分布式存储系统，通俗地讲就是要把文件存储到多个机器中。那么需要解决的第一个问题便是这些文件如何知道该保存到哪台服务器中。这里有两个思路，其中一个是设计一个控制服务器，由这个控制服务器负责统一调度，客户端请求存储一个文件时，首先与控制服务器交互，控制服务器返回需要保存到服务器的地址，读取文件时也需要与控制服务器交互，获取存储位置信息，其中HDFS、GFS等分布式存储使用此种技术，**namenode**就类似于控制服务器角色。另外一个思路是，不需要控制服务器，客户端自己计算需要存储到哪里，最简单的方式是直接取hash，比如有8台存储服务器，只需要把文件内容或者文件名取hash模8即可计算出应该存储到哪台存储服务器。但有个问题是，当服务器数量增减时，hash就失效了，几乎需要重排迁移所有数据，根本没有办法实现水平扩展，这在分布式系统中是无法忍受的。为了避免出现这种情况，引入了一致性hash算法，又称为环哈希，关于该算法可参考[深入云存储系统Swift核心组件：Ring实现原理剖析 - 牛皮糖NewPtone - 博客园](https://link.zhihu.com/?target=http%3A//www.cnblogs.com/yuxc/archive/2012/06/22/2558312.html" \t "_blank)，其中OpenStack Swift、华为FusionStorage就是使用的该方法。除了环hash，当然还有其他的类hash算法，比如**CRUSH算法**，关于CRUSH算法介绍可参考[大话Ceph--CRUSH那点事儿](https://link.zhihu.com/?target=http%3A//www.xuxiaopang.com/2016/11/08/easy-ceph-CRUSH/" \t "_blank)，其中开源分布式存储系统Ceph就是使用的该方法。需要注意的是虽然基于hash的文件分布映射方法不需要控制节点计算需要存储的位置，但仍然需要控制服务器保存一些集群元数据，比如集群的成员信息、映射规则、监控等等，如Ceph的mon服务。如果只有一个控制服务，则存在单点故障，挂掉了就会导致服务不可用。为了避免单点故障，具备高可用特点，必然需要同时启动多个控制服务，有多个控制服务就必须区分谁是leader，谁是slave，因此需要**分布式一致性来协调**选主，可以基于现有的分布式协调系统实现，如**Zookeeper、Etcd服务**等，也可以直接基于**Paxos、Raft**算法实现，Paxos相对复杂，Google的分布式锁服务Chubby就是基于**Paxos**实现的，Raft相对比较简单容易理解，可参看[Raft-Understandable Distributed Consensus](https://link.zhihu.com/?target=http%3A//thesecretlivesofdata.com/raft/" \t "_blank)动态展示了该算法的执行流程，更多关于Raft的信息可参看官方文档主页[Raft Consensus Algorithm](https://link.zhihu.com/?target=https%3A//raft.github.io/" \t "_blank)。

文件直接映射到物理主机或者物理硬盘，粒度太粗略，容易导致数据分布不均匀。如果踢掉一台服务器或者一块硬盘，需要把这台服务器的数据迁移到重映射的另一台主机，迁移数据的IO都集中在这两台主机之间，其它主机帮不上忙。于是引入了**虚拟主机概念**，OpenStack Swift中叫做partition以及**Ceph中PG**等都是类似的概念。原理就是在物理主机上面加一层逻辑主机，比如有8台物理主机，可以创建128个虚拟主机，然后把这8台物理主机映射到这128台逻辑主机上，这样相当于每一台主机都虚拟成16台虚拟主机，当然实际上不一定是按照平均分，可以根据磁盘容量映射，磁盘空间大可以映射较多的虚拟主机。当然虚拟主机数量通常都会设置成2的幂，这样很多计算都可以使用位运算进行优化，比如取模运算等。这样文件块会先根据虚拟主机计算存储位置，然后再从表中查找虚拟主机映射的物理主机，文件块分布更均匀，当踢掉一台主机时会重映射到多台主机中，数据迁移效率提升。

