



第三章

数据划分和定位

第三章、数据划分和定位

01

3.1 数据划分的意义

02

3.2 静态数据划分方法

03

3.3 动态数据分区调整

04

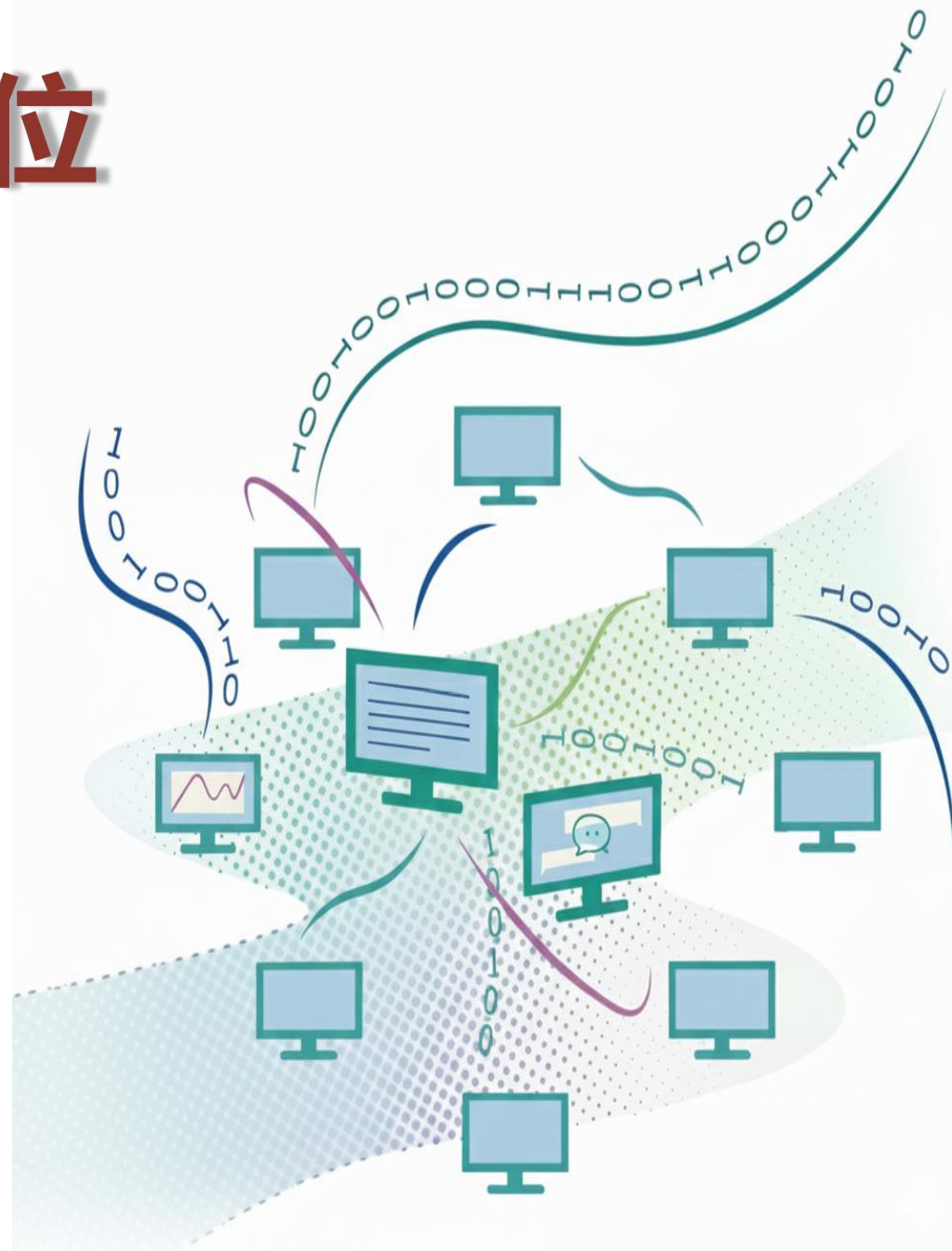
3.4 数据定位

05

3.5 一致性哈希

06

3.6 分布式文件系统-GFS



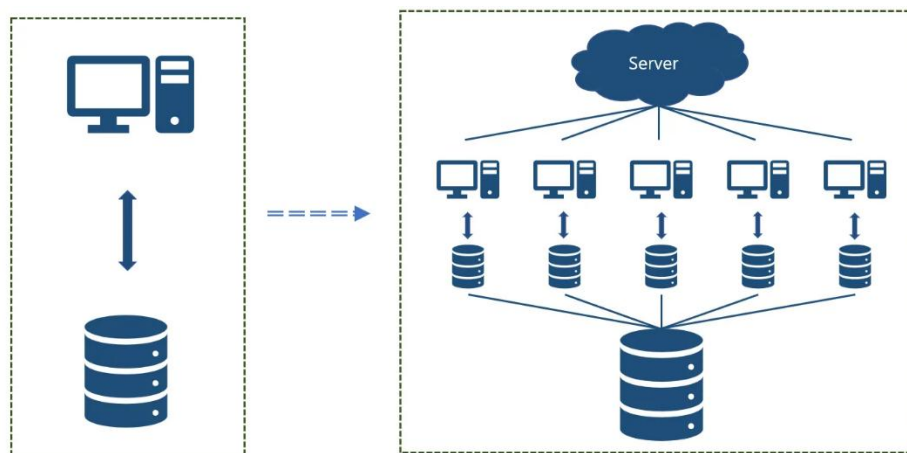
第3章 数据划分与定位

■ 数据划分

- 将大规模的数据集分割成较小、更易管理和处理的数据子集
- 将数据子集（也称为分区）分配到**不同节点上**

■ 数据定位

- 查找数据的存储位置，便于后续进行数据查找和更新等操作



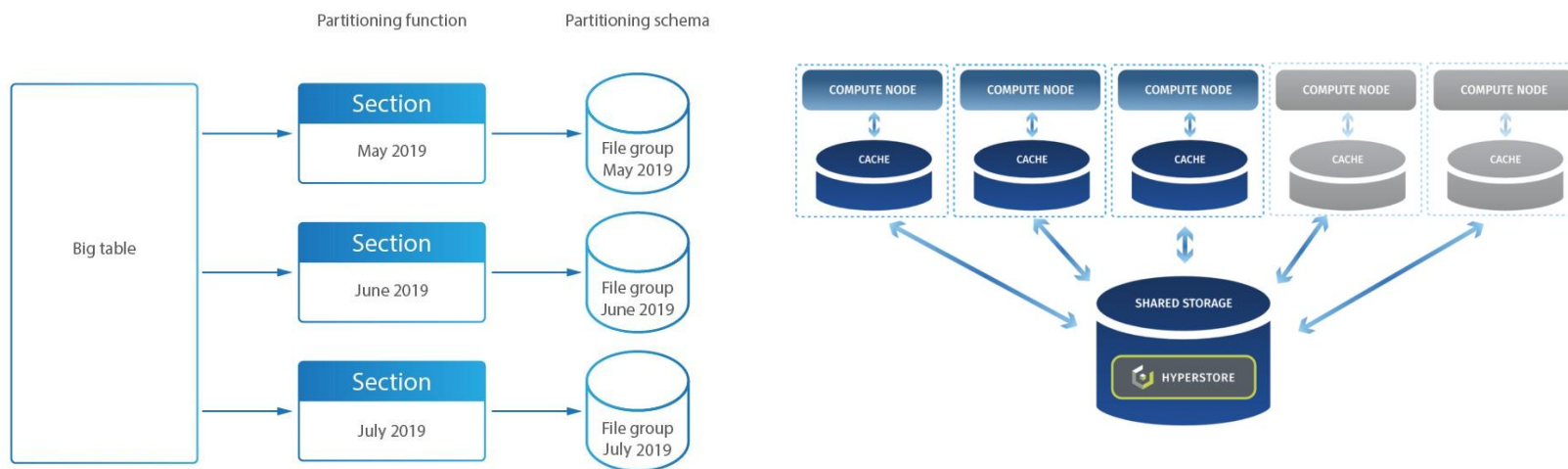
驱动案例



建材怎么摆放?

3.1 数据划分的意义

- 为什么要数据划分？
- 目的——可扩展性：可通过扩展集群机器规模提高系统性能、存储容量、计算能力的特性。
- 目标——高效性：数据分布在多个节点中并行存储和处理，提升系统存储和处理能力

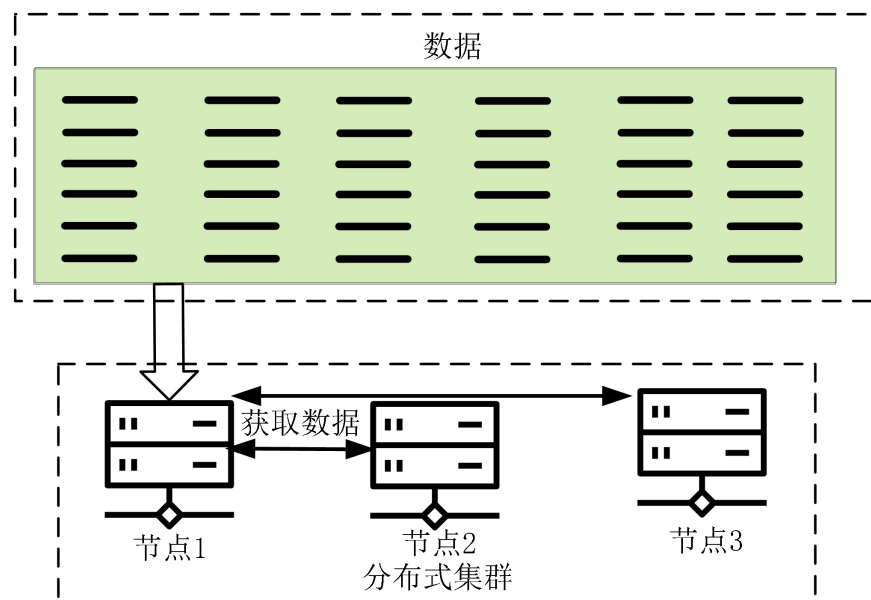


3.1.1 数据划分的目的

- 提高分布式系统的可扩展性

- 如果数据只存储在单个节点（节点1）上：

- 单个节点计算，处理性能低，系统瓶颈为节点1的处理性能，系统无法扩展

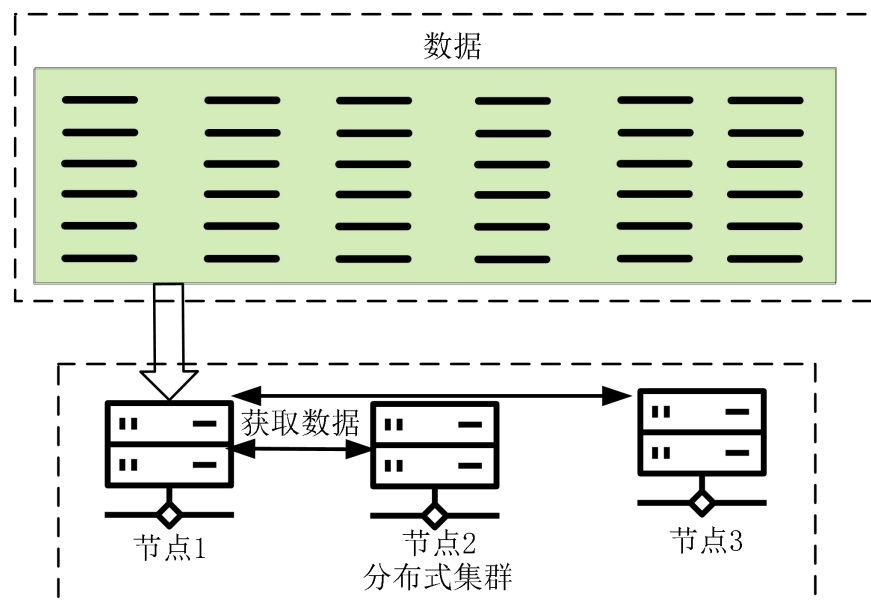


3.1.1 数据划分的目的

- 提高分布式系统的可扩展性

- 如果数据只存储在单个节点（节点1）上：

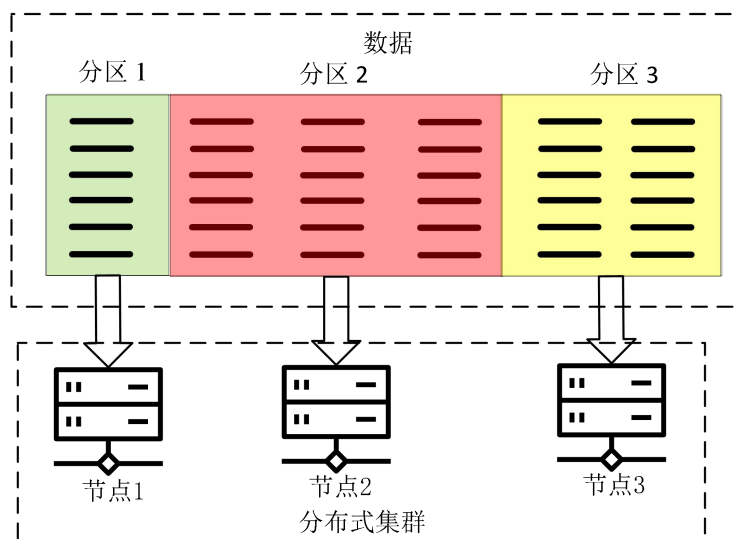
- 多个节点计算，数据传输负载过大系统的处理速度依赖于节点1的网络通信带宽



3.1.2 数据划分的目标

■ 负载均衡

- 合理分配数据，保证数据分布均衡
- 合理分配数据，保证负载分布均衡

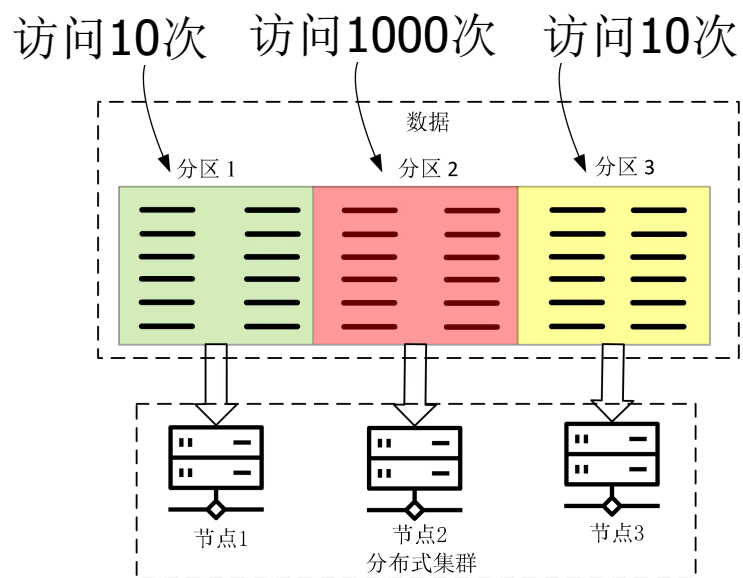


数据分布倾斜 (Data Skew)

3.1.2 数据划分的目标

■ 负载均衡

- 合理分配数据，保证数据分布均衡
- 合理分配数据，保证负载分布均衡



负载分布倾斜 (Load Skew)

不同数据类型

以元组为粒度的数据

```
mysql> select * from users order by created_at limit 100;
```

id	email	passwd	admin	name	image	created_at
0	test0@qq.com	password	1	test	about:blank	1637740065.19058
1	test1@qq.com	password	1	test	about:blank	1637740070.32804
2	test2@qq.com	password	1	test	about:blank	1637740070.62423
3	test3@qq.com	password	1	test	about:blank	1637740070.78949
4	test4@qq.com	password	1	test	about:blank	1637740070.85849
5	test5@qq.com	password	1	test	about:blank	1637740073.40182
6	test6@qq.com	password	1	test	about:blank	1637740073.44783
7	test7@qq.com	password	1	test	about:blank	1637740073.83338
8	test8@qq.com	password	1	test	about:blank	1637740073.91214
9	test9@qq.com	password	1	test	about:blank	1637740074.08125
10	test10@qq.com	password	1	test	about:blank	1637740074.1096
11	test11@qq.com	password	1	test	about:blank	1637740074.14928

文件数据



YOZO 产品介绍

文档转换引擎系统自动转换文件格式，无缝支持 doc、docx、wps、wpt、dot、rtf、xls、xlsx、et、ett、xlt、xlst、ppt、pptx、dps、dpt、pot 等主流格式的文件，通过后台配置能自动把本地或网络文档转换成 html、pdf、image 等格式，变成可在线预览文档；优化文件加载方式，doc、docx、xls 和 txt 文档预览按需获取、分页加载，减少服务器、带宽和终端压力；ppt、pptx 预览时，对左侧缩略图进行简化，大大减少服务器、带宽和终端压力，手机端预览支持拖拽缩略图

智能预览对转换过的文档提供在线预览服务，使其在各个终端自动适配，能智能分栏加载、平滑翻页，针对手机端能通过屏幕触控翻页，支持 doc、docx、xls、xlsx、ppt、pptx、pdf、txt、dwg、图片等格式；---dwg 支持预览支持网络文档，无需下载文档到本地，直接转换网络文档格式并转换为 html、pdf、image 等格式，变成可在线预览文档，直接在线预览详细内容。预览 doc、docx、ppt、pptx、dwg 和



第三章、数据划分和定位

01

3.1 数据划分的意义

02

3.2 静态数据划分方法

03

3.3 动态数据分区调整

04

3.4 数据定位

05

3.5 一致性哈希

06

3.6 分布式文件系统-GFS



3.2 静态数据划分方法

- **范围划分**

- 将关键字值位于一段连续区间内的数据划分为一个分区。

- **哈希划分**

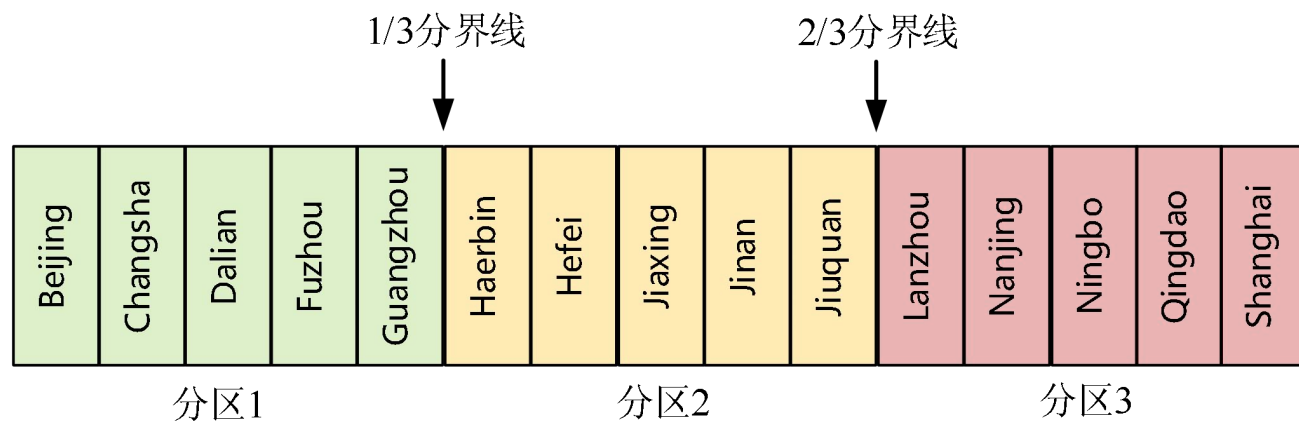
- 使用一个哈希函数，对数据进行哈希后得到一个随机数值，然后对所获得的随机数进行范围划分或根据分区数量进行取模划分。

