# Memcached

### Memcahced介绍以及安装

Memcached 是一个开源、高性能、分布式内存对象缓存系统

Memcache 和Memcached 的区别？

Memcache 是 danga.com 的一个项目，最早是为 LiveJournal 服务的，目前全世界不少人使用这个缓存项目来构建自己大负载的网站，来分担数据库的压力。

Memcached 是一个开源、高性能、分布式内存对象缓存系统

Memcache 是这个项目的名称，而 memcached 是它服务器端的主程序文件名

为什么使用memcache

 由于网站的高并发读写需求，传统的关系型由于网站的高并发读写需求，传统的关系型数据库开始出现瓶颈，例如：

1）对数据库的高并发读写：

关系型数据库本身就是个庞然大物，处理过程非常耗时（如解析SQL语句，事务处理等）。如果对关系型数据库进行高并发读写（每秒上万次的访问），那么它是无法承受的。

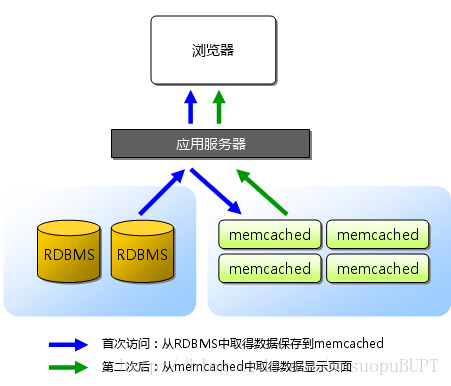
2）对海量数据的处理：

对于大型的SNS网站，每天有上千万次的数据产生（如twitter, 新浪微博）。对于关系型数据库，如果在一个有上亿条数据的数据表种查找某条记录，效率将非常低。

使用memcache能很好的解决以上问题。

在实际使用中，通常把数据库查询的结果保存到Memcache中，下次访问时直接从memcache中读取，而不再进行数据库查询操作，这样就在很大程度上减少了数据库的负担。

保存在memcache中的对象实际放置在内存中，这也是memcache如此高效的原因。



Memcahced服务端下载地址：

<http://s3.amazonaws.com/downloads.northscale.com/memcached-win64-1.4.4-14.zip>

解压然后安装并开启服务

### Memcached的简单使用

1. 使用EnyimMemcached客户端

引用命名空间 ： Install-Package EnyimMemcached

配置memcached

<configuration>

<configSections>

<sectionGroup name="enyim.com">

<section name="memcached" type="Enyim.Caching.Configuration.MemcachedClientSection,Enyim.Caching" />

</sectionGroup>

<section name="memcached" type="Enyim.Caching.Configuration.MemcachedClientSection, Enyim.Caching"/>

</configSections>

<enyim.com>

<memcached>

<servers>

<!-- put your own server(s) here-->

<add address="127.0.0.1" port="11211" />

</servers>

<socketPool minPoolSize="10" maxPoolSize="100" connectionTimeout="00:00:10" deadTimeout="00:02:00" />

</memcached>

</enyim.com>

<memcached keyTransformer="Enyim.Caching.TigerHashTransformer,Enyim.Caching">

<servers>

<add address="127.0.0.1" port="11211" />

</servers>

<socketPool minPoolSize="2" maxPoolSize="100" connectionTimeout="00:00:10" deadTimeout="00:02:00" />

</memcached>

</configuration>

简单调用

using Enyim.Caching;

using Enyim.Caching.Configuration;

using Enyim.Caching.Memcached;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Net;

using System.Net.Sockets;

using System.Text;

using System.Threading.Tasks;

namespace ClassLibrary1

{

public sealed class MemCachedHelper

{

private static MemcachedClient MemClient;

static readonly object padlock = new object();

//线程安全的单例模式

public static MemcachedClient getInstance()

{

if (MemClient == null)

{

lock (padlock)

{

if (MemClient == null)

{

MemClientInit();

}

}

}

return MemClient;

}

//初始化缓存

static void MemClientInit()

{

try

{

MemClient = new MemcachedClient("enyim.com/memcached");

}

catch (Exception ex)

{

throw ex;

}

}

//构造函数

static MemCachedHelper()

{

getInstance();

