Memcached: The Definitive Guide

第 2 版

Memcached 权威指南

燕十八 著

自学 it 网®

http://www.zixue.it

目　录

前　言

关于本书

本书是自学 it 网(http://www.zixue.it) ”高端 PHP 培训班”的内部教材，请同学们不要传播!
如果你不是高端班学员，请于 24 小时内删除本书!

学习目标

目前的 PHP,java web 程序员,3000-5000 元/月 左右的初级程序非常多,
但这些程序员想突破瓶颈,迈向中级,高级程序员则显得后劲不足.
高端培训班即针对初级&中级程序员,做突破性拔高培训!

班级	课程目的	主要内容
中级实战班	专治”做项目没思路”	Blog+CMS 双项目实战+ecshop+discuz 二次开发+Yii 框架+ 微信平台开发
高级架构班	专治”高性能+大并发”	Linux 集群 + MySQL 优化 + Nginx 高性能 + LVS 负载均衡
咨询老师: QQ 2258489282 高老师 可索取课程详细大纲		

作者简介:

燕十八 (自学 it 网 教学总监)
职 务：PHP 高级讲师、企业培训讲师

燕十八,原名刘道成，资深 PHP 工程师，拥有多年 PHP 网站开发实际经验，先后担任过高级软件开发工程师,项目经理.为国内某大型旅行社开发过在线旅游网站.热衷于主流 PHP 框架和开源产品的研究,先后在国内数高校担任 PHP 讲师,mysql 讲师.讲课风格:生动形象,于生活常见的事物中,把计算机的复杂概念讲解清楚.言辞幽默,课堂气氛轻松活泼.深得学生喜爱。

代表作: PHP 高手之路-传世经典四部
下载地址: http://www.zixue.it

第一章:memcached 介绍

1.1 memcached 是什么?

free & open source, high-performance, distributed memory object caching system
自由&开放源码, 高性能 ,分布式的内存对象缓存系统
由 livejounal 旗下的 danga 公司开发的老牌 nosql 应用.

1.2 什么是 NoSQL?

nosql，指的是非关系型的数据库。
相对于传统关系型数据库的"行与列",NoSQL 的鲜明特点为 k-v 存储(memcached,redis),
或基于文档存储(mongodb)

注: nosql -- not only sql , 不仅仅是关系型数据库, 显著特点: key-value 键值对存储,如 memcached, redis, 或基于文档存储 如,mongodb

1.3 谁在用 memcached?

第二章:memcached 基本使用

2.1 linux 下编译 memcached

2.1.1:准备编译环境

在 linux 编译,需要 gcc,make,cmake,autoconf,libtool 等工具,
这几件工具, 以后还要编译 redis 等使用,所以请先装.
在 linux 系统联网后,用如下命令安装

```
#yum install  gcc make cmake autoconf libtool
```

2.1.2: 编译 memcached

memcached 依赖于 libevent 库,因此我们需要先安装 libevent.
分别到 libevent.org 和 memcached.org 下载最新的 stable 版本(稳定版).

先编译 libevent ,再编译 memcached,
编译 memcached 时要指定 libevent 的路径.

过程如下: 假设源码在/usr/local/src 下, 安装在/usr/local 下

```
# tar zxvf libevent-2.0.21-stable.tar.gz
# cd libevent-2.0.21-stable
# ./configure --prefix=/usr/local/libevent

# 如果出错,读报错信息,查看原因,一般是缺少库

#  make && make install


# tar zxvf memcached-1.4.5.tag.gz
# cd memcached-1.4.5
#./configure--prefix=/usr/local/memcached  \
--with-libevent=/usr/local/libevent
# make && make install
```

注意: 在虚拟机下练习编译,一个容易碰到的问题---虚拟机的时间不对, 导致的 gcc 编译过程中,检测时间通不过,一直处于编译过程. 解决: # date -s ‘yyyy-mm-dd hh:mm:ss’ # clock -w # 把时间写入 cmos

2.2 memcached 的启动

```
# /usr/local/memcached/bin/memcached -m 64 -p 11211 -u nobody -vv
slab class   1: chunk size       96 perslab   10922
slab class   2: chunk size      120 perslab    8738
slab class   3: chunk size      152 perslab    6898
slab class   4: chunk size      192 perslab    5461
....
....
slab class   9: chunk size      600 perslab    1747
slab class  10: chunk size      752 perslab    1394
slab class  39: chunk size   493552 perslab       2
slab class  40: chunk size   616944 perslab       1
slab class  41: chunk size   771184 perslab       1
slab class  42: chunk size  1048576 perslab       1
```

我们发现 memcached 已经启动,并把信息输出到控制台..
如果我们想让 memcached 作为 daemon 在后台运行,只需要加-d 选项

```
# /usr/local/memcached/bin/memcached -m 64 -p 11211 -u nobody -d
```

如果想了解 -m -p 等参数的意义, 可以通过 memcached -h 查看帮助.

```
-p <num>      tcp port number to listen on (default: 11211) // 监听
的端口
-u <num>      udp port number to listen on (default: 0, off)
-s <file>     unix socket path to listen on (disables network support)
-a <mask>     access mask for unix socket, in octal (default 0700)
-l <ip_addr>  interface to listen on, default is indrr_any
-d start          tell memcached to start
-d restart        tell running memcached to do a graceful restart
-d stop|shutdown  tell running memcached to shutdown
-d install        install memcached service // 把 memcached 注册成服务
-d uninstall      uninstall memcached service
-r           maximize core file limit
-u <username> assume identity of <username> (only when run as root)
-m <num>      max memory to use for items in megabytes, default is 64
mb //分配给 memcached 的最大内存
-m           return error on memory exhausted (rather than removing
items)
```

```
-c <num>     max simultaneous connections, default is 1024 // 最大
的连接数
-k           lock down all paged memory.  note that there is a
             limit on how much memory you may lock.  trying to
             allocate more than that would fail, so be sure you
             set the limit correctly for the user you started
             the daemon with (not for -u <username> user;
             under sh this is done with 'ulimit -s -l num_kb').
-v           verbose (print errors/warnings while in event loop) //
输出错误信息
-vv          very verbose (also print client commands/reponses) //
输出所有信息
-h           print this help and exit
-i           print memcached and libevent license
-b           run a managed instanced (mnemonic: buckets)
-p <file>     save pid in <file>, only used with -d option
-f <factor>   chunk size growth factor, default 1.25 //增长因子
-n <bytes>    minimum space allocated for key+value+flags, default 48
```

附: 在 windows 下启动 memcached:

在 windows 下,下载 memcached 的 win 二进制版本.

下载地址:http://www.zixue.it/thread-9030-1-1.html

下载解压后,不用安装,通过命令行下进入到 memcached.exe 所在的目录

```
在 win 下启动
>memcached -m 64 -p 11211 -vvv
```

2.3 memcached 的连接

memcached 客户端与服务器端的通信比较简单,使用的基于文本的协议,而不是二进制协议.(http 协议也是这样), 因此我们通过 telnet 即可与 memcached 作交互.

另开一个终端,并运行 telnet 命令 (开启 memcached 的终端不要关闭)

```
# 格式 telnet host port

# telnet localhost 11211
Trying ::1...
Connected to localhost.
Escape character is '^]'.
```

2.4 memcached 的命令

分 增删改查统计 5 类,沿着这个思路来学习.

