私塾在线《高级软件架构师实战培训阶段一》

——跟着cc学架构系列精品教程

本部分课程概览

- 根据实际的应用需要，学习要用到的Memcached的知识，以快速上手、理解并掌握Memcached
- 一： Memcached简介、安装和基本使用
  包括：是什么、能干什么、特点；通过源码安装、基本的启动、运行、关闭等
- 二：Memcached的基本原理和操作命令
  包括：基本原理、操作命令set、add、replace、append、prepend、cas 、get、gets、delete、incr、decr、stats、stats sizes、stats settings、stats items、stats slabs、flush all、version等的功能和使用
- 三：理解Memcached的数据存储方式和数据过期方式
  包括：Slab Allocator内存管理方式、新建Item分配内存的过程、这种存储方式的缺点、理解Memcached的数据过期方式

本部分课程概览

- 四： Memcached的Java客户端编程
  包括：了解常见的Java客户端、理解和掌握官方的Memcached的Java客户端API、把jar包添加到本地Maven仓库、Java客户端基本写法、Memcached和Spring集成

- 五： Memcached的分布式
  包括：Memcached的分布式方式、根据余数计算分散的方式、一致性Hash算法

- 六： Memcached的内存调优以及使用的限制和建议
  包括：内存调优建议、使用Memcached-tool来辅助调优、使用Memcached的一些限制、使用Memcached的一些建议

