Nginx源码剖析之内存池，与内存管理 - 云+社区 - 腾讯云

58-72 minutes

Nginx源码剖析之内存池，与内存管理

作者：July、dreamice、阿波、yixiao。

出处：http://blog.csdn.net/v_JULY_v/。

引言

Nginx（发音同 engine x）是一款轻量级的Web 服务器/反向代理服务器及电子邮件（IMAP/POP3）代理服务器，并在一个BSD-like 协议下发行。由俄罗斯的程序设计师Igor Sysoev所开发，最初供俄国大型的入口网站及搜寻引擎Rambler（俄文：Рамблер）使用。

其特点是占有内存少，并发能力强，事实上nginx的并发能力确实在同类型的网页服务器中表现较好，目前中国大陆使用nginx网站用户有：新浪、网易、 腾讯，另外知名的微网志Plurk也使用nginx，以及诸多暂不曾得知的玩意儿。

读者可以到此处下载Nginx最新版本的源码：http://nginx.org/en/download.html。同时，本文本不想给源码太多注释，因为这不像讲解算法，算法讲解的越通俗易懂越好，而源码剖析则不同，缘由在于不同的读者对同一份源码有着不同的理解，或深或浅，所以，更多的是靠读者自己去思考与领悟。

ok，本文之中有任何疏漏或不正之处，恳请批评指正。谢谢。

Nginx源码剖析之内存池

1、内存池结构

内存相关的操作主要在文件 os/unix/ngx_alloc.{h,c} 和 core/ngx_palloc.{h,c} 中实现，ok，咱们先来看内存管理中几个主要的数据结构：

```
1. typedef struct {    //内存池的数据结构模块
2.     u_char               *last;    //当前内存分配结束位置，即下一段可分配内存的起始位置
3.     u_char               *end;     //内存池的结束位置
4.     ngx_pool_t           *next;    //链接到下一个内存池，内存池的很多块内存就是通过该指针连成链表的
5.     ngx_uint_t           failed;  //记录内存分配不能满足需求的失败次数
6. } ngx_pool_data_t;   //结构用来维护内存池的数据块，供用户分配之用。
7. struct ngx_pool_t {  //内存池的管理分配模块
```

```
8.    ngx_pool_data_t       d;       //内存池的数据块（上面已有描述），设为d
9.  size_t                  max;     //数据块大小，小块内存的最大值
10.   ngx_pool_t          *current;  //指向当前或本内存池
11.   ngx_chain_t         *chain;    //该指针挂接一个ngx_chain_t结构
12.   ngx_pool_large_t     *large;    //指向大块内存分配，nginx中，大块内存分配直接采用标准系统接口malloc
13.   ngx_pool_cleanup_t   *cleanup;  //析构函数，挂载内存释放时需要清理资源的一些必要操作
14.   ngx_log_t           *log;      //内存分配相关的日志记录
15. };
```

再来看看大块数据分配的结构体：

```
1. struct ngx_pool_large_t {
2.    ngx_pool_large_t     *next;
3. void                    *alloc;
4. };
```

其它的数据结构与相关定义：

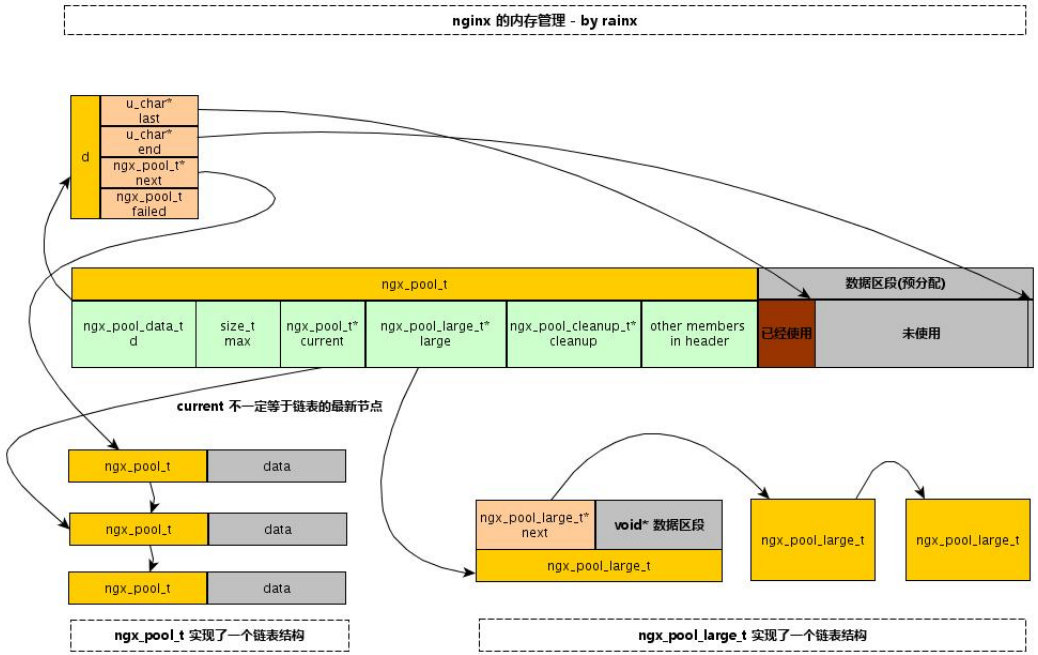
```
1. typedef struct {
2.    ngx_fd_t             fd;
3.    u_char               *name;
```

