《网络存储技术》课程作业五

一致性哈希技术概述

付容天 学号 2020211616

班级 2020211310

计算机科学与技术系

计算机学院（国家示范性软件学院）

2022 年 11 月 24 日

目录

1 引言

一致性哈希（Consistent Hashing）是一种特殊的哈希技术，由麻省理工学院在 1997 年提出，目的是解决分布式缓存的问题。当调整哈希表大小（例如移除或添加一个服务器）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。相比之下，在大多数传统的哈希表中，数组槽数的变化会导致几乎所有键重新映射，因为键和槽之间的映射通常是由模算数（Modular Arithmetic）定义的。

David Karger 等人引入了“一致性哈希”的概念。1997 年在计算理论研讨会上发表的论文《Consistent Hashing and Random Trees》首次介绍了术语“一致性哈希”，并将其解释为不断变化的网络服务器群体中分发请求的一种方式。具体来讲，在可伸缩期间或中断期间，如果添加服务器和删除服务器，则只有 num_keys/num_slots 个项目需要被重新映射。在论文中，作者对比了线性散列（Linear Hashing）处理顺序服务器添加和删除的能力，从而突显出了一致性哈希允许以任意顺序添加和删除服务器的优越性。

一致性哈希还被用于减少大型 Web 应用程序中部分系统故障的影响，以提供健壮的缓存。一致性哈希也是分布式哈希表（Distributed Hash Table，DHT）的基石，DHT 使用哈希值在分布式节点集上划分键空间，然后构建连接节点的覆盖网络，通过键来提供高效的节点检索。

在 Karger 等人的论文中，还提出了对一致性哈希算法进行评判的四个指标：

- 平衡性（Balance）：不同键的哈希结果分布均衡，尽可能的均衡地分布到各节点上，平衡性跟哈希函数关系密切，目前已经发展出了许多哈希算法都有较好的平衡性
- 单调性（Monotonicity）：当有新的节点上线后，系统中原有的键要么还是映射到原来的节点上、要么映射到新加入的节点上，而不会出现从一个原有节点重新映射到另一个原有节点的情况，即：当可用存储桶的集合发生更改时，只有在必要时才移动项目以保持均匀的分布
- 分散性（Spread）：由于客户端可能看不到后端的所有服务（例如，只能看到一部分节点），这种情况下对于固定的键，在两个客户端上可能被分散到不同的后端服务，从而降低后端存储的效率，所以算法应该尽量降低分散性（尽量保证同一个键的分配不能过于分散）
- 服务器负载均衡（Load）：负载主要是从服务器的角度来看，指各服务器的负载应该尽量均衡，至多认为有一定数目的键存储于某个节点中，一个节点不能存储过多的键

2 一致性哈希概要

2.1 一致性哈希基本技术

这里我们从一个实例开始讨论：当一个二进制大对象（Binary Large Object，BLOB）必须分配给 n 集群上的一个服务器时，如果使用标准哈希函数来计算该 BLOB 的哈希值、且假设哈希的结果值为 β，那么可以根据模算数来确定放置 BLOB 的服务器即为 $\zeta = \beta\%n$，因此该 BLOB 将被放置在这个服务器中。但这样实现的一个主要问题就是在中断或缩放期间添加或删除服务器时（例如，在集群中服务器的数量 n 更改时），每个服务器中的所有 BLOB 都应该进行重新映射和移动，这个操作的代价十分昂贵，这就成为了系统性能的一个重要瓶颈。

为了避免在整个集群中添加或删除服务器时必须重新分配每个 BLOB 的问题，一致性哈希技术被提出并实现。一致性哈希的中心思想是：我们可以使用哈希函数将 BLOB 和服务器随机映射到一个单位圆，例如 $\zeta = \phi\%(2\pi)$，其中 ϕ 是 BLOB 或服务器标识符（常见的是 IP 地址）的哈希映射结果。然后将每个 BLOB 分配给按顺时针顺序出现在圆圈上的下一个服务器。通常，二进制搜索算法或线性搜索算法可以用于查找节点或服务器以放置特定的 BLOB，对应的复杂度分别为 $O(\log N)$ 和 $O(N)$。

在请求服务时，需要将服务请求先使用哈希算法计算出哈希值，然后根据哈希值在圆上顺时针进行查找，第一台遇到的服务器就是所对应的处理请求服务器。当增加一台新的服务器（或服务器发生故障并从圆环上移除）时，受影响的数据仅仅是新添加（或删除）的服务器到其环空间中前一台的服务器（也就是顺着逆时针方向遇到的第一台服务器）之间的数据，其他都不会受到影响。

