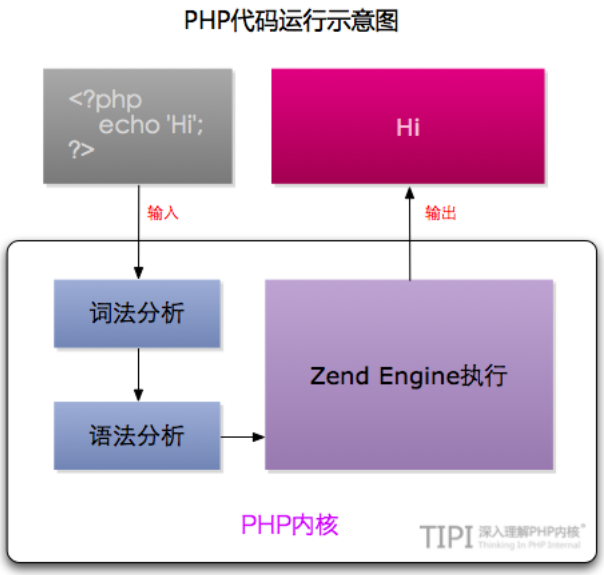
一段PHP代码的学习

创建： 张状，最新修改： 5分钟前


```php
<?php
   echo "Hello World";
   $a = 1 + 1;
   echo $a;
?>
```

运行php test.php会发生什么?



1、PHP生命周期

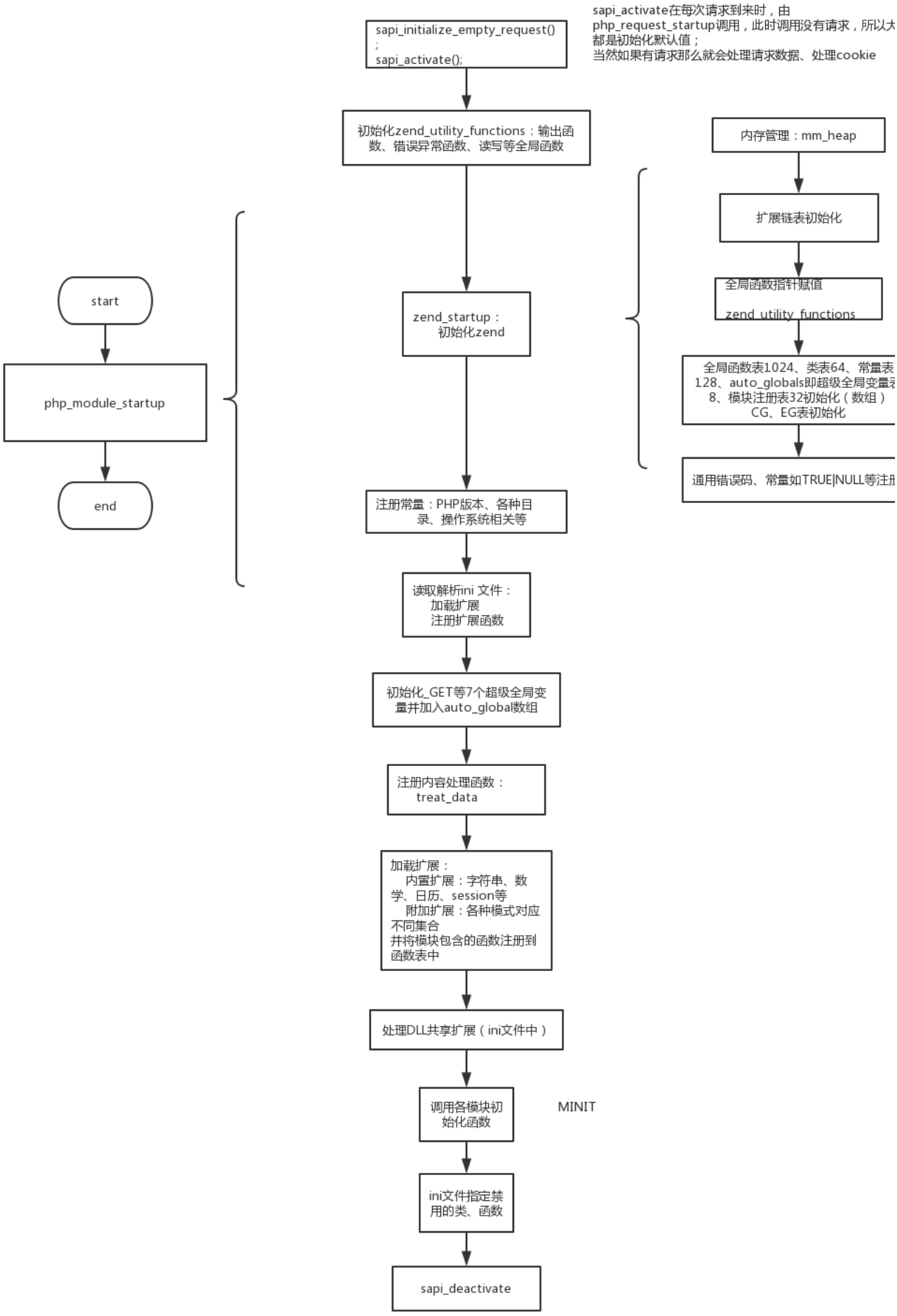
1.1 SAPI

- server application programming interface，PHP服务器端应用的编程接口
- apache（mod_php5）、CGI、IIS（ISAPI）、shell（CLI）等

正如在PC上无论安装哪些OS，只要这些OS满足PC的接口规范就可以运行，同理PHP脚本的执行可以有很多方式如web服务器、命令行、嵌入式等等，因此在PHP中需要一个统一的和据的接口----SAPI。

实现一个SAPI首先需要定义sapi_module_struct结构体（PHP-SRC/sapi/cgi/cgi_main.c:）：入口函数、请求处理函数、“析构函数”、ZEND读数据接口、ZEND写数据接口、常量如名字等

