Memcached原理与实现

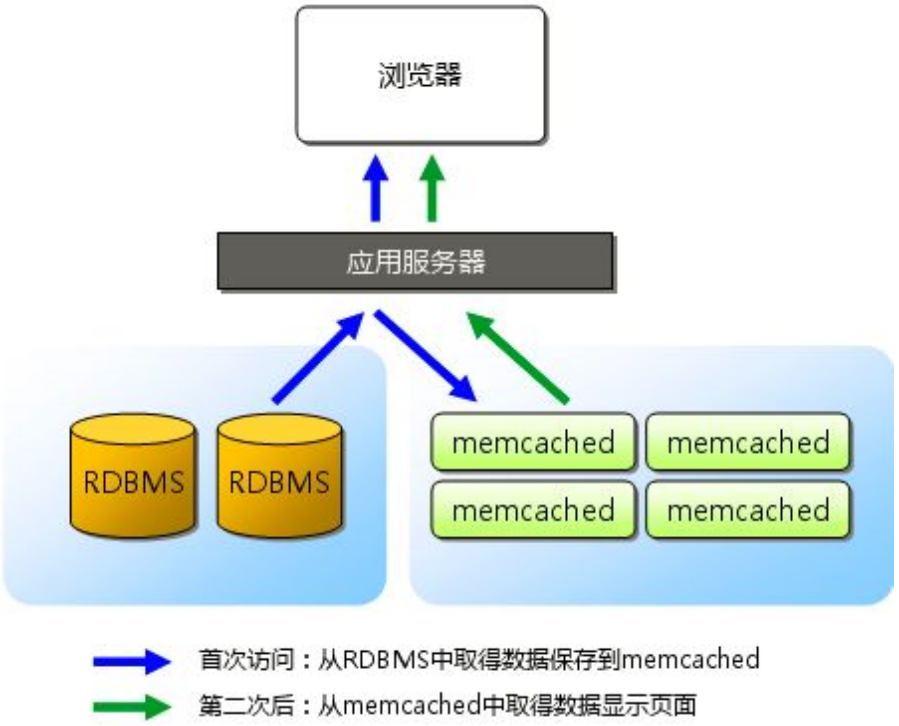
侯君

2014.09.21

Memcached简介

Memcached是一个高性能的分布式内存对象缓存系统，用于动态Web应用以减轻数据库负载。它通过在内存缓存对象来减少数据库的访问次数，从而提高动态、数据库驱动网站的访问速度。