- **其他划分**

- 轮询分区，列表分区，一致性哈希

3.2.1 范围划分

- 将关键字值位于一段连续区间内的数据划分为一个分区。
 - 1. 预先根据首字母排序好数据集，均匀划分



排序开销大

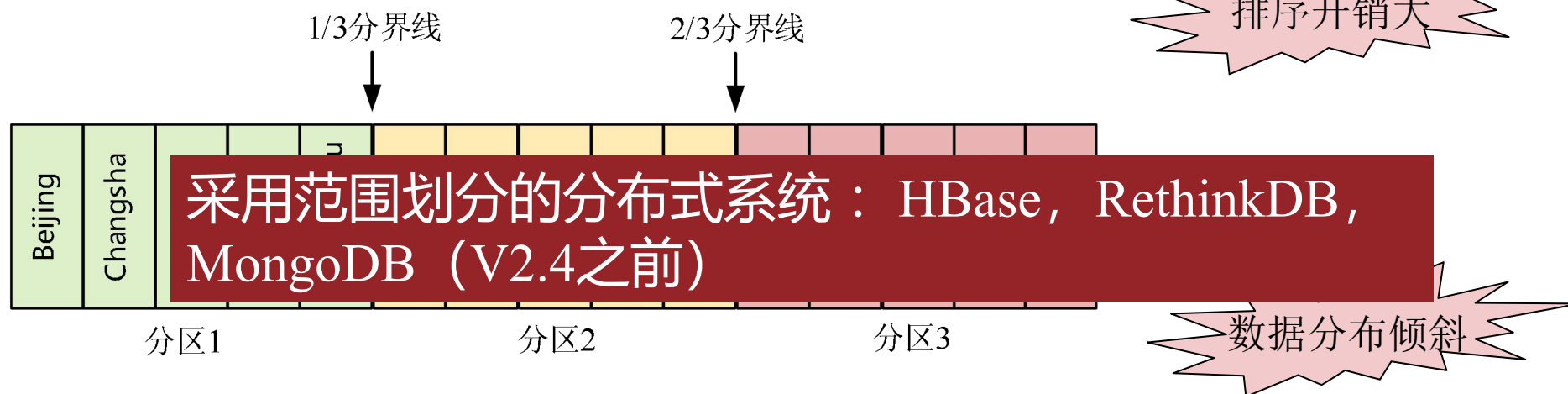
数据分布倾斜

- 2. 获得数据项的最大值和最小值，计算每个分区的分界线
- 3. 采样，近似估计范围划分的分位点

系统热点问题

3.2.1 范围划分

- 将关键字值位于一段连续区间内的数据划分为一个分区。
 - 1. 预先根据首字母排序好数据集，均匀划分

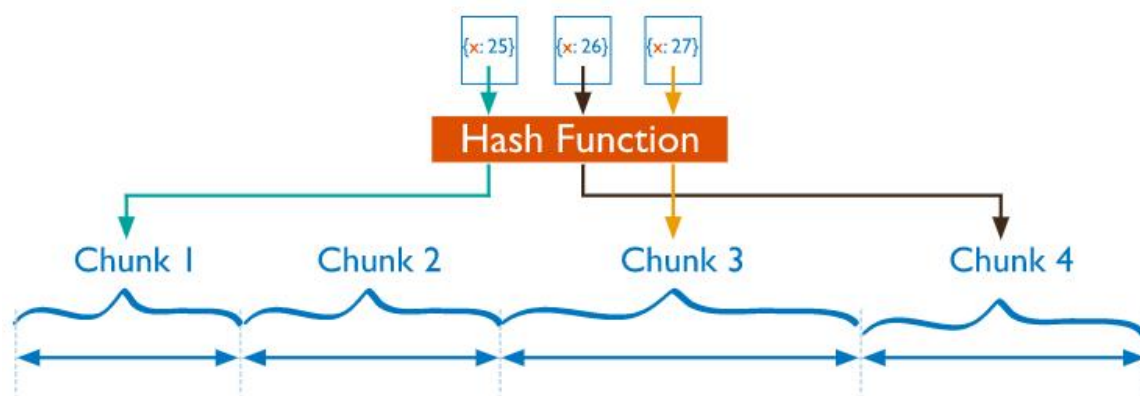


- 2. 获得数据项的最大值和最小值，计算每个分区的分界线
- 3. 采样，近似估计范围划分的分位点

系统热点问题

3.2.2 哈希划分

- 使用哈希函数将数据分配到不同的范围或数据块
 - 哈希函数将数据的某个属性或键映射为一个哈希值，然后根据哈希值进行划分

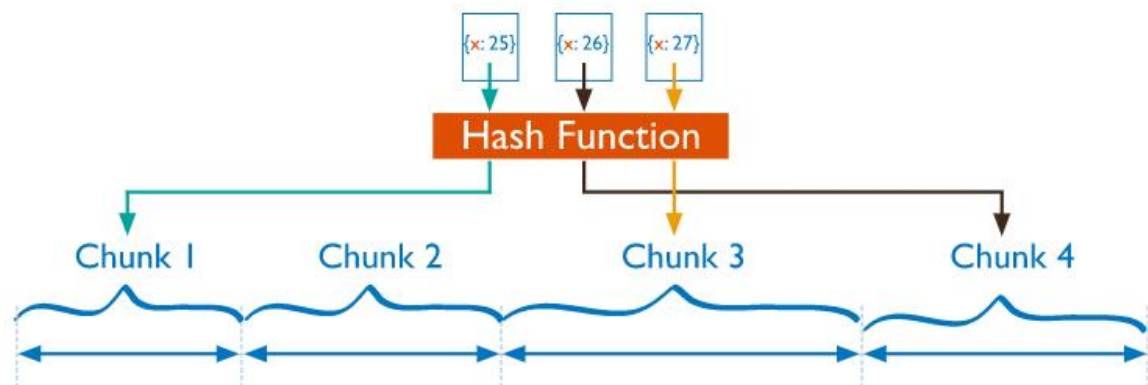


- 有效地避免系统热点问题
- 相邻的两条数据记录可能会被划分到不同的分区中，因此在执行区间查询时，需要在所有节点执行查询

混合分区策略

3.2.2 哈希划分

- 使用哈希函数将数据分配到不同的范围或数据块
 - 哈希函数将数据的某个属性或键映射为一个哈希值，然后根据哈希值进行划分



采用哈希划分的分布式系统：Cassandra, DynamoDB , MongoDB (V2.4之后)

- 有效地避免系统热点问题
- 相邻的两条数据记录可能会被划分到不同的分区中，因此在执行区间查询时，需要在所有节点执行查询

3.2.3 其他划分方法

■ 无分区

- 将整个数据完整复制多份，并在各个节点保存一份完整的数据副本

数据量较小、频繁访问、更改很少

■ 轮询分区

- 按数据读取顺序将数据循环分配到不同的分区中，每个分区内的数据量相同

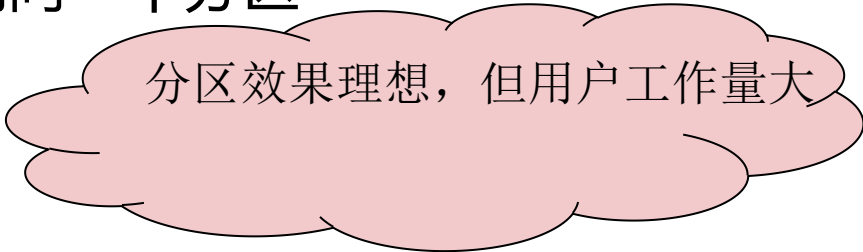
分区0	1	4	7			
分区1	2	5	8			
分区2	3	6	9			

利于负载均衡，随机性高不利于读

3.2.3 其他划分方法

- **列表分区**

- 用户手动指定哪些数据划分到同一个分区



分区效果理想，但用户工作量大

- **一致性哈希**

- 一种特殊的哈希划分方法
 - 可以自动完成负载均衡、动态负载均衡、数据定位等一系列功能
 - 在3.5详细阐述

3.2.4 热点困境

- 对某一个关键字中的数据频繁执行读写操作，无论采用何种划分方法，这些读写操作均会发生在同一个分区中
 - 例如，热点新闻的出现，使得某个关键字相关的讨论激增，从而导致相关的服务器负载过重
- 通过在关键字上添加后缀，实现将同一个关键字的数据存放在不同的分区中



划分同一关键字数据到不同分区会增加查询和读取的负担

第三章、数据划分和定位

01

3.1 数据划分的意义

02

3.2 静态数据划分方法

03

3.3 动态数据分区调整

04

3.4 数据定位

05

3.5 一致性哈希

06

3.6 分布式文件系统-GFS



3.3 动态数据分区调整

- **数据规模**发生变化，使得数据在节点之间进行重新划分和数据迁移
 - 数据量增大时，新数据与原数据该如何分配？
- **系统规模**发生变化，使得数据在节点之间进行重新划分和数据迁移
 - 节点故障时，故障节点的数据如何分配？

3.3 动态数据分区调整

- 要求
 - 数据在集群内分布更均匀，各节点上的工作负载更均衡
 - 动态调整过程中，系统能正常提供数据处理服务
 - 不同节点之间的数据迁移量要小，尽量降低网络和磁盘的I/O开销

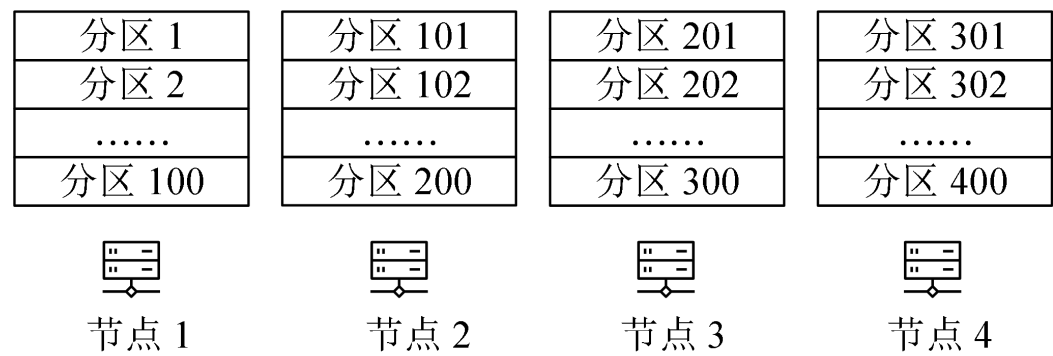
3.3 动态数据分区调整

一共有10个节点，对于待划分的值，采用模10的方法确定该值会被分配到哪个节点									
节点0	节点1	节点2	节点3	节点4	节点5	节点6	节点7	节点8	节点9
0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

仅剩9个节点，采用模9的方法确定值会被分配到哪个节点（标黄的值代表未迁移）								
节点0	节点1	节点2	节点3	节点4	节点5	节点6	节点7	节点8
0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89

3.3 动态数据分区调整

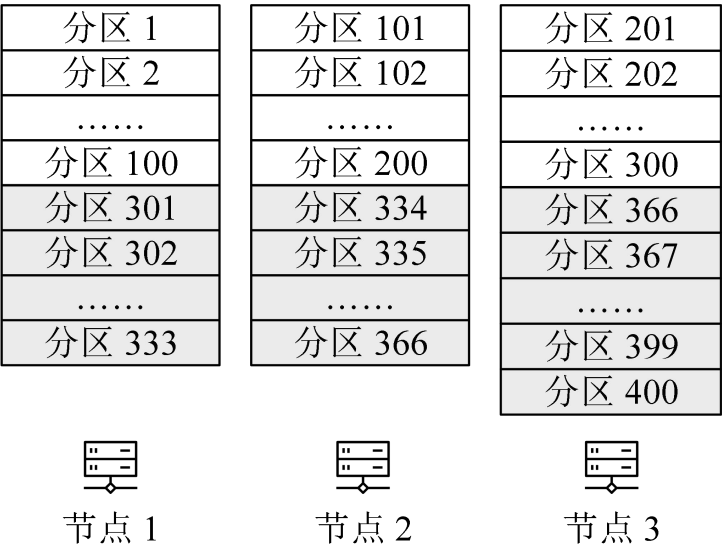
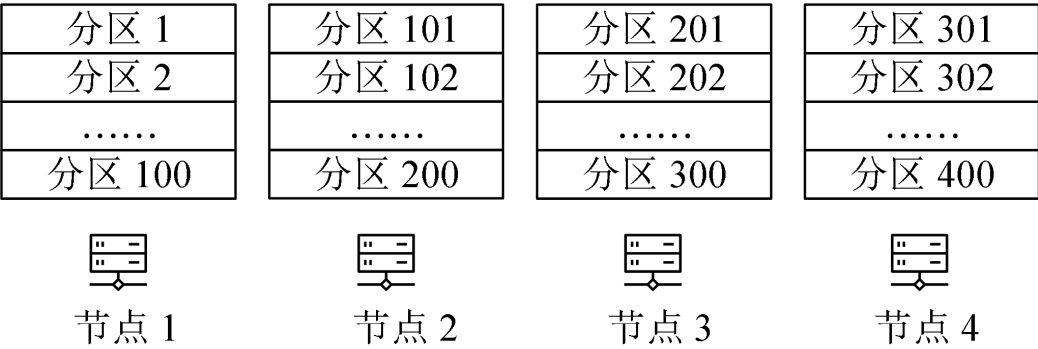
- 划分一些规模较小的分区，以小分区为粒度进行划分



- 限定分区数量
- 限定分区大小
- 限定分区比例

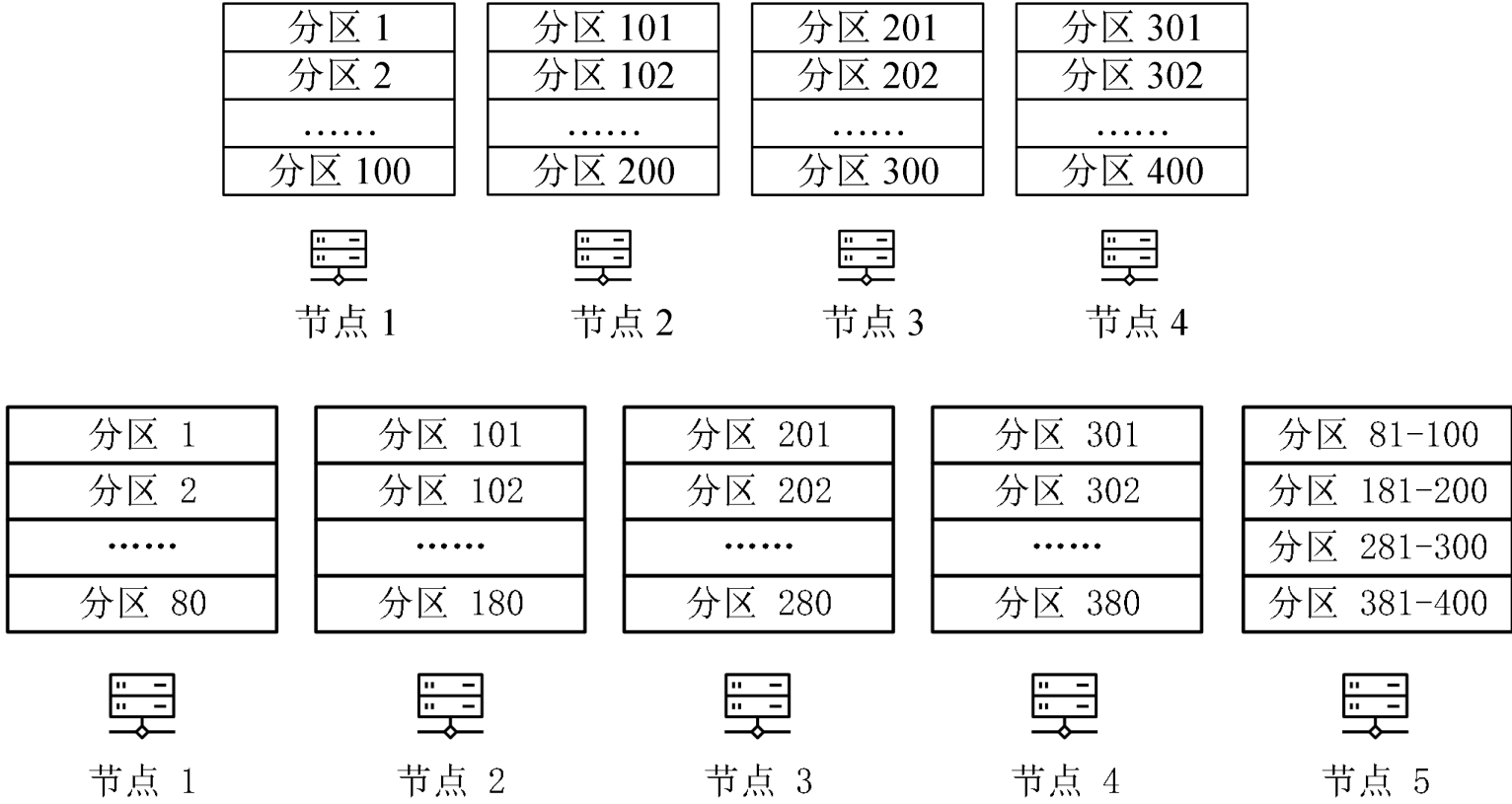
3.3.1 限定分区数量

- 节点删除时小分区的分配



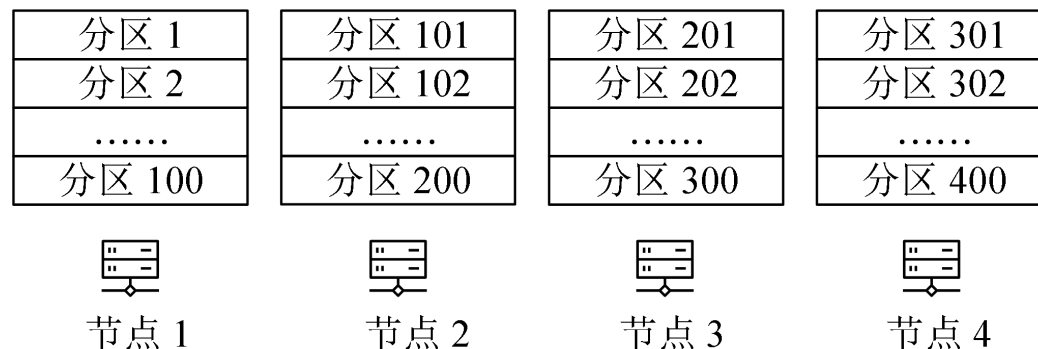
3.3.1 限定分区数量

■ 节点增加时小分区的分配



3.3.1 限定分区数量

Elasticsearch (ES)、Couchbase



- 数据增加时，采用小分区的划分方法插入现有小分区内
- 小分区在调整过程中并未发生变化，因此**关键字与所在分区的映射关系**也不会改变，改变的是小分区所在节点
- 在调整过程中，旧分区仍可处理读写操作
- 性能更强大的节点分配更多小分区，从而能分担更多负载

