memcache内核，一文搞定！面试再也不怕了！！！（值得收藏）

Original 58沈剑 架构师之路 2019-06-12 04:42

memcache是互联网分层架构中，使用最多的的KV缓存。面试的过程中，memcache相关的问题几乎是必问的，关于memcache的面试提问，你能回答到哪一个层次呢？

画外音：很可能关乎，你拿到offer的薪酬档位。

第一类问题：知道不知道

这一类问题，考察用没用过，知不知道，相对比较好回答。

关于memcache一些基础特性，使用过的小伙伴基本都能回答出来：

（1）mc的**核心职能**是KV内存管理，**value存储**最大为1M，它不支持复杂数据结构（哈希、列表、集合、有序集合等）；

（2）mc不支持持久化；

（3）mc支持key过期；

（4）mc持续运行很少会出现内存碎片，速度不会随着服务运行时间降低；

（5）mc使用非阻塞IO复用网络模型，使用监听线程/工作线程的多线程模型；

面对这类封闭性的问题，一定要斩钉截铁，毫无犹豫的给出回答。

第二类问题：为什么(why)，什么(what)

这一类问题，考察对于一个工具，只停留在使用层面，还是有原理性的思考。

memcache为什么不支持复杂数据结构？为什么不支持持久化？

业务决定技术方案，mc的诞生，以“以服务的方式，而不是库的方式管理KV内存”为**设计目标**，它颠覆的是，KV内存管理组件库，复杂数据结构与持久化并不是它的初衷。

当然，用“颠覆”这个词未必不合适，库和服务各有使用场景，只是在分布式的环境下，服务的使用范围更广。设计目标，诞生背景很重要，这一定程度上决定了实现方案，就如redis的出现，是为了有一个更好用，更多功能的缓存服务。

画外音：我很喜欢问这个问题，大部分候选人面对这个没有标准答案的问题，状态可能是蒙圈。

memcache是用什么技术实现key过期的？

懒淘汰(lazy expiration)。

memcache为什么能保证运行性能，且很少会出现内存碎片？

提前分配内存。

memcache为什么要使用非阻塞IO复用网络模型，使用监听线程/工作线程的多线程模型，有什么优缺点？

目的是提高吞吐量。

多线程能够充分的利用多核，但会带来一些锁冲突。

面对这类半开放的问题，有些并没有标准答案，一定要回答出自己的思考和见解。

第三类问题：怎么做(how) | 文本刚开始

这一类问题，探测候选人理解得有多透，掌握得有多细，对技术有多刨根究底。

画外音：所谓"好奇心"，真的很重要，只想要"一份工作"的技术人很难有这种好奇心。

memcache是什么实现内存管理，以减小内存碎片，是怎么实现分配内存的？

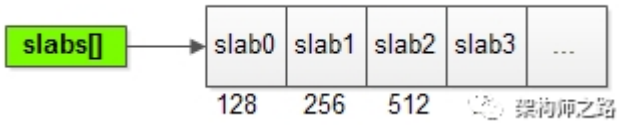
开讲之前，先解释几个非常重要的概念：

chunk：它是将内存分配给用户使用的最小单元。

item：用户要存储的数据，包含key和value，最终都存储在chunk里。

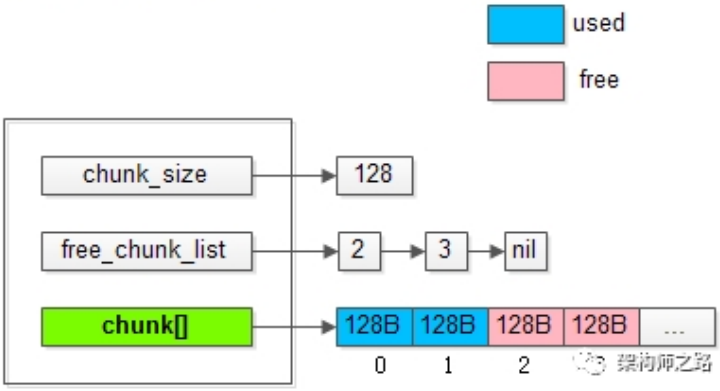
slab：它会管理一个固定chunk size的若干个chunk，而mc的内存管理，由若干个slab组成。

画外音：为了避免复杂性，本文先不引入page的概念了。



如上图所示，一系列slab，分别管理128B，256B，512B…的chunk内存单元。

将上图中管理128B的slab0放大：



能够发现slab中的一些核心数据结构是：

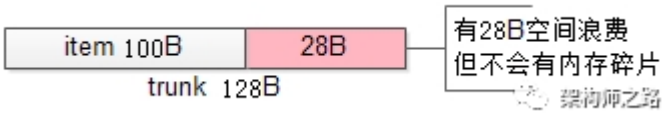
- **chunk_size**：该slab管理的是128B的chunk
- **free_chunk_list**：用于快速找到空闲的chunk
- **chunk[]**：已经预分配好，用于存放用户item数据的实际chunk空间

画外音：其实还有lru_list。

假如用户要存储一个100B的item，是如何找到对应的可用chunk的呢？

会从最接近item大小的slab的chunk[]中，通过free_chunk_list快速找到对应的chunk，如上图所示，与item大小最接近的chunk是128B。

