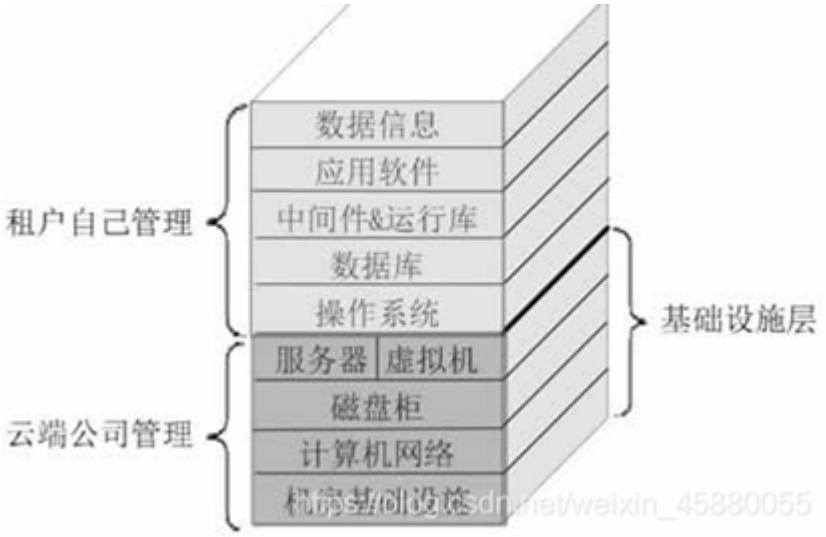
- Google MapReduce 基本工作原理
- Hadoop MapReduce 基本工作原理
- MapReduce组成
- MapReduce工作过程
- MapReduce 算法设计
    - Job Schedule 作业调度

云计算与大数据

云服务

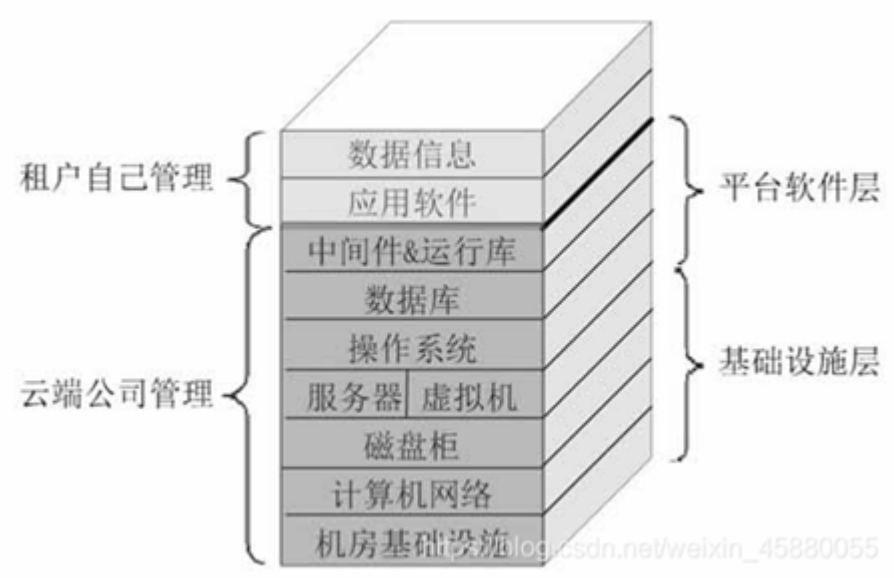
1. 基础设施即服务（**Infrastructure as a Service, IaaS**） IaaS可以提供存储、网络和防火墙等虚拟化的硬件资源，用户可以在虚拟化资源的基础上部署自身所需的数据库、应用程序。IaaS可让用户动态申请、释放资源，并且根据使用量来收费。IaaS对硬件设备等基础资源进行了封装，为用户提供了基础性的计算、存储等资源。



IaaS云服务的实际应用 备份和恢复服务。 计算服务：提供弹性资源。 内容分发网络（CDN）：把内容分发到靠近用户的地方，对于一些基于网页的应用系统，为了提高用户体验，往往在各个地方（人口稠密的地方）设立分支服务器，当用户浏览网页时，被重定向到本地 Web 服务器，所以数据必须实时分发并保持一致。 服务管理：管理云端基础设施平台的各种服务。 存储服务：提供用于备份、归档和文件存储的大规模可伸缩存储。

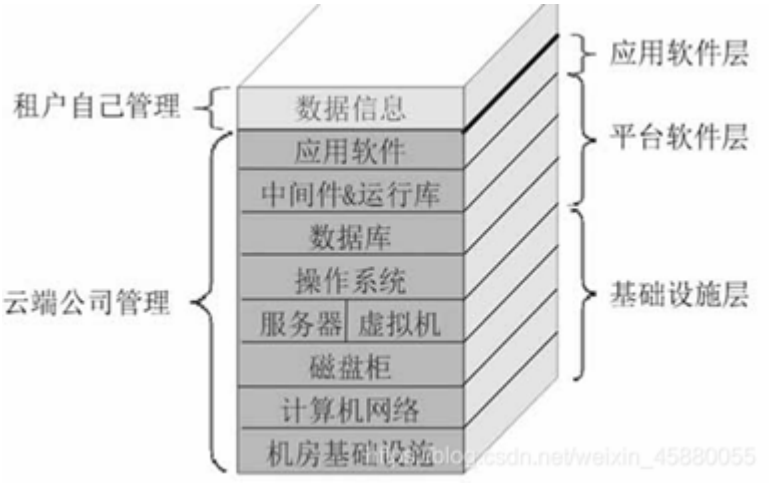
2. 平台即服务（**Platform as a Service, PaaS**） PaaS强调平台的概念，提供操作系统、编程环境、数据库、中间件和Web服务器等作为应用开发和运行的环境，用户可在此环境下开发、部署和运行各种应用。与IaaS不同的是，PaaS自身负责资源的动态扩展和容错管理，用户无须管理底层的服务器、网络和其他基础设施。

