每个php变量存在一个叫"zval"的变量容器中。一个zval变量容器，除了包含变量的类型和值，还包括两个字节的额外信息。第一个是"is\_ref"，是个bool值，用来标识这个变量是否是属于引用集合(reference set)。通过这个字节，php引擎才能把普通变量和引用变量区分开来，由于php允许用户通过使用&来使用自定义引用，zval变量容器中还有一个内部引用计数机制，来优化内存使用。第二个额外字节是"refcount"，用以表示指向这个zval变量容器的变量(也称符号即symbol)个数。

<?php  
$a = "new string";  
?>

新的**变量a**，是在当前作用域中生成的。并且生成了类型为 string 和值为new string的**变量容器**。在额外的两个字节信息中，"is\_ref"被默认设置为 FALSE，因为没有任何自定义的引用生成。"refcount" 被设定为 1，因为这里只有一个变量使用这个变量容器.

<?php  
$a = "new string";  
$b = $a;  
xdebug\_debug\_zval( 'a' );  
?>

以上例程会输出：

a: (refcount=2, is\_ref=0)='new string'

这时，引用次数是2，因为同一个变量容器被变量 a 和变量 b关联.当没必要时，php不会去复制已生成的变量容器。变量容器在”refcount“变成0时就被**销毁**. 当任何关联到某个变量容器的变量离开它的作用域(比如：函数执行结束)，或者对变量调用了函数 [unset()](http://php.net/manual/zh/function.unset.php)时，”refcount“就会减1.

当考虑像 [array](http://php.net/manual/zh/language.types.array.php)和[object](http://php.net/manual/zh/language.types.object.php)这样的复合类型时，事情就稍微有点复杂. 与 标量(scalar)类型的值不同，[array](http://php.net/manual/zh/language.types.array.php)和 [object](http://php.net/manual/zh/language.types.object.php)类型的变量把它们的成员或属性存在自己的符号表中。这意味着下面的例子将生成三个zval变量容器。

Example #5 Creating a [array](http://php.net/manual/zh/language.types.array.php) zval

<?php  
$a = array( 'meaning' => 'life', 'number' => 42 );  
xdebug\_debug\_zval( 'a' );  
?>

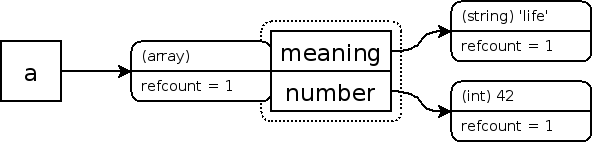
以上例程的输出类似于：

a: (refcount=1, is\_ref=0)=array (

'meaning' => (refcount=1, is\_ref=0)='life',

'number' => (refcount=1, is\_ref=0)=42

)



把数组作为一个元素添加到自己

<?php  
$a = array( 'one' );  
$a[] =& $a;  
xdebug\_debug\_zval( 'a' );  
?>

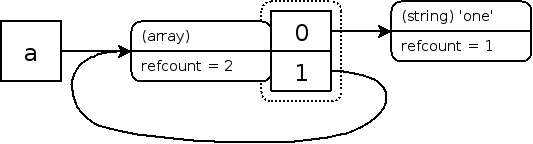
以上例程的输出类似于：

a: (refcount=2, is\_ref=1)=array (

0 => (refcount=1, is\_ref=0)='one',

1 => (refcount=2, is\_ref=1)=...

)



能看到数组变量 (a) 同时也是这个数组的第二个元素(1) 指向的变量容器中“refcount”为 *2*。上面的输出结果中的"..."说明发生了递归操作, 显然在这种情况下意味着"..."指向原始数组。

跟刚刚一样，对一个变量调用unset，将删除这个符号，且它指向的变量容器中的引用次数也减1。所以，如果我们在执行完上面的代码后，对变量$a调用unset, 那么变量 $a 和数组元素 "1" 所指向的变量容器的引用次数减1, 从"2"变成"1". 下例可以说明:

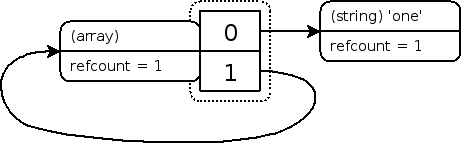
Example #9 Unsetting $a

(refcount=1, is\_ref=1)=array (

0 => (refcount=1, is\_ref=0)='one',

1 => (refcount=1, is\_ref=1)=...

)



**重点：变量指向的是容器**

尽管不再有某个作用域中的任何符号指向这个结构(就是变量容器)，由于数组元素“1”仍然指向数组本身，所以这个容器不能被清除 。因为没有另外的符号指向它，用户没有办法清除这个结构，结果就会导致**内存泄漏**。庆幸的是，php将在**脚本执行结束时清除这个数据结构**，但是在php清除之前，将耗费不少内存。如果你要实现分析算法，或者要做其他像一个子元素指向它的父元素这样的事情，这种情况就会经常发生。当然，同样的情况也会发生在对象上，实际上对象更有可能出现这种情况，因为对象总是隐式的被引用。

长时间处理大数据的脚本时这种内存泄漏会非常严重，5.3开始已经解决，小于5.3可以用父进程开启小的子进程处理数据，然后分批处理完把子进程杀掉就可以释放内存。

5.3开始采用了新的gc机制避免了循环引用的内存泄漏问题。

原理：

如果一个zval的refcount减少之后大于0，那么此zval还不能被释放，此zval可能成为一个垃圾（可能根）

A：为了避免每次变量的refcount减少的时候都调用GC的算法进行垃圾判断，此算法会先把所有前面准则3情况下的zval节点放入一个节点(root)缓冲区(root buffer)，并且将这些zval节点标记成紫色，同时算法必须确保每一个zval节点在缓冲区中之出现一次。当缓冲区被节点**塞满**的时候，GC才开始开始对缓冲区中的zval节点进行垃圾判断。

B：当缓冲区满了之后，算法以深度优先对**数组每一个节点**所包含的zval进行减1操作，为了确保不会对同一个zval的refcount重复执行减1操作，一旦zval的refcount减1之后会将zval标记成灰色。需要强调的是，这个步骤中，起初节点zval本身不做减1操作，但是如果节点zval中包含的zval又指向了节点zval（环形引用），那么这个时候需要对节点zval进行减1操作。

C：算法再次以深度优先判断每一个节点包含的zval的值，如果zval的refcount等于0，那么将其标记成白色(代表垃圾)，如果zval的refcount大于0，那么将对此zval以及其包含的zval进行refcount加1操作，这个是对非垃圾的还原操作，同时将这些zval的颜色变成黑色（zval的默认颜色属性）

D：遍历zval节点，将C中标记成白色的节点zval释放掉。

问题：当垃圾回收机制关闭时，如果根缓冲区存满了可能根，更多的可能根显然不会被记录。那些没被记录的可能根，将不会被这个算法来分析处理。如果他们是循环引用周期的一部分，将永不能被清除进而导致内存泄漏。