- php_cgi_startup入口函数：请求、shell调用PHP时的入口函数，
  - 一般都是调用php_module_startup进行初始化处理

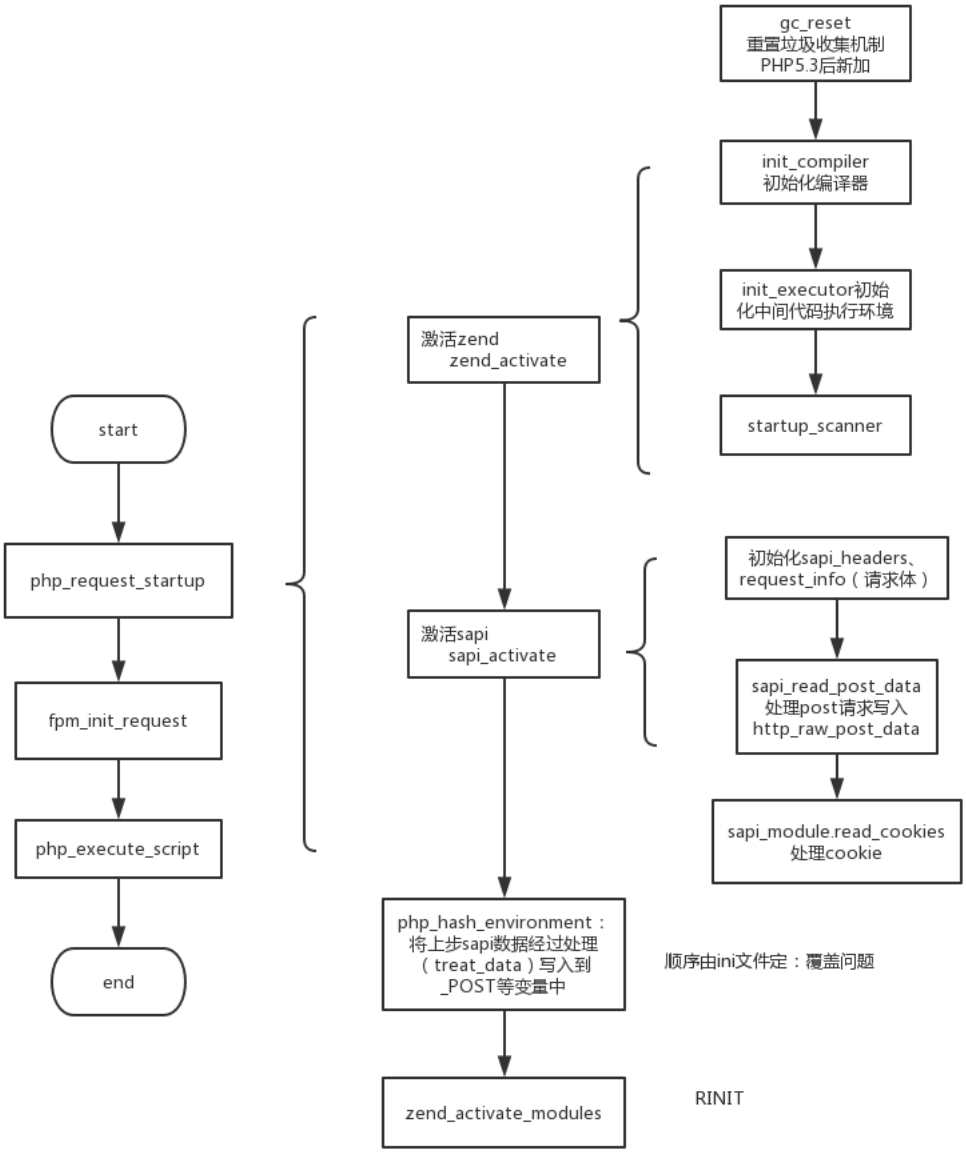


- sapi_cgi_activate请求处理函数
- sapi_cgi_deactivate请求收尾接口
- sapi_cgi_read_post读取post数据接口（CGI是stdin）
- sapi_cgi_read_cookies读取cookie接口

- sapi_cgi_register_variables对_SERVER添加变量

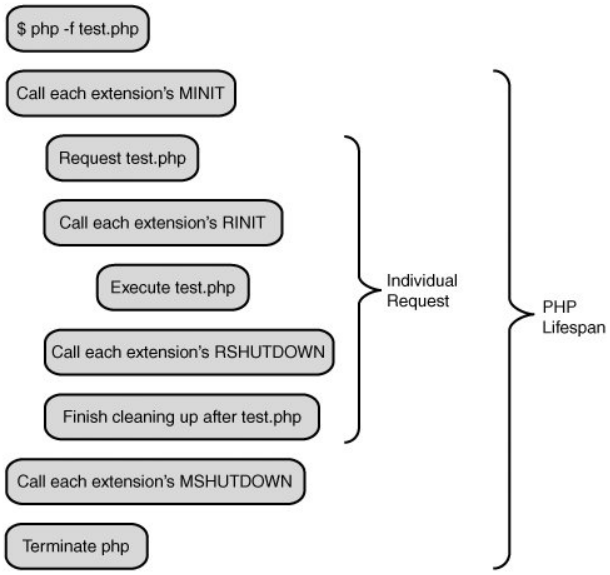
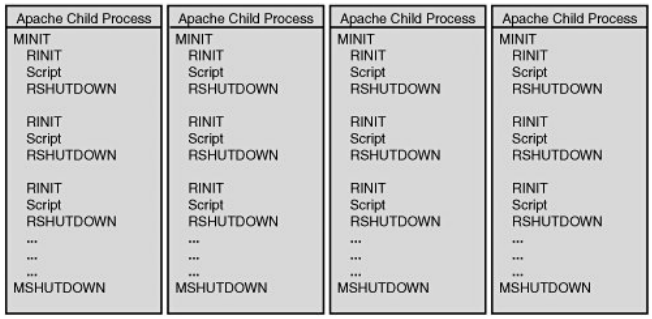
每当有请求时，启动每个模式对应的main函数：

- php_request_startup初始化请求
- php_execute_script执行脚本



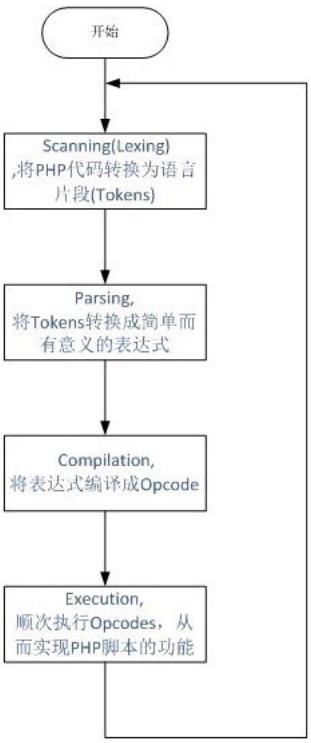
一次请求都是从sapi执行开始，然后执行php逻辑，整个过程的生命周期如下图所示：

单进程单线程	多进程	多线

单进程单线程	多进程	多线
		

1.2 脚本执行

- scanning（lex）即词法分析，将PHP代码转化为token即语言片段
  - php使用re2c，mysql使用flex。
- parsing将token转化为有意义的表达式
- compile将表达式编译为opcodes
- execution执行



1.2.1 词法分析

回到我们最开始的例子中，当运行“ php test.php ”后，经过各种初始化操作，进入脚本执行阶段，通过PHP4.2之后提供的方法“ token_get_all ” 可以获得所有的token。

代码事例	输出token

代码事例	输出token

```php
<?php
echo"<pre>";
$phpcode = '<?php  echo "Hello World!";  $a = 1 + 1;  echo $a;  ?>';
// $tokens = token_get_all($phpcontent);
// print_r($tokens);
$tokens = token_get_all($phpcode);
foreach ($tokens as $key => $token) {
    $tokens[$key][0] = token_name($token[0]);
}
print_r($tokens);
?>
```

```
[2] => Array
    (
        [0] => T_ECHO
        [1] => echo
        [2] => 1
    )

[3] => Array
    (
        [0] => T_WHITESPACE
        [1] =>
        [2] => 1
    )

[4] => Array
    (
        [0] => T_CONSTANT_ENCAPSED_STRING
        [1] => "Hello World!"
        [2] => 1
    )
```

可以发现，代码中所有的字符串、空格、关键字、变量名等都会原样解析出来。

1.2.2 解析token

- 剔除多余空格
- 使用bison生成分析器文件，在PHP通过调用lex_scan，得到一个一个简单表达式

1.2.3 编译

获得opcode，保存在zend_op结构体中，所有opcodes保持在zend_op_array中

```
struct _zend_op {
    opcode_handler_t handler; // 执行该opcode时调用的处理函数
    znode result;
    znode op1;
    znode op2;
    ulong extended_value;
    uint lineno;
    zend_uchar opcode; // opcode代码
};

//_zend_compiler_globals 编译时信息，包括函数表等
zend_compiler_globals    *compiler_globals;
//_zend_executor_globals 执行时信息
zend_executor_globals    *executor_globals;
//_php_core_globals 主要存储php.ini内的信息
php_core_globals         *core_globals;
//_sapi_globals_struct SAPI的信息
sapi_globals_struct      *sapi_globals;
```

1.2.4 执行

执行op_array即可。PHP有三种方式来进行opcode的处理：CALL，SWITCH和GOTO。

PHP默认使用CALL的方式，也就是函数调用的方式， 由于opcode执行是每个PHP程序频繁需要进行的操作，可以使用SWITCH或者GOTO的方式来分发， 通常GOTO的效率相对会高一些，提高依赖于不同的CPU。

