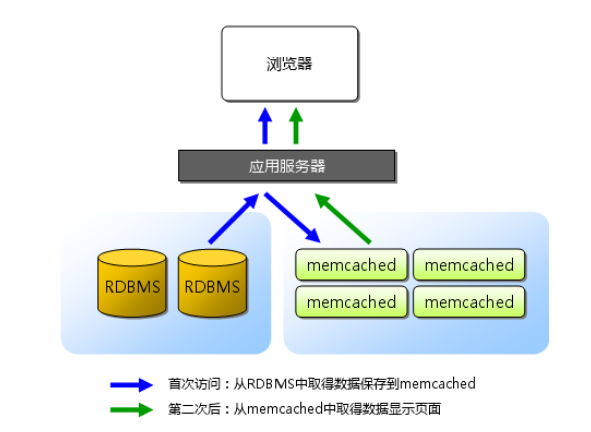
**Memcache**

## 一 MemCached概念

1 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。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。memcached是一个免费开源、高性能、分布式的内存对象缓存系统，本质上是通用的，但旨在通过加速动态Web应用程序来减轻数据库负载。

**MemCache是一个自由、源码开放、高性能、分布式的分布式内存对象缓存系统，用于动态Web应用以减轻数据库的负载。**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，从而提高了网站访问的速度。MemCaChe是一个存储键值对的HashMap，在内存中对任意的数据（比如字符串、对象等）所使用的key-value存储，数据可以来自数据库调用、API调用，或者页面渲染的结果。MemCache设计理念就是小而强大，它简单的设计促进了快速部署、易于开发并解决面对大规模的数据缓存的许多难题，而所开放的API使得MemCache能用于Java、C/C++/C#、Perl、Python、PHP、Ruby等大部分流行的程序语言。



**2 Memcache设计思想**

（1）简单的key/value存储，服务器不关心数据本身意义及结构，是可序列化数据即可。

（2）功能实现一半依赖与客户端，一半基于服务器端。

（3）各服务器间彼此无视，不在服务器间进行数据同步。

（4）O(1)的执行效率。

（5）内存空间的再利用，Lazy Expiration + LRU 机制。

**3 Memcache 与 memcached的区别**

**（1）客户端两者区别：**

两个不同版本PHP的memcached的客户端

memcache是原生版本，完全是在PHP框架内开发的，支持OO和非OO两套接口并存；而memcached是建立在libmemcached的基础上，只支持OO接口。

其他一些实现和支持方面的不同等。

**（2）服务器端两者区别：**

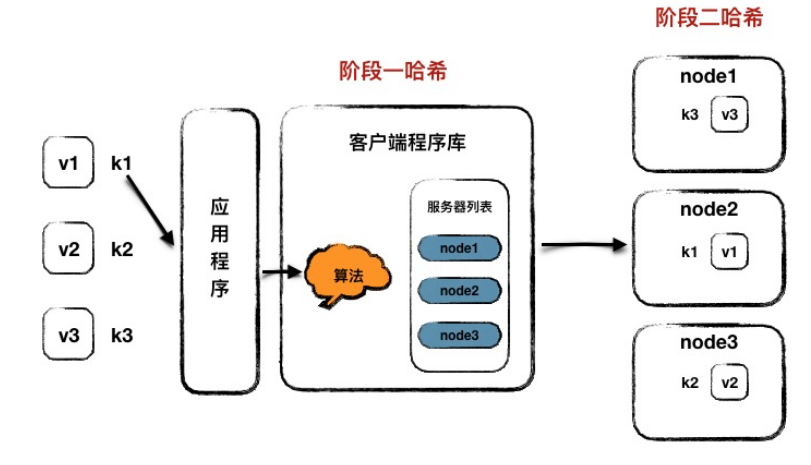
Memcache 是项目名称

Memcached 是Memcache服务器端可执行文件的名称，Memcached是以守护程序(监听)方式运行于一个或多个服务器中，随时会接收客户端的连接和操作。

