目录

什么是 memcached？

官网：http://memcached.org/

What is Memcached?

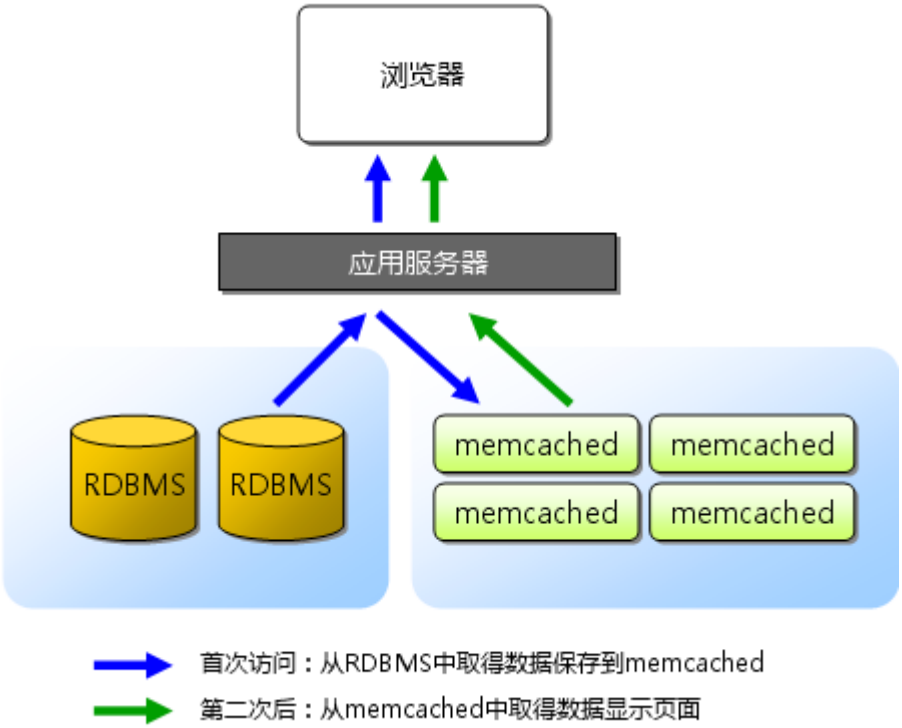
Free & open source, high-performance, distributed memory object caching system, generic in nature, but intended for use in speeding up dynamic web applications by alleviating database load.

Memcached is an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering.

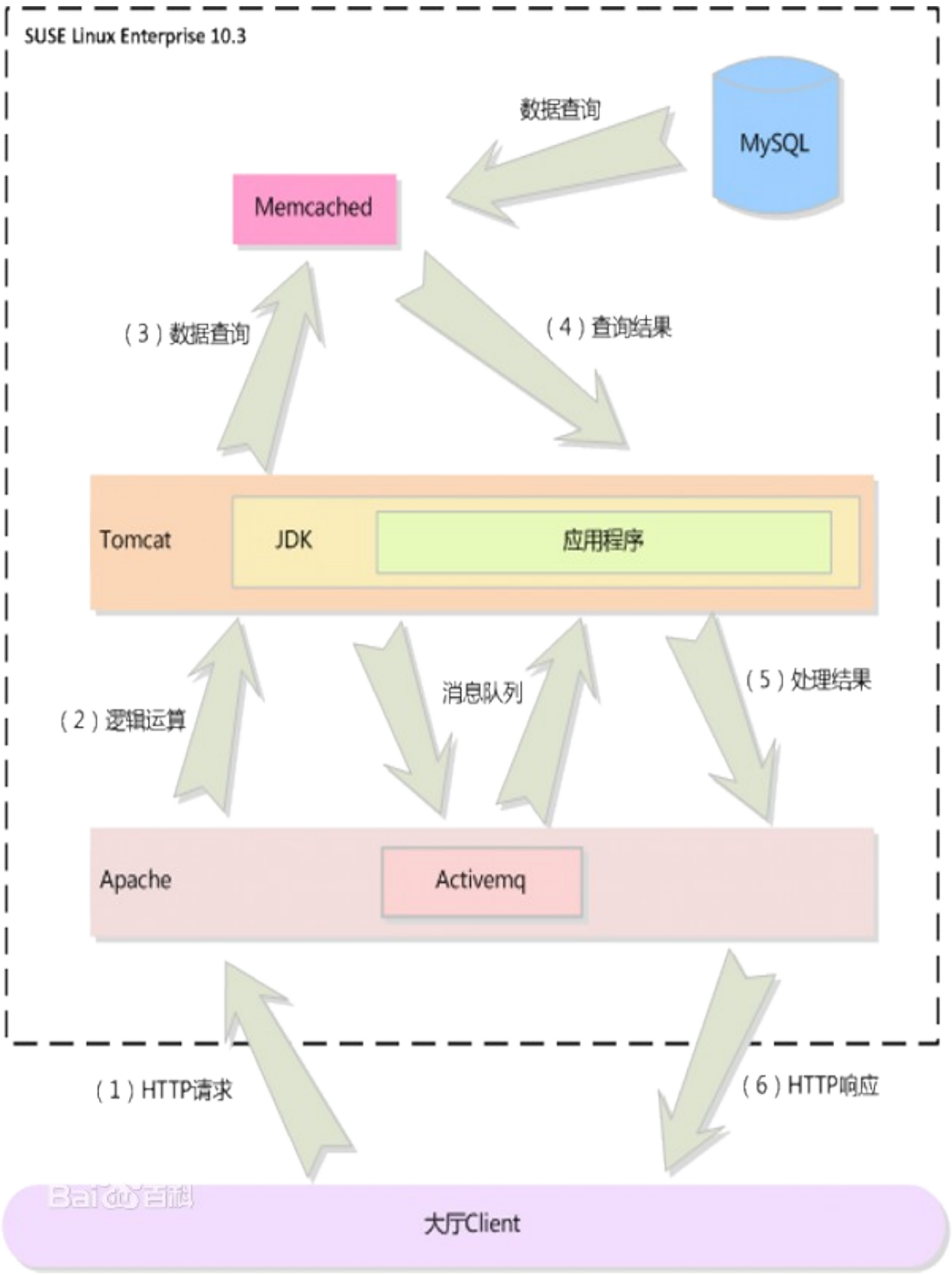
Memcached is simple yet powerful. Its simple design promotes quick deployment, ease of development, and solves many problems facing large data caches. Its API is available for most popular languages.