因此，一致性哈希的显著优点包括：

- 可扩展性：一致性哈希算法保证了增加或减少服务器时，数据存储的改变最少，相比传统哈希算法大大节省了数据移动的开销
- 更好地适应数据的快速增长：采用一致性哈希算法分布数据，当数据不断增长时，部分虚拟节点中可能包含很多数据、造成数据在虚拟节点上分布不均衡，此时可以将包含数据多的虚拟节点分裂，这种分裂仅仅是将原有的虚拟节点一分为二、不需要对全部的数据进行重新哈希和划分。虚拟节点分裂后，如果物理服务器的负载仍然不均衡，只需在服务器之间调整部分虚拟节点的存储分布。这样可以随数据的增长而动态的扩展物理服务器的数量，且代价远比传统哈希算法重新分布所有数据要小很多

2.2 经典一致性哈希（Karger Hashing）

经典一致性哈希通常是按照 Karger 等人的论文《Web Caching with Consistent Hashing》中提到的方法进行实现的，该方法也常被称为 “环割法”，下面介绍该方法的主要思路。

首先，我们将对象节点通过哈希计算之后，映射到一个范围是 $[0, 2^{32} - 1]$ 的环上，例如，假设我们有：

- Hash(object1) = key1;
- Hash(object2) = key2;
- Hash(object3) = key3;
- Hash(object4) = key4;

那么这四个对象可以按照下面的顺序存储在环上：

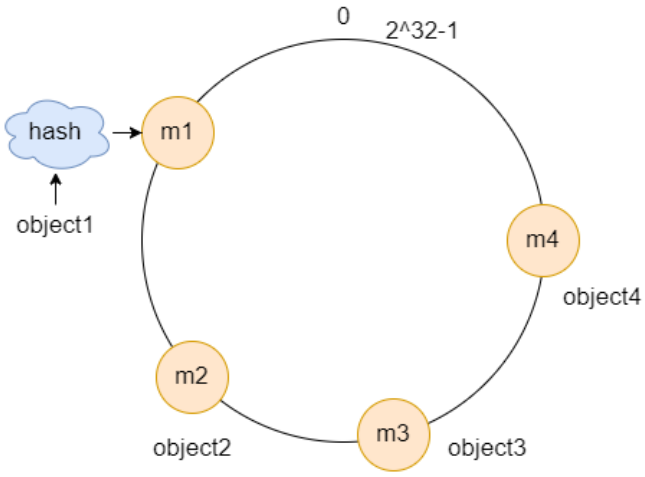

图 1: 对象映射到环上

现在假设有三台机器，那么可以通过哈希算法将机器 IP 或唯一名称映射为键，即：

- Hash(node1) = key1;
- Hash(node2) = key2;
- Hash(node3) = key3;

然后可以映射到环上，映射后的示意图如下：

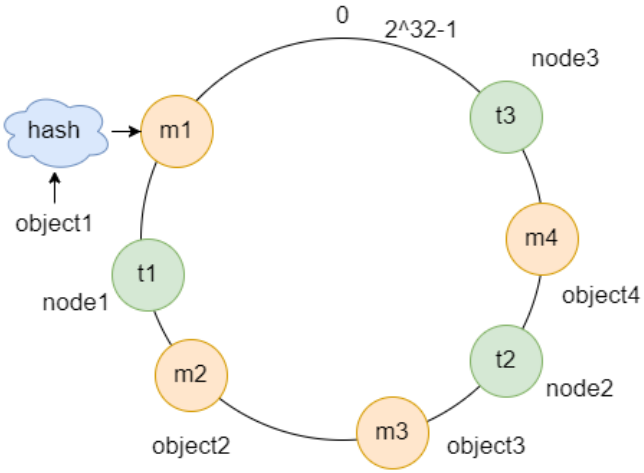

图 2: 机器映射到环上

接下来，由于数据和机器现处于统一空间，那么按照顺时针方向，我们将 object1 存储在 t3 中、object4 存储在 node2 中、object3 和 object2 则存储在 node1 中，如下图所示：

