分布式文件系统设计，该从哪些方面考虑？

转载

公众号:方志朋 于 2020-03-15 20:01:00 发布 25 收藏
文章标签： 分布式 大数据 hadoop 数据库 redis

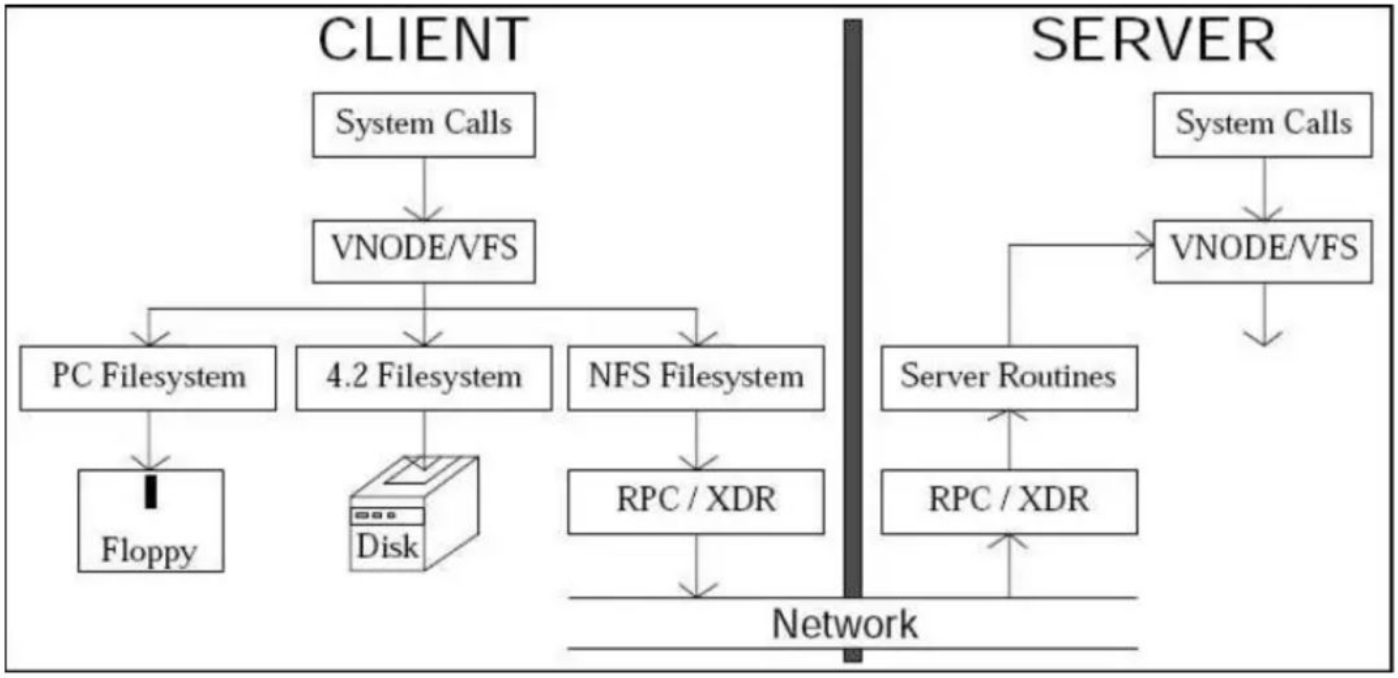
点击上方“方志朋”，选择“设为星标”

回复”666“获取新整理的面试文章

分布式文件系统是分布式领域的一个基础应用，其中最著名的毫无疑问是 HDFS/GFS。如今该领域已经趋向于成熟，但了解它的设计要点和思想，对我们将来面临类似场景 / 问题时，具有借鉴意义。并且，分布式文件系统并非只有 HDFS/GFS 这一种形态，在它之外，还有其他形态各异、各有千秋的产品形态，对它们的了解，也对扩展我们的视野有所俾益。本文试图分析和思考，在分布式文件系统领域，我们要解决哪些问题、有些什么样的方案、以及各自的选择依据。

过去的样子

在几十年以前，分布式文件系统就已经出现了，以 Sun 在 1984 年开发的“Network File System (NFS)”为代表，那时候解决的主要问题，是网络形态的磁盘，把磁盘从主机中独立出来。这样不仅可以获得更大的容量，而且还可以随时切换主机，还可以实现数据共享、备份、容灾等，因为数据是电脑中最重要的资产。NFS 的数据通信图如下。



