# memcached

memcached解析

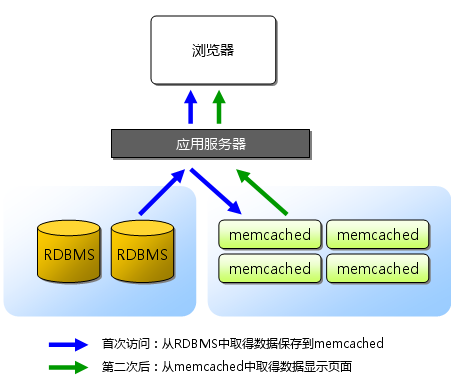
<https://kb.cnblogs.com/page/42731/>

memcached和redis比较

<https://www.biaodianfu.com/redis-vs-memcached.html>

许多Web应用都将数据保存到RDB服务端中，应用服务器从中读取数据并在浏览器中显示。但随着数据量的增大、访问的集中，就会出现RDB服务端的负担加重、数据库响应恶化、网站显示延迟等重大影响。

memcached是高性能的分布式内存缓存服务器。一般的使用目的是，通过缓存数据库查询结果，减少数据库访问次数，以提高动态Web应用的速度、 提高可扩展性。



memcached是一种基于内存的key-value存储，用来存储小块的任意数据（字符串、对象）。这些数据可以是数据库调用、API调用或者是页面渲染的结果。

memcached 的守护进程（daemon）是用C写的，但是客户端可以用任何语言来编写，并通过memcached协议与守护进程通信。但是它并不提供冗余（例如，复制其hashmap条目），当某个服务器S停止运行或崩溃了，所有存放在S上的键/值对都将丢失。

memcached处理的原子是每一个（key，value）对，key会通过一个hash算法转化成hash-key，便于查找、对比以及做到尽可能的散列。同时，memcached用的是一个二级散列，通过一张大hash表来维护。

在一个memcached的查询中，客户端先通过计算key的hash值来确定kv对所处在的服务端位置。当服务端确定后，客户端就会发送一个查询请求给对应的服务端，让它来查找确切的数据。因为这之间没有交互以及多播协议，所以 memcached交互带给网络的影响是最小化的。

## 特点

1. 协议简单：memcached的服务器客户端通信并不使用复杂的MXL等格式，而是使用简单的基于文本的协议。
2. 基于libevent的事件处理：libevent是个程序库，将Linux 的epoll、BSD类操作系统的kqueue等时间处理功能封装成统一的接口。memcached使用这个libevent库，因此能在Linux、BSD、Solaris等操作系统上发挥其高性能。
3. 内置内存存储方式：为了提高性能，memcached中保存的数据都存储在memcached内置的内存存储空间中。由于数据仅存在于内存中，因此重启memcached，**重启操作系统会导致全部数据消失**。另外，内容容量达到指定的值之后memcached回自动删除不适用的缓存。memcached本身是为缓存而设计的服务器，因此并没有过多考虑数据的永久性问题。
4. memcached不互通信的分布式：memcached尽管是“分布式”缓存服务器，但服务器端并没有分布式功能。各个memcached不会互相通信以共享信息。**他的分布式主要是通过客户端实现的**。

## 内存分配

slab allocation机制：整理内存以便重复使用。

默认情况下，服务端是用一个内置的叫块分配器（slab allocator）的组件来分配内存的。在此之前，内存分配是通过对所有的记录简单malloc/free来进行的，这种方式会导致内存碎片，加重操作系统内存管理器的负担，而采用块分配器的主要目的是为了避免内存碎片。

slab allocator的基本原理是按照预先规定的大小，将分配的内存分割成特定长度的块，以完全解决内存碎片问题。

slab allocator的原理相当简单，将分配的内存分割成各种尺寸的块（chunk），并把尺寸相同的块分成组（chunk的集合）。



而且slab allocator 还有重复使用已分配内存的目的。也就是说，分配到的内存不会释放，而是重复利用。

**slab allocator的主要术语：**

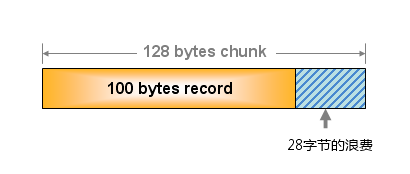
* page :分配给slab的内存空间，默认是1MB。分配给slab 之后根据slab 的大小切分成chunk；
* chunk : 用于缓存记录的内存空间；
* slab class:特定大小chunk的组。

memcached收到数据后会根据数据的大小选择最合适数据大小的slab存放。memcached中保存着slab内空闲chunk列表，根据列表选择chunk，然后把数据缓存在其中。