Memcached简介

- Memcached是什么

  Memcached是一款开源的、高性能的、分布式的内存对象缓存系统

- Memcached能干什么

  最主要的功能就是：在内存中缓存数据，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特点

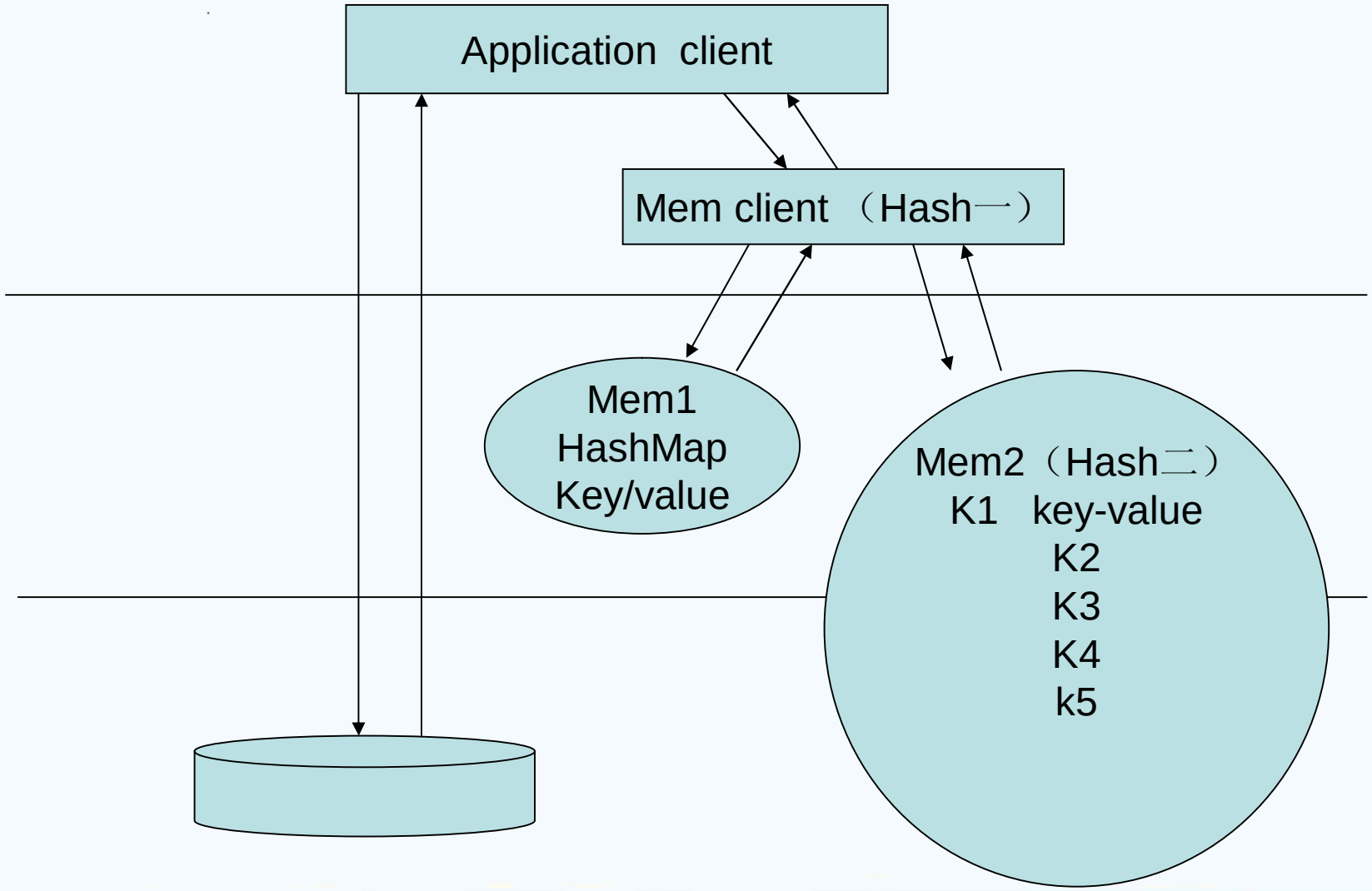
  在内存中以键/值对存储，性能好

  协议简单（基于文本行），功能强大

  基于libevent的事件处理，无阻塞通信，对内存读写速度非常快

  基于客户端的分布式，服务端多个Memcached之间不互相通信

  服务端以守护进程运行，客户端可以用任何语言来编写



Memcached安装

- 下载并安装Memcached服务器端

1：需要安装libevent，去http://libevent.org/下载 ，然后依次：

.configure --prefix=指定安装的路径 ，make ， make install

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，具有很高的性能。

2：去http://memcached.org/ 下载最新的源码包

（1）解压包，注意下载的是 tar.tar的包，不是tar.gz ，所以解压的时候，去掉z，也就是tar vxf 就可以了。

（2）进入到解压的文件夹里面

（3）第一步：需要指定libevent的路径

./configure --prefix=/usr/common/memcached --with-libevent=/usr/common/libevent/

（4）第二步：make

（5）第三步：make install

Memcached的基本使用

- 启动Memcached服务端

```
./memcached -d -m 10 -u root -l 192.168.1.106 -p 2222 -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，这里是10MB
-u是运行Memcache的用户，这里是root
-l是监听的服务器IP地址，这里指定了服务器的IP地址192.168.1.106
-p是监听的端口，这里设置了2222，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，这里设置了256
-P是设置保存Memcache的pid文件，这里是保存在 /tmp/memcached.pid

常用的还有几个需要了解：

-f 块大小增长因子，默认是1.25
-n 最小分配空间， key+value+flags 默认是 48byte
-l 每个slab page的大小
-v/-vv 详细显示工作时各种参数

- 关闭Memcached，先用 ps aux|grep memcached找到进程号，然后kill掉

Memcached的基本原理

- Memcached基本的工作原理

  Memcached是以守候程序的方式运行于一个或者多个服务器，随时等待客户端的链接，通过启动memcached服务器端，配置相应的监听IP、端口内存大小等参数，客户端可通过指定的服务器端IP，将数据以key-value的方式存储

- Memcached的两阶段哈希

  客户端存取数据时，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），进行真正的数据（item）存取

- 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

Memcached的操作命令-1

- 标准协议：Memcached所有的标准协议包含在对item执行命令过程中，一个item包含两行：

  第一行：Key Flags ExpirationTime Bytes

  Key：Key 用于查找缓存值

  Flags：一个32位的标志值，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长（以秒为单位，0表示永远）

  Bytes：在缓存中存储的字节数

  第二行：Value：存储的值（始终位于第二行）

- noreply：可以在命令的第一行后面加入noreply，以避免在处理交互命令的时候，等待服务端的返回
- 向Memcached写入值

  命令有：set、add、replace、append、prepend、cas