}

public static void Store(string Key, object Value, DateTime ExpiredAt)

{

MemClient.Store(StoreMode.Set, Key, Value, ExpiredAt);

}

public static T Get<T>(string Key)

{

return MemClient.Get<T>(Key);

}

public static void Remove(string Key)

{

MemClient.Remove(Key);

}

}

}

调用代码

MemCachedHelper.Store(“key1”,”value1”,DateTime.Now.AddDays(1))

String s = MemCachedHelper.Get<string>(“key1”);

1. 使用Memcached.ClientLibrary客户端

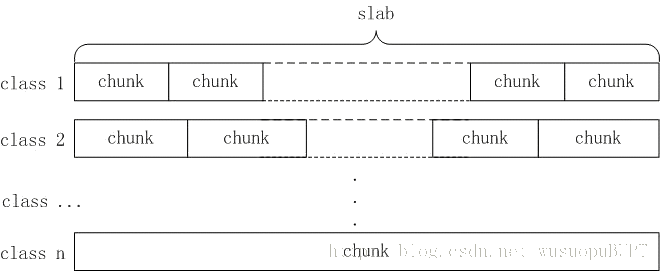
引用命名空间 ： Install-Package Memcached.ClientLibrary

略

### Memcached 内存管理

使用 slab 分配算法保存数据

slab分配算法的原理是：把固定大小（1MB）的内存分为n小块，如下图所示：



slab分配算法把每1MB大小的内存称为一个slab页，每次向系统申请一个slab页，然后再通过分隔算法把这个slab页分割成若干个小块的chunk（如上图所示），然后把这些chunk分配给用户使用，**分割算法**如下（在slabs.c文件中）：

/\*\*

\* Determines the chunk sizes and initializes the slab class descriptors

\* accordingly.

\*/

void slabs\_init(const size\_t limit, const double factor, const bool prealloc) {

int i = POWER\_SMALLEST - 1;

unsigned int size = sizeof(item) + settings.chunk\_size;

mem\_limit = limit;

if (prealloc) {

/\* Allocate everything in a big chunk with malloc 通过malloc的方式申请内存\*/

mem\_base = malloc(mem\_limit);

if (mem\_base != NULL) {

mem\_current = mem\_base;

mem\_avail = mem\_limit;

} else {

fprintf(stderr, "Warning: Failed to allocate requested memory in"

" one large chunk.\nWill allocate in smaller chunks\n");

}

}

memset(slabclass, 0, sizeof(slabclass));

while (++i < POWER\_LARGEST && size <= settings.item\_size\_max / factor) {

/\* Make sure items are always n-byte aligned 注意这里的字节对齐\*/

if (size % CHUNK\_ALIGN\_BYTES)

size += CHUNK\_ALIGN\_BYTES - (size % CHUNK\_ALIGN\_BYTES);

slabclass[i].size = size;

slabclass[i].perslab = settings.item\_size\_max / slabclass[i].size;

size \*= factor;//以1.25为倍数增大chunk

if (settings.verbose > 1) {

fprintf(stderr, "slab class %3d: chunk size %9u perslab %7u\n",

i, slabclass[i].size, slabclass[i].perslab);

}

}

power\_largest = i;

slabclass[power\_largest].size = settings.item\_size\_max;

slabclass[power\_largest].perslab = 1;

if (settings.verbose > 1) {

fprintf(stderr, "slab class %3d: chunk size %9u perslab %7u\n",

i, slabclass[i].size, slabclass[i].perslab);

}

/\* for the test suite: faking of how much we've already malloc'd \*/

{

char \*t\_initial\_malloc = getenv("T\_MEMD\_INITIAL\_MALLOC");

if (t\_initial\_malloc) {

mem\_malloced = (size\_t)atol(t\_initial\_malloc);

}

}

if (prealloc) {

slabs\_preallocate(power\_largest);