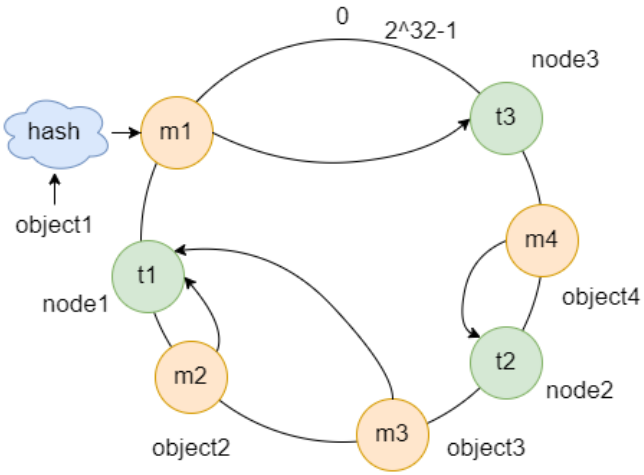

图 3: 对象存储到机器上

现在假设在上图的状态中添加一台机器，则通常模算数决定的算法由于保存键的服务器会发生巨大变化而影响缓存的命中率，但在一致性缓存中，只有增加服务器的地点逆时针方向的第一个节点会受到影响，如下图所示：

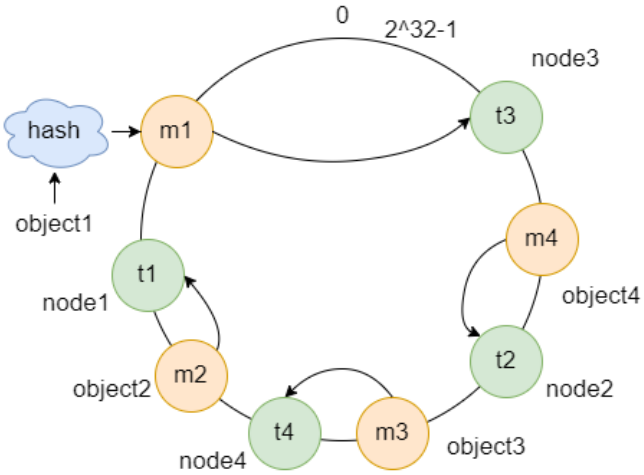

图 4: 新增机器

而如果要删除一台机器，那么根据一致性哈希的原则，也只有该机器管理的少数节点会受到影响，例如，删除图 3 中的 node1，则会得到下面的结果：

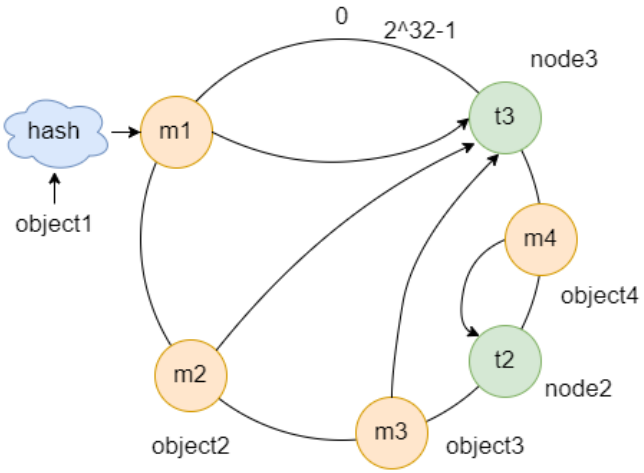

图 5: 删除机器

2.3 一致性哈希的优化

因为节点越多，它们在环上的分布就越趋向于均匀，所以我们可以向一致性哈希算法中引入虚拟节点机制，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，对于图 5 中展示的情况，m1、m2、m3 三个 object 都由 t3 进行管理，而 t2 只管理一个 m4，这造成资源分布不均衡，可能会严重影响系统性能，因此，我们可以创建 t2 和 t3 的虚拟节点，并插入到相应位置，得到如下图所示的情况：

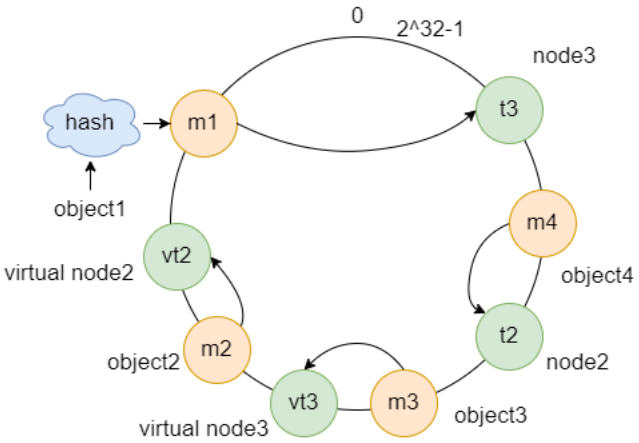

图 6: 创建虚拟节点

这样，t2 和 t3 每台机器就管理了两个节点，从而使资源分布更加均衡。并且，通过相应的管理机制，可以使虚拟节点和真实节点之间的区别对用户不可见，也就是说用户（数据）可以平等地访问虚拟节点和真实节点而不必关心他们之间的区别，这是由下面的结构实现的：

