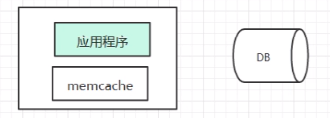
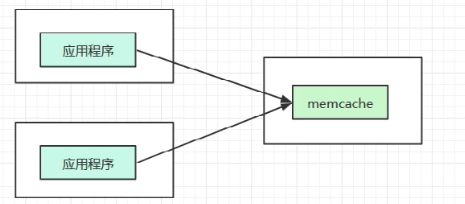
# 背景

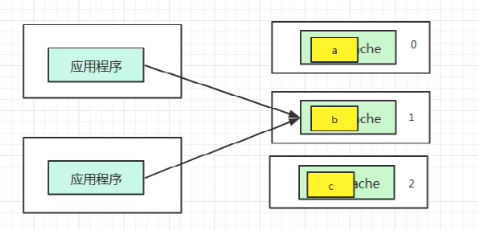
对于数据量不是很大的情况下，采用本地缓存即可解决快速访问数据库的问题：



对于数据量很大的情况下，需要将应用程序和缓存分别置于不同机器，即采用远程分布式缓存（Redis单机并发10W+）：



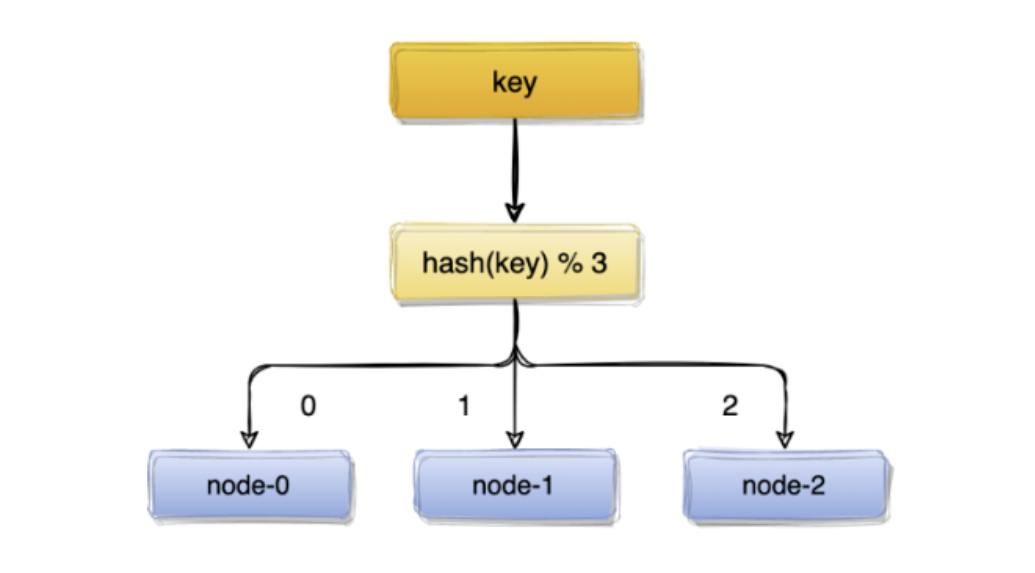
对于大型高并发系统（高并发，海量数据），针对客户端高并发Redis缓存无法满足需要，需要做缓存集群，针对海量数据，缓存数据会超过单台机器的内存限制，所以需要将数据分割，采用数据分布式存储：