**4 Memcached内置内存存储方式**

**（1）Memcached 的高性能**

首先从内存模型来研究memcached：C++里分配内存有两种方式，预先分配和动态分配内存，显然预先分配内存会使程序比较快，但是它的缺点是不能有效利用内存；而动态分配可以有效利用内存，但是会使程序运行效率下降，**memcached的内存分配就是基于以上原理，显然为了获得更快的速度，有时候我们不得不以空间换时间。**  
Memcached的高性能源于两阶段哈希(two-stage-hash)结构。Memcached就像一个巨大的、存储了很多<key, value>对的哈希表，通过key可以存储或查询任意的数据。客户端可以把数据存储在多台memcached上，当查询数据时，客户端首先参考节点列表计算出key的哈希值（阶段一哈希），进而选中一个节点；客户端将请求发送给选中的节点，然后memcached节点通过一个内部的哈希算法（阶段二哈希），查找真正的数据（item）并返回个客户端。从实现的角度看，memcached是一个非阻塞、基于事件驱动的服务器程序。  
为了提高性能，memcacahed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丢失。另外，内存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本身是为了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。



**（2）Slab Allocator 内存分配、管理机制**

**1）在该机制出现以前，内存的分配是通过对所有记录简单地进行malloc和free 来进行的。**但是，这种方式会导致内存碎片，加重操作系统内存管理器的负担，最坏的情况下，会导致操作系统比memcached 进程本身还慢。Slab Allocator 就是为解决该问题而诞生的, 它按照预先规定的大小，将分配的内存分割成特定长度的块，以完成解决内存碎片的问题。存储结构图如下：

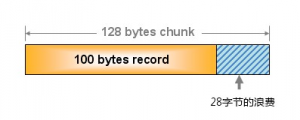


Memcached的存储涉及到slab、page、chunk三个概念  
chunk：固定大小的内存空间，用于缓存记录，默认为88Byte。  
page：分配给Slab的内存空间，默认是1M。分配给Slab之后根据Slab的大小切成chunk。  
slab：同样大小的chunk组成一类slab。

**2）.在Slab中缓存记录的原理**  
memcached 根据收到的数据大小，选择最适合数据大小的Slab，memcached中保存着Slab内空闲chunk的列表，根据该列表选择chunk，然后将数据缓存其中。Slab allocator分配的内存不会释放，而是重复利用。



**（3）.Slab Allocator 的缺点**  
由于分配的是特定长度的内存，因此无法有效的利用分配的内存。对与该问题没有完美的解决方案，但可以调节slab class的大小差别来减少空间浪费。



**（4）使用Growth Factor进行调优**  
memcached在启动时制定Growth Factor因子（通过f选项），就可以在某种程度上控制slab之间的差异。默认值时1.25.

**5 Memcached 删除机制**

**（1）Lazy Expiration**

memcached 不会释放已经分配的内存，记录过期后，客户端无法在看到这条记录，其存储空间可以再利用。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。这种技术被称为Lazy Expiration**。因此，memcached不会在过期监视上耗费CPU时间。

**（2）LRU（Least Recently Used）**

memcached会优先使用已超时的记录空间，但即使如此，也会发生追加新记录时空间不足的情况，此时就要使用名为LRU机制来分配空间。顾名思义，这是删除“最近最少使用”记录的机制。因此，当memcahced的内存空间不足时（无法从slab class获取新的空间时），就从最近未被使用的记录中搜索，并将其空间分配给新的记录。