1.2.4.1 执行上下文

在执行的过程中，变量名及指针主要存储于_zend_executor_globals的符号表中，_zend_executor_globals的结构这样的：

```

struct _zend_executor_globals {
    zval uninitialized_zval;
    zval error_zval;
    /* symbol table cache */
    zend_array *symtable_cache[SYMTABLE_CACHE_SIZE];
    zend_array **symtable_cache_limit;
    zend_array **symtable_cache_ptr;
    zend_array symbol_table;        /* main symbol table */
    HashTable included_files;   /* files already included */
    JMP_BUF *bailout;
    int error_reporting;
    int exit_status;

    struct _zend_execute_data *current_execute_data;
```

```
    HashTable *function_table;  /* function symbol table */
    HashTable *class_table;     /* class table */
    HashTable *zend_constants;  /* constants table */
    ...
};
struct _zend_execute_data {
    const zend_op       *opline;           /* executed opline               */
    zend_execute_data   *call;             /* current call                  */
    zval                *return_value;
    zend_function       *func;             /* executed function             */
    zval                 This;             /* this + call_info + num_args   */
    zend_execute_data   *prev_execute_data;
    zend_array          *symbol_table;
#if ZEND_EX_USE_RUN_TIME_CACHE
    void               **run_time_cache;   /* cache op_array->run_time_cache */
#endif
};
```

- symbol_table全局变量符号表，保存顶层作用域(即不在任何函数、对象等内)的变量
- _zend_execute_data 保存局部变量（symbol_table字段）
  - 随着执行函数变化，_zend_execute_data的symbol_table字段指向不同的符号表（函数调用时会初始化局部符号表）。
  - 其他函数的符号表放在栈中，当前函数调用完后会恢复之前的“ prev_execute_data ”。
  - 具体实现请参考源码Zend/zend_vm_execute.h：

```
/* Initialize execute_data */
execute_data = (zend_execute_data *)zend_vm_stack_alloc(
    sizeof(zend_execute_data) +
    sizeof(zval**) * op_array->last_var * (EG(active_symbol_table) ? 1 : 2) +
    sizeof(temp_variable) * op_array->T TSRMLS_CC);

EX(symbol_table) = EG(active_symbol_table);
EX(prev_execute_data) = EG(current_execute_data);
EG(current_execute_data) = execute_data;
```

- class_table类表
- zend_constants常量表
- function_table函数表

思考？静态变量去哪了？

2、 内存管理

关于PHP内部变量的实现请移步PHP7优化点-底层变量篇，在该分享中我详细介绍了PHP5和7在变量实现上的不同，并初步分析PHP7性能优化的几个点，最后给出PHP7中数组的内存管

回到我们刚开始的测试代码，对于变量a在php底层会使用zval结构体保存其值，但是我们发现zval结构体中并没有“ name ”字段，那么变量名是怎么和zval一一对应的呢？

2.1 变量和zval映射

- PHP中所有的变量都会保存在一个数组中（hash_table，符号表），同时PHP也是通过不同的数组来区分变量作用域的。
- 每当创建一个新变量时都会分配一个zval并写入其值，然后将变量名和指向该zval的指针保存在一个数组中。以后当你读取该变量时php就查找该数组获得对于zval

2.1.1 变量作用域

- php中通过一个“ _zend_executor_globals ”的结构体来保存执行相关的上下文信息（详见1.2.4.1）
- 变量的作用域通过不同的符号表来实现，所以现在我们可以理解在函数内通过使用global关键字时，为什么可以使用全局变量了。
  - 局部变量表里有一个指向全局变量表中变量的指针；

思考？1、如果在函数内global一个全局变量时，PHP是怎么处理的?

思考？2、global和_GLOBAL不同?

```
1
```

- PHP局部变量就是访问局部变量表，所以通过声明global复制到局部变量表中才可见。

2.1.2 内存管理

PHP的内存管理应该分为两个：

- 变量管理
- 内存管理

2.1.2.1 变量管理

对于变量管理主要体现在引用计数、写时复制等面向应用层的管理，我们在上个分享（php7优化点）已经涉及到以一部分，今天在具体看下PHP的垃圾回收机制：

2.1.2.1.1 PHP5.3之前

- PHP只有简单的基于引用计数的垃圾回收

```
$a = 42;        // $a -> zval_1(type=IS_LONG, value=42, refcount=1)$b = $a;        // $a, $b -> zval_1(type=IS_LONG, value=42, refcc
```

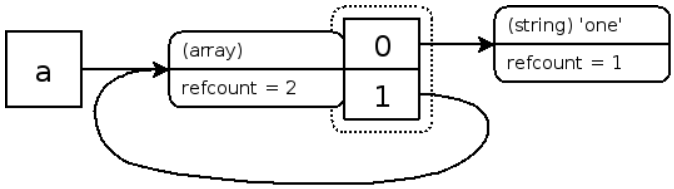
```
$c = $b;        // $a, $b, $c -> zval_1(type=IS_LONG, value=42, refcount=3)
    // 强制分离
$a += 1;        // $b, $c -> zval_1(type=IS_LONG, value=42, refcount=2)
                // $a -> zval_2(type=IS_LONG, value=43, refcount=1)
                // 因为a变了，所以新分配一个zval2给a
unset($b);      // $c -> zval_1(type=IS_LONG, value=42, refcount=1)
                // $a -> zval_2(type=IS_LONG, value=43, refcount=1)
unset($c);      // zval_1由于引用个数为0，被回收
                // $a -> zval_2(type=IS_LONG, value=43, refcount=1)
```

- 每个zval一个计数器，初始化时为1，以后每有一个新变量引用则加1，减少时减1。

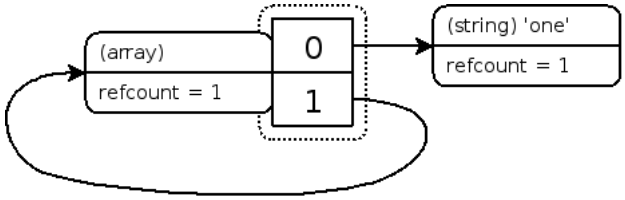
从上述代码可以看到，PHP5.3之前的垃圾回收决定性的指标是引用计数为0才可进行垃圾回收。那么如果存在循环引用的话怎么办?

```
<?php
    $a = array( 'one' );
    $a[] =& $a;
    xdebug_debug_zval( 'a' );
?>
// output
a: (refcount=2, is_ref=1)=array (
   0 => (refcount=1, is_ref=0)='one',
   1 => (refcount=2, is_ref=1)=...
)
```

上述结果可以画图展示为：能看到数组变量 a 同时也是这个数组的第二个元素指向的变量容器中 `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"..."



如果我们调用unset($a)后，将从符号表中删除这个引用，且它指向的 zval 的引用次数也减1：



尽管不再有任何作用域中的**任何**符号表中的引用指向这个 zval，由于数组元素 1 仍然指向数组本身，所以**这个 zval 不能被清除** 。因为没有另外的符号指向它，所以没有办法清除这个结构存泄漏。

