Memcached 的分析与改进*

李首扬[1] 杨 仝[1] 代亚非[1] 黄 亮[2] 郑廉清[3]

([1]北京大学 计算机系 北京 100871 [2]94782部队65分队 杭州 310021
[3]西京学院控制工程系 西安 710123)

摘要：Memcached是一种高性能分布式内存对象缓存系统，使用十分广泛；其设计目的为缓存数据库内容以加速动态Web请求，但也广泛应用于其他高性能存储，并且成为了内存Key－Value数据库的经典代表。本文对Memcached的系统结构、处理流程进行了分析，包括网络处理、哈希表的查询与维护、内存的分配与管理、冷数据的置换算法等；同时以哈希表、内存管理这两个影响性能的关键点入手，结合了Cuckoo哈希与CLOCK置换算法两种更易并行化的的算法，对Memcached现有的结构提出了较大的改动，以期提高上述处理速度，实现整体性能的提升。

关键词：内存缓存系统，Memcached，哈希表，内存管理，缓存置换算法

Analyzing and improving memcached

LI Shouyang[1], YANG Tong[1], DAI Yafei[1], HUANG Liang[2], ZHENG Lianqing[3]

([1]Department of Computer Science, Peking University, Beijing, 100871, China,
[2]65 Otryad, 94872 PLA troops, Hangzhou, 310021, Chinaa,
[3]Department of Control Egineering, Xijing University, Xi'an, 710123, China)

Abstract: Memcached, a widely used memory object caching system, is a typical case of in memory key－value storing or caching systems. This paper analyzed the architecture of memcached, including network processing, structure of the hash table, managing memory allocation, and the cache replacement algorithm. Focusing on two fields: the hash table structure, and the memory management politics, which have a significant impact on the performance and scalability of the system, this article introduces new algorithms, improving the performance of the parallelizing memory caching system. In this article, we combine Cuckoo hashing, a hashing scheme optimized for parallel accessing and scenes when read requests are greatly more than write requests; and a cache replacement algorithm, the clock algorithm, to archive higher processing rate, with an acceptable loss of cache hit rate.

Keywords: memory cache system, memcached, hash table, memory management, cache replacement algorithm

1 引言

Memcached[1]是一种应用十分广泛的内存对象缓存系统，其设计目的为缓存数据库内容以加速动态Web请求，但也广泛应用于其他方面。作为内存Key－Value存储系统的代表，Memcached采用的“使用哈希表查找对象，并通过自行管理内存提高使用效率”的整体框架被各种Key－Value存储系统使用，成为了内存

本文于2015－08－19收到。
*基金项目：国家自然科学基金重点项目（编号：61232004），国家“973”课题（编号2014CB340400），国家重点研发计划课题（2016YFB1000300），国家自然科学基金（61672061），中国科学院声学研究所合作项目“分布式缓存客户端定制开发”。

存储的范例。

本文对 Memcached 的结构进行了分析，着重研究了哈希表查询与管理、内存的分配管理这两大方面，并对这两方面进行了比较大的改动，引入新技术提升整个系统的效率与性能。

2　Memcached 的主要模块

Memcached 在启动时创建多个线程，并行处理请求；同时有单独的 LRU（Least Recently Used）对象管理线程与内存维护线程。整个系统的运行流程如图 1 所示。

