# 一、基本概念

数组（array）是PHP程序设计语言中最具特色的数据类型，它强大、灵活、便捷、使用简单，是PHP程序开发者最好的伙伴。本篇重点介绍数组的设计原理和底层业务逻辑，相关文件主要有zend\_types.h、zend\_hash.c、zend\_hash.h等。

建议阅读本篇前先阅读“类型篇”，了解变量（zval），字符串（zend\_string)，引用类型（zend\_reference）等基本数据结构。

本篇中很少有PHP脚本代码示例，因为从PHP脚本到数组的底层数据结构，中间隔着语法解析、编译和执行这一系列过程。相关示例会在语法解析、编译和执行等独立篇章中给出。

本篇使用的源码版本是8.2.5。调试环境为64位windows系统、Microsoft Visual Studio 2013。本篇重点介绍64位操作系统中的实现逻辑，忽略32位系统中的实现逻辑，也不进行差异比较。

## 一）数据结构

数组的数据结构定义在zend\_types.h文件中，定义如下：

|  |
| --- |
| typedef struct \_zend\_array zend\_array;  typedef struct \_zend\_array HashTable;  struct \_zend\_array { // 共占56bytes  zend\_refcounted\_h gc; // 计数器（参见“类型篇”），占8Bytes  union { // 联合类型，占4Bytes  struct {  ZEND\_ENDIAN\_LOHI\_4( // 分成4段，兼容高位在前和低位在前  zend\_uchar flags, // 标记，1Byte  zend\_uchar \_unused, // 空闲字节，1Byte  zend\_uchar nIteratorsCount, // 计数器数量，1Byte  zend\_uchar \_unused2) // 第二个空闲字节，1Byte  } v;  uint32\_t flags; // 标记，32位整数，4个Bytes打包在一起  } u;  uint32\_t nTableMask; // 32位整数，4个Bytes  union { // 联合类型，都是指针，64位系统中占8Bytes  uint32\_t \*arHash; // 哈希索引列表指针  Bucket \*arData; // buckets列表指针  zval \*arPacked; // 顺序数组元素列表指针  };  uint32\_t nNumUsed; // 已使用元素数，32位整数，4个Bytes  uint32\_t nNumOfElements; // 有效元素数，32位整数，4个Bytes  uint32\_t nTableSize; // 分配元素数，32位整数，4个Bytes  uint32\_t nInternalPointer; // 内部指针，32位整数，4个Bytes  zend\_long nNextFreeElement; // 下一个空元素位置，32位整数，4个Bytes  dtor\_func\_t pDestructor; // 销毁器（函数原型），64位系统占8个Bytes  }; |

如上所示，数组的结构体为\_zend\_array，它有两个别名：zend\_array和HashTable。\_zend\_array结构体占用56Bytes内存，包含10个元素：

1）gc：引用计数器，占8Bytes（详情参见“类型篇”）。

2）u：占4Bytes，它是一个联合类型，第一个子类型v被划分成4个字节：第1个字节用来存放标记（flags）；第3个字节nIteratorsCount用来存放迭代器数量；第2、4两个字节没有用到。

为了与第二个子类型（uint32\_t类型）兼容，第一个子类型v需要考虑高位在前、低位在前两种存储方式，通过ZEND\_ENDIAN\_LOHI\_4()宏程序来兼容这两种存储方式：

|  |
| --- |
| #ifdef WORDS\_BIGENDIAN  # define ZEND\_ENDIAN\_LOHI\_4(a, b, c, d) d; c; b; a; // 高位在前  #else  # define ZEND\_ENDIAN\_LOHI\_4(a, b, c, d) a; b; c; d; // 低位在前 |

3）nTableMask：32位整数，用于把哈希值转换成元素顺序号。

4）第4个元素是联合类型，它的3个子类型都是指针类型，在64位系统中占8Bytes。arHash指针用于查找哈希索引列表。arData指针用于查找键值对（Bucket）列表。arPacked指针用于查找顺序数组元素列表。**这个元素是联合类型，所以这三个指针占用的内存位置和指向的内存位置是相同的**。（ht->arHash、ht->arData、ht->arPacked占用同一块内存，指向同一个位置，实际上它们是同一个指针）。

5）nNumUsed：从开头到最后一个有效元素，这一段元素的数量。

6）nNumOfElements：数组中有效元素的数量。

7）nTableSize：给数组分配的元素总数。

8）nInternalPointer：数组的内部指针，存放元素序号。

9）nNextFreeElement：数组的下一个可用元素的编号，一般情况下与nNumUsed相同。

10）pDestructor：数组元素的销毁器。

zend\_array是一个固定大小的结构，不像zend\_string可以伸缩，它只用来存放数组的头部信息，数组的数据部分放在单独的内存块中。对于每个元素的用法，后续会展开详细介绍。

## 二）基本常量和宏程序

与数组相关的一些基本常量定义如下。

用在\_zend\_array.u.flags元素上的标记值：

|  |  |  |
| --- | --- | --- |
| **常量名** | **值** | **说明** |
| HASH\_FLAG\_PACKED | (1<<2) | 顺序数组 |
| HASH\_FLAG\_UNINITIALIZED | (1<<3) | 未初始化的数组 |
| HASH\_FLAG\_STATIC\_KEYS | (1<<4) | 静态键名（数组中只有整数索引号和保留字串作为键名） |
| HASH\_FLAG\_HAS\_EMPTY\_IND | (1<<5) | 数组中存在空间接引用元素 |

用于添加或更新元素的操作类型常量：

|  |  |  |
| --- | --- | --- |
| **常量名** | **值** | **说明** |
| HASH\_UPDATE | (1<<0) | 更新元素 |
| HASH\_ADD | (1<<1) | 添加新元素 |
| HASH\_UPDATE\_INDIRECT | (1<<2) | 更新间接引用目标 |
| HASH\_ADD\_NEW | (1<<3) | 直接添加新元素 |
| HASH\_ADD\_NEXT | (1<<4) | 直接添加新元素，自动处理索引号 |
| HASH\_LOOKUP | (1<<5) | 查找原素 |

其他常用常量：

|  |  |  |
| --- | --- | --- |
| **常量名** | **值** | **说明** |
| HT\_INVALID\_IDX | ((uint32\_t) -1) | 无效的哈希索引值 |
| HT\_MIN\_MASK | ((uint32\_t) -2) | 最小掩码 |
| HT\_MIN\_SIZE | 8 | 数组最小元素数，8 |
| HT\_MAX\_SIZE | 0x40000000 | 数组最大元素数，1G |
| ZEND\_LONG\_MIN | INT64\_MIN | 系统常量，64位整数的最小值 |
| GC\_FLAGS\_SHIFT | 0 | 位移数，0位，不需要位移 |

一些常用的宏程序定义如下：

|  |
| --- |
| // 访问数组的标识  #define HT\_FLAGS(ht) (ht)->u.flags  // 验证数组是否是顺序数组  #define HT\_IS\_PACKED(ht) ((HT\_FLAGS(ht) & HASH\_FLAG\_PACKED) != 0)  // 验证数组中是否有空元素  #define HT\_IS\_WITHOUT\_HOLES(ht) ((ht)->nNumUsed == (ht)->nNumOfElements)  // 验证哈希表是否初始化过  #define HT\_IS\_INITIALIZED(ht) ((HT\_FLAGS(ht) & HASH\_FLAG\_UNINITIALIZED) == 0)  // 把数组标记成未初始化  #define HT\_INVALIDATE(ht) do { \  HT\_FLAGS(ht) = HASH\_FLAG\_UNINITIALIZED; \  } while (0)  // 验证数组中是否只有静态键（只用整数或保留字做键名）  #define HT\_HAS\_STATIC\_KEYS\_ONLY(ht) \  ((HT\_FLAGS(ht) & (HASH\_FLAG\_PACKED|HASH\_FLAG\_STATIC\_KEYS)) != 0)  // 找到数组中第n个元素  #define ZEND\_HASH\_ELEMENT(\_\_ht, \_idx) \  ZEND\_HASH\_ELEMENT\_EX(\_\_ht, \_idx, ZEND\_HASH\_ELEMENT\_SIZE(\_\_ht))  // 找到数组中第n个元素  #define ZEND\_HASH\_ELEMENT\_EX(\_\_ht, \_idx, \_size) \  ((zval\*)(((char\*)(\_\_ht)->arPacked) + ((\_idx) \* (\_size))))  // 获取数组中单个元素大小（优化版本：避免调用 HT\_IS\_PACKED() 宏程序）  # define ZEND\_HASH\_ELEMENT\_SIZE(\_\_ht) \  (sizeof(zval) + (~HT\_FLAGS(\_\_ht) & HASH\_FLAG\_PACKED) \* ((sizeof(Bucket)-sizeof(zval))/HASH\_FLAG\_PACKED))  // 找到数组的下一个元素  #define ZEND\_HASH\_NEXT\_ELEMENT(\_el, \_size) ((zval\*)(((char\*)(\_el)) + (\_size)))  // 找到数组的上一个元素  #define ZEND\_HASH\_PREV\_ELEMENT(\_el, \_size) ((zval\*)(((char\*)(\_el)) - (\_size)))  # define HT\_IDX\_TO\_HASH(idx) (idx) // 64位系统中 index 不用转换  # define HT\_HASH\_TO\_IDX(idx) (idx) // 64位系统中 index 不用转换  // 定位数组的数据块的开头位置  #define HT\_GET\_DATA\_ADDR(ht) \  /\* 指针向左移动跳过哈希索引列表，找到数据块开头 \*/ \  ((char\*)((ht)->arData) - HT\_HASH\_SIZE((ht)->nTableMask))  // 把ht->arData指针指向元素列表开头  #define HT\_SET\_DATA\_ADDR(ht, ptr) do { \  /\* 跳过 哈希索引列表大小 \*/ \  (ht)->arData = (Bucket\*)(((char\*)(ptr)) + HT\_HASH\_SIZE((ht)->nTableMask)); \  } while (0)  // 把zval实例的类型设置为IS\_UNDEF  #define ZVAL\_UNDEF(z) Z\_TYPE\_INFO\_P(z) = IS\_UNDEF; |

ZVAL\_UNDEF()宏程序在数组操作中被大量使用，它和Z\_TYPE\_INFO\_P()等相关宏程序的详细介绍参见“类型篇”。

## 三）创建数组

创建数组时，需要先创建zend\_array（等同于HashTable）实例。内核代码中常使用emalloc(sizeof(HashTable));或malloc(sizeof(HashTable));语句创建哈希表实例。

内存管理器zend\_alloc.h中还提供了快速创建和释放数组的宏程序：

|  |
| --- |
| // 创建数组实例  #define ALLOC\_HASHTABLE(ht) (ht) = (HashTable \*) emalloc(sizeof(HashTable))  // 释放数组实例  #define FREE\_HASHTABLE(ht) efree\_size(ht, sizeof(HashTable)) |

以PHP源码中大量引用的zend\_new\_array()宏程序为例，在64位windows系统中，它调用\_zend\_new\_array()函数来创建哈希表，代码如下：

|  |
| --- |
| # define zend\_new\_array(size) \_zend\_new\_array(size)  // 快速创建并初始化zend\_array实例  ZEND\_API HashTable\* \_zend\_new\_array(uint32\_t nSize){  HashTable \*ht = emalloc(sizeof(HashTable)); // 分配内存创建哈希表  \_zend\_hash\_init\_int(ht, nSize, ZVAL\_PTR\_DTOR, 0); // 初始化哈希表  return ht; // 返回新哈希表指针  } |

如上所示，创建zend\_array实例后，需要对它进行初始化。

### 初始化数组实例

如前文代码中所示，创建zend\_array实例后，需要调用\_zend\_hash\_init\_int()函数来初始化它：

|  |
| --- |
| // 初始化哈希表，只设置基础属性，未创建元素空间。p1:哈希表，p2:元素空间数量，p3:元素销毁器，p4:是否使用malloc()函数分配数据块  static void \_zend\_hash\_init\_int(HashTable \*ht, uint32\_t nSize, dtor\_func\_t pDestructor, bool persistent){  GC\_SET\_REFCOUNT(ht, 1); // 初始引用次数为 1 次  // 类型信息：GC\_ARRAY（IS\_ARRAY）+（malloc()函数分配？GC\_PERSISTENT+不可回收：0）  GC\_TYPE\_INFO(ht) = GC\_ARRAY |  (persistent ? ((GC\_PERSISTENT|GC\_NOT\_COLLECTABLE) << GC\_FLAGS\_SHIFT) : 0);  HT\_FLAGS(ht) = HASH\_FLAG\_UNINITIALIZED; // 标记成未初始化  ht->nTableMask = HT\_MIN\_MASK; // #define HT\_MIN\_MASK ((uint32\_t) -2)  HT\_SET\_DATA\_ADDR(ht, &uninitialized\_bucket); // 设置 p1->arData 到无效位置  ht->nNumUsed = 0; // 已使用元素数 0  ht->nNumOfElements = 0; // 有效元素数 0  ht->nInternalPointer = 0; // 内部指针  ht->nNextFreeElement = ZEND\_LONG\_MIN; // 下一个空元素，指向无效位置  ht->pDestructor = pDestructor; // 销毁器  ht->nTableSize = zend\_hash\_check\_size(nSize); // 根据操作系统设置大小  } |

GC\_SET\_REFCOUNT()宏程序的相关介绍参见“类型篇”。

zend\_hash\_check\_size()函数用校正元素数量，为了让内存管理更加高效，需要把元素数量向上对齐到2的幂，函数代码注释如下：

|  |
| --- |
| static uint32\_t zend\_hash\_check\_size(uint32\_t nSize) {  if (nSize <= HT\_MIN\_SIZE) { // 给出的数量过小，（如果初始值是0，会走这里）  return HT\_MIN\_SIZE; // 使用最小值，8  } else if (UNEXPECTED(nSize >= HT\_MAX\_SIZE)) { // 数量过大  // 报错：内存不足  zend\_error\_noreturn(E\_ERROR, "Possible integer overflow in memory allocation (%u \* %zu + %zu)", nSize, sizeof(Bucket), sizeof(Bucket));  }  nSize -= 1; // -1最后会加回来  nSize |= (nSize >> 1); // 运算后，左侧最多有2个连续的1  nSize |= (nSize >> 2); // 运算后，左侧最多有4个连续的1  nSize |= (nSize >> 4); // 运算后，左侧最多有8个连续的1  nSize |= (nSize >> 8); // 运算后，左侧最多有16个连续的1  // 运算后，左侧最多有31个连续的1（初始值是0才会有32个1，这种情况在前面过滤了）  nSize |= (nSize >> 16);  return nSize + 1; // 最后加1，变成2的幂  } |

\_zend\_hash\_init\_int()函数的衍生函数和宏程序如下：

|  |
| --- |
| // 初始化数组实例，这个宏程序的引用次数最多  #define zend\_hash\_init(ht, nSize, pHashFunction, pDestructor, persistent) \  \_zend\_hash\_init((ht), (nSize), (pDestructor), (persistent))  // 直接调用 \_zend\_hash\_init\_int()函数  ZEND\_API void \_zend\_hash\_init(HashTable \*ht, uint32\_t nSize, dtor\_func\_t pDestructor, bool persistent) {  \_zend\_hash\_init\_int(ht, nSize, pDestructor, persistent);  } |

这时只创建了数组（zend\_array）实例，还没有为数组分配元素列表空间。

为了提升性能和节省内存空间，数组的元素列表有两种实现方式：**顺序数组**和**哈希表**。顺序数组在实现上比较简单，只需要一串zval实例作为数组元素即可，哈希表的结构更复杂一些。先从比较简单的顺序数组开始介绍。

# 二、顺序数组

## 一）初始化顺序数组

zend\_hash\_real\_init\_packed()函数用于初始化顺序数组并分配内存创建元素列表，它调用zend\_hash\_real\_init\_packed\_ex()函数来完成初始化操作：

|  |
| --- |
| static void zend\_hash\_real\_init\_packed\_ex(HashTable \*ht) {  void \*data; // 数据指针  // 第一步，分配内存创建元素列表  // 如果需要用malloc()函数分配内存  if (UNEXPECTED(GC\_FLAGS(ht) & IS\_ARRAY\_PERSISTENT)) {  // 计算大小并分配内存。pemalloc()宏程序会调用malloc()函数，在zend\_alloc.h中  data = pemalloc(HT\_PACKED\_SIZE\_EX(ht->nTableSize, HT\_MIN\_MASK), 1);  } else if (EXPECTED(ht->nTableSize == HT\_MIN\_SIZE)) { // 如果元素数为8个  // 计算大小并使用heap分配内存  data = emalloc(HT\_PACKED\_SIZE\_EX(HT\_MIN\_SIZE, HT\_MIN\_MASK));  } else { // 其他元素数  // 计算内存大小并使用heap分配内存  data = emalloc(HT\_PACKED\_SIZE\_EX(ht->nTableSize, HT\_MIN\_MASK));  }  // 第二步，让哈希表的->arData指针指向元素列表的开头位置  HT\_SET\_DATA\_ADDR(ht, data);  // 第三步，添加标记：顺序数组、使用静态键名  ht->u.v.flags = HASH\_FLAG\_PACKED | HASH\_FLAG\_STATIC\_KEYS;  // 第四步，顺序数组的哈希索引表里有2个元素，都设置成 -1  HT\_HASH\_RESET\_PACKED(ht);  } |

GC\_FLAGS()宏程序的相关介绍参见“类型篇”。计算大小后使用pemalloc()函数或emalloc()函数分配内存创建元素列表，这两个函数的相关介绍参见“内存管理篇”。

如上所示，初始化过程分为4步。

### 第一步，计算大小并分配内存，创建元素列表

先调用到HT\_PACKED\_SIZE\_EX()宏程序，计算需要的空间大小，相关定义如下：

|  |
| --- |
| #define HT\_MIN\_MASK ((uint32\_t) -2) // 最小掩码：-2  // 计算顺序数组的 数据大小: zval列表大小 + 哈希索引列表大小。p1:元素数，p2:掩码  #define HT\_PACKED\_SIZE\_EX(nTableSize, nTableMask) \  /\* zval列表大小 + 哈希列表大小 \*/ \  (HT\_PACKED\_DATA\_SIZE((nTableSize)) + HT\_HASH\_SIZE((nTableMask)))  // 计算zval列表大小：元素数量 \* zval实例大小  #define HT\_PACKED\_DATA\_SIZE(nTableSize) ((size\_t)(nTableSize) \* sizeof(zval))  // 把掩码换算成哈希索引列表大小。例如，掩码HT\_MIN\_MASK=-2转成正数为2，结果为2\*4 =8  #define HT\_HASH\_SIZE(nTableMask) \  /\* 掩码转成正数 x 32位整数的大小（4Bytes） \*/ \  (((size\_t)-(uint32\_t)(nTableMask)) \* sizeof(uint32\_t)) |

如上所示，元素列表的大小 = zval实例列表的大小 + 哈希索引列表大小。在顺序数组中，调用HT\_PACKED\_DATA\_SIZE()宏程序来计算元素列表大小，每个元素是一个zval实例，zval实例列表大小 = 元素数量 \* zval实例大小。哈希索引列表的大小通过HT\_HASH\_SIZE()宏程序来计算，在顺序数组中，列表固定为2个32位整数的大小，8Bytes。

### 第二步，更新元素列表指针。

调用HT\_SET\_DATA\_ADDR()宏程序，让哈希表的arData指针元素指向新数据块中元素列表的开头。

以最小的顺序数组（有8个元素）为例，顺序数的数据块结构示意图如下：

|  |  |  |
| --- | --- | --- |
| 哈希索引表 | 索引值0 | -1 |
| 索引值1 | -1 |
| arData -->  zval  元  素  列  表 | zval 0 | --- |
| zval 1 | --- |
| zval 2 | --- |
| zval 3 | --- |
| zval 4 | --- |
| zval 5 | --- |
| zval 6 | --- |
| zval 7 | --- |

如上所示，在内存中，元素列表在哈希索引列表后面。顺序数组不会用到哈希索引表，哈希索引表中固定有2个元素，它们的值都是-1。ht->arData指针指向zval元素列表的第一个实例。

### 第三步，添加标记

这里用到两个标记：HASH\_FLAG\_PACKED表示本实例为顺序数组，HASH\_FLAG\_STATIC\_KEYS标记表示，本数组中所有键名都是静态的，不会变化。在顺序数组中，并没有用到键名，所以键名不会变化，HASH\_FLAG\_STATIC\_KEYS标记也会一直存在。

### 第四步，把哈希索引列表里的两个元素都设置为-1

调用HT\_HASH\_RESET\_PACKED()宏程序，相关业务逻辑比较简单：

|  |
| --- |
| // 重置顺序数组的哈希索引列表，两个值都更新成-1  #define HT\_HASH\_RESET\_PACKED(ht) do { \  HT\_HASH(ht, -2) = HT\_INVALID\_IDX; \  HT\_HASH(ht, -1) = HT\_INVALID\_IDX; \  } while (0)  // 调用HT\_HASH\_EX()宏程序进行操作  #define HT\_HASH(ht, idx) HT\_HASH\_EX((ht)->arHash, idx)  // 找到哈希索引列表中的指定序号的元素，idx是负数，向左查找  #define HT\_HASH\_EX(data, idx) ((uint32\_t\*)(data))[(int32\_t)(idx)] |

如上所示，(ht)->arHash与(ht)->arData是同一个指针，它指向哈希索引列表的结尾，通过它向左位移，可以找到哈希索引列表中每一元素。

## 二）顺序数组中追加元素

zend\_hash\_next\_index\_insert()函数和zend\_hash\_next\_index\_insert\_new()函数用于在顺序数组中追加元素使用。

### zend\_hash\_next\_index\_insert()函数

zend\_hash\_next\_index\_insert()函数在PHP源码中被大量调用，代码如下：

|  |
| --- |
| ZEND\_API zval\* zend\_hash\_next\_index\_insert(HashTable \*ht, zval \*pData) {  // 在数组的第一个空闲位置（ht->nNextFreeElement）插入元素  // 操作类型为 HASH\_ADD | HASH\_ADD\_NEXT  return \_zend\_hash\_index\_add\_or\_update\_i(ht, ht->nNextFreeElement, pData, HASH\_ADD | HASH\_ADD\_NEXT);  } |

在顺序数组中增加、修改元素都需要用到\_zend\_hash\_index\_add\_or\_update\_i()函数，这个函数是全篇内容中最重要也最复杂的函数之一，在本章和哈希表的相关章节都会介绍。为了便于理解，还在附录中添加了此函数的完整注释和业务逻辑分支表。

\_zend\_hash\_index\_add\_or\_update\_i()函数声名如下：

|  |
| --- |
| // 数组中添加或更新元素。p1:哈希表，p2:顺序号，p3:新元素，p4:操作类型  static zval \*\_zend\_hash\_index\_add\_or\_update\_i(HashTable \*ht, zend\_ulong h, zval \*pData, uint32\_t flag) {  uint32\_t nIndex; // 哈希索引表中的序号  uint32\_t idx; // 元素序号  Bucket \*p; // 键值对  zval \*zv; // 值  // 这个判断是针对未初始化数组的，如果在下一个位置添加，并且 位置是 最小整数  if ((flag & HASH\_ADD\_NEXT) && h == ZEND\_LONG\_MIN) {  h = 0; // 位置是 0  } |

这个函数业务逻辑比较复杂，每一个操作只用到了它的小部分业务逻辑。为了便于理解，对于每一个操作，只介绍相关的业务逻辑，忽略与本次操作无关的业务逻辑。

在本次调\_zend\_hash\_index\_add\_or\_update\_i()函数，使用HASH\_ADD和HASH\_ADD\_NEXT两个标记。根据插入元素的位置，相关业务逻辑分为1种无效情况和3种有效情况，一种无效情况如下：

|  |
| --- |
| // 紧接上一块代码  if (HT\_IS\_PACKED(ht)) { // 顺序数组的处理逻辑  // 传入h参数值为ht->nNextFreeElement，在顺序数组中与ht->nNumUsed相等，不走这里  if ((flag & (HASH\_ADD\_NEW|HASH\_ADD\_NEXT)) != (HASH\_ADD\_NEW|HASH\_ADD\_NEXT)  && h < ht->nNumUsed) { |

如上所示，在顺序数组中，ht->nNextFreeElement表示下一个空闲元素的位置，ht->nNumUsed表示已使用元素数，这两个值是相等的，所以不满足条件。

**第一种有效情况：要求位置在已分配大小内，不需要分配内存，直接插入元素**。代码如下：

|  |
| --- |
| // 紧接上一块代码  } else if (EXPECTED(h < ht->nTableSize)) { // 如果位置在已分配大小内  add\_to\_packed: // 跳转点：添加到顺序数组  zv = ht->arPacked + h; // 获取指定元素  if ((flag & (HASH\_ADD\_NEW|HASH\_ADD\_NEXT)) !=  (HASH\_ADD\_NEW|HASH\_ADD\_NEXT)) { // 只有 HASH\_ADD\_NEW标记，走这里  if (h > ht->nNumUsed) { // 如果位置超过了已使用元素数量  zval \*q = ht->arPacked + ht->nNumUsed; // 定位到【最后一个已使用元素】  while (q != zv) { // 遍历【最后一个已使用元素】到目标元素之间的所有元素  ZVAL\_UNDEF(q); // 这一段元素都设置成无效（无效和空闲是不同的）  q++; // 移动到下一个元素  }  }  }  // 更新下一个空闲元素编号，和已使用元素数量  ht->nNextFreeElement = ht->nNumUsed = h + 1;  ht->nNumOfElements++; // 有效元素数 +1  if (flag & HASH\_LOOKUP) { // 没有HASH\_LOOKUP标记，不走这里  return &p->val; // 查找时返回元素  } else { // 走这个分支  ZVAL\_COPY\_VALUE(zv, pData); // 复制zval实例，并给目标对象增加引用次数  }  return zv; // 返回当前元素 |

**第二种有效情况：要求的序号大于当前数组元素数量，并且满足2个条件：a）数组元素数量翻倍后，数量可以覆盖要求的序号h；b）当前哈希表已经使用一半以上**：

|  |
| --- |
| // 紧接上一块代码  } else if ((h >> 1) < ht->nTableSize && (ht->nTableSize >> 1) < ht->nNumOfElements) {  zend\_hash\_packed\_grow(ht); // 顺序数组大小翻倍  goto add\_to\_packed; // 跳转：重新尝试添加. 并且返回 |

如上所示，**如果满足条件，分配内存，让数组的元素数量翻倍**，再跳转到第一种有效情况处理。

**第三种有效情况：要求的序号大于当前数组元素数量，并且不满足第二种有效情况。**这时扩展数组元素列表来适应指定的序号就太不经济，会占用过多内存，所以把数组转成哈希表来处理：

|  |
| --- |
| // 紧接上一块代码  } else { // 序号大于元素数，又不适合翻倍，转成哈希表  if (ht->nNumUsed >= ht->nTableSize) { // 如果空间不够用了  ht->nTableSize += ht->nTableSize; // 大小翻倍  }  convert\_to\_hash: // 跳转点：转成哈希表  zend\_hash\_packed\_to\_hash(ht); // 顺序数组转哈希表  ... // 再按哈希表处理  } |

### zend\_hash\_next\_index\_insert\_new()函数

zend\_hash\_next\_index\_insert\_new()函数与zend\_hash\_next\_index\_insert()函数相似，但增加了一个HASH\_ADD\_NEXT标记：

|  |
| --- |
| ZEND\_API zval\* zend\_hash\_next\_index\_insert\_new(HashTable \*ht, zval \*pData) {  // 在数组的第一个空闲位置（ht->nNextFreeElement）插入元素  // 操作类型为 HASH\_ADD | HASH\_ADD\_NEXT | HASH\_ADD\_NEXT  return \_zend\_hash\_index\_add\_or\_update\_i(ht, ht->nNextFreeElement, pData, HASH\_ADD | HASH\_ADD\_NEW | HASH\_ADD\_NEXT);  } |

调用\_zend\_hash\_index\_add\_or\_update\_i()函数时，HASH\_ADD\_NEXT标记会影响第一种有效情况的业务逻辑：

|  |
| --- |
| // 第一种有效情况  } else if (EXPECTED(h < ht->nTableSize)) { // 如果位置在已分配大小内  add\_to\_packed: // 跳转点：添加到顺序数组  zv = ht->arPacked + h; // 转到指定元素  **// 有 HASH\_ADD\_NEW | HASH\_ADD\_NEXT两个标记，跳过这里**  if ((flag & (HASH\_ADD\_NEW|HASH\_ADD\_NEXT)) !=  (HASH\_ADD\_NEW|HASH\_ADD\_NEXT)) {  ...  }  // 后面的业务逻辑没有变化 |

如上所示，添加HASH\_ADD\_NEXT标记后，**会直接找到目标位置插入元素，不更新这个位置前面的空闲元素**。这样前两种有效情况的操作会更少一些，速度也更快一些。但在内核中zend\_hash\_next\_index\_insert\_new()函数的使用次数比zend\_hash\_next\_index\_insert()函数少一些。

### 给未初始化的数组添加元素

在添加元素时，有可能数组并没有被初始化过，\_zend\_hash\_index\_add\_or\_update\_i()函数中有针对这种情况的处理：

|  |
| --- |
| // 紧接第三种有效情况  } else if (HT\_FLAGS(ht) & HASH\_FLAG\_UNINITIALIZED) { // 数组没有初始化过  if (h < ht->nTableSize) { // 访问索引号在分配的大小内  zend\_hash\_real\_init\_packed\_ex(ht); // 初始化顺序数组  goto add\_to\_packed; // 跳转点：添加元素  }  zend\_hash\_real\_init\_mixed(ht); // 访问索引号不在在分配的大小内，初始化成哈希表  // 再按哈希表进行处理 |

如上所示，如果访问的元素序号在数组元素数量内，初始化数组并插入元素。否则，要把数据初初始化成混合数组再进行处理，这部分业务逻辑在“哈希表”章节中介绍。

### 顺序数组增加元素空间

如前文所述，给顺序数组增加空间需要使用zend\_hash\_packed\_grow()函数：

|  |
| --- |
| ZEND\_API void zend\_hash\_packed\_grow(HashTable \*ht) {  if (ht->nTableSize >= HT\_MAX\_SIZE) { // 如果哈希表大小达到最大大小，报错: 内存溢出  zend\_error\_noreturn(E\_ERROR, "Possible integer overflow in memory allocation (%u \* %zu + %zu)", ht->nTableSize \* 2, sizeof(Bucket), sizeof(Bucket));  }  ht->nTableSize += ht->nTableSize; // 大小 翻倍  // ht->arData指针指向重新分配的内存块的，元素列表开头（可能创建了新内存块）  HT\_SET\_DATA\_ADDR(ht,  perealloc2(HT\_GET\_DATA\_ADDR(ht), // 找到原数据块开头位置  HT\_PACKED\_SIZE\_EX(ht->nTableSize, HT\_MIN\_MASK), // 计算新大小  HT\_PACKED\_USED\_SIZE(ht), // 需要复制的旧数据的大小  GC\_FLAGS(ht) & IS\_ARRAY\_PERSISTENT) // 是否使用malloc()函数分配内存  );  } |

HT\_PACKED\_USED\_SIZE()宏程序用于计算数顺序组数据块中已使用部分的大小：

|  |
| --- |
| #define HT\_PACKED\_USED\_SIZE(ht) \  /\* 哈希索引列表大小（固定8Bytes） + 已使用元素数量x元素大小 \*/ \  (HT\_HASH\_SIZE((ht)->nTableMask) + ((size\_t)(ht)->nNumUsed \* sizeof(zval))) |

perealloc2()函数的相关介绍参见“内存管理篇”。

## 三）顺序数组中更新元素

顺序数组中更新元素相关的函数有3个：

|  |
| --- |
| // 顺序数组中找到旧元素直接中断，找不到时转成哈希表进行操作  ZEND\_API zval\* zend\_hash\_index\_add(HashTable \*ht, zend\_ulong h, zval \*pData) {  return \_zend\_hash\_index\_add\_or\_update\_i(ht, h, pData, HASH\_ADD);  }  // 顺序数组中找到旧元素直接中断，找不到时转成哈希表进行操作  ZEND\_API zval\* zend\_hash\_index\_add\_new(HashTable \*ht, zend\_ulong h, zval \*pData) {  return \_zend\_hash\_index\_add\_or\_update\_i(ht, h, pData, HASH\_ADD | HASH\_ADD\_NEW);  }  // 找到旧元素都更新。找不到旧元素都添加，数组会转成哈希表  ZEND\_API zval\* zend\_hash\_index\_update(HashTable \*ht, zend\_ulong h, zval \*pData) {  return \_zend\_hash\_index\_add\_or\_update\_i(ht, h, pData, HASH\_UPDATE);  } |

如上所示，这3个函数都调用\_zend\_hash\_index\_add\_or\_update\_i()函数，只是传入的标记（flag）不同。这3个函数都可能碰到两种情况：

情况一，**当h参数的值大于等于ht->nNumUsed时**，3个函数业务逻辑都与zend\_hash\_next\_index\_insert()函数相同。

情况二，**h参数的值小于ht->nNumUsed时**，执行\_zend\_hash\_index\_add\_or\_update\_i()函数时业务逻辑少有不同。

|  |
| --- |
| if (HT\_IS\_PACKED(ht)) { // 如果是顺序数组  **// 不是 HASH\_ADD\_NEW|HASH\_ADD\_NEXT ，并且访问序号小于 ht->nNumUsed**  if ((flag & (HASH\_ADD\_NEW|HASH\_ADD\_NEXT)) != (HASH\_ADD\_NEW|HASH\_ADD\_NEXT)  && h < ht->nNumUsed) {  zv = ht->arPacked + h; // 顺序数组zval列表中，找到指定元素  if (Z\_TYPE\_P(zv) != IS\_UNDEF) { // 情况1：此位置有有效元素（不在新位置添加）  if (flag & HASH\_LOOKUP) { // 情况1.1: 如果是查找操作 HASH\_LOOKUP  return zv; // 调用zend\_hash\_index\_lookup()函数时，返回元素  }  replace: // 覆盖操作跳转点  if (flag & HASH\_ADD) { // 情况1.2: 如果是添加操作 HASH\_ADD  return NULL; // 顺序数组不允许在同一个位置添加两个元素，返回null（添加失败）  }  if (ht->pDestructor) { // 情况1.3: 不是 HASH\_LOOKUP 和 HASH\_ADD，只能是 HASH\_UPDATE  ht->pDestructor(zv); // 销毁此元素上的变量  }  ZVAL\_COPY\_VALUE(zv, pData); // 把新元素添加到指定位置，并增加引用次数  return zv; // 返回新元素  } else { // 情况2：新增元素，为了保证元素顺序，要把顺序数组转成哈希表再进行操作  goto convert\_to\_hash; // 跳转点：转成哈希表  }  ... // 转成哈希表后的操作 |

如上所示，**如果在目标位置找到有效元素**，调用zend\_hash\_index\_add()函数与zend\_hash\_index\_add\_new()函数时（都有HASH\_ADD标记），都会直接返回null，不进行更新操作，这两个函数在处理顺序数组时逻辑相同，在处理哈希表时逻辑会有不同；调用zend\_hash\_index\_update()函数时（有HASH\_LOOKUP标记），会尝试调用ht->pDestructor()函数，销毁元素（zval实例）指向的对象，再调用ZVAL\_COPY\_VALUE()宏程序把新元素复制到指定位置，并增加引用次数。

**如果目标位置的元素是无效的（类型为IS\_UNDEF）**，3个函数的处理逻辑相同，为了保证元素插入顺序，会把顺序数组转成哈希表进行操作。具体业务逻辑会在“哈希表”章节中介绍。

关于ZVAL\_COPY\_VALUE()宏程的更多介绍参见“类型篇”。

## 四）顺序数组中查找元素

在顺序数组中直接使用ht->arPacked[idx]语句即可访问指定序号的元素。也有少数情况会用到zend\_hash\_index\_lookup()函数：

|  |
| --- |
| ZEND\_API zval\* zend\_hash\_index\_lookup(HashTable \*ht, zend\_ulong h) {  return \_zend\_hash\_index\_add\_or\_update\_i(ht, h, NULL, HASH\_LOOKUP);  } |

同样是调用\_zend\_hash\_index\_add\_or\_update\_i()函数进行操作，大部分业务逻辑与前文所述相同，在查找元素时，会进行一些没有必要的操作，导致查询很低效，例如：

1、在大于等于ht->nNumUsed的位置查找元素时会给数组增加空间再进行查找，或者把顺序数组转成哈希表再进行查找；

2、在小于ht->nNumUsed的位置查找元素，找到的位置元素无效时，把顺序数组转成哈希表进行查找。

3、如果数组没有初始化过，先进行初始化。

所以在内核中很少用到zend\_hash\_index\_lookup()函数。

为了提升效率，内核中还提供了一个ZEND\_HASH\_INDEX\_LOOKUP()宏程序：

|  |
| --- |
| #define ZEND\_HASH\_INDEX\_LOOKUP(\_ht, \_h, \_ret) do { \  if (EXPECTED(HT\_FLAGS(\_ht) & HASH\_FLAG\_PACKED)) { /\* 只针对顺序数组 \*/ \  /\* 位置在有效范围内 \*/ \  if (EXPECTED((zend\_ulong)(\_h) < (zend\_ulong)(\_ht)->nNumUsed)) { \  \_ret = &\_ht->arPacked[\_h]; /\* 目标元素 \*/ \  if (EXPECTED(Z\_TYPE\_P(\_ret) != IS\_UNDEF)) { \  break; /\*元素有效可直接返回\*/ \  } \  } \  } \  /\* 哈希表，或顺序数组查找位置在有效元素外，或找到的元素无效 \*/ \  \_ret = zend\_hash\_index\_lookup(\_ht, \_h); \  } while (0) |

这个宏程序也仅有少量调用。

## 五）顺序数组中删除元素

顺序数组中删除元素使用zend\_hash\_packed\_del\_val()函数：

|  |
| --- |
| // 从顺序数组中删除指定元素  ZEND\_API void zend\_hash\_packed\_del\_val(HashTable \*ht, zval \*zv) {  // 除了哈希表指针和元素指针，还需要传入元素顺序号  \_zend\_hash\_packed\_del\_val(ht, HT\_IDX\_TO\_HASH(zv - ht->arPacked), zv);  } |

\_zend\_hash\_packed\_del\_val()函数业务逻辑如下：

|  |
| --- |
| // 顺序数组中删除元素。p1:哈希表，p2:顺序号，p3:元素指针  static void \_zend\_hash\_packed\_del\_val(HashTable \*ht, uint32\_t idx, zval \*zv) {  idx = HT\_HASH\_TO\_IDX(idx); // 64位系统中 index 不用转换  ht->nNumOfElements--; // 先减元素数量  // 第一步：修正内部指针和迭代器。如果正在删除内部指针指向的元素，或哈希表有迭代器  if (ht->nInternalPointer == idx || UNEXPECTED(HT\_HAS\_ITERATORS(ht))) {  uint32\_t new\_idx; // 临时顺序号  new\_idx = idx; // 从原顺序号开始  while (1) { // 从 idx 开始往后遍历，找后面的有效元素  new\_idx++; // 指针后移  if (new\_idx >= ht->nNumUsed) { // 到达最后一个有效元素  break; // 查找失败，跳出  } else if (Z\_TYPE(ht->arPacked[new\_idx]) != IS\_UNDEF) { // 找到有效元素  break; // 查找成功，跳出  }  }  if (ht->nInternalPointer == idx) { // 如果正在删除内部指针指向的元素  // 更新内部指针，new\_idx可能是无效值，会在后面再进行校正  ht->nInternalPointer = new\_idx;  }  zend\_hash\_iterators\_update(ht, idx, new\_idx); // 更新全部迭代器  }    // 第二步：更新已使用元素数量nNumUsed。如果正在删除最后一个有效元素  if (ht->nNumUsed - 1 == idx) {  do { // 更新已使用元素数量  ht->nNumUsed--; // 数量-1  // 只要右侧还有无效元素，就一直往左边找  } while (ht->nNumUsed > 0 &&  (UNEXPECTED(Z\_TYPE(ht->arPacked[ht->nNumUsed-1]) == IS\_UNDEF)));  // 如果内部指针指向无效位置，让它指向最后一个有效元素  ht->nInternalPointer = MIN(ht->nInternalPointer, ht->nNumUsed);  }    // 第三步：删除元素  if (ht->pDestructor) { // 如果有销毁器  zval tmp;  ZVAL\_COPY\_VALUE(&tmp, zv); // 先 复制到临时变量中（并增加对象引用次数）  ZVAL\_UNDEF(zv); // 再 把目标元素设置成无效，  ht->pDestructor(&tmp); // 再 调用销毁器销毁对象  } else { // 没有销毁器  ZVAL\_UNDEF(zv); // 直接把目标元素设置成无效  }  } |

Z\_TYPE()宏程序的相关介绍参见“类型篇”。

zend\_hash\_iterators\_update()函数用于修正迭代器的位置，将在“迭代器”章节介绍。

如上所示，删除元素时需要进行3步操作。

### 第一步，需要修正内部指针和迭代器

如果正在删除内部指针指向的元素，或哈希表有迭代器。需要修正内部指针和迭代器。修正内部指针的方式是把内部指针指向当前元素的后面一个有效元素。如果正在删除的元素刚好是最后一个有效元素，后面没有有效元素，这时会先把内部指针更新成一个无效值，在第二步操作时再进行修正。

### 第二步，更新已使用元素数量

如果正在删除最后一个有效元素，需要更新已使用元素数量：从当前位置一直向左查找有效元素，一边查找一边减少有效元素数量（ht->nNumUsed）。更新后，需要修正内部指针位置，如果它指向了无效位置，让它指向最后一个有效元素，没有有效元素时指向第一个元素。

### 第三步，删除元素

如果有销毁器，先把当前元素复制到临时zval实例中，再调用销毁器来销毁临时zval实例，这是因为销毁器可能尝试销毁zval实例，但数组中的元素列表由数组统一管理，不可以单独进行销毁。在删除操作中只把元素标记为无效，等待复用，这样可以少量内存操作次数，提升性能。

## 六）遍历顺序数组

遍历顺序数组通过宏程序来完成。主要有正序、倒序两种遍历方式。

### 正序遍历顺序数组

正序遍历顺序数组主要用到ZEND\_HASH\_PACKED\_FOREACH\_FROM()宏程序，逻辑比较简单：

|  |
| --- |
| // 从序号\_from开始遍历顺序数组，p1:数组指针，p2:遍历起始位置  #define ZEND\_HASH\_PACKED\_FOREACH\_FROM(\_ht, \_from) do { \  HashTable \*\_\_ht = (\_ht); /\* 哈希表指针 \*/ \  zend\_ulong \_idx = (\_from); /\* 开始序号 \*/ \  zval \*\_z = \_\_ht->arPacked + (\_from); /\* 开始位置 \*/ \  zval \*\_end = \_\_ht->arPacked + \_\_ht->nNumUsed; /\* 结束位置，最后一个有效元素位置 \*/ \  for (;\_z != \_end; \_z++, \_idx++) { /\* 向后依次遍历每个元素 \*/ \  (void) \_idx; \  /\* 跳过无效元素 \*/ \  if (UNEXPECTED(Z\_TYPE\_P(\_z) == IS\_UNDEF)) continue;  // 遍历结束  #define ZEND\_HASH\_FOREACH\_END() \  } \  } while (0) |

