# 一、基本概念

zend\_alloc是PHP最关键的内存管理工具。

官方介绍如下：

新的内存管理器（从PHP 5.2开始使用）会从整体上减少内存分配，并提升内存分配速度。

(来自README.md)

zend\_alloc 是为PHP设计的，对现代cpu缓存友好的内存管理工具。大部分构思来源于 jemalloc 和 tcmalloc 的实现。

所有的内存分配被划分成3种：

**巨大块** - 尺寸比 CHUNK（默认2M）更大，使用 mmap() 来进行分配。分配结果在内存中对齐到2M（大小是2M的倍数）。

**大块** - 每个 CHUNK 包含一串 4K 大小的page。大块的内存块总是对齐到 page边界（大小是page的倍数）。

**小块** -小于page大小的3/4（3k）。小块尺寸向上获取最接近的预定义数字的值。（在 ZEND\_MM\_BINS\_INFO 里共有30个预定义的数字：8, 16, 24, 32, ... 3072。）小块内存是通过RUN分配的。每个RUN被分配成一个单独的或一些连续的page。每个RUN里的空间被分配成一串空间的元素链表，占用的总空间是8 byte的倍数。

zend\_alloc 通过CHUNK从操作系统中分配内存。这些CHUNK和巨大的内存块总是与CHUNK对齐（大小是CHUNK的倍数）。所以定位包含特定指针的CHUNK总是很方便。普通CHUNK的开头保留了一个有途特殊用途的单独page，用于存放空闲page的bitset、让预定义的小尺寸有效运行的bitset、保存CHIUNK中每个page使用信息的 page 映射表等等。

zend\_alloc 提供类似 emalloc/efree/erealloc 的API，但它还提供了专用的和优化的方法来分配预定义大小的内存块。（例如 emalloc\_2(), emallc\_4(), ..., emalloc\_large() 等）当需要的内存大小已知时，这个类库使用 C 语言预处理器（preprocessor）技术和更多的专用方法来代替调用 emalloc() 。

(来自zend\_alloc.c)

本篇使用的源码版本是8.2.5。

## 一）相关文件

与zend\_alloc内存管理相关的文件主要有zend\_alloc.h，zend\_alloc.c，zend\_alloc\_sizes.h三个。zend\_alloc.h是c语言头文件，zend\_alloc.c是核心业务逻辑，zend\_alloc\_sizes.h里面存放了一些常用的常量和宏程序。

zend\_alloc\_sizes.h中定义的关键的常量如下：

|  |
| --- |
| #define ZEND\_MM\_CHUNK\_SIZE ((size\_t) (2 \* 1024 \* 1024)) // chunk大小 2M  #define ZEND\_MM\_PAGE\_SIZE (4 \* 1024) // 每个page 4K  // 一个 chunk 可分成512个 page  #define ZEND\_MM\_PAGES (ZEND\_MM\_CHUNK\_SIZE / ZEND\_MM\_PAGE\_SIZE)  #define ZEND\_MM\_FIRST\_PAGE (1) // 第一个page的编号 1  #define ZEND\_MM\_MIN\_SMALL\_SIZE 8 // 小块内存最小 8 byte  #define ZEND\_MM\_MAX\_SMALL\_SIZE 3072 // 块内存最小 3272 byte  // 大块内存最大 2m - 4k = 2044k  #define ZEND\_MM\_MAX\_LARGE\_SIZE (ZEND\_MM\_CHUNK\_SIZE - (ZEND\_MM\_PAGE\_SIZE \* ZEND\_MM\_FIRST\_PAGE)) |

## 二）主要功能

了解内存管理器的主要功能，可以从zend\_alloc.h中的标准封装宏程序（Standard wrapper macros）列表开始：

|  |  |  |  |
| --- | --- | --- | --- |
| **功能分组** | **宏程序名称** | **目标方法** | **说明** |
| **分配内存** | emalloc() | \_emalloc() | 通用方法，外部大量调用 |
| emalloc\_large() | \_emalloc\_large() | 分配大块内存，内部少量调用 |
| emalloc\_huge() | \_emalloc\_huge() | 分配巨大块内存，内部少量调用 |
| safe\_emalloc() | \_safe\_emalloc() | 分配内存，带安全保护，外部大量调用 |
| ecalloc() | \_ecalloc() | 分配内存，带安全保护，外部大量调用 |
| **释放内存** | efree() | \_efree() | 通用方法，外部大量调用 |
| efree\_large() | \_efree\_large() | 释放大块内存，内部少量调用 |
| efree\_huge() | \_efree\_huge() | 释放巨大块内存，内部少量调用 |
| **调整内存大小** | erealloc() | \_erealloc() | 通用方法，外部大量调用 |
| erealloc2() | \_erealloc2() | 调整内存大小，内部少量调用 |
| safe\_erealloc() | \_safe\_erealloc() | 调整内存大小，带安全保护，外部大量调用 |
| **创建内存并复制指定内容** | estrdup() | \_estrdup() | 使用\_emalloc()方法创建内存并复制指定字串，不传入长度，外部大量调用 |
| estrndup() | \_estrndup() | 使用\_emalloc()方法创建内存并复制指定字串，传入长度，外部大量调用 |
| / | zend\_strndup() | 使用原生方法创建内存并复制指定字串，传入长度，外部大量调用 |

可以看出，内存管理器最重要的功能是分配内存、释放内存和调整内存大小。下面以这几项主要功能为主，依次序展开介绍。

## 三）基本数据结构

内存分配中用到的主要结构如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **结构体** | **别名** | **备注** |
| - | \_zend\_alloc\_globals | zend\_alloc\_globals |  |
| heap | \_zend\_mm\_heap | zend\_mm\_heap | 堆 |
| chunk | \_zend\_mm\_chunk | zend\_mm\_chunk | 块 |
| page | \_zend\_mm\_page | zend\_mm\_page | 页 |

zend\_alloc\_globals结构体在全局只有一个实例，在zend\_alloc.c中声名。它的结构非常简单，只包含一个指向zend\_mm\_heap实例的指针。

zend\_mm\_heap结构体是整个内存的根，所有chunk都要与它关联，它除了保留主chunk的指针，还要记录内存使用情况，主要结构如下：

|  |  |  |
| --- | --- | --- |
| **元素名** | **类型** | **说明** |
| free\_slot | zend\_mm\_free\_slot[] | 空闲小块内存链表指针，30个元素 |
| huge\_list | zend\_mm\_huge\_list\* | 巨大块内存链表的指针 |
| main\_chunk | zend\_mm\_chunk\* | 主chunk |
| cached\_chunks | zend\_mm\_chunk\* | chunk链表的指针 |
| chunks\_count | int | chunk数量 |
| peak\_chunks\_count | int | chunk最大数量 |
| cached\_chunks\_count | int | 缓存chunk数量 |
| avg\_chunks\_count | double | 平均chunk数量 |
| last\_chunks\_delete\_boundary | int | chunk清理阀值 |
| last\_chunks\_delete\_count | int | 清理时删除的chunk数量 |
| real\_size | size\_t | 当前使用的内存大小 |
| limit | size\_t | 最大可内存大小 |
| overflow | int | 内存是否已经溢出 |

以上是zend\_mm\_heap的关键元素，其中前4个用于实现基本功能，后5个用于优化清理内存。

zend\_mm\_chunk结构体是PHP的基本内存分配单元，内存管理器向操作系统申请内存时，总是以chunk为单位，清理释放内存时，也是以chunk为单位。主要结构如下：

|  |  |  |
| --- | --- | --- |
| **元素名** | **类型** | **说明** |
| heap | zend\_mm\_heap\* | 指向所属heap |
| next | zend\_mm\_chunk\* | 指向前一个chunk |
| prev | zend\_mm\_chunk\* | 指向后一个chunk |
| free\_pages | uint32\_t | 空闲page数量 |
| free\_tail | uint32\_t | 末尾的空闲page数量 |
| num | uint32\_t | chunk的创建序号 |
| reserve | char[ ] | 这个没有用到 |
| heap\_slot | zend\_mm\_heap | 主chunk会用到它 |
| free\_map | zend\_mm\_page\_map | 64 Byte，也就是512 bit，每个bit标记一个page的使用状态 |
| map | zend\_mm\_page\_info[ ] | 512个page, 每个page分配一个32位整数（4Bytes）,共2K |

zend\_mm\_page结构体是内存分配的中间单位，它的大小由zend\_alloc\_sizes.h中的ZEND\_MM\_PAGE\_SIZE常量决定，值为4K。所以每个page的大小是4K。page的结构非常简单：

|  |  |  |
| --- | --- | --- |
| 元素名 | 类型 | 说明 |
| bytes | char[ZEND\_MM\_PAGE\_SIZE] | 占4K内存，并非只当成char型使用 |

## **四）启动内存管理**

在进行内存分配前，要先启动内存管理器。zend.c中的zend\_startup()方法调用zend\_alloc.c中的 start\_memory\_manager() 来启动内存管理器。zend\_startup()方法用于启动php的相关组件，无论是命令行模式、fpm模式还是调试模式，只要运行php都会调用这个方法。

调用路径如下：

|  |
| --- |
| start\_memory\_manager() -> alloc\_globals\_ctor() -> zend\_mm\_init()  ->zend\_mm\_chunk\_alloc\_int() -> zend\_mm\_mmap() |

zend\_mm\_init()方法调用zend\_mm\_chunk\_alloc\_int()方法，开辟内存创建主chunk。并初始化主chunk中自带的zend\_mm\_heap实例。从这里可以看出：

1. 主chunk的大小是ZEND\_MM\_CHUNK\_SIZE，在zend\_alloc\_sizes.h中定义，值为2M。所以php程序只要启动，就会在这里分配一个2M的空间。

2. zend\_mm\_heap实例不是单独创建的，它是主chunk中的一部分。

# **二、分配内存**

分配内存最主要的方式是调用zend\_alloc.h中的emalloc()宏程序和safe\_emalloc()宏程序，它的们调用路径如下：

|  |
| --- |
| emalloc()->\_emalloc() -> zend\_mm\_alloc\_heap()  safe\_emalloc()->\_safe\_emalloc()->\_emalloc()->zend\_mm\_alloc\_heap() |

这两个宏程序的区别是，safe\_emalloc()宏程序先调\_safe\_emalloc()方法，它带有内存溢出检测功能（具体见“带安全保护的内存分配”章节）。它们都要调用zend\_mm\_alloc\_heap()方法来分配内存。

zend\_mm\_alloc\_heap()方法根据分配的内存大小，再调用相关方法进行分配：

|  |  |
| --- | --- |
| **内存大小** | **调用方法** |
| 小于等于 ZEND\_MM\_MAX\_SMALL\_SIZE（3072Bytes） | zend\_mm\_alloc\_small() |
| 小于等于 ZEND\_MM\_MAX\_LARGE\_SIZE（2M-512Bytes） | zend\_mm\_alloc\_large() |
| 大于ZEND\_MM\_MAX\_LARGE\_SIZE（2M-512Bytes） | zend\_mm\_alloc\_huge() |

zend\_mm\_alloc\_heap()方法非常简单，代码注释如下：

|  |
| --- |
| static zend\_always\_inline void \*zend\_mm\_alloc\_heap(zend\_mm\_heap \*heap, size\_t size) {  void \*ptr;  // 如果大小在 ZEND\_MM\_MAX\_SMALL\_SIZE=3072 （4096\*3/4）内，分配 小块  if (EXPECTED(size <= ZEND\_MM\_MAX\_SMALL\_SIZE)) {  // 把size换算成 ZEND\_MM\_BINS\_INFO 数组里的行号，size 最大是3072  ptr = zend\_mm\_alloc\_small(heap, ZEND\_MM\_SMALL\_SIZE\_TO\_BIN(size));  return ptr;  // 大小在 ZEND\_MM\_MAX\_LARGE\_SIZE 内，分配 大块  } else if (EXPECTED(size <= ZEND\_MM\_MAX\_LARGE\_SIZE)) {  ptr = zend\_mm\_alloc\_large(heap, size);  return ptr;  } else { // 大小大于 ZEND\_MM\_MAX\_LARGE\_SIZE，分配 巨大块  return zend\_mm\_alloc\_huge(heap, size);  }  } |