Memcached 是一个高性能的分布式内存对象缓存系统，用于动态 Web 应用以减轻数据库负载。它通过在内存中缓存数据和对象来减少读取数据库的次数，从而提高动态、数据库驱动网站的速度。Memcached 基于一个存储键/值对的 hashmap（文末 1）。其守护进程（daemon ）是用 C 写的，但是客户端可以用任何语言来编写，并通过 memcached 协议与守护进程通信。

注释：摘自百科


浏览器
应用服务器
RDBMS
RDBMS
memcached
memcached
memcached
memcached
首次访问：从RDBMS中取得数据保存到memcached
第二次后：从memcached中取得数据显示页面

memcahced 可以做什么？能给网站带来什么好处？



memcached 是一套分布式的快取系统，当初是 Danga Interactive 为了 LiveJournal 所发展的，但被许多软件（如 MediaWiki）所使用。这是一套开放源代码软件，以 BSD license 授权协议发布。[1]

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。[1]

memcached 的 API 使用 32 位元的循环冗余校验（CRC-32）计算键值后，将资料分散在不同的机器上。当表格满了以后，接下来新增的资料会以 LRU 机制替换掉。由于 memcached 通常只是当作快取系统使用，所以使用 memcached 的应用程式在写回较慢的系统时（像是后端的数据库）需要额外的程式码更新 memcached 内的资料[1]
memcached 是以 LiveJournal 旗下 Danga Interactive 公司的 Brad Fitzpatric 为首开发的一款软件。已成为 mixi、hatena、Facebook、Vox、LiveJournal 等众多服务中提高 Web 应用扩展性的重要因素。许多 Web 应用都将数据保存到 RDBMS 中，应用服务器从中读取数据并在浏览器中显示。但随着数据量的增大、访问的集中，就会出现 RDBMS 的负担加重、数据库响应恶化、网站显示延迟等重大影响。

这时就该 memcached 大显身手了。memcached 是高性能的分布式内存缓存服务器。一般的使用目的是，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。
Memcached 的守护进程（daemon ）是用 C 写的，但是客户端可以用任何语言来编写，并

通过 memcached 协议与守护进程通信。但是它并不提供冗余（例如，复制其 hashmap 条目）；当某个服务器 s 停止运行或崩溃了，所有存放在 s 上的键/值对都将丢失。
Memcached 由 Danga Interactive 开发，其最新版本发布于 2010 年，作者为 Anatoly Vorobey 和 Brad Fitzpatrick。用于提升 LiveJournal . com 访问速度的。LJ 每秒动态页面访问量几千次，用户 700 万。Memcached 将数据库负载大幅度降低，更好的分配资源，更快速访问。

memcached 作为高速运行的分布式缓存服务器，具有以下的特点。
· 协议简单
· 基于 libevent 的事件处理
· 内置内存存储方式
· memcached 不互相通信的分布式

摘自百科

memcached 特性

协议

memcached 的服务器客户端通信并不使用复杂的 XML 等格式，而使用简单的基于文本行的协议。
因此，通过 telnet 也能在 memcached 上保存数据、取得数据。下面是例子。
$ telnet localhost 11211
Trying 127.0.0.1

Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
set foo 0 0 3（保存命令）
bar（数据）
STORED（结果）
get foo（取得命令）
VALUE foo 0 3（数据）
bar（数据）

事件处理

libevent 是个程序库，它将 Linux 的 epoll、BSD 类操作系统的 kqueue 等事件处理功能封装成统一的接口。即使对服务器的连接数增加，也能发挥 O(1)的性能。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。关于事件处理这里就不再详细介绍，可以参考 Dan Kegel 的 The C10K Problem。

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，内容容量达到指定值之后，就基于 LRU(Least Recently Used)算法自动删除不使用的缓存。memcached 本身是为缓存而设计的服务器，因此并没有过多考虑数据的永久性问题。

通信分布式

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。那么，怎样进行分布式呢？这完全取决于客户端的实现。本文也将介绍 memcached 的分布式。

Hardware Requirements

Memcached is easy to spec out hardware for. In short, it is generally low on CPU usage, will take as much memory as you give it, and network usage will vary from mild to moderate, depending on the average size of your items.

CPU Requirements

Memcached is typically light on CPU usage, due to its goal to respond very fast. Memcached is multithreaded, defaulting to 4 worker threads. This doesn't necessarily mean you have to run 100 cores to have memcached meet your needs. If you're going to need to rely on memcached's multithreading, you'll know it. For the common case, any bits of CPU anywhere is usually sufficient.

RAM Requirements

The major point of memcached is to sew together sections of memory from multiple hosts and make your app see it as one large section of memory. The more memory the better. However, don't take memory away from other services that might benefit from it.

It is helpful to have each memcached server have roughly the same amount of memory available. Cluster uniformity means you can simply add and remove servers without having to care about one's particular "weight", or having one server hurt more if it is lost.