3.3.1 限定分区数量

- 优点
 - 数据增加时，采用小分区的划分方法插入现有小分区内
 - 小分区在调整过程中并未发生变化，因此**关键字与所在分区的映射关系**也不会改变，改变的是小分区所在节点
 - 在调整过程中，旧分区仍可处理读写操作
 - 性能更强大的节点分配更多小分区，从而能分担更多负载

3.3.1 限定分区数量

- 问题
 - 分区数量通常在系统初始时确定，之后不会改变（理论上可以改变，但为了操作简单，许多系统不支持），因此需要[预估数据集规模](#)，设置足够多的分区数
 - 过大的分区：数据再平衡、故障恢复代价增大
 - 过小的分区：管理开销增大

3.3.2 限定分区大小

- 动态创建小分区
 - 对每个小分区的大小设置了上限和下限阈值
 - 小分区内的数据量超过阈值时，**分裂**成两个数据量相等的分区
 - 小分区内的数据量小于阈值时，将其与相邻小分区**合并**（与B-Tree的分裂合并操作类似）

3.3.2 限定分区大小

- 优点

- 可以根据数据总量的大小来决定小分区数量。
 - 数据量较小时，只需要非常较少的小分区即可划分数据
 - 数据量较大时，每个小分区内的数据总量可以达到小分区限定的上限
- 分裂和合并的良好伸缩性有利于数据规模变化较大的情况

3.3.2 限定分区大小

- 问题
 - 每个节点上小分区数量不平衡时，在节点间迁移小分区
 - 初始阶段数据量极少时，容易导致只有一个小分区，负载不均衡
 - 设置一个预分区

HBase, RethinkDB、MongoDB

3.3.3 限定分区比例

- 不管限定分区数量还是限定分区大小，其小分区数量与节点数量均无关。
- 对于限定分区数量的方法，其分区数量在初始阶段就已经固定。
- 对于限定分区大小的方法，其小分区的数量只与数据量成正比，而与分布式系统中节点的数量无关。

3.3.3 限定分区比例

- 分区方法
 - 每个节点上设置固定数量的小分区（例如，Cassandra每个节点默认有256个小分区）
 - 每个小分区的数据量与总数据量成正比
- 增加节点时
 - 随机选择固定数量的其他分区进行拆分，拿走这些分区的一半数据量，将另一半数据留在原节点
 - Cassandra 3.0引入优化算法，可避免不公平的拆分

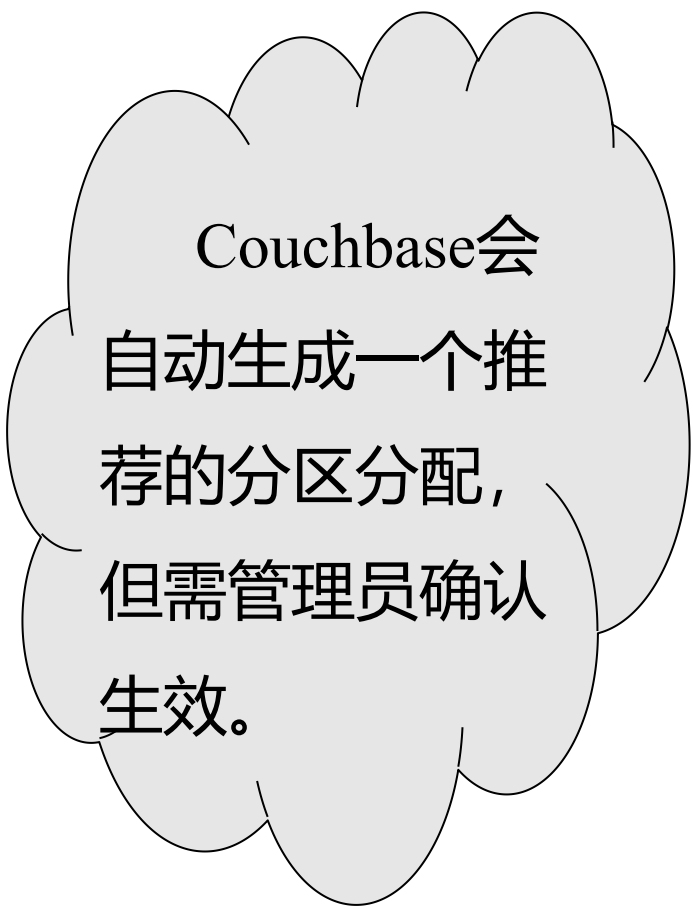
手动 or 自动再平衡

- 手动

- 分区到节点的映射由管理员显式配置

- 自动

- 由系统自动决定，何时将分区从一个节点迁移到另一个节点，无须人工干预
 - 再平衡是一个资源密集型操作，需要重新路由请求和迁移大量数据，可能导致网络或节点负载过重，降低系统性能，甚至可能引发系统故障误判



Couchbase会
自动生成一个推
荐的分区分配，
但需管理员确认
生效。

第三章、数据划分和定位

01

3.1 数据划分的意义

02

3.2 静态数据划分方法

03

3.3 动态数据分区调整

04

3.4 数据定位

05

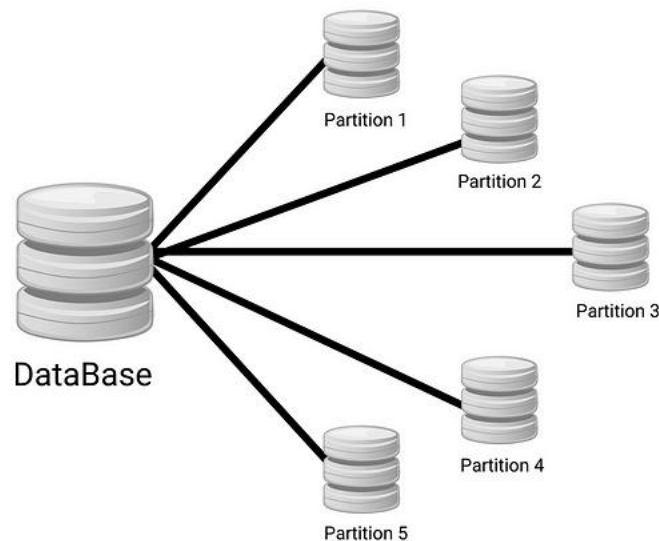
3.5 一致性哈希

06

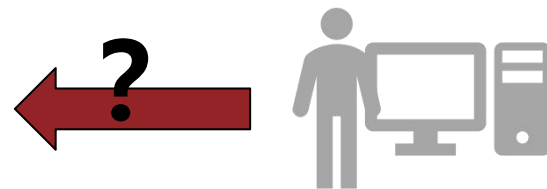
3.6 分布式文件系统-GFS



3.4 数据定位



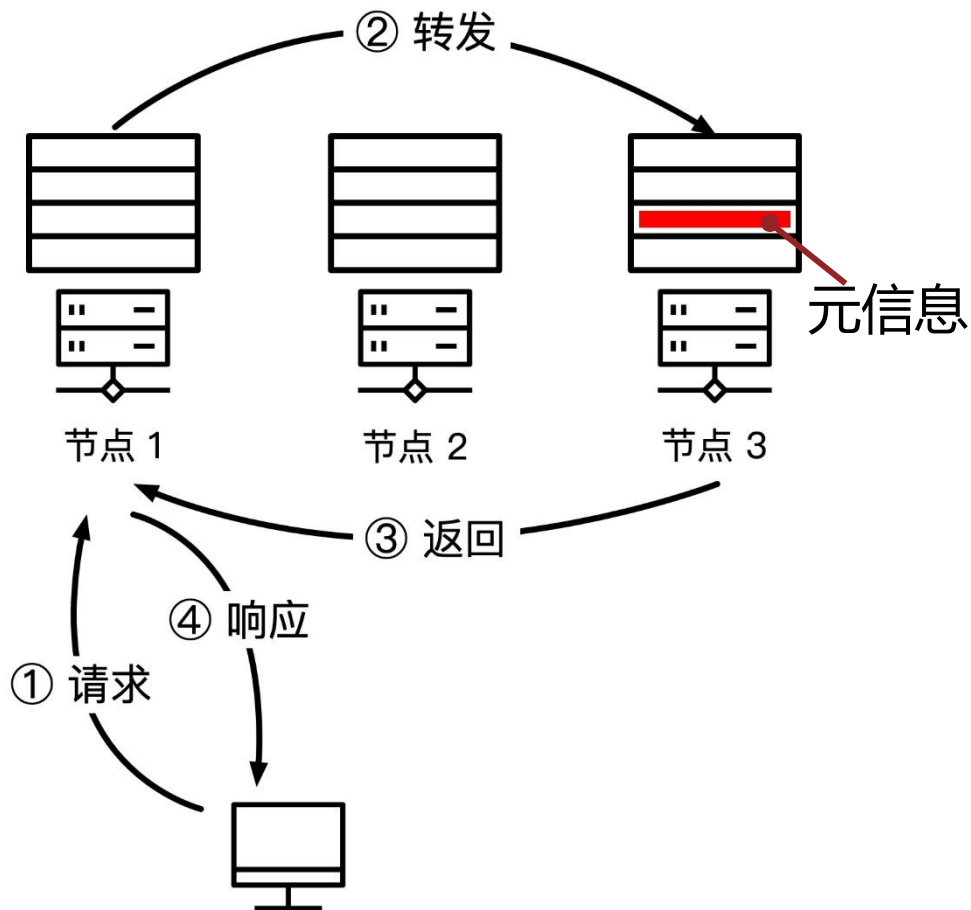
- 快速、准确地找到请求的数据所在的位置，以便进行读取、写入或其他操作



- 元信息
- 每个小分区的标识符、包含的数据范围
- 每个小分区当前所在的节点信息（节点IP、性能等）
- 分区与节点的映射关系
- 版本信息或时间戳
- 数据的冗余存储所需的副本信息

3.4 数据定位

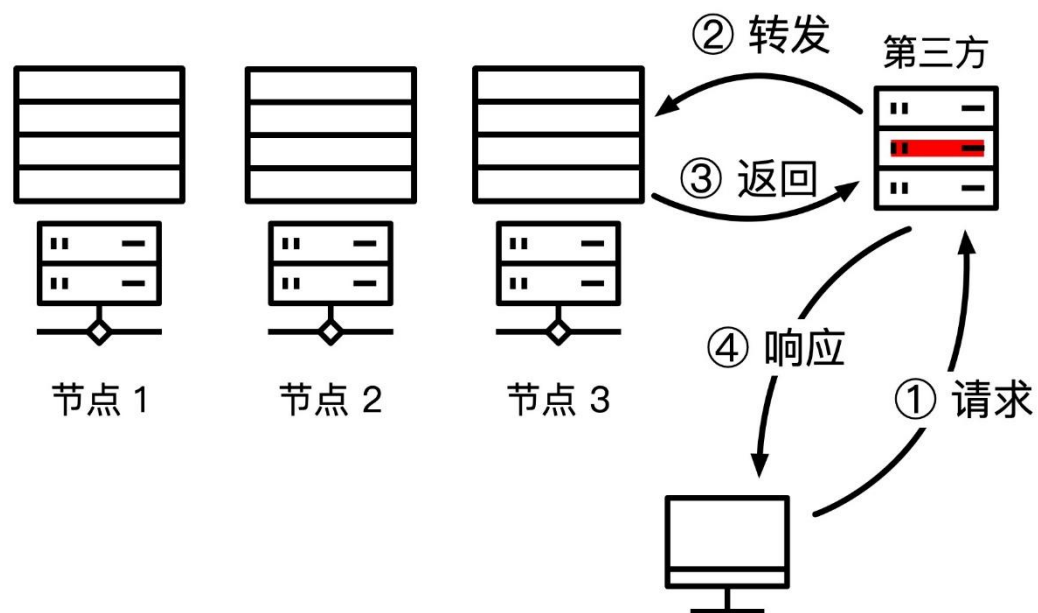
■ 元信息存储在分布式节点中



- 元信息与数据分布在不同节点上
- 客户端能随机连接到任意一个节点
- 需要同步集群中节点的元信息，增加了节点的负载和复杂性
- **Cassandra、Riak**

3.4 数据定位

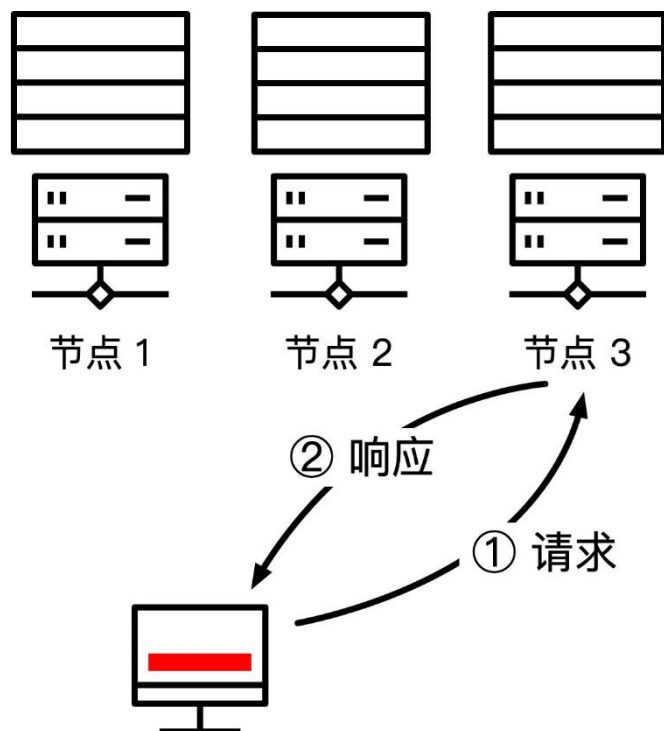
■ 元信息存储在第三方



- 客户端连接到节点以外的一个第三方独立机器
- 第三方充当客户端与分布式系统之间的路由层
- Hadoop, HBase, Kafka中的ZooKeeper

3.4 数据定位

■ 元信息存储在客户端



- 每个客户端负责管理与其相关的元数据
- 客户端与数据所在节点的直接连接

第三章、数据划分和定位

01

3.1 数据划分的意义

02

3.2 静态数据划分方法

03

3.3 动态数据分区调整

04

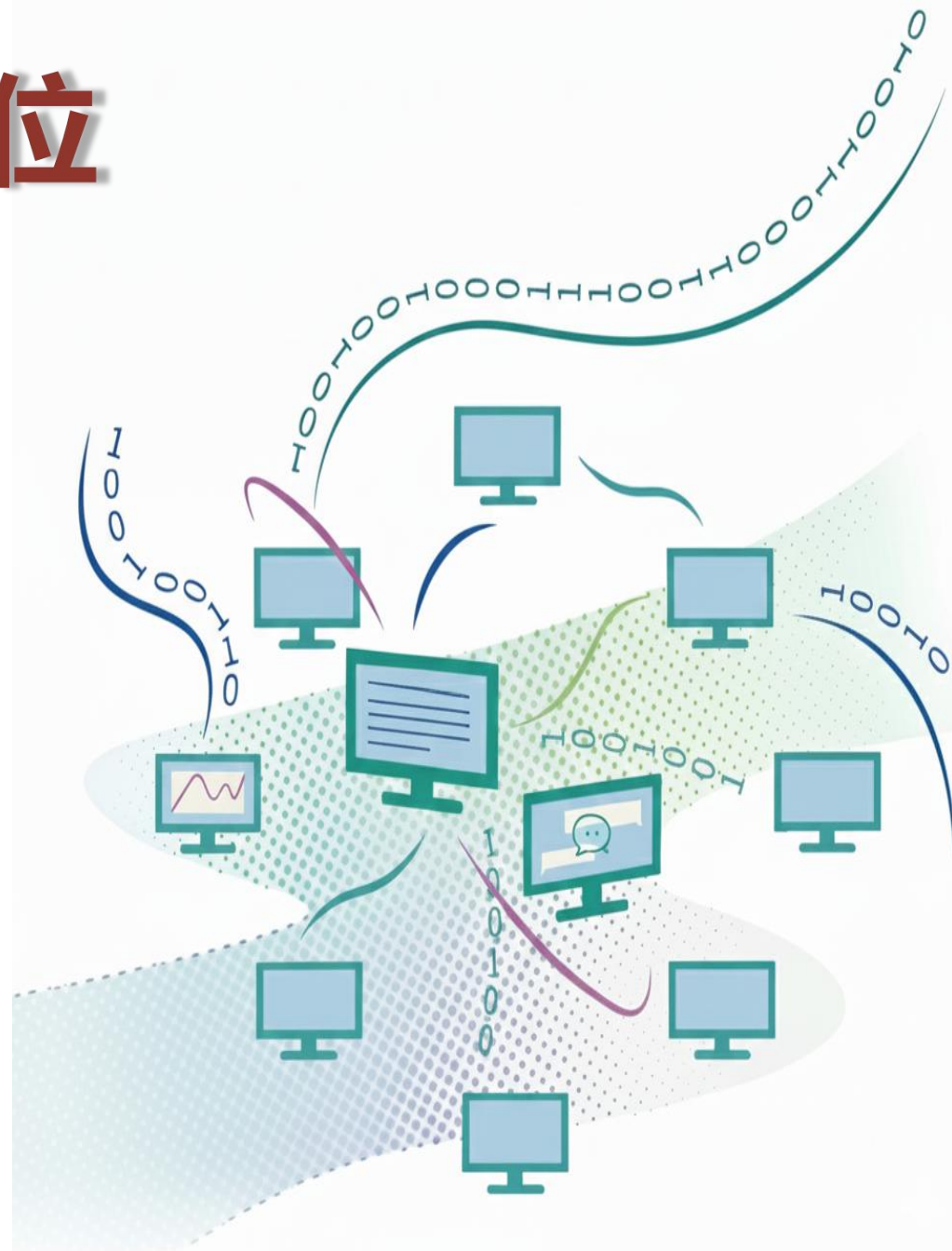
3.4 数据定位

05

3.5 一致性哈希

06

3.6 分布式文件系统-GFS



3.5 一致性哈希

- 动态分区中，简单的哈希划分：
 - 包含10个节点的分布式集群，采用 $k = N \% 10$ 来决定分区
 - 一个节点宕机后，所有数据采用 $k = N \% 9$ 决定分区
 - 大约有 $81/90 \approx 89\%$ 的数据量需要迁移
- 3.4中的数据定位方法需要维护大量的**数据与小分区**之间的映射关系和**小分区与节点**之间的映射关系等元数据。



