Lecture 22 Google 云计算平台

目录

22.1 Google文件系统GFS

22.2 分布式数据处理MapReduce

22.3 分布式锁服务Chubby

22.4 分布式结构化数据表Bigtable

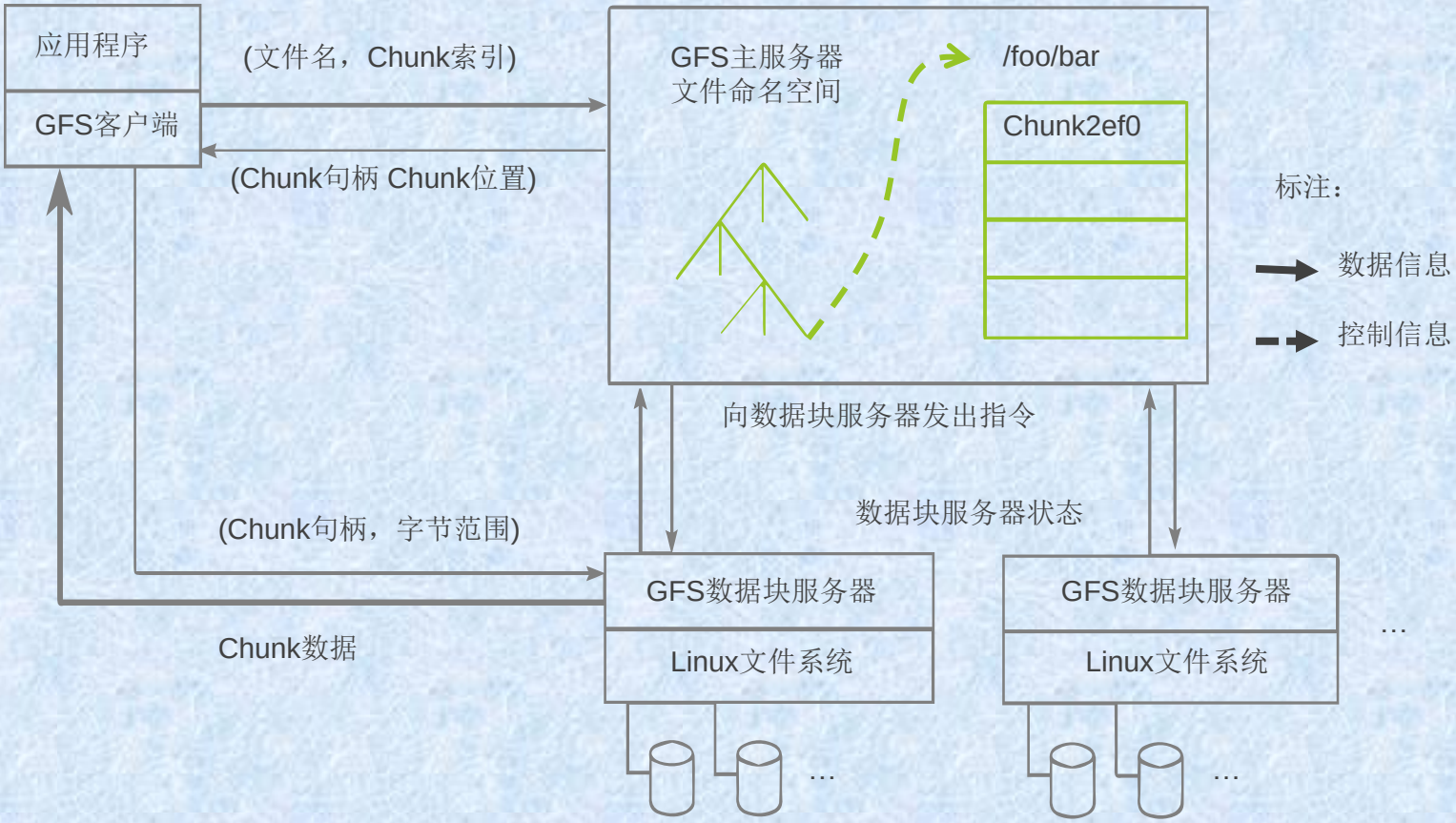
目录

22.1 Google文件系统GFS

- 22.1.1 系统架构
- 22.1.2 容错机制
- 22.1.3 系统管理技术

22.1 Google文件系统GFS

- GFS的系统架构



22.1 Google文件系统GFS

- GFS将整个系统节点分为三类角色

- 客户端首先访问Master节点，获取交互的Chunk Server信息，然后访问这些Chunk Server，完成数据存取工作。这种设计方法实现了控制流和数据流的分离
- Client与Master之间只有控制流，而无数据流
- Client与Chunk Server之间直接传输数据流，同时由于文件被分成多个Chunk进行分布式存储，Client可以同时访问多个Chunk Server，从而使得整个系统的I/O高度并行，系统整体性能得到提高大地降低了Master的负载

目录

22.1 Google文件系统GFS

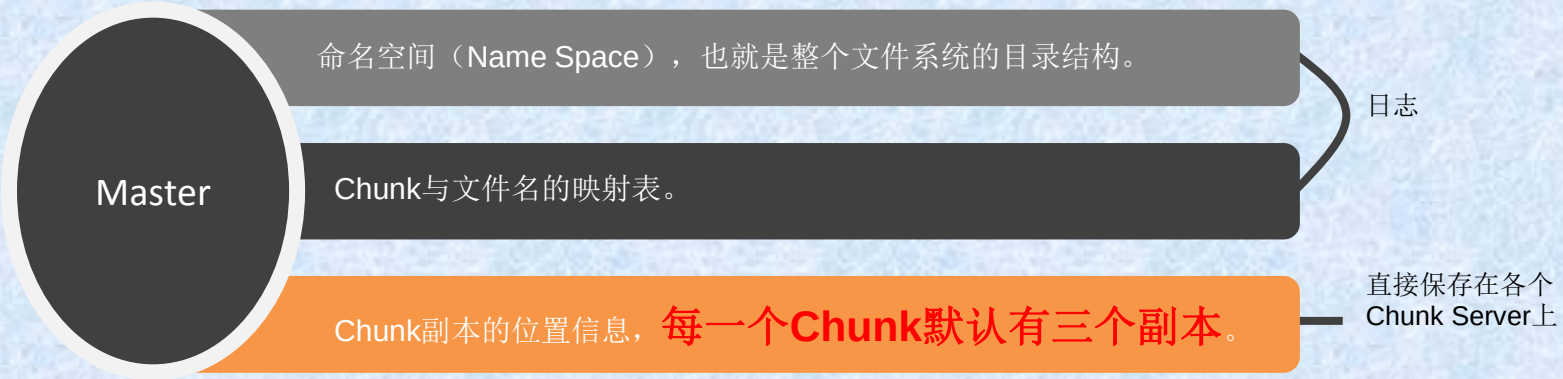
22.1.1 系统架构

22.1.2 容错机制

22.1.3 系统管理技术

22.1 Google文件系统GFS

- **Master容错**



当Master发生故障时，在磁盘数据保存完好的情况下，可以迅速恢复以上元数据的实时备份

22.1 Google文件系统GFS

- **Chunk Server容错**

GFS采用副本的方式实现Chunk Server的容错

每一个Chunk有多个存储副本（默认为三个）

对于每一个Chunk，必须将所有的副本全部写入成功，才视为成功写入

相关的副本出现丢失或不可恢复等情况，Master自动将该副本复制到其他Chunk Server

GFS中的每一个文件被划分成多个Chunk，Chunk的默认大小是**64MB**

每一个Chunk以Block为单位进行划分，大小为64KB，每一个Block对应一个32bit的校验

目录

22.1 Google文件系统GFS

22.1.1 系统架构

22.1.2 容错机制

22.1.3 系统管理技术

22.1 Google文件系统GFS

- 系统管理技术

目录

22.1 Google文件系统GFS
22.2 分布式数据处理MapReduce
22.3 分布式锁服务Chubby
22.4 分布式结构化数据表Bigtable

目录

22.2分布式数据处理MapReduce

- 22.2.1 产生背景
- 22.2.2 编程模型
- 22.2.3 实现机制
- 22.2.4 案例分析

MapReduce产生背景

- MapReduce这种并行编程模式思想最早是在1995年提出的。
- 与传统的分布式程序设计相比，MapReduce封装了并行处理、容错处理、本地化计算、负载均衡等细节，还提供了一个简单而强大的接口。
- MapReduce把对数据集的大规模操作，分发给一个主节点管理下的各分节点共同完成，通过这种方式实现任务的可靠执行与容错机制。

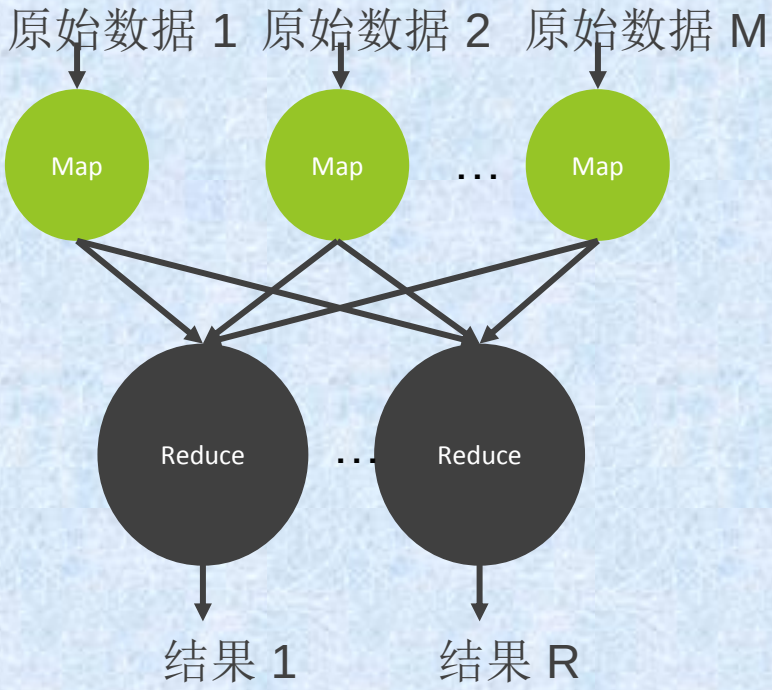
目录

22.2分布式数据处理MapReduce

- 22.2.1 编程模型
- 22.2.2 实现机制
- 22.2.3 案例分析

22.2 分布式数据处理MapReduce

- 编程模型



Map函数——对一部分原始数据进行指定的操作。每个Map操作都针对不同的原始数据，因此Map与Map之间是互相独立的，这使得它们可以充分并行化。

Reduce操作——对每个Map所产生的一部分中间结果进行合并操作，每个Reduce所处理的Map中间结果是互不交叉的，所有Reduce产生的最终结果经过简单连接就形成了完整的结果集.

22.2 分布式数据处理MapReduce

编程模型

Map: (in_key, in_value) → {(key$_j$, value$_j$) | $j = 1 \cdots k$}

Reduce: (key, [value$_1$, …, value$_m$]) → (key, final_value)