PaaS将用户与底层设施等隔离，使得用户可以专注于应用的开发



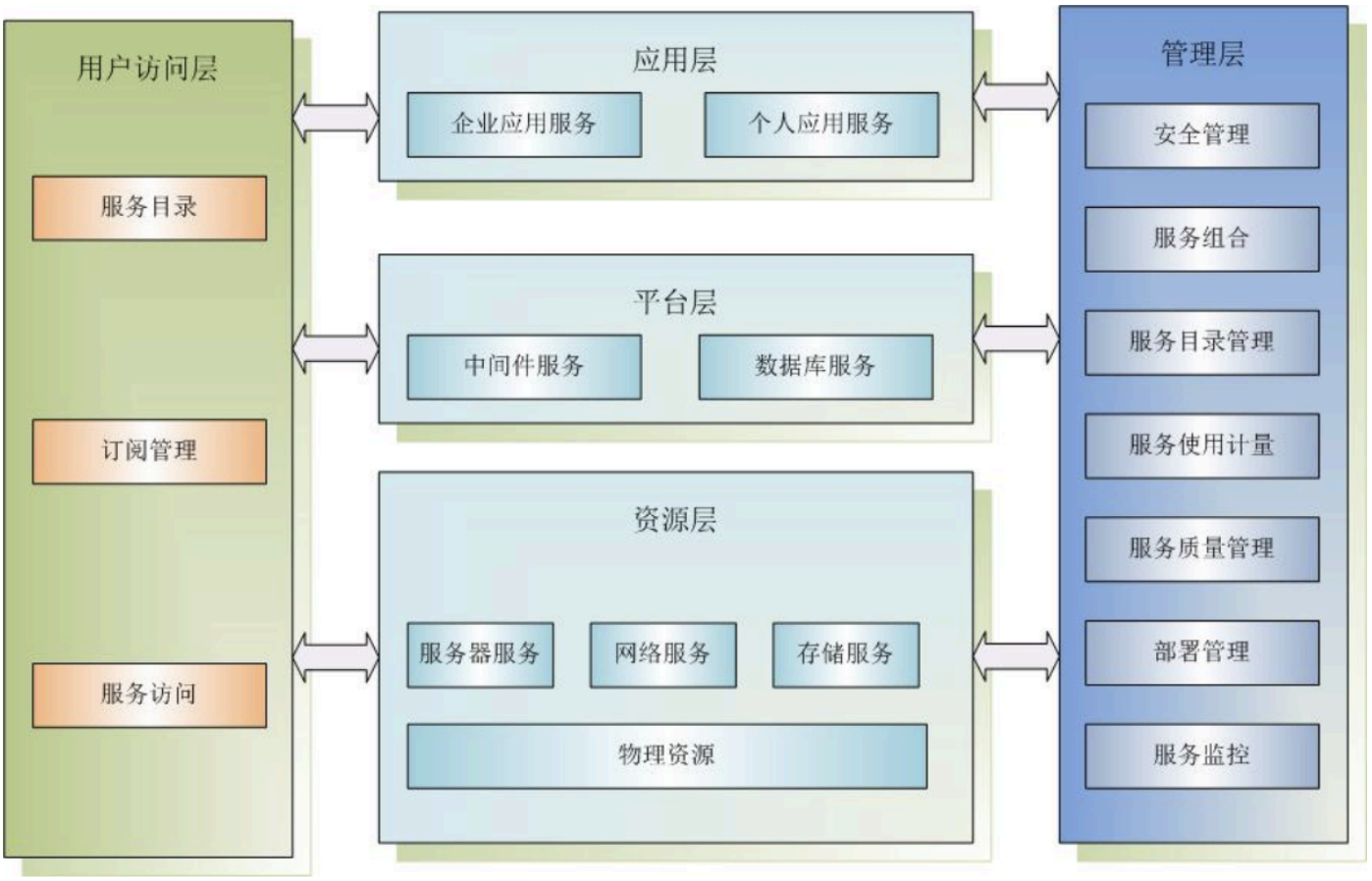
PaaS云服务的实际应用： 商业智能（BI）：用于创建仪表盘、报表系统、数据分析等应用程序的平台。 数据库：提供关系型数据库或者非关系型数据库服务。 开发和测试平台。 软件集成平台。 应用软件部署：提供应用软件部署的依赖环境。

3. 软件即服务（**Software as a Service, SaaS**） SaaS提供立即可用的软件或功能服务模块。SaaS将某些特定应用软件功能封装起来，由CSP负责管理并为客户提供服务，用户只需要通过Web浏览器、移动应用或轻量级客户端应用就可以访问这些服务



SaaS云服务的实际应用： 电子邮件和在线办公软件用于处理邮件、文字排版、电子表格和演示文档的应用软件，如谷歌邮箱、网易邮箱、微软 Office 365 在线办公、谷歌在线文档等。 内容管理系统（CMS）用于管理数字内容，包括文本、图形图像、Web页面、业务文档、数据库表单、视频、声音、XML文件等，引入版本控制、权限管理、生命周期等。财务软件。人力资源管理系统。 销售工具。 社交网络：如微信、WhatsApp、LINE 等。企业资源计划（ERP）。 谷歌在线翻译。

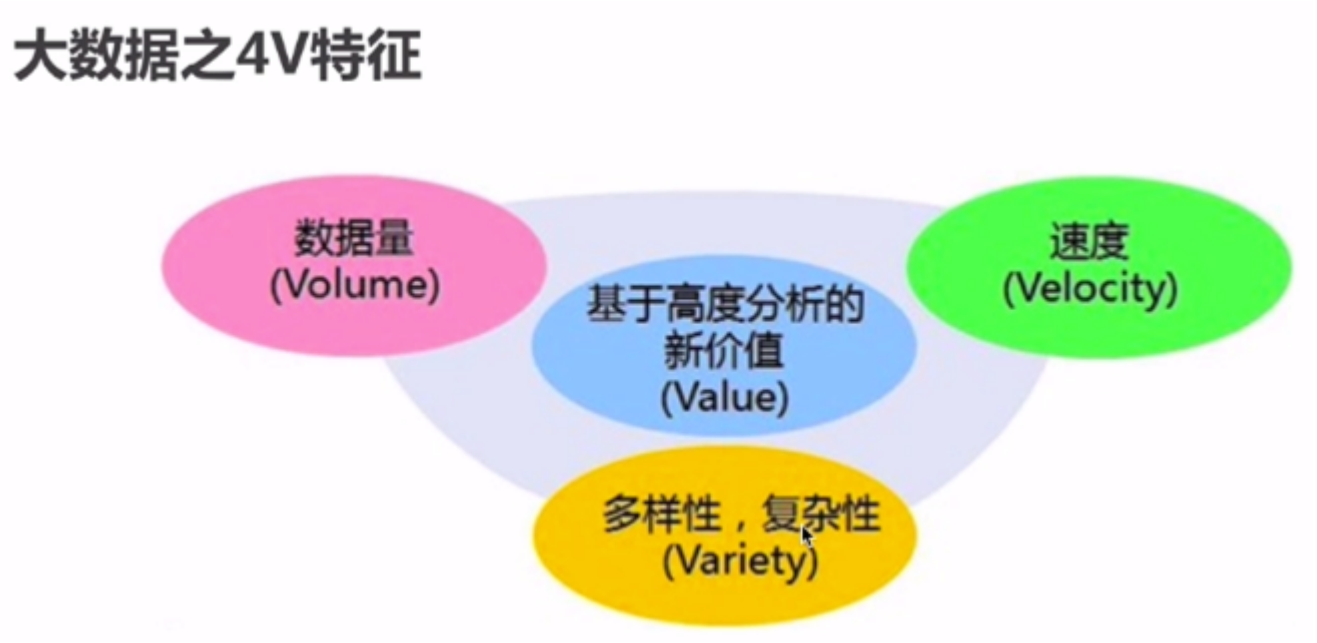
云计算的体系架构及关键技术



1. 物理资源层。 最底层的物理资源层由各种软/硬件资源构成，包括计算、网络、存储等各种资源，具体包括服务器集群（计算机）、存储设备、网络设施、数据库和软件等。
2. 虚拟化资源池层。 云计算普遍利用虚拟化或容器等技术对物理资源进行封装，构建可以共享使用的资源池，从而为上层应用和服务提供支撑。虚拟化资源池层作为实际物理资源的集成，可以更加有效地对资源进行管理和分配。
3. 管理层中间件。 为了让整个云计算可以有序地运行，还需要管理云计算的中间件，负责对云计算的用户、任务、资源和安全等进行管理。
4. SOA构建层。 SOA构建层是一个面向应用和服务的构建层，为各种应用提供各种服务，并且有效地管理各种应用，提供诸如服务注册、服务工作流等一系列用户可选择的操作，SOA构建层将云服务进行封装并提供服务接口，用户只需要调用接口就可以方便地使用云服务。