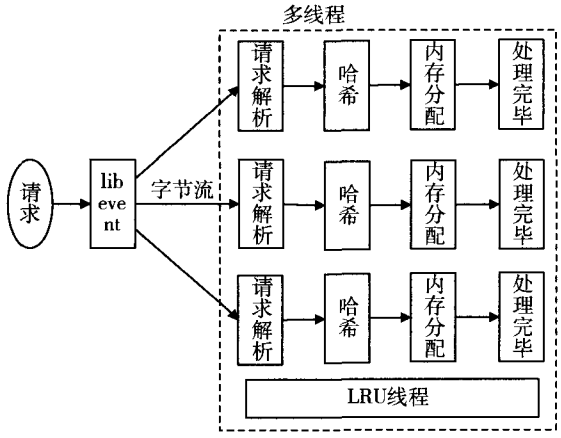

图 1　Memcached 的整体结构与处理流程

2.1　I/O 处理与请求解析

Memcached 使用 libevent[2] 库处理网络请求。libevent 是一个开源的异步事件处理库，它提供了一组 API，让程序可以设定在 I/O 事件发生时，调用一定的函数处理请求内容。libevent 支持 epoll 等 IO 监控接口，提供了较高的性能。根据设置，Memcached 启动若干个独立线程，然后将它们共同绑定到由 libevent 管理的网络端口或文件管道，以实现多线程接收并处理同一端口请求的目的。当接受到请求，网络处理模块解析其中内容，转换为 Memcached 内部数据结构，并调用对应函数进行处理。此系统较为成熟，并且为第三方提供，研究多集中在将其整体更换为硬件优化的网络处理库等，与内存存储系统的特性也关系不大，本文不加以详细研究。

2.2　哈希表查询与修改

Memcached 使用哈希表查找请求的键值。哈希算法默认为 jenkins 算法[3]，可更换为 murmur3 算法[4]；当发生冲突（两个对象的哈希值相同）时，使用链式哈希处理：在哈希值对应位置创建一个链表，链表中记录哈希值相同的所有对象，查找时逐一比对。Memcached 对哈希表主要做了两个特殊处理：

（1）大小可变的哈希表。Memcached 的哈希表大小为 2 的整数次幂，便于在计算机中保存与处理；当哈希表被“填满”，大于一定阈值时，Memcached 会重建一个容量为两倍的表，将所有键值根据新的表大小重新计算哈希值，并将对象存入新表。

（2）哈希表分区锁。当对哈希表内容进行修改时，可能会出现多线程竞争问题；使用全局锁将造成性能瓶颈，对每个表项单独加锁会浪费大量内存、当进行重建时也可能会严重影响效率。

如图 2，Memcached 将哈希表按照键值分为若干个区，每个区中的对象共用一个锁；由于线程数量远小于“区”数量，分区锁与表项单独加锁性能差距并不大，同时也减小了锁占用的空间和时间。在实践中，根据线程数量，分区数目为 1024 至 8192 不等，并为 2 的整数次幂，这样“判定哈希表项所属区”的“二次哈希”只需单次位运算即可完成，保证了性能。

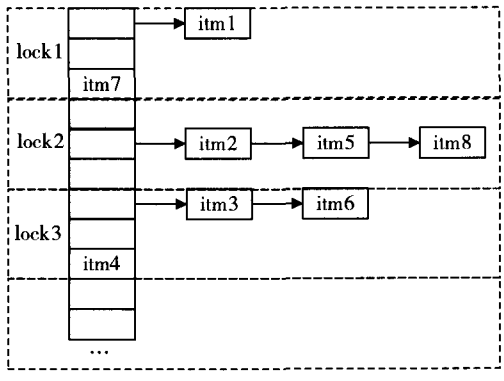

图 2　Memcached 的拉链法哈希表与分区锁

2.3　内存分配与管理

应用中，读写的对象大小不一，并且读写十分频繁，对内存管理提出了很高的要求。系统需要快速分配不同大小的内存，同时在大量的修改与删除之后避免内存碎片化，保证高利用率。

如图 3，Memcached 将内存划分为若干页（page），每页大小为 1MB；每页分为若干个大小相同的片（slab），片的大小从初始值开始，以指数增长，下一级的片大小为上一级的 1.25 倍，直至最大的片占满整页。

在分配内存时，系统将申请的内存大小“向上取整”至最近的一个片大小，并取出一个空闲片；释放内存时，整片内存回到系统中；当某种大小的片全部被分配时，则尝试申请一个整页并划分成需要的大小。这种分配方法以少量的内存浪费（不超过 20%）为代价，将大小千差万别的对象统一到了不到 20 个阶层，阻止了内存碎片化的产生。

