**memcache学习笔记**

memcache是自由、开源、高效、分布式的内存对象缓存系统，是老牌的nosql的应用。

nosql：not only sql，不仅仅是关系型数据库，有memcached、redis

nosql特点：key-value键值对存储。

基于文档存储的数据库：mongodb

# 1 linux下安装memcached

## 安装依赖包libevent

|  |  |
| --- | --- |
| 下载源码 | wget https://github.com/libevent/libevent/releases/download/release-2.0.22-stable/libevent-2.0.22-stable.tar.gz |
| 解压 | tar -zxvf libevent-2.0.22-stable.tar.gz |
| 进入目录 | cd libevent-2.0.22-stable |
| 检查编译环境 | ./configure --prefix=/usr/local/libevent |
| 编译和安装 | make && make install |

## 1.2 安装Memcached

|  |  |
| --- | --- |
| 下载源码 | wget http://www.memcached.org/files/memcached-1.4.31.tar.gz |
| 解压 | tar zxvf memcached-1.4.31.tar.gz |
| 进入目录 | cd memcached-1.4.31 |
| 检查编译环境 | ./configure --prefix=/usr/local/memcached --with-libevent=/usr/local/libevent |
| 编译和安装 | make && make install |

## 1.3 memcached常用启动参数

注意：处于安全考虑，linux下不能用root用户启动memcached。

几个重要参数的使用：

|  |  |
| --- | --- |
| -p <num> | 监听端口，默认端口11211 |
| -u <username> | 用户，在linux下时使用 |
| -m <num> | 分配给memcached的最大缓存，默认64M |
| -c <num> | 最大连接数，默认是1024 |
| -f <factor> | 增长因子 |
| -vv | 打印信息 |
| -d | 选项是启动一个守护进程 |

**(1) 启动memcached**

以守护进程启动：memcached -m 128 -u nobody -d

以打印信息方式启动：memcached -m 128 -u nobody -vv

查看memcached进程：ps -A | grep memcached

关闭memcached进程：kill -9 memcached的进程id

**(2) telnet连接memcached**

memcached的客户端和服务器端的通信使用的是文本协议，所以可以用telnet连接memcached来做交互。

在终端连接：telnet localhost 11211，然后按下ctrl+]组合键打开回显功能，再按下确认键即可。

也可以使用secureCRT连接。

# 2 memcached的命令

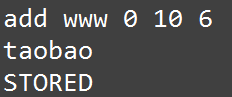
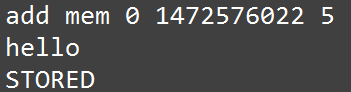
## 2.1 add命令

添加键值对，注意：key名称不存在时才能添加键值对成功。

格式：**add key flag expire lenght**

|  |  |
| --- | --- |
| key | 给值取的名字 |
| flag | 标志，要求一个正整数  意义：如果向memcached存储的是数组或对象，先要把数组或对象序列化成字符串再存入memcached，当从memcached取出该键值时，通过约定flag值来把字符串反序列化成需要的数组或对象，示例flag值： 1表示字符串，2表示反序列化成数组，3表示反序列化成对象 |
| expire | 键值对的有效期  意义：(1)设置为0，不自动失效，编译时最大常量是30天，30天后也会自动失效，也可能还不到30天就被新数据挤出去  (2)设置在将来的多少秒过期  (3)设置时间戳，指定某个日期时间点过期，常用在抢购中 |
| lenght | 值的长度，单位bytes，注意写值时也要对应此长度 |