**6 Memcached 分布式算法**

**（1）Memcached "分布式"**

特别澄清一个问题：MemCache虽然被称为"分布式缓存"，但是MemCache本身完全不具备分布式的功能，Memcache集群之间不会相互通信（与之形成对比的，比如JBoss Cache，某台服务器有缓存数据更新时，会通知集群中其他机器更新缓存或清除缓存数据），所谓的"分布式"，完全依赖于客户端程序的实现。前面已经讲过memcached的两段哈希了，数据的保存和获取都使用相同的算法。这样将不同的键保存到不同的服务器上，就实现了memcached的分布式。Memcached服务器增多后，键就会分散，即使一台memcached服务器发生故障无法连接，也不会影响其他的缓存，系统依然能继续进行。

**（2）余数分布式算法**

就是“根据服务器台数的余数进行分散”。求得键的整数哈希值，再除以服务器台数，根据其余数来选择服务器。

余数算法的缺点：余数计算的方法简单，数据的分散性也相当优秀，但也有其缺点。那就是当添加或移除服务器时，缓存重组的代价相当巨大。添加服务器后，余数就会产生巨变，这样就无法获取与保存时相同的服务器，从而影响缓存的命中率。

**（3）Consistent hashing（一致哈希）**

**（1）传统的取模方式**

例如10条数据，3个节点，如果按照取模的方式，那就是

node a: 0,3,6,9

node b: 1,4,7

node c: 2,5,8

当增加一个节点的时候，数据分布就变更为

node a:0,4,8

node b:1,5,9

node c: 2,6

node d: 3,7

总结：数据3,4,5,6,7,8,9在增加节点的时候，都需要做搬迁，成本太高

**（2）一致性哈希方式**

最关键的区别就是，对节点和数据，都做一次哈希运算，然后比较节点和数据的哈希值，数据取和节点最相近的节点做为存放节点。这样就保证当节点增加或者减少的时候，影响的数据最少。

还是拿刚刚的例子，（用简单的字符串的ascii码做哈希key）：

十条数据，算出各自的哈希值

0：192

1：196

2：200

3：204

4：208

5：212

6：216

7：220

8：224

9：228

有三个节点，算出各自的哈希值

node a: 203

node g: 209

node z: 228

这个时候比较两者的哈希值，如果大于228，就归到前面的203，相当于整个哈希值就是一个环，对应的映射结果：

node a: 0,1,2

node g: 3,4

node z: 5,6,7,8,9

这个时候加入node n, 就可以算出node n的哈希值：

node n: 216

这个时候对应的数据就会做迁移：

node a: 0,1,2

node g: 3,4

node n: 5,6

node z: 7,8,9

这个时候只有5和6需要做迁移。另外，这个时候如果只算出三个哈希值，那再跟数据的哈希值比较的时候，很容易分得不均衡，因此就引入了虚拟节点的概念，通过把三个节点加上ID后缀等方式，每个节点算出n个哈希值，均匀的放在哈希环上，这样对于数据算出的哈希值，能够比较散列的分布（详见下面代码中的replica）。通过这种算法做数据分布，在增减节点的时候，可以大大减少数据的迁移规模。