如上所示，ZEND\_HASH\_PACKED\_FOREACH\_FROM()宏程序是遍历过程的前半段业务逻辑，它实现了遍历顺序数组元素列表中的每个元素，调用这个宏程序后，即可添加需要的业务逻辑，最后调用ZEND\_HASH\_FOREACH\_END()宏程序结束遍历。但在实际使用中，并不直接调用ZEND\_HASH\_PACKED\_FOREACH\_FROM()宏程序，而是调用它的几个衍生宏程序：

|  |
| --- |
| // 从头开始遍历顺序数组  #define ZEND\_HASH\_PACKED\_FOREACH(\_ht) ZEND\_HASH\_PACKED\_FOREACH\_FROM(\_ht, 0)  // 从头开始遍历顺序数组，并通过\_val和\_h变量访问每个元素和元素顺序号  #define ZEND\_HASH\_PACKED\_FOREACH\_KEY\_VAL(ht, \_h, \_val) \  ZEND\_HASH\_PACKED\_FOREACH(ht); \  \_h = \_idx; /\* 供外部访问 \*/ \  \_val = \_z;  // 从头开始遍历顺序数组，并通过\_h和\_ptr变量访问每个元素顺序号和元素中存放的指针  #define ZEND\_HASH\_PACKED\_FOREACH\_KEY\_PTR(ht, \_h, \_ptr) \  ZEND\_HASH\_PACKED\_FOREACH(ht); \  \_h = \_idx; /\* 供外部访问 \*/ \  \_ptr = Z\_PTR\_P(\_z);  // 从头开始遍历顺序数组，并通过\_val变量访问每个元素  #define ZEND\_HASH\_PACKED\_FOREACH\_VAL(ht, \_val) /\* 逻辑类似 \*/  // 从头开始遍历顺序数组，并通过\_ptr变量访问每个元素中存放的指针  #define ZEND\_HASH\_PACKED\_FOREACH\_PTR(ht, \_ptr) /\* 逻辑类似 \*/  // 从头开始遍历顺序数组，并通过\_h变量访问每个元素的顺序号  #define ZEND\_HASH\_PACKED\_FOREACH\_KEY(ht, \_h) /\* 逻辑类似 \*/ |

Z\_PTR\_P()宏程序的相关介绍参见“类型篇”。

### 倒序遍历顺序数组

倒序遍历顺序数组与正序遍历相似，主要用到ZEND\_HASH\_PACKED\_REVERSE\_FOREACH()宏程序：

|  |
| --- |
| // 从结尾开始遍历顺序数组  #define ZEND\_HASH\_PACKED\_REVERSE\_FOREACH(\_ht) do { \  HashTable \*\_\_ht = (\_ht); /\* 哈希表指针 \*/ \  zend\_ulong \_idx = \_\_ht->nNumUsed; /\* 结束位置，最后一个有效元素位置 \*/ \  zval \*\_z = \_\_ht->arPacked + \_idx; /\* 开始位置 \*/ \  while (\_idx > 0) { /\* 倒序遍历每个元素 \*/ \  \_z--; /\* 元素指针左移 \*/ \  \_idx--; /\* 序号-1 \*/ \  (void) \_idx; \  /\* 跳过无效元素 \*/ \  if (UNEXPECTED(Z\_TYPE\_P(\_z) == IS\_UNDEF)) continue; |

它有几个衍生宏程序：

|  |
| --- |
| // 从结尾始倒序遍历顺序数组，并通过\_val和\_h变量访问每个元素和元素顺序号  #define ZEND\_HASH\_PACKED\_REVERSE\_FOREACH\_KEY\_VAL(ht, \_h, \_val) \  ZEND\_HASH\_PACKED\_REVERSE\_FOREACH(ht); \  \_h = \_idx; \  \_val = \_z;  // 从结尾始倒序遍历顺序数组，并通过\_h和ptr变量访问每个元素顺序号和元素中存放的指针  #define ZEND\_HASH\_PACKED\_REVERSE\_FOREACH\_KEY\_PTR(ht, \_h, \_ptr) \  ZEND\_HASH\_PACKED\_REVERSE\_FOREACH(ht); \  \_h = \_idx; \  \_ptr = Z\_PTR\_P(\_z);  // 从结尾始倒序遍历顺序数组，并通过\_val变量访问每个元素  #define ZEND\_HASH\_PACKED\_REVERSE\_FOREACH\_VAL(ht, \_val) /\* 逻辑类似 \*/  // 从结尾始倒序遍历顺序数组，并通过\_ptr变量访问每个元素中存放的指针  #define ZEND\_HASH\_PACKED\_REVERSE\_FOREACH\_PTR(ht, \_ptr) /\* 逻辑类似 \*/  // 从结尾始倒序遍历顺序数组，并通过\_h变量访问每个元素的顺序号  #define ZEND\_HASH\_PACKED\_REVERSE\_FOREACH\_KEY(ht, \_h) /\* 逻辑类似 \*/ |

倒序遍历的结尾也需要调用ZEND\_HASH\_FOREACH\_END()宏程序结束遍历。正序和倒序遍历顺序数组是非常常用的操作，在很多模块中都频繁调用到。

## 七）批量添加元素

有时需要向顺序数组中批量添加元素，多次调用zend\_hash\_next\_index\_insert\_new()函数效率比较低，PHP官方提供了一组宏程序用于快速执行添加操作。

### 第一步，调用ZEND\_HASH\_FILL\_PACKED()宏程序

ZEND\_HASH\_FILL\_PACKED()宏程序用于进行批量添加前的准备工作：

|  |
| --- |
| #define ZEND\_HASH\_FILL\_PACKED(ht) do { /\* 准备批量添加元素 \*/ \  HashTable \*\_\_fill\_ht = (ht); /\* 哈希表 \*/ \  zval \*\_\_fill\_val = \_\_fill\_ht->arPacked + \_\_fill\_ht->nNumUsed; /\* 定位到第一个空闲元素 \*/ \  uint32\_t \_\_fill\_idx = \_\_fill\_ht->nNumUsed; /\* 序号为现有元素数 \*/ \ |

如上所示，它会把\_\_fill\_val指针变量定位到第一个空闲元素，把变量序号放在\_\_fill\_idx变量中。

### 第二步，复制zval实例

在添加元素前，如果无法确定空间是否足够，可以调用ZEND\_HASH\_FILL\_GROW()宏程序进行检查，如果空间不够，它会调用zend\_hash\_packed\_grow()函数分配新空间：

|  |
| --- |
| #define ZEND\_HASH\_FILL\_GROW() do { \  if (UNEXPECTED(\_\_fill\_idx >= \_\_fill\_ht->nTableSize)) { /\* 如果空间已经满了 \*/ \  \_\_fill\_ht->nNumUsed = \_\_fill\_idx; /\* 更新已使用数量 \*/ \  \_\_fill\_ht->nNumOfElements = \_\_fill\_idx; /\* 更新有效元素数 \*/ \  \_\_fill\_ht->nNextFreeElement = \_\_fill\_idx; /\* 更新下一个空闲位置 \*/ \  zend\_hash\_packed\_grow(\_\_fill\_ht); /\* 增加空间 \*/ \  \_\_fill\_val = \_\_fill\_ht->arPacked + \_\_fill\_idx; /\* 定位到可用元素 \*/ \  } \  } while (0); |

如上所示，在空间够用的情况下，ZEND\_HASH\_FILL\_GROW()宏程序只进行一次if判断，调用多次也不会造成大的性能损耗。

如果已经知道要添加的元素的类型，可以使用下列快捷宏程序进行添加：

|  |
| --- |
| // 添加元素，值为null  #define ZEND\_HASH\_FILL\_SET\_NULL() ZVAL\_NULL(\_\_fill\_val)  // 添加元素，值为整数  #define ZEND\_HASH\_FILL\_SET\_LONG(\_val) ZVAL\_LONG(\_\_fill\_val, \_val)  // 添加元素，值为小数  #define ZEND\_HASH\_FILL\_SET\_DOUBLE(\_val) ZVAL\_DOUBLE(\_\_fill\_val, \_val)  // 添加元素，值为字串  #define ZEND\_HASH\_FILL\_SET\_STR(\_val) ZVAL\_STR(\_\_fill\_val, \_val)  // 添加元素，值为字串副本  #define ZEND\_HASH\_FILL\_SET\_STR\_COPY(\_val) ZVAL\_STR\_COPY(\_\_fill\_val, \_val)  // 添加元素，值为保留字串  #define ZEND\_HASH\_FILL\_SET\_INTERNED\_STR(\_val) ZVAL\_INTERNED\_STR(\_\_fill\_val, \_val) |

ZVAL\_NULL()等宏程序的介绍参见“类型篇”。

如果忽略类型，直接复制zval实例，可调用ZEND\_HASH\_FILL\_ADD()或ZEND\_HASH\_FILL\_SET()宏程序：

|  |
| --- |
| // 复制zval实例，并增加引用次数  #define ZEND\_HASH\_FILL\_SET(\_val) ZVAL\_COPY\_VALUE(\_\_fill\_val, \_val)  // 复制zval实例，并切换到下一个元素  #define ZEND\_HASH\_FILL\_ADD(\_val) do { \  ZEND\_HASH\_FILL\_SET(\_val); /\* 复制zval实例，并增加引用次数 \*/ \  ZEND\_HASH\_FILL\_NEXT(); /\* 切换到下一个位置 \*/ \  } while (0) |

如果调用ZEND\_HASH\_FILL\_ADD()宏程序，就不需要再调用ZEND\_HASH\_FILL\_NEXT()宏程序了。

### 第三步，切换到下一个元素

复制元素数据后，需要调用ZEND\_HASH\_FILL\_NEXT()宏程序切换到下一个元素：

|  |
| --- |
| /#define ZEND\_HASH\_FILL\_NEXT() do {\  \_\_fill\_val++; /\* 下一个zval \*/ \  \_\_fill\_idx++; /\* 下一个编号 \*/ \  } while (0) |

### 第四步，结束填充

填加完所有元素后，需要调用ZEND\_HASH\_FILL\_END()宏程序来结束填充：

|  |
| --- |
| #define ZEND\_HASH\_FILL\_END() \  ZEND\_HASH\_FILL\_FINISH(); \  } while (0)  #define ZEND\_HASH\_FILL\_FINISH() do { \  \_\_fill\_ht->nNumUsed = \_\_fill\_idx; /\* 更新已使用数量 \*/ \  \_\_fill\_ht->nNumOfElements = \_\_fill\_idx; /\* 更新有效元素数 \*/ \  \_\_fill\_ht->nNextFreeElement = \_\_fill\_idx; /\* 更新下一个空闲位置 \*/ \  \_\_fill\_ht->nInternalPointer = 0; /\* 内部指针复位 \*/ \  } while (0) |

给顺序数组快速添加元素的完整示例如下：

|  |
| --- |
| ZEND\_HASH\_FILL\_PACKED(ht) { // 准备向顺序数组ht中添加元素  ZEND\_HASH\_FILL\_GROW(); // 检查元素数量是否足够  ZEND\_HASH\_FILL\_SET\_NULL(); // 添加null  ZEND\_HASH\_FILL\_NEXT(); // 下一个元素  ZEND\_HASH\_FILL\_GROW(); // 检查元素数量是否足够  ZEND\_HASH\_FILL\_SET\_LONG(lz); // 添加整数  ZEND\_HASH\_FILL\_NEXT(); // 下一个元素  ZEND\_HASH\_FILL\_GROW(); // 检查元素数量是否足够  ZEND\_HASH\_FILL\_ADD(zv); // 复制zval并切换到下一个zval  ...  } ZEND\_HASH\_FILL\_END(); // 添加完毕 |

# 三、哈希表

## 一）基本概念

### Bucket结构体

哈希表与顺序数组在数据结构上的不同之处在于，它的元素不是zval实例，而是键值对（Bucket实例），Bucket结构体定义如下：

|  |
| --- |
| typedef struct \_Bucket { // 64位操作系统中占32Bytes  zval val; // 值，占16Bytes  zend\_ulong h; // 哈希值或索引号，占8Bytes  zend\_string \*key; // 键名，字符串指针，64位操作系统中占8Bytes  } Bucket; |

如上所示，Bucket结构体有3个元素：

1）val元素是zval实例，用来存放元素值；

2）h元素是无符号整数，有键名时用来存放哈希值，没有键名时存放索引号；

3）key元素是一个指针，指向存放键名（key）的zend\_string实例；当key元素值为NULL时h元素用来存放索引号。

### 哈希函数

为键名计算哈希值的函数是zend\_string\_hash\_func()，它在zend\_string.h文件中定义。这个函数使用了DJBX33A算法（由Daniel J. Bernstein发明的，乘以33并相加的算法），这个算法是已知的最好的哈希算法之一，因为它运算非常快，并且哈希值分散均匀。代码注释如下：

|  |
| --- |
| static zend\_ulong zend\_string\_hash\_func(const char \*str, size\_t len) {  zend\_ulong hash = Z\_UL(5381); // 初始哈希值 5381  // 遍历字符串，长度每次-8，指针每次向右跳8Bytes，直到最后不到8Bytes为止  for (; len >= 8; len -= 8, str += 8) {  hash = hash \* Z\_L(33 \* 33 \* 33 \* 33) + // 哈希值 \* pow(33,4)  str[0] \* Z\_L(33 \* 33 \* 33) + // 第1个Byte \* pow(33,3)  str[1] \* Z\_L(33 \* 33) + // 第2个Byte \* pow(33,2)  str[2] \* Z\_L(33) + // 第3个Byte \*33  str[3]; // 第4个Byte  hash = hash \* Z\_L(33 \* 33 \* 33 \* 33) + // 哈希值 \* pow(33,4)  str[4] \* Z\_L(33 \* 33 \* 33) + // 第5个Byte \* pow(33,3)  str[5] \* Z\_L(33 \* 33) + // 第6个Byte \* pow(33,2)  str[6] \* Z\_L(33) + // 第7个Byte \* 33  str[7]; // 第8个Byte  }  if (len >= 4) { // 如果余下的大于等于4Bytes  hash = hash \* Z\_L(33 \* 33 \* 33 \* 33) + // 哈希值 \* pow(33,4)  str[0] \* Z\_L(33 \* 33 \* 33) + // 第1个Byte \* pow(33,3)  str[1] \* Z\_L(33 \* 33) + // 第2个Byte \* pow(33,2)  str[2] \* Z\_L(33) + // 第3个Byte \*33  str[3]; // 第4个Byte  len -= 4; // 长度-4  str += 4; // 指针向右跳4  }  if (len >= 2) { // 如果余下的大于等于2Bytes  if (len > 2) { // 如果余下的是3Bytes  hash = hash \* Z\_L(33 \* 33 \* 33) + // 哈希值 \* pow(33,3)  str[0] \* Z\_L(33 \* 33) + // 第1个Byte \* pow(33,2)  str[1] \* Z\_L(33) + // 第2个Byte \* 33  str[2]; // 第3个Byte  } else { // 余下的是2Bytes  hash = hash \* Z\_L(33 \* 33) + // 哈希值 \* pow(33,2)  str[0] \* Z\_L(33) + // 第1个Byte \* 33  str[1]; // 第2个Byte  }  } else if (len != 0) { // 如果余下的等于1Byte  hash = hash \* Z\_L(33) + \*str; // 加最后一个Byte  }  return hash | Z\_UL(0x8000000000000000); // 第一个位写成1  }  # define Z\_UL(i) UINT64\_C(i) // 创建64位无符号整数  # define Z\_L(i) INT64\_C(i) // 创建64位整数 |

### 数据块结构

哈希表的数据块比顺序数组复杂一些，结构示意图如下：

|  |  |  |
| --- | --- | --- |
| 哈  希  索  引  表 | 索引值0 | -1 |
| ... | -1 |
| 索引值4 | 6 |
| ... | -1 |
| 索引值15 | -1 |
| arData -->  Bucket  元  素  列  表 | Bucket 0 | --- |
| Bucket 1 | --- |
| Bucket 2 | key:\*\*\*  val.u2.next:2 |
| Bucket 3 | --- |
| Bucket 4 | h:\*\*\*  val.u2.next:2 |
| Bucket 5 | --- |
| Bucket 6 | key:\*\*\*  val.u2.next:4 |
| Bucket 7 | --- |

如上图所示，以一个最小的哈希表（有8个元素）为例。数据块分成前后两部分，前面是哈希索引表，共有16个元素，元素类型是32位整数；后面是Bucket列表，共有8个元素。arData指针（同时也是arHash指针）指向Bucket列表的第一个元素。

哈希表中有3个元素，它们的哈希索引号相同，都是4。所以在查找时，都会先找到哈希索引表中的第4个元素，元素中记录的序号是6。它对应串联好的3个元素：Bucket6->Bucket4->Bucket2，依次比对它们的键名和索引号，即可找到匹配的元素。

在插入元素时，Bucket元素列表是从上到下依次使用，先插入的元素在上面，后插入的元素在下面，这样保证了插入顺序和读取顺序相同；但在相同哈希索引号的元素链表中，后添加的元素在链表前面，先添加的元素在链表后面。**链表元素的序号一定是从大到小排列，顺号最大的在链表开头位置**。每次添加新元素，哈希索引列表中的数字都会被更新成最后添加的元素序号。这样可以保证添加操作在O(1)的时间复杂度内完成。

对于哈希表中元素的插入、查找和删除等操作，下面将会展开详细介绍。

### 常用常量和宏程序

一些常用的宏程序列举如下：

|  |
| --- |
| // 找到哈希表中第 idx 个bucket  #define HT\_HASH\_TO\_BUCKET(ht, idx) HT\_HASH\_TO\_BUCKET\_EX((ht)->arData, idx)  // 通过偏移量找到Bucket  # define HT\_HASH\_TO\_BUCKET\_EX(data, idx) (data) + (idx) |

下面开始介绍哈希表各种操作的具体实现方式，哈希表在功能上要比顺序数组复杂很多，先从最简单的初始化开始。

## 二）初始化哈希表

创建数组（zend\_array）实例的过程已经在第一章中介绍过，创建后需要把数组初始化成哈希表。初始化哈希表使用zend\_hash\_real\_init\_mixed()函数，它调用zend\_hash\_real\_init\_mixed\_ex()函数进行操作：

|  |
| --- |
| static void zend\_hash\_real\_init\_mixed\_ex(HashTable \*ht) {  void \*data;  uint32\_t nSize = ht->nTableSize; // 元素数量，创建zend\_array实例时设置过  if (UNEXPECTED(GC\_FLAGS(ht) & IS\_ARRAY\_PERSISTENT)) { // 使用malloc()函数分配内存  // 计算 哈希表 数据大小，并分配内存创建数据块  data = pemalloc(HT\_SIZE\_EX(nSize, HT\_SIZE\_TO\_MASK(nSize)), 1);  } else if (EXPECTED(nSize == HT\_MIN\_SIZE)) { // 如果元素个数是HT\_MIN\_SIZE=8  // 计算 哈希表 数据大小，并分配内存创建数据块  data = emalloc(HT\_SIZE\_EX(HT\_MIN\_SIZE, HT\_SIZE\_TO\_MASK(HT\_MIN\_SIZE)));  ht->nTableMask = HT\_SIZE\_TO\_MASK(HT\_MIN\_SIZE); // 更新掩码  HT\_SET\_DATA\_ADDR(ht, data); // 设置 p1->arData 到 Bucket 列表开头  // 一开始只有静态键名，未更新ht->u.v.nIteratorsCount  ht->u.v.flags = HASH\_FLAG\_STATIC\_KEYS;  HT\_HASH\_EX(data, 0) = -1; // 初始化16个哈希值  ... // 顺序号1-14更新成-1  HT\_HASH\_EX(data, 15) = -1;  return; // 返回，不调用 HT\_HASH\_RESET  } else { // 元素数量不是8个  // 计算 哈希表 数据大小，并分配内存创建数据块  data = emalloc(HT\_SIZE\_EX(nSize, HT\_SIZE\_TO\_MASK(nSize)));  }  ht->nTableMask = HT\_SIZE\_TO\_MASK(nSize); // 计算并更新掩码  HT\_SET\_DATA\_ADDR(ht, data); // 设置 p1->arData 到 Bucket 列表开头  // (ht)->u.flags 一开始只有静态键名，更新ht->u.v.nIteratorsCount=0  HT\_FLAGS(ht) = HASH\_FLAG\_STATIC\_KEYS;  HT\_HASH\_RESET(ht); // 初始化哈希索引列表，把所有哈希值写成-1  } |

初始化过程分为4步。

### 第一步，计算数据块大小，并分配内存创建数据块

#### **掩码（nTableMask）**

**哈希表中有一个非常关键的元素是掩码（nTableMask）**。掩码是利用元素数量，通过HT\_SIZE\_TO\_MASK()宏程序计算获得：

|  |
| --- |
| // 计算掩码：元素数 x -2。元素数是2的幂，mask是右侧连续的n个0，左侧全部是1  // 例如 4 => -8 => 11111111 11111111 1111111 11111000  // 例如 16 => -32 => 11111111 11111111 1111111 11100000  #define HT\_SIZE\_TO\_MASK(nTableSize) ((uint32\_t)(-((nTableSize) + (nTableSize)))) |

如上所示，掩码的计算方法是nTableSize\*-2，它一定是负数。掩码的用途有以下两个：

a）在创建数据块时，使用掩码计算需要的哈希索引表元素数量，哈希索引表元素数量是掩码的相反数（-nTableSize）；

b）使用哈希值和掩码换算出哈希索引表中的元素的顺序号，例如：

|  |
| --- |
| // 或运算不可能得到0，也不可能得到比ht->nTableMask小的负数  // 0 > nIndex >= ht->nTableMask  uint32\_t nIndex = p->h | ht->nTableMask; |

如上所示，这一步操作非常关键，在代码中很常见。得到的顺序号nIndex一定是nTableMask和0之间的负数，这样通过ht->arHash指针进行偏移，即可找到哈希索引表中的元素。哈希索引表用来存放哈希值对应的Bucket元素序号。进行查找操作时，要先从哈希索引表中取得元素序号，再检索元素链表；进行添加操作时，要把新Bucket元素的序号更新到哈希索引表中。例如：

|  |
| --- |
| idx = HT\_HASH(ht, nIndex); // 从哈希索引表中取出Bucket元素序号  HT\_HASH(ht, nIndex) = idx; // 把Bucket元素序号写进哈希索引表 |

#### **计算数据块大小**

计算数据块大小用到HT\_SIZE\_EX()宏程序：

|  |
| --- |
| // 计算 哈希表 数据块大小：Bucket 列表大小 + 哈希索引列表大小  #define HT\_SIZE\_EX(nTableSize, nTableMask) \  (HT\_DATA\_SIZE((nTableSize)) + HT\_HASH\_SIZE((nTableMask)))  // 哈希表 Bucket 列表 大小： 哈希表大小 \* Bucket大小  #define HT\_DATA\_SIZE(nTableSize) ((size\_t)(nTableSize) \* sizeof(Bucket))    // 把掩码换算成哈希索引列表大小： -nTableMask \* 4 = 元素数 \* 8  #define HT\_HASH\_SIZE(nTableMask) (((size\_t)-(uint32\_t)(nTableMask)) \* sizeof(uint32\_t)) |

如上所示，元素列表的大小 = 元素列表的大小 + 哈希索引列表大小。在哈希表中，调用HT\_DATA\_SIZE()宏程序来计算元素列表大小，每个元素是一个Bucket实例，元素列表大小 = 元素数量 \* Bucket实例大小。哈希索引列表的大小通过HT\_HASH\_SIZE()宏程序来计算，一般为哈希表元素数量＊8（-nTableMask值为元素数的2倍，sizeof(uint32\_t)值为4）。

由以上算法计算出的哈希索引列表中元素数是哈希表元素数的2倍，实际上哈希索引值的数量不可能大于哈希表元素数数量，所以哈希索引表中至少有一半空间是闲置的；这样虽然浪费了一定的空间，但可以减少哈希值的碰撞机率，提升查找效率。

计算大小后使用pemalloc()函数或emalloc()函数分配内存创建元素列表，这两个函数的相关介绍参见“内存管理篇”。

### 第二步，更新元素列表指针。

调用HT\_SET\_DATA\_ADDR()宏程序，让哈希表的arData指针元素指向元素列表的开头。这一步操作与顺序数组相同。

### 第三步，添加标记

这里用到HASH\_FLAG\_STATIC\_KEYS标记，表示哈希表中所有键名都是静态的，不会变化。在哈希表中，如果有普通的字符串（zend\_string实例）作为键名，zend\_string实例可能会变化，这时就会删除HASH\_FLAG\_STATIC\_KEYS标记。如果所有元素（Bucket实例）都使用数字哈希值或使用保留字串作为键名，HASH\_FLAG\_STATIC\_KEYS标记就会一直存在。

关于“保留字串”的介绍可参见“类型篇”。

### 第四步，把哈希索引列表里的所有元素都设置为-1

在zend\_hash\_real\_init\_mixed\_ex()函数中，如果哈希表的元素数是8个，会使用HT\_HASH\_EX(data, 0) = -1;语句分别更新16个哈希值。如果哈希表的元素数量不是8个，会使用HT\_HASH\_RESET()宏程序批量更新整个哈希索引列表：

|  |
| --- |
| // 通过数据块指针，访问哈希索引列表中指定序号的元素  #define HT\_HASH\_EX(data, idx) ((uint32\_t\*)(data))[(int32\_t)(idx)]  // 把整个哈希索引列表 都设置成 HT\_INVALID\_IDX = -1  # define HT\_HASH\_RESET(ht) \  /\* memset 的 p3必须是 字节数，所以要计算 哈希索引列表的大小 \*/ \  memset(&HT\_HASH(ht, (ht)->nTableMask), HT\_INVALID\_IDX,  HT\_HASH\_SIZE((ht)->nTableMask))  // 访问哈希索引列表中指定序号的元素（uint32\_t类型）  #define HT\_HASH(ht, idx) HT\_HASH\_EX((ht)->arHash, idx) |

## 三）哈希表中添加和更新元素

向哈希表中添加和更新元素主要使用\_zend\_hash\_add\_or\_update\_i()函数和\_zend\_hash\_str\_add\_or\_update\_i()函数，这两个函数的业务逻辑类似。

先从\_zend\_hash\_add\_or\_update\_i()函数开始介绍，此函数逻辑比较复杂，处理了多种情况。它的业务逻辑可以分成前后两部分，前面一半主要用来作添加元素前的处理，后面一半是在哈希表中添加新元素。

### 添加元素前的处理

在\_zend\_hash\_add\_or\_update\_i()函数的开头，会先调用zend\_string\_hash\_val()函数为传入的key更新哈希值：

|  |
| --- |
| static zval \*\_zend\_hash\_add\_or\_update\_i(HashTable \*ht, zend\_string \*key, zval \*pData, uint32\_t flag) {  zend\_ulong h; // 哈希值  uint32\_t nIndex;  uint32\_t idx;  Bucket \*p, \*arData;  zend\_string\_hash\_val(key); // 给key更新哈希值 |

zend\_string\_hash\_val()函数在zend\_string.h文件中定义，用于为zend\_string实例计算哈希值：

|  |
| --- |
| static zend\_ulong zend\_string\_hash\_val(zend\_string \*s) {  return ZSTR\_H(s) ? ZSTR\_H(s) : zend\_string\_hash\_func(s);  }  #define ZSTR\_H(zstr) (zstr)->h // 取回 zend\_string 的哈希值（整数） |

zend\_string类型和ZSTR\_H()宏程序的更多介绍参见“类型篇”。

添加元素前的处理可分为3种情况。

#### **情况一，数组是顺序数组，或者未初始化**

相关业务逻辑如下：

|  |
| --- |
| // 紧接\_zend\_hash\_add\_or\_update\_i()函数的前一段代码  // 情况1：如果数组未初始化，或者是 顺序数组  if (UNEXPECTED(HT\_FLAGS(ht) & (HASH\_FLAG\_UNINITIALIZED|HASH\_FLAG\_PACKED))) {  // 情况1.1：如果数组未初始化  if (EXPECTED(HT\_FLAGS(ht) & HASH\_FLAG\_UNINITIALIZED)) {  zend\_hash\_real\_init\_mixed(ht); // 初始化哈希表  goto add\_to\_hash; // 跳转：向哈希表中添加元素  } else { // 情况1.2：初始化过的顺序数组  zend\_hash\_packed\_to\_hash(ht); // 顺序数组转哈希表  } |

如上所示，如果数组未初始化，调用zend\_hash\_real\_init\_mixed()函数初始化它；否则如果是顺序数组，调用zend\_hash\_packed\_to\_hash()函数把顺序数组转成哈希表：

|  |
| --- |
| ZEND\_API void zend\_hash\_packed\_to\_hash(HashTable \*ht) {  void \*new\_data, \*old\_data = HT\_GET\_DATA\_ADDR(ht); // 数据块开头  zval \*src = ht->arPacked; // 元素列表开头  Bucket \*dst;  uint32\_t i;  uint32\_t nSize = ht->nTableSize; // 元素数量  HT\_FLAGS(ht) &= ~HASH\_FLAG\_PACKED; // 删除顺序数组标记  // 计算元素数量，并分配内存创建新数据块  new\_data = pemalloc(HT\_SIZE\_EX(nSize, HT\_SIZE\_TO\_MASK(nSize)), GC\_FLAGS(ht) & IS\_ARRAY\_PERSISTENT);  ht->nTableMask = HT\_SIZE\_TO\_MASK(ht->nTableSize); // 计算掩码  HT\_SET\_DATA\_ADDR(ht, new\_data); // 让ht->arData 指向新数据块的 Bucket 列表开头  dst = ht->arData; // 新元素列表开头  for (i = 0; i < ht->nNumUsed; i++) { // 遍历旧列表中的元素  ZVAL\_COPY\_VALUE(&dst->val, src); // 把元素依次复制到新列表的Bucket.val中  dst->h = i; // 元素顺序号作为索引号  dst->key = NULL; // 没有键名  dst++; // 下一个新元素  src++; // 下一个旧元素  }  pefree(old\_data, GC\_FLAGS(ht) & IS\_ARRAY\_PERSISTENT); // 删除旧的数据块  // 整理元素，填补中间的空位，然后给所有元素更新 编号，哈希索引表，next指针  zend\_hash\_rehash(ht);  } |

zend\_hash\_rehash()函数的介绍参见“整理哈希表”章节。

#### **情况二，数组是初始化过的哈希表，并且没有传入HASH\_ADD\_NEW标记**

这部分的处理比较复杂：

|  |
| --- |
| // 情况2：已初始化过的【哈希表】,没有 HASH\_ADD\_NEW  } else if ((flag & HASH\_ADD\_NEW) == 0 || ZEND\_DEBUG) {  p = zend\_hash\_find\_bucket(ht, key); // 获取已有键值对  if (p) { // 情况2.1：如果有旧元素  zval \*data;  if (flag & HASH\_LOOKUP) { // 情况2.1.1：如果是查找，直接返回  return &p->val;  } else if (flag & HASH\_ADD) { // 情况2.1.2：如果是添加  // 情况2.1.2.1：如果没有 HASH\_UPDATE\_INDIRECT  if (!(flag & HASH\_UPDATE\_INDIRECT)) {  return NULL; // 返回 null，操作中断  }  // 情况2.1.2.2：有 HASH\_UPDATE\_INDIRECT 和 HASH\_ADD  data = &p->val;  if (Z\_TYPE\_P(data) == IS\_INDIRECT) { // 情况2.1.2.2.1：如果新值是间接引用  data = Z\_INDIRECT\_P(data); // 追踪引用目标  // 情况2.1.2.2.1.1：如果目标元素有效  if (Z\_TYPE\_P(data) != IS\_UNDEF) { // 如果引用目标有效  return NULL; // 返回 null，操作中断  }  // 情况2.1.2.2.1.2：如果目标元素无效  } else { // 情况2.1.2.2.2： 不是间接引用  return NULL; // 返回 null，操作中断  }  } else { // 情况2.1.3：其他情况  data = &p->val; // 键值对值的指针  // 情况2.1.3.1： HASH\_UPDATE\_INDIRECT 并且是间接引用  if ((flag & HASH\_UPDATE\_INDIRECT) && Z\_TYPE\_P(data) == IS\_INDIRECT) {  data = Z\_INDIRECT\_P(data); // 追踪引用目标  }  // 情况2.1.3.2：没有 HASH\_UPDATE\_INDIRECT 不处理间接引用  }  // 情况 2.1.3的两个分支都会到这里  if (ht->pDestructor) { // 如果有销毁器  ht->pDestructor(data); // 销毁键值对的值  }  ZVAL\_COPY\_VALUE(data, pData); // 把数据复制到元素中  return data; // 返回元素值（zval实例）指针  }  // 情况2.2：没找到已有元素  }  // 情况3：已初始化过的【哈希表】，HASH\_ADD\_NEW  // 可能到这里的情况：  // 情况1：如果数组未初始化，或者是顺序数组  // 情况2.1.2.2.1.2：没有传入HASH\_ADD\_NEW标记，有传入HASH\_ADD标记和HASH\_UPDATE\_INDIRECT标记，有旧元素，旧元素是间接引用类型，并且目标对象无效  // 情况2.2：已初始化过的哈希表，没有传入HASH\_ADD\_NEW标记，并且没找到已有元素  // 情况3：已初始化过的哈希表，有传入HASH\_ADD\_NEW标记  ZEND\_HASH\_IF\_FULL\_DO\_RESIZE(ht); // 如果哈希表满了，重新设置大小 |

如上所示，每一种情况的处理逻辑并不复杂，但有多个情况分支。可参见下文中的业务逻辑分支表来梳理业务逻辑。

#### **情况三，有传入HASH\_ADD\_NEW标记**

如上一段代码所示，情况三的条件没有显式声名，即前面两个if和ifelse以外的情况：有传入HASH\_ADD\_NEW标记，这种情况下不需要前期处理，直接添加元素。

ZEND\_HASH\_IF\_FULL\_DO\_RESIZE()宏程序的介绍参见“给哈希表增加空间”章节。

#### **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，业务逻辑分支表简洁直观，有助于理解复杂的业务逻辑。以上代码可以归纳成逻辑分支表如下：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **第一层条件** | **第二层** | **第三层条件** | **第四层条件** | **第五层条件** | **第六层条件** | **处理逻辑** | **分支编号** |
| 数组未初始化，或者类型是顺序数组 | 数组未初始化 | | | | | 初始化，添加元素 | 1.1 |
| 类型是顺序数组 | | | | | 顺序数组转成哈希表，添加元素 | 1.2 |
| 已初始化过的数组，并且没有传入HASH\_ADD\_NEW标记 | 位置上有旧元素 | 有传入HASH\_LOOKUP标记 | | | | 返回元素值 | 2.1.1 |
| 有传入 HASH\_ADD标记 | 没有传入HASH\_UPDATE\_INDIRECT标记 | | | 返回NULL，中断 | 2.1.2.1 |
| 有传入HASH\_UPDATE\_INDIRECT标记 | 旧元素是间接引用类型 | 目标元素有效 | 返回NULL，中断 | 2.1.2.2.1.1 |
| 目标元素无效 | 更新元素 | 2.1.2.2.1.2 |
| 旧元素不是间接引用类型 | | 返回NULL，中断 | 2.1.2.2.2 |
| 有传入  HASH\_UPDATE  标记 | 有传入HASH\_UPDATE\_INDIRECT标记  并且旧元素类型是间接引用类型 | | | 追踪引用目标，销毁引用目标并用新值替换 | 2.1.3.1 |
| 没有传入HASH\_UPDATE\_INDIRECT标记  或旧元素值不是间接引用类型 | | | 销毁旧元素值 并用新值替换 | 2.1.3.2 |
| 位置上没有旧元素 | | | | | 添加元素 | 2.2 |
| 已初始化过的哈希表，有传入HASH\_ADD\_NEW标记 | | | | | | 直接添加元素 | 3 |

如上表所示，每增加一层判断，在业务逻辑分支编号上增加一位数字。业务逻辑归纳如下：

分支1.1，数组未初始化：先初始化成哈希表，再添加元素；

分支1.2，数组类型是顺序数组：先转成哈希表，再添加元素；

分支2.1.1，没有传入HASH\_ADD\_NEW标记，有传入HASH\_LOOKUP标记，并且目标位置上已有元素：返回已有元素的值；

分支2.1.2.1，有传入HASH\_ADD标记，没有传入HASH\_ADD\_NEW标记和HASH\_UPDATE\_INDIRECT标记，并且目标位置上已有元素：返回NULL，中断；

分支2.1.2.2.1.1，没有传入HASH\_ADD\_NEW标记，有传入HASH\_ADD标记和HASH\_UPDATE\_INDIRECT标记，有旧元素，旧元素是间接引用类型，并且目标对象有效：返回NULL，中断；

分支2.1.2.2.1.2，没有传入HASH\_ADD\_NEW标记，有传入HASH\_ADD标记和HASH\_UPDATE\_INDIRECT标记，有旧元素，旧元素是间接引用类型，并且目标对象无效：更新，覆盖旧元素；

分支 2.1.2.2.2，没有传入HASH\_ADD\_NEW标记，有传入HASH\_ADD标记和HASH\_UPDATE\_INDIRECT标记，有旧元素，旧元素不是间接引用类型：返回NULL，中断；

分支 2.1.3.1，有传入HASH\_UPDATE标记和HASH\_UPDATE\_INDIRECT标记，有旧元素，并且旧元素是间接引用类型：追踪引用目标，销毁引用目标并用新值替换。

分支 2.1.3.2，有传入HASH\_UPDATE标记，没有传入HASH\_UPDATE\_INDIRECT标记，有旧元素，销毁旧元素值并用新值替。

分支 2.2，没有传入HASH\_ADD\_NEW标记，位置上没有旧元素：添加元素；

分支 3，已初始化过的哈希表，有传入HASH\_ADD\_NEW标记：添加元素。

#### **相关的衍生函数**

\_zend\_hash\_add\_or\_update\_i()函数一般不被直接调用，而是调用它的衍生函数：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **函数名** | **附加标记** | **相关分支** | **业务逻辑归纳** | **引用**  **次数** |
| zend\_hash\_add() | HASH\_ADD | 2.1.2.1，  2.2 | 有旧元素时返回NULL，否则添加新元素 | 大量 |
| zend\_hash\_update() | HASH\_UPDATE | 2.1.3.2，  2.2 | 有旧元素时覆盖旧元素，否则添加新元素 | 大量 |
| zend\_hash\_update\_ind() | HASH\_UPDATE |  HASH\_UPDATE\_INDIRECT | 2.1.3.\*，  2.2 | 有旧元素时覆盖旧元素（追踪间接引用），否则添加新元素 | 大量 |
| zend\_hash\_add\_new() | HASH\_ADD\_NEW | 3 | 直接添加新元素，不检查已有元素 | 大量 |
| zend\_hash\_lookup() | HASH\_LOOKUP | 2.1.1，  2.2 | 有旧元素时返回旧元素，否则添加NULL元素，返回NULL | 大量 |
| zend\_hash\_merge() | HASH\_UPDATE |  HASH\_UPDATE\_INDIRECT或  HASH\_ADD |  HASH\_UPDATE\_INDIRECT | 2.1.2.2.\*，  2.1.3.\*，  2.2 | 有旧元素时检查旧元素（追踪间接引用），旧元素有效则跳过，无效则覆盖掉。没有旧元素时添加新元素 | 大量 |

如上表所示，其中zend\_hash\_merge()函数用于合并数组，详细介绍参见“合并数组”章节。

需要注意的是zend\_hash\_lookup()函数，它虽然用于查找元素，但查找的元素不存在时，会新建一个值为NULL的元素，所以一般不用zend\_hash\_lookup()函数来查找元素。查找元素的函数会在后续章节中介绍。

还有一些函数是从以上列表中的函数衍生而来：

|  |  |  |
| --- | --- | --- |
| **函数名** | **说明** | **引用**  **次数** |
| zend\_hash\_add\_empty\_element() | 向数哈希表添加值为NULL的元素  调用zend\_hash\_add() | 大量 |
| zend\_hash\_add\_mem() | 给内存数据创建副本，并把副本指针添加到哈希表中  调用zend\_hash\_add() | 大量 |
| zend\_hash\_add\_new\_mem() | 给内存数据创建副本，并把副本指针直接添加到哈希表中，不检查已有元素  调用zend\_hash\_add\_new() | 少量 |
| zend\_hash\_add\_new\_ptr() | 向哈希表中直接添加指针元素，不检查已有元素  调用zend\_hash\_add\_new() | 大量 |
| zend\_hash\_add\_ptr() | 向哈希表添加指针元素  调用zend\_hash\_add() | 大量 |
| zend\_hash\_update\_ptr() | 用给定的指针覆盖哈希表中的元素  调用zend\_hash\_update()函数 | 大量 |
| zend\_hash\_update\_mem() | 给内存数据创建副本，并把副本指针覆盖到哈希表中  调用zend\_hash\_update\_ptr()函数 | 大量 |

对于zend\_hash\_add\_new()函数和它的衍生函数，要谨慎使用，因为它不对已有元素进行检查，可能导致添加相同键名（key）的元素。

### 在哈希表中添加元素

与添加元素相关的主要业务逻辑如下：

|  |
| --- |
| // 紧接前文中的代码  add\_to\_hash: // 跳转点：添加元素  // 第一步，更新标记  if (!ZSTR\_IS\_INTERNED(key)) { // 如果key不是保留字串  zend\_string\_addref(key); // 增加引用次数，自动跳过保留字串  HT\_FLAGS(ht) &= ~HASH\_FLAG\_STATIC\_KEYS; // 删除标记：静态key  }  // 第二步，更新数量  idx = ht->nNumUsed++; // 使用数+1  ht->nNumOfElements++; // 元素数+1  // 第三步，更新哈希索引列表  arData = ht->arData; // 数据表开头  p = arData + idx; // 在元素列表中找到新元素  p->key = key; // 更新新元素的key  p->h = h = ZSTR\_H(key); // 更新新元素的哈希值  nIndex = h | ht->nTableMask; // 利用哈希值 和 mask 计算索引号位置  **// 新元素的 Bucket.val.u2.next 元素中保存【哈希索引列表中的旧序号】，这样就把相同哈希值的元素串成链表了**  Z\_NEXT(p->val) = HT\_HASH\_EX(arData, nIndex);  **// 找到哈希索引号列表中的对应元素，把当新元素的序号记录进去 ，让当前元素成为链表的第一个元素**  HT\_HASH\_EX(arData, nIndex) = HT\_IDX\_TO\_HASH(idx); // 64位系统中 index 不用转换  // 第四步  if (flag & HASH\_LOOKUP) { // 如果要查询  ZVAL\_NULL(&p->val); // 元素的值写成null  } else { // 否则，添加元素  ZVAL\_COPY\_VALUE(&p->val, pData); // 把 pData 复制给 元素值  }  return &p->val; // 返回新键值对的值  } |

