## **1.Nginx内存管理介绍**

　　在C/C++语言程序设计中，通常由程序员自己管理内存的分配和释放，其方式通常是malloc(free)和new(delete)等API。这样做的缺点在于：由于所申请内存块的大小不定，当频繁使用时会造成大量的内存碎片从而降低性能。通常我们所使用的解决办法就是内存池。

　　什么是内存池呢？内存池就是在真正使用内存之前，先申请分配一定数量的、大小相等(一般情况下)的内存块留作备用。当有新的内存需求时，就从内存池中分出一部分内存块，若内存块不够再继续申请新的内存。而不是每次需要了就调用分配内存的系统API（如malloc）进行申请，每次不需要了就调用系统释放内存的API（如free）进行释放。这样做的一个显著优点是，使得内存分配效率得到提升。因此使用内存池的方式对程序所使用的内存进行统一的分配和回收，是当前最流行且高效的内存管理方法，能够在很大程度上降低内存管理的难度，减少程序的缺陷，提高整个程序的稳定性。

　　通过减少频繁的内存申请和释放可以提升效率很容易理解，那么内存池究竟是怎么提高程序的稳定性的呢？我们知道在C/C++语言中，并没有提供直接可用的垃圾回收机制，因此在程序编写中， 一个特别容易发生的错误就是内存泄露，对于运行时间短，内存需求小的程序来说，泄露一点内存除了影响程序运行效率之外可能并不会造成多大的问题。但是类似于Ngnix这样需要长期运行的web服务器程序来说，内存泄露是一件非常严重的灾难，这会使得程序由于内存耗尽而崩溃，重启之前不再能够提供相应的web服务。还有一种情况就是当内存分配与释放的逻辑在程序中相隔较远时，很容易发生内存被释放两次乃至多次的情况。使用内存池使得我们在开发程序时，只用关心内存的分配，而释放就交给内存池来完成。

　　那么内存池在Nginx中究竟是怎么使用的呢？通常我们对于每个请求或者连接都会建立相应的内存池，建立好内存池之后，我们可以直接从内存池中申请所需要的内存，而不用去管内存的释放，唯一需要注意的就是当内存池使用完成之后需要记得销毁内存池。此时，内存池会调用相应的数据清理函数（如果有的话），之后会释放在内存池中管理的内存。

　　大家可能会问，既然申请的内存在内存池销毁的时候才会被释放，这不会存在内存的浪费么？毕竟使用完了不再需要的内存为什么不立即释放而非要等到销毁内存池时才释放呢？确实存在这个问题，不过大家不用担心。在Nginx中，对于大块内存可以使用ngx\_pfree()函数提前释放。并且由于Nginx是一个纯粹的web服务器，而web服务器通常使用的协议是Http协议，并且在传输层使用的是Tcp协议，我们知道每一个tcp连接都是由生命周期的，因此基于tcp的http请求都会有一个很短暂的生命周期。对于这种拥有很短暂生命周期的请求，我们所建立的内存池的生命周期也相应会很短暂，因此其所占用的内存资源很快就可以得到释放，不会出现太多的资源浪费的问题。毕竟工程就是一种折中嘛，我们需要在内存资源浪费和减低程序内存管理难度、提升效率之间选择一个合适的权衡。

　　说了这么多，现在就让我们开始研究和学习Nginx内存管理的机制和源码吧。注：本文的讲解都是基于nginx-1.10.3版本。

## **2.Nginx内存池的逻辑结构**

　　前面提到Nginx内存管理机制其实就是内存池，其底层实现就是一个链表结构。我们需要对内存池进行管理和分配，依赖的就是ngx\_pool\_t结构体，可以认为该结构就是内存池的分配管理模块。那么内存池的逻辑结构究竟是什么样呢？其实就是一个ngx\_pool\_t结构体，在这个结构体中包含了三个部分：小块内存形成的单链表，大块内存形成的单链表和数据清理函数形成的单链表。先给出一张整个内存池内部实现的结构图，方便大家理解。具体如图2.1所示：

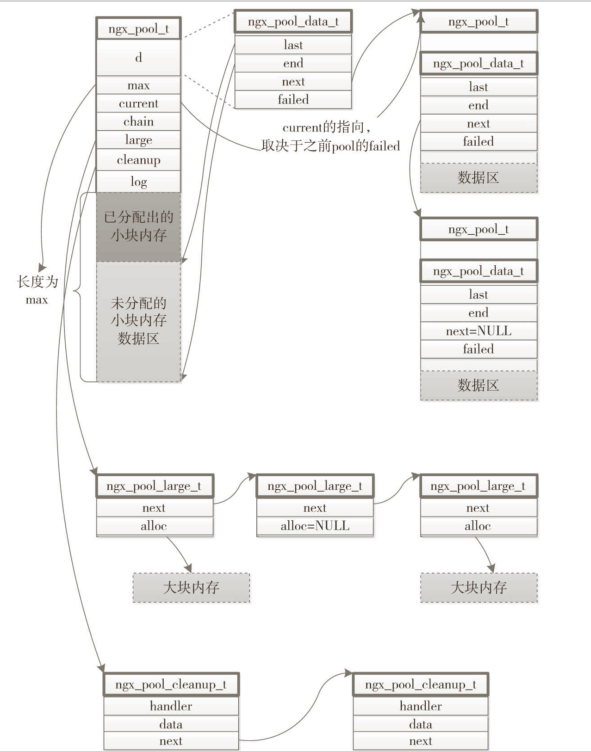


图2.1 Nginx内存池示意图

 　　图2.1完整的展示了ngx\_pool\_t内存池中小块内存、大块内存和资源清理函数链表间的关系。图中，内存池预先分配的剩余空闲内存不足以满足用户申请的内存需求，导致又分配了两个小内存池。其中原内存池的failed成员已经大于4，所以current指向了第2块小块内存池，这样当用户再次从小块内存池中请求分配内存空间时，将会直接忽略第1块小内存池，从第2块小块内存池开始遍历。从这里可以看到，我们使用的内存池确实存在当failed成员大于4之后不能利用其空闲内存的资源浪费现象（由于current指针后移）。值得注意的是：我们的第2、3块小块内存池中只包含了ngx\_pool\_t结构体和数据区，并不包含max、current、...、log。这是由于后续第1块小内存池已经包含了这些信息，后续的小块内存池不必在浪费空间存储这些信息。我们在第6小节：内存池的使用中将会有所介绍。图中共分配了3个大块内存，其中第二块的alloc为NULL（提前调用了ngx\_pfree（））。图中还挂在了两个资源清理方法。提醒一下的是：如果在这里没有弄清楚，没有关系，看完了后面的部分再回过头来理解这个示意图就能够很好的理解了。这里只是先给出一个概括性的Nginx内存池逻辑结构的介绍，先给大家留下一个大概的印象。

## **3.Nginx内存池的基本数据结构**

本部分主要介绍内存池中重要的数据结构，主要是ngx\_pool\_t，然后介绍ngx\_pool\_t中三个重要数据结构：ngx\_pool\_data\_t,ngx\_pool\_large\_t和ngx\_pool\_cleanup\_t。

