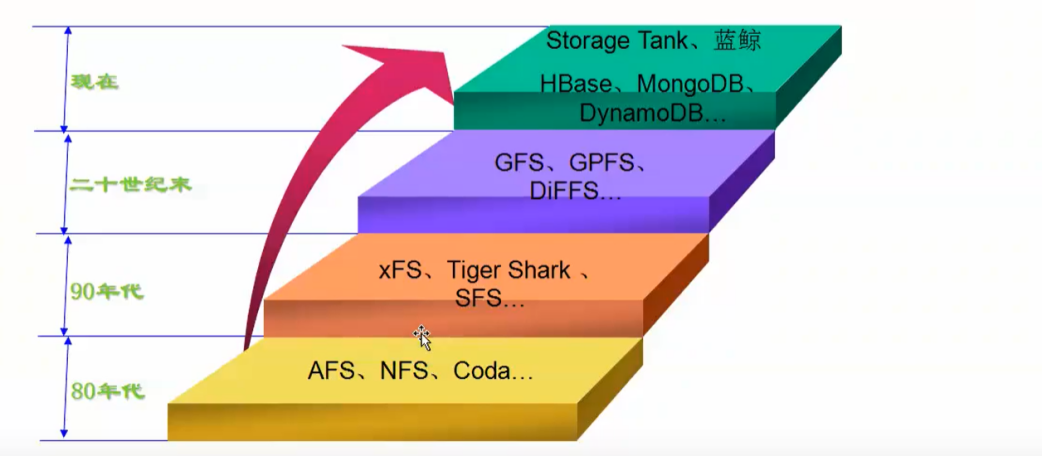
# 概述

## 概念

大量普通服务器，通过网络互连，对外作为一个整体提供存储服务。

## 发展历程

分布式存储系统发展：



### 80年代

#### AFS

1、1983年CMU和IBM合作开发Andrew文件系统（Andrew File System，AFS）

1. AFS设计的目标是将至少7000个工作站连接起来，为每个用户提供一个共享的文件系统
2. 高扩展性、网络安全性放在首位
3. 客户端高速缓存，即使网络断开，可以对部分数据缓存

#### NFS

1、1985年Sun公司基于UDP开发了网络共享文件系统（Network File System，NFS）

1. 由一系列NFS命令和进程组成的客户机/服务器（C/S）模式
2. NFS第三版，加入了基于TCP传输
3. 第三版发布六年后，NFS成为Linux中的稳定版本

#### Coda

1、1987年CMU在基于AFS的基础上开发了Coda文件系统

1. 它为Linux工作站组成的大规模分布式计算环境设计的文件系统
2. 通过两种互补机制为服务器和网络故障提供了容错机制
3. 服务器复制机制，一个文件拷贝到多个服务器上
4. 无连接操作机制，将缓存端暂时作为服务端的执行模式
5. Coda注重可靠性和性能优化
6. 它提供了高度的一致性

### 90年代

背景：

1. 进入九十年代，随着Windows的问世，极大促进了微处理器的发展和PC的广泛普及
2. 互联网和多媒体技术也发展起来
3. 一方面：对多媒体数据的实时传输需求和应用越来越流行
4. 另一方面：大规模并行计算技术的发展和数据挖掘技术应用，迫切需要能支持大容量和高速的分布式存储系统

#### xFS

1. UC Berkeley参照当时高性能多处理器领域的设计实现开发了xFS文件系统
2. xFS克服了以往分布式文件系统只适用于局域网，而不适用于广域网和大数据存储问题
3. 提出广域网进行缓存较少网络流量设计思想
4. 采用层次命名机构，减少cache一致性状态和无效写回cache一致性协议，从而减少了网络负载

### 20世纪末

**背景：**

1. 到了二十世纪末，计算机技术和网络技术得到飞速发展，磁盘存储成本不断降低
2. 磁盘容量和数据总线带宽的增长速度无法满足应用需求
3. 海量数据的存储逐渐成为互联网技术发展急需解决的问题
4. 对于分布式存储系统技术的研究越来越成熟
5. 基于光纤通道的存储区域网络（Storage Area Network）技术和网络附加存储（Network Attached Storage）技术得到了广泛应用

技术：SAN、NAS、GFS、HDFS、GPFS

#### SAN

1. 设计目标是通过将磁盘存储系统或磁带机和服务器直接相连的方式提供一个易扩展、高可靠的存储环境
2. 高可靠的光纤通道交换机和光纤通道网络协议保证各个设备间链接的可靠性和高效性
3. 设备间的连接接口主要是采用FC或者SCSI
4. 光纤通道交换机主要是为服务器和存储设备的链接提供一个成为SAN Fabric的网状拓扑结构

#### NAS

1. 通过基于TCP/IP协议的各种上层应用（NFS等）在各个工作站和服务器之间进行文件访问
2. 直接在工作站客户端和NAS文件共享设备之间建立连接
3. NAS隐藏了文件系统的底层实现，注重上层的文件服务实现，具有良好的扩展性
4. 网络阻塞，NAS性能受影响

#### GFS

1. Google为大规模分布式数据密集型应用设计的可扩展的分布式文件系统
2. Google将一万台廉价PC机器连接成一个大规模的Linux集群
3. 它具有高性能、高可用、易扩展性、超大存储容量等优点
4. Google文件系统采用单Master多ChunkServer来实现系统间的交互
5. Master中主要保存命名空间到文件映射、文件到文件块的映射、文件块到ChunkServer的映射
6. 每个文件块对应到3个ChunkServer

### 现在

产品：HBase、Cassendra、MongoDB、DynamoDB

#### HBase

1. 列存储数据库
2. 擅长以列为单位读取数据
3. 面向列存储的数据库具有高扩展性，即使数据大量增加也不会降低相应的处理速度，特别是写入速度

#### MongoDB

1. 文档型数据库它同键值（Key-Value）型的数据库类似
2. 键值型数据库的升级版，允许嵌套键值，value值是结构化数据
3. 数据库可以理解value的内容，提供复杂的查询，类似于RDBMS的查询条件

#### DynamoDB

1. Amazong公司的一个分布式存储引擎
2. 是一个经典的分布式key-value存储系统，具备去中心化，高可用，高扩展性的特点
3. 达到这个目标在很多场景中牺牲了一致性
4. Dynamo在Amazon中得到成功的应用，能够跨数据中心部署于上万个节点上提供服务，它的设计思想也被后续的许多分布式系统借鉴

## 特征

### 可扩展

高扩展性：

1. 指分布式存储系统通过扩展集群服务器规模从而提高系统存储容量、计算和性能的能力
2. 业务量增大，对底层分布式存储系统的性能要求越来越高，自动增加服务器来提升服务能力
3. Scale Up与Scale Out
4. 线性可扩展：系统整体性能与服务器数量呈线性关系

### 可用

高可用：

1. 指分布式存储系统在面对各种异常时可以提供正常服务的能力
2. 系统的可用性可以用系统停服务的时间和正常服务时间的比例来衡量
3. 4个99的可用性（99.99%）

一年停机的时间不能超过53分钟（365\*24\*60/10000=53分钟）

### 可靠

高可靠：

1. 重点只分布式系统数据安全方面的额指标
2. 数据可靠不丢失
3. 多机冗余、单机磁盘RAID等

### 高性能

高性能：

1. 衡量分布式存储系统性能常见的指标是系统的吞吐量和系统的响应延迟
2. 系统的吞吐量

是在一段时间内可以处理的请求总数，QPS（Query Per Second）和TPS（Transaction Per Second）

1. 系统的响应时间

是指某个请求发出到接收到返回结果所消耗的时间，通常用平均延迟来衡量

4、衡量分布式存储系统常见的指标是系统的吞吐量和系统的响应延迟

这两个指标是矛盾的：追求高吞吐量，比较难做到低延迟；追求低延迟，吞吐量会受影响

### 稳定性

高稳定性：

1. 这是一个综合指标
2. 考核分布式存储系统的整体健壮性
3. 任何异常，系统都能够坦然面对
4. 系统稳定性越高越好

### 一致性

数据一致性：

1. 分布式存储系统多个副本之间的数据一致性
2. 弱一致性
3. 强一致性
4. 最终一致性
5. 因果一致性
6. 顺序一致性

### 安全性

高安全性：

1. 指分布式存储系统不受恶意访问和攻击，保护存储数据不被窃取
2. 互联网是开放的
3. 任何人在任何时间任何地点通过任何方式都可以访问网站
4. 针对现存和潜在的各种攻击与窃取手段，都有相应的应对方案