- 增:　add 往内存增加一行新记录

语法: add key flag expire length 回车

```
add name 1 0 4
lily
STORED
```

key　给值起一个独特的名字
flag 标志,要求为一个正整数
expire 有效期
length 缓存的长度(字节为单位)

flag 的意义:
memcached 基本文本协议,传输的东西,理解成字符串来存储.
想:让你存一个 php 对象,和一个 php 数组,怎么办?
答:序列化成字符串,往出取的时候,自然还要反序列化成 对象/数组/json 格式等等.
这时候, flag 的意义就体现出来了.
比如, 1 就是字符串, 2 反转成数组　3,反序列化对象.....

expire 的意义:
设置缓存的有效期,有 3 种格式
1:设置秒数, 从设定开始数,第 n 秒后失效.
2:时间戳, 到指定的时间戳后失效.
比如在团购网站,缓存的某团到中午 12:00 失效. add key 0 1379209999 6
3: 设为 0. 不自动失效.

注: 有种误会,设为 0,永久有效.错误的.

1:编译 memcached 时,指定一个最长常量,默认是 30 天.
所以,即使设为 0,30 天后也会失效.
2:可能等不到 30 天,就会被新数据挤出去.

- delete 删除

delete key [time seconds]
删除指定的 key. 如加可选参数 time,则指删除 key,并在删除 key 后的 time 秒内,不允许 get,add,replace 操作此 key.

- replace 替换

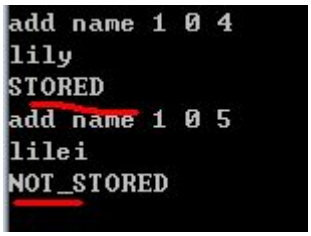
replace key flag expire length
参数和 add 完全一样,不单独写

- get 查询

get key
返回 key 的值

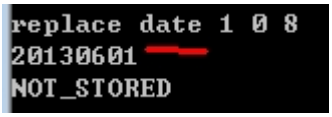
- set 是设置和修改值

参数和 add ,replace 一样,但功能不一样.
如下比较:



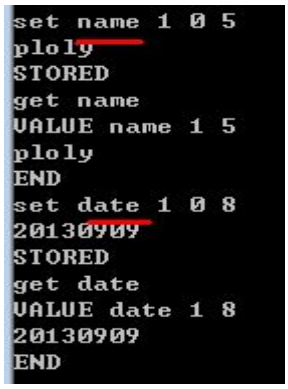
用 add 时, key 不存在,才能建立此键值.

但对于已经存在的键,可以用 replace 进行替换/更改



repalce,key 存在时,才能修改此键值,如上图,date 不存在,则没改成功.

而 set 想当于有 add replace 两者的功能.
set key flag expire leng 时
如果服务器无此键 ----> 增加的效果
如果服务器有此键 ----> 修改的效果.
如下图的演示,该图中,name 是已经存在,而 date 原本不存在. set 都可以成功设置他们.



- incr ,decr 命令:增加/减少值的大小

语法:　incr/decr key num

示例:

```
set age 0 0 2
28
stored
get age
value age 0 2
28
end
incr age 1
29
incr age 2
31
decr age 1
30
decr age 2
28
```

注意:incr,decr 操作是把值理解为 32 位无符号来+-操作的. 值在[0-2^32-1]范围内

➢ 应用场景------秒杀功能,

一个人下单,要牵涉数据库读取,写入订单,更改库存,及事务要求, 对于传统型数据库来说,压力是巨大的.

可以利用 memcached 的 incr/decr 功能, 在内存存储 count 库存量, 秒杀 1000 台

每人抢单主要在内存操作,速度非常快,

抢到 count<=1000 的号人,得一个订单号,再去另一个页面慢慢支付

- 统计命令: stats

把 memcached 当前的运行信息统计出来

stats

stat pid 2296 进程号
stat uptime 4237 持续运行时间
stat time 1370054990
stat version 1.2.6
stat pointer_size 32
stat curr_items 4 当前存储的键个数
stat total_items 13
stat bytes 236
stat curr_connections 3
stat total_connections 4
stat connection_structures 4
stat cmd_get 20
stat cmd_set 16
stat get_hits 13
stat get_misses 7 // 这 2 个参数 可以算出命中率
stat evictions 0
stat bytes_read 764
stat bytes_written 618
stat limit_maxbytes 67108864
stat threads 1
end

缓存有一个重要的概念: 命中率.
命中率是指: (查询到数据的次数/查询总数)*100%
如上, 13/(13+7) = 60+% , 的命中率.

- flush_all 清空所有的存储对象

第三章 memcached 的内存管理与删除机制

3.1:内存的碎片化

如果用 c 语言直接 malloc,free 来向操作系统申请和释放内存时,
在不断的申请和释放过程中,形成了一些很小的内存片断,无法再利用.
这种空闲,但无法利用内存的现象,---称为内存的碎片化.

3.2: slab allocator 缓解内存碎片化

memcached 用 slab allocator 机制来管理内存.
slab allocator 原理: 预告把内存划分成数个 slab class 仓库.(每个 slab class 大小 1M)
各仓库,切分成不同尺寸的小块(chunk). （图 3.2）
需要存内容时,判断内容的大小,为其选取合理的仓库.

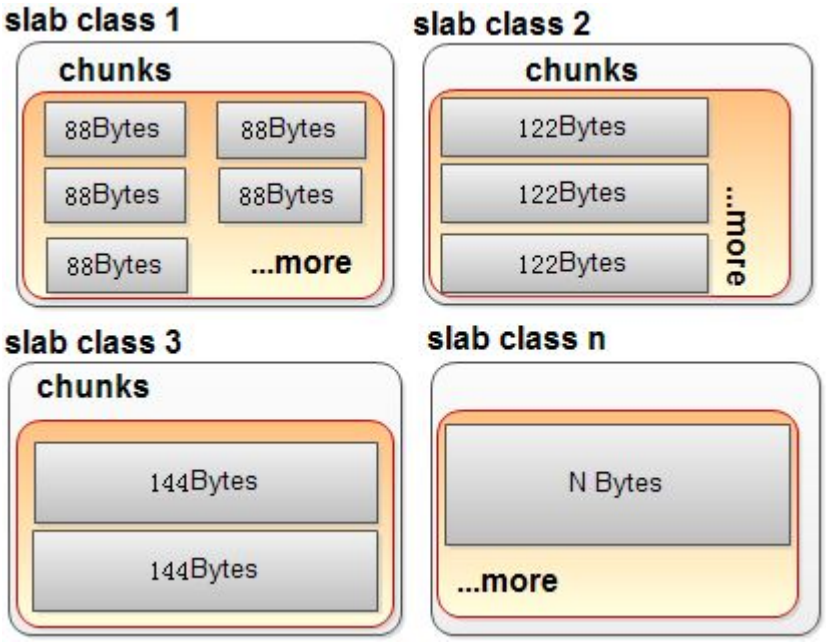


图 3.2: slab alloctor 图解

3.3 系统如何选择合适的 chunk?

memcached 根据收到的数据的大小，选择最适合数据大小的 chunk 组(slab class)（图 3.3）。memcached 中保存着 slab class 内空闲 chunk 的列表，根据该列表选择空的 chunk，然后将数据缓存于其中。

