第四章 Memcached经典问题或现象

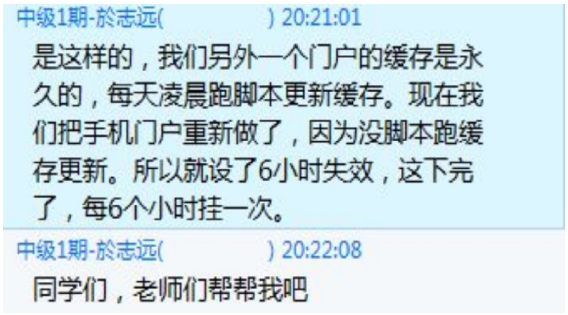
4.1 缓存雪崩现象及真实案列

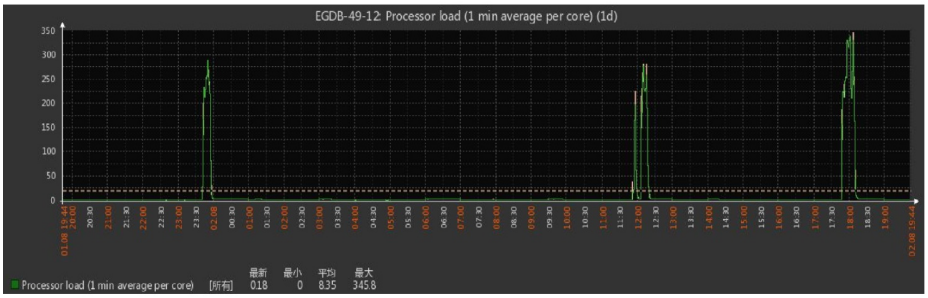
缓存雪崩一般是由某个节点缓存失效,导致其他节点的缓存命中率下降,缓存中缺失的数据去数据库查询,短时间内,造成数据库服务器崩溃.

重启DB,短期又被压垮,但缓存数据也多一些.DB反复多次重启多次,缓存重建完毕,DB才稳定运行.

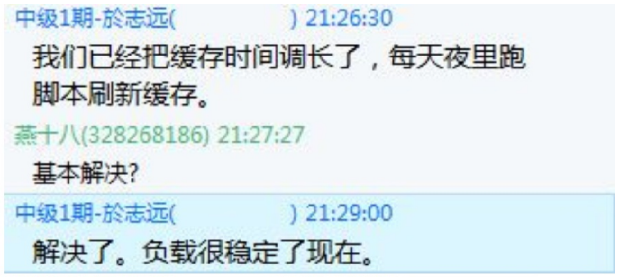
或者是由于缓存周期性的失效,比如6小时失效一次,那么每过6个小时,都会有一个请求的'峰值',严重者甚至会令DB崩溃.

