Memcached Client 使用手册

Author: cenwenchu

Email: [wenchu.cenwc@alibaba-inc.com](mailto:wenchu.cenwc@alibaba-inc.com)

Blog：<http://blog.csdn.net/cenwenchu79/>

Project: <http://code.google.com/p/memcache-client-forjava/>

目录

[Cache Client接口定义 2](#_Toc228673882)

[Cache Client依赖及配置 2](#_Toc228673883)

[Cache Client第三方依赖 2](#_Toc228673884)

[Cache Client单客户端配置 3](#_Toc228673885)

[Cache Client集群配置 4](#_Toc228673886)

[Cache Client Demo 5](#_Toc228673887)

[Cache Manager 5](#_Toc228673888)

[使用注意 6](#_Toc228673889)

[附录： 6](#_Toc228673890)

[接口定义说明： 6](#_Toc228673891)

# Cache Client接口定义



图 1 Memcached Cache Client接口类图

ICache和IMemcachedCache接口是Cache的基础接口，定义了基本的Cache操作，详细地说明参看附录，使用方式参看Demo和使用注意。ICacheManager是Cache生命周期管理类，一个应用只需要一个ICacheManager来管理所有的Cache，具体介绍参看Cache Demo中的Cache Manager介绍。

# Cache Client依赖及配置

## Cache Client第三方依赖

commons-logging-1.0.4.jar or high version

log4j-1.2.12.jar or high version

codehaus/woodstox/wstx-asl-3.2.1.jar or high version

codehaus/staxapi/stax-api-1.0.1.jar or high version

2.5.2版本以后还需要caucho/hessian/hessian-3.2.0.jar or high version

Cache Client支持默认（在Classpath中查找Memcached.xml作为客户端配置）和指定配置文件。配置文件是Cache Client的正常运行的基础，系统如果要在运行中使用Cache Client的话，必须在使用前（一般最好是应用启动时）初始化Cache Client组件（读取配置文件，创建连接池等等）。具体使用方式可以参看后面章节中的Demo。

## Cache Client单客户端配置

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

<memcached>

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

<errorHandler>com.alisoft.xplatform.asf.cache.memcached.MemcachedErrorHandler</errorHandler> </client>

<socketpool name="pool0" failover="true" initConn="5" minConn="5" maxConn="250" maintSleep="5000"

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

<servers>10.2.224.36:33001,10.2.224.46:33001</servers>

<weights>3,7</weights>

</socketpool>

</memcached>

1. 创建memcached的标签。
2. 创建 client的标签。

注意：

name 属性是程序中使用Cache的唯一标识。

socketpool 属性将会关联到后面的socketpool配置。

errorHandler 可选，用来处理出错情况。注意在Tag中不要使用空格或者Tab键。

1. 创建socketpool的标签。

注意：

name 属性和client 配置中的socketpool 属性相关联。

maintSleep属性是后台线程管理SocketIO池的检查间隔时间，如果设置为0，则表明不需要后台线程维护SocketIO线程池，默认需要管理。

socketTO 属性是Socket操作超时配置，单位ms。

aliveCheck 属性表示在使用Socket以前是否先检查Socket状态。

1. 创建 servers 标签作为socketPool的子标签.设置memcache服务端实例地址，支持多个地址设置，例如“10.2.224.36:33001” 或 “10.2.224.36:33001, 10.2.224.46:33002”.
2. 创建 weights 标签作为socketPool的子标签(可选),它表明了上面设置的服务器实例的Load权重. 例如 <weights>3,7</weights> 表示30% load 在 10.2.224.36:33001, 70% load 在 10.2.224.46:33001

好了，基础的配置就如上。现在可以直接到后面章节去学习使用Memcache客户端，或者继续看如何配置Cluster。

## Cache Client集群配置

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

<memcached>

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

<errorHandler>com.alisoft.xplatform.asf.cache.memcached.MemcachedErrorHandler</errorHandler>