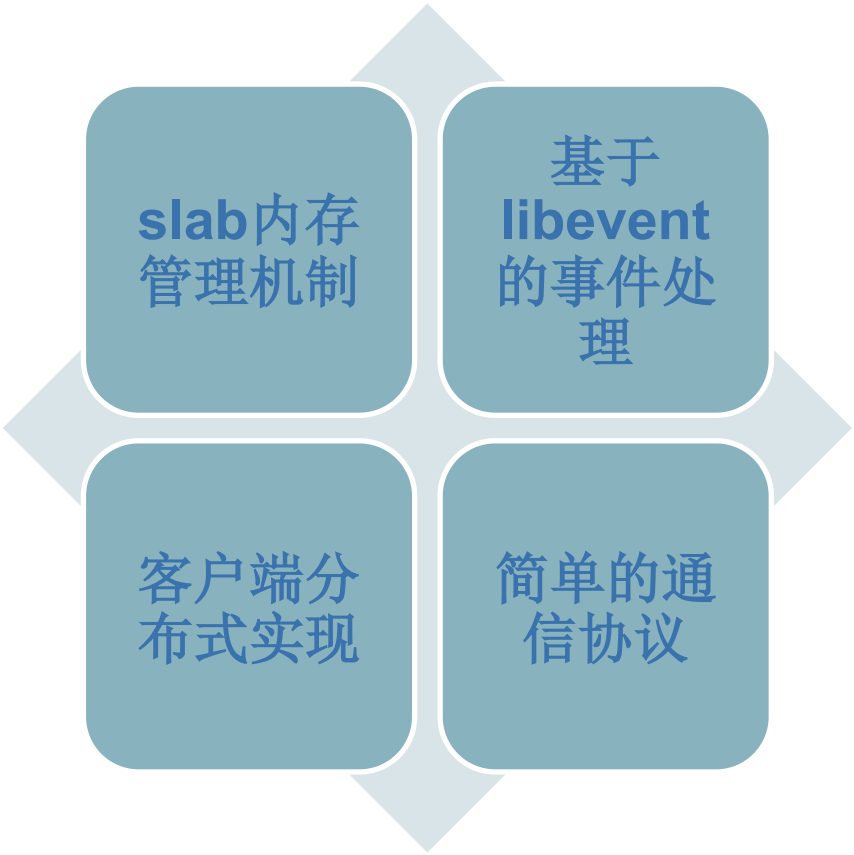
Memcached简介



Version：1.4.20

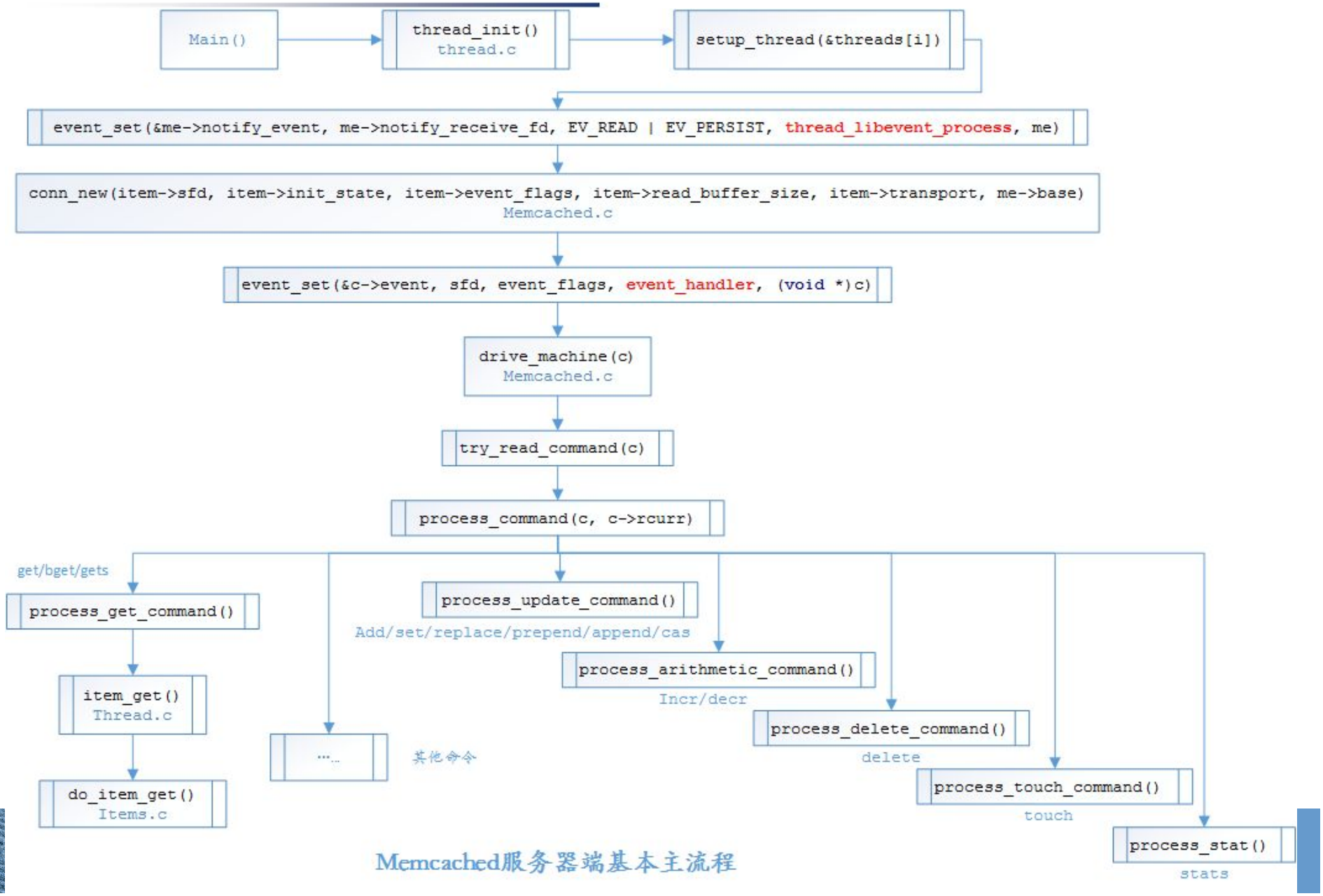
Memcached简介

关键技术

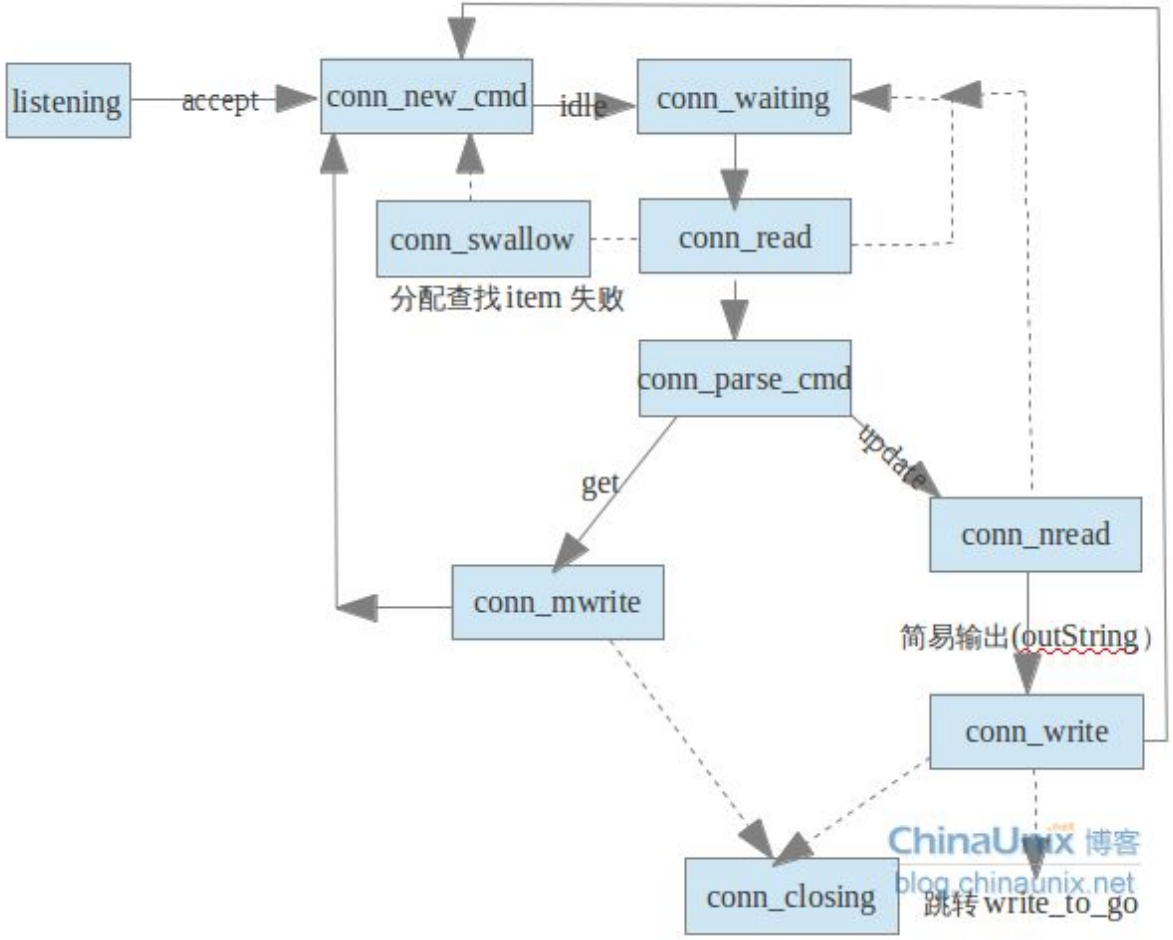


Memcached启动与初始化

执行流程



Memcached服务器端基本主流程




```
enum conn_states {
    conn_listening,  //监听状态
    conn_new_cmd,    //为新连接做一些准备
    conn_waiting,    //等待读取一个数据包
    conn_read,       //读取网络数据
    conn_parse_cmd,  //解析缓冲区的数据
    conn_write,      //简单的回复数据
    conn_nread,      //读取固定数据的网络数据
    conn_swallow,    //处理不需要的写缓冲区的数据
    conn_closing,    //关闭连接
    conn_mwrite,     //顺序的写多个item数据
    conn_max_state   //最大状态，做断言使用
};
```