### **（1）ngx\_pool\_t**

　　我们可以在Nginx的源码的src/core/目录下的nax\_palloc.h头文件中看到：

[IMG_257](https://www.cnblogs.com/uestcjoel/p/javascript:void(0);)

1 struct ngx\_pool\_s {2 ngx\_pool\_data\_t d;3 size\_t max;4 ngx\_pool\_t \*current;5 ngx\_chain\_t \*chain;6 ngx\_pool\_large\_t \*large;7 ngx\_pool\_cleanup\_t \*cleanup;8 ngx\_log\_t \*log;9 };

[IMG_258](https://www.cnblogs.com/uestcjoel/p/javascript:void(0);)

 　　并且在src/core/ngx\_core.h中：

|  |  |
| --- | --- |
| 1 | typedef struct ngx\_pool\_s        ngx\_pool\_t; |

下面将具体讲解ngx\_pool\_t结构体中每个成员的含义和用途：

 d:ngx\_pool\_data\_t结构体，描述内存池中的小块内存。当小块内存不足时，会再分配一个ngx\_pool\_t（里面含有一个新分配且未使用的小块内存空间和用于管理这块内存空间的ngx\_pool\_data\_t结构体）。这些小块内存块之间通过d中的next成员链接形成的单链表。挂在d成员上。

max:评估申请内存属于小块还是大块的标准，在x86上默认是4095字节。

current:多个小块内存构成单链表时，指向分配内存时遍历的第一个小块内存。

chain:与内存池关系不大，略过。

large:ngx\_pool\_large\_t结构体，当用户申请的内存空间大于max时，就会分配大块内存。而多个大块内存之间是通过ngx\_pool\_large\_t中的next成员链接形成的单链表。挂在large成员上。

cleanup:ngx\_pool\_cleanup\_t结构体，所有待清理的资源（例如需要关闭或者删除的文件）以ngx\_pool\_cleanup\_t对象中的next成员链接形成单链表。挂在cleanup成员上。

log:内存池中执行时输出日志的地方。

### **(a).ngx\_pool\_data\_t**

　　我们可以在Nginx的源码的src/core/目录下的nax\_palloc.h头文件中看到：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | typedef struct {      u\_char               \*last;      u\_char               \*end;      ngx\_pool\_t           \*next;      ngx\_uint\_t            failed;  } ngx\_pool\_data\_t; |

下面将具体讲解ngx\_pool\_data\_t结构体中每个成员的含义和用途：

last:指向小块内存中未分配的空闲内存的首地址。

end:指向当前小块内存的尾部。

next:同属于一个内存池的多个小块内存之间，通过next成员链接形成单链表。

failed:　每当当前的小块内存由于空闲部分较少而不能满足用户提出的内存申请请求时，failed成员就会加1。当failed成员大于4后，ngx\_pool\_t的current成员就会移向下一个小块内存，在以后分配内存时，将从下一个小块内存开始遍历。

### **(b).ngx\_pool\_large\_t**

　　我们可以在Nginx的源码的src/core/nax\_palloc.h头文件中看到：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | typedef struct ngx\_pool\_large\_s  ngx\_pool\_large\_t;    struct ngx\_pool\_large\_s {      ngx\_pool\_large\_t     \*next;      void                 \*alloc;  }; |

下面将具体讲解ngx\_pool\_large\_t结构体中每个成员的含义和用途：

next:所有大块内存通过next指针链接在一起形成单链表。

alloc:指向分配的大块内存，后面我们将会看到大块内存底层是通过ngx\_alloc分配，ngx\_free释放。释放完了之后赋值为NULL。

### **(c).ngx\_pool\_cleanup\_t**

 　　我们可以在Nginx的源码的src/core/nax\_palloc.h头文件中看到：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | typedef struct ngx\_pool\_cleanup\_s  ngx\_pool\_cleanup\_t;    struct ngx\_pool\_cleanup\_s {      ngx\_pool\_cleanup\_pt   handler;      void                 \*data;      ngx\_pool\_cleanup\_t   \*next;  }; |

 下面将具体讲解ngx\_pool\_cleanup\_t结构体中每个成员的含义和用途：

handler:初始化为NULL，需要设置的清理函数。

|  |  |
| --- | --- |
| 1 | typedef void (\*ngx\_pool\_cleanup\_pt)(void \*data); |

根据上面的声明，可以看出，ngx\_pool\_clean\_pt是一个函数指针，有一个通用型的参数data，返回类型为void。后面我们会看到当销毁内存池的时候，底层会遍历挂在cleanup成员上的单链表上的各个节点，调用各节点的数据清理函数完成相应的清理操作。这是通过回调函数实现的。

data:用于向数据清理函数传递的参数，指向待清理的数据的地址，若没有则为NULL。我们可以通过ngx\_pool\_cleanup\_add函数添加数据清理函数，当其中的参数size>0时，data不为NULL。

next:用于链接所有的数据清理函数形成单链表。由ngx\_pool\_cleanup\_add函数设置next成员，用于将当前ngx\_pool\_cleanup\_t(由ngx\_pool\_cleanup\_add函数返回)添加到cleanup链表中。

## **4.内存池基本操作介绍**

　　这一部分主要简单讲解与内存池管理有关的基本操作（共15个）。主要包括四个部分：（a）.内存池操作 (b).基于内存池的分配、释放操作 (3).随着内存池释放同步释放资源的操作 (4).与内存池无关的分配、释放操作。在第5和第6节中，我们会对部分常用内存池的操作进行代码上的详细介绍。

### **（a）.内存池操作：**

|  |  |
| --- | --- |
| 1  2  3 | ngx\_pool\_t \*ngx\_create\_pool(**size\_t** size, ngx\_log\_t \***log**);  void ngx\_destroy\_pool(ngx\_pool\_t \*pool);  void ngx\_reset\_pool(ngx\_pool\_t \*pool); |

　　ngx\_create\_pool

　　创建内存池，其参数size为整个内存的大小，包括结构管理（ngx\_pool\_t）和后续可分配的空闲内存。这意味着，size必须大于等于sizeof(ngx\_pool\_t)，通常在32位的系统是是40字节，后面我们介绍源码时会详细的介绍。通常size的默认大小为NGX\_DEFAULT\_POOL\_SIZE(#define NGX\_DEFAULT\_POOL\_SIZE    (16 \* 1024))，可以看到为16k。不用担心其不够用，因为当不够用时，Nginx会对内存池进行内存空间的扩展，也就是申请一个新的内存池（链表）节点（程序中成为一个block），然后挂在内存池的最后面。

