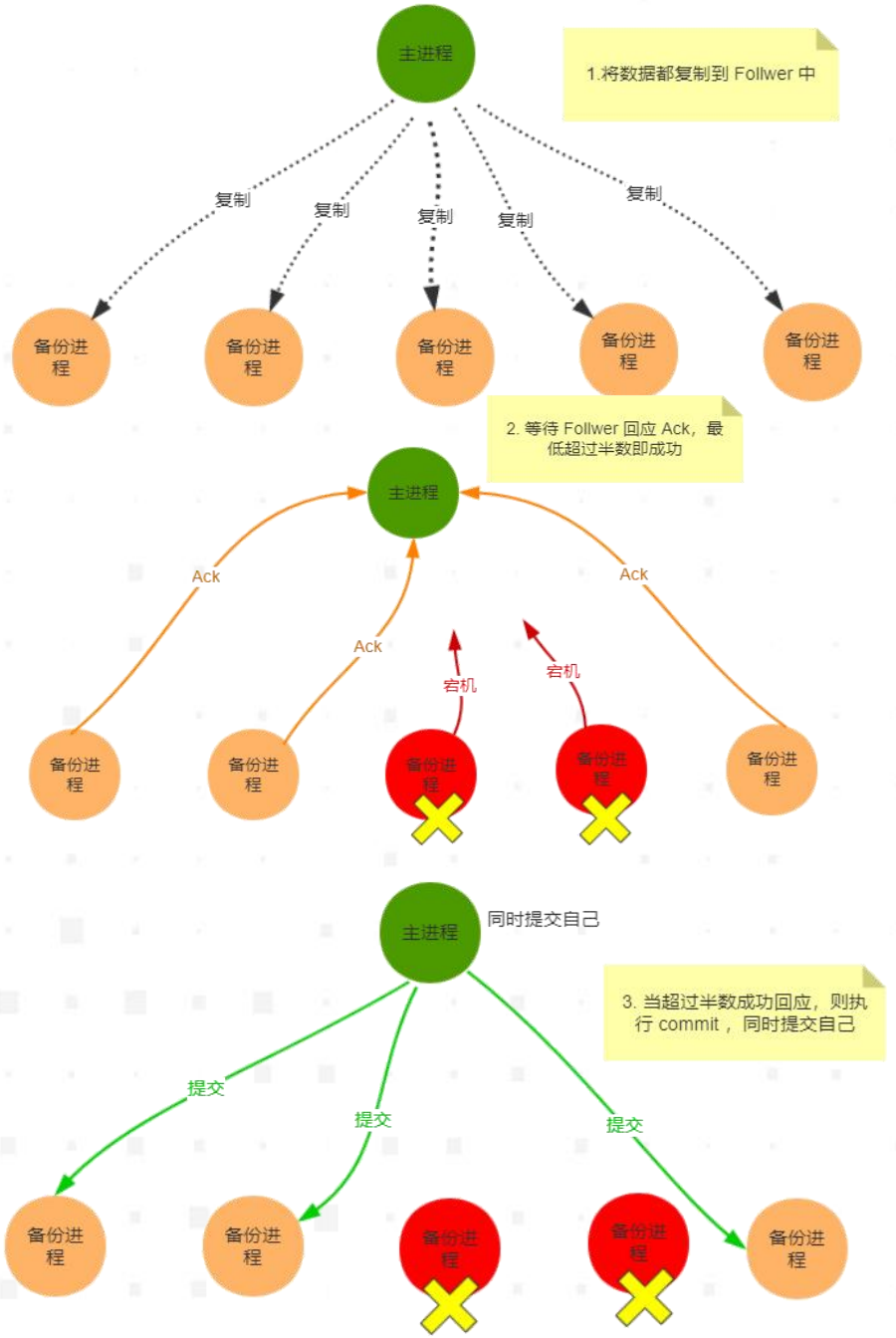
补充细节：

1. Leader 在收到客户端请求之后，会将这个请求封装成一个事务，并给这个事务分配一个全局递增的唯一 ID，称为事务ID（ZXID），ZAB 协议需要保证事务的顺序，因此必须将每一个事务按照 ZXID 进行先后排序然后处理。

2. 在 Leader 和 Follower 之间还有一个消息队列，用来解耦他们之间的耦合，解除同步阻塞。

3. zookeeper集群中为保证任何所有进程能够有序的顺序执行，只能是 Leader 服务器接受写请求，即使是 Follower 服务器接受到客户端的请求，也会转发到 Leader 服务器进行处理。

实际上，这是一种简化版本的 2PC，不能解决单点问题（即 Leader 崩溃问题）。



ZAB算法

如果 Leader 先本地提交了，然后 commit 请求没有发送出去？

1. 假设1：Leader 在复制数据给所有 Follower 之后崩溃，怎么办？
2. 假设2：Leader 在收到 Ack 并提交了自己，同时发送了部分 commit 出去之后崩溃怎么办？

针对这些问题，ZAB 定义了 2 个原则：

1. ZAB 协议确保那些已经在 Leader 提交的事务最终会被所有服务器提交。
2. ZAB 协议确保丢弃那些只在 Leader 提出/复制，但没有提交的事务。

zookeeper集群中为保证任何所有进程能够有序的顺序执行，只能是 Leader 服务器接受写请求，即使是 Follower 服务器接受到客户端的请求，也会转发到 Leader 服务器进行处理。

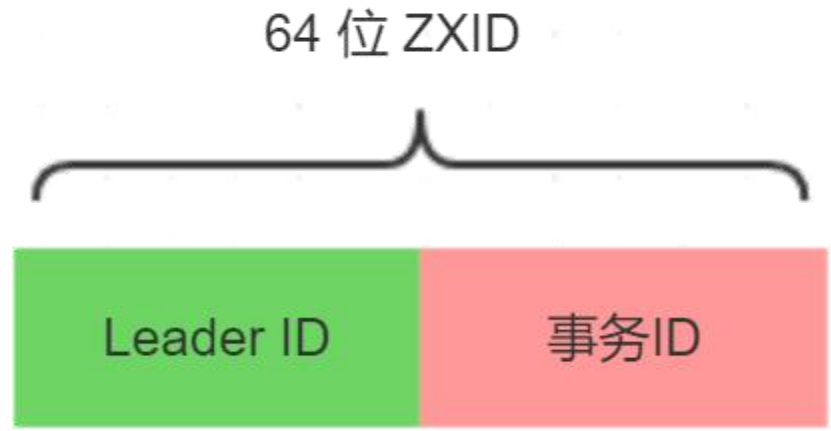
实际上，这是一种简化版本的 2PC，不能解决单点问题。

ZAB算法

Leader 服务器处理或丢弃事务依赖 ZXID

1. 在 ZAB 协议的事务编号 ZXID 设计中，ZXID 是一个 64 位的数字，其中低 32 位可以看作是一个简单的递增的计数器，针对客户端的每一个事务请求，Leader 都会产生一个新的事务 Proposal 并对该计数器进行 + 1 操作。

2. 而高 32 位则代表了 Leader 服务器上取出本地日志中最大事务 Proposal 的 ZXID，并从该 ZXID 中解析出对应的 epoch 值，然后再对这个值加一。

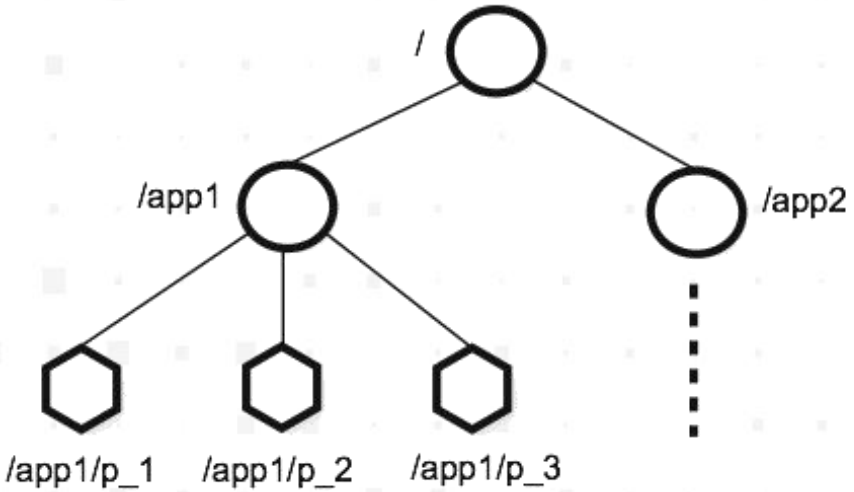


高 32 位代表了每代 Leader 的唯一性，低 32 代表了每代 Leader 中事务的唯一性。同时，也能让 Follwer 通过高 32 位识别不同的 Leader。简化了数据恢复流程。