Map输入参数：in_key和in_value，它指明了Map需要处理的原始数据

Map输出结果：一组<key,value>对，这是经过Map操作后所产生的中间结果

Reduce输入参数：
（key,[value$_1$,...,value$_m$]）

Reduce工作：
对这些对应相同key的value值进行归并处理

Reduce输出结果：
（key, final_value），所有Reduce的结果并在一起就是最终结果

目录

22.2分布式数据处理MapReduce

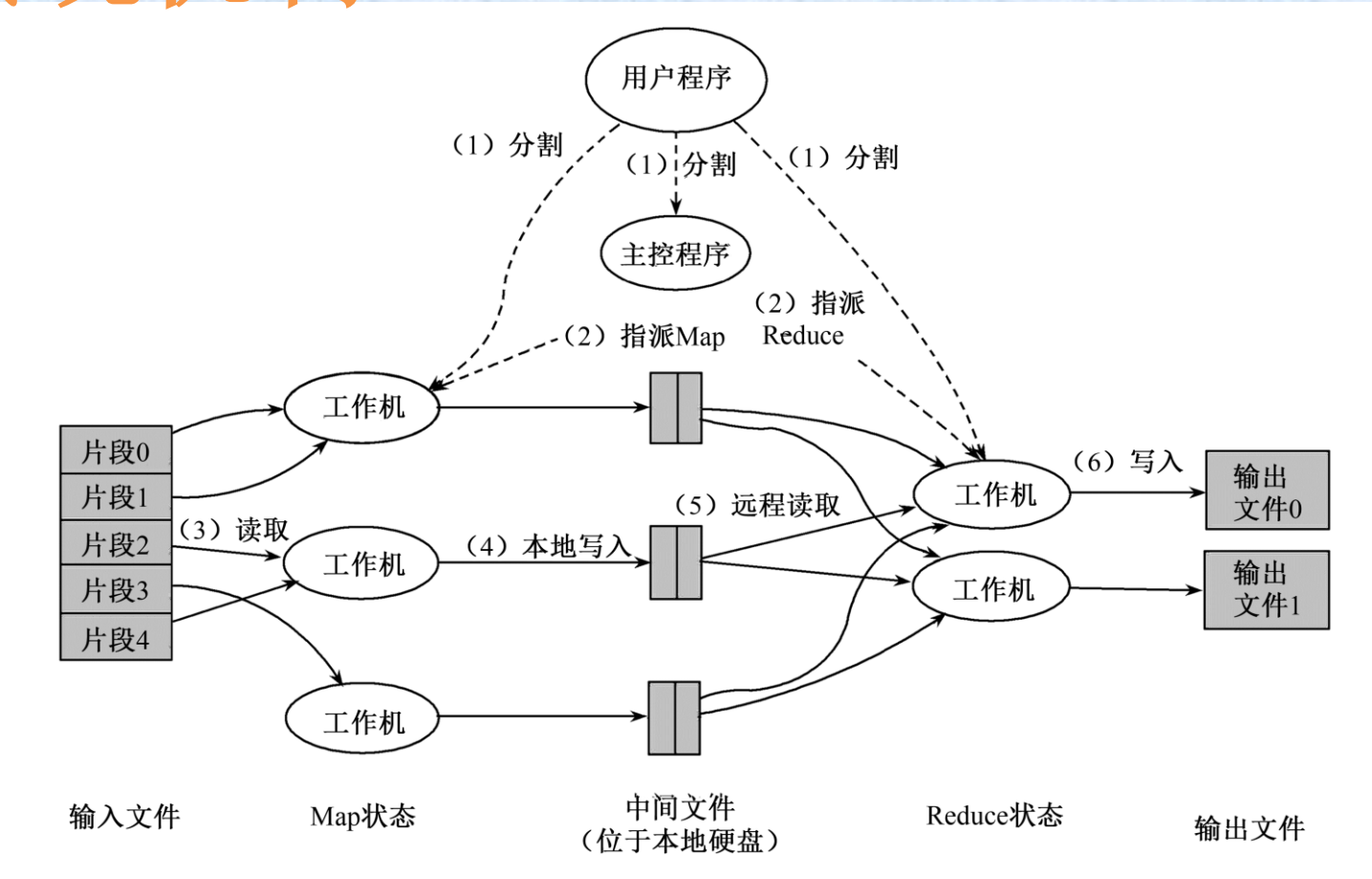
22.2.1 编程模型

22.2.2 实现机制

22.2.3 案例分析

22.2 分布式数据处理MapReduce

- 实现机制



22.2 分布式数据处理MapReduce

- 实现机制

（1）MapReduce函数首先把**输入文件分成M块**

（2）分派的执行程序中有一个**主控程序Master**

（3）一个被分配了Map任务的Worker读取并处理相关的输入块

（4）这些缓冲到内存的中间结果将被定时写到本地硬盘，这些**数据通过分区函数分成R个区**

（5）当Master通知执行Reduce的Worker关于中间<key,value>对的位置时，它调用远程过程，从Map Worker的本地硬盘上读取缓冲的中间数据

（6）Reduce Worker根据每一个唯一中间key来遍历所有的排序后的中间数据，并且把key和相关的中间结果值集合传递给用户定义的Reduce函数

（7）当所有的Map任务和Reduce任务都完成的时候，Master激活用户程序

22.2 分布式数据处理MapReduce

容错机制

由于MapReduce在成百上千台机器上处理海量数据，所以容错机制是不可或缺的。

总的来说，MapReduce通过重新执行失效的地方来实现容错

Master失效

Master会周期性地设置检查点（checkpoint），并导出Master的数据。一旦某个任务失效，系统就从最近的一个检查点恢复并重新执行。

由于只有一个Master在运行，如果Master失效了，则只能终止整个MapReduce程序的运行并重新开始。

Worker失效

Master会周期性地给Worker发送ping命令，如果没有Worker的应答，则Master认为Worker失效，终止对这个Worker的任务调度，把失效Worker的任务调度到其他Worker上重新执行。

目录

22.2分布式数据处理MapReduce

22.2.1 编程模型

22.2.2 实现机制

22.2.3 案例分析

目录

目录

22.1 Google文件系统GFS
22.2 分布式数据处理MapReduce
22.3 分布式锁服务Chubby
22.4 分布式结构化数据表Bigtable

22.3 分布式锁服务Chubby

- Chubby是Google设计的提供**粗粒度锁服务**的一个文件系统，它基于松耦合分布式系统，解决了分布的一致性问题。

通过使用Chubby的锁服务，用户可以确保数据操作过程中的一致性

Chubby作为一个稳定的存储系统存储包括元数据在内的小数据

Google内部还使用Chubby进行名字服务（Name Server）

目录

22.3 分布式锁服务Chubby

- 22.3.1 Paxos算法
- 22.3.2 Chubby系统设计
- 22.3.3 Chubby中的Paxos
- 22.3.4 Chubby文件系统
- 22.3.5 通信协议
- 22.3.6 正确性与性能

22.3 分布式锁服务Chubby

Paxos算法背景

希腊岛屿Paxon 上的执法者（legislators，后面称为牧师priest）在议会大厅（chamber）中表决通过法律，并通过服务员传递纸条的方式交流信息，每个执法者会将通过的法律记录在自己的账目（ledger）上。

问题在于执法者和服务员都不可靠，他们随时会因为各种事情离开议会大厅，并随时可能有新的执法者进入议会大厅进行法律表决，使用何种方式能够使得这个表决过程正常进行，且通过的法律不发生矛盾。

说明：不难看出故事中的议会大厅就是我们的分布式系统，牧师对应节点或进程，服务员传递纸条的过程就是消息传递的过程，法律即是我们需要保证一致性的值（value）。

牧师和服务员的进出对应着节点/网络的失效和加入，牧师的账目对应节点中的持久化存储设备。上面表决过程的正常进行可以表述为进展需求（progress requirements）：当大部分牧师在议会大厅呆了足够长时间，且期间没有牧师进入或者退出，那么提出的法案应该被通过并被记录在每个牧师的账目上。

Paxos算法是Lamport提出的一种基于消息传递的分布式一致性算法，使其获得2013年图灵奖。

Paxos算法解决的问题是分布式一致性问题，即一个分布式系统中的各个进程如何就某个值（决议）达成一致。

Paxos算法运行在允许宕机故障的异步系统中，不要求可靠的消息传递，可容忍消息丢失、延迟、乱序以及重复。它利用大多数 (Majority) 机制保证了2F+1的容错能力，即2F+1个节点的系统最多允许F个节点同时出现故障。

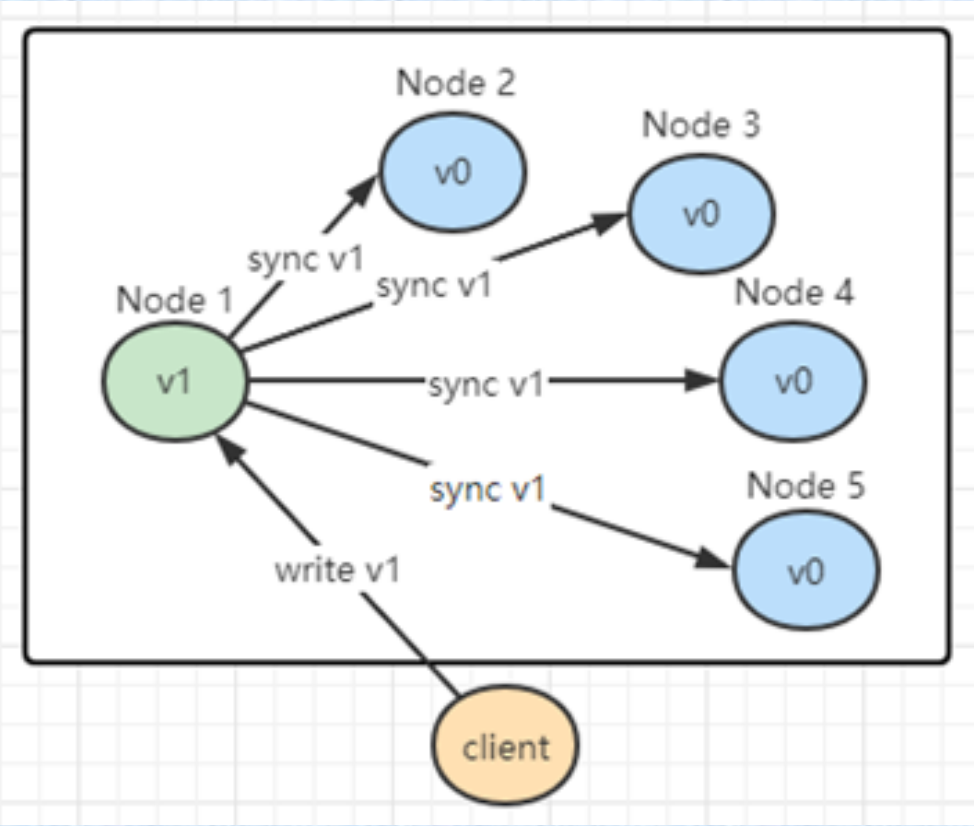
一个或多个提议进程 (Proposer) 可以发起提案 (Proposal)，Paxos算法使所有提案中的某一个提案，在所有进程中达成一致。系统中的多数派同时认可该提案，即达成了一致。最多只针对一个确定的提案达成一致。

Paxos 算法适用的几种场景：

- 一台机器中多个进程/线程达成数据一致；
- 分布式文件系统或者分布式数据库中多客户端并发读写数据；
- 分布式存储中多个副本响应读写请求的一致性。

问题的提出

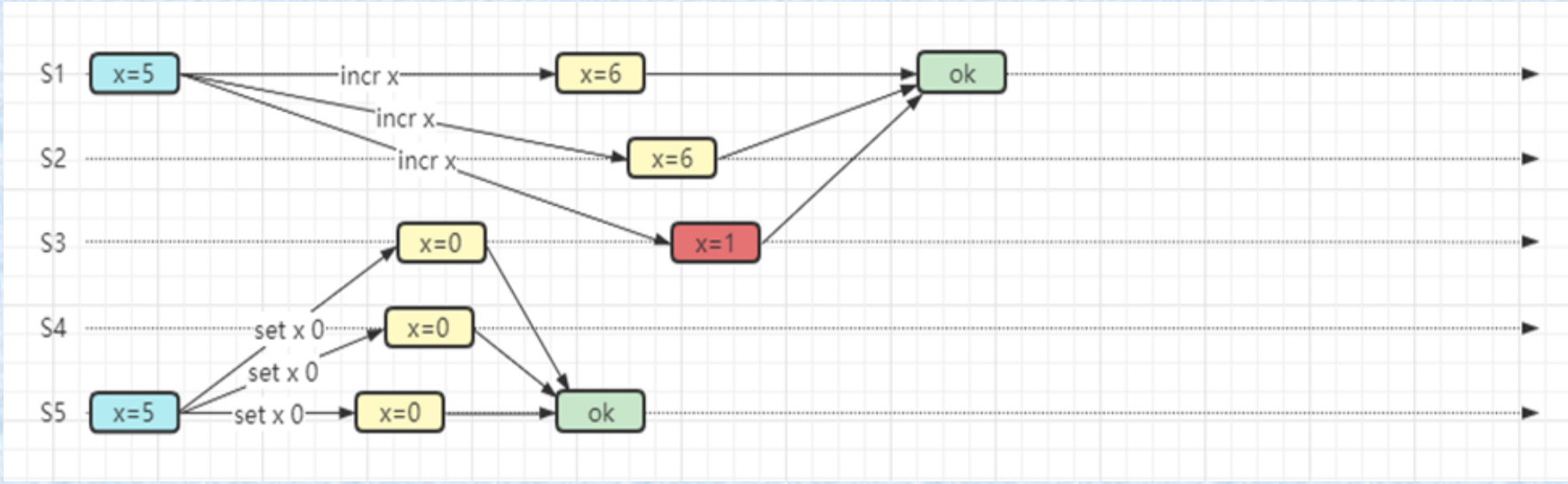
例如，集群中有 N 个节点，如果一个节点写入后要求同步到剩余 N-1 个节点后再向客户端返回 ok，虽然看起来最保险，但其中任意一个节点同步失败，势必造成整个集群不可用，能否在此基础上提高可用性呢？



问题的提出（续）

答案是 **（写）多数派**，集群节点设置为奇数，同步超过集群中 N/2 个节点成功，则向客户端返回 ok。但在分布式环境中存在顺序性问题，如下面描述的：下图的两项操作，都满足了多数派通过，但 S3 这台服务器并没有与 S1，S2 达成一致（时间序列原因，S3先执行set操作，再执行incr操作，造成了最后结果的不一致性）。

