参考：<http://blog.csdn.net/lz0426001/article/details/52634772>

**一、缓存概述**

**1.1 什么是缓存**

     缓存在wiki上的定义：用于存储数据的硬件或软件的组成部分，以使得后续更快访问相应的数据。缓存中的数据可能是提前计算好的结果、数据的副本等。典型的应用场景：有cpu cache, 磁盘cache等。本文中提及到缓存主要是指互联网应用中所使用的缓存组件。

**1.2 为什么引入缓存**

    传统的后端业务场景中，访问量以及对响应时间的要求均不高，通常只使用DB即可满足要求。这种架构简单，便于快速部署，很多网站发展初期均考虑使用这种架构。但是随着访问量的上升，以及对响应时间的要求提升，单DB无法再满足要求。这时候通常会考虑DB拆分(sharding)、读写分离、甚至硬件升级(SSD)等以满足新的业务需求。但是这种方式仍然会面临很多问题，主要体现在：

* 性能提升有限，很难达到数量级上的提升，尤其在互联网业务场景下，随着网站的发展，访问量经常会面临十倍、百倍的上涨。
* 成本高昂，为了承载N倍的访问量，通常需要N倍的机器，这个代价难以接受。

  单纯从DB层进行优化已经无法满足需求，那是否有更好的方式来解决这个问题呢？首先我们来看图1的数据，

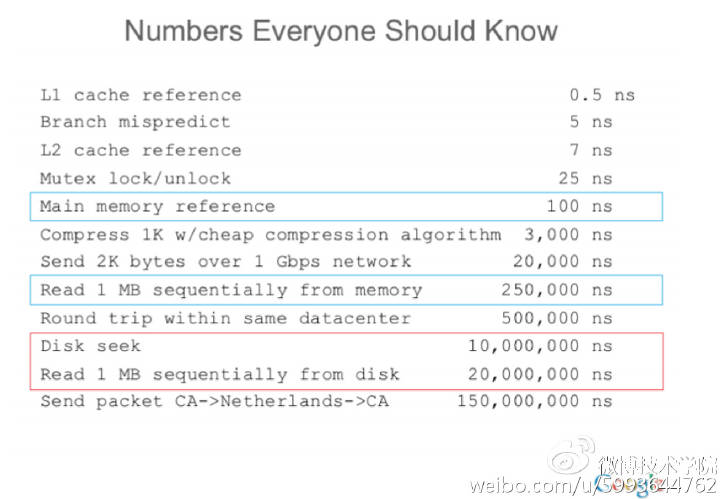


图1 不同操作的耗时

L1🡪main memory🡪disk

1🡪100🡪10,000,000倍数 1纳秒🡪100纳秒🡪10毫秒

​​    ​从数据上很容易看出，内存的访问性能明显优于磁盘。把数据放入内存中，可以提供更快的读取效率。但在互联网业务的场景下，将所有数据数据都装入内存，显然是不明智的。从图2的金字塔模型可以看到，从机械硬盘，到SSD硬盘，再到内存，速度越来越快，价格越来越贵，单位容量的价格也越来越高。所以，不可能把数据全部装入内存。

