# 前言

zend\_vm\_gen.php脚本用于：使用已有的zend\_vm\_def.h伪代码和zend\_vm\_execute.skl代码框架两个源码文件，创建zend\_vm\_execute.h和zend\_vm\_opcodes.{h,c}三个源码文件。学习它可以了解PH虚拟机的实现原理和设计思路，加深对PHP虚拟机的了解。

zend\_vm\_gen.php脚本共有4种运行模式，分别是ZEND\_VM\_KIND\_CALL、ZEND\_VM\_KIND\_SWITCH、ZEND\_VM\_KIND\_GOTO、ZEND\_VM\_KIND\_HYBRID，默认使用ZEND\_VM\_KIND\_HYBRID模式运行，所以本篇仅介绍ZEND\_VM\_KIND\_HYBRID模式的运行逻辑，忽略其他模式的运行逻辑。

## 一）相关常量

### $vm\_op\_flags全局变量

zend\_vm\_gen.php脚本中大部分常量来自于$vm\_op\_flags全局变量，代码如下：

|  |
| --- |
| // 虚拟机操作标记，数组 gen\_vm\_opcodes\_header，gen\_vm 用到  $vm\_op\_flags = array(  // 第一组，以下这几个表示操作对象类型  "ZEND\_VM\_OP\_SPEC" => 1<<0, // 表示有指定类型，不是ANY  "ZEND\_VM\_OP\_CONST" => 1<<1, // 常量  "ZEND\_VM\_OP\_TMPVAR" => 1<<2, // 普通变量、临时变量  "ZEND\_VM\_OP\_TMPVARCV" => 1<<3, // 普通变量、临时变量、编译变量    // 第二组，以下这些在优化器（Optimizer）中用到  "ZEND\_VM\_OP\_MASK" => 0xf0, // 过滤右侧第5-8位，优化器中用到  "ZEND\_VM\_OP\_NUM" => 0x10, // 右侧第5位，优化器中用到  ...  "ZEND\_VM\_EXT\_SRC" => 0x0b000000,  // unused 0x0c000000,  // 第四组，以下这两个在内核代码中没有用到  "ZEND\_VM\_NO\_CONST\_CONST" => 0x40000000,  "ZEND\_VM\_COMMUTATIVE" => 0x80000000,  );  foreach ($vm\_op\_flags as $name => $val) {  define($name, $val);  } |

相关常量定义如下：

|  |  |  |
| --- | --- | --- |
| **常量名** | **值** | **说明** |
| ZEND\_VM\_KIND\_CALL | 1 | 调用模式 |
| ZEND\_VM\_KIND\_SWITCH | 2 | switch语句模式 |
| ZEND\_VM\_KIND\_GOTO | 3 | goto语句 |
| ZEND\_VM\_KIND\_HYBRID | 4 | 混合模式 |
| ZEND\_VM\_OP\_SPEC | 1<<0 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_OP\_CONST | 1<<1 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_OP\_TMPVAR | 1<<2 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_OP\_TMPVARCV | 1<<3 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_OP\_MASK | 0xf0 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_OP\_NUM | 0x10 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_OP\_JMP\_ADDR | 0x20 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_OP\_TRY\_CATCH | 0x30 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_OP\_THIS | 0x50 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_OP\_NEXT | 0x60 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_OP\_CLASS\_FETCH | 0x70 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_OP\_CONSTRUCTOR | 0x80 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_OP\_CONST\_FETCH | 0x90 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_OP\_CACHE\_SLOT | 0xa0 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_EXT\_VAR\_FETCH | 1<<16 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_EXT\_ISSET | 1<<17 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_EXT\_CACHE\_SLOT | 1<<18 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_EXT\_ARRAY\_INIT | 1<<19 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_EXT\_REF | 1<<20 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_EXT\_FETCH\_REF | 1<<21 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_EXT\_DIM\_WRITE | 1<<22 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_EXT\_MASK | 0x0f000000 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_EXT\_NUM | 0x01000000 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_EXT\_LAST\_CATCH | 0x02000000 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_EXT\_JMP\_ADDR | 0x03000000 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_EXT\_OP | 0x04000000 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_EXT\_TYPE | 0x07000000 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_EXT\_EVAL | 0x08000000 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_EXT\_TYPE\_MASK | 0x09000000 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_EXT\_SRC | 0x0b000000 | 优化器（Optimizer）中用到 |
| ZEND\_VM\_NO\_CONST\_CONST | 0x40000000 | 只定义，内核中没有相关业务逻辑 |
| ZEND\_VM\_COMMUTATIVE | 0x80000000 | 只定义，内核中没有相关业务逻辑 |

### 其他常量

另有几个单独定义的常量：

|  |
| --- |
| if (!defined("ZEND\_VM\_KIND")) {  // Using CALL threading by default  define("ZEND\_VM\_KIND", ZEND\_VM\_KIND\_HYBRID); // 默认使用HYBRID模式运行  }  if (!defined("ZEND\_VM\_SPEC")) {  // Using specialized executor by default  define("ZEND\_VM\_SPEC", 1); // 默认使用 专用执行器  }  if (!defined("ZEND\_VM\_LINES")) {  // Disabling #line directives  define("ZEND\_VM\_LINES", 0); // 禁止使用 #line 指令  } |

## 二）gen\_vm()函数

zend\_vm\_gen.php中唯一启动语句是调用gen\_vm()函数，代码如下：

|  |
| --- |
| gen\_vm(\_\_DIR\_\_ . "/zend\_vm\_def.h", \_\_DIR\_\_ . "/zend\_vm\_execute.skl"); |

下面以gen\_vm函数为线索展开介绍zend\_vm\_gen.php脚本的业务逻辑，函数声名如下：

|  |
| --- |
| **// gen\_vm()函数的代码，第1块**  function gen\_vm($def, $skel) {  // 引用外部全局变量  global $definition\_file, $skeleton\_file, $executor\_file,  $op\_types, $list, $opcodes, $helpers, $params, $opnames,  $vm\_op\_flags, $used\_extra\_spec;  $in = @file($def); // 加载定义文件：zend\_vm\_def.h    $definition\_file = realpath($def); // 在 #line 指定中需要用到绝对路径  $skl = @file($skel); // 加载骨架 zend\_vm\_execute.skl  $skeleton\_file = realpath($skel); // 需要框架文件的绝对路径，用在 #line 指令里  // 把定义文件解析成树结构  $lineno = 0;  $handler = null;  $helper = null;  $max\_opcode\_len = 0;  $max\_opcode = 0;  // 额外数量 ： 256  $extra\_num = 256; |

# 一、解析函数名

zend\_vm\_def.h文件中存放的是虚拟机的伪代码，包含伪函数和伪常量等，zend\_vm\_gen.php脚本会把它们解析成虚拟机的C语言代码。

gen\_vm()函数最前面的业务逻辑：遍历zend\_vm\_def.h的每一行，根据每一行的内容不同，执行如下操作。

## 一）解析普通处理器（HANDLER）名

代码如下：

|  |
| --- |
| **// gen\_vm()函数的代码，第2块**  **// 1. 数据解析**  // 遍历定义 zend\_vm\_def.h 的每一行  foreach ($in as $line) {  ++$lineno;  **// 1.1 解析普通处理器。**1）普通处理器：107个  if (strpos($line,"ZEND\_VM\_HANDLER(") === 0 ||  // 2）行内处理器：1个  strpos($line,"ZEND\_VM\_INLINE\_HANDLER(") === 0 ||  // 3）热处理器：28个  strpos($line,"ZEND\_VM\_HOT\_HANDLER(") === 0 ||  // 4）热、非常量处理器：5个  strpos($line,"ZEND\_VM\_HOT\_NOCONST\_HANDLER(") === 0 ||  // 5）热、非双常量处理器：7个  strpos($line,"ZEND\_VM\_HOT\_NOCONSTCONST\_HANDLER(") === 0 ||  // 6）热、发送处理器：4个  strpos($line,"ZEND\_VM\_HOT\_SEND\_HANDLER(") === 0 ||  // 7）热、对象处理器：2个  strpos($line,"ZEND\_VM\_HOT\_OBJ\_HANDLER(") === 0 ||  // 8）冷、处理器：6个  strpos($line,"ZEND\_VM\_COLD\_HANDLER(") === 0 ||  // 9）冷、常量处理器：20个  strpos($line,"ZEND\_VM\_COLD\_CONST\_HANDLER(") === 0 ||  // 10）冷、双常量处理器：21个  strpos($line,"ZEND\_VM\_COLD\_CONSTCONST\_HANDLER(") === 0) {  // Parsing opcode handler's definition 解析操作码处理器的定义  // 检查定义是否合法  if (  preg\_match("/^ZEND\_VM\_(HOT\_|INLINE\_|HOT\_OBJ\_|HOT\_SEND\_|HOT\_NOCONST\_|HOT\_NOCONSTCONST\_|COLD\_|COLD\_CONST\_|COLD\_CONSTCONST\_)?HANDLER\(\s\*([0-9]+)\s\*,\s\*([A-Z\_]+)\s\*,\s\*([A-Z\_|]+)\s\*,\s\*([A-Z\_|]+)\s\*(,\s\*([A-Z\_|]+)\s\*)?(,\s\*SPEC\(([A-Z\_|=,]+)\)\s\*)?\)/",  $line,  $m) == 0) {  // 如果不合法，报错：无效的虚拟机处理器定义  die("ERROR ($def:$lineno): Invalid ZEND\_VM\_HANDLER definition.\n");  }  // 正则的第一段，处理类型：九种情况，可以是空 (HOT\_|INLINE\_|HOT\_OBJ\_|HOT\_SEND\_|HOT\_NOCONST\_|HOT\_NOCONSTCONST\_|COLD\_|COLD\_CONST\_|COLD\_CONSTCONST\_)  $hot = !empty($m[1]) ? $m[1] : false; // 默认不是热处理器  // 正则第二段 ([0-9]+)，第一个参数：操作码号，数字，如：24  $code = (int)$m[2];  // 正则第三段 ([A-Z\_]+)，第二个参数：操作码名，如：ZEND\_ASSIGN\_OBJ  $op = $m[3];  // 操作码长度  $len = strlen($op);  // 正则第四段，([A-Z\_|]+)，第三个参数：第一个操作对象的类型，如：：VAR|UNUSED|THIS|CV  $op1 = parse\_operand\_spec($def, $lineno, $m[4], $flags1);  // 正则第五段，([A-Z\_|]+)，第四个参数：第二个操作对象的类型，如：：CONST|TMPVAR|CV  $op2 = parse\_operand\_spec($def, $lineno, $m[5], $flags2);  // 两个操作对象的类型拚在一起，第二个参数的类型占左8位，第一个占右8位。  $flags = $flags1 | ($flags2 << 8);  // 正则第七段，([A-Z\_|]+)，扩展参数：标记，如：CACHE\_SLOT  if (!empty($m[7])) { $flags |= parse\_ext\_spec($def, $lineno, $m[7]); }  if ($len > $max\_opcode\_len) { $max\_opcode\_len = $len; } // 更新操作码最大长度  if ($code > $max\_opcode) { $max\_opcode = $code; } // 更新最大操作码名  if (isset($opcodes[$code])) { // 如果操作码号存在，报错：操作码号已经存在  die("ERROR ($def:$lineno): Opcode with code '$code' is already defined.\n");}  if (isset($opnames[$op])) { // 如果操作码存在，报错：操作码已经存在  die("ERROR ($def:$lineno): Opcode with name '$op' is already defined.\n"); }  // 创建操作码  $opcodes[$code] = array("op"=>$op,"op1"=>$op1,"op2"=>$op2,"code"=>"","flags"=>$flags,"hot"=>$hot);  // 正则第九段，([A-Z\_|=,]+)，可选参数：特殊规则，如 SPEC(OP\_DATA=CONST|TMP|VAR|CV)  if (isset($m[9])) {  // 解析专用规则：zend\_vm\_def.h，行号，规则  $opcodes[$code]["spec"] = parse\_spec\_rules($def, $lineno, $m[9]);  // 如果已经存在 NO\_CONST\_CONST 标记  if (isset($opcodes[$code]["spec"]["NO\_CONST\_CONST"])) {  // 添加 ZEND\_VM\_NO\_CONST\_CONST 标记  $opcodes[$code]["flags"] |= $vm\_op\_flags["ZEND\_VM\_NO\_CONST\_CONST"];  }  // 如果已经存在 COMMUTATIVE（交换的） 标记  if (isset($opcodes[$code]["spec"]["COMMUTATIVE"])) {  // 添加 ZEND\_VM\_COMMUTATIVE 标记  $opcodes[$code]["flags"] |= $vm\_op\_flags["ZEND\_VM\_COMMUTATIVE"];  }  }  $opnames[$op] = $code; // 操作码名=>操作码号  $handler = $code; // 操作码号  $helper = null; // 助手  $list[$lineno] = array("handler"=>$handler); // 行号：['handler'=>操作码号] |

如上所示，如果本行包含“ZEND\_VM\_HANDLER(”等10个关键字，说明本行在声名处理器，需要进行以下逻辑处理：

### 1、验证和拆分处理器名

用正则表达式把处理器名提取出来，并拆分到变量中，正则表达式可拆分成如下几段：

1）【^ZEND\_VM\_(HOT\_|INLINE\_|HOT\_OBJ\_|HOT\_SEND\_|HOT\_NOCONST\_|

HOT\_NOCONSTCONST\_|COLD\_|COLD\_CONST\_|COLD\_CONSTCONST\_)?】匹配**处理器的类型**，可以没有；

2）【HANDLER\(\s\*([0-9]+)】匹配第一个参数：操作码编号，数字，如：24；

3）【\s\*,\s\*([A-Z\_]+)】匹配第二个参数：操作码名，如：ZEND\_ASSIGN\_OBJ；

4）【\s\*,\s\*([A-Z\_|]+)】匹配第一个操作对象的类型，如：：VAR|UNUSED|THIS|CV；

5）【\s\*,\s\*([A-Z\_|]+)】匹配第二个操作对象的类型，如：：CONST|TMPVAR|CV；

6）【\s\*(,\s\*([A-Z\_|]+)\s\*)?】匹配可选扩展参数，参数在第7段表达式中；

7）【([A-Z\_|]+)】匹配可选参数：扩展参数，如：CACHE\_SLOT；

8）【(,\s\*SPEC\(([A-Z\_|=,]+)\)\s\*)?\)】匹配可选特殊标记SPEC(...)，可选，标记在第9段表达式中；

9）【([A-Z\_|=,]+)】匹配可选参数：特殊规则，如 SPEC(OP\_DATA=CONST|TMP|VAR|CV)。

如上所示，声名处理器时最多包含7个有效参数：处理器修饰符、操作码编号、操作码名、第一个操作对象的类型、第二个操作对象的类型；以及2个可选参数：带括号的参数、专用规则。

### 2、解析操作对象规则

#### **parse\_operand\_spec()函数**

解析操作对象规则主要用到parse\_operand\_spec()函数，函数定义如下：

|  |
| --- |
| function parse\_operand\_spec($def, $lineno, $str, &$flags) {  global $vm\_op\_decode; // 外部全局变量  $flags = 0;  $a = explode("|",$str); // 把类型拆分开  foreach ($a as $val) { // 遍历每一个类型  if (isset($vm\_op\_decode[$val])) { // 如果已经在 vm\_op\_decode 中，是有效类型  $flags |= $vm\_op\_decode[$val]; // 添加此类型的标记  } else { // 碰到无效类型，报错退出  die("ERROR ($def:$lineno): Wrong operand type '$str'\n");  }  }  if (!($flags & ZEND\_VM\_OP\_SPEC)) { // 如果类型集合没有SPEC标记，转成ANY  if (count($a) != 1) { // 如果0个或多个，报错：$str 是错误类型  die("ERROR ($def:$lineno): Wrong operand type '$str'\n"); }  $a = array("ANY"); // 如果是只有一个类型，类型是 ANY  }  return array\_flip($a); // 交换键和值  } |

如上所示，业务逻辑归纳如下：

1）先把规则用竖线分开，再如遍历每个类型：如果类型有效，添加到 $flags 里；

2）如果$flags类型中没有ZEND\_VM\_OP\_SPEC标记，把类型标记成ANY；

3）交换$a的键值让类型存放在键名里，并返回它。返回的数组里只包含支持的类型。

$vm\_op\_decode数组定义如下：

|  |
| --- |
| // 虚拟机操作解码，parse\_operand\_spec 用到  $vm\_op\_decode = array(  "ANY" => 0,  "CONST" => ZEND\_VM\_OP\_SPEC | ZEND\_VM\_OP\_CONST,  "TMP" => ZEND\_VM\_OP\_SPEC,  "VAR" => ZEND\_VM\_OP\_SPEC,  "UNUSED" => ZEND\_VM\_OP\_SPEC,  "CV" => ZEND\_VM\_OP\_SPEC,  "TMPVAR" => ZEND\_VM\_OP\_SPEC | ZEND\_VM\_OP\_TMPVAR,  "TMPVARCV" => ZEND\_VM\_OP\_SPEC | ZEND\_VM\_OP\_TMPVARCV,  // 上面的在生成处理器和助手时用到，下面的在优化器中用到  "NUM" => ZEND\_VM\_OP\_NUM,  "JMP\_ADDR" => ZEND\_VM\_OP\_JMP\_ADDR,  "TRY\_CATCH" => ZEND\_VM\_OP\_TRY\_CATCH,  "THIS" => ZEND\_VM\_OP\_THIS,  "NEXT" => ZEND\_VM\_OP\_NEXT,  "CLASS\_FETCH" => ZEND\_VM\_OP\_CLASS\_FETCH,  "CONSTRUCTOR" => ZEND\_VM\_OP\_CONSTRUCTOR,  "CONST\_FETCH" => ZEND\_VM\_OP\_CONST\_FETCH,  "CACHE\_SLOT" => ZEND\_VM\_OP\_CACHE\_SLOT,  ); |

如上所示，不在$vm\_op\_decode中的类型是无效的。在$vm\_op\_decode中的类型，没有ZEND\_VM\_OP\_SPEC的都会转成ANY，那么转换后只可能有如下几种结果：

1）CONST、TMP、VAR、UNUSED、CV、TMPVAR、TMPVARCV包含以上任意一个或多个（以及其他所有）的类型集合；

2）ANY。

TMPVARCV后面的标记都在优化器（Optimizer）中使用。

### 3、解析扩展规则

#### **parse\_ext\_spec()函数**

parse\_ext\_spec()函数用于解析扩展规则（前文的第7个参数），代码如下：

|  |
| --- |
| function parse\_ext\_spec($def, $lineno, $str) {  global $vm\_ext\_decode; // 外部全局变量  $flags = 0;  $a = explode("|",$str); // 用 “|” 分隔开  foreach ($a as $val) { // 遍历每一个  if (isset($vm\_ext\_decode[$val])) { // 如果类型有效，添加标记  $flags |= $vm\_ext\_decode[$val];  } else { // 碰到不合法的直接报错  die("ERROR ($def:$lineno): Wrong extended\_value type '$str'\n");  }  }  return $flags;  } |

扩展参数也使用$vm\_ext\_decode中定义的类型，把规定的类型拚在一起返回。

#### **$vm\_ext\_decode全局变量**

$vm\_ext\_decode全局变量中的标记都在优化器（Optimizer）中用到，变量定义如下：

|  |
| --- |
| // 虚拟机扩展解码：parse\_ext\_spec  $vm\_ext\_decode = array(  "NUM" => ZEND\_VM\_EXT\_NUM,  "LAST\_CATCH" => ZEND\_VM\_EXT\_LAST\_CATCH,  "JMP\_ADDR" => ZEND\_VM\_EXT\_JMP\_ADDR,  "OP" => ZEND\_VM\_EXT\_OP,  "VAR\_FETCH" => ZEND\_VM\_EXT\_VAR\_FETCH,  "ARRAY\_INIT" => ZEND\_VM\_EXT\_ARRAY\_INIT,  "TYPE" => ZEND\_VM\_EXT\_TYPE,  "EVAL" => ZEND\_VM\_EXT\_EVAL,  "TYPE\_MASK" => ZEND\_VM\_EXT\_TYPE\_MASK,  "ISSET" => ZEND\_VM\_EXT\_ISSET,  "REF" => ZEND\_VM\_EXT\_REF,  "FETCH\_REF" => ZEND\_VM\_EXT\_FETCH\_REF,  "SRC" => ZEND\_VM\_EXT\_SRC,  "CACHE\_SLOT" => ZEND\_VM\_EXT\_CACHE\_SLOT, // 没有用到  "DIM\_WRITE" => ZEND\_VM\_EXT\_DIM\_WRITE,  ); |

### 4、创建操作码处理器

在操作码数处理器数组$opcodes中，普通处理器的键名是操作码编号，值为数组，包含如下键值对：

1）op：操作码名，例如：ZEND\_NOP、ZEND\_ADD，共202个；

2）op1：op1的规则表（数组），例如：{"ANY":0}、{"CONST":0,"TMPVAR":1}，共有ANY、CONST\_FETCH、CONST、TMPVARCV、TMPVAR、CV、UNUSED、VAR、CLASS\_FETCH、TMP、THIS，11种类型，共28种组合；

3）op2：op2的规则表（数组），与op1的规则类似，共有ANY、CONST、TMPVARCV、TMPVAR、CV、TMP、VAR、UNUSED、NEXT、NUM、CACHE\_SLOT、CLASS\_FETCH、CONSTRUCTOR，13种类型，共16种组合；

4）code：业务逻辑源码，初始为空；

5）flags：op1的规则+op2的规则+扩展规则，数字，共87个；

6）hot：处理器名中是否有HOT标记，共10个：HOT\_、HOT\_NOCONSTCONST\_、COLD\_CONSTCONST\_、HOT\_NOCONST\_、COLD\_CONST\_、HOT\_SEND\_、INLINE\_、COLD\_、HOT\_OBJ\_、false。

除以上6个键值对外，还有两个键值对是在后续操作中添加的：

7）spec：专用规则，共有11种组合：

{"COMMUTATIVE":[1]}

{"NO\_CONST\_CONST":[1]}

{"SMART\_BRANCH":[0,1,2],"COMMUTATIVE":[1]}

{"SMART\_BRANCH":[0,1,2]}

{"RETVAL":[0,1]}

{"OP\_DATA":{"CONST":0,"TMP":1,"VAR":2,"CV":3}}

{"OP\_DATA":{"VAR":0,"CV":1}}

{"QUICK\_ARG":[0,1]}

{"RETVAL":[0,1],"OBSERVER":[0,1]}

{"OBSERVER":[0,1]}

{"ISSET":[0,1]}；

8）type\_spec：操作码特殊规则的触发条件，在解析专用处理器时添加，例如 ：{"267":"(op1\_info == MAY\_BE\_LONG && op2\_info == MAY\_BE\_LONG)","268":"(op1\_info == MAY\_BE\_DOUBLE && op2\_info == MAY\_BE\_DOUBLE)"}，共22条记录。

### 5、给操作码添加专用规则

#### **parse\_spec\_rules()函数**

parse\_spec\_rules()函数用来解析专用规则（从处理器名中拆分出的第9个参数），代码如下：

|  |
| --- |
| // 解析专用规则, 如 SPEC(OP\_DATA=CONST|TMP|VAR|CV) 或 SPEC(RETVAL,OBSERVER)  // 只有 OP\_DATA= 用到等号，其他情况都是逗号分隔。  // 返回结果中 OP\_DATA 和其他的区别是 OP\_DATA 是 parse\_operand\_spec 生成的 有键名的数组 ，其他是索引数组  function parse\_spec\_rules($def, $lineno, $str) {  global $used\_extra\_spec; // 引用外部全局变量  $ret = array();  $a = explode(",", $str); // 用逗号拆分  foreach ($a as $rule) { // 遍历每一个规则  $n = strpos($rule, "="); // 是否有=  if ($n !== false) { // 如果有 “= ”,OP\_DATA=  $id = trim(substr($rule, 0, $n)); // = 前面是ID  $val = trim(substr($rule, $n+1)); // = 后面是 value  switch ($id) { // 只能是OP\_DATA，  case "OP\_DATA":  // 解析操作对象的特殊属性, $devnull 是返回的 flag, 这里没有用到  $ret["OP\_DATA"] = parse\_operand\_spec($def, $lineno, $val, $devnull);  break;  default: // 有等号的不支持其他ID  die("ERROR ($def:$lineno): Wrong specialization rules '$str'\n");  }  $used\_extra\_spec[$id] = 1; // 把ID添加到额外标记列表里  } else { // 如果没有 =  switch ($rule) { // 支持的类型 7个，后面的数组表示这个选项可能接受的值  case "RETVAL":  $ret["RETVAL"] = array(0, 1);  break;  case "QUICK\_ARG":  $ret["QUICK\_ARG"] = array(0, 1);  break;  case "SMART\_BRANCH":  $ret["SMART\_BRANCH"] = array(0, 1, 2);  break;  case "NO\_CONST\_CONST":  $ret["NO\_CONST\_CONST"] = array(1);  break;  case "COMMUTATIVE":  $ret["COMMUTATIVE"] = array(1);  break;  case "ISSET":  $ret["ISSET"] = array(0, 1);  break;  case "OBSERVER":  $ret["OBSERVER"] = array(0, 1);  break;  default: // 这7个以外，报错  die("ERROR ($def:$lineno): Wrong specialization rules '$str'\n");  }  $used\_extra\_spec[$rule] = 1; // 添加额外 扩展规则标记  }  }  return $ret; // 返回规则列表  } |

如上所示，把所有规则用逗号（,）分开，并遍历每一个规则：

1）如果有等号，必须是OP\_DATA开头，否则报错。用parse\_operand\_spec()函数解析此条规则 ，放在返回结果$ret["OP\_DATA"]中；

2）没有等号，支持RETVAL等7个参数。把拆分的规则放到$used\_extra\_spec中。

如果有专用规则，解析后还有一些后续处理：

1）如果规则中有NO\_CONST\_CONST标记（不支持双常量），需要给flags添加$vm\_op\_flags["ZEND\_VM\_NO\_CONST\_CONST"]标记，

2）如果规则中有COMMUTATIVE标记（两个操作对象可互换），需要给flags添加$vm\_op\_flags["ZEND\_VM\_COMMUTATIVE"]标记。