如上所示，内存管理的很多方法第一个参数都是zend\_mm\_heap \*heap，一般情况下，全局只需要一个zend\_mm\_heap。

## **一）内存大小的对齐**

为了提高查找性能，内存管理器通常把要分配的内存大小向上取整到2的幂，例如4K,2M等。这是内存分配中很重要的一点。

这样如果对齐数是4K，需要分配3K的内存，系统会把分配的大小调整到4K,如果需要分配13K的内存，系统系统会把分配的大小调整到16K。

用于对齐内存大小的宏程序主要有2个，一个是ZEND\_MM\_ALIGNED\_OFFSET()宏程序，在分配内存和释放内存时经常会用到。另一个是zend\_mm\_alloc\_huge()方法会调用ZEND\_MM\_ALIGNED\_SIZE\_EX()宏程序来修正需要的内存空间的大小。

在内存管理中，小块内存分配比较精微复杂，巨大块内存的分配最为简单。现由浅入深，先介绍大巨块内存分配，再介绍大块和小块内存分配。

## **二）巨大块内存分配**

大于ZEND\_MM\_MAX\_LARGE\_SIZE（2M-512Bytes）的内存为巨大块的内存，需要调用zend\_mm\_alloc\_huge()方法来分配。

zend\_mm\_alloc\_huge()方法的定义如下：

|  |
| --- |
| // 分配内存时，需要传入内存所属heap指针和分配的内存大小（size）两个参数。  static void \*zend\_mm\_alloc\_huge(zend\_mm\_heap \*heap, size\_t size) |

分配内存前，先调用ZEND\_MM\_ALIGNED\_SIZE\_EX()宏程序来修正需要的内存空间的大小，并检测内存空间是否足够。

|  |
| --- |
| size\_t new\_size = ZEND\_MM\_ALIGNED\_SIZE\_EX(size, alignment); |

然后进行内存分配操作，调用路径如下：

|  |
| --- |
| zend\_mm\_alloc\_huge()-> zend\_mm\_chunk\_alloc() -> zend\_mm\_chunk\_alloc\_int()  -> zend\_mm\_mmap()  zend\_mm\_alloc\_huge()->zend\_mm\_add\_huge\_block() |

zend\_mm\_chunk\_alloc()方法用于分配chunk。

zend\_mm\_mmap()方法用于向操作系统申请内存空间，它对各种操作系统做了兼容性处理。在windows平台，调用VirtualAlloc()方法、在其他平台调用mmap()方法申请内存空间。

### 矫正内存指针

zend\_mm\_chunk\_alloc\_int()方法在第一次分配内存后，会调用ZEND\_MM\_ALIGNED\_OFFSET()宏程序检查内存指针是否已经按要求对齐到2的幂。如果没有对齐，要释放掉这块内存，并且第二次调用zend\_mm\_mmap()方法分配内存，并把指针偏移到2的幂。这个操作称为**矫正内存指针**，是非常重要的操作。在不同的平台中有不同的处理方式。

经过矫正后，chunk的开头和结尾，都会对齐到2M（ZEND\_MM\_CHUNK\_SIZE=2M）。对齐到2M是指，内存块的开头位置指针是2M的整数倍。

这样对于任何一个实例指针，只要用ZEND\_MM\_ALIGNED\_BASE()宏程序把它向左对齐到2M就可以找到所属chunk的开头，不需要在实例中多占用内存空间来存储它所属chunk的位置。

同理，对于page也一样，因为chunk的开头对齐到2M，里面的每个page都会对齐到4KBytes。查找起来非常方便。

### Windows平台的内存指针矫正

在windows操作系统中，第二次调用zend\_mm\_mmap()方法分配的内存不会使用，会直接释放掉。这次分配只是为了找到一块可用的内存空间，得到这块内存的指针后，要对它进行修正，让这个指针向右移动到对齐的位置，再调用zend\_mm\_mmap\_fixed() 方法进行第三次内存分配。

|  |
| --- |
| ptr = zend\_mm\_mmap\_fixed((void\*)((char\*)ptr + offset), size); |

这次分配的内存，指针位置是对齐的，大小也符合要求，是可用的内存。

### 非Windows平台的内存指针矫正

在其他操作系统中，不会把第二次分配的内存释放掉，而是释放掉内存块开头没有对齐的部分，和结尾多出来的部分，再来使用后面对齐的部分。

释放开头没有对齐的部分：

|  |
| --- |
| offset = alignment - offset; // 计算偏移差值  zend\_mm\_munmap(ptr, offset); // 释放这块内存开头没有对齐的部分  ptr = (char\*)ptr + offset; // 把指针移动到对齐的位置上 |

释放结尾没多出来的部分：

|  |
| --- |
| zend\_mm\_munmap((char\*)ptr + size, alignment - REAL\_PAGE\_SIZE); |

### 把巨大块内存串联成链表

分配完内存后要调用zend\_mm\_add\_huge\_block()方法把大块内存串联成链表：

|  |
| --- |
| // 创建一个链节点  zend\_mm\_huge\_list \*list = (zend\_mm\_huge\_list\*)zend\_mm\_alloc\_heap(heap, sizeof(zend\_mm\_huge\_list) ZEND\_FILE\_LINE\_RELAY\_CC ZEND\_FILE\_LINE\_ORIG\_RELAY\_CC);  list->ptr = ptr; // 巨大块内存指针  list->size = size; // 内存大小  list->next = heap->huge\_list; // next指针关联到原列表开头  heap->huge\_list = list; // 本块（的指针封装）作为huge块链表开头 |

这里用到的zend\_mm\_huge\_list结构体，它的定义非常简单：

|  |
| --- |
| typedef struct \_zend\_mm\_huge\_list zend\_mm\_huge\_list; // 别名  // 巨大块内存链表元素  struct \_zend\_mm\_huge\_list {  void \*ptr; // 巨大块内存的指针  size\_t size; // 指向内存的大小  zend\_mm\_huge\_list \*next; // 指向下一个同类元素  }; |

这样就完成了巨大块内存的分配。

## **三）bitset地图**

在大块和小块的内存分配中，需要用到bitset。顾名思义，bitset是一组二进制位的集合，用来标记一个chunk中的每个page是否已使用。zend\_mm\_bitset结构是无符号整数zend\_ulong的别名，相关定义如下：

|  |
| --- |
| // 在32位操作系统中占4Bytes，64位操作系统中占8Bytes  typedef zend\_ulong zend\_mm\_bitset; /\* 4-byte or 8-byte integer \*/  // 每个BITSET 的长度，32位系统中32，64位系统中 64  #define ZEND\_MM\_BITSET\_LEN (sizeof(zend\_mm\_bitset) \* 8)  // PAGE\_MAP 的长度32位系统 512 / 32 = 16，64位系统512 / 64 = 8  #define ZEND\_MM\_PAGE\_MAP\_LEN (ZEND\_MM\_PAGES / ZEND\_MM\_BITSET\_LEN)  // 32位系统中，4Byte\*16 = 64Bytes,  // 64位系统中，8Byte \* 8 = 64Bytes, 512个bit。  typedef zend\_mm\_bitset zend\_mm\_page\_map[ZEND\_MM\_PAGE\_MAP\_LEN]; /\* 64B \*/ |

zend\_mm\_page\_map的大小在32位和64位操作系统中都是64Bytes，512个bit，分别对应每个chunk中的512个page。每个位值为0时，表示对应序号的page是空闲状态，值为1时表示对应序号的page已经被使用。

bitset是一个比较简单的数据结构，其相关的操作方法也比较简单，主要有zend\_alloc.c中的下列方法：

|  |
| --- |
| // 计算zend\_mm\_bitset右侧连续的值为1的bit的数量  static zend\_always\_inline int zend\_mm\_bitset\_nts(zend\_mm\_bitset bitset)  // 调用 ZEND\_BIT\_TEST 宏程序，验证某一个位是否是 1  static zend\_always\_inline int zend\_mm\_bitset\_is\_set(zend\_mm\_bitset \*bitset, int bit)  // 把 zend\_mm\_bitset 的某一位设置成 1  static zend\_always\_inline void zend\_mm\_bitset\_set\_bit(zend\_mm\_bitset \*bitset, int bit)  // 把 zend\_mm\_bitset 的某一位设置成 0  static zend\_always\_inline void zend\_mm\_bitset\_reset\_bit(zend\_mm\_bitset \*bitset, int bit)  // 把 zend\_mm\_bitset 的一个区域设置成 1  static zend\_always\_inline void zend\_mm\_bitset\_set\_range(zend\_mm\_bitset \*bitset, int start, int len)  // 把 zend\_mm\_bitset 的一个区域设置成 0  static zend\_always\_inline void zend\_mm\_bitset\_reset\_range(zend\_mm\_bitset \*bitset, int start, int len)  // 检验 zend\_mm\_bitset 的一个区域 是否是空的  static zend\_always\_inline int zend\_mm\_bitset\_is\_free\_range(zend\_mm\_bitset \*bitset, int start, int len) |

除以上方法外，还有zend\_portablility.h中的ZEND\_BIT\_TEST宏脚本，用来这个脚本虽然只有一行，但考虑到了兼容32位和64位系统，略显复杂：

|  |
| --- |
| // 验证某一个位是否是1。  #define ZEND\_BIT\_TEST(bits, bit) \  (((bits)[(bit) / (sizeof((bits)[0])\*8)] >> ((bit) & (sizeof((bits)[0])\*8-1))) & 1)  // 以64位系统为例梳理运行逻辑如下  // (((bits)[(bit) / (8\*8)] >> ((bit) & (8\*8-1))) & 1)  // (((bits)[(bit) / (64)] >> ((bit) & (63))) & 1)  // (((bits)[页编号] >> ( (bit)&63 )) & 1)  // (((bits)[页编号] >> bit在块中的位置 ) & 1)  // 移动后，这个bit就到了右侧第一位 & 1  // 得到这个bit 是否是 1 |

从以上方法的声名中可以看出，bitset的操作单位是 zend\_mm\_bitset而不是zend\_mm\_page\_map。为了便于理解，可以把zend\_mm\_page\_map看作是一本地图，每个zend\_mm\_bitset是其中的一页。对于地图的操作，有时候是以zend\_mm\_page\_map为单位来进行，虽然以上方法接收的参数都是zend\_mm\_bitset指针，但只要传入第一页zend\_mm\_bitset的指针，就可以针对整个zend\_mm\_page\_map进行操作。例如：

|  |
| --- |
| zend\_mm\_bitset\_is\_set(chunk->free\_map, i) |

以上方法都会自动查找需要操作的zend\_mm\_bitset页码。

在bitset功能中有一点需要特别注意，就是内存中bit的排列顺序。在每个zend\_mm\_page\_map中所有zend\_mm\_bitset的排列是从左到右。但zend\_mm\_bitset是zend\_ulong无符号整数的别名，其中bit的排列顺序是高位在左，低位在右，所以在zend\_mm\_bitset中查找bit时需要从右向左查找。例如zend\_mm\_bitset\_set\_bit方法的实现：

|  |
| --- |
| // 把某一位设置成 1  static zend\_always\_inline void zend\_mm\_bitset\_set\_bit(zend\_mm\_bitset \*bitset, int bit)  {  // ZEND\_MM\_BITSET\_LEN 32或64, Z\_UL 转成无符号 long型  bitset[bit / ZEND\_MM\_BITSET\_LEN] |= (Z\_UL(1) << (bit & (ZEND\_MM\_BITSET\_LEN-1)));  // 以64位操作系统为例  // bitset[指定的块] |= 1 << bit & 63  } |

又如zend\_mm\_bitset\_nts()方法是计算zend\_mm\_bitset右侧的连续的1的数量，因为开头在右侧，所以此方法实际上是在计算zend\_mm\_bitset开头连续的1的数量。

## **四）大块内存分配**

大于小在ZEND\_MM\_MAX\_SMALL\_SIZE（3072Bytes）和ZEND\_MM\_MAX\_LARGE\_SIZE（2M-512Bytes）之间的内存为大块的内存，需要调用zend\_mm\_alloc\_large() 方法来分配。

调用路径如下：

|  |
| --- |
| zend\_mm\_alloc\_large()-> zend\_mm\_alloc\_large\_ex() -> zend\_mm\_alloc\_pages() |

zend\_mm\_alloc\_large\_ex()方法会调用ZEND\_MM\_SIZE\_TO\_NUM()宏程序把需要的内存大小转换成page数量。再调用zend\_mm\_alloc\_pages()方法分配内存。

zend\_mm\_alloc\_pages()方法是分配内存的关键方法，逻辑比较复杂，可归成“分段遍历bitset地图，查找可用空间”、“chunk空间不够时的处理”、“在chunk中分配需要的page”三部分，下面依次介绍。

