**HashTable和Bucket**

typedef struct \_hashtable {

uint nTableSize;

uint nTableMask;

uint nNumOfElements;

ulong nNextFreeElement;

Bucket *pInternalPointer;*

*Bucket*pListHead;

Bucket *pListTail;*

*Bucket \**arBuckets;

dtor\_func\_t pDestructor;

zend\_bool persistent;

unsigned char nApplyCount;

zend\_bool bApplyProtection;

if ZEND\_DEBUG int inconsistent;

} HashTable;

nNumOfElements

标识现在存储在数组里面的值的数量。这也是函数count的返回值

nTableSize

表示哈希表的容量。它通常是下一个大于等于nNumOfElements的2的幂值。比如，如果数组存储了32元素，那么哈希表也是32大小的容量。但如果再多一个元素添加进来，也就是说，数组现在有33个元素，那么哈希表的容量就被调整为64。 这是为了保持哈希表在空间和时间上始终有效。很明显，如果哈希表太小，那么将会有很多的冲突，而且性能也会降低。另一方面，如果哈希表太大，那么浪费内存。2的幂值是一个很好的折中方案。

nTableMask

是哈希表的容量减一。这个mask用来根据当前的表大小调整生成的哈希值。例如，”foo”真正的哈希值（使用DJBX33A哈希函数）是193491849。如果我们现在有64容量的哈希表，我们明显不能使用它作为数组的下标。取而代之的是通过应用哈希表的mask，然后只取哈希表的低位。

hash | 193491849 | 0b1011100010000111001110001001

& mask | & 63 | & 0b0000000000000000000000111111

= index | = 9 | = 0b0000000000000000000000001001

nNextFreeElement

是下一个可以使用的数字键值，当你使用$array[] = xyz是被使用到。

pInternalPointer

存储数组当前的位置。这个值在foreach遍历时可使用reset()，current()，key()，next()，prev()和end()函数访问。

pListHead和pListTail

标识了数组的第一个和最后一个元素的位置。记住：PHP的数组是有序集合。比如，[‘foo’ => ‘bar’, ‘bar’ => ‘foo’]和[‘bar’ => ‘foo’, ‘foo’ => ‘bar’]这两个数组包含了相同的元素，但却有不同的顺序。

arBuckets

是我们经常谈论的“哈希表（internal C array）”。它用Bucket \*\*来定义，因此它可以被看作数组的bucket指针（我们会马上谈论Bucket是什么）。

pDestructor

是值的析构器。如果一个值从HT中移除，那么这个函数会被调用。常见的析构函数是zval\_ptr\_dtor。zval\_ptr\_dtor会减少zval的引用数量，而且，如果它遇到o，它会销毁和释放它。

typedef struct bucket {

ulong h;

uint nKeyLength;

void *pData;*

*void*pDataPtr;

struct bucket *pListNext;*

*struct bucket*pListLast;

struct bucket *pNext;*

*struct bucket*pLast;

const char \*arKey;

} Bucket;

h

是一个哈希值（没有应用mask值映射之前的值）。

arKey

用来保存字符串键值。

nKeyLength

是对应的长度。如果是数字键值，那么这两个变量都不会被使用。

pData

及

pDataPtr

被用来存储真正的值。对PHP数组来说，它的值是一个zval结构体（但它也在其他地方使用到）。不要纠结为什么有两个属性。它们两者的区别是谁负责释放值。

pListNext

和

pListLast

标识数组元素的下一个元素和上一个元素。如果PHP想顺序遍历数组它会从pListHead这个bucket开始（在HashTable结构里面），然后使用pListNext bucket作为遍历指针。在逆序也是一样，从pListTail指针开始，然后使用pListLast指针作为变量指针。（你可以在用户代码里调用end()然后调用prev()函数达到这个效果。）

pNext

和

pLast

生成我上面提到的“可能冲突的值链表”。arBucket数组存储第一个可能值的bucket。如果该bucket没有正确的键值，PHP会查找pNext指向的bucket。它会一直指向后面的bucket直到找到正确的bucket。pLast在逆序中也是一样的原理。

你可以看到，PHP的哈希表实现相当复杂。这是它使用超灵活的数组类型要付出的代价。

**哈希表是怎么被使用的？**

Zend Engine定义了大量的API函数供哈希表使用。低级的哈希表函数预览可以在

zend\_hash.h文件里面找到。

另外Zend Engine在zend\_API.h文件定义了稍微高级一些的API。

我们没有足够的时间去讲所有的函数，但是我们至少可以查看一些实例函数，看看它是如何工作的。我们将使用array\_fill\_keys作为实例函数。

使用第二部分提到的技巧你可以很容易地找到函数在ext/standard/array.c文件里面定义了。

现在，让我们来快速查看这个函数。

跟大部分函数一样，函数的顶部有一堆变量的定义，然后调用zend\_parse\_parameters

函数：

zval *keys,*val, \*\*entry;