#### **专用规则的功能描述**

前文解析出的指令功能描述如下：

COMMUTATIVE：op1和op2两个操作对象是否可以互换；

NO\_CONST\_CONST：op1和op2都是常量时禁用；

SMART\_BRANCH：智能分支（按操作结果中的类型区分）；

RETVAL：是否有返回值；

OP\_DATA：操作数据的类型；

OBSERVER：观察者功能；

QUICK\_ARG：标记传入参数是否在12个以内；

ISSET：标记使用isset()函数或empty()函数。

## 二）解析专用处理器名

代码如下：

|  |
| --- |
| **// gen\_vm()函数的代码，第3块**  **// 1.2 解析专用处理器：**1）虚拟机类型专用处理器：2个  } else if (strpos($line,"ZEND\_VM\_TYPE\_SPEC\_HANDLER(") === 0 ||  //2）虚拟机热类型专用处理器：36个  strpos($line,"ZEND\_VM\_HOT\_TYPE\_SPEC\_HANDLER(") === 0) {  // 检查定义是否合法  if (preg\_match(  "/^ZEND\_VM\_(HOT\_|INLINE\_|HOT\_OBJ\_|HOT\_SEND\_|HOT\_NOCONST\_|HOT\_NOCONSTCONST\_)?TYPE\_SPEC\_HANDLER\(\s\*([A-Z\_|]+)\s\*,\s\*((?:[^(,]|\([^()]\*|(?R)\*\))\*),\s\*([A-Za-z\_]+)\s\*,\s\*([A-Z\_|]+)\s\*,\s\*([A-Z\_|]+)\s\*(,\s\*([A-Z\_|]+)\s\*)?(,\s\*SPEC\(([A-Z\_|=,]+)\)\s\*)?\)/",  $line,  $m) == 0) {  // 如果不合法，报错：无效的 虚拟机类型处理器 处理器定义  die("ERROR ($def:$lineno): Invalid ZEND\_VM\_TYPE\_HANDLER\_HANDLER definition.\n");  }  // 正则第一段，处理类型：六种情况，可以是空 HOT\_|INLINE\_|HOT\_OBJ\_|HOT\_SEND\_|HOT\_NOCONST\_|HOT\_NOCONSTCONST\_  $hot = !empty($m[1]) ? $m[1] : false;  // 正则第二段 [A-Z\_|]+ ，第一个参数，操作码名列表（|连接多个操作码），如：ZEND\_IS\_EQUAL|ZEND\_IS\_IDENTICAL  $orig\_op\_list = $m[2];  // 扩展编号从256开始算，+1  $code = $extra\_num++;  // 把 $orig\_op\_list 用 | 分隔开，遍历每个操作码，给普通处理器添加触发条件  foreach (explode('|', $orig\_op\_list) as $orig\_op) {  // 取得原操作码编号  $orig\_code = $opnames[$orig\_op];  // 正则第三段，(?:[^(,]|\([^()]\*|(?R)\*\))\*，第二个参数，触发条件如：(op1\_info == MAY\_BE\_DOUBLE && op2\_info == MAY\_BE\_DOUBLE)  $condition = $m[3];  // 给添加码添加专用类型，值为触发条件  $opcodes[$orig\_code]['type\_spec'][$code] = $condition;  }  // 正则第四段，[A-Za-z\_]+，第三个参数，操作码名，如：ZEND\_ASSIGN\_OBJ  $op = $m[4];  // 正则第五段，[A-Z\_|]+，第四个参数，第一个操作对象的类型，如：VAR|UNUSED|THIS|CV  $op1 = parse\_operand\_spec($def, $lineno, $m[5], $flags1);  // 正则第六段，[A-Z\_|]+，第五个参数，第二个操作对象的类型，如：CONST|TMPVAR|CV  $op2 = parse\_operand\_spec($def, $lineno, $m[6], $flags2);  // 一个标记有两个标记位置 $flags1 和 $flags2 << 8  $flags = $flags1 | ($flags2 << 8);  // 正则第八段，[A-Z\_|]+，可选参数，专用扩展标记，如： CACHE\_SLOT  if (!empty($m[8])) {  // 添加标记  $flags |= parse\_ext\_spec($def, $lineno, $m[8]);  }  // 专用扩展类型，1，？  $used\_extra\_spec["TYPE"] = 1;  // 创建操作码  $opcodes[$code] = array("op"=>$op,"op1"=>$op1,"op2"=>$op2,"code"=>"","flags"=>$flags,"hot"=>$hot,"is\_type\_spec"=>true);  // 正则第十段，[A-Z\_|=,]+，可选参数：专用规则，如 SPEC(OP\_DATA=CONST|TMP|VAR|CV)  if (isset($m[10])) {  // 解析专用规则  $opcodes[$code]["spec"] = parse\_spec\_rules($def, $lineno, $m[10]);  // 如果存在 NO\_CONST\_CONST 标记  if (isset($opcodes[$code]["spec"]["NO\_CONST\_CONST"])) {  // 添加 ZEND\_VM\_NO\_CONST\_CONST 标记  $opcodes[$code]["flags"] |= $vm\_op\_flags["ZEND\_VM\_NO\_CONST\_CONST"];  }  // 如果存在 COMMUTATIVE（交换）标记  if (isset($opcodes[$code]["spec"]["COMMUTATIVE"])) {  // 添加 ZEND\_VM\_COMMUTATIVE标记  $opcodes[$code]["flags"] |= $vm\_op\_flags["ZEND\_VM\_COMMUTATIVE"];  }  }  $opnames[$op] = $code; // 创建操作码  $handler = $code; // 处理器为当前操作码编号  $helper = null; // 定义处理器时清空助手定义  $list[$lineno] = array("handler"=>$handler); // 助手添加到列表中 |

如上所示，如果本行包含“ZEND\_VM\_TYPE\_SPEC\_HANDLER(”等7个关键字，说明本行在声名处理器，需要进行以下逻辑处理：

### 1、验证和拆分专用处理器名

用正则表达式把处理器名提取出来，并拆分到变量中，正则表达式可拆分成如下几段：

1）【^ZEND\_VM\_(HOT\_|INLINE\_|HOT\_OBJ\_|HOT\_SEND\_|HOT\_NOCONST\_|

HOT\_NOCONSTCONST\_)?】匹配**处理器的类型**，可以没有；

2）【TYPE\_SPEC\_HANDLER\(\s\*([A-Z\_|]+)】匹配第一个参数，操作码名列表（|连接多个操作码），如：ZEND\_IS\_EQUAL|ZEND\_IS\_IDENTICAL；

3）【\s\*,\s\*((?:[^(,]|\([^()]\*|(?R)\*\))\*)】匹配第二个参数，这个参数是一个表达式，用作专用处理器的触发条件，如：op->op2.num == MAY\_BE\_ANY，这里用到了递归匹配(?R)，所以表达式中可以用括号嵌套多层，为了避免把嵌套的表达式拆分成多个参数，里面使用了一层非获取匹配(?:)，外面再加一层获取匹配，保证不管表达式多复杂，都获取成1个参数；

4）【,\s\*([A-Za-z\_]+)】匹配第三个参数，专用处理器名，如："ZEND\_SUB\_LONG；

5）【\s\*,\s\*([A-Z\_|]+)】匹配第四个参数，第一个操作对象的类型，如：VAR|UNUSED|THIS|CV；

6）【\s\*,\s\*([A-Z\_|]+)】匹配第五个参数，第二个操作对象的类型，如：CONST|TMPVAR|CV；

7）【\s\*(,\s\*([A-Z\_|]+)\s\*)?】匹配专用扩展标记，提取出的标记在下一段中；

8）【([A-Z\_|]+)】匹配可选参数，专用扩展标记，如： CACHE\_SLOT；

9）【(,\s\*SPEC\(([A-Z\_|=,]+)\)\s\*)?\)】匹配SPEC(...)专用规则，匹配到的专用规则在下一段中；

10）【([A-Z\_|=,]+)】匹配专用规则，如：RETVAL,OBSERVER。

### 2、给操作码处理器添加触发条件

解析完参数后，需要给普通处理器添加触发条件，在这时候普通处理器必须已经解析完毕，也就是说，专用处理器必须在普通处理器后面声名。

从参数中找到操作码，并找到对应的普通处理器，把触发条件添加到处理器的type\_spec字段中，例如ZEND\_ADD操作码中有如下触发规则：

|  |
| --- |
| ["type\_spec"]=>  array(3) {  [258]=>  string(79) "(res\_info == MAY\_BE\_LONG && op1\_info == MAY\_BE\_LONG && op2\_info == MAY\_BE\_LONG)"  [259]=>  string(52) "(op1\_info == MAY\_BE\_LONG && op2\_info == MAY\_BE\_LONG)"  [260]=>  string(56) "(op1\_info == MAY\_BE\_DOUBLE && op2\_info == MAY\_BE\_DOUBLE)"  } |

其中每个键名是触发规则的编号，从256开始。所以$opcodes处理器列表中，前202个是普通操作码处理器，编号从256开始，后面的都是专用处理器。其中不存在45号操作码。

拆分后需要解析操作对象规则、解析扩展参数、创建操作码、给操作码添加专用规则。

### 3、创建操作码专用处理器

在操作码数组$opcodes中，专用处理器的键名是处理器生成顺序号，值为数组，包含以下键值对：

1）op：专用处理器名，例如：ZEND\_ADD\_LONG，共38个；

2）op1：op1的规则表（数组），例如：{"ANY":0}、{"CONST":0,"TMPVARCV":1}，共有ANY、CONST、TMPVARCV、TMPVAR、CV、VAR，6种类型，共7种组合；

3）op2：op2的规则表（数组），与op1的规则类似，共有ANY、CONST、TMPVARCV、CV、UNUSED、NUM，6种类型，共5种组合，与普通处理器相同；

4）code：业务逻辑源码，初始为空；

5）flags：op1的规则+op2的规则+扩展规则，数字，共12个；

6）hot：处理器名中是否有HOT标记，共2个：HOT\_、false；

7）is\_type\_spec：专用处理器的标记，值为true。

除以上7个键值对外，还有1个键值对是在后续操作中添加的：

7）spec：专用规则，共有5种组合：

{"NO\_CONST\_CONST":[1]}

{"SMART\_BRANCH":[0,1,2],"NO\_CONST\_CONST":[1],"COMMUTATIVE":[1]}

{"COMMUTATIVE":[1]}

{"SMART\_BRANCH":[0,1,2],"NO\_CONST\_CONST":[1]}

{"RETVAL":[0,1]}。

## 三）解析虚拟机助手（helper）

代码如下：

|  |
| --- |
| **// gen\_vm()函数的代码，第4块**  **// 1.3 解析虚拟机助手：普通助手：25个**  } else if (strpos($line,"ZEND\_VM\_HELPER(") === 0 ||  // 冷助手：8个  strpos($line,"ZEND\_VM\_COLD\_HELPER(") === 0 ||  // 热助手：1个  strpos($line,"ZEND\_VM\_HOT\_HELPER(") === 0) {  //检查定义是否合法  if (preg\_match(  "/^ZEND\_VM(\_INLINE|\_COLD|\_HOT)?\_HELPER\(\s\*([A-Za-z\_]+)\s\*,\s\*([A-Z\_|]+)\s\*,\s\*([A-Z\_|]+)\s\*(?:,\s\*SPEC\(([A-Z\_|=,]+)\)\s\*)?(?:,\s\*([^)]\*)\s\*)?\)/",  $line,  $m) == 0) {  // 如果不合法，报错  die("ERROR ($def:$lineno): Invalid ZEND\_VM\_HELPER definition.\n");  }  // 正则第一段，处理类型：三种情况，可以是空 \_INLINE|\_COLD|\_HOT  $inline = !empty($m[1]) && $m[1] === "\_INLINE";  $cold = !empty($m[1]) && $m[1] === "\_COLD";  $hot = !empty($m[1]) && $m[1] === "\_HOT";  // 正则第二段，[A-Za-z\_]+，第一个参数，助手名，如：zend\_sub\_helper  $helper = $m[2];  // 正则第三段，[A-Z\_|]+，第一个操作对象的类型，如：ANY  $op1 = parse\_operand\_spec($def, $lineno, $m[3], $flags1);  // 正则第四段，[A-Z\_|]+，第二个操作对象的类型，如：ANY  $op2 = parse\_operand\_spec($def, $lineno, $m[4], $flags2);  // 正则第六段，[^)]\*，c语言函数用到的形式参数，可以多个  $param = isset($m[6]) ? $m[6] : null;  //  if ((ZEND\_VM\_KIND == ZEND\_VM\_KIND\_GOTO  || ZEND\_VM\_KIND == ZEND\_VM\_KIND\_SWITCH  // ZEND\_VM\_KIND\_HYBRID + \_HOT 模式  || (ZEND\_VM\_KIND == ZEND\_VM\_KIND\_HYBRID && $hot))  && $param) {  foreach (explode(",", $param ) as $p) { // 拆分参数列表，遍历每个参数  $p = trim($p);  if ($p !== "") { // 如果参数存在，添加到参数列表里  $params[$p] = 1;  }  }  }  // 创建助手  $helpers[$helper] = array("op1"=>$op1,"op2"=>$op2,"param"=>$param,"code"=>"","inline"=>$inline,"cold"=>$cold,"hot"=>$hot);    $handler = null; // 定义助手时清空处理器定义  $list[$lineno] = array("helper"=>$helper); // 把助手添加到列表中 |

### 1、验证和拆分助手名

用正则表达式把处理器名提取出来，并拆分到变量中，正则表达式可拆分成如下几段：

1）【^ZEND\_VM(\_INLINE|\_COLD|\_HOT)?】匹配**处理器的类型**，三种情况，可以是空 \_INLINE|\_COLD|\_HOT；

2）【\_HELPER\(\s\*([A-Za-z\_]+)】匹配第一个参数，第一个参数，助手名，如：zend\_sub\_helper；

3）【\s\*,\s\*([A-Z\_|]+)】匹配第二个参数，第一个操作对象的类型，如：ANY；

4）【\s\*,\s\*([A-Z\_|]+)】匹配第三个参数，第二个操作对象的类型，如：ANY；

5）【\s\*(?:,\s\*SPEC\(([A-Z\_|=,]+)\)\s\*)?】匹配可选特殊标记SPEC(...)，可选，实际上助手中并未用到这个参数；

6）【(?:,\s\*([^)]\*)\s\*)?\)】匹配第五个参数，逗号开头，一直匹配到括号结尾，这一段都算第五个参数，可选，它生成C语函数时用到的形式参数表，如：zval \*op\_1, zval \*op\_2；

### 2、创建虚拟机助手

在助手数组$helpers中，普通处理器的键名是助手名称，值为数组，包含如下键值对：

1）op1：op1的规则表（数组），共有ANY、CONST、TMPVAR、CV、VAR，5种类型，共3种组合：{"ANY":0}、{"VAR":0,"CV":1}、{"CONST":0,"TMPVAR":1,"CV":2}；

2）op2：op1的规则表（数组），共有ANY、UNUSED，2种类型，共2种组合：{"ANY":0}、{"UNUSED":0}；

3）param：生成c语言函数代码时用的额外参数，共有6种情况：

a）zval \*op\_1, zval \*op\_2"；

b）"zval \*op\_1；

c）int type；

d）uint32\_t \_arg\_num, zval \*\_arg；

e）uint32\_t try\_catch\_offset, uint32\_t op\_num；

f）空。

4）code：业务逻辑源码，初始为空；

5）inline：助手声名语句中是否有“\_INLINE”关键字，值为true或false；

6）cold：助手声名语句中是否有“\_COLD”关键字，值为true或false；

7）hot：助手声名语句中是否有“\_HOT”关键字，值为true或false。

## 四）解析ZEND\_OP\_DATA操作码

代码如下：

|  |
| --- |
| **// gen\_vm()函数的代码，第5块**  **// 1.4 虚拟机定义操作码：1个**  } else if (strpos($line,"ZEND\_VM\_DEFINE\_OP(") === 0) {  // 检查定义是否合法  if (preg\_match(  "/^ZEND\_VM\_DEFINE\_OP\(\s\*([0-9]+)\s\*,\s\*([A-Z\_]+)\s\*\);/",  $line,  $m) == 0) {  // 如果不合法，报错：无效的虚拟机操作定义  die("ERROR ($def:$lineno): Invalid ZEND\_VM\_DEFINE\_OP definition.\n");  }  // 正则第一段，[0-9]+，第一个参数，操作码编号，如：137  $code = (int)$m[1];  // 正则第二段，[A-Z\_]+，第二个参数，操作码名，如：ZEND\_OP\_DATA  $op = $m[2];  $len = strlen($op); // 操作码长度  // 更新最大长度  if ($len > $max\_opcode\_len) { $max\_opcode\_len = $len; }  // 更新最大值  if ($code > $max\_opcode) { $max\_opcode = $code; }  $opcodes[$code] = array("op"=>$op,"code"=>""); // 创建操作码源码为空  $opnames[$op] = $code; // 创建操作码名 |

如上所示，这部分业务逻辑是定义新操作码，只有一个地方用到：

|  |
| --- |
| ZEND\_VM\_DEFINE\_OP(137, ZEND\_OP\_DATA); |

用正则表达式从以上定义中抽取参数：

1）【ZEND\_VM\_DEFINE\_OP\(\s\*([0-9]+)】匹配第一个参数：操作码编号，数字，如：137；

2）【\s\*,\s\*([A-Z\_]+)\s\*\);】匹配第二个参数：操作码名，大写字母和正蓝线，如：ZEND\_OP\_DATA。

以上业务逻辑会定义一个新操作码，操作码号为137，操作码名为ZEND\_OP\_DATA，对应的业务逻辑为空。

## 五）拼接函数内容源码

代码如下：

|  |
| --- |
| **// gen\_vm()函数的代码，第6块**  // 这两行是添加非定义行的代码行  } else if ($handler !== null) { **// 1.5：如果正在解析处理器**  $opcodes[$handler]["code"] .= $line; // 把 这行代码 添加到当前处理器的代码中  } else if ($helper !== null) { **// 1.6：如果正在解析操作助手**  $helpers[$helper]["code"] .= $line; // 把这行代码添加到 操作助手的代码中  }  }  ksort($opcodes); // 键名排序操作码 |

如上所示，这部分业务逻辑是要把每个处理器或助手的业务逻辑源代码拼接起来，然后把组合好的$opcodes操作码处理器数组按键名排序。

## 小结

这一步操作比较复杂，先遍历一遍zend\_vm\_def.h文件，把所有的方法声名类语句过滤出来，并对每个声名语句进行解析，把解析好的处理器放到$opcodes数组中，把解析好的助手放到$helpers数组中，然后把组合好的$opcodes操作码处理器数组按键名排序。

在处理过程中，所有的处理器和助手都会被添加到$list数组中：

1）对于普通处理器，键名为行号，值为array("handler"=>操作码号)；

2）对于专用处理器，键名为行号，值为array("handler"=>处理器序号,也是处理器在$opcodes中的键名)；

3）对于助手，键名为行号，值为array("helper"=>助手名)。

例如：

|  |
| --- |
| array(273) { // 包含所有的处理器和助手，除了ZEND\_NOP  [26]=> array(1) { ["helper"]=> string(15) "zend\_add\_helper" }  [47]=> array(1) { ["handler"]=> int(1) }  [85]=> array(1) { ["helper"]=> string(15) "zend\_sub\_helper" }  [106]=> array(1) { ["handler"]=> int(2) }  ... |

共解析出240个操作码处理器，普通操作码处理器编号从0（ZEND\_NOP）到202（ZEND\_CALLABLE\_CONVERT），共202个（没有45号）；专用操作码处理器编号从256（ZEND\_RECV\_NOTYPE）到293（ZEND\_FE\_FETCH\_R\_SIMPLE），共38个。共解析出34个助手。

# 二、处理操作码的互相引用

有一些操作码的处理逻辑中会引用其他操作码的处理器，这一步处理这种引用。代码如下：

|  |
| --- |
| **// gen\_vm()函数的代码，第7块**  // 2. 为操作码处理器查找 那些被其他操作码用过的处理器  // Search for opcode handlers those are used by other opcode handlers  // 遍历， 操作码编号索引的 处理器列表  foreach ($opcodes as $dsc) {  // 如果业务逻辑中只有一行调用 ZEND\_VM\_DISPATCH\_TO\_HANDLER()函数  if (preg\_match("/^\s\*{\s\*ZEND\_VM\_DISPATCH\_TO\_HANDLER\(\s\*([A-Z\_]\*)\s\*\)\s\*;\s\*}\s\*/", $dsc["code"], $m)) {  $op = $m[1]; // 第一段 [A-Z\_]\*， 引用的操作码名  // 给当前操作码添加别名，别名为被引用的操作码名  $opcodes[$opnames[$dsc['op']]]['alias'] = $op;  // 如果有其他业务逻辑，但调用了ZEND\_VM\_DISPATCH\_TO\_HANDLER()函数  } else if (preg\_match\_all("/ZEND\_VM\_DISPATCH\_TO\_HANDLER\(\s\*([A-Z\_]\*)\s\*\)/m", $dsc["code"], $mm, PREG\_SET\_ORDER)) {  foreach ($mm as $m) {  $op = $m[1]; // 被引用的操作码名  $code = $opnames[$op]; // 查到被引用的操作码编号  $opcodes[$code]['use'] = 1; // 被引用的操作码添加一个use 键  }  }  } |

如上所示，遍历所有操作码，碰到使用ZEND\_VM\_DISPATCH\_TO\_HANDLER()函数在一个操作码处理器中引用另一个操作码处理器时，分成两种情况处理：

1）如果处理器用只调用了ZEND\_VM\_DISPATCH\_TO\_HANDLER(目标操作码)函数，给当前操作码处理器添加一个别名，别名为被引用的操作码的名称，例如：

|  |
| --- |
| [39]=> array(7) {  ["op"]=> string(24) "ZEND\_PRE\_DEC\_STATIC\_PROP"  ["alias"]=> string(24) "ZEND\_PRE\_INC\_STATIC\_PROP"  ... } |

如上所示，ZEND\_PRE\_DEC\_STATIC\_PROP操作码被标记成ZEND\_PRE\_INC\_STATIC\_PROP操作码的别名；

2）如果处理器除了调用ZEND\_VM\_DISPATCH\_TO\_HANDLER(目标操作码)函数，还有其他业务逻辑，给被引用的操作码处理器添加键use，值为1，例如：

|  |
| --- |
| [67]=> array(7) {  ["op"]=> string(13) "ZEND\_SEND\_REF"  ["use"]=> int(1)  ... } |

如上所示，ZEND\_SEND\_REF操作码被ZEND\_SEND\_VAR\_EX操作码的专用处理器引用过，它的处理器多了一个use键，值为1。

ZEND\_VM\_DISPATCH\_TO\_HANDLER()函数并不是真正的函数，它是在zend\_vm\_def.h文件中使用的伪代码，共有8处调用，统计如下：

|  |  |  |
| --- | --- | --- |
| **操作码** | **引用的操作码** | **别名或use** |
| ZEND\_PRE\_DEC\_OBJ | ZEND\_PRE\_INC\_OBJ | 别名 |
| ZEND\_POST\_DEC\_OBJ | ZEND\_POST\_INC\_OBJ | 别名 |
| ZEND\_PRE\_DEC\_STATIC\_PROP | ZEND\_PRE\_INC\_STATIC\_PROP | 别名 |
| ZEND\_POST\_DEC\_STATIC\_PROP | ZEND\_POST\_INC\_STATIC\_PROP | 别名 |
| ZEND\_FETCH\_DIM\_FUNC\_ARG | ZEND\_FETCH\_DIM\_W | use |
| ZEND\_FETCH\_OBJ\_FUNC\_ARG | ZEND\_FETCH\_OBJ\_W | use |
| ZEND\_INIT\_ARRAY | ZEND\_ADD\_ARRAY\_ELEMENT | use |
| ZEND\_SEND\_VAR\_EX | ZEND\_SEND\_REF | use |

# 三、生成zend\_vm\_opcodes文件

## 一）生成zend\_vm\_opcodes.h文件

代码如下：

|  |
| --- |
| **// gen\_vm()函数的代码，第8块**  // 3. 生成操作码定义 zend\_vm\_opcodes.h  $str = gen\_vm\_opcodes\_header($opcodes, $max\_opcode, $max\_opcode\_len, $vm\_op\_flags);  // 写入文件 zend\_vm\_opcodes.h  write\_file\_if\_changed(\_\_DIR\_\_ . "/zend\_vm\_opcodes\_2.h", $str);  // 生成操作码完毕  echo "zend\_vm\_opcodes.h generated successfully.\n"; |

gen\_vm\_opcodes\_header()函数和write\_file\_if\_changed()函数的业务逻辑比较简单，暂不介绍，生成的zend\_vm\_opcodes.h文件内容也比较简单，内容如下：

|  |
| --- |
| #ifndef ZEND\_VM\_OPCODES\_H /\* 防止重复定义 \*/  #define ZEND\_VM\_OPCODES\_H  #define ZEND\_VM\_SPEC 1 /\* 默认为1 \*/  #define ZEND\_VM\_LINES 0 /\* 默认为0 \*/  // 4种运行模式  #define ZEND\_VM\_KIND\_CALL 1  #define ZEND\_VM\_KIND\_SWITCH 2  #define ZEND\_VM\_KIND\_GOTO 3  #define ZEND\_VM\_KIND\_HYBRID 4  // 64位windows系统中，没有gnuc，所以运行模式为 ZEND\_VM\_KIND\_CALL  #if (defined(\_\_GNUC\_\_) && defined(HAVE\_GCC\_GLOBAL\_REGS))  # define ZEND\_VM\_KIND ZEND\_VM\_KIND\_HYBRID  #else  # define ZEND\_VM\_KIND ZEND\_VM\_KIND\_CALL  #endif  // ...  // 脚本中定义的 $vm\_op\_flags数组，输出成常量，共32个  #define ZEND\_VM\_OP\_SPEC 0x00000001  // ...  // 两个操作对象的标记掩码，op1占最右8位，op2占op1右侧8位  #define ZEND\_VM\_OP1\_FLAGS(flags) (flags & 0xff)  #define ZEND\_VM\_OP2\_FLAGS(flags) ((flags >> 8) & 0xff)  BEGIN\_EXTERN\_C()  // 这3个函数的业务逻辑在 zend\_vm\_opcodes.c中  ZEND\_API const char\* ZEND\_FASTCALL zend\_get\_opcode\_name(zend\_uchar opcode);  ZEND\_API uint32\_t ZEND\_FASTCALL zend\_get\_opcode\_flags(zend\_uchar opcode);  ZEND\_API zend\_uchar zend\_get\_opcode\_id(const char \*name, size\_t length);  END\_EXTERN\_C()  // 以下是$opcodes中的202个普通操作码常量，常量名是操作码名，值为操作码号  #define ZEND\_NOP 0  // ...  #endif |

如上所示，由于64位windows中没有GNUC，所以执行时的模式是ZEND\_VM\_KIND\_CALL。

## 二）生成zend\_vm\_opcodes.c文件

这部分代码比较简单，只是简单的输出内容，生成的zend\_vm\_opcodes.c文件内容也比较简单，文件内容如下：

|  |
| --- |
| #include <stdio.h>  #include <zend.h>  #include <zend\_vm\_opcodes.h>  // $opcodes处理器中，202个普通操作码名加一个NULL  static const char \*zend\_vm\_opcodes\_names[203] = {  "ZEND\_NOP",  ...  };  // $opcodes处理器中，202个普通操作码的flag，NULL的flag为0  static uint32\_t zend\_vm\_opcodes\_flags[203] = {  0x00000000,  ...  };  // 通过操作码号获取操作码名  ZEND\_API const char\* ZEND\_FASTCALL zend\_get\_opcode\_name(zend\_uchar opcode) {  if (UNEXPECTED(opcode > ZEND\_VM\_LAST\_OPCODE)) {  return NULL;  }  return zend\_vm\_opcodes\_names[opcode];  }  // 通过操作码号获取flag  ZEND\_API uint32\_t ZEND\_FASTCALL zend\_get\_opcode\_flags(zend\_uchar opcode) {  if (UNEXPECTED(opcode > ZEND\_VM\_LAST\_OPCODE)) {  opcode = ZEND\_NOP;  }  return zend\_vm\_opcodes\_flags[opcode];  }  // 通过操作码名获取操作码号，找不到时返回203  ZEND\_API zend\_uchar zend\_get\_opcode\_id(const char \*name, size\_t length) {  zend\_uchar opcode;  for (opcode = 0; opcode < (sizeof(zend\_vm\_opcodes\_names) / sizeof(zend\_vm\_opcodes\_names[0])) - 1; opcode++) {  const char \*opcode\_name = zend\_vm\_opcodes\_names[opcode];  if (opcode\_name && strncmp(opcode\_name, name, length) == 0) {  return opcode;  }  }  return ZEND\_VM\_LAST\_OPCODE + 1;  } |