</client>

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

<errorHandler>com.alisoft.xplatform.asf.cache.memcached.MemcachedErrorHandler</errorHandler>

</client>

<socketpool name="pool0" failover="true" initConn="5" minConn="5" maxConn="250" maintSleep="5000"

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

<servers>10.2.224.36:33001,10.2.224.46:33001</servers>

</socketpool>

<socketpool name="pool0-bck" failover="true" initConn="5" minConn="5" maxConn="250" maintSleep="5000" nagle="false" socketTO="3000" aliveCheck="true">

<servers>10.2.224.36:33002,10.2.224.46:33002</servers>

</socketpool>

<cluster name="cluster1" mode="active">//mode = active,standby

<memCachedClients> mclient0, mclient0-bck</memCachedClients>

</cluster>

</memcached>

Memcache是集中式的Cache，因此它存在单点问题（虽然数据可以分散到多台服务器上，但是还会丢失部分数据）。为了解决单点问题，Memcache客户端支持配置集群。

集群配置很简单. 1.创建cluster标签 2. 创建memCachedClients 标签作为cluster的子标签，然后将客户端配置到memCachedClients 标签中。3.可以配置cluster mode（如果没有设置mode属性，默认采用active）。

集群当前的特性：

1. 集群中多节点软负载均衡。（当前采用简单的Hash算法加取余来分发数据）
2. 数据在多节点上异步冗余存储。（防止数据丢失最基本要求）
3. 节点不可用切换功能。（当根据算法分发到某一失败节点时可以转向到其他可用节点）
4. 节点恢复可用后数据Lazy复制。（当A,B两台机器作为集群的时候，如果A出现了问题，系统会去B获取数据，当A正常以后，如果应用在A中没有拿到数据可以去B获取数据，并且复制到A上，这种方式也是一种lazy的复制。）

# Cache Client Demo

在Google项目中有sample包，内部包含了测试用例和测试所需要的配置文件(配置文件中的具体服务器端口和地址需要自己修改)。

下载地址为：<http://memcache-client-forjava.googlecode.com/files/alisoft-xplatform-asf-cache-sample%282.5version%29.zip>

类IMemcachedCacheTest是Cache功能单元测试类。

类StressCacheTest是Cache压力测试类。

类MemcachedClusterTest是Cache集群测试类。

## Cache Manager

在使用Cache Client的应用中都需要有一个Cache Manager来管理各个Cache Client的生命周期。Cache Manager通过读取配置初始化各个Cache Client，应用通过Cache Manager获取到Cache Client进行数据交互。Cache Manager结束后，Cache Client也就被结束并释放。**Cache Manager也可以动态的重新载入配置文件，实现动态扩容。**

建议一个应用只需要设置一个Cache Manager，在应用启动时将Cache Manager启动，在应用结束时将Cache Manager结束。测试用例代码如下：

**static** ICacheManager<IMemcachedCache> *manager*;**//可以直接设定为静态的单例**

@BeforeClass

**public** **static** **void** setUpBeforeClass() **throws** Exception

{

*manager* = CacheUtil.*getCacheManager*(IMemcachedCache.**class**,

MemcachedCacheManager.**class**.getName());**//manager初始化，可以通过配置来替换**CacheManager实现

*manager*.setConfigFile("memcached1.xml");**//设置Cache Client配置文件**

*manager*.setResponseStatInterval(5\*1000);**//设置Cache响应统计间隔时间，不设置则不进行统计**

*manager*.start();**//Manager启动**

}

@AfterClass

**public** **static** **void** tearDownAfterClass() **throws** Exception

{

*manager*.stop();**//manager结束**

}

通过manager.reload("memcached\_cluster2.xml");可以实现动态扩容，同时支持将memcached\_cluster2.xml为<http://10.2.226.41/sip/memcached_cluster2.xml>，实现远程获取配置。具体使用，请参看集群测试用例中的代码。（请注意测试用例中的sleep代码，这些是系统在关闭和启动时的延时）

