·经典评述·

Nginx Slab 算法研究

宋雅琴[1,2] 郭志川[1]

([1] 中国科学院声学研究所国家网络新媒体工程技术研究中心 北京 100190 [2] 中国科学院大学 北京 100190)

摘要: Nginx 设计了简单的内存池进行内存管理来降低开发中对内存资源管理的复杂度。Nginx 各进程间使用共享内存的方式共享数据,而对共享内存的内存池进行管理的方法是基于经典的 Slab 算法,其通过构造小的内存块来避免内存碎片、使用链表方式连接有限的页面来提高分配速率。本文详细介绍 Nginx 的 Slab 算法,对其进行总结,并与经典的 Linux 内核的 Slab 算法对比。

关键词: 内存管理,Slab 算法,Nginx 服务器,内存池,内存碎片

Nginx Slab Algorithm Research

SONG Yaqin[1,2], GUO Zhichuan[1]

([1]National Network New Media Engineering Research Center, Institute of Acoustics, Chinese Academy of Sciences, Beijing, 100190, China, [2]University of Chinese Academy of Sciences, Beijing, 100190, China)

Abstract: Nginx designed a simple memory pool for memory management to reduce the complexity of memory management. Nginx uses shared memory to share data among processes. The way to manage memory pool with shared memory is based on the classic Slab algorithm, which avoids memory fragmentation by constructing small blocks of memory and increases memory allocation rate by using linked lists to connect limited pages. This article details Nginx's Slab algorithm, summarizes it, and compares it with the classic Slab algorithm in Linux kernel.

Keywords: Memory Management, Slab Algorithm, Nginx Server, Memory Pool, Memory Fragmentation

0 引言

Nginx (engine x) 是一个高性能的 HTTP 和反向代理轻量级服务器,也是一个 IMAP/POP3/SMTP 服务器[1]。它具有性能高、扩展性强、可靠性强、低内存消耗、高并发,热部署等优点而被广泛应用。

内存分配是计算机程序中最常使用的操作之一[2]。内存管理方法应该是根据特定应用程序的需求来权衡管理内存所需的时间和最大化用于一般应用的可用内存。一般来说,内存管理运行在两个层次: 用户空间和内核空间。在两种层次下,负责内存管理的代码统称为内存分配器[3]。内核内存分配器(kernel - memory allocator , KMA) 是操作系统内核的一部分,负责处理操作系统子系统的请求,同时为应用程序进程提供内存。用户内存分配器[4] (user - level memory allocator, UMA) 负责处理应用程序中的动态内存分配[5-7] (dynamic storage allocation, DSA) 。

Nginx 设计了简单的内存池来进行内存管理[8],在使用内存之前,预先申请分配一定数量的、大小相等

本文于 2018 - 01 - 09 收到。

(一般情况下) 的内存块留作备用。当有小块内存需求时,就从内存池中分出一部分内存块,若内存块不够用时,再继续申请新的内存; 若有大块内存申请时,仍然调用系统函数向操作系统申请。在释放内存的时候,Nginx 只释放大块内存链表中注册的内存,小块内存会在销毁内存池的时候一并释放,从而减少向操作系统申请内存的次数、避免内存泄漏、降低开发中对内存资源管理的复杂度,也减少了内存碎片[9]的存在。

Nginx 被设计为一个多进程的程序,如果进程间需要交互各种不同大小的对象,需要共享一些复杂的数据结构,那么直接使用内存池的方法给定一块共享内存而不进行管理则会造成效率低下,故使用高效的、适合小块内存频繁分配和释放的 Slab 算法内存管理机制来帮助快速实现多种对象间的跨 Nginx worker 进程通信,从而提高分配效率、减少内存碎片。

1　Linux Slab 算法

1994 年,Sun 的工程师 Jeff Bonwick 提出了 Slab 算法[10],并在 SunOS 中应用。1999 年,Linux Kernel 2.2.0 版本使用了 Slab 算法分配器,其源码位于文件/mm/slab.c。