### 分段遍历bitset地图，查找可用空间

首先，在整个chunk的地图中，查找可用的空间，查找时把地图分为多个段，每个段由n（n>=0，一般情况下n>0，地图开头时可能n=0）个连续的1和m（m>=0。一般情况下m>0，地图结尾时可能m=0）个连续的0组成。每个段的结尾必定是0或地图结尾。

例如，32位操作系统中的一个zend\_mm\_bitset占4个Byte,32个bit，存储示意图如下：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Byte1 | | | | | | | | Byte2 | | | | | | | | Byte3 | | | | | | | | Byte4 | | | | | | | |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 段5 | | | 段4 | | | | | | | | | | | | | | | | | 段3 | | | | | | 段2 | | | | 段1 | |

依次遍历每个段：先跳过前面的1，然后计算后面的0的数量，如果0的数量大于所需要的page数量，说明这一块空间可用。

这部分主体逻辑比较简单，但有几个有些复杂的优化：

#### **1）在所有段中查找最佳可用位置**

但为了提高内存空间利用率，这里需要查找最佳可用位置。

在所有可用段（空闲page数量>=需要的page数量）中，找到空闲page数最接近所需要page数的段，这个段就是**最佳可用位置**。

例如：如果需要分配3个page，段3和段4都能满足要求，段3里有5个空闲page，段4里有13个空闲page。相对来讲段3是更加合适的位置。

如果在某一段里找到的空闲page数刚好和需要的相等，那就不需要再查找其他的段，可以马上跳出循环进行后续操作；否则为了找到最佳可用位置，要遍历每一个段，记录最佳位置中第一个空闲page的序号，供分配时使用。

#### **2）段的切换**

对于从一个段到下一个段的切换，这里做了一个简单又巧妙的处理，使用位运算来进行，主要代码是如下两行：

|  |
| --- |
| tmp &= tmp + 1; // 把右侧连续的1变成0，跳过这些0就可以找到下一段的开头  tmp |= tmp - 1; // 把右侧连续的0变成1，跳过这此1就可以找到空闲位置的开头 |

以上面示意图中的bitset为例，推演运算逻辑如下：

|  |
| --- |
| // 初始bitset为: 11100000 00000000 11110000 01001100  // 首次循环时，找到段1开始序号1，结尾序号2，有 2 个空闲page    tmp &= tmp + 1; => 11100000 00000000 11110000 01001100  tmp |= tmp - 1; => 11100000 00000000 11110000 01001111，  // 下一次循环时，找到段2，开始序号1，结尾序号6，有 2 个空闲位置    tmp &= tmp + 1; => 11100000 00000000 11110000 01000000  tmp |= tmp - 1; => 11100000 00000000 11110000 01111111  // 下一次循环时，找到段3，开始序号1，结尾序号12，有 5 个空闲位置    tmp &= tmp + 1; => 11100000 00000000 11110000 00000000  tmp |= tmp - 1; => 11100000 00000000 11111111 11111111  // 下一次循环时，找到段4，开始序号1，结尾序号30，有 13 个空闲位置  tmp &= tmp + 1; => 11100000 00000000 00000000 00000000  tmp |= tmp - 1; => 11111111111111111111111111111111111  // 下一次循环时，检查到此页中没有可用位置，会跳过此页到下一页。 |

如上所示，通过反复把右侧遍历过的段填充成1，每次循环都从开头位置开始计算，查找第一个段中的可用page数量。这样遍历时并不是一个bit一个bit地查找，而是一段一段跳着查找，逻辑简单并高效。

#### **3）zend\_mm\_bitset\_nts()方法**

zend\_mm\_bitset\_nts()方法用来计算一个zend\_mm\_bitset右侧（开头）连续的1的数量。代码不算复杂：

|  |
| --- |
| static zend\_always\_inline int zend\_mm\_bitset\_nts(zend\_mm\_bitset bitset)  {  int n;  // 如果所有位都是1，直接返回长度  if (bitset == (zend\_mm\_bitset)-1) return ZEND\_MM\_BITSET\_LEN;  n = 0;    #if SIZEOF\_ZEND\_LONG == 8 // 如果是64位系统  if (sizeof(zend\_mm\_bitset) == 8) {  if ((bitset & 0xffffffff) == 0xffffffff) {n += 32; bitset = bitset >> Z\_UL(32);}  }  #endif  if ((bitset & 0x0000ffff) == 0x0000ffff) {n += 16; bitset = bitset >> 16;}  if ((bitset & 0x000000ff) == 0x000000ff) {n += 8; bitset = bitset >> 8;}  if ((bitset & 0x0000000f) == 0x0000000f) {n += 4; bitset = bitset >> 4;}  if ((bitset & 0x00000003) == 0x00000003) {n += 2; bitset = bitset >> 2;}  return n + (bitset & 1);  } |

从代码中可以看出，在查找右侧的1时，并不是依次遍历每个bit，而是使用二分法查找。在32位系统中最多只需要比较5次，在64位系统中最多比较6次，即可计算出结尾的1的数量。

计算右侧连续的0的数量时，使用zend\_bitset.h中的zend\_ulong\_ntz() 方法，逻辑类似。

#### **4）zend\_mm\_bitset的切换**

在进入一个新的页时，先进行整数比较，看这个页的整数值是否等于 (zend\_mm\_bitset)-1。-1在内存中的表示是所有位都是1，所以碰到-1可以直接跳过这个页。这样只进行1次比较就可以确定64（或32）个bit的状态，不需要调用zend\_mm\_bitset\_nts()方法来计算1的数量。在内存使用比较紧凑时，这样比较效率会非常高。

同理，在检查连续的空闲位置时，先进行整数比较，看这个页的整数值是否等于0，如果等于0，说明这一页标记的所有page都是空闲状态，不再需要比较每一个bit。

|  |
| --- |
| zend\_mm\_alloc\_small()-> zend\_mm\_alloc\_small\_slow() -> zend\_mm\_alloc\_pages() |

#### **5）zend\_mm\_chunk结构体中的free\_tail元素**

free\_tail元素用来标记整个chunk中，最后一段空闲page的开头位置。例如：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| zend\_mm\_chunk中page的编号从0到511,共512个 | | | | | | | | |
| ... | page504  已使用 | page505  空闲 | page506  空闲 | page507  空闲 | page508  空闲 | page509  空闲 | page510  空闲 | page511  空闲 |
|  |  | <- free\_tail记录末尾空闲段的开头位置序号 505 | | | | | | |

在查找空闲位置时，free\_tail后面的zend\_mm\_bitset都不需要再进行查找。

有了以上几点优化，在bitset地图中查找空闲page时，速度极快效率极高。

### chunk空间不够时的处理

如果当前chunk里的空间不够，需要分成几种情况进行处理：

#### **1）尝试查找后面的chunk**

如果当前chunk的next指针没有指向main\_chunk，说明当前chunk不是chunk链表中的最后一个，后面还有可用的chunk。需要切换到下一个chunk里，继续进行查找。

|  |
| --- |
| chunk = chunk->next; // 切换到下一个 chunk  steps++; // 记录前进 1 步 |

#### **2）尝试使用缓存chunk**

当一个chunk不再使用时，为了减少频繁地创建和释放chunk带来的性能损耗，内存管理器不会直接释放它，而是把它放到缓存chunk列表中，以备后续使用。当缓存列表数量达到限制时，多余的chunk会被直接释放。(参见后续章节）

如果当前chunk的next指针指向main\_chunk，说明后面没有可用chunk。需要先查找当前heap中cached\_chunks指针指向的缓存chunk列表，如果有缓存chunk，优先使用它。代码注释如下：

|  |
| --- |
| if (heap->cached\_chunks) { // 如果有缓存chunks，先使用缓存chunk  heap->cached\_chunks\_count--; // 缓存 chunk 数 -1  chunk = heap->cached\_chunks; // 找到缓存chunk  heap->cached\_chunks = chunk->next; // 缓存chunk指向下一个  } |

#### **3）创建并初始化chunk**

如果没有缓存的chunk可用，需要创建新的chunk。在创建前，要先检查内存使用是否可能超出限制，如果会超出限制，要调用zend\_mm\_gc()方法进行垃圾回收（参见“内存的自动回收机制”章节）。相关代码注释如下：

|  |
| --- |
| // 如果可用内存小于 ZEND\_MM\_CHUNK\_SIZE，内存不足  if (UNEXPECTED(ZEND\_MM\_CHUNK\_SIZE > heap->limit - heap->real\_size)) {  if (zend\_mm\_gc(heap)) { // 回收空闲内存  .....  }  } |

如果内存没有超出限制大小，使用zend\_mm\_chunk\_alloc() 方法创建chunk，调用路径如下：

|  |
| --- |
| zend\_mm\_chunk\_alloc()-> zend\_mm\_chunk\_alloc\_int() |

zend\_mm\_chunk\_alloc\_int()方法在巨大块内存分配章节中已经介绍过。

新chunk的大小为ZEND\_MM\_CHUNK\_SIZE=2M。

创建chunk后，需要更新heap中记录的内存使用量（heap->real\_size）和chunk数量（heap->real\_size）。

然后调用zend\_mm\_chunk\_init() 方法初始化chunk。

|  |
| --- |
| static zend\_always\_inline void zend\_mm\_chunk\_init(zend\_mm\_heap \*heap, zend\_mm\_chunk \*chunk)  {  // 关联到所属 heap  chunk->heap = heap;  // 把自己放在chunk链(环）的最后，next指针指向main\_chunk  chunk->next = heap->main\_chunk;  chunk->prev = heap->main\_chunk->prev;  chunk->prev->next = chunk;  chunk->next->prev = chunk;  // 空闲page数 = 总page数-1  chunk->free\_pages = ZEND\_MM\_PAGES - ZEND\_MM\_FIRST\_PAGE;  // free\_tail 初始值为1  chunk->free\_tail = ZEND\_MM\_FIRST\_PAGE;  // chunk编号，前一个chunk的序号+1  chunk->num = chunk->prev->num + 1;  // 地图中第1个page标记成已使用  chunk->free\_map[0] = (1L << ZEND\_MM\_FIRST\_PAGE) - 1;  // 添加大块内存标记，page数量是1  chunk->map[0] = ZEND\_MM\_LRUN(ZEND\_MM\_FIRST\_PAGE);  } |

业务逻辑非常简单，新chunk的第一个page要留出来作特殊用途；在map中给第一个page添加LRUN标记。

### 在chunk中分配需要的page

经过上面的操作，现在已经找到了可用的chunk来分配需要的page空间。

这里有一个小优化：如果需要的page数少于8个，说明是小块内存分配，需要把这个chunk提到chunk链的最前面，紧跟着main\_chunk。在一般的应用场景里，小块内存的使用率最高，把存放小块内存的chunk放在最前面可以提升查找性能。

再进行最后的更新，然后返回新分配page指针。

|  |
| --- |
| // 更新剩余page数量  chunk->free\_pages -= pages\_count;  // 更新 map 把分配的这一串page标记成已使用  zend\_mm\_bitset\_set\_range(chunk->free\_map, page\_num, pages\_count);  // 添加大块内存标记，并记录page数量  chunk->map[page\_num] = ZEND\_MM\_LRUN(pages\_count);  // 如果是在最后的空闲区域中分配  if (page\_num == chunk->free\_tail) {  // 最后的可用页码后移  chunk->free\_tail = page\_num + pages\_count;  }  // 找到并返回分配的page串开头地址  return ZEND\_MM\_PAGE\_ADDR(chunk, page\_num); |

