**1.为什么需要内存池**

    为什么需要内存池？

a. 在大量的小块内存的申请和释放的时候，能更快地进行内存分配（对比malloc和free）

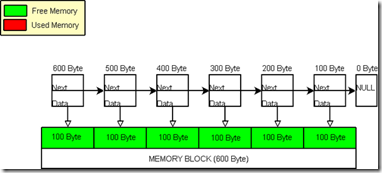
b.减少内存碎片，防止内存泄露。

**2.内存池的原理**

    内存池的原理非常简单，用申请一块较大的内存来代替N多的小内存块，当有需要malloc一块比较小的内存是，直接拿这块大的内存中的地址来用即可。

    当然，这样处理的缺点也是很明显的，申请一块大的内存必然会导致内存空间的浪费，但是比起频繁地malloc和free，这样做的代价是非常小的，这是典型的以空间换时间。

    一个典型的内存池如下图所示：

[](http://images.cnblogs.com/cnblogs_com/sld666666/WindowsLiveWriter/nginx3nginx_112BD/MemoryPool_Step5_2.png)

**图一：一个典型的内存池**

    首先定义这样一个结构体：

http://www.cnblogs.com/Images/OutliningIndicators/ContractedBlock.gifhttp://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif typedef struct MemoryBlock

{

char \*Data ; //数据

std::size\_t DataSize ; //总的大小

std::size\_t UsedSize ; //已经用了的大小

MemoryBlock\*Next ;

} MemoryBlock;

    一个内存池就是这样一连串的内存块组成。当需要用到内存的时候，调用此内存池定义好的接口GetMemory(),而需要删除的时候FreeMemory（）。

    而GetMemory和FreeMemory干了什么呢？GetMemory只是简单返回内存池中可用空间的地址。而FreeMemory干了两件事情：一： 改变UsedSize 的值，二：重新初始化这一内存区域。

**3.nginx中的内存池**

**3.1 nginx内存池的结构表示**

    首先我们看一下nginx内存池的定义：

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif代码

struct ngx\_pool\_s {  
 ngx\_pool\_data\_t d;//表示数据区域  
 size\_t max;//内存池能容纳数据的大小  
 ngx\_pool\_t \* current;//当前内存池块（nginx中的内存池是又一连串的内存池链表组成的，应该指向最后一个fail > 4的内存池的后继）  
 ngx\_chain\_t\* chain;//主要为了将内存池连接起来  
 ngx\_pool\_large\_t\* large;//大块的数据  
 ngx\_pool\_cleanup\_t\* cleanup;//清理函数  
 ngx\_log\_t\* log;//写log  
};

nginx中的内存池和普通的有比较大的不同。nginx中的内存池是由N个内存池链表

组成的，当一个内存池满了以后，就会从下一个内存池中提取空间来使用。

对于ngx\_pool\_data\_t的定义非常简单：

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

其中last表示当前数据区域的已经使用的数据的结尾。

end表示当前内存池的结尾。

next表示下一个内存池，前面已经说过，再nignx中，当一个内存池空间

不足的时候，它不会扩大其空间，而是再新建一个内存池，组成一个内存池链表。

failed标志申请内存的时候失败的次数。

在理解了这个结构体后面的就非常简单了。

current 表示当前可分配内存的内存池。

chain表示内存池链表。

large表示大块的数据。

对于ngx\_pool\_large\_t定义如下：

struct ngx\_pool\_large\_s {  
 ngx\_pool\_large\_t\* next;  
 void\* alloc;  
};