基于这样的策略：当 Follower 链接上 Leader 之后，Leader 服务器会根据自己服务器上最后被提交的 ZXID 和 Follower 上的 ZXID 进行比对，比对结果要么回滚，要么和 Leader 同步。

Zookeeper

- ZooKeeper 是 Apache 软件基金会的一个软件项目，它为大型分布式计算提供开源的分布式配置服务、同步服务和命名注册。
- ZooKeeper 的架构通过冗余服务实现高可用性。
- Zookeeper 的设计目标是将那些复杂且容易出错的分布式一致性服务封装起来，构成一个高效可靠的原语集，并以一系列简单易用的接口提供给用户使用。
- 一个典型的分布式数据一致性的解决方案，分布式应用程序可以基于它实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能。



目 录

3.1 Google文件系统GFS

3.2 分布式数据处理MapReduce

3.3 分布式锁服务Chubby

3.4 分布式结构化数据表Bigtable

3.4 分布式结构化数据表Bigtable

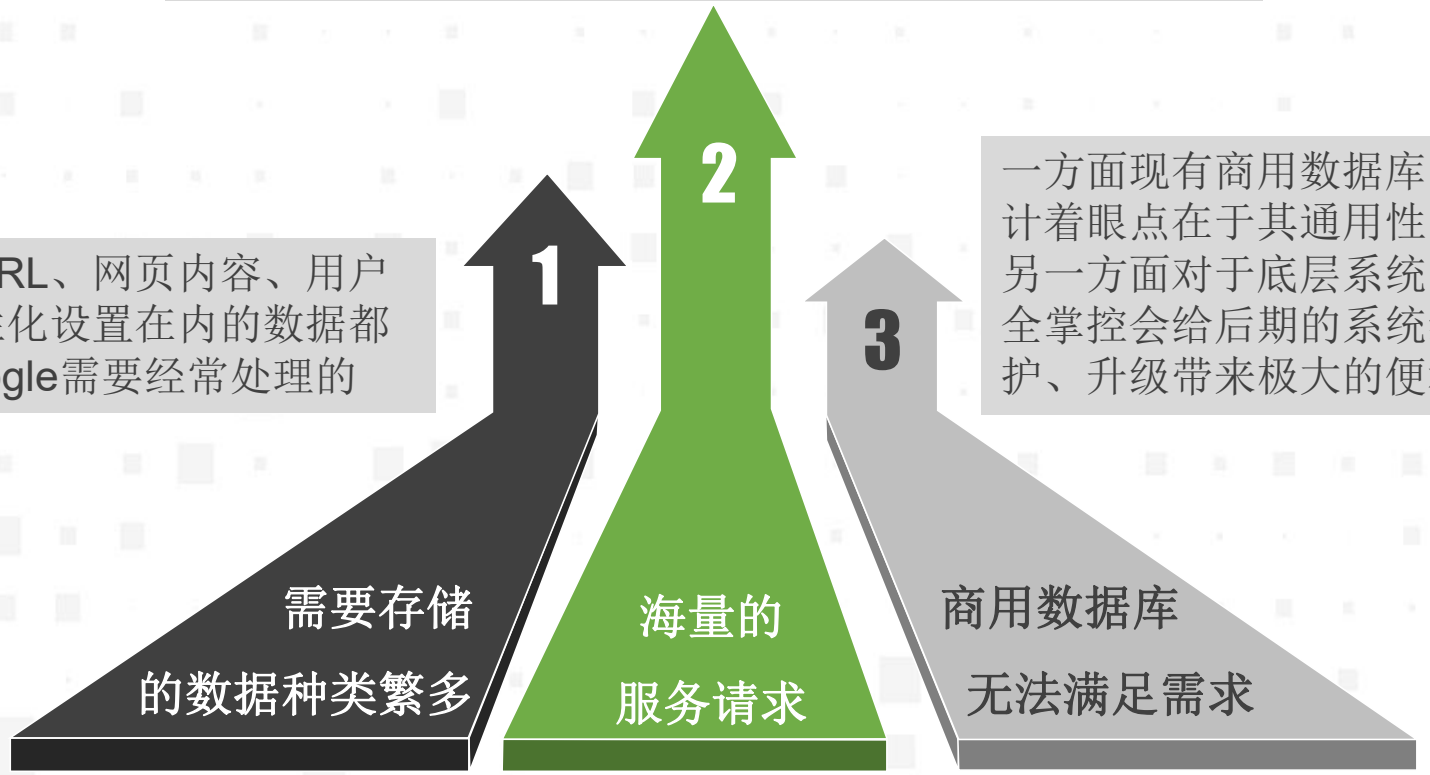
- 3.4.1 设计动机与目标
- 3.4.2 数据模型
- 3.4.3 系统架构
- 3.4.4 主服务器
- 3.4.5 子表服务器
- 3.4.6 性能优化

3.4 分布式结构化数据表Bigtable

- **Bigtable 的设计动机**

Google运行着目前世界上最繁忙的系统，它每时每刻处理的客户服务请求数量是普通的系统根本无法承受的

包括URL、网页内容、用户的个性化设置在内的数据都是Google需要经常处理的



一方面现有商用数据库的设计着眼点在于其通用性。另一方面对于底层系统的完全掌控会给后期的系统维护、升级带来极大的便利

3.4 分布式结构化数据表Bigtable

- **Bigtable 应达到的基本目标**

广泛的适用性	Bigtable是为了满足一系列Google产品而并非特定产品的存储要求。
很强的可扩展性	根据需要随时可以加入或撤销服务器
高可用性	确保几乎所有的情况下系统都可用
简单性	底层系统的简单性既可以减少系统出错的概率，也为上层应用的开发带来便利

3.4 分布式结构化数据表Bigtable

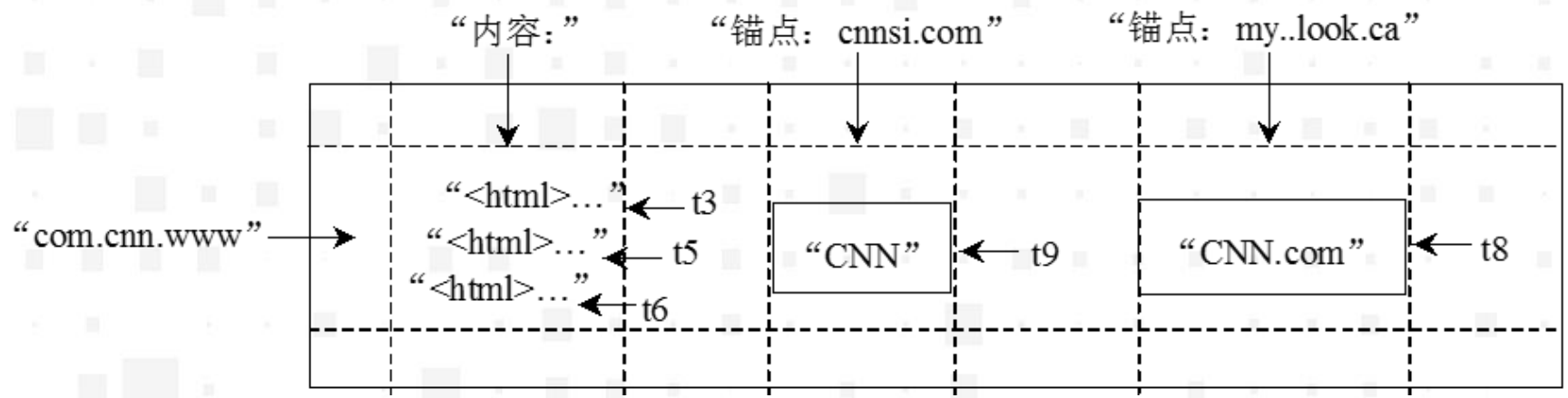
- 3.4.1 设计动机与目标
- 3.4.2 数据模型
- 3.4.3 系统架构
- 3.4.4 主服务器
- 3.4.5 子表服务器
- 3.4.6 性能优化

3.4 分布式结构化数据表Bigtable

- **Bigtable数据的存储格式**

Bigtable是一个分布式多维映射表，表中的数据通过一个行关键字（Row Key）、一个列关键字（Column Key）以及一个时间戳（Time Stamp）进行索引

Bigtable的存储逻辑可以表示为：
(row:string, column:string, time:int64)→string



3.4 分布式结构化数据表Bigtable

行

- Bigtable的行关键字可以是任意的字符串，但是大小不能够超过64KB
- 表中数据都是根据行关键字进行排序的，排序使用的是词典序
- 同一地址域的网页会被存储在表中的连续位置
- 倒排便于数据压缩，可以大幅提高压缩率