这样就完成了大块内存的分配。

## **五）小块内存分配**

大于小在ZEND\_MM\_MAX\_SMALL\_SIZE（3072Bytes）以下的内存为小块内存。

### 计算配需要的pages数量

小块内存并不是每次只分配1块，而是每次分配一串。参见ZEND\_MM\_BINS\_INFO()宏程序中预先定义的配置列表。小块内存的尺寸被分成30种，最小8Bytes，最大3072Bytes：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **num列**  **行号** | **size列**  **大小（Bytes）** | **elements列**  **每次分配数量** | **总大小（Bytes）** | **pages列**  **占用page数** | **page使用率** |
| **0** | **8** | 512 | 4096 | 1 | 100.00% |
| **1** | **16** | 256 | 4096 | 1 | 100.00% |
| **2** | **24** | 170 | 4080 | 1 | 99.61% |
| **3** | **32** | 128 | 4096 | 1 | 100.00% |
| **4** | **40** | 102 | 4080 | 1 | 99.61% |
| **5** | **48** | 85 | 4080 | 1 | 99.61% |
| **6** | **56** | 73 | 4088 | 1 | 99.80% |
| **7** | **64** | 64 | 4096 | 1 | 100.00% |
| **8** | **80** | 51 | 4080 | 1 | 99.61% |
| **9** | **96** | 42 | 4032 | 1 | 98.44% |
| **10** | **112** | 36 | 4032 | 1 | 98.44% |
| **11** | **128** | 32 | 4096 | 1 | 100.00% |
| **12** | **160** | 25 | 4000 | 1 | 97.66% |
| **13** | **192** | 21 | 4032 | 1 | 98.44% |
| **14** | **224** | 18 | 4032 | 1 | 98.44% |
| **15** | **256** | 16 | 4096 | 1 | 100.00% |
| **16** | **320** | 64 | 20480 | 5 | 100.00% |
| **17** | **384** | 32 | 12288 | 3 | 100.00% |
| **18** | **448** | 9 | 4032 | 1 | 98.44% |
| **19** | **512** | 8 | 4096 | 1 | 100.00% |
| **20** | **640** | 32 | 20480 | 5 | 100.00% |
| **21** | **768** | 16 | 12288 | 3 | 100.00% |
| **22** | **896** | 9 | 8064 | 2 | 98.44% |
| **23** | **1024** | 8 | 8192 | 2 | 100.00% |
| **24** | **1280** | 16 | 20480 | 5 | 100.00% |
| **25** | **1536** | 8 | 12288 | 3 | 100.00% |
| **26** | **1792** | 16 | 28672 | 7 | 100.00% |
| **27** | **2048** | 8 | 16384 | 4 | 100.00% |
| **28** | **2560** | 8 | 20480 | 5 | 100.00% |
| **29** | **3072** | 4 | 12288 | 3 | 100.00% |

为便于理解，直接举例说明：

|  |
| --- |
| 当需要5Bytes内存时，先找到大于5Bytes的最小配置，就是8Bytes。8Bytes的内存，每次分配512块，占用1个page。  当需要1025Bytes内存时，先找到大于它的最小配置，就是1280Bytes。1280Bytes的内存，每次分配16块，占用5个page。 |

从以上表格中可以看出，这样分配内存，并不是每一次都能占满所有的page。其中page使用率最低的是160Bytes这一档，每次分配25块，占用1个page，但是只使用page里面的4000Bytes内存，另外96个Bytes被浪费掉了；但page使用率仍然能达到97.66，整体浪费的内存很少。

从上面举出的两个例子中可以看出，除了分配时浪费的内存，使用时也会浪费一些内存。

在php程序运行的大部分场景里，都需要用到大量的小块内存，来存放诸如字串或数字等变量。这些小块内存会被频繁地分配和释放，这样分配小块内存虽然会浪费一定的内存，但可以避免频繁地向操作系统申请和释放内存，极大地提升了内存管理效率。

在内存计算机内存管理中，常常需要考虑时间和空间的平衡，牺牲少量的空间来换取时间是非常普遍的做法。

分配小块内存前，需要调用zend\_mm\_small\_size\_to\_bin()方法来把内存大小换算成使用的配置信息行号。方法声名如下：

|  |
| --- |
| static zend\_always\_inline int zend\_mm\_small\_size\_to\_bin(size\_t size) |

ZEND\_MM\_BINS\_INFO()中的列表并不是直接使用的，在运行时会分散存放到3个全局变量里方便调用：

bin\_data\_size变量存放列表中的size列；

bin\_elements变量存放列表中的elements列；

bin\_pages变量存放列表中的page列。

得到配置行号后，就可以在bin\_pages变量中取得需要分配的page数量。

### 分配pages，更新地图信息

小块内存需要调用zend\_mm\_alloc\_small()方法来分配，调用路径如下：

|  |
| --- |
| zend\_mm\_alloc\_small()-> zend\_mm\_alloc\_small\_slow() -> zend\_mm\_alloc\_pages() |

zend\_mm\_alloc\_small()方法并不复杂，有空闲内存时直接使用，没有时调用zend\_mm\_alloc\_small\_slow()方法进行分配。代码注释如下：

|  |
| --- |
| static zend\_always\_inline void \*zend\_mm\_alloc\_small(zend\_mm\_heap \*heap, int bin\_num）{  // 如果有空闲的位置可用（heap->free\_slot[bin\_num] 下面有一串空闲的small 块）  if (EXPECTED(heap->free\_slot[bin\_num] != NULL)) {  zend\_mm\_free\_slot \*p = heap->free\_slot[bin\_num]; // 直接使用这个位置  heap->free\_slot[bin\_num] = p->next\_free\_slot; // 把列表开头 指向下一个位置  return p;  // 没有空闲的位置可用  } else {  // 分配 page 创建并初始化small 块,返回一串small 块的开头  return zend\_mm\_alloc\_small\_slow(heap, bin\_num);  }  } |

注意，分配小块内存时，要传入ZEND\_MM\_BINS\_INFO()中的配置信息行号bin\_num，而不是需要的内存大小。一般使用ZEND\_MM\_SMALL\_SIZE\_TO\_BIN()把需要的内存大小转换成配置信息行号。

比较复杂的是zend\_mm\_alloc\_small\_slow()方法，它是小块内存分配的重点，声名如下：

|  |
| --- |
| // bin\_num参数是通过 zend\_mm\_small\_size\_to\_bin()方法获取到的配置信息行号  static zend\_never\_inline void \*zend\_mm\_alloc\_small\_slow(zend\_mm\_heap \*heap, uint32\_t bin\_num ZEND\_FILE\_LINE\_DC ZEND\_FILE\_LINE\_ORIG\_DC) |

和分配大块内存一样，先调用zend\_mm\_alloc\_pages()方法来分配page串。然后更新每一个page的地图信息：

|  |
| --- |
| // 在第一个page的地图中存放配置行号，并添加ZEND\_MM\_IS\_SRUN标记  chunk->map[page\_num] = ZEND\_MM\_SRUN(bin\_num);  // 如果有后面的page  if (bin\_pages[bin\_num] > 1) {  // page序号，从1开始  uint32\_t i = 1;  // 遍历后面的page  do {  // 在地图中存放 page序号、配置信息行号，并添加 ZEND\_MM\_IS\_SRUN 和 ZEND\_MM\_IS\_LRUN 标记  chunk->map[page\_num+i] = ZEND\_MM\_NRUN(bin\_num, i);  i++;  } while (i < bin\_pages[bin\_num]);  } |

### chunk中的地图信息

在chunk的定义中可以看到，除了用于标记每个page是否使用的bitset地图，还有一个map元素。map元素是一串共512个32位整数，每个page分配一个，用于存储page的地图信息。

相关mask定义如下：

|  |
| --- |
| #define ZEND\_MM\_LRUN\_PAGES\_MASK 0x000003ff  #define ZEND\_MM\_SRUN\_BIN\_NUM\_MASK 0x0000001f  #define ZEND\_MM\_SRUN\_FREE\_COUNTER\_MASK 0x01ff0000  #define ZEND\_MM\_NRUN\_OFFSET\_MASK 0x01ff0000 |

可以看出，每个32位整数被分成3段，从左向右：第一段占7位，第二段占9位，第三段占其余的16位。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Byte1 | | | | | | | | Byte2 | | | | | | | | Byte3 | | | | | | | | Byte4 | | | | | | | |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 第1段，最左侧7位 | | | | | | | 第2段，左侧8-17位 | | | | | | | | | 第3段，最右侧16位 | | | | | | | | | | | | | | | |

段1主要用来存放ZEND\_MM\_IS\_LRUN和ZEND\_MM\_IS\_SRUN标记，只使用前两个位。

段2和段3用于存放两个数字，在不同的状态中用法略有不同。

page在使用过程中有4种新状态：

1）**空状态**：新分配的chunk中没有使用过的page，地图中没有标记。

2）**LRUN状态**：只有ZEND\_MM\_IS\_LRUN标记和1个数字，由ZEND\_MM\_LRUN()宏程序来实现。用于标记大块内存。

3）**SRUN状态**：只有ZEND\_MM\_IS\_SRUN标记和1个或2个数字，由ZEND\_MM\_SRUN()宏程序和ZEND\_MM\_SRUN\_EX()宏程序来实现。用于标记小块内存的第1个page。

4）**NRUN状态**：有ZEND\_MM\_IS\_LRUN和ZEND\_MM\_IS\_SRUN两个标记，并且有2个数字，由ZEND\_MM\_NRUN()宏程序来实现。用于标记小块内存的非第2-n个page。

这几个宏程序的逻辑比较简单，描述如下：

|  |
| --- |
| // 业务逻辑：0x40000000 | count  #define ZEND\_MM\_LRUN(count)  // 业务逻辑：0x80000000 | bin\_num  #define ZEND\_MM\_SRUN(bin\_num)  // 业务逻辑：0x80000000 | bin\_num | count << 16  #define ZEND\_MM\_SRUN\_EX(bin\_num, count)  // 业务逻辑：0x80000000 | 0x40000000 | bin\_num | offset << 16  #define ZEND\_MM\_NRUN(bin\_num, offset) |

page地图操作相关方法的注释，见“其他相关方法->page地图操作相关”章节。

### 格式化小块内存列表

这里用到一个特殊的结构体zend\_mm\_free\_slot，定义如下：

|  |
| --- |
| typedef struct \_zend\_mm\_free\_slot zend\_mm\_free\_slot;  // 空闲内存位置， 多包装一层 是为了方便地指向同类原素  struct \_zend\_mm\_free\_slot {  zend\_mm\_free\_slot \*next\_free\_slot;  }; |

这个结构体非常简单，只包含一个指向同类实例的指针。它用于把分配好的小块内存串联成链，每个闲置的小块内存中，存放指向下一个闲置小块的指针：

|  |
| --- |
| // 小块内存列表结尾处，转成 zend\_mm\_free\_slot 指针  end = (zend\_mm\_free\_slot\*)((char\*)bin + (bin\_data\_size[bin\_num] \* (bin\_elements[bin\_num] - 1)));  // 小块内存链表开头的指针，每个配置一个指针，共30个指针  // heap->free\_slot[bin\_num] 本身就是第一个元素，所以它里面的指针要指向第二个元素  heap->free\_slot[bin\_num]= p = (zend\_mm\_free\_slot\*)((char\*)bin + bin\_data\_size[bin\_num]);  do { // 遍历小内列表，把元素串成链  // bin\_data\_size : ZEND\_MM\_BINS\_INFO()中的 size列  // 每个元素被转成指向下一个元素的指针  p->next\_free\_slot = (zend\_mm\_free\_slot\*)((char\*)p + bin\_data\_size[bin\_num]);  p = (zend\_mm\_free\_slot\*)((char\*)p + bin\_data\_size[bin\_num]); // 跳到下一个位置  } while (p != end);  p->next\_free\_slot = NULL; // 最后一个元素指针为NULL |

小块内存本来已经是列表了为什么要串成链呢？这是因为在使用中不断有新元素被占用，又有旧元素被随机释放，在列表中被占用的内存和闲置内存会随机排列。被占用的小块内存会从链中被删除掉，回收的闲置小块内存会串联到链里，闲置内存始终保持链状结构，方便调用。

## **六）带安全保护的内存分配**

safe\_emalloc()宏程序和ecalloc()宏程序也是常用的分配内存方法，与emalloc()宏程序不同的是，它们在分配内存前进行了内存溢出检测。它们的调用路径如下：

|  |
| --- |
| ecalloc()->\_ecalloc()->\_emalloc()->zend\_mm\_alloc\_heap()  safe\_emalloc->\_safe\_emalloc()->\_emalloc()->zend\_mm\_alloc\_heap() |

这两个方法的内务逻辑都非常简单，它们的区别是：

1）\_ecalloc()方法会把分配好的内存块全部格式化成0，\_safe\_emalloc()方法不会。

2）\_safe\_emalloc()方法多接收一个offset参数，\_ecalloc()方法没有。

代码注释如下：

|  |
| --- |
| ZEND\_API void\* ZEND\_FASTCALL \_ecalloc(size\_t nmemb, size\_t size)  {  void \*p;  size = zend\_safe\_address\_guarded(nmemb, size, 0); // 检测内存是否会溢出  p = \_emalloc(size); // 分配内存  memset(p, 0, size); // 新分配的内存全写成0  return p;  }  ZEND\_API void\* ZEND\_FASTCALL \_safe\_emalloc(size\_t nmemb, size\_t size, size\_t offset)  {  // 检测内存是否会溢出，并分配内存  return \_emalloc(zend\_safe\_address\_guarded(nmemb, size, offset) );  } |