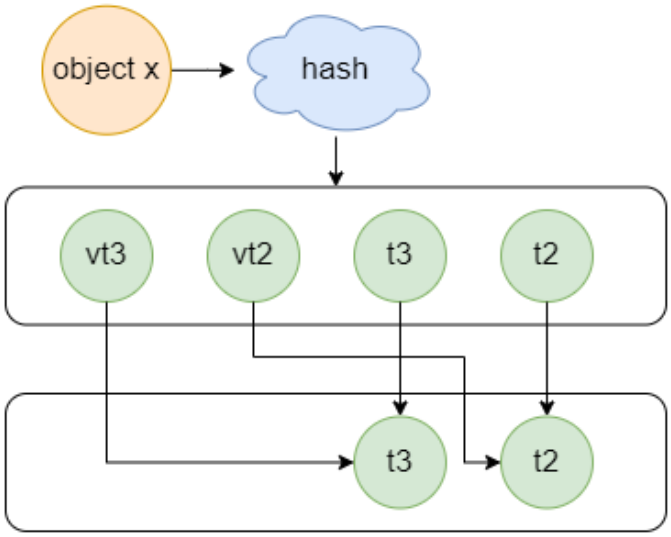

图 7: 虚拟节点映射关系

通常而言，当我们添加的虚拟节点越多，虚拟节点分布就越平均，数据倾斜的程度就越小。但是也不能无限地增加虚拟节点的个数，这是因为如果存在大量的虚拟节点，节点的查找性能就成为必须考虑的因素。

在实际使用中，每一个真实节点，都对应 N 个虚拟节点。所有的虚拟节点都在哈希环上随机分布（均匀分布）。那么，类似地，请求计算哈希后，可以先顺序查找找到相邻的虚拟节点，然后再查询虚拟节点对应的真实节点（如上图 7 所示）。每个真实节点都对应为若干的虚拟节点，虚拟节点按顺序分布在哈希环上，但是，虚拟节点间的距离不一定相等（平衡性）。因此在增删节点时，可能会涉及到大量虚拟节点范围变更，这会产生很多数据迁移，因为节点存储范围持

续变动。所以，我们可以将哈希环均匀分割为 Q 份（Q 应该远大于节点数量）。假设每份数据存 N 份，那么每 N 个节点就负责存储上游的 N 个间隔对应的键。当有节点退出时，其他节点均分其持有的虚拟节点（也可以选择触发一次重新分配）缺点是需要保存每一个虚拟节点到节点的对应关系。

在实际使用中，我们还可以使用红黑树（Red-Black Tree）来加快查找速度。

2.4 一致性哈希的实用扩展

前面我们提到，服务器负载均衡是一致性哈希性能的重要评判因素。为了在实践中有效地实现负载平衡，需要对基本技术进行大量扩展。在上面的基本方案中，如果一个服务器发生故障，它的所有 BLOB 都会按顺时针顺序重新分配给下一个服务器，这可能会使该服务器的负载加倍。这可能是不可取的。为了确保在服务器出现故障时更均匀地重新分配 BLOB，可以将每个服务器散列到单位圆上的多个位置。当服务器出现故障时，分配给单位圆上每个副本的 BLOB 将按顺时针顺序重新分配给不同的服务器，从而更均匀地重新分配 BLOB。

另一个扩展是，如果单个 BLOB 在过去的一段较短的时间内被访问很多次，并且这个 BLOB 必须托管在多个服务器中时，我们可以通过按顺时针顺序遍历单位圆、从而将 BLOB 分配给多个连续的服务器。需要注意的是，当两个 BLOB 在单位圆中彼此靠近散列并且同时在过去一段较短时间内被访问很多次时，则会出现更复杂的实际考虑。在这种情况下，两个 BLOB 将使用单位圆中的同一组连续服务器。这种情况可以通过每个 BLOB 选择不同的哈希函数来将服务器映射到单位圆从而进行改善。