关键技术 （1）虚拟化技术；（2）分布式并行编程模型技术；（3）分布式数据存储技术；（4）分布式任务调度技术；（5）监控管理技术；（6）云计算安全保障机制；（7）云计算网络技术；（8）绿色节能技术。

大数据



新4V：

1. 变化性（Variable） 在不同的场景、不同的研究目标下，数据的结构和意义可能会发生变化，在实际研究中要考虑具体的场景和研究目标。
2. 真实性（Veracity） 获取真实、可靠的数据是保证分析结果准确、有效的前提，只有对包含大部分真实而准确的数据集进行处理才能获取正确、有意义的结果。
3. 波动性（Volatility） 数据本身包含的噪声，以及数据分析流程的不规范，采用不同的算法或不同分析过程与手段时可能会得到不稳定的分析结果。
4. 可视化（Visualization） 在大数据环境下，通过便于洞察的数据可视化方式可以更加直观地阐述数据的本质意义，帮助人们理解数据、解释结果

分布式系统概念

什么是分布式系统

分布式系统实际上就是研究如何协调这些联网的计算机来共同完成任务

特点:

1. 多进程，分布式系统中有多个进程并发运行；
2. 不共享操作系统，通过网络通信传递消息来协作；
3. 不共享时钟，所以很难只通过时间来定义两个事件的顺序；

为什么需要分布式

1. 高性能。 由于计算机硬件存在无法突破的物理限制，随着芯片工艺逼近极限，摩尔定律已失效，于是现代CPU将多个CPU拼在一起以获得更高的性能，这就是多核CPU架构。多核CPU架构存在物理限制，而且成本会迅速上升，很多公司难以承受大型机高昂的成本。
2. 可扩展性。 目前很多应用程序都是数据密集型的，应用程序大部分时间在存储和处理数据。随着业务扩展、用户增长或者历史数据积累，单台计算机只能扩展到有限的程度，无法满足需求。通过构建一个数据分布在多台计算机上的分布式存储系统，我们能够将集群扩展到单机系统根本无法想象的规模。
3. 高可用性 通过构建分布式系统，冗余多份数据来保证数据可用性；或者通过冗余计算实现服务切换，即在两台计算机上运行完全相同的任务，其中一台发生了故障，可以切换到另一台
4. 保证一致性

系统模型

> 两将军问题和拜占庭将军问题告诉我们，现实中的分布式系统，节点和网络都可能出现各种各样的故障。我们的系统模型就是根据不同种类的故障抽线出来的

网络链路模型

假设有一个发送者和接收者，它们通过一个双向的链路通信。一个链路有两个最基本的事件：发送事件，即将一条消息发送到链路上；接收事件，链路返回一条消息。我们将发送者和接收者通信的网络链路分为

1. 可靠链路 也称为完美链路，既不会丢失消息，也不会凭空捏造消息，但它可能对消息重新排序
2. 公平损失链路 消息可能会丢失、重复或重新排序，但消息最终总会到达
3. 任意链路 是最弱的一种网络链路模型。这种网络链路允许任意的网络链路执行任何操作，可能有恶意软件修改网络数据包和流量

节点故障模型

构建一个可靠的网络链路并不像想象中那么不切实际。然而，我们还必须考虑到，节点在回复消息时可能出现故障，从而导致该消息永久丢失。

崩溃-停止：指一个节点停止工作后永远不会恢复。这可能是不可恢复的硬件故障

崩溃-恢复：允许节点重新启动并继续执行剩余的步骤，一般通过持久化存储必要的状态信息来容忍这种故障类型。

拜占庭故障：如同拜占庭将军问题一样，故障的节点可能不只会宕机，还可能以任意方式偏离算法，甚至恶意破坏系统。

按时间划分系统模型

同步系统模型：指一个消息的响应时间在一个有限且已知的时间范围内。

异步系统模型：指一个消息的响应时间是无线的，无法知道一条消息什么时候会到达。

消息传递语义

> 分布式系统中的各个节点之间通过互相传递消息来协作，由于网络和节点不可靠，这些消息可能会丢失，为了解决消息丢失问题，会让节点重复发送信息，这意味着消息可能会发送多次。这种重复传递消息的行为可能会产生灾难性的副作用。比如，如果该重传的消息正好是用户银行账户的扣款信息，处理不当的话，该客户可能因为一笔交易被收取两次费用。

按照消息传递和处理次数，有如下几种可能的消息传递语义：

1. 最多一次：消息最多传递一次，消息可能丢失，但不会重复；
2. 至少一次：系统保证每条消息至少会发送一次，消息不会丢失，但在有故障的情况下可能导致消息重复发送；
3. 精确一次：消息只会被精确传递一次，这种语义是人们实际想要的，且仅有一次，消息不丢失不重复。

   我们关心的是消息被处理的次数，而不是消息被送达的次数。精确传递一次难以实现，因为想要建立可靠的链路，就必须重复传递某些消息。

一致性算法

分区

将一个数据集拆分为多个较小的数据集，同时将存储和处理这些较小数据集的责任分配给分布式系统中的不同节点

垂直分区

对表的列进行拆分，将某些列的整列数据拆分到特定的分区，并放入不同的表中（减少了表的宽度，每个分区都包含了其中的列对应的所有行）

水平分区

对表的行进行拆分，将不同的行放入不同的表中，所有在表中定义的列在每个分区中都能找到，所以表的特性依然得以保留（一个包含十年订单记录的表可以水平拆分为十个不同的分区，每个分区包含其中一年的记录）

分区算法

1. 范围分区（Range Partitioning） 指根据指定的关键字将数据集拆分为若干连续的范围，每个范围存储到一个单独的节点上

   优点：

   1. 实现起来相对容易；
   2. 能够对用来进行范围分区的关键字执行范围查询；
   3. 当使用分区键进行范围查询的范围较小且位于同一个节点时，性能良好；
   4. 很容易通过修改范围边界增加或减少范围数据，能够简单有效地调整范围

   缺点：

   1. 无法使用分区键之外的其它关键字进行范围查询；
   2. 当查询的范围较大且位于多个节点时，性能较差；
   3. 可能产生数据分布不均或请求流量不均的问题，导致某些数据的热点现象

2. 哈希分区（Hash Partitioning） 将指定的关键字经过一个哈希函数的计算，根据计算得到的值来决定该数据集的分区

优点： 数据的分布几乎是随机的，所以分布相对均匀，能够在一定程度上避免热点问题

缺点：

1. 在不额外存储数据的情况下，无法执行范围查询；
2. 在添加或删除节点时，由于每个节点都需要一个相应的哈希值，所以增加节点需要修改哈希函数，这会导致许多现有的数据都要重新映射，引起数据大规模移动，并且在此期间，系统可能无法继续工作；

3. 一致性分区（Consistent Hashing） 一致性哈希（Consistent Hashing）是一种用于分布式系统的哈希算法，旨在解决节点增加或删除时造成的大规模数据重新分配问题，从而提高系统的扩展性和稳定性。

- 基本原理

1. 哈希环：

   - 一致性哈希将整个哈希值空间（通常为一个大整数集合，如0到2^32-1）组织成一个环，称为哈希环。
   - 环的起点和终点是相连的，形成一个闭合的圆环结构。

2. 节点映射：

   - 每个存储节点（服务器）通过哈希函数映射到哈希环上的一个点。
   - 例如，节点A通过哈希函数得到一个值hash(A)，并将其映射到环上的一个位置。

3. 数据映射：

   - 每个数据对象也通过哈希函数映射到哈希环上的一个点。
   - 例如，数据对象D通过哈希函数得到一个值hash(D)，并将其映射到环上的一个位置。

4. 数据存储规则：

   - 数据对象被存储在顺时针方向上距离它最近的节点上。
   - 例如，如果数据对象D映射到的位置hash(D)在节点A和节点B之间，并且顺时针方向最近的是节点B，则数据D存储在节点B上。