Linux[11,12]的内存管理子系统可以分为 5 个部分: 物理内存管理器、内核内存管理器、内核虚拟内存管理器、虚拟内存管理器和用户内存管理器[13]。其中,物理内存管理器负责物理内存的分配、释放和回收,它以页为单位进行管理,其目的是提高性能、减少碎块,Linux 常用的页分配器是 Buddy System[14]。内核内存管理器专门复杂内核中小块内存的分配和释放,Linux 的内核内存管理器采用 3 个分配器,分别是 Slab、Slub[15]、Slob 分配器[3]。

Slab 分配器使用 Best - Fit 算法,基于对象进行管理,即内核中的数据结构(例如: task_struct,file_struct 等)。Slab 分配器考虑到复杂的数据结构,在大多数情况下,为了分配和释放所作的初始化和销毁对象的花销却超过了分配和释放本身的花销。因此使用对象缓存技术来处理频繁分配和释放的对象,其想法是在应用之间保存对象的初始状态 - 它的构造状态,使得对象不必每次被使用的时候都要进行销毁和重新创建,从而提高时间效率,避免内存碎片产生。

slab 是 Slab 分配器的主要单元。系统分配和回收的基本单元都是 slab。一个 slab 包含一或多页虚拟连续内存,其被分割成相等大小的数据块,并且有一个引用计数来表示有多少数据块已经被分配。

图 1 给出了 Slab 结构的高层组织结构。在最高层是 cache_chain,这是一个 slab 单元中缓存的链接列表。可以用来查找最适合所需要的分配大小的缓存(遍历列表)。cache_chain 的每个元素都是一个 *km*em_cache 结构的引用(称为一个 cache 高速缓存)。它定义了一个要管理的给定大小的对象池。缓存区的管理者纪录了该类对象的所有信息,包括对象的大小,性质(如是否用于 dma 操作),slab 的各项参数及使用情况等。

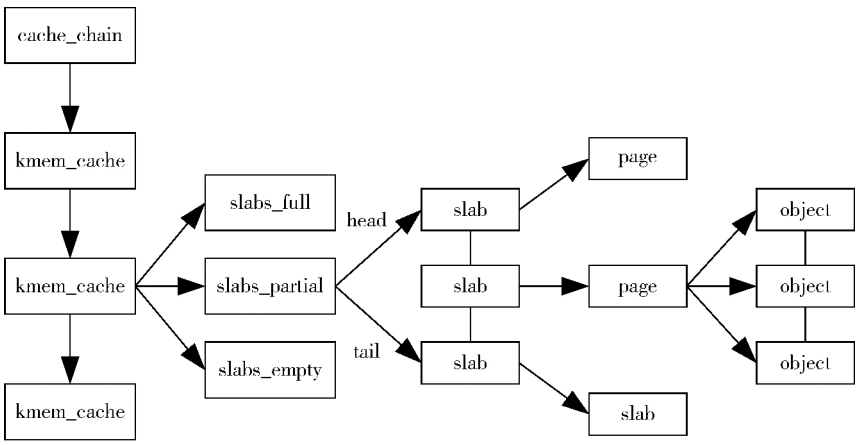

图 1　Linux Slab 分配器主要结构

每个缓存维护它所有 slab 的循环、双向链表。每一个缓存包含了三种 slabs 列表,slab 是一段连续的内存块(通常都是页面)。三种 slab 列表: ①slabs_full: 完全分配的 slab; ②slabs_partial: 部分分配的 slab; ③slabs_empty: 空 slab,或者没有对象被分配。

当 Slab 分配器接收到内存申请时,根据所申请内存的大小找到合适的 cache ,从 cache 管理的半满 slab 列表中分配对象,若失败则从全空 slab 中分配对象,若还不成功则说明 cache 中没有空闲对象,需要为 cache 创建一个新的 slab ,从新的 slab 中分配空闲对象。对象释放过程中,不仅要清空对象占用的空间,而且还要调整对象所属 slab 的状态,判断是否改变此 slab 在 cache 中的位置。