2.5 一致性哈希的复杂度

假定我们有 N 个节点和 K 个键，那么复杂度可以总结如下（与经典哈希进行对比）：

	经典哈希	**一致性哈希**
添加节点	$O(K)$	$O(K/N + \log N)$
删除节点	$O(K)$	$O(K/N + \log N)$
添加键	$O(1)$	$O(\log N)$
删除键	$O(1)$	$O(\log N)$

其中 $O(K/N)$ 是重新分配键的平均成本，而 $O(\log N)$ 的复杂度则是因为需要在节点之间进行二进制搜索（Binary Search Algorithm）才能找到环上的下一个节点。

3 经典技术介绍

3.1 Rendezvous（HRW）

Rendezvous 哈希或最高权重（Highest Random Weight，HRW）哈希允许客户端在一组可能的 n 个选项中对一组 k 个选项达成分布式协议，因此 Rendezvous 哈希特别适用于客户端，用于分布式非中心化的客户端各自决定要请求的服务端。

Rendezvous 哈希比经典的一致性哈希（Karger Hashing）更简单也更通用，Karger 哈希可以作为 Rendezvous 哈希中一种 $k = 1$ 的特殊情况来进行考虑。并且，和 Karger 哈希相比，Rendezvous 哈希一个很大的改进就是没有虚拟节点的概念，因此增删节点十分方便。但是 Rendezvous 哈希每次进行查询操作的时候都需要进行复杂度为 $O(n)$ 的遍历，这在一定程度上限制了系统性能。

3.1.1 Rendezvous 概述

Rendezvous 哈希是由 David Thaler 和 Chinya Ravishankar 于 1996 年发明的，并于 1997 年以文献形式发表。鉴于其简单性和通用性，因此在现实世界的应用程序中，Rendezvous 哈希比 Karger 哈希更受欢迎。Rendezvous 哈希很早就在许多应用程序中使用，包括移动缓存、路由器设计、安全密钥建立以及和分布式数据库等。

第一个使用 Rendezvous 哈希的应用是使 Internet 上的多播（Multicast）客户端（在诸如 MBONE 的上下文中）能够以分布式方式识别多播集合点。它在 1998 年被微软的缓存阵列路由协议（Cache Array Routing Protocol，CARP）用于分布式缓存协调和路由。

Rendezvous 哈希解决了分布式哈希表的问题：给定一个对象，一组客户端如何对于 n 个站点确定放置该对象的位置？并且需要保证不同用户访问该对象时可以选择出同样的站点。对于这个问题，Rendezvous 哈希的思路是：给每个站点每个对象的权重，并将对象分配给得分最高的站点。通过在 n 个站点之间对哈希函数 $h(\cdot)$ 达成一致，可以确保不同站点将会访问同一站点。

具体来讲，对于对象 O_i，站点 S_j 被定义为具有分数 $w_{i,j} = h(O_i, S_j)$，然后将对象分配到具有最大分数的节点上去，每个客户端可以独立计算分数 $w_{i,j=1,2,\cdots,n}$ 并从中选择出同样的结果。特别地，对于一般的分布式 $k-$ 协议，用户可以选择前 k 个最大分数对应的节点。

根据上面的分析，我们可以将 Rendezvous 哈希的特点总结如下：

- 低开销：使用的哈希函数是高效的，因此客户端的开销非常低
- 负载平衡：由于哈希函数是随机的，因此 n 个站点都有同样可能接收到对象，所以负载在站点之间是均匀的。当然，Rendezvous 哈希允许一些站点

具有和其他站点不同的容量，例如一个容量是其他站点容量的两倍的站点只需要在列表中显示两次即可

- 高命中率：由于所有客户端都同意将对象放入到同一站点，因此每次将对象提取出来或者放置的命中率是非常高的
- 中断最少：当站点出现故障的时候，只需要重新映射到该站点的对象即可完成故障处理
- 支持分布式 $k-$ 协议：客户端只需要简单地选择出分数排名靠前的 k 个站点即可

现在来详细介绍一下上述特点“负载均衡”中某些站点具有不同容量的情况，这种情况被称为加权变化。在 Rendezvous 哈希的标准实现中，每个站点都接收静态相等比例的键。然而，当站点具有不同的处理或保存分配给它们的键的能力时，这种行为是不受欢迎的。例如，如果其中一个节点的存储容量是其他节点的两倍，那么如果算法可以考虑到这一点，则这个更强大的节点将收到的键数量是其他节点的两倍，这将是有益的。处理这种情况的一种直接机制是为该站点分配两个虚拟位置，这样如果较大站点的虚拟位置中的任何一个具有最高哈希值，该节点就会收到密钥。但是当相对权重不是整数倍时，这种策略就不起作用了。例如，如果一个节点的存储容量增加了 42%，则需要按不同比例添加许多虚拟节点，从而导致性能大幅下降，现在已经有了一些比较新的工作在研究如何处理这种情况。