这表明需要设计一种算法实现**多数派内部一致性。**



Paxos算法设计

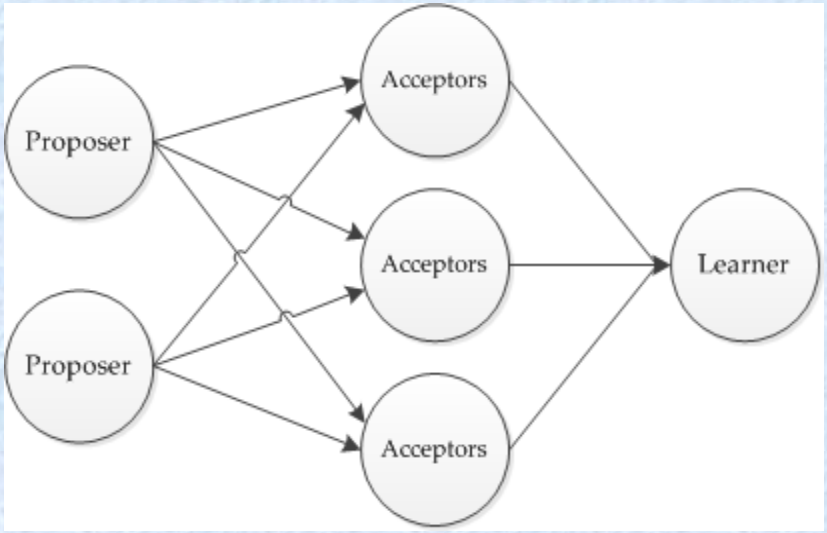
Paxos将系统中的角色分为提议者 (Proposer)，决策者 (Acceptor)，和最终决策学习者 (Learner):

Proposer: 提出提案 (Proposal)。Proposal信息包括提案编号 (Proposal ID) 和提议的值 (Value)。

Acceptor：参与决策，回应Proposers的提案。收到Proposal后可以接受提案，若Proposal获得多数Acceptors的接受，则称该Proposal被批准。

Learner：不参与决策，从Proposers/Acceptors学习最新达成一致的提案（Value）。

在多副本状态机中，每个副本同时具有Proposer、Acceptor、Learner三种角色。

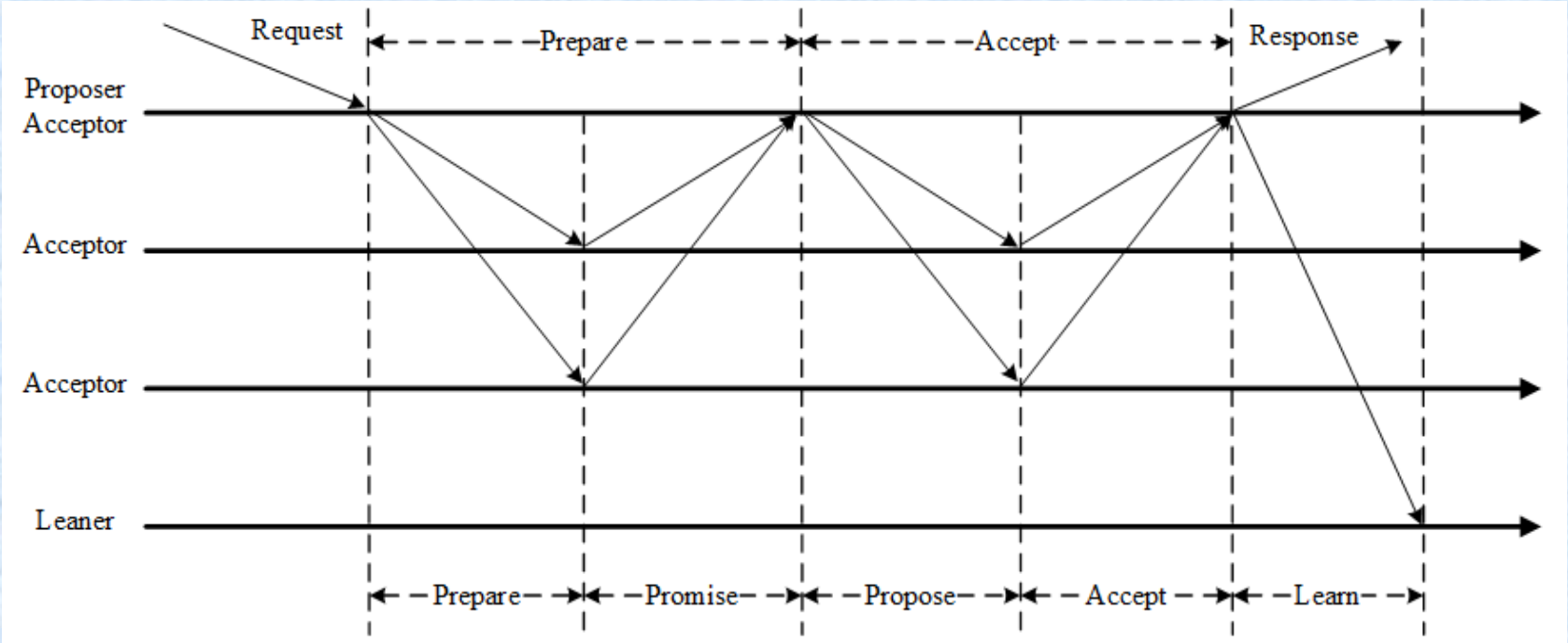


Paxos算法通过一个决议分为三个阶段：

第一阶段：Prepare阶段。Proposer向Acceptors发出Prepare请求，Acceptors针对收到的Prepare请求进行Promise承诺。

第二阶段：Accept阶段。Proposer收到多数Acceptors承诺的Promise后，向Acceptors发出Propose请求，Acceptors针对收到的Propose请求进行Accept处理。

第三阶段：Learn阶段。Proposer在收到多数Acceptors的Accept之后，标志着本次Accept成功，决议形成，将形成的决议发送给所有Learners。



Paxos算法流程中的每条消息描述如下：

Prepare: Proposer生成全局唯一且递增的Proposal ID (可使用时间戳加Server ID)，向所有Acceptors发送Prepare请求，这里无需携带提案内容，只携带Proposal ID即可。

Promise: Acceptors收到Prepare请求后，做出“两个承诺，一个应答”。
两个承诺：
1. 不再接受Proposal ID小于等于（<= ）当前请求的Prepare请求。
2. 不再接受Proposal ID小于（< ）当前请求的Propose请求。

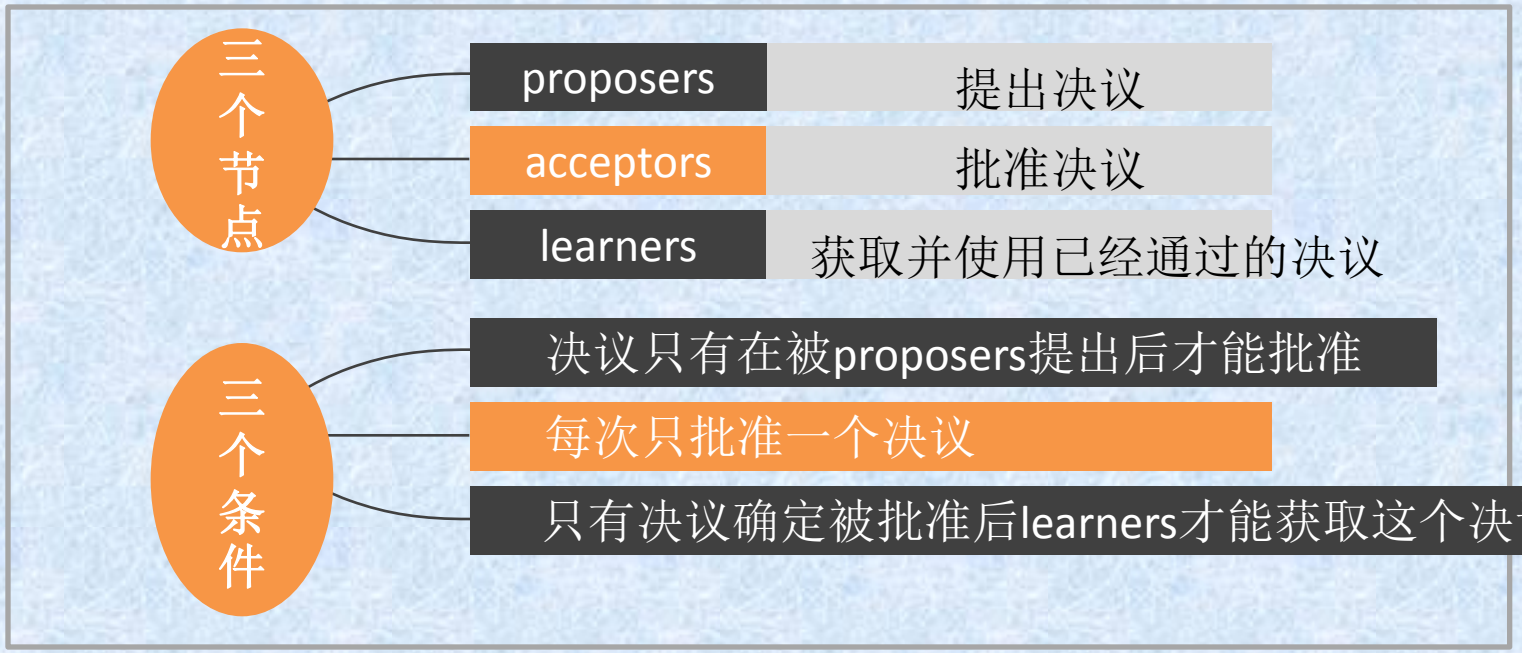
一个应答：
不违背以前作出的承诺下，回复已经Accept过的提案中Proposal ID最大的那个提案的Value和Proposal ID，没有则返回空值。

Propose: Proposer 收到多数Acceptors的Promise应答后，从应答中选择Proposal ID最大的提案的Value，作为本次发起的提案。如果所有应答的提案Value均为空值，则可以自己随意决定提案Value。然后携带当前Proposal ID，向所有Acceptors发送Propose请求。

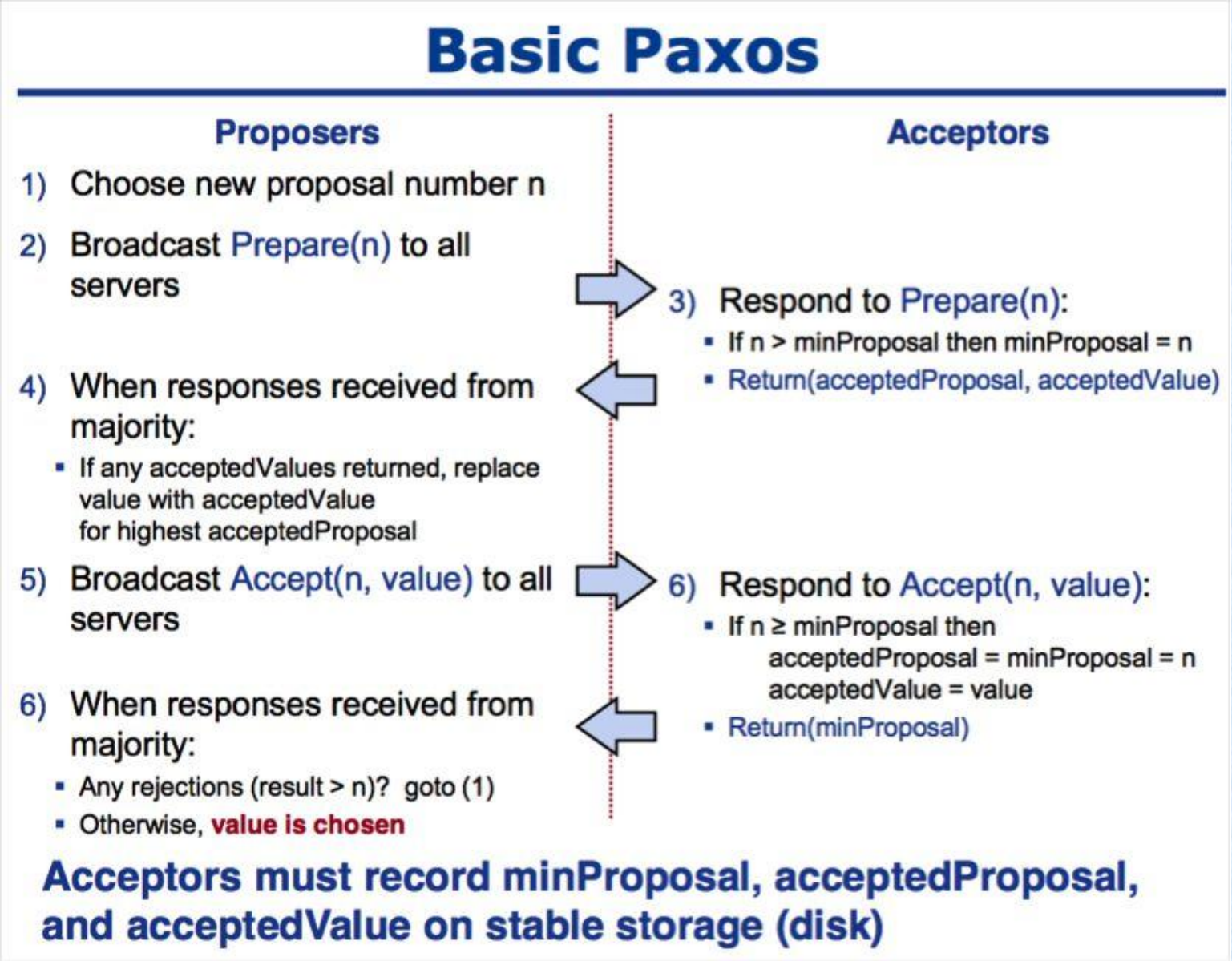
Accept: Acceptor收到Propose请求后，在不违背自己之前作出的承诺下，接受并持久化当前Proposal ID和提案Value。

