



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。

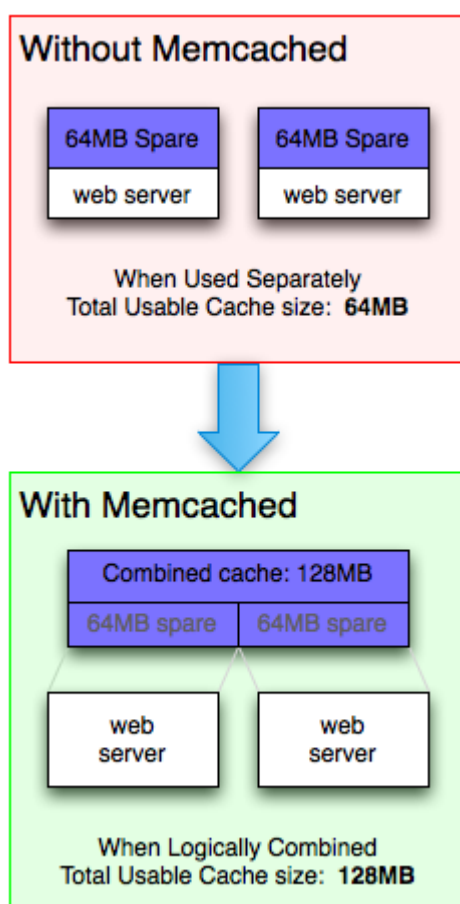


图 1：两种分布式服务器内存管理模型

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

以下是样例运行代码：

```
import memcache

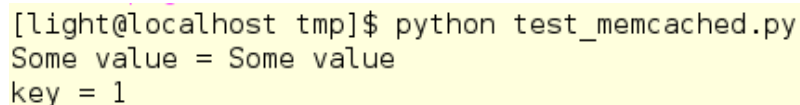
mc=memcache.Client(['127.0.0.1:11211'], debug=0)

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：



```
[light@localhost tmp]$ python test_memcached.py
Some value = Some value
key = 1
```

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 information 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。

```
[light@localhost tmp]$ python memcached_benchmark.py
Benchmarking memcache...
test_set: 0.221552 seconds
test_set_get: 0.407614 seconds
test_random_get: 0.739716 seconds
test_set_same: 0.193550 seconds
test_set_big_object (100 objects): 0.041949 seconds
test_set_get_big_object (100 objects): 0.072365 seconds
test_set_big_string (100 objects): 0.022940 seconds
test_set_get_big_string (100 objects): 0.040152 seconds
test_get: 0.177621 seconds
test_get_big_object (100 objects): 0.031220 seconds
test_get_multi: 0.031423 seconds
test_p_app_get: 0.000824 seconds
Total_time is 1.980926
---
```

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 Expiration 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阈值，将触发 swap 的操作，Redis 根据“ $\text{swappiness} = \text{age} * \log(\text{size_in_memory})$ ”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。

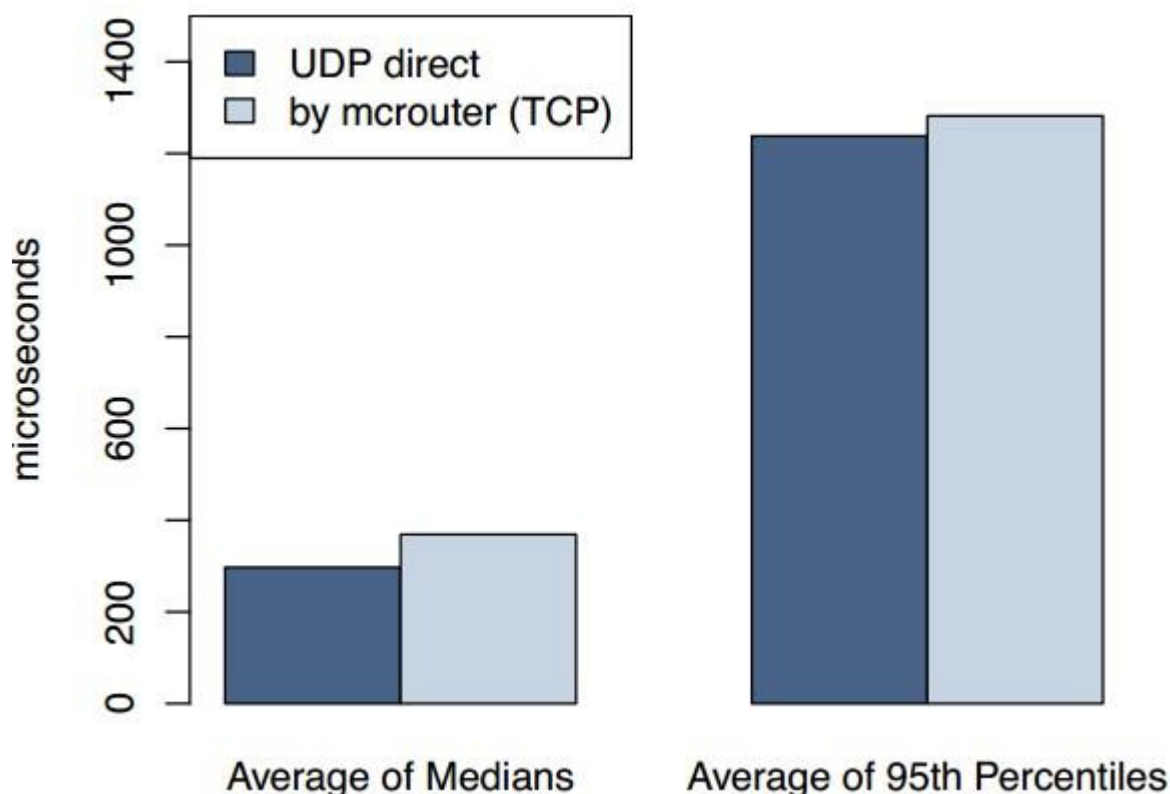


图 4: 经过 mcrouter 以后 UDP, TCP 得到的延迟

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>