ZSTR\_IS\_INTERNED()宏程序、Z\_NEXT()宏程序和zend\_string\_addref()函数的介绍参见“类型篇”。

如上所示，添加操作分为几个步骤：

#### **第一步，更新哈希表的标记（flag）**

如果键名不是保留字，给键名（key）增加引用次数，并删除数组的HASH\_FLAG\_STATIC\_KEYS标记。

#### **第二步，给哈希表增加元素个数并找到需要使用的新元素**

ht->nNumUsed和ht->nNumOfElements两个元素各加1。这里获取新元素的方法是：

|  |
| --- |
| idx = ht->nNumUsed++;  p = arData + idx; |

如上所示，每次添加新元素都在已使用元素列表的结尾添加新元素，如果元素列表中间有删除过元素，这个空间不会被复用。如果要复用元素列表中间的空元素，需要满足两个条件之一：

1）调用zend\_hash\_rehash()函数（“整理哈希表”章节中介绍）整理哈希表后，会清除元素列表中间的空元素；

2）如果空元素后面的所有元素都被删除，重新计算ht->nNumUsed时，会把结尾的空元素排除在外，再添加元素时，结尾的空元素会被复用。

例如：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 有效元素1 | 无效元素 | 有效元素2 | 未使用 | 未使用 | ... |
| 位置1 | 位置2 | 位置3 | 位置4 | 位置5 | ... |

如上所示，元素列表中有2个有效元素，中间有1个无效元素，这时nNumUsed值为3（从开头到最后一个有效元素的长度），增加元素时，会使用位置4。当有效元素2被删除时，nNumUsed的值会被更新成1，这时如果增加新元素，位置2就可以被复用了。

这部分业务逻辑会在“哈希表中删除元素”章节中详细介绍。

#### **第三步，更新哈希值对应的元素链表**

这一步有两个操作：

a）在新元素的 Bucket.val.u2.next 元素中保存“哈希索引列表中的旧序号”。这样就把同样哈希值的元素串成一串，新来的在开头，旧元素排在后面。最后一个元素的next 元素值为-1。

b）把新元素的序号更新到哈希索引列表中，让哈希索引列表中的元素指向Bucket串的开头。

哈希表中最关键的步骤是通过哈希值定位到对应Bucket实例，通过哈希值定位到对应的Bucket实例需要3个步骤：

**1）用哈希值定位到哈希索引列表中的元素**。

nIndex = h | ht->nTableMask；这个语句在哈希表中使用频率很高，是哈希表的核心算法。如前文所述，掩码（ht->nTableMask）是负数，它的前n位都是1，用哈希值（h变量）和它进行或运算，会得到一个负数nIndex，大小在ht->nTableMask到0之间，不包含0。这个负数（nIndex）是一个序号，调用HT\_HASH\_EX(ht->arData, nIndex)宏程序，即可定位哈希索引列表中的元素。

**2）**哈希索引列表中的元素是32位整数，存放Bucket元素在列表中的序号，通过它，可以**定位到Bucket元素**。所以在创建新元素时，也要更新这个整数，把新元素的索引号存放进去。

**3）**由于哈希值可能冲突，多个元素的哈希值可能指向哈希索引列表中的同一个元素，这时取出的Bucket序号也是相同的。所以每个Bucket序号对应的不是一个Bucket实例，而是一串Bucket实例组成的链表。这一串实例中，每个Bucket实例的Bucket.val.u2.next元素里存放下一个Bucket实例的顺序号。所以用键名（key）或索引号（index）查找元素时，**需要遍历这个链，逐个比较键名元素中的字串是否匹配以及哈希值（也是索引号）是否匹配**，如果都匹配，这个元素才是正确的元素。

#### **第四步，更新Bucket的值**

调用用ZVAL\_COPY\_VALUE()宏程序把传入的zval实例复制到Bucket实例中，并为目标元素增加引用次数。

### \_zend\_hash\_str\_add\_or\_update\_i()函数

\_zend\_hash\_str\_add\_or\_update\_i()函数的业务逻辑与\_zend\_hash\_add\_or\_update\_i()函数一致，不同的是它接收的键名不是zend\_string实例，而是普通字符串和长度：

|  |
| --- |
| static zval \*\_zend\_hash\_str\_add\_or\_update\_i(HashTable \*ht, const char \*str, size\_t len, zend\_ulong h, zval \*pData, uint32\_t flag) |

它的衍生函数列表如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名** | **附加标记** | **业务逻辑归纳** | **引用**  **次数** |
| zend\_hash\_str\_add() | HASH\_ADD | 有旧元素返回NULL，否则添加新元素 | 大量 |
| zend\_hash\_str\_add\_new() | HASH\_ADD\_NEW | 直接添加新元素，不检查已有元素 | 大量 |
| zend\_hash\_str\_update() | HASH\_UPDATE | 有旧元素时覆盖旧元素，否则添加新元素 | 大量 |
| zend\_hash\_str\_update\_ind() | HASH\_UPDATE |  HASH\_UPDATE\_INDIRECT | 有旧元素时覆盖旧元素（追踪到间接引用目标），否则添加新元素 | 少量 |

以上几个函数的衍生函数列表如下：

|  |  |  |
| --- | --- | --- |
| **函数名** | **说明** | **引用**  **次数** |
| zend\_hash\_str\_add\_empty\_element() | 向数哈希表添加值为NULL的元素  调用zend\_hash\_str\_add()函数 | 少量 |
| zend\_hash\_str\_add\_mem() | 给内存数据创建副本，并把副本指针添加到哈希表中  调用zend\_hash\_str\_add()函数 | 大量 |
| zend\_hash\_str\_add\_new\_mem() | 给内存数据创建副本，并把副本指针直接添加到哈希表中，不检查已有元素  调用zend\_hash\_str\_add\_new()函数 | 无 |
| zend\_hash\_str\_add\_new\_ptr() | 向哈希表中直接添加指针元素，不检查已有元素  调用zend\_hash\_str\_add\_new()函数 | 少量 |
| zend\_hash\_str\_add\_ptr() | 向哈希表中添加指针元素  调用zend\_hash\_str\_add()函数 | 大量 |
| zend\_hash\_str\_update\_mem() | 给内存数据创建副本，并把副本指针更新到哈希表中  调用zend\_hash\_str\_update\_ptr()函数 | 大量 |
| zend\_hash\_str\_update\_ptr() | 把指针更新到哈希表中  调用zend\_hash\_str\_update()函数 | 大量 |

对于zend\_hash\_str\_add\_new()函数和它的衍生函数，要谨慎使用，因为它不对已有元素进行检查，可能导致添加相同键名（key）的元素。

## 四）哈希表中删除元素

### 通过Bucket指针删除元素

通过Bucket指针在哈希表中删除元素可使用zend\_hash\_del\_bucket()函数：

|  |
| --- |
| // 在哈希表中删除Bucket。p1:哈希表，p2:Bucket指针  ZEND\_API void zend\_hash\_del\_bucket(HashTable \*ht, Bucket \*p) {  // 哈希表中删除指定 序号的 bucket。p1:哈希表，p2:序号，p3:bucket指针  \_zend\_hash\_del\_el(ht, HT\_IDX\_TO\_HASH(p - ht->arData), p);  } |

如上所示，它调用\_zend\_hash\_del\_el()函数删除哈希表中指定序号的元素，函数业务逻辑如下：

|  |
| --- |
| // 哈希表中删除指定 序号的 bucket。p1:哈希表，p2:序号，p3:bucket指针  static void \_zend\_hash\_del\_el(HashTable \*ht, uint32\_t idx, Bucket \*p) {  Bucket \*prev = NULL;  uint32\_t nIndex;  uint32\_t i;  nIndex = p->h | ht->nTableMask; // 利用哈希值和掩码计算哈希索引表元素序号  i = HT\_HASH(ht, nIndex); // 从哈希索引列表中取出元素序号  // 如果元素序号和传入序号不符，说明要删除的元素不是Bucket链表的第一个  if (i != idx) {  prev = HT\_HASH\_TO\_BUCKET(ht, i); // 取出链表第一个Bucket  **// 如果Bucket不是指向要删除的元素，顺着链表一直找**  while (Z\_NEXT(prev->val) != idx) {  i = Z\_NEXT(prev->val); // 前下个Bucket的序号  prev = HT\_HASH\_TO\_BUCKET(ht, i); // 切换到下一个Bucket  }  }  if (p->key) { // 如果有键名  zend\_string\_release(p->key); // 释放zend\_string实例  p->key = NULL; // 清空指针  }  \_zend\_hash\_del\_el\_ex(ht, idx, p, prev); // 删除元素  } |

如上所示，先找到链表中的上一个元素（指向当前元素），再删除键名（key），最后调用 \_zend\_hash\_del\_el\_ex()函数删除元素，函数业务逻辑如下：

|  |
| --- |
| // 删除Bucket（key在外面处理）。p1:哈希表，p2:元素序号，p3:要删除的元素，p4:前一个元素  static void \_zend\_hash\_del\_el\_ex(HashTable \*ht, uint32\_t idx, Bucket \*p, Bucket \*prev) {  // 第一步，在Bucket链表上摘除本元素  if (prev) { // 如果有前一个,让前一个的next跳过本元素  Z\_NEXT(prev->val) = Z\_NEXT(p->val);  } else { // 没有前一个，说明本原素是链表开头  // 摘除元素链表的头元素，让哈希索引表指向第二个元素  HT\_HASH(ht, p->h | ht->nTableMask) = Z\_NEXT(p->val);  }  idx = HT\_HASH\_TO\_IDX(idx); // 64位系统中 index 不用转换  ht->nNumOfElements--; // 元素数量 -1  // 第二步，如果内部指针正指向 idx, 或哈希表有迭代器，更新内部指针和迭代器  if (ht->nInternalPointer == idx || UNEXPECTED(HT\_HAS\_ITERATORS(ht))) {  uint32\_t new\_idx;  new\_idx = idx;  while (1) { // 找到后面的第一个有效元素  new\_idx++; // 从下一个元素开始找  if (new\_idx >= ht->nNumUsed) { // 到达结尾元素  break; // 查找失败，跳出  } else if (Z\_TYPE(ht->arData[new\_idx].val) != IS\_UNDEF) { // 找到第一个有效元素  break; // 查找成功，跳出  }  }  if (ht->nInternalPointer == idx) { // 更新内部指针  ht->nInternalPointer = new\_idx; // new\_idx可能是无效值，后续会校正  }  zend\_hash\_iterators\_update(ht, idx, new\_idx); // 更新迭代器  }  if (ht->nNumUsed - 1 == idx) { // 第三步，如果当前元素是最后一个，更新使用元素数量  do { // 向左找，找到最后一个有效元素  ht->nNumUsed--; // 使用元素数-1  // 如果元素无效，并且还没到达开头位置，继续查找  } while (ht->nNumUsed > 0 &&  (UNEXPECTED(Z\_TYPE(ht->arData[ht->nNumUsed-1].val) == IS\_UNDEF)));  // 校正内部指针，如果指出有效范围，让它指向最后一个有效元素  ht->nInternalPointer = MIN(ht->nInternalPointer, ht->nNumUsed);  }  第四步，销毁元素  if (ht->pDestructor) { // 如果有销毁器  zval tmp;  ZVAL\_COPY\_VALUE(&tmp, &p->val); // 先把要销毁的变量复制出来  ZVAL\_UNDEF(&p->val); // 把当前元素设置成 IS\_UNDEF  ht->pDestructor(&tmp); // 删除旧变量  } else { // 没有销毁器，把当前元素设置成 IS\_UNDEF  ZVAL\_UNDEF(&p->val);  }  } |

如上所示，业务逻辑可分为4步：

第一步，在Bucket链表上摘除本元素；

第二步，如果使用了内部指针或迭代器，更新内部指针和迭代器；

第三步，如果当前元素是最后一个已使用元素，更新使用元素数量；

第四步，销毁元素。使用ZVAL\_UNDEF()宏程序把元素标记成无效，实际上只是把zval.type\_info更新成IS\_UNDEF（值为0），并没有删除zval实例中的数据。

HT\_HAS\_ITERATORS()宏程序的介绍参见“数组迭代器”章节。

### 通过键名（字符串类型）删除元素

通过字符串类型的键名删除元素，可使用zend\_hash\_str\_del()函数：

|  |
| --- |
| ZEND\_API zend\_result zend\_hash\_str\_del(HashTable \*ht, const char \*str, size\_t len){  zend\_ulong h;  uint32\_t nIndex;  uint32\_t idx;  Bucket \*p;  Bucket \*prev = NULL;  h = zend\_inline\_hash\_func(str, len); // 计算作为数组key的哈希值  nIndex = h | ht->nTableMask; // 取得哈希索引表中的序号  idx = HT\_HASH(ht, nIndex); // 取得元素序号  while (idx != HT\_INVALID\_IDX) { // 序号有效，继续查找  p = HT\_HASH\_TO\_BUCKET(ht, idx); // 找到哈希表中第bucket  // 哈希值相同，有key，并且key也相同，才算找到  if ((p->h == h) && p->key && zend\_string\_equals\_cstr(p->key, str, len)) {  zend\_string\_release(p->key); // 释放键名字符串  p->key = NULL; // 清空Bucket的键名指针  \_zend\_hash\_del\_el\_ex(ht, idx, p, prev); // 删除键值对的值  return SUCCESS; // 返回成功  }  // 不匹配，查找下一个  prev = p; // 切换到下一个元素  idx = Z\_NEXT(p->val); // 取出下一个元素的序号  }  return FAILURE; // 没找到要删除的元素，返回失败  } |

zend\_string\_equals\_cstr()函数用于比较两个字符串是否相等，详情参见“类型篇”。

### 通过键名（zend\_string实例）删除元素

通过zend\_string类型的键名删除元素，可使用zend\_hash\_del()函数：

|  |
| --- |
| ZEND\_API zend\_result zend\_hash\_del(HashTable \*ht, zend\_string \*key) {  ... // 前面与 zend\_result zend\_hash\_str\_del()函数逻辑相似  while (idx != HT\_INVALID\_IDX) { // 如果顺序号有效，进行检查  p = HT\_HASH\_TO\_BUCKET(ht, idx); // 取出此元素  // 指针值相同 或 哈希值和键名都匹配  if ((p->key == key) || (p->h == h && p->key &&  zend\_string\_equal\_content(p->key, key))) {  zend\_string\_release(p->key); // 删除key  p->key = NULL; // 指针置空  ... // 后面与 zend\_result zend\_hash\_str\_del()函数逻辑相似 |

如上所示，函数的业务逻辑与zend\_hash\_str\_del()相似，只是在比对键名时有些不同，因为Bucket实例的key元素也是zend\_string实例，所以比较键名时，可以先比较它和传入的键名（zend\_string实例）地址是否相同，如果地址相同，直接匹配成功。

### 通过键名（zend\_string实例）删除元素并处理间接引用目标

zend\_hash\_del\_ind()函数用于，通过键名（zend\_string实例）删除间接引用目标：

|  |
| --- |
| ZEND\_API zend\_result zend\_hash\_del\_ind(HashTable \*ht, zend\_string \*key) {  ... // 前面与 zend\_hash\_str\_del()函数逻辑相似  while (idx != HT\_INVALID\_IDX) {// 如果顺序号有效，进行检查  p = HT\_HASH\_TO\_BUCKET(ht, idx); // 找到Bucket元素  // key指针相同 或 （哈希值相同并且键名字符串相同）  if ((p->key == key) || (p->h == h && p->key &&  zend\_string\_equal\_content(p->key, key))) {  if (Z\_TYPE(p->val) == IS\_INDIRECT) { // 如果只间接引用  zval \*data = Z\_INDIRECT(p->val); // 追踪间接引用对象  if (UNEXPECTED(Z\_TYPE\_P(data) == IS\_UNDEF)) { // 如果目标无效  return FAILURE; // 返回 失败  } else { // 目标元素有效  if (ht->pDestructor) { // 有销毁器，销毁引用目标  zval tmp;  ZVAL\_COPY\_VALUE(&tmp, data); // 复制到临时变量中，并增加引用次数  ZVAL\_UNDEF(data); // 把目标对象设置成元素  ht->pDestructor(&tmp); // 销毁临时对象，须处理引用次数  } else { // 无销毁器，只把引用对象标记成无效  ZVAL\_UNDEF(data);  }  // 添加标记：哈希表包含空的间接引用元素  HT\_FLAGS(ht) |= HASH\_FLAG\_HAS\_EMPTY\_IND;  }  // 删除间接引用元素不调用 \_zend\_hash\_del\_el\_ex()函数 , 交给销毁器处理  } else { // 普通元素，常规删除  zend\_string\_release(p->key); // 删除key  p->key = NULL; // 清空key指针  \_zend\_hash\_del\_el\_ex(ht, idx, p, prev); // 删除键值对的值  }  return SUCCESS; // 返回成功  ... // 后面与 zend\_result zend\_hash\_str\_del()函数逻辑相似 |

如上所示，业务逻辑框架与zend\_result zend\_hash\_str\_del()函数相似，但是在删除元素时，增加了对间接引用类型的处理。

### 通过键名（字符串类型）删除元素并处理间接引用目标

zend\_hash\_str\_del\_ind()函数的业务逻辑与zend\_hash\_del\_ind()函数类似，只是传入的参数类型不同：

|  |
| --- |
| // 删除元素并处理间接引用目标。p1:哈希表指针，p2:键字符串指针，p3:键字符串长度  ZEND\_API zend\_result zend\_hash\_str\_del\_ind(HashTable \*ht, const char \*str, size\_t len) |

## 五）使用键名（key）查找哈希表中的元素

哈希表中查找元素，按参数类型不同可分为两组。

### 通过普通字符串键名进行查找

zend\_hash\_str\_find\_bucket()函数用于使用普通字符串作为键名查找元素，并返回Bucket指针：

|  |
| --- |
| static Bucket \*zend\_hash\_str\_find\_bucket(const HashTable \*ht, const char \*str, size\_t len, zend\_ulong h) {  uint32\_t nIndex;  uint32\_t idx;  Bucket \*p, \*arData;  arData = ht->arData; // 元素列表开头    nIndex = h | ht->nTableMask; // 计算哈希索引表中的顺序号  idx = HT\_HASH\_EX(arData, nIndex); // 元素顺序号  while (idx != HT\_INVALID\_IDX) { // 元素顺序号有效  p = HT\_HASH\_TO\_BUCKET\_EX(arData, idx); // 取出键值对  // 匹配条件：哈希值相同， 有key ，key里面的字串一致  if ((p->h == h)  && p->key  && zend\_string\_equals\_cstr(p->key, str, len)) {  return p;  }  idx = Z\_NEXT(p->val); // 匹配失败，切换到下一个元素  }  return NULL; // 查找元素失败  } |

zend\_hash\_str\_find\_bucket()函数有一些衍生函数，列表如下：

|  |  |  |
| --- | --- | --- |
| **函数名** | **说明** | **引用**  **次数** |
| zend\_hash\_str\_find() | 调用zend\_hash\_str\_find\_bucket()函数  查找并返回元素值，查找失败时返回NULL | 大量 |
| zend\_hash\_str\_find\_ptr() | 先给键名计算哈希值，再调用zend\_hash\_str\_find()函数，  查找并返回元素值中的指针，查找失败时返回NULL | 大量 |
| zend\_hash\_str\_find\_ind() | 调用zend\_hash\_str\_find()函数  查找并返回元素值，碰到间接引用类型时，追踪引用对象。查找失败或引用目标无效时返回NULL | 少量 |
| zend\_hash\_str\_find\_ptr\_lc() | 先为键名创建小写副本，再调用zend\_hash\_str\_find\_ptr()函数进行查找 | 少量 |
| zend\_hash\_str\_find\_deref() | 调用zend\_hash\_str\_find()函数进行查找，如果元素值有效，先减少元素值的引用次数，再返回元素值 | 大量 |
| zend\_hash\_str\_exists\_ind() | 调用zend\_hash\_str\_find()函数  查找并检查元素是否是有效的间接引用类型实例 | 无 |
| zend\_hash\_str\_exists() | 调用zend\_hash\_str\_find()函数  检查键名对应的元素是否存在 | 大量 |

### 通过zend\_string类型的键名进行查找

zend\_hash\_find\_bucket()函数用于使用zend\_string实例作为键名查找元素，并返回Bucket指针：

|  |
| --- |
| static Bucket \*zend\_hash\_find\_bucket(const HashTable \*ht, const zend\_string \*key) {  uint32\_t nIndex;  uint32\_t idx;  Bucket \*p, \*arData;  arData = ht->arData; // 元素列表开头  nIndex = ZSTR\_H(key) | ht->nTableMask; // 哈希索引表中的元素序号  idx = HT\_HASH\_EX(arData, nIndex); // 元素列表中的序号    if (UNEXPECTED(idx == HT\_INVALID\_IDX)) { // 如果 idx 无效（-1）  return NULL; // 查找失败  }  p = HT\_HASH\_TO\_BUCKET\_EX(arData, idx); // 找到链表第一个元素  if (EXPECTED(p->key == key)) { // **成功条件：键名和传入的zend\_string实例是同一个对象**  return p; // 返回Bucket指针  }  while (1) { // 遍历元素链表  // **成功条件：哈希值，并且键名存在 并且键名和传入的key匹配（大小写敏感）**  if (p->h == ZSTR\_H(key) &&  EXPECTED(p->key) &&  zend\_string\_equal\_content(p->key, key)) {  return p; // 匹配成功  }  idx = Z\_NEXT(p->val); // 下一个元素的序号  if (idx == HT\_INVALID\_IDX) { // 到达链表尽头  return NULL; // 查找失败  }  p = HT\_HASH\_TO\_BUCKET\_EX(arData, idx); // 取出下一个元素  if (p->key == key) { **// 成功条件：键名和传入的zend\_string实例是同一个对象**  return p; // 匹配成功  }  }  } |

zend\_hash\_find\_bucket()函数有一些衍生函数：

|  |  |  |
| --- | --- | --- |
| **函数名** | **说明** | **引用**  **次数** |
| zend\_hash\_find() | 为键名计算哈希值，并调用zend\_hash\_find\_bucket()函数进行查找，查找成功时返回元素值，失败时返回NULL | 大量 |
| zend\_hash\_find\_known\_hash() | 不计算哈希值，调用zend\_hash\_find\_bucket()函数进行查找，查找成功时返回元素值，失败时返回NULL | 大量 |
| zend\_hash\_exists() | 调用zend\_hash\_find()函数  检查键名对应的元素是否存在 | 大量 |
| zend\_hash\_exists\_ind() | 调用zend\_hash\_find()函数  查找并检查元素是否是有效的间接引用类型实例 | 少量 |
| zend\_hash\_find\_deref() | 调用zend\_hash\_find()函数进行查找，如果元素值有效，先减少元素值的引用次数，再返回元素值 | 少量 |
| zend\_hash\_find\_ptr() | 调用zend\_hash\_find()函数  查找并返回元素值中的指针 | 大量 |
| zend\_hash\_find\_ptr\_lc() | 先为键名创建小写副本，再调用zend\_hash\_find\_ptr()函数进行查找 | 少量 |
| zend\_hash\_find\_ind() | 调用zend\_hash\_find()函数  查找并返回元素值，碰到间接引用类型时，追踪引用对象。查找失败或引用目标无效时返回NULL | 少量 |
| zend\_hash\_find\_ex() | 有哈希值时调用zend\_hash\_find\_known\_hash()函数，  否则调用zend\_hash\_find()函数进行查找 | 少量 |
| zend\_hash\_find\_ex\_ind() | 调用zend\_hash\_find\_ex()函数  查找并返回元素值，碰到间接引用类型时，追踪引用对象，查找失败或引用目标无效时返回NULL | 少量 |
| zend\_hash\_find\_ex\_ptr() | 调用zend\_hash\_find\_ex()函数  查找并返回元素值中的指针 | 少量 |

## 六）通过索引号进行操作

哈希表中，除了按键名（key）操作，还可以按索引号（index）操作，按索引号操作时，函数调用方式与顺序数组相似，但哈希表中存储顺序与顺序数组不同。在哈希表中，元素都是按插入顺序保存的，先插入的元素保存在前面，后插入的元素保存在后面。

现用php脚本举例说明：

|  |
| --- |
| <?php  $a=[]; // 创建顺序数组  $a[5]=5; // 顺序数组中插入元素，序号5前面会有空元素  $a[2]=2; // 为了保证读取元素时索引号5在2前面，要把顺序数组转成哈希表  var\_dump($a);  ------打印结果------  array(2) {  [5]=> int(5)  [2]=> int(2)  } |

按索引号操作的大部分函数业务逻辑比较简单，通过调用\_zend\_hash\_index\_add\_or\_update\_i()函数来完成操作。

### 通过索引号添加元素

通过索引号添加元素的相关函数列表如下：

|  |  |  |
| --- | --- | --- |
| **函数名** | **说明** | **引用**  **次数** |
| zend\_hash\_index\_add() | 调用\_zend\_hash\_index\_add\_or\_update\_i()函数  数组中添加元素，找到旧元素会替换掉 | 大量 |
| zend\_hash\_index\_add\_new() | 调用\_zend\_hash\_index\_add\_or\_update\_i()函数  不查找旧元素，直接添加新元素 | 大量 |
| zend\_hash\_index\_add\_empty\_element() | 调用zend\_hash\_index\_add()函数  数组中添加值为 null的元素 | 少量 |
| zend\_hash\_index\_add\_mem() | 调用zend\_hash\_index\_add()函数 数组中添加内存副本指针 | 少量 |
| zend\_hash\_index\_add\_new\_ptr() | 调用zend\_hash\_index\_add\_new()函数 不查找旧元素，直接添加新元素 | 少量 |
| zend\_hash\_index\_add\_ptr() | 调用zend\_hash\_index\_add()函数 数组中添加指针 | 少量 |
| zend\_hash\_next\_index\_insert() | 调用\_zend\_hash\_index\_add\_or\_update\_i()函数  数组中添加元素，找到旧元素会替换掉 | 大量 |
| zend\_hash\_next\_index\_insert\_mem() | 调用zend\_hash\_next\_index\_insert()函数 插入内存副本指针，找到旧元素会替换掉。 | 少量 |
| zend\_hash\_next\_index\_insert\_ptr() | 调用zend\_hash\_next\_index\_insert()函数 插入指针，找到旧元素会替换掉。 | 大量 |

如上所示，添加元素主要用到\_zend\_hash\_index\_add\_or\_update\_i()函数的3个衍生函数：zend\_hash\_index\_add()、zend\_hash\_index\_add\_new()和zend\_hash\_next\_index\_insert()，以及它们的衍生函数。

它们在调用\_zend\_hash\_index\_add\_or\_update\_i()函数时略有不同：

1）zend\_hash\_index\_add()函数，在哈希表中找到旧元素会替换掉；

2）zend\_hash\_index\_add\_new()函数，不查找旧元素，直接添加新元素；

3）zend\_hash\_next\_index\_insert()函数不需要传入索引号，它会按现有元素数量自动生成索引号。这3个函数的代码如下：

|  |
| --- |
| ZEND\_API zval\* zend\_hash\_index\_add(HashTable \*ht, zend\_ulong h, zval \*pData) {  return \_zend\_hash\_index\_add\_or\_update\_i(ht, h, pData, HASH\_ADD);  }  ZEND\_API zval\* zend\_hash\_index\_add\_new(HashTable \*ht, zend\_ulong h, zval \*pData) {  return \_zend\_hash\_index\_add\_or\_update\_i(ht, h, pData, HASH\_ADD | HASH\_ADD\_NEW);  }  ZEND\_API zval\* zend\_hash\_next\_index\_insert(HashTable \*ht, zval \*pData) {  return \_zend\_hash\_index\_add\_or\_update\_i(ht, ht->nNextFreeElement, pData, HASH\_ADD | HASH\_ADD\_NEXT);  } |

对于zend\_hash\_index\_add\_new()函数和它的衍生函数，要谨慎使用，因为它不对已有元素进行检查，可能导致添加相同索引号的元素。

### 通过索引号更新元素

通过索引号更新元素主要用到以下几个函数：

|  |  |  |
| --- | --- | --- |
| **函数名** | **说明** | **引用**  **次数** |
| zend\_hash\_index\_update() | 调用\_zend\_hash\_index\_add\_or\_update\_i()函数  数组中添加元素，找到旧元素会替换掉 | 大量 |
| zend\_hash\_index\_update\_ptr() | 调用zend\_hash\_index\_update()函数  数组中添加指针，找到旧元素会替换掉 | 大量 |
| zend\_hash\_index\_update\_mem() | 调用zend\_hash\_index\_update()函数 数组中添加内存副本指针，找到旧元素会替换掉 | 少量 |

通过索引号添和更新元素时，主要用到\_zend\_hash\_index\_add\_or\_update\_i()函数的后半段业务逻辑：

|  |
| --- |
| static zval \*\_zend\_hash\_index\_add\_or\_update\_i(HashTable \*ht, zend\_ulong h, zval \*pData, uint32\_t flag) {  // ... 前半段业务逻辑用来处理顺序数组和未初始化数组，已在顺序数组中介绍过  } else { // 情况3：如果数组是哈希表（初始化过）。 只有这里单独用到 HASH\_ADD\_NEW  // 情况3.1: 如果不是 HASH\_ADD\_NEW 添加新元素，那就是 【查找】 或 【更新】  if ((flag & HASH\_ADD\_NEW) == 0) {  // 【查找】 或 【更新】都要先查找  // 使用 哈希值(zend\_ulong) 获取哈希表里【没有key，只有哈希值的键值对】  p = zend\_hash\_index\_find\_bucket(ht, h);  if (p) { // 情况3.1.1: 如果能查到旧元素  if (flag & HASH\_LOOKUP) { // 情况3.1.1.1如果是【查找】  return &p->val; // 直接返回  }  // 情况3.1.1.2是【添加】或【更新】  zv = &p->val; // 取出键值对的值 指针  goto replace; // 转到替换，并返回  }  // 情况3.1.2: 【查找】 或 【更新】时，没有旧值，往下走。  // 【查找】应返回空元素，【更新】应插入新元素  }  // 情况3.2: HASH\_ADD\_NEW  // 【查找】不可能出现。 HASH\_ADD\_NEW 和 HASH\_LOOKUP 不同时使用  // 【更新】应插入新元素（HASH\_ADD | HASH\_ADD\_NEW，或 HASH\_ADD | HASH\_ADD\_NEW | HASH\_ADD\_NEXT)  // 哈希表有没有 HASH\_ADD\_NEXT 没区别。  ZEND\_HASH\_IF\_FULL\_DO\_RESIZE(ht); // 如果哈希表满了，改变大小  }  idx = ht->nNumUsed++; // 索引号，自增  nIndex = h | ht->nTableMask; // 计算哈希索引表中的序号  p = ht->arData + idx; // 找到指定的 bucket  Z\_NEXT(p->val) = HT\_HASH(ht, nIndex); // 新元素串在相同哈希值元素列表的开头  HT\_HASH(ht, nIndex) = HT\_IDX\_TO\_HASH(idx); // 哈希索引表中的元素指向链表开头  if ((zend\_long)h >= ht->nNextFreeElement) { // 如果哈希值大于下一个空元素  // 更新下一个空元素序号，序号是 h+1, 防止超长。  ht->nNextFreeElement = (zend\_long)h < ZEND\_LONG\_MAX ? h + 1 : ZEND\_LONG\_MAX;  }  ht->nNumOfElements++; // 元素数量+1  p->h = h; // 更新哈希值  p->key = NULL; // key 为null  if (flag & HASH\_LOOKUP) { // 如果是查询  ZVAL\_NULL(&p->val); // 查询时把val更新成NULL  } else { // 不是查询  ZVAL\_COPY\_VALUE(&p->val, pData); // 更新键值对的值  }  return &p->val; // 返回 键值对的值  } |

\_zend\_hash\_index\_add\_or\_update\_i()函数比较复杂，代码量比较大，在介绍顺序数组和哈希表时，把此函数拆分开进行介绍。此函数非常重要，完整的代码注释和业务逻辑分支表可在附录中查看。

### 通过索引号删除元素

见下文“通过序号（索引号）删除数组元素”章节中的zend\_hash\_index\_del()函数。

### 通过索引号查找元素

按索引号查找元素主要用到以下几个函数：

|  |  |  |
| --- | --- | --- |
| **函数名** | **说明** | **引用**  **次数** |
| zend\_hash\_index\_find\_bucket() | 在哈希表中使用索引号查找，并返回元素值 | 少量 |
| zend\_hash\_index\_find() | 兼容顺序数组和哈希表的查找函数  对哈希表调用zend\_hash\_index\_find\_bucket()函数查找，并返回元素值 | 大量 |
| \_zend\_hash\_index\_find() | 调用zend\_hash\_index\_find\_bucket()函数  在哈希表中查找，并返回元素值 | 少量 |
| zend\_hash\_index\_find\_ptr() | 调用zend\_hash\_index\_find()函数进行查找  查找成功返回元素值中的指针，失败返回NULL | 大量 |
| zend\_hash\_index\_find\_deref() | 调用zend\_hash\_index\_find()函数进行查找，  减少元素值的引用次数，返回元素值 | 少量 |
| zend\_hash\_index\_exists() | 检验索引号是否存在  return zend\_hash\_index\_find(ht, h) != NULL; | 大量 |

如上所示，通过索引号查找哈希表元素用到zend\_hash\_index\_find\_bucket()函数和它的衍生函数。

zend\_hash\_index\_find\_bucket()函数用于通过索引号查找没有键名的Bucket元素，查找成功时返回Bucket指针：

|  |
| --- |
| static Bucket \*zend\_hash\_index\_find\_bucket(const HashTable \*ht, zend\_ulong h) {  uint32\_t nIndex;  uint32\_t idx;  Bucket \*p, \*arData;  arData = ht->arData; // 元素列表开头  nIndex = h | ht->nTableMask; // 计算哈希索引表中的元素序号  idx = HT\_HASH\_EX(arData, nIndex); // 取出元素序号  while (idx != HT\_INVALID\_IDX) { // 序号有效，一直找  p = HT\_HASH\_TO\_BUCKET\_EX(arData, idx); // 取出Bucket元素  if (p->h == h && !p->key) { // 成功条件：如果哈希值匹配，并且元素没有键名  return p; // 返回元素  }  idx = Z\_NEXT(p->val); // 切换到下一个序号  }  return NULL; // 查找失败  } |

其他几个衍生函数逻辑都比较简单。

## 七）整理哈希表

zend\_hash\_rehash()函数用于整理哈希表：

|  |
| --- |
| ZEND\_API void zend\_hash\_rehash(HashTable \*ht) {  Bucket \*p;  uint32\_t nIndex, i; |

此函数的业务逻辑有些复杂，在整理哈希表时，可能会碰到3种情况。

### 1）情况1，整理空哈希表

整理空哈希表的业务逻辑比较简单：

|  |
| --- |
| // 紧接上一块代码  if (UNEXPECTED(ht->nNumOfElements == 0)) { // 情况1：如果里面没有元素  if (!(HT\_FLAGS(ht) & HASH\_FLAG\_UNINITIALIZED)) { // 如果不是 未初始化状态  ht->nNumUsed = 0; // 已用元素数为0  HT\_HASH\_RESET(ht); // 哈希索引列表的所有元素都设置成 -1  }  // 如果哈希表未初始化，什么也不做  return; // 中断  } |

### 2）情况2，整理元素连续的希表

如果哈希表不是空的，但其中的元素是连续的，没有无效位置，处理逻辑也相对简单：

|  |
| --- |
| // 紧接上一块代码，  // 如果哈希表里有元素  HT\_HASH\_RESET(ht); // 哈希索引列表的所有元素都设置成 -1  i = 0; // 遍历起始序号  p = ht->arData; // bucket列表开始头  if (HT\_IS\_WITHOUT\_HOLES(ht)) { // 情况2：如果 有元素 且 中间没有空元素  do { // 遍历全部元素，刷新哈希索引表和元素链表  nIndex = p->h | ht->nTableMask; // 计算哈希索引表的元素序号  Z\_NEXT(p->val) = HT\_HASH(ht, nIndex); // 当前元素指向原来的链表开头  // 序号存入哈希索引表中，让当前元素变成链表开头  HT\_HASH(ht, nIndex) = HT\_IDX\_TO\_HASH(i);  p++; // 下一个  } while (++i < ht->nNumUsed); |

如上所示，遍历每一个元素，刷新哈希索引表，并重建每一个哈希值对应的元素链表。

### 2）情况3，整理元素不连续的希表

如果哈希表不为空，并且元素列表的使用不是连续的，需要先通过一次遍历，找到不连续的位置：

|  |
| --- |
| } else { // 情况3：有元素且 中间有空元素  uint32\_t old\_num\_used = ht->nNumUsed; // 原使用数量  do { // 遍历使用过的元素  // 子元素情况3.1：找到空位（无效元素），需要把后面的元素移过来填充空位  if (UNEXPECTED(Z\_TYPE(p->val) == IS\_UNDEF)) {  uint32\_t j = i; // j是空位序号  Bucket \*q = p; // q是空位置的bucket指针，p是移动指针  if (EXPECTED(!HT\_HAS\_ITERATORS(ht))) { // 子元素情况3.1.1：哈希表没有迭代器  ... **// 没有迭代器时的处理逻辑**  } else { // 子元素情况3.1.2： 哈希表有迭代器  ... **// 有迭代器时的处理逻辑**  }  ht->nNumUsed = j; // 更新已用元素数  break; // 整理完后面的元素后，直接跳出循环  }  // 子元素情况3.2：在发现空元素前，处理有效元素，更新它的序号和next指针  nIndex = p->h | ht->nTableMask; // 计算哈希索引表中的序号  // q作为新的链表开头，指向原来的开头位置（zval.u2.next里保存原来的头元素序号）  Z\_NEXT(p->val) = HT\_HASH(ht, nIndex);  // 找到哈希索引列表中的指定索，把顺序号写进去  HT\_HASH(ht, nIndex) = HT\_IDX\_TO\_HASH(i);  p++; // 后移  } while (++i < ht->nNumUsed); // 遍历使用过的元素 |

如上所示，使用一个while循环遍历所有元素，在碰到空元素前，先处每一个正常元素（情况3.2）。碰到空元素时，需要把后面的元素迁移过来，填充空位置，保证元素列表的使用是连续的。

#### **没有附加迭代器时的处理**

在上一块代码中，如果哈希表没有附加迭代器，处理逻辑如下：

|  |
| --- |
| while (++i < ht->nNumUsed) { // 从空位置向后查找有效元素  p++; // 逐个检查是否无效  if (EXPECTED(Z\_TYPE\_INFO(p->val) != IS\_UNDEF)) { // 如果p元素有效  ZVAL\_COPY\_VALUE(&q->val, &p->val); // 把有效元素的值复制到空元素  q->h = p->h;// 哈希值复制到空元素  nIndex = q->h | ht->nTableMask; // 计算哈希索引列表中的元素序号  q->key = p->key; // 复制键名（指针）到空元素  // q作为新的链表开头，指向原来的开头位置（zval.u2.next里保存原来的头元素序号）  Z\_NEXT(q->val) = HT\_HASH(ht, nIndex);  // 找到哈希索引列表中的指定索，把顺序号写进去  HT\_HASH(ht, nIndex) = HT\_IDX\_TO\_HASH(j); // 64位系统中 index 不用转换  if (UNEXPECTED(ht->nInternalPointer == i)) { // 如果内部指针指向本元素  ht->nInternalPointer = j; // 更新内部的指针位置  }  q++; // 下一个 bucket  j++; // 下一个编号  }  } |

如上所示，通过一个while循环遍历空元素后面所有元素，逐个检查是否是无效元素。如果碰到有效元素，就把它迁移到前面选好的位置中去。在这一步里，变量i用来保证遍历不会超出范围。指针q一开始指向空位，指针p一直向后查找有效元素，找到有效元素后填充到指针q指向的位置，然后指针q后移一个元素，保证迁移后元素的使用是连续的，如此重复直到所有元素都迁移完毕。

迁移的过程中，都是用后面的元素覆盖前面的元素，后面会有一些元素没有被清理掉，但这不影响使用，因为迁移完所有元素后，会更新哈希表的使用元素数ht->nNumUsed（情况3.1的末尾，代码ht->nNumUsed = j;），序号大于ht->nNumUsed的元素无论里面是否有数据，都会当成闲置空间来处理。

示意图如下，元素迁移前，ht->nNumUsed值为8：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 元素1 | 无效 | 元素2 | 无效 | 元素3 | 无效 | 无效 | 元素4 |
| 位置1 | 位置2 | 位置3 | 位置4 | 位置5 | 位置6 | 位置7 | 位置8 |

元素迁移后，ht->nNumUsed值为4：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 元素1 | 元素2 | 元素3 | 元素4 | 元素3 | 无效 | 无效 | 元素4 |
| 位置1 | 位置2 | 位置3 | 位置4 | 位置5 | 位置6 | 位置7 | 位置8 |

位置4后面的元素，无论有无数据都会被当成空闲空间处理。

在遍历过程中，每次填充完毕后需要处理元素链表，把元素添加到所属链表的开头。还需要处理内部指针，让指针跟随元素，指向新的位置。

#### **有附加迭代器时的处理**

如果此哈希表附加迭代器，在每一次迁移元素后，都需要增加一些处理迭代器的业务逻辑：

|  |
| --- |
| uint32\_t iter\_pos = zend\_hash\_iterators\_lower\_pos(ht, 0); // 所有迭代器的最小迭代位置（序号）  while (++i < ht->nNumUsed) { // 从空位置向后查找有效元素  p++;  if (EXPECTED(Z\_TYPE\_INFO(p->val) != IS\_UNDEF)) { // 如果p元素有效  ... // 迁移元素过程相同  // 如果迁移的元素在迭代器位置之后，本次修改会影响到迭代过程  if (UNEXPECTED(i >= iter\_pos)) {  do {  // 校正迭代器指向位置，保证下一个访问的元素正确  zend\_hash\_iterators\_update(ht, iter\_pos, j);  // 所有迭代器中最小的迭代位置，从iter\_pos+1开始，起始点一直向后推直到所有受影响的位置都被校正过  iter\_pos = zend\_hash\_iterators\_lower\_pos(ht, iter\_pos + 1);  } while (iter\_pos < i); // 只要还有迭代器中有位置在i前面，就一直查找  }  q++; // 下一个 bucket  j++; // 下一个编号  }  } |

如上所示，迭代器的校正在每次迁移元素后都会触发，这样能保证每次处理的位置都应该校正到指向当前元素。在所有元素迁移完毕后，还需要再更新一次迭代器，保证所有指向列表结尾的迭代器指向正确：

|  |
| --- |
| if (UNEXPECTED(HT\_HAS\_ITERATORS(ht))) {  // 校正指向元素列表结尾的迭代器  \_zend\_hash\_iterators\_update(ht, old\_num\_used, ht->nNumUsed);  } |

## 八）遍历哈希表

遍历哈希表通过宏程序来完成，主要有正序、倒序两种遍历方式。

### 正序遍历哈希表

正序遍历顺序数组主要用到ZEND\_HASH\_MAP\_FOREACH\_FROM()宏程序：

|  |
| --- |
| // 遍历哈希表。p1:哈希表指针，p2:是否追踪间接引用，p3:起始元素序号  #define ZEND\_HASH\_MAP\_FOREACH\_FROM(\_ht, indirect, \_from) do { \  HashTable \*\_\_ht = (\_ht); /\* 哈希表指针 \*/ \  Bucket \*\_p = \_\_ht->arData + (\_from); /\* 开始位置 \*/ \  Bucket \*\_end = \_\_ht->arData + \_\_ht->nNumUsed; /\* 结束位置 \*/ \  for (; \_p != \_end; \_p++) { /\* 遍历每一个 Bucket \*/ \  zval \*\_z = &\_p->val; /\* 值的指针 \*/ \  /\* 如果要追踪间接引用，并且当前值是间接引用类型 \*/ \  if (indirect && Z\_TYPE\_P(\_z) == IS\_INDIRECT) { \  \_z = Z\_INDIRECT\_P(\_z); /\* 追踪间接引用目标 \*/ \  } \  /\* 如果 当前元素的值无效，跳过 \*/ \  if (UNEXPECTED(Z\_TYPE\_P(\_z) == IS\_UNDEF)) continue; |