示例：

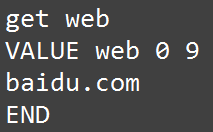
  

## 2.2 get命令

获取键的值

格式：**get key**

示例：



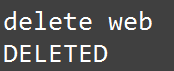
## 2.3 delete命令

删除键值对

格式：**delete key [seconds]**

可选参数seconds表示删除键值后，N秒内都不可再用这个key名，目的是让页面的缓存代谢完毕。

示例：

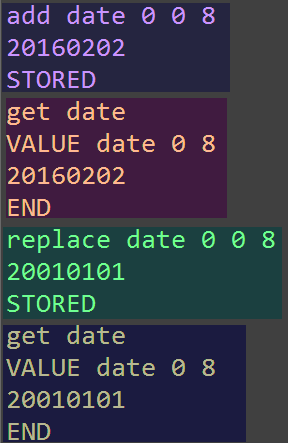


## 2.4 replace命令

修改键的值，参数和get命令一样，注意replace的对已经存在的key才能修改成功。

格式：**replace key flag expire lenght**

示例：

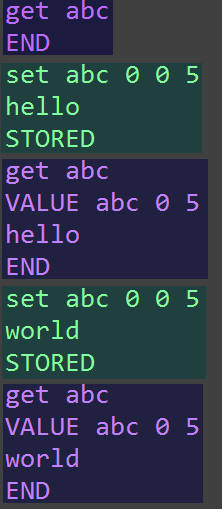


## 2.5 set命令

同时具有add和replace两着的功能，当key不存在则添加，当key存在值修改key的值。

格式：**set key flag expire lenght**

示例：



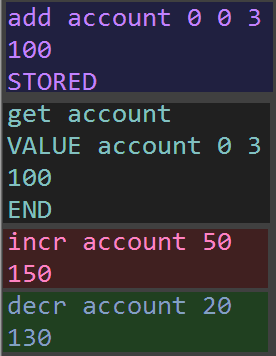
## 2.6 incr和decr命令

incr增加值的大小，decr减少值的大小，把值理解为32位无符号数来加减操作，所以值的范围0～。

格式： **incr key num**

**decr key num**

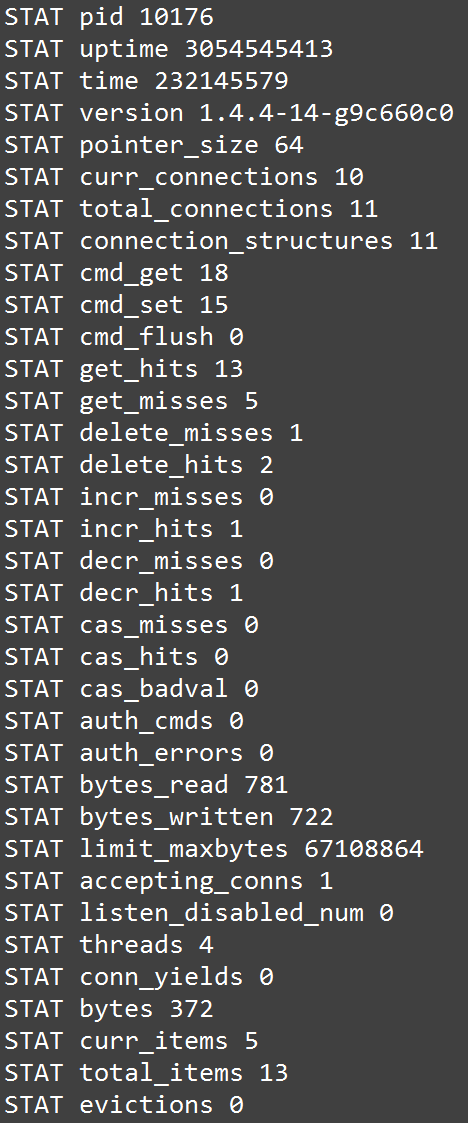
实例：



应用场景：如秒杀功能（抢购手机），一个人下订单的时候，要牵涉到读取数据库、写入订单、更改库存、事务等，对于传统数据库来说，瞬间压力是巨大。为了解决传统数据库压力，可以利用memcached的incr/decr功能(存操作数据是非常迅速)，在内存存储商品的数量，有人抢到一个订单号时，商品数量减1，当商品数量减到0时，停止抢购，抢到订单号的人，再跳转的另一个页面慢慢支付。这样就分担了传统数据库的瞬间压力。

## 2.7 stats命令

stats统计缓存状态。



几个比较重要的状态数据：

|  |  |  |
| --- | --- | --- |
| **名称** | **值** | **说明** |
| pid | 10176 | memcached进程号 |
| uptime | 3054545413 | 已经运行时间(微秒) |
| time | 232145579 | 当前时间戳(秒) |
| curr\_connections | 10 | 当前连接数 |
| total\_connections | 11 | 总共连接数 |
| cmd\_get | 18 | 一共想memcached请求获取多少次数据 |
| get\_hits | 13 | 请求成功命中多少次 |
| get\_misses | 5 | 请求失败次数 |
| curr\_items | 5 | 当前存储键值对的个数 |
| total\_items | 13 | 总共存储键值对的个数 |