1：set：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

2：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，服务器响应 NOT_STORED

3：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，服务器响应NOT_STORED

Memcached的操作命令-2

4：append：是在现有缓存数据后面新增数据。如果key不存在，服务器响应NOT_STORED

5：prepend：是在现有缓存数据前面新增数据。如果key不存在，服务器响应NOT_STORED

6：cas（Check And Set ）：检查和更新，只有从你读取数据后，别人没有更新这个数据，才能够正确保存。就是版本控制，通常和gets配合使用

- 获取数据的命令有：get 、 gets
  get用来获取数据，gets获取的是数据+版本号

- 删除数据的命令：delete

- incr/decr命令：如果缓存数据中存储的是数字形式的字符串，则可以使用incr/decr 对数据进行递增和递减操作，操作后的值不会为负数

Memcached的操作命令-3

n stats命令：查询服务器的运行状态和其他内部数据，包含如下这些：

1：pid ：服务器进程 ID
2：uptime ：服务器运行时间，单位秒
3：time：服务器当前的 UNIX 时间
4：version ：服务器的版本号
5：libevent：libevent的版本
6：pointer_size ：服务器操作系统位数
7：rusage_user：该进程累计的用户时间
8：rusage_system：该进程累计的系统时间
9：curr_connections ：当前连接数
10：total_connections ：服务器启动后总连接数
11：connection_structures ：服务器分配的连接结构的数量
12：reserved_fds：内部使用的misc fds 数量
13：cmd_get ：获取请求数量
14：get_hits ：获取成功的总次数，命中次数
15：get_misses ：获取失败的总次数
16：cmd_set ：存储请求数量
17：cmd_flush ：flush请求的数量
18：cmd_touch：touch请求的数量

分析CPU占用是否高

分析连接数是否过多

分析命中率

Memcached的操作命令-4

19：delete_misses ：删除失败次数
20：delete_hits ：删除命中
21：incr_misses ：递增失败次数
22：incr_hits ：递增命中次数
23：decr_misses ：递减命中次数
24：decr_hits ：递减失败次数
25：cas_misses ：Cas 原子设置操作失败次数
26：cas_hits ：Cas 命中次数
27：cas_badval ：Cas 操作找到 key，但是版本过期，没有设置成功
28：touch_hits：touch命中次数
29：touch_misses：touch失败次数
30：auth_cmds ：认证次数（包括成功和失败）
31：auth_errors ：认证失败次数
32：bytes ：已用缓存空间
33：bytes_read ：总共获取的数据量
34：bytes_written ：总写入数量数
35：limit_maxbytes ：总允许写入的数据量，和分配的内存有关
36：accepting_conns：允许的总连接数

分析字节数流量

Memcached的操作命令-5

37：listen_disabled_num ：监听失败的次数
38：threads：需要的工作线程数
39：hash_bytes：当前使用的Hash table容量大小
40：hash_is_expanding：指定Hash table是否自动增长
41：malloc_fails：malloc内存分配失败的次数
42：curr_items ：当前缓存 item 数量
43：total_items ：从服务启动后，总的存储缓存 item 数量
44：evictions ：通过删除 item 释放内存的次数

分析对象LRU频率

- 这些数据隐含的几个基本关系：

1：缓存命中率 = get_hits/cmd_get * 100%
2：get_misses的数字加上get_hits应该等于cmd_get

- stats sizes命令：输出所有Item的大小和个数，注意：会锁定服务，暂停处理请求