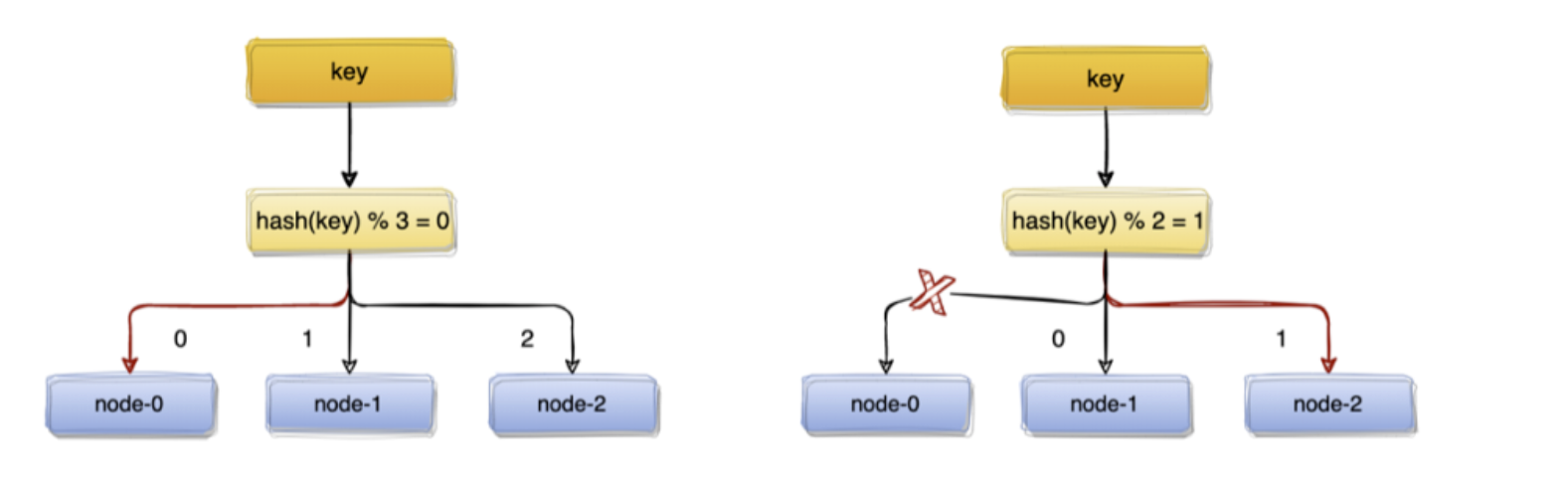
考虑这么一种场景：我们有三台缓存服务器编号node0、node1、node2，现在有3000万个key，希望可以将这些个key均匀的缓存到三台机器上，你会想到什么方案呢？

我们可能首先想到的方案是：取模算法hash（key）%N，即：对key进行hash运算后取模，N是机器的数量。

这样，对key进行hash后的结果对3取模，得到的结果一定是0、1或者2，正好对应服务器node0、node1、node2，存取数据直接找对应的服务器即可，简单粗暴，完全可以解决上述的问题。



取模算法虽然使用简单，但对机器数量取模，在集群扩容和收缩时却有一定的局限性：因为在生产环境中根据业务量的大小，调整服务器数量是常有的事，而服务器数量N发生变化后hash（key）%N计算的结果也会随之变化！



比如：一个服务器节点挂了，计算公式从hash（key）% 3变成了hash（key）% 2，结果会发生变化，此时想要访问一个key，这个key的缓存位置大概率会发生改变，那么之前缓存key的数据也会失去作用与意义。

大量缓存在同一时间失效，造成缓存的雪崩，进而导致整个缓存系统的不可用，这基本上是不能接受的。为了解决优化上述情况，一致性hash算法应运而生。

应用程序要想读取不同节点上的缓存数据，可以采用如下方法：

1. 哈希求余：hash(key) % 缓存集群节点数
2. 一致性hash
3. 一致性hash+虚拟节点

# 原理

一致性哈希算法在1997年由麻省理工学院提出，是一种特殊的哈希算法，在移除或者添加一个服务器时，能够尽可能小地改变已存在的服务请求与处理请求服务器之间的映射关系；

一致性哈希解决了简单哈希算法在分布式哈希表（Distributed Hash Table，DHT）中存在的动态伸缩等问题。

一致性hash算法本质上也是一种取模算法。不过，**不同于按服务器数量取模的哈希算法，一致性hash是对固定值2^32取模**。

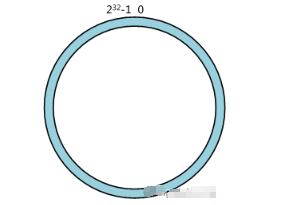
IPv4的地址是4组8位2进制数组成，所以用2^32可以保证每个IP地址会有唯一的映射。