启动初始化参数

 # ./memcached -d -m 2048 -l 10.0.0.40 -p 11211

"a:" //unix socket的权限位信息，unix socket的权限位信息和普通文件的权限位信息一样
"p:" //memcached监听的TCP端口值，默认是11211
"s:" //unix socket监听的socket文件路径
"U:" //memcached监听的UDP端口值，默认是11211
"m:" //memcached使用的最大内存值，默认是64M
"M" //当memcached的内存使用完时，不进行LRU淘汰数据，直接返回错误，该选项就是关闭LRU
"c:" //memcached的最大连接数,如果不指定，按系统的最大值进行
"k" //是否锁定memcached所持有的内存，如果锁定了内存，其他业务持有的内存就会减小
"v" //调试信息
"d" //设定以daemon方式运行
"l:" //绑定的ip信息，如果服务器有多个ip，可以在多个ip上面启动多个Memcached实例，注意：这个不是可接收的IP地址
"u:" //memcached运行的用户，如果以root启动，需要指定用户，否则程序错误，退出。
"P:" //memcached以daemon方式运行时，保存pid的文件路径信息
"f:" //内存的扩容因子，这个关系到Memcached内部初始化空间时的一个变化，后面详细说明
"n:" //chunk的最小大小(byte)，后续的增长都是该值*factor来进行增长的
"t:" //内部worker线程的个数，默认是4个，最大值推荐不超过64个
"L" //指定内存页的大小，默认内存页大小为4K，页最大不超过2M，调大页的大小，可有效减小页表的大小,提高内存访问的效率
"B:" //memcached内部使用的协议，支持二进制协议和文本协议，早期只有文本协议，二进制协议是后续加上的
"I:" //单个item的最大值，默认是1M,可以修改，修改的最小值为1k,最大值不能超过128M
"S" //打开sasl安全协议
"o:" //有四个参数项可以设置:
…

memcached.c/settings_init()

```
static void settings_init(void) {
    settings.use_cas = true;
    settings.access = 0700;
    settings.port = 11211;
    settings.udpport = 11211;
    /* By default this string should be NULL for getaddrinfo() */
    settings.inter = NULL;
    settings.maxbytes = 64 * 1024 * 1024; /* default is 64MB */
    settings.maxconns = 1024;         /* to limit connections-related memory to about 5MB */
    settings.verbose = 0;
    settings.oldest_live = 0;          /* 是否开启过期删除机制 */
    settings.evict_to_free = 1;       /* push old items out of cache when memory runs out */
    settings.socketpath = NULL;       /* by default, not using a unix socket */
    settings.factor = 1.25;           /* slab增长因子 */
    settings.chunk_size = 48;         /* space for a modest key and value */
    settings.num_threads = 4;         /* N workers */
    settings.num_threads_per_udp = 0;
    settings.prefix_delimiter = ':';
    settings.detail_enabled = 0;
    settings.reqs_per_event = 20;
    settings.backlog = 1024;
    settings.binding_protocol = negotiating_prot;
    settings.item_size_max = 1024 * 1024; /* The famous 1MB upper limit. */
    settings.maxconns_fast = false;
    settings.lru_crawler = false;
    settings.lru_crawler_sleep = 100;
    settings.lru_crawler_tocrawl = 0;
    settings.hashpower_init = 0;
    settings.slab_reassign = false;
    settings.slab_automove = 0;
    settings.shutdown_command = false;
    settings.tail_repair_time = TAIL_REPAIR_TIME_DEFAULT;
    settings.flush_enabled = true;
}
```

系统资源初始化

- Hash表初始化
- 统计信息的初始化
- 工作线程
- 网络连接
- 内存

Hash表初始化

```
#define HASHPOWER_DEFAULT 16
unsigned int hashpower = HASHPOWER_DEFAULT;
assoc_init(settings.hashpower_init);      // hashpower_init为0

void assoc_init(const int hashtable_init) {
    if (hashtable_init) {                  //按设置值进行初始化，如果没有则使用默认值16
        hashpower = hashtable_init;
    }
    primary_hashtable = calloc(hashsize(hashpower), sizeof(void *));  //返回指向分配起始地址的指针
    if (! primary_hashtable) {
        fprintf(stderr, "Failed to init hashtable.\n");
        exit(EXIT_FAILURE);
    }
    STATS_LOCK();
    stats.hash_power_level = hashpower;
    stats.hash_bytes = hashsize(hashpower) * sizeof(void *);
    STATS_UNLOCK();
}
```

