(/)

\{C}ODEA1K(S) => ZERO UNDERSTANDING PLACEBO IN SOURCE LEVEL (͡° ͜ʖ ͡°) (/ATOM.XML)

HOME (/) / SERIES (/CATEGORIES)

Understanding The Memcached Source Code - Slab I

SEPTEMBER 12, 2018 (/UNDERSTANDING-MEMCACHED-SOURCE-CODE-I/) / MEMCACHED SOURCE CODE (/CATEGORIES/MEMCACHED-SOURCE-CODE/)

(/cn/understanding-memcached-source-code-I/)

♠ **slab allocator (I - this article , II** (/understanding-memcached-source-code-II/) **, III)** (/understanding-memcached-source-code-III/) is the core module of the cache system, which largely determines how efficient the bottleneck resource, memory, can be utilized. The other 3 parts, namely,

♥ **LRU algorithm (I** (/understanding-memcached-source-code-IV/) **, II** (/understanding-memcached-source-code-V/) **, III)** (/understanding-memcached-source-code-VI/) for entry expiration; and an

♣ **event driven model (I** (/understanding-memcached-source-code-VII/) **, II** (/understanding-memcached-source-code-VIII/) **, III)** (/understanding-memcached-source-code-IX/) based on libevent; and

♦ **consistent hashing** (/understanding-memcached-source-code-X-consistent-hashing/) for data distribution,

are built around it.

Variants of **slab allocator** is implemented in other systems, such as nginx and Linux kernel, to fight a common problem called **memory fragmentation**. And this article will, of course, focus on **Memcached**'s implementation of the algorithm.

memcached version: 1.4.28

Firstly, let's answer some questions.

Introduction

What is a slab

slabs are pre-allocated 1M memory chunks that can be subdivided for numerous objects. They are grouped into **slab class**es to serve allocation requests for various sizes.

What is memory fragmentation, how it occurs

In particular, **slab allocator** curbs **internal memory fragmentation**. This kind of fragmentation exits within an allocated memory chunk. In the context of OS kernel, for instance, the fundamental unit allocated by memory management sub-system is called a *page*.

> On the other hand, **external memory fragmentation** exists across chunks, and the solution of which (keyword: buddy) belongs to another story.

The most common phenomenon where **internal fragmentation** causes the problem is as following:

1) `malloc` of small objects is called a lot of times; and in the meantime;

2) `free` of those objects is called a lot of times.

The above process generates (a lot of) nominal “free” memory that cannot be utilized, as the discrete holes of various sizes, or **fragments**, can not be reused by subsequent `malloc` s for any objects that are larger than them.

Why memory fragmentation is bad

The impact of **memory fragmentation** is similar to that of **memory leak** - periodical system reboot is inevitable whenever the fragments accumulate to a certain level, which, increase the complexity in system operation, or even worse, leads to bad user experiences.

How the problem is fixed

Slab allocator does not eliminate **internal fragmentation**. Instead, it converges the fragments and locks them in fixated memory locations. This is done by 1) categorizing objects of similar sizes in **classes**; and 2) allocating objects belonging to the same **class** only on the same group of “**slab**s”, or, a **slab class**.

The detail devil is in the code, so we start reading the code.

reminder: Memcached version is 1.4.28

The core data structure in use

```
typedef struct {
    unsigned int size;      /* sizes of items */
    unsigned int perslab;   /* how many items per slab */

    void *slots;           /* list of item ptrs */
    unsigned int sl_curr;   /* total free items in list */

    unsigned int slabs;     /* how many slabs were allocated for this cla

    void **slab_list;       /* array of slab pointers */
    unsigned int list_size; /* size of prev array */

    size_t requested; /* The number of requested bytes */
} slabclass_t;

static slabclass_t slabclass[MAX_NUMBER_OF_SLAB_CLASSES];
```

slabclass_t@slabs.c

Module initialization

In this section we examine `slabs_init` that initializes `slabclass[MAX_NUMBER_OF_SLAB_CLASSES]` array. In particular, this process initializes the values of two fields, i.e., `slabclass_t.size` , the item (object) size of each **slab class**, and `slabclass_t.perslab` the item number one **slab** contains. This method is called from here as one of the *init* steps before the logic enters the *main even loop*.