部署在主机上的客户端，通过 TCP/IP 协议把文件命令转发到远程文件 Server 上执行，整个过程对主机用户透明。

到了互联网时代，流量和数据快速增长，分布式文件系统所要解决的主要场景变了，开始需要非常大的磁盘空间，这在磁盘体系上垂直扩容是无法达到的，必须要分布式，同时分布式架构下，主机都是可靠性不是非常好的普通服务器，因此容错、高可用、持久化、伸缩性等指标，就成为必须要考量的特性。

对分布式文件系统的要求

对一个分布式文件系统而言，有一些特性是必须要满足的，否则就无法有竞争力。主要如下：

应该符合 POSIX 的文件接口标准，使该系统易于使用，同时对于用户的遗留系统也无需改造；

对用户透明，能够像使用本地文件系统那样直接使用；

持久化，保证数据不会丢失；

具有伸缩性，当数据压力逐渐增长时能顺利扩容；

具有可靠的安全机制，保证数据安全；

数据一致性，只要文件内容不发生变化，什么时候去读，得到的内容应该都是一样的。

除此之外，还有些特性是分布式加分项，具体如下：

支持的空间越大越好；

支持的并发访问请求越多越好；

性能越快越好；

硬件资源的利用率越高越合理，就越好。

架构模型

从业务模型和逻辑架构上，分布式文件系统需要这几类组件：

存储组件：负责存储文件数据，它要保证文件的持久化、副本间数据一致、数据块的分配 / 合并等等；

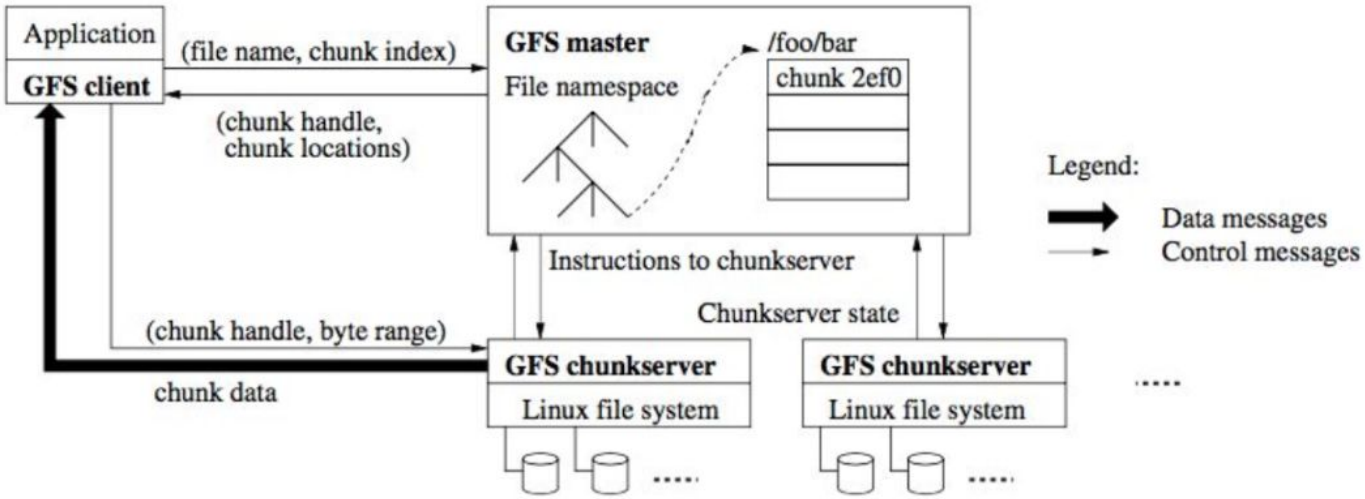
管理组件：负责 meta 信息，即文件数据的元信息，包括文件存放在哪台服务器上、文件大小、权限等，除此之外，还要负责对存储组件的管理，包括存储组件所在的服务器是否正常存活、是否需要数据迁移等；