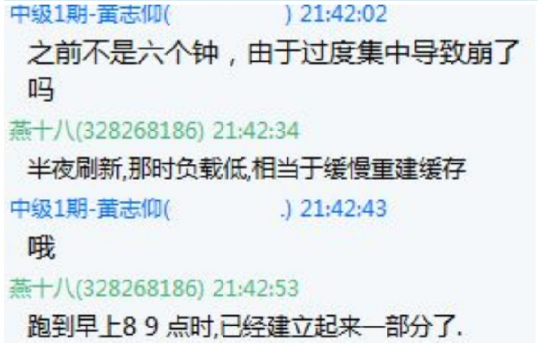
实际案列

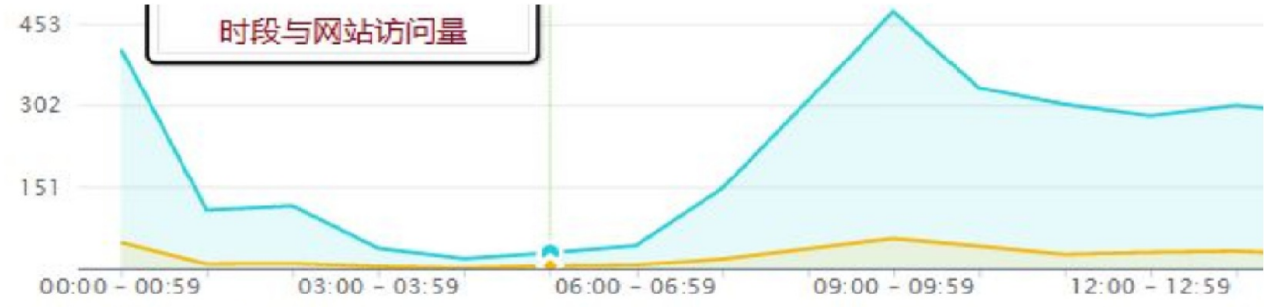




学员的解决方案







讨论的解决方案

把缓存的设置为在3到9小时之间随机失效,这样不同时失效可以把工作分担到各个时间点上去.

需要注意的是,在这个场景下,我们使用到的代码知识仅仅只有flush_all,这是服务器工程问题,并没有统一的答案.但是如何解决memcache服务器当机的情况呢?这就需要算法来帮助我们解决问题,即我们后面所要将的一致性哈希算法.

4.2 缓存的无底洞

该问题由facebook的工作人员提出的,facebook在2010年左右,memcached节点已经到达了3000多个,缓存数千G的内容.

他们发现了一个问题--memcached连接频率的增高,导致效率下降了,于是增加了更多的节点,但是添加了节点之后,发现因连接频率导致的问题仍然存在,并没有好转,他们将这种现象称为'无底洞现象';

原文 //highscalability.com/blog/2009/10/26/facebooks-memcached-multiget-hole-more-machines-more-capacit.html

4.2.1 multiget-hole问题分析

键值数据库或者缓存系统，由于通常采用hash函数将key映射到对应的实例，造成key的分布与业务无关，但是由于数据量、访问量的需求，需要使用分布式后（无论是客户端一致性哈性、redis-cluster、codis），批量操作比如批量获取多个key(例如redis的mget操作)，通常需要从不同实例获取key值，相比于单机批量操作只涉及到一次网络操作，分布式批量操作会涉及到多次网络io。

以学生信息为例:

- 'stu-li-age'22
- 'stu-li-height' 170
- 'stu-li-area' 'hebei'
- 'stu-li-score' 87

当我们的服务器增加之后,学生li的用户信息,也被散落在更多的节点,所以,同样得到相同的个人信息,节点越多,需要连接的节点也更多,对于,memcached的连接数,并没有随着节点的增多而降低.

4.2.2 无底洞现象带来的危害

(1) 客户端一次批量操作会涉及多次网络操作，也就意味着批量操作会随着实例的增多，耗时会不断增大。

(2) 服务端网络连接次数变多，对实例的性能也有一定影响。

4.2.3 解决方案

官方回应 //dormando.livejournal.com/521163.html

当然,我们希望将一个人的信息尽可能多的落在同一个服务器节点上,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.

我们在设计memcached,redis这种kv存储的NoSQL数据库的key时,可以参照MySQL中的表/列的设计

uid	name	age	height	area	score
1	lisi	22	170	hebei	87

设计memcached的键的格式为"表:主字段:主键值:字段名"的格式,例如

- `user:uid:1:name lisi`
- `user:uid:1:age 22`
- ....