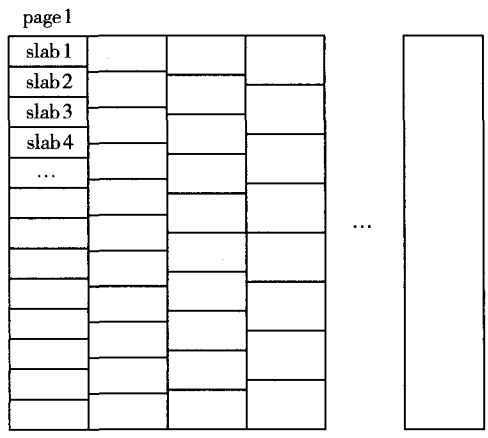

图 3 Memcached 的内存分片结构

2.4 内存缓存置换

以 LRU 算法为代表的缓存置换算法是 Memcached 等缓存系统的核心之一，由于内存容量有限，系统需要适时丢弃访问频次较低的对象（或移动至读写速度较慢的外存）。如何选择被丢弃的对象以提高缓存命中率，同时又尽量降低维护对象热度的开销，一直是研究的热门话题。Memcached 对不同大小的“片”独立处理，分别独立进行缓存置换，由内存管理部分来协调“阶层”之间的内存分配。Memcached 使用两种缓存置换算法，一种是“标准 LRU”，一种我们暂且命名为“分层近似 LRU”。

（1）标准 LRU。标准 LRU 是简单明确的实现了严格的 LRU（最近最少使用）算法，丢弃的对象始终为最近一次访问距今时间最久的对象。对每个片大小的“层级”，Memcached 维护了一个双向链表，将所有对象连在一起；当一个对象被访问时，将其从链表中“弹出”，然后再加入链表头。这样，整个链表按照最近访问次序排序，当需要丢弃对象时，直接从链表尾部选取对象，根据其属性进行处理即可。

（2）分层近似 LRU。此算法是 Memcached 新版本中引入的可选算法。它并不是严格的 LRU 算法，而是在上述标准 LRU 的基础上做了一些修改和简化。

标准 LRU 无论读还是写，都会触发链表操作，将对象移至链表最前；在 Memcached 的实际应用中，对数据的访问均为读多写少，同时访问频率明显不均衡，大部分访问集中在少数对象；这种情况下，少部分热对象会被频繁移动，拖累整个系统的性能。

Memcached 首先去掉了所有读请求的链表移动操作，仅会更新对象的最近访问时间，由扫描链表尾部的回收函数判定对象并非“冷对象”而重新加入链表；实际应用中，这一步便可剩下大量链表维护操作。

简单的忽略读请求会使得 LRU 链表中热度混杂，失去其本来的意义；为了与其配合，同时也进一步优化性能，Memcached 如图 4 将每个“片层级”的链表拆分为热（HOT）、温（WARM）、冷（COLD）三个层级，分别保存三种热度的对象，在置换对象时，对三个表分别处理。创建新对象时，新对象均会进入热表，这里的对象会接受较少的移动，以尽量提升少数热对象的访问性能；而访问频次较低的对象并不会直观的从热到温再到冷，而是无论原先在热表还是温表，均会被移动至冷表，对象一旦“变冷”，就只能变“温”而无法重新进入热表，而在这两个表内对象将会被更大概率的被移出或丢弃。

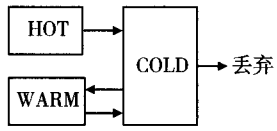

图 4 热度分层 LRU 的移动示意

这种算法缺点十分显而易见：一个对象永远不会从“冷”回“热”，当它被频繁访问时可能会降低性能；但从实际应用出发，Memcached 面对的热数据大多数为新数据，此算法能避免被偶然访问的冷数据与热数据争抢资源，实际性能得到了提升。

3 对哈希表和内存管理模块的改进

哈希表被广泛应用于多个领域，包括路由表查找[5,6]，布隆过滤器[7,8]等。哈希表的查询与维护、内存的分配与管理（包括对象置换）是内存缓存系统的核心，这两方面的算法设计直接影响了整个缓存系统的性能。如上文所述，Memcached 在这两方面采用的是较为简单成熟的方案，随后根据实际应用进行一些变通与