列

- 将其组织成所谓的列族（Column Family）
- 族名必须有意义，限定词则可以任意选定
- 组织的数据结构清晰明了，含义也很清楚
- 族同时也是Bigtable中访问控制（Access Control）的基本单元

时间戳

- Google的很多服务比如网页检索和用户的个性化设置等都需要保存不同时间的数据，这些不同的数据版本必须通过时间戳来区分。
- Bigtable中的时间戳是64位整型数，具体的赋值方式可以用户自行定义

3.4 分布式结构化数据表Bigtable

3.4.1 设计动机与目标

3.4.2 数据模型

► 3.4.3 系统架构

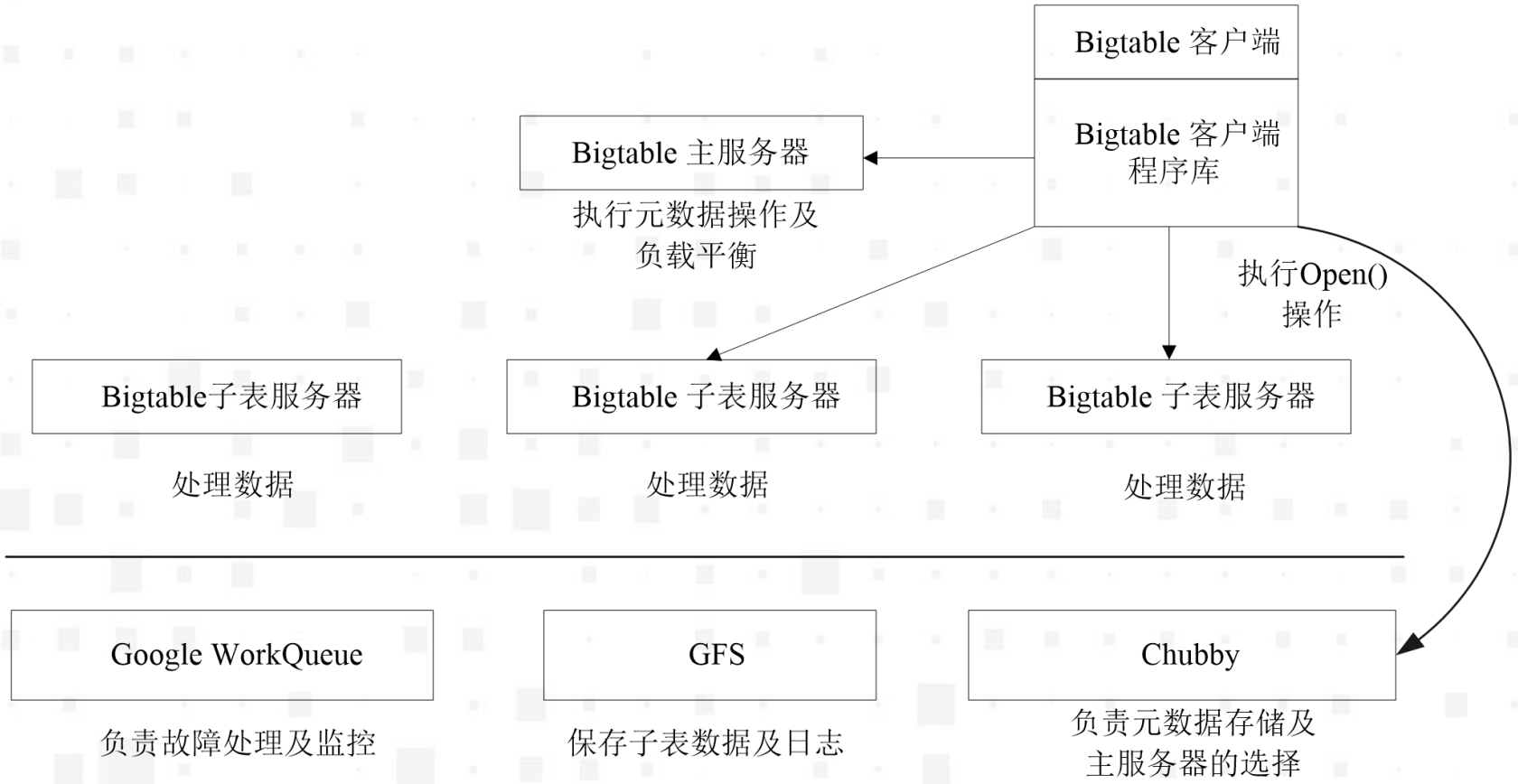
3.4.4 主服务器

3.4.5 子表服务器

3.4.6 性能优化

3.4 分布式结构化数据表Bigtable

- **Bigtable 基本架构**



3.4 分布式结构化数据表Bigtable

- **Bigtable 中 Chubby 的主要作用**

作用一	选取并保证同一时间内只有一个主服务器（Master Server）。
作用二	获取子表的位置信息。
作用三	保存Bigtable的模式信息及访问控制列表。