}

}

上面代码中的slabclass是一个类型为slabclass\_t结构的数组，其定义如下：

typedef struct {

unsigned int size; /\* 每个item大小 sizes of items \*/

unsigned int perslab; /\* 每个page中包含多少个item how many items per slab \*/

void \*\*slots; /\* 空闲的item指针 list of item ptrs \*/

unsigned int sl\_total; /\* 以分配空闲的item 个数 size of previous array \*/

unsigned int sl\_curr; /\* 当前空闲的item位置(也就是实际空闲item个数)，从后往前的 first free slot \*/

void \*end\_page\_ptr; /\* 指向最后一个页面中空闲的item开始位置 pointer to next free item at end of page, or 0 \*/

unsigned int end\_page\_free; /\* 最后一个页面，item个数 number of items remaining at end of last alloced page \*/

unsigned int slabs; /\* 实际使用slab(page)个数 how many slabs were allocated for this class \*/

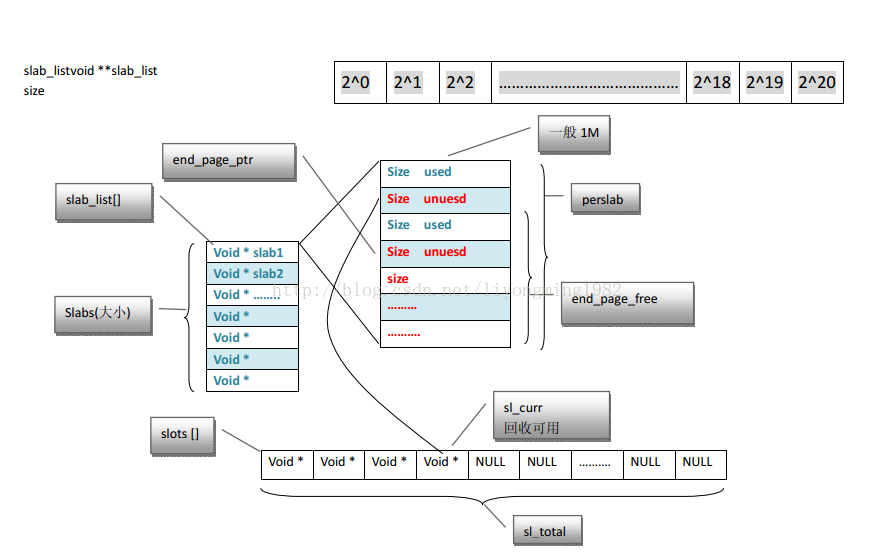
void \*\*slab\_list; /\* 所有page的指针 array of slab pointers \*/

unsigned int list\_size; /\* 已经分配page指针个数 size of prev array \*/

unsigned int killing; /\* index+1 of dying slab, or zero if none \*/

size\_t requested; /\* 所有被使用了的内存的大小 The number of requested bytes \*/

} slabclass\_t;



说明一下 slabclass 结构体的各个属性:

(1) size 和 perslab

size 定义该 slabclass 的 chunk 大小, perslab 表示每个 slab 可以切分成多少个 chunk, 如果一个 slab 等于

1M, 那么就有 perslab = 1M / size

(2) slots 和 sl\_curr

slots 是回收的 item 链表, 从某个 slabclass 分配出去一个 item, 当 item 回收的时候,

不是把这 item 使用的内存交还给 slab, 而是让这个 item 挂在 slots 链表的尾部. sl\_curr 表示当前链表中

有多少个回收而来的空闲 item.

(3) slab\_list 和 list\_size

前面说过, 初始时, memcached 为每个 slabclass 分配一个 slab, 当这个 slab 内存块使用完后, memcached

就分配一个新的 slab, 所以 slabclass 可以拥有多个 slab, 这些 slab 就是通过 slab\_list 数组来管理的, list\_size

表示当前 slabclass 有多少个 slab.

(4) end\_page\_ptr 和 end\_page\_free

在 subclass 内, 只有最后一个 slab 存在空闲的内存, 其它 slab 的 chunk 都分配出去了, end\_page\_ptr

指向最后一个 slab 中的空闲内存块, end\_page\_free 表示最后一个 slab 中还剩下多少个空闲 chunk.

(5) static item \*heads[LARGEST\_ID];

static item \*tails[LARGEST\_ID];

当 memcached 没有足够的内存使用时, 必须选择性地回收一些 item, 回收采用 LRU 算法, 这就需要维护

一个按照最近访问时间排序的 LRU 队列. 在 memcached 中,

每个 slabclass 维护一个链表, 比如 slabclass[i] 的链表头指针为 heads[i], 尾指针为 tails[i],

已分配出去的 item 都存储在链表中. 而且链表中 item 按照最近访问时间排序, 这样一些链表相当于LRU 队列.