2.1.2.1.2 PHP5.3之后

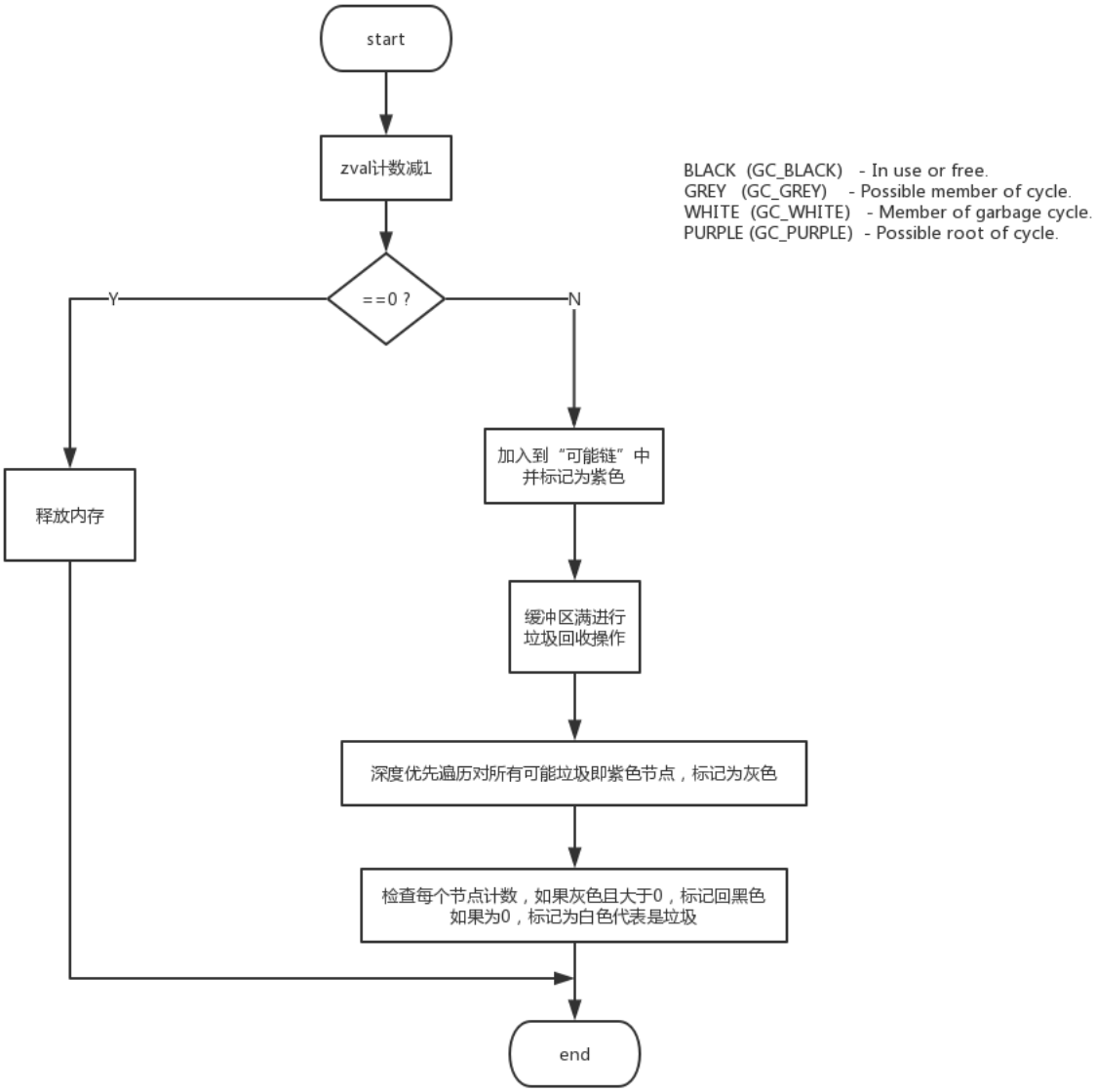
- php5.3之后垃圾回收算法还是以引用计数为基础，但是加上了同步回收的概念。这个算法由IBM的工程师在论文 Concurrent Cycle Collection in Reference Counted Systems 中提出回收机制中也有在使用该算法。传送门：垃圾回收算法

以下为网上资料整理，对论文调研完后接下来会定期更新。

新算法的核心思想为：

- 如果一个zval的引用计数增加，那么不是垃圾；
- 如果一个zval的引用计算减少到0，那么直接释放， 也不是垃圾；
- 如果一个zval的引用计算减少但是大于0，那么此zval还不能释放，可能是垃圾。加入到possible列表中。

所以我们仅仅需要对满足第三个准则的zval进行处理即可：



2.1.2.2 内存管理

zend引擎针对内存的操作封装了一层，用于替换直接的内存操作：malloc、free等，实现了更高效率的内存利用，其实现主要参考了tcmalloc的设计：emalloc、efree、estrdup等。

PHP内存池是内核中最底层的内存操作，定义了三种粒度的内存块：chunk、page、slot，每个chunk的大小为2M，page大小为4KB，一个chunk被切割为512个page，而一个或若干个page又被分为若干个slot，所以申请内存时按照不同的申请大小决定具体的分配策略：

- Huge(chunk): 申请内存大于2M，直接调用系统分配，分配若干个chunk
- Large(page): 申请内存大于3072B(3/4 page_size)，小于2044KB(511 page_size)，分配若干个page
- Small(slot): 申请内存小于等于3072B(3/4 page_size)，内存池提前定义好了30种同等大小的内存(8,16,24,32，...3072)，他们分配在不同的page上(不同大小的内存可能会分配在多个page上)，申请内存时直接在对应page上查找可用位置。
- chunk是zend引擎向OS申请内存的**唯一粒度**
- large、slot内存管理都是通过页来实现。
  - slot“切割”页即可；

I 主要数据结构

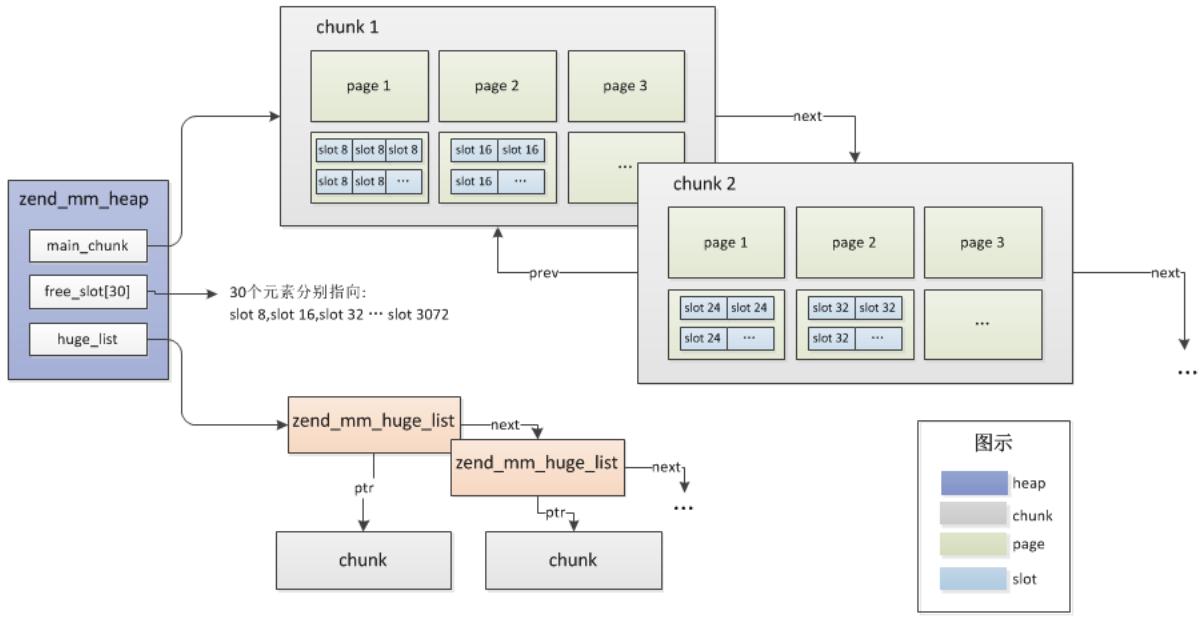
```
struct _zend_mm_heap {
#if ZEND_MM_STAT
    size_t             size; //当前已用内存数
    size_t             peak; //内存单次申请的峰值
#endif
    zend_mm_free_slot *free_slot[ZEND_MM_BINS]; // 小内存分配的可用位置链表，ZEND_MM_BINS等于30，即此数组表示的是各种大小内存对应的链表头部
    ...

    zend_mm_huge_list *huge_list;              //大内存链表，已分配的

    zend_mm_chunk     *main_chunk;             //指向chunk链表头部
    zend_mm_chunk     *cached_chunks;          //缓存的chunk链表
    int                chunks_count;           //已分配chunk数
    int                peak_chunks_count;      //当前request使用chunk峰值
    int                cached_chunks_count;    //缓存的chunk数
    double             avg_chunks_count;       //chunk使用均值，每次请求结束后会根据peak_chunks_count重新计算：(avg_chunks_count+peak_
}
```