它们在分配内存时都调用到zend\_safe\_address\_guarded()方法，它会再调用zend\_safe\_address()方法，来检测本次内存分配是否会溢出。这两个方法都在zend\_multiply.h文件中定义。

zend\_safe\_address()根据不同的操作系统和编译环境，有7种不同的定义，现仅以64位windows操作系统的定为例，方法定义如下：

|  |
| --- |
| static zend\_always\_inline size\_t zend\_safe\_address(size\_t nmemb, size\_t size, size\_t offset, bool \*overflow)  {  size\_t res = nmemb \* size + offset; // 要分配的大小，整数型  double \_d = (double)nmemb \* (double)size + (double)offset; // 结尾处指针，小数型  double \_delta = (double)res - \_d; // 整数型和小数型的差，正常应该为0  if (UNEXPECTED((\_d + \_delta ) != \_d)) { // 如果\_delta不为0  \*overflow = 1; // 状态标记成溢出  return 0; // 返回否  }  \*overflow = 0; // 状态标记成未溢出  return res; // 返回要分配的大小  }) |

以上业务逻辑是在模拟分配内存时的指针数值计算，检查指针数值是否溢出，如果溢出返回0，否则返回检验可用的内存尺寸(size)。

## **七）小结**

内存分配时根据分配的大小分为3种情况，巨大块、大块、小块。巨大块内存分配最为简单；大块内存需要分配chunk和page，比较复杂；小块内存的内存在分配page后还需要分割并串成链表，最为复杂。可由浅入深，逐步深入学习。

# **三、释放内存**

释放内存分配最主要的方式是调用zend\_alloc.h中的efree()宏程序，它的调用路径如下：

|  |
| --- |
| efree() -> \_efree() -> zend\_mm\_free\_heap() |

zend\_mm\_free\_heap()方法根据内存大小，再调用子方法进行释放。

|  |  |
| --- | --- |
| **内存大小** | **调用方法** |
| 小块内存 | zend\_mm\_free\_small() |
| 大块内存 | zend\_mm\_free\_large() |
| 巨大块内存 | zend\_mm\_free\_huge() |

## **一）判断内存块大小**

zend\_mm\_free\_heap()方法定义如下：

|  |
| --- |
| static zend\_always\_inline void zend\_mm\_free\_heap(zend\_mm\_heap \*heap, void \*ptr） |

可以看出，它只接收内存块指针，不需要传入内存块的大小。对于内存块的大小，它有自己的判断逻辑：

小块和大块内存都要用到chunk里的page，在每个chunk中，第一个page是留做特殊用途的，所以小块和大块内存的指针都不会对齐到chunk开头，只有巨大块内存的指针会对齐到chunk开头。使用ZEND\_MM\_ALIGNED\_OFFSET()来获取内存指针到chunk开头的偏移位置，如果偏移位置是0，调用zend\_mm\_free\_huge()方法释放块大块内存。

存放小块内存的page，在page地图中有ZEND\_MM\_IS\_SRUN标记，如果碰到有这个标记的page，调用zend\_mm\_free\_small()方法来释放小块内存。如果没有这个标记，调用zend\_mm\_free\_large()方法来释放大块内存。

zend\_mm\_free\_heap()方法的业务逻辑注释如下：

|  |
| --- |
| // 计算这个指针相对 ZEND\_MM\_CHUNK\_SIZE 的偏移量  size\_t page\_offset = ZEND\_MM\_ALIGNED\_OFFSET(ptr, ZEND\_MM\_CHUNK\_SIZE);  if (UNEXPECTED(page\_offset == 0)) { // 如果没有偏移量，当成huge  if (ptr != NULL) { // 并且指针有效  zend\_mm\_free\_huge(heap, ptr); // 释放这个chunk  }  // 有偏移量： large 或 small 都不会占用chunk里的第一个page,所以一定有偏移量  } else {  // 取回所在chunk的指针  zend\_mm\_chunk \*chunk = (zend\_mm\_chunk\*)ZEND\_MM\_ALIGNED\_BASE(ptr, ZEND\_MM\_CHUNK\_SIZE);  // 通过偏移量计算出page 编号  int page\_num = (int)(page\_offset / ZEND\_MM\_PAGE\_SIZE);  zend\_mm\_page\_info info = chunk->map[page\_num]; // 找到地图信息  // 如果是小块内存  if (EXPECTED(info & ZEND\_MM\_IS\_SRUN)) {  // 释放小块 ，第三个参数： page地图信息 中的配置编号  zend\_mm\_free\_small(heap, ptr, ZEND\_MM\_SRUN\_BIN\_NUM(info));  // 不是小块，当成large 块  } else /\* if (info & ZEND\_MM\_IS\_LRUN) \*/ {  // 从地图信息中，取得连续的page 数量  int pages\_count = ZEND\_MM\_LRUN\_PAGES(info);  // 释放大块内存  zend\_mm\_free\_large(heap, chunk, page\_num, pages\_count);  }  } |

## **二）释放巨大块内存**

释放巨大块内存使用zend\_mm\_free\_huge()方法，它的业务逻辑比较简单：

|  |
| --- |
| // 找到并删除 zend\_mm\_huge\_list链表元素，返回元素中存储的内存大小  size = zend\_mm\_del\_huge\_block(heap, ptr);  zend\_mm\_chunk\_free(heap, ptr, size); // 这里才是真正的释放内存 |

zend\_mm\_del\_huge\_block()方法负责遍历巨大块内存链表，找到指向要删除的内存的节点（zend\_mm\_huge\_list实例，参见“巨大块内存分配”章节），删除这个节点，并返回节点中存储的内存大小。

zend\_mm\_chunk\_free()方法调用zend\_mm\_munmap()方法来释放内存。

## **三）释放大块内存**

释放大块内存使用zend\_mm\_free\_large()方法，它的定义如下：

|  |
| --- |
| // 除heap外还要传入3个参数：  // chunk：要释放的内存的所属chunk  // page\_num：要释放的内存的起始page编号  // pages\_count：要释放的page数量  static zend\_always\_inline void zend\_mm\_free\_large(zend\_mm\_heap \*heap  , zend\_mm\_chunk \*chunk, int page\_num, int pages\_count) |

调用路径如下：

|  |
| --- |
| zend\_mm\_free\_large() ->zend\_mm\_free\_pages() -> zend\_mm\_free\_pages\_ex()  ->zend\_mm\_delete\_chunk()->zend\_mm\_chunk\_free()->zend\_mm\_munmap() |

### 把需要释放的page标记成空闲

zend\_mm\_free\_pages\_ex()方法把需要释放的page标记成空闲，它比zend\_mm\_free\_large()方法多接收1个参数：

|  |
| --- |
| // freechunk：是否删除空chunk  static zend\_always\_inline void zend\_mm\_free\_pages\_ex(zend\_mm\_heap \*heap  , zend\_mm\_chunk \*chunk, uint32\_t page\_num, uint32\_t pages\_count, int free\_chunk) |

zend\_mm\_free\_pages\_ex()方法的业务逻辑注释如下：

|  |
| --- |
| chunk->free\_pages += pages\_count; // 增加可用page数  // 更新bitset地图，把相应的page标记成空闲  zend\_mm\_bitset\_reset\_range(chunk->free\_map, page\_num, pages\_count);  chunk->map[page\_num] = 0; // 重置地图信息  // 如果被删除的page串的结尾是 == 所有已用page的结尾（后面全是空闲page）  if (chunk->free\_tail == page\_num + pages\_count) {  chunk->free\_tail = page\_num; // 更新结尾空闲page序号  }  // 如果要求释放chunk 并且chunk不是main\_chunk 并且 chunk是空的  if (free\_chunk && chunk != heap->main\_chunk && chunk->free\_pages == ZEND\_MM\_PAGES - ZEND\_MM\_FIRST\_PAGE) {  zend\_mm\_delete\_chunk(heap, chunk); // 删除这个chunk  } |

### 删除chunk前的优化操作

满足条件时，会调用zend\_mm\_delete\_chunk()方法，它并不会直接删除chunk，而是进行了一些优化操作，最重要的优化是把要删除的chunk放进缓存chunk链表中，以备循环使用。

#### **添加缓存chunk的条件**

把回收的chunk添加进缓存链表需要满足两个条件中任意一个：

1. 当前使用的chunk数小于平均使用的chunk数（heap->avg\_chunks\_count + 0.1）。

2. 当前使用的chunk数大于等于4个，并且当前使用的chunk数等于上次清理chunk时更新的阀值（heap->last\_chunks\_delete\_boundary）。

#### **删除chunk时的特殊处理**

如果被要删除的chunk的序号大于缓存chunk链表中第一个chunk的序号。需要删除缓存chunk列表的第一个chunk，并把当前要删除的chunk放到缓存链表的开头。

每个空chunk大小都是一样的，使用时没有区别。但这么做可以让序号最大的chunk要么在使用中，要么在缓存中，不会被删掉。这样新建chunk时自增的序号就不会重复。

zend\_mm\_delete\_chunk()方法的业务逻辑注释如下：

|  |
| --- |
| // 先把自己从chunk链条上摘出来  chunk->next->prev = chunk->prev;  chunk->prev->next = chunk->next;  heap->chunks\_count--; // 使用chunk数 -1  // 如果chunk数和缓存chunk数加起来，小于平均使用chunk数  // 或者 （ chunk数达到删除阀值 并且 最后一次删除时chunk数>=4个 ）  // 把chunk添加到缓存列表中  if (heap->chunks\_count + heap->cached\_chunks\_count < heap->avg\_chunks\_count + 0.1  || (heap->chunks\_count == heap->last\_chunks\_delete\_boundary  && heap->last\_chunks\_delete\_count >= 4)) {  heap->cached\_chunks\_count++; // 缓存chunk链表串中的chunk数加+1  chunk->next = heap->cached\_chunks; // next指针指向缓存chunk链表的第一个元素  heap->cached\_chunks = chunk; // 让本chunk作为缓存chunk链表的第一个元素  // 不满足添加缓存条件，触发清理  } else {  heap->real\_size -= ZEND\_MM\_CHUNK\_SIZE; // 减少已用内存数量  if (!heap->cached\_chunks) { // 如果没有缓存chunk  // 如果当前 chunk 数不等于上次清理阀值  if (heap->chunks\_count != heap->last\_chunks\_delete\_boundary) {  // 更新清理阀值为：当前使用的chunk数  heap->last\_chunks\_delete\_boundary = heap->chunks\_count;  heap->last\_chunks\_delete\_count = 0; // 上次清理数量，初始为0  // 当前 chunk 数等于上次清理阀值  } else {  heap->last\_chunks\_delete\_count++; // 最后清理时chunk数+1  }  }  // 如果没有缓存 chunk 或 当前 chunk 序号大于 最后一个缓存 chunk 序号  if (!heap->cached\_chunks || chunk->num > heap->cached\_chunks->num) {  zend\_mm\_chunk\_free(heap, chunk, ZEND\_MM\_CHUNK\_SIZE); // 释放chunk  // 如果有缓存 并且 当前chunk序号小于缓存 chunk  } else {  // next指针指向缓存chunk列表第二个元素  chunk->next = heap->cached\_chunks->next;  // 释放原第一个缓存chunk  zend\_mm\_chunk\_free(heap, heap->cached\_chunks, ZEND\_MM\_CHUNK\_SIZE);  heap->cached\_chunks = chunk; // 当前chunk排到缓存第一个  }  } |

### 释放chunk

如果满足释放条件，会调用zend\_mm\_chunk\_free()方法释放chunk。这时内存被操作系统回收，chunk被真正释放掉了。

## **四）释放小块内存**

释放小块内存使用zend\_mm\_free\_small()方法，它的业务逻辑非常简单，只需要把这块内存放到空闲内存链表的开头即可，内存并没有真的被释放掉。代码注释如下：

|  |
| --- |
| zend\_mm\_free\_slot \*p;  p = (zend\_mm\_free\_slot\*)ptr; // 指针转成 zend\_mm\_free\_slot  p->next\_free\_slot = heap->free\_slot[bin\_num]; // 指向目前的空闲块链表开头  heap->free\_slot[bin\_num] = p; // 把可用块链表开头指针指向这个块 |

在大部分应用场景中，小块内存的使用最为频繁，所以针对小块内存的管理做了非常精巧的设计：

1）使用zend\_mm\_alloc\_small\_slow()方法每次分配一串小块内存，这样可以大大减少分配次数。

2）用一个链表来管理空闲的小块内存。这样空闲的小块内存都在链表里，已使用的小块内存都链接在各自的上下文变量中。

3）从空闲列表中分配小块内存，和把回回收的小块内存放回到空闲列表里，是使用频率最高的操作。这两项操作逻辑非常简单高效，可以让性能达到最佳。

在以上业务逻辑中不难看出，小块内存并没有真正被回收释放，因为真正的内存分配和释放是以chunk为单位来进行的。要真正释放小块内存，需要用到内存的自动回收机制。

## **五）内存的自动回收机制**

一般情况下，使用php时都会开启内存限制，如果没有缓存chunk可用，并且已使用内存数量（heap->real\_size）到达限制大小，在分配内存时，会触发内存回收机制。

