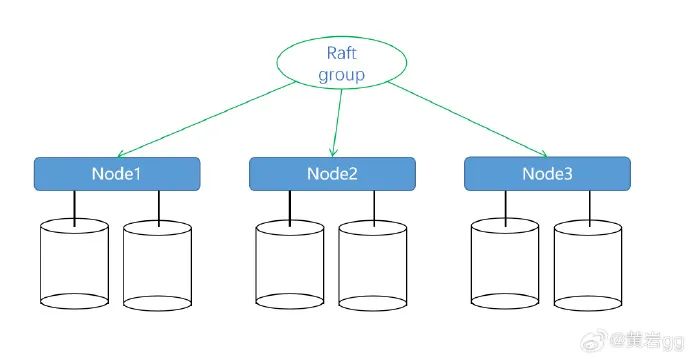
什么是分布式存储？先给他下个定义，满足下面四个特征的存储系统，就是分布式存储：

* 【软件+服务器 => 专业存储】分布式存储是通过网络连接起来的多个服务器，运行特殊软件，构成像磁盘阵列那样的专业存储系统。
* 【用软件整合硬盘 => 存储池】不同服务器节点的软件之间协同工作，把服务器上的硬盘都整合起来，成为一个大的存储池。
* 【容忍故障】这个存储系统可以容忍硬盘故障，也可以容忍服务器故障。
* 【可扩展】存储池容量不足，或者性能不足，可以通过增加硬盘，或者增加服务器，来扩展存储池的容量，或者扩展性能。

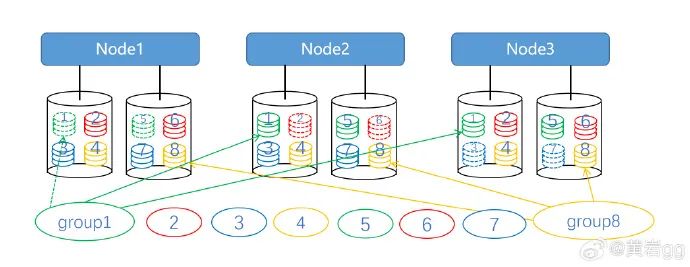
软件怎么整合硬盘呢？怎么容忍故障呢？下图是一个容易想到的方法，三个服务器节点构成一个组，把数据在这三个节点之间复制三份，用Raft这类共识协议来保证一致性。当其中一个节点故障了，换一个新节点进来；一个硬盘故障了，就换一个硬盘进来。这样，上面4个特征的前3个都可以满足。



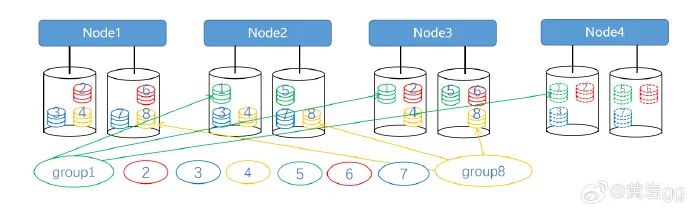
唯独无法满足第4个特征【可扩展】，怎么办呢？

实际上，按照硬盘来构成Raft组，也无法扩展。解决扩展性问题的方法是：把硬盘分为小盘，用小盘来构成多个Raft组。每3个小盘构成的一个组，在Ceph中被叫做PG（Placement Group），放置组。在 zStorage 中借用了Ceph的概念，也被称为PG。zStorage与Ceph的不同在于：zStorage用了Raft，Ceph使用了自定义的共识协议。

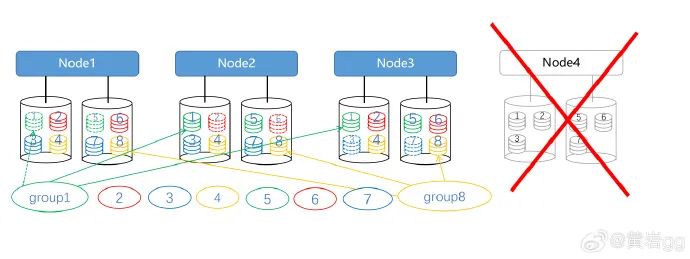
如下图所示，每个硬盘被划分4个小盘，3个节点，一共6个硬盘，24个小盘。每3个小盘构成1个Raft组，这24个小盘构成8个Raft组。



