Memcached存储结构分析

Memcached是分布式内存缓存系统，主要作用是缓存后端数据库产生的数据，以此来减少数据库访问次数，从而提高加载动态网站的速度。缓存的概念可以说遍布计算机机各个方面，通常我们所说的缓存指的是CPU中用于解决内存和CPU速度不匹配的硬件。要弄清楚Memcached到底为什么能起到缓存作用，首先还得了解典型Web架构。

通常我们访问一个动态网站时，除了返回静态页面还需要请求后台数据库，一旦涉及数据库请求就免不了需要大量I/O操作，这部分时间将会占到整个从提交请求到收到结果这段时间的大部分，所以如果能够不通过常规数据库操作就能得到结果就能大大提高访问速度。Memcached维护了一大片内存区域，这些内存区域用来存储之前被访问的数据库反馈信息，当用户再一次请求这些数据时，就可以直接从内存缓存系统中得到，这样就免去了查找数据库的工作。

将Memcached定义为数据库本身是不够妥当的，因为它并不能存下所有我们想要存下的数据，它所存储的只是Mysql这类关系数据库内容的子集。所以，这里只将其作为一个Web后端缓存系统来理解比较合适。

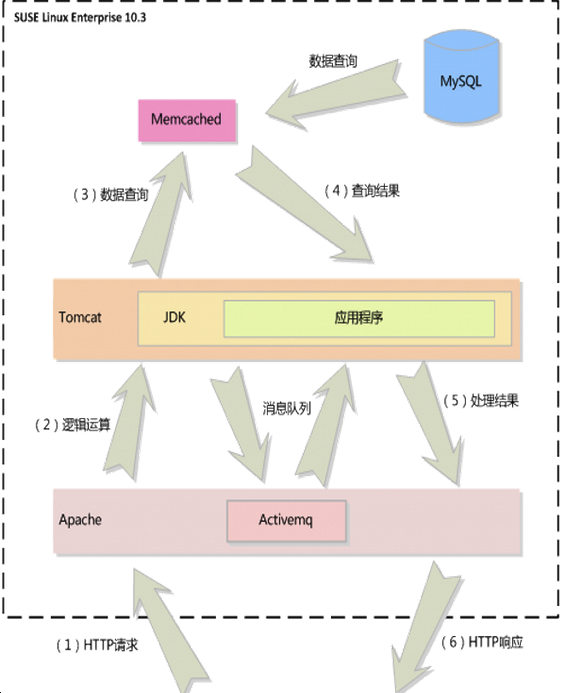


图1：Memcached工作层

这里很清楚地展示了Memcached所起的作用，相比于缓存服务器squid的作用，二者差别还是很大的，有兴趣的可以把squid也看一下，就可以对前端（代理）缓存和后端缓存的区别和作用了解更多了。

Memcached被叫做分布式缓存系统，但是各个独立的系统并不存在直接的通信，那又是怎么实现分布式的呢？这就要先聊一聊Memcached的数据存储方式了。相比于常规的关系型数据库，它在存储数据时通过key-value键值对来存储数据，其中value就是实际需要存储的数据，而key则是用来定位的。数据结构中哈希存储方式通过存储数据计算哈希值，然后根据得到的哈希值指定具体的存储位置，这里虽然在实现上有所不同但是思想上确是相近的。我们通过下面存取的例子来了解更多细节。

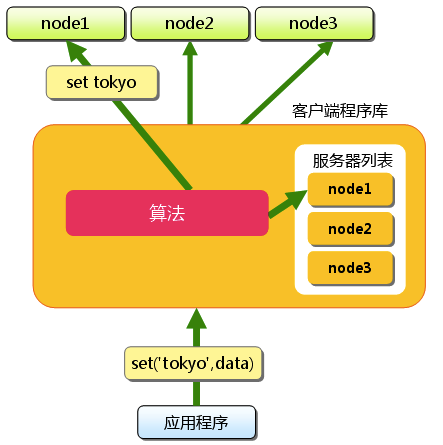


图2：添加数据实现分布式

可见，在向Memcached各个节点中写入数据前，首先需要使用键（key）的内容，通过服务器端程序（对于Memcached服务器而言算是客户端，如图）计算得到具体需要将该键的数据存储在具体某个Memcached节点上，这里所说的节点既有可能是部署在同一台机器上的，也有可能是部署在别的及机器上的，所以通过键的映射就可以将数据均衡地分配到不同的节点上，从而实现“分布式”存储。