### 易于维护

### 低成本

## 分类

### 数据类型

#### 非结构化数据

1. 指其字段长度不等，并且每个字段的记录又可以由可重复或不可重复的子字段构成、
2. 没有规律
3. 文本、图像、声音、影视等等

#### 半结构化数据

1. 介于完全结构化数据（如关系型数据库、面向对象数据库中的数据）和完全无结构的数据（如声音、图像文件等）之间的数据
2. HTML文档就属于半结构化数据，它一般是自描述的，数据的结构和内容混在一起，没有明显的区分

#### 结构化数据

1. 结构化数据即行数据，存储在数据库里，可以用二维表结构来逻辑表达实现的数据
2. 数据模式和内容是分开的
3. 数据的模式需要预先定义

### 分布式存储类型

针对非结构化数据、半结构化数据、结构化数据等不同数据类型，有对应的不同存储系统。

#### 分布式文件系统

1. 存储大量的文件、图片、音频、视频等非结构化数据
2. 这些数据以对象的形式组织，对象之间没有关系
3. 这些数据都是二进制数据
4. GFS、HDFS

#### 分布式Key-Value系统

1. 提供存储关系简单的半结构化数据
2. 提供基于key的增删改查操作
3. 缓存、固化存储
4. Redis、Memcached、DynamoDB等

#### 分布式数据库

1. 存储结构化数据，提供SQL关系查询语句，支持多表关联，嵌套子查询等
2. MySQL Sharding集群、MongoDB等等

### 数据模型分类

#### 文件

目录树的方式组织文件

#### 文件型

MySQL、Oracle

#### 键值存储型

Redis、Memcached、Tokyo Cabinet

#### 列存储型

HBase、Cassadra

#### 文档型

MongoDB、CouchDB

#### 图形数据库

Neo4J、InfoGrid、Infinite Graph

# 原理与设计

## 单机存储

### 硬件基础

### 存储引擎

存储引擎是存储系统的发动机，决定了存储系统能够提供的功能和性能。存储系统提供功能：CRUD。

存储引擎分类：B树，哈希，LSM存储引擎。

#### B树存储引擎

特点：

1、基于B树实现

2、支持单条记录的CRUD

3、支持顺序扫描，范围查找

4、RDBMS使用较多：MySQL、InnoDB聚簇索引、B+树

#### 哈希存储引擎

1. 基于哈希表结构的键值存储系统
2. 常用实现方式：数组+链表（hashcode存储在数据中，相同的放在链表中）
3. 支持create、update、delete、随机read（不支持顺序和范围的读）
4. 读时间复杂度：O(1)

#### LSM存储引擎

LSM（Log Structured Merge Tree）特点：

1. 对数据的修改增量保存在内存中，达到指定条件后（通常是数量和时间间隔），批量将更新操作持久化到磁盘
2. 读取数据时需要合并磁盘中的历史数据和内存中最近修改操作
3. LSM优势在于通过批量写入，规避了随机写入问题，提高了写入性能
4. LSM劣势在于读取需要合并磁盘数据和内存数据

注：因为内存中最新的写操作还没有回写到磁盘，所以读的时候需要将磁盘的数据和内存中的数据进行整合，这个就比较消耗时间了。

1. 如何避免内存数据丢失？

借助commit log，首先将修改操作写入到commit log中，即使机器重启内存中数据丢失，但是仍然可以通过commit log从磁盘恢复，保证数据的可靠性。

**典型案例设计：LevelDB**

注：分布式数据库Google Spannner、OceanBase、TiDB都采用这种索引。

### 数据模型

数据模型是存储系统的外壳。

数据模型分类：

#### 文件

以目录树的方式组织文件，比如Linux、Windows、Mac

#### 关系

每个关系是一个表格，由多个行组成，每个行由多个列组成，比如MySQL、Oracle

#### V

比如Redis、Memcached

#### 列存储

比如HBASE、Cassadra

#### 图形

比如Neo4J、InfoGrid、Infinite Graph

#### 文档

比如MongoDB、CouchDB

### 事务及并发控制

事务的四个基本属性：原子性、一致性、隔离性、持久性

并发控制：

1. 锁粒度

发展历程：Process（进程）->DB->Table->Row

1. 提供Read并发，Read不加锁：
2. 写时复制
3. MVCC

### 故障恢复

一般通过操作日志实现数据恢复。

### 数据压缩

## 多机存储

单机存储与多机存储：

1. 单机存储的原理在多机存储任然可用
2. 多机存储基于单机存储

多机数据分布区别于单机存储，数据分布在多个节点上，在多个节点之间需要实现负载均衡。

### 数据分布方式

1. 静态方式
2. 取模
3. uid%32
4. 动态方式
5. 一致性hash（机器宕机后，可以自动调整）
6. 数据漂移问题（由于网络抖动，对节点1执行update操作的时候，致节点1出现网络闪断，数据自动转移到节点2，此时在节点2进行update操作，而后节点1很快就网络恢复，则此时产生脏数据）

### 负载均衡

方式：

1. 多节点之间数据的均衡
2. 负载高的节点迁移到负载低的节点：比如MongoDB采用Auto-Sharding

### 复制

分布式存储多个副本，保证可靠性和可用性。主要是借助commit log实现。

### 故障检测

主要是通过心跳机制实现，如果有问题，则需要做数据迁移和数据恢复。

## FLP原理

FLP Impossibility（FLP不可能性）是分布式领域中一个非常著名的结论，1985年Fischer、Lynch and Patterson三位作者发表论文，并获得Dijkstra奖。

在异步消息通信场景，即使只有一个进程失败，没有任何方法能够保证非丢失进程达到一致性。

FLP系统模型基于以下几个假设：

1. 异步通信

与同步通信最大区别是没有始终、不能时间同步、不能使用超时、不能探测失败、消息可任意延迟、消息可乱序。

1. 通信健壮性

只要进程非失败，消息虽会被无限延迟，但最终会被送达，并且消息仅会被送达一次（重复保证）。

1. Fail-Stop模型

进程失败不再处理任何消息。

1. 失败进程数量

最多一个进程失败

**FLP带给我们的启示：**

1、1985年FLP证明了异步通信中不存在任何一致性的分布式算法（FLP Impossibility）

1. 人们开始寻找分布式系统设计的各种因素（因此出现了CAP定理）  
    3、一致性算法既然不存在，如果能够找到一些设计因素，适当取舍以最大限度满足实现系统需求成为当时的重要议题
2. CAP理论出现

## CAP定理

2000年Berkerly的Eric Brewer教授提出了著名的CAP理论：一致性（Consistency）、可用性（Availability）、分区容错性（Tolerance of network Partition），在分布式环境下，三者不可能同时满足。