传统哈希算法有这样的一个缺陷，如果增加服务器数量，则原来哈希计算出来的存储节点会失效，即缓存失效，造成雪崩。

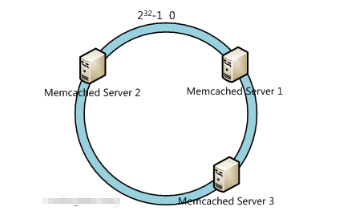
一致性哈希算法将经过哈希计算后的结果映射到一个环上，然后顺时针查找，下一个节点就是需要存放数据的节点。缓存机器数量发生变化的时候，一致性哈希算法可以使得大部分缓存数据可以被访问，即**不是所有缓存数据失效，只有少部分缓存失效**。

## 步骤

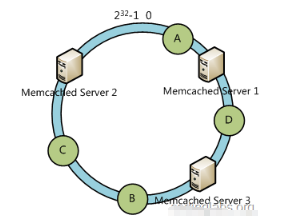
简单来说，一致性哈希将整个哈希值空间组织成一个虚拟的圆环，如假设某哈希函数H的值空间为0 - 232-1（即哈希值是一个32位无符号整形），整个哈希空间环如下：



整个空间按顺时针方向组织。0和232-1在零点中方向重合。下一步将各个服务器使用H进行一个哈希，具体可以选择服务器的ip或主机名作为关键字进行哈希，这样每台机器就能确定其在哈希环上的位置，这里假设将上文中三台服务器使用ip地址哈希后在环空间的位置如下：



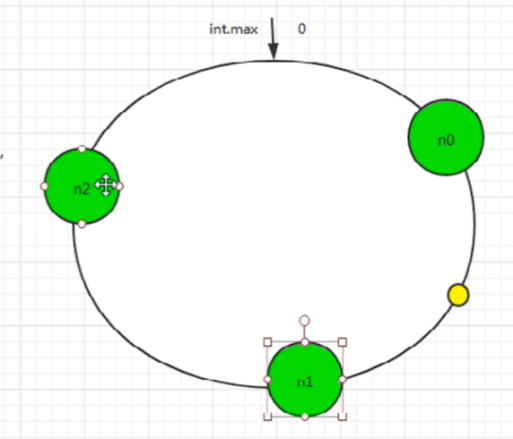
接下来使用如下算法定位数据访问到相应服务器：将数据key使用相同的函数H计算出哈希值h，根据h确定此数据在环上的位置，从此位置沿环顺时针“行走”，第一台遇到的服务器就是其应该定位到的服务器。例如我们有A、B、C、D四个数据对象，经过哈希计算后，在环空间上的位置如下：



根据一致性哈希算法，数据A会被定为到Server 1上，D被定为到Server 3上，而B、C分别被定为到Server 2上。

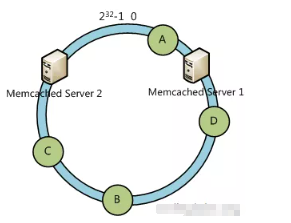
总结，一致性hash算法的步骤：

1. hash值是个整数非负，非负整数的值范围做成一个圆环；
2. 对集群的节点的某个属性（比如节点名）求hash值，放到环上；
3. 对数据key求hash值，也放到环上，按照顺时针方向找到离它最近的节点，放在它上面。



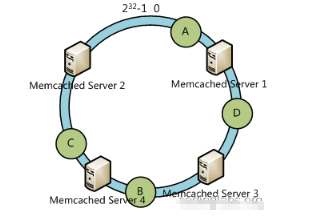
## 容错性/扩展性

现假设Server 3宕机了：



可以看到此时A、C、B不会受到影响，只有D节点被重定位到Server 2。一般的，在一致性哈希算法中，如果一台服务器不可用，则受影响的数据仅仅是此服务器到其环空间中前一台服务器（即顺着逆时针方向行走遇到的第一台服务器）之间数据，其它不会受到影响。