4. ngx_log_t *log;

5. } ngx_pool_cleanup_file_t;

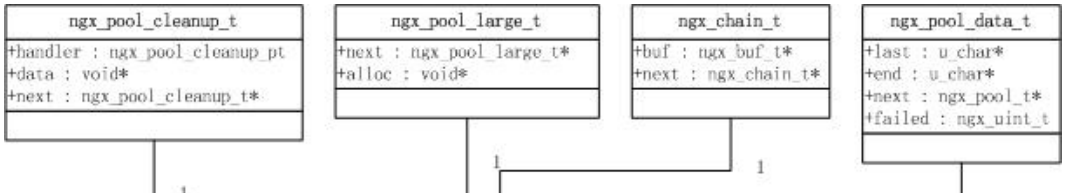
6. #define NGX_MAX_ALLOC_FROM_POOL (ngx_pagesize - 1) //在x86体系结构下，该值一般为4096B，即4K

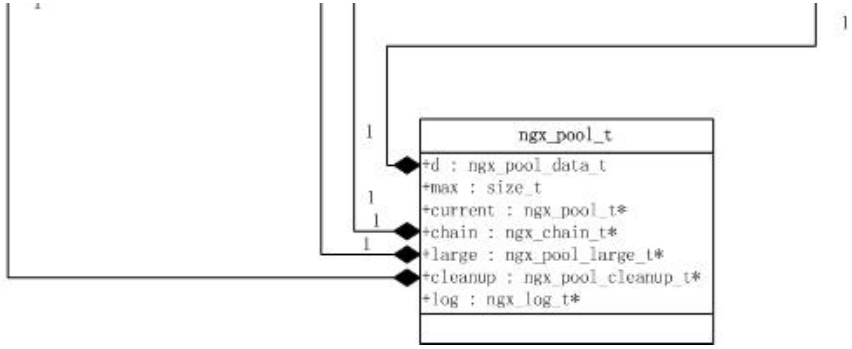
上述这些数据结构的逻辑结构图如下(下图最左上角部分没有与上文的第一个数据结构内的ngx_uint_t对应起来，特此说明)：



1.1、ngx_pool_t的逻辑结构

再看一下用UML绘制的ngx_pool_t的逻辑结构图：





在下一节，我们将会深入分析内存管理的主要函数。

Nginx源码剖析之内存管理

2、内存池操作

2.1、创建内存池

```
1. ngx_pool_t *
2. ngx_create_pool(size_t size, ngx_log_t *log)
3. {
4.     ngx_pool_t  *p;
5.
6.     p = ngx_memalign(NGX_POOL_ALIGNMENT, size, log);
7. //ngx_memalign()函数执行内存分配，该函数的实现在src/os/unix
   /ngx_alloc.c文件中（假定NGX_HAVE_POSIX_MEMALIGN被定
   义）：
8.
```