由分割算法的源代码可知，slab算法按照不同大小的chunk分割slab页，而不同大小的chunk以factor（默认是1.25）倍增大。

分配的原则是, 优先从 slots 指向的空闲链表中分配, 空闲链表没有, 才从 slab 中分配一个空闲的 chunk.

内存分配

static void \*do\_slabs\_alloc(const size\_t size, unsigned int id) {

slabclass\_t \*p;

void \*ret = NULL;

item \*it = NULL;

if (id < POWER\_SMALLEST || id > power\_largest) {//判断id是否会导致slabclass[]数组越界

MEMCACHED\_SLABS\_ALLOCATE\_FAILED(size, 0);

return NULL;

}

p = &slabclass[id];//获取slabclass[id]的引用

assert(p->sl\_curr == 0 || ((item \*)p->slots)->slabs\_clsid == 0);//判断slabclass[id]是否有剩余的chunk

if (! (p->sl\_curr != 0 || do\_slabs\_newslab(id) != 0)) {//如果slabclass[id]中已经没有空余chunk并且试图向系统申请一个“页”（slab）的chunk失败,则返回NULL

/\* We don't have more memory available \*/

ret = NULL;

} else if (p->sl\_curr != 0) {//slabclass[id]的空闲链表中还有chunk，则直接将其分配出去

it = (item \*)p->slots;//获取空闲链表的头指针

p->slots = it->next;//将头结点指向下一个结点（取下头结点）

if (it->next) it->next->prev = 0;//将新头结点的prev指针置空

p->sl\_curr--;//减少slabclass[id]空闲链表中的chunk计数

ret = (void \*)it;//将头结点赋给ret指针

}

if (ret) {//请求成功

p->requested += size;//更新slabclass[id]所分配的内存总数

MEMCACHED\_SLABS\_ALLOCATE(size, id, p->size, ret);

} else {

MEMCACHED\_SLABS\_ALLOCATE\_FAILED(size, id);

}

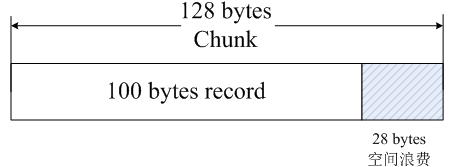
return ret;

}

当Memcached接收到客户端发送过来的数据时首先会根据收到数据的大小选择一个最合适的Slab Class，然后通过查询Memcached保存着的该Slab Class内空闲Chunk的列表就可以找到一个可用于存储数据的Chunk。

当一条数据库过期或者丢弃时，该记录所占用的Chunk就可以回收，重新添加到空闲列表中。从以上过程我们可以看出Memcached的内存管理制效率高，而且不会造成内存碎片，但是它最大的缺点就是会导致空间浪费。因为每个 Chunk都分配了特定长度的内存空间，所以变长数据无法充分利用这些空间。

如图所示，将100个字节的数据缓存到128个字节的Chunk中，剩余的28个字节就浪费掉了。



item

为实际存储数据的结构体，由两部分组成，属性信息和数据部分，数据部分包括cas，key和真实的value信息。

typedef struct \_stritem {

struct \_stritem \*next;//item在slab中存储时，是以双链表的形式存储的,next即后向指针

struct \_stritem \*prev;//prev为前向指针

struct \_stritem \*h\_next;//Hash桶中元素的链接指针

rel\_time\_t time; //最近访问时间

rel\_time\_t exptime;//过期时间

int nbytes;//数据大小

unsigned short refcount;//引用次数

uint8\_t nsuffix;

uint8\_t it\_flags;

uint8\_t slabs\_clsid;//标记item属于哪个slabclass下

uint8\_t nkey; //key的长度

union {

uint64\_t cas;

char end;

} data[];//真实的数据信息

} item;