**一致性：**

1. Read的数据总是最新的（Write的结果）
2. 分为弱一致性和强一致性

**可用性：**机器或者系统部分发生故障，仍然能够正常提供读写服务

**分区容错性：**机器故障、网络故障、机房故障等异常情况下任然能够满足一致性和可用性。

分布式存储系统需要能够自动容错，也就是说分区容忍性需要保证。

**保证一致性：**

强同步复制，主副本网络异常，写操作被阻塞，可用性无法保证。

**保证可用性：**

异步复制机制，保证了分布式存储系统的可用性，强一致性无法保证。

一致性和可用性需要折中权衡：

1. 不允许数据丢失（强一致性）：金融领域
2. 小概率丢失允许（可用性）：消息

## BASE理论

## 2PC协议

2PC（Two Phase Commit）实现分布式事务。

包括两类节点：协调者（1个），事务参与者（多个）。

每个节点都会记录commit log，保证数据可靠性。

**两节点提交由两个阶段组成：**

1. 请求阶段（Prepare Phase）

协调者通知参与者准备提交或者取消事务，之后进入表决阶段，每个参与者将告知协调者自己的决策。

1. 提交阶段（Commit Phase）
2. 收到参与者的所有决策后，进行决策：

提交

取消

1. 通知参与者执行操作：

所有参与者都同意，提交

否则取消

1. 参与者收到协调者的通知后执行操作

**2PC协议是阻塞式：**

1. 事务参与者可能发生故障

设置超时时间

1. 协调者可能发生故障
2. 日志记录
3. 备用协调者

应用：分布式事务

## Paxos协议

Paxos协议用于解决多个节点之间的一致性问题。

只有一个主节点，主节点挂掉，通过选举产生新的节点，主节点往往以操作日志的形式同步备节点。

主要包括两种角色：提议者（Proposer）和接受者（Acceptor）。

执行步骤：

1. 批准

Proposer发送accept消息到Acceptor要求接受某个提议者

1. 确认

如果超过一半的Acceptor接受，意味着提议者生效，Proposer发送acknowledge消息通知所有的Acceptor提议生效

Paxos协议用法：

1. 实现全局的锁服务或者命名和配置服务：Apache Zookeeper
2. 将用户数据复制到多个数据中心：Google Megastore

2PC与Paxos：

1. 作用不同：

2PC协议保证多个数据分片上操作的原子性

Paxos协议保证一个数据分片多个副本之间数据的一致性

1. 两者结合使用：

2PC最大缺陷无法处理协调者宕机，Paxos可以处理协调者宕机

注：OceanBase采用的就是2PC+Paxos方案控制分布式事务。

# 分析对比



## GFS

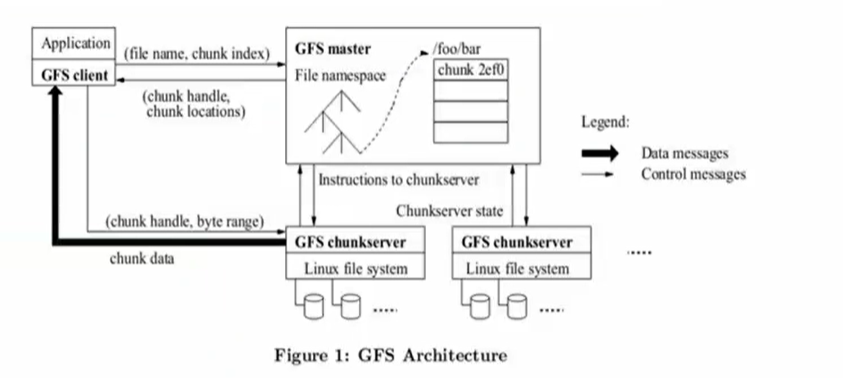
### 简介

GFS（Google File System）为存储海量搜索数据设计的文件系统（最初是为了存储Google的网页数据），是可扩展的分布式文件系统。运行于廉价的普通硬件，提供容错功能，提供总体性能较高的服务。

基于假设：

1. 基于普通PC，单机不可靠，集群能检测和恢复
2. 存储较多的文件
3. 大量的顺序读，少量的随机读
4. 直接append
5. 带宽要求高

### 架构设计



GFS master：

元数据服务器节点，只有一个。

GFS chunkserver：

数据存储节点，多个。

GFS client：

客户端节点，多个。

#### GFS master

GFS的元数据服务器节点，负责维护GFS的元数据，作用主要包括：

1. 命名空间
2. 访问控制
3. 文件和块映射
4. 块存储节点
5. 垃圾回收
6. 负载均衡
7. 心跳：获取chunkserver的心跳，处理

#### GFS chunkserver

主要特点：

1. 大文件分为多个块，每个块有ID
2. 每块存储在chunkserver上
3. 块大小设置64M：主要是考虑元数据大小影响交互次数
4. 可靠性：采用3个副本，分布在不同的chunkserver上

#### GFS client

应用方嵌入client代码，client作为代理与GFS master和GFS 春款server交互。

### 原理

读取数据流程：

1. client指定读取某个文件的某段数据

因为数据块是定长，client计算包含几个数据块，client将file name,chunk index发送给master

1. Master根据file name查找命名空间和文件->块映射表，得到需要的数据块副本地址，master将chunk handler，chunk locations所有副本的抵制反馈给client
2. Client选择一个副本，和chunkserver交互，获取数据
3. Chunkserver获取数据返回给client

### 应用

使用场景：

1. 海量大文件
2. 海量读
3. 不修改文件
4. 高并发
5. 网页数据
6. 视频（非直播）
7. 日志

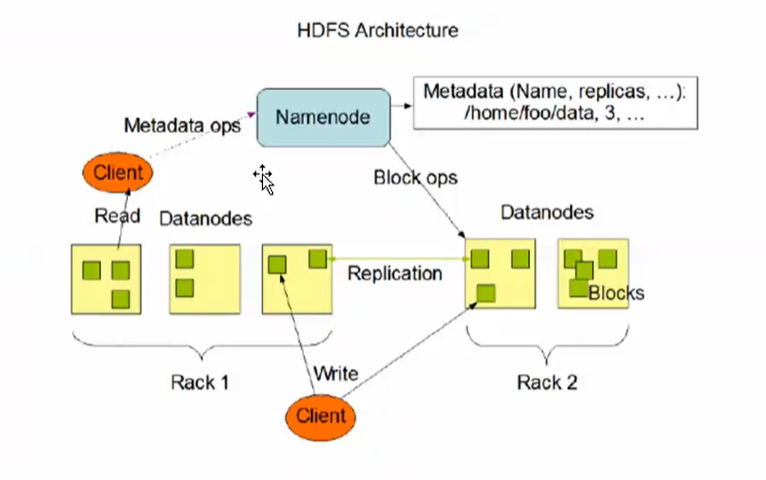
## HDFS

### 简介

HDFS（Hadoop File System）是GFS的一个开源实现版本，按照GFS设计实现。

注：Hadoop包括HDFS（存储）和MapReduce（计算）。

### 架构设计



Namenode：

元数据服务器节点，1个。

Datanodes：

数据存储服务器节点，多个。

Client：

客户端节点，多个。

#### Namenode

#### Datanodes

#### Client

数据预取和缓存。

#### JAVA

开发。

### 应用

1. 超大文件
2. 基于流式数据访问：一次写入，多次读取
3. 高吞吐量：以数据延迟为代价

## MongoDB

### 简介

MongoDB是面向文档的NoSQL数据库，取名来自“humongous”，是开源的数据库，采用C++编写。

具有可拓展课高性能的特征：

1. 可扩展（scalable）：主从方式、副本
2. 高性能：所有操作都在内存中

关键特性：

1. Document-Oriented Storage
2. Full Index Support
3. Replication & High Availability
4. Auto-Sharding
5. Rich Quering
6. Updates
7. Map/Reduce
8. GridFS