**7 MemCached使用场景**

访问频繁的数据库数据（身份token、首页动态）

访问频繁的查询条件和结果

作为Session的存储方式（提升Session存取性能）

页面缓存

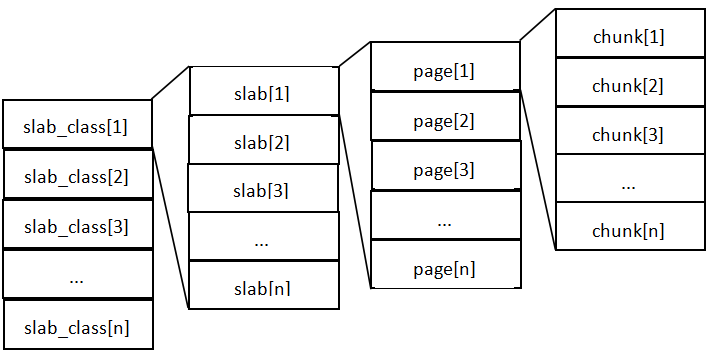
更新频繁的非重要数据（访客量、点击次数）

大量的hot数据

**8 MemCached工作原理**

MemCached采用了C/S架构，在Server端启动后，**以守护程序的方式，监听客户端的请求。**启动时可以指定监听的IP（服务器的内网ip/外网ip）、端口号（所以做分布式测试时，一台服务器上可以启动多个不同端口号的MemCached进程）、使用的内存大小等关键参数。一旦启动，服务就会一直处于可用状态。为了提高性能，MemCached缓存的数据全部存储在MemCached管理的内存中，所以重启服务器之后缓存数据会清空，不支持持久化。

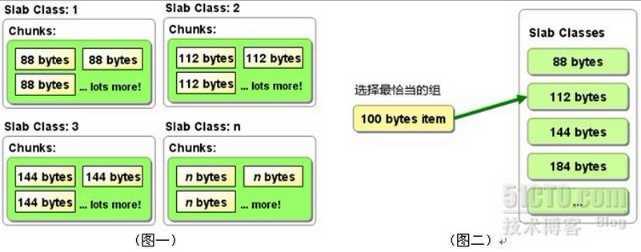
**（1）Memcached内存管理**



slab\_class里，存放的是一组组chunk大小相同的slab每个slab里面包含若干个page，page的默认大小是1M，如果slab大小100M，就包含100个page每个page里面包含若干个chunk，chunk是数据的实际存放单位，每个slab里面的chunk大小相同

**（2）内存分配方式**

Memcached利用slab allocation机制来分配和管理内存，**它按照预先规定的大小，将分配的内存分割成特定长度的内存块，再把尺寸相同的内存块分成组，数据在存放时，根据键值大小去匹配slab大小，找就近的slab存放，所以存在空间浪费现象。**而传统的内存管理方式是，使用完通过malloc分配的内存后通过free来回收内存，这种方式容易产生内存碎片并降低操作系统对内存的管理效率。存放数据时，首先slab要申请内存，申请内存是以page为单位的。所以在放入第一个数据的时候，无论大小为多少，都会有1M大小的page被分配给该slab。申请到page后，slab会将这个page的内存按chunk的大小进行切分，这样就变成了一个chunk数组，最后从这个chunk数组中选择一个用于存储数据。

示例：  


MemCache中的value存放位置是由value的大小决定，value会被存放到与chunk大小最接近的一个slab中，比如slab[1]的chunk大小为88字节、slab[2]的chunk大小为112字节、slab[3]的chunk大小为144字节（默认相邻slab内的chunk基本以1.25为比例进行增长，MemCache启动时可以用-f指定这个比例），那么一个100字节的value，将被放到2号slab中。

## 二 MemCached的安装

**1 下载MemCached软件**

**2 将memcached.exe文件拷贝到应用目录下，**注意目录不能出现中文和空格等特殊字符

**3 在memcached.exe所在目录下**，打开cmd终端(注意：需要以admin的身份来打开)

memcached.exe -d install 安装

memcached.exe -d uninstall 卸载

**4 启动memcached**

Memcached.exe -d start

使用netstat -an命令查看是否启动成功，如果查看到11211端口在监听，说明启动ok，如果启动不成功，可以使用memcached.exe -p 端口号 的方式来启动，这种方式是以shell的方式来启动的，启动之后，窗口不能关闭，一旦窗口关闭，程序也就终止了。

**5 常用命令介绍**

**（1）存储及删除命令，简单易用，使用语法如下：**

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

<value>

参数说明如下：

command set/add/replace

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数

value 存储的值（始终位于第二行）

set 命令用于向缓存添加新的键值对，如果已经存在，则之前的值将被替换。设置成功，服务器会使用单词STORED进行响应。

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。

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。

delete 删除命令的语法：command <key>，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。