如上所示，ZEND\_HASH\_MAP\_FOREACH\_FROM()宏程序是遍历过程的前半段业务逻辑，它实现了遍历哈希表元素列表中的每个元素，调用这个宏程序后，即可添加需要的业务逻辑，最后调用ZEND\_HASH\_FOREACH\_END()宏程序结束遍历。但在实际使用中，并不直接调用ZEND\_HASH\_MAP\_FOREACH\_FROM()宏程序，而是调用它的几组衍生宏程序。

#### **第一组，遍历哈希表并访问Bucket实例**

相关宏程序如下：

|  |
| --- |
| // 辅助宏程序，从头开始遍历哈希表  #define ZEND\_HASH\_MAP\_FOREACH(\_ht, indirect) ZEND\_HASH\_MAP\_FOREACH\_FROM(\_ht, indirect, 0)  // 遍历哈希表，通过 \_bucket 变量访问每个Bucket实例  #define ZEND\_HASH\_MAP\_FOREACH\_BUCKET(ht, \_bucket) \  ZEND\_HASH\_MAP\_FOREACH(ht, 0); \  \_bucket = \_p; /\* 供外部访问的Bucket指针 \*/  // ZEND\_HASH\_MAP\_FOREACH\_BUCKET()宏程序的别名  #define ZEND\_HASH\_FOREACH\_BUCKET(ht, \_bucket) \  ZEND\_HASH\_MAP\_FOREACH\_BUCKET(ht, \_bucket) |

#### **第二组，遍历哈希表并分别访问哈希值、键名和元素值**

本组共有7个宏程序：

|  |
| --- |
| // 遍历哈希表，通过\_h、 \_key、 \_val变量访问每个哈希值、键名和元素值  #define ZEND\_HASH\_MAP\_FOREACH\_KEY\_VAL(ht, \_h, \_key, \_val) \  ZEND\_HASH\_MAP\_FOREACH(ht, 0); \  /\* 供外部访问 \*/ \  \_h = \_p->h; \  \_key = \_p->key; \  \_val = \_z;  // 遍历哈希表，通过 \_val 变量访问每个元素值  #define ZEND\_HASH\_MAP\_FOREACH\_VAL(ht, \_val) // 逻辑类似  // 遍历哈希表，通过\_h变量访问每个哈希值  #define ZEND\_HASH\_MAP\_FOREACH\_NUM\_KEY(ht, \_h) // 逻辑类似    // 遍历哈希表，通过 \_key 变量访问每个键名  #define ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY(ht, \_key) // 逻辑类似    // 遍历哈希表，通过\_h、 \_key变量访问每个哈希值和键名  #define ZEND\_HASH\_MAP\_FOREACH\_KEY(ht, \_h, \_key) // 逻辑类似    // 遍历哈希表，通过\_h、\_val变量访问每个哈希值和元素值  #define ZEND\_HASH\_MAP\_FOREACH\_NUM\_KEY\_VAL(ht, \_h, \_val) // 逻辑类似    // 遍历哈希表，通过\_key、 \_val变量访问每个键名和元素值  #define ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY\_VAL(ht, \_key, \_val) // 逻辑类似 |

#### **第三组，遍历哈希表并分别访问哈希值、键名和元素值中的指针**

本组共有4个宏程序：

|  |
| --- |
| // 遍历哈希表，通过\_h、 \_key、 \_ptr变量访问每个哈希值、键名和元素值中的指针  #define ZEND\_HASH\_MAP\_FOREACH\_KEY\_PTR(ht, \_h, \_key, \_ptr) \  ZEND\_HASH\_MAP\_FOREACH(ht, 0); \  /\* 供外部访问 \*/ \  \_h = \_p->h; /\* 供外部访问的哈希值 \*/ \  \_key = \_p->key; /\* 供外部访问的键（zend\_string实例）指针 \*/ \  \_ptr = Z\_PTR\_P(\_z); /\* 供外部访问的元素值中的指针 \*/    // 遍历哈希表，通过\_h、\_ptr变量访问每个哈希值和元素值中的指针  #define ZEND\_HASH\_MAP\_FOREACH\_NUM\_KEY\_PTR(ht, \_h, \_ptr) // 逻辑类似    // 遍历哈希表，通过\_key、\_ptr变量访问每个键名和元素值中的指针  #define ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY\_PTR(ht, \_key, \_ptr) // 逻辑类似    // 遍历哈希表，通过\_ptr变量访问每个元素值中的指针  #define ZEND\_HASH\_MAP\_FOREACH\_PTR(ht, \_ptr) // 逻辑类似 |

#### **第四组，遍历哈希表并分别访问哈希值、键名和元素值，自动追踪间接引用**

本组共有3个宏程序：

|  |
| --- |
| // 遍历哈希表，自动追踪间接引用，通过\_h、 \_key、 \_val变量访问每个哈希值、键名和元素值  #define ZEND\_HASH\_MAP\_FOREACH\_KEY\_VAL\_IND(ht, \_h, \_key, \_val) \  ZEND\_HASH\_MAP\_FOREACH(ht, 1); \  \_h = \_p->h; /\* 供外部访问的哈希值 \*/ \  \_key = \_p->key; /\* 供外部访问的键（zend\_string实例）指针 \*/ \  \_val = \_z; /\* 供外部访问的元素值 \*/    // 遍历哈希表，自动追踪间接引用，通过\_val变量访问每个元素值  #define ZEND\_HASH\_MAP\_FOREACH\_VAL\_IND(ht, \_val) // 逻辑类似    // 遍历哈希表，自动追踪间接引用，通过\_key、 \_val变量访问每个键名和元素值  #define ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY\_VAL\_IND(ht, \_key, \_val) // 逻辑类似 |

#### **第五组，从指定位置开始遍历哈希表并分别访问哈希值、键名和元素值**

本组共有4个宏程序：

|  |
| --- |
| // 从指定位置开始遍历哈希表，通过 \_key，\_val 变量访问每个键和值  #define ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY\_VAL\_FROM(ht, \_key, \_val, \_from) \  ZEND\_HASH\_MAP\_FOREACH\_FROM(ht, 0, \_from); \  \_key = \_p->key; /\* 供外部访问 \*/ \  \_val = \_z;    // 从指定位置开始遍历哈希表，通过\_bucket变量访问每个元素，可指定起始位置，不追踪间接引用  #define ZEND\_HASH\_MAP\_FOREACH\_BUCKET\_FROM(ht, \_bucket, \_from) // 逻辑类似    // 从指定位置开始遍历哈希表，通过 \_ptr 变量访问每个元素值中的指针  #define ZEND\_HASH\_MAP\_FOREACH\_PTR\_FROM(ht, \_ptr, \_from) // 逻辑类似  // ZEND\_HASH\_MAP\_FOREACH\_BUCKET\_FROM()宏程序的别名  #define ZEND\_HASH\_FOREACH\_BUCKET\_FROM(ht, \_bucket, \_from) \  ZEND\_HASH\_MAP\_FOREACH\_BUCKET\_FROM(ht, \_bucket, \_from) |

### 倒序遍历哈希表

倒序遍历顺序数组与正序遍历相似，主要用到ZEND\_HASH\_MAP\_REVERSE\_FOREACH()宏程序：

|  |
| --- |
| #define ZEND\_HASH\_MAP\_REVERSE\_FOREACH(\_ht, indirect) do { \  HashTable \*\_\_ht = (\_ht); /\* 哈希表指针 \*/ \  uint32\_t \_idx = \_\_ht->nNumUsed; /\* 已使用数量 \*/ \  Bucket \*\_p = \_\_ht->arData + \_idx; /\* 列表结尾 \*/ \  zval \*\_z; /\* 临时变量 \*/ \  for (\_idx = \_\_ht->nNumUsed; \_idx > 0; \_idx--) { /\* 倒序遍历 \*/ \  \_p--; /\* 指针左移1个 \*/ \  \_z = &\_p->val; /\* bucket里的值 \*/ \  if (indirect && Z\_TYPE\_P(\_z) == IS\_INDIRECT) { /\* 如果是间接引用并需要追踪引用 \*/ \  \_z = Z\_INDIRECT\_P(\_z); /\* 追踪间接引用 \*/ \  } \  if (UNEXPECTED(Z\_TYPE\_P(\_z) == IS\_UNDEF)) continue; /\* 跳过无效元素 \*/ |

以及几组衍生宏程序，这些宏程序的业务逻辑与正序遍历类似。

#### **第一组，倒序遍历哈希表并访问Bucket实例**

本组共有2个宏程序：

|  |
| --- |
| // 倒序遍历哈希表，通过 \_bucket 变量访问每个Bucket实例  #define ZEND\_HASH\_MAP\_REVERSE\_FOREACH\_BUCKET(ht, \_bucket) \  ZEND\_HASH\_MAP\_REVERSE\_FOREACH(ht, 0); \  \_bucket = \_p; /\* 供外部访问 \*/  // ZEND\_HASH\_MAP\_REVERSE\_FOREACH\_BUCKET()宏程序的别名  #define ZEND\_HASH\_REVERSE\_FOREACH\_BUCKET(ht, \_bucket) \  ZEND\_HASH\_MAP\_REVERSE\_FOREACH\_BUCKET(ht, \_bucket) |

#### **第二组，倒序遍历哈希表并分别访问哈希值、键名和元素值**

本组共有7个宏程序：

|  |
| --- |
| // 倒序遍历哈希表，通过\_h、 \_key、 \_val变量访问每个哈希值、键名和元素值  #define ZEND\_HASH\_MAP\_REVERSE\_FOREACH\_KEY\_VAL(ht, \_h, \_key, \_val) // 逻辑类似    // 倒序遍历哈希表，通过 \_val 变量访问每个元素值  #define ZEND\_HASH\_MAP\_REVERSE\_FOREACH\_VAL(ht, \_val) // 逻辑类似  // 倒序遍历哈希表，通过\_h变量访问每个哈希值  #define ZEND\_HASH\_MAP\_REVERSE\_FOREACH\_NUM\_KEY(ht, \_h) // 逻辑类似  // 倒序遍历哈希表，通过 \_key 变量访问每个键名  #define ZEND\_HASH\_MAP\_REVERSE\_FOREACH\_STR\_KEY(ht, \_key) // 逻辑类似  // 倒序遍历哈希表，通过\_h、 \_key变量访问每个哈希值和键名  #define ZEND\_HASH\_MAP\_REVERSE\_FOREACH\_KEY(ht, \_h, \_key) // 逻辑类似  // 倒序遍历哈希表，通过\_h、\_val变量访问每个哈希值和元素值  #define ZEND\_HASH\_MAP\_REVERSE\_FOREACH\_NUM\_KEY\_VAL(ht, \_h, \_val) // 逻辑类似  // 倒序遍历哈希表，通过\_key、 \_val变量访问每个键名和元素值  #define ZEND\_HASH\_MAP\_REVERSE\_FOREACH\_STR\_KEY\_VAL(ht, \_key, \_val) // 逻辑类似 |

#### **第三组，倒序遍历哈希表并分别访问哈希值、键名和元素值中的指针**

本组共有4个宏程序：

|  |
| --- |
| // 倒序遍历哈希表，通过\_h、 \_key、 \_ptr变量访问哈希值、键名和元素值中的指针  #define ZEND\_HASH\_MAP\_REVERSE\_FOREACH\_KEY\_PTR(ht, \_h, \_key, \_ptr) // 逻辑类似  // 倒序遍历哈希表，通过\_h、\_ptr变量访问哈希值和元素值中的指针  #define ZEND\_HASH\_MAP\_REVERSE\_FOREACH\_NUM\_KEY\_PTR(ht, \_h, \_ptr) // 逻辑类似  // 倒序遍历哈希表，通过\_key、\_ptr变量访问键名和元素值中的指针  #define ZEND\_HASH\_MAP\_REVERSE\_FOREACH\_STR\_KEY\_PTR(ht, \_key, \_ptr) // 逻辑类似  // 倒序遍历哈希表，通过\_ptr变量访问元素值中的指针  #define ZEND\_HASH\_MAP\_REVERSE\_FOREACH\_PTR(ht, \_ptr) // 逻辑类似 |

#### **第四组，倒序遍历哈希表并分别访问哈希值、键名和元素值，自动追踪间接引用**

本组共有3个宏程序：

|  |
| --- |
| // 倒序遍历哈希表，自动追踪间接引用，通过\_h、 \_key、 \_val变量访问哈希值、键名和元素值  #define ZEND\_HASH\_MAP\_REVERSE\_FOREACH\_KEY\_VAL\_IND(ht, \_h, \_key, \_val) // 逻辑类似    // 倒序遍历哈希表，自动追踪间接引用，通过\_val变量访问元素值  #define ZEND\_HASH\_MAP\_REVERSE\_FOREACH\_VAL\_IND(ht, \_val) // 逻辑类似    // 倒序遍历哈希表，自动追踪间接引用，通过\_key、 \_val变量访问键名和元素值  #define ZEND\_HASH\_MAP\_REVERSE\_FOREACH\_STR\_KEY\_VAL\_IND(ht, \_key, \_val) // 逻辑类似 |

#### **倒序遍历时自动删除每个元素**

如果想在倒序遍历时，处理完每个元素后把元素删除掉，可以在遍历的结尾调用ZEND\_HASH\_MAP\_FOREACH\_END\_DEL()宏程序，代码如下：

|  |
| --- |
| // 一定在倒序遍历后使用，因为正序遍历没有 \_idx 变量  #define ZEND\_HASH\_MAP\_FOREACH\_END\_DEL() \  \_\_ht->nNumOfElements--; /\* 删除1个元素（计数减1） \*/ \  do { \  /\* j是当前Bucket的序号，由于从\_idx = nNumUsed开始倒序遍历，所以要\_idx-1 \*/ \  uint32\_t j = HT\_IDX\_TO\_HASH(\_idx - 1); \  uint32\_t nIndex = \_p->h | \_\_ht->nTableMask; /\* 取得哈希值索引表的元素序号 \*/ \  uint32\_t i = HT\_HASH(\_\_ht, nIndex); /\* 取得Bucket列表中的顺序号 \*/ \  if (UNEXPECTED(j != i)) { /\* 如果当前Bucket不是Bucket链表的第一个元素 \*/ \  Bucket \*prev = HT\_HASH\_TO\_BUCKET(\_\_ht, i); /\* 链表的第一个Bucket元素 \*/ \  /\* 顺着链表查找，找到指向序号j的元素 \*/ \  while (Z\_NEXT(prev->val) != j) { /\* 不是指向j就一直找 \*/ \  i = Z\_NEXT(prev->val); /\* 取出序号 \*/ \  prev = HT\_HASH\_TO\_BUCKET(\_\_ht, i); /\* 切换到链表的下一个元素 \*/ \  } \  /\* 这时一定找到了指向j的元素，让它指向j的下一个元素，把j从链上摘除 \*/ \  Z\_NEXT(prev->val) = Z\_NEXT(\_p->val); \  } else { /\* 是相同哈希值元素链表的第一个元素 \*/ \  /\* 把链表第一个元素摘除，让哈希索引表指向链表第二个元素 \*/ \  HT\_HASH(\_\_ht, nIndex) = Z\_NEXT(\_p->val); \  } \  } while (0); \  } \  \_\_ht->nNumUsed = \_idx; /\* 整个遍历全结束后，更新使用数量，应该为0 \*/ \  } while (0) |

它有一个别名：

|  |
| --- |
| // ZEND\_HASH\_MAP\_FOREACH\_END\_DEL()宏程序的别名  #define ZEND\_HASH\_FOREACH\_END\_DEL() ZEND\_HASH\_MAP\_FOREACH\_END\_DEL() |

调用示例：

|  |
| --- |
| ZEND\_HASH\_MAP\_REVERSE\_FOREACH\_STR\_KEY\_VAL(ht, key, zv) { // 倒序遍历哈希表  ... // 业务逻辑  } ZEND\_HASH\_MAP\_FOREACH\_END\_DEL(); // 在遍历的过程中，依次删除每个元素 |

## 九）其他常用操作

### 快速创建哈希表

\_zend\_new\_array\_0()函数可以用来快速创建哈希表：

|  |
| --- |
| ZEND\_API HashTable\* \_zend\_new\_array\_0(void) {  HashTable \*ht = emalloc(sizeof(HashTable)); // 分配内存创建哈希表  // 初始化哈希表，只设置基础属性，未创建元素空间  \_zend\_hash\_init\_int(ht, HT\_MIN\_SIZE, ZVAL\_PTR\_DTOR, 0);  return ht;  } |

### 给哈希表增加空间

给哈希表增加空间使用zend\_hash\_do\_resize()函数：

|  |
| --- |
| static void zend\_hash\_do\_resize(HashTable \*ht) {  // 如果使用率小于97%（32/33的近似值）  // 占用元素数 > 有效元素数 + 有效元素数/32 => 32/33 > 有效元素数 / 占用元素数  if (ht->nNumUsed > ht->nNumOfElements + (ht->nNumOfElements >> 5)) {  zend\_hash\_rehash(ht); // 整理哈希表  // 如果使用率大于97%，并且大小小于最大值，让哈希表大小翻倍  } else if (ht->nTableSize < HT\_MAX\_SIZE) {  void \*new\_data, \*old\_data = HT\_GET\_DATA\_ADDR(ht); // 旧数据块指针  uint32\_t nSize = ht->nTableSize + ht->nTableSize; // 元素数量翻倍  Bucket \*old\_buckets = ht->arData; // 旧元素指针  ht->nTableSize = nSize; // 更新大小  // 计算数据块大小，分配内存创建数据块。这里重新分配内存，而不是更改已有内存大小  new\_data = pemalloc(HT\_SIZE\_EX(nSize, HT\_SIZE\_TO\_MASK(nSize)), GC\_FLAGS(ht) & IS\_ARRAY\_PERSISTENT);  ht->nTableMask = HT\_SIZE\_TO\_MASK(ht->nTableSize); // 重新计算 nTableMask  HT\_SET\_DATA\_ADDR(ht, new\_data); // ht->arData 指针指向新数据块  // 把旧数据复制过来  memcpy(ht->arData, old\_buckets, sizeof(Bucket) \* ht->nNumUsed);  pefree(old\_data, GC\_FLAGS(ht) & IS\_ARRAY\_PERSISTENT); // 释放旧数据块  zend\_hash\_rehash(ht); // 整理哈希表  } else { // 大小达到最大值  // 报错：内存分配时溢出  zend\_error\_noreturn(E\_ERROR, "Possible integer overflow in memory allocation (%u \* %zu + %zu)", ht->nTableSize \* 2, sizeof(Bucket) + sizeof(uint32\_t), sizeof(Bucket));  }  } |

如上所示，此函数会在利用率（有效元素数/占用元素数）小于32/33（约97%）时，调用zend\_hash\_rehash()函数整理哈希表，利用率大于等于97%并且元素数不超出限制时：1）重新分配数据块内存；2）把数据复制到新数据块里；3）再把旧的数据块删掉。没有调整原有内存大小，原因是数据块前半部分的哈希索引列表需要重新整理，整个数据块中的数据位置都要移动，工作量比较大，不如直接重新分配。

但zend\_hash\_rehash()函数中没有判断哈希表中已经分配的空间的使用情况，所以一般不直接调用它，而是调用它的衍生宏程序ZEND\_HASH\_IF\_FULL\_DO\_RESIZE()：

|  |
| --- |
| #define ZEND\_HASH\_IF\_FULL\_DO\_RESIZE(ht) \  if ((ht)->nNumUsed >= (ht)->nTableSize) { /\* 如果分配的空间都被占用 \*/ \  zend\_hash\_do\_resize(ht); /\* 调整大小，开辟新空间 \*/ \  } |

这里需要注意这3个参数的不同：分配元素数（ht->nTableSize）是指实例上分配了多少个元素的内存；占用元素数（ht->nNumUsed）里面包含了不连续的有效元素和无效元素，整理哈希表时，会把无效元素清理掉；有效元素数（ht->nNumOfElements）是指已经存放了有效数据的元素数。

### 计算哈希表元素数

由于哈希表的元素中可能有间接引用类型，所以计算哈希表有效元素数时，要排除那些引用目标无效的间接引用元素。计算哈希表元素数可使用zend\_array\_count()函数：

|  |
| --- |
| ZEND\_API uint32\_t zend\_array\_count(HashTable \*ht) {  uint32\_t num;  // 如果有【包含无效间接引用】标记  if (UNEXPECTED(HT\_FLAGS(ht) & HASH\_FLAG\_HAS\_EMPTY\_IND)) {  num = zend\_array\_recalc\_elements(ht); // 获取有效元素数量，排除无效间接引用  if (UNEXPECTED(ht->nNumOfElements == num)) { // 如果数量和有效元素数相等  HT\_FLAGS(ht) &= ~HASH\_FLAG\_HAS\_EMPTY\_IND; // 删除【含无效间接引用】标记  }  } else if (UNEXPECTED(ht == &EG(symbol\_table))) { // 如果是运行时全局变量中的符号表  num = zend\_array\_recalc\_elements(ht); // 取得有效元素数量，排除目标无效的间接引用  } else { // 其他情况直接取计算好的数量  num = zend\_hash\_num\_elements(ht);  }  return num;  } |

zend\_array\_recalc\_elements()函数用在有效元素中过滤掉无效的间接引用元素返回剩余的数量：

|  |
| --- |
| // 取得有效元素数量，排除目标无效的间接引用  static uint32\_t zend\_array\_recalc\_elements(HashTable \*ht) {  zval \*val;  uint32\_t num = ht->nNumOfElements; // 有效元素数  ZEND\_HASH\_MAP\_FOREACH\_VAL(ht, val) { // 遍历所有元素  if (Z\_TYPE\_P(val) == IS\_INDIRECT) { // 间接引用  // 如果引用目标无效  if (UNEXPECTED(Z\_TYPE\_P(Z\_INDIRECT\_P(val)) == IS\_UNDEF)) {  num--; // 减掉1个  }  }  } ZEND\_HASH\_FOREACH\_END();  return num; // 返回数量  } |

zend\_hash\_num\_elements()函数用于直接获得哈希表头信息中存储的有效元素数：

|  |
| --- |
| static uint32\_t zend\_hash\_num\_elements(const HashTable \*ht) {  return ht->nNumOfElements;  } |

### 哈希表转换成顺序数组

#### **1）哈希表导出顺序数组**

zend\_array\_to\_list()函数用于把哈希表的元素转存到新建的顺序数组中，转存时只按元素在内存中的顺序排列，原哈希表无变化：

|  |
| --- |
| // 把哈希表的元素添加到新建顺序数组中  ZEND\_API HashTable\* zend\_array\_to\_list(HashTable \*source) {  // 创建新数组，元素数和源数组相同  HashTable \*result = \_zend\_new\_array(zend\_hash\_num\_elements(source));  zend\_hash\_real\_init\_packed(result); // 初始化成顺序数组  ZEND\_HASH\_FILL\_PACKED(result) { // 准备开始填充顺序数组  zval \*entry;  ZEND\_HASH\_FOREACH\_VAL(source, entry) { // 遍历原哈希表  // 如果元素是引用类型 并且引用次数是1  if (UNEXPECTED(Z\_ISREF\_P(entry) && Z\_REFCOUNT\_P(entry) == 1)) {  entry = Z\_REFVAL\_P(entry); // 追踪引用目标  }  Z\_TRY\_ADDREF\_P(entry); // 元素增加引用次数  ZEND\_HASH\_FILL\_ADD(entry); // 填充到顺序数组  } ZEND\_HASH\_FOREACH\_END(); // 遍历源表结束  } ZEND\_HASH\_FILL\_END(); // 填充结束  return result; // 返回顺序数组指针  } |

关于引用类型以及Z\_ISREF\_P()、Z\_REFVAL\_P()、Z\_TRY\_ADDREF\_P()和Z\_REFCOUNT\_P()等宏程序的相关介绍参见“类型篇“。

#### **2）哈希表直接转成顺序数组**

zend\_hash\_to\_packed()函数用于把哈希表直接转成顺序数组：

|  |
| --- |
| ZEND\_API void zend\_hash\_to\_packed(HashTable \*ht) {  void \*new\_data, \*old\_data = HT\_GET\_DATA\_ADDR(ht); // 旧列表地址  Bucket \*src = ht->arData;  zval \*dst;  uint32\_t i;  // 分配内存，创建新数据块  new\_data = pemalloc(HT\_PACKED\_SIZE\_EX(ht->nTableSize, HT\_MIN\_MASK), GC\_FLAGS(ht) & IS\_ARRAY\_PERSISTENT);  // 添加标记，顺序数组，静态key  HT\_FLAGS(ht) |= HASH\_FLAG\_PACKED | HASH\_FLAG\_STATIC\_KEYS;  ht->nTableMask = HT\_MIN\_MASK; // 使用最小掩码 -2  HT\_SET\_DATA\_ADDR(ht, new\_data); // ht->arData 指向新数据块  HT\_HASH\_RESET\_PACKED(ht); // 顺序数组的哈希索引表里有2个元素，都设置成 -1  dst = ht->arPacked; // 新数据块开头  // 把 键值对的值依次添加到顺序数组中（这里没检查空和IS\_UNDEF）  for (i = 0; i < ht->nNumUsed; i++) {  ZVAL\_COPY\_VALUE(dst, &src->val); // 复制每个值  dst++;  src++;  }  pefree(old\_data, GC\_FLAGS(ht) & IS\_ARRAY\_PERSISTENT); // 删除旧数据块  } |

### 把哈希表截短到指定长度

zend\_hash\_discard()函数用于把哈希表截短到指定的长度：

|  |
| --- |
| ZEND\_API void zend\_hash\_discard(HashTable \*ht, uint32\_t nNumUsed) {  Bucket \*p, \*end, \*arData;  uint32\_t nIndex; // 索引号  arData = ht->arData; // 元素列表开头  p = arData + ht->nNumUsed; // 有效元素列表结尾  end = arData + nNumUsed; // 保留的列表结尾位置指针  ht->nNumUsed = nNumUsed; // 更新结尾位置  // 从右到左遍历从元素结尾到要求的结尾这一段  while (p != end) {  p--;  // 跳过无效元素  if (UNEXPECTED(Z\_TYPE(p->val) == IS\_UNDEF)) continue;  ht->nNumOfElements--; // 减少有效元素数  nIndex = p->h | ht->nTableMask; // 用哈希值计算哈希索引列表序号  // 在相同哈希值元素列表中删除这个元素，用它指向的元素做开头  HT\_HASH\_EX(arData, nIndex) = Z\_NEXT(p->val);  }  } |

### 快速添加元素

以下函数可以快速给哈希表添加元素，但它们在添加元素时都不检查键名是否重复，所以要在保证安全的前提下调用它们，在PHP源码中这几个函数都只有少量调用。

#### **\_zend\_hash\_append\_ex()函数**

\_zend\_hash\_append\_ex()函数用于快速给哈希表添加键值对，值为zval实例：

|  |
| --- |
| // 添加成对的键值，p1:哈希表，p2:键名，p3:是否把键名当成保留字处理  static zval \*\_zend\_hash\_append\_ex(HashTable \*ht, zend\_string \*key, zval \*zv, bool interned) {  uint32\_t idx = ht->nNumUsed++; // 使用数+1  uint32\_t nIndex;  Bucket \*p = ht->arData + idx; // 新元素指针  ZVAL\_COPY\_VALUE(&p->val, zv); // 复制 zval 给 Bucket  if (!interned && !ZSTR\_IS\_INTERNED(key)) { // 如果不是保留字调用，key 也不是保留字  HT\_FLAGS(ht) &= ~HASH\_FLAG\_STATIC\_KEYS; // 删除【静态键名】标记  zend\_string\_addref(key); // key 添加引用数  zend\_string\_hash\_val(key); // key 计算哈希值  }  p->key = key; // key 和 hash 复制给新元素p  p->h = ZSTR\_H(key);  nIndex = (uint32\_t)p->h | ht->nTableMask; // 使用取得哈希索引表中的元素序号  Z\_NEXT(p->val) = HT\_HASH(ht, nIndex); // 新元素的next指向旧的哈希索引表头元素  HT\_HASH(ht, nIndex) = HT\_IDX\_TO\_HASH(idx); // 把新元素放在哈希索引表开头  ht->nNumOfElements++; // 有效元素数 +1  return &p->val; // 返回新元素的值  } |

它有一个衍生函数\_zend\_hash\_append():

|  |
| --- |
| static zval \*\_zend\_hash\_append(HashTable \*ht, zend\_string \*key, zval \*zv) {  return \_zend\_hash\_append\_ex(ht, key, zv, 0);  } |

#### \_zend\_hash\_append\_ind()函数

\_zend\_hash\_append\_ind()函数用于快速给哈希表添加键值对，值为间接引用类型：

|  |
| --- |
| // 添加间接引用元素，p1:哈希表，p2:键名，p3:添加的指针  static void \_zend\_hash\_append\_ind(HashTable \*ht, zend\_string \*key, zval \*ptr) {  uint32\_t idx = ht->nNumUsed++; // 使用数+1  uint32\_t nIndex;  Bucket \*p = ht->arData + idx; // 新元素指针  ZVAL\_INDIRECT(&p->val, ptr); // 新元素值标记成间接引用，并把指针复制给它  if (!ZSTR\_IS\_INTERNED(key)) { // key 不是保留字串  HT\_FLAGS(ht) &= ~HASH\_FLAG\_STATIC\_KEYS; // 删除HASH\_FLAG\_STATIC\_KEYS标记  zend\_string\_addref(key); // key 添加引用数  zend\_string\_hash\_val(key); // key 计算哈希值  }  p->key = key; // 复制键名和哈希值  p->h = ZSTR\_H(key);  nIndex = (uint32\_t)p->h | ht->nTableMask; // 哈希索引表中的元素序号  Z\_NEXT(p->val) = HT\_HASH(ht, nIndex); // 新元素的next指向旧的哈希索引表头元素  HT\_HASH(ht, nIndex) = HT\_IDX\_TO\_HASH(idx); // 把新元素放在哈希索引表开头  ht->nNumOfElements++; // 有效元素数 +1  } |

ZVAL\_INDIRECT()宏程序的相关介绍参见“类型篇”。

#### \_zend\_hash\_append\_ptr\_ex()函数

\_zend\_hash\_append\_ptr\_ex()函数用于快速给哈希表添加键值对，值为指针：

|  |
| --- |
| // 哈希表中添加指针元素，p1:哈希表，p2:键名，p3:添加的指针，p4:是把key当成内置字符串处理  static zval \*\_zend\_hash\_append\_ptr\_ex(HashTable \*ht, zend\_string \*key, void \*ptr, bool interned) {  uint32\_t idx = ht->nNumUsed++; // 使用数+1  uint32\_t nIndex;  Bucket \*p = ht->arData + idx; // 新元素指针  ZVAL\_PTR(&p->val, ptr); // 新元素值类型为指针，并把指针复制给它  if (!interned && !ZSTR\_IS\_INTERNED(key)) { // 如果 不是保留字调用，key 也不是保留字  HT\_FLAGS(ht) &= ~HASH\_FLAG\_STATIC\_KEYS; // 删除HASH\_FLAG\_STATIC\_KEYS标记  zend\_string\_addref(key); // key 添加引用数  zend\_string\_hash\_val(key); // key 计算哈希值  }  p->key = key; // 复制键名和哈希值  p->h = ZSTR\_H(key);  nIndex = (uint32\_t)p->h | ht->nTableMask; // 哈希索引表中的元素序号  HT\_HASH(ht, nIndex) = HT\_IDX\_TO\_HASH(idx); // 新元素的next指向旧的哈希索引表头元素  ht->nNumOfElements++; // 有效元素数 +1  return &p->val; // 返回新元素的的zval对象指针  } |

ZVAL\_PTR()宏程序的相关介绍参见“类型篇”。

它有一个衍生函数\_zend\_hash\_append\_ptr()：

|  |
| --- |
| static zval \*\_zend\_hash\_append\_ptr(HashTable \*ht, zend\_string \*key, void \*ptr) {  return \_zend\_hash\_append\_ptr\_ex(ht, key, ptr, 0);  } |

### 给哈希表的元素更换键名（key）

zend\_hash\_set\_bucket\_key()函数用于给哈希表的元素更换键名，这是特别能体现哈希表的复杂性的操作。代码注释如下：

|  |
| --- |
| // 给哈希表的元素更换键名（key），p1:哈希表，p2:元素指针，p3:新键名  ZEND\_API zval\* zend\_hash\_set\_bucket\_key(HashTable \*ht, Bucket \*b, zend\_string \*key) {  uint32\_t nIndex;  uint32\_t idx, i;  Bucket \*p, \*arData;  (void)zend\_string\_hash\_val(key); // 计算哈希值  p = zend\_hash\_find\_bucket(ht, key); // 查找已有元素  if (UNEXPECTED(p)) { // 如果key已经被使用了  // 如果key属于正确的键值对，返回元素值。否则返回null，中断操作，因为不能有两个元素使用同一个key  return (p == b) ? &p->val : NULL;  }  // 如果key不在哈希表中，继续。key 不可以相同，但哈希值可以相同  if (!ZSTR\_IS\_INTERNED(key)) { // 如果不是保留字  zend\_string\_addref(key); // 添加key的引用次数  HT\_FLAGS(ht) &= ~HASH\_FLAG\_STATIC\_KEYS; // 删除静态键名标记  }  arData = ht->arData; // 元素列表开头  idx = HT\_IDX\_TO\_HASH(b - arData); // 计算此元素的序号  nIndex = b->h | ht->nTableMask; // 找到原哈希索引表的序号  i = HT\_HASH\_EX(arData, nIndex); // 找到原柚哈希值元素链表的头元素  if (i == idx) { // 如果两个序号相同，说明当前元素是链表的第一个  // 让哈希索引表元素指向下一个元素（把当前元素摘除）  HT\_HASH\_EX(arData, nIndex) = Z\_NEXT(b->val);  } else { // 当前元素不是链表的第一个  p = HT\_HASH\_TO\_BUCKET\_EX(arData, i); // 找到链表的头元素  while (Z\_NEXT(p->val) != idx) { // 如果不是指向b，继续找  i = Z\_NEXT(p->val); // 链表下一个元素的序号  p = HT\_HASH\_TO\_BUCKET\_EX(arData, i); // 链表下一个元素  }  // 让指向b的元素，指向b的下一个元素（把当前元素b从链表里摘除）  Z\_NEXT(p->val) = Z\_NEXT(b->val);  }  zend\_string\_release(b->key); // 删除当前元素b的key  idx = b - arData; // b的顺序号  b->key = key; // 新的key 和 哈希值  b->h = ZSTR\_H(key); // 计算新的哈希值  nIndex = b->h | ht->nTableMask; // 用新哈希值计算哈希索引表的元素序号    idx = HT\_IDX\_TO\_HASH(idx); // 新的顺序号 ，64位系统中不用转换  i = HT\_HASH\_EX(arData, nIndex); // 新的哈希值的 元素链表的 头元素序号  // 如果是 -1 或当前元素b的顺序号更大，把当前元素b串在链表开头  if (i == HT\_INVALID\_IDX || i < idx) {  Z\_NEXT(b->val) = i; // 当前元素b，指向链表的头元素  HT\_HASH\_EX(arData, nIndex) = idx; // 元素b的顺序号存进哈希索引表  } else { // 如果有比自己顺序大的有效元素。  // 找到可以插入的位置。 把自己插在p后面，保证链表按元素序号从大到小排列  p = HT\_HASH\_TO\_BUCKET\_EX(arData, i); // 原来的头元素  // 顺着链表找到最后一个【顺序号大于当前元素b的顺序号】的元素  while (Z\_NEXT(p->val) != HT\_INVALID\_IDX && Z\_NEXT(p->val) > idx) {  i = Z\_NEXT(p->val); // 不是要找的元素，取出下一个元素的序号  p = HT\_HASH\_TO\_BUCKET\_EX(arData, i); // 下一个元素  }  Z\_NEXT(b->val) = Z\_NEXT(p->val); // 当前元素b的next指p的next  Z\_NEXT(p->val) = idx; // p的next 指向当前元素b，这样就把当前元素b加到链表中了  }  return &b->val; // 返回元素的值  } |

# 四、内部指针（nInternalPointer）

在数组结构体中，内部指针元素（nInternalPointer）的类型是32位无符号整数（uint32\_t）,里面存放元素序号，它是用来迭代数组的辅助工具。迭代（iterate）与前文中的遍历（foreach）概念有些类似，但也有几点不同：

1）遍历通常在短时间内完成，数组被认为是静态的，在遍历过程中不会发生变化。迭代不关心完成时间，数组被认为是动态的，所以修改数组（尤其是删除元素）时需要考虑正在进行的迭代不能出错。

2）遍历有正序和倒序两种方式，每种方式进行过程中不能发生变化；迭代可以从任意一个位置随时掉转方向，向前或向后进行。

## 一）获取当前有效元素

内部指针的初始值为0。

### 获取当前元素的序号

zend\_hash\_get\_current\_pos()函数调用\_zend\_hash\_get\_current\_pos()进行操作用，于获取当前元素的序号：

|  |
| --- |
| static HashPosition \_zend\_hash\_get\_current\_pos(const HashTable \*ht) {  return \_zend\_hash\_get\_valid\_pos(ht, ht->nInternalPointer); // 获取有效序号  } |

\_zend\_hash\_get\_valid\_pos()函数用于查找给定位置后面的第一个有效元素序号：

|  |
| --- |
| static HashPosition \_zend\_hash\_get\_valid\_pos(const HashTable \*ht, HashPosition pos) {  if (HT\_IS\_PACKED(ht)) { // 如果是顺序数组  // 如果在有效范围内，当前元素无效，查检下一个  while (pos < ht->nNumUsed && Z\_ISUNDEF(ht->arPacked[pos])) {  pos++;  }  } else { 哈希表  // 如果在有效范围内，当前元素无效，查检下一个  while (pos < ht->nNumUsed && Z\_ISUNDEF(ht->arData[pos].val)) {  pos++;  }  }  return pos; // 返回有效元素序号或ht->nNumUsed  } |

如上所示，如果给出的位置后面没有有效元素，会返回ht->nNumUsed。

### 获取当前元素的值

zend\_hash\_get\_current\_data()函数用于获取数组的当前元素的值：

|  |
| --- |
| static zval\* zend\_hash\_get\_current\_data(HashTable \*ht) {  return zend\_hash\_get\_current\_data\_ex(ht, &ht->nInternalPointer);  } |

如上所示，它调用zend\_hash\_get\_current\_data\_ex()函数进行操作，并把内部指针位置作为参数传入。zend\_hash\_get\_current\_data\_ex()函数先获取正确的元素序号，再根据数组类型查找元素，代码如下：

|  |
| --- |
| ZEND\_API zval\* zend\_hash\_get\_current\_data\_ex(HashTable \*ht, HashPosition \*pos) {  uint32\_t idx;  Bucket \*p;  idx = \_zend\_hash\_get\_valid\_pos(ht, \*pos); // 右侧第一个有效位置  if (idx < ht->nNumUsed) { // 如果位置有效  if (HT\_IS\_PACKED(ht)) { // 如果是顺序数组  return &ht->arPacked[idx]; // 返回第 idx+1 个元素  }  p = ht->arData + idx; // 是哈希表，第 idx+1 个Bucket  return &p->val; // 返回键值对的值  } else {  return NULL; // 位置无效  }  } |

如上所示，如果给出的位置后面没有有效元素，会返回NULL。

### 获取当前元素值中的指针

zend\_hash\_get\_current\_data\_ptr()宏程序用于获取当前元素值中的指针：

|  |
| --- |
| #define zend\_hash\_get\_current\_data\_ptr(ht) \  zend\_hash\_get\_current\_data\_ptr\_ex(ht, &(ht)->nInternalPointer) |

zend\_hash\_get\_current\_data\_ptr\_ex()函数定义如下：

|  |
| --- |
| static void \*zend\_hash\_get\_current\_data\_ptr\_ex(HashTable \*ht, HashPosition \*pos) {  zval \*zv;  zv = zend\_hash\_get\_current\_data\_ex(ht, pos); // 获取右侧第一个有效元素的值  if (zv) {  return Z\_PTR\_P(zv); // 返回值中的指针  } else {  return NULL;  }  } |

### 获取当前元素的键名

zend\_hash\_get\_current\_key()函数用于获取当前元素的键名：

|  |
| --- |
| static int zend\_hash\_get\_current\_key(const HashTable \*ht, zend\_string \*\*str\_index,  zend\_ulong \*num\_index) {  return zend\_hash\_get\_current\_key\_ex(ht, str\_index, num\_index, &ht->nInternalPointer);  } |

zend\_hash\_get\_current\_key\_ex()函数定义如下：

|  |
| --- |
| // 获取右侧第一个有元素，根据类型，把键返回在 str\_index 或 num\_index 参数里返回  // p1:哈希表，p2:键名（返回值），p3:索引号（返回值），p4:当前位置  ZEND\_API int zend\_hash\_get\_current\_key\_ex(const HashTable \*ht, zend\_string \*\*str\_index, zend\_ulong \*num\_index, const HashPosition \*pos) {  uint32\_t idx;  Bucket \*p;  idx = \_zend\_hash\_get\_valid\_pos(ht, \*pos); // 当前位置  if (idx < ht->nNumUsed) { // 如果位置有效  if (HT\_IS\_PACKED(ht)) { // 顺序数组  \*num\_index = idx; // 引用返回顺序号作为索引值  return HASH\_KEY\_IS\_LONG; // 返回：键为整数类型  }  p = ht->arData + idx; // 哈希表, 取得键值对  if (p->key) { // 如果有键名  \*str\_index = p->key; // 引用返回键名，不返回哈希值  return HASH\_KEY\_IS\_STRING; // 返回：键为字符串类型  } else {  \*num\_index = p->h; // 引用返回哈希值，不返回键名  return HASH\_KEY\_IS\_LONG; // 返回：键为整数类型  }  }  return HASH\_KEY\_NON\_EXISTENT; // 返回查找失败  } |

如上所示，查找到有效元素后，通过引用返回键名和索引号，通过return语句返回类型。类型用3个常量来表示：

|  |  |  |
| --- | --- | --- |
| **常量名** | **值** | **说明** |
| HASH\_KEY\_IS\_STRING | 1 | 键名是字串类型 |
| HASH\_KEY\_IS\_LONG | 2 | 键名是整数类型 |
| HASH\_KEY\_NON\_EXISTENT | 3 | 查找有效元素失败 |