缓存的重要概念缓存命中率：

一般缓存命中率达到50%左右，达到80%是非常高的了。

## 2.8 flush\_all命令

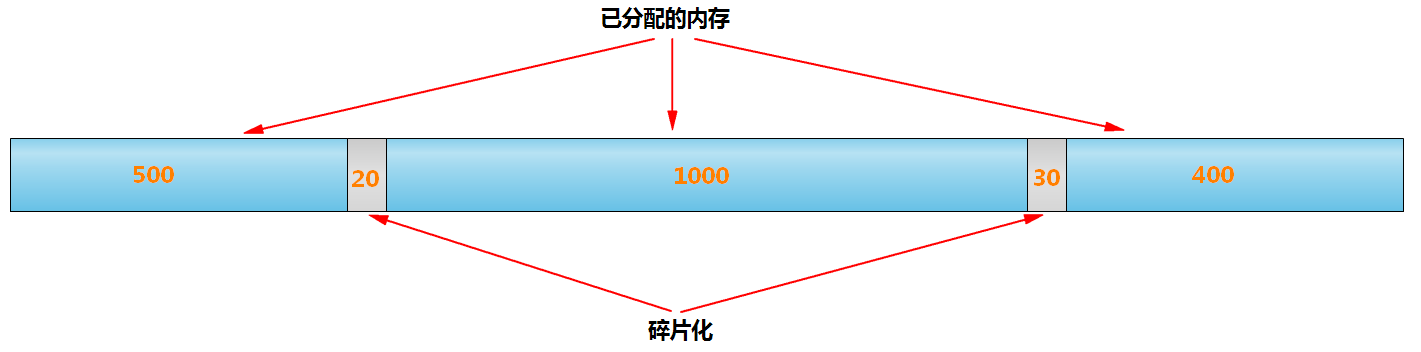
清空所有缓存对象，应该慎用。

# 3 memcached内存分配机制

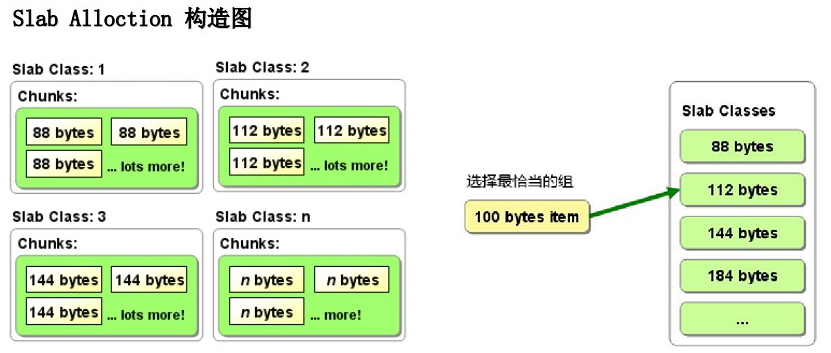
内存碎片化：空闲，但无法被利用的内存。

产生原因：多次申请和释放过程中形成的零碎内存空间，但无法利用。

示意图：

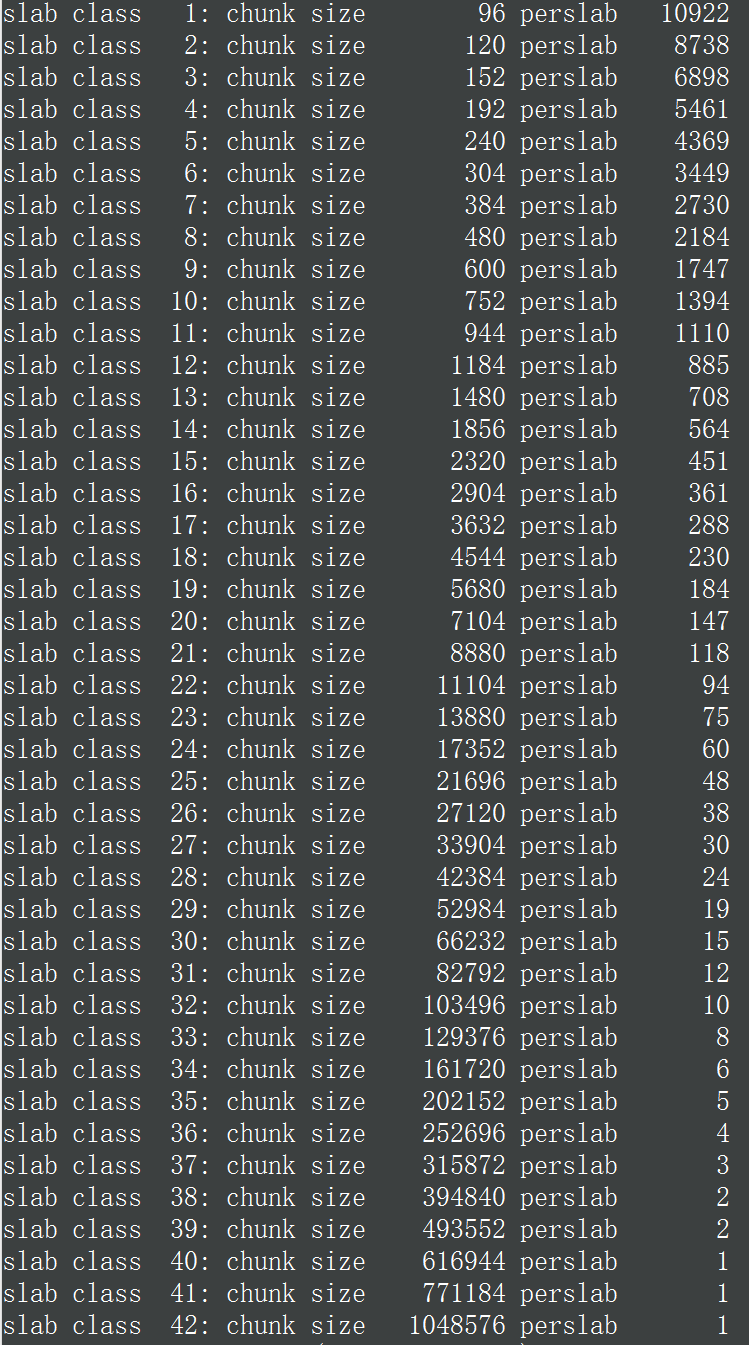


memcached内存分配机制是按照预先规定的大小，将分配的内存分割成特定长度的块(chunk)，并把尺寸相同的块分成组，从而尽可能缓解内存碎片的问题。



如上图的100字节存入112字节的chunk，剩下的12字节则就浪费了，所以内存碎片化不能完全解决，只能缓解。

注意：如果把100字节存入memcached，但是112字节仓库满已经存满了，这时并不会寻找更大的144字节仓库来储存，而是把112字节仓库的旧数据踢掉，再把100字节存进来。



从上图中可以看到，数据块chunk增长是有规律的，增长因子默认是1.25倍，根据网站缓存的大小，可以在启动时通过-f <factor>选项来修改增长因子。

# 4 memcached的过期删除机制

当某个值过期后，并没有从内存删除，因此stats统计时，curr\_item有其信息。

如果之前没有get过，将不会自动删除，当某个新值去占用它的位置时，当成空chunk来占用。

当取其值时，判断是否过期，如果过期，返回空，curr\_item就减少。

数据过期只是不让用户看到而已，并没有在过期的瞬间立即从内存删除，这种行为称为惰性失效，优点是节省CPU时间和检测的成本。

操作系统内存管理常用LRU和FIFO删除

LRU：least recently used 最近最少用的。

FIFO：first in first out 先进先出。

删除原理：LRU原则，当某个单元被请求时，维护一个计数器哪个key最近最少被使用，当储存仓满时，又有新数据添加时，就会把最近最少用的key踢出去，即使该key设置为永久的。

# 5 memcached的参数长度设置

key长度：250 bytes

value长度：1M，一般使用value存储文本，如新闻列表等。

内存限制：32位系统最大设置为2G

如果有10G数据需要缓存，一般不会单实例装10G，建议开启多个实例来缓存10G数据，例如在不同机器或同一台机器的不同端口作为多个实例。

# 6 安装php-memcached扩展

注意php的memcached扩展有两个，比较像，分别是php-memcache，php-memcached。从官网<http://php.net/manual/en/index.php>查找相应的模块下载安装。