# 使用注意

1. 使用Clear方法时，如果立即去获取数据可能会因为回收速度问题导致获取到被删除的数据，需要有部分的延时，采用sleep来避免问题发生。
2. Manager的reload会重新载入配置，但是对于服务端的数据不会删除，因此需要注意新的配置中集群数据相互拷贝问题。同时reload以后对于cache client需要重新获取对象，否则保留的还是原来的cache client，将无法使用。（数据复制采用lazy的方式，如果需要立即全部重新分配，可以采用Manager的ClusterCopy接口实现）
3. 对于get,put联合操作的一些场景，建议采用计数器来实现原子操作。（计数器需要通过getCounter,storeCounter,incr,decr等等接口来操作，不是普通的get,put操作）
4. 可以使用add,replace来满足一些需要对内容是否存储有不同策略的场景。
5. 使用本地Cache和Memcached Cache组合的情况中，在缓存数据到本地以后，如果对此数据作了修改，会让本地缓存失效，但是无法通知到其他应用或者其他服务器的本地缓存。
6. 集群的active和standby两种模式，前者速度可能在某些情况下稍慢（当key的确没有存在于集群任何一节点时，active模式会去尝试两个节点获取数据），但是具有数据恢复功能，后者速度比较快，但是没有数据恢复功能。

# 附录：

## 接口定义说明：

/\*\*

\* Cache统一接口

\* **@author** wenchu.cenwc

\*

\*/

**public** **interface** ICache<K,V>

{

/\*\*

\* 保存数据

\* **@param** key

\* **@param** value

\* **@return**

\*/

**public** V put(K key,V value);

/\*\*

\* 保存有有效期的数据

\* **@param** key

\* **@param** value

\* **@param** 有效期(取的是客户端时间)