3.4 分布式结构化数据表Bigtable

3.4.1 设计动机与目标

3.4.2 数据模型

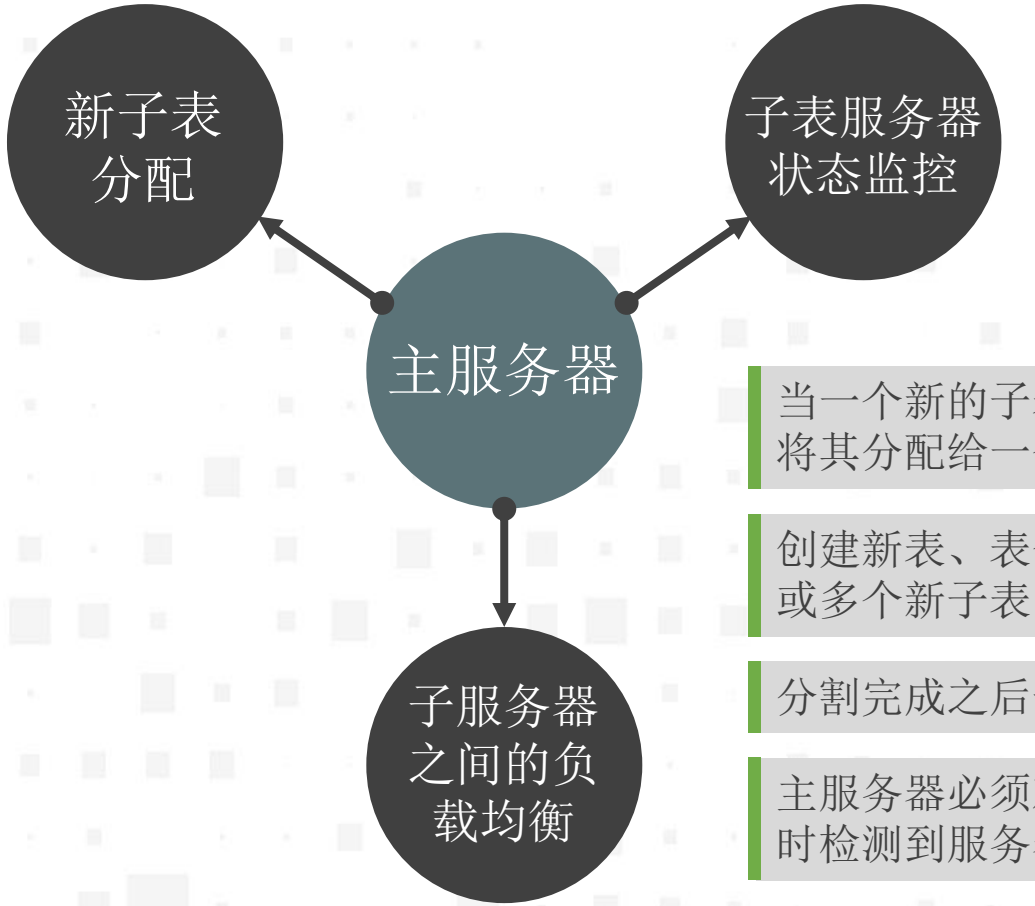
3.4.3 系统架构

► 3.4.4 主服务器

3.4.5 子表服务器

3.4.6 性能优化

3.4 分布式结构化数据表Bigtable



当一个新的子表产生时，主服务器通过一个加载命令将其分配给一个空间足够的子表服务器。

创建新表、表合并以及较大子表的分裂都会产生一个或多个新子表。

分割完成之后子服务器需要向主服务发出一个通知。

主服务器必须对子表服务器的状态进行监控，以便及时检测到服务器的加入或撤销

3.4 分布式结构化数据表Bigtable

- **Bigtable 中 Chubby 的主要作用**

3.4 分布式结构化数据表Bigtable

3.4.1 设计动机与目标

3.4.2 数据模型

3.4.3 系统架构

3.4.4 主服务器

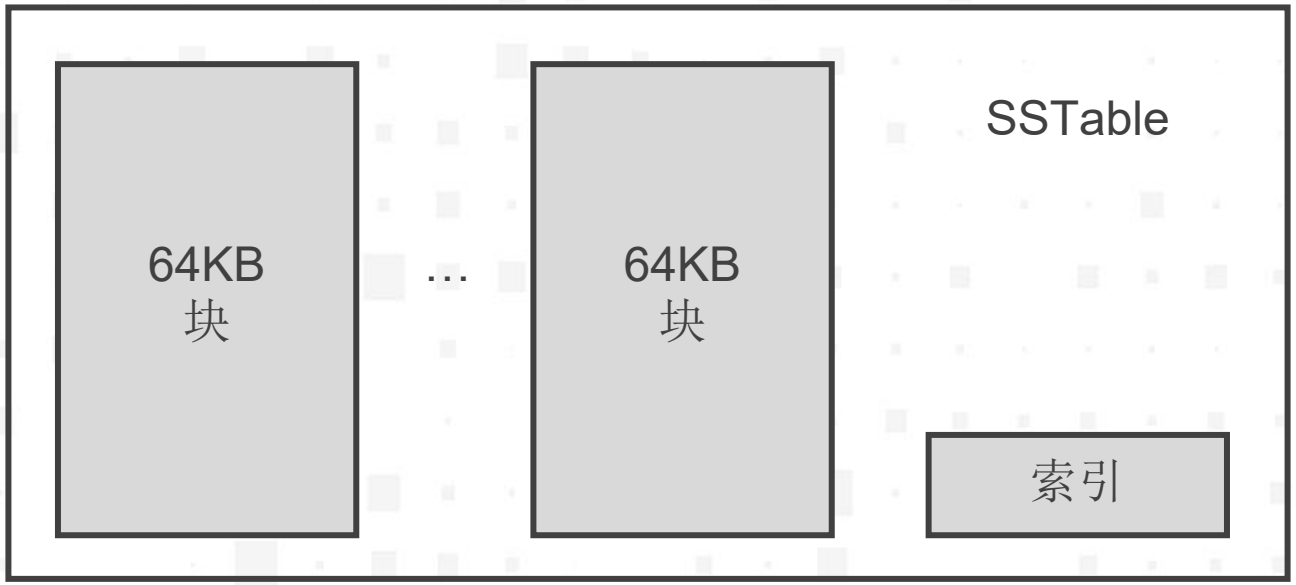
► 3.4.5 子表服务器

3.4.6 性能优化

3.4 分布式结构化数据表Bigtable

- **SSTable 格式的基本示意**