Learn: Proposer收到多数Acceptors的Accept后，决议形成，将形成的决议发送给所有Learners。

Paxos算法条件


三个节点
proposers
提出决议
acceptors
批准决议
learners
获取并使用已经通过的决议
三个条件
决议只有在被proposers提出后才能批准
每次只批准一个决议
只有决议确定被批准后learners才能获取这个决议

Paxos算法伪码


Basic Paxos
Proposers
Acceptors
1) Choose new proposal number n
2) Broadcast Prepare(n) to all servers
3) Respond to Prepare(n):
If n > minProposal then minProposal = n
Return(acceptedProposal, acceptedValue)
4) When responses received from majority:
If any acceptedValues returned, replace value with acceptedValue for highest acceptedProposal
5) Broadcast Accept(n, value) to all servers
6) Respond to Accept(n, value):
If n ≥ minProposal then
acceptedProposal = minProposal = n
acceptedValue = value
Return(minProposal)
6) When responses received from majority:
Any rejections (result > n)? goto (1)
Otherwise, value is chosen
Acceptors must record minProposal, acceptedProposal, and acceptedValue on stable storage (disk)

1）获取一个Proposal ID n，为了保证Proposal ID唯一，可采用时间戳+Server ID生成；

2）Proposer向所有Acceptors广播Prepare(n)请求；

3）Acceptor比较n和minProposal，如果n>minProposal，minProposal=n，并且将 acceptedProposal 和 acceptedValue 返回；

4）Proposer接收到过半数回复后，如果发现有acceptedValue返回，将所有回复中acceptedProposal最大的acceptedValue作为本次提案的value，否则可以任意决定本次提案的value；

5）到这里可以进入第二阶段，广播Accept (n, value) 到所有节点；

6）Acceptor比较n和minProposal，如果n>=minProposal，则acceptedProposal=minProposal=n，acceptedValue=value，本地持久化后，返回；否则，返回minProposal。

7）提议者接收到过半数请求后，如果发现有返回值result >n，表示有更新的提议，跳转到1）；否则value达成一致

22.3 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。

22.3 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”

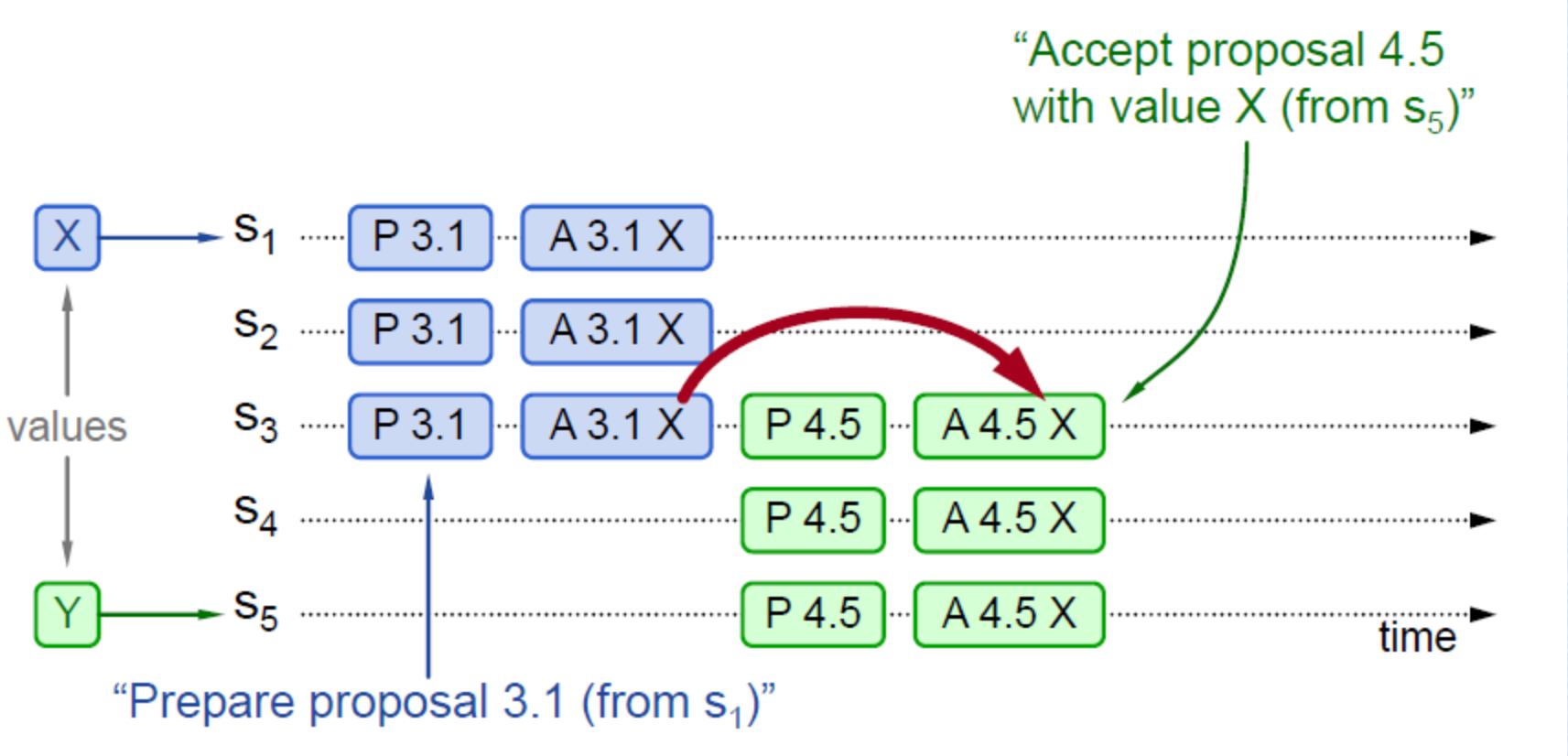
acceptors 收到后，如果提案的编号大于它已经回复的所有消息，则acceptors将自己上次的批准回复给proposers，并不再批准小于n的提案。

2

批准阶段

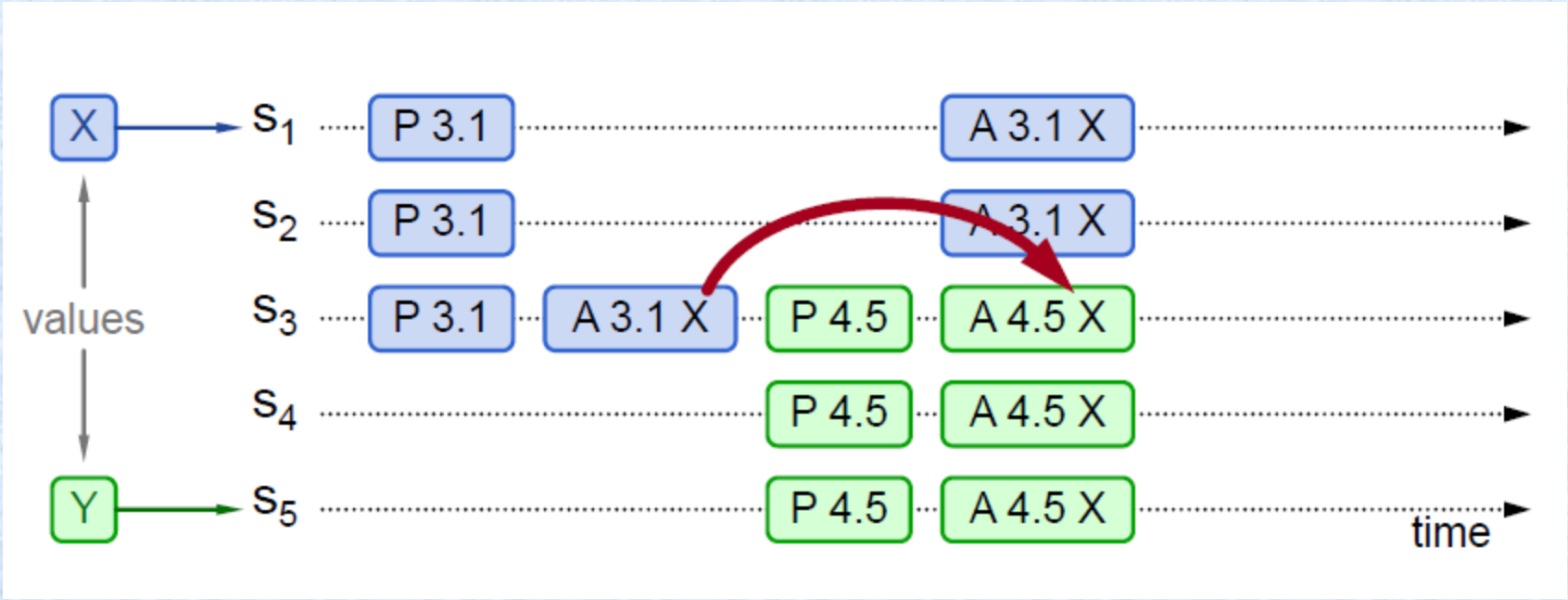
当proposers接收到acceptors 中的这个“多数派”的回复后，就向回复请求的acceptors发送accept请求，在符合acceptors一方的约束条件下，acceptors收到accept请求后即批准这个请求。

图中P代表Prepare阶段，A代表Accept阶段。3.1代表Proposal ID为3.1，其中3为时间戳，1为Server ID。X和Y代表提议Value。