### 架构

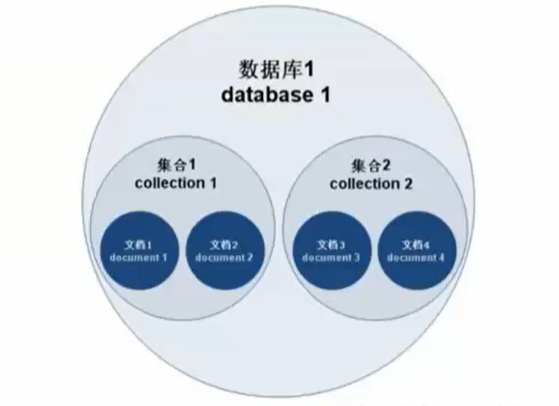
#### 逻辑关系图

文档（document）、集合（collection）、数据库（database）

MongoDB与RDBMS数据逻辑结构对比：



#### 层次关系图



#### 系统结构

**数据存储**

1. 默认存储路径：

/data/db（数据）

/data/log（日志）

1. 数据库数据组成：

数据库名.ns

数据库名.0 数据库名.1......数据库名.n

MongoDB生态系统

MongoDB集群组成：

1. 数据服务节点（mongod）
2. 路由节点（mongos）
3. 配置节点（mongod）->config server
4. 投票节点/表决节点（mongod）->arbiter

### 应用

适用场景：

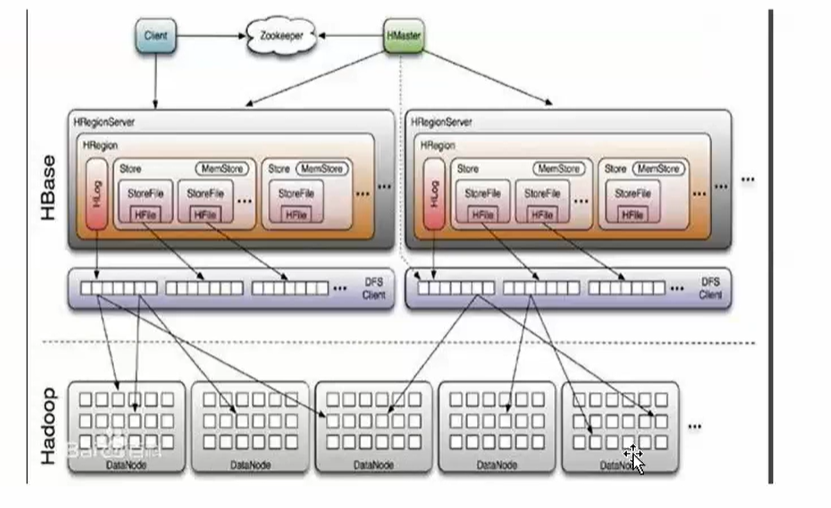
1. Web应用程序
2. 敏捷开发
3. 分析和日志
4. 缓存
5. 可变schema

## HBase

### 简介

HBase（Hadoop Database）是Hadoop数据库，面向列的开源数据库，基于HDFS分布式文件存储。

### 架构



#### Hclient

基于HBase的RPC机制与HMaster和HRegionServer通信。

管理类操作：HClient与HMasterRPC

数据读写类操作：HClient与HRegionServer RPC

#### HMatser

没有单点，可以启动多个HMaster，基于Zookeeper的Master Election机制保证一个Master运行。

作用包括：

1. 负责Table和Region的管理工作
2. 管理用户对Table的CURD
3. 管理HRegionServer的负载均衡，调整Region分布

在Region分裂后，负责新的Region分配

在HRegionServer停机后，负责此机器上Regions的数据迁移

#### HReginServer

HRegionServer是HBase最核心的模块，作用包括：

1. 负责响应用户I/O请求
2. 想HDFS文件系统中读写数据
3. 管理一系列HRegion对象

每个HRegion对应了Table中的一个Region

HRegion中有多个HStore组成

每个HStore对应了Table中的一个Column Family存储

Column Family是一个集中的存储单元

具备I/O特性的Column放在一个Column Family中最高效

#### Zookeeper

1. 存储ROOT表地址
2. 存储HMaster地址
3. HRegionServer会注册到Zookeeper中
4. HMaster可以随时感知各个HRegionServer的健康状态
5. Zk避免了HMaster的单点故障

### 应用

适用场景：

1. 半结构化或者结构化数据：数据结构字段不够确定
2. 记录非常稀疏：

RDBMS列固定

NULL列浪费

NULL Column不会被存储

节省空间

提高读性能

1. 多版本数据
2. 超大数据量：分库分表、HBase自动水平切分

# 一致性

## 简介

**什么是一致性？**

1. 通常指关联数据是否完整和正确
2. 多个数据中心数据一致
3. 每个访问用户看到的数据具有一致性

**什么是原子性？**

1. 一个事务中的所有操作，要么全部完成，要么全部不完成
2. 不会结束在中间的某个状态
3. 事务在执行过程中发生错误，会被回滚到事务开始前的状态

**原子性和一致性的区别及联系：**

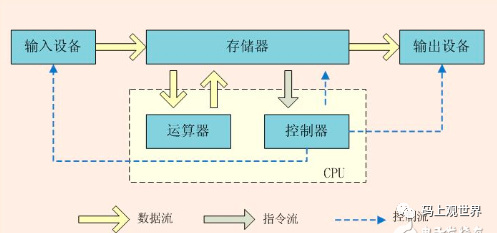
1. 保证原子性，一致性必要满足
2. 一致性保证，不要求必须满足原子性
3. 充分非必要条件
4. 2PC：原子性问题
5. Paxos：一致性问题

## 一致性模型

在计算机的物理世界中，每个系统的进程都是有距离的，比如一个没有在CPU本地缓存的值距离内存条的距离通常为30cm，不考虑CPU装载值等操作开销，仅仅按照信号传输来算，也需要1纳秒（10的负9次方秒）的时间（光信号速度为30万公里每秒）。

如果把本地取值扩展到跨数千公里范围外的计算机，这个时间达到数百毫秒。这就是说计算机物理世界的操作都是耗时的或者说操作是有延时的，因此可以说，客观世界的约束性才是导致上层应用复杂性的根源。而产生这种约束性的直接原因在于1945年美籍匈牙利人冯-诺依曼提出的经典冯-诺依曼计算机存储结构：该结构将指令数据和存储数据公用一个存储器，在控制器的作用下，运算器从存储器中加载指令和数据完成计算之后数据写回存储器，运算器和控制器组成中央处理器（CPU）。

如果输入和输出数据不在存储器中，还需要从外部输入设备读取数据和写出到外部输出设备，输入设备和输出设备组成I/O设备，这样一个微型的计算机结构就成型了，如下图所示：



然而这种结构有个很大的弊端：CPU的处理频率远高于存储器，另外运算器的容量也远小于主存储器，因此大量的数据存取和运算就导致了系统瓶颈。为了解决存储器和CPU运算速度不匹配的问题，其中一种解决办法是利用数据的局部性原理，引入CPU高速缓存（L1 cache），在此基础上，还有二级甚至三级缓存。于是导致了新的问题：当存储器中的数据更新，如果无法及时通知缓存失效，就会导致存储器的数据跟缓存数据不一致。

可以说正是因为这种存储和计算分离的结构问题导致了数据不一致问题。这个是一致性问题产生的微观原因，如果将这种结构逐步向上层推进，会发现计算机领域的很多系统设计也都存在这种问题，比如线程中的局部存储和主存中的数据，数据库或者其他外存设备的数据与内存中缓存的数据，分布式系统中的计算与存储分离架构如AWS的计算集群和外部存储系统S3都会存在数据不一致问题，而且随着层级的上升，不一致问题越来越明显，为解决一致性问题引入的方案越来越复杂。

比如在多线程并发写入和读取一个未加同步限制的共享变量，由于每个线程调度顺序的不确定性，导致对公共变量的读取并不能像在单线程中那么“确定“，对于调用方来说，这个读取是不符合预期，也可以说是不准确的。那如何让系统给出符合预期的准确结果呢？在介绍如何解决一致性问题之前，有必须先介绍下一致性模型：给定一些涉及操作与状态的规则，随着操作的演进，系统将一直遵循这些规则。

一致性模型是所有被允许的操作记录的集合。当我们运行一个程序，经过一系列集合中允许的操作，特定的执行结果总是一致的。如果程序意外地执行了非集合中的操作，我们就称执行记录是非一致的。如果任意可能的执行操作都在这个被允许的操作集合内，那么系统就满足一致性模型。因为对一致性要求的程度不同，实现一致性模型的复杂度和代价也不同。