此结构体的定义也是非常简单的。一个内存地址的指针已经指向下一个地址的指针。

这里再解释下为什么需要有large数据块。当一个申请的内存空间大小比内存池的大小还要大的时候，

malloc一块大的空间，再内存池用保留这个地址的指针。

Cleanup保持存着内存池被销毁的时候的清理函数。

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

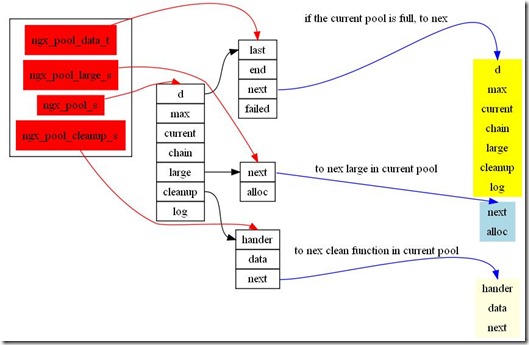
struct ngx\_pool\_cleanup\_s {  
 ngx\_pool\_cleanup\_pt handler;  
 void\* data;  
 ngx\_pool\_cleanup\_t\* next;  
};

ngx\_pool\_cleanup\_pt 是一个函数指针的典型用法，

在这个结果中保存这需要清理的数据指针以及相应的清理函数， 让内存池销毁

或其他需要清理内存池的时候，可以调用此结构体中的handler。

    下面是我画的一张nginx的内存池的结构图。

[](http://images.cnblogs.com/cnblogs_com/sld666666/WindowsLiveWriter/nginx3nginx_112BD/ngx_pool_2.jpg)

**3.2 nginx内存池源代码分析**

    要大体了解一个内存池，只需要了解其池子的创建，内存的分配以及池子的销毁即可。下面就分析下

ngx\_pool\_t的这个3个方面。注：其中有些代码可能与ngx\_pool中的源代码有所差异，但是整体意思绝对是一样的，本人的修改，只是为了更好的分析,比如 我就把所有写log的过程都去掉了。

**3.2.1 ngx\_create\_pool**

创建一个内存池

http://www.cnblogs.com/Images/OutliningIndicators/ContractedBlock.gifhttp://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif ngx\_pool\_t\* ngx\_create\_poo(size\_t size)

{

ngx\_pool\_t\* p;

p = (ngx\_pool\_t\*)malloc(size);

if (!p){

return NULL;

}

//计算内存池的数据区域

p->d->last = (u\_char\*)p + sizeof(ngx\_pool\_t);

p->d->end = (u\_char\*)p + size;

p->d->next = NULL;//下个内存池

p->d->failed = 0;

size = size - sizeof(ngx\_pool\_t);;

p->max = size;//最大数据

//我现在还是是一个单一的内存池

p->current = p;

p->chain = NULL;

//只有在需要的时候才分配大的内存区域

p->large = NULL;

p->cleanup = NULL;

return p;

}

    nginx内存池的创建非常简单，申请一个size大小的内存，把它分配给 ngx\_poo\_t。

**3.2.2 ngx\_palloc**

从内存池中分配内存.

http://www.cnblogs.com/Images/OutliningIndicators/ContractedBlock.gifhttp://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif void\* ngx\_palloc(ngx\_pool\_t\* pool, size\_t size)

{

u\_char\* m;

ngx\_pool\_t\* p;

//遍历内存池，拿出可用的内存区域

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;//用掉了当然要改变\*last了

return m;

}

p = p->d->next;

} while (p);

return ngx\_palloc\_block(pool, size);

//所有的内存池都已经满了，我要在当前的pool中增加一个

}

//申请的内存超过了内存池的大小，所以用

return ngx\_palloc\_large(pool, size);

}

    这个函数从内存池用拿出内存，如果当前内存池已满，到下一个内存池，如果所有的内存池已满，增加一个新的内存池，如果申请的内存超过了内存池的最大值，从\*large中分配

**3.3.3 ngx\_destroy\_pool**

内存池的销毁

http://www.cnblogs.com/Images/OutliningIndicators/ContractedBlock.gifhttp://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif void ngx\_destroy\_pool(ngx\_pool\_t\* pool)

{

ngx\_pool\_t \*p, \*n;

ngx\_pool\_large\_t \*l;

ngx\_pool\_cleanup\_t \*c;

//调用清理函数

for (c = pool->cleanup; c; c = c->next) {

if (c->handler) {

c->handler(c->data);