接口组件：提供接口服务给应用使用，形态包括 SDK(Java/C/C++ 等)、CLI 命令行终端、以及支持 FUSE 挂载机制。

而在部署架构上，有着“中心化”和“无中心化”两种路线分歧，即是否把“管理组件”作为分布式文件系统的中心管理节点。两种路线都有很优秀的产品，下面分别介绍它们的区别。

有中心节点

以 GFS 为代表，中心节点负责文件定位、维护文件 meta 信息、故障检测、数据迁移等管理控制的职能，下图是 GFS 的架构图。



该图中 GFS master 即为 GFS 的中心节点，GF chunkserver 为 GFS 的存储节点。其操作路径如下：

Client 向中心节点请求“查询某个文件的某部分数据”；

中心节点返回文件所在的位置 (哪台 chunkserver 上的哪个文件) 以及字节区间信息；

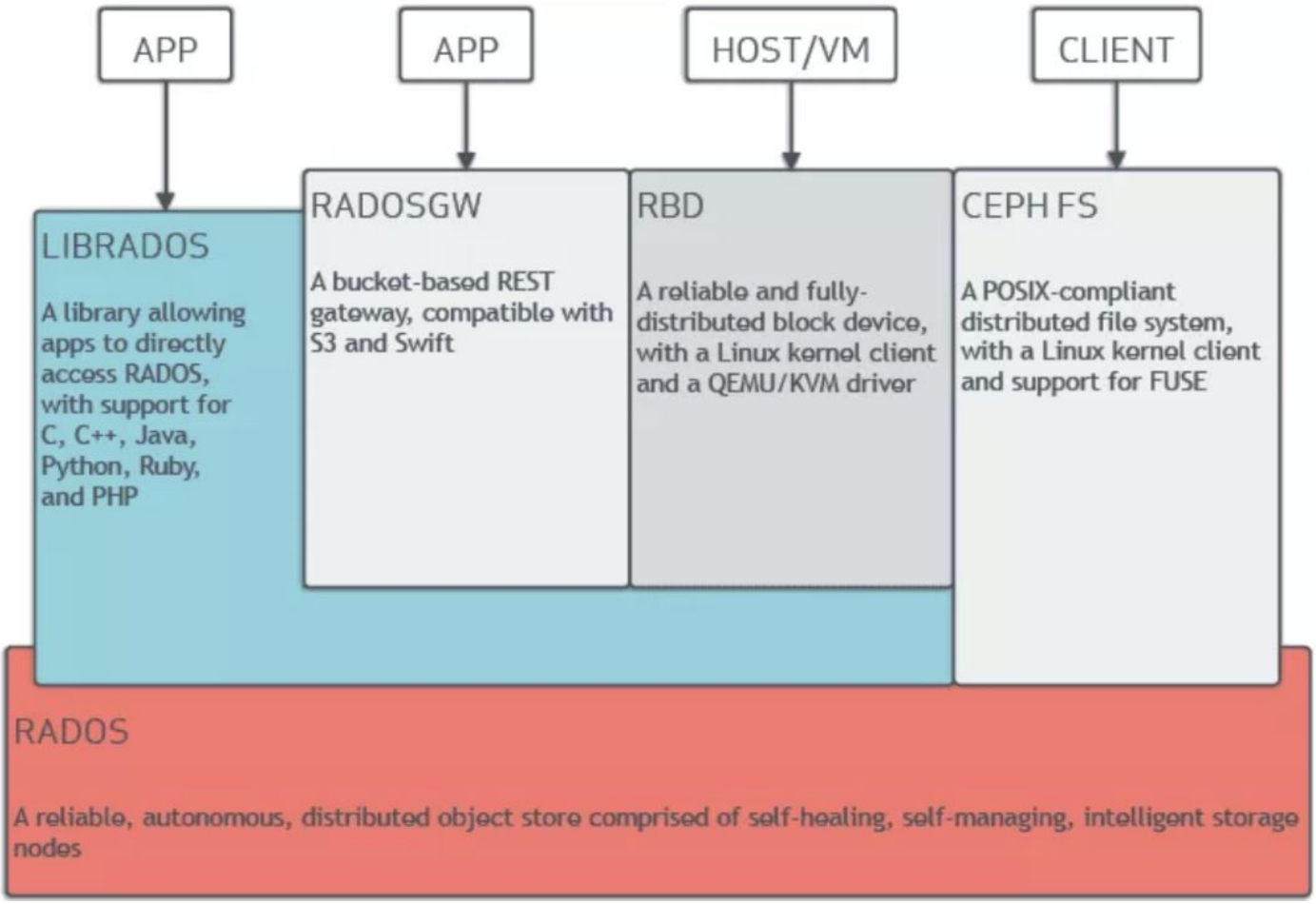
Client 根据中心节点返回的信息，向对应的 chunk server 直接发送数据读取的请求；