解决了文件**如何分布的问题**，自然会遇到的问题是如果文件很大怎么办，可能在一台服务器根本存不下，即使存下了，也会导致各个服务器的磁盘利用率不均衡，甚至可能出现大量存储碎片。于是我们自然想到的是把文件分块，然后基于块存储，比如按照64MB大小分块，如果存储一个2GB的文件，则需要把文件分割成32个块，然后逐块存储，存储位置仍然使用前面提到的hash算法。分块是存储密度更大、更紧凑，几乎所有的分布式存储系统都会使用**分块技术**。

接下来，将考虑存储系统的**数据可靠性**（数据不丢）以及**可用性**（数据可访问）问题，如果其中一个服务器坏了怎么办？显然可能出现一个文件的某些块不能访问了，文件读取失败。为了解决这个问题，最容易想到的方法是使用**冗余技术**，即每一个块，我都存储多份，并分布到不同的服务器中，这样即使其中一个服务器宕机了，也能从其他服务器中读取块，这个和**RAID 1**技术原理是一样的。存储多少份呢，这个需要权衡成本以及数据可靠性要求，通常来说存储三份就够了。有人会说，存储三份，相当于使用了三倍的存储空间，这样存储资源是不是有点太浪费了，而又不想牺牲数据可靠性。我们学习算法时经常使用时间换空间的思想，计算换存储，这个仍然可以从RAID实现中获取灵感，以**RAID 5**为例，通过**奇偶校验**恢复数据，存储利用率为(n-1)/n，相比RAID 1的1/2提高了存储利用率，并且具有RAID 1一样的可靠性，但需要耗费CPU计算奇偶位。奇偶校验只能缺一位，自然可以想到进一步泛化，于是引入**纠删码技术，**原理其实就是类似解线性方程，关于纠删码技术介绍可以参考[Erasure Code - EC纠删码原理](https://link.zhihu.com/?target=https%3A//blog.csdn.net/shelldon/article/details/54144730)。几乎所有的分布式存储系统都使用了冗余副本技术，大多数都会支持纠删码，比如Ceph、Swift。

无论使用纯副本技术还是结合纠删码，必然还是需要把一个块复制多份存储，写入多份，这里假设副本数为3份。这些副本如何写入呢？当拿到三个副本的位置后，客户端可以同时写入三个副本，这种方式称为**直接复制(direct replication)**，这样的问题是客户端会同时占用3倍的业务网络带宽，吞吐量也只有1/3，glusterfs采用的是这种复制策略。另一种方式是客户端只选择其中一个主节点写入数据，当写完第一个节点的数据后，由第一个节点复制到第二个节点，再由第二个节点复制到第三个节点，以此类推直到写完所有的副本，这种方式称为**链式复制(chain replication)**，Ceph、HDFS都是采用的该种策略，这样由于客户端其实只是写了一份数据，不占用额外的业务网络，而存储节点之间的复制可以是一个专门的存储网，不影响业务网络。关于chain replication可以参考论文[Chain Replication for Supporting High Throughput and Availability](https://link.zhihu.com/?target=http%3A//www.cs.cornell.edu/home/rvr/papers/OSDI04.pdf" \t "_blank)。