优化；这样的架构有着很大的提升空间。一直以来，学术界都在对这些方面进行改进，根据不同的情景提出新的算法。针对实际应用场景中，读多写少，并且硬件高度并行化的特点，本文结合 Cuckoo 哈希与 CLOCK 置换算法，提出了在高并发场景下提升 Memcached 性能的解决方案。

3.1　哈希表 - Cuckoo 与 DCuckoo

Memcached 的哈希表结构虽然哈希函数本身性能优秀，但仍然是单哈希函数、拉链法处理哈希冲突的传统哈希表；其分区锁机制也只能说是针对高并发的变通优化，并未能利用如今高度并行化的硬件。

本文直接更换哈希表结构，思想是利用应用读多写少的特性，将查找操作尽量并行化处理，考虑取舍牺牲一定修改操作的性能。选择的结果是 DCuckoo 哈希[9]：

DCuckoo[10] 哈希由 Cuckoo 哈希演变而来。Cuckoo 哈希使用多个独立的哈希函数，对每个请求的键值计算出这独立的若干个函数，查找对象时，查询所有函数位置；对读操作，任意一个函数位置命中则为命中，全部未命中则为查找失败；对写操作，有任意位置为空则插入，当各位置都已满无法插入新对象时启动“踢”算法：如图 5，有两个哈希函数，每个对象引出的箭头指向对象另一个哈希函数的位置，左图新插入的对象所对应位置分别被 A 与 C 占据；对象“踢走”A，使 A 到自己另一个哈希函数的 B 位置；而 A 递归踢走 B，最终形成了右图的结果。关于哈希函数的数量、“踢走”哪个对象的选择，都是可定义内容。Cuckoo 哈希的一个缺点是它的“踢”操作可能会陷入死循环或者很难得到结果，在应用中一般是设置若干次（如 500）次“踢”操作还未插入，则认为本次插入失败；对于缓存系统，此时也可以考虑将路径上某一个对象直接置换丢弃掉，从而保证新对象插入的成功，两个选择都是可以接受的；DCuckoo 则是结合了 Cuckoo 哈希与另一种并行哈希算法，D - left 哈希；D - left 中，哈希表被分为 D 个子表，每个表有各自的哈希函数；插入时，在 D 个子表中选择最“空”的表插入；查询时，则是并行查询。DCuckoo 结合了两个哈希算法的特点，D 个子表均可实行 Cuckoo 的“踢走”操作，实现了更高的哈希表装载率与更好的并行度。

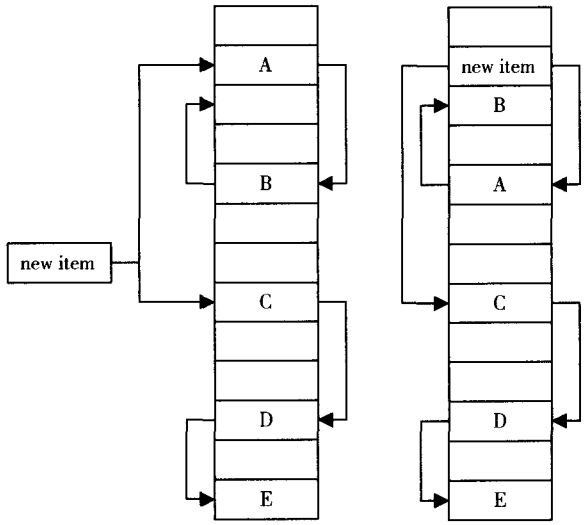

图 5　Cuckoo 哈希插入示意图

3.2　内存管理 - CLOCK 算法

Memcached 采用的按大小分级、随后采取进行标准或修改版 LRU 方法可以提供较好的性能，但对要求极高并发，而不太要求数据持久性的内存缓存系统而言还有很大的提升空间。

一个重要思想就是“近似 LRU”，并不严格维护 LRU 关系，而是通过一些算法，尽可能让被丢弃对象是系统里较老的对象。实践中著名例子是 Redis[11]：其并不维护节点的 LRU 顺序关系，而是从整个内存中随机抽取一定数量的对象，从中选择“最旧”的对象并丢弃。这种随机化算法理论效果会差很多，但在实际测试表现中，其与 LRU 的命中率并没有非常大的区别，而其维护简便的优点可以带来明显的性能提升。