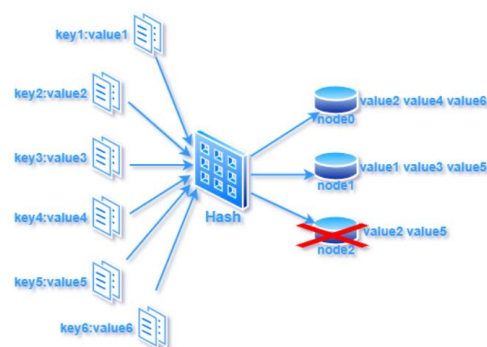
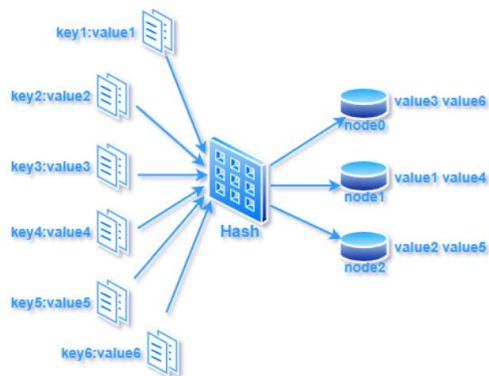
大量数据迁移



维护大量元数据

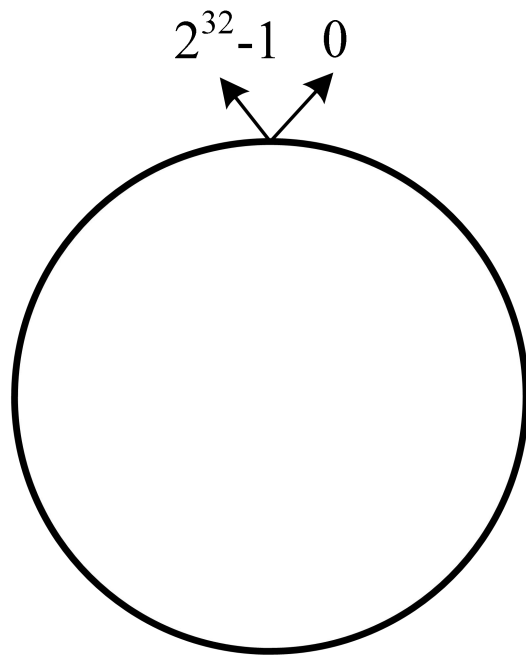
3.5 一致性哈希

- 一致性哈希是1997年由麻省理工学院的Karger等人提出的，其最初目的是解决内容分发网络等互联网的缓存问题。
- 当网络中添加或者移除一个服务器时，该算法能够在尽可能减少数据在服务器间移动的情况下，达到新的均衡状态。这样可以尽可能小地改变已存在的服务请求与处理请求服务器之间的映射关系。



3.5.1 一致性哈希算法

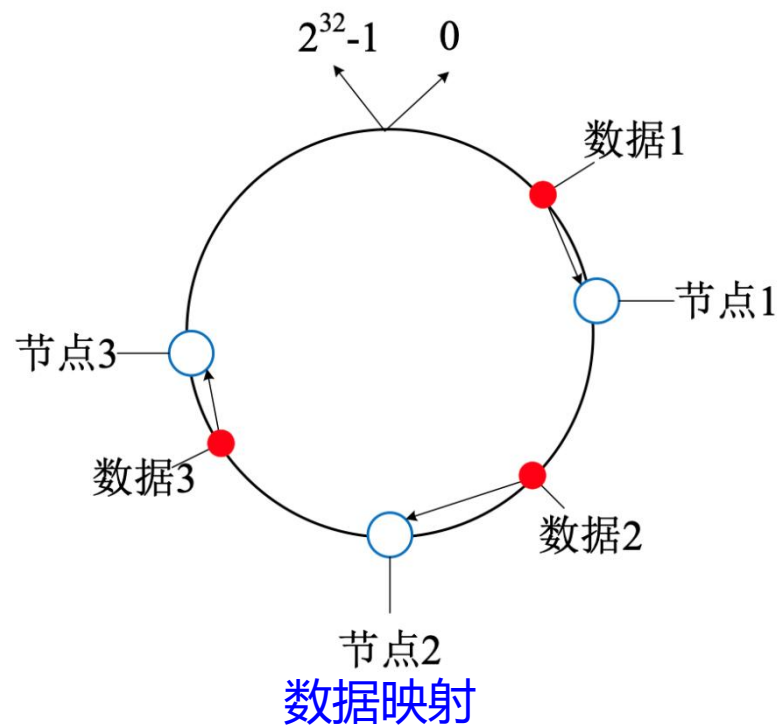
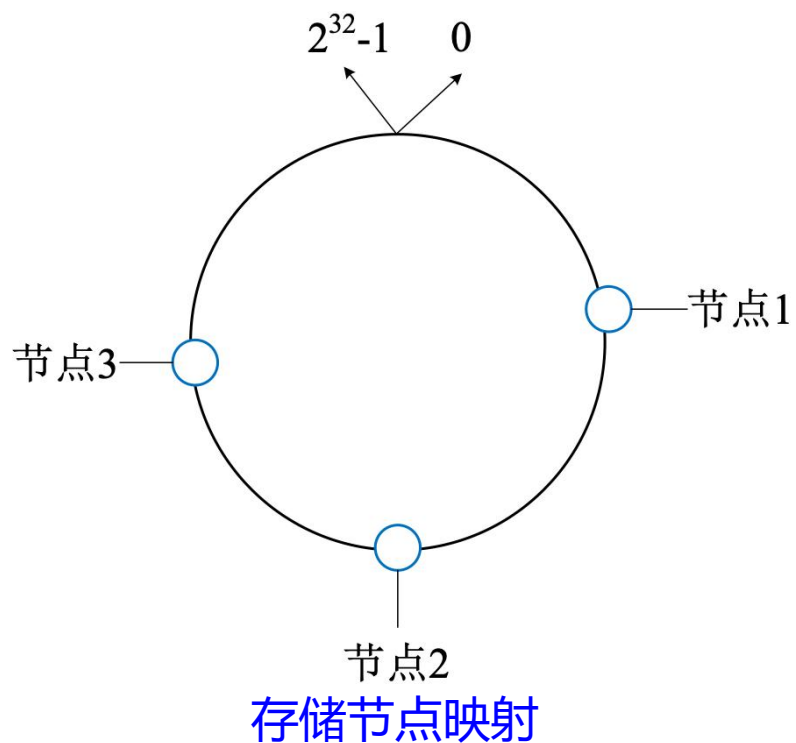
- 哈希环



- 哈希值空间为 $0 \sim 2^{32}-1$
- 一个按顺时针方向组织的虚拟圆环
- 0 与 $2^{32}-1$ 在零点方向重合

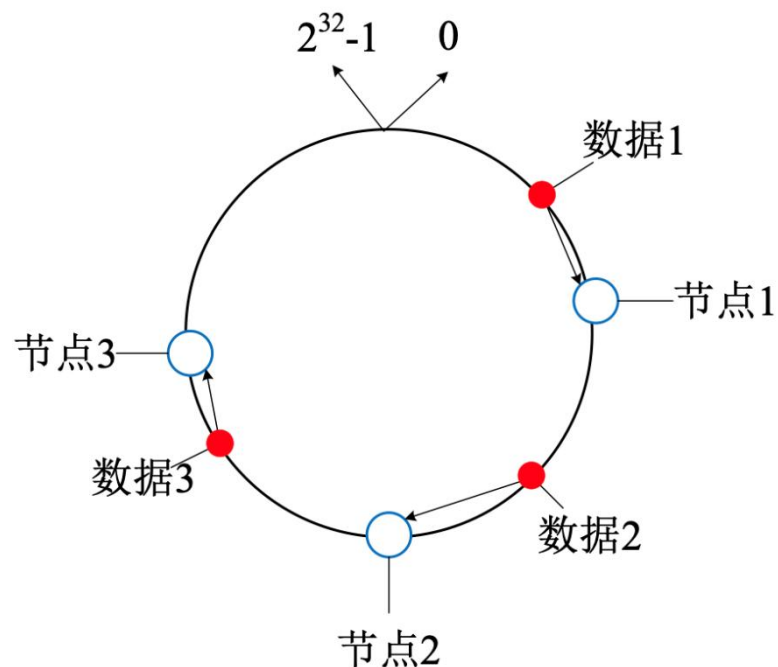
3.5.1 一致性哈希算法

- 一致性哈希的数据划分过程
 - 1. 对存储节点进行哈希
 - 2. 对数据记录进行哈希，按顺时针方向存储



3.5.2 一致性哈希数据定位

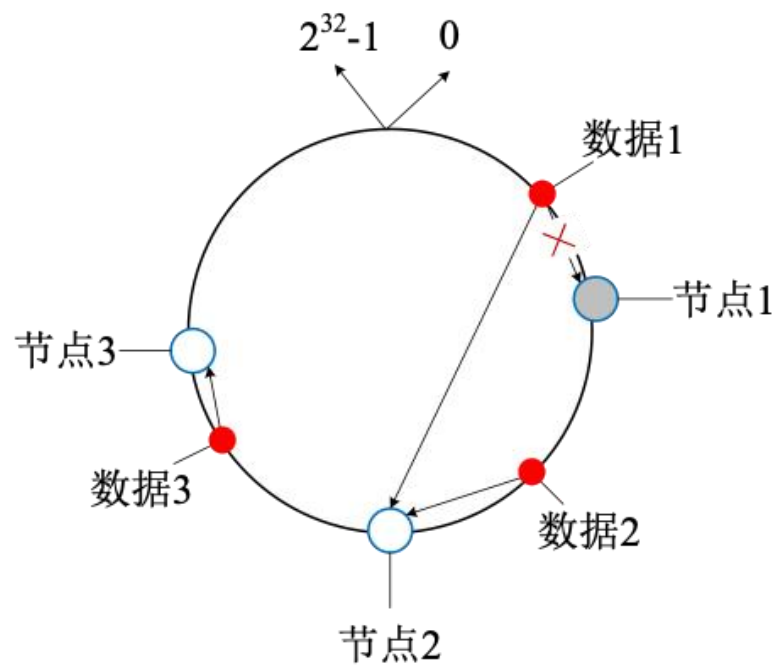
- 1.定位数据记录所在的节点
- 2. 触发该节点的查询操作，返回查询结果



- 查询数据2时
- 计算数据2的哈希值在环上的位置
- 比较其哈希值与存储节点的哈希值，采用**二分查找**，可以发现在节点1和节点2的哈希值之间，则说明存储在节点2中。

