我对于Memcached的接触，还是在去年看了CSDN的一系列国外大型网站架构设计而开始的。最初的时候只是简单的封装了Memcached Java版的客户端，主要是对于配置的简化以及Memcached多点备份作了一些工作，然后就作为ASF的组件一部分提供给其他Team使用。其实看过Memcached Java客户端代码的人就会了解其实客户端的事情很简单，就是要有一套高性能的Socket通信框架以及对Memcached的私有协议实现的接口，自己去做这些事情也是很简单的，不过既然有可以满足自己需求的开源部分，那么就去实现自己需要的但没有实现的。这里我用的是Whalin的客户端版本，这里为什么还要提出来讲这个，后面会提到。

       在对Java客户端作了简单封装和扩展以后，由于其他Team使用的没有什么特殊需求，也就没有再去做太多的修改，直到最近自己的服务集成平台需要做服务访问控制，才重新丰富了Cache组件，也就是这个过程中对于Memcached的一些特性和小的细节有了一些新的认识。

       作为服务集成平台需要对服务有所监控，包括访问频率控制以及访问次数控制。频率控制其实很类似于硬件方面的频率控制，例如硬件可以对IP的高频率访问视为攻击，列入黑名单。而作为服务的访问，对于服务访问者的控制其实涉及到了业务参数，那么硬件就不是很适合去做这方面的控制，为此我也考虑了很久，最开始打算在Apache上做一个模块控制，但是最后觉得还是放在后面的业务框架上做这件事情。当然后面我说说的方案可能并不好，但是也算是一种想法。要把频繁的访问数据记录下来同时分析，那么数据库肯定是不行的，最简单的方式就是采用Cache，又因为是集群范围内的控制，那么集中式Cache就非Memcached莫数了（分布式的Cache传播本身损耗太大，集中式Cache本来的最大缺点就是单点，但作简单的备份操作就可以基本解决此类问题）。

       作为解决这个问题的方法来说只需要实现两部分工作：访问计数器，定时任务。定时任务在我做日志分析框架的时候都是采用了Jdk5的Concurrent包里面的ScheduledExecutorService，这个作简单的循环任务足够用了，同时也是有很好的多线程异步支持，复杂一点么用Quartz。计数器就要靠Memcached来实现了，本来一般的Cache最大的问题就是高并发下的事务保证，如果采用Get+Set来完成计数的话，那么高并发下计数器就会出现读写不一致性的问题，幸好Memcached提供了计数累加功能，让这种累加动作能够在服务端一次做好，服务端控制并发写入，保证数据的一致性。

下面就看看以下几个方法：

boolean storeCounter(String key, long count)：存储key的计数器，值为count。

long getCounter(String key)：获取key的计数器，如果不存在返回-1。

long addOrDecr(String key, long decr)：计数器值减去decr，如果计数器不存在，保存decr作为计数器值

long addOrIncr(String key, long inc)：计数器值增加inc，如果计数器不存在，保存inc作为计数器值

long decr(String key, long decr)：与addOrDecr不同的是在计数器不存在的时候不保存任何值，返回-1

long incr(String key, long inc) ：与addOrIncr不同的是在计数器不存在的时候不保存任何值，返回-1

这里需要说明几点：

storeCounter和普通的set方法不同，如果通过set方式置入key:value的话，getCounter等其他四个方法都认为技术器不存在。所以Counter的存储方式是和普通内容存储不同的。

在不同的场景要慎用addOrXXXX和XXXX的方法，两者还是有比较大的区别的。

计数器没有提供移除特殊方法，使用delete方法可以移除计数器，但是频繁的delete和addOrXXXX有时候会出现一些奇怪的问题（例如同名的计数器就没有办法再次被创建，不过这个还需要进一步的去研究一下看看）。一般情况下如果计数器的key不是很多，同时也会被复用，那么可以通过置为0或者减去已经分析过的数量来复位。