内存回收机制主要由zend\_mm\_gc() 方法来实现，在zend\_mm\_alloc\_pages()、zend\_mm\_realloc\_huge()、zend\_mm\_alloc\_huge()方法中都有调用到它。

zend\_mm\_gc() 方法的实现分成两个步骤：

### 第一步：整理小块内存

首先要遍历所有空闲列表（heap->free\_slot[]），遍历列表中的每一块空闲内存，计算每一串page中的空闲小块内存数量。如果有整串的空闲page，更新标记。代码注释如下：

|  |
| --- |
| zend\_mm\_free\_slot \*p, \*\*q;  p = heap->free\_slot[i];  // 遍历一串所有的小块内存  while (p != NULL) {  // 取回所在chunk的指针  chunk = (zend\_mm\_chunk\*)ZEND\_MM\_ALIGNED\_BASE(p, ZEND\_MM\_CHUNK\_SIZE);  page\_num = (int)(page\_offset / ZEND\_MM\_PAGE\_SIZE); // 到找所属page序号  info = chunk->map[page\_num]; // 找到 page 地图  // 如果有 LRUN 标记。说明是横跨多个 page 的串  if (info & ZEND\_MM\_IS\_LRUN) {  page\_num -= ZEND\_MM\_NRUN\_OFFSET(info); // 通过偏移量找到这个串的第一个page  info = chunk->map[page\_num]; // 找到这一页地图  }    free\_counter = ZEND\_MM\_SRUN\_FREE\_COUNTER(info) + 1; // 空闲数量+1  // 如果所有小块内存都是空闲的，说明这一串page没有用到  if (free\_counter == bin\_elements[i]) {  has\_free\_pages = true; // 标记成有空闲页（整串）  }  chunk->map[page\_num] = ZEND\_MM\_SRUN\_EX(i, free\_counter); // 中记录空闲块数量  p = p->next\_free\_slot; // 找到一串中的下一个 small 块  } |

如果查找到有可回收的page串，再次遍历整个链，把准备回收的小块内存，从空闲链上摘除。这里巧妙地应用了指向指针的指针（zend\_mm\_free\_slot \*\*q），简洁高效地实现了业务逻辑：

|  |
| --- |
| q = &heap->free\_slot[i]; // q一直指向可能需要更新的指针  p = \*q; // 链中第一个元素的指针  while (p != NULL) { // 遍历整个链  ......  if (ZEND\_MM\_SRUN\_FREE\_COUNTER(info) == bin\_elements[i]) { // 如果整串page都空闲  p = p->next\_free\_slot; // 跳到下一个小块  \*q = p; // 更新前面准备好的指针，让它指向下一块，这样就把当前块从链条中摘除了  } else { // 如果不是整串配置都空闲  // 如果下一个元素要被回收，要更新p->next\_free\_slot，所以先放一个指针指向它  q = &p->next\_free\_slot;  p = \*q; // 跳到下一个小块  }  } |

### 第二步：整理检查并回收空闲的chunk

这一步要从main\_chunk开始，遍历所有chunk。对每个使用中的chunk进行两步操作：

1）在每个chunk从头到尾按顺序遍历page，碰到空闲的整串小块内存，就回收它；碰到大块内存直接跳过。对于使用中的小块内存串，要清空前面调用ZEND\_MM\_SRUN\_EX()宏程序写入的空闲块数量free\_counter，这个空闲块数量只有在回收过程中出现。

2）遍历后，检查chunk是否有使用中的page，如果没有，调用zend\_mm\_delete\_chunk() 方法回收这个chunk。

代码注释如下：

|  |
| --- |
| chunk = heap->main\_chunk; // 从main\_chunk开始  do {  i = ZEND\_MM\_FIRST\_PAGE; // 从第一页开始  while (i < chunk->free\_tail) { // 按顺序遍历，从头到最后一个已使用的page  if (zend\_mm\_bitset\_is\_set(chunk->free\_map, i)) { // 如果这个page 已使用  info = chunk->map[i]; // 地图信息  if (info & ZEND\_MM\_IS\_SRUN) { // 如果是 小块内存  // 取回 ZEND\_MM\_BINS\_INFO() 配置行号  int bin\_num = ZEND\_MM\_SRUN\_BIN\_NUM(info);  int pages\_count = bin\_pages[bin\_num]; // 连续使用的page数量  // 如果整串page全部空闲（空闲的数量 = 配置数量）  if (ZEND\_MM\_SRUN\_FREE\_COUNTER(info) == bin\_elements[bin\_num]) {  zend\_mm\_free\_pages\_ex(heap, chunk, i, pages\_count, 0); // 回收这串page  collected += pages\_count; // 记录回收数量  } else { // page中还有在使用的块  // 重新记录 bin\_num ，这里清空了 前面的 free\_counter 记录  chunk->map[i] = ZEND\_MM\_SRUN(bin\_num);  }  i += bin\_pages[bin\_num]; // 跳过存放小块内存的page串  } else { // 不是小块就是大块，巨大块不在这个链里  i += ZEND\_MM\_LRUN\_PAGES(info); // 取得 大块内存的 的page数 ，跳过它  }  } else { // 如果没使用，跳过  i++;  }  }  // 如果整个chunk都空闲了  if (chunk->free\_pages == ZEND\_MM\_PAGES - ZEND\_MM\_FIRST\_PAGE) {  zend\_mm\_chunk \*next\_chunk = chunk->next; // 下一个chunk的指针  zend\_mm\_delete\_chunk(heap, chunk); // 回收这个chunk  chunk = next\_chunk; // 处理下个chunk  } else { // chunk 还在使用  chunk = chunk->next; // 处理下一个chunk  }  } while (chunk != heap->main\_chunk); // 遍历所有chunk |

需要注意的是：

1）第一步并没有挨个遍历所有的page。对于大块和小块内存，只要找到这串page的开头，就可以决定是否回收这串page；然后跳过整串page。这样一段一段跳着遍历，速度很快。

2）整个回收过程中，只是先回收扑空闲的小块内存链，再回收空闲的chunk。并没有做碎片整理类的工作，所以回收能力也比较有限。

3）回收机制是在分配内存时触发，做的工作少，花费的时间也少。如果要做进一步的回收，会让对分配内存时速度更慢。时间和空间的平衡，总是无法两全齐美。

## **六）小结**

回收内存的业务逻辑本身不如分配内存复杂，但为了提高性能，在过程中进行了很一些优化。zend\_mm\_gc()垃圾回收机制就是为了优化小块内存而生。

# **四、调整内存大小**

释放内存分配最主要的方式是调用zend\_alloc.h中的erealloc()宏程序和safe\_erealloc()宏程序，它们的调用路径如下：

|  |
| --- |
| erealloc() -> \_erealloc() -> zend\_mm\_realloc\_heap()  safe\_erealloc()->\_safe\_erealloc()->\_erealloc()-> zend\_mm\_realloc\_heap() |

safe\_erealloc()方法在调整内存大小前，先进行内存溢出检测（参见：“带安全保护的内存分配”章节），后续操作与erealloc()相同，都是调用zend\_mm\_realloc\_heap()方法进行操作。

zend\_mm\_realloc\_heap()方法的逻辑比较复杂，从它开始介绍。

## **一）zend\_mm\_realloc\_heap()方法**

zend\_mm\_realloc\_heap()方法声名如下：

|  |
| --- |
| // 参数use\_copy\_size：是否只复制要求的尺寸  // 参数copy\_size：要求复制的尺寸  static zend\_always\_inline void \*zend\_mm\_realloc\_heap(zend\_mm\_heap \*heap, void \*ptr, size\_t size, bool use\_copy\_size, size\_t copy\_size) |

zend\_mm\_realloc\_heap()方法的业务逻辑比较复杂，里面有10多种情况的处理，5层嵌套的if判断。

### 业务逻辑分支表

为了便于理解，可以把业务逻辑梳理成图表。图表看似复杂，其实结构非常简单清晰，能把复杂业务逻辑分割成多个分支，各个分支之间耦合很小；并且每个分支的逻辑都比较简单，逐个攻破难度都不大，这样就有效地降低了整体的理解难度。整理图表虽然要花一些时间，但可以把学习难度转化成多一些工作量，不失为学习的好方法。

zend\_mm\_realloc\_heap()方法的业务逻辑分支如下：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **第1层 条件** | **第2层**  **条件** | **第3层 条件** | **第4层 条件** | **第5层 条件** | **处理逻辑** | **分支**  **编号** |
| 原本是 巨大块内存 | 原指针无效 | | | | 分配新内存，调用 \_zend\_mm\_alloc() | 1.1 |
| 原指针有效 | | | | 调整大小,调用 zend\_mm\_realloc\_huge() | 1.2 |
| 原本是 大块或小块内存 | 原本是小块 | 如果要把内存分配得更小 | 如果能转成 更小的小块内存 | | 转成更小的小块内存 删除原来的小块内存 | 2.1.1.1 |
| 不能转成更小一级的小块内存 | | 什么也不做 | 2.1.1.2 |
| 如果要更大尺寸，但还在小块内存范围内 | | | 转成更大的小块内存 删除原来的小块内存 | 2.1.2 |
| 需要大块或巨大块内存 | | | 调用 zend\_mm\_realloc\_slow() | 2.1.3 |
| 原本是大块 | 如果需要的尺寸在大块内存范围内 | 如果新的和旧的 大小一样 | | 什么也不做 | 2.2.1.1 |
| 如果要分配得更小 | | 大块直接截短 | 2.2.1.2 |
| 如果要分配得更大 | 如果本块后面有足够多的空闲page | 添加page | 2.2.1.3.1 |
| page不够用 | 调用 zend\_mm\_realloc\_slow() | 2.2.1.3.2 |
| 需要的尺寸 不在大块范围内 | | | 调用 zend\_mm\_realloc\_slow() | 2.2.2 |

如上所示，每多嵌套一层条件，表格中也多一列条件，对应的单元格中有条件的简述。右侧第一列为条件分支编号，每多一层条件加一个数字，这样从分支编号上就可以追踪条件判断的路径。右侧第二列为此分支的处理逻辑简述。

从业务逻辑分支表中可以看出，大部分情况都在zend\_mm\_realloc\_heap()方法中处理完了。表格原本是按方法中的业务逻辑顺序，从上到下整理，但各分支之间没有依赖，所以可以不必依照顺序，把业务逻辑归纳说明。

### 业务逻辑归纳说明

#### **1）内部处理完毕的分支**

分支2.1.1.1，把小块内存转成更小块内存。如果开启了use\_copy\_size，并且copy\_size小于新分配的大小，只复制copy\_size规定大小的内容。

分支2.1.1.2，如果小块内存不能转成更小的小块，什么也不用做。如果开启了use\_copy\_size，并且copy\_size小于内存原本大小，只复制copy\_size规定大小的内容。

分支2.1.2，把小块内存转成大一些的小块内存。

分支2.2.1.1，大块内存调整时，没有改变大小，什么也不做。

分支2.2.1.2，大块内存截短，截短后大小还在大块范围内，进行截短操作。

分支2.2.1.3.1，大块内存增加长度，增加后大小还在大块范围内，并且后面有足够多page，追加一些page。

以上分支的逻辑都不复杂，参见代码注释。

#### **2）use\_copy\_size参数**

由于分支众多，use\_copy\_size参数比较容易让人迷惑。通过以上分支的归纳可以看出，只有分支2.1.1.2和分支2.1.2使用use\_copy\_size参数。

#### **3）需要调用 zend\_mm\_realloc\_slow()方法处理的分支**

分支2.1.3，把小块内存转成大块或巨大块内存。

分支2.2.1.3.2，给大块内存增加尺寸，但page不够用。

分支2.2.2，把大块内存转成小块或巨大块内存。

zend\_mm\_realloc\_slow()方法的业务逻辑很简单，代码注释如下：

|  |
| --- |
| static zend\_never\_inline void \*zend\_mm\_realloc\_slow(zend\_mm\_heap \*heap, void \*ptr, size\_t size, size\_t copy\_size )  {  void \*ret;  ret = zend\_mm\_alloc\_heap(heap, size); // 分配内存  memcpy(ret, ptr, copy\_size); // 把原有内容copy过来  zend\_mm\_free\_heap(heap, ptr); // 释放原有内容  return ret; // 返回新内存地址  } |