如果是顺序数组，会把元素序号通过num\_index参数返回。如果是哈希表，有键名时优先通过str\_index参数返回键名，不返回索引号；没有键名时通过num\_index参数返回索引号。

zend\_hash\_get\_current\_key\_type()函数与zend\_hash\_get\_current\_key()函数类似，但只返回类型，它调用zend\_hash\_get\_current\_key\_type\_ex()函数进行处理，函数代码如下：

|  |
| --- |
| static int zend\_hash\_get\_current\_key\_type(HashTable \*ht) {  return zend\_hash\_get\_current\_key\_type\_ex(ht, &ht->nInternalPointer);  } |

zend\_hash\_get\_current\_key\_type\_ex()函数与前文中的zend\_hash\_get\_current\_key\_ex()函数逻辑类似，但只返回类型。

zend\_hash\_get\_current\_key\_zval()函数与zend\_hash\_get\_current\_key()函数类似，但只通过key参数引用返回键名的值，不返回键名的类型，它调用zend\_hash\_get\_current\_key\_zval\_ex()函数进行处理，函数代码如下：

|  |
| --- |
| static void zend\_hash\_get\_current\_key\_zval(const HashTable \*ht, zval \*key){  zend\_hash\_get\_current\_key\_zval\_ex(ht, key, &ht->nInternalPointer);  } |

zend\_hash\_get\_current\_key\_type\_ex()函数与前文中的zend\_hash\_get\_current\_key\_ex()函数逻辑类似，但只通过key参数引用返回键名的值，不返回键名的类型。

## 二）内部指针向前（forward）移动1个元素

向前（forward）是指向着结尾的方向。zend\_hash\_move\_forward()函数用于把内部指针向结尾方向移动，跳过当前有效元素，指向下一个有效元素或ht->nNumUsed：

|  |
| --- |
| static zend\_result zend\_hash\_move\_forward(HashTable \*ht) {  // 这里传入的是指针的地址  return zend\_hash\_move\_forward\_ex(ht, &ht->nInternalPointer);  } |

实际操作在zend\_hash\_move\_forward\_ex()函数中进行：

|  |
| --- |
| ZEND\_API zend\_result zend\_hash\_move\_forward\_ex(HashTable \*ht, HashPosition \*pos) {  uint32\_t idx;  idx = \_zend\_hash\_get\_valid\_pos(ht, \*pos); // 当前有效位置，有效时指向当前元素  if (idx < ht->nNumUsed) { // 位置有效  if (HT\_IS\_PACKED(ht)) { // 如果是顺序数组  while (1) { // 向结尾方向查找  idx++; // 向结尾方向移动1个（先移动，后比较，第一次移动跳过了当前元素）  if (idx >= ht->nNumUsed) { // 到达有效元素结尾  \*pos = ht->nNumUsed; // 指针更新为有效元素数量，这时没有当前元素了  return SUCCESS; // 没有当前元素，但指针移动成功，返回成功  }  if (Z\_TYPE(ht->arPacked[idx]) != IS\_UNDEF) { // 找到有效元素  \*pos = idx; // 指针更新为有效元素的序号  return SUCCESS; // 成功，并且有当前元素  }  }  } else { // 哈希表  while (1) { // 向结尾方向查找  idx++; // 向结尾方向移动1个（先移动，后比较，第一次移动跳过了当前元素）  if (idx >= ht->nNumUsed) { // 到达有效元素结尾  \*pos = ht->nNumUsed; // 指针更新为有效元素数量，这时没有当前元素了  return SUCCESS; // 没有当前元素，但指针移动成功，返回成功 }  if (Z\_TYPE(ht->arData[idx].val) != IS\_UNDEF) { // 找到有效元素  \*pos = idx; // 指针更新为有效元素的位置  return SUCCESS; // 成功，并且有当前元素  }  }  }  } else { // 没有有效位置，指针也无法移动  return FAILURE; // 失败  }  } |

如上所示，业务逻辑的重点是向结尾方向移动，跳过1个有效元素，不论后面是否还有有效元素，都返回成功。如果后面还有有效元素，会把位置指针（pos）更新成这个元素的序号；如果后面没有有效元素，把指针更新成使用元素数（ht->nNumUsed），这样下一次向进行前移或后移操作时都会返回失败。

## 三）内部指针向后（backwards）移动1个元素

向后（backwords）是指向着开头的方向。zend\_hash\_move\_backwards()函数用于把内部指针向开头方向移动，指向最近一个有效元素：

|  |
| --- |
| static zend\_result zend\_hash\_move\_backwards(HashTable \*ht) {  return zend\_hash\_move\_backwards\_ex(ht, &ht->nInternalPointer);  } |

实际操作在zend\_hash\_move\_backwards\_ex()函数中进行：

|  |
| --- |
| ZEND\_API zend\_result zend\_hash\_move\_backwards\_ex(HashTable \*ht, HashPosition \*pos) {  uint32\_t idx = \*pos;  if (idx < ht->nNumUsed) { // 目标序号在已使用范围内  if (HT\_IS\_PACKED(ht)) { // 顺序数组  while (idx > 0) { // 向开头方向查找  idx--; // 指针向开头方向移动1个（先移动，后比较，返回有效元素序号）  if (Z\_TYPE(ht->arPacked[idx]) != IS\_UNDEF) { // 如果元素有效  \*pos = idx; // 使用这个序号  return SUCCESS; // 返回成功  }  }  } else { // 哈希表  while (idx > 0) { // 向开头方向查找  idx--; // 指针向开头方向移动1个（先移动，后比较，返回有效元素序号）  if (Z\_TYPE(ht->arData[idx].val) != IS\_UNDEF) { // 如果元素有效  \*pos = idx; // 使用这个序号  return SUCCESS; // 返回成功  }  }  }  \*pos = ht->nNumUsed; // 如果没找到有效元素，指向最后一个元素后面  return SUCCESS; // 返回成功  } else { // 没有有效位置，指针也无法移动  return FAILURE; // 返回失败  }  } |

如上所示，指针向开头方向移动，找到最近的一个有效元素，把指针更新成它的序号。如果直到开头都没找到有效元素，会把指针更新成使用元素数（ht->nNumUsed），这样下次调用前移或后移都会返回失败。

## 四）重置（reset）内部指针

如前文所述，向前移动或向后移动到达尽头后，指针都会被更新成使用元素数（ht->nNumUsed），这样下次调用前移或后移都会返回失败。如果想要开始下一轮迭代，需要先对指针进行重置操作。

zend\_hash\_internal\_pointer\_reset()函数用来重置内部指针：

|  |
| --- |
| static void zend\_hash\_internal\_pointer\_reset(HashTable \*ht) {  zend\_hash\_internal\_pointer\_reset\_ex(ht, &ht->nInternalPointer);  } |

实际操作是把指针位置设置为0，在zend\_hash\_internal\_pointer\_reset\_ex()函数中进行：

|  |
| --- |
| ZEND\_API void zend\_hash\_internal\_pointer\_reset\_ex(HashTable \*ht, HashPosition \*pos) {  \*pos = \_zend\_hash\_get\_valid\_pos(ht, 0); // 指针更新成开头第一个有效元素的序号  } |

## 五）内部指针置于结尾

如果想要倒序迭代数组，可以先调用zend\_hash\_internal\_pointer\_end()函数把指针置于数组结尾：

|  |
| --- |
| static void zend\_hash\_internal\_pointer\_end(HashTable \*ht) {  zend\_hash\_internal\_pointer\_end\_ex(ht, &ht->nInternalPointer);  } |

实际操作在zend\_hash\_internal\_pointer\_end\_ex()函数中进行：

|  |
| --- |
| ZEND\_API void zend\_hash\_internal\_pointer\_end\_ex(HashTable \*ht, HashPosition \*pos) {  uint32\_t idx;  idx = ht->nNumUsed; // 使用元素数  if (HT\_IS\_PACKED(ht)) { // 如果是顺序数组  while (idx > 0) { // 倒着查找  idx--; // 序号 -1  if (Z\_TYPE(ht->arPacked[idx]) != IS\_UNDEF) { // 如果元素有效  \*pos = idx; // 返回编号  return; // 结束  }  }  } else { // 哈希表  while (idx > 0) { // 倒着遍历  idx--; // 序号 -1  if (Z\_TYPE(ht->arData[idx].val) != IS\_UNDEF) { // 如果元素有效  \*pos = idx; // 返回编号  return; // 结束  }  }  }  \*pos = ht->nNumUsed; // 如果找不到，返回最后一个元素位置  } |

如上所示，从结尾开始倒序查找，找到第一个有效元素后把指针更新成它的序号。

## 六）检查结尾方向是否还有元素

zend\_hash\_has\_more\_elements()函数用于检查从指向位置向结尾方向，是否还有有效元素：

|  |
| --- |
| static zend\_result zend\_hash\_move\_forward(HashTable \*ht) {  return zend\_hash\_move\_forward\_ex(ht, &ht->nInternalPointer);  } |

实际操作在zend\_hash\_has\_more\_elements()函数中进行：

|  |
| --- |
| static zend\_result zend\_hash\_has\_more\_elements\_ex(HashTable \*ht, HashPosition \*pos) {  // 只有返回值为HASH\_KEY\_NON\_EXISTENT时算失败，其他都算成功  return (zend\_hash\_get\_current\_key\_type\_ex(ht, pos) == HASH\_KEY\_NON\_EXISTENT ? FAILURE : SUCCESS);  } |

zend\_hash\_get\_current\_key\_type\_ex()函数相关内容已经在前文中介绍过。

# 五、数组迭代器

数组迭代器（HashTableIterator），简称迭代器，是遍历数组用的辅助工具。当多个业务逻辑同时迭代一个数组时，需要分别记录它们的迭代位置。每个数组只有一个内部指针（nInternalPointer），无法满足需求，这时需要使用迭代器。迭代器相关结构体定义如下：

|  |
| --- |
| typedef uint32\_t HashPosition; // 元素序号，32位无符号整数  typedef struct \_HashTableIterator { // 迭代器一般都是运行时创建，放在 EG(ht\_iterators) 里  HashTable \*ht; // 数组指针  HashPosition pos; // 指向位置  } HashTableIterator; |

相关的宏程序比较简单：

|  |
| --- |
| // 获取数组的迭代器数量，数组的迭代器数量保存在 ht->u.v.nIteratorsCount 元素中  #define HT\_ITERATORS\_COUNT(ht) (ht)->u.v.nIteratorsCount  // 创建时检查数组的迭代器数量是否溢出（每个数组最多255个迭代器）  #define HT\_ITERATORS\_OVERFLOW(ht) (HT\_ITERATORS\_COUNT(ht) == 0xff)  // 检查数组是否有关联迭代器  #define HT\_HAS\_ITERATORS(ht) (HT\_ITERATORS\_COUNT(ht) != 0)  // 更新数组的迭代器数量  #define HT\_SET\_ITERATORS\_COUNT(ht, iters)  do { HT\_ITERATORS\_COUNT(ht) = (iters); } while (0)  // 数组的迭代器数量 +1  #define HT\_INC\_ITERATORS\_COUNT(ht) \  HT\_SET\_ITERATORS\_COUNT(ht, HT\_ITERATORS\_COUNT(ht) + 1)  // 数组的迭代器数量 -1  #define HT\_DEC\_ITERATORS\_COUNT(ht) \  HT\_SET\_ITERATORS\_COUNT(ht, HT\_ITERATORS\_COUNT(ht) - 1) |

## 一）创建迭代器

迭代器是在运行时使用的，在程序的解析和编译过程中不会用到它。

运行时全局变量的结构体为zend\_executor\_globals，在zend\_globals\_macros.h文件中定义，其中有几个元素与迭代器有关，定义如下：

|  |
| --- |
| uint32\_t ht\_iterators\_count; // 分配的迭代器数量，初始值16  uint32\_t ht\_iterators\_used; // 已使用的迭代器数量，初始值0  HashTableIterator \*ht\_iterators; // 迭代器列表指针  HashTableIterator ht\_iterators\_slots[16]; //初始迭代器列表，16个元素 |

这几个元素会随着运行时全局变量一同创建和销毁，可使用EG()宏程序来访问它们。ht\_iterators元素是迭代器列表的指针，ht\_iterators\_slots元素是初始迭代器列表，有16个元素。EG(ht\_iterators)指针一开始是指向初始迭代器列表EG(ht\_iterators\_slots)，当初始列表不够用时，再分配内存创建新列表。

## 二）给数组关联迭代器

给数组关联迭代器使用zend\_hash\_iterator\_add()函数：

|  |
| --- |
| // 给数组添加迭代器，p1:数组指针，p2:初始序号  ZEND\_API uint32\_t zend\_hash\_iterator\_add(HashTable \*ht, HashPosition pos) {  HashTableIterator \*iter = EG(ht\_iterators); // 迭代器列表指针  HashTableIterator \*end = iter + EG(ht\_iterators\_count); // 迭代器列表末尾  uint32\_t idx;  if (EXPECTED(!HT\_ITERATORS\_OVERFLOW(ht))) { // 如果数组的迭代器数量没溢出  HT\_INC\_ITERATORS\_COUNT(ht); // 数组的迭代器数量 +1  }  while (iter != end) { // 遍历所有迭代器，查找空闲的  if (iter->ht == NULL) { // 如果此是空闲的（没有关联数组），直接复用  iter->ht = ht; // 关联到当前数组  iter->pos = pos; // 迭代器记录当前位置  idx = iter - EG(ht\_iterators); // 迭代器的顺序号  // 如果大于使用过的最大编号  if (idx + 1 > EG(ht\_iterators\_used)) {  EG(ht\_iterators\_used) = idx + 1; // 更新使用过的最大编号  }  return idx; // 返回迭代器序号  }  iter++; // 下一个迭代器  }  **// 运行到这里说明没有闲置迭代器**  if (EG(ht\_iterators) == EG(ht\_iterators\_slots)) { // 如果使用的是初始迭代器列表  // 重新创建列表，16+8=24个迭代器  EG(ht\_iterators) = emalloc(sizeof(HashTableIterator) \* (EG(ht\_iterators\_count) + 8));  // 把原有迭代器列表复制过来  memcpy(EG(ht\_iterators), EG(ht\_iterators\_slots), sizeof(HashTableIterator) \*  EG(ht\_iterators\_count));  } else { // 如果使用的是单独分配的列表。调整列表大小，增加8个迭代器  EG(ht\_iterators) = erealloc(EG(ht\_iterators), sizeof(HashTableIterator) \*  (EG(ht\_iterators\_count) + 8));  }  iter = EG(ht\_iterators) + EG(ht\_iterators\_count); // 新创建的8个迭代器中的第一个  EG(ht\_iterators\_count) += 8; // 迭代器数量+8  iter->ht = ht; // 关联数组  iter->pos = pos; // 记录位置  memset(iter + 1, 0, sizeof(HashTableIterator) \* 7); // 清空后面的7个迭代器  idx = iter - EG(ht\_iterators); // 计算迭代器顺序号  EG(ht\_iterators\_used) = idx + 1; // 使用迭代器数量 +1  return idx; // 返回顺序号  } |

如上所示，创建过程分为几步：

1）检查当前数组迭代器数量是否溢出，如果没有溢出，给数量+1；

2）先在原列表中查找空闲迭代器，如果有空闲，直接使用；

3）如果没有空闲，给迭代器列表增加空间，每次增加8个元素。使用新分配的迭代器。

注意：HT\_INC\_ITERATORS\_COUNT()宏程序给当前数组增加迭代器数量，EG(ht\_iterators\_used) 里面存的是全局所有迭代器数量，两个数量作用不同，需要分别更新。

## 三）解除迭代器与数组的关联

### zend\_hash\_iterators\_remove()函数

zend\_hash\_iterators\_remove()函数用于在销毁数组时，批量移除此数组的迭代器：

|  |
| --- |
| static zend\_always\_inline void zend\_hash\_iterators\_remove(HashTable \*ht) {  if (UNEXPECTED(HT\_HAS\_ITERATORS(ht))) { // 如果此数组有关联迭代器  \_zend\_hash\_iterators\_remove(ht);  }  } |

删除操作在\_zend\_hash\_iterators\_remove()函数中完成，用此函数删除的迭代器，会被标记成异常状态，不可重用。函数代码如下：

|  |
| --- |
| #define HT\_POISONED\_PTR ((HashTable \*) (intptr\_t) -1) // 有毒的指针，指向无效的内存位置  static zend\_never\_inline void \_zend\_hash\_iterators\_remove(HashTable \*ht) {  HashTableIterator \*iter = EG(ht\_iterators); // 迭代器列表开头  HashTableIterator \*end = iter + EG(ht\_iterators\_used); // 迭代器列表结尾  while (iter != end) { // 遍历所有迭代器  if (iter->ht == ht) { // 如果是从关联到此数组  iter->ht = HT\_POISONED\_PTR; // 标记成有毒，HT\_POISONED\_PTR状态不可复用  }  iter++;  }  } |

如上所示，迭代器被标记成HT\_POISONED\_PTR状态后就不可以复用了，但迭代器并没有被删除。

### zend\_hash\_iterator\_del()函数

zend\_hash\_iterator\_del()函数用于解除指定序号的迭代器与数组关联：

|  |
| --- |
| ZEND\_API void zend\_hash\_iterator\_del(uint32\_t idx) {  HashTableIterator \*iter = EG(ht\_iterators) + idx; // 获取指定迭代器  // 如果迭代器有关联数组，迭代器指向的原数组没有被删除，原数组的迭代器数量没有溢出  if (EXPECTED(iter->ht) && EXPECTED(iter->ht != HT\_POISONED\_PTR)  && EXPECTED(!HT\_ITERATORS\_OVERFLOW(iter->ht))) {  HT\_DEC\_ITERATORS\_COUNT(iter->ht); // 减少数组的迭代器数量  }  iter->ht = NULL; // 清空迭代器的数组指针  if (idx == EG(ht\_iterators\_used) - 1) { // 如果正在删除最后一个已使用的迭代器  while (idx > 0 && EG(ht\_iterators)[idx - 1].ht == NULL) { // 遍历末尾所有空迭代器  idx--; // 最后一个有效序号递减  }  EG(ht\_iterators\_used) = idx; // 更新迭代器数量  }  } |

如上所示，删除迭代器只是把它清空并解除和数组的关联，迭代器还可以被复用。

## 四）更新迭代器指向位置

### 更新指定数组的迭代器

\_zend\_hash\_iterators\_update()函数在章节“顺序数组中删除元素”和“数组中删除元素”中都用到过。它用于更新指定数组的迭代器：

|  |
| --- |
| // 批量更新数组的迭代器位置，把指向某数组的from位置的迭代器全都改成指向to位置  ZEND\_API void \_zend\_hash\_iterators\_update(HashTable \*ht, HashPosition from, HashPosition to) {  HashTableIterator \*iter = EG(ht\_iterators); // 迭代器列表开头  HashTableIterator \*end = iter + EG(ht\_iterators\_used); // 迭代器列表结尾  while (iter != end) { // 遍历全部迭代器  if (iter->ht == ht && iter->pos == from) { // 如果是此数组的迭代器，并且位置正在from  iter->pos = to; // 把位置更新到 to  }  iter++; // 下一个  }  } |

### 为迭代器批量进行位移操作

zend\_hash\_iterators\_advance()函数用于为指定数组的所有迭代器批量进行位移操作：

|  |
| --- |
| // 所有属于当前数组的迭代器，位置后移 +step  ZEND\_API void zend\_hash\_iterators\_advance(HashTable \*ht, HashPosition step) {  HashTableIterator \*iter = EG(ht\_iterators);  HashTableIterator \*end = iter + EG(ht\_iterators\_used);  while (iter != end) { // 遍历所有迭代器  if (iter->ht == ht) { // 如果是属于当前数组的迭代器  iter->pos += step; // 位移  }  iter++; // 下一个迭代器  }  } |

## 五）查询正在迭代的最小序号

zend\_hash\_iterators\_lower\_pos()函数用来查找某一数组的所有迭代器，返回正在迭代的最小序号：

|  |
| --- |
| ZEND\_API HashPosition zend\_hash\_iterators\_lower\_pos(HashTable \*ht, HashPosition start) {  HashTableIterator \*iter = EG(ht\_iterators); // 迭代器列表开头  HashTableIterator \*end = iter + EG(ht\_iterators\_used); // 迭代器列表结尾  HashPosition res = ht->nNumUsed; // 默认返回数组的使用元素数  while (iter != end) { // 遍历全部迭代器  if (iter->ht == ht) { // 如果是此数组的迭代器  if (iter->pos >= start && iter->pos < res) { // 序号在有效范围内  res = iter->pos; // 使用更小的序号  }  }  iter++; // 下一个迭代器  }  return res;  } |

## 六）获取指定序号迭代器所指向位置

### zend\_hash\_iterator\_pos()函数

zend\_hash\_iterator\_pos()函数用于获取指定序号迭代器的指向位置，如果迭代器和数组不匹配，更新迭代器，关联到此数组：

|  |
| --- |
| ZEND\_API HashPosition zend\_hash\_iterator\_pos(uint32\_t idx, HashTable \*ht) {  HashTableIterator \*iter = EG(ht\_iterators) + idx; // 指定序号的迭代器  if (UNEXPECTED(iter->ht != ht)) { // 如果迭代器和数组不匹配  // 如果迭代器有关联数组 并且关联的数组没有被删除 并且原数组迭代器数量没有满  if (EXPECTED(iter->ht) && EXPECTED(iter->ht != HT\_POISONED\_PTR)  && EXPECTED(!HT\_ITERATORS\_OVERFLOW(iter->ht))) {  HT\_DEC\_ITERATORS\_COUNT(iter->ht); // 原数组迭代器数量 -1  }  if (EXPECTED(!HT\_ITERATORS\_OVERFLOW(ht))) { // 如果新数组迭代器数量没有满  HT\_INC\_ITERATORS\_COUNT(ht); // 迭代器数量 +1  }  iter->ht = ht; // 迭代器关联到新数组  iter->pos = \_zend\_hash\_get\_current\_pos(ht); // 位置取新数组迭代器中当前位置最小值  }  return iter->pos; // 返回位置（int类型）  } |

### zend\_hash\_iterator\_pos\_ex()函数

zend\_hash\_iterator\_pos\_ex()函数用于获取指定序号迭代器的指向位置，如果迭代器指向的数组不是当前数组，给数组创建副本，让迭代器关联到新副本，代码如下：

|  |
| --- |
| ZEND\_API HashPosition zend\_hash\_iterator\_pos\_ex(uint32\_t idx, zval \*array) {  HashTable \*ht = Z\_ARRVAL\_P(array); // 泛类型变量中获取数组  HashTableIterator \*iter = EG(ht\_iterators) + idx; // EG里第 idx 个迭代器  if (UNEXPECTED(iter->ht != ht)) { // 如果迭代器指向的数组不是当前数组  // 如果迭代器有指定数组 并且 指定的数组没有被删除  if (EXPECTED(iter->ht) && EXPECTED(iter->ht != HT\_POISONED\_PTR)  // 并且 当前数组没有溢出  && EXPECTED(!HT\_ITERATORS\_OVERFLOW(ht))) {  // 迭代器指向数组的 迭代器数量 -1  HT\_DEC\_ITERATORS\_COUNT(iter->ht);  }  SEPARATE\_ARRAY(array); // 数组创建副本，原数组减少引用次数，array指向新副本  ht = Z\_ARRVAL\_P(array); // 数组副本指针  if (EXPECTED(!HT\_ITERATORS\_OVERFLOW(ht))) { // 如果迭代器数量没有满  HT\_INC\_ITERATORS\_COUNT(ht); // 迭代器数量+1  }  iter->ht = ht; // 迭代器指向当前数组  iter->pos = \_zend\_hash\_get\_current\_pos(ht); // 迭代器位置  }  return iter->pos; // 返回位置（int类型）  } |

SEPARATE\_ARRAY()宏程序的介绍参见“为数组创建副本”章节。

## 七）使用迭代器迭代数组

使用迭代器迭代数组的过程与使用内部指针相同，调用的函数也相同，主要用到以下函数：

|  |  |
| --- | --- |
| **函数名** | **说明** |
| \_zend\_hash\_get\_valid\_pos() | 获取迭代位置后面的第一个有效元素序号 |
| zend\_hash\_get\_current\_data\_ex() | 获取迭代位置后面的第一个有效元素值 |
| zend\_hash\_get\_current\_data\_ptr\_ex() | 获取当前指向元素值中的指针 |
| zend\_hash\_get\_current\_key\_ex() | 获取当前指向元素的键名 |
| zend\_hash\_move\_forward\_ex() | 把迭代指针从当前位置向结尾方向移动 |
| zend\_hash\_move\_backwards\_ex() | 把迭代指针从当前位置向开头方向移动 |
| zend\_hash\_internal\_pointer\_reset\_ex() | 重置迭代器位置到0 |
| zend\_hash\_internal\_pointer\_end\_ex() | 把迭代器位置指向数组结尾 |
| zend\_hash\_has\_more\_elements() | 检查从迭代位置向结尾方向，是否还有有效元素 |

以上函数都在内部指针章节中介绍过。

# 六、兼容顺序数据和哈希表的操作

在操作数组时，通常不关心它是顺序数组还是哈希表，有一些函数和宏程序是对顺序数组和哈希表通用的，它们让操作更方便。

## 一）忽略类型的遍历操作

### 正序遍历顺序数组

正序遍历数组，主要用到ZEND\_HASH\_FOREACH\_FROM()宏程序，它从指定元素序号开始，一直遍历到元素列表结尾，代码如下：

|  |
| --- |
| #define ZEND\_HASH\_FOREACH\_FROM(\_ht, indirect, \_from) do { \  HashTable \*\_\_ht = (\_ht); /\* 哈希表指针 \*/ \  zend\_ulong \_\_h; /\* 哈希值 \*/ \  zend\_string \*\_\_key = NULL; /\* 键名，默认为NULL \*/ \  uint32\_t \_idx = (\_from); /\* 开始序号 \*/ \  size\_t \_size = ZEND\_HASH\_ELEMENT\_SIZE(\_\_ht); /\* 获取数组中单个元素大小 \*/ \  /\* 找到数组中第n个元素。p1:哈希表，p2:元素序号，p3:元素大小 \*/ \  zval \*\_\_z = ZEND\_HASH\_ELEMENT\_EX(\_\_ht, \_idx, \_size); \  uint32\_t \_count = \_\_ht->nNumUsed - \_idx; /\* 需要遍历的数量 = 使用数量-当前序号 \*/ \  for (;\_count > 0; \_count--) { /\* 遍历每个元素 \*/ \  zval \*\_z = \_\_z; /\* 临时变量 \*/ \  if (HT\_IS\_PACKED(\_\_ht)) { /\* 顺序数组 \*/ \  \_\_z++; /\* 下一个元素 \*/ \  \_\_h = \_idx; /\* 序号 作为 哈希值 \*/ \  \_idx++; /\* 序号+1 \*/ \  } else { /\* 哈希表 \*/ \  Bucket \*\_p = (Bucket\*)\_\_z; /\* 临时变量，转为bucket \*/ \  \_\_z = &(\_p + 1)->val; /\* Bucket 中的 zval \*/ \  \_\_h = \_p->h; /\* Bucket 中的 哈希值 \*/ \  \_\_key = \_p->key; /\* Bucket 中的键名 \*/ \  if (indirect && Z\_TYPE\_P(\_z) == IS\_INDIRECT) { /\* 如果值是间接引用 \*/ \  \_z = Z\_INDIRECT\_P(\_z); /\* 追踪间接引用 \*/ \  } \  } \  (void) \_\_h; (void) \_\_key; (void) \_idx; \  /\* 如果元素无效 或 间接引用目标无效，跳过 \*/ \  if (UNEXPECTED(Z\_TYPE\_P(\_z) == IS\_UNDEF)) continue; |

与遍历顺序数组和哈希表类似，ZEND\_HASH\_FOREACH\_FROM()宏程序**在遍历顺序数组时，\_\_key变量会一直是NULL**。用于调用的是几组衍生的宏程序。

#### **第一组，遍历数组并分别访问哈希值、键名和元素值**

本组共有7个宏程序：

|  |
| --- |
| // 辅助宏程序，从头开始遍历数组  #define ZEND\_HASH\_FOREACH(\_ht, indirect) ZEND\_HASH\_FOREACH\_FROM(\_ht, indirect, 0)  // 遍历数组，通过\_h、 \_key、 \_val变量访问每个哈希值、键名和元素值  #define ZEND\_HASH\_FOREACH\_KEY\_VAL(ht, \_h, \_key, \_val) \  ZEND\_HASH\_FOREACH(ht, 0); \  \_h = \_\_h; /\* 供外部访问 \*/ \  \_key = \_\_key; \  \_val = \_z;    // 遍历数组，通过\_h变量访问每个哈希值  #define ZEND\_HASH\_FOREACH\_NUM\_KEY(ht, \_h) // 逻辑类似    // 遍历数组，通过 \_key 变量访问每个键名  #define ZEND\_HASH\_FOREACH\_STR\_KEY(ht, \_key) // 逻辑类似    // 遍历数组，通过\_h、 \_key变量访问每个哈希值和键名  #define ZEND\_HASH\_FOREACH\_KEY(ht, \_h, \_key) // 逻辑类似  // 遍历数组，通过\_h、\_val变量访问每个哈希值和元素值  #define ZEND\_HASH\_FOREACH\_NUM\_KEY\_VAL(ht, \_h, \_val) // 逻辑类似    // 遍历数组，通过\_key、 \_val变量访问每个键名和元素值  #define ZEND\_HASH\_FOREACH\_STR\_KEY\_VAL(ht, \_key, \_val) // 逻辑类似 |

#### **第二组，遍历数组并分别访问哈希值、键名和元素值中的指针**

本组共有3个宏程序：

|  |
| --- |
| // 遍历数组，通过\_h、 \_key、 \_ptr变量访问每个哈希值、键名和元素值中的指针  #define ZEND\_HASH\_FOREACH\_KEY\_PTR(ht, \_h, \_key, \_ptr) \  ZEND\_HASH\_FOREACH(ht, 0); \  \_h = \_\_h; /\* 供外部访问 \*/ \  \_key = \_\_key; \  \_ptr = Z\_PTR\_P(\_z);  // 遍历数组，通过\_h、\_ptr变量访问每个哈希值和元素值中的指针  #define ZEND\_HASH\_FOREACH\_NUM\_KEY\_PTR(ht, \_h, \_ptr) // 逻辑类似  // 遍历数组，通过\_key、\_ptr变量访问每个键名和元素值中的指针  #define ZEND\_HASH\_FOREACH\_STR\_KEY\_PTR(ht, \_key, \_ptr) // 逻辑类似 |

#### **第三组，遍历数组并分别访问哈希值、键名和元素值，自动追踪间接引用**

本组共有3个宏程序：

|  |
| --- |
| // 遍历数组，自动追踪间接引用，通过\_h、 \_key、 \_val变量访问每个哈希值、键名和元素值  #define ZEND\_HASH\_FOREACH\_KEY\_VAL\_IND(ht, \_h, \_key, \_val) \  ZEND\_HASH\_FOREACH(ht, 1); \  \_h = \_\_h; /\* 供外部访问 \*/ \  \_key = \_\_key; \  \_val = \_z;  // 遍历数组，自动追踪间接引用，通过\_val变量访问每个元素值  #define ZEND\_HASH\_FOREACH\_VAL\_IND(ht, \_val) // 逻辑类似  // 遍历数组，自动追踪间接引用，通过\_key、 \_val变量访问每个键名和元素值  #define ZEND\_HASH\_FOREACH\_STR\_KEY\_VAL\_IND(ht, \_key, \_val) // 逻辑类似 |

#### **第四组，遍历数组并访问每个元素值**

本组共有3个宏程序：

|  |
| --- |
| // 遍历数组  #define \_ZEND\_HASH\_FOREACH\_VAL(\_ht) do { \  HashTable \*\_\_ht = (\_ht); /\* 哈希表指针 \*/ \  uint32\_t \_count = \_\_ht->nNumUsed; /\* 使用元素数 \*/ \  size\_t \_size = ZEND\_HASH\_ELEMENT\_SIZE(\_\_ht); /\* 获取数组中单个元素大小 \*/ \  zval \*\_z = \_\_ht->arPacked; /\* 元素列表开头 \*/ \  /\* 遍历每个元素，由于Bucket的开头是zval，所以这样遍历是可以兼容哈希表的 \*/ \  for (; \_count > 0; \_z = ZEND\_HASH\_NEXT\_ELEMENT(\_z, \_size), \_count--) { \  /\* 跳过未定义的元素 \*/ \  if (UNEXPECTED(Z\_TYPE\_P(\_z) == IS\_UNDEF)) continue;    // 遍历数组，通过val变量访问每个元素值  #define ZEND\_HASH\_FOREACH\_VAL(ht, \_val) \  \_ZEND\_HASH\_FOREACH\_VAL(ht);  \_val = \_z; /\* 供外部访问 \*/    // 遍历数组，通过val变量访问每个元素值中的指针  #define ZEND\_HASH\_FOREACH\_PTR(ht, \_ptr) \  \_ZEND\_HASH\_FOREACH\_VAL(ht); \  \_ptr = Z\_PTR\_P(\_z); /\* 供外部访问 \*/ |

如上所示，这组宏程序比较特别，不是像其他宏程序一样把顺序数组和哈希表分开处理，而是直接用zval指针遍历每个元素。由于哈希表Bucket元素的开头是一个zval元素，所以这个遍历方式是可以兼容顺序数组和哈希表的，只是在遍历哈希表时，只访问元素值，不访问键名和哈希值。

#### **第五组，从指定位置开始遍历数组并访问每个元素值**

本组共有2个宏程序：

|  |
| --- |
| // 从指定位置开始遍历数组，通过\_key、 \_val变量访问每个键名和元素值  #define ZEND\_HASH\_FOREACH\_STR\_KEY\_VAL\_FROM(ht, \_key, \_val, \_from) \  ZEND\_HASH\_FOREACH\_FROM(ht, 0, \_from); \  \_key = \_\_key; /\* 供外部访问 \*/ \  \_val = \_z;    // 从指定位置开始遍历数组，通过\_ptr变量访问每个元素值中的指针  #define ZEND\_HASH\_FOREACH\_PTR\_FROM(ht, \_ptr, \_from) // 逻辑类似 |

### 倒序遍历顺序数组

倒序遍历数组，主要用到ZEND\_HASH\_REVERSE\_FOREACH()宏程序：

|  |
| --- |
| // 倒序遍历数组。p1:数组，p2:是否自动追踪间接引用目标  #define ZEND\_HASH\_REVERSE\_FOREACH(\_ht, indirect) do { \  HashTable \*\_\_ht = (\_ht); \  uint32\_t \_idx = \_\_ht->nNumUsed; \  zval \*\_z; \  zend\_ulong \_\_h; \  zend\_string \*\_\_key = NULL; \  size\_t \_size = ZEND\_HASH\_ELEMENT\_SIZE(\_\_ht); /\* 获取数组中单个元素大小 \*/ \  /\* 找到顺序数组中第n个元素。p1:哈希表，p2:元素序号，p3:元素大小 \*/ \  zval \*\_\_z = ZEND\_HASH\_ELEMENT\_EX(\_\_ht, \_idx, \_size); \  for (;\_idx > 0; \_idx--) { /\* 倒序遍历 \*/ \  if (HT\_IS\_PACKED(\_\_ht)) { /\* 顺序数组 \*/ \  \_\_z--; /\* 上一个元素 \*/ \  \_z = \_\_z; /\* 临时变量 \*/ \  \_\_h = \_idx - 1; /\* 顺序号 作为 哈希值 \*/ \  } else { /\* 哈希表 \*/ \  Bucket \*\_p = (Bucket\*)\_\_z; /\* 元素转成 Bucket \*/ \  \_p--; /\* 上一个元素 \*/ \  \_\_z = &\_p->val; /\* 指向bucket中的zval实例 \*/ \  \_z = \_\_z; /\* 临时变量，用于检查间接引用 \*/ \  \_\_h = \_p->h; /\* 指向bucket中的哈希值 \*/ \  \_\_key = \_p->key; /\* 指向bucket中的键名 \*/ \  /\* 如果需要解引用 并且 碰到间接引用 \*/ \  if (indirect && Z\_TYPE\_P(\_z) == IS\_INDIRECT) { \  \_z = Z\_INDIRECT\_P(\_z); /\* 追踪引用目标 \*/ \  } \  } \  (void) \_\_h; (void) \_\_key; (void) \_\_z; \  /\* 如果元素无效 或 间接引用目标无效，跳过 \*/ \  if (UNEXPECTED(Z\_TYPE\_P(\_z) == IS\_UNDEF)) continue; |

与正序遍历类似，ZEND\_HASH\_REVERSE\_FOREACH()宏程序**在遍历顺序数组时，\_\_key变量会一直是NULL**。用于直接调用的是几组衍生的宏程序。

#### **第一组，倒序遍历数组并分别访问哈希值、键名和元素值**

本组共有6个宏程序：

|  |
| --- |
| // 倒序遍历数组，通过\_h、 \_key、 \_val变量访问每个哈希值、键名和元素值  #define ZEND\_HASH\_REVERSE\_FOREACH\_KEY\_VAL(ht, \_h, \_key, \_val) \  ZEND\_HASH\_REVERSE\_FOREACH(ht, 0); /\* 不追踪间接引用目标 \*/ \  \_h = \_\_h; /\* 供外部访问 \*/ \  \_key = \_\_key; \  \_val = \_z;    // 倒序遍历数组，通过\_h变量访问每个哈希值  #define ZEND\_HASH\_REVERSE\_FOREACH\_NUM\_KEY(ht, \_h) // 逻辑类似    // 倒序遍历数组，通过 \_key 变量访问每个键名  #define ZEND\_HASH\_REVERSE\_FOREACH\_STR\_KEY(ht, \_key) // 逻辑类似  // 倒序遍历数组，通过\_h、 \_key变量访问每个哈希值和键名  #define ZEND\_HASH\_REVERSE\_FOREACH\_KEY(ht, \_h, \_key) // 逻辑类似    // 倒序遍历数组，通过\_h、\_val变量访问每个哈希值和元素值  #define ZEND\_HASH\_REVERSE\_FOREACH\_NUM\_KEY\_VAL(ht, \_h, \_val) // 逻辑类似    // 倒序遍历数组，通过\_key、 \_val变量访问每个键名和元素值  #define ZEND\_HASH\_REVERSE\_FOREACH\_STR\_KEY\_VAL(ht, \_key, \_val) // 逻辑类似 |

#### **第二组，倒序遍历数组并分别访问哈希值、键名和元素值中的指针**

本组共有3个宏程序：

|  |
| --- |
| // 倒序遍历数组，通过\_h、 \_key、 \_ptr变量访问每个哈希值、键名和元素值中的指针  #define ZEND\_HASH\_REVERSE\_FOREACH\_KEY\_PTR(ht, \_h, \_key, \_ptr) // 逻辑类似  ZEND\_HASH\_REVERSE\_FOREACH(ht, 0); /\* 不追踪间接引用目标 \*/ \  \_h = \_\_h; /\* 供外部访问 \*/ \  \_key = \_\_key; \  \_ptr = Z\_PTR\_P(\_z);    // 倒序遍历数组，通过\_h、\_ptr变量访问每个哈希值和元素值中的指针  #define ZEND\_HASH\_REVERSE\_FOREACH\_NUM\_KEY\_PTR(ht, \_h, \_ptr) // 逻辑类似    // 倒序遍历数组，通过\_key、\_ptr变量访问每个键名和元素值中的指针  #define ZEND\_HASH\_REVERSE\_FOREACH\_STR\_KEY\_PTR(ht, \_key, \_ptr) // 逻辑类似 |

#### **第三组，倒序遍历数组并分别访问哈希值、键名和元素值，自动追踪间接引用**

本组共有3个宏程序：

|  |
| --- |
| // 倒序遍历数组，自动追踪间接引用，通过\_h、 \_key、 \_val变量访问每个哈希值、键名和元素值  #define ZEND\_HASH\_REVERSE\_FOREACH\_KEY\_VAL\_IND(ht, \_h, \_key, \_val) // 逻辑类似  ZEND\_HASH\_REVERSE\_FOREACH(ht, 1); /\* 追踪间接引用目标 \*/ \  \_h = \_\_h; \  \_key = \_\_key; \  \_val = \_z;    // 倒序遍历数组，自动追踪间接引用，通过\_val变量访问每个元素值  #define ZEND\_HASH\_REVERSE\_FOREACH\_VAL\_IND(ht, \_val) // 逻辑类似    // 倒序遍历数组，自动追踪间接引用，通过\_key、\_val变量访问每个键名和元素值  #define ZEND\_HASH\_REVERSE\_FOREACH\_STR\_KEY\_VAL\_IND(ht, \_key, \_val) // 逻辑类似 |

#### **第四组，倒序遍历数组并访问每个元素值**

本组共有3个宏程序：

|  |
| --- |
| // 倒序遍历数组  #define \_ZEND\_HASH\_REVERSE\_FOREACH\_VAL(\_ht) do { \  HashTable \*\_\_ht = (\_ht); /\* 哈希表指针 \*/ \  uint32\_t \_idx = \_\_ht->nNumUsed; /\* 从最大序号开始 \*/ \  /\* 获取数组中单个元素大小 \*/ \  size\_t \_size = ZEND\_HASH\_ELEMENT\_SIZE(\_\_ht); \  /\* 找到顺序数组中第n个元素。p1:哈希表，p2:元素序号，p3:元素大小 \*/ \  zval \*\_z = ZEND\_HASH\_ELEMENT\_EX(\_\_ht, \_idx, \_size); \  for (;\_idx > 0; \_idx--) { /\* 倒序遍历 \*/ \  /\* 找到数组的上一个元素，p1:元素列表指针，p2:元素大小 \*/ \  \_z = ZEND\_HASH\_PREV\_ELEMENT(\_z, \_size); \  /\* 跳过无效元素 \*/ \  if (UNEXPECTED(Z\_TYPE\_P(\_z) == IS\_UNDEF)) continue;    // 倒序遍历数组，通过val变量访问每个元素值  #define ZEND\_HASH\_REVERSE\_FOREACH\_VAL(ht, \_val) \  \_ZEND\_HASH\_REVERSE\_FOREACH\_VAL(ht); \  \_val = \_z; /\* 供外部访问 \*/    // 倒序遍历数组，通过val变量访问每个元素值中的指针  #define ZEND\_HASH\_REVERSE\_FOREACH\_PTR(ht, \_ptr) \  \_ZEND\_HASH\_REVERSE\_FOREACH\_VAL(ht); \  \_ptr = Z\_PTR\_P(\_z); /\* 供外部访问 \*/ |

如上所示，与正序遍历的第四组相似，这组宏程序比较特别，用zval指针遍历每个元素，在遍历哈希表时，只访问元素值，不访问键名和哈希值。

与遍历相关的很多宏程序在全部代码中都没有被调用过，不过仍然可以通过它们来理解程序设计思路，现在没有用到的函数和宏程序在未来可能会用到。

## 二）创建并初始化数组的数据块

zend\_hash\_real\_init()用于初始化数组，根据传入参数决定类型为顺序数组或哈希表，它调用zend\_hash\_real\_init\_ex()函数进行处理：

|  |
| --- |
| static void zend\_hash\_real\_init\_ex(HashTable \*ht, bool packed) {  if (packed) { // 顺序数组  zend\_hash\_real\_init\_packed\_ex(ht); // 初始化顺序数组  } else { // 哈希表  zend\_hash\_real\_init\_mixed\_ex(ht); // 初始化哈希表  }  } |