如上所示，操作码的编号是从0到202，但中间缺少45号，所以只有202个编号。关于操作码的更多介绍参见“编译与执行篇”。

# 四、生成 zend\_vm\_execute.h文件

生成 zend\_vm\_execute.h文件是最重要也是最复杂的操作，先打开文件准备写入内容：

|  |
| --- |
| **// gen\_vm()函数的代码，第9块**  // 5. 生成 zend\_vm\_execute.h  // Generate zend\_vm\_execute.h  // 打开文件  $f = fopen(\_\_DIR\_\_ . "/zend\_vm\_execute.h", "w+") or die("ERROR: Cannot create zend\_vm\_execute.h\n");  // 执行器文件绝对路径  $executor\_file = realpath(\_\_DIR\_\_ . "/zend\_vm\_execute.h"); |

写入的内容比较多，下面逐步进行介绍。

## 一）生成文件头部内容

这部分业务逻辑比较简单，只是简单的输出内容：

|  |
| --- |
| **// gen\_vm()函数的代码，第10块**  // 5.1插入头部内容  out($f, HEADER\_TEXT);  ...  out($f, "255\n};\n\n"); |

生成的zend\_vm\_execute.h最前面的一段内容，内容如下：

|  |
| --- |
| // windows系统中一些错误信息的启用和关闭  #ifdef ZEND\_WIN32  # pragma warning(disable : 4101)  # pragma warning(once : 6235)  // ...  #endif  // 256 个用户操作码处理器指针  static user\_opcode\_handler\_t zend\_user\_opcode\_handlers[256] = {  (user\_opcode\_handler\_t)NULL,  // ...  };  // 256 个用户操作码的序号  static zend\_uchar zend\_user\_opcodes[256] = {0,  1,2,...,255  }; |

## 二）调用gen\_executor()函数生成执行器代码

代码如下：

|  |
| --- |
| **// gen\_vm()函数的代码，第11块**  // 5.2 生成专用执行器  // Generate specialized executor  // 参数：zend\_vm\_execute.h文件句柄，zend\_vm\_execute.skl文件内容，ZEND\_VM\_SPEC=1  gen\_executor($f, $skl, ZEND\_VM\_SPEC, ZEND\_VM\_KIND, "execute", "zend\_vm\_init");  out($f, "\n"); |

gen\_executor()函数的业务逻辑比较复杂，将在下一章里重点介绍。

以下生成5个特殊函数。

## 三）生成zend\_vm\_get\_opcode\_handler\_idx()函数代码

这部分代码比较简单，业务逻辑框架如下：

|  |
| --- |
| **// gen\_vm()函数的代码，第12块**  // 获取操作码处理器序号，两个参数  out($f, "static uint32\_t ZEND\_FASTCALL zend\_vm\_get\_opcode\_handler\_idx(uint32\_t spec, const zend\_op\* op)\n");  // 输出内容 ...  if (isset($used\_extra\_spec["OP\_DATA"]) || isset($used\_extra\_spec["RETVAL"]) ||  isset($used\_extra\_spec["QUICK\_ARG"]) || isset($used\_extra\_spec["SMART\_BRANCH"]) ||  isset($used\_extra\_spec["ISSET"]) || isset($used\_extra\_spec["OBSERVER"])) {  $else = "";  out($f, "\tif (spec & SPEC\_EXTRA\_MASK) {\n");  if (isset($used\_extra\_spec["RETVAL"])) {  // 输出内容 ...  $else = "} else ";  }  if (isset($used\_extra\_spec["QUICK\_ARG"])) {  // 输出内容 ...  $else = "} else ";  }  if (isset($used\_extra\_spec["OP\_DATA"])) {  // 输出内容 ...  $else = "} else ";  }  if (isset($used\_extra\_spec["ISSET"])) {  // 输出内容 ...  $else = "} else ";  }  if (isset($used\_extra\_spec["SMART\_BRANCH"])) {  // 输出内容 ...  $else = "} else ";  }  if (isset($used\_extra\_spec["OBSERVER"])) {  // 输出内容 ...  $else = "} else ";  }  if ($else !== "") {  out($f, "\t\t}\n");  }  out($f, "\t}\n");  }  // 输出内容 ... |

如上所示，输出的内容主要受到用到的扩展规则$used\_extra\_spec的影响，变量内容如下：

|  |
| --- |
| array(9) {  ["COMMUTATIVE"]=> int(1)  ["NO\_CONST\_CONST"]=> int(1)  ["SMART\_BRANCH"]=> int(1)  ["RETVAL"]=> int(1)  ["OP\_DATA"]=> int(1)  ["OBSERVER"]=> int(1)  ["QUICK\_ARG"]=> int(1)  ["TYPE"]=> int(1)  ["ISSET"]=> int(1)  } |

上述代码生成的zend\_vm\_get\_opcode\_handler\_idx函数C语言代码如下：

|  |
| --- |
| // 获取opcode句柄  static uint32\_t ZEND\_FASTCALL zend\_vm\_get\_opcode\_handler\_idx(uint32\_t spec, const zend\_op\* op){  static const int zend\_vm\_decode[] = {  \_UNUSED\_CODE, /\* 0 = IS\_UNUSED，UNUSED类型 \*/  \_CONST\_CODE, /\* 1 = IS\_CONST，CONST类型 \*/  \_TMP\_CODE, /\* 2 = IS\_TMP\_VAR，TMP类型 \*/  \_UNUSED\_CODE, /\* 3 \*/  \_VAR\_CODE, /\* 4 = IS\_VAR，VAR类型 \*/  \_UNUSED\_CODE, /\* 5 \*/  \_UNUSED\_CODE, /\* 6 \*/  \_UNUSED\_CODE, /\* 7 \*/  \_CV\_CODE /\* 8 = IS\_CV ，CV类型 \*/  };  uint32\_t offset = 0; // 从0开始  // 用掩码过滤，如果op1有ANY以外的类型  if (spec & SPEC\_RULE\_OP1) offset = offset \* 5 + zend\_vm\_decode[op->op1\_type];  // 用掩码过滤，如果op2有ANY以外的类型  if (spec & SPEC\_RULE\_OP2) offset = offset \* 5 + zend\_vm\_decode[op->op2\_type];  if (spec & SPEC\_EXTRA\_MASK) { // 如果有扩展规则  if (spec & SPEC\_RULE\_RETVAL) { // 有返回结果规则  offset = offset \* 2 + (op->result\_type != IS\_UNUSED);  // 有观察者规则  if ((spec & SPEC\_RULE\_OBSERVER) && ZEND\_OBSERVER\_ENABLED) {  offset += 2;  }  } else if (spec & SPEC\_RULE\_QUICK\_ARG) { // 有传入参数规则  offset = offset \* 2 + (op->op2.num <= MAX\_ARG\_FLAG\_NUM); // 12个参数内  } else if (spec & SPEC\_RULE\_OP\_DATA) { // 有操作数据规则  offset = offset \* 5 + zend\_vm\_decode[(op + 1)->op1\_type];  } else if (spec & SPEC\_RULE\_ISSET) { // 使用isset()或empty()函数  offset = offset \* 2 + (op->extended\_value & ZEND\_ISEMPTY);  } else if (spec & SPEC\_RULE\_SMART\_BRANCH) { // 智能分支  offset = offset \* 3;  if (op->result\_type == (IS\_SMART\_BRANCH\_JMPZ|IS\_TMP\_VAR)) { // 跳转操作码  offset += 1;  // 跳转操作码  } else if (op->result\_type == (IS\_SMART\_BRANCH\_JMPNZ|IS\_TMP\_VAR)) {  offset += 2;  }  } else if (spec & SPEC\_RULE\_OBSERVER) { // 有观察者规则  offset = offset \* 2;  if (ZEND\_OBSERVER\_ENABLED) { // 已启用观察者  offset += 1;  }  }  }  // 添加特殊启动规则  return (spec & SPEC\_START\_MASK) + offset;  } |

如上所示，函数接收两个参数：1）spec表示操作码的第一个处理器在处理器列表中的序号；2）此操作码。函数业务逻辑可归纳如下：

一、如果op1有ANY以外的类型，offset累加op1的类型（这里offset初始是0，offset\*5无意义）；

二、如果op2有ANY以外的类型，offset翻5倍（因为op2有5种基本有效类型：CONST、TMP、VAR、CV、UNUSED，会产生5个分支），再累加op2的类型；

三、如果有扩展规则：

1）如果有【RETVAL规则】：offset翻2倍（因为RETVAL规则有2种状态：启用和未启用，会产生2个分支），累加操作结果有效（op->result\_type != IS\_UNUSED)；这时候如果开启了观察者模式，offset再累加2（这相当于观察者模式的第三种状态：RETVAL+观察者）；

2）如果有【QUICK\_ARG规则】：offset翻2倍（因为QUICK\_ARG规则有2种状态：激活和未激活，会产生2个分支），累加：参数数量是否在12个以内（op->op2.num <= MAX\_ARG\_FLAG\_NUM)；

3）如果有【OP\_DATA规则】：offset翻5倍（因为op\_data有5种有基本效类型：CONST、TMP、VAR、CV、UNUSED，会产生5个分支），累加op\_data的类型；

4）如果有【ISSET规则】：offset翻2倍（因为ISSET规则有2种状态：激活和未激活，会产生2个分支），累加：是否使用empty函数（op->extended\_value & ZEND\_ISEMPTY)；

5）如果有【SMART\_BRANCH规则】：offset翻3倍（因为SMART\_BRANCH规则有3种状态，会产生3个分支）；如果结果的操作码是（IS\_SMART\_BRANCH\_JMPZ|IS\_TMP\_VAR）offset累加1；如果结果的操作码是（IS\_SMART\_BRANCH\_JMPNZ|IS\_TMP\_VAR）offset累加2；

6）如果有【OBSERVER规则】：offset翻2倍（因为OBSERVER规则有2种状态：启用和未启用，会产生2个分支）；如果启用了观察者（ZEND\_OBSERVER\_ENABLED），累加1。

四、返回，offset+spec的右侧16位。

MAX\_ARG\_FLAG\_NUM常量的值是12。SPEC\_EXTRA\_MASK常量用于过滤32位整数的左侧14位，SPEC\_START\_MASK常量用于过滤32位整数的右侧16位，详情参见“gen\_executor()函数-DEFINES标记”。

## 四）生成 zend\_vm\_get\_opcode\_handler\_func()函数代码

代码如下：

|  |
| --- |
| **// gen\_vm()函数的代码，第13块**  // 默认情况下用 ZEND\_VM\_KIND\_HYBRID 模式生成  if (ZEND\_VM\_KIND == ZEND\_VM\_KIND\_HYBRID) {  out($f, "#if ZEND\_VM\_KIND == ZEND\_VM\_KIND\_HYBRID\n");  out($f,"static const void \*zend\_vm\_get\_opcode\_handler\_func(zend\_uchar opcode, const zend\_op\* op)\n");  out($f, "{\n");  out($f, "\tuint32\_t spec = zend\_spec\_handlers[opcode];\n");  out($f, "\treturn zend\_opcode\_handler\_funcs[zend\_vm\_get\_opcode\_handler\_idx(spec, op)];\n");  out($f, "}\n\n");  out($f, "#endif\n\n");  } |

zend\_vm\_get\_opcode\_handler\_func()函数的业务逻辑比较简单，代码如下：

|  |
| --- |
| #if ZEND\_VM\_KIND == ZEND\_VM\_KIND\_HYBRID  static const void \*zend\_vm\_get\_opcode\_handler\_func(zend\_uchar opcode, const zend\_op\* op) {  uint32\_t spec = zend\_spec\_handlers[opcode];  return zend\_opcode\_handler\_funcs[zend\_vm\_get\_opcode\_handler\_idx(spec, op)];  }  #endif |

在zend\_vm\_gen.php脚本中默认用ZEND\_VM\_KIND\_HYBRID模式生成代码，但在64位windows操作系统中，运行时并不是ZEND\_VM\_KIND\_HYBRID模式，而是ZEND\_VM\_KIND\_CALL模式，运行时没有用到这个函数。

## 五）生成zend\_vm\_set\_opcode\_handler()函数代码

代码如下：

|  |
| --- |
| **// gen\_vm()函数的代码，第14块**  // 5.5 生成 特殊函数3: zend\_vm\_set\_opcode\_handler 函数  out($f, "ZEND\_API void ZEND\_FASTCALL zend\_vm\_set\_opcode\_handler(zend\_op\* op)\n");  out($f, "{\n");  out($f, "\tzend\_uchar opcode = zend\_user\_opcodes[op->opcode];\n");  out($f, "\n");  out($f, "\tif (zend\_spec\_handlers[op->opcode] & SPEC\_RULE\_COMMUTATIVE) {\n");  out($f, "\t\tif (op->op1\_type < op->op2\_type) {\n");  out($f, "\t\t\tzend\_swap\_operands(op);\n");  out($f, "\t\t}\n");  out($f, "\t}\n");  out($f, "\top->handler = zend\_opcode\_handlers[zend\_vm\_get\_opcode\_handler\_idx(zend\_spec\_handlers[opcode], op)];\n");  out($f, "}\n\n"); |

zend\_vm\_set\_opcode\_handler()函数用于给操作码链接它对应的处理器，函数的业务逻辑比较简单，代码如下：

|  |
| --- |
| ZEND\_API void ZEND\_FASTCALL zend\_vm\_set\_opcode\_handler(zend\_op\* op){  zend\_uchar opcode = zend\_user\_opcodes[op->opcode];　// 用户操作码  // 如果是可交换两个操作对象  if (zend\_spec\_handlers[op->opcode] & SPEC\_RULE\_COMMUTATIVE) {  if (op->op1\_type < op->op2\_type) {　// 让op\_type大的放在前面。  zend\_swap\_operands(op);　// 交换两个操作对象  }  }  // 关联特定的处理器  op->handler = zend\_opcode\_handlers[zend\_vm\_get\_opcode\_handler\_idx(  zend\_spec\_handlers[opcode], op)];  } |

如上所示，函数中用到的zend\_user\_opcodes、zend\_spec\_handlers变量都在zend\_vm\_init()函数中填充内容，详情可参见“生成zend\_vm\_init()函数”章节。

## 六）生成zend\_vm\_set\_opcode\_handler\_ex()函数代码

zend\_vm\_set\_opcode\_handler\_ex()函数在优化器（Optimizer）中用到，与核心业务逻辑关联不大，暂时略过。

## 七）生成zend\_vm\_call\_opcode\_handler()函数代码

zend\_vm\_call\_opcode\_handler()函数只有在调试时用到，与核心业务逻辑关联不大，暂时略过。

# 五、gen\_executor()函数

gen\_executor()函数是本篇重点，用于以zend\_vm\_execute.skl文件为框架，生成zend\_vm\_execute.h文件中的主体业务逻辑，函数的业务逻辑框架如下：

|  |
| --- |
| // 前3个参数：zend\_vm\_execute.h文件句柄，zend\_vm\_execute.skl文件内容，1  function gen\_executor($f, $skl, $spec, $kind, $executor\_name, $initializer\_name) {  global $params, $skeleton\_file, $line\_no, $gen\_order;  $gen\_order = null; // 默认没有 zend\_vm\_order.txt 文件  $switch\_labels = array();  $lineno = 0;  foreach ($skl as $line) { // 遍历 zend\_vm\_execute.skl文件每一行  // 找到这些标记，每行最多1个  if (preg\_match("/(.\*)[{][%]([A-Z\_]\*)[%][}](.\*)/", $line, $m)) {  switch ($m[2]) { // 按标记名处理，第二段，标记名  case "DEFINES": // 业务逻辑代码 ...  break;  case "EXECUTOR\_NAME":  out($f, $m[1].$executor\_name.$m[3]."\n");  break;  case "HELPER\_VARS": // 业务逻辑代码 ...  break;  case "INTERNAL\_LABELS": // 业务逻辑代码 ...  break;  case "ZEND\_VM\_CONTINUE\_LABEL": // 业务逻辑代码 ...  break;  case "ZEND\_VM\_DISPATCH": // 业务逻辑代码 ...  break;  case "INTERNAL\_EXECUTOR": // 业务逻辑代码 ...  break;  case "EXTERNAL\_EXECUTOR": // 其他运行模式 ...  break;  case "INITIALIZER\_NAME":  // $initializer\_name 是传入的变量  out($f, $m[1].$initializer\_name.$m[3]."\n");  break;  case "EXTERNAL\_LABELS": // 业务逻辑代码 ...  break;  }  } else { // 如果没有特殊标记，直接复制这一行  out($f, $line); // 原样输出内容  }  }  } |

如上所示，业务逻辑可归纳为，遍历zend\_vm\_execute.skl文件每一行，在行中查找{%NAME%}模式的标签，每行最多1个：

1）如果找到，把标记替换为指定的内容，把替换好的内容输出到zend\_vm\_execute.h文件；

2）如果没找到标签，把内容直接输出到zend\_vm\_execute.h文件。

支持的标记共有以下10个：

|  |  |  |
| --- | --- | --- |
| **标记** | **说明** | **引用次数** |
| DEFINES | 常量、操作码处理器、助手，共生成52000余行代码 | 1 |
| EXECUTOR\_NAME | 固定字串"execute"  用于生成execute\_ex()函数中的内容 | 3 |
| HELPER\_VARS | C程序用到的变量，根据不同环境做了很多优化  用于生成execute\_ex()函数中的内容 | 1 |
| INTERNAL\_LABELS | 生成3451个标签  用于生成execute\_ex()函数中的内容 | 1 |
| ZEND\_VM\_CONTINUE\_LABEL | 声名变量  用于生成execute\_ex()函数中的内容 | 1 |
| ZEND\_VM\_DISPATCH | 调用操作码处理器  用于生成execute\_ex()函数中的内容 | 1 |
| INTERNAL\_EXECUTOR | 跳转到下一个操作码，或退出  用于生成execute\_ex()函数中的内容 | 1 |
| EXTERNAL\_EXECUTOR | 默认模式下用不到 | 1 |
| INITIALIZER\_NAME | 固定字串"zend\_vm\_init"，函数名 | 1 |
| EXTERNAL\_LABELS |  | 1 |

下面以zend\_vm\_execute.skl文件中的内容框架为线索，依次详细介绍。

## 一）DEFINES标记

DEFINES标记是zend\_vm\_execute.skl文件中最先用到的标记。业务逻辑的前半段用来定义一些常量和宏程序，后半段用来创建操作码处理器和助手，函盖了zend\_vm\_execute.h中90%以上的内容。

### 1、定义常量和宏程序

这部分代码比较简单，生成的内容如下：

|  |
| --- |
| // 一些常量  #define SPEC\_START\_MASK 0x0000ffff // 掩码，过滤右侧16位  #define SPEC\_EXTRA\_MASK 0xfffc0000 // 掩码，过滤左侧14位（抛掉右面两位）  #define SPEC\_RULE\_OP1 0x00010000 // 左侧第16位，表示op1有指定类型  #define SPEC\_RULE\_OP2 0x00020000 // 左侧第15位，表示op2有指定类型  #define SPEC\_RULE\_OP\_DATA 0x00040000 // 左侧第14位，表示op\_data有指定类型  #define SPEC\_RULE\_RETVAL 0x00080000 // 左侧第13位，表示op\_data有指定类型  #define SPEC\_RULE\_QUICK\_ARG 0x00100000 // 左侧第12位，表示QUICK\_ARG规则  #define SPEC\_RULE\_SMART\_BRANCH 0x00200000 // 左侧第11位，SMART\_BRANCH规则  #define SPEC\_RULE\_COMMUTATIVE 0x00800000 // 左侧第9位，COMMUTATIVE规则  #define SPEC\_RULE\_ISSET 0x01000000 // 左侧第8位，ISSET规则  #define SPEC\_RULE\_OBSERVER 0x02000000 // 左侧第7位，OBSERVER规则  static const uint32\_t \*zend\_spec\_handlers;  static const void \* const \*zend\_opcode\_handlers;  static int zend\_handlers\_count;  #if (ZEND\_VM\_KIND == ZEND\_VM\_KIND\_HYBRID) /\* windows用不到 \*/  ...  #endif  // windows走这里  #if (ZEND\_VM\_KIND != ZEND\_VM\_KIND\_HYBRID) || !ZEND\_VM\_SPEC  static const void \*zend\_vm\_get\_opcode\_handler(zend\_uchar opcode, const zend\_op\* op);  #endif  #if (ZEND\_VM\_KIND == ZEND\_VM\_KIND\_HYBRID) /\* windows用不到 \*/  ...  #else /\* windows走这里 \*/  **// 使用zend\_vm\_get\_opcode\_handler()函数代替zend\_vm\_get\_opcode\_handler\_func()函数**  # define zend\_vm\_get\_opcode\_handler\_func zend\_vm\_get\_opcode\_handler  #endif  // windows里3个空的宏程序  #ifndef VM\_TRACE  # define VM\_TRACE(op)  #endif  #ifndef VM\_TRACE\_START  # define VM\_TRACE\_START()  #endif  #ifndef VM\_TRACE\_END  # define VM\_TRACE\_END()  #endif  #if (ZEND\_VM\_KIND == ZEND\_VM\_KIND\_HYBRID) /\* windows用不到 \*/  ...  #endif  #ifdef ZEND\_VM\_FP\_GLOBAL\_REG /\* windows用不到 \*/  ...  #else /\* windows走这里，处理器参数的定义和读取 \*/  # define ZEND\_OPCODE\_HANDLER\_ARGS zend\_execute\_data \*execute\_data  # define ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU execute\_data  # define ZEND\_OPCODE\_HANDLER\_ARGS\_DC , ZEND\_OPCODE\_HANDLER\_ARGS  # define ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU\_CC , ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU  #endif  // gcc用的，windows里没有  #if defined(ZEND\_VM\_FP\_GLOBAL\_REG) && defined(ZEND\_VM\_IP\_GLOBAL\_REG)  ...  #else /\* window走这里 \*/  # define ZEND\_OPCODE\_HANDLER\_RET int  # define ZEND\_VM\_TAIL\_CALL(call) return call  # define ZEND\_VM\_CONTINUE() return 0  # define ZEND\_VM\_RETURN() return -1  # define ZEND\_VM\_HOT  // windows: 空  # define ZEND\_VM\_COLD ZEND\_COLD ZEND\_OPT\_SIZE  #endif  typedef ZEND\_OPCODE\_HANDLER\_RET (ZEND\_FASTCALL \*opcode\_handler\_t) (ZEND\_OPCODE\_HANDLER\_ARGS);  #define DCL\_OPLINE  // #define EX(element) ((execute\_data)->element)  // windows 不走这里  #ifdef ZEND\_VM\_IP\_GLOBAL\_REG  ...  #else /\* windows走这里，操作码相关的宏程序 \*/  # define OPLINE EX(opline)  // 获取当前正在执行的操作码  # define USE\_OPLINE const zend\_op \*opline = EX(opline);  # define LOAD\_OPLINE()  # define LOAD\_OPLINE\_EX()  // zend\_execute.c中定义：OPLINE++  # define LOAD\_NEXT\_OPLINE() ZEND\_VM\_INC\_OPCODE()  # define SAVE\_OPLINE()  # define SAVE\_OPLINE\_EX()  #endif  // windows中: return 0;  #define HANDLE\_EXCEPTION() ZEND\_ASSERT(EG(exception)); LOAD\_OPLINE(); ZEND\_VM\_CONTINUE()  // windows中: return 2;  #define HANDLE\_EXCEPTION\_LEAVE() ZEND\_ASSERT(EG(exception)); LOAD\_OPLINE(); ZEND\_VM\_LEAVE()  #if defined(ZEND\_VM\_FP\_GLOBAL\_REG) /\* windows用不到 \*/  ...  #elif defined(ZEND\_VM\_IP\_GLOBAL\_REG) /\* windows用不到 \*/  ...  #else /\* windows走这里 \*/  # define ZEND\_VM\_ENTER\_EX() return 1  # define ZEND\_VM\_ENTER() return 1  # define ZEND\_VM\_LEAVE() return 2  #endif  // windows中: return zend\_interrupt\_helper\_SPEC(execute\_data)  #define ZEND\_VM\_INTERRUPT() ZEND\_VM\_TAIL\_CALL(zend\_interrupt\_helper\_SPEC(ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU));  // windows中: zend\_interrupt\_helper\_SPEC(execute\_data);  #define ZEND\_VM\_LOOP\_INTERRUPT() zend\_interrupt\_helper\_SPEC(ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU);  // 分派，处理操作码并return  // windows: return p1  #define ZEND\_VM\_DISPATCH(opcode, opline) ZEND\_VM\_TAIL\_CALL(((opcode\_handler\_t)zend\_vm\_get\_opcode\_handler\_func(opcode, opline))(ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU));  // 这两函数的业务逻辑在后面定义  static ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL zend\_interrupt\_helper\_SPEC(ZEND\_OPCODE\_HANDLER\_ARGS);  static ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL ZEND\_NULL\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS); |

### 2、生成处理器和助手

代码如下：

|  |
| --- |
| // 如果用 HYBRID 模式生成  if ($kind == ZEND\_VM\_KIND\_HYBRID) {  // 生成执行器代码，这是最重要的部分 zend\_vm\_execute.h文件句柄, 1, 运行模式, 标签前面的内容。**注意这里传入的运行模式是ZEND\_VM\_KIND\_CALL**  gen\_executor\_code($f, $spec, ZEND\_VM\_KIND\_CALL, $m[1]);  out($f,"\n");  out($f,"#if (ZEND\_VM\_KIND == ZEND\_VM\_KIND\_HYBRID)\n");  out($f,"# undef ZEND\_VM\_TAIL\_CALL\n");  out($f,"# undef ZEND\_VM\_CONTINUE\n");  out($f,"# undef ZEND\_VM\_RETURN\n");  out($f,"\n");  out($f,"# define ZEND\_VM\_TAIL\_CALL(call) call; ZEND\_VM\_CONTINUE()\n");  out($f,"# define ZEND\_VM\_CONTINUE() HYBRID\_NEXT()\n");  out($f,"# define ZEND\_VM\_RETURN() goto HYBRID\_HALT\_LABEL\n");  out($f,"#endif\n\n");  } |

gen\_executor\_code()函数的调用逻辑比较复杂，在下一章中单独介绍。

## 二）生成execute\_ex()函数

zend\_vm\_execute.skl中定义的框架代码如下：

|  |
| --- |
| // 这几行windows中用不到  #if (ZEND\_VM\_KIND != ZEND\_VM\_KIND\_CALL) && (ZEND\_GCC\_VERSION >= 4000) && !defined(\_\_clang\_\_)  # pragma GCC push\_options  # pragma GCC optimize("no-gcse")  # pragma GCC optimize("no-ivopts")  #endif  // execute\_ex()函数  ZEND\_API void {%EXECUTOR\_NAME%}\_ex(zend\_execute\_data \*ex){  DCL\_OPLINE  {%HELPER\_VARS%}  {%INTERNAL\_LABELS%}  LOAD\_OPLINE();  ZEND\_VM\_LOOP\_INTERRUPT\_CHECK();  while (1) {  {%ZEND\_VM\_CONTINUE\_LABEL%}  {%ZEND\_VM\_DISPATCH%} {  {%INTERNAL\_EXECUTOR%}  }  }  zend\_error\_noreturn(E\_CORE\_ERROR, "Arrived at end of main loop which shouldn't happen");  }  // 这几行windows中用不到  #if (ZEND\_VM\_KIND != ZEND\_VM\_KIND\_CALL) && (ZEND\_GCC\_VERSION >= 4000) && !defined(\_\_clang\_\_)  # pragma GCC pop\_options  #endif |

代码中用到了EXECUTOR\_NAME、HELPER\_VARS、INTERNAL\_LABELS、ZEND\_VM\_CONTINUE\_LABEL、ZEND\_VM\_DISPATCH、INTERNAL\_EXECUTOR，6个标记。

### 1、EXECUTOR\_NAME标记

EXECUTOR\_NAME标记的处理逻辑如下：

|  |
| --- |
| case "EXECUTOR\_NAME":  out($f, $m[1].$executor\_name.$m[3]."\n");  break; |