Avoid Swapping

Assign physical memory, with a few percent extra, to a memcached server. Do not over-allocate memory and expect swap to save you. Performance will be very, very poor. Take extra care to monitor if your server is using swap, and tune if necessary.

Is High Speed RAM Necessary?

Not so much, no. Getting that extra high speed memory will not likely net you visible benefits, unless you are issuing extremely high read traffic to memcached.

Hardware Layouts

Running Memcached on Webservers

An easy layout is to use spare memory on webservers or compute nodes that you may have. If you buy a webserver with 4G of RAM, but your app and OS only use 2G of RAM at most, you could assign 1.5G or more to memcached instances.

This has a good tradeoff of spreading memory out more thinly, so losing any one webserver will not cause as much pain.

Caveats being extra maintenance, and keeping an eye on your application's multi-get usage, as it can end up accessing every memcached in your list. You also run a risk of pushing a machine into swap or killing memcached if your app has a memory leak. Often it's a good idea to run hosts with very little swap, or no swap at all. Better to let an active service die than have it turn into a tarpit.

Running Memcached on Databases

Not a great idea. Don't do it. If you have a database host, give as much ram as possible to it. When cache misses do happen, you'll get more benefit from ensuring your indexes and data are already in memory.

Using Dedicated Hosts

Using dedicated hardware for memcached means you don't have to worry about other programs on the machine interfering with memcached. You can put a lot of memory (64G+) into a single host and have fewer machines for your memory requirements.

This has an added benefit of being able to more easily expand large amounts of memory space. Instead of adding new webservers that may go idle, you can add specialized machines to throw gobs of RAM at the problem.

This ends up having several caveats. The more you compress down your memcached cluster, the more pain you will feel when a host dies.

Lets say you have a cache hitrate of 90%. If you have 10 memcached servers, and 1 dies, your hitrate may drop to 82% or so. If 10% of your cache misses are getting through, having that jump to 18% or 20% means your backend is suddenly handling **twice** as many requests as before. Actual impact will vary since databases are still decent at handling repeat queries, and your typical cache miss will often be items that the database would have to look up regardless. Still, **twice**!

So lets say you buy a bunch of servers with 144G of ram, but you can only afford 4 of them. Now when you lose a single server, 25% of your cache goes away, and your hitrate can tank even harder.

Capacity Planning

Given the above notes on hardware layouts, be sure you practice good capacity planning. Get an idea for how many servers can be lost before your application is overwhelmed. Make sure you always have more than that.

If you cannot take down memcached instances, you ensure that upgrades (hardware or software), and normal failures are excessively painful. Save yourself some anguish and plan ahead.

Network

Network requirements will vary greatly by the average size of your memcached items. Your application should aim to keep them small, as it can mean the difference between being fine with gigabit inter-switch uplinks, or being completely toast.

Most deployments will have low requirements (< 10mbps per instance), but a heavy hit service can be quite challenging to support. That said, if you're resorting to infiniband or 10 gigabit ethernet to hook up your memcached instances, you could probably benefit from spreading them out more.

摘自：https://code.google.com/p/memcached/wiki/NewHardware

memcached 安装部署

参考：https://code.google.com/p/memcached/wiki/NewStart

yum 源安装

```
yum install memcached
```

源码安装

```
yum install libevent-devel
wget http://memcached.org/latest
tar -zxvf memcached-1.x.x.tar.gz
cd memcached-1.x.x
```

```
./configure --prefix=/usr/local/memcached
```

```
make && make test
sudo make install
```

Building an RPM

```
yum install gcc libevent libevent-devel
```

```
echo "%_topdir /home/you/rpmbuild" >> ~/.rpmmacros
mkdir -p /home/you/rpmbuild/{SPECS,BUILD,SRPMS,RPMS,SOURCES}
```

author：云尘　　网站：www.jikecloud.com

```
wget http://memcached.org/latest
rpmbuild -ta memcached-1.x.x.tar.gz
```

配置 Memcache

vi /etc/sysconfig/memcached
文件中内容如下
PORT=”11211″ 端口
USER=”root” 使用的用户名
MAXCONN=”1024″ 同时最大连接数
CACHESIZE=”64″ 使用的内存大小
OPTIONS=”" 附加参数

对部分参数的官网建议

Connection Limit

By default the max number of concurrent connections is set to 1024. Configuring this correctly is important. Extra connections to memcached may hang while waiting for slots to free up. You may detect if your instance has been running out of connections by issuing a `stats` command and looking at "listen_disabled_num". That value should be zero or close to zero.

Memcached can scale with a large number of connections very simply. The amount of memory overhead per connection is low (even lower if the connection is idle), so don't sweat setting it very high.

Lets say you have 5 webservers, each running apache. Each apache process has a MaxClients setting of 12. This means that the maximum number of concurrent connections you may receive is 5 x 12 (60). Always leave a few extra slots open if you can, for administrative tasks, adding more webservers, crons/scripts/etc.

Threading

Threading is used to scale memcached across CPU's. Its model is by "worker threads", meaning that each thread makes itself available to process as much as possible. Since using libevent allows good scalability with concurrent connections, each thread is able to handle many clients.

This is different from some webservers, such as apache, which use one process or one thread per active client connection. Since memcached is highly efficient, low numbers of threads are fine. In webserver land, it means it's more like nginx than apache.

By default 4 threads are allocated. Unless you are running memcached extremely hard, you should not set this number to be any higher. Setting it to very large values (80+) will not make it run any faster.