　　ngx\_destory\_pool

　　销毁内存池，它会执行通过ngx\_pool\_cleanup\_add函数添加的各种资源清理方法，然后释放大块内存，最后把整个pool分配的内存释放掉。

　　ngx\_reset\_pool

　　重置内存池，即将在内存池中原有的内存释放后继续使用。后面我们会看到，这个方法是把大块的内存释放给操作系统，而小块的内存则在不释放的情况下复用。

### **(b).基于内存池的分配、释放操作**

|  |  |
| --- | --- |
| 1  2  3  4  5 | void \*ngx\_palloc(ngx\_pool\_t \*pool, **size\_t** size);  void \*ngx\_pnalloc(ngx\_pool\_t \*pool, **size\_t** size);  void \*ngx\_pcalloc(ngx\_pool\_t \*pool, **size\_t** size);  void \*ngx\_pmemalign(ngx\_pool\_t \*pool, **size\_t** size, **size\_t** alignment);  ngx\_int\_t ngx\_pfree(ngx\_pool\_t \*pool, void \*p); |

    　 ngx\_palloc

　　分配地址对齐的内存。内存对齐可以减少cpu读取内存的次数，代价是存在一些内存浪费。

　　ngx\_pnalloc

　　同ngx\_palloc，区别是分配内存时不考虑对齐。

　　ngx\_pcalloc

　　同ngx\_palloc，区别是分配完对齐的内存后，再调用memset全部初始化为0。

　　ngx\_pmemalign

　　按参数alignment进行地址对齐来分配内存。注意，这样分配的内存不管申请的size有多小，都不会使用小块内存，它们直接从进程的堆中分配，并挂在大块内存组成的large单链表中。

　　ngx\_pfree

　　提前释放大块内存。由于其实现是遍历large单链表，寻找ngx\_pool\_large\_t对应的alloc成员后调用ngx\_free(alloc)，实际上是直接调用free(alloc)，释放内存给操作系统，将ngx\_pool\_large\_t移出链表并删除。效率不高。

### **(c).随着内存池释放同步释放资源的操作**

|  |  |
| --- | --- |
| 1  2  3  4 | ngx\_pool\_cleanup\_t \*ngx\_pool\_cleanup\_add(ngx\_pool\_t \*p, **size\_t** size);  void ngx\_pool\_run\_cleanup\_file(ngx\_pool\_t \*p, ngx\_fd\_t fd);  void ngx\_pool\_cleanup\_file(void \*data);  void ngx\_pool\_delete\_file(void \*data); |

　　  ngx\_pool\_cleanup\_add

　　添加一个需要在内存释放时同步释放的资源。该方法会返回一个ngx\_pool\_cleanup\_t结构体，而我们得到该结构体后需要设置ngx\_pool\_cleanup\_t的handler成员为释放资源时执行的方法。ngx\_pool\_clean\_add的参数size，当它不为0时，会分配size大小的内存，并将ngx\_pool\_cleanup\_t的data成员指向该内存，这样可以利用这段内存传递参数，供资源清理函数使用。当size为0时，data将为NULL。

　　ngx\_pool\_run\_cleanup\_file

　　在内存释放前，如果需要提前关闭文件（调用ngx\_pool\_cleanup\_add添加的文件，同时ngx\_pool\_cleanup\_t的handler成员被设置为ngx\_pool\_cleanup\_file），则调用该方法。

　　ngx\_pool\_cleanup\_file

　　以关闭文件来释放资源的方法，可以设置到ngx\_pool\_cleanup\_t的handler成员。

　　ngx\_pool\_delete\_file

　　以删除文件来释放资源的方法，可以设置到ngx\_pool\_cleanup\_t的handler成员。

### **（d）.与内存池无关的分配、释放操作**

|  |  |
| --- | --- |
| 1 | void \*ngx\_alloc(**size\_t** size, ngx\_log\_t \***log**);void \*ngx\_calloc(**size\_t** size, ngx\_log\_t \***log**);#define ngx\_free          **free** |

　　这部分的声明和定义实际上并不在src/core/ngx\_palloc.h中，而是在/src/os/unix/ngx\_alloc.h中。

　　ngx\_alloc

　　从操作系统中分配内存，通过调用malloc实现。

　　ngx\_calloc

　　从操作系统中分配内存并全部初始化为0，通过调用malloc和memset实现。

　　ngx\_free

　　从上面的宏定义可以看到，其就是free函数，释放内存到操作系统。

## **5.内存池管理源码详解**

 　　本部分的源码可以在src/core/ngx\_palloc.h、src/core/ngx\_palloc.c、src/os/unix/ngx\_alloc.h和src/os/unix/ngx\_alloc.c中找到。内存池的管理主要包括内存池的创建、销毁以及重置操作。我们通过对源码的分析来研究和学习Nginx的内存管理技术。

### **（a）.内存池的创建**

　　创建内存池的操作主要由ngx\_create\_pool()函数完成，代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | 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;  } |

　　在这段代码中，首先通过ngx\_memalign()函数申请对齐的内存，其大小为size个字节。如果内存申请失败，则返回NULL，否则对ngx\_pool\_t结构体中的成员进行初始化。在进行初始化之前，让我们先讨论以下什么是小块内存？

|  |  |
| --- | --- |
| 1  2  3  4  5 | /\*   \* NGX\_MAX\_ALLOC\_FROM\_POOL should be (ngx\_pagesize - 1), i.e. 4095 on x86.   \* On Windows NT it decreases a number of locked pages in a kernel.   \*/  #define NGX\_MAX\_ALLOC\_FROM\_POOL  (ngx\_pagesize - 1) |

 　　这是ngx\_palloc.h中的一个注释及宏定义，从中我们可以看到在x86系统4095字节是一个标准。因为ngx\_pagesize中存放的是当前Nginx服务器运行的系统中一页内存页的大小，而在x86的系统上就是4KB。由于存在减1的关系，这意味着在x86系统上，小于等于4095字节的内存被称为小块内存，而大于4095字节的内存被称为大块内存。当然这并不是绝对的，在上述源码中，我们看到如果传递的参数size满足:size - sizeof(ngx\_pool\_t) < NGX\_MAX\_ALLOC\_FROM\_POOL时，其max的值为size（小于NGX\_MAX\_ALLOC\_FROM\_POOL），而当size不满足上述不等式时，其值为NGX\_MAX\_ALLOC\_FROM\_POOL。也就是说NGX\_MAX\_ALLOC\_FROM\_POOL是一个最大的门限，申请的小块内存的大小应该不超过其大小。在初始化max之后，我们将last指向分配好的空闲内存空间的首地址，end指向内存池的尾部。并将next初始化为NULL，failed的值初始化为0。然后再将current指向这块内存池的首地址，large和cleanup也被初始化为NULL，最后返回指向分配好的内存空间的首地址。为了更加清晰地展示内存池的创建过程，下面将会举一个例子来说明。但是在这之前，我们先来分析以下ngx\_memalign（）函数的实现源码。