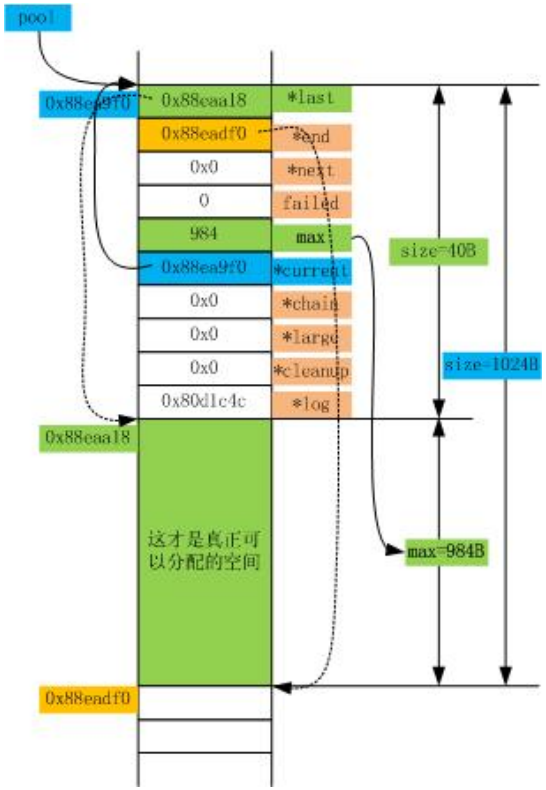
```
 9. if (p == NULL) {
10. return NULL;
11.     }
12.
13.     p->d.last = (u_char *) p + sizeof(ngx_pool_t);
14.     p->d.end = (u_char *) p + size;
15.     p->d.next = NULL;
16.     p->d.failed = 0;
17.
18.     size = size - sizeof(ngx_pool_t);
19.     p->max = (size < NGX_MAX_ALLOC_FROM_POOL) ? size : NGX_MAX_ALLOC_FROM_PO
20. //最大不超过4095B，别忘了上面
    NGX_MAX_ALLOC_FROM_POOL的定义
21.
22.     p->current = p;
23.     p->chain = NULL;
24.     p->large = NULL;
25.     p->cleanup = NULL;
26.     p->log = log;
```

27.

28. **return** p;

29. }

例如，调用ngx_create_pool(1024, 0x80d1c4c)后，创建的内存池物理结构如下图：



紧接着，咱们就来分析下上面代码中所提到的：ngx_memalign()函数。

1. **void** *

2. ngx_memalign(**size_t** alignment, **size_t** size, ngx_log_t *log)

```
3. {
4. void  *p;
5. int    err;
6.
7.     err = posix_memalign(&p, alignment, size);
8. //该函数分配以alignment为对齐的size字节的内存大小，其中p指向
   分配的内存块。
9.
10. if (err) {
11.         ngx_log_error(NGX_LOG_EMERG, log, err,
12. "posix_memalign(%uz, %uz) failed", alignment, size);
13.         p = NULL;
14.     }
15.
16.     ngx_log_debug3(NGX_LOG_DEBUG_ALLOC, log, 0,
17. "posix_memalign: %p:%uz @%uz", p, size, alignment);
18.
19. return p;
20. }
```

```
21. //从这个函数的实现体，我们可以看到
    p = ngx_memalign(NGX_POOL_ALIGNMENT, size, log);
22. //函数分配以NGX_POOL_ALIGNMENT字节对齐的size字节的内
    存，在src/core/ngx_palloc.h文件中：
23. #define NGX_POOL_ALIGNMENT       16
```