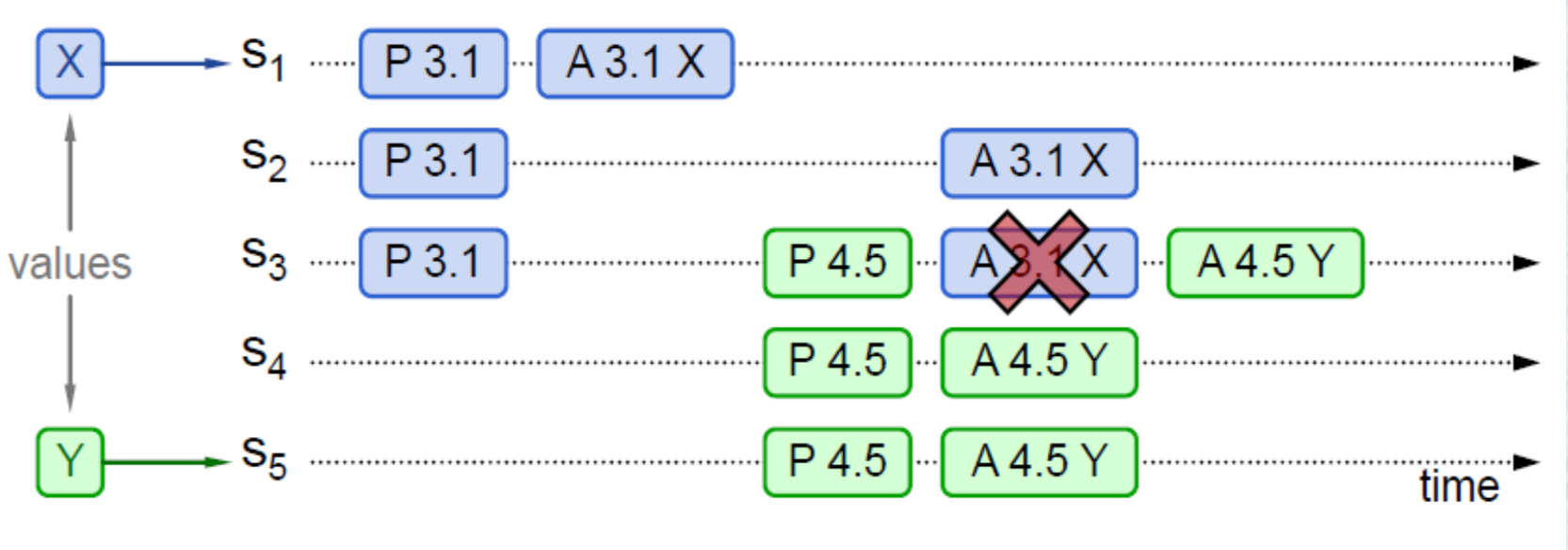
实例1

P 3.1达成多数派，其Value(X)被Accept，然后P 4.5学习到Value(X)，并Accept



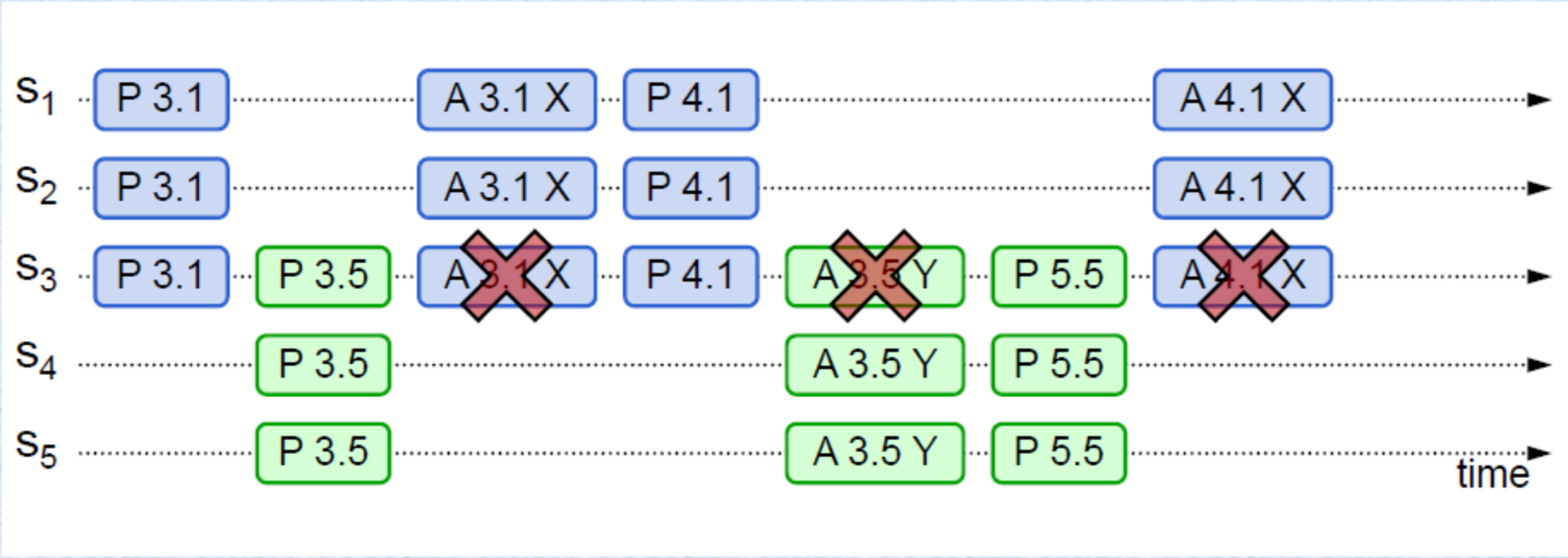
实例2

P 3.1没有被Accept（只有S3 Accept），但是被S4在P 4.5学习到，S4在P 4.5将自己的Value由Y替换为X，Accept（X）。



实例3

P 3.1没有被多数派Accept（只有S1 Accept），同时也没有被P 4.5学习到。由于P 4.5 Propose的所有应答，均未返回Value，则P 4.5可以Accept自己的Value (Y)。后续P 3.1的Accept (X) 会失败，已经Accept的S1，会被覆盖。



Paxos算法可能形成活锁而永远不会结束

回顾两个承诺之一，Acceptor不再应答Proposal ID小于等于当前请求的Prepare请求。意味着需要应答Proposal ID大于当前请求的Prepare请求。

两个Proposers交替Prepare成功，而Accept失败，形成活锁（Livelock）。

目录

22.3 分布式锁服务Chubby

22.3.1 Paxos算法

22.3.2 Chubby系统设计

22.3.3 Chubby中的Paxos

22.3.4 Chubby文件系统

22.3.5 通信协议

22.3.6 正确性与性能

分布式锁 (Distributed Lock)

用于实现在分布式系统中多个进程对临界资源的互斥访问，保证分布式系统数据的一致性。分布式协调技术的核心就是实现分布式锁。

分布式锁特点

（1）互斥性：在分布式系统环境下，同一时间内不同节点的不同线程对特殊资源的互斥访问；

（2）高可用的获取锁和释放锁；

（3）高性能的获取锁和释放锁；

（4）可重入性：同一个节点上的同一个线程如果获取了锁之后还可以可以再次获取这个锁；

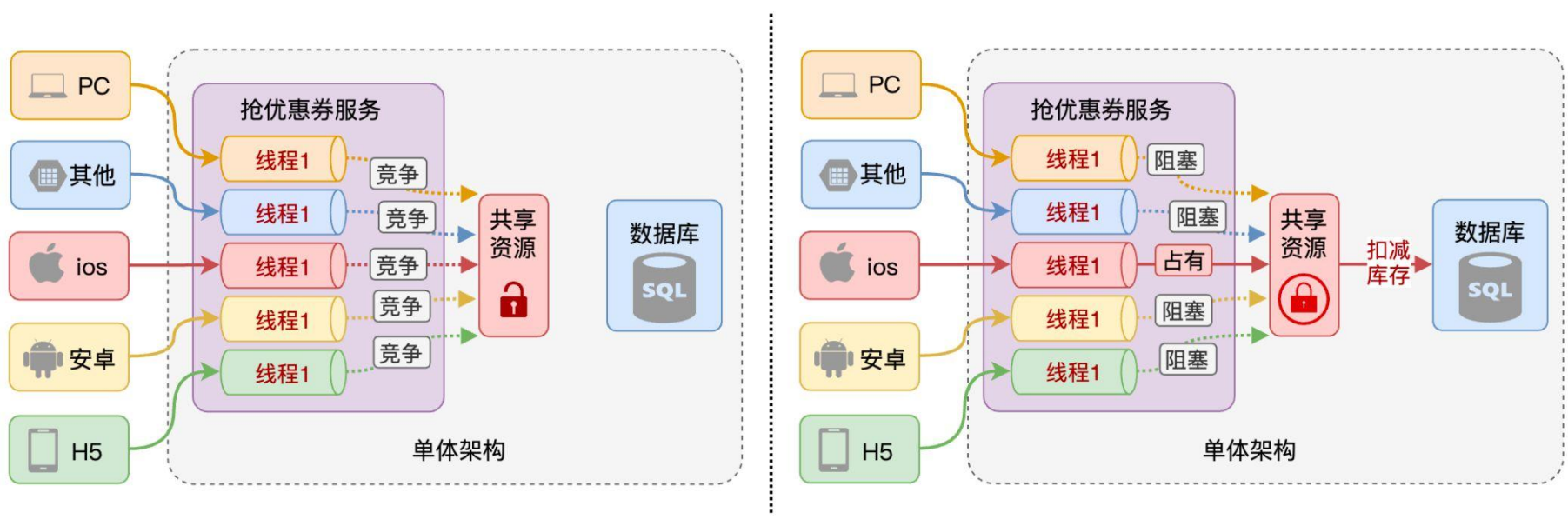
（5）锁超时：具备锁超时失效机制，防止死锁；

（6）非阻塞：没有获取到锁将直接接返回获取锁失败支持阻塞和非阻塞；

（7）支持公平锁和非公平锁(可选)：公平锁是按照请求加锁的顺序获取锁，非公平锁即随机获取锁。

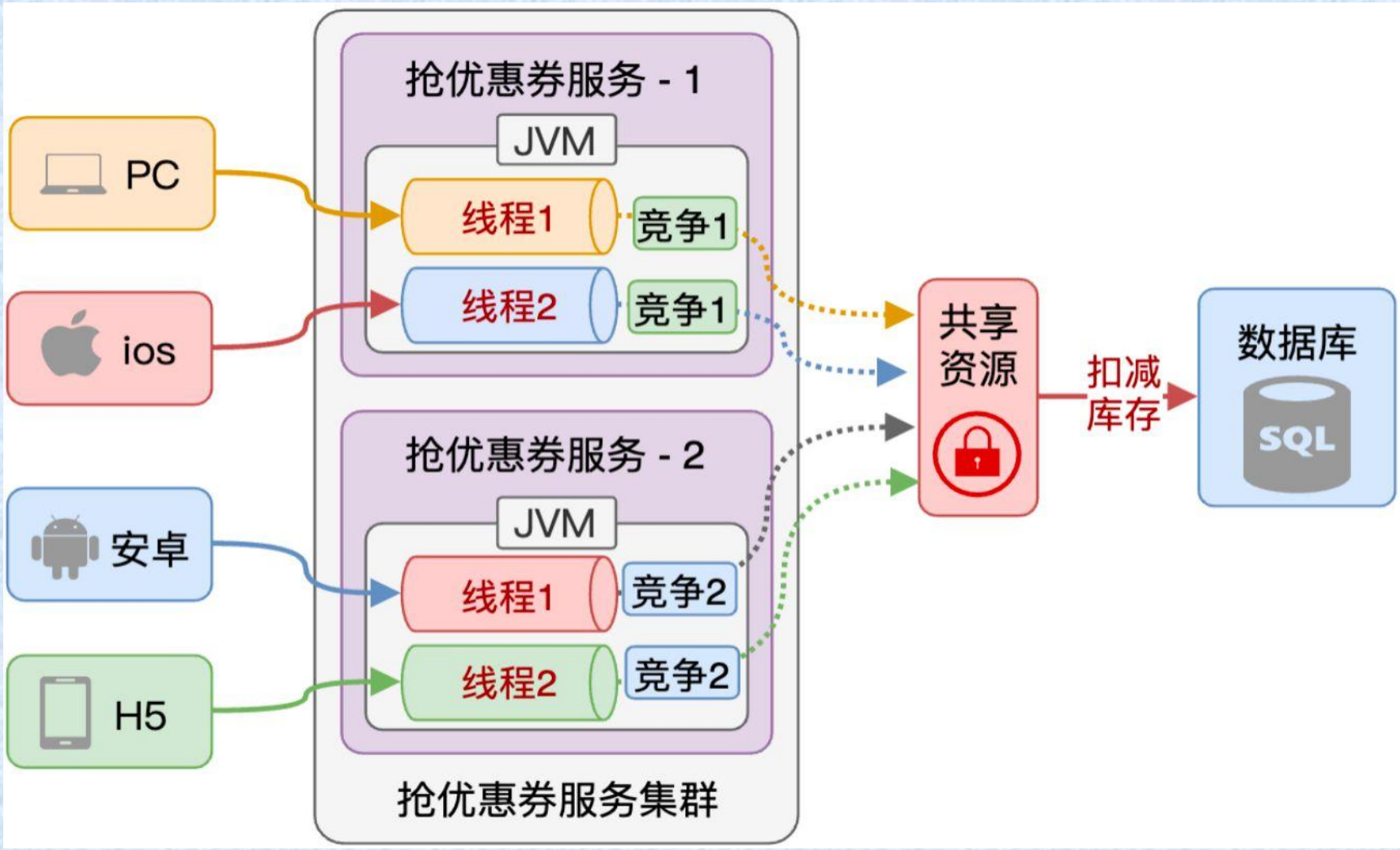
单机锁

在单机环境下，也就是单个JVM环境下多线程对共享资源的并发更新处理，可以简单地使用JDK提供的ReentrantLock对共享资源进行加锁处理。



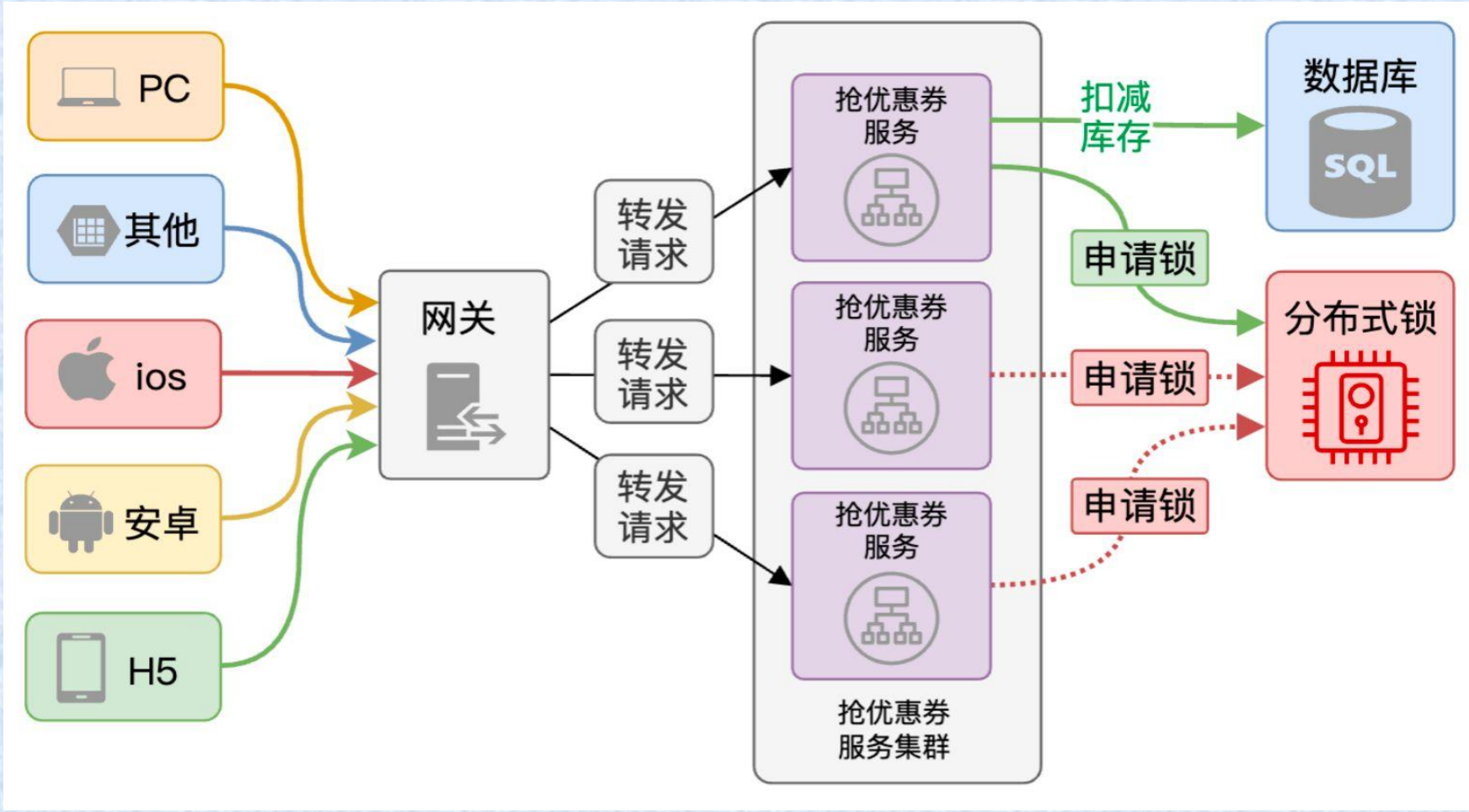

```
ReentrantLock lock = new ReentrantLock();
try {
        lock.lock();  //处理共享资源
 } finally {
       lock.unlock();
}
```