SSTable是Google为Bigtable设计的内部数据存储格式。所有的SSTable文件都存储在GFS上，用户可以通过键来查询相应的值。



3.4 分布式结构化数据表Bigtable

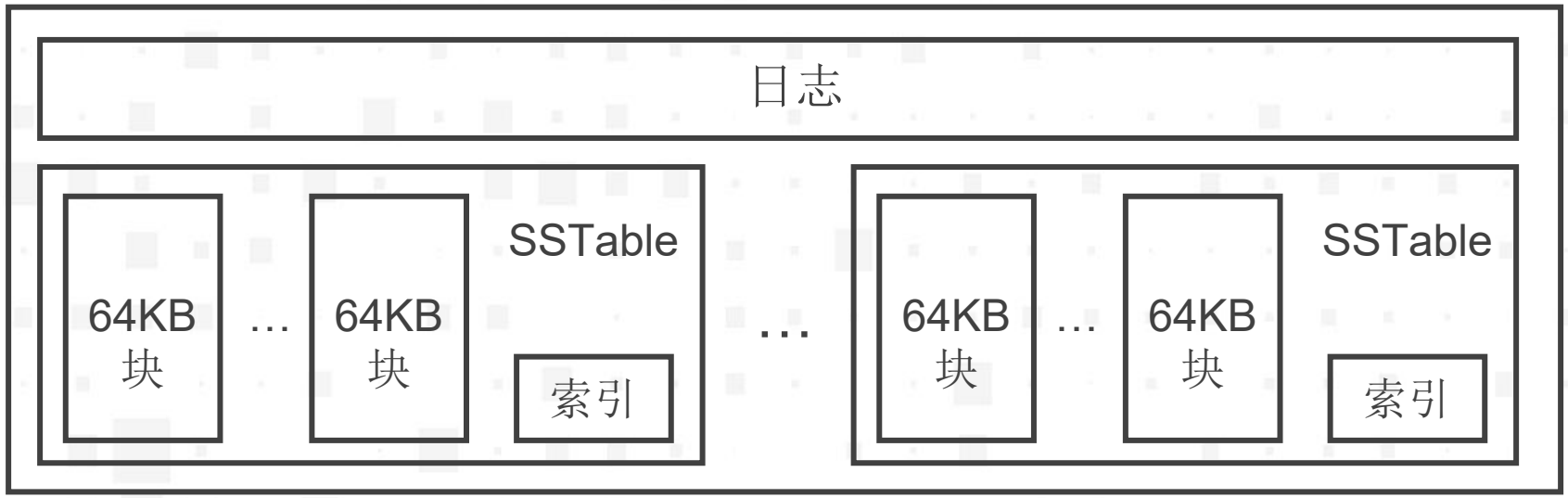
- 子表实际组成

不同子表的SSTable可以共享

每个子表服务器上仅保存一个日志文件

Bigtable规定将日志的内容按照键值进行排序

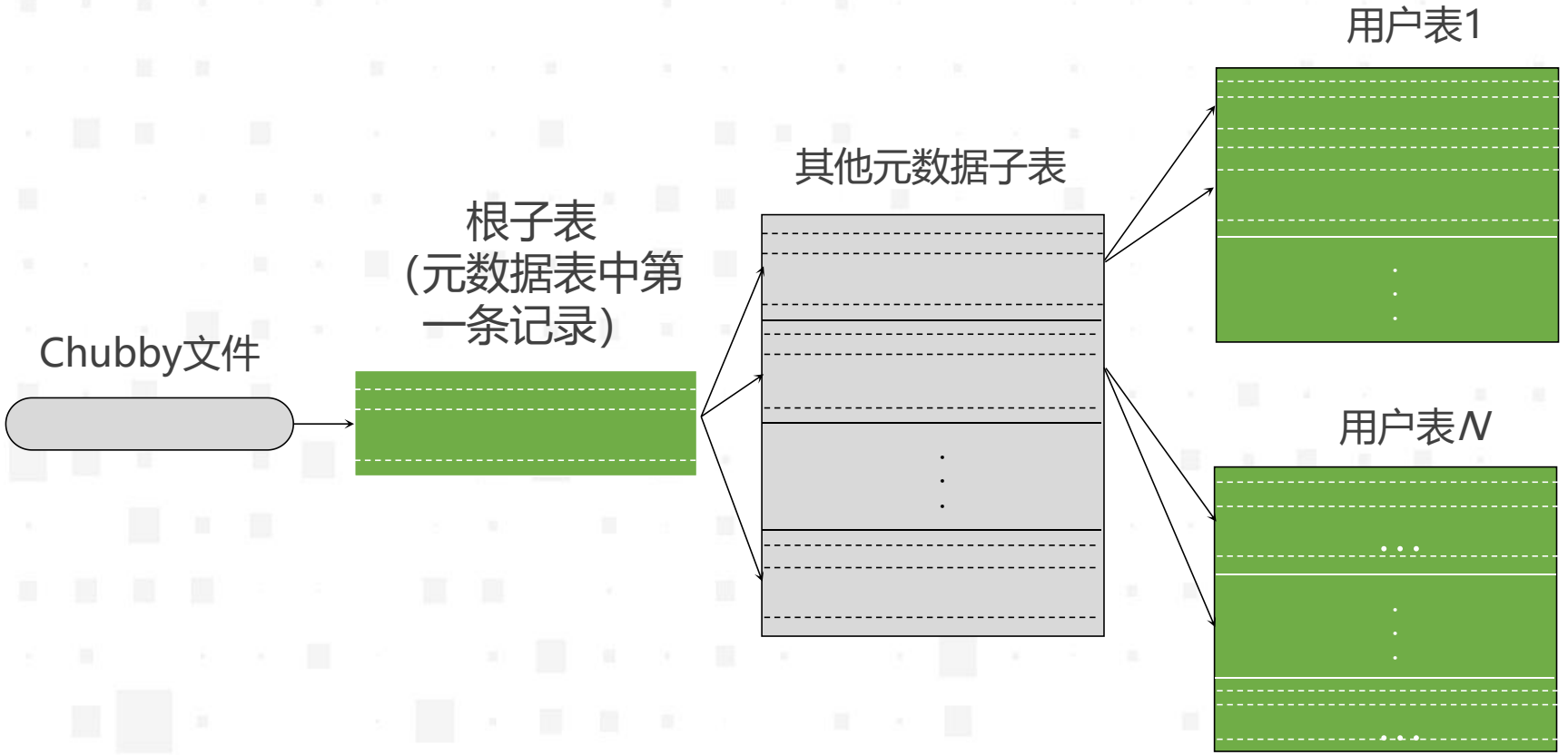
每个子表服务器上保存的子表数量可以从几十到上千不等，通常情况下是100个左右



3.4 分布式结构化数据表Bigtable

- 子表地址组成

Bigtable系统的内部采用的是一种类似B+树的三层查询体系

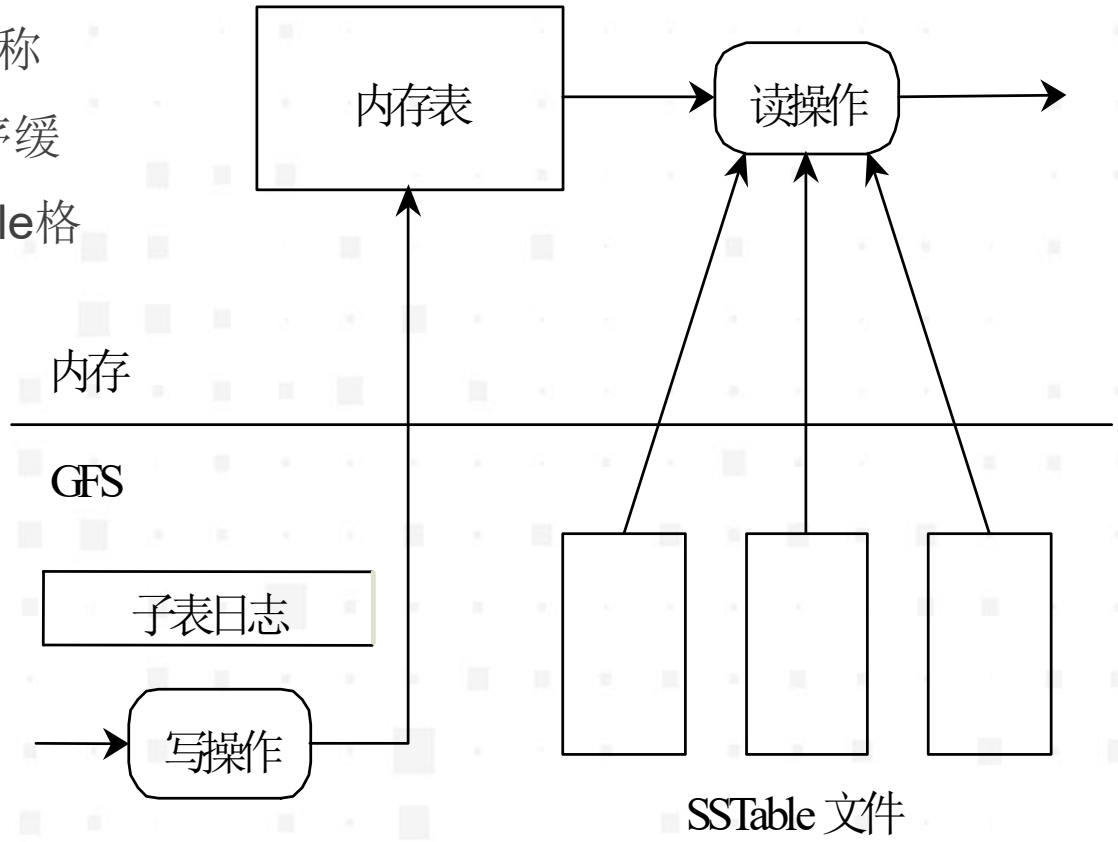


3.4 分布式结构化数据表Bigtable

- **Bigtable 数据存储及读/写操作**

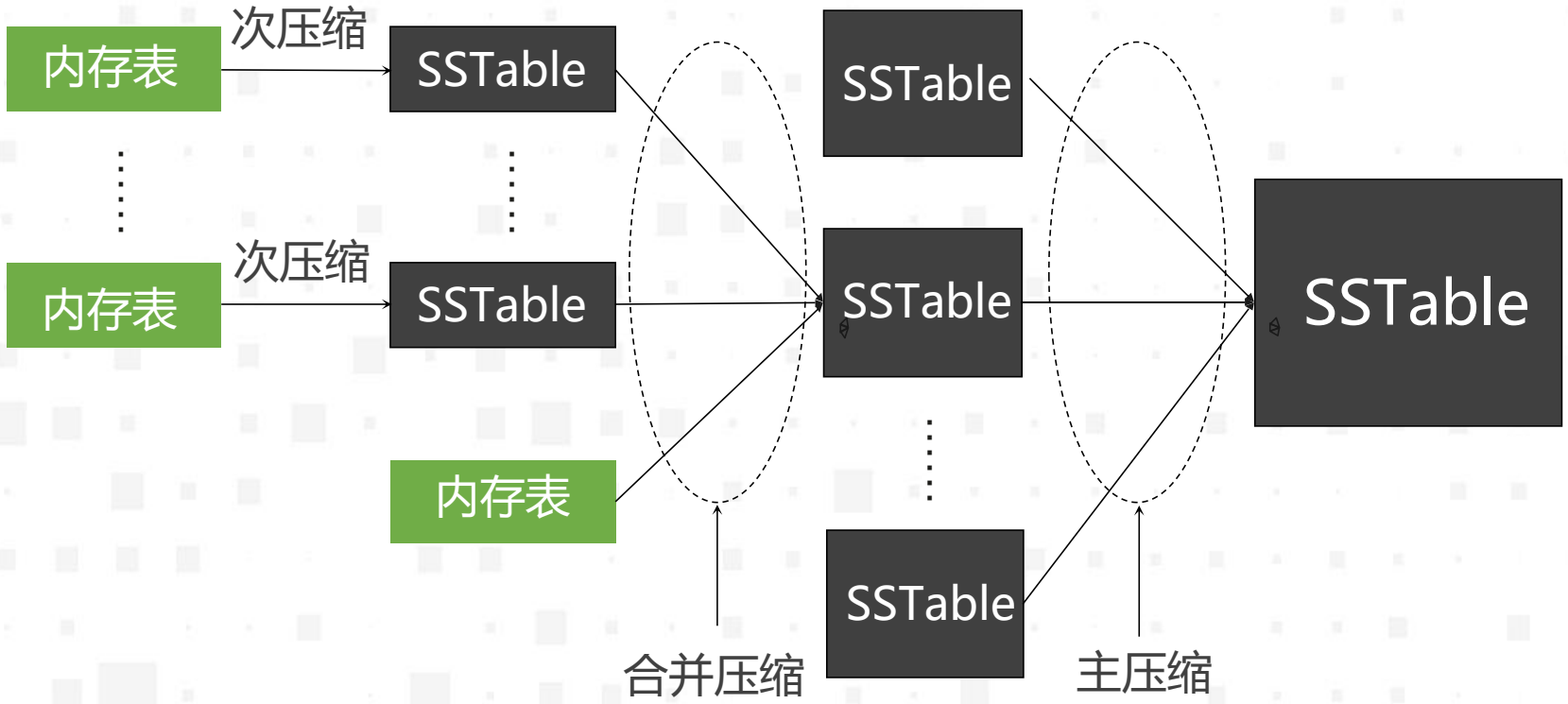
较新的数据存储在内存中一个称为内存表（Memtable）的有序缓冲里，较早的数据则以SSTable格式保存在GFS中。