- 优势

1. 节点增加或删除的影响范围小：

- 在增加一个节点时，只需将新节点顺时针方向最近的数据重新分配到新节点，其他数据不受影响。
- 在删除一个节点时，只需将该节点上的数据重新分配到顺时针方向最近的节点，其他数据不受影响。

2. 负载均衡：

   - 数据在环上较为均匀地分布，避免了某些节点过于繁忙而其他节点空闲的问题。

- 缺点：

  1. 当系统节点太少时，容易产生数据分布不均的问题；
  2. 当一个节点发生异常需要下线时，该节点的数据全部转移到顺时针的方向上，从而导致顺时针方向节点存储大量数据，大量负载会倾斜到该节点

  解决方法：引入虚拟节点，虚拟节点并不是真实的物理服务器，虚拟节点是实际节点在哈希环上的副本，一个物理节点不再只对应哈希环上一个点，而是对应多个节点

复制

> 指将同一份数据冗余存储在多个节点上，节点间通过网络来同步数据，使之保持一致。

单主复制

> 也叫主从复制或主从同步，即指定系统中的一个副本为主节点，客户端的写请求必须发送到主节点；其余的副本称为从节点，从节点只能处理读请求，并从主节点同步最新的数据。

主节点收到写请求时，除了将数据写入本地存储，还要负责将这次数据变更同步给所有从节点，以确保所有的副本保持数据一致

1. 同步复制 同步复制中的主节点执行完一个写请求后，必须等待所有的从节点都执行完毕，并收到确认信息后，才可以回复客户端写入成功
2. 异步复制 异步复制中主节点执行完写请求后，会立即将结果返回给客户端，无须等待其他副本是否写入完成

3. 半同步复制 是介于同步复制和异步复制之间的一种复制机制。主节点只需要等待至少一个从节点同步写操作并返回完成信息即可，不需要等待所有的节点都完成。这等价于，有一个从节点是同步复制，其余的从节点是异步复制，保证至少有两个节点拥有最新的数据副本，该从节点能随时接替主节点的工作。

- 单主复制优点：

  简单易懂，易于实现；仅在主节点执行并发写操作，能够保证操作的顺序，避免了还要考虑如何处理各个节点数据冲突这类复杂的情况，这个特性使得单主复制更容易支持事务类操作；对于大量读请求工作负载的系统，单主复制是可扩展的，可以通过增加多个从节点来提升读的性能。

- 单主复制缺点

  面对大量写请求工作负载时系统很难进行扩展，因为系统只有一个主节点，写请求的性能瓶颈由单个节点（主节点）决定； 当主节点宕机时，从节点提升为主节点不是即时的，可能会造成一些停机时间，甚至产生错误

多主复制

增加多个主节点来分担写请求的负载，这种由多个节点充当主节点的数据复制方式成为多主复制

解决冲突方法：

1. 由客户端解决冲突：在客户端下次读取系统中冲突数据的时候，将冲突的数据全部返回给客户端，客户端选择合适的数据并返回给存储系统，存储系统以此数据作为最终确认的数据，覆盖所有冲突的数据。

2. 最后写入胜利：让系统中的每个节点为每个写入请求标记上唯一时间戳或唯一自增ID，当冲突发生时，系统选择具有最新时间戳或最新ID版本的数据，并丢弃其他写入的数据。

3. 因果关系跟踪：系统使用一种算法来跟踪不同请求之间的因果关系，并以此判断请求的先后顺序

无主复制

没有主节点和从节点，客户端不仅向一个节点发送写请求，而是将请求发送到多个节点，在某些情况下甚至会发送给所有节点。

基于Quorum（法定人数）的数据冗余机制：分布式系统中用来保证数据冗余和最终一致性的一种算法。在无主复制中，Quorum机制用于多副本数据的一致性维护，即前面提到的客户端要向一些节点发送读写请求，Quorum机制就用于确定到底要多少个节点才足够，以及如果我们增加或减少读写请求的节点数量，系统会发生怎样的变化。

CAP定理

指出在一个异步网络环境中，对于一个分布式读写存储系统来说，只能满足以下三项中的两项，而不可能满足全部三项：

> 一致性（Consistency） ：客户端访问所有节点，返回的都是同一份最新的数据。
>
> 可用性（Availability） ：每次请求都能获取非错误的响应，但不保证获取的数据是最新数据（？）。
>
> 分区容错性（Partition Tolerance） ：节点之间由于网络分区而导致消息丢失的情况下，系统仍能继续正常运行。（什么叫网络分区）

一致性模型

一致性模型和复制有着密切的关系。复制既带来了高可用性和高性能等好处，但也带来了多个副本如何保持数据一致这个问题，尤其是写操作何时，以何种方式更新到所有副本决定了分布式系统付出怎样的性能代价

以数据为中心的一致性模型

以数据为中心的一致性模型旨在为数据存储系统提供一个系统级别的全局一致性视图，我们讨论这类一致性模型的角度都是在并发的客户端（或进程）同时更新数据时，考虑系统每个副本的数据是否一致，以及系统提供的一致性

1. 线性一致性 意味着分布式系统的所有操作看起来都是原子的，整个分布式系统看起来好像只有一个节点

   给定一个执行历史，执行历史根据并发操作可以扩展为多个顺序历史，只要从中找到一个合法的顺序历史，那么该执行历史就是线性一致性

   也就是说，只要我们能够把执行历史转为顺序历史，然后判断顺序历史是否合法，就能知道这个执行历史是否满足线性一致性。

约束：在将执行历史转变成顺序历史的过程中，如果两个操作是顺序关系，那么它们的先后关系必须保持相同；如果两个操作是并发关系，则它们可以按任何顺序排列

两个约束条件： 1、顺序记录中的任何一次读必须读到最近一次写入的数据； 2、顺序记录要跟全局时钟下的顺序一致；

2. 顺序一致性 顺序一致性是一种比线性一致性弱一些的一致性模型，同样允许对并发操作历史进行重新排列，但它的约束比线性一致性要弱，顺序一致性只要求同一个客户端（或进程）的操作在排序后保持先后顺序不变，但不同客户端（或进程）之间的先后顺序是可以任意改变的

3. 因果一致性 因果一致性是比顺序一致性更弱一些的一致性模型，它与顺序一致性一样不依赖于全局操作的顺序。因果一致性要求，必须以相同的顺序看到因果相关的操作，而没有因果关系的并发操作可以被不同的进程以不同的顺序观察到。 最典型的因果关系就是社交网络中的发帖和评论关系，根据因果关系，必须先有发帖才能有对于该帖子的评论，所以发帖操作必须在评论操作之前。

4. 最终一致性 最终指的是并没有指定系统必须达到稳定状态的硬性时间，这听起来很不可靠，但是在实践中，这个模型工作得很好，当下许多追求高性能的分布式存储系统都是使用最终一致性模型的。 例如，只要系统最终能够达到一个稳定的状态，在某个阶段，系统各节点处理客户端的操作顺序可以不同，读操作也不需要返回最新的写操作的结果。在最终的状态下，只要不再执行写操作，读操作将返回相同的、最新的结果，这就是最终一致性