如下图所示，增加一个服务器节点之后，小盘总数量不变，还是24个。但是以前3个节点中的部分小盘中的数据，被迁移到了新的服务器节点上，这样每个硬盘中只有3个小盘了，那么每个小盘的空间也就自动扩大了。就是通过这种方式，实现自动扩容的。



如下图所示，4个节点的情况下，如果一个节点故障了，那么再把这个故障节点上的小盘的数据，迁移到剩下的3个节点上，当然，前提是剩余存储空间足够。这样迁移之后，数据还能保持三副本。

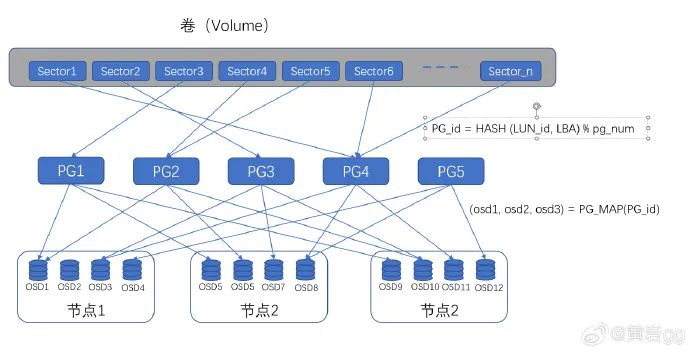


分布式存储系统如何把数据分散（Sharding）存到不同的Raft组中（也叫做PG）？

分布式存储系统对外提供的接口有三种：对象接口、块接口、文件接口。下面以块接口为例，讲一下如何分散（Sharding）存数据。

所谓块接口，在客户端看来就是，它里面有多个虚拟的硬盘，存储术语叫做LUN。其实，每个虚拟的硬盘，就是从这个高可靠的、可扩展的存储池中，划分出的一部分存储空间。也就是，客户端看到虚拟硬盘的所有磁道（sectors），在存储池中都对应一个Raft组。每个Raft组里面都承载了很多个磁道（sector）。

这些Sector与Raft组怎么对应呢？如下图所示，每个虚拟的硬盘，有一个Lun\_ID；每个磁道，有一个LBA地址。把LUN\_id和LBA地址这两个值，交给hash算法，可以计算出一个PG\_id，用这个PG\_id就可以寻址唯一Raft组了（Raft组也叫做PG）。



如果读完了以上这三条，那么恭喜，您已经掌握了90%分布式存储系统基本原理。

# **[海量小文件存储与Ceph实践](https://www.cnblogs.com/wuhuiyuan/p/ceph-small-file-compound-storage.html" \o "发布于 2015-08-15 13:59)**

　海量小文件存储（简称LOSF，lots of small files）出现后，就一直是业界的难题，众多博文(如[[1]](http://blog.csdn.net/liuaigui/article/details/9981135" \t "https://www.cnblogs.com/wuhuiyuan/p/_blank))对此问题进行了阐述与分析，许多互联网公司也针对自己的具体场景研发了自己的存储方案（如taobao开源的[TFS](http://code.taobao.org/p/tfs/wiki/index/" \t "https://www.cnblogs.com/wuhuiyuan/p/_blank)，facebook自主研发的[Haystack](https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Beaver.pdf" \t "https://www.cnblogs.com/wuhuiyuan/p/_blank)），还有一些公司在现有开源项目(如hbase，fastdfs，mfs等)基础上做针对性改造优化以满足业务存储需求；

一.　　通过对若干分布式存储系统的调研、测试与使用，与其它分布式系统相比，海量小文件存储更侧重于解决两个问题：

　　1. 海量小文件的元数据信息组织与管理: 对于百亿量级的数据，每个文件元信息按100B计算，元信息总数据量为1TB，远超过目前单机服务器内存大小；若使用本地持久化设备存储，须高效满足每次文件存取请求的元数据查询寻址(对于上层有cdn的业务场景，可能不存在明显的数据热点)，为了避免单点，还要有备用元数据节点；同时，单组元数据服务器也成为整个集群规模扩展的瓶颈；或者使用独立的存储集群存储管理元数据信息，当数据存储节点的状态发生变更时，应该及时通知相应元数据信息进行变更；