**（2）读取命令**

get 命令用于检索与之前添加的键值对相关的值。当使用一个键来调用 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。get命令的key可以表示一个或者多个键，键之间以空格隔开。

gets 命令比普通的get命令多返回一个数字。这个数字可以检查数据是否发生变化：当key对应的数据变化时，这个数字也会改变。

cas 即check and set，只有当最后一个参数和gets所获取的参数匹配时才能存储，否则返回“EXISTS”。

**（3）统计命令**

stats 显示服务器信息、统计数据

stats settings 显示所有的参数设置

stats slabs 显示各个slab的信息，包括chunk的大小、数目、使用情况等

stats items 显示各个slab的item信息

stats cachedump slab\_id limit\_num 查看指定slab前limit\_num个item，[key，expiration\_time]

flush\_all 用于清理缓存中所有键值对

stats reset 清空统计数据

## 三 操作MemCached

**1 使用telnet操作**

（1）登录telnet，连接memcached服务

telnet 127.0.0.1 11211

（2） 添加

基本语法：

add key 0(压缩标志位) 缓存时间(秒) 数据大小(字符)

举例：

add name 0 30 6

chhliu

返回STORED说明添加成功

（3） 获取

举例：

get name

返回VALUE name 0 6

chhliu

END

则说明成功

隔30s后再获取，会返回END，说明缓存过期

（4）修改

set key 0 存放时间 数据大小

replace key 0 存放时间 数据大小

（5） 删除

基本语法：

delete key

**2 使用java操作**

（1） 新建一个maven项目，并加入依赖的jar包，pom文件如下：

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.chhliu.myself</groupId>

<artifactId>memcached</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>memcached</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.apache.common</groupId>

<artifactId>pool</artifactId>

<version>1.5.6</version>

<scope>system</scope>

<systemPath>${basedir}/lib/commons-pool-1.5.6.jar</systemPath>

</dependency>

<dependency>

<groupId>org.apache</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.6.1</version>

<scope>system</scope>

<systemPath>${basedir}/lib/slf4j-api-1.6.1.jar</systemPath>

</dependency>

<dependency>

<groupId>org.apache</groupId>

<artifactId>slf4j-simple</artifactId>

<version>1.6.1</version>

<scope>system</scope>

<systemPath>${basedir}/lib/slf4j-simple-1.6.1.jar</systemPath>

</dependency>

<dependency>

<groupId>org.apache</groupId>

<artifactId>java\_memcached-release</artifactId>

<version>2.6.6</version>

<scope>system</scope>

<systemPath>${basedir}/lib/java\_memcached-release\_2.6.6.jar</systemPath>

</dependency>

</dependencies>

</project>

（2） 新建测试类，代码如下：

public class MyDefCache {

public static void main(String[] args) {

MemCachedClient client = new MemCachedClient();

String[] addr = { "127.0.0.1:11211", "127.0.0.1:11212", "127.0.0.1:11213" };

Integer[] weights = { 3, 3, 3 };

SockIOPool pool = SockIOPool.getInstance();

pool.setServers(addr);

pool.setWeights(weights);

pool.setInitConn(5);

pool.setMinConn(5);

pool.setMaxConn(200);

pool.setMaxIdle(1000 \* 30 \* 30);

pool.setMaintSleep(30);

pool.setNagle(false);

pool.setSocketTO(30);

pool.setSocketConnectTO(0);

pool.initialize();

// 将数据放入缓存

client.set("name", "chhliu");

// 将数据放入缓存,并设置失效时间

client.set("company", "tencent", 60);

// 获取缓存数据

String name = (String) client.get("name");

String company = (String) client.get("company");

System.out.println("name:"+name+" company:"+company);

// 删除缓存数据

// client.delete("company");

// 获取缓存数据

// company = (String) client.get("company");

// System.out.println("after delete company:"+company);

}

}

（3） 以上代码中涉及到的API，在后面统一说明。

从上面的代码可以看出，我们连接了3台memcached服务器，分别为127.0.0.1:11211,127.0.0.1:11212,127.0.0.1:11213