为什么不会出现内存碎片呢？

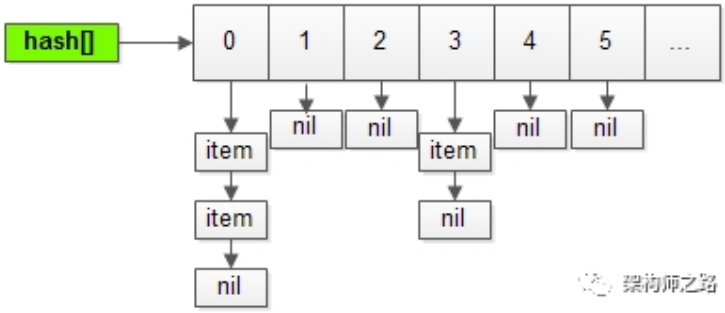


拿到一个128B的chunk，去存储一个100B的item，余下的28B不会再被其他的item所使用，即：实际上浪费了存储空间，来减少内存碎片，保证访问的速度。

画外音：理论上，内存碎片几乎不存在。

memcache通过slab，chunk，free_chunk_list来快速分配内存，**存储用户的item，那它又是如何快速实现key的查找的呢？**

没有什么特别算法：



- 通过hash表实现快速查找
- 通过链表来解决冲突

用最朴素的方式，实现key的快速查找。

随着item的个数不断增多，hash冲突越来越大，hash表如何保证查询效率呢？

当item总数达到hash表长度的1.5倍时，hash表会动态扩容，rehash将数据重新分布，以保证查找效率不会不断降低。

扩展hash表之后，同一个key在新旧hash表内的位置会发生变化，**如何保证数据的一致性，以及如何保证迁移过程服务的可用性呢**（肯定不能加一把大锁，迁移完成数据，再重新服务吧）？

哈希表扩展，数据迁移是一个耗时的操作，会有一个专门的线程来实施，为了避免大锁，采用的是“分段迁移”的策略。

当item数量达到阈值时，迁移线程会分段迁移，对hash表中的一部分桶进行**加锁，迁移数据，解锁**：

- 一来，保证不会有长时间的阻塞，影响服务的可用性
- 二来，保证item不会在新旧hash表里不一致

新的问题来了，对于已经存在与旧hash表中的item，可以通过上述方式迁移，那么**在item迁移的过程中，如果有新的item插入，是应该插入旧hash表还是新hash表呢？**

memcache的做法是，判断旧hash表中，item应该插入的桶，是否已经迁移至新表中：

- **如果已经迁移**，则item直接插入新hash表
- **如果还没有被迁移**，则直接插入旧hash表，未来等待迁移线程来迁移至新hash表

为什么要这么做呢，**不能直接插入新hash表吗？**

memcache没有给出官方的解释，楼主揣测，这种方法能够保证一个桶内的数据，只在一个hash表中（要么新表，要么旧表），任何场景下都不会出现，旧表新表查询两次，以提升查询速度。

memcache是怎么实现key过期的，懒淘汰(lazy expiration)具体是怎么玩的？

实现“超时”和“过期”，最常见的两种方法是：

- 启动一个超时线程，对所有item进行扫描，如果发现超时，则进行超时回调处理
- 每个item设定一个超时信号通知，通知触发超时回调处理

这两种方法，都需要有额外的资源消耗。

mc的查询业务非常简单，只会返回cache hit与cache miss两种结果，这种场景下，非常适合使用懒淘汰的方式。

懒淘汰的核心是：

- item不会被主动淘汰，即没有超时线程，也没有信号通知来主动检查
- item每次会查询(get)时，检查一下时间戳，如果已经过期，被动淘汰，并返回cache miss

举个例子，假如set了一个key，有效期100s：

- 在第50s的时候，有用户查询(get)了这个key，判断未过期，返回对应的value值
- 在第200s的时候，又有用户查询(get)了这个key，判断已过期，将item所在的chunk释放，返回cache miss

这种方式的实现代价很小，消耗资源非常低：

- 在item里，加入一个过期时间属性
- 在get时，加入一个时间判断

内存总是有限的，chunk数量有限的情况下，能够存储的item个数是有限的，**假如chunk被用完了，该怎么办？**

仍然是上面的例子，假如128B的chunk都用完了，用户又set了一个100B的item，**要不要挤掉已有的item？**

要。

这里的**启示**是：

（1）即使item的有效期设置为“永久”，也可能被淘汰；

（2）如果要做全量数据缓存，一定要仔细评估，cache的内存大小，必须大于，全量数据的总大小，否则很容易采坑；

挤掉哪一个item？怎么挤？

这里涉及LRU淘汰机制。

如果操作系统的内存管理，最常见的淘汰算法是FIFO和LRU：

- **FIFO**(first in first out)：最先被set的item，最先被淘汰
- **LRU**(least recently used)：最近最少被使用(get/set)的item，最先被淘汰

使用LRU算法挤掉item，需要增加两个属性：

- 最近item访问计数
- 最近item访问时间

并增加一个LRU链表，就能够快速实现。

画外音：所以，管理chunk的每个slab，除了free_chunk_list，还有lru_list。

思路比结论重要。

架构师之路-分享技术思路

文章较长，若有收获，帮忙转发+再看一下。

调研：面试memcache内核，你在哪个档位？

Read more