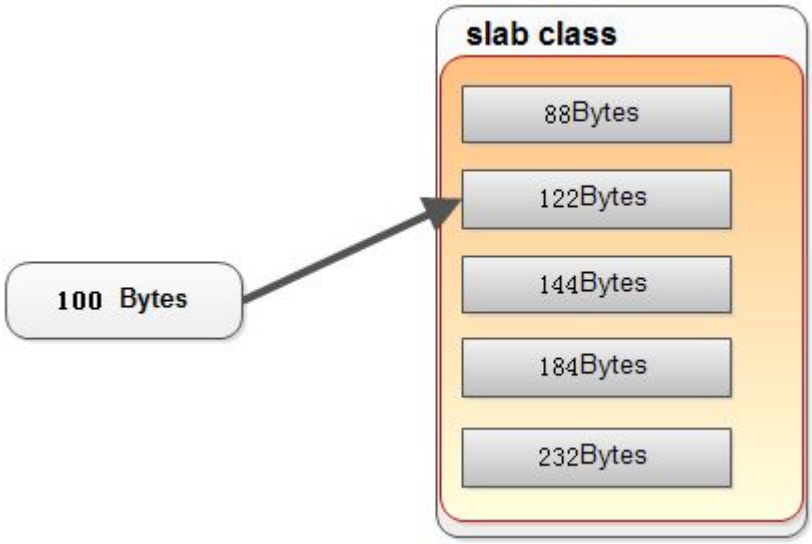


图 3.3: item 大小与 chunk 的选择

警示: 如果有 100byte 的内容要存,但 122 大小的仓库中的 chunk 满了 并不会寻找更大的,如 144 的仓库来存储, 而是把 122 仓库的旧数据踢掉！详见过期与删除机制

3.4 固定大小 chunk 带来的内存浪费

由于 slab allocator 机制中, 分配的 chunk 的大小是”固定”的, 因此, 对于特定的 item,可能造成内存空间的浪费.
比如, 将 100 字节的数据缓存到 122 字节的 chunk 中, 剩余的 22 字节就浪费了(图 3.4)

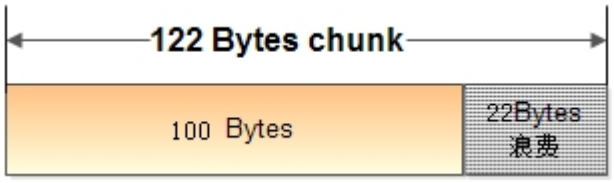


图 3.4: chunk 空间的利用

对于 chunk 空间的浪费问题,无法彻底解决,只能缓解该问题.
开发者可以对网站中缓存中的 item 的长度进行统计,并制定合理的 slab class 中的 chunk 的大小.
可惜的是,我们目前还不能自定义 chunk 的大小,但可以通过参数来调整各 slab class 中 chunk 大小的增长速度. 即增长因子, grow factor!

3.5 grow factor 调优

memcached 在启动时可以通过-f 选项指定 Growth Factor 因子, 并在某种程度上控制 slab 之间的差异. 默认值为 1.25. 但是,在该选项出现之前,这个因子曾经固定为 2,称为"powers of 2"策略。

我们分别用 grow factor 为 2 和 1.25 来看一看效果:

```
>memcached -f  2  -vvv

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  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
```

可见，从 128 字节的组开始，组的大小依次增大为原来的 2 倍.

来看看 f=1.25 时的输出:

```
>memcached -f 1.25 -vvv
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  36: chunk size 250376 perslab    4
slab class  37: chunk size 312976 perslab    3
slab class  38: chunk size 391224 perslab    2
slab class  39: chunk size 489032 perslab    2
```

对比可知, 当 f=2 时, 各 slab 中的 chunk size 增长很快,有些情况下就相当浪费内存.
因此,我们应细心统计缓存的大小,制定合理的增长因子.

> 注意:
> 当 f=1.25 时,从输出结果来看,某些相邻的 slab class 的大小比值并非为 1.25,可能会觉得有些计算误差，这些误差是为了保持字节数的对齐而故意设置的.

3.6 memcached 的过期数据惰性删除

1: 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
2: 当某个新值去占用他的位置时,当成空 chunk 来占用.
3: 当 get 值时,判断是否过期,如果过期,返回空,并且清空, curr_item 就减少了.

即--这个过期,只是让用户看不到这个数据而已,并没有在过期的瞬间立即从内存删除.
这个称为 lazy expiration, 惰性失效.
好处--- 节省了 cpu 时间和检测的成本

3.7: memcached 此处用的 lru 删除机制.

如果以 122byte 大小的 chunk 举例, 122 的 chunk 都满了, 又有新的值(长度为 120)要加入, 要挤掉谁?
memcached 此处用的 lru 删除机制.
(操作系统的内存管理,常用 fifo,lru 删除)
lru: least recently used 最近最少使用
fifo: first in ,first out

原理: 当某个单元被请求时,维护一个计数器,通过计数器来判断最近谁最少被使用.
就把谁 t 出.

注: 即使某个 key 是设置的永久有效期,也一样会被踢出来! 即--永久数据被踢现象

3.8 memcached 中的一些参数限制

key 的长度: 250 字节, (二进制协议支持 65536 个字节)
value 的限制: 1m, 一般都是存储一些文本,如新闻列表等等,这个值足够了.
内存的限制: 32 位下最大设置到 2g.

如果有 30g 数据要缓存,一般也不会单实例装 30g, (不要把鸡蛋装在一个篮子里),
一般建议 开启多个实例(可以在不同的机器,或同台机器上的不同端口)

第四章 编译 PHP 及 memcached 扩展

4.1 编译 apache+php

到 http://httpd.apache.org 下载 httpd 的源码,http://www.php.net 下载 php 的源码

- apache 编译:

```
#1 解压

# tar zxvf http-2.2.45.tar.gz
# cd http-2.2.45

# ./configure --prefix=/usr/local/httpd (你也可以指定自己的路径)

#make && make install
```

- php 编译并与 apache 整合:

```
#1 编译 php

# yum install libxml2 libxml2-devel
# tar zxvf php-xxx.tar.gz
# cd php-xxx
#./configure--prefix=/usr/local/php \
--with-apxs2=/usr/local/httpd/bin/apxs
# make && make install

# 2. 与 apache 整合


# vim 编辑 http.conf,添加如下

# addtype application/x-httpd-php .php

# 3: 重启 apache
```

注: 如果在 configure 过程中,提示缺少 libxml2 的库,则如下操作: #yum install libxml2 libxml2-devel

4.2 编译 php-memcache 扩展

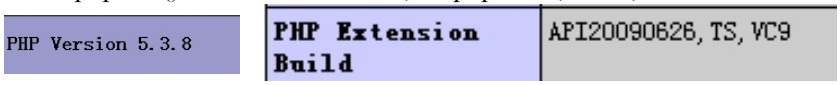
以后的开发中,动手编译 PHP 的各种扩展是很容易碰到,此以 memcache 扩展编译为例,讲解 PHP 扩展的通用编译流程

1: 到软件的官方(如 memcached)或 pecl.php.net 去寻找扩展源码并下载解压
2: 进入到 path/memcache 目录

```
3: 根据当前的 php 版本动态的创建扩展的 configure 文件

#/xxx/path/php/bin/phpize \
--with-php-config=/xxx/path/php/bin/php-config
4: ./configure  -with-php-config=/xxx/path/php/bin/php-config
5: make && make install

6:把生成的.so 扩展，在 php.ini 里引入.

7:重启 apache
```