- flush_all命令：使内存中所有的item失效。加入参数则表示在N秒后失效。这个操作并不会真的释放内存空间，而是标志所有的item为失效
- version命令：查看版本

Memcached的操作命令-6

- stats settings查看设置

maxbytes：最大字节数限制，0无限制
maxconns：允许最大连接数
tcpport：TCP端口
udpport：UDP端口
verbosity：日志0=none,1=som,2=lots
oldest：最老对象过期时间
evictions：on/off，是否禁用LRU
domain_socket：socket的domain
umask：创建Socket时的umask
growth_factor：增长因子
chunk_size：key+value+flags大小
num_threads：线程数，可以通过-t设置，默认4
stat_key_prefix：stats分隔符
detail_enabled：yes/no，显示stats细节信息
reqs_per_event：最大IO吞吐量(每event)
cas_enabled：yes/no，是否启用CAS，-C禁用
tcp_backlog：TCP监控日志
auth_enabled_sasl：yes/no，是否启用SASL验证

Memcached的操作命令-7

- stats items数据项统计
  number：该slab中对象数，不包含过期对象
  age：LRU队列中最老对象的过期时间
  evicted：LRU释放对象数
  evicted_nonzero：设置了非0时间的LRU释放对象数
  evicted_time：最后一次LRU秒数，监控频率
  outofmemory：不能存储对象次数，使用-M会报错
  tailrepairs：修复slabs次数
  reclaimed：使用过期对象空间存储对象次数

Memcached的操作命令-8

- stats slabs区块统计
  chunk_size：chunk大小，byte
  chunks_per_page：每个page的chunk数量
  total_pages：page数量
  total_chunks：chunk数量*page数量
  get_hits：get命中数
  cmd_set：set数
  delete_hits：delete命中数
  incr_hits：incr命中数
  decr_hits：decr命中数
  cas_hits：cas命中数
  cas_badval：cas数据类型错误数
  used_chunks：已被分配的chunk数
  free_chunks：剩余chunk数
  free_chunks_end：分完page浪费chunk数
  mem_requested：请求存储的字节数
  active_slabs：slab数量
  total_malloced：总内存数量
  被浪费内存数=((total_chunks或者used_chunks) * chunk_size) - mem_requested，如果太大，需要调整factor

理解Memcached的数据存储方式-1

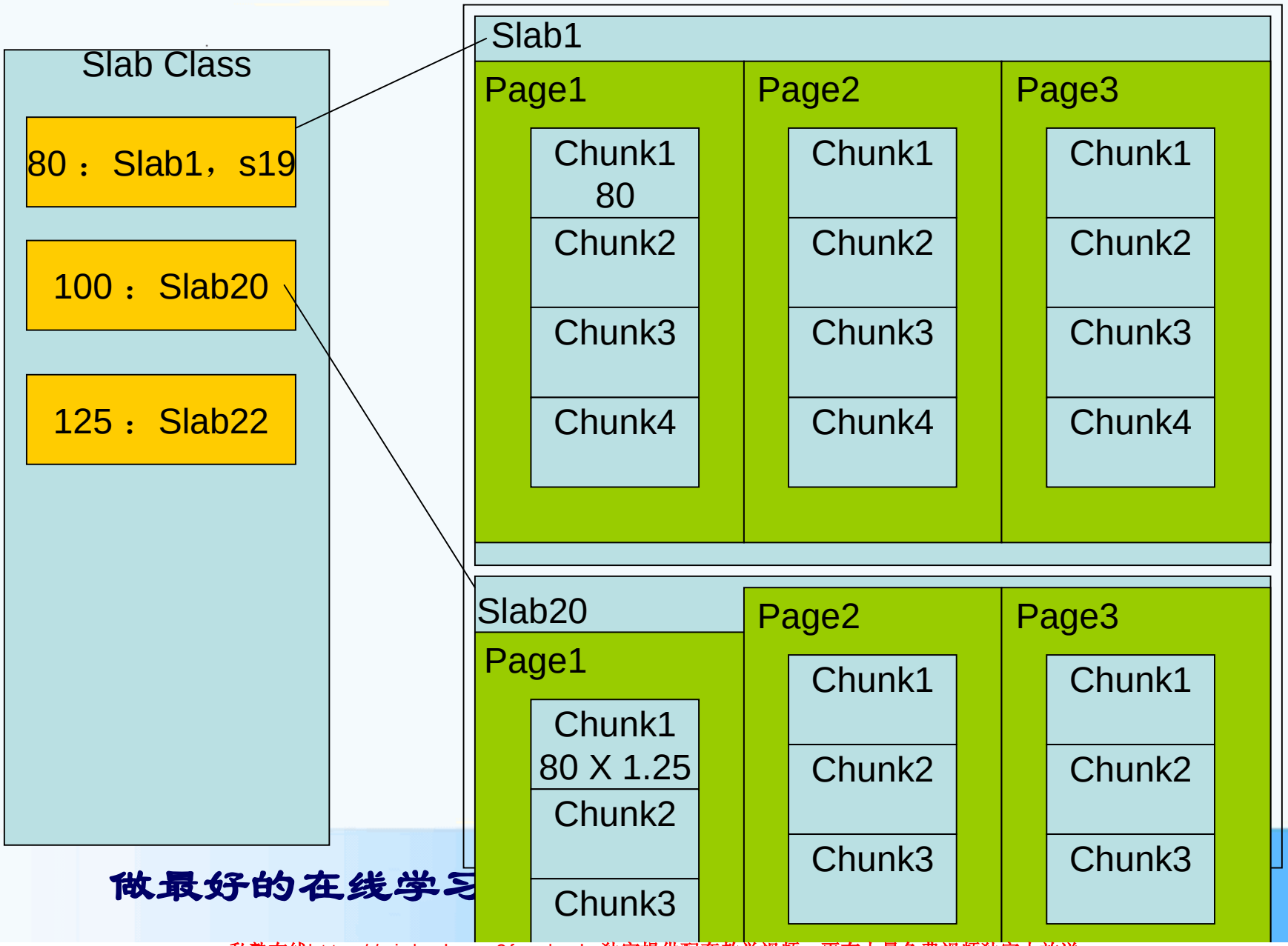
- Memcached的数据存储方式被称为Slab Allocator，其基本方式是：

1：先把内存分成很多个Slab，这个大小是预先规定好的，以解决内存碎片的问题。分配给Slab的内存空间被称为Page，默认是1M。一个Slab下可以有多个Page。

2：然后把一个Page分成很多个chunk块，chunk块是用于缓存记录的空间。Chunk的大小是先有一个基本值，然后根据增长因子来增大。

3：slab class：内存区类别（48byte-1M），每个类别有一个slab classId

4：Memcached里面保存着slab内空闲的chunk列表，当收到要保存的item的时候，它会根据item的大小，去选择一个最合适的slab，然后找到空闲的chunk，把数据存放进去



理解Memcached的数据存储方式-2

- 新建Item分配内存过程

1：快速定位slab classid，先计算Item长度

key键长＋flag+suffix（16字节）+value值长＋结构大小(32字节)，如90byte

如果>1MB，无法存储丢弃

取最小冗余的slab class

如：有48,96,120，存90会选择96

2：按顺序寻找可用chunk

（1）slot：检查slab回收空间slot里是否有剩余chunk

delete：delete时标记到slot