3 Nginx Slab 算法

3.1 Nginx 使用 Slab 方法

Nginx 是多进程的,其进程间的通信分为三种类别: Nginx 与操作系统通信,master 进程与 worker 进程通信,worker 进程间通信,分别用信号、套接字、共享内存作为传递消息的传递方式。

Nginx 各进程间共享数据的主要方式就是使用共享内存,Nginx 中 worker 进程间则是通过比较快速的共享内存进行通信的。共享内存是 Linux 下提供的最基本的进程间通信方法,它通过 mmap 系统或者 shmget 系统调用在内存中创建了一块连续的线性地址空间,而通过 munmap 系统或者 shmdt 系统调用可以释放这块内存。

在许多场景下,不同的 Nginx 请求间必须交互才能执行下去,例如限制一个客户端能够并发访问的请求数。可是 Nginx 被设计为一个多进程的程序,服务更强壮的另一面就是,Nginx 请求可能是分布在不同的进程上的。然而如果进程间需要交互各种不同大小的对象,需要共享一些复杂的数据结构,如链表、树、图等,而且 Nginx 采用了异步非阻塞的方式来处理请求,其单机支持 10 万以上的并发连接,使得进程之间交互频繁,那么直接使用内存池的方法给定一块共享内存而不进行管理的话会造成效率低下,故而使用了高效的、适合小块内存频繁分配和释放的 slab 内存管理机制来帮助快速实现多种对象间的跨 Nginx worker 进程通信。

Nginx 使用 ngx_shm_zone_t 进行内存池管理,其中定义了 ngx_shm_t 结构体,用于描述具体一块共享内存。操作 ngx_shm_t 结构体的方法有以下两个: ngx_shm_alloc 用于分配新的共享内存,ngx_shm_free 用于释放已经存在的共享内存。

3.2 Nginx Slab 分配器

Nginx 采用 Slab 方法管理共享内存,即可以将一页内存分割成多个大小相等的内存块,不同的页分割出来的内存块大小可以不同。用 page 数组管理页面,用 slots 数组管理一页分割出来的 slot 内存块。

Nginx Slab 分配器的思想主要包括 5 个要点:

(1) 把整块内存按 4KB 分为许多页。

(2) 基于空间换时间的思想,Slab 内存分配器会把请求分配的内存大小简化为极为有限的几种,其按 2 的倍数,将内存块分为 8、16、32、64……字节,当申请的字节数大于 8 小于等于 16 时,就会使用 16 字节的内存块,以此类推。

(3) 将页面中分为空闲页、半满页、全满页。全满页脱离链表,分配内存时不应再访问到它。空闲页是超然的,如果这个页面曾经为 32 字节的内存块服务,在它又成为空闲页时,下次便可以为 128 字节的内存块服务。因此,所有的空闲页会单独构成一个空闲页链表。

(4) 让有限的几种页面构成链表,且各链表按序保存在数组中,使得用直接寻址法就可以快速找到。这里,slots 数组采用散列表的思想,用快速的直接寻址方式将半满页展现在使用者面前。

(5) 这里不仅支持小于 4KB 的内存块分配,也支持大于 4KB 的内存分配请求。

3.3 Nginx Slab 内存布局

如图 2 中所示,Slab 管理机制将内存大体上分为 ngx_slab_pool_t 结构体、slots 数组、page 数组、由于对齐被浪费的空间、数据区等模块,其中各模块的功能和作用如下:

(1) ngx_slab_pool_t 结构体: 包含 Slab 管理的汇总信息,在内存的管理过程中,内存的分配、回收、定位等操作都依赖于这些数据,具体含义见代码 1 注释。