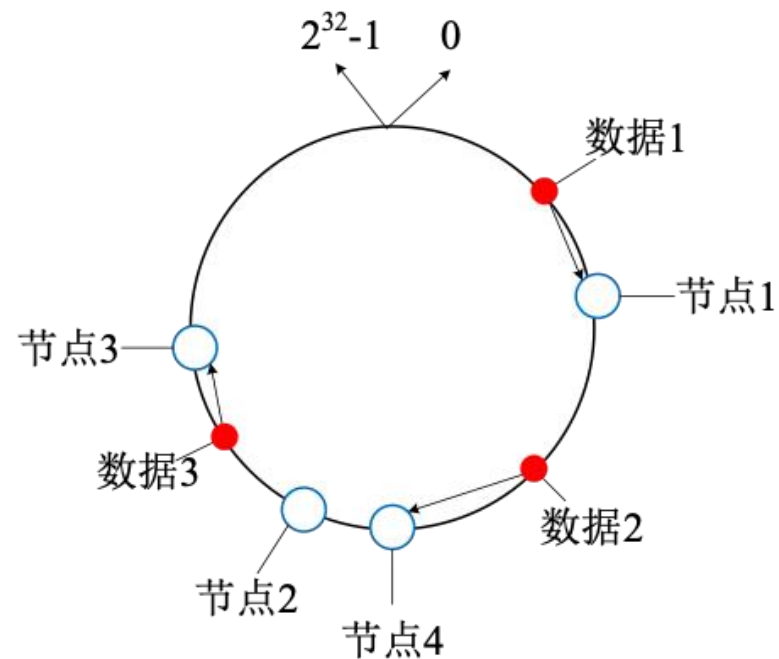
3.5.3 数据再划分

■ 节点删除



节点1故障

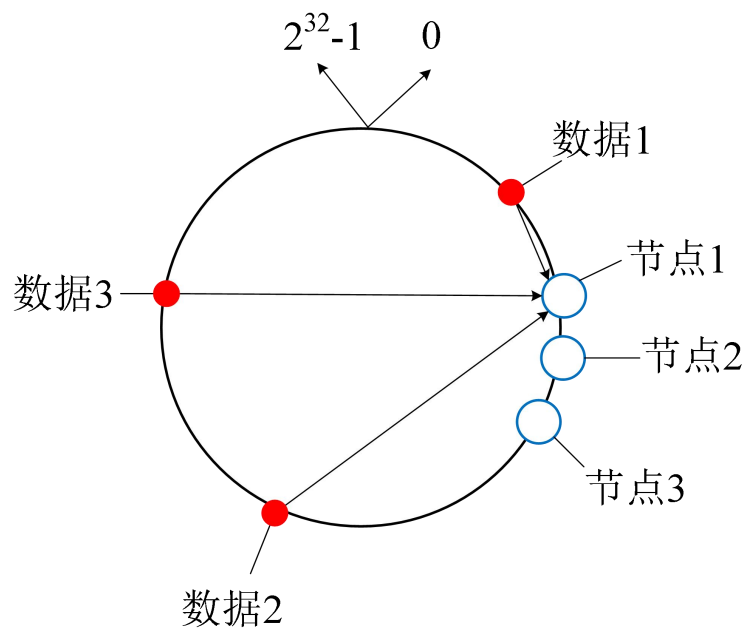
■ 节点新增



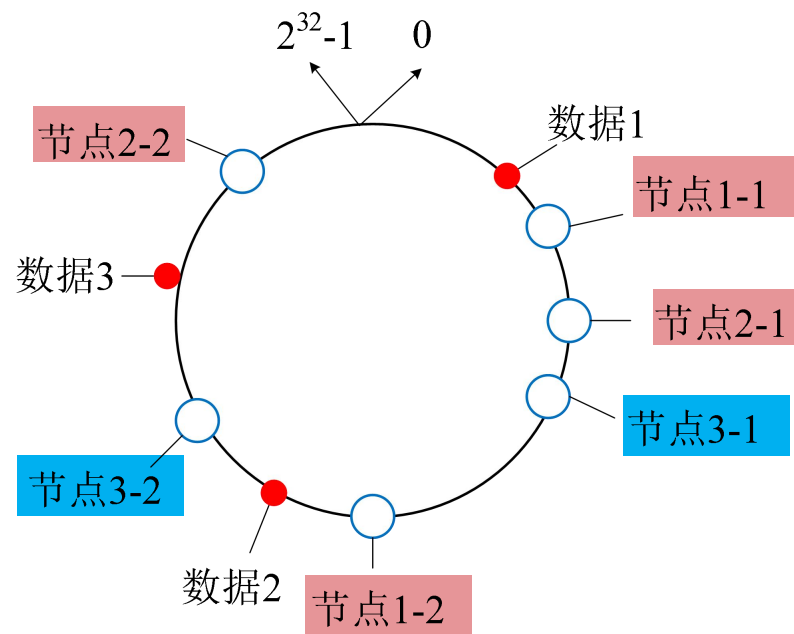
新增节点4

3.5.4 数据倾斜与虚拟节点

- 数据倾斜
- 哈希函数选择不当，大部分数据划分到少数节点



- 虚拟节点
- 将每个物理服务节点映射为多个虚拟节点



第三章、数据划分和定位

01

3.1 数据划分的意义

02

3.2 静态数据划分方法

03

3.3 动态数据分区调整

04

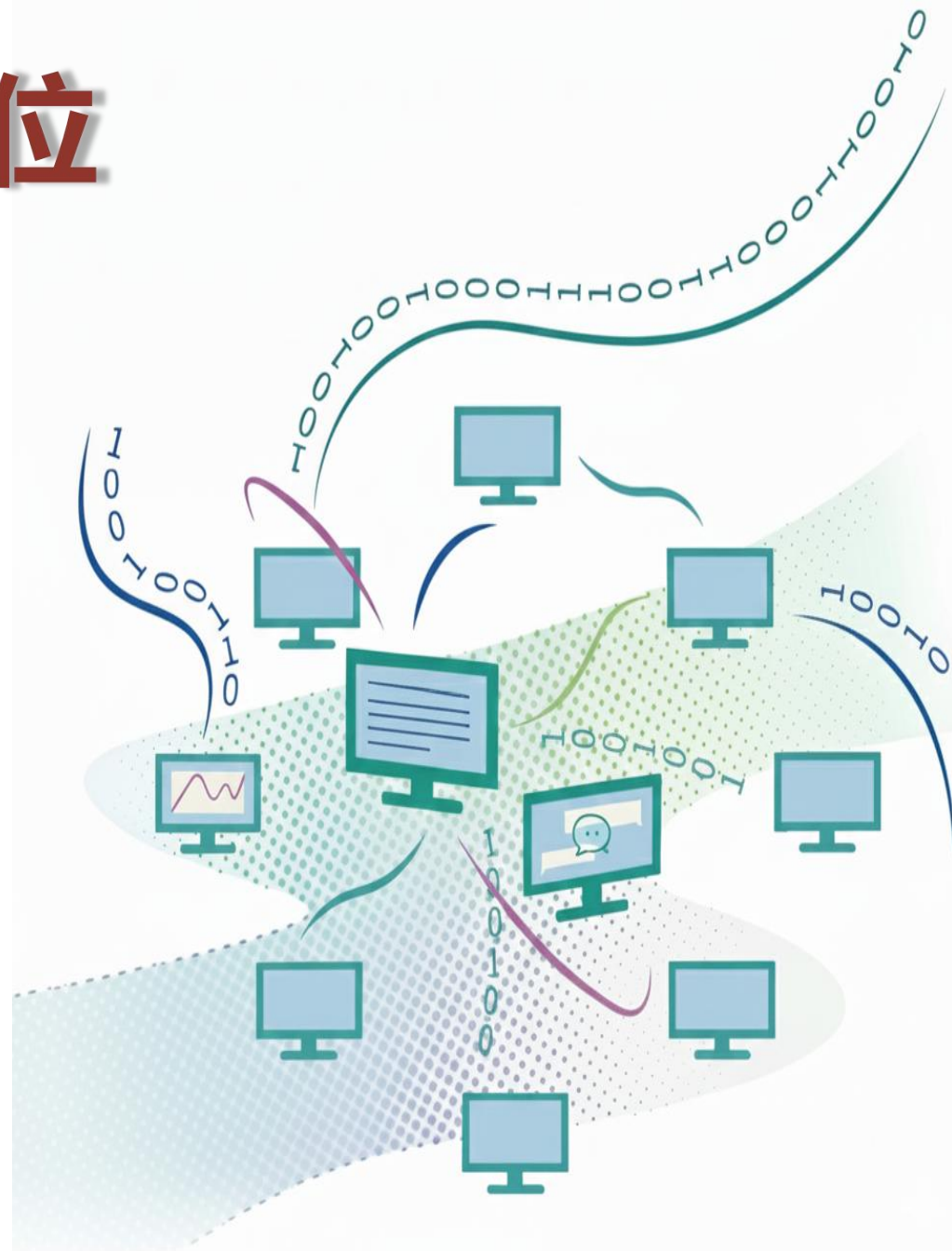
3.4 数据定位

05

3.5 一致性哈希

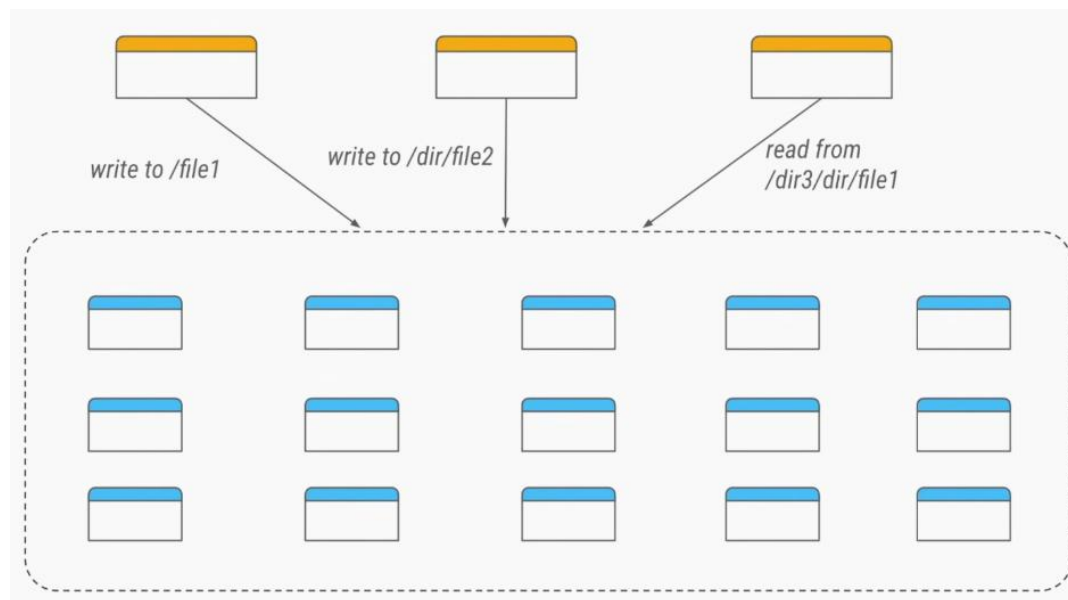
06

3.6 分布式文件系统-GFS



3.6 分布式文件系统-GFS

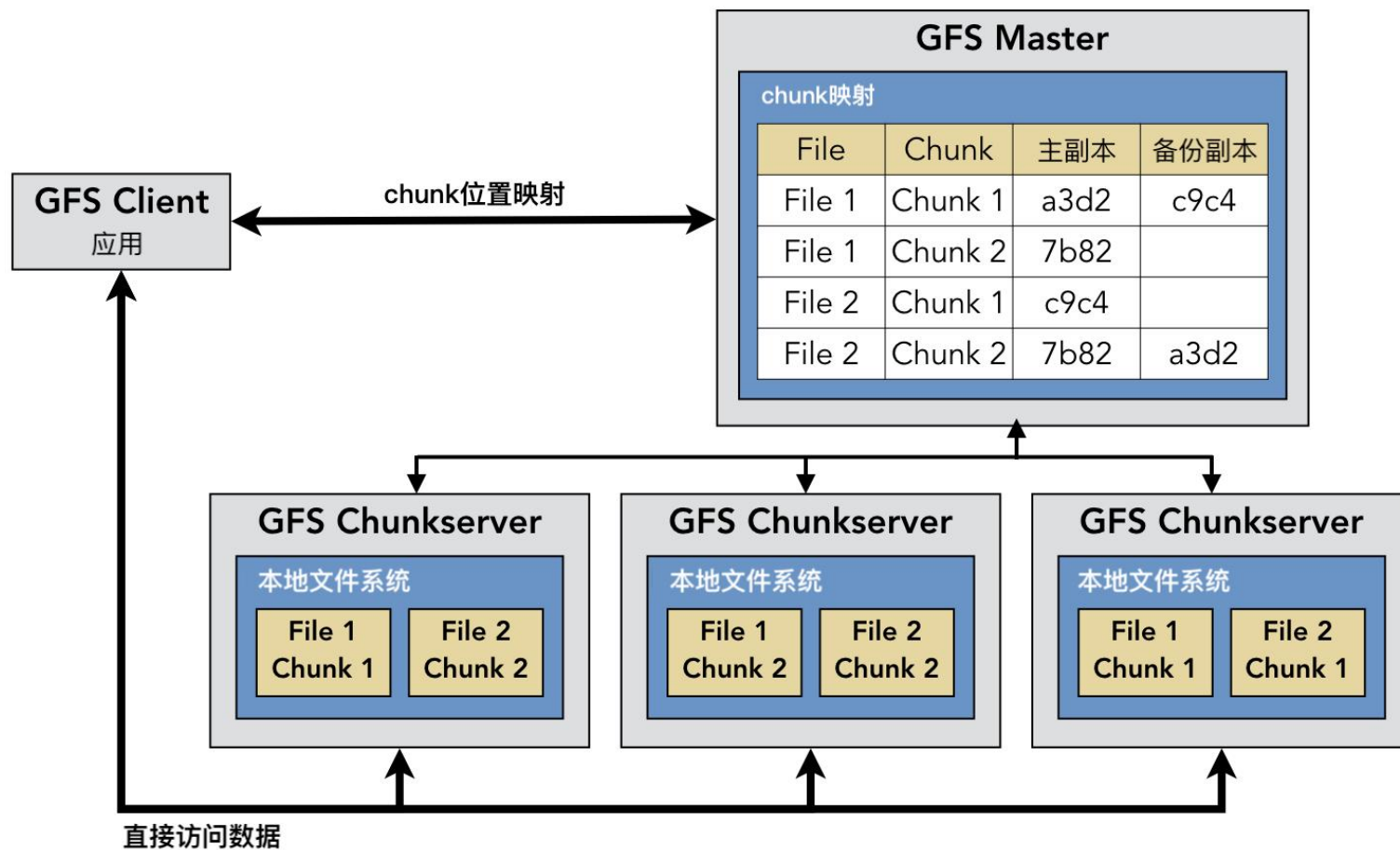
- **分布式文件系统 (Distributed File System, DFS)**
 - 将文件分布在多个物理节点上，进行存储与管理
 - 用户访问这些文件时，就像它们存储在单一的本地文件系统中一样。



■ 著名的分布式文件系统

- Hadoop Distributed File System (HDFS): Doug 根据 GFS 论文的指导
- Ceph: Sage Weil 的博士论文项目
- Lustre: Peter Braam 的一个研究项目
- Amazon S3 (Simple Storage Service), EFS 等产品
-

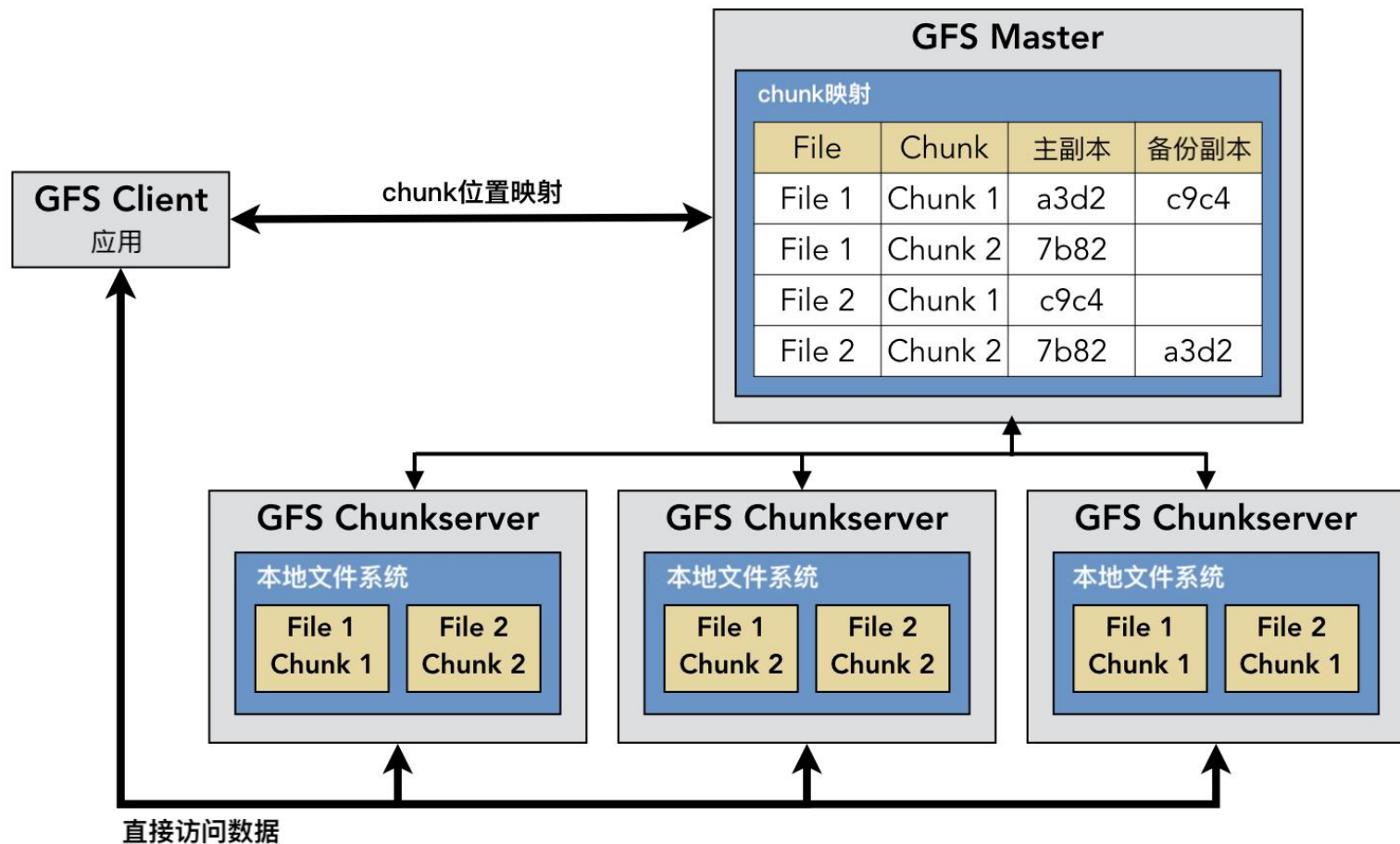
3.6.1 GFS架构

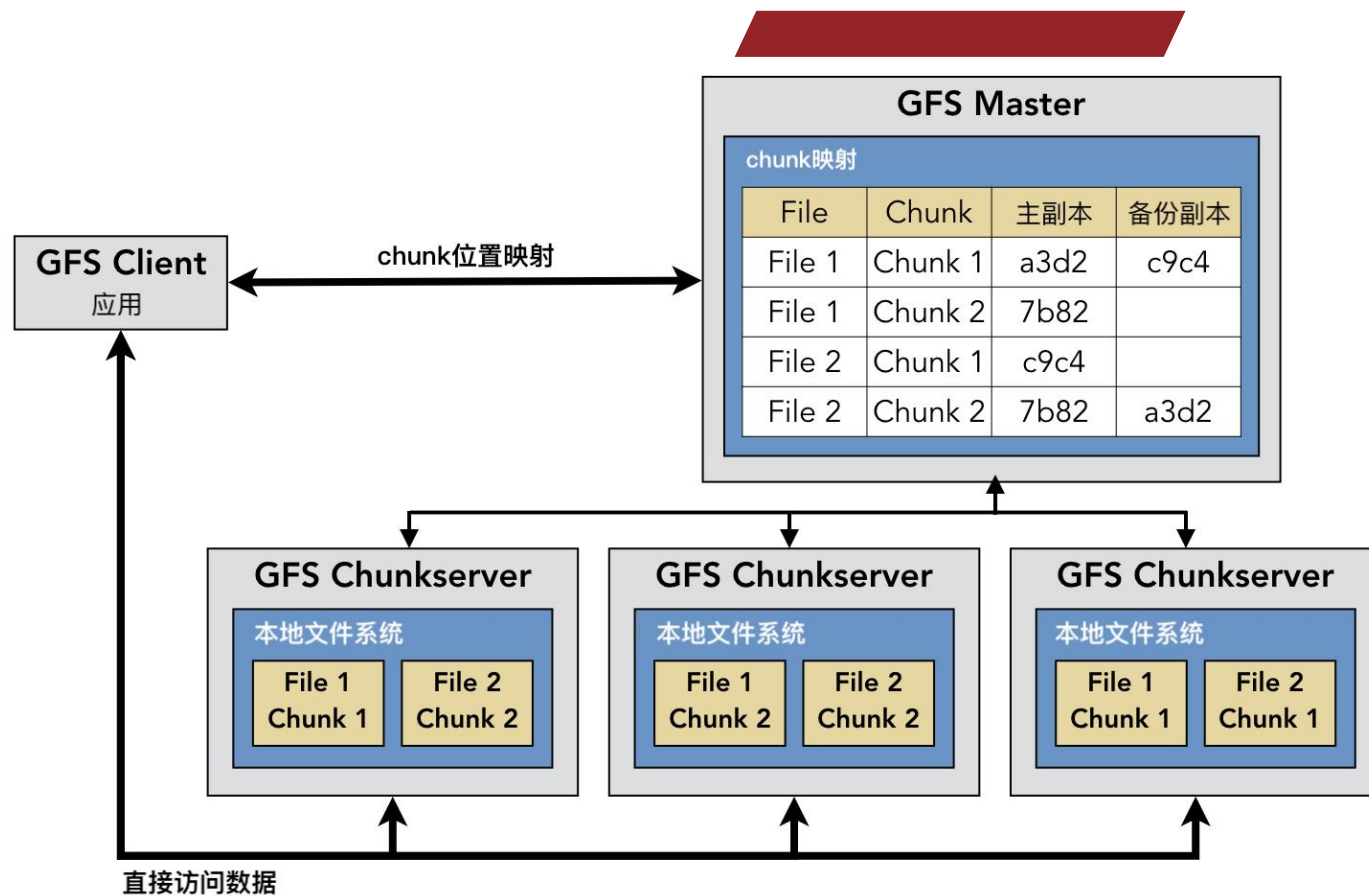


- 主从架构
- 三个重要角色

3.6.1 GFS架构

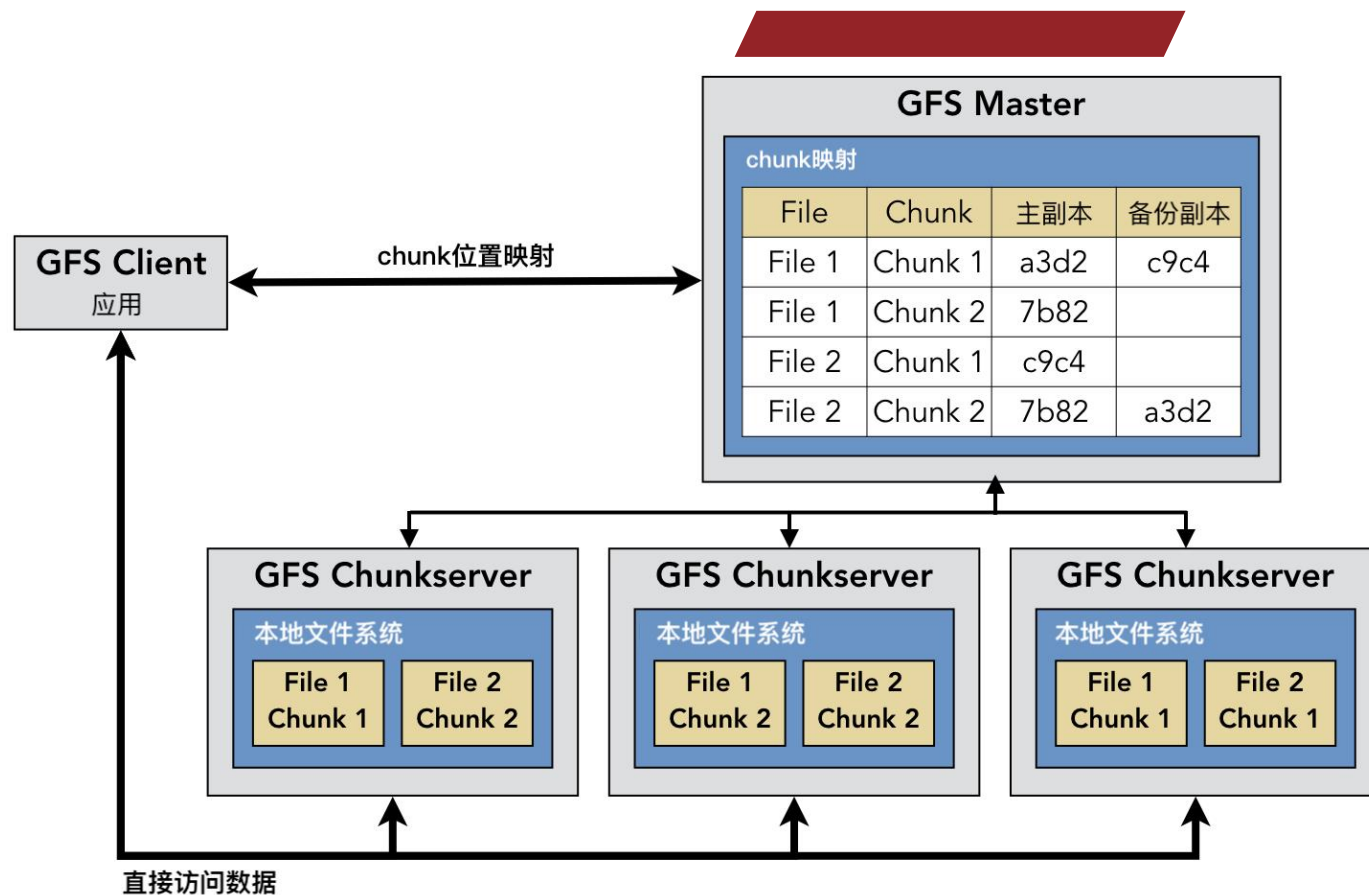
- 块 (Chunk)
 - 数据管理的最小单元
 - 大小通常为64MB
 - 每个chunk默认存储多个副本（通常是3个）在不同的chunk server上