\* **@return**

\*/

**public** V put(K key,V value, Date expiry);

/\*\*

\* 保存有有效期的数据

\* **@param** key

\* **@param** value

\* **@param** 设置有效期为距离当前时间后TTL秒。

\* **@return**

\*/

**public** V put(K key,V value, **int** TTL);

/\*\*

\* 获取缓存数据

\* **@param** key

\* **@return**

\*/

**public** V get(K key);

/\*\*

\* 移出缓存数据

\* **@param** key

\* **@return**

\*/

**public** V remove(K key);

/\*\*

\* 删除所有缓存内的数据

\* **@return**

\*/

**public** **boolean** clear();

/\*\*

\* 缓存数据数量（Memcached接口当前不支持）

\* **@return**

\*/

**public** **int** size();

/\*\*

\* 缓存所有的key的集合

\* **@return**

\*/

**public** Set<K> keySet();

/\*\*

\* 缓存的所有value的集合

\* **@return**

\*/

**public** Collection<V> values();

/\*\*

\* 是否包含了指定key的数据

\* **@param** key

\* **@return**

\*/

**public** **boolean** containsKey(K key);

/\*\*

\* 释放Cache占用的资源

\*/

**public** **void** destroy();

}

/\*\*

\* Memcached Cache的接口定义

\* **@author** wenchu.cenwc<wenchu.cenwc@alibaba-inc.com>

\*/

**public** **interface** IMemcachedCache **extends** ICache<String,Object>

{

/\*\*

\* 降低memcache的交互频繁造成的性能损失，因此采用本地cache结合memcache的方式

\* **@param** key

\* **@param** 本地缓存该数据有效秒数

\* **@return**

\*/

**public** Object get(String key,**int** localTTL);

/\*\*

\* 获取多个keys对应的值

\* **@param** keys

\* **@return**

\*/

**public** Object[] getMultiArray(String[] keys);

/\*\*

\* 获取多个keys对应的key&value Entrys

\* **@param** keys

\* **@return**

\*/

**public** Map<String,Object> getMulti(String[] keys);

/\*\*

\* key所对应的是一个计数器，实现增加inc的数量

\* **@param** key

\* **@param** inc

\* **@return**

\*/

**public** **long** incr(String key,**long** inc);

/\*\*

\* key所对应的是一个计数器，实现减少decr的数量

\* **@param** key

\* **@param** decr

\* **@return**

\*/

**public** **long** decr(String key,**long** decr);

/\*\*

\* key所对应的是一个计数器，实现增加inc的数量

\* **@param** key

\* **@param** inc

\* **@return**

\*/

**public** **long** addOrIncr(String key,**long** inc);

/\*\*

\* key所对应的是一个计数器，实现减少decr的数量

\* **@param** key

\* **@param** decr

\* **@return**

\*/

**public** **long** addOrDecr(String key,**long** decr);

/\*\*

\* 存储计数器

\* **@param** key

\* **@param** count

\*/

**public** **void** storeCounter(String key,**long** count);

/\*\*

\* 获取寄存器，-1表示不存在

\* **@param** key

\*/

**public** **long** getCounter(String key);

/\*\*

\* 这个接口返回的Key如果采用fast模式，

\* 那么返回的key可能已经被清除或者失效，但是在内存中还有痕迹，如果是非fast模式，那么就会精确返回，但是效率较低

\* **@param** 是否需要去交验key是否存在

\* **@return**

\*/

**public** Set<String> keySet(**boolean** fast);

/\*\*

\* 统计服务器的Slab的情况

\* **@return**

\*/

**public** MemcacheStatsSlab[] statsSlabs();

/\*\*

\* 统计Memcache使用的情况

\* **@return**

\*/

**public** MemcacheStats[] stats();

/\*\*

\* 统计Items的存储情况

\* **@param** servers

\* **@return**

\*/

@SuppressWarnings("unchecked")

**public** Map statsItems();

/\*\*

\* 统计Cache的响应时间（必需设置statisticsInterval大于0才会开始统计）

\* **@return**

\*/

**public** MemcachedResponse statCacheResponse();

/\*\*

\* 设置统计时间，单位为秒

\* **@param** checkInterval

\*/

**public** **void** setStatisticsInterval(**long** checkInterval);

/\*\*

\* 保存数据,前提是key不存在于memcache中，否则保存不成功

\* **@param** key

\* **@param** value

\* **@return**

\*/

**public** **boolean** add(String key,Object value);

/\*\*

\* 保存有有效期的数据，前提是key不存在于memcache中，否则保存不成功

\* **@param** key

\* **@param** value

\* **@param** 有效期

\* **@return**

\*/

**public** **boolean** add(String key,Object value, Date expiry);

/\*\*

\* 保存数据,前提是key必须存在于memcache中，否则保存不成功

\* **@param** key

\* **@param** value

\* **@return**

\*/

**public** **boolean** replace(String key,Object value);

/\*\*

\* 保存有有效期的数据，前提是key必须存在于memcache中，否则保存不成功

\* **@param** key

\* **@param** value

\* **@param** 有效期

\* **@return**

\*/

**public** **boolean** replace(String key,Object value, Date expiry);

/\*\*

\* 异步存入数据，当前立即返回，稍后存入数据

\* **@param** key

\* **@param** value

\*/

**public** **void** asynPut(String key,Object value);

/\*\*

\* 异步累减计数器，不保证累减成功

\* **@param** key

\* **@param** decr

\*/

**public** **void** asynAddOrDecr(String key,**long** decr);

/\*\*

\* 异步累加计数器，不保证累加成功

\* **@param** key

\* **@param** incr

\*/

**public** **void** asynAddOrIncr(String key,**long** incr);

/\*\*

\* 异步累减计数器，不保证累减成功

\* **@param** key

\* **@param** decr

\*/

**public** **void** asynDecr(String key,**long** decr);

/\*\*

\* 异步累加计数器，不保证累加成功

\* **@param** key

\* **@param** incr

\*/

**public** **void** asynIncr(String key,**long** incr);

/\*\*

\* 异步存储计数器,不保证保存成功

\* **@param** key

\* **@param** count

\*/

**public** **void** asynStoreCounter(String key,**long** count);

}