chunk server 返回数据。

在这种方案里，一般中心节点并不参与真正的数据读写，而是将文件 meta 信息返回给 Client 之后，即由 Client 与数据节点直接通信。其主要目的是降低中心节点的负载，防止其成为瓶颈。这种有中心节点的方案，在各种存储类系统中得到了广泛应用，因为中心节点易控制、功能强大。

无中心节点

以 ceph 为代表，每个节点都是自治的、自管理的，整个 ceph 集群只包含一类节点，如下图 (最下层红色的 RADOS 就是 ceph 定义的“同时包含 meta 数据和文件数据”的节点)。



无中心化的最大优点是解决了中心节点自身的瓶颈，这也就是 ceph 号称可以无限向上扩容的原因。但由 Client 直接和 Server 通信，那么 Client 必须要知道，当对某个文件进行操作时，它该访问集群中的哪个节点。ceph 提供了一个很强大的原创算法来解决这个问题——CRUSH 算法。

持久化

对于文件系统来说，持久化是根本，只要 Client 收到了 Server 保存成功的回应之后，数据就不应该丢失。这主要是通过多副本的方式来解决，但在分布式环境下，多副本有这几个问题要面对。

如何保证每个副本的数据是一致的?

如何分散副本，以使灾难发生时，不至于所有副本都被损坏?

怎么检测被损坏或数据过期的副本，以及如何处理?

该返回哪个副本给 Client?

如何保证每个副本的数据是一致的

同步写入是保证副本数据一致的最直接的办法。当 Client 写入一个文件的时候，Server 会等待所有副本都被成功写入，再返回给 Client。

这种方式简单、有保障，唯一的缺陷就是性能会受到影响。假设有 3 个副本，如果每个副本需要 N 秒，则可能会阻塞 Client 3N 秒的时间，有几种方式，可以对其进行优化：

并行写：由一个副本作为主副本，并行发送数据给其他副本；

链式写：几个副本组成一个链 (chain)，并不是等内容都接受到了再往后传播，而是像流一样，边接收上游传递过来的数据，一边传递给下游。

还有一种方式是采用 CAP 中所说的 W+R>N 的方式，比如 3 副本 (N=3) 的情况，W＝2，R＝2，即成功写入 2 个就认为成功，读的时候也要从 2 个副本中读。这种方式通过牺牲一定的读成本，来降低写成本，同时增加写入的可用性。这种方式在分布式文件系统中用地比较少。

如何分散副本，以使灾难发生时，不至于所有副本都被损坏

这主要避免的是某机房或某城市发生自然环境故障的情况，所以有一个副本应该分配地比较远。它的副作用是会带来这个副本的写入性能可能会有一定的下降，因为它离 Client 最远。所以如果在物理条件上无法保证够用的网络带宽的话，则读写副本的策略上需要做一定考虑。可以参考同步写入只写 2 副本、较远副本异步写入的方式，同时为了保证一致性，读取的时候又要注意一些，避免读取到异步写入副本的过时数据。

怎么检测被损坏或数据过期的副本，以及如何处理

如果有中心节点，则数据节点定期和中心节点进行通信，汇报自己的数据块的相关信息，中心节点将其与自己维护的信息进行对比。如果某个数据块的 checksum 不对，则表明该数据块被损坏了；如果某个数据块的 version 不对，则表明该数据块过期了。

如果没有中心节点，以 ceph 为例，它在自己的节点集群中维护了一个比较小的 monitor 集群，数据节点向这个 monitor 集群汇报自己的情况，由其来判定是否被损坏或过期。

当发现被损坏或过期副本，将它从 meta 信息中移除，再重新创建一份新的副本就好了，移除的副本在随后的回收机制中会被收回。

该返回哪个副本给 Client

这里的策略就比较多了，比如 round-robin、速度最快的节点、成功率最高的节点、CPU 资源最空闲的节点、甚至就固定选第一个作为主节点，也可以选择离自己最近的一个，这样对整体的操作完成时间会有一定节约。

伸缩性

存储节点的伸缩

当在集群中加入一台新的存储节点，则它主动向中心节点注册，提供自己的信息，当后续有创建文件或者给已有文件增加数据块的时候，中心节点就可以分配到这台新节点了，比较简单。但有一些问题需要考虑。

如何尽量使各存储节点的负载相对均衡?