       有上面的一套计数器机制就可以很方便的实现Memcached的计数功能，但是又一个问题出现了，如何让定时任务去遍历计数器，分析计数器是否到了阀值，触发创建黑名单记录的工作。早先我同事希望我能够提供封装好的keySet接口，但是我自己觉得其实作为Cache来说简单就是最重要的，Cache不需要去遍历。首先使用Cache的角色就应该知道Key，然后去Cache里面找，找不到就去后台例如DB里面去搜索，然后将搜索的结果在考虑更新到Cache里面，这样才是最高效并且最可靠的，Cache靠不住阿，随时都可能会丢失或者崩溃，因此作为类似于一级缓存或者这类数据完整性要求不高，性能要求很高的场景使用最合适。当时就没有提供这样的接口，直到今天自己需要了，才考虑如何去做这件事情。

       开始考虑是否能够将key都记录在另外的Cache中或者是Memcached中，首先在高并发下更新操作就是一大问题，再者Memcached的内存分配回收机制以及Value的大小限制都不能满足这样的需求，如果使用数据库，那么频繁更新操作势必不可行，采用异步缓存刷新又有一个时间间隔期，同时更新也不是很方便。最后考虑如果能够让Memcached实现Keyset那么就是最好的解决方案，网上搜索了一下，找到一种策略，然后自己优化了一下，优化后的代码如下：

    @SuppressWarnings("unchecked")

**public** Set keySet(**int** limit,**boolean** fast)

    {

       Set<String> keys = **new** HashSet<String>();

       Map<String,Integer> dumps = **new** HashMap<String,Integer>();

       Map slabs = getCacheClient().statsItems();

**if** (slabs != **null** && slabs.keySet() != **null**)

       {

           Iterator itemsItr = slabs.keySet().iterator();

**while**(itemsItr.hasNext())

           {

              String server = itemsItr.next().toString();

              Map itemNames = (Map) slabs.get(server);

              Iterator itemNameItr = itemNames.keySet().iterator();

**while**(itemNameItr.hasNext())

              {

                  String itemName = itemNameItr.next().toString();

                  // itemAtt[0] = itemname

                   // itemAtt[1] = number

                   // itemAtt[2] = field

                   String[] itemAtt = itemName.split(":");

**if** (itemAtt[2].startsWith("number"))

                       dumps.put(itemAtt[1], Integer.parseInt(itemAtt[1]));

              }

           }

**if** (!dumps.values().isEmpty())

           {

              Iterator<Integer> dumpIter = dumps.values().iterator();

**while**(dumpIter.hasNext())

              {

**int** dump = dumpIter.next();

                  Map cacheDump = statsCacheDump(dump,limit);

                  Iterator entryIter = cacheDump.values().iterator();

**while** (entryIter.hasNext())

                   {

                       Map items = (Map)entryIter.next();

                       Iterator ks = items.keySet().iterator();

**while**(ks.hasNext())

                       {

                          String k = (String)ks.next();

**try**

                          {

                              k = URLDecoder.decode(k,"UTF-8");

                          }

**catch**(Exception ex)

                          {

                              Logger.error(ex);

                          }

**if** (k != **null** && !k.trim().equals(""))

                          {

**if** (fast)

                                 keys.add(k);

**else**

**if** (containsKey(k))

                                     keys.add(k);

                          }

                       }

                   }

              }

           }

       }

**return** keys;

    }

对于上面代码的了解需要从Memcached内存分配和回收机制开始，以前接触Memcached的时候只是了解，这部分代码写了以后就有些知道怎么回事了。Memcached为了提高内存的分配和回收效率，采用了slab和dump分区的概念。Memcached一大优势就是能够充分利用Memory资源，将同机器或者不同机器的Memcached服务端组合成为对客户端看似统一的存储空间，Memcached可以在一台机器上开多个端口作为服务端多个实例，也可以在多台机器上开多个服务实例，而slab就是Memcached的服务端。下面是我封装后的Cache配置：

<?xml version="1.0" encoding="UTF-8"?>

<memcached>

    <client name="mclient0" compressEnable="true" defaultEncoding="UTF-8" socketpool="pool0">