因此，nginx的内存池分配，是以16字节为边界对齐的。

2.1、销毁内存池　接下来，咱们来看内存池的销毁函数，pool指向需要销毁的内存池

```
1. void
2. ngx_destroy_pool(ngx_pool_t *pool)
3. {
4.     ngx_pool_t          *p, *n;
5.     ngx_pool_large_t    *l;
6.     ngx_pool_cleanup_t  *c;
7.
8. for (c = pool->cleanup; c; c = c->next) {
9. if (c->handler) {
10.             ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, pool->log, 0,
11. "run cleanup: %p", c);
12.             c->handler(c->data);
```

```
13.         }
14.     }
15. //前面讲到，cleanup指向析构函数，用于执行相关的内存池销毁之
    前的清理工作，如文件的关闭等，
16. //清理函数是一个handler的函数指针挂载。因此，在这部分，对内存
    池中的析构函数遍历调用。
17.
18. for (l = pool->large; l; l = l->next) {
19.         ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, pool->log, 0, "free: %p", l->alloc);
20.
21. if (l->alloc) {
22.             ngx_free(l->alloc);
23.         }
24.     }
25. //这一部分用于清理大块内存，ngx_free实际上就是标准的free函
    数，
26. //即大内存块就是通过malloc和free操作进行管理的。
27.
28. #if (NGX_DEBUG)
29.
```

```
30. /**
31.     * we could allocate the pool->log from this pool
32.     * so we can not use this log while the free()ing the pool
33.     */
34.
35. for (p = pool, n = pool->d.next; /** void */; p = n, n = n->d.next) {
36.         ngx_log_debug2(NGX_LOG_DEBUG_ALLOC, pool->log, 0,
37. "free: %p, unused: %uz", p, p->d.end - p->d.last);
38.
39. if (n == NULL) {
40. break;
41.         }
42.     }
43. //只有debug模式才会执行这个片段的代码，主要是log记录，用以跟
    踪函数销毁时日志记录。
44. #endif
45.
46. for (p = pool, n = pool->d.next; /** void */; p = n, n = n->d.next) {
47.         ngx_free(p);
```

```
48.
49. if (n == NULL) {
50. break;
51.         }
52.     }
53. }
54. //该片段彻底销毁内存池本身。
```

该函数将遍历内存池链表，所有释放内存，如果注册了clenup(也是一个链表结构)，亦将遍历该cleanup链表结构依次调用clenup的handler清理。同时，还将遍历large链表，释放大块内存。

2.3、重置内存池

void ngx_reset_pool(ngx_pool_t *pool)　重置内存池，将内存池恢复到刚分配时的初始化状态，注意内存池分配的初始状态时，是不包含大块内存的，因此初始状态需要将使用的大块内存释放掉，并把内存池数据结构的各项指针恢复到初始状态值。代码片段如下：

```
1. void
2. ngx_reset_pool(ngx_pool_t *pool)
3. {
4.     ngx_pool_t        *p;
5.     ngx_pool_large_t  *l;
```

```
6.
7. for (l = pool->large; l; l = l->next) {
8. if (l->alloc) {
9.             ngx_free(l->alloc);
10.         }
11.     }
12. //上述片段主要用于清理使用到的大块内存。
13.
14.     pool->large = NULL;
15.
16. for (p = pool; p; p = p->d.next) {
17.         p->d.last = (u_char *) p + sizeof(ngx_pool_t);
18.     }
19. }
```

这里虽然重置了内存池，但可以看到并没有释放内存池中被使用的小块内存，而只是将其last指针指向可共分配的内存的初始位置。这样，就省去了内存池的释放和重新分配操作，而达到重置内存池的目的。　　上面我们主要阐述了内存池管理的几个函数，接下来我们深入到如何从内存池中去申请使用内存。

2.4、分配内存（重点）

2.4.1、ngx_palloc 与ngx_pnalloc函数　这两个函数的参数都为(ngx_pool_t *pool, size_t size)，且返回类型为void*，唯一的区别是ngx_palloc从pool内存池分配以NGX_ALIGNMENT对齐的内存，而ngx_pnalloc分配适合size大小的内存，不考虑内存对齐。　我们在这里只分析ngx_palloc，对于ngx_pnalloc其实现方式基本类似，便不再赘述。文件：src/core/ngx_palloc.c

```
1. void *
2. ngx_palloc(ngx_pool_t *pool, size_t size)
3. {
4.     u_char      *m;
5.     ngx_pool_t  *p;
6.
7. //判断待分配内存与max值
8. //1、小于max值，则从current结点开始遍历pool链表
9. if (size <= pool->max) {
10.
11.         p = pool->current;
12.
13. do {
14. //执行对齐操作，
```

```
15. //即以last开始，计算以NGX_ALIGNMENT对齐的偏移位置指针，
16.             m = ngx_align_ptr(p->d.last, NGX_ALIGNMENT);
17.
18. //然后计算end值减去这个偏移指针位置的大小是否满足索要分配的
    size大小，
19. //如果满足，则移动last指针位置，并返回所分配到的内存地址的起
    始地址；
20. if ((size_t) (p->d.end - m) >= size) {
21.                 p->d.last = m + size;
22. //在该结点指向的内存块中分配size大小的内存
23.
24. return m;
25.             }
26.
27. //如果不满足，则查找下一个链。
28.             p = p->d.next;
29.
30.         } while (p);
31.
32. //如果遍历完整个内存池链表均未找到合适大小的内存块供分配，则
```