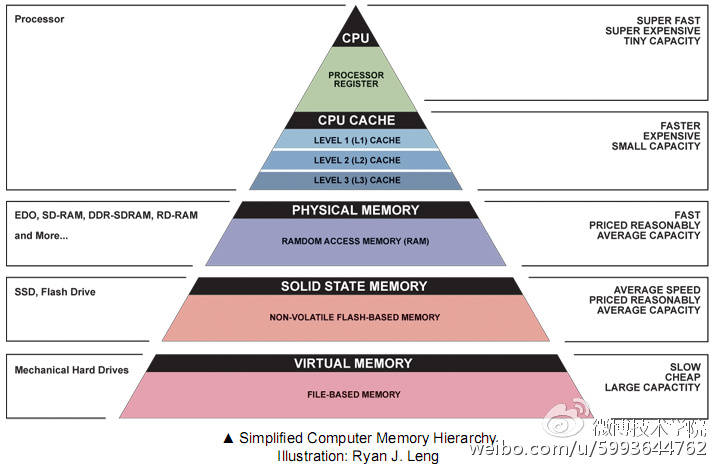


图2 不同存储介质性能、容量、价格的对比

​    同时大部分的业务场景下，80%的访问量都集中在20%的热数据上(适用二八原则)。因此，通过引入缓存组件，将高频访问的数据，放入缓存中，可以大大提高系统整体的承载能力，原有单层DB的数据存储结构，也变为Cache+DB的结构，如图3。

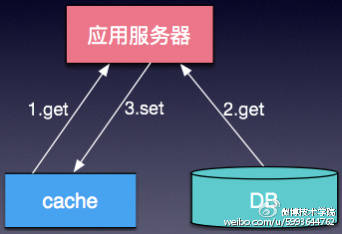


图3 Cahce+DB的架构模型

​    在数据层引入缓存，有以下几个好处：

* 提升数据读取速度
* 提升系统扩展能力，通过扩展缓存，提升系统承载能力
* 降低存储成本，Cache+DB的方式可以承担原有需要多台DB才能承担的请求量，节省机器成本

    根据业务场景，通常缓存有以下几种使用方式

* 懒汉式(写时触发)：写入DB后, 然后把相关的数据也写入Cache
* 饥饿式(读时触发)：先查询DB里的数据, 然后把相关的数据写入Cache
* 定期刷新：适合周期性的跑数据的任务，或者列表型的数据，而且不要求绝对实时性

*iiiiiiiiii*

**​二、常用缓存介绍**

**2.1 缓存分类**

    缓存大致可以分为两类，一种是应用内缓存，比如Map(简单的数据结构)，以及EH Cache(Java第三方库)，另一种就是缓存组件，比如Memached，Redis。

**2.2 Memcached简介**

**2.2.1 概述**

Memcached是开源的，高性能的，可分布式部署，用于网站提速，减轻数据库负载的缓存组件，有如下特点：

* 高性能Key-Value存储
* 协议简单：简单文本协议、二进制协议
* 支持数据过期
* LRU剔除算法
* 多线程
* slab内存管理
* 客户端实现分布式

**2.2.2 内存管理**

    Memcached使用Slab Allocator的机制来实现分配、管理内存，按照预先规定的大小，将分配的内存分割成特定长度的块，以此来解决内存碎片问题，如图4：

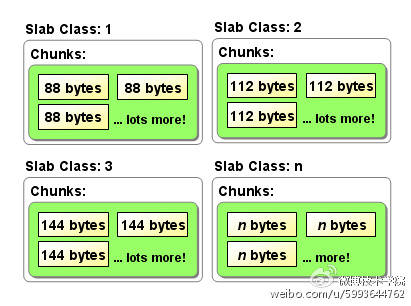


图4 Slab Allocator的机制

​slab是特定大小的chunk的组，并根据增长因子(factor)，划分成不同slab class。分配内存时，每次分配一个page(默认1MB)给某个slab，并根据slab的大小，切分成多个chunk，数据存储在chunk中。Memcached根据收到的数据的大小，选择最适合数据大小的slab，如图5。

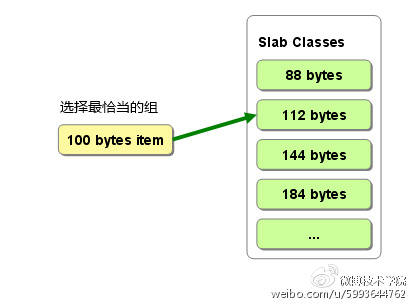


图5 slab class选取

​    Memcached的Slab Allocator内存分配方式会存在内存浪费的问题，主要有以下几种情况：

* chunk浪费: 缓存数据没有填充满chunk
* page浪费: 一个page的容量不能被slab的大小整除。
* slab浪费: 某个slab的内存没有完全被利用，只存储了少量数据，却占用一个page。