3.1.2 和 Karger 哈希的比较

Karger 哈希通过将站点均匀随机地映射到称为令牌的单位圆上的点来运行。对象也被映射到单位圆并放置在其令牌是第一个遇到的站点中，从对象的位置顺时针方向移动。当一个站点被移除时，它拥有的对象将转移到拥有下一个顺时针移动的令牌的站点。如果每个站点都映射到大量令牌，这将在其余站点之间以相对统一的方式重新分配对象。

如果通过散列站点 ID 的 200 个变体将站点随机映射到圆上的点，则任何对象的分配都需要为每个站点存储或重新计算 200 个散列值。然而，与给定站点关联的标记可以预先计算并存储在排序列表中，只需要对对象应用一次散列函数，然后进行二分搜索来计算分配。然而，即使每个站点有许多令牌，一致性哈希的基本版本也可能无法在站点上均匀地平衡对象，因为当一个站点被删除时，分配给它的每个对象只会分布在与该站点具有令牌一样多的其他站点上。

相比之下，Rendezvous 哈希无论在概念上还是实践上都要简单许多，我们只需要给定统一的哈希函数以及评分函数即可开始工作。Rendezvous 哈希也不需要像 Karger 哈希那样预先计算或存储令牌，这不仅有助于性能提高，而且减少了系统出错的可能性。

事实上，我们可以认为 Karger 哈希是 Rendezvous 哈希的一个 $k = 1$ 的特例，只需要将哈希函数和评分函数进行定制化修改即可。

3.1.3 Rendezvous 哈希的变体

当站点规模 n 非常大时，我们可以使用一种基于“骨架”的变体来进行处理。这种方法创建了一个虚拟的层次架构，并将复杂度优化到了 $O(\log n)$ 级别，大幅提高了性能。

该变体的主要思路是：首先选择常数 m 并将 n 站点分为 $c = n/m$ 个集群 $C_1 = \{S_1, S_2, \cdots, S_m\}$、$C_2 = \{S_{m+1}, S_{m+2}, \cdots, S_{2m}\}$ 等，然后构建层次结构，并将这些集群放置在树的叶子节点（虚拟节点）上，并且给每个虚拟节点进行编号，然后可以依据这些编号进行哈希，从而可以很快地找到目标位置。其中 m 可以根据预期故障率和所需负载平衡程度等因素进行选择，更大的 m 的代价是更高的搜索开销。

在 2005 年，Christian Schindelhauer 和 Gunnar Schomaker 还描述了一种重新加权哈希分数的对数方法，这种方法在节点权重发生变化或添加或删除节点时不需要负载因子的相对缩放。这使得在对节点进行加权时具有完美精度和完美稳定性的双重好处，因为只需要将最少数量的键重新映射到新节点。

3.2 Jump Consistent Hash

Jump Consistent Hash 是由 John Lamping 和 Eric Veach 发明的一致性哈希算法，该算法与 2014 年首次发表在论文《A Fast, Minimal Memory, Consistent Hash Algorithm》中。这个算法精简到可以用几行代码来描述，相比传统的环形一致性哈希，Jump Consistent Hash 的空间复杂度更低，且没有内存占用。

3.2.1 一致性和均匀性

通常而言，当给哈希增加一个新桶时，需要对已有的键进行重新映射。一致性（monotonicity）是指在这个重新映射的过程中，键要么保留在原来的桶中，要么移动到新增加的桶中。如果键移动到原有的其他桶中，就不满足一致性要求。这也正是一致性哈希区别于传统哈希的特征，传统的哈希在增加一个新桶时，一般会对键进行随机重新的随机映射，键很可能移动到其他原有的桶中。

均匀性（balance）是指键会等概率地映射到每个桶中，不会出现某个桶里有大量键、某个桶里键很少甚至没有键的情况。这是一致性哈希和传统哈希都有的特征，因为这是由哈希函数的特征来保证的。