读和写操作有很大的差异性



3.4 分布式结构化数据表Bigtable

- 三种形式压缩之间的关系



3.4 分布式结构化数据表Bigtable

3.4.1 设计动机与目标

3.4.2 数据模型

3.4.3 系统架构

3.4.4 主服务器

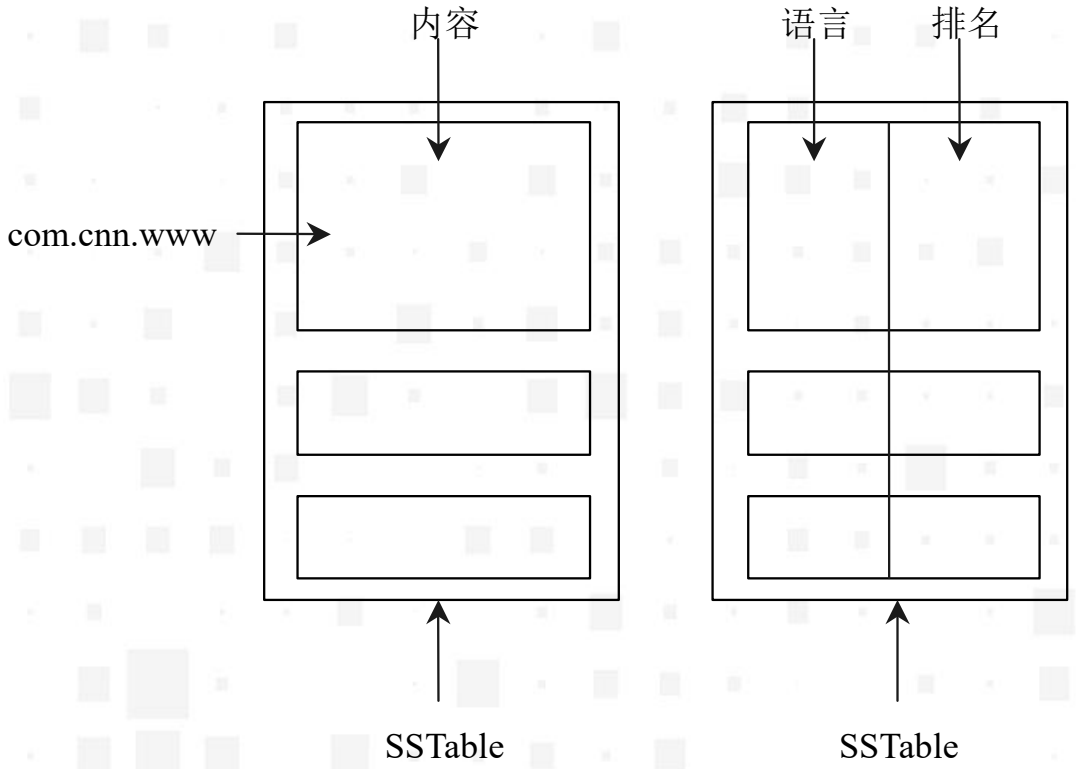
3.4.5 子表服务器

► 3.4.6 性能优化

3.4 分布式结构化数据表Bigtable

- 局部性群组

Bigtable允许用户将原本并不存储在一起的数据以列族为单位，根据需要组织在一个单独的SSTable中，以构成一个局部性群组。



用户可以只看自己感兴趣的内容。

对于一些较小的且会被经常读取的局部性群组，明显地改善读取效率。

3.4 分布式结构化数据表Bigtable

- 压缩

压缩可以有效地节省空间，Bigtable中的压缩被应用于很多场合。首先压缩可以被用在构成局部性群组的SSTable中，可以选择是否对个人的局部性群组的SSTable进行压缩。

1. 利用Bentley & McIlroy方式（BMDiff）在大的扫描窗口将常见的长串进行压缩

2. 采取Zippy技术进行快速压缩，它在一个16KB大小的扫描窗口内寻找重复数据，这个过程非常快

3.4 分布式结构化数据表Bigtable

- 布隆过滤器

Bigtable向用户提供了一种称为**布隆过滤器**的数学工具。布隆过滤器是巴顿•布隆在**1970**年提出的，实际上它是一个很长的**二进制向量**和一系列**随机映射函数**，在读操作中确定子表的位置时非常有用。

优点	缺点
布隆过滤器的速度快，省空间	在某些情况下它会将不存在的子表判断为存在
不会将一个存在的子表判定为不存在	

BloomFilter 计算 计算器

假设m 是该bit数组的大小，k 是哈希函数的个数，n 是插入的元素的个数。

- 假设hash函数以等概率条件选择并设置bit位为“1”，则其概率为 $\frac{1}{m}$
- 在进行元素插入时的hash操作中没有被置为1的概率是 $1-\frac{1}{m}$
- 在经过k个哈希函数之后，该位仍然没有被置“1”的概率是 $(1-\frac{1}{m})^{k}$ ^{kn}
- 插入了n 个元素，该位仍然没有被置“1”的概率是 $(1-\frac{1}{m})^{kn}$
- 该位被置“1”的概率是 $1-(1-\frac{1}{m})^{kn}$
- 检测某一元素是否在该集合中，则表明需要判断是否所有hash值对应的位都置1，但是该方法可能会错误的认为原本不在集合中的元素是在BloomFilter中的，即导致误判率的发生 $[1-(1-\frac{1}{m})^{kn}]^{k}$

$$P(error) = f(k) = (1-e^{\frac{kn}{m}})^{k} = 2^{ln2\cdot\frac{m}{n}} \approx 0.6158\cdot\frac{m}{n}$$

$\lim_{x->\infty}(1-\frac{1}{x})^{-x}=e$ $[1-(1-\frac{1}{m})^{kn}]^{k} \approx (1-e^{-\frac{kn}{m}})^{k}$

Bloom Filter优缺点

1、优点

- 布隆过滤器本质上是一种数据结构，是一种比较巧妙的概率型数据结构
- 插入和查询非常高效，占用空间少（只需要m个bit位）

2、缺点

- 其具有一定概率的误判性（False Positive），即Bloom Filter认为存在的东西很有可能不存在。
- 若不进行计数操作，BloomFilter无法进行删除操作。

Bloom Filter的应用

- 网络缓存：可以用于加速网络缓存，以避免不必要的网络流量。例如，Google Chrome浏览器使用Bloom filter来判断一个URL是否已经被缓存，如果已经缓存，则可以直接从本地获取，而无需从网络下载。
- 搜索引擎：可以用于快速判断一个词语是否出现在索引中。例如，Elasticsearch搜索引擎使用Bloom filter来加速查询操作。
- 恶意软件检测：可以用于快速判断一个文件是否为恶意软件。例如，ClamAV杀毒软件使用Bloom filter来快速扫描计算机上的文件。
- 数据库查询：可以用于快速判断一个值是否存在于数据库中。例如，Apache Cassandra、RocksDB、LevelDB数据库使用Bloom filter来优化查询操作。
- 分布式系统：可以用于快速判断一个键是否存在于分布式系统中。例如，Redis分布式缓存系统使用Bloom filter来避免在集群中的所有节点上进行无意义的查询操作。
- 垃圾邮件过滤：可以用于快速判断一封电子邮件是否为垃圾邮件。例如，SpamAssassin反垃圾邮件软件使用Bloom filter来过滤垃圾邮件。

3.5 HBase数据库

Hbase是基于Hadoop的开源分布式数据库，它以Google的BigTable为原型，设计并实现了具有 高可靠性 高性能 列存储 可伸缩 实时读写 的分布式数据库系统。