执行ngx_palloc_block()来分配。

```
33.
34. //ngx_palloc_block()函数为该内存池再分配一个block，该block的大小为链表中前面每一个block大小的值。
35. //一个内存池是由多个block链接起来的。分配成功后，将该block链入该poll链的最后，
36. //同时，为所要分配的size大小的内存进行分配，并返回分配内存的起始地址。
37. return ngx_palloc_block(pool, size);    //下文a节分析
38.
39.     }
40. //2、如果大于max值，则执行大块内存分配的函数ngx_palloc_large，在large链表里分配内存
41. return ngx_palloc_large(pool, size);        //下文b节分析
42. }
```

例如，在2.1节中创建的内存池中分配200B的内存，调用ngx_palloc(pool, 200)后，该内存池物理结构如下图：

a、待分配内存小于max值的情况　同样，紧接着，咱们就来分析上述代码中的ngx_palloc_block()函数：

```
1. static void *
2. ngx_palloc_block(ngx_pool_t *pool, size_t size)
```

```
3. {
4.     u_char      *m;
5. size_t       psize;
6.     ngx_pool_t  *p, *new, *current;
7.
8.     psize = (size_t) (pool->d.end - (u_char *) pool);
9. //计算pool的大小，即需要分配的block的大小
10.
11.     m = ngx_memalign(NGX_POOL_ALIGNMENT, psize, pool->log);
12. if (m == NULL) {
13. return NULL;
14.     }
15. //执行按NGX_POOL_ALIGNMENT对齐方式的内存分配，假设能够
    分配成功，则继续执行后续代码片段。
16.
17. //这里计算需要分配的block的大小
18. new = (ngx_pool_t *) m;
19.
20. new->d.end = m + psize;
```

```
21. new->d.next = NULL;
22. new->d.failed = 0;
23. //执行该block相关的初始化。
24.
25.     m += sizeof(ngx_pool_data_t);
26. //让m指向该块内存ngx_pool_data_t结构体之后数据区起始位置
27.     m = ngx_align_ptr(m, NGX_ALIGNMENT);
28. new->d.last = m + size;
29. //在数据区分配size大小的内存并设置last指针
30.
31.     current = pool->current;
32.
33. for (p = current; p->d.next; p = p->d.next) {
34. if (p->d.failed++ > 4) {
35.             current = p->d.next;
36. //失败4次以上移动current指针
37.         }
38.     }
39.
```

```
40.     p->d.next = new;
41. //将分配的block链入内存池
42.
43.     pool->current = current ? current : new;
44. //如果是第一次为内存池分配block，这current将指向新分配的
    block。
45.
46. return m;
47. }
```

注意：该函数分配一块内存后，last指针指向的是ngx_pool_data_t结构体(大小16B)之后数据区的起始位置，而创建内存池时时，last指针指向的是ngx_pool_t结构体(大小40B)之后数据区的起始位置。 结合2.8节的内存池的物理结构，更容易理解。 b、待分配内存大于max值的情况　如2.4.1节所述，如果分配的内存大小大于max值，代码将跳到ngx_palloc_large(pool, size)位置， ok，下面进入ngx_palloc_large(pool, size)函数的分析：

```
1. //这是一个static的函数，说明外部函数不会随便调用，而是提供给内
   部分配调用的，
2. //即nginx在进行内存分配需求时，不会自行去判断是否是大块内存还
   是小块内存，
3. //而是交由内存分配函数去判断，对于用户需求来说是完全透明的。
```

```
4. static void *
5. ngx_palloc_large(ngx_pool_t *pool, size_t size)
6. {
7. void              *p;
8.     ngx_uint_t         n;
9.     ngx_pool_large_t  *large;
10.
11.    p = ngx_alloc(size, pool->log);  //下文紧接着将分析此ngx_alloc函
    数
12. if (p == NULL) {
13. return NULL;
14.    }
15.
16.    n = 0;
17.
18. //以下几行，将分配的内存链入pool的large链中，
19. //这里指原始pool在之前已经分配过large内存的情况。
20. for (large = pool->large; large; large = large->next) {
21. if (large->alloc == NULL) {
```

```
22.             large->alloc = p;
23. return p;
24.         }
25.
26. if (n++ > 3) {
27. break;
28.         }
29.     }
30.
31. //如果该pool之前并未分配large内存，则就没有ngx_pool_large_t来
    管理大块内存
32. //执行ngx_pool_large_t结构体的分配，用于来管理large内存块。
33.     large = ngx_palloc(pool, sizeof(ngx_pool_large_t));
34. if (large == NULL) {
35.         ngx_free(p);
36. return NULL;
37.     }
38.
39.     large->alloc = p;
```

```
40.     large->next = pool->large;
41.     pool->large = large;
42.
43. return p;
44. }
```