$executor\_name参数是调用gen\_executor()函数时传入的，代码如下：

|  |
| --- |
| gen\_executor($f, $skl, ZEND\_VM\_SPEC, ZEND\_VM\_KIND, "execute", "zend\_vm\_init"); |

如上所示，传入的值为固定字串"execute"，所以EXECUTOR\_NAME标记都会被替换成"execute"。

### 2、HELPER\_VARS标记

代码如下：

|  |
| --- |
| case "HELPER\_VARS":  if ($kind == ZEND\_VM\_KIND\_SWITCH) { // ZEND\_VM\_KIND\_HYBRID 模式用不到这里  // ...  }  if ($kind != ZEND\_VM\_KIND\_CALL && count($params)) { // $params为空数组，这块用不到  // ...  }  // 如果不是用 ZEND\_VM\_KIND\_CALL 也不是用 ZEND\_VM\_KIND\_HYBRID 模式生成  if ($kind != ZEND\_VM\_KIND\_CALL && $kind != ZEND\_VM\_KIND\_HYBRID) {  // 其他运行模式 ...  } else { // 如果是用 ZEND\_VM\_KIND\_CALL 或 ZEND\_VM\_KIND\_HYBRID 模式生成  **// HELPER\_VARS，这整块代码中只有这块的输出有效**  out($f,"#if defined(ZEND\_VM\_IP\_GLOBAL\_REG) || defined(ZEND\_VM\_FP\_GLOBAL\_REG)\n");  out($f,$m[1]."struct {\n");  // 输出内容 ...  }  break; |

如上所示，用到的助手参数$param为空，最终输出的内容如下：

|  |
| --- |
| #if defined(ZEND\_VM\_IP\_GLOBAL\_REG) || defined(ZEND\_VM\_FP\_GLOBAL\_REG)  struct {  #ifdef ZEND\_VM\_IP\_GLOBAL\_REG  const zend\_op \*orig\_opline;  #endif  #ifdef ZEND\_VM\_FP\_GLOBAL\_REG  zend\_execute\_data \*orig\_execute\_data;  #ifdef ZEND\_VM\_HYBRID\_JIT\_RED\_ZONE\_SIZE  char hybrid\_jit\_red\_zone[ZEND\_VM\_HYBRID\_JIT\_RED\_ZONE\_SIZE];  #endif  #endif  } vm\_stack\_data;  #endif  #ifdef ZEND\_VM\_IP\_GLOBAL\_REG  vm\_stack\_data.orig\_opline = opline;  #endif  #ifdef ZEND\_VM\_FP\_GLOBAL\_REG  vm\_stack\_data.orig\_execute\_data = execute\_data;  execute\_data = ex;  #else  **// windows中没用到gcc，实际用到的代码只有以下一行**  **zend\_execute\_data \*execute\_data = ex;**  #endif |

以上是zend\_vm\_execute.h中实际生成的代码，但windows中没有用到gcc，所以大部分代码都没有用到，实际用到的代码只有以下一行：

|  |
| --- |
| zend\_execute\_data \*execute\_data = ex; |

### 3、INTERNAL\_LABELS标记

代码如下：

|  |
| --- |
| case "INTERNAL\_LABELS":  // 如果用 ZEND\_VM\_KIND\_GOTO 或 ZEND\_VM\_KIND\_HYBRID 模式生成  if ($kind == ZEND\_VM\_KIND\_GOTO || $kind == ZEND\_VM\_KIND\_HYBRID) {  // 如果用 ZEND\_VM\_KIND\_HYBRID 模式生成  if ($kind == ZEND\_VM\_KIND\_HYBRID) {  **// 如果运行时是 ZEND\_VM\_KIND\_HYBRID 模式，windows中用不到这块代码**  out($f,"#if (ZEND\_VM\_KIND == ZEND\_VM\_KIND\_HYBRID)\n");  }  $prolog = $m[1];  out($f,$prolog."if (UNEXPECTED(execute\_data == NULL)) {\n");  out($f,$prolog."\tstatic const void \* const labels[] = {\n");  **// 需要注意ZEND\_VM\_KIND\_HYBRID模式时传入的是ZEND\_VM\_KIND\_GOTO模式**  gen\_labels($f, $spec, ($kind == ZEND\_VM\_KIND\_HYBRID) ? ZEND\_VM\_KIND\_GOTO : $kind, $prolog."\t\t", $specs);  out($f,$prolog."\t};\n");  out($f,$prolog."\tzend\_opcode\_handlers = (const void \*\*) labels;\n");  out($f,$prolog."\tzend\_handlers\_count = sizeof(labels) / sizeof(void\*);\n");  if ($kind == ZEND\_VM\_KIND\_HYBRID) { // 如果用 ZEND\_VM\_KIND\_HYBRID 模式生成  out($f,$prolog."\tmemset(&hybrid\_halt\_op, 0, sizeof(hybrid\_halt\_op));\n");  // 输出内容 ...  } else {  // 其他运行模式 ...  }  out($f,$prolog."}\n");  // 如果用 ZEND\_VM\_KIND\_HYBRID 模式生成  if ($kind == ZEND\_VM\_KIND\_HYBRID) {  out($f,"#endif\n");  }  } else {  skip\_blanks($f, $m[1], $m[3]);  }  break; |

这部分生成的代码如下：

|  |
| --- |
| **// windows中执行模式为ZEND\_VM\_KIND\_CALL，所以用不到这块代码**  #if (ZEND\_VM\_KIND == ZEND\_VM\_KIND\_HYBRID)  if (UNEXPECTED(execute\_data == NULL)) {  static const void \* const labels[] = { **// 重点是这里生的labels共3451 个**  (void\*)&&ZEND\_NOP\_SPEC\_LABEL,  ...  };  zend\_opcode\_handlers = (const void \*\*) labels;  zend\_handlers\_count = sizeof(labels) / sizeof(void\*);  memset(&hybrid\_halt\_op, 0, sizeof(hybrid\_halt\_op));  hybrid\_halt\_op.handler = (void\*)&&HYBRID\_HALT\_LABEL;  #ifdef ZEND\_VM\_HYBRID\_JIT\_RED\_ZONE\_SIZE  memset(vm\_stack\_data.hybrid\_jit\_red\_zone, 0, ZEND\_VM\_HYBRID\_JIT\_RED\_ZONE\_SIZE);  #endif  if (zend\_touch\_vm\_stack\_data) {  zend\_touch\_vm\_stack\_data(&vm\_stack\_data);  }  goto HYBRID\_HALT\_LABEL;  }  #endif |

如上所示，这部分调用gen\_labels()函数，生成3400多个函数指针的指针，这块业务逻辑比较复杂，但只有ZEND\_VM\_KIND\_HYBRID运行模式下才用到，**windows中默认使用ZEND\_VM\_KIND\_CALL模式运行，这整块代码都用不到**。

由于生成C语言源码时使用ZEND\_VM\_KIND\_HYBRID模式，所以skip\_blanks()函数也没有用到。

#### **skip\_blanks()函数**

skip\_blanks()函数用于，不解析标签，直接把标签前后的内容拼接起来，代码如下：

|  |
| --- |
| // clear, 输出：跳过空行  function skip\_blanks($f, $prolog, $epilog) {  if (trim($prolog) != "" || trim($epilog) != "") {  out($f, $prolog.$epilog);  }  } |

### 4、ZEND\_VM\_CONTINUE\_LABEL标记

代码如下：

|  |
| --- |
| case "ZEND\_VM\_CONTINUE\_LABEL":  // 如果用 ZEND\_VM\_KIND\_CALL 或 ZEND\_VM\_KIND\_HYBRID 模式生成  if ($kind == ZEND\_VM\_KIND\_CALL || $kind == ZEND\_VM\_KIND\_HYBRID) {  out($f,"#if !defined(ZEND\_VM\_FP\_GLOBAL\_REG) || !defined(ZEND\_VM\_IP\_GLOBAL\_REG)\n");  out($f,$m[1]."\tint ret;".$m[3]."\n");  out($f,"#endif\n");  }  // 其他运行模式 ...  break; |

这部分业务逻辑比较简单，生成的内容如下：

|  |
| --- |
| #if !defined(ZEND\_VM\_FP\_GLOBAL\_REG) || !defined(ZEND\_VM\_IP\_GLOBAL\_REG)  int ret;  #endif |

以上代码在windows中运行时会用到。

### 5、ZEND\_VM\_DISPATCH标记

代码如下：

|  |
| --- |
| case "ZEND\_VM\_DISPATCH":  switch ($kind) { // ZEND\_VM\_KIND\_HYBRID模式用到下面两个case中的业务逻辑  // 其他运行模式 ...  // 如果用 HYBRID 模式生成  case ZEND\_VM\_KIND\_HYBRID:  // 如果运行时是 ZEND\_VM\_KIND\_HYBRID 模式  out($f,"#if (ZEND\_VM\_KIND == ZEND\_VM\_KIND\_HYBRID)\n");  // 把标签替换成 HYBRID\_SWITCH()  out($f, $m[1]."HYBRID\_SWITCH()".$m[3]."\n");  out($f,"#else\n");  case ZEND\_VM\_KIND\_CALL:  out($f,"#if defined(ZEND\_VM\_FP\_GLOBAL\_REG) && defined(ZEND\_VM\_IP\_GLOBAL\_REG)\n");  // 输出内容 ...  break;  }  break; |

这部分业务逻辑比较简单，生成的内容如下：

|  |
| --- |
| #if (ZEND\_VM\_KIND == ZEND\_VM\_KIND\_HYBRID)  HYBRID\_SWITCH() {  #else  #if defined(ZEND\_VM\_FP\_GLOBAL\_REG) && defined(ZEND\_VM\_IP\_GLOBAL\_REG)  ((opcode\_handler\_t)OPLINE->handler)(ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU);  if (UNEXPECTED(!OPLINE)) {  **#else /\* windows中只用到下面这一行 \*/**  **if (UNEXPECTED((ret = ((opcode\_handler\_t)OPLINE->handler)(ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU)) != 0)) {**  #endif  #endif |

如上所示在windows中运行时，仅用到一行代码：

|  |
| --- |
| if (UNEXPECTED((ret = ((opcode\_handler\_t)OPLINE->handler)(  ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU)) != 0)) { |

### 6、INTERNAL\_EXECUTOR标记

代码如下：

|  |
| --- |
| // 内部执行器  case "INTERNAL\_EXECUTOR":  if ($kind != ZEND\_VM\_KIND\_CALL) { // 如果不是用 ZEND\_VM\_KIND\_CALL 模式生成  if ($kind == ZEND\_VM\_KIND\_HYBRID) { // 如果用 ZEND\_VM\_KIND\_HYBRID 模式生成  // 如果运行时是 ZEND\_VM\_KIND\_HYBRID 模式  out($f,"#if (ZEND\_VM\_KIND == ZEND\_VM\_KIND\_HYBRID)\n");  }  // zend\_vm\_execute.h文件句柄, 1, ZEND\_VM\_KIND, 标签前面的内容  // $switch\_labels 是引用返回  gen\_executor\_code($f, $spec, $kind, $m[1], $switch\_labels);  }  // 如果用 ZEND\_VM\_KIND\_CALL 或 ZEND\_VM\_KIND\_HYBRID 模式生成  if ($kind == ZEND\_VM\_KIND\_CALL || $kind == ZEND\_VM\_KIND\_HYBRID) {  if ($kind == ZEND\_VM\_KIND\_HYBRID) { // 如果用 ZEND\_VM\_KIND\_HYBRID 模式生成  out($f,"#else\n");  }  //  out($f,  $m[1]."execute\_data = vm\_stack\_data.orig\_execute\_data;\n" .  // 拼接其他内容 ...  "#endif\n");  if ($kind == ZEND\_VM\_KIND\_HYBRID) { // 如果用 ZEND\_VM\_KIND\_HYBRID 模式生成  out($f,"#endif\n");  }  }  break; |

输出的内容如下：

|  |
| --- |
| #if (ZEND\_VM\_KIND == ZEND\_VM\_KIND\_HYBRID) /\* windows中不走这里 \*/  HYBRID\_CASE(ZEND\_ASSIGN\_STATIC\_PROP\_OP\_SPEC):  VM\_TRACE(ZEND\_ASSIGN\_STATIC\_PROP\_OP\_SPEC)  ZEND\_ASSIGN\_STATIC\_PROP\_OP\_SPEC\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU);  HYBRID\_BREAK();  **// 省略大量代码 ...**  **#else /\* windows中，使用ZEND\_VM\_KIND\_CALL 运行模式，走这里 \*/**  #ifdef ZEND\_VM\_FP\_GLOBAL\_REG /\* windows中不走这里 \*/  execute\_data = vm\_stack\_data.orig\_execute\_data;  # ifdef ZEND\_VM\_IP\_GLOBAL\_REG  opline = vm\_stack\_data.orig\_opline;  # endif  return;  **#else /\* windows中，走这里 \*/**  **if (EXPECTED(ret > 0)) {**  **execute\_data = EG(current\_execute\_data);**  **ZEND\_VM\_LOOP\_INTERRUPT\_CHECK();**  **} else {**  # ifdef ZEND\_VM\_IP\_GLOBAL\_REG /\* windows中不走这里 \*/  opline = vm\_stack\_data.orig\_opline;  # endif  **return;**  **}**  #endif  #endif |

如上所示，调用gen\_executor\_code()函数时生成了大量代码，但在windows中使用ZEND\_VM\_KIND\_CALL 运行模式，这部分代码都用不到。

windows中运行如下代码：

|  |
| --- |
| **if (EXPECTED(ret > 0)) {**  **execute\_data = EG(current\_execute\_data);**  **ZEND\_VM\_LOOP\_INTERRUPT\_CHECK();**  **} else {**  **return;**  **}** |

### 小结

经过以上复杂的业务逻辑，生成的execute\_ex()函数代码量非常大，PHP8.2.5的源码中，在zend\_execute.h文件中从52308行到60367，共8059行代码。但在windows中，不使用gnuc语言，并且使用ZEND\_VM\_KIND\_CALL模式运行，有效代码只有十几行。

windows中的有效代码如下：

|  |
| --- |
| void execute\_ex(zend\_execute\_data \*ex) {  zend\_execute\_data \*execute\_data = ex;  ZEND\_VM\_LOOP\_INTERRUPT\_CHECK();  while (1) {  int ret;  if (UNEXPECTED((ret = ((opcode\_handler\_t)OPLINE->handler)(  ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU)) != 0)) {  if (EXPECTED(ret > 0)) {  execute\_data = EG(current\_execute\_data);  ZEND\_VM\_LOOP\_INTERRUPT\_CHECK();  } else {  return;  }  }  }  zend\_error\_noreturn(E\_CORE\_ERROR, "Arrived at end of main loop which shouldn't happen");  } |

虽然在windows中execute\_ex()函数的业务逻辑非常简单，但大部分生产环境中，PHP是在linux环境中运行的，所以了解这整个生成过成对深入学习PHP底层源码会有帮助。

## 三）生成zend\_execute()函数

zend\_execute()函数中只用到EXECUTOR\_NAME这一个标记，所以生成此函数的过程非常简单，zend\_vm\_execute.skl中的代码如下：

|  |
| --- |
| **ZEND\_API void zend\_{%EXECUTOR\_NAME%}(zend\_op\_array \*op\_array,**  **zval \*return\_value)**{  // 省略原样输出的代码 ...  **zend\_{%EXECUTOR\_NAME%}\_ex(execute\_data);**  /\* Observer end handlers are called from ZEND\_RETURN \*/  zend\_vm\_stack\_free\_call\_frame(execute\_data);  } |

生成的代码如下：

|  |
| --- |
| ZEND\_API void zend\_execute(zend\_op\_array \*op\_array, zval \*return\_value){  // 省略原样输出的代码 ..  zend\_execute\_ex(execute\_data);  /\* Observer end handlers are called from ZEND\_RETURN \*/  zend\_vm\_stack\_free\_call\_frame(execute\_data);  } |

zend\_execute()函数在PHP虚拟机运行中很重要，用于在外部启动虚拟机。大部分情况下zend\_execute\_ex()函数就是前文中生成的execute\_ex()函数，在一些特殊模式下运行时，也会被替换成其他函数。

更多内容参见“编译和执行篇”。

## 四）EXTERNAL\_EXECUTOR标记

代码如下：

|  |
| --- |
| case "EXTERNAL\_EXECUTOR":  // 默认运行模式是ZEND\_VM\_KIND\_HYBRID，这块代码用不到  if ($kind == ZEND\_VM\_KIND\_CALL) {  gen\_executor\_code($f, $spec, $kind, $m[1]);  }  break; |

如上所示，在ZEND\_VM\_KIND\_HYBRID模式下运行时，这块业务逻辑会被跳过，所以暂不介绍。

## 五）生成zend\_vm\_init()函数

生成zend\_vm\_init()函数需要用到INITIALIZER\_NAME、EXTERNAL\_LABELS两个标记。zend\_vm\_execute.skl中的代码如下：

|  |
| --- |
| void {%INITIALIZER\_NAME%}(void) {  {%EXTERNAL\_LABELS%}  VM\_TRACE\_START();  } |

### 1、INITIALIZER\_NAME标记

代码如下：

|  |
| --- |
| case "INITIALIZER\_NAME":  // $initializer\_name 是传入的参数  out($f, $m[1].$initializer\_name.$m[3]."\n");  break; |

如上所示，$initializer\_name参数是调用gen\_executor()函数时传入的参数：

|  |
| --- |
| gen\_executor($f, $skl, ZEND\_VM\_SPEC, ZEND\_VM\_KIND, "execute", "zend\_vm\_init"); |

所以INITIALIZER\_NAME会被替换成"zend\_vm\_init"。

### 2、EXTERNAL\_LABELS标记

标记的处理逻辑如下：

|  |
| --- |
| case "EXTERNAL\_LABELS":  // 生成初始化 zend\_opcode\_handlers 数组的代码  $prolog = $m[1]; // 标签前面的部分代码  if ($kind == ZEND\_VM\_KIND\_GOTO) { // ZEND\_VM\_KIND\_GOTO 模式，没用到  // 其他运行模式 ...  } else { **// 如果运行时不是 ZEND\_VM\_KIND\_GOTO 模式，用到**  // 定义 labels  out($f,$prolog."static const void \* const labels[] = {\n");  **// $specs 是在这里引用返回的, 这里生成430行代码**  **gen\_labels($f, $spec, ($kind == ZEND\_VM\_KIND\_HYBRID) ?**  **ZEND\_VM\_KIND\_CALL : $kind, $prolog."\t", $specs, $switch\_labels);**  out($f,$prolog."};\n");  out($f,$prolog."static const uint32\_t specs[] = {\n");  **gen\_specs($f, $prolog."\t", $specs); // 生成specs数组元素**  out($f,$prolog."};\n");  // 如果用 ZEND\_VM\_KIND\_HYBRID 模式生成  if ($kind == ZEND\_VM\_KIND\_HYBRID) {  // 输出内容 ...  }  // 输出内容 ...  // 如果用 ZEND\_VM\_KIND\_HYBRID 模式生成  if ($kind == ZEND\_VM\_KIND\_HYBRID) {  out($f,"#endif\n");  }  }  break; |

如上所示，业务逻辑中主要用到gen\_labels()函数和gen\_specs()函数生成相关内容，其中gen\_specs()函数中使用的数据$specs是调用gen\_labels()函数时引用返回的。gen\_labels()函数的业务逻辑比较复杂，将在后续章节中详细介绍。

#### **gen\_specs()函数**

gen\_specs()函数用于生成zend\_vm\_init()函数中specs数组中的元素，业务逻辑比较简单，代码如下：

|  |
| --- |
| // 生成处理器偏移量  function gen\_specs($f, $prolog, $specs) {  $lastdef = array\_pop($specs);  $last = 0;  foreach ($specs as $num => $def) { // 遍历规则列表  while (++$last < $num) {  out($f, "$prolog$lastdef,\n"); // 中间不连续的全部用最后一个填充  }  $last = $num;  out($f, "$prolog$def,\n");  }  while ($last++ < 255) { // 不足256个补充到256个，用最后一个填充  out($f, "$prolog$lastdef,\n");  }  } |

### 小结

zend\_vm\_init()函数用于初始化虚拟机。经过以上操作以后，生成的zend\_vm\_init()函数框架如下：

|  |
| --- |
| void zend\_vm\_init(void){  // 这里都是通过gen\_labels()函数创建的处理器名，3451行，很多重复  static const void \* const labels[] = {  ZEND\_NOP\_SPEC\_HANDLER,  ...  ZEND\_NULL\_HANDLER  };    // 这里是通过gen\_specs()函数创建的所有操作码的对应配置，257 行  static const uint32\_t specs[] = {  0, // 第一个数字表示此操作码的第一个处理器的编号  1 | SPEC\_RULE\_OP1 | SPEC\_RULE\_OP2,  ...  3450,  };  zend\_opcode\_handlers = labels; // 操作码处理器数据  zend\_handlers\_count = sizeof(labels) / sizeof(void\*); // 处理器数量  zend\_spec\_handlers = specs;  VM\_TRACE\_START();  } |

如上所示，labels变量和zend\_opcode\_handlers变量中存放的是所有操作码处理器，用于在zend\_vm\_set\_opcode\_handler()函数中为操作码关联处理器。

specs变量和zend\_spec\_handlers变量中存放每个操作码的第一个处理器的序号，以及此操作码的相关配置。例如，specs中的第二个元素“1 | SPEC\_RULE\_OP1 | SPEC\_RULE\_OP2”表示1号操作码（ZEND\_ADD）的两个操作对象都有类型规则。序号使用32位整数的右侧16位，附加标记使用左侧16位，在zend\_vm\_get\_opcode\_handler\_idx()函数中会通过掩码把它们分开。

在zend\_vm\_set\_opcode\_handler()函数中，会使用zend\_spec\_handlers变量配合调用zend\_vm\_get\_opcode\_handler\_idx()函数，计算处理器序号。详情参见“生成zend\_vm\_execute.h文件”章节。

## 六）生成其他函数

在zend\_vm\_execute.skl文件中，zend\_vm\_init()函数后面，还有以下函数和变量，内容中没有标记，全都直接输出到zend\_vm\_execute.h中：

|  |
| --- |
| static HashTable \*zend\_handlers\_table = NULL;  void zend\_vm\_dtor(void){  // 省略业务逻辑...  }  static void init\_opcode\_serialiser(void){  // 省略业务逻辑...  }  ZEND\_API void ZEND\_FASTCALL zend\_serialize\_opcode\_handler(zend\_op \*op){  // 省略业务逻辑...  }  ZEND\_API void ZEND\_FASTCALL zend\_deserialize\_opcode\_handler(zend\_op \*op){  // 省略业务逻辑...  }  ZEND\_API const void\* ZEND\_FASTCALL zend\_get\_opcode\_handler\_func(const zend\_op \*op){  // 省略业务逻辑...  }  ZEND\_API const zend\_op \*zend\_get\_halt\_op(void){  // 省略业务逻辑...  }  ZEND\_API int zend\_vm\_kind(void){  return ZEND\_VM\_KIND;  } |

以上函数的生成过程中无复杂业务逻辑。

# 六、gen\_executor\_code()函数

本章重点介绍gen\_executor\_code()函数，它用于为操作码的普通处理器、专用处理器和助手生成C语言程序代码。

gen\_executor\_code()函数在上一章节中，DEFINES标记的解析过程中调用，调用代码如下：

|  |
| --- |
| // 生成执行器代码，参数：zend\_vm\_execute.h文件句柄, 1, 1, 标签前面的内容  gen\_executor\_code($f, $spec, ZEND\_VM\_KIND\_CALL, $m[1]); |

如上所示，传入的4个参数：

1）zend\_vm\_execute.h文件句柄；

2）数值1；

3）ZEND\_VM\_KIND\_CALL，整数值为1；

4）DEFINES标记前面的内容，标记在文件开头，所以这个参数实际为空。

gen\_executor\_code()函数还INTERNAL\_EXECUTOR标记的解析过程中调用，调用代码如下：

|  |
| --- |
| if ($kind != ZEND\_VM\_KIND\_CALL) {  ...  gen\_executor\_code($f, $spec, $kind, $m[1], $switch\_labels);  } |

这次调用时$kind的值为ZEND\_VM\_KIND\_HYBRID。

gen\_executor\_code()函数前半部分代码用于生成处理器和助手，代码如下：

|  |
| --- |
| **// gen\_executor\_code()函数，第1块代码**  // 生成操作操作码处理器（专用或非专用）和助手  // 参数：zend\_vm\_execute.h文件句柄, 1, ZEND\_VM\_KIND, 标签前面的内容 , 返回值  function gen\_executor\_code($f, $spec, $kind, $prolog, &$switch\_labels = array()) {  global $list, $opcodes, $helpers, $op\_types\_ex, $gen\_order;  $op1t = $op\_types\_ex; // 操作对象的类型表  foreach ($op1t as $op1) { // 遍历所有类型op1可能的类型  $op2t = $op\_types\_ex; // 操作对象的类型表  foreach ($op2t as $op2) { // 遍历所有类型op2可能的类型  // 用原本顺序遍历 助手 和 处理器（$list 里是处理器和助手 ）  foreach ($list as $lineno => $dsc) {  if (isset($dsc["handler"])) { // 如果类型是处理器  **// 检查并生成处理器，参见下文中第2块代码 ...**  } else if (isset($dsc["helper"])) { // 类型是助手  **// 检查并生成助手，参见下文中第3块代码 ...**  }  }  }  }  **// 后半部分，参见下文中第4块代码 ...** |

如上所示，先遍历两个操作对象的类型的所有可能组合，对每个组合：如果有匹配的处理器或助手，就生成它的C语言代码。

$op\_types\_ex全局变量中声名了8种操作对象的可用类型，内容如下：

|  |
| --- |
| $op\_types\_ex = array(  "ANY", "CONST", "TMPVARCV", "TMPVAR", "TMP", "VAR", "UNUSED", "CV",  ); |

## 一）生成处理器

生成处理器，代码如下：

|  |
| --- |
| **// gen\_executor\_code()函数，第2块代码**  $num = $dsc["handler"]; // 操作码编号  // 一个 【处理器】 有多少个分支是由 操作对象类型规则 和 专用规则 共同决定的  foreach (**extra\_spec\_handler**($opcodes[$num]) as $extra\_spec) {  // 如果处理器支持这种类型组合  if (isset($opcodes[$num]["op1"][$op1]) && isset($opcodes[$num]["op2"][$op2])) {  // 生成处理器代码  // zend\_vm\_execute.h文件句柄, 1, ZEND\_VM\_KIND, 操作码名，第一个对象的类型，第二个操作对象的类型，是否有use，源码，行号，处理器，额外规则，$switch\_labels 引用返回  **gen\_handler**($f, 1, $kind, $opcodes[$num]["op"], $op1, $op2, isset($opcodes[$num]["use"]), $opcodes[$num]["code"], $lineno, $opcodes[$num], $extra\_spec, $switch\_labels);  }  } |

要理解这块业务逻辑先要理解extra\_spec\_handler()函数，extra\_spec\_handler()函数用于把处理器中的spec规则处理成集合。再遍历集合：为每个匹配的类型对创建处理器函数。所以一个处理器生成多少个函数分支，是由专用规则（spec）、op1的类型规则，op2的类型规则三者共同决定的。