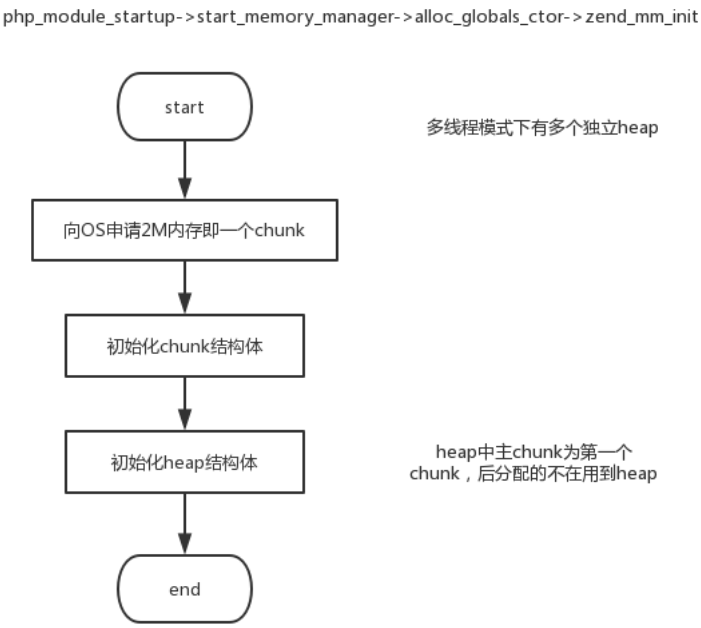
```
18
19  struct _zend_mm_chunk {
20      zend_mm_heap      *heap; //指向heap
21      zend_mm_chunk     *next; //指向下一个chunk
22      zend_mm_chunk     *prev; //指向上一个chunk
23      int                free_pages; //当前chunk的剩余page数
24      int                free_tail;              /* number of free pages at the end of chunk */
25      int                num;
26      char               reserve[64 - (sizeof(void*) * 3 + sizeof(int) * 3)];
27      zend_mm_heap       heap_slot; //heap结构，只有主chunk会用到
28      zend_mm_page_map   free_map; //标识各page是否已分配的bitmap数组，总大小512bit，对应page总数，每个page占一个bit位
29      zend_mm_page_info  map[ZEND_MM_PAGES]; //各page的信息：当前page使用类型(用于large分配还是small)、占用的page数等
30  };
31
32  //按固定大小切好的small内存槽
33  struct _zend_mm_free_slot {
34      zend_mm_free_slot *next_free_slot;//此指针只有内存未分配时用到，分配后整个结构体转为char使用
35  };
```

三者关系：

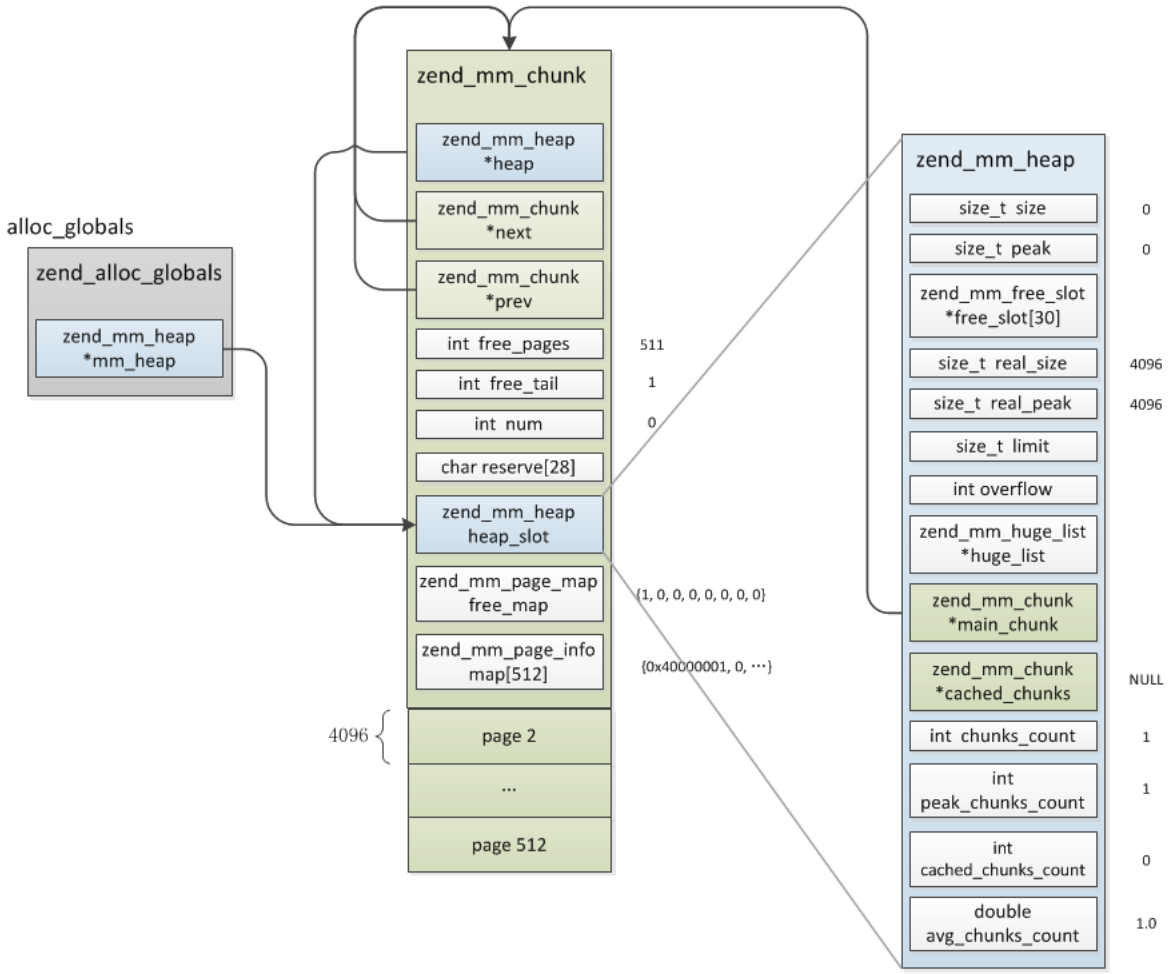


II 初始化过程

- heap中主chunk只有第一个chunk的heap会用到，后面分配的chunk不再用到heap。
- 另外没有做小内存切割slot；
- 每个chunk第一页保存chunk-header信息