上述代码中，调用ngx_alloc执行内存分配：

```
1. void *
2. ngx_alloc(size_t size, ngx_log_t *log)
3. {
4. void  *p;
5.
6.     p = malloc(size);
7. //从这里可以看到，ngx_alloc实际上就是调用malloc函数分配内存
   的。
8.
9. if (p == NULL) {
10.         ngx_log_error(NGX_LOG_EMERG, log, ngx_errno,
11. "malloc() %uz bytes failed", size);
12.     }
```

```
13.
14.     ngx_log_debug2(NGX_LOG_DEBUG_ALLOC, log, 0, "malloc: %p:%uz", p, size);
15.
16. return p;
17. }
```

2.4.2、ngx_pcalloc与ngx_pmemalign函数

ngx_pcalloc是直接调用palloc分配好内存，然后进行一次0初始化操作。ngx_pcalloc的源码如下：

```
1. void *
2. ngx_pcalloc(ngx_pool_t *pool, size_t size)
3. {
4. void *p;
5.
6.     p = ngx_palloc(pool, size);
7. if (p) {
8.         ngx_memzero(p, size);
9.     }
10.
11. return p;
```

```
12. }
```

ngx_pmemalign将在分配size大小的内存并按alignment对齐，然后挂到large字段下，当做大块内存处理。ngx_pmemalign的源码如下：

```
 1. void *
 2. ngx_pmemalign(ngx_pool_t *pool, size_t size, size_t alignment)
 3. {
 4. void            *p;
 5.     ngx_pool_large_t  *large;
 6.
 7.     p = ngx_memalign(alignment, size, pool->log);
 8. if (p == NULL) {
 9. return NULL;
10.     }
11.
12.     large = ngx_palloc(pool, sizeof(ngx_pool_large_t));
13. if (large == NULL) {
14.         ngx_free(p);
15. return NULL;
16.     }
```

```
17.
18.     large->alloc = p;
19.     large->next = pool->large;
20.     pool->large = large;
21.
22. return p;
23. }
```

其余的不再详述。nginx提供给我们使用的内存分配接口，即上述本2.4节中这4种函数，至此，都已分析完毕。

2.5、释放内存

```
1. if (p == l->alloc) {
2.         ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, pool->log, 0,
3. "free: %p", l->alloc);
4.         ngx_free(l->alloc);
5.         l->alloc = NULL;
6.
7. return NGX_OK;
8.     }
9. }
```

```
10.
11. return NGX_DECLINED;
```

需要注意的是该函数只释放large链表中注册的内存，普通内存在ngx_destroy_pool中统一释放。 2.6、注册cleanup

```
1. ngx_pool_cleanup_t *
2. ngx_pool_cleanup_add(ngx_pool_t *p, size_t size)
3. {
4.     ngx_pool_cleanup_t  *c;
5.
6.     c = ngx_palloc(p, sizeof(ngx_pool_cleanup_t));
7. if (c == NULL) {
8. return NULL;
9.     }
10.
11. if (size) {
12.         c->data = ngx_palloc(p, size);
13. if (c->data == NULL) {
14. return NULL;
15.         }
```

```
16.
17.     } else {
18.         c->data = NULL;
19.     }
20.
21.     c->handler = NULL;
22.     c->next = p->cleanup;
23.
24.     p->cleanup = c;
25.
26.     ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, p->log, 0, "add cleanup: %p", c);
27.
28. return c;
29. }
```

2.7、文件相关　一些文件相关的操作函数如下，此处就不在详述了。

```
1. void
2. ngx_pool_run_cleanup_file(ngx_pool_t *p, ngx_fd_t fd)
3. {
```

```
4. //....
5. }
6.
7. void
8. ngx_pool_cleanup_file(void *data)
9. {
10. //....
11. }
12.
13. void
14. ngx_pool_delete_file(void *data)
15. {
16. //...
17. }
```