### extra\_spec\_handler()函数

spec在普通处理器和专用处理器中共有15种组合，详情参见“数据解析-解析处理器名”和“数据解析-解析专用处理器名”章节。

extra\_spec\_handler()函数会把上面的组合处理成迪卡尔集，例如：

|  |
| --- |
| 组合： {"RETVAL":[0,1],"OBSERVER":[0,1]}  处理结果：  [  {"RETVAL":0,"OBSERVER":0},  {"RETVAL":1,"OBSERVER":0},  {"RETVAL":0,"OBSERVER":1},  {"RETVAL":1,"OBSERVER":1}  ] |

比较特殊的是带有OP\_DATA键的规则，例如：

|  |
| --- |
| 组合： {"OP\_DATA":{"CONST":0,"TMP":1,"VAR":2,"CV":3}}  处理结果：  [  {"OP\_DATA":"CONST"},  {"OP\_DATA":"TMP"},  {"OP\_DATA":"VAR"},  {"OP\_DATA":"CV"}  ] |

extra\_spec\_handler()函数代码如下：

|  |
| --- |
| function extra\_spec\_handler($dsc) {  global $op\_types\_ex;  if (!isset($dsc["spec"])) { // 如果没有特殊标记，返回空数组  return array(array());  }  $specs = $dsc["spec"]; // 它原本是由 parse\_spec\_rules 这个方法生成的有键名的数组  if (isset($specs["OP\_DATA"])) { // 单独处理 OP\_DATA，把它转成索引数组  $op\_data\_specs = $specs["OP\_DATA"];  $specs["OP\_DATA"] = array(); // 重置这个列表，下面把它转成索引数组  foreach ($op\_types\_ex as $op\_data) { // 遍历所有类型，类型在下面要作为 $mode 使用  if (isset($dsc["spec"]["OP\_DATA"][$op\_data])) { // 如果原列表中有这个类型  $specs["OP\_DATA"][] = $op\_data; // 把这个类型添加到新列表中  }  }  }    $f = function($specs) use (&$f) { // 递归处理  $spec = key($specs); // 取出第一个键名  $top = array\_shift($specs); // 弹出第一个键值  if ($specs) { // 如果还有其他规则  $next = $f($specs); // 递归操作下一个  } else { // 如果为空，  $next = array(array()); // $next 最初是两层空数组  }  $ret = array(); //  // 遍历 。这里是递归的业务逻辑，前面的全都卡住，从最后一个倒着来  foreach ($next as $existing) {  foreach ($top as $mode) { // 每一组里面，在最前面添加 array($spec => $mode)  $ret[] = array($spec => $mode) + $existing;  }  }  return $ret; // 返回副本集  };  // 开始执行获取每个 spec对应每个mode 的列表 例如： [ [RETVAL=>0],[RETVAL=>1] ]  return $f($specs); // 返回的是个纯索引数组  } |

如上所示，业务逻辑分为两部分：

1）如果有$specs["OP\_DATA"]，把它处理成索引数组；

2）采用递归遍历，每次处理$specs中的一个键，遍历它的每一个值：为已有集合创建副本，把这个值添加到集合副本中；再用所有副本组合成新集合。这样就构成了迪卡尔集。

### gen\_handler()函数

gen\_handler()函数用于生成处理器C语言代码，注意这里传入的是ZEND\_VM\_KIND\_CALL模式，函数声名如下：

|  |
| --- |
| **// gen\_handler()函数，第1块代码**  // 参数说明：zend\_vm\_execute.h文件句柄, 1, ZEND\_VM\_KIND\_CALL, 操作码名，第一个，第二个操作对象，是否被其他处理器引用，源码，行号，整个处理器，专用规则，$switch\_labels 引用返回  function gen\_handler($f, $spec, $kind, $name, $op1, $op2, $use, $code, $lineno, $opcode, $extra\_spec = null, &$switch\_labels = array()) {  global $definition\_file, $prefix, $opnames, $gen\_order;  static $used\_observer\_handlers = array(); |

函数的业务逻辑比较复杂，主要分为以下步骤。

#### **第一步，跳过别名**

代码如下：

|  |
| --- |
| **// gen\_handler()函数，第2块代码**  // 如果处理器是其他处理器的别名，并且 （$spec为1，所以可以忽略第二个条件）  if (isset($opcode['alias']) && ($spec || $kind != ZEND\_VM\_KIND\_SWITCH)) {  return; // 跳过此处理器  } |

如上所示，如果处理器是其他处理器的别名，直接返回。

#### **第二步，跳过不满足专用规则的处理器**

代码如下：

|  |
| --- |
| **// gen\_handler()函数，第3块代码**  // 如果 $spec==1 并且 此处理器不合规则，跳过此处理器  if ($spec && skip\_extra\_spec\_function($op1, $op2, $extra\_spec)) {  return;  } |

##### skip\_extra\_spec\_function()函数

skip\_extra\_spec\_function()函数用来检测，此处理器的对象类型和专用规则是否存在冲突，如果存在冲突，则跳过这个处理器，不生成它。代码如下：

|  |
| --- |
| // 参数表：第一，第二个操作对象，专用规则  function skip\_extra\_spec\_function($op1, $op2, $extra\_spec) {  global $commutative\_order;  // 情况1：如果规则不允许两个常量，但两个操作对象都是常量  if (isset($extra\_spec["NO\_CONST\_CONST"]) &&  $op1 == "CONST" && $op2 == "CONST") {  return true; // 跳过这个处理器  }  // 情况2：如果参数可交换，并且 第一个操作对象的交换顺序号小于第二个  if (isset($extra\_spec["COMMUTATIVE"]) &&  $commutative\_order[$op1] < $commutative\_order[$op2]) {  return true; // 跳过这个处理器  }  return false; // 其他情况不跳过  } |

如上所示，碰到以下两种情况需可以跳过此处理器：

1）禁用双常量（NO\_CONST\_CONST）：专用规则要求，此处理器的两个操作对象不可以都是常量，但此时两个操作对象的类型恰巧都是常量；

2）可交换时去重复：如果此处理器的专用规则中有，操作对象可交换（COMMUTATIVE），那么两个操作对象孰前孰后就不重要了，只生成一个处理器函数即可。为了防止生成多余的处理器函数，当第一个操作对象的类型的交换序号（commutative\_order）小于第二个时，跳过生成此处理器。

##### $commutative\_order全局变量

$commutative\_order全局变量是操作对象类型的交换顺序号，两个操作对象可交换时，顺序号大的放在前面，顺序号小的放在后面，顺序号定义如下：

|  |
| --- |
| // 操作对象类型的交换顺序，序号大的在前  $commutative\_order = array(  "ANY" => 0,  "TMP" => 1,  "VAR" => 2,  "CONST" => 0,  "UNUSED" => 0,  "CV" => 4,  "TMPVAR" => 2,  "TMPVARCV" => 4,  ); |

#### **第三步，跳过无效的智能分支**

代码如下：

|  |
| --- |
| **// gen\_handler()函数，第4块代码**  // 处理器为 "cold"类型，两个都是常量的情况，跳过 SMART\_BRANCH 规则  // 如果额外规则包含智能分支  if (isset($extra\_spec["SMART\_BRANCH"])) {  // 如果 hot 类型是 HOT\_NOCONSTCONST\_（hot不允许两个常量） 或 COLD\_CONSTCONST\_（cold允许两个常量）  if ($opcode["hot"] === 'HOT\_NOCONSTCONST\_'  || $opcode["hot"] === 'COLD\_CONSTCONST\_') {  if (($op1 === 'CONST') && ($op2 === 'CONST')) { // 如果两个操作对象都是常量  if ($extra\_spec["SMART\_BRANCH"] == 0) { // 如果没有智能分支  unset($extra\_spec["SMART\_BRANCH"]); // 删除智能分支键名  } else { // 如果有智能分支，跳过  return;  }  }  }  } |

如上所示，如果同时满足以下3个条件，则跳过此处理器：

1）此处理器有有效的智能分支（SMART\_BRANCH）；

2）处理器的hot标记值为HOT\_NOCONSTCONST\_或COLD\_CONSTCONST\_；

3）两个操作对象的类型都是常量（CONST）。

#### **第四步，对于命名参数，跳过QUICK\_ARG规则**

代码如下：

|  |
| --- |
| **// gen\_handler()函数，第5块代码**  // 参数有名称时跳过开启了QUICK\_ARG规则的处理器  if (isset($extra\_spec["QUICK\_ARG"])) { // 如果专用规则包含 QUICK\_ARG  if ($op2 === "CONST") { // 第二个操作对象是常量  if ($extra\_spec["QUICK\_ARG"] == 0) { // 如果没有启用QUICK\_ARG，删除它  unset($extra\_spec["QUICK\_ARG"]);  } else { // QUICK\_ARG 有效，直接返回  return;  }  }  } |

如上所示，如果此处理器添加了快捷参数（QUICK\_ARG）规则，并且op2是常量（表示在操作已命名参数）：

1）如果QUICK\_ARG状态为关闭，删除QUICK\_ARG规则；

2）否则跳过生成此处理器。

#### **第五步，跳过重复的观察者处理器**

代码如下：

|  |
| --- |
| **// gen\_handler()函数，第6块代码**  // 为 OBSERVER 处理器跳过所有规则  /\* Skip all specialization for OBSERVER handlers \*/  // 如果额外规则包含 OBSERVER 并且值为 1  if (isset($extra\_spec["OBSERVER"]) && $extra\_spec["OBSERVER"] == 1) {  if (isset($extra\_spec["RETVAL"])) { // 如果有 RETVAL 规则  if ($extra\_spec["RETVAL"] == 0) { // 值为否，删除它  unset($extra\_spec["RETVAL"]);  } else { // 直接返回  return;  }  }  if ($op1 != "ANY" || $op2 != "ANY") { // 如果其中一个操作对象类型不是 ANY  if (!isset($used\_observer\_handlers[$kind][$opcode["op"]])) {  $used\_observer\_handlers[$kind][$opcode["op"]] = true; // 添加到观察者列表  $op1 = "ANY"; // 两个操作对象类型都换成ANY  $op2 = "ANY";  } else { // 直接返回  return;  }  }  } |

如上所示，如果处理器专用规则中有有观察者（OBSERVER），分为两步检验：

1）如果有RETVAL，返回；

2）如果其中一个操作对象类型不是 ANY：

a）如果此处理器不在观察者处理器列表中，把它放入列表，并把两处参数类型都换成ANY；

b）如果此处理器已经在观察者列表中，跳过这个处理器。

#### **第六步，生成C语言函数**声名语句

代码如下：

|  |
| --- |
| **// gen\_handler()函数，第7块代码**  $additional\_func = false; // 默认不添加附加方法  // 操作码名 + \_SPEC + 两个操作对象类型的前缀 + extra\_spec\_name()函数生成后缀  $spec\_name = $name.($spec?"\_SPEC":"").$prefix[$op1].$prefix[$op2].  ($spec?extra\_spec\_name($extra\_spec):"");  switch ($kind) { // 按生成模式处理  case ZEND\_VM\_KIND\_HYBRID: // DEFINES 标记中不用这种模式  // ....  case ZEND\_VM\_KIND\_CALL: // DEFINES 标记中用到这种模式  // 如果用 ZEND\_VM\_KIND\_HYBRID 模式生成  if ($opcode["hot"] && ZEND\_VM\_KIND == ZEND\_VM\_KIND\_HYBRID  && is\_hot\_handler($opcode["hot"], $op1, $op2, $extra\_spec)) {  if (isset($opcode["use"])) {  out($f,"static zend\_always\_inline ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL {$spec\_name}\_INLINE\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS)\n");  $additional\_func = true;  } else {  out($f,"static ZEND\_VM\_HOT ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL {$spec\_name}\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS)\n");  }  } else if ($opcode["hot"] && is\_cold\_handler($opcode["hot"], $op1, $op2, $extra\_spec)) {  out($f,"static ZEND\_VM\_COLD ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL {$spec\_name}\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS)\n");  } else {  out($f,"static ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL {$spec\_name}\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS)\n");  }  break;  // 其他运行模式 ...  } |

如上所示，基本的函数名称的生成规则是：

操作码名 + \_SPEC + 两个操作对象类型的前缀 + extra\_spec\_name()函数生成后缀。

生成的C语言函数名分为如下几种情况：

1）如果通过is\_hot\_handler()函数验证，此处理器为热处理器：

a）如果此处理器被其他处理器引用，声名语句如下：

|  |
| --- |
| static zend\_always\_inline ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL {$spec\_name}\_INLINE\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS) |

在以上这种情况中，生成的\_INLINE\_HANDLER()处理器不能被直接调用，所以还添加了一个$additional\_func标记，在在gen\_handler()函数的最后，如果检测到这个标记，会再生成一个附加函数来调用这里声名的\_INLINE\_HANDLER()处理器。

b）如果此处理器没有被其他处理器引用，声名语句如下：

|  |
| --- |
| static ZEND\_VM\_HOT ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL {$spec\_name}\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS) |

2）如果通过is\_cold\_handler()函数验证，是冷处理器，声名语句如下：

|  |
| --- |
| static ZEND\_VM\_COLD ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL {$spec\_name}\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS) |

3）其他情况，不是冷处理器，也不是热处理器，声名语句如下：

|  |
| --- |
| static ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL {$spec\_name}\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS) |

如上所示，所有处理器名都是以“\_HANDLER”结尾，以上几种情况的区别可简单归纳如下：

1）被其他处理器引用的热处理器，带“\_INLINE\_”标记，一共2个，例如：

|  |
| --- |
| sstatic zend\_always\_inline ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL ZEND\_FETCH\_OBJ\_R\_SPEC\_UNUSED\_CONST**\_INLINE\_**HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS) |

2）普通热处理器，带“ZEND\_VM\_HOT”标记，一共191个，例如：

|  |
| --- |
| static **ZEND\_VM\_HOT** ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL ZEND\_INIT\_METHOD\_CALL\_SPEC\_UNUSED\_CONST\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS) |

3）冷处理器带“ZEND\_VM\_COLD”标记，一共77个，例如：

|  |
| --- |
| static **ZEND\_VM\_COLD** ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL ZEND\_DO\_ICALL\_SPEC\_OBSERVER\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS) |

4）其他情况，不带以上3个标记，一共732个，例如：

|  |
| --- |
| **static ZEND\_OPCODE\_HANDLER\_RET** ZEND\_FASTCALL ZEND\_PRE\_INC\_STATIC\_PROP\_SPEC\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS) |

##### extra\_spec\_name()函数

extra\_spec\_name()函数用于为函数名生成专用规则后缀，代码如下：

|  |
| --- |
| function extra\_spec\_name($extra\_spec) {  global $prefix;  $s = "";  if (isset($extra\_spec["OP\_DATA"])) { // 如果规则包含 OP\_DATA 类型  $s .= "\_OP\_DATA" . $prefix[$extra\_spec["OP\_DATA"]]; // 添加操作数据类型前缀  }  if (isset($extra\_spec["RETVAL"])) { // 如果规则包含 RETVAL  // 添加 "\_RETVAL\_" . USED 或 UNUSED  $s .= "\_RETVAL\_".($extra\_spec["RETVAL"] ? "USED" : "UNUSED");  }  if (isset($extra\_spec["QUICK\_ARG"])) { // 如果规则包含 QUICK\_ARG  if ($extra\_spec["QUICK\_ARG"]) { // 并且启用 QUICK\_ARG  $s .= "\_QUICK"; // 添加  }  }  if (isset($extra\_spec["SMART\_BRANCH"])) { // 如果规则包含 SMART\_BRANCH  if ($extra\_spec["SMART\_BRANCH"] == 1) { // 按规则添加 \_JMPZ 或 \_JMPNZ  $s .= "\_JMPZ";  } else if ($extra\_spec["SMART\_BRANCH"] == 2) {  $s .= "\_JMPNZ";  }  }  if (isset($extra\_spec["ISSET"])) { // 如果规则包含 ISSET  if ($extra\_spec["ISSET"] == 0) { // 未启用 添加 \_SET，启用 添加 \_EMPTY  $s .= "\_SET";  } else {  $s .= "\_EMPTY";  }  }  if (isset($extra\_spec["OBSERVER"])) { // 如果规则包含 OBSERVER  if ($extra\_spec["OBSERVER"]) { // 如果规则启用 添加后缀 \_OBSERVER  $s .= "\_OBSERVER";  }  }  return $s;  } |

如上所示，根据扩展规则OP\_DATA、QUICK\_ARG、SMART\_BRANCH、SMART\_BRANCH、ISSET、OBSERVER的实际情况来添加后缀。

##### $prefix全局变量

$prefix全局变量存放各类型对应代码中的前缀，代码如下：

|  |
| --- |
| // 类型前缀  $prefix = array(  "ANY" => "",  "TMP" => "\_TMP",  "VAR" => "\_VAR",  "CONST" => "\_CONST",  "UNUSED" => "\_UNUSED",  "CV" => "\_CV",  "TMPVAR" => "\_TMPVAR",  "TMPVARCV" => "\_TMPVARCV",  ); |

##### is\_hot\_handler()函数

is\_hot\_handler()函数用来检验一个处理器是否是热处理器，代码如下：

|  |
| --- |
| // 是否是热助手 ：gen\_handler()调用  function is\_hot\_handler($hot, $op1, $op2, $extra\_spec) {  // 如果关闭智能分支，返回false  if (isset($extra\_spec["SMART\_BRANCH"]) && $extra\_spec["SMART\_BRANCH"] == 0) {  return false;  }  // 如果开启了 观察者模式 返回false  if (isset($extra\_spec["OBSERVER"]) && $extra\_spec["OBSERVER"] == 1) {  return false;  }  if ($hot === 'HOT\_' || $hot === 'INLINE\_') { // 情况1: HOT\_ 或 INLINE\_ 前缀，算hot  return true;  } else if ($hot === 'HOT\_NOCONST\_') { // 情况2: 不允许常量  return ($op1 !== 'CONST'); // 并且 op1 不是常量  } else if ($hot === 'HOT\_NOCONSTCONST\_') { // 情况3: 不允许两个常量  return (($op1 !== 'CONST') || ($op2 !== 'CONST')) ; // 并且有一个不是常量  } else if ($hot === 'HOT\_OBJ\_') { // 情况4: 要求是对象  // op1 是 UNUSED 或 op1 是 CV 并且 op2 是常量  return (($op1 === 'UNUSED') || ($op1 === 'CV')) && ($op2 === 'CONST');  } else if ($hot === 'HOT\_SEND\_') { // 情况5: 如果是 HOT\_SEND\_  return !empty($extra\_spec["QUICK\_ARG"]); // 扩展规则里必须包含 QUICK\_ARG  } else {  return false;  }  } |

##### is\_cold\_handler()函数

is\_cold\_handler()函数用来检验一个处理器是否是冷处理器，代码如下：

|  |
| --- |
| // 是否是冷助手：gen\_handler()调用  function is\_cold\_handler($hot, $op1, $op2, $extra\_spec) {  if ($hot === 'COLD\_') { // 标识是 COLD\_，返回 true  return true;  // 如果有附加规则 OBSERVER，并且启用 ，返回 true  } else if (isset($extra\_spec["OBSERVER"]) && $extra\_spec["OBSERVER"] == 1) {  return true;  } else if ($hot === 'COLD\_CONST\_') { // 冷、常量  return ($op1 === 'CONST'); // 必须第一个是常量  } else if ($hot === 'COLD\_CONSTCONST\_') { // 冷、两个操作对象都是常量  return ($op1 === 'CONST' && $op2 === 'CONST'); // 必须两个都是常量  } else if ($hot === 'HOT\_OBJ\_') { // 热对象  return ($op1 === 'CONST'); // 必须第一个是常量  } else if ($hot === 'HOT\_NOCONST\_') { // 热非常量  return ($op1 === 'CONST'); // 必须第一个是常量  } else if ($hot === 'HOT\_NOCONSTCONST\_') { // 热，不允许两个常量  return ($op1 === 'CONST' && $op2 === 'CONST'); // 必须两个都是常量  } else { // 其他都false  return false;  }  } |

#### **第七步，生成C语言函数业务逻辑源码**

代码如下：

|  |
| --- |
| **// gen\_handler()函数，第8块代码**  // 参数表：zend\_vm\_execute.h文件句柄, 1, ZEND\_VM\_KIND, 操作码名，第一第二个操作对象，函数名，额外规则  gen\_code($f, $spec, $kind, $code, $op1, $op2, $name, $extra\_spec); |

gen\_code()函数比较复杂，将在下一章中详细介绍。

#### **第八步，生成附加函数**

在前文中第6步中，如果此处理器是被其他处理器引用的热处理器，需要生成附加函数，代码如下：

|  |
| --- |
| **// gen\_handler()函数，第9块代码**  if ($additional\_func) { // 生成附加函数  out($f,"static ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL {$spec\_name}\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS)\n");  out($f,"{\n");  out($f,"\tZEND\_VM\_TAIL\_CALL({$spec\_name}\_INLINE\_HANDLER(  ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU));\n");  out($f,"}\n");  out($f,"\n");  }  # define ZEND\_VM\_TAIL\_CALL(call) return call |

如上所示这个函数，用于供外部调用，由它来引用{$spec\_name}\_INLINE\_HANDLER()函数，完成整个调用。ZEND\_VM\_TAIL\_CALL()宏程序在解析DEFINES()标签时已经定义好了，详情可参见“gen\_executor()函数-DEFINES标签”章节。

## 二）生成助手

生成助手，代码如下：

|  |
| --- |
| **// gen\_executor\_code()函数，第3块代码**  $num = $dsc["helper"];  // 一个 【助手】 有多少个分支是由 操作对象类型规则 和 专用规则 共同决定的  foreach (**extra\_spec\_handler**($helpers[$num]) as $extra\_spec) {  // 如果助手支持这种类型组合  if (isset($helpers[$num]["op1"][$op1]) && isset($helpers[$num]["op2"][$op2])) {  // 参数：zend\_vm\_execute.h文件句柄, 1, ZEND\_VM\_KIND, 操作码名  **gen\_helper**($f, 1, $kind, $num, $op1, $op2, $helpers[$num]["param"], $helpers[$num]["code"], $lineno, $helpers[$num]["inline"], $helpers[$num]["cold"], $helpers[$num]["hot"], $extra\_spec);  }  } |

如上所示，只要op1和op2的类型匹配，就调用gen\_helper()函数生成助手代码。

### gen\_helper()函数

gen\_helper()函数用于生成助手的C语言代码，注意这里传入的是ZEND\_VM\_KIND\_CALL模式，函数声名如下：

|  |
| --- |
| **// gen\_helper()函数，第1块代码**  // 生成助手C语言函数：gen\_executor\_code()函数 用到  function gen\_helper($f, $spec, $kind, $name, $op1, $op2, $param, $code, $lineno, $inline, $cold = false, $hot = false, $extra\_spec = null) {  global $definition\_file, $prefix; |

#### **第一步，HYBRID模式下跳过非热助手**

代码如下：

|  |
| --- |
| **// gen\_helper()函数，第2块代码**  // 如果处理器是其他处理器的别名，并且 （$spec为1，所以可以忽略第二个条件）  if (isset($opcode['alias']) && ($spec || $kind != ZEND\_VM\_KIND\_SWITCH)) {  return; // 跳过此处理器  } |

如上所示，如果处理器是其他处理器的别名，直接返回。

#### **第二步，跳过不满足专用规则的助手**

代码如下：

|  |
| --- |
| **// gen\_helper()函数，第3块代码**  // 如果 $spec==1 并且 此处理器不合规则，跳过此处理器  if ($spec && skip\_extra\_spec\_function($op1, $op2, $extra\_spec)) {  return;  } |

#### **第三步，生成助手函数名**

代码如下：

|  |
| --- |
| **// gen\_helper()函数，第4块代码**  // 按额外规则 ，生成后缀  $spec\_name = $name.($spec?"\_SPEC":"").$prefix[$op1].$prefix[$op2].  ($spec?extra\_spec\_name($extra\_spec):""); |

以上两步业务逻辑与gen\_handler()中相似。

#### **第四步，按不同模式操作**

##### ZEND\_VM\_KIND\_HYBRID模式

代码如下：

|  |
| --- |
| **// gen\_helper()函数，第5块代码**  switch ($kind) { // 按照选择的运行模式，生成助手实体  case ZEND\_VM\_KIND\_HYBRID: // HYBRID 模式  out($f, $spec\_name . "\_LABEL:\n"); // 直接输出 规则名字\_LABEL:  break; |

以上业务逻辑只有一次生效，是在解析INTERNAL\_EXECUTOR标签时，生成一个名为zend\_leave\_helper\_SPEC\_LABEL的goto标签。INTERNAL\_EXECUTOR标签中的内容在windows操作系统中没有用到。

##### ZEND\_VM\_KIND\_CALL模式

代码如下：

|  |
| --- |
| **// gen\_helper()函数，第6块代码**  case ZEND\_VM\_KIND\_CALL: // CALL 模式，用到  if ($inline) { // 有 $inline 标记  $zend\_attributes = " zend\_always\_inline";  $zend\_fastcall = "";  } else { // 没有 $inline 标记  if ($cold) { // 如果是冷的，多一个 ZEND\_COLD 标记  $zend\_attributes = " zend\_never\_inline ZEND\_COLD";  } else { 不是冷的  $zend\_attributes = " zend\_never\_inline";  }  //  $zend\_fastcall = " ZEND\_FASTCALL";  }  if ($param == null) { // 参数为空，生成没有参数的 助手  out($f, "static$zend\_attributes ZEND\_OPCODE\_HANDLER\_RET$zend\_fastcall $spec\_name(ZEND\_OPCODE\_HANDLER\_ARGS)\n");  } else { // 参数不为空，生成有参数的助手  out($f, "static$zend\_attributes ZEND\_OPCODE\_HANDLER\_RET$zend\_fastcall $spec\_name($param ZEND\_OPCODE\_HANDLER\_ARGS\_DC)\n");  }  break;  // 其他运行模式 ...  } |

如上所示，业务逻辑可归纳为以下两步：

一）有$inline标记时，函数名添加zend\_always\_inline修饰符；没有$inline标记时，添加zend\_never\_inline、ZEND\_FASTCALL两个修饰符，如果是冷处理器，还需要添加ZEND\_COLD修饰符；

二）把修饰符和函数名写入到zend\_vm\_execute.h中，有形式参数时添加形式参数。

在生成过程中，大助手函数都是通过这块业务逻辑生成的。

#### **第五步，生成C语言函数业务逻辑源码**

代码如下：

|  |
| --- |
| **// gen\_helper()函数，第7块代码**  // 参数表：zend\_vm\_execute.h文件句柄, 1, ZEND\_VM\_KIND, 操作码名，第一第二个操作对象，函数名，额外规则  gen\_code($f, $spec, $kind, $code, $op1, $op2, $name, $extra\_spec); |

## 三）后续处理

最后要生成一个null处理器。在生虚拟机成处理器和助手代码时，$kind参数的值为ZEND\_VM\_KIND\_CALL模式。此模式的处理逻辑如下：

|  |
| --- |
| **// gen\_executor\_code()函数，第4块代码**  // 为未定义操作码生成处理器  switch ($kind) { // 根据不同模式进行处理  case ZEND\_VM\_KIND\_CALL: // 重点  gen\_null\_handler($f);  break;  // 其他运行模式 ...  }  } |

### gen\_null\_handler()函数

gen\_null\_handler()函数用于为未定义的操作码生成一个处理器，代码如下：

|  |
| --- |
| // 生成未定义操作码的处理器（CALL 线程模式）  function gen\_null\_handler($f) {  static $done = 0;  if (!$done) { // 只生成1次  $done = 1;  out($f,"static ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL ZEND\_NULL\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS)\n");  // 输出内容 ...  }  } |

生成的C语言函数如下：

|  |
| --- |
| static ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL  ZEND\_NULL\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS){  USE\_OPLINE  SAVE\_OPLINE();  zend\_error\_noreturn(E\_ERROR, "Invalid opcode %d/%d/%d.", OPLINE->opcode, OPLINE->op1\_type, OPLINE->op2\_type);  ZEND\_VM\_NEXT\_OPCODE(); /\* Never reached \*/  } |