统计信息初始化

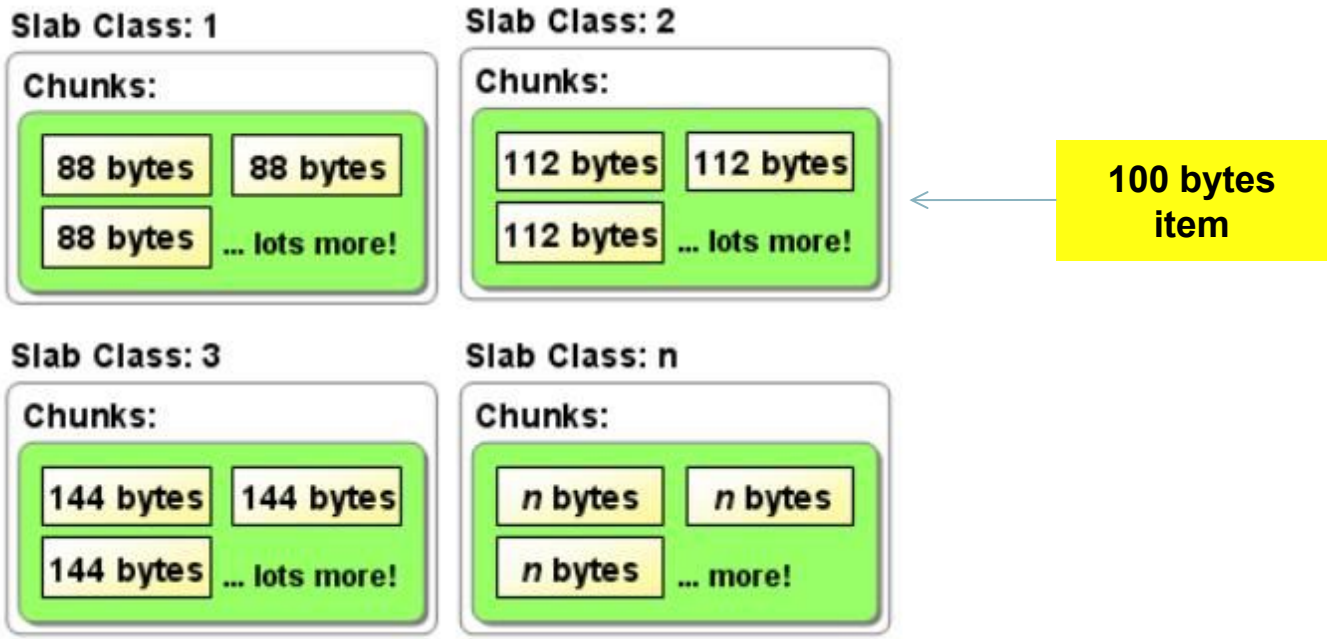
```
struct stats {
    pthread_mutex_t mutex;
    unsigned int  curr_items;          /* 当前缓存中存储的对象数量 */
    unsigned int  total_items;         /*memcached从启动到现在，存储的所有对象的数量，包括被删除的 */
    uint64_t      curr_bytes;          /* 缓存对象的存储空间，单位为bytes  */
    unsigned int  curr_conns;
    unsigned int  total_conns;
    uint64_t      rejected_conns;
    uint64_t      malloc_fails;
    unsigned int  reserved_fds;
    unsigned int  conn_structs;
    uint64_t      get_cmds;            /* 累积get数据的数量 */
    uint64_t      set_cmds;            /* 累积set数据的数量  */
    uint64_t      touch_cmds;
    uint64_t      get_hits;    /* 获取数据成功次数*/
    uint64_t      get_misses;  /* 获取数据失败次数*/
    uint64_t      touch_hits;
    uint64_t      touch_misses;
    uint64_t      evictions;
    uint64_t      reclaimed;
    time_t        started;         /* when the process was started */
    bool          accepting_conns;  /* whether we are currently accepting */
    uint64_t      listen_disabled_num;
    unsigned int  hash_power_level; /* Better hope it's not over 9000 */
    uint64_t      hash_bytes;      /* size used for hash tables */
    bool          hash_is_expanding; /* If the hash table is being expanded */
    uint64_t      expired_unfetched; /* items reclaimed but never touched */
    uint64_t      evicted_unfetched; /* items evicted but never touched */
    bool          slab_reassign_running; /* slab reassign in progress */
    uint64_t      slabs_moved;      /* times slabs were moved around */
    bool          lru_crawler_running; /* crawl in progress */
};
```

```
/*初始化stats命令，*/
static void stats_init(void) {
    stats.curr_items = stats.total_items = stats.curr_conns = stats.total_conns = stats.conn_structs = 0;
    stats.get_cmds = stats.set_cmds = stats.get_hits = stats.get_misses = stats.evictions = stats.reclaimed
= 0;
    stats.touch_cmds = stats.touch_misses = stats.touch_hits = stats.rejected_conns = 0;
    stats.malloc_fails = 0;
    stats.curr_bytes = stats.listen_disabled_num = 0;
    stats.hash_power_level = stats.hash_bytes = stats.hash_is_expanding = 0;
    stats.expired_unfetched = stats.evicted_unfetched = 0;
    stats.slabs_moved = 0;
    stats.accepting_conns = true; /* assuming we start in this state. */
    stats.slab_reassign_running = false;
    stats.lru_crawler_running = false;

    /* make the time we started always be 2 seconds before we really
       did, so time(0) - time.started is never zero.  if so, things
       like 'settings.oldest_live' which act as booleans as well as
       values are now false in boolean context... */
    process_started = time(0) - ITEM_UPDATE_INTERVAL - 2;
    stats_prefix_init();  //初始化存放stats信息的内存空间
}
```