2.8、内存池的物理结构　针对本文前几节的例子，画出的内存池的物理结构如下图。

从该图也能看出2.4节的结论，即内存池第一块内存前40字节为ngx_pool_t结构，后续加入的内存块前16个字节为ngx_pool_data_t结构，这两个结构之后便是真正可以分配内存区域。

全文总结

来自淘宝数据共享平台blog内的一篇文章对上述Nginx源码剖析之内存池，与内存管理总结得很好，特此引用之，作为对上文全文的一个总结：

Nginx的内存池实现得很精巧，代码也很简洁。总的来说，所有的内存池基本都一个宗旨：申请大块内存，避免“细水长流”。 **3.1、创建一个内存池** nginx内存池主要有下面两个结构来维护，他们分别维护了内存池的头部和数据部。此处数据部就是供用户分配小块内存的地方。

```
1. //该结构用来维护内存池的数据块，供用户分配之用。
2. typedef struct {
3.     u_char *last; //当前内存分配结束位置，即下一段可分配内存的起始位置
4.     u_char *end; //内存池结束位置
5.     ngx_pool_t *next; //链接到下一个内存池
6.     ngx_uint_t failed; //统计该内存池不能满足分配请求的次数
7. } ngx_pool_data_t;
8. //该结构维护整个内存池的头部信息。
9. struct ngx_pool_s {
10.     ngx_pool_data_t d; //数据块
11. size_t max;  //数据块的大小，即小块内存的最大值
12.     ngx_pool_t *current;    //保存当前内存池
```

```
13.     ngx_chain_t *chain; //可以挂一个chain结构
14.     ngx_pool_large_t *large;    //分配大块内存用，即超过max的内存请求
15.     ngx_pool_cleanup_t *cleanup;    //挂载一些内存池释放的时候，同时释放的资源。
16.     ngx_log_t *log;
17. };
```

有了上面的两个结构，就可以创建一个内存池了，nginx用来创建一个内存池的接口是：

> ngx_pool_t *ngx_create_pool(size_t size, ngx_log_t *log)（位于src/core/ngx_palloc.c中）；

调用这个函数就可以创建一个大小为size的内存池了。

ngx_create_pool接口函数就是分配上图这样的一大块内存，然后初始化好各个头部字段（上图中的彩色部分）。红色表示的四个字段就是来自于上述的第一个结构，维护数据部分，　　由图可知：last是用户从内存池分配新内存的开始位置，end是这块内存池的结束位置，所有分配的内存都不能超过end。蓝色表示的max字段的值等于整个数据部分的长度。**用户请求的内存大于max时，就认为用户请求的是一个大内存，此时需要在紫色表示的large字段下面单独分配；用户请求的内存不大于max的话，就是小内存申请，直接在数据部分分配，此时将会移动last指针**（具体见上文2.4.1节）。 **3.2、分配小块内存(size <= max)**　上面创建好了一个可用的内存池了，也提到了小块内存的分配问题。nginx提供给用户使用的内存分配接

口有：

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_palloc和ngx_pnalloc都是从内存池里分配size大小内存，至于分得的是小块内存还是大块内存，将取决于size的大小；他们的不同之处在于，palloc取得的内存是对齐的，pnalloc则否。 ngx_pcalloc是直接调用palloc分配好内存，然后进行一次0初始化操作。 ngx_pmemalign将在分配size大小的内存并按alignment对齐，然后挂到large字段下，当做大块内存处理。下面用图形展示一下分配小块内存的模型：

上图这个内存池模型是由上3个小内存池构成的，由于第一个内存池上剩余的内存不够分配了，于是就创建了第二个新的内存池，第三个内存池是由于前面两个内存池的剩余部分都不够分配，所以创建了第三个内存池来满足用户的需求。 由图可见：所有的小内存池是由一个单向链表维护在一起的。这里还有两个字段需要关注，failed和current字段。failed表示的是当前这个内存池的剩余可用内存不能满足用户分配请求的次数，即是说：一个分配请求到来后，在这个内存池上分配不到想要的内存，那么就failed就会增加1；这个分配请求将会递交给下一个内存池去处理，如果下一个内存池也不能满足，那么它的failed也会加1，然后将请求继续往下传递，直到满足请求为止（如果没有现成的内存池来满足，会再创建一个新的内存池）。 current字段会随着failed的增加而发生改变，如果current指向的内存