当我们用哈希算法计算字段的分布规则的时候,可以只截取`key`到`user:uid:1`,这样每个用户的信息都会分别被存放在一起,提高了我们的连接效率;

第五章 Memcached的内存管理与删除机制

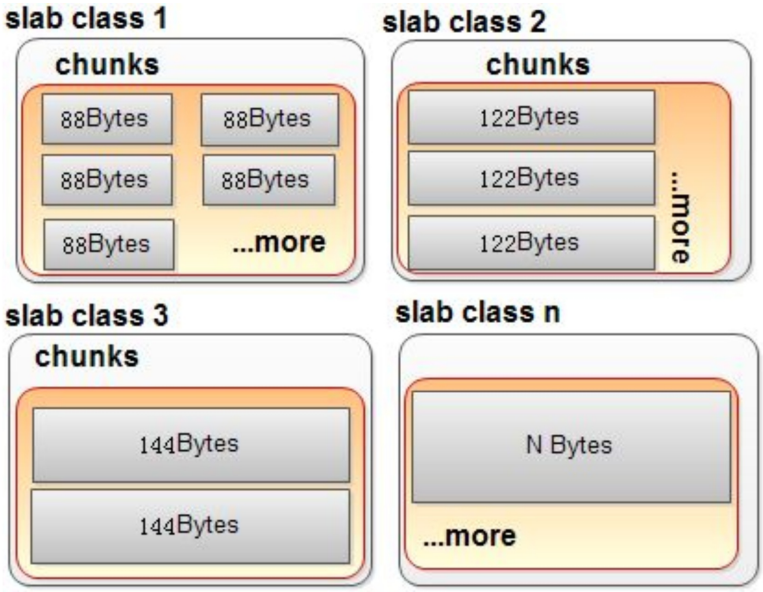
5.1 内存的碎片化

如果在c语言直接malloc,free来想操作系统申请和释放内存,在不断的申请和释放的过程当中,形成了一些很小的内存碎片,无法再被利用,这种空闲但是无法利用内存的现象,称之为内存的碎片化

5.2 slab allocator 缓解内存的碎片化

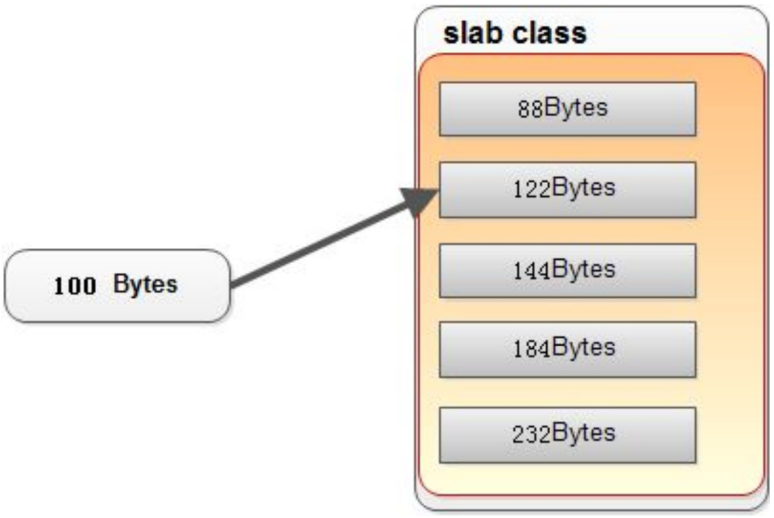
memcached使用 slab allocator 机制来管理内存.

slab allocator原理:预先把内存划分为数个slab class库(每个slab class 大小为1M),各个仓库被切分成不同尺寸的小块(chunk),需要存内容的时候,判断内容的大小,为其选取合理的仓库.



5.3 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),memcached中保留着slab class 内空闲的chunk列表,根据该列表选择空的chunk,然后将数据缓存其中.



> 注意:
> 如果有100byte的内容要存储,但是122大小的仓库中的chunk已经满了,并不会寻找更大的,如144的仓库来存储,而是把122仓库的数据踢掉!

5.4 固定大小的chunk带来的内存浪费

由于slab allocator机制中,分配chunk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22字节就浪费了.

对于chunk空间浪费的问题,我们无法彻底解决,只能缓解该问题.