在微服务架构多实例的环境下，每一个服务都有多个节点。再用ReentrantLock就没办法控制了，因为这时候这些任务是跨JVM的，不再是简单单体应用，需要协同多个节点信息，共同获取锁的竞争情况。



分布式锁

通常是把锁和应用分开部署，把这个锁做成一个公用的组件，然后多个不同应用的不同计算节点，在分布式环境中都去共同访问这个组件（有多种实现方式）。



22.3 分布式锁服务Chubby

分布式锁是控制分布式系统之间同步访问共享资源的一种方式。

Chubby是一种面向松耦合的分布式系统的锁服务，通常用于为一个由适度规模的大量计算节点构成的的松耦合分布式系统提供高可用的分布式锁服务。锁服务的目的是允许它的客户端进程同步彼此的操作，并对当前所处环境的基本状态信息达成一致。因此，Chubby的主要设计目标是为一个由适度大规模的客户端进程组成的分布式场景提供高可用的锁服务，以及易于理解的API接口定义。

值得一提的是，在Chubby的设计过程中，系统的吞吐量和存储容量并不是首要考虑的因素。Chubby的客户端接口设计非常类似于文件系统结构，不仅能够对Chubby上的整个文件进行读写操作，还能够添加对文件节点的锁控制，并且能够订阅Chubby服务端发出的一系列文件变动的事件通知。

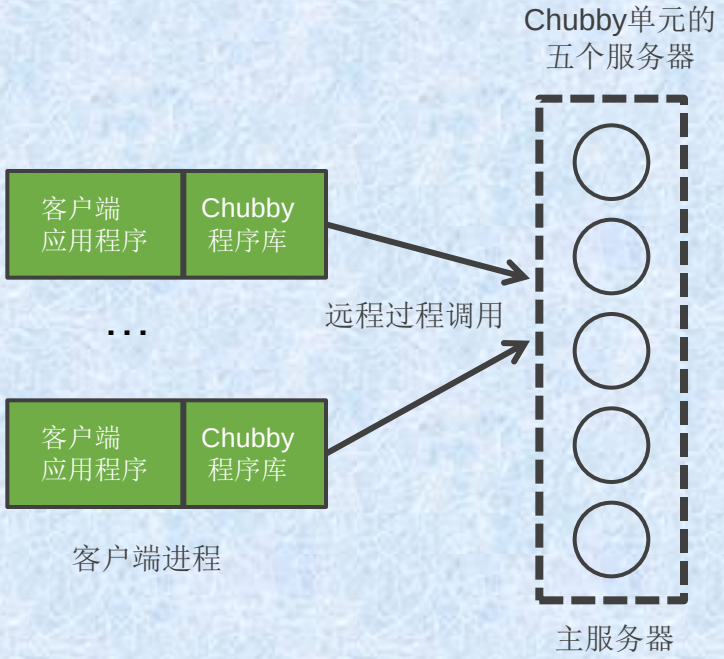
22.3 分布式锁服务Chubby

- Chubby的设计目标主要有以下几点

1	高可用性和高可靠性	4	服务信息的直接存储
2	高扩展性	5	支持缓存机制
3	支持粗粒度的建议性锁服务	6	支持通报机制

22.3 分布式锁服务Chubby

- **Chubby的基本架构**



客户端

在客户这一端每个客户应用程序都有一个Chubby程序库（Chubby Library），客户端的所有应用都是通过调用这个库中的相关函数来完成的。

服务器端

服务器一端称为Chubby单元，一般是由五个称为副本（Replica）的服务器组成的，这五个副本在配置上完全一致，并且在系统刚开始时处于对等地位。

目录

22.3 分布式锁服务Chubby

22.3.1 Paxos算法

22.3.2 Chubby系统设计

22.3.3 Chubby中的Paxos

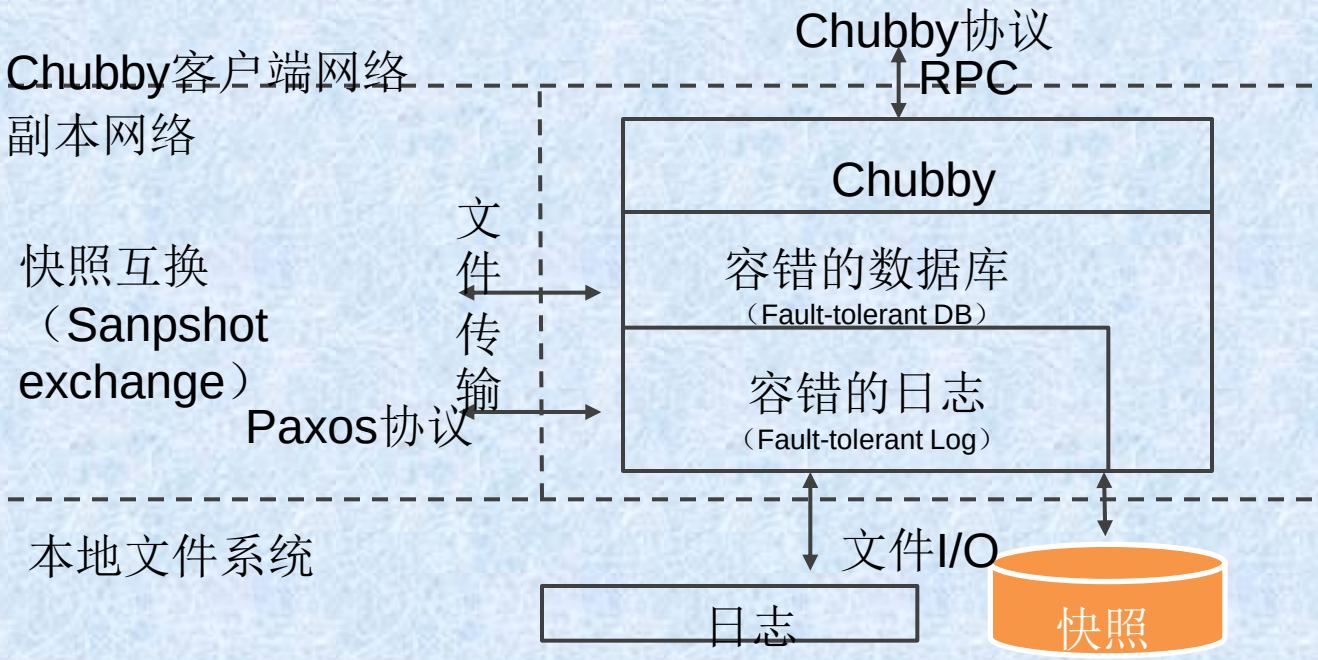
22.3.4 Chubby文件系统

22.3.5 通信协议

22.3.6 正确性与性能

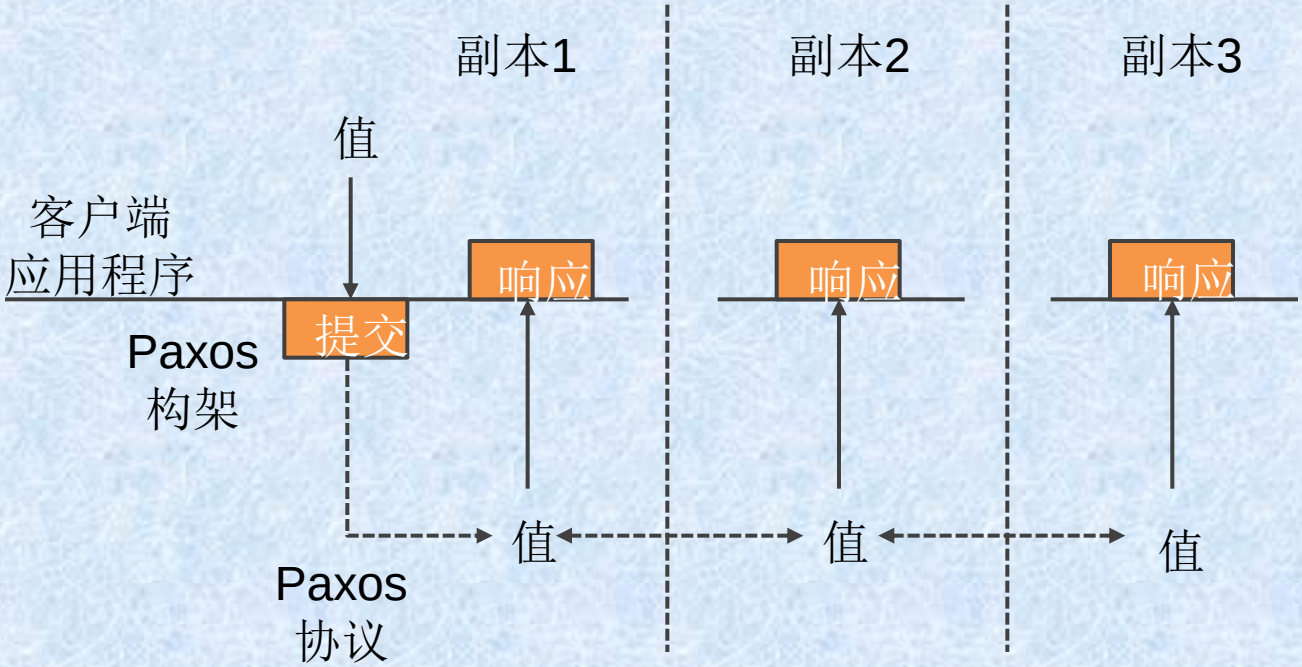
22.3 分布式锁服务Chubby

- 单个Chubby副本结构



22.3 分布式锁服务Chubby

- 容错日志的API



目录

22.3 分布式锁服务Chubby

22.3.1 Paxos算法

22.3.2 Chubby系统设计

22.3.3 Chubby中的Paxos

22.3.4 Chubby文件系统

22.3.5 通信协议

22.3.6 正确性与性能

22.3 分布式锁服务Chubby

- 单调递增的64位编号

❶ 实例号 Instance Number	新节点实例号必定大于旧节点的实例号。
❷ 内容生成号 Content Generation Number	文件内容修改时该号增加。
❸ 锁生成号 Lock Generation Number	锁被用户持有时该号增加。
❹ ACL生成号 ACL Generation Number	ACL名被覆写时该号增加。

22.3 分布式锁服务Chubby

常用的句柄函数及作用

函数名称	作　用
Open()	打开某个文件或者目录来创建句柄
Close()	关闭打开的句柄，后续的任何操作都将中止
Poison()	中止当前未完成及后续的操作，但不关闭句柄
GetContentsAndStat()	返回文件内容及元数据
GetStat()	只返回文件元数据
ReadDir()	返回子目录名称及其元数据
SetContents()	向文件中写入内容
SetACL()	设置ACL名称
Delete()	如果该节点没有子节点的话则执行删除操作
Acquire()	获取锁
Release()	释放锁
GetSequencer()	返回一个sequencer
SetSequencer()	将sequencer和某个句柄进行关联
CheckSequencer()	检查某个sequencer是否有效