        <!--errorHandler></errorHandler-->

    </client>

    <client name="mclient1" compressEnable="true" defaultEncoding="UTF-8" socketpool="pool1">

        <!--errorHandler></errorHandler-->

    </client>

    <client name="mclient11" compressEnable="true" defaultEncoding="UTF-8" socketpool="pool11">

        <!--errorHandler></errorHandler-->

    </client>

    <socketpool name="pool0" failover="true" initConn="10" minConn="5" maxConn="250" maintSleep="0"

        nagle="false" socketTO="3000" aliveCheck="true">

        <servers>10.2.225.210:13000,10.2.225.210:13001,10.2.225.210:13002</servers>

    </socketpool>

    <socketpool name="pool1" failover="true" initConn="10" minConn="5" maxConn="250" maintSleep="0"

        nagle="false" socketTO="3000" aliveCheck="true">

        <servers>10.2.225.210:13000</servers>

    </socketpool>

    <socketpool name="pool11" failover="true" initConn="10" minConn="5" maxConn="250" maintSleep="0"

        nagle="false" socketTO="3000" aliveCheck="true">

        <servers>10.2.225.210:13000</servers>

    </socketpool>

    <cluster name="cluster1">

        <memCachedClients>mclient1,mclient11</memCachedClients>

    </cluster>

</memcached>

可以看到其实pool才是最终连接服务端的配置，看看pool0，它会连接10.2.225.210:13000,10.2.225.210:13001,10.2.225.210:13002这些机器和他们的端口，但是对于使用pool0的mclient0来说它仅仅只是知道有一个叫做mclient0的cache可以保存数据。此时slab就有三个：10.2.225.210:13000和10.2.225.210:13001和10.2.225.210:13002。

当一个key:value要被放入到Memcached中，首先Memcached会根据key的hash算法获取到hash值来选择被分配的slab，然后根据value选择适合的dump区。所谓dump区其实就是根据value的大小来将内存按照存储单元内容大小分页。这个是可以配置Memcached的，例如Memcached将slab中的内存划分成4个dump，第一dump区存储0-50k大小的数据，第二dump区存储50-100k的数据，第三dump区存储100-500k的数据,第四dump区存储500-1000K的数据。那么当key:value需要被写入的时候，很容易定位到value所处的dump，分配内存给value。这种分dump模式简化内存管理，加速了内存回收和分配。但是这里需要注意的几点就是，首先当你的应用场景中保存的数据大小离散度很高，那么就不是很适合Memcached的这种分配模式，容易造成浪费，例如第一dump区已经满了，第二第三dump区都还是只有一个数据，那么第二第三dump区不会被回收，第二第三dump区的空间就浪费了。同时Memcached对于value的大小支持到1M,大于1M的内容不适合Memcached存储。其实在Cache的设计中这样的情况发生本来就证明设计有问题，Cache只是加速，一般保存都是较小的id或者小对象，用来验证以及为数据定位作精准细化，而大数据量的内容还是在数据库等存储中。

知道了基本的分配机制以后再回过头来看看代码：

Map slabs = getCacheClient().statsItems();//获取所有的slab

//用来收集所有slab的dump号

**while**(itemsItr.hasNext())

           {

              String server = itemsItr.next().toString();

              Map itemNames = (Map) slabs.get(server);

              Iterator itemNameItr = itemNames.keySet().iterator();

**while**(itemNameItr.hasNext())

              {

                  String itemName = itemNameItr.next().toString();

                  // itemAtt[0] = itemname

                   // itemAtt[1] = number

                   // itemAtt[2] = field

                   String[] itemAtt = itemName.split(":");

// 如果是itemName中是:number来表示，那么证明是一个存储数据的dump，还有一些是age的部分

**if** (itemAtt[2].startsWith("number"))

                   dumps.put(itemAtt[1], Integer.parseInt(itemAtt[1]));

              }

           }

        //根据收集到的dump来获取keys

**if** (!dumps.values().isEmpty())

           {

              Iterator<Integer> dumpIter = dumps.values().iterator();