## 三）使用序号（或索引号）查找元素

### zend\_hash\_index\_find()函数

zend\_hash\_index\_find()函数用于使用序号（顺序数组）或索引号（哈希表）查找元素：

|  |
| --- |
| ZEND\_API zval\* zend\_hash\_index\_find(const HashTable \*ht, zend\_ulong h) {  Bucket \*p;  if (HT\_IS\_PACKED(ht)) { // 如果是顺序数组  if (h < ht->nNumUsed) { // 顺序号在有效范围内  zval \*zv = ht->arPacked + h; // 取得元素  if (Z\_TYPE\_P(zv) != IS\_UNDEF) { // 如果元素有效，返回元素  return zv;  }  }  return NULL; // 查询越界或元素无效，返回NULL  }  // 哈希表，使用索引号查找元素，返回元素值  p = zend\_hash\_index\_find\_bucket(ht, h);  return p ? &p->val : NULL; // 如果元素有效，返回元素值，否则返回NULL  } |

### ZEND\_HASH\_INDEX\_FIND()宏程序

ZEND\_HASH\_INDEX\_FIND()宏程序与zend\_hash\_index\_find()函数功能类似，但多了一个跳转功能，当查找失败时，会使用goto语进行跳转：

|  |
| --- |
| //与zend\_hash\_index\_find()函数的区别是多一个跳转点  #define ZEND\_HASH\_INDEX\_FIND(\_ht, \_h, \_ret, \_not\_found) do { \  if (EXPECTED(HT\_FLAGS(\_ht) & HASH\_FLAG\_PACKED)) { /\* 如果是顺序数组 \*/ \  if (EXPECTED((zend\_ulong)(\_h) < (zend\_ulong)(\_ht)->nNumUsed)) { /\* 顺序号有效 \*/ \  \_ret = &\_ht->arPacked[\_h]; /\* 获取这个元素 \*/ \  if (UNEXPECTED(Z\_TYPE\_P(\_ret) == IS\_UNDEF)) { /\* 元素无效 \*/ \  goto \_not\_found; /\* 跳转 \*/ \  } \  } else { /\* 顺序号越界 \*/ \  goto \_not\_found; /\* 跳转 \*/ \  } \  } else { /\* 哈希表 \*/ \  \_ret = \_zend\_hash\_index\_find(\_ht, \_h); /\* 用索引号查找 \*/ \  if (UNEXPECTED(\_ret == NULL)) { /\* 如果没找到 \*/ \  goto \_not\_found; /\* 跳转 \*/ \  } \  } \  } while (0) |

## 四）删除哈希表中的元素

### 通过序号（索引号）删除数组元素

通过序号删除元素，可使用zend\_hash\_index\_del()函数：

|  |
| --- |
| ZEND\_API zend\_result zend\_hash\_index\_del(HashTable \*ht, zend\_ulong h) {  uint32\_t nIndex;  uint32\_t idx;  Bucket \*p;  Bucket \*prev = NULL;  if (HT\_IS\_PACKED(ht)) { // 如果是顺序数组  if (h < ht->nNumUsed) { // 查找位置在已使用部分  zval \*zv = ht->arPacked + h; // 找到元素  if (Z\_TYPE\_P(zv) != IS\_UNDEF) { // 如果元素有效  // 通过序号删除顺序数组中的元素  \_zend\_hash\_packed\_del\_val(ht, HT\_IDX\_TO\_HASH(h), zv);  return SUCCESS; // 返回成功  }  // 失败情况1，元素无效  }  // 失败情况2：元素不在使用范围内  return FAILURE; // 以上两种情况返回失败  }    // 不是顺序数组，是哈希表：取得元素索引列表中的序号  nIndex = h | ht->nTableMask;  idx = HT\_HASH(ht, nIndex); // 取得元素序号  while (idx != HT\_INVALID\_IDX) { // 序号有效，检查此元素  p = HT\_HASH\_TO\_BUCKET(ht, idx); // 取得Bucket  if ((p->h == h) && (p->key == NULL)) { // 如果哈希值相同，并且没有键名  \_zend\_hash\_del\_el\_ex(ht, idx, p, prev); // 删除键值对的值  return SUCCESS; // 返回成功  }  prev = p; // 切换到当前元素  idx = Z\_NEXT(p->val); // 链表中下一个元素序号  }  return FAILURE; // 未找到要删除的元素，返回失败  } |

如上所示，在哈希表中，用索引号删除元素时，查询的是哈希值与序号匹配，但没有键名的元素。

Z\_TYPE\_P()宏程序的相关介绍参见“类型篇“。

## 五）清空或销毁数组

### 彻底销毁并释放数组

zend\_array\_destroy()函数用于彻底销毁并释放数组，在php源码中被大量使用。它分成几步操作：1）先把自身从自动垃圾回收列表中移除；2）再销毁数组的每个元素；3）最后释放zend\_array实例。代码注释如下：

|  |
| --- |
| ZEND\_API void zend\_array\_destroy(HashTable \*ht) {  GC\_REMOVE\_FROM\_BUFFER(ht); // 不需要再对这个哈希表进行自动垃圾回收  GC\_TYPE\_INFO(ht) = GC\_NULL // 类型更改成GC\_NULL  if (ht->nNumUsed) { // 如果有有效的元素  // 如果销毁器不是 ZVAL\_PTR\_DTOR  if (UNEXPECTED(ht->pDestructor != ZVAL\_PTR\_DTOR)) {  zend\_hash\_destroy(ht); // 销毁哈希表  goto free\_ht; // 跳到释放内存, 删除zend\_array实例  }  // 销毁器是 ZVAL\_PTR\_DTOR，zval\_ptr\_dtor()函数的别名  if (HT\_IS\_PACKED(ht)) { // 顺序数组  zval \*zv = ht->arPacked; // 遍历的开始位置  zval \*end = zv + ht->nNumUsed; // 遍历的结束位置  do { // 遍历所有元素  i\_zval\_ptr\_dtor(zv); // 尝试销毁目标对象  } while (++zv != end);  } else { // 哈希表  Bucket \*p = ht->arData; // 遍历的开始位置  Bucket \*end = p + ht->nNumUsed; // 遍历的结束位置  if (HT\_HAS\_STATIC\_KEYS\_ONLY(ht)) { // 如果只有静态键名  do { // 遍历所有元素  i\_zval\_ptr\_dtor(&p->val); // 尝试销毁目标对象  } while (++p != end);  } else if (HT\_IS\_WITHOUT\_HOLES(ht)) { // 有非静态键名，中间没有空元素  do { // 遍历所有元素  i\_zval\_ptr\_dtor(&p->val); // 尝试销毁目标对象  if (EXPECTED(p->key)) { // 如果有键名  zend\_string\_release\_ex(p->key, 0); // 释放键名  }  } while (++p != end);  } else { // 有动态键名，中间有空元素  do { // 遍历所有元素  if (EXPECTED(Z\_TYPE(p->val) != IS\_UNDEF)) { // 如果元素有效  i\_zval\_ptr\_dtor(&p->val); // 尝试销毁目标对象  if (EXPECTED(p->key)) { // 如果有键名  zend\_string\_release\_ex(p->key, 0); // 释放键名  }  }  } while (++p != end);  }  }  } else if (EXPECTED(HT\_FLAGS(ht) & HASH\_FLAG\_UNINITIALIZED)) { // 数组未初始化过  goto free\_ht; // 跳到释放内存, 删除zend\_array实例  }  efree(HT\_GET\_DATA\_ADDR(ht)); // 销毁数组的数据块  free\_ht:  zend\_hash\_iterators\_remove(ht); // 删除此数组的迭代器  FREE\_HASHTABLE(ht); // 释放数组, 删除zend\_array实例  } |

GC\_REMOVE\_FROM\_BUFFER()宏程序的介绍参见“垃圾回收篇”。

ZVAL\_PTR\_DTOR和i\_zval\_ptr\_dtor()函数的介绍在“类型篇”中。

FREE\_HASHTABLE()宏程序在zend\_alloc.h中定义，用于释放zend\_array实例：

|  |
| --- |
| #define FREE\_HASHTABLE(ht) efree\_size(ht, sizeof(HashTable)) |

zend\_array\_destroy()函数有两个衍生函数zend\_array\_release()和zend\_hash\_release()，用于释放引用数量为0的哈希表，不能对有IS\_ARRAY\_IMMUTABLE标记的数组进行操作：

|  |
| --- |
| static void zend\_hash\_release(zend\_array \*array) {  if (!(GC\_FLAGS(array) & IS\_ARRAY\_IMMUTABLE)) { // 如果不是不可更改  if (GC\_DELREF(array) == 0) { // 引用次数-1后，如果次数为0  zend\_hash\_destroy(array); // 删除 哈希表  // 释放内存，销毁数组实例（头信息），zend\_array\_release()函数没有这一行  pefree(array, GC\_FLAGS(array) & IS\_ARRAY\_PERSISTENT);  }  }  } |

zend\_hash\_release()函数与zend\_array\_release()函数的不同之处是：zend\_array\_release()函数不调用pefree()函数释放数组实例。

### 清空数组

清空数组可使用zend\_hash\_clean()函数，它的业务逻辑与zend\_hash\_destroy()函数相似，只是在销毁完所有元素后并没有释放数组的数据块，只是把数组复位到空置状态：

|  |
| --- |
| ZEND\_API void zend\_hash\_clean(HashTable \*ht) {  ... // 先销毁每个元素，逻辑与zend\_hash\_destroy()函数相同，然后把数组复位到空置状态  ht->nNumUsed = 0; // 使用元素为0  ht->nNumOfElements = 0; // 元素数为0  ht->nNextFreeElement = ZEND\_LONG\_MIN; // 无效序号  ht->nInternalPointer = 0; // 内部指针  } |

### 销毁数组的数据块

销毁数组的数据块，可调用zend\_hash\_destroy()函数，这个函数的逻辑并不复杂，但为了在循环中减少判断次数，它分成多种情况进行处理，这样虽然增加了代码行数，但提升了性能，这种优化在PHP内核代码中很常见。zend\_hash\_destroy()函数在PHP代码中被大量调用。

代码注释如下：

|  |
| --- |
| ZEND\_API void zend\_hash\_destroy(HashTable \*ht) {  if (ht->nNumUsed) { // 如果有有效元素  if (HT\_IS\_PACKED(ht)) { // 情况1 : 如果是顺序数组  if (ht->pDestructor) { // 有销毁器  zval \*zv = ht->arPacked; // 遍历开始位置，元素列表开头  zval \*end = zv + ht->nNumUsed; // 遍历结束位置，元素列表结尾  if (HT\_IS\_WITHOUT\_HOLES(ht)) { // 如果没有空元素  do { // 遍历销毁每一个元素  ht->pDestructor(zv);  } while (++zv != end);  } else { // 有空元素  do { // 遍历销毁每一个有效元素  if (EXPECTED(Z\_TYPE\_P(zv) != IS\_UNDEF)) { // 只销毁有效元素  ht->pDestructor(zv); // 调用销毁器进行销毁  }  } while (++zv != end);  }  }  zend\_hash\_iterators\_remove(ht); // 删除迭代器（解除关联）  } else { // 情况2：哈希表  Bucket \*p = ht->arData; // 遍历的开始位置  Bucket \*end = p + ht->nNumUsed; // 遍历的结束位置  if (ht->pDestructor) { // 分支2.1 : 有销毁器  if (HT\_HAS\_STATIC\_KEYS\_ONLY(ht)) { // 2.1.1如果只有静态键名  if (HT\_IS\_WITHOUT\_HOLES(ht)) { // 如果没有空元素  do { // 用销毁器删除每一个元素  ht->pDestructor(&p->val);  } while (++p != end);  } else { // 有空元素  do { // 遍历所有元素  if (EXPECTED(Z\_TYPE(p->val) != IS\_UNDEF)) { // 销毁有效元素  ht->pDestructor(&p->val);  }  } while (++p != end);  }  } else if (HT\_IS\_WITHOUT\_HOLES(ht)) { // 2.1.2 有动态键, 没有空元素  do { // 遍历所有元素  ht->pDestructor(&p->val); // 调用销毁器删除每一个 值  if (EXPECTED(p->key)) { // 如果有键名，删除它  zend\_string\_release(p->key);  }  } while (++p != end);  } else { // 2.1.3 有动态键, 有空元素  do { // 遍历所有元素  if (EXPECTED(Z\_TYPE(p->val) != IS\_UNDEF)) { // 只操作有效元素  ht->pDestructor(&p->val); // 调用销毁器，销毁每一个元素值  if (EXPECTED(p->key)) { // 如果有键名，删除键名  zend\_string\_release(p->key);  }  }  } while (++p != end);  }  } else { // 分支2.2 : 没有销毁器，只销毁所有动态键名  if (!HT\_HAS\_STATIC\_KEYS\_ONLY(ht)) { // 如果有动态键名  do { // 销毁每一个键名  if (EXPECTED(p->key)) { // 如果有键名  zend\_string\_release(p->key); // 删除键名  }  } while (++p != end);  }  }  zend\_hash\_iterators\_remove(ht); // 解除所有迭代器关联  }  // 如果没有初始化过  } else if (EXPECTED(HT\_FLAGS(ht) & HASH\_FLAG\_UNINITIALIZED)) {  return; //无需操作  }  // 销毁哈希表数据块  pefree(HT\_GET\_DATA\_ADDR(ht), GC\_FLAGS(ht) & IS\_ARRAY\_PERSISTENT);  } |

### 优雅地逆序销毁数组的数据列表

zend\_hash\_graceful\_reverse\_destroy()函数可以优雅地逆序销毁数组的数据列表，它在PHP代码中被少量调用：

|  |
| --- |
| ZEND\_API void zend\_hash\_graceful\_reverse\_destroy(HashTable \*ht) {  uint32\_t idx;  idx = ht->nNumUsed; // 占用元素数  if (HT\_IS\_PACKED(ht)) { // 顺序数组  zval \*zv = ht->arPacked + ht->nNumUsed; // 元素列表结尾  while (idx > 0) { // 序号到0为止  idx--; // 序号-1  zv--; // 指针左移  // 跳过无效元素  if (UNEXPECTED(Z\_TYPE\_P(zv) == IS\_UNDEF)) continue;  // 顺序数组删除指定 序号的元素，64位系统序号不用转换  \_zend\_hash\_packed\_del\_val(ht, HT\_IDX\_TO\_HASH(idx), zv);  }  } else { // 哈希表  Bucket \*p = ht->arData + ht->nNumUsed; // 元素列表结尾  while (idx > 0) { // 序号到0为止  idx--; // 序号-1  p--; // Bucket 指针左移  if (UNEXPECTED(Z\_TYPE(p->val) == IS\_UNDEF)) continue; // 跳过无效元素  // 哈希表中删除指定 序号的bucket，64位系统中序号不用转换  \_zend\_hash\_del\_el(ht, HT\_IDX\_TO\_HASH(idx), p);  }  }  if (!(HT\_FLAGS(ht) & HASH\_FLAG\_UNINITIALIZED)) { // 如果数组初始化过，释放数据块  pefree(HT\_GET\_DATA\_ADDR(ht), GC\_FLAGS(ht) & IS\_ARRAY\_PERSISTENT);  }  } |

## 六）批量处理数组元素

用于批量处理数组元素的函数有4个。

### zend\_hash\_apply()函数

zend\_hash\_apply()函数用于批量处理数组元素，它会对数组的每个元素应用传入的函数，并根据函数的返回结果进行后续操作：

|  |
| --- |
| // 对每个元素应用传入的函数。p1:数组，p2:函数  ZEND\_API void zend\_hash\_apply(HashTable \*ht, apply\_func\_t apply\_func) {  uint32\_t idx;  int result;  if (HT\_IS\_PACKED(ht)) { // 顺序数组  for (idx = 0; idx < ht->nNumUsed; idx++) { // 遍历每个元素  zval \*zv = ht->arPacked + idx; // 取出元素  if (UNEXPECTED(Z\_TYPE\_P(zv) == IS\_UNDEF)) continue; // 跳过无效元素  result = apply\_func(zv); // 对元素应用函数  if (result & ZEND\_HASH\_APPLY\_REMOVE) { // 如果返回结果要求删除元素  \_zend\_hash\_packed\_del\_val(ht, HT\_IDX\_TO\_HASH(idx), zv); // 删除元素  }  if (result & ZEND\_HASH\_APPLY\_STOP) { // 如果返回结果要求中断  break; // 中断  }  }  } else { // 哈希表  for (idx = 0; idx < ht->nNumUsed; idx++) { // 遍历每个元素  Bucket \*p = ht->arData + idx; // 取出元素  if (UNEXPECTED(Z\_TYPE(p->val) == IS\_UNDEF)) continue; // 跳过无效的  result = apply\_func(&p->val); // 对元素应用函数  if (result & ZEND\_HASH\_APPLY\_REMOVE) { // 如果返回结果要求删除元素  \_zend\_hash\_del\_el(ht, HT\_IDX\_TO\_HASH(idx), p); // 删除元素  }  if (result & ZEND\_HASH\_APPLY\_STOP) { // 如果返回结果要求中断  break; // 中断  }  }  }  } |

如上所示，业务逻辑比较简单，其中apply\_func参数的类型是函数原型apply\_func\_t，定义如下：

|  |
| --- |
| typedef int (\*apply\_func\_t)(zval \*pDest); |

此函数原型只接收一个参数，类型为zval实例指针。函数的返回值必须是以下3个常量之一：

|  |  |  |
| --- | --- | --- |
| **常量名** | **值** | **说明** |
| ZEND\_HASH\_APPLY\_KEEP | 0 | 指令：继续遍历 |
| ZEND\_HASH\_APPLY\_REMOVE | 1<<0 | 指令：删除元素并继续遍历 |
| ZEND\_HASH\_APPLY\_STOP | 1<<1 | 指令：中断遍历 |

### zend\_hash\_reverse\_apply()函数

如果需要倒序遍历哈希表，可以使用zend\_hash\_reverse\_apply()函数，它与zend\_hash\_apply()函数的业务逻辑很相似，只是遍历数组时采用倒序遍历。

### zend\_hash\_apply\_with\_argument()函数

zend\_hash\_apply\_with\_argument()函数与zend\_hash\_apply()函数类似，增加了一个附加参数：

|  |
| --- |
| ZEND\_API void zend\_hash\_apply\_with\_argument(HashTable \*ht,  apply\_func\_arg\_t apply\_func, void \*argument) |

它接收的函数原型是apply\_func\_arg\_t:

|  |
| --- |
| typedef int (\*apply\_func\_arg\_t)(zval \*pDest, void \*argument); |

此函数原型除数组元素外，只接收一个参数，类型为未知类型指针（void\*）,在遍历元素进行处理时，也会附带这个参数：

|  |
| --- |
| result = apply\_func(zv, argument); // 处理顺序数组时的调用  result = apply\_func(&p->val, argument); // 处理哈希表时的调用 |

### zend\_hash\_apply\_with\_arguments()函数

zend\_hash\_apply\_with\_arguments()函数与zend\_hash\_apply\_with\_argument()函数类似，不过可以传入任意多个附加函数：

|  |
| --- |
| ZEND\_API void zend\_hash\_apply\_with\_arguments(HashTable \*ht,  apply\_func\_args\_t apply\_func, int num\_args, ...) |

它接收的函数原型是apply\_func\_arg\_t：

|  |
| --- |
| // p1:元素值，p2:附加参数数，p3:附加参数列表，p4:键名  typedef int (\*apply\_func\_args\_t)(zval \*pDest, int num\_args, va\_list args,  zend\_hash\_key \*hash\_key); |

此函数原型除数组元素外，还接收3个参数：附加参数数量、附加参数列表、键名,在遍历元素进行处理时，也会附带这几个参数。其中键名的类型是zend\_hash\_key：

|  |
| --- |
| typedef struct \_zend\_hash\_key {  zend\_ulong h; // 哈希值  zend\_string \*key; // zend\_string实例指针  } zend\_hash\_key; |

在遍历元素进行处理时，也会附带这些参数。函数的逻辑框架与zend\_hash\_apply()函数相似，只是在处理每一个元素时有些不同，遍历顺序数组时对于每一个元素的调用逻辑如下：

|  |
| --- |
| // 顺序数组中对于每一个元素的调用逻辑  va\_start(args, num\_args); // 开始获取参数列表  hash\_key.h = idx; // 顺序数组元素序号  hash\_key.key = NULL;  // 调用用户函数  result = apply\_func(zv, num\_args, args, &hash\_key);  ... // 根据返回结果进行操作  va\_end(args); // 停止获取参数列表 |

遍历哈希表时对于每一个元素的调用逻辑如下：

|  |
| --- |
| // 哈希表中对于每一个元素的调用逻辑  va\_start(args, num\_args); // 开始获取参数列表  hash\_key.h = p->h; // Bucket元素哈希值  hash\_key.key = p->key; // Bucket元素键名  // 调用用户函数  result = apply\_func(&p->val, num\_args, args, &hash\_key);  ... // 根据返回结果进行操作  va\_end(args); // 停止获取参数列表...  va\_end(args); |

如上所示，对于顺序数组和哈希表，调用逻辑没有太大的差别，只是对hash\_key的处理不一样，顺序数组没有键名，只在hash\_key.h中存放元素的顺序号，hash\_key.key的值为NULL。

## 七）为数组创建副本

为数组创建副本使用zend\_array\_dup()函数，它会调用到zend\_array\_dup\_packed\_elements()、zend\_array\_dup\_value()、zend\_array\_dup\_elements()，zend\_array\_dup\_element()等函数，整个操作过程比较繁琐。

先从zend\_array\_dup()函数开始，代码如下：

|  |
| --- |
| ZEND\_API HashTable\* zend\_array\_dup(HashTable \*source) {  uint32\_t idx;  HashTable \*target;  ALLOC\_HASHTABLE(target); // 分配内存创建数组  GC\_SET\_REFCOUNT(target, 1); // 引用数为1  GC\_TYPE\_INFO(target) = GC\_ARRAY; // 类型为可回收的数组  target->pDestructor = ZVAL\_PTR\_DTOR; // 使用通用销毁器    if (source->nNumOfElements == 0) { // 情况1： 如果源数组里没有元素  **// 复制基本信息**  HT\_FLAGS(target) = HASH\_FLAG\_UNINITIALIZED; // 未初始化  target->nTableMask = HT\_MIN\_MASK; // -2  target->nNumUsed = 0; // 占用元素数 0  target->nNumOfElements = 0; // 有效元素数 0  target->nNextFreeElement = source->nNextFreeElement; // 复制下一个空闲元素序号  target->nInternalPointer = 0; // 内部指针指向 0  target->nTableSize = HT\_MIN\_SIZE; // 8  HT\_SET\_DATA\_ADDR(target, &uninitialized\_bucket); // h1->arData 指向无效位置  } else if (GC\_FLAGS(source) & IS\_ARRAY\_IMMUTABLE) { // 情况2： 源数组不可更改  ... **// 复制基本信息**  if (HT\_IS\_PACKED(source)) { // 源数组是顺序数组  // 分配内存，添加内存指针  HT\_SET\_DATA\_ADDR(target, emalloc(HT\_PACKED\_SIZE(target)));  target->nInternalPointer = source->nInternalPointer; // 复制内部指针  // 从源数据块复制数据  memcpy(HT\_GET\_DATA\_ADDR(target), HT\_GET\_DATA\_ADDR(source), HT\_PACKED\_USED\_SIZE(source));  } else { // 源数组是哈希表  HT\_SET\_DATA\_ADDR(target, emalloc(HT\_SIZE(target))); // 分配内存，添加内存指针  target->nInternalPointer = source->nInternalPointer; // 复制内部指针  // 从源数据块复制数据  memcpy(HT\_GET\_DATA\_ADDR(target), HT\_GET\_DATA\_ADDR(source),  HT\_USED\_SIZE(source));  }  } else if (HT\_IS\_PACKED(source)) { // 情况3：源数组是顺序数组  ... **// 复制基本信息**  // 分配内存，添加内存指针  HT\_SET\_DATA\_ADDR(target, emalloc(HT\_PACKED\_SIZE\_EX(target->nTableSize, HT\_MIN\_MASK)));  // 内部指针，如果源数组内部指针比占用元素数大，内部指针写 0  target->nInternalPointer =  (source->nInternalPointer < source->nNumUsed) ?  source->nInternalPointer : 0;  HT\_HASH\_RESET\_PACKED(target); // 顺序数组的哈希索引列表里有2个元素，都设置成 -1  if (HT\_IS\_WITHOUT\_HOLES(target)) { // 如果新数组没有无效元素  zend\_array\_dup\_packed\_elements(source, target, 0); // 复制元素，有无效元素  } else { // 新数组中有无效元素  zend\_array\_dup\_packed\_elements(source, target, 1); // 复制元素，没有无效元素  }  } else { // 情况4：源数组是哈希表  HT\_FLAGS(target) = HT\_FLAGS(source) & HASH\_FLAG\_MASK; // 复制标记和基本信息  target->nTableMask = source->nTableMask; // 复制掩码  target->nNextFreeElement = source->nNextFreeElement; // 下一个空闲元素序号  // 内部指针，如果源数组内部指针比元素数大，内部指针写 0  target->nInternalPointer =  (source->nInternalPointer < source->nNumUsed) ?  source->nInternalPointer : 0;  target->nTableSize = source->nTableSize; // 复制元素数  HT\_SET\_DATA\_ADDR(target, emalloc(HT\_SIZE(target))); // 分配内存，添加内存指针  HT\_HASH\_RESET(target); // 重置哈希索引列表，所有元素都设置成 -1  if (HT\_HAS\_STATIC\_KEYS\_ONLY(target)) { // 如果目标数组只包含静态键名  if (HT\_IS\_WITHOUT\_HOLES(source)) { // 源数组有无效元素  idx = zend\_array\_dup\_elements(source, target, 1, 0);// 复制元素，有无效元素  } else { // 源数组中没有无效元素  idx = zend\_array\_dup\_elements(source, target, 1, 1); // 复制元素，没有无效元素  }  } else { // 目标数组中有动态键名  if (HT\_IS\_WITHOUT\_HOLES(source)) { // 如果源数组有无效元素  idx = zend\_array\_dup\_elements(source, target, 0, 0);// 复制元素  } else { // 源数组中没有无效元素  idx = zend\_array\_dup\_elements(source, target, 0, 1);  }  }  target->nNumUsed = idx; // 占用元素数  target->nNumOfElements = idx; // 有效元素数  }  return target;  } |

如上所示，函数代码量比较大，但逻辑并不复杂，业务逻辑分支表如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **第一层条件** | **第二层条件** | **第三层条件** | **处理逻辑** |
| 源数组里没有元素 | | | 复制基本信息 |
| 源数组有元素，不可更改 | 源数组是顺序数组 | | 开辟内存创建新数据块，复制内存数据 |
| 源数组是哈希表 | |
| 源数组是普通顺序数组 | 新数组中没有无效元素 | | 复制基本信息和原素 |
| 新数组中有无效元素 | | 复制基本信息和原素 |
| 源数组是哈希表 | 如果目数组表只包含静态键名 | 源数组有无效元素 | 复制基本信息和原素 |
| 源数组中没有无效元素 | 复制基本信息和原素 |
| 目标数组中有动态键名 | 如果源数组有无效元素 | 复制基本信息和原素 |
| 源数组中没有无效元素 | 复制基本信息和原素 |

处理空数组和不可更改的数组时，逻辑比较简。这些看似复杂的业务逻辑主要是针对可更改的顺序数组和哈希表有不同的处理，调用路径如下：

|  |
| --- |
| // 处理可更改的顺序数组，函数调用路径如下  zend\_array\_dup()  ->zend\_array\_dup\_packed\_elements()  ->zend\_array\_dup\_value()  // 处理可更改的哈希表，函数调用路径如下  zend\_array\_dup()  ->zend\_array\_dup\_elements()  // 这里调用zend\_array\_dup\_value()函数后，还要复制哈希表元素的键名  ->zend\_array\_dup\_element()  ->zend\_array\_dup\_value() |

如上所示，顺序数组和哈希表复制元素的业务逻辑是比较相似的，但处理哈希表更麻烦一些。

zend\_array\_dup()函数调用到的相关宏程序如下：

|  |
| --- |
| // 计算顺序数组的 数据大小: 顺序数据大小 + 哈希->索引列表大小  #define HT\_PACKED\_SIZE(ht) HT\_PACKED\_SIZE\_EX((ht)->nTableSize, (ht)->nTableMask)    // 计算已使用大小， nTableMask 部分算已用  #define HT\_USED\_SIZE(ht) \  /\* 通过 nTableMask 计算 哈希->索引列表 大小 \*/ \  (HT\_HASH\_SIZE((ht)->nTableMask) + ((size\_t)(ht)->nNumUsed \* sizeof(Bucket)))  // 计算哈希表数据块大小：Bucket 列表大小 + 哈希->索引 列表大小。p1:哈希表  #define HT\_SIZE(ht) \  /\* 计算哈希表数据大小：Bucket 列表大小 + 哈希->索引 列表大小 \*/ \  HT\_SIZE\_EX((ht)->nTableSize, (ht)->nTableMask) |

### 复制顺序数组中的元素

zend\_array\_dup\_packed\_elements()函数用于复制顺序数组的全部元素：

|  |
| --- |
| static void zend\_array\_dup\_packed\_elements(HashTable \*source, HashTable \*target, bool with\_holes) {  zval \*p = source->arPacked; // 源数组开头  zval \*q = target->arPacked; // 新数组结尾  zval \*end = p + source->nNumUsed; // 源数组结尾  do { // 从头到尾遍历数组  if (!zend\_array\_dup\_value(source, target, p, q, 1, with\_holes)) { // 复制，如果失败  if (with\_holes) { // 并且数组本来就有空元素  ZVAL\_UNDEF(q); // 目标元素设置成 IS\_UNDEF  }  // 如果失败并且本来没有空元素，新数组中会跳过这个元素  }  p++; q++; // 新旧数组找下一个元素  } while (p != end);  } |

zend\_array\_dup\_value()函数用于复制数组的每个元素：

|  |
| --- |
| static bool zend\_array\_dup\_value(HashTable \*source, HashTable \*target, zval \*data, zval \*dest, bool packed, bool with\_holes) {  if (with\_holes) { // 如果有空元素  // 如果不是顺序数组 并且 源数据是 间接引用  if (!packed && Z\_TYPE\_INFO\_P(data) == IS\_INDIRECT) {  data = Z\_INDIRECT\_P(data); // 追踪引用对象，不会影响原始数据  }  if (UNEXPECTED(Z\_TYPE\_INFO\_P(data) == IS\_UNDEF)) { // 如果源数据无效  return 0; // 返回失败  }  } else if (!packed) { // 如果是哈希表  if (Z\_TYPE\_INFO\_P(data) == IS\_INDIRECT) { // 如果是间接引用  data = Z\_INDIRECT\_P(data);  if (UNEXPECTED(Z\_TYPE\_INFO\_P(data) == IS\_UNDEF)) { // 如果引用对象无效  return 0; // 返回失败  }  }  }  do {  if (Z\_OPT\_REFCOUNTED\_P(data)) { // 元素可引用计数  // 如果类型是引用 并且引用数量是1 并且（引用对象类型不是数组 或 引用对象不是源数组）  if (Z\_ISREF\_P(data) && Z\_REFCOUNT\_P(data) == 1 &&  (Z\_TYPE\_P(Z\_REFVAL\_P(data)) != IS\_ARRAY ||  Z\_ARRVAL\_P(Z\_REFVAL\_P(data)) != source)) {  data = Z\_REFVAL\_P(data); // 追踪引用目标  if (!Z\_OPT\_REFCOUNTED\_P(data)) { // 如果引用对象不可计数  break; // 如果类型是引用 并且引用数量是1 并且 引用对象不是源数组 并且 引用目标不可计数  }  }  // 如果data不是引用类型 或 引用次数不是1 或 （引用对象类型是数组 并且是 源数组）  Z\_ADDREF\_P(data); // 添加引用计数。 break 可以跳过这个。  }  } while (0); // while(0) 的目的是，为了用 break  ZVAL\_COPY\_VALUE(dest, data); // 复制元素zval  return 1;  } |

如上所示，这里有一个特殊的判断：如果（源数据可计数）并且（是引用类型）并且（引用数为1）并且（引用目标不是数组或引用目标不是源数据本身）并且（引用目标不可计数），则不给引用目标增加引用次数；其他情况都给源数据（或引用目标）增加引用次数。

Z\_ADDREF\_P()宏程序的相关介绍参见“类型篇”。

### 复制哈希表中的元素

zend\_array\_dup\_elements()函数用于复制哈希表中的全部元素：

|  |
| --- |
| static uint32\_t zend\_array\_dup\_elements(HashTable \*source, HashTable \*target, bool static\_keys, bool with\_holes) {  uint32\_t idx = 0;  Bucket \*p = source->arData; // 源数组元素列表开头  Bucket \*q = target->arData; // 新数组元素列表开头  Bucket \*end = p + source->nNumUsed; // 源组元素列表结尾  do {  // 只要有一个元素复制失败，就走里面的逻辑，因为两个数组的顺序号对不上了  if (!zend\_array\_dup\_element(source, target, idx, p, q, 0, static\_keys, with\_holes)) {  uint32\_t target\_idx = idx; // 源表的元素序号  idx++; p++; // 源表的元素和序号都向前走1个  while (p != end) { // 遍历源数据从 p到最后的元素  // 依次复制元素，如果复制成功  if (zend\_array\_dup\_element(source, target, target\_idx, p, q, 0, static\_keys, with\_holes)) {  if (source->nInternalPointer == idx) { // 如果碰到源数组内部指针指向的元素  // 目标数组内部指针也跟着指向这个元素  target->nInternalPointer = target\_idx;  }  // 复制成功后移动目标数组的序号和当前元素指针  target\_idx++; q++;  }  // 无论是否成功，移动源数组的序号和当前元素指针  idx++; p++;  }  return target\_idx; // 返回目标数组当前序号  }  // 如果没出错，序号和两个数组的元素指针都向前移  idx++; p++; q++;  } while (p != end);  // 如果在第一层while里所有元素都复制成功，返回当前顺序号  return idx;  } |

zend\_array\_dup\_element()函数用来复制单个哈希表元素：

|  |
| --- |
| static bool zend\_array\_dup\_element(HashTable \*source, HashTable \*target, uint32\_t idx, Bucket \*p, Bucket \*q, bool packed, bool static\_keys, bool with\_holes) {  // 如果复制元素失败，返回false  if (!zend\_array\_dup\_value(source, target, &p->val, &q->val, packed, with\_holes)) {  return 0;  }  if (!packed) { // 如果不是顺序数组  uint32\_t nIndex;  q->h = p->h; // 复制键名和哈希值  q->key = p->key;  if (!static\_keys && q->key) { // 如果数组有动态key，并且当前元素有key  zend\_string\_addref(q->key); // 增加引用次数，它会自动跳过保留字串  }  nIndex = q->h | target->nTableMask; // 取得哈希索引表中的序号  // 新元素指向同哈希值元素链表开头  Z\_NEXT(q->val) = HT\_HASH(target, nIndex);  // 新元素作为相同哈希值元素链表的开头  HT\_HASH(target, nIndex) = HT\_IDX\_TO\_HASH(idx);  }  return 1;  } |

### 衍生的宏程序

在zend\_types.h中定义了几个衍生的宏程序。

SEPARATE\_ARRAY()宏程序用于给zval实例指向的数组创建副本，并让zval实例指向新副本：

|  |
| --- |
| #define SEPARATE\_ARRAY(zv) do { \  zval \*\_\_zv = (zv); \  zend\_array \*\_arr = Z\_ARR\_P(\_\_zv); \  /\* 引用次数大于1才有必要创建分支 \*/ \  if (UNEXPECTED(GC\_REFCOUNT(\_arr) > 1)) { \  ZVAL\_ARR(\_\_zv, zend\_array\_dup(\_arr)); \  /\* 原实例减少引用次数 \*/ \  GC\_TRY\_DELREF(\_arr); \  } \  } while (0) |

SEPARATE\_ZVAL\_NOREF()宏程序用给zval实例指向的数组创建副本：

|  |
| --- |
| // 如果是数组，按需要创建副本，其他类型不操作  #define SEPARATE\_ZVAL\_NOREF(zv) do { \  zval \*\_zv = (zv); \  if (Z\_TYPE\_P(\_zv) == IS\_ARRAY) { \  SEPARATE\_ARRAY(\_zv); \  } \  } while (0) |

ZVAL\_DUP()宏程序用于复制zval实例并增加引用次数；如果zval实例指向的目标是数组，给数组创建副本并让zval实例指向新副本：

|  |
| --- |
| // 复制 zval 变量和附加信息。 如果类型是数组，创建副本  #define ZVAL\_DUP(z, v) do { \  zval \*\_z1 = (z); \  const zval \*\_z2 = (v); \  zend\_refcounted \*\_gc = Z\_COUNTED\_P(\_z2); /\* 变量指向的实例 \*/ \  uint32\_t \_t = Z\_TYPE\_INFO\_P(\_z2); /\* 取出类型信息 \*/ \  if ((\_t & Z\_TYPE\_MASK) == IS\_ARRAY) { /\* 如果源实例类型为数组 \*/ \  ZVAL\_ARR(\_z1, zend\_array\_dup((zend\_array\*)\_gc)); /\* 创建副本 \*/ \  } else { /\* 如果源实例类型不是数组 \*/ \  if (Z\_TYPE\_INFO\_REFCOUNTED(\_t)) { /\* 如果支持引用计数 \*/ \  GC\_ADDREF(\_gc); /\* 引用数+1 \*/ \  } \  ZVAL\_COPY\_VALUE\_EX(\_z1, \_z2, \_gc, \_t); /\* 复制zval \*/ \  } \  } while (0) |

ZVAL\_DUP()宏程序用于追踪引用类型的引用目标；如果zval实例指向的目标是数组，给数组创建副本并让zval实例指向新副本；如果zval实例指向引用数为1的引用器，释放引用器：

|  |
| --- |
| #define SEPARATE\_ZVAL(zv) do { \  zval \*\_zv = (zv); \  if (Z\_ISREF\_P(\_zv)) { /\* 如果是引用类型 \*/ \  zend\_reference \*\_r = Z\_REF\_P(\_zv); /\* 引用实例 \*/ \  ZVAL\_COPY\_VALUE(\_zv, &\_r->val); /\* 追踪引用对象 \*/ \  if (GC\_DELREF(\_r) == 0) { /\* 如果引用实例只有1次引用 \*/ \  efree\_size(\_r, sizeof(zend\_reference)); /\* 释放引用实例 \*/ \  } else if (Z\_OPT\_TYPE\_P(\_zv) == IS\_ARRAY) { /\* 否则（不可释放）：如果是数组 \*/ \  ZVAL\_ARR(\_zv, zend\_array\_dup(Z\_ARR\_P(\_zv))); /\* 创建副本，使用副本 \*/ \  break; /\* 完毕 \*/ \  } else if (Z\_OPT\_REFCOUNTED\_P(\_zv)) { /\* 否则（不可释放且不是数组）：如果可计数 \*/ \  Z\_ADDREF\_P(\_zv); /\* 增加引用次数，不是数组不用真的separate \*/ \  break; /\* 完毕 \*/ \  } \  } \  if (Z\_TYPE\_P(\_zv) == IS\_ARRAY) { /\* 数组这里单独处理 \*/ \  SEPARATE\_ARRAY(\_zv); /\* 数组创建副本 \*/ \  } \  } while (0) |

## 八）检查数组是否是连续列表

zend\_array\_is\_list()函数用来检查数组是否是连续列表，连续列表是指：空数组或数组元素从第一个开始使用，元素列表中间没有无效元素；如果是哈希表，还要求只能有索引元素，也就是所有元素不可以有键名。函数代码如下：

|  |
| --- |
| static bool zend\_array\_is\_list(zend\_array \*array) {  zend\_ulong expected\_idx = 0;  zend\_ulong num\_idx;  zend\_string\* str\_idx;  if (zend\_hash\_num\_elements(array) == 0) { // 空数组是连续列表  return 1; // 返回：是连续列表  }  if (HT\_IS\_PACKED(array)) { // 顺序数组  if (HT\_IS\_WITHOUT\_HOLES(array)) { // 没有无效元素  return 1; // 返回：是连续列表  }  ZEND\_HASH\_PACKED\_FOREACH\_KEY(array, num\_idx) { // 有无效元素，遍历  if (num\_idx != expected\_idx++) { // 如果有效元素顺序号不是从0开始或不连续  return 0; // 返回：不是连续列表  }  } ZEND\_HASH\_FOREACH\_END();  } else { // 哈希表  ZEND\_HASH\_MAP\_FOREACH\_KEY(array, num\_idx, str\_idx) { // 遍历  // 只检查索引元素，如果有索引号。索引号不是从0开始，或不连续  if (str\_idx != NULL || num\_idx != expected\_idx++) {  return 0; // 返回否  }  } ZEND\_HASH\_FOREACH\_END();  }  return 1; // 返回：是连续列表  } |

## 九）给数组增加空间

zend\_hash\_extend()函数用于给数组增加空间：

|  |
| --- |
| // 给数组增加空间。p1:数组，p2:元素数量，p3:是否顺序数组  ZEND\_API void zend\_hash\_extend(HashTable \*ht, uint32\_t nSize, bool packed) {  if (nSize == 0) return; // 如果目标大小是0 ，返回，什么也不做  // 情况1：如果数组没有初始化过  if (UNEXPECTED(HT\_FLAGS(ht) & HASH\_FLAG\_UNINITIALIZED)) {  if (nSize > ht->nTableSize) { // 如果目标大小 大于 现有大小  ht->nTableSize = zend\_hash\_check\_size(nSize); // 重新计算大小  }  zend\_hash\_real\_init(ht, packed); // 初始化  } else { // 情况2：已经初始化过  if (packed) { // 情况2.1：顺序数组  if (nSize > ht->nTableSize) { // 如果目标大小大于 现有大小  ht->nTableSize = zend\_hash\_check\_size(nSize); // 计算并应用新大小  // 调整原数块大小，并让arData指针指向新数据块（因为perealloc2()函数可能重新创建新数据块）  HT\_SET\_DATA\_ADDR(ht, perealloc2(HT\_GET\_DATA\_ADDR(ht), HT\_PACKED\_SIZE\_EX(ht->nTableSize, HT\_MIN\_MASK), HT\_PACKED\_USED\_SIZE(ht), GC\_FLAGS(ht) & IS\_ARRAY\_PERSISTENT));  }  } else { // 情况2.2：哈希表  if (nSize > ht->nTableSize) { // 如果目标大小大于 现有大小  void \*new\_data, \*old\_data = HT\_GET\_DATA\_ADDR(ht); // 取得旧数据块指针  Bucket \*old\_buckets = ht->arData; // 旧数据列表开头  nSize = zend\_hash\_check\_size(nSize); // 计算新大小  ht->nTableSize = nSize; // 应用新的大小  // 计算数据块大小，分配内存创建数据块  new\_data = pemalloc(HT\_SIZE\_EX(nSize, HT\_SIZE\_TO\_MASK(nSize)), GC\_FLAGS(ht) & IS\_ARRAY\_PERSISTENT);  ht->nTableMask = HT\_SIZE\_TO\_MASK(ht->nTableSize); // 重新计算掩码  HT\_SET\_DATA\_ADDR(ht, new\_data); // ht->arData指针指向新数据块  // 把旧元素列表复制过来，不要哈希索引表  memcpy(ht->arData, old\_buckets, sizeof(Bucket) \* ht->nNumUsed);  pefree(old\_data, GC\_FLAGS(ht) & IS\_ARRAY\_PERSISTENT); // 释放旧数据块  zend\_hash\_rehash(ht); // 整理哈希表  }  }  }  } |