目录

22.3 分布式锁服务Chubby

22.3.1 Paxos算法

22.3.2 Chubby系统设计

22.3.3 Chubby中的Paxos

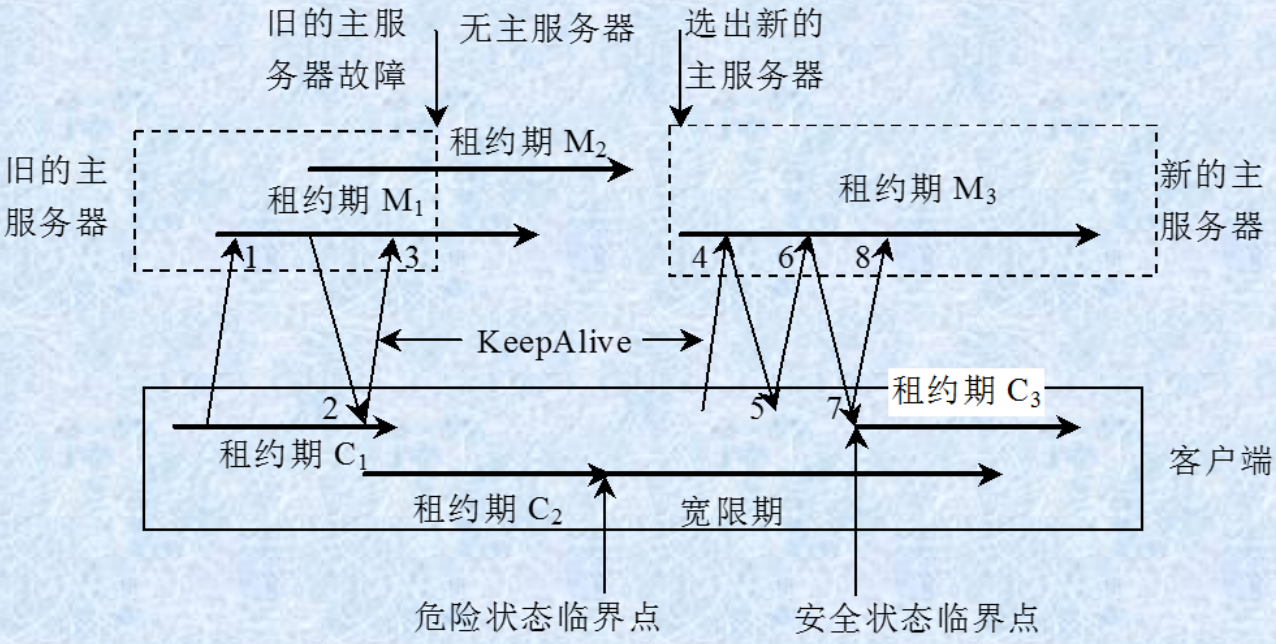
22.3.4 Chubby文件系统

22.3.5 通信协议

22.3.6 正确性与性能

22.3 分布式锁服务Chubby

- Chubby客户端与服务器端的通信过程



22.3 分布式锁服务Chubby

- 可能出现的两种故障

1 客户端租约过期

2 主服务器出错

目录

22.3 分布式锁服务Chubby

22.3.1 Paxos算法

22.3.2 Chubby系统设计

22.3.3 Chubby中的Paxos

22.3.4 Chubby文件系统

22.3.5 通信协议

22.3.6 正确性与性能

22.3 分布式锁服务Chubby

- 正确性与性能

一致性

每个Chubby单元是由五个副本组成的，这五个副本中需要选举产生一个主服务器，这种选举本质上就是一个一致性问题

安全性

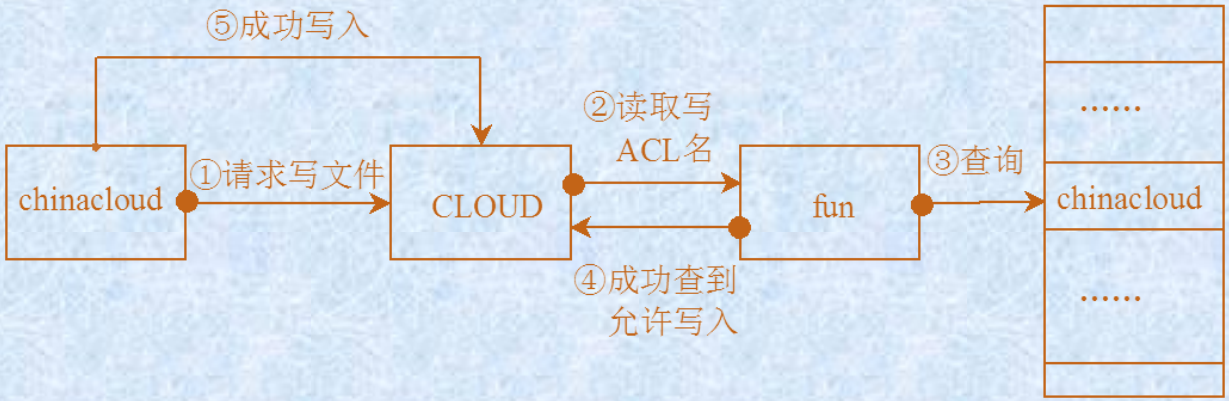
采用的是ACL形式的安全保障措施。只要不被覆写，子节点都是直接继承父节点的ACL名

性能优化

提高主服务器默认的租约期、使用协议转换服务将Chubby协议转换成较简单的协议、客户端一致性缓存等

22.3 分布式锁服务Chubby

- **Chubby 的 ACL 机制**



用户chinacloud提出向文件CLOUD中写入内容的请求。CLOUD首先读取自身的写ACL名fun，接着在fun中查到了chinacloud这一行记录，于是返回信息允许chinacloud对文件进行写操作，此时chinacloud才被允许向CLOUD写入内容。其他的操作和写操作类似。

目录

目录

22.1 Google文件系统GFS
22.2 分布式数据处理MapReduce
22.3 分布式锁服务Chubby
22.4 分布式结构化数据表Bigtable

目录

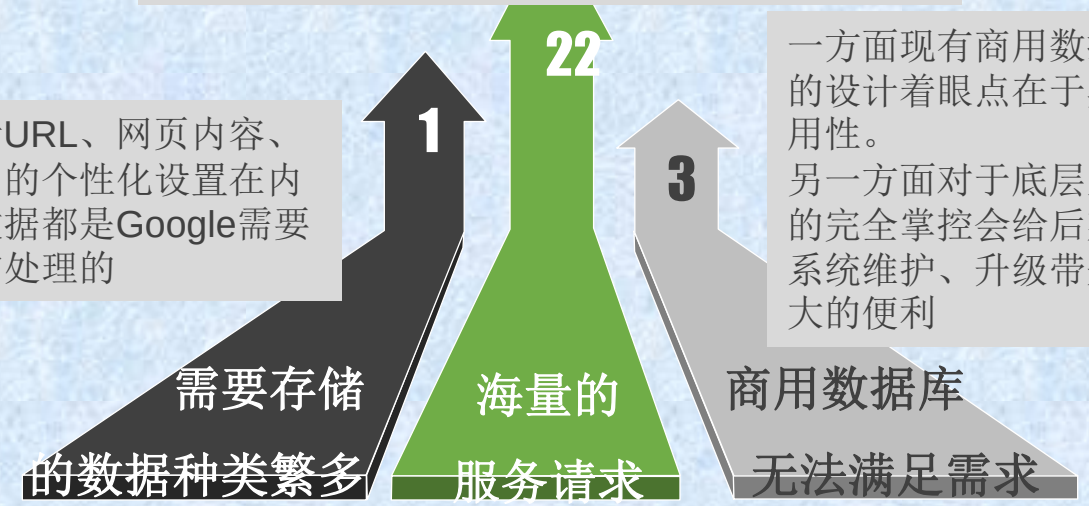
22.4 分布式结构化数据表Bigtable

- 22.4.1 设计动机与目标
- 22.4.2 数据模型
- 22.4.3 系统架构
- 22.4.4 主服务器
- 22.4.5 子表服务器
- 22.4.6 性能优化

22.4 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

22.4 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

目录

22.4 分布式结构化数据表Bigtable

22.4.1 设计动机与目标

22.4.2 数据模型

22.4.3 系统架构

22.4.4 主服务器

22.4.5 子表服务器

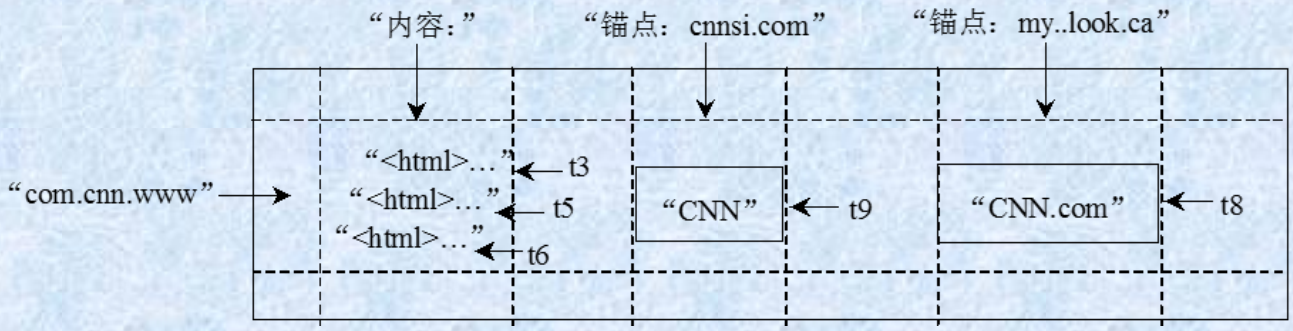
22.4.6 性能优化

22.4 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



22.4 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
- 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