启动及测试 memcached

源码安装启动

/usr/local/bin/memcached -d -m 10 -u root -l 192.168.0.200 -p 12000 -c 256 -P /tmp/memcached.pid

-d 选项是启动一个守护进程，
-m 是分配给 Memcache 使用的内存数量，单位是 MB，我这里是 10MB，
-u 是运行 Memcache 的用户，我这里是 root，
-l 是监听的服务器 IP 地址，如果有多个地址的话，我这里指定了服务器的 IP 地址 192.168.0.200，
-p 是设置 Memcache 监听的端口，我这里设置了 12000，最好是 1024 以上的端口，
-c 选项是最大运行的并发连接数，默认是 1024，我这里设置了 256，按照你服务器的负载量来设定，
-P 是设置保存 Memcache 的 pid 文件，我这里是保存在 /tmp/memcached.pid，

详细参数介绍查看帮助信息

yum 安装启动

service memcached start

测试 memcached

```
[root@client01 ~]# netstat -tunlp |grep memcached
tcp        0      0 0.0.0.0:11211               0.0.0.0:*                   LISTEN      24173/memcached
tcp        0      0 :::11211                    :::*                        LISTEN      24173/memcached
udp        0      0 0.0.0.0:11211               0.0.0.0:*                               24173/memcached
udp        0      0 :::11211                    :::*                                    24173/memcached
[root@client01 ~]#
```

```
$ echo "stats settings" | nc localhost 11211
STAT maxbytes 67108864
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 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
END
```

php 访问 memcahed 插件安装部署

php 要访问 memcached，必须得有扩展驱动，就像 php 访问 mysql、mongodb 一样需要相关的扩展驱动

```
[root@client01 ~]# php -m  |grep mongo
mongo
[root@client01 ~]# php -m  |grep mysql
mysql
mysqli
pdo_mysql
[root@client01 ~]# php -m  |grep mem
memcache
memcached
[root@client01 ~]#
```

PHP 共有 2 种 Memcache 扩展，一个叫 Memcache，另一个叫 Memcached

Memcached 比较新，它依赖于 limemcached 库才能运行，不过它能完成基于 Memcache 服务的几乎所有功能，比如：Memcached::getResultCode ，它能返回上一次操作 Memcache 的结果，而 Memcache 则没有这个功能

二者选一就行了

php 扩展 memcache 部署

官网：https://pecl.php.net/package/memcache

github 地址：https://github.com/tricky/php-memcache

git clone https://github.com/tricky/php-memcache

依然 zlib

cd php-memcache
phpize
./configure
make && make install

记录下安装成功后的提示，类似于：
Installing shared extensions: /usr/lib/php/modules/

修改 **php** 配置文件

(关键字：extension=)
加上 extension=memcached.so

(关键字：handler)
加上 session.save_handler=memcached
session.save_path= "memcached 服务器 ip:端口"

php 扩展 memcached 部署

官网：https://pecl.php.net/package/memcached

github：https://github.com/php-memcached-dev/php-memcached

需要依赖 libmemcached 自行安装

```
$ phpize
$ ./configure
$ make
$ make test
```

和上面一样配置 php.ini 文件

测试是否安装成功

[root@client01 ~]# php -m |grep mem
memcache
memcached

测试：客户端向服务端读写数据

[root@client01 mlmwei]# cat mem.php
<?php
$memcache = new Memcache;
$memcache->connect('192.168.218.199', 11211) or die ("Could not connect");
$version = $memcache->getVersion();
echo "Server's version: ".$version."\n";
$tmp_object = new stdClass;
$tmp_object->str_attr = 'test';
$tmp_object->int_attr = 123;
$memcache->set('key', $tmp_object, false, 10) or die ("Failed to save data at the server");
echo "Store data in the cache (data will expire in 10 seconds)\n";
$get_result = $memcache->get('key');
echo "Data from the cache:\n";
var_dump($get_result);
?>

返回结果：

```
[root@client01 mlmwei]# php mem.php
Server's version: 1.4.24
Store data in the cache (data will expire in 10 seconds)
Data from the cache:
object(stdClass)#3 (2) {
  ["str_attr"]=>
  string(4) "test"
  ["int_attr"]=>
  int(123)
}
```

也可以网页访问

参考 http://luozlinux.blog.51cto.com/1388902/1226948

memcached 基本管理操作

详细参考：https://code.google.com/p/memcached/wiki/NewCommands

启动命令参考上面小结

基本管理命令

连接
telnet 127.0.0.1 11211

您将使用五种基本 memcached 命令执行最简单的操作。这些命令和操作包括：

- set
- add
- replace

- get
- delete

前三个命令是用于操作存储在 memcached 中的键值对的标准修改命令。它们都非常简单易用，且都使用如下 所示的语法：

command <key> <flags> <expiration time> <bytes>
<value>

参数	用法
key	key 用于查找缓存值
flags	可以包括键值对的整型参数，客户机使用它存储关于键值对的额外信息
expiration time	在缓存中保存键值对的时间长度（以秒为单位，0 表示永远）
bytes	在缓存中存储的字节点
value	存储的值（始终位于第二行）

现在，我们来看看这些命令的实际使用。

set
set 命令用于向缓存添加新的键值对。如果键已经存在，则之前的值将被替换。
注意以下交互，它使用了 set 命令：
set userId 0 0 5
12345
STORED

如果使用 set 命令正确设定了键值对，服务器将使用单词 **STORED** 进行响应。本示例向缓存中添加了一个键值对，其键为 userId，其值为 12345。并将过期时间设置为 0，这将向 memcached 通知您希望将此值存储在缓存中直到删除它为止。

add
仅当缓存中不存在键时，add 命令才会向缓存中添加一个键值对。如果缓存中已经存在键，则之前的值将仍然保持相同，并且您将获得响应 **NOT_STORED**。
下面是使用 add 命令的标准交互：
set userId 0 0 5
12345
STORED

add userId 0 0 5

```
55555
NOT_STORED

add companyId 0 0 3
564
STORED
```

replace

仅当键已经存在时，replace 命令才会替换缓存中的键。如果缓存中不存在键，那么您将从 memcached 服务器接受到一条 **NOT_STORED** 响应。