写入多份数据，如何保证这些副本数据都是一样的，如何保证三个数据同步呢，万一哪台服务器挂了写不进去怎么办。于是引入了**一致性策略**。最简单的方法，就是等所有的副本都完成时才返回结果，这样保证写入的三个副本肯定没有问题，这就是强一致性，其中Ceph就是使用的**强一致性模型**，强一致性能够保证多副本完全一致，并且不会读取脏数据，但是性能不好，万一有一台服务器巨慢则会拖垮整个集群，典型的木桶效应，因此强一致性天生难以支持跨区域部署，因为跨区域的远端时延太长了，导致存储系统性能低。为了避免这种情况，我们可以适当放宽条件，即只要保证一半以上的服务器写入成功即返回，这样即使其中有少数服务器拖后腿也没有关系，不用等，让他自个慢慢同步，最终一致即可。这就是典型的**最终一致性模型**，OpenStack Swift即采用该种策略，这种模型能够提高读写性能，但可能读取脏数据，比如刚好读到还没有来得及同步的服务器的数据块。事实上高性能和强一致性是两者不可兼得的，这就是著名的**CAP理论**，这里的C代表一致性，A代表可用性（在一定时间内，用户的请求都会得到正确的应答），P代表分区容错。正常情况下，存储系统的所有节点都是互通的，处在一个网络连通区域中，如果有些节点之间不连通了（节点挂了或者网络故障），这就相当于把一个网络连通区域割裂了几个区域，彼此不能通信了，因此叫做分区。分布式存储系统要系统出现分区时数据不丢（可靠性），数据可访问（可用性），避免脑裂，因此P是100%需要满足的，否则稍微一个网络抖动，数据就损坏了。剩下的就是C和P之间的权衡，这个就看你要设计成什么存储系统了，如果一致性不那么重要，比如对象存储，上传了一个新文件，即使马上读不到数据也无所谓，但是可能需要支持大规模的对象写入，因此更关注A，设计为**AP存储系统**。而对于一些实时性要求高的系统，必须保证写入后数据一定能够读到正确的数据（而不是脏数据），就必须牺牲吞吐量，因此设计为**CP存储系统**。

为了节省存储空间，可能会用到**压缩技术**，压缩大家都很熟悉了，这里不多介绍。

如果是一个海量分布式存储系统，尤其是提供公有云服务，比如网盘服务，肯定会有用户上传一模一样的文件，为了节省成本，自然想到避免存储重复的文件，这就是**重删技术（**[Data deduplication](https://link.zhihu.com/?target=https%3A//en.wikipedia.org/wiki/Data_deduplication)），可参考[int32bit：百度云的「极速秒传」使用的是什么技术？](https://www.zhihu.com/question/21275365/answer/288821134)，简单理解就是客户端上传文件时，先在本地计算下hash指纹，然后上传到服务器比对，如果指纹一样，说明文件已经存在，此时不需要上传文件内容，直接链接下即可，不仅节省了存储空间(比压缩更省），还节省了上传时间，实现秒传。我了解的Fusion Storage是实现了重删技术，OpenStack Swift、Ceph貌似都没有。

另一个问题是，如果集群彻底瘫了，数据就彻底没了，这可不能忍。为了解决这个问题，你自然会想到使用复制手段，即**备份技术，**把文件复制存储到其它廉价存储服务器中，比如S3。当用户执行save操作时，复制这个文件并重命名为xxx-20180312233020(时间戳），这样非常容易就能恢复到备份的任意版本，由于每次都要拷贝整个文件，因此称为全量备份(full backup)。每次都复制显然耗时耗空间，自然想到只复制上一次备份后改变的内容，这样就可以节省存储空间，即增量(incremental backup)备份。注意，备份一定要拷贝到其它存储系统，如果仅仅是拷贝到当前存储系统，不叫备份，只能叫副本，集群瘫了，数据仍然不能恢复。

以上备份技术需要用户自己手动执行，如果没有实时备份，集群突然挂了，数据还是会丢。因此需要采取**容灾策略**，其中一个容灾策略就是异地同步技术，或者叫做复制技术(geo-replication/mirror)，这个类似于mysql的主从同步，即在异地建立一个一模一样的集群，这个集群正常情况下不向用户提供存储服务，仅仅同步本地的集群数据，当本地的集群挂了，能够自动切换到异地集群，服务依然可用。注意这个和副本之间完全同步不一样，复制技术通常采用异步策略，基于操作日志replay，mysql使用binlog，ceph使用journal日志。ceph的rbd mirror就是采用的此种技术，关于rbd mirro介绍参考[Ceph Jewel Preview: Ceph RBD mirroring](https://link.zhihu.com/?target=https%3A//www.sebastien-han.fr/blog/2016/03/28/ceph-jewel-preview-ceph-rbd-mirroring/" \t "_blank)。

