

软件体系结构课程报告

题目 Memcached框架系统架构及战术分析

姓名与学号 许是程 3100000210

授课老师 李石坚

年级与专业 2010级 计算机科学与技术

Memcached框架系统架构及战术分析

# 一、简介

Memcached是一个高性能、分布式内存对象缓存系统，主要用于通过减少数据库调用来加速动态网页应用。由LiveJournal的Brad Fitzpatrick开发，但目前被许多网站使用，如Digg, Facebook, Wikipedia等。

Memcached允许你把系统中的冗余内存分配给内存紧缺的部分使用，使你能更好地使用内存。考虑一个分布式服务器，如果没有Memcached，则每个节点只能使用其自身的内存，而Memcached则允许一个节点使用其他节点的内存。

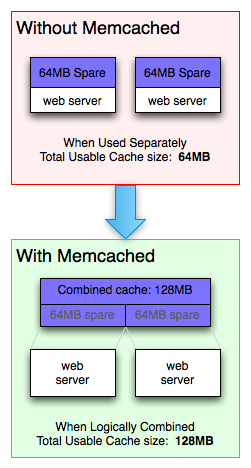


图1：两种分布式服务器内存管理模型

图1上半部分显示的是经典情形下的服务器端的内存分配，从中可以看出，大量内存被浪费了。一方面是整个集群所拥有的内存其实只有每个服务器的内存，另一方面是节点和节点间的数据需要保持同步，因此也耗费了大量资源在维护节点间的数据一致性。

图1下半部分则是使用了Memcached后的内存分配，此时每个节点的内存被统一抽象成了一个内存池，数据请求只需要访问整个集群中的一个地方，减少了节点间的数据传输。同时，当你想要扩大内存时，并不需要扩大每个节点的内存，而只需要增加服务器节点就可以。

Memcached缺乏认证以及安全管制，这代表应该将memcached服务器放置在防火墙后。Memcached的API使用三十二比特的循环冗余校验（CRC-32）计算键值后，将数据分散在不同的机器上。当表格满了以后，接下来新增的数据会以LRU机制替换掉。由于memcached通常只是当作高速缓存系统使用，所以使用memcached的应用程序在写回较慢的系统时（像是后端的数据库）需要额外的代码更新memcached内的数据。

# 二、配置过程

* 实验平台：Fedora 19, x86\_64
* 安装Memcached：sudo yum install memcached
* 配置Memcached：sudo vi /etc/sysconfig/memcached

# 主要需要根据本机内存大小调整CACHESIZE，其单位是MB

PORT="11211"

USER="memcached"

MAXCONN="1024"

CACHESIZE="512"

OPTIONS=""

* 运行Memcached：sudo service memcached start
* 如果需要Memcached在启动时自动运行：sudo chkconfig memcached on
* Python-memcached是一个Memcached的纯Python接口，由于最近的一个python项目正好间接使用到了他，所以我们以此为例来观察其运行情况。
* 安装python-memcached：sudo easy\_install python-memcached
* 查看运行时配置：echo "stats settings" | nc localhost 11211

STAT maxbytes 536870912

STAT maxconns 1024

STAT tcpport 11211

STAT udpport 11211

STAT inter NULL

STAT verbosity 0

STAT oldest 0

STAT evictions on

STAT domain\_socket NULL

STAT umask 700

STAT growth\_factor 1.25

STAT chunk\_size 48

STAT num\_threads 4

STAT num\_threads\_per\_udp 4

STAT stat\_key\_prefix :

STAT detail\_enabled no

STAT reqs\_per\_event 20

STAT cas\_enabled yes

STAT tcp\_backlog 1024

STAT binding\_protocol auto-negotiate

STAT auth\_enabled\_sasl no

STAT item\_size\_max 1048576

STAT maxconns\_fast no

STAT hashpower\_init 0

STAT slab\_reassign no

STAT slab\_automove 0

STAT tail\_repair\_time 3600

STAT flush\_enabled yes

END

# 三、运行过程

## 3.1 Memcached常用方法

memcache其实是一个map结构，最常用的几个函数：

* 保存数据
* set(key,value,timeout) 把key映射到value，timeout指的是什么时候这个映射失效
* add(key,value,timeout) 仅当存储空间中不存在键相同的数据时才保存
* replace(key,value,timeout) 仅当存储空间中存在键相同的数据时才保存
* 获取数据
* get(key) 返回key所指向的value
* get\_multi(key1,key2,key3,key4) 可以非同步地同时取得多个键值， 比循环调用get快数十倍
* 删除数据
* delete(key, timeout) 删除键为key的数据，timeout为时间值，禁止在timeout时间内名为key的键保存新数据（set函数无效）。