## 6.1 window下安装php-memcached

window下只需下载一个php-memcached.dll，在网上下载的php-memcached.dll必须满足下面3个条件该扩展才能使用，该3个条件可以通过phpinfo()函数来查看。

|  |  |
| --- | --- |
| Compiler |  |
| php版本 |  |
| PHP Extension Build |  |

**(1) 根据参数下载相应的版本**

http://pecl.php.net/package/memcache/3.0.8/windows

例如下载文件php\_memcache-3.0.8-5.5-ts-vc11-x64.zip，

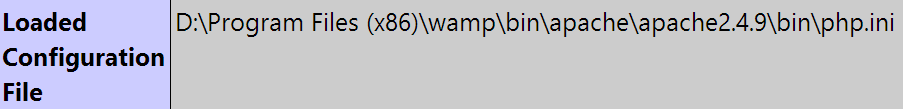
通过phpinfo()函数来查看扩展文件夹



把压缩包的php\_memcache.dll复制到扩展文件夹ext文件夹下面。

**(2) 修改配置文件php.ini**

注意：需要通过phpinfo()函数来查看php.ini位置，集成安装的php.ini文件有可能复制一份到apache文件夹下面。



然后在配置文件php.ini添加extension=php\_memcache.dll

使用./php.exe -m查看memcache模块是否存在。

**(3) 重启apache，查看通过phpinfo()函数查看是否存在memcache模块。**

## 6.2 linux下安装php-memcached

php扩展下载网站：http://pecl.php.net

安装依赖包：yum install libmemcached libmemcached-devel

适合php7下载：wget <https://github.com/php-memcached-dev/php-memcached/tree/php7>

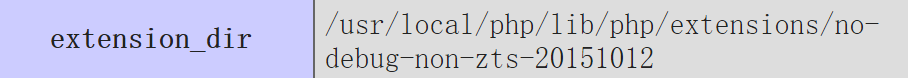
**(1) 安装php-memcached**

|  |  |
| --- | --- |
| 下载源码 | wget https://pecl.php.net/get/memcached-2.2.0.tgz |
| 解压 | tar -zxvf memcached-2.2.0.tgz |
| 进入目录 | cd memcached-2.2.0 |
| 生成配置文件 | /usr/local/php/bin/phpize --with=/usr/local/php/bin/php-config |
| 检查编译环境 | ./configure --with-php-config=/usr/local/php/bin/php-config --with-libmemcached-dir |
| 编译和安装 | make && make install |

**(2) 配置php.ini**

通过phpinfo()函数输出，在页面上看到的配置文件路径和扩展包的路径





打开配置文件：vim /usr/local/php/lib/php.ini

(如果配置文件不存在则从php安装包中复制复制一份配置文件)

查找.so，在附近添加下面内容：

|  |
| --- |
| extension=/usr/local/php/lib/php/extensions/no-debug-zts-20121212/memcached.so |

**(3) 检验安装扩展是否成功**

重启apache或nginx

通过phpinfo()函数输出，查看页面上有没有memcached模块。

## 6.3 两种php的memcached扩展使用

官方使用文档：http://php.net/docs.php

**(1) php-memcache扩展使用**

|  |
| --- |
| $memcache = new Memcache; // 实例化对象  $memcache->connect('memcache\_host', 11211); // 连接缓存服务器  $memcache->add('var\_key', 'test variable', false, 30); // 添加数据  $memcache->get('var\_key'); // 获取数据  $memcache->replace("test\_key", "some variable", false, 30); // 修改数据  $memcache->delete('test\_key'); // 删除数据  $memcache->close(); // 关闭连接 |

范例：

|  |
| --- |
| <?php  $memcache = new Memcache; // 实例化对象 if ($memcache->connect('192.168.8.102', 11211)==false){ // 连接缓存服务器  echo "连接缓存服务器失败";  return; }  $mkey='sit';  // 从缓存服务器中获取key的值，并判断值key是否存在，不存在则添加数据 if ($memcache->get($mkey)==false){// 获取数据并判断是否存在  // 这里可以从数据库读取值  $memcache->add($mkey, 'test variable', false, 3600); // 添加数据  echo "缓存".$mkey."成功"; }else{  echo $mkey."存在"; }  $memcache->close(); // 关闭连接 |

**(2) php-memcached扩展使用**

|  |
| --- |
| $mem = new Memcached; // 实例化对象  $mem->addServer('memcache\_host', 11211); // 连接缓存服务器  $mem->add('var\_key', 'test variable', 3600); // 添加数据  $mem->get('var\_key'); // 获取数据  $mem->replace ("test\_key", "some variable",3600); // 修改数据  $meme->delete('test\_key'); // 删除数据  $mem->quit (); // 关闭连接 |