按照一致性要求的严格程度不同，有以下几种类型：

顺序一致性（Sequential consistency）

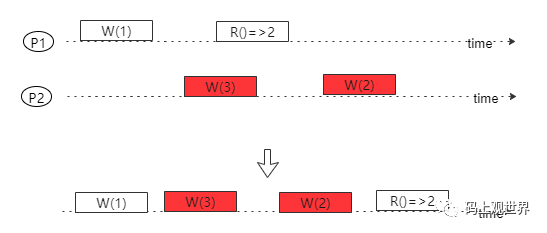
线性一致性（Linearizability）

因果一致性（Casual consistency）

### 顺序一致性

顺序一致性概念是1979年Lamport 在论文《How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs 》中提出，该概念基于在多处理器环境下如何就存储器和处理器数据达成顺序一致：假设执行结果与这些处理器以某一串行顺序执行的结果相同，同时每个处理器内部操作的执行看起来又与程序描述的顺序一致。满足该条件的多处理器系统我们就认为是顺序一致的。实际上，处理器可以看做一个进程或者一个线程，甚至是一个分布式系统。顺序一致性概念里面有两个约束：

1. 执行结果与这些处理器以某一串行顺序执行的结果相同多处理器在并发执行下会产生多种运行顺序的组合，如果能够找到一种合法的执行顺序，其结果跟把多处理器串行执行的结果一样，就可以认为符合顺序一致性模型，也就是可以做到顺序一致性。当然在设计这样的系统还需要其他的保证，比如事件通知等。这里的合法就是第二个约束的内容。
2. 每个处理器内部操作的执行看起来又与程序描述的顺序一致 一个处理器内部的操作顺序按照程序定义的顺序执行。比如在一个线程内部，其操作都是串行的，该约束很容易保证。举一个顺序一致性模型的例子：两个线程P1和P2，两种类型的操作：W(x)对共享变量写操作，R()=>x对共享变量的读操作。因为操作是有时间延迟，所以用一个矩形表示：左边沿表示操作开始，右边沿表示操作结束，沿着时间轴方向排列，但P1的R操作和P2的W操作有时间上的重叠。



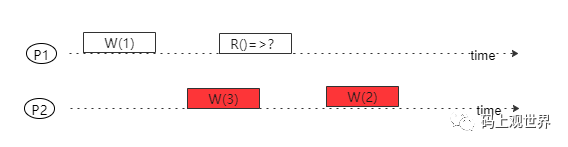
从图中可以看到两个线程如果按照W(1)=>W(3)=>W(2)=>R()，就可以保证顺序一致性，因为线程内部的两个操作都跟原来顺序保持相对一致，但是R操作时间上在W(2)后面执行。

### 线性一致性/原子一致性/严格一致性

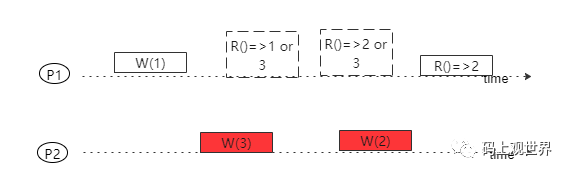
线性一致性(Linearizability)，或称原子一致性或严格一致性，指的是**程序在执行顺序组合中存在可线性化点P的执行模型**，这意味着一个操作将在程序的调用和返回之间的某个点P生效，之后被系统中并发运行的所有其他线程所感知。

线性一致性概念是1990年Maurice Herlihy · Jeannette M Wing在论文《Linearizability: A Correctness Condition for Concurrent Objects》中提出。

为理解该概念，先介绍一致性模型中同样存在的happen-before原则：任意的读写操作事件都分为调用发起和请求响应，比如读操作发起和读操作响应，如果某事件A的发起时刻在某事件B的响应时刻之后，那么就说事件B happen-before 事件A。在线程内部，所有事件都满足happen-before原则，满足此原则的操作，后续的读操作一定能够感知（获取）到写操作的结果。但是在多线程环境下，就不一定满足了，同样那上图来说明：



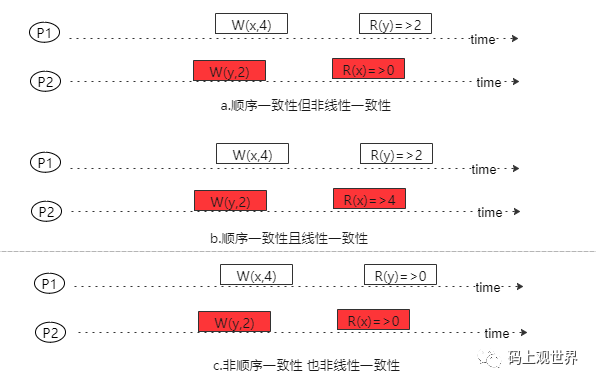
因为在进程P1内部R和W满足happen-before原则，因此W的写能够被R读取到，但是P1的R和P2的W【W(3)】不一定。我们主要关注进程P1和P2在时间上有重叠的读和写操作：在P1读取过程中，W写的响应还没返回，在写开始和返回的区间内，W写可能已经完成，也可能还在进行中，因此R读取的结果可能是1也可能是3，如果考虑到写操作的可中断性，这个值还可能是其他值，只有当P1的读和W满足上述原则，读取的值才是确定的，如下图：



但线性一致性要求如果在这个区间某个点，写操作一旦生效，那么后续的任何读一定能够获取到最新值，这意味着该写操作要么成功要么失败（所有后续读操作仍然读取到原来的值），也就是满足原子性，那么也就符合线性一致性。用直白话说就是，在并发场景下，一个线程对共享变量的操作能立即被其他线程感知到。

由于操作的延迟性，操作在发起和获得响应之间已经发生，因此线性一致性允许在操作被响应前是可以被其他线程访问到，而实现这种一致性要求操作必须是原子的：如果一个操作对于系统的其他部分是不可中断的，那这个操作就可以称之为原子的，线性的，不可分割的，那这个操作就可以称之为具有线性一致性的特性。

比较上述两种一致性模型可知：**线性一致性是对顺序一致性的加强，两者都要保证在线程内部操作的相对顺序，但线性一致性暗含着一个全局时钟，所有线程按实际发生的时间顺序执行，而顺序一致性只需要保证相对顺序即可**。满足线性一致性一定满足顺序一致性，反正不成立。为加深对两者模型的理解，再看下面的示例：



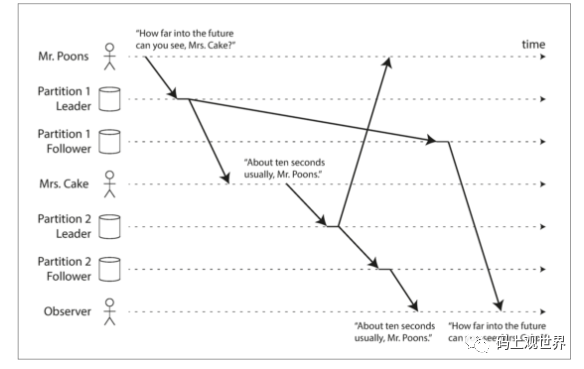
简单解释下：a中只要保证单进程的内部执行顺序基础上，R(x)和R(y)分别在W(x,4)和W(y,2)之后就得到期望结果（不论结果正确与否），符合顺序一致性，但是因为从全局时钟来看，R(x)在W(x)之后但是不能读取最新值，所以不符合线性一致性。在b中能读取到最新值，所以符合线性一致性。c因为x和y都不能读取最新值，所以不满足顺序一致性更不谈线性一致性。

### 因果一致性

线性一致性要求所有线程的操作按照一个绝对的时钟顺序执行，这意味着线性一致性是限制并发的，否则这种顺序性就无法保证。

由于在真实环境中很难保证绝对时钟同步，因此线性一致性是一种理论，实现线性一致性的代价也最高，但是实战中可以弱化部分线性一致性：只保证有因果关系的事件的顺序，没有因果关系的事件可以并发执行。