**2.2.3 剔除算法**

    Memcached采用LRU淘汰算法，在容量满的时候进行数据剔除。不过该淘汰算法只在Slab内部进行，也就是说，某个Slab容量已满时，只会在该Slab内部进行数据剔除，而不会影响其它Slab。这种局部剔除的策略也带来了一个问题：Slab钙化。

**2.2.4 Slab钙化**

某个Memcached实例的Slab状态如下：



​如果此时，业务数据大小出现变更，比如超过152字节。就会出现以下问题：



​​    新的数据(需要使用chunk size为192)，只能最多使用500M，而原有slab class(120和152)没释放，尽管数据都已过期，因为淘汰策略是淘汰相同slab class的数据，所以一直利用不上120和152的内存，这种情况会导致缓存命中率急剧下降。

如果发生这种情况，有以下几种解决方案：

* 重启Memcached实例

    简单粗暴，需要避免单点问题，避免出现雪崩

* 随机过期

     过期淘汰策略也支持淘汰其他slab class的数据，twitter和facebook等均作了类似支持

* 通过slab\_reassign、slab\_authmove

    官方1.4.11版开始支持此功能

**2.3 Redis简介**

Redis是开源的，高性能的，支持分布式，支持多数据结构的缓存组件。特点如下

* 高性能Key-Value存储
* 丰富的数据结构：string、list、hash、set、zset、hypeloglog
* 支持数据过期：主动过期+惰性过期
* 支持多种LRU策略：volatile-lru、volatile-ttl 等
* 内存管理：tcmaloc、jemalloc
* 内存存储+磁盘持久化: rdb、aof
* 支持主从复制
* 单线程

     微博使用的是内部定制的redis版本，支持热升级、基于aof的增量复制、新数据类型longset和bloomfilter等特性。但在微博的场景下，redis更多被用存储和计数器场景，缓存主要以memcached为主，因此这里不再过多介绍redis。

*iiiiiiiiii*

**三、分布式缓存实现**

**3.1 概述**

    构建大型互联网系统会面临很多的挑战，主要有：

* 百万级QPS的资源调用 (高并发)
* 99.99%的可用性 (高可用)
* 毫秒级的核心请求响应时间 (高性能)

     设计这样的互联网系统，不可避免的要考虑使用分布式缓存，并从可用性、并发性、性能多个方面进行综合考量。

**3.2 分布式缓存的实现方式**

**3.2.1 数据分片**

数据分片就是把数据均匀分散到多个实例中。数据分片可以采用以下几种规则：区间分片、hash分片、 slot分片。对于hash分片，主要的哈希算法有静态哈希和一致性哈希，静态哈希和一致性哈希对比如下：

* 静态哈希(取模求余)

*优点*：算法简单

*缺点：*加减节点时震荡厉害, 命中率下降厉害

* 一致性哈希

*优点：*加减节点时震荡较小, 保持较高命中率

*缺点：*自动rehash场景下会数据不一致的问题(同一份数据的请求在不同节点漂移)

    数据分片的实现方式分为三种：

* 客户端实现：memcached，redis 2.x

客户端实现数据分片

*优点：*简单，容易实现

*缺点：*扩缩容需要重新上线，手动数据迁移；

* proxy实现：twemproxy，codis，微博内部实现了CacheService

    通过引入一层代理，将数据分片策略放在代理层实现，客户端通过代理来访问数据。

*优点：*逻辑在proxy实现，客户端使用简单，支持多语言

*缺点：*数据访问多一跳，有一定的性能损耗

* 服务端实现：redis 3.x，cassandra

    由缓存组件本身，实现数据分片机制。

*优点：*扩缩容方便，自动数据迁移

*缺点：*数据存储与分布式逻辑耦合在一起，服务端复杂

**3.2.2 可用性**

    线上使用过程中，如果出现某些缓存实例不可用，大量请求穿透会给DB带来巨大的压力，极端情况会导致雪崩场景，这需要有更好的方式保证缓存的高可用。于是我们采用主从(Master/Slave)的架构，如图6。也就是在原有单层缓存的结构下，增加一层Slave，来保证即使某个Master节点宕机，整个缓存层依然是可用的，不会出现大量请求穿透DB的情况。