　　关于ngx\_memalign()的细节我们可以在src/os/unix/ngx\_alloc.c中看到其源码，前面部分是声明，后面是定义。如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | /\*   \* Linux has memalign() or posix\_memalign()   \* Solaris has memalign()   \* FreeBSD 7.0 has posix\_memalign(), besides, early version's malloc()   \* aligns allocations bigger than page size at the page boundary   \*/    #if (NGX\_HAVE\_POSIX\_MEMALIGN || NGX\_HAVE\_MEMALIGN)    void \*ngx\_memalign(**size\_t** alignment, **size\_t** size, ngx\_log\_t \***log**);    #else    #define ngx\_memalign(alignment, size, log)  ngx\_alloc(size, log)    #endif |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | #if (NGX\_HAVE\_POSIX\_MEMALIGN)    void \*  ngx\_memalign(**size\_t** alignment, **size\_t** size, ngx\_log\_t \***log**)  {      void  \*p;  **int**    err;        err = posix\_memalign(&p, alignment, size);        if (err) {          ngx\_log\_error(NGX\_LOG\_EMERG, **log**, err,                        "posix\_memalign(%uz, %uz) failed", alignment, size);          p = NULL;      }        ngx\_log\_debug3(NGX\_LOG\_DEBUG\_ALLOC, **log**, 0,                     "posix\_memalign: %p:%uz @%uz", p, size, alignment);        return p;  }    #elif (NGX\_HAVE\_MEMALIGN)    void \*  ngx\_memalign(**size\_t** alignment, **size\_t** size, ngx\_log\_t \***log**)  {      void  \*p;        p = memalign(alignment, size);      if (p == NULL) {          ngx\_log\_error(NGX\_LOG\_EMERG, **log**, ngx\_errno,                        "memalign(%uz, %uz) failed", alignment, size);      }        ngx\_log\_debug3(NGX\_LOG\_DEBUG\_ALLOC, **log**, 0,                     "memalign: %p:%uz @%uz", p, size, alignment);        return p;  }    #endif |

　　我们还需要知道的就是在linux系统下，分配内存有三个系统调用，如果不考虑内存对齐，则有malloc()；如果考虑内存对齐，则有：memalign()和posix\_memalign()；从ngx\_memalign()的具体声明和实现中，我们可以看出这其实一个条件编译。如果系统定义了NGX\_HAVE\_POSIX\_MEMALIGN，则调用posix\_memalign()申请对齐的内存；如果系统定义了NGX\_HAVE\_MEMALIGN，则调用memalign()申请对齐的内存；并且这两种内存对齐默认都是基于16字节的。否则直接调用ngx\_alloc()，而ngx\_alloc()直接调用malloc()申请不对齐的内存。讲完了内存池中三种申请内存的方式之后，我们可以开始讲解创建内存池的实例了。

　　比如说我们需要创建一个大小为1024字节的内存池作为一个分配模块：

|  |  |
| --- | --- |
| 1 | ngx\_pool\_t \*pool = ngx\_create\_pool (1024,  **log**); |

　　为了方便，我们不妨假设申请的这块内存的起始地址为10。执行完创建内存池的操作后，内存中的分布情况如图5.1所示：

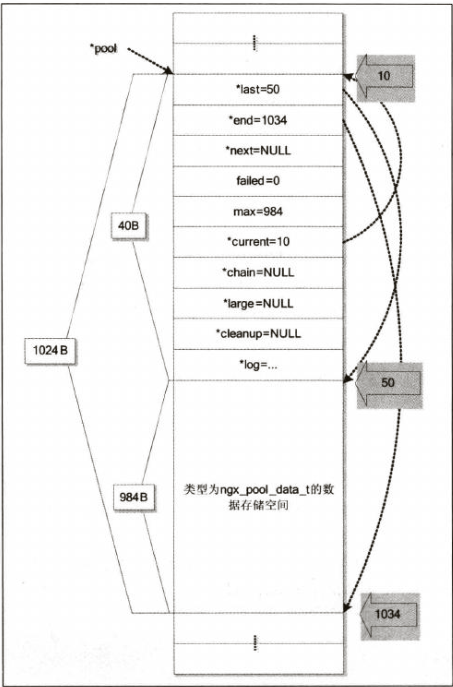


图5.1 创建内存池内存片段图

　　从执行结果可以看出：创建的内存池总共占用了1024个字节，起始地址为10，结束地址为1034。指向内存池的指针为pool。last指针为50（10+40），因为起始地址是10，而ngx\_pool\_t结构体所占用的内存空间为40字节，怎么计算得到的呢？其实很简单，只需要考虑结构体在内存中的对齐问题即可。在x86中（x64中指针在内存中占用8字节而不是4字节）如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | typedef struct {      u\_char               \*last;//4字节      u\_char               \*end;//4字节      ngx\_pool\_t           \*next;//4字节      ngx\_uint\_t            failed;//4字节  } ngx\_pool\_data\_t;      struct ngx\_pool\_s {      ngx\_pool\_data\_t       d;//16字节  **size\_t**                max;//4字节      ngx\_pool\_t           \*current;//4字节      ngx\_chain\_t          \*chain;//4字节      ngx\_pool\_large\_t     \*large;//4字节      ngx\_pool\_cleanup\_t   \*cleanup;//4字节      ngx\_log\_t            \***log**;//4字节  }; |

　　我们可以计算得到，在x86的系统中ngx\_pool\_t结构体各个成员变量占用的空间为40字节。因此last的值为50。end的值为10+1024=1034。max的值为1024-40=984。current=10。可以看到：

在物理内存中，申请到的内存空间被分为了两部分，前面一部分是ngx\_pool\_t内存管理结构各个成员变量所占用的空间，此处为40字节。后面部分的984字节的空闲空间才是我们可以在后续的程序中真正可以利用的，用来存放数据的。以上就是Nging内存池创建的主要原理和具体实现。

### **（b）.内存池的销毁**

　　销毁内存池的工作主要由ngx\_destroy\_pool()函数完成。代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | void  ngx\_destroy\_pool(ngx\_pool\_t \*pool)  {      ngx\_pool\_t          \*p, \*n;      ngx\_pool\_large\_t    \*l;      ngx\_pool\_cleanup\_t  \*c;        for (c = pool->cleanup; c; c = c->next) {          if (c->handler) {              ngx\_log\_debug1(NGX\_LOG\_DEBUG\_ALLOC, pool->**log**, 0,                             "run cleanup: %p", c);              c->handler(c->data);          }      }    #if (NGX\_DEBUG)        /\*       \* we could allocate the pool->log from this pool       \* so we cannot use this log while free()ing the pool       \*/        for (l = pool->large; l; l = l->next) {          ngx\_log\_debug1(NGX\_LOG\_DEBUG\_ALLOC, pool->**log**, 0, "free: %p", l->alloc);      }        for (p = pool, n = pool->d.next; /\* void \*/; p = n, n = n->d.next) {          ngx\_log\_debug2(NGX\_LOG\_DEBUG\_ALLOC, pool->**log**, 0,                         "free: %p, unused: %uz", p, p->d.end - p->d.last);            if (n == NULL) {              break;          }      }    #endif        for (l = pool->large; l; l = l->next) {          if (l->alloc) {              ngx\_free(l->alloc);          }      }        for (p = pool, n = pool->d.next; /\* void \*/; p = n, n = n->d.next) {          ngx\_free(p);            if (n == NULL) {              break;          }      }  } |