所谓因果也可以用前文中的happen-before原则来阐述：如果A事件 happen-before B事件，那么事件A，B之间存在因果关系。在分布式系统设计中，经常会因为副本数据同步的延迟导致因果关系颠倒的现象，以下引用一个问答系统中的案例：



 图中先有问然后才有答，但是因为在副本数据同步的时候，问的数据同步落后于答的数据同步，从而导致在观察者看来，先有答然后才有问。

## 分类

### 强一致性

读出的数据始终为最新写入的数据，这种一致性只有全局始终存在时才会有可能，分布式系统不太可能完全实现（即便实现投入成本很高）。

### 弱一致性

### 最终一致性

在没有更新的情况下，更新最终会通过网络传播到所有副本点，所有副本点最终会一致，也就是说使用者在最终某个时间点前的中间过程无法保证看到的是最新写入的数据。

采用最终一致性模型的一个关键要求：读取脏数据是可以接受的。

分布式系统基本都是这种一致性模型。

## 实现方案

分布式系统如何做到一致性：

1. 分布式锁
2. 分布式事务：2PC、3PC
3. 异步化：借助消息队列

### 分布式锁

1. 控制分布式系统中同步访问共享资源的一种方式
2. 不同的系统、模块需要访问共享资源的时候，需要互斥来防止彼此干扰，从而保证一致性，在这种情况下要使用分布式锁

#### DB锁

#### Redis

1. 单进程单线程模式
2. 采用队列模式将并发访问变成串行访问
3. 多客户端对Redis的连接并不存在竞争关系

基于Redis实现方式：

##### 命令

1、实现分布式锁的四个命令：

1. SETNX命令（SET if Not eXists）

语法：SETNX key value

功能：当且仅当key不存在，将key值设为value，并返回1；若给定的key已经存在，则SETNX不做任何动作，并返回0。

1. GETSET命令

语法：GETSET key value

功能：将给定key的值设置为value，并返回key的旧值，当key不存在时，返回nil

1. GET命令

语法：GET key

功能：返回key锁对应的字符串值，如果key不存在，则返回nil。

1. DEL命令

语法：DEL key[KEY...]

功能：删除给定的一个或多个key，不存在的key会被忽略，返回成功删除的数量

##### 操作步骤

2、操作步骤

1）第一步加锁

SETNX加锁实现：

调用SETNX foo.lock[current unix time]

返回为1获取到锁；返回为0没有获取到锁

2）第二步解锁

完成操作后，释放锁：DEL foo.lock

##### 死锁

3、死锁

死锁情况：

1. 加锁进程崩溃
2. 加锁进程释放锁没有成功（网络异常等）
3. 死锁

解决方案：

1. 需要检查锁的时效性
2. 一段时间后，锁失效
3. 在大并发情况下，同时检测锁失效并简单粗暴的删除死锁，再通过SETNX上锁，可能会导致竞争条件的产生，即多个客户端同时获取锁

##### 锁竞争

4、锁竞争

1）C1获取锁，并崩溃

2）C2和C3调用SETNX上锁返回0后，获得foo.lock的时间戳，通过比对时间戳，发现锁超时

3）C2向foo.lock发送DEL命令

4）C2向foo.lock发送SETNX获取锁

5）C3向foo.lock发送DEL命令，此时C3发送DEL时，其实DEL掉的是C2的锁

6）C3向foo.lock发送SETNX获取锁

7）此时C2和C3都获取了锁，产生了锁竞争情况

解决方案：

使用GETSET

##### 建议

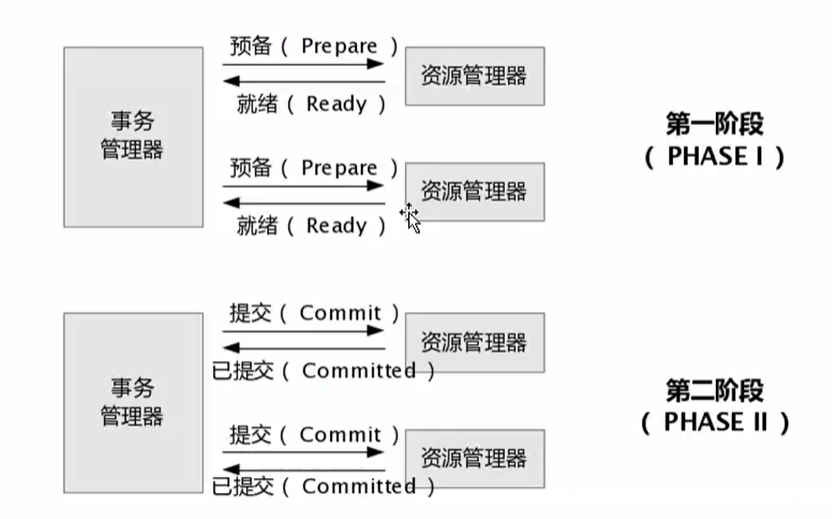
#### 应用场景

在很多互联网产品应用中，需要分布式锁处理：

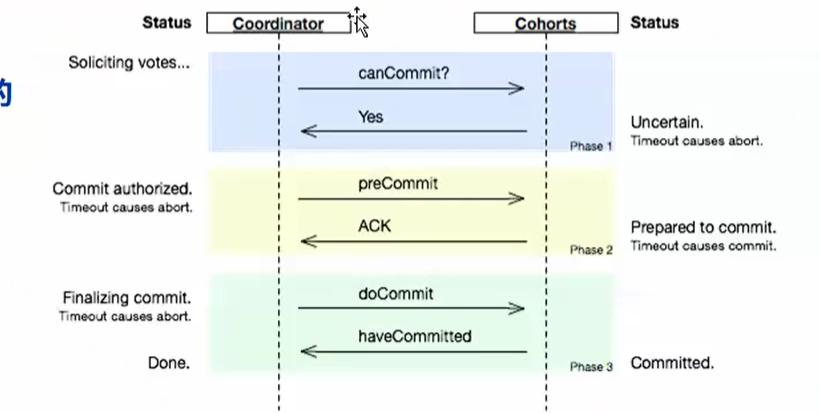
秒杀，全局递增ID，商品下订单，抢红包等

### 分布式事务

#### 2PC



#### 3PC



### 异步消息

## 总结

如果深究一致性的语义还是略有差别，这里简单归为3类介绍其区别：

Coherence更关注多核共享存储架构的cache数据一致性；

Consistence更关注分布式系统的一致性，该一致性往往以客户端为视角，比如数据库的事务一致性（ACID），分布式系统副本数据同步的CAP理论中的C都指的是这个一致性。尽管两者还有一定差异：ACID关注的是系统内部的数据一致性，CAP关注的是多副本数据对外界的一致性；

Consensus关注的是达成一致性的手段或者协议，是一种过程，比如通过Paxos达成共识，一致性是共识的目的。

几个C虽有差别，但也不是毫无关系，只是看待问题的角度和层次的差别，理论很多是相通的。

# 高可用

## 简介

高可用是分布式存储系统数据安全方面的指标，就是数据可靠不丢失。

数据高可靠性是企业的命根，一定要保证！

单机：磁盘RAID，日志记录

多机：日志记录，冗余

什么是分布式存储系统高可用性？

1. 指系统在面对各种异常时可以提供正常服务的能力
2. 系统的可用性可以用系统停服务的时间和正常服务的时间的比例来衡量
3. 系统的不可用时间（故障时间=故障修复时间点-故障发生时间点）
4. 指标