在近似 LRU 算法中，一个代表是操作系统内存管理经常使用的 CLOCK 算法[12,13]：内存以块为单位进行分配与释放，并不维护 LRU 的链表，而是为每一个块维护一个“热度”的参数；当对一个块内对象进行访问时，“热度”提升，而下降由一个单独的“指针”线程完成。指针线程就像钟表里的指针一样，不断地扫过每个区块，对其指向的块进行“降温”操作；当一个块降到“冰点”以下，那么它就可以被回收。

3.3　结合新算法的流程与优势

使用 Cuckoo 哈希与 CLOCK 近似 LRU 算法后，Memcached 的性能可以得到提高，可并发程度有明显的改善。

首先,Cuckoo 哈希的多哈希算法相互独立,可以做到完全并行;无论是在多核分别运行,还是在 GPU 等众核平台分块计算,都十分方便移植应用;之后,对每一个哈希结果都只有一次访存与判断,没有任何链表之类的扫描操作,使得查询时间严格达到 O(1),在 SIMD(单指令流多数据流)类并行系统上可以高效运行。

在哈希表查询到对象之后,CLOCK 算法免去了对 LRU 哈希表的繁琐维护与整个链表加锁的性能瓶颈,对数据热度的维护简化为每个数据块独立的变量加减,与单独线程的内存扫描,在最大程度上减少了系统并行化的瓶颈,易于扩展到更多的线程数量。

4 结束语

Memcached 作为内存缓存系统的代表,结构清晰,算法较为传统,可提供稳定的服务,但面对当前硬件发展及并发访问要求的提高,其架构已经遇到瓶颈难以提升。本文以 Memcached 为样本分析了内存缓存系统的结构和操作流程,同时对哈希表与内存管理两个核心模块使用替代算法进行优化,提高了系统的并发性能。

参 考 文 献

[1] Dormando. memcached - a distributed memory object caching system [OL]. 2016 - 7 - 13/2016 - 7 - 24. https://memcached.org.

[2] Provos N, Mathewson N. libevent [OL]. http://libevent.org/. 2014 - 01 - 05. 2016 - 07 - 24. http://libevent.org/

[3] Jenkins B. Hash functions[J]. Version 19/02/15. https://code.google.com/p/smhasher/wiki/MurmurHash3.

[4] Appleby A. murmurhash3[J]. 2011.

[5] T. Yang, R. Duan, et al.. CLUE: Achieving fast update over compressed table for parallel lookup with reduced dynamic redundancy. in Proc. IEEE ICDCS, 2012.

[6] Yang T, Xie G, Li Y B, et al. Guarantee TP lookup performance with FIB explosing[J]. ACM SIGCOMM Computer Communication Review, 2015, 44(4): 39 - 50

[7] Yang, Tong, Liu, Alex X, Shahzad, Muhammad et al. A shifting bloom filter framework for set queries[C].//Proc of the the VLDB Endowment. New York: ACM, 2016, 9(5): 1 - 13

[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors[J]. Communications of the ACM, 1970, 13(7): 422 - 426

[9] Pagh R, Rodler F F. Cuckoo hashing[C]//European Symposium on Algorithms. Springer Berlin Heidelberg, 2001: 121 - 133

[10] 蒋捷, 杨仝, 张梦瑜, 代亚非, 黄亮, 郑廉清. DCuckoo:基于片内摘要的高性能哈希表[C]. 第 22 届全国信息存储技术学术会议. 2016.

[11] Redis Labs. Using Redis as an LRU cache [OL]. 2016 - 7 - 24/2016 - 7 - 24. http://redis.io/topics/lru - cache.

[12] Corbato F J. A paging experiment with the multics system[R]. MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[13] Fan B. Andersen D G, Kaminsky M. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing [C]//NSDI 2013: 371-384

作者简介

李首扬,(1994.4 -),男,硕士研究生,研究方向:分布式系统。

通信作者:杨仝,北京大学计算机系,助理研究员。