# nginx基础设施

## 内存池

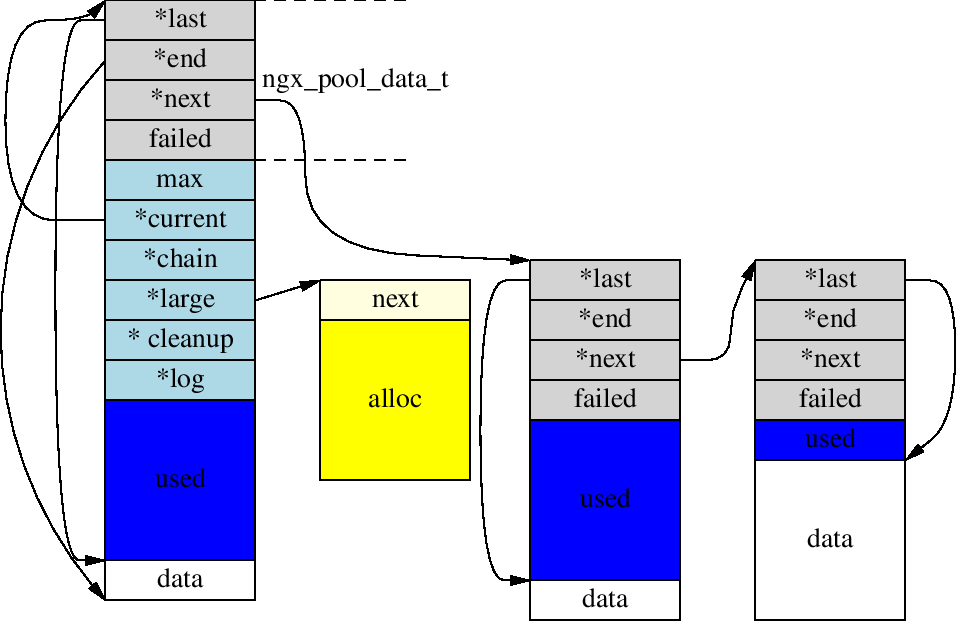
### 简介:

Nginx里内存的使用大都十分有特色:申请了永久保存,抑或伴随着请求的结束而全部释放,还有写满了缓冲再从头接着写.这么做的原因也主要取决于Web Server的特殊的场景,内存的分配和请求相关,一条请求处理完毕,即可释放其相关的内存池,降低了开发中对内存资源管理的复杂度,也减少了内存碎片的存在.

所以在Nginx使用内存池时总是只申请,不释放,使用完毕后直接destroy整个内存池.我们来看下内存池相关的实现。

### 结构:

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;  
};  
  
struct ngx\_pool\_large\_s {  
 ngx\_pool\_large\_t \*next;  
 void \*alloc;  
};  
  
typedef struct {  
 u\_char \*last;  
 u\_char \*end;  
 ngx\_pool\_t \*next;  
 ngx\_uint\_t failed;  
} ngx\_pool\_data\_t;



### 实现:

这三个数据结构构成了基本的内存池的主体.通过ngx\_create\_pool可以创建一个内存池,通过ngx\_palloc可以从内存池中分配指定大小的内存。

ngx\_pool\_t \*  
ngx\_create\_pool(size\_t size, ngx\_log\_t \*log)  
{  
 ngx\_pool\_t \*p;  
  
 p = ngx\_memalign(NGX\_POOL\_ALIGNMENT, size, log);  
 if (p == NULL) {  
 return NULL;  
 }  
  
 p->d.last = (u\_char \*) p + sizeof(ngx\_pool\_t);  
 p->d.end = (u\_char \*) p + size;  
 p->d.next = NULL;  
 p->d.failed = 0;  
  
 size = size - sizeof(ngx\_pool\_t);  
 p->max = (size < NGX\_MAX\_ALLOC\_FROM\_POOL) ? size : NGX\_MAX\_ALLOC\_FROM\_POOL;  
  
 p->current = p;  
 p->chain = NULL;  
 p->large = NULL;  
 p->cleanup = NULL;  
 p->log = log;  
  
 return p;  
}

这里首申请了一块大小为size的内存区域，其前sizeof(ngx\_pool\_t)字节用来存储ngx\_pool\_t这个结构体自身自身.所以若size小于sizeof(ngx\_pool\_t)将会有coredump的可能性。