### slab allocator的问题

虽然slab allocator解决了内存碎片的问题，但是也带来的新的问题，由于分配的都是特定长度的内存，因此无法有效利用分配的内存。



比如上图，会有28 byte的浪费。这个问题没有完美的解决方案，官方文档中记录**如果可以预先知道客户端发送的数据的共用大小**，或者缓存数据大小相同的情况下，只要使用适合数据大小的组的列表，就可以减少浪费。

### 使用growth factor进行调优

memcached在启动时指定growth factor，可以在某种程度上控制slab之间的差异，默认值是1.25。在此之前，默认值是2。

如果设置growth factor为2，在启动的时候可以看到：

slab class 1: chunk size 128 perslab 8192

slab class 2: chunk size 256 perslab 4096

slab class 3: chunk size 512 perslab 2048

slab class 4: chunk size 1024 perslab 1024

slab class 5: chunk size 2048 perslab 512

slab class 6: chunk size 4096 perslab 256

slab class 7: chunk size 8192 perslab 128

slab class 8: chunk size 16384 perslab 64

slab class 9: chunk size 32768 perslab 32

slab class 10: chunk size 65536 perslab 16

slab class 11: chunk size 131072 perslab 8

slab class 12: chunk size 262144 perslab 4

slab class 13: chunk size 524288 perslab 2

（1MB=524288\*2）

这样设置的问题是slab之间的间距比较大，有些情况下会相当的浪费内存空间，为了减少内存的浪费，设置了默认值为1.25。

slab class 1: chunk size 88 perslab 11915

slab class 2: chunk size 112 perslab 9362

slab class 3: chunk size 144 perslab 7281

slab class 4: chunk size 184 perslab 5698

slab class 5: chunk size 232 perslab 4519

slab class 6: chunk size 296 perslab 3542

slab class 7: chunk size 376 perslab 2788

slab class 8: chunk size 472 perslab 2221

slab class 9: chunk size 592 perslab 1771

slab class 10: chunk size 744 perslab 1409

这时组间差距要比2的时候小很多了，从结果来看，并不是完全按照1.25间距来设置的，原因是保持字节的对其故意设置的。

服务端对key和data都有相应的限制，key的长度不能超过250字节，data也不能超过块大小的限制1MB。因为客户算所使用的hash算法，并不会考虑到每个服务端的内存大小。理论上客户端会分配概率上等量的kv对给每个服务端，这样如果每个服务端的内存都不太一样，那可能会**导致内存使用率的降低。**所以一种替代的解决方案是，根据每个服务端的内存大小，找出他们的最大公约数，然后在每个服务端上开n个容量等于最大公约数的 instance，这样就等于拥有了多个容量大小一样的子服务端，从而提供整体的内存使用率。

## 数据删除机制

memcached不会释放已分配的内存。记录超时后，客户端就无法再看见该记录（invisible），但是其存储空间即可重复使用。memcached内部不会监视记录是否过期，而是在get时查看记录的时间戳，检查记录是否过期。 这种技术被称为lazy（惰性）expiration。因此，memcached不会在过期监视上耗费CPU时间。

memcached会优先使用已超时的记录空间，但即使如此，也会发生追加新纪录时空间不足的情况。此时就要使用名为Least Recently Used (LRU)机制来分配空间。这就是删除最少使用的记录的机制。因此当memcached的内存空间不足时（无法从slab class）获取到新空间时，就从最近未使用的记录中搜索，并将空间分配给新的记录。从缓存的实用角度来看，该模型十分理想。

不过，有些情况下LRU机制反倒会造成麻烦。memcached启动时通过“-M”参数可以禁止LRU。指定“-M”参数启动后，内存用尽时memcached会返回错误。 话说回来，memcached毕竟不是存储器，而是缓存，所以推荐使用LRU。