开发者可以对网站中缓存的item长度进行统计,并制定合理的slab class中的chunk的大小,虽然我们目前并不能直接自定义chunk的大小,但是可以通过参数来调整slab class 中chunk的增长速度,即增长因子,grow factor!

5.5 grow factor 调优

memcached在启动的时候可以通过-f 选项指定增长因子,并在某种成都上控制slab之间的差异,默认值为1.25,但是,在这个选项出现之前,这个因子曾经固定为2,称为'power of 2'策略

我们分别用grow factor 为1.25和2看一看效果

```
[root@whitsats bin]# ./memcached -u white -p 11211 -m 1 -vvv -f 1.25
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   5: chunk size       240 perslab    4369
slab class   6: chunk size       304 perslab    3449
slab class   7: chunk size       384 perslab    2730
slab class   8: chunk size       480 perslab    2184
......
```

```
[root@whitsats bin]# ./memcached -u white -p 11211 -m 1 -vvv -f 2
slab class   1: chunk size        96 perslab   10922
slab class   2: chunk size       192 perslab    5461
slab class   3: chunk size       384 perslab    2730
slab class   4: chunk size       768 perslab    1365
slab class   5: chunk size      1536 perslab     682
slab class   6: chunk size      3072 perslab     341
slab class   7: chunk size      6144 perslab     170
slab class   8: chunk size     12288 perslab      85
slab class   9: chunk size     24576 perslab      42
slab class  10: chunk size     49152 perslab      21
slab class  11: chunk size     98304 perslab      10
slab class  12: chunk size    196608 perslab       5
slab class  13: chunk size    524288 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,我们应该细心统计缓存大小,指定合理的增长因子

> 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,这些计算误差是为了保持整数的对齐而故意设置的.

5.6 memcached的过期数据惰性删除

1. 当某个值过期之后,并没有从内存中删除,因此,stats统计时,curr_item有其信息
2. 当get时,判断获取到的值是否过期,如果过期,返回空并且清空该值,curr_item就减少了
3. 当某个新值去占用它的位置时,作为空的chunk来看待

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,该机制被称为 lazy expiration,惰性失效,这种失效机制能够极大的节

省cpu时间和检测的成本

5.7 memcached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
- rand

LRU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.
就把谁踢出出,有时候这个算法会将永久数据踢出

5.8 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.

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.

> 在1.5以上版本的memcache中，当chunk被占满时，我们一旦对该slab进行任何的操作，该slab即会自动清除已过期数据。

解决方案: 永久数据和非永久数据分开放

第六章 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(在不同的机器上或者在同一台机器上的不同端口)

第七章 分布式集群

7.1 分布式集群算法

7.1.1 memcached是如何实现分布式的

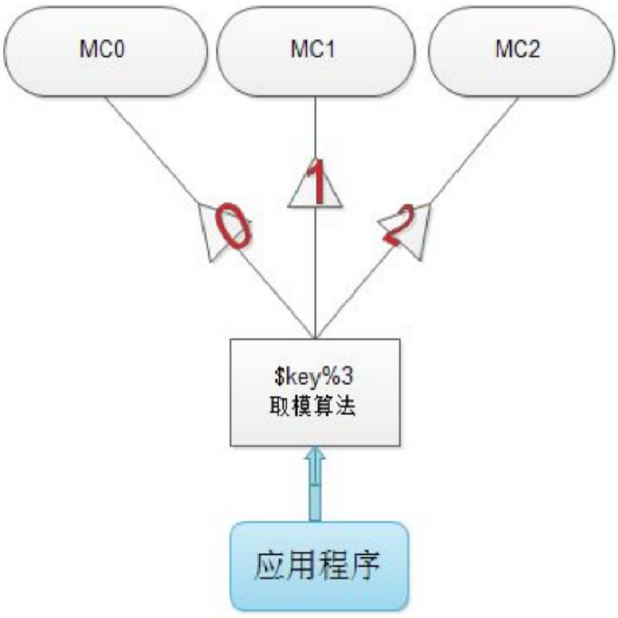
前面的课我们介绍memcached是一个分布式缓存,但是memcached并不像mongodb一样允许配置多个节点,并且节点之间自动分配数据.

也就是说 memcached节点之间,是不互相通信的.因此,memcached的分布式,是靠用户设计算法,把数据分布在多个memcached节点中.

分布式其实就是多台服务器分配工作量的方法.

7.1.2 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.



取模算法对命中率的影响

假设有8台服务器,突然down掉一台,则余数底数变成7,那么后果会变成什么样?

