Memcached中的分布式思想

Memcached号称是高性能的分布式缓存系统。说到分布式，Memcached是值得我们来分析分析的。它的分布式机制和一般的分布式服务系统是不同的，分布式服务系统各节点之间是有通信的，目的是为了保证数据的一致性。但是对于Memcached来说，各个节点之间是没有通信的，也就是说Memcached的集群是一组单点服务，各节点之间是不通信的。而其所有的和分布式有关的地方都是在客户端实现的。关于这一点在Memcached的github一篇文章中[《A Story of Caching》](https://github.com/memcached/memcached/wiki/TutorialCachingStory)是有体现的，有兴趣的可以去看一看这篇文章，通过这篇文章可以对Memcached有一个清晰的认识。

下面我们先通过一个案例来验证一下我们上面的说法。

例一

有两台Memcached服务。我们使用php来向Memcached中添加数据。

代码一

$MEMCACHE\_SERVERS = array(

"192.168.5.111", //memcached1

"192.168.5.102" //memcached2

);

$memcache = new Memcache();

foreach($MEMCACHE\_SERVERS as $server){

$memcache->addServer ( $server ); //将两个台服务器地址都添加进连接池

}

$memcache->set("onmpw", 5);

$memcache->set("jiyi", 4);

上面php代码执行以后，按照分布式系统原理来说应该两台服务器中都有key为onmpw和jiyi的数据，但是事实是这样吗，我们可以分别登录到两台服务上进行查看。

Connect 5.111

# telnet 192.168.5.111 11211

Trying 192.168.5.111...

Connected to 192.168.5.111 (192.168.5.111).

Escape character is '^]'.

get jiyi

VALUE jiyi 768 1

4

END

get onmpw //在5.111上并没有onmpw这条数据

END

Connect 5.102

Trying 192.168.5.102...

Connected to 192.168.5.102 (192.168.5.102).

Escape character is '^]'.

get onmpw

VALUE onmpw 768 1

5

END

get jiyi //在5.102上同样是没有jiyi这条数据

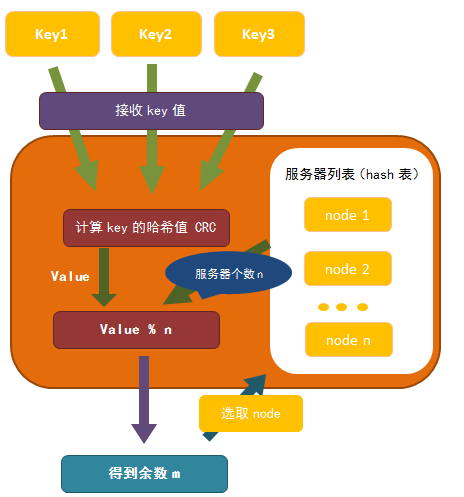
END

由此可见在Memcached的服务端并不去同步集群中的数据。

那数据又是怎么将不同的键值对分发到不同的节点上去的呢，这就是客户端所要实现的分布式了。在这里会涉及到一种算法——hash算法。这里面涉及到两种hash算法：一种是取余数计算分散，一种是一致性hash算法(Consistent Hashing)。一致性hash算法在 Consistent Hashing算法入门及php实现 这篇文章中有比较系统的表述。而对于取余数方式的表述却有欠详细，所以在这里对此做一个稍详细的表述，以弥补那篇文章的不足。

一般的hash算法——根据余数计算分散

首先是根据集群服务的节点数创建一个哈希表，然后根据键名计算出键名的整数哈希值，用该哈希值对节点数取余，最后根据余数在哈希表中取出节点。



其中在计算key的哈希值的时候使用的是CRC（循环冗余校验）。下面使用php简化的代码来进行说明

代码二

$nodes = array('node1','node2','node3'); //nodes代表服务器节点

$keys = array('onmpw', 'jiyi', 'onmpw\_key', 'jiyi\_key', 'www','www\_key'); //这是key值

foreach( $keys as $key) {

$crc = crc32($key); // CRC値

$mod = $crc % ( count($nodes) );

$server = $nodes[ $mod ]; // 根据余数选择服务器

printf("%s => %s\n", $key, $server);