- HBase适合于存储非结构化数据
- Hbase是基于列的而不是基于行的模式
- Hbase在Hadoop之上提供了类似于BigTable的能力

3.5 HBase数据库

- **Hbase数据模型**

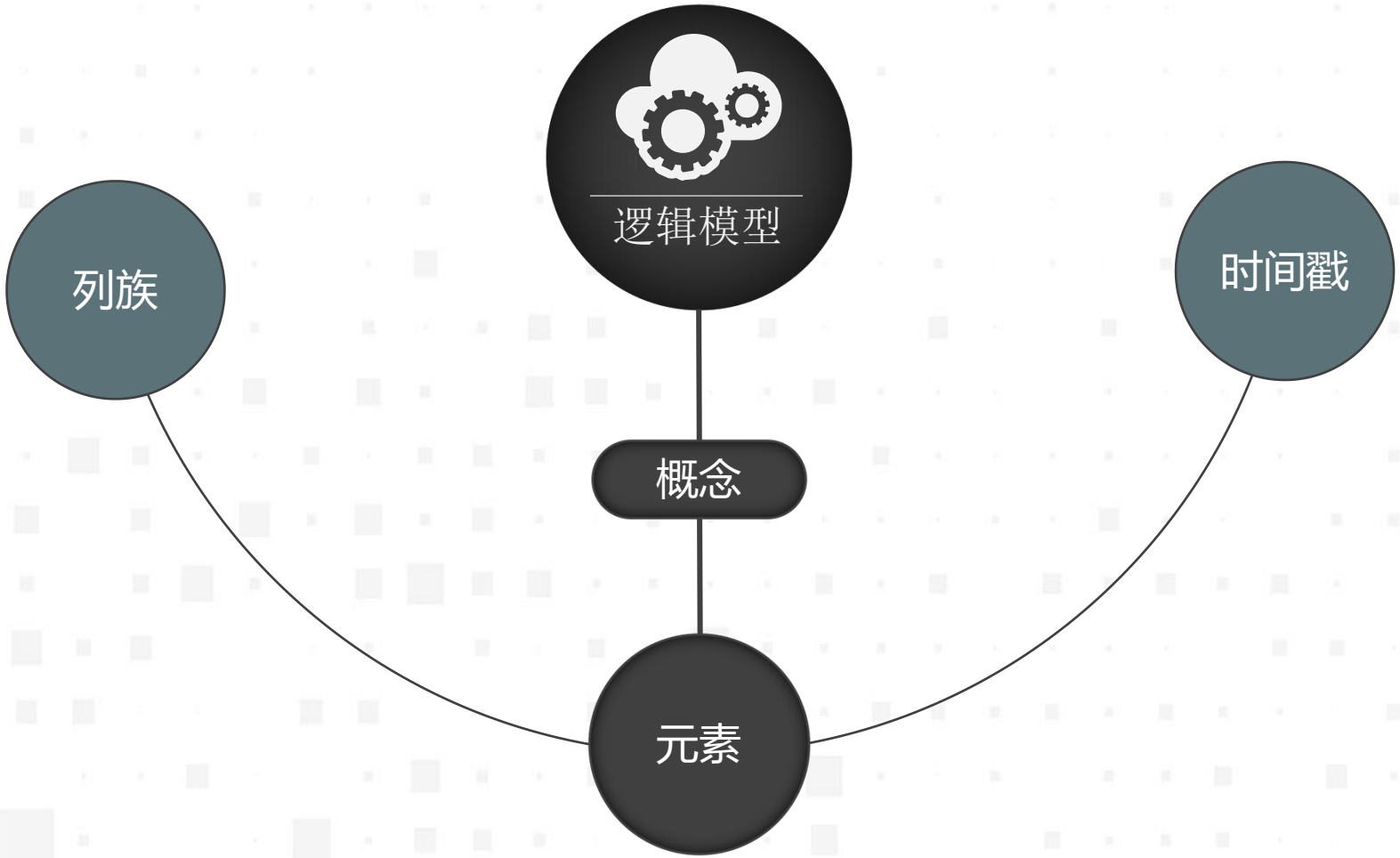
数据库一般以“表”的形式存储结构化数据

Hbase也以“表”的形式存储数据

数据的逻辑模型	用户对数据的组织形式
数据的物理模型	Hbase里数据在HDFS上的具体存储形式

3.5 HBase数据库

Hbase数据模型



行和列确定的存储单元

3.5 HBase数据库

Hbase数据模型

- 表中仅有一行数据，行的唯一标识为com.cnn.www，对这行数据的每一次逻辑修改都有一个时间戳关联对应。
- 表中共有四列：contents:html，anchor:cnnsi.com，anchor:my.look.ca，mime:type，每一列以前缀的方式给出其所属的列族。

行键	时间戳	列族contents	列族anchor	列族mime
"com.cnn.www"	t9		anchor:cnnsi.com="CNN"	
	t8		anchor:my.look.ca="CNN.com"	
	t6	contents:html="<html>…"		mime:type="text/html"
	t5	contents:html="<html>…"		
	t6	contents:html="<html>…"		

3.5 HBase数据库

• Hbase数据模型

行键是数据行在表中的唯一标识，并作为检索记录的主键。

在Hbase中访问表中的行有三种方式

- 通过单个行键访问
- 给定行键的范围访问
- 全表扫描

Hbase提供了两个版本的回收方式：

1. 对每个数据单元，只存储指定个数的最新版本
2. 保存最近一段时间内的版本（如七天），客户端可以按需查询

元素由行键、列（<列族>:<限定符>）和时间戳唯一确定，元素中的数据以字节码的形式存储，没有类型之分。

3.5 HBase数据库

Hbase数据模型

概念模型中的一个行进行分割
并按照列族存储

表中的空值是不被存储的

如果没有指名时间戳，则返回指定列的最新数据值

可以随时向表中的任何一个列添加新列，而不需要事先声明

3.5 HBase数据库

- Hbase数据模型

行键	时间戳	列族contents
"com.cnn.www"	t6	contents:html="<html>…"
	t5	contents:html="<html>…"
	t3	contents:html="<html>…"
行键	时间戳	列族anchor
"com.cnn.www"	t9	anchor:cnnsi.com= "CNN"
	t8	anchor:my.look.ca= "CNN.com"
行键	时间戳	列族mime
"com.cnn.www"	t6	mime:type="text/html"

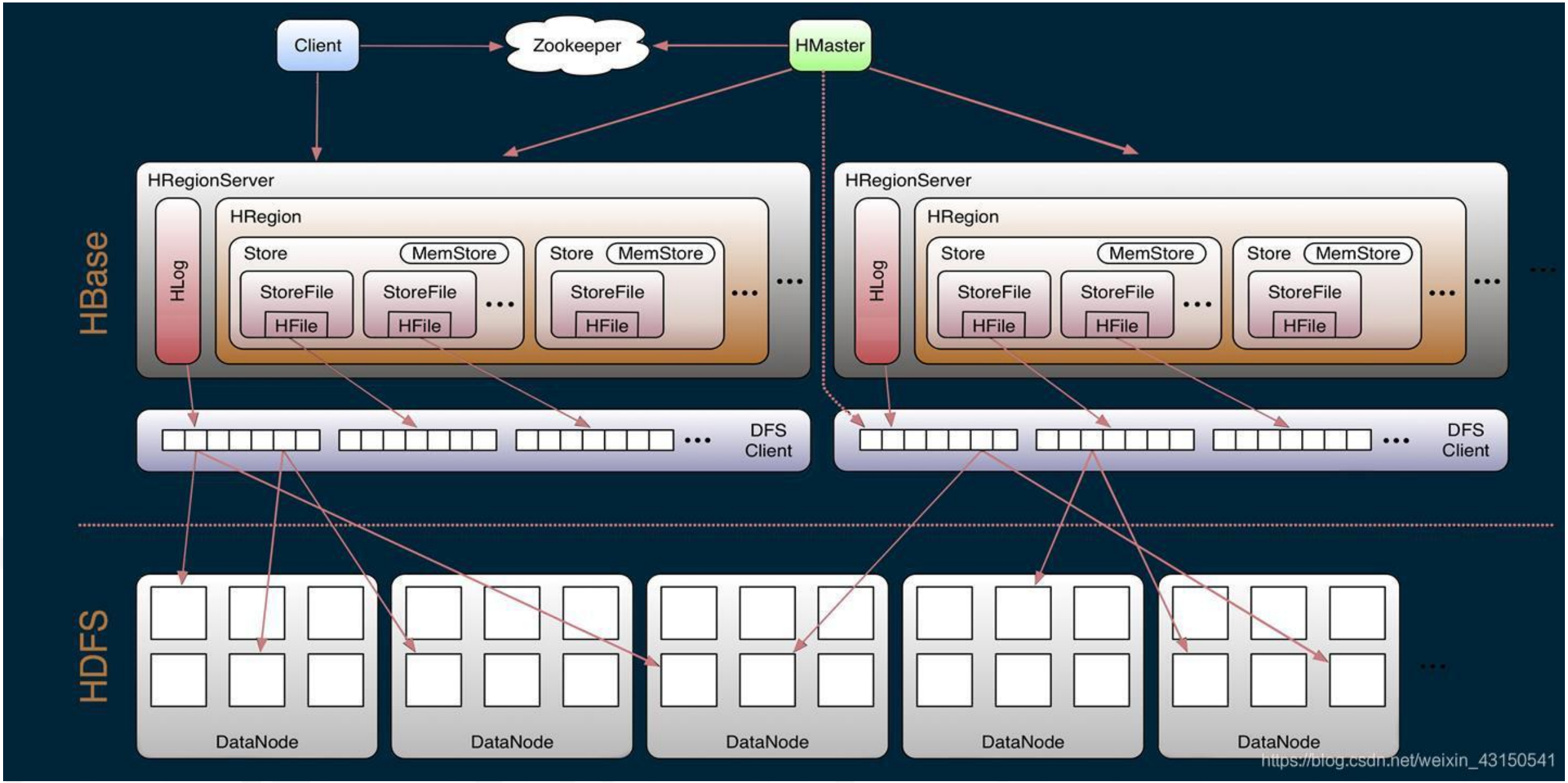
3.5 HBase数据库

Hbase采用master/slave架构

主节点运行的服务称为HMaster

从节点服务称为HRegionServer

底层采用HDFS存储数据


Client
Zookeeper
HMaster
HBase
HRegionServer
HLog
HRegion
Store
MemStore
StoreFile
HFile
DFS Client
HDFS
DataNode
https://blog.csdn.net/weixin_43150541