如上所示，函数的业务逻辑是报错：“无效的操作码”，并附加两个操作对象的类型。

## 四）函数名修饰符

在以上的生成过程中，常常见到函数名前添加ZEND\_FASTCALL、ZEND\_COLD等修饰符，这些大多与兼容性有关，在zend\_portability.h中定义，现将常量用的修饰符整理如下：

|  |
| --- |
| #if defined(\_\_GNUC\_\_) && ZEND\_GCC\_VERSION >= 3004 && defined(\_\_i386\_\_)  ...  #elif defined(\_MSC\_VER) && \_MSC\_VER >= 1800 && !defined(\_\_clang\_\_)  # define ZEND\_FASTCALL \_\_vectorcall /\* windows10 中这行定义生效 \*/  #endif  #if defined(\_\_GNUC\_\_) && ZEND\_GCC\_VERSION >= 4003  ...  #else /\* windows10 中以下定义生效，ZEND\_HOT、ZEND\_COLD都是空 \*/  # define ZEND\_COLD  # define ZEND\_HOT  # define ZEND\_OPT\_SIZE  # define ZEND\_OPT\_SPEED  #endif  #if ZEND\_DEBUG || defined(ZEND\_WIN32\_NEVER\_INLINE)  ...  #else  # if defined(\_\_GNUC\_\_)  ...  # elif defined(\_MSC\_VER) /\* windows10 中以下定义生效 \*/  # define zend\_always\_inline \_\_forceinline  # define zend\_never\_inline \_\_declspec(noinline)  # else  ...  # endif  #endif /\* ZEND\_DEBUG \*/ |

还有几个常量在zend\_vm\_execute.h中定义如下：

|  |
| --- |
| #if defined(ZEND\_VM\_FP\_GLOBAL\_REG) && defined(ZEND\_VM\_IP\_GLOBAL\_REG)  ...  #else /\* window走这里 \*/  # define ZEND\_OPCODE\_HANDLER\_RET int  # define ZEND\_VM\_TAIL\_CALL(call) return call  # define ZEND\_VM\_CONTINUE() return 0  # define ZEND\_VM\_RETURN() return -1  # define ZEND\_VM\_HOT  # define ZEND\_VM\_COLD ZEND\_COLD ZEND\_OPT\_SIZE  #endif |

以上常量在WINDOWS 10中取值如下：

|  |  |
| --- | --- |
| **常量名** | **值** |
| ZEND\_FASTCALL | \_\_vectorcall |
| ZEND\_COLD | 空 |
| ZEND\_HOT | 空 |
| ZEND\_OPT\_SIZE | 空 |
| ZEND\_OPT\_SPEED | 空 |
| zend\_always\_inline | \_\_forceinline |
| zend\_never\_inline | \_\_declspec(noinline) |
| END\_OPCODE\_HANDLER\_RET | int |
| ZEND\_VM\_HOT | 空 |
| ZEND\_VM\_COLD | ZEND\_COLD ZEND\_OPT\_SIZE |

# 七、gen\_code()函数

gen\_code()函数用于生成处理器和助手的C语言函数代码。

zend\_vm\_def.h文件并非是可运行的C语言代码，里面包含很多伪码，主要是伪常量（例如OP1\_TYPE）和伪函数（例如GET\_OP1\_ZVAL\_PTR），这些伪常量和伪函数都没有被声名过，它们会被gen\_code()函数替换成可用的常量名和函数名。

gen\_code()函数声名如下：

|  |
| --- |
| **// gen\_code()函数，第1块代码**  // 参数表：zend\_vm\_execute.h文件句柄, 1, ZEND\_VM\_KIND, 操作码名，第一第二个操作对象，操作码名，额外规则  function gen\_code($f, $spec, $kind, $code, $op1, $op2, $name, $extra\_spec=null) {  global $op1\_type, $op2\_type, ...; // 引用外部全局变量 |

函数业务逻辑的前半部分是一组替换操作，业务逻辑框架如下：

|  |
| --- |
| **// gen\_code()函数，第2块代码**  // 替换规则 ：把函数和变量的调用，替换成各类型专用的调用语句  $specialized\_replacements = array(  "/OP1\_TYPE/" => $op1\_type[$op1],  "/OP2\_TYPE/" => $op2\_type[$op2],  ...  );  // 替换源码中的内容  $code = preg\_replace(array\_keys($specialized\_replacements), array\_values($specialized\_replacements), $code);  ... |

如上所示，这一步要进行很多正则表达式替换，下面分成几组介绍这些替换规则。

## 一）访问操作对象的变量

### 1、规则列表

替换规则代码如下：

|  |
| --- |
| "/OP1\_TYPE/" => $op1\_type[$op1],  "/OP2\_TYPE/" => $op2\_type[$op2],  "/GET\_OP1\_ZVAL\_PTR\(([^)]\*)\)/" => $op1\_get\_zval\_ptr[$op1],  "/GET\_OP2\_ZVAL\_PTR\(([^)]\*)\)/" => $op2\_get\_zval\_ptr[$op2],  "/GET\_OP1\_ZVAL\_PTR\_DEREF\(([^)]\*)\)/" => $op1\_get\_zval\_ptr\_deref[$op1],  "/GET\_OP2\_ZVAL\_PTR\_DEREF\(([^)]\*)\)/" => $op2\_get\_zval\_ptr\_deref[$op2],  "/GET\_OP1\_ZVAL\_PTR\_UNDEF\(([^)]\*)\)/" => $op1\_get\_zval\_ptr\_undef[$op1],  "/GET\_OP2\_ZVAL\_PTR\_UNDEF\(([^)]\*)\)/" => $op2\_get\_zval\_ptr\_undef[$op2],  "/GET\_OP1\_ZVAL\_PTR\_PTR\(([^)]\*)\)/" => $op1\_get\_zval\_ptr\_ptr[$op1],  "/GET\_OP2\_ZVAL\_PTR\_PTR\(([^)]\*)\)/" => $op2\_get\_zval\_ptr\_ptr[$op2],  "/GET\_OP1\_ZVAL\_PTR\_PTR\_UNDEF\(([^)]\*)\)/" => $op1\_get\_zval\_ptr\_ptr\_undef[$op1],  "/GET\_OP2\_ZVAL\_PTR\_PTR\_UNDEF\(([^)]\*)\)/" => $op2\_get\_zval\_ptr\_ptr\_undef[$op2],  "/GET\_OP1\_OBJ\_ZVAL\_PTR\(([^)]\*)\)/" => $op1\_get\_obj\_zval\_ptr[$op1],  "/GET\_OP2\_OBJ\_ZVAL\_PTR\(([^)]\*)\)/" => $op2\_get\_obj\_zval\_ptr[$op2],  "/GET\_OP1\_OBJ\_ZVAL\_PTR\_UNDEF\(([^)]\*)\)/" => $op1\_get\_obj\_zval\_ptr\_undef[$op1],  "/GET\_OP2\_OBJ\_ZVAL\_PTR\_UNDEF\(([^)]\*)\)/" => $op2\_get\_obj\_zval\_ptr\_undef[$op2],  "/GET\_OP1\_OBJ\_ZVAL\_PTR\_DEREF\(([^)]\*)\)/" => $op1\_get\_obj\_zval\_ptr\_deref[$op1],  "/GET\_OP2\_OBJ\_ZVAL\_PTR\_DEREF\(([^)]\*)\)/" => $op2\_get\_obj\_zval\_ptr\_deref[$op2],  "/GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR\(([^)]\*)\)/" => $op1\_get\_obj\_zval\_ptr\_ptr[$op1],  "/GET\_OP2\_OBJ\_ZVAL\_PTR\_PTR\(([^)]\*)\)/" => $op2\_get\_obj\_zval\_ptr\_ptr[$op2],  "/GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR\_UNDEF\(([^)]\*)\)/" => $op1\_get\_obj\_zval\_ptr\_ptr\_undef[$op1],  "/GET\_OP2\_OBJ\_ZVAL\_PTR\_PTR\_UNDEF\(([^)]\*)\)/" => $op2\_get\_obj\_zval\_ptr\_ptr\_undef[$op2],  "/FREE\_OP1\(\)/" => $op1\_free\_op[$op1],  "/FREE\_OP2\(\)/" => $op2\_free\_op[$op2],  "/FREE\_OP1\_IF\_VAR\(\)/" => $op1\_free\_op\_if\_var[$op1],  "/FREE\_OP2\_IF\_VAR\(\)/" => $op2\_free\_op\_if\_var[$op2],  ...  "/OP\_DATA\_TYPE/" => $op\_data\_type[isset($extra\_spec['OP\_DATA']) ? $extra\_spec['OP\_DATA'] : "ANY"],  "/GET\_OP\_DATA\_ZVAL\_PTR\(([^)]\*)\)/" => $op\_data\_get\_zval\_ptr[isset($extra\_spec['OP\_DATA']) ? $extra\_spec['OP\_DATA'] : "ANY"],  "/GET\_OP\_DATA\_ZVAL\_PTR\_UNDEF\(([^)]\*)\)/" => $op\_data\_get\_zval\_ptr\_undef[isset($extra\_spec['OP\_DATA']) ? $extra\_spec['OP\_DATA'] : "ANY"],  "/GET\_OP\_DATA\_ZVAL\_PTR\_DEREF\(([^)]\*)\)/" => $op\_data\_get\_zval\_ptr\_deref[isset($extra\_spec['OP\_DATA']) ? $extra\_spec['OP\_DATA'] : "ANY"],  "/GET\_OP\_DATA\_ZVAL\_PTR\_PTR\(([^)]\*)\)/" => $op\_data\_get\_zval\_ptr\_ptr[isset($extra\_spec['OP\_DATA']) ? $extra\_spec['OP\_DATA'] : "ANY"],  "/FREE\_OP\_DATA\(\)/" => $op\_data\_free\_op[isset($extra\_spec['OP\_DATA']) ? $extra\_spec['OP\_DATA'] : "ANY"], |

如上所示，以上规则用于在zend\_vm\_def.h文件中替换内容，，现将它们的意义和用法归纳如下。

#### **a）获取操作对象的类型**

以下伪常量用于获取操作对象和操作数据的类型：

|  |  |  |  |
| --- | --- | --- | --- |
| **伪常量名** | **对应的规则变量** | **引用次数** | **功能说明** |
| OP1\_TYPE | $op1\_type | 267 | 获取操作对象类型 |
| OP2\_TYPE | $op2\_type | 184 |
| OP\_DATA\_TYPE | $op\_data\_type | 18 | 获取操作数据的类型 |

#### **b）访问操作对象的变量**

以下伪常量用访问操作对象和操作数据的变量：

|  |  |  |  |
| --- | --- | --- | --- |
| **伪函数名** | **对应的规则变量** | **引用次数** | **功能说明** |
| GET\_OP1\_ZVAL\_PTR() | $op1\_get\_zval\_ptr | 31 | 访问操作对象的变量 |
| GET\_OP2\_ZVAL\_PTR() | $op2\_get\_zval\_ptr | 40 |
| GET\_OP1\_ZVAL\_PTR\_PTR() | $op1\_get\_zval\_ptr\_ptr | 8 | 访问操作对象的变量，解间接引用 |
| GET\_OP2\_ZVAL\_PTR\_PTR() | $op2\_get\_zval\_ptr\_ptr | 1 |
| GET\_OP1\_ZVAL\_PTR\_DEREF() | $op1\_get\_zval\_ptr\_deref | 6 | 访问操作对象的变量，解引用 |
| GET\_OP2\_ZVAL\_PTR\_DEREF() | $op2\_get\_zval\_ptr\_deref | 4 |
| GET\_OP1\_ZVAL\_PTR\_UNDEF() | $op1\_get\_zval\_ptr\_undef | 75 | 访问操作对象的变量，CV类型直接获取 |
| GET\_OP2\_ZVAL\_PTR\_UNDEF() | $op2\_get\_zval\_ptr\_undef | 63 |
| GET\_OP1\_ZVAL\_PTR\_PTR\_UNDEF() | $op1\_get\_zval\_ptr\_ptr\_undef | 25 | 访问操作对象的变量，解间接引用，CV类型直接获取 |
| GET\_OP2\_ZVAL\_PTR\_PTR\_UNDEF() | $op2\_get\_zval\_ptr\_ptr\_undef | 0 |
| GET\_OP1\_OBJ\_ZVAL\_PTR() | $op1\_get\_obj\_zval\_ptr | 2 | 访问操作对象的变量，unused类型返回$this |
| GET\_OP2\_OBJ\_ZVAL\_PTR() | $op2\_get\_obj\_zval\_ptr | 2 |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_UNDEF() | $op1\_get\_obj\_zval\_ptr\_undef | 4 | 访问操作对象的变量，CV类型直接获取，unused类型返回$this |
| GET\_OP2\_OBJ\_ZVAL\_PTR\_UNDEF() | $op2\_get\_obj\_zval\_ptr\_undef | 0 |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_DEREF() | $op1\_get\_obj\_zval\_ptr\_deref | 0 | 访问操作对象的变量，var类型解引用，unused返回$this |
| GET\_OP2\_OBJ\_ZVAL\_PTR\_DEREF() | $op2\_get\_obj\_zval\_ptr\_deref | 0 |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR() | $op1\_get\_obj\_zval\_ptr\_ptr | 0 | 访问操作对象的变量，解间接引用，unused返回$this |
| GET\_OP2\_OBJ\_ZVAL\_PTR\_PTR() | $op2\_get\_obj\_zval\_ptr\_ptr | 0 |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR\_UNDEF() | $op1\_get\_obj\_zval\_ptr\_ptr\_undef | 11 | 访问操作对象的变量，解间接引用，unused返回$this，CV类型直接获取 |
| GET\_OP2\_OBJ\_ZVAL\_PTR\_PTR\_UNDEF() | $op2\_get\_obj\_zval\_ptr\_ptr\_undef | 0 |
| GET\_OP\_DATA\_ZVAL\_PTR() | $op\_data\_get\_zval\_ptr | 5 | 读取操作数据变量 |
| GET\_OP\_DATA\_ZVAL\_PTR\_UNDEF() | $op\_data\_get\_zval\_ptr\_undef | 3 | 读取操作数据变量，CV类型直接获取 |
| GET\_OP\_DATA\_ZVAL\_PTR\_DEREF() | $op\_data\_get\_zval\_ptr\_deref | 0 | 读取操作数据变量，普通变量和CV类型解引用 |
| GET\_OP\_DATA\_ZVAL\_PTR\_PTR() | $op\_data\_get\_zval\_ptr\_ptr | 2 | 访问操作数据变量，解间接引用 |

以上伪函数可分为几种类型：

1）“\_ZVAL\_PTR”结尾，表示访问操作对象或操作数据的变量；

2）“\_ZVAL\_PTR\_PTR”结尾，表示访问操作对象或操作数据的变量，遇到间接引用（INDIRECT）时追踪引用目标；

3）“\_ZVAL\_PTR\_DEREF”结尾，表示访问操作对象或操作数据的变量，遇到引用类型（zend\_reference）时追踪引用目标；

4）“\_ZVAL\_PTR\_UNDEF”结尾，表示不关心是否引用，直接返回操作对象的的变量，这个伪函数主要针对CV类型，后续会详细介绍CV类型的操作；

5）“\_ZVAL\_PTR\_PTR\_UNDEF”结尾，表示访问操作对象或操作数据的变量，遇到间接引用（INDIRECT）时追踪引用目标，但CV类型不追踪，直接返回变量；

6）“\_OBJ\_”关键字，表示访问对象类型，碰到UNUSED类型时会返回&EX(This)；

7）“\_DATA\_”关键字，表示访问操作数据。

关于间接引用（INDIRECT）和引用类型（zend\_reference）的更多介绍参见“类型篇”。

#### **c）销毁操作对象的变量**

以下伪常量用销毁操作对象和操作数据的变量：

|  |  |  |  |
| --- | --- | --- | --- |
| **伪函数名** | **对应的规则变量** | **引用次数** | **功能说明** |
| FREE\_OP1 | $op1\_free\_op | 126 | 释放操作对象的变量 |
| FREE\_OP2 | $op2\_free\_op | 64 |
| FREE\_OP1\_IF\_VAR | $op1\_free\_op\_if\_var | 8 | 如果类型是VAR释放操作对象的变量 |
| FREE\_OP2\_IF\_VAR | $op2\_free\_op\_if\_var | 0 |
| FREE\_OP\_DATA | $op\_data\_free\_op | 16 | 释放操作数据变量 |

下面介绍伪函数在每种类型中对应的实际操作。

### 2、访问普通（VAR类型）变量

VAR类型表示普通变量，对应的操作如下。

|  |  |
| --- | --- |
| **伪常量名/伪函数名** | **实际常量名/函数名** |
| OP1\_TYPE | IS\_VAR |
| GET\_OP1\_ZVAL\_PTR() | \_get\_zval\_ptr\_var() |
| GET\_OP1\_ZVAL\_PTR\_PTR() | \_get\_zval\_ptr\_ptr\_var() |
| GET\_OP1\_ZVAL\_PTR\_DEREF() | \_get\_zval\_ptr\_var\_deref() |
| GET\_OP1\_ZVAL\_PTR\_UNDEF() | \_get\_zval\_ptr\_var() |
| GET\_OP1\_ZVAL\_PTR\_PTR\_UNDEF() | \_get\_zval\_ptr\_ptr\_var() |
| GET\_OP1\_OBJ\_ZVAL\_PTR() | \_get\_zval\_ptr\_var() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_UNDEF() | \_get\_zval\_ptr\_var() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_DEREF() | \_get\_zval\_ptr\_var\_deref() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR() | \_get\_zval\_ptr\_ptr\_var() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR\_UNDEF() | \_get\_zval\_ptr\_ptr\_var() |
| FREE\_OP1() | zval\_ptr\_dtor\_nogc() |
| FREE\_OP1\_IF\_VAR() | zval\_ptr\_dtor\_nogc() |
| OP\_DATA\_TYPE | IS\_VAR |
| GET\_OP\_DATA\_ZVAL\_PTR() | \_get\_zval\_ptr\_var() |
| GET\_OP\_DATA\_ZVAL\_PTR\_UNDEF() | \_get\_zval\_ptr\_var() |
| GET\_OP\_DATA\_ZVAL\_PTR\_DEREF() | \_get\_zval\_ptr\_var\_deref() |
| GET\_OP\_DATA\_ZVAL\_PTR\_PTR() | \_get\_zval\_ptr\_ptr\_var() |
| FREE\_OP\_DATA() | zval\_ptr\_dtor\_nogc() |

### 3、访问临时（TMP类型）变量

TMP类型表示临时变量，对应的操作如下。

|  |  |
| --- | --- |
| **伪常量名/伪函数名** | **实际常量名/函数名** |
| OP1\_TYPE | IS\_TMP\_VAR |
| GET\_OP1\_ZVAL\_PTR() | \_get\_zval\_ptr\_tmp() |
| GET\_OP1\_ZVAL\_PTR\_PTR() | NULL |
| GET\_OP1\_ZVAL\_PTR\_DEREF() | \_get\_zval\_ptr\_tmp() |
| GET\_OP1\_ZVAL\_PTR\_UNDEF() | \_get\_zval\_ptr\_tmp() |
| GET\_OP1\_ZVAL\_PTR\_PTR\_UNDEF() | NULL |
| GET\_OP1\_OBJ\_ZVAL\_PTR() | \_get\_zval\_ptr\_tmp() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_UNDEF() | \_get\_zval\_ptr\_tmp() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_DEREF() | \_get\_zval\_ptr\_tmp() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR() | NULL |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR\_UNDEF() | NULL |
| FREE\_OP1() | zval\_ptr\_dtor\_nogc() |
| FREE\_OP1\_IF\_VAR() |  |
| OP\_DATA\_TYPE | IS\_TMP\_VAR |
| GET\_OP\_DATA\_ZVAL\_PTR() | \_get\_zval\_ptr\_tmp() |
| GET\_OP\_DATA\_ZVAL\_PTR\_UNDEF() | \_get\_zval\_ptr\_tmp() |
| GET\_OP\_DATA\_ZVAL\_PTR\_DEREF() | \_get\_zval\_ptr\_tmp() |
| GET\_OP\_DATA\_ZVAL\_PTR\_PTR() | NULL |
| FREE\_OP\_DATA() | zval\_ptr\_dtor\_nogc() |

### 4、访问常量（CONST类型）

CONST类型表示常量，对应的操作如下。

|  |  |
| --- | --- |
| **伪常量名/伪函数名** | **实际常量名/函数名** |
| OP1\_TYPE | IS\_CONST |
| GET\_OP1\_ZVAL\_PTR() | RT\_CONSTANT() |
| GET\_OP1\_ZVAL\_PTR\_PTR() | NULL |
| GET\_OP1\_ZVAL\_PTR\_DEREF() | RT\_CONSTANT() |
| GET\_OP1\_ZVAL\_PTR\_UNDEF() | RT\_CONSTANT() |
| GET\_OP1\_ZVAL\_PTR\_PTR\_UNDEF() | NULL |
| GET\_OP1\_OBJ\_ZVAL\_PTR() | RT\_CONSTANT() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_UNDEF() | RT\_CONSTANT() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_DEREF() | RT\_CONSTANT() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR() | NULL |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR\_UNDEF() | NULL |
| FREE\_OP1() |  |
| FREE\_OP1\_IF\_VAR() |  |
| OP\_DATA\_TYPE | IS\_CONST |
| GET\_OP\_DATA\_ZVAL\_PTR() | RT\_CONSTANT() |
| GET\_OP\_DATA\_ZVAL\_PTR\_UNDEF() | RT\_CONSTANT() |
| GET\_OP\_DATA\_ZVAL\_PTR\_DEREF() | RT\_CONSTANT() |
| GET\_OP\_DATA\_ZVAL\_PTR\_PTR() | NULL |
| FREE\_OP\_DATA() |  |

如上所示，常量的操作非常简单，读取时只能用RT\_CONSTANT()宏程序，并且不支持销毁。

### 5、访问编译（CV类型）变量

CV类型表示编译变量，对应的操作如下。

|  |  |
| --- | --- |
| **伪常量名/伪函数名** | **实际常量名/函数名** |
| OP1\_TYPE | IS\_CV |
| GET\_OP1\_ZVAL\_PTR() | \_get\_zval\_ptr\_cv\_\\1() |
| GET\_OP1\_ZVAL\_PTR\_PTR() | \_get\_zval\_ptr\_cv\_\\1() |
| GET\_OP1\_ZVAL\_PTR\_DEREF() | \_get\_zval\_ptr\_cv\_deref\_\\1() |
| GET\_OP1\_ZVAL\_PTR\_UNDEF() | EX\_VAR() |
| GET\_OP1\_ZVAL\_PTR\_PTR\_UNDEF() | EX\_VAR() |
| GET\_OP1\_OBJ\_ZVAL\_PTR() | \_get\_zval\_ptr\_cv\_\\1() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_UNDEF() | EX\_VAR() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_DEREF() | \_get\_zval\_ptr\_cv\_deref\_\\1() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR() | \_get\_zval\_ptr\_cv\_\\1() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR\_UNDEF() | EX\_VAR() |
| FREE\_OP1() |  |
| FREE\_OP1\_IF\_VAR() |  |
| OP\_DATA\_TYPE | IS\_CV |
| GET\_OP\_DATA\_ZVAL\_PTR() | \_get\_zval\_ptr\_cv\_\\1() |
| GET\_OP\_DATA\_ZVAL\_PTR\_UNDEF() | EX\_VAR |
| GET\_OP\_DATA\_ZVAL\_PTR\_DEREF() | \_get\_zval\_ptr\_cv\_deref\_\\1() |
| GET\_OP\_DATA\_ZVAL\_PTR\_PTR() | \_get\_zval\_ptr\_cv\_\\1() |
| FREE\_OP\_DATA() |  |

如上所示，CV类型的变量操作比较复杂，主要在于\_get\_zval\_ptr\_cv\_\\1()和\_get\_zval\_ptr\_cv\_deref\_\\1()这两个替换规则，对应多个函数。

|  |  |
| --- | --- |
| **替换规则** | **对应的函数** |
| \_get\_zval\_ptr\_cv\_\\1() | \_get\_zval\_ptr\_cv\_BP\_VAR\_R() |
| \_get\_zval\_ptr\_cv\_BP\_VAR\_IS() |
| \_get\_zval\_ptr\_cv\_BP\_VAR\_RW() |
| \_get\_zval\_ptr\_cv\_BP\_VAR\_W() |
| \_get\_zval\_ptr\_cv\_deref\_\\1() | \_get\_zval\_ptr\_cv\_deref\_BP\_VAR\_R |

### 6、访问未使用（UNUSED类型）变量

UNUSED类型表示未使用的变量，对应的操作如下。

|  |  |
| --- | --- |
| **伪常量名/伪函数名** | **实际常量名/函数名** |
| OP1\_TYPE | IS\_UNUSED |
| GET\_OP1\_ZVAL\_PTR() | NULL |
| GET\_OP1\_ZVAL\_PTR\_PTR() | NULL |
| GET\_OP1\_ZVAL\_PTR\_DEREF() | NULL |
| GET\_OP1\_ZVAL\_PTR\_UNDEF() | NULL |
| GET\_OP1\_ZVAL\_PTR\_PTR\_UNDEF() | NULL |
| GET\_OP1\_OBJ\_ZVAL\_PTR() | &EX(This) |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_UNDEF() | &EX(This) |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_DEREF() | &EX(This) |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR() | &EX(This) |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR\_UNDEF() | &EX(This) |
| FREE\_OP1() |  |
| FREE\_OP1\_IF\_VAR() |  |
| OP\_DATA\_TYPE | IS\_UNUSED |
| GET\_OP\_DATA\_ZVAL\_PTR() | NULL |
| GET\_OP\_DATA\_ZVAL\_PTR\_UNDEF() | NULL |
| GET\_OP\_DATA\_ZVAL\_PTR\_DEREF() |  |
| GET\_OP\_DATA\_ZVAL\_PTR\_PTR() | IS\_UNUSED |
| FREE\_OP\_DATA() | NULL |

如上所示UNUSED类型的变量不支持销毁操作，一般情况下都返回NULL，在对象中使用时返回&EX(This)，相当于PHP代码中的$this。

### 6、访问TMPVAR类型变量

TMPVAR类型表示临时变量或普通变量，对应的操作如下。

|  |  |
| --- | --- |
| **伪常量名/伪函数名** | **实际常量名/函数名** |
| OP1\_TYPE | (IS\_TMP\_VAR|IS\_VAR) |
| GET\_OP1\_ZVAL\_PTR() | \_get\_zval\_ptr\_var() |
| GET\_OP1\_ZVAL\_PTR\_PTR() | ??? |
| GET\_OP1\_ZVAL\_PTR\_DEREF() | ??? |
| GET\_OP1\_ZVAL\_PTR\_UNDEF() | \_get\_zval\_ptr\_var() |
| GET\_OP1\_ZVAL\_PTR\_PTR\_UNDEF() | ??? |
| GET\_OP1\_OBJ\_ZVAL\_PTR() | \_get\_zval\_ptr\_var() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_UNDEF() | \_get\_zval\_ptr\_var() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_DEREF() | ??? |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR() | ??? |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR\_UNDEF() | ??? |
| FREE\_OP1() | zval\_ptr\_dtor\_nogc() |
| FREE\_OP1\_IF\_VAR() | ??? |
| OP\_DATA\_TYPE | (IS\_TMP\_VAR|IS\_VAR) |
| GET\_OP\_DATA\_ZVAL\_PTR() | \_get\_zval\_ptr\_var() |
| GET\_OP\_DATA\_ZVAL\_PTR\_UNDEF() | \_get\_zval\_ptr\_var() |
| GET\_OP\_DATA\_ZVAL\_PTR\_DEREF() | ??? |
| GET\_OP\_DATA\_ZVAL\_PTR\_PTR() | ??? |
| FREE\_OP\_DATA() | zval\_ptr\_dtor\_nogc() |