　　我们可以看到，销毁内存池的主要步骤为：先通过遍历挂在cleanup上数据清理函数链表，通过回调函数handler做相应的数据清理；中间输出部分只与调试程序相关，可忽略。然后遍历挂在large上的大块内存链表，调用ngx\_free()函数释放节点所占的大块内存空间；最后，遍历挂在d->next上的小块内存池链表，释放小块内存池（包括管理结构和数据区）占用的空间，在这一步中，我们首先清理了第一块ngx\_pool\_t（包括了large、cleanup等成员）代表的小块内存池，然后再清理剩下的其他小块内存池。经过以上三个过程，就可以完成数据清理、释放整个内存池占用的内存空间，并销毁内存池。需要注意的是：由于内存池的结构，我们必须最后清理管理结构ngx\_pool\_t（第一块小块内存池），因为如果先清理第一块ngx\_pool\_t代表的内存池的话，我们就找不到挂在large和cleanup上的单链表了，因为我们清理了其单链表的第一个节点。

### **(c).内存池的重置**

　　重置内存池，就是将内存池分配到初始分配的状态。这是由ngx\_reset\_pool()函数完成的。代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | void  ngx\_reset\_pool(ngx\_pool\_t \*pool)  {      ngx\_pool\_t        \*p;      ngx\_pool\_large\_t  \*l;        for (l = pool->large; l; l = l->next) {          if (l->alloc) {              ngx\_free(l->alloc);          }      }        for (p = pool; p; p = p->d.next) {          p->d.last = (u\_char \*) p + sizeof(ngx\_pool\_t);          p->d.failed = 0;      }        pool->current = pool;      pool->chain = NULL;      pool->large = NULL;  } |

　　我们可以看到，重置内存池十分简单。首先将挂在large上的大块内存链表上的各个节点释放掉，并将pool->large赋值为NULL。之后，将所有小块内存池构成的单链表中的所有节点结尾的last指针重置到刚分配时的位置。小块内存中存储的数据并没有被释放，其在以后的内存池使用的过程中将会被覆盖更新。这可以减少内存分配的次数，提升内存重用率。但会浪费一些内存空间。

## **6.内存池使用源码详解**

 　　内存池创建好之后，如何进行使用呢？这些内存使用完了之后是如何进行回收利用的呢？下面的部分将会详细的介绍内存池的使用。

### **（a）.从内存池中申请内存**

　　在Nginx中，基于内存池的申请方法主要有ngx\_palloc、ngx\_pnalloc、ngx\_pcalloc和ngx\_pmemalign共4种方法。而不基于内存池，直接从操作系统中申请内存的主要有ngx\_alloc和ngx\_calloc共两种方法。在这一小节中，我们只讲述从内存池中申请内存相关的4中方法。而其他的部分将会在后面的小节进行讲解。

　　基于内存池的4中内存申请方法的区别在第4章：内存池API介绍中已经详细阐述了。此处不再赘述。

#### **（1）.ngx\_palloc**

　　下面给出源码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | void \*  ngx\_palloc(ngx\_pool\_t \*pool, **size\_t** size)  {  #if !(NGX\_DEBUG\_PALLOC)      if (size <= pool->max) {          return ngx\_palloc\_small(pool, size, 1);      }  #endif        return ngx\_palloc\_large(pool, size);  } |

　　从其实现中，我们可以看出，ngx\_palloc()总共有两个参数，第一个是在那个内存池上申请内存（之前我们曾经提到过通常为每个Http请求或者连接创建一个内存池，此处需要传递的参数就是这些内存池对应的指针），另一个参数是size，表示申请内存的大小。进入函数后，首先是判断申请的内存大小和max（小块内存标准）的关系，如果size<max，就调用ngx\_palloc\_small()函数申请内存。否则调用ngx\_palloc\_large()函数申请内存。下面让我们先来看ngx\_palloc\_small()函数的源码，如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | static ngx\_inline void \*  ngx\_palloc\_small(ngx\_pool\_t \*pool, **size\_t** size, ngx\_uint\_t align)  {      u\_char      \*m;      ngx\_pool\_t  \*p;        p = pool->current;        do {          m = p->d.last;            if (align) {              m = ngx\_align\_ptr(m, 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);  } |

　　从上述源码中，我们可以看到，该函数从current指向的内存池（小块内存池链表）中开始循环遍历。在每一次遍历中，我们首先获得目前内存池中未分配的空闲内存的首地址last，并赋值给m，然后由于从ngx\_palloc()函数中传递过来的align=1，因此调用ngx\_align\_ptr()，这是个什么呢？仅从此我们不能判断其是函数还是宏，下面我们给出其源码，在src/core/ngx\_config.h中，如下所示：

|  |  |
| --- | --- |
| 1  2 | #define ngx\_align\_ptr(p, a)                                                   \      (u\_char \*) (((**uintptr\_t**) (p) + ((**uintptr\_t**) a - 1)) & ~((**uintptr\_t**) a - 1)) |

　　可以看出，这是一个宏定义，该操作比较巧妙，用于计算以参数a对齐后的偏移指针p。实际上，我们最后分配的内存空间就是从对齐后的偏移指针开始的，这可能会浪费少数几个字节，但却能提高读取效率。接着分析ngx\_palloc-small()函数中的源码，在调用完宏ngx\_align\_ptr(m, NGX\_ALIGNMENT)后我们得到了以默认参数16对齐的偏移指针m。此时，我们已经拥有了对齐后的空闲内存地址空间的首地址m和尾部地址end，我们就可以计算出该块内存池（一个block）剩余的空闲内存空间大小：p->d.end - m。那么这个剩余的空闲内存空间是否一定能满足用户的内存申请请求（size个字节）呢？答案是否定的。因此我们需要将从current开始的每一个小块内存池的剩余空闲内存空间和size进行比较，遍历链表直到找到满足申请大小（size个字节）的小块内存池。如果小块内存池链表上的某块小块内存能够满足需求，那么我们就将从Nginx的内存池中划分出内存空间，并更新last的值（将last的值后移size个字节），然后返回m。

　　如果遍历完整个小块内存池都没有找到满足申请大小的内存，则程序调用ngx\_palloc\_block()函数。其源码如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | static void \*  ngx\_palloc\_block(ngx\_pool\_t \*pool, **size\_t** size)  {      u\_char      \*m;  **size\_t**       psize;      ngx\_pool\_t  \*p, \*new;        psize = (**size\_t**) (pool->d.end - (u\_char \*) pool);        m = ngx\_memalign(NGX\_POOL\_ALIGNMENT, psize, pool->**log**);      if (m == NULL) {          return NULL;      }        new = (ngx\_pool\_t \*) m;        new->d.end = m + psize;      new->d.next = NULL;      new->d.failed = 0;        m += sizeof(ngx\_pool\_data\_t);      m = ngx\_align\_ptr(m, NGX\_ALIGNMENT);      new->d.last = m + size;        for (p = pool->current; p->d.next; p = p->d.next) {          if (p->d.failed++ > 4) {              pool->current = p->d.next;          }      }        p->d.next = new;        return m;  } |