}

}

//释放大块的内存

for (l = pool->large; l; l = l->next) {

if (l->alloc) {

free(l->alloc);

}

}

//小块的内存，真正意义上的内存池

for (p = pool, n = pool->d->next; /\* void \*/; p = n, n = n->d->next) {

free(p);

//如果当前内存池为空，之后的毕为空

if (n == NULL) {

break;

}

}

}

    销毁一个内存池其实就是干了三件事， 调用清理函数， 释放大块的内存，释放内存池，需要注意的

一点是在nginx中， 小块内存除了在内存池被销毁的时候都是不能被释放的。

**3.3.4 ngx\_palloc\_block**

    前面说过，在nginx中，当内存池满了以后，会增加一个新的内存池。这个动作就是靠ngx\_palloc\_block函数实现的。

http://www.cnblogs.com/Images/OutliningIndicators/ContractedBlock.gifhttp://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif static void\* ngx\_palloc\_block(ngx\_pool\_t\* pool, size\_t size)

{

u\_char \*m;

size\_t psize;

ngx\_pool\_t \*p, \*pnew, \*current;

psize = (size\_t) (pool->d->end - (u\_char \*) pool);

m = (u\_char\*)malloc(psize);

if (!m){

return NULL;

}

//一个新的内存池

pnew = (ngx\_pool\_t\*) m;

pnew->d->end = m +psize;

pnew->d->next = NULL;

pnew->d->failed = 0;

//是不是和ngx\_palloc很相似啊

m += sizeof(ngx\_pool\_data\_t);

pnew->d->last = m + size;

current = pool->current;

//遍历到内存池链表的末尾

for (p = current; p->d->next; p = p->d->next) {

if (p->d->failed++ > 4) {//为什么4？推测是个经验值

current = p->d->next;

}

}

p->d->next = pnew;

pool->current = current ? current : pnew;

return m;

}

    这个函数就是申请了一块内存区域，变为一个内存池，然后把它连接到原来内存池的末尾。Current应该指向最后一个failed > 4 的内存池的后继。

**3.3.5 ngx\_palloc\_large 和ngx\_pfree**

    在nginx中，小块内存除了在内存池销毁之外是不能释放的，但是大块内存却可以，这两个函数就是用来控制大块内存的申请和释放， 代码也非常简单，调用malloc申请内存，连接到ngx\_pool\_large\_t中 和 调用free释放内存。这里就不贴上代码了。

**3.3.6 ngx\_reset\_pool**

 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) {

// 释放大内存块，小内存块只有在撤销内存池时才释放

if (l->alloc) {

ngx\_free(l->alloc);

}

}

pool->large = NULL;

// 重置进程池链表中的每个进程池的可分配起始地址

for (p = pool; p; p = p->d.next) {

p->d.last = (u\_char \*) p + sizeof(ngx\_pool\_t);

}

}

**3.3.7 ngx\_destroy\_pool**

void ngx\_destroy\_pool(ngx\_pool\_t \*pool)

{

ngx\_pool\_t \*p, \*n;

ngx\_pool\_large\_t \*l;

ngx\_pool\_cleanup\_t \*c;

// 执行释放内存时的清理回调函数

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);

}

}

// 释放大内存块

for (l = pool->large; l; l = l->next) {

ngx\_log\_debug1(NGX\_LOG\_DEBUG\_ALLOC, pool->log, 0, "free: %p", l->alloc);

if (l->alloc) {

ngx\_free(l->alloc);

}

}

// 释放内存池链表中的每一个内存池

for (p = pool, n = pool->d.next; /\* void \*/; p = n, n = n->d.next) {

ngx\_free(p);

if (n == NULL) {

break;

}

}

}

**4. 小结**

     nginx的内存池的代码也先介绍到这里，其实nginx内存池功能强大，所以代码也比较复杂，这里只是列出了内存池的大体流程，还有很到一部分代码未列出来。