(2) slots 数组: slots 数组各成员分别负责固定大小的内存块的分配和回收。在 Nginx 中 slots [0] – slots [8] 分别负责区间在 [1 – 8]、[9 – 16]、[17 – 32]、[33 – 64]、[65 – 128]、[129 – 256]、[257 – 512]、[513 – 1024]、[1025 – 2048] 字节大小内存的分配,但为方便内存块的分配和回收,每个内存块的大小为各区间的上限 (8、16、32、64、128、256、512、1024、2048) 。假如应用进程请求申请 5 个字节的空间,因 5 处在 [1 – 8] 的区间内,因此由 slots [0] 负责该内存的分配,但区间 [1 – 8] 的上限为 8,因此即使申请 5 个字节,却依然分配 8 字节给应用进程,以此类推。

(3) page 数组: page 数组各成员分别负责可分配空间中各页的查询、分配和回收。数组中每个元素对应一个页面。

(4) 数据区: 真正分配给用户的内存,用于存储数据,这块内存才是真正使用的内存。内存池中的其它数据都是用于管理这块内存的。Slab 在逻辑上将数据区划分成多个大小为 4K 的内存页,每页内存与 page 数组成员一一对应,由 page 数组各成员负责各内存页的分配和回收。

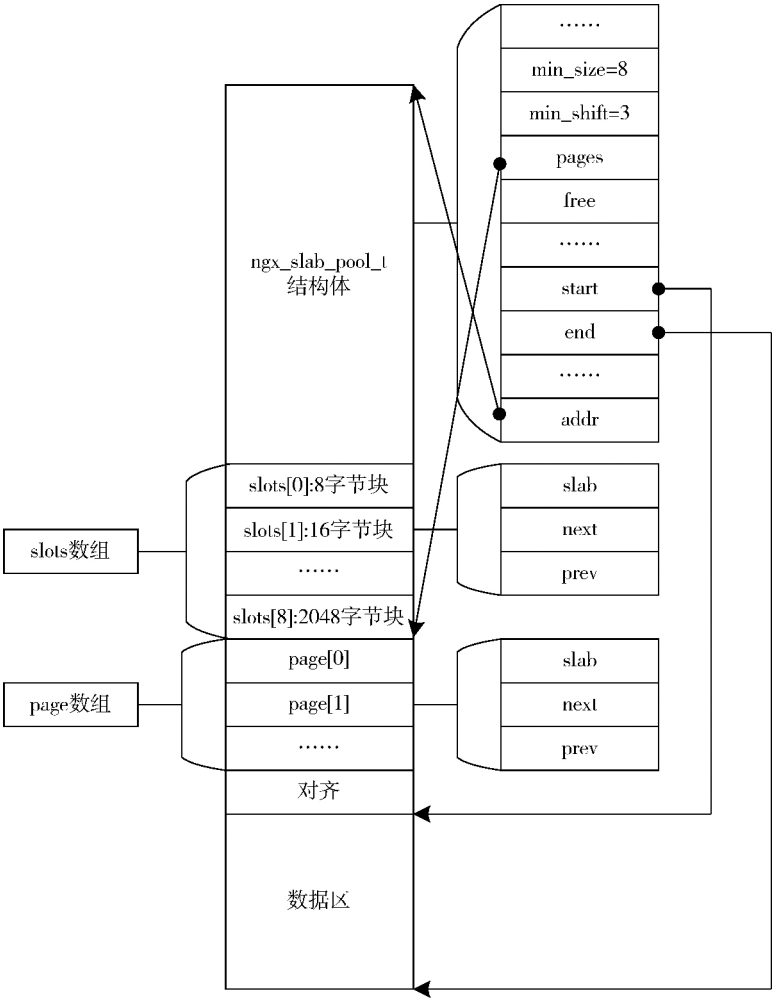

图 2　一个 Nginx Slab 共享内存池中的内存布局

(5) 由于对齐被浪费的空间: 按照每页 4K 的大小对空间进行划分时,满足 4K 的空间,将作为可分配空间被 page 数组进行管理,而最后剩余的不足 4K 的内存将会被舍弃,也就是被浪费了。