如上所示，同时支持TMP/VAR三种类型的操作并不多，而且只能使用\_get\_zval\_ptr\_var()函数调用EX\_VAR()宏程序从操作数据中获取变量，不能进行追踪引用等操作。

### 7、访问TMPVARCV类型变量

VAR类型表示临时变量或普通变量或编译变量，对应的操作如下。

|  |  |
| --- | --- |
| **伪常量名/伪函数名** | **实际常量名/函数名** |
| OP1\_TYPE | (IS\_TMP\_VAR|IS\_VAR|IS\_CV) |
| GET\_OP1\_ZVAL\_PTR() | ??? |
| GET\_OP1\_ZVAL\_PTR\_PTR() | ??? |
| GET\_OP1\_ZVAL\_PTR\_DEREF() | ??? |
| GET\_OP1\_ZVAL\_PTR\_UNDEF() | EX\_VAR() |
| GET\_OP1\_ZVAL\_PTR\_PTR\_UNDEF() | ??? |
| GET\_OP1\_OBJ\_ZVAL\_PTR() | ??? |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_UNDEF() | EX\_VAR() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_DEREF() | ??? |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR() | ??? |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR\_UNDEF() | ??? |
| FREE\_OP1() | ??? |
| FREE\_OP1\_IF\_VAR() | ??? |
| OP\_DATA\_TYPE | (IS\_TMP\_VAR|IS\_VAR|IS\_CV) |
| GET\_OP\_DATA\_ZVAL\_PTR() | ??? |
| GET\_OP\_DATA\_ZVAL\_PTR\_UNDEF() | EX\_VAR() |
| GET\_OP\_DATA\_ZVAL\_PTR\_DEREF() | ??? |
| GET\_OP\_DATA\_ZVAL\_PTR\_PTR() | ??? |
| FREE\_OP\_DATA() | ??? |

如上所示，同时支持TMP/VAR/CV三种类型的操作非常少，而且只能使用EX\_VAR()宏程序从操作数据中获取变量，不能进行追踪引用等操作，且不支持销毁操作。

### 8、访问ANY类型变量

VAR类型表示任意类型，对应的操作如下。

|  |  |
| --- | --- |
| **伪常量名/伪函数名** | **实际常量名/函数名** |
| OP1\_TYPE | opline->op1\_type |
| GET\_OP1\_ZVAL\_PTR() | get\_zval\_ptr() |
| GET\_OP1\_ZVAL\_PTR\_PTR() | get\_zval\_ptr\_ptr() |
| GET\_OP1\_ZVAL\_PTR\_DEREF() | get\_zval\_ptr\_deref() |
| GET\_OP1\_ZVAL\_PTR\_UNDEF() | get\_zval\_ptr\_undef() |
| GET\_OP1\_ZVAL\_PTR\_PTR\_UNDEF() | get\_zval\_ptr\_ptr\_undef() |
| GET\_OP1\_OBJ\_ZVAL\_PTR() | get\_obj\_zval\_ptr() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_UNDEF() | get\_obj\_zval\_ptr\_undef() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_DEREF() | get\_obj\_zval\_ptr() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR() | get\_obj\_zval\_ptr\_ptr() |
| GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR\_UNDEF() | get\_obj\_zval\_ptr\_ptr() |
| FREE\_OP1() | FREE\_OP() |
| FREE\_OP1\_IF\_VAR() | zval\_ptr\_dtor\_nogc() |
| OP\_DATA\_TYPE | (opline+1)->op1\_type |
| GET\_OP\_DATA\_ZVAL\_PTR() | get\_op\_data\_zval\_ptr\_r() |
| GET\_OP\_DATA\_ZVAL\_PTR\_UNDEF() | get\_op\_data\_zval\_ptr\_undef() |
| GET\_OP\_DATA\_ZVAL\_PTR\_DEREF() | get\_op\_data\_zval\_ptr\_deref\_r() |
| GET\_OP\_DATA\_ZVAL\_PTR\_PTR() | get\_zval\_ptr\_ptr() |
| FREE\_OP\_DATA() | FREE\_OP() |

ANY类型并不是实际存在的类型，在实际操作中碰到的操作对象变量一定是VAR/TMP/CV/CONST/UNUSED五种类型之一，所以ANY类型的操作函数中，需要对这5种类型分别处理。以get\_zval\_ptr()宏程序为例，它调用\_get\_zval\_ptr()函数进行处理，函数代码如下：

|  |
| --- |
| static zend\_always\_inline zval \*\_get\_zval\_ptr(int op\_type, znode\_op node, int type ){  if (op\_type & (IS\_TMP\_VAR|IS\_VAR)) { // 如果类型是 变量或临时变量  if (!ZEND\_DEBUG || op\_type == IS\_VAR) { // 非调试模式 或 变量  return \_get\_zval\_ptr\_var(node.var EXECUTE\_DATA\_CC);  } else { // 调试模式 并且 是临时变量  ZEND\_ASSERT(op\_type == IS\_TMP\_VAR); // 必须是临时变量  // 通过偏移量获取 执行数据 中的 第n个 zval, 不可以是引用类型  return \_get\_zval\_ptr\_tmp(node.var EXECUTE\_DATA\_CC);  }  } else { // 类型是其他  if (op\_type == IS\_CONST) { // 类型是常量  return RT\_CONSTANT(opline, node); // 在p1中通过偏移量p2.constant 获取变量  } else if (op\_type == IS\_CV) { // 类型是编译变量  // 根据序号 和 操作类型，返回zval  return \_get\_zval\_ptr\_cv(node.var, type EXECUTE\_DATA\_CC);  } else { // 其他类型（UNUSED）  return NULL;  }  }  } |

如上所示，\_get\_zval\_ptr()函数中处理各种类型。所以在ANY类型和在5种准确类型的处理中，处理方式有时会有不同，主要在以下两个操作：

1）在调用GET\_OP1\_OBJ\_ZVAL\_PTR\_DEREF()和GET\_OP2\_OBJ\_ZVAL\_PTR\_DEREF()这两个伪函数时，在ANY类型的处理中，碰到VAR类型时会替换成调用\_get\_zval\_ptr\_var()函数直接获取变量；在VAR类型的处理中，会替换成调用\_get\_zval\_ptr\_var\_deref()函数，获取变量并追踪引用目标；

2）在调用GET\_OP1\_OBJ\_ZVAL\_PTR\_PTR\_UNDEF()和GET\_OP2\_OBJ\_ZVAL\_PTR\_PTR\_ UNDEF()伪函数时，在ANY类型的处理中，碰到CV类型时会替换成调用\_get\_zval\_ptr\_cv()函数获取变量；在CV类型的处理中，会使用EX\_VAR()宏程序直接获取变量。

## 二）辅助功能

### 1、ZEND\_VM\_SPEC伪常量

替换规则代码如下：

|  |
| --- |
| // op1和op2 都是ANY。 23次引用。  "/\!ZEND\_VM\_SPEC/m" => ($op1!="ANY"||$op2!="ANY")?"0":"1",  // op1或op2有一个不是ANY。  "/ZEND\_VM\_SPEC/m" => ($op1!="ANY"||$op2!="ANY")?"1":"0", |

ZEND\_VM\_SPEC伪常量的值，在两个操作对象的类型都是ANY时是0，否则是1。它用来作为辅助判断条件，提升判断的效率，例如：

|  |
| --- |
| if (ZEND\_VM\_SPEC && OP1\_TYPE == IS\_CONST && OP2\_TYPE == IS\_CONST) |

如果两个操作对象类型都是ANY，代码会变成：

|  |
| --- |
| if (0 && OP1\_TYPE == IS\_CONST && OP2\_TYPE == IS\_CONST) |

这样就不用进行后面两个操作对象类型的判断了。

### 2、goto跳转语句

替换规则代码如下：

|  |
| --- |
| "/ZEND\_VM\_C\_LABEL\(\s\*([A-Za-z\_]\*)\s\*\)/m" => "\\1".(($spec && $kind != ZEND\_VM\_KIND\_CALL)?("\_SPEC".$prefix[$op1].$prefix[$op2].extra\_spec\_name($extra\_spec)):""),  "/ZEND\_VM\_C\_GOTO\(\s\*([A-Za-z\_]\*)\s\*\)/m" => "goto \\1".(($spec && $kind != ZEND\_VM\_KIND\_CALL)?("\_SPEC".$prefix[$op1].$prefix[$op2].extra\_spec\_name($extra\_spec)):""), |

如上所示，有两行替换规则，其中ZEND\_VM\_C\_LABEL()伪函数用于生成C语言中的goto跳转标签，ZEND\_VM\_C\_GOTO()伪函数用于生成C语言中的goto跳转语句，例如：

|  |
| --- |
| ZEND\_VM\_C\_GOTO(add\_double);  ZEND\_VM\_C\_LABEL(add\_double): |

生成的代码如下：

|  |
| --- |
| goto add\_double;  add\_double: |

### 3、替换#if

替换规则代码如下：

|  |
| --- |
| "/^#(\s\*)if\s+1\s\*\\|\\|.\*[^\\\\]$/m" => "#\\1if 1",  "/^#(\s\*)if\s+0\s\*&&.\*[^\\\\]$/m" => "#\\1if 0",  "/^#(\s\*)elif\s+1\s\*\\|\\|.\*[^\\\\]$/m" => "#\\1elif 1",  "/^#(\s\*)elif\s+0\s\*&&.\*[^\\\\]$/m" => "#\\1elif 0", |

如上所示，这几行规则用来替换预处理程序中的条件判断，实际上并没有用到。

### 4、RETURN\_VALUE\_USED宏程序

替换规则代码如下：

|  |
| --- |
| "/RETURN\_VALUE\_USED\(opline\)/" => isset($extra\_spec['RETVAL']) ? $extra\_spec['RETVAL'] : "RETURN\_VALUE\_USED(opline)", |

如上所示，RETURN\_VALUE\_USED是虚拟机中是实际存在的宏程序。当不存在RETVAL专用规则时，“RETURN\_VALUE\_USED(opline)”语句会被原样保留。否则替换成RETVAL专用规则的开启状态。

### 5、MAX\_ARG\_FLAG\_NUM常量

替换规则代码如下：

|  |
| --- |
| "/arg\_num <= MAX\_ARG\_FLAG\_NUM/" => isset($extra\_spec['QUICK\_ARG']) ? $extra\_spec['QUICK\_ARG'] : "arg\_num <= MAX\_ARG\_FLAG\_NUM", |

如上所示，MAX\_ARG\_FLAG\_NUM是虚拟机中是实际存在的常量。当不存在QUICK\_ARG专用规则时，“arg\_num <= MAX\_ARG\_FLAG\_NUM”语句会被原样保留。否则替换成QUICK\_ARG专用规则的开启状态。

### 6、ZEND\_ISEMPTY伪常量

替换规则代码如下：

|  |
| --- |
| "/opline->extended\_value\s\*&\s\*ZEND\_ISEMPTY/" => isset($extra\_spec['ISSET']) ?  ($extra\_spec['ISSET'] == 0 ? "0" : "1") : "\\0", |

如上所示，此规则会把“opline->extended\_value & ZEND\_ISEMPTY”这个语句，替换成：

1）当启用了ISSET专用规则时：1；

2）当未启用ISSET专用规则时：0；

3）其他情况（不存在ISSET专用规则）时，原样保留此语句。

## 三）智能分支（SMART\_BRANCH）

智能分支（SMART\_BRANCH）是唯一一个有3个取值的专用规则，3可能的值分别是0，1，2；这是在parse\_spec\_rules()函数中规定好的。

### 1、ZEND\_VM\_SMART\_BRANCH()伪函数

替换规则代码如下：

|  |
| --- |
| "/ZEND\_VM\_SMART\_BRANCH\(\s\*([^,)]\*)\s\*,\s\*([^)]\*)\s\*\)/" =>  isset($extra\_spec['SMART\_BRANCH']) ?  (  $extra\_spec['SMART\_BRANCH'] == 1 ?  "ZEND\_VM\_SMART\_BRANCH\_JMPZ(\\1, \\2)" :  (  $extra\_spec['SMART\_BRANCH'] == 2 ?  "ZEND\_VM\_SMART\_BRANCH\_JMPNZ(\\1, \\2)" :  "ZEND\_VM\_SMART\_BRANCH\_NONE(\\1, \\2)"  )  )  : "ZEND\_VM\_SMART\_BRANCH(\\1, \\2)", |

如上所示，ZEND\_VM\_SMART\_BRANCH()伪函数接收2个参数，例如：

|  |
| --- |
| ZEND\_VM\_SMART\_BRANCH(ret == 0, 1); |

根据$extra\_spec['SMART\_BRANCH']的取值不同，伪函数会被替换成多个函数：

1）值为1：替换成ZEND\_VM\_SMART\_BRANCH\_JMPZ()函数；

2）值为2：替换成ZEND\_VM\_SMART\_BRANCH\_JMPNZ()函数；

3）值为0：替换成ZEND\_VM\_SMART\_BRANCH\_NONE()函数。

### 2、ZEND\_VM\_SMART\_BRANCH\_TRUE()伪函数

替换规则代码如下：

|  |
| --- |
| "/ZEND\_VM\_SMART\_BRANCH\_TRUE\(\s\*\)/" =>  isset($extra\_spec['SMART\_BRANCH']) ?  (  $extra\_spec['SMART\_BRANCH'] == 1 ?  "ZEND\_VM\_SMART\_BRANCH\_TRUE\_JMPZ()" :  (  $extra\_spec['SMART\_BRANCH'] == 2 ?  "ZEND\_VM\_SMART\_BRANCH\_TRUE\_JMPNZ()" :  "ZEND\_VM\_SMART\_BRANCH\_TRUE\_NONE()")  )  : "ZEND\_VM\_SMART\_BRANCH\_TRUE()" |

如上所示，根据$extra\_spec['SMART\_BRANCH']的取值不同，伪函数会被替换成多个函数：

1）值为1：替换成ZEND\_VM\_SMART\_BRANCH\_TRUE\_JMPZ()函数；

2）值为2：替换成ZEND\_VM\_SMART\_BRANCH\_TRUE\_JMPNZ()函数；

3）值为0：替换成ZEND\_VM\_SMART\_BRANCH\_NONE()函数。

### 3、ZEND\_VM\_SMART\_BRANCH\_FALSE()伪函数

替换规则代码如下：

|  |
| --- |
| "/ZEND\_VM\_SMART\_BRANCH\_FALSE\(\s\*\)/" =>  isset($extra\_spec['SMART\_BRANCH']) ?  (  $extra\_spec['SMART\_BRANCH'] == 1 ?  "ZEND\_VM\_SMART\_BRANCH\_FALSE\_JMPZ()" :  (  $extra\_spec['SMART\_BRANCH'] == 2 ?  "ZEND\_VM\_SMART\_BRANCH\_FALSE\_JMPNZ()" :  "ZEND\_VM\_SMART\_BRANCH\_FALSE\_NONE()"  )  )  : "ZEND\_VM\_SMART\_BRANCH\_FALSE()", |

如上所示，根据$extra\_spec['SMART\_BRANCH']的取值不同，伪函数会被替换成多个函数：

1）值为1：替换成ZEND\_VM\_SMART\_BRANCH\_FALSE\_JMPZ()函数；

2）值为2：替换成ZEND\_VM\_SMART\_BRANCH\_FALSE\_JMPNZ()函数；

3）值为0：替换成ZEND\_VM\_SMART\_BRANCH\_FALSE\_NONE()函数。

## 四）观察者（OBSERVER）

与观察者（OBSERVER）相关的伪函数和伪常量共有以下7个。除ZEND\_OBSERVER\_ENABLED伪常量外，其他伪常量和伪函数只在开启了观察者功能时才会生效，未启用观察者功能时，它们会被替换成空。

### 1、ZEND\_OBSERVER\_ENABLED伪常量

替换规则代码如下：

|  |
| --- |
| "/ZEND\_OBSERVER\_ENABLED/" => isset($extra\_spec['OBSERVER']) && $extra\_spec['OBSERVER'] == 1 ? "1" : "0", |

如上所示，ZEND\_OBSERVER\_ENABLED伪常量用于标记观察者是否启用。

### 2、ZEND\_OBSERVER\_USE\_RETVAL伪常量

替换规则代码如下：

|  |
| --- |
| "/ZEND\_OBSERVER\_USE\_RETVAL/" => isset($extra\_spec['OBSERVER']) && $extra\_spec['OBSERVER'] == 1 ? "zval observer\_retval" : "", |

在启用观察者后，ZEND\_OBSERVER\_USE\_RETVAL伪常量会被替换成如下C语言语句：

|  |
| --- |
| // 声名一个zval类型的临时变量observer\_retval  zval observer\_retval |

### 3、ZEND\_OBSERVER\_USE\_RETVAL()伪函数

替换规则代码如下：

|  |
| --- |
| "/ZEND\_OBSERVER\_SET\_RETVAL\(\)/" => isset($extra\_spec['OBSERVER']) && $extra\_spec['OBSERVER'] == 1 ? "if (!return\_value) { return\_value = &observer\_retval; }" : "", |

在启用观察者后，ZEND\_OBSERVER\_USE\_RETVAL()伪函数会被替换成如下C语言语句：

|  |
| --- |
| // 使用前面声名的临时变量地址作为返回值  if (!return\_value) { return\_value = &observer\_retval; } |

### 4、ZEND\_OBSERVER\_FREE\_RETVAL()伪函数

替换规则代码如下：

|  |
| --- |
| "/ZEND\_OBSERVER\_FREE\_RETVAL\(\)/" => isset($extra\_spec['OBSERVER']) && $extra\_spec['OBSERVER'] == 1 ? "if (return\_value == &observer\_retval) { zval\_ptr\_dtor\_nogc(&observer\_retval); }" : "", |

在启用观察者后，ZEND\_OBSERVER\_FREE\_RETVAL()伪函数会被替换成如下C语言语句：

|  |
| --- |
| // 销毁作为返回值的临时变量  if (return\_value == &observer\_retval) { zval\_ptr\_dtor\_nogc(&observer\_retval); } |

### 5、ZEND\_OBSERVER\_SAVE\_OPLINE()伪函数

替换规则代码如下：

|  |
| --- |
| "/ZEND\_OBSERVER\_SAVE\_OPLINE\(\)/" => isset($extra\_spec['OBSERVER']) && $extra\_spec['OBSERVER'] == 1 ? "SAVE\_OPLINE()" : "", |

在启用观察者后，ZEND\_OBSERVER\_SAVE\_OPLINE()伪函数会被替换成如下C语言语句：

|  |
| --- |
| // Windows中此宏程序无业务逻辑  SAVE\_OPLINE() |

### 6、ZEND\_OBSERVER\_FCALL\_BEGIN()伪函数

替换规则代码如下：

|  |
| --- |
| "/ZEND\_OBSERVER\_FCALL\_BEGIN\(\s\*(.\*)\s\*\)/" => isset($extra\_spec['OBSERVER']) ?  ($extra\_spec['OBSERVER'] == 0 ? "" : "zend\_observer\_fcall\_begin(\\1)")  : "", |

在启用观察者后，ZEND\_OBSERVER\_FCALL\_BEGIN()伪函数会被替换成如下C语言语句：

|  |
| --- |
| // 传入的任意多个参数会保留并传递给此函数  zend\_observer\_fcall\_begin() |

### 7、ZEND\_OBSERVER\_FCALL\_END()伪函数

替换规则代码如下：

|  |
| --- |
| "/ZEND\_OBSERVER\_FCALL\_END\(\s\*([^,]\*)\s\*,\s\*(.\*)\s\*\)/" => isset($extra\_spec['OBSERVER']) ?  ($extra\_spec['OBSERVER'] == 0 ? "" : "zend\_observer\_fcall\_end(\\1, \\2)")  : "", |

在启用观察者后，ZEND\_OBSERVER\_FCALL\_END()伪函数会被替换成如下C语言语句：

|  |
| --- |
| // 传入的2个参数会保留并传递给此函数  zend\_observer\_fcall\_end() |

## 五）与运行模式有关的功能

有些替换规则与运行模式有关，在生虚拟机成处理器和助手代码时，$kind参数的值为ZEND\_VM\_KIND\_CALL模式。此模式下的处理逻辑如下：

|  |
| --- |
| **// gen\_code()函数，第3块代码**  switch ($kind) {  case ZEND\_VM\_KIND\_CALL:  $code = preg\_replace\_callback(  array(  "/EXECUTE\_DATA(?=[^\_])/m", // 后面是非获取匹配  "/ZEND\_VM\_DISPATCH\_TO\_HANDLER\(\s\*([A-Z\_]\*)\s\*\)/m",  "/ZEND\_VM\_DISPATCH\_TO\_HELPER\(\s\*([A-Za-z\_]\*)\s\*(,[^)]\*)?\)/m",  ),  function($matches) use ($spec, $prefix, $op1, $op2, $extra\_spec, $name) {  **// 处理逻辑见下文 ...**  },  $code);  break;  // 其他运行模式 ...  } |

如上所示，主要处理EXECUTE\_DATA()、ZEND\_VM\_DISPATCH\_TO\_HANDLER()、ZEND\_VM\_DISPATCH\_TO\_HELPER()三个伪函数。

### 1、EXECUTE\_DATA()伪常量

匹配正则表达式"/EXECUTE\_DATA(?=[^\_])/m"，处理逻辑如下：

|  |
| --- |
| if (strncasecmp($matches[0], "EXECUTE\_DATA", strlen("EXECUTE\_DATA")) == 0) {  return "execute\_data"; |

如上所示，strncasecmp()函数用于比较前n个字符大小，大小写不敏感。如果匹配到的字串是"EXECUTE\_DATA"，返回字串"execute\_data"。"/EXECUTE\_DATA(?=[^\_])/m"正则表达式的后半部分是非获取匹配，所以这次处理只把"EXECUTE\_DATA"替换成"execute\_data"，没有其他改动。

### 2、ZEND\_VM\_DISPATCH\_TO\_HANDLER()伪函数

匹配正则表达式"/ZEND\_VM\_DISPATCH\_TO\_HANDLER\(\s\*([A-Z\_]\*)\s\*\)/m"，处理逻辑如下：

|  |
| --- |
| **// 紧接上一块代码**  } else if (strncasecmp($matches[0], "ZEND\_VM\_DISPATCH\_TO\_HANDLER", strlen("ZEND\_VM\_DISPATCH\_TO\_HANDLER")) == 0) {  global $opcodes, $opnames;  $handler = $matches[1];  $opcode = $opcodes[$opnames[$handler]];  $inline =  // 如果用 ZEND\_VM\_KIND\_HYBRID 模式生成，并且操作码被其他操作码引用  ZEND\_VM\_KIND == ZEND\_VM\_KIND\_HYBRID && isset($opcode["use"]) &&  // 并且被调用的操作码是热处理器  is\_hot\_handler($opcode["hot"], $op1, $op2, $extra\_spec) &&  // 并且当前正在处理的操作码也是热处理器  is\_hot\_handler($opcodes[$opnames[$name]]["hot"], $op1, $op2, $extra\_spec) ?  "\_INLINE" : ""; // 满足全部条件才添加\_INLINE关键字  return "ZEND\_VM\_TAIL\_CALL(" . opcode\_name($handler, $spec, $op1, $op2, $extra\_spec) . $inline . "\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU))"; |

如上所示，先匹配到ZEND\_VM\_DISPATCH\_TO\_HANDLER()伪函数，传入的参数是要跳转到的操作码名；再通过opcode\_name()函数找到指定的处理器名,使用goto语句跳转过去。

例如：

|  |
| --- |
| ZEND\_VM\_COLD\_CONST\_HANDLER(94, ZEND\_FETCH\_OBJ\_FUNC\_ARG, CONST|TMP|VAR|UNUSED|THIS|CV, CONST|TMPVAR|CV, FETCH\_REF|CACHE\_SLOT){  ...  **ZEND\_VM\_DISPATCH\_TO\_HANDLER(ZEND\_FETCH\_OBJ\_R);**  ...  } |

以上伪码生成多个C语言函数，其中之一代码如下：

|  |
| --- |
| static ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL ZEND\_FETCH\_OBJ\_FUNC\_ARG\_SPEC\_VAR\_CV\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS){  ... **ZEND\_VM\_TAIL\_CALL(ZEND\_FETCH\_OBJ\_R\_SPEC\_TMPVAR\_CV\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU));**  ...  } |

is\_hot\_handler()函数已经在“gen\_executor\_code()函数”章节中介绍过。$opnames变量中的名称是在解析处理器名时添加进去的，详情参见第一章“解析处理器名”。

#### **opcode\_name()函数**

opcode\_name()函数用于生成操作码对应的函数名，代码如下：

|  |
| --- |
| function opcode\_name($name, $spec, $op1, $op2, $extra\_spec) {  global $prefix, $opnames, $opcodes;  $extra = "";  if (isset($opnames[$name])) { //操作码名  $opcode = $opcodes[$opnames[$name]]; // 操作码  if (!isset($opcode["op1"][$op1])) { **// 第一个操作对象的类型适配**  // op1 是 TMP 或 VAR 并且支持 TMPVAR 类型  if (($op1 == 'TMP' || $op1 == 'VAR') && isset($opcode["op1"]["TMPVAR"])) {  $op1 = "TMPVAR"; // p1 使用 TMPVAR 类型  // op1 是 TMP 或 VAR 并且支持 TMPVARCV 类型  } else if (($op1 == 'TMP' || $op1 == 'VAR') &&  isset($opcode["op1"]["TMPVARCV"])) {  $op1 = "TMPVARCV"; // p1 使用 TMPVARCV 类型  // op1 是 CV 并且支持 TMPVARCV 类型  } else if ($op1 == 'CV' && isset($opcode["op1"]["TMPVARCV"])) {  $op1 = "TMPVARCV"; // p1 使用 TMPVARCV 类型  } else if (isset($opcode["op1"]["ANY"])) { // op1 支持 ANY类型  $op1 = "ANY"; // p1 使用 ANY 类型  } else if ($spec) { // 如果还是没匹配到，但是有 $spec  // 无法处理的code 分发到无效处理器  return "ZEND\_NULL";  }  }    if (!isset($opcode["op2"][$op2])) { **// 第二个操作对象的类型适配**  if (($op2 == 'TMP' || $op2 == 'VAR') && isset($opcode["op2"]["TMPVAR"])) {  $op2 = "TMPVAR";  } else if (($op2 == 'TMP' || $op2 == 'VAR') &&  isset($opcode["op2"]["TMPVARCV"])) {  $op2 = "TMPVARCV";  } else if ($op2 == 'CV' && isset($opcode["op2"]["TMPVARCV"])) {  $op2 = "TMPVARCV";  } else if (isset($opcode["op2"]["ANY"])) {  $op2 = "ANY";  } else if ($spec) {  return "ZEND\_NULL";  }  }  if (isset($extra\_spec, $opcode["spec"])) { // 如果有额外规则 ，并且此操作码有规则限制  // 按额外规则 ，生成后缀。 获取两个规则表里交叉的  $extra = extra\_spec\_name(array\_intersect\_key($extra\_spec, $opcode["spec"]));  }  }  // 操作码名 + 规则 + op1前缀 + op2前缀 + 扩展规则  return $name . ($spec ? "\_SPEC" : "") . $prefix[$op1] . $prefix[$op2] . $extra;  } |