运行结果如下：

name:chhliu company:tencent

我们再通过前面的方式来看下这2条数据是怎么分布的，

（4） 通过从远程查看服务器上的值，说明memcached已经实现了分布式的缓存，那么memcached是怎么来实现分布式缓存的了，下面我们来了解下：

@Memcached是基于C/S的架构实现的

@MemCached是基于客户端来实现分布式的，Server端不支持分布式缓存

@动态建立连接，每次操作MemCached服务端的时候，通过客户端动态建立连接，例如name:chhliu添加的时候是添加到127.0.0.1:11211服务端，当我们需要获取的时候，客户端会只连接127.0.0.1:11211服务器，而不是同时连了上面3个服务器。也就是说MemCached客户端每次只保证一个连接。

@散列算法，可以根据权重来动态计算缓存数据的分布，数据缓存更均匀，合理。

## 四 由Memcached分布式思想，设计分布式数据库

以下是我在使用MemCached的过程中的一些感悟和理解，其实MemCached实现分布式的思想很优秀，在很多需要分布式的地方都很实用。以下是用这种思想，实现的一个分布式数据库系统。

上图是一个整体的架构，主要是运用了Memcached分布式的实现思想，这种架构下的分布式数据库结构有如下优点：

1 实现分布式的数据库操作：各主数据库之间无须互相通信同步数据，节约带宽，主从之间通信，备份数据。

2 提供高可用性：主从结构，无缝切换。

3 C/S架构，直接通过客户端来实现Server端的分布式操作。

4 实现了数据库服务和业务程序的分离，当业务程序不可用或者是业务服务器宕机的时候，对数据库服务无任何影响。

5 动态的多节点负载均衡。

6 客户端保证每次只连接一个主数据库服务端，并不是同时建立多个连接，也就是说在真正需要建立连接的时候才会建立一个连接，减少维护连接的开销。

7 可扩展性好，横向扩展简单方便，只需要将主从Server端中的主数据库在客户端注册即可。

8 集群中性能好，每个集群中的节点类似于单节点下的主从备份结构。

## 五 Memcached Java Client API详解

**1 SockIOPool**

这个类用来创建管理客户端和服务器通讯连接池，客户端主要的工作包括数据通讯 服务器定位 hash码生成等都是由这个类完成的。

· public static SockIOPool getInstance()

· 获得连接池的单态方法。这个方法有一个重载方法getInstance( String poolName )，每个poolName只构造一个SockIOPool实例。缺省构造的poolName是default。

· 如果在客户端配置多个memcached服务，一定要显式声明poolName。

· public void setServers( String[] servers )

· 设置连接池可用的cache服务器列表，server的构成形式是IP:PORT（如：127.0.0.1:11211）

· public void setWeights( Integer[] weights )

· 设置连接池可用cache服务器的权重，和server数组的位置一一对应

· 其实现方法是通过根据每个权重在连接池的bucket中放置同样数目的server（如下代码所示），因此所有权重的最大公约数应该是1，不然会引起bucket资源的浪费

· public void setInitConn( int initConn )

· 设置开始时每个cache服务器的可用连接数

· public void setMinConn( int minConn )

· 设置每个服务器最少可用连接数

· public void setMaxConn( int maxConn )

· 设置每个服务器最大可用连接数

· public void setMaxIdle( long maxIdle )

· 设置可用连接池的最长等待时间

· public void setMaintSleep( long maintSleep )

· 设置连接池维护线程的睡眠时间

· 设置为0，维护线程不启动

· 维护线程主要通过log输出socket的运行状况，监测连接数目及空闲等待时间等参数以控制连接创建和关闭。

· public void setNagle( boolean nagle )

· 设置是否使用Nagle算法，因为我们的通讯数据量通常都比较大（相对TCP控制数据）而且要求响应及时，因此该值需要设置为false（默认是true）

· public void setSocketTO( int socketTO )