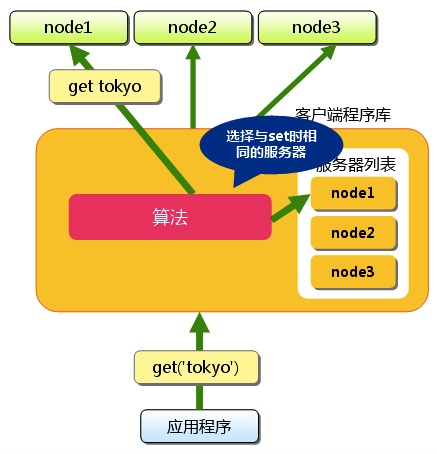


图3：获得数据实现分布式

获取数据实现分布式依旧需要通过服务器端程序来实现，对于请求的数据，首先对键值（key）计算一次，得到其映射的节点，因为算法是相同的，所以可以映射到和存储时完全相同的节点，之后从该节点中取得数据。

关于Memcached的哈希算法实现并不是本文的重点，所以这里不会多少文字去描述。

提到Memcached通常都会用来和另一个基于key-value的缓存系统Redis进行比较，这里不会做过多的比较分析。对于二者而言，最大的区别就是数据的可持久化问题了，那么什么是数据持久化呢？简单来说就像是磁盘存储和内存的区别一样，存储在磁盘中的数据即使断电了也会一直保存，但是内存中的数据一旦断电就不再存在了。而上面提到的Memcached是不支持数据持久化的，这点和Redis不一样。也就是说，在运行过程中只要断电或者重启Memcached服务器都会丢失之前所有的缓存数据，这当然对大部分情况来说并不是好消息，要弄明白为什么Memcached为什么不能支持数据持久化还得从它存储模型说起。

Memcached之所以能起到缓存作用是因为提前已经将原本要从数据库中读取的数据缓存在了内存中，而内存本身是易失性材料，所以其特性正是因为内存本身性质决定的。所以就引出了一个最大的问题：如何管理内存？

我们知道，抛开花哨的概念，Memcached启动后，本身就是一个进程，这点和一般程序没有太大区别。所以在这点上，它可用的线性地址空间和一般程序也没有太大差别，而要提供内存缓存服务的话，首先需要考虑的就是如何从操作系统手中得到足够存储下相应数据的内存块。这里需要被缓存的数据均是在Memcached启动后才产生的，所以也只能在堆区动态分配内存了。这似乎看起来顺理成章，每一次有缓存请求时在堆区分配一块存储空间，查找时就把此空间数据返回，看起来也挺不错。

但是，很快我们就发现几个难以忍受的问题。一是数据块分配和回收会很困难，大小不一的数据块频繁请求回收很快会产生大量内存碎片。二是每一个数据块无论大小都要调用malloc和free来完成操作，但是调用malloc就会间接调用brk和mmap两个系统调用函数，这样在频繁写入时效率会很低。三是分配到的数据块大小不一，管理起来非常麻烦，尤其是记录后端反馈数据通常不会太大，也就是说小数据占据绝大多数的情况下，管理数据将会非常麻烦。基于上述三点，使用基于malloc的分配方式并不适合用作缓存系统。

考虑到上面几种可能出现的问题，Memcached使用Slab分配器用作内存分配及管理，Slab最初是在Sun提出并用于操作系统内存分配的分配器，其最大的特征是所有的操作都是基于一类对象进行的，用最简单的话来说就是，Slab分配器将原本完整的大块内存空间均匀切分成多个等大的内存块集合，每个集合中包含多个等大的内存块，内存分配是根据Slab中已经分好的内存块作为基本单位进行的。这种方式非常适合频繁的分配操作，所以无论是内核中Slab的使用还是这里的Memcached都有非常出色的性能。不过Memcached在实现上还是针对最基本的Slab分配器做了一些改变。

之前看了Memcached关于内存分配部分的源码（slabs.h及slabs.c，可以只看slabs.c部分即可，如果要弄懂初始化过程，还需要看一下memcached.h中setting结构及其初始化），比较清楚其内部实现，如果要能比较清晰地看懂下文，最好看一下我github上对源码的注释。