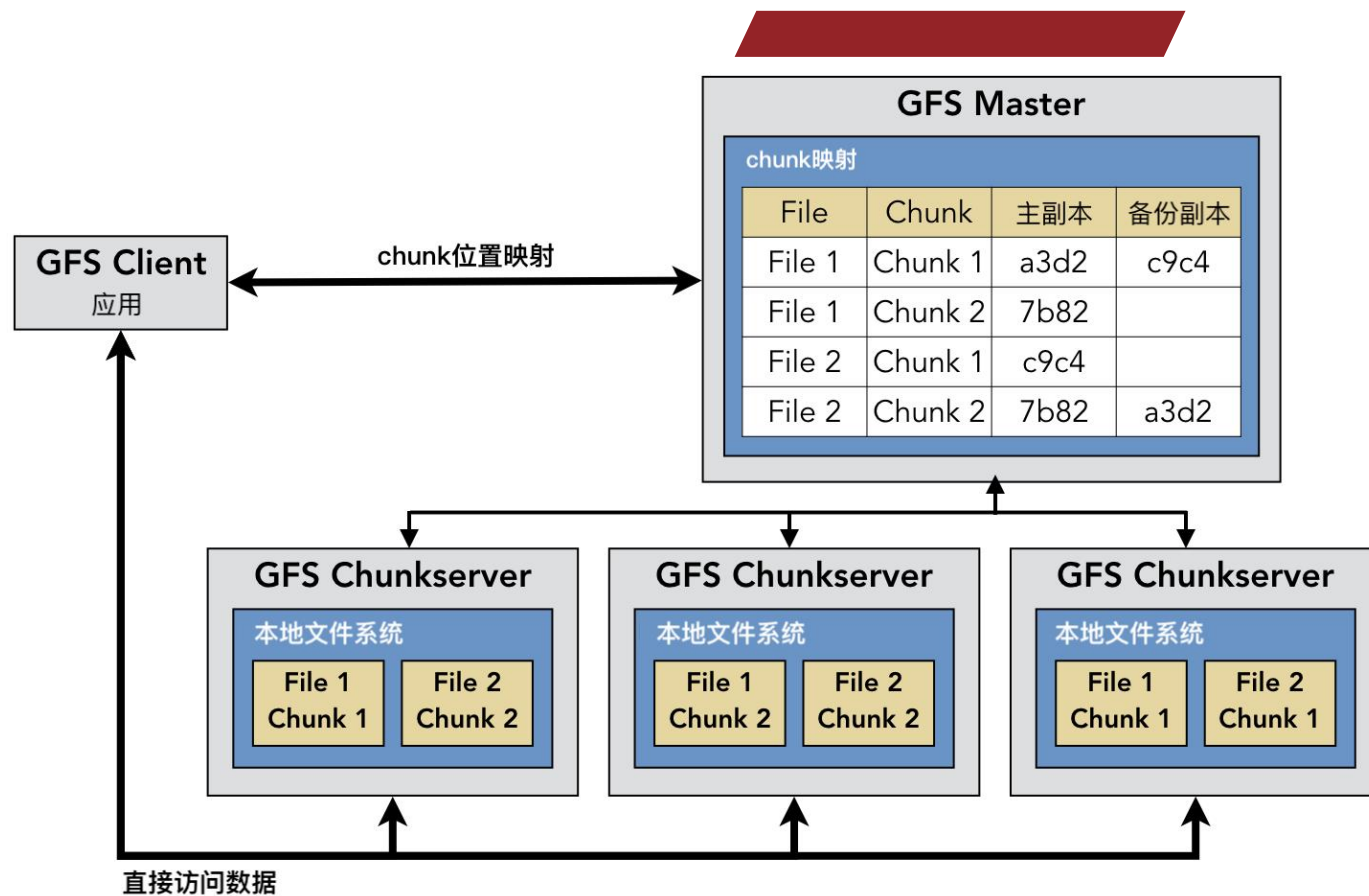


■ 块服务器 (Chunk Server)

- 存储chunk
- 处理读写请求
- chunk副本管理
- 心跳和状态报告

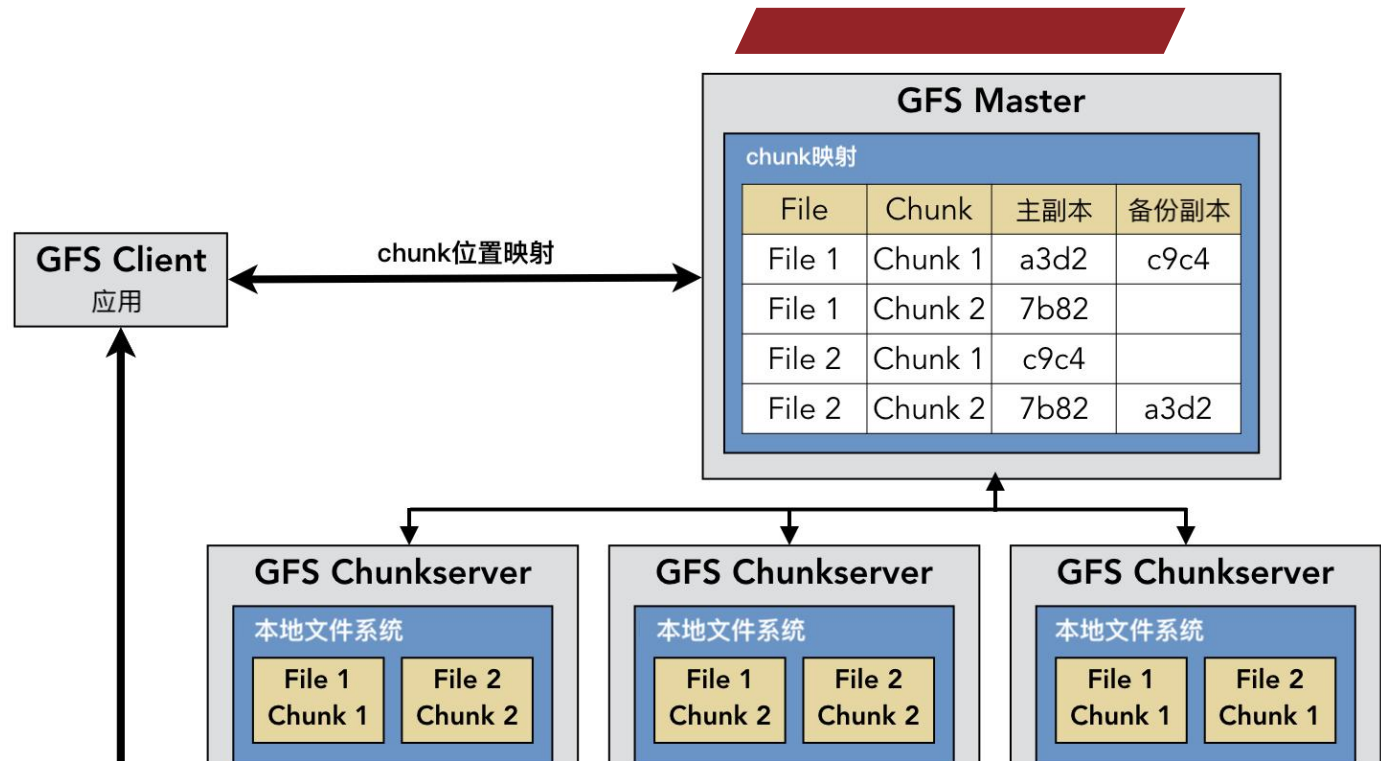


- 元数据 (Metadata) (存储于master中)
 - 文件和目录信息
 - 块的位置信息
 - 文件到chunk的映射



■ 客户端 (Client)

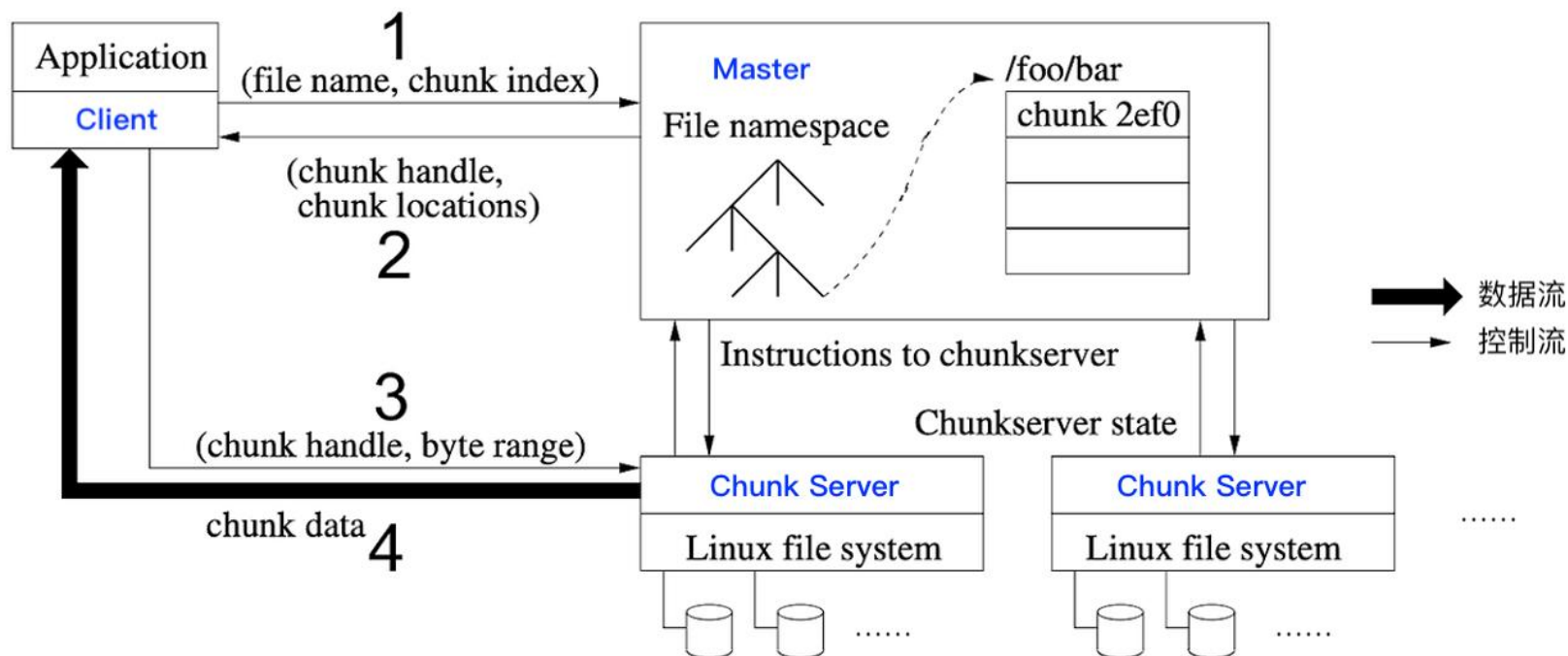
- 请求发起者
- 与master、chunk server通信
- 一致性和错误处理
- 缓存管理



■ 主节点 (Master)

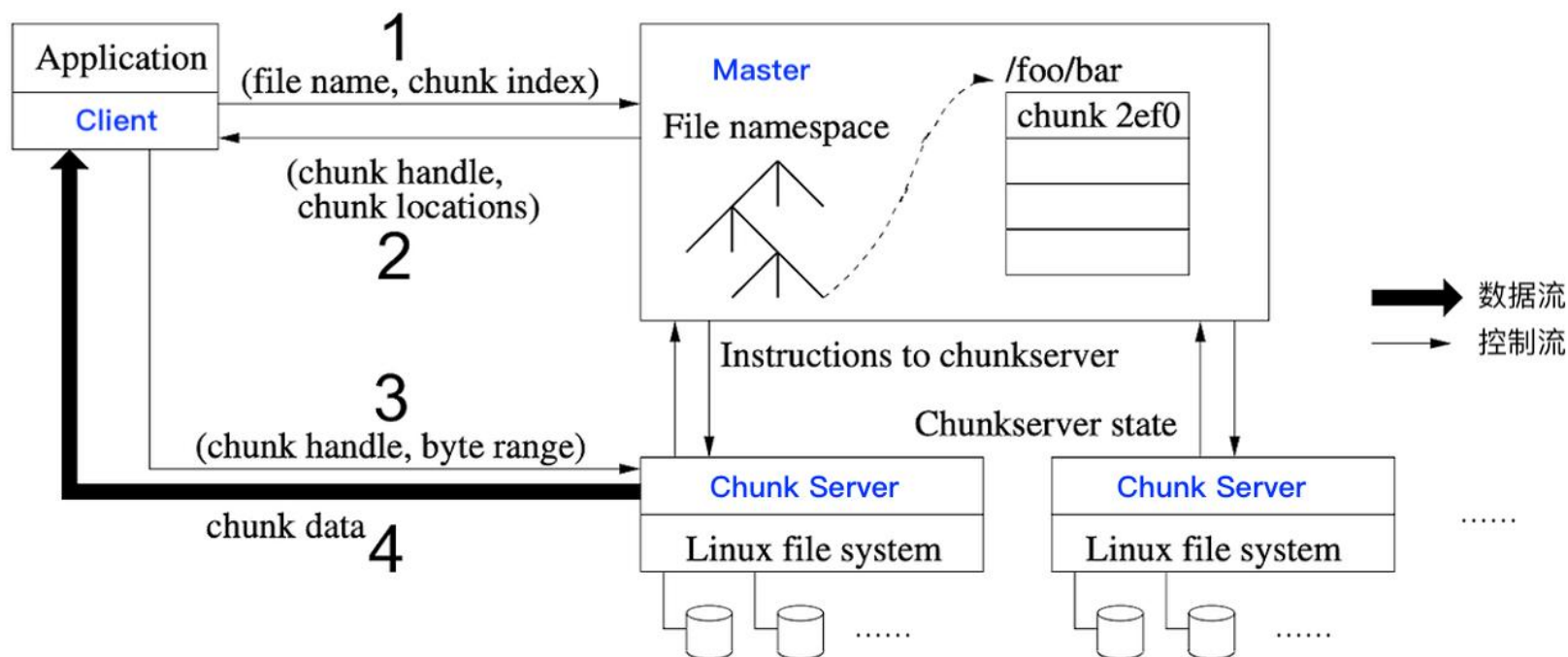
- 元数据管理
- 命名空间操作和锁定
- 块位置查找
- 负载均衡和块副本放置
- 垃圾收集
- 心跳检测和系统健康监测

3.6.2 GFS工作流程-读文件



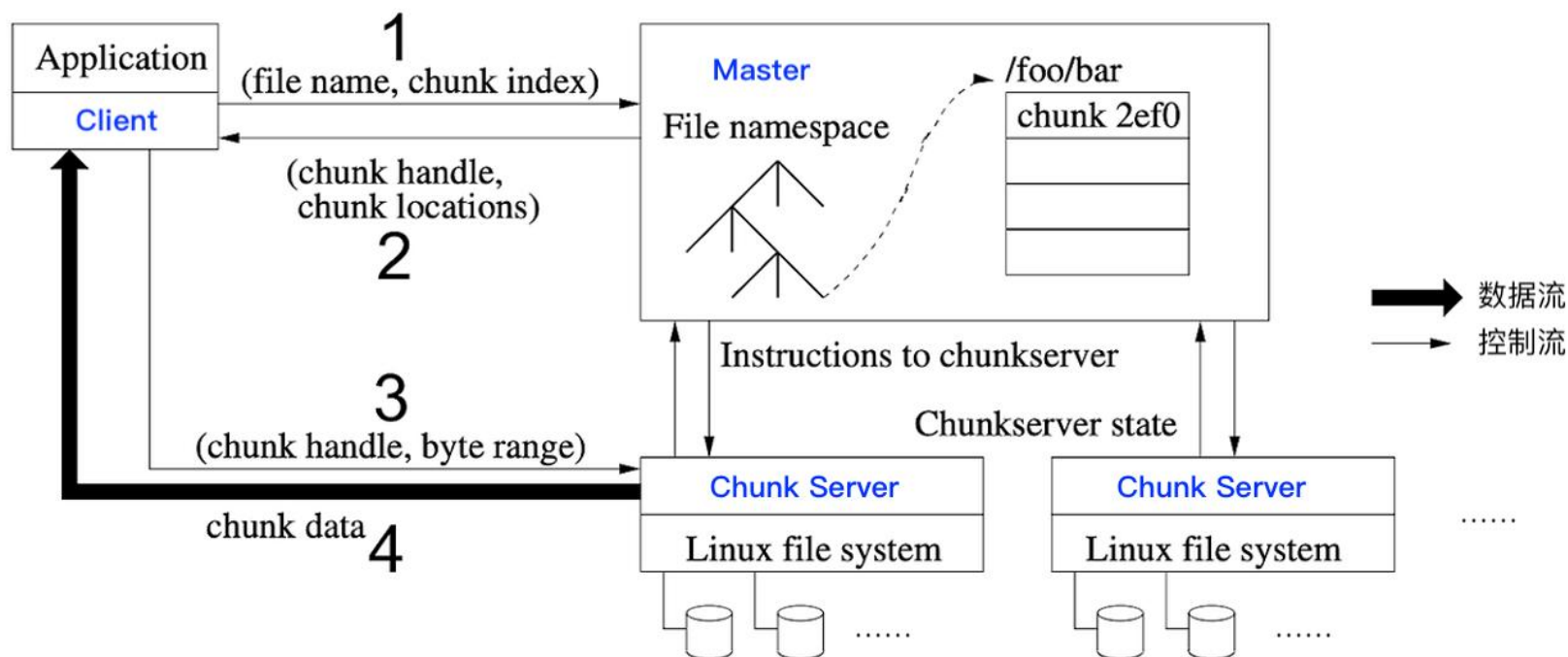
1. 程序或者GFS客户端有一个文件名和它想从文件的某个位置读取的偏移量 (offset) , 应用程序会将这些信息发送给Master节点。

