在当前众多的开源对象存储项目中，Ceph与OpenStack Swift 无疑是其中的佼佼者。

Ceph用C++编写而Swift用Python编写，表面上性能应当是Ceph占优，但是由于Ceph采用强一致性设计，需要等到所有的数据副本都被修改了之后才能进行读取操作。这种模式读取效率较高，但是写入效率低。而swift采用最终一致性设计，读写都比较频繁的场景，所以采用了比较折中的策略，即写操作需要满足至少一半以上成功 W >N/2，再保证读操作与写操作的副本集合至少产生一个交集，即 R+W>N。相对而言Swift的存储方式更灵活一些，可以牺牲一定的一致性从而提高读取速度。而一致性可以通过后台的方式完成同步，从而保证数据的完整。所以，在具体的应用上，swift的读写效率是要高于ceph的。

根据CAP原理，swift可以在跨度很广的地理空间上部署swift集群。

虽然Ceph在性能上不如swift，但是它可以直接对接Swift接口。

Ceph很重要的一个短板就是安全性。云计算节点上的RADOS客户端直接与RADOS服务器交互所使用的网络与Ceph用于未加密复制流量的网络相同。如果某个Ceph客户端节点被入侵，攻击者便能得到存储网络的所有流量。

ceph毕竟不是一个专门的对象存储系统，其对象存储服务其实是在block服务上模拟出来的。Ceph这样的系统一般不支持跨机房，跨地域的大规模部署。如果部署只在单一地域，没有计划扩展到多个地域时，Ceph会是很好的选择。但是，如果要考虑大规模部署的话，Swift可能更为适合。

Ceph是十分出色的块存储系统，而swift适合用于绝大多数对象存储。

CEPH

1. CRUSH算法

Crush(Controlled Replication Under Scalable Hashing)算法是ceph的两大创新之一，ceph使用crush算法完成数据的寻址操作。Crush 在一致性哈希算法的基础上很好地考虑了容灾域的隔离。

ceph设计了CRUSH(一个可扩展的伪随机数据分布算法)，用在分布式对象存储系统上，可以有效映射数据对象到存储设备上(不需要中心设备)。因为大型系统的结构式动态变化的，CRUSH能够处理存储设备的添加和移除，并最小化由于存储设备的的添加和移动而导致的数据迁移。

<https://www.cnblogs.com/chenxianpao/p/5568207.html>

1. 高可用

Ceph中的数据副本数量可以由管理员自行定义，并可以通过CRUSH算法指定副本的物理存储位置以分隔故障域，支持数据强一致性；ceph可以忍受多种故障场景并自动尝试并行修复。

1. 高扩展性

Ceph不同于swift，客户端所有的读写操作都要经过代理节点。一旦集群并发量增大时，代理节点很容易带来单点故障（为了提高并发量，swift的代理节点可以建立多个）。Ceph本身没有主控节点，扩展起来比较容易，并且理论上，它的性能会随着磁盘数量的增加而线性增长。

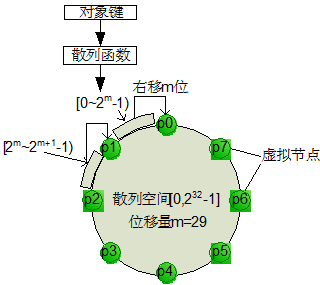
http://www.jianshu.com/p/25163032f57f

Swift

Swift构筑在比较便宜的标准硬件存储基础设施之上，无需采用RAID，通过在软件层面引入一致性散列技术和数据冗余性，牺牲一定程度的数据一致性来达到高可用性和可伸缩性，支持多租户模式、容器和对象读写操作，适合解决互联网的应用场景下非结构化数据存储问题。

1. 一致性散列技术（Consistent Hashing）

Swift 是基于一致性散列技术，通过计算可将对象均匀分布到虚拟空间的虚拟节点上，在增加或删除节点时可大大减少需移动的数据量；虚拟空间大小通常采用 2 的 n 次幂，便于进行高效的移位操作；然后通过独特的数据结构 Ring（环）再将虚拟节点映射到实际的物理存储设备上，完成寻址过程。



以逆时针方向递增的散列空间有 4 个字节长共 32 位，整数范围是[0~232-1]；将散列结果右移 m 位，可产生 232-m个虚拟节点，例如 m=29 时可产生 8 个虚拟节点。

1. 数据一致性模型

按照 Eric Brewer 的 CAP（Consistency，Availability，Partition Tolerance）理论，无法同时满足 3 个方面，Swift 放弃严格一致性（满足 ACID 事务级别），而采用最终一致性模型（Eventual Consistency），来达到高可用性和无限水平扩展能力。为了实现这一目标，Swift 采用 Quorum 仲裁协议(Quorum 有法定投票人数的含义)：

（1）定义：N：数据的副本总数；W：写操作被确认接受的副本数量；R：读操作的副本数量

（2）强一致性：R+W>N，以保证对副本的读写操作会产生交集，从而保证可以读取到最新版本；如果 W=N，R=1，则需要全部更新，适合大量读少量写操作场景下的强一致性；如果 R=N，W=1，则只更新一个副本，通过读取全部副本来得到最新版本，适合大量写少量读场景下的强一致性。