4.3 windows 下安装 php-memcached 扩展

1) 通过 phpinfo()观察如下 3 个参数,即 php 版本, ts/nts, vc6/vc9



2) 根据上步中的参数,到 http://downloads.php.net/pierre/ 下载匹配的 memcache.dll
3) 再次观察 phpinfo()信息,找出 extension_dir, 并把下载的 memcache.dll 放入该路径.
4) 并修改 php.ini, 加入 extension=php_memcache.dll,引入该 dll
5) 重启 apache

第五章 memcached 实战

任何东西,都有其适用场景,在合适的场景下,才能发挥更好的作用.
对于 memcached,使用内存来存取数据,一般情况下,速度比直接从数据库和文件系统读取要快.
memcached 的最常用场景是利用其”读取快”来保护数据库,防止频率读取数据库.
也有的项目中,利用其”存储快”的特点来实现主从数据库的消息同步.
本章我们将对这些应用场景进行探讨.

5.1 缓存数据库查询结果

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(图 5.1)。

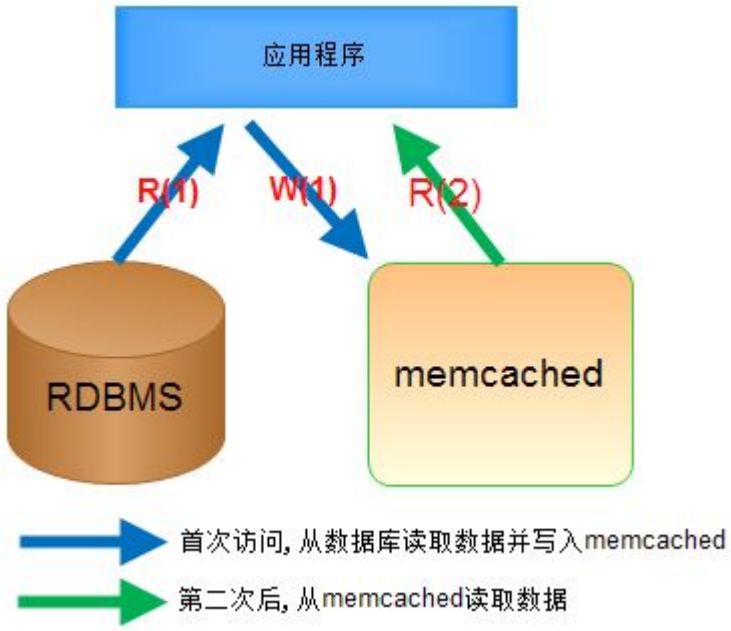
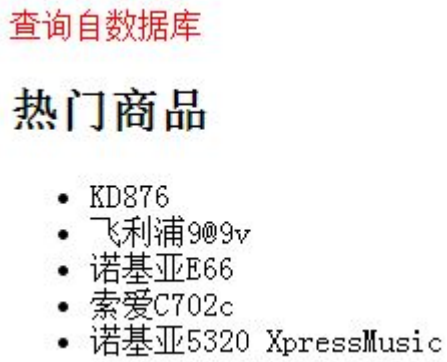


图 5.1: memcached 缓存查询结果

伪代码及效果如下:

```
<?php
$sql = 'select goods_id,goods_name from ecs_goods where is_hot=1 limit
5';


// 判断 memcached 中是否缓存热门商品,如果没有,则查询数据库
$hot = array();
if( !($hot=$memcache->get($sql)) ) {

    $hot = $mysql->getAll($sql);

    echo '<font color="red">查询自数据库</font>';


    //从数据库取得数据后,把数据写入 memcached

    $memcache->add($sql,$hot,0,300); // 并设置有效期 300 秒


} else {

    echo '<font color="red">查询自 memcached</font>';

}
?>
```



(第 1 次查询　来自数据库)

(再次查询　来自 memcached 缓存)

5.2 中继 MySQL 主从延迟数据

注:本节的例子来自百度文库的一位工程师,在此表示感谢.

MySQL 在做 replication 时,主从复制之间必然要经历一个复制过程,即主从延迟的时间.
尤其是主从服务器处于异地机房时,这种情况更加明显.
把 facebook 官方的一篇技术文章,其加州的主数据中心到弗吉尼亚州的主从同步延期达到 70ms;

考虑如下场景:
①: 用户 U 购买电子书 B, insert into Master (U,B);
②: 用户 U 观看电子书 B, select 购买记录[user='A',book='B'] from Slave.
③: 由于主从延迟,第②步中无记录,用户无权观看该书.

这时,可以利用 memached 在 master 与 slave 之间做过渡(图 5.2):
①: 用户 U 购买电子书 B, memcached->add('U:B',true)
②: 主数据库 insert into Master (U,B);
③: 用户 U 观看电子书 B, select 购买记录[user='U',book='B'] from Slave.
如果没查询到,则 memcached->get('U:B'),查到则说明已购买但 Slave 延迟.
④: 由于主从延迟,第②步中无记录,用户无权观看该书.

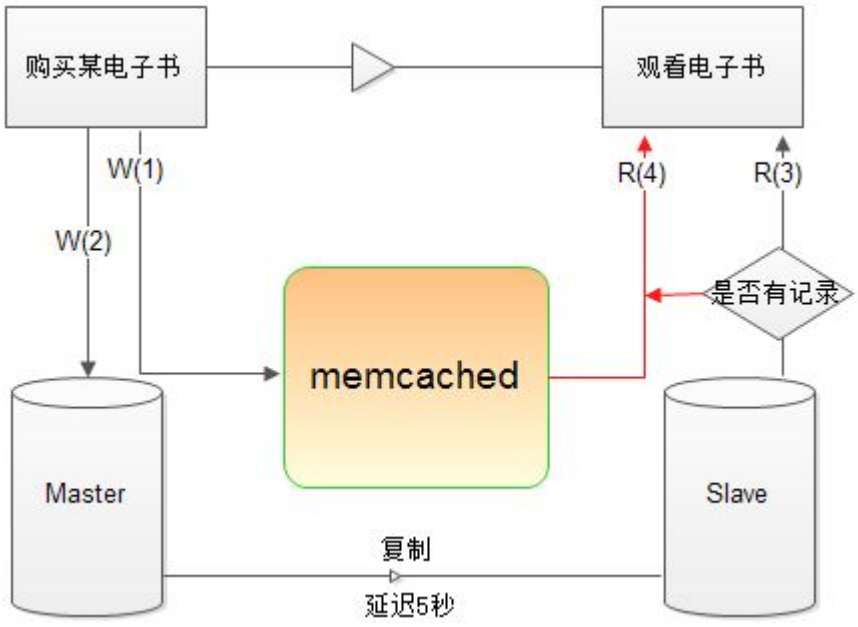


图 5.2: memcached 中继主从延迟数据

第六章 分布式集群算法

6.1 memcached 如何实现分布式?

在第 1 章中,我们介绍 memcached 是一个”分布式缓存”,然后 memcached 并不像 mongoDB 那样,允许配置多个节点,且节点之间”自动分配数据”.
就是说--memcached 节点之间,是不互相通信的.
因此,memcached 的分布式,要靠用户去设计算法,把数据分布在多个 memcached 节点中.