下面是使用 replace 命令的标准交互：

```
replace accountId 0 0 5
67890
NOT_STORED

set accountId 0 0 5
67890
STORED

replace accountId 0 0 5
55555
STORED
```

最后两个基本命令是 get 和 delete。这些命令相当容易理解，并且使用了类似的语法，如下所示：

```
command <key>
```

接下来看这些命令的应用。

get

get 命令用于检索与之前添加的键值对相关的值。您将使用 get 执行大多数检索操作。

下面是使用 get 命令的典型交互：

```
set userId 0 0 5
12345
STORED

get userId
VALUE userId 0 5
12345
END
```

get bob
END

如您所见，get 命令相当简单。您使用一个键来调用 get，如果这个键存在于缓存中，则返回相应的值。如果不存在，则不返回任何内容。

delete

最后一个基本命令是 delete。delete 命令用于删除 memcached 中的任何现有值。您将使用一个键调用 delete，如果该键存在于缓存中，则删除该值。如果不存在，则返回一条 **NOT_FOUND** 消息。

下面是使用 delete 命令的客户机服务器交互：

set userId 0 0 5
98765
STORED

delete bob
NOT_FOUND

delete userId
DELETED

get userId
END

可以在 memcached 中使用的两个高级命令是 gets 和 cas。gets 和 cas 命令需要结合使用。您将使用这两个命令来确保不会将现有的名称/值对设置为新值（如果该值已经更新过）。我们来分别看看这些命令。

gets

gets 命令的功能类似于基本的 get 命令。两个命令之间的差异在于，gets 返回的信息稍微多一些：64 位的整型值非常像名称/值对的“版本”标识符。

下面是使用 gets 命令的客户机服务器交互：

set userId 0 0 5
12345
STORED

get userId
VALUE userId 0 5
12345

END

gets userId
VALUE userId 0 5 **4**
12345
END

考虑 get 和 gets 命令之间的差异。gets 命令将返回一个额外的值 — 在本例中是整型值 4，用于标识名称/值对。如果对此名称/值对执行另一个 set 命令，则 gets 返回的额外值将会发生更改，以表明名称/值对已经被更新。清单 6 显示了一个例子：

set userId 0 0 5
33333
STORED

gets userId
VALUE userId 0 5 **5**
33333
END

您看到 gets 返回的值了吗？它已经更新为 5。您每次修改名称/值对时，该值都会发生更改。

cas
cas（check 和 set）是一个非常便捷的 memcached 命令，用于设置名称/值对的值（如果该名称/值对在您上次执行 gets 后没有更新过）。它使用与 set 命令相类似的语法，但包括一个额外的值：gets 返回的额外值。
注意以下使用 cas 命令的交互：
set userId 0 0 5
55555
STORED

gets userId
VALUE userId 0 5 **6**
55555
END

cas userId 0 0 5 **6**
33333
STORED

如您所见，我使用额外的整型值 6 来调用 gets 命令，并且操作运行非常顺序。现在，我们来看看清单 7 中的一系列命令：

```
set userId 0 0 5
55555
STORED

gets userId
VALUE userId 0 5 8
55555
END

cas userId 0 0 5 6
33333
EXISTS
```

注意，我并未使用 gets 最近返回的整型值，并且 cas 命令返回 EXISTS 值以示失败。从本质上说，同时使用 gets 和 cas 命令可以防止您使用自上次读取后经过更新的名称/值对。

缓存管理命令

最后两个 memcached 命令用于监控和清理 memcached 实例。它们是 stats 和 flush_all 命令。

stats

stats 命令的功能正如其名：转储所连接的 memcached 实例的当前统计数据。在下例中，执行 stats 命令显示了关于当前 memcached 实例的信息：

```
STAT pid 22459                  进程 ID
STAT uptime 1027046               服务器运行秒数
STAT time 1273043062               服务器当前 unix 时间戳
STAT version 1.4.4                服务器版本
STAT pointer_size 64               操作系统字大小(这台服务器是 64 位的)
STAT rusage_user 0.040000            进程累计用户时间
STAT rusage_system 0.260000           进程累计系统时间
STAT curr_connections 10             当前打开连接数
STAT total_connections 82            曾打开的连接总数
```

STAT connection_structures 13 服务器分配的连接结构数
STAT cmd_get 54 执行 get 命令总数
STAT cmd_set 34 执行 set 命令总数
STAT cmd_flush 3 指向 flush_all 命令总数
STAT get_hits 9 get 命中次数
STAT get_misses 45 get 未命中次数
STAT delete_misses 5 delete 未命中次数
STAT delete_hits 1 delete 命中次数
STAT incr_misses 0 incr 未命中次数
STAT incr_hits 0 incr 命中次数
STAT decr_misses 0 decr 未命中次数
STAT decr_hits 0 decr 命中次数
STAT cas_misses 0 cas 未命中次数
STAT cas_hits 0 cas 命中次数
STAT cas_badval 0 使用擦拭次数
STAT auth_cmds 0
STAT auth_errors 0
STAT bytes_read 15785 读取字节总数
STAT bytes_written 15222 写入字节总数
STAT limit_maxbytes 1048576 分配的内存数（字节）
STAT accepting_conns 1 目前接受的链接数
STAT listen_disabled_num 0
STAT threads 4 线程数
STAT conn_yields 0
STAT bytes 0 存储 item 字节数
STAT curr_items 0 item 个数
STAT total_items 34 item 总数
STAT evictions 0 为获取空间删除 item 的总数