　　对此问题，tfs/fastdfs设计时，就在文件名中包含了部分元数据信息，减小了元数据规模，元数据节点只负责管理粒度更大的分片结构信息(如tfs的block)；商用分布式文件系统龙存，通过升级优化硬件，使用分布式元数据架构——多组(每组2台)IO性能更好的ssd服务器——存储集群的元数据信息，满足单次io元数据查询的同时，也实现了元数据存储的扩展性；Haystack Directory模块提供了图片逻辑卷到物理卷轴的映射存储与查询功能，使用Replicated Database存储，并通过cache集群来降低延时提高并发，其对外提供的读qps在百万量级；

1. 本地磁盘文件的存储与管理(本地存储引擎)：对于常见的linux文件系统，读取一个文件通常需要三次磁盘IO(读取目录元数据到内存，把文件的inode节点装载到内存，最后读取实际的文件内容)；按目前主流2TB~4TB的sata盘，可存储2kw~4kw个100KB大小的文件，由于文件数太多，无法将所有目录及文件的inode信息缓存到内存，很难实现每个图片读取只需要一次磁盘IO的理想状态，而长尾现象使得热点缓存无明显效果；当请求寻址到具体的一块磁盘，如何减少文件存取的io次数，高效地响应请求(尤其是读)已成为必须解决的另一问题；

对此问题，有些系统(如tfs，Haystack)采用了小文件合并存储+索引文件的优化方案，此方案有许多益处：

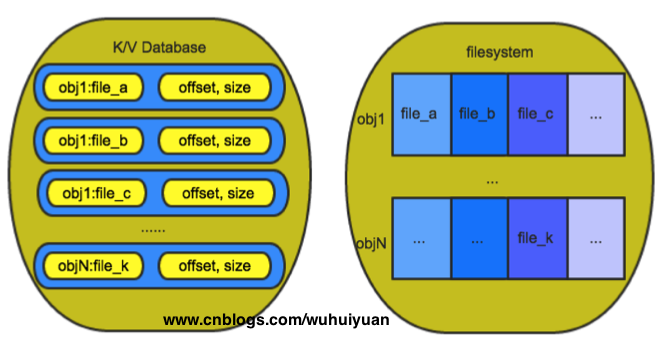
1. 合并后的合并大文件通常在64MB，甚至更大，单盘所存储的合并大文件数量远小于原小文件的数量，其inode等信息可以全部被cache到内存，减少了一次不必要的磁盘IO；
2. 索引文件通常数据量(通常只存储小文件所在的合并文件，及offset和size等关键信息)很小，可以全部加载到内存中，读取时先访问内存索引数据，再根据合并文件、offset和size访问实际文件数据，实现了一次磁盘IO的目的；

c.单个小文件独立存储时，文件系统存储了其guid、属主、大小、创建日期、访问日期、访问权限及其它结构信息，有些信息可能不是业务所必需的，在合并存储时，可根据实际需要对文件元数据信息裁剪后在做合并，减少空间占用。除了合并方法外，还可以使用IO性能更好的SSD等设备，来实现高效响应本地io请求的目标。