Slab管理内存机制简介

内存的分配和回收通过Slab allocator实现，设计类似于内存池，按照预先规定的大小，将分配的内存分割成特定长度的块(chunk)，并将大小相同的chunk集合成组(class)



各class的chunk size按照factor增长，请求内存时，找到最适合item大小的chunk 所在的slab class，然后从该class中找到空闲的chunk分配出去。

基本数据结构-item

Structure for storing items within memcached
_stritem *next // 指向链表下一个item
_stritem *prev // 指向上一个item
_stritem *h_next // 指向hash bucket的下一项
rel_time_t time // 最近访问时间
rel_time_t exptime // 过期时间
int nbytes // 存放数据大小
unsigned short refcount // 引用计数

union {
uint64_t cas;
char end;
} data[]; // 存放数据

uint8_t nsuffix, it_flags, slabs_clsid, nkey 其它

Item由两部分组成，一部分记录结构属性，另一部分为数据。

item 结构体的定义使用了一个常用的技巧: 定义空数组 data，用来指向 item 数据部分的首地址, 使用空数组的好处是 data 指针本身不占用任何存储空间, 为 item 分配存储空间后, data 自然而然就指向数据部分的首地址。

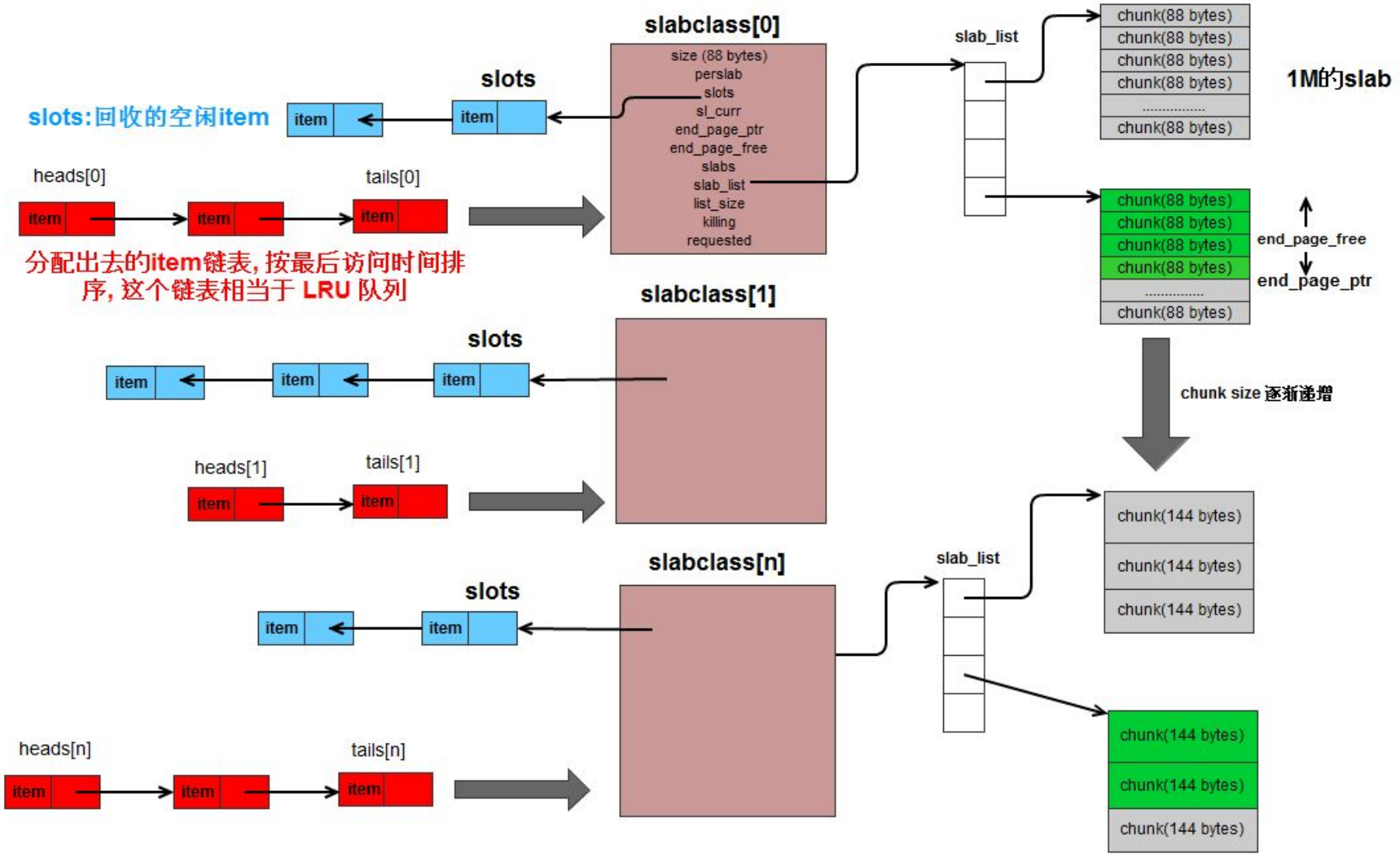
基本数据结构-slab

Structure for Slab(slabclass_t)
unsigned int size // 每个chunk的大小
unsigned int perslab // 一个slab存放chunk的数量
void *slots // 空闲可插入item的插槽
unsigned int sl_curr // 回收的空闲item数
unsigned int slabs // class中分配的slabs数目
void **slab_list // 存储slab指针的数组
unsigned int list_size
unsigned int killing //dying slab
size_t requested ？？