范例：

|  |
| --- |
| $mem = new Memcached; // 实例化对象 if ($mem->addServer('192.168.8.102', 11211)==false){ // 连接缓存服务器  echo "连接缓存服务器失败";  return; }  $mkey='sit';  // 从缓存服务器中获取key的值，并判断值key是否存在，不存在则添加数据 if ($mem->get($mkey)==false){// 获取数据并判断是否存在  // 这里可以从数据库读取值  $mem->add($mkey, 'test variable', 3600); // 添加数据  echo "缓存".$mkey."成功"; }else{  echo $mkey."存在"; }  $mem->quit(); // 关闭连接 |

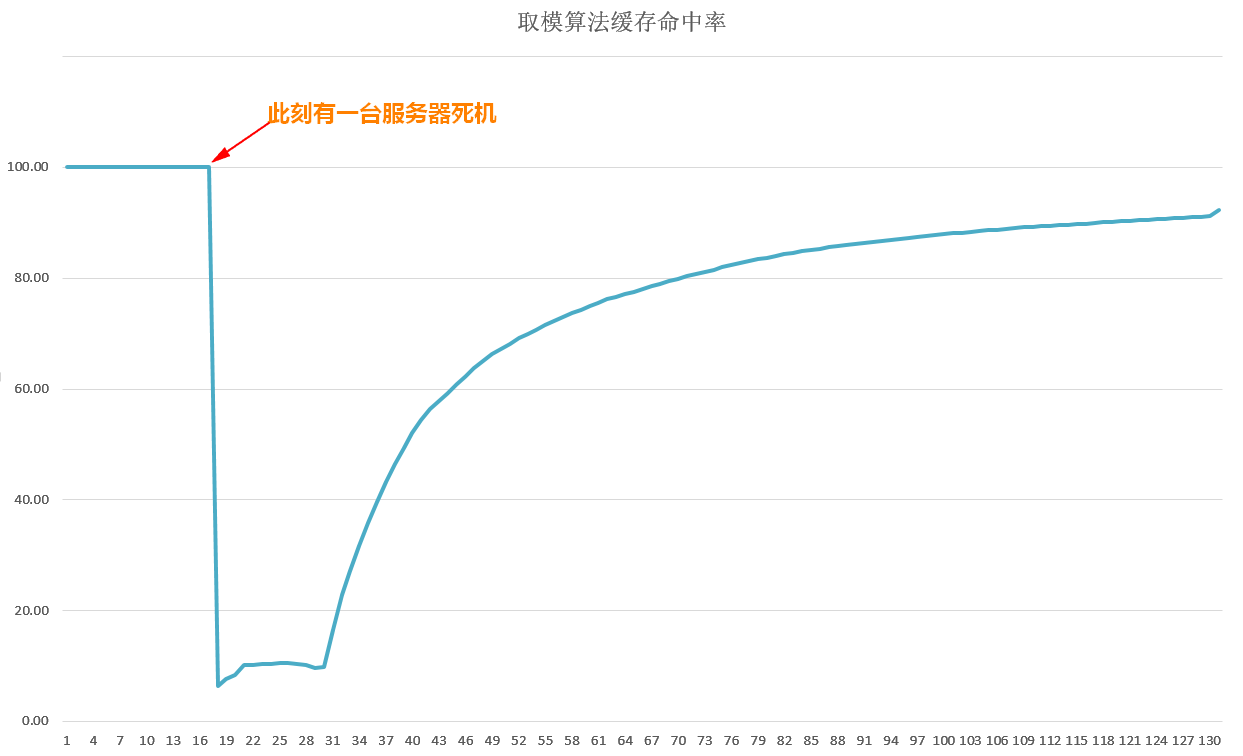
# 7 分布式算法之取模算法(求余)缺陷

假如有8台服务器，如有一台服务器down掉，则求余的底数变成1，则此时的命中率：=12.5%

从数学上归纳，有N台服务器，有一台服务器down掉，短时间内的命中率极速下降，命中率=

也就是说服务器越多，down机后命中率越低，后果越严重。

实验：有5台缓存服务器，在某一时刻突然有一台服务器死机后记录的缓存命中率，如下图所示，死机瞬间缓存命中率降到大概，此时数据库压力非常大，有可能会压垮数据库。随着时间推移，逐渐恢复命中率。