3.6.2 GFS工作流程-读文件



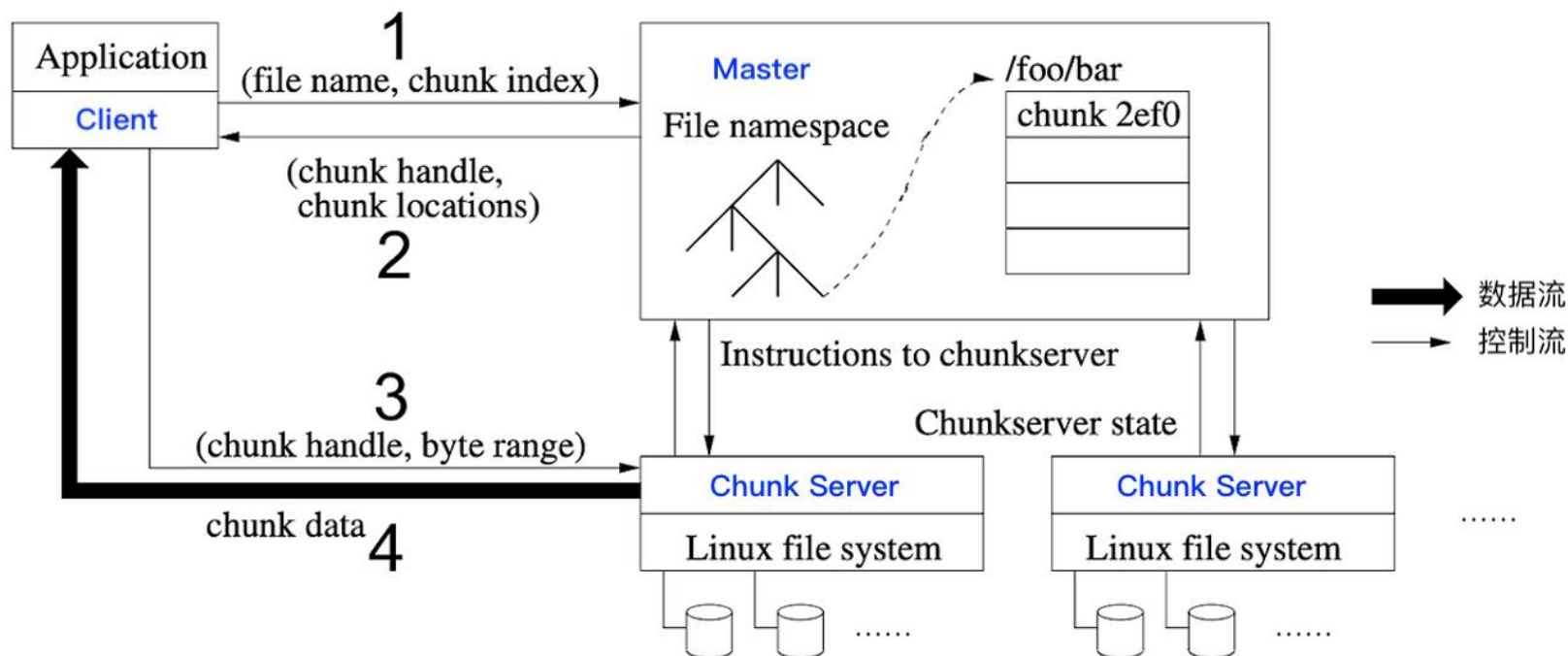
2. Master节点会从自己的file表单中查询文件名，得到Chunk ID的数组。之后Master再从Chunk表单中找到存有Chunk的服务器列表，并将列表返回给客户端。

3.6.2 GFS工作流程-读文件



3. 客户端会选择一个网络上最近的服务器，客户端会与选出的Chunk服务器通信，将Chunk Handle和偏移量发送给该Chunk服务器。

3.6.2 GFS工作流程-读文件



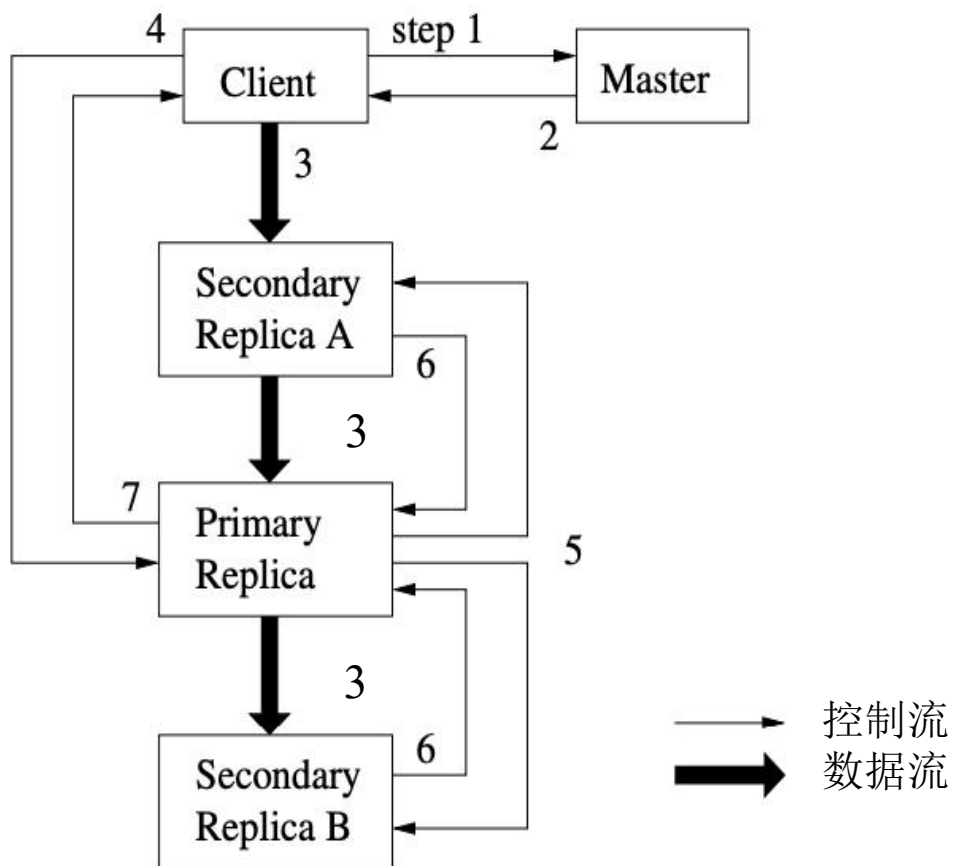
4. Chunk服务器根据文件名找到对应的Chunk文件，之后从文件中读取对应的数据段，并将数据返回给客户端。如果读取失败，客户端可能会重试其他副本。

3.6.2 GFS工作流程

租约 (Lease)

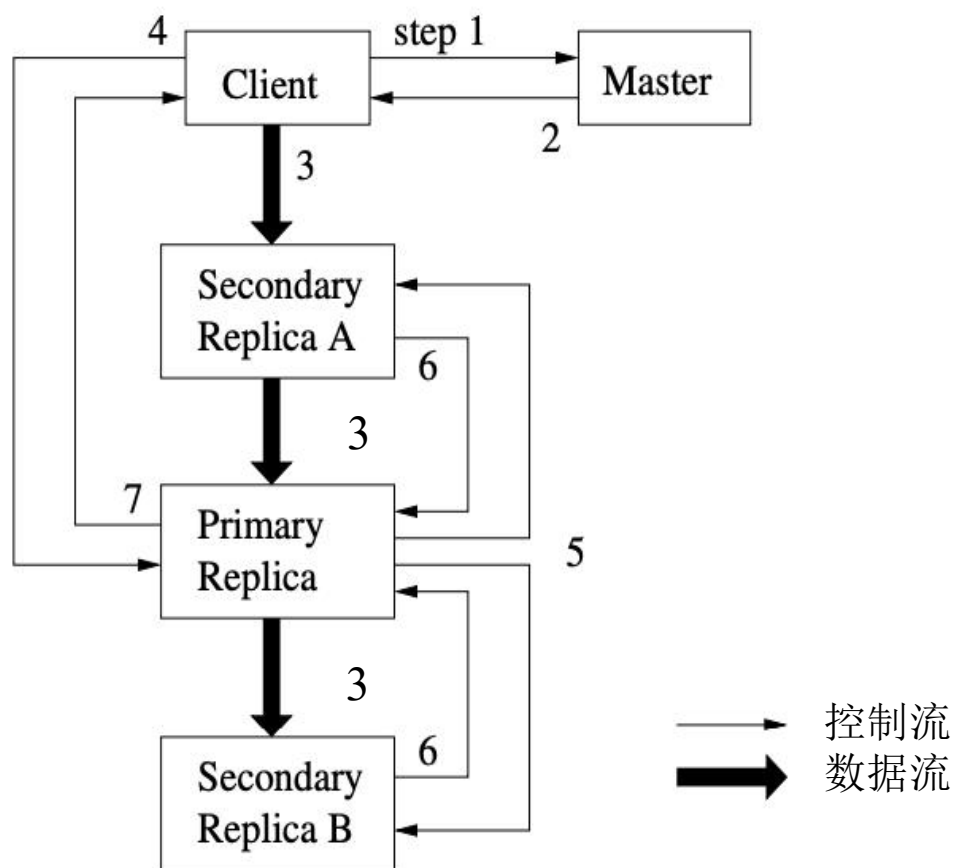
- 如果每次写操作都由master协调，那么master会成为性能瓶颈
- 既然写的内容最终落实到chunk server里的chunk上，那么，能不能把命令传给某个chunk server直接写？
- 由于多个chunk server都可能有副本，哪个chunk来负责协调？
 - 租约授予
 - 主副本，二级副本
 - 租约有效期：60s
 - 租约续期、租约过期

3.6.2 GFS工作流程-写文件



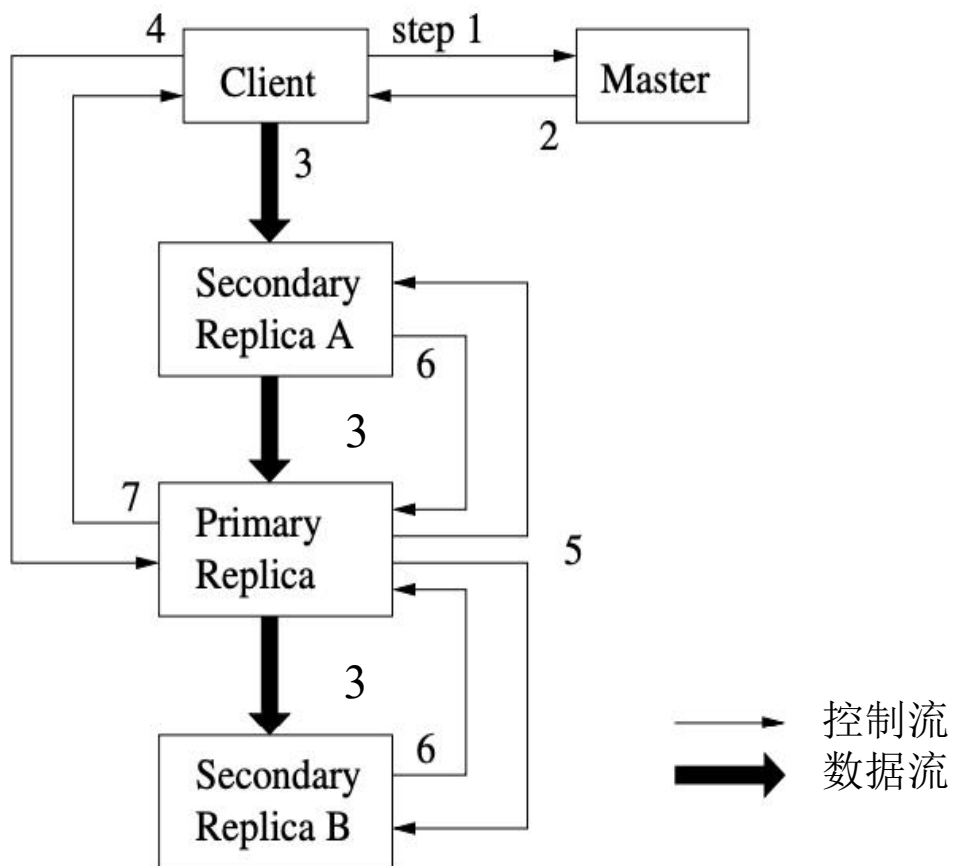
1. 客户机向Master节点询问哪一个Chunk服务器持有当前的租约，以及其它副本的位置。如果没有一个Chunk持有租约，Master节点就选择其中一个副本建立一个租约

3.6.2 GFS工作流程-写文件



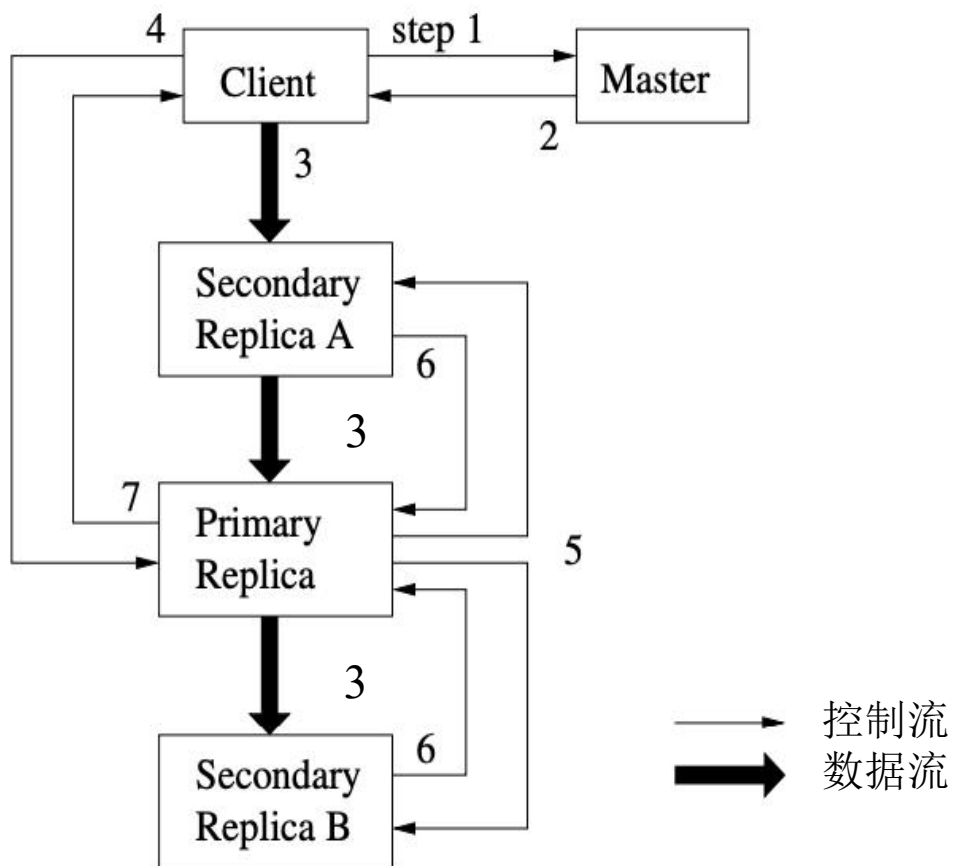
2. Master节点将主Chunk的标识符以及其它副本（又称为secondary副本、二级副本）的位置返回给客户机。客户机缓存这些数据以便后续的操作。只有在主Chunk不可用，或者主Chunk回复信息表明它已不再持有租约的时候，客户机才需要重新跟Master节点联系。

3.6.2 GFS工作流程-写文件



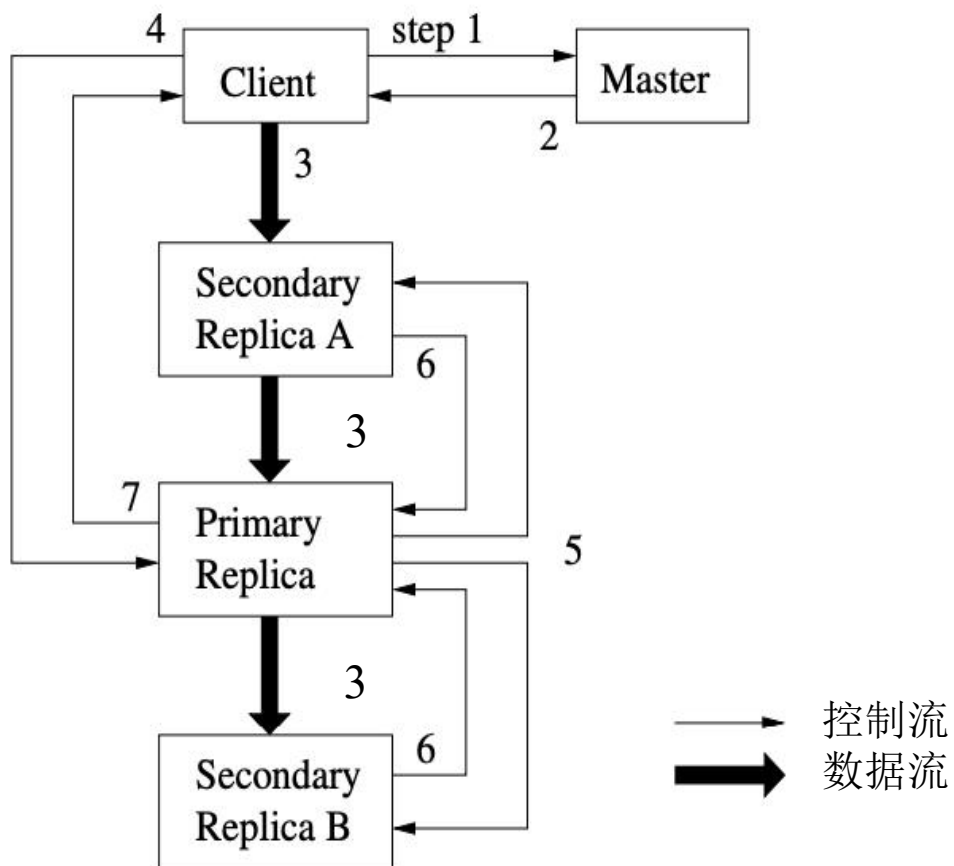
3. 客户机把数据推送到所有的副本上。
Chunk服务器接收到数据并保存在它的内部缓存中，一直到数据被使用或者过期交换出去。

3.6.2 GFS工作流程-写文件



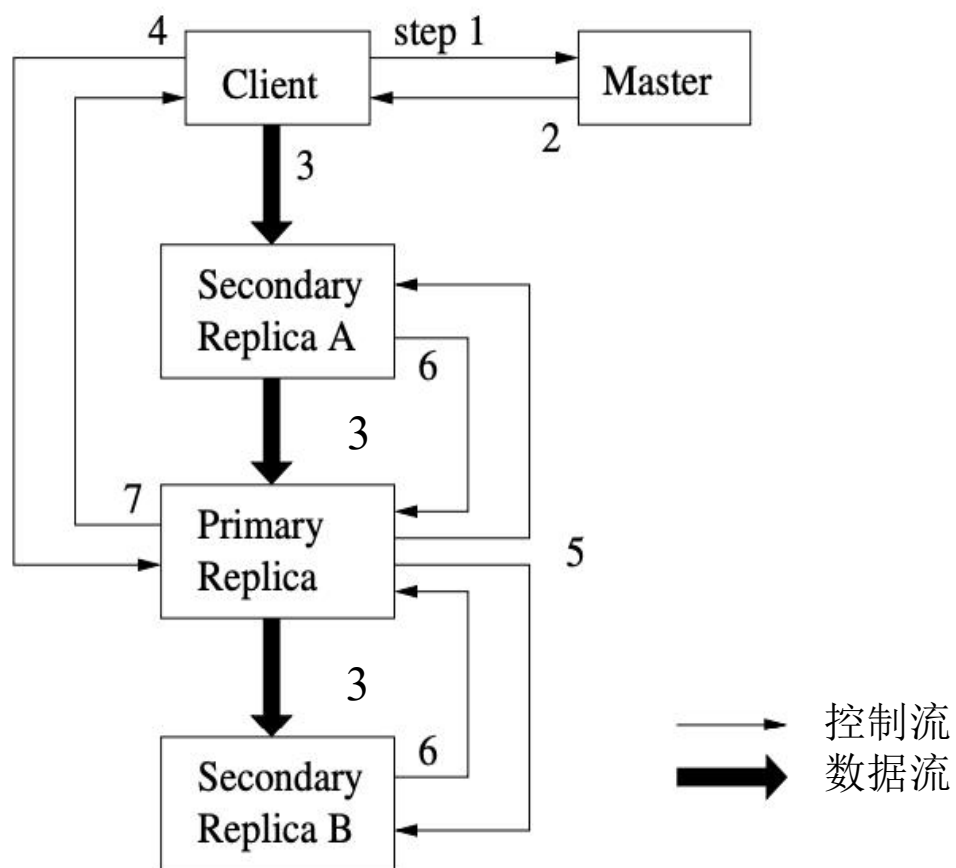
4. 当所有的副本都确认接收到了数据，客户机发送写请求到主Chunk服务器。主Chunk为接收到的所有操作分配连续的序列号，这些操作可能来自不同的客户机，序列号保证了操作顺序执行。