此处的大多数输出都非常容易理解。稍后在讨论缓存性能时，我还将详细解释这些值的含义。至于目前，我们先来看看输出，然后再使用新的键来运行一些 set 命令，并再次运行 stats 命令，注意发生了哪些变化。

flush_all

flush_all 是最后一个要介绍的命令。这个最简单的命令仅用于清理缓存中的所有名称/值对。如果您需要将缓存重置到干净的状态，则 flush_all 能提供很大的用处。下面是一个使用 flush_all 的例子：

```
set userId 0 0 5
55555
STORED
```

get userId
VALUE userId 0 5
55555
END

flush_all
OK

get userId
END

参考：http://blog.csdn.net/zzulp/article/details/7823511

memcached 管理与监控工具 MemAdmin

官网：http://www.junopen.com/memadmin/

MemAdmin 是一款可视化的 Memcached 管理与监控工具，使用 PHP 开发，体积小，操作简单。

主要功能：

- 服务器参数监控：STATS、SETTINGS、ITEMS、SLABS、SIZES 实时刷新
- 服务器性能监控：GET、DELETE、INCR、DECR、CAS 等常用操作命中率实时监控
- 支持数据遍历，方便对存储内容进行监视
- 支持条件查询，筛选出满足条件的 KEY 或 VALUE
- 数组、JSON 等序列化字符反序列显示
- 兼容 memcache 协议的其他服务，如 Tokyo Tyrant (遍历功能除外)
- 支持服务器连接池，多服务器管理切换方便简洁

界面预览

更多请参考官网

安装部署

author：云尘　　网站：www.jikecloud.com

非常简单，先搭建一个 php web 运行环境，apache、 nginx 都可以

然后将下载的 memadmin 解压到根目录中

直接网页访问就好了

网页访问登录后添加 memcahed 服务器，就可以开始监控了

这些操作都很简单，自行研究

memcached 集群部署（待增加）

memcached 集群管理（待增加）

Hashmap

摘自：http://baike.baidu.com/view/1487140.htm

基于哈希表的 Map 接口的实现。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。（除了非同步和允许使用 null 之外，HashMap 类与 Hashtable 大致相同。）此类不保证映射的顺序，特别是它不保证该顺序恒久不变。 此实现假定哈希函数将元素适当地分布在各桶之间，可为基本操作（get 和 put）提供稳定的性能。迭代 collection 视图所需的时间与 HashMap 实例的“容量”（桶的数量）及其大小（键-值映射关系数）成比例。所以，如果迭代性能很重要，则不要将初始容量设置得太高（或将加载因子设置得太低）。

重要参数

HashMap 的实例有两个参数影响其性能：初始容量 和加载因子。容量是哈希表中桶的数量，初始容量只是哈希表在创建时的容量。加载因子 是哈希表在其容量自动增加之前可以达到多满的一种尺度。当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行 rehash 操作（即重建内部数据结构），从而哈希表将具有大约两倍的桶数。在 Java 编程语言中，加载因子默认值为 0.75，默认哈希表元为 101[1]。

同步机制

注意，此实现不是同步的。[1]如果多个线程同时访问一个哈希映射，而其中至少一个线程从结构上修改了该映射，则它必须 保持外部同步。（结构上的修改是指添加或删除一个或多个映射关系的任何操作；以防止对映射进行意外的非同步访问，如下：

Map m = Collections.synchronizedMap(new HashMap(...));

迭代器

由所有此类的“collection 视图方法”所返回的迭代器都是快速失败 的：在迭代器创建之后，如果从结构上对映射进行修改，除非通过迭代器本身的 remove 方法，而不冒在将来不确定的时间发生任意不确定行为的风险。

注意，迭代器的快速失败行为不能得到保证，一般来说，存在非同步的并发修改时，编写依赖于此异常的程序的做法是错误的，正确做法是：迭代器的快速失败行为应该仅用于检测程序错误。

遍历 Hash 中的元素

在 Hash 中可以直接使用一下方法遍历(所有键)KeySet

然后通过键可以找出需要的值

```
HashMap<String,String> mp = new  HashMap<String,String>();
for (String i : mp.keySet()) { //String 是 mp 中的键的对应类型 i 是对应的 KeySet 中的每一个键值
                System.out.println( mp.get(i));
    }
```

JAVA

基本概念

基于哈希表的 Map 接口的实现。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。（除了不同步和允许使用 null 之外，HashMap 类与 Hashtable 大致相同。）此类不保证映射的顺序，特别是它不保证该顺序恒久不变。另外，HashMap 是非线程安全的，也就是说在多线程的环境下，可能会存在问题，而 Hashtable 是线程安全的。

设计思路

此实现假定哈希函数将元素正确分布在各桶之间，可为基本操作（get 和 put）提供稳定的性能。迭代集合视图所需的时间与 HashMap 实例的“容量”（桶的数量）及其大小（键-值映射关系数）的和成比例。所以，如果迭代性能很重要，则不要将初始容量设置得太高（或将加载因子设置得太低）。 HashMap 的实例有两个参数影响其性能：初始容量和加载因子。容量是哈希表中桶的数量，初始容量只是哈希表在创建时的容量。加载因子是哈希表在其容量自动增加之前可以达到多满的一种尺度。当哈希表中的条目数超出了加载因子与当前容量的乘积时，通过调用 rehash 方法将容量翻倍。 通常，默认加载因子 (.75) 在时间和空间成本上寻求一种折衷。加载因子过高虽然减少了空间开销，但同时也增加了查询成本（在大多数 HashMap 类的操作中，包括 get 和 put 操作，都反映了这一点）。在设置初始容量时应该考虑到映射中所需的条目数及其加载因子，以便最大限度地降低 rehash 操作次数。如果初始容量大于最大条目数除以加载因子，则不会发生 rehash 操作。 如果很多映射关系要存储在 HashMap 实例中，则相对于按需执行自动的 rehash 操作以增大表的容量来说，使用足够大的初始容量创建它将使得映射关系能更有效地存储注意，此实现不是同步的。如果多个线程同时访问此映射，而其中至少一个线程从结构上修改了该映射，则它必须保持外部同步。（结构上的修改是指添加或删除一个或多个映射