　　既然当前整个内存池都不能满足用户内存的申请，而我们的操作系统明明还有内存可用（资源耗尽的情况除外），那我们总不能拒绝用户的合理请求吧。ngx\_palloc\_block()函数就是应对这种情况而出现的。该函数实现了对内存池的扩容。

　　需要注意的是，由于我们遍历完了整个链表，因此此时的pool指针指向的是内存池链表的最后一个节点。所以说在ngx\_palloc\_block()中计算的是当前内存池最后一个节点的大小psize。该大小为需要扩展的空间大小。然后，我们调用前面提到过的ngx\_memalgin()函数申请新的内存空间，大小为psize，作为新的小块内存池节点。之后，我们将这个节点挂在内存池的最后面。具体怎么实现的呢？我们来详细的看一看。

　　首先将这个新节点进行初始化，包括d->end、d->next、d->failed。然后将指向这块内存的首地址m后移sizeof(ngx\_pool\_data\_t)，大家可能还记得我们在创建内存池ngx\_pool\_create()时，内存池中空闲地址的首地址是在整个内存池的首地址的基础上后移了sizeof(ngx\_pool\_t)，那么为什么此处创建新的内存池节点只需要后移sizeof(ngx\_pool\_data\_t)呢？在x86系统上，sizeof(ngx\_pool\_data\_t)对应16个字节，而sizeof(ngx\_pool\_t)对应40个字节。其实大家仔细想一想，我们创建的内存池是小块内存池链表的第一个节点，这个节点中除了包含ngx\_pool\_data\_t结构体之外，还需要包含large指针、cleanup指针等。而小块内存池后面的节点均没有必要包含这些成员，因为我们的large链表和cleanup链表是直接且仅仅挂在小块内存池链表的第一个节点上的。不需要再挂到后续的其他小块内存池链表的结构上。这么想是不是觉得比较合理呢？答案就是这样的。但是我们之前的重置内存池操作中，并没有把后续的从第二个节点开始的小块内存池链表上的空闲内存的起始地址初始化为(u\_char \*)p + sizeof (ngx\_pool\_data\_t)，而是将所有节点（包括第一个）的空闲内存地址初始化为(u\_char \*)p + sizeof (ngx\_pool\_t)。这样做会浪费一些内存空间，但是整个重置内存池操作会简单一点点。因为不用区分第一个节点和其他节点。如果区分的话，我们需要让第一个节点的空闲内存的起始地址初始化为(u\_char \*)p + sizeof (ngx\_pool\_t)，将其他节点的空闲内存的起始地址初始化为(u\_char \*)p + sizeof (ngx\_pool\_data\_t)。我们的Nginx源码就是这么实现的。大家知道就行了。因为这并不会影响内存池的使用。

　　在完成对新的内存池节点的初始化之后。我们需要将这个节点加入到小块内存池链表的尾部。具体怎么实现的呢？

　　首先我们找到current指针，并根据这个指针遍历小块内存池链表，在每一个遍历中，我们将每个节点的failed成员加1（这是因为你们这些节点不能给我分配内存啊，不然也不会调用我，因此对你们的failed成员统统加1）。并且加1之后，进行判断，如果某个节点的failed成员的值大于4，那么就将current指向下一个节点（下次再分配内存时将会自动忽略这个节点）。

　　在遍历完小块内存池的链表后，我们的pool指针已经指向了链表的最后一个节点，因此在链表的尾部插入一个节点非常简单，p->d.next = new这个语句就能完成。之后返回这个指向这个新节点的空闲内存空间的首地址。

　　上述就是ngx\_palloc\_small()函数完成的功能，内容比较多大家可能都忘了，我们还没有讲解ngx\_palloc()函数的另外一个部分：ngx\_palloc\_large()，这个函数是用于当用户申请的内存大小大于我们的小块内存标准max的情况。下面我们将会看到，这种情况下，申请的内存将被当作是大数据块，将会被挂在large链表上。先给出ngx\_palloc\_large()的源码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | static void \*  ngx\_palloc\_large(ngx\_pool\_t \*pool, **size\_t** size)  {      void              \*p;      ngx\_uint\_t         n;      ngx\_pool\_large\_t  \*large;        p = ngx\_alloc(size, pool->**log**);      if (p == NULL) {          return NULL;      }        n = 0;        for (large = pool->large; large; large = large->next) {          if (large->alloc == NULL) {              large->alloc = p;              return p;          }            if (n++ > 3) {              break;          }      }        large = ngx\_palloc\_small(pool, sizeof(ngx\_pool\_large\_t), 1);      if (large == NULL) {          ngx\_free(p);          return NULL;      }        large->alloc = p;      large->next = pool->large;      pool->large = large;        return p;  } |

　　从上面的代码中我们可以看出我们首先调用ngx\_alloc()函数申请一块大小为size的内存空间，ngx\_alloc()函数实际上就是简单的封装了以下malloc()函数，后面我们会详细的讲解。这里知道它是由malloc实现的就好了。申请完内存之后，开始遍历large链表，找到链表中alloc为NULL的节点，用alloc指向刚申请到的内存空间并返回。注意这段循环代码至多执行3次，如果在3次后都没有找到alloc为NULL的节点，就会退出循环，继续执行后面的代码。限制代码执行的次数是为了提升内存分配的效率，因为large链表可能会很大。

　　之后，我们调用ngx\_palloc\_small()重新申请一块大小为sizeof(ngx\_pool\_large\_t)结构体大小的内存，建立一个新节点。最后我们把新建立的节点插入到large链表的头部，返回申请的内存空间的起始地址。为什么是插入头部而不是插入尾部呢？这里面其实是有依据的，因为我们之前为了防止large过大将遍历large链表的次数设置为3，如果插在尾部，那么遍历链表前面的三个节点就没有意义了，因为每次都可能会遍历不到后面的空闲节点，而导致每次都需要重新建立新节点。并且插入头部，从头部开始遍历也会使得效率比较高。因为这样遍历到空闲的大块内存节点的概率会高很多。

#### **(2).ngx\_pnalloc**

　　先给出其源码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | void \*  ngx\_pnalloc(ngx\_pool\_t \*pool, **size\_t** size)  {  #if !(NGX\_DEBUG\_PALLOC)      if (size <= pool->max) {          return ngx\_palloc\_small(pool, size, 0);      }  #endif        return ngx\_palloc\_large(pool, size);  } |

　　我们可以看到，ngx\_pnalloc()和ngx\_palloc()非常相似，唯一的区别就是ngx\_pnalloc()中调用的是ngx\_palloc\_small(pool, size, 0)，而ngx\_palloc()中调用的是ngx\_palloc\_small(pool, size, 1)。那么实际上的含义有什么区别呢？ngx\_pnalloc()分配内存时不考虑内存数据对齐，而ngx\_palloc()分配内存时考虑内存数据对齐。

#### **(3).ngx\_pcalloc**

　　我们先给出其源码，如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | 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;  } |