以客户端为中心的一致性模型

> 以客户端为中心的一致性模型，从客户端的角度来观察分布式系统，不再从系统的角度考虑每个副本的数据是否一致，而是考虑客户端的读写请求的结果，从而推断出系统的一致性。

1. 单调读一致性模型 单调读一致性必须满足，如果客户端读到关键字x的值为v，那么该客户端对于x的任何后续的读操作必须返回v或比v更新的值，即确保客户端不会读到旧的值。

2. 单调写一致性 必须满足，同一个客户端（或进程）的写操作在所有副本上都以同样的顺序执行，即保证客户端的写操作是串行的。例如，客户端先执行写操作x=0再执行写操作x=1，如果另一个客户端不停地读x的值，那么会读到x的值为0再为1，不会先读到1再读到0

3. 读你所写一致性 也称为读己之写一致性，该一致性要求，当写操作完成后，在同一副本或其他副本上的读操作必须能够读到写入的值。注意，读你所写一致性必须是单个客户端（进程）。

4. PRAM一致性 也称为FIFO一致性，翻译为流水线随机访问存储器一致性，它由单调读、单调写和读你所写三个一致性模型组成。PRAM一致性要求：同一个客户端的多个写操作，将被所有的副本按照同样的执行顺序观察到，但不同客户端发出的写操作可以以不同的执行顺序被观察到。

5. 读后写一致性模型 要求：同一个客户端对于数据项x，如果先读到了写操作w1的结果v，那么之后的写操作w2保证基于v或比v更新的值。读后写一致性其实还约束了写操作的顺序，写操作w1一定发生在w2之前

隔离级别

> 在分布式系统中，隔离级别（Isolation Levels）是事务处理的重要概念，它决定了多个事务并发执行时如何相互影响。隔离级别的主要目的是在性能和数据一致性之间找到一个平衡。

1. 读未提交（Read Uncommitted）

描述: 事务可以读取其他未提交事务的数据。

优点: 性能最好，因为没有任何锁和等待。

缺点: 最不安全，可能会发生脏读（Dirty Read），即读取到未提交的脏数据。

适用场景: 适用于对数据一致性要求较低，但对性能要求极高的场景。

2. 读已提交（Read Committed）

描述: 事务只能读取其他已提交事务的数据。

优点: 避免了脏读。

缺点: 可能会发生不可重复读（Non-repeatable Read），即在同一事务中两次读取同一数据可能会得到不同的结果。

适用场景: 适用于大多数在线事务处理（OLTP）系统，平衡了数据一致性和性能。

3. 可重复读（Repeatable Read）

描述: 事务在开始时所读取的数据在整个事务期间不会被改变，即相同的数据在同一事务中多次读取时结果相同。

优点: 避免了脏读和不可重复读。

缺点: 可能会发生幻读（Phantom Read），即在同一事务中两次读取结果集中出现或消失了行。

适用场景: 适用于需要较高数据一致性但仍能接受一定并发性能损失的场景。

4. 可串行化（Serializable）

描述: 事务完全隔离，事务按顺序一个接一个执行，仿佛是串行执行的。

优点: 最安全的隔离级别，避免了脏读、不可重复读和幻读。

缺点: 性能最差，因为并发性大大降低，需要大量的锁和等待。

适用场景: 适用于对数据一致性要求极高且可以接受低并发性能的场景，如银行转账系统。

分布式共识

分布式共识是指在分布式系统中，多个节点就某一值达成一致的过程。共识问题的解决对保证系统的可靠性和一致性至关重要，尤其是在面对网络分区、节点故障等情况时。

异步系统中的共识

在异步系统中，节点之间的通信延迟不确定，消息可能会延迟甚至丢失，节点可能会随时失败或重启。需要处理以下问题：

1. **网络不可靠**：消息可能会丢失、延迟或乱序。
2. **节点故障**：节点可能会随时崩溃或变得不可访问。
3. **分区问题**：网络可能会出现分区，使得部分节点彼此无法通信。

Paxos

> Paxos是一种分布式共识算法，用于在异步系统中保证多个节点就某一值达成一致。Paxos协议的核心概念包括提议者（Proposer）、接受者（Acceptor）和学习者（Learner）。

- 提议者（**Proposer**）：提出提议，并负责协调达成共识的过程。
- 接受者（**Acceptor**）：接受或拒绝提议，并存储提议信息。
- 学习者（**Learner**）：学习已达成的提议结果。

Paxos协议分为两个阶段：

1. 准备阶段（**Prepare Phase**）：
    - 提议者向接受者发送prepare请求，包含一个提议编号n。
    - 接受者收到prepare请求后，如果n大于其已响应的最大提议编号，则承诺不再接受小于n的提议，并返回其已经接受的编号最大且编号小于n的提议（如果有的话）。
2. 接受阶段（**Accept Phase**）：
    - 提议者根据接受者返回的提议决定新的提议值，并向接受者发送accept请求。
    - 接受者收到accept请求后，如果提议编号不小于其已响应的最大提议编号，则接受该提议并记录下来，否则拒绝。

通过这两个阶段，Paxos协议保证即使在存在节点故障和网络分区的情况下，也能最终达成一致。

分布式锁服务 Chubby

- 提供排他锁和共享锁，用于协调分布式系统中的资源访问。
- 使用Paxos协议确保锁服务的高可用性和一致性。

Chubby的基本思路是所有集群中的服务器通过Chubby最终选出一个主服务器（Master Server），然后由这个主服务器来协调工作。

Chubby的核心组件

1. **Client Library**（客户端库）: 提供API接口，应用程序通过它与Chubby服务交互。
2. **Chubby Cell**（**Chubby**单元）: 由一组Chubby服务器组成，每个单元管理一组锁和文件。
3. **Master Server**（主服务器）: 每个Chubby单元有一个主服务器，负责处理客户端请求和管理复制状态。
4. **Replica Servers**（副本服务器）: 其他服务器作为副本服务器，存储主服务器的状态副本，并参与Paxos共识过程。

Chubby的工作原理

1. 主从架构和**Paxos**协议

Chubby采用主从架构，其中一个服务器被选举为主服务器，其他服务器作为副本服务器。选举过程使用Paxos协议来确保在所有服务器之间达成一致，选出一个主服务器。

- 选举主服务器: 当一个新的Chubby单元启动或当前主服务器失效时，系统会使用Paxos协议进行选举，选出一个新的主服务器。
- 状态复制: 主服务器负责处理客户端请求，并将状态变化复制到副本服务器。通过Paxos协议，确保状态在副本服务器之间的一致性，即使主服务器发生故障，系统仍能恢复并继续工作。

2. 锁服务

Chubby提供排他锁和共享锁，用于分布式系统中资源的协调访问。

- 排他锁（**Exclusive Lock**）: 只有一个客户端能持有，保证资源的独占访问。
- 共享锁（**Shared Lock**）: 多个客户端可以同时持有，允许资源的共享访问。

锁的获取和释放操作都通过主服务器处理，主服务器将这些操作同步到副本服务器，以确保锁状态的一致性。

3. 文件系统接口

Chubby提供一个简单的文件系统接口，支持创建、删除、读写文件和目录。文件系统接口用于存储元数据和应用配置数据。

4. 会话和租约

Chubby使用会话和租约机制来管理客户端和服务器之间的连接。

5. 故障恢复

Chubby通过Paxos协议实现故障恢复和高可用性：

- **主服务器故障**: 如果主服务器故障，副本服务器将通过Paxos协议选举出一个新的主服务器。
- **副本服务器故障**: 副本服务器故障时，系统仍能继续工作，并在故障服务器恢复后同步状态。