怎样保证新加入的节点，不会因短期负载压力过大而崩塌?

如果需要数据迁移，那如何使其对业务层透明?

如何尽量使各存储节点的负载相对均衡

首先要有评价存储节点负载的指标。有多种方式，可以从磁盘空间使用率考虑，也可以从磁盘使用率 +CPU 使用情况 + 网络流量情况等做综合判断。一般来说，磁盘使用率是核心指标。

其次在分配新空间的时候，优先选择资源使用率小的存储节点；而对已存在的存储节点，如果负载已经过载、或者资源使用情况不均衡，则需要做数据迁移。

怎样保证新加入的节点，不会因短期负载压力过大而崩塌

当系统发现当前新加入了一台存储节点，显然它的资源使用率是最低的，那么所有的写流量都路由到这台存储节点来，那就可能造成这台新节点短期负载过大。因此，在资源分配的时候，需要有预热时间，在一个时间段内，缓慢地将写压力路由过来，直到达成新的均衡。

如果需要数据迁移，那如何使其对业务层透明？

在有中心节点的情况下，这个工作比较好做，中心节点就包办了——判断哪台存储节点压力较大，判断把哪些文件迁移到何处，更新自己的 meta 信息，迁移过程中的写入怎么办，发生重命名怎么办。无需上层应用来处理。

如果没有中心节点，那代价比较大，在系统的整体设计上，也是要考虑到这种情况，比如 ceph，它要采取逻辑位置和物理位置两层结构，对 Client 暴露的是逻辑层 (pool 和 place group)，这个在迁移过程中是不变的，而下层物理层数据块的移动，只是逻辑层所引用的物理块的地址发生了变化，在 Client 看来，逻辑块的位置并不会发生改变。

中心节点的伸缩

如果有中心节点，还要考虑它的伸缩性。由于中心节点作为控制中心，是主从模式，那么在伸缩性上就受到比较大的限制，是有上限的，不能超过单台物理机的规模。我们可以考虑各种手段，尽量地抬高这个上限。有几种方式可以考虑：

以大数据块的形式来存储文件——比如 HDFS 的数据块的大小是 64M，ceph 的的数据块的大小是 4M，都远远超过单机文件系统的 4k。它的意义在于大幅减少 meta data 的数量，使中心节点的单机内存就能够支持足够多的磁盘空间 meta 信息。

中心节点采取多级的方式——顶级中心节点只存储目录的 meta data，其指定某目录的文件去哪台次级总控节点去找，然后再通过该次级总控节点找到文件真正的存储节点；

中心节点共享存储设备——部署多台中心节点，但它们共享同一个存储外设 / 数据库，meta 信息都放在这里，中心节点自身是无状态的。这种模式下，中心节点的请求处理能力大为增强，但性能会受一定影响。iRODS 就是采用这种方式。

高可用性

中心节点的高可用

中心节点的高可用，不仅要保证自身应用的高可用，还得保证 meta data 的数据高可用。

meta data 的高可用主要是数据持久化，并且需要备份机制保证不丢。一般方法是增加一个从节点，主节点的数据实时同步到从节点上。也有采用共享磁盘，通过 raid1 的硬件资源来保障高可用。显然增加从节点的主备方式更易于部署。

meta data 的数据持久化策略有以下几种方式

> 直接保存到存储引擎上，一般是数据库。直接以文件形式保存到磁盘上，也不是不可以，但因为 meta 信息是结构化数据，这样相当于自己研发出一套小型数据库来，复杂化了。
>
> 保存日志数据到磁盘文件 (类似 MySQL 的 binlog 或 Redis 的 aof)，系统启动时在内存中重建成结果数据，提供服务。修改时先修改磁盘日志文件，然后更新内存数据。这种方式简单易用。