我们常用来分配内存的有三个接口:ngx\_palloc，ngx\_pnalloc，ngx\_pcalloc。

分别来看下它们的实现：

void \*  
ngx\_palloc(ngx\_pool\_t \*pool, size\_t size)  
{  
 u\_char \*m;  
 ngx\_pool\_t \*p;  
  
 if (size <= pool->max) {  
  
 p = pool->current;  
  
 do {  
 m = ngx\_align\_ptr(p->d.last, NGX\_ALIGNMENT);  
  
 if ((size\_t) (p->d.end - m) >= size) {  
 p->d.last = m + size;  
  
 return m;  
 }  
  
 p = p->d.next;  
  
 } while (p);  
  
 return ngx\_palloc\_block(pool, size);  
 }  
  
 return ngx\_palloc\_large(pool, size);  
}  
  
  
void \*  
ngx\_pnalloc(ngx\_pool\_t \*pool, size\_t size)  
{  
 u\_char \*m;  
 ngx\_pool\_t \*p;  
  
 if (size <= pool->max) {  
  
 p = pool->current;  
  
 do {  
 m = p->d.last;  
  
 if ((size\_t) (p->d.end - m) >= size) {  
 p->d.last = m + size;  
  
 return m;  
 }  
  
 p = p->d.next;  
  
 } while (p);  
  
 return ngx\_palloc\_block(pool, size);  
 }  
  
 return ngx\_palloc\_large(pool, size);  
}  
  
  
void \*  
ngx\_pcalloc(ngx\_pool\_t \*pool, size\_t size)  
{  
 void \*p;  
  
 p = ngx\_palloc(pool, size);  
 if (p) {  
 ngx\_memzero(p, size);  
 }  
  
 return p;

}

ngx\_pcalloc其只是ngx\_palloc的一个封装，将申请到的内存全部初始化为0。

ngx\_palloc相对ngx\_pnalloc，其会将申请的内存大小向上扩增到NGX\_ALIGNMENT的倍数，以方便内存对齐，减少内存访问次数。

Nginx的内存池不仅用于内存方面的管理，还可以通过`ngx\_pool\_cleanup\_add`来添加内存池释放时的回调函数，以便用来释放自己申请的其他相关资源。

ngx\_pool\_cleanup\_t \*  
ngx\_pool\_cleanup\_add(ngx\_pool\_t \*p, size\_t size)  
{  
ngx\_pool\_cleanup\_t \*c;}  
c = ngx\_palloc(p, sizeof(ngx\_pool\_cleanup\_t));  
if (c == NULL) {  
 return NULL;  
}  
  
if (size) {  
 c->data = ngx\_palloc(p, size);  
 if (c->data == NULL) {  
 return NULL;  
 }  
  
} else {  
 c->data = NULL;  
}  
  
c->handler = NULL;  
c->next = p->cleanup;  
  
p->cleanup = c;  
  
ngx\_log\_debug1(NGX\_LOG\_DEBUG\_ALLOC, p->log, 0, "add cleanup: %p", c);  
  
return c;

}

从代码中可以看出，这些由自己添加的释放回调是以链表形式保存的，也就是说你可以添加多个回调函数来管理不同的资源。

## 共享内存

### slab算法

## buffer管理

### buffer重用机制

### buffer防拷贝机制

## chain管理

### chain重用机制

## aio原理

## 锁实现

## 基本数据结构

## 时间缓存

## 文件缓存

## log机制