6.2 分布式之取模算法

最容易想到的算法是取模算法,即 N 个节点要,从 0->N-1 编号.
key 对 N 取模,余 i,则 key 落在第 i 台服务器上(图 5.2).

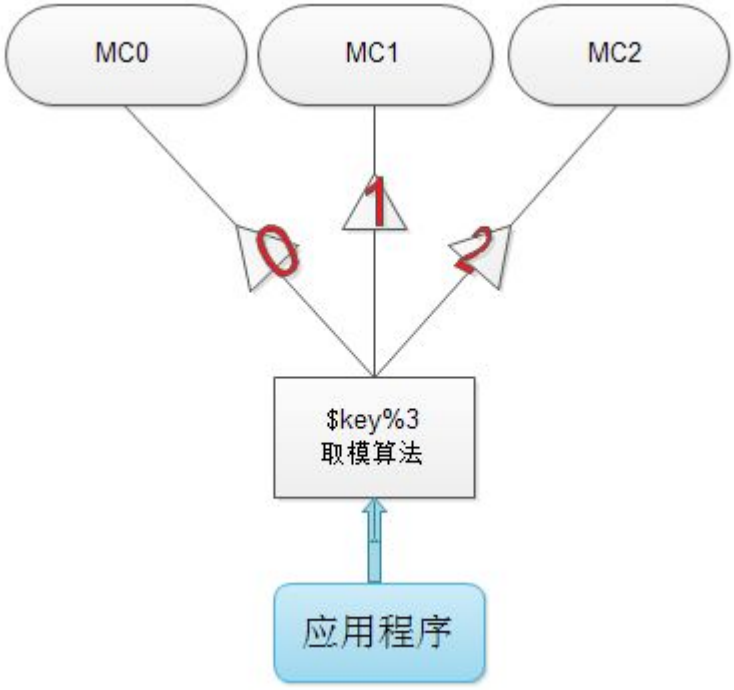


图 6.2: 取模算法实现分布式缓存

6.3 取模算法对缓存命中率的影响

假设有 8 台服务器，运行中,突然 down 一台，则求余的底数变成 7
后果:

```
key0%8==0, key0%7 ==0 hits
....
key6%8==6, key6%7== 6 hits
key7%8==7, key7%7==0 miss
key9%8==1, key9%7 == 2 miss
    ...
key55%8 ==7  key55%7 == 6 miss
```

一般地,我们从数学上归纳之:
有 N 台服务器，变为 N-1 台，
每 N*(N-1)个数中，只有(n-1)个单元,%N, %(N-1)得到相同的结果

所以 命中率在服务器 down 的短期内，急剧下降至 (N-1)/(N*(N-1)) = 1/(N-1)
所以: 服务器越多，则 down 机的后果越严重!

6.4　一致性哈希算法原理

通俗理解一致性哈希:

把各服务器节点映射放在钟表的各个时刻上, 把 key 也映射到钟表的某个时刻上.

该 key 沿钟表顺时针走,碰到的第 1 个节点即为该 key 的存储节点(图 5.4).

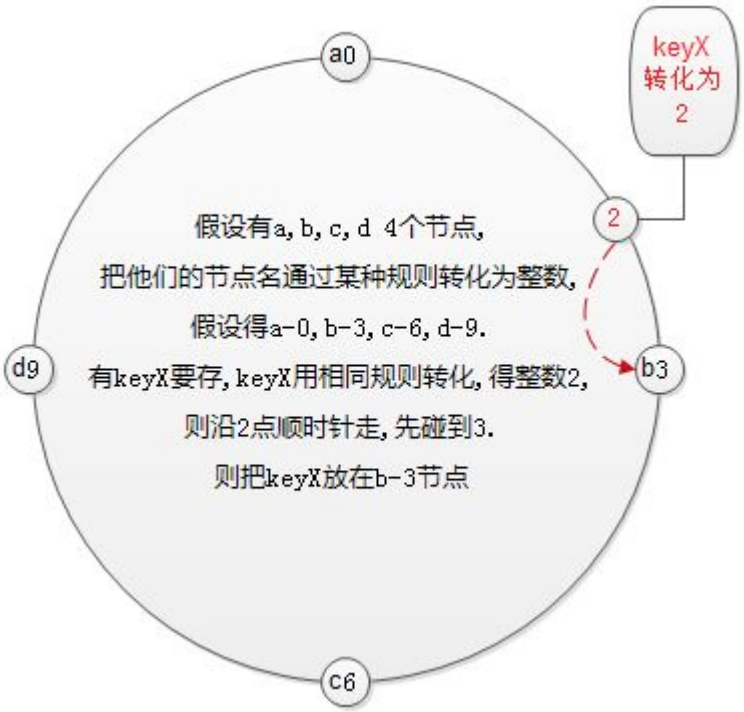


图 6.4: 一致性哈希查找节点

疑问 1: 时钟上的指针最大才 11 点,如果我有上百个 memcached 节点怎么办?

答: 时钟只是为了便于理解做的比喻,在实际应用中,我们可以在圆环上分布[0,2^32-1]的数字,这样,全世界的服务器都可以装下了.

疑问 2: 我该如何把"节点名","键名"转化成整数?

答: 你可以用现在的函数,如 crc32().

也可以自己去设计转化规则,但注意转化后的碰撞率要低.

即不同的节点名,转换为相同的整数的概率要低.

6.5　一致性哈希对其他节点的影响

通过图 5.5 可以看出,当某个节点 down 后,只影响该节点顺时针之后的 1 个节点,而其他节点不受影响.因此，Consistent Hashing 最大限度地抑制了键的重新分布(图 5.5).