　　当然，在合并存储优化方案中，删除或修改文件操作可能无法立即回收存储空间，对于存在大量删除修改的业务场景，需要再做相应的考量。

二.　　[Ceph](http://ceph.com/" \t "https://www.cnblogs.com/wuhuiyuan/p/_blank)是近年越来越被广泛使用的分布式存储系统，其重要的创新之处是基于[CRUSH](http://ceph.com/papers/weil-crush-sc06.pdf" \t "https://www.cnblogs.com/wuhuiyuan/p/_blank)算法的计算寻址，真正的分布式架构、无中心查询节点，理论上无扩展上限(更详细ceph介绍见网上相关文章)；Ceph的基础组件RADOS本身是对象存储系统，将其用于海量小文件存储时，CRUSH算法直接解决了上面提到的第一个问题；不过Ceph OSD目前的存储引擎([Filestore](http://www.wzxue.com/ceph-filestore/" \t "https://www.cnblogs.com/wuhuiyuan/p/_blank),[KeyValuestore](http://www.wzxue.com/ceph-keyvaluestore/" \t "https://www.cnblogs.com/wuhuiyuan/p/_blank))对于上面描述的海量小文件第二个问题尚不能很好地解决；ceph社区曾对此问题做过描述并提出了基于rgw的一种[方案](https://wiki.ceph.com/Planning/Blueprints/Giant/rgw:_compound_object_(phase_1)" \t "https://www.cnblogs.com/wuhuiyuan/p/_blank)(实际上，在实现本文所述方案过程中，发现了社区上的方案)，不过在最新代码中，一直未能找到方案的实现；

　　我们在Filestore存储引擎基础上对小文件存储设计了优化方案并进行实现，方案主要思路如下：将若干小文件合并存储在RADOS系统的一个对象(object)中，<小文件的名字、小文件在对象中的offset及小文件size>组成kv对，作为相应对象的扩展属性(或者omap，本文以扩展属性表述，ceph都使用kv数据库实现，如leveldb)进行存储，如下图所示，对象的扩展属性数据与对象数据存储在同一块盘上；



使用本结构存储后，write小文件file\_a操作分解为:

1. 对某个object调用append小文件file\_a；2)将小文件file\_a在相应object的offset和size，及小文件名字file\_a作为object的扩展属性存储kv数据库。

read小文件file\_a操作分解为：

1. 读取相应object的file\_a对应的扩展属性值(及offset，size)；
2. 读取object的offset偏移开始的size长度的数据。

对于删除操作，直接将相应object的file\_a对应的扩展属性键值删除即可，file\_a所占用的存储空间延迟回收，回收方案以后讨论。另外，Ceph本身是强一致存储系统，其内在机制可以保证object及其扩展属性数据的可靠一致；

　　由于对象的扩展属性数据与对象数据存储在同一块盘上，小文件的读写操作全部在本机本OSD进程内完成，避免了网络交互机制潜在的问题；另一方面，对于写操作，一次小文件写操作对应两次本地磁盘随机io(逻辑层面)，且不能更少，某些kv数据库(如leveldb)还存在write amplification问题，对于写压力大的业务场景，此方案不能很好地满足；不过对于读操作，我们可以通过配置参数，尽量将kv数据保留在内存中，实现读取操作一次磁盘io的预期目标；

　　如何选择若干小文件进行合并，及合并存储到哪个对象中呢？最简单地方案是通过计算小文件key的hash值，将具有相同hash值的小文件合并存储到id为对应hash值的object中，这样每次存取时，先根据key计算出hash值，再对id为hash值的object进行相应的操作；关于hash函数的选择，(1)可使用最简单的hash取模，这种方法需要事先确定模数，即当前业务合并操作使用的object个数，且确定后不能改变，在业务数据增长过程中，小文件被平均分散到各个object中，写压力被均匀分散到所有object（即所有物理磁盘，假设object均匀分布）上；object文件大小在一直增长，但不能无限增长，上限与单块磁盘容量及存储的object数量有关，所以在部署前，应规划好集群的容量和hash模数。(2)对于某些带目录树层次信息的数据，如/a/b/c/d/efghi.jpg，可以将文件的目录信息作为相应object的id，及/a/b/c/d，这样一个子目录下的所有文件存储在了一个object中，可以通过rados的listxattr命令查看一个目录下的所有文件，方便运维使用；另外，随着业务数据的增加，可以动态增加object数量，并将之前的object设为只读状态（方便以后的其它处理操作），来避免object的无限增长；此方法需要根据业务写操作量及集群磁盘数来合理规划当前可写的object数量，在满足写压力的前提下将object大小控制在一定范围内。

