# 字节对齐

## 常用操作

### 宏ngx\_align(d, a)

#define ngx\_align(d, a) (((d) + (a - 1)) & ~(a - 1))

作用是将d的值以a为单位，向上对齐到a的倍数。即将d以a字节为基准向上对齐。

举个例子说明，假设a=4，d=11。计算之后的结果是12，就是向上对齐到了a的整数倍。

# 内存池ngx\_pool\_s

## 数据结构

### ngx\_pool\_s

struct ngx\_pool\_s {

ngx\_pool\_data\_t d;

size\_t max;

ngx\_pool\_t \*current;

ngx\_chain\_t \*chain;

ngx\_pool\_large\_t \*large;

ngx\_pool\_cleanup\_t \*cleanup;

ngx\_log\_t \*log;

};

ngx\_pool\_s结构可以看成为内存池的标识，用来管理一块或者多块内存。

**d**：说明如下

typedef struct {

u\_char \*last;

u\_char \*end;

ngx\_pool\_t \*next;

ngx\_unit\_t failed;

} ngx\_pool\_data\_t;

last：内存空间的起始地址。

end：内存空间的结束地址。

next：指向下一块内存。在ngx\_palloc\_block中赋值。

failed：从当前内存空间分配内存失败的次数。

**max**：能够从d成员所管理的内存空间分配的内存大小最大值。如果实际需要分配的内存大小大于max，会被认为是大内存，大内存则由large成员管理。

**current**：ngx\_pool\_t当前所指向的内存空间。该字段在当前内存空间分配内存失败的时候发生变化。

**chain**：

**large**：用于大内存的管理。

**cleanup**：指向的数据结构是用来管理一种能由使用者自定义释放函数的内存。

**log**：

## 常用操作

### 创建内存池ngx\_create\_pool

入参size：内存池的大小。

根据size的大小申请一块内存p。这块内存包括了ngx\_pool\_t结构。因此实际可以用于分配的内存空间大小为size – sizeof(ngx\_pool\_t)。

max成员的值最大不能超过ngx\_pageszie – 1，如果size小于ngx\_pagesize – 1，则max = size。ngx\_pagesize在x86的机器上是4096。

current指向上面申请的内存p。

### 销毁内存池ngx\_destory\_pool

先根据用户自定义的函数处理cleanup内存池中的内存。cleanup内存本身是不需要在自定义函数中删除的。因为这块内存要么是从小内存分配的要么是从大内存分配的。接下来就是逐个删除大内存链表和小内存链表中的内存。

### 分配内存ngx\_palloc

入参pool：指定从哪个内存池中分配内存。

入参size：需要分配的内存大小。

size <= pool->max，**小内存分配ngx\_palloc\_small**：

1. 先判断current所指向的内存空间中的剩余空间是否满足所需分配的内存大小。
2. 满足，p->d.last经过字节对齐处理之后所指向的内存地址就是待分配内存的起始地址。返回给上层。
3. 不满足，则查找下一块内存的剩余空间是否满足要求。

什么时候会有下一块内存空间？

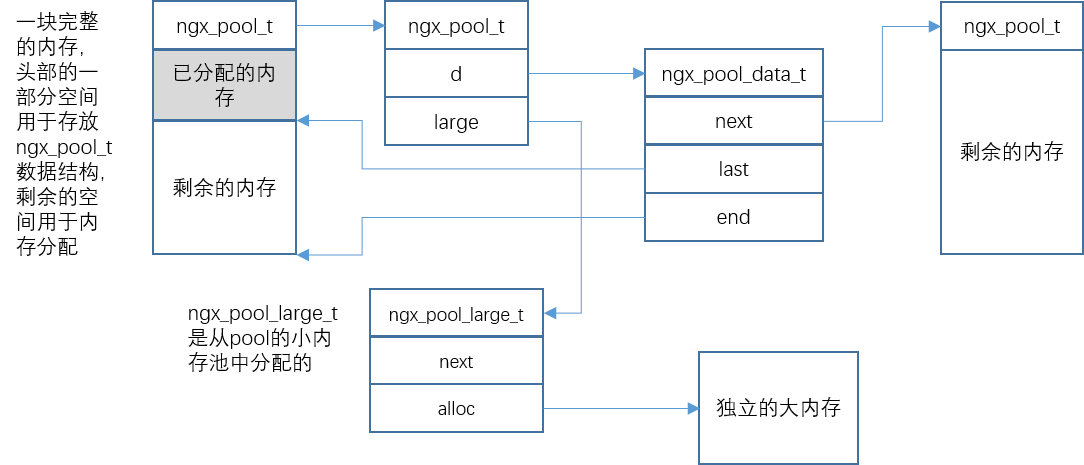
ngx\_pool\_t创建之初只管理了一个内存空间，当这个内存空间剩余大小不能满足分配内存的需要时，会调用**ngx\_palloc\_block**创建一块新的相同大小的内存空间new。新创建的内存new挂在current的next指针。