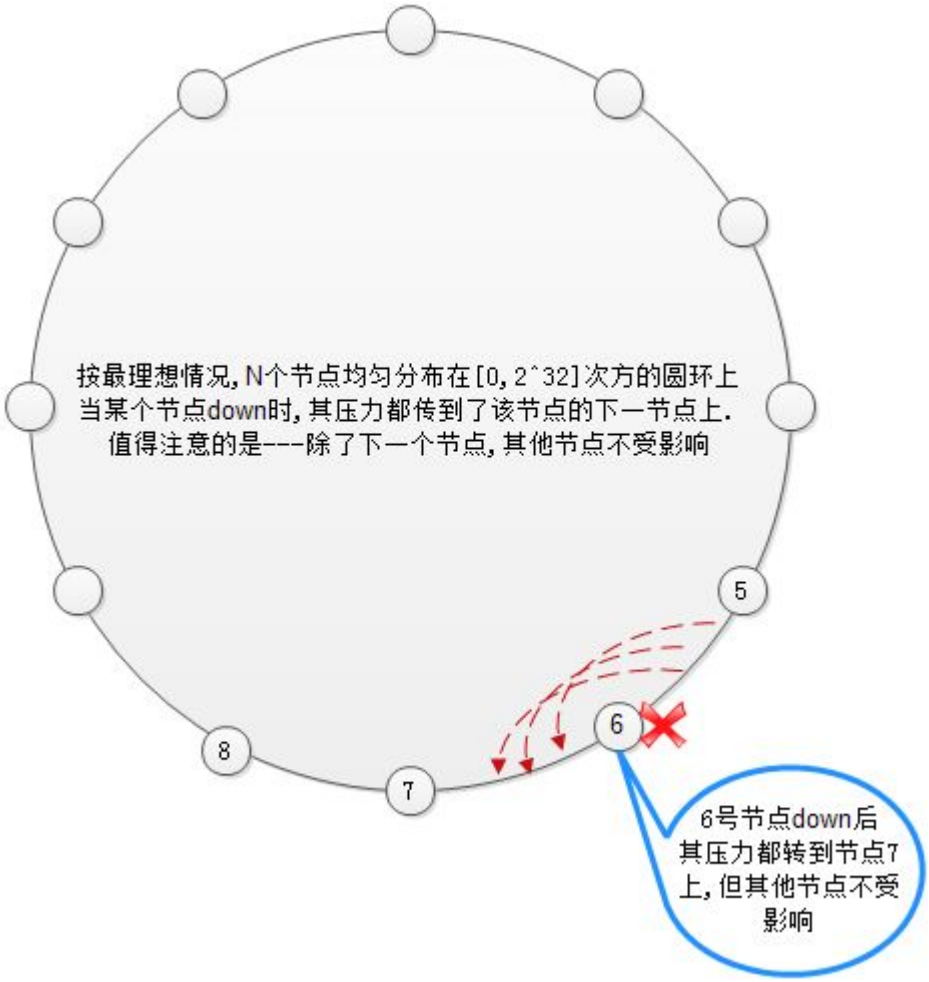


图 6.5: 一致性哈希中节点的影响范围

6.6 一致性哈希+虚拟节点对缓存命中率的影响

由图 5.5 中可以看到,理想状态下,
1) 节点在圆环上分配分配均匀,因此承担的任务也平均,但事实上, 一般的 Hash 函数对于节点在圆环上的映射,并不均匀.
2) 当某个节点 down 后,直接冲击下 1 个节点,对下 1 个节点冲击过大,能否把 down 节点上的压力平均的分担到所有节点上?

完全可以---引入虚拟节点来达到目标 (图 5.6)
虚拟节点即----N 个真实节点,把每个真实节点映射成 M 个虚拟节点, 再把 M*N 个虚拟节点,散列在圆环上. 各真实节点对应的虚拟节点相互交错分布
这样,某真实节点 down 后,则把其影响平均分担到其他所有节点上.

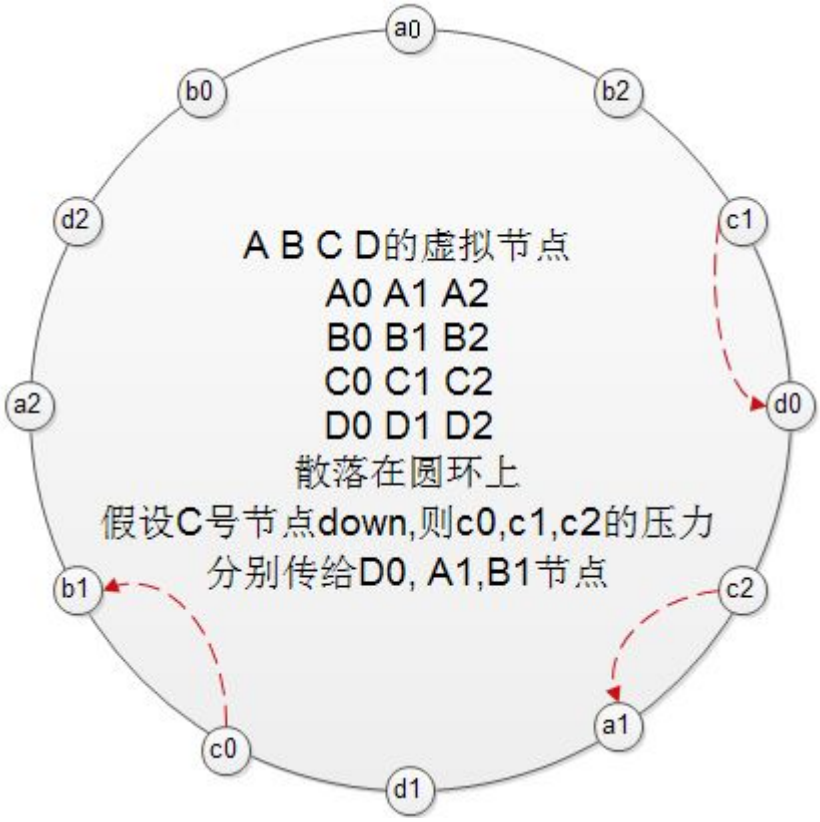


图 6.6: 引入虚拟节点

6.7　一致性哈希的 PHP 实现

```
/*
实现一致性哈希分布的核心功能.
*/
// 需要一个把字符串转成整数的接口
interface hasher {
    public function _hash($str);
}

interface distribution {
    public function lookup($key);
}

class Consistent implements hasher,distribution {
    protected $_nodes = array();
    protected $_postion = array();

    protected $_mul = 64; //每个节点对应 64 个虚节点


    public function _hash($str) {

        return sprintf('%u',crc32($str)); // 把字符串转成 32 位符号整数

    }

    // 核心功能
    public function lookup($key) {
        $point = $this->_hash($key);


        $node = current($this->_postion); //先取圆环上最小的一个节点,当
成结果
        foreach($this->_postion as $k=>$v) {
            if($point <= $k) {
                $node = $v;
                break;
            }
        }
        reset($this->_postion);
        return $node;
```

```
    }

    public function addNode($node) {
        if(isset($this->nodes[$node])) {
            return;
        }

        for($i=0; $i<$this->_mul; $i++) {
            $pos = $this->_hash($node . '-' . $i);
            $this->_postion[$pos] = $node;
            $this->_nodes[$node][] = $pos;
        }

        $this->_sortPos();
    }

    // 循环所有的虚节点,谁的值==指定的真实节点 ,就把他删掉
    public function delNode($node) {
        if(!isset($this->_nodes[$node])) {
            return;
        }

        foreach($this->_nodes[$node] as $k) {
               unset($this->_postion[$k]);
        }

        unset($this->_nodes[$node]);
    }

    protected function _sortPos() {
        ksort($this->_postion,SORT_REGULAR);
    }
}

// 测试
$con = new Consistent();
$con->addNode('a');
$con->addNode('b');
$con->addNode('c');
$key = 'www.zixue.it';

echo '此 key 落在',$con->lookup($key),'号节点';
```

第七章 一致性哈希算法实验课

7.1 实验目的

测试 memcached 缓存服务器有 N 台变为 N-1 台时候，取模分布式算法与一致性哈希对缓存命中率的不同影响

7.2 实验原理

相同硬件环境、操作系统、相同数据环境，建立 5 个 memcached 节点，令数据的有效期为永久有效，用取模分布式算法建立缓存，缓存稳定后，观察命中率，命中率稳定后，减少 1 个节点，观察命中率的变化，直到命中率再次稳定。

恢复实验环境，再次建立 5 个 memcached 节点，令数据的有效期为永久。用一致性哈希算法建立缓存，缓存稳定后，观察命中率，命中率稳定后，减少 1 个节点，观察命中率的变化，直到命中率再次稳定。

备注: 节点 down 之前,保持缓存永久有效,命中率为 100%的目的是防止缓存失效对命中率产生干扰. 因为我们的目的在于比较节点 down 之前和 down 之后的缓存命中率变化. 因此,屏蔽其他可能对命中率干扰的要素.