　　本方案是为小文件(1MB及以下)设计的，对于稍大的文件存储(几十MB甚至更大)，如何使用本方案存储呢？我们将大文件large\_file\_a做stripe切片分成若干大小一样(如2MB，可配置，最后一块大小可能不足2MB)的若干小块文件：large\_file\_a\_0, large\_file\_a\_1 ... large\_file\_a\_N，并将每个小块文件作为一个独立的小文件使用上述方案存储，分片信息(如总片数，当前第几片，大文件大小，时间等)附加在每个分片数据开头一并进行存储，以便在读取时进行解析并根据操作类型做相应操作。

　　根据业务的需求，我们直接基于librados接口进行封装，提供如下操作接口供业务使用（c++描述）：

[IMG_257](javascript:void(0);)

int WriteFullObj(const std::string& oid, bufferlist& bl, int create\_time = GetCurrentTime());

int Write(const std::string& oid, bufferlist& bl, uint64\_t off, int create\_time = GetCurrentTime());

int WriteFinish(const std::string& oid, uint64\_t total\_size, int create\_time = GetCurrentTime());

int Read(const std::string& oid, bufferlist& bl, size\_t len, uint64\_t off);

int ReadFullObj(const std::string& oid, bufferlist& bl, int\* create\_time = NULL);

int Stat(const std::string& oid, uint64\_t \*psize, time\_t \*pmtime, MetaInfo\* meta = NULL);

int Remove(const std::string& oid);

int BatchWriteFullObj(const String2BufferlistHMap& oid2data, int create\_time = GetCurrentTime());

[IMG_258](javascript:void(0);)

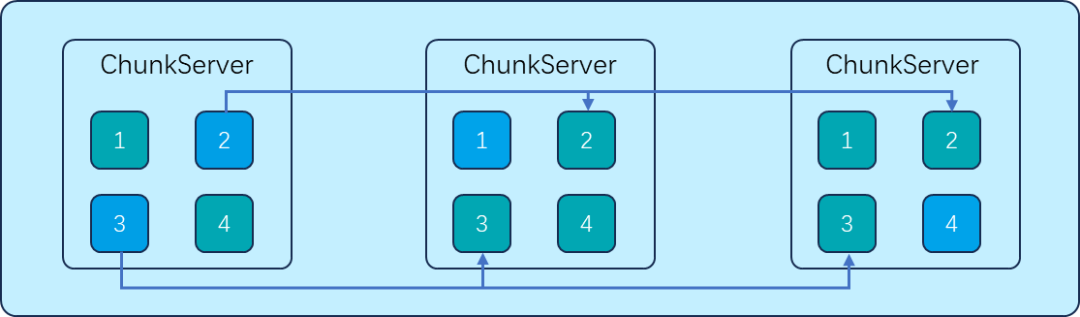
对于写小文件可直接使用WriteFullObj；对于写大文件可使用带offset的Write，写完所有数据后，调用WriteFinish；对于读取整个文件可直接使用ReadFullObj；对于随机读取文件部分数据可使用带offset的Read；Stat用于查看文件状态信息；Remove用于删除文件；当使用第二种hash规则时，可使用BatchWriteFullObj提高写操作的吞吐量。

# zStorage线程模型

**单线程，一台服务器上，一个PG对应一个线程**

# zStorage的raft组

在 zStorage 中，Raft协议主要用在数据面的ChunkServer中，通过Raft协议保证数据副本间的一致性。为了分散负载以及让系统可以水平扩展，我们采用了多个Raft组，每个Raft组负责部分数据的同步。



有4个Raft组，每个Raft组有3个副本，所有副本分布在3个不同的ChunkServer上。每个Raft组有1个Leader副本（蓝色），两个Follower副本。

# zStorage 的Raft快照

## 快照目的

<https://blog.csdn.net/weixin_54551388/article/details/136467523>

在[Raft](https://so.csdn.net/so/search?q=Raft&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_54551388/article/details/_blank)协议中，如果Raft组某个成员的数据不完整，缺失部分或者全部数据，Raft会用复制状态机快照的方式把数据同步给这个成员。Raft采用复制状态机模型，状态机的Raft保证状态机指令序列的一致性，这个指令序列存储在Raft日志中。

如果某个Raft成员发生了故障，后来经过维修恢复了，或者给这个Raft成员换了一台新服务器后重新上线运行了，此时需要把故障期间增加的新数据同步给这个成员。Raft的做法是，把这个成员缺失的日志数据重新复制给它。有了日志数据，再把这些日志中的指令序列在状态机上重新应用一遍，状态机中的数据就更新了。