current什么是否发生变化？

如果current节点next已经挂载了超过4个ngx\_pool\_t节点，说明这些节点都无法满足内存申请的需求。此时就会将current指针指向本次新申请的new。

**大内存分配ngx\_palloc\_large**：

1. 根据需要分配的内存大小，直接从系统中申请一块内存。
2. 从小内存池中申请一个ngx\_pool\_large\_t结构，用来管理大内存。
3. 大内存的链表里如果有超过3个大内存空间，那么新申请的大内存就加入到链表头部。



### 分配自定义释放操作的内存ngx\_pool\_cleanup\_add

从入参指定的内存池分配一块ngx\_pool\_cleanup\_t用来管理

struct ngx\_pool\_cleanup\_s {

ngx\_pool\_cleanup\_pt handler;

void \*data;

ngx\_pool\_cleanup\_t \*next;

};

**handler**：指向由使用者设定的释放函数的函数指针。

**data**：指向入参指定大小的内存空间，这块内存可能是从小内存分配的也可能是从大内存分配的。

**next**：cleanup内存可以组成一个链表。

每次新申请的cleanup内存是加入在指定内存池cleanup链表头部。

### 释放分配的内存ngx\_pfree

该函数仅用来释放大内存。入参p为需要释放的内存，入参pool为申请内存时使用的内存池。函数实现很简单，遍历大内存的链表，找到对应的内存块删除即可。

这里有个值得注意的点是，仅仅是删除了从系统中申请的大内存，却没有删除用于管理大块内存的ngx\_pool\_large\_t结构，这个结构下次申请大内存时可以复用。

# 哈希表ngx\_hash\_t

nginx的哈希表不支持动态增加或者删除元素。哈希表中的所有成员都是在创建哈希表的时候就加入哈希表中的。也正是因为这个特点，nginx的哈希表在设计上尽量使用连续的内存空间存储哈希表的成员，从而减少了内存碎片的几率。

## 数据结构

### ngx\_hash\_t

typedef struct {

ngx\_hash\_elt\_t \*\*buckets;

ngx\_uint\_t size;

} ngx\_hash\_t;

**bucket**：哈希表中桶的数组。在ngx\_hash\_init中赋值，指向一块连续的内存，包含N个ngx\_hash\_elt\_t结构。其中N等于下面size的值。

**size**：哈希表中桶的数量。数量是在ngx\_hash\_init中计算得到的。

### ngx\_hash\_elt\_t

typedef struct {

void \*value;

u\_short len;

u\_char name[1];

} ngx\_hash\_elt\_t;

**value**：kv键值对中的value。

**len**：key的长度。

**name**：kv键值对中的key，变长数组

### ngx\_hash\_init\_t

typedef struct {

ngx\_hash\_t \*hash;

ngx\_hash\_key\_pt key;

ngx\_uint\_t max\_size;

ngx\_uint\_t bucket\_size;

char \*name;

ngx\_pool\_t \*pool;

ngx\_pool\_t \*temp\_pool;

} ngx\_hash\_init\_t;

**hash**：指向哈希表数据结构的指针，在初始化哈希表的时候动态分配。

**key**：

**max\_size**：哈希桶的最大数量。调用初始化哈希表函数之前就需要设定好。

**bucket\_size**：每个哈希表桶的大小，桶就是用来存放键值对。

**name**：哈希结构的名字，用于日志输出。

**pool**：哈希表申请内存所使用的内存池，在调用初始化哈希表函数之前就要设定好。

**temp\_pool**：

### ngx\_hash\_key\_t

typedef struct {

ngx\_str\_t key;

ngx\_uint\_t key\_hash;

void \*value;