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。

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 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。

Slabclass结构示意图



内存初始化 slab.c/slabs_init()

Memcached首次默认分配64MB的内存空间，之后所有数据都是在该片空间进行存储。

slabs_init(settings.maxbytes, settings.factor, preallocate);

Slab分配机制的缺点

存在内存碎片，解决办法是根据实际item的大小，预先调整factor的大小

基本操作命令

•存储命令

set/add/replace/append/prepend/cas

•读取命令

get/gets

•显示统计状态命令

stats

•其他命令

flush_all,vsersion,quit

存储命令

命令格式
<command name> <key> <flags> <exptime> <bytes>
<data block>

命令解释：

<command name>	Set/add/replace/append
<key>	**关键字**
<flags>	**标识关键字**
<exptime>	**数据的存活时间，0表永远**
<bytes>	**存储字节数**
<data block>	**存储的数据块**

- set：key如果存在，可以对其进行更新
- add: 只有数据不存在时才进行添加
- replace: 只有数据存在时在进行replace
- append: 在现有缓存数据后添加数据
- prepend: 在现有缓存数据前添加数据
- cas: check and set，只有当最后一个参数和gets所获取的参数匹配时才能返回，否则返回“EXISTS”

读取命令

命令格式
get <key>

- gets：比get多返回一个值，用以确定value是否改变，当value改变时，该返回值也会变化

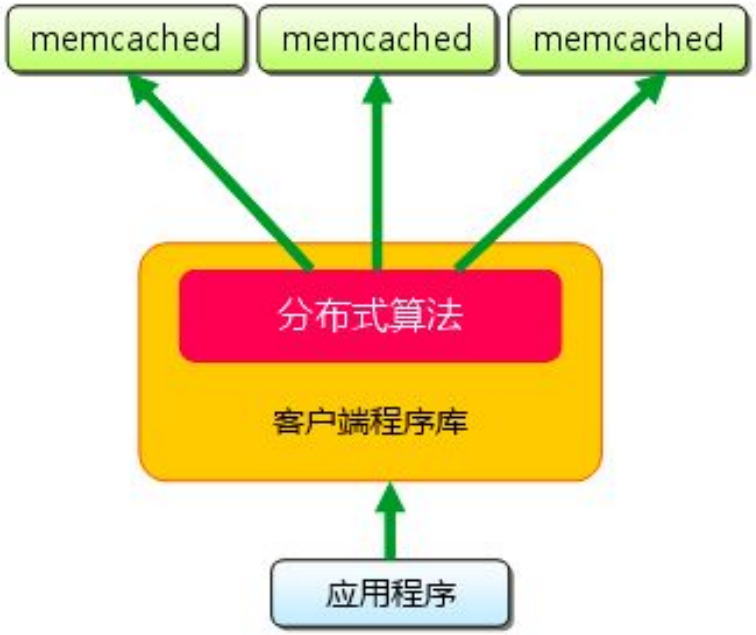
其他命令

命令格式
flush_all

- flush_all：清理缓存中的所有key/value。由于memcached的lazy expiration机制和删除机制，item占用的内存不会立即回收。