分布式应用协调器 Zookeeper

核心组件

1. **Znode（数据节点）**: Zookeeper中的基本数据单元，类似于文件系统中的文件和目录。Znode可以存储数据并形成树状结构。
2. **Zookeeper Ensemble（集群）**: 由多个Zookeeper服务器组成的集群。集群中的一个节点被选举为Leader，其余为Follower。
3. **Client（客户端）**: 使用Zookeeper API与集群进行交互的应用程序。

数据模型

Zookeeper的数据模型是一个分层的树状结构，类似于文件系统。每个节点称为Znode，分为持久节点和临时节点两种：

- **持久节点（Persistent Znode）**: 一旦创建，除非被显式删除，否则永久存在。
- **临时节点（Ephemeral Znode）**: 由客户端创建，与客户端会话绑定，当会话结束或超时，节点自动删除。

一致性协议

Zookeeper使用Zab协议（Zookeeper Atomic Broadcast）来实现一致性和高可用性。

- **Leader选举**: 当Zookeeper集群启动或当前Leader失效时，集群会通过选举机制选出一个新的Leader。
- **事务处理**: 所有的写请求（如创建、更新、删除Znode）都必须通过Leader处理，Leader将这些请求转换为事务提案，并广播给所有Follower。Follower确认接收到

提案后，Leader提交事务，确保数据一致性。
- **数据同步**: Leader负责与Follower同步数据，保证所有节点的数据一致。

主要功能

1. **命名服务**: 提供分布式系统中的唯一命名空间，用于节点之间的命名和查找。
2. **配置管理**: 支持分布式系统中的配置集中管理和动态更新。
3. **分布式锁**: 提供高效的分布式锁机制，协调多个客户端对共享资源的访问。
4. **集群管理**: 支持集群节点的动态加入和退出，监控节点的状态和健康状况。
5. **同步和协调**: 通过Watcher机制和数据存储，支持分布式系统中的事件驱动和任务协调。

工作流程

1. **启动和初始化**: Zookeeper集群启动时，各个服务器节点相互连接并进行Leader选举，选出一个Leader，其他节点作为Follower。
2. **客户端连接**: 客户端通过Zookeeper API连接到集群中的任一服务器，服务器为客户端分配会话ID，并维护会话的心跳和超时。
3. **数据操作**: 客户端可以对Znode进行创建、读取、更新和删除操作。写操作由Leader处理并同步给Follower，读操作可以由任一服务器处理。
4. **事件通知**: 客户端可以在Znode上设置Watcher，当Znode状态发生变化时，Zookeeper服务器会通知相应的客户端。

Zookeeper通过其简洁的接口和高效的实现，成为许多分布式系统（如Hadoop、Kafka）的核心协调组件。

Zookeeper和Chubby的异同点

- 相同点： 两者的数据模型相同，都是树形的层级目录结构，类似传统文件系统。 两者的节点相同，都分为临时节点和持久型节点。 Chubby的订阅和Zookeeper的观察标志类似。 写或更新数据操作都需要在主服务器上完成。
- 不同点： Chubby强调系统的可靠性以及高可用性等，而不追求处理高吞吐量；Zookeeper能处理高吞吐量。 Chubby只有主节点能提供读数据的服务，Zookeeper中从节点也能提供读数据服务。 一致性协议上Chubby使用PAXOS和Zookeeper使用ZAB。 Chubby的主节点有租约，租约到期没问题继续续约，Zookeeper谁是主节点就一直是谁，除非人为改变或发生故障等，没有租约概念。

分布式事务

原子提交（Atomic Commit）

原子提交协议确保在分布式系统中的事务要么全部提交成功，要么全部回滚失败，不会出现部分成功、部分失败的状态。

两阶段提交协议（2PC）

两阶段提交协议通过两个阶段来确保事务的原子性：

1. 准备阶段（**Prepare Phase**）
    - 协调者（Coordinator）向所有参与者（Participant）发送准备请求（prepare request）。
    - 每个参与者执行事务操作，但不提交，仅记录操作日志，并返回准备好的消息（vote-commit）或拒绝准备的消息（vote-abort）。
2. 提交阶段（**Commit Phase**）
    - 如果所有参与者都返回准备好的消息，协调者向所有参与者发送提交请求（commit request），各参与者提交事务。
    - 如果有任意一个参与者返回拒绝准备的消息，协调者向所有参与者发送回滚请求（rollback request），各参与者回滚事务。

2PC的主要优点是其实现简单，能够保证事务的原子性和一致性，但也有缺点，如存在阻塞问题和单点故障风险。

三阶段提交协议（3PC）

三阶段提交协议在两阶段提交的基础上增加了一个预提交阶段，以减少阻塞问题和单点故障风险：

1. 准备阶段（**Prepare Phase**）
    - 协调者向所有参与者发送准备请求。
2. 预提交阶段（**Pre-Commit Phase**）

- 如果所有参与者都返回准备好的消息，协调者向所有参与者发送预提交请求（pre-commit request）。
- 参与者执行事务操作并准备提交，返回预提交确认消息（ack）。

3. 提交阶段（**Commit Phase**）

   - 协调者接收到所有参与者的预提交确认消息后，向所有参与者发送提交请求（commit request），各参与者提交事务。
   - 如果在任意阶段出现失败，协调者发送回滚请求，各参与者回滚事务。

Paxos 提交算法（Paxos Commit Algorithm）

Paxos 提交算法基于 Paxos 共识算法，用于确保分布式事务的一致性和容错性。它的基本思路是通过一组节点之间的投票来决定事务的提交或回滚。

主要步骤:

1. **准备阶段**：协调者（Proposer）向所有接受者（Acceptors）发送准备请求（Prepare Request），包含提议编号。
2. **接受阶段**：接受者收到准备请求后，如果提议编号大于已处理的提议编号，则返回承诺（Promise）不再接受较小编号的提议。
3. **提交阶段**：协调者收到多数接受者的承诺后，发送提交请求（Commit Request），如果多数接受者同意，则事务提交。

基于 Quorum 的提交协议（Quorum-Based Commit Protocol）

基于 Quorum 的提交协议利用多数投票机制（Quorum）来决定事务的提交或回滚，以提高系统的容错性和可用性。

主要步骤:

1. **准备阶段**：协调者向所有参与者发送准备请求，参与者记录操作并返回响应。
2. **Quorum 确认**：协调者等待至少 Quorum 数量的参与者返回准备好的消息。
3. **提交阶段**：如果收到的准备好的消息达到 Quorum 数量，协调者发送提交请求；否则发送回滚请求。
4. **参与者执行**：参与者接收到提交请求后提交事务，接收到回滚请求后回滚事务。

Saga 事务（Saga Transactions）

Saga 事务是一种长事务管理模式，通过将一个长事务分解为一系列独立的子事务，每个子事务都有对应的补偿操作，确保最终一致性。

主要步骤:

1. **分解事务**：将长事务分解为多个独立的子事务，每个子事务都可以独立提交。
2. **执行子事务**：依次执行每个子事务。
3. **补偿操作**：如果某个子事务失败，则执行相应的补偿操作回滚之前的子事务。

总结

- **两阶段提交（2PC）**：实现简单，但存在阻塞和单点故障风险。
- **三阶段提交（3PC）**：减少阻塞，提高容错性，但实现复杂。
- **Paxos 提交算法**：高容错性，适用于动态环境，但性能开销大。
- **基于 Quorum 的提交协议**：提高容错性，减少单点故障，但需要处理网络分区问题。
- **Saga 事务**：适用于长时间运行的事务，通过补偿操作确保最终一致性，但需要复杂的业务逻辑支持。