3.6.2 GFS工作流程-写文件



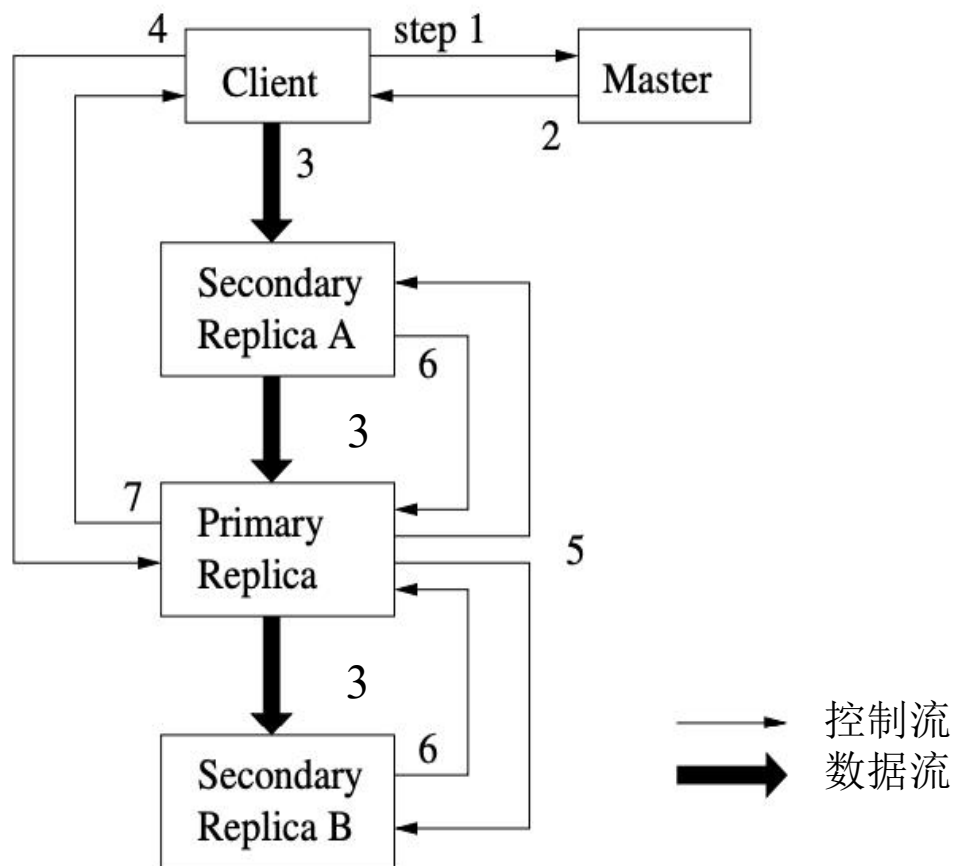
5. 主Chunk把写请求传递到所有的二级副本。每个二级副本依照主Chunk分配的序列号以相同的顺序执行这些操作。

3.6.2 GFS工作流程-写文件



6. 所有的二级副本回复主Chunk，它们已经完成了操作。

3.6.2 GFS工作流程-写文件



7. 主Chunk服务器回复客户机。任何副本产生的任何错误都会返回给客户机。在出现错误的情况下，写入操作可能在主Chunk和一些二级副本执行成功。如果主Chunk上失败了，操作就不会被分配序列号，也不会被传递。客户端的请求被确认为失败。

3.6.3 一致性模型

	write	record append
串行操作成功	defined	defined
并行操作成功	consistent but undefined (一致但非确定)	interspersed with inconsistent (有些许不一致)
失败	inconsistent (不一致)	

- GFS将写操作分为两种：
 - write：允许用户指定offset，进行随机写
 - record append：不允许指定offset，只能不断往后append写

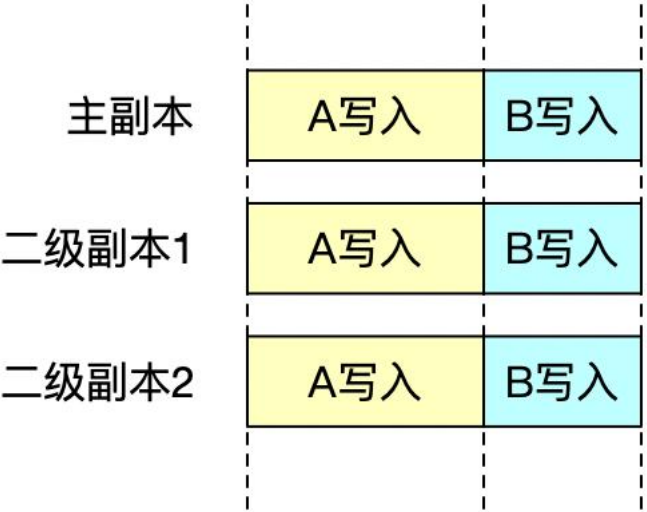
3.6.3 一致性模型

	write	record append
串行操作成功	defined	defined
并行操作成功	consistent but undefined (一致但非确定)	interspersed with inconsistent (有些许不一致)
失败	inconsistent (不一致)	

- GFS将文件状态分为两种：
 - consistent: 一致的。一个chunk的多个副本存的数据都是一致的。
 - defined: 确定的。不仅consistent，而且读到的数据是确定的，即先前写入的完整数据。
 - 一致性严格程度：defined > undefined but consistent > inconsistent

3.6.3 一致性模型

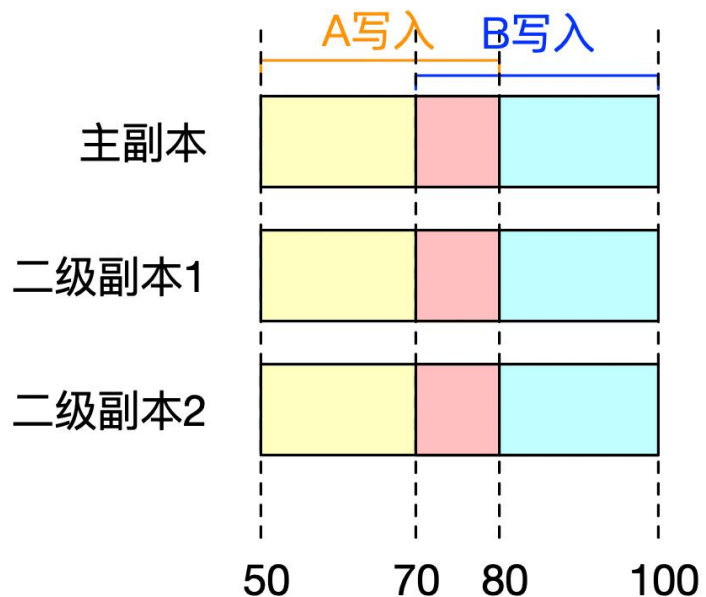
	write	record append
串行操作成功	defined	defined interspersed with inconsistent (有些许不一致)
并行操作成功	consistent but undefined (一致但非确定)	
失败	inconsistent (不一致)	



- 串行写操作成功时：
 - 多个副本之间的数据肯定彼此一致，即 consistent
 - 又由于只有一个并发，用户写入的是什么自然存放的也是什么，即 defined

3.6.3 一致性模型

	write	record append
串行操作成功	defined	defined
并行操作成功	consistent but undefined (一致但非确定)	interspersed with inconsistent (有些许不一致)
失败	inconsistent (不一致)	



- 并发写操作成功时：
 - 多个客户端并发地向同一个文件指定位置写入数据，数据有可能出现交叉。

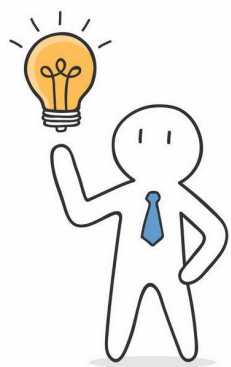
3.6.3 一致性模型



- 为什么并发写入成功时会consistent but undefined?
 - 数据的写入顺序并不需要通过Master来协调，而是直接发送给主副本的chunk server，从指定的offset处开始写
 - 随机写操作大概率会横跨多个chunk（chunk大小为64MB），随机数据的写入并非原子性的，也没有事务性保证
 - GFS允许几百个客户端读写访问，那该怎么避免这种一致但非确定的状态呢？

3.6.3 一致性模型

	write	record append
串行操作成功	defined	defined interspersed with inconsistent (有些许不一致)
并发操作成功	consistent but undefined (一致但非确定)	
失败	inconsistent (不一致)	

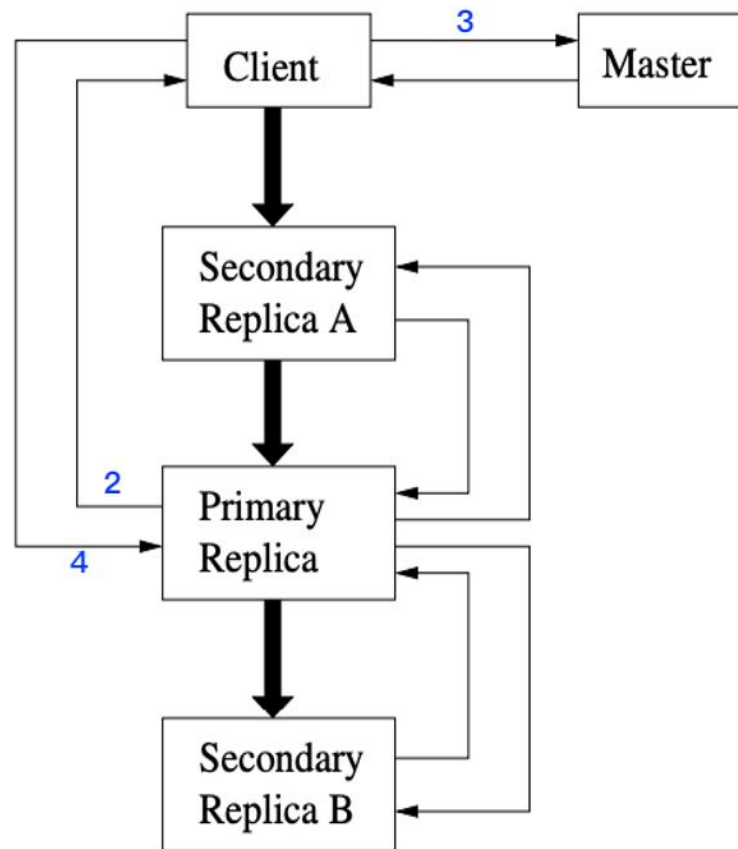


- **record append** 操作：
并不指定在chunk的什么offset位置写入
而是告诉chunk的主副本服务器 “我要追加记录”
由主副本服务器根据当前chunk位置决定写入的offset
从根本上避免了不同客户端写入内容的覆盖

3.6.3 一致性模型

■ record append具体流程

1. 主副本chunk server校验当前chunk剩余空间
 - 如果空间足够，就执行append
 - 如果空间不足，就用空数据填满（单次append大小规定为16MB）
2. 主副本chunk server告诉客户端 “在下一个chunk上写入”
3. 客户端重新从master获取新的chunk位置
4. 向新的chunk所在的主副本chunk server发起写指令，重复1-5
 - 主副本执行chunk的操作指令后，会向二级副本发送相同指令



3.6.3 一致性模型

	write	record append
串行操作成功	defined	defined
并发操作成功	consistent but undefined (一致但非确定)	interspersed with inconsistent (有些许不一致)
失败	inconsistent (不一致)	

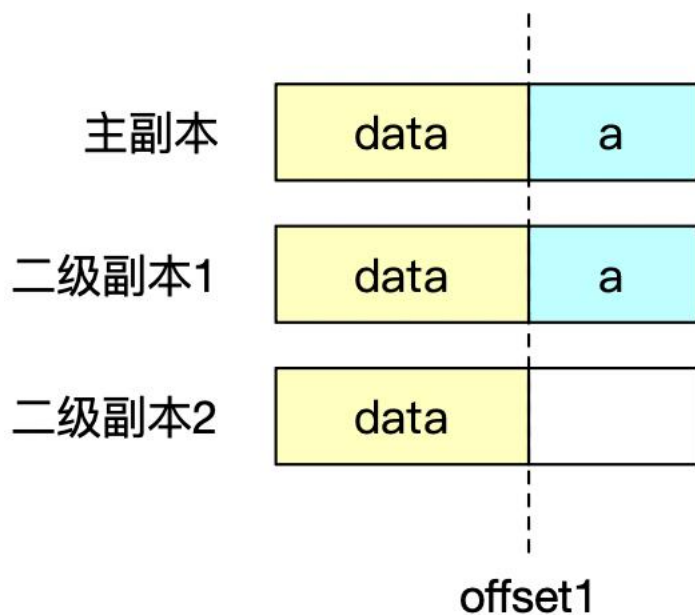
- 当某一副本写失败时，会在失败的chunk offset处填充空数据，三个副本一起重新写入一次



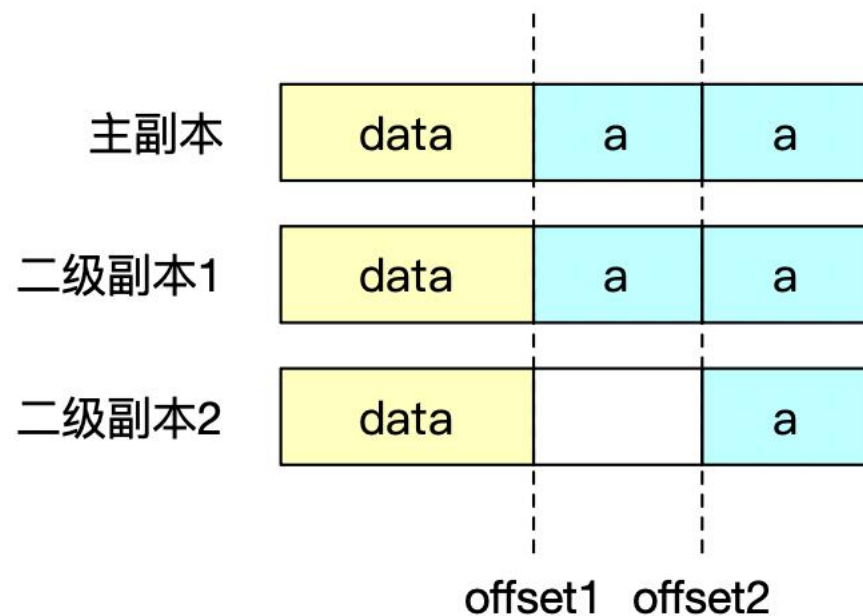
3.6.3 一致性模型

■ record append 不一致案例1

- 客户端追加了一个数据 a，第一次执行一半二级副本2失败了：



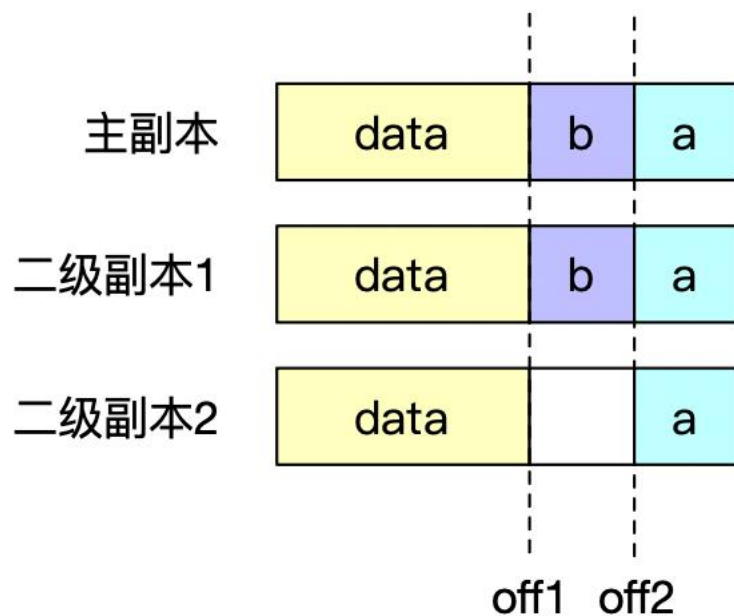
- 客户端重新发起一次追加写操作：



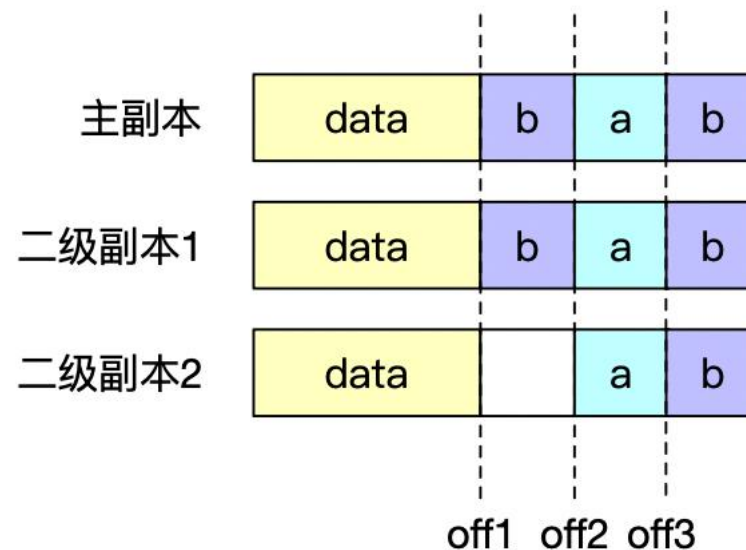
3.6.3 一致性模型

■ record append不一致案例2

- 两个客户端分别向同一个文件追加数据a和数据b，执行一半二级副本2失败了：



- 发送数据b的客户端重新发起一次追加写操作：



3.6.3 一致性模型

	write	record append
串行操作成功	defined	defined
并发操作成功	consistent but undefined (一致但非确定)	interspersed with inconsistent (有些许不一致)
失败	inconsistent (不一致)	

- 不同的副本中可能被追加了不同的次数，但“至少一次”是defined的，这就是At Least Once，GFS承诺的一致性。
- 可见GFS对写入数据的一致性保障相当低，它只是保障了所有数据追加至少被写入一次

小节：数据划分与定位

- 数据划分的意义
- 静态数据划分方法
 - 范围划分，哈希划分，热点困境
- 动态数据分区调整
 - 限定分区数量，限定分区大小，限定分区比例
- 数据定位
 - 元信息存储在分布式节点、第三方、客户端
- 一致性哈希
- GFS
 - 架构，工作流程，一致性模型

第三章 完