3.5 HBase数据库

- **Hbase架构**

1）Client

Client端使用Hbase的RPC机制与HMaster和HRegionServer进行通信

2）ZooKeeper

存储了ROOT表的地址、HMaster的地址和HRegionServer地址

3）HMaster

Hbase主节点，将Region分配给HRegionServer，协调HRegionServer的负载并维护集群状态

4）HRegionServer

HRegionServer主要负责响应用户I/O请求，向HDFS文件系统中读写数据

3.5 HBase数据库

- Hbase部署

1. 部署前提
2. Hbase 部署规划
3. 部署Hbase
4. 配置Hbase
5. HDFS里新建Hbase存储目录
6. 启动Hbase集群

3.5 HBase数据库

- **Hbase接口**

Hbase提供了诸多访问接口，下面简单罗列各种访问接口。

Native Java API	最常规和高效的访问方式，适合Hadoop MapReduce Job并行批处理Hbase表数据。
Hbase Shell	Hbase的命令行工具，最简单的接口，适合管理、测试时使用。
Thrift Gateway	利用Thrift序列化技术，支持C++，PHP，Python等多种语言，适合其他异构系统在线访问Hbase表数据。

3.5 HBase数据库

- **Hbase接口**

【例】按要求完成问题：

①假定MySQL里有member表，要求使用Hbase的Shell接口，在Hbase中新建并存储此表。

②简述Hbase是否适合存储问题①中的结构化数据，并简单叙述Hbase与关系型数据库的区别。

身份ID	姓名	性别	年龄	教育	职业	收入
201401	aa	0	21	e0	p3	m
201402	bb	1	22	e1	p2	l
201403	cc	1	23	e2	p1	m

3.5 HBase数据库

【例】解：

下面将姓名、性别、年龄这三个字段抽象为个人属性（personalAttr），教育、职业、收入抽象为社会属性（socialAttr），personalAttr列族包含name、gender和age三个限定符；同理socialAttr下包含edu、prof、inco三个限定符。

Key行键	Value列键					
	列族personalAttr			列族socialAttr		
身份ID	姓名	性别	年龄	教育	职业	收入
201401	aa	0	21	e0	p3	M
201402	bb	1	22	e1	p2	L
201403	cc	1	23	e2	P1	M

3.5 HBase数据库

按上述思路，iClient上依次执行如下命令：

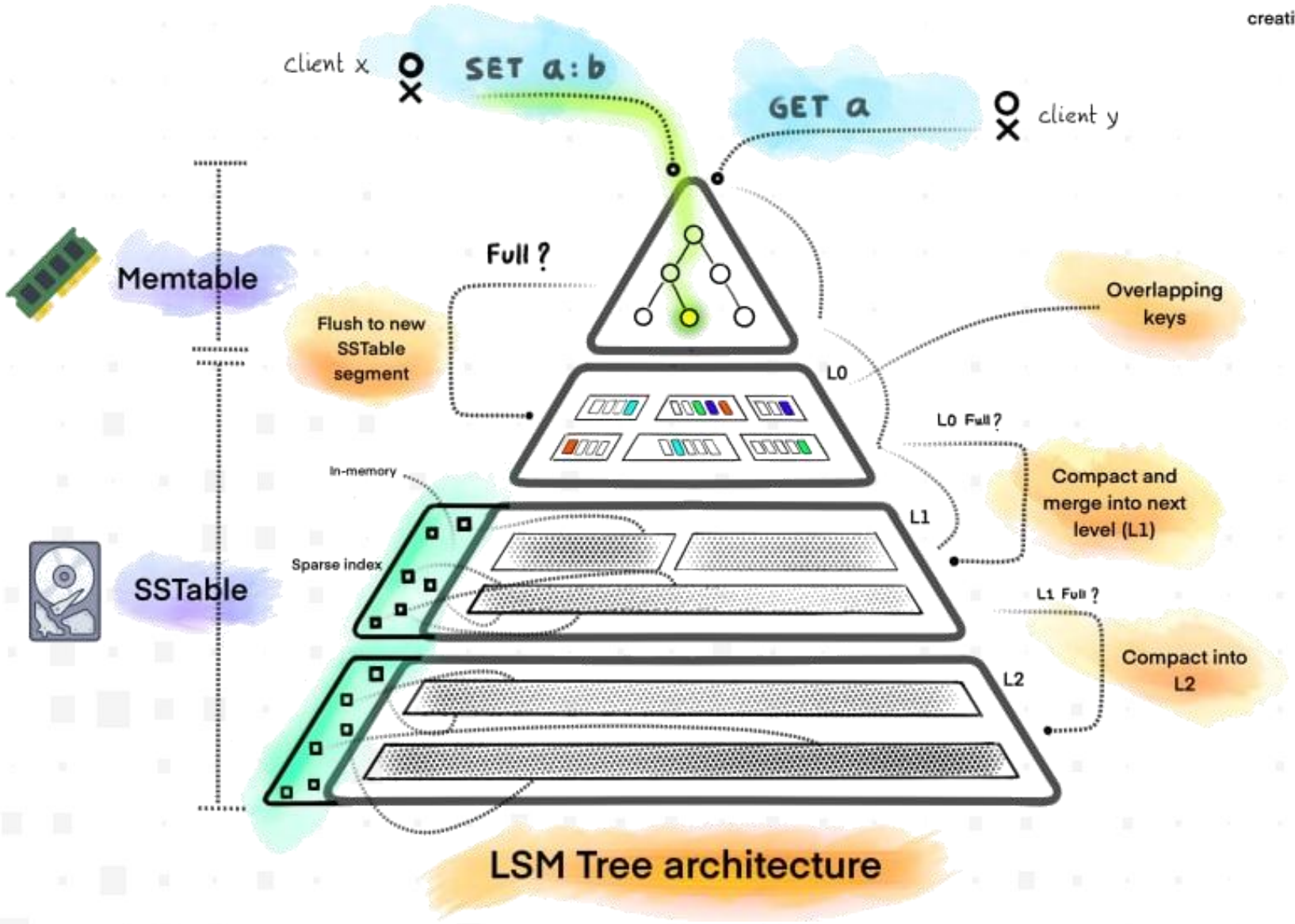
```
[root@iClient ~]# hbase shell                                          #进入Hbase命令行
hbase(main):001:0> list                                                #查看所有表
hbase(main):002:0> create 'member','id','personalAttr','socialAttr'    #创建member表
hbase(main):003:0> list
hbase(main):004:0> scan 'member'                                       #查看member内容
hbase(main):005:0> put 'member','201401','personalAttr:name','aa'      #向member表中插入数据
hbase(main):006:0> put 'member','201401','personalAttr:gender','0'
hbase(main):007:0> put 'member','201401','personalAttr:age','21'
hbase(main):008:0> put 'member','201401','socialAttr:edu','e0'
hbase(main):009:0> put 'member','201401','socialAttr:job','p3'
hbase(main):010:0> put 'member','201401','socialAttr:imcome','m'
hbase(main):011:0> scan 'member'
hbase(main):012:0> disable 'member'                                    #废弃member表
hbase(main):013:0> drop 'member'                                       #删除member表
hbase(main):014:0> quit
```

3.5 HBase数据库

下面简单罗列Hbase和关系型数据库的区别：

1	Hbase只提供字符串这一种数据类型，其他数据类型的操作只能靠用户自行处理，而关系型数据库有丰富的数据类型；
2	Hbase数据操作只有很简单的插入、查询、删除、修改、清空等操作，不能实现表与表关联操作，而关系型数据库有大量此类SQL语句和函数；
3	Hbase基于列式存储，每个列族都由几个文件保存，不同列族的文件是分离的，关系型数据库基于表格设计和行模式保存；
4	Hbase修改和删除数据实现上是插入带有特殊标记的新记录，而关系型数据库是数据内容的替换和修改；
5	Hbase为分布式而设计，可通过增加机器实现性能和数据增长，而关系型数据库很难做到这一点。

LSM-Tree


creativcoder.dev
client x
SET a:b
GET a
client y
Full ?
Memtable
Flush to new SSTable segment
Overlapping keys
L0
L0 Full ?
Compact and merge into next level (L1)
In-memory
L1
Sparse index
SSTable
L1 Full ?
Compact into L2
L2
LSM Tree architecture