## 十）计算最大值或最小值

zend\_hash\_minmax()函数用于获取数组的最大值或最小值：

|  |
| --- |
| // 获取数组的最大值或最小值。p1:数组，p2:比较函数，p3:为1时取最大，为0时取最小  ZEND\_API zval\* zend\_hash\_minmax(const HashTable \*ht, compare\_func\_t compar, uint32\_t flag) {  uint32\_t idx;  zval \*res;  if (ht->nNumOfElements == 0 ) { // 空数组  return NULL;  }  if (HT\_IS\_PACKED(ht)) { // 顺序数组  zval \*zv;  idx = 0;  while (1) { // 如果所有元素都是无效，返回null  if (idx == ht->nNumUsed) { // 到结尾  return NULL; // 返回NULL  }  // 找到第一个有效元素，跳出循环  if (Z\_TYPE(ht->arPacked[idx]) != IS\_UNDEF) break;  idx++; // 序号+1  }  res = ht->arPacked + idx; // 默认值是刚刚找到的第一个有效元素  for (; idx < ht->nNumUsed; idx++) { // 遍历后面的元素  zv = ht->arPacked + idx; // 取出每个元素  // 跳过无效元素  if (UNEXPECTED(Z\_TYPE\_P(zv) == IS\_UNDEF)) continue;  if (flag) { // flag不为0，取最大值  if (compar(res, zv) < 0) { // 当前元素更大  res = zv; // 使用当前元素  }  } else { // flag不为0，取最小值  if (compar(res, zv) > 0) { // 当前元素更小  res = zv; // 使用当前元素  }  }  }  } else { // 哈希表, 逻辑与上面类似  Bucket \*p;  idx = 0;  while (1) { // 如果所有元素都是未定义，返回null  if (idx == ht->nNumUsed) { // 到结尾  return NULL; // 返回NULL  }  // 找到第一个有效元素，跳出循环  if (Z\_TYPE(ht->arData[idx].val) != IS\_UNDEF) break;  idx++; // 序号+1  }  res = &ht->arData[idx].val; // 默认刚刚找到的第一个有效元素  for (; idx < ht->nNumUsed; idx++) { // 遍历后面的元素，取出最大或最小元素  p = ht->arData + idx; // 取出每个元素  if (UNEXPECTED(Z\_TYPE(p->val) == IS\_UNDEF)) continue; // 跳过无效元素  if (flag) { // flag不为0，取最大值  if (compar(res, &p->val) < 0) { // 当前元素更大  res = &p->val; // 使用当前元素  }  } else { // flag不为0，取最小值  if (compar(res, &p->val) > 0) { // 当前元素更小  res = &p->val; // 使用当前元素  }  }  }  }  return res; // 返回查找的值  } |

## 十一）把数组元素复制到新数组

zend\_hash\_copy()函数用于把一个数组的元素复制到另一个数组：

|  |
| --- |
| ZEND\_API void zend\_hash\_copy(HashTable \*target, HashTable \*source, copy\_ctor\_func\_t pCopyConstructor) {  uint32\_t idx;  zval \*new\_entry, \*data;  if (HT\_IS\_PACKED(source)) { // 顺序数组  for (idx = 0; idx < source->nNumUsed; idx++) { // 遍历所有元素  zval \*zv = source->arPacked + idx; // 取得元素  if (UNEXPECTED(Z\_TYPE\_P(zv) == IS\_UNDEF)) continue; // 跳过无效元素  // 找到旧元素时更新，找不到旧元素时添加  new\_entry = zend\_hash\_index\_update(target, idx, zv);  if (pCopyConstructor) { // 如果复制后需要构造元素  pCopyConstructor(new\_entry); // 调用构造方法构造新元素  }  }  return;  }  // 哈希表  for (idx = 0; idx < source->nNumUsed; idx++) { // 遍历所有元素  Bucket \*p = source->arData + idx; // 取出每个元素  if (UNEXPECTED(Z\_TYPE(p->val) == IS\_UNDEF)) continue; // 跳过无效  data = &p->val; // bucket中的元素指针  if (Z\_TYPE\_P(data) == IS\_INDIRECT) { // 哈希表中可能有间接引用，如果是间接引用  data = Z\_INDIRECT\_P(data); // 追踪到目标对象  if (UNEXPECTED(Z\_TYPE\_P(data) == IS\_UNDEF)) { // 跳过无效  continue;  }  }  if (p->key) { // 如果有键名  // 找到旧元素时更新，找不到时添加  new\_entry = zend\_hash\_update(target, p->key, data);  } else { // 没有键名只有哈希值  // 按索引号操作，找到旧元素时更新，找不到时添加  new\_entry = zend\_hash\_index\_update(target, p->h, data);  }  if (pCopyConstructor) { // 如果复制后需要构造元素  pCopyConstructor(new\_entry); // 调用构造方法构造新元素  }  }  } |

## 十二）合并数组

### zend\_hash\_merge()函数

合并数组可调用zend\_hash\_merge()函数：

|  |
| --- |
| // 合并两个数组。p1:目标数组，p2:源数组，p3:复制元素时调用的构造方法，p4:是否覆盖元素  ZEND\_API void zend\_hash\_merge(HashTable \*target, HashTable \*source, copy\_ctor\_func\_t pCopyConstructor, bool overwrite) |

函数的代码比较长，但逻辑并不复杂，主要是遍历源数组并尝试把每个元素插入到新数组中。业务逻辑可归纳成几种情况：

|  |  |  |
| --- | --- | --- |
|  | **覆盖已有元素** | **跳过已有元素** |
| **顺序数组**  **元素** | 调用zend\_hash\_index\_update()函数 | 调用zend\_hash\_index\_add()函数 |
| **哈希表中有键名（key）的元素** | 调用\_zend\_hash\_add\_or\_update\_i()函数  操作标记为：HASH\_UPDATE | HASH\_UPDATE\_INDIRECT | 调用\_zend\_hash\_add\_or\_update\_i()函数  操作标记为：HASH\_ADD | HASH\_UPDATE\_INDIRECT |
| **哈希表中有索引号的元素（没有键名key）** | 调用zend\_hash\_index\_update()函数 | 调用zend\_hash\_index\_add()函数 |

要求覆盖已有元素时，顺序数组的操作如下：

|  |
| --- |
| for (idx = 0; idx < source->nNumUsed; idx++) { // 遍历所有可用元素  s = source->arPacked + idx; // 取出元素  if (UNEXPECTED(Z\_TYPE\_P(s) == IS\_UNDEF)) { // 跳过未定义  continue;  }  t = zend\_hash\_index\_update(target, idx, s); // 找到旧元素时更新,找不到旧元素时添加  if (pCopyConstructor) { // 添加成功后，如果有构造方法  pCopyConstructor(t); // 调用构造方法  }  } |

要求覆盖已有元素时，哈希表的操作如下：

|  |
| --- |
| for (idx = 0; idx < source->nNumUsed; idx++) { // 源数组是哈希表  p = source->arData + idx; // 每个键值对  s = &p->val; // 值  if (UNEXPECTED(Z\_TYPE\_P(s) == IS\_INDIRECT)) { // 如果是间接引用  s = Z\_INDIRECT\_P(s); // 追踪到被引用对象  }  if (UNEXPECTED(Z\_TYPE\_P(s) == IS\_UNDEF)) { // 跳过未定义  continue;  }  if (p->key) { // 如果有键名（key）  **// 通过键名更新目标数组，（不覆盖时，标记为HASH\_ADD | HASH\_UPDATE\_INDIRECT）**  t = \_zend\_hash\_add\_or\_update\_i(target, p->key, s, HASH\_UPDATE | HASH\_UPDATE\_INDIRECT);  if (pCopyConstructor) { // 添加成功后，如果有构造方法  pCopyConstructor(t); // 调用构造方法  }  } else { // 没有key，按索引号操作  t = zend\_hash\_index\_update(target, p->h, s); // 找到旧元素更新元素，找不到时添加元素  if (pCopyConstructor) { // 添加成功后，如果有构造方法  pCopyConstructor(t); // 调用构造方法  }  }  } |

要求跳过已有元素（overwrite参数值为0）时，处理逻辑类似。需要注意的是在处理目标哈希表中无效的间接引用元素时，使用了如下处理逻辑：

|  |
| --- |
| t = \_zend\_hash\_add\_or\_update\_i(target, p->key, s, HASH\_ADD | HASH\_UPDATE\_INDIRECT); |

如上所示，这是唯一一个地方使用HASH\_ADD和HASH\_UPDATE\_INDIRECT标记，调用\_zend\_hash\_add\_or\_update\_i()函数，这样碰到目标哈希表中的**无效间接引用元素**时，会用源数组中的元素覆盖它。

### zend\_hash\_merge\_ex()函数

如果在合并数组时，需要使用自己定义的检验方法来检查每个元素是否需要合并，可以调用 zend\_hash\_merge\_ex()函数：

|  |
| --- |
| // 合并数组。p1:目标数组，p2:源数组，p3:复制元素构造方法，p4:合并检验函数，p5:合并检验用的 参数  ZEND\_API void zend\_hash\_merge\_ex(HashTable \*target, HashTable \*source, copy\_ctor\_func\_t pCopyConstructor, merge\_checker\_func\_t pMergeSource, void \*pParam){  uint32\_t idx;  Bucket \*p;  zval \*t;  for (idx = 0; idx < source->nNumUsed; idx++) { // 遍历源数组  p = source->arData + idx; // 获取每个元素  if (UNEXPECTED(Z\_TYPE(p->val) == IS\_UNDEF)) continue; // 跳过无效元素  // 调用自定义方法检查是否可以 合并，如果可以  if (zend\_hash\_replace\_checker\_wrapper(target, &p->val, p->h, p->key, pParam, pMergeSource)) {  // 更新目标数组有旧数据时覆盖，没有时插入新元素  t = zend\_hash\_update(target, p->key, &p->val);  if (pCopyConstructor) { // 如果有构造方法  pCopyConstructor(t); // 对新元素调用构造方法  }  }  }  } |

zend\_hash\_replace\_checker\_wrapper()函数用于调用自定义函数检查每个元素是否可以合并：

|  |
| --- |
| // 调用自定义方法检查是否可以合并  // p1:数组，p2:源数组，p3:哈希值，p4:key（zend\_string），p5:参数指针，p6:自定义检验方法  static bool zend\_hash\_replace\_checker\_wrapper(HashTable \*target, zval \*source\_data, zend\_ulong h, zend\_string \*key, void \*pParam, merge\_checker\_func\_t merge\_checker\_func) {  zend\_hash\_key hash\_key; // 临时 hash\_key 实例  hash\_key.h = h; // 哈希值  hash\_key.key = key; // key  // 调用自定义函数检查可否合并，p1:目标数组，p2:源数据，p3: hash\_key实例，p4:参数指针  return merge\_checker\_func(target, source\_data, &hash\_key, pParam);  } |

自定义函数需要符合merge\_checker\_func\_t函数原型：

|  |
| --- |
| // 函数原型，检查数组元素可否合并，p1:数组，p2:源数据，p3: hash\_key实例，p4:参数指针  typedef bool (\*merge\_checker\_func\_t)(HashTable \*target\_ht, zval \*source\_data, zend\_hash\_key \*hash\_key, void \*pParam); |

## 十三）给数组排序

zend\_hash\_sort()函数用于给数组的排序：

|  |
| --- |
| static void zend\_hash\_sort(HashTable \*ht, bucket\_compare\_func\_t  compare\_func, zend\_bool renumber) {  // 此处传入的排序函数为zend\_sort()函数  zend\_hash\_sort\_ex(ht, zend\_sort, compare\_func, renumber);  } |

zend\_sort()函数使用插入排序和快速排序，源码注释在附录中。

它调用zend\_hash\_sort\_ex()函数进行处理，过程比较复杂，主要是因为要处理重新编号：

|  |
| --- |
| // 使用给定的方法比较，排序。如果需要重排序号，转成顺序数组，否则转成哈希表。 p1:数组，p2:排序函数，p3:比较函数，p4:是否需要重新编号  ZEND\_API void zend\_hash\_sort\_ex(HashTable \*ht, sort\_func\_t sort, bucket\_compare\_func\_t compar, bool renumber) {  Bucket \*p;  uint32\_t i, j;  // 步骤1：跳过无需处理的情况（0个元素 或 1个元素并且不重新编号）  if (!(ht->nNumOfElements>1) && !(renumber && ht->nNumOfElements>0)) {  return; // 中断  }  // 步骤2：顺序数组先转成哈希表  if (HT\_IS\_PACKED(ht)) { // 顺序数组  zend\_hash\_packed\_to\_hash(ht); // 先转成哈希表  }  // 步骤3：整理数组，整理后不再有无效元素，再记录原始序号  if (HT\_IS\_WITHOUT\_HOLES(ht)) { // 如果没有空元素  for (i = 0; i < ht->nNumUsed; i++) { // 遍历每个元素  Z\_EXTRA(ht->arData[i].val) = i; // 把原顺序号存放在 值的 zval.u2.extra 里  }  } else { // 有空元素  // 删除空元素并保存原顺序号，经过这个操作哈希索引列表就无效了  for (j = 0, i = 0; j < ht->nNumUsed; j++) { // 遍历所有的元素  p = ht->arData + j; // 取出每个元素  if (UNEXPECTED(Z\_TYPE(p->val) == IS\_UNDEF)) continue; // 跳过无效元素  if (i != j) { // 如果出现了（任意多个）无效元素  ht->arData[i] = \*p; // 当前元素向左移放到无效位置，这里直接复制Bucket实例  }  Z\_EXTRA(ht->arData[i].val) = i; // 记录新顺序号（因为变更了位置）  i++; // 有效元素数  }  ht->nNumUsed = i; // 有效元素数量  }  // 步骤4：复位哈希索引列表  if (!(HT\_FLAGS(ht) & HASH\_FLAG\_PACKED)) { // 如果是哈希表  // 因为前面移动了元素位置，现在哈希索引表已经失效了  HT\_HASH\_RESET(ht); // 哈希索引表的所有元素都设置成-1  }    // 步骤5：给元素列表排序。  // 比较函数：使用 compar() 函数作比较  // 交换函数的选择规则：  // 如果需要重排序号用 zend\_hash\_bucket\_renum\_swap()函数 (交换两个键值对的值，没有交换key和哈希值)  // 不需要重新排序，顺序数组用 zend\_hash\_bucket\_packed\_swap（交换两个Bucket元素的值和哈希值，没有交换key）  // 不需要重新排序，哈希表用 zend\_hash\_bucket\_swap(交换两个键值对的 key，值和哈希)  sort((void \*)ht->arData, ht->nNumUsed, sizeof(Bucket), (compare\_func\_t) compar,  (swap\_func\_t)(renumber? zend\_hash\_bucket\_renum\_swap :  (HT\_IS\_PACKED(ht) ? zend\_hash\_bucket\_packed\_swap : zend\_hash\_bucket\_swap)));  // 步骤6：内部指针复位  ht->nInternalPointer = 0;  // 步骤7：如果要求重新编号，重新编号并清空键名（key）  if (renumber) { // 如果要求重新编号（强转成纯顺序数组）  for (j = 0; j < i; j++) { // 遍历所有有效元素（i，也是所有占用元素）  p = ht->arData + j; // 找到元素  p->h = j; // 哈希值为顺序号  if (p->key) { // 如果有键名  zend\_string\_release(p->key); // 释放key  p->key = NULL; // 清空键名  }  }  ht->nNextFreeElement = i; // 重新编号才更新，下一个空闲元素编号  }  // 步骤8：顺序数组不重新编号要转哈希表，哈希表重新编号要转顺序数组。  if (HT\_IS\_PACKED(ht)) { // 如果是顺序数组  if (!renumber) { // 如果不用重新编号  zend\_hash\_packed\_to\_hash(ht); // 转成哈希表  }  } else { // 哈希表  if (renumber) { // 如果要求 重新编号  void \*new\_data, \*old\_data = HT\_GET\_DATA\_ADDR(ht); // 旧数据块开头位置  Bucket \*old\_buckets = ht->arData; // 旧buckets列表开头  zval \*zv; // 临时变量  // 分配内存创建数据块，元素数和现有的一样  new\_data = pemalloc(HT\_PACKED\_SIZE\_EX(ht->nTableSize, HT\_MIN\_MASK), (GC\_FLAGS(ht) & IS\_ARRAY\_PERSISTENT));  // 添加【顺序数组】标记 和 【使用含静态键名】标记  HT\_FLAGS(ht) |= HASH\_FLAG\_PACKED | HASH\_FLAG\_STATIC\_KEYS;  ht->nTableMask = HT\_MIN\_MASK; // 顺序数组掩码 -2  HT\_SET\_DATA\_ADDR(ht, new\_data); // ht->arData 指向新数据块元素列表开头  p = old\_buckets; // 旧buckets开头开始遍历  zv = ht->arPacked; // 新顺序数组元素列表开头开始遍历  for (i = 0; i < ht->nTableSize; i++) { // 遍历所有元素  ZVAL\_COPY\_VALUE(zv, &p->val); // 把bucket的值复制到 顺序列表里  zv++; // 下一个顺序 zval  p++; // 下一个bucket  }  pefree(old\_data, GC\_FLAGS(ht) & IS\_ARRAY\_PERSISTENT); // 释放旧数据块  HT\_HASH\_RESET\_PACKED(ht); // 顺序数组的哈希索引列表里有2个元素，都设置成 -1  } else { // 不重新编号  zend\_hash\_rehash(ht); // 整理哈希表  }  }  } |

Z\_EXTRA()宏程序的相关介绍参见“类型篇”。

排序过程有8个步骤：

步骤1：跳过无需处理的情况（0个元素 或 1个元素并且不重新编号，这两种情况不处理）；

步骤2：顺序数组先转成哈希表；

步骤3：整理哈希表，整理后不再有无效元素；

步骤4：复位哈希索引列表；

步骤5：给元素列表排序；

步骤6：内部指针复位到0；

步骤7：如果要求重新编号，要重新编号并清空键名（key）；

步骤8：顺序数组不重新编号要转哈希表，哈希表重新编号要转顺序数组。

函数的sort参数是传入的排序函数，用于给元素列表排序，函数原型是sort\_func\_t，定义如下：

|  |
| --- |
| typedef void (\*sort\_func\_t)(void \*, size\_t, size\_t, compare\_func\_t, swap\_func\_t); |

compar函数用于比较两个Bucket实例的大小，函数原型是bucket\_compare\_func\_t，定义如下：

|  |
| --- |
| typedef int (\*bucket\_compare\_func\_t)(Bucket \*a, Bucket \*b); |

元素交换函数原型为swap\_func\_t：

|  |
| --- |
| typedef void (\*swap\_func\_t)(void \*, void \*); |

可能作为交换函数的3个函数：

|  |
| --- |
| // 交换两个键值对的 key，值和哈希  ZEND\_API void zend\_hash\_bucket\_swap(Bucket \*p, Bucket \*q) {  zval val; zend\_ulong h; zend\_string \*key;  val = p->val; h = p->h; key = p->key; // p保存在临时变量中  p->val = q->val; p->h = q->h; p->key = q->key; // q覆盖p  q->val = val; q->h = h; q->key = key; // 临时变量覆盖q  }  // 交换两个键值对的值，没有交换key和哈希值  ZEND\_API void zend\_hash\_bucket\_renum\_swap(Bucket \*p, Bucket \*q) // 逻辑类似  // 交换两个Bucket元素的（值和哈希值），没有交换Key  ZEND\_API void zend\_hash\_bucket\_packed\_swap(Bucket \*p, Bucket \*q) // 逻辑类似 |

## 十四）比较两个数组的大小

zend\_hash\_compare()函数用于比较两个数组的大小：

|  |
| --- |
| // 比较两个数组，p1:数组1，p2:数组2，p3:比较函数，p4:为1时按顺序比较，为0时按键名比较  ZEND\_API int zend\_hash\_compare(HashTable \*ht1, HashTable \*ht2, compare\_func\_t compar, bool ordered) {  int result;  if (ht1 == ht2) { // 如果两个指针指向同一个数组  return 0;  }  // 递归保护，保护一个数组就够了。 如果第二个在第一个里被引用，会导致递归侦测失败  if (UNEXPECTED(GC\_IS\_RECURSIVE(ht1))) { // 如果有递归标记，报错  zend\_error\_noreturn(E\_ERROR, "Nesting level too deep - recursive dependency?");  }  GC\_TRY\_PROTECT\_RECURSION(ht1); // 尝试添加递归保护  // 比较两个数组，impl是implement的缩写  result = zend\_hash\_compare\_impl(ht1, ht2, compar, ordered);  GC\_TRY\_UNPROTECT\_RECURSION(ht1); // 尝试删除递归保护  return result;  } |

代码中用到了递归保护，相关常量和宏程序定义如下：

|  |
| --- |
| #define GC\_PROTECTED (1<<5) // 递归保护标记，用于检测递归  #define GC\_IS\_RECURSIVE(p) (GC\_FLAGS(p) & GC\_PROTECTED) // 检查是否有递归  #define GC\_PROTECT\_RECURSION(p) do { /\* 添加递归标记 \*/ \  GC\_ADD\_FLAGS(p, GC\_PROTECTED); \  } while (0)  #define GC\_UNPROTECT\_RECURSION(p) do { /\* 删除递归标记 \*/ \  GC\_DEL\_FLAGS(p, GC\_PROTECTED); \  } while (0)  #define GC\_TRY\_PROTECT\_RECURSION(p) do { /\* 如果可更改，添加递归标记 \*/ \  if (!(GC\_FLAGS(p) & GC\_IMMUTABLE)) GC\_PROTECT\_RECURSION(p); \  } while (0)  #define GC\_TRY\_UNPROTECT\_RECURSION(p) do { /\* 如果可更改，删除递归标记 \*/ \  if (!(GC\_FLAGS(p) & GC\_IMMUTABLE)) GC\_UNPROTECT\_RECURSION(p); \  } while (0) |

zend\_hash\_compare\_impl()函数是zend\_hash\_compare()函数的具体实现逻辑（implement），真正的比较在这个函数中进行：

|  |
| --- |
| // 比较两个数组，p1:数组1，p2:数组2，p3:比较函数，p4:为1时按顺序比较，为0时按键名比较  static int zend\_hash\_compare\_impl(HashTable \*ht1, HashTable \*ht2, compare\_func\_t compar, bool ordered) {  uint32\_t idx1, idx2; // 顺序号  zend\_string \*key1, \*key2; // 键名  zend\_ulong h1, h2; // 哈希值  zval \*pData1, \*pData2; // 元素指针  int result;  // 先比较元素数，元素数多的大  if (ht1->nNumOfElements != ht2->nNumOfElements) {  return ht1->nNumOfElements > ht2->nNumOfElements ? 1 : -1;  }  // 最外层循环，遍历数组1所有元素  for (idx1 = 0, idx2 = 0; idx1 < ht1->nNumUsed; idx1++) {  // 步骤1：找到数据1的下一个有效元素  if (HT\_IS\_PACKED(ht1)) { // 顺序数组  pData1 = ht1->arPacked + idx1;  h1 = idx1;  key1 = NULL;  } else { // 哈希表  Bucket \*p = ht1->arData + idx1;  pData1 = &p->val;  h1 = p->h;  key1 = p->key;  }  if (Z\_TYPE\_P(pData1) == IS\_UNDEF) continue; // 跳过无效元素，找到数组1的有效元素    // 步骤2：找到数组2的下一个有效元素，进行基本比较  if (ordered) { // 如果要求按顺序比较  if (HT\_IS\_PACKED(ht2)) { // 如果第二个数组是顺序数组  while (1) { // 遍历  pData2 = ht2->arPacked + idx2; // 每个元素  h2 = idx2;  key2 = NULL;  // 跳过无效元素，找到数组2的下一个有效元素  if (Z\_TYPE\_P(pData2) != IS\_UNDEF) break;  idx2++; // 序号+1  }  } else { // 第二个数组是哈希表  while (1) { // 遍历  Bucket \*p;  p = ht2->arData + idx2; // 获取元素  pData2 = &p->val;  h2 = p->h;  key2 = p->key;  // 跳过无效元素，找到数组2的下一个有效元素  if (Z\_TYPE\_P(pData2) != IS\_UNDEF) break;  idx2++; // 序号+1  }  }  // 已经找到数组2的有效元素，开始比较  if (key1 == NULL && key2 == NULL) { // 如果两个元素都没有key  if (h1 != h2) { // 两个数字索引号不同  return h1 > h2 ? 1 : -1; // 数字索引号大的算大  }  // 数字索引号相同，继续往下走  } else if (key1 != NULL && key2 != NULL) { // 两个都有key  if (ZSTR\_LEN(key1) != ZSTR\_LEN(key2)) { // 如果key不一样长  // 比较key的长度，key长的一个算大  return ZSTR\_LEN(key1) > ZSTR\_LEN(key2) ? 1 : -1;  }  // 在内存中比较键名，key1,key2  result = memcmp(ZSTR\_VAL(key1), ZSTR\_VAL(key2), ZSTR\_LEN(key1));  if (result != 0) { // 如果两个key不相等  return result; // 返回比较结果，key大的算大  }  } else { // 一个有key，一个没有  return key1 != NULL ? 1 : -1; // 有key的算大  }  idx2++; // 数组2的序号，可以转到下一个了  } else { // 如果没要求按顺序. 那就是按键名比对，不使用索引号  if (key1 == NULL) { // 数组1的元素没有key  // 用数组1的元素的索引号，在数组2里查找  pData2 = zend\_hash\_index\_find(ht2, h1);  if (pData2 == NULL) { // 数组2里找不到这个索引号  return 1; // 算数组1大  }  } else { // 数组1的元素有key  // 用数组1的元素的键名，在数组2里查找  pData2 = zend\_hash\_find(ht2, key1);  if (pData2 == NULL) { // 数组2里找不到这个键名  return 1; // 算数组1大  }  }  }    // 两个数组都有有效元素，并且键名（或索引号）相同  // 步骤3：追踪间接引用对象  if (Z\_TYPE\_P(pData1) == IS\_INDIRECT) { // 如果第一个是间接引用  pData1 = Z\_INDIRECT\_P(pData1); // 追踪引用目标  }  if (Z\_TYPE\_P(pData2) == IS\_INDIRECT) { // 如果第二个是间接引用  pData2 = Z\_INDIRECT\_P(pData2); // 追踪引用目标  }  // 步骤4：比较。有效元素大于无效元素  if (Z\_TYPE\_P(pData1) == IS\_UNDEF) { // 第一个无效  if (Z\_TYPE\_P(pData2) != IS\_UNDEF) { // 第二个有效  return -1; // 返回，第二个大  }  } else if (Z\_TYPE\_P(pData2) == IS\_UNDEF) { // 第一个有效，第二个无效  return 1; // 返回，第一个大  } else { // 两个都有效  // 【比对】。这里才是真正的比对，绝大部分逻辑都是在处理特殊情况。  result = compar(pData1, pData2);  if (result != 0) { // 如果不相等，返回比较结果  return result;  }  }  }  return 0; // 所有的比较都分不出大小，返回相同  } |

compar参数是自定义的比较函数，它必须与函数原型是compare\_func\_t相一致：

|  |
| --- |
| typedef int (\*compare\_func\_t)(const void \*, const void \*); |

ZSTR\_VAL()、ZSTR\_LEN()宏程序的介绍参见“类型篇”。

## 十五）取得数组的下一个空闲元素编号

zend\_hash\_next\_free\_element()函数用于取得数组的下一个空闲元素编号：

|  |
| --- |
| static zend\_long zend\_hash\_next\_free\_element(const HashTable \*ht) {  return ht->nNextFreeElement;  } |

## 十六）zend\_type.h文件中定义的常量和宏程序

zend\_type.h文件中定义了PHP内核中的基本数据类型（在“类型篇”中有详细介绍），包括数组在内。还定义了一些与数组相关的常量和宏程序，这些常量和宏程序都很常用。

常量列表如下：

|  |  |  |
| --- | --- | --- |
| **常量名** | **值** | **说明** |
| IS\_ARRAY | 7 | 表示数组类型 |
| GC\_ARRAY | IS\_ARRAY | 数组类型 |
| IS\_ARRAY\_EX | IS\_ARRAY| (IS\_TYPE\_REFCOUNTED << 8) | (IS\_TYPE\_COLLECTABLE << 8) | 可引用计数，可回收的数组 |
| IS\_ARRAY\_IMMUTABLE | GC\_IMMUTABLE | 不可更改的数组 |
| IS\_ARRAY\_PERSISTENT | GC\_PERSISTENT | 用malloc()函数创建的数组 |
| GC\_IMMUTABLE | (1<<6) | 不可更改的实例 |
| GC\_PERSISTENT | (1<<7) | 用malloc()函数创建的实例 |
| GC\_FLAGS\_SHIFT | 0 | 位移数，0位，不需要位移 |

宏程序列表如下：

|  |
| --- |
| // 检验zval 类型是否是数组  #define Z\_IMMUTABLE(zval) (Z\_TYPE\_INFO(zval) == IS\_ARRAY)  // 检验zval 类型是否是数组，优化版  #define Z\_OPT\_COPYABLE(zval) (Z\_OPT\_TYPE(zval) == IS\_ARRAY)  // 访问 zval中的 数组指针  #define Z\_ARR(zval) (zval).value.arr  // 通过指针访问 zval中的数组指针  #define Z\_ARR\_P(zval\_p) Z\_ARR(\*(zval\_p))  // Z\_ARR的别名  #define Z\_ARRVAL(zval) Z\_ARR(zval)  // 通过指针访问 zval中的数组指针  #define Z\_ARRVAL\_P(zval\_p) Z\_ARRVAL(\*(zval\_p))  // 给zval 添加 zend\_array 实例指针, 类型更新为可回收的数组  #define ZVAL\_ARR(z, a) do { \  zend\_array \*\_\_arr = (a); \  zval \*\_\_z = (z); \  Z\_ARR\_P(\_\_z) = \_\_arr; \  Z\_TYPE\_INFO\_P(\_\_z) = IS\_ARRAY\_EX; \  } while (0)  // 给使用malloc()函数创建数组, 类型更新为可回收的数组  #define ZVAL\_NEW\_PERSISTENT\_ARR(z) do { \  zval \*\_\_z = (z); \  zend\_array \*\_arr = \  (zend\_array \*) malloc(sizeof(zend\_array)); \  Z\_ARR\_P(\_\_z) = \_arr; \  Z\_TYPE\_INFO\_P(\_\_z) = IS\_ARRAY\_EX; \  } while (0)  // 更新zval实例的指针和类型  # define ZVAL\_COPY\_VALUE\_EX(z, v, gc, t) \  do { \  Z\_COUNTED\_P(z) = gc; /\* 实例指针 \*/ \  Z\_TYPE\_INFO\_P(z) = t; /\* type\_info是整数 \*/ \  } while (0) |

## 十七）让zval实例指向空数组

ZVAL\_EMPTY\_ARRAY()宏程序用于把zval实例设置成数组IS\_ARRAY类型并把它指向空数组：

|  |
| --- |
| #define ZVAL\_EMPTY\_ARRAY(z) do { \  zval \*\_\_z = (z); \  Z\_ARR\_P(\_\_z) = (zend\_array\*)&zend\_empty\_array; \  Z\_TYPE\_INFO\_P(\_\_z) = IS\_ARRAY; \  } while (0) |

zend\_empty\_array变量是预定义好的全局变量：

|  |
| --- |
| extern ZEND\_API const HashTable zend\_empty\_array;  ZEND\_API const HashTable zend\_empty\_array = {  .gc.refcount = 2, // 2次引用  .gc.u.type\_info = IS\_ARRAY | (GC\_IMMUTABLE << GC\_FLAGS\_SHIFT), // 不可更改  .u.flags = HASH\_FLAG\_UNINITIALIZED, // 未初始化过  .nTableMask = HT\_MIN\_MASK, // ((uint32\_t) -2)  .arData = (Bucket\*)&uninitialized\_bucket[2], // 指向无效位置  .nNumUsed = 0, // 占用元素0  .nNumOfElements = 0, // 有效元素0  .nTableSize = HT\_MIN\_SIZE, // 元素数 8  .nInternalPointer = 0, // 内部指针指向序号0  .nNextFreeElement = 0, // 下一个空元素，无  .pDestructor = ZVAL\_PTR\_DTOR // 销毁器  };  // 有0个元素的顺序数组数据块，前面哈希索引列表为2个-1，后面元素空间大小为0  static const uint32\_t uninitialized\_bucket[-HT\_MIN\_MASK] =  {HT\_INVALID\_IDX, HT\_INVALID\_IDX}; |

ZVAL\_PTR\_DTOR销毁器的更多介绍参见“类型篇”。

# 七、符号表（symtable）和属性表（proptable）

**符号表（symtable）**是一种特殊的哈希表，它不可以用数字字符串作键名（key）。如果碰到数字字符串的键名，需要把它转成整数，当成索引号来操作。

**属性表（proptable）**是一个只不包含数字索引的哈希表，它的所有元素必须有字符串键名。

## 一）符号表的常用函数列表

PHP内核中提供了符号表的操作函数：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/**  **用途** | **处理数字字串键名时**  **调用的函数** | **处理非数字字串键名时**  **调用的函数** | **引用**  **次数** |
| zend\_symtable\_add\_new()  添加新元素 | zend\_hash\_index\_add\_new() | zend\_hash\_add\_new() | 少量 |
| zend\_symtable\_update()  更新元素 | zend\_hash\_index\_update() | zend\_hash\_update() | 大量 |
| zend\_symtable\_update\_ind()  更新元素，追踪间接引用目标 | zend\_hash\_index\_update() | zend\_hash\_update\_ind() | 无 |
| zend\_symtable\_del()  删除元素 | zend\_hash\_index\_del() | zend\_hash\_del() | 少量 |
| zend\_symtable\_del\_ind()  删除元素，追踪间接引用目标 | zend\_hash\_index\_del() | zend\_hash\_del\_ind() | 无 |
| zend\_symtable\_find()  查找元素 | zend\_hash\_index\_find() | zend\_hash\_find() | 大量 |
| zend\_symtable\_find\_ind()  查找元素，追踪间接引用目标 | zend\_hash\_index\_find() | zend\_hash\_find\_ind() | 无 |
| zend\_symtable\_exists()  通过键名检验元素是否存在 | zend\_hash\_index\_exists() | zend\_hash\_exists() | 少量 |
| zend\_symtable\_exists\_ind()  检验元素，追踪间接引用目标 | zend\_hash\_index\_exists() | zend\_hash\_exists\_ind() | 无 |
| zend\_symtable\_str\_update()  更新元素 | zend\_hash\_index\_update() | zend\_hash\_str\_update() | 少量 |
| zend\_symtable\_str\_update\_ind()  更新元素，追踪间接引用目标 | zend\_hash\_index\_update() | zend\_hash\_str\_update\_ind() | 少量 |
| zend\_symtable\_str\_del()  删除元素 | zend\_hash\_index\_del() | zend\_hash\_str\_del() | 少量 |
| zend\_symtable\_str\_del\_ind()  删除元素，追踪间接引用目标 | zend\_hash\_index\_del() | zend\_hash\_str\_del\_ind() | 无 |
| zend\_symtable\_str\_find()  查找元素 | zend\_hash\_index\_find() | zend\_hash\_str\_find() | 少量 |
| zend\_symtable\_str\_exists()  通过键名检验元素是否存在 | zend\_hash\_index\_exists() | zend\_hash\_str\_exists() | 少量 |
| zend\_symtable\_str\_find\_ptr()  查找元素，返回元素中的指针 | zend\_hash\_index\_find\_ptr() | zend\_hash\_str\_find\_ptr() | 无 |

以上函数表中的符号表处理函数结构都比较类似，先判断传入的键名是否是数字符串，再调用相应的函数进行处理，第二列和第三列的函数都在前文中介绍过。

函数列表中所有不带有“\_str\_”字样的函数，都接收zend\_string实例作为键名。以zend\_symtable\_add\_new()函数为例：

|  |
| --- |
| static \*zend\_symtable\_add\_new(HashTable \*ht, zend\_string \*key, zval \*pData) {  zend\_ulong idx;  if (ZEND\_HANDLE\_NUMERIC(key, idx)) { // 检查是否是数字字串，并返回转换好的整数  return zend\_hash\_index\_add\_new(ht, idx, pData); // 使用哈希值插入元素  } else { // 普通字符串key  return zend\_hash\_add\_new(ht, key, pData); // 使用键名插入元素  }  } |

函数列表中所有带有“\_str\_”字样的函数，都接收原生字符串类型的键名。以zend\_symtable\_str\_exists()函数为例：

|  |
| --- |
| static bool zend\_symtable\_str\_exists(HashTable \*ht, const char \*str, size\_t len) {  zend\_ulong idx;  if (ZEND\_HANDLE\_NUMERIC\_STR(str, len, idx)) { //检查是否是数字字串，返回转换好的整数  return zend\_hash\_index\_exists(ht, idx);  } else { // 普通字符串key  return zend\_hash\_str\_exists(ht, str, len); // 使用键名插入元素  }  } |

ZEND\_HANDLE\_NUMERIC()宏程序用于检查字符串是否可以转成整数，代码如下：

|  |
| --- |
| #define ZEND\_HANDLE\_NUMERIC(key, idx) \  /\* zend\_string实例转成原生字符串进行处理 \*/ \  ZEND\_HANDLE\_NUMERIC\_STR(ZSTR\_VAL(key), ZSTR\_LEN(key), idx)  #define ZEND\_HANDLE\_NUMERIC\_STR(key, length, idx) \  /\* 检查是否是数字字串。p1:字串，p2:字串长度，p3:返回转换好的整数 \*/ \  \_zend\_handle\_numeric\_str(key, length, &idx) |

\_zend\_handle\_numeric\_str()函数用于检查整数字符串：

|  |
| --- |
| // 检查数字字符串，并转换成整数。p1:字符串，p2:长度，p3:转换成的整数  static bool \_zend\_handle\_numeric\_str(const char \*key, size\_t length, zend\_ulong \*idx) {  const char \*tmp = key; // 先进行简单检查  if (EXPECTED(\*tmp > '9')) { // 如果第一个字符大于字符9，返回false  return 0;  } else if (\*tmp < '0') { // 第一个字符小于字符0  if (\*tmp != '-') { // 如果不是负号，返回false  return 0;  }  tmp++; // 第一个字符是负号，检查下一个字符  if (\*tmp > '9' || \*tmp < '0') { // 第二个字符不是数字字符  return 0; // 返回false  }  }  return \_zend\_handle\_numeric\_str\_ex(key, length, idx); // 通过检查，尝试转换成整数  } |

\_zend\_handle\_numeric\_str\_ex()函数用于把整数字符串转成整数：

|  |
| --- |
| ZEND\_API bool \_zend\_handle\_numeric\_str\_ex(const char \*key,size\_t length, zend\_ulong \*idx){  const char \*tmp = key; // 开头字符  const char \*end = key + length; // 结尾字符  if (\*tmp == '-') { // 允许 负号开头  tmp++;  }  // 如果'0'开头，或位数超长，或当前是32位机器并且溢出最大值  if ((\*tmp == '0' && length > 1)  || (end - tmp > MAX\_LENGTH\_OF\_LONG - 1)  // 32位机器长度溢出的条件：长度满， 开头字符大于2（最大值是 2147483647），认为溢出  || (SIZEOF\_ZEND\_LONG == 4 &&  end - tmp == MAX\_LENGTH\_OF\_LONG - 1 &&  \*tmp > '2')) { /\* overflow \*/  return 0;  }  \*idx = (\*tmp - '0'); // 通过和字符'0'的ascii码序差异，算出数值  while (1) {  ++tmp; // 下一个字符  if (tmp == end) { // 如果到达最后一个字符  if (\*key == '-') { // 如果开头是负号  // 负数的结对值可以比 ZEND\_LONG\_MAX 大1 ，如果-1 还大于 ZEND\_LONG\_MAX，就会溢出  if (\*idx-1 > ZEND\_LONG\_MAX) { // 也可以是 if (\*idx > ZEND\_LONG\_MAX +1 )  return 0; // 无法转换，返回false  }  \*idx = 0 - \*idx; // 转成负数  } else if (\*idx > ZEND\_LONG\_MAX) { // 如果正数溢出  return 0; // 返回false  }  return 1; // 无特殊情况，返回 true  }  if (\*tmp <= '9' && \*tmp >= '0') { // 如果字符合法  \*idx = (\*idx \* 10) + (\*tmp - '0'); // 原数值\*10 + 新数值  } else { // 只要有一个字符不合法，返回 false  return 0;  }  }  }  # define MAX\_LENGTH\_OF\_LONG 20 // 64位系统中整数最长20位 |

## 二）初始化符号表

ZEND\_INIT\_SYMTABLE()宏程序用于把数组（zend\_array实例）初始化成符号表：

|  |
| --- |
| #define ZEND\_INIT\_SYMTABLE(ht) \  /\* 初始化哈希表，只设置基础属性，未创建元素空间 \*/ \  ZEND\_INIT\_SYMTABLE\_EX(ht, 8, 0) |

它调用ZEND\_INIT\_SYMTABLE\_EX()宏程序：

|  |
| --- |
| // 初始化符号表，只设置基础属性，未创建元素空间。p1:哈希表，p2:元素空间数量，p3:是否永久  #define ZEND\_INIT\_SYMTABLE\_EX(ht, n, persistent) \  zend\_hash\_init(ht, n, NULL, ZVAL\_PTR\_DTOR, persistent) |

## 二）清空符号表

zend\_symtable\_clean()函数用来清空符号表：

|  |
| --- |
| // 清空符号表, 销毁所有元素，只有 zend\_execute.c 中调用一次  ZEND\_API void zend\_symtable\_clean(HashTable \*ht) {  Bucket \*p, \*end;  if (ht->nNumUsed) { // 使用元素数 不是0  p = ht->arData; // 元素列表开头  end = p + ht->nNumUsed; // 元素列表结尾  if (HT\_HAS\_STATIC\_KEYS\_ONLY(ht)) { // 如果 只包含静态key  do { // 遍历每一个元素  i\_zval\_ptr\_dtor(&p->val); // 依次销毁 每个元素  } while (++p != end);  } else if (HT\_IS\_WITHOUT\_HOLES(ht)) { // 如果 没有空元素  do { // 遍历每一个元素  i\_zval\_ptr\_dtor(&p->val); // 依次销毁 每个元素  if (EXPECTED(p->key)) { // 如果键名有效  zend\_string\_release(p->key); // 要释放键名字串  }  } while (++p != end);  } else { // 有动态key  do { // 遍历每一个元素  if (EXPECTED(Z\_TYPE(p->val) != IS\_UNDEF)) { // 元素有效  i\_zval\_ptr\_dtor(&p->val); // 销毁元素  if (EXPECTED(p->key)) { // 如果有键名  zend\_string\_release(p->key); // 释放键名  }  }  } while (++p != end);  }  HT\_HASH\_RESET(ht); // 重置哈希索引列表，所有元素都设置成 -1  }  ht->nNumUsed = 0; // 使用数0  ht->nNumOfElements = 0; // 元素数0  ht->nNextFreeElement = ZEND\_LONG\_MIN; // 下一个可用，指向无效位置  ht->nInternalPointer = 0; // 内部指针 0  } |