其中,slots 数组和 page 数组成员都是由 ngx_slab_page_t 结构来表示的,ngx_slab_page_t 结构有 3 个变量: slab、pre 和 next。这 3 个变量在分配四种不同类型内存块时意义如表 1 所示。

3.4　Nginx Slab 源码分析

说明: 本分析是基于 2015 – 06 – 16 发布的 Nginx – 1.9.2 版本代码。其具体代码在/src/core/ngx_slab.c 文件中,对应的头文件/src/core/ngx_slab.h。

3.4.1　ngx_slab.h 头文件

ngx_slab.h 头文件中定义了两个重要的数据结构以及声明了几个函数原型。

(1) 代码 1: ngx_slab_pool_t 结构体。

```
typedef struct {    /* 整个内存区的管理结构* /
size_t min_size;  /*  最小分配单元,默认为 8 字节 * /
size_t min_shift;  /*  最小分配单元对应的位移,默认为 3 * /
ngx_slab_page_t * pages;  /*  页数组,指向数组首地址 * /
ngx_slab_page_t free;  /*  空闲页链表 * /
u_char * start;        /*  可分配空间的起始地址 * /
u_char * end;          /*  内存块的结束地址 * /
...                    /*  其他变量成员( 省略) * /
} ngx_slab_pool_t
```

(2) 代码 2: ngx_slab_page_t 结构体。

```
typedef struct ngx_slab_page_s  ngx_slab_page_t;
/* 用于表示 page 数组单元和 slots 数组单元* /
struct ngx_slab_page_s {
uintptr_t               slab;
ngx_slab_page_t  * next;
uintptr_t               prev;
};
```

表 1 四类内存中 ngx_slab_page_t 的各成员意义

	slab	next	prev
小块内存	该页面存放的等长内存块的大小	指向双向链表中的下一个元素,如果不在双向链表中,则为 0	低 2 位为 11,NGX_SLAB_SMALL
中块内存	作为 bitmap 表示页上内存块是否被使用		低 2 位为 10,NGX_SLAB_EXACT
大块内存	高位表示 bitmap,低位表示存放的内存块大小		低 2 位为 01,NGX_SLAB_BIG
超大内存	一页或多页中的第一页的 slab 前 3 位设为 NGX_SLAB_PAGE_START,其余页设为 NGX_SLAB_PAGE_BUSY		低 2 位为 00,NGX_SLAB_PAGE

3.5 函数声明

(1) 初始化函数。

```
void ngx_slab_init( ngx_slab_pool_t * pool)
```

(2) 内存分配函数。

```
void * ngx_slab_alloc( ngx_slab_pool_t * pool, size_t size) ;
void * ngx_slab_alloc_locked( ngx_slab_pool_t * pool, size_t size) ;
/* locked 指在调用该函数前进行加锁保护 * /
void * ngx_slab_calloc( ngx_slab_pool_t * pool, size_t size) ;
void * ngx_slab_calloc_locked( ngx_slab_pool_t * pool, size_t size) ;
/* alloc 和 calloc 的区别在于是否在分配的同时将内存清零 * /
```

(3) 内存释放函数。

```
void ngx_slab_free( ngx_slab_pool_t * pool, void * p) ;
void ngx_slab_free_locked( ngx_slab_pool_t * pool, void * p) ;
```

3.5.1 ngx_slab.c 源文件

分析本文所使用的模型:

考虑 32 位机器,uintptr_t 占 4 字节,slab 共 32 位,即可以表示 32 个块的 bitmap。如果块数多于 32,使用页数据空间的开始几个 uintptr_t 空间来表示 bitmap。