池的failed达到了4的话，current就指向下一个内存池了。猜测：4这个值应该是Nginx作者的经验值，或者是一个统计值（详见上文2.4.1节a部分）。 **3.3、大块内存的分配(size > max)** 大块内存的分配请求不会直接在内存池上分配内存来满足，而是直接向操作系统申请这么一块内存（就像直接使用malloc分配内存一样），然后将这块内存挂到内存池头部的large字段下。内存池的作用在于解决小块内存池的频繁申请问题，对于这种大块内存，是可以忍受直接申请的。

同样，用图形展示大块内存申请模型：

注意每块大内存都对应有一个头部结构（next&alloc），这个头部结构是用来将所有大内存串成一个链表用的。这个头部结构不是直接向操作系统申请的，而是当做小块内存（头部结构没几个字节）直接在内存池里申请的。这样的大块内存在使用完后，可能需要第一时间释放，节省内存空间，因此nginx提供了接口函数： ngx_int_t ngx_pfree(ngx_pool_t *pool, void *p)；此函数专门用来释放某个内存池上的某个大块内存，p就是大内存的地址。 ngx_pfree只会释放大内存，不会释放其对应的头部结构，毕竟头部结构是当做小内存在内存池里申请的；遗留下来的头部结构会作下一次申请大内存之用。

3.4、cleanup资源

可以看到所有挂载在内存池上的资源将形成一个循环链表，一路走来，发现链表这种看似简单的数据结构却被频繁使用。 由图可知，每个需要清理的资源都对应有一个头部结构，这个结构中有一个关键的字段handler，handler是一个函数指针，在挂载一个资源到内存池上的时候，同时也会注册一个清理资源的函数到这个handler上。即是说，内存池在清理cleanup的时候，就是调用这个handler来

清理对应的资源。　比如：我们可以将一个开打的文件描述符作为资源挂载到内存池上，同时提供一个关闭文件描述的函数注册到handler上，那么内存池在释放的时候，就会调用我们提供的关闭文件函数来处理文件描述符资源了。 **3.5、内存的释放**　nginx只提供给了用户申请内存的接口，却没有释放内存的接口，那么nginx是如何完成内存释放的呢？总不能一直申请，用不释放啊。针对这个问题，nginx利用了web server应用的特殊场景来完成；　一个web server总是不停的接受connection和request，所以nginx就将内存池分了不同的等级，有进程级的内存池、connection级的内存池、request级的内存池。也就是说，创建好一个worker进程的时候，同时为这个worker进程创建一个内存池，待有新的连接到来后，就在worker进程的内存池上为该连接创建起一个内存池；连接上到来一个request后，又在连接的内存池上为request创建起一个内存池。这样，在request被处理完后，就会释放request的整个内存池，连接断开后，就会释放连接的内存池。因而，就保证了内存有分配也有释放。

小结：通过内存的分配和释放可以看出，nginx只是将小块内存的申请聚集到一起申请，然后一起释放。避免了频繁申请小内存，降低内存碎片的产生等问题。

参考文献

1. 朋友dreamice：http://bbs.chinaunix.net/thread-3626006-1-1.html；
2. 友人阿波：http://blog.csdn.net/livelylittlefish/article/details/6586946；

3. 朋友dreamice’blog：http://blog.chinaunix.net/space.php?uid=7201775；
4. 淘宝数据共享平台博客：http://www.tbdata.org/archives/1390。

后记

今闲来无事，拿着个nginx源码在编译器上做源码剖析，鼓捣了一下午，至晚上不料中途停电，诸多部分未能保存。然不想白忙活，又花费了一个晚上，终至补全，方成上文，并修订至五日凌晨三点。同时，也参考和借鉴了dreamice、阿波等朋友们及yixiao等大牛的作品，异常感谢。读者若有兴趣，还可以看看sgi stl 的内存池及其管理（或者，日后自个也写下）。

OK，最后，本文若有任何疏漏之处，望不吝赐教与批评指正。谢谢，完。July、二零一一年十二月五日凌晨。