当前内存服务 + 日志文件持久化是主流方式。一是纯内存操作，效率很高，日志文件的写也是顺序写；二是不依赖外部组件，独立部署。

为了解决日志文件会随着时间增长越来越大的问题，以让系统能以尽快启动和恢复，需要辅助以内存快照的方式——定期将内存 dump 保存，只保留在 dump 时刻之后的日志文件。这样当恢复时，从最新一次的内存 dump 文件开始，找其对应的 checkpoint 之后的日志文件开始重播。

存储节点的高可用

在前面“持久化”章节，在保证数据副本不丢失的情况下，也就保证了其的高可用性。

性能优化和缓存一致性

这些年随着基础设施的发展，局域网内千兆甚至万兆的带宽已经比较普遍，以万兆计算，每秒传输大约 1250M 字节的数据，而 SATA 磁盘的读写速度这些年基本达到瓶颈，在 300-500M/s 附近，也就是纯读写的话，网络已经超过了磁盘的能力，不再是瓶颈了，像 NAS 网络磁盘这些年也开始普及起来。

但这并不代表，没有必要对读写进行优化，毕竟网络读写的速度还是远慢于内存的读写。常见的优化方法主要有：

内存中缓存文件内容；

预加载数据块，以避免客户端等待；

合并读写请求，也就是将单次请求做些积累，以批量方式发送给 Server 端。

缓存的使用在提高读写性能的同时，也会带来数据不一致的问题：

会出现更新丢失的现象。当多个 Client 在一个时间段内，先后写入同一个文件时，先写入的 Client 可能会丢失其写入内容，因为可能会被后写入的 Client 的内容覆盖掉；

数据可见性问题。Client 读取的是自己的缓存，在其过期之前，如果别的 Client 更新了文件内容，它是看不到的；也就是说，在同一时间，不同 Client 读取同一个文件，内容可能不一致。

这类问题有几种方法：

文件只读不改：一旦文件被 create 了，就只能读不能修改。这样 Client 端的缓存，就不存在不一致的问题；

通过锁：用锁的话还要考虑不同的粒度。写的时候是否允许其他 Client 读? 读的时候是否允许其他 Client 写? 这是在性能和一致性之间的权衡，作为文件系统来说，由于对业务并没有约束性，所以要做出合理的权衡，比较困难，因此最好是提供不同粒度的锁，由业务端来选择。但这样的副作用是，业务端的使用成本抬高了。

安全性

由于分布式文件存储系统，肯定是一个多客户端使用、多租户的一个产品，而它又存储了可能是很重要的信息，所以安全性是它的重要部分。

主流文件系统的权限模型有以下这么几种。

DAC: 全称是 Discretionary Access Control，就是我们熟悉的 Unix 类权限框架，以 user-group-privilege 为三级体系，其中 user 就是 owner，group 包括 owner 所在 group 和非 owner 所在的 group、privilege 有 read、write 和 execute。这套体系主要是以 owner 为出发点，owner 允许谁对哪些文件具有什么样的权限。

MAC: 全称是 Mandatory Access Control，它是从资源的机密程度来划分。比如分为“普通”、“机密”、“绝密”这三层，每个用户可能对应不同的机密阅读权限。这种权限体系起源于安全机构或军队的系统中，会比较常见。它的权限是由管理员来控制和设定的。Linux 中的 SELinux 就是 MAC 的一种实现，为了弥补 DAC 的缺陷和安全风险而提供出来。关于 SELinux 所解决的问题可以参考 What is SELinux?

RBAC: 全称是 Role Based Access Control，是基于角色 (role) 建立的权限体系。角色拥有什么样的资源权限，用户归到哪个角色，这对应企业 / 公司的组织机构非常合适。RBAC 也可以具体化，就演变成 DAC 或 MAC 的权限模型。

市面上的分布式文件系统有不同的选择，像 ceph 就提供了类似 DAC 但又略有区别的权限体系，Hadoop 自身就是依赖于操作系统的权限框架，同时其生态圈内有 Apache Sentry 提供了基于 RBAC 的权限体系来做补充。

其他

空间分配

有连续空间和链表空间两种。连续空间的优势是读写快，按顺序即可，劣势是造成磁盘碎片，更麻烦的是，随着连续的大块磁盘空间被分配满而必须寻找空洞时，连续分配需要提前知道待写入文件的大小，以便找到合适大小的空间，而待写入文件的大小，往往又是无法提前知道的 (比如可编辑的 word 文档，它的内容可以随时增大)；