#### **4）处理巨大块内存**

分支1.1，如果原本没有巨大块内存，调用\_zend\_mm\_alloc()方法来分配。\_zend\_mm\_alloc()方法调用路径如下：

|  |
| --- |
| \_zend\_mm\_alloc() -> zend\_mm\_alloc\_heap() |

zend\_mm\_alloc\_heap()方法在分配内存章节已经介绍过。

分支1.1，如果已经存在巨大内存块，需要调用zend\_mm\_realloc\_huge()方法来调整大小，这个方法比较复杂，后面单独介绍。

### zend\_mm\_realloc\_heap()方法的代码注释

zend\_mm\_realloc\_heap()方法代码注释如下，为了方便阅读，在每个条件语句上都添加了分支编号，编号与前文一致：

|  |
| --- |
| // 找到 相对于 ZEND\_MM\_CHUNK\_SIZE 的偏移量  page\_offset = ZEND\_MM\_ALIGNED\_OFFSET(ptr, ZEND\_MM\_CHUNK\_SIZE);  if (UNEXPECTED(page\_offset == 0)) { // 情况1：无偏移，是huge块  if (EXPECTED(ptr == NULL)) { // 情况1.1：如果指针无效  return \_zend\_mm\_alloc(heap, size); // 直接开辟新内存  } else { // 情况1.2：原huge块有效  return zend\_mm\_realloc\_huge(heap, ptr, size, copy\_size); // 调整huge块大小  }  } else { // 情况2：有偏移，是small 或 large  // 取回所在chunk的指针  zend\_mm\_chunk \*chunk = (zend\_mm\_chunk\*)ZEND\_MM\_ALIGNED\_BASE(ptr, ZEND\_MM\_CHUNK\_SIZE);  int page\_num = (int)(page\_offset / ZEND\_MM\_PAGE\_SIZE); // 计算page页码  zend\_mm\_page\_info info = chunk->map[page\_num]; // 地图信息    if (info & ZEND\_MM\_IS\_SRUN) { // 情况2.1： 如果是small 块  // 取回 ZEND\_MM\_BINS\_INFO() 中的配置行号  int old\_bin\_num = ZEND\_MM\_SRUN\_BIN\_NUM(info);    do { // 这个do是为了break  old\_size = bin\_data\_size[old\_bin\_num]; // ZEND\_MM\_BINS\_INFO中的 size  if (size <= old\_size) { // 情况2.1.1：如果想把内存划分得更小  // 情况2.1.1.1： 旧的行号>0 并且新尺寸小于 旧序号的前一个行号的尺寸  // 必须满足这个要求才可以把内存划得更小  if (old\_bin\_num > 0 && size < bin\_data\_size[old\_bin\_num - 1]) {  // 计算 size 在 ZEND\_MM\_BINS\_INFO 数组里的序号，size 最大是3072  ret = zend\_mm\_alloc\_small(heap, ZEND\_MM\_SMALL\_SIZE\_TO\_BIN(size));  // 要复制的内容大小，use\_copy\_size：只复制这么多，多余的丢掉  copy\_size = use\_copy\_size ? MIN(size, copy\_size) : size;  memcpy(ret, ptr, copy\_size); // 复制内容到新地址  zend\_mm\_free\_small(heap, ptr, old\_bin\_num); // 释放原来的小块  } else { // 情况2.1.1.2： 如果不能把内存变得更小，就不重新分配  ret = ptr;  }    // 情况2.1.2：size <= 3072，目标是大一些的 small 块  } else if (size <= ZEND\_MM\_MAX\_SMALL\_SIZE) {  ret = zend\_mm\_alloc\_small(heap, ZEND\_MM\_SMALL\_SIZE\_TO\_BIN(size)); // 分配 small 块  copy\_size = use\_copy\_size ? MIN(old\_size, copy\_size) : old\_size; // 确定要copy的大小  memcpy(ret, ptr, copy\_size); // 复制内容  zend\_mm\_free\_small(heap, ptr, old\_bin\_num); // 释放原来的small 块  } else { // 情况2.1.3： 需要的 尺寸大于 small 块，从small 块转到large或huge块  break; // 调用zend\_mm\_realloc\_slow()方法  }    return ret; // 返回指针  } while (0);    // 情况2.2：如果是 large 块  } else /\* if (info & ZEND\_MM\_IS\_LARGE\_RUN) \*/ {  // 这一串page的总大小  old\_size = ZEND\_MM\_LRUN\_PAGES(info) \* ZEND\_MM\_PAGE\_SIZE;  // 情况2.2.1：如果需要的尺寸在 large 大小范围内  if (size > ZEND\_MM\_MAX\_SMALL\_SIZE && size <= ZEND\_MM\_MAX\_LARGE\_SIZE) {  new\_size = ZEND\_MM\_ALIGNED\_SIZE\_EX(size, ZEND\_MM\_PAGE\_SIZE); // 大小向后对齐  if (new\_size == old\_size) { // 情况2.2.1.1：如果新的和旧的大小一样  return ptr; // 直接返回，不用重新分配了  // 情况2.2.1.2：如果要分配得更小，large 块直接截短就行，不需要像small 块去考虑配置  } else if (new\_size < old\_size) {  int new\_pages\_count = (int)(new\_size / ZEND\_MM\_PAGE\_SIZE); // 需要的 page 数量  // 新旧相差的 page 数量 。  int rest\_pages\_count = (int)((old\_size - new\_size) / ZEND\_MM\_PAGE\_SIZE);  chunk->map[page\_num] = ZEND\_MM\_LRUN(new\_pages\_count); // 更新这个配置的地图信息，large块的page数量  chunk->free\_pages += rest\_pages\_count; // 把多出来page加到空闲中  // 结尾这一段标记成空闲  zend\_mm\_bitset\_reset\_range(chunk->free\_map, page\_num + new\_pages\_count, rest\_pages\_count);  return ptr; // 返回指针    } else { // 情况2.2.1.3：如果要分配得更大  int new\_pages\_count = (int)(new\_size / ZEND\_MM\_PAGE\_SIZE); // 新的页数  int old\_pages\_count = (int)(old\_size / ZEND\_MM\_PAGE\_SIZE); // 旧的页数  // 情况2.2.1.3.1：如果后面有足够多的page， 并且 这些 page 全是空闲的  if (page\_num + new\_pages\_count <= ZEND\_MM\_PAGES &&  zend\_mm\_bitset\_is\_free\_range(chunk->free\_map, page\_num + old\_pages\_count, new\_pages\_count - old\_pages\_count)) {  chunk->free\_pages -= new\_pages\_count - old\_pages\_count; // 减少空闲page  // 把后面的page标记成使用  zend\_mm\_bitset\_set\_range(chunk->free\_map, page\_num + old\_pages\_count, new\_pages\_count - old\_pages\_count);  // 更新地图，把新的page 数写进进去（只更新page串中第一个page的地图）  chunk->map[page\_num] = ZEND\_MM\_LRUN(new\_pages\_count);  return ptr; // 返回指针  }  // 情况2.2.1.3.2： 如果后面的page不够用，调用zend\_mm\_realloc\_slow()方法  } // else结束，无其他情况  }  // 情况2.2.2： 如果需要的尺寸不在 large 块范围内，调用zend\_mm\_realloc\_slow()方法  } // else结束，无其他情况  } // else结束，无其他情况  // 处理前面没有覆盖到的情况：情况2.1.3、情况2.2.1.3.2、情况2.2.2  copy\_size = MIN(old\_size, copy\_size);// 要复制的大小  return zend\_mm\_realloc\_slow(heap, ptr, size, copy\_size); // 重新创建内存 |

## **二）zend\_mm\_realloc\_huge()方法**

调整已有的巨大块内存大小，需要用到zend\_mm\_realloc\_huge()方法，此方法和相关方法的业务逻辑都不太复杂，但有一些针对不同操作系统的优化。代码注释如下：

|  |
| --- |
| static zend\_never\_inline void \*zend\_mm\_realloc\_huge(zend\_mm\_heap \*heap, void \*ptr, size\_t size, size\_t copy\_size)  {  size\_t old\_size;  size\_t new\_size;  old\_size = zend\_mm\_get\_huge\_block\_size(heap, ptr); // 先获取原来的大小  // 如果要求大小大于：2m-4Byte = 2044K（2044K测试过）  if (size > ZEND\_MM\_MAX\_LARGE\_SIZE) {  #ifdef ZEND\_WIN32 // windows操作系统  // 在windows系统中无法随时扩大huge块的空间，每次增加2M，减少调整次数。  // REAL\_PAGE\_SIZE=4K，ZEND\_MM\_CHUNK\_SIZE=2M, 向后对齐到2M  new\_size = ZEND\_MM\_ALIGNED\_SIZE\_EX(size, MAX(REAL\_PAGE\_SIZE, ZEND\_MM\_CHUNK\_SIZE));  #else // 其他操作系统  // 向后对齐 到 REAL\_PAGE\_SIZE  new\_size = ZEND\_MM\_ALIGNED\_SIZE\_EX(size, REAL\_PAGE\_SIZE);  #endif  if (new\_size == old\_size) { // 情况1，如果调整前后大小一样  zend\_mm\_change\_huge\_block\_size(heap, ptr, new\_size);  return ptr; // 内存指针  } else if (new\_size < old\_size) { // 情况2，要求减少空间  if (zend\_mm\_chunk\_truncate(heap, ptr, old\_size, new\_size)) { // 减少空间，如果成功  heap->real\_size -= old\_size - new\_size; // 减少已使用内存  zend\_mm\_change\_huge\_block\_size(heap, ptr, new\_size ); // 更新块尺寸  return ptr; // 内存指针  }  // 如果 zend\_mm\_chunk\_truncate() 失败，要调用 zend\_mm\_realloc\_slow()方法  } else /\* if (new\_size > old\_size) \*/ { // 情况3，要求 增加空间  if (zend\_mm\_chunk\_extend(heap, ptr, old\_size, new\_size)) { // 增加空间，如果成功  heap->real\_size += new\_size - old\_size; // 增加已使用内存  zend\_mm\_change\_huge\_block\_size(heap, ptr, new\_size ); // 更新块尺寸  return ptr; // 内存指针  }  // 如果 zend\_mm\_chunk\_extend() 失败，调用 zend\_mm\_realloc\_slow()方法  }  }  // 可能到这里的情况：要求的大小小于 2044K 或 前面调整内存大小失败  return zend\_mm\_realloc\_slow(heap, ptr, size, MIN(old\_size, copy\_size) );  } |

zend\_mm\_realloc\_huge()方法还调用到3个方法，zend\_mm\_change\_huge\_block\_size()方法、zend\_mm\_chunk\_truncate()方法和zend\_mm\_chunk\_extend()方法。

在调用zend\_mm\_chunk\_truncate()方法和zend\_mm\_chunk\_extend()方法返回失败时，会再调用zend\_mm\_realloc\_slow()方法进行分配。

### zend\_mm\_change\_huge\_block\_size()方法

zend\_mm\_change\_huge\_block\_size()方法用于更新巨大块内存的大小，业务逻辑比较简单，代码注释如下：

|  |
| --- |
| static void zend\_mm\_change\_huge\_block\_size(zend\_mm\_heap \*heap, void \*ptr, size\_t size)  {  zend\_mm\_huge\_list \*list = heap->huge\_list;  while (list != NULL) { // 遍历巨大块列表  if (list->ptr == ptr) { // 找到这个块  list->size = size; // 更新size  return; // 完成  }  list = list->next; // 没找到，找下一个  }  } |

### zend\_mm\_chunk\_truncate()方法

zend\_mm\_chunk\_truncate()方法用于截短已有的巨大块内存，业务逻辑比较简单，代码注释如下：

|  |
| --- |
| static int zend\_mm\_chunk\_truncate(zend\_mm\_heap \*heap, void \*addr, size\_t old\_size, size\_t new\_size)  {  #ifndef \_WIN32 // windows系统  // 释放一部分内存 从old\_size 到 new\_size，失败时交给zend\_mm\_realloc\_slow()方法  zend\_mm\_munmap((char\*)addr + new\_size, old\_size - new\_size);  return 1;  #else // 非windows系统什么也不做，交给zend\_mm\_realloc\_slow()方法  return 0;  #endif  } |