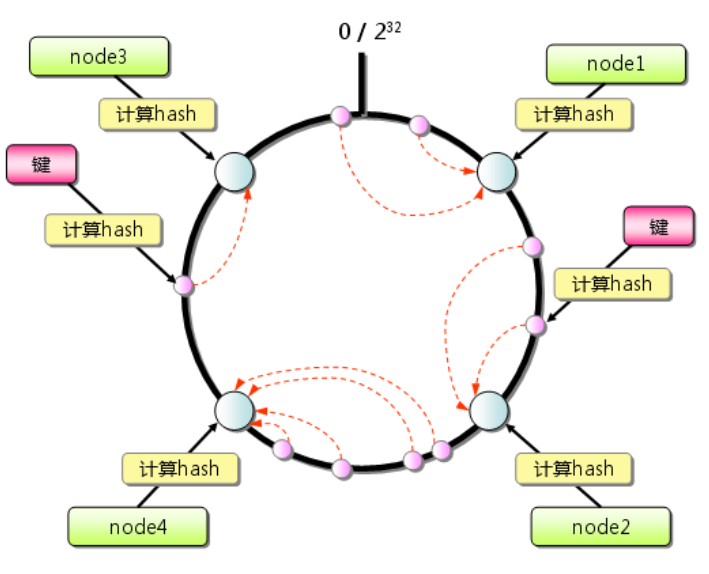


# 8 分布式算法之一致性哈希算法

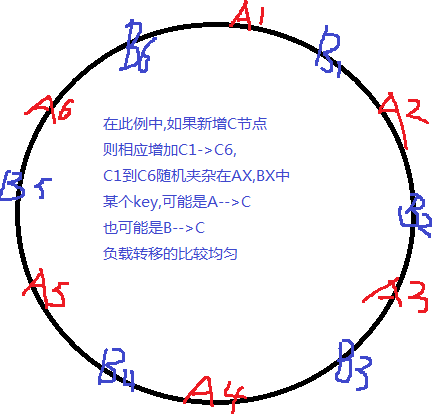
一致性哈希算法解决了取模算法的缺陷，把缓存服务器死机后的压力分散到其他缓存服务器上，瞬间缓存命中率=

原理

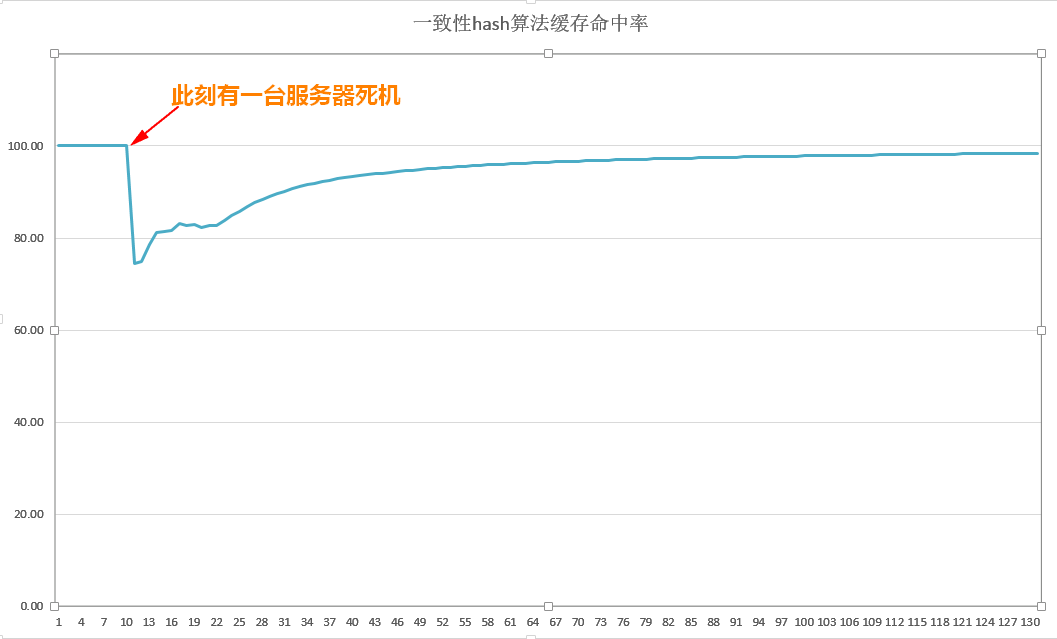
首先求出memcached服务器（节点）的哈希值，并将其配置到0～的圆（continuum）上。然后用同样的方法求出存储数据的键的哈希值，并映射到圆上。然后从数据映射到的位置开始顺时针查找，将数据保存到找到的第一个服务器上。如果超过仍然找不到服务器，就会保存到第一台memcached服务器上，如下图所示。