### 3、ZEND\_VM\_DISPATCH\_TO\_HELPER()伪函数

匹配正则表达式"/ZEND\_VM\_DISPATCH\_TO\_HELPER\(\s\*([A-Za-z\_]\*)\s\*(,[^)]\*)?\)/m"，处理逻辑如下：

|  |
| --- |
| **// 紧接上一块代码**  } else { // 处理ZEND\_VM\_DISPATCH\_TO\_HELPER()伪函数  if (isset($matches[2])) { // 如果有第二个参数  // 第二个参数是从第2到N个参数，这里跳过1个参数，取后面2个参数  $args = substr(preg\_replace("/,\s\*[A-Za-z0-9\_]\*\s\*,\s\*([^,)\s]\*)\s\*/", ", $1", $matches[2]), 2);  // 所有传递的op1和op2都是两个操作对象的类型  $t = "ZEND\_VM\_TAIL\_CALL(" . helper\_name($matches[1], $spec, $op1, $op2, $extra\_spec) . "(" . $args. " ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU\_CC))";  return $t;  }  // 所有传递的op1和op2都是两个操作对象的类型  return "ZEND\_VM\_TAIL\_CALL(" . helper\_name($matches[1], $spec, $op1, $op2, $extra\_spec) . "(ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU))";  return $t;  } |

如上所示，正则表达式在匹配时会获取两段内容，第一段是调用的助手名，第二段是后面的参数表，按匹配结果是否有第二段内容，分为两种情况处理：

**1）如果匹配结果有第二段内容**，例如：

|  |
| --- |
| ZEND\_VM\_DISPATCH\_TO\_HELPER(zend\_cannot\_pass\_by\_ref\_helper, \_arg\_num, arg\_num, \_arg, arg); |

这时匹配到的第二段内容是“ \_arg\_num, arg\_num, \_arg, arg”这一段。然后使用正则表达式过虑掉其中的第1，3，5...个参数，保留第2，4，6...个参数，去掉前两个字符（逗号和空格）。把处理好的字串添加到C语言调用语句中。替换好的C语言语句如下：

|  |
| --- |
| ZEND\_VM\_TAIL\_CALL(zend\_cannot\_pass\_by\_ref\_helper\_SPEC(arg\_num, arg  ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU\_CC)); |

**2）如果匹配结果没有第二段内容**，例如：

|  |
| --- |
| ZEND\_VM\_DISPATCH\_TO\_HELPER(zend\_missing\_arg\_helper); |

替换好的C语言语句如下：

|  |
| --- |
| ZEND\_VM\_TAIL\_CALL(zend\_missing\_arg\_helper\_SPEC(ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU)); |

以上两种情况中都需要调用helper\_name()函数生成助手名，helper\_name()函数的业务逻辑与opcode\_name()函数类似。

# 八、gen\_labels()函数

gen\_labels()函数用于生成处理器相关的标签列表，这个函数会被调用2次，分别在gen\_executor()函数中的INTERNAL\_LABELS和EXTERNAL\_LABELS标记处理过程中，代码如下：

|  |
| --- |
| ...  case "INTERNAL\_LABELS":  if ($kind == ZEND\_VM\_KIND\_GOTO || $kind == ZEND\_VM\_KIND\_HYBRID) {  ...  **// 需要注意ZEND\_VM\_KIND\_HYBRID模式时传入的是ZEND\_VM\_KIND\_GOTO模式**  **gen\_labels($f, $spec, ($kind == ZEND\_VM\_KIND\_HYBRID) ? ZEND\_VM\_KIND\_GOTO : $kind, $prolog."\t\t", $specs);**  ...  case "EXTERNAL\_LABELS":  $prolog = $m[1];  if ($kind == ZEND\_VM\_KIND\_GOTO) { // ZEND\_VM\_KIND\_GOTO 模式，没用到  // 其他运行模式 ...  } else { **// 如果运行时不是 ZEND\_VM\_KIND\_GOTO 模式，用到**  // 定义 labels  out($f,$prolog."static const void \* const labels[] = {\n");  **// $specs 是在这里引用返回的, 这里生成430行代码**  **gen\_labels($f, $spec, ($kind == ZEND\_VM\_KIND\_HYBRID) ?**  **ZEND\_VM\_KIND\_CALL : $kind, $prolog."\t", $specs, $switch\_labels);**  ... |

如上所示，关键业务逻辑可归纳如下：

1）在处理INTERNAL\_LABELS标记时，使用ZEND\_VM\_KIND\_GOTO模式调用gen\_labels()函数，用于生成execute\_ex()函数中的标签；

2）在处理EXTERNAL\_LABELS标记时，使用ZEND\_VM\_KIND\_CALL模式调用gen\_labels()函数，用于生成zend\_vm\_init()函数。

## 一）第一步

gen\_labels()函数定义如下：

|  |
| --- |
| **// gen\_labels()函数第1块代码**  // 生成 操作码处理器 数组（专用|非专用）：gen\_executor 调用  function gen\_labels($f, $spec, $kind, $prolog, &$specs, $switch\_labels = array()) {  global $opcodes, $opnames, $op\_types, $prefix, $op\_types\_ex;  $list = [];  $next = 0;  $label = 0;  if ($spec) { // 这里默认是1  foreach ($opcodes as $num => $dsc) { // 遍历 操作码编号索引的 处理器列表 |

如上所示，函数90%的代码都在这个foreach循环中，用来处理数据，下面展开介绍。

### 1、组装$specs数组

循环中最先做的事情是组装$specs数组，这个数组使用引用返回，供外部使用。代码如下：

|  |
| --- |
| **// gen\_labels()函数第2块代码**  if (isset($dsc['alias'])) { // 如果处理器是别名  $specs[$num] = $specs[$opnames[$dsc['alias']]]; // 在规则表中，使用原名称的配置  continue; // 跳过其他操作  }  // $num是操作码号，$label是这个操作码第一个处理器的序号，转成字串，序号在后面$generate闭包中更新  $specs[$num] = "$label";  $spec\_op1 = $spec\_op2 = $spec\_extra = false; // 先清空操作对象规则和额外规则  $def\_op1\_type = $def\_op2\_type = "ANY"; // 两个操作对象类型都是 ANY  $next = $num + 1; // 下一个编号  if (isset($dsc["op1"]) && !isset($dsc["op1"]["ANY"])) { // 如果op1有类型限制 ，并且不是ANY  $count = 0;  foreach ($op\_types\_ex as $t) { // 遍历所有可能的类型  if (isset($dsc["op1"][$t])) { // 如果p1支持此类型  $def\_op1\_type = $t; // 更新类型名，多个的话这里是最后一个  $count++; // 每找到一个支持的类型，计数+1  }  }  if ($count > 1) { // 如果op1支持1个以上的类型  $spec\_op1 = true; // 有专用规则  $specs[$num] .= " | SPEC\_RULE\_OP1"; // 添加专用规则标记  $def\_op1\_type = "ANY"; // 类型为 ANY  }  }  if (isset($dsc["op2"]) && !isset($dsc["op2"]["ANY"])) { // 第二个操作对象，不是any  **// 处理逻辑与op1相同 ...**  }  // 把扩展标记的迪卡尔集，按键名合并到一起  $spec\_extra = call\_user\_func\_array("array\_merge", extra\_spec\_handler($dsc) ?: array(array()));  $flags = extra\_spec\_flags($spec\_extra); // 根据$extra\_spec，返回额外专用标记列表  if ($flags) { // 规则里添加额外专用标记  $specs[$num] .= " | " . implode(" | ", $flags);  }  if ($num >= 256) { // 如果序号大于256，说明是专用处理器  $opcodes[$num]['spec\_code'] = $specs[$num]; // 把规则 记录进来  unset($specs[$num]); // 删除这个规则, $specs中只保留普通操作码的规则  } |

如上所示，业务逻辑可归纳为如下几步：

1）$specs数组中添加新元素，键名是当前操作码编号，值为当前操作码的第一个处理器在lables中的序号，这个序号在调用$generate闭包时更新；

2）如果op1支持1个以上ANY以外的有效类型，添加SPEC\_RULE\_OP1标记；

3）如果op2支持1个以上ANY以外的有效类型，添加SPEC\_RULE\_OP2标记；

4）如果有额外的专用规则，添加对应的规则标记。这一步有3个子步骤：

a）调用extra\_spec\_handler()函数获取此操作码的所有扩展规则与可选值的集合（详情可参见extra\_spec\_handler()函数的介绍），再调用array\_merge函数把它合并成一维数组，例如：

|  |
| --- |
| 扩展专用规则组合： {"RETVAL":[0,1],"OBSERVER":[0,1]}  extra\_spec\_handler()函数的处理结果：  [  {"RETVAL":0,"OBSERVER":0},  {"RETVAL":1,"OBSERVER":0},  {"RETVAL":0,"OBSERVER":1},  {"RETVAL":1,"OBSERVER":1}  ]  array\_merge()函数的处理结果：  {"RETVAL":1,"OBSERVER":1} |

b）调用extra\_spec\_flags()函数获取这些规则在C语言中对应的常量；

c）把这几个常量用竖线（|）连接起来，拼接到$specs对应的元素后面。

当整个遍历完成后，$specs数组的内容如下：

|  |
| --- |
| {  "0":"0",  "1":"1 | SPEC\_RULE\_OP1 | SPEC\_RULE\_OP2",  "2":"26 | SPEC\_RULE\_OP1 | SPEC\_RULE\_OP2",  "3":"51 | SPEC\_RULE\_OP1 | SPEC\_RULE\_OP2 | SPEC\_RULE\_COMMUTATIVE",  ...  "18":"386 | SPEC\_RULE\_OP1 | SPEC\_RULE\_OP2 | SPEC\_RULE\_SMART\_BRANCH | SPEC\_RULE\_COMMUTATIVE",  "19":"461 | SPEC\_RULE\_OP1 | SPEC\_RULE\_OP2 | SPEC\_RULE\_SMART\_BRANCH | SPEC\_RULE\_COMMUTATIVE",  ...  "25":"986 | SPEC\_RULE\_OP\_DATA",  ...  "37":"1248 | SPEC\_RULE\_OP1",  "38":"1253",  ...  "294":"3450"  } |

如上所示，有的操作码没有任何规则标记，有的有1个标记，最多的有4个标记，18、19号操作码有4个标记。标记前面的数字是$label值，它在后面的操作中更新，下文中介绍。

#### **extra\_spec\_flags()函数**

extra\_spec\_flags()函数会按照$extra\_spec中有的扩展规则，返回规则对应的C语言常量，代码如下：

|  |
| --- |
| // 根据$extra\_spec，返回额外专用标记列表：gen\_labels 调用  function extra\_spec\_flags($extra\_spec) {  $s = array();  if (isset($extra\_spec["OP\_DATA"])) { $s[] = "SPEC\_RULE\_OP\_DATA"; }  if (isset($extra\_spec["RETVAL"])) { $s[] = "SPEC\_RULE\_RETVAL"; }  if (isset($extra\_spec["QUICK\_ARG"])) { $s[] = "SPEC\_RULE\_QUICK\_ARG"; }  if (isset($extra\_spec["SMART\_BRANCH"])) { $s[] = "SPEC\_RULE\_SMART\_BRANCH"; }  if (isset($extra\_spec["COMMUTATIVE"])) { $s[] = "SPEC\_RULE\_COMMUTATIVE"; }  if (isset($extra\_spec["ISSET"])) { $s[] = "SPEC\_RULE\_ISSET"; }  if (isset($extra\_spec["OBSERVER"])) { $s[] = "SPEC\_RULE\_OBSERVER"; }  return $s;  } |

### 2、遍历op1和op2可用类型的闭包函数

代码如下：

|  |
| --- |
| **// gen\_labels()函数第3块代码**  // 闭包函数：调用后返回一个遍历op1可用类型的闭包函数  $foreach\_op1 = function($do) use ($dsc, $op\_types) {  return function($\_, $op2) use ($do, $dsc, $op\_types) { // 返回一个闭包  // 遍历6个有效类型，不直接遍历$dsc["op1"]是因为里面有特殊标记  foreach ($op\_types as $op1) {  // 如果不是ANY， 那就是 "CONST", "TMP", "VAR", "UNUSED", "CV" 之一  if ($op1 != "ANY") {  if (!isset($dsc["op1"][$op1])) { // 如果op1不支持在这个类型  if ($op1 == "TMP" || $op1 == "VAR") { // 如果类型是 TMP 或 VAR  if (isset($dsc["op1"]["TMPVAR"])) { // 如果支持 TMPVAR 类型  $op1 = "TMPVAR"; // 使用 TMPVAR  } else if (isset($dsc["op1"]["TMPVARCV"])) { // 如果支持TMPVARCV类型  $op1 = "TMPVARCV"; // 使用 TMPVARCV 类型  } else { // 其他情况  $op1 = "ANY"; // 使用 ANY  }  // 如果op1类型是CV 并且支持 TMPVARCV 类型  } else if ($op1 == "CV" && isset($dsc["op1"]["TMPVARCV"])) {  $op1 = "TMPVARCV"; // 返回 TMPVARCV 类型  } else { // 其他情况  $op1 = "ANY"; // 使用 ANY  }  }  $do($op1, $op2);  }  // 不处理ANY  }  };  };  // 闭包函数：调用后返回一个遍历op2可用类型的闭包函数  $foreach\_op2 = function($do) use ($dsc, $op\_types) {  return function($op1, $\_) use ($do, $dsc, $op\_types) {  // 内部业务逻辑与 $foreach\_op1 相似 ...  };  }; |

如上所示，$foreach\_op1闭包被调用时会创建一个新闭包函数，业务逻辑归纳如下：

遍历 CONST，TMP，VAR，UNUSED，CV这5种类型：

1）如果不支持此类型，先确定备用类型：

不支持TMP时，依次尝试使用TMPVAR，TMPVARCV如果两个都不支持，使用ANY；

不支持VAR时，依次尝试使用TMPVAR，TMPVARCV如果两个都不支持，使用ANY；

不支持CV时，尝试使用TMPVARCV如果不支持，使用ANY；

不支持CONST类型时，使用ANY；

不支持UNUSED时，使用ANY作为备用类型。

2）使用确定好的类型作为参数，调用$do函数。

$foreach\_op2闭包的业务逻辑与$foreach\_op1类似，只是处理第二个操作对象的可用类型。

#### **$op\_types全局变量**

$op\_types中存放了6种有效类型，代码如下：

|  |
| --- |
| $op\_types = array( "ANY", "CONST", "TMP", "VAR", "UNUSED", "CV",); |

如上所示，这里存放6种初始型，TMPVAR和TMPVARCV两个是组合类型。

### 3、遍历op\_data可用类型的闭包函数

代码如下：

|  |
| --- |
| **// gen\_labels()函数第4块代码**  // 闭包函数：调用后返回一个遍历op\_data可用类型的闭包函数，可传入$extra\_spec  $foreach\_op\_data = function($do) use ($dsc, $op\_types) {  return function($op1, $op2, $extra\_spec = array()) use ($do, $dsc, $op\_types) {  // 遍历6个有效类型，不直接遍历$dsc["spec"]["OP\_DATA"]是因为里面有特殊标记  foreach ($op\_types as $op\_data) {  // 如果不是ANY， 那就是 "CONST", "TMP", "VAR", "UNUSED", "CV" 之一  if ($op\_data != "ANY") {  if (!isset($dsc["spec"]["OP\_DATA"][$op\_data])) { // 如果不支持此类型  if ($op\_data == "TMP" || $op\_data == "VAR") { // 如果类型是 TMP 或 VAR  if (isset($dsc["spec"]["OP\_DATA"]["TMPVAR"])) { // 如果支持 TMPVAR  $op\_data = "TMPVAR"; // 类型为 TMPVAR  // 否则如果支持 TMPVARCV  } else if (isset($dsc["spec"]["OP\_DATA"]["TMPVARCV"])) {  $op\_data = "TMPVARCV"; // 类型为 TMPVARCV  } else { // 其他情况  $op\_data = "ANY"; // 类型为 ANY  }  // 如果类型是 CV 并且支持 TMPVARCV  } else if ($op\_data == "CV" && isset($dsc["OP\_DATA"]["TMPVARCV"])) {  $op\_data = "TMPVARCV"; // 类型为 TMPVARCV  } else { // 其他情况  $op\_data = "ANY"; // 类型为 ANY  }  }  $do($op1, $op2, array("OP\_DATA" => $op\_data) + $extra\_spec);  }  }  };  }; |

$foreach\_op\_data闭包的业务逻辑与$foreach\_op1类似，只是处理操作数据的可用类型。

### 4、遍历专用规则的闭包函数

代码如下：

|  |
| --- |
| **// gen\_labels()函数第5块代码**  // 闭包函数：调用后返回一个遍历op\_data可用类型的闭包函数，可传入$extra\_spec  $foreach\_extra\_spec = function($do, $spec) use ($dsc) {  // 所有传递的op1和op2都是两个操作对象的类型  return function($op1, $op2, $extra\_spec = array()) use ($do, $spec, $dsc) {  foreach ($dsc["spec"][$spec] as $val) { // 遍历所有规则  // 所有传递的op1和op2都是两个操作对象的类型  // 把额外规则附加到 规则 后面一块处理。相同索引号后面不会覆盖前面的  $do($op1, $op2, array($spec => $val) + $extra\_spec);  }  };  }; |

如上所示，$foreach\_op1闭包被调用时会创建一个新闭包函数，业务逻辑归纳如下：

遍历此操作码支持的额外专用规则：为每个规则调用$do函数。

### 5、创建生成器$generate

代码如下：

|  |
| --- |
| $generate = function ($op1, $op2, $extra\_spec = array()) use ($f, $kind, $dsc, $prefix, $prolog, $num, $switch\_labels, &$label, &$list) {  global $commutative\_order;  // 检验是否定义了专用处理器  // 如果op1 op2都支持，并且（没有OP\_DATA的类型要求 或 有匹配的OP\_DATA类型）  if (isset($dsc["op1"][$op1]) && isset($dsc["op2"][$op2]) &&  (!isset($extra\_spec["OP\_DATA"]) ||  isset($dsc["spec"]["OP\_DATA"][$extra\_spec["OP\_DATA"]]))) {  if (skip\_extra\_spec\_function($op1, $op2, $extra\_spec)) { // 如果需要跳过此额外规则  gen\_null\_label($f, $kind, $prolog); // 创建 null label  $list[$label] = null; // 值为null 的第1/2种情况  $label++; // 数量增加  return;  }  if (isset($extra\_spec["SMART\_BRANCH"])) { // 有智能分枝标记  // HOT不允许双常量，或COLD允许双常量  if ($dsc["hot"] === 'HOT\_NOCONSTCONST\_'  || $dsc["hot"] === 'COLD\_CONSTCONST\_') {  if (($op1 === 'CONST') && ($op2 === 'CONST')) { // 并且碰到双常量  unset($extra\_spec["SMART\_BRANCH"]); // 删除智能分支  }  }  }    // 为有名字的参数跳过 QUICK\_ARG 规则  if (isset($extra\_spec["QUICK\_ARG"])) { // 如果有 QUICK\_ARG  if ($op2 === "CONST") { // 第二个操作对象类型是常量（变量名）  unset($extra\_spec["QUICK\_ARG"]); // 删除 QUICK\_ARG  }  }  // 为观察者处理器 跳过所有规则  if (isset($extra\_spec["OBSERVER"]) && $extra\_spec["OBSERVER"] == 1) {  if (isset($extra\_spec["RETVAL"])) { // 如果有 RETVAL 规则  unset($extra\_spec["RETVAL"]); // 删除它  }  if ($op1 != "ANY" || $op2 != "ANY") { // 如果两个操作对象有一个不是ANY  $op1 = "ANY"; // 两个都换成ANY  $op2 = "ANY";  }  }    // 按额外规则 ，生成后缀  $spec\_name = $dsc["op"]."\_SPEC".$prefix[$op1].$prefix[$op2].  extra\_spec\_name($extra\_spec);    switch ($kind) { // 按模式处理  case ZEND\_VM\_KIND\_CALL: // call模式，用于生成zend\_vm\_init()函数中的标签  out($f,"$prolog{$spec\_name}\_HANDLER,\n");  break;  // 其他运行模式 ...  case ZEND\_VM\_KIND\_GOTO: // goto模式，用于生成execute\_ex()函数中的标签  out($f,$prolog."(void\*)&&{$spec\_name}\_LABEL,\n");  break;  // HYBRID模式不生成任何东西  }  $list[$label] = $spec\_name; // 值为规则名  $label++;  } else { // 其他情况  gen\_null\_label($f, $kind, $prolog); // 指针指向未知操作码  $list[$label] = null; // 值为null 的第2/2种情况  $label++; // 数量+1  }  }; |

如上所示，业务逻辑可归纳如下：

一、如果op1和op2都有指定的类型，并且（扩展专用规则中没有指定OP\_DATA的类型 或 有指定OP\_DATA的规则，并且类型匹配正确）：

第1步，调用skip\_extra\_spec\_function()函数检查是否要跳过，如果要跳过，创建空标签，continue跳到下一个；

第2步，如果此处理器是不允许双常量的热处理器或者是允许双常量的冷处理器，并且op1和op2都是常量：不允许使用智能分支，删除智能分支规则；

第3步，参数有名字的时（op2是常量类型的变量名），跳过QUICK\_ARG规则；

第4步，如果开启了观察者，要删除RETVAL规则；只要op1和op2有一个是ANY类型，就要把两个都改成ANY；

第5步，生成名称：操作码名 + "\_SPEC" + op1类型 + op2类型 + 专用规则后缀；

第6步，ZEND\_VM\_KIND\_CALL模式下，把处理器名写入文件；ZEND\_VM\_KIND\_GOTO模式下，把标签名写入文件；

第7步，把第5步中生成的名称添加到$list数组中，键名为标签序号；然后把标签数量+1。

二、如果op1和op2有一个没有指定类型：调用gen\_null\_label()函数生成无效标签。

三、每种情况业务逻辑结束前，都会调用$label++，更新已有处理器序号（也是数量）,$label会在生成$specs数组时用到。

skip\_extra\_spec\_function()函数、extra\_spec\_name()函数和$prefix全局变量已经在“gen\_executor\_code()函数”章节中介绍过。

#### **gen\_null\_label()函数**

gen\_null\_label()函数用于生成空的处理器名或标签，代码如下：

|  |
| --- |
| function gen\_null\_label($f, $kind, $prolog) {  switch ($kind) {  case ZEND\_VM\_KIND\_CALL: // CALL 模式  out($f,$prolog."ZEND\_NULL\_HANDLER,\n"); // 空处理器  break;  case ZEND\_VM\_KIND\_GOTO: // GOTO 模式  out($f,$prolog."(void\*)&&ZEND\_NULL\_LABEL,\n"); // 空标签  break;  // 其他运行模式，忽略 ...  }  } |

### 6、执行生成器

代码如下：

|  |
| --- |
| // 最后调用的是 $generate  $do = $generate;  if ($spec\_extra) { // 如果有额外的专用规则  foreach ($spec\_extra as $extra => $devnull) { // 遍历此操作码的额外专用规则，只用键名  if ($extra == "OP\_DATA") { // 有OP\_DATA时没有其他规则  $do = $foreach\_op\_data($do); // 给生成器添加OP\_DATA规则  } else { // 有其他额外专用规则时，没有OP\_DATA  $do = $foreach\_extra\_spec($do, $extra); // 给生成器添加其他规则  }  }  }  if ($spec\_op2) { // 如果有操作码2的规则  $do = $foreach\_op2($do); // 给生成器添加op2的规则  }    if ($spec\_op1) { // 如果有操作码1的规则  $do = $foreach\_op1($do); // 给生成器添加op1的规则，倒着加是为了正着运行  }  $do($def\_op1\_type, $def\_op2\_type); // 运行生成器  }} // foreach结束，if结束 |

如上所示，调用闭包时先包装特殊规则，再包装op2的装饰，最后包装op1的闭包。这是因为在运行闭包时，会从外向里运行，op1的闭包在最外层，op2的在里层，扩展专用规则和操作数据的闭包在最里层。

每一层都遍历了可用的类型，这样分几层调用，就遍历了所有可用的组合，其中有些组合使用了备用类型，所以类型组合会有重复。

## 二）第二步

代码如下：

|  |
| --- |
| switch ($kind) {  case ZEND\_VM\_KIND\_CALL: // call模式  out($f,$prolog."ZEND\_NULL\_HANDLER\n");  break;  // 其他运行模式 ...  case ZEND\_VM\_KIND\_GOTO: // godo模式  out($f,$prolog."(void\*)&&ZEND\_NULL\_LABEL\n");  break;  }  //  $specs[$num + 1] = "$label";  // 写入 zend\_vm\_handlers.h  $l = fopen(\_\_DIR\_\_ . "/zend\_vm\_handlers.h", "w+") or die("ERROR: Cannot create zend\_vm\_handlers.h\n");  out($l, "#define VM\_HANDLERS(\_) \\\n"); // 先写宏名称  foreach ($list as $n => $name) { // 遍历所有，序号=>名字，这里面有大量重复的  if (null !== $name) { // 跳过名称为null的（op1或op2有一个为ANY类型）  out($l, "\t\_($n, $name) \\\n");  }  }    out($l, "\t\_($n+1, ZEND\_NULL)\n"); // 最后加一个null的  fclose($l);  } |

如上所示，业务逻辑可归纳为如下步骤：

1）添加一个空的标签或处理器；

2）生成zend\_vm\_handlers.h文件。

# 附录

## 一）未介绍的函数和常量

### is\_hot\_helper()

gen\_code()函数，ZEND\_VM\_KIND\_HYBRID模式用到，代码如下：

|  |
| --- |
| // 是否是“热”助手( 有hot 标记 )：gen\_code 调用  function is\_hot\_helper($name) {  global $helpers;  if (isset($helpers[$name]["hot"])) {  return $helpers[$name]["hot"];  }  return false;  } |

### is\_inline\_hybrid\_handler()

gen\_handler()函数，ZEND\_VM\_KIND\_HYBRID模式用到，代码如下：

|  |
| --- |
| // 所有传递的op1和op2都是两个操作对象的类型  // clear, 是否是行内混合处理器（$hot === 'INLINE\_'）：gen\_handler。要这么多参数做什么，又不用？  function is\_inline\_hybrid\_handler($name, $hot, $op1, $op2, $extra\_spec) {  return ($hot === 'INLINE\_');  } |

# Change Log

2025.6.27 创建。

2025.7.8 这一篇是辅助内容，本来没打算花太多时间认真写，没想到也越写越有味道~。今天居然用到了5级标题，第一次。gen\_code()，很多环节都打通了，后面还有很多内容要慢慢理解，这个非常重要。

2025.7.9 把所有变量访问又梳理一遍，基本OK了。

2025.7.10 整理变量访问部分。 需要加一章操作码的寻址方式，还要加一章各类型的解释，内容越来越多了，认知负荷好大，再写一阵看有没有必要拆成两篇。

2025.7.11 各种变量类型到底有什么区别，主要是TMPVARCV。还有函数次序是如何生成的，这两块重点内容。gen\_code()函数完毕。gen\_labels()也浮出水面了。

2025.7.12 gen\_labels()理清楚就不觉得复杂了。 整理完成七七八八了，再好好梳理一下。

2025.7.15 什么是冷热，很多相关概念。

2025.7.16 冷热，附加规则已基本弄清楚，常量也基本弄清楚。

2025.7.17 label,specs基本都OK了。留到最后的问题是冷和热，根据相关常量定义，冷和热更像是个细节优化，暂时放放。本篇基本OK，本来想调一下格式把前4章合成1章，有空再弄，前4章一共30页，分不分其实也都可以。

2025.7.18 zend\_vm\_set\_opcode\_handler\_ex()对应的是特殊处理器的条件优化，关闭优化器时用不到它。