下面考虑另外一种情况，如果我们在系统中增加一台服务器Memcached Server 4：



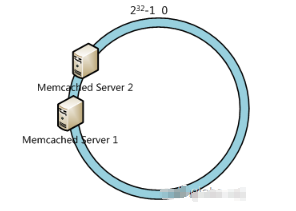
此时A、D、C不受影响，只有B需要重定位到新的Server 4。一般的，在一致性哈希算法中，如果增加一台服务器，则受影响的数据仅仅是新服务器到其环空间中前一台服务器（即顺着逆时针方向行走遇到的第一台服务器）之间数据，其它不会受到影响。

综上所述，一致性哈希算法对于节点的增减都只需重定位环空间中的一小部分数据，具有较好的容错性和可扩展性。

## 虚拟节点

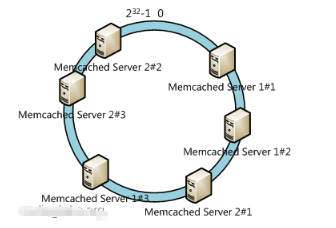
但是**一致性哈希算法存在hash偏斜的问题**，即缓存不均匀，导致系统崩溃。针对这种问题，可以增加服务器，使得尽可能均匀分布，为此引入虚拟节点（缓存读写🡪虚拟节点🡪真实节点🡪读写）。

一致性哈希算法在服务节点太少时，容易因为节点分部不均匀而造成数据倾斜问题。 例如我们的系统中有两台服务器，其环分布如下：



此时必然造成大量数据集中到Server 1上，而只有极少量会定位到Server 2上。为了解决这种数据倾斜问题，一致性哈希算法引入了虚拟节点机制，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个此服务节点，称为虚拟节点。

具体做法可以在服务器ip或主机名的后面增加编号来实现。例如上面的情况，我们决定为每台服务器计算三个虚拟节点，于是可以分别计算“Memcached Server 1#1”、“Memcached Server 1#2”、“Memcached Server 1#3”、“Memcached Server 2#1”、“Memcached Server 2#2”、“Memcached Server 2#3”的哈希值，于是形成六个虚拟节点：



同时数据定位算法不变，只是多了一步虚拟节点到实际节点的映射，例如定位到“Memcached Server 1#1”、“Memcached Server 1#2”、“Memcached Server 1#3”三个虚拟节点的数据均定位到Server 1上。这样就解决了服务节点少时数据倾斜的问题。在实际应用中，通常将虚拟节点数设置为32甚至更大，因此即使很少的服务节点也能做到相对均匀的数据分布。

# 应用

一致性hash在分布式系统中应该是实现负载均衡的首选算法，它的实现比较灵活，既可以在客户端实现，也可以在中间件上实现，比如日常使用较多的缓存中间件memcached和redis集群都有用到它。

memcached的集群比较特殊，严格来说它只能算是伪集群，因为它的服务器之间不能通信，请求的分发路由完全靠客户端来的计算出缓存对象应该落在哪个服务器上，而它的路由算法用的就是一致性hash。

还有redis集群中hash槽的概念，虽然实现不尽相同，但思想万变不离其宗。

其它的应用场景还有很多：

RPC框架Dubbo用来选择服务提供者

分布式关系数据库分库分表：数据与节点的映射关系

LVS负载均衡调度器

应用一致性哈希算法：缓存（Redis/memcache），Hadoop，搜索引擎（ES），分布式数据库。即数据分布式存储的场景，需要考虑使用一致性哈希算法。

## 缓存

## Hadoop

## ES

## 分布式数据存储

分布式数据库领域的应用：

1. 水平拓展：

https://blog.csdn.net/linuxheik/article/details/51025879

https://blog.csdn.net/ydyang1126/article/details/70313981

1. 其他

https://yq.aliyun.com/articles/57954

https://www.cnblogs.com/lonecloud/p/8093328.html

比如在分布式数据库中动态扩容的方案中，需要采用一致性哈希解决。