exptime：get时检查的过期对象标记到slot

（2）end_page_ptr：检查page中是否有剩余chunk

（3）memory：内存还有剩余空间可以用于开辟新的slab

（4）LRU

- Memcached的数据存储方式的缺点

由于chunk的大小是预先分配好的特定长度，因此如果数据不能完全填满chunk，那么剩余的空间就浪费了

理解Memcached的数据过期方式

- Lazy Expiration（延迟/惰性 过期）

  Memcached不会监控记录是否过期，而是在外部来获取数据的时候，才检查记录的时间戳，因此称为Lazy Expiration。

- LRU（Least Recently Used 最近最少使用）

  当空间不足的时候，Memcached会优先使用已经过期的数据空间，如果还不够，那么就会把最近最少使用的对象的空间释放出来使用。

- 懒惰删除机制

  删除item对象时，不释放内存，作删除标记，指针放入slot回收插槽，下次分配的时候直接使用

- 要特别注意：Memcached的LRU不是全局的，而是针对slab的，可以说是区域性的

Memcached的Java客户端-1

- Memcached的守护进程是用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的Java客户端可以查看http://code.google.com/p/memcached/wiki/Clients，大致有：
  Java memcached client/danga
  spymemcached
  Xmemcached
- 官方的Memcached的Java客户端API，主要提供的调用类是SockIOPool和MemCachedClient
- SockIOPool

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：127.0.0.1:1111）

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，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。

Memcached的Java客户端-2

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）

ublic 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

Memcached的Java客户端-3

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服务时使用一致性hash方法

public void initialize()

设置完pool参数后最后调用该方法，启动pool。

Memcached的Java客户端-4

- 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

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

add方法 ： 将数据添加到cache服务器,保存成功则返回true。如果存在同样key，则返回false

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。

get方法 ：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。

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

Memcached的Java客户端-5

gets方法

gets除了会返回缓存值外，还会返回当前缓存的版本号，一般是用于协同CAS完成原子操作使用

getMultiArray方法 ：返回缓存的数组

cas方法

原子设置缓存操作，通过版本号casUnique保证设置的唯一性，如果发现服务器的缓存版本与传入的不同，则放弃设置缓存，返回false

storeCounter方法 ：

初始化一个计数器

getCounter方法 ：

获取当前的计数器值

incr方法：

对计数器增量操作

decr方法：

对计数器减量操作

addOrIncr，addOrDecr：key存在则与incr和decr相同，不存在则相当于storeCounter

Memcached的Java客户端-6

- 在Maven工程中使用的时候

1：先要把jar包添加到本地仓库，方法如下：