　　从其实现可以看出，ngx\_pcalloc()和ngx\_palloc()非常的相似，唯一的区别就是ngx\_pcalloc()函数将刚申请到的内存空间全部初始化为0。

#### **(4).ngx\_pmemalign**

　　我们给出其源码，如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | void \*  ngx\_pmemalign(ngx\_pool\_t \*pool, **size\_t** size, **size\_t** alignment)  {      void              \*p;      ngx\_pool\_large\_t  \*large;        p = ngx\_memalign(alignment, size, pool->**log**);      if (p == NULL) {          return NULL;      }        large = ngx\_palloc\_small(pool, sizeof(ngx\_pool\_large\_t), 1);      if (large == NULL) {          ngx\_free(p);          return NULL;      }        large->alloc = p;      large->next = pool->large;      pool->large = large;        return p;  } |

　　从其源码实现中，我们可以看出ngx\_pmemalign()函数首先调用ngx\_memalign()函数来申请对齐的内存地址空间。然后ngx\_palloc\_small()函数来建立一个新的大数据块节点。并将ngx\_pmemalign()函数申请的内存空间直接挂在新建的大块数据节点的alloc成员上。最后再将新建的大数据块节点挂在大块内存组成的单链表中。

　　上面就是整个基于内存池申请内存的4种方法的源码实现及其分析。下面我们会继续讲解释放内存和回收内存。

　　ngx\_pfree()函数用于提前释放大块内存。

### **（b）.释放内存**

　　此处我们将介绍基于内存池的内存释放操作函数ngx\_pfree()，与内存池无关的内存释放操作ngx\_free()将在后面被讲解。

　　在Nginx中，小块内存并不存在提前释放这么一说，因为其占用的内存较少，不太需要被提前释放。但是对于非常大的内存，如果它的生命周期远远短于所属的内存池，那么在内存池销毁之前提前释放它就变得有意义了。下面先给出其源码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | ngx\_int\_t  ngx\_pfree(ngx\_pool\_t \*pool, void \*p)  {      ngx\_pool\_large\_t  \*l;        for (l = pool->large; l; l = l->next) {          if (p == l->alloc) {              ngx\_log\_debug1(NGX\_LOG\_DEBUG\_ALLOC, pool->**log**, 0,                             "free: %p", l->alloc);              ngx\_free(l->alloc);              l->alloc = NULL;                return NGX\_OK;          }      }        return NGX\_DECLINED;  } |

　　从其实现中可以看出，ngx\_pfree()函数的实现十分简单。通过遍历large单链表，找到待释放的内存空间（alloc所指向的内存空间），然后调用ngx\_free()函数释放内存。后面我们会看到ngx\_free()函数是free()函数的一个简单封装。释放alloc所占用的空间后，将alloc设置为NULL。我们需要注意的是：ngx\_pfree()函数仅仅释放了large链表上每个节点的alloc成员所占用的空间，并没有释放ngx\_pool\_large\_t结构所占用的内存空间。如此实现的意义在于：下次分配大块内存时，会期望复用这个ngx\_pool\_large\_t结构体。从这里可以想到，如果large链表中的元素很多，那么ngx\_pfree()的遍历耗损的性能是不小的，如果不能确定内存确实非常大，最好不要调用ngx\_pfree。

### **（c）.随着内存池释放同步释放资源的操作**

　　在Nginx服务器程序中，有些数据类型在回收其所占的资源时不能直接通过释放内存空间的方式进行，而需要在释放之前对数据进行指定的数据清理操作。ngx\_pool\_cleanup\_t结构体的函数指针handler就是这么一个数据清理函数，其data成员就指向要清理的数据的内存地址。我们将要清理的方法和数据存放到ngx\_pool\_cleanup\_t结构体中，通过next成员组成内存回收链表，就可以实现在释放内存前对数据进行指定的数据清理操作。而与这些操作相关的方法有：ngx\_pool\_cleanup\_add()、ngx\_pool\_run\_cleanup\_file()、ngx\_pool\_cleanup\_file()和ngx\_pool\_delete\_file()共4种。下面我们将分别讲解这些操作。

#### **（1）.ngx\_pool\_cleanup\_add()**

　　这个方法的目的是为了添加一个需要在内存池释放时同步释放的资源。我们依照惯例还是先给出其源码，然后对源码进行分析和学习。其源码如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | 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;  } |

　　从其实现中我们可以看出，我们首先调用ngx\_palloc()函数申请cleanup单链表中的一个新节点（指向ngx\_pool\_cleanup\_t结构体的指针），然后根据参数size是否为0决定是否需要申请存放目标数据的内存空间。当size>0时，调用ngx\_palloc()函数申请大小为size个字节的用于存放待清理的数据的内存空间。这些要清理的数据存储在ngx\_pool\_cleanup\_t结构体的data成员指向的内存空间中。这样可以利用这段内存传递参数，供清理资源的方法使用。当size=0时，data为NULL。最后将新生成的ngx\_pool\_cleanup\_t结构体挂在cleanup单链表的头部。返回一个指向ngx\_pool\_cleanup\_t结构体的指针。而我们得到后需要设置ngx\_pool\_cleanup\_t的handler成员为释放资源时执行的方法。

返回的指向ngx\_pool\_cleanup\_t结构体的指针具体怎么使用呢？我们对ngx\_pool\_cleanup\_t结构体的data成员指向的内存空间填充目标数据时，将会为handler成员指定相应的函数。

#### **（2）.ngx\_pool\_run\_cleanup\_file()**

　　在内存池释放前，如果需要提前关闭文件，则调用该方法。下面给出其源码，如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | void  ngx\_pool\_run\_cleanup\_file(ngx\_pool\_t \*p, ngx\_fd\_t fd)  {      ngx\_pool\_cleanup\_t       \*c;      ngx\_pool\_cleanup\_file\_t  \*cf;        for (c = p->cleanup; c; c = c->next) {          if (c->handler == ngx\_pool\_cleanup\_file) {                cf = c->data;                if (cf->fd == fd) {                  c->handler(cf);                  c->handler = NULL;                  return;              }          }      }  } |

　　再给出ngx\_pool\_cleanup\_file结构体的声明和定义（在src/core/ngx\_palloc.h头文件中），如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5 | typedef struct {      ngx\_fd\_t              fd;      u\_char               \*name;      ngx\_log\_t            \***log**;  } ngx\_pool\_cleanup\_file\_t; |

　　从上述源码中，我们可以看出，ngx\_pool\_run\_cleanup\_file()通过遍历cleanup单链表，寻找单链表上的一个节点，这个节点满足handler（函数指针）等于ngx\_pool\_cleanup\_file（在与函数名相关的表达式中，函数名会被编译器隐式转换成函数指针）。由于ngx\_pool\_cleanup\_t结构体的data成员经常会指向ngx\_pool\_cleanup\_file\_t（在后面的ngx\_pool\_cleanup\_file()函数中我们可以看到），我们将这个节点data指针赋值给cf（ngx\_pool\_cleanup\_t结构指针）。之后如果传递过来的参数fd与cf->fd相同的话（代表我们找到了需要提前关闭的文件描述符fd），就提前执行ngx\_pool\_cleanup\_file（fd），进行文件的关闭操作。