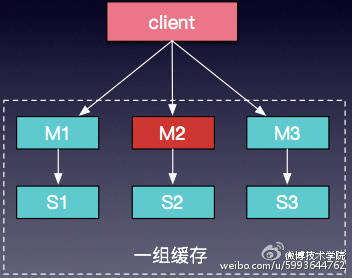


图6 master/slave缓存模型

​​

**3.2.3 扩展性**

主从结构部署已经能很好的满足大多数业务场景，但是在微博这种存在突发热点引起流量骤增的业务场景下仍然存在一定的问题，这种方式并不能很方便的进行横向扩展，如果直接在原有的缓存中增加新的节点，就需要涉及到数据迁移等工作。为了解决横向扩展的问题，增加了L1 Cache，实现了多级缓存，L1 Cache的容量一般小于Master容量，也就是说L1 Cache中的数据热度要更高；同时L1 Cache可以有多组，需要横向扩展的时候，只需要成组扩容L1 Cache即可，如图7。

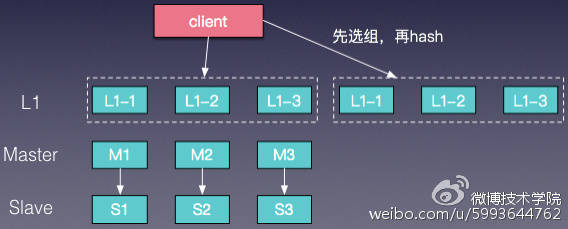


图7 L1 cache

    在L1 Cache这种结构下，Master/Slave的访问量会小很多，会出现Master/Slave数据变冷的情况，为了改善这种情况，我们把Master/Slave在逻辑上也作为L1 Cache的一组，这样就保证了Master/Slave的热度。如图8，是把Master作为L1的一组。

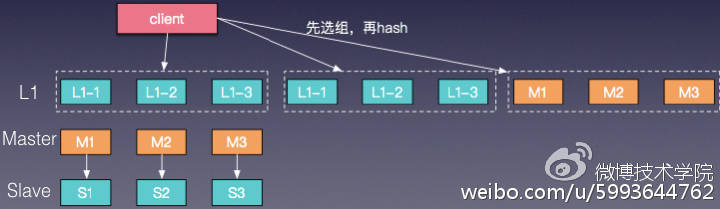


图8 master作为L1 cache

​​​

*iiiiiiiiii*

**四、缓存设计实践**

**4.1 Memached Multiget-Hole(multigut黑洞)**

    在Memcached采用数据分片方式部署的情况下，对于multiget命令来说，部署更多的节点，并不能提升multiget的承载量，甚至出现节点数越多，multiget的效率反而会降低，这就是multiget黑洞。这是由于执行multiget命令时，会对每一个节点进行访问，通常SLA取决于最慢最坏的节点，而且节点数增多，出问题的概率也增大，客户端处理的压力也会增大。通常在数据分片时，我们推荐4~8个节点左右。解决multiget黑洞有两种方式可供参考：

* 使用多副本的方式扩容，增加multiget的承载量
* 通过业务层面来控制，multiget的keys尽可能放在同一个节点上，但具体实施时较难操作，可行性不是很高。