```
mvn install:install-file -Dfile=java_memcached-release_2.6.6.jar -
DgroupId=com.danga -DartifactId=memcached -Dversion=2.6.6 -
Dpackaging=jar -DgeneratePom=true
```

2：然后：在pom中添加lib依赖：

```
<dependency>
    <groupId>com.danga</groupId>
    <artifactId>memcached</artifactId>
    <version>2.6.6</version>
</dependency>
```

Memcached的Java客户端-7

```
public class CacheHelper {
    private static MemCachedClient mcc = new MemCachedClient();
    private CacheHelper() {     }
    static {
        String[] servers = { "192.168.1.106:2222" };
        Integer[] weights = { 2 };
        SockIOPool pool = SockIOPool.getInstance();
        pool.setServers(servers);
        pool.setWeights(weights);
        pool.setInitConn(5);
        pool.setMinConn(5);
        pool.setMaxConn(250);
        pool.setMaxIdle(1000 * 60 * 60 * 6);
        pool.setMaintSleep(30);

        pool.setNagle(false);// 禁用nagle算法
        pool.setSocketConnectTO(0);
        pool.setSocketTO(3000);// 3秒超时
        pool.setHashingAlg(3);//设置为一致性hash算法
        pool.initialize();
    }
    public static MemCachedClient getMemCachedClient(){
     return mcc;
    }
}
```

Memcached和Spring集成开发-1

```
<!--memcached 客户端 SocketPool-->
    <bean id="memcachedPool" class="com.danga.MemCached.SockIOPool" factory-
    method="getInstance" init-method="initialize" destroy-method="shutDown">
        <constructor-arg><value>neeaMemcachedPool</value></constructor-arg>
            <property name="servers">
                <list>
                    <value>192.168.1.106:2222</value>
                    <value>192.168.1.106:2223</value>
                </list>
            </property>
            <property name="weights">
                    <list>
                      <value>1</value>
                      <value>2</value>
                   </list>
            </property>
            <property name="initConn">
                <value>5</value>
            </property>
            <property name="minConn">
                <value>5</value>
            </property>
```

Memcached和Spring集成开发-2