关系的操作；仅改变与实例已经包含的键关联的值不是结构上的修改。）这一般通过对自然封装该映射的对象进行同步操作来完成。如果不存在这样的对象，则应该使用 Collections.synchronizedMap 方法来“包装”该映射。最好在创建时完成这一操作，以防止对映射进行意外的不同步访问，如下所示： Map m = Collections.synchronizedMap(new HashMap(...)); 由所有此类的“集合视图方法”所返回的迭代器都是快速失败的：在迭代器创建之后，如果从结构上对映射进行修改，除非通过迭代器自身的 remove 或 add 方法，其他任何时间任何方式的修改，迭代器都将抛出 ConcurrentModificationException。因此，面对并发的修改，迭代器很快就会完全失败，而不冒在将来不确定的时间任意发生不确定行为的风险。 注意，迭代器的快速失败行为不能得到保证，一般来说，存在不同步的并发修改时，不可能作出任何坚决的保证。快速失败迭代器尽最大努力抛出。编写依赖于此异常程序的方式是错误的，正确做法是：迭代器的快速失败行为应该仅用于检测程序错误。

重写方法

前面介绍了，HashMap 是基于 HashCode 的，在所有对象的超类 Object 中有一个 HashCode()方法，但是它和 equals 方法一样，并不能适用于所有的情况，这样我们就需要重写自己的 HashCode()方法。下面就举这样一个例子：

```
import java.util.*;
public class Exp2 {
public static void main(String[] args) {
HashMap h2 = new HashMap();
for (int i = 0; i < 10; i++) {
h2.put(new Element(i), new Figureout());
System.out.println("h2:");
System.out.println("Get the result for Element:");}
Element test = new Element(3);
if (h2.containsKey(test)) {System.out.println((Figureout) h2.get(test));} else {
System.out.println("Not found");}
class Element {
int number;
public Element(int n) {
number = n;}
class Figureout {
Random r = new Random();
boolean possible = r.nextDouble() > 0.5;
public String toString() {
if (possible) {
return "OK!";
} else {
return "Impossible!";
}
}
```

}

在这个例子中，Element 用来索引对象 Figureout,也即 Element 为 key，Figureout 为 value。在 Figureout 中随机生成一个浮点数，如果它比 0.5 大，打印"OK!"，否则打印"Impossible!"。之后查看 Element(3)对应的 Figureout 结果如何。

结果却发现，无论你运行多少次，得到的结果都是"Not found"。也就是说索引 Element(3)并不在 HashMap 中。这怎么可能呢？

原因得慢慢来说：Element 的 HashCode 方法继承自 Object，而 Object 中的 HashCode 方法返回的 HashCode 对应于当前的地址，也就是说对于不同的对象，即使它们的内容完全相同，用 HashCode（）返回的值也会不同。这样实际上违背了我们的意图。因为我们在使用 HashMap 时，希望利用相同内容的对象索引得到相同的目标对象，这就需要 HashCode()在此时能够返回相同的值。在上面的例子中，我们期望 new Element(i)(i=5)与 Element test=new Element(5)是相同的，而实际上这是两个不同的对象，尽管它们的内容相同，但它们在内存中的地址不同。因此很自然的，上面的程序得不到我们设想的结果。下面对 Element 类更改如下：

```
class Element {
int number;
public Element(int n) {
number = n;
}
public int hashCode() {
return number;
}
public boolean equals(Object o) {
return (o instanceof Element) && (number == ((Element) o).number);
}
}
```

在这里 Element 覆盖了 Object 中的 hashCode()和 equals()方法。覆盖 hashCode()使其以 number 的值作为 hashcode 返回，这样对于相同内容的对象来说它们的 hashcode 也就相同了。而覆盖 equals()是为了在 HashMap 判断两个 key 是否相等时使结果有意义（有关重写 equals()的内容可以参考我的另一篇文章《重新编写 Object 类中的方法》）。修改后的程序运行结果如下：

```
h2:
Get the result for Element:
Impossible!
```

请记住：如果你想有效的使用 HashMap，你就必须重写在其的 HashCode()。

重写原则

还有两条重写 HashCode()的原则：

不必对每个不同的对象都产生一个唯一的 hashcode，只要你的 HashCode 方法使 get()能够得到 put()放进去的内容就可以了。即"不唯一原则"。

生成 hashcode 的算法尽量使 hashcode 的值分散一些，不要很多 hashcode 都集中在一个范围内，这样有利于提高 HashMap 的性能。即"分散原则"。

至于第二条原则的具体原因，有兴趣者可以参考 Bruce Eckel 的《Thinking in Java》，在那里有对 HashMap 内部实现原理的介绍，这里就不赘述了。

掌握了这两条原则，你就能够用好 HashMap 编写自己的程序了。不知道大家注意没有，java.lang.Object 中提供的三个方法：clone()，equals()和 hashCode()虽然很典型，但在很多情况下都不能够适用，它们只是简单的由对象的地址得出结果。这就需要我们在自己的程序中重写它们，其实 java 类库中也重写了千千万万个这样的方法。利用面向对象的多态性——覆盖，Java 的设计者很优雅的构建了 Java 的结构，也更加体现了 Java 是一门纯 OOP 语言的特性。