```
key0 % 8 = 0    key0 % 7 = 0   hits
......
key6 % 8 = 6    key6 % 7 = 6   hits
key7 % 8 = 7    key7 % 7 = 0   miss
key8 % 8 = 0    key8 % 7 = 1   miss
......
key56 % 8 = 0    key56 % 8 = 0   hits
```

一般的,我们由数学归纳法总结可得

如果有N台服务器,down掉了一台,变为N-1,则每N*(N-1)个数据中,只有N-1个数据能够得到相同的映射;

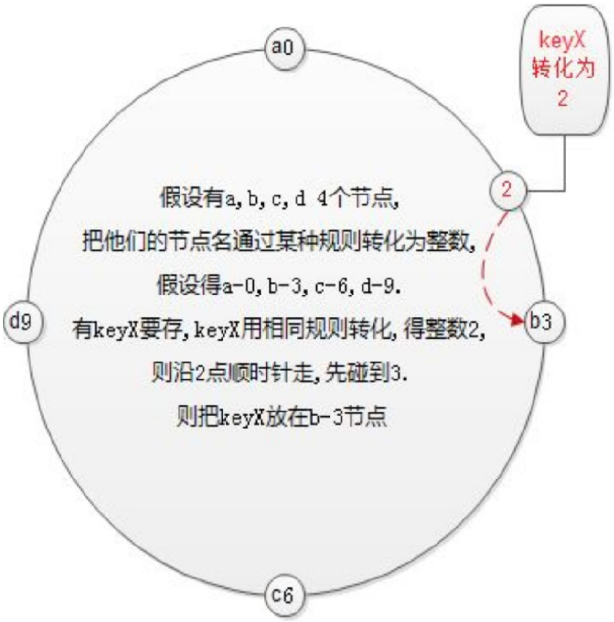
所以,命中率在服务器down掉的短期内,急剧下降到(N-1)/(N*(N-1)) = 1/N ,由此可知，采用取模算法分配数据的服务器越多,down机的后果越严重.

那么,有没有一种方法能够帮助我们提高在down机之后的命中率?因为照理而言,如果有一台服务器down掉了,我们的命中率最高可以到达(N-1)/N,服务器越多,影响越小

7.1.3 一致性哈希算法

在新浪的redis服务器中,就大量的使用了一致性哈希算法.

通俗的理解一致性哈希算法,我们可以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



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?

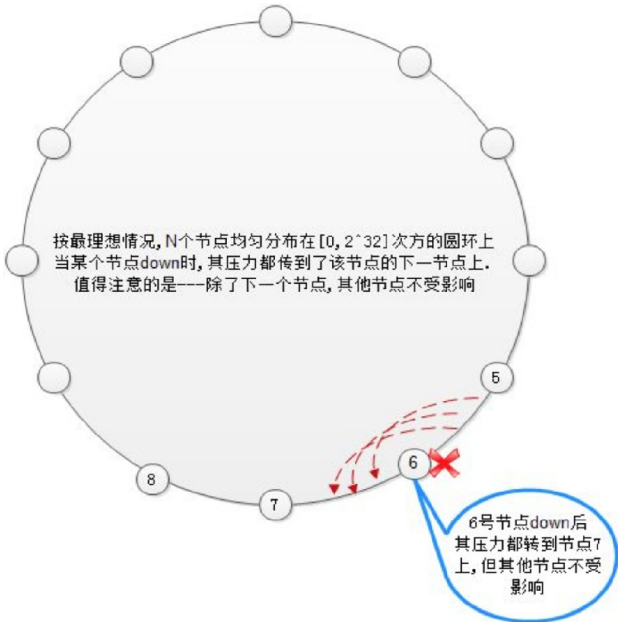
答: 时钟只是为了便于理解做的比喻,在实际应用中,我们可以在圆环上分布[0,2^32-1]的数字,
这样,全世界的服务器都可以装下了.

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.

7.1.3.1 一致性哈希对其他节点的影响

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.



如果6号节点down掉,那么原有的6号服务器上的数据miss,6号服务器的访问压力转移到到7号,而其他节点不收影响,理论命中率为(N-1)/N;

但是如果真的这样设计,那么7号节点也可能短时间down掉,从后导致以后的服务器压力越来越大,逐一崩溃,有没有一种方式来帮助我们的其他均摊6号节点down机之后的压力呢?

7.1.3.2 虚拟节点