页大小 ngx_pagesize = 4KB,ngx_pagesize_shift = 12

最小分配单元 min_size = 8Bytes,min_shift = 3

ngx_slab_max_size = ngx_pagesize/2 = 4KB/2 = 2KB

ngx_slab_exact_size = ngx_pagesize / (8* sizeof(uintptr_t)) = 4KB/(8* 4) = 128Bytes

ngx_slab_exact_shift = 7

此时,分级数为 pagesize_shift - min_shift = 9,即对应 ngx_slab_init( ) 中的 n 为 9,即 slots 数组中有 9 个元素。

3.5.2 初始化新创建的 Slab 共享内存池

```
void ngx_slab_init( ngx_slab_pool_t * pool) ;
```

初始化 Slab 共享内存池: 初始化所用模型中的各个参数,以及将 ngx_slab_pool_t 结构体中各个变量初始化为图 2 所示。其中,进行了将实际的页起始地址对齐到 pagesize 的操作。

3.5.3　加锁保护的内存分配函数

```
void *  ngx_slab_alloc_locked( ngx_slab_pool_t *  pool, size_t size) ;
```

对于给定 size,从内存池 pool 中分配内存。

Slab 中将不等长的内存大小 size 分为如表 1 中 4 个大类,分别进行如下处理。

(1) 如果 size > = ngx_slab_max_size,即超大内存,则:

计算出需要多少个连续页面,从 free 空闲链表中进行查找,如果有则直接返回内存块首地址,没有则返回 NULL 表示失败。

(2) 如果 size < ngx_slab_max_size,计算出此 size 的移位数 shift 以及此 size 对应的 slots 数组,并且若链表中有空余 slot 块。

i. 如果 size > ngx_slab_exact_size,大块内存,则:

设置 slab 的高 16 位存放 slot 对应的 bitmap,并且该 bitmap 的低位对应页面中高位的 slot 块。slab 的低 16 为存储 slot 块大小的位移 shift。

设置 page 数组单元的 prev 为 NGX_SLAB_BIG。

若页面中的全部 slot 块都被使用,则将此页面从 slots 数组元素的链表中移除。

ii. 如果 size = = ngx_slab_exact_size,中块内存,则:

设置 slab 存储 slot 的 bitmap。

设置 prev 为 NGX_SLAB_EXACT。

若全部 slot 块都被使用,则从链表中移除。

iii. 如果 size < ngx_slab_exact_size,小块内存,则:

计算需要多少 map 来存放 bitmap,用页面中的前几个 slot 块存放 bitmap。

设置 prev 为 NGX_SLAB_SMALL。

若全部 slot 块都被使用,则从链表中移除。

(3) 如果 size < ngx_slab_max_size,计算出此 size 的移位数 shift 以及此 size 对应的 slots 数组,并且链表中没有空余的 slot 块。

首先,在 free 空闲链表中找到一个空闲的页面分配给 slots 数组元素中的链表。其次,根据大块内存、中块内存、小块内存区别设置 slab、prev 等。最后,将分配的页面链入 slots 数组元素的链表中。

(4) 查找成功则返回分配的内存块首地址,即指针 p; 否则失败,返回 NULL。

3.5.4　加锁保护的内存释放函数

```
void   ngx_slab_free_locked( ngx_slab_pool_t *  pool, void *  p) ;
```

根据给定的指针 p,释放内存池 pool 中相应内存块。

(1) 找到 p 对应的内存块和对应的 pages 数组元素。

(2) 根据 pages 数组元素的 prev 确定页面类型。如果页面为小块内存、中块内存或大块内存,则根据表 1 四种意义找到对应 slot 块并设置为可用,并且,如果整个页面都可用,则调用 ngx_slab_free_pages(pool, page, 1) 将该页面归入 free 空闲链表中; 如果页面为超大内存,则计算需要归还页面的个数 size,并调用 ngx_slab_free_pages(pool, &pool - >pages[n], size) 将该 size 个页面归入 free 空闲链表中。