分布式存储系统不仅需要大量的磁盘IO，还需要网络IO，然而网络带宽必然是有限的，有限的资源就必然需要合理的分配。如果某个用户持续不断的读写，抢占大量的IO带宽，则必然导致其它用户性能下降，甚至出现饿死状况。因此需要公平控制用户的IO资源使用情况，于是引入了**QoS**。QoS的目标是要实现系统IOPS的调度分配，对单一客户端的IOPS、IO带宽进行限制，不能让某个客户端独占了整个系统的IOPS。QoS限制客户端能够使用的最大值称为**limit**，即上限。注意，QoS不是仅仅有limit就够了，为了避免某些客户端迟迟得不到IO调度而被饿死，QoS还包含一个下限控制，称为**reservation**。上限limit容易理解，这个下限就容易弄混，因为有人会想，如果我的这个客户端就是没有IO需求，那它的IOPS就是0，这个下限有什么意义。其实这个下限是一个承诺，当客户端有大于reservation的IOPS请求时，系统能够保证给予不小于reservation的IOPS，如果客户端本身就不需要大于的reservation的值，那自然不需要分配其IOPS。因此，这个reservation翻译为预留更合适，系统调度IOPS时，会优先满足reservation的值，多余的再根据实际情况分配。当然在同时满足了请求的上限和下限下，不同的请求IOPS仍然不同，优先级也有可能不一样，因此还需要一个控制指标，称为分配比例。系统会根据权重去分配，能者多得，这样才能真正发挥资源的最大价值。关于QoS的实现，有两种思路，一种是直接使用Linux系统的**cgroup**实现，QEMU对虚拟机的磁盘QoS控制就是使用的该原理，这种方式不依赖于存储系统本身的实现。另一种就是QoS由存储系统自己，对某个节点实现QoS可参考VMware在OSDI发表的一篇论文[mClock: handling throughput variability for hypervisor IO scheduling](https://link.zhihu.com/?target=https%3A//labs.vmware.com/download/122/" \t "_blank)，而dmClock即分布式的mClock，实现分布式系统的QoS控制，目前是作为Ceph的一个子项目，[ceph/dmclock](https://link.zhihu.com/?target=https%3A//github.com/ceph/dmclock" \t "_blank)，关于dmClock的介绍可参考[虚拟化I/O QoS mClock算法介绍](https://link.zhihu.com/?target=https%3A//blog.csdn.net/u011364612/article/details/53608278)。

为了继续下文内容，需要先介绍下几个不同的存储服务（接口）：

* 块存储：即提供裸的块设备服务，裸设备什么都没有，需要用户自己创建分区、创建文件系统、挂载到操作系统才能用，挂一个块存储设备到操作系统，相当于插一个新U盘。只实现了read、write、ioctl等接口。SAN、LVM、Ceph RBD、OpenStack Cinder等都属于块存储服务。
* 文件存储：可以简单理解为分布式文件系统，通常实现了POSIX接口，不需要安装文件系统，直接像NFS一样挂载到操作系统就能用。典型的文件存储如NAS、HDFS、CephFS、GlusterFS、OpenStack Manila等。
* 对象存储：提供Web存储服务，通过HTTP协议访问，只需要Web浏览器即可使用，不需要挂载到本地操作系统，实现的接口如GET、POST、DELETE等，典型的对象存储如百度网盘、S3、OpenStack Swift、Ceph RGW等。