```
        <property name="maxConn">
            <value>250</value>
        </property>
        <property name="maintSleep">
            <value>30</value>
        </property>
        <property name="nagle">
            <value>false</value>
        </property>
        <property name="maxIdle">
            <value>6000</value>
        </property>
        <property name="socketTO">
            <value>3000</value>
        </property>
    </bean>
    <!--memcached client-->
    <bean id="memCachedClient" class="com.danga.MemCached.MemCachedClient">
        <constructor-arg>
            <value>neeaMemcachedPool</value>
        </constructor-arg>
    </bean>
```

理解Memcached的分布式方式

- Memcached尽管是“分布式”的缓存系统，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
- Memcached的分布式客户端

  客户端可以通过配置SockIOPool的servers参数保存服务器地址列表，通过weight参数配置每台服务器的权重。SockIOPool提供了连接池的服务，可以通过SockIOPool来配置memcahce服务器相关信息，如最大连接数，最小连接数等。

  一个key只能存放在一台Memcache服务器上，是不会在多个服务器上有多份拷贝的，这样的话既可以防止出现刷新不同步的情况，也可以避免磁盘空间的浪费

- Memcached的分布式特点

1：服务器端不关心分布式
2：依靠客户端来实现分布式
3：客户端存储着可以访问到的Memcached服务器列表
4：在客户端用算法来保证，对同样key值的数据，读写都操作同一个服务器

分布式中根据余数计算分散的方式

- 根据余数计算分散

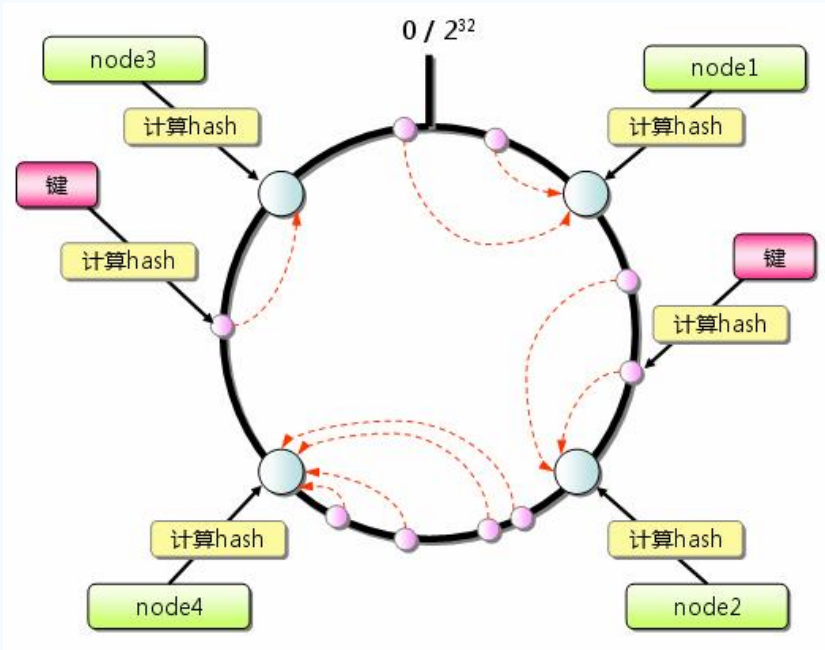
  Memcached的分布式方法简单来说，就是“根据服务器台数的余数进行分散”。求得键的整数哈希值，再除以服务器台数，根据其余数来选择服务器。

- 根据余数计算分散的缺点

  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

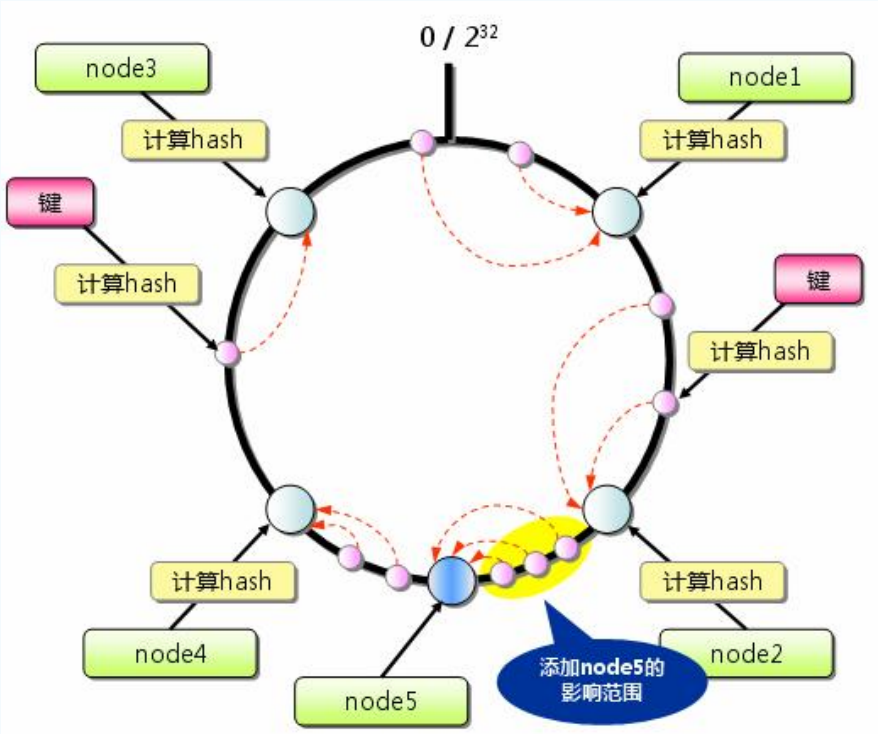
分布式中的一致性Hash算法-1

- Consistent Hashing方式如下：首先求出Memcached服务器（节点）的哈希值，并将其配置到0～2的32次方的圆上。然后用同样的方法求出存储数据的键的哈希值，并映射到圆上。然后从数据映射到的位置开始顺时针查找，将数据保存到找到的第一个服务器上。如果超过232仍然找不到服务器，就会保存到第一台memcached服务器上。



分布式中的一致性Hash算法-2

- 如果要添加一台memcached服务器。余数分布式算法由于保存键的服务器会发生巨大变化而影响缓存的命中率，但Consistent Hashing中，只有在continuum上增加服务器的地点逆时针方向的第一台服务器上的键会受到影响。



分布式中的一致性Hash算法-3

- 从上面的分析可知，Consistent Hashing最大限度地抑制了键的重新分布。而且，有的Consistent Hashing的实现方法还采用了虚拟节点的思想。使用一般的hash函数的话，服务器的映射地点的分布非常不均匀。因此，使用虚拟节点的思想，为每个物理节点（服务器）在continuum上分配100～200个点。这样就能抑制分布不均匀，最大限度地减小服务器增减时的缓存重新分布。

- 使用Consistent Hashing算法的memcached客户端函数库进行测试的结果是，由服务器台数（n）和增加的服务器台数（m）计算增加服务器后的命中率计算公式如下： (1 - n/(n+m)) * 100

Memcached内存调优建议-1

- 调优的目标

  提高内存利用率，减少内存浪费

  提高命中率

- 调优方法：

  1：f参数：

  factor增长因子，默认为1.25，曾经为2，值越小，slab中chunk size差距越小，内存浪费越小。1.25适合缓存几百字节的对象。

  2：n参数：chunk初始值

- 根据数据分布调整factor

  建议：计算一下数据的预期平均长度，调整factor，以获得最恰当的设置。

1：非均匀分布，即数据长度集中在几个区域内，如保存用户Session

2：更极端的状态是等长数据，如定长键值，定长数据，多见于访问、在线统计等

Memcached内存调优建议-2

- 其它常见的问题

1：slab尾部剩余空间
解决办法：规划slab=chunk*n整数倍

2：slab中chunk利用率低：申请的slab只存放了一个Item
解决办法：规划slab=chunk

3：chunk存储Item浪费
如Item是100，存到128字节chunk，就有28字节浪费
解决办法：规划chunk=Item

使用Memcached-tool来辅助调优

- 可以使用Memcached的创造者Brad写的名为memcached-tool的Perl脚本，来方便地获得slab的使用情况，可以从下面的地址获得脚本：
  https://raw.github.com/memcached/memcached/master/scripts/memcached-tool
- 使用方法也极其简单：