7.3 实验文件

文件名	作用
Config.php	配置 memcached 结点信息和哈希策略。
Hash.php	分布式实现类
Initdata.php	初始化各结点信息
Exec.php	减少结点后模拟请求
Load.php	统计平均命中率
Index.html	Ajax 请求 load.php,动态生成图表

7.4 试验步骤

7.4.1 设置分布式策略为一致性哈希

I. 打开 5 个 memcached 进程。
II. 运行 initdata.php 初始化数据。
III. 打开 index.html,此时命中率理论值为 100%。
IV. 运行 exec.php。
V. 观察命中率的变化。

7.4.2 设置分布式策略为取模算法

VI. 打开 10 个 memcached 进程。
VII. 运行 initdata.php 初始化数据。
VIII. 运行 index.html,此时命中率理论值为 100%。
IX. 运行 exec.php。
X. 观察命中率的变化。

7.5 试验数据及曲线

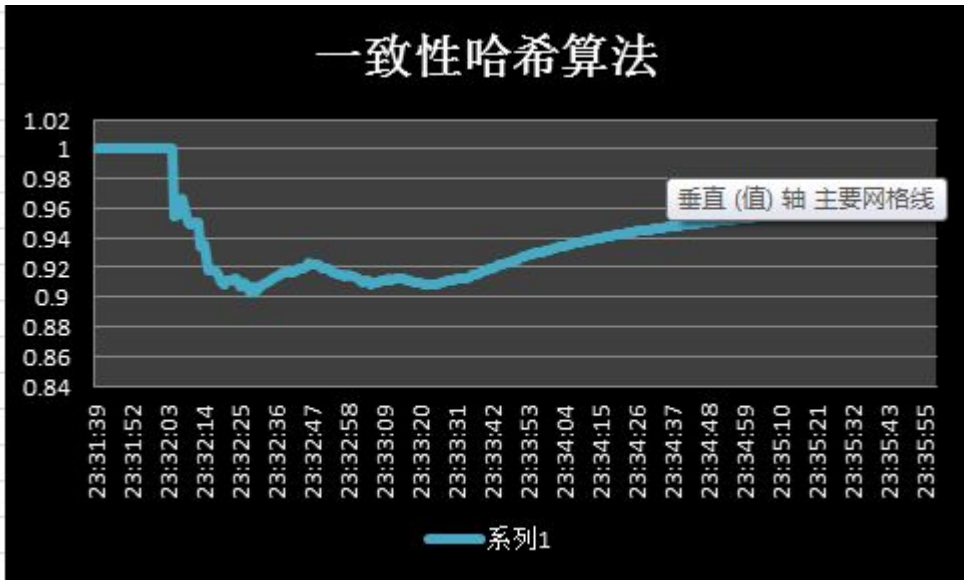


图 7.5: 一致性哈希减少节点后命中率的变化

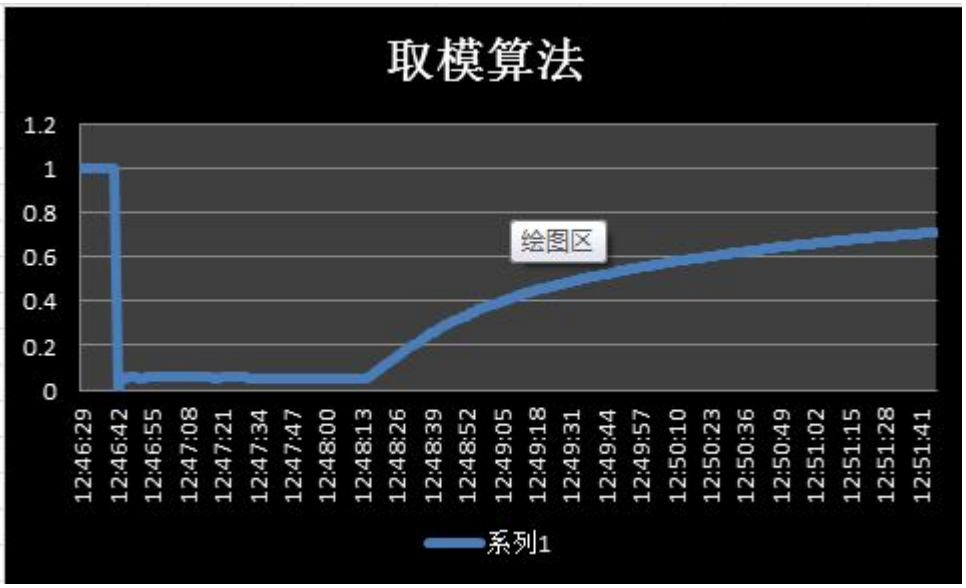


图 7.5: 取模算法实现分布式缓存

7.6 实验思考

1.从数学角度分析，当 memcahed 结点由 N->N-1 时，取模算法的命中率降为多少?一致性哈希的命中率降为多少?
2.当 memcached 节点越多时，一致性哈希算法对缓存的命中率比取模算法对缓存的命中率要高很多。
3.取模哈希算法，简单快速，缺点是在 memcached 节点增加或者删除的时候，原有的缓存数据将大规模失效，命中率大受影响，如果节点数多，缓存数据多，重建缓存的代价太高。一致性哈希算法， 最大限度地减小服务器增减时的缓存重新分布。

第八章　memcached 经典问题或现象

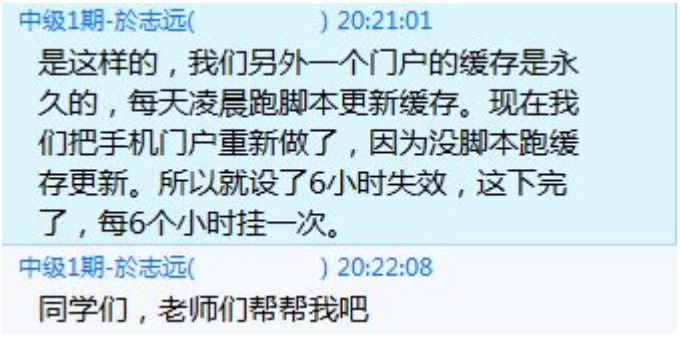
8.1 缓存雪崩现象及真实案例

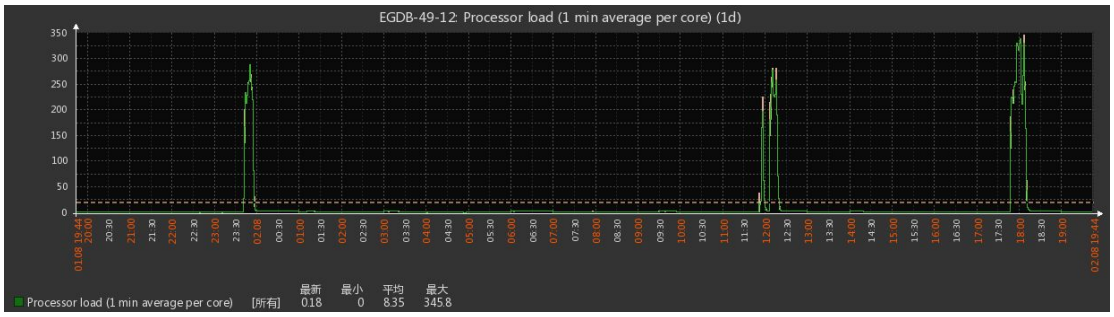
缓存雪崩一般是由某个缓存节点失效,导致其他节点的缓存命中率下降, 缓存中缺失的数据去数据库查询.短时间内,造成数据库服务器崩溃.