经过长时间运行之后，Raft日志条目数量会越来越多，这些日志会占用很多存储空间，因此Raft需要定期删除部分日志。Raft的做法是定期把内存中的状态机数据转存一份到硬盘上，这份状态机数据就是Raft的快照。转存的过程，Raft叫做打快照。日志中的一部分旧条目数据，已经被更新到硬盘快照中，这部分旧的日志数据也就不再需要了，可以从硬盘上删除。

打了快照之后，如果某个Raft成员发生了故障，又重新上线了，需要给这个成员同步快照和日志两种数据，不能只是同步快照数据。

# Raft client

Raft的client会把所有的请求发到leader上执行，在client刚启动时，会随机选择集群中的一个server

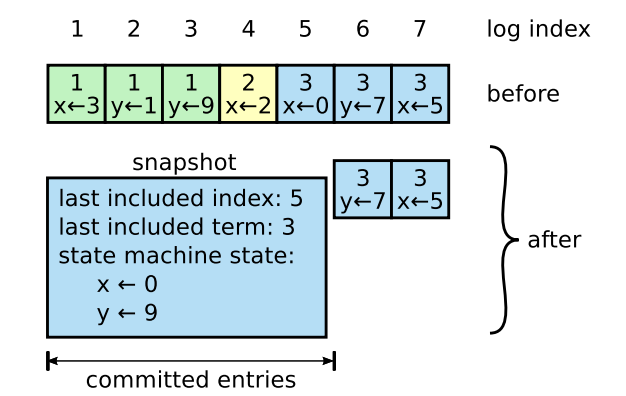
* 如果选择的server是leader，那么client会把请求发到该server上
* 如果选择的server不是leader，该server会把leader的地址告诉给client，后续client会把请求发给该leader
* 如果此时没有leader，那么client会timeout，client会重试其他server，直到找到leader

## 快照方法

Raft的日志会随着处理客户端请求数量的增多而不断增大，在实际系统中，日志不可能会无限地增长，原因如下：

* 占用的存储空间随着日志增多而增加
* 日志越多，server当掉重启时需要回放的时间就越长

因此，需要定期地清理日志，Raft采用最简单的快照方法。对系统当前做快照时，会把当前状态持久化到存储中，然后到快照点的日志项都可以被删除。

[](http://oserror.com/images/raft_log_compaction.png)

Raft算法中每个server单独地做快照，即把当前状态机的状态写入到存储中（状态机中的状态都是已提交的log entry回放出来的）。除了状态机的状态外，Raft快照中还需要一些元数据信息，包括如下：

快照中包含的最后一个log entry的index和term，记录这些信息的目的是为了使得AppendEntriesRPC的一致性检查能通过，因为，在复制紧跟着快照后的log entry时，AppendEntries RPC带上需要复制的log entry前一个log entry的(index, iterm)，即快照的最后一个log entry的(index，term)，因此，快照中需要记录最后一个log entry的(index，term)

为了支持集群成员变更，快照中保存的元数据还会存储集群最新的配置信息。

当server完成快照后，可以删除快照最后一个log entry及其之前所有的log entry，以及之前的快照。

虽然每个server是独立地做快照的，但是也有可能存在需要leader向follower发送整个快照的情况，例如，一个follower的日志处于leader的最近一次快照之前，恰好leader做完快照之后把其快照中的log entry都删除了，这时，leader就无法通过发送log entry来同步了，只能通过发送完整快照。

leader通过InstallSnapshot RPC来完成发送快照的功能，follower收到此RPC后，根据不同情况会有不同的处理：

* 当follower中缺失快照中的日志时

follower会删除掉其上所有日志，并清空状态机

* 当follower中拥有快照中所有的日志时

follower会删掉快照所覆盖的log entry，但快照后所有日志都保留。备注：这里论文中没有提是否还是从leader接受快照，个人觉得follower可以自己做快照，并拒绝掉leader发快照的RPC请求