· 设置socket的读取等待超时值

· public void setSocketConnectTO( int socketConnectTO )

· 设置socket的连接等待超时值

· public void setAliveCheck( boolean aliveCheck )

· 设置连接心跳监测开关。

· 设为true则每次通信都要进行连接是否有效的监测，造成通信次数倍增，加大网络负载，因此该参数应该在对HA要求比较高的场合设为TRUE，默认状态是false。

· public void setFailback( boolean failback )

· 设置连接失败恢复开关

· 设置为TRUE，当宕机的服务器启动或中断的网络连接后，这个socket连接还可继续使用，否则将不再使用，默认状态是true，建议保持默认。

· public void setFailover( boolean failover )

· 设置容错开关

· 设置为TRUE，当当前socket不可用时，程序会自动查找可用连接并返回，否则返回NULL，默认状态是true，建议保持默认。

· public void setHashingAlg( int alg )

· 设置hash算法

· alg=0 使用String.hashCode()获得hash code,该方法依赖JDK，可能和其他客户端不兼容，建议不使用

· alg=1 使用original 兼容hash算法，兼容其他客户端

· alg=2 使用CRC32兼容hash算法，兼容其他客户端，性能优于original算法

· alg=3 使用MD5 hash算法

· 采用前三种hash算法的时候，查找cache服务器使用余数方法。采用最后一种hash算法查找cache服务时使用consistent方法。

· public void initialize()

· 设置完pool参数后最后调用该方法，启动pool。

**2 MemCachedClient**

· public void setCompressEnable( boolean compressEnable )

· 设定是否压缩放入cache中的数据

· 默认值是ture

· 如果设定该值为true，需要设定CompressThreshold?

· public void setCompressThreshold( long compressThreshold )

· 设定需要压缩的cache数据的阈值

· 默认值是30k

· public void setPrimitiveAsString( boolean primitiveAsString )

· 设置cache数据的原始类型是String

· 默认值是false

· 只有在确定cache的数据类型是string的情况下才设为true，这样可以加快处理速度。

· public void setDefaultEncoding( String defaultEncoding )

· 当primitiveAsString为true时使用的编码转化格式

· 默认值是utf-8

· 如果确认主要写入数据是中文等非ASCII编码字符，建议采用GBK等更短的编码格式

· cache数据写入操作方法

· set方法

· 将数据保存到cache服务器，如果保存成功则返回true

· 如果cache服务器存在同样的key，则替换之

· set有5个重载方法，key和value是必须的参数，还有过期时间，hash码，value是否字符串三个可选参数

· add方法

· 将数据添加到cache服务器,如果保存成功则返回true

· 如果cache服务器存在同样key，则返回false

· add有4个重载方法，key和value是必须的参数，还有过期时间，hash码两个可选参数

· replace方法

· 将数据替换cache服务器中相同的key,如果保存成功则返回true

· 如果cache服务器不存在同样key，则返回false

· replace有4个重载方法，key和value是必须的参数，还有过期时间，hash码两个可选参数

· 建议分析key的规律，如果呈现某种规律有序，则自己构造hash码，提高存储效率

· cache数据读取操作方法

· 使用get方法从cache服务器获取一个数据

· 如果写入时是压缩的或序列化的，则get的返回会自动解压缩及反序列化

· get方法有3个重载方法，key是必须的参数，hash码和value是否字符串是可选参数

· 使用getMulti方法从cache服务器获取一组数据

· get方法的数组实现，输入参数keys是一个key数组

· 返回是一个map

· 通过cache使用计数器

· 使用storeCounter方法初始化一个计数器

· 使用incr方法对计数器增量操作

· 使用decr对计数器减量操作

## 六、总结

1 memcache已经分配的内存不会再主动清理。

2 memcache分配给某个slab的内存页不能再分配给其他slab。

3 flush\_all不能重置memcache分配内存页的格局，只是给所有的item置为过期。

4 memcache最大存储的item(key+value)大小限制为1M，这由page大小1M限制

5 由于memcache的分布式是客户端程序通过hash算法得到的key取模来实现，不同的语言可能会采用不同的hash算法，同样的客户端程序也有可能使用相异的方法，因此在多语言、多模块共用同一组memcached服务时，一定要注意在客户端选择相同的hash算法