## 3.2 使用Python调用Memcached方法

1. 以下是样例运行代码：

import memcache

mc=memcache.Client(['127.0.0.1:11211'], debug=0)

mc.set("some\_key", "Some value")

value=mc.get("some\_key")

print "Some value = {0}".format(value)

mc.set("another\_key", 3)

mc.delete("another\_key")

mc.set("key", "1")

mc.incr("key")

mc.decr("key")

value=mc.get("key")

print "key = {0}".format(value)

运行结果截图如下：

图2：memcached测试程序截图

## 运行Python-memcached的Benchmark程序

以下是由Amir Salihefendic所编写的benchmark程序

#!/usr/bin/env python

import time

import random

import string

import sys

global total\_time

def run\_test(func, name):

sys.stdout.write(name + ': ')

sys.stdout.flush()

start\_time = time.time()

try:

func()

except:

print "failed or not supported"

global options

if options.verbose:

import traceback; traceback.print\_exc()

else:

end\_time = time.time()

global total\_time

total\_time += end\_time - start\_time

print "%f seconds" % (end\_time - start\_time)

class BigObject(object):

def \_\_init\_\_(self, letter='1', size=10000):

self.object = letter \* size

def \_\_eq\_\_(self, other):

return self.object == other.object

class Benchmark(object):

def \_\_init\_\_(self, module, options):

self.module = module

self.options = options

self.init\_server()

self.test\_set()

self.test\_set\_get()

self.test\_random\_get()

self.test\_set\_same()

self.test\_set\_big\_object()

self.test\_set\_get\_big\_object()

self.test\_set\_big\_string()

self.test\_set\_get\_big\_string()

self.test\_get()

self.test\_get\_big\_object()

self.test\_get\_multi()

self.test\_p\_app\_get()

#self.test\_get\_list()

def init\_server(self):

#self.mc = self.module.Client([self.options.server\_address])

self.mc = self.module.Client(["127.0.0.1:11211"])

self.mc.set('bench\_key', "E" \* 50)

num\_tests = self.options.num\_tests

self.keys = ['key%d' % i for i in xrange(num\_tests)]

self.values = ['value%d' % i for i in xrange(num\_tests)]

import random

self.random\_keys = ['key%d' % random.randint(0, num\_tests) for i in xrange(num\_tests \* 3)]

def test\_set(self):

set\_ = self.mc.set

pairs = zip(self.keys, self.values)

def test():

for key, value in pairs:

set\_(key, value)

def test\_loop():

for i in range(10):

for key, value in pairs:

set\_(key, value)

run\_test(test, 'test\_set')

for key, value in pairs:

self.mc.delete(key)

def test\_set\_get(self):

set\_ = self.mc.set

get\_ = self.mc.get

pairs = zip(self.keys, self.values)

def test():

for key, value in pairs:

set\_(key, value)

result = get\_(key)

assert result == value

run\_test(test, 'test\_set\_get')

#for key, value in pairs:

# self.mc.delete(key)

def test\_random\_get(self):

get\_ = self.mc.get

set\_ = self.mc.set

value = "chenyin"

def test():

index = 0

for key in self.random\_keys:

result = get\_(key)

index += 1

if(index % 5 == 0):

set\_(key, value)

run\_test(test, 'test\_random\_get')

def test\_set\_same(self):

set\_ = self.mc.set

def test():

for i in xrange(self.options.num\_tests):

set\_('key', 'value')

def test\_loop():

for i in range(10):

for i in xrange(self.options.num\_tests):

set\_('key', 'value')

run\_test(test, 'test\_set\_same')

self.mc.delete('key')

def test\_set\_big\_object(self):

set\_ = self.mc.set

# libmemcached is slow to store large object, so limit the

# number of objects here to make tests not stall.

pairs = [('key%d' % i, BigObject()) for i in xrange(100)]

def test():

for key, value in pairs:

set\_(key, value)

run\_test(test, 'test\_set\_big\_object (100 objects)')

for key, value in pairs:

self.mc.delete(key)

def test\_set\_get\_big\_object(self):

set\_ = self.mc.set

get\_ = self.mc.get

# libmemcached is slow to store large object, so limit the

# number of objects here to make tests not stall.

pairs = [('key%d' % i, BigObject()) for i in xrange(100)]

def test():

for key, value in pairs:

set\_(key, value)

result = get\_(key)

assert result == value

run\_test(test, 'test\_set\_get\_big\_object (100 objects)')

#for key, value in pairs:

# self.mc.delete(key)

def test\_set\_get\_big\_string(self):

set\_ = self.mc.set

get\_ = self.mc.get

# libmemcached is slow to store large object, so limit the

# number of objects here to make tests not stall.

pairs = [('key%d' % i, 'x' \* 10000) for i in xrange(100)]

def test():

for key, value in pairs:

set\_(key, value)

result = get\_(key)

assert result == value

run\_test(test, 'test\_set\_get\_big\_string (100 objects)')

def test\_set\_big\_string(self):

set\_ = self.mc.set

# libmemcached is slow to store large object, so limit the

# number of objects here to make tests not stall.

pairs = [('key%d' % i, 'x' \* 10000) for i in xrange(100)]

def test():

for key, value in pairs:

set\_(key, value)

run\_test(test, 'test\_set\_big\_string (100 objects)')

for key, value in pairs:

self.mc.delete(key)

def test\_get(self):

pairs = zip(self.keys, self.values)

for key, value in pairs:

self.mc.set(key, value)

get = self.mc.get

def test():

for key, value in pairs:

result = get(key)

assert result == value

run\_test(test, 'test\_get')

for key, value in pairs:

self.mc.delete(key)

def test\_get\_big\_object(self):

pairs = [('bkey%d' % i, BigObject('x')) for i in xrange(100)]

for key, value in pairs:

self.mc.set(key, value)

get = self.mc.get

expected\_values = [BigObject('x') for i in xrange(100)]

def test():

for i in xrange(100):

result = get('bkey%d' % i)

assert result == expected\_values[i]

run\_test(test, 'test\_get\_big\_object (100 objects)')

for key, value in pairs:

self.mc.delete(key)

def test\_p\_app\_get(self):

keys = ['Users/getUserById/1/False', '1\_db:users\_map', '2\_db:table\_group', '2\_db:shard\_infoN3']

def test():

for key in keys:

self.mc.get(key)

run\_test(test, 'test\_p\_app\_get')

def test\_get\_multi(self):

pairs = zip(self.keys, self.values)

for key, value in pairs:

self.mc.set(key, value)

keys = self.keys

expected\_result = dict(pairs)

def test():

result = self.mc.get\_multi(keys)

assert result == expected\_result

run\_test(test, 'test\_get\_multi')

for key, value in pairs:

self.mc.delete(key)

def test\_get\_list(self):

pairs = zip(self.keys, self.values)

for key, value in pairs:

self.mc.set(key, value)

keys = self.keys

expected\_result = self.values

def test():

result = self.mc.get\_list(keys)

assert result == expected\_result

run\_test(test, 'test\_get\_list')

for key in self.keys:

self.mc.delete(key)

def main(module\_name, module):

from optparse import OptionParser

parser = OptionParser()

parser.add\_option('-a', '--server-address', dest='server\_address',

default='127.0.0.1:11211',

help="address:port of memcached [default: 127.0.0.1:11211]")

parser.add\_option('-n', '--num-tests', dest='num\_tests', type='int',

default=1000,

help="repeat counts of each test [default: 1000]")

parser.add\_option('-v', '--verbose', dest='verbose',

action='store\_true', default=False,

help="show traceback infomation if a test fails")

global options

options, args = parser.parse\_args()

global total\_time

total\_time = 0

print "Benchmarking %s..." % module\_name

Benchmark(module, options)

print "Total\_time is %f" % total\_time

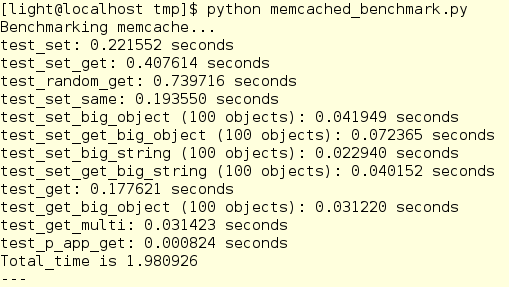
print '---'

if \_\_name\_\_ == '\_\_main\_\_':

import memcache

main('memcache', memcache)

该程序测试了memcached对不同大小的object在get和set操作上的速度，每个操作都执行了1000次，最终结果取这1000次的平均值。

图3：memcached的benchmark程序截图

# 三、优缺点

## 3.1 优点

* 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。
* 基于libevent的事件处理：libevent是个程序库，他将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。
* 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。
* Memcached删除数据时数据不会真正从memcached中消失。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 透明），其存储空间即可重复使用。
* Lazy Expriationmemcached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。这种技术称为lazy expiration.因此memcached不会再过期监视上耗费CPU时间。

## 3.2 缺点

1. 数据是保存在内存当中的，一旦服务进程重启，数据会全部丢失。
2. Memcached以root权限运行，而且Memcached本身没有任何权限管理和认证功能，安全性不足。