而链表空间的优势是磁盘碎片很少，劣势是读写很慢，尤其是随机读，要从链表首个文件块一个一个地往下找。

为了解决这个问题，出现了索引表——把文件和数据块的对应关系也保存一份，存在索引节点中 (一般称为 i 节点)，操作系统会将 i 节点加载到内存，从而程序随机寻找数据块时，在内存中就可以完成了。通过这种方式来解决磁盘链表的劣势，如果索引节点的内容太大，导致内存无法加载，还有可能形成多级索引结构。

文件删除

实时删除还是延时删除? 实时删除的优势是可以快速释放磁盘空间；延时删除只是在删除动作执行的时候，置个标识位，后续在某个时间点再来批量删除，它的优势是文件仍然可以阶段性地保留，最大程度地避免了误删除，缺点是磁盘空间仍然被占着。在分布式文件系统中，磁盘空间都是比较充裕的资源，因此几乎都采用逻辑删除，以对数据可以进行恢复，同时在一段时间之后 (可能是 2 天或 3 天，这参数一般都可配置)，再对被删除的资源进行回收。

怎么回收被删除或无用的数据? 可以从文件的 meta 信息出发——如果 meta 信息的“文件 - 数据块”映射表中包含了某个数据块，则它就是有用的；如果不包含，则表明该数据块已经是无效的了。所以，删除文件，其实是删除 meta 中的“文件 - 数据块”映射信息 (如果要保留一段时间，则是把这映射信息移到另外一个地方去)。

面向小文件的分布式文件系统

有很多这样的场景，比如电商——那么多的商品图片、个人头像，比如社交网站——那么多的照片，它们具有的特性，可以简单归纳下：

每个文件都不大；

数量特别巨大；

读多写少；

不会修改。

针对这种业务场景，主流的实现方式是仍然是以大数据块的形式存储，小文件以逻辑存储的方式存在，即文件 meta 信息记录其是在哪个大数据块上，以及在该数据块上的 offset 和 length 是多少，形成一个逻辑上的独立文件。这样既复用了大数据块系统的优势和技术积累，又减少了 meta 信息。

文件指纹和去重

文件指纹就是根据文件内容，经过算法，计算出文件的唯一标识。如果两个文件的指纹相同，则文件内容相同。在使用网络云盘的时候，发现有时候上传文件非常地快，就是文件指纹发挥作用。云盘服务商通过判断该文件的指纹，发现之前已经有人上传过了，则不需要真的上传该文件，只要增加一个引用即可。在文件系统中，通过文件指纹可以用来去重、也可以用来判断文件内容是否损坏、或者对比文件副本内容是否一致，是一个基础组件。

文件指纹的算法也比较多，有熟悉的 md5、sha256、也有 google 专门针对文本领域的 simhash 和 minhash 等。

总结

分布式文件系统内容庞杂，要考虑的问题远不止上面所说的这些，其具体实现也更为复杂。本文只是尽量从分布式文件系统所要考虑的问题出发，给予一个简要的分析和设计，如果将来遇到类似的场景需要解决，可以想到“有这种解决方案”，然后再来深入研究。

同时，市面上也是存在多种分布式文件系统的形态，下面就是有研究小组曾经对常见的几种分布式文件系统的设计比较。

几种分布式文件系统的比较

	HDFS	iRODS	Ceph	GlusterFS	Lustre
Architecture	Centralized	Centralized	Distributed	Decentralized	Centralized
Naming	Index	Database	CRUSH	EHA	Index
API	CLI, FUSE REST, API	CLI, FUSE API	FUSE, mount REST	FUSE, mount	FUSE
Fault detection	Fully connect.	P2P	Fully connect.	Detected	Manually
System availability	No failover	No failover	High	High	Failover
Data availability	Replication	Replication	Replication	RAID-like	No
Placement strategy	Auto	Manual	Auto	Manual	No
Replication	Async.	Sync.	Sync.	Sync.	RAID-like
Cache consistency	WORM, lease	Lock	Lock	No	Lock
Load balancing	Auto	Manual	Manual	Manual	No

从这里也可以看到，选择其实很多，并不是 GFS 论文中的方式就是最好的。在不同的业务场景中，也可以有更多的选择策略。