目录

22.4 分布式结构化数据表Bigtable

22.4.1 设计动机与目标

22.4.2 数据模型

22.4.3 系统架构

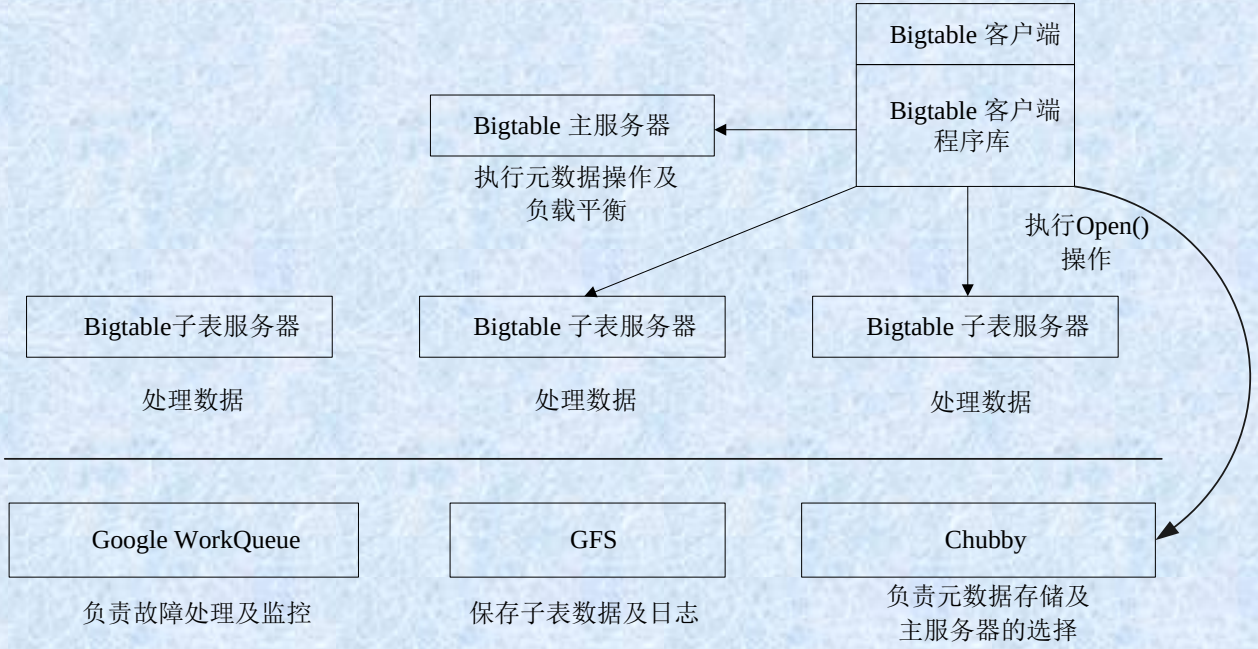
22.4.4 主服务器

22.4.5 子表服务器

22.4.6 性能优化

22.4 分布式结构化数据表Bigtable

- Bigtable 基本架构



22.4 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。

目录

22.4 分布式结构化数据表Bigtable

22.4.1 设计动机与目标

22.4.2 数据模型

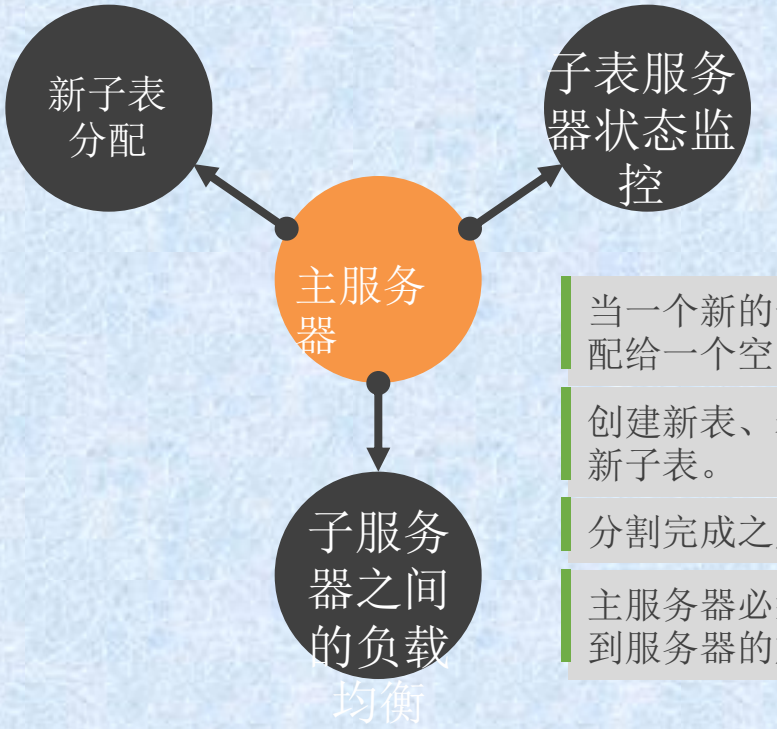
22.4.3 系统架构

22.4.4 主服务器

22.4.5 子表服务器

22.4.6 性能优化

22.4 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

22.4 分布式结构化数据表Bigtable

- **Bigtable 中 Chubby 的主要作用**

目录

22.4 分布式结构化数据表Bigtable

22.4.1 设计动机与目标

22.4.2 数据模型

22.4.3 系统架构

22.4.4 主服务器

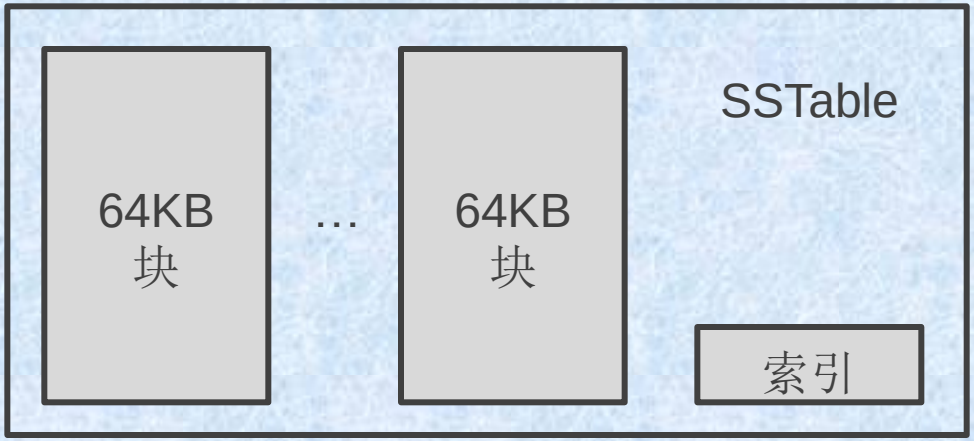
22.4.5 子表服务器

22.4.6 性能优化

22.4 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。



22.4 分布式结构化数据表Bigtable

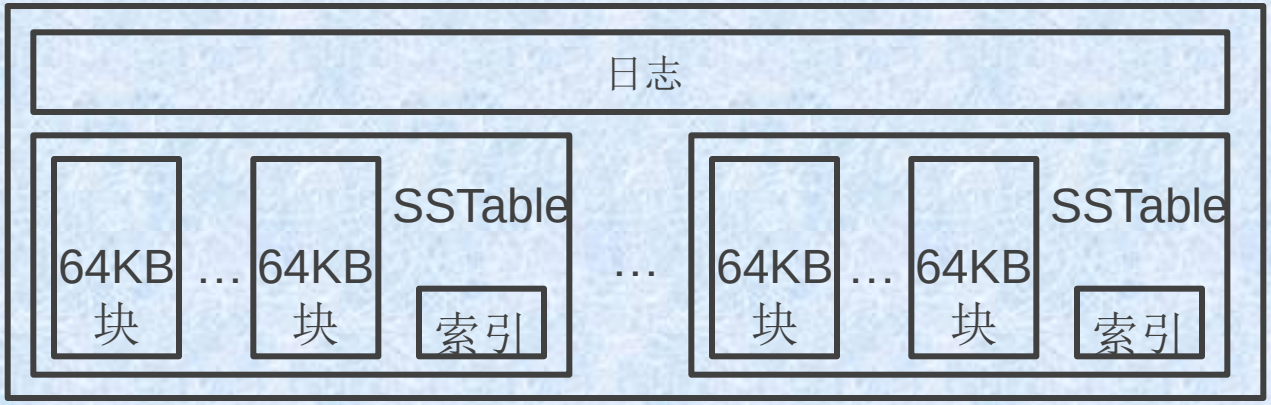
- 子表实际组成

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

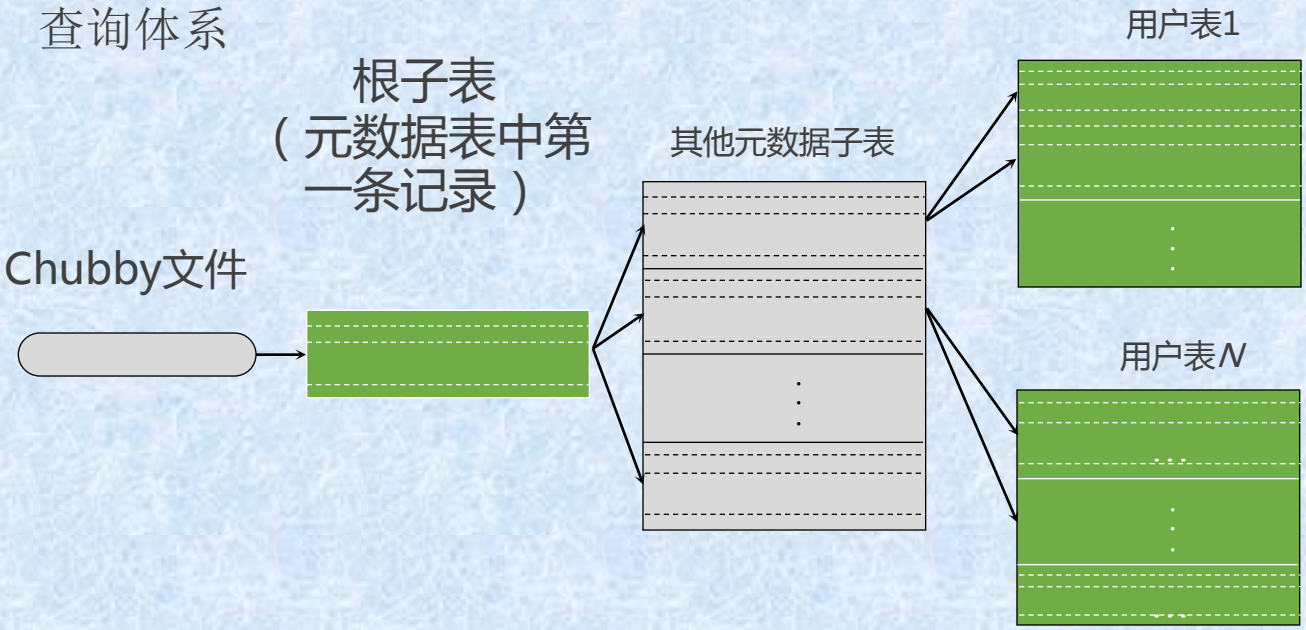
每个子表服务器上保存的子表数量可以从几十到上千不等，通常情况下是100个左右



22.4 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

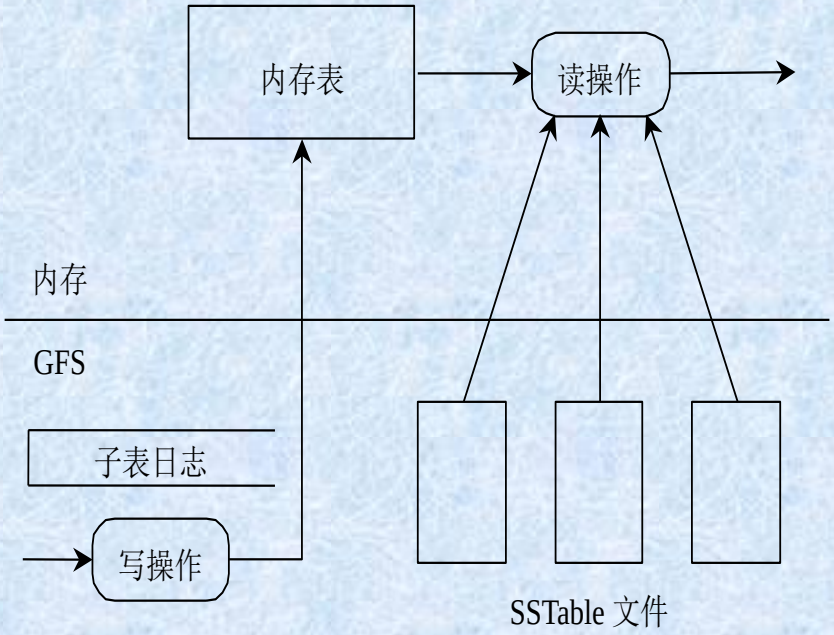


22.4 分布式结构化数据表Bigtable

- **Bigtable 数据存储及读/写操作**

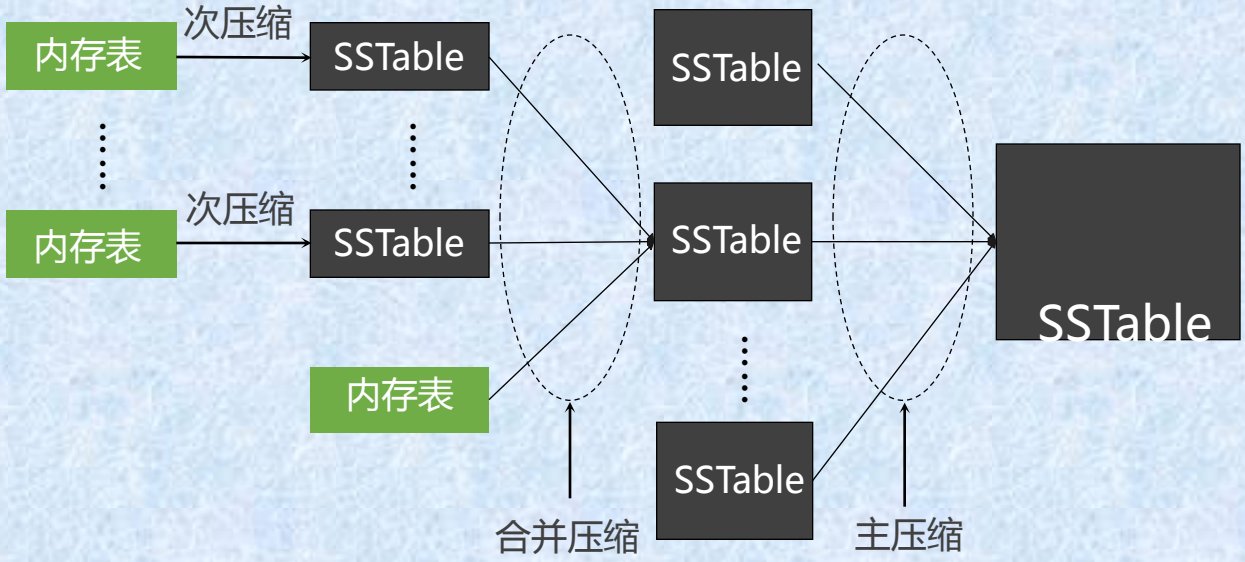
较新的数据存储在内存中一个称为内存表（Memtable）的有序缓冲里，较早的数据则以SSTable格式保存在GFS中。

读和写操作有很大的差异性



22.4 分布式结构化数据表Bigtable

- 三种形式压缩之间的关系



目录

22.4 分布式结构化数据表Bigtable

22.4.1 设计动机与目标

22.4.2 数据模型

22.4.3 系统架构

22.4.4 主服务器

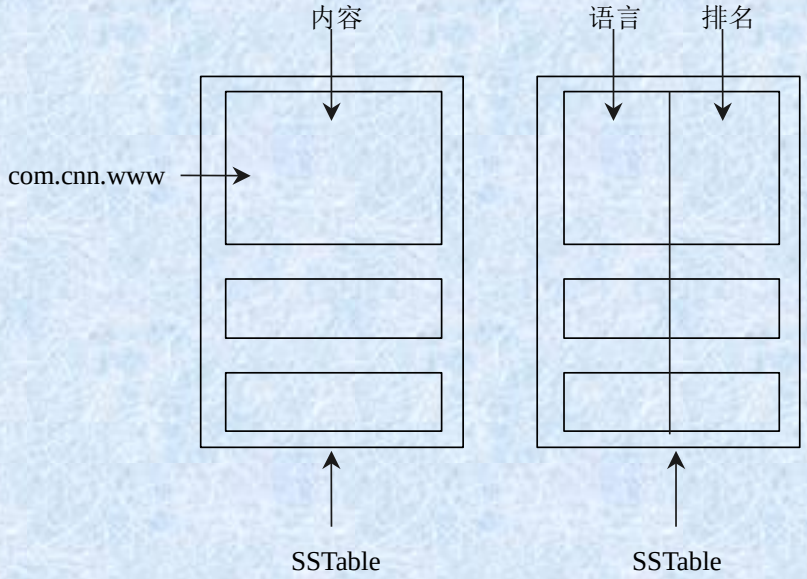
22.4.5 子表服务器

22.4.6 性能优化

22.4 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。

22.4 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

22.4 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	