### zend\_mm\_chunk\_extend()方法

zend\_mm\_chunk\_extend()方法用于为已有的巨大块内存追加长度，业务逻辑比较简单，代码注释如下：

|  |
| --- |
| static int zend\_mm\_chunk\_extend(zend\_mm\_heap \*heap, void \*addr, size\_t old\_size, size\_t new\_size)  {  #ifdef HAVE\_MREMAP // 如果有mremap()方法  // 使用mremap()方法调整内存大小  void \*ptr = mremap(addr, old\_size, new\_size, 0);  if (ptr == MAP\_FAILED) { // 如果调整失败，返回0。交给zend\_mm\_realloc\_slow()方法  return 0;  }  return 1; // 调整成功返回1  #elif !defined(\_WIN32) // 如果不是windows操作系统  // zend\_mm\_mmap\_fixed 失败时，交给zend\_mm\_realloc\_slow()方法  return (zend\_mm\_mmap\_fixed((char\*)addr + old\_size, new\_size - old\_size) != NULL);  #else // windows 无法动态扩展内存大小。交给zend\_mm\_realloc\_slow()方法  return 0;  #endif  } |

## **三）erealloc2方法**

erealloc2()方法与erealloc()的调用路径全相同，只是在调用zend\_mm\_realloc\_heap()方法时，打开了复制大小限制：

|  |
| --- |
| // 第三个参数1，为只复制指定大小的内存  return zend\_mm\_realloc\_heap(AG(mm\_heap), ptr, size, 1, copy\_size); |

关于此限制的影响，在介绍zend\_mm\_realloc\_heap()方法时已经说明，代码中也有注释。

## **四）小结**

不难发现，zend\_mm\_realloc\_heap()方法和zend\_mm\_realloc\_huge()方法高度依赖zend\_mm\_realloc\_slow()方法。

其实可以这样理解：zend\_mm\_realloc\_slow()方法是通用的调整内存大小的方法，zend\_mm\_realloc\_huge()方法、zend\_mm\_realloc\_slow()以及其他相关方法都是针对各种特殊情况进行的优化。可见为了提升性能，内存管理器的设计者下过很多功夫。

# **五、其他相关方法**

## **一）estrdup()宏程序及相关方法**

estrdup()宏程序和estrndup()宏程序的调用路径如下：

|  |
| --- |
| estrdup(s)->\_estrdup()  estrndup(s, length)->\_estrndup() |

\_estrdup()方法和\_estrndup()方法用于创建内存，并把给定的字串(char \*)复制到新内存里。

zend\_strndup()方法与前面两个方法类似，不过它不使用内存管理器，而是使用原生的malloc()方法分配内存。

这几个方法业务逻辑简单且相似，但在php扩展程序中大量被调用。\_estrndup()方法的代码注释如下：

|  |
| --- |
| ZEND\_API char\* ZEND\_FASTCALL \_estrndup(const char \*s, size\_t length)  {  char \*p;  if (UNEXPECTED(length + 1 == 0)) { // 检测整数溢出  zend\_error\_noreturn(E\_ERROR, "Possible integer overflow in memory allocation (1 \* %zu + 1)", length);  }  p = (char \*) \_emalloc(length + 1); // 分配内存, 要多分配一个Byte用来存放结束字符 \0  memcpy(p, s, length); // 复制内容，按指定长度  p[length] = 0; // 添加 \0  return p;  } |

## **二）调用原生方法管理内存**

与原生内存管理方法相关的宏程序也被大量调用。它们都带有persistent参数，当传入1时，表示使用原生内存管理方法。例如：

|  |
| --- |
| // persistent参数传入1时，表示使用原生内存管理方法  #define pefree(ptr, persistent) ((persistent)?free(ptr):efree(ptr))  #define pemalloc(size, persistent) ((persistent)?\_\_zend\_malloc(size):emalloc(size))  #define pecalloc(nmemb, size, persistent) ((persistent)?\_\_zend\_calloc((nmemb), (size)):ecalloc((nmemb), (size)))  #define perealloc(ptr, size, persistent) ((persistent)?\_\_zend\_realloc((ptr), (size)):erealloc((ptr), (size)))  #define pestrdup(s, persistent) ((persistent)?\_\_zend\_strdup(s):estrdup(s))  #define pestrndup(s, length, persistent) ((persistent)?zend\_strndup((s),(length)):estrndup((s),(length)))  #define safe\_pemalloc(nmemb, size, offset, persistent) ((persistent)?\_safe\_malloc(nmemb, size, offset):safe\_emalloc(nmemb, size, offset))  #define safe\_perealloc(ptr, nmemb, size, offset, persistent) ((persistent)?\_safe\_realloc((ptr), (nmemb), (size),(offset)):safe\_erealloc((ptr), (nmemb), (size), (offset))) |

当启用原生内存管理方法时，调用路径如下：

|  |
| --- |
| pefree()->free()  pemalloc()->\_\_zend\_malloc()->malloc()  pecalloc()->\_\_zend\_calloc()->\_\_zend\_malloc()->malloc()  perealloc()->\_\_zend\_realloc()->realloc()  pestrdup()->\_\_zend\_strdup()->strdup()  pestrndup()->malloc()  safe\_pemalloc()->\_safe\_malloc()->pemalloc()->\_\_zend\_malloc()->malloc()  safe\_perealloc()->\_safe\_realloc()->perealloc()->\_\_zend\_realloc()->realloc() |

最终调用的是free(),malloc(),realloc(),strdup()这几个原生方法。可以看出内存管理器提供的相关接口，命名都与原生方法相似：efree()、emalloc()、erealloc()、estrdup()都只在在原生方法名前面加了一个字母e。

## **三）page地图操作相关**

以下是操作page地图时用到的常量和宏程序，在管理小块和大块内存时经常用到：

|  |
| --- |
| #define ZEND\_MM\_IS\_LRUN 0x40000000 // 大块标记，0100 + 4\*7=28 个0  #define ZEND\_MM\_IS\_SRUN 0x80000000 // 小块标记，1000 + 4\*7=28 个0  #define ZEND\_MM\_LRUN\_PAGES\_MASK 0x000003ff // 大块内存的page数量掩码, 右侧 10个1  #define ZEND\_MM\_SRUN\_BIN\_NUM\_MASK 0x0000001f // 小块内存的page数量掩码, 右侧 5个1  #define ZEND\_MM\_SRUN\_FREE\_COUNTER\_MASK 0x01ff0000 // 左侧8-16位，共9个1  #define ZEND\_MM\_NRUN\_OFFSET\_MASK 0x01ff0000 // 左侧8-16位，共9个1  #define ZEND\_MM\_LRUN\_PAGES\_OFFSET 0 // 大块内存数量的  #define ZEND\_MM\_SRUN\_BIN\_NUM\_OFFSET 0 // bin\_num 配置序号的偏移位数  #define ZEND\_MM\_SRUN\_FREE\_COUNTER\_OFFSET 16 // 空闲数量的偏移位数  #define ZEND\_MM\_NRUN\_OFFSET\_OFFSET 16 // 存放offset 的偏移位数  // 取回large块使用的page数量  #define ZEND\_MM\_LRUN\_PAGES(info) (((info) & ZEND\_MM\_LRUN\_PAGES\_MASK) >> ZEND\_MM\_LRUN\_PAGES\_OFFSET)  // 取得当前 page 在 ZEND\_MM\_BINS\_INFO() 中的配置编号，最大31  #define ZEND\_MM\_SRUN\_BIN\_NUM(info) (((info) & ZEND\_MM\_SRUN\_BIN\_NUM\_MASK) >> ZEND\_MM\_SRUN\_BIN\_NUM\_OFFSET)  // 垃圾回收过程中, 取回空闲page数量（内存回收时临时用到，用完就清空了）  #define ZEND\_MM\_SRUN\_FREE\_COUNTER(info) (((info) & ZEND\_MM\_SRUN\_FREE\_COUNTER\_MASK) >> ZEND\_MM\_SRUN\_FREE\_COUNTER\_OFFSET)  // 在存放小块内存串时，取回page在串里的序号（page数肯定大于1）  #define ZEND\_MM\_NRUN\_OFFSET(info) (((info) & ZEND\_MM\_NRUN\_OFFSET\_MASK) >> ZEND\_MM\_NRUN\_OFFSET\_OFFSET)  // 添加大块内存标记 ： 0x40000000 | count。 count 是pages 在large块里的序号  #define ZEND\_MM\_LRUN(count) (ZEND\_MM\_IS\_LRUN | ((count) << ZEND\_MM\_LRUN\_PAGES\_OFFSET))  // 记录配置编号 并 添加小块内存标记 ： 0x80000000 | bin\_num  #define ZEND\_MM\_SRUN(bin\_num) (ZEND\_MM\_IS\_SRUN | ((bin\_num) << ZEND\_MM\_SRUN\_BIN\_NUM\_OFFSET))  // 写入bin\_num和空闲page数量： 0x80000000 | bin\_num | count << 16 （内存回收时临时用到，用完就清空了）  #define ZEND\_MM\_SRUN\_EX(bin\_num, count) (ZEND\_MM\_IS\_SRUN | ((bin\_num) << ZEND\_MM\_SRUN\_BIN\_NUM\_OFFSET) | ((count) << ZEND\_MM\_SRUN\_FREE\_COUNTER\_OFFSET))  // 同时添加大块、小块标记，并记录配置行号和偏移量 ：0x80000000 | 0x40000000 | bin\_num | offset << 16  #define ZEND\_MM\_NRUN(bin\_num, offset) (ZEND\_MM\_IS\_SRUN | ZEND\_MM\_IS\_LRUN | ((bin\_num) << ZEND\_MM\_SRUN\_BIN\_NUM\_OFFSET) | ((offset) << ZEND\_MM\_NRUN\_OFFSET\_OFFSET)) |

## **四）哈希表相关**

哈希表相关的宏程序有两个，都被大量调用，它们的业务逻辑非常简单：

|  |
| --- |
| // 分配内存创建哈希表  #define ALLOC\_HASHTABLE(ht) (ht) = (HashTable \*) emalloc(sizeof(HashTable))  // 释放哈希表  #define FREE\_HASHTABLE(ht) efree\_size(ht, sizeof(HashTable)) |

## **五）其他辅助类**

以下是一些频繁使用的计算内存大小类宏程序，逻辑不算复杂但大量使用位运算：

|  |
| --- |
| // size向后对齐到8： ( size + 7 ) & -8。 +7然后清除最后3个bit  #define ZEND\_MM\_ALIGNED\_SIZE(size) (((size) + ZEND\_MM\_ALIGNMENT - 1) & ZEND\_MM\_ALIGNMENT\_MASK)  // 所有的alignment一定是2的幂  // size向后对齐到alignment：(size + n个1) & n个0。  #define ZEND\_MM\_ALIGNED\_SIZE\_EX(size, alignment) \  (((size) + ((alignment) - 1)) & ~((alignment) - 1))  // 计算 size（通常是指针）到指定块大小的偏移量。与size % alignment相等，但位运算更快  #define ZEND\_MM\_ALIGNED\_OFFSET(size, alignment) \  (((size\_t)(size)) & ((alignment) - 1))  // 把 size 向左对齐到 alignment。常用于找到内存块(chunk或page)的开头  #define ZEND\_MM\_ALIGNED\_BASE(size, alignment) \  (((size\_t)(size)) & ~((alignment) - 1))  // 计算一个 size 是多少个 alignment，除不尽的向前进1  #define ZEND\_MM\_SIZE\_TO\_NUM(size, alignment) \  (((size\_t)(size) + ((alignment) - 1)) / (alignment)) |

# **六、结语**

本篇涵盖了zend\_alloc内存管理工具的主要功能。另外一些内容例如：内存使用统计、自定义内存管理方法以及调试相关的方法与主体功能关系不大，本篇没介有绍。内存限制(ZEND\_MM\_LIMIT)相关代码比较简单，为了节省篇幅，做了一些删减。

# 未解疑问

zend\_mm\_hugepage()是在做什么？

删除chunk时，if (!heap->cached\_chunks)，不是初始状态也会发生吗？

\_\_builtin\_constant\_p 是什么

It contains bitset of free pages, few bitset for available runs of predefined small sizes, map of pages that keeps information about usage of each page in this CHUNK, etc. 这句话怎么译比较好？

# Change Log

2024.10.29 创建

2024.10.29 巨大块内存分配

2024.10.30 bitset地图

2024.11.2 遍历bitset中的优化。大块内存分配；创建chunk

2024.11.6 小块内存分配

2024.11.7 释放内存

2024.11.8 垃圾回收

2024.11.9 调整内存大小，内存保护

2024.11.10 estrdump()方法，原生内存管理方法