6 启动memcached时可以通过-M参数禁止LRU替换，在内存用尽时add和set会返回失败

7 memcached启动时指定的是数据存储量，没有包括本身占用的内存、以及为了保存数据而设置的管理空间。因此它占用的内存量会多于启动时指定的内存分配量，这点需要注意。

8 memcache存储的时候对key的长度有限制，php和C的最大长度都是250

9 MemCache中可以保存的item数据量是没有限制的，只要内存足够

10 MemCache单进程在32位机中最大使用内存为2G，这个之前的文章提了多次了，64位机则没有限制

11 Key最大为250个字节，超过该长度无法存储

12 单个item最大数据是1MB，超过1MB的数据不予存储

13 MemCache服务端是不安全的，比如已知某个MemCache节点，可以直接telnet过去，并通过flush\_all让已经存在的键值对立即失效，所以MemCache服务器最好配置到内网环境，通过防火墙制定可访问客户端

14 不能够遍历MemCache中所有的item，因为这个操作的速度相对缓慢且会阻塞其他的操作

15 MemCached的高性能源自于两阶段哈希结构：第一阶段在客户端，通过Hash算法根据Key值算出一个节点；第二阶段在服务端，通过一个内部的Hash算法，查找真正的item并返回给客户端。从实现的角度看，MemCache是一个非阻塞的、基于事件的服务器程序

16 MemCache设置添加某一个Key值的时候，传入expiry为0表示这个Key值永久有效，这个Key值也会在30天之后失效，见memcache.c的源代码

## 七、Redis和memcaceh对比

1、Redis和Memcache都是将数据存放在内存中，都是内存数据库。不过memcache还可用于缓存其他东西，例如图片、视频等等；

2、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；

3、[虚拟内存](https://www.baidu.com/s?wd=%E8%99%9A%E6%8B%9F%E5%86%85%E5%AD%98&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHcvrjTdrH00T1Y4rjDYnju-njb1nycLmW-b0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHbzPHfYP10LPjDznWnkn1T3Ps" \t "https://www.cnblogs.com/timothy-lai/p/_blank)--Redis当物理内存用完时，可以将一些很久没用到的value 交换到磁盘；

4、过期策略--memcache在set时就指定，例如set key1 0 0 8,即永不过期。Redis可以通过例如expire 设定，例如expire name 10；

5、分布式--设定memcache集群，利用magent做一主多从;redis可以做一主多从。都可以一主一从；

6、存储数据安全--memcache挂掉后，数据没了；redis可以定期保存到磁盘（持久化）；

7、[灾难恢复](https://www.baidu.com/s?wd=%E7%81%BE%E9%9A%BE%E6%81%A2%E5%A4%8D&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHcvrjTdrH00T1Y4rjDYnju-njb1nycLmW-b0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHbzPHfYP10LPjDznWnkn1T3Ps" \t "https://www.cnblogs.com/timothy-lai/p/_blank)--memcache挂掉后，数据不可恢复; redis数据丢失后可以通过aof恢复；

8、Redis支持数据的备份，即master-slave模式的数据备份；

9、应用场景不一样：Redis出来作为NoSQL数据库使用外，还能用做[消息队列](http://www.cnblogs.com/timothy-lai/p/5711840.html" \t "https://www.cnblogs.com/timothy-lai/p/_blank)、数据堆栈和数据缓存等；Memcached适合于缓存SQL语句、数据集、用户临时性数据、延迟查询数据和session等。