} ngx\_hash\_key\_t;

**key**：kv键值对的key。

**key\_hash**：key对应的哈希值。

**value**：kv键值对的value。

## 常用操作

### 宏NGX\_HASH\_ELT\_SIZE

作用是用来计算ngx\_hash\_elt\_t结构大小，也就是计算一个键值对的大小。

#define NGX\_HASH\_ELT\_SIZE(name)

(sizeof(void\*) + ngx\_align((name)->key.len + 2, sizeof(void \*)))

第一个sizeof(void \*)对应的就是ngx\_hash\_elt\_t结构中的value成员的大小。

+2则是对应ngx\_hash\_elt\_t中len成员的大小。

name->key.len则是key的长度。ngx\_hash\_elt\_t结构中的name[1]成员实际上是一个动态数组。用来保存key。

ngx\_align在这里的作用则是将name和len成员的长度以4字节（32位系统是4字节，如果是64位就是8字节）对齐，实际上就是为了保证和value成员进行字节对齐。

也正是因为name[1]是动态数组，所以不能直接通过sizeof(ngx\_hash\_elt\_t)来计算该结构的大小。

### 哈希表初始化ngx\_hash\_init

入参hinit：需要被初始化的ngx\_hash\_init\_t结构。

入参names：存放了所有要加入哈希表的键值对

入参nelts：要加入哈希表的键值对数量。

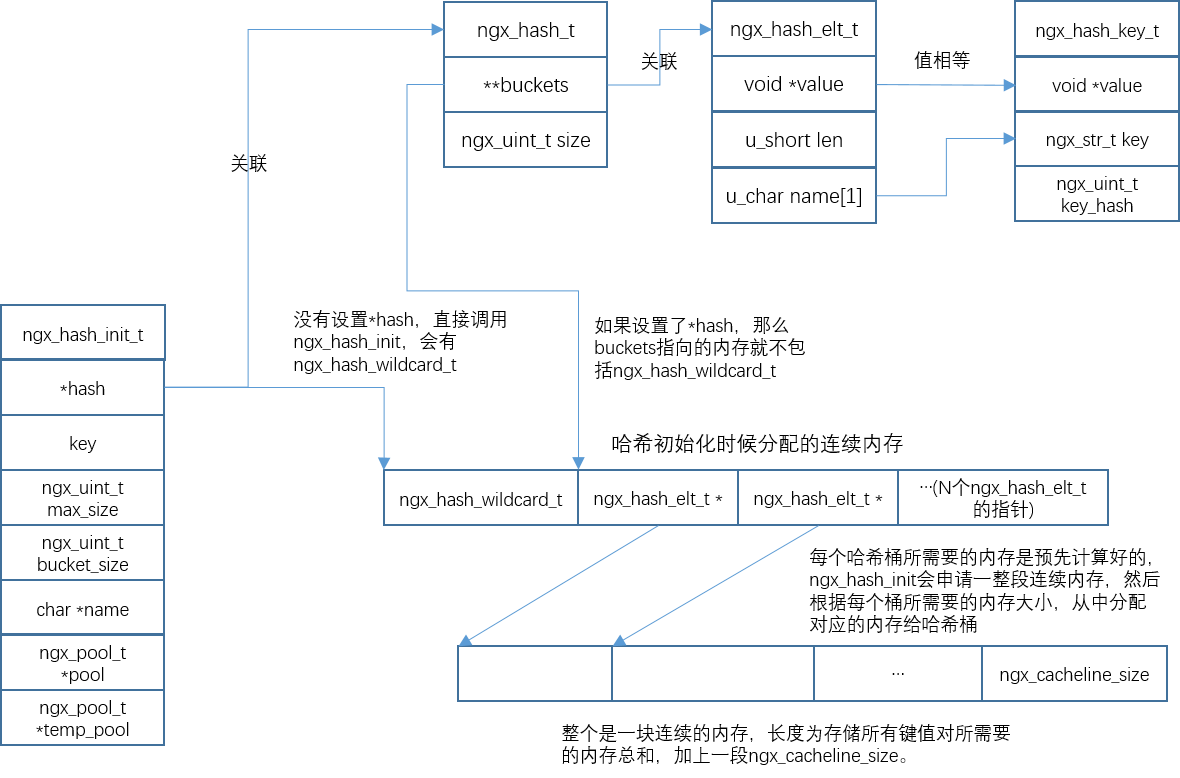
出参：NGX\_OK表示成功，NGX\_ERROR表示失败。

在调用ngx\_hash\_init之前，ngx\_hash\_init\_t中的max\_size是一定要设置的。

整个函数的实现可以总结为

1. 判断哈希桶的大小是否能够存放任意一个需要保存的键值对。
2. 找到能够保存所有键值对的哈希桶数量。
3. 计算每个哈希桶所需要的内存大小以及所有哈希桶所需的内存大小。
4. 分配哈希桶内存。
5. 将所有键值对存入哈希表中对应的哈希桶，并为每个保存了键值对的哈希桶加上一个值为NULL的成员，表示结束。

函数细节可以参考代码注释。



### 哈希表查找ngx\_hash\_find

入参hash：待查找的哈希表。

入参key：用于查找value的哈希值。

入参name：用于查找value的key值。

入参len：name的长度，name是一个变长数组。

出参：void类型指针，指向了value对应的内存。

用哈希值key对哈希桶总数取模，找到对应的哈希桶。哈希桶中可能存放了多个键值对。这时候就要遍历哈希桶通过name进行匹配找到对应的value。