1）2个9的高可用（99%）

基本可用，一年停机的时间不能超过88个小时（365\*24\*60/100=88小时）

2）3个9的高可用

较高可用性，一年停机的时间不能超过9小时（365\*24\*60/1000=9小时）

3）4个9的可用性（99.99%）

具备自动恢复能力的高可用，一年停机的时间不超过53分钟（365\*24\*69/10000=53分钟）

4）5个9的高可用

极高可用性，一年停机的时间不能超过5分钟（365\*24\*60/100000=5分钟）

较多的大网站可用性不足2个9（88小时），设计的目标是做到4个9，具备自动恢复能力的高可用。

## 实现方案

### 硬件层面

传统企业级应用（非常昂贵）：

1. 昂贵的硬件设备
2. IBM的大型机、中型机甚至大型机
3. EMC存储设备
4. Oracle数据库

互联网公司的方案：

1. PC级服务器（价格较低）
2. 低级的PC服务器一年宕机一次是大概率事件
3. 高强度频繁读写普通硬盘，损坏的概率更高一些
4. 硬件可用性进一步降低（单机重点关注的是磁盘）

#### 机器

#### CPU

#### 内存

#### 硬盘

磁盘分类：SATA->SCSI->SAS->SSD

从左往右，性能越来越好，价格越来越高，可用性越来越高，尽可能采用高可用的磁盘。

**单磁盘可用性依然保证不了，怎么办？**

采用RAID磁盘冗余阵列，独立磁盘构成具有冗余能力的阵列，由很多价格低廉的磁盘组成容量较大的磁盘组。并行读写，提升性能。数据恢复能力，任意磁盘故障，可以读取数据，数据重构植入新硬盘。

RAID0：数据分条，即条带华

RAID1：冗余

RAID10：RAID0+RAID1

RAID01：RAID1+RAID0

RAID5：分布式奇偶校验独立磁盘

**磁盘整体故障或机器故障怎么保证高可用性？**

多机：存储系统多机冗余、数据存储多机冗余

#### 网卡

### 软件层面

#### 操作系统

#### 文件系统

#### 存储系统软件

#### 整体架构

高可用架构分层设计：

1. 数据服务和逻辑服务分离：数据存储、业务逻辑
2. 逻辑服务和接入服务分离：业务逻辑、接入层
3. 接入服务和展示服务分离：接入层、数据展示

高可用架构分层设计原则：

1. 分层服务功能单一
2. 分层间低耦合：接口交互
3. 分层内高内聚：功能聚焦单一

#### 日志

日志记录：

1. commit log
2. Binlog

日志作用：

1. 避免操作丢失
2. 用于数据恢复

#### 冗余

数据存储层冗余：数据多个副本，这样就可以实现数据的高可靠性，从而实现访问的高可靠性。

**如何实现数据存储层的冗余？**

1. 基于日志
2. Master-Slave
3. Replic-Sets（自动选主）
4. 业务层双写

数据复制：

1. 基于日志
2. MS架构，应用只写一份数据，M同步到S：MySQL、MongoDB（早期使用）
3. Replic Set：MongoDB

##### 日志

##### Master-Slave

##### Replic-Sets

##### 业务层双写

存储层多主对等结构，存储模块层对存储层双写（MySQL MMH架构）。特点：

1. 较为灵活
2. 数据模块层成本较高

#### 数据备份

##### 数据冷备份

冷备份是比较古老有效的数据保护手段，将数据复制到某种存储介质上（磁盘），系统存在故障，从冷备份设备中恢复数据。

优点：

1. 简单、廉价
2. 成本和技术难度都比较低

缺点：

1. 定期备份
2. 数据一致性保证不了
3. 数据恢复时间长，系统高可用保证不了

##### 数据热备份

热备份是online备份，提供更好的高可用性。包括异步热备份和同步热备份。

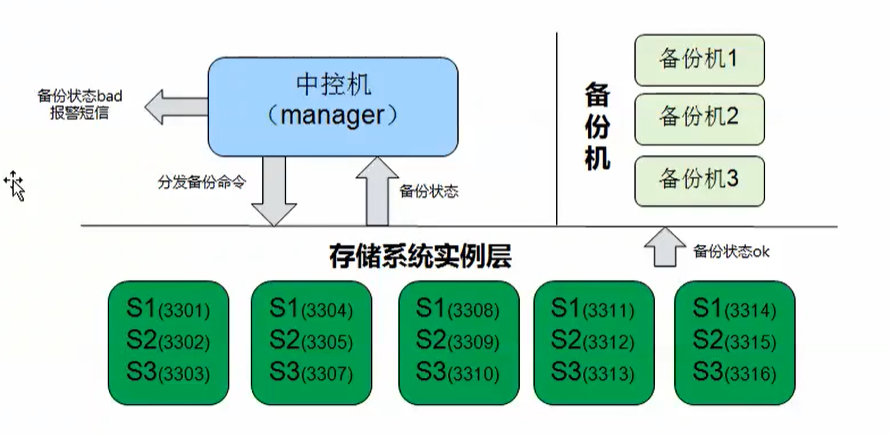
异步热备份：

1. 多个数据副本写入异步完成
2. 应用程序写入成功一份数据后，即返回
3. 由存储系统异步写入其他副本

同步热备份：

1. 多份数据副本写入同步完成
2. 应用程序收到所有副本的写入成功
3. 应用程序收到部分服务的写入成功，可能都成功了（网络故障无法返回写入成功响应）
4. 为了提高写入性能，应用程序并发写入数据
5. 没有主从之分，完全对等
6. 响应延迟是最慢的那台服务器，而不是所有延迟之和

数据备份管理系统：



#### 失效转移机制

失效确认：是否宕机，通过心跳确认。

##### 访问转移

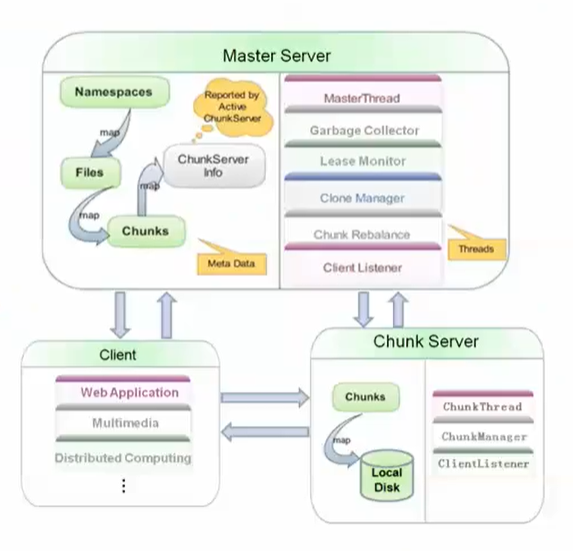
1. 访问路由到非宕机机器
2. 存储数据完全一致

##### 数据恢复

1. Master-Slave
2. 日志

## 案例分析

### 整体架构



MasterServe：

1. 主要负责对元数据进行管理，包括命名空间
2. 文件到文件块的映射，文件块到ChunkServer的映射

ChunkServer：

1. 负责管理文件块的I/O操作
2. 根据MasterServer的指令对文件块进行新建、删除和复制操作

Client：

1. 为应用端提供文件操作接口
2. 包括文件的创建、追加、读取、删除等操作

### ChunkServer宕机

ChunkServer廉价机器，宕机比较常见。

方案：

1. ChunkServer与MasterServer之间通过HeartBeat机制来通知其状态。
2. 需要将ChunkServer上所拥有的文件块拷贝到其他ChunkServer上，维持用户指定的文件块拷贝数，以保证数据的可用性。
3. 定期运行的监控线程会将该ChunkServer上的所有文件块加入到克隆队列中。

克隆线程最后即开始克隆该ChunkServer上的所有文件块到其他ChunkServer上。

在克隆过程中，该ChunkServer可能又恢复正常。此时，监控线程会将克隆队列中未进行克隆的，属于该ChunkServer的文件块移除，不需要再克隆。而已经成功克隆的块，将取代原有的块。

### 文件块数据校验

由于组成系统的成百上千台ChunkServer可能包含数千个磁盘，在数据频繁读写的情况下经常会发生磁盘损坏而导致数据的完整性遭到破坏，因此每台ChunkServer都会通过数据校验的方式来发现已损坏的数据。