3.5.5　页面分配函数

```
static ngx_slab_page_t *  ngx_slab_alloc_pages( ngx_slab_pool_t *  pool, ngx_uint_t pages) ;
```

从内存池 pool 中分配 pages 个页面,并将页面从 free 中摘除,返回对应页面的 ngx_slab_page_t 管理结构。

(1) 遍历 free 空闲链表,不断循环,若存在符合 page 数组元素的 slab > = pages,则说明存在符合条件的连续 pages 个页面。

(2) 若 slab > pages,正常分配,并更新 slab = slab - pages,让下次可以从 page[pages]开始分配的页的

next 和 prev 指向 pool - >free,只要页的 next 和 prev 指向了 free,则表示可以从该页开始分配。

(3) 若 slab = = pages,分配后再没有页面,直接将 page 数组移出 free 链表。

(4) 如果分配了连续多个页面,后续的 page 数组元素需要初始化: slab = NGX_SLAB_PAGE_BUSY; next = NULL; prev = NGX_SLAB_PAGE。

(5) 如果找到满足要求的连续 pages 个页面,则返回对应页面 ngx_slab_page_t 管理结构; 否则返回错误信息和 NULL。

3.5.6 *页面释放函数*

```
static void ngx_slab_free_pages( ngx_slab_pool_t * pool, ngx_slab_page_t * page,ngx_uint_t pages) ;
```

从内存池 pool 中释放 page 页开始的 pages 个页面,并将页面重新链入 free 中。

(1) 判断 pages 是否是多页,如果多页,对后面的页面的 ngx_slab_page_t 进行清空处理。

(2) 如果要释放页面的对应的 page 数组单元还挂接在 slots 分级链表下,解除 slots 和 page 的关联。

(3) 使用 join = page + page - >slab; 如果要释放的页面与原 free 中可用页面是相邻的页面,进行合并。

(4) 将待释放的空闲页管理单元挂接到 free 链表的首部。

3.5.7 *出错处理函数*

```
static void ngx_slab_error( ngx_slab_pool_t * pool, ngx_uint_t level, char * text) {
ngx_log_error( level, ngx_cycle - >log, 0, "%s%s", text, pool - >log_ctx) ;
}
```

其余几个函数与所介绍函数的差别我们在函数声明的注释中已经说明,不再赘述。

4 总结和分析

4.1 Nginx Slab 算法总结

在 ngx_slab. c 源码中,有以下几个优点:

(1) 由于采用了内存对齐等机制,Nginx Slab 中的内存分配可以很好地实现“自我管理”,也就是说在给出起始地址的情况下能够方便快捷地正确推算出相关管理结构的位置和内容,而无需额外的存储结构。

(2) 使用 prev 的后两位,用于标记页面的类型,方便、高效。

(3) 充分利用计算机的 2 进制特点,代码中充斥着位移,效率非常高。

(4) 构造小的内存块(2 的幂次) 用于存储对象来避免内存碎片。

(5) 让有限的几种页面构成链表,且各链表按序保存在数组中,使得用直接寻址法就可以快速找到。

缺点: 由于使用 2 的幂次的方式构造内存块,如果请求内存位于两个幂次之间,会造成内部碎片。

4.2 Nginx Slab 与 Linux Slab 对比

(1) 代码量。Linux Kernel 2.6.22 版本,4519 行。Nginx 1.9.2 版本,790 行。

(2) 主要思想。

Linux slab: 基于对象进行管理,考虑复杂数据结构的初始化时间大于其分配和释放的时间,故而使用缓存对象的技术。

Nginx slab: 将一页分割为多个大小相同的内存块的分区,采用直接寻址法就可以快速找到,并且减少了内存浪费。

(3) 结构。

Linux slab: 使用缓存区、三种 slab 列表等,结构复杂、维护困难。

Nginx slab: 结构较简单。

总之,Nginx Slab 可以看成是对经典 Slab 算法的学习,并根据 Nginx 应用程序的具体特点进行了一定的精简和改变,其通过构造小的内存块来避免内存碎片、使用链表方式连接有限的页面来提高分配速率,从而