**4.2 反向Cache**

     反向Cache就是将一个不存在的key放在缓存中，也就是在缓存中存一个空值。在某些场景下，比如微博维度的计数场景，若采用cache+DB的存储方式，由于大多数的微博并不存在转发、评论计数，这种场景下，就会出现由于大量访问不存在计数的mid，导致DB压力居高不下的情况。通过在cache中存一个null值，可减少对DB的穿透。当然这也存在潜在的风险或问题：

* 如果每次都是不同的mid，缓存效果可能不明显
* 需要更多的缓存容量

**4.3 缓存Fail-Fast (快速失败)**

    当缓存层某个节点出现故障时，会导致请求持续穿透到存储层，使请求响应时间长(需要等到读写故障缓存节点超时)，并且存储层负载居高不下。这就需要在使用缓存时考虑快速失败机制。快速失败指的是：当出现故障节点时，标识故障节点为不可用节点（策略举例：连续N次请求都出现超时，标识M时间段内为不可用)，读写不可用节点快速返回。通过快速失败策略，解决请求响应时间长问题，保证SLA。

**4.4 缓存无过期(Cache is Storage)**

    缓存无过期是指缓存中存储全量数据，不存在数据穿透的情况。 相比于缓存+DB的访问模型，使用内存存储简单可靠，但相应的内存成本也较高。选择内存缓存还是内存存储，需要结合具体的业务场景做权衡，比如单纯为解决Dog-Pile Effect而采用内存存储的话，内存成本可能就无法接受。通常情况下，内存存储模式，适合总体数据量很小，但是访问量巨大的业务场景，比如微博应用(来自weibo.com，weico等)列表。

**4.5 dog-pile effect (狗桩效应)**

    狗桩效应是由于极热访问的缓存数据失效，大量请求发现没有缓存，进而穿透至DB，导致数据库load瞬间飙高甚至宕机。这是一个典型的并发访问穿透问题，理想情况下缓存失效对数据库应该只有一次穿透。要解决这个问题，首先从代码层面就要考虑到并发穿透的情况，保证一个进程只有一次穿透；同时，可以考虑使用基于mc的分布式锁来控制。不过使用分布式锁来实现会较为繁琐，通常在代码层面进行控制，就可以得到很好的效果。

**4.6 极热点数据场景**

    微博在遇到一些突发事件时(如文章事件)，流量会出现爆发式的增长，大量的热点数集中访问，导致某个缓存资源遇到性能瓶颈(比如明星的数据所在的端口)，最终接口响应变慢影响正常的服务。为了应对这个问题我们在前端使用local cache, 以缓解后端缓存的压力。但是有些业务场景下，由于各种海量业务数据的冲刷，前端使用 local cache，命中率可能不高，性能提升也不明显，这种业务场景下可以考虑引入L1结构，通过部署多组小容量的L1缓存来应对突然的访问量增长。

**4. 7避免雪崩**

     雪崩效应是由于缓存服务器宕机等原因导致命中率降低，大量的请求穿透到数据库，导致数据库被冲垮，业务系统出现故障，服务很难再短时间内回复。避免雪崩主要从以下几方面考虑：

* 缓存高可用—master-slave

    避免单点故障，保证缓存高命中率

* 降级和流控-拒绝非核心功能的请求

    故障期间通过降级非核心功能来保证核心功能可用性

    故障期间通过拒掉部分请求保证有部分请求还能正常响应

* 清楚后端资源容量

    更好的预知风险点，提前做好准备

    即使出现问题，也便于更好的流控(具体应该放量多少)