当然我们可以通过在其他ChunkServer上的文件块备份来恢复数据，但是在两个ChunkServer之间进行数据块的比较显然是不合实际的。

基于校验和来对其拥有的文件块数据完整性进行校验。

系统会对没64KB的数据生成32位的校验和，并将其存储到本地文件。

当客户端或其他CHunkServer从该ChunkServer上读取数据时，它首先会对读取的数据进行校验，从而保证不会将已损坏的数据返回。一旦在校验时出错，则向请求端返回错误信息，并将MasterServer报告。请求端收到错误消息后悔向另一个文件块拷贝数据，而MasterServer对该文件进行重新拷贝，当前的文件块生成后再通知ChunkServer将已损坏的文件块删除。

在ChunkServer空闲的时候，还会对那些处于非激活状态的文件块进行验证，一旦发现坏块，MasterServer可以及时地创建新的文件块备份。

### 文件块拷贝

在ChunkServer发生宕机，报告坏的文件块，文件块的拷贝数不足，客户端发生读写错误等情况时，就需要对文件进行拷贝。

MasterServer中有一个单独的拷贝管理线程存在，该线程维护一个队列，该队列中存放需要进行拷贝的文件块信息。同时可控管理线程管理多个拷贝线程，拷贝线程的数量可以由配置文件进行配置。拷贝管理线程将任务（待克隆的文件块）下发给各个拷贝线程，执行文件块的拷贝任务。

拷贝队列是一个优先级队列，MasterServer根据多个因素来评估文件块的优先级，从而决定哪些文件块将被优先拷贝。

评估的因素包括：该文件块所剩余的拷贝数，该文件块是否有客户端在等待读写，该文件块损坏了几个拷贝，该文件块在过去的一段时间内被读写的次数。

### 垃圾回收

当一个文件被删除时，系统会立即将该操作记录到日志文件。但是系统并不会立即将该文件删除，而是将该文件通过特殊的标记重命名。系统会对这些带有特殊标记的文件进行定期扫描，如果该文件已经超过预先设置的时间段（比如24小时，可以在配置文件中设置），系统才会真正将该文件删除。在该时间段内，文件仍然可以通过该特殊的文件名访问，如果要恢复该文件，只需要简单的将文件名改为一般的文件名即可。

系统提供的文件延迟删除功能保证了文件在被误删的情况下在一段时间仍然可以被快速恢复。

# 负载均衡

## 简介

## 类型

### 客户端请求调度的负载均衡

1. 一台服务器所要承担的任务分发到两台或多台服务器上
2. 这些服务器是等价的，都可以独立的对外提供服务而不需要其他服务器的辅助
3. 与单台服务器相比，在相同的时间内可以完成更多的任务量，并且用户请求可以得到更加快速的处理

### 文件块存储的负载均衡

1. 文件块分配策略
2. 通过设计算法选取最优的若干台（由用户配置）ChunkServer存放文件块
3. 对于大文件，要尽量将其文件块分散到不同的ChunkServer上
4. 对于小文件要尽量将其存放到文件块数量较少的ChunkServer上
5. Rebalance策略

将负载较重的ChunkServer上的文件块转移到负载较轻的ChunkServer上

## 实现方案

### 客户端负载均衡算法

常见的客户端负载均衡算法：

#### 基于轮询算法的DNS（Round-robin Domain Name Server）

1. 客户端通过单一的域名访问集群服务器
2. RR-DNS通过轮询算法将该域名映射为不同的服务器IP地址
3. 实现服务器端的负载均衡

存在问题：

1. 由于客户端和服务器之间可能存在多台DNS服务器，很可能会将客户端请求转发到相同的服务器上
2. 在大规模客户端访问的情况下，很容易导致服务器间负载的不均衡现象

#### 最小连接数算法

1. 是对RR-DNS算法改进
2. 基于RR-DNS的方式把客户端请求定位到不同的服务器
3. 集群中的所有服务器都保存了其他服务器的连接数情况，当客户端建立TCP连接时，服务器会判断是否需要将该连接进行重定向。如果是，则将该连接重定向到当前连接数最少的服务器
4. 与RR-DNS算法相比，可以使得负载均衡更加动态化

存在问题：

1. 并没有考虑到服务的真正负载情况
2. 事实上连接数最小的服务并非一定就是负载最小的服务器，这同服务器具体执行的任务情况有关
3. 两种算法只是在一定程度上实现了服务器间的负载均衡
4. 由于DNS协议本身的一些特性（比如缓存和无效机制），也给负载均衡带来了一些局限性

#### 基于存储服务器负载的客户端负载均衡算法

背景：

系统在运行时，集群中每一台服务器的负载情况都是动态变化的。在设计时必须要根据服务器的实际负载情况来进行任务的分配。系统可以通过对服务器上的各项负载参数进行分析来进行调度。

负载参数的选择：CPU利用率、内存利用率、网络流量及磁盘容量等。

实际选取：CPU利用率、内存利用率、网络流量作为监测对象，通过这些参数可以计算出服务器的负载情况，进行合理调度。

设计思路：

1. MasterServer根据ChunkServer提供的负载信息来对请求进行集中调度的方式来实现ChunkServer间请求处理的负载均衡
2. 防止集群中ChunkServer负载普通都比较重的情况下MasterServer仍向其转发客户端请求，系统额外添加了一个enable参数，当ChunkServer的负载超过最大负载值（可以由管理员设定）时，会将enable设置为false
3. ChunkServer负载算法如下

假设系统中有n台ChunkServer，loadinfo[i][j]（1≤i≤n，1≤j≤3）分别表示第i台ChunkServer的CPU利用率，内存使用率及网络流量。W[j]（1≤j≤3）分别表示CPU利用率，内存利用率及网络流量的权值。则第i台ChunkServer的负载：



### 文件块存储负载均衡算法

#### 文件块分配策略

#### Rebalance策略

## 案例分析

# 高性能

## 简介

## 实现方案

### 硬件

#### CPU

#### 内存

#### 磁盘

#### 网卡

### 软件

#### 元数据服务器

元数据服务器：

1. 请求侦听：客户端、ChunkServer
2. 请求处理
3. 元数据管理
4. 文件、文件块管理
5. 心跳检测
6. ChunkServer负载均衡
7. 客户端租约管理

高性能方案：

1. Single Master Server设计

简化设计，元数据的一致性、文件块的分配策略及文件块的rebalance等。

1. 硬件层面：高性能机器
2. 架构层面：

客户端数据缓存，客户端就不需要每次都跟MasterServer进行交互，减少请求次数，直到缓存过期或文件元数据变化，避免了MasterServer热点和性能瓶颈。

1. 热备机制
2. Single Master Server
3. Stand-by Master Server

#### 客户端

客户端功能：

1. 目录管理
2. 文件管理
3. 数据流的操作
4. 资源锁的操作

高性能方案：

1. 并发性

读取文件块分散到多个ChunkServer

客户端多线程并发读取

1. 本地缓存

热点数据，本地缓存。

1. 配合远端

ChunkServer选择，负载较轻。

#### 存储节点

# 安全

## 简介

## 解决方案

### 鉴权方式

访问控制：对用户的访问权限进行管理，防止对分布式存储系统中数据的窃取和破坏。

实现方式：

1. 基于IP

IP白名单

IP黑名单

1. 基于用户授权

一般用户

读写用户

超级用户

1. 基于token

Token统一登陆

登陆授权后，再继续操作，否则拒之门外

### 数据加密方式

怎么保证数据安全性？

1. 加密
2. 加密安全通道的建立
3. 解决客户端与服务端实现加密会话问题

### 数字签名

## 安全攻击及防范手段

### XSS攻击

### 注入攻击

### CSRF

# 监控

## 简介

## 监控手段

## 数据采集方法

## 框架设计

## 第三方监控

## 性能评估

# 架构设计

# 技术展望