}

首先求得key的crc值，然后和node的个数取余数。上面代码的执行结果如下

onmpw => node2  
jiyi => node2  
onmpw\_key => node1  
jiyi\_key => node2  
www => node1  
www\_key => node3

根据上面的结果，我们看到onmpw、jiyi和jiyi\_key分散到node2，onmpw\_key和www分散到node1，www分散到node3。

php客户端分布式实现

在php实现的Memcache的客户端是支持余数分散这种算法的。下面我们来看一下其源码中涉及该算法的部分。这里使用的php-memcache版本为memcache-3.0.6

mmc\_t \*mmc\_standard\_find\_server(void \*s, const char \*key, unsigned int key\_len TSRMLS\_DC) {

mmc\_standard\_state\_t \*state = s;

if (state->num\_servers > 1) {

/\* "new-style" hash \*/

//计算key值的哈希值

unsigned int hash = (mmc\_hash(state->hash, key, key\_len) >> 16) & 0x7fff;

//用得到的哈希值对节点数取余

return state->buckets[(hash ? hash : 1) % state->num\_buckets];

}

return state->buckets[0];

}

我们看到，计算hash值的方式是使用mmc\_hash()函数得到一个值，然后将该值向右移动16位，最后再和0x7fff做‘与’运算。

这里我们在看mmc\_hash()是如何计算key的值的。

#define mmc\_hash(hash, key, key\_len) ((hash)->finish((hash)->combine((hash)->init(), (key), (key\_len))))

太坑了，这算神马函数？各位先不要着急，这里无非是一个宏定义，可以简单的认为mmc\_hash(hash, key, key\_len)其实就是一个值，这个值是什么呢？是由((hash)->finish((hash)->combine((hash)->init(), (key), (key\_len))))来得到的。其实主要的部分还是finish()、combine()和init()函数。我们来看这三个函数的实现。

上面三个函数由以下结构体定义

typedef struct mmc\_hash\_function {

mmc\_hash\_function\_init init;

mmc\_hash\_function\_combine combine;

mmc\_hash\_function\_finish finish;

} mmc\_hash\_function\_t;

同时对于mmc\_hash\_function\_t做了如下的映射

extern mmc\_hash\_function\_t mmc\_hash\_crc32;

也就是说这些函数的实现是在mmc\_hash\_crc32中实现的。继续追寻，发现下面的代码

mmc\_hash\_function\_t mmc\_hash\_crc32 = {

mmc\_hash\_crc32\_init,

mmc\_hash\_crc32\_combine,

mmc\_hash\_crc32\_finish

};

追根寻底，其实init()、combine()和finish()函数分别对应mmc\_hash\_crc32\_init()、mmc\_hash\_crc32\_combine()和mmc\_hash\_crc32\_finish()。下面看这些函数的实现

static unsigned int mmc\_hash\_crc32\_init() { return ~0; }

static unsigned int mmc\_hash\_crc32\_finish(unsigned int seed) { return ~seed; }

static unsigned int mmc\_hash\_crc32\_combine(unsigned int seed, const void \*key, unsigned int key\_len)

{

const char \*p = (const char \*)key, \*end = p + key\_len;

while (p < end) {

CRC32(seed, \*(p++));

}

return seed;

}

在上面的代码中我们是不是看到了熟悉的身影。底层的实现也是用的crc32函数（该函数在php源码的ext/standard/crc32.h中有定义，有兴趣的可以去研究一下）。顺便多说一句，在mmc\_hash\_crc32\_init()函数中返回的值是~0，它的值为-1。而’~’表示是按位取反。在定义函数的返回值的类型时是unsigned（无符号）的。所以-1的二进制表示为11111111111111111111111111111111（总共32位）。

我们看到，使用余数计算的实现方法很简单，而且其数据也是比较分散的。但是这种方式也是有很大的缺陷的，当我们需要横向扩展节点的时候，缓存重组的代价也是非常大的。下面我们修改代码二，在nodes中添加node4。

代码三

$nodes = array('node1','node2','node3','node4'); //nodes代表服务器节点

$keys = array('onmpw', 'jiyi', 'onmpw\_key', 'jiyi\_key', 'www','www\_key'); //这是key值

foreach( $keys as $key) {

$crc = crc32($key); // CRC値

$mod = $crc % ( count($nodes) );