## 三）符号表转属性表

zend\_symtable\_to\_proptable()函数可以把属性表转成符号表：

|  |
| --- |
| ZEND\_API HashTable\* zend\_symtable\_to\_proptable(HashTable \*ht) {  zend\_ulong num\_key;  zend\_string \*str\_key;  zval \*zv;  if (UNEXPECTED(HT\_IS\_PACKED(ht))) { // 情况1：如果是顺序数组  goto convert; // 转换  }  ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY(ht, str\_key) { // 情况2：如果是哈希表，遍历哈希表  if (!str\_key) { // 只要有一个元素没有字符串键名  goto convert; // 转换  }  } ZEND\_HASH\_FOREACH\_END();  // 情况3：哈希表所有元素全都有字符串key，已经满足属性表要求  if (!(GC\_FLAGS(ht) & IS\_ARRAY\_IMMUTABLE)) { // 如果不是不可更改  GC\_ADDREF(ht); // 增加引用次数  }  return ht; // 返回哈希表  convert: // 转换  {  // 创建新数组，元素数与源数组相同  HashTable \*new\_ht = zend\_new\_array(zend\_hash\_num\_elements(ht));  ZEND\_HASH\_FOREACH\_KEY\_VAL(ht, num\_key, str\_key, zv) { // 遍历哈希表  if (!str\_key) { // 如果 没有key  str\_key = zend\_long\_to\_str(num\_key); // 索引号转字符串  zend\_string\_delref(str\_key); // 字符串减少引用次数  }  do { // 为了break  if (Z\_OPT\_REFCOUNTED\_P(zv)) { // 元素是可计数类型  // 引用类型 并且 引用数是1  if (Z\_ISREF\_P(zv) && Z\_REFCOUNT\_P(zv) == 1) {  zv = Z\_REFVAL\_P(zv); // 追踪到引用目标  if (!Z\_OPT\_REFCOUNTED\_P(zv)) { // 如果目标值不是可计数类型  break; // 跳出  }  }  Z\_ADDREF\_P(zv); // 可计数，元素增加引用次数  }  } while (0);  zend\_hash\_update(new\_ht, str\_key, zv); // 元素插入到新哈希表中  } ZEND\_HASH\_FOREACH\_END(); // 结束遍历  return new\_ht; // 返回新哈希表  }  } |

## 四）属性表转符号表

zend\_proptable\_to\_symtable()函数可以把属性表转成符号表：

|  |
| --- |
| ZEND\_API HashTable\* zend\_proptable\_to\_symtable(HashTable \*ht, bool always\_duplicate) {  zend\_ulong num\_key;  zend\_string \*str\_key;  zval \*zv;  if (!HT\_IS\_PACKED(ht)) { // 如果不是顺序数组  ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY(ht, str\_key) { // 遍历哈希表  // 有字串key 并且 字串是数字字串  if (str\_key && ZEND\_HANDLE\_NUMERIC(str\_key, num\_key)) {  goto convert; // 转换  }  } ZEND\_HASH\_FOREACH\_END();  }  if (always\_duplicate) { // 如果总是需要复制  return zend\_array\_dup(ht); // 返回副本  }  if (EXPECTED(!(GC\_FLAGS(ht) & IS\_ARRAY\_IMMUTABLE))) { // 如果不是不可更改  GC\_ADDREF(ht); // 增加引用数  }  return ht; // 返回ht（always\_duplicate为真时是副本）  convert: // 转换  {  // 创建新数组，元素数与原数组相同  HashTable \*new\_ht = zend\_new\_array(zend\_hash\_num\_elements(ht));  ZEND\_HASH\_MAP\_FOREACH\_KEY\_VAL\_IND(ht, num\_key, str\_key, zv) { // 遍历  do {  if (Z\_OPT\_REFCOUNTED\_P(zv)) { // 如果值是可计数的  // 如果值是引用类型 并且 引用数是1  if (Z\_ISREF\_P(zv) && Z\_REFCOUNT\_P(zv) == 1) {  zv = Z\_REFVAL\_P(zv); // 追踪到引用目标  if (!Z\_OPT\_REFCOUNTED\_P(zv)) { // 如果目标不可计数  break; // 跳过计数  }  }  Z\_ADDREF\_P(zv); // 值增加引用数  }  } while (0);  // 如果没有字串键 或 使用了数字字串键  if (!str\_key || ZEND\_HANDLE\_NUMERIC(str\_key, num\_key)) {  // 把元素用数字索引号插入到新数组  zend\_hash\_index\_update(new\_ht, num\_key, zv);  } else { // 其他情况  zend\_hash\_update(new\_ht, str\_key, zv); // 按字串key更新元素  }  } ZEND\_HASH\_FOREACH\_END();  return new\_ht; // 返回新表  }  } |

# **八、附录**

## 一）\_zend\_hash\_index\_add\_or\_update\_i()函数

\_zend\_hash\_index\_add\_or\_update\_i()函数用于通过元素顺序号（顺序数组）或索引号（哈希表），添加、修改、查找指定元素。此函数非常重要，已经在顺序数组和哈希表章节中介绍过，为了便于理解，再附加此函数的完整注释和业务逻辑分支表，以及衍生函数列表。

代码注释如下：

|  |
| --- |
| static zval \*\_zend\_hash\_index\_add\_or\_update\_i(HashTable \*ht, zend\_ulong h, zval \*pData, uint32\_t flag) {  uint32\_t nIndex; // 哈希索引表中的序号  uint32\_t idx; // 元素序号  Bucket \*p; // 键值对  zval \*zv; // 值  // 这个判断是针对未初始化数组的，如果在下一个位置添加，并且 位置是 最小整数  if ((flag & HASH\_ADD\_NEXT) && h == ZEND\_LONG\_MIN) {  h = 0; // 位置是 0  }  // 看似复杂，重点是情况 1 和情况 3，哈希表和顺序数组分开处理，互相独立，只有replace一个跳转点交叉。  if (HT\_IS\_PACKED(ht)) { // 情况1：如果是顺序数组  // 情况1.1 ：如果不是HASH\_ADD\_NEW|HASH\_ADD\_NEXT ( 必定也有 HASH\_ADD ） ，并且指向使用过的位置。6个子方法里只有 zend\_hash\_next\_index\_insert\_new 一定不走这里。  if ((flag & (HASH\_ADD\_NEW|HASH\_ADD\_NEXT)) !=  (HASH\_ADD\_NEW|HASH\_ADD\_NEXT) && h < ht->nNumUsed) {  zv = ht->arPacked + h; // 顺序数组zval列表中，找到指定元素  // 情况1.1.1 ：如果位置有有效元素（不在新位置添加）  if (Z\_TYPE\_P(zv) != IS\_UNDEF) {  // 情况1.1.1.1:如果是查找操作（没有 HASH\_ADD\_NEW 和 HASH\_ADD\_NEXT）  if (flag & HASH\_LOOKUP) {  return zv; // 返回元素  }  replace: // 覆盖操作跳转点，这里只更新zval，一次引用（哈希表需要更新元素时）  if (flag & HASH\_ADD) { // 情况1.1.1.2: 如果是添加操作  return NULL; // 添加失败返回null（顺序数组不允许在同一位置添加两个元素）  }  // 情况1.1.1.3.：不是 HASH\_LOOKUP，HASH\_ADD，只能是 HASH\_UPDATE  if (ht->pDestructor) { // 情况1.1.1.3.1 有销毁器  ht->pDestructor(zv); // 销毁此元素上的变量  }  // 情况1.1.1.3.2：没有销毁器  ZVAL\_COPY\_VALUE(zv, pData); // 把新元素添加到指定位置  return zv; // 返回新元素  // 情况1.1.2 ：如果不是按顺序使用的。必须要保持这个顺序。  // （确定不是顺序数组了，即使是查询也要先转换成哈希表。）  } else { /\* we have to keep the order :( \*/  goto convert\_to\_hash; // 转成哈希表，为了保持顺序  }  // 情况1.2 ： 顺序数组，HASH\_ADD\_NEW|HASH\_ADD\_NEXT ( 必定也有 HASH\_ADD ）或 访问未使用过的序号  } else if (EXPECTED(h < ht->nTableSize)) { // 如果位置在已有大小内  add\_to\_packed: // 添加到顺序数组，跳转点。2次引用（改变大小 或 初始化后）  zv = ht->arPacked + h; // 转到指定元素  // 如果 不是 HASH\_ADD\_NEW|HASH\_ADD\_NEXT( 必定也有 HASH\_ADD ）  // 那就是因为访问未使用过的编号才进来的  if ((flag & (HASH\_ADD\_NEW|HASH\_ADD\_NEXT)) !=  (HASH\_ADD\_NEW|HASH\_ADD\_NEXT)) {  // 如果位置超过了当前已使用位置  if (h > ht->nNumUsed) {  zval \*q = ht->arPacked + ht->nNumUsed; // 转到 已用元素最后 位置  while (q != zv) { // 遍历最后一个元素到当前元素之间（不包含当前元素）  ZVAL\_UNDEF(q); // 这一段 元素设置成 未定义  q++; // 移动到下一个元素  }  // 但这里并没有把数组转成哈希表，要下一次查询或添加元素时才转换  }  // 如果顺序刚好接上（ h == ht->nNumUsed ），直接到这里，不需要调整位置  }  // 下一个空元素编号 和 已使用元素数 = h+1，中间那段空的也算已使用了  ht->nNextFreeElement = ht->nNumUsed = h + 1;  ht->nNumOfElements++; // 有效元素数 +1  if (flag & HASH\_LOOKUP) { // 情况1.2.1: 如果是查找  ZVAL\_NULL(zv); // 新元素设置为null  } else { // 情况1.2.2: 其他操作  ZVAL\_COPY\_VALUE(zv, pData); // 把新元素添加到指定位置  }  return zv; // 返回当前元素  // 情况1.3 ：顺序数组，HASH\_ADD\_NEW|HASH\_ADD\_NEXT ( 必定也有 HASH\_ADD ）或 访问未使用过的序号  // 如果， 哈希值/2 小于 tablesize 并且 tablesize/2 小于元素数  // 翻倍后 序号可以直接覆盖要求的哈希值  // 并且 哈希表已经使用一半以上  } else if ((h >> 1) < ht->nTableSize &&  (ht->nTableSize >> 1) < ht->nNumOfElements) {  zend\_hash\_packed\_grow(ht); // 大小翻倍  goto add\_to\_packed; // 重新尝试添加. 并且返回  // 情况1.4：顺序数组，HASH\_ADD\_NEW|HASH\_ADD\_NEXT ( 必定也有 HASH\_ADD ）或 访问未使用过的序号  // 其他情况，序号大于元素数，又不适合翻倍，转成哈希表。  } else {  if (ht->nNumUsed >= ht->nTableSize) { // 如果已使用大小比初始大小大  ht->nTableSize += ht->nTableSize; // 大小翻倍  }  convert\_to\_hash: // 转成哈希表，跳转点, 1次引用  // 顺序数组转哈希表。这个转换 删除 了顺序数组标记 HASH\_FLAG\_PACKED  zend\_hash\_packed\_to\_hash(ht);  // 再进行 增改查操作  }  // 情况2： 如果没初始化过。（既不是 哈希表 也不是 顺序数组）  } else if (HT\_FLAGS(ht) & HASH\_FLAG\_UNINITIALIZED) {  // 情况2.1: 哈希值可以作为索引号，没有越界，当成顺序数组处理  if (h < ht->nTableSize) {  zend\_hash\_real\_init\_packed\_ex(ht); // 初始化顺序数组  goto add\_to\_packed; // 重新尝试添加, 并返回  }  // 情况2.2: 哈希值越界，当成哈希表处理  // 初始化哈希表  zend\_hash\_real\_init\_mixed(ht);  } else { // 情况3：如果 是哈希表（初始化过）。 只有这里单独用到 HASH\_ADD\_NEW  // 情况3.1:  // 如果不是 HASH\_ADD\_NEW 添加新元素，那就是 【查找】 或 【更新】  // 或者是调试模式（不相干）  if ((flag & HASH\_ADD\_NEW) == 0 || ZEND\_DEBUG) {  // 【查找】 或 【更新】都要先查找  // 使用 哈希值(zend\_ulong) 获取哈希表里【没有key，只有哈希值的键值对】  p = zend\_hash\_index\_find\_bucket(ht, h);  if (p) { // 情况3.1.1: 如果能查到旧元素  if (flag & HASH\_LOOKUP) { // 情况3.1.1.1如果是【查找】  return &p->val; // 直接返回  }  // 情况3.1.1.2是【添加】或【更新】  zv = &p->val; // 取出键值对的值 指针  goto replace; // 转到替换，并返回  }  // 情况3.1.2: 【查找】 或 【更新】时，没有旧值，往下走。  // 【查找】应返回空元素，【更新】应插入新元素  }  // 情况3.2: HASH\_ADD\_NEW  // 【查找】不可能出现。 HASH\_ADD\_NEW 和 HASH\_LOOKUP 不同时使用  // 【更新】应插入新元素（HASH\_ADD | HASH\_ADD\_NEW，或 HASH\_ADD | HASH\_ADD\_NEW | HASH\_ADD\_NEXT)  // 哈希表有没有 HASH\_ADD\_NEXT 没区别。  ZEND\_HASH\_IF\_FULL\_DO\_RESIZE(ht); // 如果哈希表满了，改变大小  }  **// 后半段业务逻辑**  // 会到达这里的情况 （一定是哈希表操作）  // 情况1.1.2：顺序数组，不是按顺序使用的（当前元素在已用编号内，但值为未定义），转成哈希表。  // 情况1.4：是按顺序新增，序号大于元素数，又不适合翻倍，转成哈希表。  // 情况2.2：未初始化的哈希表，访问序号大于元素数（无法直接当成顺序数组使用），初始化成哈希表。  // 情况3.1.2：已初始化的哈希表，【查找】 或 【更新】时，没有旧值  // 情况3.2：哈希表中插入新元素    idx = ht->nNumUsed++; // 索引号，自增  nIndex = h | ht->nTableMask; // 计算哈希索引表中的序号  p = ht->arData + idx; // 找到指定的 bucket  Z\_NEXT(p->val) = HT\_HASH(ht, nIndex); // 新元素串在相同哈希值元素列表的开头  HT\_HASH(ht, nIndex) = HT\_IDX\_TO\_HASH(idx); // 哈希索引表中的元素指向链表开头  if ((zend\_long)h >= ht->nNextFreeElement) { // 如果哈希值大于下一个空元素  // 更新下一个空元素序号，序号是 h+1, 防止超长。  ht->nNextFreeElement = (zend\_long)h < ZEND\_LONG\_MAX ? h + 1 : ZEND\_LONG\_MAX;  }  ht->nNumOfElements++; // 元素数量+1  p->h = h; // 更新哈希值  p->key = NULL; // key 为null  if (flag & HASH\_LOOKUP) { // 如果是查询（查询也先把val更新了）  ZVAL\_NULL(&p->val); // 值为null  } else { // 不是查询  ZVAL\_COPY\_VALUE(&p->val, pData); // 更新键值对的值  }  return &p->val; // 返回 键值对的值  } |

### 业务逻辑分支表

业务逻辑分支表如下：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **第一层条件** | **第二层条件** | **第三层条件** | **第四层条件** | **第五层条件** | **处理逻辑** | **分支编号** |
| 如果是顺序数组 | 操作不是HASH\_ADD\_NEW|  HASH\_ADD\_NEXT  并且访问已占用的元素 | 如果位置有有效元素 | HASH\_LOOKUP，查找元素 | | 返回元素 | 1.1.1.1 |
| HASH\_ADD，添加元素 | | 返回NULL | 1.1.1.2 |
| HASH\_UPDATE，  更新元素 | 有销毁器 | 销毁此元素，写入新元素 | 1.1.1.3.1 |
| 没销毁器 | 用新元素覆盖 | 1.1.1.3.2 |
| 位置没有有效元素，说明不是按顺序使用的 | | | 转哈希表保持顺序，走后半段添加或查找 | 1.1.2 |
| 元素序号在已分配元素范围内但未占用 | HASH\_LOOKUP，查找元素 | | | 填充无效元素，插入NULL，返回新元素 | 1.2.1 |
| HASH\_ADD，添加元素 | | | 填充无效元素，插入元素，返回新元素 | 1.2.2 |
| 序号超限，翻倍后可以满足此序号，有效元素大于已分配元素的一半 | | | | 元素数量翻倍，添加元素 | 1.3 |
| 其他情况：HASH\_ADD\_NEXT访问未占用元素，或不适合元素翻倍 | | | | 转成哈希表，走后半段添加或查找 | 1.4 |
| 数组没初始化过 | 访问序号小于nTableSize | | | | 初始化顺序数组，添加或查找元素 | 2.1 |
| 访问序号大于等于nTableSize | | | | 初始化成哈希表，走后半段添加或查找 | 2.2 |
| 是初始化过的哈希表 | 如果不是 HASH\_ADD\_NEW 添加新元素，那就是 【查找】 或 【更新】 | 如果能查到旧元素 | HASH\_LOOKUP，查找元素 | | 返回此元素 | 3.1.1.1 |
| HASH\_ADD或HASH\_UPDATE | | 替换元素，并返回新元素 | 3.1.1.2 |
| 查不到旧元素 | | | 检查空间是否足够，走后半段添加或查找 | 3.1.2 |
| HASH\_ADD\_NEW | | | | 检查空间是否足够，走后半段添加或查找 | 3.2 |

如上表所示，每增加一层判断，在业务逻辑分支编号上增加一位数字。业务逻辑归纳如下：

分支 1.1.1.1，顺序数组，HASH\_LOOKUP查找操作，访问已占用的元素，位置有有效元素：返回旧元素；

分支 1.1.1.2，顺序数组，HASH\_ADD新增操作，访问已占用的元素，位置有有效元素：返回NULL；

分支 1.1.1.3.1，顺序数组，HASH\_UPDATE更新操作，访问已占用的元素，位置有有效元素，有销毁器：返销毁旧元素，写入新元素；

分支 1.1.1.3.2，顺序数组，HASH\_UPDATE更新操作，访问已占用的元素，位置有有效元素，没有销毁器：用新元素覆盖旧元素；

分支 1.1.2，顺序数组，访问已占用的元素，位置没有有效元素，说明不是按顺序使用的：转哈希表保持元素添加顺序，走后半段添加或查找；

分支 1.2.1，顺序数组，元素序号在已分配元素范围内但未占用，HASH\_LOOKUP查找操作：把这个元素前的空闲元素都标记成无效：插入NULL，返回新插入的元素；

分支 1.2.2，顺序数组，元素序号在已分配元素范围内但未占用，HASH\_ADD添加操作：把这个元素前的空闲元素都标记成无效：插入元素，返回新插入的元素；

分支 1.3，顺序数组，序号超过已分配元素数，元素数翻倍后可以满足此序号，有效元素大于已分配元素的一半：分配空间元素数量翻倍，添加元素；

分支 1.4，顺序数组其他情况，HASH\_ADD\_NEXT访问未占用元素，或不适合元素翻倍：转成哈希表，走后半段添加或查找；

分支 2.1，数组没初始化过，访问序号小于nTableSize：初始化顺序数组，添加或查找元素；

分支 2.2，数组没初始化过，访问序号大于等于nTableSize：初始化成哈希表，走后半段添加或查找；

分支 3.1.1.1，初始化过的哈希表，HASH\_LOOKUP查找操作，如果能查到旧元素：返回此元素；

分支 3.1.1.2，初始化过的哈希表，HASH\_ADD添加或HASH\_UPDATE更新操作，如果能查到旧元素：替换元素，并返回；

分支 3.1.2，初始化过的哈希表，初始化过的哈希表，HASH\_ADD添加或HASH\_UPDATE更新操作，查不到旧元素：检查空间是否足够，走后半段添加或查找；

分支 3.2，初始化过的哈希表，HASH\_ADD\_NEW新增操作，检查空间是否足够：走后半段添加或查找。

其中分支1.1.2、分支1.4、分支2.2、分支3.2会用到函数的后半段业务逻辑：在哈希表中添加或查找。

### 相关的衍生函数

\_zend\_hash\_index\_add\_or\_update\_i()函数一般不被直接调用，而是调用它的衍生函数：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名** | **附加标记** | **可能用到**  **的分支** | **业务逻辑归纳** |
| zend\_hash\_index\_add() | HASH\_ADD | 1.1.2  1.2.2  1.3  3.1.1.2  3.1.2 | 顺序数组中找到旧元素直接中断。哈希表中找到旧元素会替换掉 |
| zend\_hash\_index\_add\_new() | HASH\_ADD | HASH\_ADD\_NEW | 1.1.1.2  1.1.2  1.2.2  1.3  3.1.1.2  3.1.2 | 顺序数组，碰到旧元素时新增元素。哈希表不查找旧元素，直接添加新元素 |
| zend\_hash\_index\_update() | HASH\_UPDATE | 1.1.1.3.1  1.1.1.3.2  1.1.2  1.3  3.1.1.2  3.1.2 | 顺序数组和哈希表中，找到旧元素都更新，找不到旧元素都添加 |
| zend\_hash\_next\_index\_insert() | HASH\_ADD | HASH\_ADD\_NEXT | 1.1.1.2  1.1.2  1.2.2  1.3  1.4  3.1.1.2  3.1.2 | 顺序数组中找到旧元素直接中断。在哈希表中找到旧元素会替换掉 |
| zend\_hash\_next\_index\_insert\_new() | HASH\_ADD | HASH\_ADD\_NEW | HASH\_ADD\_NEXT | 1.2.2  1.3  1.4  3.2 | 在顺序数组和哈希表中都直接添加元素，不查旧元素 |
| zend\_hash\_index\_lookup() | HASH\_LOOKUP | 1.1.1.1  1.2.1  3.1.1.1  1.1.2  1.3  3.1.1.1  3.1.2 | 通过序号（或索引号）查找元素 |

除以上表格中列出的分支外，所有函数都有可能用到分支2.1和2.2。

## 二）声名但无调用的函数和宏程序

以下函数和宏程序在全部PHP代码中没有被调用过。

### zend\_hash\_graceful\_destroy()函数

zend\_hash\_graceful\_destroy()函数用于优雅地正序销毁数组的数据列表，定义如下：

|  |
| --- |
| ZEND\_API void zend\_hash\_graceful\_destroy(HashTable \*ht) {  uint32\_t idx;  if (HT\_IS\_PACKED(ht)) { // 如果是顺序数组  zval \*zv = ht->arPacked; // 元素列表开头  for (idx = 0; idx < ht->nNumUsed; idx++, zv++) { // 遍历元素列表  // 如果 碰到无效元素，跳过  if (UNEXPECTED(Z\_TYPE\_P(zv) == IS\_UNDEF)) continue;  // 顺序数组删除指定 序号 的元素。删除元素主要靠销毁器(pDestructor)完成。  \_zend\_hash\_packed\_del\_val(ht, HT\_IDX\_TO\_HASH(idx), zv);  }  } else { // 哈希表  // bucket列表开头  Bucket \*p = ht->arData;  for (idx = 0; idx < ht->nNumUsed; idx++, p++) { // 遍历所有 bucket  // 如果 碰到无效元素，跳过  if (UNEXPECTED(Z\_TYPE(p->val) == IS\_UNDEF)) continue;  // 哈希表中删除指定 序号的 bucket  \_zend\_hash\_del\_el(ht, HT\_IDX\_TO\_HASH(idx), p);  }  }  if (!(HT\_FLAGS(ht) & HASH\_FLAG\_UNINITIALIZED)) { // 如果数组已初始化，说明有数据块  // 释放数组数据块  pefree(HT\_GET\_DATA\_ADDR(ht), GC\_FLAGS(ht) & IS\_ARRAY\_PERSISTENT);  }  } |

### zend\_hash\_add\_or\_update()函数

zend\_hash\_add\_or\_update()函数是\_zend\_hash\_add\_or\_update\_i()函数的各种调用情况的综合描述：

|  |
| --- |
| ZEND\_API zval\* zend\_hash\_add\_or\_update(HashTable \*ht, zend\_string \*key, zval \*pData, uint32\_t flag) {  if (flag == HASH\_ADD) { // 添加元素  return zend\_hash\_add(ht, key, pData); // 插入元素，有旧元素中断并返回null  } else if (flag == HASH\_ADD\_NEW) { // 直接添加元素  return zend\_hash\_add\_new(ht, key, pData); // 不查检，直写入指定位置元素  } else if (flag == HASH\_UPDATE) { // 更新元素  return zend\_hash\_update(ht, key, pData); // 不管找不到到元素，都更新指定位置的元素  } else { //其他情况  // 操作必须是 HASH\_UPDATE|HASH\_UPDATE\_INDIRECT  ZEND\_ASSERT(flag == (HASH\_UPDATE|HASH\_UPDATE\_INDIRECT));  // 不管找不到到元素，都更新指定位置的元素。如果元素是间接引用，追踪引用对象。  return zend\_hash\_update\_ind(ht, key, pData);  }  } |

### zend\_hash\_index\_add\_or\_update()函数

zend\_hash\_index\_add\_or\_update()函数是\_zend\_hash\_index\_add\_or\_update\_i()函数的所有调用的综合描述：

|  |
| --- |
| ZEND\_API zval\* zend\_hash\_index\_add\_or\_update(HashTable \*ht, zend\_ulong h, zval \*pData, uint32\_t flag) {  if (flag == HASH\_ADD) {  // 顺序数组中找到旧元素直接中断，在哈希表中找到旧元素会替换掉  return zend\_hash\_index\_add(ht, h, pData);  } else if (flag == (HASH\_ADD|HASH\_ADD\_NEW)) {  // 顺序数组，碰到旧的新增。哈希表不查找旧元素，直接添加新元素  return zend\_hash\_index\_add\_new(ht, h, pData);  } else if (flag == (HASH\_ADD|HASH\_ADD\_NEXT)) {  ZEND\_ASSERT(h == ht->nNextFreeElement);  // （哈希值是最小整数时，归0）顺序数组中找到旧元素直接中断  return zend\_hash\_next\_index\_insert(ht, pData);  } else if (flag == (HASH\_ADD|HASH\_ADD\_NEW|HASH\_ADD\_NEXT)) {  ZEND\_ASSERT(h == ht->nNextFreeElement);  // （哈希值是最小整数时，归0）在顺序数组和哈希表中都直接添加  return zend\_hash\_next\_index\_insert\_new(ht, pData);  } else {  ZEND\_ASSERT(flag == HASH\_UPDATE);  // 找到旧元素都更新。找不到旧元素都添加，数组会转成哈希表  return zend\_hash\_index\_update(ht, h, pData);  }  } |

### zend\_hash\_str\_add\_or\_update()函数

zend\_hash\_str\_add\_or\_update()函数是\_zend\_hash\_str\_add\_or\_update\_i()函数的所有调用的综合描述：

|  |
| --- |
| ZEND\_API zval\* zend\_hash\_str\_add\_or\_update(HashTable \*ht, const char \*str, size\_t len, zval \*pData, uint32\_t flag) {  if (flag == HASH\_ADD) {  // （强转哈希表）添加，碰到已有元素返回null。否则添加  return zend\_hash\_str\_add(ht, str, len, pData);  } else if (flag == HASH\_ADD\_NEW) {  // （强转哈希表）添加，不检查已有，直接找到位置，写入元素  return zend\_hash\_str\_add\_new(ht, str, len, pData);  } else if (flag == HASH\_UPDATE) {  // （强转哈希表）更新，有旧元素时替换旧元素，否则写入新元素  return zend\_hash\_str\_update(ht, str, len, pData);  } else {  ZEND\_ASSERT(flag == (HASH\_UPDATE|HASH\_UPDATE\_INDIRECT));  //（强转哈希表）更新，有旧元素时，追踪引用目标，替换旧元素。没有旧元素写入新元素  return zend\_hash\_str\_update\_ind(ht, str, len, pData);  }  } |

### 内存管理器zend\_alloc.h中的宏程序

|  |
| --- |
| // 生产环境中，emalloc\_rel()函数效果与emalloc()函数相同，efree\_size\_rel()函数与efree\_size()函数效果相同，所以下面这两个宏程序与上面两个功能相同  #define ALLOC\_HASHTABLE\_REL(ht) (ht) = (HashTable \*) emalloc\_rel(sizeof(HashTable))  #define FREE\_HASHTABLE\_REL(ht) efree\_size\_rel(ht, sizeof(HashTable)) |

## 三）其他不常用的函数和宏程序

### zend\_new\_pair()函数

zend\_new\_pair()函数用于创建一个包含两个元素的顺序数组，并添加两个传入的元素：

|  |
| --- |
| // 创建一个包含两个元素的顺序数组  ZEND\_API HashTable\* zend\_new\_pair(zval \*val1, zval \*val2) {  zval \*zv;  HashTable \*ht = emalloc(sizeof(HashTable)); // 分配内存创建哈希表  // 初始化哈希表，使用heap创建元素列表  \_zend\_hash\_init\_int(ht, HT\_MIN\_SIZE, ZVAL\_PTR\_DTOR, 0);  // 占用元素数，有效元素数，下一个空闲元素序号都是2（这样就不会分配8个元素了）  ht->nNumUsed = ht->nNumOfElements = ht->nNextFreeElement = 2;  zend\_hash\_real\_init\_packed\_ex(ht); // 初始化成顺序数组  zv = ht->arPacked; // 元素列表开头  ZVAL\_COPY\_VALUE(zv, val1); // 复制第一个元素  zv++; // 指向第二个元素  ZVAL\_COPY\_VALUE(zv, val2); // 复制第二个元素  return ht; // 返回哈希表指针  } |

## 五）排序算法

PHP内核中用到的排序算法有插入排序和快速排序两种。

### 插入排序

zend\_insert\_sort()函数用于插入排序，插入排序算法的时间复杂度是O(n\*n)，代码如下：

|  |
| --- |
| ZEND\_API void zend\_insert\_sort(void \*base, size\_t nmemb, size\_t siz, compare\_func\_t cmp, swap\_func\_t swp) {  switch (nmemb) { // 根据元素数量进行处理  case 0: // 0 个，1个，不用排  case 1:  break;  case 2: // 2个元素  zend\_sort\_2(base, (char \*)base + siz, cmp, swp);  break;  case 3: // 3个元素  zend\_sort\_3(base, (char \*)base + siz, (char \*)base + siz + siz, cmp, swp);  break;  case 4: // 4个元素  {  size\_t siz2 = siz + siz;  zend\_sort\_4(base, (char \*)base + siz, (char \*)base + siz2, (char \*)base + siz + siz2, cmp, swp);  }  break;  case 5: // 5个元素  {  size\_t siz2 = siz + siz;  zend\_sort\_5(base, (char \*)base + siz, (char \*)base + siz2, (char \*)base + siz + siz2, (char \*)base + siz2 + siz2, cmp, swp);  }  break;  default: // 5个以上元素  {  char \*i, \*j, \*k;  char \*start = (char \*)base; // 开始位置  char \*end = start + (nmemb \* siz); // 结束位置  size\_t siz2= siz + siz; // siz的2倍  char \*sentry = start + (6 \* siz); // 哨兵 最少6个，哨兵是第6个位置  // 从第二个，到哨兵位置（前面这一段每次跳一个来比较）  for (i = start + siz; i < sentry; i += siz) {  j = i - siz; // 前一个位置  if (!(cmp(j, i) > 0)) { // 如果前面的不大于后面的  continue; // 下一个  }  // 如果前面的大于后面的，要找位置，让 i 冒泡上来  while (j != start) { // 从 j 开始往前找，一直找到开头  j -= siz; // 往前一个  if (!(cmp(j, i) > 0)) { // 如果任何前面的元素 不大于 i元素  j += siz; // 找到了，j 后移，这就是插入位置  break;  }  }  for (k = i; k > j; k -= siz) { // 从i 到 j  swp(k, k - siz); // 两两交换，让 i 冒泡上来  }  }    // 从哨兵位置，到最后（哨兵后面这一段每次跳2个来比较）  for (i = sentry; i < end; i += siz) {  j = i - siz; // 前面一个元素  if (!(cmp(j, i) > 0)) { // 如果前面的不大于后面的  continue; // 下一个  }  do { // 如果前面的大于后面的  j -= siz2; // 向前跳两个  // 前面的 不大于 i（ 这里是插入位置，要比较2个元素）  if (!(cmp(j, i) > 0)) {  j += siz; // 后移一个  if (!(cmp(j, i) > 0)) { // 后一个还不大于 i  j += siz; // 再后移一个（因为前面跳了2个）  }  break; // 跳出  }  // 如果在中间没找到位置  if (j == start) { // 如果到开头  break; // 跳出  }  if (j == start + siz) { // 如果是开头第2个  j -= siz; // 第一个（还是要和开头的比较一下，因为没比过）  if (cmp(i, j) > 0) { // i大于j, 不需要移到开头。  j += siz; // 右移（插在1，2号元素中间）  }  break; // 跳出  }  } while (1);  for (k = i; k > j; k -= siz) { // 从i 到 j  swp(k, k - siz); // 两两交换，让i冒泡上去  }  }  }  break;  }  } |

如上所示，当元素数在5个以内时，有几个提升性能的优化算法。

排序2个元素的优化算法：

|  |
| --- |
| static inline void zend\_sort\_2(void \*a, void \*b, compare\_func\_t cmp, swap\_func\_t swp) {  if (cmp(a, b) > 0) { // 如果第一个大于第二个  swp(a, b); // 交换位置  }  } |

排序3个元素的优化算法：

|  |
| --- |
| static inline void zend\_sort\_3(void \*a, void \*b, void \*c, compare\_func\_t cmp, swap\_func\_t swp){  if (!(cmp(a, b) > 0)) { // 第一个不大于第二个，前两个不用动  if (!(cmp(b, c) > 0)) { // 第二个不大于第三个，后两个不用动，直接返回  return;  }  swp(b, c); // 第二个大于第三个，后两个交换  if (cmp(a, b) > 0) { // 交换后第一个大于第二个  swp(a, b); // 前两个交换  }  return; // 完毕  }  if (!(cmp(c, b) > 0)) { // 第一个大于第二个，且 第二个大于等于第三个（一个比一个大）  swp(a, c); // 交 1，3 二个元素  return;  }  // 第一个大于第二个，且第二个小于第三个（后面两个不用换）  swp(a, b); // 先交换前两个  if (cmp(b, c) > 0) { // 如果交换后第二个大于第三个  swp(b, c); // 交换后两个  }  } |

排序4个元素的优化算法：

|  |
| --- |
| static void zend\_sort\_4(void \*a, void \*b, void \*c, void \*d, compare\_func\_t cmp, swap\_func\_t swp) {  zend\_sort\_3(a, b, c, cmp, swp); // 先排前3个  if (cmp(c, d) > 0) { // 如果第三个大于第4个, 让第四个冒泡上去  swp(c, d);  if (cmp(b, c) > 0) {  swp(b, c);  if (cmp(a, b) > 0) {  swp(a, b);  }  }  }  } |

排序5个元素的优化算法：

|  |
| --- |
| static void zend\_sort\_5(void \*a, void \*b, void \*c, void \*d, void \*e, compare\_func\_t cmp, swap\_func\_t swp) {  zend\_sort\_4(a, b, c, d, cmp, swp); // 先排序前4个  if (cmp(d, e) > 0) { // 如果第五个大于第四个，让第五个冒泡上去  swp(d, e);  if (cmp(c, d) > 0) {  swp(c, d);  if (cmp(b, c) > 0) {  swp(b, c);  if (cmp(a, b) > 0) {  swp(a, b);  }  }  }  }  } |

### 快速排序

快速排序是所有排序算法中举足轻重的一个，有O(log(2)n)的时间复杂度，性能优异，它也是最常用的排序算法之一。PHP中的快速排序使用的是从LLVM的c++类库中选取的函数，在PHP中，函数名为zend\_sort()，代码注释如下：

|  |
| --- |
| ZEND\_API void zend\_sort(void \*base, size\_t nmemb, size\_t siz, compare\_func\_t cmp, swap\_func\_t swp) {  while (1) { // 死循环  if (nmemb <= 16) { // 如果少于等于16个，只有这里有return  // 使用插入排序  zend\_insert\_sort(base, nmemb, siz, cmp, swp);  return; // 这里是唯一出口  } else { // 大于16个，采用快速排序  char \*i, \*j;  char \*start = (char \*)base; // 开始位置  // 结束 位置（这里没有元素，end 不会直接使用）  char \*end = start + (nmemb \* siz);  // 偏移量 = 元素数量/2，向下取整  size\_t offset = (nmemb >> Z\_L(1));  // 中心点（如果17个元素，会指向第9个，18个元素，也指向第9个）  char \*pivot = start + (offset \* siz);  // 右移10位，如果还有效 （数量大于1024个元素，已测试）。采用5段比较。  if ((nmemb >> Z\_L(10))) {  // 1/4位置  size\_t delta = (offset >> Z\_L(1)) \* siz;  // 开始位置 1/4位置 1/2位置 3/4位置 结尾位置，比较这5个位置  zend\_sort\_5(start, start + delta, pivot, pivot + delta, end - siz, cmp, swp);  } else { // 否则 采用3段比较。  // 开始位置 1/2位置 结尾位置，比较这3个位置  zend\_sort\_3(start, pivot, end - siz, cmp, swp);  }  // 前面这一段是为了保证中间数尽量大小适中，这个很重要，可以有效提高快排效率。  // 这样选出的中心点一定有一个元素比它大，也一定有一个元素比它小（或相等），它不需要被排到两端。    // 交换第2个元素和中心点 （第一个不用动了，前面刚刚排序过了，一定比中心点小（或等））  // 中心点先放在固定位置，最后移动到正确的位置。  swp(start + siz, pivot);  // 修正中心点指针（还是指向原来的那个元素）  pivot = start + siz;    // 这是真正需要排序的部分，第3个元素到最后一个。  i = pivot + siz; // 中心向右1个（第三个元素）  j = end - siz; // 最后1个    while (1) { // 死循环，只有goto才会跳出  // 跳过左边比中心点小的元素（这些元素不用移动）  while (cmp(pivot, i) > 0) { // 中点大于i  i += siz; // i右移  // 位置交叉（不需要交换），不需要和结尾点比较，因为前面选中心点时保证了，结尾一定比中心点大。  if (UNEXPECTED(i == j)) {  goto done; // 排序完成  }  }  j -= siz; // j左移  if (UNEXPECTED(j == i)) { // 位置交叉（不需要交换）  goto done; // 排序完成  }  // 跳过右边比中心点大的元素（这些元素不用移动）  while (cmp(j, pivot) > 0) { // j大于中点  j -= siz; // j左移  if (UNEXPECTED(j == i)) { // 位置交叉（不需要交换）  goto done; // 排序完成  }  }  // 这时候找到了左边第一个比中心点大（等）的，和右边第一个比中心点小（等）的  swp(i, j); // 交换i,j  i += siz; // i 右移  if (UNEXPECTED(i == j)) { // 位置交叉（不需要交换）  goto done; // 排序完成  }  }  // 这轮操作死循环产生的结果是，【最后交叉的这个点】的 左边没有比 pivot 大的，右边没有比 pivot 小的元素。  done:  // 把交叉点左边一个元素存放中心点。中心点的位置一定要正确，这个很重要。  // 因为i是先找的，这时候它要么是需要移位的元素，要么是结尾元素，动它左边的元素肯定不会有问题。  // 把这个元素和pivot交换，是为了把pivot位置放对，这个元素存放的位置还只是临时位置，后面会继续排，  // 但它肯定在交叉点左边，就不会错了。  swp(pivot, i - siz);  // 这里的思路是：交叉两边元素少的一边递归，多的一边继续快速排序。    // 这时候 i-siz 是中心点。  // 如果中心点左边的元素 没有右边的多。中心点 - 开始 < 结尾 -i  if ((i - siz) - start < end - i) {  // 递归：排序左边一段  zend\_sort(start, (i - start)/siz - 1, siz, cmp, swp);  base = i; // 右边一半继续循环排序  nmemb = (end - i)/siz; // 数量 = 后面留下的部分，这部分要重新排。  // 如果中心点左边的元素 比右边的多。中心点 - 开始 >= 结尾 -i  } else {  // 递归：排序右边一段  zend\_sort(i, (end - i)/siz, siz, cmp, swp);  // 数量 = i -1（-1是因为 中心点 不用排了）  nmemb = (i - start)/siz - 1;  }  }  }  } |

## 六）常用术语

大量引用：指函数和宏程序有大于等于5个文件引用。

少量引用：指函数和宏程序有少于5个文件引用。

顺序数组的元素：指顺序数组的zval元素。

哈希表的元素：指哈希表的Bucket元素。

哈希表的元素值：指哈希表的Bucket元素中的val元素，它本身是一个zval实例。

键名：Bucket实例中的key指针指向的zend\_string实例。

# 结语

数组是PHP中非常关键的部分，代码量大，提供的功能也很丰富。其中顺序数组相关的内容大多比较简单，哈希表相关的内容要复杂很多，但只要理清哈希表的数据结构和最基本的增删改查元素的算法，其他内容也会比较容易理解。

对于最复杂的\_zend\_hash\_index\_add\_or\_update\_i()函数等相关内容，最好能尝试运行调试几次，有助于加深理解。

# 未解疑问

zend\_array\_dup\_value()函数中的复杂判断是什么？zend\_symtable\_to\_proptable()里面也用到。

# Change Log

2025.1.8 创建

2025.1.9 基本概念，整理一些函数

2025.1.14 顺序数组中追加和遍历元素

2025.1.15 顺序数组中更新、查找元素

2025.1.16 顺序数组删除元素、未初始化数组中插入元素，批量添加元素

2025.1.17 遍历都over，只差一个尾删除ZEND\_HASH\_MAP\_FOREACH\_END\_DEL，哈希表增加元素弄了一半。

2025.1.18 遍历尾删除ZEND\_HASH\_MAP\_FOREACH\_END\_DEL完成。发现i\_zval\_ptr\_dtor应该写在类型篇里。迭代器弄了一半，兼容性程序弄了一些。

2025.1.19 zend\_hash\_rehash()函数完毕。整理符号表，做到一半。

2025.1.21 内部指针over

2025.1.22 哈希表添加元素，业务逻辑分支表

2025.1.23 哈希表的增加和修改，zend\_array\_dup()函数

2025.1.24 销毁数组，计算元素数量。完成了一些零散函数的注释，准备过年

2025.2.01 过完年。哈希表中通过索引号进行操作

2025.2.02 通过索引号进行操作over。内容添加完，开始进行第二阶段整理，预计1周。哈希表结构示意图。简单处理完未缠目。

2025.2.03 整理，解决未解疑问，[A-Z][A-Z0-9\_]+正则用于搜索常量和宏程序。测试zend\_hash\_index\_add\_new()等函数。

2025.2.04 添加附录术语表。函数引用数先不加了，因为有提供源码的函数都比较重要。整体检查，修订。review到40页。

2025.2.04 继续review112页。

2025.2.08 初步完成。因为排序算法又增加了8页代码注释，本来不想放这么多代码，但排序算法很重要。tudo:比较函数有必要加他业务逻辑分支图。这一篇用了整整1个月时间。