这两个特征是 Jump Consistent Hash 最主要的设计目标，因此，计算当桶数量变化时究竟有哪些输出需要发生相应的变化就是非常重要的。

3.2.2 推导过程

先来考虑如何满足一致性要求。假设一致性哈希函数为 $h(key, n)$，其中 key 为要放入元素的键值，n 为桶的个数，函数返回的是给 key 分配的桶的序号（序号范围是 $[0, n-1]$，那么为了满足一致性要求，有：

$$h(key, n) = 0$$

$$h(key, n) = \begin{cases} h(key, n-1) & \text{扩容时}key\text{仍然在原桶中} \\ n-1 & \text{扩容时}key\text{移动到新增加的桶中} \end{cases}$$

根据这个递归公式，我们可以很自然地想到如下两个方法来满足一致性要求：

1. 直接使用这个递推公式。开始桶的总数为 1，所有的 key 都放在第 0 个桶中。然后每增加一个桶，生成一个随机数，当这个随机数为奇数时，将 key 放在保持在原始桶中，当这个 key 为偶数时，将 key 移动到新增的桶中
2. 开始桶的总数为 1，所有的 key 都放在第 0 个桶中，同时生成一个大于当前桶数的随机数。每增加一个新桶时，判断当前桶总数是否超过这个随机数。如果未超过（桶数小于或等于这个随机数），则将 key 保留在原来的桶中；如超过，则将 key 移动到新增加的桶中，同时重新生成一个大于当前桶数的随机数，后续增加新桶时，使用和前面相同的逻辑进行判断

但是，在第一个方法中，如果桶的总数为 3，则 key 被分配到 0 号桶的概率为 $\frac{1}{4}$，很明显不符合均匀性；在第二个方法中如果桶的总数为 2，则 key 被分配到 0 号桶的概率接近 1，因此也不满足均匀性。

也就是说，上面两个思路虽然可以满足一致性要求，但是无法满足均匀性要求。我们可以对其进行修改使其满足均匀性。对于第一个方法，修改为：如果当前桶数为 k 且新增一个桶，那么 key 移动到新桶的概率为 $\frac{1}{k+1}$，这样算法就可以满足均匀性，这是因为此时每个 key 被分配到每个桶的概率仍是 $\frac{1}{n}$。

对于第二个方法，问题关键在于随机数生成，我们可以对随机数的生成规则进行限定，从而使最后的实现满足均匀性要求。具体而言，在论文中给出了这样的随机数生成规则：假设上一次 key 跳变的桶的序号为 b、也即 $h(key, b+1) = b$，那么生成的随机数 $f = (b+1)/r$（其中 r 是区间 $(0, 1)$ 中的均匀随机浮点数），使用这个随机数 f 就可以使最后的实现符合均匀性要求。该思路的代码实现正是论文中的

```
----------------------------JCH 哈希 BEGIN-------------------------
int32_t JumpConsistentHash(uint64_t key, int32_t num_buckets) {
        int64_t b = -1, j = 0;
        while (j < num_buckets) {
                b = j;
                key = key * 2862933555777941757ULL + 1;
                j = (b + 1) * (double(1LL << 31) / double((key >> 33) + 1));
```

```
        }
        return b;
}
----------------------------JCH 哈希 END--------------------------
```

其中 JumpConsistentHash 是一个一致性哈希函数，它把一个 key 一致性地映射到给定几个槽位中的一个上，输入 key 和槽位数量 num_buckets，输出映射到的槽位标号。

现在我们已经解决了一致性和均匀性要求，接下来考虑具体实现中一些问题。当 $n = 1$ 时，所有的 key 都要映射到一个槽位上，函数返回 0，即 $h(key, 1) = 0$。当 $n = 2$ 时，为了映射的均匀，每个槽要映射到 $K/2$ 个 key，因此需要 $K/2$ 的 key 进行重新映射。以此类推，当槽位数量由 n 变为 $n+1$ 时，需要 $K/(n+1)$ 个 key 进行重新映射。那么，每次需要重新映射多少份，才可以保证映射均匀？而且，哪些 key 要被重新映射呢？也就是说在新加槽位的时候，要让哪些 key 跳到新的槽位、哪些 key 又留在老地方不动呢？这里我们就可以用到前面所述第一个办法的思路了，移动到新桶的概率为 $\frac{1}{k+1}$，每次使用一个随机数来进行判断，代码为：

```
--------------------------随机数判断 BEGIN----------------------
int ch(int key, int n) {
        random.seed(key);
        int id = 0;
        for (int j = 1; j < n; j++)
        if (random.next() < 1.0/(j+1))
                id = j;
        return id;
}
--------------------------随机数判断 END--------------------------
```