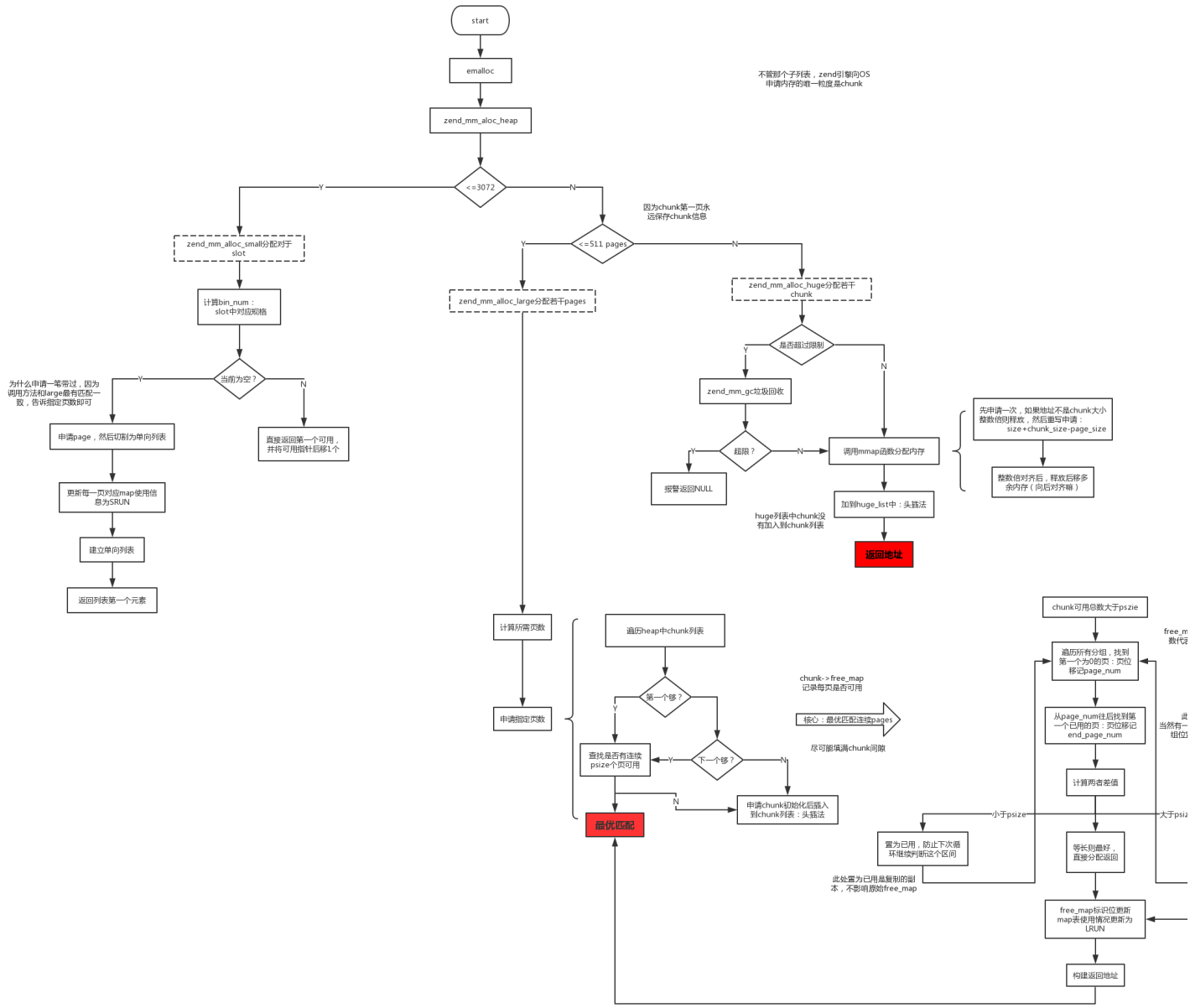


初始化后模型图如下：



III 内存分配

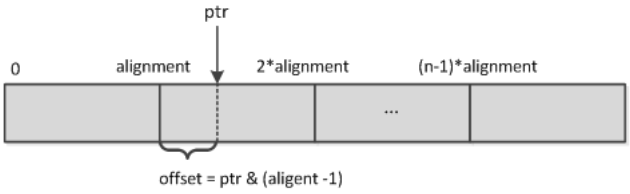
我把三种分配方式列在一起方便比较：



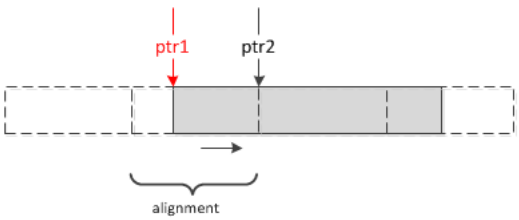
a. huge

先介绍一个重要的宏：

- `#define ZEND_MM_ALIGNED_OFFSET(size, alignment)  (((size_t)(size)) & ((alignment) - 1))`
  - 通过该宏可以容易得到：按alignment对齐的内存地址距离上一个alignment整数倍内存地址的偏移或者说offset，但是alignment必须是2的n次方。
  - 也可以通过算术表达式实现：offset = ptr - (ptr/alignment取整)*alignment。



huge内直接通过mmap向OS申请内存，但是如果zend_mm_mmap返回的地址不是alignment（即chunk_size 2MB）的整数倍，会将该内存释放回给OS，然后按照"size + chunk_size - p后对齐到整数倍，那么前面的部分会释放给OS（释放：zend_mm_munmap）；



- 假设调整后申请整个灰色大小，zend_mm_mmap返回ptr1，但是zend引擎会对齐到chunk整数倍并将前面的内存释放（因为是向后对齐，所以释放前面），**最终使用ptr2.**
- 另外，此时申请的chunk并没有加入到chunk列表中。

最后将该内存块加入到huge列表中（头插法）；

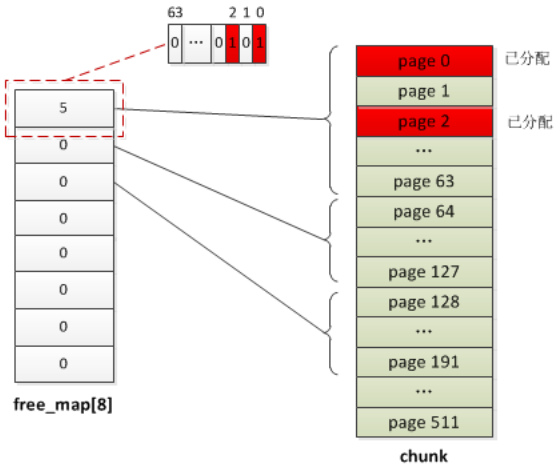
b. large

- 重点：尽可能填满chunk的空隙；

- 遍历chunk列表找到第一个满足连续个数为pages的chunk，然后进行最最优匹配。
- 如果最后一个也没有，那么申请一个新的chunk并初始化后插入到chunk列表中（头插法）。

free_map作用简介：从上面介绍可知1个chunk包括512个页，无论是large还是small都是申请page内存，所以需要记录chunk中page使用情况。既然仅需要记录该页是否可用，那么一个b
是这么做的。free_map就是8、16个整数的数组（以下介绍默认64位OS）。

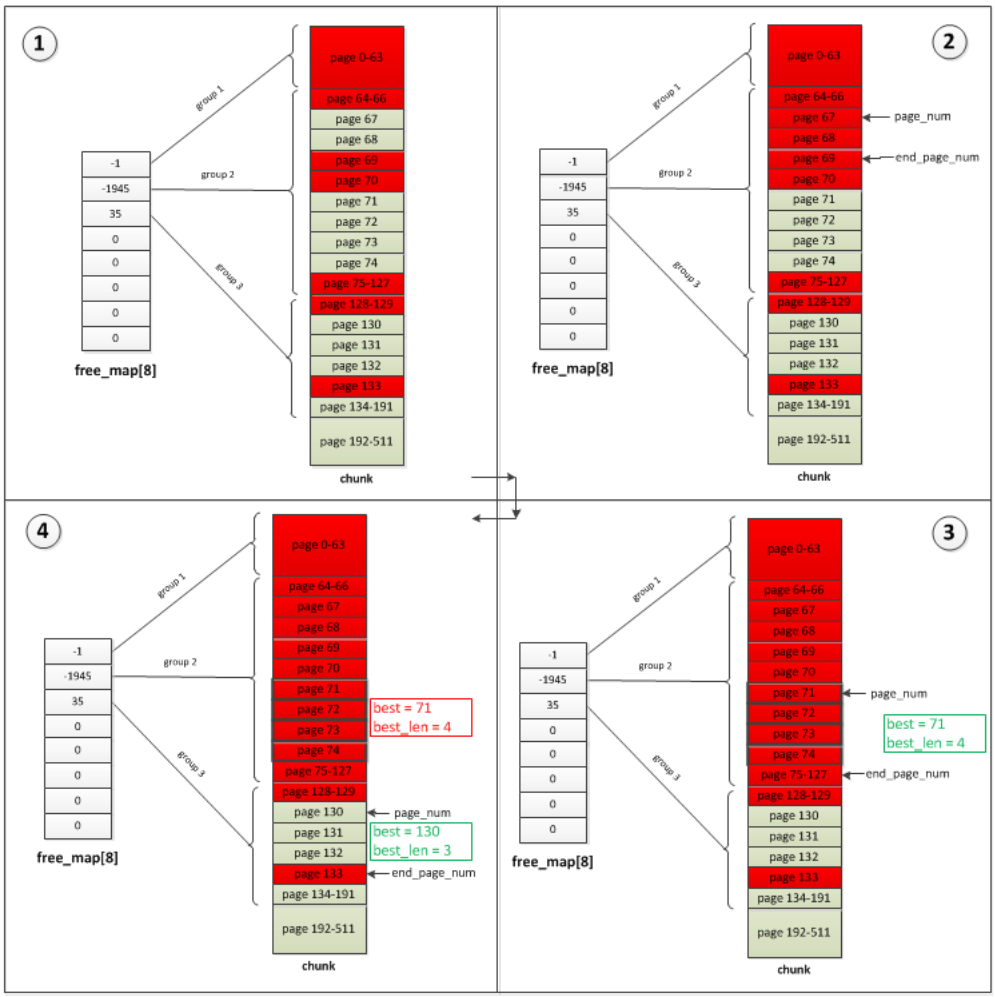
- 第0个记录0～63页的使用情况，同理7代表448～511页使用情况，比如当前chunk的0、2页已经分配，那么free_map[0] = 5 （**** 0000 0101）
- 操作时：先复制一份free_map，接下来的操作都是在该副本map上（我看代码时这个地方就是一概而过没当回事，所以后来一直纠结怎么释放参考最优匹配算法第2步中置为已用状
map中修改，不影响原始map，所以不用考虑释放的问题。我当时一直觉得是原始map表，耽误好多时间～）