（3）弱一致性：R+W<=N，如果读写操作的副本集合不产生交集，就可能会读到脏数据；适合对一致性要求比较低的场景。

Swift 针对的是读写都比较频繁的场景，所以采用了比较折中的策略，即写操作需要满足至少一半以上成功 W >N/2，再保证读操作与写操作的副本集合至少产生一个交集，即 R+W>N。Swift 默认配置是 N=3，W=2>N/2，R=1 或 2，即每个对象会存在 3 个副本，这些副本会尽量被存储在不同区域的节点上；W=2 表示至少需要更新 2 个副本才算写成功；当 R=1 时意味着某一个读操作成功便立刻返回，此种情况下可能会读取到旧版本（弱一致性模型）；当 R=2 时，需要通过在读操作请求头中增加 x-newest=true 参数来同时读取 2 个副本的元数据信息，然后比较时间戳来确定哪个是最新版本（强一致性模型）；如果数据出现了不一致，后台服务进程会在一定时间窗口内通过检测和复制协议来完成数据同步，从而保证达到最终一致性。

1. 环的数据结构

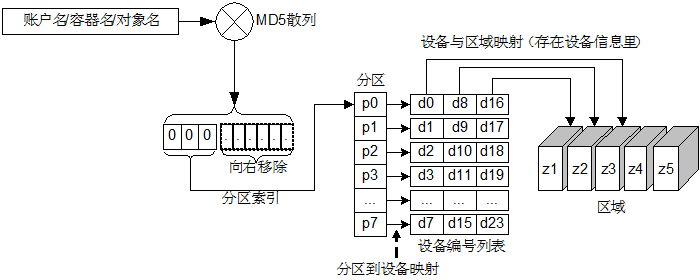
环是为了将虚拟节点（分区）映射到一组物理存储设备上，并提供一定的冗余度而设计的，其数据结构由以下信息组成：

（1）存储设备列表、设备信息包括唯一标识号（id）、区域号（zone）、权重（weight）、IP 地址（ip）、端口（port）、设备名称（device）、元数据（meta）。

（2）分区到设备映射关系（replica2part2dev\_id 数组)

（3）计算分区号的位移(part\_shift 整数，即图 1 中的 m)

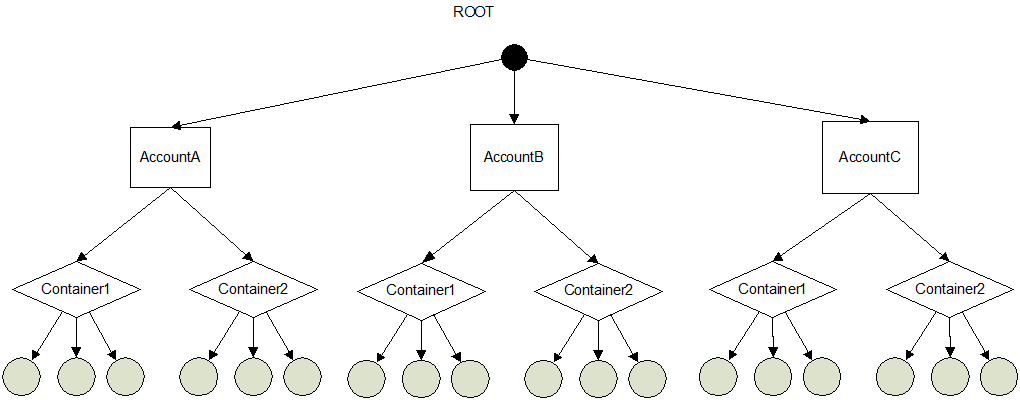
以查找一个对象的计算过程为例：



使用对象的层次结构 account/container/object 作为键，使用 MD5 散列算法得到一个散列值，对该散列值的前 4 个字节进行右移操作得到分区索引号，移动位数由上面的 part\_shift 设置指定；按照分区索引号在分区到设备映射表（replica2part2dev\_id）里查找该对象所在分区的对应的所有设备编号，这些设备会被尽量选择部署在不同区域（Zone）内，区域只是个抽象概念，它可以是某台机器，某个机架，甚至某个建筑内的机群，以提供最高级别的冗余性，建议至少部署 5 个区域；权重参数是个相对值，可以来根据磁盘的大小来调节，权重越大表示可分配的空间越多，可部署更多的分区。

总的来说，Ring引入一致性哈希的原因是为了减少由于增加结点导致数据项移动的数量来提高单调性；引入partition的原因是为了减少由于节点数过少导致移动过多的数据项(数据负载不均衡)；引入replica的原因是防止数据单点、提高冗余性；引入zone的原因是为了保证分区容忍性；引入weight的原因是为了保证partition分配的均衡。

5. 数据模型



Swift将整个存储分为三个层次：Account、Container 和 Object。

这里的Account本身只是一个存储区域，并不代表认证系统里的“账号”，但是通常会让每个Account对应一个租户。这就是为什么我们作为一个OpenStack用户在使用Swift时，只能看到Container和Object，而看不到Account的原因，如果这个用户切换到另一个租户下，他将看到属于另一个租户也是是另一个Account下的Container和Object。