重启 DB,短期又被压跨,但缓存数据也多一些.
DB 反复多次启动多次,缓存重建完毕,DB 才稳定运行.

或者,是由于缓存周期性的失效,比如每 6 小时失效一次,那么每 6 小时,将有一个请求”峰值”,严重者甚至会令 DB 崩溃.

8.1.1 雪崩真实案例

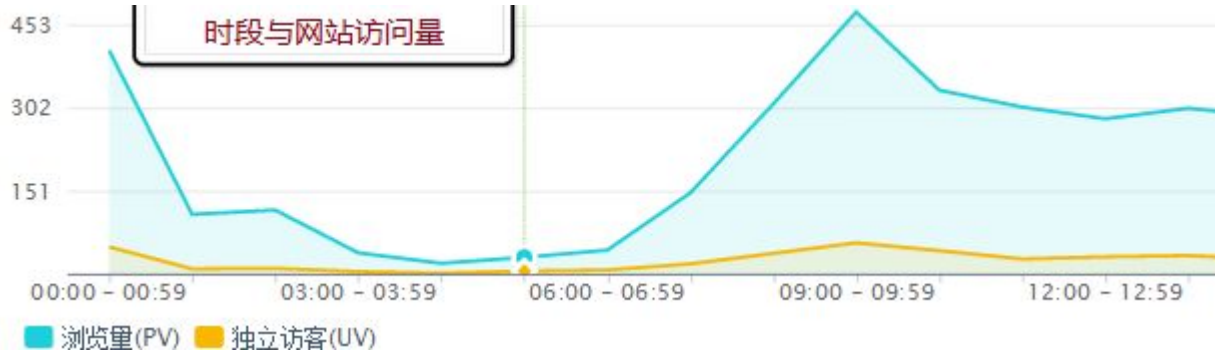




8.1.2 案例中的解决方案

- 该学员使用的解决方案

中级1期-於志远(　　) 21:26:30
我们已经把缓存时间调长了，每天夜里跑脚本刷新缓存。
燕十八(328268186) 21:27:27
基本解决?
中级1期-於志远(　　) 21:29:00
解决了。负载很稳定了现在。
中级1期-黄志仰(　　) 21:42:02
之前不是六个钟，由于过度集中导致崩了吗
燕十八(328268186) 21:42:34
半夜刷新,那时负载低,相当于缓慢重建缓存
中级1期-黄志仰(　　.) 21:42:43
哦
燕十八(328268186) 21:42:53
跑到早上8 9 点时,已经建立起来一部分了.



- 讨论的解决方案:
  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,把工作分担到各个时间点上去.

8.2 缓存的无底洞现象 multiget-hole

该问题由 facebook 的工作人员提出的, facebook 在 2010 年左右,memcached 节点就已经达 3000 个.缓存数千 G 内容.
他们发现了一个问题---memcached 连接频率,效率下降了,于是加 memcached 节点,
添加了后,发现因为连接频率导致的问题,仍然存在,并没有好转,称之为”无底洞现象”.
原文见:
http://highscalability.com/blog/2009/10/26/facebooks-memcached-multiget-hole-more-machines-more-capacit.html

8.2.1 multiget-hole 问题分析

以用户为例: user-133-age, user-133-name,user-133-height .....N 个 key,
当服务器增多,133 号用户的信息,也被散落在更多的节点,
所以,同样是访问个人主页,得到相同的个人信息, 节点越多,要连接的节点也越多.
对于 memcached 的连接数,并没有随着节点的增多,而降低. 于是问题出现.

8.2.2 multiget-hole 解决方案:

把某一组 key,按其共同前缀,来分布.
比如 user-133-age, user-133-name,user-133-height 这 3 个 key,
在用分布式算法求其节点时,应该以 'user-133'来计算,而不是以 user-133-age/name/height 来计算.
这样,3 个关于个人信息的 key,都落在同 1 个节点上,访问个人主页时,只需要连接 1 个节点.
问题解决.
官方回应:http://dormando.livejournal.com/521163.html

事实上: NoSQL 和传统的 RDBMS,并不是水火不容,两者在某些设计上,是可以相互参考的. 对于 memcached, redis 这种 kv 存储, key 的设计,可以参考 MySQL 中表/列的设计. 比如: user 表下,有 age 列,name 列,身高列, 对应的 key,可以用 user:133:age = 23, user:133:name = 'lisi', user:133:height = 168;

8.3 永久数据被踢现象

网上有人反馈为"memcached 数据丢失",明明设为永久有效,却莫名其妙的丢失了.

其实,这要从 2 个方面来找原因:
即前面介绍的 惰性删除,与 LRU 最近最少使用记录删除.
分析(图 7.3-):
1:如果 slab 里的很多 chunk,已经过期,但过期后没有被 get 过, 系统不知他们已经过期.
2:永久数据很久没 get 了,不活跃,如果新增 item,则永久数据被踢了.
3: 当然,如果那些非永久数据被 get,也会被标识为 expire,从而不会再踢掉永久数据

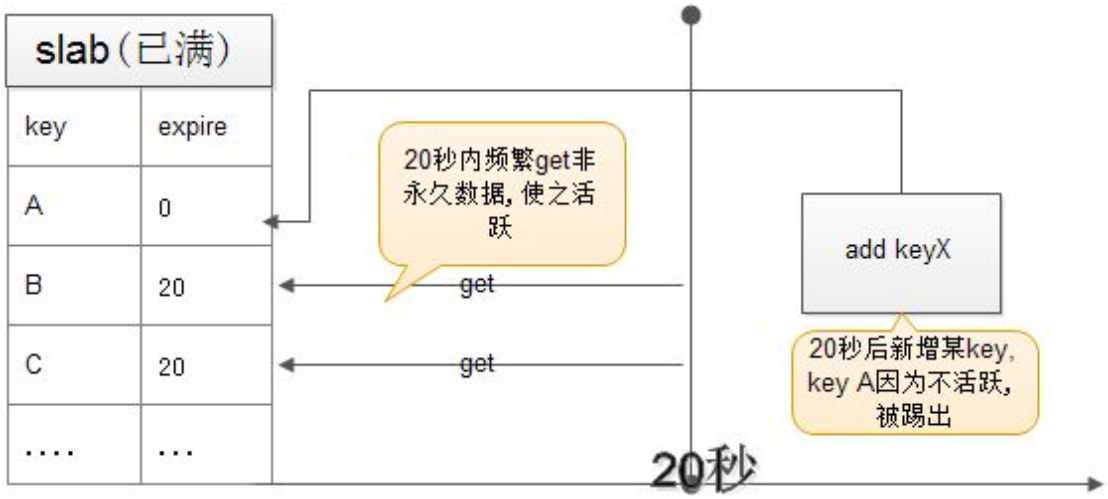


图 8.3: 永久数据被踢

解决方案: 永久数据和非永久数据分开放