对于Raft快照，关于性能需要考虑的点有：

* server何时做快照，太频繁地做快照会浪费磁盘I/O；太不频繁会导致server当掉后回放时间增加，可能的方案为当日志大小到一定空间时，开始快照。备注：如果所有server做快照的阈值空间都是一样的，那么快照点也不一定相同，因为，当server检测到日志超过大小，到其真正开始做快照中间还存在时间间隔，每个server的间隔可能不一样
* 写快照花费的时间很长，不能让其影响正常的操作。可以采用copy-on-write操作，例如linux的fork

## Raft日志复制原理

<https://www.bilibili.com/video/BV1Ap4y1n7z7/?spm_id_from=333.337.search-card.all.click&vd_source=b205d76f467a1db2a1841491b422cfe0>

[动画演示 raft 在脑裂发生之后仍然可以正常工作吗？](https://www.bilibili.com/video/BV1Fi4y147ad/?spm_id_from=333.337.search-card.all.click&vd_source=b205d76f467a1db2a1841491b422cfe0)

[Raft算法Leader选举、脑裂后选举、日志复制、修复不一致日志和数据安全](https://www.bilibili.com/video/BV1so4y1r7eM/?spm_id_from=333.337.search-card.all.click&vd_source=b205d76f467a1db2a1841491b422cfe0)

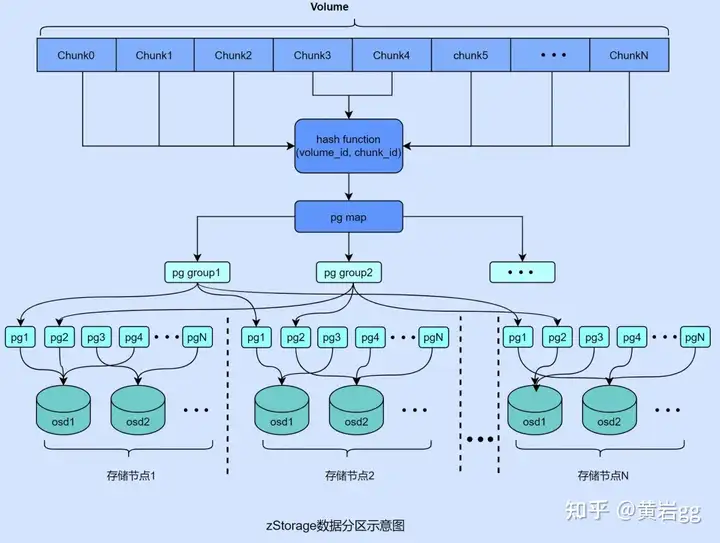
[请描述Raft算法和ZAB算法有哪些异同点，并描述具体细节](https://www.bilibili.com/video/BV1Bj411u7An/?spm_id_from=333.337.search-card.all.click&vd_source=b205d76f467a1db2a1841491b422cfe0)

# osD

为了异步事件处理模型和无锁化编程，zStorage的数据分区结构经过了精心设计。核心思想是将数据均匀地分布在各个硬盘上，并分配给不同CPU核处理。

采用了存储池和PG的概念，一个存储池包含多个PG（例如1024个），然后通过改进的CRUSH算法将PG映射到各个硬盘上，基本保证了各个硬盘上PG数量的均匀分布。接着，将卷按照4MB为单位划分为多个chunk，并根据卷ID和chunk ID将Chunk映射到某个PG上，保证了每个PG内存放了同样多的Chunk。尽管上述方式在4MB 粒度下基本保证了数据均匀分布，

但对于单流顺序读写模式存在一个问题：在某个时刻，IO负载只会落到某一个chunk上，无法发挥存储节点的多核、多盘的并发能力。因此，zStorage采用了条带化的方法，将多个chunk组成一个类似RAID0的条带组，使得顺序IO也能利用存储节点的并发处理能力，提升整体吞吐量。



# LSM Tree

## SSTable

[为什么大数据NoSQL采用LSM树结构](https://www.bilibili.com/video/BV18A411B7YY/?spm_id_from=333.337.search-card.all.click&vd_source=b205d76f467a1db2a1841491b422cfe0)

<https://blog.csdn.net/huanhuilong/article/details/130828038>