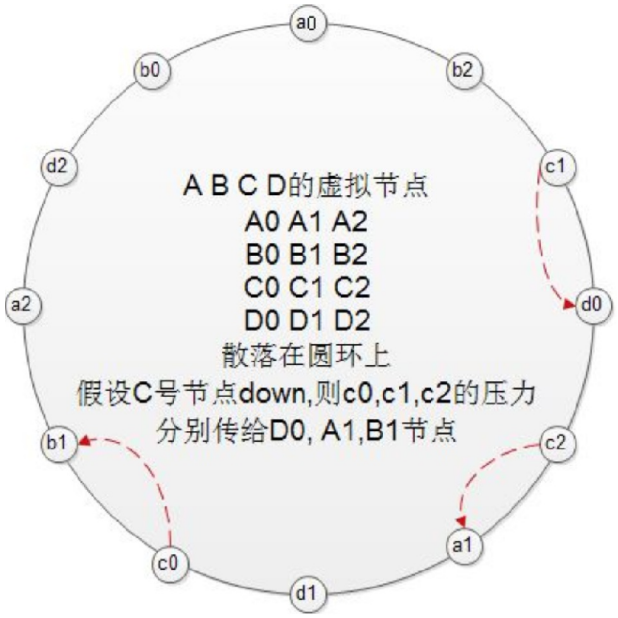
在我们认定的理想态下存在如下几个问题:

1. 节点在圆环上分配不均匀,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,每个节点承担的压力也不同
2. 当某个节点down后,直接冲击下一个节点,对下一个节点的冲击过大,能否把down节点上的压力平均分担到所有节点上?

因此我们可以引入虚拟节点来达到目标

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,各个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,则把其影响均匀分摊到其他的所有节点之上.



可能上图还是有些理想,因为crc32算出的数值并不一定能够均匀分布,但是我们可以增多虚拟节点,让节点的分布尽量均匀.

7.1.3.3 利用PHP实现一次性hash算法

```
$m = new Memcached();
$m->setOptions(array(
    Memcached::OPT_DISTRIBUTION=>Memcached::DISTRIBUTION_CONSISTENT,
    Memcached::OPT_LIBKETAMA_COMPATIBLE=>true,
    Memcached::OPT_REMOVE_FAILED_SERVERS=>true,
));

$m->addServers(array(
    array('localhost', 11211),
    array('localhost', 11212),
    array('localhost', 11213),
    //array('localhost', 11214),
));

$m->set('key1', 1);
$m->set('key2', 'abc');
$m->set('key3', array('foo', 'bar'));
$m->set('key4', new stdClass);
$m->set('key5', 'pigfly');
$m->set('key6', 999);
var_dump($m->get('key1'), $m->get('key2'), $m->get('key3'), $m->get('key4'), $m->get('key5'), $m->get('key6'));
```