分析

在 Java 的世界里，无论类还是各种数据，其结构的处理是整个程序的逻辑以及性能的关键。由于本人接触了一个有关性能与逻辑同时并存的问题，于是就开始研究这方面的问题。找遍了大大小小的论坛，也把《Java 虚拟机规范》，《apress,.java.collections.(2001),.bm.ocr.6.0.shareconnector》，和《Thinking in Java》翻了也找不到很好的答案，于是一气之下把 JDK 的 src 解压出来研究，豁然开朗，遂写此文，跟大家分享感受和顺便验证我理解还有没有漏洞。 这里就拿 HashMap 来研究吧。

HashMap 可谓 JDK 的一大实用工具，把各个 Object 映射起来，实现了“键 - - 值”对应的快速存取。但实际里面做了些什么呢？

在这之前，先介绍一下负载因子和容量的属性。大家都知道其实一个 HashMap 的实际容量就是因子*容量，其默认值是 16×0.75=12； 这个很重要，对效率有一定影响！当存入 HashMap 的对象超过这个容量时，HashMap 就会重新构造存取表。这就是一个大问题，我后面慢慢介绍，反正，如果你已经知道你大概要存放多少个对象，最好设为该实际容量的能接受的数字。

两个关键的方法，put 和 get：

先有这样一个概念，HashMap 是声明了 Map，Cloneable, Serializable 接口，和继承了 AbstractMap 类，里面的 Iterator 其实主要都是其内部类 HashIterator 和其他几个 iterator 类实现，当然还有一个很重要的继承了 Map.Entry 的 Entry 内部类，由于大家都有源代码，大家有兴趣可以看看这部分，我主要想说明的是 Entry 内部类。它包含了 hash，value，key 和 next 这四个属性，很重要。put 的源码如下

public Object put(Object key, Object value) {

Object k = maskNull(key);

这个就是判断键值是否为空，并不很深奥，其实如果为空，它会返回一个 static Object 作为键值，这就是为什么 HashMap 允许空键值的原因。

int hash = hash(k);

int i = indexFor(hash, table.length);

这连续的两步就是 HashMap 最牛的地方！研究完我都汗颜了，其中 hash 就是通过 key 这个 Object 的 hashcode 进行 hash，然后通过 indexFor 获得在 Object table 的索引值。

table？？？不要惊讶，其实 HashMap 也神不到哪里去，它就是用 table 来放的。最牛的就是用 hash 能正确的返回索引。其中的 hash 算法，我跟 JDK 的作者 Doug 联系过，他建议我看看《The art of programing vol3》可恨的是，我之前就一直在找，我都找不到，他这样一提，我就更加急了，可惜口袋空空啊！！！

不知道大家有没有留意 put 其实是一个有返回的方法，它会把相同键值的 put 覆盖掉并返回旧的值！如下方法彻底说明了 HashMap 的结构，其实就是一个表加上在相应位置的 Entry 的链表：

for (Entry e = table[i]; e != null; e = e.next) {
if (e.hash == hash && eq(k, e.key)) {
Object oldvalue = e.value;
e.value = value; //把新的值赋予给对应键值。
e.recordAccess(this); //空方法，留待实现
return oldvalue; //返回相同键值的对应的旧的值。
}
}
modCount++; //结构性更改的次数
addEntry(hash, k, value, i); //添加新元素，关键所在！
return null; //没有相同的键值返回
}

我们把关键的方法拿出来分析：

void addEntry(int hash, Object key, Object value, int bucketIndex) {
table[bucketIndex] = new Entry(hash, key, value, table[bucketIndex]);

因为 hash 的算法有可能令不同的键值有相同的 hash 码并有相同的 table 索引，如：key=“33”和 key=Object g 的 hash 都是 - 8901334，那它经过 indexfor 之后的索引一定都为 i，这样在 new 的时候这个 Entry 的 next 就会指向这个原本的 table[i]，再有下一个也如此，形成一个链表，和 put 的循环对定 e.next 获得旧的值。到这里，HashMap 的结构，大家也十分明白了吧？

if (size++ >= threshold) //这个 threshold 就是能实际容纳的量
resize(2 * table.length); //超出这个容量就会将 Object table 重构

所谓的重构也不神，就是建一个两倍大的 table（我在别的论坛上看到有人说是两倍加 1，把我骗了），然后再一个个 indexfor 进去！注意！！这就是效率！！如果你能让你的 HashMap 不需要重构那么多次，效率会大大提高！

说到这里也差不多了，get 比 put 简单得多，大家，了解 put，get 也差不了多少了。对于 collections 我是认为，它是适合广泛的，当不完全适合特有的，如果大家的程序需要特殊的用途，自己写吧，其实很简单。（作者是这样跟我说的，他还建议我用 LinkedHashMap,我看了源码以后发现，LinkHashMap 其实就是继承 HashMap 的，然后 override 相应的方法，有兴趣的同人，自己 looklook）建个 Object table，写相应的算法，就 ok 啦。

举个例子吧，像 Vector，list 啊什么的其实都很简单，最多就多了的同步的声明，其实如果要实现像 Vector 那种，插入，删除不多的，可以用一个 Object table 来实现，按索引存取，添加等。

如果插入，删除比较多的，可以建两个 Object table，然后每个元素用含有 next 结构的，一个 table 存，如果要插入到 i，但是 i 已经有元素，用 next 连起来，然后 size++，并

在另一个 table 记录其位置。