帮助快速实现多种对象间的跨 Nginx worker 进程通信。

5 结束语

本文首先介绍经典的 Linux 内核的 Slab 算法,然后着重从应用场景、技术原理、内存布局、源码分析等方面详细介绍 Nginx 使用 Slab 算法,最后对其进行总结,并与 Linux Slab 算法对比,对 Nginx Slab 算法的学习以及进一步改进有着重要意义。

参 考 文 献

[1] 苗泽. Nginx 高性能 Web 服务器详解[M]. 北京:电子工业出版社, 2013.
[2] Uresh Vahalia. UNIX Internals: The New Frontiers[M]. Upper Saddle River, NJ: Prentice Hall, 1995.
[3] Tais B. Ferreira, Rivalino Matias, Autran Macedo, et al. An Experimental Comparison Analysis of Kernel – level Memory Allocators[J]. Acm Symposium on Applied Computing, 2015, 13(4): 2054 – 2059.
[4] Taís Borges Ferreira, Márcia Aparecida Fernandes, Rivalino Matias Jr. A Comprehensive Complexity Analysis of User – level Memory Allocator Algorithms[C]. 2012 Brazilian Symposium on Comporting System Engineering. Uberlandia, Brazil, 2012: 99 – 104.
[5] I. Puaut. Real – Time Performance of Dynamic Memory Allocation Algorithms[C]. 4th Euromicro Conference on Real – time Systems. IEEE, 2002, 17(2): 41.
[6] Jiayang Yu, Ruonan Rao. A Method for Solving the Performance Isolation Problem in PaaS Based on Forecast and Dynamic Programming[C]. Fourth International Conference on Computational and Information Sciences. IEEE, 2012: 947 – 950.
[7] 陈君, 樊皓, 吴京洪. 基于 TLSF 算法改进的动态内存管理算法研究[J]. 网络新媒体技术,2016, 5(3): 55 – 60.
[8] 陶辉. 深入理解 Nginx[M]. 北京:电子工业出版社, 2013.
[9] TIAN Ling – ping. Research on Memory Management in Embedded Operating System[J]. Computer Knowledge and Technology, 2006, 11: 161 – 163.
[10] Jeff Bonwick. The Slab Allocator: An Object – Caching Kernel Memory Allocator[C]. USENIX Summer. Sun Microsystems, 1994: 87 – 98.
[11] Bovet, D. P., Cesati. Understanding the Linux Kernel (3rd Edition)[M]. Sebastopol, CA: O'Reilly Media, Inc, 2006.
[12] Negus, C. Linux Bible(8th Edition)[M]. Indianapolis, IN: John Wiley & Sons, Inc, 2012.
[13] 刘建君. 嵌入式 Linux 内存管理机制的研究[D]. 沈阳:沈阳工业大学, 2011.
[14] Michael B. Crouse, H. T. Kung. Nested Buddy System: A New Block Address Allocation Scheme for ISPs and IaaS Providers[C]. 8th International Conference on Cloud Computing Technology and Science. IEEE, 2016: 51 – 58.
[15] Liu Song, Qin Xiao – jun. Parallelly Refill SLUB Objects Freed in Slow Paths: An Approach to Exploit the Use – After – Free Vulnerabilities in Linux Kernel[C]. 17th International Conference on Parallel and Distributed Computing, Applications and Technologies. IEEE, 2016: 387 – 390.

作者简介

宋雅琴,女,(1993.3 –),博士生, 研究方向:信号与信息处理、网络新媒体技术。
郭志川,男,(1975.1 –),博士,副研究员,研究方向:网络新媒体技术、智能终端技术。