HashPosition pos;

if (zend\_parse\_parameters(ZEND\_NUM\_ARGS() TSRMLS\_CC, "az", &keys, &val) == FAILURE) {

return;

}

很明显，az参数说明第一个参数类型是数组（即变量keys），第二个参数是任意的zval（即变量val）。

解析完参数后，返回数组就被初始化了：

array\_init\_size(return\_value,zend\_hash\_num\_elements(Z\_ARRVAL\_P(keys));

这一行包含了array API里面存在的三步重要的部分：

1. Z\_ARRVAL\_P宏从zval里面提取值到哈希表。
2. zend\_hash\_num\_elements提取哈希表元素的个数（nNumOfElements属性）。
3. array\_init\_size使用size变量初始化数组。

因此，这一行使用与键值数组一样大小来初始化数组到return\_value变量里。

这里的size只是一种优化方案。

函数也可以只调用array\_init(return\_value)，这样随着越来越多的元素添加到数组里，PHP就会多次重置数组的大小。

通过指定特定的大小，PHP会在一开始就分配正确的内存空间。

数组被初始化并返回后，函数用跟下面大致相同的代码结构，使用while循环变量keys数组：

zend\_hash\_internal\_pointer\_reset\_ex(Z\_ARRVAL\_P(keys), &pos);

while (zend\_hash\_get\_current\_data\_ex(Z\_ARRVAL\_P(keys), (void \*\*)&entry, &pos) == SUCCESS) {

zend\_hash\_move\_forward\_ex(Z\_ARRVAL\_P(keys), &pos);

}

这可以很容易地翻译成PHP代码：

reset($keys);

while (null !== $entry = current($keys)) {

next($keys);

}

跟下面的一样：

foreach ($keys as $entry) {

// some code

}

唯一不同的是，C的遍历并没有使用内部的数组指针，而使用它自己的pos变量来存储当前的位置。

在循环里面的代码分为两个分支：一个是给数字键值，另一个是其他键值。数字键值的分支只有下面的两行代码：

zval\_add\_ref(&val);

zend\_hash\_index\_update(Z\_ARRVAL\_P(return\_value),

Z\_LVAL\_PP(entry), &val,

sizeof(zval \*), NULL);

这看起来太直接了：首先值的引用增加了（添加值到哈希表意味着增加另一个指向它的引用），然后值被插入到哈希表中。zend\_hash\_index\_update宏的参数分别是，需要更新的哈希表Z\_ARRVAL\_P(return\_value)，整型下标

Z\_LVAL\_PP(entry)，值&val，值的大小sizeof(zval \*)以及目标指针(这个我们不关注，因此是NULL）。

非数字下标的分支就稍微复杂一点：

zval key, *key\_ptr =*entry;

if (Z\_TYPE\_PP(entry) != IS\_STRING) {

key = *\*entry;*

*zval\_copy\_ctor(&key);*

*convert\_to\_string(&key);*

*key\_ptr = &key;*

*}*

*zval\_add\_ref(&val);*

*zend\_symtable\_update(Z\_ARRVAL\_P(return\_value), Z\_STRVAL\_P(key\_ptr), Z\_STRLEN\_P(key\_ptr) + 1, &val, sizeof(zval*), NULL);

if (key\_ptr != \*entry) {

zval\_dtor(&key);

}

首先，使用convert\_to\_string将键值转换为字符串（除非它已经是字符串了）。在这之前，entry被复制到新的key变量。key = \*\*entry这一行实现。另外，

zval\_copy\_ctor函数会被调用，不然复杂的结构（比如字符串或数组）不会被正确地复制。

上面的复制操作非常有必要，因为要保证类型转换不会改变原来的数组。如果没有copy操作，强制转换不仅仅修改局部的变量，而且也修改了在键值数组中的值（显然，这对用户来说非常意外）。

显然，循环结束之后，复制操作需要再次被移除，zval\_dtor(&key)

做的就是这个工作。zval\_ptr\_dtor和zval\_dtor的不同是zval\_ptr\_dtor只会在refcount变量为0时销毁zval变量，而zval\_dtor会马上销毁它，而不是依赖

refcount的值。这就为什么你看到zval\_pte\_dtor使用”normal”变量而zval\_dtor

使用临时变量，这些临时变量不会在其他地方使用。而且，zval\_ptr\_dtor

会在销毁之后释放zval的内容而zval\_dtor不会。因为我们没有malloc()任何东西，因此我们也不需要free()，因此在这方面，zval\_dtor做了正确的选择。

现在来看看剩下的两行（重要的两行^^）：

zval\_add\_ref(&val);

zend\_symtable\_update(Z\_ARRVAL\_P(return\_value), Z\_STRVAL\_P(key\_ptr), Z\_STRLEN\_P(key\_ptr) + 1, &val, sizeof(zval \*), NULL);

这跟数字键值分支完成后的操作非常相似。不同的是，现在调用的是

zend\_symtable\_update而不是zend\_hash\_index\_update，而传递的是键值字符串和它的长度。