In this step slab_sizes and settings.factor jointly control the routes in which sizes of each **slab class** are decided, they are:

a) if slab_sizes is not `NULL` , the values within the array are used directly; and

b) otherwise, the sizes are calculated as *base size* × *n* × `settings.factor` where *n* is the index within `slabclass` .

Besides the default values, the two arguments can be set at runtime as well.

The other two arguments of this method `settings.maxbytes` and `preallocate` will be discussed soon (../understanding-memcached-source-code-II). For now we set `false` to `preallocate` and ignore the relevant logic flow.

Next we look at the `slabs_init` itself.

```
void slabs_init(const size_t limit, const double factor, const bool preal
    int i = POWER_SMALLEST /* scr: 1 */ - 1;
    unsigned int size = sizeof(item) + settings.chunk_size; // scr: -----
...
    memset(slabclass, 0, sizeof(slabclass));

    while (++i < MAX_NUMBER_OF_SLAB_CLASSES-1) {
        if (slab_sizes != NULL) { // scr: ------------------------------
            if (slab_sizes[i-1] == 0)
                break;
            size = slab_sizes[i-1];
        } else if (size >= settings.item_size_max / factor) {
            break;
        }
        /* Make sure items are always n-byte aligned */
        if (size % CHUNK_ALIGN_BYTES) // scr: --------------------------
            size += CHUNK_ALIGN_BYTES - (size % CHUNK_ALIGN_BYTES);

        slabclass[i].size = size;
        slabclass[i].perslab = settings.item_size_max / slabclass[i].size
        if (slab_sizes == NULL)
            size *= factor; // scr: -------------------------------------
        if (settings.verbose > 1) {
            fprintf(stderr, "slab class %3d: chunk size %9u perslab %7u\n
                    i, slabclass[i].size, slabclass[i].perslab);
        }
    }
    // scr: ------------------------------------------------------------
    power_largest = i;
    slabclass[power_largest].size = settings.item_size_max;
    slabclass[power_largest].perslab = 1;
...
}
```

slabs_init@slabs.c

Route a

1) use the values in `slab_sizes`;

2) align the `size` to `CHUNK_ALIGN_BYTES`, and give the result to `slabclass[i].size`;

3) calculate the `slabclass[i].perslab`;

5) use the `settings.item_size_max` to initialize the last **slab class**.

Note that settings.item_size_max is the size of each **slab**, hence it is also the max size of items that are allocated on *slabs*. Likewise, the value of settings.item_size_max can be decided in runtime.

Route b

1) calculate the *base size* with *settings.chunk_size* plus the extra bytes for metadata (`item` will be discussed in following articles);

2) align the `size` to `CHUNK_ALIGN_BYTES` , and give the result to `slabclass[i].size` ; (same to route a)

3) calculate the `slabclass[i].perslab` ; (same to route a)

4) calculate the size for the next `slab class` using `factor` (`settings.factor`);

5) use the `settings.item_size_max` to initialize the last **slab class**. (same to route a)

References

memcached wiki (https://github.com/memcached/memcached/wiki)

第2回　memcachedのメモリストレージを理解する (http://gihyo.jp/dev/feature/01/memcached/0002)

Memcached源码分析之存储机制Slabs（7） (https://blog.csdn.net/initphp/article/details/44888555)

Understanding Malloc (https://gokulvasanblog.wordpress.com/2016/07/11/understanding-malloc-part1/)

Ch8 - Slab Allocator (https://www.kernel.org/doc/gorman/html/understand/understand011.html)

The Slab Allocator:An Object-Caching Kernel Memory Allocator (https://www.usenix.org/legacy/publications/library/proceedings/bos94/full_papers/bonwick.a)

That's it. Did I make a serious mistake? or miss out on anything important? Or you simply like the read. Link me on (https://medium.com/source-code/understanding-the-memcached-source-code-slab-i-9199de613762) -- I'd be chuffed to hear your feedback.

MEMCACHED SOURCE CODE (/CATEGORIES/MEMCACHED-SOURCE-CODE/)
CACHE (/TAGS/CACHE/) MEMCACHED (/TAGS/MEMCACHED/) SLAB ALLOCATOR (/TAGS/SLAB-ALLOCATOR/)

NEWER

Understanding The Memcached Source Code - Slab II
(/understanding-memcached-source-code-II/)

OLDER

setsockopt, TCP_NODELAY and Packet Aggregation I
(/network-essentials-setsockopt-TCP_NODELAY/)

Contents