## 3.3 与Redis比较

* Redis不仅仅支持简单的k/v类型的数据，同时还提供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，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用。在Redis中，并不是所有的数据都一直存储在内存中的。
* Redis只会缓存所有的 key的信息，如果Redis发现内存的使用量超过了某一个阀值，将触发swap的操作，Redis根据“swappability = age\*log(size\_in\_memory)”计算出哪些key对应的value需要swap到磁盘。然后再将这些key对应的value持久化到磁盘中，同时在内存中清除。这种特性使得Redis可以 保持超过其机器本身内存大小的数据。当然，机器本身的内存必须要能够保持所有的key，毕竟这些数据是不会进行swap操作的。同时由于Redis将内存 中的数据swap到磁盘中的时候，提供服务的主线程和进行swap操作的子线程会共享这部分内存，所以如果更新需要swap的数据，Redis将阻塞这个 操作，直到子线程完成swap操作后才可以进行修改。
* 两者都经过了良好的设计，在０～300个client的并发GET/SET下，throughput 都在保持在10万/秒以上。
* memcached的性能比redis要好很多(数倍)，这也比较容易理解。但往往瓶颈会在client或者网络等地方。

# 四、战术思路

针对数据保存在内存中的问题，一方面通过增加服务器数量，另一方面可以采取更改Memcached的源代码，增加定期写入硬盘的功能。

针对安全性问题，可以将Memcached服务绑定在内网IP上，通过防火墙进行防护。

针对读取问题，可以在memcache前再加一个Oscache等本地缓存，减少对memcache的读操作，从而减小网络开销，提高性能。

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。

Facebook公司也大量使用了Memcached进行缓存，它对Memcached也有一些改进，可以参考：

* Apache 进程连接开销问题：实现了一个针对 TCP/UDP 的共享的进程连接缓冲池。共享的方式比针对单连接独占内存的方式节省不少内存资源。
* 并行请求和批处理我俄尼：构建web应用代码，目的是最小化对于页面请求回应所必要的网络往返数。Facebook构建了有向无环图（DAG）用来表示数据间的依赖。web服务器使用DAG来最大化可以并发读取的项目数。平均来说，这些批量请求对于每个请求包含24个主键。  
    
  客户端-服务器通信：memcached服务器不会直接通信。如果适当，会将系统的复杂度嵌入无状态的客户端，而不是memcached服务器。这极大地简化了memcached，使程序员专注于针对更有限的用例提供高性能。保持客户端的无状态使得快速迭代开发成为可能，同时也简化了部署流程。客户端的逻辑可以提供为两种组件：可以嵌入应用的一个库，或者做为一个名为mcrouter的独立的代理程序。这个代理提供memcached服务器的借口，对不同服务器之间的请求/回复进行路由。

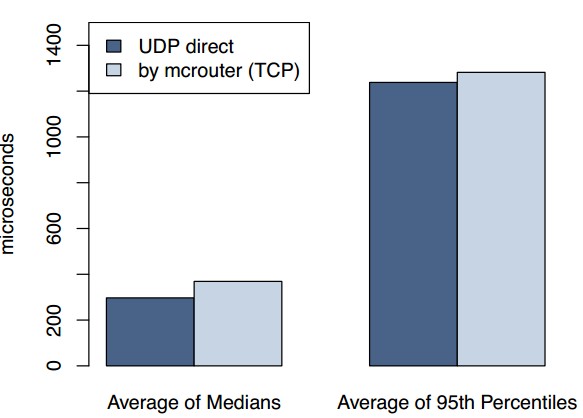


图 4: 经过mcrouter以后 UDP, TCP得到的延迟

# 五、与HDFS的区别

Memcached和HDFS虽然都可以进行数据存储，但是两者有本质的差别。Memcached主要是对数据库进行缓存，主要考虑数据存储的灵活性和读取速度；HDFS主要考虑的是数据的吞吐量，以支持Hadoop的大数据量处理。

# 参考资料

1. About Memcached, Official website of memcached, <http://memcached.org/about>
2. memcached, 维基百科，自由的百科全书, <http://zh.wikipedia.org/wiki/Memcached>
3. Install Memcached on Fedora 20 19 CentOS Red Hat (RHEL) 6.5 5.10, If Not True Then False, <http://www.if-not-true-then-false.com/2010/install-memcached-on-centos-fedora-red-hat/>
4. Memcached 安装 使用(Python操作), 落, <http://liluo.org/blog/2011/03/memcached-install-and-using/>
5. 使用memcache的几个优点, 某人的栖息地, <http://www.ooso.net/archives/306>
6. redis相对于memcached的一些优点, guoyang1987的ChinaUnix博客, <http://blog.chinaunix.net/uid-20683856-id-3359973.html>
7. Redis 和 Memcached 各有什么优缺点，主要的应用场景是什么样的？, 知乎, <http://www.zhihu.com/question/19829601>
8. Facebook 对 Memcache 伸缩性的增强, 技术翻译 – 开源中国社区, <http://www.oschina.net/translate/scaling-memcache-facebook>