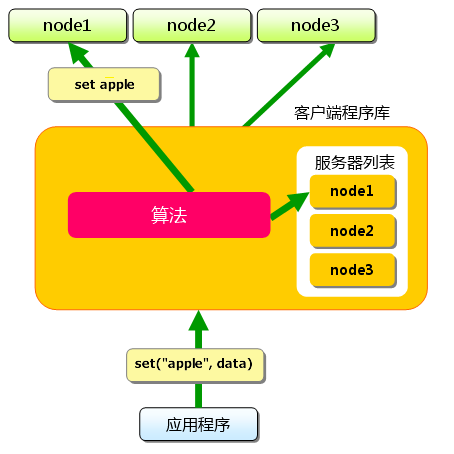
对于缓存存储容量满的情况下的删除需要考虑多种机制，一方面是按队列机制，一方面应该对应缓存对象本身的优先级，根据缓存对象的优先级进行对象的删除。

## 分布式

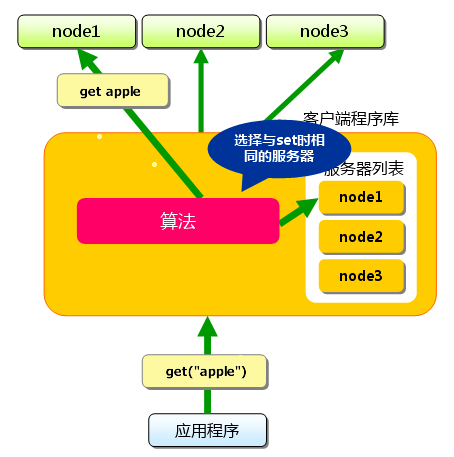
memcached虽然称为分布式缓存服务器，但服务器端并没有分布式的功能。memcached的分布式完全是有客户端实现的。这种分布式是memcached的最大特点。

假设memcached服务器有node1~node3三台，应用程序要保存键名为"apple"、"orange"、"pear"的数据。

首先向memcached中添加" apple "，当" apple "传给客户端程序库之后，客户端根据算法决定"apple"存放的memcached服务器，服务器选定之后命令它存储" apple "以及对应的值。



获取时也要将要获取的键"apple"传递给函数库。函数库通过与数据保存时相同的算法，根据键的值"apple"选择服务器。由于算法相同，所以会找到之前存放的服务器，然后发送get命令。只要数据没有被删除，那么客户端就可以get到对应的值。



这样，将不同的键保存在不同的服务器上，就实现了分布式。memcached服务器增多后，键就会分散，即使一台memcached服务器发生故障无法连接，也不会影响其他的缓存，系统依然能继续运行。

如果根据键计算存放的服务器位置采用的是hash法，那么普通的hash方法自然会遇到集群扩大，之前数据无效的问题。这种时候要采用一致性哈希或者改进过的一致性哈希，就能很好解决集群扩容的问题。

## 缓存多副本

缓存多副本主要是用于在缓存数据存放时存储缓存数据的多个副本，以防止缓存失效。缓存失效发生在以下几种情况：

1. 缓存超时被移除（正常失效）
2. 缓存由于存储空间限制被移除（异常失效）
3. 由于缓存节点变化而导致的缓存失效（异常失效）

在缓存多副本的情况下，需要重新考虑缓存的分布式分布策略。其次缓存的多个副本实际本身是可能的多个读的节点，可以做为分布式的并行读，这是另外一个可以考虑的问题。

## 缓存数据的一致性问题

缓存数据尽量只读，因此缓存本身是不适合大量写和更新操作的数据场景的。对于读的情况下，如果存在数据变化，一种是同时更新缓存和数据库。一种是直接对缓存数据进行失效处理。

# redis

# 区别

* redis与 memcached相比，redis支持key-value数据类型，同事支持list、set、hash等数据结构的存储；
* redis支持数据的备份，即master-slave模式的数据备份；
* redis支持数据的持久化；
* redis在很多方面支持数据库的特性，可以这样说他就是一个数据库系统，而memcached只是简单地K/V缓存；
* 它们在性能方面差别不是很大，读取方面尤其是针对批量读取性能方面memcached占据优势。当然redis也有他的优点，如持久性、支持更多的数据结构。

所以在选择方面如果有持久方面的需求或对数据类型和处理有要求的应该选择redis。如果简单的key/value 存储应该选择memcached。