在整体空闲页够的情况下，我们继续看下最优匹配连续size个page的过程（当然有可能连续的不够，那么此时会新申请一个chunk）：

1. 首先从第一个page分组(page 0-63)开始检查，如果当前分组无可用page(即free_map[x] = -1)则进入下一分组，直到当前分组有空闲page，然后进入step2；
2. 当前分组有可用page，首先找到第一个可用page的位置，记作page_num，接着从page_num开始向下找第一个已分配page的位置，记作end_page_num，这个地方需要注意，如page都是可用的则会进入下一分组接着搜索，直到找到为止；
3. 计算两个游标的差值len即代表可用间隙中page个数，然后用该差值和psize比较：
   a. 如果len = psize，最优匹配，直接从page_num分配psize个页；
   b. 如果len > psize，次优匹配，先记录page_num以及len，如果有更优或者说更小的间隙时替换；
   c. 如果len < psize，无效间隙，直接置为已用状态（再强调一下是副本map，不影响实际free_map）；

如下例子，初始状态为图1：



1. 直接跳过第一个分组，在第二个分组查找发现有空闲；（G1代表第一分组）
2. 在G2中到可用page为67，然后向下找第一个已用page为69，差值即可用间隙长度为2，小于指定长度3，标示该间隙无效，置为已用状态如图2；
3. 继续遍历G2，找到可用page为71，接下来第一个不可用为75，差值即间隙为4，大于指定3，暂记住起始71和长度4，然后标记71～74已使用如图3；
4. 在G2中已无可用page，继续遍历chunk其他分组，在G3中发现可用130～132，正好为3，完美匹配，直接分配即可；

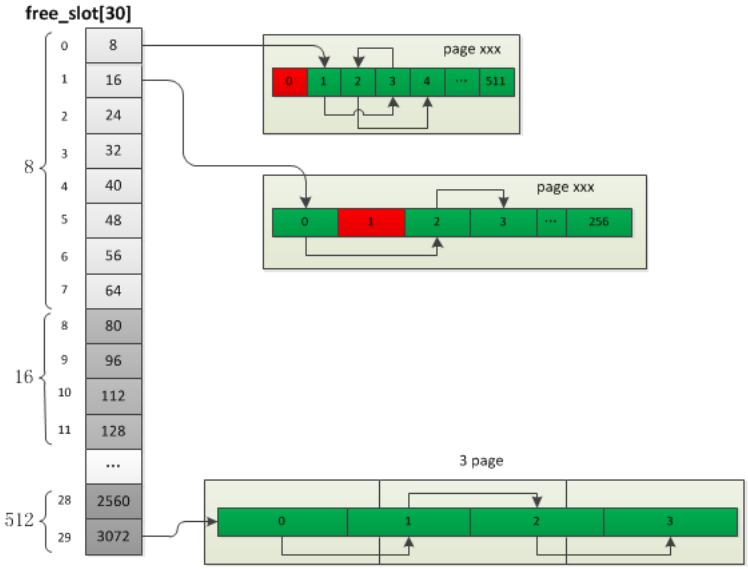
page分配完成后会将free_map对应整数的bit位从page_num至(page_num+page_count)置为1，同时将chunk->map[page_num]置为`ZEND_MM_LRUN(pages_count)`，表示page_num至
(page_num+page_count)这些page是被Large分配占用的。

c. slot

- slot为包含**30个单向列表**的数组：最小的slot大小为8byte，前8个slot依次递增8byte，后面每隔4个递增值乘以2。具体定义在zend/zend_alloc_sizes.h中（slot编号、对应大小、数
- 每一个slot永远指向第一个可用的内存块；

分配过程如下：

1. 先在heap->free_slot中找到对应规则链表，然后不为空则返回当前第一个，并后移一个元素；
2. 如果为空，那么先申请配置的页数，然后切割为对应大小的“块”并连接成单向列表；
3. 释放时，采用头插法放到链表头部；



思考？我们配置的memory_limit是什么时候起效果的？

2.1.2.3 内存释放

与申请时3中不同分配方法一一对应，zend引擎释放内存也存在3个调用；

```
#define efree(ptr)                      _efree((ptr) ZEND_FILE_LINE_CC ZEND_FILE_LINE_EMPTY_CC)
#define efree_large(ptr)                _efree_large((ptr) ZEND_FILE_LINE_CC ZEND_FILE_LINE_EMPTY_CC)
#define efree_huge(ptr)                 _efree_huge((ptr) ZEND_FILE_LINE_CC ZEND_FILE_LINE_EMPTY_CC)
```

释放时传递的是指针，那么如何定位到chunk的呢？

另外如果一个chunk所有的page都释放了，zend引擎是不是就把内存chunk还给OS了呢？

2.1.2.4 系统内存管理

OS内存的延迟分配：用户申请内存的时候，只是给它分配了一个线性区（也就是虚存），并没有分配实际物理内存；只有当用户使用这块内存的时候，内核才会分配具体的物理页面给用宝贵的物理内存。内核释放物理页面是通过释放线性区，找到其所对应的物理页面，将其全部释放的过程。

- 我们平时写C/C++代码都是通过malloc等glibc提供的内存方法进行操作，zend引擎调用的是mmap方法；

一般linux操作系统会提供如下几个系统调用来实现内存的分配和回收：brk & sbrk、mmap & unmmap。

- 当malloc函数申请超过128K内存时调用mmap，否则使用brk即可。

操作系统内存管理，待整理。。。

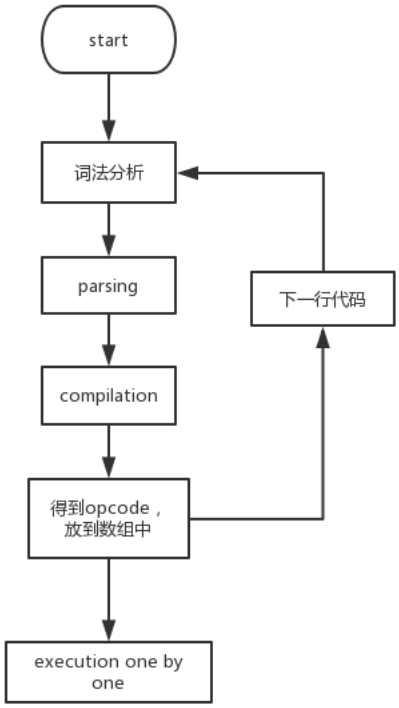
3、技术分享-未回答清楚问题记录

3.1 词法、语法、opcode生成过程不应该是上图所示；

刚神@胡刚提出的问题：1.2脚本执行过程不应该是图中展示的样子。

解释：提供的图中可能会造成一定的理解偏差。首先我们以正常逻辑来看这张图，给大家的最直观的理解就是一段PHP代码（记为**codes**），首先进行scan词法分析（lex，re2c）获得to tokens进行有效表达式解析parse得到所有的有效表达式，最后对这些有效表达式进行编译compile得到opcodes-array，Zend引擎在一条一条的执行这些opcodes。这个就是这张图片最不