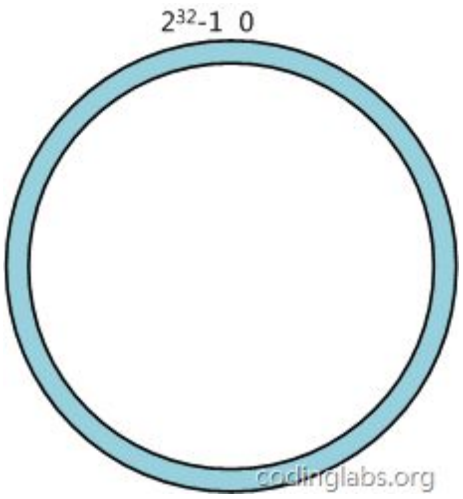
```
item *do_item_get(const char *key, const size_t nkey, const uint32_t hv) {
…
 /* 找到item，检查是否expire */
   if (it != NULL) {
      //判断是否启用过期删除机制
      if (settings.oldest_live != 0 && settings.oldest_live <= current_time &&
         it->time <= settings.oldest_live) {
         do_item_unlink(it, hv);    //从hash表和LRU中移除
         do_item_remove(it);
         it = NULL;
         if (was_found) {
            fprintf(stderr, " -nuked by flush");
         } /* 判断item是否过期 */
      } else if (it->exptime != 0 && it->exptime <= current_time) {
         do_item_unlink(it, hv);
         do_item_remove(it);
         it = NULL;
         if (was_found) {
            fprintf(stderr, " -nuked by expire");
         }
      } else {
         it->it_flags |= ITEM_FETCHED;
         DEBUG_REFCNT(it, '+');
      }
   }
…
}
```

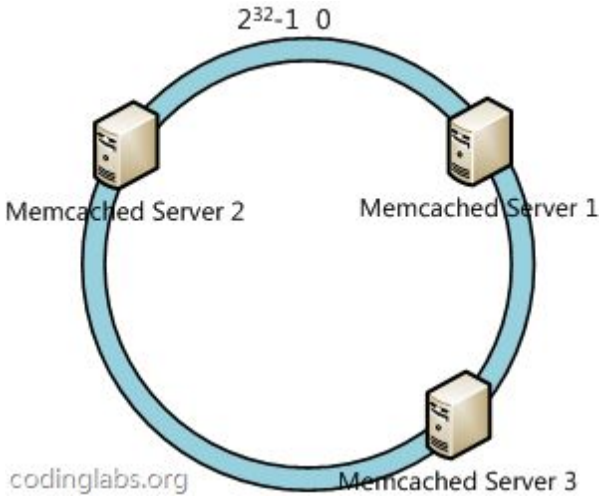
客户端分布式实现

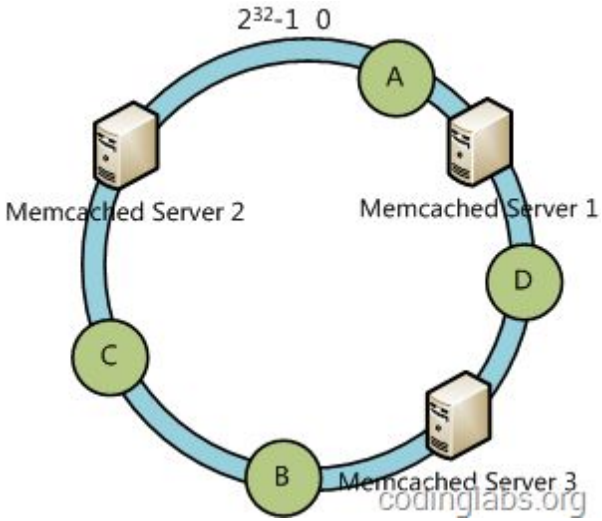


- 根据服务器台数的余数进行分散，求得键的整数哈希值，除以服务器台数，根据余数选择服务器。
- Consistent Hash：

Consistent Hash







将哈希值空间[0,2^32-1]组织成一个虚拟的圆环

选择服务器IP或主机名使用函数H进行hash，确定节点在环上的位置

将key用相同函数H计算哈希值，确定key在环上位置，从该位置顺时针行走，遇到的第一个服务器就是存储节点

参考文献

- Memcached源码分析
- Consistent Hash算法简介

Thanks!
Q&A