**4.8 数据一致性**

    我们知道，在CAP理论下，只能取其二，而无法保证全部。在分布式缓存中，通常要保证可用性(A)和可扩展性(P)，并折中采用数据最终一致性，最终一致性包括：

* Master与副本一致性
* Cache与Storage一致性
* 业务各维度缓存数据一致性

**4.9 缓存容量规划**

    进行缓存容量规划时，主要从以下几个方面进行考虑：

* 请求量
* 命中率：预热，防止雪崩
* 网络带宽：网卡、交换机
* 存储容量：预估存储大小，过期策略、剔除率
* 连接数

*iiiiiiiiii*

**五、微博缓存中间件CacheService**

    随着微博业务数据的增长，缓存集群的规模也越来越大，直接使用裸缓存资源的方式也逐渐暴露出来一些问题：

* 缓存资源变更复杂度高

节点变更、扩缩容需要业务方变更配置重新上线，而且资源变更过程中业务方密切关注缓存的服务状况。

* 高可用策略无法复用

微博平台使用java语言开发，java client定制了保证缓存高可用性的多级缓存访问策略，公司其它使用非java语言(php、go)的部门无法使用。

* 运维复杂度高

    缺少简单友好的统一运维管理平台负责缓存资源的申请、分配、部署、变更、回收等操作。运维操作没有实现全界面化，且自动化程度不高。

* 缺少SLA指标保证

    缓存资源异常导致业务出现问题时，缺少业务缓存SLA指标的监控及处理，更多的是依赖用户投诉后执行后续的运维处理，整个过程周期耗时较长，对业务影响较大。

    为了应对以上问题，2013年底开始推进缓存服务化。设计了内部缓存服务化体系CacheService, 目前已在微博多个业务场景中使用。整体架构设计如图9：

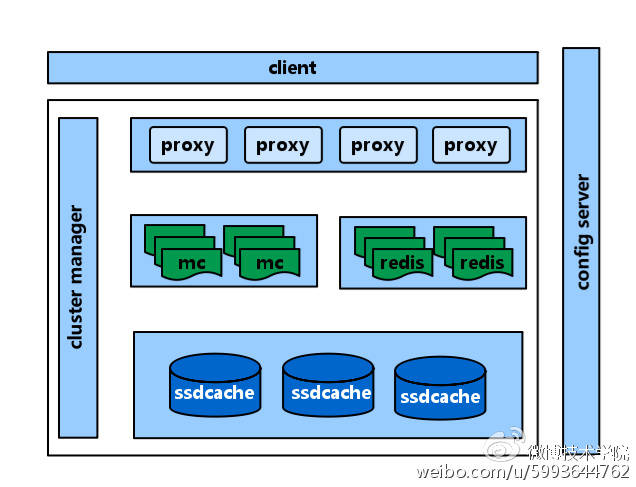


图9 cacheservice架构设计

​    整体框架包括以下几个组件：

**代理层**

    代理业务端的请求，并基于设定的路由规则转发到后端的 cache 资源，它本身是无状态的。支持配置服务化、多级cache访问策略、集群间数据复制、S4LRU等特性。

**资源层**

    实际的数据存储引擎，初期支持 memcached，后续又扩展了 Redis、SSDCache 组件，其中 SSDCache 是为了降低服务成本，内部开发的基于 SSD 的存储组件，用于缓存介于 memory 和 DB 之间的 warm 数据。

**客户端**

    业务只需要简单配置所使用的服务池名group和业务标识namespace即可使用

java业务方：通过配置中心获取可访问的proxy节点

php业务方：本地proxy节点 or DNS

**配置中心 ConfigServer**

     微博内部的配置服务中心，主要是管理静态配置和动态命名服务的一个远程服务，并能够在配置发生变更的时候实时通知监听（me：可用zookeeper实现）的 ConfigClient。

**集群管理系统 ClusterManager**

管理缓存的整个生命周期，包括缓存资源的申请、上线、变更、下线等；管理集群中所有组件的运行状态及业务的SLA指标，出现异常时自动触发运维操作。

    缓存服务化的道路还是很长，未来还需要进一步的对CacheService服务化框架及其相关组件持续优化；需要逐步完善服务化体系中冷热数据分级存储机制降低服务成本；需要进一步完善集群管理组件降低运维复杂度等。