**while**(dumpIter.hasNext())

              {

**int** dump = dumpIter.next();

// statsCacheDump支持三个参数String[],int,int，第一个参数可以省略，默认填入null，表示从那些slab中获取dump号为第二个参数的keys，如果是null就从当前所有的slab中获取。第二个参数表示dump号，第三个参数表示返回最多多少个结果。

                  Map cacheDump = statsCacheDump(dump,limit);

                  Iterator entryIter = cacheDump.values().iterator();

**while** (entryIter.hasNext())

                   {

                        Map items = (Map)entryIter.next();

                        Iterator ks = items.keySet().iterator();

**while**(ks.hasNext())

                    {

                        String k = (String)ks.next();

**try**

                        {

//这里为什么要作decode，因为其实在我使用的这个java客户端存储的时候，默认会把key都作encoding一次，所以必须要做，不然会出现问题。

                            k = URLDecoder.decode(k,"UTF-8");

                        }

**catch**(Exception ex)

                        {

                            Logger.error(ex);

                        }

**if** (k != **null** && !k.trim().equals(""))

                        {

//这里的fast参数是在方法参数中传入，作用是什么，其实采用这种搜索slab以及dump的方式获取keys会发现返回的可能还有一些已经移除的内容的keys，如果觉得需要准确的keys，就在做一次contains的检查，不过速度就会有一定的影响。

**if** (fast)

                               keys.add(k);

**else**

**if** (containsKey(k))

                                   keys.add(k);

                        }

                    }

                   }

              }

           }

至此，整个keySet的问题解决了，对于即时监控也基本都作好了，这里需要把过程中的两件小事情说一下。

1.    statsCacheDump始终不能用。

刚开始的时候statsCacheDump方法始终报错说连接超时，跟踪到了java客户端代码中发现并不是什么连接超时，只是服务端返回了错误信息，而客户端认为还没有结束一直等待，导致超时。我就顺手给java客户端的开发人员mail了信息求助（代码里面有email）。再仔细看了看出错信息，返回的是不认识该指令的错误，因此就去解压memcached的服务端，看了看它的协议说明，这个Stat方法还是有的，很奇怪，没有办法了，虽然自己对于c不是很懂，但起码大致看懂逻辑还是不难，下载了Memcached的源码一看，发现居然对于StatsCacheDump这个方法调用必须还有一个参数limit，在我手头的客户端代码里面就没有这个参数，所以错误了，本来想扩展一下那个方法，但是那个方法中实现的不是很好，都是private的不容易扩展，这时候居然收到其中一个客户端开发者的回复邮件，说我手头的代码太老了，同时不建议去实现keyset，认为这样比较低效。我去下载了一个新版本，看了看源码果然已经修复了，我就回了邮件表示感谢，同时也和他说明了这么做的原因。因此大家如果要和我一样写上面的代码，就需要它2.0.1的那个版本。这里对那些国外的开源工作者表示敬佩，对于开发者是很负责任的。

2．关于fast那个选项

    这个是我加上去的，做了一下测试，例如我先执行如下代码：

    Cache.set(“key1”,”value1”);

Cache.set(“key2”,”value2”);

Cache.flushAll(null);

Cache.set(“key3”,”value3”);

Cache.set(“key4”,”value4”);

Boolean fast = true;

Set keys = Cache.keySet(fast);

System.out.println(keys);

Fast = false;

keys = Cache.keySet(fast);

System.out.println(keys);

得到的结果为：

Key1,key2,key3,key4

Key3,key4

可以看到其实如果通过StatsCacheDump来获取得到的keys会参杂一些已经失效的keys,只是没有回收，本来尝试获取时间戳来做判断，不过还不如使用containsKey来的有效。

同时这里采用containsKey而不是用get，就是因为counter是不能用get获得的，即使counter存在。

这些就是今天在使用Memcached所收获的，分享一下，如果有一些理解上的偏差也希望能够被指出。