并发控制（Concurrency Control）

并发控制在分布式系统中用于管理多个事务同时访问共享资源时的数据一致性和完整性。主要的并发控制机制包括：

1. 锁机制（**Locking Mechanisms**）
    - **两阶段锁协议（2PL, Two-Phase Locking）**: 将事务的锁操作分为扩展阶段和收缩阶段，确保事务在释放任何锁之前不会再请求新的锁，以保证可串行化。
    - **读写锁（Read-Write Lock）**: 允许多个事务同时读取数据，但只有一个事务能写数据。
2. 时间戳排序（**Timestamp Ordering**）
    - 每个事务在开始时获取一个全局唯一的时间戳，并根据时间戳排序来决定事务的执行顺序，以确保一致性。

3. 乐观并发控制（**Optimistic Concurrency Control**）
    - 假设事务冲突很少，事务在执行期间不使用锁，而是在提交时检查冲突，如果发现冲突则回滚并重试。
4. 多版本并发控制（**MVCC, Multi-Version Concurrency Control**）
    - 为每个数据项维护多个版本，事务在读取数据时读取特定版本，写入数据时创建新版本，从而避免读写冲突。

Percolator

> Percolator是Google设计的一种分布式事务处理系统，旨在支持大规模数据的增量处理。它通过多版本并发控制（MVCC）和两阶段提交（2PC）来实现分布式事务。

主要特点

- 多版本并发控制（**MVCC**）
    - Percolator使用MVCC来管理数据版本，每个数据项有多个版本，事务在读取数据时读取特定版本，从而避免读写冲突。
- 两阶段提交（**2PC**）
    - 使用两阶段提交协议来保证事务的原子性和一致性。协调者在准备阶段向参与者发送准备请求，所有参与者准备好后，协调者在提交阶段发送提交请求。

工作流程

1. 事务开始
    - 事务开始时，获取一个全局唯一的时间戳，作为事务的标识。
2. 读取数据
    - 事务读取数据时，根据时间戳读取特定版本的数据。
3. 写入数据
    - 事务写入数据时，创建新的数据版本，并记录到本地临时存储。
4. 提交事务
    - 提交阶段使用两阶段提交协议，协调者向所有参与者发送准备请求，所有参与者准备好后，协调者发送提交请求，各参与者提交事务。
5. 回滚事务

- 如果任意阶段出现错误或冲突，协调者发送回滚请求，各参与者回滚事务操作。

分布式文件系统

Google File System (GFS)

- 工作机制
  - 文件分块：文件被分割成固定大小的块（Chunk），每个块由多个副本存储在不同的 ChunkServer 上。
  - 元数据管理：Master 节点负责存储元数据（如块的位置信息），并管理 ChunkServer 的状态。
  - 数据读写：客户端直接与 ChunkServer 交互进行数据读写，Master 仅负责提供块的位置。
  - 容错与恢复：Master 监控 ChunkServer 的状态，通过心跳机制检测故障，并自动重新复制丢失的块。

Hadoop Distributed File System (HDFS)

HDFS 体系结构

> HDFS采用了主从（Master/Slave）结构模型，一个HDFS集群包括一个名称节点（NameNode）和若干个数据节点（DataNode）

1. **NameNode**
   - NameNode 是 HDFS 的主节点，负责管理文件系统的元数据，包括文件系统的目录结构、文件和块的映射信息，以及块的位置。
2. **DataNode**
   - DataNode 是存储节点，负责存储实际的数据块。每个文件被分成多个块，每个块默认大小为 128MB，存储在多个 DataNode 上。
3. **Secondary NameNode**
   - Secondary NameNode 并不是 NameNode 的备份，而是定期与 NameNode 进行元数据快照和编辑日志合并，以减少 NameNode 恢复时的启动时间。
4. 客户端（**Client**）

    - HDFS 交互的接口。客户端可以向 HDFS 写入数据、读取数据、创建和删除文件或目录。

HDFS 存储原理

- 冗余数据保存
    - 数据副本机制: 数据存储时，通常会将数据分成多个块，并在不同的节点上保存多个副本。这样即使某个节点发生故障，其他节点上的副本仍然可以提供数据服务。
- 数据存取策略
    - 均匀分布、热点数据放置、数据本地性等
- 数据错误与恢复
    - 数据校验、故障检测、数据恢复、日志记录

HDFS 读写过程

写数据过程

1. 请求写入
    - 客户端请求写入文件，向 NameNode 发送创建文件请求。
2. 分配块
    - NameNode 创建文件条目并为文件分配第一个数据块，返回包含数据块 ID 和 DataNode 列表的响应。
3. 数据写入
    - 客户端将数据分块，并依次向 DataNode 写入数据。每个 DataNode 接收到数据后，将数据传递给下一个 DataNode，直到所有副本都写入完成。
4. 确认写入
    - DataNode 返回确认信息，客户端接收到所有确认后，通知 NameNode 写入完成。

读数据过程

1. 请求读取
    - 客户端向 NameNode 发送读取文件请求。
2. 获取块信息
    - NameNode 返回文件的块信息和存储这些块的 DataNode 列表。
3. 数据读取

- 客户端直接连接到最近的 DataNode 读取数据。如果读取失败，客户端会尝试从其他副本读取。

4. **数据校验**
   - 读取过程中，DataNode 会对数据进行校验，确保数据完整性。

	HDFS	GFS
安全模式	获知数据块副本状态，若副本数目不足，则复制副本至安全数目，因此 HDFS 具备安全模式。	GFS 不具备安全模式。当 API 读取副本失败时，Master 负责发起复制任务。
空间回收机制	文件删除时，仅仅删除目录结构，实际数据的删除在等待一段时间后实施，这样便于恢复文件，HDFS 具备空间回收机制。	GFS 并不复制文件，即不具备空间回收机制。
中心服务器模式	单一中心服务器模式，存在单点故障。	多台物理服务器，选择一台对外服务，损坏时可选择另外一台提供服务。
数据服务器管理模式	DataNode 通过心跳的方式告知 NameNode 其生存状态。NameNode 损坏后，NameNode 恢复时需要花费一段时间获知 DataNode 的状态。	ChunkServer 在 Chubby 中获取独占锁表示其生存状态，Master 通过轮询这些独占锁获知 ChunkServer 的生存状态，当 Master 损坏时，替补服务器可以快速获知 ChunkServer 状态。

分布式存储系统

BigTable

主要特点

1. **分布式设计**：数据被分割、分布和存储在多个节点上，通过 GFS 提供的底层存储支持。
2. **列式存储**：数据以列族（Column Family）的形式组织，每个列族可以包含多个列，适用于稀疏数据和大量属性的存储。
3. **稀疏矩阵索引**：采用稀疏矩阵索引结构，支持高效的数据查找和范围查询。
4. **自动分片**：数据自动分片和负载均衡，支持动态扩展和收缩。
5. **高可用性**：通过数据的多副本和自动故障转移实现高可用性。

HBase

HBase 是 Apache Hadoop 项目的一部分，是一个开源的分布式列式存储系统，设计用于存储和处理大规模结构化数据。它是 BigTable 的开源实现，提供了类似于 BigTable