把图片更改一下：



这样大家理解起来可能就比较方便了（但是上图也有一定误差，下面我也会说到：

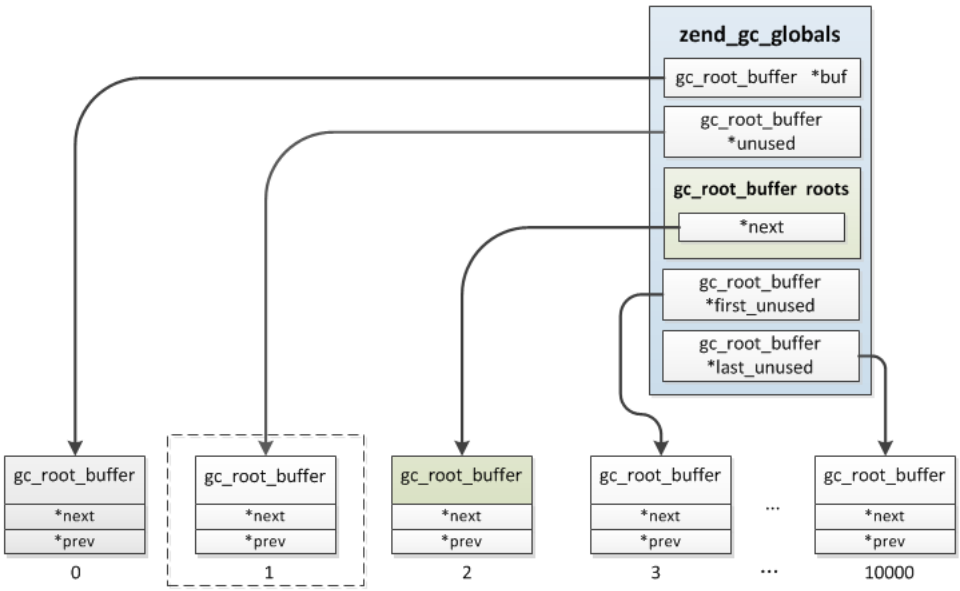
- 在整个过程中，处理单位不是我们理解的整个PHP脚本的所有代码，而是**以行为单位**，每一行处理完成后会生成对应的“ zend_op ”，然后放到全局op数组中，在处理下一行代码；
  - 以前的理解是：词法分析得到结果即所有tokens，然后将tokens给语法分析，然后语法分析结果给编译，然后生成opcode，**这是不正确的。**
  - 另外处理第一步不是词法分析，而是语法分析：处理时由语法分析调用词法分析来完成，所以实现上仍然是先产出tokens，然后生成有效表达式，但是入口却是语法分析；

3.2 PHP5.3之后垃圾回收算法

@万方提出问题垃圾回收内部是怎么维护的?

我们首先明确PHP对垃圾的定义：GC判断是否为垃圾的一个重要标准是有没有变量名指向变量容器zval。

PHP内部结构是这样的，我们重点关心最下面一层gc_root_buffer这个链表就可以：这个链表保存的就是PHP运行过程中所有的“ 疑似垃圾 ”；



通过配置文件可以配置root-buffer的大小即垃圾回收站，如果超过配置个数会自动触发垃圾回收机制。

那么深度优先遍历是什么情况呢?

- 对于普通结构如整形等，那么直接按着链接遍历即可；
- 对于array也比较好理解，因为所有成员都在arData数组中，直接遍历arData即可；
- **但是**如果数组中各元素仍是array、object或者引用，则一直递归进行深度优先遍历；

那么又会引入一个问题，由于object可以动态添加属性，那么怎么处理?

因为成员属性除了明确的在类中定义的那些外还可以动态创建，动态属性保存于zend_obejct->properties这个数组中，而普通属性保存在zend_object.properties_table数组中，结果就是c分散在两个位置，那么遍历时会分别遍历而导致遍历两遍吗？答案是否定的。PHP在创建动态属性时也会把**全部普通属性也加到zend_obejct->properties哈希表中**，指向原zend_object的属性，这样一来GC遍历object的成员时就可以像array那样遍历zend_obejct->properties即可。

3.3 内存管理时，为什么需要将已遍历的page置为已用?

先直接给出答案：PHP内存large分配有两次遍历：第一层是chunk链的遍历，第二层是每个chunk的free_map数组的遍历；我们当时讨论的遍历是第二层遍历，但是我当时讲的以页为单
确的（sorry），

首先确定下一些基础知识：

- free_map记录该chunk的使用情况：一般来说chunk有512个page，free_map有8个long元素，每个long元素64bit就可以记录64个page使用情况；
- 我们知道如果long元素每一个bit值都是1，那就意味着这个long元素值就是-1也就是说对应的64个page都是已用的；

那么对于这两次遍历，第一层链表没有太多好说的，对于第二层遍历就是如果存在某个long值不是-1则意味着该值对应的64个page页面存在可用页面或者说存在间隙。所以说我们找空隙
个8个long元素，找到不等于-1的过程，根本没有针对page遍历这个概念（因为遍历的是free_map）。。。我的错，当时误导大家了……(*☻-☻*)

- 记录当前free_map的下标，假如某元素代表的64个page组里面有两个空隙，如果不把第一个置为已用，那么下次遍历的时候还会检测到这个间隙。
  - 假设我们free_map[1]有两个间隙A、B，遍历时发现A不满足，那么如果不把A间隙置为已用，由于free_map[1]还没有等于-1，所以还会继续遍历free_map[1]，下次找间隙
  - 如果发现free_map[1]没有可用的了，那么就可以遍历free_map[2]了。

大家当时对这个都有疑问，而我没有回答清楚，当时看资料和代码时的想法：这个地方还挺好理解就没有详细去想为什么。所以个人对这个地方就是达到“熟悉但不能说是理解”的状态，t
白。

3.3.1 扩展：为什么不用针对page的游标呢?

这个又要回到PHP内部结构的地方了，我们知道对于任何一个chunk，其第一个page是用来记录本chunk的使用情况，我们经常说的free_map就是放在第一个page中。

既然可以通过free_map数组每一个元素的值来判断是否有可用间隙，即通过判断元素值是否等于-1即可，如果没有直接跨过这64个pages，如果有那么直接找到对应间隙即可，所以我们
页一个一个的去遍历呢?

附件列表　　隐藏图片附件

名称	大小	创建人	日期
zend_gc_2.png	31 kB	张状	2018-05-07 15:14:31
未命名文件 (35).png	16 kB	张状	2018-05-07 14:33:03
free_slot.png	21 kB	张状	2018-05-02 20:44:24
未命名文件 (34).png	260 kB	张状	2018-05-02 20:36:39
free_map_1.png	83 kB	张状	2018-05-02 20:01:31
free_map (1).png	21 kB	张状	2018-05-02 17:02:27
chunk_alloc.png	4 kB	张状	2018-05-02 16:15:30
align.png	6 kB	张状	2018-05-02 16:08:39
chunk_init.png	76 kB	张状	2018-05-02 15:14:48
未命名文件 (33).png	30 kB	张状	2018-05-02 15:14:21
zend_heap.png	35 kB	张状	2018-05-02 15:00:13
未命名文件 (32).png	56 kB	张状	2018-05-01 22:13:28
leak-array.png	5 kB	张状	2018-05-01 21:46:44
loop-array.png	6 kB	张状	2018-05-01 21:44:55
6091a33fb5474116ec73b7edb.png	165 kB	张状	2018-05-01 17:45:53
e84146c32b03673470ac2c337.png	90 kB	张状	2018-05-01 17:45:38
02-01-013-multithreaded-lift-cycle.png	423 kB	张状	2018-05-01 17:38:03
02-01-02-multiprocess-life-cycle.png	381 kB	张状	2018-05-01 17:37:54
02-01-01-cgi-lift-cycle.png	758 kB	张状	2018-05-01 17:37:43
未命名文件 (29).png	59 kB	张状	2018-05-01 17:37:22
未命名文件 (31).png	65 kB	张状	2018-05-01 17:33:21
未命名文件 (30).png	60 kB	张状	2018-05-01 17:21:54
未命名文件 (28).png	59 kB	张状	2018-05-01 17:15:12
未命名文件 (27).png	133 kB	张状	2018-05-01 16:44:39
02-00-php-inernal.png	45 kB	张状	2018-05-01 15:05:58

赞　成为第一个赞同者