### 计算哈希值ngx\_hash\_key

入参data：数组，就是键值对中的key。

入参len：data数组的长度。

出参：键值对对应的哈希值。

从data下标0开始，将data中的每个成员转换成整数之后 \* 31 + 下一个成员。循环这个过程。

# 动态数组ngx\_array\_t

## 数据结构

### ngx\_array\_t

typedef struct {

void \*elts;

ngx\_uint\_t nelts;

size\_t size;

ngx\_uint\_t nalloc;

ngx\_pool\_t \*pool;

} ngx\_array\_t;

**elts**：指向动态组数所使用内存的指针。

**nelts**：动态数组中已保存的成员个数。

**size**：动态数组中每个成员的大小。

**nalloc**：动态数组能够保存的成员个数。

**pool**：动态数组所使用的内存池。

## 常用方法

### ngx\_array\_create

入参p：用来创建动态数组的内存池。

入参n：动态数组初始能够容纳的成员个数。

入参size：数组成员的大小。

出参：ngx\_array\_t\*指向动态数组的指针。

1. 调用ngx\_palloc从内存池中分配一个ngx\_array\_t的数据结构，用来管理动态数组。
2. ngx\_array\_init对ngx\_array\_t数据结构进行初始化。
3. 根据入参传入的成员个数和每个成员的大小，创建动态内存。

### ngx\_array\_push

入参a：需要加入的成员的动态数组指针。

出参：void\*实际上指向用户保存新加入数据的数组内存指针。

1. 判断动态数组是否需要扩容。
2. 先说不需要扩容的情况，那就是直接进行指针便宜，将数组空闲内存的首指针返回。
3. 需要的扩容的话，则看数组使用的内存池还有多少剩余空间。如果动态数组是内存池中最近一块分配的内存，同时内存池又有剩余的空间，那么就不需要重新分配内存。直接在原来的内存空间上向后扩展即可。扩展的大小是一个数组成员的大小。
4. 如果不满足上面的条件，那就只能从内存池中重新分配一块新的内存，大小是当前动态数组内存大小的2倍。然后将当前数组中保存的数据拷贝到新的内存中。
5. 需要注意的是，分配新的内存之后，动态数组使用的旧内存并没有返回给内存池。

### ngx\_array\_destroy

入参a：需要销毁的动态数组。

出参：无。

1. 如果动态数组的内存是内存池最近分配的一块内存，那么直接通过偏移内存池的指针，就可以将内存返回给内存池。同时还会把ngx\_array\_t使用的内存返回给内存池。
2. 如果不满足上面的条件，那这个函数没有做任何处理，动态数组的内存并不会返回给内存池。