其含义为：对每个 key，用这个 key 做随机数种子，得到一个关于 key 的随机序列。为了保证槽位数量由 j 变为 $j+1$ 时由 $1/(j+1)$ 占比的数据会跳到新槽位 $(j+1)$，可以用如下的条件来决定是否重新映射：如果 $random.next() < 1/(j+1)$ 则跳，否则留下。

3.2.3 Jump Consistent Hash 缺陷

虽然 Jump Consistent Hash 在很多方面都很好，但它也在一些方面具有一定的缺陷：

- 不支持设置哈希桶的权重，这点不如其他哈希方法
- 仅能在末尾增加和删除桶，不能删除中间的哈希桶，也就是说，如果节点数会随机减少（比如 web caching 场景下服务器节点随机故障），那么假设节

点 $i(0 <= i < num\_buckets)$ 故障，此时如果简单地把 $num\_buckets$ 减 1，就会导致编号在 i 之后的节点上的数据全部错位到前一个节点。因此 Jump Consistent Hash 不适合节点随机故障的场景

我们可以采用增加一个中间层的方法来解决这两个问题：增加一层虚拟桶，使用 Jump Consistent Hash 来将 key 分配到虚拟桶中，然后在虚拟桶和实际桶之间建立一个映射关系。这样我们就可以通过映射关系来设置实际桶的权重；也可以在任意位置删除和添加实际桶，只需要维护好映射关系即可。当然，这样做的代价就是，算法本来可以的无内存占用的，现在需要有一块内存来维护映射关系了。

3.3 Maglev 哈希

Maglev 是 Google 开发的基于 kernal bypass 技术实现的 4 层负载均衡，它具有非常强大的负载性能。Maglev 在负载均衡算法上采用自行开发的一致性哈希算法被称为 Maglev Hashing。

Maglev 哈希的基本要求就是使映射尽量均匀，并尽量把槽位变化时的映射变化降到最小（即尽量避免全局重新映射），可以发现，这和 Jump Consistent Hash 的均匀性和一致性要求在表述上非常相似。

Maglev 哈希的主要内容是：

1. 为每个槽位生成一个偏好序列，尽量均匀随机
2. 建表：每个槽位轮流用自己的偏好序列填充查找表
3. 查表：哈希后取余数的方法做映射

先来看 Maglev 哈希如何建立查找表（lookup table）。先新建一张大小为 M 的待填充的空表 entry。并为每个槽位生成一个大小为 M 的序列 permutation，称为“偏好序列”。然后，按照偏好序列中数字的顺序，每个槽位轮流填充查找表。将偏好序列中的数字当做查找表中的目标位置，把槽位标号填充到目标位置。如果填充的目标位置已经被占用，则顺延至该序列的下一个。

在有了表之后，Maglev 哈希可以很容易地找到对应的资源，只需要对输入 key 做哈希再取余（即经典的模算数哈希函数），即可映射到表中一个槽位。也就是说，当输入一个 key 时，映射到目标槽位的过程就是 $entry[hash(k)\%M]$。

现在来处理一个剩下的问题：偏好序列 permutation 是如何确定的？因为我们需要使用这个偏好序列作为填表时的目标位置，因此偏好序列的确定方法可以参考哈希思路，选取两个无关哈希函数 h_1 和 h_2，假设一个槽位的名字为 b，则用这两个无关哈希函数计算出偏移量 offset 和跳跃量 skip

$$\text{offset} = h_1(b)\%M$$
$$\text{skip} = h_b(b)\%(M-1)+1$$

然后对每个 j 即可计算出偏好序列 permutation 中的所有数字，计算原理为：

$$\text{permutation}[j] = (\text{offset} + j \times \text{skip})\%M$$

可以看到，这是一种类似二次哈希（Double Hashing）的方法，使用了两个独立无关的哈希函数来减少映射结果的碰撞次数，提高随机性。但是，生成偏好序列 permutation 其实也可以采取其他的计算方法，但无论何种方式，目的都是一样的，生成的偏好序列满足随机性和均匀性。

4 结语

随着互联网系统日益复杂，高效的存储和快速的访问概念变得越来越重要，一致性哈希（Consistent Hashing）是一个非常重要的概念，它可以提供基于哈希表的存储和查找服务。一致性哈希技术有效地提高了网络数据服务的性能、增加了系统的鲁棒性，这使其实际性能表现超出了我们的预期！

在本文中，我简要梳理了一致性哈希技术的一些常见内容，感觉收获颇丰，同时又惊叹于计算机工业界实践成果之富，这值得我们持续不断地学习与探索！