Consistent Hashing的实现方法还采用了虚拟节点的思想。使用一般的hash函数的话，服务器的映射地点的分布非常不均匀。因此，使用虚拟节点的思想，为每个物理节点（服务器）在圆环上分配100～200个点。这样就能抑制分布不均匀，最大限度地减小服务器增减时的缓存重新分布，如下图所示：



实验：有5台缓存服务器，每台服务器分配100个虚拟节点，再某时刻有一台服务器死机后记录的缓存命中率，如下图所示。死机瞬间缓存命中率降到大概，死机的缓存压力分散到其他4台服务器上，此时数据库压力不是很大。随着时间推移逐渐恢复命中率。



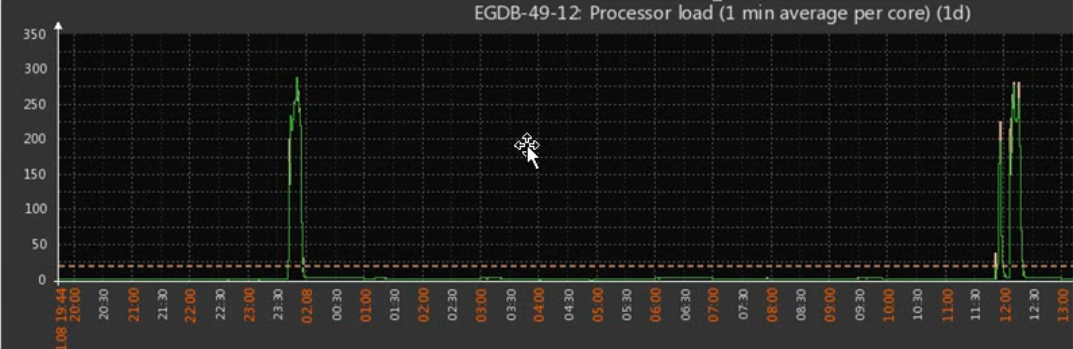
# 9 实际中碰到的经典问题或现象

## 9.1 缓存雪崩现象

原因：一般是由某个节点失效导致其他节点的缓存命中率下降，缓存中缺失的数据要到数据库查询，短时间内造成数据库服务崩溃。

现象：重启数据库后短时间内又被压垮，但缓存也多了一些，反复重启数据库多次，缓存重建完毕，数据库才能稳定运行。

案例：某网站访问量比较大，遇到缓存周期性失效问题，每6小时失效一次，刚好在失效期间有一个请求“峰值”，导致数据库数据库崩溃。



问题：为什么添加数据时间不一样，失效时间也集中在一两个小时内呢？

答：因为刚上线时，缓存为空，用户访问时就只能从数据库取数据，加入每秒3000访问量，一两个小时内已经把用户数据缓存满了，短期内就可以把所有缓存生成。

解决方案：

1. 把缓存设置为随机3到9小时的生命周期，这样不同时失效，把工作分担到其他时间点上。
2. 选择在半夜3点到5点之间使缓存失效。

## 9.2 缓存的无底洞现象

memcached连接频繁，效率下降，增加了节点，但是发现因为连接频繁导致的问题依然存在，并没有好转，称为缓存无底洞现象。

原因：以用户为例，usename age gerden等N个key，当缓存服务器增多，用户的信息key也会被散落在更多的节点，所以访问个人主页时，要连接节点就会也多， memcached连接数随节点增多而增加。

解决方案：把某一组key按其共同前缀来分布，例如user-188-age、user-188-name、user-188-gerden等key，在分布式算法求其节点时，应该以"user-188"来计算节点值，这样相关集中的key都落在同一个节点上，在访问个人信息时，只需连接1个节点就完成。

## 9.3 永久数据被踢现象

插入数据时是设置永久有效，过一段时间却莫名其妙的消失了，也就是memcached数据丢失了。

原因：虽然设置key为永久有效，但是一直都没有get，随着时间推移，该key越来越旧，期间也有其他被访问过key设置的失效时间也过了，memcached使用的是惰性删除机制(key失效了，但不get就不知道该key失效)，当插入新数据时，发现储存已满了，这时需要删除老旧的key(包括设置为永久有效的key)，把新数据插入进来。 memcached删除数据核心：惰性删除和最近最少使用记录。

解决方案：永久数据和非永久数据分开。