添加item

do\_item\_alloc

mutex\_lock(&cache\_lock);//执行LRU锁 存储时，会尝试从LRU中选择合适的空间的空间

int tries = 5;//如果LRU中尝试5次还没合适的空间，则执行申请空间的操作

search = tails[id];//第id个LRU表的尾部

/\* We walk up \*only\* for locked items. Never searching for expired

for (; tries > 0 && search != NULL; tries--, search=search->prev) {

uint32\_t hv = hash(ITEM\_key(search), search->nkey, 0);//获取分段锁

if ((search->exptime != 0 && search->exptime < current\_time) || (search->time <= oldest\_live && oldest\_live <= current\_time)) { //过期时间的判断

it = search;

} else if ((it = slabs\_alloc(ntotal, id)) == NULL) {//申请合适的slabclass

......

}

break;

}

if (!tried\_alloc && (tries == 0 || search == NULL))//5次循环查找，未找到合适的空间

it = slabs\_alloc(ntotal, id);//则从内存池申请新的空间

return it;

}

注：每次新分配item时，先进行5次循环：检查LRU尾部item是否过期，过期则重用，否则尝试slabs\_alloc申请新内存； 若5次后仍未获取可用内存，则再次尝试slabs\_alloc申请内存；

删除过期item:

延迟删除过期item到查找时进行，可以提高memcache的效率，因为不必每时每刻检查过期item,从而提高CPU工作效率

使用LRU(Least Recently Used 近期最少使用)算法淘汰数据：

/\*

\* try to get one off the right LRU

\* don't necessariuly unlink the tail because it may be locked: refcount>0

\* search up from tail an item with refcount==0 and unlink it; give up after 50

\* tries

\*/

if (tails[id] == 0) {

itemstats[id].outofmemory++;

return NULL;

}

for (search = tails[id]; tries > 0 && search != NULL; tries--, search=search->prev) {

if (search->refcount == 0) { //refcount==0的情况，释放掉

if (search->exptime == 0 || search->exptime > current\_time) {

itemstats[id].evicted++;

itemstats[id].evicted\_time = current\_time - search->time;

STATS\_LOCK();

stats.evictions++;

STATS\_UNLOCK();

}

do\_item\_unlink(search);

break;

}

}

it = slabs\_alloc(ntotal, id);

if (it == 0) {

itemstats[id].outofmemory++;

/\* Last ditch effort. There is a very rare bug which causes

\* refcount leaks. We've fixed most of them, but it still happens,

\* and it may happen in the future.

\* We can reasonably assume no item can stay locked for more than

\* three hours, so if we find one in the tail which is that old,

\* free it anyway.

\*/

tries = 50;

for (search = tails[id]; tries > 0 && search != NULL; tries--, search=search->prev) {

if (search->refcount != 0 && search->time + 10800 < current\_time) { //最近3小时没有被访问到的情况，释放掉

itemstats[id].tailrepairs++;

search->refcount = 0;

do\_item\_unlink(search);

break;

}

}

it = slabs\_alloc(ntotal, id);

if (it == 0) {

return NULL;

}

}

从item列表的尾部开始遍历，找到refcount==0的chunk,调用do\_item\_unlink()函数释放掉，另外，search->time+10800<current\_time(即最近3小时没有被访问过的item)，也释放掉--这就是LRU算法的原理。

### Memcached和Redis的区别

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--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--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、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。

 Redis支持服务器端的数据操作：Redis相比Memcached来说，拥有更多的数据结构和并支持更丰富的数据操作，通常在Memcached里，你需要将数据拿到客户端来进行类似的修改再set回去。这大大增加了网络IO的次数和数据体积。在Redis中，这些复杂的操作通常和一般的GET/SET一样高效。所以，如果需要缓存能够支持更复杂的结构和操作，那么Redis会是不错的选择。

 内存使用效率对比：使用简单的key-value存储的话，Memcached的内存利用率更高，而如果Redis采用hash结构来做key-value存储，由于其组合式的压缩，其内存利用率会高于Memcached。

 性能对比：由于Redis只使用单核，而Memcached可以使用多核，所以平均每一个核上Redis在存储小数据时比Memcached性能更高。而在100k以上的数据中，Memcached性能要高于Redis，虽然Redis最近也在存储大数据的性能上进行优化，但是比起Memcached，还是稍有逊色。