有些存储系统只提供以上某种接口，有些存储系统则能够同时支持以上三种存储服务接口，比如Ceph。

块存储最典型的使用场景是作为虚拟机的磁盘。虚拟机通常需要申请几十GB到几TB的虚拟硬盘，但虚拟机实际上并不是一下就真的会用那么多的存储，如果申请多少就分配多少，显然会造成磁盘空间利用率不高，不能实现超售。因此引入了精简配置(thin provision），这个其实不难理解，就是类似于Linux的稀疏文件(sparse file)，关于Linux稀疏文件介绍可参考[int32bit: sparse文件处理与传输](https://link.zhihu.com/?target=http%3A//int32bit.me/2016/06/01/Sparse%25E6%2596%2587%25E4%25BB%25B6%25E5%25A4%2584%25E7%2590%2586%25E5%2592%258C%25E4%25BC%25A0%25E8%25BE%2593/" \t "_blank)，简单理解就是当申请一个20GB的虚拟磁盘时并不会立即真正从硬盘中分配空间，而是用多少分配多少，因此可以创建总大小远大于实际物理磁盘空间大小的磁盘，实现磁盘超售。但需要注意控制超售率，避免虚拟磁盘写满时，物理磁盘空间不足导致数据损坏。

分布式存储和本地存储一样，自然需要有版本控制，用户可以随时回滚到任意一个时间点的存储状态，或者基于某个时间点的版本修改，类似于git的checkout以及branch操作。当然你可以使用备份技术实现，只是太耗时耗空间。为了实现这个功能，引入**快照技术(snapshot)，**快照的功能形如其名，就是把当前的存储状态拍个照保存下来，以后可以随时回滚到某个快照时刻的状态。可以在快照的基础上，创建一个一样的磁盘卷，称为克隆（clone），注意和复制（copy）的不同，创建的新卷并没有拷贝源数据，也没有分配任何存储空间，只是画了一个指针指向原来的快照卷，秒级完成。OpenStack使用Ceph存储后端能够秒级创建虚拟机就是这个原因。当用户读取数据时，如果自己的卷没有找到，则需要在它parent中去找，如何写入则取决于采用何种快照方式。快照的实现有两种方式，一种是COW（Copy On Write，写时拷贝），快照是只读的，不允许修改，当用户有新的数据写入克隆的磁盘时，会首先从快照中拷贝一份数据写到另一个分配的空间，然后把修改的数据覆盖新分配的空间，相当于两次写操作。当磁盘创建快照，克隆，再创建快照，再克隆，形成一条很长的克隆链，当读取数据时，需要从当前卷开始查找，找不到查找其parent卷，直到查找到base镜像，写入数据也类似，当当前卷的块没有时，需要从其parent中依次查找，然后拷贝到自己的卷中，再写入新的数据。显然，当链越来越长时，卷的读写性能越来越差，因此需要控制链的长度。另一种快照技术是ROW(Redirect On Write)，这个和COW不同，当有新数据写入时，直接写入一个新分配的区，然后修改卷的指针指向新的区地址，只有一次写操作，关于COW与ROW介绍可以参考[ROW/COW 快照技术原理解析](https://link.zhihu.com/?target=https%3A//blog.csdn.net/jmilk/article/details/65629391)。

虚拟机可能同时挂了多个虚拟硬盘，需要对这个虚拟机打快照，此时需要保证所有的卷的快照时一致的，而不能出现各个卷快照点不一致。于是引入了一致性快照技术(Consistency Snapshot Group)，参考[存储专栏：深度解读高端存储的快照技术\_存储在线](https://link.zhihu.com/?target=http%3A//www.dostor.com/article/2013-09-04/7686494_2.shtml)。

以上，花了两个晚上粗略总结了分布式存储系统的一些关键技术，参考了很多博客文章，在原文中都有标明。需要强调的是真正实现分布式存储系统的技术远不止这些，这里仅仅作为抛砖引玉。其它的技术，诸如副本不一致时如何同步，新增或者减少节点时数据如何迁移，引入缓存提高性能等等，待有时间再补充。