首先需要弄清slabs.c中slabclass\_t这个结构体定义。

|  |
| --- |
| typedef struct {  unsigned int size; /\* sizes of items \*/  unsigned int perslab; /\* how many items per slab \*/  void \*slots; /\* list of item ptrs \*/  unsigned int sl\_curr; /\* total free items in list \*/  unsigned int slabs; /\* how many slabs were allocated for this class \*/  void \*\*slab\_list; /\* array of slab pointers \*/  unsigned int list\_size; /\* size of prev array \*/  size\_t requested; /\* The number of requested bytes \*/  } slabclass\_t; |

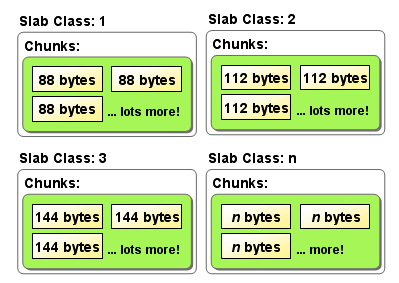


图4：Slab class

在初始化时会定义大小为64的上述结构体数组用来表示64个Slab类，从上面结构体定义就可以看出，slabclass结构体并不直接包含可用存储空间，仅仅包含指向存储区域的指针，所以这里的Slab类结构主要是为了从宏观上去管理Memcched存储空间的。

为了讲清楚Slab是到底如何管理存储空间，首先需要了解一些在Slab相关结构及其彼此间的关系。

Slab class：同一个Slab类中的数据块大小都是相等的定义在上面的结构体中，其中size表示这个类中数据块的大小，perslab表示该类中拥有的数据块数量。

Slab\_list array：上面定义中slab\_list指向了指针数组，该数组中每个元素都是指向一个Slab（实际分配了存储空间的容器）的指针，上面结构体中list\_size变量表示的就是此指针数组的大小。

Slab：具体的容器，初始化时，每个Slab class先分配一个Slab。

Page：初始时和Slab容器等大。

Chunk：每个Slab中数据块大小，分配管理存储空间的基本单位。

Item\_chunk：包含四部分[item\_header | key-value]，其中key-value指的是实际存储的数据部分。具体的item结构定义如下。

|  |
| --- |
| typedef struct \_strchunk {  struct \_strchunk \*next; /\* points within its own chain. \*/  struct \_strchunk \*prev; /\* can potentially point to the head. \*/  struct \_stritem \*head; /\* always points to the owner chunk \*/  int size; /\* available chunk space in bytes \*/  int used; /\* chunk space used \*/  int nbytes; /\* used. \*/  unsigned short refcount; /\* used? \*/  uint8\_t nsuffix; /\* unused \*/  uint8\_t it\_flags; /\* ITEM\_\* above. \*/  uint8\_t slabs\_clsid; /\* Same as above. \*/  char data[];  } item\_chunk; |

举个例子就说说如果我们存储的数据key-value共30Bytes，则组成一个item基本单位的大小为item\_header大小 + 30Bytes，则会找出一个能够放得下且最接近item\_chunk大小的chunk用来存储它，但是这种方式会产生内部碎片（chunk块内），如图6所示。

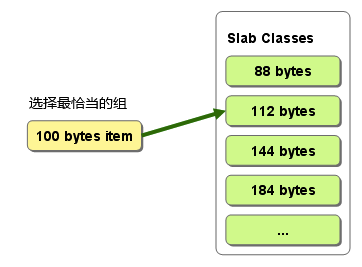


图5：chunk选择

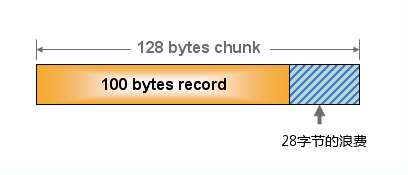


图6：内部碎片

sl\_cuur：该指针指向空闲链表，当Memcached中数据失效后并不是直接将分配的存储空间回收，而是将这块空间挂在空闲链表上，空闲链表采用的是双向链表结构存储的。

更详细的Slab分配机制这里就不详细描述了，有兴趣的可以去看我的slabs.c注释，会有更多chunk增长系数、LRU回收算法等更详细的内容。