```
$ perl memcached-tool.pl 主机名:端口 选项
```

可用的选项有：不写,display，move，dump，stats，settings，sizes

查看slabs使用状况时无需指定选项，默认就是这个，基本列的含义如下：

1：	#	slab class编号
2：	Item_Size	Chunk大小
3：	Max_age	LRU内最旧的记录的生存时间
4：	1MB_pages	分配给Slab的页数
5：	Count	Slab内的记录数
6：	Full?	Slab内是否含有空闲chunk

Memcached的限制和使用建议-1

- 在Memcached中可以保存的item数据量是没有限制的，只要内存足够
- Memcached单进程最大使用内存为2G，要使用更多内存，可以分多个端口开启多个Memcached进程
- Memcached设置Item为最大30天的过期时间,设置为永久的也会在这个时间过期，常量REALTIME_MAXDELTA为60*60*24*30控制
- Memcached缺乏认证以及安全管制，应该将Memcached服务器放置在防火墙后
- 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，当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
- 最大键长为250字节，大于该长度无法存储，常量KEY_MAX_LENGTH 250控制
- 单个item最大数据是1MB，超过1MB数据不予存储，使用常量POWER_BLOCK 1048576进行控制，它是默认的slab大小，修改后需要重新编译
- 最大同时连接数是200，通过 conn_init()中的freetotal进行控制，最大软连接数是1024，通过settings.maxconns=1024 进行控制

Memcached的限制和使用建议-2

- Memcached不实现冗余机制，也不做任何的容错处理

在节点失效的情况下，集群没有必要做任何容错处理。如果发生了节点失效，应对的措施完全取决于用户。节点失效时，下面列出几种方案供您选择：

1：忽略它！在失效节点被恢复或替换之前，还有很多其他节点可以应对节点失效带来的影响

2：把失效的节点从节点列表中移除。做这个操作千万要小心！在余数式哈希算法下，客户端添加或移除节点，会导致所有的缓存数据不可用！因为哈希参照的节点列表变化了，大部分key会因为哈希值的改变而被映射到不同的节点上

3：启动热备节点，接管失效节点所占用的IP。这样可以防止哈希紊乱

4：如果希望添加和移除节点，而不影响原先的哈希结果，可以使用一致性哈希算法

5：两次哈希（reshing）。当客户端存取数据时，如果发现一个节点down了，就再做一次哈希（哈希算法与前一次不同），重新选择另一个节点（需要注意的时，客户端并没有把down的节点从节点列表中移除，下次还是有可能先哈希到它）。如果某个节点时好时坏，两次哈希的方法就有风险了，好的节点和坏的节点上都可能存在脏数据

Memcached的限制和使用建议-3

- 通常使用Memcached的目的是，通过缓存数据库查询结果，减少数据库访问次数；还有就是缓存热点数据，以提高Web应用的速度、提高可扩展性，比如：

1：缓存简单的查询结果：查询缓存存储了给定查询语句对应的整个结果集，最合适缓存那些经常被用到，但不会改变的SQL语句对应查询到的结果集，比如载入特定的过滤内容

2：缓存简单的基于行的查询结果

3：缓存的不只是SQL数据，可以缓存常用的热点数据，比如页面，以节省CPU时间

- 使用分层的缓存

Memcached可以高速处理大量的缓存数据，但是还是要根据系统的情况考虑维护多层的缓存结构。除了Memcached缓存之外，还可以通过本地缓存（如ehcache、oscache等）建立起多级缓存。

例如，可以采用本地缓存缓存一些基本数据，例如少量但访问频繁的数据（如产品分类，连接信息，服务器状态变量，应用配置变量等），缓存这些数据并让他们尽可能的接近处理器是有意义的 ，这样可以帮助减少生成页面的时间，并且在 memcached 失效的情况下可以增加可靠性。

Memcached的限制和使用建议-4

- 特别注意：当数据更新时需要更新缓存

- 预热你的缓存

  如果有一个很高访问率的站点，一开始缓存是空的，然后一大群人点击站点，在填充缓存的过程中，数据库可能会承受不住压力。

  解决：可以先把常用的需要缓存的数据想办法填充到Memcached里，比如：可以写一些脚本来缓存通用的页面；也可以写一个命令行工具来填充缓存

- Memcached的典型适用场景

  1：分布式应用

  2：数据库前段缓存

  3：服务器间数据共享

  4：变化频繁，查询频繁的数据，但是不一定写入数据库，比如：用户在线状态

  5：变化不频繁，查询频繁，不管是否入库，都比较适合使用

Memcached的限制和使用建议-5

- 不适合使用Memcached的场景

1：变化频繁，一变化就要入库类的应用，比如股票，金融

2：那些不需要“分布”的，不需要共享的，或者干脆规模小到只有一台服务器的应用，memcached不会带来任何好处，相反还会拖慢系统效率，因为网络连接同样需要资源

3：缓存对象的大小大于1MB

4：key的长度大于250字符

5：虚拟主机不让运行memcached服务

如果应用本身托管在低端的虚拟私有服务器上，像vmware，xen这类虚拟化技术并不适合运行memcached。Memcached需要接管和控制大块的内存，如果memcached管理的内存被OS交换出去，memcached的性能将大打折扣。

6：应用运行在不安全的环境中

Memcached为提供任何安全策略，仅仅通过telnet就可以访问到memcached。如果应用运行在共享的系统上，需要着重考虑安全问题。

7：业务本身需要的是持久化数据或者说需要的应该是database

Memcached的批量导入导出

- 通常，不应该对Memcached中的item进行批量导入导出

  因为Memcached是一个非阻塞的服务器。任何可能导致memcached暂停或瞬时拒绝服务的操作都应该值得深思熟虑。想象看，如果缓存数据在导出导入之间发生了变化，您就需要处理脏数据了；如果缓存数据在导出导入之间过期了，您又怎么处理这些数据呢？

  不过在一个场景倒是很有用。如果您有大量的从不变化 的数据，并且希望缓存很快热（warm）起来，批量导入缓存数据是很有帮助的。

- 如果确实需要把memcached中的item批量导出导入，怎么办？

  在某些场景下，确实需要批量导出和导入，比如处理“惊群”问题（ 节点都失效了，反复的查询让数据库不堪重负），或者存在优化不好的查询等。

  一种可行的解决方案是：使用MogileFS（或者CouchDB等类似的软件）来存储item，然后把item计算出来并dump到磁盘上。MogileFS可以很方便地覆写item，并提供快速地访问。甚至可以把MogileFS中的item缓存在memcached中，这样可以加快读取速度。MogileFS+Memcached的组合可以加快缓存不命中时的响应速度，提高应用的可用性。