#### **（3）.ngx\_pool\_cleanup\_file()**

　　该方法以关闭文件的方式来释放资源，可以被设置为ngx\_pool\_cleanup\_t的handler成员（函数指针）。我们给出其源码实现，如下所示:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | void  ngx\_pool\_cleanup\_file(void \*data)  {      ngx\_pool\_cleanup\_file\_t  \*c = data;        ngx\_log\_debug1(NGX\_LOG\_DEBUG\_ALLOC, c->**log**, 0, "file cleanup: fd:%d",                     c->fd);        if (ngx\_close\_file(c->fd) == NGX\_FILE\_ERROR) {          ngx\_log\_error(NGX\_LOG\_ALERT, c->**log**, ngx\_errno,                        ngx\_close\_file\_n " \"%s\" failed", c->name);      }  } |

　　可以看出，ngx\_pool\_cleanup\_t结构的data成员指向ngx\_pool\_cleanup\_file\_t结构体（前面讲解ngx\_pool\_run\_cleanup\_file()提到过）。之后直接调用ngx\_close\_file（）函数关闭对应的文件。而ngx\_close\_file（）底层是是通过close()函数实现的。

#### **(4).ngx\_pool\_delete\_file()**

　　以删除文件来释放资源的方法，可以设置到ngx\_pool\_cleanup\_t的handler成员。我们先给出其源码，如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | void  ngx\_pool\_delete\_file(void \*data)  {      ngx\_pool\_cleanup\_file\_t  \*c = data;        ngx\_err\_t  err;        ngx\_log\_debug2(NGX\_LOG\_DEBUG\_ALLOC, c->**log**, 0, "file cleanup: fd:%d %s",                     c->fd, c->name);        if (ngx\_delete\_file(c->name) == NGX\_FILE\_ERROR) {          err = ngx\_errno;            if (err != NGX\_ENOENT) {              ngx\_log\_error(NGX\_LOG\_CRIT, c->**log**, err,                            ngx\_delete\_file\_n " \"%s\" failed", c->name);          }      }        if (ngx\_close\_file(c->fd) == NGX\_FILE\_ERROR) {          ngx\_log\_error(NGX\_LOG\_ALERT, c->**log**, ngx\_errno,                        ngx\_close\_file\_n " \"%s\" failed", c->name);      }  } |

　　可以看出，ngx\_pool\_cleanup\_t结构的data成员指向ngx\_pool\_cleanup\_file\_t结构体，在程序中我们先将传递过来的参数data（待清理的目标数据）赋值给c，然后对c的成员name（文件名称）调用ngx\_delete\_file()函数，完成对文件的删除操作，之后调用ngx\_close\_file()函数关闭相应的文件流（关闭这个文件流可以阻止删除的文件再次被访问，并且释放FILE结构使得它可以被做用于其他的文件），这就是我们为什么在删除对应的文件后还需要关闭打开的文件流的原因。

　　补充一下：ngx\_close\_file和ngx\_delete\_file其实是一个宏定义，我们可以在src/os/unix/ngx\_files.h中看到其具体实现，如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | #define ngx\_close\_file           close  #define ngx\_close\_file\_n         "close()"      #define ngx\_delete\_file(name)    unlink((const char \*) name)  #define ngx\_delete\_file\_n        "unlink()" |

　　可以看到，ngx\_close\_file其实就是close，在Nginx服务器程序编译阶段仅仅做一个简单的替换。ngx\_delete\_file(name)也是一个宏定义，本质上为unlink((const char \*) name)，该函数会删除参数name指定的文件。

（d）.与内存池无关的资源分配、释放操作

 　　与内存池无关的内存分配和释放操作主要有ngx\_alloc()、ngx\_calloc()和ngx\_free()共3中操作方法。下面我们将继续讲解它们的具体实现。

#### **（1）.ngx\_alloc()**

　　ngx\_alloc()函数直接从操作系统中申请内存，其实现是对malloc()函数的一个简单封装。我们可以在src/os/unix/ngx\_alloc.c中找到其源码。如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | void \*  ngx\_alloc(**size\_t** size, ngx\_log\_t \***log**)  {      void  \*p;        p = **malloc**(size);      if (p == NULL) {          ngx\_log\_error(NGX\_LOG\_EMERG, **log**, ngx\_errno,                        "malloc(%uz) failed", size);      }        ngx\_log\_debug2(NGX\_LOG\_DEBUG\_ALLOC, **log**, 0, "malloc: %p:%uz", p, size);        return p;  } |

　　可以看到，其实现非常简单。仅仅是封装了malloc()函数，并做了一些日志和调试方面的处理。

#### **（2）.ngx\_calloc()**

　　ngx\_calloc()和ngx\_alloc()非常相似，唯一的区别是在调用malloc()函数申请完内存之后，会调用ngx\_memzero()函数将内存全部初始化为0。ngx\_memzero()就是memset()函数。

#### **（3）.ngx\_free()**

　　我们可以在src/os/unix/ngx\_alloc.h中看到其源码，如下所示：

|  |  |
| --- | --- |
| 1 | #define ngx\_free          free |

　　可以看到Nginx程序释放内存的函数非常简单，和销毁内存池中用的是同一个(free)。这里需要再次说明的是：对于在不同场合下从内存池中申请的内存空间的释放时机是不一样的。一般只有大数据块才直接调用ngx\_free()函数进行释放，其他数据空间的释放都是在内存池销毁的时机完成的，不需要提前完成。

　　至此，Nginx与内存相关的操作的源码实现已基本讲完了。大家如果想进一步研究和学习Nginx内存管理机制，可以从官方下载Nginx源码，从源码中去发现Nginx降低系统内存开销的方法。

## **7.小结**

　　所有的讲解都讲述完了，我们来进行总结一下。在第1节中，我们介绍了Nginx的内存管理机制-内存池的基本原理和使用内存池管理Nginx服务器程序带来的好处。为了方便大家对内存池结构的理解，我们在第2节中特意给出了ngx\_pool\_t内存池的示意图2.1，并简单的阐述了这个图的具体含义。在此基础上，我们继续在第3节中讲述了与内存池相关的重要的数据结构，主要包括ngx\_pool\_t、ngx\_pool\_data\_t、ngx\_pool\_large\_t和ngx\_pool\_cleanup\_t。然后为了给大家一个内存池操作方法的宏观介绍，我们在第4节讲述了内存的主要操作方法（共15个分成4类）。之后在第5节中我们详细介绍了内存池的管理，主要包括内存池的创建、销毁和重置。在第6节中我们详细介绍了内存池的使用，主要包括从内存池中如何申请内存、释放内存和回收内存。这两个小结是整个Nginx内存管理的精华部分，我们在这部分中详细的分析Nginx的源码实现，从源码的角度去讲解Nginx内存管理用到的技术，方便我们在以后的程序设计中可以借鉴和学习。最后，希望这篇文章能真正帮助到大家学习Nginx。