$server = $nodes[ $mod ]; // 根据余数选择服务器

printf("%s => %s\n", $key, $server);

}

再次运行代码发现其结果如下

onmpw => node4  
jiyi => node4  
onmpw\_key => node3  
jiyi\_key => node3  
www => node2  
www\_key => node3

我们看上面的结果，发现只有www\_key命中了。像这种情况在添加节点的时候命中率降低，当添加Memcached服务器的时候缓存效率会瞬间降低，这时候负载会集中到数据库的服务器上面，从而有可能导致数据库由于瞬间压力过大而无法提供正常的服务。

所以说，还需要一种新的方式来解决集群横向扩展的问题。因此随着该问题的出现，新的分布式方法也诞生了——一致性Hash算法（Consistent Hashing）。

**Consistent Hashing**

一致性hash算法和一般的hash算法一样，都是对集群中的节点建立一个字典。因为该算法在Consistent Hashing算法入门及php实现 这篇文章中有详细的描述，并且有简单的php代码的实现，所以本章不再对Consistent Hashing有过多的赘述。

和一般的hash算法不同的是该方式相对来说在有节点添加或减少的时候能最大限度的减少数据的移动。也就是说命中率会有相当大的提高。那它是怎么来实现的呢？

首先求出memcached服务器（节点）的哈希值， 并将其配置到0～1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1仍然找不到服务器，就会保存到第一台memcached服务器上。

具体在PHP-Memcache客户端中的实现代码的核心部分如下

void mmc\_consistent\_add\_server(void \*s, mmc\_t \*mmc, unsigned int weight)

{

mmc\_consistent\_state\_t \*state = s;

int i, key\_len, points = weight \* MMC\_CONSISTENT\_POINTS;

unsigned int seed = state->hash->init(), hash;

/\* buffer for "host:port-i\0" \*/

char \*key = emalloc(strlen(mmc->host) + MAX\_LENGTH\_OF\_LONG \* 2 + 3);

key\_len = sprintf(key, "%s:%d-", mmc->host, mmc->tcp.port);

seed = state->hash->combine(seed, key, key\_len);

/\* add weight \* MMC\_CONSISTENT\_POINTS number of points for this server \*/

/\* 申请保存所有节点及其各自的虚拟节点的空间 \*/

state->points = erealloc(state->points, sizeof(\*state->points) \* (state->num\_points + points));

for (i=0; i<points; i++) {

key\_len = sprintf(key, "%d", i);

// 对每个节点及虚拟节点计算hash值

hash = state->hash->finish(state->hash->combine(seed, key, key\_len));

// 保存个虚拟节点的hash值和该节点的信息

state->points[state->num\_points + i].server = mmc;

state->points[state->num\_points + i].point = hash;

}

state->num\_points += points;

state->num\_servers++;

state->buckets\_populated = 0;

efree(key);

}

上面的代码是对服务节点的分散，使每个节点及其各自的虚拟节点随机的均匀分布在圆上。

mmc\_t \*mmc\_consistent\_find\_server(void \*s, const char \*key, unsigned int key\_len TSRMLS\_DC)

{

mmc\_consistent\_state\_t \*state = s;

if (state->num\_servers > 1) {

unsigned int hash;

if (!state->buckets\_populated) {

mmc\_consistent\_populate\_buckets(state);

}

//计算key的hash值

hash = mmc\_hash(state->hash, key, key\_len);

return state->buckets[hash % MMC\_CONSISTENT\_BUCKETS];

}

//没有找到相应的服务节点，则返回第一个节点

return state->points[0].server;

}

这段核心代码是在存储一个键值对的时候根据该键值对的key去寻找要存储的服务节点。同样的，是对key值进行hash计算，然后根据计算出的hash值，该key肯定也会分布在圆的某个位置上，然后从该位置开始顺时针查找将要遇到的第一个服务节点。如果一直到末尾都没有遇到服务节点，则直接返回第一个服务节点。这也就是 return state->points[0].server的意义所在。

总结：以上两种方式都在PHP-Memcache客户端中已经实现。还是那句话，对于Memcached的分布式思想都是在客户端中体现的。不同的客户端实现的方式可能有所不同，但是相信基本思想都是一样的。 同时也欢迎大家能给出不同的意见。