的数据模型和接口。

主要特点

1. 基于 **Hadoop**：HBase 构建在 Hadoop 文件系统（HDFS）之上，利用 Hadoop 的分布式计算和存储能力。
2. **列式存储**：数据以列族的形式存储，每个列族可以包含多个列，支持动态列。
3. **高可用性**：通过数据的多副本和主从架构实现高可用性和容错性，支持自动故障转移。
4. **面向列的查询**：支持快速的列级别查询和范围查询，适用于分析型应用场景。
5. **线性可扩展性**：支持动态扩展和收缩，可以根据需求添加或移除节点。

HBase 与 BigTable 的比较

分布式计算模型

Google MapReduce 基本工作原理

Google MapReduce 是 Google 开发的一种用于大规模数据处理的编程模型和计算框架。其基本工作原理如下：

1. **Map** 阶段：
    - 输入数据被分割成多个小的数据块，每个数据块由一个 Map 任务处理。Map 任务将输入数据经过映射函数处理，生成中间键值对。
2. **Shuffle** 阶段：
    - 中间键值对按照键值进行排序，并根据键的哈希值将相同键的值分配到同一个 Reduce 任务上。这个过程称为 Shuffle。
3. **Reduce** 阶段：
    - Reduce 任务对分配到同一键的值进行聚合计算，生成最终的结果。

Google MapReduce 通过自动分配任务、数据并行处理和容错性等特性，实现了高效的大规模数据处理。

Hadoop MapReduce 基本工作原理

其基本工作原理与 Google MapReduce 类似：

1. **Map** 阶段：
    - 输入数据被切分成多个 InputSplit，每个 InputSplit 由一个 Map 任务处理。Map 任务将输入数据经过映射函数处理，生成中间键值对。
2. **Shuffle** 阶段：
    - 中间键值对按照键值进行排序，并根据键的哈希值将相同键的值分配到同一个 Reduce 任务上。这个过程由框架自动完成。
3. **Reduce** 阶段：
    - Reduce 任务对分配到同一键的值进行聚合计算，生成最终的结果。

Hadoop MapReduce 提供了高可用性、高可扩展性和容错性等特性，广泛应用于大规模数据处理。

MapReduce组成

> MapReduce 把任务分为 Map 阶段和 Reduce 阶段。开发人员使用存储在HDFS 中数据（可实现快速存储），编写 Hadoop 的 MapReduce 任务。由于 MapReduce工作原理的特性， Hadoop 能以并行的方式访问数据，从而实现快速访问数据

MapReduce体系结构主要由四个部分组成，分别是：Client、JobTracker、TaskTracker以及Task

1. Client 用户编写的MapReduce程序通过Client提交到JobTracker端 用户可通过Client提供的一些接口查看作业运行状态
2. JobTracker JobTracker是MapReduce作业的主管，负责作业的调度、监控和管理。它的主要作用包括：
    - 资源调度：JobTracker负责监控集群中各个节点的资源使用情况，并根据作业的需求进行资源调度，确保作业能够顺利执行。
    - 作业调度：JobTracker接收Client提交的作业，并将作业分解成一个个任务（Task），然后将这些任务分配给各个TaskTracker执行。

- 任务监控：JobTracker负责跟踪任务的执行状态、资源使用情况等信息，并在任务失败或超时时进行处理，例如重新分配任务或标记任务为失败。
- 故障恢复：JobTracker会周期性地备份作业执行的状态信息，以防止JobTracker自身发生故障时可以快速恢复。

3. TaskTracker TaskTracker是运行在各个数据节点上的守护进程，负责执行由JobTracker分配的任务。它主要有两个作用：
   - 执行Map和Reduce任务：TaskTracker接收JobTracker分配的任务，启动对应的Mapper或Reducer任务，并监控任务的执行情况。一旦任务完成，TaskTracker会将结果通知给JobTracker。
   - 向JobTracker汇报状态：TaskTracker会周期性地向JobTracker汇报自己的状态信息，包括可用资源、运行的任务等。这样，JobTracker可以及时了解到各个节点的负载情况，从而做出更好的调度决策。
4. Task 在MapReduce中，Task是指作业中的一个具体的执行单元，它可以是Map任务或Reduce任务。每个作业都由若干个Map任务和Reduce任务组成。Task的主要作用包括：
   - 执行Map或Reduce逻辑：Map任务负责处理输入数据并生成中间结果，而Reduce任务负责对中间结果进行合并和处理。Task根据作业的配置，执行相应的Map或Reduce逻辑。
   - 处理数据分片：Map任务负责处理输入数据的分片，将其转换为键值对。Reduce任务则负责合并和处理Map任务输出的中间结果。
   - 与TaskTracker通信：Task与TaskTracker之间通过心跳机制进行通信，以便TaskTracker可以监控任务的执行状态，并及时处理异常情况。

MapReduce工作过程

1. 输入阶段 输入数据通常存储在Hadoop分布式文件系统（HDFS）中，可以是一个或多个文件，每个文件可能被分成多个数据块，分布在不同的数据节点上。 在MapReduce作业开始时，输入阶段会从HDFS中并行读取数据，每个数据节点负责读取自己所存储的数据块，并将数据发送给相应的Mapper任务进行处理。
2. 映射阶段（Map Phase） 在映射阶段，每个Mapper任务会将**输入数据按照一定的逻辑拆分成<key, value>对**，并对每个<key, value>对执行一次映射操作。
3. 分区、排序和合并阶段： 在映射阶段输出结果后，MapReduce框架会对Mapper输出的中间结果进行分区和排序，以便将具有相同键的数据合并在一起。 这个过程可以让相同键的数据落到同一个Reducer任务上，减少数据的传输量和提高Reducer的处理效率。
4. 洗牌阶段（Shuffle Phase）： 洗牌阶段是MapReduce的核心步骤之一，**它负责将Mapper输出的中间结果按照键的哈希值范围进行重新分配，以便将具有相同键的**

数据发送到同一个**Reducer**任务上。 这个过程需要进行大量的网络传输和数据交换，因为中间结果需要从各个Mapper节点传输到相应的Reducer节点。

5. 归约阶段（Reduce Phase）： 在归约阶段，每个Reducer任务会接收来自多个Mapper节点的中间结果，对相同键的数据进行归约操作，例如求和、计数等。 归约操作的结果会被写入到输出文件中，形成最终的输出结果。
6. 输出阶段： 最后，输出阶段将Reducer任务的输出结果写入到HDFS中指定的输出目录中，这样就完成了MapReduce作业的执行。 输出结果通常可以是一个或多个文件，可以是文本文件、序列化文件等形式，具体格式由用户指定。

MapReduce 算法设计

MapReduce 算法设计主要包括以下几个步骤：

1. 问题分解：
    - 将复杂的问题分解成多个独立的子问题，每个子问题可以独立地在 MapReduce 框架上进行处理。
2. **Mapper** 设计：
    - 设计 Mapper 阶段的映射函数，将输入数据转换成中间键值对，以便进行后续的聚合计算。
3. **Reducer** 设计：
    - 设计 Reducer 阶段的聚合函数，对相同键的值进行聚合计算，生成最终的结果。
4. 数据分区和排序：
    - 对中间键值对进行分区和排序，确保相同键的值被分配到同一个 Reduce 任务上，并按照键值排序。

Job Schedule 作业调度

作业调度是指将用户提交的作业分配到集群中的计算节点上执行的过程，主要包括作业的调度和任务的调度两个层面。

1. 作业调度：

- 根据集群资源和作业优先级等因素，将用户提交的作业分配到集群中的某个计算节点上执行。

2. 任务调度：

- 作业执行过程中，将作业拆分成多个任务，并根据任务的依赖关系和数据位置等因素，将任务调度到不同的计算节点上执行。